DATA STRUCTURE

CSE 213

Prepared for
Richard Philip
City University

Prepared by
khairul Islam
ID:171442608

November 05, 2018

Table of Contents

1. INTRODUCTION TO DATA STRUCTURE :

Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way. so that various operations can be performed on it easily. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have some data which has, player's name "Virat" and age 26. Here "Virat" is of String data type and 26 is of integer data type.

We can organize this data as a record like Player record, which will have both player's name and age in it. Now we can collect and store player's records in a file or database as a data structure. For example: "Dhoni" 30, "Gambhir" 31, "Sehwag" 33.

If you are aware of Object Oriented programming concepts, then a class also does the same thing, it collects different type of data under one single entity. The only difference being, data structures provides for techniques to access and manipulate data efficiently.

In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily. It represents the knowledge of data to be organized in memory. It should be designed and implemented in such a way that it reduces the complexity and increases the efficiency.

2. BASIC TYPES OF DATA STRUCTURES

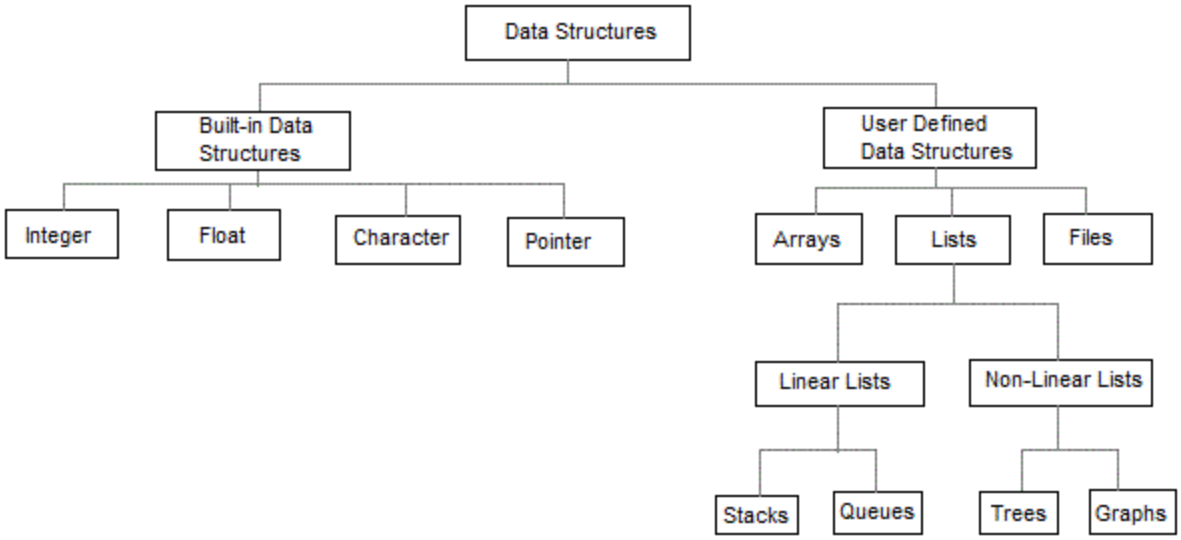
As we have discussed above, anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as **Primitive Data Structures**.

Then we also have some complex Data Structures, which are used to store large and connected data. Some example of **Abstract Data Structure** are :

- Linked List
- Tree

- Graph
- Stack, Queue
- etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required. We will look into these data structures in more details in our later lessons.



INTRODUCTION TO DATA STRUCTURES

The data structures can also be classified on the basis of the following characteristics:

Characterstic	**Description**
Linear	In Linear data structures,the data items are arranged in a linear sequence. Example: **Array**

Non-Linear	In Non-Linear data structures,the data items are not in sequence. Example: **Tree**, **Graph**
Homogeneous	In homogeneous data structures,all the elements are of same type. Example: **Array**
Non-Homogeneous	In Non-Homogeneous data structure, the elements may or may not be of the same type. Example: **Structures**
Static	Static data structures are those whose sizes and structures associated memory locations are fixed, at compile time. Example: **Array**
Dynamic	Dynamic structures are those which expands or shrinks depending upon the program need and its execution. Also, their associated memory locations changes. Example: **Linked List created using pointers**

Here, we will discuss them in two parts.

1.Midterm

2.Final

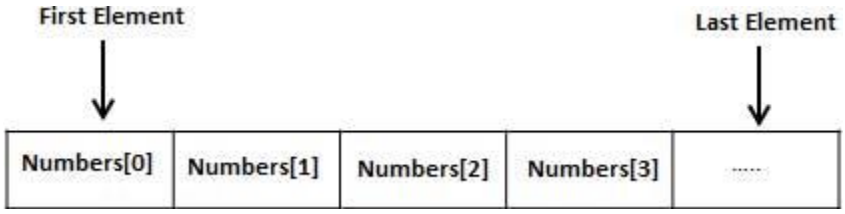
3. Midterm

3.1 ARRAY

3.1.1 One-dimensional array

1D Array is a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



In our course, we have learnt what is array, how to store data of array, how to print element of array. There we have learnt about 1D array.

3.1.2 Two-dimensional array

In C programming, we can create an array of arrays known as multidimensional array. For example,

float x[3][4];

Here, x is a two-dimensional (2d) array. The array can hold 12 elements. You can think the array as table with 3 row and each row has 4 column.

Similarly, we can declare a three-dimensional (3d) array. For example,

float y[2][4][3];

Here, The array y can hold 24 elements.

You can think this example as: Each 2 elements have 4 elements, which makes 8 elements and each 8 elements can have 3 elements. Hence, the total number of elements is 24.

How to initialize a multidimensional array?

There is more than one way to initialize a multidimensional array.

Initialization of a two dimensional array

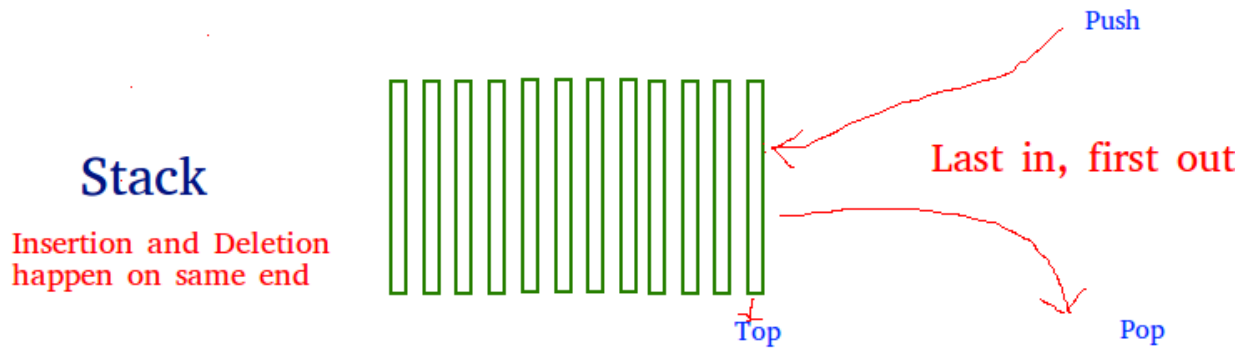
// Different ways to initialize two dimensional array

```
int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[2][3] = {1, 3, 0, -1, 5, 9};
```

Above code are three different ways to initialize a two dimensional arrays.

3.2 STACK

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).

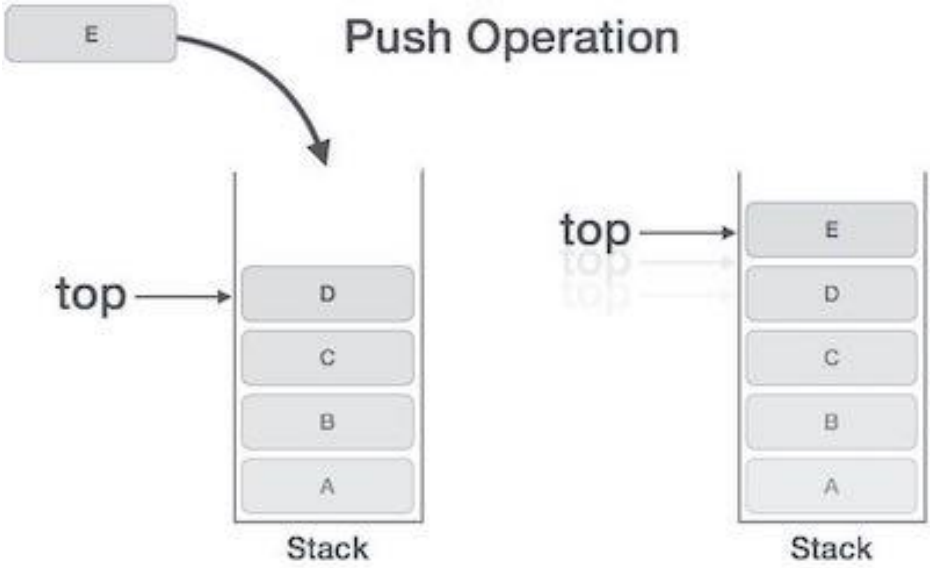


There are many real-life examples of a stack. Consider an example of plates stacked over one another in the canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost

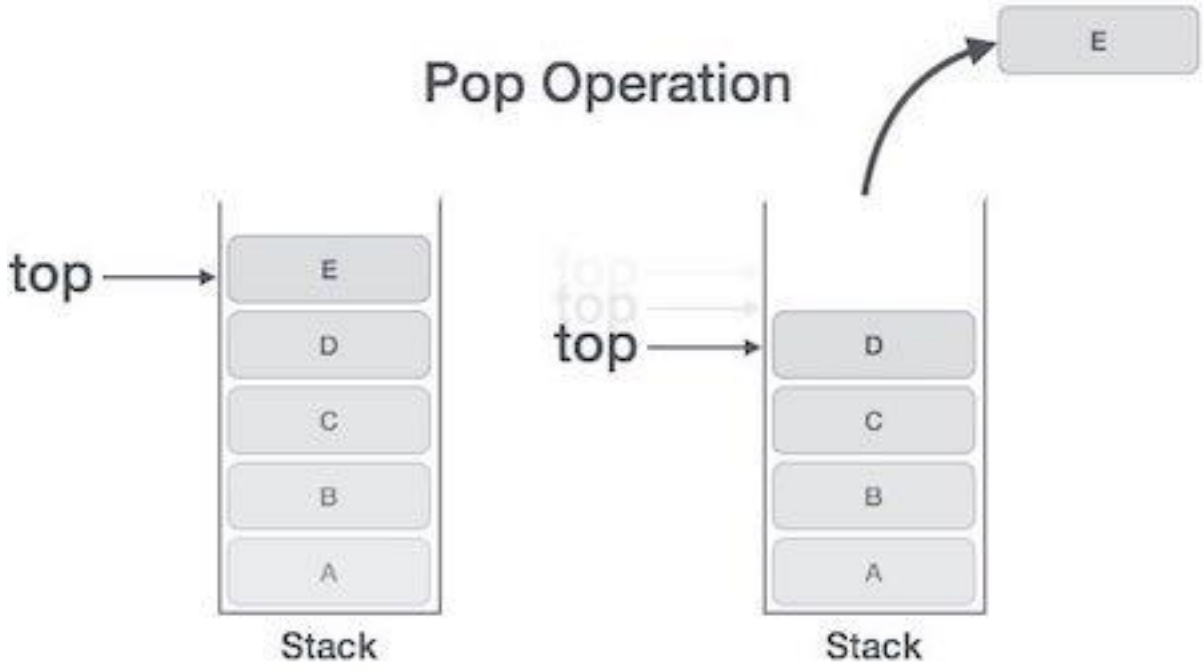
position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO(Last In First Out)/FILO(First In Last Out) order.

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 stack. Consider the simple example of plates stacked over one another in 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 bottommost 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:
push(), pop(), isEmpty() and peek() all take O(1) time. We do not run any loop in any of these operations.

Applications of stack:

Balancing of symbols

Infix to Postfix /Prefix conversion

Redo-undo features at many places like editors, photoshop.

Forward and backward feature in web browsers

Used in many algorithms like Tower of Hanoi, tree traversals, stock span problem, histogram problem.

Other applications can be Backtracking, Knight tour problem, rat in a maze, N queen problem and sudoku solver

In Graph Algorithms like Topological Sorting and Strongly Connected Components

Implementation:

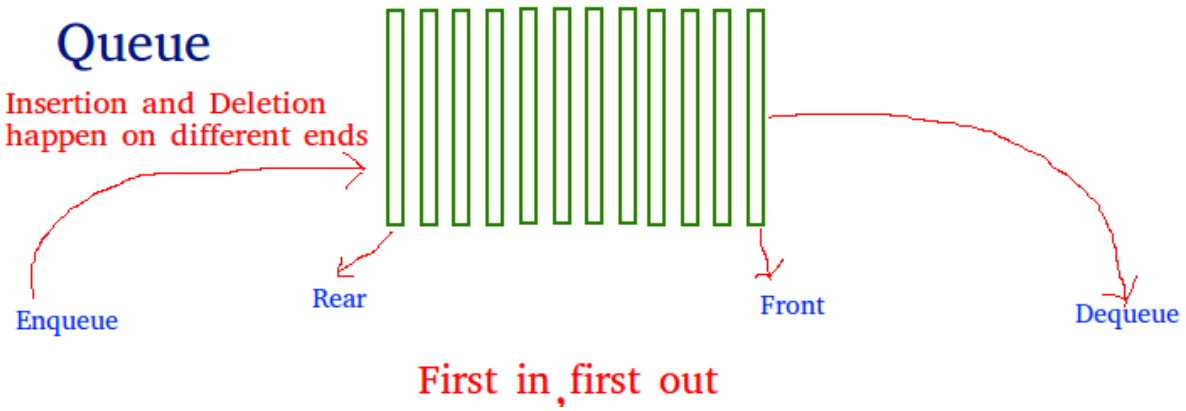
There are two ways to implement a stack:

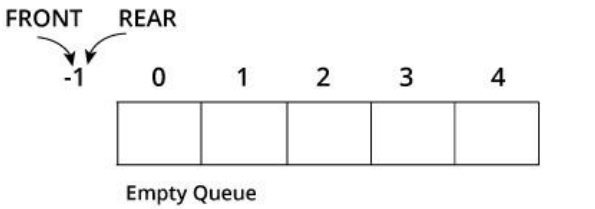
1.Using array

2.Using linked list

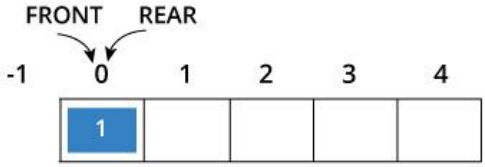
3.3 QUEUE

A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a 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 item the most recently added; in a queue, we remove the item the least recently added.

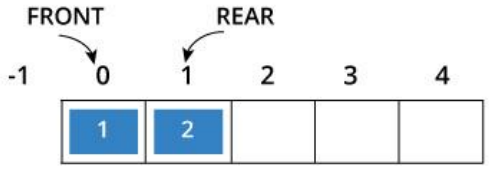




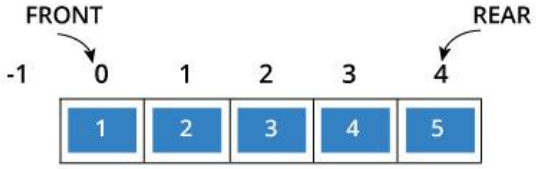
Empty Queue



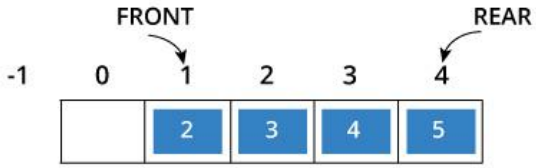
EnQueue first element



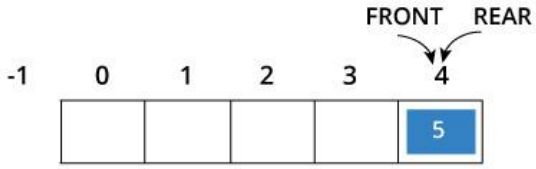
EnQueue



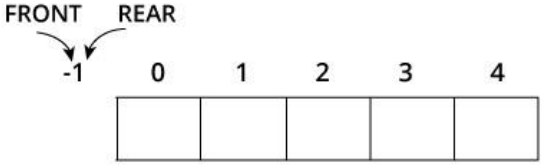
EnQueue



DeQueue



DeQueue last element



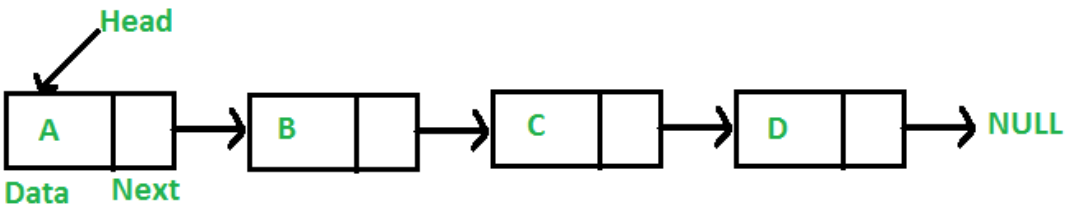
Empty Queue

In queue also, there have several function by which we can do the operation of queue. They are:

- enQueue
- deQueue
- Display_queue
- Is_full
- Is_empty and etc.

3.4 LINKED LIST

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

Why Linked List?

Arrays can be used to store linear data of similar types, but arrays have following limitations.

1) 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.

2) Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to shifted.

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.

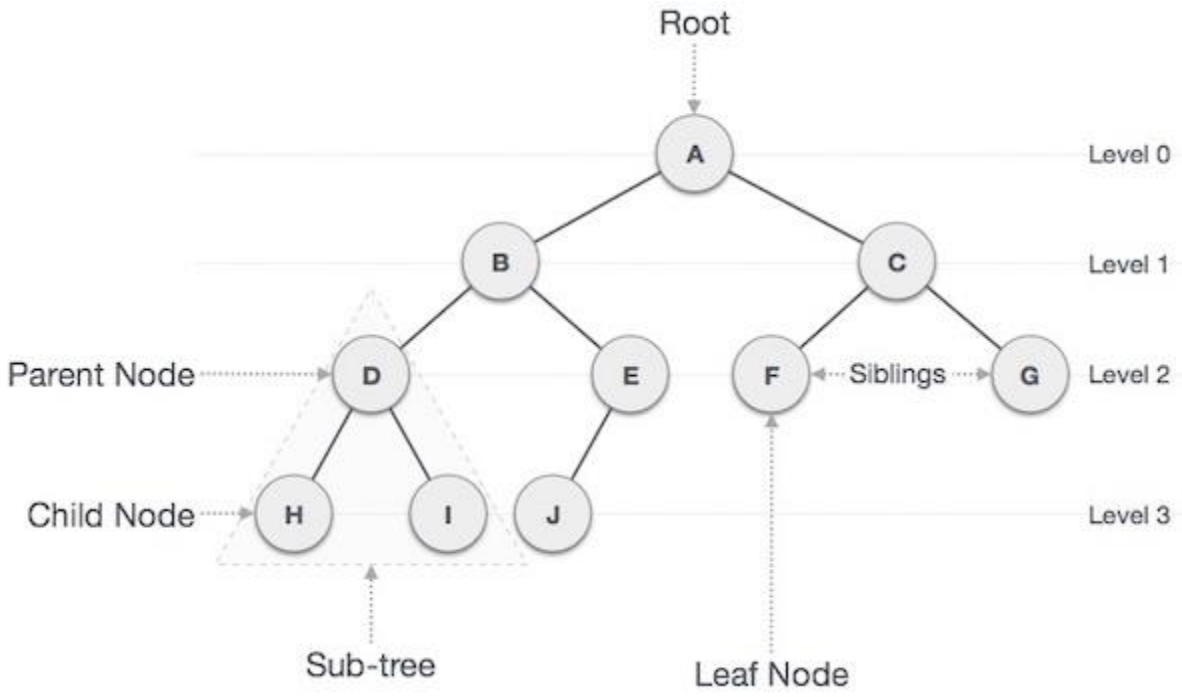
Advantages over arrays
1) Dynamic size
2) Ease of insertion/deletion

4. FINAL

4.1 TREE

Tree represents the nodes connected by edges. We will discuss binary tree or binary search tree specifically.

Binary Tree is a special data structure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.



4.1.1 Important Terms

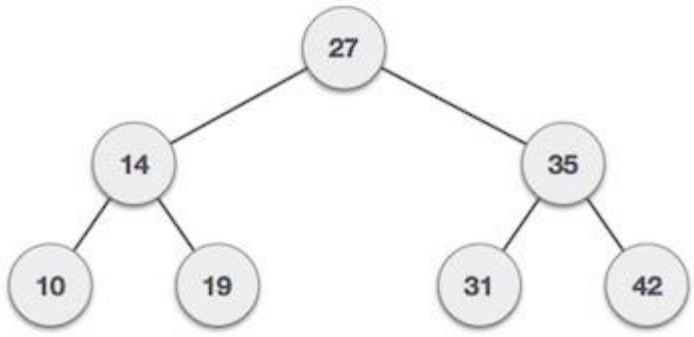
Following are the important terms with respect to tree.

- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** − Any node except the root node has one edge upward to a node called parent.
- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.
- **Subtree** − Subtree represents the descendants of a node.
- **Visiting** − Visiting refers to checking the value of a node when control is on the node.
- **Traversing** − Traversing means passing through nodes in a specific order.

- **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

4.1.2 Binary Search Tree Representation

Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.



We're going to implement tree using node object and connecting them through references.

4.1.3 Tree Node

The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

```
struct node {
  int data;
  struct node *leftChild;
  struct node *rightChild;
};
```

In a tree, all nodes share common construct.

4.1.4 BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following −

- **Insert** − Inserts an element in a tree/create a tree.
- **Search** − Searches an element in a tree.
- **Preorder Traversal** − Traverses a tree in a pre-order manner.
- **Inorder Traversal** − Traverses a tree in an in-order manner.
- **Postorder Traversal** − Traverses a tree in a post-order manner.

We shall learn creating (inserting into) a tree structure and searching a data item in a tree in this chapter. We shall learn about tree traversing methods in the coming chapter.

4.1.5 Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

4.2 SORTING

Sorting is a process of ordering or placing a list of elements from a collection in some kind of order. It is nothing but storage of data in sorted order. Sorting can be done in ascending and descending order. It arranges the data in a sequence which makes searching easier.

For example, suppose we have a record of employee. It has following data:

Employee No.
Employee Name
Employee Salary
Department Name

Here, employee no. can be takes as key for sorting the records in ascending or

descending order. Now, we have to search a Employee with employee no. 116, so we don't require to search the complete record, simply we can search between the Employees with employee no. 100 to 120.

4.2 1 Sorting Techniques

Sorting technique depends on the situation. It depends on two parameters.

1. Execution time of program that means time taken for execution of program.
2. Space that means space taken by the program.

Sorting techniques are differentiated by their efficiency and space requirements.

Sorting can be performed using several techniques or methods, as follows:

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5. Heap Sort

4.2.2 Bubble Sort

- Bubble sort is a type of sorting.
- It is used for sorting 'n' (number of items) elements.
- It compares all the elements one by one and sorts them based on their values.

4. 2.3 Insertion Sort

- Insertion sort is a simple sorting algorithm.
- This sorting method sorts the array by shifting elements one by one.
- It builds the final sorted array one item at a time.
- Insertion sort has one of the simplest implementation.
- This sort is efficient for smaller data sets but it is insufficient for larger lists.
- It has less space complexity like bubble sort.

- It requires single additional memory space.
- Insertion sort does not change the relative order of elements with equal keys because it is stable.

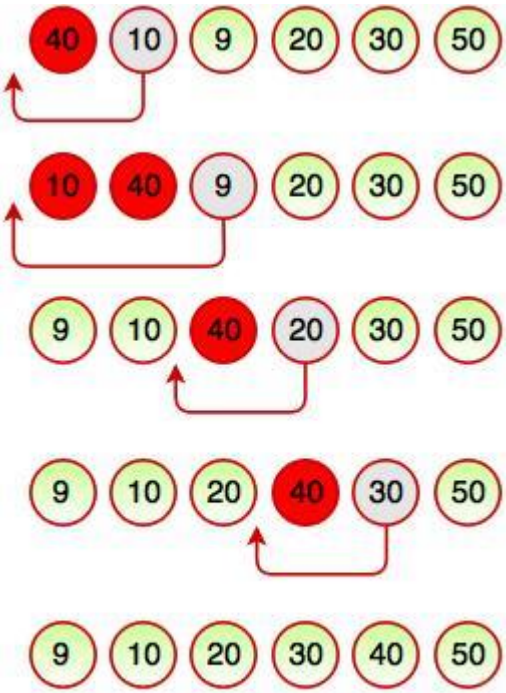


Fig. Working of Insertion Sort

- The above diagram represents how insertion sort works. Insertion sort works like the way we sort playing cards in our hands. It always starts with the second element as key. The key is compared with the elements ahead of it and is put it in the right place.
- In the above figure, 40 has nothing before it. Element 10 is compared to 40 and is inserted before 40. Element 9 is smaller than 40 and 10, so it is inserted before 10 and this operation continues until the array is sorted in ascending order.

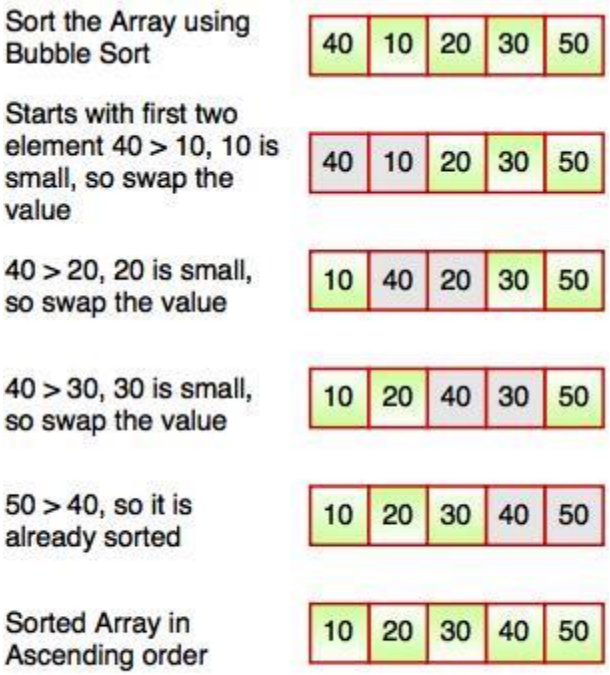


Fig. Working of Bubble Sort

- The above diagram represents how bubble sort actually works. This sort takes $O(n^2)$ time. It starts with the first two elements and sorts them in ascending order.
- Bubble sort starts with first two elements. It compares the element to check which one is greater.
- In the above diagram, element 40 is greater than 10, so these values must be swapped. This operation continues until the array is sorted in ascending order.

With All of these topics ,we have learnt max heap, min heap and some other topics. There have time complexity in our lesson. Our honorable teacher discussed about BFS and DFS also. All of these were in our course outline and we have successfully completed them.

References

1.Array. https://www.tutorialspoint.com/cprogramming/c_arrays.htm

2.Stack. https://www.geeksforgeeks.org/stack-data-structure/

3.Queue. https://www.geeksforgeeks.org/queue-data-structure/

4.Linked list. https://www.geeksforgeeks.org/data-structures/linked-list/

5. Tree. https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm

6. https://www.studytonight.com/data-structures/introduction-to-data-structures