CP-2 SEM IMPORTANT

UNIT-1

1. Naive and Euclid's algorithm of GCD.

Ans:

* Naive approach :-

Traverse all the numbers from min(A,B) to 1 and check whether the current number divides both A and B. If yes, it is the GCD of A and B.

Pseudo code :-

```
int GCD(int A, int B)
{
  int m = min(A,B), gcd;
  for(int i=m; i>0; --i)
     if(A%.i==0 && B%.i==0)
     {
        gcd = i;
        return gcd;
     }
}
```

Time complexity :- O(min(A,B)).

* Euclid's algorithm :-

The idea behind this algorithm is

GCD(A,B) = GCD(B, A%.B).

It will recurse until A%.B = 0.

Eg:- If a = 24 and b = 10, then find GCD

Step 1 :- GCD(24,10) = GCD(10, 24%.10) = GCD(10,4)

Step 2 :- GCD(10,4) = GCD(4, 10%.4) = GCD(4,2)

Step 3 :- GCD(4,2) = GCD(2, 4%.2) = GCD(2,0)

⇒ GCD(2,0) = 2 [∵ B = 0]

Time complexity :- O(log(max(A,B)))

2. Program for extended Euclid's algorithm.

Ans:

```
#include <iostream>
using namespace std;
int gcdExtended(int a, int b, int *x, int *y)
{
  if (a == 0)
  {
    *x = 0;
    *y = 1;
    return b;
  }
  int x1, y1;
  int gcd = gcdExtended(b%a, a, &x1, &y1);
  *x = y1 - (b/a) * x1;
  *y = x1;
  return gcd;
}
int main()
{
  int x, y;
  int a = 35, b = 15;
  cout<<"GCD is:"<<gcdExtended(a, b, &x, &y);
  return 0;
}
```

Output:

```
GCD is:5
```

3. Extended Euclid's algorithm with example program.

Ans:

Extended Euclidean algorithm :-

GCD(A,B) can always be represented in the form of an equation.

i.e., $Ax + By = GCD(A,B) \rightarrow (1)$

Where the coefficients x and y will be used to find the modular multiplicative inverse. The coefficients can be zero, positive or negative in value.

This algorithm is an extended form of Euclid's algorithm. which takes two inputs A and B and returns GCD (A,B) and coefficients of the above equation as output.

Eg:- If A = 30 and B = 20 then

$30 * (1) + 20 * (-1) = 10 = GCD(30,20)$

Key idea behind this algorithm :-

From Euclidean algorithm the GCD(A,B) can be represented as GCD(B, A%B)

$\Rightarrow B \cdot x_1 + (A \% B) \cdot y_1 = GCD(A,B) \rightarrow (2)$

Note :- $A \% B = A - B * \lfloor A/B \rfloor$ where $\lfloor \rfloor$ means floor value

substitute this in eqn (2)

$\Rightarrow B \cdot x_1 + (A - \lfloor A/B \rfloor \cdot B) \cdot y_1 = GCD(A,B)$

$\Rightarrow B(x_1 - \lfloor A/B \rfloor \cdot y_1) + A \cdot y_1 = GCD(A,B) \rightarrow (3)$

(5)

By comparing eqn (1) & (3), we get

$x = y_1$

$y = x_1 - \lfloor A/B \rfloor \cdot y_1$

These equations are key in understanding the extended Euclidean algorithm.

Program:

```
#include <iostream>
using namespace std;
int gcdExtended(int a, int b, int *x, int *y)
{
  if (a == 0)
  {
    *x = 0;
    *y = 1;
    return b;
  }
  int x1, y1;
  int gcd = gcdExtended(b%a, a, &x1, &y1);
  *x = y1 - (b/a) * x1;
  *y = x1;
  return gcd;
}
int main()
{
  int x, y;
  int a = 35, b = 15;
  cout<<"GCD is:"<<gcdExtended(a, b, &x, &y);
  return 0;
}
```

Output:

GCD is:5

4. Optimized method of Binary Exponential.

Ans:

Exponentiation means raising a number to a power. If we raise 'a' to the power 'b', it means we multiply a with itself b times.

a^b = a*a*a*a.....(*b times*)

If we look carefully at the definition, it means that to find a raised to the power of b or a^b, we would need to perform the multiplication operation b times.

Program:

```
#include <iostream>
using namespace std;
int main()
{
    int a = 4, b = 8;
    int answer = 1;
    for(int i=1; i<=b; i++)
    {
       answer = answer*a;
    }
    cout<<"4 raised to the power 8 is : "<<answer<<endl;
    return 0;
}
```

Output:

4 raised to the power 8 is : 65536

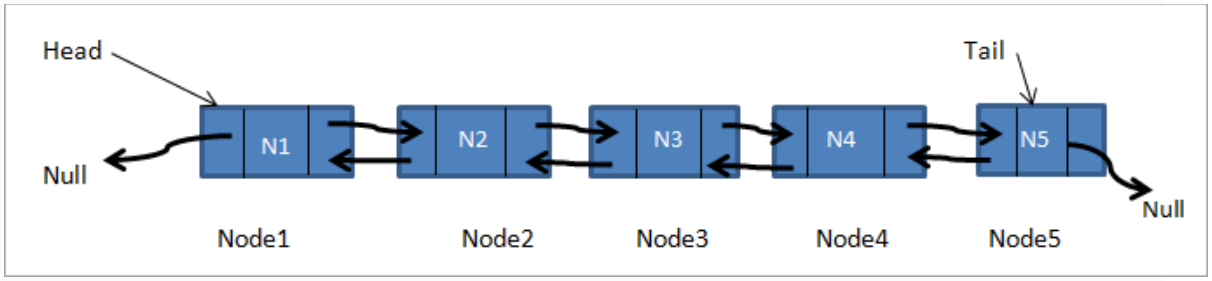
UNIT-2

1. Double Linked List and Circular Linked List

Ans:

Double Linked List:

In double linked list the previous pointer of the head is set to NULL as this is the first node. The next pointer of the tail node is set to NULL as this is the last node.



In the above figure, we see that each node has two pointers, one pointing to the previous node and the other pointing to the next node. Only the first node (head)

has its previous node set to null and the last node (tail) has its next pointer set to null.

As the doubly linked list contains two pointers i.e., previous and next, we can traverse it into the directions forward and backward. This is the main advantage of doubly linked list over the singly linked list.

Basic Operations:

Insertion

Insertion operation of the doubly linked list inserts a new node in the linked list. Depending on the position where the new node is to be inserted, we can have the following insert operations.

- **Insertion at front –** Inserts a new node as the first node.
- **Insertion at the end –** Inserts a new node at the end as the last node.
- **Insertion before a node –** Given a node, inserts a new node before this node.
- **Insertion after a node –** Given a node, inserts a new node after this node.

Deletion

Deletion operation deletes a node from a given position in the doubly linked list.

- **Deletion of the first node –** Deletes the first node in the list
- **Deletion of the last node –** Deletes the last node in the list.
- **Deletion of a node given the data –** Given the data, the operation matches the data with the node data in the linked list and deletes that node.

Traversal

Traversal is a technique of visiting each node in the linked list. In a doubly linked list, we have two types of traversals as we have two pointers with different directions in the doubly linked list.

- **Forward traversal –** Traversal is done using the next pointer which is in the forward direction.
- **Backward traversal –** Traversal is done using the previous pointer which is the backward direction.

Reverse

This operation reverses the nodes in the doubly linked list so that the first node becomes the last node while the last node becomes the first node.

Search

Search operation in the doubly linked list is used to search for a particular node in the linked list. For this purpose, we need to traverse the list until a matching data is found.

Program:

```
#include <iostream>
using namespace std;
struct Node
{
  int data;
  struct Node *prev;
  struct Node *next;
};
struct Node* head = NULL;
void insert(int newdata)
{
  struct Node* newnode = (struct Node*) malloc(sizeof(struct Node));
  newnode->data = newdata;
```

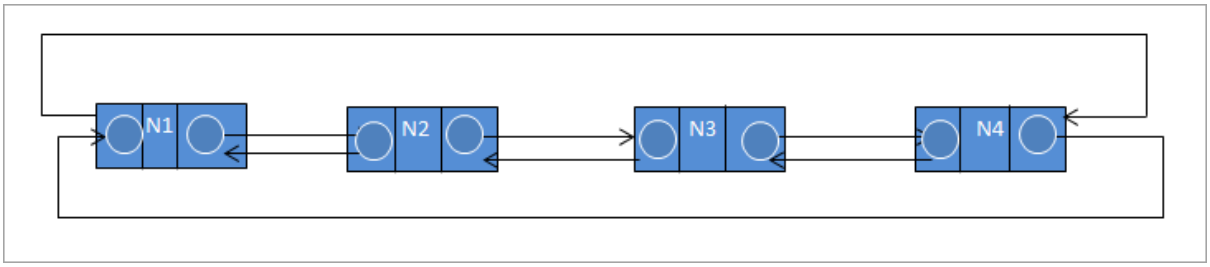
```
  newnode->prev = NULL;
  newnode->next = head;
  if(head != NULL)
  head->prev = newnode ;
  head = newnode;
}
void display()
{
  struct Node* ptr;
  ptr = head;
  while(ptr != NULL)
  {
    cout<< ptr->data <<" ";
    ptr = ptr->next;
  }
}
int main()
{
  insert(3);
  insert(1);
  insert(7);
  insert(2);
  insert(9);
  cout<<"The double linked list is: ";
  display();
  return 0;
}
```

Output:

The double linked list is: 9 2 7 1 3

Circular Linked List:

A circular linked list can be a singly linked list or a doubly linked list. In a doubly circular linked list, the previous pointer of the first node is connected to the last node while the next pointer of the last node is connected to the first node.



Basic Operations:

Insertion
We can insert a node in a circular linked list either as a first node (empty list), in the beginning, in the end, or in between the other nodes. Let us see each of these insertion operations using a pictorial representation below.

- **Insert in an empty list**
- **Insert at the beginning of the list**
- **Insert at the end of the list**
- **Insert in between the list**

Deletion
The deletion operation of the circular linked list involves locating the node that is to be deleted and then freeing its memory.

Traversal
Traversal is a technique of visiting each and every node. In linear linked lists like singly linked list and doubly linked lists, traversal is easy as we visit each node and stop when NULL is encountered.

Program:

```
#include <iostream>
using namespace std;
struct Node
{
  int data;
  struct Node *next;
};
struct Node* head = NULL;
void insert(int newdata) {
  struct Node *newnode = (struct Node *)malloc(sizeof(struct Node));
  struct Node *ptr = head;
  newnode->data = newdata;
  newnode->next = head;
  if (head!= NULL)
  {
    while (ptr->next != head)
    ptr = ptr->next;
    ptr->next = newnode;
  }
```

```
else
  newnode->next = newnode;
  head = newnode;
}
void display()
{
  struct Node* ptr;
  ptr = head;
  do
  {
    cout<<ptr->data <<" ";
    ptr = ptr->next;
  } while(ptr != head);
}
int main()
{
  insert(3);
  insert(1);
  insert(7);
  insert(2);
  insert(9);
  cout<<"The circular linked list is: ";
  display();
  return 0;
}
```

Output:

The circular linked list is: 9 2 7 1 3

2. **Code for deleting the middle element from the singly linked list.**

Ans:

```
#include <iostream>
using namespace std;
struct Node
{
    int data;
    struct Node* next;
};
```

```
int countOfNodes(struct Node* head)
{
    int count = 0;
    while (head != NULL)
    {
            head = head->next;
            count++;
    }
    return count;
}
struct Node* deleteMid(struct Node* head)
{
    if (head == NULL)
            return NULL;
    if (head->next == NULL)
    {
            delete head;
            return NULL;
    }
    struct Node* copyHead = head;
    int count = countOfNodes(head);
    int mid = count / 2;
    while (mid-- > 1)
            head = head->next;
    head->next = head->next->next;
    return copyHead;
}
void printList(struct Node* ptr)
{
    while (ptr != NULL)
    {
            cout << ptr->data << "->";
            ptr = ptr->next;
    }
    cout << "NULL\n";
}
Node* newNode(int data)
{
    struct Node* temp = new Node;
    temp->data = data;
    temp->next = NULL;
    return temp;
}
```

```
int main()
{
    struct Node* head = newNode(1);
    head->next = newNode(2);
    head->next->next = newNode(3);
    head->next->next->next = newNode(4);
    cout << "Given Linked List\n";
    printList(head);
    head = deleteMid(head);
    cout << "Linked List after deletion of middle\n";
    printList(head);
    return 0;
}
```

Output:

Given Linked List

1->2->3->4->NULL

Linked List after deletion of middle

1->2->4->NULL

3. Code for inserting the middle element in the circular linked list

Ans:

```
#include<iostream>
using namespace std;
class Node
{
   public:
   int data;
   Node *next;
};
int size = 0;
void insert(Node** head, int data)
{
   Node* newNode = new Node();
   newNode->data = data;
   if(*head == NULL)
   {
      *head = newNode;
```

```
        (*head)->next = *head;
        size++;
        return;
    }
    Node* curr = *head;
    while(curr->next != *head){
        curr = curr->next;
    }
    curr->next = newNode;
    newNode->next = *head;
    *head = newNode;
    size++;
}
void insertMiddle(Node** head, int data)
{
    Node* newNode = new Node();
    newNode->data = data;
    if(*head == NULL)
    {
        insert(head, data);
        return;
    }
    Node* temp = *head;
    int mid = (size % 2 == 0) ? (size/2) : (size+1)/2;
    if(mid == size)
    {
        newNode->next = *head;
        (*head)->next = newNode;
        size++;
        return;
    }
    while(--mid)
    {
        temp = temp->next;
    }
    newNode->next = temp->next;
    temp->next = newNode;
    size++;
}
void display(Node* head)
{
    if(head == NULL)
        return;
```

```
    cout << "Linked List: ";
    Node* temp = head;
    do
    {
       cout <<temp->data << " ";
       temp = temp->next;
    }
    while(temp!=head);
    cout << "\n" << endl;
}
int main()
{
    Node* head = NULL;
    insert(&head,20);
    insert(&head,4);
    display(head);
    insertMiddle(&head, 8);
    display(head);
    return 0;
}
```

Output:

Linked List: 4 20

Linked List: 4 8 20

UNIT-3

1. Linear Search

Ans:

```
#include<iostream>
using namespace std;
int linearSearch(int a[],int size,int searchValue)
{
    for(int i=0;i<size;i++)
    {
            if(searchValue==a[i])
            {
                    return i;
            }
    }
```

```
    return -1;
}
int main()
{
    int a[]={45,78,50,80,75,98,83};
    int userValue;
    cout<<"Enter an integer value:"<<endl;
    cin>>userValue;
    int result=linearSearch(a,7,userValue);
    if(result>=0)
    {
        cout<<"The number "<<a[result]<<" was found at the element with
        index: "<<result<<endl;
    }
    else
    {
        cout<<"The number "<<userValue<<" was not found"<<endl;
    }
    return 0;
}
```

Output:

Enter an integer value:

78

The number 78 was found at the element with index: 1

2. Quick Sort

Ans:

```
#include<iostream>
using namespace std;
int Partition(int a[],int start, int end)
{
    int pivot=a[end];
    int pIndex=start;
    for(int i=start; i < end; i++)
    {
        if(a[i]<pivot)
        {
            int temp=a[i];
            a[i]=a[pIndex];
            a[pIndex]=temp;
```

```
                pIndex++;
            }
    }
    int temp=a[end];
    a[end]=a[pIndex];
    a[pIndex]=temp;
}
void QuickSort(int a[],int start, int end)
{
    if(start<end)
    {
        int p=Partition(a,start,end);
        QuickSort(a,start,(p-1));
        QuickSort(a,(p+1),end);
    }
}
int main()
{
    int size=0;
    cout<<"Enter size of array:"<<endl;
    cin>>size;
    int myarray[size];
    cout<<"Enter "<<size<<" integers in any order:"<<endl;
    for(int i=0;i<size;i++)
    {
        cin>>myarray[i];
    }
    cout<<"Before Sorting:"<<endl;
    for(int i=0;i<size;i++)
    {
        cout<<myarray[i]<<" ";
    }
    cout<<endl;
    QuickSort(myarray,0,(size-1));
    cout<<"After Sorting:"<<endl;
    for(int i=0;i<size;i++)
    {
        cout<<myarray[i]<<" ";
    }
    return 0;
}
```

Output:

Enter size of array:

6

Enter 6 integers in any order:

45 78 10 80 102 83

Before Sorting:

45 78 10 80 102 83

After Sorting:

10 45 78 80 83 102

UNIT-4

1. Stack using array and linked list.

Ans:

Stack Using Array:

```
#include <iostream>
using namespace std;
int stack[100], n=100, top=-1;
void push(int val)
{
  if(top>=n-1)
  cout<<"Stack Overflow"<<endl;
  else
  {
    top++;
    stack[top]=val;
  }
}
void pop()
{
  if(top<=-1)
  cout<<"Stack Underflow"<<endl;
  else
  {
    cout<<"The popped element is "<< stack[top] <<endl;
    top--;
  }
```

```
}
void display()
{
  if(top>=0)
  {
    cout<<"Stack elements are:";
    for(int i=top; i>=0; i--)
    cout<<stack[i]<<" ";
    cout<<endl;
  }
  else
  cout<<"Stack is empty"<<endl;
}
int main()
{
  int ch, val;
  cout<<"1) Push in stack"<<endl;
  cout<<"2) Pop from stack"<<endl;
  cout<<"3) Display stack"<<endl;
  cout<<"4) Exit"<<endl;
  do
  {
    cout<<"Enter choice: "<<endl;
    cin>>ch;
    switch(ch)
    {
      case 1:
      {
        cout<<"Enter value to be pushed:"<<endl;
        cin>>val;
        push(val);
        break;
      }
      case 2:
      {
        pop();
        break;
      }
      case 3:
      {
        display();
        break;
      }
```

```
        case 4:
        {
          cout<<"Exit"<<endl;
          break;
        }
        default:
        {
          cout<<"Invalid Choice"<<endl;
        }
      }
   }
   while(ch!=4);
   return 0;
}
```

Stack Using Linked List:

```
#include <iostream>
using namespace std;
struct Node
{
int data;
Node *link;
};
Node *top = NULL;
bool isempty()
{
 if(top == NULL)
 return true; else
 return false;
}
void push (int value)
{
  Node *ptr = new Node();
  ptr->data = value;
  ptr->link = top;
  top = ptr;
}
void pop ( )
{
 if ( isempty() )
  cout<<"Stack is Empty";
 else
```

```
 {
  Node *ptr = top;
  top = top -> link;
  delete(ptr);
 }
}
void showTop()
{
 if ( isempty() )
  cout<<"Stack is Empty";
 else
  cout<<"Element at top is : "<< top->data;
}
void displayStack()
{
 if ( isempty() )
  cout<<"Stack is Empty";
 else
 {
  Node *temp=top;
  while(temp!=NULL)
  {   cout<<temp->data<<" ";
   temp=temp->link;
  }
  cout<<"\n";
 }
 }
int main()
{
 int choice, flag=1, value;
 while( flag == 1)
 {
 cout<<"\n1.Push 2.Pop 3.showTop 4.displayStack 5.exit\n";
 cin>>choice;
 switch (choice)
    {
        case 1: cout<<"Enter Value:\n";
                cin>>value;
                push(value);
                break;
        case 2: pop();
                break;
        case 3: showTop();
```

```
            break;
        case 4: displayStack();
            break;
        case 5: flag = 0;
            break;
    }
 }
return 0;
}
```

2. Queue using array and linked list.

Ans:

Queue Using Array:

```
#include <iostream>
using namespace std;
int queue[100], n = 100, front = - 1, rear = - 1;
void Insert()
{
  int val;
  if (rear == n - 1)
  cout<<"Queue Overflow"<<endl;
  else
  {
    if (front == - 1)
    front = 0;
    cout<<"Insert the element in queue : "<<endl;
    cin>>val;
    rear++;
    queue[rear] = val;
  }
}
void Delete()
{
  if (front == - 1 || front > rear)
  {
    cout<<"Queue Underflow ";
    return ;
  }
  else
  {
    cout<<"Element deleted from queue is : "<< queue[front] <<endl;
```

```
    front++;;
  }
}
void Display()
{
  if (front == - 1)
  cout<<"Queue is empty"<<endl;
  else
  {
    cout<<"Queue elements are : ";
    for (int i = front; i <= rear; i++)
    cout<<queue[i]<<" ";
      cout<<endl;
  }
}
int main()
{
  int ch;
  cout<<"1) Insert element to queue"<<endl;
  cout<<"2) Delete element from queue"<<endl;
  cout<<"3) Display all the elements of queue"<<endl;
  cout<<"4) Exit"<<endl;
  do
  {
    cout<<"Enter your choice : "<<endl;
    cin>>ch;
    switch (ch)
    {
      case 1: Insert();
      break;
      case 2: Delete();
      break;
      case 3: Display();
      break;
      case 4: cout<<"Exit"<<endl;
      break;
      default: cout<<"Invalid choice"<<endl;
    }
  }
  while(ch!=4);
  return 0;
}
```

Queue Using Linked List:

```
#include <iostream>
using namespace std;
struct node
{
  int data;
  struct node *next;
};
struct node* front = NULL;
struct node* rear = NULL;
struct node* temp;
void Insert()
{
  int val;
  cout<<"Insert the element in queue : "<<endl;
  cin>>val;
  if (rear == NULL)
  {
    rear = (struct node *)malloc(sizeof(struct node));
    rear->next = NULL;
    rear->data = val;
    front = rear;
  }
  else
  {
    temp=(struct node *)malloc(sizeof(struct node));
    rear->next = temp;
    temp->data = val;
    temp->next = NULL;
    rear = temp;
  }
}
void Delete()
{
  temp = front;
  if (front == NULL)
  {
    cout<<"Underflow"<<endl;
    return;
  }
  else
  if (temp->next != NULL)
```

```
  {
    temp = temp->next;
    cout<<"Element deleted from queue is : "<<front->data<<endl;
    free(front);
    front = temp;
  }
  else
  {
    cout<<"Element deleted from queue is : "<<front->data<<endl;
    free(front);
    front = NULL;
    rear = NULL;
  }
}
void Display()
{
  temp = front;
  if ((front == NULL) && (rear == NULL))
  {
    cout<<"Queue is empty"<<endl;
    return;
  }
  cout<<"Queue elements are: ";
  while (temp != NULL)
  {
    cout<<temp->data<<" ";
    temp = temp->next;
  }
  cout<<endl;
}
int main()
{
  int ch;
  cout<<"1) Insert element to queue"<<endl;
  cout<<"2) Delete element from queue"<<endl;
  cout<<"3) Display all the elements of queue"<<endl;
  cout<<"4) Exit"<<endl;
  do
  {
    cout<<"Enter your choice : "<<endl;
    cin>>ch;
    switch (ch)
    {
```

```
        case 1: Insert();
        break;
        case 2: Delete();
        break;
        case 3: Display();
        break;
        case 4: cout<<"Exit"<<endl;
        break;
        default: cout<<"Invalid choice"<<endl;
      }
   }
   while(ch!=4);
   return 0;
}
```

UNIT-5

1. Binary tree and binary search tree operations and traversals.

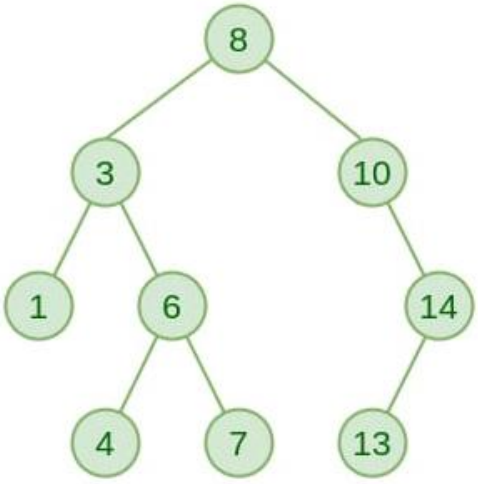
Ans:

Binary Search Tree is a node-based binary tree data structure 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.

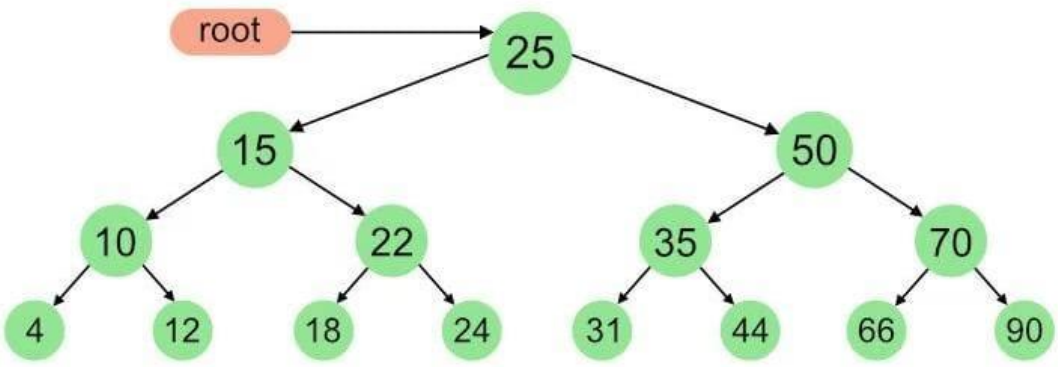
The left and right subtree each must also be a binary search tree.



Operations:

- Create: creates an empty tree.
- Insert: insert a node in the tree.
- Search: Searches for a node in the tree.
- Delete: deletes a node from the tree.
- Inorder: in-order traversal of the tree.
- Preorder: pre-order traversal of the tree.
- Postorder: post-order traversal of the tree.

Traversals of BST:



Inorder Traversal:

1. Traverse the left subtree, i.e., call Inorder(left->subtree)

2. Visit the root.

3. Traverse the right subtree, i.e., call Inorder(right->subtree)

In order traversal for the above-given figure is:

4,10,12,15,18,22,24,25,31,35,44,50,66,70,90

Preorder Traversal

1. Visit the root.

2. Traverse the left subtree, i.e., call Preorder(left->subtree)

3. Traverse the right subtree, i.e., call Preorder(right->subtree)

Preorder traversal for the above-given figure is:

25,15,10,4,12,22,18,24,50,35,31,44,70,66,90

Postorder Traversal

1. Traverse the left subtree, i.e., call Postorder(left->subtree)

2. Traverse the right subtree, i.e., call Postorder(right->subtree)

3. Visit the root

Postorder traversal for the above-given figure is:

4,12,10,18,24,22,15,31,44,35,66,90,70,50,25

2. Graph terminology and graph traversals.

Ans:

Graphs in data structures are non-linear data structures made up of a finite number of nodes or vertices and the edges that connect them. Graphs in data structures are used to address real-world problems in which it represents the problem area as a network like telephone networks, circuit networks, and social networks. For example, it can represent a single user as nodes or vertices in a telephone network, while the link between them via telephone represents edges.

There are 2 types of graphs:

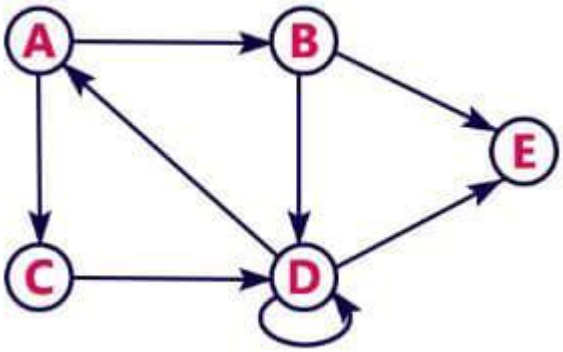
- Directed
- Undirected

Directed graph

A graph with only directed edges is said to be a directed graph.

Example

The following directed graph has 5 vertices and 8 edges. This graph G can be defined as G = (V, E), where V = {A,B,C,D,E} and E = {(A,B), (A,C) (B, E), (B,D), (D, A), (D, E),(C,D),(D,D)}.

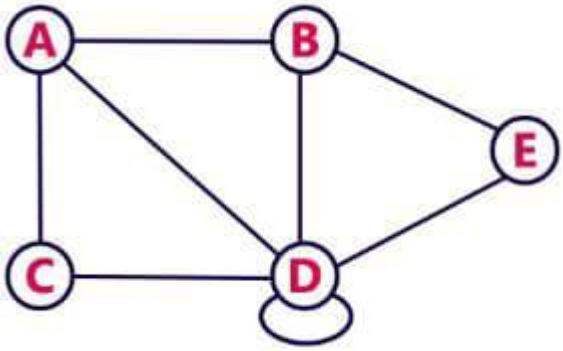


Undirected graph

A graph with only undirected edges is said to be an undirected graph.

Example

The following is an undirected graph.



There are two graph traversal techniques

1. Depth First Search (DFS)
2. Breadth First Search (BFS)

Depth First Search:

DFS stands for Depth First Search, is one of the graph traversal algorithms that use Stack data structure. In DFS Traversal go as deep as possible of the graph and then backtrack once reached a vertex that has all its adjacent vertices already visited.

Breadth First Search:

Breadth-first search graph traversal techniques use a queue data structure as an auxiliary data structure to store nodes for further processing. The size of the queue will be the maximum total number of vertices in the graph.

UNIT-6

1. How 0/1 knapsack problem can be solved by using dynamic program approach.

Ans:

```
#include <iostream>
#include <climits>
using namespace std;
int knapSack(int v[], int w[], int n, int W)
{
if (W < 0)
return INT_MIN;
if (n < 0 || W == 0)
return 0;
int in = v[n] + knapSack(v, w, n - 1, W - w[n]);
int ex = knapSack(v, w, n - 1, W);
return max (in, ex);
}
int main()
{
```

```
    int v[] = { 10, 20, 30, 40, 60, 70 };
    int w[] = { 1, 2, 3, 6, 7, 4 };
    int W = 7;
    int n = sizeof(v) / sizeof(v[0]);
    cout << "Knapsack value is " << knapSack(v, w, n - 1, W);
    return 0;
}
```

Output:

Knapsack value is 100

2. Algorithm for implementing Egg dropping problem.

Ans:

The following is a description of the instance of this famous puzzle involving N = 2 eggs and a building with K = 36 floors. Suppose that we wish to know which stories in a 36-story building are safe to drop eggs from, and which will cause the eggs to break on landing. We make a few assumptions:

- An egg that survives a fall can be used again.
- A broken egg must be discarded.
- The effect of a fall is the same for all eggs.
- If an egg breaks when dropped, then it would break if dropped from a higher floor.
- If an egg survives a fall, then it would survive a shorter fall.
- It is not ruled out that the first-floor windows break eggs, nor is it ruled out that the 36th-floor does not cause an egg to break.

If only one egg is available and we wish to be sure of obtaining the right result, the experiment can be carried out in only one way. Drop the egg from the first-floor window; if it survives, drop it from the second-floor window. Continue upward until it breaks. In the worst case, this method may require 36 droppings.

Optimal Substructure:

When we drop an egg from floor x, there can be two cases (1) The egg breaks (2) The egg does not break.

1. If the egg breaks after dropping from 'xth' floor, then we only need to check for floors lower than 'x' with remaining eggs as some floors should exist lower than 'x' in which the egg would not break, so the problem reduces to x-1 floors and n-1 eggs.
2. If the egg doesn't break after dropping from the 'xth' floor, then we only need to check for floors higher than 'x'; so the problem reduces to 'k-x' floors and n eggs.

Since we need to minimize the number of trials in the worst case, we take a maximum of two cases. We consider the max of the above two cases for every floor and choose the floor which yields the minimum number of trials.

Below is the illustration of the above approach:

K ==> Number of floors

N ==> Number of Eggs

eggDrop(N, K) ==> Minimum number of trials needed to find the critical floor in worst case.

eggDrop(N, K) = 1 + min{max(eggDrop(N – 1, x – 1), eggDrop(N, K – x)), where x is in {1, 2, …, K}}

3. Suppose there is a building with 'n' floors, if we have 'm' eggs, then how can we find the minimum number of drops needed to find a floor from which it is safe to drop an egg without breaking it.

Ans:

```
#include <iostream>
#include<climits>
using namespace std;
int max(int a, int b) { return (a > b) ? a : b; }
int eggDrop(int n, int k)
{
    if (k == 1 || k == 0)
            return k;
    if (n == 1)
            return k;
    int min = INT_MAX, x, res;
    for (x = 1; x <= k; x++)
    {
            res = max(eggDrop(n - 1, x - 1), eggDrop(n, k - x));
            if (res < min)
                    min = res;
    }
    return min + 1;
}
int main()
{
    int n = 2, k = 10;
    cout << "Minimum number of trials in worst case with "
            << n << " eggs and " << k << " floors is "
            << eggDrop(n, k) << endl;
    return 0;
}
```

Output:

Minimum number of trials in worst case with 2 eggs and 10 floors is 4