Chapter 1: Concept of data Structures

Introduction:

Data Types:

- A data type in programming, is a classification that specifies which type of value a variable has and what type of mathematical, relational or logical operations can be applied to it.
- A data type is the collection of values and a set of operations on the values.
- For example:
  - A string is a data type that is used to classify text.
  - An integer is a data type that is used to classify whole numbers.

Data types	Examples
Integer	1,3,15,67...
String	Hello world, ram, sita...
Float	3.14, 6.78...

Data Structure:

- A data structure is a systematic way of organizing data in a computer so that it can be used effectively.
- Data structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way.
- Data structure is about rendering data elements in terms of some relationship, for better organization and storage.
- Data structure is classified into the following two categories
  - **Primitive Data Structure:** They are basic structure and are directly operated by machine instructions.
  - **Non-Primitive Data Structure:** These are more sophisticated data structure and are derived from primitive data structure. The non-primitive data structure emphasis on structuring of group of homogeneous and heterogeneous data items.

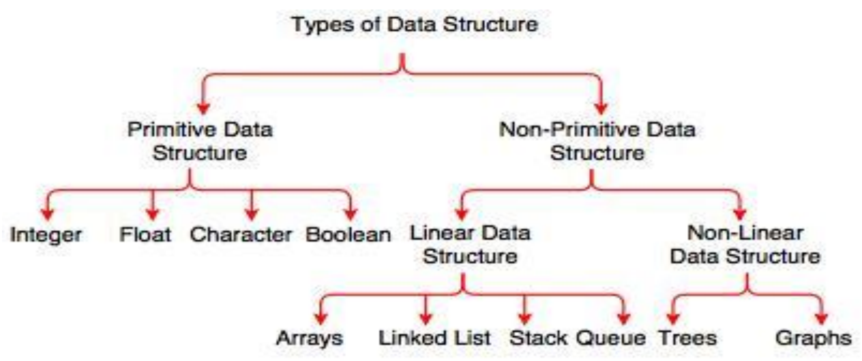
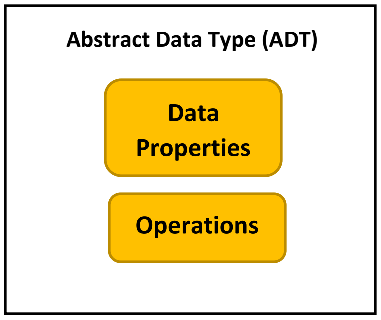


Fig. Types of Data Structure

- Data Structure = Organized Data + Allowed Operations
- Commonly used operations are Searching, Inserting, Deleting, Sorting etc.
- Example
  - An array is a data structure for storing more than one data item that has a similar data type. The items of an array are allocated at adjacent memory locations.
  - Following are the operations supported by an array.
    - Traverse: Print all the array elements one by an array.
    - Insertion: Adds an element at the given index.
    - Deletion: Deletes an element at the given index.
    - Search: Searches an element using the given index or by the value.
    - Update: Updates an element at the given index.

Abstract Data Type (ADT)

- A useful tool for specifying the logical properties of a datatype is called Abstract Data Type (ADT).
- ADT's specification describes what data can be stored (the characteristics of ADT) and how it can be used (the operations) but not how it is implemented or represented in the program.



- ADT is a type for objects whose behavior is defined by a set of values and a set of operations.
- An abstract data type is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects.
- It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.

Specification of ADT:

For illustration, consider rational number in the form –

$$\left\{\frac{p}{q} / \; p, q \in Z \text{ and } q \neq 0\right\}$$

The above equation indicates that any rational number will be in the form of p divided by q, where p and q are integers (the set Z) and the denominator q is not equal to zero.

In this case, the sum of two rational numbers (a1/a2) and (b1/b2) would be –

$$\frac{a_1}{a_2} + \frac{b_1}{b_2} = \frac{a_1 * b_2 + b_1 * a_2}{a_2 * b_2}$$

Specification of any ADT consists of two parts:

1. **value definition:**
    - Here, we specify the set of possible values taken by the ADT along with some conditions or constraints with which the ADT bounds.
    - It contains definition clause and condition clause.
2. **operator definition:**
    - Here, various operations which are imposed on ADT are defined. This part contains 3 sections viz. a header, the preconditions (which is optional) and the postconditions. The term 'abstract' in the header indicates that this is not a C function; rather it is an ADT operator definition. This term also indicates that the role of an ADT is a purely logical definition of a new data type.

The following listing gives the ADT for rational numbers. The value definition here indicates the constraints on rational number. The operator definition parts contain the definition for various operations like creation of rational number, addition and multiplication of rational numbers and for checking the equality of two rational numbers.

```
// value definition
abstract typedef<integer, integer> RATIONAL;
condition RATIONAL[1] != 0;

//Operator definition
abstract RATIONAL createrational (a, b)
int a,b;
precondition b!=0;
postcondition   createrational [0] = = a;
                         createrational [1] = = b;

abstract RATIONAL add(a,b)
RATIONAL a, b;
postcondition add[0] = =  a[0]*b[1] + b[0]*a[1];
                     add[1] = = a[1]*b[1];
abstract RATIONAL mul(a,b)
RATIONAL a, b;
postcondition mul[0] = = a[0]*b[0];
                       mul[1] = = a[1]*b[1];

abstract equal(a, b)
RATIONAL a,b;
postcondition
                 equal = = (a[0]*b[1] = = b[0]*a[1]);
```

Chapter 1: Concept of data Structures

Algorithms

An algorithm is a set of rules for carrying out calculations either by hand or machine. An algorithm is a sequence of computational steps that transform the input into the output. An algorithm performed on data that have to be organized in data structure. An algorithm is an abstraction of a program to execution on a physical machine.

Algorithm Design:

An algorithm design is a specific method to create a mathematical process in solving problem.

Steps for developing algorithm:

1. Problem definition
2. Specification of algorithm
3. Design an algorithm
4. Checking the correctness of algorithm
5. Analysis of algorithm
6. Implementation of algorithm
7. Problem Testing
8. Documentation

Algorithm Efficiency:

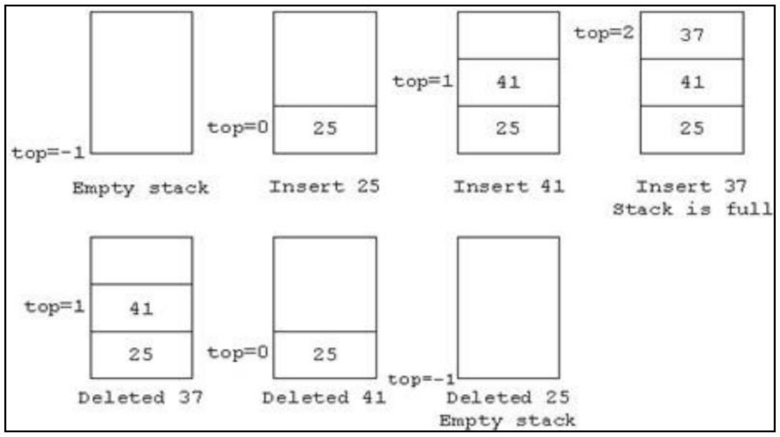
How fast is algorithm?

How much money does it cost?

Chapter 2. Stack and Queue

Stack:

- Stack is an ordered collection of data in which insertion and deletion operation is performed at only one end called the **Top of the Stack (TOS)**.
- The element last inserted will be the first to be deleted. Hence, stack is known as Last in First out (**LIFO**) structure.
- Stack Operations:
    - **Push:** It is used to add an item in the stack. If the stack is full, then it is said to be **overflow condition.**
    - **Pop:** It is used to remove an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an **Underflow condition.**
- Initial value of top is -1 i.e., **top = -1**
- Example: Stack Operations for the size of stack 3



Operations:

```
Push Operation:-
        If (top == (max-1)]
             Printf("stack over flow");
        Else
        {
        Top == top + 1;
        Stack_arr[top] = pushed –item;
        }

Pop Operation:-
        If (top ==-1)
             Printf("stack underflow");
        Else
        {
             Printf("Poped element is %d",stack_arr[top]);

                 Top=top -1;

         }
```

Algorithm for Push and Pop Operation:

Push:

```
1. Start
2. If(top=size-1)              // Check for stack overflow
        Print" Stack is full"
3. Otherwise,
     i.   Increase top by 1
     ii.  Read data and store at top position
4. Stop
```

Pop:

```
1. Start
2. If(top=-1)                  //Check for stack Underflow

           Print" Stack is empty"

     Otherwise,

           Decrease top by 1

3. Stop
```

Stack Application: Evaluation of infix, postfix and prefix expressions

- An expression is defined as number of operands or data items combined with several operators.
- Three types of notations for an expression:
    - **Infix notation:**
        - An expression where operators are used in-between operands. e.g., A+B
        - It is easy for us human to read, write, and speak in infix notation but the same does not go well with computing devices.
    - **Prefix Notation:**
        - An expression where the operator is written ahead of operands. e.g., +AB
    - **Postfix Notation:**
        - An expression where the operator is written after the operands. e.g., AB+
- Operator and their precedence level:
    - Operator precedence determines which operator is performed first in an expression with more than one operator with different precedence.

Operator	**Precedence**	**Value**	**Associativity**
Exponentiation (**$, ^**)	Highest	3	Right to left
***, /, %**	Next highest	2	Left to Right
+, -	Lowest	1	Left to Right

Note: **[{( )}]** will be evaluated first.

Algorithm to convert infix to postfix:

```
1. Let Infix be a string that stores infix expression and Postfix be a string that stores the
   postfix result.
   Scan the Infix Expression character from left to right.
   1. Start
   2. If character is operand
          Append it to postfix
   3. If character is operator
          i. If stack is empty OR stack's top is '(' OR precedence of character >
             precedence of stack's top
                 Push character to the stack
         ii. Else
                 While stack's top is operator AND precedence of stack's top >=
                 precedence of character
                         Append the operator from stack to Postfix and pop it from stack
                 Push character to the stack
   4. If character is'('
          Push character to the stack.
   5. If character is ')'
          Append the operator from stack to postfix and pop if from stack until stack's
          top is '('
          Pop '('from stack
   6. Repeat 2 to 5 until infix expression is scanned.
   7. Append the operator from stack to postfix and pop it from stack until stack is empty.
   8. stop
```

Example 1: Infix Expression: (A +B) *C

Scanned Character	Stack	Postfix
(	(	
A	(	A
+	(+	A
B	(+	AB
)		AB+
*	*	AB+
C	*	AB+C
		AB+C*

Postfix: **AB+C***

Example 2: Infix Expression: $\boldsymbol{A+B-(C*D/E+F)-G*H}$ **[2073 Bhadra]**

Scanned Character	Stack	Postfix
A		**A**
+	**+**	**A**
B	**+**	**AB**
-	**-**	**AB+**
(	**-(**	**AB+**
C	**-(**	**AB+C**
*****	**-(***	**AB+C**
D	**-(***	**AB+CD**
/	**-(/**	**AB+CD***
E	**-(/**	**AB+CD*E**
+	**-(+**	**AB+CD*E/**
F	**-(+**	**AB+CD*E/F**
)	**-**	**AB+CD*E/F+**
-	**-**	**AB+CD*E/F+-**
G	**-**	**AB+CD*E/F+-G**
*****	**-***	**AB+CD*E/F+-G**
H	**-***	**AB+CD*E/F+-GH**
		AB+CD*E/F+-GH*-

Q. $A + B - C * (D - E + F/G)/H$

Q. $A * B/C - D + (E/F * G)/(K - L)$

Algorithm to convert Infix to Prefix:

1. Reverse the infix expression
2. Obtain the "nearly" postfix expression of the modified expression (If Associativity is Left to Right Then Push)
3. Reverse the postfix expression

Example 1: Infix expression: (A +B) *C

2. Reversing the infix expression

   C * (B+A)

3. Converting modified expression to postfix expression

Scanned Character	Stack	Postfix
C		C
*	*	C
(	*(	C
B	*(	CB
+	*(+	CB
A	*(+	CBA
)	*	CBA+
		CBA+*

Postfix: **CBA+***

4. Reversing the postfix expression

   ***+ABC**

Algorithm to evaluate the postfix expression:

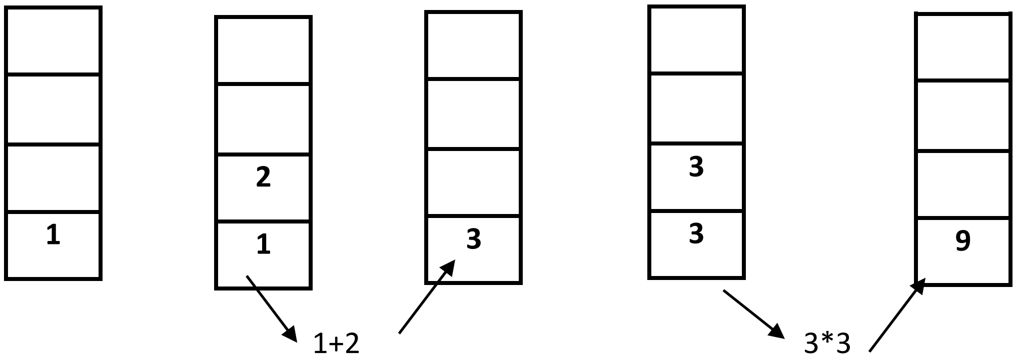
1. Start
2. If character is number

   Push on the stack

3. If character is operator(op)
   a. Val1=pop
   b. Val2=pop
   c. Perform result=val2 op val1
   d. Push the result into stack
4. Repeat 2 to 3 until postfix expression is scanned.
5. Output the result
6. Stop

Example 1: AB+C*

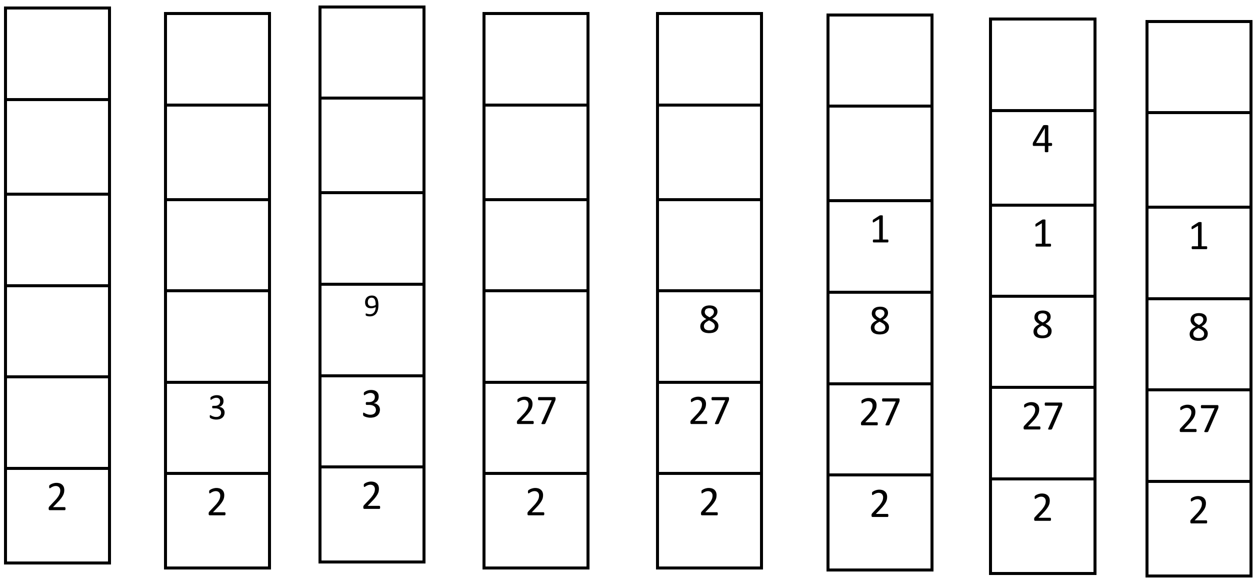
Let A=1, B=2 and C=3

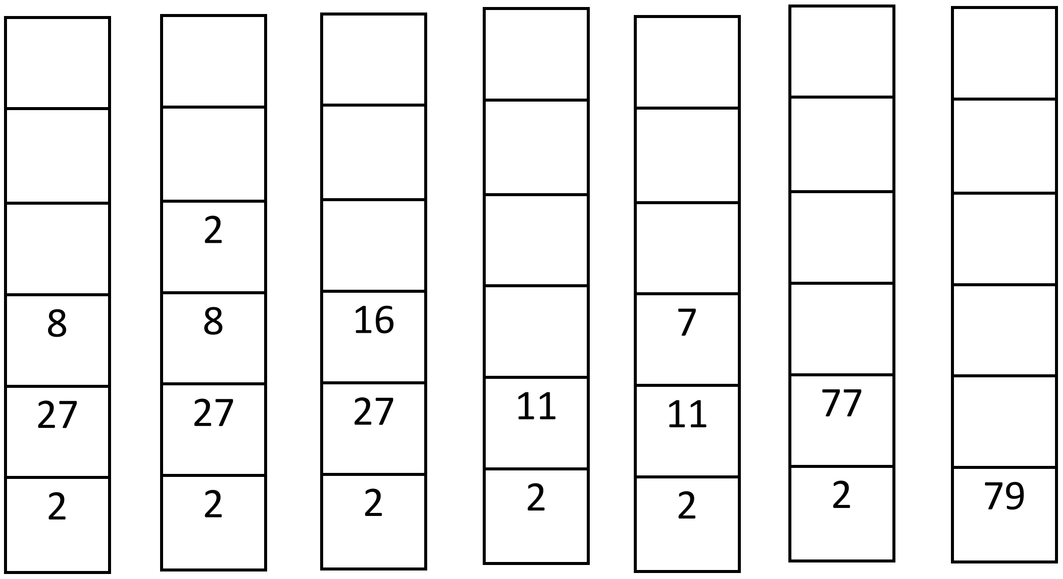


Ans: 9

Example 2: ABC*DEF^/G*-H*+

Let A= 2, B= 3, C=9, D=8, E=1, F=4, G= 2, H=7

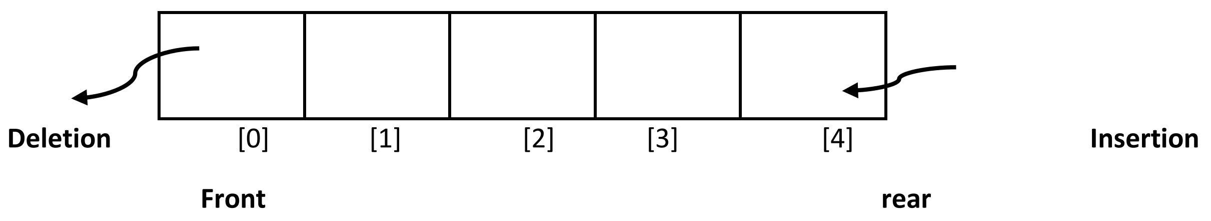




Ans: 79

Queue:

- A queue is an ordered collections of items from which items may be deleted at one end called the front of the queue and in to which items may be inserted at the other end called rear of the queue.
- It follows ***FIFO*** policy.
- **Queue operations:**
    - **Enqueue:** It adds an item to the queue. If the queue is full, then it is said to be an overflow condition.
    - **Dequeue:** It removes an item from the queue. The items are removed in the same order in which they are added. If the queue is empty, then it is said to be an Underflow condition.



Linear Queue:

- A linear queue is a linear data structure that serves the request first, which has been arrived first. It consists of data elements which are connected in a linear fashion.
- Initial condition:

  Front=0 and Rear=-1
- **Algorithm for Insertion:**

```
1. Start
2. If (rear = = MaxSize -1)
       Printf("queue overflow");
3. Else
       If (front = = -1)
           Front = 0;
       Rear = rear + 1;
       Queue [rear] =item;
4. Stop
```

- **Algorithm for Deletion:**

```
1. Start
2. If (front = = -1)|| (front >rear)

        Printf ("Queue under flow");

        Return;

3.  Else
        Printf ("Element deleted from queue is: %d", queue [front]);

        front = front +1;

4. Stop
```

Example:

REAR=-1 and FRONT=0

After 1 insertion REAR=0 and insert item 10 into queue.

REAR=0 and FRONT=0

10				

Now insert 20 into queue

REAR=1 and FRONT=0

10	*20*			

Suppose now we delete one item from queue, as we know that deletion can be done from FRONT end in queue.
Now, REAR=1 and FRONT=1

	20			

Now we insert 30, 40 and 50 into queue respectively.
REAR=2 and FRONT=1

	20	*30*		

REAR=3 and FRONT=1

	20	*30*	*40*	

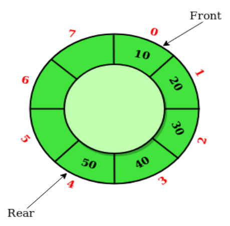
REAR=4 and FRONT=1

	20	*30*	*40*	*50*

Circular Queue:

- A circular queue is one in which the insertion of new element is done at the very first location of the queue if the last location of the queue is full.
- It connects the last position of the queue to the first position of the queue.
- Initial Condition:

    Front=-1 and rear =-1



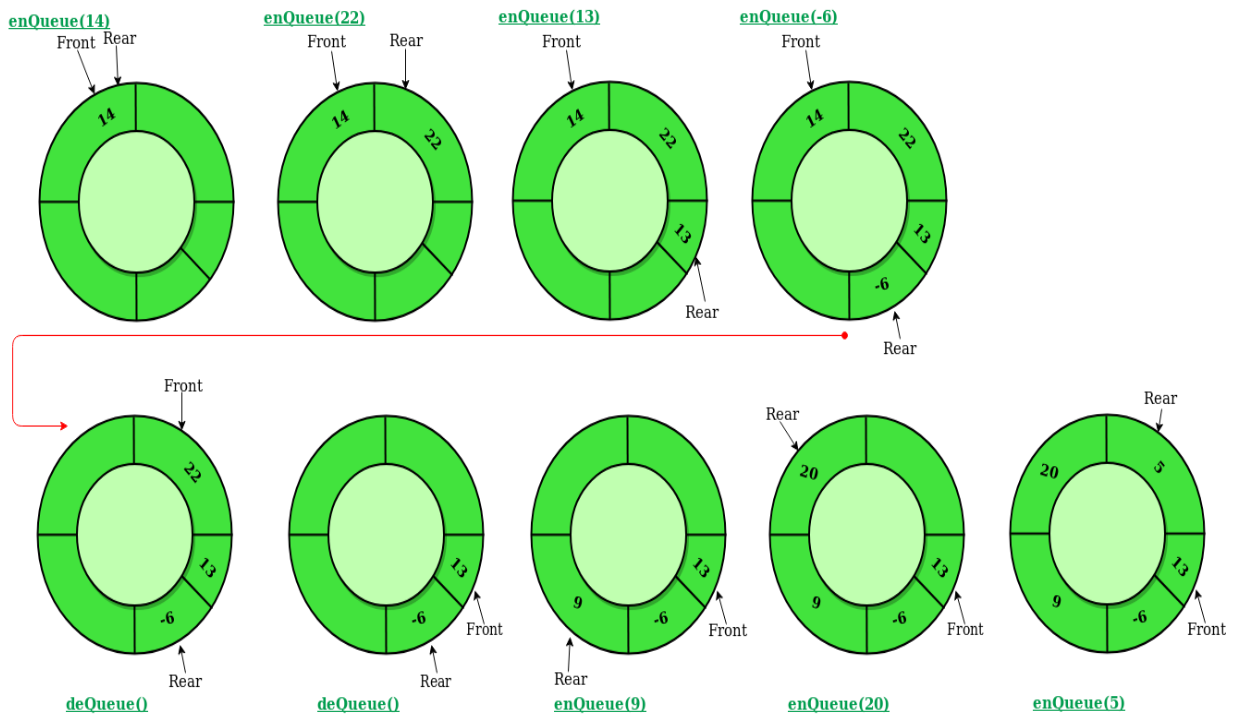
- **Algorithm for Insertion:**

```
1. Start
2. if((front == 0 && rear == MAX-1) || (front == rear+1))
        Print "Queue Overflow" and return
3. otherwise
    i.  if(front == -1)
            Set front = 0 and rear = 0
    ii. otherwise
            Increase rear as, rear = (rear + 1) % MAX
4. Insert the item rear position cqueue[rear] = item
5. Stop
```

- **Algorithm for Deletion:**

```
1. Start
2. if(front == -1)
        Print "Queue Underflow" and return
3. otherwise
4. Element deleted from queue is cqueue[front]
    i.  if(front == rear)
            Set front = -1 and rear = -1
    ii. otherwise
            Increase front as, front = (front + 1) % MAX
5. Stop
```

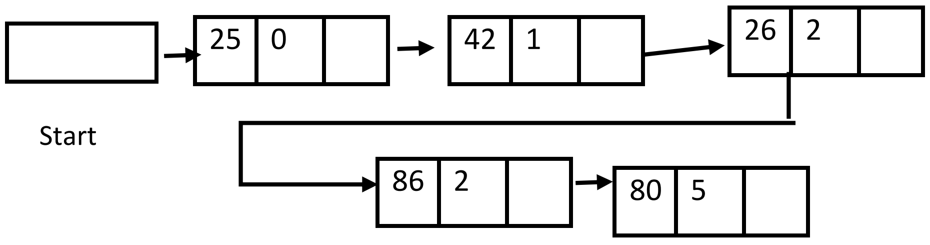
Example:



Priority Queue:

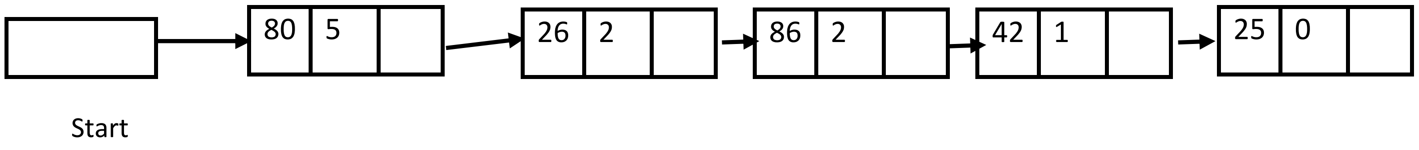
- Priority queue is an extension of queue with following properties:
  - Every item has a priority associated with it.
  - An element with higher priority is de-queued before an element with lower priority.
  - Two elements have the same priority, they are served according to their order in the queue.
- **Ascending Priority queue:**
  - An ascending priority queue is a collection of items into which items can be inserted arbitrarily and from which only the smallest items can be removed.
  - Lower priority number has higher priority.
  - For example: A queue may be viewed as ascending priority queue whose elements are ordered by the time of insertion. 0 has high priority

Item	Priority
25	0
80	5
26	2
42	1
86	2



- **Descending Priority Queue:**
    - A descending priority queue is similar but allows deletion of only the largest items.
    - Higher priority number to high priority
    - For example: A stack may be viewed as **descending priority** queue whose elements are ordered by the time of insertion. The element that was inserted last has the greatest insertion –time value and is the only that can be retrieved.

Item	Priority
25	0
80	5
26	2
42	1
86	2



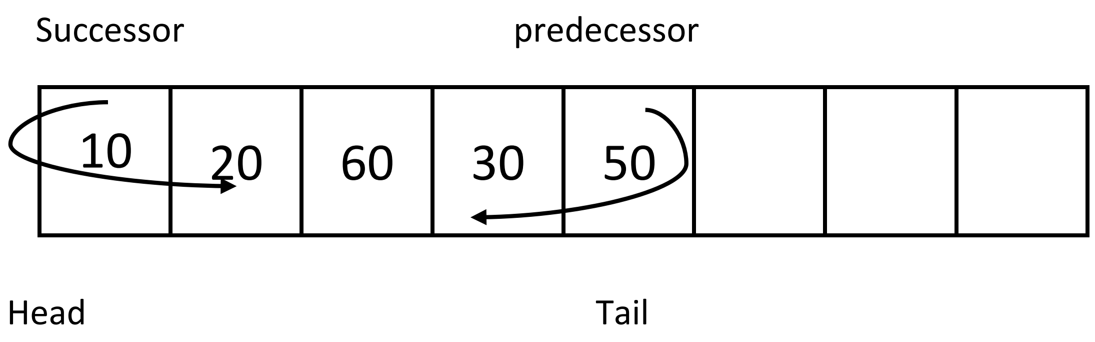
Application of Priority Queue:

- It is used in data compression techniques like Huffman code.
- Priority queues are used to select the next process to run.
- It is used in bandwidth management to prioritize the important data packet.
- Used in algorithms like Dijkstra's shortest path algorithm, heap sort algorithm, etc.

Chapter 3 List

Definition

- List means collection of elements in sequential order to which addition & deletion can be made.
- The first element of a list is called the head of list & the last is called the tail of the list.
- The next element of the head of the list is called its successor. The previous element to the tail (if it is not head of the list) is called its predecessor. Clearly a head doesn't have as predecessor & a tail doesn't have a successor. Any other element of the list has both one successor & one predecessor.

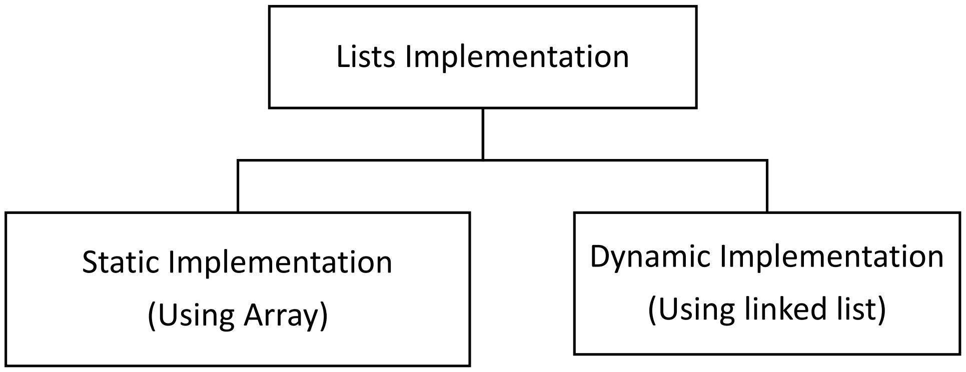


Operations perform in list:-

1. Traversing an array list
2. Searching an element in the list
3. Insertion of an element in the list
4. Deletion of an element in the list.

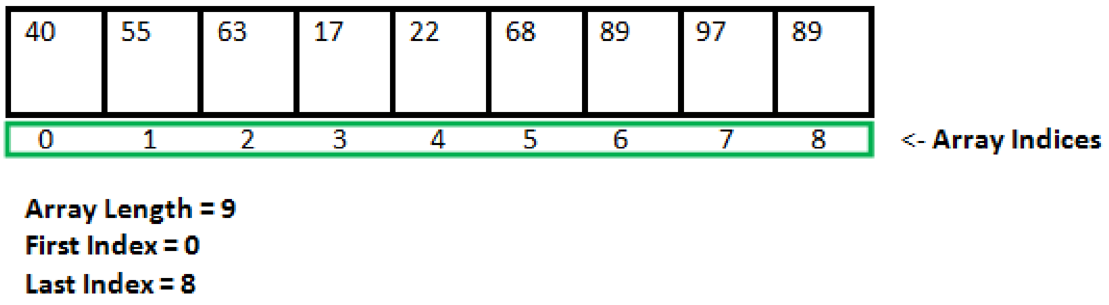
- In memory we can store the list in two ways:
- One way to store the elements at sequential memory location. This implementation is called **static implementation** and is done using array.
- The other way is we can use pointers or links to attach the elements sequentially. This is called **dynamic implementation**.

Chapter 3 List



Storing a list in a static data structure:

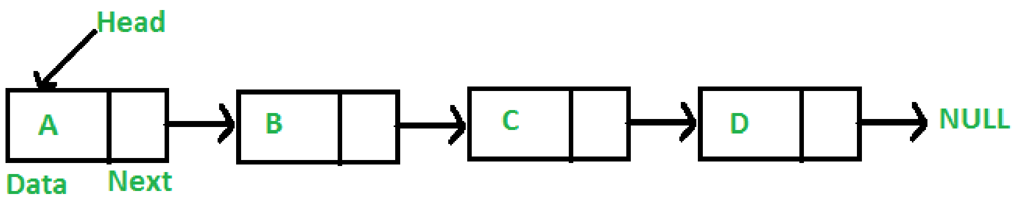
- This implementation stores the list in an array.
- The position of each element is given by an index from 0 to n-1, where n is the number of elements.
- Given any index, the element with that index can be accessed in constant time – i.e. the time to access does not depend on the size of the list.
- To add an element at the end of the list, the time taken does not depend on the size of the list. However, the time taken to add an element at any other point in the list does depend on the size of the list, as all subsequent elements must be shifted up. Additions near the start of the list take longer than additions near the middle or end.
- When an element is removed, subsequent elements must be shifted down, so removals near the start of the list take longer than removals near the middle or end.



Chapter 3 List

Storing a list in a dynamic data structure (Linked List):

- The Link List is stored as a sequence of linked nodes. As in the case of the stack and the queue, each node in a linked list contains data and a reference to the next node.
- The list can grow and shrink as needed the position of each element is given by an index from 0 to n-1, where n is the number of elements.
- Given any index, the time taken to access an element with that index depends on the index. This is because each element of the list must be traversed until the required index is found.
- The time taken to add an element at any point in the list does not depend on the size of the list, as no shifts are required. It does, however, depend on the index. Additions near the end of the list take longer than additions near the middle or start. The same applies to the time taken to remove an element.
- The first node is accessed using the name LinkedList.Head
- Its data is accessed using LinkedList.Head.DataItem

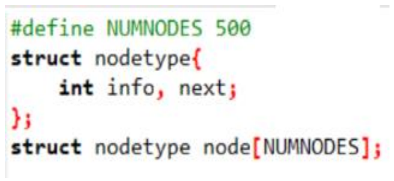


Static Data Structure vs Dynamic Data Structure:

- Static Data structure has fixed memory size whereas in Dynamic Data Structure, the size can be randomly updated during run time which may be considered efficient with respect to memory complexity of the code. Static Data Structure provides easier access to elements with respect to dynamic data structure. Unlike static data structures, dynamic data structures are flexible.

Array Implementation of List:

- This implementation stores the list in an array.



```
#define NUMNODES 500
struct nodetype{
    int info, next;
};
struct nodetype node[NUMNODES];
```

Insertion:

- Input the array elements, the position of the new element to be inserted and the new element.
- Insert the new element at that position and shift the rest of the elements to right by one position.

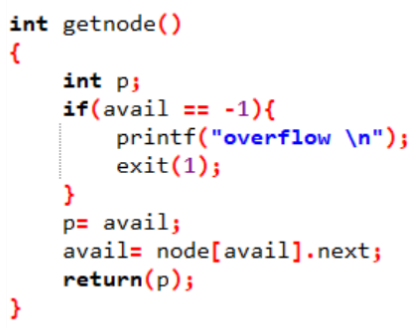
Algorithm

1. Get the **element value** which needs to be inserted.
2. Get the **position** value.
3. Check whether the position value is valid or not.
4. If it is **valid**,

   Shift all the elements from the last index to position index by 1 position to the **right**.

   insert the new element in **arr[position]**
5. Otherwise,

   Invalid Position



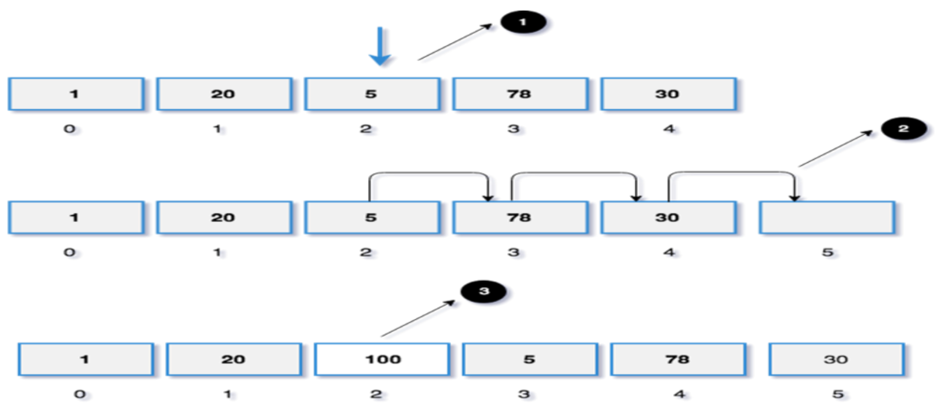
```
int getnode()
{
    int p;
    if(avail == -1){
        printf("overflow \n");
        exit(1);
    }
    p= avail;
    avail= node[avail].next;
    return(p);
}
```

Example1:

Let's take an array of 5 integers.

1, 20, 5, 78, 30.

If we need to insert an element 100 at position 2, the execution will be,

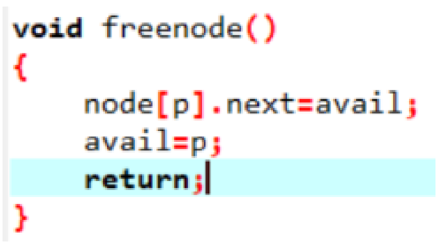


Deletion

- Input the array elements, the position of the element to be deleted and the element.
- Delete the element and shift the rest of the elements to left by one position.

Algorithm

```
1. Find the given element in the given array and note the index.
2. If the element found,
    Shift all the elements from index + 1 by 1 position to the left.
    Reduce the array size by 1.
3. Otherwise, print "Element Not Found"
```



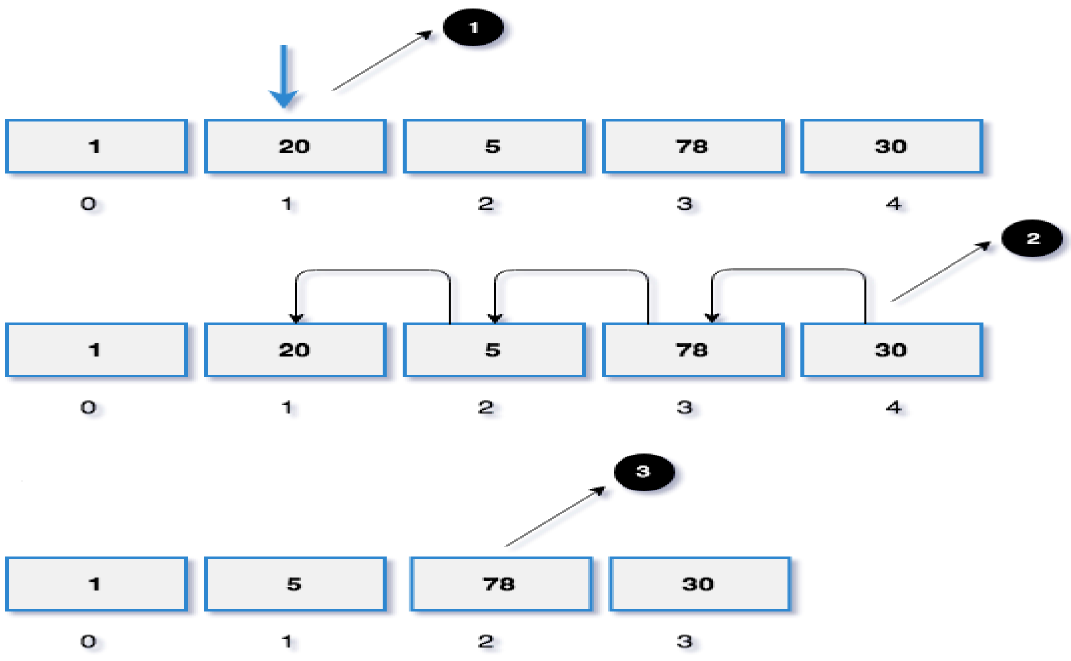
```
void freenode()
{
    node[p].next=avail;
    avail=p;
    return;
}
```

Example1:

Let's take an array of 5 elements.

1, 20, 5, 78, 30.

If we remove element 20 from the array, the execution will be,

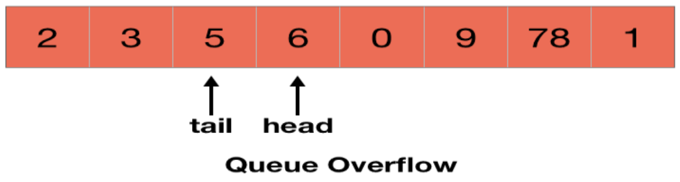


Queue as list:

We will maintain two pointers - *tail* and *head* to represent a queue. *head* will always point to the oldest element which was added and *tail* will point where the new element is going to be added. Setting Q.tail = -1 and Q.head = -1

Insertion:

```
if ((Q.head=Q.tail+1)|| (Q.tail = Q.size-1 && Q.head =0))
      Error "Queue Overflow"
Else
      if (Q.head ==-1)
              set Q.tail = 0 and Q.head = 0
       else
              Q.tail = (Q.tail+1) % size
       Q[Q.tail] = x
```



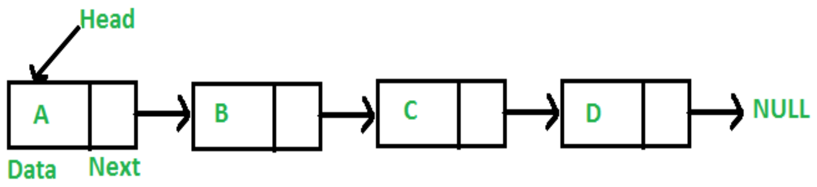
Deletion:

```
if Q.head == -1
    Error “Queue Underflow”
else
    x = Q[Q.head]
    if Q.head == Q.tail
       set Q.head = -1 and Q.tail = -1
    else
        Q.head = (Q.head+1) % size
```

Chapter 4 Linked list

Introduction:

- A linked list is a collection of elements called 'nodes' where each node consists of two parts:
    - **Info**: Actual element to be stored in the list. It is called data field.
    - **Next**: one or more link that points to next and previous node in the list. It is also called pointer field.

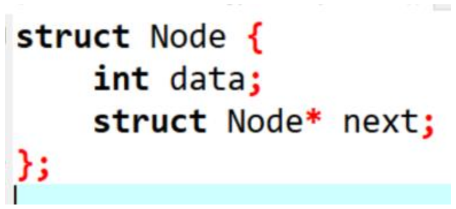


- The link list is a dynamic structure i.e. it grows or shrinks depending on different operations performed. The whole list is pointed to by an external pointer called head which contains the address of the first node. It is not the part of linked list.
- The last node has some specified value called NULL as next address which means the end of the list.

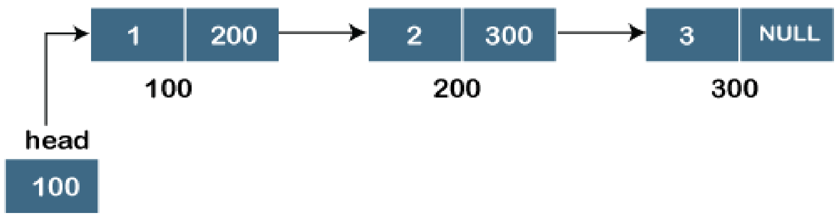
Types of Linked List:

1. Single linked list:

- It is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type. The node contains a pointer to the next node means that the node stores the address of the next node in the sequence. A single linked list allows traversal of data only in one way.

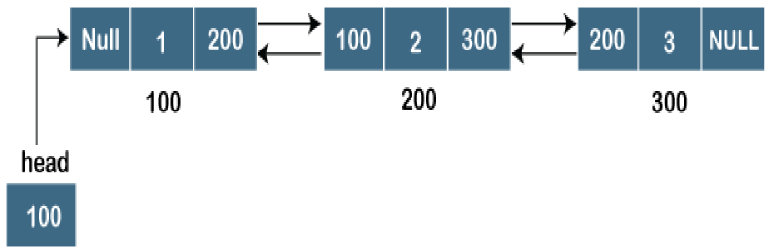


```
struct Node {
    int data;
    struct Node* next;
};
```



2. Doubly Linked list:

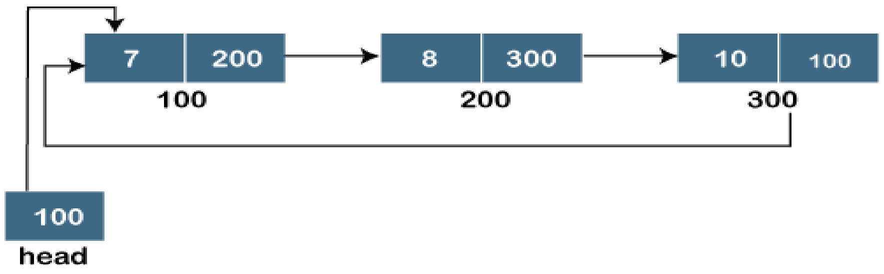
- A doubly linked list or a two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in sequence, Therefore, it contains three parts are data, a pointer to the next node, and a pointer to the previous node. This would enable us to traverse the list in the backward direction as well.



```
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};
```

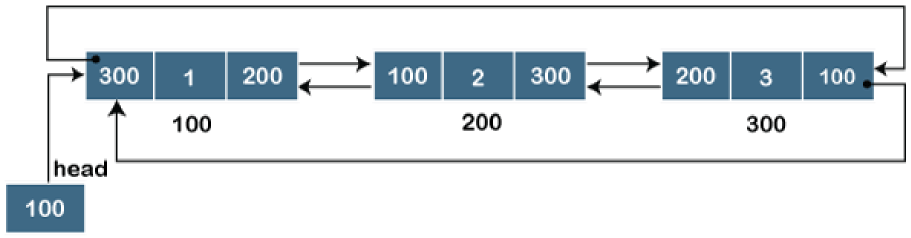
3. Circular linked list:

- A circular linked list is that in which the last node contains the pointer to the first node of the list. While traversing a circular liked list, we can begin at any node and traverse the list in any direction forward and backward until we reach the same node we started. Thus, a circular linked list has no beginning and no end.



4. Doubly Circular Linked List:

- A Doubly Circular linked list or a circular two-way linked list is a more complex type of linked-list that contains a pointer to the next as well as the previous node in the sequence. The circular doubly linked list does not contain null in the previous field of the first node.



Chapter 4 Linked list

Dynamic implementation:

Dynamic Memory Allocation: it is a procedure in which the size of data structure is changed during the runtime.

Malloc ():

It is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn't Initializes memory at execution time so that it has initializes each block with the default garbage value initially.

```
ptr = (cast-type*) malloc(byte-size)
```

e.g. ptr= (int*) malloc(100*size of (int));

Calloc ():

"calloc" or "contiguous allocation" method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value '0' and has two parameters.

ptr = (cast-type*)calloc(n, element-size);

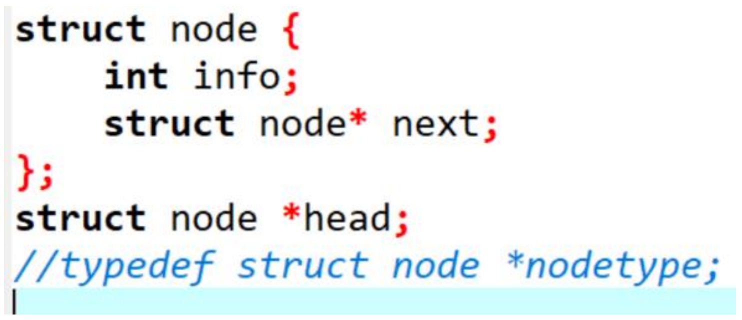
here, n is the no. of elements and element-size is the size of each element.

Free ():

It is used to dynamically de-allocate the memory. The memory allocated using functions malloc () and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Free (ptr);

Representation in C:



```
struct node {
    int info;
    struct node* next;
};
struct node *head;
//typedef struct node *nodetype;
```

For getnode ():

- Let getnode() be a function that allocates memory for a node, assigns data to the node's info, makes node's next pointer NULL and returns the address of the node.

```
struct node *getnode()
{
    struct node *ptr;
    ptr=(struct node *) malloc(sizeof(struct node));
    printf("Enter a data:");
    scanf("%d",&ptr->info);
    prt->next=NULL;
    return ptr;
}
```

- It allocates the memory for a node dynamically. It is a user defined function that returns a pointer to newly created node.

Operations in Single linked list:

Insertion:

The insertion into a singly linked list can be performed at different positions.

1. **Insertion at beginning:**

   It involves inserting any element at the front of the list. We just need make the new node as the head of the list.

   Algorithm:

   1. Create a node using malloc function

      ```
      newnode=(struct node *)malloc(sizeof(struct node));
      ```

   2. Assign data to info field of new node

      ```
      newnode->info = data;
      ```

   3. if head is NULL then set head=newnode and exit

      ```
      head=newnode;
      ```

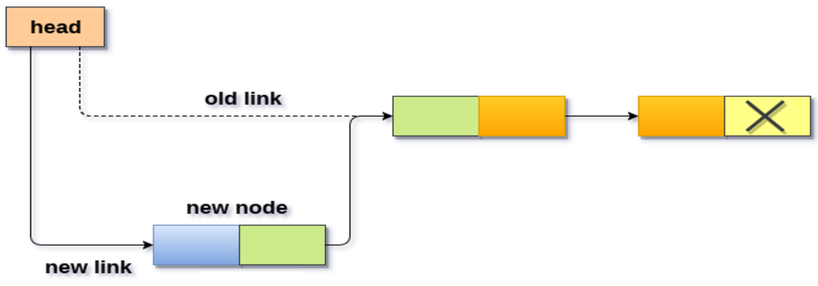
   4. otherwise
      1. Set next of newnode to head

         ```
         newnode->next=head;
         ```

      2. Set the head pointer to point to the new node

         ```
         head=newnode;
         ```

   5. End



2. Insertion at end of the list:

It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one.

Algorithm:

1. Create a node using malloc function

```
newnode=(struct node *)malloc(sizeof(struct node));
```

2. Assign data to info field of new node

```
newnode->info = data;
```

3. Set next of newnode to NULL

```
newnode->next =NULL;
```

4. if head is NULL then set head=newnode and exit
5. Otherwise
    i. Set

    ```
    ptr=head;
    ```

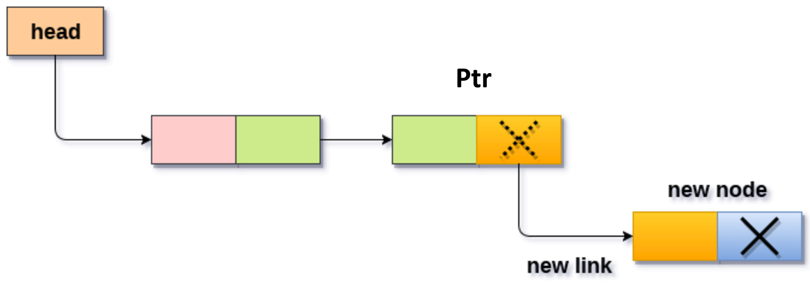
    ii. Find the last node

    ```
    while(ptr->next !=NULL)
    {
            ptr=ptr->next ;
    }
    ```

    iii. Set ptr->next =newnode

    ```
    ptr->next =newnode;
    ```

6. end



3. Insertion at specified position:

It involves insertion at specified position of the linked list. We need to skip the desired number of nodes in order to reach the node at which the new node will be inserted.

1. Create a node using malloc function

```
newnode=(struct node *)malloc(sizeof(struct node));
```

2. Assign data to info field of new node

```
newnode->info = data;
```

3. Enter the position of a node at which you want to insert a newnode.
4. Let the position be pos
5. Set

```
ptr=head;
```

```
for(i=0;i<pos-1;i++)
{ ptr=ptr->next;
        if(ptr==NULL)
        {
                printf("\nPosition not found:[Handle with care]\n");
                return;
        }
}
```

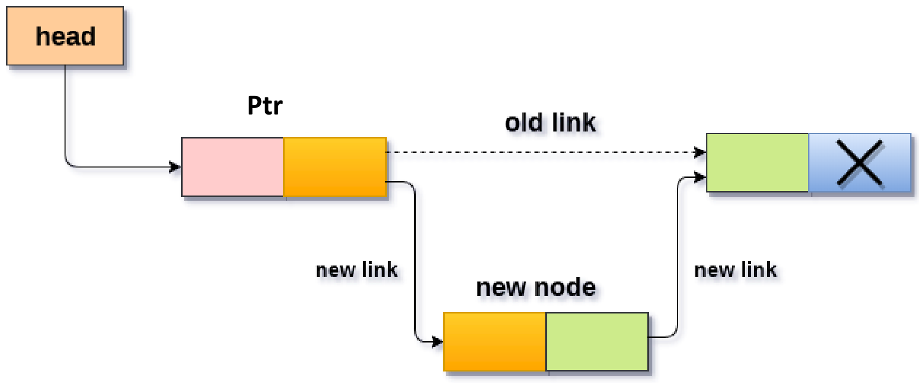
6. Set

```
newnode->next =ptr->next ;
```

7. Set next of ptr to point to the newnode

```
ptr->next=newnode;
```

8. End



Deletion:

1. Deletion at beginning:

It involves deletion of a node from the beginning of the list.

Let head be the pointer to the first node in the linked list

1. If (head==NULL) then print void deletion and exit i.e.

```
if(ptr==NULL)
{
        printf("\nList is Empty:\n");
        return;
}
```

2. Otherwise store the address of the first node in temporary variable ptr

```
ptr=head;
```

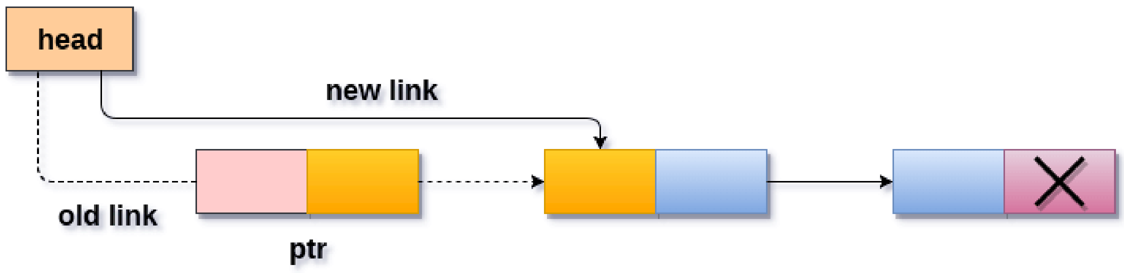
3. Set head of the next node to head

```
head=head->next ;
```

4. Free the memory reserved by temp variable

```
free(ptr);
```

5. End



2. Deletion at the end of the list:

It involves deleting the last node of the list.

1. If (head==NULL) then print void deletion and exit i.e.

```
if(ptr==NULL)
{
        printf("\nList is Empty:\n");
        return;
}
```

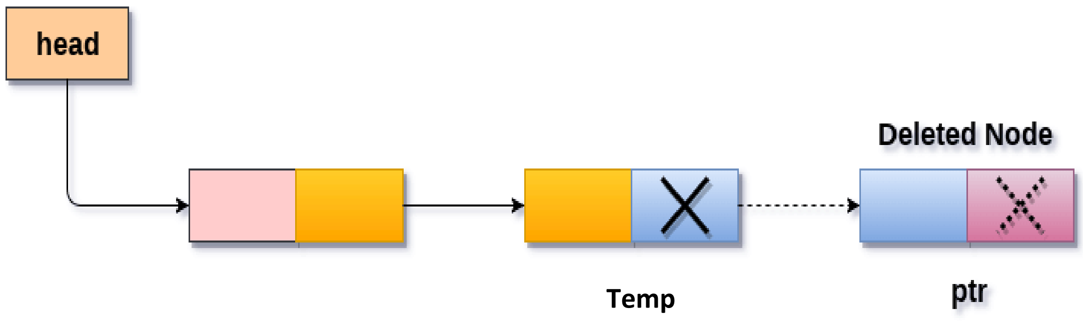
2. Otherwise if (head->next==NULL) then set ptr=head, head=NULL and free ptr. i.e.

```
else if(head->next ==NULL)
{
        ptr=head;
        head=NULL;
        printf("\n The deleted element is:%d \t",ptr->info);
        free(ptr);
}
```

3. Otherwise

```
ptr=head;
while(ptr->next!=NULL)
{
        temp=ptr;
        ptr=ptr->next;
}
temp->next=NULL;
printf("\n The deleted element is:%d \t",ptr->info);
free(ptr);
```

4. End



3. **Deletion of specified node:**

   It involves deletion of the specified node in the list. We need to skip the desired number of nodes to reach the node which will be deleted.

1. If head=NULL print empty list and exit i.e.

```
if(head==NULL)
{
        printf("\n The List is Empty: \n");
        exit(0);
}
```

2. Otherwise
   i. Enter the position pos of the node to be deleted
   ii. If pos=0
      i. Set ptr=head and head=head->next and free ptr i.e.

```
ptr=head;
head=head->next ;
printf("\n The deleted element is:%d \t",ptr->info  );
free(ptr);
```

   iii. Otherwise
      i. Set

```
ptr=head;
for(i=0;i<pos;i++)
 {
  temp=ptr;
  ptr=ptr->next ;
        if(ptr==NULL)
        {
                printf("\n Position not Found: \n");
                return;
        }
 }
```

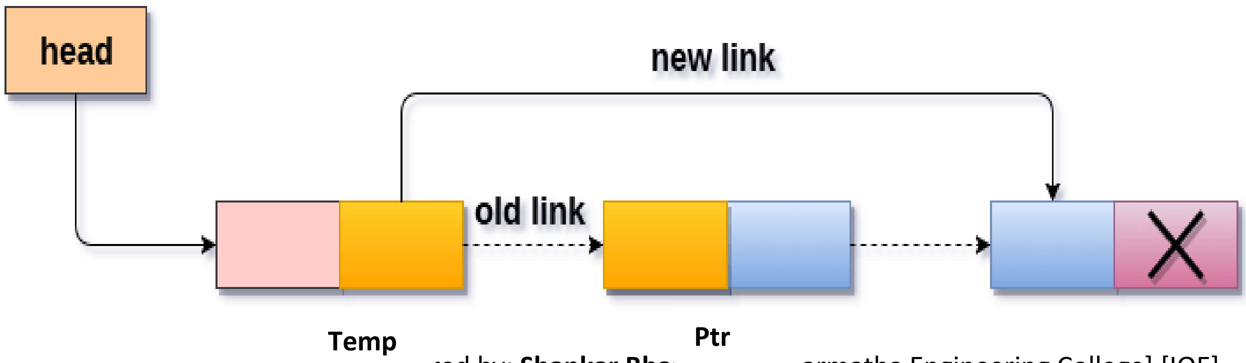
      ii. Set

```
temp->next =ptr->next ;
```

      iii. Free ptr i.e.

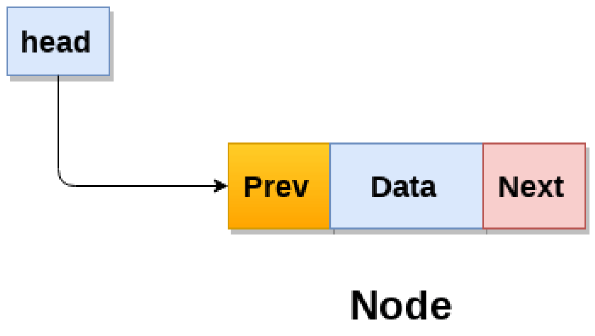
```
free(ptr);
```

3. End



Doubly Linked list:

A doubly linked list is one in which all nodes are linked together by multiple number of links which helps in accessing both the successor node and predecessor node for the given node position. It is bi-directional traversing. Each node in a doubly linked list has two pointer fields and one data field. The pointer fields are used to point successor and predecessor node.



Representation in C:

```
struct node
{
        int info;
        struct node *next;
        struct node *prev;
};
struct node *head=NULL;
//typedef struct node *nodetype;
```

Chapter 4 Linked list

Operations in linked list:

Insertion:

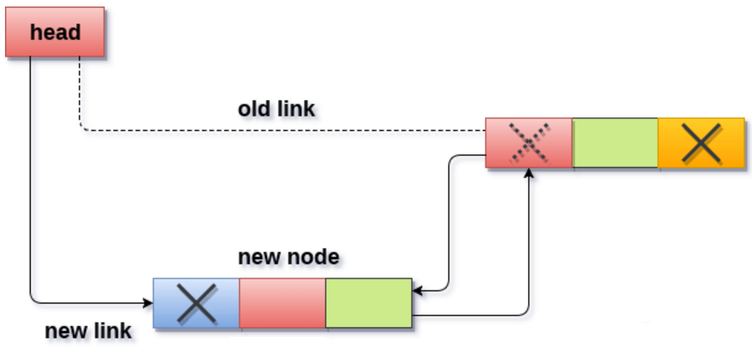
The insertion into a doubly linked list can be performed at different positions.

1. **Insertion at beginning:**

   It involves inserting any element at the front of the list. We just need to make the new node as the head of the list.

   Algorithm:

```
1. Create a node using malloc function
   newnode=(struct node *)malloc(sizeof(struct node));
2. Assign data to info field of new node
   newnode->info = data;
3. Set
   newnode->prev =NULL;
4. Set
   newnode->next=NULL;
5. if head is NULL then set head=newnode
           head=newnode;
6. otherwise
      i. Set next of newnode to head
           newnode->next=head;
      ii. Set prev of head to newnode
           head->prev =newnode;
      iii. Set the head pointer to point to the new node
           head=newnode;
7. End
```



2. Insertion at end of the list:

It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one.

Algorithm:

1. Create a node using malloc function

```
newnode=(struct node *)malloc(sizeof(struct node));
```

2. Assign data to info field of new node

```
newnode->info = data;
```

3. Set prev of newnode to NULL

```
newnode->prev =NULL;
```

4. Set next of newnode to NULL

```
newnode->next =NULL;
```

5. if head is NULL then set head=newnode and exit
6. Otherwise
    i. Set

    ```
    ptr=head;
    ```

    ii. Find the last node

    ```
    while(ptr->next !=NULL)
    {
            ptr=ptr->next ;
    }
    ```

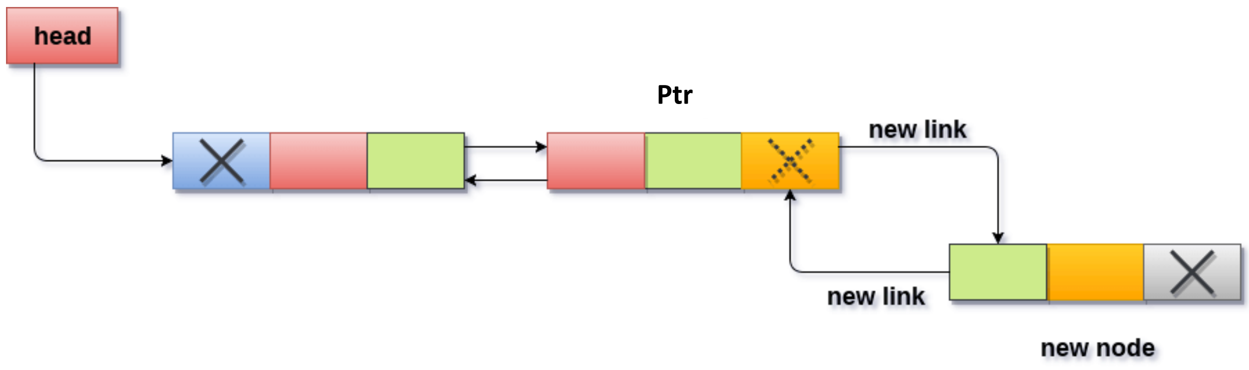
    iii. Set ptr->next =newnode

    ```
    ptr->next =newnode;
    ```

    iv. Set

    ```
    newnode->prev=ptr;
    ```

7. End



3. Insertion at specified position:

It involves insertion at specified position of the linked list. We need to skip the desired number of nodes in order to reach the node at which the new node will be inserted.

1. Create a node using malloc function

```
newnode=(struct node *)malloc(sizeof(struct node));
```

2. Assign data to info field of new node

```
newnode->info = data;
```

3. Set prev of newnode to NULL

```
newnode->prev =NULL;
```

4. Set next of newnode to NULL

```
newnode->next =NULL;
```

5. Enter the position of a node at which you want to insert a newnode.
6. Let the position be pos
7. Set

```
ptr=head;


for(i=0;i<pos-1;i++)
{ ptr=ptr->next;
        if(ptr==NULL)
        {
                printf("\nPosition not found:[Handle with care]\n");
                return;
        }
}
```

8. Set

```
newnode->next =ptr->next ;
```

9. Set

```
newnode->prev=ptr;
```

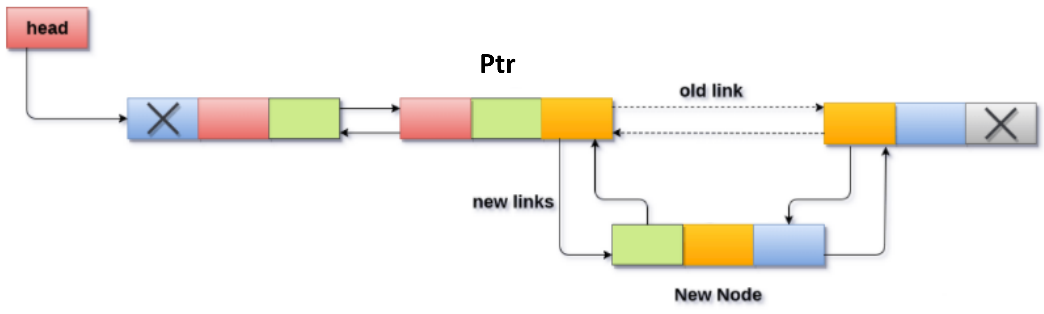
10. Set

```
ptr->next->prev=newnode;
```

11. Set

```
ptr->next=newnode;
```

12. End



Deletion:

1. Deletion at beginning:

It involves deletion of a node from the beginning of the list.

Let head be the pointer to the first node in the linked list

1. If (head==NULL) then print void deletion and exit i.e.

```
if(ptr==NULL)
{
        printf("\nList is Empty:\n");
        return;
}
```

2. Otherwise store the address of the first node in temporary variable ptr

```
ptr=head;
```

3. Set head of the next node to head

```
head=head->next ;
```

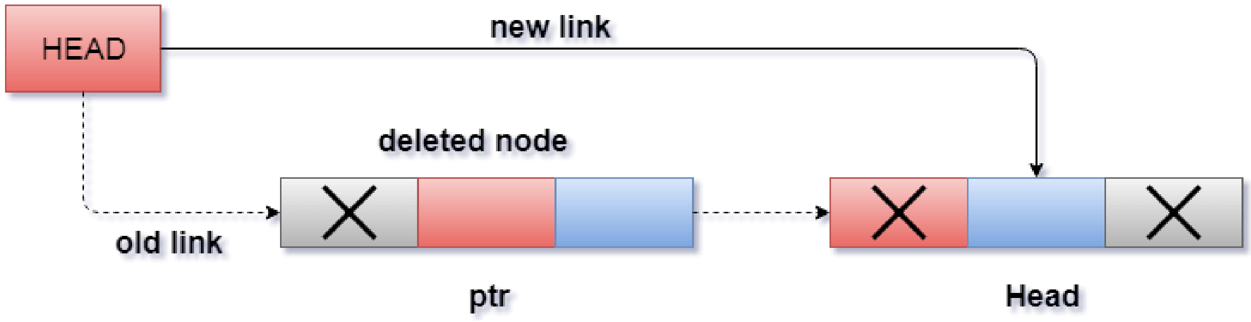
4. Set

```
head->prev=NULL;
```

5. Free the memory reserved by temp variable

```
free(ptr);
```

6. End



2. Deletion at the end of the list:

It involves deleting the last node of the list.

1. If (head==NULL) then print void deletion and exit i.e.

```
if(ptr==NULL)
{
        printf("\nList is Empty:\n");
        return;
}
```

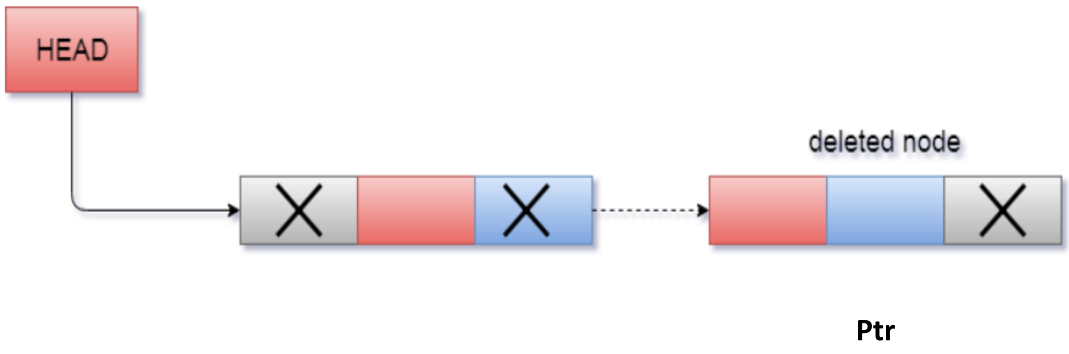
2. Otherwise if (head->next==NULL) then set ptr=head, head=NULL and free ptr. i.e.

```
else if(head->next ==NULL)
{
        ptr=head;
        head=NULL;
        printf("\n The deleted element is:%d \t",ptr->info);
        free(ptr);
}
```

3. Otherwise

```
ptr=head;
while(ptr->next!=NULL)
{
        ptr=ptr->next;
}
ptr->prev->next=NULL;
printf("\n The deleted element is:%d \t",ptr->info);
free(ptr);
```

4. End



4. Deletion of specified node:

It involves deletion of the specified node in the list. We need to skip the desired number of nodes to reach the node which will be deleted.

1. If head=NULL print empty list and exit i.e.

```
if(head==NULL)
{
        printf("\n The List is Empty: \n");
        exit(0);
}
```

2. Otherwise
3. Enter the position pos of the node to be deleted
4. If pos=0

```
ptr=head;
head=head->next ;
head->prev=NULL;
printf("\n The deleted element is:%d \t",ptr->info  );
free(ptr);
```

5. Otherwise
   i. Set

```
ptr=head;
for(i=0;i<pos;i++)
 {

  ptr=ptr->next ;
        if(ptr==NULL)
        {
                printf("\n Position not Found: \n");
                return;
        }
}
```

   ii. Set

```
ptr->prev->next =ptr->next ;
```

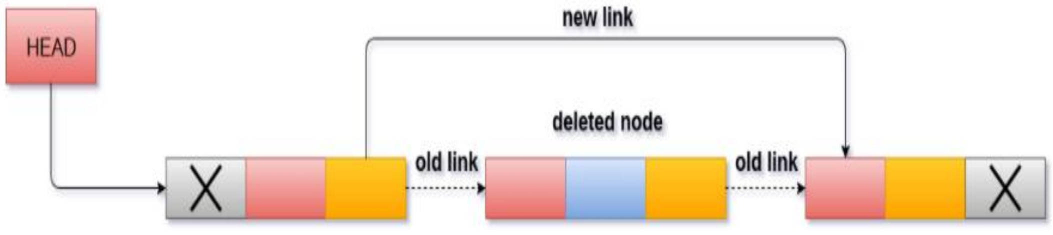
   iii. Set

```
ptr->next->prev=ptr->prev;
```

   iv. Free ptr i.e.

```
free(ptr);
```

6. End



Ptr

Advantages of Doubly linked list:

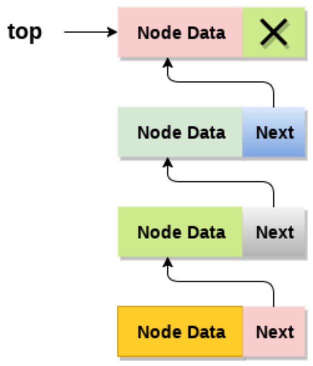
- Reversing the doubly linked list is very easy.
- The traversal of this doubly linked list is bidirectional which is not possible in a singly linked list.
- Deletion of nodes is easy as compared to a Singly Linked List. A singly linked list deletion requires a pointer to the node and previous node to be deleted but in the doubly linked list, it only required the pointer which is to be deleted.

Disadvantages of Doubly linked list:

- It uses extra memory when compared to the array and singly linked list.
- Since elements in memory are stored randomly, therefore the elements are accessed sequentially no direct access is allowed.

Linked stacks and queues

A stack can be easily implemented through the linked list. In stack Implementation, a stack contains a top pointer. Which is “head” of the stack where pushing and popping items happens at the head of the list. Each node contains a pointer to its immediate successor node in the stack.



Stack

Push Operation:

It is similar to the insertion at the beginning of the link list. It involves inserting any element at the beginning of the list.

1. Create a node using malloc function

   ```
   newnode=(struct node *)malloc(sizeof(struct node));
   ```

2. Assign data to info field of new node

   ```
   newnode->info = data;
   ```

3. Set

   ```
   newnode->next =NULL;
   ```

4. if top == NULL then set head=newnode and exit

   ```
   top=newnode;
   ```

5. otherwise
   1. Set next of newnode to top

      ```
      newnode->next=top;
      ```

   2. Set the top pointer to point to the new node

      ```
      top=newnode;
      ```

6. Print item pushed
7. End

Pop Operation:

It is similar to the deletion from the beginning of the link list. It involves deletion of a node from the beginning of the list.

Let top be the pointer to the first node in the linked list

1. If (top==NULL) then print void deletion and exit i.e.

```
if(top==NULL)
{
        printf("\nList is Empty:\n");
        return;
}
```

2. Otherwise
   i. store the address of the first node in temporary variable ptr

   ```
   ptr=top;
   ```

   ii. Set top of the next node to top

   ```
   top=top->next ;
   ```

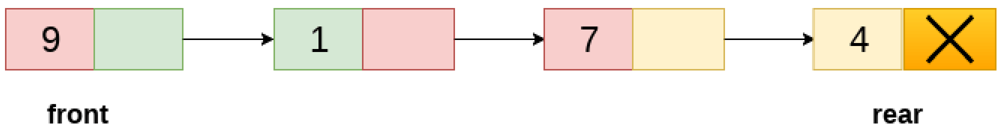
   iii. Free the memory reserved by temp variable

   ```
   free(ptr);
   ```

3. End

Linked list as Queue:

Each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory. In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.



Enqueue Operation:

The insert operation append the queue by adding an element to the end of the queue. The new element will be the last element of the queue.

```
1. Allocate the memory for the new node
       newnode=(struct node *)malloc(sizeof(struct node));
2. Assign data to info field of new node
       newnode->info = data;
3. If front=NULL then
   i. Set
       front=newnode;
       rear=newnode;
   ii. Set
       front->next=NULL;
       rear->next=NULL;
4. Otherwise
   i. Set
       rear->next = newnode;
       rear = newnode;
       rear->next = NULL;
5. End
```

Dequeue Operation:

Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not.

```
1. If front=NULL
   i. Print underflow and exit i.e.
       printf("\nUNDERFLOW\n");
       return;
2. Otherwise
       ptr = front;
       front = front -> next;
       free(ptr);
3. End
```

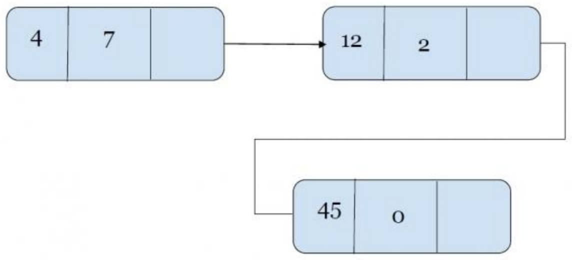
Adding two polynomials using Linked List:

In the Polynomial linked list, the coefficients and exponents of the polynomial are defined as the data node of the list.

For adding two polynomials that are stored as a linked list. We need to add the coefficients of variables with the same power. In a linked list node contains 3 members, coefficient value link to the next node. We check values at the exponent value of the node. For the same values of exponent, we will add the coefficients.

A linked list that is used to store Polynomial looks like −

Polynomial: $4x^7 + 12x^2 + 45$



We start with the term which is of highest degree in any of the polynomials. If there is no item having same exponent, we simply append the term to the new list, and continue with the process. Wherever we find that the exponent match, we simply add the coefficients and then store the term in the new list. If one list gets exhausted earlier and the other list still contains some lower order terms than simply append the remaining terms to the new list.

Example:

Input:

p1= $13x^8 + 7x^5 + 32x^2 + 54$

p2= $3x^{12} + 17x^5 + 3x^3 + 98$

Output:

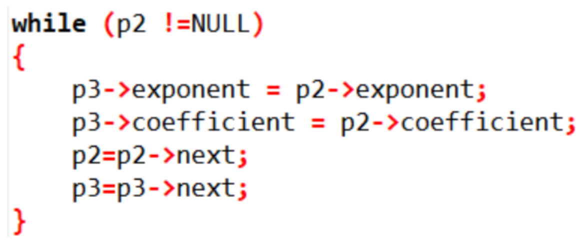
P3= $3x^{12} + 13x^8 + 24x^5 + 3x^3 + 32x^2 + 152$

Algorithm:

Let phead1, phead2 and phead3 represent the pointer of the three lists under consideration. We want to add phead1 and phead2, and store the result in phead3. This addition can be performed using the procedure below.

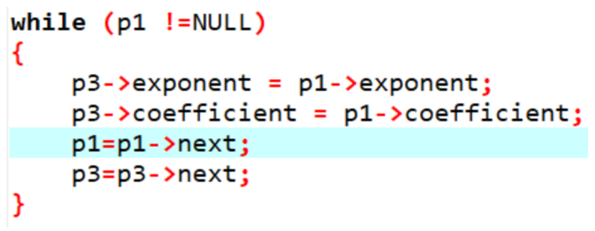
p1=phead1; p2=phead2; p3=phead3;

1. If both the polynomials are null then return
2. else if **p1 = Null** then



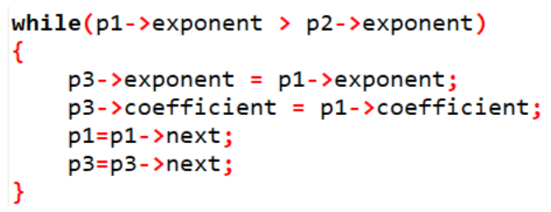
```
while (p2 !=NULL)
{
     p3->exponent = p2->exponent;
     p3->coefficient = p2->coefficient;
     p2=p2->next;
     p3=p3->next;
}
```

3. Else if **p2 = Null** then



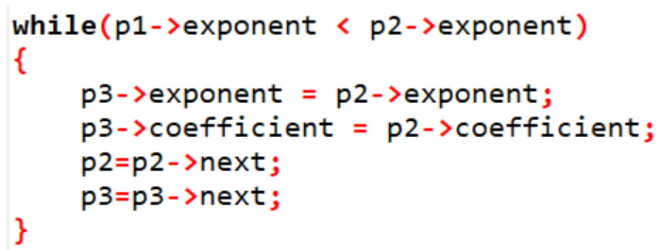
```
while (p1 !=NULL)
{
     p3->exponent = p1->exponent;
     p3->coefficient = p1->coefficient;
     p1=p1->next;
     p3=p3->next;
}
```

4. Else
   a. case 1



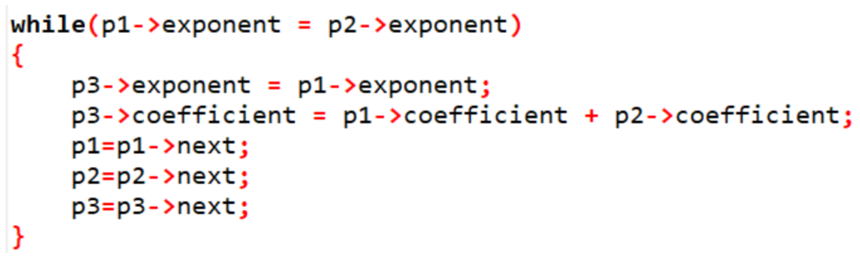
```
while(p1->exponent > p2->exponent)
{
     p3->exponent = p1->exponent;
     p3->coefficient = p1->coefficient;
     p1=p1->next;
     p3=p3->next;
}
```

   b. case 2



```
while(p1->exponent < p2->exponent)
{
     p3->exponent = p2->exponent;
     p3->coefficient = p2->coefficient;
     p2=p2->next;
     p3=p3->next;
}
```

   c. Case 3



```
while(p1->exponent = p2->exponent)
{
     p3->exponent = p1->exponent;
     p3->coefficient = p1->coefficient + p2->coefficient;
     p1=p1->next;
     p2=p2->next;
     p3=p3->next;
}
```

5. End

Chapter 4 Linked list

Circular linked list

It is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.

Advantages of Circular Linked Lists:

1. Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
2. Useful for implementation of queue.
3. Circular linked list are useful in applications to repeatedly go around the list like CPU scheduling.

Operations:

Insertion:

At the beginning:

1. Create a node using malloc function

```
newnode=(struct node *)malloc(sizeof(struct node));
```

2. Assign data to info field of new node

```
newnode->info = data;
```

3. If list is empty i.e.

```
if(head==NULL)
{
        head=newnode;
        newnode->next=head;
}
```

4. Otherwise

```
ptr=head;
while(ptr->next !=head)
{
        ptr=ptr->next ;
}

ptr->next =newnode;
head=newnode;
```

5. End

CHAPTER 5 RECURSION

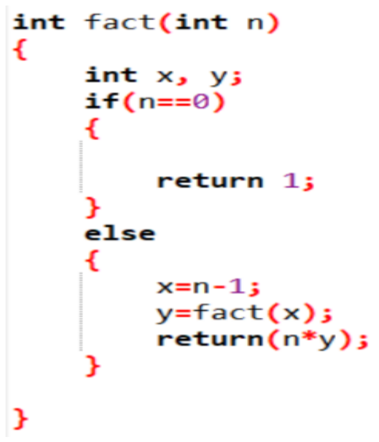
The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Recursion is used to solve problems involving iterations in reverse order. It solves a problem by reducing it to an instance of the same problem with smaller input. Recursion is an alternative to each iteration in making a function executes repeatedly.

Properties of Recursion:

A recursive function can go infinite liked a loop, to avoid infinite running of recursive function. There are two properties that a recursive function must have:

1. **Base Criteria:** There must be at least one base criteria or conditions such that when this condition is met the function stops calling itself recursively.
2. **Progressive Call:** The recursive calls should progress in such a way that each time a recursion call is made, it comes closer to the base case.

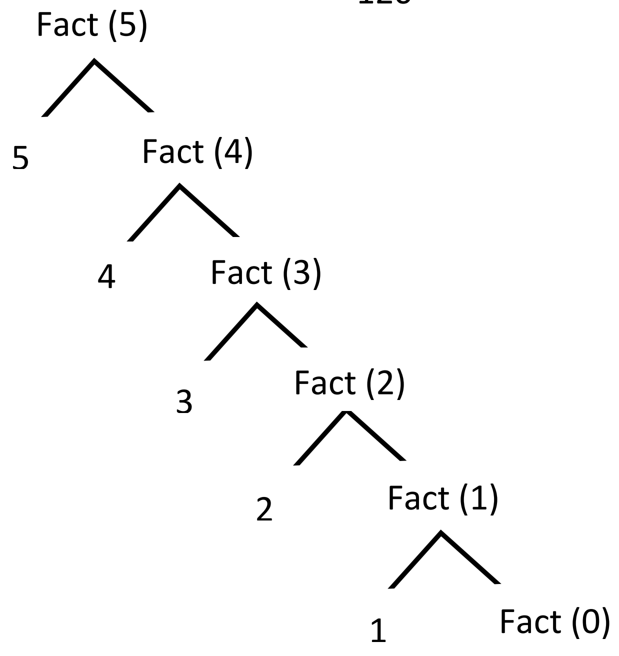
The classic example of recursive programming involves computing factorials. The factorial of a number is computed as that number times all of the numbers below it up to and including 1.



```
int fact(int n)
{
     int x, y;
     if(n==0)
     {
          return 1;
     }
     else
     {
          x=n-1;
          y=fact(x);
          return(n*y);
     }

}
```

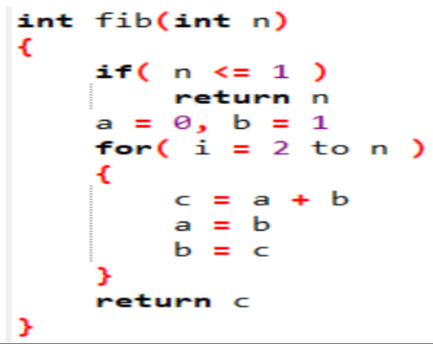
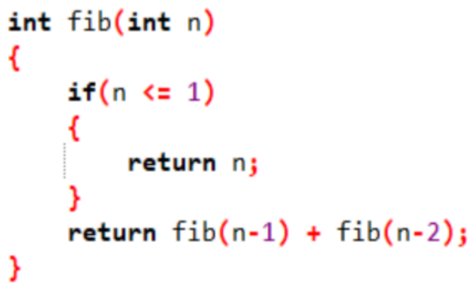
5!
=5*4!
=5*4*3!
=5*4*3*2!
=5*4*3*2*1!
=5*4*3*2*1*0!
=5*4*3*2*1
=5*4*3*2
=5*4*6
=5*24
=120

Recursion Tree for fact (5)



CHAPTER 5 RECURSION

Difference between Iteration and Recursive function:

Iteration	Recursion
• It is a process of executing statements repeatedly, until some specific condition is specified	• Recursion is a technique of defining anything in terms of itself
• Iteration involves four clear cut steps, initialization, condition, execution and updating	• There must be an base condition inside the recursive function specifying stopping condition
• The value of control variable moves towards the value in condition	• The function state converges towards the base case
• Any recursive problem can be solved iteratively	• Not all problems has recursive solution
• Iteration code tends to be bigger in size	• Recursion decrease the size of code
• An iteration does not use the stack so it's faster than recursion.	• It is usually much slower because all function calls must be stored in a stack to allow the return back to the caller functions.
• Iteration consumes less memory.	• Recursion uses more memory than iteration.
• E.g. 	• E.g. 

```
int fib(int n)
{
        if( n <= 1 )
            return n
        a = 0, b = 1
        for( i = 2 to n )
        {
            c = a + b
            a = b
            b = c
        }
        return c
}
```

```
int fib(int n)
{
    if(n <= 1)
    {
        return n;
    }
    return fib(n-1) + fib(n-2);
}
```

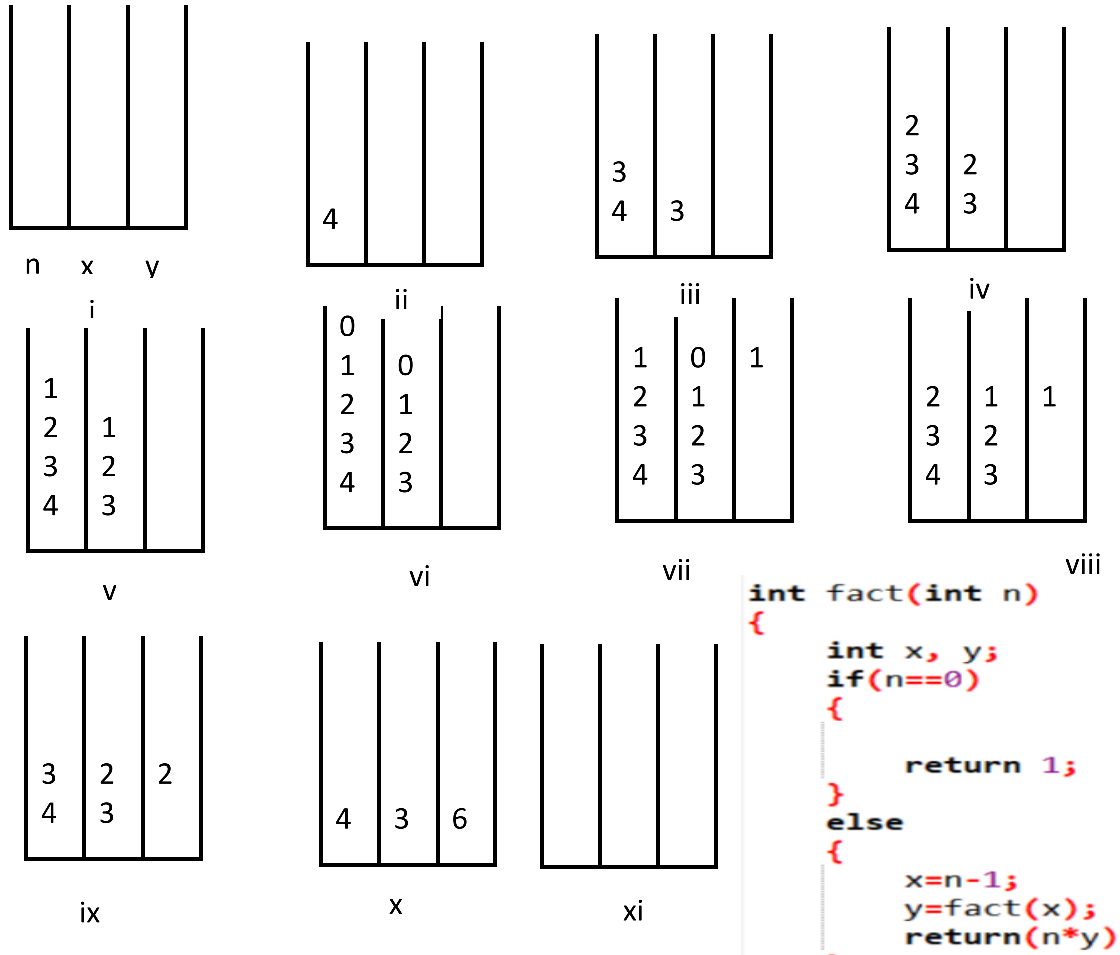
CHAPTER 5 RECURSION

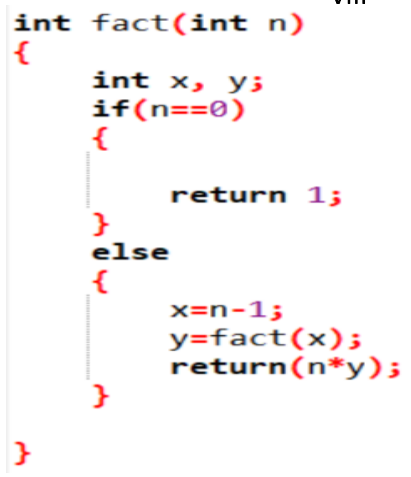
Recursive Program Using Stack:

Recursive functions use something called “the call stack.” When a program calls a function, that function goes on top of the call stack.

Stack is used to keep the successive generations of local variables, the parameters and the returned values. This stack is maintained by the C system and lies inside and invisible to the users.

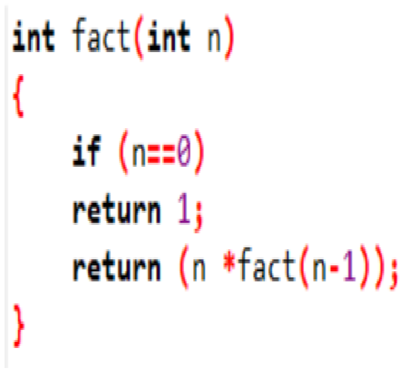
Each time that a recursive function is entered, a new allocation of its variables is pushed on top of the stack. Any reference to a local variables or parameter is through the current top of the stack. When the function returns, the stack is popped, the top allocation is freed, and the previous allocation becomes the current stack top to be used for referencing local variables. Figure below shows a snapshots of the stack as execution of the fact function proceeds.



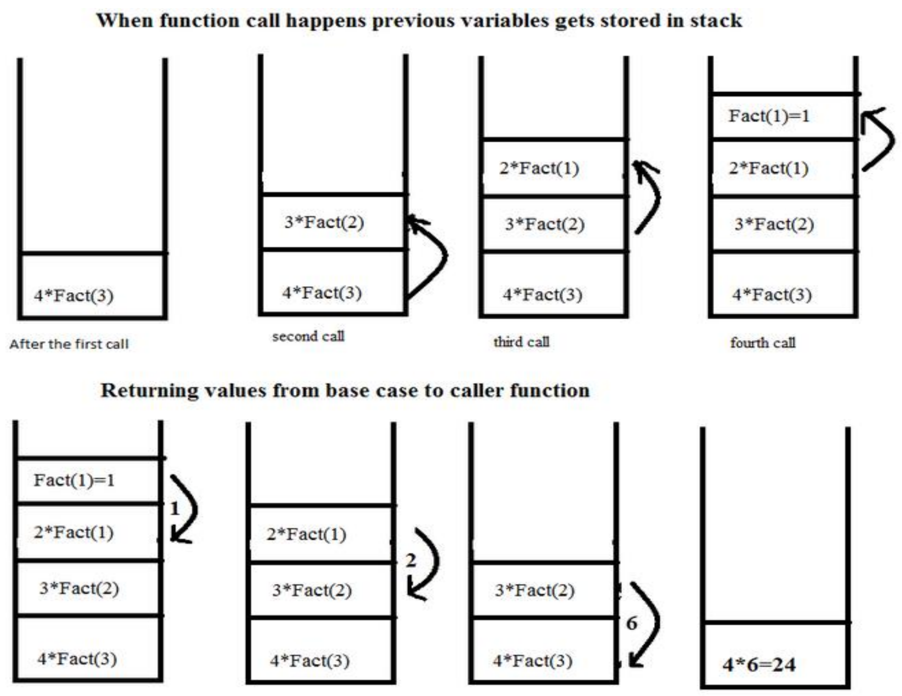


```
int fact(int n)
{
    int x, y;
    if(n==0)
    {

        return 1;
    }
    else
    {
        x=n-1;
        y=fact(x);
        return(n*y);
    }

}
```

CHAPTER 5 RECURSION

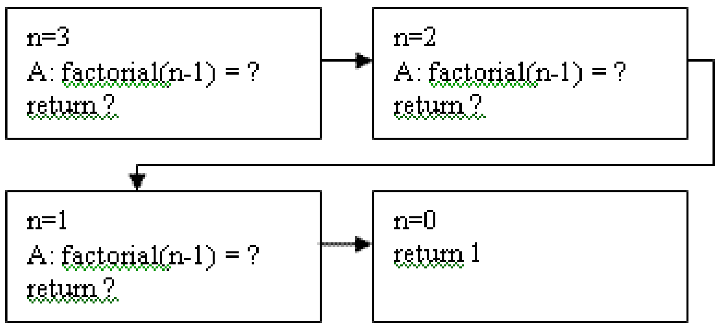


```
int fact(int n)
{
    if (n==0)
    return 1;
    return (n *fact(n-1));
}
```



Box Trace: It helps to understand how a recursive call works.

- Label each recursive call in the body of the recursive method.
- Represent each call to the method by a new box in which you note the method's local environment.
- Draw an arrow from the statement that initiates the recursive process to the first box.
- After you create the new box and arrow, start executing them body of the method.



CHAPTER 5 RECURSION

Time Complexity: we try to figure out the number of times a recursive call is being made. If n number of times a recursive call is made then the time complexity of recursive function is O (2^n) while iterative function is O (n).

Space Complexity: Space complexity is counted as what amount of extra space is required for a module to execute. In iterations, the compiler hardly requires any extra space. The compiler keeps updating the values of variables used in and space complexity is O (1). But in recursion, the system needs to store activation record each time a recursive call is made and space complexity is O (n).

Recursion Tree:

- Recursion tree is another method for solving the recurrence relations.
- A recursion tree is a tree where each node represents the cost of a certain recursive sub-problem.
- We sum up the values in each node to get the cost of the entire algorithm.

Types of Recursive Functions:

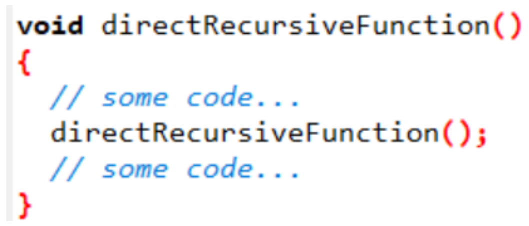
A recursive method is characterized based on:

- Whether the method calls itself or not (direct or indirect recursion)
- Whether there are pending operations at each recursive call (tail recursive or not)

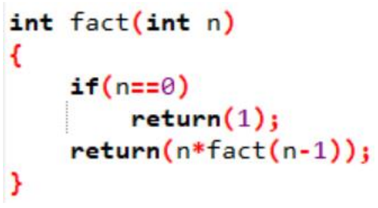
Direct and Indirect Recursion:

Direct Recursion:

If a function calls itself, it's known as direct recursion. A function f1 is called direct recursive if it calls the same function say f1. E.g.



```
void directRecursiveFunction()
{
  // some code...
  directRecursiveFunction();
  // some code...
}
```

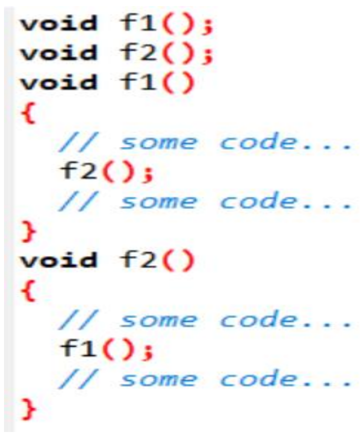


```
int fact(int n)
{
    if(n==0)
        return(1);
    return(n*fact(n-1));
}
```

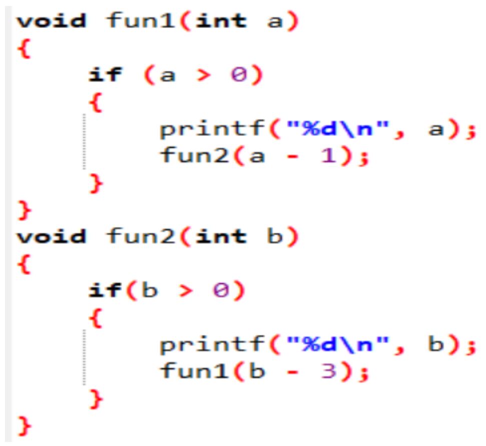
CHAPTER 5 RECURSION

Indirect Recursion:

When a function is mutually called by another function in a circular manner, the function is called an indirect recursion function. If the function f1 calls another function f2 and f2 calls f1 then it is indirect recursion (or mutual recursion). E.g.



```
void f1();
void f2();
void f1()
{
  // some code...
  f2();
  // some code...
}
void f2()
{
  // some code...
  f1();
  // some code...
}
```

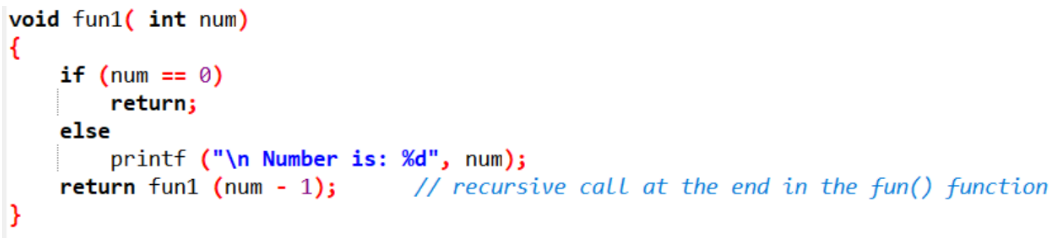


```
void fun1(int a)
{
    if (a > 0)
    {
        printf("%d\n", a);
        fun2(a - 1);
    }
}
void fun2(int b)
{
    if(b > 0)
    {
        printf("%d\n", b);
        fun1(b - 3);
    }
}
```

Tail and Non-Tail Recursion:

Tail Recursion:

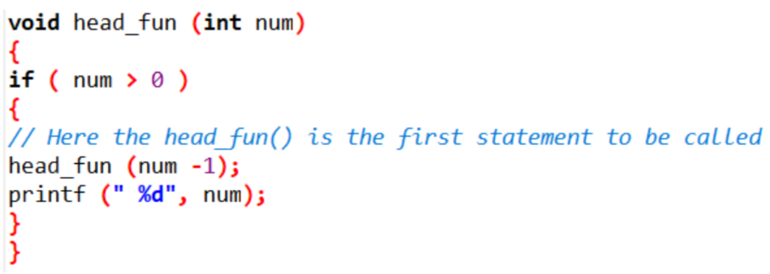
A recursive function is called the tail-recursive if the function makes recursive calling itself, and that *recursive call is the last statement executes by the function.* After that, there is no function or statement is left to call the recursive function.



```
void fun1( int num)
{
    if (num == 0)
        return;
    else
        printf ("\n Number is: %d", num);
    return fun1 (num - 1);      // recursive call at the end in the fun() function
}
```

Non-Tail / Head Recursion:

A function is called the non-tail or head recursive if a function makes a recursive call itself, the recursive call will be the first statement in the function. It means there should be no statement or operation is called before the recursive calls.

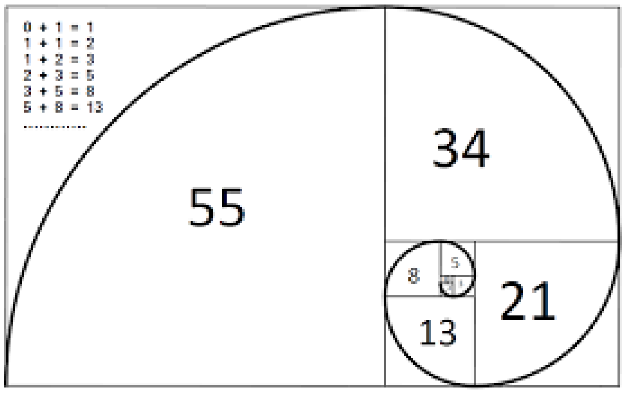


```
void head_fun (int num)
{
if ( num > 0 )
{
// Here the head_fun() is the first statement to be called
head_fun (num -1);
printf (" %d", num);
}
}
```

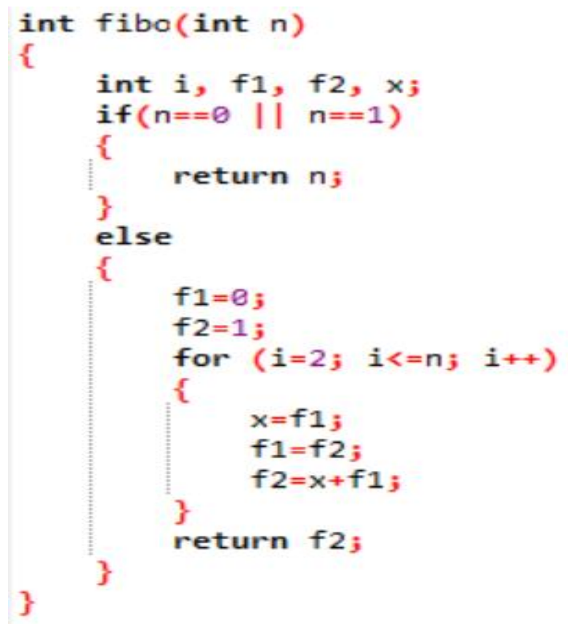
CHAPTER 5 RECURSION

Fibonacci Series:

Fibonacci series is a series of numbers formed by the addition of the preceding two numbers in the series. The first two terms are zero and one respectively. The terms after this are generated by simply adding the previous two terms.

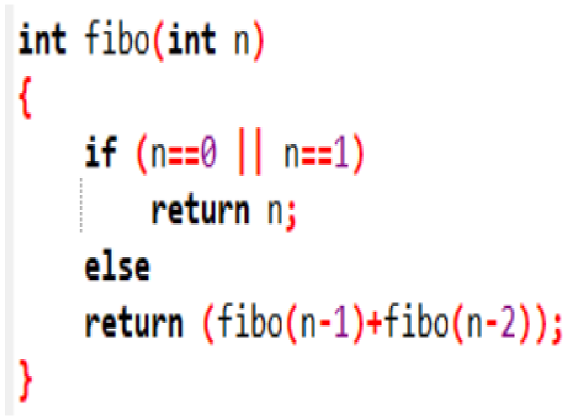


Iteration



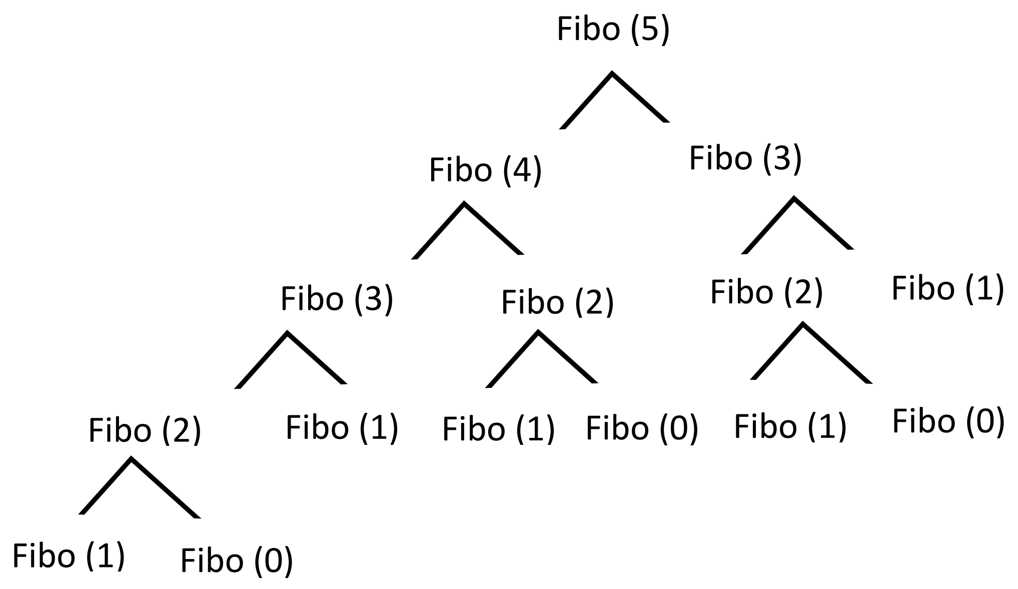
```
int fibo(int n)
{
      int i, f1, f2, x;
      if(n==0 || n==1)
      {
            return n;
      }
      else
      {
            f1=0;
            f2=1;
            for (i=2; i<=n; i++)
            {
                  x=f1;
                  f1=f2;
                  f2=x+f1;
            }
            return f2;
      }
}
```

Recursion



```
int fibo(int n)
{
      if (n==0 || n==1)
            return n;
      else
      return (fibo(n-1)+fibo(n-2));
}
```

Tree diagram for fibo (5)

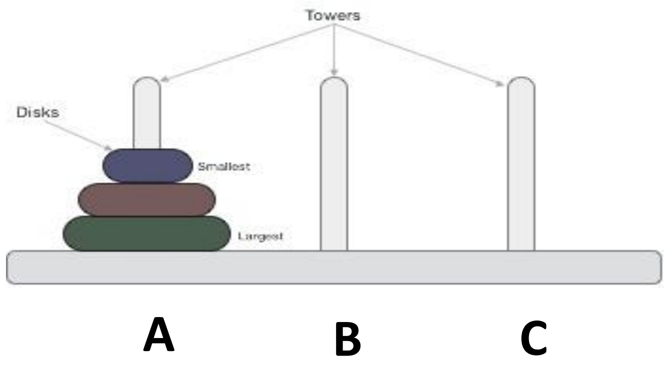


CHAPTER 5 RECURSION

Tower of Hanoi (TOH)

It is also called the problem of Benares Temple or Tower of Brahma or Lucas' Tower. The TOH puzzle was introduced to the west by the French mathematician Edouard Lucas in 1883. Numerous myths regarding the puzzle popped up almost immediately, including one about an Indian temple in Kashi Vishwanath containing a large room with three time-worn posts in it, surrounded by 64 golden disks.

It is a mathematical game or puzzle consisting of three towers (pegs) and a number of disks of various diameters, which can slide onto any tower.



Rules for TOH:

The objective to move all the disks from the peg A to peg C, using peg B as auxiliary. The rules to be followed are:

- Only the top disk on any peg may be moved to any other peg.
- Only one disk can be moved among the towers at any given time.
- Larger disk may never rest on a smaller one.

Recurrence Relation for TOH:

Base case : $H_1 = 1$ (for one disk)

Recursive case: $H_n = H_{n-1} + 1 + H_{n-1}$

$$\begin{aligned} H_n &= 2H_{n-1} + 1 \\ &= 2(2H_{n-2} + 1) + 1 \\ &= 4H_{n-2} + 2 + 1 \\ &= 4(2H_{n-3} + 1) + 2 + 1 \\ &= 2^{n-1} + 2^{n-2} + \dots + 4 + 2 + 1 \qquad [\because a_n = ar^{n-1}] \\ &\quad . \\ &\quad . \\ &= 2^n - 1 \qquad [\because S_n = \frac{a(r^n-1)}{(r-1)}, r \neq 1] \end{aligned}$$

CHAPTER 5 RECURSION

Algorithm for TOH:

To move n disks from A to C, using B as auxiliary:

1. Declare and initialize necessary variables.

   n = number of disks

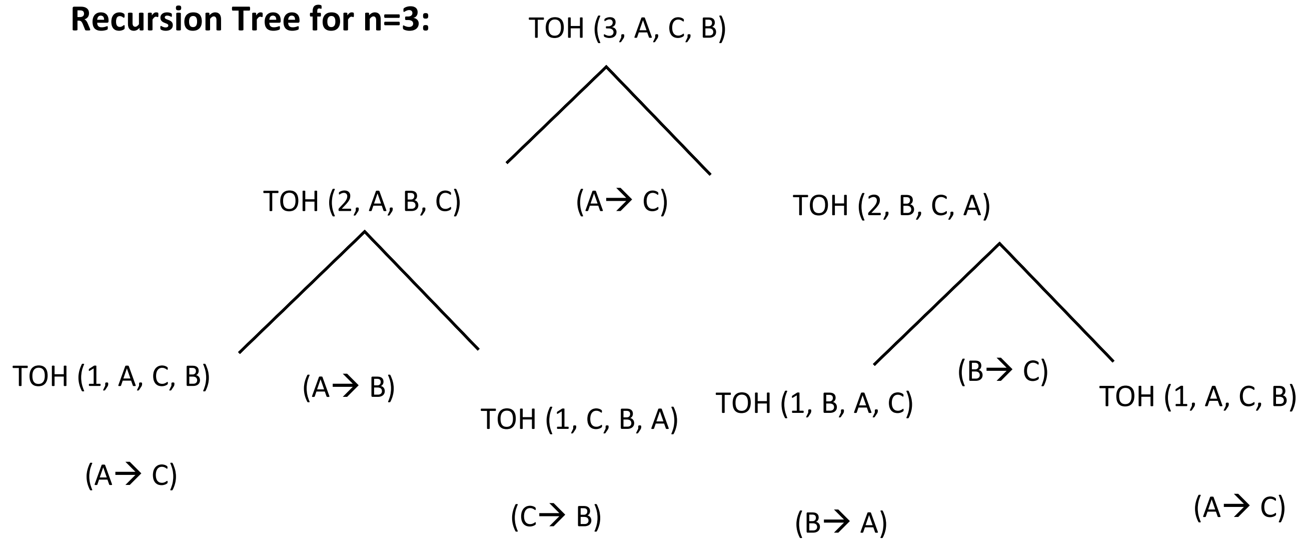
   A='A', B='B', C='C', for three pegs being used.

2. If n == 1,
   - Move the single disk from A to C and stop.
3. Otherwise
   - Move the top n-1 disks from A to B, using C as auxiliary.
   - Move the remaining disk from A to C.
   - Move the n-1 disks from B to C, using A as auxiliary.
4. stop

Algorithm for n=3 disks:

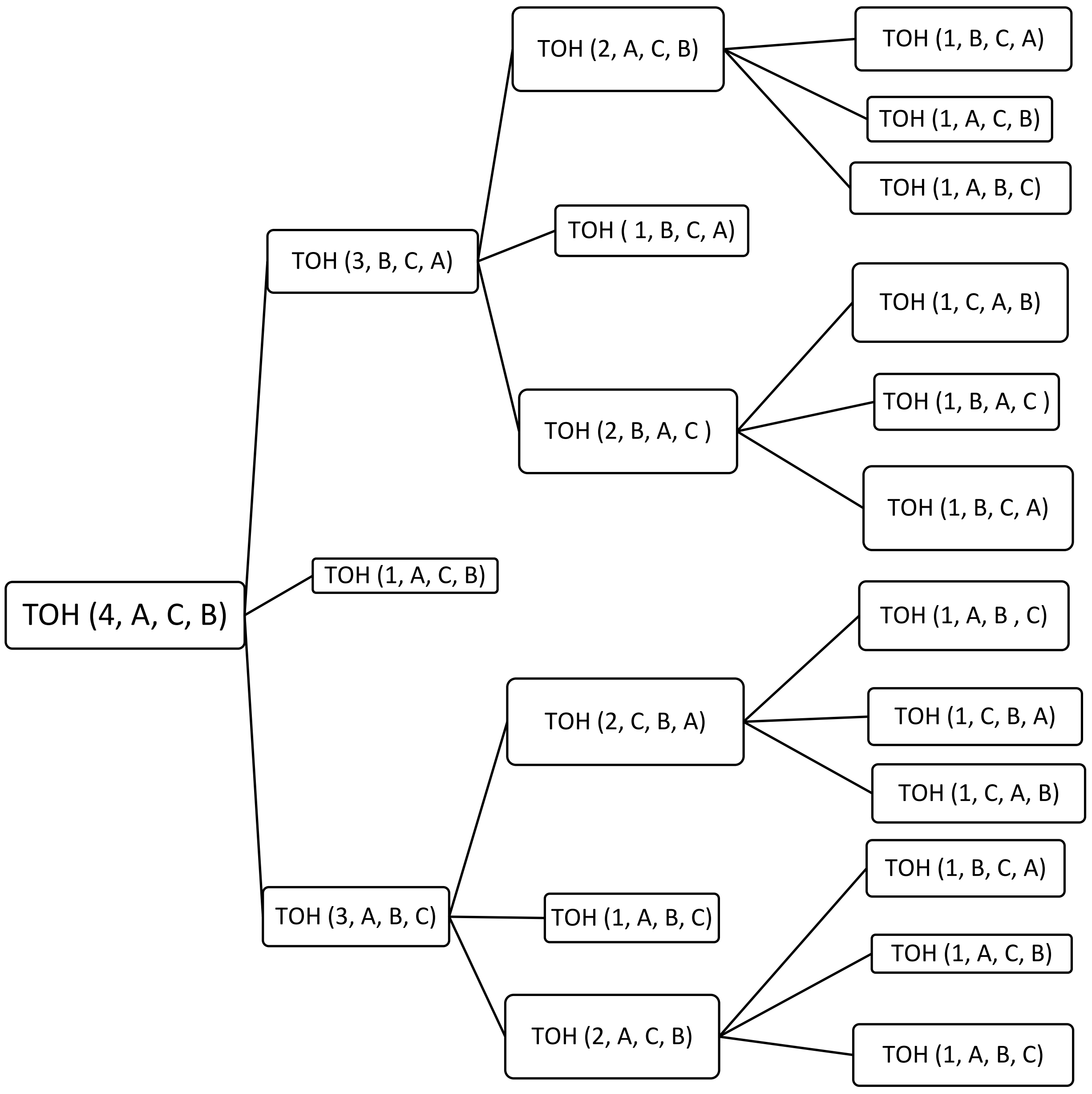
1. Move disk 1 from peg A to peg C
2. Move disk 2 from peg A to peg B
3. Move disk 1 from peg C to peg B
4. Move disk 3 from peg A to peg C
5. Move disk 1 from peg B to peg A
6. Move disk 2 from peg B to peg C
7. Move disk 1 from peg A to peg C

Recursion Tree for n=3:



Algorithm for n=4 disks:

1. Move disk 1 from peg A to peg B
2. Move disk 2 from peg A to peg C
3. Move disk 1 from peg B to peg C
4. Move disk 3 from peg A to peg B
5. Move disk 1 from peg C to peg A
6. Move disk 2 from peg C to peg B
7. Move disk 1 from peg A to peg B
8. Move disk 4 from peg A to peg C
9. Move disk 1 from peg B to peg C
10. Move disk 2 from peg B to peg A
11. Move disk 1 from peg C to peg A
12. Move disk 3 from peg B to peg C
13. Move disk 1 from peg A to peg B
14. Move disk 2 from peg A to peg C
15. Move disk 1 from peg B to peg C



CHAPTER 5 RECURSION

Applications of Tower of Hanoi:

- It is used in psychological research on problem-solving.
- It is used in physical design of the game components.
- It is used as a backup rotation scheme when performing computer data backups where multiple tapes/media are involved.

Application of Recursion:

- The most important data structure 'Tree' doesn't exist without recursion we can solve that in iterative way also but that will be a very tough task.
- The mathematical problem can't be solved in general, but that can only be solved using recursion up to a certain extent.
- Sorting algorithms (Quick sort, Merge sort, etc.) uses recursion.
- All the puzzle games (Chess, Candy crush, etc.) broadly uses recursion.
- It is the backbone of searching, which is most important thing.
- This is the backbone of AI.

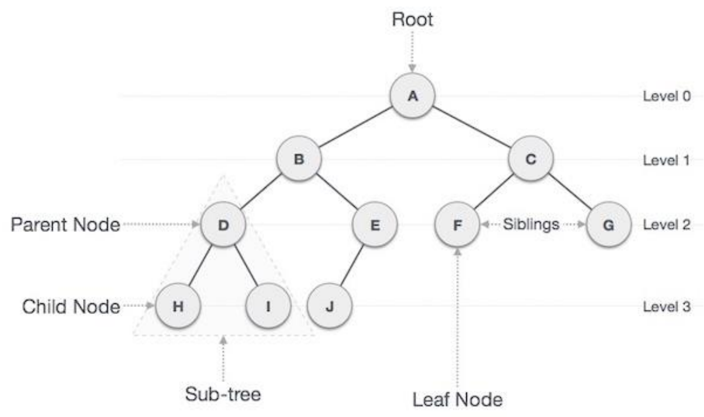
CHAPTER 6 TREE

Definition:

A tree is a nonlinear data structure in which items are arranged in a sorted Sequence, It is used to represent hierarchical relationship existing among several data items. Each node of a tree may or may not pointing more than one node.

It is a finite set of one or more data items (nodes) such that there is a special data item called the root. Its remaining data items are portioned into no. of mutually exclusive subsets, each of which is itself a tree and they are called subtree.

It represents the nodes connected by edges:



Tree terminology:

Root: - The first node in a hierarchical arrangement of data items is root.

Node: - Each data item in a tree is called a node.

Degree of a node: - It is the no. of subtrees of a node in a given tree from fig. Here,

- Degree of node A = 2
- Degree of node D = 2
- Degree of node F = 0

Degree of a tree: - It is the maximum degree of nodes in a given tree here degree of tree is 2.

Terminal node / Leaf node: - A node with degree 0 is terminal node.

Non terminal node / Parent Node: - Any node except the root whose degree is not zero is call non terminal node.

Keys: - it represents a value of a node based on which a search operation is to be carried out for a node

Siblings: - The children nodes of a given parent nodes are called siblings. F and G are siblings of parent node C & so on.

CHAPTER 6 TREE

Edge: - It is a connecting line of two nodes i.e. line drawn from one node to another.

Path: - It is a sequence of consecutive edge from the source node to the destination node.

Traversing: -It means passing through the nodes in certain order.

Level: - The entire tree structure is leveled in such a way that the root rode is always at level zero. Then its immediate children are at level 1 and there immediate children are at level 2 and so on up to the terminal nodes. Here, the level of a tree is 3.

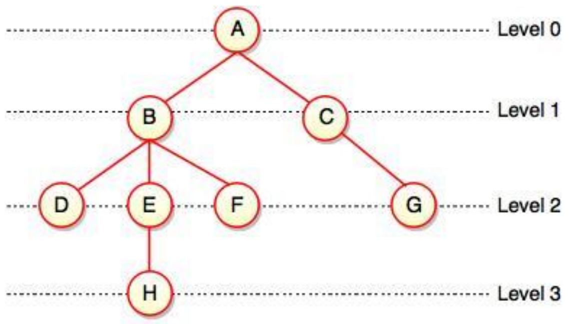
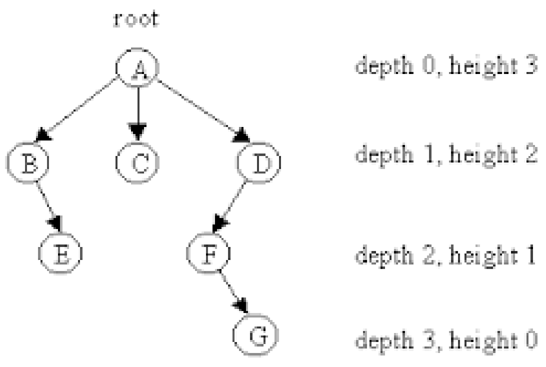


Fig. Levels of Tree

Depth or height: - Height of a node represents the number of edges on the longest path between that node and the leaf node. Depth of a node represents the number of edges from the tree's root node to the node. It is the maximum label of any node in a given tree here, the depth of a tree is 3.



A tree of height 3

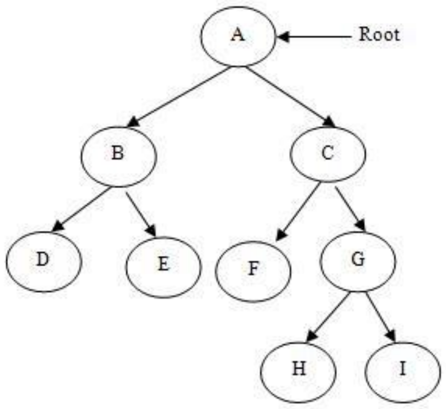
Binary tree:

A binary tree is a finite set of data items which is either empty or consist of a single item called the root and two disjoint binary trees the left subtree and right subtree. In binary tree the maximum degree of any node is at most 2 i.e. each node having 0, 1, or 2 degree node.

CHAPTER 6 TREE

Strictly Binary tree:

- Strictly binary tree is a tree where every node other than the leaves has the two children or, all of the nodes in a strictly binary tree are of degree zero or two, never degree one.
- A strictly binary tree with n leaves always contains 2n -1 nodes.

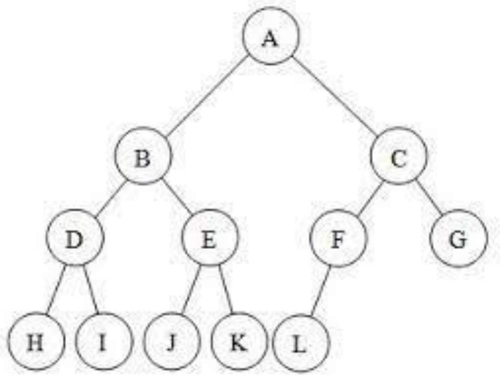


For 1 leaf no. of nodes =1
For 2 leaf no. of nodes =3
For 3 leaf no. of nodes =5
.
.
For n leaf no. of nodes =1+ (n-1)*2
=2n-1

$$\because The\ n^{th}\ term\ of\ AP\ is$$
$$= a + (n-1)d$$

Almost completely Binary Tree:

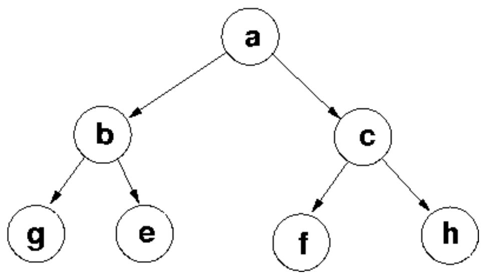
- A Binary Tree is almost complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible. A binary tree of depth d is an almost binary tree if:
  - Any node *'nd'* at level less than*'d-1'* has two sons.
  - For any node *'nd'* in the tree with a right descendant at level *d, nd* must have a left son and every left descendant of *nd* is either a leaf at level *d* or has two sons.



CHAPTER 6 TREE

Complete /Perfect Binary Tree:

- Every node except the leaf nodes have two children and every level (last level too) is completely filled. If ‘m’ nodes are present at level ‘k’ then there are ‘2m’ nodes at level ‘k+1’. There are 2^k nodes at level ‘k’.
- If ‘h’ be the height, then total no. of leaves $\mathbf{2^h}$, and total no. of nodes $\mathbf{2^{h+1}-1}$



Binary Tree Traversal:

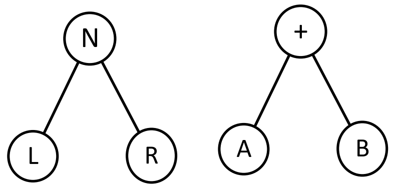
Traversal is a process to visit all the nodes of a tree and may print their values too. Unlike linear data structures (Array, Linked List, Queues, Stacks, etc.) which have only one logical way to traverse them, trees can be traversed in different ways. There are two types of traversal.

I. **Depth-First Traversal:**

Visit nodes in order of increasing depth. It starts with the root node and first visits all nodes of one branch as deep as possible of the chosen Node and before backtracking, it visits all other branches in a similar fashion. While traversing a tree root is denoted by ‘N’ left subtree as ‘L’ and right subtree as ‘R’. Following are the generally used ways for traversing trees.

- **In-order Traversal: LNR**
  1. Traverse the left subtree in Inorder
  2. Visit the root.
  3. Traverse the right subtree in Inorder

  E.g. A + B



Algorithm:

```
1. ptr=root
2. Inorder (ptr);
3. If (ptr != NULL)
      a. Inorder (ptr->left);
      b. Print “ ptr->info”;
      c. Inorder (ptr->right);
```

- **Pre-order Traversal: NLR**
  1. Visit the root.
  2. Traverse the left subtree in Preorder
  3. Traverse the right subtree in Preorder

  E.g. + AB

  Algorithm:

```
1. ptr=root
2. Preorder (ptr);
3. If (ptr != NULL)
   a. Print " ptr->info";
   b. Preorder (ptr->left);
   c. Preorder (ptr->right);
```

- **Post-order Traversal: LRN**
  1. Traverse the left subtree in Postorder
  2. Traverse the right subtree in Postorder
  3. Visit the root.

  E.g. AB+

  Algorithm:

```
1. ptr=root
2. Postorder (ptr);
3. If (ptr != NULL)
   a. Postorder (ptr->left);
   b. Postorder (ptr->right);
   c. Print " ptr->info";
```

CHAPTER 6 TREE

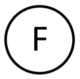
Constructing the tree from preorder and in order traversal:

- In pre order traversal, scan the nodes one by one and keep them inserting in tree.
- In order traversal, put the cross mark over the node which has been inserted.
- To insert a node in its proper position in the tree we look at that node the in order traversal & insert it according to its position with respect to the crossed nodes:-

Inorder: $EACKFHDBG$ (LNR)

Preorder: $FAEKCDHGB$ (NLR)

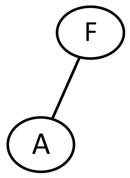
Insert F



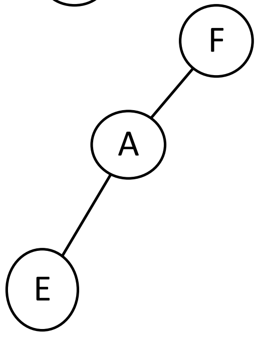
Inorder: EACK**F**HDBG (LNR)

Preorder: $FAEKCDHGB$ (NLR)

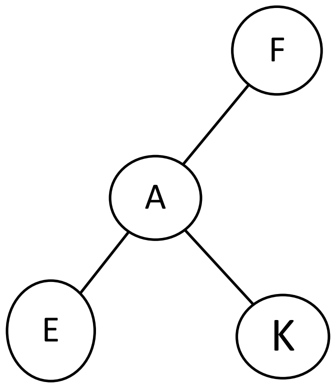
Insert A



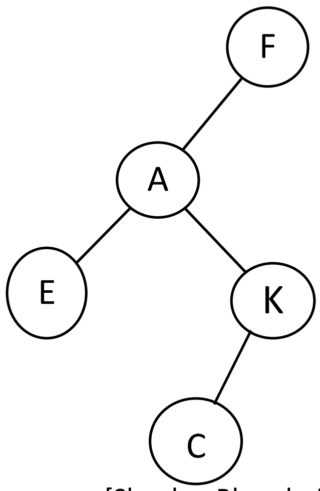
Insert E



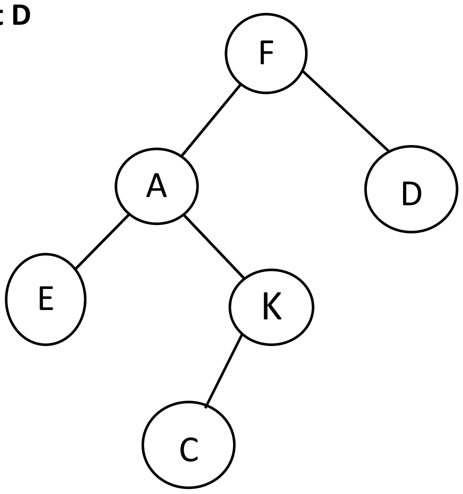
Insert K



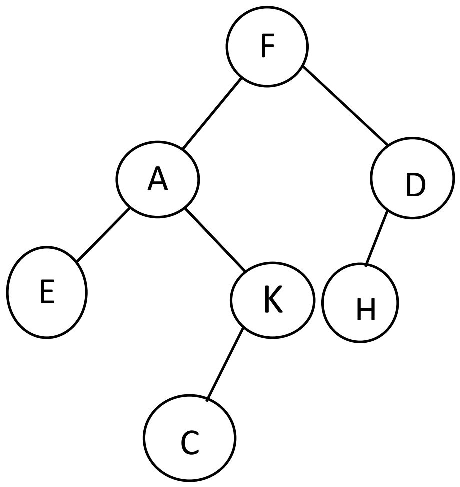
Insert C



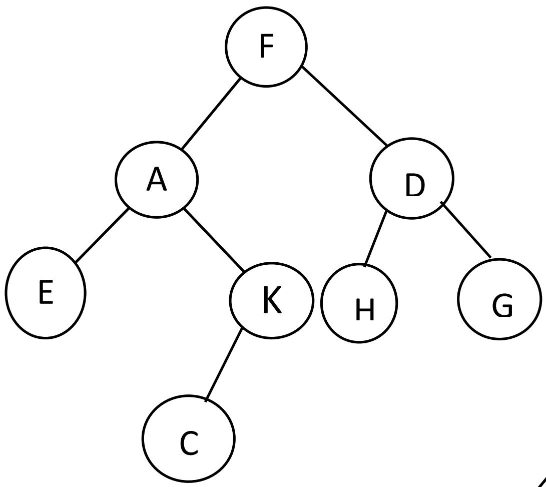
Insert D



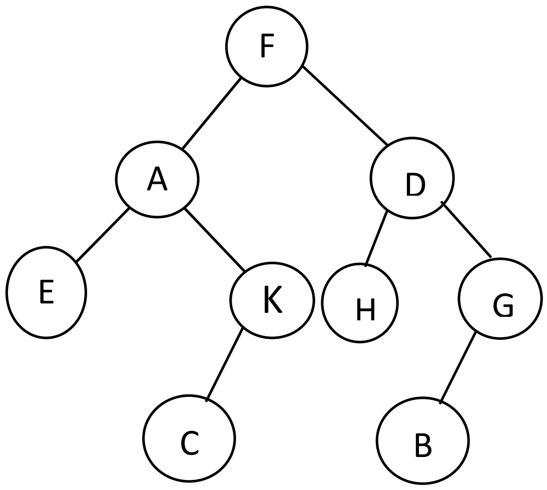
Insert H



Insert G



Insert B



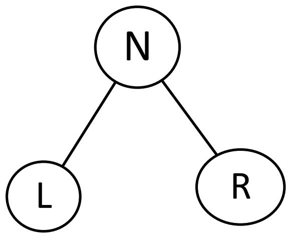
CHAPTER 6 TREE

Binary search tree: -

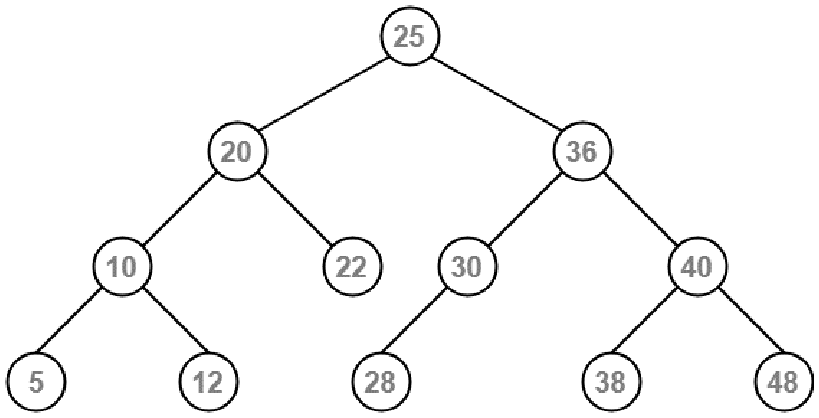
A binary search tree is a node-based binary tree in which each node has value greater than every node of left sub tree and less than every node of right sub tree which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

There must not be duplicate nodes and R > N > L



E.g.



Algorithm for inserting a new node in BST:

Insert function is used to add a new element in a binary search tree at appropriate location.

```
1. Allocate memory for node
       newnode = (struct node *) malloc(sizeof (struct node));
2. Set the data part to the value and set the left and right pointer of node, point to NULL.
       newnode -> info = data;
       newnode -> left = NULL;
       newnode -> right = NULL;
3. Assign pointer 'tree' as 'root' node of the tree.
       tree = root;
4. If (tree == NULL)
       Set tree = newnode;
5. Else if (data < tree -> info)
       i.   If(tree -> left == NULL) then
                tree -> left = newnode;
       ii.  else
                tree = tree -> left;
       iii. Repeat step from 4
6. Else if (data > tree -> info)
       i.   If ( tree -> right == NULL) then
                tree -> right = newnode;
       ii.  Else
                tree = tree -> right;
       iii. Repeat step from 4
7. Else if ( data == tree->info)
8. Display " Duplicate Data"
9. End
```

CHAPTER 6 TREE

Example 1:

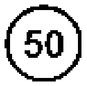
Construct a Binary Search Tree (BST) for the following sequence of numbers-

50, 70, 60, 20, 90, 10, 40, 100

When elements are given in a sequence,

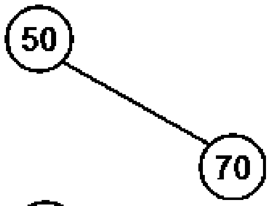
- Always consider the first element as the root node.
- Consider the given elements and insert them in the BST one by one.

Insert: 50



Insert: 70

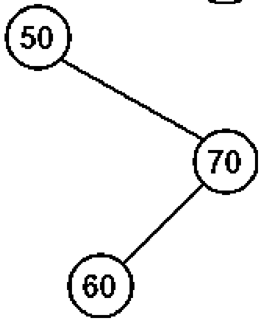
As 70 > 50, so insert 70 to the right of 50.



Insert: 60

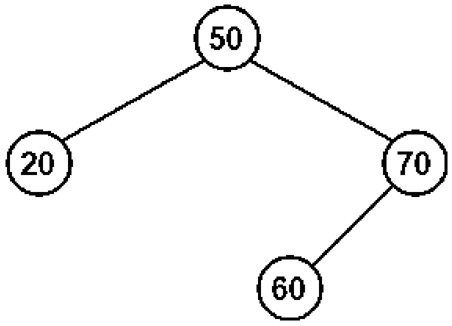
As 60 > 50, so insert 60 to the right of 50.

As 60 < 70, so insert 60 to the left of 70.



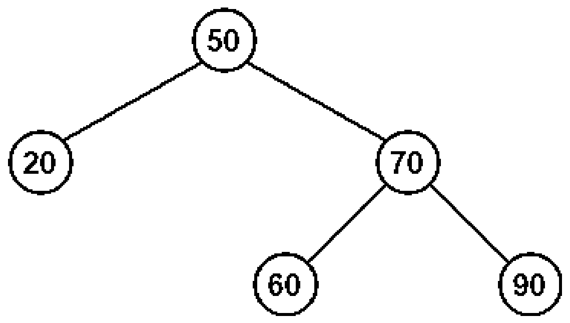
Insert: 20

As 20 < 50, so insert 20 to the left of 50.



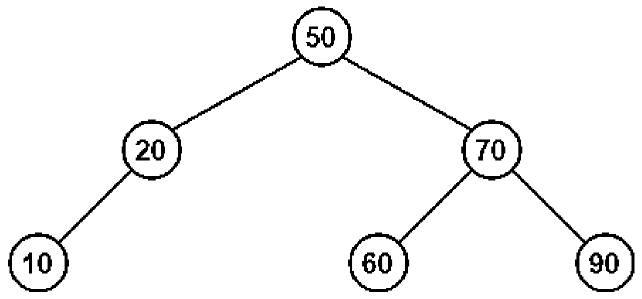
Insert: 90

As 90 > 50, so insert 90 to the right of 50.
As 90 > 70, so insert 90 to the right of 70.



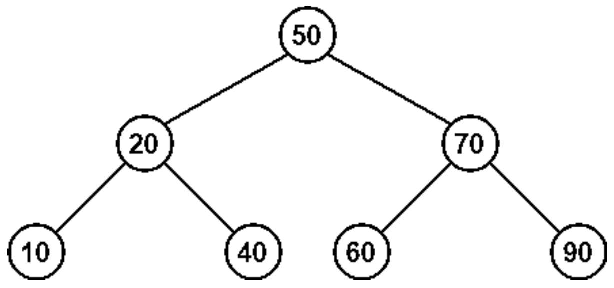
Insert: 10

As 10 < 50, so insert 10 to the left of 50.
As 10 < 20, so insert 10 to the left of 20.



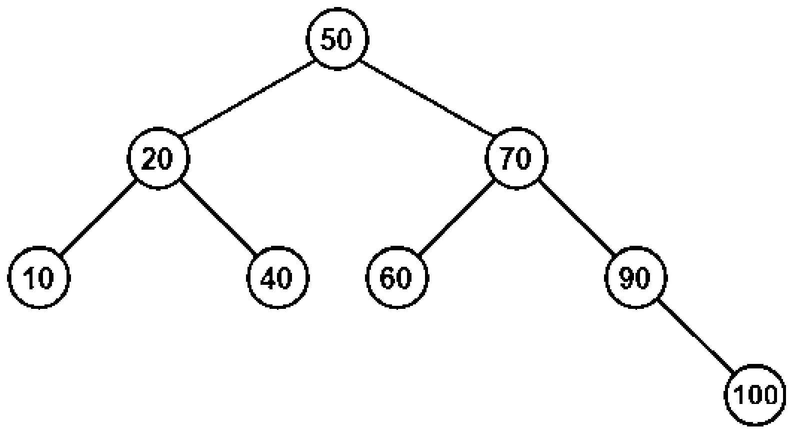
Insert: 40

As 40 < 50, so insert 40 to the left of 50.
As 40 > 20, so insert 40 to the right of 20.



Insert: 100

As 100 > 50, so insert 100 to the right of 50.
As 100 > 70, so insert 100 to the right of 70.
As 100 > 90, so insert 100 to the right of 90.

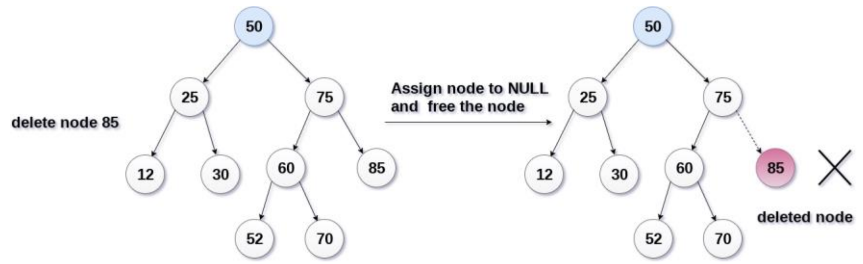


Deletion:

Delete function is used to delete the specified node from a binary search tree. However, we must delete a node from a binary search tree in such a way, that the property of binary search tree doesn't violate. There are three situations of deleting a node from binary search tree.

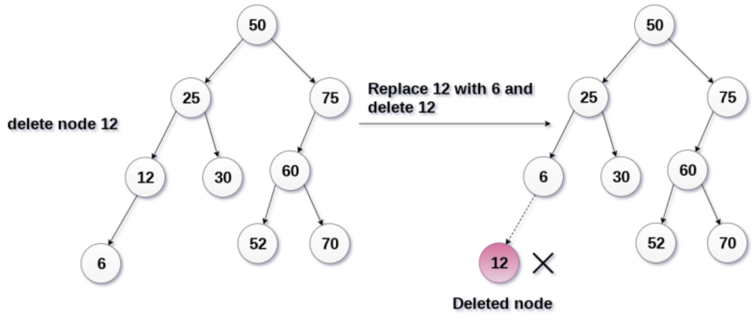
The node to be deleted is a leaf node:

It is the simplest case, in this case, replace the leaf node with the NULL and simple free the allocated space.



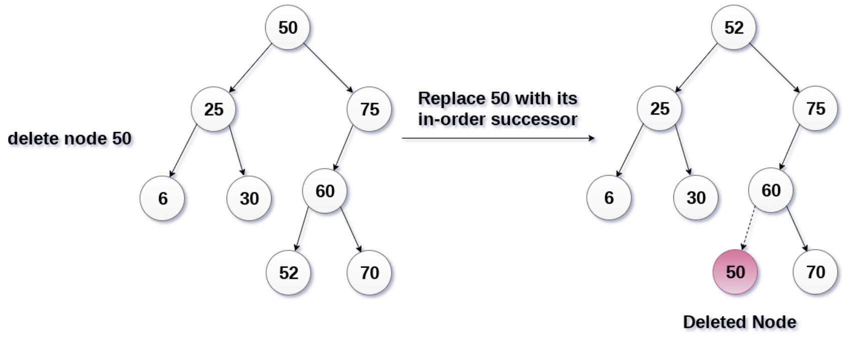
The node to be deleted has only one child:

In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.



The node to be deleted has two children:

The node which is to be deleted, is replaced with its in-order successor or predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated space.



Algorithm:

```
      For function deletenode (root, info)
1)    If (root == NULL)

                  Item not found in the tree and return NULL

2)    if ( data < root -> info )

                  Set root->left = deletenode (root->left, info);

3)    else if ( data  > tree-> info)

                  Set root->right =deletenode (root->right, info);

4)    else
                // for no child
                1.  if (root-> left == NULL && root -> right == NULL)
                        i.    Set free (root)
                        ii.   Return NULL
                // node with one child
                2.  Else if (root -> left == NULL) then
                        i.    ptr= root -> right
                        ii.   free (root)
                        iii.  return ptr;
                3.  else if (root -> right == NULL) then
                        i.    ptr = root -> left
                        ii.   free (root)
                        iii.  return ptr
                // node with two children
                4.  Get the Inorder successor (smallest in the right subtree)
                        i.    Ptr= root -> right;
                              // to find leftmost leaf
                        ii.   While (ptr -> left != NULL)
                              ptr = ptr -> left;
                    // copy the Inorder successor's content to this node
                5.  root -> info = ptr -> info
                6.  root -> right = deletenode ( root -> right, ptr -> key);
5)    Return root and End
```

CHAPTER 6 TREE

AVL balanced trees and balancing algorithm:

AVL tree is a self-balancing binary search tree in which the difference of heights of left and right subtrees of any node is less than or equal to one which is termed as balance factor. The technique of balancing the height of binary trees was developed by Adelson, Velskii, and Landi and hence given the short form as AVL tree or Balanced Binary Tree. Every sub-tree is itself an AVL tree. If the difference is more than one, then the tree is rebalanced by applying certain rule of rotation.

Balance Factor (BF) = $H_L - H_R$

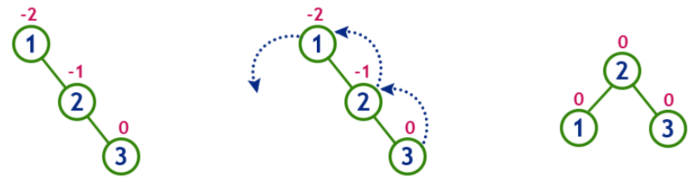
And for AVL tree, $\mathbf{| H_L - H_R | \leq 1}$

If BF is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations:

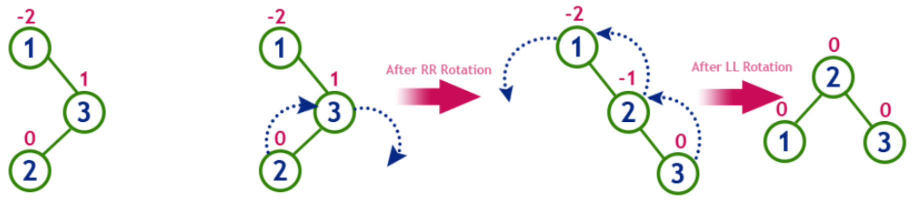
1. Right-Right Rotation:

   - If BF of node is -2 and the BF of the right child id < 0. A single 'left' rotation



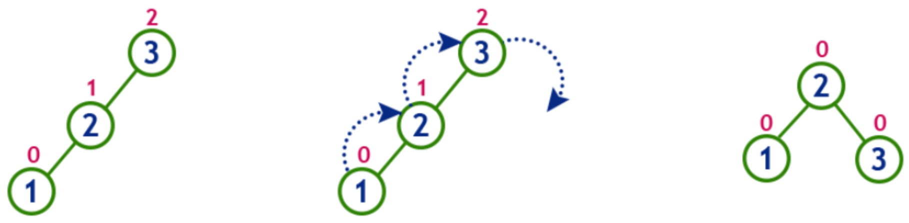
2. Right-Left rotation:

   - If BF of node is -2 and the BF of the right child id > 0 A 'right rotation followed by a 'left' rotation



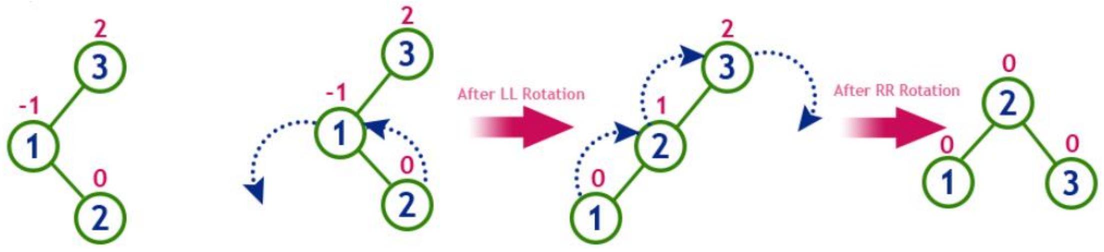
3. Left-Left rotation:

   - If BF of node is 2 and the BF of the left child id > 0. A single 'right' rotation



4. Left-Right rotation:

- If BF of node is 2 and the BF of the left child id < 0. A 'left rotation followed by a 'right' rotation

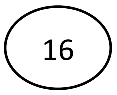


Example 1:

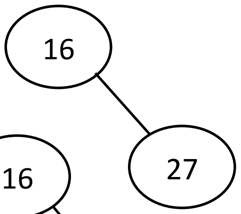
Construct AVL tree using following sequence of data:

$$16, 27, 9, 11, 36, 54, 81, 63$$

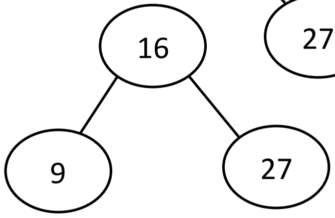
Insert 16



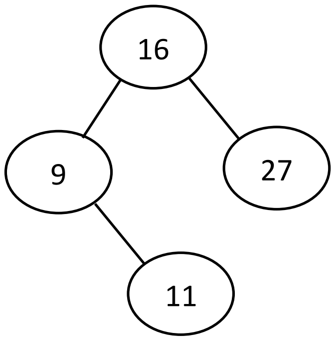
Insert 27



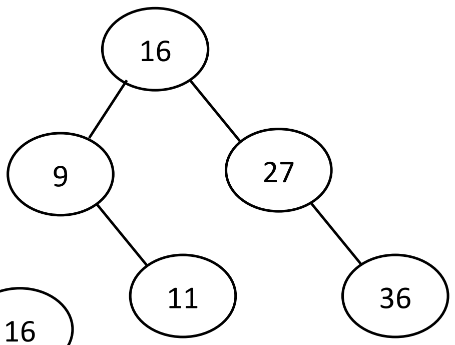
Insert 9



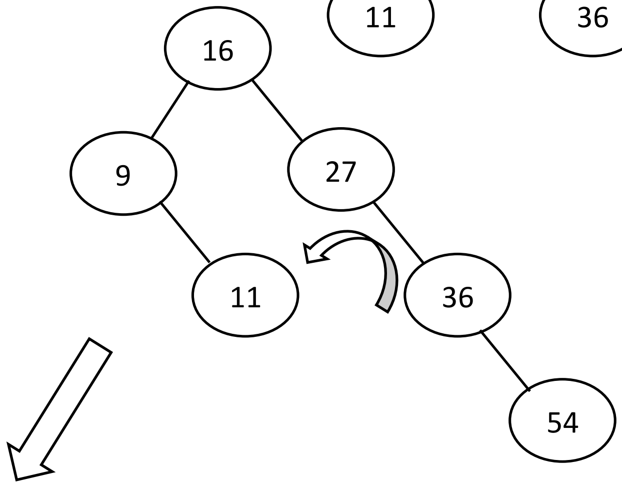
Insert 11

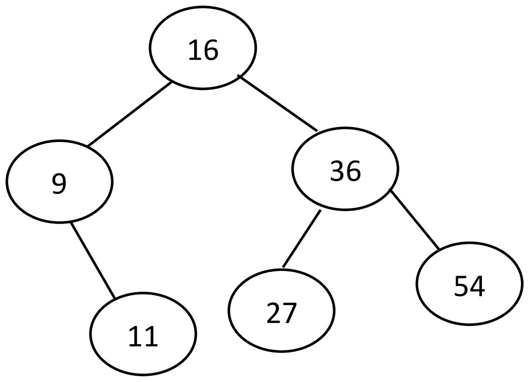


Insert 36

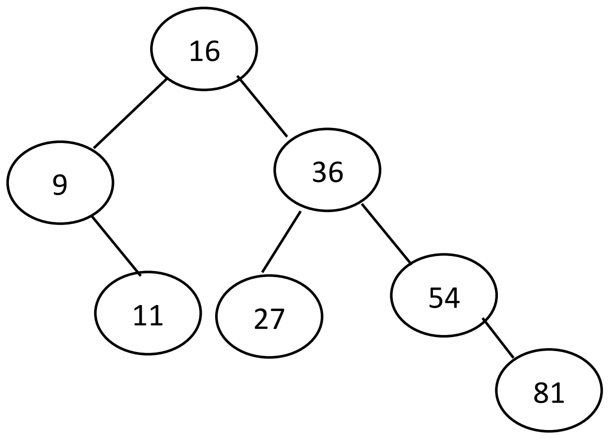


Insert 54

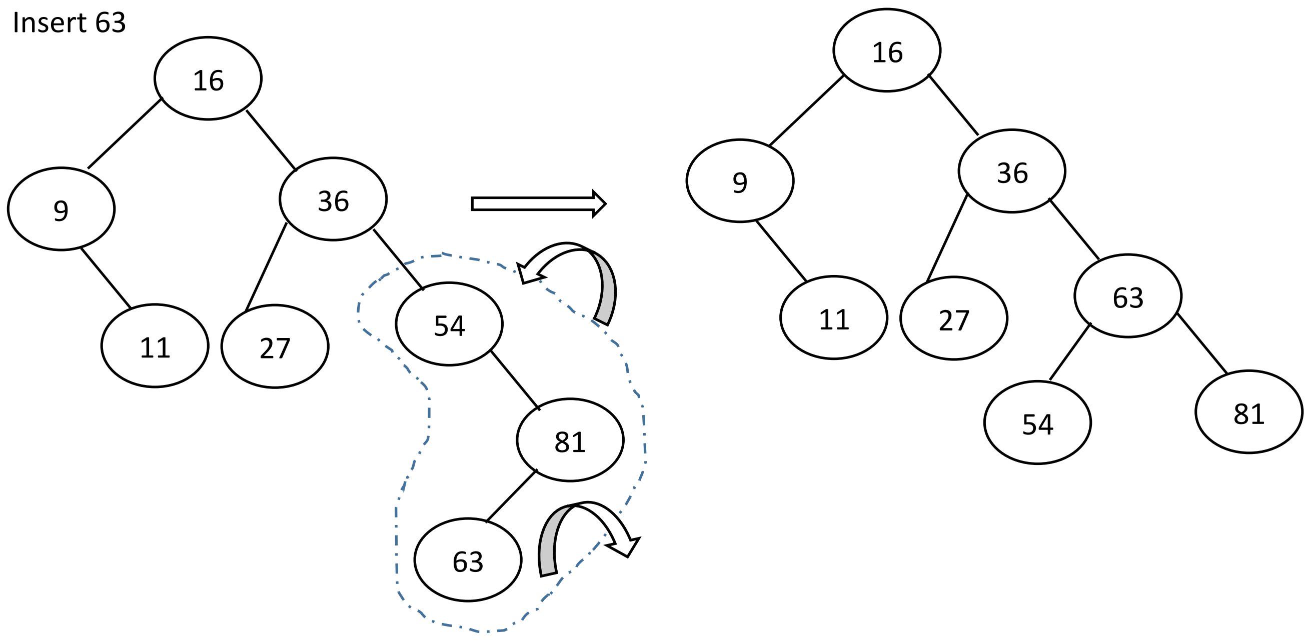




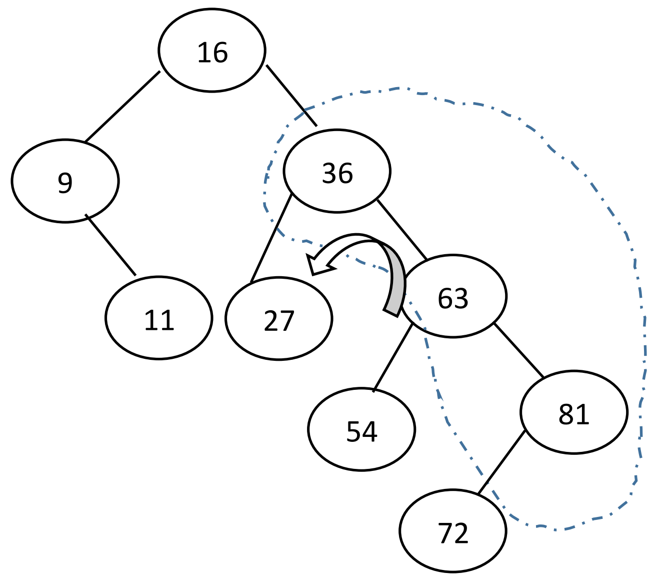
Insert 81

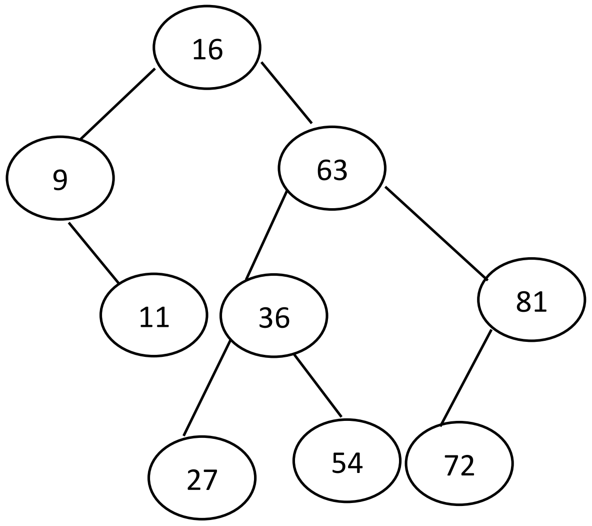


Insert 63



Insert 72



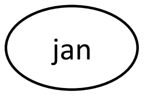


Example 2:

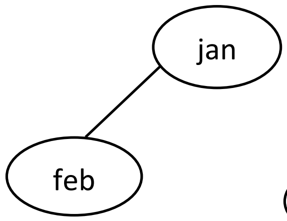
Create the AVL tree for the following data:

jan, feb, mar, apr, may, jun, jul, aug, sep, oct, noc, dec.

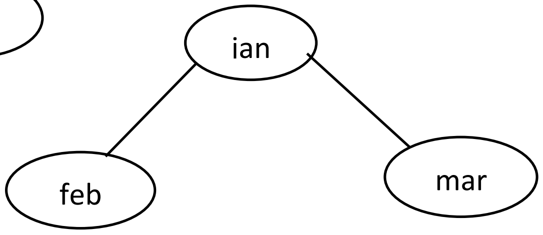
Insert: jan



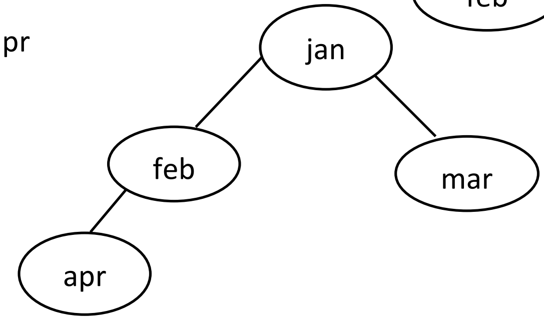
Insert: feb



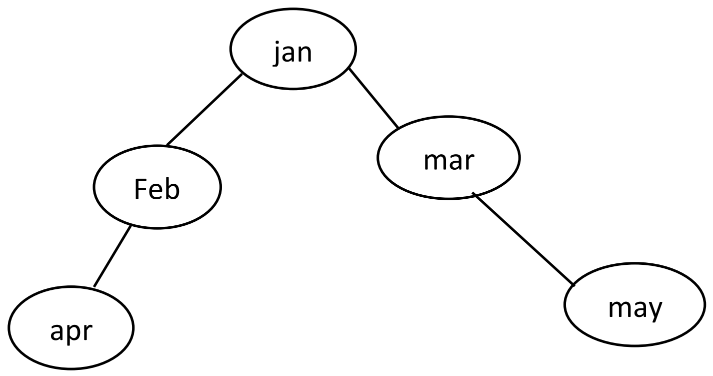
Insert: mar



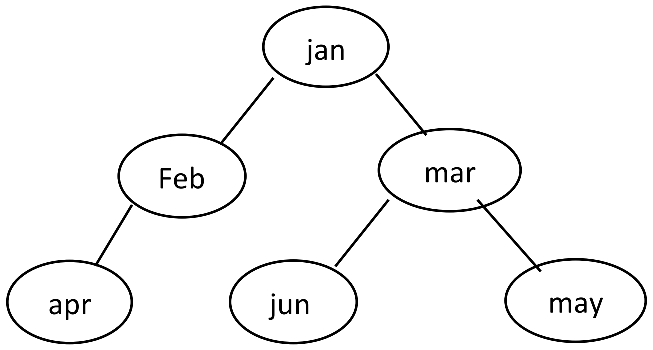
Insert: apr



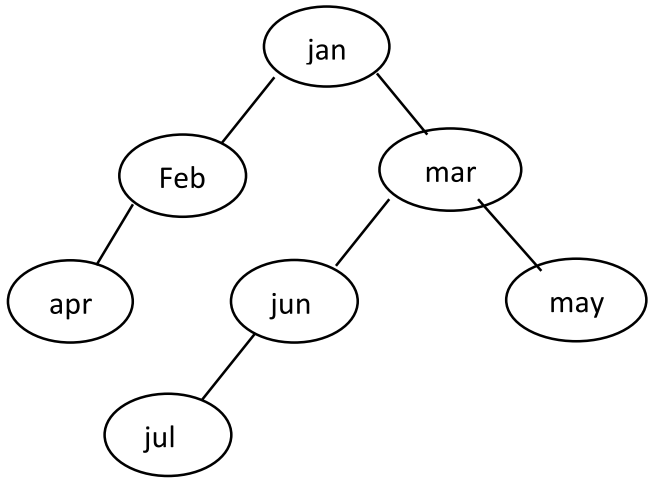
Insert: may



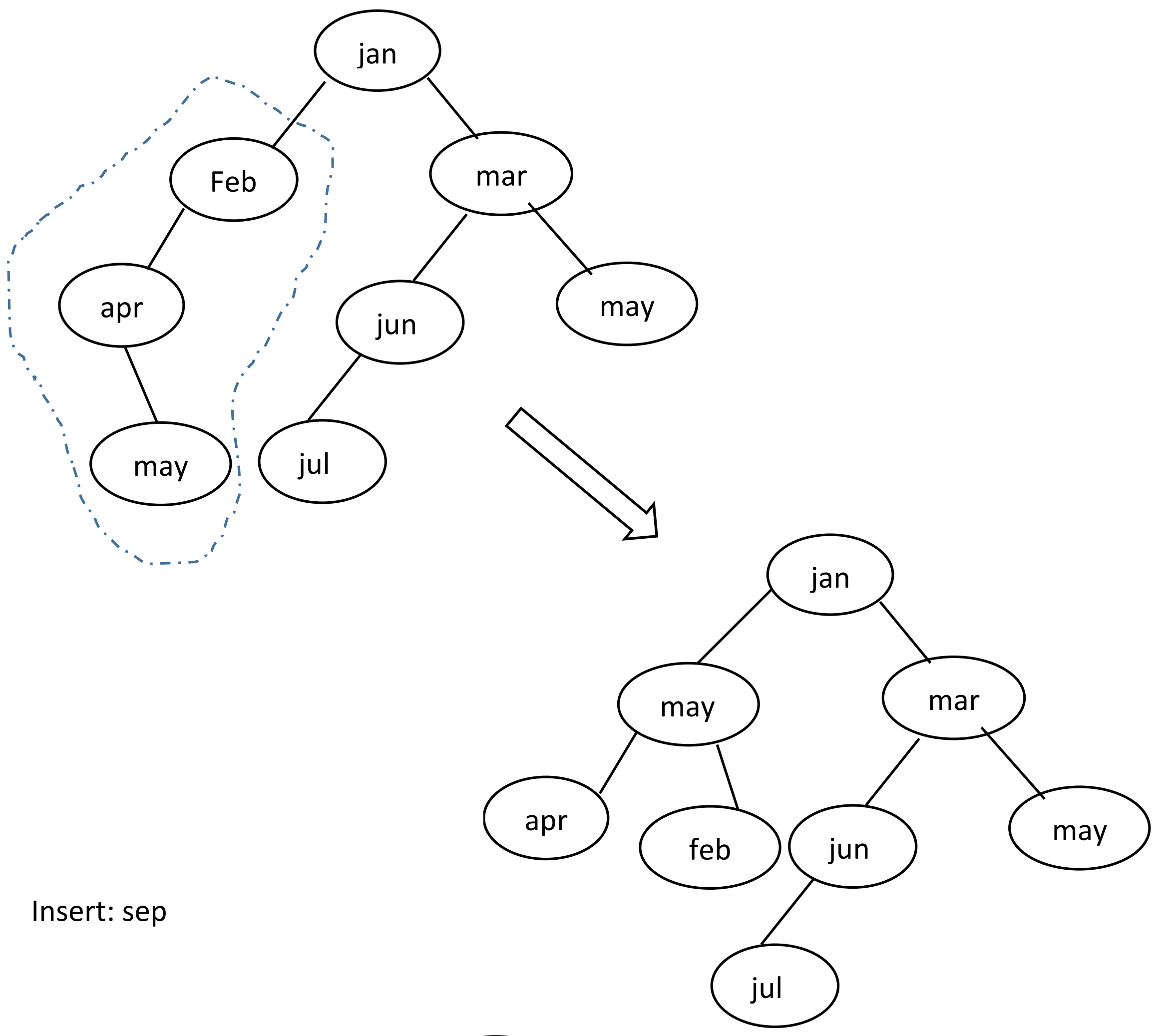
Insert: jun



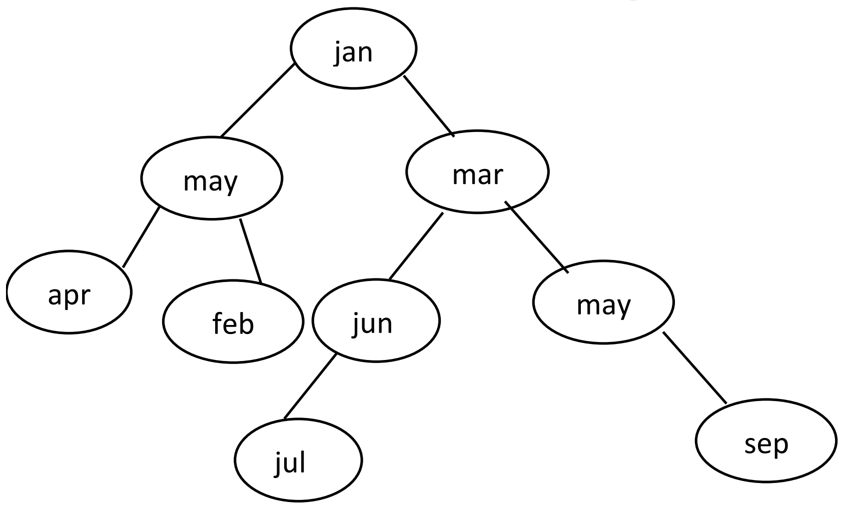
Insert: jul



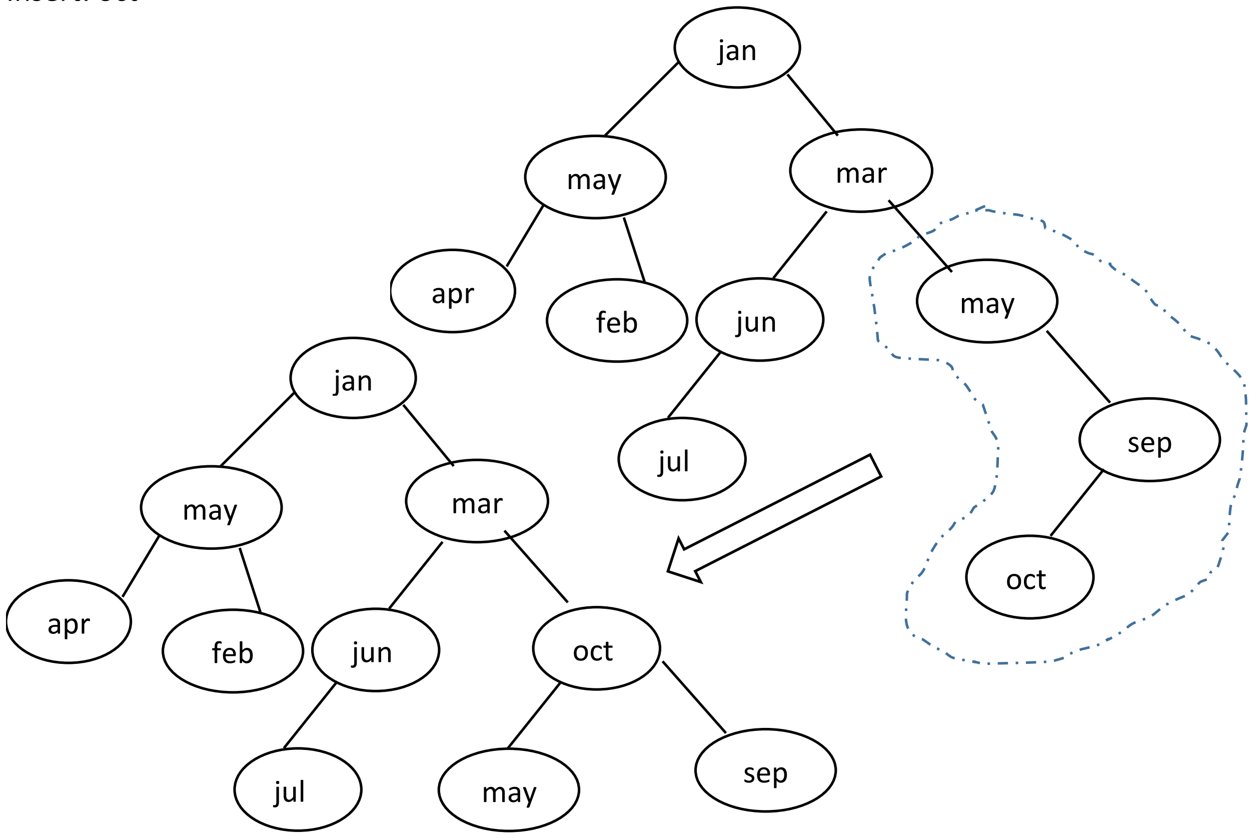
Insert: aug



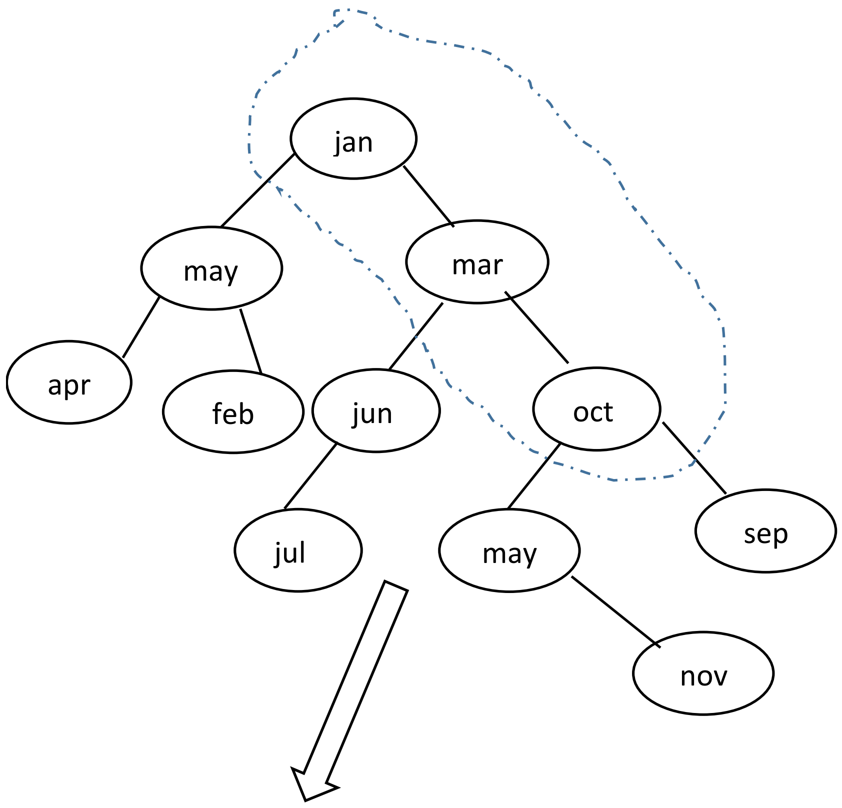
Insert: sep

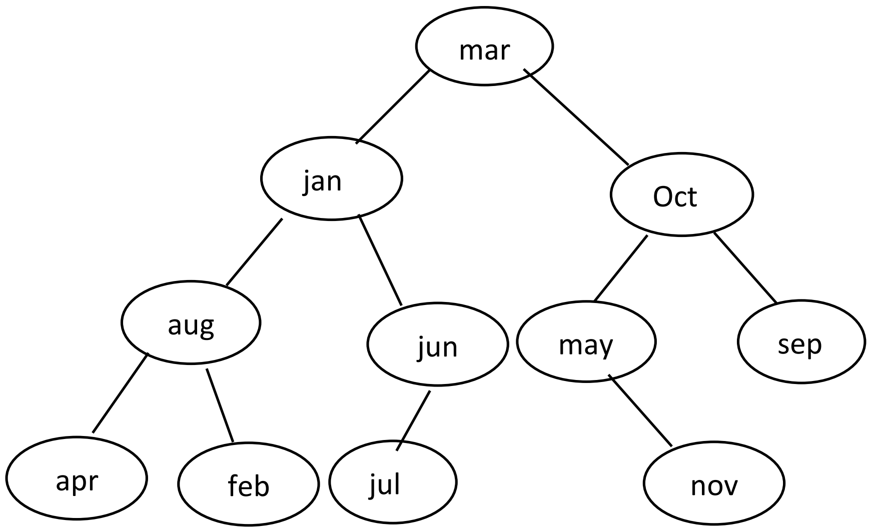


Insert: oct

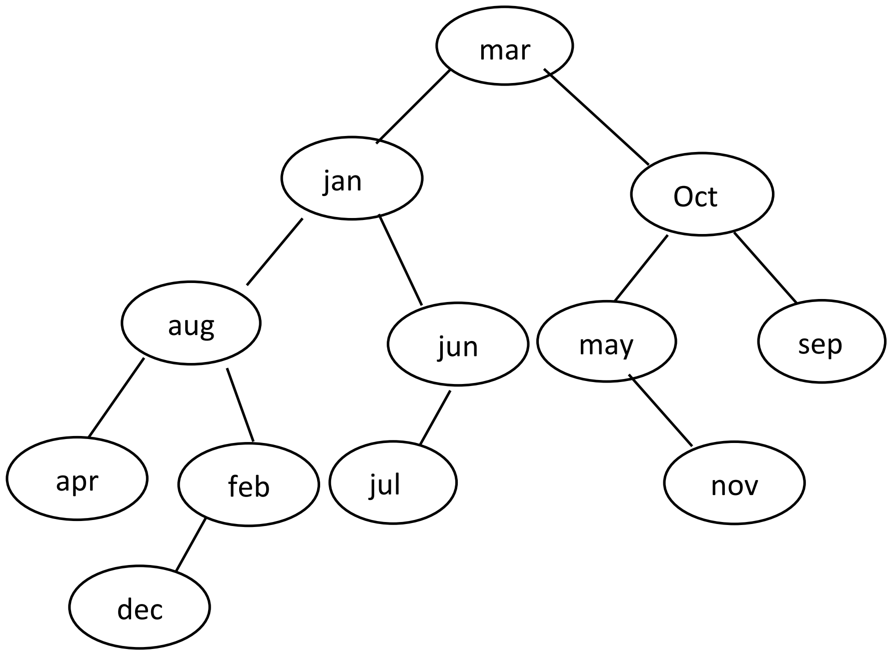


Insert: nov




mar
jan
Oct
aug
jun
may
sep
apr
feb
jul
nov

Insert: dec


mar
jan
Oct
aug
jun
may
sep
apr
feb
jul
nov
dec

CHAPTER 6 TREE

The Huffman algorithm:

Huffman Coding is a technique of compressing data to reduce its size without losing any of the details. It was first developed by David Huffman in 1951. Huffman Coding is generally useful to compress the data in which there are frequently occurring characters.

Properties:

- Set of symbols to be transmitted or stored along with their frequencies/ probabilities/ weights and a tree-like data structure with minimum weighted path length from root is formed which can be used for generating the binary codes.
- It is a famous algorithm used for lossless data encoding.
- It uses variable-length encoding scheme for assigning binary codes to characters depending on how frequently they occur in the given text. The character that occurs most frequently is assigned the smallest code and the one that occurs least frequently gets the largest code.
- Generally, bit ‘0’ represents the left child and bit ‘1’ represents the right child.

Algorithm:

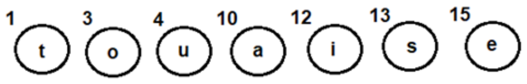
1. Create a leaf node for each character and build a min heap using all the nodes (The frequency value is used to compare two nodes in min heap)
2. Repeat Steps 3 to 5 while heap has more than one node
3. Extract two nodes, say x and y, with minimum frequency from the heap
4. Create a new internal node z with x as its left child and y as its right child. Also **frequency(z)= frequency(x)+frequency(y)**
5. Add z to min heap
6. Last node in the heap is the root of Huffman tree

Example 1:

Create Huffman Tree for the following characters along with their frequencies.

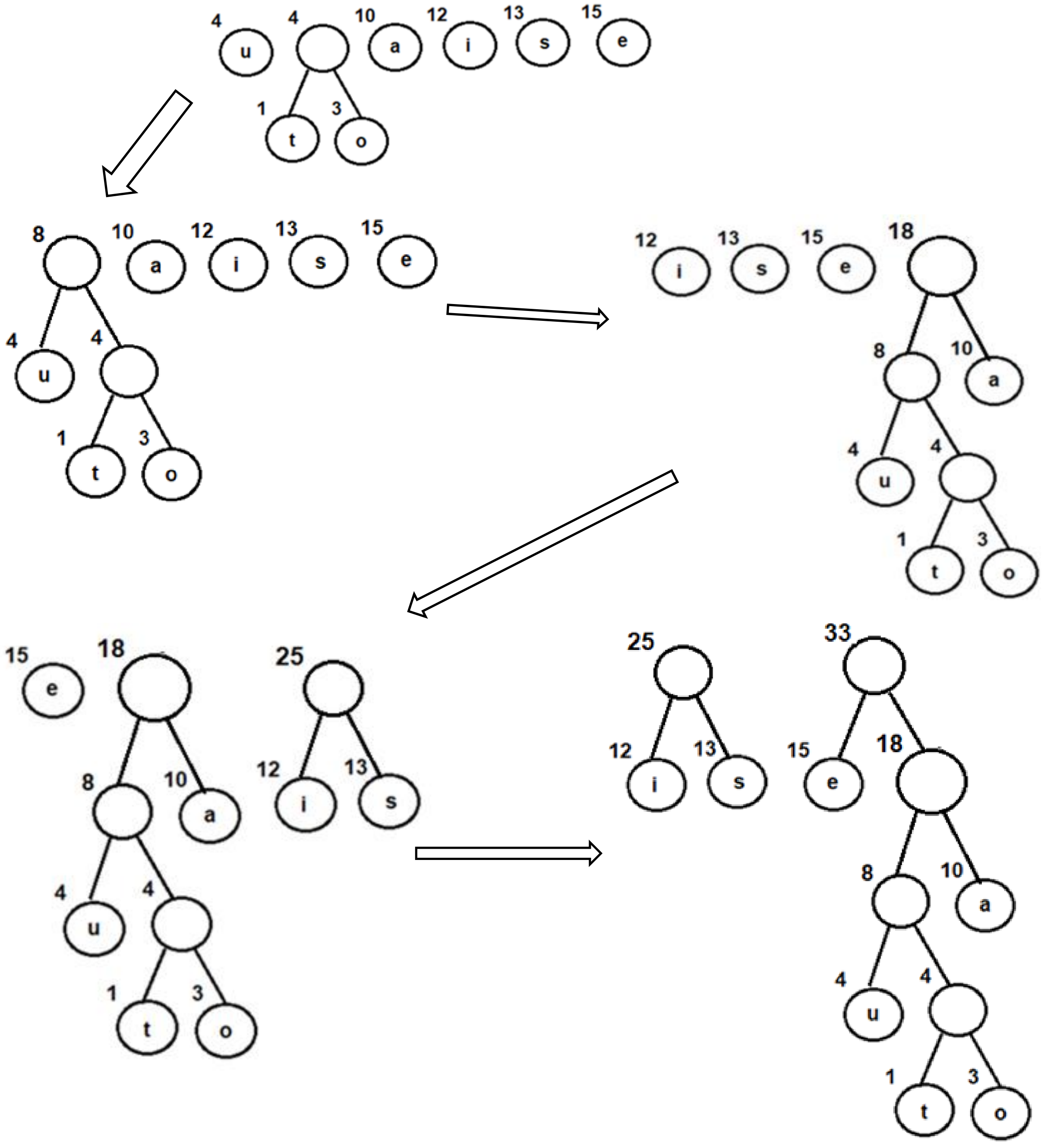
Characters	Frequencies
a	10
e	15
i	12
o	3
u	4
s	13
t	1

Create leaf nodes for all the characters and add them to the min heap.

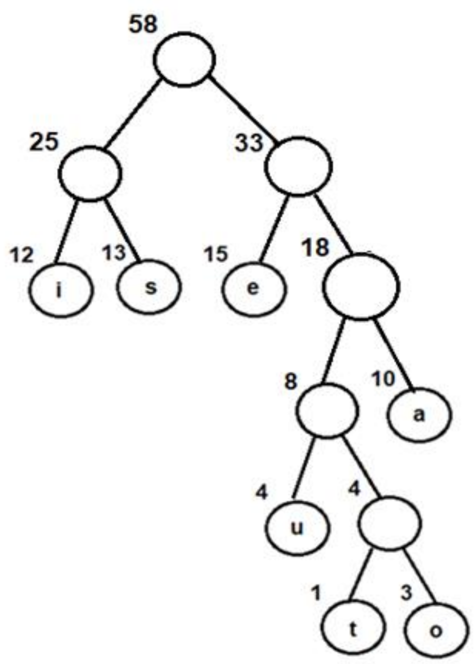


Extract two nodes, with minimum frequency from the heap and create a new internal node by adding.

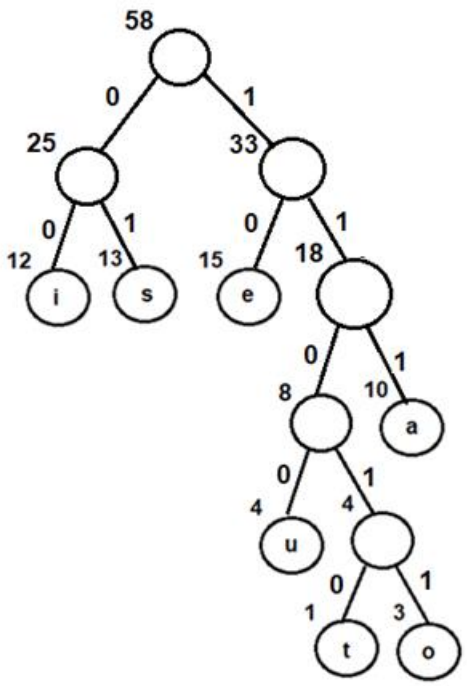
Add the new internal node to the min heap.



Last node in the heap is the root of Huffman tree.



Traverse the tree starting from root node. Add 0 to array while traversing the left child and add 1 to array while traversing the right child. Assigning binary codes to Huffman tree



We get prefix-free and variable-length binary codes with minimum expected code word length.

Characters	Binary Codes
i	00
s	01
e	10
u	1100
t	11010
o	11011
a	111

CHAPTER 6 TREE

B-Tree: "Balanced Tree"

- It is also known as balanced sorted tree.
- The height of the tree must be kept to a minimum.
- The leaves of the tree must all be the same level.
- The root has at least two subtree unless it is the only node in the tree.
- All nodes except the leaves must have at least some minimum number of children.
- Every node has maximum m children, if a B- tree has the order m.
- Every node has maximum (m-1) keys.
- Min Children:
    - For leaf: 0
    - For root: 2
    - Internal nodes: [m/2]
- Min keys:
    - Root node: 1
    - All other nodes: [m/2]-1

Insertion:

Example 1:

Consider another example for B-Tree of order 5

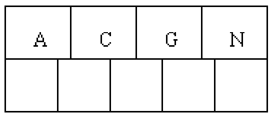
C N G A H E K Q M F W L T Z D P R X Y S

Order 5 means that a node can have a maximum of 5 children and 4 keys

All nodes other than the root must have a minimum of 2 keys

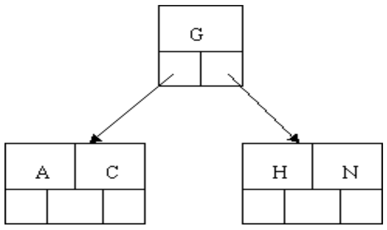
Step 1:

The first 4 letters get inserted into the same node



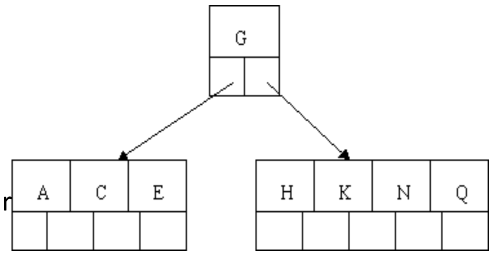
Step 2:

When we try to insert the H, we find no room in this node, so we split it into 2 nodes, moving the median item G up into a new root node



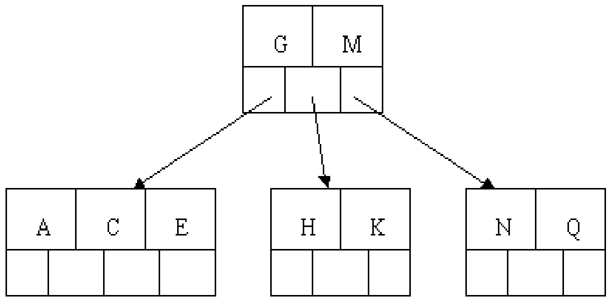
Step 3:

Inserting E, K, and Q proceeds without requiring any splits



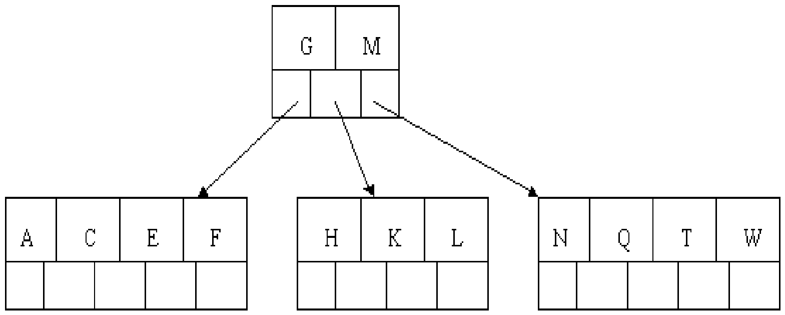
Step 4:

Inserting M requires a split. Note that M happens to be the median key and so is moved up into the parent node.



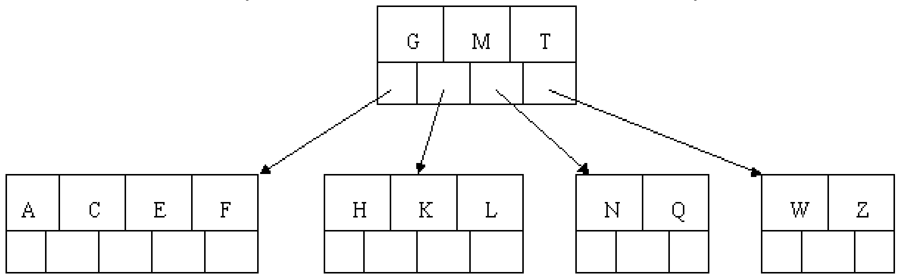
Step 5:

The letters F, W, L, and T are then added without needing any split.



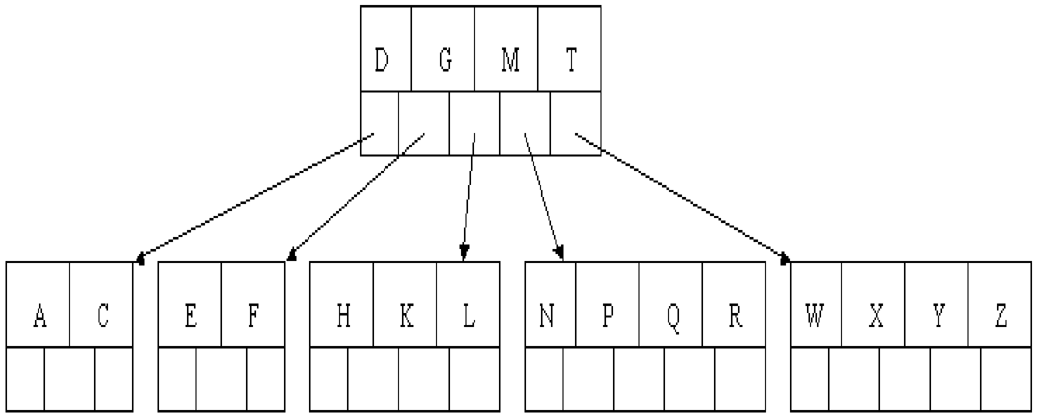
Step 6:

When Z is added, the rightmost leaf must be split. The median item T is moved up into the parent node.



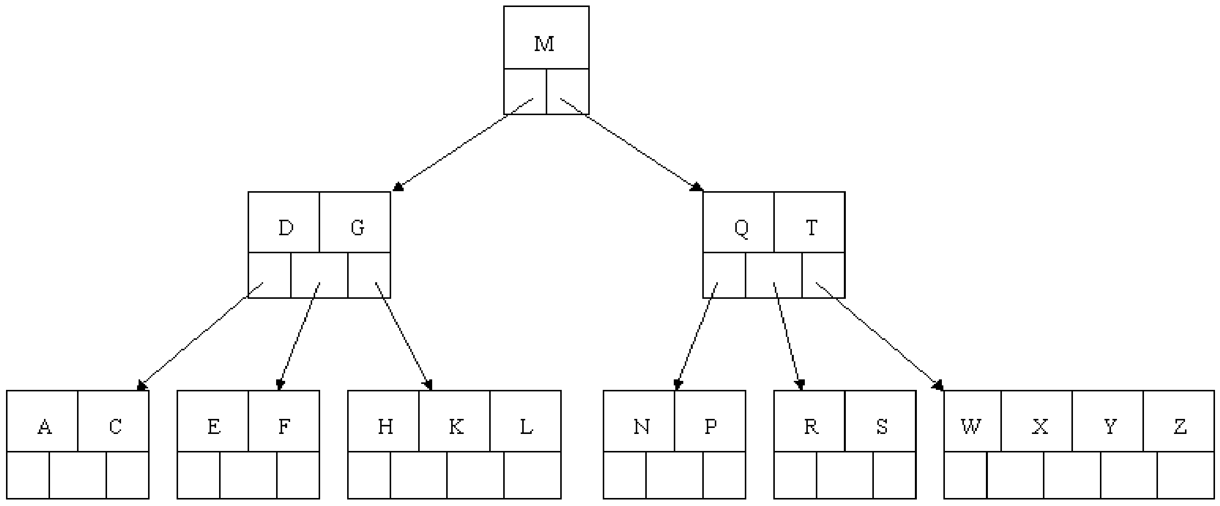
Step 7:

The insertion of D causes the leftmost leaf to be split. D happens to be the median key and so is the one moved up into the parent node. The letters P, R, X, and Y are then added without any need of splitting.



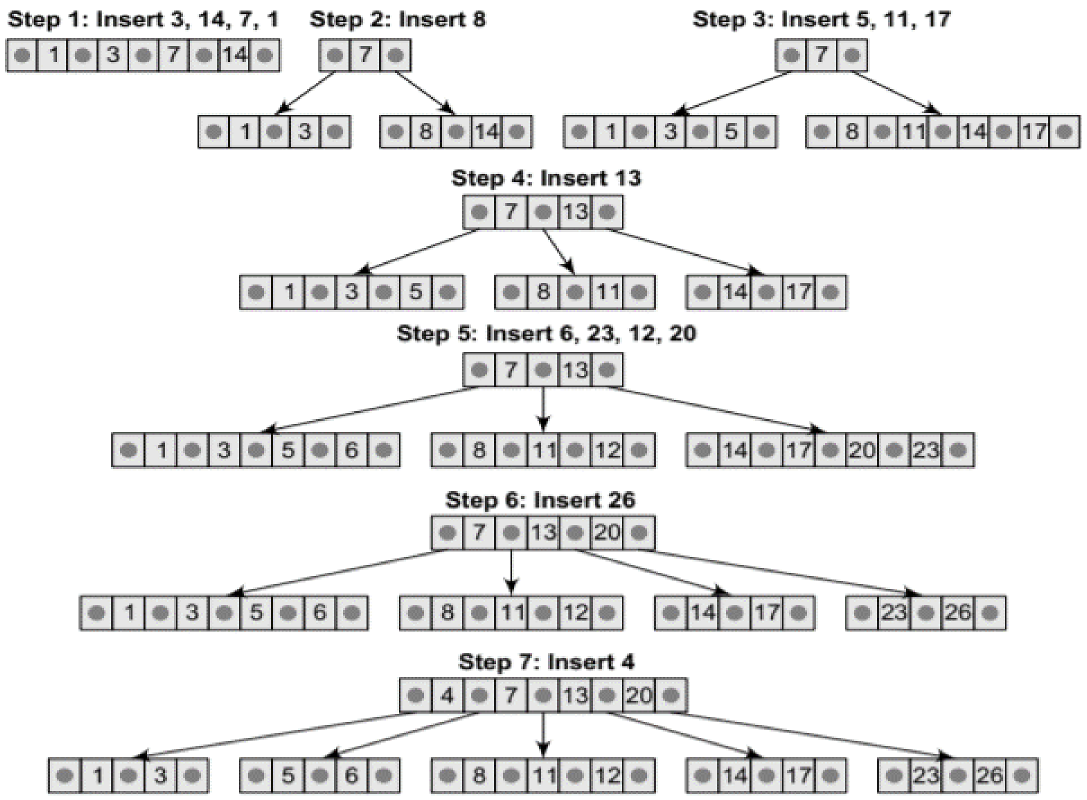
Step 8:

- Finally, when S is added, the node with N, P, Q, and R splits, sending the median Q up to the parent. However, the parent node is full, so it splits, sending the median M up to form a new root node. Final B-Tree look like

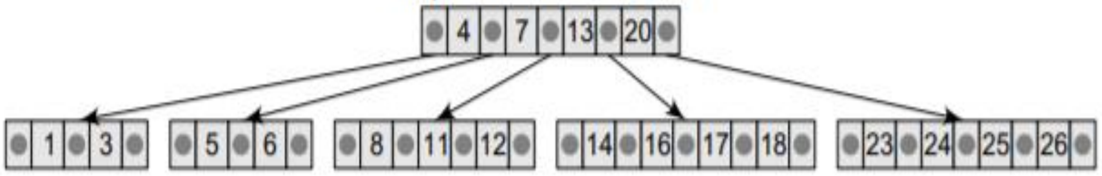


Example 2. Create a B tree of order 5 by inserting the following elements:

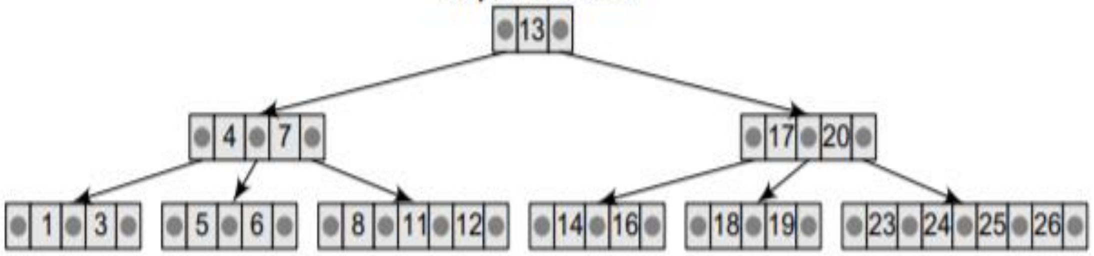
$3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, and\ 19.$



Step 8: Insert 16, 18, 24, 25



Step 9: Insert 19



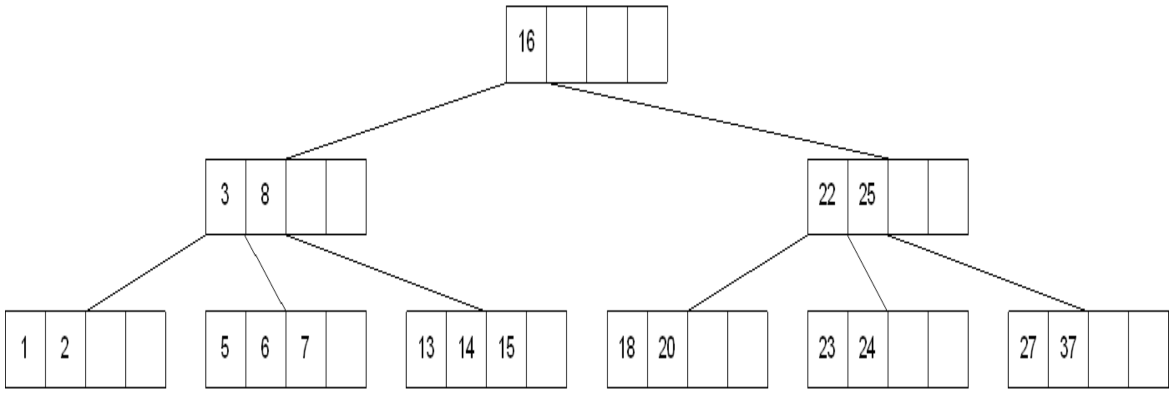
Deletion:

There are two main cases to be considered:

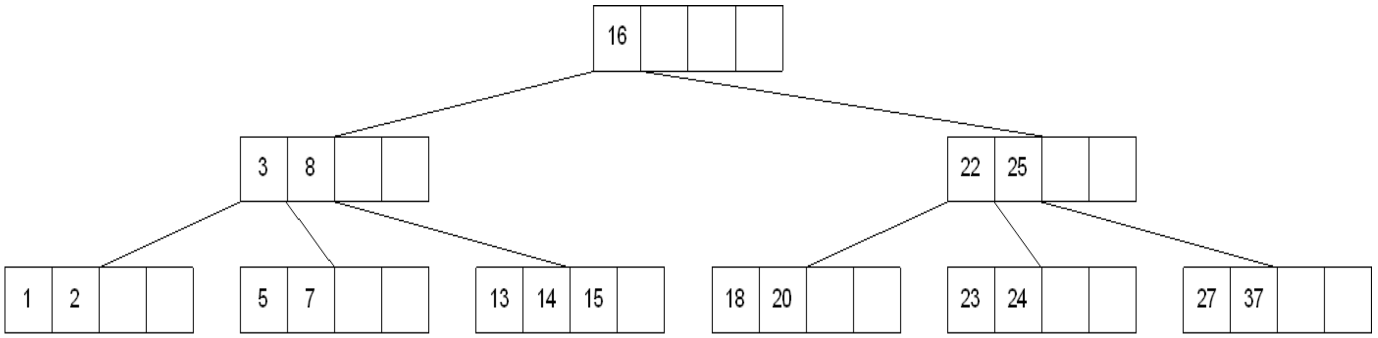
- Deletion from a leaf
- Deletion from a non-leaf

Case 1: Deletion from a leaf

a. If the leaf has at least (m-1)/2 data after deleting the desired value, the remaining larger values are moved to "fill the gap".

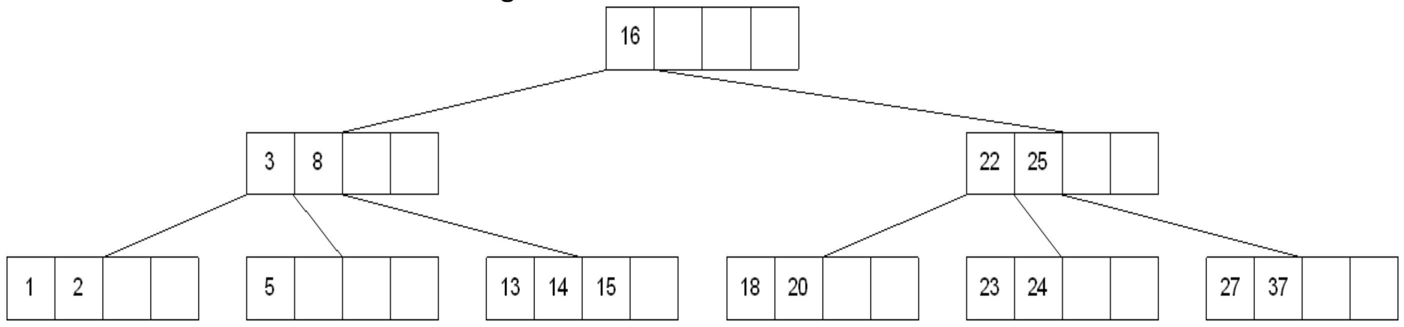


Deleting 6 from the above tree



b. If the leaf is less than (m-1)/2 after deleting the desired value (known as underflow), two things could happen

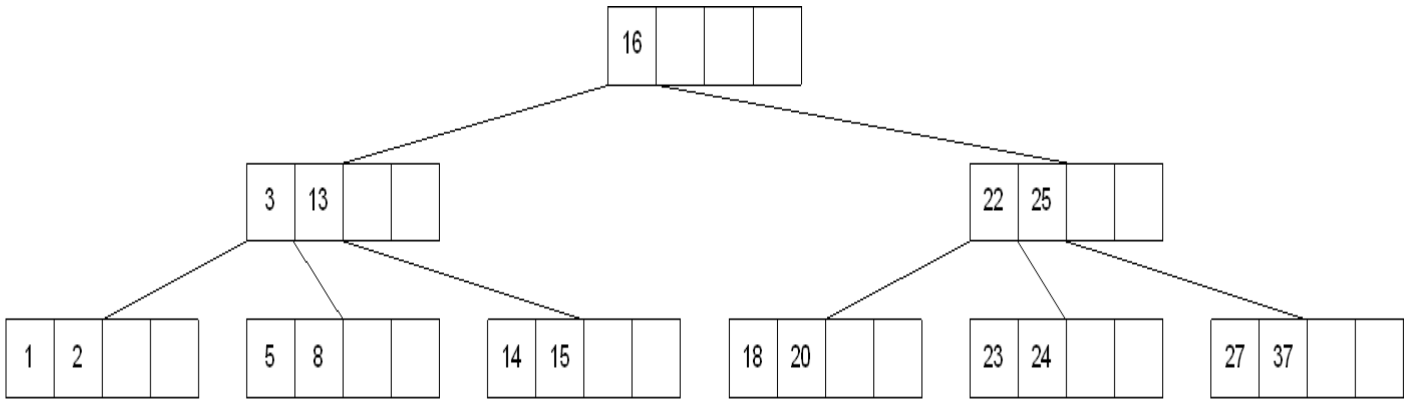
Deleting 7 from the tree above results in



i) If there is a left or right sibling with the number of keys exceeding the minimum requirement,

a) all of the keys from the leaf and sibling will be redistributed between them by moving the separator key from the parent to the leaf and

b) moving the middle key from the node and the sibling combined to the parent



ii) If the number of keys in the sibling does not exceed the minimum requirement,

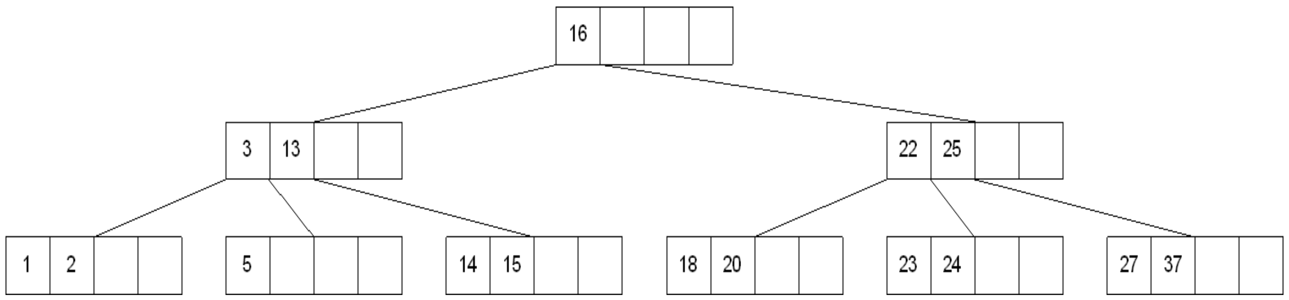
a) The leaf and sibling are merged by putting the keys from the leaf, the sibling, and the separator from the parent into the leaf.

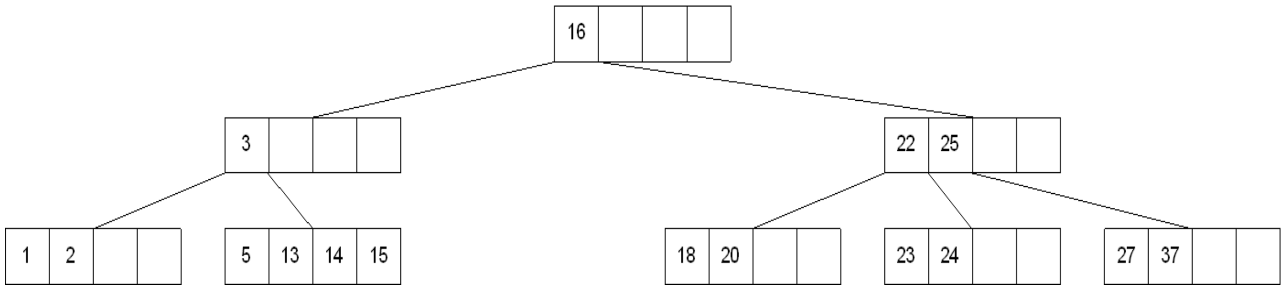
b) The sibling node is discarded and the keys in the parent are moved to "fill the gap".

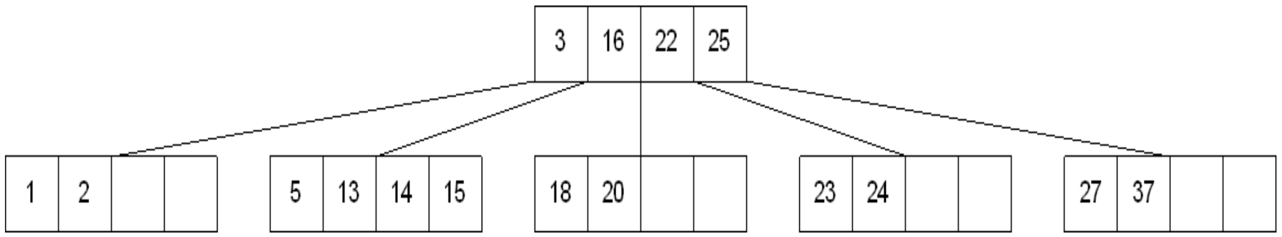
c) If the parent itself is underflow, treat the parent as a leaf and continue repeating from step a) until

The minimum requirement is met or the root of the tree is reached.

Deleting 8 from the tree above results in



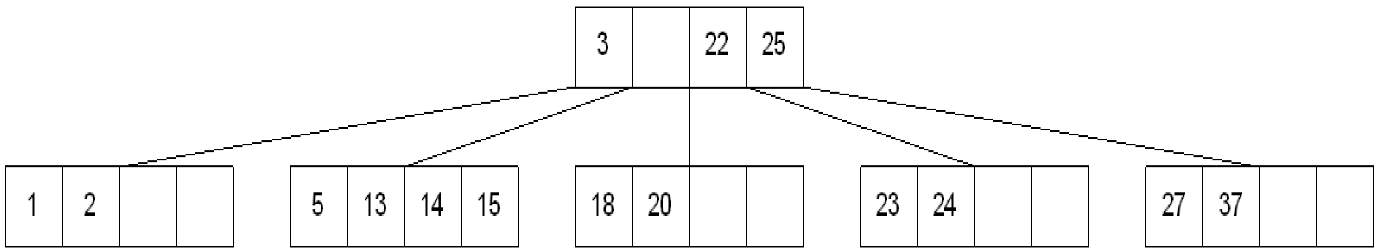




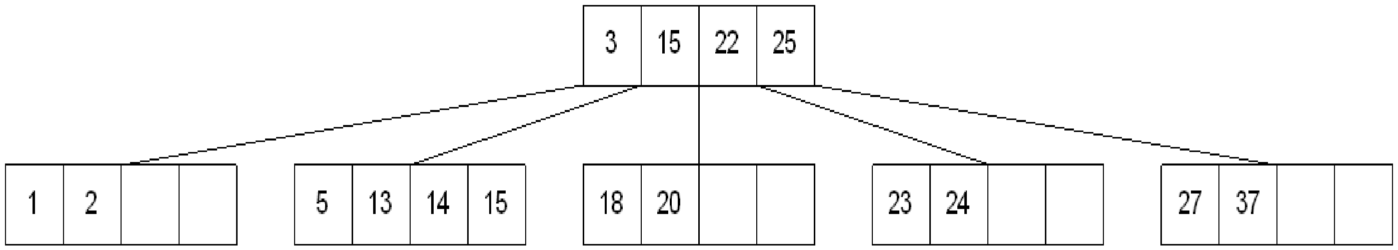
Case 2: Deletion from a non-leaf node

a. The key to be deleted will be replaced by its immediate predecessor (or successor)
b. Then the predecessor (or successor) will be deleted since it can only be found in a leaf node.

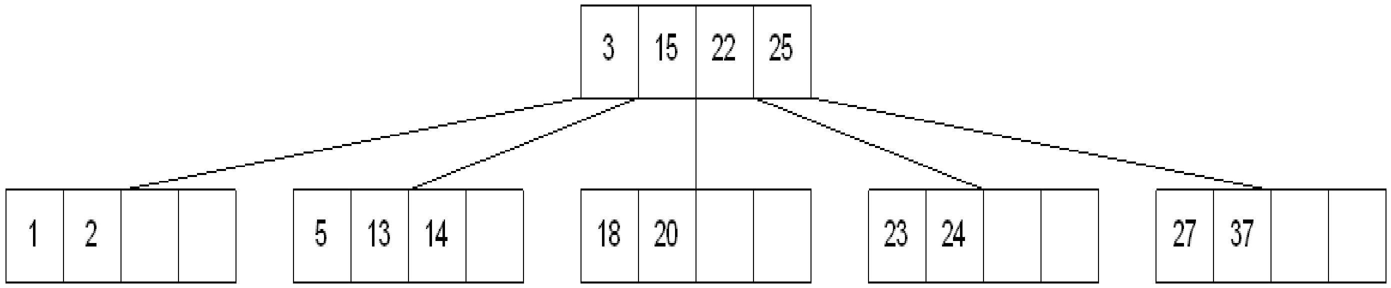
Deleting 16 from the tree above results in



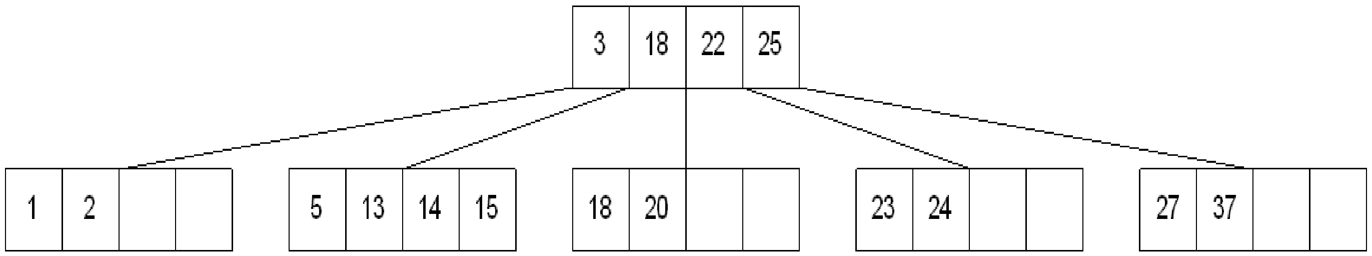
The "gap" is filled in with the immediate predecessor



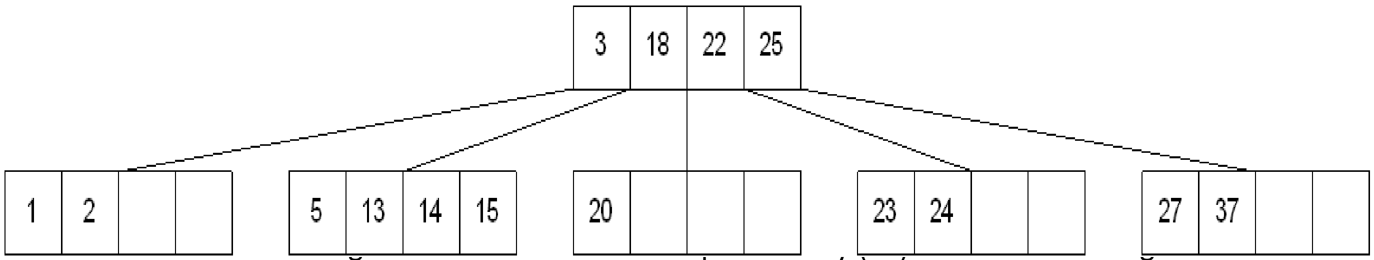
And then the immediate predecessor is deleted



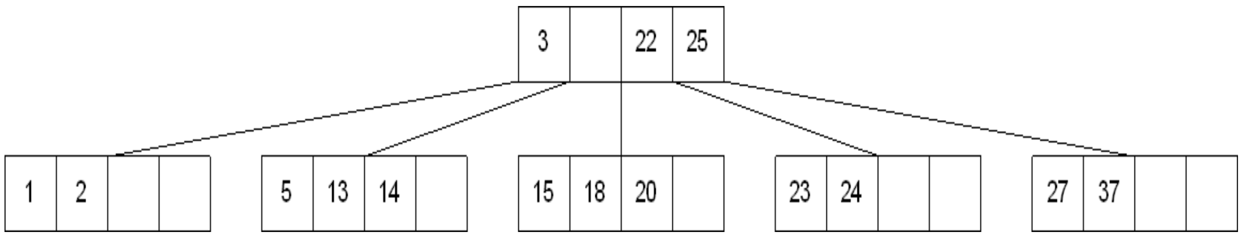
If the immediate successor had been chosen as the replacement



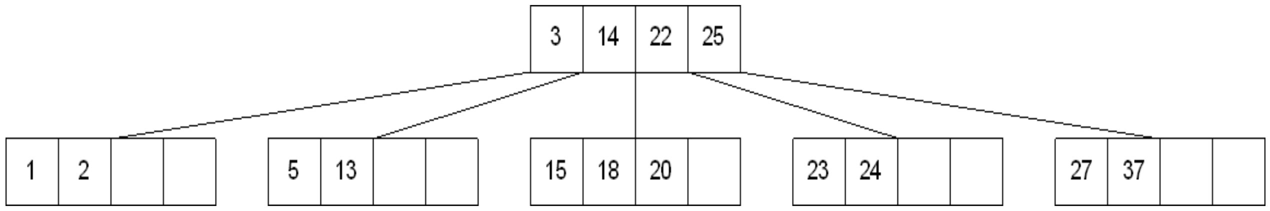
Deleting the successor



values. They are divided between the 2 nodes



And then the middle value is moved to the parent

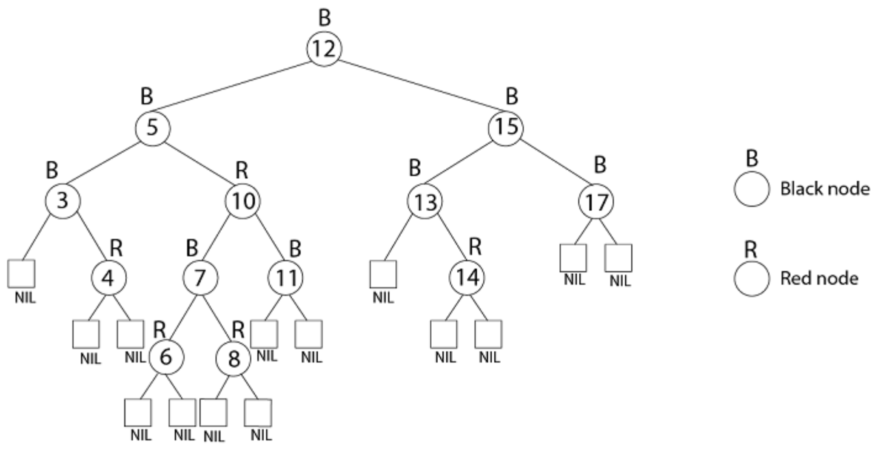


Red Black Tree:

A Red Black Tree is a type of self-balancing binary search tree, in which every node is colored with a red or black. The red black tree satisfies all the properties of the binary search tree but there are some additional properties which were added in a Red Black Tree.

Properties of Red Black Tree:

1. The root node should always be black in color.
2. Every nil child of a node is black in red black tree.
3. The children of a red node are black. It can be possible that parent of red node is black node.
4. All the nil have the same black depth. Every simple path from the root node to the (downward) nil node contains the same number of black nodes.

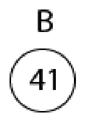


Insertion:

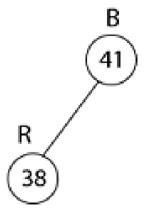
- Insert the new node the way it is done in Binary Search Trees.
- Color the node red
- If an inconsistency arises for the red-black tree, fix the tree according to the type of discrepancy.

Example: Show the red-black trees that result after successively inserting the keys 41,38,31,12,19,8 into an initially empty red-black tree.

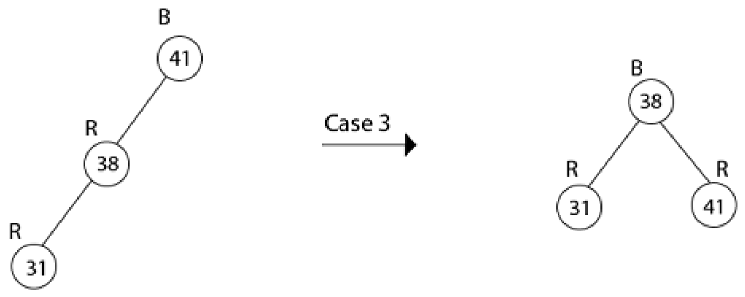
Insert 41



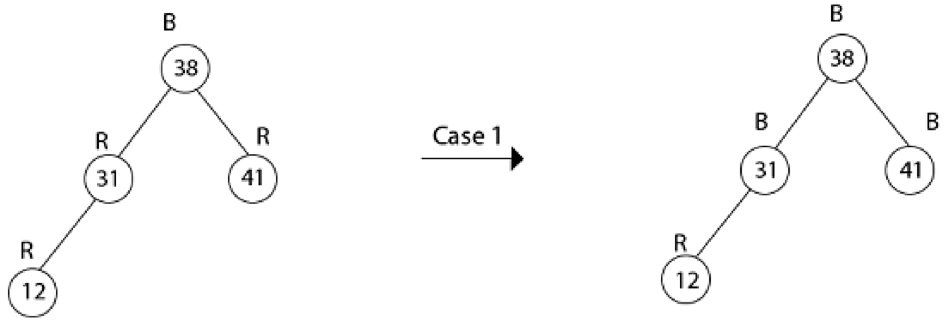
Insert 38



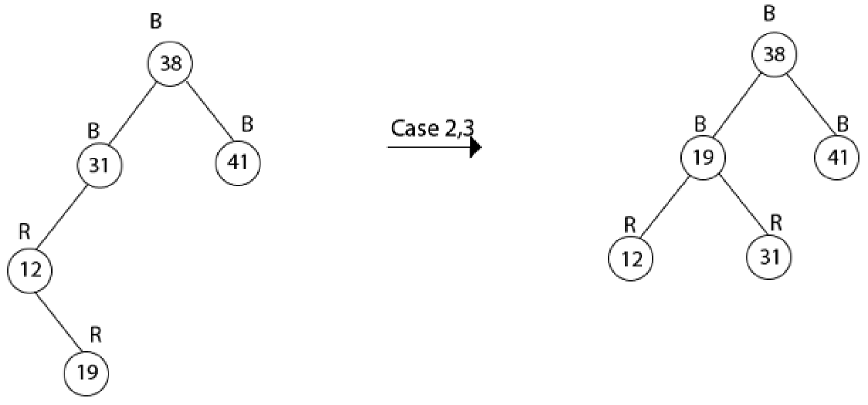
Insert 31



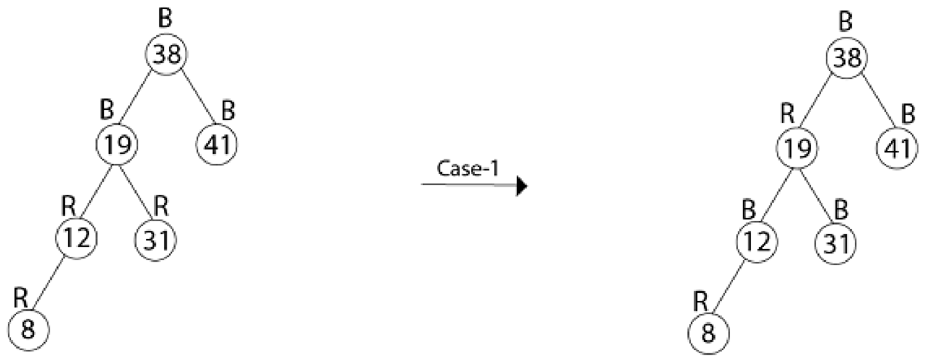
Insert 12



Insert 19



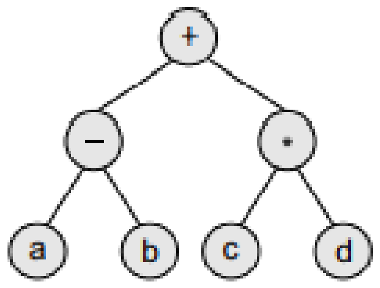
Insert 8



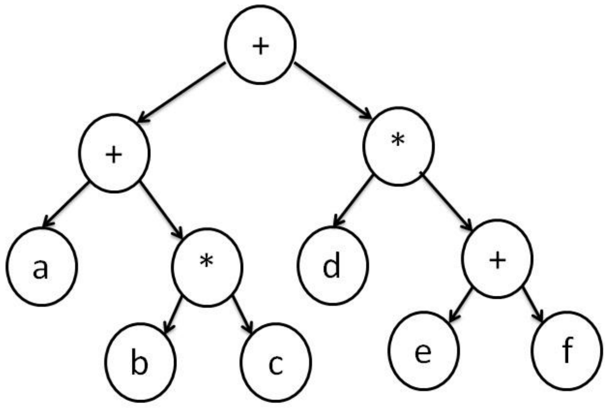
Binary Expression Tree:

The expression tree is a binary tree which is used to store algebraic expressions in which each internal node corresponds to the operator and each leaf node corresponds to the operand.

Expression: $(a - b) + (c * d)$



Expression: $a + (b * c) + d * (e + f)$



Chapter 7 Sorting

- **Sorting means arranging the elements of an array so that they are placed in some relevant order which may be either ascending or descending.**
- Sorting can be done based on key references which can be numbers, alphabets etc.
- if A is an array, then the elements of A are arranged in a sorted order (ascending order) in such a way that A[0] < A[1] < A[2] < ...... < A[N-1].
- For example, if we have an array that is declared and initialized as
- A[] = {21, 34, 11, 9, 1, 0, 22};
- Then the sorted array (ascending order) can be given as: A[] = {0, 1, 9, 11, 21, 22, 34};
- Examples of sorting in real life scenarios :
  - Telephone directories in which names are sorted by location, category (business or residential), and then in an alphabetical order.
  - In a library, the information about books can be sorted alphabetically based on titles and then by authors' names.
  - Customers' addresses can be sorted based on the name of the city and then the street.
- Sorting can be categorized in two different categories:

Types of sorting:

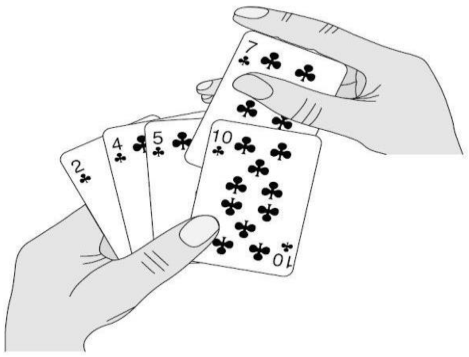
1. **Internal Sorting:**
   - The sorting is done within the computer main memory and all the data to be sorted is stored in main memory.
   - It is performed when the data to be sorted is small enough to fit in main memory. (It is used when the size of input is small.)
   - In it, the storage device used is only main memory (RAM).
   - Examples: Insertion sort, Quick Sort, Bubble Sort, etc.

2. **External Sorting:**
   - The sorting is done in external file disk and data is stored outside the main memory like on disk and only loaded into memory in small chunks.
   - It is usually applied when data can't fit in main memory entirely. (It is used when the size of input is large.)
   - In it, the storage device used are main memory (RAM) and secondary memory (Hard Disk).
   - Examples: External Merge Sort, External Radix Sort, Four Tape Sort.

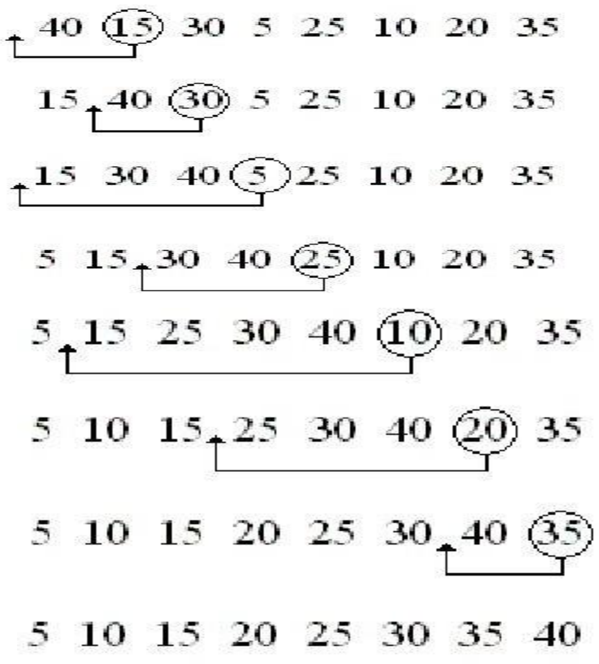
Chapter 7 Sorting

Insertion sort:

- Insertion sort is implemented by inserting a particular data item in its proper position.
- Any unsorted data item is kept on swapping with its previous data items until its proper position is not found.
- The number of swapping makes the previous data items to shift for the new data item to take its position in order.
- Once the new data item is inserted, the next data item after it is chosen for next insertion.
- The process continues until all data items are sorted.
- We all are familiar with this technique of sorting, as we usually use it for ordering a deck of cards.



- It is efficient for smaller data sets, but very inefficient for larger lists.
- Less efficient as compared to other more advanced algorithms such as quick Sort, heap sort, and merge sort.



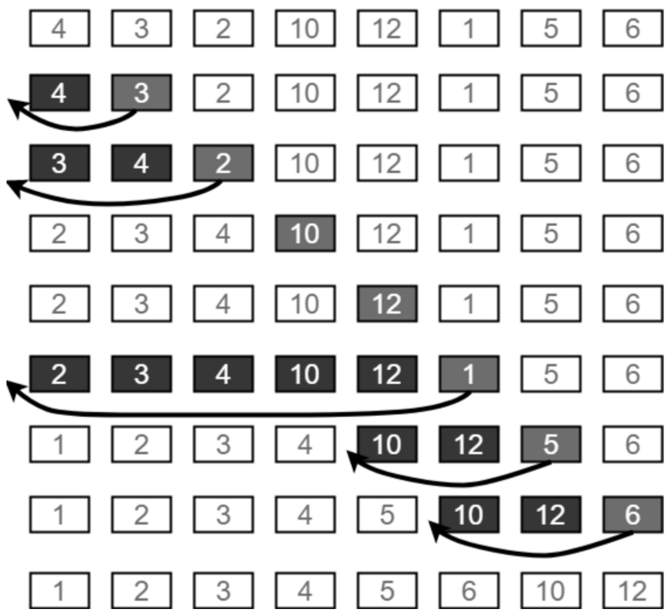
Chapter 7 Sorting

Algorithm:

Step 1 - If it is the first element, it is already sorted. Return 1;
Step 2 – Pick next element
Step 3 – Compare with all elements in the sorted sub-list
Step 4 – If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.
Step 5 – Insert the value
Step 6 – Repeat until list is sorted

```
//Insertion sort logic
 For( i = 1 ; i < size ; i++)
{
   temp = list[i];
   j = i – 1;
   while ((temp < list[j])&& (j ≥ 0))
{
     list[j + 1] = list[j];
     j = j – 1;
   }
   list[j + 1] = temp;
 }
```

Example 1:



Efficiency:
Nested loops

Worst Case : O(n^2)
Best Case : Ω(n)
Average Case : Θ(n^2)

Example 2:

Consider the following unsorted list of elements...

15	20	10	30	50	18	5	45

Asume that sorted portion of the list empty and all elements in the list are in unsorted portion of the list as shown in the figure below...

Sorted | Unsorted

15	20	10	30	50	18	5	45

Move the first element 15 from unsorted portion to sorted portion of the list.

Sorted | Unsorted

15	20	10	30	50	18	5	45

To move element 20 from unsorted to sorted portion, Compare 20 with 15 and insert it at correct position

Sorted | Unsorted

15	20	10	30	50	18	5	45

To move element 10 from unsorted to sorted portion, Compare 10 with 20 and it is smaller so swap. Then compare 10 with 15 again smaller swap. And 10 is insert at its correct position in sorted portion of the list.

Sorted | Unsorted

10	15	20	30	50	18	5	45

To move element 30 from unsorted to sorted portion, Compare 30 with 20, 15 and 10. And it is larger than all these so 30 is directly inserted at last position in sorted portion of the list.

Sorted | Unsorted

10	15	20	30	50	18	5	45

To move element 50 from unsorted to sorted portion, Compare 50 with 30, 20, 15 and 10. And it is larger than all these so 50 is directly inserted at last position in sorted portion of the list.

Sorted | Unsorted

10	15	20	30	50	18	5	45

To move element 18 from unsorted to sorted portion, Compare 18 with 30, 20 and 15. Since 18 is larger than 15, move 20, 30 and 50 one position to the right in the list and insert 18 after 15 in the sorted portion.

Sorted | Unsorted

10	15	18	20	30	50	5	45

To move element 5 from unsorted to sorted portion, Compare 5 with 50, 30, 20, 18, 15 and 10. Since 5 is smaller than all these element, move 10, 15, 18, 20, 30 and 50 one position to the right in the list and insert 5 at first position in the sorted list.

Sorted | Unsorted

5	10	15	18	20	30	50	45

To move element 45 from unsorted to sorted portion, Compare 45 with 50 and 30. Since 45 is larger than 30, move 50 one position to the right in the list and insert 45 after 30 in the sorted list.

Sorted | Unsorted

5	10	15	18	20	30	45	50

Unsorted portion of the list has became empty. So we stop the process. And the final sorted list of elements is as follows...

5	10	15	18	20	30	45	50

Chapter 7 Sorting

Selection Sort:

Selection Sort algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

Algorithm:

Step 1 – Set MIN to location 0

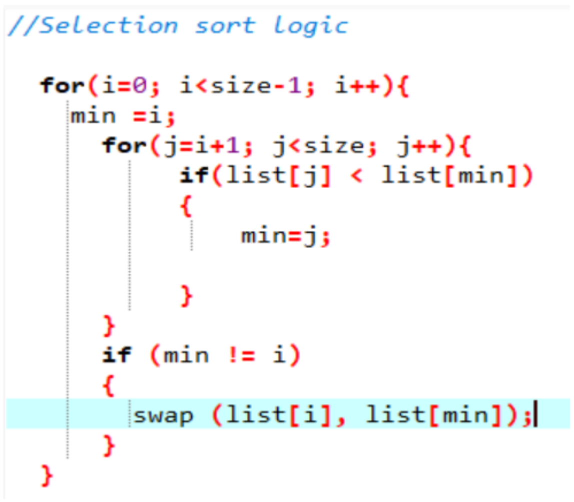
Step 2 – Search the minimum element in the list

Step 3 – Swap with value at location MIN

Step 4 – Increment MIN to point to next element

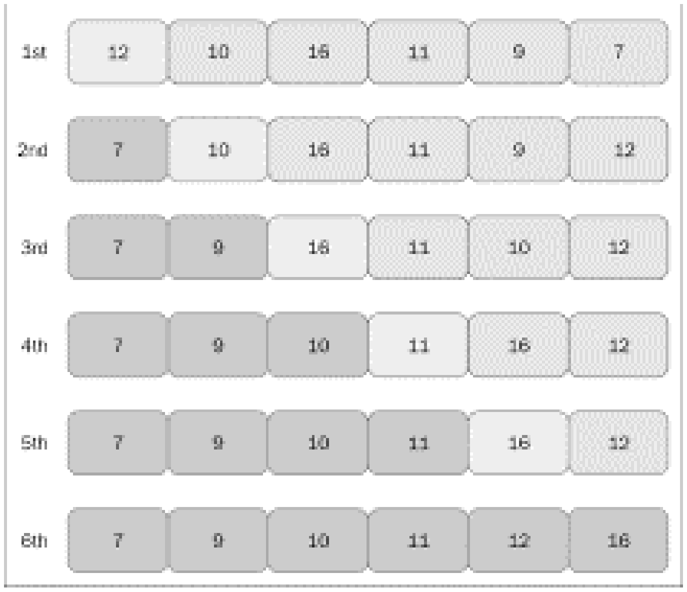
Step 5 – Repeat until list is sorted

Worst Case : O(n^2)
Best Case : Ω(n^2)
Average Case : Θ(n^2)



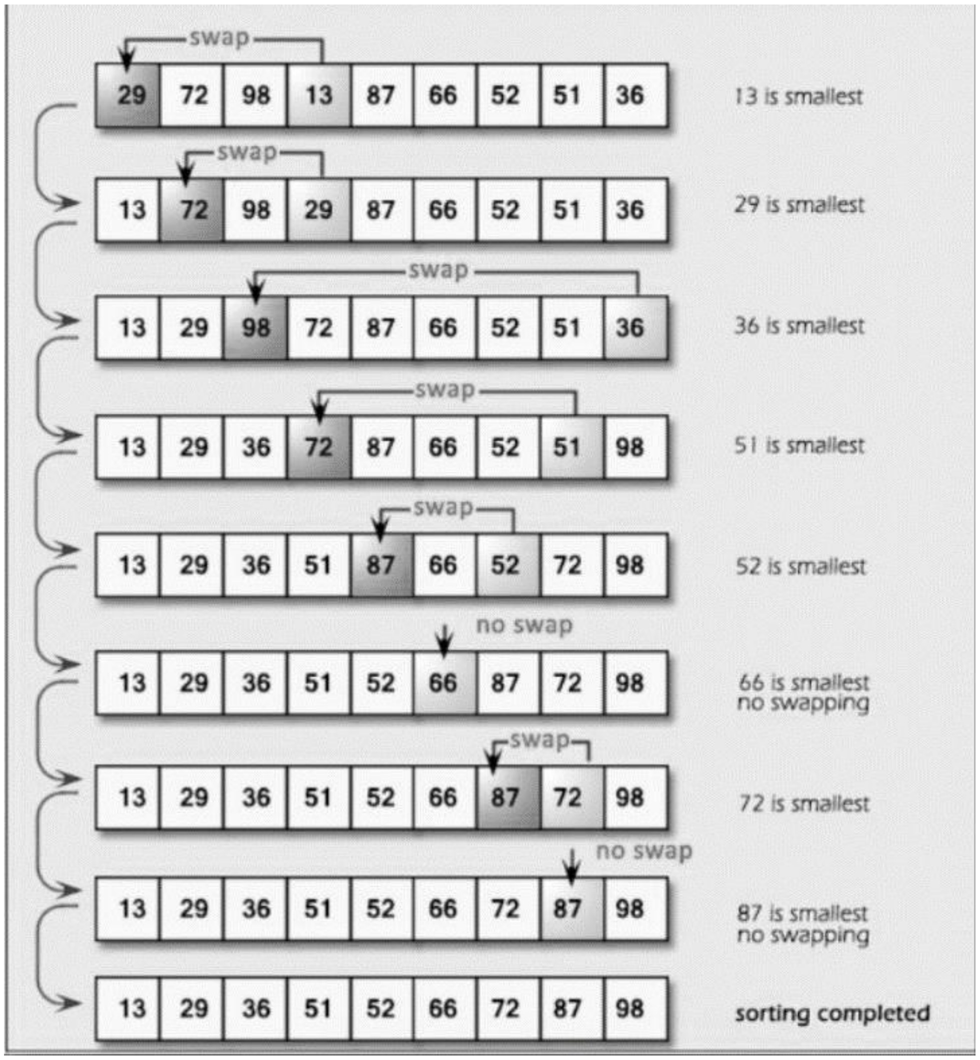
```
//Selection sort logic

  for(i=0; i<size-1; i++){
    min =i;
      for(j=i+1; j<size; j++){
           if(list[j] < list[min])
           {
               min=j;

           }
      }
      if (min != i)
      {
          swap (list[i], list[min]);
      }
  }
```

Example 1:



1st	12	10	16	11	9	7
2nd	7	10	16	11	9	12
3rd	7	9	16	11	10	12
4th	7	9	10	11	16	12
5th	7	9	10	11	16	12
6th	7	9	10	11	12	16

Example 2:



Bubble Sort/ Exchange sort:

- a simple sorting algorithm which
    - repeatedly steps through the list to be sorted
    - compares each pair of adjacent items
    - swaps them if they are in the wrong order
    - The passing through the list is continued until the swapping is not required (i.e. the list sorted)
- it is a comparison sort
- It is called Bubble Sort because the data gradually bubbles up in its proper position
- In each pass at least one data is bubbled up in its proper position

Example: 6 1 3 2 7

First Pass:

$(\mathbf{6}\ \mathbf{1}\ 3\ 2\ 7)(\mathbf{1}\ \mathbf{6}\ 3\ 2\ 7), Swap\ since\ 6\ >\ 1.$

$(1\ \mathbf{6}\ \mathbf{3}\ 2\ 7)(1\ \mathbf{3}\ \mathbf{6}\ 2\ 7), Swap\ since\ 6\ >\ 3$

$(1\ 3\ \mathbf{6}\ \mathbf{2}\ 7)(1\ 3\ \mathbf{2}\ \mathbf{6}\ 7), Swap\ since\ 6\ >\ 2$

$(1\ 3\ 2\ \mathbf{6}\ \mathbf{7}), does\ not\ swap$

Second Pass:

$(\mathbf{1}\ \mathbf{3}\ 2\ 6\ 7)$

$(1\ \mathbf{3}\ \mathbf{2}\ 6\ 7)(1\ \mathbf{2}\ \mathbf{3}\ 6\ 7), Swap\ since\ 3\ >\ 2$

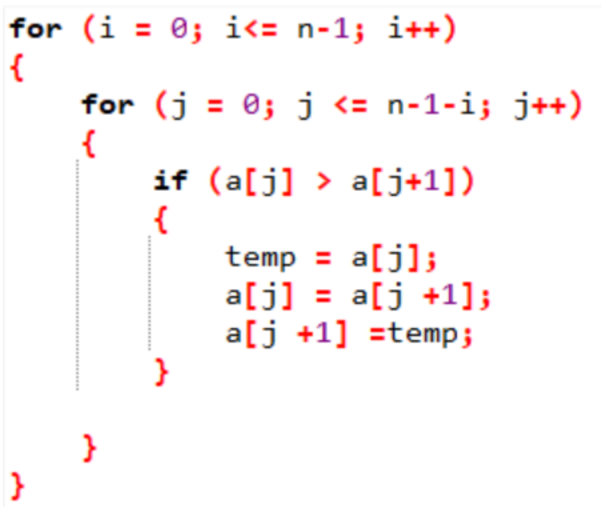
$(1\ 2\ \mathbf{3}\ \mathbf{6}\ 7)(1\ 2\ \mathbf{3}\ \mathbf{6}\ 7)$

$(1\ 2\ 3\ \mathbf{6}\ \mathbf{7})$

- List is already sorted, but our algorithm does not know. Hence one more pass to see if further swapping has to be done

Third Pass:

- (**1 2 3 6 7**), No swap up to the last comparison, hence the list is sorted



```
for (i = 0; i<= n-1; i++)
{
    for (j = 0; j <= n-1-i; j++)
    {
        if (a[j] > a[j+1])
        {
            temp = a[j];
            a[j] = a[j +1];
            a[j +1] =temp;
        }

    }
}
```

Algorithm:

The basic methodology of the working of bubble sort is given as follows:

$(a) In\ Pass\ 1, A[0] and\ A[1] are\ compared, then\ A[1] is\ compared\ with\ A[2],$

$A[2] is\ compared\ with\ A[3], and\ so\ on. Finally, A[N-2] is\ compared\ with\ A[N-1].$

$Pass\ 1\ involves\ n-1\ comparisons\ and\ places\ the\ biggest\ element\ at\ the$

$highest\ index\ of\ the\ array.$

$(b) In\ Pass\ 2, A[0] and\ A[1] are\ compared, then\ A[1] is\ compared\ with\ A[2],$

$A[2] is\ compared\ with\ A[3], and\ so\ on. Finally, A[N-3] is\ compared\ with\ A[N-2].$

$Pass\ 2\ involves\ n-2\ comparisons\ and\ places\ the\ second\ biggest\ element\ at\ the$

$second\ highest\ index\ of\ the\ array.$

$(c)\ In\ Pass\ n-1, A[0]\ and\ A[1]\ are\ compared\ so\ that\ A[0].$

Efficiency:

Nested loops and for n items, n possible swaps

Worst Case : O(n^2)
Best Case : Ω(n^2)
Average Case : Θ(n^2)

Chapter 7 Sorting

Example 1:

A[] = {30, 52, 29, 87, 63, 27, 19, 54}

Pass 1:

(a) Compare 30 and 52. Since 30 < 52, no swapping is done.
(b) Compare 52 and 29. Since 52 > 29, swapping is done.
30, **29, 52,** 87, 63, 27, 19, 54
(c) Compare 52 and 87. Since 52 < 87, no swapping is done.
(d) Compare 87 and 63. Since 87 > 63, swapping is done.
30, 29, 52, **63, 87,** 27, 19, 54
(e) Compare 87 and 27. Since 87 > 27, swapping is done.
30, 29, 52, 63, **27, 87,** 19, 54
(f) Compare 87 and 19. Since 87 > 19, swapping is done.
30, 29, 52, 63, 27, **19, 87,** 54
(g) Compare 87 and 54. Since 87 > 54, swapping is done.
30, 29, 52, 63, 27, 19, **54, 87**

Observe that after the end of the first pass, the largest element is placed at the highest index of the array. All the other elements are still unsorted.

Pass 2:

(a) Compare 30 and 29. Since 30 > 29, swapping is done.
29, 30, 52, 63, 27, 19, 54, 87
(b) Compare 30 and 52. Since 30 < 52, no swapping is done.
(c) Compare 52 and 63. Since 52 < 63, no swapping is done.
(d) Compare 63 and 27. Since 63 > 27, swapping is done.
29, 30, 52, **27, 63,** 19, 54, 87
(e) Compare 63 and 19. Since 63 > 19, swapping is done.

29, 30, 52, 27, **19, 63,** 54, 87
(f) Compare 63 and 54. Since 63 > 54, swapping is done.
29, 30, 52, 27, 19, **54, 63,** 87

Observe that after the end of the second pass, the second largest element is placed at the second highest index of the array. All the other elements are still unsorted.

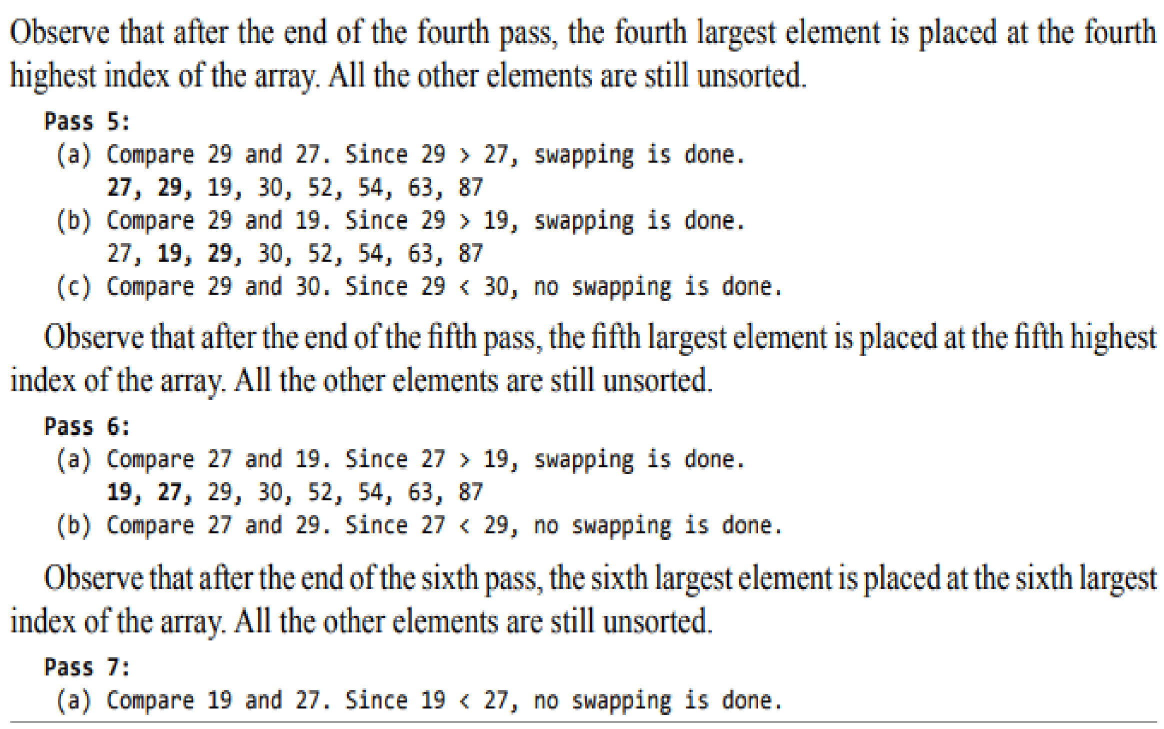
Pass 3:

(a) Compare 29 and 30. Since 29 < 30, no swapping is done.
(b) Compare 30 and 52. Since 30 < 52, no swapping is done.
(c) Compare 52 and 27. Since 52 > 27, swapping is done.
29, 30, **27, 52,** 19, 54, 63, 87
(d) Compare 52 and 19. Since 52 > 19, swapping is done.
29, 30, 27, **19, 52,** 54, 63, 87
(e) Compare 52 and 54. Since 52 < 54, no swapping is done.

Observe that after the end of the third pass, the third largest element is placed at the third highest index of the array. All the other elements are still unsorted.

Pass 4:

(a) Compare 29 and 30. Since 29 < 30, no swapping is done.
(b) Compare 30 and 27. Since 30 > 27, swapping is done.
29, **27, 30,** 19, 52, 54, 63, 87
(c) Compare 30 and 19. Since 30 > 19, swapping is done.
29, 27, **19, 30,** 52, 54, 63, 87
(d) Compare 30 and 52. Since 30 < 52, no swapping is done.



Observe that after the end of the fourth pass, the fourth largest element is placed at the fourth highest index of the array. All the other elements are still unsorted.

```
Pass 5:
(a) Compare 29 and 27. Since 29 > 27, swapping is done.
    27, 29, 19, 30, 52, 54, 63, 87
(b) Compare 29 and 19. Since 29 > 19, swapping is done.
    27, 19, 29, 30, 52, 54, 63, 87
(c) Compare 29 and 30. Since 29 < 30, no swapping is done.
```

Observe that after the end of the fifth pass, the fifth largest element is placed at the fifth highest index of the array. All the other elements are still unsorted.

```
Pass 6:
(a) Compare 27 and 19. Since 27 > 19, swapping is done.
    19, 27, 29, 30, 52, 54, 63, 87
(b) Compare 27 and 29. Since 27 < 29, no swapping is done.
```

Observe that after the end of the sixth pass, the sixth largest element is placed at the sixth largest index of the array. All the other elements are still unsorted.

```
Pass 7:
(a) Compare 19 and 27. Since 19 < 27, no swapping is done.
```

Merge sort:

- It is a divide and conquer algorithm
- At first we divide the given list of item
  - list is divided into two parts from middle
  - The process is repeated until each sub list contain exactly 1 item
- Now is the turn for sort and combine (conquer)
  - A list with a single element is considered sorted automatically
  - Pair of list is sorted and merged into one (i.e. approx. n/2 sub lists of size 2)
  - The sort and merge is keep on repeated until a single list of size n is found
- The overall dividing and conquering is done recursively
- To sort A[p .... r]: (p=starting index , r=ending index)

Divide Step:

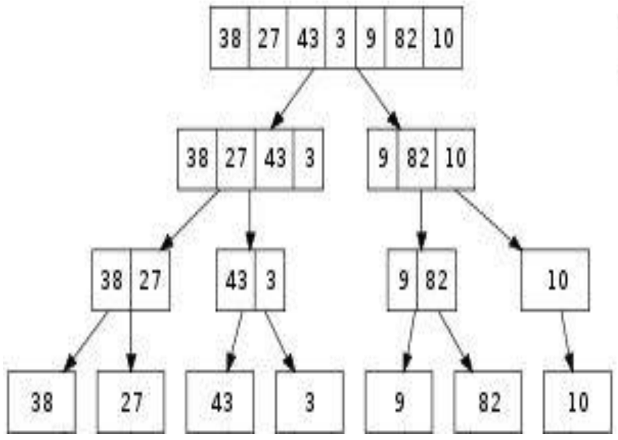
- If a given array A has zero or one element, simply return; it is already sorted.
- Otherwise, split A [p ... r] into two sub arrays A [p ... q] and A [q + 1 … r], each containing about half of the elements of A [p ... r]. That is,
- q is the halfway point of A[p .. r].

Conquer Step

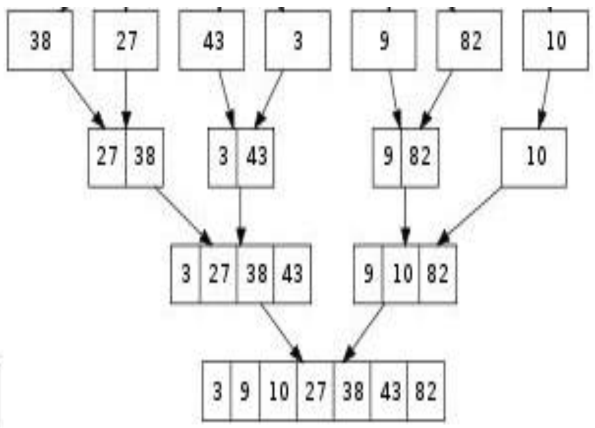
- Conquer by recursively sorting the two sub arrays A [p ... q] and A [q + 1 … r].

Combine Step

- Combine the elements back in A [p … r] by merging the two sorted sub arrays A [p ... q] and A [q + 1 ... r] into a sorted sequence.
- To accomplish this step, we will define a procedure MERGE (A, p, q, r).



Divide



Conquer

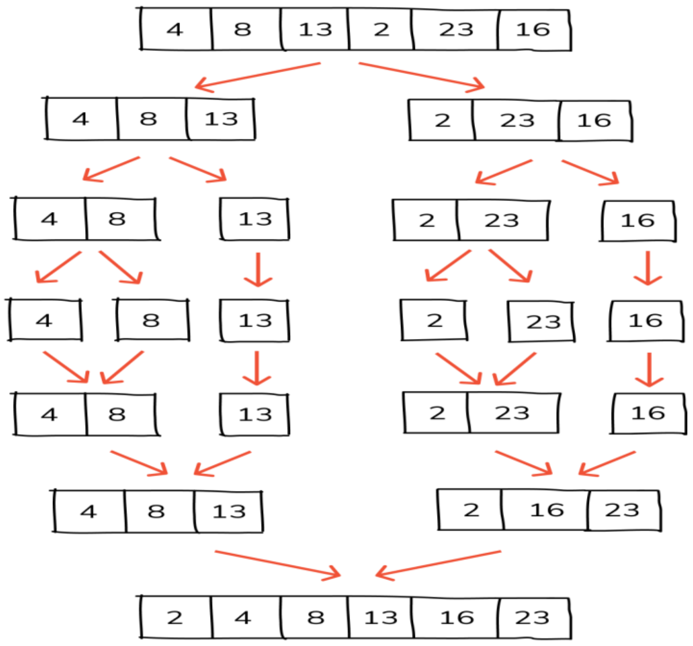
```
Algorithm MergeSort(Arr, start, end)
1. If start < end Then
2.    mid = (start + end)/2
3.    MergeSort(Arr, start, mid)
4.    MergeSort(Arr, mid + 1, end)
5.    Merge(Arr, start, mid, end)
```

```
Algorithm Merge(Arr, start, mid, end)
1.  temp = Create array temp of same size Arr
2.  i = start, j = mid + 1, k = 0
3.  While i <= mid and j <= end
4.    If Arr[i] > Arr[j] Then
5.      temp[k++] = Arr[j++]
6.    Else
7.       temp[k++] = Arr[i++]
8.  While i <= mid
9.     temp[k++] = Arr[i++]
10. While j <= end
11.    temp[k++] = Arr[j++]
12. Loop from p = start to end
13.   Arr[p] = temp[p]
```

Efficiency:
Divide and conquer (tree structure) : log n
However n sub-lists need to be sorted

Worst Case : $\boldsymbol{O}(n * logn)$
Best Case : $O(\, n * logn\,)$
Average Case : $O(\, n * logn\,)$

Example 1:



Redix sort:

Radix sort is the linear sorting algorithm that is used for integers. In Radix sort, there is digit by digit sorting is performed that is started from the least significant digit to the most significant digit.

The process of radix sort works similar to the sorting of students names, according to the alphabetical order.

Algorithm:

Step 1 – Find largest element in the given array and number of digits in the largest element.

Step 2 - Define 10 queues each representing a bucket for each digit from 0 to 9.

Step 3 - Consider the least significant digit of each number in the list which is to be sorted.

Step 4 - Insert each number into their respective queue based on the least significant digit.

Step 5 - Group all the numbers from queue 0 to queue 9 in the order they have inserted into their respective queues.

Step 6 - Repeat from step 4 based on the next least significant digit.

Step 7 - Repeat from step 3 until all the numbers are grouped based on the most significant digit.

Chapter 7 Sorting

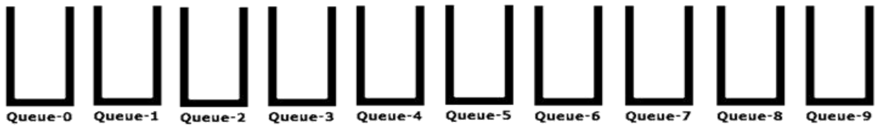
Example 1:

$$\textbf{Worst Case}: \; O(n)$$
$$\textbf{Best Case}: \; O(\,n\,)$$
$$\textbf{Average Case}: \; O(\,n\,)$$

Consider the following list of unsorted integer numbers

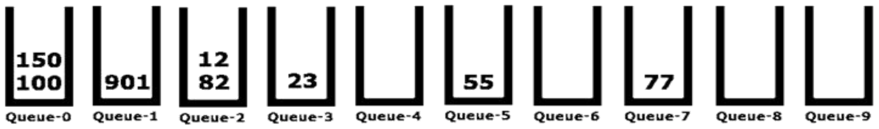
82, 901, 100, 12, 150, 77, 55 & 23

Step 1 - Define 10 queues each represents a bucket for digits from 0 to 9.



Step 2 - Insert all the numbers of the list into respective queue based on the Least significant digit (once placed digit) of every number.

8**2**, 90**1**, 10**0**, 1**2**, 15**0**, 7**7**, 5**5** & 2**3**

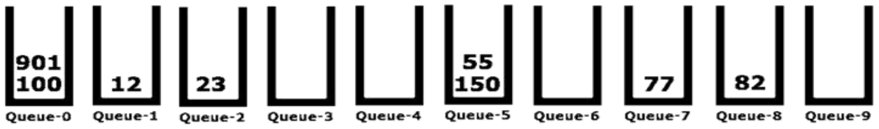


Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

100, 150, 901, 82, 12, 23, 55 & 77

Step 3 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Tens placed digit) of every number.

1**0**0, 1**5**0, 9**0**1, **8**2, **1**2, **2**3, **5**5 & **7**7

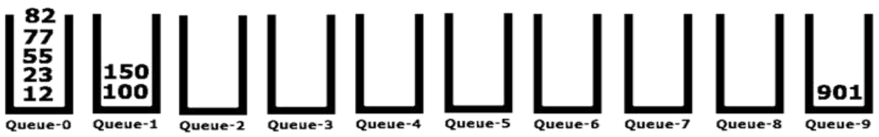


Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

100, 901, 12, 23, 150, 55, 77 & 82

Step 4 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Hundres placed digit) of every number.

100, **9**01, 12, 23, **1**50, 55, 77 & 82



Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

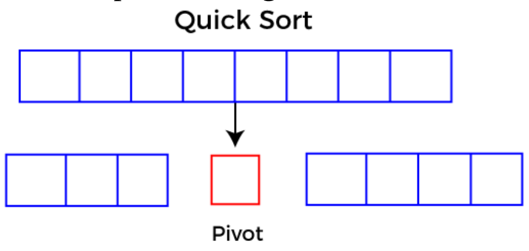
12, 23, 55, 77, 82, 100, 150, 901

List got sorted in the incresing order.

Chapter 7 Sorting

Quick sort:

- Also called partition-exchange sort
- Uses divide and conquer algorithm
- One pivot element is chosen from within the list
- The list is divided into two partitions
    - All values less than the pivot are placed on left side of pivot
    - All greater values are placed on right side of the pivot
- After a single pass, the pivot is in its proper position
- The left and right partitions are sorted recursively using the same method
- Joining the left sorted, pivot and right sorted results with the list in sorted order



Partition 1	Pivot	Partition 2

Values <=Pivot　　　　　　　　　　Values > Pivot

Algorithm:

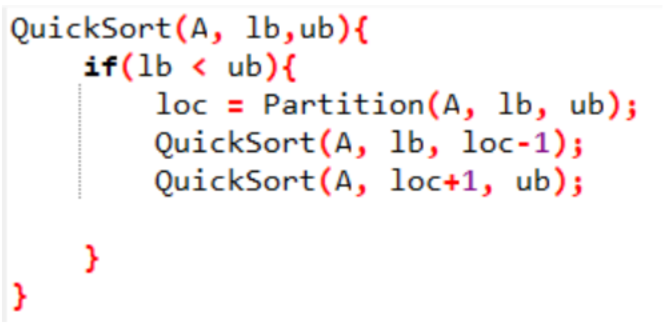
- Declare and initialize necessary variables array size, pivot,
- A[n]; array to be sorted, lb=0; first index of array, ub=n-1; last index of array

Step 1 − Make the left-most index value pivot

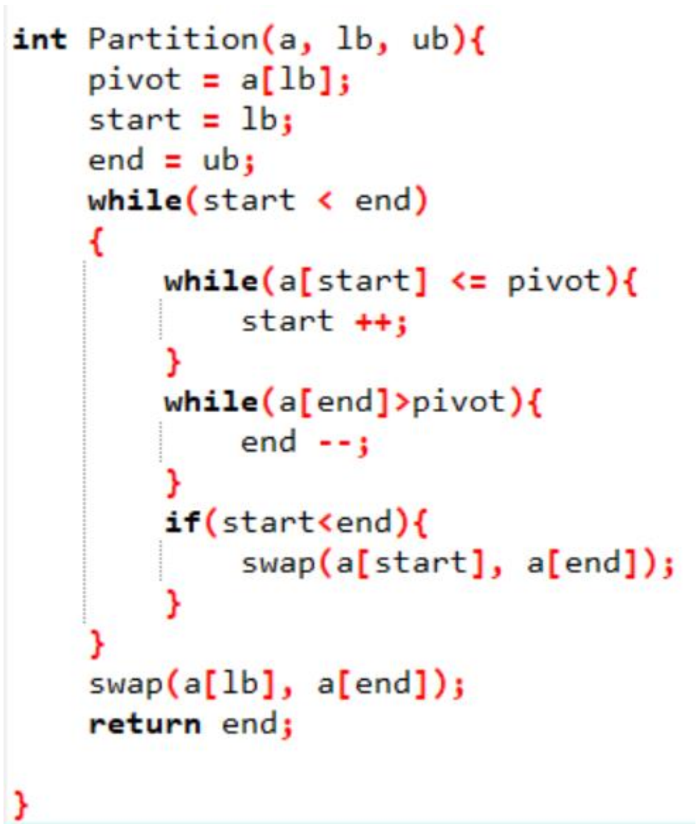
Step 2 − partition the array using pivot value

Step 3 − quicksort left partition recursively

Step 4 − quicksort right partition recursively



```
QuickSort(A, lb,ub){
    if(lb < ub){
        loc = Partition(A, lb, ub);
        QuickSort(A, lb, loc-1);
        QuickSort(A, loc+1, ub);

    }
}
```

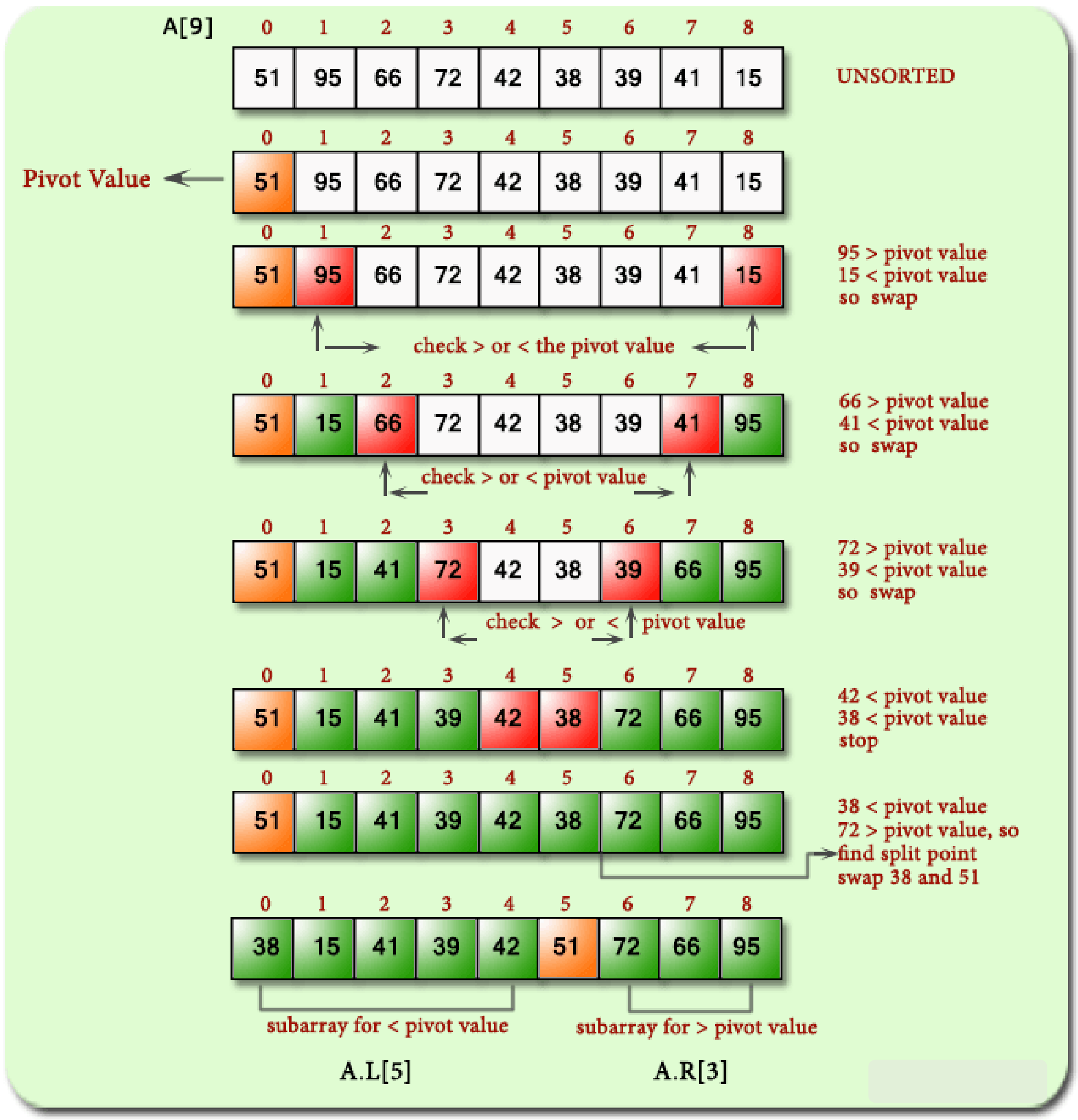


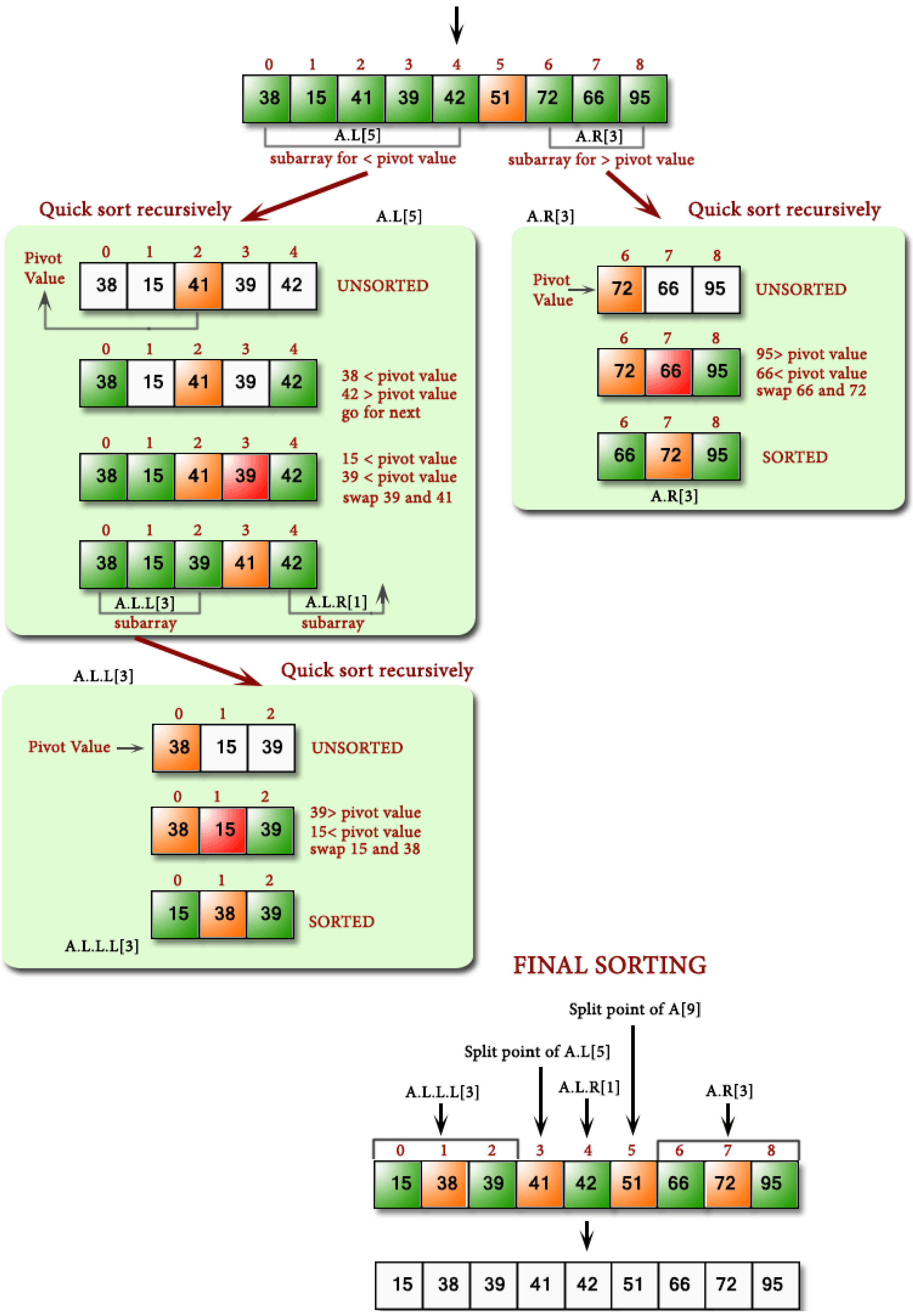
```
int Partition(a, lb, ub){
    pivot = a[lb];
    start = lb;
    end = ub;
    while(start < end)
    {
        while(a[start] <= pivot){
            start ++;
        }
        while(a[end]>pivot){
            end --;
        }
        if(start<end){
            swap(a[start], a[end]);
        }
    }
    swap(a[lb], a[end]);
    return end;

}
```

Worst Case : $\boldsymbol{O}(n^2)$
Best Case : $\boldsymbol{O}(n \log n)$
Average Case : $\boldsymbol{O}(n \log n)$

Example 1:

Quick Sort




0 1 2 3 4 5 6 7 8
38 15 41 39 42 51 72 66 95
A.L[5]
subarray for < pivot value
A.R[3]
subarray for > pivot value
Quick sort recursively
A.L[5]
Pivot Value
38 15 41 39 42
UNSORTED
38 < pivot value
42 > pivot value
go for next
15 < pivot value
39 < pivot value
swap 39 and 41
38 15 39 41 42
A.L.L[3] subarray
A.L.R[1] subarray
Quick sort recursively
A.R[3]
Pivot Value
72 66 95
UNSORTED
95> pivot value
66< pivot value
swap 66 and 72
66 72 95
SORTED
A.R[3]
A.L.L[3]
Quick sort recursively
Pivot Value
38 15 39
UNSORTED
39> pivot value
15< pivot value
swap 15 and 38
15 38 39
SORTED
A.L.L.L[3]
FINAL SORTING
Split point of A[9]
Split point of A.L[5]
A.L.L.L[3]
A.L.R[1]
A.R[3]
15 38 39 41 42 51 66 72 95
15 38 39 41 42 51 66 72 95

Chapter 7 Sorting

Shell sort:

Shell sort is the generalization of insertion sort, which overcomes the drawbacks of insertion sort by comparing elements separated by a gap of several positions.

It first sorts elements that are far apart from each other by swapping and successively reduces the gap between the elements to be sorted. This gap is called as interval. The interval between the elements is reduced based on the sequence used. Some of the optimal sequences that can be used in the shell sort algorithm are:

- Shell's original sequence: $N/2, N/4, \ldots, 1$
- Knuth's Formula $= h * 3 + 1$ →where $-$ h is interval with initial value 1
- Hibbard's increments: $1, 3, 7, 15, 31, 63, 127, 255, 511 \ldots$
- Pratt: $1, 2, 3, 4, 6, 9, 8, 12, 18, 27, 16, 24, 36, 54, 81 \ldots$

Best Case:	O(n*log n)
Average Case:	O(n*log n)
Worst Case:	O(n^2)

Algorithm:

Step 1 – for the size of array 'N'.

Step 2 – Divide the list into smaller sub-list of interval N/2.

Step 3 – Sort these sub-lists using **insertion sort.**

Step 3 – Repeat until complete list is sorted.

Shell Sort(a, n) // 'a' is the given array, 'n' is the size of array

```
• $for\ (interval = n/2;\ interval >= 1;\ interval\ /= 2)$
• $for\ (j = interval;\ j < n;\ j++)$
• $for\ (i = j - interval;\ i >= 0;\ i -= interval)$
• $if\ (\ a[i + interval] > a[i]\ )$
      o $break$
• $otherwise$
      o $swap\ (\ a[i + interval], a[i]\ )$
• $End\ Shell\ Sort$
```

Example:

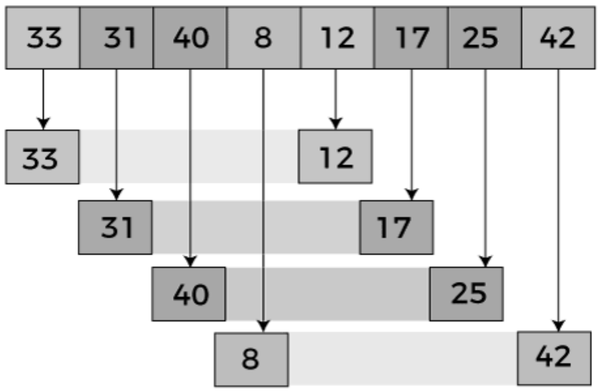
Let the elements of array are –

33	31	40	8	12	17	25	42

We will use the original sequence of shell sort, i.e., N/2, N/4...1 as the intervals.

Here, in the first loop, the element at the 0th position will be compared with the element at 4th position. If the 0th element is greater, it will be swapped with the element at 4th position. Otherwise, it remains the same. This process will continue for the remaining elements.

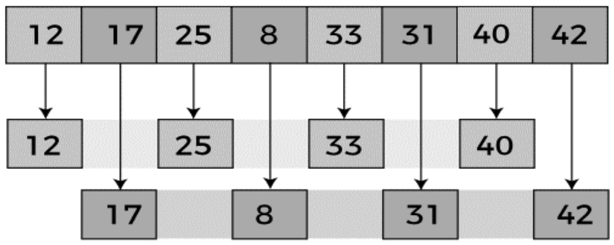
At the interval of 4, the sub lists are {33, 12}, {31, 17}, {40, 25}, {8, 42}.



After comparing and swapping, the updated array will look as follows –

12	17	25	8	33	31	40	42

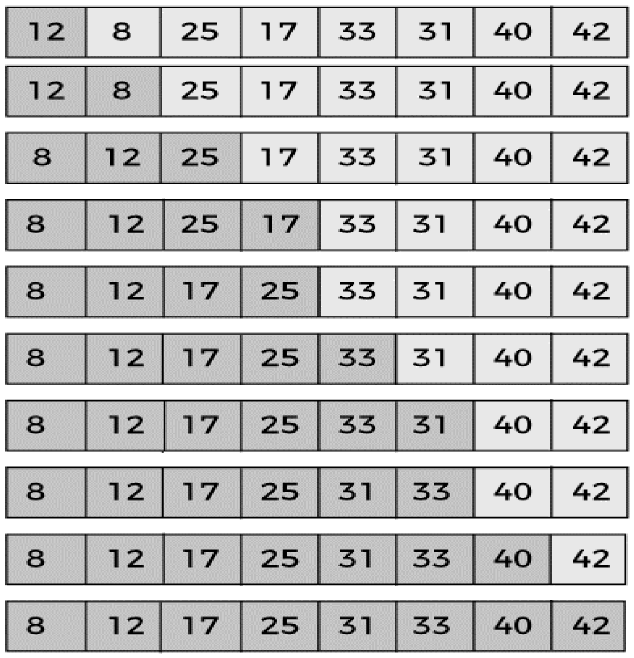
In the second loop, elements are lying at the interval of 2 (n/4 = 2), where n = 8.



After comparing and swapping, the updated array will look as follows –

12	8	25	17	33	31	40	42

In the third loop, elements are lying at the interval of 1 (n/8 = 1), where n = 8.



12	8	25	17	33	31	40	42
12	8	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42

Chapter 7 Sorting

Heap sort as a priority queue:

Heap sort is one of the sorting algorithms used to arrange a list of elements in order. Heap sort algorithm uses one of the tree concepts called Heap Tree. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list. Heap sort is the in-place sorting algorithm.

Algorithm:

Step 1 - Construct a **Binary Tree** with given list of Elements.

Step 2 - Transform the Binary Tree into **Max Heap.**

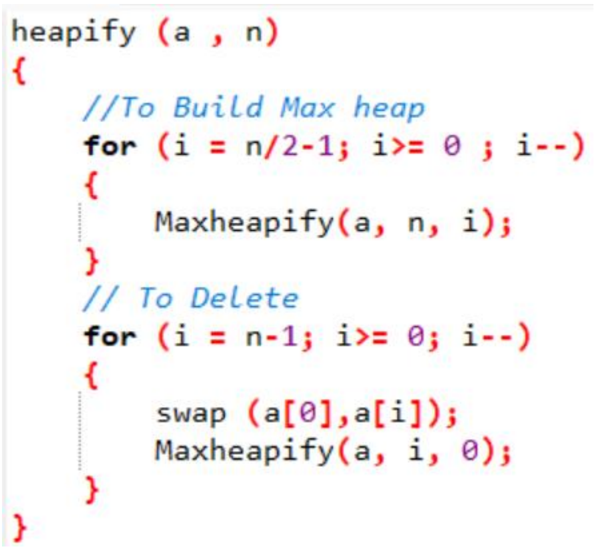
Step 3 - Delete the root element from Max Heap using **Heapify** method.

Step 4 - Put the deleted element into the Sorted list.

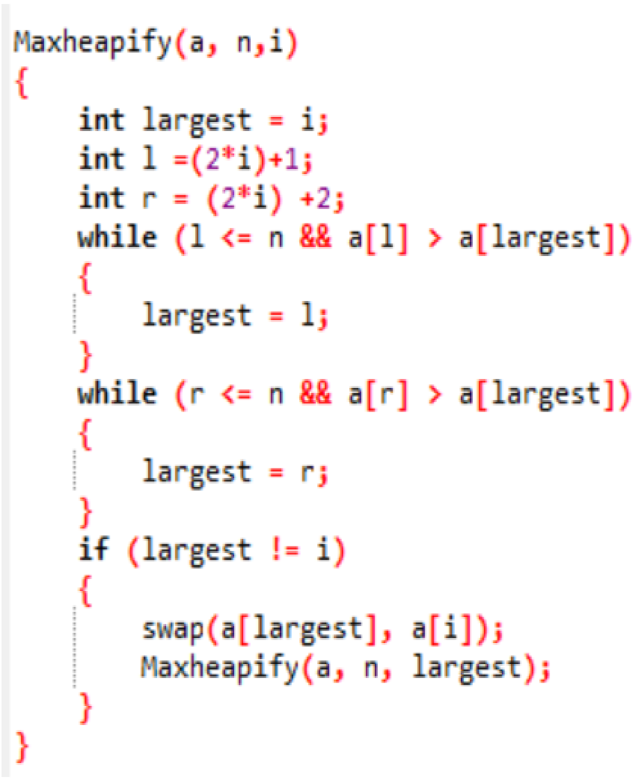
Step 5 - Repeat the same until Max Heap becomes empty.

Step 6 - Display the sorted list.

Heapify (a, n) // 'a' is the given array, 'n' is the size of array



```
heapify (a , n)
{
    //To Build Max heap
    for (i = n/2-1; i>= 0 ; i--)
    {
        Maxheapify(a, n, i);
    }
    // To Delete
    for (i = n-1; i>= 0; i--)
    {
        swap (a[0],a[i]);
        Maxheapify(a, i, 0);
    }
}
```



```
Maxheapify(a, n,i)
{
    int largest = i;
    int l =(2*i)+1;
    int r = (2*i) +2;
    while (l <= n && a[l] > a[largest])
    {
        largest = l;
    }
    while (r <= n && a[r] > a[largest])
    {
        largest = r;
    }
    if (largest != i)
    {
        swap(a[largest], a[i]);
        Maxheapify(a, n, largest);
    }
}
```

$\boldsymbol{Worst\ Case}: \ \boldsymbol{O}(n \log n)$
$\boldsymbol{Best\ Case}: \ \boldsymbol{O}(n \log n)$
$\boldsymbol{Average\ Case}: \ \boldsymbol{O}(n \log n)$

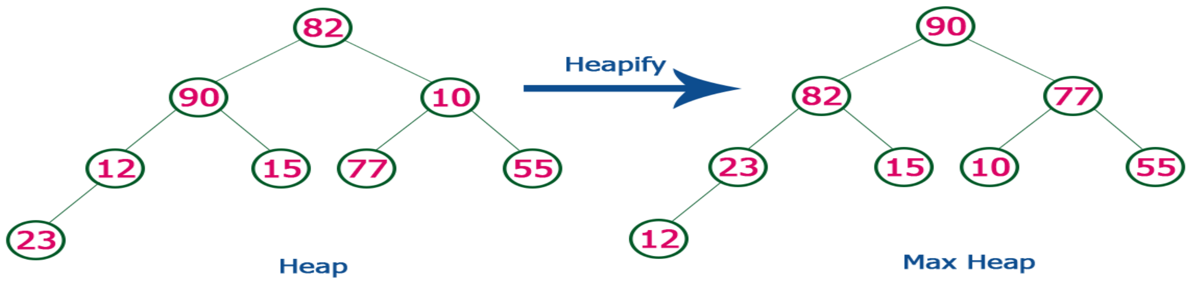
Chapter 7 Sorting

Example 1:

Consider the following list of unsorted numbers which are to be sort using Heap Sort

82, 90, 10, 12, 15, 77, 55, 23

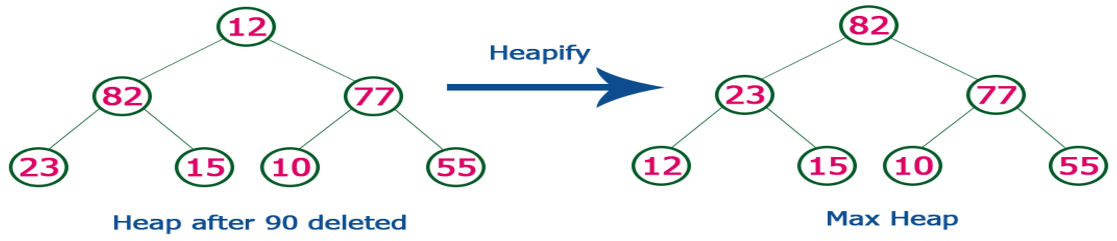
Step 1 - Construct a Heap with given list of unsorted numbers and convert to Max Heap



list of numbers after heap converted to Max Heap

90, 82, 77, 23, 15, 10, 55, 12

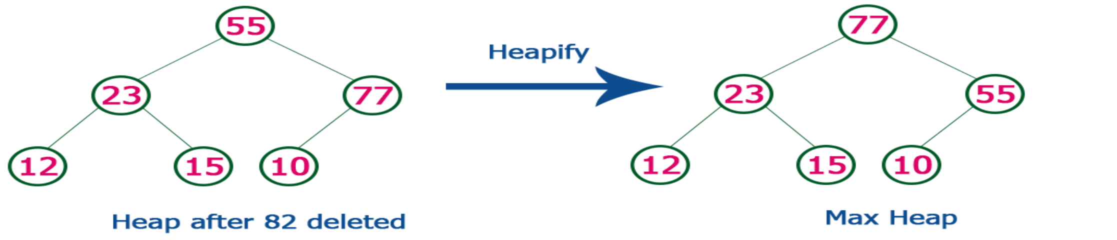
Step 2 - Delete root (**90**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 90 with 12.

82, 23, 77, 12, 15, 10, 55, **90**

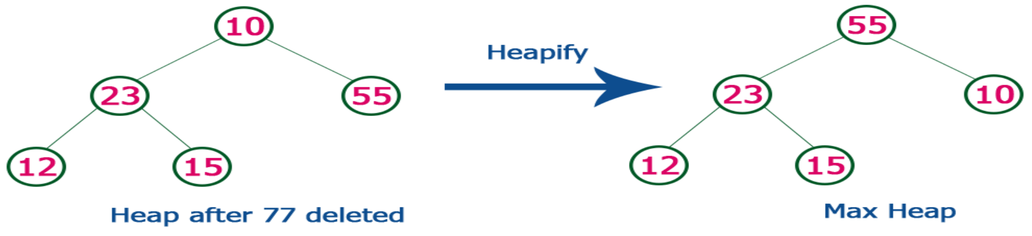
Step 3 - Delete root (**82**) from the Max Heap. To delete root node it needs to be swapped with last node (**55**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 82 with 55.

77, 23, 55, 12, 15, 10, **82**, **90**

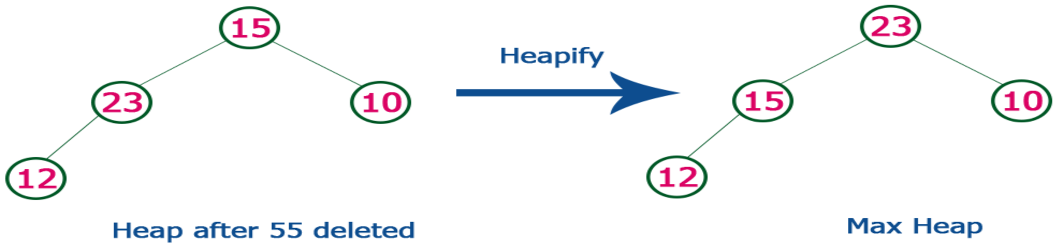
Step 4 - Delete root (**77**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 77 with 10.

55, 23, 10, 12, 15, **77**, **82**, **90**

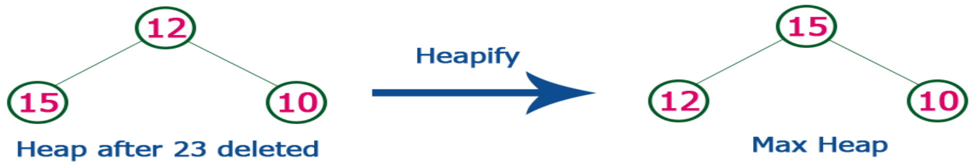
Step 5 - Delete root (**55**) from the Max Heap. To delete root node it needs to be swapped with last node (**15**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 55 with 15.

23, 15, 10, 12, **55**, **77**, **82**, **90**

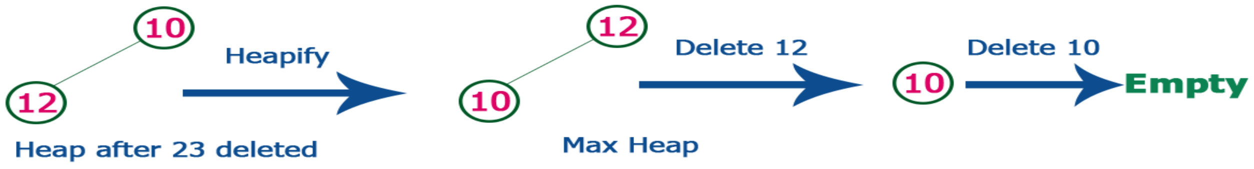
Step 6 - Delete root (**23**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 23 with 12.

15, 12, 10, **23**, **55**, **77**, **82**, **90**

Step 7 - Delete root (**15**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after Deleting 15, 12 & 10 from the Max Heap.

10, 12, 15, 23, 55, 77, 82, 90

Whenever Max Heap becomes Empty, the list get sorted in Ascending order

Big 'O' notation and Efficiency of sorting:

Big - Oh notation is used to define the upper bound of an algorithm in terms of Time Complexity. That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.

Big-O Analysis of Algorithms:

We can express algorithmic complexity using the big-O notation. For a problem of size N:

- A constant-time function/method is "order 1" : O(1)
- A linear-time function/method is "order N" : O(N)
- A quadratic-time function/method is "order N squared" : $O(N^2)$

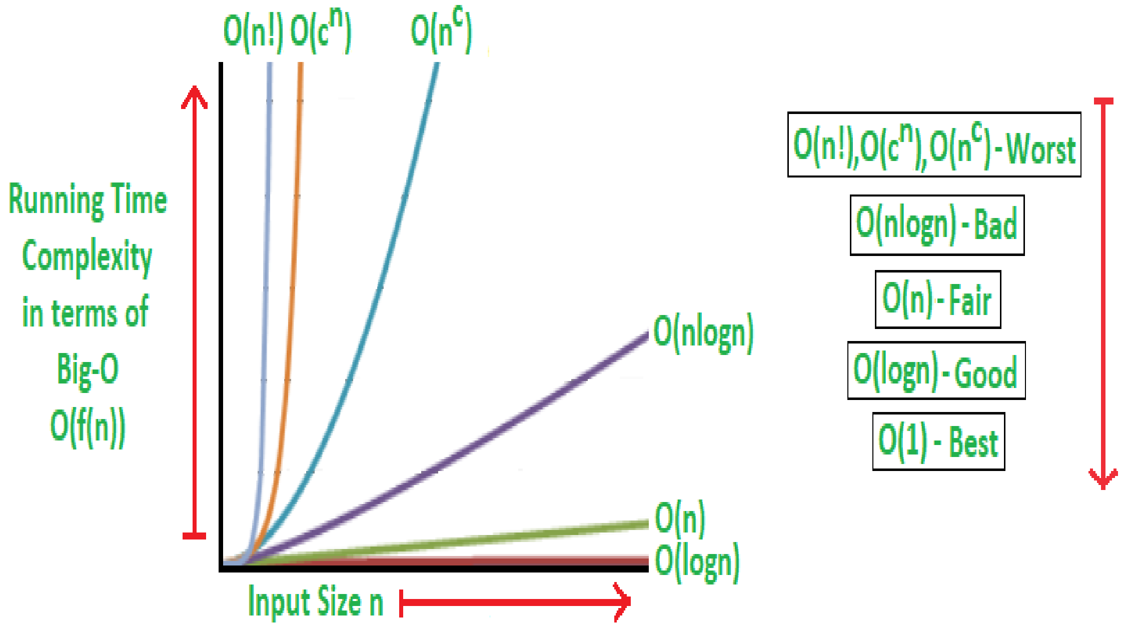
Chapter 7 Sorting

The general step wise procedure for Big-O runtime analysis is as follows:

1. Figure out what the input is and what n represents.
2. Express the maximum number of operations, the algorithm performs in terms of n.
3. Eliminate all excluding the highest order terms.
4. Remove all the constant factors.

Runtime Analysis of Algorithms:

- $A\ logarithmic\ algorithm -\ O(logn)$
  $Runtime\ grows\ logarithmically\ in\ proportion\ to\ n.$
- $A\ linear\ algorithm -\ O(n)$
  $Runtime\ grows\ directly\ in\ proportion\ to\ n.$
- $A\ superlinear\ algorithm -\ O(nlogn)$
  $Runtime\ grows\ in\ proportion\ to\ n.$
- $A\ polynomial\ algorithm -\ O(n^c)$
  $Runtime\ grows\ quicker\ than\ previous\ all\ based\ on\ n.$
- $A\ exponential\ algorithm -\ O(c^n)$
  $Runtime\ grows\ even\ faster\ than\ polynomial\ algorithm\ based\ on\ n.$
- $A\ factorial\ algorithm -\ O(n!)$
  $Runtime\ grows\ the\ fastest\ and\ becomes\ quickly\ unusable\ for\ even\ small\ values\ of\ n.$



Chapter 7 Sorting

Algorithmic Examples of Runtime Analysis:

- $Logarithmic\ algorithm - O(logn) - Binary\ Search.$
- $Linear\ algorithm - O(n) - Linear\ Search.$
- $Super\ linear\ algorithm - O(nlogn) - Heap\ Sort, Merge\ Sort.$
- $Polynomial\ algorithm - O(n\hat{}c) - Bubble\ Sort, Selection\ Sort, Insertion\ Sort, Bucket\ Sort.$
- $Exponential\ algorithm - O(c\hat{}n) - Tower\ of\ Hanoi.$
- $Factorial\ algorithm - O(n!) - Determinant\ Expansion\ by\ Minors.$

Analysis of Bubble sort

- In Bubble Sort, n-1 comparisons will be done in the 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on.

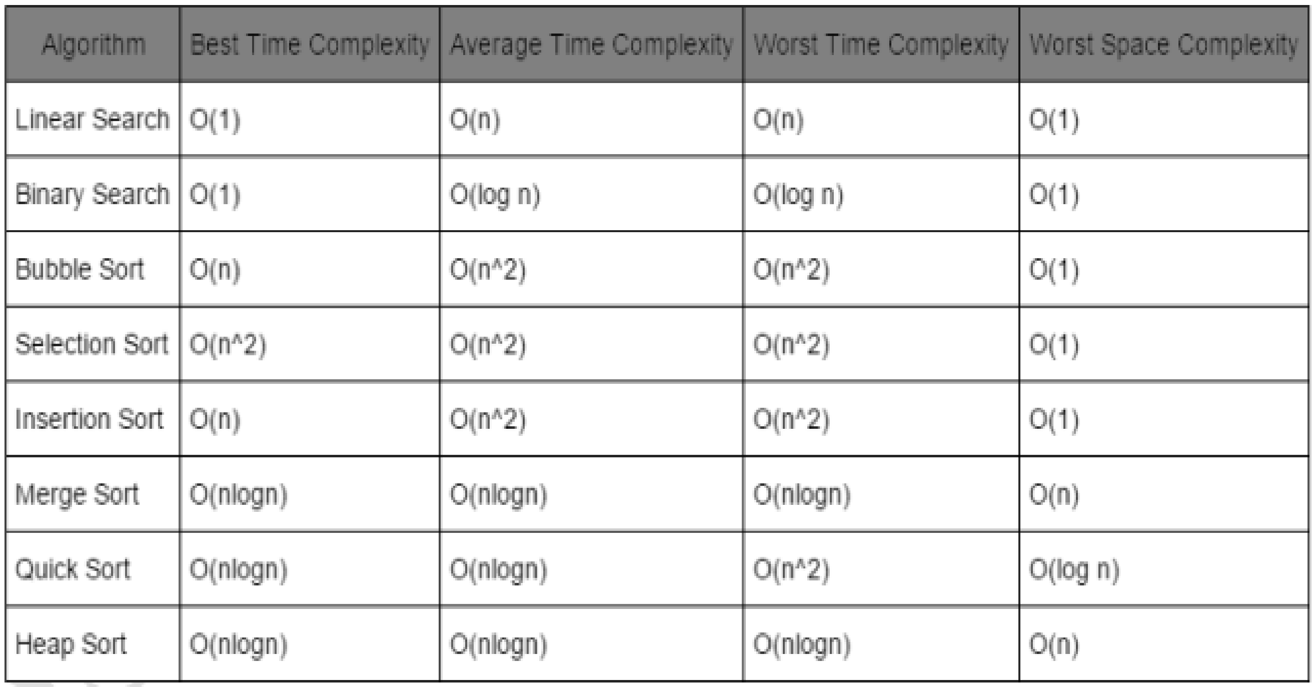
$$So\ the\ total\ number\ of\ comparisons\ will\ be,$$

$$(n-1) + (n-2) + (n-3) + \ldots\ldots + 3 + 2 + 1$$

$$Sum = \frac{n(n-1)}{2} \qquad \because \text{Sn = n/2 [2a + (n-1) d]}$$

$$O(n^2)$$

- Hence the time complexity of Bubble Sort is O(n^2).



Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(log n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)

Chapter 8 Searching

It is a process of finding an element within the list of elements stored in any order.

It is not necessary that the data item we are searching for must be present in the list.

If the searched item is present in the list then the searching algorithm (or program) can find that data item, in which case we say that the search is successful, but if the searched item is not present in the list, then it cannot be found and we say that the search is unsuccessful.

Types of Searching:

1. Linear /Sequential Searching:

- It is the simplest technique to find out an element in an unordered list.
- We search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.
- Suppose there are 'n' elements organized sequentially on a List. The number of comparisons required to retrieve an element from the list, purely depends on where the element is stored in the list. If it is the first element, one comparison will do; if it is second element two comparisons are necessary and so on. On an average you need [(n+1)/2] comparison's to search an element. If search is not successful, you would need 'n' comparisons. **The time complexity of linear search is O (n).**

Steps:

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with the first element in the list.

Step 3 - If both are matched, then display "Given element is found!!!" and terminate the function

Step 4 - If both are not matched, then compare search element with the next element in the list.

Step 5 - Repeat steps 3 and 4 until search element is compared with last element in the list.

Step 6 - If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

Chapter 8 Searching

Suppose we have the following unsorted list: 45, 39, 8, 54, 77, 38, 24, 16, 4, 7, 9, and 20.

Index	0	1	2	3	4	5	6	7	8	9	10	11
list	45	39	8	54	77	38	24	16	4	7	9	20

- 24 is compared with first element (45). If not matched, move to next element.
- 24 is compared with second element (39). If not matched, move to next element.
- 24 is compared with third element (8). If not matched, move to next element.
- 24 is compared with fourth element (54). If not matched, move to next element.
- 24 is compared with fifth element (77). If not matched, move to next element.
- 24 is compared with Sixth element (38). If not matched, move to next element.
- 24 is compared with seventh element (24).Matched.

Sequential search efficiency:

- The number of comparisons of keys done in sequential search of a list of length n is

i.	Unsuccessful search	: n comparisons
ii.	Successful search, best case	: 1 comparison
iii.	Successful search, worst case	: n comparisons
iv.	Successful search, average case	: (n + 1)/2 comparisons

v. In any case, the number of comparison is O(n)

2. Binary Search:

- It is an extremely efficient algorithm.
- This search technique searches the given item in minimum possible comparisons.
- To do the binary search, first we have to sort the array elements.
- The logic behind this technique is given below.
  i. First find the middle element of the array.
  ii. Compare the middle element with an item.
  iii. There are three cases:
    a. If it is a desired element then search is successful,
    b. If it is less than the desired item then search only in the first half of the array.
    c. If it is greater than the desired item, search in the second half of the array.
  iv. Repeat the same steps until an element is found or search area is exhausted.

Requirements:

i. The list must be ordered.
ii. Rapid random access is required, so we cannot use binary search for a linked list.

Chapter 8 Searching

Binary search efficiency:

- i. In all cases, the no. of comparisons in proportional to n. Hence, no. of comparisons in binary search is O (log n), where n is the no. of items in the list.
- ii. Obviously binary search is faster than sequential search, but there is an extra overhead in maintaining the list ordered.
- iii. Binary search is best suited for lists that are constructed and sorted once, and then repeatedly searched.

Algorithm:

Given a table k of n elements in searching order searching for value x.

```
1. $Initialize: low \leftarrow 0, high \leftarrow n-1$
2. $Perform\ Search: Repeat\ through\ step\ 4\ while\ low\ <=\ high.$
3. $Obtain\ index\ of\ midpoint\ of\ interval: mid \leftarrow (low\ +\ high)/2$
4. $Compare:$
        $If\ X < k[mid] then\ high \leftarrow mid\ -1$
        $Else\ if\ X > k[mid] then\ low \leftarrow mid + 1$
        $Else\ Write\ (“Search\ is\ successful”)$
        $Return\ (mid)$
5. $(“Search\ is\ unsuccessful”)$
        $Return$
6. $Finished$
```

Example 1:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
22	43	68	100	120	330	420	555	560	570

Search 43

Insertion	Low	High	Mid	Remarks
1.	0	9	4	X<k[4]
2.	0	3	1	X=k[1]

Data found in Location 1.

Search 333

Insertion	Low	High	Mid	Remarks
1.	0	9	4	X>k[4]
2.	5	9	7	X<k[7]
3.	5	6	5	X>k[5]
4.	6	6	6	X<k[6]
5.	6	5	5	Low>High

Search Value 333 is not found.

Example 2

Let us illustrate binary search on the following 12 elements:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
Elements	4	7	8	9	16	20	24	38	39	45	54	77

If we are searching for x = 4: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 1, high = 2, mid = 3/2 = 1, check 4, ***found***

If we are searching for x = 7: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 1, high = 2, mid = 3/2 = 1, check 4
low = 2, high = 2, mid = 4/2 = 2, check 7, ***found***

If we are searching for x = 8: (This needs 2 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8, ***found***

If we are searching for x = 9: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 4, high = 5, mid = 9/2 = 4, check 9, ***found***

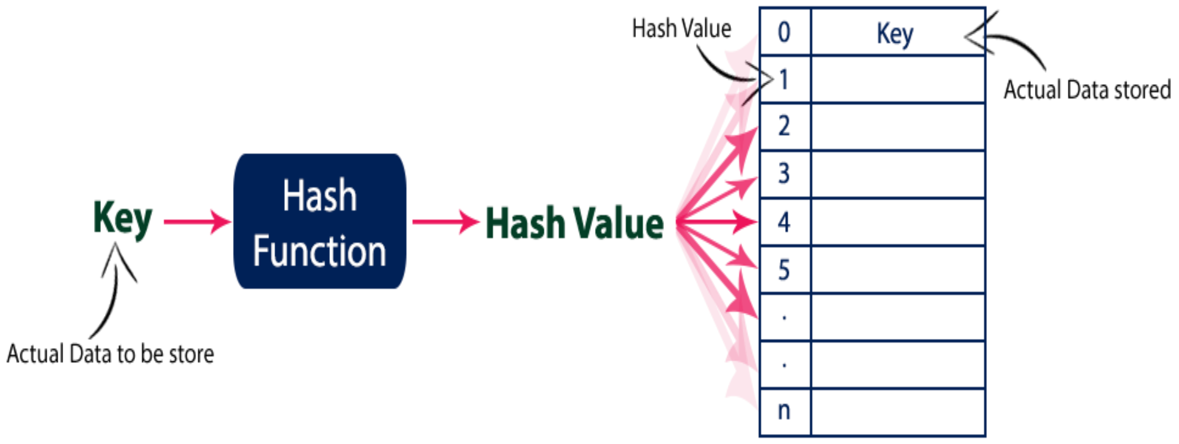
If we are searching for x = 16: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 4, high = 5, mid = 9/2 = 4, check 9
low = 5, high = 5, mid = 10/2 = 5, check 16, ***found***

If we are searching for x = 20: (This needs 1 comparison)
low = 1, high = 12, mid = 13/2 = 6, check 20, ***found***

Chapter 8 Searching

Hashing:

- ***Hashing is the process of indexing and retrieving element in a data structure to provide faster way of finding the element using hash key***. Hash key is a value which provides the index value where the actual data is to likely to be stored in data structure.
- It is the technique of representing longer records by shorter values called ***keys***. Which are generated from a string of text using a mathematical function.
- The keys are placed in a table called hash table where the keys are compared for finding the roots.



Why hashing?

If the array is not sorted, the search might require examining each and all elements of the array. If the array is sorted, we can use the binary search, and therefore reduce the worse-case runtime complexity to O (log n). We could search even faster if we know in advance the index at which that value is located in the array. Suppose we do have that function that would tell us the index for a given value. With this function our search is reduced to just one probe, giving us a ***constant runtime O (1).*** Though, it cannot guarantee a constant runtime for every case. Such a function is called a hash function. A hash function is a function which when given a key, generates an address in the table.

Hash Tables:

- A hash table is a collection of items which are stored in such a way as to make it easy to find them later. Each position of the hash table, often called a slot, can hold an item and is named by an integer value starting at 0. For example, we will have a slot named 0, a slot named1, a slot named 2, and so on.

Chapter 8 Searching

- Hash table is just an array which maps a key (data) into data structure with the help of hash function such that insertion, deletion and search operations can be performed with constant time complexity.
- In hash table, data is stored in an array format, where each data value has its own unique index value. Access to data becomes fast if we know the index of desired data.

Hash Function:

- ***Hash function is a mathematical formula which takes a piece of data as input and outputs an integer (i.e. hash value) which maps the data to a particular index in hash table.***
- The main aim of a hash function is that elements should be uniformly distributed. It produces a unique set of integers within some suitable range in order to reduce the number of collisions.
- In practice, there is no hash function that eliminates collisions completely. A good hash function can only minimize the number of collisions by spreading the elements uniformly throughout the array.

Characteristics of Good Hash Function

A good hash function should have the following characteristics:

- Minimize collision
- Be easy and quick to compute
- Distribute key values evenly in the hash table
- Use all the information provided in the key
- Have a high load factor for a given set of keys

How to Choose Hash Function?

The basic problems associated with the creation of hash tables are:

- An efficient hash function should be designed so that it distributes the index values of inserted objects uniformly across the table.
- An efficient collision resolution algorithm should be designed so that it computes an alternative index for a key whose hash index corresponds to a location previously inserted in the hash table.
- We must choose a hash function which can be calculated quickly, returns values within the range of locations in our table, and minimizes collisions.

Chapter 8 Searching

Different Hash Functions:

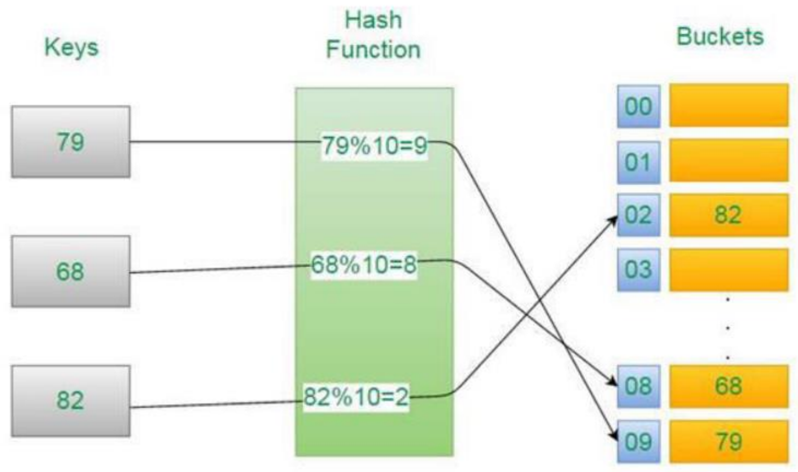
1. Folding Method:

- In this method, the given key is partitioned into subparts k1, K2, k3, k4 ...... kn each of which has the same length as the required address. Now add all these parts together and ignore the carry.
- For example:- if number of buckets be 100 and last address/index be 99, then the given key for which hash code is calculated is divided into parts of two digits from beginning as shown below:
- h(95073) = h(95 + 07 + 3)
    - = h(105) //ignoring the carry = 5
- Example: Given a hash table of 100 locations, calculate the hash value using folding method for keys 5678, 321, and 34567.
- Since there are 100 memory locations to address, we will break the key into parts where each part (except the last) will contain two digits. The hash values can be obtained as shown below:

Key	**5678**	**321**	**34567**
Parts	56 and 78	32 and 1	34,56 and 7
Sum	134	33	97
Hash Value	34(ignore the last carry)	33	97

2. Division Method:

- It is the most simple method of hashing an integer x. ***This method divides x by M (slots available) and then uses the remainder obtained.***
- In this case, the hash function can be given as **z**
- For example: - Let us say apply division approach to find hash value for some values considering number of buckets be 10 as shown below.



3. Mid-Square Method:

- It is a good hash function which works in two steps:

Step 1: Square the value of the key. That is, find k^2.

Step 2: Extract the middle r digits of the result obtained in Step 1.

- The algorithm works well because most or all digits of the key value contribute to the result. This is because all the digits in the original key value contribute to produce the middle digits of the squared value. Therefore, the result is not dominated by the distribution of the bottom digit or the top digit of the original key value.
- In it, the same r digits must be chosen from all the keys. Therefore, the hash function can be given as: **h (k) = s** where s is obtained by selecting r digits from k^2.
- Example: Calculate the hash value for keys 1234 and 5642 using the mid-square method. The hash table has 100 memory locations.
- Note that the hash table has 100 memory locations whose indices vary from 0 to 99. This means that only two digits are needed to map the key to a location in the hash table, so r = 2.
- When k = 1234, k^2 = 1522756, h (1234) = 2
- When k = 5642, k^2 = 31832164, h (5642) = 3
- Observe that the 3rd and 4th digits starting from the right are chosen.

Chapter 8 Searching

Collisions:

- There is possibility that two keys result in same value. ***The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique***.
- Two records cannot be stored in the same location of a hash table normally.

Collision Resolution and Clustering:

- Except direct hashing, none of the hashing methods are one-to- one mapping. Collisions are likely even if we have big table to store keys and hence require collision resolution techniques. Each collision resolution method can be used independently with each hash function. As data are added and collision are resolved, hashing tend to cause data to group within the list.

Collision Resolution Techniques:

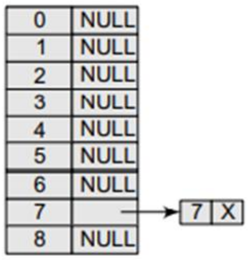
1. **Separate Chaining (Open Hashing)**
2. **Open Addressing (Closed Hashing)**
    a. **Linear Probing**
    b. **Quadratic Hashing**
    c. **Double Hashing**

Separate Chaining (Open Hashing):

- In chaining, each location in a hash table stores a pointer to a linked list that contains all the key values that were hashed to that location. As new collisions occur, ***the linked list grows to accommodate those collisions forming a chain***.
- Chained hash tables with linked lists are widely used due to the simplicity of the algorithms to insert, delete, and search a key. The code for these algorithms is exactly the same as that for inserting, deleting, and searching a value in a single linked list.
- Insert the keys 7, 24, 18, 52, 36, 54, 11, and 23 in a chained hash table of 9 memory locations. Use h (k) = k mod m. In this case, m=9. Initially, the hash table can be given as:

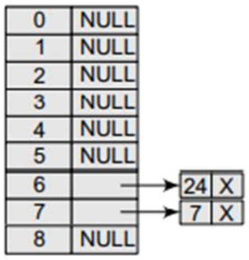
Step 1 Key = 7
h(k) = 7 mod 9
= 7

Create a linked list for location 7 and store the key value 7 in it as its only node.



Step 2 Key = 24
h(k) = 24 mod 9
= 6

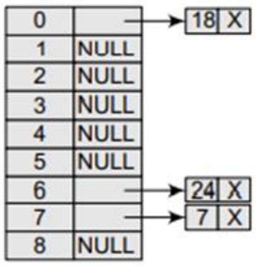
Create a linked list for location 6 and store the key value 24 in it as its only node.



0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	NULL
8	NULL

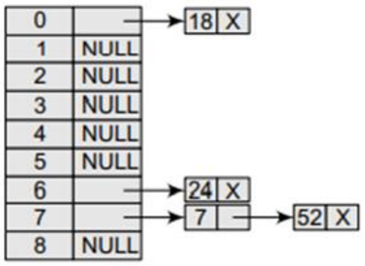
Step 3 Key = 18
h(k)= 18 mod 9 = 0

Create a linked list for location 0 and store the key value 18 in it as its only node.



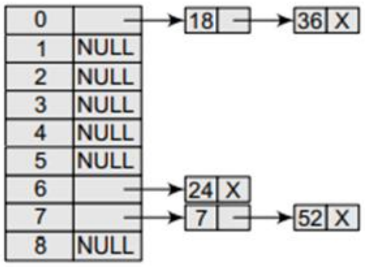
Step 4 Key = 52
h(k) = 52 mod 9 = 7

Insert 52 at the end of the linked list of location 7.



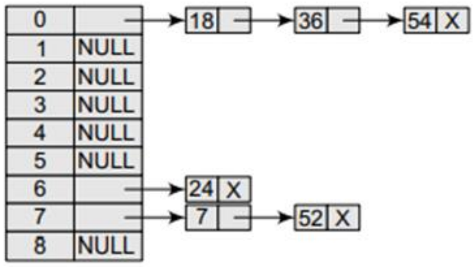
Step 5: Key = 36
h(k)= 36 mod 9 = 0

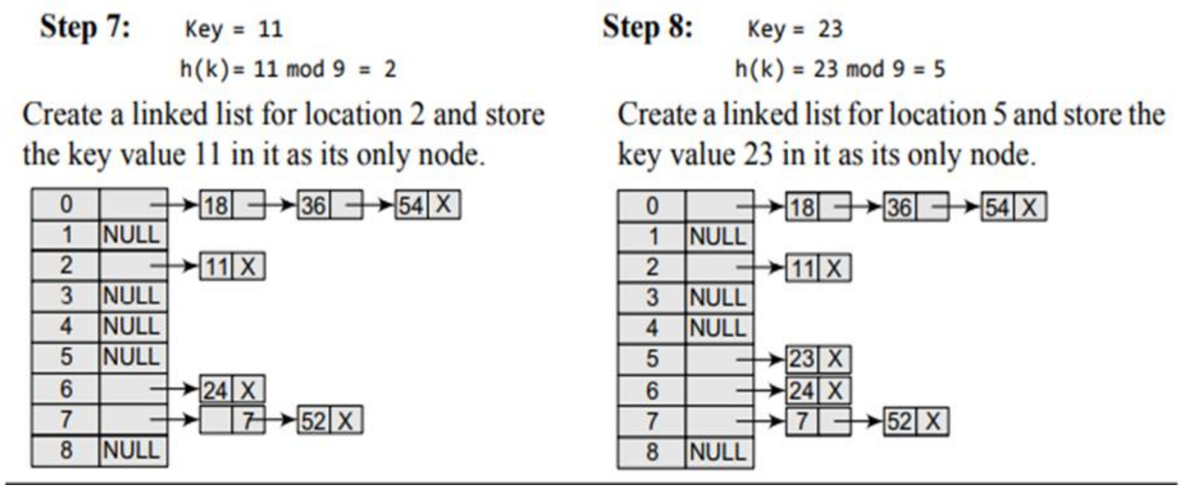
Insert 36 at the end of the linked list of location 0.



Step 6: Key = 54
h(k) = 54 mod 9 = 0

Insert 54 at the end of the linked list of location 0.

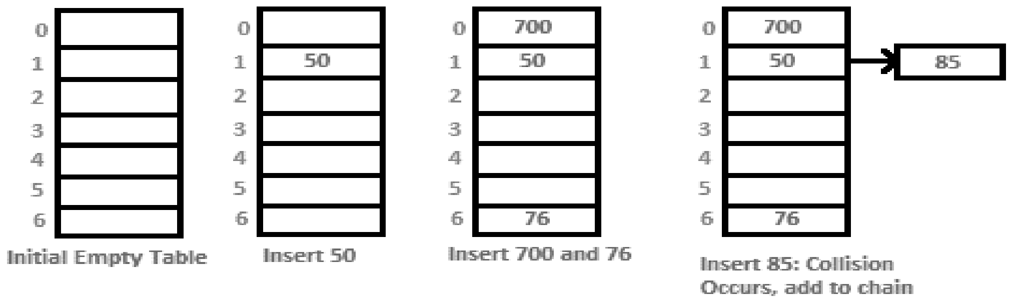


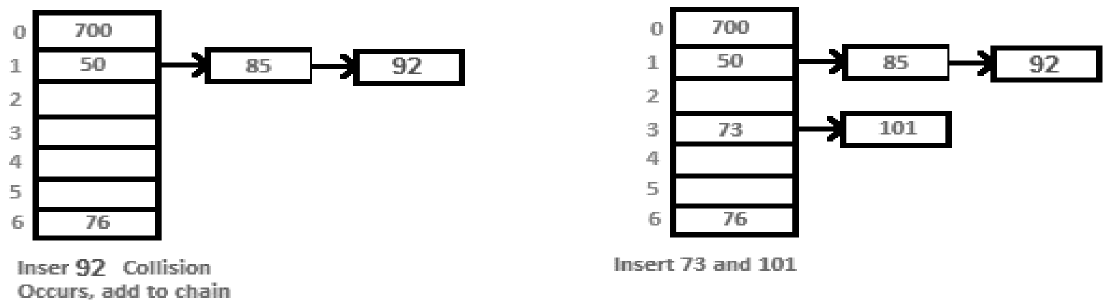

Step 7: Key = 11
h(k) = 11 mod 9 = 2
Create a linked list for location 2 and store the key value 11 in it as its only node.
0 1 NULL 2 3 NULL 4 NULL 5 NULL 6 7 8 NULL
18 36 54 X
11 X
24 X
7 52 X
Step 8: Key = 23
h(k) = 23 mod 9 = 5
Create a linked list for location 5 and store the key value 23 in it as its only node.
0 1 NULL 2 3 NULL 4 NULL 5 6 7 8 NULL
18 36 54 X
11 X
23 X
24 X
7 52 X

Example1 :

Let us consider a simple hash function as "key mod 7" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

Then, h(50) = 50 mod 7 = 1
h(700) = 700 mod 7 = 0
h(76) = 76 mod 7 = 6
h(85) = 85 mod 7 = 1
h(92) = 92 mod 7 = 1
h(73) = 73 mod 7 = 3
h(101) = 101 mod 7 = 3





Advantages:

- Simple to implement.
- Hash table never fills up, we can always add more elements to the chain.
- Less sensitive to the hash function or load factors.
- It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

- Cache performance of chaining is not good as keys are stored using linked list. Open addressing provides better cache performance as everything is stored in same table.
- Wastage of Space (Some Parts of hash table are never used)
- If the chain becomes long, then search time can become O(n) in the worst case.
- Uses extra space for links.

Open Addressing:

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, size of table must be greater than or equal to total number of keys

Insert (k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.

Search (k): Keep probing until slot's key doesn't become equal to k or an empty slot is reached.

Delete (k): Delete operation is interesting. If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as "deleted".

Insert can insert an item in a deleted slot, but search doesn't stop at a deleted slot.

Open Addressing is done following ways:

a. **Linear Probing:**

In linear probing, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also. Let hash(x) be the slot index computed for x using hash function and m be the table size.

$$\boldsymbol{hash\,(\,x\,)\; = \;k\; mod\; m}$$

$$\boldsymbol{hash(\,x, i)\; = \;(\,hash\,(\,x\,) + \;i\,)\,\%\,m\; where\; i\; is\; prob\; or\; collision\; number}$$

- Calculate the hash key. **h´(k) = k** $\boldsymbol{mod\ m}$
- If hashTable [key] is empty, store the value directly, hashTable [key] = data.
- If the hash index already has some value, check for next index.
- $\boldsymbol{h(k, i) = (h'(k) + i) mod\ m;}$
- If the next index is available hashTable [key], store the value. Otherwise try for next index.
- Do the above process till we find the space.

Example 1:

Consider a hash table of size 10. Using linear probing, insert the keys 72, 27, 36, 24, 63, 81, 92 and 101 into the table.

- Let h'(k) = k mod m, m = 10

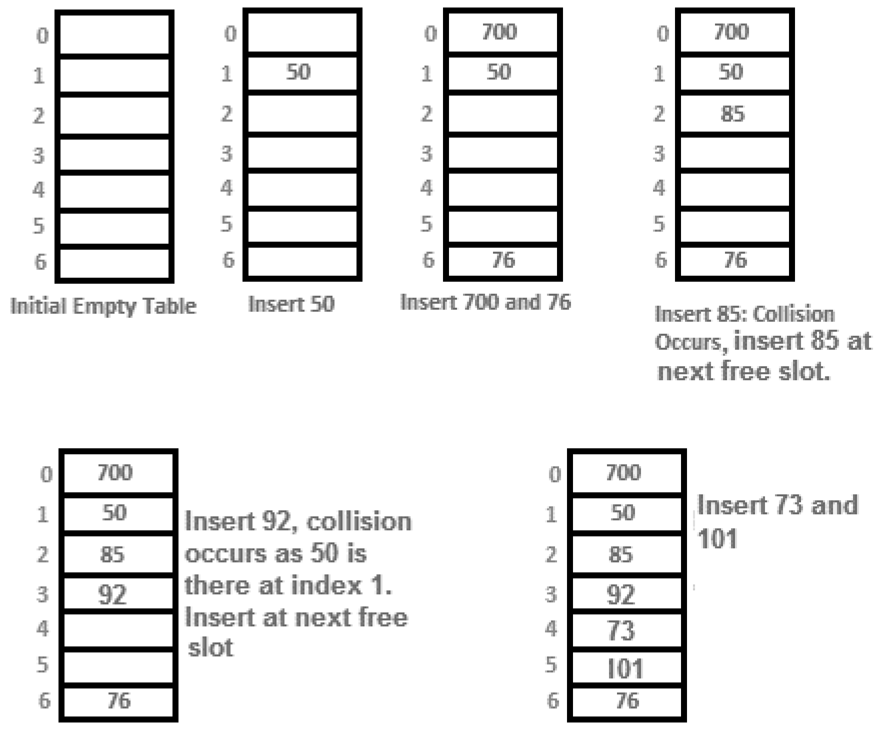
k	**h(k,i)= (h'(k) + i) mod 10**
72	h(72,0)=(72 mod 10 + 0) mod 10=2 mod 10 =2
27	h(27,0)=(27 mod 10 + 0) mod 10=7 mod 10 =7
36	h(36,0)=(36 mod 10 + 0) mod 10=6 mod 10 =6
24	h(24,0)=(24 mod 10 + 0) mod 10=4 mod 10 =4
63	h(63,0)=(63 mod 10 + 0) mod 10=3 mod 10 =3
81	h(81,0)=(81 mod 10 + 0) mod 10=1 mod 10 =1
92	h(92,0)=(92 mod 10 + 0) mod 10=2 mod 10 =2(A[2] is occupied) Then i=1, h(92,1)=(92 mod 10 + 1) mod 10=3 mod 10 =3(A[3] is occupied) Then i=2, h(92,2)=(92 mod 10 + 2) mod 10=4 mod 10 =4(A[4] is occupied) Then i=3, h(92,3)=(92 mod 10 + 3) mod 10=5 mod 10 =5
101	h(101,0)=(101 mod 10 + 0) mod 10=1 mod 10 =1 (A[1] is occupied) Then i=1, h(101,1)=(101 mod 10 + 1) mod 10=2 mod 10 =2(A[2] is occupied) Repeat process until i=7.

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
	81	**72**	**63**	**24**	**92**	**36**	**27**	**101**	

Example 2:

Let us consider a simple hash function as “key mod 7” and sequence of keys

as 50, 700, 76, 85, 92, 73, 101. Use linear probling and insert the keys into table.


0
1
2
3
4
5
6
Initial Empty Table
50
Insert 50
700
50
76
Insert 700 and 76
700
50
85
76
Insert 85: Collision Occurs, insert 85 at next free slot.
700
50
85
92
76
Insert 92, collision occurs as 50 is there at index 1. Insert at next free slot
700
50
85
92
73
101
76
Insert 73 and 101

Advantages:

- No extra space

Disadvantages:

- Searching Difficult
- Primary Clustering
- Secondary Clustering

b. **Quadratic Hashing:**

- In this technique, if a value is already stored at a location generated by h(k), then the following hash function is used to resolve the collision:

$$\boldsymbol{h(k,i) = [h'(k) + i^2]\,mod\ m}\ where\ m\ is\ the\ size\ of\ the\ hash\ table,$$

$$\boldsymbol{or}$$

$$\boldsymbol{h(k,i) = (h'(k) + c1 * i + c2 * i^2)\%\, m}$$

$$\boldsymbol{h'(k) = (k\ mod\ m)}, i\ is\ the\ probe\ number\ that\ varies\ from\ 0\ to\ m-1.$$

- It eliminates the primary clustering phenomenon of linear probing because instead of doing a linear search, it does a quadratic search.

$let\ hash(x)be\ the\ slot\ index\ computed\ using\ hash\ function.$

$If\ slot\ hash(x)\ \%\ S\ is\ full, then\ we\ try\ (hash(x)\ +\ 1*1)\ \%\ S$

$If\ (hash(x) +\ 1*1)\%\ S\ is\ also\ full, then\ we\ try\ (hash(x) +\ 2*2)\%\ S$

$If\ (hash(x)\ +\ 2*2)\ \%\ S\ is\ also\ full, then\ we\ try\ (hash(x)\ +\ 3*3)\ \%\ S$

……………………………………………

…………………………………………..

Example 1:

Consider a hash table of size 10. Using quadratic probing, insert the keys 72, 27, 36, 24, 63, 81, and 101 into the table.

Let h' (k) = k mod m,

m = 10

k	h(k,i)=(h'(k)+i^2) mod 10
72	h(72,0)=(72 mod 10 + 0^2) mod 10=2 mod 10 =2
27	h(27,0)=(27 mod 10 + 0^2) mod 10=7 mod 10 =7
36	h(36,0)=(36 mod 10 + 0^2) mod 10=6 mod 10 =6
24	h(24,0)=(24 mod 10 + 0^2) mod 10=4 mod 10 =4
63	h(63,0)=(63 mod 10 + 0^2) mod 10=3 mod 10 =3
81	h(81,0)=(81 mod 10 + 0^2) mod 10=1 mod 10 =1
101	h(101,0)=(101 mod 10 + 0^2) mod 10=1 mod 10 =1 [is occupied] h(101,1)=(101 mod 10 + 1^2) mod 10=2 mod 10 =2 [is occupied] h(101,2)=(101 mod 10 + 2^2) mod 10=5 mod 10 =5

0	1	2	3	4	5	6	7	8	9
	81	72	63	24	101	36	27		

Example 2:

Let us consider a simple hash function as "key mod 10" and sequence of keys as 42, 16, 91, 33, 18, 27, 36.

k	h(k,i)=(h'(k)+i^2) mod 10
42	h(42,0)=(42 mod 10 + 0^2) mod 10=2 mod 10 =2
16	h(16,0)=(16 mod 10 + 0^2) mod 10=6 mod 10 =6
91	h(91,0)=(91 mod 10 + 0^2) mod 10=1 mod 10 =1
33	h(33,0)=(33mod 10 + 0^2) mod 10=3 mod 10 =3
18	h(18,0)=(18mod 10 + 0^2) mod 10=8 mod 10 =8
27	h(27,0)=(27mod 10 + 0^2) mod 10=7 mod 10 =7
36	h(36,0)=(36 mod 10 + 0^2) mod 10=6 mod 10 =6 [is occupied] h(36,1)=(36 mod 10 +1^2) mod 10=7 mod 10 =7 [is occupied] h(36,2)=(36 mod 10 + 2^2) mod 10=10 mod 10 =0

0	1	2	3	4	5	6	7	8	9
36	91	42	33			16	27	18	

Advantages:

- No extra space
- Primary Clustering Resolved

Disadvantages:

- Secondary Clustering
- No guarantee of finding slot

c. **Double Hashing:**

- It uses one hash value and then repeatedly steps forward an interval until an empty location is reached. The interval is decided using a second, independent hash function, hence the name double hashing.
- In double hashing, we use another hash function hash2(x) and look for $i * hash2(x)$ slot in i'th rotation.

$$\boldsymbol{hash1(\ x\ ) =\ k\ mod\ m}$$

$$\boldsymbol{hash\ (\ x, i\ ) = \left(hash1\ (\ x\ ) + i * hash2(x)\right) mod\ m}$$

$$\boldsymbol{where\ i\ is\ the\ probe\ number\ that\ varies\ from\ 0\ to\ m-1, and}$$

$$\boldsymbol{hash2(x) =\ R - k\ mod\ R\ , R\ is\ a\ prime\ number\ less\ than\ the\ table\ size\ m}$$

$Let\ hash(x)\ be\ the\ slot\ index\ computed\ using\ hash\ function.$

$If\ slot\ hash(x)\ \%\ m\ is\ full, then\ we\ try\ (hash(x)\ +\ 1 * hash2(x))\ \%\ m$

$If\ \left(hash(x) +\ 1 * hash2(x)\right)\%\ m\ is\ also\ full, then\ we\ try\ \left(hash(x) +\ 2 * hash2(x)\right)\%\ m$

$If\ \left(hash(x) +\ 2 * hash2(x)\right)\%\ m\ is\ also\ full, then\ we\ try\ \left(hash(x) +\ 3 * hash2(x)\right)\ \%\ m$

....

....

Example 1:

Consider a hash table of size = 10. Using double hashing, insert the keys 72, 27, 36, 24, 63, 81, 92, and 101 into the table. Take h1 = (k mod 10) and h2 = (k mod 8). [Let m = 10]

k	$h(k,i)=[h_1(k)+ih_2(k)]\text{mod } m$
72	h(72,0)=[72 mod 10 + 0(72 mod 8)]mod 10=2
27	h(27,0)=[27 mod 10 + 0(27 mod 8)]mod 10=7
36	h(36,0)=[36 mod 10 + 0(36 mod 8)]mod 10=6
24	h(24,0)=[24 mod 10 + 0(24 mod 8)]mod 10=4
63	h(63,0)=[63 mod 10 + 0(63 mod 8)]mod 10=3
81	h(81,0)=[81 mod 10 + 0(81 mod 8)]mod 10=1
92	h(92,0)=[92 mod 10 + 0(92 mod 8)]mod 10=2 [Collision since A[2] is occupied.] h(92,1)=[92 mod 10 + 1(92 mod 8)]mod 10=(2+4) mod 10 = 6 [Collision since A[6] is occupied.] h(92,2)=[92 mod 10 + 2(92 mod 8)]mod 10=(2+2*4) mod 10= 0
101	h(101,0)=[101 mod 10 + 0(101 mod 8)]mod 10=1[Collision since A[1] is occupied.] h(101,1)=[101 mod 10 + 1(101 mod 8)]mod 10=6[Collision since A[6] is occupied.] h(101,2)=[101 mod 10 + 2(101 mod 8)]mod 10=1[Collision since A[1] is occupied.] Repeat the entire process until a vacant location is found. We will see that we have to probe many times to insert the key 101 in the hash table.

0	1	2	3	4	5	6	7	8	9
92	**81**	**72**	**63**	**24**		**36**	**27**		

Advantages:

- No extra space
- No primary Clustering
- No Secondary Clustering

Disadvantages:

- Requires more computation time as two hash functions need to be computed.

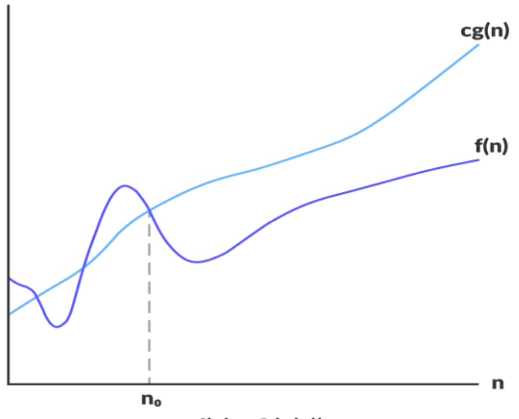
Chapter 9 Growth Functions

- Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.
- When it comes to analyzing the complexity of any algorithm in terms of time and space, we can never provide an exact number to define the time required and the space required by the algorithm, instead we express it using some standard notations, also known as Asymptotic Notations.
- Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.
- For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case. But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case. When the input array is neither sorted nor in reverse order, than it takes average time.
- Types of Asymptotic Notation
    1. **Big-O Notation (O)** – Big O notation specifically describes worst case scenario.
    2. **Omega Notation (Ω)** – Omega (Ω) notation specifically describes best case scenario.
    3. **Theta Notation (θ)** – This notation represents the average complexity of an algorithm.

Big-O Notation (O):

Big O notation specifically describes scenario. It represents the **upper bound** running time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

$$O(g(n)) = \{ f(n) : \textit{there exist positive constants } c \textit{ and } n_0 \textit{ such that } \mathbf{0} \le \boldsymbol{f(n)} \le \boldsymbol{c * g(n)} \textit{ for all } n >= n_0 \}$$



f(n) = O(g(n))

Example:

$f(n) = 3n + 2$

$g(n) = n$

$can\ we\ write\ f(n) = O(g(n))?$
$Condition\ to\ be\ satisfied\ is\ 0 <= f(n) <= c * g(n)$

$0 <= 3n + 2 <= c * n$

$0 <= 3n + 2 <= 4 * n\ for\ all\ n0 >= 2$

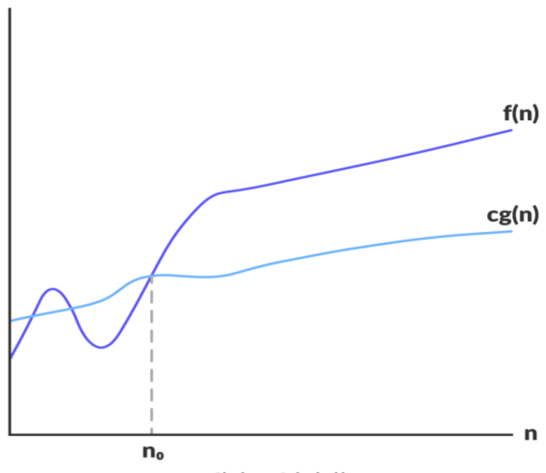
$So, f(n) = O(g(n))$

$3n + 2 = O(n)$

Big-Omega Notation (Ω):

Omega notation represents the **lower bound** of the running time of an algorithm. Thus, it provides the **best case** complexity of an algorithm.

$$\Omega(g(n)) = \{ f(n): there\ exist\ positive\ constants\ c\ and\ n_0$$
$$such\ that\ \mathbf{0} \leq \boldsymbol{cg(n)} \leq \boldsymbol{f(n)}$$
$$for\ all\ n \geq n_0 \}$$



f(n) = Ω(g(n))

Example:

$f(n) = 3n + 2$

$g(n) = n$

$can\ we\ write\ f(n) = \Omega\ (g(n))?$

$Solution:$

$Condition\ to\ be\ satisfied\ is\ 0\ <=\ c * g(n)\ <=\ f(n)$

$0\ <=\ c * n\ <=\ 3n + 2$

$0\ <=\ n\ <=\ 3n + 2\ for\ all\ n0 >= 1$

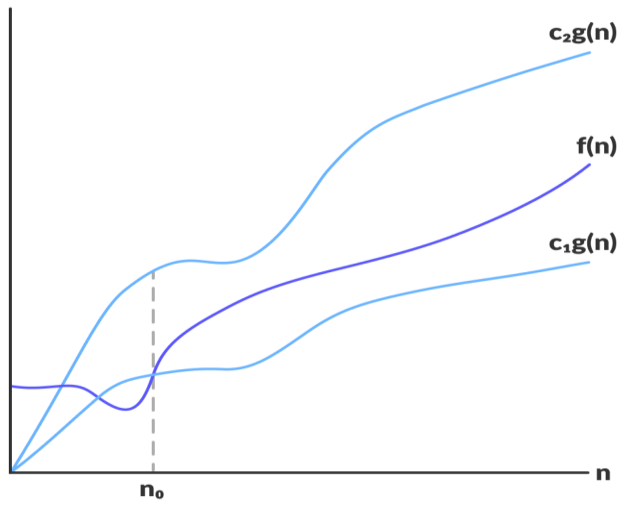
$So, f(n) = \Omega\ (g(n))$

$3n + 2 = \Omega\ (n)$

Theta Notation (θ):

This notation describes both upper bound and lower bound of an algorithm so we can say that it defines **exact asymptotic behavior**. In the real case scenario the algorithm not always run on best and worst cases, the average running time lies between best and worst and can be represented by the theta notation. Thus, it provides the **average case** complexity of an algorithm.

$$\Theta(g(n)) = \{\ f(n): there\ exist\ positive\ constants\ c1, c2\ and\ n_0$$
$$such\ that\ \mathbf{0}\ \leq\ \boldsymbol{c1g(n)} \leq\ \boldsymbol{f(n)} \leq\ \boldsymbol{c2g(n)}$$
$$for\ all\ n\ \geq n_0\ \}$$



f(n) = Θ(g(n))

Chapter 9 Growth Functions

Example:

$f(n) = 3n + 2$

$g(n) = n$

$can\ we\ write\ f(n) = \theta\ (g(n))?$

$Solution:$

$Condition\ to\ be\ satisfied\ is\ c1 * g(n)\ <=\ f(n)\ <=\ c2 * g(n)$

$c1n\ <=\ 3n + 2\ <=\ c2n$

$n\ <=\ 3n + 2\ <=\ 5n\ for\ all\ n0 >= 1$

$So, f(n) =\ \theta\ (g(n))$

$3n + 2 =\ \theta\ (n)$

Little o Notation:

Little o notation is used to describe an upper bound that cannot be tight. In other words, **loose upper bound** of f(n). It is formally defined as:

$$O(g(n))\ = \{\, f(n) :\ for\ any\ real\ constants\ c > 0, there\ exists\ an\ integer\ constant\ n_0 \geq 1, such\ that\ \mathbf{0}\ \leq\ \boldsymbol{f(n)} <\ \boldsymbol{c * g(n)}\ for\ all\ n\ >=\ n_0\ \}$$

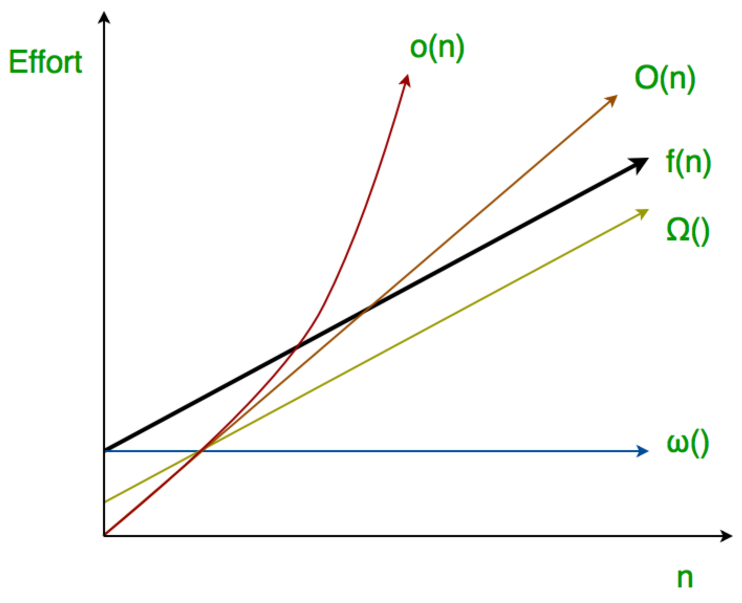
Using mathematical relation, we can say that f(n) = o(g(n)) means, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

Little ω notation:

$$\omega(g(n))\ = \{\, f(n): for\ any\ \ real\ constants\ c > 0, there\ exists\ an\ integer\ constant\ \ n_0\ \geq 1, such\ that\ \mathbf{0}\ \leq\ \boldsymbol{c * g(n)} <\ \boldsymbol{f(n)}\ for\ all\ n\ \geq\ n_0 \}$$

Note: Omega (ω) is a rough estimate of the order of the growth whereas Big Omega (Ω) may represent exact order of growth.

Chapter 9 Growth Functions



Chapter 10 Graphs

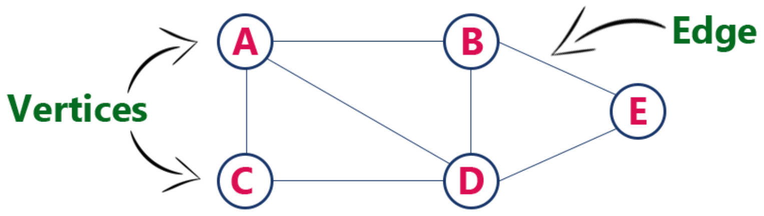
Representation and Application:

Graph is a non-linear data structure. It contains a set of points known as nodes (or vertices) and a set of links known as edges (or Arcs). Here edges are used to connect the vertices.

A graph G can be defined as an ordered set G (V, E) where V (G) represents the finite and non-empty set of vertices and E(G) represents the set of edges which are used to connect these vertices.

Example:

The following is a graph with 5 vertices and 6 edges.
$This\ graph\ G\ can\ be\ defined\ as\ G\ =\ (V\ ,E\ )$
$Where\ V\ =\ \{A, B, C, D, E\}\ and\ E\ =\ \{(A, B), (A, C)(A, D), (B, D), (C, D), (B, E), (E, D)\}$.



Graph terminologies:

Vertex: An individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.

Edge: An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (starting Vertex, ending Vertex).In above graph, the link between vertices A and B is represented as (A, B).

Adjacent Vertices: Two vertices **u** and **v** in an undirected graph **G** are called adjacent in **G** if **u** and **v** are endpoints of an edge **e** of **G**. Such an edge **e** is called incident with the vertices **u** and **v** and **e** is said to connect **u** and **v**.

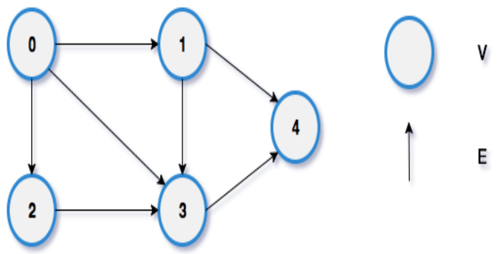
Degree of Vertex: The degree of a vertex in an undirected graph is the number of edges incident with it. A loop at a vertex contributes twice to the degree of that vertex. The degree of the vertex v is denoted by deg(v). A vertex of degree zero is called isolated. A vertex with degree one is called pendant. Example: deg (A) = 3, deg (B) =3 deg (E)=2.

In-degree: The in-degree of a vertex v is the number of edges that are incoming - towards v (head of edge).It is denoted by **deg– (u).**

Out-degree: the out-degree of a vertex v is the number of edges that are outgoing from v (tail of edge).It is denoted by **deg+ (u).**

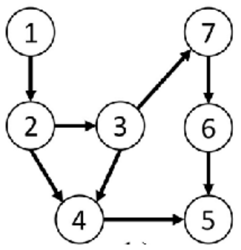
Chapter 10 Graphs

Node (u)	Deg-(u)	Deg+(u)
0	0	3
1	1	2
2	1	1
3	3	1
4	2	0

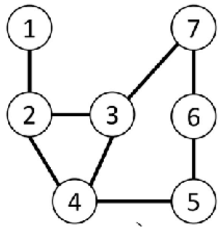


Types of Graph:

- **Directed Graph:**
    - In a directed graph, the connection between two nodes is one-directional.
    - It is a graph in which each edge has a direction to its successor.
    - It is a graph with only directed edges.



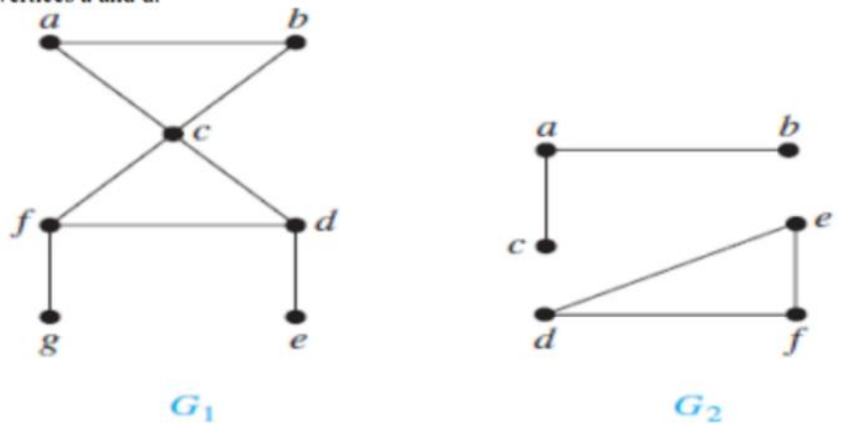
- **Undirected Graph:**
    - In an undirected graph, all connections are bi-directional.
    - It is a graph in which there is no direction on the edges. The flow between two vertices can go in either direction.



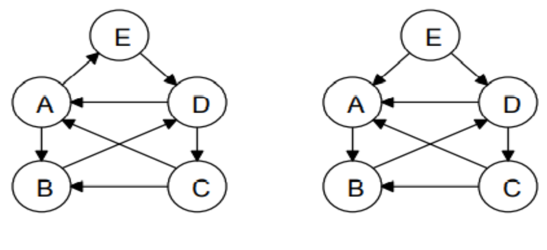
- **Connected Graph**: An undirected graph is called connected if there is a path between every pair of distinct vertices of the graph.

- **Not-Connected Graph**: An undirected graph that is not connected is called disconnected

Example: G1 is the connected graph because for every pair of distinct vertices there is a path between them and G2 is the not-connected graph because there is no path between vertices a and d.



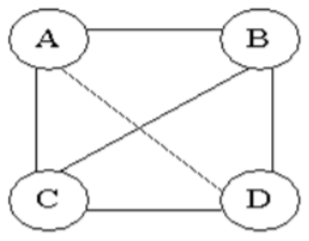
- **Strongly Connected Graph**: A directed graph is strongly connected if there is a path from A to B and from B to A whenever A and B are vertices in the graph.
- **Weakly Connected Graph:** A directed graph is weakly connected if there is a path between every two vertices in the underlying undirected graph. That is, a directed graph is weakly connected if and only if there is always a path between two vertices when the directions of the edges are disregarded.



Strongly Connected　　Weakly Connected

Complete Graph:

- A graph in which any V node is adjacent to all other nodes present in the graph is known as a complete graph. An undirected graph contains the edges that are equal to $edges\ =\ n(n-1)/2$ where n is the number of vertices present in the graph. The following figure shows a complete graph.



A complete graph.

Regular Graph:

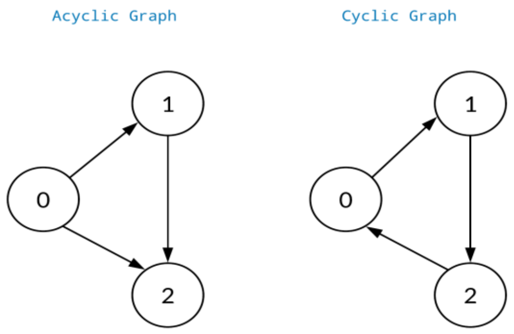
Chapter 10 Graphs

- It is the graph in which nodes are adjacent to each other, i.e., each node is accessible from any other node.

Cycle Graph:

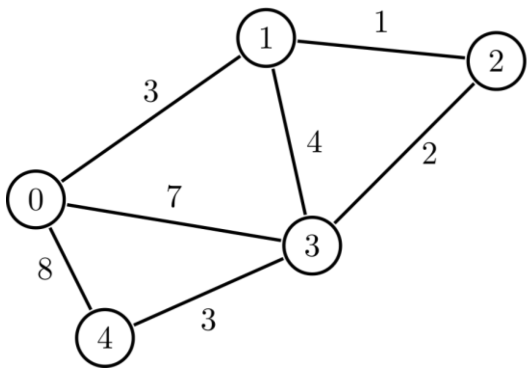
- A graph having cycle is called cycle graph. In this case the first and last nodes are the same. A closed simple path is a cycle.

Acyclic Graph: A graph without cycle is called acyclic graphs.



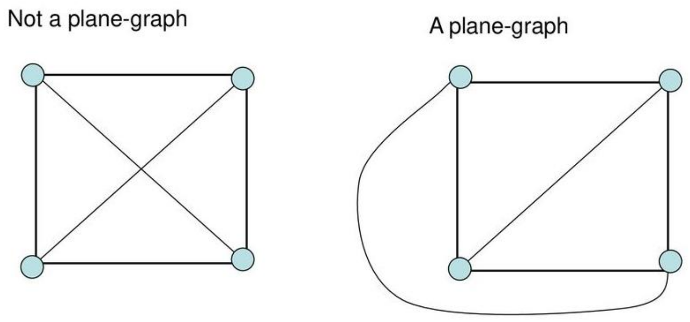
Weighted Graph:

- Graphs that have a number assigned to each edge are called weighted graphs.
- In a weighted graph, each edge has an associated numerical value, called the weight of the edge. Edge weights may represent distances, costs, etc.
- Example: In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



Planar Graph:

- A graph is called planar if it can be drawn in the plane without any edges crossing, i.e., it can be drawn on the plane in such a way that its edges intersect only at their endpoints. In other words, it can be drawn in such a way that no edges cross each other.

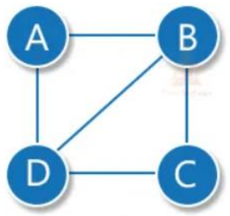


Representation of Graphs:

Chapter 10 Graphs

1. **Adjacency matrix :**

- In it, we have a matrix of order n*n where n is the number of nodes in the graph. The matrix represents the mapping between various edges and vertices.
- In the matrix, each row and column represents a vertex. The values determine the presence of edges.
- Let Aij represents each element of the adjacency matrix. Then,
- For an undirected graph, the value of Aij is 1 if there exists an edge between i and j. Otherwise, the value of Aij is 0.

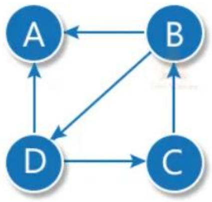


Undirected graph

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 0 & 1 \\ B & 1 & 0 & 1 & 1 \\ C & 0 & 1 & 0 & 1 \\ D & 1 & 1 & 1 & 0 \end{array}$$

Adjacency matrix

- For a directed graph, the value of Aij is 1 only if there is an edge from i to j i.e. **i** is the initial node and **j** is the terminal node.



Directed graph

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 0 & 0 & 0 \\ B & 1 & 0 & 0 & 1 \\ C & 0 & 1 & 0 & 0 \\ D & 1 & 0 & 1 & 0 \end{array}$$

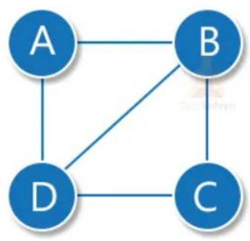
Adjacency matrix

- The time complexity of the adjacency matrix is $O(n^2)$.

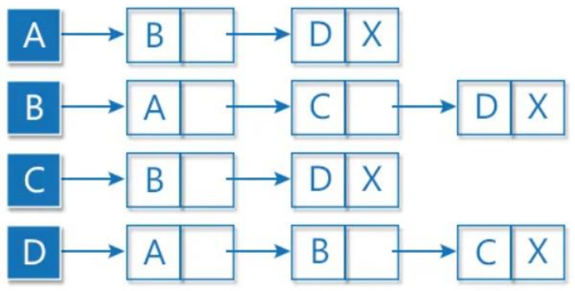
2. **Adjacency list:**

- The adjacency list is an array of linked lists where the array denotes the total vertices and each linked list denotes the vertices connected to a particular node.
- In a linked list, the most important component is the pointer named 'Head' because this single pointer maintains the whole linked list. For linked list representation, we will have total pointers equal to the number of nodes in the graph.
- For an undirected graph, we will link all the edges in the list that are connected to a node as shown:

Chapter 10 Graphs

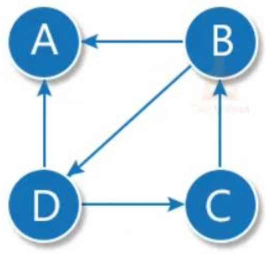


Undirected graph

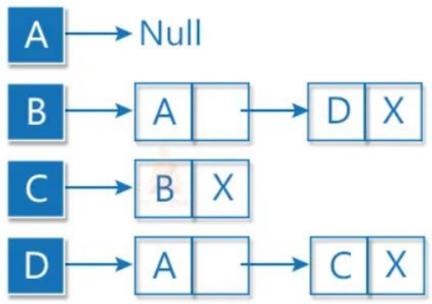


Adjacency List

- In a directed graph, we will link only the initial nodes in the list as shown:



Directed graph



Adjacency List

Applications of Graph

- In Computer science graphs are used to represent the flow of computation.
- Google maps uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.
- Computer networks: Local area network, Internet, Web
- Facebook’s Friend suggestion algorithm uses graph theory. Facebook is an example of undirected graph.
- In World Wide Web, web pages are considered to be the vertices. There is an edge from a page u to other page v if there is a link of page v on page u. This is an example of Directed graph. It was the basic idea behind Google Page Ranking Algorithm.

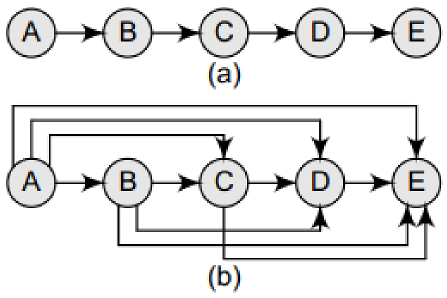
Chapter 10 Graphs

Transitive closure:

- Transitive Closure it the reachability matrix to reach from vertex u to vertex v of a graph. One graph is given, we have to find a vertex v which is reachable from another vertex u, for all vertex pairs (u, v).
- It states if there is a path from vertex a to b then there should be an edge from a to b.
- For finding transitive closure of a graph
- Add an edge from a to c if there exists a path from a to b and b to c
- Repeat this process of adding edge until no new edges are added.
- Hence, it can be defined as, If G=(V,E) in a graph then its transitive closure can be defined as G*=(V,E*) where E*={(Vi,Vj): there exists a path from Vi to Vj in G}

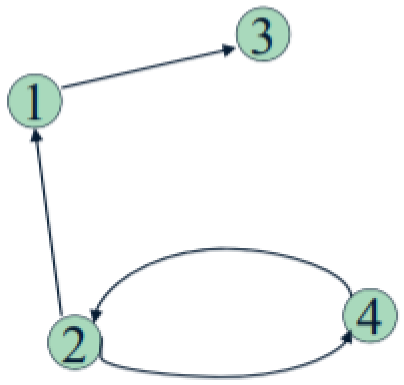
Transitive Closure of a Directed Graph

- Given a directed graph, find out if a vertex j is reachable from another vertex i for all vertex pairs (i, j) in the given graph. Here reachable mean that there is a path from vertex i to j. The reach-ability matrix is called the transitive closure of a graph.
- Transitive closure is also stored as a matrix T, so if T[1][5] = 1, then node 5 can be reached from node 1 in one or more hops.



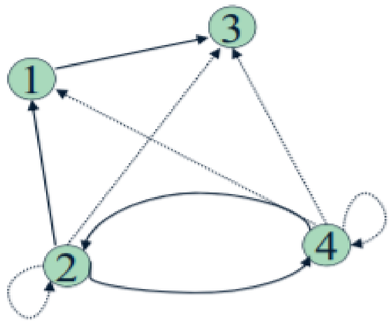
(a) A graph G and its (b) transitive closure G*

- For example, consider below graph



V	1	2	3	4
1	0	0	1	0
2	1	0	0	1
3	0	0	0	0
4	0	1	0	0

- Transitive closure of above graphs is



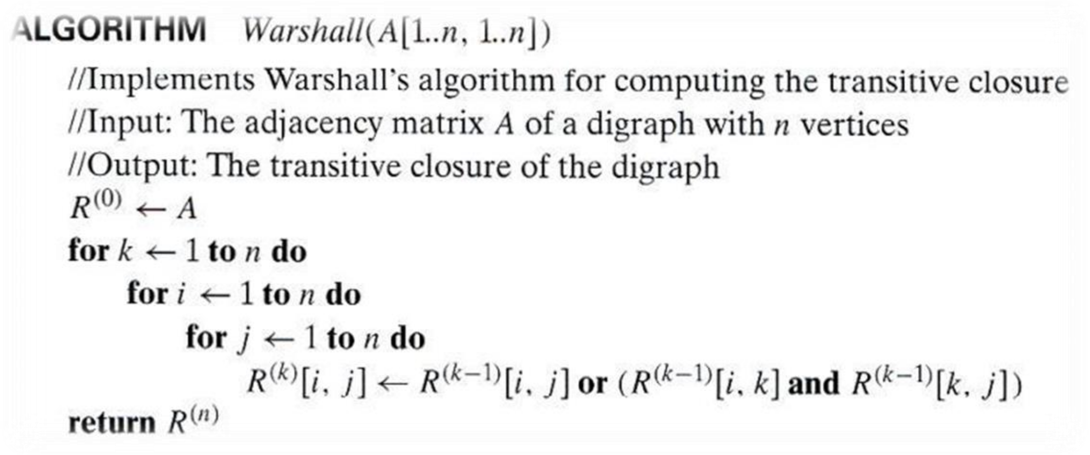
V	1	2	3	4
1	0	0	1	0
2	1	1	1	1
3	0	0	0	0
4	1	1	1	1

Warshall's algorithm for finding transitive closure from diagraph:

- Recurrence relating elements $R^{(k)}$ to elements of $R^{(k-1)}$ is:

$$R^{(k)}[i,j] = R^{(k-1)}[i,j] \text{ or } (R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j])$$

- It implies the following rules for generating $R^{(k)}$ from $R^{(k-1)}$:
  Rule 1 If an element in row i and column j is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$
  Rule 2 If an element in row i and column j is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ if and only if the element in its row i and column k and the element in its column j and row k are both 1's in $R^{(k-1)}$.

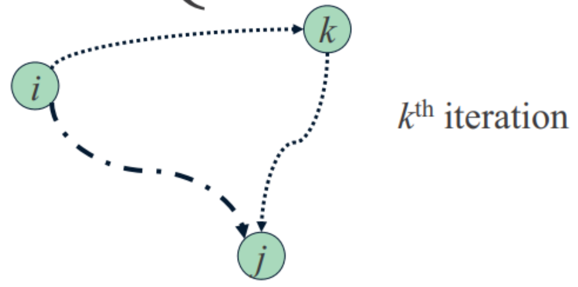


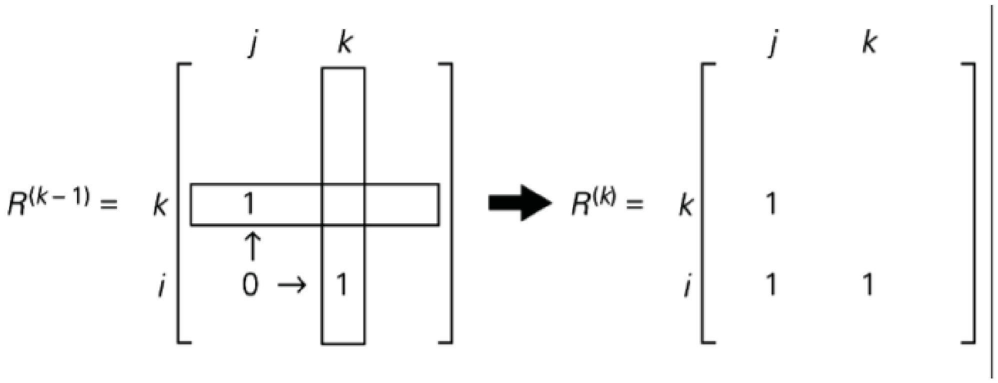
```
ALGORITHM    Warshall(A[1..n, 1..n])
    //Implements Warshall's algorithm for computing the transitive closure
    //Input: The adjacency matrix A of a digraph with n vertices
    //Output: The transitive closure of the digraph
    $R^{(0)} \leftarrow A$
    for $k \leftarrow 1$ to $n$ do
        for $i \leftarrow 1$ to $n$ do
            for $j \leftarrow 1$ to $n$ do
                $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ or $(R^{(k-1)}[i, k]$ and $R^{(k-1)}[k, j])$
    return $R^{(n)}$
```

The main idea of these graphs is described as follows:

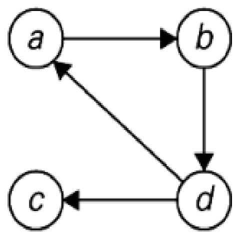
- The vertices i, j will be contained a path if
- The graph contains an edge from i to j; or
- The graph contains a path from i to j with the help of vertex 1; or
- The graph contains a path from i to j with the help of vertex 1 and/or vertex 2; or
- The graph contains a path from i to j with the help of any other vertices.

$$R^{(k)}[i,j] = \begin{cases} R^{(k-1)}[i,j] & \text{(path using just } 1,\ldots,k\text{-}1) \\ \quad\text{or} & \\ (R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j]) & \text{(path from } i \text{ to } k \text{ and from } k \text{ to } i \text{ using just } 1,\ldots,k\text{-}1) \end{cases}$$





Example 1: Fig: Rule for changing zero's in Warshall's Algorithm



(a)

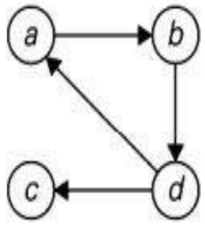
$$A = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

(b)

$$T = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 1 & 1 & 1 & 1 \\ b & 1 & 1 & 1 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

(c)

Figure. (a) Digraph. (b) Its adjacency matrix. (c) Its transitive closure.



$$R^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

Ones reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

$$R^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & \mathbf{1} & 1 & 0 \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex *a* (note a new path from *d* to *b*); boxed row and column are used for getting $R^{(2)}$.

$$R^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & \mathbf{1} \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & \mathbf{1} \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$$R^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 1 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a*, *b*, and *c* (no new paths); boxed row and column are used for getting $R^{(4)}$.

$$R^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & \mathbf{1} & 1 & \mathbf{1} & 1 \\ b & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., *a*, *b*, *c*, and *d* (note five new paths).

Example 2:

Apply Warshall's algorithm to find the transitive closure of the digraph defined by the following adjacency matrix.

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Applying Warshall's algorithm yields the following sequence of matrices

$$R^{(0)} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$R^{(1)} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$R^{(2)} = \begin{bmatrix} 0 & 1 & \mathbf{1} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

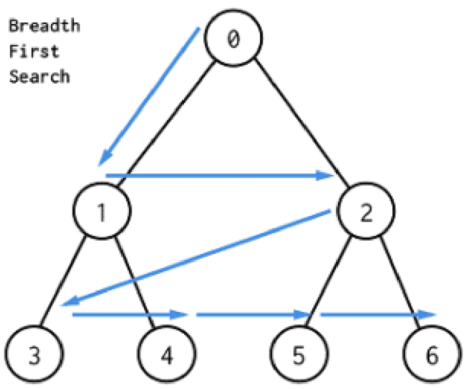
$$R^{(3)} = \begin{bmatrix} 0 & 1 & 1 & \mathbf{1} \\ 0 & 0 & 1 & \mathbf{1} \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

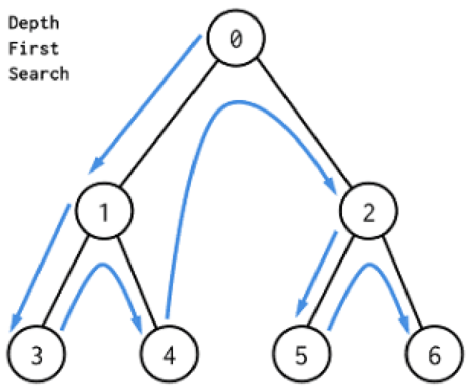
$$R^{(4)} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = T$$

Chapter 10 Graphs

Traversing a Graph:

- Graph traversal is a technique used for searching a vertex in a graph.
- The graph traversal is also used to decide the order of vertices is visited in the search process.
- A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.
- There are two graph traversal techniques and they are as follows:
    1. DFS (Depth First Search)
    2. BFS (Breadth First Search)





DFS (Depth First Search):

DFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal. We use the following steps to implement DFS traversal...

Step 1 - Define a Stack of size total number of vertices in the graph.

Step 2 - Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.

Step 3 - Visit any one of the non-visited adjacent vertices of a vertex which is at the top of stack and push it on to the stack.

Step 4 - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.

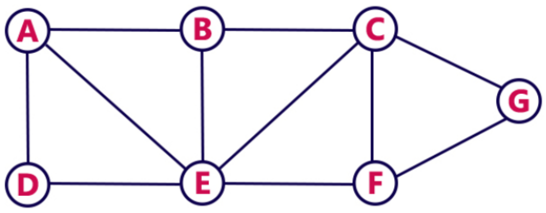
Step 5 - When there is no new vertex to visit then use back tracking and pop one vertex from the stack.

Step 6 - Repeat steps 3, 4 and 5 until stack becomes Empty.

Step 7 - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

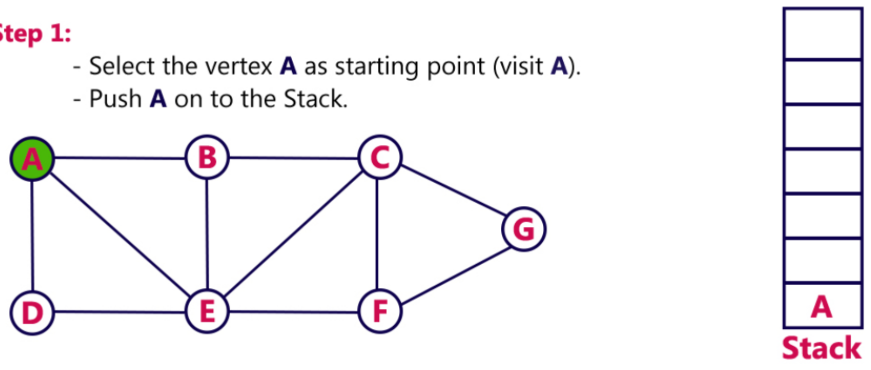
Example 1:

Consider the following example graph to perform DFS traversal



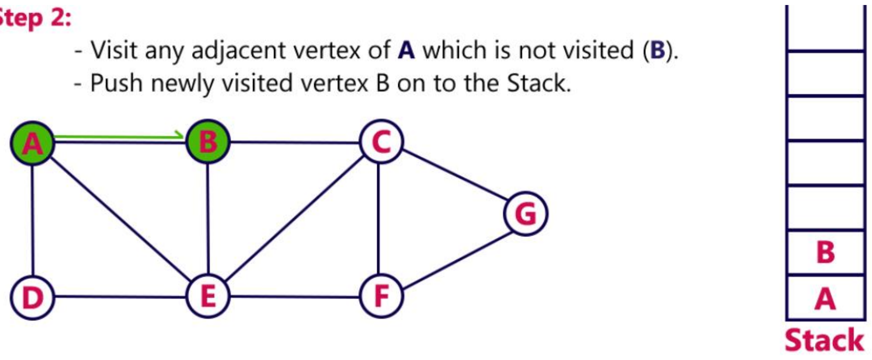
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



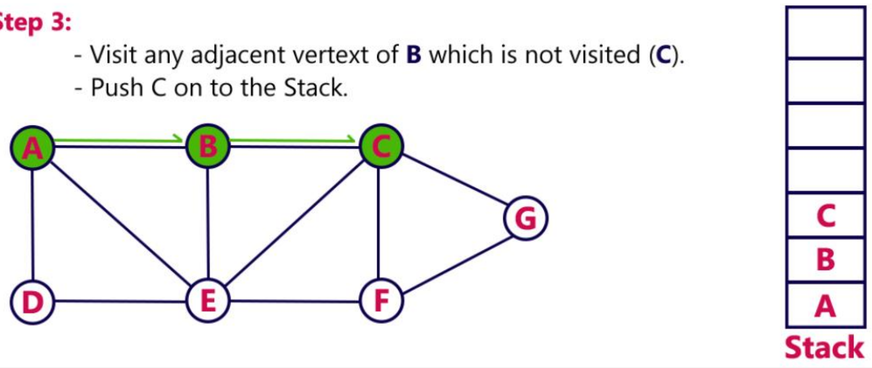
Step 2:

- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



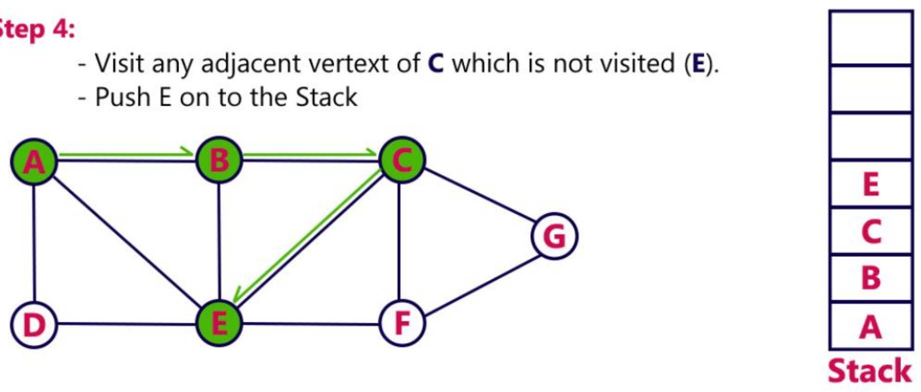
Step 3:

- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.



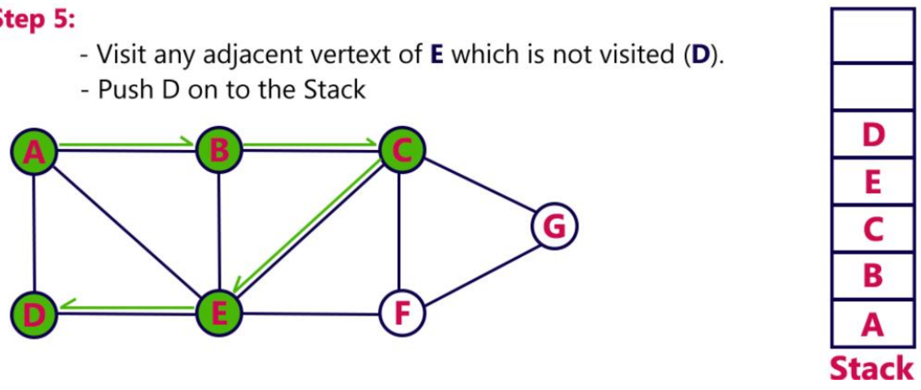
Step 4:

- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack



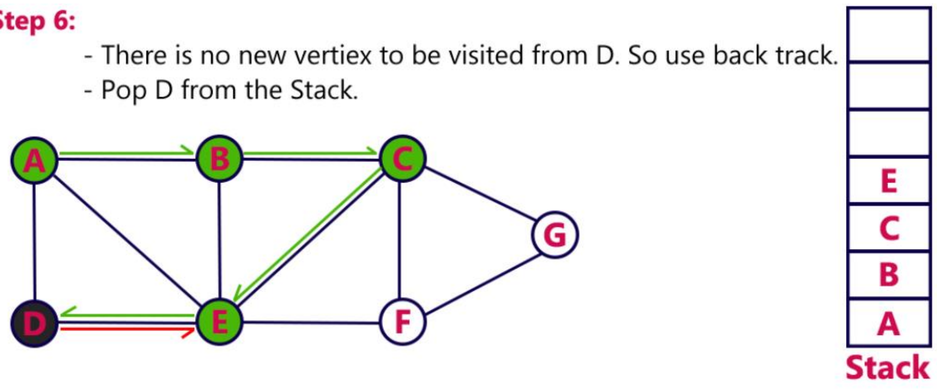
Step 5:

- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack



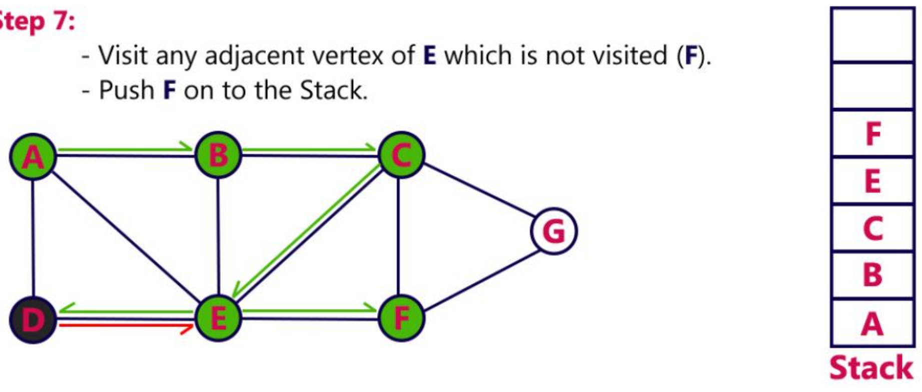
Step 6:

- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.



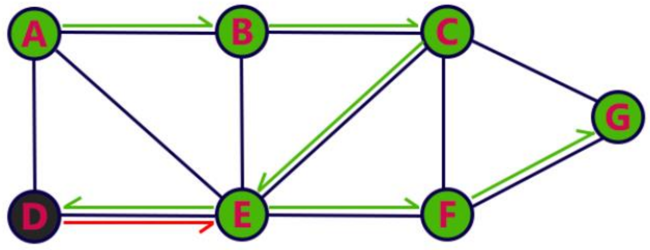
Step 7:

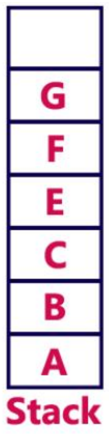
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.



Step 8:

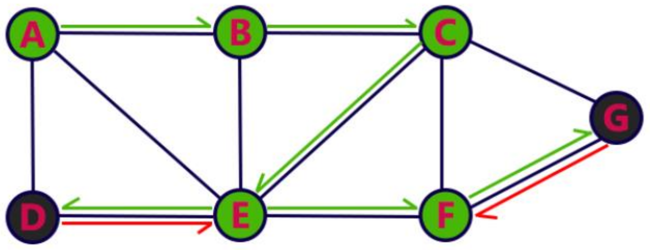
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

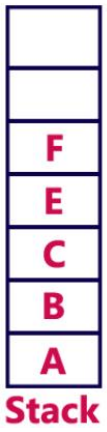




Step 9:

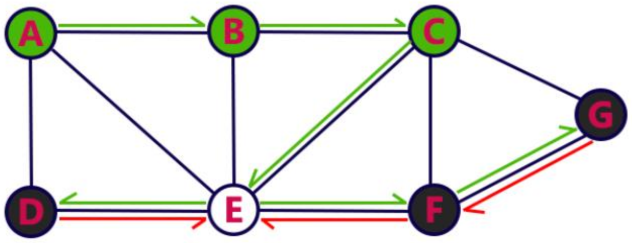
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

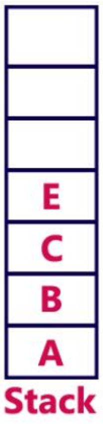




Step 10:

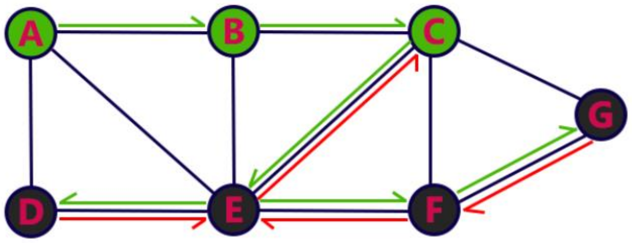
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

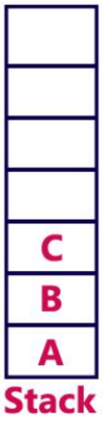




Step 11:

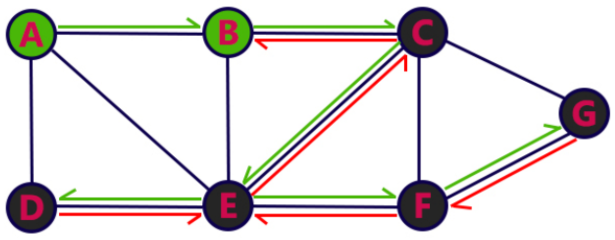
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

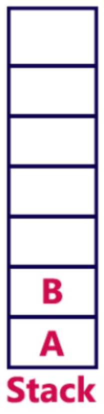




Step 12:

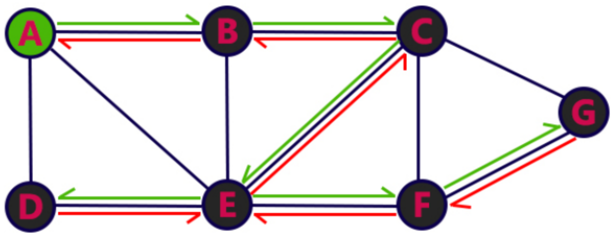
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

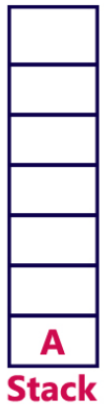




Step 13:

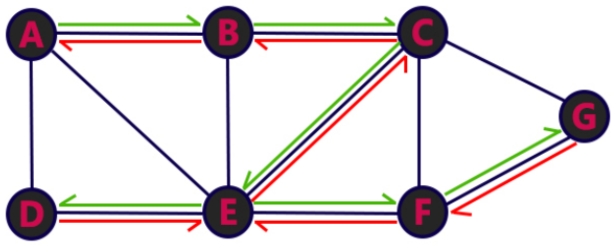
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

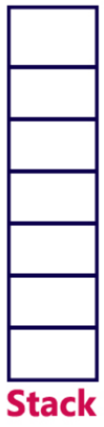




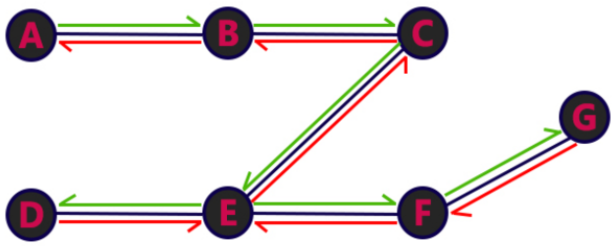
Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.

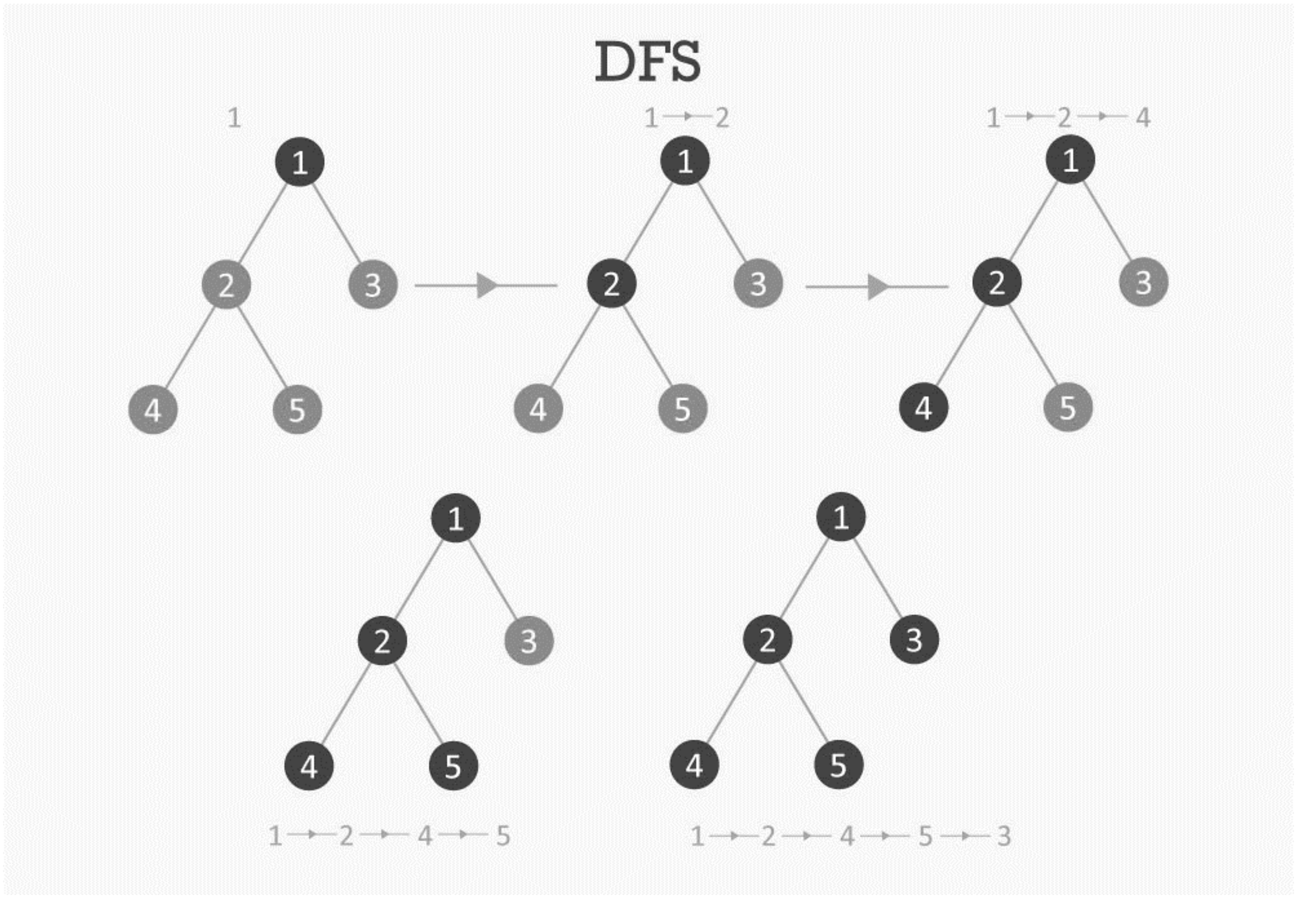




- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



Example 2:



Chapter 10 Graphs

BFS (Breadth First Search)

BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

We use the following steps to implement BFS traversal...

Step 1 - Define a Queue of size total number of vertices in the graph.

Step 2 - Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.

Step 3 - Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.

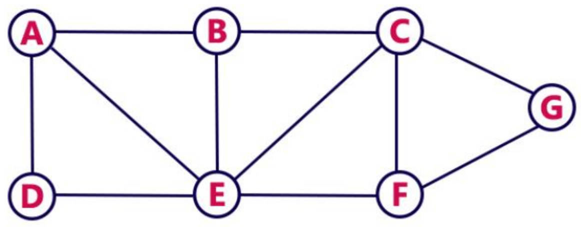
Step 4 - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

Step 5 - Repeat steps 3 and 4 until queue becomes empty.

Step 6 - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

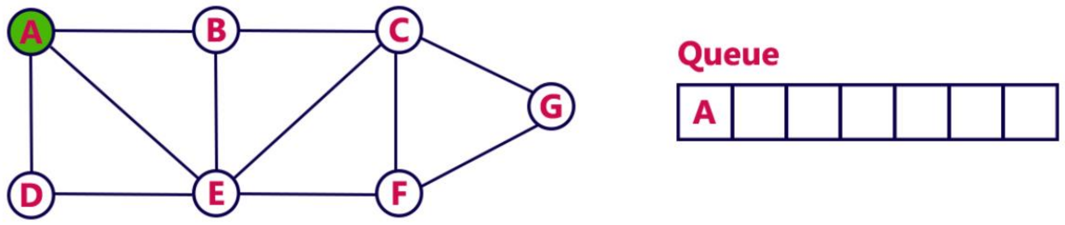
Example 1:

Consider the following example graph to perform BFS traversal



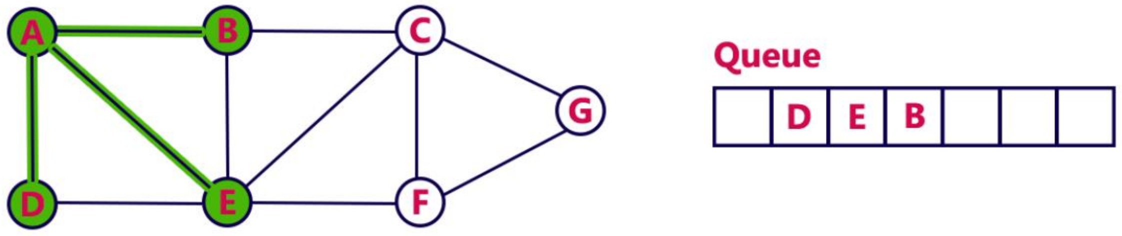
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



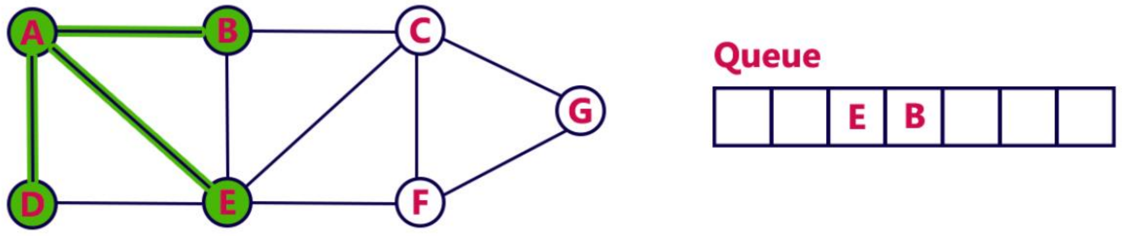
Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



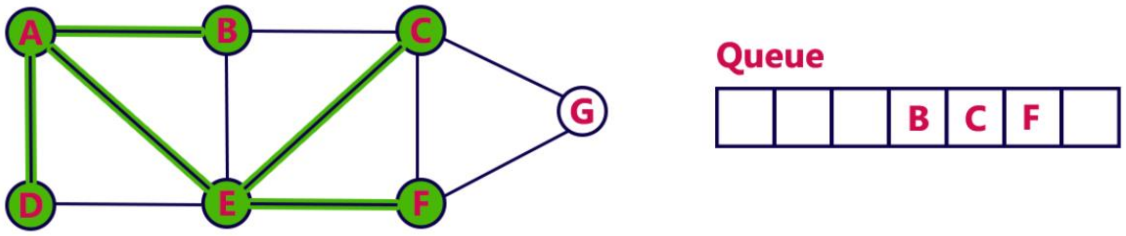
Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.



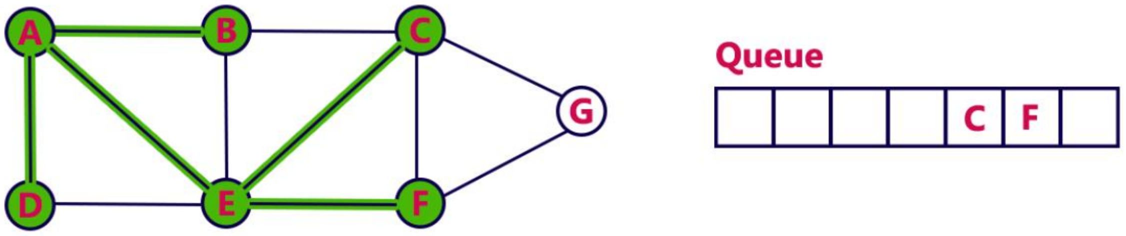
Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.



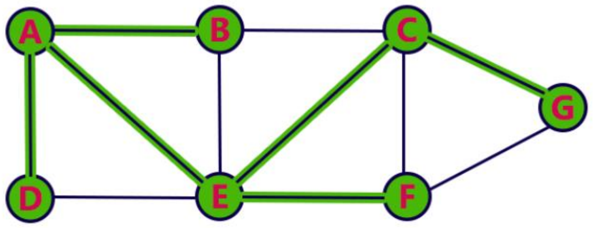
Step 5:

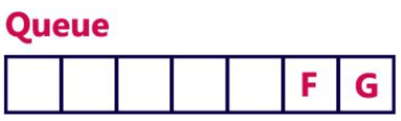
- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



Step 6:

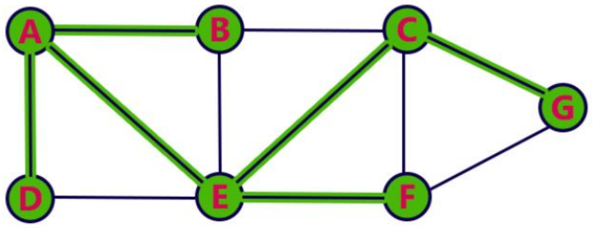
- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

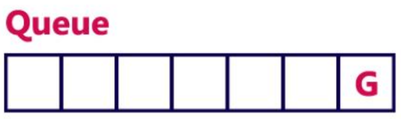




Step 7:

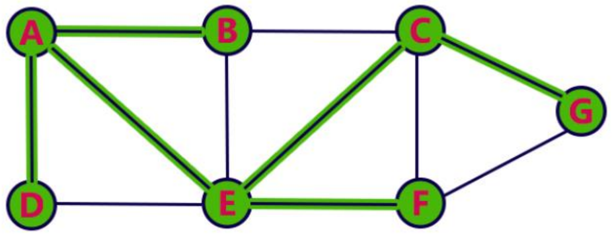
- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

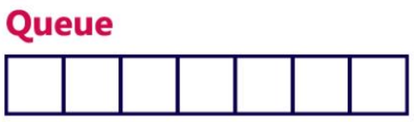




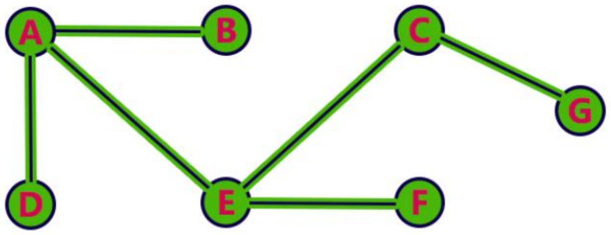
Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.





- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...

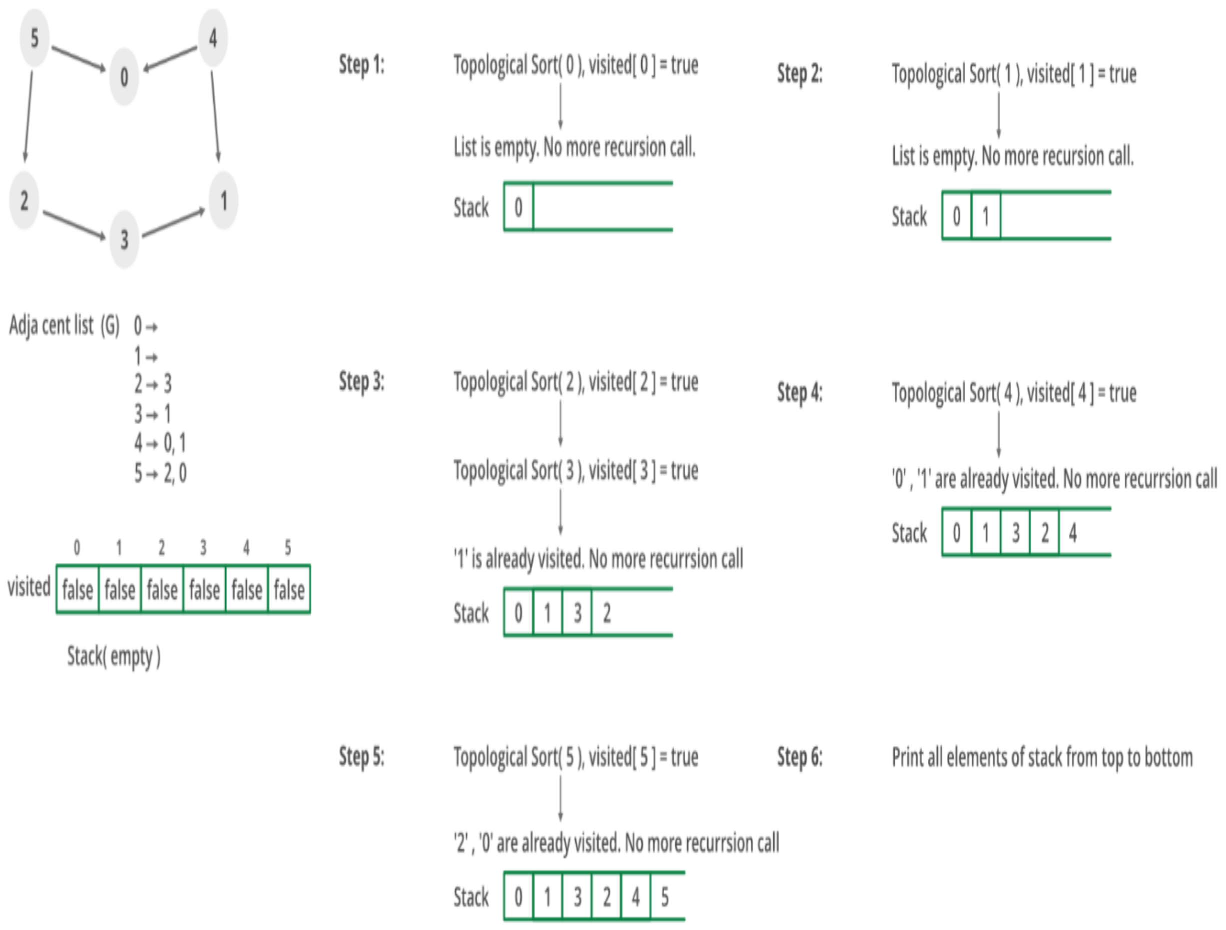


Topological Sort

- ***Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering***. Topological sorting for a graph is not possible if the graph is not a DAG.

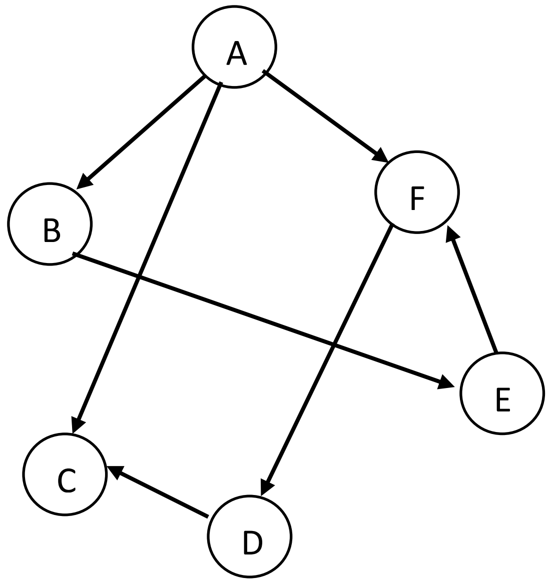
Topological Sort using Depth First Search (DFS)

- Use temporary stack to store the vertex.
- Maintain a visited [] to keep track of already visited vertices.
- In DFS we print the vertex and make recursive call to the adjacent vertices but here we will make the recursive call to the adjacent vertices and then push the vertex to stack.
- Observe closely the previous step, it will ensure that vertex will be pushed to stack only when all of its adjacent vertices (descendants) are pushed into stack.
- Finally print the stack.
- For disconnected graph, Iterate through all the vertices, during iteration, at a time consider each vertex as source (if not already visited).


5
4
0
2
1
3
Adja cent list (G) 0 →
1 →
2 → 3
3 → 1
4 → 0, 1
5 → 2, 0
0 1 2 3 4 5
visited false false false false false false
Stack(empty)
Step 1:
Topological Sort(0), visited[0] = true
List is empty. No more recursion call.
Stack 0
Step 2:
Topological Sort(1), visited[1] = true
List is empty. No more recursion call.
Stack 0 1
Step 3:
Topological Sort(2), visited[2] = true
Topological Sort(3), visited[3] = true
'1' is already visited. No more recurrsion call
Stack 0 1 3 2
Step 4:
Topological Sort(4), visited[4] = true
'0' , '1' are already visited. No more recurrsion call
Stack 0 1 3 2 4
Step 5:
Topological Sort(5), visited[5] = true
'2' , '0' are already visited. No more recurrsion call
Stack 0 1 3 2 4 5
Step 6:
Print all elements of stack from top to bottom

- For example, a topological sorting of the following graph is “5 4 2 3 1 0”.
- There can be more than one topological sorting for a graph.

- For example, another topological sorting of the following graph is “4 5 2 3 1 0”.

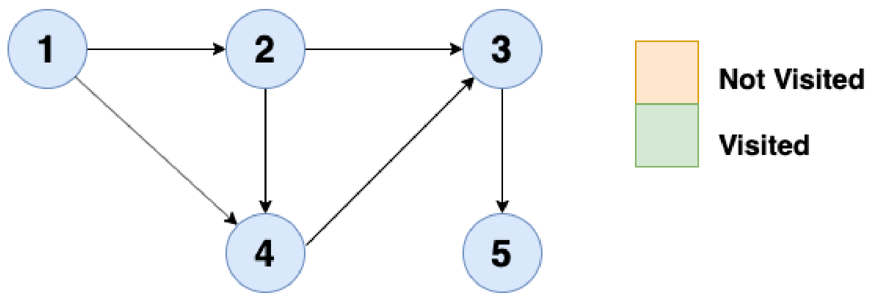


- For example, a topological sorting of the following graph is “A B E F D C ”.
- There can be more than one topological sorting for a graph.

Topological Sort using Breadth First Search (BFS)/ Kahn’s Algorithm

For that, we will maintain an array **T[ ],** which will store the ordering of the vertices in topological order. We will store the number of edges that are coming into a vertex in an array **in_degree[N],** where the ***i-th*** element will store the number of edges coming into the vertex ***i.*** We will also store whether a certain vertex has been visited or not in **visited[N].** We will follow the below steps:

- First, take out the vertex whose in_degree is 0. That means there is no edge that is coming into that vertex.
- We will append the vertices in the Queue and mark these vertices as visited.
- Now we will traverse through the queue and in each step we will dequeue () the front element in the Queue and push it into the **T.**
- Now, we will put out all the edges that are originated from the front vertex which means we will decrease the in_degree of the vertices which has an edge with the front vertex.
- Similarly, for those vertices whose in_degree is 0, we will push it in Queue and also mark that vertex as visited.



Step 1

Queue = [1]

in_degree[ ]	0	1	2	2	1
	1	2	3	4	5

T = [ ]

Step 2

Queue = [2]

in_degree[ ]	0	0	2	1	1
	1	2	3	4	5

T = [1]

Step 3

Queue = [4]

in_degree[ ]	0	0	1	0	1
	1	2	3	4	5

T = [1, 2]

Step 4

Queue = [3]

in_degree[ ]	0	0	0	0	1
	1	2	3	4	5

T = [1, 2, 4]

Step 5

Queue = [5]

in_degree[ ]	0	0	0	0	0
	1	2	3	4	5

T = [1, 2, 4, 3]

Step 6

Queue = [ ]

in_degree[ ]	0	0	0	0	0
	1	2	3	4	5

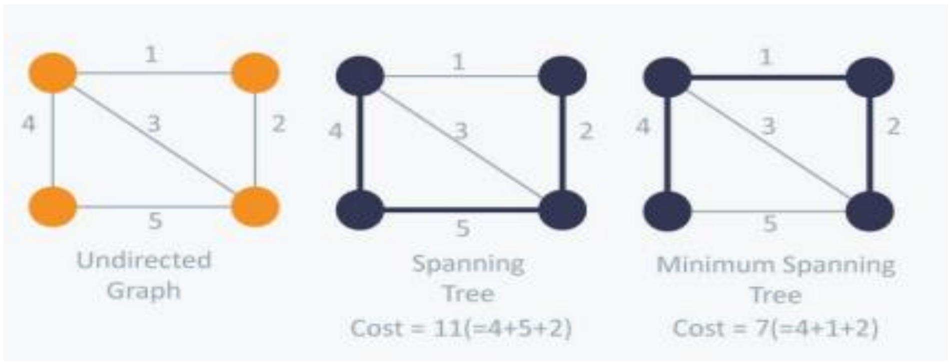
T = [1, 2, 4, 3, 5]

Chapter 10 Graphs

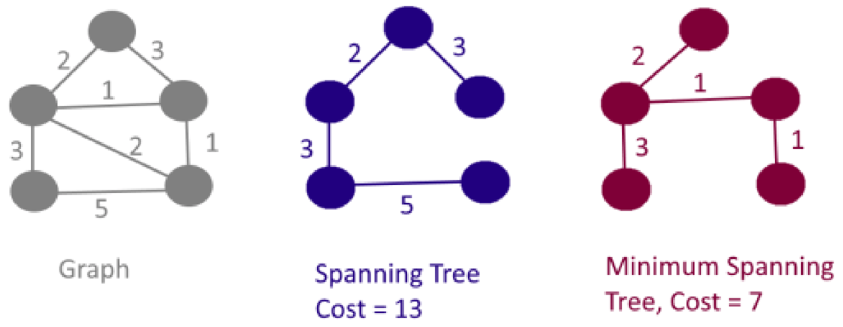
Minimum Spanning Tree:

- Given a ***connected and undirected graph***, a spanning tree of that graph is a subgraph that is a tree and ***connects all the vertices together and the graph doesn't have any nodes which loop back to itself.***
- A single graph can have many different spanning trees.
- A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree.
- The weight of a spanning tree is the sum of weights given to each edge of the spanning tree
- In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.
- The total number of spanning trees with n vertices that can be created from a complete graph is equal to $n^{(n-2)}$.
- A minimum spanning tree **has (V – 1) edges** where V is the number of vertices in the given graph.

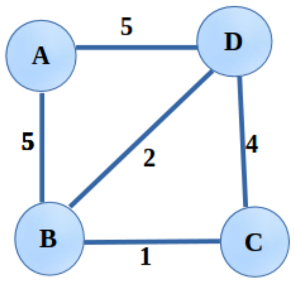
Example 1:

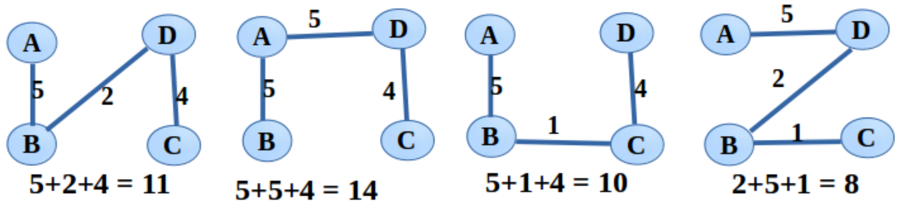


Example 2:



Example 3:





Kruskal's algorithm:

Below are the steps for finding MST using Kruskal's algorithm

Step 1: Remove all loops and parallel edges. In case of parallel edges, keep the one which has the least cost associated and remove all others.

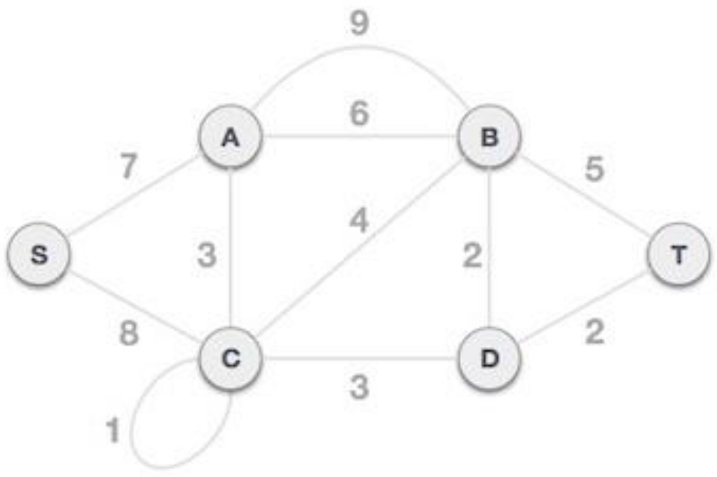
Step 2: Arrange all edges in their increasing order of weight

Step 3: Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.

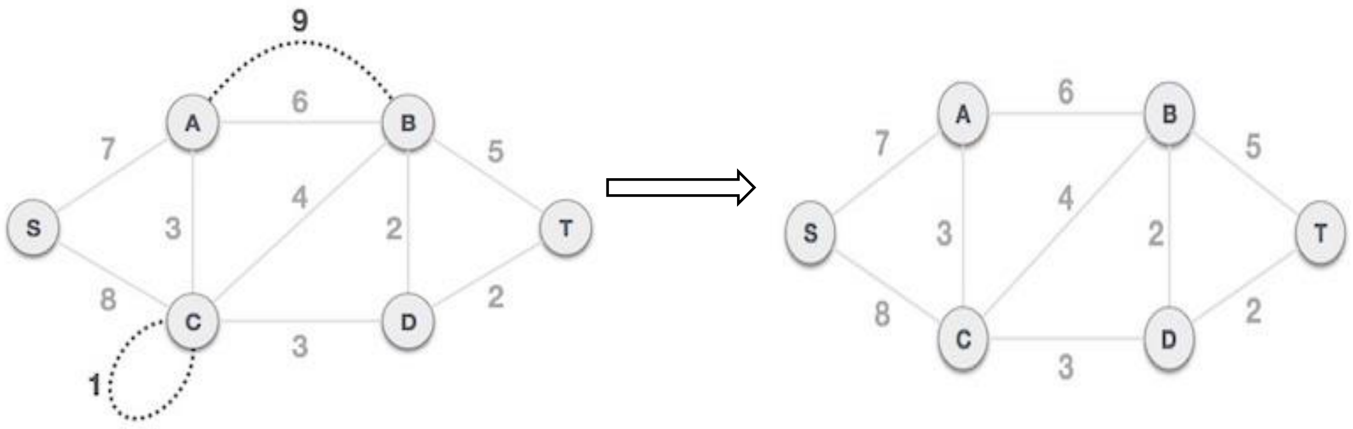
Step 4: Repeat step 3 until there are (V-1) edges in the spanning tree.

Chapter 10 Graphs

Example 1:



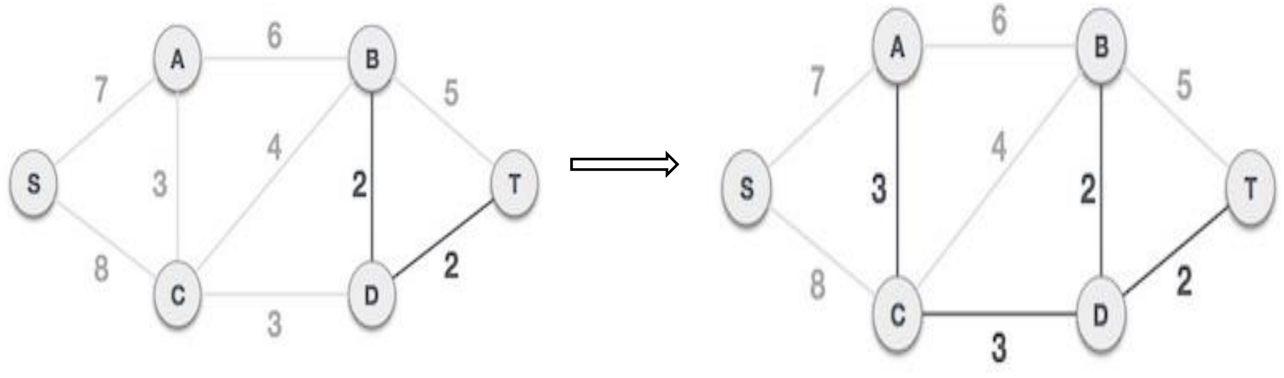
- Remove all loops and parallel edges. In case of parallel edges, keep the one which has the least cost associated and remove all others.



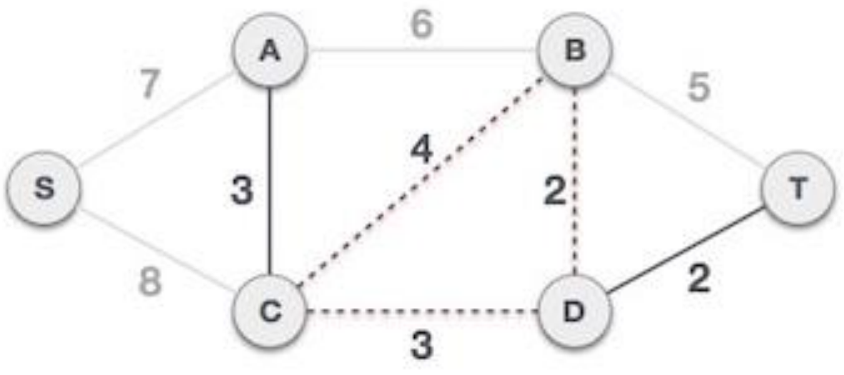
- Arrange all edges in their increasing order of weight. Create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

- Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.



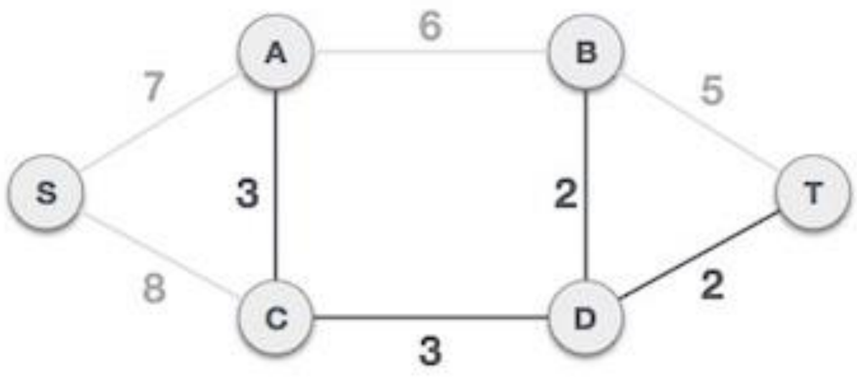
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



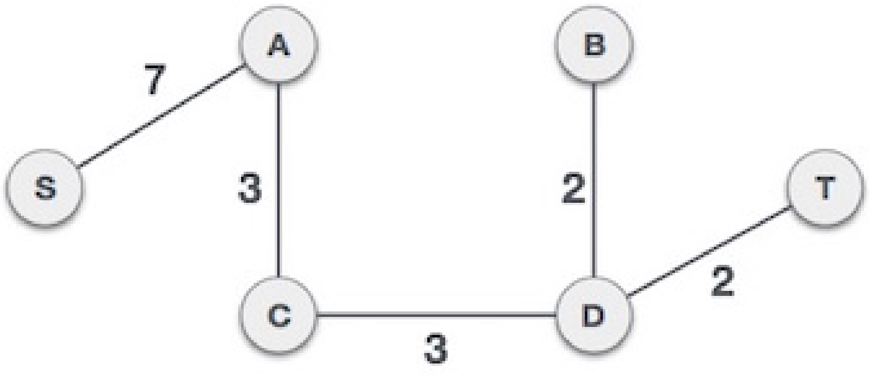
We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on

By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

Prim's Algorithm is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.



Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.



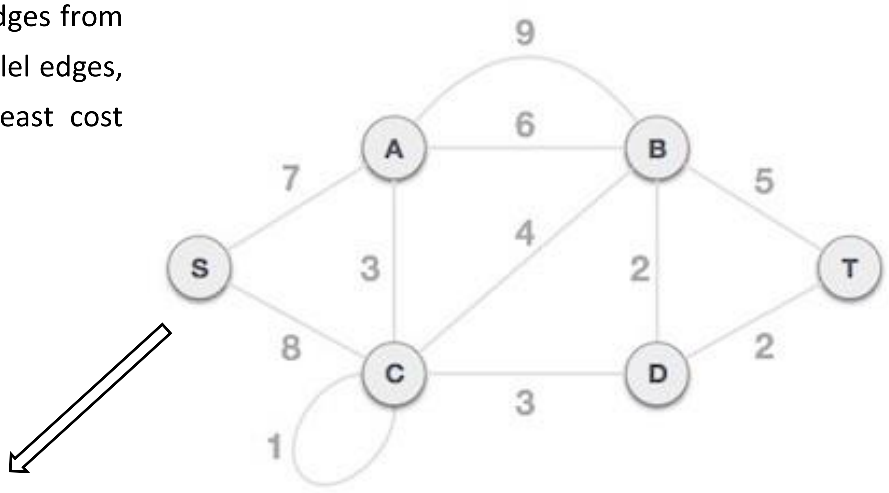
The steps to implement the prim's algorithm are given as follows -

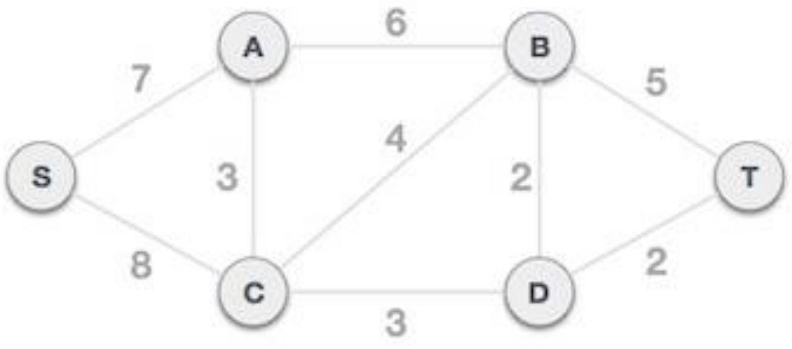
- First, remove all loops and parallel edges and we have to initialize an MST with the randomly chosen vertex.

- Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.
- Repeat step 2 until the minimum spanning tree is formed.

Example 1:

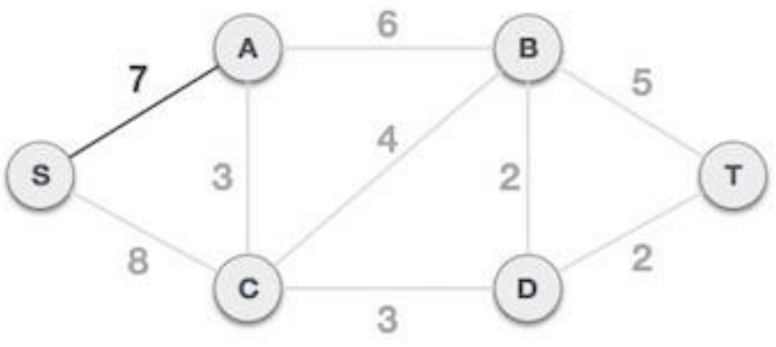
Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all other



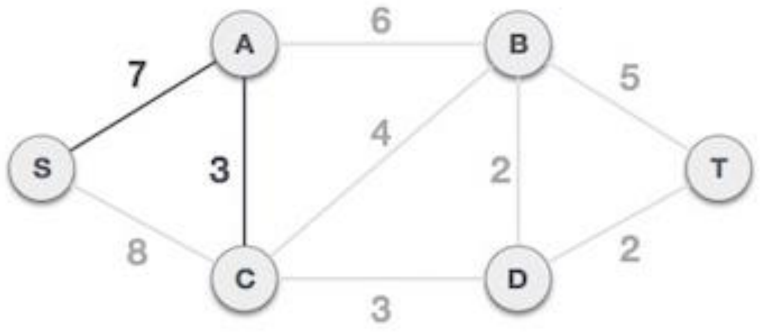


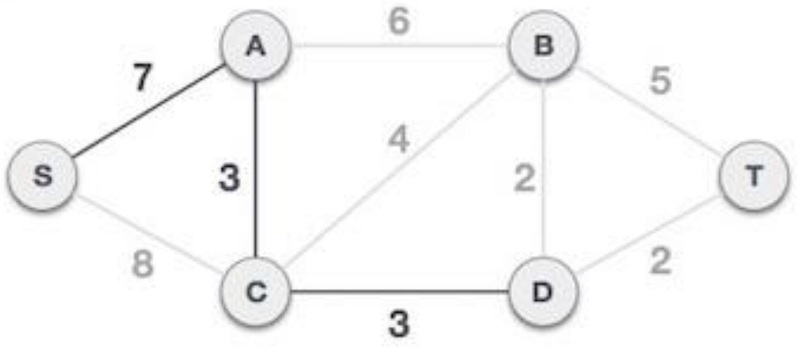
Choose any arbitrary node as root node

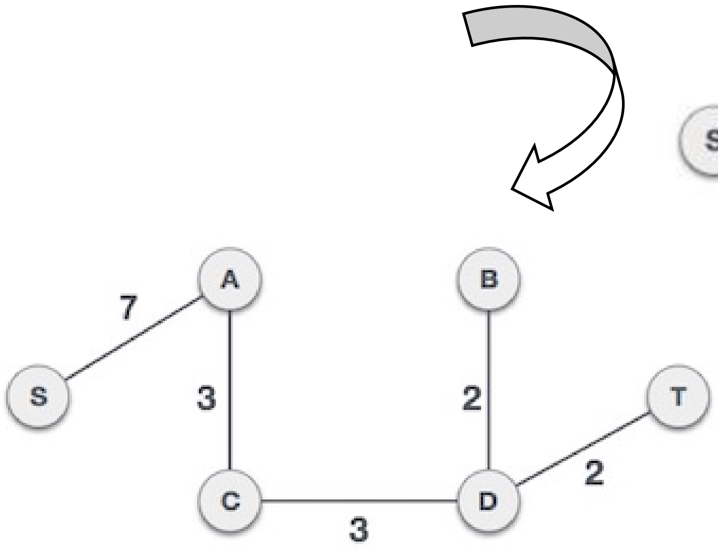
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any node can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.



Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree i.e., S-7-A-3-C.

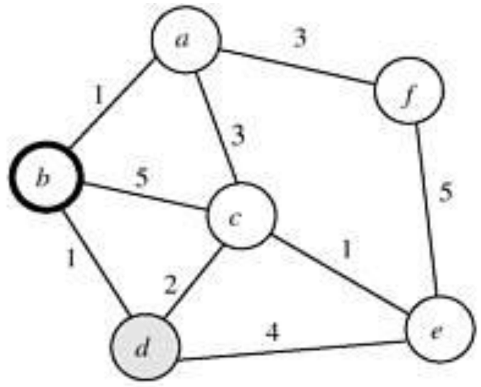






Example 2:

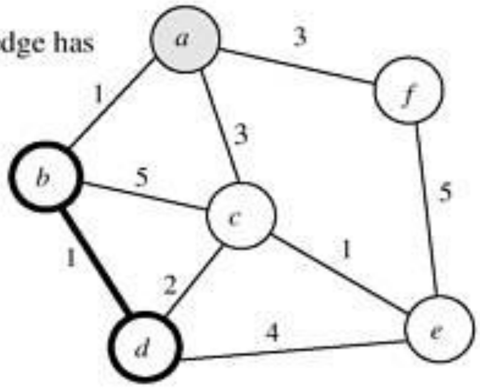
(a) Original graph



$$\begin{array}{c|cccccc} & a & b & c & d & e & f \\ \hline d[\,] & 1 & 0 & 5 & 1 & \infty & \infty \end{array}$$

$$\begin{array}{c} a \\ b \\ c \\ d \\ e \\ f \end{array}\begin{bmatrix} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 2 & \infty & \infty & \infty & 5 & 0 \end{bmatrix}$$

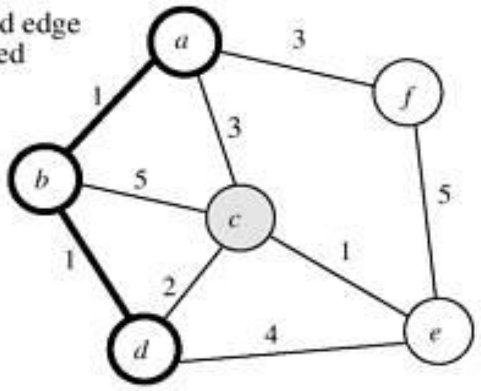
(b) After the first edge has been selected



$$\begin{array}{c|cccccc} & a & b & c & d & e & f \\ \hline d[\,] & 1 & 0 & 2 & 1 & 4 & \infty \end{array}$$

$$\begin{array}{c} a \\ b \\ c \\ d \\ e \\ f \end{array}\begin{bmatrix} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 2 & \infty & \infty & \infty & 5 & 0 \end{bmatrix}$$

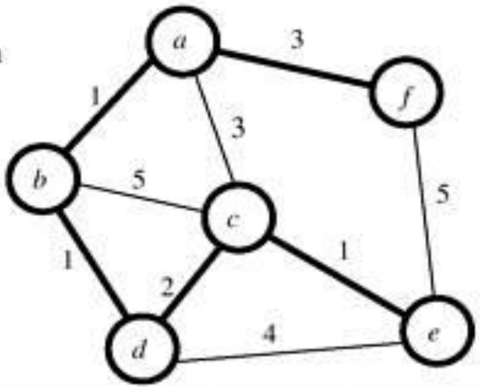
(c) After the second edge has been selected



$$\begin{array}{c|cccccc} & a & b & c & d & e & f \\ \hline d[\,] & 1 & 0 & 2 & 1 & 4 & 3 \end{array}$$

$$\begin{array}{c} a \\ b \\ c \\ d \\ e \\ f \end{array}\begin{bmatrix} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 2 & \infty & \infty & \infty & 5 & 0 \end{bmatrix}$$

(d) Final minimum spanning tree



$$\begin{array}{c|cccccc} & a & b & c & d & e & f \\ \hline d[\,] & 1 & 0 & 2 & 1 & 1 & 3 \end{array}$$

$$\begin{array}{c} a \\ b \\ c \\ d \\ e \\ f \end{array}\begin{bmatrix} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 2 & \infty & \infty & \infty & 5 & 0 \end{bmatrix}$$

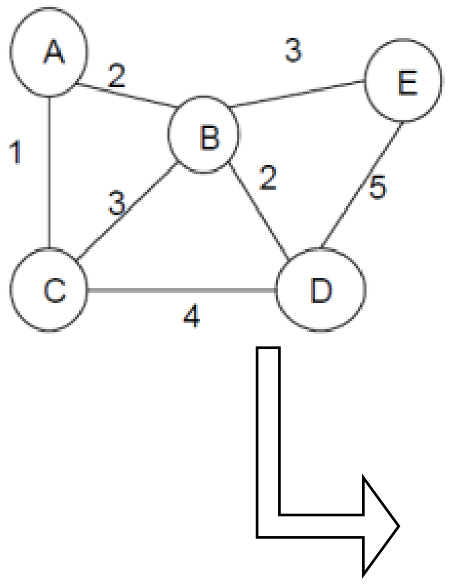
Chapter 10 Graphs

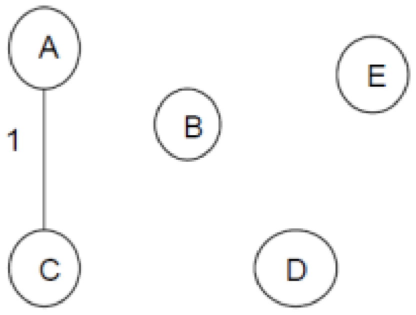
Round Robin Algorithm:

This method provides better performance when the number of edges is low. Initially each node is considered to be a partial tree. Each partial tree is maintained in a queue Q.

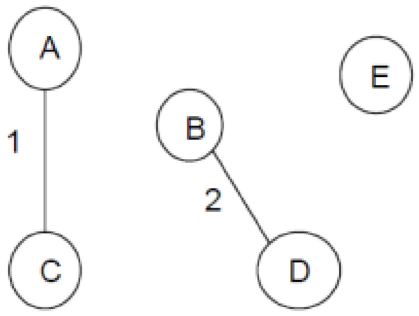
- Priority queue is associated with each partial tree, which contains all the arcs ordered by their weights.
- The algorithm proceeds by removing a partial tree, T1, from the front of Q, finding the minimum weight arc a in T1; deleting from Q, the tree T2, at the other end of arc a; combining T1 and T2 into a single new tree T3 and at the same time combining priority queues of T1 and T2 and adding T3 at the rear of priority queue.
- This continues until Q contains a single tree, the minimum spanning tree.

Q	Priority queue
{A}	1, 2
{B}	2, 2, 3, 3
{C}	1, 3, 4
{D}	2, 4, 5
{E}	3, 5

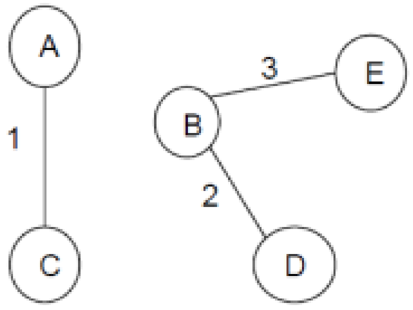




Q	Priority queue
{B}	2, 2, 3, 3
{D}	2, 4, 5
{E}	3, 5
{A, C}	2, 3, 4

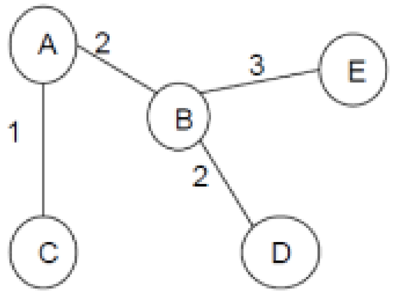


Q	Priority queue
{E}	3, 5
{A, C}	2, 3, 4
{B, D}	2, 3, 3, 4, 5



Q	Priority queue
{A, C}	2, 3, 4
{E, B, D}	2, 3, 4, 5, 5

Q	Priority queue
{A, C, E, B, D}	3, 3, 4, 4, 5, 5



Only 1 partial tree is left in the queue, which is the required minimum spanning tree.

Chapter 10 Graphs

Shortest path algorithm

Greedy Algorithm:

A greedy algorithm is a simple, intuitive algorithm that is used in optimization problems. ***The algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem.***

It follows local optimal choice of each stage with intend of finding global optimum.

Greedy algorithms are quite successful in some problems, such as Huffman encoding which is used to compress data, or Dijkstra's algorithm, which is used to find the shortest path through a graph.

Chapter 10 Graphs

Dijkstra's shortest path algorithm

Given a graph and a source vertex in the graph, find shortest paths from source to all vertices in the given graph.

Algorithm

1. Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

2. Assign a distance value to all vertices in the input graph. Initialize all distance values as

INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

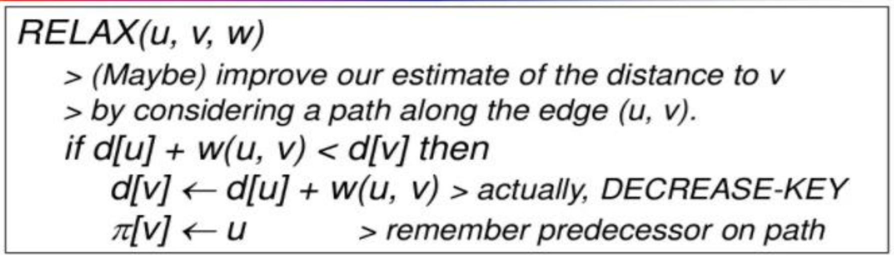
3. While sptSet doesn't include all vertices

   a) Pick a vertex u which is not there in sptSet and has minimum distance value.

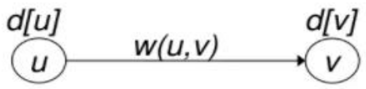
   b) Include u to sptSet.

   c) Update distance value of all adjacent vertices of u.

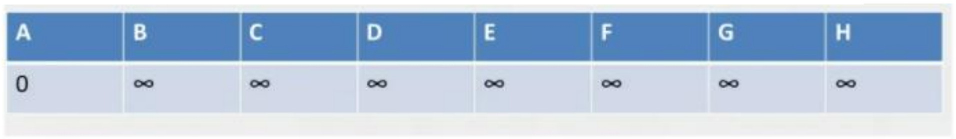
   To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.



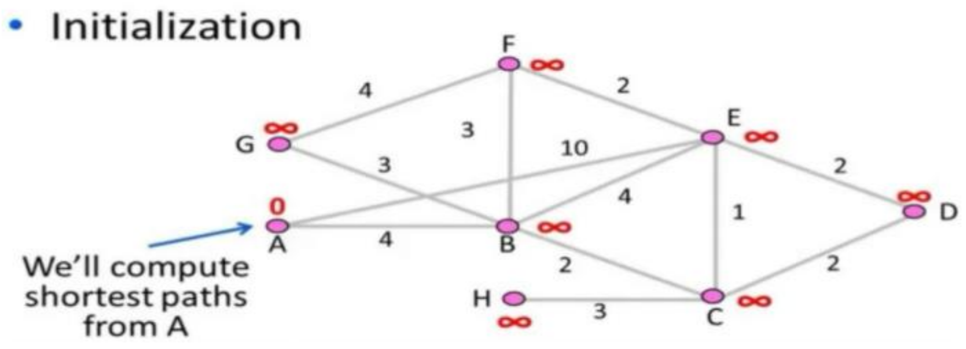
```
RELAX(u, v, w)
    > (Maybe) improve our estimate of the distance to v
    > by considering a path along the edge (u, v).
    if $d[u] + w(u, v) < d[v]$ then
        $d[v] \leftarrow d[u] + w(u, v)$   > actually, DECREASE-KEY
        $\pi[v] \leftarrow u$              > remember predecessor on path
```



Example:

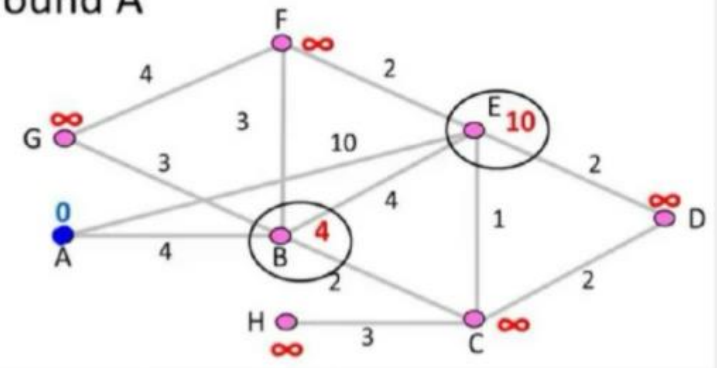


A	B	C	D	E	F	G	H
0	∞	∞	∞	∞	∞	∞	∞



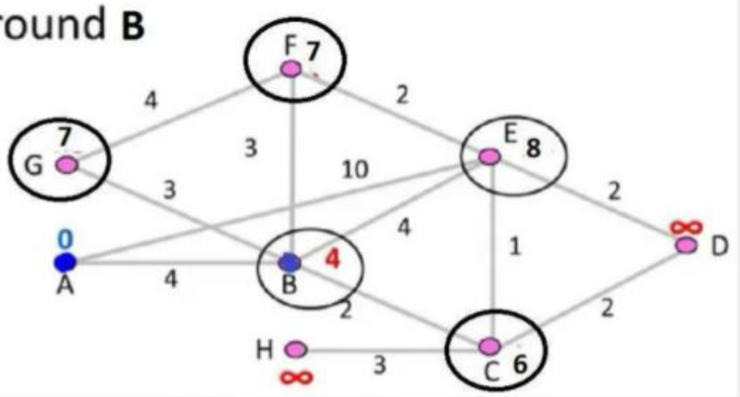
A	B	C	D	E	F	G	H
0	4	∞	∞	10	∞	∞	∞

- Relax around A



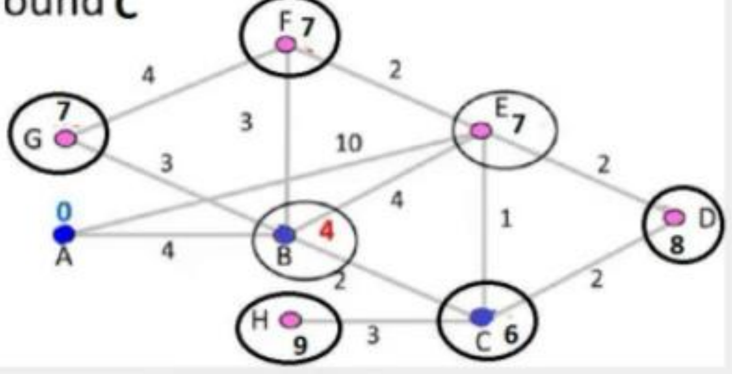
A	B	C	D	E	F	G	H
0	4	6	∞	8	7	7	∞

- Relax around **B**



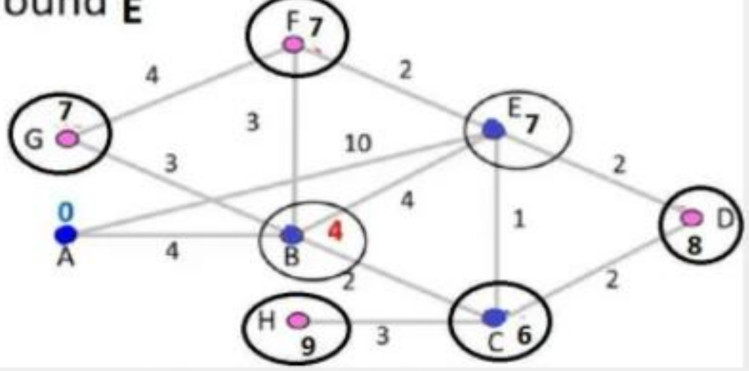
A	B	C	D	E	F	G	H
0	4	6	8	7	7	7	9

- Relax around **C**



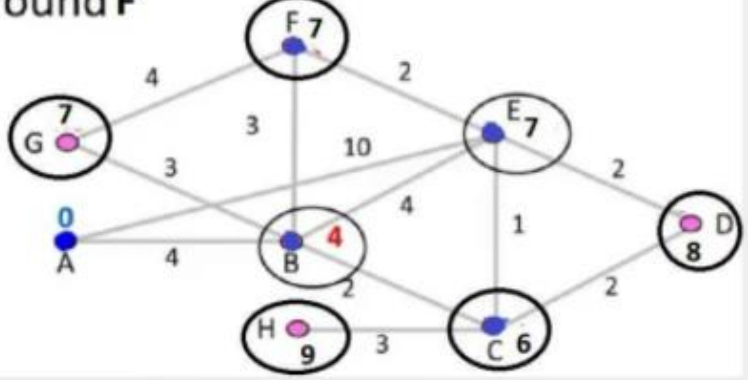
A	B	C	D	E	F	G	H
0	4	6	8	7	7	7	9

- Relax around **E**



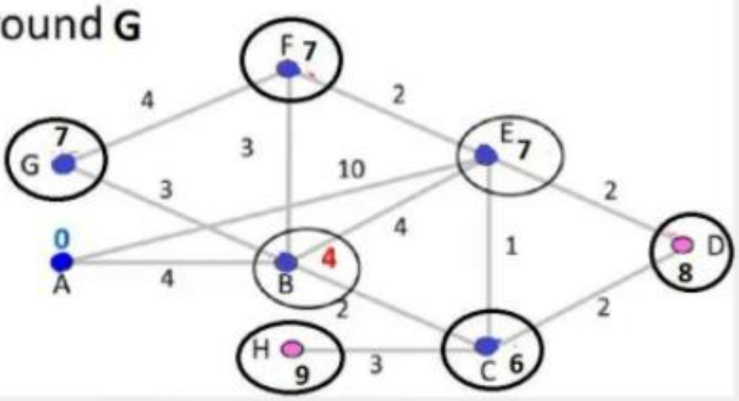
A	B	C	D	E	F	G	H
0	4	6	8	7	7	7	9

- Relax around **F**



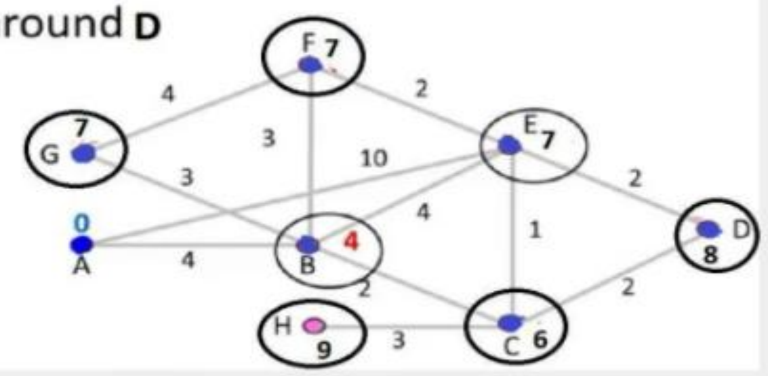
A	B	C	D	E	F	G	H
0	4	6	8	7	7	7	9

- Relax around **G**



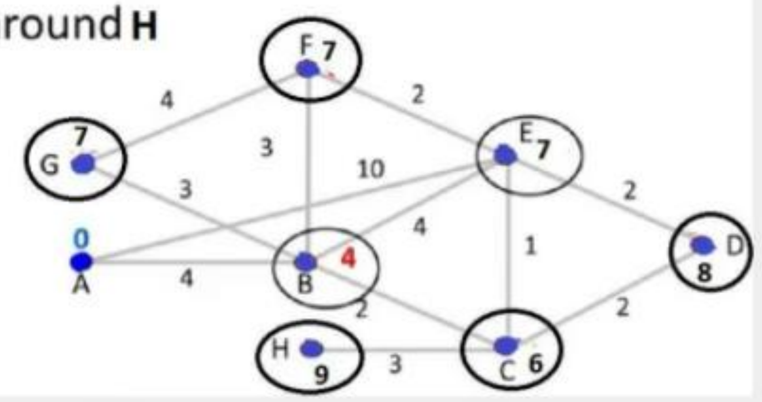
A	B	C	D	E	F	G	H
0	4	6	8	7	7	7	9

- Relax around **D**



A	B	C	D	E	F	G	H
0	4	6	8	7	7	7	9

- Relax around **H**



- So , the shortest path from source node A to all other nodes are –

  A – B = 4

  A – C = 6

  A – D = 8

  A – E = 7

  A – F = 7

  A – G = 7

  A – H = 9