

Data Structures

Introduction and time complexity

An **algorithm** is the step-by-step unambiguous instructions to solve a given problem.

1. The set of instructions should produce correct output.
2. Should take minimum possible resources.

Five steps for thinking for an algorithm:

1. Constraints
2. Ideas Generation
3. Complexities
4. Coding
5. Testing

Constraints

1. Knowing the algorithms is not sufficient to designing a good software.
2. There may be some data, which is missing that you need to before beginning to solve a problem.
3. In this step, we will capture all the constraints about the problem. We should never try to solve a problem that is not completely defined.

For example : When the problem statement says that write an algorithm to sort numbers.

Basic guideline for the Constraints:

- 1. What is data type of values needs to sort**
 - 2. If the data is numeric (int or float)**
 - 1. How many numbers of elements in the array**
 - 2. What is the range of value in each element.**
 - 3. Does the array contain unique data or not?**
- String data**
- 1. Min and max length**
 - 2. Unique data or not**

Idea Generation

Following is the strategy that you need to follow to solve an unknown problem:

1. Try to simplify the task in hand.
2. Think of a suitable data-structure.
3. Think about similar problems you have already solved.

Complexities

1. The solution should be fast and should have reasonable memory requirement.
2. You should be able to do Big-O analysis.
3. Sometime taking some bit more space saves a lot of time and make your algorithm much faster.

Coding

1. Now, after capturing all the constraints of the problem, deriving few solutions, evaluating the complexities of the various solutions, you can pick one solution to do final coding.
2. Never ever, jump into coding before discussing constraints, Idea generation and complexity
3. Small functions need to be created so that the code is clean and managed.

Testing

1. Once coding is done ,it is a good practice that you go through your code line by line with some small test case. This is just to make sure your code is working as it is supposed to work.

Algorithm analysis: Algorithm analysis helps us to determine which algorithm is most efficient in terms of time and space consumed.

Goal of the Analysis of Algorithms: The goal of the analysis of algorithms is to compare algorithms (or solutions) mainly in terms of running time but also in terms of other factors (e.g., memory, developer effort, etc.)

To compare algorithms: will you consider

Execution times ?

Number of statements ?

Rate of growth ?

Types of Analysis

- **Worst case**
 - Defines the input for which the algorithm takes a long time (slowest time to complete).
 - Input is the one for which the algorithm runs the slowest.
- **Best case**
 - Defines the input for which the algorithm takes the least time (fastest time to complete).
 - Input is the one for which the algorithm runs the fastest.
- **Average case**
 - Provides a prediction about the running time of the algorithm.
 - Run the algorithm many times, using many different inputs that come from some distribution that generates these inputs, compute the total running time (by adding the individual times), and divide by the number of trials.
 - Assumes that the input is random.

For a given algorithm, we can represent the best, worst and average cases in the form of expressions. As an example, let $f(n)$ be the function which represents the given algorithm.

$$f(n) = n^2 + 500 \quad \text{for worst case}$$

$$f(n) = n + 100n + 500 \quad \text{for best case}$$

Big-O Notation [Upper Bounding Function]

Omega-Q Notation [Lower Bounding Function]

Theta-Θ Notation [Order Function]

We generally focus on the upper bound (O) because knowing the lower bound Ω of an algorithm is of no practical importance, and we use the Theta notation if the upper bound (O) and lower bound (Ω) are the same.

$$T(n) = 2n^2 + 3n + 1$$

- Drop lower order terms
- Drop all the constant multipliers

$$\begin{aligned} T(n) &= \underline{2n^2} + \underline{3n+1} \\ &\approx O(n^2) \end{aligned}$$

1) Loop

```
for( i = 1; i<=n; i++ ){
    x=y+z;
}
```

2) Nested Loop

```
for( i = 1; i<=n; i++ ){
    for( j = 1; j<=n; j++ ){
        x=y+z;
    }
}
```

```
for( i = 1; i<=n; i++ ){
    x=y+z; //constant time = C
}
= Cn .
=  $\mathcal{O}(n)$ 
```

```
for( i = 1; i<=n; i++ ){ //n times
    for( j = 1; j<=n; j++ ){ //n times
        x=y+z; //constant time
    }
}
=  $\mathcal{O}(n^2)$ 
```

4) If-else statements

```
if(condition)
{
    - - -  $\mathcal{O}(n)$ 
}
else *
{
    - - -  $\mathcal{O}(n^2)$ 
}
```

```
if(condition)
{
    - - -  $\mathcal{O}(n)$ 
}
=  $\mathcal{O}(n^2)$ 
else ✓
{
    - - -  $\mathcal{O}(n^2)$ 
}
```

3) Sequential Statements

i) $a = a + b;$ // constant time = c_1

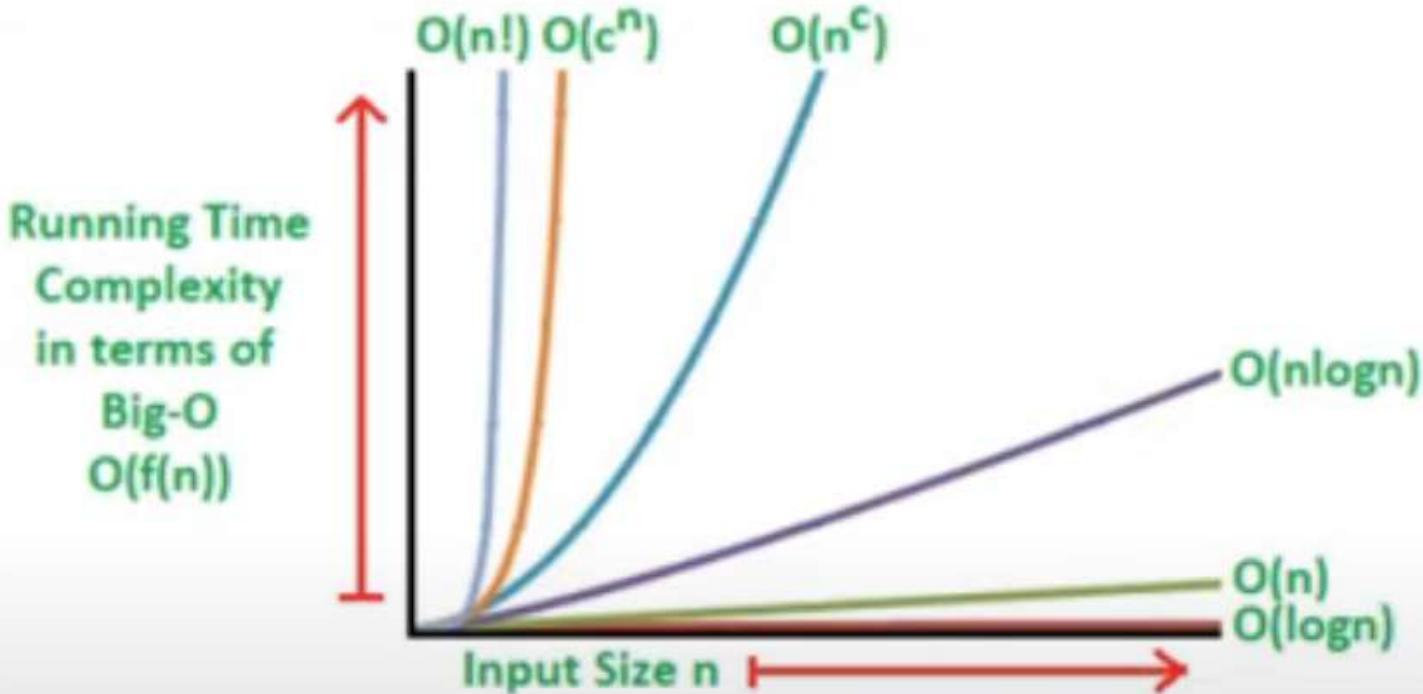
ii) $for(i = 1; i \leq n; i++)\{$ $x = y + z;$ $c_2 n$
 $}$

iii) $for(j = 1; j \leq n; j++)\{$ $c = d + e;$ $c_3 n$
 $}$

i) $a = a + b;$ // constant time = c_1

ii) $for(i = 1; i \leq n; i++)\{$ $x = y + z;$ $c_2 n$ $= c_1 + c_2 n + c_3 n$
 $}$

iii) $for(j = 1; j \leq n; j++)\{$ $c = d + e;$ $c_3 n$
 $}$ $= \mathcal{O}(n)$



$O(n!), O(c^n), O(n^c)$ - Worst

$O(n \log n)$ - Bad

$O(n)$ - Fair

$O(\log n)$ - Good

$O(1)$ - Best

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^c) < O(n!)$$

```
if(i > j ){
    j>23 ? cout<<j : cout<<i;
}
```

O(1)

```
for(i= 0 ; i < n; i++){
    cout<< i << " ";
    i++;
}
```

O(n)

```
for(i= 0 ; i < n; i++){
    for(j = 0; j<n ;j++){
        cout<< i << " ";
    }
}
```

O(n^2)

```
int i = n;
while(i){
    cout << i << " ";
    i = i/2;
}
```

O(log n)

```
for(i= 0; i < n; i++){
    for(j = 1; j < n; j = j*2){
        cout << i << " ";
    }
}
```

O($n \log n$)

```
int a = 0;
for (i = 0; i < N; i++) {
    for (j = N; j > i; j--) {
        a = a + i + j;
    }
}
```

$O(N^2)$

```
int i, j, k = 0;
for (i = n / 2; i <= n; i++) {
    for (j = 2; j <= n; j = j * 2) {
        k = k + n / 2;
    }
}
```

$O(n \log n)$

```
int a = 0, i = N;
while (i > 0) {
    a += i;
    i /= 2;
}
```

$O(\log N)$

```
function(int n)
{
    if (n==1)
        return;
    for (int i=1; i<=n; i++)
    {
        for (int j=1; j<=n; j++)
        {
            printf("*");
            break;
        }
    }
}

static void function(int n)
{
    int count = 0;
    for (int i = n / 2; i <= n; i++)
        for (int j = 1; j <= n; j = 2 * j)
            for (int k = 1; k <= n; k = k * 2)
                count++;
}
```

1

```
void function(int n)
{
    int i, count =0;
    for(i=1 ; i*i <=n ; i++)
        count++;
}
```

2

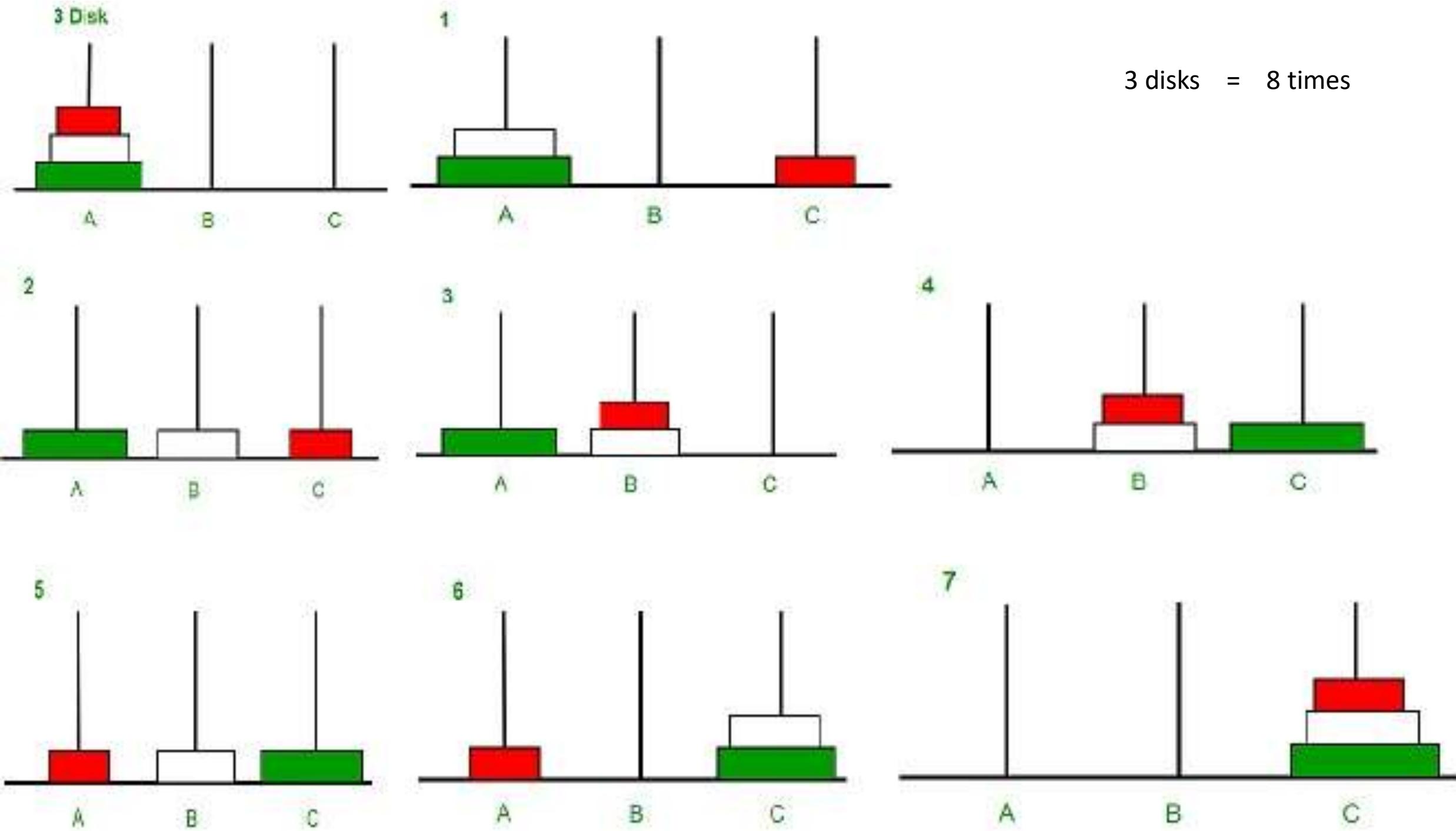
```
void funtion(int n)
{
    for(int I = 1;i<=n ; i++)
        for(int j = 1; j <= n ; j+=i)
            printf("*");
}
```

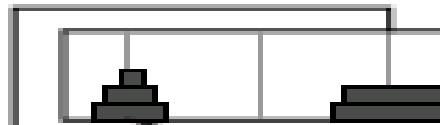
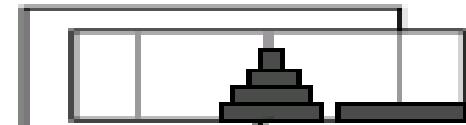
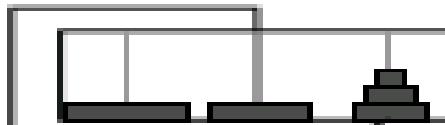
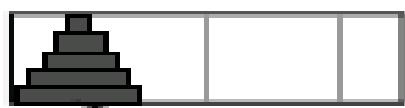
3

```
int fun(int n) {
    int i = 0, j = 0, k = 0, m = 0;
    for (i = 0; i<n ; i++)
        for(j = I; j<n ; j++)
            for(k=j+1 ; k<n ; k++)
                m+=1;
    return m;
}
```

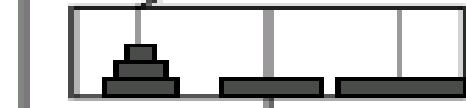
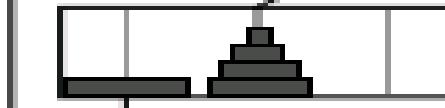
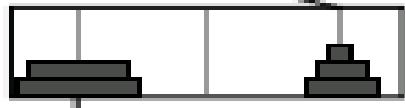
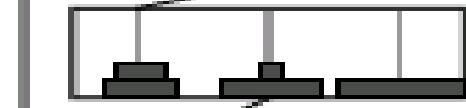
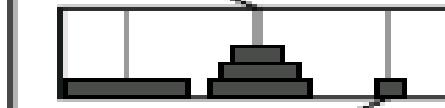
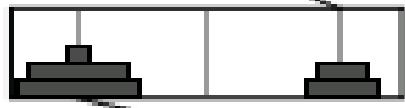
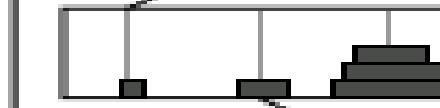
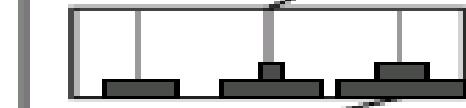
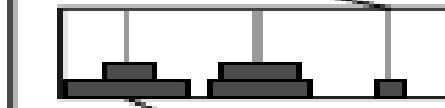
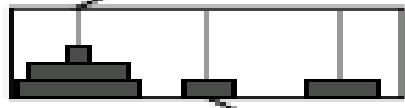
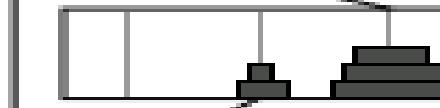
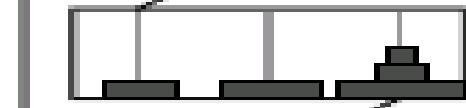
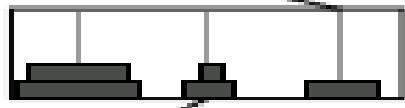
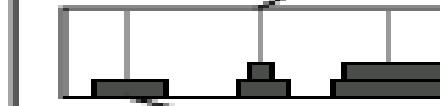
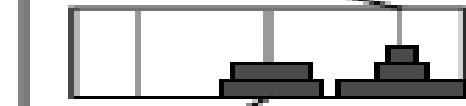
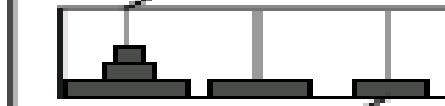
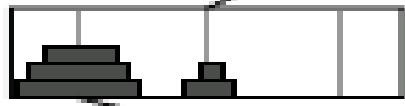
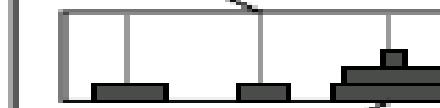
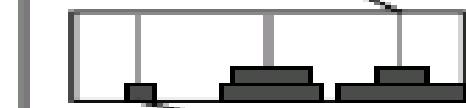
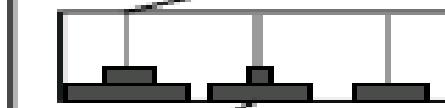
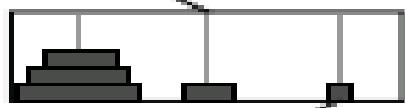
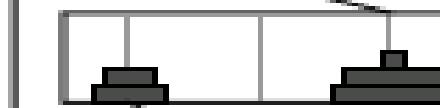
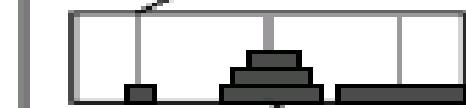
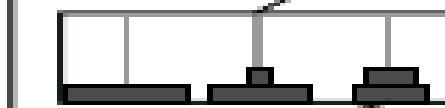
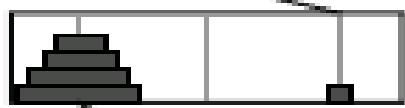
Time complexity with examples:

| Time Complexity | Name | Example |
|-----------------|--------------------|---|
| 1 | Constant | Adding an element to the front of a linked list |
| $\log n$ | Logarithmic | Finding an element in a sorted array |
| n | Linear | Finding an element in an unsorted array |
| $n \log n$ | Linear Logarithmic | Sorting n items by 'divide-and-conquer' - Mergesort |
| n^2 | Quadratic | Shortest path between two nodes in a graph |
| n^3 | Cubic | Matrix Multiplication |
| 2^n | Exponential | The Towers of Hanoi problem |





5 disks
 2^5 times



Data Structures

Data Type:

- A data type defines a domain of allowed values and the operations that can be performed on those values. For example int, float, char data types are provided in C
- If an application needs to use a data type which is not provided as primitive data type of the language, then it is programmer's responsibility to specify the values and operations for that data type and implement it. For example data type for date is not provided in C, and we need dates to be stored and processed in our program then we need to define and implements the date data type.

Abstract Data Type:

- ADT is a concept that defines a data type logically. It specifies a set of data and collection of operations that can be performed on that data.
- ADT only mentions what operations are to be performed but not how these operations will be implemented.
- It does not specify how data will be organized in memory and what algorithms will be used for implementing operations.
- It is called abstract because it gives an implementation independent view.
- ADT is just like a black box which hides the inner structure and design of the data type.

Example of ADT:

List ADT

A list contains elements of same type arranged in sequential order and following operations can be performed on the list:

1. `Initialize()` - Initialize the list to be empty
2. `get()` - returns an element from the list at any given position
3. `insert()` - Inserts a new element at any position of the list
4. `remove()` - Removes the first occurrence of any element from a non empty list
5. `removeAt()` - Removes the element from a specific location from a non empty list
6. `replace()` - Replace an element at any position by another element
7. `size()` - Returns number of elements in the list
8. `isEmpty()` - Returns true if the list is empty, otherwise false
9. `isFull()` - Returns true if the list is full, otherwise false

Data Structures:

- Data structures is a programming construct used to implement an ADT.
- It is physical implementation of ADT.
- It contains collection of variables for storing data specified in the ADT.
- All operations specified in ADT are implemented through functions.

ADT is logical view of data and the operations to manipulate the data while Data Structures is the actual representation of data in memory and the algorithms to manipulate the data

ADT is a logical description while Data Structure is concrete

ADT is what is to be done and data structure is how to do it.

ADT is used by client program.

Advantages of Data Structures are-

Efficiency
Reusability
Abstraction

List of DS:

- Array
- Stack
- Queue
- Linked List
- Binary Tree
- BST
- Hash
- Graph
- searching and sorting algorithms
- time complexity and space complexity

Algorithm classification

- Algorithms that use a similar problem-solving approach can be grouped together
- The purpose is not to be able to classify an algorithm as one type or another, but to highlight the various ways in which a problem can be attacked

A short list of categories

- Algorithm types we will consider include:

- Brute force algorithms
- Greedy algorithms
- Simple recursive algorithms
- Backtracking algorithms
- Divide and conquer algorithms
- Dynamic programming algorithms
- Randomized algorithms

Brute force algorithm

- Brute Force is a straightforward approach of solving a problem based on the problem statement. It is one of the easiest approaches to solve a particular problem.
- It is useful for solving small size dataset problem.
- Some examples of brute force algorithms are:
 - Bubble-Sort · Selection-Sort · Sequential search in an array
 - Computing pow (a, n) by multiplying a, n times · String matching
 - Exhaustive search etc.
- Most of the times, other algorithm techniques can be used to get a better solution of the same problem

Brute force algorithm

- Brute force is the first algorithm that comes into mind when we see some problem.
- They are the simplest algorithms that are very easy to understand.
- These algorithms rarely provide an optimum solution. Many cases we need to find other effective algorithm that is more efficient than the brute force method.
- This is the most simple to understand the kind of problem solving technique

Greedy algorithms

- Greedy algorithms are generally used to solve optimization problems.
- To find the solution that minimizes or maximizes some value (cost/profit/count etc.).
- In greedy algorithm, solution is constructed through a sequence of steps.
 - At each step, choice is made which is locally optimal.
 - We always take the next data to be processed depending upon the dataset which we have already processed and then choose the next optimum data to be processed.
- Greedy algorithms may not always give optimum solution.
- Greedy is a strategy that works well on optimization problems with the following characteristics:
 - 1. Greedy choice: A global optimum can be arrived at by selecting a local optimum.
 - 2. Optimal substructure: An optimal solution to the problem is made from optimal solutions of sub problems.

Greedy algorithms

- Some examples of Greedy algorithms are:
 - Minimal spanning tree: Prim's algorithm, Kruskal's algorithm
 - Dijkstra's algorithm for single-source shortest path
- Approximate solutions:
- Coin exchange problem

Simple recursive algorithms I

- A simple recursive algorithm:
 - Solves the base cases directly
 - Recurs with a simpler subproblem
 - Does some extra work to convert the solution to the simpler subproblem into a solution to the given problem

Example recursive algorithms

- To count the number of elements in a list:
 - If the list is empty, return zero; otherwise,
 - Step past the first element, and count the remaining elements in the list
 - Add one to the result
- To test if a value occurs in a list:
 - If the list is empty, return false; otherwise,
 - If the first thing in the list is the given value, return true; otherwise
 - Step past the first element, and test whether the value occurs in the remainder of the list

Backtracking algorithms

- Backtracking algorithms are based on a depth-first recursive search
- A backtracking algorithm:
 - Tests to see if a solution has been found, and if so, returns it; otherwise
 - For each choice that can be made at this point,
 - Make that choice
 - Recur
 - If the recursion returns a solution, return it
 - If no choices remain, return failure
- Example: searching key of a lock from available bunch of keys.

Divide and Conquer

- Divide-and-Conquer algorithms works by recursively breaking down a problem into two or more subproblems (divide), until these sub problems become simple enough so that can be solved directly (conquer).
- The solution of these sub problems is then combined to give a solution of the original problem.
- Divide-and-Conquer algorithms involve basic three steps
 - Divide the problem into smaller problems.
 - Conquer by solving these problems.
 - Combine these results together.
- In divide-and-conquer the size of the problem is reduced by a factor (half, one-third etc.), While in decrease-and-conquer the size of the problem is reduced by a constant.

Divide and Conquer

- Examples of divide-and-conquer algorithms:
 - Merge-Sort algorithm (recursion)
 - Quicksort algorithm (recursion)
 - Computing the length of the longest path in a binary tree (recursion)
 - Computing Fibonacci numbers (recursion)
- Examples of decrease-and-conquer algorithms:
 - Computing $\text{POW}(a, n)$ by calculating $\text{POW}(a, n/2)$ using recursion
 - Binary search in a sorted array (recursion)
 - Searching in BST

Dynamic programming algorithms

- A dynamic programming algorithm remembers past results and uses them to find new results
- Dynamic programming is generally used for optimization problems
 - Multiple solutions exist, need to find the “best” one
 - Requires “optimal substructure” and “overlapping subproblems”
 - Optimal substructure: Optimal solution contains optimal solutions to subproblems
 - Overlapping subproblems: Solutions to subproblems can be stored and reused in a bottom-up fashion
- . Dynamic Programming (DP) is a simple technique but it can be difficult to master.

Dynamic programming algorithms

- Dynamic programming and memorization work together.
- By using memorization [maintaining a table of sub problems already solved], dynamic programming reduces the exponential complexity to polynomial complexity ($O(n^2)$, $O(n^3)$, etc.) for many problems.
- The major components of DP are:
 - Recursion: Solves sub problems recursively.
 - Memorization: Stores already computed values in table (Memoization means caching).

Dynamic Programming = Recursion + Memorization

Let us take an example

Find Maximum Value Contiguous Subsequence: Given an array of n numbers, give an algorithm for finding a contiguous subsequence $A(i) \dots A(j)$ for which the sum of elements is maximum.

Example: {-2, **11**, -4, **13**, -5, 2} \rightarrow 20 and {1, -3, **4**, -2, -1, **6**} \rightarrow 7

Note: The algorithms doesn't work if the input contains all negative numbers. It returns 0 if all numbers are negative.

Example: {-2, **11**, -4, **13**, -5, 2} → 20 and {1, -3, **4**, -2, -1, **6**} → 7

```
int MaxContiguousSum(int A[], int n) {
    int maxSum = 0;
    for(int i = 0; i < n; i++) // for each possible start point
        for(int j = i; j < n; j++) // for each possible end point
            {
                int currentSum = 0;
                for(int k = i; k <= j; k++)
                    currentSum += A[k];
                if(currentSum > maxSum)
                    maxSum = currentSum;
            }
    return maxSum;
}
```

Time Complexity: $O(n^3)$. Space Complexity: $O(1)$.

Example: {-2, **11**, -4, **13**, -5, 2} → 20 and {1, -3, 4, **-2**, -1, 6} → 7

```
int MaxContiguousSum(int A[], int n) {
    int maxSum = 0;
    for( int i = 0; i < n; i++) {
        int currentSum = 0;
        for( int j = i; j < n; j++) {
            currentSum += a[j];
            if(currentSum > maxSum)
                maxSum = currentSum;
        }
    }
    return maxSum;
}
```

Time Complexity: $O(n^2)$. Space Complexity: $O(1)$.

Example: {-2, 11, -4, 13, -5, 2} → 20 and {1, -3, 4, -2, -1, 6} → 7

```
int MaxContiguousSum(int A[], int n) {
    int M[n] = 0, maxSum = 0;
    if(A[0] > 0)
        M[0] = A[0];
    else M[0] = 0;
    for( int i = 1; i < n; i++ ) {
        if( M[i-1] + A[i] > 0)
            M[i] = M[i-1] + A[i];
        else      M[i] = 0;
    }
    for( int i = 0; i < n; i++ )
        if(M[i] > maxSum)
            maxSum = M[i];
    return maxSum;
}
```

Time Complexity: O(n). Space Complexity: O(n).

Example: $\{-2, 11, -4, 13, -5, 2\} \rightarrow 20$ and $\{1, -3, 4, -2, -1, 6\} \rightarrow 7$

```
int MaxContiguousSum(int A[], int n) {
    int sumSoFar = 0, sumEndingHere = 0;
    for(int i = 0; i < n; i++) {
        sumEndingHere = sumEndingHere + A[i];
        if(sumEndingHere < 0) {
            sumEndingHere = 0;
            continue;
        }
        if(sumSoFar < sumEndingHere)
            sumSoFar = sumEndingHere;
    }
    return sumSoFar;
}
```

Time Complexity: O(n). Space Complexity: O(1).

Examples of Dynamic Programming Algorithms:

- Many string algorithms including longest common subsequence, longest increasing subsequence, longest common substring, edit distance.
- Algorithms on graphs can be solved efficiently: Bellman-Ford algorithm for finding the shortest distance in a graph
- Subset Sum

Randomized algorithms

- A randomized algorithm uses a random number at least once during the computation to make a decision
 - Example: In Quicksort, using a random number to choose a pivot
 - Example: Trying to factor a large prime by choosing random numbers as possible divisors



Thank You!!!!



Recursion

Recursive function

- A recursive function is a function that calls itself, directly or indirectly.
- A recursive function consists of two parts:
Termination Condition and Body (which include recursive expansion).
 1. Termination Condition: A recursive function always contains one or more terminating condition. A condition in which recursive function is processing as a simple case and do not call itself.
 2. Body (including recursive expansion): The main logic of the recursive function contained in the body of the function. It also contains the recursion expansion statement that in turn calls the function itself.

Recursive function

All recursive functions works in 2 phases –

Winding phase: It begins when the recursive function is called for the first time, and each recursive call continues the winding phase.

In this phase function keeps on calling itself and no return statements are executed.

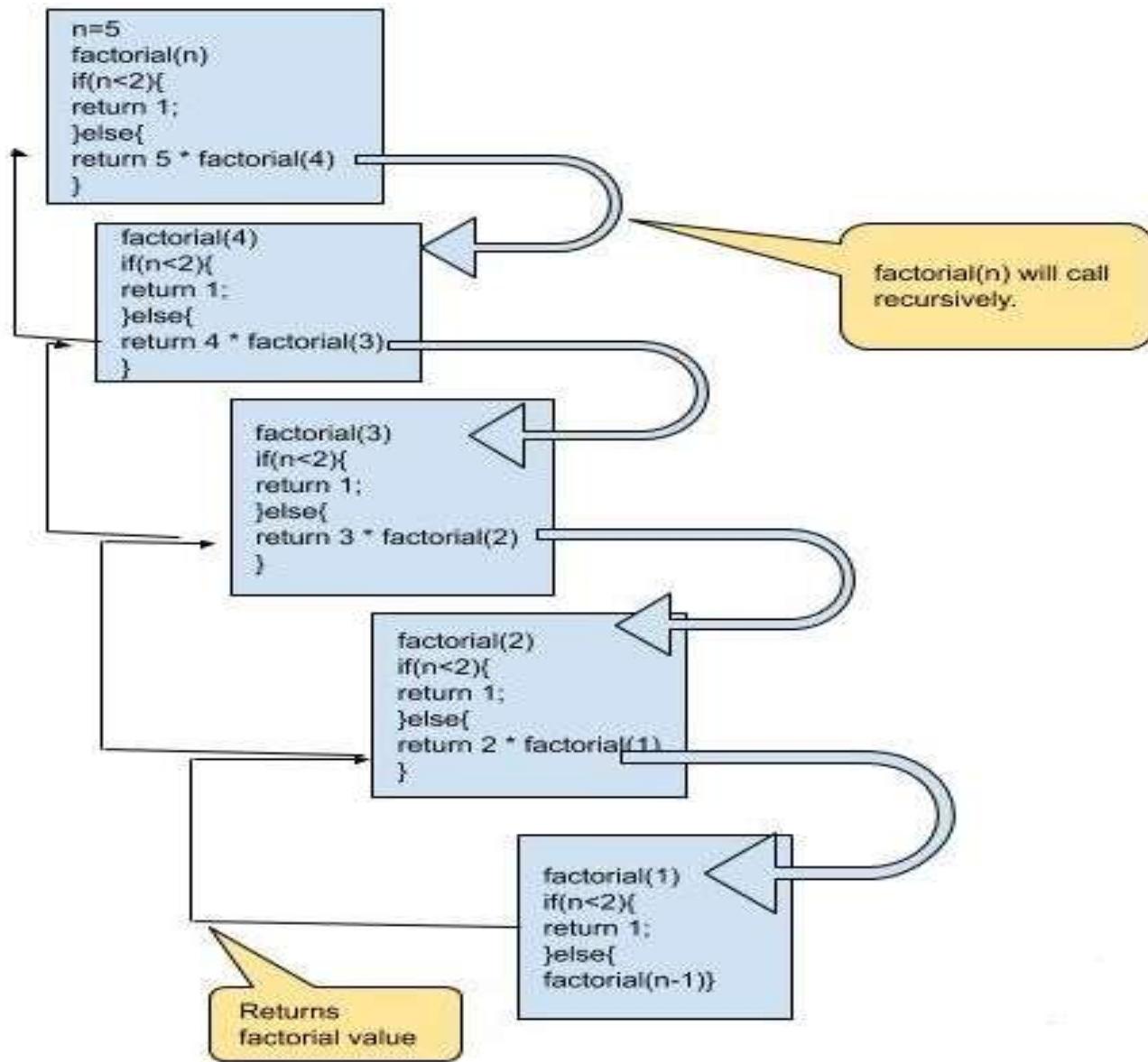
This phase terminates when the terminating condition becomes true in a call

Recursive function

Unwinding phase: It begins when the termination condition become true and function calls starts returning in reverse order till the first instance of function returns.

In this phase the control returns through each instance of function.

In some algorithms we need to perform some work while returning from recursive calls, so we write that particular code just after the recursive call



```

main()
{
    sysout(fact(5));
}

public static int fact(int n)
{
    if(n<2) return 1;
    else return n * fact(n-1);
}

```

```
main()
{
    fun(1);
}
```

forward recursive statement: The statements which gets executed during winding phase.

```
void fun(int n)
{
    if(n>5)
        return;
    sysout(n); //forward recursive
    fun(n+1);
    return;
}
```

backward recursive statement: The statements which gets executed during unwinding phase.

```
void fun(int n)
{
    if(n>5)
        return;
    fun(n+1);
    sysout(n); //backward recursive
    return;
}
```

Properties of recursive algorithm

- 1) A recursive algorithm must have a termination condition.
- 2) A recursive algorithm must change its state, and move towards the termination condition. Without termination condition, the recursive function may run forever and will finally consume all the stack memory
- 3) A recursive algorithm must call itself.

Note: The speed of a recursive program is slower because of stack overheads. If the same task can be done using an iterative solution (loops), then we should prefer an iterative solution (loops) in place of recursion to avoid stack overhead.

Why Recursion?

- Recursive code is generally shorter and easier to write than iterative code.
- Recursion is most useful for tasks that can be defined in terms of similar subtasks.
- For example, sort, search, and traversal problems often have simple recursive solutions.

Important points

- Recursive algorithms have two types of cases, recursive cases and base cases.
- Every recursive function case must terminate at a base case.
- Generally, iterative solutions are more efficient than recursive solutions [due to the overhead of function calls].
- A recursive algorithm can be implemented without recursive function calls using a stack, but it's usually more trouble.
- Some problems are best suited for recursive solutions while others are not.

Example Algorithms of Recursion

- Fibonacci Series, Factorial Finding
- Merge Sort, Quick Sort
- Binary Search
- Tree Traversals and many Tree Problems: InOrder, PreOrder PostOrder
- Graph Traversals: DFS [Depth First Search] and BFS [Breadth First Search]
- Divide and Conquer Algorithms

Some more examples

```
P s v main()
{
    int arr[]={10,20,30,40,50,60,70,80,90,100};
    display(arr,0);
}

static void display(int a[],int i)
{
    if(i>9)
        return;
    display(a,i+1);
    sysout(a[i]);
}
```

Some more examples

```
P s v main()
{
    int arr[] = {10,20,30,40,50,60,70,80,90,100};
    sysout("Sum = " + sum(arr,0));
}

static int sum(int []a,int i)
{
    if(i>9)
        return 0;
    sysout(a[i]);
    return a[i] + sum(a,i+1);
}
```

Some more examples

```
public static int cnt(int n)
{
    if(n/10 == 0) return 1;

    return 1 + cnt(n/10);
}
```

```
public static int sum_of_digit(int n)
{
    if(n/10 == 0) return n;

    return n%10 + sum_of_digit(n/10);
}
```

```
public static void displayR(int n)
{
    if(n/10==0)
    {
        sysout(n);
        return;
    }
    sysout(n%10);
    displayR(n/10);
}
```

```
public static int power(int a, int p)
{
    if(p==0) return 1;

    return a * power(a,p-1);
}
```

LinkedList and recursive function

```
static void disp(listNode temp)
{
    if(temp==null) return;
    disp(temp.getNext());
    sysout(temp.getData());
}
```

Function Call:

```
disp(ll.getHead());
```

```
public static listNode reverse(listNode head)
{
    listNode temp;
    if(head.getNext()==null) return head;
    temp=reverse( head.getNext());
    head.getNext().setNext(head);
    head.setNext(null);
    return temp;
}
```

Function Call:

```
list.setHead(reverse(list.getHead()));
sysout(list);
```

LinkedList and recursive function

```
int length(listNode *p)
{
    if(p == null) return 0;
    return 1 + length(p.getNext());
}
```

Function Call:

```
sysout(ll.length(ll.getHead()));
```

```
int sum_nodes(listNode *p)
{
    if(p == null) return 0;
    return p.getData() + sum_nodes(p.getNext());
}
```

Function Call:

```
sysout(ll.sum_nodes(ll.getHead()));
```

```
package for_DSA;

import java.util.Scanner;

class listNodeInt {
    private int data;
    private listNodeInt next;

    public listNodeInt()
    {
        data=0;
        next=null;
    }

    public listNodeInt(int d)
    {
        data=d;
        next=null;
    }

    public void setData(int d)
    {
        data=d;
    }

    public void setNext(listNodeInt n)
    {
        next=n;
    }

    public int getData()
    {
        return data;
    }

    public listNodeInt getNext()
    {
        return next;
    }
}

class linkedlist_int{
    protected listNodeInt head;

    public linkedlist_int()
    {
        head=null;
    }

    public listNodeInt getHead()
    {
        return head;
    }

    public void setHead(listNodeInt h)
    {
        head=h;
    }
}
```

```

public void insert_last(int d)
{
    ListNodeInt new_node=new ListNodeInt(d);

    if(head==null)
    {
        head=new_node;
        return;
    }

    ListNodeInt iter=head;
    while(iter.getNext() !=null)
        iter=iter.getNext();

    iter.setNext(new_node);

    return;
}

public void insert_first(int d)
{
    ListNodeInt new_node=new ListNodeInt(d);

    new_node.setNext(head);
    head=new_node;
    return;
}

public void insert_at_pos(int d, int pos)
{
    ListNodeInt new_node=new ListNodeInt(d);

    if(head==null)
    {
        head=new_node;
        return;
    }

    if(pos==1)
    {
        new_node.setNext(head);
        head=new_node;
        return;
    }

    ListNodeInt iter=head;

    for(int i=1;i<pos-1 && iter.getNext() !=null;i++)
        iter=iter.getNext();

    new_node.setNext(iter.getNext());
    iter.setNext(new_node);
    return;
}

public int delete_first()
{
    int d=-999;

```

```

listNodeInt deletable;
if(head==null)
{
    System.out.print("invalid..");
    return d;
}
if(head.getNext()==null)
{
    d=head.getData();
    head=null;
    return d;
}
deletable=head;
head=head.getNext();
d=deletable.getData();
deletable=null;
return d;
}

public int delete_last()
{
    listNodeInt deletable;
    int d=-999;
    if(head==null)
    {
        System.out.print("invalid..");
        return d;
    }
    if(head.getNext()==null)
    {
        d=head.getData();
        head=null;
        return d;
    }
    listNodeInt iter=head;

    while(iter.getNext().getNext()!=null)
        iter=iter.getNext();

    deletable = iter.getNext();
    iter.setNext(null);
    d=deletable.getData();
    deletable=null;
    return d;
}

public int delete_by_pos(int pos)
{
    int d=-999;
    if(head==null)
    {
        System.out.print("invalid..");
        return d;
    }
    if(pos==1)
    {
        d=head.getData();
        head=head.getNext();
        return d;
    }
}

```

```

        }
        listNodeInt deletable,iter=head;
        for(int i=1;i<pos-1 && iter.getNext()!=null;i++)
            iter=iter.getNext();

        if(iter.getNext()!=null)
        {
            deletable=iter.getNext();
            d=deletable.getData();
            iter.setNext(iter.getNext().getNext());
        }
        else
            System.out.print("\n Invalid position.. cannot
delete...\n");

        return d;
    }

    public void insert_before(int d,int before)
    {
        listNodeInt new_node= new listNodeInt(d);

        if((head==null) || (before == head.getData()))
        {
            new_node.setNext(head);
            head=new_node;
            return;
        }
        listNodeInt iter=head;
        while(iter.getNext()!=null &&
iter.getNext().getData()!=before)
            iter=iter.getNext();

        new_node.setNext(iter.getNext());
        iter.setNext(new_node);
        return;
    }

    public String toString()
    {
        String str=new String();
        listNodeInt iter = head;
        if(iter==null)
            return "List is Empty\n";

        str="\n List : ";
        while(iter!=null)
        {
            str=str+" "+iter.getData();
            iter = iter.getNext();
        }

        str=str+"\n";
        return str;
    }

    public class LinkedList_Int_Main {

```

```

public static void main(String[] args) {
    linkedlist_int ll = new linkedlist_int();
    Scanner sc=new Scanner(System.in);
    int choice, data, pos, cnt;
    do {
        System.out.print("1. Insert First\n");
        System.out.print("2. Insert Last \n");
        System.out.print("3. Insert by position \n");
        System.out.print("4. Insert before data \n");
        System.out.print("5. Delete First \n");
        System.out.print("6. Delete Last \n");
        System.out.print("7. Delete by position \n");
        System.out.print("8. Print LL \n");
        System.out.print("9. Exit \n");

        choice = sc.nextInt();
        switch(choice)
        {
            case 1:
                System.out.print("Enter data to insert : ");
                data=sc.nextInt(); //20
                ll.insert_first(data);
                break;
            case 2:
                System.out.print("Enter data to insert : ");
                data=sc.nextInt();
                ll.insert_last(data);
                break;
            case 3:
                System.out.print("Enter data to insert : ");
                data=sc.nextInt();
                System.out.print("Enter position : ");
                pos=sc.nextInt();
                ll.insert_at_pos(data, pos);
                break;
            case 4:
                System.out.print("Enter data to insert : ");
                data=sc.nextInt();
                System.out.print("Enter data before which to be
inserted : ");
                pos=sc.nextInt();
                ll.insert_before(data, pos);
                break;
            case 5:
                data = ll.delete_first();
                System.out.print("Data deleted : " + data
+"\\n");
                break;
            case 6:
                data = ll.delete_last();
                System.out.print("Data deleted : " + data
+"\\n");
                break;
            case 7:
                System.out.print("Enter position to delete : ");
                pos=sc.nextInt();
                data = ll.delete_by_pos(pos);
                System.out.print("Data deleted : " + data
+"\\n");
        }
    }
}

```

```
        break;
    case 8:
        System.out.print(l1);
        break;
    case 9: break;
    default: System.out.print("Invalid choice..\n");
} //end of switch
}while(choice!=9);
sc.close();
}//end of main

}
```

```

1. count number of nodes in SinglyLL

public int countNode()
{
    int cnt;
    SListNode iter;

    for(cnt=0,iter=head; iter!=null ; cnt++,iter=iter.getNext()) ;

    return cnt;
}

public SListNode middle_ref()
{
    int cnt = countNode();
    SListNode iter=head;

    for(int i=1 ; i<=cnt /2 ; i++,iter=iter.getNext());

    return iter;
}

public int middle_val()
{
    int cnt = countNode();
    SListNode iter=head;

    for(int i=1 ; i<=cnt /2 ; i++,iter=iter.getNext());

    return iter.getData();
}

public void disp_alternate_node()
{
    SListNode iter = head;

    while(iter!=null)
    {
        sysout(iter.getData());
        if(iter.getNext()==null)
            break;
        iter = iter.getNext().getNext();

    }
}

public void dele_alternate_node()
{
    SListNode iter = head, deletable;

    while(iter!=null)
    {
        deletable = iter.getNext();
        if(deletable!=null)
            iter.setNext(deletable.getNext());
    }
}

```

```

        iter=iter.getNext();
        deletable = null;
    }
}

public void reverse()
{
    SListNode prev, curr, next;

    prev=null;
    curr=head;

    while(curr!=null)
    {
        next = curr.getNext();
        curr.setNext(prev);
        prev = curr;
        curr = next;
    }
    head=prev;
}

```

stack using LL

use class SListNode for class StackLL

```

class SListNodeChar
{
    char ch;
    SListNodeChar next;

    public SListNodeChar()
    {
        ch = ' ';
        next = null;
    }

    public SListNodeChar(char c)
    {
        ch = c;
        next = null;
    }

    public void setData(char c)
    {
        ch = c;
    }

    public char getData()
    {
        return ch;
    }

    public SListNodeChar getNext()
    {
        return next;
    }
}

```

```

        }
    public void setNext(SListNodeChar n)
    {
        next = n;
    }
}

class StackLL {

    SListNodeChar top;

    public StackLL() { top = null; }

    public void push(char d)
    {
        SListNodeChar new_node = new SListNodeChar(d);

        if(top == null)
        {
            top = new_node;
            return;
        }
        new_node.setNext(top);
        top = new_node;
        return;
    }

    public char pop()
    {
        char d='@';
        SListNodeChar deletable;

        if(top==null)
        {
            sysout("Stack underflow...");
            return d;
        }
        d = top.getData();

        deletable = top;
        top = top.getNext();
        deletable = null;
        return d;
    }

    public char peek()
    {
        char d='@';
        SListNodeChar deletable;

        if(top==null)
        {
            sysout("Stack underflow...");

```

```

        return d;
    }
    d = top.getData();

    return d;
}

public boolean isEmpty()
{
    if(top == null)
        return true;
    else
        return false;
}

}

class QueueLL{
    SListNode front;

    public QueueLL() { front = null; }

    public void Enqueue(int d)
    {
        SListNode new_node = new SListNode(d);
        SListNode iter;

        if(front==null)
        {
            front = new_node;
            return;
        }
        iter = front;

        while(iter.getNext()!=null)
            iter=iter.getNext();

        iter.setNext(new_node);
    }

    return;
}

public int Dequeue()
{
    SListNode deletable;
    int d=-999;

    if(front==null)
    {
        retrun d;
    }

    d=front.getData();
    deletable = front;
}

```

```
    front = front.getNext();
    deletable = null;
    return d;
}

public boolean isEmpty()
{
    if(front == null)
        return true;
    else
        return false;
}
```



```

package Infoway_code;

import java.util.Scanner;
class SListNode{
    int data;
    SListNode next;

    public SListNode() {
        data = 0;
        next = null;
    }

    public SListNode(int d) {
        data = d;
        next = null;
    }
    public int getData() {
        return data;
    }
    public void setData(int d) {
        data = d;
    }
    public SListNode getNext() {
        return next;
    }
    public void setNext(SListNode n) {
        next = n;
    }
}

class CircularLL {

    SListNode last;

    public CircularLL()
    {
        last = null;
    }

    public void insert_first(int d)
    {
        SListNode new_node = new SListNode(d);

        if(last == null)
        {
            last = new_node;
            last.setNext(last);
            return;
        }
        new_node.setNext(last.getNext());
        last.setNext(new_node);
        return;
    }

    public void insert_last(int d)
    {
        SListNode new_node = new SListNode(d);

```

```

        if(last == null)
        {
            last = new_node;
            last.setNext(last);
            return;
        }

        new_node.setNext(last.getNext());
        last.setNext(new_node);
        last = new_node;
        return;
    }

    public String toString()
    {
        String str="";
        SListNode iter;

        if(last == null)
        {
            str="Empty..";
            return str;
        }

        str = "\n List ->";

        iter = last.getNext();

        do {
            str = str + "->" + iter.getData();

            iter = iter.getNext();

        }while(iter!=last.getNext());
        str = str + "\n";

        return str;
    }

    public void insert_by_pos(int d, int pos)
    {
        SListNode new_node = new SListNode(d);
        SListNode iter;
        int i;

        if(pos==1)
        {
            if(last == null)
            {
                last = new_node;
                last.setNext(last);
                return;
            }
            else
            {

                new_node.setNext(last.getNext());

```

```

        last.setNext(new_node);
        return;
    }
}

iter=last.getNext();

for(i=1; i<pos-1 && iter.getNext()!=last.getNext() ;
i++,iter=iter.getNext()) ;

if((i==pos-1)&&(iter.getNext()!=last.getNext()))
{
    new_node.setNext(iter.getNext());
    iter.setNext(new_node);
    return;
}

if((i==pos-1) && ( iter==last))
{
    new_node.setNext(last.getNext());
    last.setNext(new_node);
    last = new_node;
    return;
}

}
}

public int del_first()
{
    int d = -999;
    if(last==null)
    {
        return d;
    }

    if(last.getNext()==last)
    {
        last.setNext(null);
        d=last.getData();
        last=null;

        return d;
    }

    SListNode deletable = last.getNext();

    last.setNext(deletable.getNext());
    d=deletable.getData();

    deletable = null;
    return d;
}

}

public class CirculcarLLMain {

    public static void main(String[] args) {
        CircularLL ll = new CircularLL();
        Scanner sc=new Scanner(System.in);

```

```

int choice, data, pos, cnt;
do {
    System.out.print("1. Insert First\n");
    System.out.print("2. Insert Last \n");
    System.out.print("3. Insert at position \n");
    System.out.print("4. Delete data \n");
    System.out.print("5. Print LL \n");
    System.out.print("6. Exit \n");

    choice = sc.nextInt();
    switch(choice)
    {
        case 1:
            System.out.print("Enter data to insert :
") ;
            data=sc.nextInt(); //20
            ll.insert_first(data);
            break;
        case 2:
            System.out.print("Enter data to insert :
") ;
            data=sc.nextInt();
            ll.insert_last(data);
            break;
        case 3:
            System.out.print("Enter data to insert :
") ;
            data=sc.nextInt();
            System.out.print("Enter position : ");
            pos=sc.nextInt();
            ll.insert_by_pos(data, pos);
            break;
        case 4:
            //System.out.print("Enter data to insert :
") ;
            //data=sc.nextInt();
            data=ll.del_first();
            System.out.print("Data deleted : " + data
+"\\n");
            break;
        case 5:
            System.out.print(ll);
            break;
        case 6: break;
        default: System.out.print("Invalid
choice..\n");
    } //end of switch
}while(choice!=6);
}
}

```

Doubly LinkedList

```

package Infoway_code;

import java.util.Scanner;

```

```

class DblyNode {
    int data;
    DblyNode prev, next;

    public DblyNode()
    {
        data = 0;
        prev = next = null;
    }

    public DblyNode(int d)
    {
        data = d;
        prev = next = null;
    }

    int getData()
    {
        return data;
    }

    DblyNode getPrev()
    {
        return prev;
    }

    DblyNode getNext()
    {
        return next;
    }

    void setData(int d)
    {
        data = d;
    }

    void setPrev(DblyNode p)
    {
        prev = p;
    }

    void setNext(DblyNode n)
    {
        next = n;
    }
}

class DblyLinkedList{
    DblyNode head;

    public void createList(int n)
    {
        int i,val;
        Scanner sc = new Scanner(System.in);
        for(i=1;i<=n;i++)
        {
            System.out.println("Enter data.");

```

```

        val = sc.nextInt();

        insert_last(val);
    }

}

void insert_last(int d)
{
    DblyNode new_node = new DblyNode(d);

    if(head == null)
    {
        head = new_node;
        return;
    }

    DblyNode iter = head;

    while(iter.getNext() != null)
        iter = iter.getNext();

    new_node.setPrev(iter);
    iter.setNext(new_node);

    return;
}

void insert_first(int d)
{
    DblyNode new_node = new DblyNode(d);

    if(head == null)
    {
        head = new_node;
        return;
    }

    new_node.setNext(head);
    head.setPrev(new_node);
    head = new_node;
    return;
}

public void insert_before(int d,int ele)
{
    DblyNode new_node = new DblyNode(ele);

    if(head==null)
    {
        System.out.println("Empty..");
        return;
    }
    if(head.getData() == d)
    {
        new_node.setNext(head);
        head.setPrev(new_node);
        head=new_node;
    }
}

```

```

        return;
    }
DblyNode iter = head;

while((iter.getData()!=d)&&(iter!=null))
    iter = iter.getNext();

if(iter==null)
{
    System.out.println("Not found..");
    return;
}

new_node.setPrev(iter.getPrev());
new_node.setNext(iter);
iter.getPrev().setNext(new_node);
iter.setPrev(new_node);
return;
}

public String toString()
{
    String str=" ";
    DblyNode iter = head;

    while(iter!=null)
    {
        str = str + "-> " +iter.getData();
        iter = iter.getNext();
    }
    str = str + "\n";
    return str;
}

public int del_by_pos(int p)
{
    int d=-999;
    DblyNode del;
    if(head==null)
    {
        System.out.println("List is empty ...");
        return d;
    }

    if(p==1)
    {
        del = head;
        head = head.getNext();
        if( head !=null) {
            head.setPrev( null );
        }
        d=del.getData();
        del=null;
        return d;
    }
    int i;
    DblyNode iter = head;
}

```

```

        for( i = 1; i < p; i++ ) {
            iter = iter.getNext();
            if( iter == null ) {
                return d;
            }
        }
        iter.getPrev().setNext(iter.getNext());
        if(iter.getNext() !=null)
            iter.getNext().setPrev(iter.getPrev());

        del = iter;
        d = del.getData();
        del = null;
        return d;
    }
}

public class DblyLinkedListMain {

    public static void main(String[] args) {
        DblyLinkedList dl = new DblyLinkedList();
        Scanner sc=new Scanner(System.in);
        int choice, data, pos,ele;
        do {
            System.out.print("1. Create List\n");
            System.out.print("2. Add at beginning \n");
            System.out.print("3. Add at end \n");
            System.out.print("4. insert_before \n");
            System.out.print("5. Print LL \n");
            System.out.print("6. Delete by position \n");
            System.out.print("7. Exit ");
            choice = sc.nextInt();
            switch(choice)
            {
                case 1:
                    dl.createList(5);
                    break;
                case 2:
                    System.out.print("Enter data to insert : ");
                    data=sc.nextInt();
                    dl.insert_first(data);
                    break;
                case 3:
                    System.out.print("Enter data to insert : ");
                    data=sc.nextInt();
                    dl.insert_last(data);
                    break;
                case 4:
                    System.out.print("Enter data before : ");
                    data=sc.nextInt();
                    System.out.print("Enter Element : ");
                    ele=sc.nextInt();
                    dl.insert_before(data, ele);
                    break;
                case 5:
                    System.out.print(dl);
                    break;
                case 6:
            }
        }
    }
}

```

```
        System.out.print("Enter pos to add : ");
        pos=sc.nextInt();
        System.out.println(dl.del_by_pos(pos));
        break;
    case 7:break;
    default: System.out.print("Invalid choice..\n");
    }//end of switch
}while(choice!=7);
sc.close();

}
```

Stack and Queue

Data Structure: Stack

A stack is a simple data structure used for storing data

In a stack, the order in which the data arrives is important.

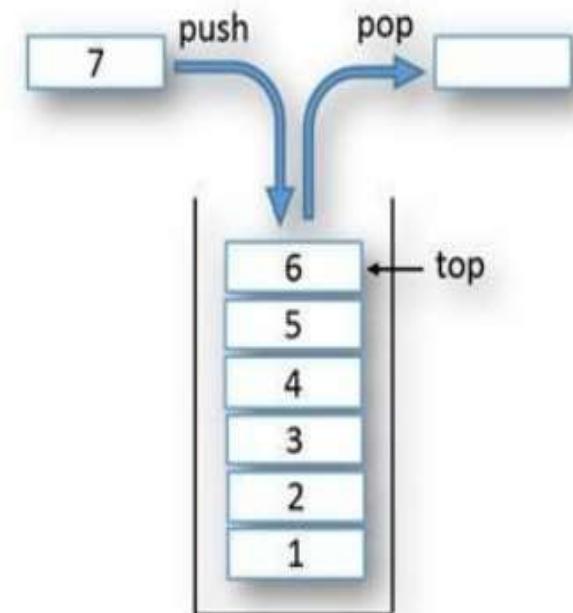
A stack is an ordered list in which insertion and deletion are done at one end, called top.

The last element inserted is the first one to be deleted. Hence, it is called the Last in First out (LIFO) or First in Last out (FILO) list.

Operations on stack

1. When an element is inserted in a stack, the operation is called push.
2. When an element is removed from the stack, the concept is called pop.
3. Returns the last inserted element without removing it, is called peek.
4. Trying to pop out an empty stack is called underflow.
5. Trying to push an element in a full stack is called overflow.

Generally, we treat underflow and overflow as exceptions.



Stack ADT

The following operations make a stack an ADT. (For simplicity, assume the data is an integer type.)

Main stack operations

- Push (int data): Inserts data onto stack.
- int Pop(): Removes and returns the last inserted element from the stack.

Auxiliary stack operations

- int peek(): Returns the last inserted element without removing it.
- int Size(): Returns the number of elements stored in the stack.
- int IsEmptyStack(): Indicates whether any elements are stored in the stack or not.
- int IsFullStack(): Indicates whether the stack is full or not.

Applications of Stack

Following are some of the applications in which stack DS play an important role.

Direct applications

- Balancing of symbols
- Infix-to-postfix conversion
- Evaluation of postfix expression
- Implementing function calls (including recursion)
- Finding of spans (finding spans in stock markets, refer to Problems section)
- Page-visited history in a Web browser [Back Buttons]
- Undo sequence in a text editor
- Matching Tags in HTMLAnd XML

Indirect applications

- Auxiliary data structure for other algorithms (Example: Tree traversal algorithms)

Implementation: There are many ways of implementing stack ADT.

- Simple array based implementation
- Dynamic array based implementation
- Linked lists implementation

Performance:

| | |
|--|--------|
| Space Complexity (for n push operations) | $O(n)$ |
| Time Complexity of Push() | $O(1)$ |
| Time Complexity of Pop() | $O(1)$ |
| Time Complexity of Size() | $O(1)$ |
| Time Complexity of IsEmptyStack() | $O(1)$ |
| Time Complexity of IsFullStack() | $O(1)$ |
| Time Complexity of DeleteStackQ | $O(1)$ |

Data Structure: Queue

A queue is a data structure used for storing data (similar to Linked Lists and Stacks).

In queue, the order in which data arrives is important.

Definition: A queue is an ordered list in which insertions are done at one end (rear) and deletions are done at other end (front).

The first element to be inserted is the first one to be deleted. Hence, it is called First in First out (FIFO) or Last in Last out (LILO) list.

Operations on queue

1. When an element is inserted in a queue, the concept is called EnQueue.
2. when an element is removed from the queue, the concept is called DeQueue .
3. Returns the element at front without removing it, is called peek.
4. DeQueueing an empty queue is called underflow.
5. EnQueueing an element in a full queue is called overflow..

Generally, we treat underflow and overflow as exceptions.



Queue ADT

The following operations make a queue an ADT. (For simplicity, assume the data is an integer type.)

Main queue operations

- EnQueue(int data): Inserts an element at the end of the queue
- int DeQueue(): Removes and returns the element at the front of the queue

Auxiliary stack operations

- int Front(): Returns the element at the front without removing it
- int QueueSize(): Returns the number of elements stored in the queue
- int IsEmptyQueueQ: Indicates whether elements are stored in the queue or not

Applications of Queue

Following are some of the applications that uses queue.

Direct applications

- Operating systems schedule jobs (with equal priority) in the order of arrival (e.g., a print queue).
- Simulation of real-world queues such as lines at a ticket counter or any other first-come first-served scenario requires a queue.
- Multiprogramming.
- Asynchronous data transfer (file IO, pipes, sockets).
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.

Indirect applications

- Auxiliary data structure for algorithms (Example: Tree traversal algorithms)

Implementation: There are many ways of implementing queue ADT.

- Simple array based implementation
- Dynamic array based implementation
- Linked lists implementation

Performance:

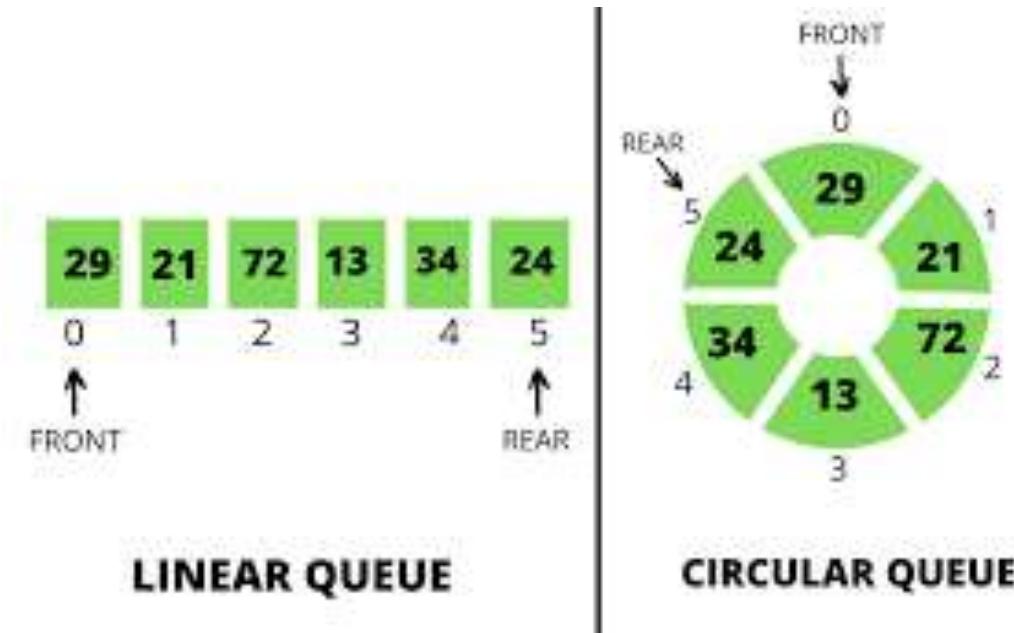
| | |
|--|--------|
| Space Complexity (for n push operations) | $O(n)$ |
| Time Complexity of Push() | $O(1)$ |
| Time Complexity of Pop() | $O(1)$ |
| Time Complexity of Size() | $O(1)$ |
| Time Complexity of IsEmptyStack() | $O(1)$ |
| Time Complexity of IsFullStack() | $O(1)$ |
| Time Complexity of DeleteStackQ | $O(1)$ |

Circular Queue ADT

We reuse the array indexes in circular fashion to insert elements in queue.

Simple array implementation for queue is not efficient

We treat the last element and the first array elements as contiguous.



```
package for_DSA;           /* Stack of int data */

import java.util.Scanner;

class intStack{
    private int arr[];
    private int top;

    public intStack()
    {
        arr=new int[10];
        top=-1;
    }

    public intStack(int s)
    {
        arr=new int[s];
        top=-1;
    }

    public boolean isFull()
    {
        if(top==arr.length-1)
            return true;
        else
            return false;
    }

    public boolean isEmpty()
    {
        if(top==-1)
            return true;
        else
            return false;
    }

    public void push(int data)
    {
        if(!isFull()) //if(top<arr.length-1)
        {
            top=top+1;
            arr[top]=data;
        }
        else System.out.print("stack overflow.. can't push..\n");

        return;
    }

    public int pop()
    {
        int data=-999;
        if(!isEmpty()) //      if(top!=-1)
        {
            data=arr[top];
            top=top-1;
        }
        else System.out.print("stack underflow.. can't pop..\n");

        return data;
    }
}
```

```

public int peek()
{
    int data=-999;
    if(top===-1)
        System.out.print("Stack underflow.. can not perform
peek..\n");
    else
        data=arr[top];

    return data;
}

public String toString()
{
    int i;
    String str=new String();
    if(isEmpty())
    {
        System.out.print("Stack is empty..\n");
        return " ";
    }
    str=str + "Stack elements are: ";
    for(i=top;i>=0;i--)
        str=str+" "+arr[i];

    return str;
}//end of toString
}//end of class

public class intStackMain {

    public static void main(String[] args) {
        int choice, data;
        intStack st=new intStack();
        Scanner sc=new Scanner(System.in);
        do {
            System.out.print("\n1. Push \n");
            System.out.print("2. Pop \n");
            System.out.print("3. Peek \n");
            System.out.print("4. Display \n");
            System.out.print("5. Exit \n");
            choice = sc.nextInt();
            switch(choice)
            {
                case 1:
                    System.out.print("Enter data to be pushed : ");
                    data=sc.nextInt(); //20
                    st.push(data);
                    break;
                case 2:
                    data=st.pop();
                    System.out.print("Popped data : " + data + "\n");
                    break;
                case 3:
                    data=st.peek();
                    System.out.print("Data at top : " + data + "\n");
                    break;
                case 4:

```

```

        System.out.print(st);
        break;
    case 5: break;
    default: System.out.print("Invalid choice..\n");
} //end of switch
}while(choice!=5);
sc.close();
}//end of main
}//end of class

```

```

package for_DSA;           //Stack of student object

import java.util.Scanner;

class Student{
    private int rno;
    String name;
    float marks;
    public Student(int rno, String name, float marks) {
        super();
        this.rno = rno;
        this.name = name;
        this.marks = marks;
    }
    @Override
    public String toString() {
        return "Student [rno=" + rno + ", name=" + name + ", marks=" +
marks + "]";
    }
    public int getRno() {
        return rno;
    }
    public void setRno(int rno) {
        this.rno = rno;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public float getMarks() {
        return marks;
    }
    public void setMarks(float marks) {
        this.marks = marks;
    }
}

class studentStack{
    private Student arr[];
    private int top;

    public studentStack()

```

```

{
    arr=new Student[10];
    top=-1;
}

public studentStack(int s)
{
    arr=new Student[s];
    top=-1;
}

public boolean isFull()
{
    if(top==arr.length-1)
        return true;
    else
        return false;
}

public boolean isEmpty()
{
    if(top==-1)
        return true;
    else
        return false;
}

public void push(Student data)
{
    if(!isFull()) //if(top<arr.length-1)
    {
        top=top+1;
        arr[top]=data;
    }
    else System.out.print("stack overflow.. can't push..\n");

    return;
}

public Student pop()
{
    Student data=null;
    if(!isEmpty()) //      if(top!=-1)
    {
        data=arr[top];
        top=top-1;
    }
    else System.out.print("stack underflow.. can't pop..\n");

    return data;
}

public Student peek()
{
    Student data=null;
    if(top==-1)
        System.out.print("Stack underflow.. can not perform
peek..\n");
    else
        data=arr[top];
}

```

```

        return data;
    }

public String toString()
{
    int i;
    String str=new String();
    if(isEmpty())
    {
        System.out.print("Stack is empty..\n");
        return " ";
    }
    str=str + "Stack elements are: \n ";
    for(i=top;i>=0;i--)
        str=str+" \n "+arr[i];

    return str;
}//end of toString
}//end of class

```

```

public class studentStackMain{

public static void main(String []str)
{
    int choice;
    Student data;
    int r, age;
    String name;
    studentStack st=new studentStack();
    Scanner sc=new Scanner(System.in);
    do {
        System.out.print("\n1. Push \n");
        System.out.print("2. Pop \n");
        System.out.print("3. Peek \n");
        System.out.print("4. Display \n");
        System.out.print("5. Exit \n");
        choice = sc.nextInt();
        switch(choice)
        {
            case 1:
                System.out.print("Enter rno : ");
                r=sc.nextInt();
                System.out.print("Enter age : ");
                age=sc.nextInt();
                System.out.print("Enter name : ");
                name=sc.next();
                data = new Student(r,name,age);
                st.push(data);
                break;
            case 2:
                data=st.pop();
                System.out.print("Popped data : " + data + "\n");
                break;
            case 3:
                data=st.peek();
                System.out.print("Data at top : " + data + "\n");
        }
    }
}

```

```

        break;
    case 4:
        System.out.print(st);
        break;
    case 5: break;
    default: System.out.print("Invalid choice..\n");
    }//end of switch
}while(choice!=5);
sc.close();
}//end of main
}//end of class

```

```

package for_DSA;           // simple queue of int

import java.util.Scanner;

class intQueue {
    private int arr[];
    private int front,rear,size;

    public intQueue()
    {
        front=rear=-1;
        size=10;
        arr=new int[size];
    }

    public intQueue(int s)
    {
        front=rear=-1;
        size=s;
        arr=new int[size];
    }

    public void insert(int d)
    {
        if(isFull())
        {
            System.out.print("Queue overflow..");
            return ;
        }

        if(front==-1)
            front=0;

        rear=rear+1;
        arr[rear]=d;
        return;
    }

    public int del()
    {
        int d=-999;
        if(isEmpty())
        {
            System.out.print("underflow... ");
            return d;
        }
    }
}

```

```

        }
        d=arr[front];
        front=front+1;
        return d;
    }

    public int peek()
    {
        int d=-999;
        if(isEmpty())
        {
            System.out.print("underflow...");
            return d;
        }
        d=arr[front];
        return d;
    }

    public boolean isEmpty()
    {
        if((front== -1) || (front == rear+1))
            return true;
        else
            return false;
    }

    public boolean isFull()
    {
        if(rear==size-1)
            return true;
        else
            return false;
    }

    public String toString()
    {
        int i;
        String str=new String();
        if(isEmpty())
        {
            System.out.print("Empty..");
            return " ";
        }
        str=str+"Queue :";
        for(i=front;i<=rear;i++)
            str=str+" "+arr[i];

        str=str+"\n";
        return str;
    }
}//end of class

public class intQueueMain {
    public static void main(String[] args) {
        int choice, data;
        intQueue st=new intQueue();
        Scanner sc=new Scanner(System.in);
        do {
            System.out.print("1. Insert \n");
            System.out.print("2. Remove \n");

```

```

        System.out.print("3. Peek \n");
        System.out.print("4. Display \n");
        System.out.print("5. Exit \n");
        choice = sc.nextInt();
        switch(choice)
        {
            case 1:
                System.out.print("Enter data to be pushed : ");
                data=sc.nextInt(); //20
                st.insert(data);
                break;
            case 2:
                data=st.del();
                System.out.print("Popped data : " + data + "\n");
                break;
            case 3:
                data=st.peek();
                System.out.print("Data at top : " + data + "\n");
                break;
            case 4:
                System.out.print(st);
                break;
            case 5: break;
            default: System.out.print("Invalid choice..\n");
        }//end of switch
    }while(choice!=5);
    sc.close();
}//end of main

}

```

```

package for_DSA;           //Circular queue of int

import java.util.Scanner;

class intCircularQueue{
    private int arr[];
    private int front,rear,size;

    public intCircularQueue()
    {
        front=rear=-1;
        size=10;
        arr=new int[size];
    }

    public intCircularQueue(int s)
    {
        front=rear=-1;
        size=s;
        arr=new int[size];
    }

    public boolean isEmpty()
    {
        if(front== -1)
            return true;

```

```

        else
            return false;
    }

public boolean isFull()
{
    if(((front==0)&&(rear==size-1))||(front==rear+1))
        return true;
    else
        return false;
}

public void insert(int d)
{
    System.out.println("\nfront = "+front+" rear = "+rear);
    if(isFull())
    {
        System.out.println("Overflow...");
        return;
    }

    if(front==-1)
        front=0;

    if(rear==size-1)
        rear=0;
    else
        rear=rear+1;

    arr[rear]=d;

    return;
}

public int del()
{
    int d=-999;
    if(isEmpty())
    {
        System.out.println("Underflow...");
        return d;
    }

    d=arr[front];

    if(front==rear) // last element
    {
        front=rear=-1;
    }
    else if(front==size-1)
        front=0;
    else
        front=front+1;

    return d;
}//end of del

public int peek()
{
    if(!isEmpty())

```

```

        return arr[front];
    else
        return -999;
}

public String toString()
{
    String str=new String();
    int i;
    if(isEmpty())
    {
        return "Empty...";
    }

    i=front;
    if(front<=rear)
    {
        while(i<=rear)
        {
            str=str+" "+arr[i];
            i=i+1;
        }
    }
    else
    {
        while(i<=size-1)
        {
            str = str +" "+arr[i];
            i=i+1;
        }
        i=0;
        while(i<=rear)
        {
            str = str +" "+arr[i];
            i=i+1;
        }
    }//end of else
    str=str+"\n";
    return str;
}
}//end of class

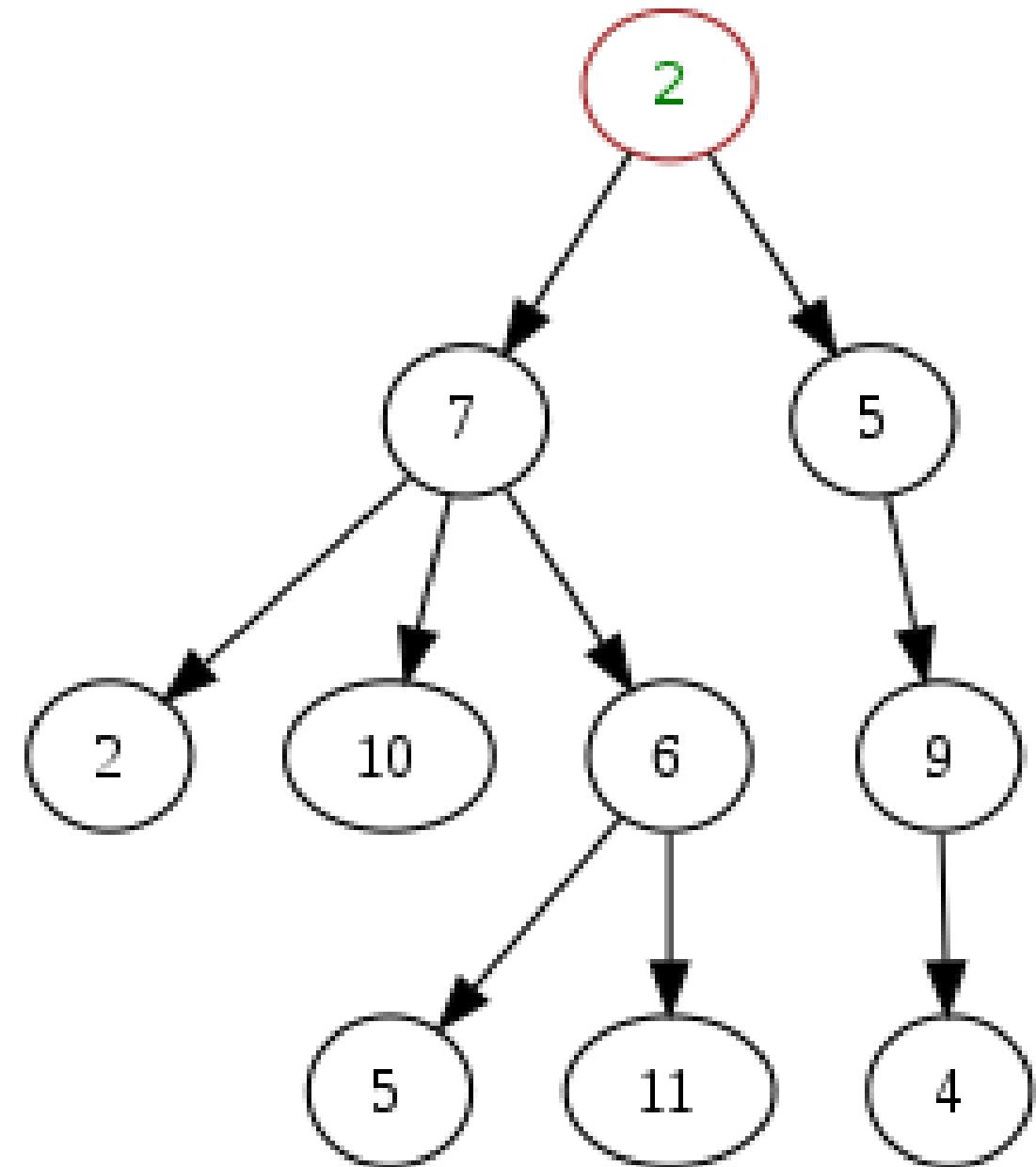
public class CircularQueueMain {
    public static void main(String[] args) {
        int choice, data;
        intCircularQueue st=new intCircularQueue();
        Scanner sc=new Scanner(System.in);
        do {
            System.out.print("1. Insert \n");
            System.out.print("2. Remove \n");
            System.out.print("3. Peek \n");
            System.out.print("4. Display \n");
            System.out.print("5. Exit \n");
            choice = sc.nextInt();
            switch(choice)
            {
                case 1:
                    System.out.print("Enter data to be pushed : ");
                    data=sc.nextInt(); //20

```

```
        st.insert(data);
        break;
    case 2:
        data=st.del();
        System.out.print("Popped data : " + data + "\n");
        break;
    case 3:
        data=st.peek();
        System.out.print("Data at top : " + data + "\n");
        break;
    case 4:
        System.out.print(st);
        break;
    case 5: break;
    default: System.out.print("Invalid choice..\n");
} //end of switch
}while(choice!=5);
sc.close();
}//end of main
}
```

Trees

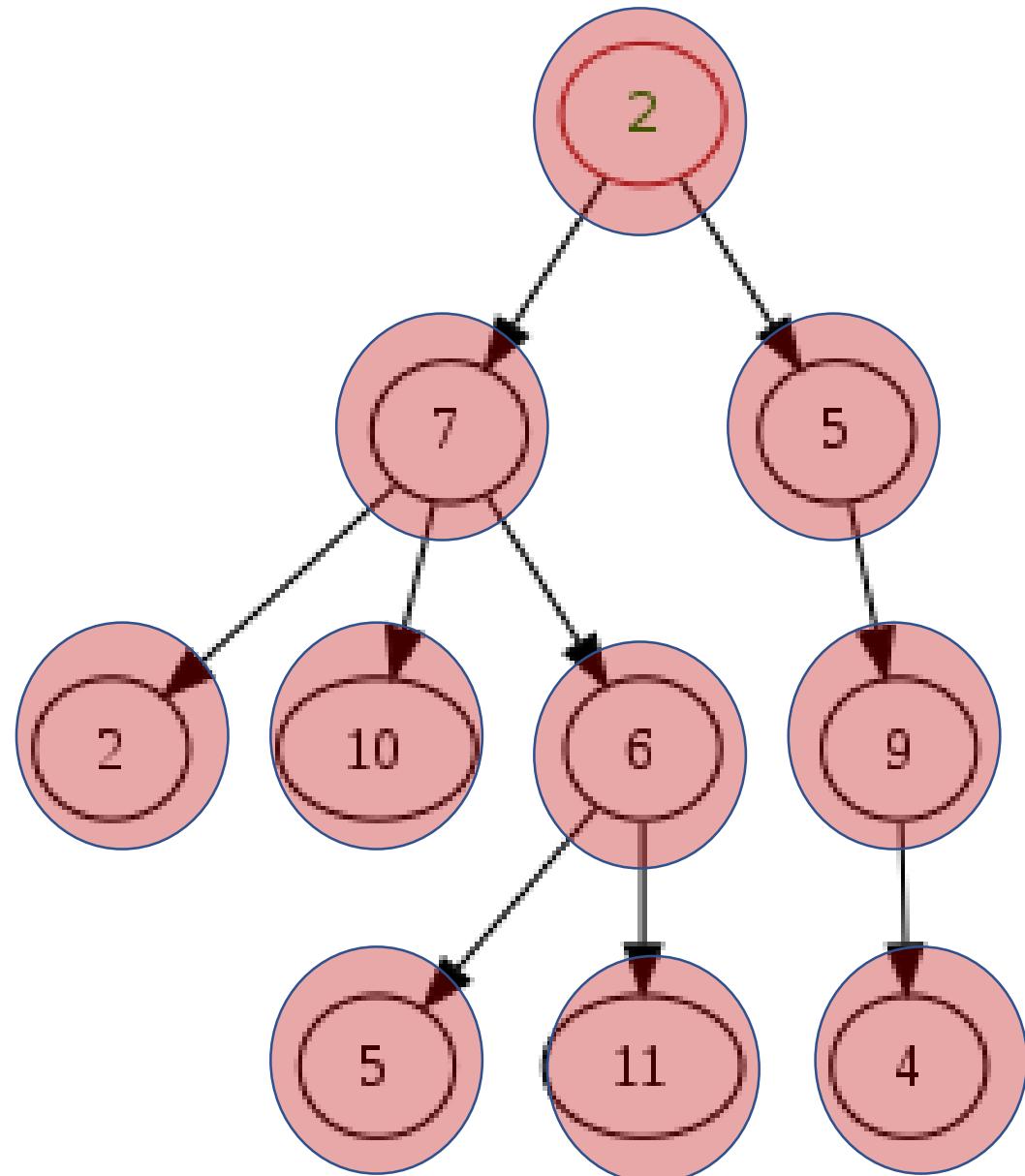
Introduction to trees



Terminology

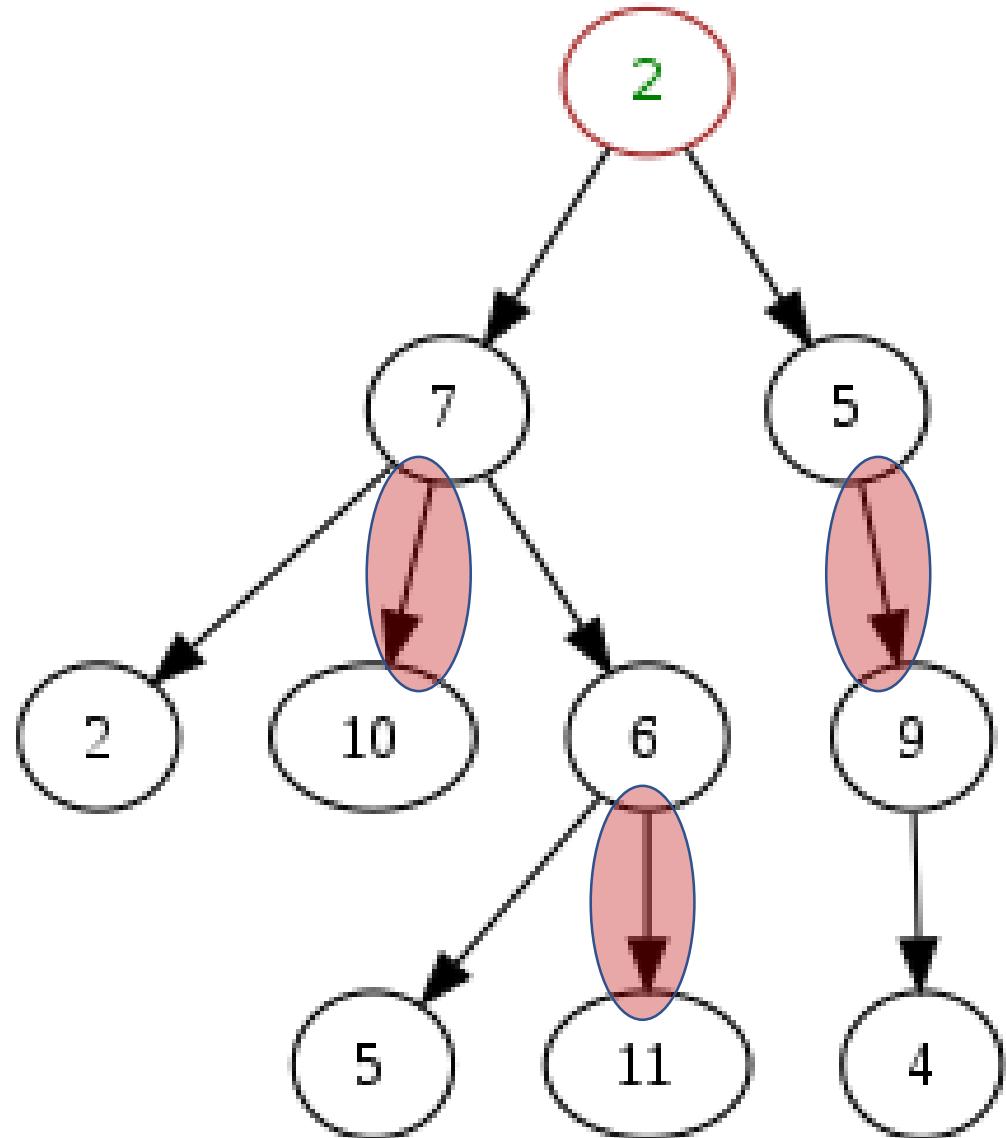
- ***Node***

- Edge
- Root
- Path
- Children
- Parent
- Sibling
- Subtree
- Leaf Node



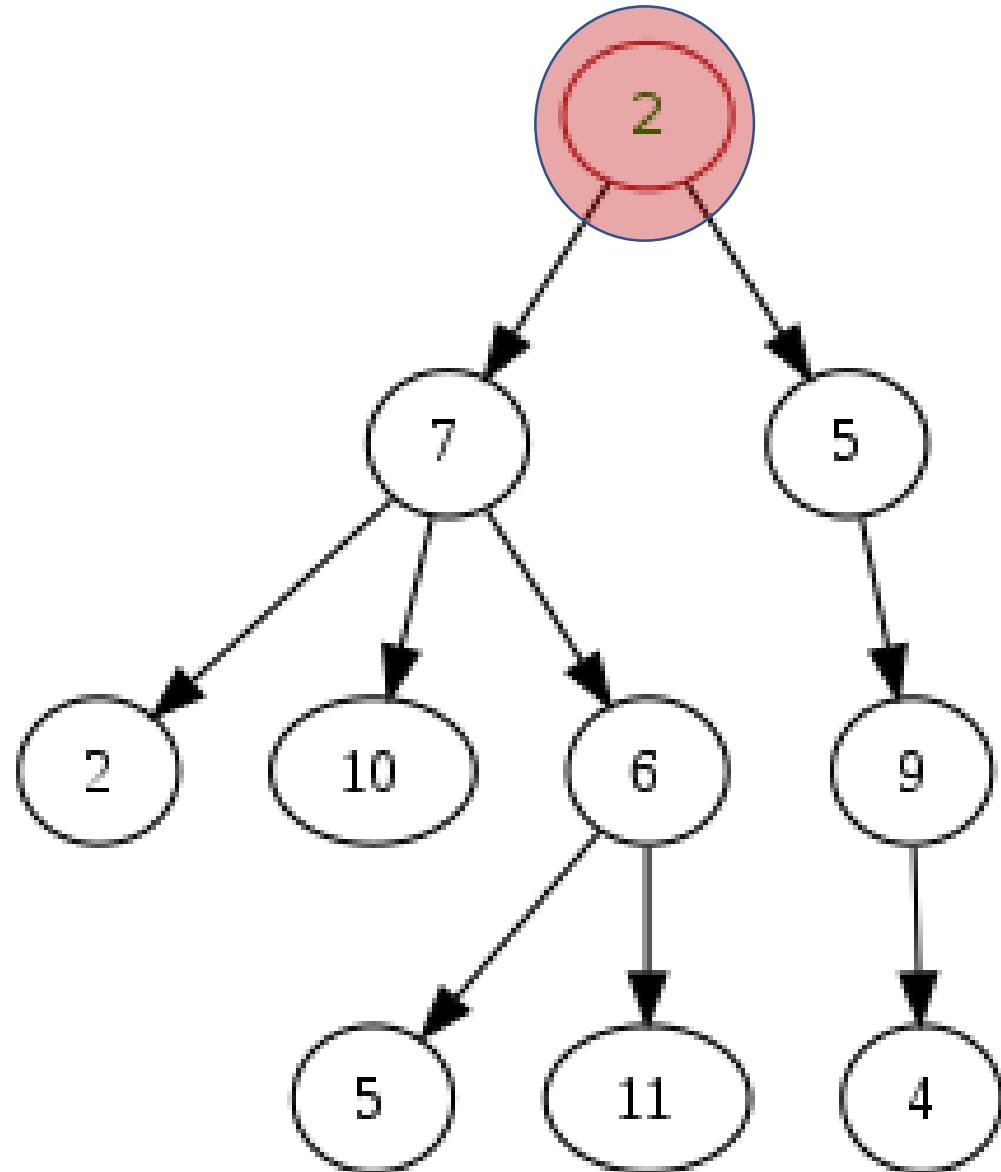
Terminology

- Node
- ***Edge***
- Root
- Path
- Children
- Parent
- Sibling
- Subtree
- Leaf Node



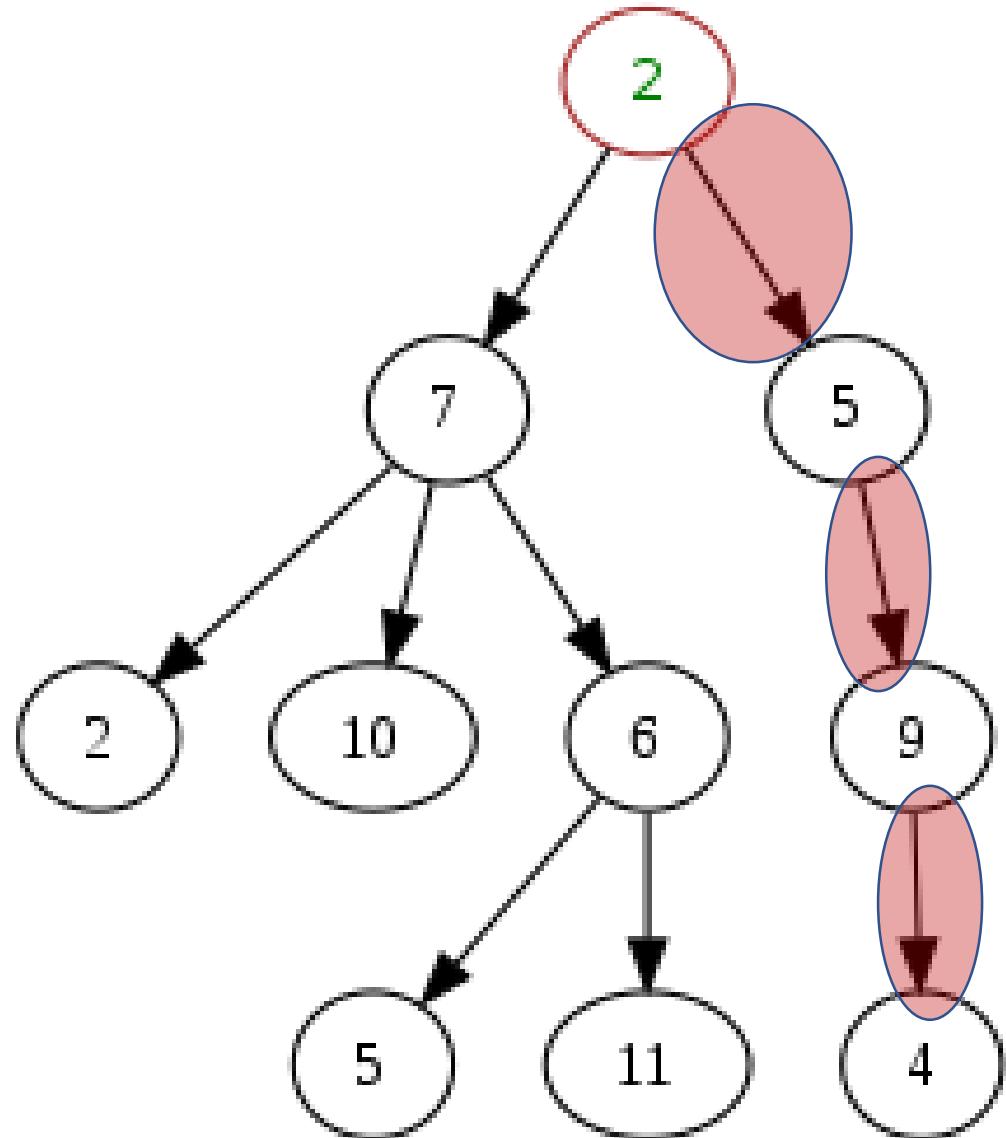
Terminology

- Node
- Edge
- ***Root***
- Path
- Children
- Parent
- Sibling
- Subtree
- Leaf Node



Terminology

- Node
 - Edge
 - Root
- ***Path***
- Children
 - Parent
 - Sibling
 - Subtree
 - Leaf Node

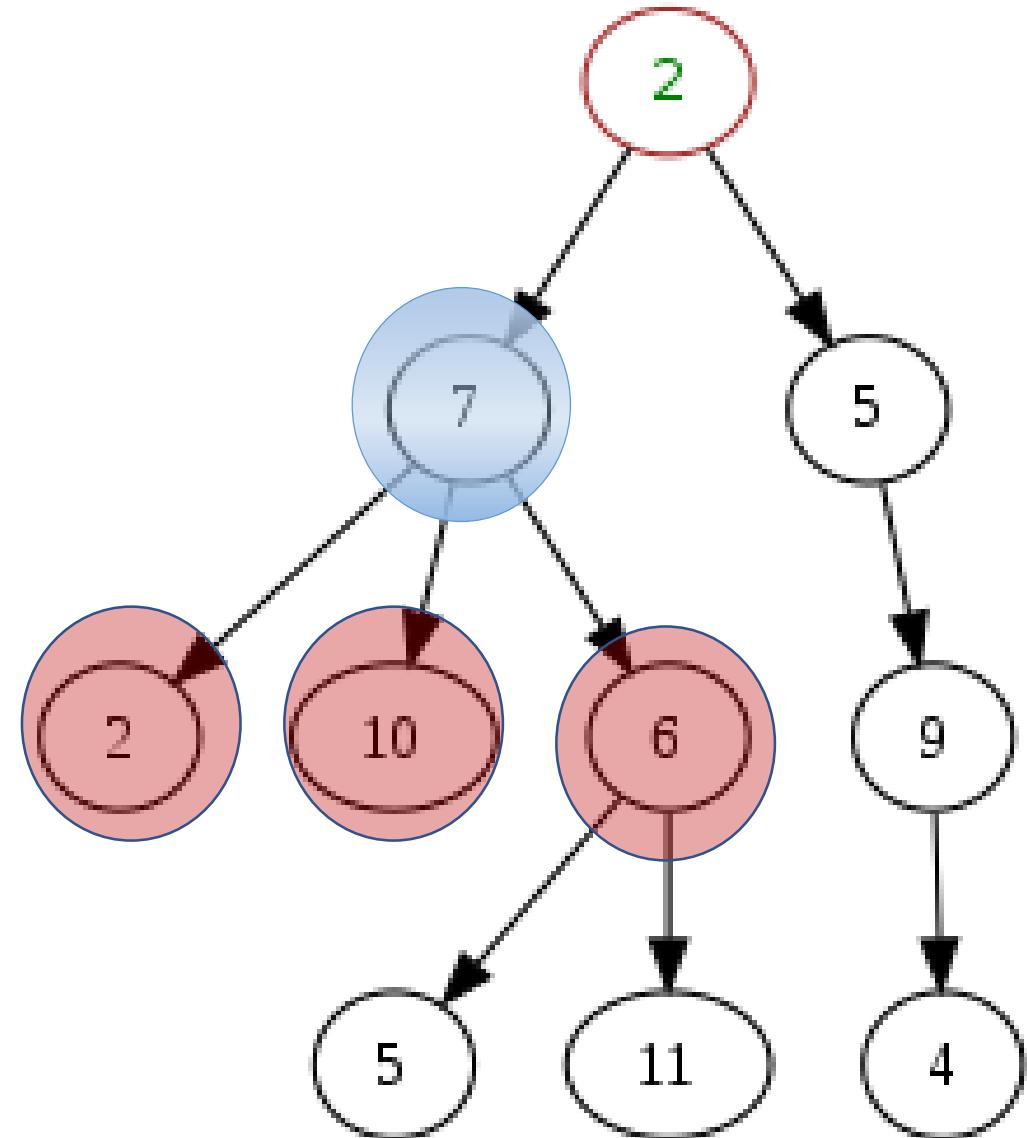


Terminology

- Node
- Edge
- Root
- Path

• *Children*

- Parent
- Sibling
- Subtree
- Leaf Node

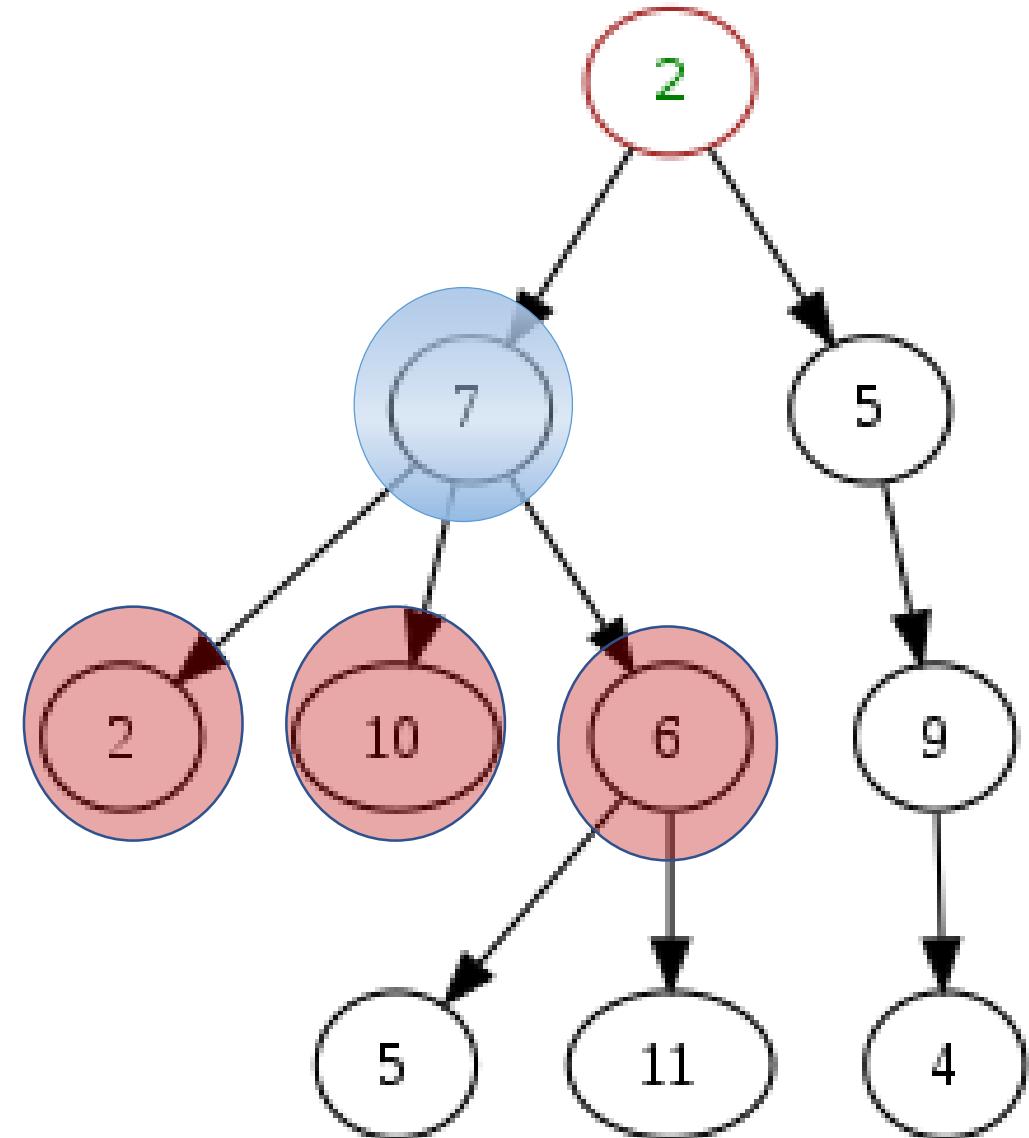


Terminology

- Node
- Edge
- Root
- Path
- Children

• *Parent*

- Sibling
- Subtree
- Leaf Node

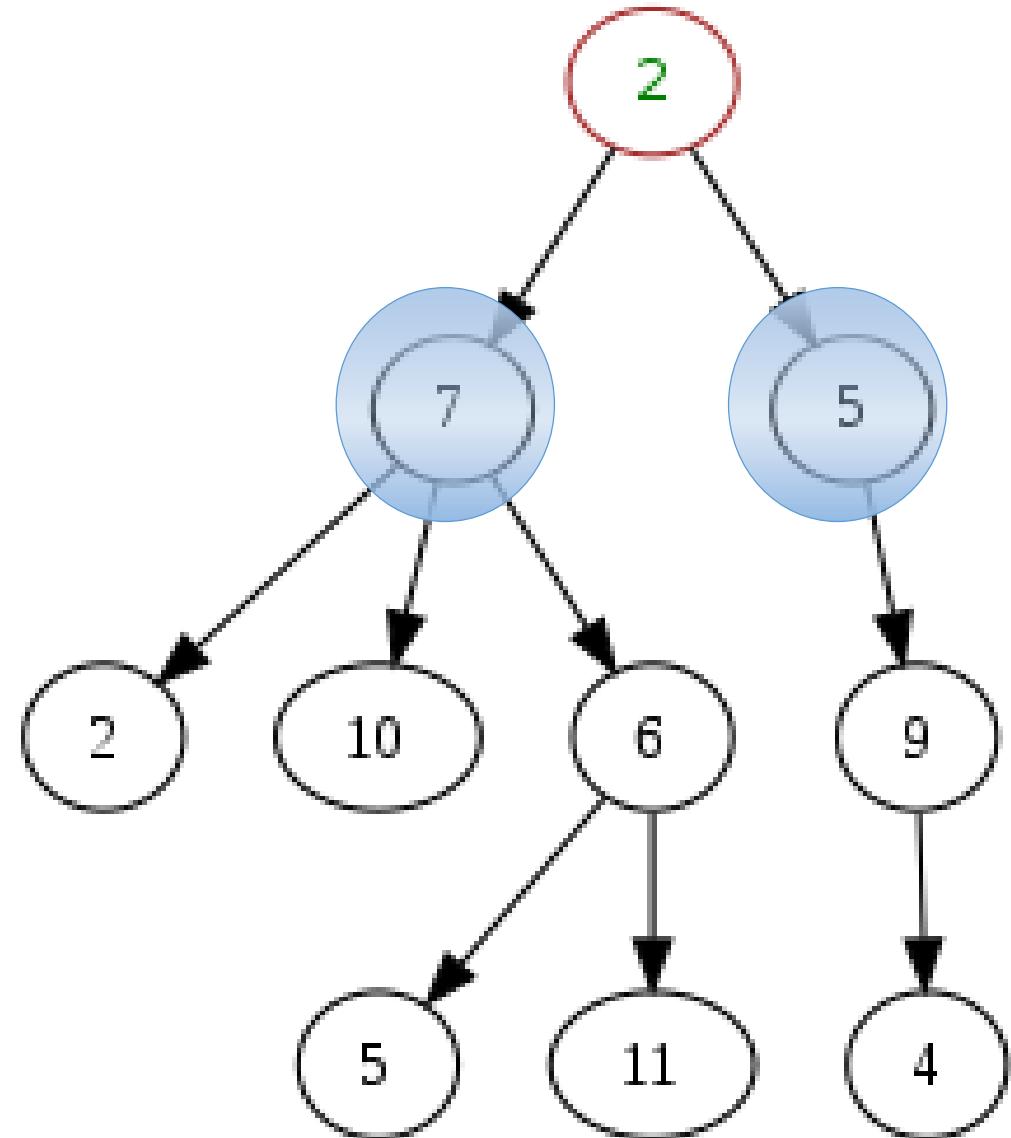


Terminology

- Node
- Edge
- Root
- Path
- Children
- Parent

• *Sibling*

- Subtree
- Leaf Node

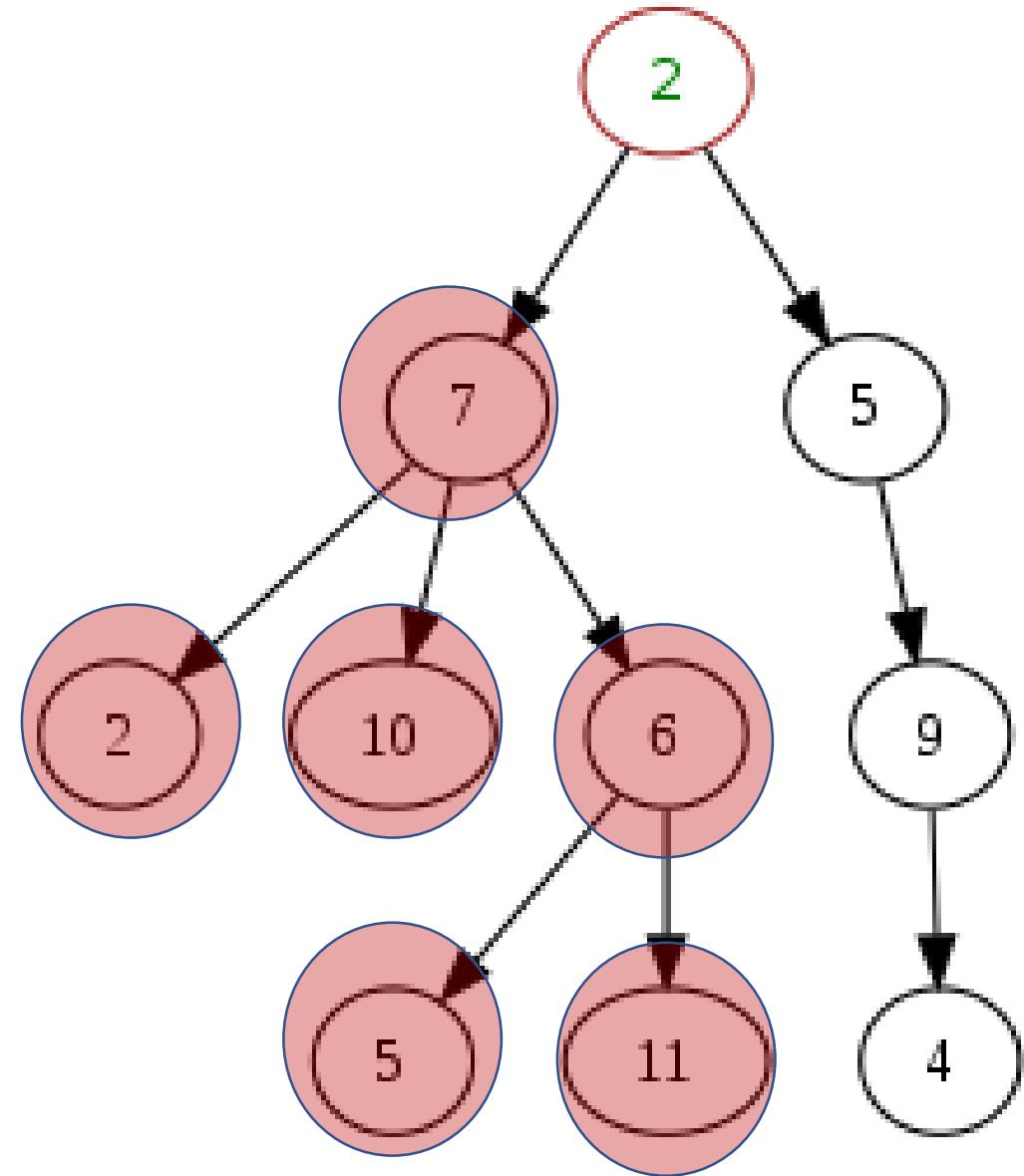


Terminology

- Node
- Edge
- Root
- Path
- Children
- Parent
- Sibling

Subtree

- Leaf Node

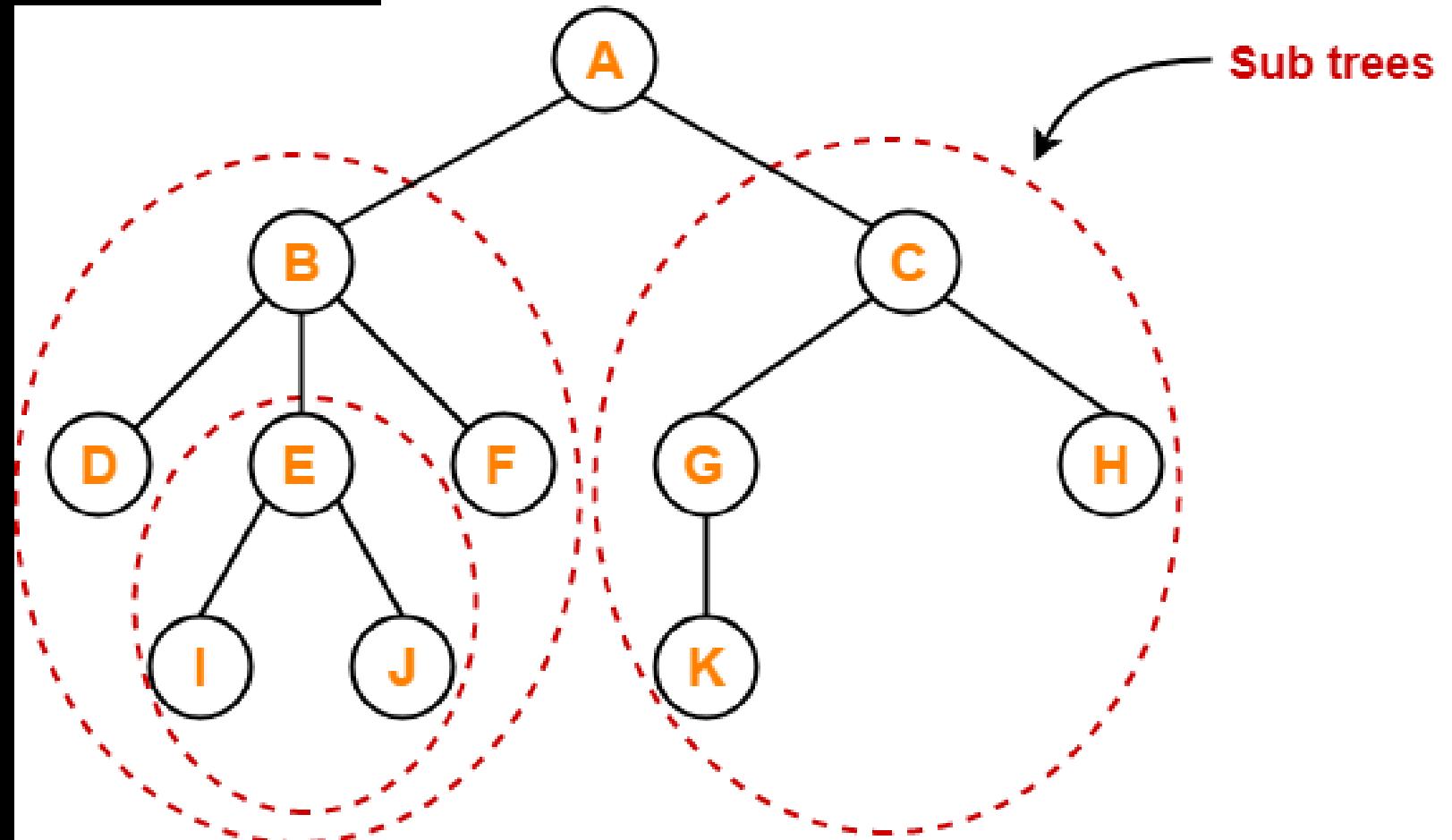


Terminology

- Node
- Edge
- Root
- Path
- Children
- Parent
- Sibling

• Subtree

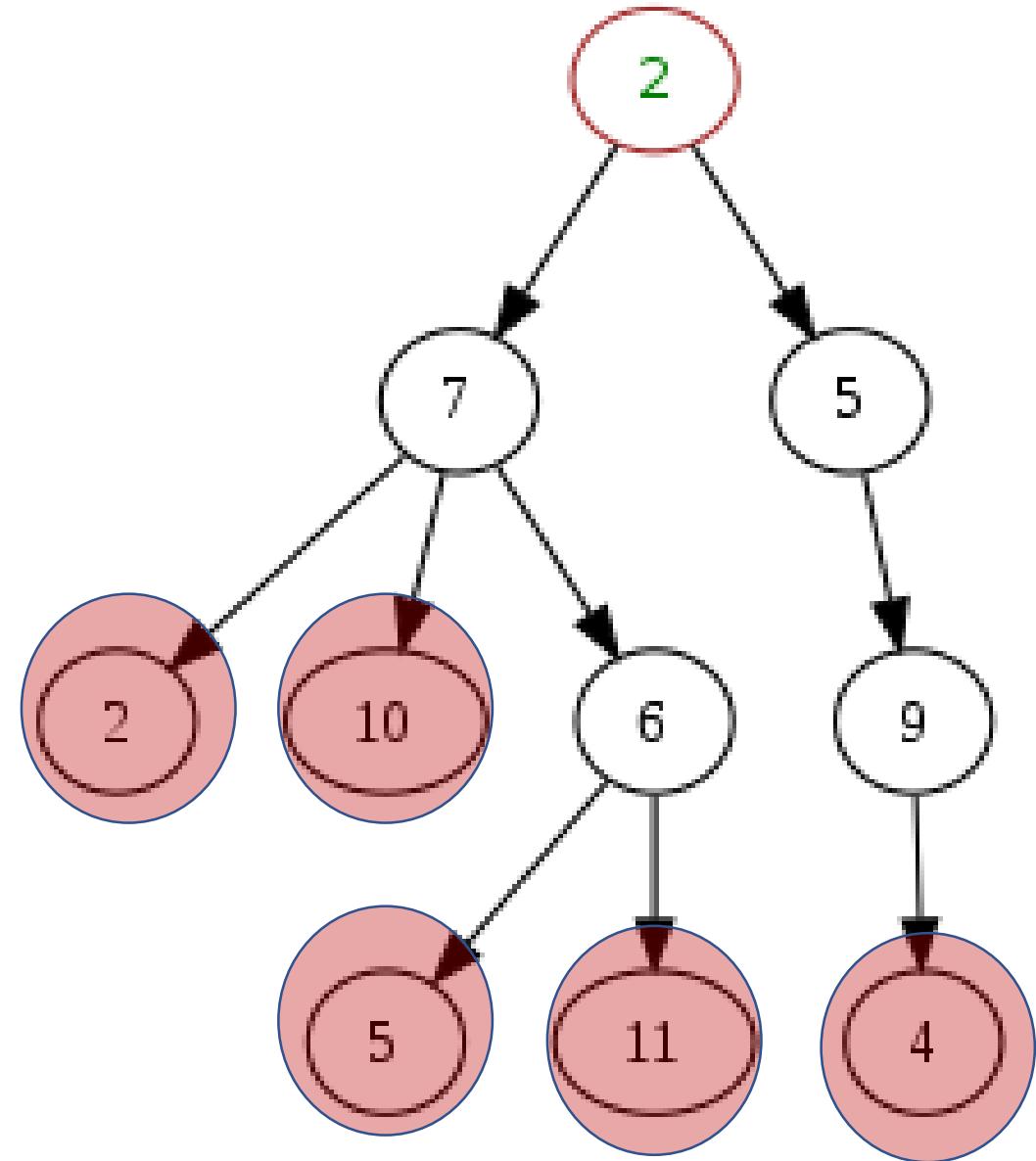
- Leaf Node



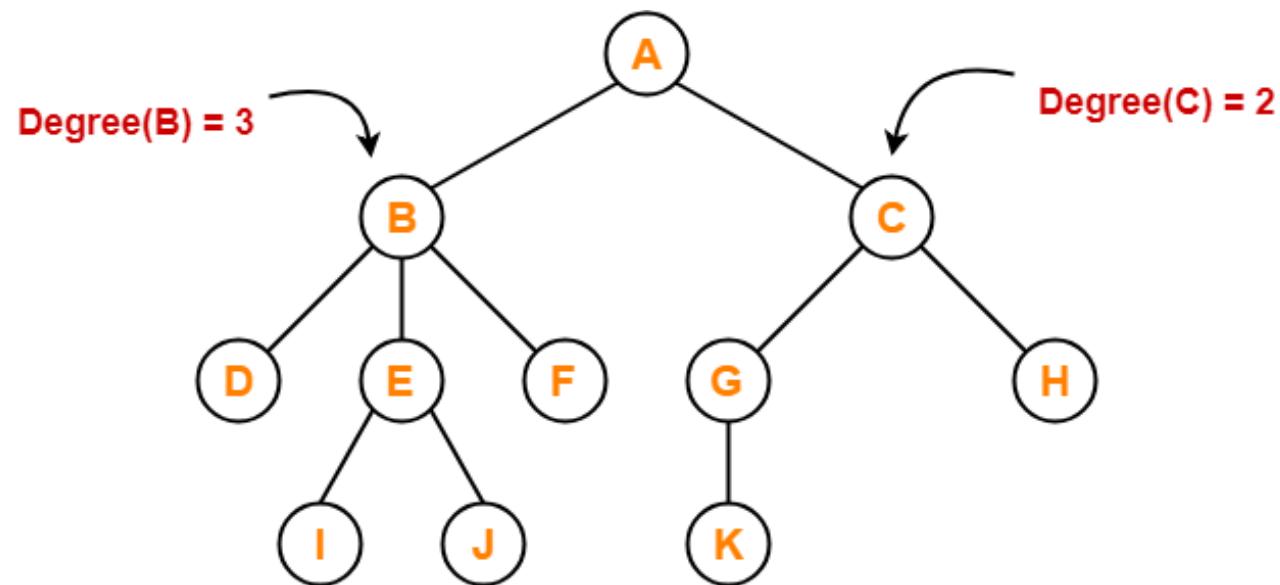
Terminology

- Node
- Edge
- Root
- Path
- Children
- Parent
- Sibling
- Subtree

• LeafNode

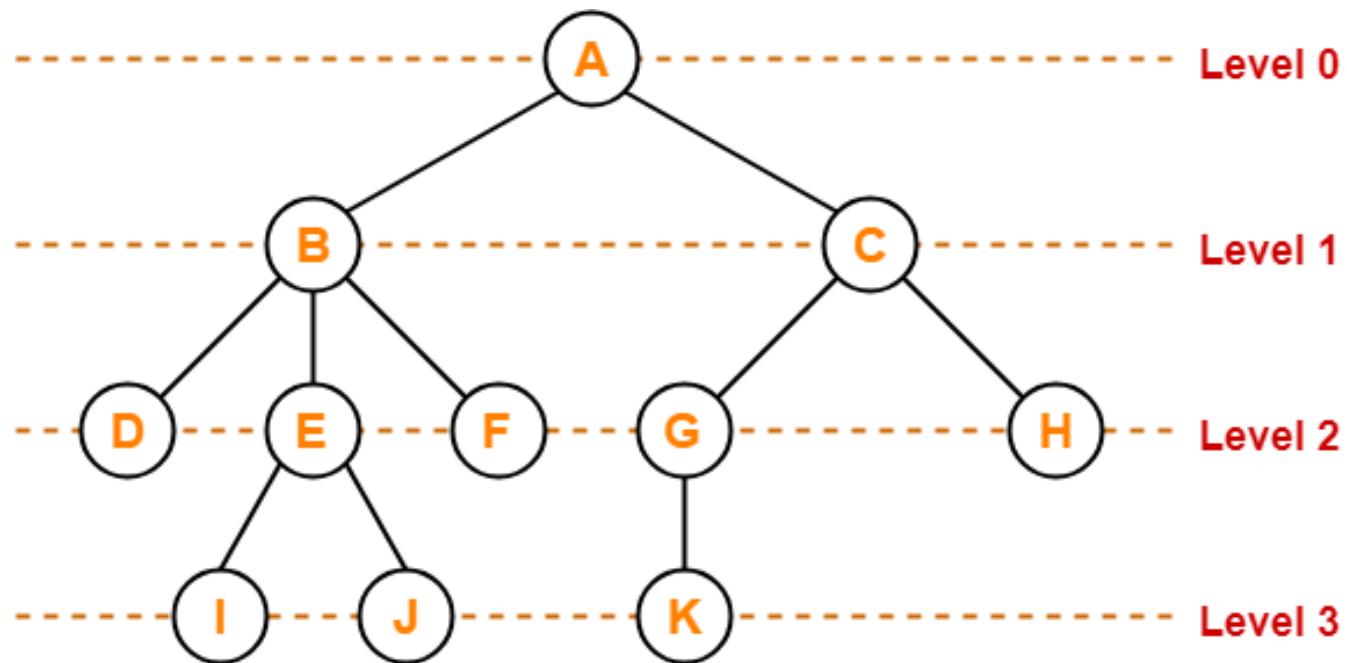


Degree



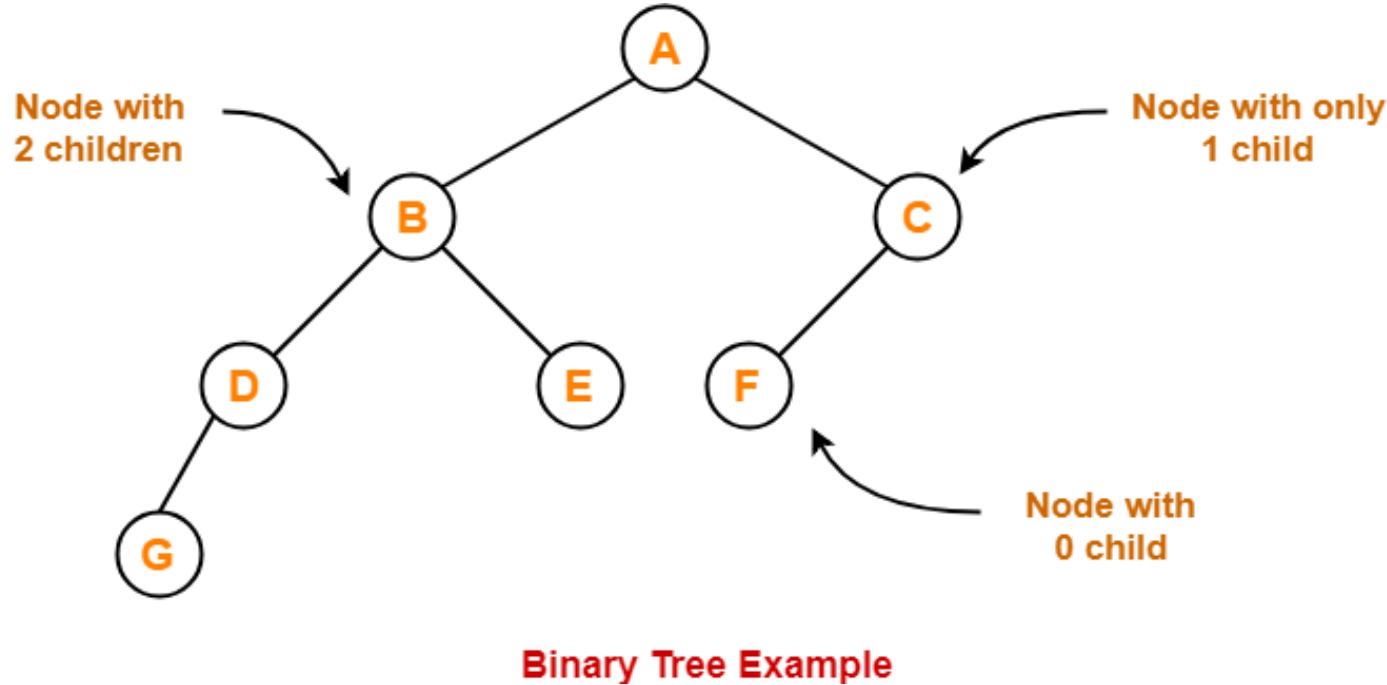
- **Degree of a node** is the total number of children of that node.
- **Degree of a tree** is the highest degree of a node among all the nodes in the tree.

Level



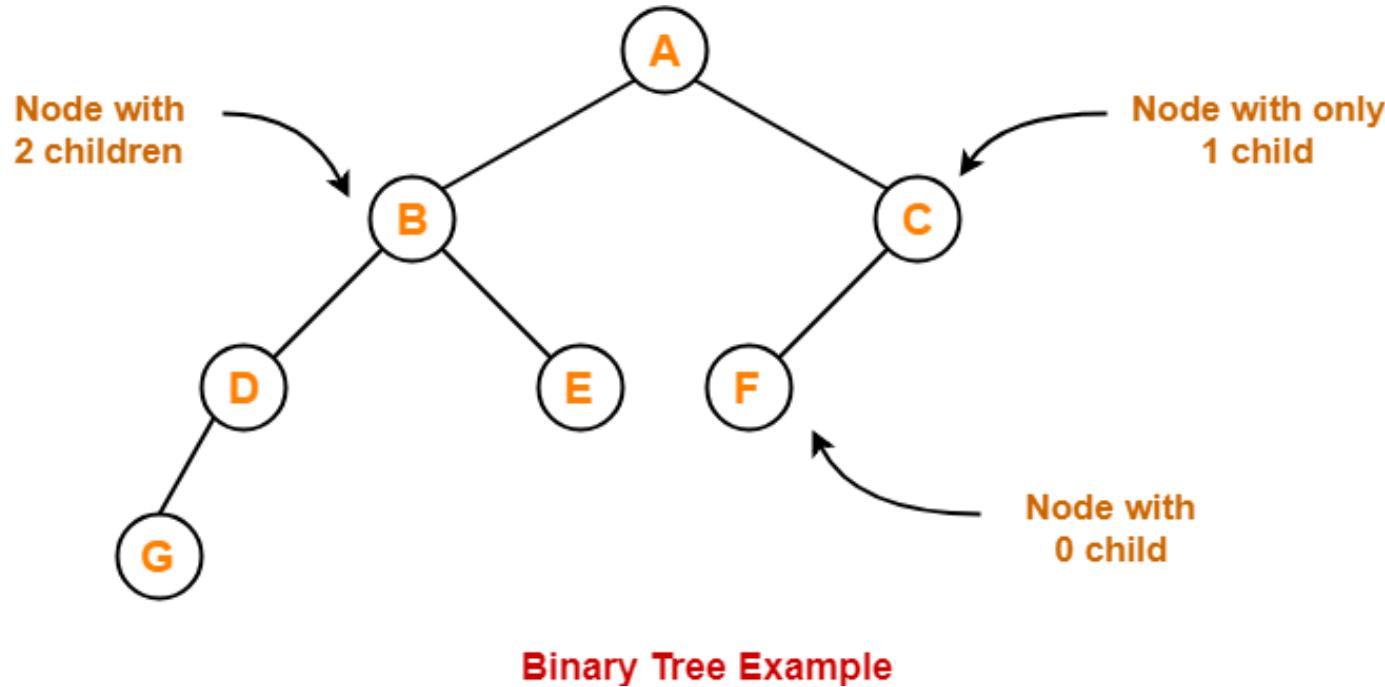
- In a tree, each step from top to bottom is called as **level of a tree**.
- The level count starts with 0 and increments by 1 at each level or step.

Binary Tree



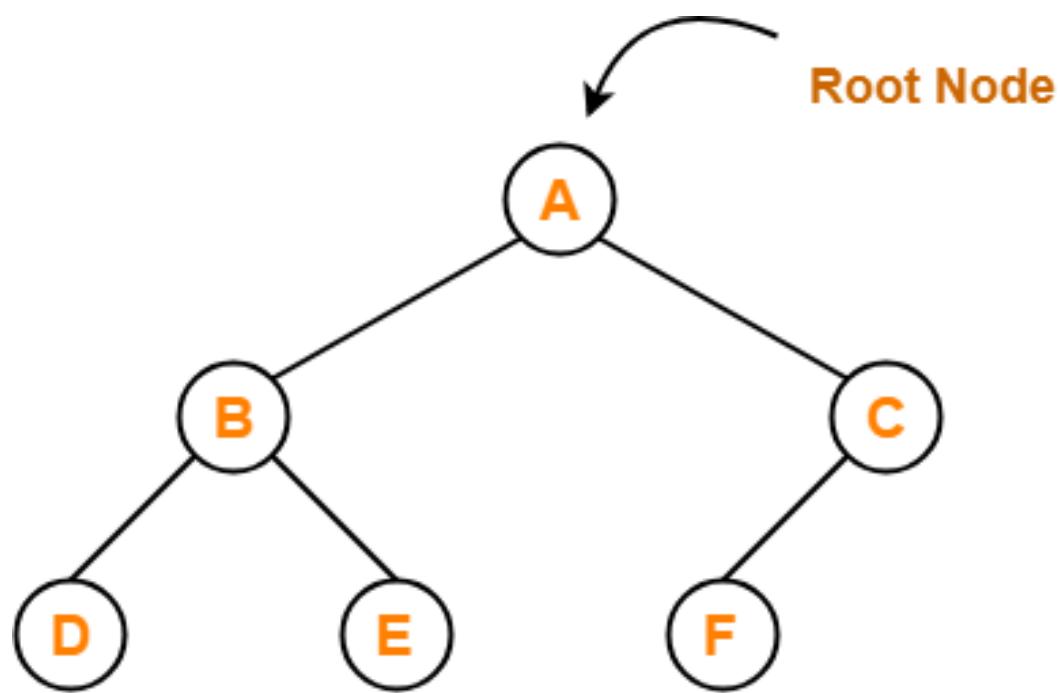
- Binary tree is a special tree data structure in which each node can have at most 2 children.
- Thus, in a binary tree, each node has either 0 child or 1 child or 2 children.

Types of Binary Trees



- Rooted Binary Tree
- Full / Strictly Binary Tree
- Complete / Perfect Binary Tree
- Almost Complete Binary Tree
- Skewed Binary Tree

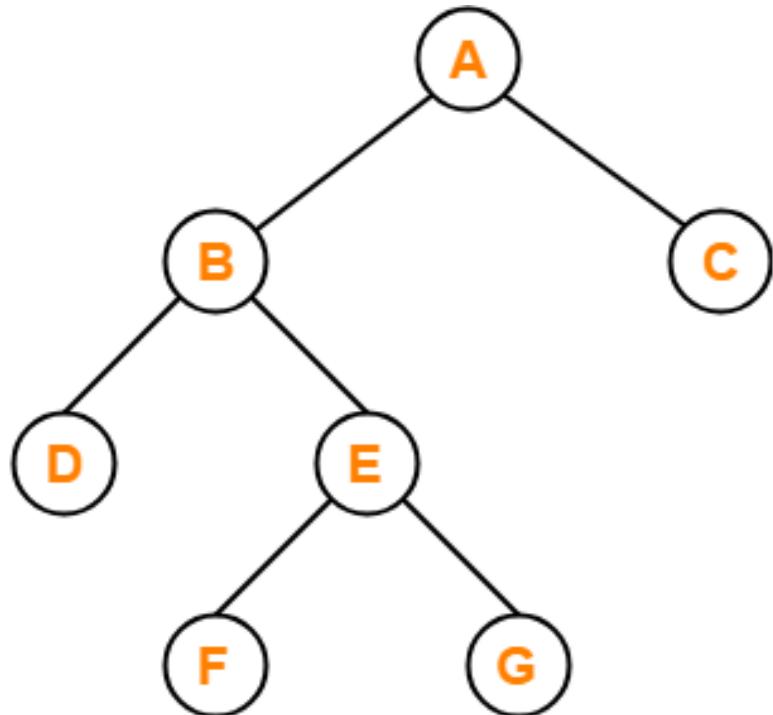
Types of Binary Trees



Rooted Binary Tree

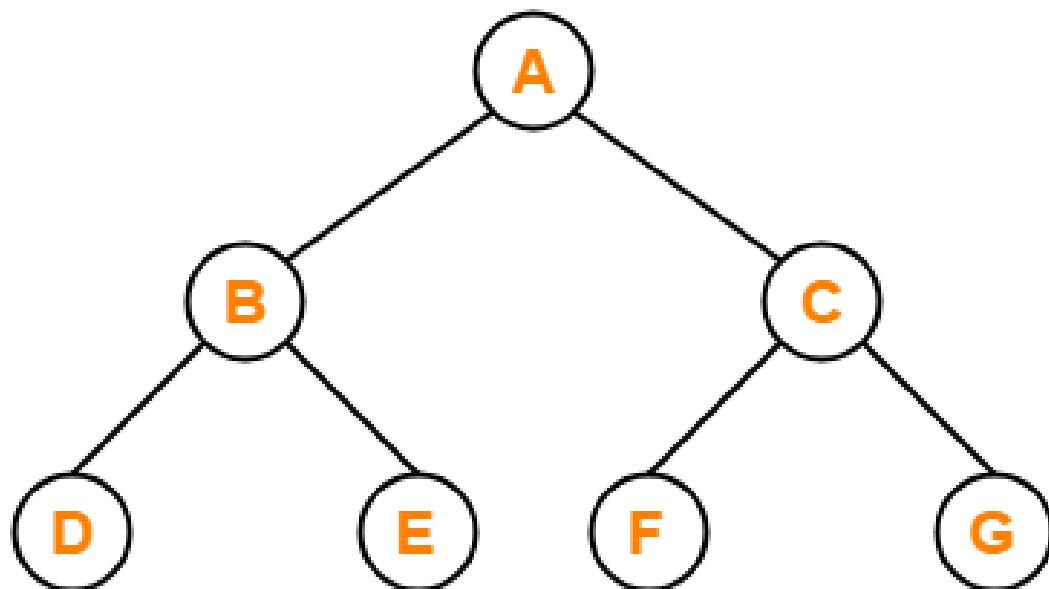
- Rooted Binary Tree
 - *It has a root node.*
 - *Each node has at most 2 children.*

Types of Binary Trees



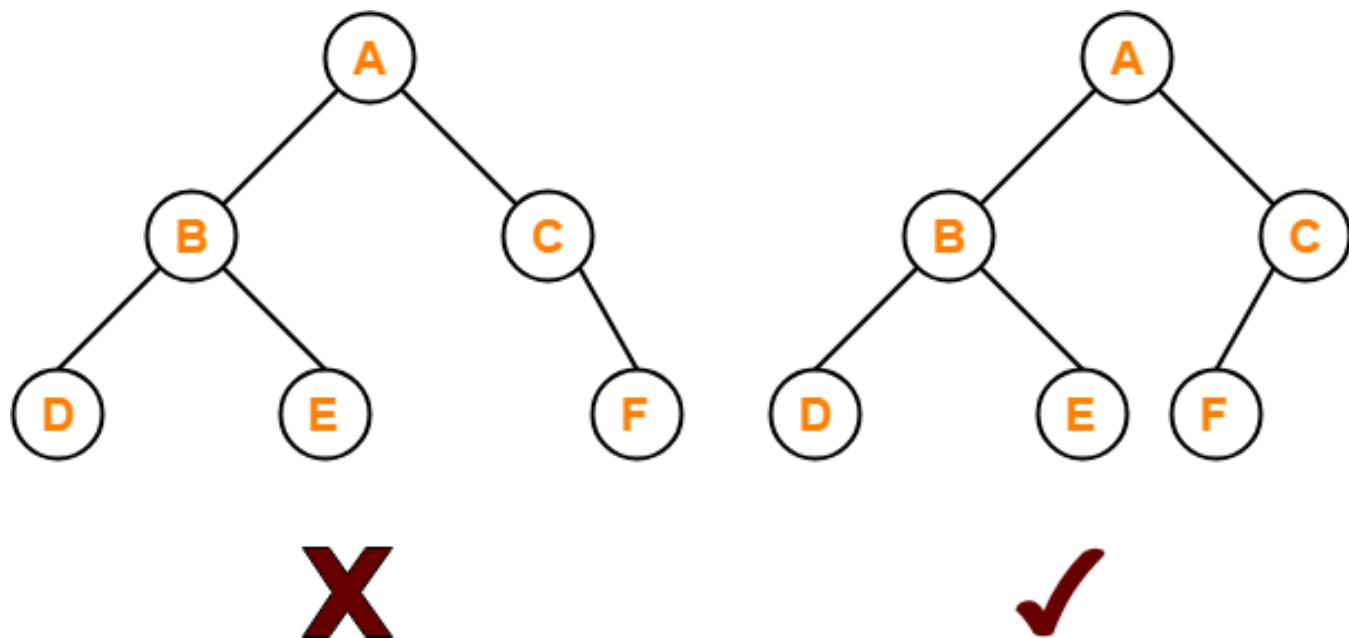
- **Full / Strictly Binary Tree**
 - A binary tree in which every node has either 0 or 2 children is called as a **Full binary tree**.

Types of Binary Trees



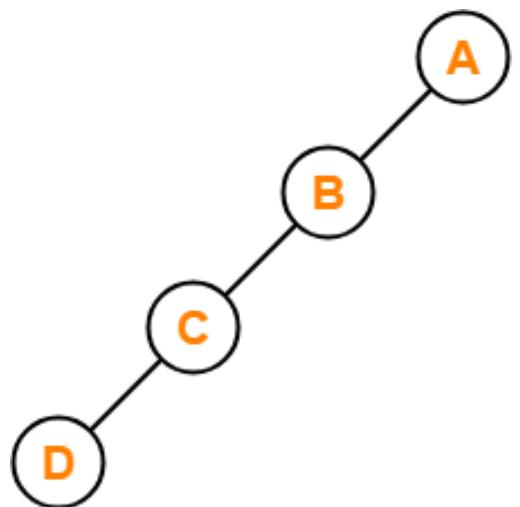
- **Complete / Perfect Binary Tree**
 - Every internal node has exactly 2 children.
 - All the leaf nodes are at the same level.

Types of Binary Trees

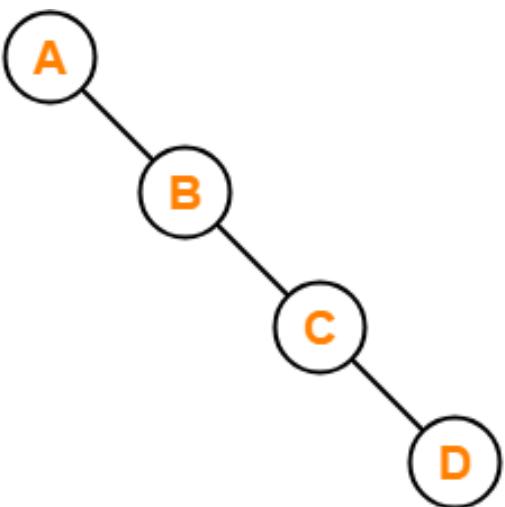


- **Almost Complete Binary Tree**
 - All the levels are completely filled except possibly the last level.
 - The last level must be strictly filled from left to right.

Types of Binary Trees



Left Skewed Binary Tree



Right Skewed Binary Tree

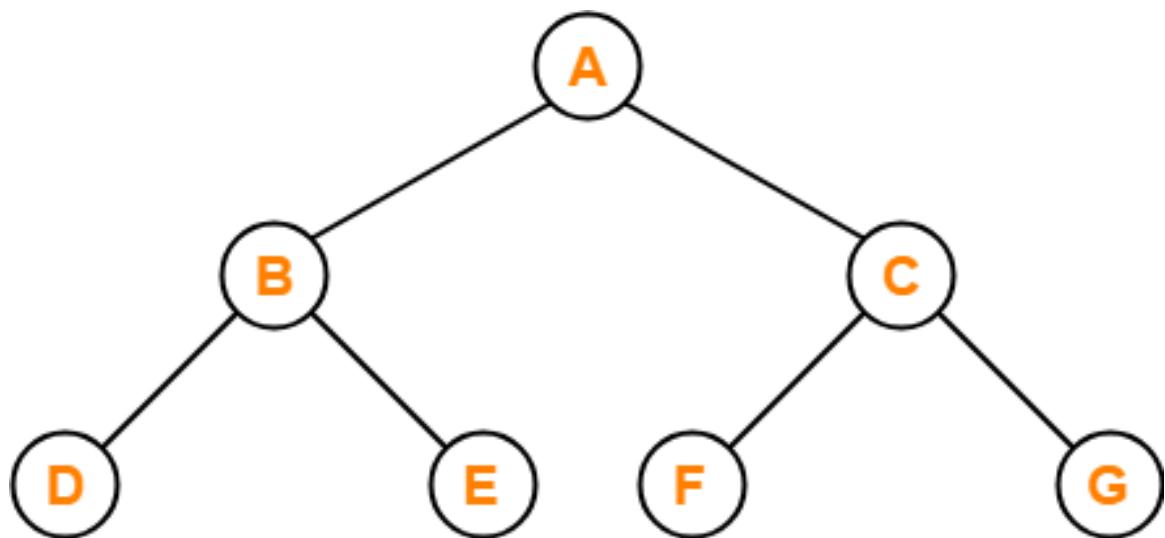
- **Skewed Binary Tree**
 - All the nodes except one node has one and only one child.
 - The remaining node has no child.

Tree Traversal

Tree Traversal Techniques

- Depth First Traversal
 - Preorder Traversal
 - Inorder Traversal
 - Postorder Traversal
- Breadth First Traversal

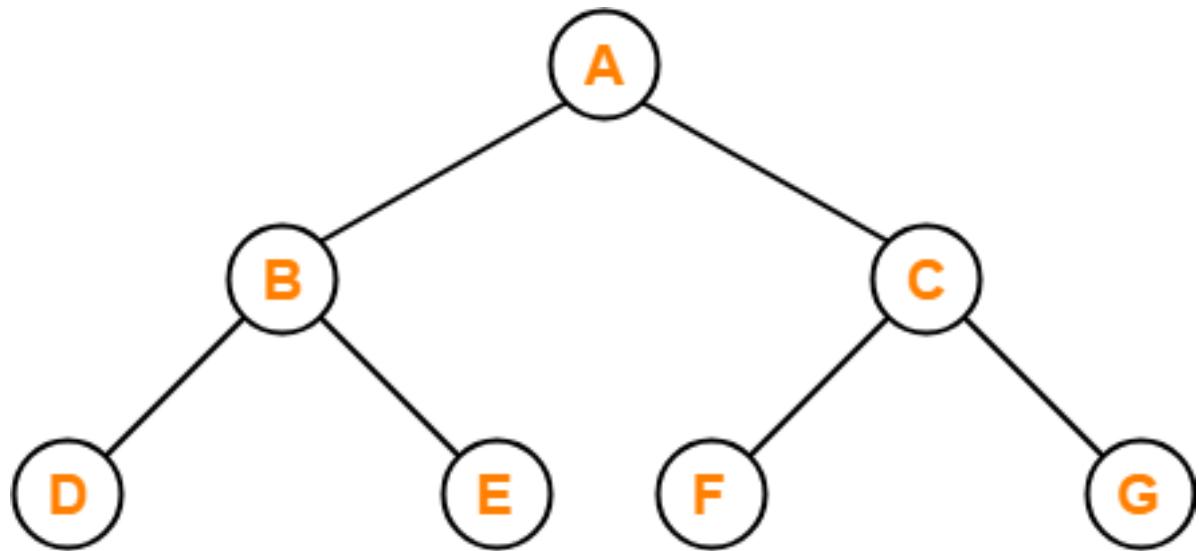
Preorder Traversal



- Visit the root
- Traverse the left sub tree
- Traverse the right sub tree

Preorder Traversal : A , B , D , E , C , F , G

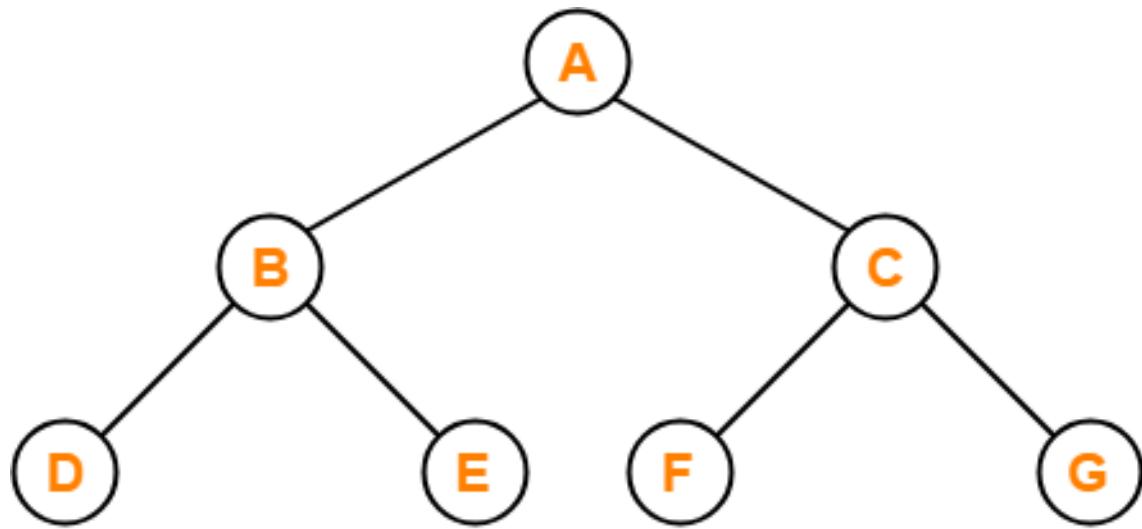
Inorder Traversal



- Traverse the left sub tree
- Visit the root
- Traverse the right sub tree

Inorder Traversal : D , B , E , A , F , C , G

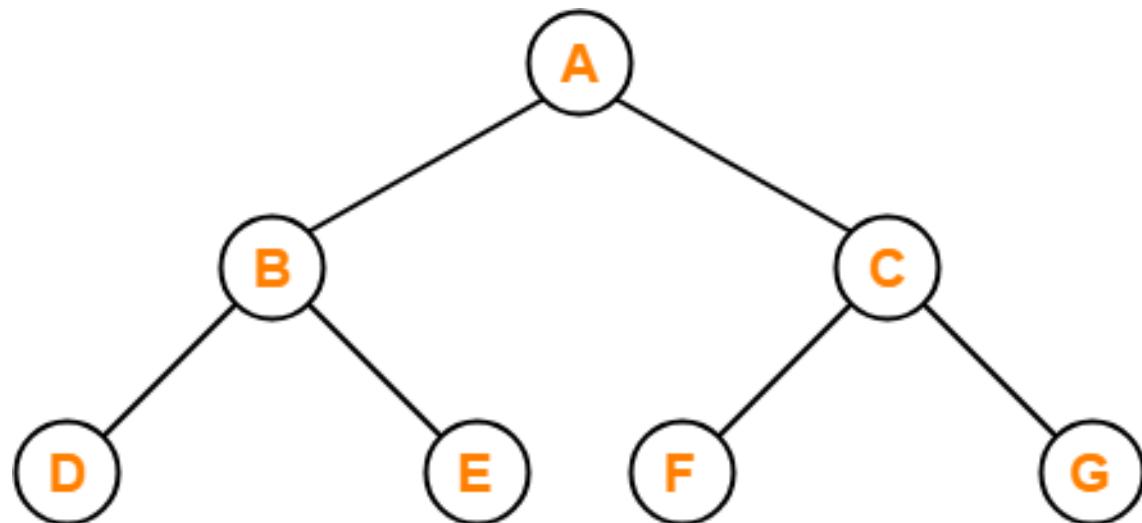
Postorder Traversal



Postorder Traversal : D , E , B , F , G , C , A

- Traverse the left sub tree
- Traverse the right sub tree
- Visit the root

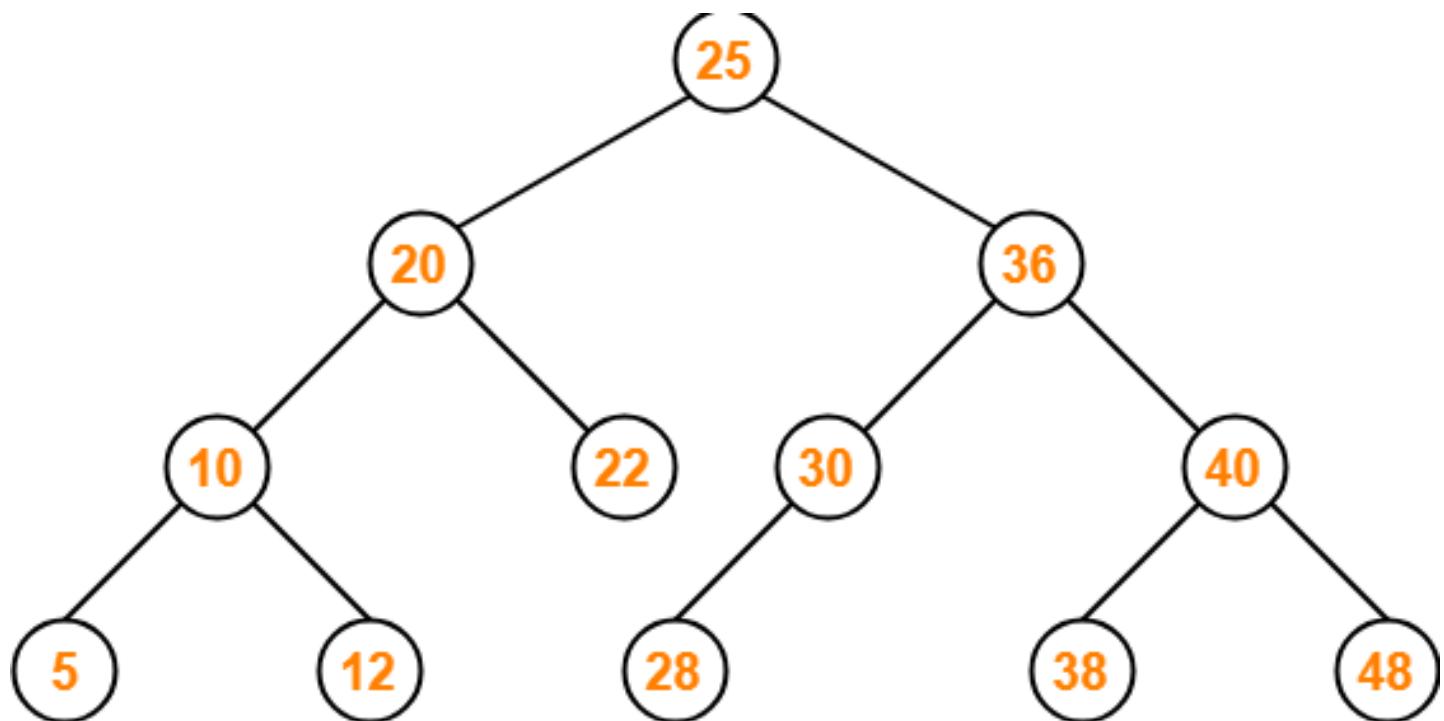
Breadth First Traversal



Level Order Traversal : A , B , C , D , E , F , G

- Breadth First Traversal of a tree prints all the nodes of a tree level by level.
- Breadth First Traversal is also called as **Level Order Traversal**

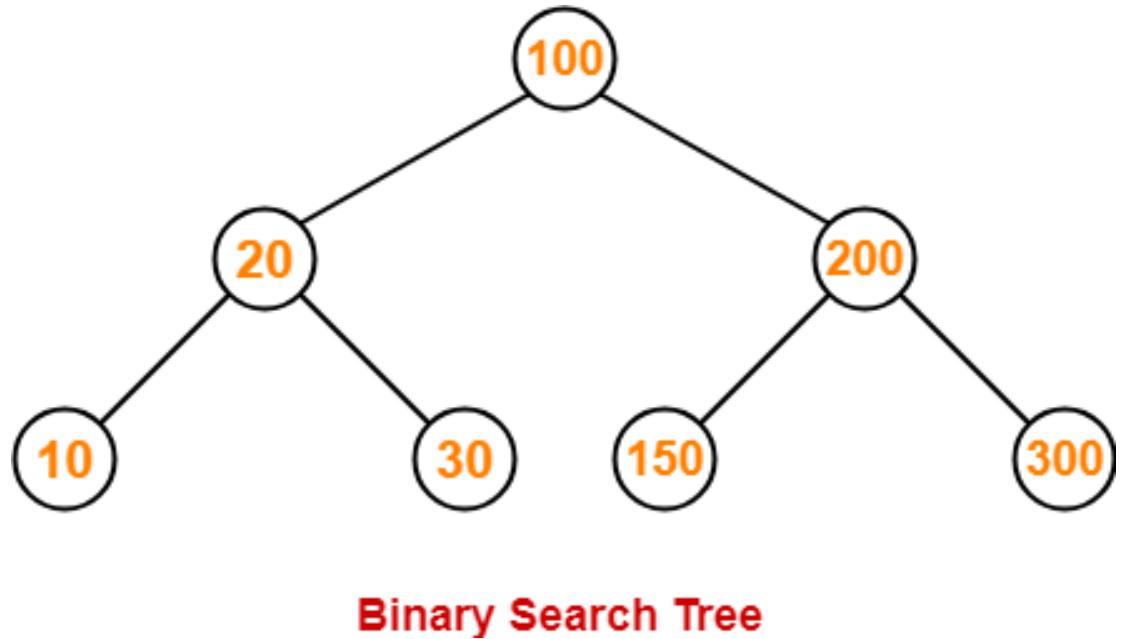
Binary Search Tree



- Smaller values in its left sub tree
- Larger values in its right sub tree

Binary Search Tree

BST Traversal



- **Preorder Traversal-**

100 , 20 , 10 , 30 , 200 , 150 , 300

- **Inorder Traversal-**

10 , 20 , 30 , 100 , 150 , 200 , 300

- **Postorder Traversal-**

10 , 30 , 20 , 150 , 300 , 200 , 100

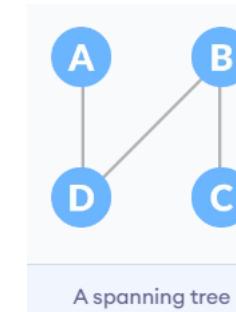
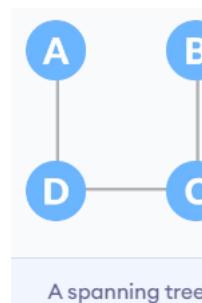
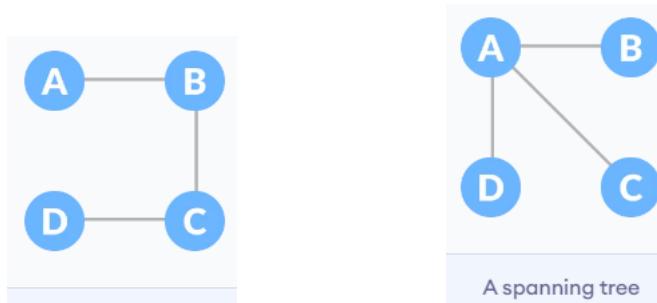
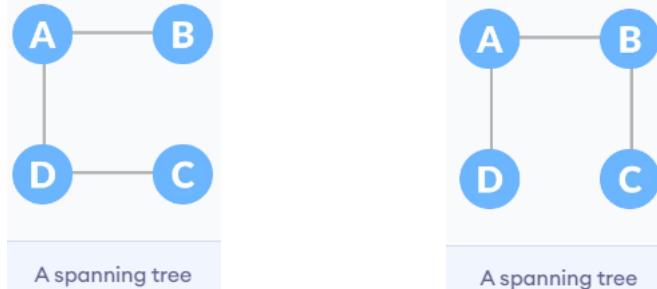
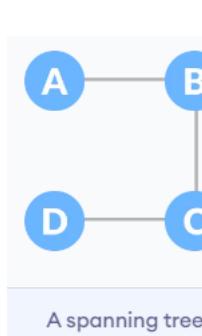
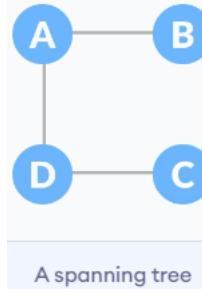
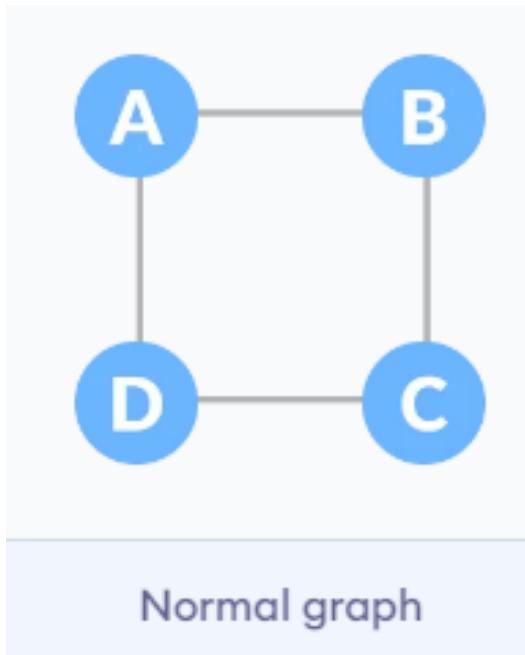
Spanning Tree

Spanning tree

A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges.

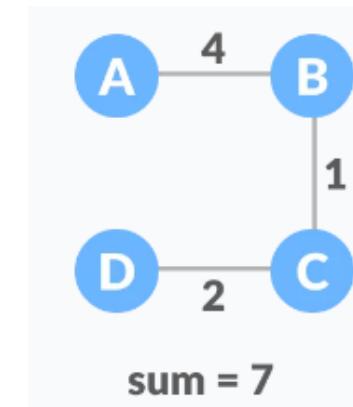
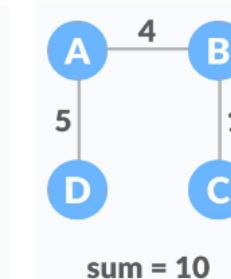
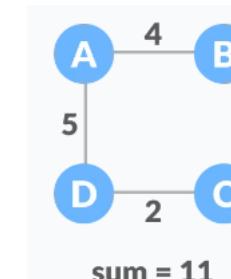
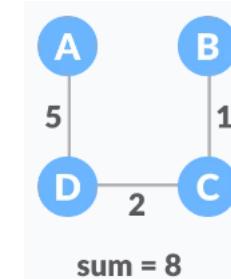
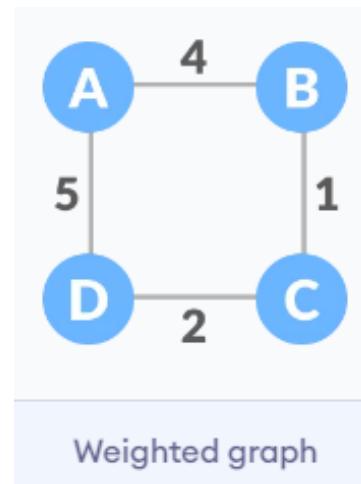
The edges may or may not have weights assigned to them.

Example of a Spanning Tree



Minimum Spanning Tree

- A minimum spanning tree is a spanning tree in which the sum of the weight of the edges is as minimum as possible.

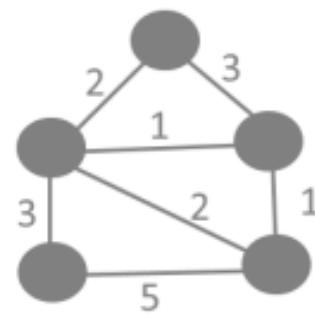


Minimum Spanning Tree

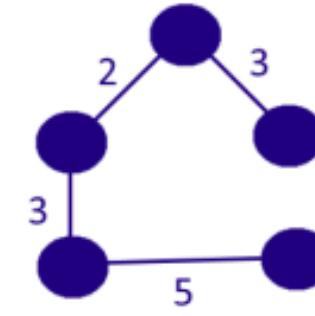
- The minimum spanning tree from a graph is found using the following algorithms:
 - Prim's Algorithm
 - Kruskal's Algorithm

Kruskal's algorithm

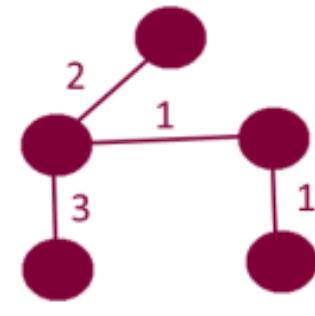
- Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which
 - form a tree that includes every vertex
 - has the minimum sum of weights among all the trees that can be formed from the graph



Graph



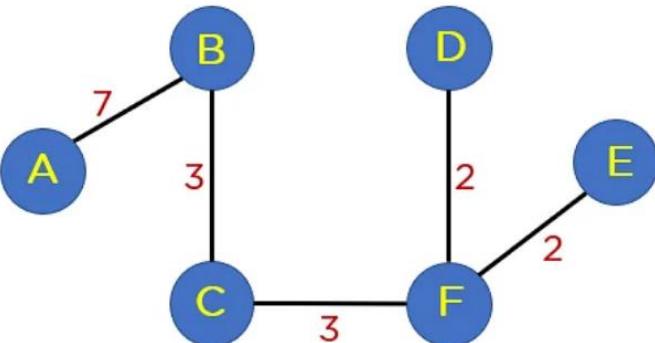
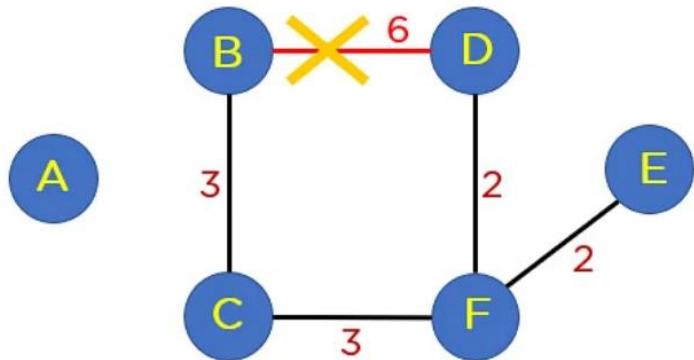
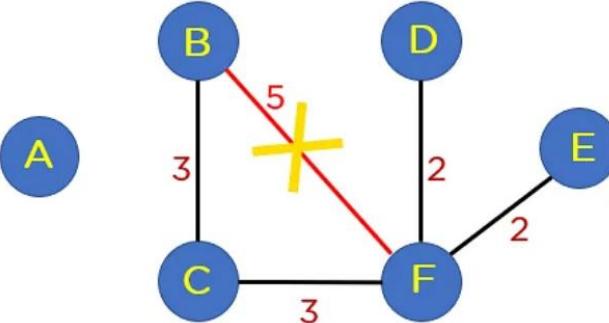
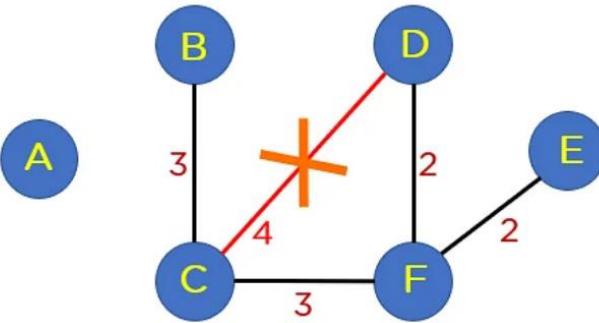
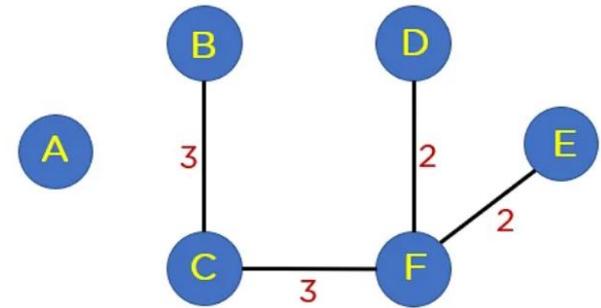
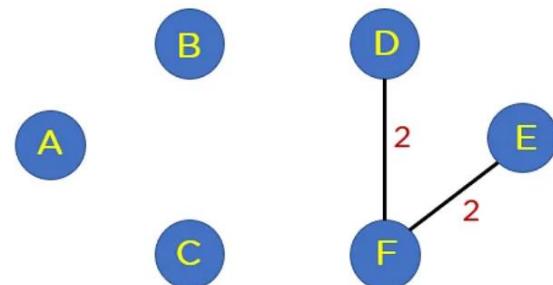
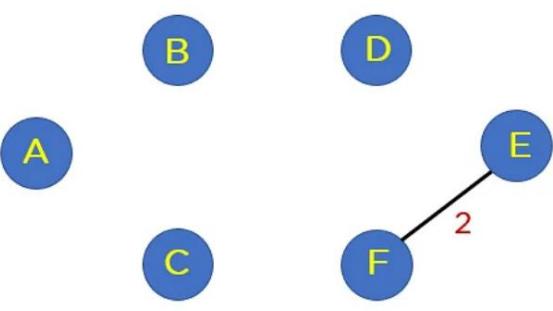
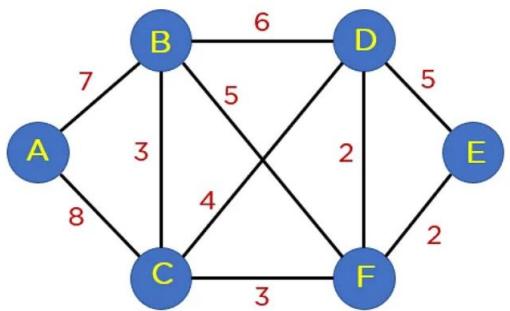
Spanning Tree
Cost = 13



Minimum Spanning
Tree, Cost = 7

Kruskal's algorithm

- We start from the edges with the lowest weight and keep adding edges until we reach our goal.
- The steps for implementing Kruskal's algorithm are as follows:
 - Sort all the edges from low weight to high
 - Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
 - Keep adding edges until we reach all vertices.



Prim's algorithm

- We start from one vertex and keep adding edges with the lowest weight until we reach our goal.
- The steps for implementing Prim's algorithm are as follows:
 - Initialize the minimum spanning tree with a vertex chosen at random.
 - Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
 - Keep repeating step 2 until we get a minimum spanning tree

AVL trees

AVL Trees

- 1 .These are height balanced binary search trees, We balance height of a BST, because we don't want trees with nodes which have large height
2. This can be attained if both subtrees of each node have roughly the same height.
3. AVL tree is a binary search tree where the height of the two subtrees of a node differs by at most one

Height of a null tree is -1

AVL Tree

Que. How to find out balance of a BST.

Ans: By finding the balance factor of a BST.

Balance factor = height of left subtree – height of right subtree

All node's balance factor should be $\{-1, 0, 1\}$

If any node's balance factor is less than -1 or more than 1 then it is not balanced search tree.

If an insertion cause an imbalance, which nodes can be affected?

Nodes on the path of the inserted node.

Let U be the node nearest to the inserted one which has an imbalance.

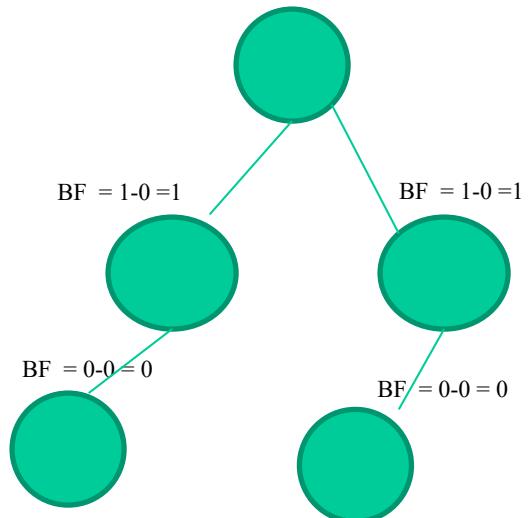
insertion in the left subtree of the left child of U

insertion in the right subtree of the left child of U

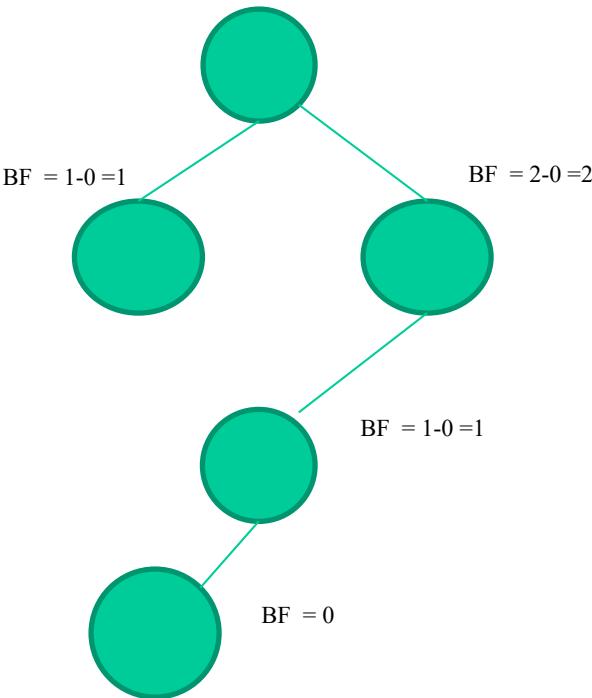
insertion in the left subtree of the right child of U

insertion in the right subtree of the right child of U

BF = 2-2 = 0



BF = 1-3 = -2

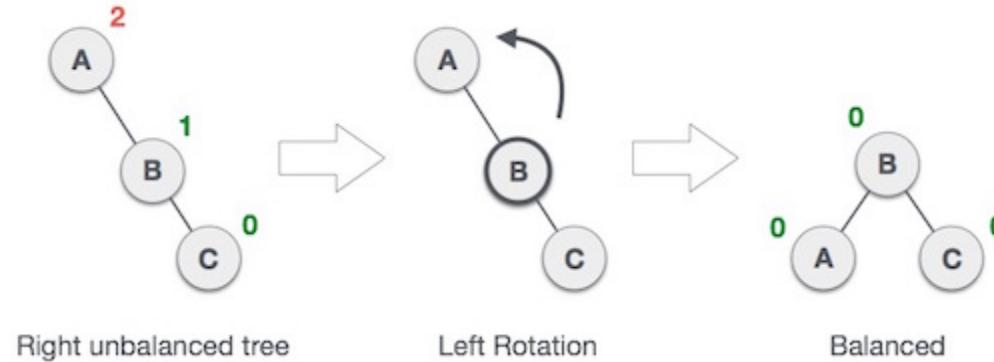


So, at the time of insertion we need to check balance factor of node and if BF is more than 1, then we need to perform rotation.

Rotation is performed always on 3 nodes

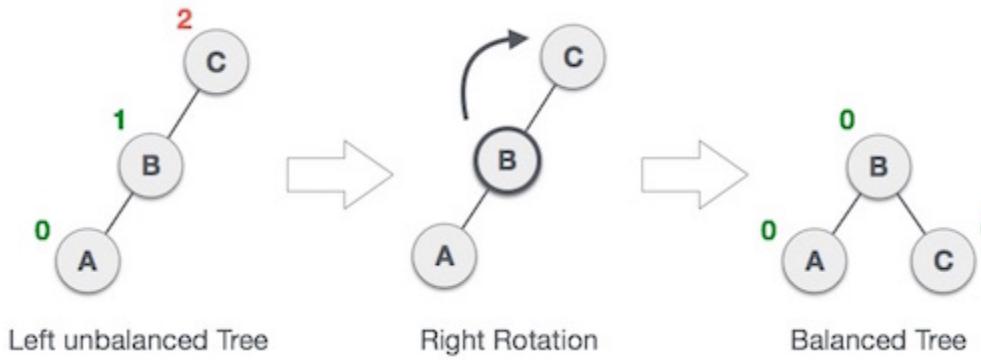
LL rotation

----- RR imbalance



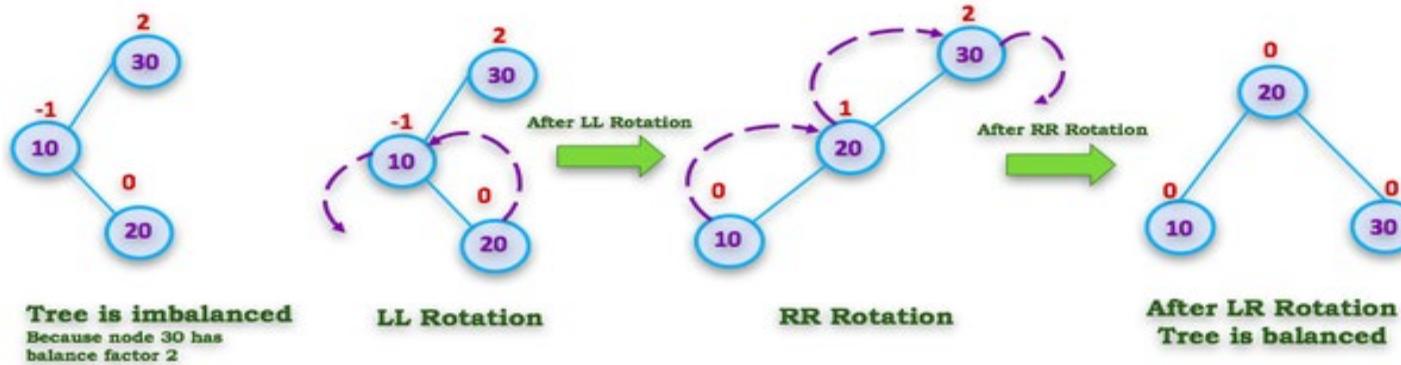
RR rotation

----- LL imbalance



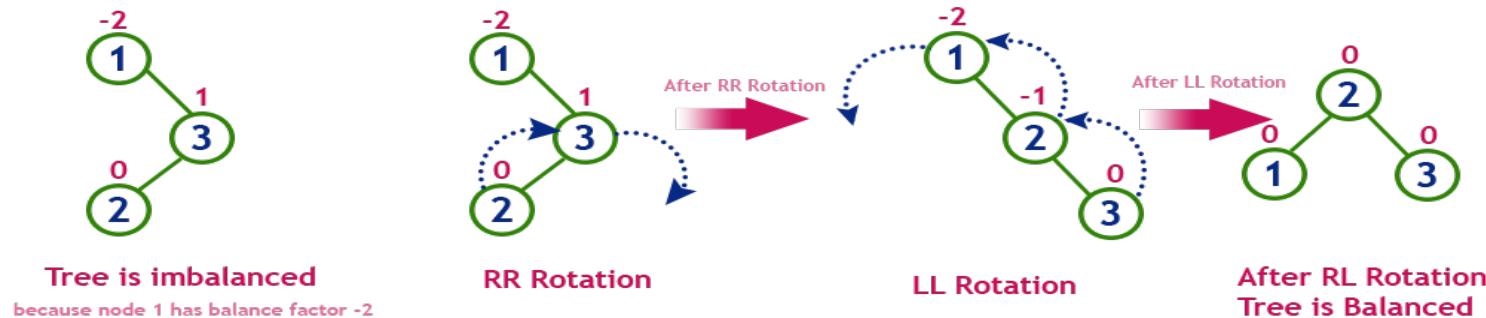
LR rotation

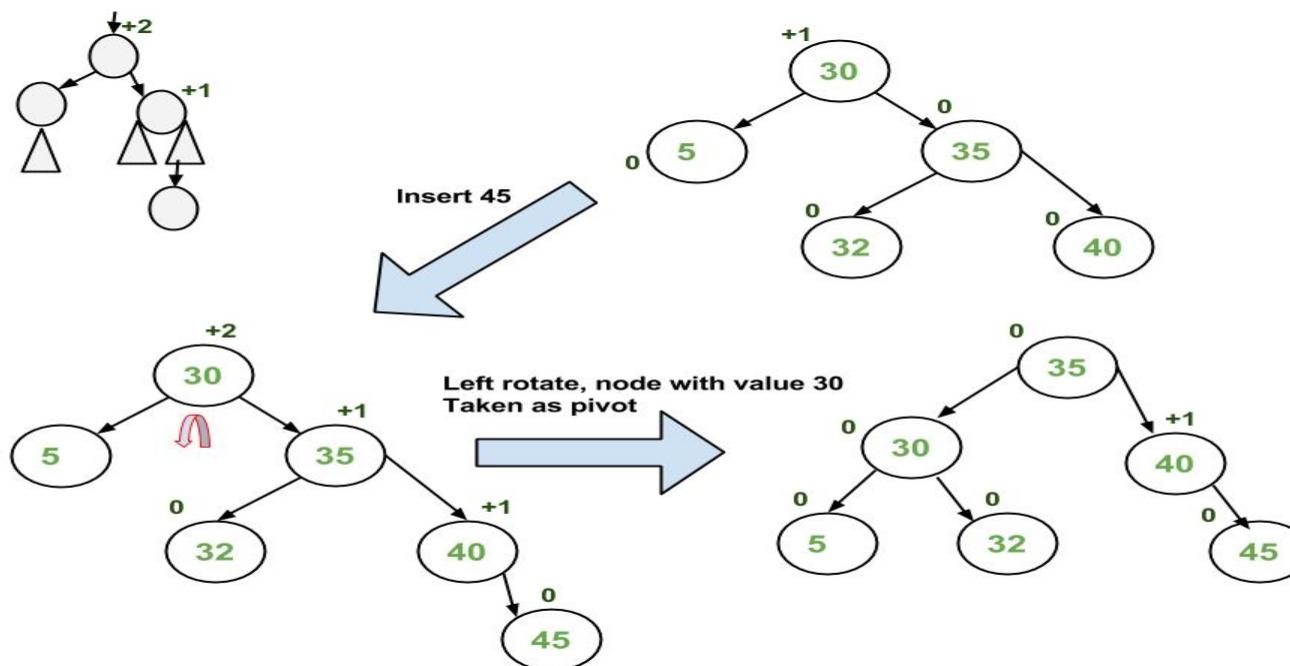
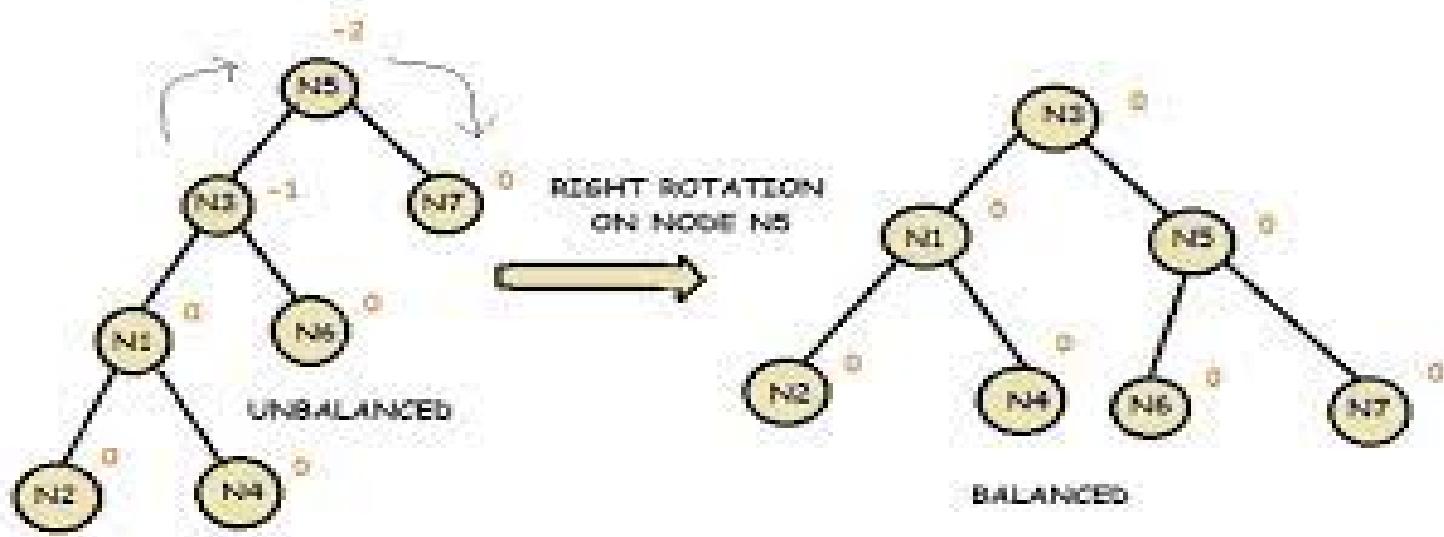
Insert 30,10 and 20



RL rotation

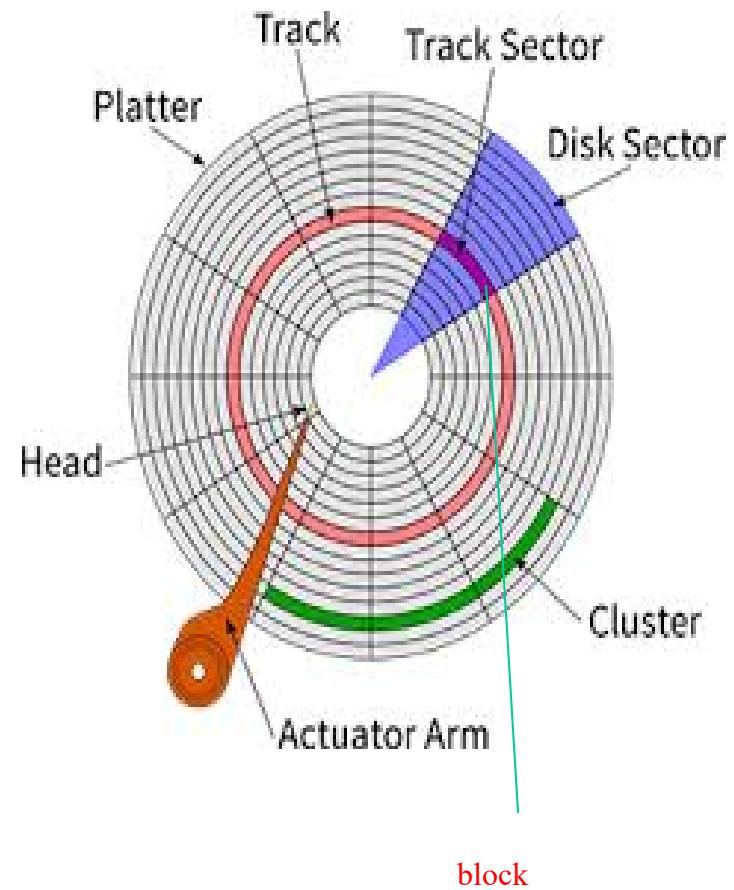
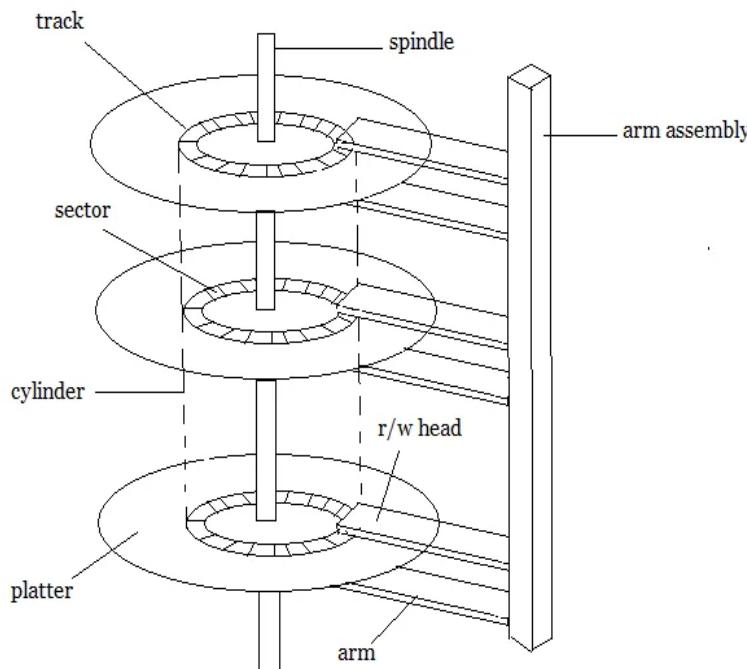
insert 1, 3 and 2





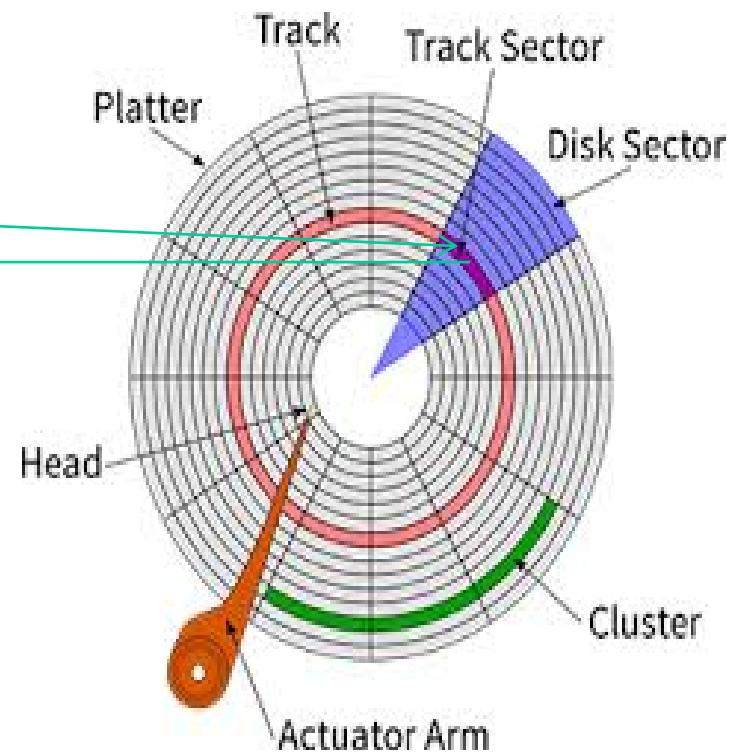
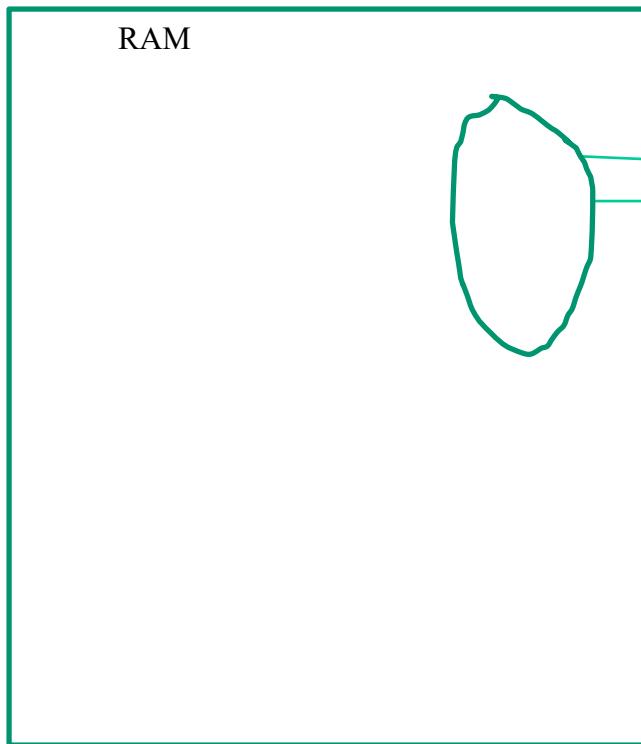
Multiway Search Tree:

1. Disk structure
2. How data is stored
3. What is indexing
4. Multilevel indexing
5. M-Way tree
6. B tree
7. B+ tree



Block: block address means track number + sector number
 Typical block size is 512 bytes.
 We always Read and write in terms of block

Each byte can be accessed by offset.



Data has to be brought into RAM, now how is stored in HDD is DBMS and how it is placed in RAM is DS.

Size of row is 128 bytes

First name – 25

Last name – 25

Address – 50

City – 18

Id - 10

| First Name | Last Name | Address | City | Age |
|------------|-----------|---------------------|----------|-----|
| Mickey | Mouse | 123 Fantasy Way | Anaheim | 73 |
| Bat | Man | 321 Cavern Ave | Gotham | 54 |
| Wonder | Woman | 987 Truth Way | Paradise | 39 |
| Donald | Duck | 555 Quack Street | Mallard | 65 |
| Bugs | Bunny | 567 Carrot Street | Rascal | 58 |
| Wiley | Coyote | 999 Acme Way | Canyon | 61 |
| Cat | Woman | 234 Purrfect Street | Hairball | 32 |
| Tweety | Bird | 543 | Itotitaw | 28 |

No. of records per block = $512/128 = 4$

For 100 records = $100/4 = 25$ blocks for
100 records

In case u perform search for an employee,
we need no. of blocks, we need to access
25 blocks. **Can we do it faster???**

Yes, create index and keep it on disk say
on a block.. How many blocks will be
needed??

Index: 10 for id and 6 for
pointer = 16 bytes

Id pointer

1 Adress

2 Address

3 address

For index

No of entries per block

$512/16 = 32$

Total 100 records

$100/32 = 3.2$ say 4 blocks
are needed for index

1000 records : 250 blocks

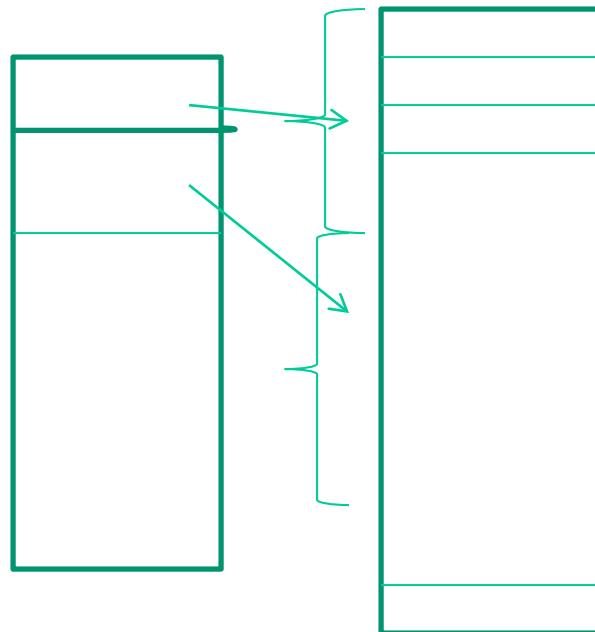
After indexing : 40 blocks

so for 100 records 4 blocks

for $1000 - 40$ blocks

Create index above an index is possible:

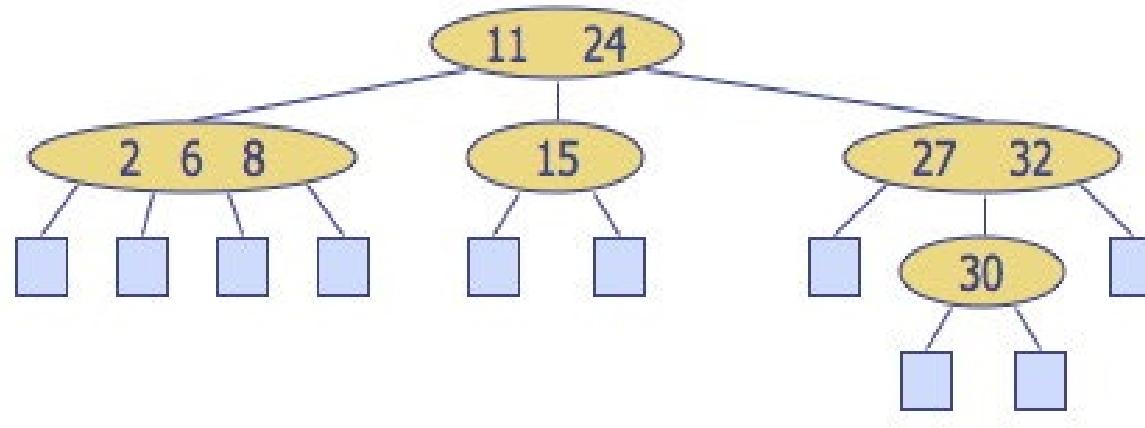
So now per block we can have 32 entries



M-way search tree: each node can have m children, and m-1 keys

Keys – 2 keys

Children – maximum 3, so this is 3-way search tree



Height balanced m-way search tree is B tree.

B-Tree:

1. Root can have min 2 children
2. All leaf nodes are at same level
3. All non leaf nodes should have at least $m/2$ children
4. The creation process is bottom up.

Linked List of students

1. Insert first, last, before Rno, insert sorted on rno
 2. delete by rno
 3. search rno, update marks (update marks)
 4. Search -
Show the records of all students whose name starts from alphabet 'C'
Show the records of all students total marks > 275
transfer these records into a data file
 5. Exit
-

2 LLs are identical or not

```
public static boolean isIdentical(SListNode iter1, SListNode iter2)
{
    while(iter1!=null && iter2!=null)
    {
        if(iter1.getData() != iter2.getData())
            return false;

        iter1=iter1.getNext();
        iter2=iter2.getNext();
    }

    if((iter1==null) && (iter2==null))
        return true;
    else
        return false;
}

public static boolean isPalindrome(List_other ll)
{
    SListNode slow, fast, slowp, second_list;
    boolean res;

    slow = fast = slowp = ll.getHead();

    while((fast!=null) && (fast.getNext()!=null))
    {
        slowp = slow;
        slow = slow.getNext();
        fast = fast.getNext().getNext();
    }

    if(fast.getNext() == null) //odd nodes
    {
        second_list= slow.getNext();
        second_list = reverse(second_list);
        slowp.setNext(null);
        res = isIdentical(ll.getHead(),second_list);
        second_list = reverse(second_list);
        slowp.setNext(slow);
        slow.setNext(second_list);
    }
}
```

```

    }
else
{
    second_list= slowp.getNext();
    second_list = reverse(second_list);
    slow.setNext(null);
    res = isIdentical(ll.getHead(),second_list);
    second_list = reverse(second_list);
    slow.setNext(second_list);
}

return res;
}

```

```

class BTNode {
    private int data;
    private BTNode left, right;

    public BTNode() { data = 0 ; left = null ; right = null; }

    public BTNode(int d) { data = d ; left = null ; right = null; }

    int getData() { return data; }

    BTNode getLeft() { return left; }

    BTNode getRight() { return right; }

    void setData(int d) { data =d ; }

    void setLeft(BTNode l) { left = l; }

    void setRight(BTNode r) { right = r; }

}

class BinaryTree {
    private BTNode root;

    public BinaryTree() { root = null; }

    public void setRoot(BTNode r) { root = r; }

    public BTNode getRoot() { return root; }

    public BTNode createNode(int d)
    {
        BTNode new_node = new BTNode(d);
        return new_node;
    }
}

```

```

public void preOrder()
{
    visit_Preorder(root);
}

private void visit_Preorder(BTNode r)
{
    if(r==null) return;

    sysout(" " + r.getData());
    visit_Preorder(r.getLeft());
    visit_Preorder(r.getRight());
}

public void insert(int d)
{
    BTNode new_node = new BTNode(d)

    BTNode ref;

    BTNodeQueueNode q = new BTNodeQueueNode();

    if(root == null)
    {
        root = new_node;
        return;
    }

    q.add(root);

    while(!q.isEmpty())
    {
        ref = q.remove();
        if(ref.getLeft() == null)
        {
            ref.setLeft(new_node);
            return;
        }
        else
            q.add(ref.getLeft());

        if(ref.getRight() == null)
        {
            ref.setRight(new_node);
            return;
        }
        else
            q.add(ref.getRight());
    }
}

public void level_traverse()
{
    BTNode ref = root;
}

```

```
if(root == null)    return;  
  
Queue <BTNode> q = new LinkedList<BTNode>();  
  
q.add(ref);  
while(!q.isEmpty())  
{  
    ref = q.remove();  
    sysout(ref.getData() + "    ");  
    if(ref.getLeft() !=null)  
        q.add(ref.getLeft());  
    if(ref.getRight() !=null)  
        q.add(ref.getRight());  
}  
}
```

```
public static void main(String []a)
```

```
{  
    BinaryTree bt = new BinaryTree();  
  
    bt.setRoot(bt.createNode(10));  
    bt.getRoot().setRight(bt.createNode(20));  
  
    bt.getRoot().setLeft(bt.createNode(7));  
    bt.preOrder();  
}
```

```
class BTNodeQueueNode {  
    BTNode data;  
    BTNodeQueueNode
```

```
sum leaf node, sum non leaf nodes, find min value, find max value  
count level, mirror image, isIdentical
```

```
public int sum_leafNode()  
{  
    int sum = 0;  
  
    BTNode iter = root;  
  
    //check if root is null and return  
    Queue <BTNode>q = new LinkedList<BTNode>();  
  
    q.add(iter);  
    while(!q.isEmpty())  
    {  
        iter = q.remove();  
        if((iter.getLeft() == null) && (iter.getRight() == null))  
        {  
            sysout(" "+iter.getData());  
            sum = sum + iter.getData();  
        }  
  
        if(iter.getLeft() !=null)  
            q.add(iter.getLeft());  
  
        if(iter.getRight() !=null)  
            q.add(iter.getRight());  
  
    }//end of loop  
    return sum;  
}  
  
public int min()  
{  
    int min = root.getData();  
  
    BTNode iter = root;  
  
    //check if root is null and return  
    Queue <BTNode>q = new LinkedList<BTNode>();  
  
    q.add(iter);  
    while(!q.isEmpty())  
    {  
        iter = q.remove();  
  
        if(iter.getData() < min)  
            min = iter.getData();  
  
        if(iter.getLeft() !=null)  
            q.add(iter.getLeft());  
  
        if(iter.getRight() !=null)  
            q.add(iter.getRight());  
    }
```

```

// write function for max

public int count_level()
{
    int level = 0;
    BTNode iter = root;
    BTNode dummy = new BTNode(-999);
    //check if root is null and return
    Queue <BTNode>q = new LinkedList<BTNode>();

    q.add(iter);
    q.add(dummy);

    while(!q.isEmpty())
    {
        iter = q.remove();

        if(iter.getData() == -999)
        {
            level++;
            q.add(dummy);
            iter=q.remove();
        }

        if(iter.getLeft() != null)
            q.add(iter.getLeft());

        if(iter.getRight() != null)
            q.add(iter.getRight());
    }

    return level;
}

public void swap(BTNode p1)
{
    BTNode t;
    t=p1.getLeft();
    p1.setLeft(p1.getRight());
    p1.setRight(t);
}

public void mirror_image()
{
    BTNode iter = root;

    //check if root is null and return
    Queue <BTNode>q = new LinkedList<BTNode>();

    q.add(iter);
    while(!q.isEmpty())
    {
        iter = q.remove();

        swap(iter);

        if(iter.getLeft() != null)

```

```

        q.add(iter.getLeft());

        if(iter.getRight() !=null)
            q.add(iter.getRight());
    }
    return;
}//end of function

public static boolean isIdentical(BTNode r1, BTNode r2)
{
    BTNode iter1 = r1, iter2 = r2;
    //check in case any one or both BTNode is null, return false

    Queue <BTNode>q1 = new LinkedList<BTNode>();
    Queue <BTNode>q2 = new LinkedList<BTNode>();

    q1.add(iter1);
    q2.add(iter2);

    while((!q1.isEmpty()) && (!q2.isEmpty()))
    {
        iter1=q1.remove();
        iter2=q2.remove();

        if(iter1.getData() != iter2.getData())
            return false;

        if(iter1.getLeft() !=null)
            q1.add(iter1.getLeft());

        if(iter1.getRight() !=null)
            q1.add(iter1.getRight());

        if(iter2.getLeft() !=null)
            q2.add(iter2.getLeft());

        if(iter2.getRight() !=null)
            q2.add(iter2.getRight());
    }

    if(q1.isEmpty() && q2.isEmpty())
        return true;
    else
        return false;
}

```

BST - Binary Search Tree

```

insert, preorder, inorder, postorder, min, max, leaf node, find_height,
del_node

bt.insert(60);

```

```

class BST_Rec {
    BTNode root;

    public BST_Rec()
    {
        root=null;
    }

    public void insert(int d)
    {
        root = insert_BST(root,d);
    }

    private BTNode insert_BST(BTNode ptr, int d)
    {
        if(ptr == null)
            ptr = new BTNode(d);

        else if(d < ptr.getData())
            ptr.setLeft(insert_BST(ptr.getLeft(),d));

        else if(d > ptr.getData())
            ptr.setRight(insert_BST(ptr.getRight(),d));

        else
            sysout("\n Duplicate data...");

        return ptr;
    }

    public void preOrder()
    {
        visit_preOrder(root);
    }

    private void visit_preOrder(BTNode r)
    {
        if(r==null)  return;

        sysout(" " +r.getData());
        visit_preOrder(r.getLeft());

        visit.preOrder(r.getRight());
    }

    public BTNode search(int key)
    {
        return BST_Search(root,key);
    }

    private BTNode BST_Search(BTNode r, int k)
    {
        if(r == null)
        {
            sysout("Key not found...\n");
            return null;
        }
    }
}

```

```

        }
        else if(k < r.getData())
            return BST_Search(r.getLeft(),k);

        else if(k > r.getData())
            return BST_Search(r.getRight(),k);

        else return r;
    }

public int find_min()
{
    return min_BST(root);
}

private int min_BST(BTNode r)
{
    if(r.getLeft()==null)
        return r.getData();

    return min_BST(r.getLeft());
}

}

public class BSTMain {
    public static void main(String []a)
    {
        BST_Rec bst = new BST_Rec();

        bst.insert(20);
        bst.insert(25);
        bst.insert(15);
        bst.insert(35);
        bst.insert(22);
        bst.insert(32);
        bst.insert(8);
        bst.insert(10);
        bst.insert(18);
    }
}

```



```

public void insert(int d)
{
    BTNode ptr, par;

    if(root == null) { root = new BTNode(d); return; }
    ptr=root;

    while(ptr!=null)
    {
        par = ptr;
        if(d < ptr.getData())
            ptr = ptr.getLeft();

        if(d > ptr.getData())
            ptr = ptr.getRight();

        else
        {
            sysout("Duplicate data..."); 
            return;
        }
    }
    ptr = new BTNode(d);

    if( d < par.getData())
        par.setLeft(ptr);
    else
        par.setRight(ptr);

    return;
}

public BTNode search(int d)
{
    BTNode ptr = root;

    while(ptr!=null)
    {
        if( d == ptr.getData())
            return ptr;
        else if( d < ptr.getData())
            ptr = ptr.getLeft();
        else ptr = ptr.getRight();
    }

    sysout("Not found..."); 
    return null;
}

public int min_nonRec()
{
    BTNode ptr = root;

    if(ptr == null) { sysout("Tree is empty.."); return; }

    while(ptr.getLeft() !=null)
        ptr = ptr.getLeft();

```

```

        return ptr.getData();
    }

public void del(int d)
{
    root = del_node_BST(root, d);
}

private BTNode del_node_BST(BTNode r, int d)
{
    BTNode deletable, succ;

    if(r == null)
    {
        sysout("Empty...");
        return null;
    }
    if( d < r.getData())           // delete from left subtree
        r.setLeft(del_node_BST(r.getLeft(),d));

    else if( d > r.getData())    // delete from right subtree
        r.setRight(del_node_BST(r.getRight(),d));

    else
    {
        if(r.getLeft() != null && r.getRight() != null)
        {
            succ = r.getRight();
            while(succ.getLeft() !=null)
                succ = succ.getLeft();

            r.setData(succ.getData());
            r.setRight(del_node_BST(r.getRight(),succ.getData()));
        }
        else
        {
            deletable = r;
            if(r.getLeft() != null)    // only left child
                r = r.getLeft();
            else if(r.getRight() != null) // only right child
                r = r.getRight();
            else
                r = null;

            deletable= null;
        }
    }
    return r;
}//end of function

public int height()
{
    return find_ht(root);
}

```

```
int find_ht(BTNode ptr)
{
    int h_left, h_right;

    if(ptr == null)  return 0;

    h_left = find_ht(ptr.getLeft());
    h_right = find_ht(ptr.getRight());

    if( h_left > h_right)
        return 1 + h_left;
    else
        return 1 + h_right;
}
```


Sequential search

Binary search

In an array search a value.

23

```
public static int search(int arr[], int key)
{
    int ;

    for(i=0 ; i<arr.length; i++)
    {
        if(arr[i] == key)
        {
            return i;
        }
    }
    return -999;
}
```

```
public static int binary_search(int arr[],int key)
```

```
{
    int low, high, mid;
    low = 0;
    high = arr.length-1;

    mid = (low+high)/2;

    while((arr[mid] !=key)&&(low<high))
    {
        if(key < arr[mid])
            high = mid - 1;
        else
            low = mid + 1;

        mid = (low + high) /2;
    }

    if(arr[mid] == key)
        return mid;
    else
        return -999;
}
```

```
public static int BinSearchRotatedArray(int arr[],int key)
```

```
{

while(low<=high)
{
    mid = (low+high)/2;

    if(arr[mid] == key)  return mid;

    else if(arr[low] <= arr[mid])
    {
        if(key>=arr[low] && key<arr[mid])
            high = mid - 1;
    }
}
```

```

        else
            low = mid + 1;
    }
else if(key>arr[mid] && key<=arr[high])
    low = mid + 1;
else
    high = mid - 1;
}
return -999;
}

```

| Infix | Postfix | Prefix |
|-------------------|---------|---------|
| A + B * C + D ==> | ABC*D+ | ++A*BCD |
| (A+B) * (C+D) ==> | AB+CD+* | *+AB+CD |
| A * B + C * D ==> | AB*CD*+ | +*AB*CD |
| A + B + C + D ==> | AB+C+D+ | ++ABCD |

```

public class Conversion {
    String exp;

    public Conversion(String s)
    {
        exp = new String(s);
    }

    public int perce(char ch)
    {
        switch(ch) {
            case '(' : return 0;
            case '+' :
            case '-' : return 1;
            case '*' :
            case '/' :
            case '%' : return 2;
            case '^' : return 3;
            default   : return 0;
        }
    }

    public String infix_to_postfix()
    {
        myCharStack st = new myCharStack();           //st=
post=AB+CD+*
        char symbol, next;
        int i, p =0;
        char post[] = new char[exp.length+1];

        for(i=0 ; i<exp.length ;i++)
        {
            symbol = exp.charAt(i);                //symbol = (
            if(symbol != 32)
            {
                switch(symbol) {
                    case '(' : st.push(symbol);

```

```

        break;

    case ')' : while((next=st.pop() != ')')
                      post[p++] = next;
        break;

    case '+' :
    case '-' :
    case '*' :
    case '/' :
    case '%' :
    case '^' : while((!st.isEmpty() && (prece(st.peek()) >=
prece(symbol)))
                    {
                        post[p++] = st.pop();
                    }

                st.push(symbol);
                break;

            default : post[p++] = symbol;
        }
    } //end of if 32
} //end of loop

while(!st.isEmpty())
    post[p++] = st.pop();

return new String(post);
}

public String infix_to_prefix()
{
    myCharStack st = new myCharStack();
    char symbol, next;
    int i, p =0;
    char pre[] = new char[exp.length+1];

    for(i=exp.length-1 ; i>=0 ;i--)
    {
        symbol = exp.charAt[i];
        if(symbol != 32)
        {
            switch(symbol) {
            case ')' : st.push(symbol);
                break;

            case '(' : while((next=st.pop() != ')'))
                            pre[p++] = next;
                break;

            case '+' :
            case '-' :
            case '*' :
            case '/' :
            case '%' :
            case '^' : while((!st.isEmpty() && (prece(st.peek()) >
prece(symbol)))

```

```

        {
            pre[p++] = st.pop();
        }

        st.push(symbol);
        break;

    default : pre[p++] = symbol;
}
}//end of if 32
}//end of loop

while(!st.isEmpty())
    pre[p++] = st.pop();

//reverse pre

return new String(post);
}

```

```

int pow(int a,int b)
{
    int i,p=1;
    for(i=1 ; i<=b ;i++)
        p = p * a;

    return p;
}

// will work on single digit operands

public int eval_pre(String prefix)      // ++A*BCD      ++5*342      st =
19
{
    int a,b,result,i;

    myIntStack st = new myIntStack();

    char pre[] = new char[exp.length+1];

    pre = prefix.toCharArray();

    for( i=pre.length-1; i>=0 ;i--)
    {
        if(pre[i] >='0' && pre[i]<='9')
        {
            st.push(pre[i] - 48);
        }
        else
        {
            if(!st.isEmpty())  b = st.pop();
            if(!st.isEmpty())  a = st.pop();

            switch(pre[i])
            {
                case '+' :  res = b + a; break;

```

```
        case '-' : res = b - a; break;
        case '+' : res = b * a; break;
        case '/' : res = b / a; break;
        case '%' : res = b % a; break;
        case '^' : res = pow(b,a); break;
    }
    st.push(res);
}//end of else
}//end of for
return st.pop();
}//end of fucntion
```

Sorting

Types of sorting

- Internal Sorting: If the data to be sorted is small and can be kept in main memory and sorting is performed, then it is called as internal sorting.
- External Sorting: If the data is large which can't be placed in main memory at a time, then the data that is currently being sorted is brought in main memory and rest is on secondary memory. This type of sorting is external sorting.
- **We are discussing only Internal Sorting as part of our syllabus.**

Classification of Sorting Algorithm

- **By Number of Comparisons:** In this method, sorting algorithms are classified based on the number of comparisons.
- For comparison based sorting algorithms, best case behavior is $O(n \log n)$ and worst case behavior is $O(n^2)$.
- Comparison-based sorting algorithms evaluate the elements of the list by key comparison operation and need at least $O(n \log n)$ comparisons for most inputs.

- **By Number of Swaps:** In this method, sorting algorithms are categorized by the number of swaps.
- **By Memory Usage:** Some sorting algorithms are such, that they need $O(1)$ or $O(\log n)$ memory to create auxiliary locations for sorting the data temporarily.
- **By Recursion:** Sorting algorithms are either recursive [quick sort] or non-recursive [selection sort, and insertion sort], and there are some algorithms which use both (merge sort).

- **By Stability:** Sorting algorithm is stable if the relative order of original list is maintained in case of duplicate key values.
- **By Adaptability:** With a few sorting algorithms, the complexity changes based on pre-sortedness [quick sort]: presortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive

Bubble-Sort

- Bubble-Sort is the slowest algorithm for sorting, but it is heavily used, as it is easy to implement.
- In Bubble-Sort, we compare each pair of adjacent values.
- We want to sort values in increasing order so if the second value is less than the first value then we swap these two values. Otherwise, we will go to the next pair.
- We will have N number of passes to get the array completely sorted. After the first pass, the largest value will be in the rightmost position.

Bubble Sort: Performance

Worst case complexity : $O(n^2)$

Best case complexity (Improved version) : $O(n)$

Average case complexity (Basic version) : $O(n^2)$

Worst case space complexity : $O(1)$ auxiliary

Selection-Sort

- Selection sort is an in-place sorting algorithm.
- Selection sort works well for small files. It is used for sorting the files with very large values and small keys.
- Here selection is made based on keys and swaps are made only when required.
- Advantages • Easy to implement • In-place sort (requires no additional storage space)
- Disadvantages • Time complexity is $O(n^2)$

Algorithm

1. Find the minimum value in the list
2. Swap it with the value in the current position
3. Repeat this process for all the elements until the entire array is sorted This algorithm is called selection sort since it repeatedly selects the smallest element.

Performance of selection sort

- Worst case complexity : $O(n^2)$
- Best case complexity : $O(n^2)$
- Average case complexity : $O(n^2)$
- Worst case space complexity: $O(1)$ auxiliar

Insertion Sort

- It is a simple and efficient comparison sort.
- Each iteration removes an element from the input data and inserts it into the correct position in the list being sorted.
- The choice of the element being removed from the input is random and this process is repeated until all input elements have gone through.

Advantages

- Simple implementation
- Efficient for small data
- Adaptive: If the input list is presorted [may not be completely] then insertions sort takes $O(n + d)$, where d is the number of inversions
- Practically more efficient than selection and bubble sorts, even though all of them have $O(n^2)$ worst case complexity
- Stable: Maintains relative order of input data if the keys are same
- In-place: It requires only a constant amount $O(1)$ of additional memory space

Performance of insertion sort

- Worst case complexity : $O(n^2)$
- Best case complexity : $O(n^2)$
- Average case complexity : $O(n^2)$
- Worst case space complexity: $O(n^2)$ $O(1)$ auxiliar

Merge Sort

- Merge sort is based divide and conquer strategy.
- Can be implemented as external sorting, when the dataset is so big that it is impossible to load the whole dataset into memory, here sorting is done in chunks.
- Merging is the process of combining two sorted files to make one bigger sorted file.
- Selection is the process of dividing a file into two parts: k smallest elements and $n - k$ largest elements.
- Selection and merging are opposite operations
 - selection splits a list into two lists
 - merging joins two files to make one file

Performance

- Worst case complexity : $O(n \log n)$
- Best case complexity : $O(n \log n)$
- Average case complexity : $O(n \log n)$
- Worst case space complexity: $O(n)$ auxiliary

Quick Sort

- It is an example of a divide-and-conquer algorithmic technique.
- It is also called partition exchange sort.
- It uses recursive calls for sorting the elements, and it is one of the famous algorithms among comparison-based sorting algorithms.
- Divide: The array $A[\text{low} \dots \text{high}]$ is partitioned into two non-empty sub arrays $A[\text{low} \dots q]$ and $A[q + 1 \dots \text{high}]$, such that each element of $A[\text{low} \dots \text{high}]$ is less than or equal to each element of $A[q + 1 \dots \text{high}]$. The index q is computed as part of this partitioning procedure.
- Conquer: The two sub arrays $A[\text{low} \dots q]$ and $A[q + 1 \dots \text{high}]$ are sorted by recursive calls to Quick sort.

Algorithm

- The recursive algorithm consists of four steps:
 - 1) If there are one or no elements in the array to be sorted, return.
 - 2) Pick an element in the array to serve as the “pivot” point. (Usually the left-most element in the array is used.)
 - 3) Split the array into two parts – one with elements larger than the pivot and the other with elements smaller than the pivot.
 - 4) Recursively repeat the algorithm for both halves of the original array.

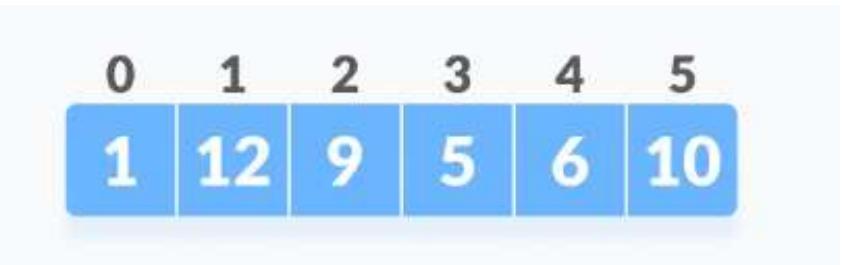
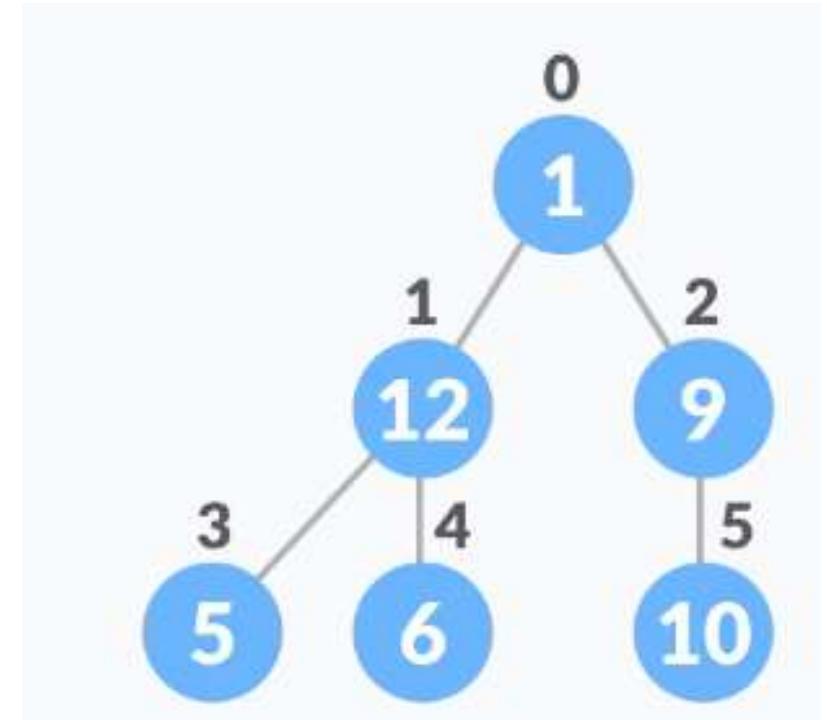
Performance

- Worst case Complexity: $O(n^2)$
- Best case Complexity: $O(n \log n)$
- Average case Complexity: $O(n \log n)$
- Worst case space Complexity: $O(1)$

Heapsort

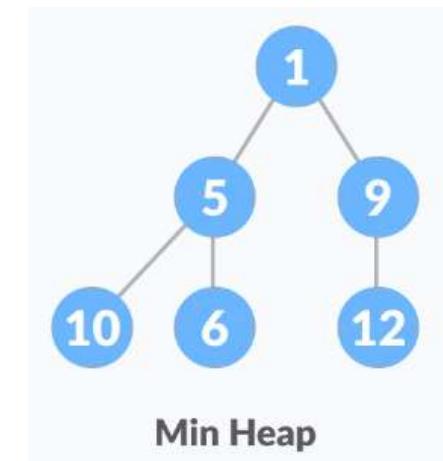
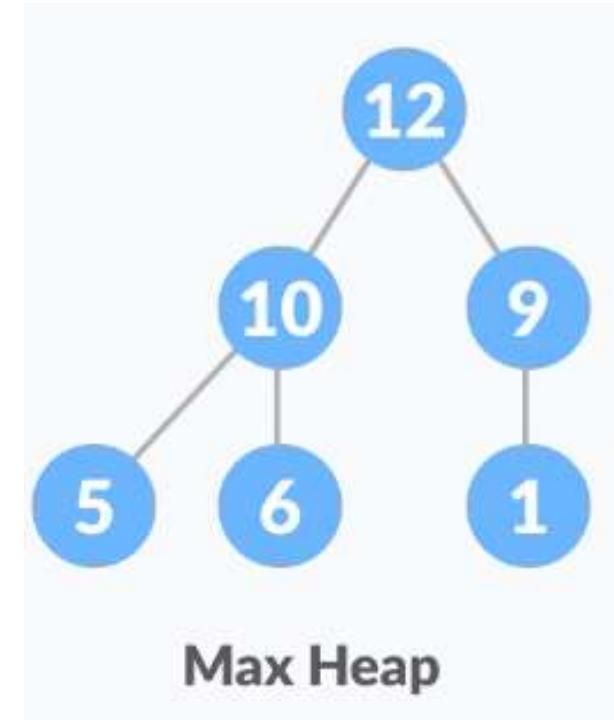
Relationship between Array Indexes and Tree Elements

- A complete binary tree has an interesting property that we can use to find the children and parents of any node.
- If the index of any element in the array is i
 - the element in the index $2i+1$ will become the left child
 - the element in the index $2i+2$ will become the right child
- Also, the parent of any element at index i is given by the lower bound of $(i-1)/2$.



Heap Data Structure

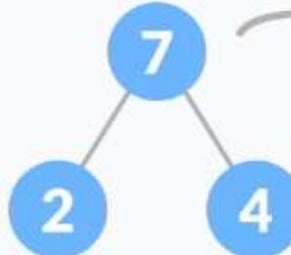
- Heap is a special tree-based data structure
- A binary tree is said to follow a heap data structure if
 - it is a complete binary tree
 - All nodes in the tree follow the property that
 - They are greater than their children : Max-Heap
 - Or all nodes are smaller than their children : Min-Heap



What is Heapify

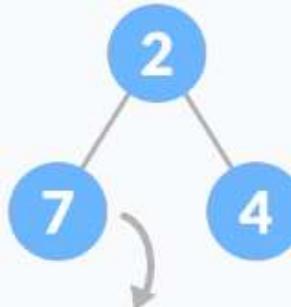
- Heapify is the process of creating a heap data structure from a binary tree.

Scenario-1

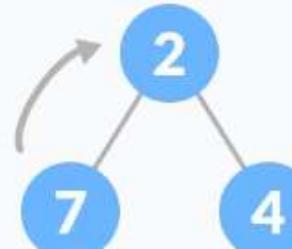


parent is already
the largest

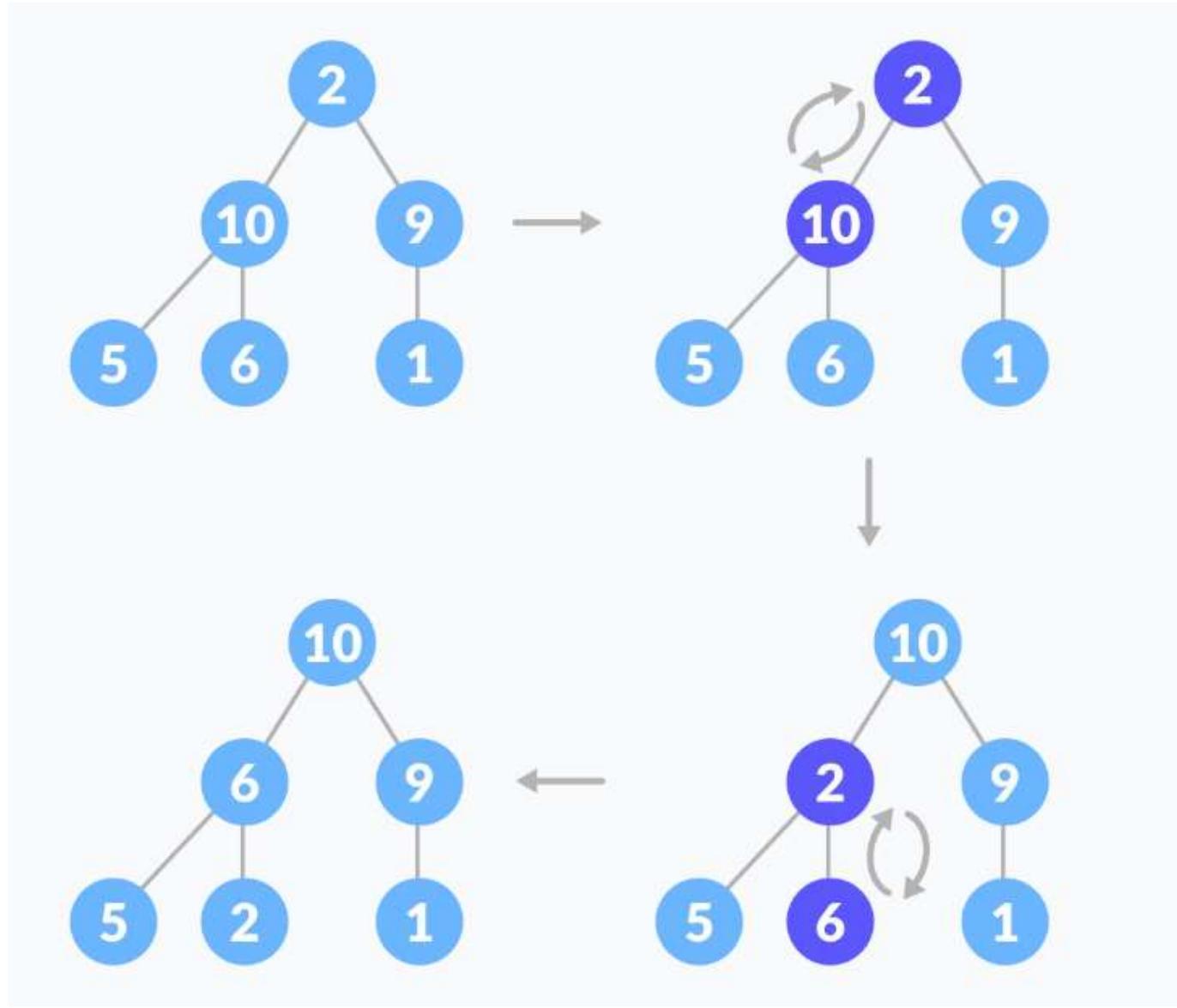
Scenario-2



child is greater
than the parent



parent is now
the largest



Max-Heapify Operation

Algorithm 1: Max-Heapify Pseudocode

Data: B : input array; s : an index of the node
Result: Heap tree that obeys max-heap property

Procedure Max-Heapify(B, s)

```
    left = 2s;
    right = 2s + 1;
    if left ≤ B.length and B[left] > B[s] then
        | largest = left;
    else
        | largest = s;
    end
    if right ≤ B.length and B[right] > B[largest] then
        | largest = right;
    end
    if largest ≠ s then
        | swap(B[s], B[largest]);
        | Max-Heapify(B, largest);
    end
end
```

Building max-heap

- To build a max-heap from any tree, we can thus start heapifying each sub-tree from the bottom up and end up to the root element.
- Start our algorithm with a node that is at the lowest level of the tree and has children node ($n/2 - 1$)
- Continue this process and make sure all the subtrees are following the max-heap property

```
// Build heap (rearrange array)
for (int i = n / 2 - 1; i >= 0; i--)
    heapify(arr, n, i);
```

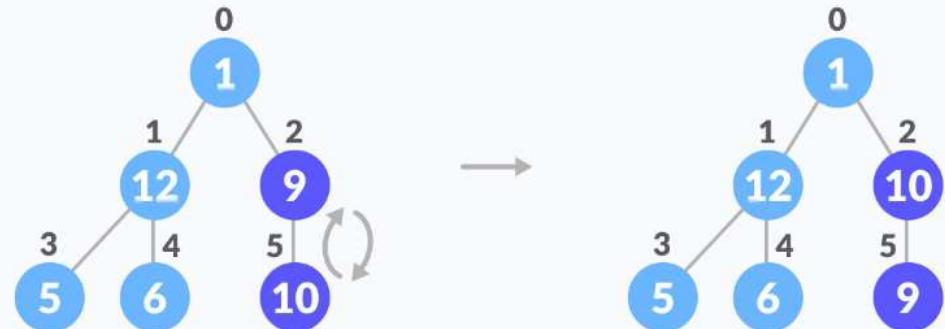
Example

arr 0 1 2 3 4 5
 1 12 9 5 6 10

n = 6

i = $6/2 - 1 = 2$ # loop runs from 2 to 0

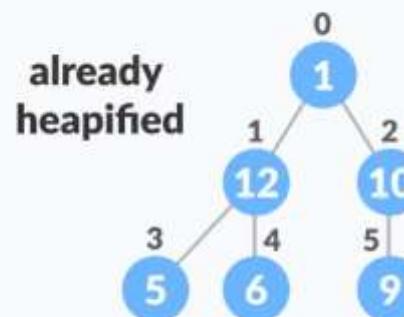
i = 2 → heapify(arr, 6, 2)



0 1 2 3 4 5
1 12 9 5 6 10

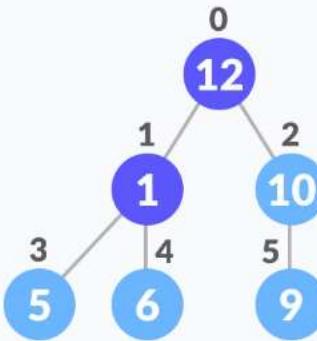
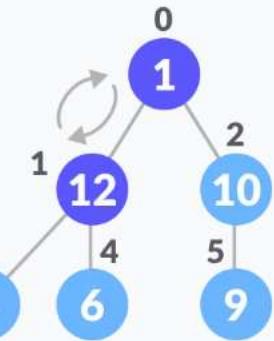
0 1 2 3 4 5
1 12 10 5 6 9

i = 1 → heapify(arr, 6, 1)



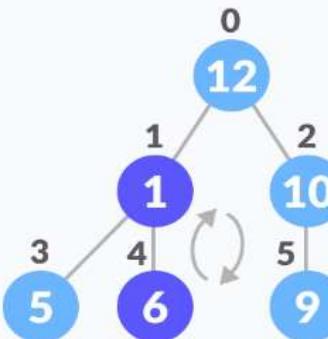
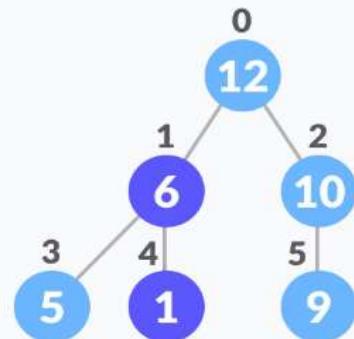
0 1 2 3 4 5
1 12 10 5 6 9

$i = 0 \longrightarrow \text{heapify(arr, } 6, 0)$



| | | | | | |
|---|----|----|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
| 1 | 12 | 10 | 5 | 6 | 9 |

| | | | | | |
|----|---|----|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
| 12 | 1 | 10 | 5 | 6 | 9 |



| | | | | | |
|----|---|----|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
| 12 | 6 | 10 | 5 | 1 | 9 |

| | | | | | |
|----|---|----|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
| 12 | 1 | 10 | 5 | 6 | 9 |

Heap Sort



Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.



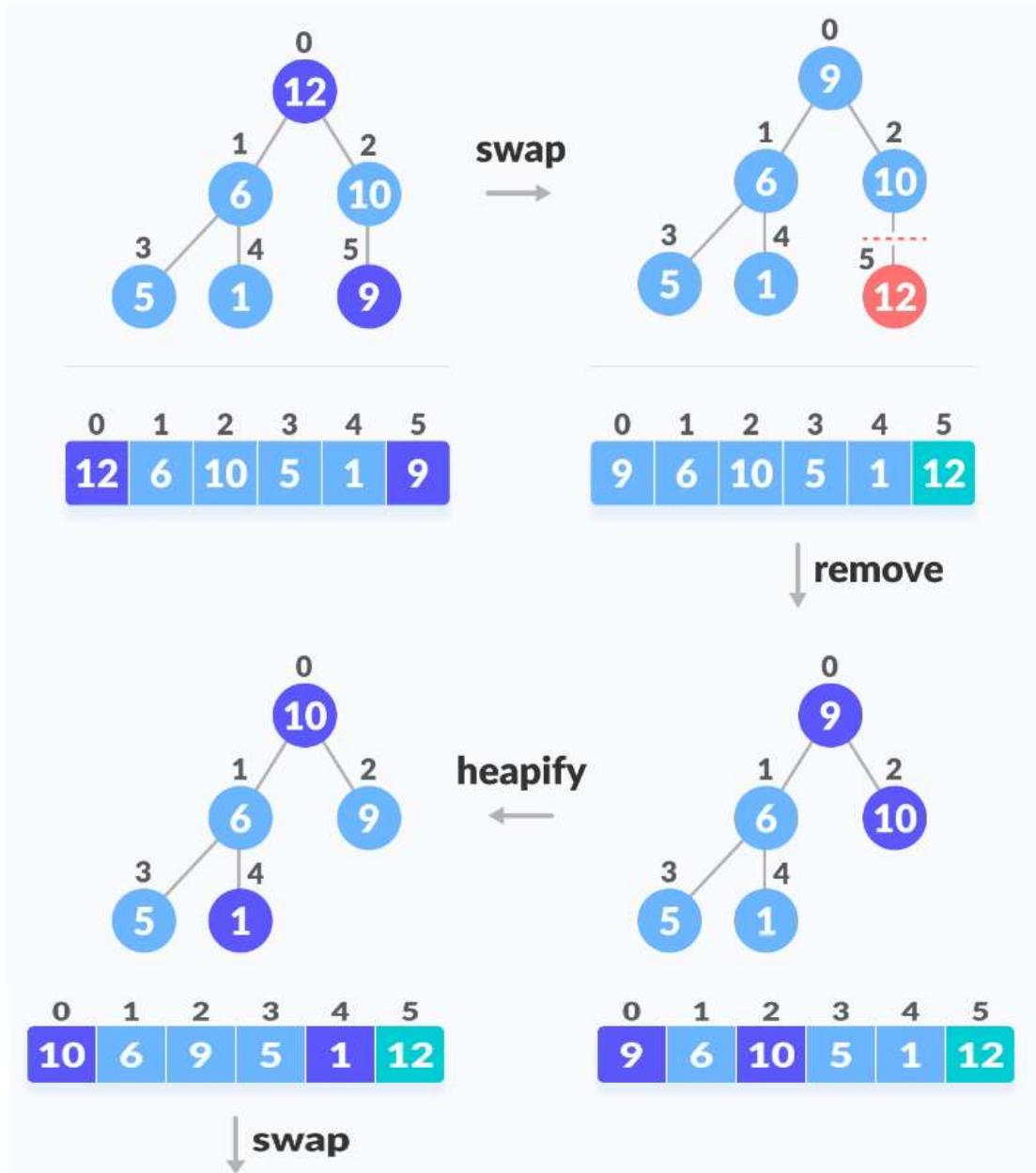
Swap: the root element and the last array element

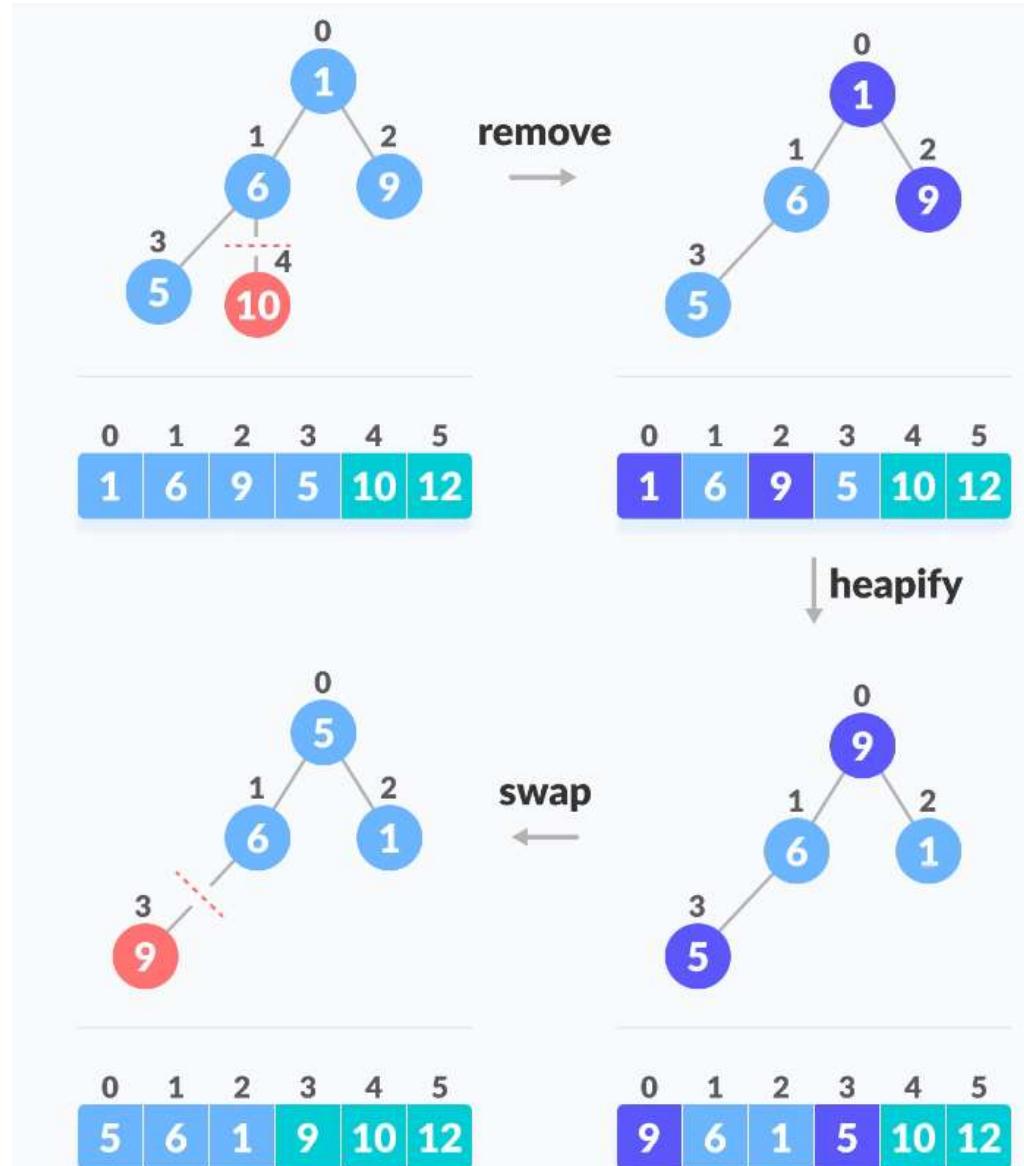


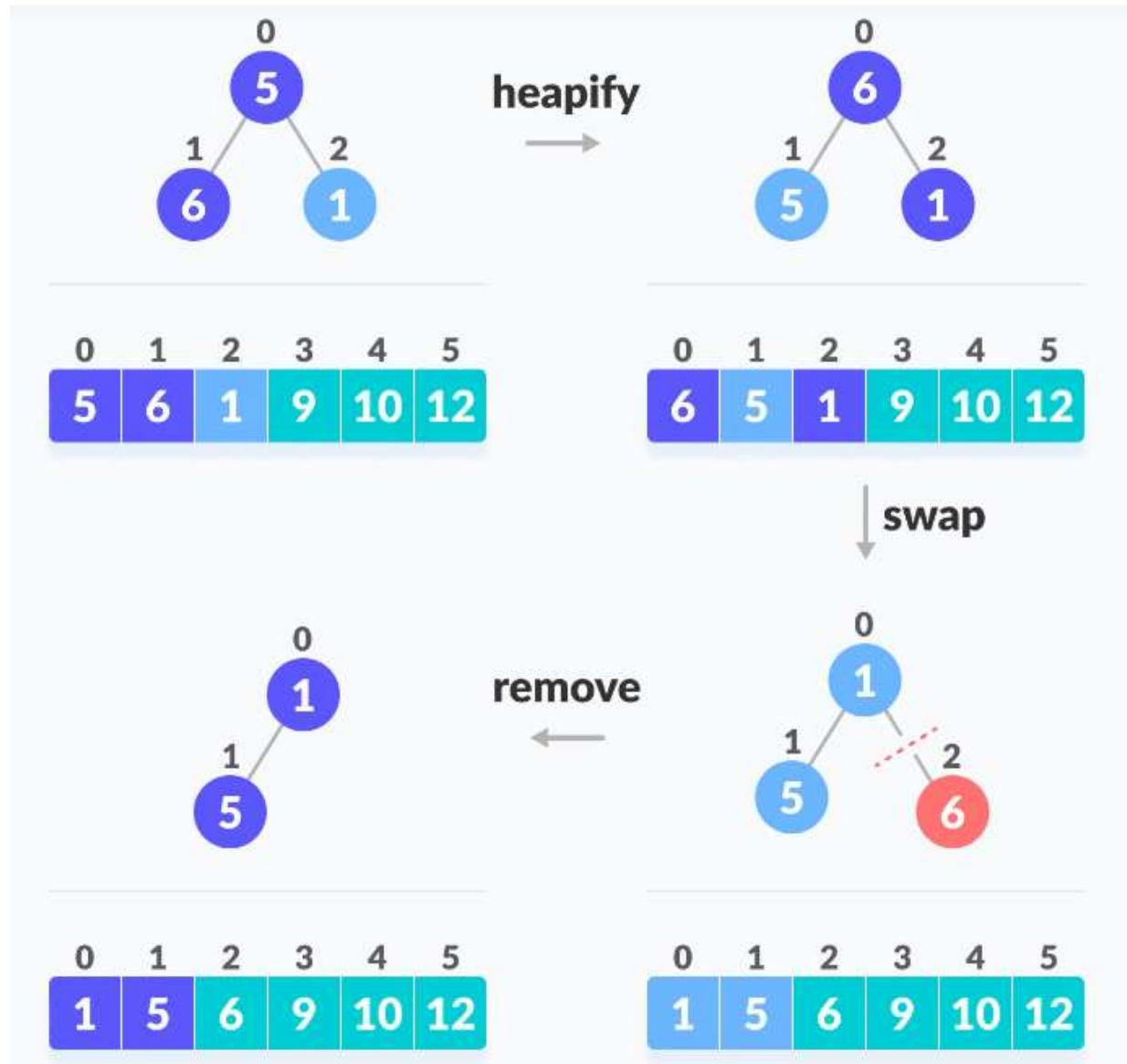
Remove: Reduce the size of the heap by 1.

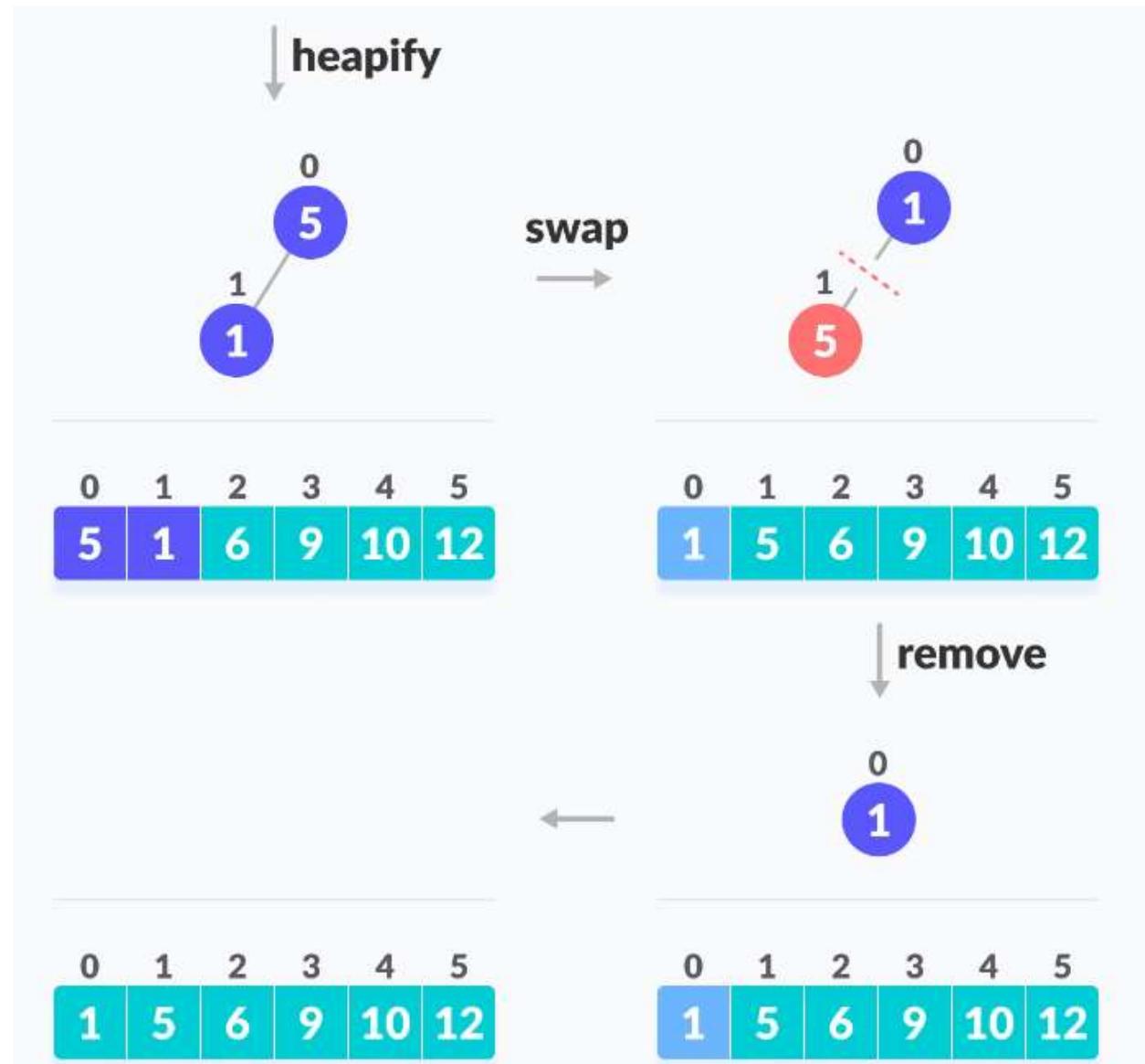


Heapify: Heapify the root element again so that we have the highest element at root.









```
// Heap sort
for (int i = n - 1; i >= 0; i--) {
    swap(&arr[0], &arr[i]);

    // Heapify root element to get highest element at root again
    heapify(arr, i, 0);
}
```

Heap Sort Complexity

| Time Complexity | |
|------------------|---------------|
| Best | $O(n \log n)$ |
| Worst | $O(n \log n)$ |
| Average | $O(n \log n)$ |
| Space Complexity | $O(1)$ |

```

public static void Bubble_sort(int arr[],int n)
{
    int i,j,t;
    int flag = 0;

    for(i=0;i<=n;i++)
    {
        flag=0;
        for(j=0;j<n-i;j++)
        {
            if(arr[j] > arr[j+1])
            {
                flag=1;
                t=arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = t;
            }
        }
        if(flag==0)
            break;
        sysout("\n pass" + i + " ");
        display(arr,n);
    }
}

```

```

public static void Selection_sort(int arr[],int n)
{
    int i,j,t,min;

    for(i=0;i < n;i++)
    {
        min = i;
        for(j=i+1; j<=n ;j++)
        {
            if(arr[min] > arr[j])  min=j;
        }
        if(i!=min)
        {
            t=arr[i];
            arr[i]=arr[min];
            arr[min]=t;
        }
    }
}

```

```

public static void insertion_sort(int arr[],int n)
{
    int i,j,k;

    for(i=1;i<=n;i++)
    {
        k=arr[i];

```

```

    for(j=i-1;j>=0 && k<arr[j];j--)
        arr[j+1] = arr[j];

    arr[j+1]=k;
} //end of outer loop

}



---


arr[10] = {12,23,44,56,67,78,88,99,-999};

i

brr[10] = {2,4,5,78,90,890,5678,-999};

j

res[20] = {2, 4, 5, 12, 23, 44,56, 67,78,88,90,99,890,5678,-999}

tar

public static void merge_Ver1(int arr[],int brr[],int res[])
{
    int i,j,tar;
    i=0; j=0; tar =0;

    while((arr[i]!=-999) && (brr[j]!=-999))
    {
        if(arr[i] < brr[j])
        {
            res[tar] = arr[i];
            i=i+1;
            tar=tar+1;
        }
        else if(brr[j] < arr[i])
        {
            res[tar] = brr[j];
            j=j+1;
            tar = tar+1;
        }
        else
        {
            res[tar]=arr[i];
            i++ ; j++; tar++;
        }
    }

    while(arr[i]!=-999)
    {
        res[tar] = arr[i];
        tar++; i++;
    }

    while(brr[j]!=-999)
    {
        res[tar] = brr[j];
        tar++; j++;
    }
}

```

```
 }  
}  


---


```

```
public static void merge_Ver2(int arr[],int tar[],int low1,int high1,int  
low2,int high2)  
{  
    int i,j,k;  
    i=low1;  
    j=low2;  
  
    k=low1;  
  
    while((i<=high1) && (j<=high2))  
    {  
        if(arr[i] < arr[j])  
        {  
            tar[k] = arr[i];  
            i++; k++;  
        }  
        else if(arr[j] < arr[i])  
        {  
            tar[k] = arr[j];  
            j++; k++;  
        }  
        else  
        {  
            tar[k] = arr[i];  
            i++; j++; k++;  
        }  
    }  
    while(i<=high1)  
    {  
        tar[k] = arr[i];  
        i++; k++;  
    }  
    while(j <= high2)  
    {  
        tar[k] = arr[j];  
        j++; k++;  
    }  
}//end of function
```

```
public static void merge_sort(int arr[], int low, int high)  
{  
    int mid;  
    int temp[] = new int[arr.length];  
  
    if(low < high)  
    {  
        mid = (low + high) / 2;  
        merge_sort(arr, low, mid);  
        merge_sort(arr, mid+1, high);  
    }
```

```

        merge_Ver2(arr,temp,low,mid,mid+1,high);
        copy_arr(arr,temp,low,high);

    }

}

public static void copy(int arr[], int temp[], int low,int high)
{
    int i;
    for(i=low; i<=high;i++)
        arr[i] = temp[i];
}

```

Quick sort

48 44 19 59 72 80 42 65 82 8 95 68

```

public static int partition(int arr[], int low, int high)
{
    int left, right, pivot, t;

    pivot=arr[low];
    left=low;
    right =high;

    while(left <= high)
    {
        while((arr[left]<=pivot) && (left<high))
            left++;

        while(arr[right] > pivot)
            right--;

        if(left < right)
        {
            t=arr[left]; arr[left] = arr[right] ; arr[right] = t;
            left++; right--;
        }
        else left++;
    }

    arr[low] = arr[right];
    arr[right] =pivot;

    return right;
}// end of function

public static void QuickSort(int arr[], int low,int high)
{
    int pivloc;

    if(low>=high)  return;

```

```
pivloc = partition(arr,low,high);  
QuickSort(arr,low,pivloc-1);  
QuickSort(arr,pivloc+1, high);  
}
```

```
public static void Bubble_sort(int arr[],int n)
{
    int i,j,t;
    int flag = 0;

    for(i=0;i<n;i++)
    {
        flag=0;
        for(j=0;j<n-i;j++)
        {
            if(arr[j] > arr[j+1])
            {
                flag=1;
                t=arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = t;
            }
        }
        if(flag==0)
            break;
        sysout("\n pass" + i +" ");
        display(arr,n);
    }
}
```

```
public static void Selection_sort(int arr[],int n)
{
    int i,j,t,min;

    for(i=0;i < n;i++)
    {
        min = i;
        for(j=i+1; j<=n ;j++)
        {
            if(arr[min] > arr[j]) min=j;
        }
        if(i!=min)
        {
            t=arr[i];
            arr[i]=arr[min];
            arr[min]=t;
        }
    }
}
```

```
public static void insertion_sort(int arr[],int n)
{
```

```

int i,j,k;

for(i=1;i<=n;i++)
{
    k=arr[i];

    for(j=i-1;j>=0 && k<arr[j];j--)
        arr[j+1] = arr[j];

    arr[j+1]=k;
}//end of outer loop

}



---


arr[10] = {12,23,44,56,67,78,88,99,-999};

i

brr[10] = {2,4,5,78,90,890,5678,-999};

j

res[20] = {2, 4, 5, 12, 23, 44,56, 67,78,88,90,99,890,5678,-999}

tar

public static void merge_Ver1(int arr[],int brr[],int res[])
{
    int i,j,tar;
    i=0; j=0; tar =0;

    while((arr[i]!=-999) && (brr[j]!=-999))
    {
        if(arr[i] < brr[j])
        {
            res[tar] = arr[i];
            i=i+1;
            tar=tar+1;
        }
        else if(brr[j] < arr[i])
        {
            res[tar] = brr[j];
            j=j+1;
            tar = tar+1;
        }
        else
        {
            res[tar]=arr[i];
            i++ ; j++; tar++;
        }
    }

    while(arr[i]!=-999)
    {
        res[tar] = arr[i];
        tar++; i++;
    }
}

```

```

while(brr[j] != -999)
{
    res[tar] = brr[j];
    tar++; j++;
}

```

```

public static void merge_Ver2(int arr[],int tar[],int low1,int high1,int
low2,int high2)
{
    int i,j,k;
    i=low1;
    j=low2;

    k=low1;

    while((i<=high1) && (j<=high2))
    {
        if(arr[i] < arr[j])
        {
            tar[k] = arr[i];
            i++; k++;
        }
        else if(arr[j] < arr[i])
        {
            tar[k] = arr[j];
            j++; k++;
        }
        else
        {
            tar[k] = arr[i];
            i++; j++; k++;
        }
    }
    while(i<=high1)
    {
        tar[k] = arr[i];
        i++; k++;
    }
    while(j <= high2)
    {
        tar[k] = arr[j];
        j++; k++;
    }
}//end of function

```

```

public static void merge_sort(int arr[], int low, int high)
{
    int mid;
    int temp[] = new int[arr.length];

    if(low < high)

```

```

{
    mid = (low + high) / 2;
    merge_sort(arr,low,mid);
    merge_sort(arr,mid+1,high);

    merge_Ver2(arr,temp,low,mid,mid+1,high);
    copy_arr(arr,temp,low,high);
}

public static void copy(int arr[], int temp[], int low,int high)
{
    int i;
    for(i=low; i<=high;i++)
        arr[i] = temp[i];
}

```

Quick sort

48 44 19 59 72 80 42 65 82 8 95 68

```

public static int partition(int arr[], int low, int high)
{
    int left, right, pivot, t;

    pivot=arr[low];
    left=low;
    right =high;

    while(left <= high)
    {
        while((arr[left]<=pivot) && (left<high))
            left++;

        while(arr[right] > pivot)
            right--;

        if(left < right)
        {
            t=arr[left];    arr[left] = arr[right] ; arr[right] = t;
            left++; right--;
        }
        else left++;
    }

    arr[low] = arr[right];
    arr[right] =pivot;

    return right;
}// end of function

```

```
public static void QuickSort(int arr[], int low,int high)
{
    int pivloc;

    if(low>=high)    return;

    pivloc = partition(arr,low,high);

    QuickSort(arr,low,pivloc-1);
    QuickSort(arr,pivloc+1, high);
}
```

Hashing, Hash Function and Hash Table

What is Hashing

- Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects.
- Examples of how hashing is used in our lives:
 - In universities, each student is assigned a unique roll number that can be used to retrieve information about them.
 - In libraries, each book is assigned a unique number that can be used to determine information about the book
- **Hashing** is the solution that can be used in almost all such situations and worst case complexity of hashing is still $O(n)$ (better than BST which is $O(\log n)$), but it gives $O(1)$ on the average.

What is Hashing (Cond...)

- Here we use ***hash function*** that converts a given input number or any other key to a smaller number and uses the small number as index in a table called ***hash table***.
- Hash Function
 - Hash function maps a big number or string to a small integer that can be used as index in hash table
- Hash Table
 - An array that stores pointers to records corresponding to a given input number

HashTable ADT

The common operations for hash table are:

- **CreatHashTable**: Creates a new hash table
- **HashSearch**: Searches the key in hash table
- **HashInsert**: Inserts a new key into hash table
- **HashDelete**: Deletes a key from hash table
- **DeleteHashTable**: Deletes the hash table

We can use array as a hash table.

For understanding the use of hash tables, let us consider the following example:

Give an algorithm for printing the first repeated character if there are duplicated elements in it. Let us think about the possible solutions.

The simple and brute force way of solving is: given a string, for each character check whether that character is repeated or not.

The time complexity of this approach is $O(n)$ with $O(1)$ space complexity.

Better Solution: worst case complexity = O(n)

```
char FirstRepeatedChar ( char *str ) {  
    int i, len=strlen(str);  
    int count[256]; // additional array  
    for(i=0; i<256; ++i)  
        count[i] = 0;  
    for(i=0; i<len; ++i) {  
        if(count[str[i]]==1) {  
            printf("%c", str[i]);  
            break;  
        }  
        else    count[str[i]]++;  
    }  
    if(i==len)  
        printf("No Repeated Characters");  
    return 0;  
}
```

- Using simple arrays is not the correct choice for solving the problems where the possible keys are very big
- Creating a huge array and storing the counters is not possible.
- That means there are a set of universal keys and limited locations in the memory.
- To solve this problem we need to somehow map all these possible keys to the possible memory locations.
- The process of mapping the keys to locations is called **hashing**.

Components of Hashing

Hashing has four key components:

- 1) Hash Table
- 2) Hash Functions
- 3) Collisions
- 4) Collision Resolution Techniques

Hash Table

A Hash-Table is a data structure, which maps keys to values.

It is an array of size M to store objects references.

Hash table is a generalization of array. With an array, we store the element whose key is k at a position k of the array

Hash Function

A hash function is a function, which generates an index in a table for a given object.

A hash function which generate a unique index for every object is called the perfect hash function.

Most simple hash function

```
unsigned int Hash(int key, int tableSize)//division method
{
    unsigned int hashValue = 0;
    hashValue = key;
    return hashValue % tableSize;
}
```

There are many hash functions, but this is the minimum that it should do. Various hash generation logics will be added to this function to generate a better hash.

Characteristics of Good Hash Functions

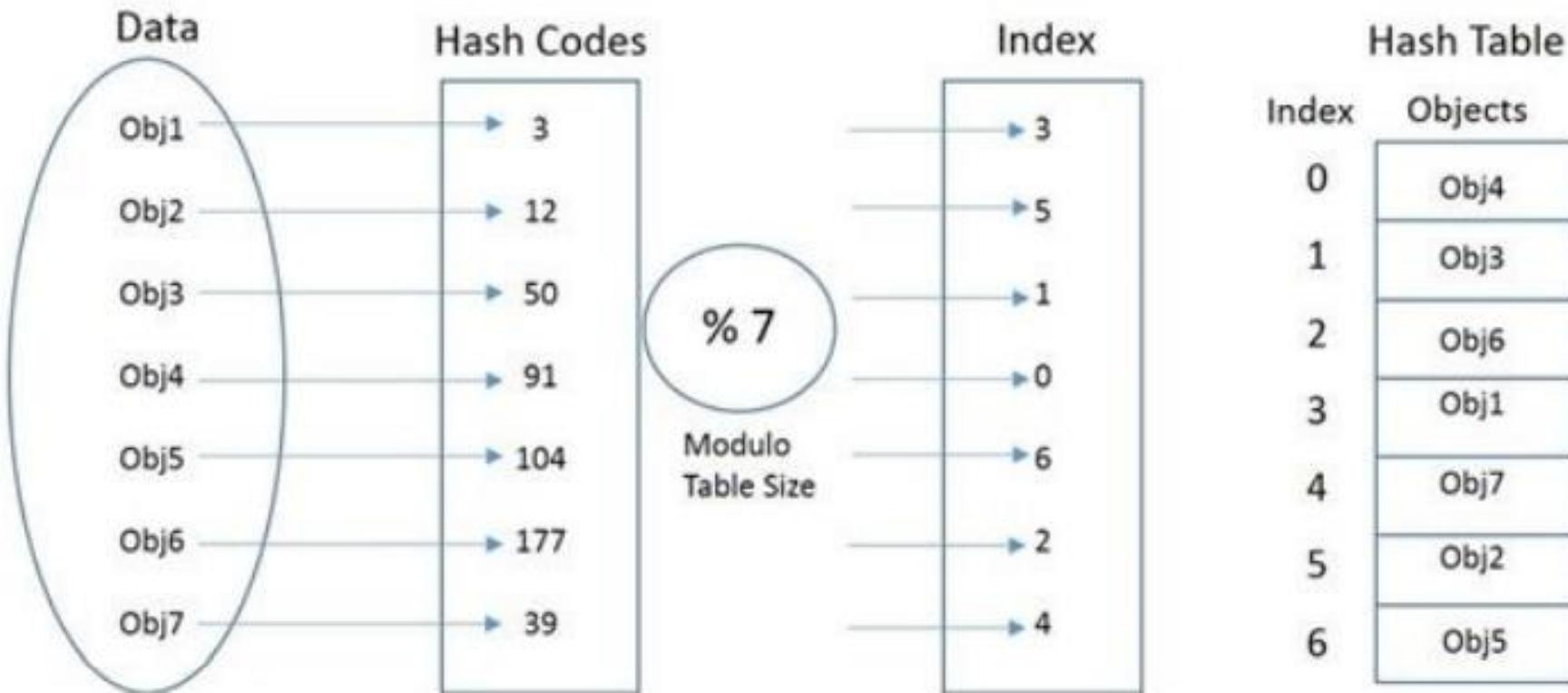
A good hash function should have the following characteristics:

- Minimize collision
- Be easy and quick to compute
- It should provide a uniform distribution of hash values
- Use all the information provided in the key
- Have a high load factor for a given set of keys

Load factor = Number of elements in Hash-Table / Hash-Table size

Hash Function

Object -----> Index

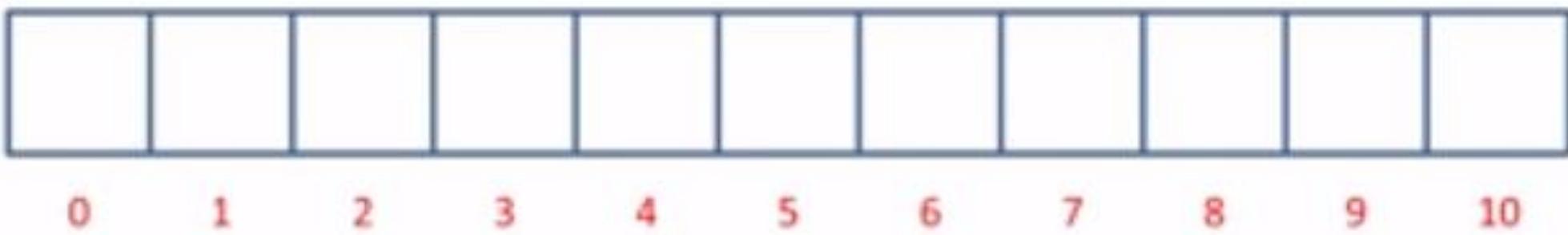


The process of **storing objects** using a hash function is as follows:

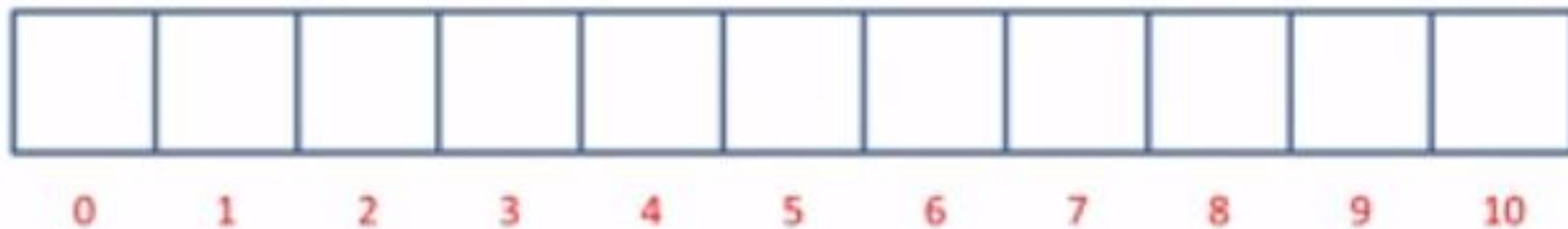
1. Create an array of size M to store objects, this array is called Hash-Table.
2. Find a hash code of an object by passing it through the hash function.
3. Take modulo of hash code by the size of Hash-table to get the index of the table where object will be stored.
4. Finally store the object in the designated index.

The process of **searching objects** in Hash-Table using a hash function is as follows:

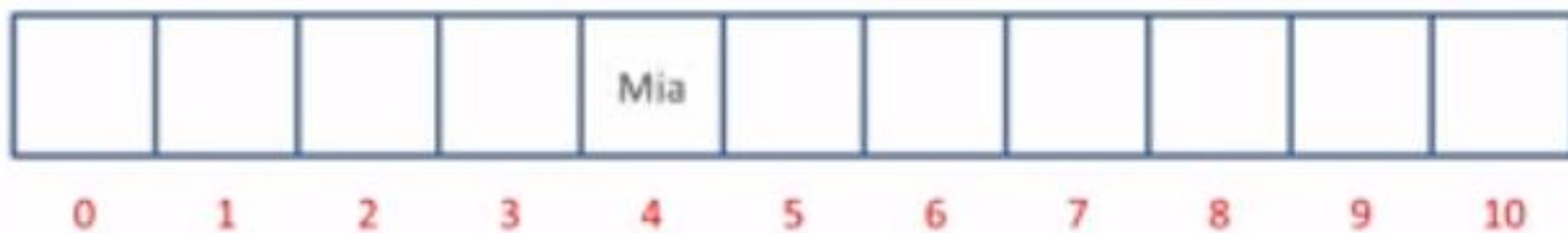
1. Find a hash code of the object we are searching for by passing it through the hash function.
2. Take modulo of hash code by the size of Hash-table to get the index of the table where objects are stored.
3. Finally, retrieve the object from the designated index



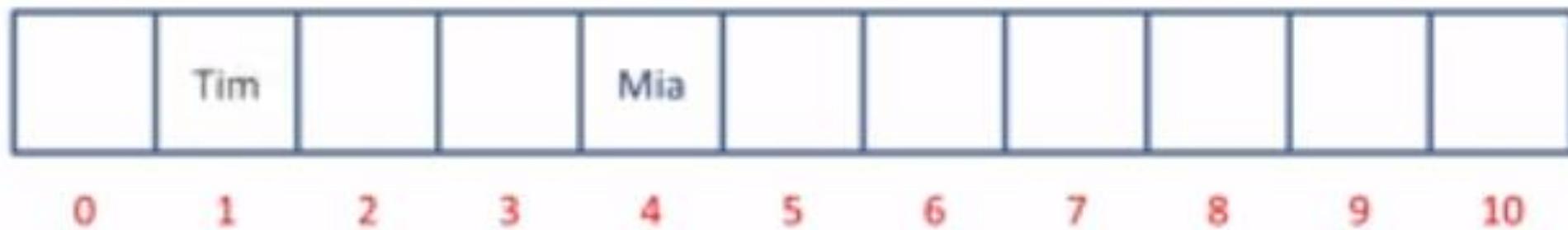
Mia M 77 i 105 a 97 279 4



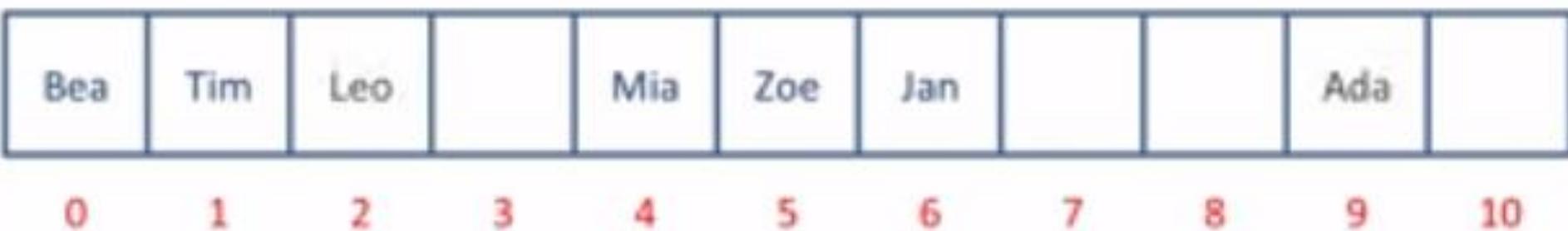
Mia M 77 i 105 a 97 279 4



| | | | | | | | | |
|-----|---|----|---|-----|---|-----|-----|---|
| Mia | M | 77 | i | 105 | a | 97 | 279 | 4 |
| Tim | T | 84 | i | 105 | m | 109 | 298 | 1 |



| | | | | | | | | |
|-----|---|----|---|-----|---|-----|-----|---|
| Mia | M | 77 | i | 105 | a | 97 | 279 | 4 |
| Tim | T | 84 | i | 105 | m | 109 | 298 | 1 |
| Bea | B | 66 | e | 101 | a | 97 | 264 | 0 |
| Zoe | Z | 90 | o | 111 | e | 101 | 302 | 5 |
| Jan | J | 74 | a | 97 | n | 110 | 281 | 6 |
| Ada | A | 65 | d | 100 | a | 97 | 262 | 9 |
| Leo | L | 76 | e | 101 | o | 111 | 288 | 2 |



| | | | | | | | | |
|-----|---|----|---|-----|---|-----|-----|----|
| Mia | M | 77 | i | 105 | a | 97 | 279 | 4 |
| Tim | T | 84 | i | 105 | m | 109 | 298 | 1 |
| Bea | B | 66 | e | 101 | a | 97 | 264 | 0 |
| Zoe | Z | 90 | o | 111 | e | 101 | 302 | 5 |
| Jan | J | 74 | a | 97 | n | 110 | 281 | 6 |
| Ada | A | 65 | d | 100 | a | 97 | 262 | 9 |
| Leo | L | 76 | e | 101 | o | 111 | 288 | 2 |
| Sam | S | 83 | a | 97 | m | 109 | 289 | 3 |
| Lou | L | 76 | o | 111 | u | 117 | 304 | 7 |
| Max | M | 77 | a | 97 | x | 120 | 294 | 8 |
| Ted | T | 84 | e | 101 | d | 100 | 285 | 10 |

| | | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Bea | Tim | Leo | Sam | Mia | Zoe | Jan | Lou | Max | Ada | Ted |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

0 1 2 3 4 5 6 7 8 9 10

Index number = sum ASCII codes Mod size of array

| | | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Bea | Tim | Leo | Sam | Mia | Zoe | Jan | Lou | Max | Ada | Ted |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

$$\text{Ada} = (65 + 100 + 97) = \textcolor{red}{262}$$

Find Ada

$$262 \bmod 11 = \textcolor{red}{9}$$

myData = Array(\textcolor{red}{9})

| | | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Bea | Tim | Leo | Sam | Mia | Zoe | Jan | Lou | Max | Ada | Ted |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

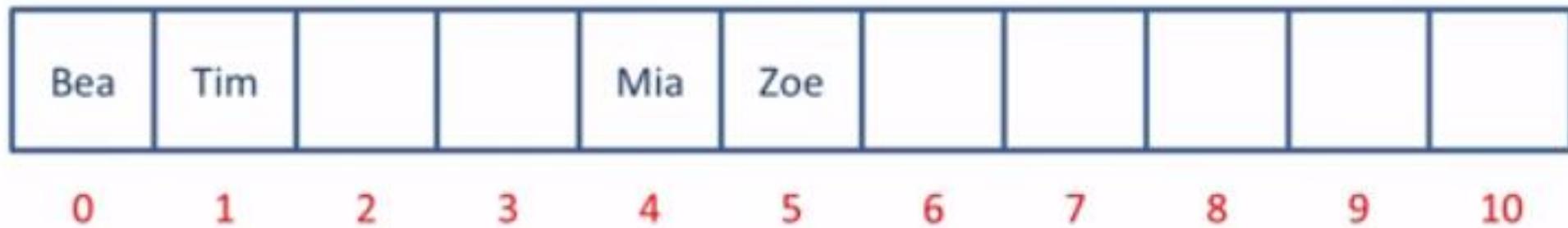
| | | | | | | | | | | |
|---|---|---|--|--|---|--|---|---|--|---|
| Bea 27/01/1942 English Astronomer | Tim 08/06/1955 English Inventor | Leo 31/12/1945 American Mathematician | Sam 27/04/1791 American Inventor | Mia 20/02/1988 Russian Space Station | Zoe 19/06/1978 American Actress | Jan 13/03/1956 Polish Logician | Lou 27/12/1822 French Biologist | Max 23/04/1858 German Physicist | Ada 10/12/1815 English Mathematician | Ted 17/06/1937 American Philosopher |
|---|---|---|--|--|---|--|---|---|--|---|

0 1 2 3 4 5 6 7 8 9 10

Understand Collision

| | | | | | | | | |
|-----|---|----|---|-----|---|-----|-----|---|
| Mia | M | 77 | i | 105 | a | 97 | 279 | 4 |
| Tim | T | 84 | i | 105 | m | 109 | 298 | 1 |
| Bea | B | 66 | e | 101 | a | 97 | 264 | 0 |
| Zoe | Z | 90 | o | 111 | e | 101 | 302 | 5 |

Collision is the condition where two records needs to be stored in the same location, which is not possible.



Understand Collision

| | | | | | | | | |
|-----|---|----|---|-----|---|-----|-----|---|
| Mia | M | 77 | i | 105 | a | 97 | 279 | 4 |
| Tim | T | 84 | i | 105 | m | 109 | 298 | 1 |
| Bea | B | 66 | e | 101 | a | 97 | 264 | 0 |
| Zoe | Z | 90 | o | 111 | e | 101 | 302 | 5 |
| Sue | S | 83 | u | 117 | e | 101 | 301 | 4 |



Collision Handling

- The process of finding an alternate location is called collision resolution
- Therefore, to maintain the performance of a hash table, it is important to manage collisions through various collision resolution techniques.
 - Open addressing
 - Linear Probing
 - Quadratic Probing
 - Double Hashing
 - Closed addressing

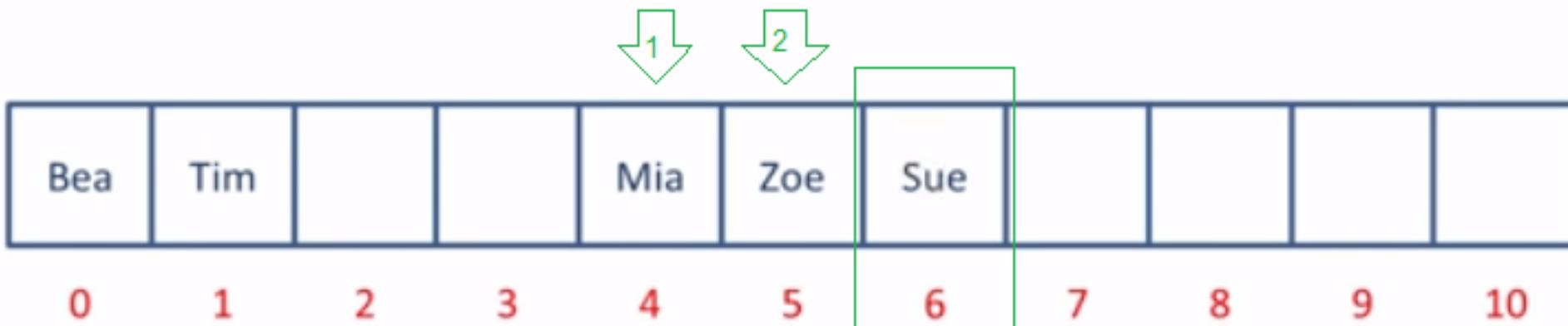
Linear Probing

- The function for rehashing is the following:
- $\text{rehash(key)} = (n + 1) \% \text{tablesize}$

One of the problems with linear probing is that table items tend to cluster together in the hash table. This means that the table contains groups of consecutively occupied locations that are called clustering.

Open Addressing (Linear)

| | | | | | | | | |
|-----|---|----|---|-----|---|-----|-----|---|
| Mia | M | 77 | i | 105 | a | 97 | 279 | 4 |
| Tim | T | 84 | i | 105 | m | 109 | 298 | 1 |
| Bea | B | 66 | e | 101 | a | 97 | 264 | 0 |
| Zoe | Z | 90 | o | 111 | e | 101 | 302 | 5 |
| Sue | S | 83 | u | 117 | e | 101 | 301 | 4 |



| | | | | | | | | |
|-----|---|----|---|-----|---|-----|-----|---|
| Mia | M | 77 | i | 105 | a | 97 | 279 | 4 |
| Tim | T | 84 | i | 105 | m | 109 | 298 | 1 |
| Bea | B | 66 | e | 101 | a | 97 | 264 | 0 |
| Zoe | Z | 90 | o | 111 | e | 101 | 302 | 5 |
| Sue | S | 83 | u | 117 | e | 101 | 301 | 4 |
| Len | L | 76 | e | 101 | n | 110 | 287 | 1 |



| | | | | | | | | |
|-----|---|----|---|-----|---|-----|-----|---|
| Mia | M | 77 | i | 105 | a | 97 | 279 | 4 |
| Tim | T | 84 | i | 105 | m | 109 | 298 | 1 |
| Bea | B | 66 | e | 101 | a | 97 | 264 | 0 |
| Zoe | Z | 90 | o | 111 | e | 101 | 302 | 5 |
| Sue | S | 83 | u | 117 | e | 101 | 301 | 4 |
| Len | L | 76 | e | 101 | n | 110 | 287 | 1 |
| Moe | M | 77 | o | 111 | e | 101 | 289 | 3 |
| Lou | L | 76 | o | 111 | u | 117 | 304 | 7 |
| Rae | R | 82 | a | 97 | e | 101 | 280 | 5 |
| Max | M | 77 | a | 97 | x | 120 | 294 | 8 |
| Tod | T | 84 | o | 111 | d | 100 | 295 | 9 |

| | | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Bea | Tim | Len | Moe | Mia | Zoe | Sue | Lou | Rae | Max | Tod |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

0 1 2 3 4 5 6 7 8 9 10

Find Rae

$$\text{Rae} = (82 + 97 + 101) = \textcolor{red}{280}$$

$$280 \bmod 11 = \textcolor{red}{5}$$

myData = Array(\textcolor{red}{5})

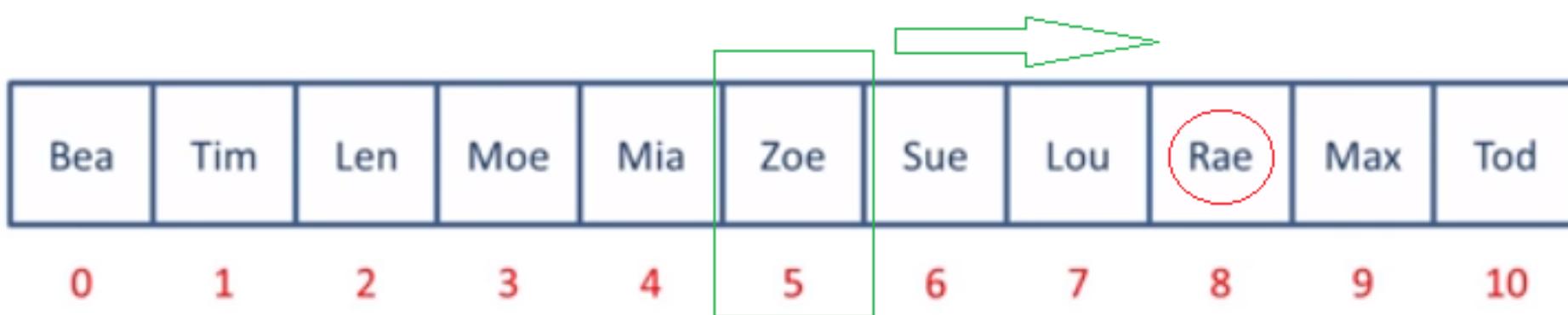
| | | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Bea | Tim | Len | Moe | Mia | Zoe | Sue | Lou | Rae | Max | Tod |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Find Rae

$$\text{Rae} = (82 + 97 + 101) = \textcolor{red}{280}$$

$$280 \bmod 11 = \textcolor{red}{5}$$

myData = Array(\textcolor{red}{5})



Quadratic Probing

- The interval between probes increases proportionally to the hash value
- The problem of Clustering can be eliminated if we use the quadratic probing method.
- In quadratic probing, we start from the original hash location i . If a location is occupied, we check the locations $i + 1^2, i + 2^2, i + 3^2, i + 4^2$
- The function for rehashing is the following:
$$\text{rehash(key)} = (n + k^2) \% \text{tablesize}$$

Double Hashing

- The interval between probes is computed by another hash function. Double hashing reduces clustering in a better way.

The second hash function h_2 should be: $h_2(\text{key}) \neq 0$ and $h_2 \neq h_1$

We first probe the location $h_1(\text{key})$.

If the location is occupied, we probe the location $h_1(\text{key}) + h_2(\text{key})$,
 $h_1(\text{key}) + 2 * h_2(\text{key})$, ...

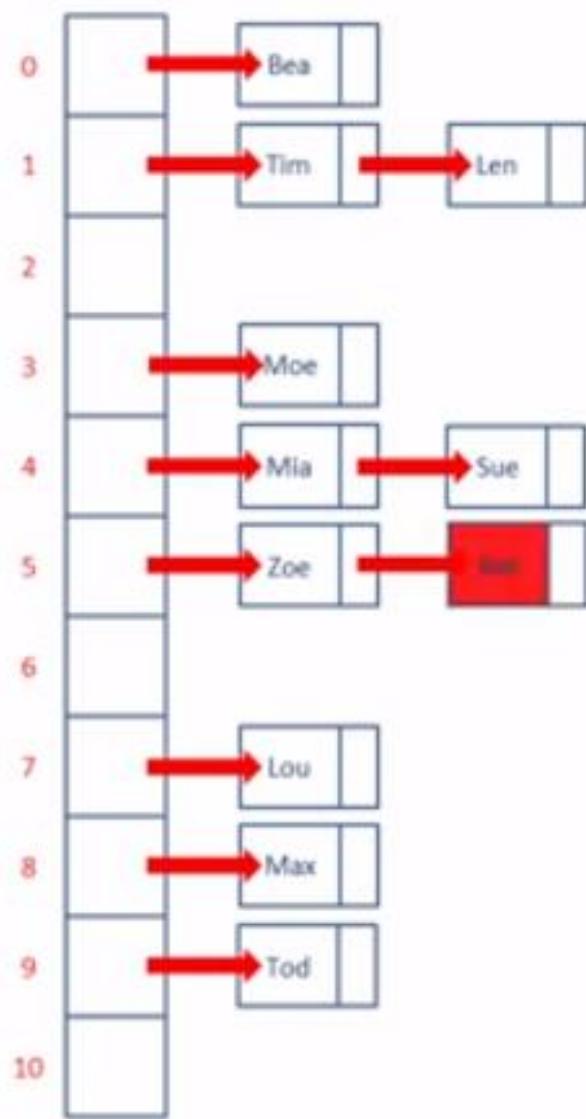
Example: Table size is 11 (0..10) Hash Function: assume $h_1(\text{key}) = \text{key} \bmod 11$ and $h_2(\text{key}) = 7 - (\text{key} \bmod 7)$

Closed Addressing (Non-Linear)





| | | | | | | | | |
|-----|---|----|---|-----|---|-----|-----|---|
| Mia | M | 77 | i | 105 | a | 97 | 279 | 4 |
| Tim | T | 84 | i | 105 | m | 109 | 298 | 1 |
| Bea | B | 66 | e | 101 | a | 97 | 264 | 0 |
| Zoe | Z | 90 | o | 111 | e | 101 | 302 | 5 |
| Sue | S | 83 | u | 117 | e | 101 | 301 | 4 |
| Len | L | 76 | e | 101 | n | 110 | 287 | 1 |
| Moe | M | 77 | o | 111 | e | 101 | 289 | 3 |
| Lou | L | 76 | o | 111 | u | 117 | 304 | 7 |



Find Rae $280 \bmod 11 = 5$

`myData = Array(5)`

Objectives of Hash Function

- Minimize collisions
- Uniform distribution of hash values
- Easy to calculate
- Resolve any collisions

```
package for_DSA;

class Employee {
    int empno;
    String ename;
    float sal;
    int age;
    public Employee(int empno, String ename, float sal, int age)
    {
        super();
        this.empno = empno;
        this.ename = ename;
        this.sal = sal;
        this.age = age;
    }
    public int getEmpno() {
        return empno;
    }
    public void setEmpno(int empno) {
        this.empno = empno;
    }
    public String getEname() {
        return ename;
    }
    public void setEname(String ename) {
        this.ename = ename;
    }
    public float getSal() {
        return sal;
    }
    public void setSal(float sal) {
        this.sal = sal;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
```

```

package for_DSA;

import java.util.Scanner;

class Hash_DoubleHashing{
    Record table[];

    public Hash_DoubleHashing()
    {
        table = new Record[10];
        for(int i=0;i<10;i++)
        {
            table[i] = new Record();
            table[i].status = type_of_record.EMPTY;
        }

        for(int i=0;i<10;i++)
            System.out.print(table[i].status);
    }
    public Hash_DoubleHashing(int size)
    {
        table = new Record[size];
        for(int i=0;i<size;i++)
            table[i].status = type_of_record.EMPTY;
    }

    public void insert(Employee emprec)
    {
        int i, location, h,h1;

        int key = emprec.getEmpno();           /*Extract key from
the record*/
        h = hash(key);

        location = h;
        h1 = hash1(key);

        for( i=1; i<=9; i++ )
        {
            if(table[location].status == type_of_record.EMPTY
|| table[location].status == type_of_record.DELETED)
            {
                table[location].info = emprec;
                table[location].status =
type_of_record.OCCUPIED;
                System.out.print("Record inserted\n\n");
                return;
            }
            if(table[location].info.getEmpno() == key)
            {
                System.out.print("Duplicate key\n\n");
            }
        }
    }
}

```

```

                    return;
                }
                location = (h+i*h1)%10;
            }
            System.out.print("Record can't be inserted : Table
overFlow\n\n");
        }
        int search(int key)
        {
            int i, h, location;
            h = hash(key);

            location = h;
            for( i=1; i<=9; i++ )
            {
                if( table[location].status ==
type_of_record.EMPTY )
                    return -1;
                if( table[location].info.getEmpno() == key)
                    return location;
                location = ( h + i ) % 10;

            }
            return -1;
        }
        public void del(int key)
        {
            int location = search(key);
            if(location == -1)
                System.out.print("Key not found\n");
            else
                table[location].status = type_of_record.DELETED;
        }/*End of del()*/
        public int hash(int key)
        {
            return (key%10);
        }

        public int hash1(int key)
        {
            return (7-key%7);
        }
        public void display()
        {
            int i;
            for(i=0; i<10; i++)
            {
                System.out.print("    "+i );
                if(table[i].status== type_of_record.OCCUPIED)
                {
                    System.out.print("Occupied
"+table[i].info.getEmpno()+"    "+table[i].info.getEname());
                }
            }
        }
    }
}

```

```

        System.out.print("    "+ table[i].info.age);
    }
    else if(table[i].status== type_of_record.DELETED)
        System.out.print("Deleted\n");
    else
        System.out.print("Empty\n");
}
}

public class Hash_DoubleHashing_Main {
    public static void main(String[] args) {
        Employee e;
        int eno,age;
        String name;
        float s;
        int i;
        Scanner sc=new Scanner(System.in);
        Hash_DoubleHashing hl = new Hash_DoubleHashing();
        for(i=0;i<5;i++)
        {
            System.out.println("\n Enter empno name salary
age of employee: ");
            eno=sc.nextInt();
            name = sc.next();
            age=sc.nextInt();
            s=sc.nextFloat();
            e = new Employee(eno,name,s,age);
            hl.insert(e);
        }
        hl.display();
        System.out.println("search for key
="+hl.search(9889));
        hl.del(9889);
        hl.display();
        hl.insert(new Employee(9889,"Palu",56000f,56));
        System.out.println("\n\n");
        hl.display();
        sc.close();
    }
}

```

```

package for_DSA;

import java.util.Scanner;

enum type_of_record {EMPTY, DELETED,OCCUPIED};

class Record {
    Employee info;
    type_of_record status;
}

class Hash_LinearProbing{
    Record table[];

    public Hash_LinearProbing()
    {
        table = new Record[10];
        for(int i=0;i<10;i++)
        {
            table[i] = new Record();
            table[i].status = type_of_record.EMPTY;
        }

        for(int i=0;i<10;i++)
            System.out.print(" "+table[i].status);
    }
    public Hash_LinearProbing(int size)
    {
        table = new Record[size];
        for(int i=0;i<size;i++)
            table[i].status = type_of_record.EMPTY;
    }

    public void insert(Employee emprec)
    {
        int i, location, h;

        int key = emprec.getEmpno();           /*Extract key from
the record*/
        h = hash(key);

        location = h;
        for( i=1; i<=9; i++ )
        {
            if(table[location].status == type_of_record.EMPTY
|| table[location].status == type_of_record.DELETED)
            {
                table[location].info = emprec;
                table[location].status =
type_of_record.OCCUPIED;
                System.out.print("Record inserted\n\n");
            }
        }
    }
}

```

```

                return;
            }
            if(table[location].info.getEmpno() == key)
            {
                System.out.print("Duplicate key\n\n");
                return;
            }
            location = ( h + i) % 10;
        }
        System.out.print("Record can't be inserted : Table
overflow\n\n");
    }
    int search(int key)
    {
        int i, h, location;
        h = hash(key);

        location = h;
        for( i=1; i<=9; i++ )
        {
            if( table[location].status ==
type_of_record.EMPTY )
                return -1;
            if( table[location].info.getEmpno() == key)
                return location;
            location = ( h + i ) % 10;

        }
        return -1;
    }
    public void del(int key)
    {
        int location = search(key);
        if(location == -1)
            System.out.print("Key not found\n");
        else
            table[location].status = type_of_record.DELETED;
    }/*End of del()*/
    public int hash(int key)
    {
        return (key%10);
    }

    public void display()
    {
        int i;
        for(i=0; i<10; i++)
        {
            System.out.print("    "+i );
            if(table[i].status== type_of_record.OCCUPIED)
            {
                System.out.print("Occupied
");
            }
        }
    }
}

```

```

"+table[i].info.getEmpno()+"    "+table[i].info.getEname());
                                System.out.print("    "+ table[i].info.age);
}
else if(table[i].status== type_of_record.DELETED)
    System.out.print("Deleted\n");
else
    System.out.print("Empty\n");
}
}
}

public class Hash_LinearProbing_Main {
    public static void main(String[] args) {
        Employee e;
        int eno,age;
        String name;
        float s;
        int i;
        Scanner sc=new Scanner(System.in);
        Hash_LinearProbing hl = new Hash_LinearProbing();
        for(i=0;i<5;i++)
        {
            System.out.println("\n Enter empno name salary
age of employee: ");
            eno=sc.nextInt();
            name = sc.next();
            age=sc.nextInt();
            s=sc.nextFloat();
            e = new Employee(eno,name,s,age);
            hl.insert(e);
        }
        hl.display();
        System.out.println("search for key
="+hl.search(9889));

        hl.del(9889);
        hl.display();
        hl.insert(new Employee(9889,"Palu",56000f,56));
        System.out.println("\n\n");
        hl.display();
        sc.close();
    }
}

```

```

package for_DSA;

import java.util.Scanner;

class Hash_QuadrticProbing{
    Record table[];

    public Hash_QuadrticProbing()
    {
        table = new Record[10];
        for(int i=0;i<10;i++)
        {
            table[i] = new Record();
            table[i].status = type_of_record.EMPTY;
        }

        for(int i=0;i<10;i++)
            System.out.print(" "+table[i].status);
    }
    public Hash_QuadrticProbing(int size)
    {
        table = new Record[size];
        for(int i=0;i<size;i++)
        {
            table[i] = new Record();
            table[i].status = type_of_record.EMPTY;
        }
    }

    public void insert(Employee emprec)
    {
        int i, location, h;

        int key = emprec.getEmpno();           /*Extract key from
the record*/
        h = hash(key);

        location = h;
        for( i=1; i<=9; i++ )
        {
            if(table[location].status == type_of_record.EMPTY
            || table[location].status == type_of_record.DELETED)
            {
                table[location].info = emprec;
                table[location].status =
type_of_record.OCCUPIED;
                System.out.print("Record inserted\n\n");
                return;
            }
            if(table[location].info.getEmpno() == key)
            {

```

```

        System.out.print("Duplicate key\n\n");
        return;
    }
    location = ( h + i*i) % 10;
}
System.out.print("Record can't be inserted : Table
overFlow\n\n");
}
int search(int key)
{
    int i, h, location;
    h = hash(key);

    location = h;
    for( i=1; i<=9; i++ )
    {
        if( table[location].status ==
type_of_record.EMPTY )
            return -1;
        if( table[location].info.getEmpno() == key)
            return location;
        location = ( h + i*i ) % 10;

    }
    return -1;
}
public void del(int key)
{
    int location = search(key);
    if(location == -1)
        System.out.print("Key not found\n");
    else
        table[location].status = type_of_record.DELETED;
}/*End of del()*/
public int hash(int key)
{
    return (key%10);
}

public void display()
{
    int i;
    for(i=0; i<10; i++)
    {
        System.out.print("    "+i );
        if(table[i].status== type_of_record.OCCUPIED)
        {
            System.out.print("Occupied
"+table[i].info.getEmpno()+"    "+table[i].info.getEname());
            System.out.print("    "+ table[i].info.age);
        }
    }
}

```

```

        else if(table[i].status== type_of_record.DELETED)
            System.out.print("Deleted\n");
        else
            System.out.print("Empty\n");
    }
}
}

public class Hash_QuadrticProbingMain {

    public static void main(String[] args) {
        Employee e;
        int eno,age;
        String name;
        float s;
        int i;
        Scanner sc=new Scanner(System.in);
        Hash_QuadrticProbing hl = new Hash_QuadrticProbing();
        for(i=0;i<5;i++)
        {
            System.out.println("\n Enter empno name salary
age of employee: ");
            eno=sc.nextInt();
            name = sc.next();
            age=sc.nextInt();
            s=sc.nextFloat();
            e = new Employee(eno,name,s,age);
            hl.insert(e);
        }
        hl.display();
        System.out.println("search for key
="+hl.search(9889));

        hl.del(9889);
        hl.display();
        hl.insert(new Employee(9889,"Palu",56000f,56));
        System.out.println("\n\n");
        hl.display();
        sc.close();
    }
}

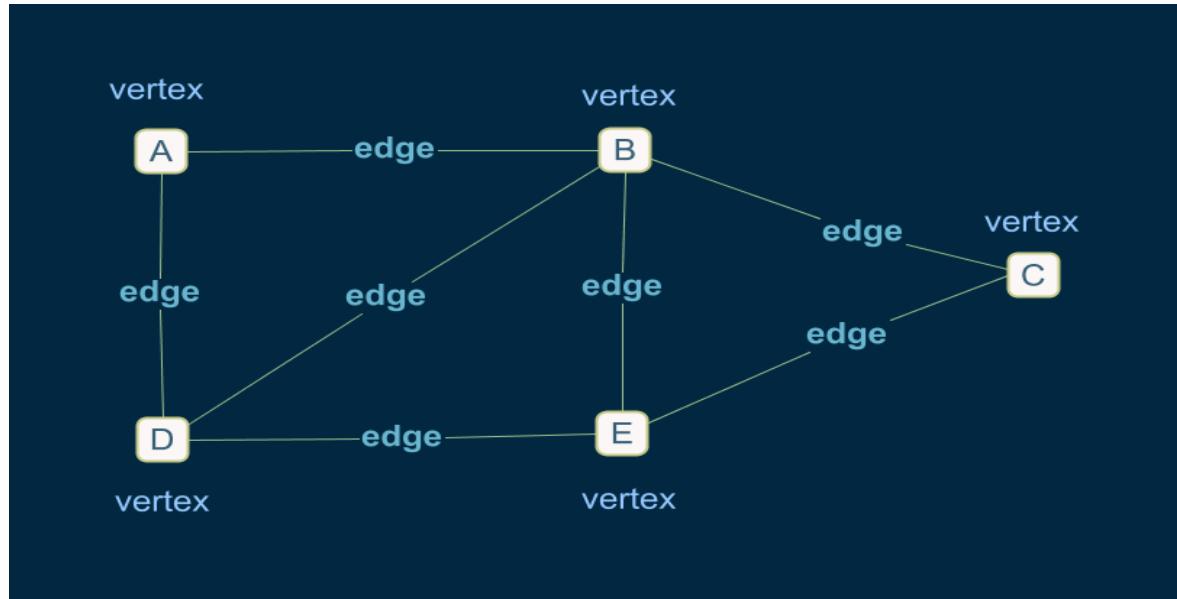
```

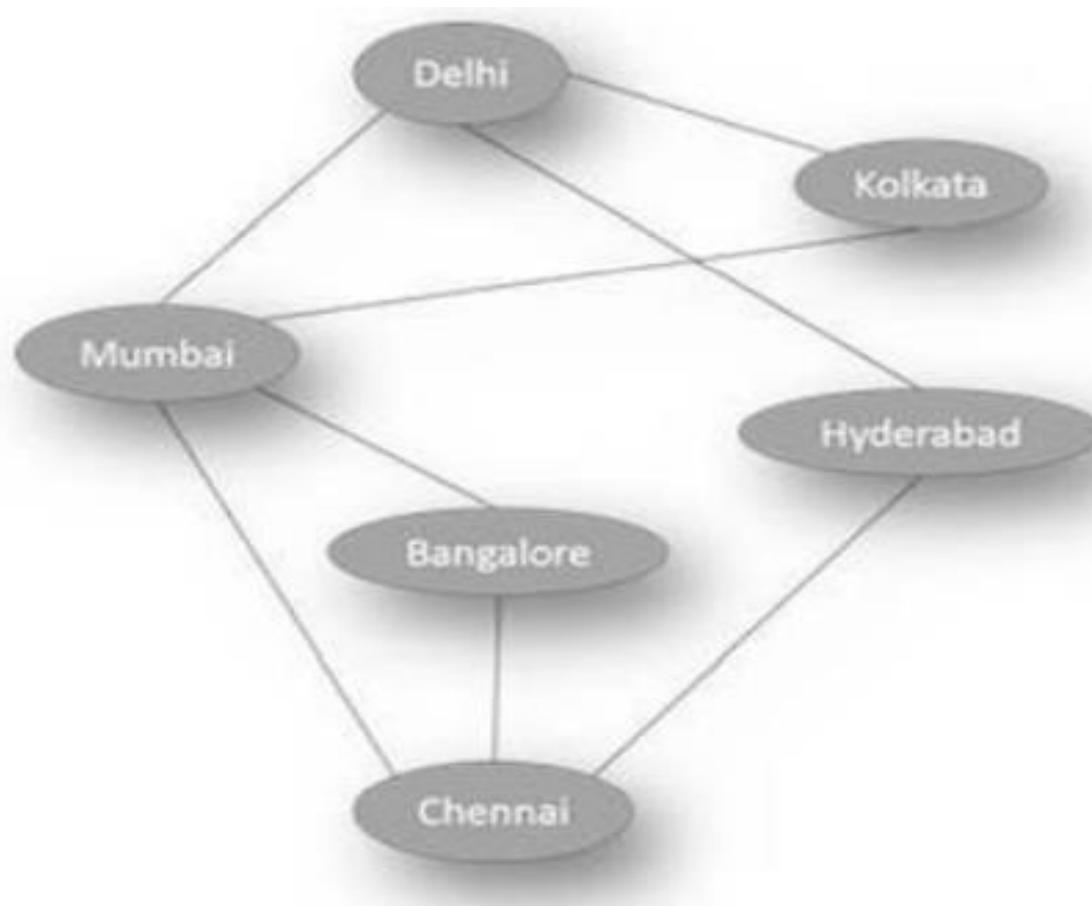
Graphs

What is a graph?

A Graph is represented by G where $G = (V, E)$, where V is a finite set of points called Vertices and E is a finite set of Edges.

Each edge is a connecting line of two vertices and is denoted by a pair (u, v) where u, v belongs to set of vertices V .





A graph can be of 2 types

Formal definition of graphs

A graph G is defined as follows:

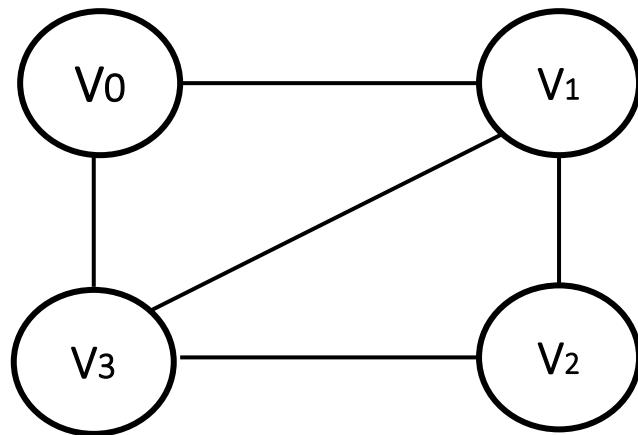
$$G=(V,E)$$

$V(G)$: a finite, nonempty set of vertices

$E(G)$: a set of edges (pairs of vertices)

Undirected graphs

When the edges in a graph have no direction, the graph is called *undirected*. The graph below has 4 vertices and 5 edges

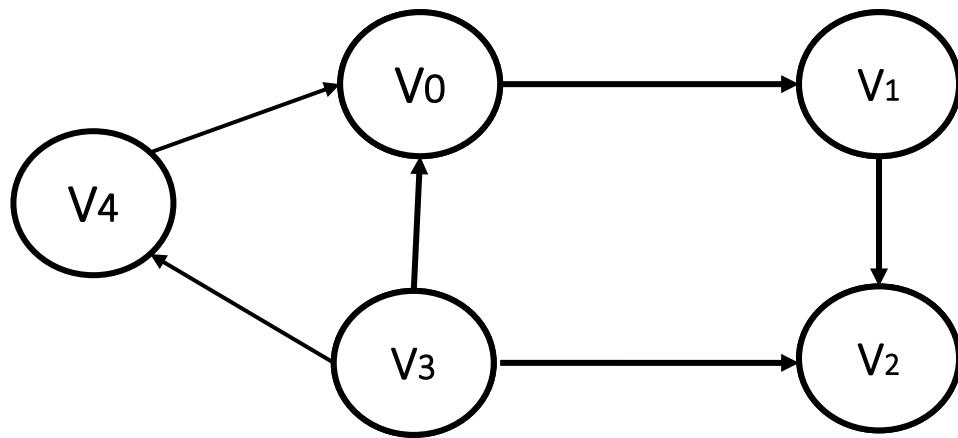


$$V(G) = \{ V_0, V_1, V_2, V_3 \}$$

$$E(G) = \{ (V_0, V_1), (V_0, V_3), (V_1, V_2), (V_1, V_3), (V_2, V_3) \}$$

Directed graphs

When the edges in a graph have a direction, the graph is called *directed* (or *digraph*). The graph below has 5 vertices and 6 edges



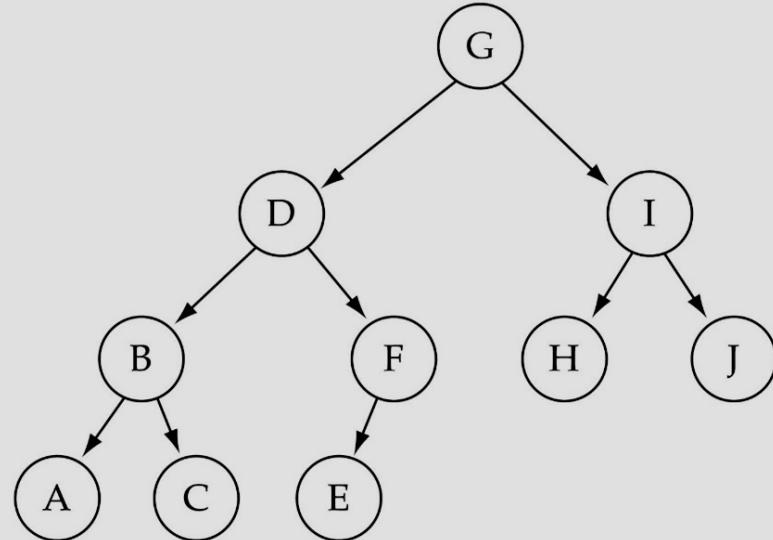
$$V(G) = \{ V_0, V_1, V_2, V_3, V_4 \}$$

$$E(G) = \{ (V_0, V_1), (V_1, V_2), (V_3, V_2), (V_3, V_0), (V_3, V_4), (V_4, V_0), \}$$

Trees vs graphs

Trees are special cases of graphs!!

(c) Graph3 is a directed graph.

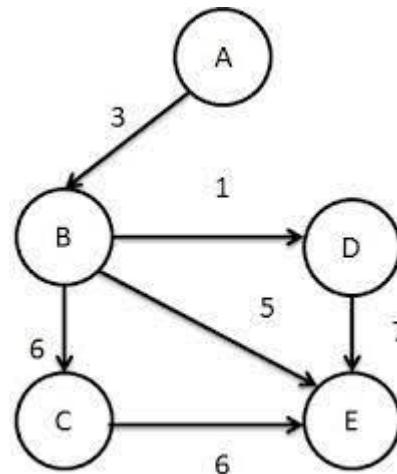
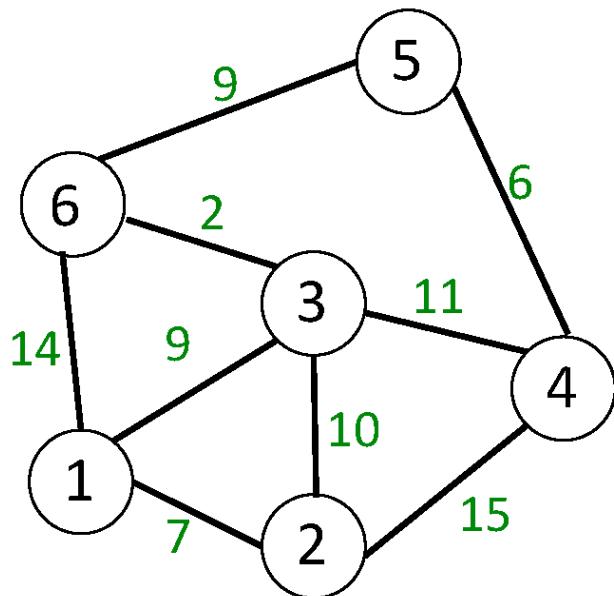


$$V(\text{Graph3}) = \{ A, B, C, D, E, F, G, H, I, J \}$$

$$E(\text{Graph3}) = \{ (G, D), (G, I), (D, B), (D, F), (I, H), (I, J), (B, A), (B, C), (F, E) \}$$

Graph terminology

Weighted Graph: A graph is weighted if its edges have been assigned a non negative value as weight. The weight on the edge may represent cost, length or distance associated with the edge.



Graph terminology

Subgraph : A graph H is said to be subgraph of another graph G, if the vertex set of H is subset of vertex set of G and edge set is H is subset of edge set of G

Adjacency: Adjacency is a relation between two vertices of a graph. A vertex v is adjacent to another vertex u if there is an edge from vertex u to vertex v.

In an undirected graph, if we have edge (u,v) , then u is adjacent to v and v is adjacent to u. So adjacency relation is symmetric in undirected graph.

Graph terminology

In directed graph, if we have edge (u,v) , then u is adjacent to v , and v is adjacent from u .

Incidence: Incidence is a relation between vertex and an edge of a graph.

In an undirected graph the edge (u,v) is incident on vertices u and v .

In directed graph the edge (u,v) is incident from vertex u and incident to vertex v .

Graph terminology

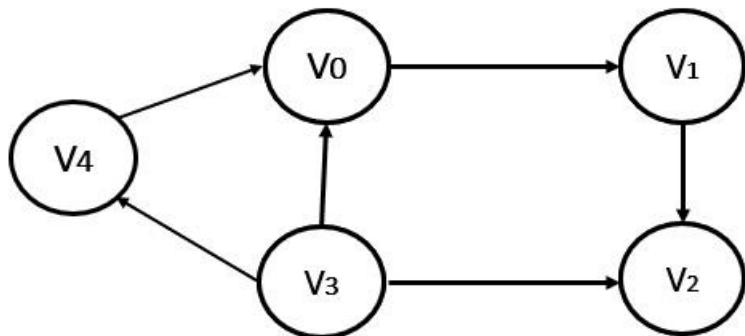
Path: A Path is a sequence of edges between two vertices.

In graph below,

$V_3 \rightarrow V_4 \rightarrow V_0 \rightarrow V_1 \rightarrow V_2$ is a path

Length of a Path: The length of a Path is the total number of edges included in the path.

Reachable: If there is Path P from vertex u to vertex v, then vertex v is said to be reachable from vertex u via path P. For e.g V_2 is reachable from V_3 , but V_3 is not reachable from V_4

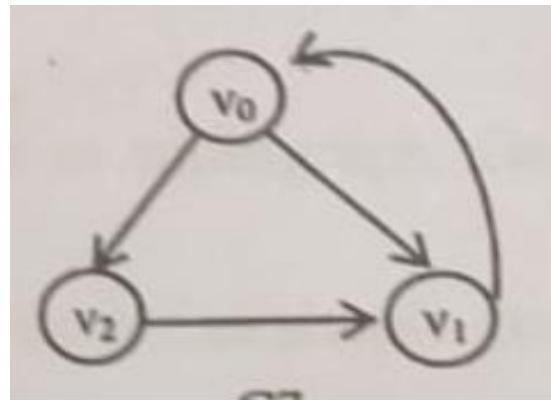


Graph terminology

Simple Path: A path in which all vertices are distinct

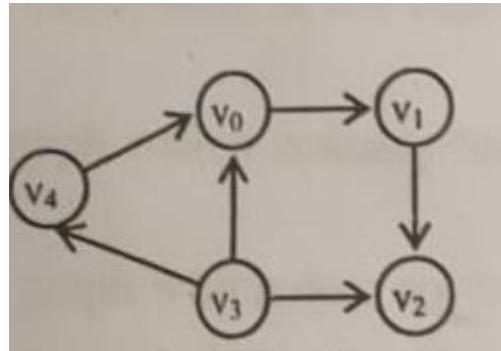
Cycle: A Cycle is a path that starts and ends at the same vertex and include at least one vertex. There has to be at least 3 vertices to form a cycle in undirected graph

Cyclic graph: A graph that has one or more cycles is called a cyclic graph.



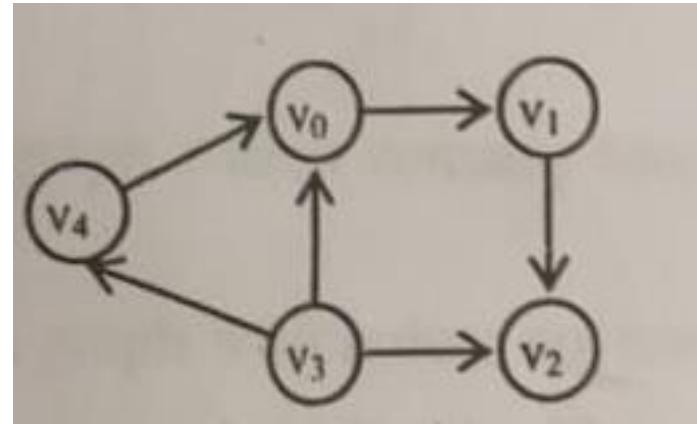
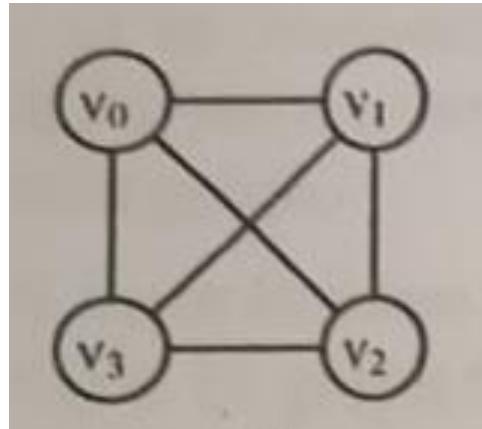
Graph terminology

Acyclic graph: A graph that has no cycle is called Acyclic graph.



DAG: A directed acyclic graph is named as DAG.

Degree: In an undirected graph, the degree of a vertex is the number of edges incident on it.

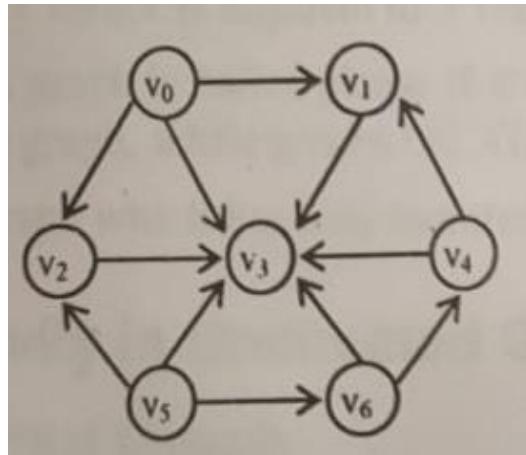


In directed graph, each vertex has an indegree and an outdegree. The degree of each vertex in directed graph is sum of its indegree and outdegree.

Indegree of a vertex is number of edges entering into vertex
Outdegree of a vertex is number of edges leaving the vertex.

Graph terminology

Source: A vertex, which has no incoming edge, but has outgoing edges is called as a source



Vertex V0 and V5 are source

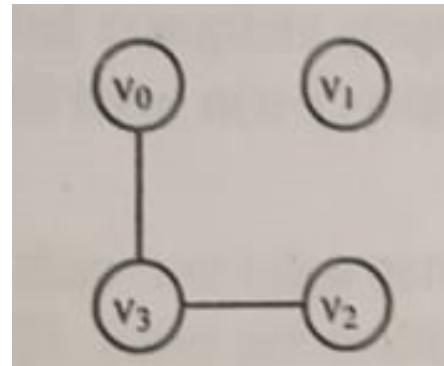
Sink: A vertex, which has no outgoing edge but has incoming edges is called as a sink

Vertex V3 is sink

Graph terminology

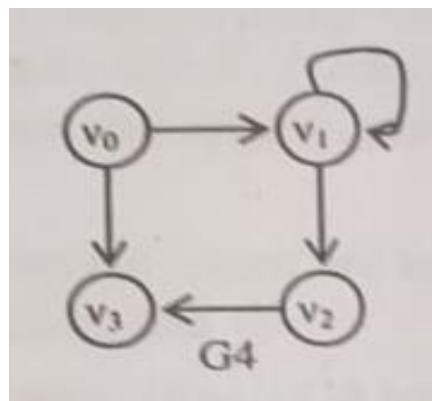
Pendent vertex: A vertex in a digraph is pendent if its indegree is 1 and outdegree is 0

Isolated vertex: If the degree of a vertex is 0, then it is called as Isolated vertex



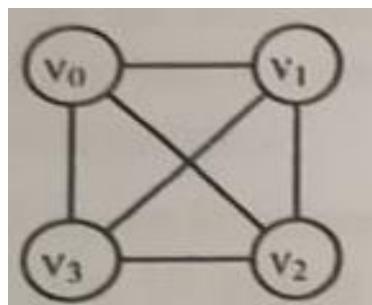
v1 is isolated

Loop: A self loop is an edge that connects a vertex to itself.

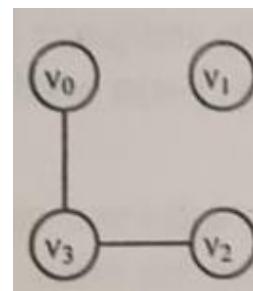


Loop at V1

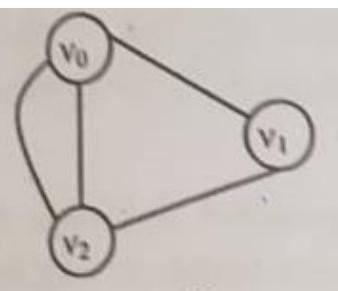
Planar graph: A graph which can drawn on plane paper without any two edges intersecting each other.



Not a planar

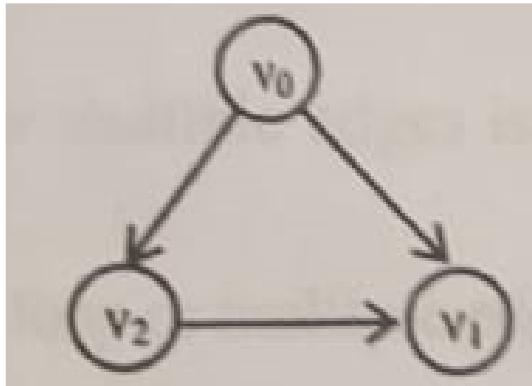


planar



Graph terminology

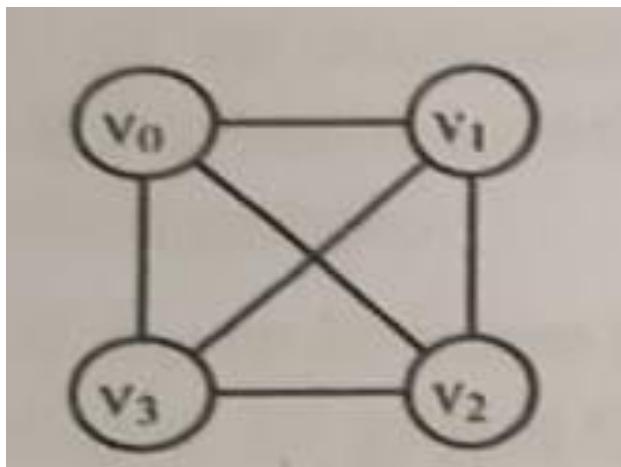
Successor and predecessor:



v0 is predecessor of v1
and v1 is successor of v0

Graph terminology

Regular graph: A graph in which each vertex is adjacent to the same number of vertices.



Every vertex is adjacent to 3 vertices

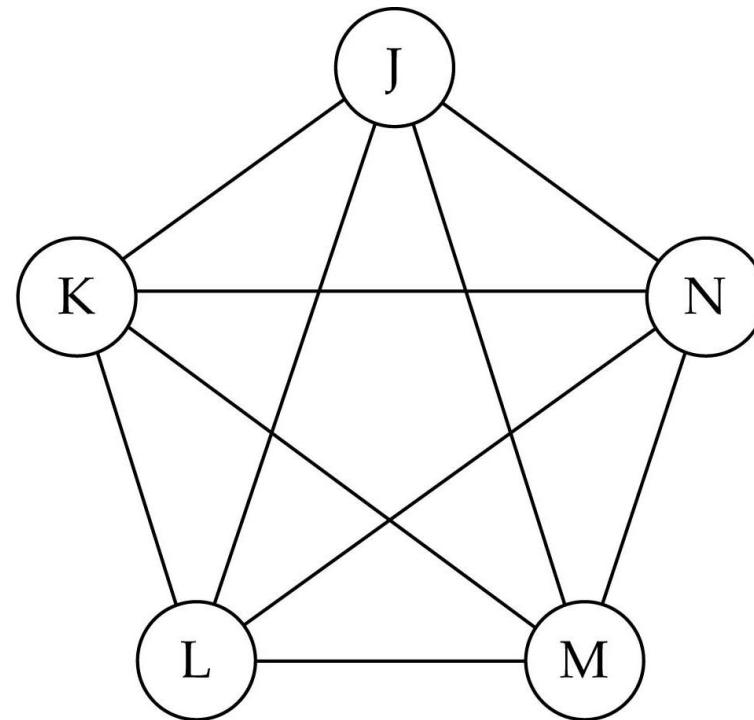
Null graph: A graph which has only isolated vertices is called Null graph.

Graph terminology

What is the number of edges in a complete undirected graph with N vertices?

$$N * (N-1) / 2$$

$$O(N^2)$$



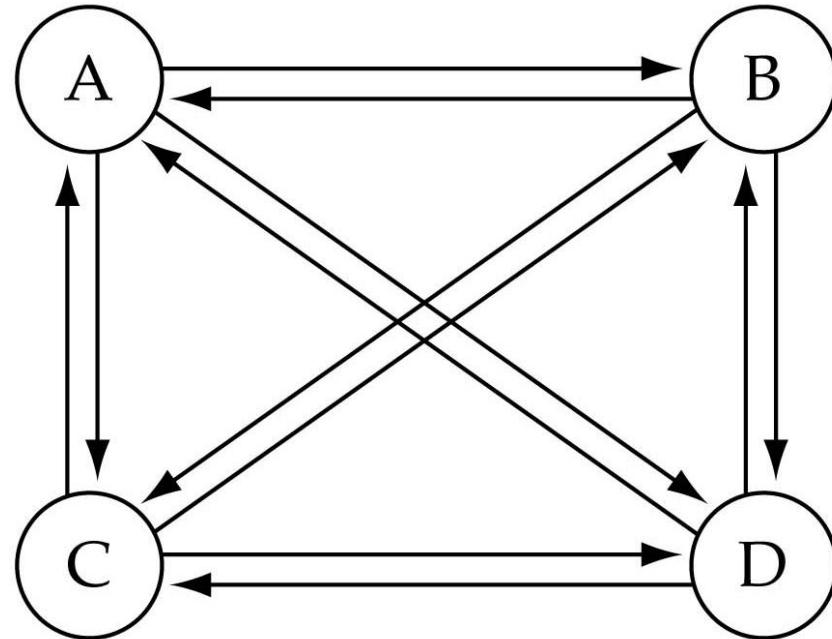
(b) Complete undirected graph.

Graph terminology (cont.)

- ❑ What is the number of edges in a complete directed graph with N vertices?

$$N * (N-1)$$

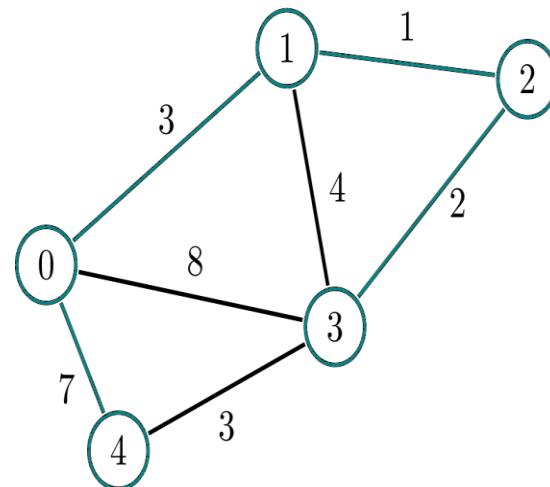
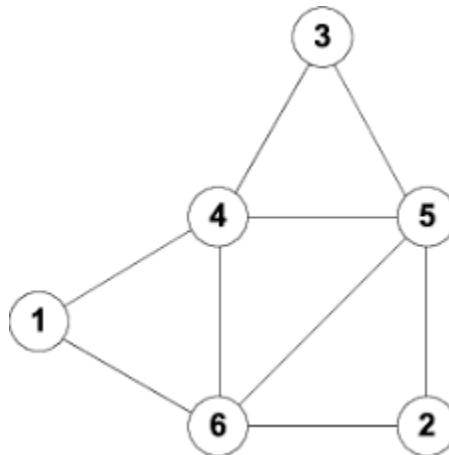
$O(N^2)$



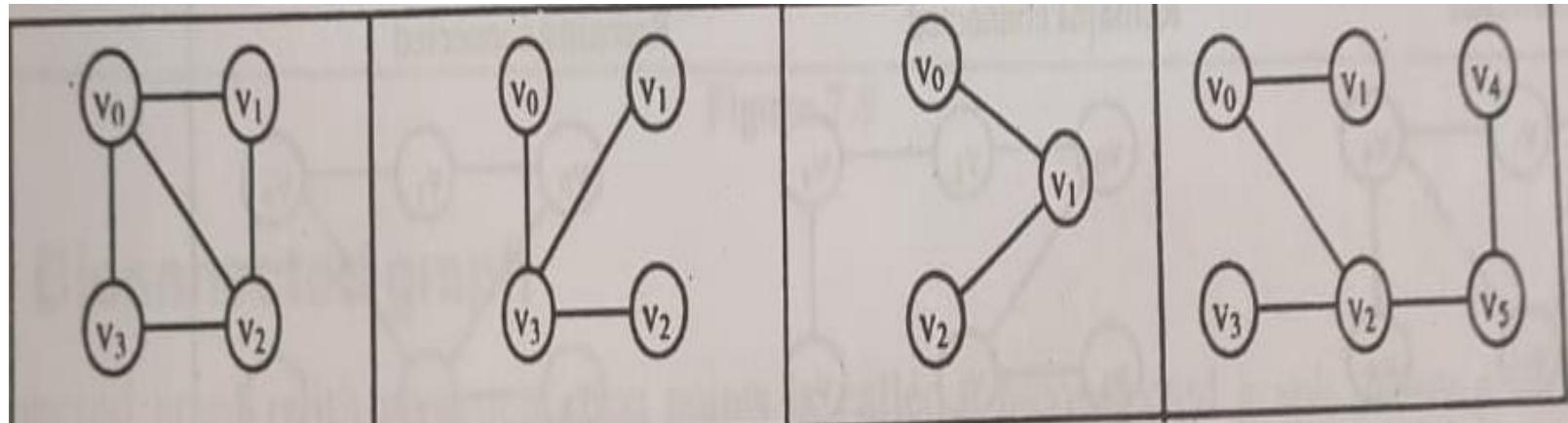
(a) Complete directed graph.

Graph terminology (cont.)

Connected graph: An undirected graph is said to be connected if there is a path from any vertex to any other vertex, or any vertex is reachable from any other vertex.

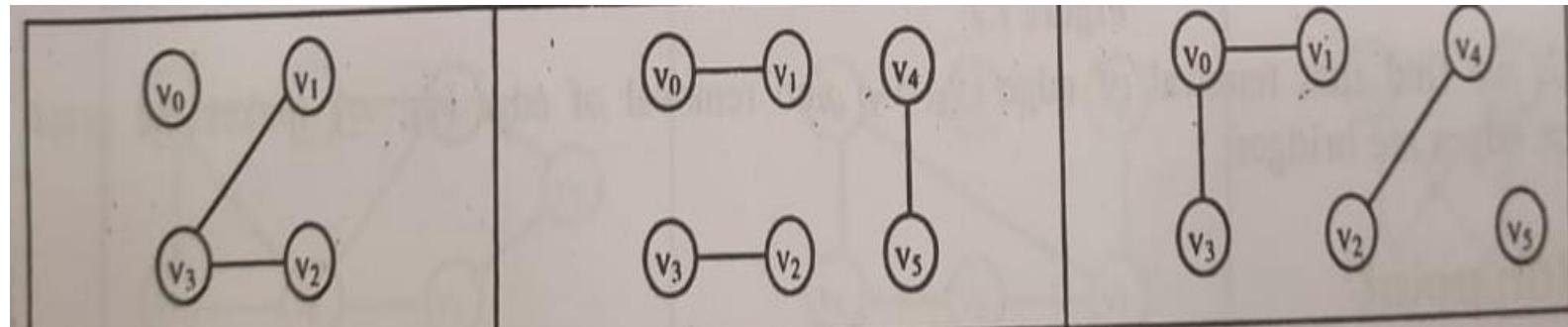
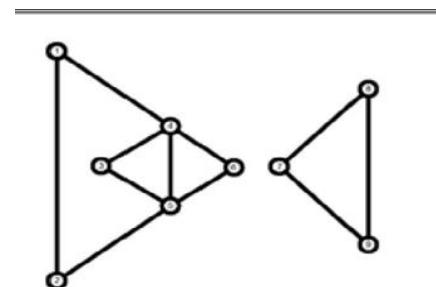
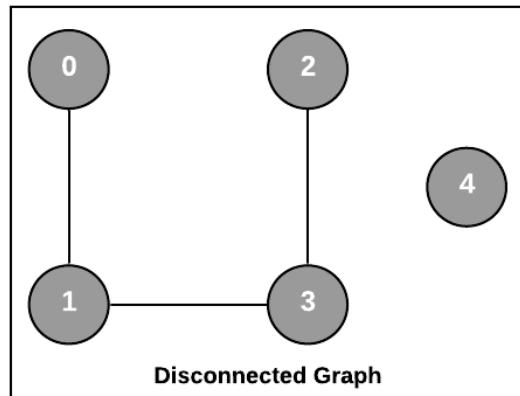


Graph terminology (cont.)



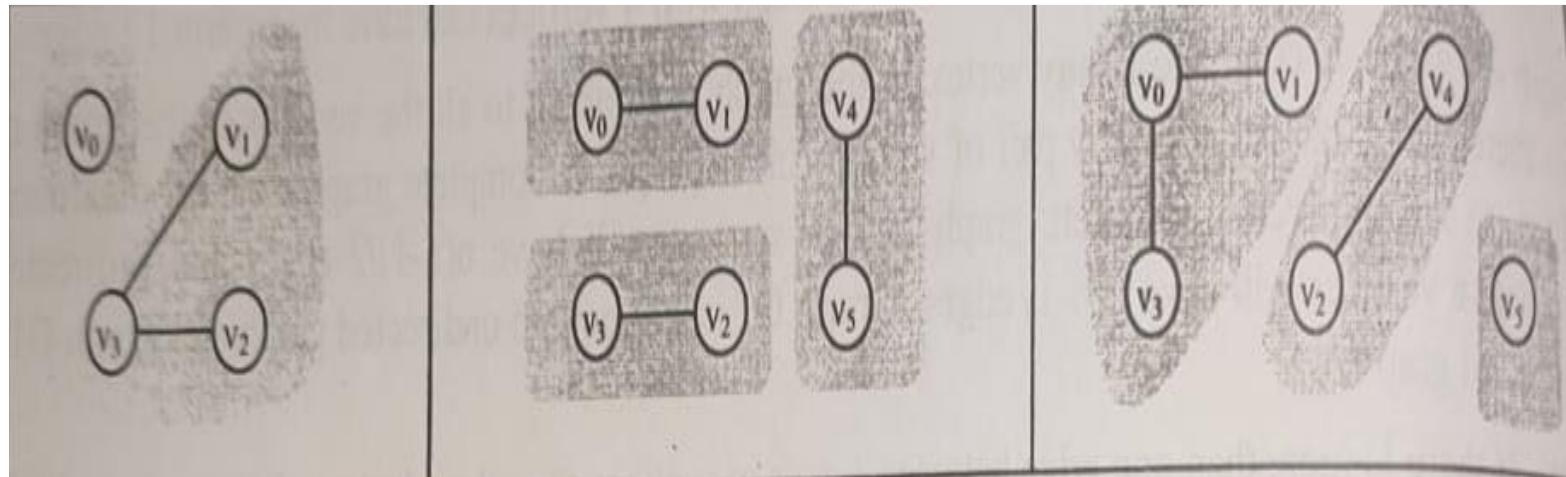
Graph terminology (cont.)

Disconnected undirected graph:

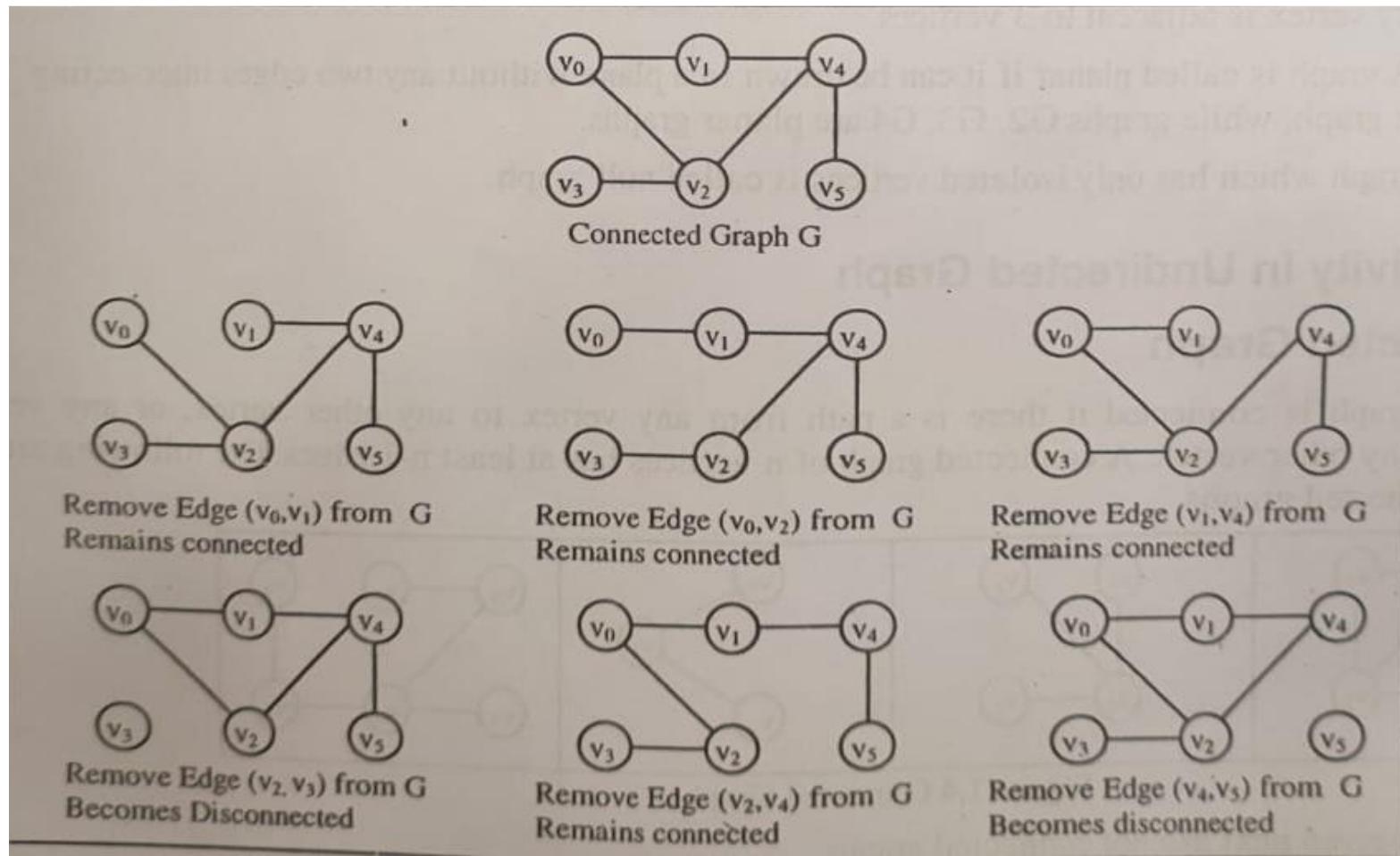


Graph terminology (cont.)

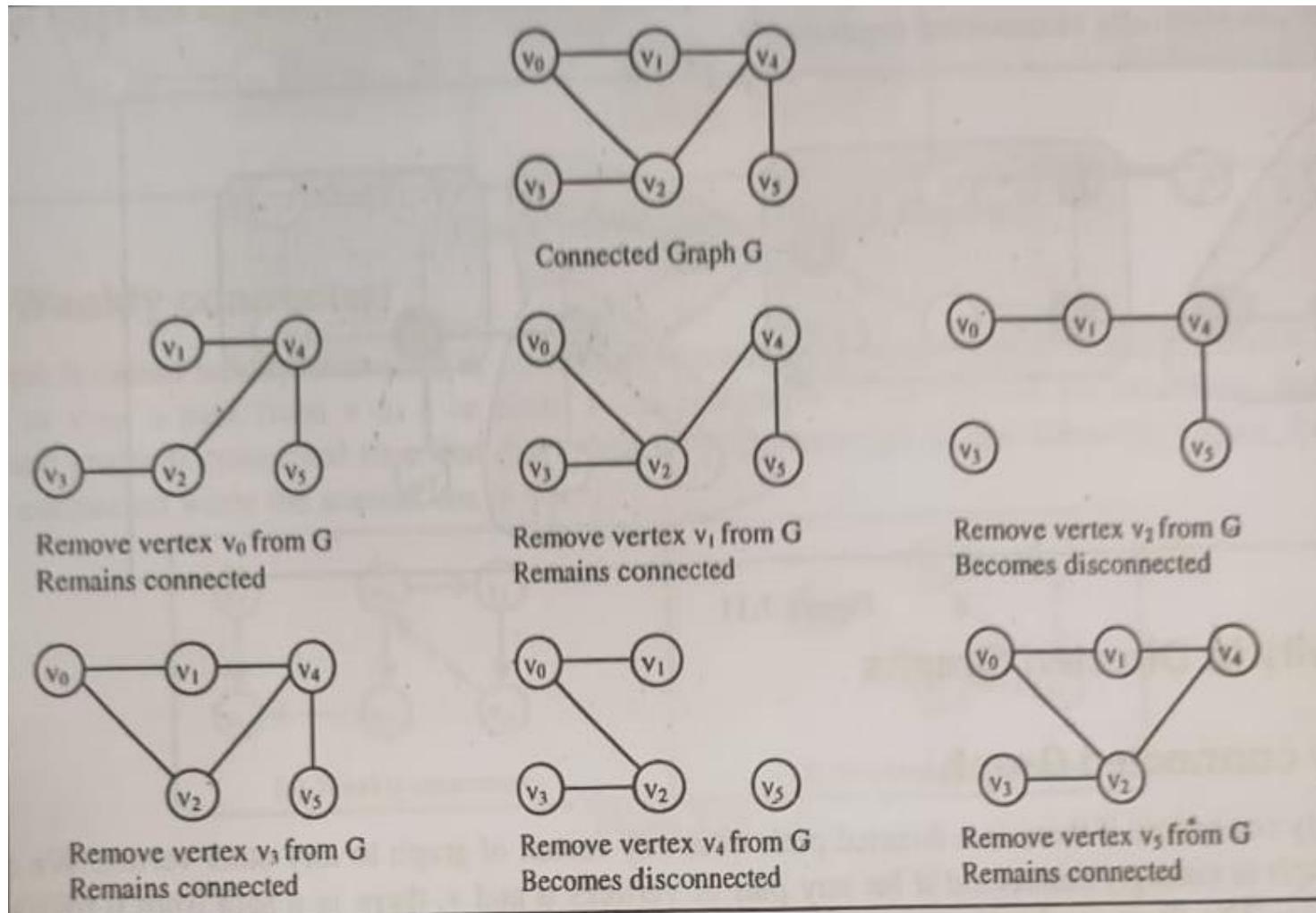
Connected components: An undirected graph which is not connected may have different parts of the graph which are connected. These parts are called as connected components.



Bridge: If on removing an edge from a connected graph, the graph becomes disconnected then that edge is called as bridge. Consider graph below and remove edge one by one and identify bridge.

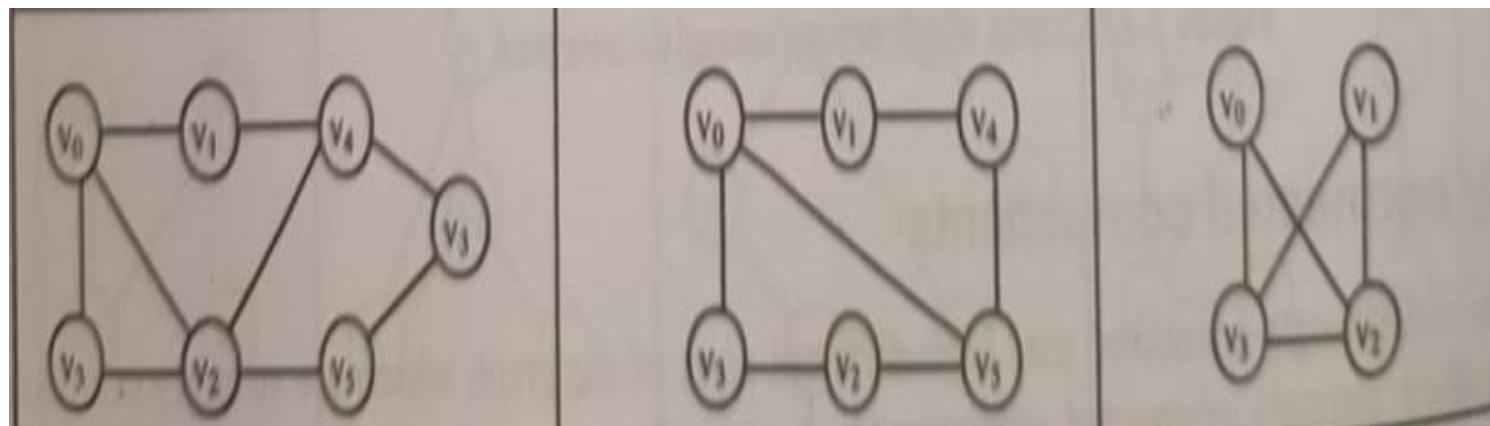


Articulation point: If on removing a vertex from a connected graph, the graph becomes disconnected then that vertex is called the articulation point.

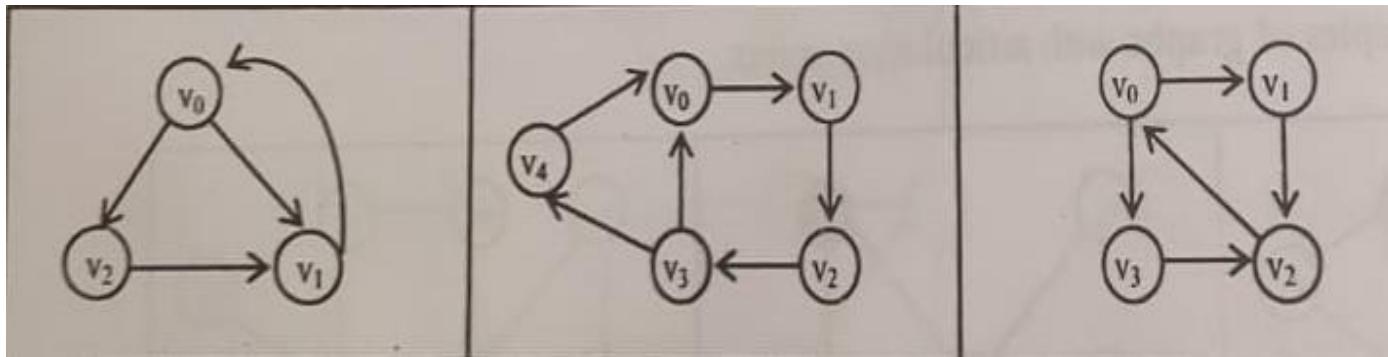


Graph terminology (cont.)

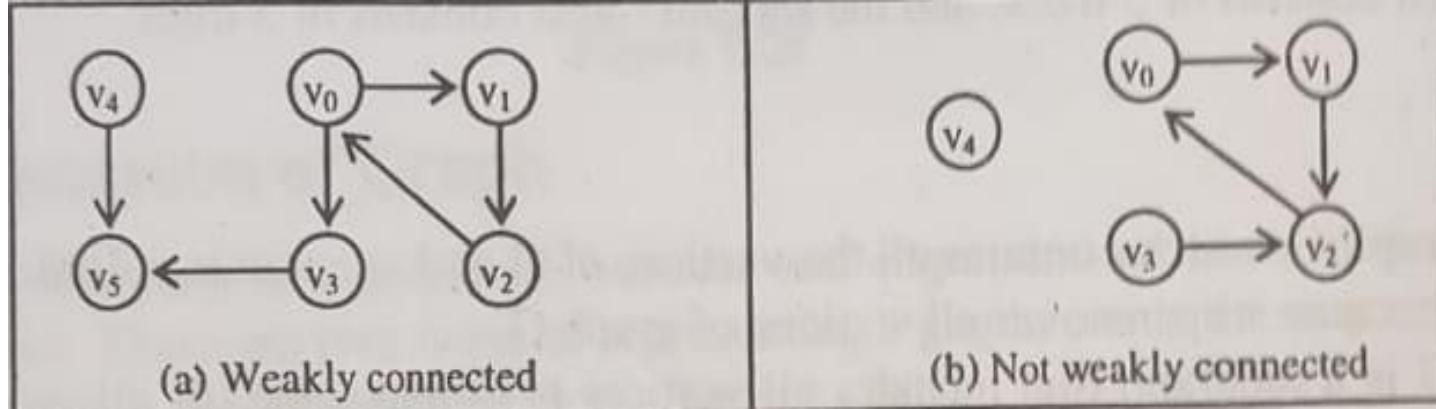
Biconnected graph: A connection graph with no articulation point is called a biconnected graph



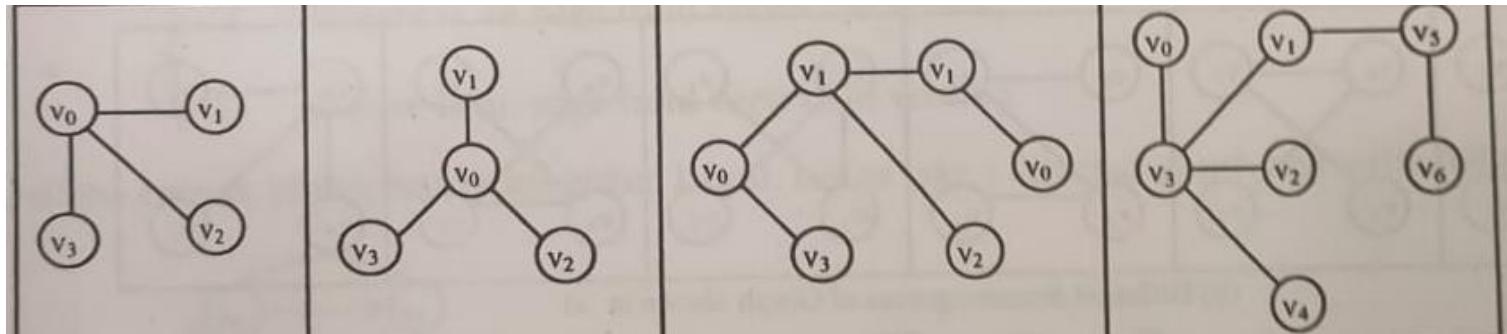
Strongly connected graph: A directed graph is strongly connected if there is a direct path from any vertex of graph to any other vertex.



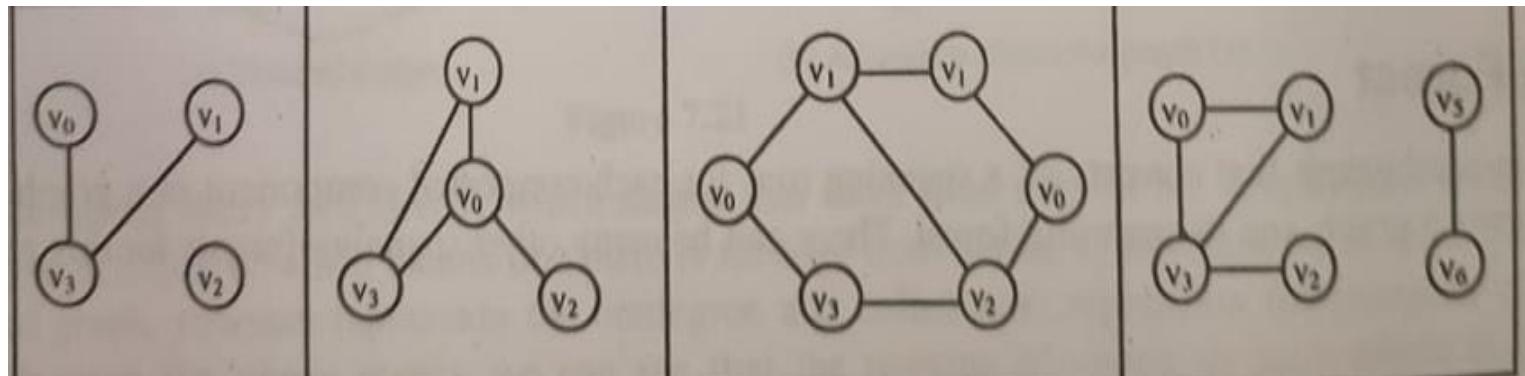
Weakly connected graph: A directed graph is weakly connected if for any pair of vertices u and v, there is a path from u to v or a path from v to u or both.



Tree: An undirected connected graph is called as Tree if there are no cycles in it. A tree with n vertices will have exactly $n-1$ edges.

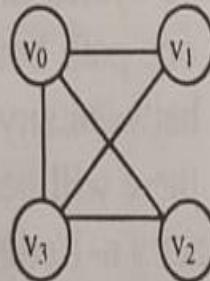


The following examples are graphs, which are not trees

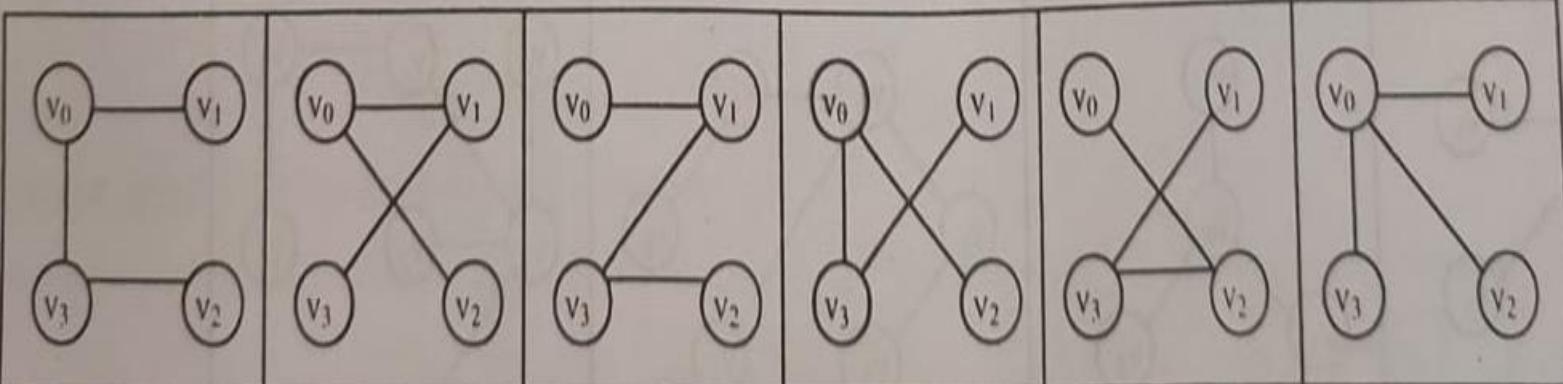


Spanning Tree: A subgraph T of a connected graph G , which contains all the vertices of G and is a tree is called a spanning tree of G .

Spanning tree of a graph is not unique, there can be more than one spanning trees of a graph.



(a) A Connected Graph



(b) Different Spanning trees of Graph shown in (a)

Applications of Graphs

- Representing relationships between components in electronic circuits
- Transportation networks: Highway network, Flight network
- Computer networks: Local area network, Internet, Web
- Databases: For representing ER (Entity Relationship) diagrams in databases, for representing dependency of tables in databases

Representation of Graph

We have mainly two parts in a graph, vertices and edges, and we have to design a Data Structure keeping in mind these parts. There are two ways of representing graph.

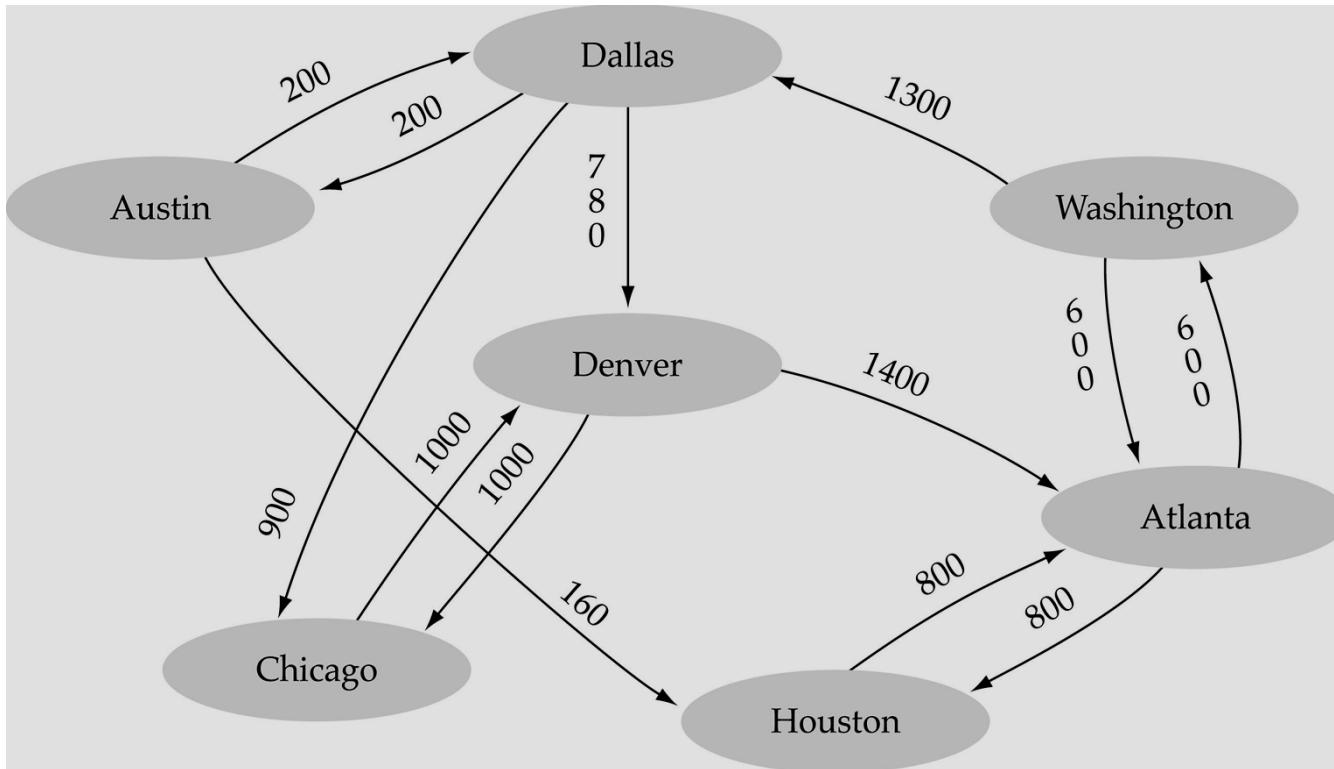
1. Sequential representation (Adjacency matrix)
2. Linked representation (Adjacency list)

Adjacency Matrix

1. One of the ways to represent a graph is to use two-dimensional matrix.
2. Each combination of row and column represent a vertex in the graph.
3. The value stored at the location row v and column u is the edge from vertex v to vertex u.
4. The nodes that are connected by an edge are called adjacent nodes. This matrix is used to store adjacent relation so it is called the Adjacency Matrix.
5. In the below diagram, we have a graph and its Adjacency matrix.

Graph terminology (cont.)

Weighted graph: a graph in which each edge carries a value

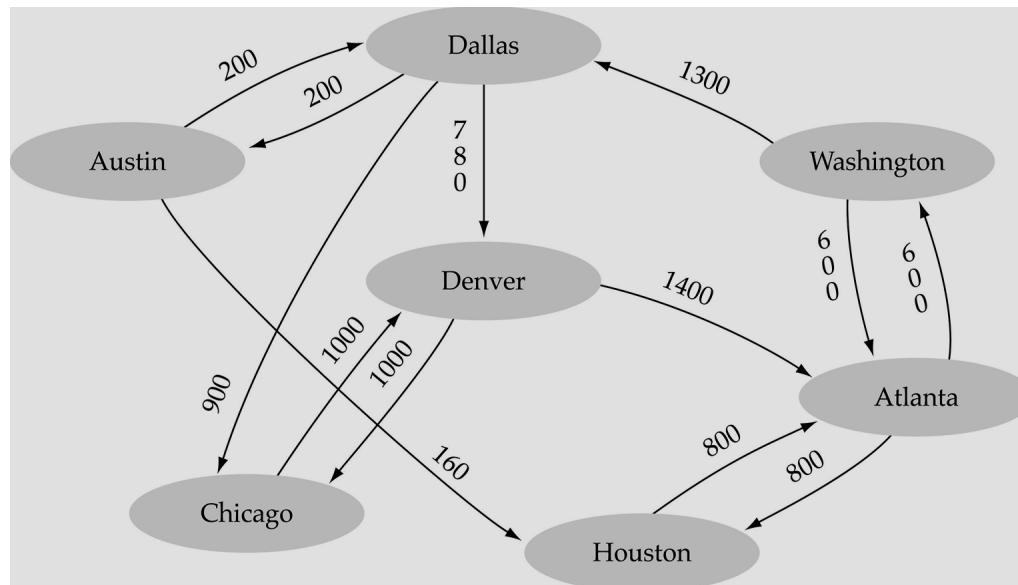


Graph implementation

Array-based implementation

A 1D array is used to represent the vertices

A 2D array (adjacency matrix) is used to represent the edges



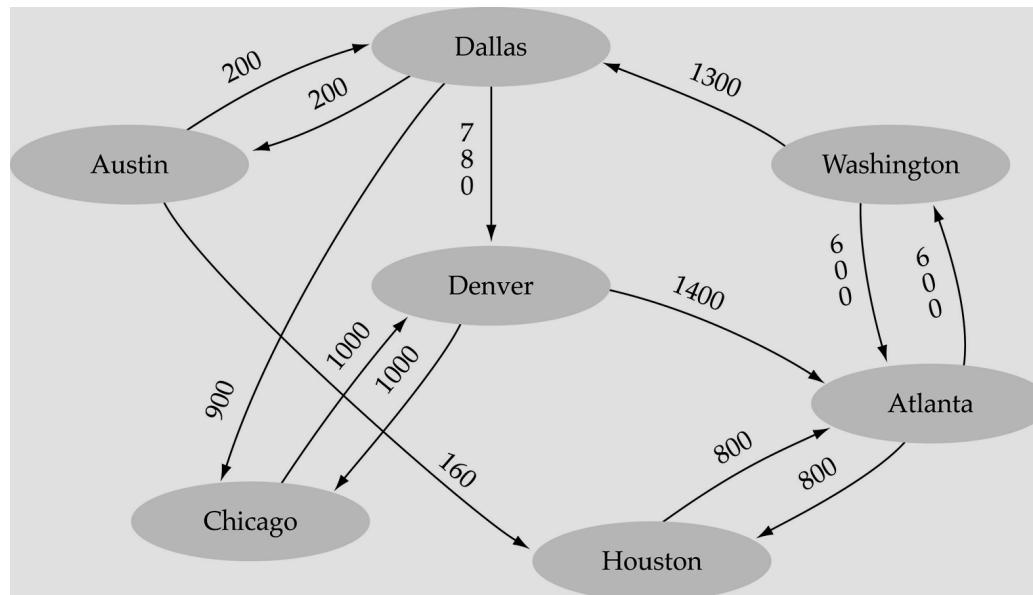
| graph | | | | | | | | | | | |
|----------------|--------------|--|--|--|--|--------|--|-----|------|------|--|
| .numVertices 7 | | | | | | | | | | | |
| .vertices | | | | | | .edges | | | | | |
| [0] | "Atlanta " | | | | | [0] | 0 | 0 | 0 | 0 | |
| [1] | "Austin " | | | | | [1] | 0 | 0 | 0 | 200 | |
| [2] | "Chicago " | | | | | [2] | 0 | 0 | 0 | 1000 | |
| [3] | "Dallas " | | | | | [3] | 0 | 200 | 900 | 0 | |
| [4] | "Denver " | | | | | [4] | 1400 | 0 | 1000 | 0 | |
| [5] | "Houston " | | | | | [5] | 800 | 0 | 0 | 0 | |
| [6] | "Washington" | | | | | [6] | 600 | 0 | 0 | 1300 | |
| [7] | | | | | | [7] | • | • | • | • | |
| [8] | | | | | | [8] | • | • | • | • | |
| [9] | | | | | | [9] | • | • | • | • | |
| | | | | | | | [0] | [1] | [2] | [3] | |
| | | | | | | | [4] | [5] | [6] | [7] | |
| | | | | | | | [8] | [9] | | | |
| | | | | | | | (Array positions marked '•' are undefined) | | | | |

Graph implementation (cont.)

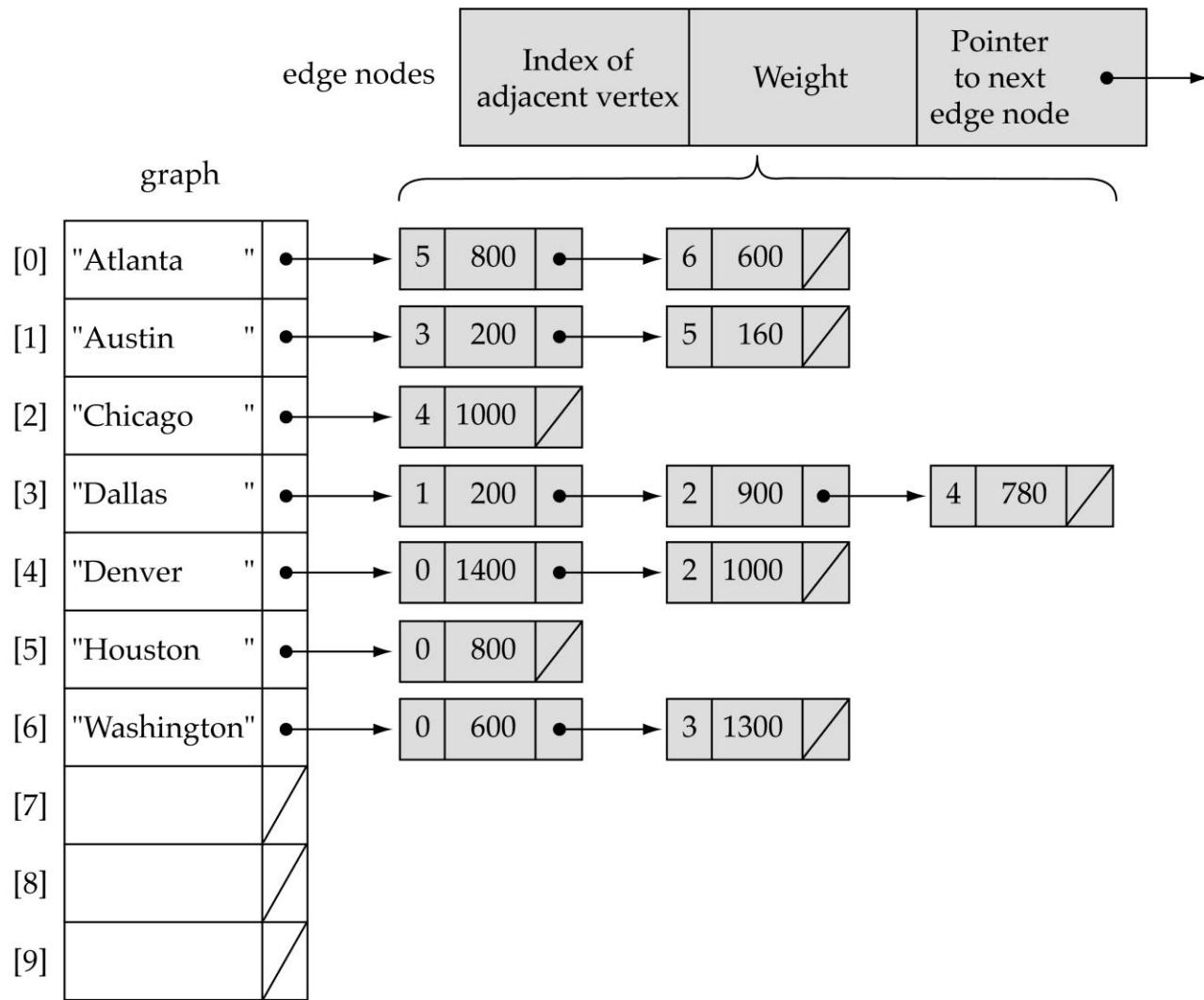
Linked-list implementation

A 1D array is used to represent the vertices

A list is used for each vertex v which contains the vertices which are adjacent from v (adjacency list)



(a)



Adjacency matrix vs. adjacency list representation

Adjacency matrix

Good for dense graphs -- $|E| \sim O(|V|^2)$

Memory requirements: $O(|V| + |E|) = O(|V|^2)$

Connectivity between two vertices can be tested quickly

Adjacency list

Good for sparse graphs -- $|E| \sim O(|V|)$

Memory requirements: $O(|V| + |E|) = O(|V|)$

Vertices adjacent to another vertex can be found quickly

Graph searching

Problem: find a path between two nodes of the graph

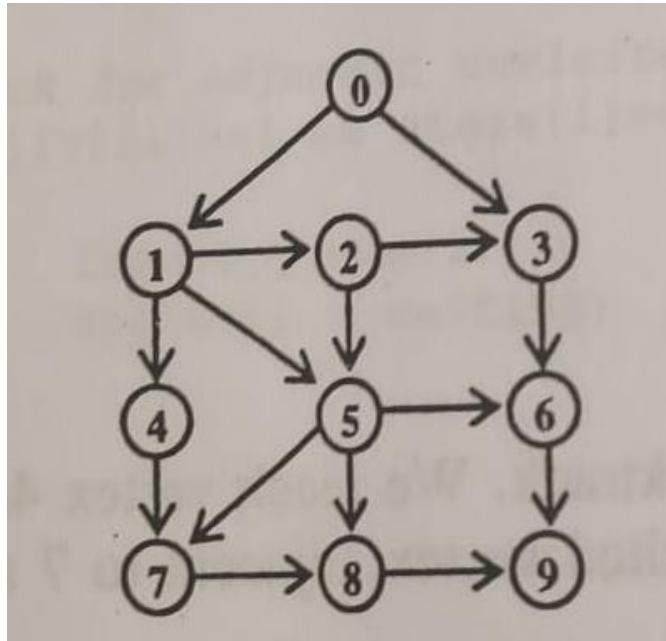
Methods: Depth-First-Search (**DFS**) or Breadth-First-Search (**BFS**)

Depth-First-Search (DFS)

Algorithm steps for DFS : Initially stack is empty

1. Push the starting node in the stack.
2. Loop until the stack is empty.
3. Pop the node from the stack inside loop.
4. If popped vertex is in initial state, visit it and change its state to visited.
5. Push all unvisited vertices adjacent to the popped vertex.
6. Repeat steps 3 to 5 until the stack is empty.

Note: There is no restriction on the order in which the successors of a vertex are visited.



Start vertex is 0, so push 0

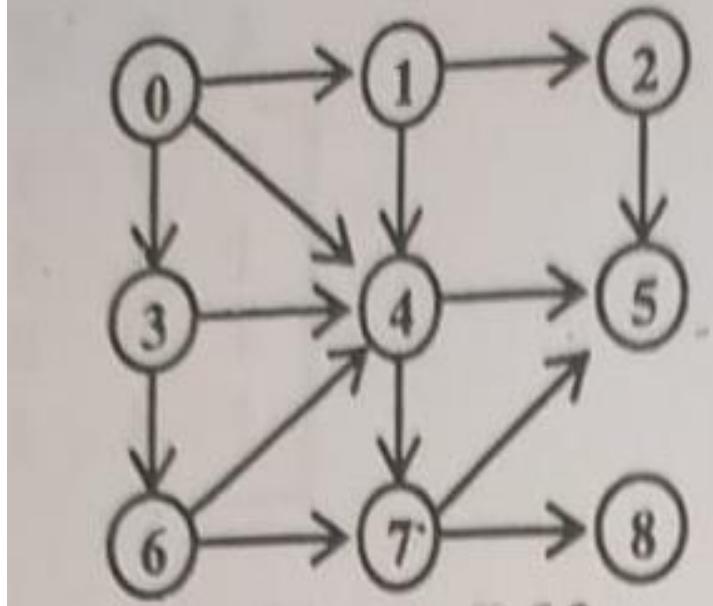
| | | | |
|--------|---------|--------------|---------------------|
| Pop 0: | visit 0 | push 3, 1 | stack 3, 1 |
| Pop 1: | visit 1 | push 5, 4, 2 | stack 3, 5, 4, 2 |
| Pop 2: | visit 2 | push 5, 3 | stack 3, 5, 4, 5, 3 |
| Pop 3: | visit 3 | push 6 | stack 3, 5, 4, 5, 6 |
| Pop 6: | visit 6 | push 9 | stack 3, 5, 4, 5, 9 |
| Pop 9: | visit 9 | | stack 3, 5, 4, 5 |
| Pop 5: | visit 5 | push 8, 7 | stack 3, 5, 4, 8, 7 |
| Pop 7: | visit 7 | push 8 | stack 3, 5, 4, 8, 8 |
| Pop 8: | visit 8 | | stack 3, 5, 4, 8 |
| Pop 8: | | | stack 3, 5, 4 |
| Pop 4: | visit 4 | | stack 3, 5 |
| Pop 5: | | | |
| Pop 3: | | | |

DFS traversal : 0 1 2 3 6 9 5 7 8 4

Breadth-First-Searching (BFS)

Algorithm steps for BFS : Initially queue is empty, and all vertices are at initial state.

1. Insert the starting node into the queue, change its state to waiting
2. Loop until the Queue is empty.
3. Remove a node from the queue inside loop, visit it and change its state to visited
4. Look for the adjacent vertices of the deleted node, and insert only those vertices in the queue, which are at initial state. Change the state all these vertices to waiting
6. Repeat steps 3 to 5 until Queue is empty



Start vertex is 0, so insert 0 in queue

| | | |
|-------------------|----------------|------------------|
| Remove 0: visit 0 | insert 1, 3, 4 | queue 1, 3, 4 |
| Remove 1: visit 1 | insert 2 | queue 3, 4, 2 |
| Remove 3: visit 3 | insert 6 | queue 4, 2, 6 |
| Remove 4: visit 4 | insert 5, 7 | queue 2, 6, 5, 7 |
| Remove 2: visit 2 | insert | queue 6, 5, 7 |
| Remove 6: visit 6 | insert | queue 5, 7 |
| Remove 5: visit 5 | insert | queue 7 |
| Remove 7: visit 7 | insert 8 | queue 8 |
| Remove 8: visit 8 | insert | queue EMPTY |

Here 3 states are maintained, initial, waiting and visited. The vertices that are in waiting or visited state are not inserted in queue.

Thanks

```

class myGraph {

    int adj[][];
    int n;

    public myGraph(){
        n=10;
        adj = new int[n][n];
    }

    public myGraph(int s){
        n=s;
        adj = new int[n][n];

        //initialize all array element to 0
    }

    public void createGraph()
    {
        int i,max_edges,origin,dest;

        max_edges = n*(n-1);

        for(i=0 ; i<max_edges ; i++)
        {
            sysout("\n Enter edge (-1 -1) to quit");
            origin = sc.nextInt();
            dest = sc.nextInt();

            if((origin)==-1) && (dest == -1))
                break;

            if((origin) < 0) || (dest < 0 ) || (origin > =n) || (dest>=n) )
            {
                sysout("Invalid input");
                i--;
            }
            else
                adj[origin][dest] = 1;
        }
        //end of loop
    } //end of function

    public void insert_edge(int origin, int dest)
    {
        if((origin<0) || (origin >=n))
        {
            sysout("\nInvalid origin....");
            return;
        }

        if ((dest<0) || (dest >=n))
        {
            sysout("\nInvalid dest....");
            return;
        }

        adj[origin][dest] = 1;
    }
}

```

```

}

public void del_edge(int origin, int dest)
{
    if((origin<0) || (origin >=n))
    {
        sysout("\nInvalid origin....");
        return;
    }

    if ((dest<0) || (dest >=n))
    {
        sysout("\nInvalid dest....");
        return;
    }
    if(adj[origin][dest] == 0)
    {
        sysout("\n This edge does not exist..."); 
        retrun;
    }

    adj [origin] [dest]=0;
    return;
}

public void display()
{
    int i,j;

    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            sysout(" " + adj[i][j]);
        }
        sysout("\n");
    }
}

public void DF_Traverse(int v)      // state is 0 for initial, 1 for
visited
{
    intStack st = new intStack(100);
    int i;

    int state[] =new int(10);

    for(i=0;i<n;i++)
        state[i] = 0;

    st.push(v);

    while(!st.isEmpty())
    {
        v=st.pop();

        if(state[v] == 0)
        {

```

```

        sysout(" " + v);
        state[v] = 1;
    }
    for(i=0;i<n;i++)
    {
        if(adj[v][i] == 1 && state[i] == 0)
            st.push(i);
    }
}
}

public void BF_Traverse(int v) // state = 0 for initial, 1 for waiting
and 2 for visited
{
    int i;
    int state[] = new int(50);

    for(i=0;i<n;i++)
        state[i] = 0;

    intQueue q = new intQueue(50);

    q.insert(v);
    state[v] = 1

    while(!q.isEmpty())
    {
        v = q.del();
        sysout(" " + v);
        state[v] = 2;

        for(i=0 ; i<n ; i++)
        {
            if(adj[v][i] == 1 && state[i] == 0)
            {
                q.insert(i);
                state[i] = 1;
            }
        }
    }
}

}

public class Graph_Main{

    public static void main()
    {
        myGraph g = new myGraph(10);
        //g.createGraph();

        g.insert_edge(0,3);
        g.insert_edge(0,2);
        g.insert_edge(1,2);
        g.insert_edge(1,3);
        g.insert_edge(2,3);
    }
}
```

```
g.insert_edge(4, 3);
g.insert_edge(0, 3);
g.insert_edge(0, 3);
g.insert_edge(0, 3);

g.BF_Traverse(0);

g.DF_Traverse(0);
}
```

```

class Graph_for_Dijkstra {
    int adj[][];
    int n;
    int status[];
    int prede[];
    int pathlength[];

    Graph_for_Dijkstra(int d)
    {
        n=d;
        adj = new int[n][n];
        prede = new int[n];
        pathlength = new int[n];
        status = new int[n];

        for(int i=0;i<n;i++)
            status[i] = 0;

    }

    public void insert_edge(int origin,int dest, int weight)
    {
        //same function

        adj[origin][dest]=weight;
    }

    public int min_length()
    {

    }

    public void Dijkstra(int s)
    {
        int i, current;

        for(i=0;i<n;i++)
        {
            prede[i] = -1;
            pathlength[i] = 9999;
            status[i] = 0;
        }

        pathlength[s]=0;

        while(true)
        {
            /* Search for temp vertex with minimum pathlength and make it
current */
            current = min_temp();

            if(current == -1)
                return;

            status[current] = 1;

```

```

        for(i=0 ; i<n ; i++)
    {
        /*Check all adjacent temp vertices */

        if(adj[current][i] != 0 && status[i] == 0)
        {
            if(pathlength[current] + adj[current][i] < pathlength[i])
            {
                pathlength[i] = pathlength[current] + adj[current][i];
                prede[i] = current;
            }
        }
    }
}//end of while
}//end of function

/*Returns the temp vertex with minimum length of pathlength,
 returns -1 if no temp vertex left
 all temp vertices left have pathlength 9999

*/
public int min_temp()
{
    int i;
    int min = 9999;
    int k=-1;

    for(i=0;i<n;i++)
    {
        if(status[i] == 0 && pathlength[i] < min)
        {
            min = pathlength[i];
            k = i;
        }
    }
    return k;
}

public void findPath(int s, int v) //s=0 v=5
{
    int i,count=0, shortDist=0;
    int u;
    int path[] = new int[50];

    while(v != s)
    {
        count++; //count = 4
        path[count] = v;
        u = prede[v]; //u=2
        shortDist = shortDist+adj[u][v];
        v = u; //v = 0
    }
    count++; //5
    path[count] = s;

    System.out.print("\n Shortest Path...");
    for(i=count; i>=1 ;i--)

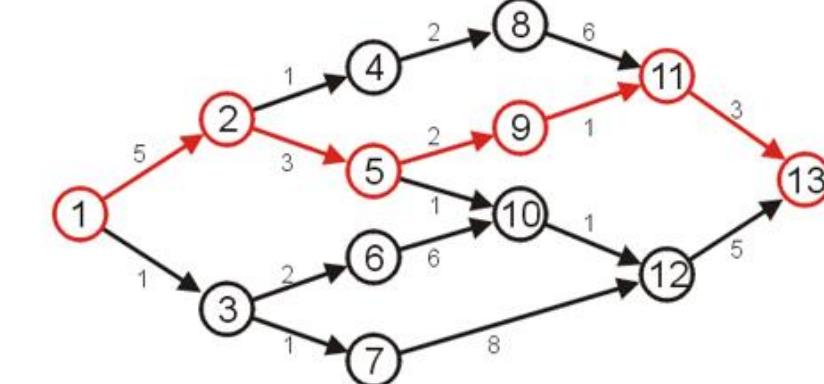
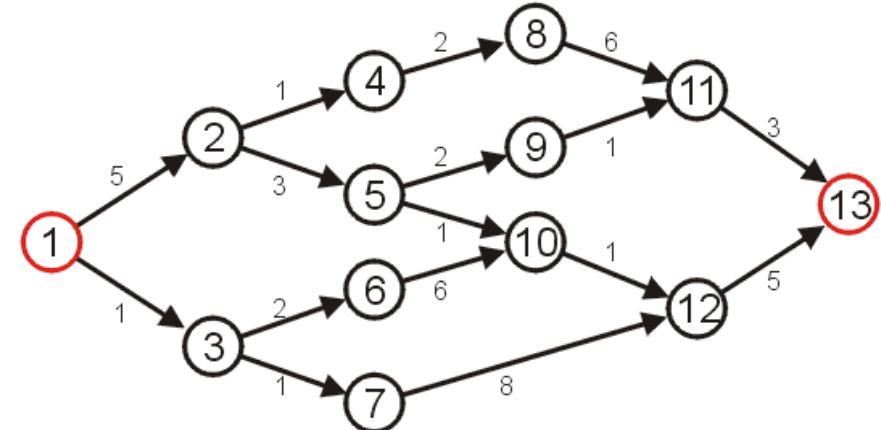
```

```
    sysout(path[i]);  
    sysout("Short dist..." + shortDist);  
}//end of findPAth  
  
path 5 4 6 2 0
```

Dijkstra's algorithm

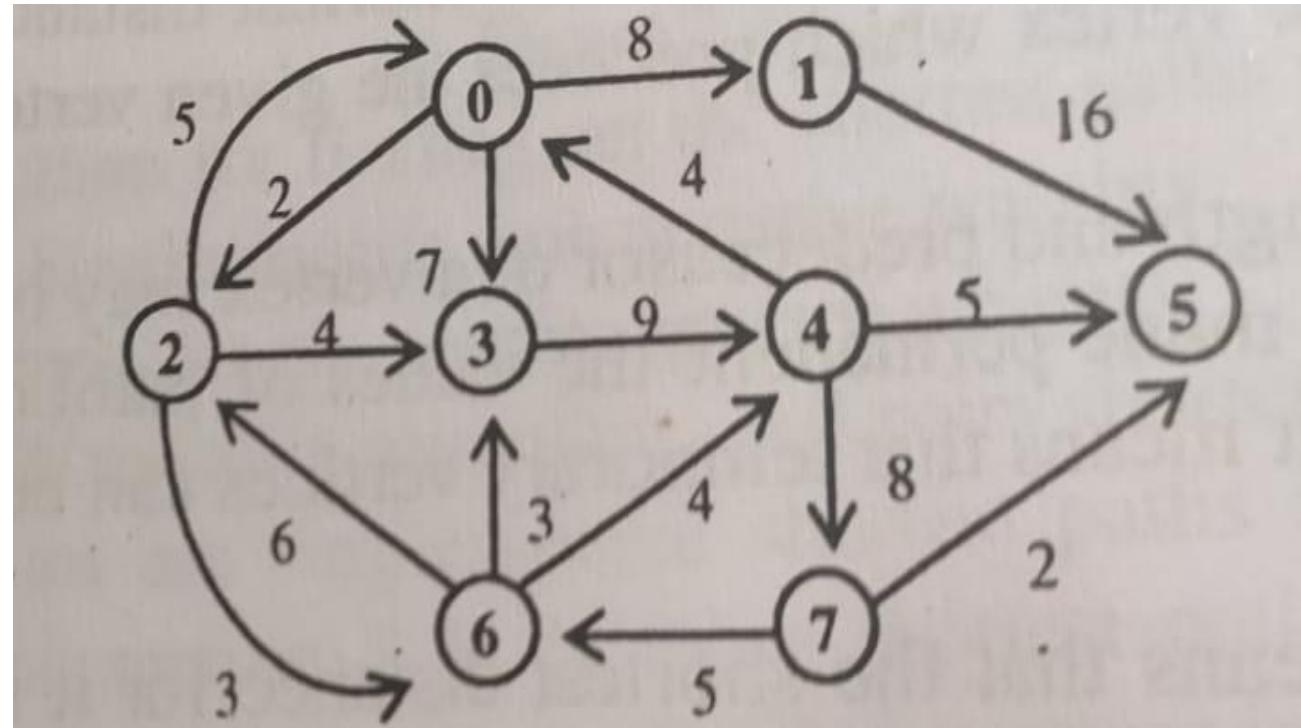
Shortest Path

- Given the graph below, suppose we wish to find the shortest path from vertex 1 to vertex 13
- After some consideration, we may determine that the shortest path is as follows, with length 14
- Other paths exists, but they are longer



Single-Source Shortest Path Problem

Single-Source Shortest Path Problem - The problem of finding shortest paths from a source vertex v to all other vertices in the graph.



Dijkstra's algorithm

Dijkstra's algorithm - is a solution to the single-source shortest path problem in graph theory.

Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

Approach: Greedy

Input: Weighted graph $G=\{E,V\}$ and source vertex $v \in V$, such that all edge weights are nonnegative

Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices

Dijkstra's algorithm

- A. Initialize the pathlength of vertices to infinity(9999) and predecessor of all vertices to NIL(-1). Make the status of all vertices temporary.
- B. Make the pathlength of source vertex equal to 0
- C. From all the temporary vertices in the graph, find out the vertex that has minimum value of pathlength, make it permanent and now this is our current vertex.(if there are many such vertices, then any one can be taken)
- D. Examine all the temporary vertices adjacent to the current vertex. The value of pathlength is recalculated for all these temporary successors of current and relabelling is done if required.

Dijkstra's algorithm

Suppose s is the source vertex, current is the current vertex and v is a temporary vertex adjacent to current .

If $\text{pathlength}(\text{current}) + \text{weight}(\text{current}, v) < \text{pathlength}(v)$

It means that path from s to v via current is smaller than the path currently assigned to v . We need to update the previous path.

$$\text{pathlength}(v) = \text{pathlength}(\text{current}) + \text{weight}(\text{current}, v)$$

$$\text{predecessor}(v) = \text{current}$$

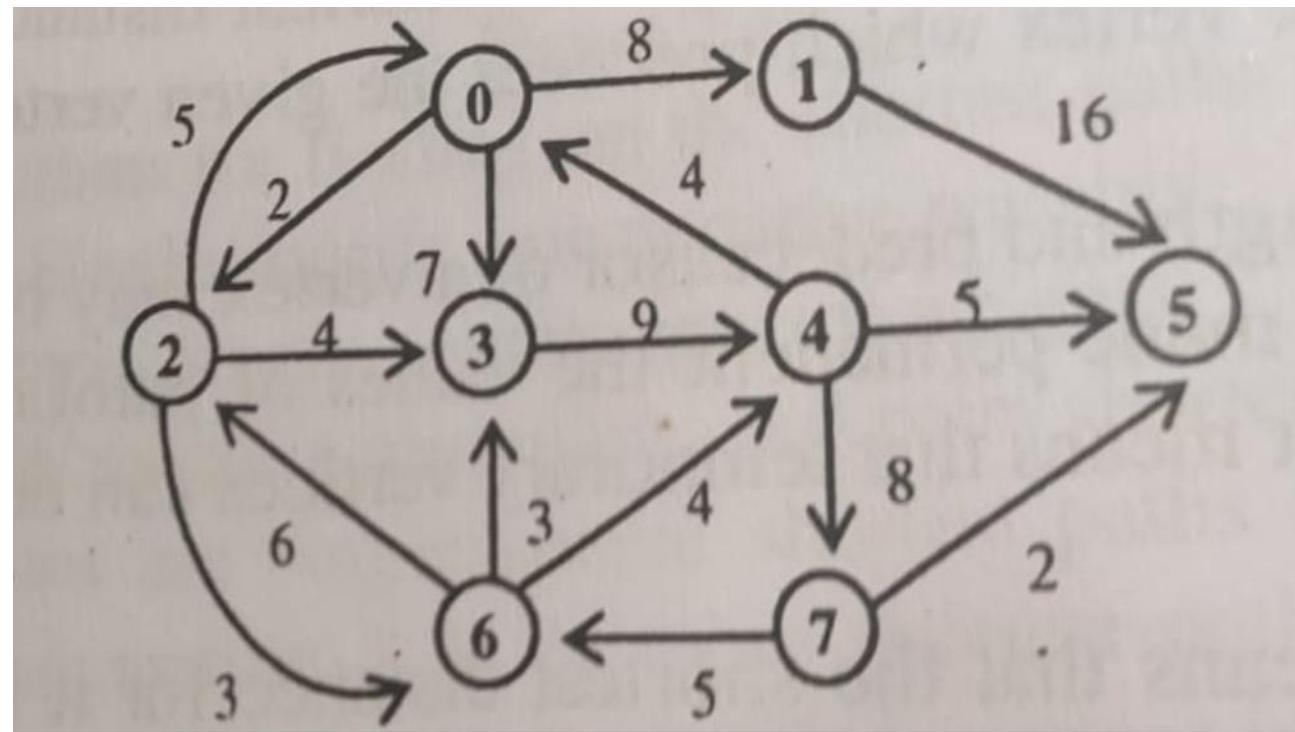
Dijkstra's algorithm

If $\text{pathlength}(\text{current}) + \text{weight}(\text{current}, v) \geq \text{pathlength}(v)$

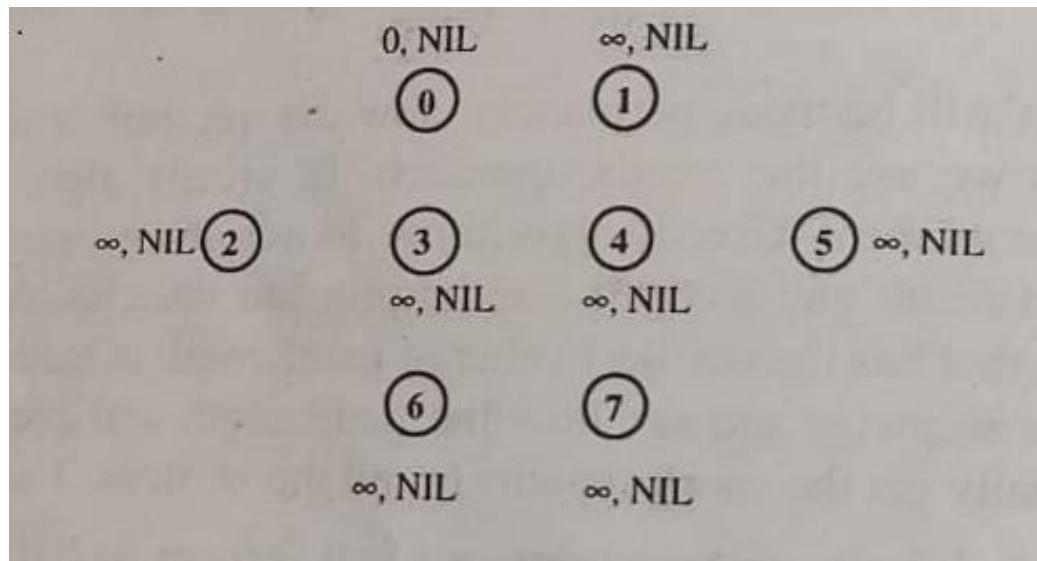
In this case vertex v is not relabelled and the values of pathlength and predecessor for vertex v remains unchanged.

E. Repeat steps C and D until there is no temporary vertex left in the graph

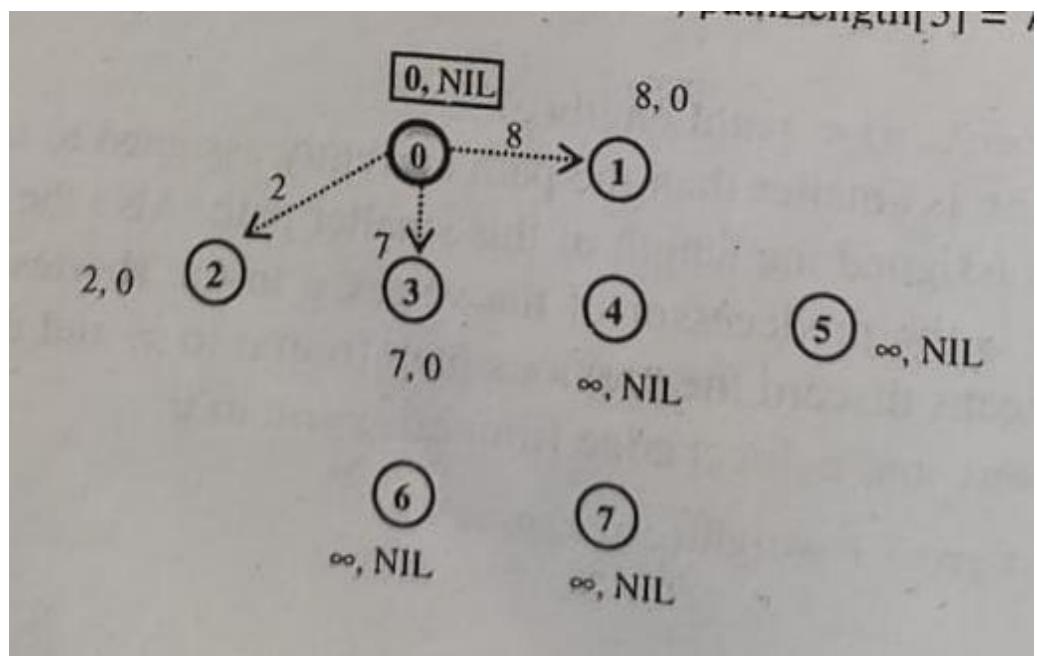
Dijkstra's algorithm on the graph below



1



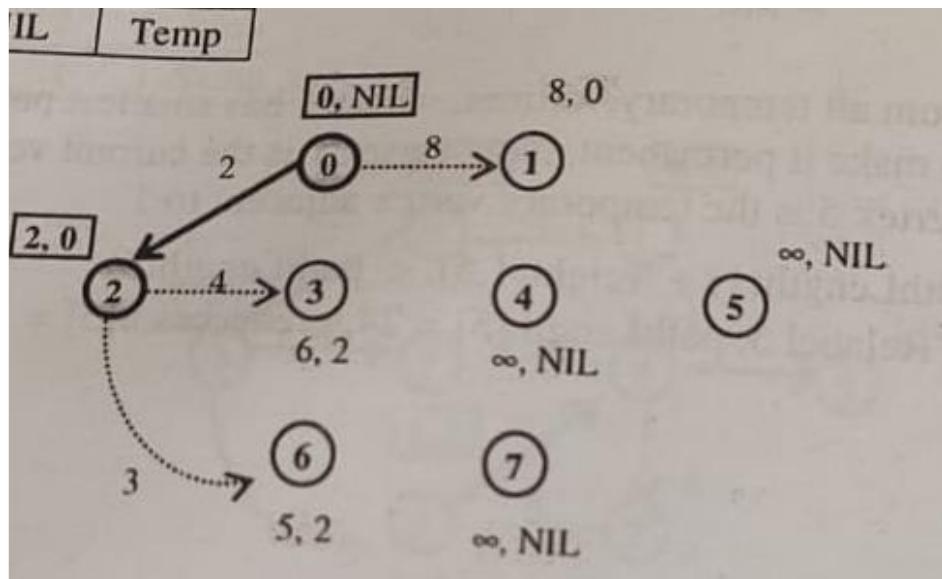
2



| Vertex | path Length | prede-cessor | status |
|--------|-------------|--------------|--------|
| 0 | 0 | NIL | Temp |
| 1 | ∞ | NIL | Temp |
| 2 | ∞ | NIL | Temp |
| 3 | ∞ | NIL | Temp |
| 4 | ∞ | NIL | Temp |
| 5 | ∞ | NIL | Temp |
| 6 | ∞ | NIL | Temp |
| 7 | ∞ | NIL | Temp |

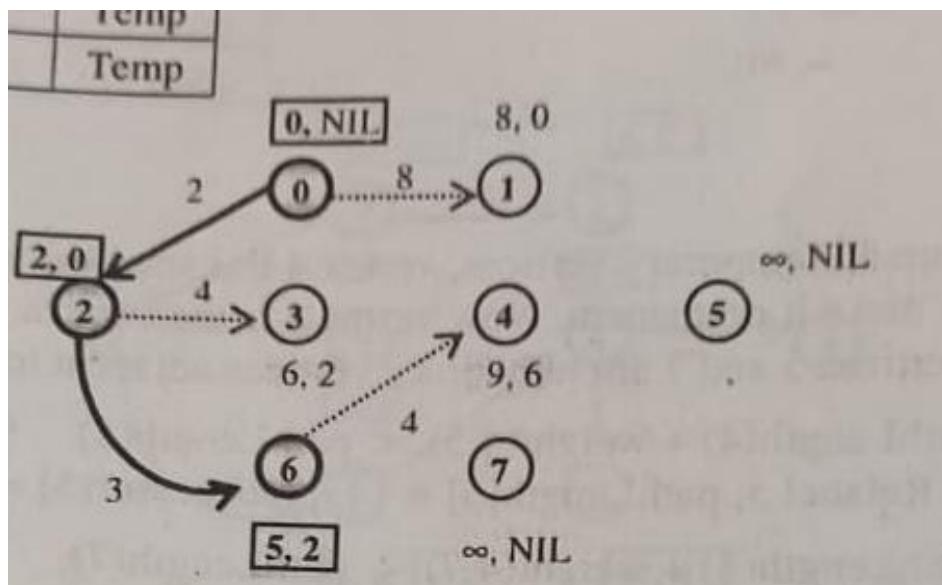
| Vertex | path Length | prede-cessor | status |
|--------|-------------|--------------|--------|
| 0 | 0 | NIL | Perm |
| 1 | 8 | 0 | Temp |
| 2 | 2 | 0 | Temp |
| 3 | 7 | 0 | Temp |
| 4 | ∞ | NIL | Temp |
| 5 | ∞ | NIL | Temp |
| 6 | ∞ | NIL | Temp |
| 7 | ∞ | NIL | Temp |

3

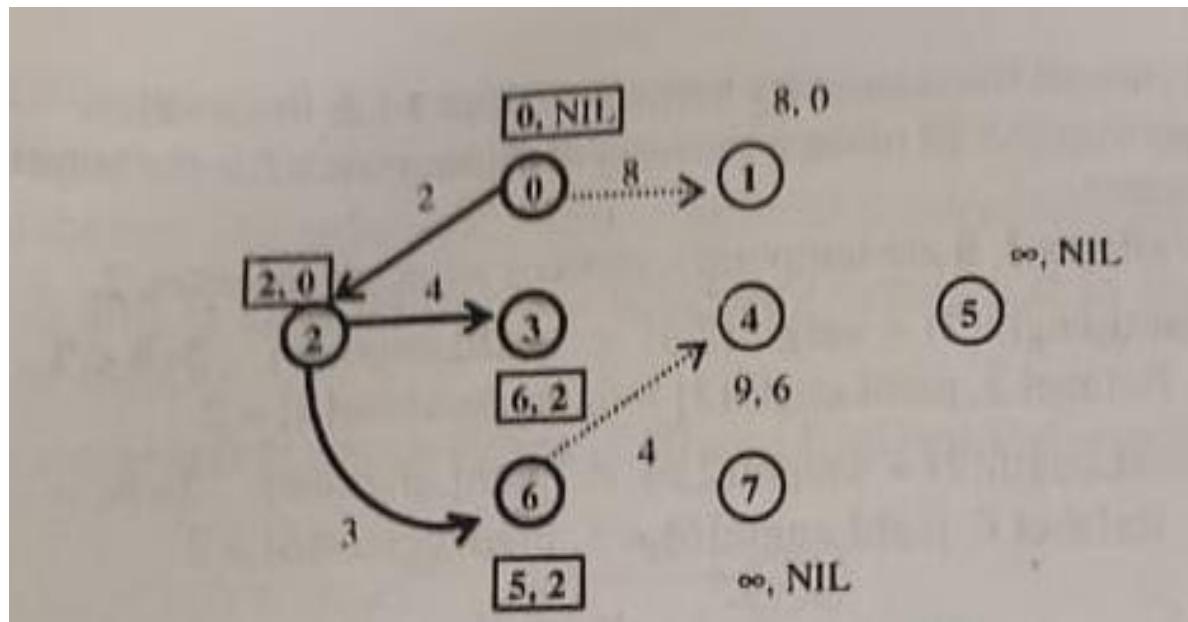


| Vertex | path Length | prede-cessor | status |
|--------|-------------|--------------|--------|
| 0 | 0 | NIL | Perm |
| 1 | 8 | 0 | Temp |
| 2 | 2 | 0 | Perm |
| 3 | 6 | 2 | Temp |
| 4 | ∞ | NIL | Temp |
| 5 | ∞ | NIL | Temp |
| 6 | 5 | 2 | Temp |
| 7 | ∞ | NIL | Temp |

4

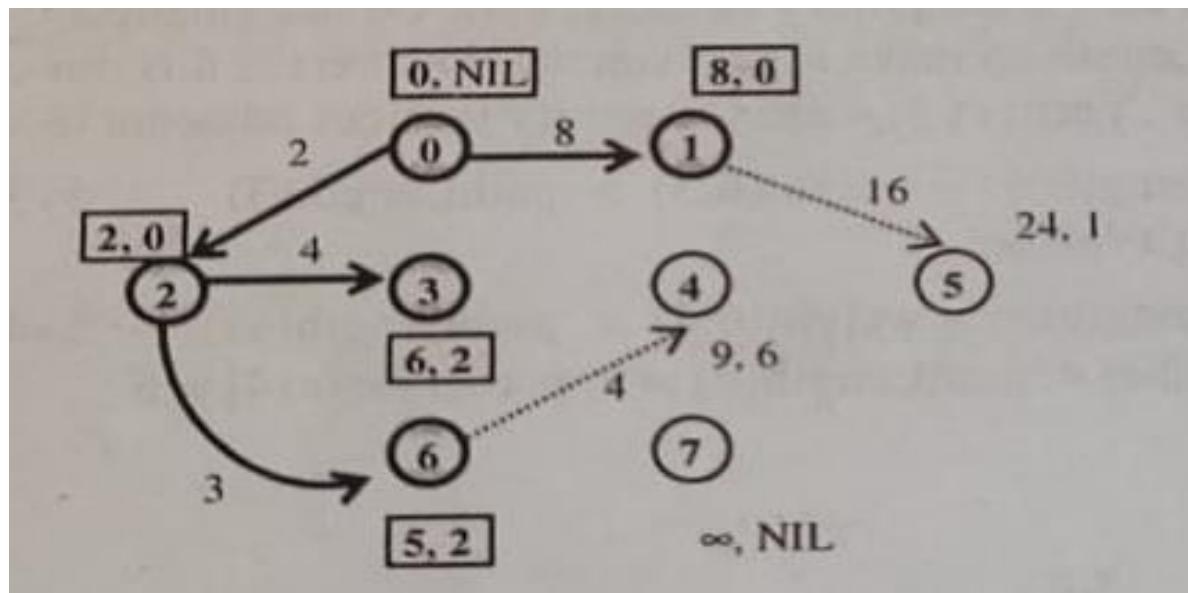


| Vertex | path Length | prede-cessor | status |
|--------|-------------|--------------|--------|
| 0 | 0 | NIL | Perm |
| 1 | 8 | 0 | Temp |
| 2 | 2 | 0 | Perm |
| 3 | 6 | 2 | Temp |
| 4 | 9 | 6 | Temp |
| 5 | ∞ | NIL | Temp |
| 6 | 5 | 2 | Perm |
| 7 | ∞ | NIL | Temp |



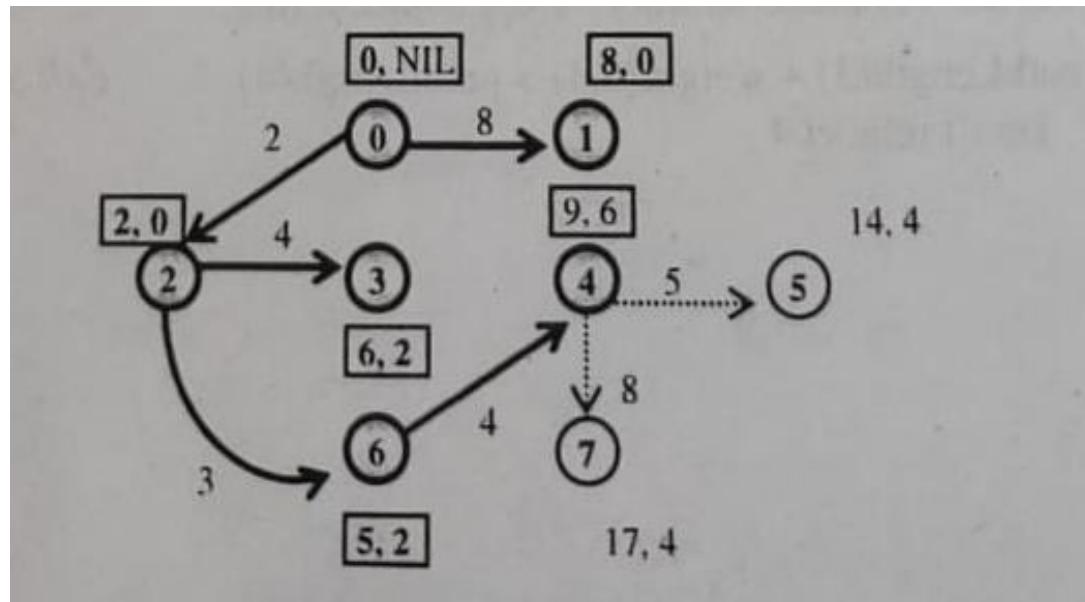
5

| Vertex | path Length | prede-cessor | status |
|--------|-------------|--------------|--------|
| 0 | 0 | NIL | Perm |
| 1 | 8 | 0 | Temp |
| 2 | 2 | 0 | Perm |
| 3 | 6 | 2 | Perm |
| 4 | 9 | 6 | Temp |
| 5 | ∞ | NIL | Temp |
| 6 | 5 | 2 | Perm |
| 7 | ∞ | NIL | Temp |

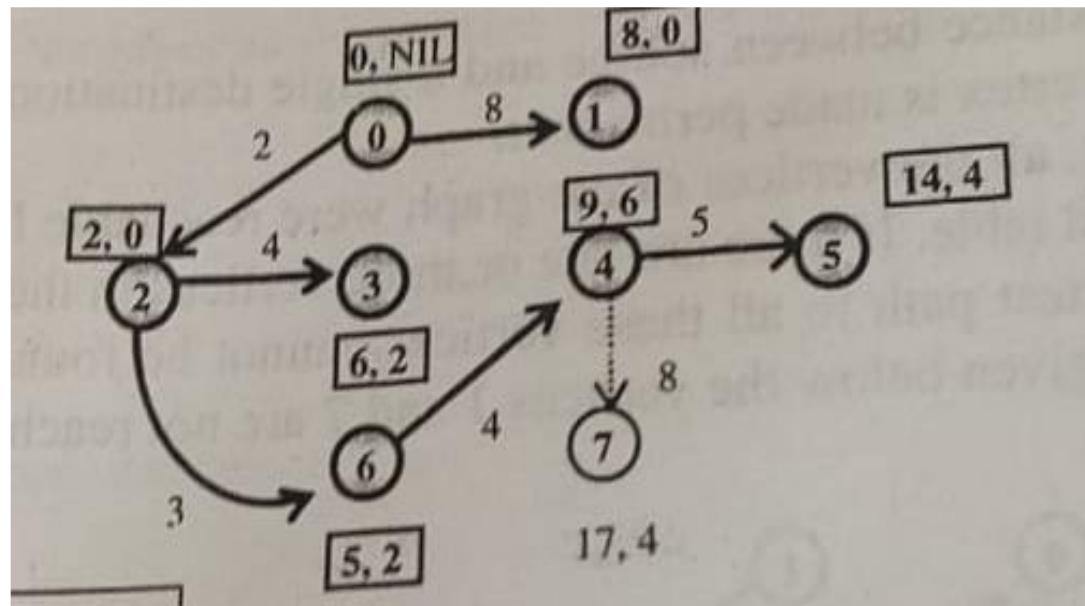


6

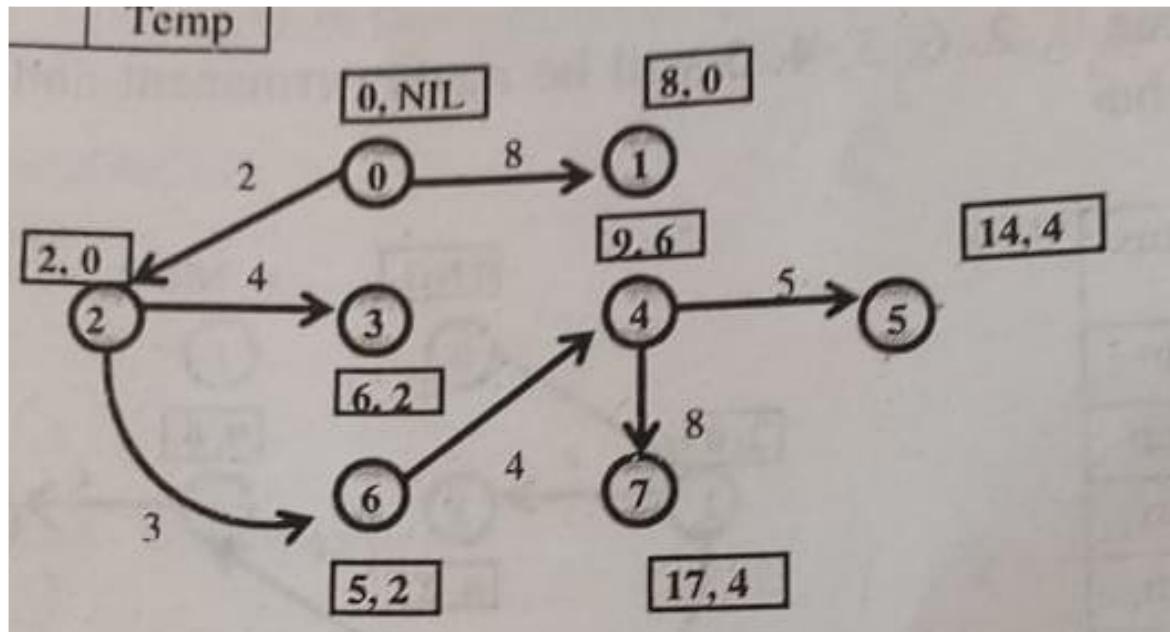
| Vertex | path Length | prede-cessor | status |
|--------|-------------|--------------|--------|
| 0 | 0 | NIL | Perm |
| 1 | 8 | 0 | Perm |
| 2 | 2 | 0 | Perm |
| 3 | 6 | 2 | Perm |
| 4 | 9 | 6 | Temp |
| 5 | 24 | 1 | Temp |
| 6 | 5 | 2 | Perm |
| 7 | ∞ | NIL | Temp |



| Vertex | path Length | prede-cessor | status |
|--------|-------------|--------------|--------|
| 0 | 0 | NIL | Perm |
| 1 | 8 | 0 | Perm |
| 2 | 2 | 0 | Perm |
| 3 | 6 | 2 | Perm |
| 4 | 9 | 6 | Perm |
| 5 | 14 | 4 | Temp |
| 6 | 5 | 2 | Perm |
| 7 | 17 | 4 | Temp |



| Vertex | path Length | prede-cessor | status |
|--------|-------------|--------------|--------|
| 0 | 0 | NIL | Perm |
| 1 | 8 | 0 | Perm |
| 2 | 2 | 0 | Perm |
| 3 | 6 | 2 | Perm |
| 4 | 9 | 6 | Perm |
| 5 | 14 | 4 | Perm |
| 6 | 5 | 2 | Perm |
| 7 | 17 | 4 | Temp |



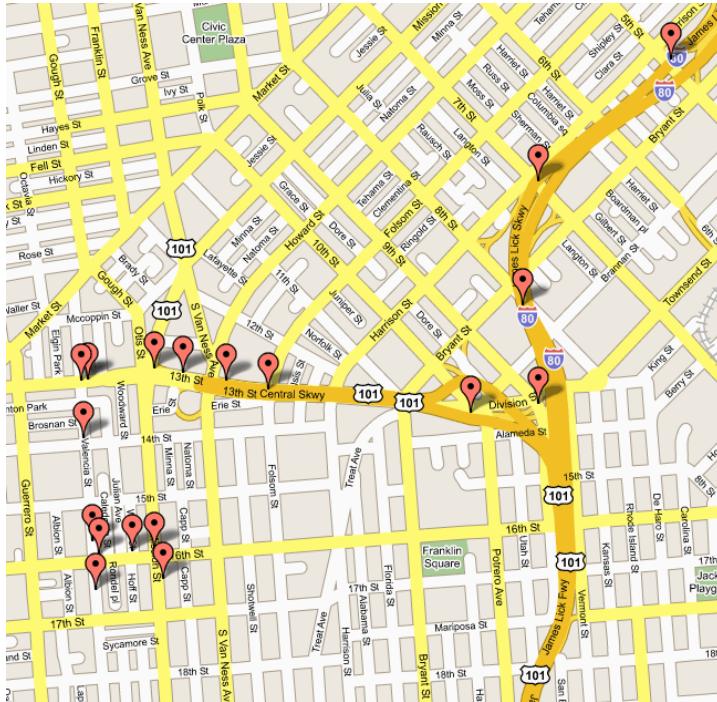
| Vertex | path Length | prede-cessor | status |
|--------|-------------|--------------|--------|
| 0 | 0 | NIL | Perm |
| 1 | 8 | 0 | Perm |
| 2 | 2 | 0 | Perm |
| 3 | 6 | 2 | Perm |
| 4 | 9 | 6 | Perm |
| 5 | 14 | 4 | Perm |
| 6 | 5 | 2 | Perm |
| 7 | 17 | 4 | Perm |

DIJKSTRA'S ALGORITHM - WHY USE IT?

- As mentioned, Dijkstra's algorithm calculates the shortest path to every vertex.
- However, it is about as computationally expensive to calculate the shortest path from vertex u to every vertex using Dijkstra's as it is to calculate the shortest path to some particular vertex v .
- Therefore, anytime we want to know the optimal path to some other vertex from a determined origin, we can use Dijkstra's algorithm.

Applications of Dijkstra's Algorithm

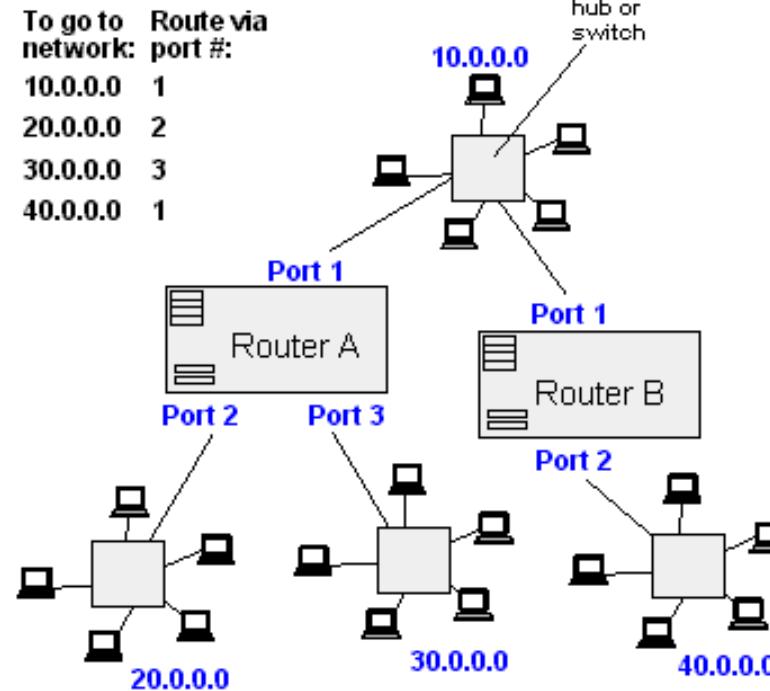
- Traffic Information Systems are most prominent use
- Mapping (Map Quest, Google Maps)
- Routing Systems



From Computer Desktop Encyclopedia
© 1998 The Computer Language Co. Inc.

Router A Routing Table

| To go to network: | Route via port #: |
|-------------------|-------------------|
| 10.0.0.0 | 1 |
| 20.0.0.0 | 2 |
| 30.0.0.0 | 3 |
| 40.0.0.0 | 1 |



Bellman Ford algorithm

Bellman Ford Algorithm

- The bellman ford algorithm works even when there are negative weight edges in the graph. It does not work if there is some cycle in the graph whose total weight is negative.
- In Dijkstra algorithm, we make a vertex Permanent at each step.
- In Bellman Ford algorithm the shortest distance is finalized at the end of the algorithm
- Here we drop the concept of making vertices permanent.
- Dijkstra's algorithm is called as label setting algorithm and Bellman Ford algorithm is called as label correcting algorithm

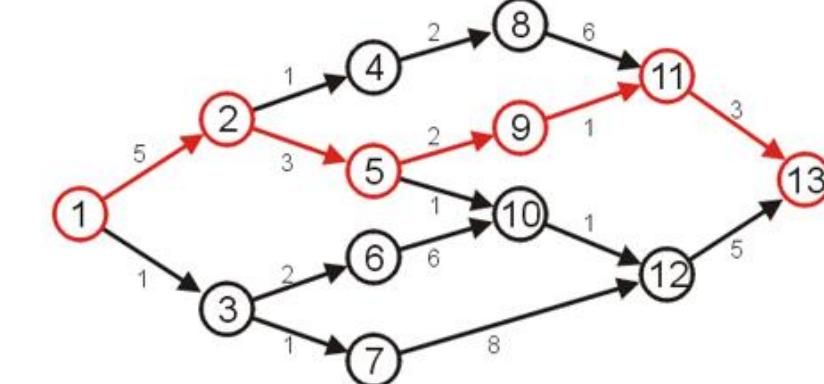
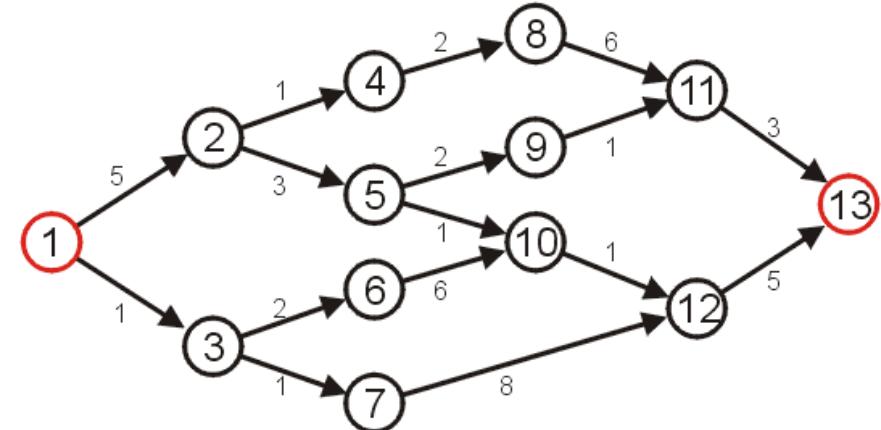
Bellman Ford algorithm

- A. Initialize the pathlength of all vertices to infinity, predecessor to NIL
- B. Make the pathlength of source vertex equal to 0 and insert it into queue
- C. Delete vertex from queue and make it current vertex
- D. Examine all the vertices adjacent to the current. Check the condition of minimum weight for these vertices and do the relabelling if required, as in Dijkstra's algorithm.
- E. Each label that is relabelled is inserted into queue provided it is not already present in the queue.
- F. Repeat steps C, D, E till queue becomes empty.

Dijkstra's algorithm

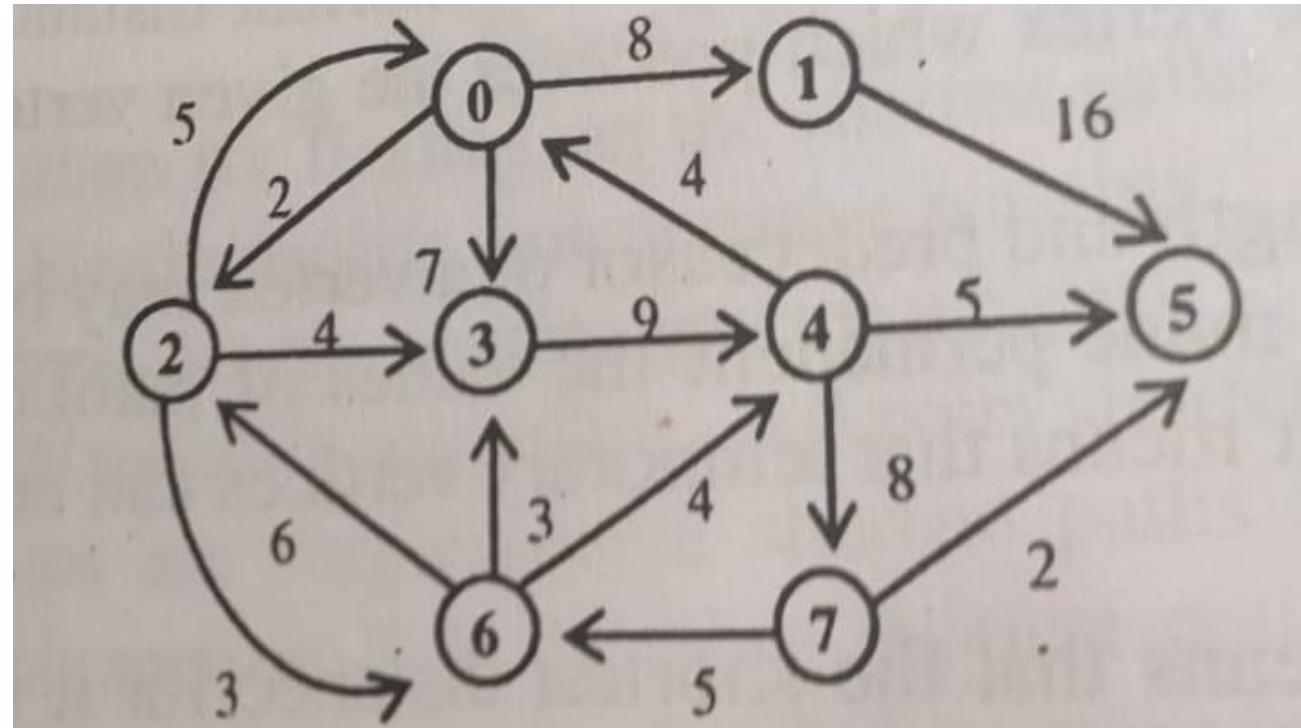
Shortest Path

- Given the graph below, suppose we wish to find the shortest path from vertex 1 to vertex 13
- After some consideration, we may determine that the shortest path is as follows, with length 14
- Other paths exists, but they are longer



Single-Source Shortest Path Problem

Single-Source Shortest Path Problem - The problem of finding shortest paths from a source vertex v to all other vertices in the graph.



Dijkstra's algorithm

Dijkstra's algorithm - is a solution to the single-source shortest path problem in graph theory.

Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

Approach: Greedy

Input: Weighted graph $G=\{E,V\}$ and source vertex $v \in V$, such that all edge weights are nonnegative

Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices

Dijkstra's algorithm

- A. Initialize the pathlength of vertices to infinity(9999) and predecessor of all vertices to NIL(-1). Make the status of all vertices temporary.
- B. Make the pathlength of source vertex equal to 0
- C. From all the temporary vertices in the graph, find out the vertex that has minimum value of pathlength, make it permanent and now this is our current vertex.(if there are many such vertices, then any one can be taken)
- D. Examine all the temporary vertices adjacent to the current vertex. The value of pathlength is recalculated for all these temporary successors of current and relabelling is done if required.

Dijkstra's algorithm

Suppose s is the source vertex, current is the current vertex and v is a temporary vertex adjacent to current .

If $\text{pathlength}(\text{current}) + \text{weight}(\text{current}, v) < \text{pathlength}(v)$

It means that path from s to v via current is smaller than the path currently assigned to v . We need to update the previous path.

$$\text{pathlength}(v) = \text{pathlength}(\text{current}) + \text{weight}(\text{current}, v)$$

$$\text{predecessor}(v) = \text{current}$$

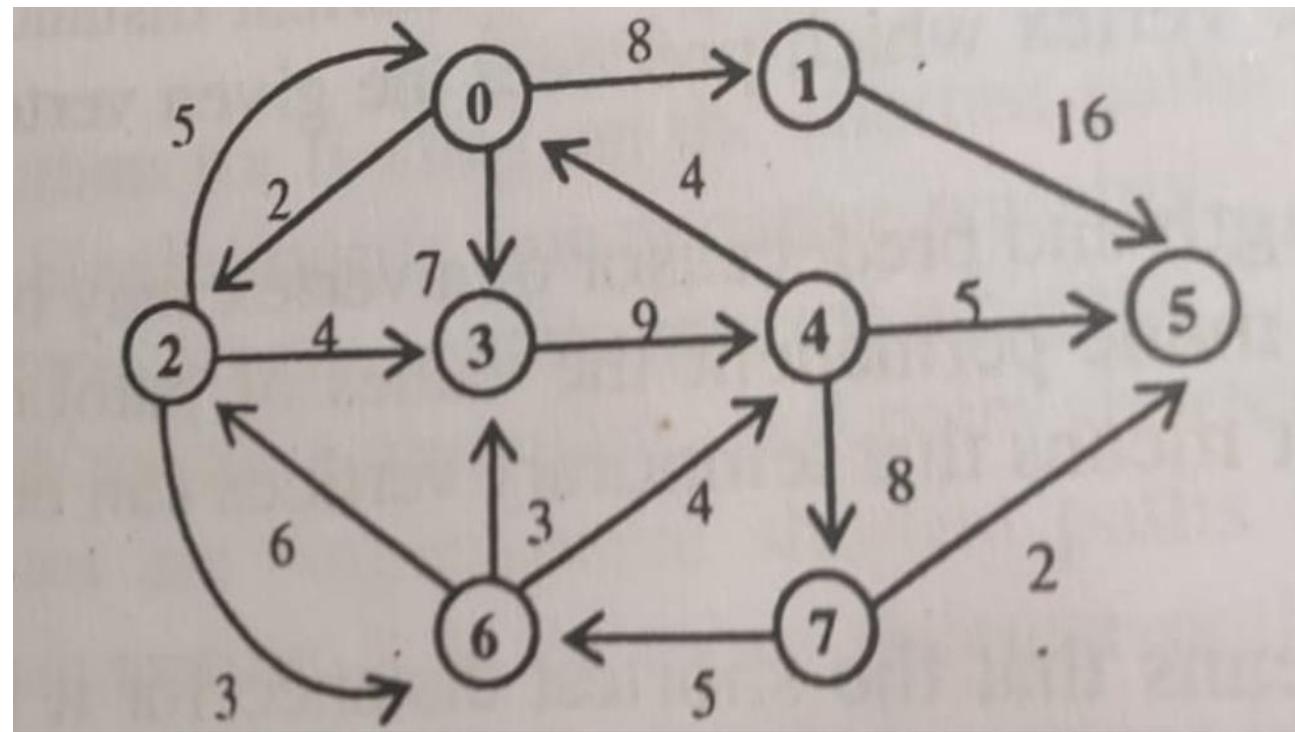
Dijkstra's algorithm

If $\text{pathlength}(\text{current}) + \text{weight}(\text{current}, v) \geq \text{pathlength}(v)$

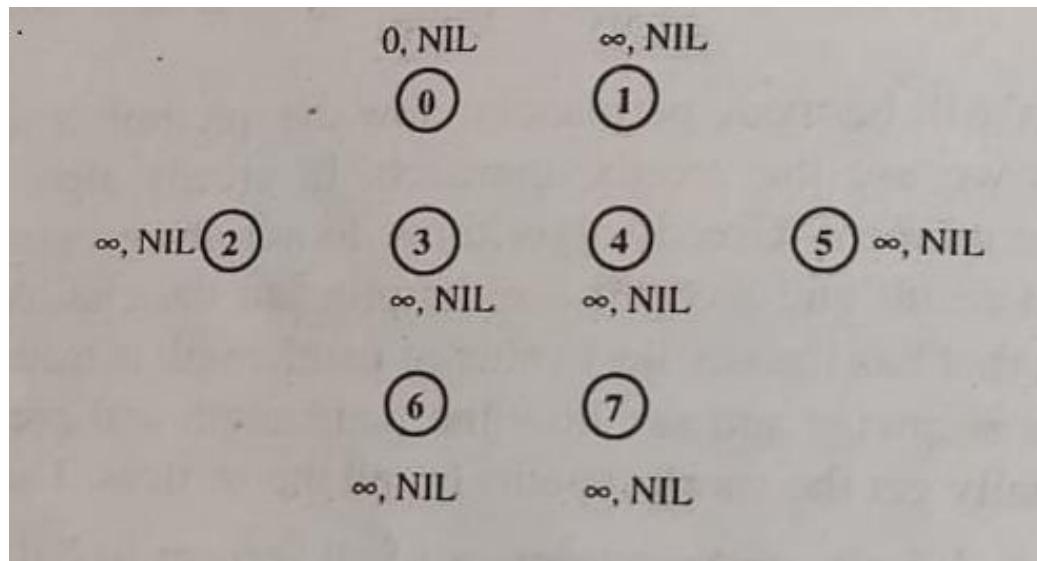
In this case vertex v is not relabelled and the values of pathlength and predecessor for vertex v remains unchanged.

E. Repeat steps C and D until there is no temporary vertex left in the graph

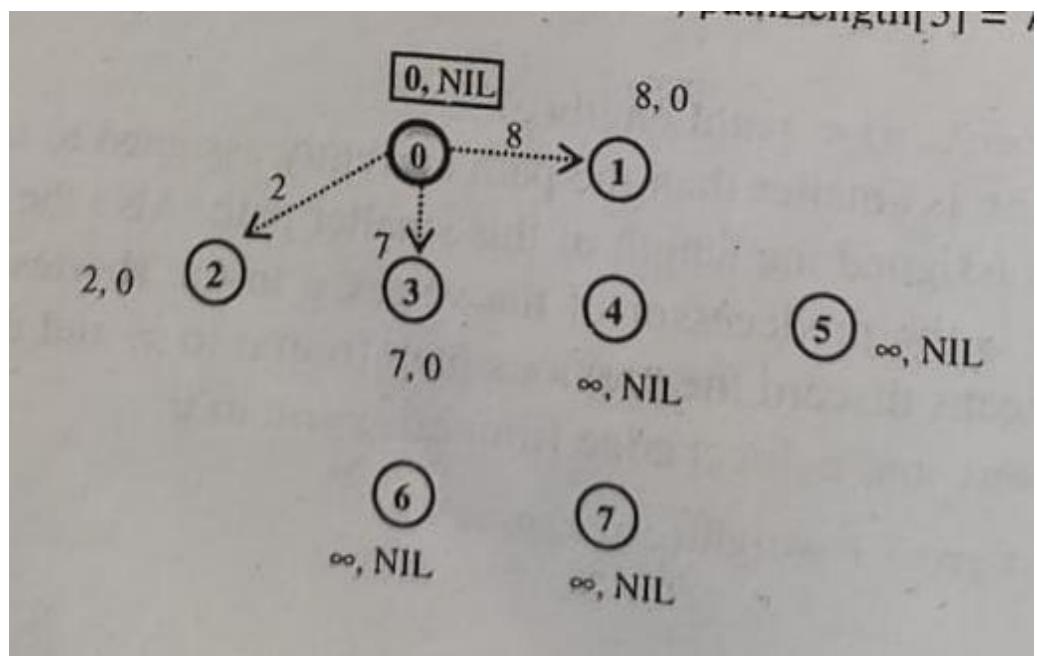
Dijkstra's algorithm on the graph below



1



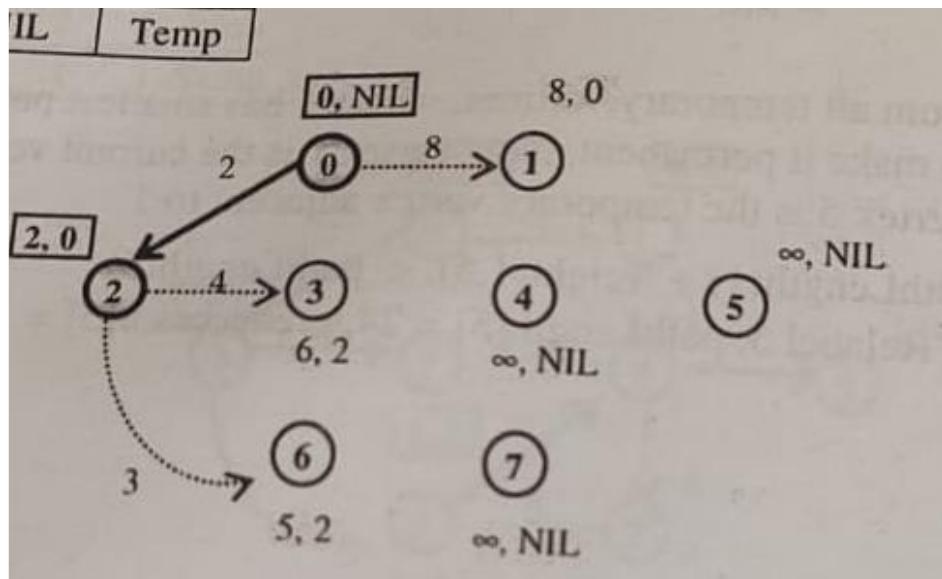
2



| Vertex | path Length | prede-cessor | status |
|--------|-------------|--------------|--------|
| 0 | 0 | NIL | Temp |
| 1 | ∞ | NIL | Temp |
| 2 | ∞ | NIL | Temp |
| 3 | ∞ | NIL | Temp |
| 4 | ∞ | NIL | Temp |
| 5 | ∞ | NIL | Temp |
| 6 | ∞ | NIL | Temp |
| 7 | ∞ | NIL | Temp |

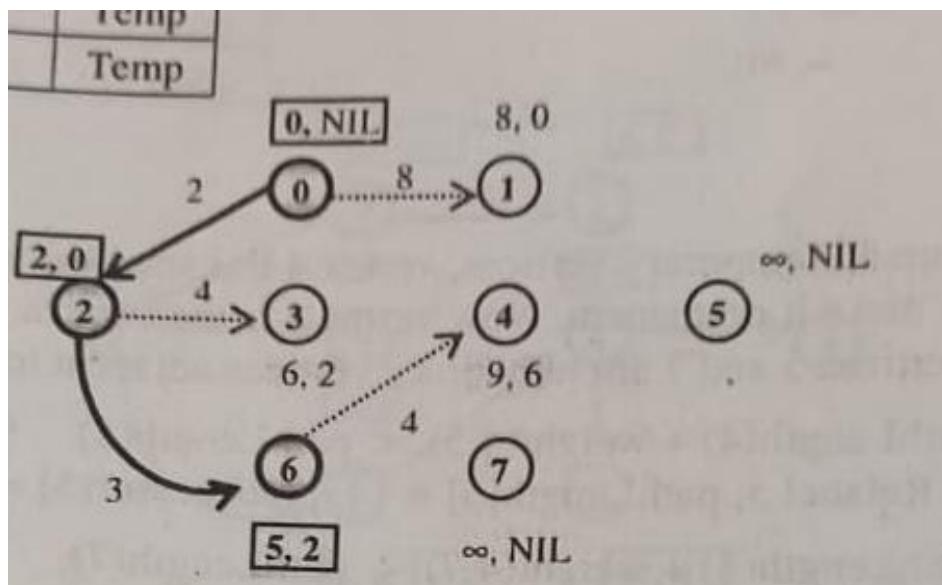
| Vertex | path Length | prede-cessor | status |
|--------|-------------|--------------|--------|
| 0 | 0 | NIL | Perm |
| 1 | 8 | 0 | Temp |
| 2 | 2 | 0 | Temp |
| 3 | 7 | 0 | Temp |
| 4 | ∞ | NIL | Temp |
| 5 | ∞ | NIL | Temp |
| 6 | ∞ | NIL | Temp |
| 7 | ∞ | NIL | Temp |

3

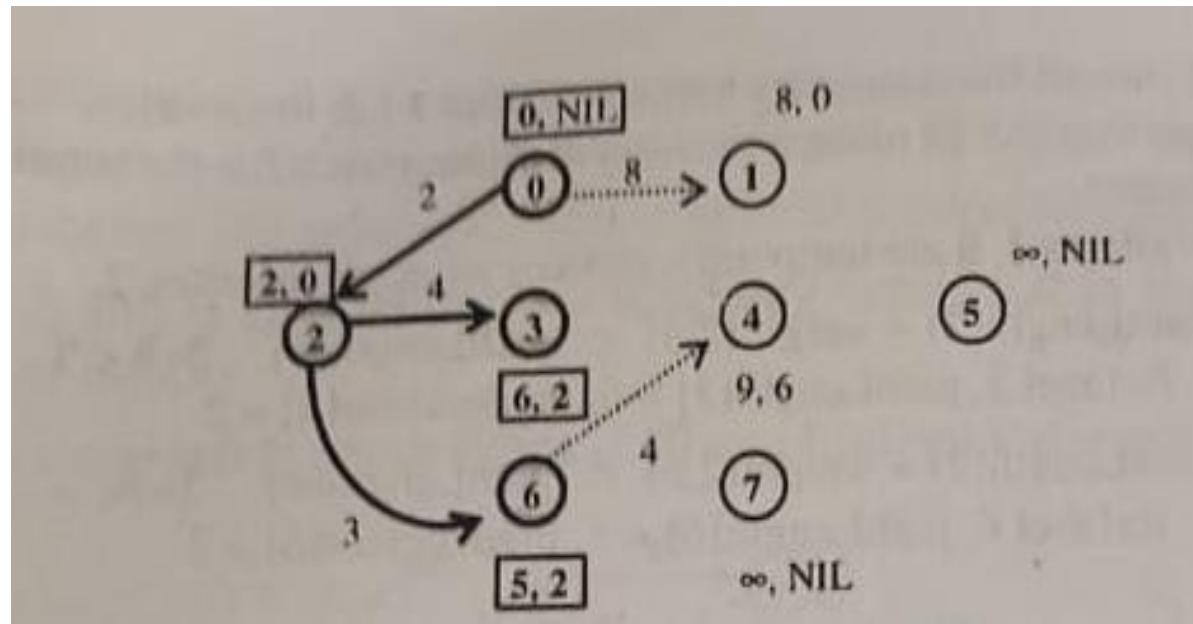


| Vertex | path Length | prede-cessor | status |
|--------|-------------|--------------|--------|
| 0 | 0 | NIL | Perm |
| 1 | 8 | 0 | Temp |
| 2 | 2 | 0 | Perm |
| 3 | 6 | 2 | Temp |
| 4 | ∞ | NIL | Temp |
| 5 | ∞ | NIL | Temp |
| 6 | 5 | 2 | Temp |
| 7 | ∞ | NIL | Temp |

4

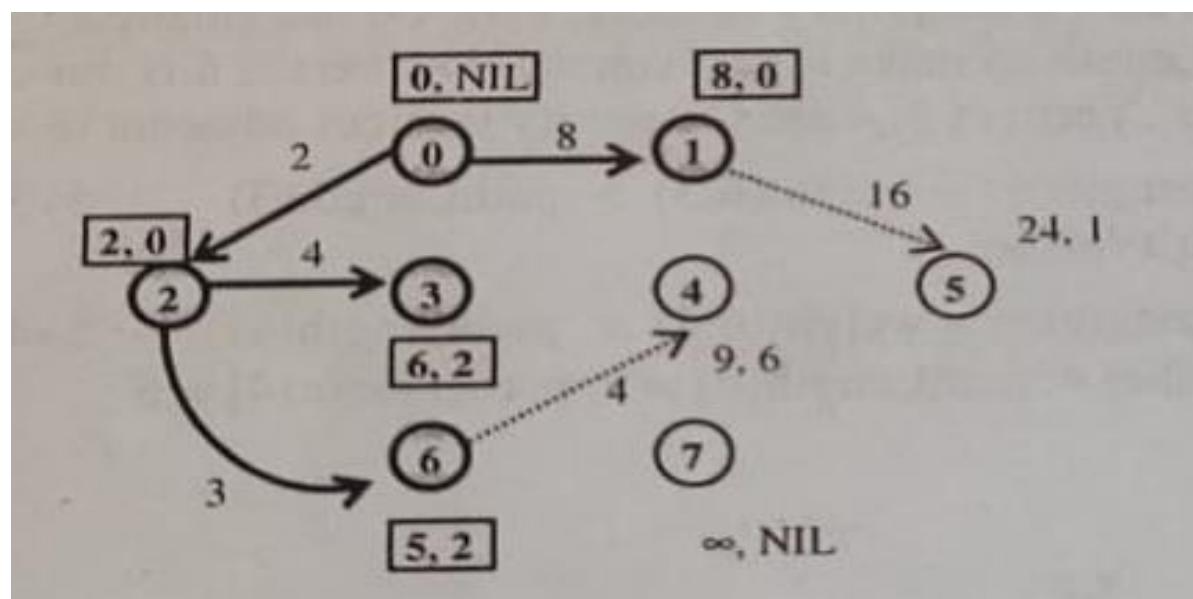


| Vertex | path Length | prede-cessor | status |
|--------|-------------|--------------|--------|
| 0 | 0 | NIL | Perm |
| 1 | 8 | 0 | Temp |
| 2 | 2 | 0 | Perm |
| 3 | 6 | 2 | Temp |
| 4 | 9 | 6 | Temp |
| 5 | ∞ | NIL | Temp |
| 6 | 5 | 2 | Perm |
| 7 | ∞ | NIL | Temp |



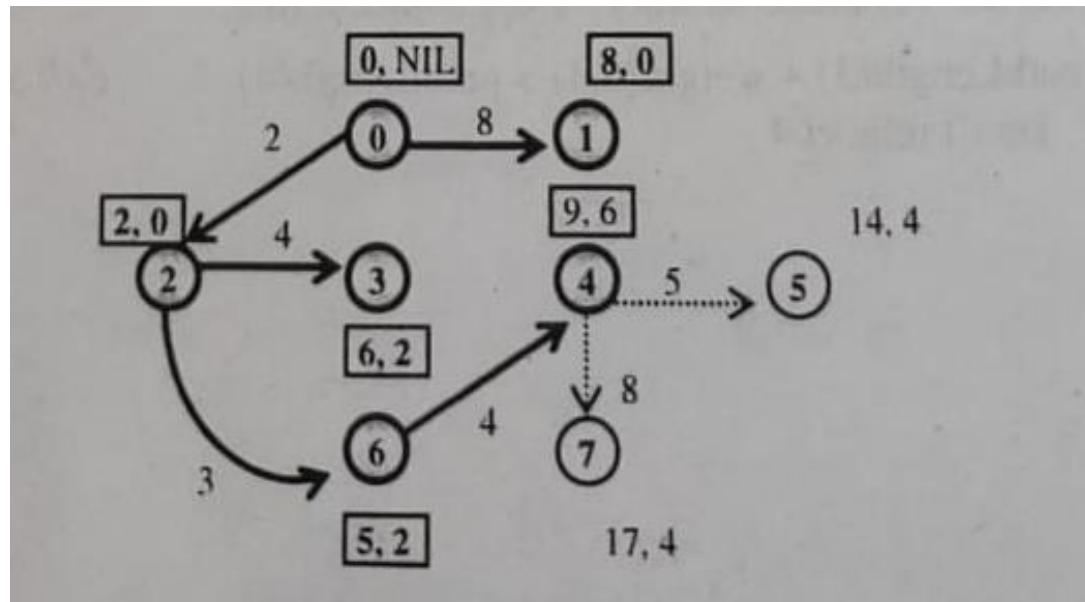
5

| Vertex | path Length | prede-cessor | status |
|--------|-------------|--------------|--------|
| 0 | 0 | NIL | Perm |
| 1 | 8 | 0 | Temp |
| 2 | 2 | 0 | Perm |
| 3 | 6 | 2 | Perm |
| 4 | 9 | 6 | Temp |
| 5 | ∞ | NIL | Temp |
| 6 | 5 | 2 | Perm |
| 7 | ∞ | NIL | Temp |

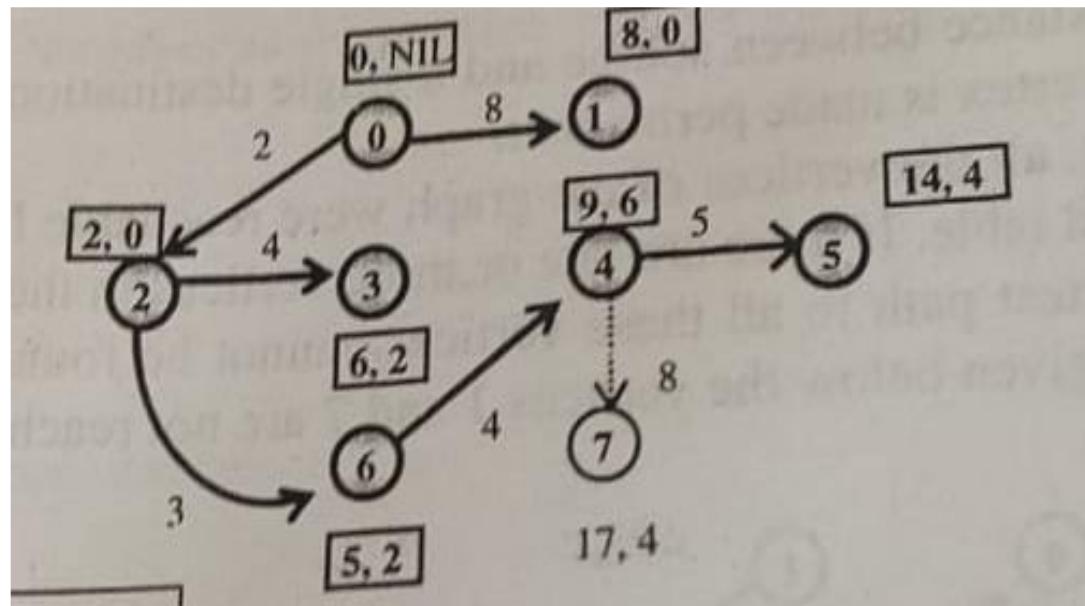


6

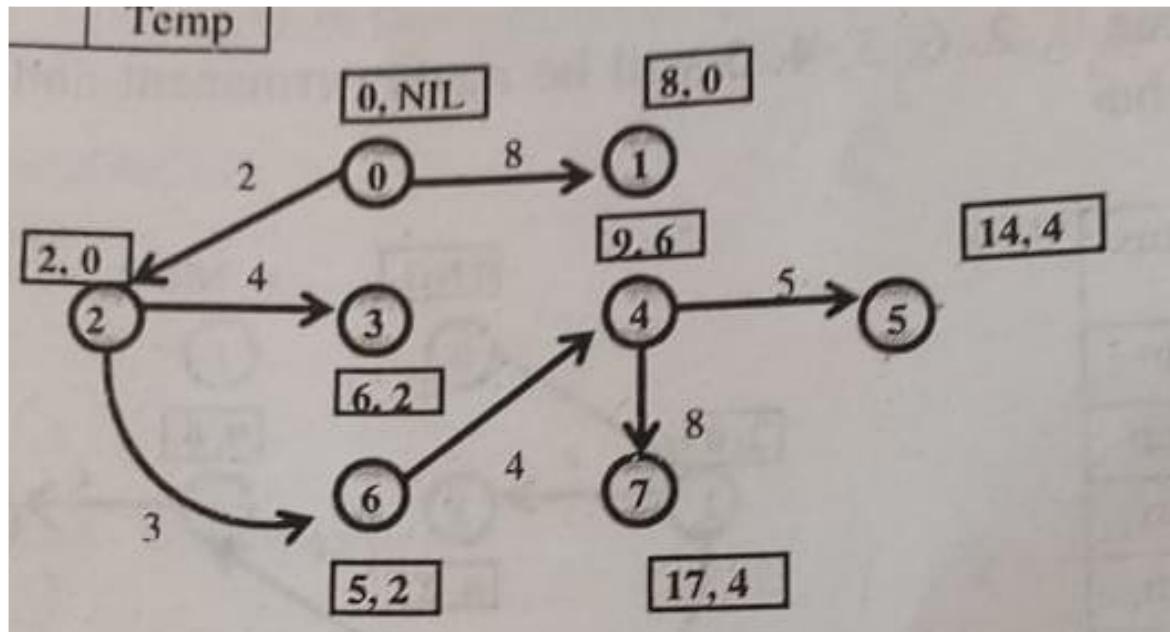
| Vertex | path Length | prede-cessor | status |
|--------|-------------|--------------|--------|
| 0 | 0 | NIL | Perm |
| 1 | 8 | 0 | Perm |
| 2 | 2 | 0 | Perm |
| 3 | 6 | 2 | Perm |
| 4 | 9 | 6 | Temp |
| 5 | 24 | 1 | Temp |
| 6 | 5 | 2 | Perm |
| 7 | ∞ | NIL | Temp |



| Vertex | path Length | prede-cessor | status |
|--------|-------------|--------------|--------|
| 0 | 0 | NIL | Perm |
| 1 | 8 | 0 | Perm |
| 2 | 2 | 0 | Perm |
| 3 | 6 | 2 | Perm |
| 4 | 9 | 6 | Perm |
| 5 | 14 | 4 | Temp |
| 6 | 5 | 2 | Perm |
| 7 | 17 | 4 | Temp |



| Vertex | path Length | prede-cessor | status |
|--------|-------------|--------------|--------|
| 0 | 0 | NIL | Perm |
| 1 | 8 | 0 | Perm |
| 2 | 2 | 0 | Perm |
| 3 | 6 | 2 | Perm |
| 4 | 9 | 6 | Perm |
| 5 | 14 | 4 | Perm |
| 6 | 5 | 2 | Perm |
| 7 | 17 | 4 | Temp |



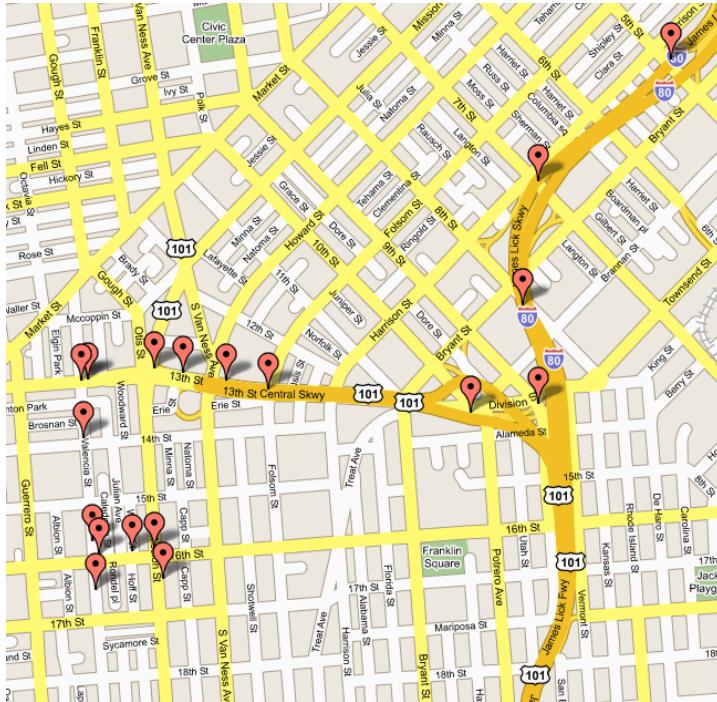
| Vertex | path Length | prede-cessor | status |
|--------|-------------|--------------|--------|
| 0 | 0 | NIL | Perm |
| 1 | 8 | 0 | Perm |
| 2 | 2 | 0 | Perm |
| 3 | 6 | 2 | Perm |
| 4 | 9 | 6 | Perm |
| 5 | 14 | 4 | Perm |
| 6 | 5 | 2 | Perm |
| 7 | 17 | 4 | Perm |

DIJKSTRA'S ALGORITHM - WHY USE IT?

- As mentioned, Dijkstra's algorithm calculates the shortest path to every vertex.
- However, it is about as computationally expensive to calculate the shortest path from vertex u to every vertex using Dijkstra's as it is to calculate the shortest path to some particular vertex v .
- Therefore, anytime we want to know the optimal path to some other vertex from a determined origin, we can use Dijkstra's algorithm.

Applications of Dijkstra's Algorithm

- Traffic Information Systems are most prominent use
- Mapping (Map Quest, Google Maps)
- Routing Systems



From Computer Desktop Encyclopedia
© 1998 The Computer Language Co. Inc.

Router A Routing Table

| To go to network: | Route via port #: |
|-------------------|-------------------|
| 10.0.0.0 | 1 |
| 20.0.0.0 | 2 |
| 30.0.0.0 | 3 |
| 40.0.0.0 | 1 |

