Annexure-I

SUMMER TRAINING REPORT

A Training Report

Data Structure and Algorithms by GeeksForGeeks

Submitted in partial fulfillment of the requirements for the award of degree of

Bachelor of Technology (Computer Science and Engineering)

Submitted to LOVELY PROFESSIONAL UNIVERSITY,PHAGWARA, PUNJAB

SUBMITTED BY

Name – NIKHIL VEMULAPALLI

Registration number – 12114413

Signature of the student – v.nikhil

Annexure-II: STUDENT DECLARATION

I hereby declare that I have completed my six weeks summer training in Data structure and Algorithm (self-paced) from May2023 to Jun2023 under the guidance of Geeks for Geeks. I have declared that I have worked with full dedication during these six weeks of training and my learning outcomes fulfil the requirements of training for the award of degree of Computer Science and Engineering, Lovely Professional University, Phagwara.

Name of student: Nikhil Vemulapalli

Registration no: 12114413

Student Signature: V.Nikhil

Date: 22-07-2023

CERTIFICATE

CERTIFICATE

OF COURSE COMPLETION

THIS IS TO CERTIFY THAT

Nikhil Vemulapalli

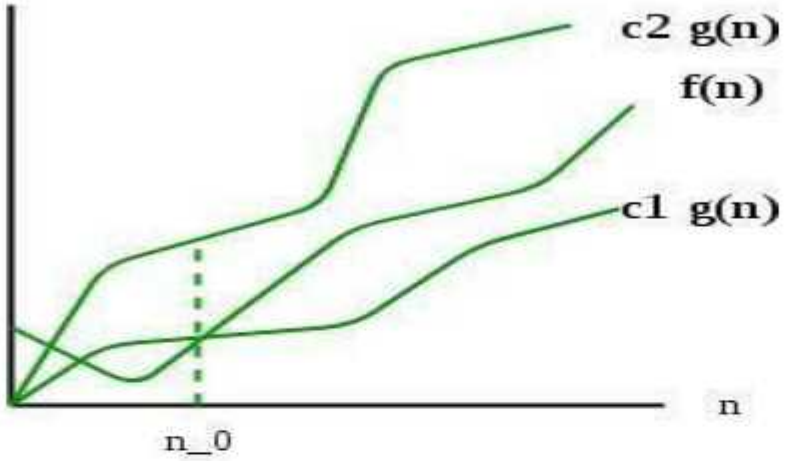
has successfully completed the course on DSA Self paced of duration 8 weeks.

Sandeep Jain

Mr. Sandeep Jain
Founder & CEO, GeeksforGeeks
https://media.geeksforgeeks.org/courses/certificates/f6ac453d8b63ede67050f7e12bb7a156.pdf

Contents Table

Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent the time complexity of algorithms:



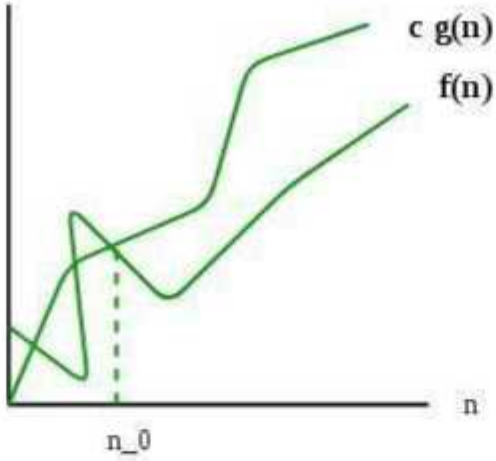
f(n) = theta(g(n))

1) Θ Notation: The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior. A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression.

3n3 + 6n2 + 6000 = Θ(n3) Dropping lower order terms is always fine because there will always be a n0 after which Θ(n3) has higher values than Θn2) irrespective of the constants involved. For a given function g(n), we denote Θ(g(n)) is following set of functions.

Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that
0 <= c1*g(n) <= f(n) <= c2*g(n) for all n >= n

The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n (n >= n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.



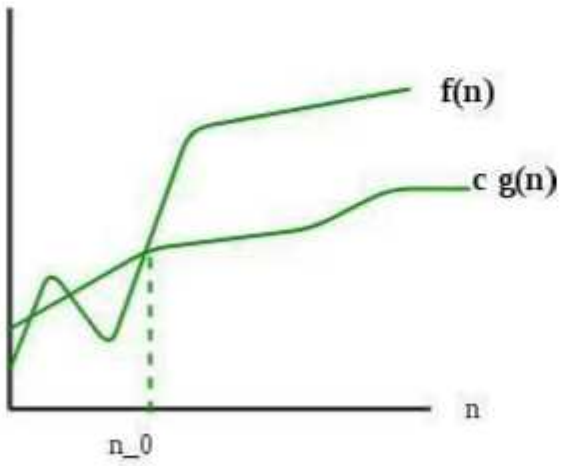
f(n) = O(g(n))

2) Big O Notation: The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. W e can safely say t hat the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time. If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and
worst cases:

1. The worst case time complexity of Insertion Sort is Θ(n^2).
2. The best case time complexity of Insertion Sort is Θ(n).

The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

O(g(n)) = { f(n): there exist positive constants c and n0 such that 0 <= f(n) <= c*g(n) for all n >= n0}



f(n) = Omega(g(n))

3) Ω Notation: Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

Ω Notation can be useful when we have lower bound on time complexity of an algorithm. The Omega notation is the least used notation among all three.

For a given function g(n), we denote by Ω(g(n)) the set of functions.

Ω (g(n)) = {f(n): there exist positive constants c and n0 such that 0 <= c*g(n) <= f(n) for all n >= n0}.

It is important to analyze an algorithm after writing it to find its efficiency in terms of time and space in order to improve it if possible.

When it comes to analyzing algorithms, the asymptotic analysis seems to be the best way possible to do so. This is because asymptotic analysis analyzes algorithms in terms of the input size. It checks how are the time and space growing in terms of the input size. In this post, we will take an example of Linear Search and analyze it using asymptotic analysis.

We can have three cases to analyze an algorithm:

1. Worst Case
2. Average Case
3. Best Case

Below is the algorithm for performing linear search:

```
// Linearly search x in arr[].
// If x is present then return the index,
// otherwise return -1
search(int arr[], int n, int x)
{
int i;
for (i=0; i
{ if (arr[i] ==
x)
return true;
} return
false; }
```

Worst Case Analysis (Usually done) in the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes the maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array.

When x is not present, the search () functions compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be O(N), where N is the number of elements in the array.

Average Case Analysis (Sometimes done) in average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of

cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array). So we sum all the cases and divide the sum by (N+1). Following is the value of average case time complexity.

$$\text{Average Case Time} = \frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)}$$

$$= \frac{\theta((n+1)*(n+2)/2)}{(n+1)}$$

$$= \Theta(N)$$

Best Case Analysis (Bogus) : In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on N). So time complexity in the best case would be O(1)

Important Points:

- Most of the times, we do the worst case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is a good piece of information.
- The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.
- The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

Chapter – 2
Recursion, Bit Magic, Basic Mathematics

What is Recursion?

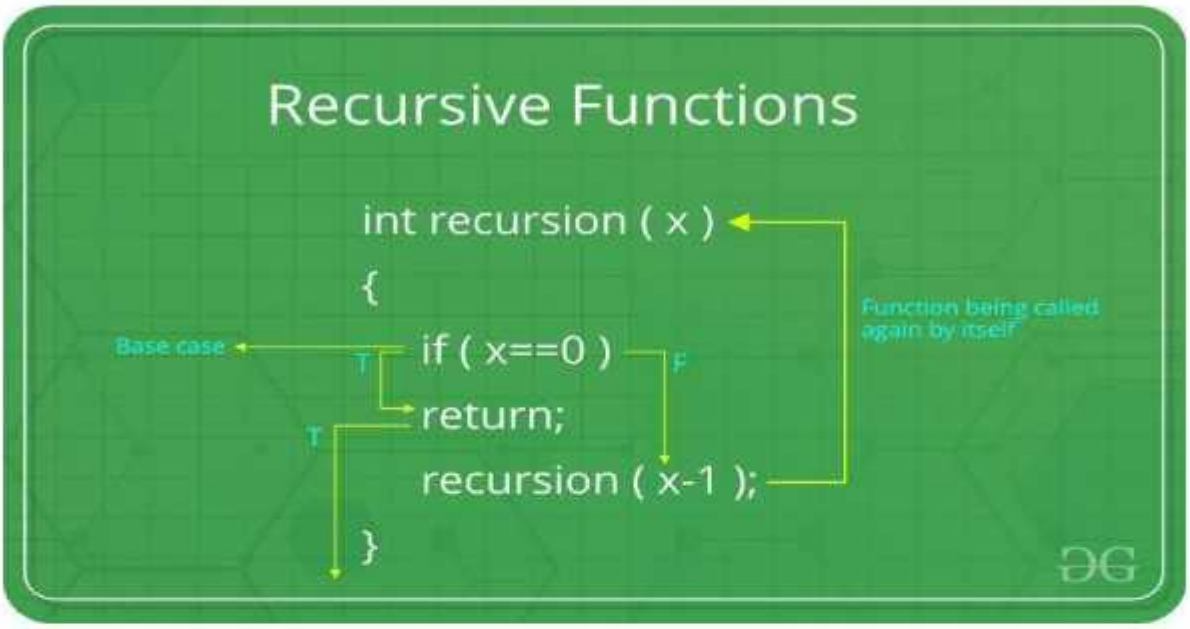
The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is
called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. Examples of
such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.

What is base condition in recursion?

In recursive program, the solution to base case is provided and solution of bigger problem is expressed in terms of smaller problems.

```
int fact(int n)
{ if (n < = 1) // base case return
      1;
      else return n*fact(n1); }
```

In the above example, base case for n < = 1 is defined and larger value of number can be solved by converting to smaller one till base case is reached.



How a particular problem is solved using recursion?

The idea is represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop recursion. For example, we compute factorial n if we know factorial of (n-1). Base case for factorial would be n = 0. We return 1 when n = 0.

Why Stack Overflow error occurs in recursion? If base case is not reached or not defined, then stack overflow problem may arise. Let us take an example to understand this.

```
int fact(int n)
{
// wrong base case (it may cause
// stack overflow).
if (n ==
100) return
1; else return
n*fact(n-1); }
```

If fact(10) is called, it will call fact(9), fact(8), fact(7) and so on but number will never reach 100. So, the base case is not reached. If the memory is exhausted by these functions on stack, it will cause stack overflow error.

How memory is allocated to different function calls in recursion?

When any function is called from main(), the memory is allocated to it on stack. A recursive function calls itself, the memory for called function is allocated on top of memory allocated to calling function and different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called and memory is de-allocated and the process continues.

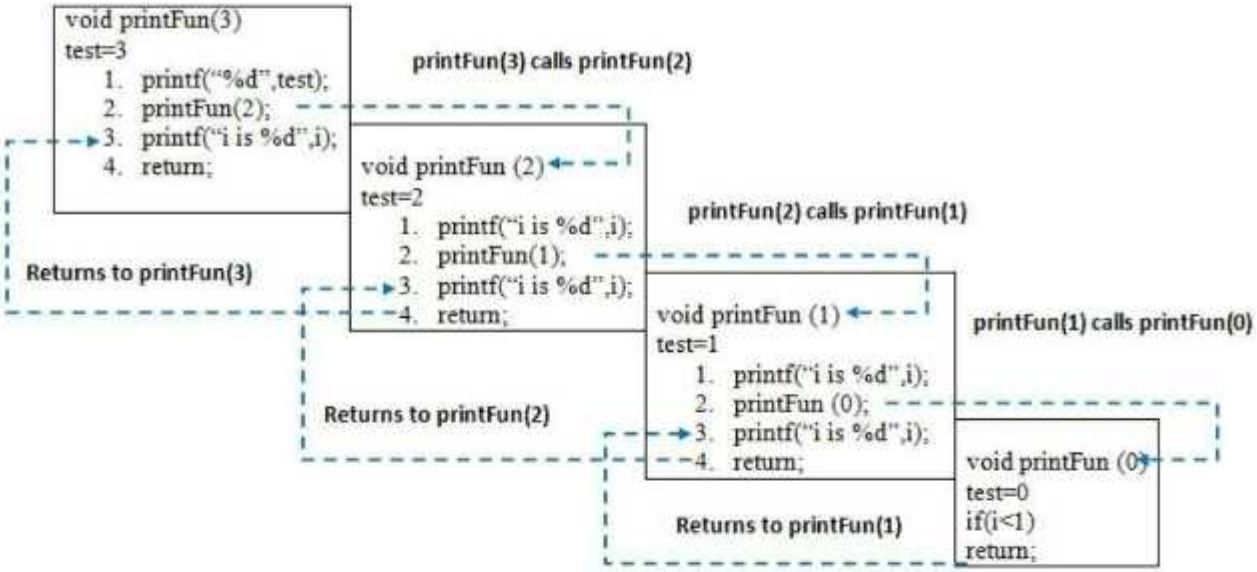
Let us take the example how recursion works by taking a simple function:

```
void printFun(int test)
{ if (test <
1) return; else
{
print test; printFun(test1);
// Calling function
printFun() int test = 3;
printFun(test);
```

Output :

3 2 1 1 2 3

When **printFun(3)** is called from main(), memory is allocated to **printFun(3)** and a local variable test is initialized to 3 and statement 1 to 4 are pushed on the stack as shown in below diagram. It first prints '3'. In statement 2, **printFun(2)** is called and memory is allocated to **printFun(2)** and a local variable test is initialized to 2 and statement 1 to 4 are pushed in the stack. Similarly, **printFun(2)** calls **printFun(1)** and **printFun(1)** calls **printFun(0)**. **printFun(0)** goes to if statement and it return to **printFun(1)**. Remaining statements of **printFun(1)** are executed and it returns to **printFun(2)** and so on. In the output, value from 3 to 1 are printed and then 1 to 3 are printed. The memory stack has been shown in below diagram.


void printFun(3)
test=3
1. printf("%d",test);
2. printFun(2);
3. printf("i is %d",i);
4. return;
printFun(3) calls printFun(2)
void printFun (2)
test=2
1. printf("i is %d",i);
2. printFun(1);
3. printf("i is %d",i);
4. return;
printFun(2) calls printFun(1)
Returns to printFun(3)
void printFun (1)
test=1
1. printf("i is %d",i);
2. printFun (0);
3. printf("i is %d",i);
4. return;
printFun(1) calls printFun(0)
Returns to printFun(2)
void printFun (0)
test=0
if(i<1)
return;
Returns to printFun(1)

Disadvantage of Recursion: Note that both recursive and iterative programs have same problem solving powers, i.e., every recursive program can be written iteratively and vice versa is also true. Recursive program has greater space requirements than iterative program as all functions will remain in stack until base case is reached. It also has greater time requirements because of function calls and return overhead.

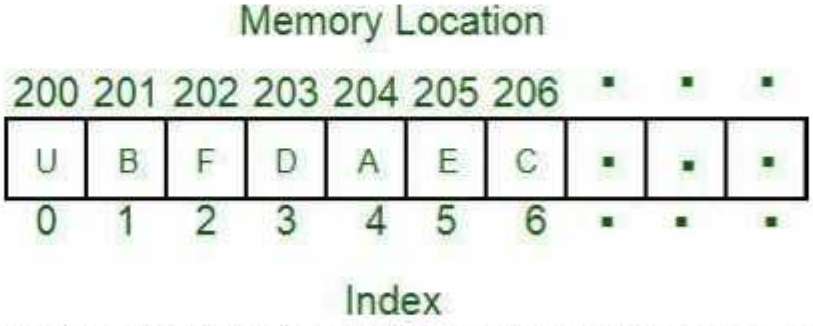
Advantages of Recursion: Recursion provides a clean and simple way to write code. Some problems are inherently recursive like tree traversals, Tower of Hanoi, etc. For such problems it is preferred to write recursive code. We can write such codes also iteratively with the help of stack data structure.

Arrays

An array is a collection of items of same data type stored at contiguous memory locations. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array.

For simplicity, we can think of an array a fleet of stairs where on each step is placed a value (let's say one of your friends). Here, you can identify the location of any of your friends by simply knowing the count of the step they are on.

Remember: "Location of next index depends on the data type we use".



The above image can be looked as a top-level view of a staircase where you are at the base of staircase. Each element can be uniquely identified by their index in the array (in a similar way as you could identify your friends by the step on which they were on in the above example).

Defining an Array: Array definition are similar to defining any other variable. There are two things need to be kept in mind, data type of the array elements and the size of the array. The size of the array is fixed and the memory for an array needs to be allocated before use, size of an array cannot be increased or decreased dynamically.

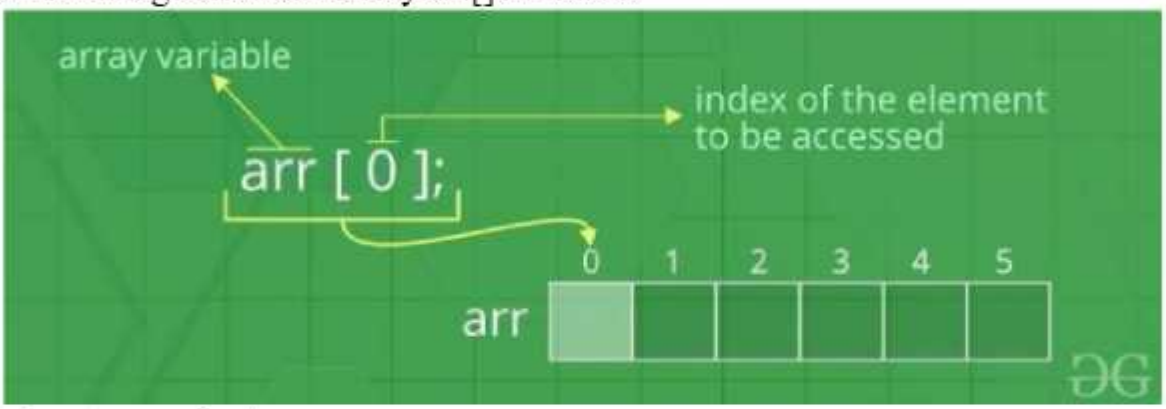
Generally, arrays are declared as:

```
dataType arrayName[arraySize];
```

An array is distinguished from a normal variable by brackets [and].

Accessing array elements: Arrays allows to access elements randomly. Elements in an array can be accessed using indexes. Suppose an array named arr stores N elements. Indexes in an array are in the range of 0 to N-1, where the first element is present at 0-th index and consecutive elements are placed at consecutive indexes. Element present at i^{th} index in the array arr[] can be accessed as arr[i].

Below image shows an array arr[] of size 5:



Advantages of using arrays:

- Arrays allow random access of elements. This makes accessing elements by position faster.
- Arrays have better cache locality that can make a pretty big difference in performance.

Examples:

```
// A character array in C/C++/Java

char arr1[] = {'g', 'e', 'e', 'k', 's'};


// An Integer array in C/C++/Java int arr2[] = {10, 20, 30, 40, 50};


// Item at i'th index in array is typically accessed

// as "arr[i]". For example arr1[0] gives us 'g'

// and arr2[3] gives us 40.
```

Searching in an Array

Searching an element in an array means to check if a given element is present in an array or not. This can be done by accessing elements of the array one by one starting from the first element and checking if any of the element matches with the given element.

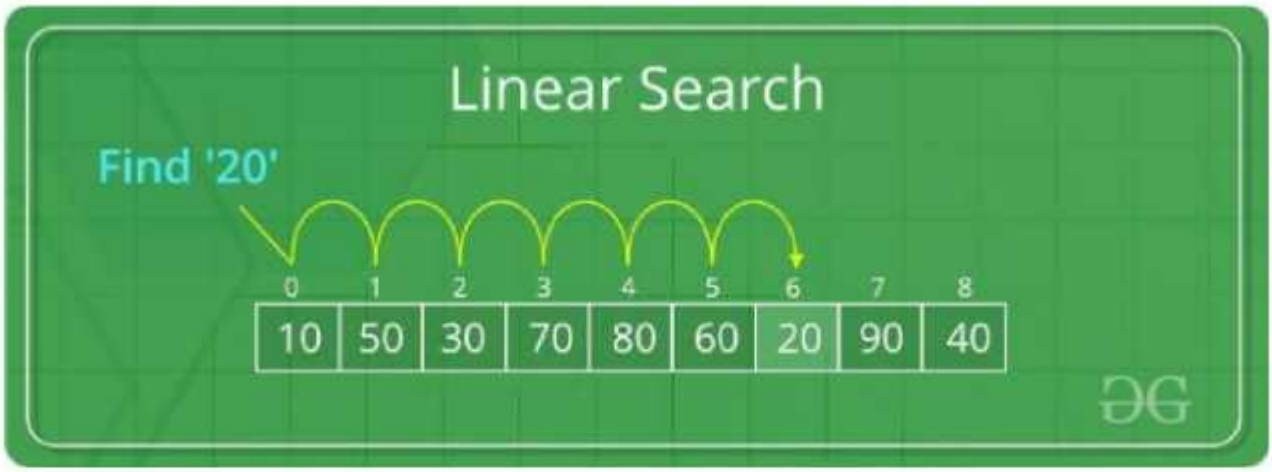
We can use loops to perform the above operation of array traversal and access the elements using indexes.
Suppose the array is named arr[] with size N and the element to be searched is referred as key. Below is the algorithm to perform to the search operation in the given array.

```
for(i = 0; i < N; i++)
{ if(arr[i] == key)
      { print "Element Found";
      } else
      { print "Element not Found"; }
}
```

Time Complexity of this search operation will be O(N) in the worst case as we are checking every element of the array from 1st to last, so the number of operations is N.

Chapter – 3 Searching, Sorting and Hashing

Linear Search means to sequentially traverse a given list or array and check if an element is present in the respective array or list. The idea is to start traversing the array and compare elements of the array one by one starting from the first element with the given element until a match is found or end of the array is reached.



Examples :

```
Input : arr[] = {10, 20, 80, 30, 60, 50,
110, 100, 130, 170}
key = 110; Output :
6
Element 110 is present at index 6

Input : arr[] = {10, 20, 80, 30, 60, 50,
          110, 100, 130, 170} key = 175;
Output : -1
Element 175 is not present in arr[].
```

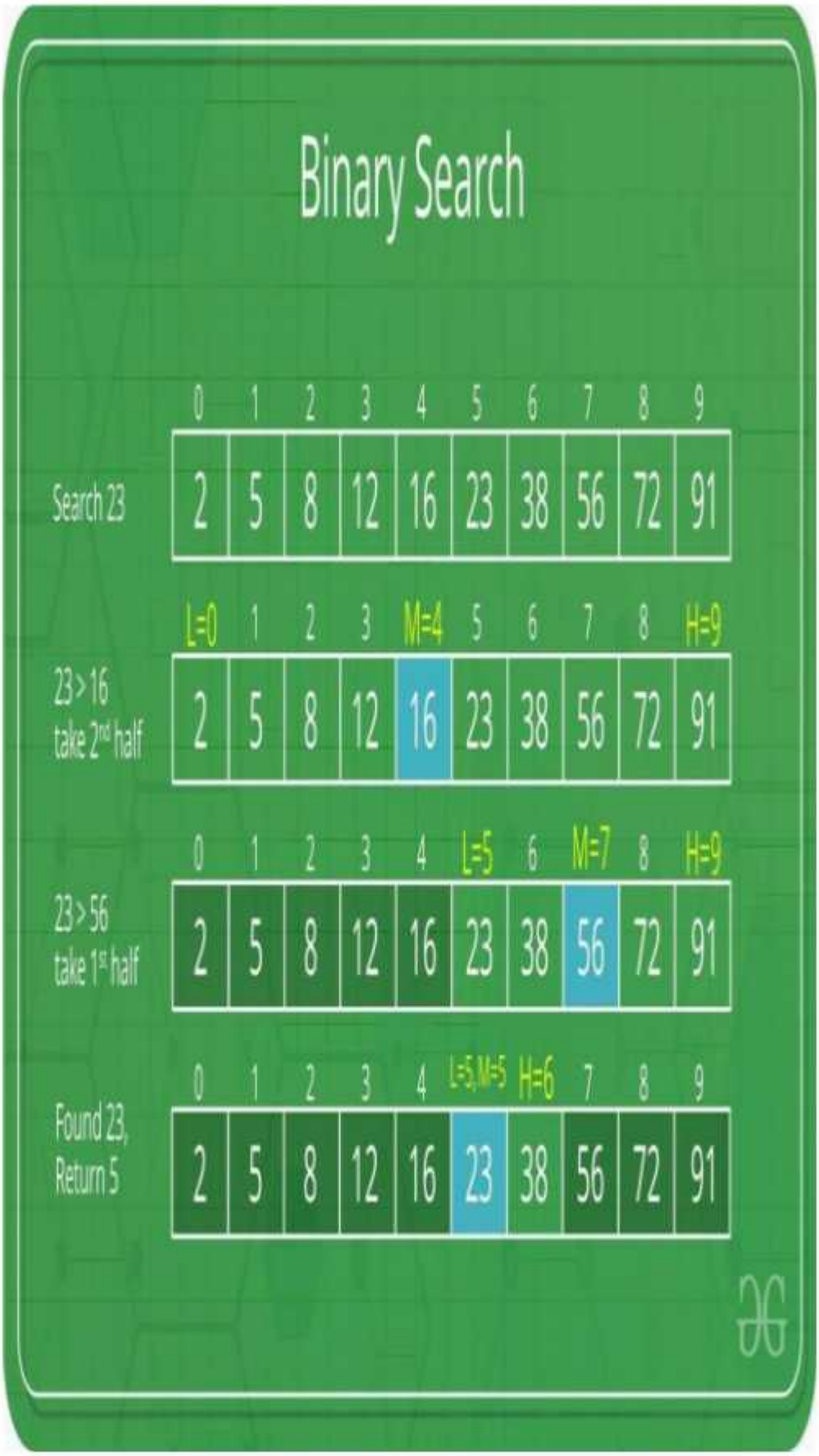
Problem: Given an array **arr[]** of **N** elements, write a function to search a given element **X** in arr[].

Algorithm:

- Start from the leftmost element of arr[] and one by one compare X with each element of arr[].
- If X matches with an element, return the index.
- If X doesn't match with any of elements and end of the array is reached, return -1

Binary Search is a searching algorithm for searching an element in a sorted list or array. Binary Search is efficient than Linear Search algorithm and performs the search operation in logarithmic time complexity for sorted arrays or lists.

Binary Search performs the search operation by repeatedly dividing the search interval in half. The idea is to begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.


Binary Search
Search 23
0 1 2 3 4 5 6 7 8 9
2 5 8 12 16 23 38 56 72 91
23 > 16
take 2nd half
L=0 1 2 3 M=4 5 6 7 8 H=9
2 5 8 12 16 23 38 56 72 91
23 > 56
take 1st half
0 1 2 3 4 L=5 6 M=7 8 H=9
2 5 8 12 16 23 38 56 72 91
Found 23,
Return 5
0 1 2 3 4 L=5,M=5 H=6 7 8 9
2 5 8 12 16 23 38 56 72 91

Problem: Given a sorted array arr[] of N elements, write a function to search a given element X in arr[] using *Binary Search Algorithm*.

Algorithm: We basically ignore half of the elements just after one comparison.

- Compare X with the middle element of the array.
- If X matches with middle element, we return the mid index.
- Else If X is greater than the mid element, then X can only lie in right half subarray after the mid element. So we will now look for X in only the right half ignoring the complete left half.
- Else if X is smaller, search for X in the left half ignoring the right half.

Sorting any sequence means to arrange the elements of that sequence according to some specific criterion.

For Example, the array arr[] = {5, 4, 2, 1, 3} after *sorting in increasing order* will be: arr[] = {1, 2, 3, 4, 5}. The same array after *sorting in descending order* will be: arr[] = {5, 4, 3, 2, 1}.

In-Place Sorting: An in-place sorting algorithm uses constant extra space for producing the output (modifies the given array only). It sorts the list only by modifying the order of the elements within the list.

In this tutorial, we will see three of such in-place sorting algorithms, namely:

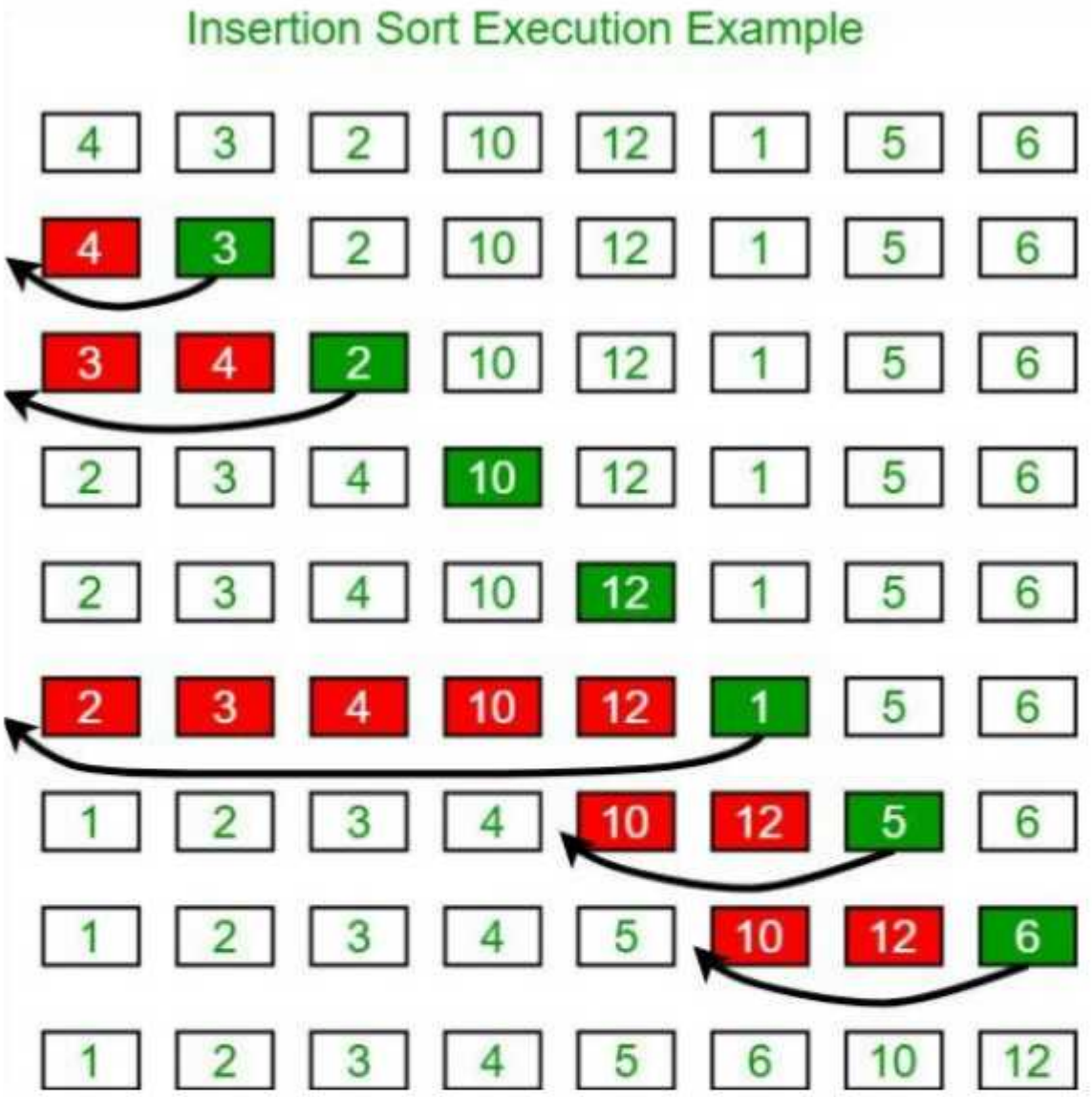
- Insertion Sort
- Selection Sort
- Bubble Sort

Insertion Sort

Insertion Sort is an In-Place sorting algorithm. This algorithm works in a similar way of sorting a deck of playing cards.

The idea is to start iterating from the second element of array till last element and for every element insert at its correct position in the subarray before it.

In the below image you can see, how the array [4, 3, 2, 10, 12, 1, 5, 6] is being sorted in increasing order following the insertion sort algorithm.



Algorithm:

Step 1: If the current element is 1st element of array, it is already sorted. Step 2: Pick next element
Step 3: Compare the current element will all elements in the sorted sub-array before it.
Step 4: Shift all of the elements in the sub-array before the current element which are greater than the current element by one place and insert the current element at the new empty space.
Step 5: Repeat step 2-3 until the entire array is sorted.

Another Example: arr[] = {**12**, 11, 13, **5**, 6}

Let us loop for i = 1 (second element of the array) to 4 (Size of input array - 1).

- *i = 1*, Since 11 is smaller than 12, move 12 and insert 11 before 12. **11, 12,** 13, 5, 6
- *i = 2*, 13 will remain at its position as all elements in A[0..I-1] are smaller than 13 **11, 12, 13,** 5, 6
- *i = 3*, 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position. **5, 11, 12, 13,** 6
- *i = 4*, 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.

5, 6, 11, 12,13

Bubble Sort

Bubble Sort is also an in-place sorting algorithm. This is the simplest sorting algorithm and it works on the principle that:

In one iteration if we swap all adjacent elements of an array such that after swap the first element is less than the second element then at the end of the iteration, the first element of the array will be the minimum element.

Bubble-Sort algorithm simply repeats the above steps N-1 times, where N is the size of the array.

Example: Consider the array, arr[] = {5, 1, 4, 2, 8}.

- **First Pass:** (**5 1** 4 2 8) --> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.

(1 **5 4** 2 8) --> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) --> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) --> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

- **Second Pass:** (**1 4** 2 5 8) --> (**1 4** 2 5 8)

(1 **4 2** 5 8) --> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) --> (1 2 **4 5** 8)
(1 2 4 **5 8**) --> (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

- **Third Pass:** (**1 2** 4 5 8) --> (**1 2** 4 5 8)

(1 **2 4** 5 8) --> (1 **2 4** 5 8)
(1 2 **4 5** 8) --> (1 2 **4 5** 8)
(1 2 4 **5 8**) --> (1 2 4 **5 8**)

Time Complexity: **$O(N^2)$**

Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1. The subarray which is already sorted.
2. Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Following example explains the above steps:

arr[] = 64 25 12 22 11.

// Find the minimum element in arr[0...4]
// and place it at beginning
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4] 11
12 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4] 11
12 **22** 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning
of arr[3...4]
11 12 22 **25** 64

Hashing is a method of storing and retrieving data from a database efficiently.

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

1. Insert a phone number and corresponding information.
2. Search a phone number and fetch the information.
3. Delete a phone number and related 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 arrays and keep the data sorted, then a phone number can be searched in O(Logn) time using Binary Search, but insert and delete operations become 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(Logn) 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 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 is 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 a 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 Array, 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.
Hashing is an improvement over Direct Access Table. The idea is to use hash function that converts a given phone number or any other key to a smaller number and uses the small number as index in a table called hash table.

Hash Function: A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in 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. 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.

Collision Handling: Since a hash function gets 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 collisions:

- **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.
- **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 slots until the desired element is found or it is clear that the element is not in the table.

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 the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed).

Important Operations:

- Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.
- Search(k): Keep probing until slot's key doesn't become equal to k or an empty slot is reached.
- Delete(k): ***Delete operation is interesting***. If we simply delete a key, then search

may fail. So slots of deleted keys are marked specially as "deleted".

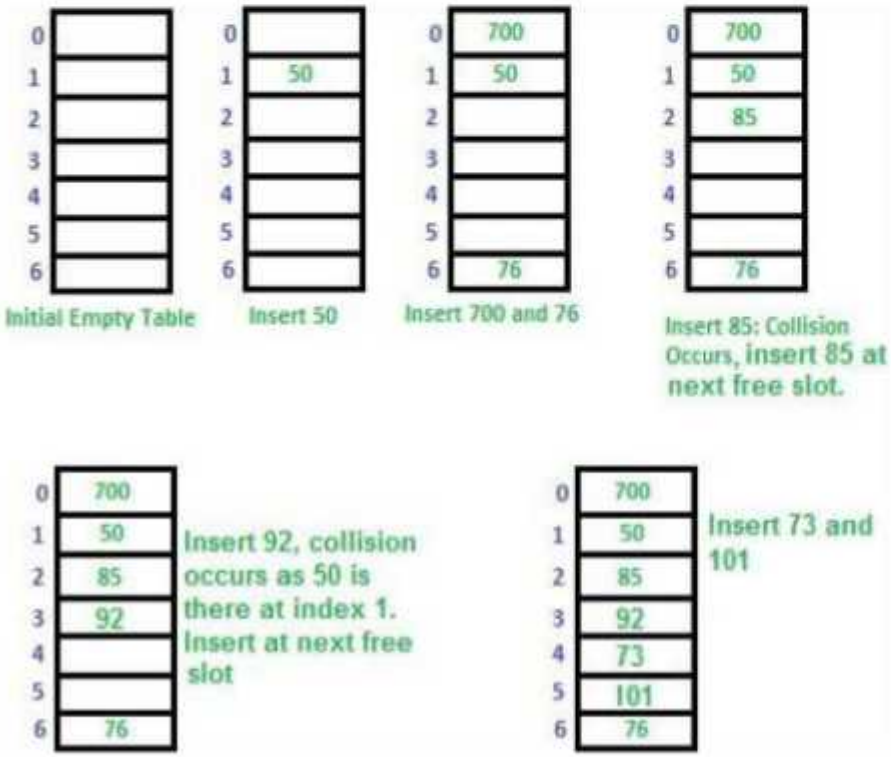
Insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

Open Addressing is done in following ways:

1. ***Linear Probing:*** In linear probing, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also. let **hash(x)** be the slot index computed 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
   .....
   .....

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



Clustering: The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.

2. ***Quadratic Probing*** We look for i^2'th slot in i'th iteration.

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

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

3. **Double Hashing** we use another hash function hash2(x) and look for i*hash2(x) slot in i'th rotation.

Let hash(x) be the slot index computed using hash
Function. 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
.....
....

See this for step by step diagrams.

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.
- 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.

S.No.	**Seperate Chaining**	**Open Addressing**
1.	Chaining is Simpler to implement.	Open Addressing requires more computation. In chaining, Hash table never fills up, we can always add
2.	More elements to chain.	In open addressing, table may become full.

Chaining is less sensitive to the hash function or load open addressing requires extra care for to avoid

3.	Factors. Clustering and load factor.	Chaining is mostly used when it is unknown how many and Open addressing is used when the frequency and
4.	How frequently keys may be inserted or deleted. Number of keys is known. Cache performance of chaining is not good as keys are stored	Open addressing provides better cache performance
5.	Using linked list. As everything is stored in the same table.	

Wastage of Space (Some Parts of hash table in chaining are In Open addressing, a slot can be used even if an

6. Never used). Input doesn't map to it.
7. Chaining uses extra space for links. No links in Open addressing

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

m = Number of slots in the hash table n = Number of keys to be inserted in the hash table

Load factor $\alpha = n/m$ (< 1)

Expected time to search/insert/delete $< 1/(1 - \alpha)$

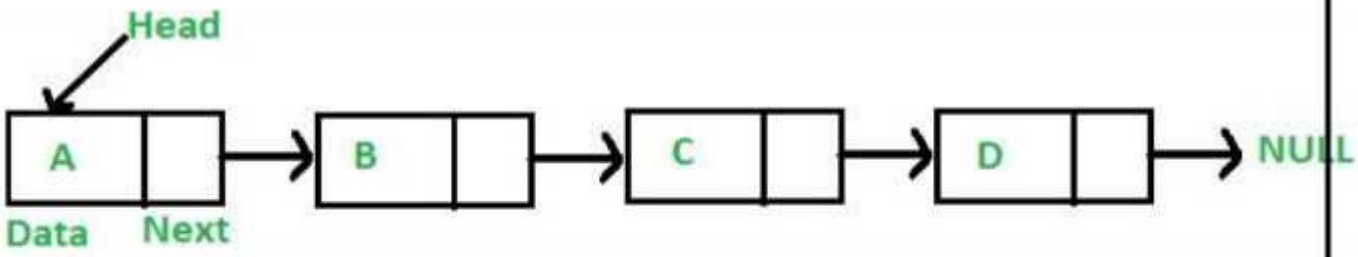
So Search, Insert and Delete take $(1/(1 - \alpha))$ time

Chapter-5 Linked List,Stack, queue

Linked Lists are linear or sequential data structures in which elements are stored at noncontiguous memory location and are linked to each other using pointers.

Like arrays, linked lists are also linear data structures but in linked lists elements are not stored at contiguous memory locations. They can be stored anywhere in the memory but for sequential access, the nodes are linked to each other using pointers.

Data: This part stores the data value of the node. That is the information to be stored at the current node.
Next Pointer: This is the pointer variable or any other variable which stores the address of the next node in the memory.



Advantages of Linked Lists over Arrays: Arrays can be used to store linear data of similar types, but arrays have the following limitations:

The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage. On the other hand, linked lists are dynamic and the size of the linked list can be incremented or decremented during runtime.
Inserting a new element in an array of elements is expensive, because a room has to be created for the new elements and to create room, existing elements have to shift.

For example, in a system, if we maintain a sorted list of IDs in an array id[].

```
id[] = [1000, 1010, 1050, 2000, 2040].
```

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000). Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

On the other hand, nodes in linked lists can be inserted or deleted without any shift operation and is efficient than that of arrays.

1. Random access is not allowed in Linked Lists. We have to access elements sequentially starting from the first node. So we cannot do a binary search with linked lists efficiently with its default implementation. Therefore, lookup or search operation is costly in linked lists in comparison to arrays.
2. Extra memory space for a pointer is required with each element of the list.
3. Not cache friendly. Since array elements are present at contiguous locations, there is a locality of reference which is not there in case of linked lists.

Representation of Linked Lists

A linked list is represented by a pointer to the first node of the linked list. The first node is called the head node of the list. If the linked list is empty, then the value of the head node is NULL.

Disadvantages of Linked Lists:
Each node in a list consists of at least two parts:

1. data
2. Pointer (Or Reference) to the next node

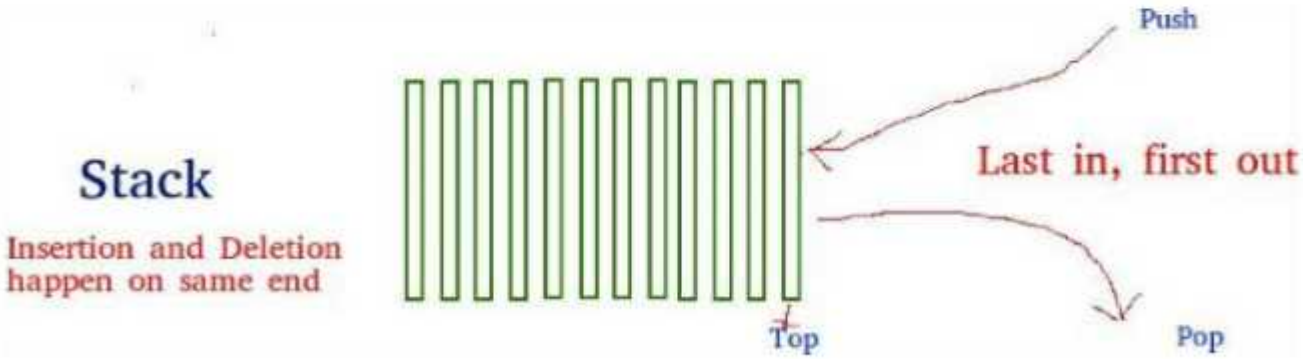
.

Stacks:
The ***Stack*** is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

- The LIFO order says that the element which is inserted at the last in the Stack will be the first one to be removed. In LIFO order insertion takes place at the rear end of the stack and deletion occurs at the front of the stack.
- The FILO order says that the element which is inserted at the first in the Stack will be the last one to be removed. In FILO order insertion takes place at the rear end of the stack and deletion occurs at the front of the stack.

Mainly the following three basic operations are performed in the stack:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false.
-



How to understand a stack practically?

There are many real-life examples of a stack. Consider the simple example of plates stacked over one another in a canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottom most position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO/FILO order.

Time Complexities of operations on stack: The operations push(), pop(), isEmpty() and peek() all take O(1) time. We do not run any loop in any of these operations.

Implementation: There are two ways to implement a stack.

- Using array
- Using linked list

Pros: The linked list implementation of stack can grow and shrink according to the needs at runtime. **Cons:** Requires extra memory due to involvement of pointers.

Applications of stack:

- Stacks can be used to check for the balancing of paranthesis in an expression.
- Infix to Postfix/Prefix conversion.
- Redo-undo features at many places like editors, photoshop.
- Forward and backward feature in web browsers And Many More...

QUEUE:

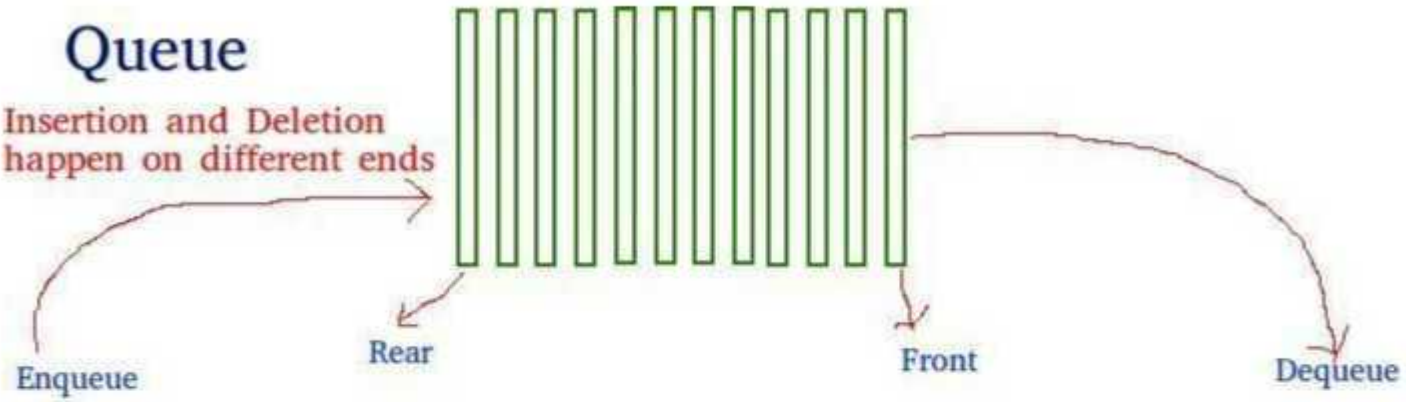
Like *Stack* data structure, ***Queue*** is also a linear data structure which follows a particular order in which the operations are performed. The order is **F**irst **I**n **F**irst **O**ut (FIFO) which means that the element which is inserted first in the queue will be the first one to be removed from the queue. A good example of queue is any queue of consumers for a resource where the consumer that came first is served first.

The difference between stacks and queues is in removing. In a stack we remove the most recently added item; in a queue, we remove the least recently added item.

Operations on Queue: Mainly the following four basic operations are performed on queue:

- **Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.
- **Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.

- **Front:** Get the front item from queue.
- **Rear:** Get the last item from queue.



Array implementation Of Queue: For implementing a *queue*, we need to keep track of two indices, front and rear. We enqueue an item at the rear and dequeue an item from the front. If we simply increment front and rear indices, then there may be problems, the front may reach the end of the array. The solution to this problem is to increase front and rear in a circular manner.

Consider an Array of size **N** is taken to implement a queue. Initially, the size of the queue will be zero(0). The total capacity of the queue will be the size of the array i.e. N. Now initially,the index *front* will be equal to 0, and *rear* will be equal to N-1. Every time on inserting an item, the index *rear* will increment by one, so increment it as: **rear = (rear + 1)%N** and everytime on removing an item, the front index will shift to right by 1 place so increment it as: **front = (front + 1)%N**. **Example**: *Array = queue[N]. front = 0, rear = N-1.*

N = 5.

Operation 1: *enque(5);*
front = 0, rear = (N-1 + 1)%N = 0.
Queue contains: [5]. **Operation 2**: *enque(10);* front = 0, rear = (rear + 1)%N = (0 + 1)%N = 1. Queue contains: [5, 10]. **Operation 3**: *enque(15);* front = 0, rear = (rear + 1)%N = (1 + 1)%N = 2.
Queue contains: [5, 10, 15].
Operation 4:
deque(); print queue[front]; front = (front + 1)%N = (0 + 1)%N = 1. Queue contains: [10, 15].

Time Complexity: Time complexity of all operations like enqueue(), dequeue(), isFull(), isEmpty(), front() and rear() is O(1). There is no loop in any of the operations.

Applications of Queue: Queue is used when things don't have to be processed immediatly, but have to be processed in **F**irst **I**n**F**irst **O**ut order like Breadth First Search. This property of Queue makes it also useful in following kind of scenarios.

1.When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.

2.When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

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

Below is a pictorial representation of Circular Linked List

Implementation:

To implement a circular singly linked list, we take an external pointer that points to the last node of the list. If we

have a pointer last pointing to the last node, then *last -> next* will point to the first node.

The pointer ***last*** points to node **Z** and ***last -> next*** points to the node **P**.

Why have we taken a pointer that points to the last node instead of first node? For insertion of node in the beginning we need traverse the whole list. Also, for insertion and the end, the whole list has to be traversed. If instead of start pointer we take a pointer to

the last node then in both the cases there won't be any need to traverse the whole list. So insertion in the begging or at the end takes constant time irrespective of the length of the list.

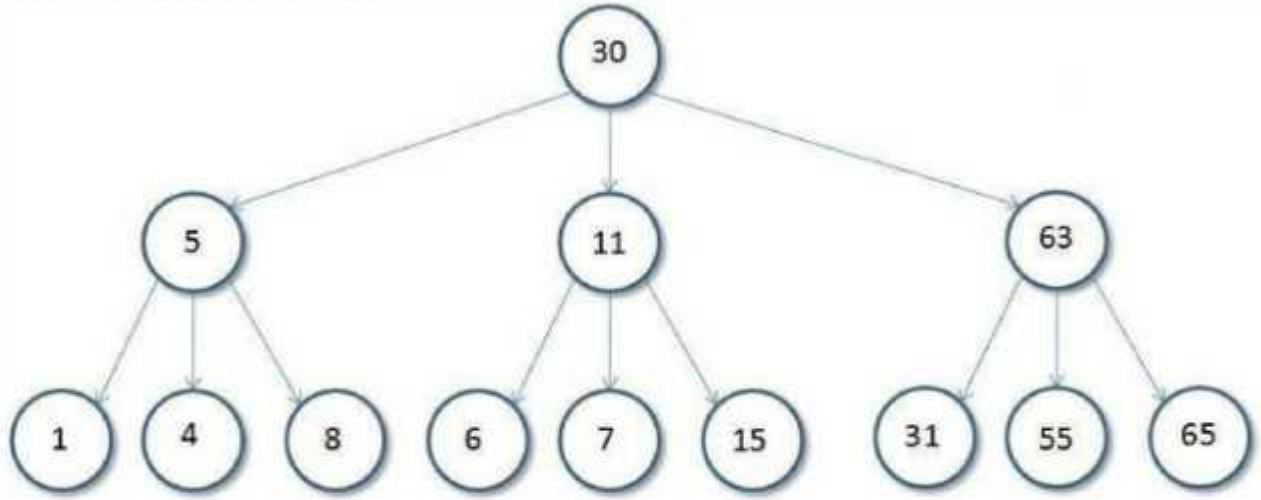
Chapter-6

Tree, BST, Heap, Graph

Trees:

A Tree is a non-linear data structure where each node is connected to a number of nodes with the help of pointers or references.

A Sample tree is as shown below:



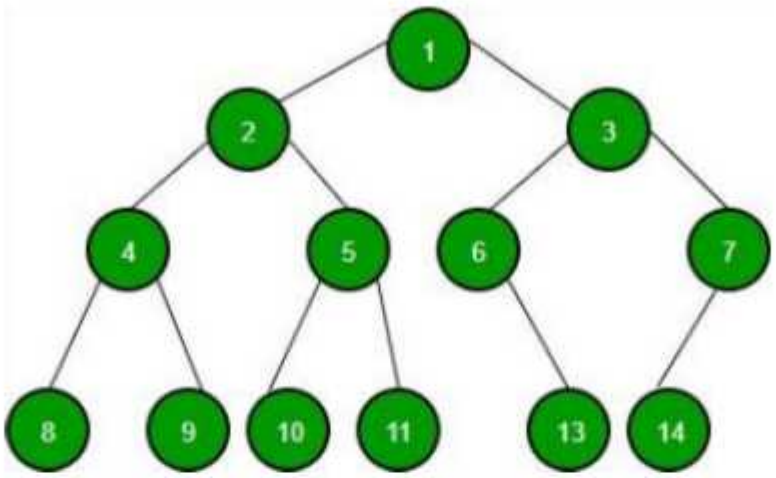
Basic Tree Terminologies:

- **Root**: The root of a tree is the first node of the tree. In the above image, the root node is the node 30.
- **Edge**: An edge is a link connecting any two nodes in the tree. For example, in the above image there is an edge between node **11** and **6**.
- **Siblings**: The children nodes of same parents are called siblings of each other. That is, the nodes with same parents are called siblings. In the above tree, nodes *5, 11 and 63* are siblings.
- **Leaf Node**: A node is said to be the leaf node if it has no children. In the above tree, node **15** is one of the leaf node.
- **Height of a Tree**: Height of a tree is defined as the total number of levels in the tree or the length of the path from the root node to the node present at the last level. The above tree is of height **2**.

Binary Tree

A Tree is said to be a Binary Tree if all of its nodes have atmost 2 children. That is, all of its node can have either no child, 1 child or 2 child nodes.

Below is a sample **Binary Tree**:



Properties of a Binary Tree:

The maximum number of nodes at level 'l' of a binary tree is (2^{l-1}). Level of root is 1.

This can be proved by induction.
For root, l = 1, number of nodes = $2^{1-1} = 1$
Assume that the maximum number of nodes on level l is 2^{l-1}.
Since in Binary tree every node has at most 2 children, next level would have twice nodes, i.e $2 * 2_{l-1}$.
Maximum number of nodes in a binary tree of height 'h' is $(2^h - 1)$.
Here height of a tree is the maximum number of nodes on the root to leaf path. The height of a tree with a single node is considered as 1.
This result can be derived from point 2 above. A tree has maximum nodes if all levels have maximum nodes. So maximum number of nodes in a binary tree of height h is $1 + 2 + 4 + .. + 1$.
This is a simple geometric series with h terms and sum of this series is $2^h - 1$.
In some books, the height of the root is considered as 0. In this convention, the above formula becomes
$_{h+1} - 1$.

In a Binary Tree with N nodes, the minimum possible height or the minimum number of levels is $Log_2(N+1)$. This can be directly derived from point 2 above. If we consider the convention where height of a leaf node is considered as 0, then above formula for minimum possible height becomes $Log_2(N+1) - 1$.

4. ***A Binary Tree with L leaves has at least ($Log_2L + 1$) levels.*** A Binary tree has maximum number of leaves (and minimum number of levels) when all levels are fully filled. Let all leaves be at level l, then below is true for number of leaves L.

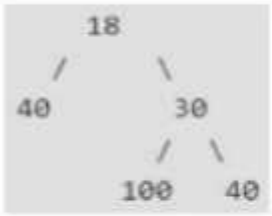
   $L <= 2^{l-1}$ [From Point 1] $l = Log_2L + 1$ where l is the minimum number of levels.

5. ***In a Binary tree where every node has 0 or 2 children, number of leaf nodes is always one more than nodes with two children.***

   $L = T + 1$
   Where L = Number of leaf nodes
   T = Number of internal nodes with two children

Types of Binary Trees: Based on the structure and number of parents and children nodes, a Binary Tree is classified into the following common types:

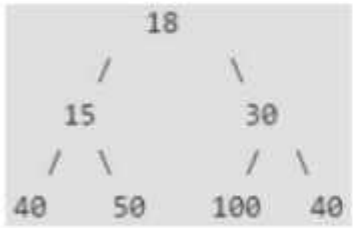
- **Full Binary Tree**: A Binary Tree is full if every node has either 0 or 2 children. The following are examples of a full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaves have two children.



In a Full Binary, number of leaf nodes is number of internal nodes plus 1.

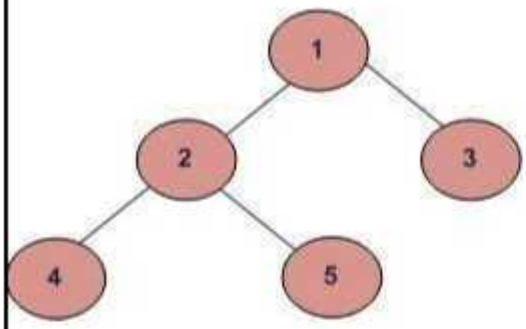
• **Complete Binary Tree**: A Binary Tree is complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible .Following are examples of Complete Binary Trees:

• **Perfect Binary Tree**: A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at the same level.
Following are examples of Perfect Binary Trees:



A Perfect Binary Tree of height h (where height is the number of nodes on the path from the root to leaf) has 2h – 1 node.

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



- Inorder (Left, Root, Right) : 4 2 5 1 3 • Preorder (Root, Left, Right) : 1 2 4 5 3.
- Postorder (Left, Right, Root) : 4 5 2 3 1

Lets look at each of these tree traversal algorithms in details:

- **Inorder Traversal :** In Inorder traversal a node is processed after processing of all nodes in its left subtree. The right subtree of the node is processed after processing the node itself.

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left->subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right>subtree)

Example: Inorder traversal for the above-given tree is 4 2 5 1 3.

- **Preorder Traversal:** In preorder traversal a node is processed before processing any of the nodes in its subtree.

Example: Preorder traversal for the above-given tree is 1 2 4 5 3.

Algorithm Preorder(tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(rightsubtree)

- **Postorder Traversal:** In post order traversal, a node is processed after processing all of the nodes in its subtrees.

Algorithm Postorder(tree)

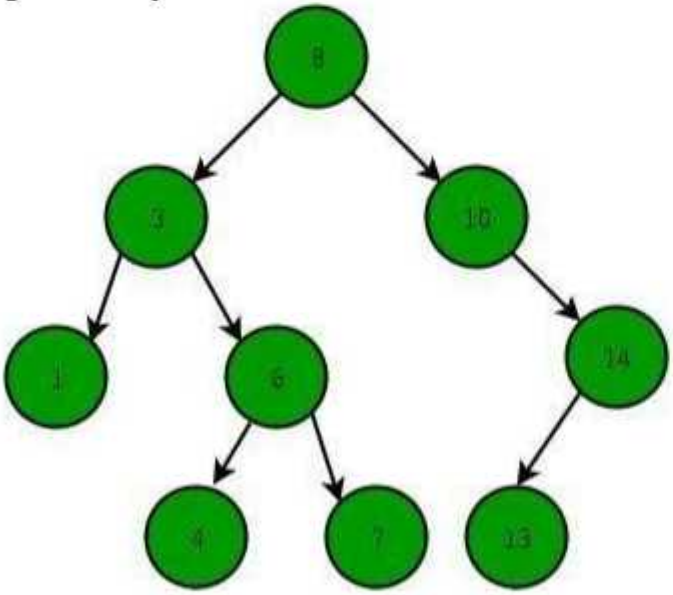
1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(rightsubtree)
3. Visit the root.

Binary Search Tree:

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than or equal to the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree. There must be no duplicate nodes.

Sample Binary Search Tree:



The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast in comparison to normal

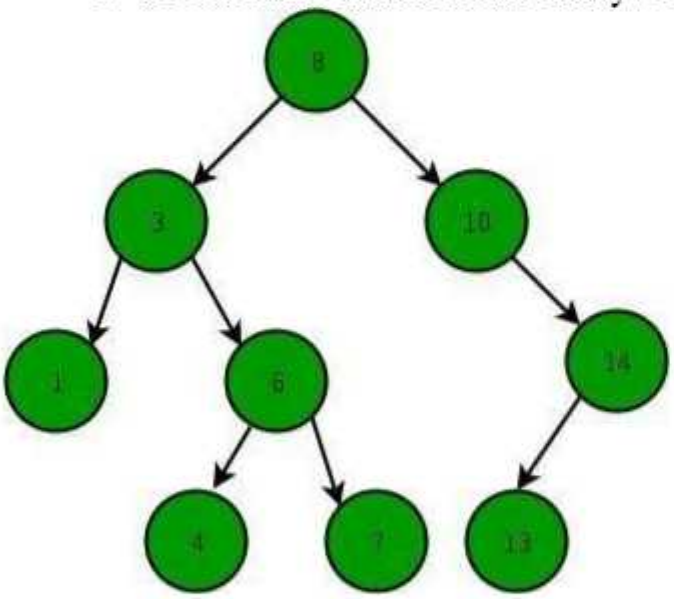
Searching a Key

Binary Trees. If there is no ordering, then we may have to compare every key to search a given key.

Using the property of Binary Search Tree, we can search for an element in O(h)

To search a given key in Binary Search Tree, first c ompare it with root, if the key is present at root, return root. If the key is greater than the root's key, we recur for the right subtree of the root node. Otherwise, we recur for the left subtree.

1. Start from root.
2. Compare the key element with root, if less than root, then recur for left subtree, else recur for right subtree.
3. If element to search is found anywhere, return true, else return false.




```
Step 1:   Compare 6 with 8. Since 6 is less than 8, move to left subtree.
Step 2:   Compare 6 with 3. Since 6 is greater than 3, move to its right subtree.
Step 3:   Comapre 6 with 6.
Node Found.
```

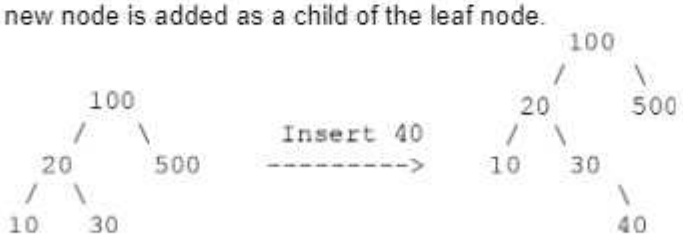
Inserting a new node in the Binary Search Tree is always done at the leaf nodes to maintain the order of nodes in the Tree. The idea is to start searching the given node to be inserted from the root node till we hit a leaf node.

For Example:

time complexity where **h** is the height of the given BST.

Insertion of a Key

Once a leaf node is found, the new node is added as a child of the leaf node.



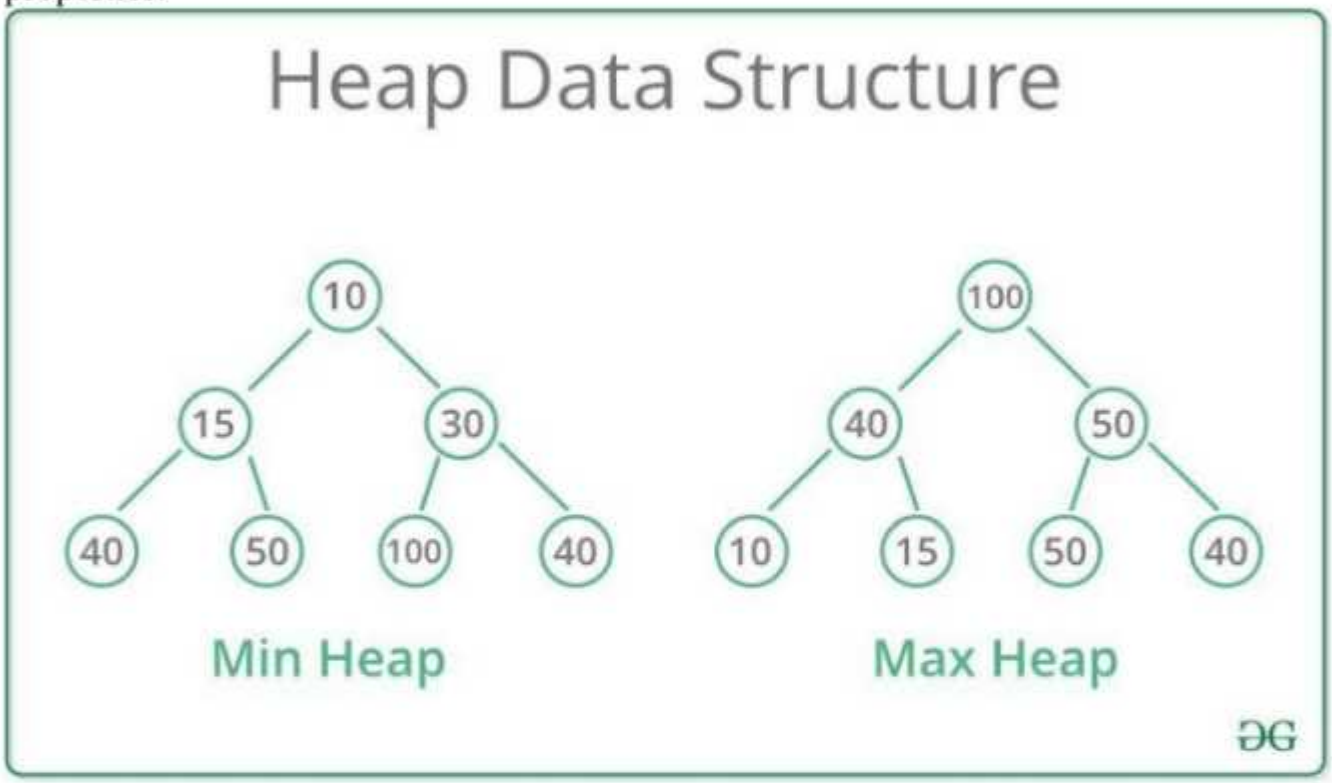
Heaps:

A Heap is a Tree-based data structure, which satisfies the below properties:

1. A Heap is a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible).

2. A Heap is either Min Heap or Max Heap. In a Min-Heap, the key at root must be minimum among all keys present in the Binary Heap. The same property must be recursively true for all nodes in the Tree. Max Heap is similar to MinHeap.

Binary Heap: A Binary Heap is a heap where each node can have at most two children. In other words, a Binary Heap is a complete Binary Tree satisfying the above-mentioned properties.



Representing Binary Heaps

Since a Binary Heap is a complete Binary Tree, it can be easily represented using Arrays.

• The root element will be at Arr[0]. • Below table shows indexes of other nodes for the i^{th} node, i.e., Arr[i]:

Arr[(i-1)/2] Returns the parent node
Arr[(2*i)+1] Returns the left child node
Arr[(2*i)+2] Returns the right child node

Getting Maximum Element: In a Max-Heap, the maximum element is always present at the root node which is the first element in the array used to represent the Heap. So, the maximum

element from a max heap can be simply obtained by returning the root node as Arr[0] in O(1) time complexity.

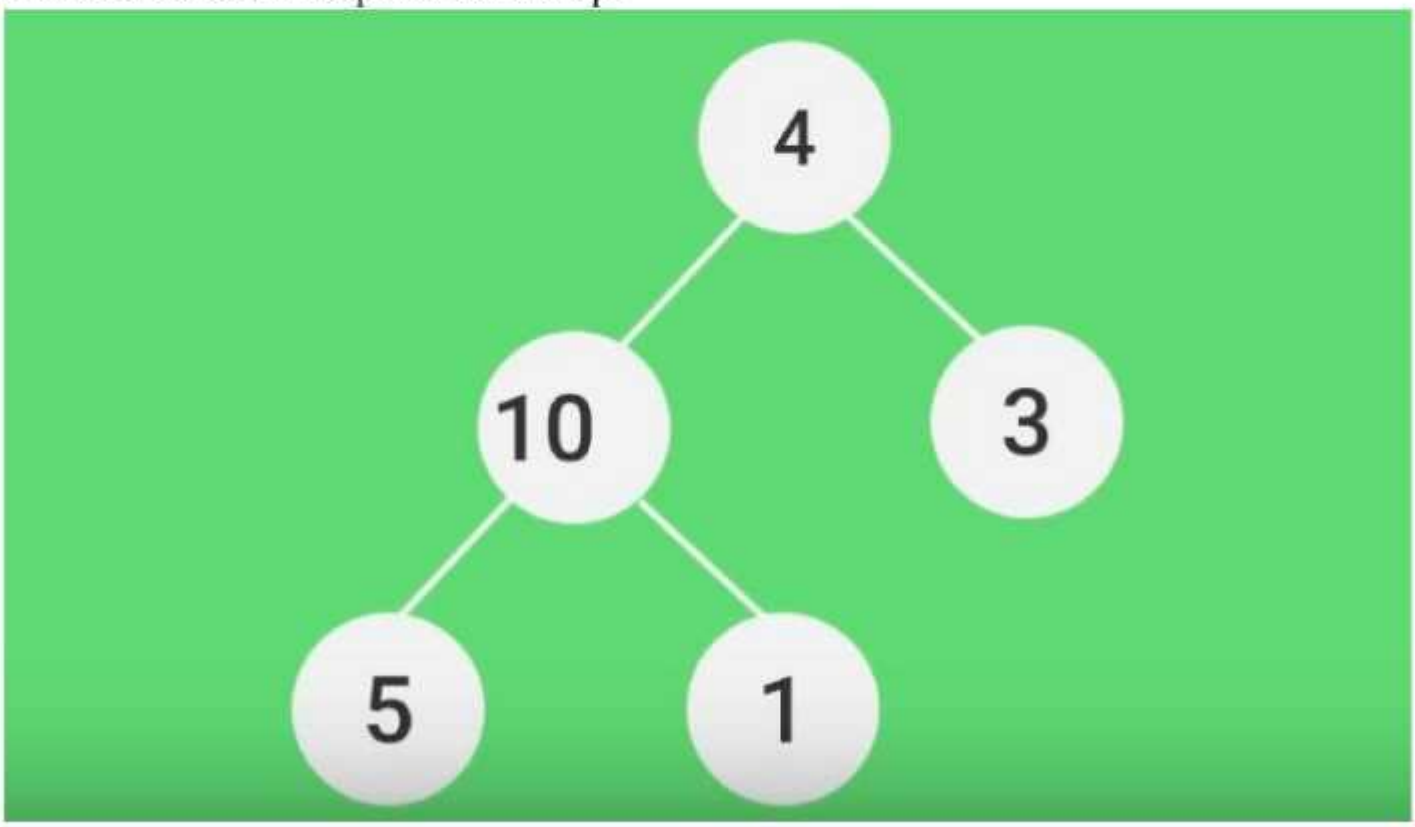
Getting Minimum Element: In a Min-Heap, the minimum element is always present at the root node which is the first element in the array used to represent the Heap. So, the minimum element from a minheap can be simply obtained by returning the root node as Arr[0] in O(1) time complexity.

Heapifying an Element

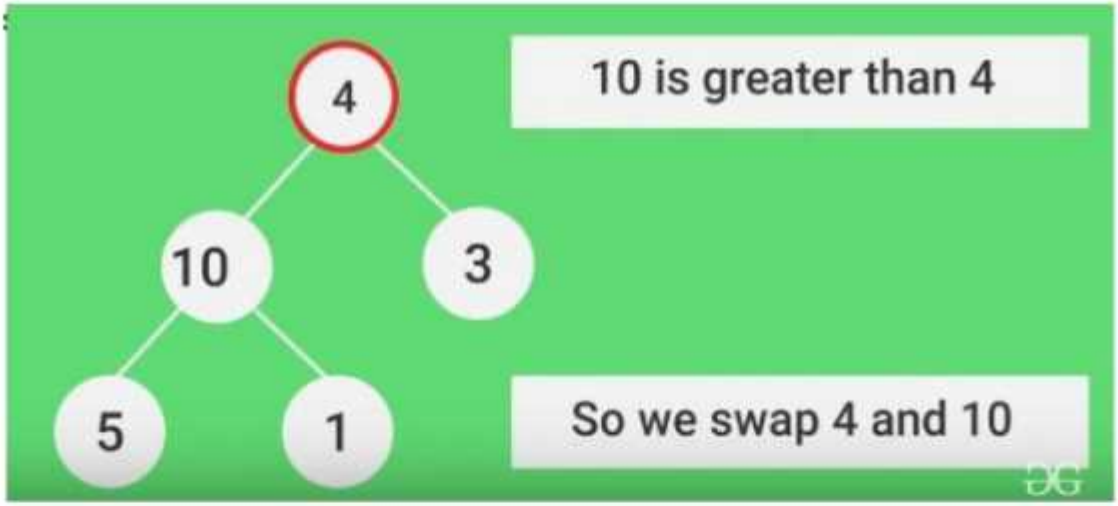
Generally, on inserting a new element onto a Heap, it does not satisfy the property of Heap as stated above on its own. The process of placing the element at the correct location so that it satisfies the Heap property is known as Heapify.

Heapifying in a Max Heap: The property of Max Heap says that every node's value must be greater than the values of its children nodes. So, to **heapify** a particular node swap the value of the node with the maximum value of its children nodes and continue the process until all of the nodes below it satisfies the Heap property.

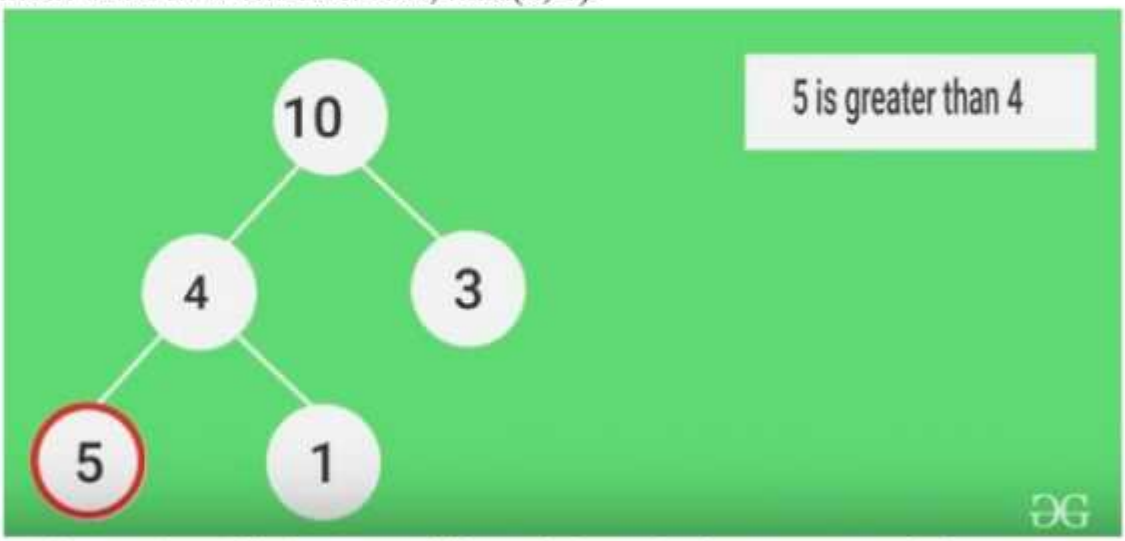
Consider the below heap as a Max-Heap:



In the above heap, node **4** does not follow the Heap property. Let's heapify the root node, ***node 4***.

- 

- **Step 2**: Again the node 4 does not follow the heap property. Swap the node 4 with the maximum of its new childrens i.e., max(5, 1).



- The Node 4 is now heapified successfully and placed at it's correct position.

Time Complexity: The time complexity to heapify a single node is **O(h)**, where h is equal to **log(N)** in a complete binary tree where N is the total number of nodes.

Graphs:

A ***Graph*** is a data structure that consists of the following two components:

1. A finite set of vertices also called nodes.
2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered

because (u, v) is not the same as (v, u) in case of a directed graph(digraph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

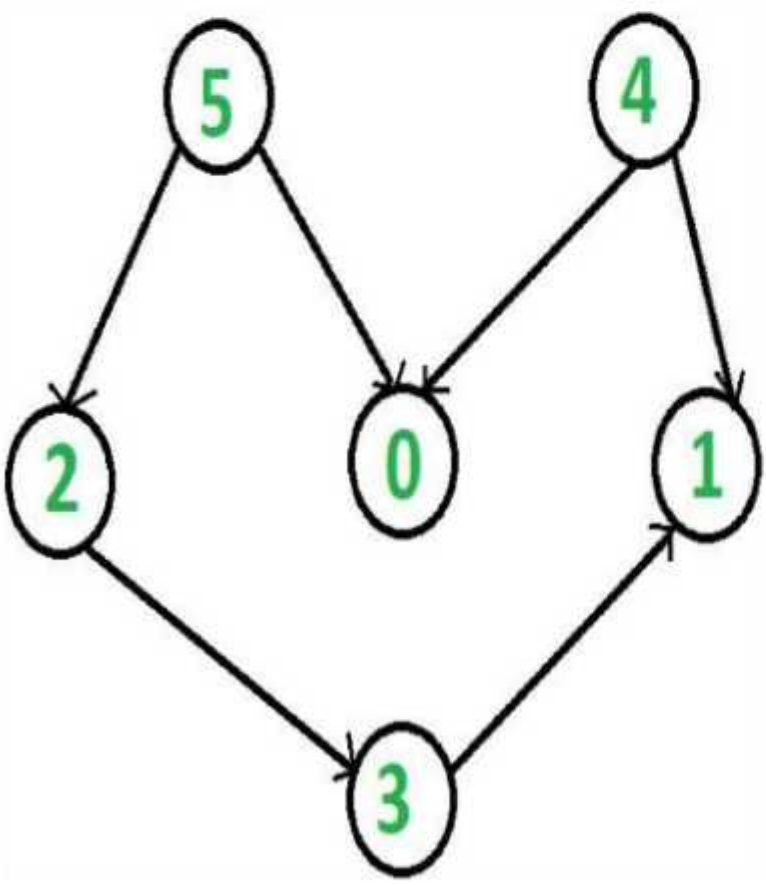
Graphs are used to represent many real-life applications:

- Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. For example Google GPS
- Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender and locale.

Directed and Undirected Graphs

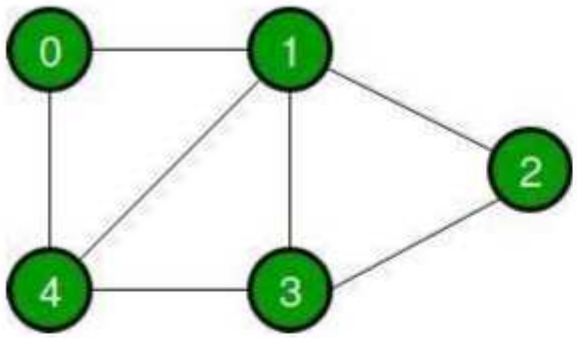
- **Directed Graphs**: The Directed graphs are such graphs in which edges are directed in a single direction.

For Example, the below graph is a directed graph:



- **Undirected Graphs**: Undirected graphs are such graphs in which the edges are directionless or in other words bi-directional. That is, if there is an edge between vertices **u** and **v** then it means we can use the edge to go from both **u to v** and **v to u**.

Following is an example of an undirected graph with 5 vertices:



Representing Graphs

Following two are the most commonly used representations of a graph:

1. Adjacency Matrix.
2. Adjacency List.

Let us look at each one of the above two method in details:

• **Adjacency Matrix:** The Adjacency Matrix is a 2D array of size V x 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. 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.

The adjacency matrix for the above example undirected graph is:

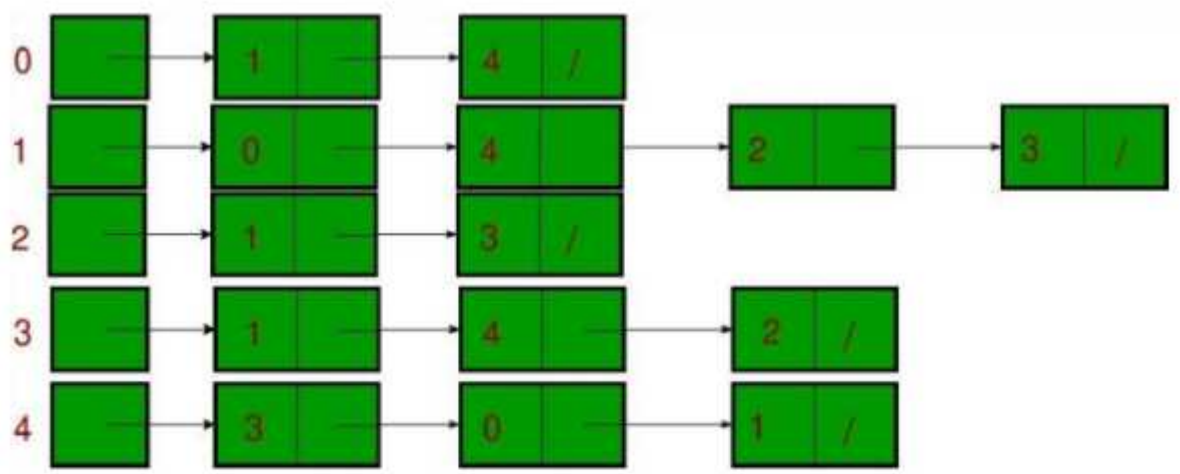
	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

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 O(1).

Cons: Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.

Please see this for a sample Python implementation of adjacency matrix.

Adjacency List: Graph can also be implemented using an array of lists. That is every index of the array will contain a complete list. Size of the array is equal to the number of vertices and every index **i** in the array will store the list of vertices connected to the vertex numbered *i*. Let the array be array[]. An entry array[i] represents the list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above example undirected graph.



Below is the implementation of the adjacency list representation of Graphs:

Note: In below implementation, we use dynamic arrays (vector in C++/ArrayList in Java) to represent adjacency lists instead of a linked list. The vector implementation has advantages of cache friendliness.

Chapter:7 Greedy, Dynamic

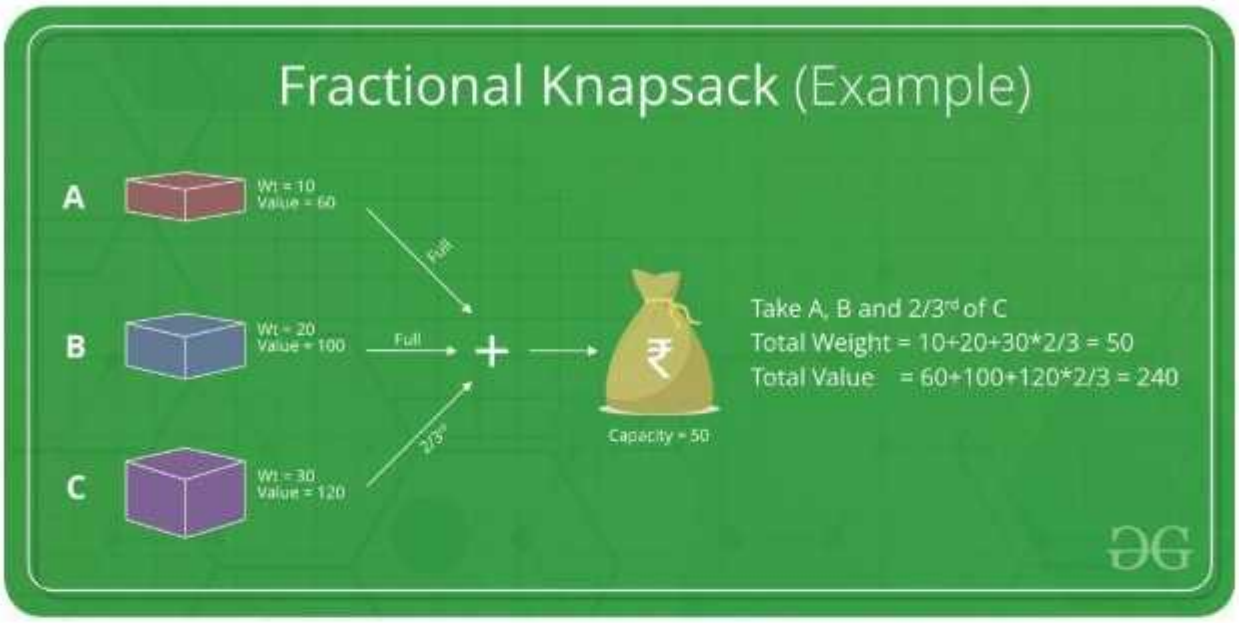
Programming Greedy Technique:

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So the problems where choosing locally optimal also leads to the global optimal solution are best fit for Greedy.

For example, consider the **Fractional Knapsack Problem**. The problem states that:

Given a list of elements with specific values and weights associated with them, the task is to fill a Knapsack of weight W using these elements such that the value of knapsack is maximum possible.

***Note**: You are allowed to take a fraction of an element also in order to maximize the value.*



In general, the **Greedy Algorithm** can be applied to solve a problem if it satisfies the below property:

At every step, we can make a choice that looks best at the moment, and we get the optimal solution of the complete problem.

Let us consider one more problem known as the **Activity Selection Problem** to understand the use of Greedy Algorithms.

Problem: You are given N activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.

```
Example 1: Consider the following 3 activities sorted by
finish time.
     start[] = {10, 12, 20}; finish[]
     = {20, 25, 30};
A person can perform at most two activities.
The maximum set of activities that can be
executed is {0, 2} [ These are indexes in
start[] and finish[] ]

Example 2: Consider the following 6 activities
sorted by finish time.
     start[] = {1, 3, 0, 5, 8, 5}; finish[]
     = {2, 4, 6, 7, 9, 9};
A person can perform at most four activities. The
maximum set of activities that can be executed is
```

```
{0, 1, 3, 4} [ These are indexes in start[] and finish[]
]
```

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.

To do this:

1. Sort the activities according to their finishing time.
2. Select the first activity from the sorted array and print it.
3. Do following for remaining activities in the sorted array.
    - If the start time of this activity is greater than or equal to the finish time of previously selected activity then select this activity and print it.

Dynamic Programming:

Dynamic Programming is an algorithmic approach to solve some complex problems easily and save time and number of comparisons by storing the results of past computations. The basic idea of dynamic programming is to store the results of previous calculation and reuse it in future instead of recalculating them.

We can also see Dynamic Programming as dividing a particular problem into subproblems and then storing the result of these subproblems to calculate the result of the actual problem.

Consider the problem to ***find the N-th Fibonacci number***.

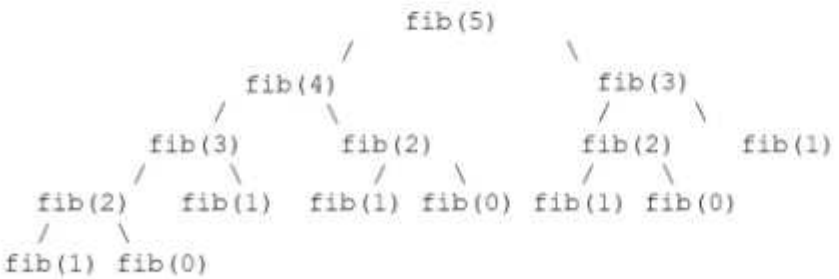
We know that n-th fibonacci number fib(n) can be defined as:

```
fib(n) = fib(n-1) + fib(n-2), where n >= 2.

and,

fib(0) = 0
fib(1) = 1
```

We can see that the above function fib() to find the nth fibonacci number is divided into two subproblems fib(n-1) and fib(n-2) each one of which will be further divided into subproblems and so on.

The first few Fibonacci numbers are:

1, 1, 2, 3, 5, 8, 13, 21, 34,.....
Below is the recursion tree for the recursive solution to find the N-th Fibonacci number:



We can see that the function fib(3) is being called 2 times. If we would have stored the value of fib(3), then instead of computing it again, we could have reused the old stored value.

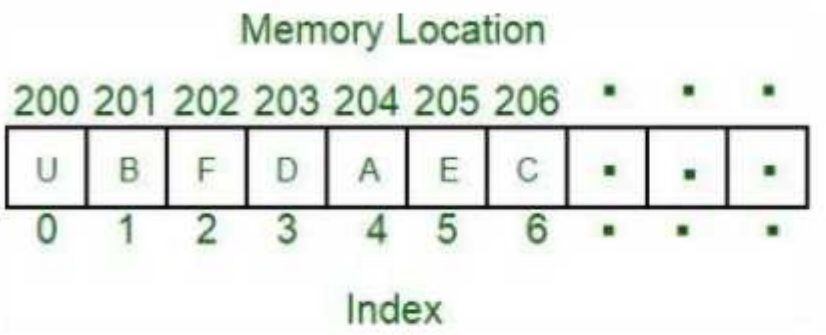
The **time complexity** of the *recursive solution* is **exponential**.

Chapter-8 Application of Data structure

A data structure is a particular way of organizing data in a computer so that it can be used effectively. In this article, the real-time applications of all the data structures are discussed.

Application of Arrays:

Arrays are the simplest data structures that stores items of the same data type. A basic application of Arrays can be storing data in tabular format. For example, if we wish to store the contacts on our phone, then the software will simply place all our contacts in an array.

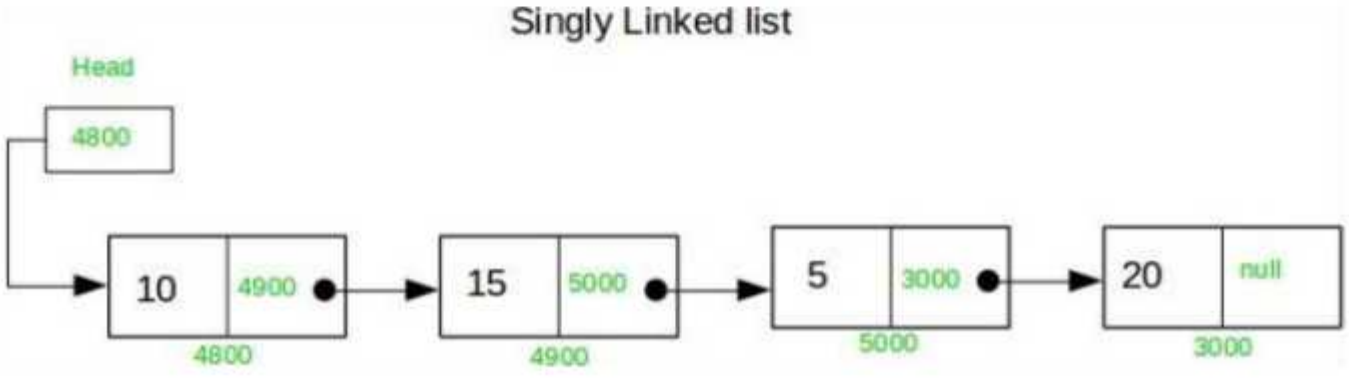


Some other applications of the arrays are:

1. Arrangement of leader-board of a game can be done simply through arrays to store the score and arrange them in descending order to clearly make out the rank of each player in the game.
2. A simple question Paper is an array of numbered questions with each of them assigned to some marks.
3. 2D arrays, commonly known as, matrix, are used in image processing.
4. It is also used in speech processing, in which each speech signal is an array.

Application of Linked Lists:

A linked list is a sequence data structure, which connects elements, called nodes, through links.

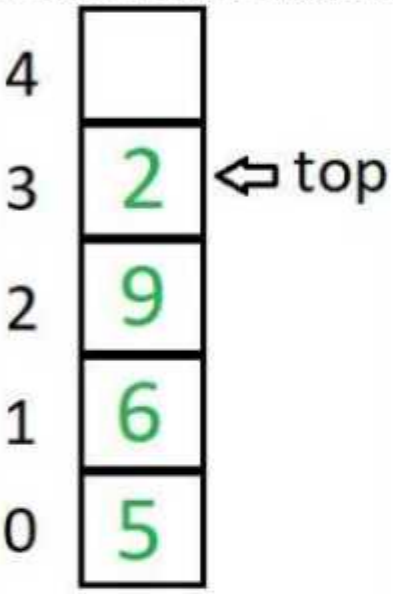


Some other applications of the linked list are:

1. Images are linked with each other. So, an image viewer software uses a linked list to view the previous and the next images using the previous and next buttons.
2. Web pages can be accessed using the previous and the next URL links which are linked using linked list.
3. The music players also use the same technique to switch between music.
4. To keep the track of turns in a multi player game, a circular linked list is used.

Application of Stack:

A stack is a data structure which uses LIFO order.



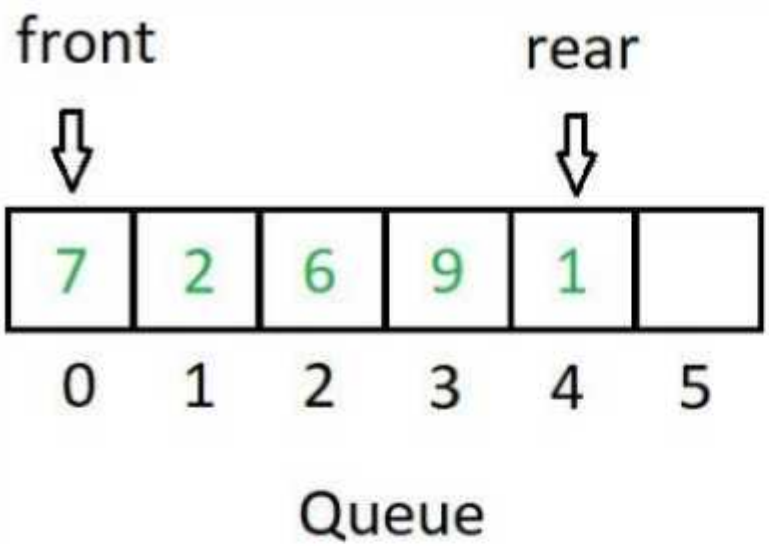
Stack

Some Applications of a stack are:

1. Converting infix to postfix expressions.
2. Undo operation is also carried out through stacks.
3. Syntaxes in languages are parsed using stacks.
4. It is used in many virtual machines like JVM.

Application of Queu

e: A queue is a data structure which uses FIFO order.

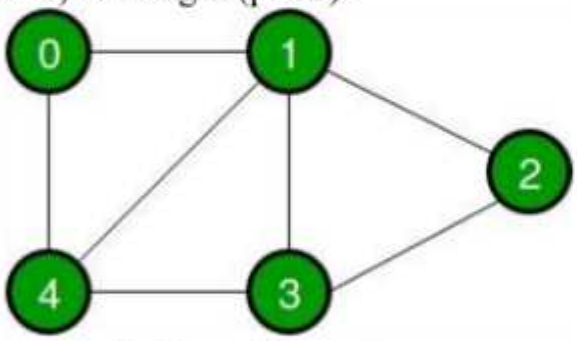


Some applications of a queue are:

1. Operating System uses queue for job scheduling.
2. To handle congestion in networking queue can be used.
3. Data packets in communication are arranged in queue format.

Application of Graph:

Graph is a data structure where data is stored in a collection of interconnected vertices (nodes) and edges (paths).

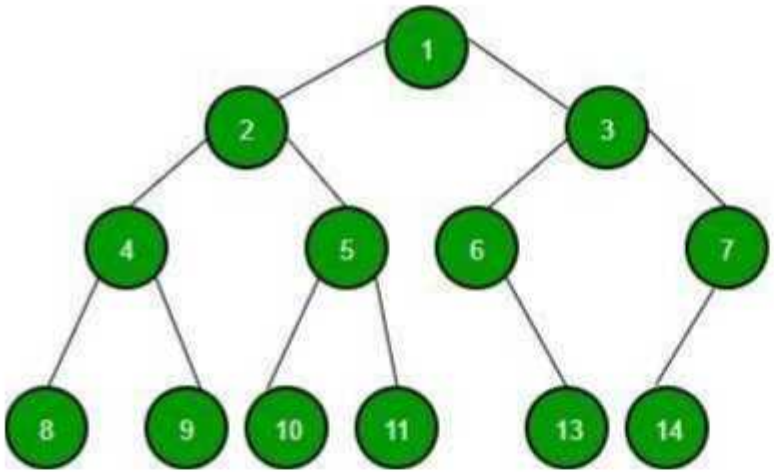


Some applications of a graph are:

1. Facebook's Graph API uses the structure of Graphs.
2. Google's Knowledge Graph also has to do something with Graph.
3. Dijkstra algorithm or the shortest path first algorithm also uses graph structure to finding the smallest path between the nodes of the graph.
4. GPS navigation system also uses shortest path APIs.

Application of Tree:

Trees are hierarchical structures having a single root node.



Some applications of the trees are:

1. XML Parser uses tree algorithms.
2. Decision-based algorithm is used in machine learning which works upon the algorithm of tree.
3. Databases also uses tree data structures for indexing.
4. Domain Name Server(DNS) also uses tree structures.

Application of Hash Tables:

Hash Tables are store data in key-value pairs. It only stores data which has a key associated with it. Inserting and Searching operations are easily manageable while using Hash Tables.

Some applications of a hashtable are:

1. Data stored in databases is generally of the key-value format which is done through hash tables.
2. Every time we type something to be searched in google chrome or other browsers, it generates the desired output based on the principle of hashing.
3. Message Digest, a function of cryptography also uses hashing for creating output in such a manner that reaching to the original input from that generated output is almost next to impossible.
4. In our computers we have various files stored in it, each file has two very crucial information that is, filename and file path, in order to make a connection between the filename to its corresponding file path hash tables are used.

Application of Heap:

A Heap is a special case of a binary tree where the parent nodes are compared to their children with their values and are arranged accordingly. Some applications of heaps are:

1. In heapsort Algorithm, which is an algorithm for sorting elements in either min heap(the key of the parent is less than or equal to those of its children) or max heap(the key of the parent is greater than or equal to those of its children), sorting is done with the creation of heaps.
2. Heaps are used to implement a priority queue where priority is based on the order of heap created.
3. Systems concerned with security and embedded system such as Linux Kernel uses Heap Sort because of the O(n log(n)).
4. If we are stuck in finding the K^{th}smallest (or largest) value of a number then heaps can solve the problem in an easy and fast manner.

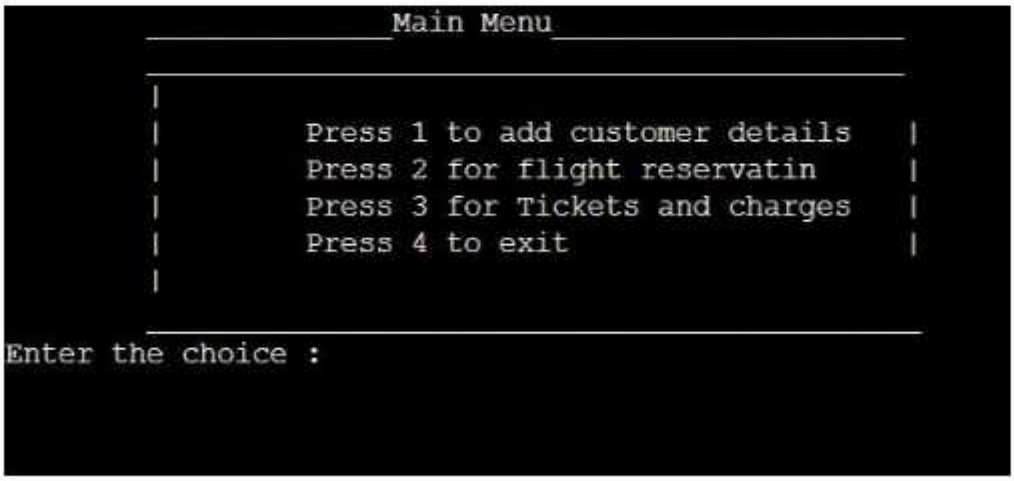
Chapter 9

Project undertaken:

Flight Management System

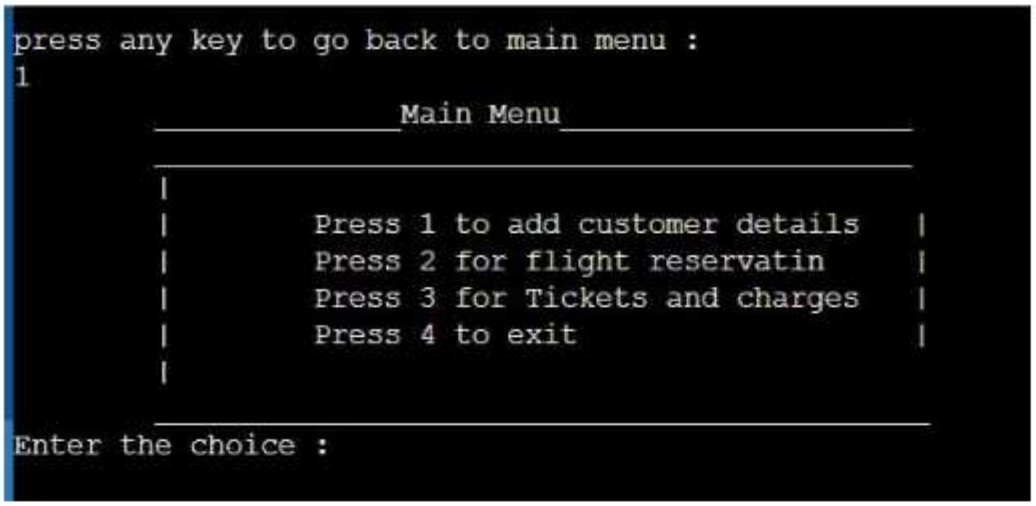
PROJECT SCREEN SHOTs

Starting Main Menu we can select the options by pressing the numbers corresponding to the relevant options.

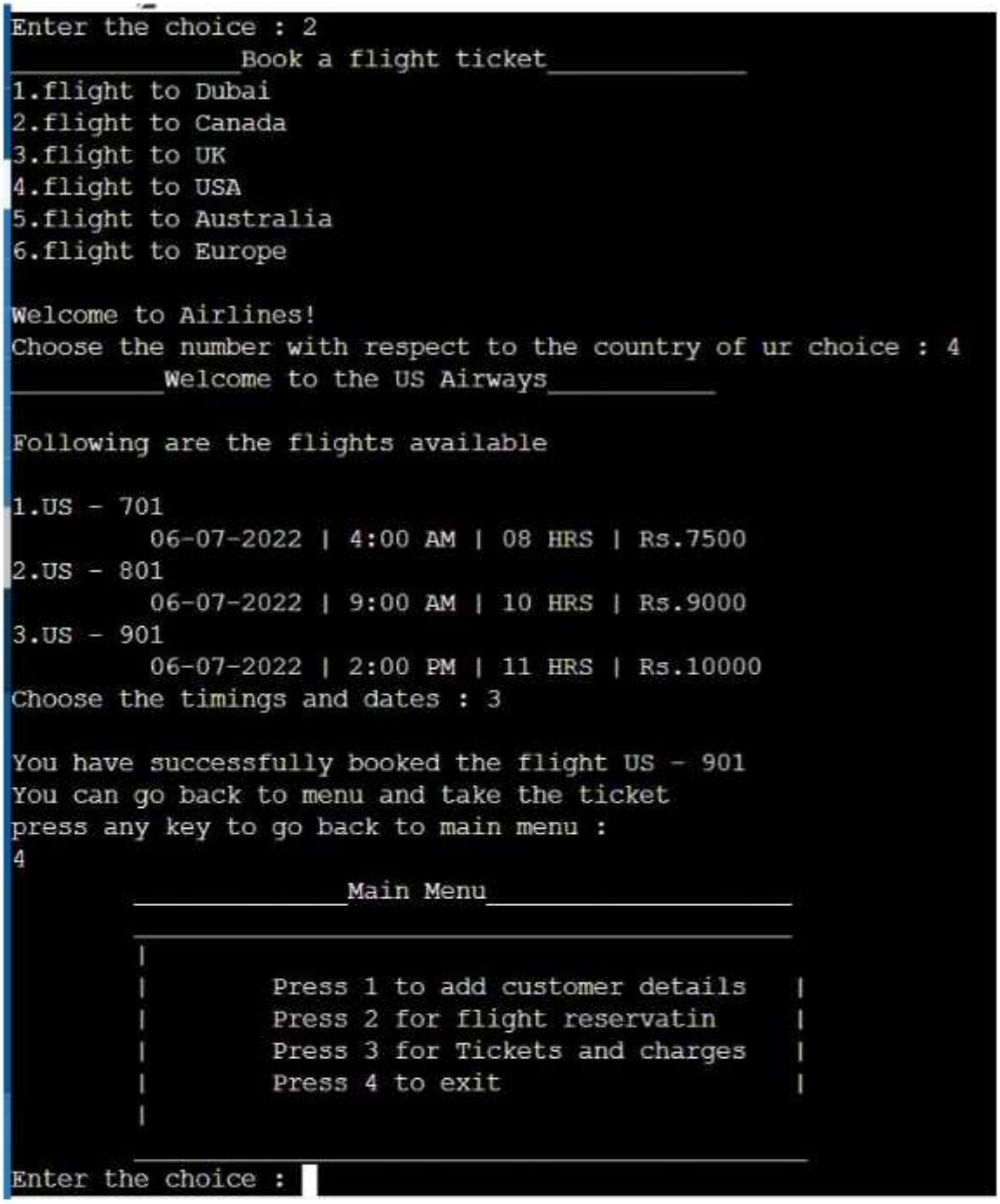


As I am a new User ,pressing 1 to enter user details after that Entered Id,Name,age,gender etc…And it is showing that details are saved and asking me to press any key to go back.

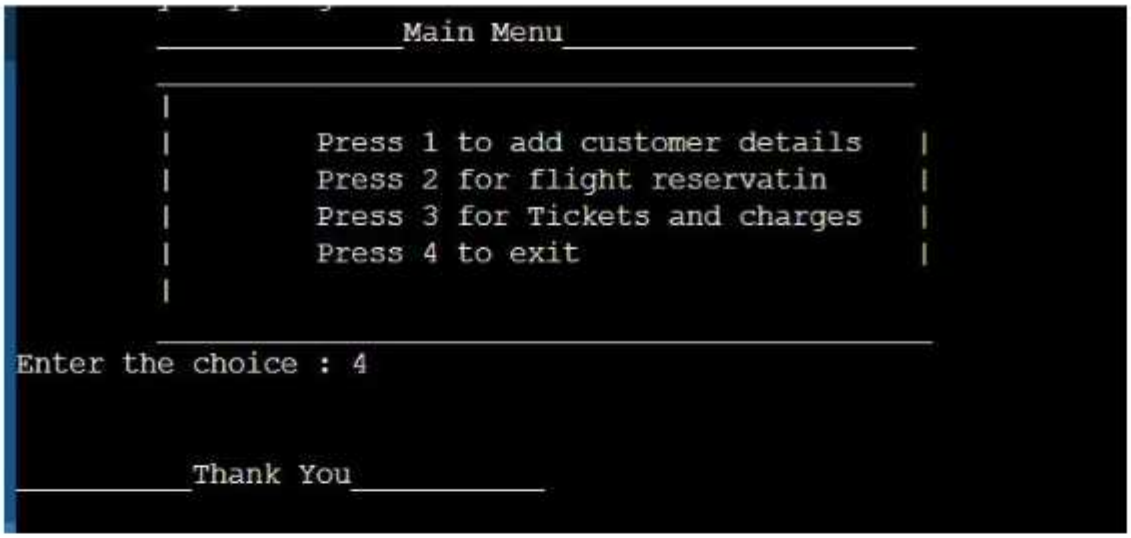
So, I pressed 1 as random number to go bac and it is back to Main Menu.



So, after that I am going to do my flight reservation by pressing 2, and it shows me the flight are travelling to the specific country and asking me to choose the country I want to travel. So, I pressed 4 which is USA and it shows me the flights available. And I selected one of the flight. So, it shows me that booking is successful……check below pic.



Finally I pressed 4 to exit the MainMenu Coz I completed my Booking .



So, Here is my "records.txt" Where I am storing user Details.

main.cpp records.txt

```
1   _____TATA Airlines_____
2   _____Tickets_____
3   ____________________________
4   Customer Id: 101
5   Customer Name: JnaneswarRao
6   Customer Gender: male
7       Description
8
9   Destinatinon: USA
10  Flight cost: 10000
11
```

Sorce code:

```
#include<bits/stdc++.h>

using namespace std;
void mainMenu();
class Management
{
 public:
        Management()
        {
                mainMenu();
        }
};
class Details
{
 public:
        static string name,gender;
        int phoneNo;
        int age;
        string add;
        static int cId;
        char arr[100];
        void information()
        {
                cout<<"\nEnter the customer ID : ";
                cin>>cId;
                cout<<"\nEnter the name : ";
                cin>>name;
                cout<<"\nEnter the age : ";
                cin>>age;
                cout<<"\nAdress : ";
                cin>>add;
                cout<<"\nGender : ";
                cin>>gender;
                cout<<"Your details are saved with us\n"<<endl;

        }
};

int Details::cId;
string Details::name;
string Details::gender;

class Registration
{
 public:
 static int choice;
 int choice1;
 int back;
 static float charges;
```

```
void flights()
{
	string flightN[]={"Dubai","Canada","UK","USA","Australia","Europe"};
	for(int a=0;a<6;a++)
	{
		cout<<(a+1)<<".flight to "<<flightN[a]<<endl;
	}
	cout<<"\nWelcome to Airlines!"<<endl;
	cout<<"Choose the number with respect to the country of ur choice : ";
	cin>>choice;
	switch(choice)
	{
		case 1:
			{
				cout<<"_________Welcome to the Dubai Emirates__________\n"<<endl;
				cout<<"Following are the flights available\n"<<endl;
				cout<<"1.DUB - 498"<<endl;
				cout<<"\t 06-07-2022 | 4:00 AM | 08 HRS | Rs.7500"<<endl;
				cout<<"2.DUB - 602"<<endl;
				cout<<"\t 06-07-2022 | 9:00 AM | 10 HRS | Rs.9000"<<endl;
				cout<<"3.DUB - 204"<<endl;
				cout<<"\t 06-07-2022 | 2:00 PM | 11 HRS | Rs.10000"<<endl;
				cout<<"Choose the timings and dates : ";
				cin>>choice1;
				if(choice1==1)
				{
					charges=7500;
					cout<<"\nYou have successfully booked the flight DUB - 498"<<endl;
					cout<<"You can go back to menu and take the ticket"<<endl;
				}
				else if(choice1==2)
				{
					charges=9000;
					cout<<"\nYou have successfully booked the flight DUB - 602"<<endl;
					cout<<"You can go back to menu and take the ticket"<<endl;
				}
				else if(choice1==3)
				{
					charges=10000;
					cout<<"\nYou have successfully booked the flight DUB - 204"<<endl;
					cout<<"You can go back to menu and take the ticket"<<endl;
				}
				else
				{
					cout<<"Invalid input , shifting to the previous menu"<<endl;
					flights();
				}

				cout<<"press any key to go back to main menu : "<<endl;
				cin>>back;
				if(back==1)
				{
					mainMenu();
				}
```

```
                else
                {
                        mainMenu();
                }
                break;


        }


case 2:
        {
                cout<<"_________Welcome to the Canadian Airlines__________\n"<<endl;
                cout<<"Following are the flights available\n"<<endl;
                cout<<"1.CA - 101"<<endl;
                cout<<"\t 06-07-2022 | 4:00 AM | 08 HRS | Rs.7500"<<endl;
                cout<<"2.CA - 201"<<endl;
                cout<<"\t 06-07-2022 | 9:00 AM | 10 HRS | Rs.9000"<<endl;
                cout<<"3.CA - 301"<<endl;
                cout<<"\t 06-07-2022 | 2:00 PM | 11 HRS | Rs.10000"<<endl;
                cout<<"Choose the timings and dates : ";
                cin>>choice1;
                if(choice1==1)
                {
                        charges=7500;
                        cout<<"\nYou have successfully booked the flight CA - 101"<<endl;
                        cout<<"You can go back to menu and take the ticket"<<endl;
                }
                else if(choice1==2)
                {
                        charges=9000;
                        cout<<"\nYou have successfully booked the flight CA - 201"<<endl;
                        cout<<"You can go back to menu and take the ticket"<<endl;
                }
                else if(choice1==3)
                {
                        charges=10000;
                        cout<<"\nYou have successfully booked the flight CA - 301"<<endl;
                        cout<<"You can go back to menu and take the ticket"<<endl;
                }
                else
                {
                        cout<<"Invalid input , shifting to the previous menu"<<endl;
                        flights();
                }

                cout<<"press any key to go back to main menu : "<<endl;
                cin>>back;
                if(back==1)
                {
                        mainMenu();
                }
                else
                {
```

```
                mainMenu();
            }
            break;

        }

case 3:
        {
            cout<<"_________Welcome to the UK Airways_____________\n"<<endl;
            cout<<"Following are the flights available\n"<<endl;
            cout<<"1.UK - 401"<<endl;
            cout<<"\t 06-07-2022 | 4:00 AM | 08 HRS | Rs.7500"<<endl;
            cout<<"2.UK - 501"<<endl;
            cout<<"\t 06-07-2022 | 9:00 AM | 10 HRS | Rs.9000"<<endl;
            cout<<"3.UK - 601"<<endl;
            cout<<"\t 06-07-2022 | 2:00 PM | 11 HRS | Rs.10000"<<endl;
            cout<<"Choose the timings and dates : ";
            cin>>choice1;
            if(choice1==1)
            {
                charges=7500;
                cout<<"\nYou have successfully booked the flight UK - 401"<<endl;
                cout<<"You can go back to menu and take the ticket"<<endl;
            }
            else if(choice1==2)
            {
                charges=9000;
                cout<<"\nYou have successfully booked the flight UK - 501"<<endl;
                cout<<"You can go back to menu and take the ticket"<<endl;
            }
            else if(choice1==3)
            {
                charges=10000;
                cout<<"\nYou have successfully booked the flight UK - 601"<<endl;
                cout<<"You can go back to menu and take the ticket"<<endl;
            }
            else
            {
                cout<<"Invalid input , shifting to the previous menu"<<endl;
                flights();
            }

            cout<<"press any key to go back to main menu : "<<endl;
            cin>>back;
            if(back==1)
            {
                mainMenu();
            }
            else
            {
                mainMenu();
            }
            break;
```

```
        }
case 4:
        {
                cout<<"_________Welcome to the US Airways_________\n"<<endl;
                cout<<"Following are the flights available\n"<<endl;
                cout<<"1.US - 701"<<endl;
                cout<<"\t 06-07-2022 | 4:00 AM | 08 HRS | Rs.7500"<<endl;
                cout<<"2.US - 801"<<endl;
                cout<<"\t 06-07-2022 | 9:00 AM | 10 HRS | Rs.9000"<<endl;
                cout<<"3.US - 901"<<endl;
                cout<<"\t 06-07-2022 | 2:00 PM | 11 HRS | Rs.10000"<<endl;
                cout<<"Choose the timings and dates : ";
                cin>>choice1;
                if(choice1==1)
                {
                        charges=7500;
                        cout<<"\nYou have successfully booked the flight US - 701"<<endl;
                        cout<<"You can go back to menu and take the ticket"<<endl;
                }
                else if(choice1==2)
                {
                        charges=9000;
                        cout<<"\nYou have successfully booked the flight US - 801"<<endl;
                        cout<<"You can go back to menu and take the ticket"<<endl;
                }
                else if(choice1==3)
                {
                        charges=10000;
                        cout<<"\nYou have successfully booked the flight US - 901"<<endl;
                        cout<<"You can go back to menu and take the ticket"<<endl;
                }
                else
                {
                        cout<<"Invalid input , shifting to the previous menu"<<endl;
                        flights();
                }

                cout<<"press any key to go back to main menu : "<<endl;
                cin>>back;
                if(back==1)
                {
                        mainMenu();
                }
                else
                {
                        mainMenu();
                }
                break;

        }

case 5:
        {
                cout<<"_________Welcome to the Australia Airways_________\n"<<endl;
```

```
                cout<<"Following are the flights available\n"<<endl;
                cout<<"1.UK - 1001"<<endl;
                cout<<"\t 06-07-2022 | 4:00 AM | 08 HRS | Rs.7500"<<endl;
                cout<<"2.CA - 2001"<<endl;
                cout<<"\t 06-07-2022 | 9:00 AM | 10 HRS | Rs.9000"<<endl;
                cout<<"3.CA - 3001"<<endl;
                cout<<"\t 06-07-2022 | 2:00 PM | 11 HRS | Rs.10000"<<endl;
                cout<<"Choose the timings and dates : ";
                cin>>choice1;
                if(choice1==1)
                {
                        charges=7500;
                        cout<<"\nYou have successfully booked the flight Australia - 1001"<<endl;
                        cout<<"You can go back to menu and take the ticket"<<endl;
                }
                else if(choice1==2)
                {
                        charges=9000;
                        cout<<"\nYou have successfully booked the flight Australia - 2001"<<endl;
                        cout<<"You can go back to menu and take the ticket"<<endl;
                }
                else if(choice1==3)
                {
                        charges=10000;
                        cout<<"\nYou have successfully booked the flight Australia - 3001"<<endl;
                        cout<<"You can go back to menu and take the ticket"<<endl;
                }
                else
                {
                        cout<<"Invalid input , shifting to the previous menu"<<endl;
                        flights();
                }

                cout<<"press any key to go back to main menu : "<<endl;
                cin>>back;
                if(back==1)
                {
                        mainMenu();
                }
                else
                {
                        mainMenu();
                }

                break;
        }

case 6:
        {
                cout<<"_________Welcome to the Europian Airways__________\n"<<endl;
                cout<<"Following are the flights available\n"<<endl;
                cout<<"1.EU - 4001"<<endl;
                cout<<"\t 06-07-2022 | 4:00 AM | 08 HRS | Rs.7500"<<endl;
                cout<<"2.EU - 5001"<<endl;
```

```
                cout<<"\t 06-07-2022 | 9:00 AM | 10 HRS | Rs.9000"<<endl;
                cout<<"3.EU - 6001"<<endl;
                cout<<"\t 06-07-2022 | 2:00 PM | 11 HRS | Rs.10000"<<endl;
                cout<<"Choose the timings and dates : ";
                cin>>choice1;
                if(choice1==1)
                {
                        charges=7500;
                        cout<<"\nYou have successfully booked the flight EU - 4001"<<endl;
                        cout<<"You can go back to menu and take the ticket"<<endl;
                }
                else if(choice1==2)
                {
                        charges=9000;
                        cout<<"\nYou have successfully booked the flight EU - 5001"<<endl;
                        cout<<"You can go back to menu and take the ticket"<<endl;
                }
                else if(choice1==3)
                {
                        charges=10000;
                        cout<<"\nYou have successfully booked the flight EU - 6001"<<endl;
                        cout<<"You can go back to menu and take the ticket"<<endl;
                }
                else
                {
                        cout<<"Invalid input , shifting to the previous menu"<<endl;
                        flights();
                }

                cout<<"press any key to go back to main menu : "<<endl;
                cin>>back;
                if(back==1)
                {
                        mainMenu();
                }
                else
                {
                        mainMenu();
                }
                break;

        }
    default :
        {
                cout<<"Invalid Input , shifting to main Menu !"<<endl;
                mainMenu();
                break;
        }

    }

}
};
```

```
float Registration::charges;
int Registration::choice;

class Ticket : public Registration,public Details
{
 public:

        void bill()
        {
                string destination="";
                ofstream outf("records.txt");
                {
                        outf<<"_________TATA Airlines_____________"<<endl;
                        outf<<"_________Tickets_________________"<<endl;
                        outf<<"_____________________________________"<<endl;

                        outf<<"Customer Id: "<<Details::cId<<endl;
                        outf<<"Customer Name: "<<Details::name<<endl;
                        outf<<"Customer Gender: "<<Details::gender<<endl;
                        outf<<"\tDescription"<<endl<<endl;

                        if(Registration::choice==1)
                        {
                                destination="Dubai";
                        }
                        else if(Registration::choice==2)
                        {
                                destination="Canada";
                        }
                        else if(Registration::choice==3)
                        {
                                destination="UK";
                        }
                        else if(Registration::choice==4)
                        {
                                destination="USA";
                        }
                        else if(Registration::choice==5)
                        {
                                destination="Australia";
                        }
                        else if(Registration::choice==6)
                        {
                                destination="Europe";
                        }
                        outf<<"Destinatinon: "<<destination<<endl;
                        outf<<"Flight cost: "<<Registration::charges<<endl;

                }
                outf.close();
        }
        void display()
        {
                ifstream ifs("records.txt");
```

```
		{
			if(ifs)
			{
				while(!ifs.eof())
				{
					ifs.getline(arr,100);
					cout<<arr<<endl;
				}
			}
			else if(!ifs)
			{
				cout<<"File error"<<endl;
			}

		}
		ifs.close();
	}
};

void mainMenu()
{
 int lchoice;
 int schoice;
 int back;
 cout<<"\t_____Main Menu_____"<<endl;
 cout<<"\t_____"<<endl;
 cout<<"\t|\t\t\t\t\t\t"<<endl;

 cout<<"\t|\t Press 1 to add customer details   |"<<endl;
 cout<<"\t|\t Press 2 for flight reservatin     |"<<endl;
 cout<<"\t|\t Press 3 for Tickets and charges   |"<<endl;
 cout<<"\t|\t Press 4 to exit                   |"<<endl;
 cout<<"\t|\t\t\t\t\t\t"<<endl;
 cout<<"\t_____"<<endl;

 cout<<"Enter the choice : ";
 cin>>lchoice;
 Details d;
 Registration r;
 Ticket t;
 switch(lchoice)
 {
	case 1:
	{
		cout<<"_____Customers_____"<<endl;
		d.information();
		cout<<"press any key to go back to main menu : "<<endl;
		cin>>back;
		if(back==1)
		{
			mainMenu();
		}
		else
		{
```

```
                mainMenu();
            }
            break;
        }
        case 2:
        {
            cout<<"_____Book a flight ticket_____"<<endl;
            r.flights();
            break;
        }
        case 3:
        {
            cout<<"_____Get your ticket_____"<<endl;
            t.bill();
            cout<<"press 1 to show your ticket : ";
            cin>>back;
            if(back==1)
            {
                t.display();
                cout<<"Press any key to go back : ";
                cin>>back;
                if(back==1)
                {
                    mainMenu();
                }
                else
                {
                    mainMenu();
                }
            }
            else
            {
                mainMenu();
            }
            break;
        }
        case 4:
        {
            cout<<"\n\n_____Thank You_____"<<endl;
            break;
        }
        default:
        {
            cout<<"Invalid Input , please try again !!\n"<<endl;
            mainMenu();
            break;
        }

 }
}

int main()
{
 Management obj;
```

```
    return 0;
}
```

Learning Outcome from training/technology learnt

- Basics of data structures
- Analysis of algorithms
- Arrays
- Strings/queue/stack
- Sorting
- Insertion
- Graphs
- Trees
- Heap
- Hashing
- Time complexities
- Space complexities

Conclusion:

As you've seen, data structures are the essential building blocks that we use to organize all of our digital information. Choosing the right data structure allows us to use the algorithms we want and keeps our code running smoothly. Understanding data structures and how to use them well can play a vital role in many situations including:

1. Technical interviews in which you may be asked to evaluate and determine runtime for data structures given specific algorithms
2. Day-to-day work for many software engineers who manipulate data stored in structures.
3. Data science work where data is stored and accessed

References:

https://www.geeksforgeeks.org https://www.en-wikipedia.com https://www.stackoverflow.com