Jaewan

Ps-28

SUNBEAM
Institute of Information Technology
Placement Initiative

Data Structures and Algorithms

Data Structure:

- Structure of Data
  - The organization of data in memory.
  - Operations performed on that data.
- Types of data structures:
  - Linear - Array, Stack, Queue, Linked List.
  - Non-Linear - Tree, Graph
  - Associative:- data is stored in key value pair e.g. Hash Table
- Usually data structures are represented as "Abstract Data Types".
- Example Data Structures:
  - Array, Stack, Queue, Linked List, Tree, Graph, Hash Table, etc.

Algorithm:

- An algorithm is a **finite set of instructions**, if followed, accomplishes a particular task.
- Algorithms are written to solve problems. One problem may have multiple solutions, so we need to decide which algorithm is best amongst them to solve a particular problem.
- There are two main measures to decide the **efficiency** of an algorithm: **time** and **space.**
- **Analysis or Performance of an algorithm** refers to the task of determining how much **computing time** (time) and **storage** (space) requires running to completion.

Space Complexity

- Space Complexity of an algorithm is **the amount of memory it needs to run to completion.**
- Space required by a program = **code space** (instructions resides in memory) + **data space** (space taken by variables and constants)
- The space required has two components:
  - **Fixed component:** it is independent of instance characteristics. It includes **instruction space, space required by simple variables, aggregate variables and constants.**
  - **Variable Component:** is depends on **instance characteristics** such as size of the input . It includes the space needed by **reference variables** and **recursion stack.**
- The space required S(P) of any algorithm/program P may therefore be written as: S(P) = C + S(instance characteristics) = C + $S_P(n)$; where C is a constant and n is the input size.

```
int sum(int arr[],int n) //arr is an array of size n
{
      int s = 0;
      for(int i = 0; i < n; i++)
            a = s + arr[i];
      return s;
}
```

- **Space required = Code space + Space required by variables**

Scanned by CamScanner

- $S_P(n)$ = Space required by variables
- $S_P(n)$ = n (for arr) + 1 (for n) + 1 (for s) + 1 (for i) + 2 for(0 and 1)
- $S_P(n) = n + 5$
- $S_P(n) >= n$

- **Space Complexity for Recursive Algorithms:**

```
int Rsum(int arr[], int n)//recursive: where arr is an array of
size n
{
      if(n <= 0)
            return 0;
      else
            return Rsum(arr,n-1) + arr[n-1];
}
```

- **Space required = code space + space required by variables + stack space.**
  - Recursion stack space includes space for **formal parameters, local variables** and **return address** = 1 (for n) + 1 (for pointer to A) + 1 (for return address)
  - The depth of recursion is = n+1
  - **S_P>= 3(n+1)**
- **Recursive algorithms have very high space complexity.**

Time Complexity

- Time Complexity of an algorithm is **the amount of computer time it needs to run to completion.**
- It is the sum of **compile time** (fixed components) + **run or execution time**(depends on instance characteristics).
- The execution time of a statement depends on **type of operations involved, values, the type of machine** and **the type of environment in which the statement is executed.**
- **Best Case Analysis:** If algorithm takes **minimum time** to solve the problem for given set of input, it is called as best case time complexity.
- **Worst Case Analysis:** If algorithm takes **maximum time** to solve the problem for a given set of input is is called as worst case time complexity, it is the special case of interest for the mathematics.
- **Average Case Analysis:** Input sequence which is neither best nor worst is called as average case. Average case analysis represents general behavior of an algorithm.
- In most of the cases average and worst cases has same magnitude.

- **Asymptotic Notations** are the mathematical tool to find time or space complexity of an algorithm without implementing it in programming language.
  - This measure is independent of machine specific constants.
  - Asymptotic notation is a way of describing major component of the cost of entire algorithm.
  - Primitive operation is the major component of the cost.

Scanned by CamScanner

- Asymptotic notations is a way of describing major component of the cost of entire algorithm

- **Assumptions while doing complexity analysis:**
  - Actual cost of operation is not considered.
  - Abstract cost c is ignored: **$O(c.n^2)$ reduces to $O(n^2)$**
  - Only leading term of polynomial is considered: **$O(n^3 + n)$ reduces to $O(n^3)$**
  - Drop multiplicative or divisive constant if any: **$O(2n^2)$ and $O(n^2/2)$ both reduces to $O(n^2)$**
  - Various notations like **Big Oh (O)**, **Big Omega (Ω)andBig Theta (Θ)** are used to describe the asymptotic running time of an algorithm.
  - Above notations are useful in algorithm analysis to find **best, worst** and **averagecomplexity** of algorithms in terms of **primitive operations.**

- **Big Oh notation (O):** It is used to define an **asymptotic upper bound for algorithm,** it means running time of algorithm cannot be more than its asymptotic upper bound for any random sequence of data.
  - **Definition:** A function **f(n)** is a member of **O(g(n))** or we write **f(n) = O(g(n))** or say **f(n) is of O(g(n))**, if there exists positive constants c and n_0 such that, **0 <= f(n) <= c.g(n) for all n >= n_0.**
  - **O(g(n)) = { f(n): there exists a positive constant c and n_0 such that 0 <= f(n) <= c.g(n) for all n >= n_0 }.**
  - **O -notation** is used to describe an upper bound for the **worst case running time** of an algorithm.

- **Big Omega notation(Ω):** It is used to define an **asymptotic lower bound.**
  - **Ω(g(n)) = { f(n): there exists positive constants c and n_0, such that 0 <= c.g(n) <= f(n) for all n >= n_0 }.**
  - **Ω-notation** is used to describe a lower bound for the **best case running time** of an algorithm.
  - **Insertion Sort:** is **$O(n^2)$** and is **Ω(n).**

- **Big Theta notation (Θ):** There is separate notation for **asymptotically tight bounds;Θ-notation** is used to define an **asymptotic tight bound.**
  - We say g(n) is a tight bound for f(n), if there exists positive constants c1,c2 and n_0 such that **0 <= c1.g(n) <= f(n) <= c2.g(n)** for all n >= n_0. It is denoted as **f(n) =Θ(g(n)).**

Scanned by CamScanner

- **Definition:** A function **f(n)** is a member of **Θ(g(n))** or we write **f(n) = Θ(g(n))** or we say **f(n) is of Θ(g(n))** if there exists a positive constants c and n_0 such that, **0 <= c1.g(n) <= f(n) <= c2.g(n)** for all n >= n_0.
- **Θ(g(n)) = { f(n): there exists positive constants c1,c2 and n_0 such that 0 <= c1.g(n) <= f(n) <= c2.g(n), for all n >= n_0}.**
- **f(n) is equal to g(n) within a constant factor, the function f(n) must be non-negative.f(n) = Θ(g(n)).**
- The function g is both, an upper as well as lower bound for f.
- Describes the growth of the function more accurately than **O** and **Ω.**
- The running time of an algorithm is **Θ(g(n)) iff (if and onlyif)**Its worst case running time is **O(g(n))** and Its best case running time is **Ω(g(n)).**

Analysis of Iterative Programs

- **O(1)**
  - time complexity of a function(or set of elements) is considered as O(1), if it doesn't contain a loop, recursion and call to any other non-constant time function.
  - **//set of non-recursive and non-loop statements**
  - e.g. **swap()** function has **O(1)** time complexity
  - A loop or recursion that runs a constant number of times is also considered as **O(1),** e.g. the following loop is: O(1)

```
//here c is constant
for( int i = 1; i <= c ; i++ ){
      //some O(1) expressions
}
```

- **O(n)**
  - time complexity of a loop is considered as O(n) if the loop variable is **incremented/decremented** by a constant amount. For example following function has O(n) time complexity.

```
//here c is any positive integer constant
for( int i = 0 ; i <= n ; i += c ){
      //some O(1) expressions
}
OR
for(int i = n ; i > 0 ; i  -=  c ){
      //some O(1) expressions
}
```

- **$O(n^c)$:**

Scanned by CamScanner

- time complexity of nested loops is equal to the number of times the innermost statement is executed. For example the following sample loops have **O(n^2)** time complexity.

```
for(int i = 0 ; i <= n ; i += c) //-> loop runs n times
{
        for(int j = 0 ; j <= n ; j += c ){ // -> loop runs n times
                //some O(1) expressions
        }
}
for(int i = n ; i > 0 ; i -= c ) // -> loop runs n times
{
        for(int j = i+1 ; j <= n ; j += c ){// -> loop runs n times
          //some O(1) expressions
        }
}
```

- for example **selection sort** and **insertion sort** have **O(n^2)** time complexity.

- **O(log n):**
  - time complexity of a loop is considered as **O(log n)** if the loop variables **divided/multiplied** by a constant amount.

```
for(int i = 0 ; i <= n ; i *= c ){
  //some O(1) expressions
}
for(int i = n ; i > 0 ; i /= c ){
  //some O(1) expressions
}
```

  - e.g. **binary search** has **O(log n)** time complexity.

- **O(log(log n)):**
  - time complexity of a loop is considered as **O(log(log n))** if the loop variables is **reduced/increased exponentially** by a constant amount.

```
//here c is a constant greater than 1
for(int i = 2 ; i <= n ; i *= pow(i,c) ) {
    //some O(1) expressions
}
//here fun is sqrt or cuberoot or any other constant
root
for(int i = n ; i > 0 ; i = fun(i) ){
    //some O(1) expressions
}
```


Scanned by CamScanner

- **How to combine time complexities of consecutive loops?**

When there are consecutive loops, we calculate time complexity as sum of the time complexities of indivisual loops.

```
for(int i = 1 ; i <= m ; i += c ){
    //some O(1) expressions
}
for(int i = 1 ; i <= n ; i += c ){
    //some O(1) expressions
}
```

Time complexity of above code is **O(m) + O(n)** which is **O(m+n)**

If **m == n**, the time complexity becomes **O(2n)** which is **O(n)**.

Scanned by CamScanner

SEARCHING TECHNIQUES:

1. Linear Search:

- It is also known as **sequential search.**
- Suppose **A** is an array of size **n**, we want to search an element from this array. Let us call this element as **key.**
- Linear Searching is very simple way of searching. Key element is compared with one by one all index location of A.
- Algorithm halts in two cases: key element is found or entire array is scanned.

```
int LinearSearch(int A[],int key)
{
   //A is an array of size n, and key is the element to be
search
   for( int index = 0 ; index < n ; index++ )
   {
         if A[index] == key
         then
               return index;
   }
   return -1;
}
```

- **Complexity Analysis:**

Best Case:

- Algorithm needs minimum number of comparisons if key element is found on very first location.
- In best case, size of input array does not matter. If key element is on the first position it takes only one comparison irrespective of array length. Hence the running time of linear search in best case is, **T(n) = 1.**

Worst Case:

- Algorithm performs maximum number of comparisons if key element is on the last location or it is not present at all. Entire array needs to be scanned.
- No. of comparisons linearly grow with the size of input. Hence the running time of linear search in worst case is, **T(n) = O(n).**

Average Case:

- Average case occurs when element is neither on first location nor on last. Key element may be near to beginning or may be near to end, or it may be somewhere near to middle. So on an average, algorithm does (n/2) comparisons.
- **T(n) = O(n/2) => O(n).**

2. Linear Search Recursive:

```
int RecLinearSearch(int A[], int index, int key)
{
   // A is the array of size n, key is the ele to be search
```


Scanned by CamScanner

```
    if(   index == n )
          retrun -1;
    if( A[index] == key )
          return index;
    else
          returnRecLinearSearch(A,index-1,key);
}
```

3. Binary Search:

- Binary search is **efficient** than linear search.
- For binary search array must be sorted, which is not required in linear search.
- Binary search reduces search space by half in every iteration, whereas in linear search, search space is reduced by one in each iteration.
- If there are **n** elements in an array, binary search and linear search has to search among (n/2) and (n-1) elements respectively in second iteration.
- In third iteration, binary search has to scan only (n/4) elements, whereas linear search has to scan (n-2) elements. This shows that binary search would hit the bottom very quickly.

```
int BinarySearch(int A[],int left,int right,int key)
{
   while( left <= right )
   {
         mid = (left + right)/2;
         if( A[mid] == key )
               return mid;
         if( key < A[mid] )
               right = mid-1;
         else
               left = mid+1;
   }
   return -1;
}
```

- **Complexity Analysis:**

Best Case:

- On very first attempt, a key element is compared with middle element of an array. If key element is on middle position of array, algorithm does only one comparison, irrespective of size of array.
- Hence best case running time of algorithm would be, **T(n) = O(1).**

Worst Case:

- Search space is reduced by half in every iteration, so maximum **$\log_2 n$** divisions are possible. If element is at leaf of the tree or it's not present at all, then algorithm does **$\log_2 n$** comparisons, which are maximum. No. of comparisons grow in logarithmic order of input size, Hence, worst case running time of algorithm would be, **T(n) = O($\log_2 n$)**

Average Case:

Scanned by CamScanner

- Average case for binary search occurs when key element is neither on middle nor at the leaf level of the tree. On an average, it does half of the $\log_2 n$ comparisons, which will turn out as, **$T(n) = O(\log_2 n)$**

Recurrence Equation for Binary Search and Running time of the recurrence is $O(\log_2 n)$:
In each iteration, binary search does one comparison and creates the new problem of size n/2.
So recurrence equation of binary search is given as,
$T(n) = T(n/2) + 1, n > 1$
$T(n) = 1, n = 1$ i.e. only one comparison is needed when there is only one element in a array, that's the trivial case.

This is the boundary condition for recurrence.
Let us solve this by iterative approach,
$T(n) = T(n/2) + 1$ ..... (I)
put n = n/2 in equation (I) to find T(n/2)
$\Rightarrow T(n/2) = T(n/4) + 1$ .... (II)
substitute value of T(n/2) in equation (I)
$\Rightarrow T(n) = (T(n/4) + 1) + 1$
$\Rightarrow T(n/2^2) + 2$ .....(III)
put n = n/2 in equation (II) to find T(n/4)
$\Rightarrow T(n/2/2) = T(n/2/4) + 2 + 1$
$\Rightarrow T(n/4) = T(n/8) + 1$
substitute value of T(n/4) in equation (III),
$T(n) = T(n/8) + 3$
.
.
.
after k iterations
$T(n) = T(n/2^k) + k$
suppose $n = 2^k$
$\Rightarrow \log n = \log 2^k$ ..... by taking log on both sides
$\Rightarrow k = \log_2 n$
$T(2^k) = T(2^k/2^k) + k \Rightarrow T(1) + k$
$\Rightarrow T(1) = 1$
So, $T(n) = 1 + k \Rightarrow T(n) = 1 + \log_2 n \Rightarrow T(n) = O(\log_2 n)$

SORTING ALGORITHMS:

SELECTION SORT:

- Selection sort is a sorting algorithm, specifically an **in-place comparison sort.**
- It has **$O(n^2)$** time complexity, making it **inefficient** on large lists, and generally performs worse than the similar insertion sort.
- Selection sort is noted for its **simplicity**, and it has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

Scanned by CamScanner

- The algorithm divides the input list into two parts: the sub list of items already sorted, which is built up from left to right at the front (left) of the list, and the sub list of items remaining to be sorted that occupy the rest of the list.
- Initially the sorted sub list is empty and the unsorted sub list is the entire input list.
- The algorithm proceeds by finding the smallest (or largest depending on sorting order) element in the unsorted sub list, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sub list boundaries one element to right.

Time Complexity:

- Worst Case Time Complexity: $O(n^2)$
- Average Case Time Complexity: $O(n^2)$
- Best Case Time Complexity: $O(n^2)$

BUBBLE SORT:

- Bubble sort, sometimes referred to as **Sinking Sort.**
- It is a simple sorting algorithm that repeatedly steps through the list to be sorted, **compares each pair of adjacent items and swaps them if they are in the wrong order.** The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.
- The algorithm, which is a comparison sort, is named for the way smaller or larger elements "bubble" to the top of the list.
- Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort.

Complexity:

- Worst Case Time Complexity: O(n2)
- Average Case Time Complexity: O(n2)
- Best Case Time Complexity: O(n)

INSERTION SORT:

- Insertion Sort is a **simple** sorting algorithm, that builds the final sorted array (or list) one item at a time.
- When people manually sort cards in a bridge hand, most use a method that is similar to insertion sort.
- It is **much less efficient on large lists** than more advanced algorithms such as quick sort, merge sort and heap sort.

Advantages:

- simple implementation
- **efficient for (quite) small data sets**, much like other quadratic sorting algorithms.
- More efficient in practice than most other simple quadratic (i.e. $O(n^2)$) algorithms such as selection sort or bubble sort.
- **Adaptive:** i.e. efficient for data sets that are already substantially sorted.

Scanned by CamScanner

- **Stable:** i.e. does not change the relative order of elements with equal keys
- **In-place:** i.e. only requires constant amount **O(1)** of additional memory space.
- **Online:** i.e. can sort a list as it receives it.

Best Case Time Complexity: O(n)
The best case input is an array that is already sorted.
Worst Case Time Complexity: $O(n^2)$
The simplest worst case input is an array sorted in reverse order
Average Case Time Complexity: $O(n^2)$
However, insertion sort is one of the fastest algorithm for sorting very small arrays, even faster than quick sort; indeed good quick sort implementations uses insertion sort for arrays smaller than certain threshold

QUICK SORT:

- Quick Sort is **divide-and-Conquer** algorithm.
- Quick Sort first divides a large array into two smaller sub arrays: the low elements and high elements.
- Quick Sort can then recursively sort sub-arrays.

Steps are:
1. Pick an element, called **pivot**, from the array
2. **Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater that the pivot come after it. (equal value can go either way). After this partitioning, the pivot is in its final position. This is called a partitioning operation.**
3. Recursively apply the above steps to the sub array of elements with smaller value and separately to the sub-array of elements with greater values.

- **The base case of recursion is array of size zero or one,** which never need to sorted.
- The pivot selection and partitioning steps can be done in several different ways: the choice of specific implementation schemes greatly affects the algorithm's performance.
- There are many different versions of quick sort that pick pivot in different ways.
  1. always pick first element as a pivot
  2. always pick last element as a pivot
  3. pick a random element as pivot
  4. pick median as a pivot

```
Algorithm QuickSort(int A[],int left,int right)
{
  int i = left;
  int j = right;

  //recursive calling termination condition
  if (left >= right)
       return;
  while( i < j )
  {
       While( i <= right && A[i] <= A[left] )
```

Scanned by CamScanner

```
            i++;
        while( A[j] > A[left])
            j--;
        //if i and j are not crossed then swap them
        If( i < j )
            SWAP(A[i],A[j]);
    }//end of while loop

    //swap the pivot element with jth pos element
    SWAP(A[left], A[j]);
    //apply quick sort on left sub array
    QuickSort(A,left,j-1);
    //apply quick sort on right sub array
    QuickSort(A,j+1,right);
}//end of algorithm
```

Analysis of Quick Sort:

- Time taken by Quick Sort in general can be written as following:

$T(n) = T(k) + T(n-k-1) + \Theta(n)$.

- The first two terms are for two recursive calls, the last term is for the partition process, k is the number of elements which are smaller than pivot.
- the time taken by Quick Sort depends upon **the input array** and **partition strategy.**

Worst Case: $O(n^2)$

- **The worst case occurs when the partition process always picks greatest or smallest element as pivot.** If we consider partition strategy where first element always picked as a pivot, the worst case would occur when the array is already sorted in ascending or descending order.
- Following is the recurrence for worst case:

  $\Rightarrow T(n) = T(0) + T(n-1) + \Theta(n)$

  $\Rightarrow T(n) = T(n-1) + \Theta(n)$

the solution of above recurrence is $\Theta(n^2)$

Best Case: $\Theta(n \text{ Log } n)$

- **the best case occurs when the partition process always picks the middle element as a pivot.**
- Following is the recurrence for the best case

  $T(n) = 2T(n/2) + \Theta(n)$
- the solution of above recurrence is **$\Theta(n \text{ Log } n)$**, can be solved by using master theorem.

Average Case: $O(n \text{ Log } n)$

- To do average case analysis, we need to consider all possible permutation of array and calculate time taken by every permutation which doesn't look easy.

Scanned by CamScanner

- We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case:

T(n) = T(n/9) + T(9n/10) + Θ(n)
solution of above recurrence is also **O(n Log n)**.

- Although the worst case time complexity of Quick Sort is **$O(n^2)$** which is more than many other sorting algorithms like Merge Sort and Heap Sort, **Quick Sort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real world data.**
- Quick Sort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data.
- However, **Merge Sort is generally considered better when data is huge and stored in external storage.**

Stack (ADT):

- Operations:
  - Push()
  - Pop()
  - Peek()
  - IsEmpty()
  - IsFull()
- Data Organization In Memory:
  - Using Arrays:
    - int arr[MAX];
    - int top;
  - Using Linked List;
    - node *head;
    - Push() ==>AddFirst();
    - Pop() ==>DelFirst();
    - Peek() ==> return head->data;
    - IsEmpty() ==> return head==NULL;

Queue (ADT)

- Operations:
  - Push()
  - Pop()
  - Peek()
  - IsEmpty()
  - IsFull()
- Data organization in memory:
  - Using linked list (with head & tail pointer):
    - front => node *head;
    - rear => node *tail;
    - Push() =>AddLast()
    - Pop() =>DelFirst()
    - Peek() => return head->data;

Scanned by CamScanner

- IsEmpty() => return head==NULL;
- Using arrays:
  - int arr[MAX];
  - int front;
  - int rear;

Linear Queue:

```
1. Init()
   front = rear = -1;
2. Push()
   rear ++;
   arr[rear] = ele;
3. Pop()
   front ++;
4. Peek()
   return arr[front+1];
5. IsEmpty()
   front == rear
6. IsFull()
   rear==MAX-1
```

Circular Queue:

- In linear queue, when rear reaches MAX-1, even if few locations are vacant, it shows queue is full. In other words, in linear queue there is no proper utilization of memory.
- To overcome this limitation, circular queue is implemented.
- In circular queue, front & rear are incremented in circular fashion i.e. 0, 1, 2, ..., MAX-1, 0, 1, ...

Implementation 1

```
1. Init()
   front = rear = -1;
2. Push()
   rear = (rear + 1) % MAX;
   arr[rear] = ele;
3. Pop()
   front = (front + 1) % MAX;
   if(front==rear)
        front = rear = -1;
4. Peek()
   temp = (front + 1) % MAX;
   return arr[temp];
5. IsEmpty()
   front == rear && front == -1
6. IsFull()
   (front==-1 && rear==MAX-1) || (front==rear && front!=-1)
```


Scanned by CamScanner

Implementation 2 – Using count

```
1. Init()
   front = rear = -1;
   count = 0;
2. Push()
   rear = (rear + 1) % MAX;
   arr[rear] = ele;
   count++;
3. Pop()
   front = (front + 1) % MAX;
   count--;
4. Peek()
   temp = (front + 1) % MAX;
   return arr[temp];
5. IsEmpty()
   count == 0
6. IsFull()
   count == MAX
```

Types of Queues:

1. Linear Queue:
   a. No proper utilization of memory.
2. Circular Queue:
   a. Proper utilization of memory.
   b. Increments rear and front in circular fashion i.e. 0, 1, 2, 3, ..., MAX-1, 0, 1, ...
3. Priority Queue:
   a. Each element is associated with some priority.
   b. Element with highest priority is popped out first. (No FIFO behavior).
   c. Typically it is implemented as sorted linked list. While insertion element is added at appropriate position as per its priority. So insertion time complexity will be O(n). While deleting first (head) element is deleted (as it is element with highest priority).
4. Double Ended Queue (deque):
   a. Can push/pop elements from both sides i.e. front as well as rear.
   b. Usually implemented as doubly linked list with head & tail.

Stack Vs Queue:

	Stack	Queue
1	Stack is LIFO utility data structure.	Queue is FIFO utility data structure.
2	Push and Pop operations are done from the same end i.e. "top".	Push and Pop operations are done from different ends i.e. "rear" and "front" respectively.
3	There are no types of stack. However in single array you can implement two or more stacks (called as multi-stack approach).	There are four types of queues i.e. linear, circular, priority and deque.
4	Applications: • Function activation records are created on the stack	Applications: • Printer maintains queue of documents to be

Scanned by CamScanner

for each function call. • To solve infix expression by converting to prefix or postfix. • Parenthesis balancing • To implement algos like Depth First Search.	printed. • OS uses queues for many functionalities: Ready queue, Waiting queue, Message queue. • To implement algos like Breadth First Search.

STL: Standard Template Library:

`#include <stack>`
`stack<char> s;`
`s.push('A');`
`s.pop();`
`ele = s.top(); // s.peek()`
`s.empty();  // s.isempty()`
	`#include <queue>`
	`queue<char> q;`
	`q.push('A');`
	`q.pop();`
	`ele = q.front(); // q.peek()`
	`q.empty();  // q.isempty()`

Infix, Prefix, Postfix:

- These are notations to represent math equations.
  - Infix: A + B -> For human
  - Prefix: + A B -> For computers
  - Postfix: A B + -> For computers
- Infix expression:
  - 4 + 3 - (9 - 5) / 2 * 8 - 6
- To convert Infix to Prefix/Postfix considers the priorities:
  1. ( )
  2. $ or ^ i.e. Exponential Operator
  3. * / % (left to right)
  4. + - (left to right)

Infix to Postfix using stack:

- Traverse Infix string element by element from left to right.
- If operand is found, then append to the postfix string.
- If operator is found, then push it on the stack.
- If priority of topmost of stack is greater or equal to priority of current operator, pop it from the stack and append to postfix string. Repeat this step if required.
- When infix string is completed, pop all operators from the stack and append to postfix string (one by one).
- If opening '(' is found, then push it on the stack.
- If closing ')' is found, pop all operators from the stack and append to postfix string (one by one) until '(' is found on stack. Also pop and discard that '('.
- Example: 4 + 3 - (9 - 5) / 2 * 8 - 6

Symbol	Postfix	Stack
4	4	
+	4	+
3	4 3	+

Scanned by CamScanner

-	4 3 +	-
(	4 3 +	- (
9	4 3 + 9	- (
-	4 3 + 9	- (-
5	4 3 + 9 5	- (-
)	4 3 + 9 5 -	-
/	4 3 + 9 5 -	- /
2	4 3 + 9 5 - 2	- /
*	4 3 + 9 5 - 2 /	- *
8	4 3 + 9 5 - 2 / 8	- *
-	4 3 + 9 5 - 2 / 8 * -	-
6	4 3 + 9 5 - 2 / 8 * - 6	-
	4 3 + 9 5 - 2 / 8 * - 6 -	

Infix to Prefix using stack:

1. Traverse Infix string element by element from right to left.
2. If operand is found, then append to the prefix string.
3. If operator is found, then push it on the stack.
4. If priority of topmost of stack is greater than priority of current operator, pop it from the stack and append to prefix string. Repeat this step if required.
5. When infix string is completed, pop all operators from the stack and append to prefix string (one by one).
6. If closing ')' is found, then push it on the stack.
7. If opening '(' is found, pop all operators from the stack and append to prefix string (one by one) until ')' is found on stack. Also pop and discard that ')'.
8. Reverse prefix string.

Postfix Evaluation:

1. Traverse postfix string from left to right.
2. If operand is found, push it on the stack.
3. If operator is found, pop two operands from the stack; calculate result and push it on stack.
   a. First popped is second operand; while second popped is first operand.
4. Repeat steps 1-2, until postfix string is completed.
5. At the end, pop final result from the stack.
6. Example: 4 3 + 9 5 - 2 / 8 * - 6 -

Symbol	Stack	Operation
4	4	
3	4 3	
+	7	4 + 3 = 7
9	7 9	
5	7 9 5	
-	7 4	9 - 5 = 4
2	7 4 2	
/	7 2	4 / 2 = 2
8	7 2 8	
*	7 16	2 * 8 = 16
-	-9	7 - 16 = -9

Scanned by CamScanner

6	-9 6	
-	-15	-9 - 6 = -15

Prefix Evaluation:

1. Traverse prefix string from right to left.
2. If operand is found, push it on the stack.
3. If operator is found, pop two operands from the stack; calculate result and push it on stack.
4. First popped is first operand; while second popped is second operand.
5. Repeat steps 1-2, until prefix string is completed.
6. At the end, pop final result from the stack.

Linked List:

- **Terminologies:**
  - Linked list is a linear data structure that contains multiple records linked to each other.
  - Each item in the list is called as "Node".
  - Each node contains data and pointer (address) to the next node.
  - Typically linked lists are implemented as self-referential structure/class. The structure/class contains pointer of the same type to hold address of next node.
- **LinkedList (ADT):**
  - The commonly supported operations in a linked list is called as:
    - A. AddFirst()
    - B. AddLast()
    - C. InsertAtPos()
    - D. Search()
    - E. DeleteFirst()
    - F. DeleteLast()
    - G. DeleteAtPos()
    - H. Clear()
    - I. Sort()
    - J. Reverse()
- Linked list in C++:

```
class list;
class node
{
  private:
      int data;
      node *next;
  public:
      node(intval=0);
      friend class list;
};
class list
{
private:
      node *next;
   public:
      list();
      ~list();
      addfirst(intval);
```


Scanned by CamScanner

```
        addlast(intval);
        insert(intval, intpos);
        void delfirst();
        void dellast();
        void del(intpos);
        node* search(int key);
        void sort();
        void reverse();
        void clear();
};
```

- Types of Linked Lists:
  - Singly Linear Linked List
    - A. Singly Linked List – only head pointer:
      1. Add First:O(1)

```
// create and initialize node
newnode = new node(val);
// if list is empty, newnode is first node.
if(head==NULL)
      head = newnode;
else
{
      //newnode's next --> head
      newnode->next = head;
      // head -->newnode
      head= newnode;
}
```

      2. Add Last: O(n)

```
// create and initialize node
newnode = new node(val);
// if list is empty, newnode is first node.
if(head==NULL)
      head = newnode;
else
{
      // traverse till last node
      trav = head;
      while(trav->next!=NULL)
            trav = trav->next;
      // trav's next -->newnode
      trav->next = newnode;
}
```

      3. Insert at position: O(n)

```
// if list is empty or first pos, newnode is
```

Scanned by CamScanner

```
first node.
if(head==NULL || pos==1)
      addfirst(val);
else
{
      // create and initialize node
      newnode = new node(val);
      // traverse till pos-1 node
      trav = head;
      for(i=1; i<pos-1; i++)
            trav = trav->next;
      // newnode's next -->trav's next
      newnode->next = trav->next;
      // trav's next -->newnode
      trav->next = newnode;
}
```

4. Delete First: O(1)

```
if(head!=NULL)
{
      // keep address of first node in temp
pointer
      temp = head;
      // take head to next node.
      head = head->next;
      // delete temp node.
      delete temp;
}
```

5. Delete At Position: O(n)

```
// if node to be deleted is first or list is
empty
if(pos==1 || head==NULL)
      delfirst();
else
{
      // traverse till pos-1
      trav = head;
      for(i=1; i<pos-1; i++)
            trav = trav->next;
      // temp should point to the node to be
deleted.
      temp = trav->next;
      // trav's next --> temp's next
      trav->next = temp->next;
      // delete temp node
      delete temp;
}
```

Scanned by CamScanner

6. Delete All: O(n)

```
// delete all nodes one by one, until list
become empty
while(head!=NULL)
      delfirst();
```

7. Traverse Linked List:

```
// start traversing from first node
trav = head;
while(trav!=NULL)
{
      // visit current node
      printf("%d, ", trav->data);
      // go to the next node
      trav = trav->next;
} // repeat until end of list is reached
```

B. Singly Linked List –head and tail pointer:

1. Add First: O(1)

```
// create and initialize node
newnode = new node(val);
// if list is empty, newnode is first node.
if(head==NULL)
      head = tail =newnode;
else
{
      // newnode's next --> head
      newnode->next = head;
      // head -->newnode
      head = newnode;
}
```

2. Add Last: O(1)

```
// create and initialize node
newnode = new node(val);
// if list is empty, newnode is first node.
if(head==NULL)
      head = tail = newnode;
else
{
      // tail's next -->newnode
      tail->next = newnode;
      // tail -->newnode
      tail = newnode;
}
```

Scanned by CamScanner

3. **Delete First: O(1)**

```
if(head!=NULL)
{
        // keep address of first node in temp
pointer
        temp = head;
        // take head to next node.
        head = head->next;
        // if deleted node was last node, then
        if(head==NULL)
                tail = NULL;
        // delete temp node.
        delete temp;
}
```

- **Singly Circular Linked List:**

  Last node's next pointer keeps address of first (head) node.

1. **Add First: O(n)**

```
// create and initialize node
newnode = new node(val);
// if list is empty, newnode is first node.
if(head==NULL)
{
        head = newnode;
        newnode->next = newnode;
}
else
{
        // traverse till last node
        trav = head;
        while(trav->next!=head)
                trav = trav->next;
        // trav's next -->newnode
        trav->next = newnode;
        // newnode's next --> head
        newnode->next = head;
        // head -->newnode
        head = newnode;
}
```

2. **Add Last: O(n)**

```
// if list is empty, newnode is first node.
if(head==NULL)
{
        head = newnode;
        newnode->next = newnode;
```


Scanned by CamScanner

```
}
else
{
        // traverse till last node
        trav = head;
        while(trav->next!=head)
                trav = trav->next;
        // trav's next -->newnode
        trav->next = newnode;
        // newnode's next --> head
        newnode->next = head;
}
```

3. Insert at position: O(n)

```
// if list is empty or first pos, newnode is first
node.
if(head==NULL || pos==1)
        addfirst(val);
else
{
        // create and initialize node
        newnode = new node(val);
        // traverse till pos-1 node
        trav = head;
        for(i=1; i<pos-1; i++)
                trav = trav->next;
        // newnode's next -->trav's next
        newnode->next = trav->next;
        // trav's next -->newnode
        trav->next = newnode;
}
```

4. Delete First: O(n)

```
// if head is null, return
if(head==NULL)
        return;
// if there is a single node, delete it
else if(head==head->next)
{
        delete head;
        head = NULL;
}
// if there are more than one node
else
{
        // keep address of head node in temp pointer
        temp = head;
        // traverse till last node
```

Scanned by CamScanner

```
        trav = head;
        while(trav->next!=head)
              trav = trav->next;
        // take head to next node.
        head = head->next;
        // update last node's next -->new head
        trav->next = head;
        // delete temp node.
        delete temp;
}
```

5. **Delete At Position: O(n)**

```
// if node to be deleted is first or list is empty
if(pos==1 || head==NULL)
      delfirst();
else
{
      // traverse till pos-1
      trav = head;
      for(i=1; i<pos-1; i++)
            trav = trav->next;
      // temp should point to the node to be
deleted.
      temp = trav->next;
      // trav's next --> temp's next
      trav->next = temp->next;
      // delete temp node
      delete temp;
}
```

6. **Traverse Linked List:**

```
// start traversing from first node
trav = head;
do
{
      // visit current node
      printf("%d, ", trav->data);
      // go to the next node
      trav = trav->next;
} while(trav!=head);
// repeat until end of list is reached
```

- **Doubly Linear Linked List**

  Each node contains data, address of next node and address of previous node.

  1. **Add First: O(1)**

```
// create and initialize node
newnode = new node(val);
```


Scanned by CamScanner

```
// if list is empty, newnode is first node.
if(head==NULL)
        head = newnode;
else
{
        // newnode's next --> head
        newnode->next = head;
        // cur head's prev -->newnode
        head->prev = newnode;
        // head -->newnode
        head = newnode;
}
```

2. Add Last: O(n)

```
// create and initialize node
newnode = new node(val);
// if list is empty, newnode is first node.
if(head==NULL)
        head = newnode;
else
{
        // traverse till last node
        trav = head;
        while(trav->next!=NULL)
                trav = trav->next;
        // trav's next -->newnode
        trav->next = newnode;
        // newnode'sprev --> last node (trav)
        newnode->prev = trav;
}
```

3. Insert at position: O(n)

```
// if list is empty or first pos, newnode is first
node.
if(head==NULL || pos==1)
        addfirst(val);
else
{
        // create and initialize node
        newnode = new node(val);
        // traverse till pos-1 node
        trav = head;
        for(i=1; i<pos-1; i++)
                trav = trav->next;
        // newnode's next -->trav's next
        newnode->next = trav->next;
        // newnode'sprev -->trav
        newnode->prev = trav;
```


Scanned by CamScanner

```
        // next node's prev -->newnode
        if(trav->next!=NULL)
            trav->next->prev = newnode;
        // trav's next -->newnode
        trav->next = newnode;
}
```

4. Delete First: O(1)

```
if(head!=NULL)
{
        // keep address of first node in temp pointer
        temp = head;
        // take head to next node.
        head = head->next;
        // delete temp node.
        delete temp;
        // if node deleted is not last, make its prev
NULL
        if(head!=NULL)
            head->prev = NULL;
}
```

5. Delete At Position: O(n)

```
// if node to be deleted is first or list is empty
if(pos==1 || head==NULL)
        delfirst();
else
{
        // traverse till pos
        trav = head;
        for(i=1; i<pos; i++)
            trav = trav->next;
        // trav'sprev node's next -->trav's next
        trav->prev->next = trav->next;
        // trav's next node's prev -->trav'sprev
        if(trav->next!=NULL)
            trav->next->prev = trav->prev;
        // delete trav node
        delete trav;
}
```

6. Delete All: O(n)

```
// delete all nodes one by one, until list become
empty
while(head!=NULL)
        delfirst();
```


Scanned by CamScanner

7. Traverse Linked List:

```
// start traversing from first node
trav = head;
while(trav!=NULL)
{
      // visit current node
      printf("%d, ", trav->data);
      // go to the next node
      trav = trav->next;
} // repeat until end of list is reached
```

8. Traverse Linked List – In Reverse Direction:

```
// traverse till last node
trav = head;
while(trav->next!=NULL)
      trav = trav->next;
// start traversing from last node
while(trav!=NULL)
{
      // visit current node
      printf("%d, ", trav->data);
      // go to the prev node
      trav = trav->prev;
} // repeat until beginning of list is reached
```

- Doubly Circular Linked List

  Each node contains data, address of next node and address of previous node. Last node's next contains address of first node, while first node's prev contains address of last node.

1. Add First: O(1)

```
// create and initialize node
newnode = new node(val);
// if list is empty, newnode is first node.
if(head==NULL)
{
      head = newnode;
      newnode->next = newnode;
      newnode->prev = newnode;
}
else
{
      // jump to the last node
      tail = head->prev;
      // newnode's next --> head
      newnode->next = head;
      // newnode'sprev -->tail
      newnode->prev = tail;
```

Scanned by CamScanner

```
        // tail's next -->newnode
        tail->next = newnode;
        // cur head's prev -->newnode
        head->prev = newnode;
        // head -->newnode
        head = newnode;
}
```

2. **Add Last: O(1)**

```
// create and initialize node
newnode = new node(val);
// if list is empty, newnode is first node.
if(head==NULL)
{
        head = newnode;
        newnode->next = newnode;
        newnode->prev = newnode;
}
else
{
        // jump to the last node
        tail = head->prev;
        // newnode's next --> head
        newnode->next = head;
        // newnode'sprev --> tail
        newnode->prev = tail;
        // tail's next -->newnode
        tail->next = newnode;
        // cur head's prev -->newnode
        head->prev = newnode;
}
```

	Array	Linked List
1.	Array cannot grow or shrink dynamically (realloc() is not efficient).	Linked list can grow/shrink dynamically.
2.	Array has contiguous memory. Hence random access is possible.	Nodes are not in contiguous memory. Hence only sequential access is allowed.
3.	Arrays do not have memory overheads.	Linked lists have memory overheads e.g. each node contains address of 'next' node.
4.	Insert/Delete at middle position need to shift remaining elements. Hence will be slower.	Insert/Delete at middle position need to modify links (next/prev pointers). Hence will be faster.
5.	To deal to fixed number of elements and frequent random access, arrays are better.	To deal to dynamic number of elements and frequent insertion/deletion operators, linked lists are better.

Scanned by CamScanner

	Singly Linked List with Head	Singly Linked List with Head & Tail
1.	Add last operation is slower i.e. O(n)	Add last operation is faster i.e. O(1)
2.	No overload.	Extra overhead i.e. tail pointer.
3.	Simplified list manipulation operations.	List manipulation need to consider tail pointer wherever appropriate.

	Singly Linked List	Doubly Linked List
1.	Unidirectional access.	Bi-directional access.
2.	For search and delete operation two pointers are required (Need to maintain prev pointer) during traversing.	For search and delete operation only one pointer is required (No need to maintain prev pointer) during traversing.
3.	Simplified list manipulation operations.	List manipulation need to consider prev pointer of each node in all operations.

	Doubly Linked List	Doubly CircularLinked List
1.	Traversing till last node is slower (add last operation) i.e. O(n)	Traversing till last node is faster (add last operations) i.e. O(1)

Scanned by CamScanner

Trees

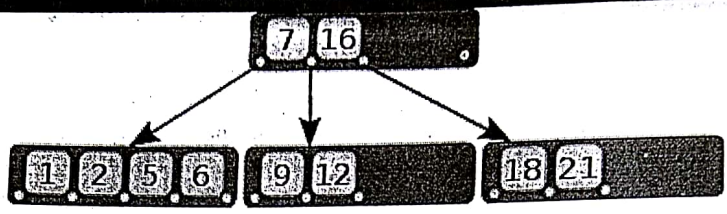
- **Terminologies:**
  - Tree is a non-linear data structure in which one specially designated node is called as "root" (i.e. starting point of the tree) and remaining elements can be partitioned into "m" disjoint subsets so that each subset is a tree itself.
  - Number of child nodes for any node is called as "degree of a node".
  - Maximum degree of a child in tree is called as "degree of the tree".
  - All the nodes in the path from the root to that node, are called "ancestors" of that node.
  - All the nodes accessible from the given node, are called as "descendants" of that node.
  - Child nodes of the same parent are called as "Siblings".
  - Terminal node of the tree is called as "leaf node". Leaf node does not have child nodes.
  - Level of node:
    - Indicates position of the node in tree hierarchy.
    - Level of child = Level of parent + 1
    - Level of root = 0
  - The maximum level of a node is "Height of tree".
  - Tree with no nodes (i.e. empty tree) is called as "Null Tree". Height of Null tree is -1.
- **Types of Trees:**
  - **Binary Tree:**
    - Tree in which each node has maximum two child nodes, is called as "binary tree".
    - Binary tree has degree 2. Hence it is also called as 2-tree.
  - **Binary Search Tree:**
    - The binary tree in which left child node is always smaller and right child node is always greater or equal to the (parent) node; is called as "Binary Search Tree".
    - Searching is faster. Time complexity:O(h). Where "h" is height of the tree.
  - **Complete Binary Tree:**
    - Binary tree in which all leaf nodes are at the same level.
  - **Strictly Binary Tree:**
    - Binary tree in which each non-leaf node has exact two child nodes.
  - **Full Binary Tree:**
    - Binary tree with its full capacity for the given height.
    - In other words, adding one more node will increase height of the tree.
    - It is always complete as well as strictly binary tree.
    - Number of elements = $2^{(h+1)} - 1$
  - **Almost complete binary tree:**
    - The binary tree which follows two conditions:
      - All leaf nodes are at level h or h-1.
      - All leaf nodes at last level (h), are aligned to left as much as possible.
  - **Threaded Binary Tree:**

Scanned by CamScanner

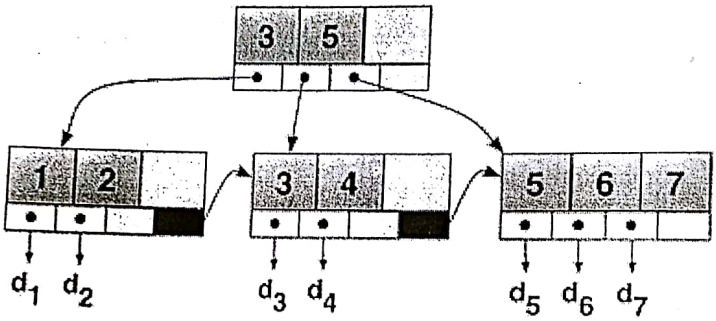
- These trees will allow very fast in-order traversal (ascending or descending).
- Binary tree in which right null link of each node is replaced by address of its in-order successor; such tree is called as "right threaded binary tree".
- Binary tree in which left null link of each node is replaced by address of its in-order predecessor; such tree is called as "left threaded binary tree".
- If right as well as left null links of each node is replaced by its in-order successor and in-order predecessor respectively; then such tree is called as "in threaded binary tree".

- **Skewed Binary Tree:**
  - The binary tree in which only left link is used to keep address of child node (right link of each node is kept null), is called as "left skewed binary tree".
  - The binary tree in which only right link is used to keep address of child node (left link of each node is kept null), is called as "right skewed binary tree".
  - Height of skewed binary tree is maximum for given number of nodes.Hence searching will be too slow i.e. O(n)
- **Balanced binary tree:**
  - In binary search trees, searching is faster if height of the tree is kept as minimum as possible.
  - The binary search tree with minimum possible height (for given number of nodes) is called as "balanced binary search tree" or "height balanced binary search tree".
  - Balance factor of node = height of left sub tree - height of right sub tree.
  - The binary search tree, in which balance factor of each node is in range of -1 to 1, is called as "balanced binary search tree".
  - On any binary search tree, series ofappropriate left rotations and right rotations on appropriate nodes can convert it into "balanced binary search tree".
    - If balance factor of a node < -1, apply left rotation on that node.
    - If balance factor of a node > +1, apply right rotation on that node.
- **AVL Tree:**
  - Invented by "GreogyAdelson-Velsky" and "E.M. Landis".
  - Self-balancing binary search tree.
  - After each insert or delete operation, appropriate rotations are applied to make tree balanced again.
- **B Tree:**
  - The B-tree is a generalization of a binary search tree in that a node can have more than two children.
  - In B-trees, internal (non-leaf) nodes can have a variable number of child nodes within some pre-defined range.
  - In order to maintain the pre-defined range, internal nodes may be joined or split.
  - Each internal node of a B-tree will contain a number of keys. The keys act as separation values which divide its sub-trees.
  - A B-tree is kept balanced by requiring that all leaf nodes be at the same depth (level).

Scanned by CamScanner



- **B+ Tree:**
  - Like B trees but with few changes as follows:
  - In the B+ tree, copies of the keys are stored in the internal nodes.
  - The keys and records are stored in leaves.
  - A leaf node may include a pointer to the next leaf node to speed sequential access.



Binary Search Tree:

- Implementation(Design) of binary search tree:

```
    class tree;
    class node
    {
   private:
      int data;
      node *left, *right;
  public:
     node(intval=0);
     friend class tree;
  };
  class tree
  {
 private:
    node *root;
 public:
    tree();
    void add(intval); // add node
    node* search(int key); // search node
    node* search(int key, node **parent);// search node with
parent
    void preorder(node *trav); // preorder traversal
```


Scanned by CamScanner

```
        void preorder();// wrapper for preorder(root);
        void inorder(node *trav); // inorder traversal
        void inorder(); // wrapper for inorder(root);
        void postorder(node *trav); // postorder traversal
        void postorder();     // wrapper for postorder(root);
        void height(node *trav); // height of tree
        void height();  // wrapper for height(root);
        void clear(node *trav); // delete node and its
descendants
        void clear();    // wrapper for clear(root);
        void preorder_nonrec(); // preorder traversal non-
recursive
        void inorder_nonrec(); // inorder traversal non-recursive
        void postorder_nonrec(); // postorder traversal non-
recursive
        void del(int key); // search and delete node
    };
```

- Add a node in tree:

```
void add(intval)
{
        // create and initialize node
        node *newnode = new node(val);
        // if tree is empty
        if (root == NULL)
                root = newnode;
        else
        {
                node *trav = root;
                while (true)
                {
                        if (val<trav->data)
                        {
                                if (trav->left == NULL)
                                {
                                        trav->left = newnode;
                                        break;
                                }
                                trav = trav->left;
                        }
                        else
                        {
                                if (trav->right == NULL)
                                {
                                        trav->right = newnode;
                                        break;
```


Scanned by CamScanner

```
                        }
                        trav = trav->right;
                    }
                }
            }
        }
```

- **Tree Traversal:**
  - Pre-order: Parent , Left, Right
  - In-order: Left, Parent, Right
  - Post-order: Left, Right, Parent

```
void preorder(node *trav)
{
    if (trav == NULL)
        return;
    printf("%d, ", trav->data);
    preorder(trav->left);
    preorder(trav->right);
}
void inorder(node *trav)
{
    if (trav == NULL)
        return;
    inorder(trav->left);
    printf("%d, ", trav->data);
    inorder(trav->right);
}
void postorder(node *trav)
{
    if (trav == NULL)
        return;
    postorder(trav->left);
    postorder(trav->right);
    printf("%d, ", trav->data);
}
```

- Delete All Nodes in Tree:

```
void clear(node *trav)
{
    if (trav == NULL)
        return;
    clear(trav->left);
    clear(trav->right);
    delete trav;
}
void clear()
{
```


Scanned by CamScanner

```
        clear(root);
        root = NULL;
}
```

- **Height of tree:**

```
int height(node *trav)
{
        if (trav == NULL)
                return -1;
        int hl = height(trav->left);
        inthr = height(trav->right);
        int max = (hl >hr ? hl : hr);
        return max + 1;
}
int height()
{
        return height(root);
}
```

- **Search a node with its parent:**

```
node* search(int key, node **parent)
{
        node *trav = root;
        *parent = NULL;
        while (trav != NULL)
        {
                if (key == trav->data)
                        return trav;
                *parent = trav;
                if (key <trav->data)
                        trav = trav->left;
                else
                        trav = trav->right;
        }
        *parent = NULL;
        return NULL;
}
```

- Delete a node:
  - if node's left is null:

```
if(temp->left==NULL)
{
        if(temp==root)
                root = temp->right;
        else if(temp==parent->left)
                parent->left = temp->right;
        else
```


Scanned by CamScanner

```
            parent->right = temp->right;
}
```

- if node's right is null:

```
if(temp->right==NULL)
{
        if(temp==root)
                root = temp->left;
        else if(temp==parent->left)
                parent->left = temp->left;
        else
                parent->right = temp->left;
}
```

- if node's left as well as right is NOT null:

```
if(temp->left!=NULL && temp->right!=NULL)
{
        // find the succ of temp along with its (succ)
parent
        parent = temp;
        succ = temp->right;
        while(succ->left!=NULL)
        {
                parent = succ;
                succ = succ->left;
        }
        // replace temp data by succ data
                temp->data = succ->data;
        // mark succ for deletion
        temp = succ;
}
```

- deleting node (full) - steps:
  - Search node to be deleted along with its parent.
  - If node's left as well as right is NOT null, deletion code for that.
  - If node's left is null, deletion code for that.
  - Else if node's right is null, deletion code for that.
  - delete memory of temp node.

- Pre-order non-recursive:

```
void preorder_nonrec()
{
        printf("NONREC PRE : ");
        node *trav = root;
        stack<node*> s;
        while (trav != NULL || !s.empty())
        {
                while (trav != NULL)
                {
                        printf("%d, ", trav->data);
                        if (trav->right != NULL)
```


Scanned by CamScanner

```
                    s.push(trav->right);
                trav = trav->left;
            }
            if (!s.empty())
            {
                trav = s.top();
                s.pop();
            }
        }
        printf("\n");
}
```

- In-order non-recursive:

```
void inorder_nonrec()
{
    printf("NONREC IN  : ");
    node *trav = root;
    stack<node*> s;
    while (trav != NULL || !s.empty())
    {
        while (trav != NULL)
        {
            s.push(trav);
            trav = trav->left;
        }
        if (!s.empty())
        {
            trav = s.top();   s.pop();
            printf("%d, ", trav->data);
            trav = trav->right;
        }
    }
    printf("\n");
}
```

- Post-order non-recursive:

```
void postorder_nonrec()
{
    printf("NONREC POST: ");
    node *trav = root;
    stack<node*> s;
    while (trav != NULL || !s.empty())
    {
        while (trav != NULL)
        {
            s.push(trav);
            trav = trav->left;
        }
```


Scanned by CamScanner

```
            if (!s.empty())
            {
                trav = s.top(); s.pop();
                if (trav->right == NULL || trav->right-
>visited == true)
                {
                    printf("%d, ", trav->data);
                    trav->visited = true;
                    trav = NULL;
                }
                else
                {
                    s.push(trav);
                    trav = trav->right;
                }
            }
        }
        printf("\n");
}
```

- **Depth First Search:**

```
node* bfs(int key)
{
    node *trav;
    queue<node*> q;
    //push root on queue
    q.push(root);
    //repeat until queue is empty
    while (!q.empty())
    {
        // pop an ele from queue
        trav = q.front();    q.pop();
        // compare with ele to search
        if (key == trav->data)
            return trav;
        // if trav has left child, push on queue
        if (trav->left != NULL)
            q.push(trav->left);
        // if trav has right child, push on queue
        if (trav->right != NULL)
            q.push(trav->right);
    }
    return NULL;
}
```

- **Breadth First Search:**

```
node* dfs(int key)
{
```


Scanned by CamScanner

```
        node *trav;
        stack<node*> s;
        //push root on stack
        s.push(root);
        //repeat until stack is empty
        while (!s.empty())
        {
                // pop an ele from stack
                trav = s.top();  s.pop();
                // compare with ele to search
                if (key == trav->data)
                        return trav;
                // if trav has right child, push on stack
                if (trav->right != NULL)
                        s.push(trav->right);
                // if trav has left child, push on stack
                if (trav->left != NULL)
                        s.push(trav->left);
        }
        return NULL;
}
```

Array Implementation of tree:

- A binary tree can be simulated in an array i.e. parent-child relationship can be maintained.
- Typically array begin with index 1. Then
  - parent index = child index / 2
  - left child index = parent index * 2
  - right child index = parent index * 2 + 1
- Array implementation of almost complete binary tree, is called as "Heap" data structure.
- If each parent element in heap is greater than both of its child elements, then it is called as "Max Heap".
- If each parent element in heap is less than both of its child elements, then it is called as "Min Heap".
- Heap sort algorithm uses Max heap (ascending sort) or Min heap (descending sort).

Scanned by CamScanner

HASHING:

Suppose we want to design a system for **storing employee records keyed using phone numbers,** and we want following queries to be performed efficiently:

- **insert** a phone number and corresponding information
- **delete** a phone number and related information
- **search** a phone number and fetch the information

We can think of using the following data structures to maintain information about different phone numbers.

1. **array: of phone numbers and records**
2. **linked list: of phone numbers and records**
3. **balanced binary search tree with phone numbers as keys**
4. **direct access table**

- For **arrays and linked lists:** we need to search in a linear fashion which can be costly in practice. If we use array and keep the data stored, then a phone number can be searched in **O(log n)** time using binary search, but insert and delete operations becomes costly as we have to maintain sorted order.
- With **balanced binary search tree,** we get moderate search, insert and delete times. All of these operations can be guaranteed to be in **O(log n)** time.
- Another solution that one can think of is to use a **direct access table:** where we make a big array and use phone numbers as index in the array. An entry in an array is NIL if phone number is not present, else the array entry stores pointer to records corresponding to phone number.
- **Time complexity wise this solution is the best among all, we can do all operations in O(1) time.**
- For example to insert a phone number, we create a record with details of given phone number, use phone number as index and store the pointer to the created record in table.
- This solution has many practical limitations. First problem with this solution **extra space required is huge.**
- For example **if phone number is n digits, we need** $O(m*10^n)$ space for table where **m is size of pointer to record.** Another problem is an integer in a programming language may not store n digits.
- Due to above limitations **Direct Access Table** cannot always be used.
- **Hashing is the solution that can be used in almost all such situations and performs extremely well compared to above data structures like Arrays, Linked List, Balanced BST in practice.**
- **With hashing we get O(1) search time on average(under reasonable assumptions) and O(n) in worst case.**
- **Space required is: average and worst case: O(n)**
- The **average case** and **worst case** time complexity of **search, insert** and **delete** operations takes **O(1)** time.
- **Hashing is an improvement on Direct Access Table.**
- **The idea is to use hash function that converts a given phone number or any other key into a smaller number and uses the small number as index in a table called hash table.**

Scanned by CamScanner

- **Hash Table: Hash Table (hash map)** is a data structure which implements an **associative array abstract data type, a structure that can map keys to values.**

- **Associative array:** In computer science **associative array, map, symbol table,** or **dictionary** is an abstract data type composed of a collection of (key, value) pairs, such that each possible key appears at most once in the collection.

  Operations associated with this data type as follows:
  1. the addition of pair of the collection
  2. the removal of a pair from the collection
  3. the modification of an existing pair
  4. the lookup of a value associated with a particular key

- In **associative array, the association between a key and value is often known as a binding,** and the same word binding may also used to refer to the process of creating a new association.

- The operations that are usually defined for an associative array are:
  1. Add or insert: add a new (key, value) pair to the collection. The arguments to this operation are the key and value.
  2. Reassign: replace the value in one of the (key, value) pairs that are already in the collection, binding an old key to a new value. As with an insertion, the argument to this operation is the key and the value.
  3. Remove or Delete: remove a (key, value) pair from the collection, unbinding a given key from its value. The argument to this operation is key.
  4. Lookup: find the value (if any) that is bound to a given key. The argument to this operation is the key, the value is returned from the operation. If no value is found, some associative array raise an exception

- A hash table uses **hash function** to compute an index into an array of **buckets** or **slots**, from which the desired value can be found.

- **Hash Function:**
- A Hash Function, **is any function that can be used to map data of arbitrary size to data of fixed size.**
- A function **that converts a given big phone number into a small practical integer value.** The mapped integer value is used as an index in the hash table.
- In simple terms, **a hash function maps a big number or string to a small integer that can be used as index in hash table.**
- The values returned by hash function are called as **hash values, hash codes, digests** or simply **hashes.**

- **Applications of hash table:**

Scanned by CamScanner

- widely used in **computer software for rapid data lookup.** Hash functions **accelerates table or database lookup by detecting duplicated records in a large file.**
- They are also used in **Cryptography.**
- Hash functions are also used **to build caches for large data sets stored in slow media.**

- A good hash function should have following properties:
  1. Efficiently Computable
  2. Should uniformly distribute the keys(each table position equally likely for each key).

- For example for phone numbers, a bad hash function is to take first three digits. A better function is consider last three digits. Please note that this may not be the best hash function. There may be better ways.

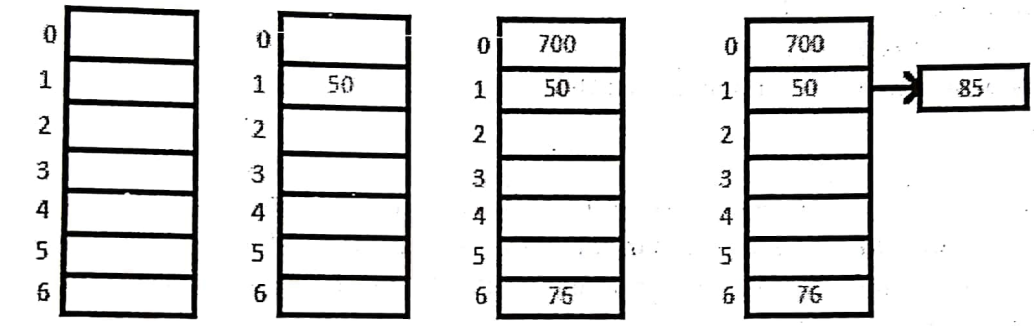
- **Hash Table:**
- **An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number, has hash function value equal to the index for the entry.**
- In Computing, a **hash table (hash map) is a data structure which implements an associative array abstract data type, a structure that can map keys to values.**
- A hash table uses **hash function** to compute an index into an array of **buckets** or **slots**, from which the desired value can be found.
- Ideally, **the hash function will assign each key to a unique bucket/slot,** but most hash table designs employ an imperfect hash function, which might cause **hash collisions** where **the hash function generates the same index for more than one key.**
- **Hash functions** are used in hash tables, **to quickly locate a data record** (e.g. a **dictionary** definition) given its search key(the headword). Specifically, **the hash function is used to map the search key to an index;** the index gives the place in the hash table where the corresponding record should be stored. Hash tables, in turn, **are used to implement associative arrays and dynamic sets.**

- ***Collision Handling:***
- **Collision:** Since a hash function gives us a small number for a big key, there is possibility that two keys result in same value. **The situation where a newly inserted key maps to an already occupied slot in hash table is called collision** and must be handled using some collision handling technique.
- **Following are the ways to handle the collisions:**
  1) **Separate Chaining:** The idea is **to make each cell of hash table point to a linked list of records that have same hash function value.** Chaining is simple, but requires additional memory outside the table.
  2) **Open Addressing:** In open addressing, **all elements are stored in the hash table itself.** Each table entry contains either a record or NIL. When searching for an element, we one by one examine table's slots until the desired element is found or it is clear that the element is not in the table.

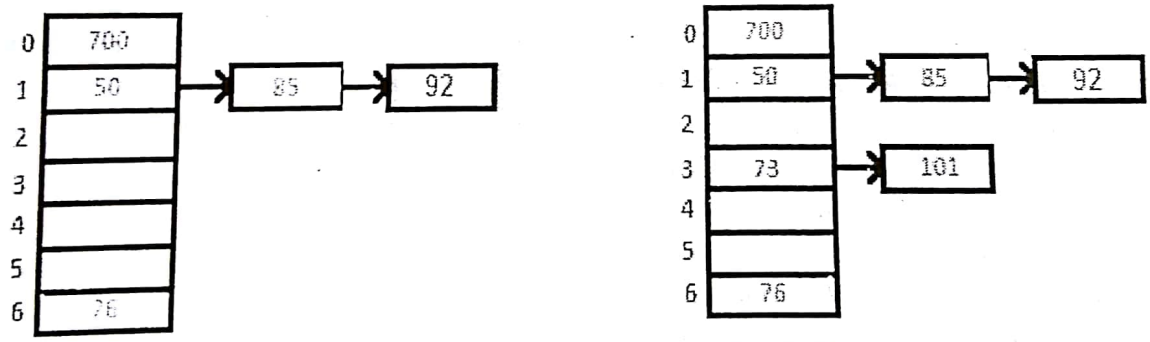
Scanned by CamScanner

- **Separate Chaining:**
  - **What is Collision?**
    Since a hash function gets us a small number for a key which is big integer or string, there is possibility that two keys result in same value. **The situation where newly inserted key maps to an already occupied slot in hash table is called collision** and must be handled using some collision handling technique.

  - The idea is **to make each cell of hash table point to a linked list of records that have same hash function value.**
  - Let us consider a simple hash function as **"key mod 7"** and sequence keys as **50, 700, 76, 85, 92, 73, 101.**



Initial Empty Table | Insert 50 | Insert 700 and 76 | Insert 85: Collision Occurs, add to chain



Insert 92: Collision Occurs, add to chain | Insert 73 and 101

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

Scanned by CamScanner

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

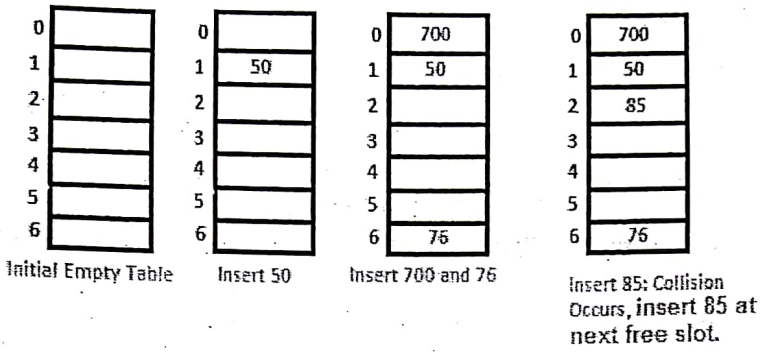
- **Performance of Chaining:**
  Performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of table (simple uniform hashing)

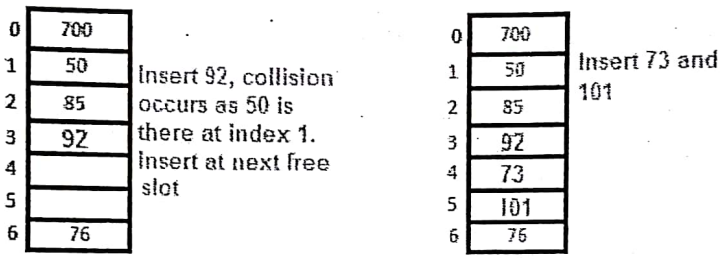
  m = number of slots in hash table
  n = number of keys to be inserted in hash table
  Load Factor => LF = n / m
  expected time to search: **O(1 + LF)**
  expected time to insert/delete = **O(1 + LF)**
  Time complexity of search, insert and delete is: O(1)
  if LF is O(1).

- **Open Addressing:**
- Like Separate Chaining, Open Addressing is a method for handling collisions.
- In **Open Addressing, all elements are stored in the hash table itself. So at any point, size of table must be greater than or equal to total number of keys (note that we can increase table size by copying old data if needed).**
- **Insert(k):Keep probing until an empty slot is not found. Once an empty slot is found, insert key.**
- **Search(k):Keep probing until slot's key doesn't become equal to k or an empty slot is reached.**
- **Delete(k):Delete operation is interesting. If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as "deleted".**
- **Insert can insert an item in a deleted slot, but search doesn't stop at a deleted slot.**
  [Probe: investigate, to search into, look into]
- **Open Addressing is done following ways:**
  a) **Linear Probing:** In Linear Probing, we linearly probe for the next slot. For example, typical gap between two probes is 1 as taken in below example also.
  Let **hash(x)** be **the slot index computed using hash function** and S be **the table size.**
  if slot **hash(x) % S** is full, then we try **(hash(x)+1)%S**
  if**(hash(x)+1)%S** is also full, then we try **(hash(x)+2)%S**
  if**(hash(x)+2)%S** is also full, then we try **(hash(x)+3)%S**
  .....
  .....

Scanned by CamScanner

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





- **Clustering:** The main problem with Linear Probing is **Clustring, many consecutive elements forms groups and it starts taking time to find a free slot or to search an element.**

b) **Quadratic Probing:** We look for $i^{2\,th}$ slot in i^{th} iteration
Let **hash(x)** be **the slot index computed using hash function** and **S be the table size.**

If slot **hash(x)%S** is full, then we try **(hash(x)+1*1)%S**
If **(hash(x)+1*1)%S** is also full, then we try **(hash(x)+2*2)%S**
If **(hash(x)+2*2)%S** is also full, then we try **(hash(x)+3*3)%S**
.....
.....

c) **Double Hashing:** We use another hash function **hash2(x)** and look for **i*hash2(x)** slot in i^{th} iteration.
Let **hash(x)**be the slot index computed using hash function and S be the table size.

If slot **hash(x)%S** is full, then we try **(hash(x)+1* hash2(x))%S**
If **(hash(x)+1*hash2(x))%S** is also full, then we try **(hash(x)+2*hash2(x))%S**
If **(hash(x)+2*hash2(x))%S** is also full, then we try **(hash(x)+3*hash2(x))%S**
.....
.....

- **Comparison of above three:**
- **Linear Probing** has the **best cache performance**, but suffers from **clustering**. One more advantage of Linear Probing is **easy to compute.**

Scanned by CamScanner

- **Quadratic Probing** lies between the two in terms of cache performance and clustering.
- **Double Hashing** has **poor cache performance** but **no clustering.** Double Hashing **requires more computation time as two hash functions need to be computed.**

- **Open Addressing v/s Separate Chaining:**
  - **Advantages of Chaining:**
    1. Chaining is **simpler to implement**
    2. In Chaining, **Hash table never fills up**, we can always add more elements to chain. In open addressing, **table may become full.**
    3. Chaining is **less sensitive to the hash function or load factor.**
    4. Chaining is **mostly used when it is unknown how many and how frequently keys may be inserted or deleted.**
    5. Open Addressing **requires extra care for to avoid clustering and load factor.**
  - **Advantages of Open Addressing:**
    1. Cache Performance of chaining is not good, as keys are stored using linked list. **Open addressing provides better cache performance** as everything is stored in same table.
    2. **Wastage of Space** (Some parts of hash table in chaining are never used). In Open Addressing, a slot can be used even if an input doesn't map to it.
    3. Chaining **uses extra space for links.**

- **Performance of Open Addressing:**

Like Chaining, performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of table (simple uniform hashing).

m = no. of slots in hash table
n = no. of keys to be inserted in hash table

load factor LF = n / m (< 1)
expected time to search/insert/delete < 1/(1-LF)
so search, insert and delete takes: O(1/(1-LF)) time.

Scanned by CamScanner

GRAPH:

- A graph is a **non-linear advanced data structure** that consists of two components:
  1. A finite set of **vertices** also called as **nodes**
  2. A finite set of ordered pair of the form (u, v) called as **edge** or **arc.**

  -> The pair is ordered because (u, v) is not same as (v, u) in case of directed graph(**di-graph**).

  -> The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v.

  -> The edges may contain **weight/cost**.

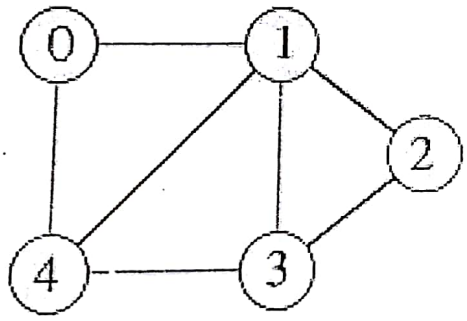
  Different data structures for the representation of the graphs are used in practice:

- **Applications of Graph:**
- Graphs are used to represent many real life applications:
- Graphs are used to represent networks. The network may includes paths in a city or telephone network or circuit network.
- Graphs are also used in social network sites like LinkedIn, Facebook. For example in Facebook each person represented with a vertex (or a node). Each node is a structure and contains information like person_id, name, gender and locale.
- Graphs can be model many types of relations and processes in physical, biological, social and information systems. Many practical problems can be represented by graphs.

- **Representation of Graph:**

  Graph can be represented by two ways:

  1. Adjacency Matrix (using 2D array)
  2. Adjacency List (using linked list)



1. Adjacency Matrix (using 2D array):

- Adjacency matrix is 2D array of size V*V where V is the number of vertices in a graph.
- Let the 2D array be: adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.
- Adjacency matrix for undirected graph is always symmetric matrix.
- Adjacency matrix is also used to represent weighted graphs. If adj[i][j]=w, then there is an edge from vertex i to vertex j with weight w.
- Example: Un-directed Un-weighted Graph: G

  V = { 0,1,2,3,4 } and

  E = { (0,1), (0,4),(1,2), (1,3), (1,4), (2,3), (3,4) }.

MATRIX REPRESENTATION OF ABOVE GRAPH:

Scanned by CamScanner

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

PROS:

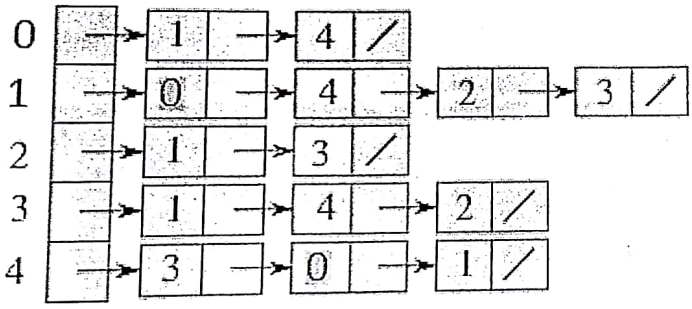
- Representation is easier to implement and follow.
- Removing an edge takes O(1) time.
- Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done in O(1).

CONS:

- Consumes more space $O(V^2)$. Even if the graph is sparse (contains less no. of edges), it consumes the same space
- Adding a vertex is $O(V^2)$ time.

2. Adjacency List (using linked list):

- Following is the **adjacency list representation** of the above graph.



- An array of linked list is used.
- Size of the array is equal to number of vertices
- Let the array be arr[]. An entry of arr[i] represents the linked list of vertices adjacent to the ith vertex.
- This representation can also used to represent a weighted graph.
- The weights of edges can be stored in nodes of linked lists.

PROS:

- Saves space O(|V|+|E|).
- In the worst case, there can be C(V,2) number of edges in a graph, thus consuming $O(V^2)$ space.
- Adding vertex is easier.

CONS:

- Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done O(V).

Scanned by CamScanner

SPANNING TREE: is a sub-graph of a graph which remains connected after removing one/more edges from it and does not contains a cycle.

FOREST:

- A Forest is an undirected graph, all of whose components are trees.
- In another words, the graph consists of a disjoint union of trees.
- Equivalently, a forest is an undirected cycle free graph.

GRAPH TRAVERSAL ALGORITHMS:

BFS: BREADTH FIRST TRAVERSAL ALGORITHM

1. push any node (starting vertex) into the queue and mark it
2. pop vertex from the queue and visit it (display)
3. push its all unmarked adjacent vertices (neighbors) into the queue and mark them
4. repeat steps 2&3 until queue is empty.

DFS: DEPTH FIRST TRAVERSAL ALGORITHM

1. Push any node (starting vertex) into the stack and mark it
2. Pop vertex from the stack and visit it (display)
3. Push all its unmarked adjacent vertices (neighbors) into the stack and mark them
4. Repeat steps 2 & 3 until stack is empty.

Greedy Algorithms: (Activity Selection Problem)

- Greedy is an algorithm paradigm **that builds up a solution piece by piece, always choosing the next that offers the most obvious and immediate benefit.**
- Greedy algorithms are used for **optimization problems.**
- Optimization problems can be solved using Greedy if the problems have the following property: **at every step, we can make a choice that looks best at that moment, and we get the optimal solution of the complete problem.**
- If Greedy algorithms can solve a problem then generally it becomes the best method to solve that problem.
- The **Greedy algorithms** are in general **more efficient** than other techniques like **Dynamic Programming**.
- But Greedy algorithms cannot always be applied, for example, **Fractional Knapsack Problem** can be solved using Greedy, but **0-1 Knapsack** cannot be solved by using Greedy.
- Following are some standard algorithms that are Greedy Algorithms:
  1. **Dijkstra's Shortest Path:** Dijkstra's algorithm is very similar to Prim's algorithm. **The shortest path tree is built up, edge by edge.** We maintain two sets: set of vertices already included in tree and the set of vertices not yet included. The Greedy choice is to pick the edge that connects the two sets and is on the smallest weight path from source to the set that contains not yet included vertices.
  2. **Kruskal's Minimum Spanning Tree (MST):** In Kruskal's Algorithm, **we create a MST by picking edges one by one.** The Greedy choice is to pick the smallest weight edge that doesn't cause a cycle in the MST constructed so far.

Scanned by CamScanner

3. **Prim's Minimum Spanning Tree (MST):** In Prim's algorithm **we create a MST by picking edges one by one.** We maintain two sets: set of vertices already included in MST and the set of vertices not included yet. **The Greedy choice is to pick the smallest weight edge that connects the two sets.**
4. **Huffman Coding: Huffman Coding is a loss-less compression technique.** It assigns variable length bit codes to different characters. The Greedy choice is to assign least bit length code to the most frequent character.

- Greedy Algorithms are sometimes also used **to get approximation for hard optimization problems.** For example, **Travelling Salesman Problem** is a **NP Hard Problem.** A Greedy choice for this problem is to pick the nearest unvisited city from the current city at every step. This solutions doesn't always produce the best optimal solution, but can be used to get an approximate optimal solution.

- **The Greedy choice is to always pick the next activity whose finish time is least among the remaining activities and the start time is more than or equal to the finish time of previously selected activity.** We can sort the activities according to their finishing time so that **we always consider the next activity as minimum finishing time activity.**

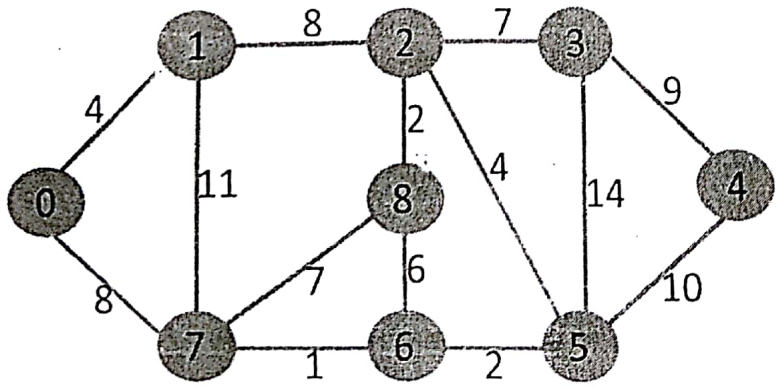
  1. Sort the activities according to their finishing time
  2. Select the first activity from the sorted array and print it
  3. Do the following for remaining activities in the sorted array.

- if the start time of this activity is greater than the finish time of previously selected activity then select this activity and print it.

Dijkstra's Algorithm (Shortest Path Algorithm):

- Given an undirected weighted input graph and a source vertex in graph, find shortest paths from source to all vertices in the given graph.
- Dijsktra's Algorithm is very similar to Prim's algorithm for Minimum Spanning Tree.
- Like Prim's MST, we generate a SPT(Shortest Path Tree) with given source as root.
- **We maintain two sets: one set contains vertices included in shortest path tree, and another set includes vertices not included in shortest path tree.**
- **At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has minimum distance from source.**

Example input graph:



Scanned by CamScanner

- Time complexity of the implementation is **O(V^2)**, if the input graph is represented using adjacency list, it can be reduced to **O(E. log V)** with the help of binary heap.
- Dijkstra's algorithm doesn't work for graphs with negative weight edges. Bellman-Ford algorithm can be used for the same.

PRIM'S ALGORITHM:

- Algorithm to find **Minimum Spanning Tree (MST) for a weighted undirected** graph.
- This means **it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.**
- The algorithm operates by building this MST, one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.
- The algorithm was developed in **1930** by Czech mathematician **Vojtech Jarnik** and later rediscovered and republished by computer scientists **Robert C. Prim in 1957** and **Edsger W. Dijkstra in 1959.** Therefore it is also called as the **DJP algorithm, Jarnik's algorithm,** the **Prim-Jarnikalgorithm** or the **Prim's-Dijsktra algorithm.**

TIME COMPLEXITY:

- The time complexity of Prim's algorithm depends on the data structure used for the graph and for ordering the edges by weight, which can be done by using a priority queue.

Minimum edge weight data structure	Time Complexity(total)
Adjacency matrix, searching	O(|V^2|)
Binary heap and adjacency list	O((|V|+|E|) log |V|) =O(|E| log |V|)
Fibonacci heap and adjacency list	O(|E| + |V| log |V|)

- A simple implementation of Prim's, using an **adjacency matrix** or **adjacency list** graph representation and linearly searching an array of weights to find the minimum weight edge, to add requires O(|V^2|) running time.
- However, this running time can be greatly improved further by using **heaps** to implement finding minimum weight edges in the algorithms inner loop.
- A first improved version uses a heap to store all edges on the input graph, ordered by their weight. This leads to an **O(|E| log |E|)** worst case running time. But storing vertices instead of edges can improve it still further. The heap should order the vertices by the smallest edge-weight that connects them to any vertex in the partially constructed minimum spanning tree MST (or infinity if no such edge exists).
- Every time a vertex V is chosen and added to the MST, a decrease key operation is performed on all vertices w outside the partial MST such that v is connected to w, setting the key to the minimum of its previous value and the edge cost of (v, w).
- Using a simple **binary heap** data structure, Prim's algorithm can now be shown to run in **O(|E| log |V|)** where |E| is the number of edges and |V| is the number of vertices.
- Using a more sophisticated **Fibonacci heap**, this can be brought down to **O(|E|+|V| log |V|),** which is asymptotically faster when the graph is dense enough that |E| is **w(|V|)** and linear time when |E| is at least |V| log |V|. For graphs of even greater density (having at

Scanned by CamScanner

least $|V|^c$ edges for some c > 1), Prim's algorithm can be made to run in linear time even more simply, by using a d-ary heap in place of Fibonacci heap.

KRUSKAL'S ALGORITHM:

Minimum Spanning Tree:

- Given a **connected** and **undirected** graph, a spanning tree of that graph is a sub-graph that is a tree and connects all the vertices together.
- A single graph can have many different spanning trees.
- A **Minimum Spanning Tree (MST)** or **minimum weighted spanning tree** for a weighted, connected and undirected graph is **a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of spanning tree is the sum of weights given to each edge of the spanning tree.**
- A minimum spanning tree has **(V-1)** edges where **V** is the number of vertices in the given graph.

Steps for finding MST using Kruskal's Algorithm:

1. Sort all the edges in ascending order of their weights
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge, else discard it. [This step uses **Union-Find Algorithm** to detect cycle.]
3. Repeat step #2 until there are (V-1) edges in the spanning tree.

- The algorithm is **Greedy Algorithm.** The Greedy choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far.

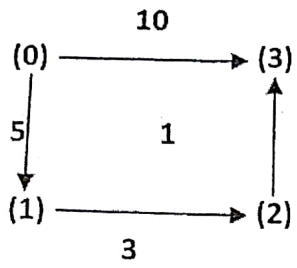
- **Time Complexity: O(E log E) or O(E log V).**
- Sorting of edges takes O(E log E) time. After sorting, we iterate through all edges and apply find-union algorithm.
- The find-union operation can take almost O(log V) time.
- So overall time complexity is: O(E log E + E log V) time
- The value of E can be at most $O(V^2)$, so O(log V) and O(log E) are same, therefore overall time complexity is: O(E log V).

Floyd And Warshall's Algorithm:

- A weighted directed graph is a collection of vertices connected by weighted edges (where as weight is some real number).
- This algorithm is used for solving the All Pair Shortest Path Problem.
- The problem is to find shortest distances between every pair of vertices in a given edge weighted directed graph.
- Dijsktra's Algorithm doesn't works with negative edges.
- Note that value of **graph[i][j] = 0**: if i == j, and **graph[i][j] = ∞ (infinity)**: if there is no direct edge from vertex i to j.

Scanned by CamScanner

Input Graph:



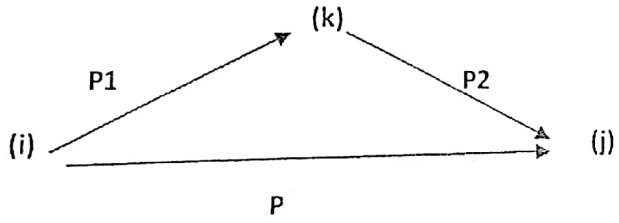
Input:
Matrix Representation:

	0	1	2	3
0	0	5	∞	10
1	∞	0	3	∞
2	∞	∞	0	1
3	∞	∞	∞	0

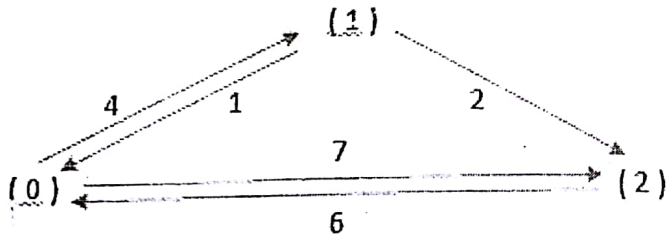
Output:
Shortest Distance Matrix:

	0	1	2	3
0	0	5	8	9
1	∞	0	3	4
2	∞	∞	0	1
3	∞	∞	∞	0

- We initialize the solution matrix same as the input graph matrix as a first step.
- Then we update the solution matrix by considering all vertices as an intermediate vertex.
- The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.
- When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of source and destination vertices respectively, there are two possible cases.
  1) k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.
  2) k is an intermediate vertex in shortest path from i to j.
  We update the value of dist[i][j] = dist[i][k] + dist[k][j], if dist[i][j] >dist[i][k] + dist[k][j]



Example:



	0	1	2
0	0	4	7
1	1	0	2
2	6	∞	0

Scanned by CamScanner

Output: Shortest distance to all vertices from src. If there is negative weight cycle, then shortest distance are not calculated, negative weight cycles is reported.

Step 1:

- This step initializes distances from source to all vertices as infinite and distance to source itself as 0.
- Create an array **dist[]** of size **|V|** with all values as infinite except dist[src], where src is source vertex.

Step 2:

- This step calculates shortest distances: do following **|V|-1** times, where |V| is the number of vertices in given graph.
- do following for each edge: u-v
  if(dist[v] >dist[u] + weight(u, v))
  dist[v] = dist[u] + weight(u, v)

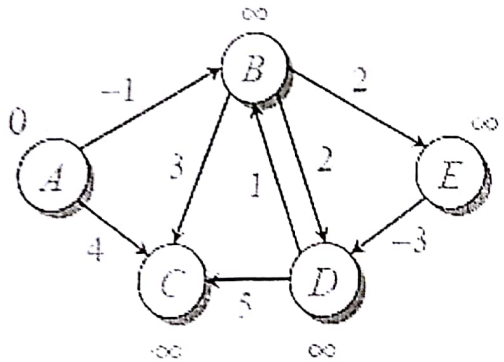
Step 3:

- This step reports if there is a negative weight cycle in graph. Do following for each edge u-v:
  if(dist[v] >dist[u] + weight(u, v)) -> then graph contains negative weight cycle

- the idea of step 3 is, step 2 guarantees shortest distances if graph doesn't contains negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle.

Example:

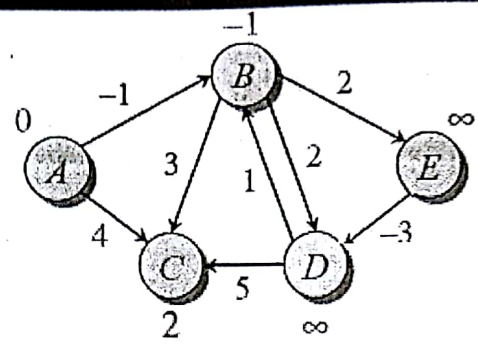
- Let the given source be 0. Initialize all distances as infinite, except the distance to source itself. Total no. of vertices in the graph is 5, so all edges must be processed 4 times.



A	B	C	D	E
0	∞	∞	∞	∞

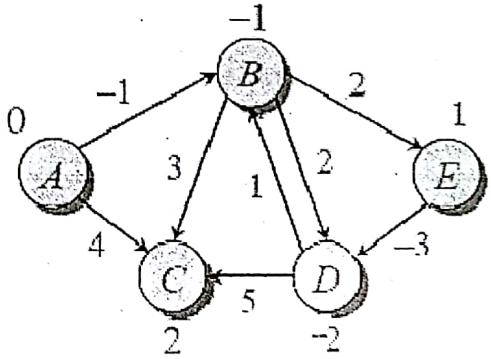
- We get following distances when all edges are processed first time. The first row in figure shows initial distances. The second row shows distances when edges (B,E), (D,B), (B,D) and (A,B) are processed. The third row shows distances when (A,C) is processed. The fourth row shows when (D,C), (B,C) and (E,D) are processed.

Scanned by CamScanner



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞

- **The first iteration guarantees to give all shortest paths which are at most 1 edge long.** We get following distances when all edges are processed second time(the last row shows final values)



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	∞	1
0	−1	2	1	1
0	−1	2	−2	1

- **The second iteration guarantees to give all shortest paths which at most 2 edges long.** The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.

A* Search Algorithm:

- A* Search is the one of the best and popular technique used in path finding and graph traversals.
- This is the most interesting part of A* search algorithm, that is it used in games.
- A* search algorithm is often used to find the shortest path from one point to another point. You can use this for each enemy to find a path to the goal.
- The worst case time complexity is O(E), where E is the number of edges in the graph.
- Auxiliary space in the worst case we can have all the edges inside the open list, so required auxiliary space in worst case is O(V), where V is the total number of vertices.

Johnson's Algorithm for All-pairs shortest paths:

- The problem is to find shortest paths between every pair of vertices in a given weighted directed graph and weights may be negative.

Scanned by CamScanner

- We have discussed Floyd Warshall's algorithm for this problem, using this algorithm, we can find all pair shortest paths in **$O(V^2 \log V + VE)$** time.
- Johnson's algorithm uses both **Dijkstra's** and **Bellman-Ford** as subroutines.
- The idea of Johnson's algorithm is to reweight all edges and make them all positive, then apply Dijkstra's algorithm for every vertex.

Scanned by CamScanner