A REPORT OF FOUR WEEK TRAINING

at

GURU NANAK DEV ENGINEERING COLLEGE

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE AWARD OF THE DEGREE OF

BACHELOR OF TECHNOLOGY

(Information Technology)

JULY-AUGUST ,2022

SUBMITTED BY:

Shivay Bhandari

1905398

Department of Information Technology,

GURU NANAK DEV ENGINEERING COLLEGE LUDHIANA

(An Autonomous College Under UGC ACT)

GURU NANAK DEV ENGINEERING COLLEGE, LUDHIANA

Candidate's Declaration

I SHIVAY BHADNARI hereby declare that I have undertaken four week training "Guru Nanak Dev Engineering College" during a period from July to August in partial fulfillment of requirements for the award of degree of B.Tech (Information Technology) at GURU NANAK DEV ENGINEERING COLLEGE, LUDHIANA. The work which is being presented in the training report submitted to Department of Information Technology at GURU NANAK DEV ENGINEERING COLLEGE, LUDHIANA is an authentic record of training work.

Signature of the Student

The four week industrial training Viva–Voce Examination of Shivay Bhandari has been held on -01-2023 and accepted.

Signature of Internal Examiner   Signature of External Examiner

Abstract

The primary objective of data structures is used for organizing the data. It is also used for processing, retrieving, and storing data. During the training, I have been taught about competitive coding in which I have learned basic data structures and algorithms and also learned how to implement all these concepts with the help of programming language i.e. C++ or Python. Here I have revised about basic C++ programming concepts i.e. functions, pointers, OOPS, reference variables to implement the concept of data structure and algorithm. In data structure, I have learned about Array, Linked List, Stack, Queues and Hashing. In algorithm section, I have learned about analyse and design of algorithm, asymptotic notation, basic recursion, searching and sorting. All these concepts are backbone to develop any software and build a career in software development in IT sector.

ACKNOWLEDGEMENT

I am highly grateful to the Dr. Sehijpal Singh, Principal, Guru Nanak Dev Engineering College (GNDEC), Ludhiana, for providing this opportunity to carry out the minor project work at Space shooter application

The constant guidance and encouragement received from Dr. Kulvinder Singh Mann, H.O.D., IT Department, GNDEC Ludhiana has been of great help in carrying out the project work and is acknowledged with reverential thanks.

I would like to express a deep sense of gratitude and thanks profusely to Dr. Kamal Kaur, without her wise counsel and able guidance, it would have been impossible to complete the project in this manner.

I express gratitude to other faculty members of Information Technology Department of GNDEC for their intellectual support throughout the course of this work.

Finally, I am indebted to all whosoever have contributed in this report work.

Shivay Bhandari

List of Figures

Contents

1 INTRODUCTION

1.1 Introduction to organization

Guru Nanak Dev Engineering College, one of the prestigious, oldest and minority institution of Northern India, was established under the aegis of Nankana Sahib Education Trust (NSET) in 1956. NSET was founded in memory of the most sacred temple of Nankana Sahib, birth place of Guru Nanak Dev Ji. Shiromani Gurudwara Prabandhak Committee, Amritsar, a premier organization of universal brotherhood, which is the main force behind the mission of "Removal of Economic Backwardness through Technology" and NSET with the same mission, established a Polytechnic in 1953 and Guru Nanak Dev Engineering College (GNDEC) in 1956.

1.2 Introduction to Competitive Programming

1.2.1 Functions

A function refers to a group of statements that takes input, processes it, and returns an output. The idea behind a function is to combine common tasks that are done repeatedly. If you have different inputs, you will not write the same code again. You will simply call the function with a different set of data called parameters. Each JAVA program has at least one function, the main() function. You can divide your code into different functions. This division should be such that every function does a specific task.

Need of functions - There are numerous benefits associated with the use of functions. Each function puts related code together. This makes it easier for programmers to understand code.Functions make programming easier by eliminating code repetition.Functions facilitate code reuse. You can call the same function to perform a task at different sections of the program or even outside the program.

There are two basic types of function :

Built in function - These functions are part of the compiler package. These are part of standard library made available by the compiler. For example, exit(), sqrt(), pow(), length() etc. are library functions (built-in functions). Built-in functions helps you, to use function without declaring and defining it. To use built-in

functions in a program, you have to only include that header file where the required function is defined.

Example:

```
include<iostream.h>

include<string.h>

include<conio.h>

void main()


clrscr();

char str[80];

cout«"Enter any string (line):";cin.getline(str, 80);

int len = strlen(str);

cout«"of the string is: "«len;
```

```
getch();
```

User defined function -The user-defined functions are created by you i.e., the programmer. These functions are created as per requirements of your program.Here is an example, demonstrating user-defined functions in C++.

```
include<iostream.h>
include<conio.h>
void startmsg(void);
void main()

clrscr();
startmsg(); // function calling
int num;
cout«"Enter a number: ";
cin»num;
cout«"You entered: "«num;
getch();

void startmsg(void) // function definition

cout«"Welcome to gndecludhiana";
cout«"This is C++ Function Types Tutorial";
```

1.2.2 Pointers

A pointer is a variable that stores a memory address, for the purpose of acting as an alias to what is stored at that address. a pointer is a reference, but a reference is not necessarily a pointer. Pointers are a particular implementation of the concept of a reference, and the term tends to be used only for languages that gives direct access to the memory address.

Pointers are symbolic representations of addresses. They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures. Iterating over elements in arrays or other data structures is one of the main use of pointers. The address of the variable you're working with is assigned to the pointer variable that points to the same data type (such as an int or string).

Syntax

```
datatype *varname;
int *ptr; // ptr can point to an address which holds int data.
```

How to use a pointer?

• Define a pointer variable

• Assigning the address of a variable to a pointer using the unary operator () which returns the address of that variable.

• Accessing the value stored in the address using unary operator (*) which returns the value of the variable located at the address specified by its operand.

```
include <bits/stdc++.h>
usingnamespacestd;
voidgeeks()

intvar = 20;

// declare pointer variable
int* ptr;

// note that data type of ptr and var must be same
ptr = var;

// assign the address of a variable to a pointer
cout« "Value at ptr = "«ptr« "";
cout« "Value at var = "« var « "";
cout« "Value at *ptr = "« *ptr« "";

// Driver program
intmain()
```

```
geeks();
return0;
```

1.2.3 Object Oriented Programming (OOPS)

OOPs, or Object-oriented programming is an approach or a programming pattern where the programs are structured around objects rather than functions and logic. It makes the data partitioned into two memory areas, i.e., data and functions, and helps make the code flexible and modular. Object-oriented programming mainly focuses on objects that are required to be manipulated. In OOPs, it can represent data as objects that have attributes and functions. There are some basic concepts that act as the building blocks of OOPs.

- Classes Objects
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Object -An Object can be defined as an entity that has a state and behavior, or in other words, anything that exists physically in the world is called an object. It can represent a dog, a person, a table, etc.

Classes - Class can be defined as a blueprint of the object. It is basically a collection of objects which act as building blocks. A class contains data members (variables) and member functions. These member functions are used to manipulate the data members inside the class.

Abstraction - Abstraction helps in the data hiding process. It helps in displaying the essential features without showing the details or the functionality to the user. It avoids unnecessary information or irrelevant details and shows only that specific part which the user wants to see. Encapsulation -The wrapping up of data and functions together in a single unit is known as encapsulation. It can be achieved by making the data members' scope private and the member function's scope public to access these data members. Encapsulation makes the data non-accessible to the outside world.

Inheritance -Inheritance is the process in which two classes have an is-a relationship among each other and objects of one class acquire properties and features of the other class. The class which inherits the features is known as the child class, and the class whose features it inherited is called the parent class. For example, Class Vehicle is the parent class, and Class Bus, Car, and Bike are child classes.

Polymorphism -Polymorphism means many forms. It is the ability to take more than one form. It is a feature that provides a function or an operator with more than one definition. It can be implemented using function overloading, operator overload, function overriding, virtual function.

1.3 DS and algorithms

1.3.1 Introduction to DS and algorithms

What is Data Structure and Algorithms?

A data structure is defined as a particular way of storing and organizing data in our devices to use the data efficiently and effectively. The main idea behind using data structures is to minimize the time and space complexities. An efficient data structure takes minimum memory space and requires minimum time to execute the data.

Algorithm is defined as a process or set of well-defined instructions that are typically used to solve a particular group of problems or perform a specific type of calculation. To explain in simpler terms, it is a set of operations performed in a step-by-step manner to execute a task.

Applications of Data Structure and Algorithms:

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms:

Search Algorithm to search an item in a data structure.

Sort Algorithm to sort items in a certain order.

Insert Algorithm to insert item in a data structure.

Update Algorithm to update an existing item in a data structure.

Delete Algorithm to delete an existing item from a data structure

1.3.2 Analyze and design algorithms

What is meant by Algorithm Analysis?

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Algorithms are often quite different from one another, though the objective of these algorithms are the same. For example, we know that a set of numbers can be sorted using different algorithms. Number of comparisons performed by one algorithm may vary with others for the same input. Hence, time complexity of those algorithms may differ. At the same time, we need to calculate the memory space required by each algorithm.

Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis

Worst-case The maximum number of steps taken on any instance of size

Best-case The minimum number of steps taken on any instance of size

Average case An average number of steps taken on any instance of size

Amortized A sequence of operations applied to the input of size averaged over time.

1.3.3 Asymptotic notation:

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value. For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case. When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations:

a. Big-O notation:
Big O notation is an asymptotic notation that measures the performance of an algorithm by simply providing the order of growth of the function.

This notation provides an upper bound on a function which ensures that the function never grows faster than the upper bound. So, it gives the least upper bound on a function so that the function never grows faster than this upper bound. It is the formal way to express the upper boundary of an algorithm running time. It measures the worst case of time complexity or the algorithm's longest amount of time to complete its operation. It is represented as shown below:

The idea of using big o notation is to give an upper bound of a particular function, and eventually it leads to give a worst-time complexity. It provides an assurance that a particular function does not behave suddenly as a quadratic or a cubic fashion, it just behaves in a linear manner in a worst-case.

b. Omega notation:
It basically describes the best-case scenario which is opposite to the big o notation.

It is the formal way to represent the lower bound of an algorithm's running time. It measures the best amount of time an algorithm can possibly take to complete or the best-case time complexity.

It determines what the fastest time that an algorithm can run is. If we required that an algorithm takes at least certain amount of time without using an upper bound, we use big- notation i.e. the Greek letter "omega". It is used to bind the growth of running time for large input size.

If f(n) and g(n) are the two functions defined for positive integers, then f(n) = (g(n)) as f(n) is Omega of g(n) or f(n) is on the order of g(n)) if there exists constants c and no such that:
f(n)>=c.g(n) for all nno and c>0

As we can see in the above figure that g(n) function is the lower bound of the f(n) function when the value of c is equal to 1. Therefore, this notation gives the fastest running time. But, we are not more interested in finding the fastest running time, we are interested in calculating the worst-case scenarios because we want to check our algorithm for larger input that what is the worst time that it will take so that we can take further decision in the further process.

c. Theta notation:
The theta notation mainly describes the average case scenarios. It represents the realistic time complexity of an algorithm. Every time, an algorithm does not perform worst or best, in real-world problems, algorithms mainly fluctuate between the worst-case and best-case, and this gives us the average case of the algorithm. Big theta is mainly used when the value of worst-case and the best-case is same.

It is the formal way to express both the upper bound and lower bound of an algorithm running time. Let's understand the big theta notation mathematically: Let f(n) and g(n) be the functions of n where n is the steps required to execute the program then: f(n)= g(n) The above condition is satisfied only if when c1.g(n)<=f(n)<=c2.g(n) where the function is bounded by two limits, i.e., upper and lower limit, and f(n) comes in between. The condition f(n)= g(n) will be true if and only if c1.g(n) is less than or equal to f(n) and c2.g(n) is greater than or equal to f(n). The graphical representation of theta notation is given below:

1.3.4 Basic recursion algorithms on memory level

.

What is Recursion? The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. Using a 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. A recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems. Many more recursive calls can be generated as and when required. It is essential to know that we should provide a certain case in order to terminate this recursion process. So we can say that every time the function calls itself with a simpler version of the original problem.

Need of Recursion:
Recursion is an amazing technique with the help of which we can reduce the length of our code and make it easier to read and write. It has certain advantages over the iteration technique which will be discussed later. A task that can be defined with its similar subtask, recursion is one of the best solutions for it. For example; The Factorial of a number.

Memory Allocation in Recursion: When any function is called from main(), the memory is allocated to it on the stack. A recursive function calls itself, the memory for a called function is allocated on top of memory allocated to the calling function and a 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.

When a recursive function is called for the first time, a space is set aside in the memory to execute this call and the function body is executed. Then a second call to a function is made; again a space is set for this call, and so on. In other words, memory spaces for each function call are arranged in a stack. Each time a function is called; its memory area is placed on the top of the stack and is removed when the execution of the call is completed.

2 TRAINING WORK UNDERTAKEN

2.1 Arrays

The array is that type of structure that stores homogeneous elements at memory locations which are contiguous. The same types of objects are stored sequentially in an array. The main idea of an array is that multiple data of the same type can be stored together. Before storing the data in an array, the size of the array has to be defined. Any element in the array can be accessed or modified and the elements stored are indexed to identify their locations.An array can be explained with the help of a simple example of storing the marks for all the students in a class. Suppose there are 20 students, then the size of the array has to be mentioned as 20. Marks of all the students can then be stored in the created array without the need for creating separate variables for marks for every student. Simple traversal of the array can lead to the access of the elements.

CODE:

```
include <iostream>
usingnamespace std;
int main()
intarr[5]=10, 0, 20, 0, 30;
for (inti = 0; i< 5; i++)
cout«arr[i]«"";
```

Types of Arrays: There are two types of C++ arrays:

One-Dimensional Array:

This is an array in which the data items are arranged linearly in one dimension only. It is commonly called a 1-D array.

Syntax: datatype array-name[size];

The array-name is the name of the array.

The size is the number of items to be stored in the array.

Multi-dimensional Array:

This is an array in which data items are arranged to form an array of arrays. A multi-dimensional array can have any number of dimensions, but two-dimensional and three-dimensional arrays are common.

Syntax:

datatype array-name[d1][d2][d3]...[dn];

The array-name is the name of the array that will have n dimensions.

Two Dimensional Array:

A 2D array stores data in a list with 1-D array. It is a matrix with rows and columns. To declare a 2D array, use the following syntax:

type array-Name [x][y];

The type must be a valid C++ data type. See a 2D array as a table, where x denotes the number of rows while y denotes the number of columns. This means that you identify each element in a 2D array using the form a[x][y], where x is the number of row and y the number of columns in which the element belongs.

2.2 Linked Lists

The linked list is that type of data structure where separate objects are stored sequentially. Every object stored in the data structure will have the data and a reference to the next object. The last node of the linked list has a reference to null. The first element of the linked list is known as the head of the list. There are many differences between a linked list to the other types of data structures. These are in terms of memory allocation, the internal structure of the data structure, and the operations carried on the linked list. Getting to an element in a linked list is a slower process compared to the arrays as the indexing in an array helps in locating the element. However, in the case of a linked list, the process has to start from the head and traverse through the whole structure until the desired element is reached. In contrast to this, the advantage of using linked lists is that the addition or deletion of elements at the beginning can be done very quickly. There are three types of linked lists:

Single Linked List: This type of structure has the address or the reference of the next node stored in the current node. Therefore, a node which at the last has the address and reference as a NULL.

A Double Linked List: As the name suggests, each node has two references associated with it. One reference directs to the previous node while the second reference points to the next node. Traversal is possible in both directions as reference is available for the previous nodes. Also, explicit access is not required for deletion.

Linked List circular: The nodes in a circular linked list are connected in a way that a circle is formed. As the linked list is circular there is no end and hence no NULL. This type of linked list can follow the structure of both singly or doubly. There is no specific starting node and any node from the data can be the starting node. The reference of the last node points towards the first node.

2.3 Stack

The stack is another type of structure where the elements stored in the data structure follow the rule of LIFO (last in, first out) or FILO (First In Last Out). Two types of operations are associated with a stack i.e. push and pop. Push is used when an element has to be added to the collection and pop is used when the last element has to be removed from the collection. Extraction can be carried out for only the last added element.

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.

2.4 Queue

Queue is the type of data structure where the elements to be stored follow the rule of First In First Out (FIFO). The particular order is followed for performing the required operations over the elements. The difference of a queue from that of a stack lies in the removal of an element, where the most recently added object is removed first in a stack. Whereas, in the case of a queue, the element that was added first is removed first. Both the end of the data structure is used for the insertion and the removal of data.
The two main operations governing the structure of the queue are enqueue, and dequeue. Enqueue refers to the process where inserting an element is allowed to the collection of data and dequeue refers to the process where removal of elements is allowed, which is the first element in the queue in this case.

Example of the queue: Similar to those queues made while waiting for the bus or anywhere, the data structure too follows the same pattern. We can imagine a person waiting for the bus and standing at the first position as the person that came to the queue first. This person will be the first one who will get onto a bus, i.e. exit the queue. Queues are applied when multiple users are sharing the same resources and they have to be served on the basis of who has come first on the server.
In the above program, the function Insert() inserts an element into the queue. If the rear is equal to n-1, then the queue is full and overflow is displayed. If front is -1, it is incremented by 1. Then rear is incremented by 1 and the element is inserted in index of rear. In the function Delete(), if there are no elements in the queue then it is underflow condition. Otherwise the element at front is displayed and front is incremented by one

2.5 Sorting

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order. The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats.

2.5.1 Bubble Sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of (n2) where n is the number of items.

Two elements are said to be out-of-order if they do not follow the desired order. In a list which had elements 5, 3, 4, 2 in it. Here, 5 and 3 are out-of-order if we want to arrange the list in ascending order. Furthermore, 5 and 3 are in order if we want to sort the list in descending order.

In bubble sort, the repetition continues till the list we get the sorted list. Bubble compares all the elements in a list successively and sorts them based on their values.

Since it compares all the elements one by one, bubble sort is a slow algorithm and performs inefficiently in real-world scenarios. It is generally used for educational purposes or to determine if the sorted given list is already sorted or not.

2.5.2 Insertion Sort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Algorithm for Insertion Sort:
Step 1 If the element is the first one, it is already sorted.
Step 2 – Move to next element
Step 3 Compare the current element with all elements in the sorted array
Step 4 – If the element in the sorted array is smaller than the current element, iterate to the next element. Otherwise, shift all the greater element in the array by one position towards the right
Step 5 Insert the value at the correct position
Step 6 Repeat until the complete list is sorted

2.5.3 Merge Sort

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being (n log n), it is one of the most respected algorithms. Merge sort first divides the array into equal halves and then combines them in a sorted manner.

Think of it as a recursive algorithm continuously splits the array in half until it cannot be further divided. This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion. If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves. Finally, when both halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.

2.5.4 Quick Sort

QuickSort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

Always pick the first element as a pivot.

Always pick the last element as a pivot

Pick a random element as a pivot.

Pick median as the pivot.

The key process in quickSort is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

2.6 Notion of Hashing and Collision

2.6.1 Hashing:

Hashing is a technique of mapping a large set of arbitrary data to tabular indexes using a hash function. It is a method for representing dictionaries for large datasets. This is usually represented by a shorter, fixed-length value or key that represents and makes it easier to find or employ the original string. The most popular use for hashing is the implementation of hash tables. Hashing is a popular technique for storing and retrieving data as fast as possible. The main reason behind using hashing is that it gives optimal results as it performs optimal searches.

Why to use Hashing? : If you observe carefully, in a balanced binary search tree, if we try to search , insert or delete any element then the time complexity for the same is O(logn). Now there might be a situation when our applications want to do the same operations in a faster way i.e. in a more optimized way and here hashing comes into play. In hashing, all the above operations can be performed in O(1) i.e. constant time. It is important to understand that the worst case time complexity for hashing remains O(n) but the average case time complexity is O(1). Let a hash function H(x) maps the value at the index x%10 in an Array. For example if the list of values is [11,12,13,14,15] it will be stored at positions 1,2,3,4,5 in the array or Hash table respectively.

2.6.2 Collision:

A hash collision or clash is when two pieces of data in a hash table share the same hash value. The hash value in this case is derived from a hash function which takes a data input and returns a fixed length of bits.

Although hash algorithms have been created with the intent of being collision resistant, they can still sometimes map different data to the same hash (by virtue of the pigeonhole principle). Malicious users can take advantage of this to mimic, access, or alter data.

2.6.3 Hash Functions

A hash function is any function that can be used to map data of arbitrary size to fixed-size values. A hash function takes a group of characters (called a key) and maps it to a value of a certain length (called a hash value or hash). The hash value is representative of the original string of characters, but is normally smaller than the original.

Hashing is used with a database to enable items to be retrieved more quickly. Hashing can also be used in the encryption and decryption of digital signatures. The hash function transforms the digital signature, then both the hash value and signature are sent to the receiver. The receiver uses the same hash function to generate the hash value and then compares it to that received with the message. If the hash values are the same, it is likely that the message was transmitted without errors.

One example of a hash function is called folding. This takes an original value, divides it into several parts, then adds the parts and uses the last four remaining digits as the hashed value or key.

Another example is called digit rearrangement. This takes the digits in certain positions of the original value, such as the third and sixth numbers, and reverses their order. It then uses the number left over as the hashed value.
A good hash function should have the following properties:
1. Efficiently computable.
2. Should uniformly distribute the keys (Each table position equally likely for each key)

2.6.4 Method of Collision Handling

There are mainly two methods to handle collision:

- Separate Chaining
- Open Addressing

Separate Chaining:
The idea behind separate chaining is to implement the array as a linked list called a chain. Separate chaining is one of the most popular and commonly used techniques in order to handle collisions.

The linked list data structure is used to implement this technique. So what happens is, when multiple elements are hashed into the same slot index, then these elements are inserted into a singly-linked list which is known as a chain. Here, all those elements that hash into the same slot index are inserted into a linked list. Now, we can use a key K to search in the linked list by just linearly traversing. If the intrinsic key for any entry is equal to K then it means that we have found our entry. If we have reached the end of the linked list and yet we haven't found our entry then it means that the entry does not exist. Hence, the conclusion is that in separate chaining, if two different elements have the same hash value then we store both the elements in the same linked list one after the other.

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

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, the 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). This approach is also known as closed hashing.

This entire procedure is based upon probing. We will understand the types of probing ahead:

• Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.

• Search(k): Keep probing until the 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 the search may fail. So slots of deleted keys are marked specially as "deleted". The insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

3 Results and discussions

3.1 Snapshots of Topics covered

1. Arrays

An example of how arrays work is shown below:

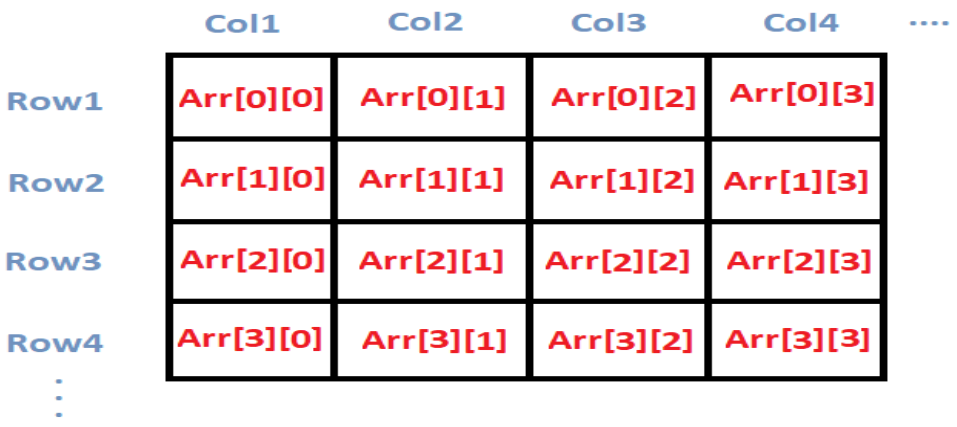


Figure 1: Arrays

2. Linkedlists

Every object stored in the data structure will have the data and a reference to the next object. The last node of the linked list has a reference to null.

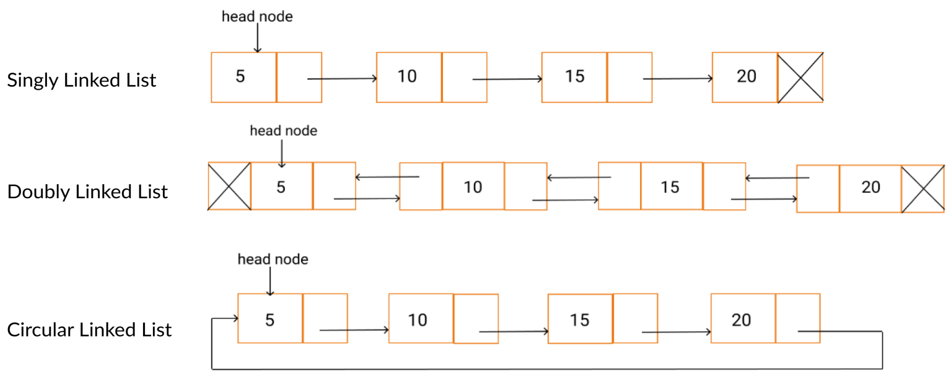


Figure 2: Linkedlist working

3.Sorting

The working of sorting techniques like bubble sort and merge sort is shown below:

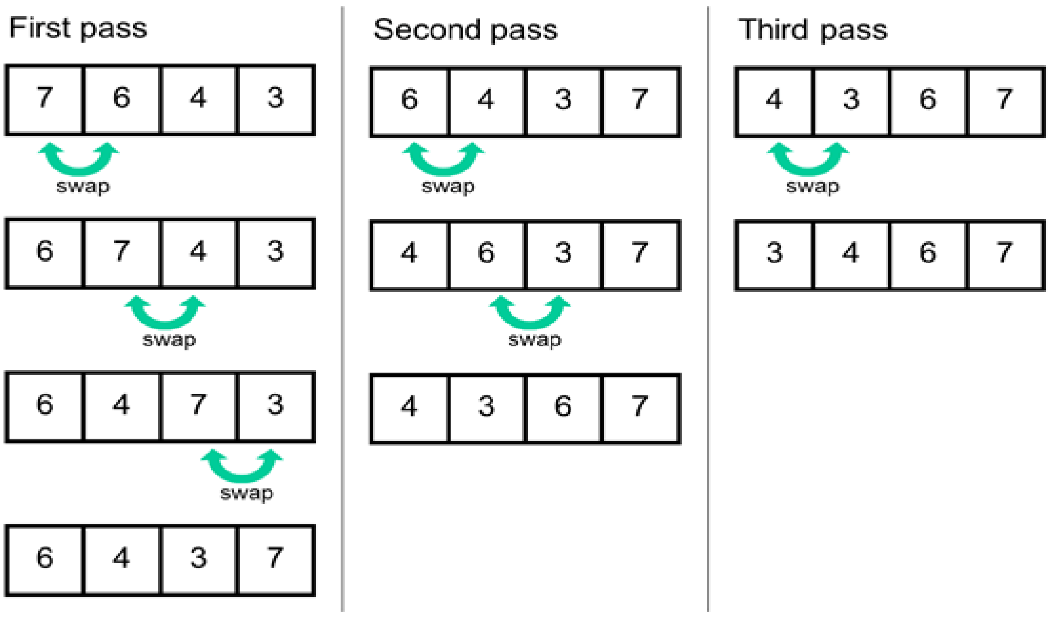
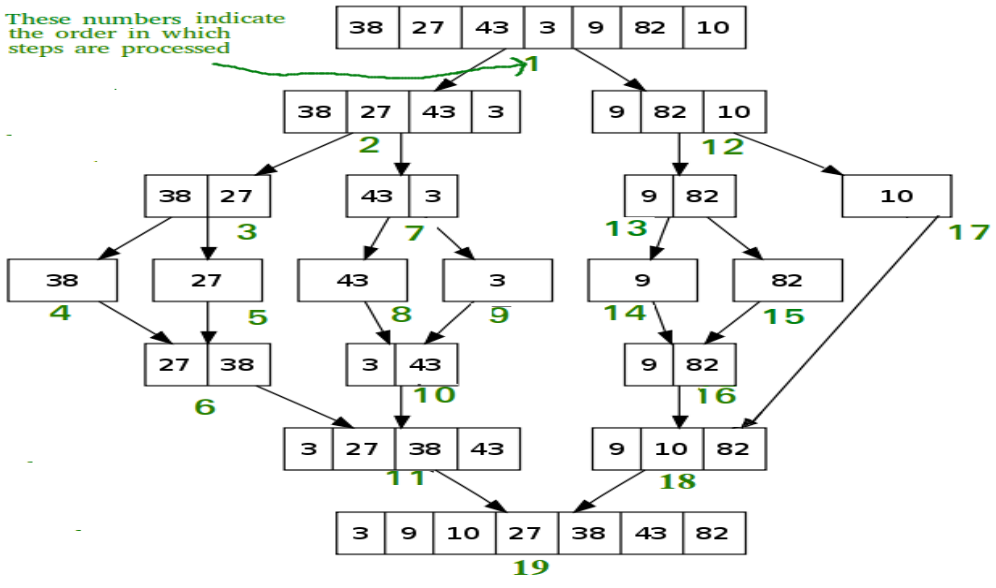


Figure 3: Bubble sort


These numbers indicate the order in which steps are processed

merge sort

4. Stack

The elements stored in the data structure follow the rule of LIFO (last in, first out) or FILO (First In Last Out).

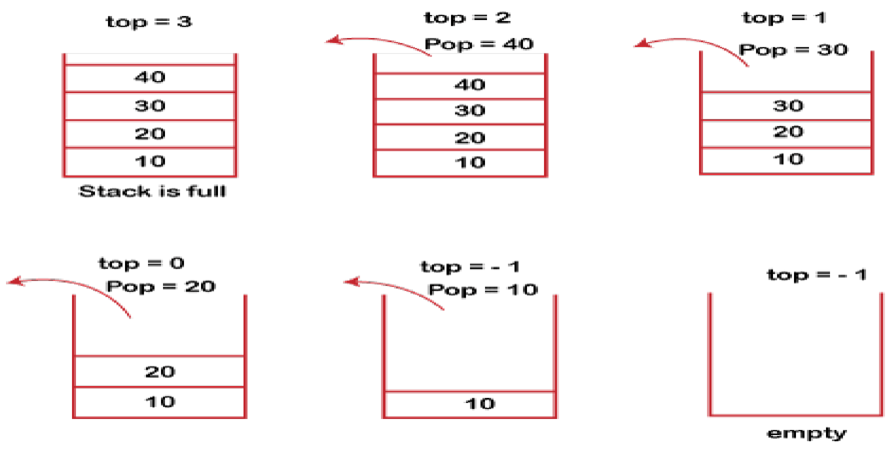


Figure 4: StackWorking

5. Queue

The elements to be stored follow the rule of First In First Out (FIFO). The particular order is followed for performing the required operations over the elements

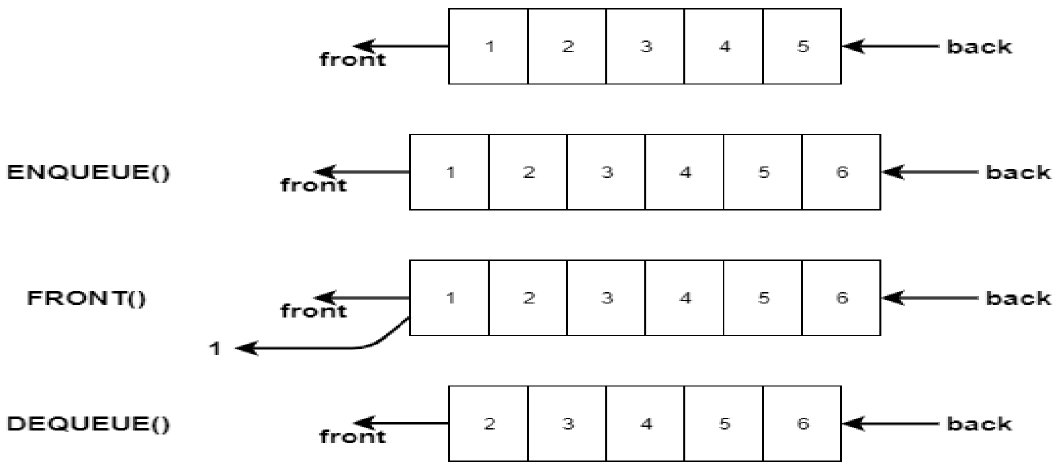


Figure 5: Queue Working

4 Conclusions and Future Scope

4.1 Conclusion of Training Work

The data structure is simply organizing the data in the memory, so that memory can be used efficiently and the process of data retrieval becomes fast. There are many kinds of data structures that are categorized into different categories, some data structures are array, stack, queue, linked list, etc.

With the help of data structure, we can minimize the search speed which will eventually increase the response time and processor speed as well. Data can be organized in such a manner, that while searching for a particular item we need not search entire data, Or the most relevant data can be kept on priority and it can be accessed more quickly than others, and many more. So, the data structure is the backbone of today's highly efficient and faster-moving world.

If Data structure is studied along with Algorithms, we will be able to:

Write optimized code: Once you will get to know about different data structures then you can code more efficient programs, you can identify which data structure is best for which kind of data, and code it accordingly.

Better utilization of Time Memory: Time and memory are the most costlier element. With the help of data structure, you will be able to write code that is really quick and uses less memory space.

Job opportunities: There is a high demand for good Software developers worldwide and with sufficient knowledge of data structure and algorithms you can easily crack the interviews of Google, Microsoft, Amazon, and many more.

4.2 Future Scope of Training work

Data structures will become more amorphous and distributed. Data structures will become more and more indistinguishable from the techniques used to manipulate them. More heuristic approaches will be employed to relate huge data stores, and those relationships will become more probabilistic. Data structures will become more deeply embedded with representational approaches to make it more actively computable. Expect to see the term "Smart Data" pop up in the near future. Artificial General Intelligence, once enabled, will alter our notion and relationship with data forever.

5 Problem Based on Training

5.1 C++ Program to Check for balanced paranthesis by using Stacks

Here we will discuss how to check the balanced brackets using stacks. We not only check the opening and closing brackets but also check the ordering of brackets. For an example we can say that the expression "[() ()]" it is correct, but "[]" it is not correct.

1. Input: Some expression with brackets "()[]"
2. Output: They are balanced

5.1.1 Algorithm

- Define a stack to hold brackets
- Traverse the expression from left to right
- If the character is opening bracket (, or or [, then push it into stack
- If the character is closing bracket), or] Then pop from stack, and if the popped character is matched with the starting bracket then it is ok. otherwise they are not balanced.
- After traversal if the starting bracket is present in the stack then it is not balanced.

5.1.2 Code

```
include<iostream>
include<stack>
using namespace std;
bool isBalanced(string expr)
stack<char> s;
char ch;
for (int i=0; i<expr.length(); i++)  //for each character in the expression, check conditions
if (expr[i]==’(’||expr[i]==’[’||expr[i]==’’)  //when it is opening bracket, push into stack
s.push(expr[i]);
continue;
if (s.empty()) //stack cannot be empty as it is not opening bracket, there must be closing bracket
```

```
return false;
switch (expr[i])
case ’)’: //for closing parenthesis, pop it and check for braces and square brackets
ch = s.top();
s.pop();
if (ch==’’ || ch==’[’)
return false;
break;
case ’’: //for closing braces, pop it and check for parenthesis and square brackets
ch = s.top();
s.pop();
if (ch==’(’ || ch==’[’)
return false;
break;
case ’]’: //for closing square bracket, pop it and check for braces and parenthesis
ch = s.top();
s.pop();
if (ch ==’(’ || ch == ’’)
return false;
break;
return (s.empty()); //when stack is empty, return true
main()
string expr = "[()()]";
if (isBalanced(expr))
cout « "Balanced";
else
cout « "Not Balanced";
```

6 References

1. https://www.w3schools.com/cpp/
2. https://www.youtube.com/watch?v=Xx1tWbb37hY
3. https://www.javatpoint.com/data-structure-tutorial
4. https://www.geeksforgeeks.org/data-structures/