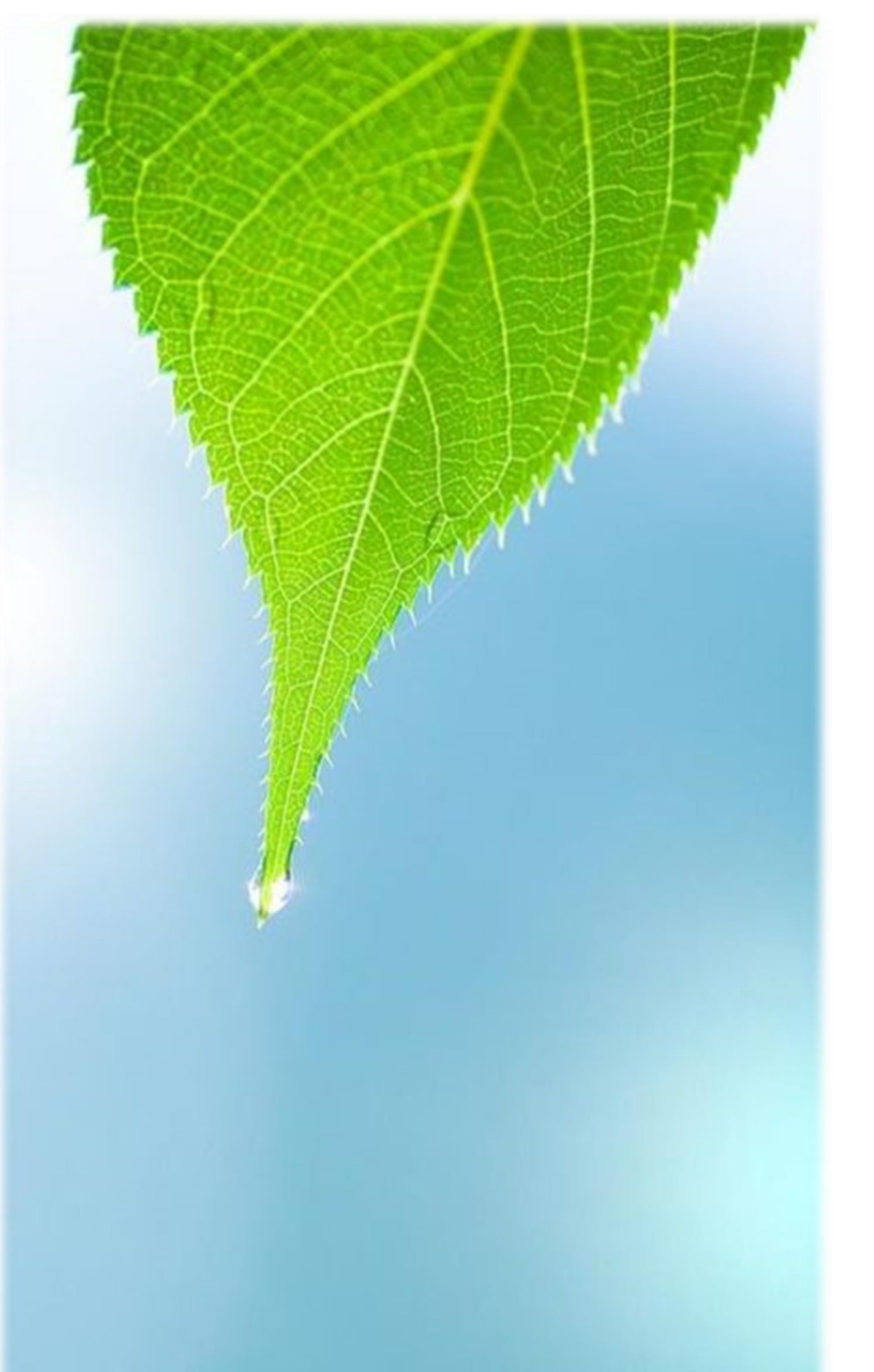
Data Structures & Algorithms

2023

Lab Manual

Faculty of Computer Science and Engineering, GIKI



Contents

I) OBE COURSE OUTLINES 10
II) BENCHMARK REPORT 13
III) LAB EVALUATION AND RUBRICS 16
IV) WEEKLY CONTENTS BREAKDOWN 18
Lab#01 Structures, Pointers, Arrays 22
1.1 C++ Structures 22
1.1.1 Defining a Structure 22
1.1.2 Accessing Structure Members 23
1.1.3 Structures as Function Arguments 24
1.1.4 Pointers to Structures 25
1.1.5 The typedef Keyword 26
1.2 Introduction to Pointers in Data Structure 27
1.2.1 Why do We Need Pointers in Data Structure? 27
1.2.1.1 Control Program Flow 27
1.2.1.2 secondary data structures 28
1.2.1.3 Dynamic Memory Allocation 28
1.2.2 How do Pointers Work in Data Structure? 28
C Program on Pointers 30
1.2.3 Disadvantage Of Pointers 31
1.3 Dynamic Allocation of 2D Arrays in C++ (with code) 31
1.3.1 What is a Dynamic 2D Array in C++? 32
1.3.2 Methods to Dynamically Allocate a 2D Array 32
1.3.2.1 Single Pointer Method 32
1.3.2.2 Using an Array of Pointer 34
Practice Problems: 36
Lab#02 Abstract Data types and Single Linked List 39
2.1 Introduction to ADT 39
2.1.1 Abstract Data Type Model 39
2.2 Introduction to List 40
2.2.1 Why Do We Need a Linked List? 40
2.2.2 Types of Linked List 42
2.2.2.1 Singly Linked List 42
2.2.2.2How Can We Declare a Linked List? 42
2.2.3 Operations on Linked List 43

2.2.3.1 Insertion..... 44
2.2.3.2 Deletion..... 46
2.2.3.3 Traversal..... 47
2.2.3.4 Search..... 48
2.2.4 Implementation of Linked List in C++..... 48
2.2.5 Time Complexity of Linked List..... 49
2.2.6 Advantages / Disadvantages of Linked List..... 50
2.2.7 Applications of Linked List..... 51
Practice Problems:..... 52
Lab 03..... 54
Doubly Linked List..... 54
3.1 Doubly Linked List..... 54
3.2 Representation of Doubly Linked List..... 54
3.3 Insertion on a Doubly Linked List..... 56
3.3.1 Insertion at the Beginning..... 56
Code for Insertion at the Beginning..... 57
3.3.2 Insertion in between two nodes..... 58
Code for Insertion in between two Nodes..... 59
3.3.3. Insertion at the End..... 61
3.4 Deletion from a Doubly Linked List..... 63
3.4.1 Delete the First Node of Doubly Linked List..... 63
3.4.2 Deletion of the Inner Node..... 64
3.3.3 Delete the Last Node of Doubly Linked List..... 65
3.4 Doubly Linked List Code in C++..... 66
3.5 Doubly Linked List Complexity..... 71
3.6 Doubly Linked List Applications..... 71
3.7 Singly Linked List Vs Doubly Linked List..... 72
3.8 Practice Problems..... 72
Lab# 04..... 75
Stack Data Structure in C++..... 75
4.1 Introduction to Stack in C++..... 75
4.1.1 LIFO(Last In First Out):..... 75
4.2 Basic Operations..... 76
4.2.1 PUSH..... 76
4.2.2 POP..... 76

4.2.3 Illustration 76
4.3 Array Implementation For Stack 77
4.4 Linked List Implementation for Stack: 80
4.5 Analysis of Stack Operations 84
4.6 Applications Of Stack 84
Practice Problems: 85
Lab# 05 87
Queue Data Structure In C++ 87
5.1 Introduction To Queue In C++ With Illustration 87
5.2 Basic Operations 87
5.2.1 Enqueue 88
5.2.2 Dequeue 88
5.2.3 Illustration 88
5.3 Array Implementation For Queue 89
5.4 Linked List Implementation for Queue: 93
5.5 Stack Vs. Queue 95
5.6 Applications Of Queue 96
Conclusion 96
Practice Problems: 96
Lab#06 98
Sorting Algorithms I 98
6.1 Bubble Sort 98
6.1.1 How does Bubble Sort work? 98
6.1.2 Implementing Bubble Sort Algorithm: Suppose we are trying to sort the elements in ascending order. 1. First Iteration (Compare and Swap: 98
6.1.2.1 Bubble Sort Algorithm 100
6.1.3 Bubble Sort Implementation in C++ 100
6.1.4 Complexity Analysis of Bubble Sort: 101
6.1.5 Applications of Bubble Sort: 101
6.1.6 Advantages of Bubble Sort: 101
6.1.7 Drawbacks of Bubble Sort: 101
6.2 Insertion Sort – Data Structure and Algorithm C++ 102
6.2.1 How does Insertion Sort Work? 102
6.2.1.1 Illustration: 102
6.2.2 Insertion Sort Algorithm 104

6.2.3 Insertion Sort Implementation in C++ 104
6.2.4 Complexity Analysis of Insertion Sort: 105
6.2.5 Applications of Insertion Sort 105
6.2.6 Advantages of Insertion Sort: 106
6.2.7 Disadvantages of Insertion Sort: 106
6.3 Selection Sort 106
6.3.1 How Selection Sort Works? 106
6.3.1.1 Illustration: 106
6.3.2 Selection Sort Algorithm 108
6.3.3 Insertion Sort Implementation in C++ 109
6.3.4 Complexity Analysis of Selection Sort 110
6.3.5 Selection Sort Applications 110
6.3.6 Advantages of Selection Sort 110
6.3.7 Disadvantages of Selection Sort 110
Practice Problems: 110
Lab#07 114
Sorting Algorithms II 114
7.1 Merge Sort 114
7.1.1 How does Merge Sort work? 114
7.1.2 Illustration: 114
7.1.3 Merge Sort Implementation in C++ 116
7.1.4 Complexity Analysis of Merge Sort: 120
7.1.5 Applications of Merge Sort: 120
7.1.6 Advantages of Merge Sort: 120
7.1.7 Drawbacks of Merge Sort: 121
7.2 QuickSort – Data Structure and Algorithm C++ 121
7.2.1 How does QuickSort work? 121
7.2.2 Choice of Pivot: 122
7.2.3 Partition Algorithm: 122
7.2.4 Illustration of Quicksort: 123
7.2.5 Code implementation of the Quick Sort 124
7.2.6 Complexity Analysis of Quick Sort: 126
7.2.7 Advantages of Quick Sort: 126
7.2.8 Disadvantages of Quick Sort: 126
Practice Problems: 126

Lab#08 128
Binary Search Tree (BST) 128
8.1 Binary Search Tree (BST) 128
8.1.1 Properties of BST: 128
8.1.2 Example 128
8.2 Operations on a Binary Search Tree 129
8.2.1 Search Operation in BST 129
8.2.1.1 Algorithm: 129
8.2.1.2 Illustration: 130
8.2.2 Insertion Operation in BST 131
8.2.2.1 Algorithm: 131
8.2.2.2 Illustration: 132
8.2.3 Deletion Operation in BST 133
8.3 Implementation in C++ 137
8.4 Binary Search Tree Complexities 139
8.5 Binary Search Tree Applications 140
8.6 Advantages of Binary Search Tree: 140
8.6.1 Disadvantages of Binary Search Tree: 140
Practice Problems: 141
9.1 AVL Tree 144
9.2 AVL Tree Balance Factor 144
9.3 Operations on an AVL tree 144
9.3.1 Rotating the subtrees in an AVL Tree 144
Left Rotate 145
Right Rotate 146
9.3.2 Algorithm to insert a newNode 149
9.4 Algorithm to Delete a node 154
9.5 AVL Tree Implementation in C++ 157
9.6 Complexities of Different Operations on an AVL Tree 164
9.7 AVL Tree Applications 164
Practice Problems: 165
Lab#10 167
Graph Algorithms 167
10.1 Graph Data Structure 167
10.2 Why Graph Algorithms are Important? 168

10.3 An application in real life 168
10.4 Type of graphs 168
10.5 Operations on Graphs in Data Structures 169
10.5.1 Creating Graph OR Graph representation 169
10.5.*2* Adjacency Matrix 169
10.5.*3* Adjacency List 169
10.5.4 Insert Vertex 171
10.5.5 Delete Vertex 171
10.5.6 Insert Edge 172
10.5.7 Delete Edge 172
10.6 Graph Traversal Algorithm 172
10.6.1 Breadth-First Search or BFS 172
10.6.*2* C++ code for BFS implementation 173
10.6.*3* Depth-First Search or DFS 175
10.6.*4* code for DFS implementation 176
Visualisation 176
Implementation of DFS in C++ 178
10.7 Difference between BFS and DFS 180
Practice Problems: 181
Lab#11 Hash Table and Hash Function in Data Structures 183
11.1 Definition of C++ Hash Table 183
11.2 Need for Hash data structure 183
11.3 Components of Hashing 183
11.4 How does Hashing work? 184
11.5 C++ Implementation of Hash Table 186
11.6 What is a Hash function? 190
11.6.1 Types of Hash functions: 190
11.6.2 Properties of a Good hash function 190
11.6.3 Problem with Hashing 191
11.6.4 What is collision? 191
11.6.5 How to handle Collisions? 191
1) Separate Chaining 192
2) Open Addressing 195
2.a) Linear Probing 195
11.7 What is meant by Load Factor in Hashing? 204

11.8 What is Rehashing? 204
11.9 Applications of Hash Data structure 204
11.10 Advantages of Hash Data structure 205
11.11 Disadvantages of Hash Data structure 205
Practice Problems: 205
Lab 12 208
Open Ended Lab 208
I. REFERENCES 209
THE END 209

I) OBE COURSE OUTLINES

Engr. Amna Arooj
FACULTY OF COMPUTER SCIENCE AND ENGINEERING (FCSE)
GHULAM ISHAQ KHAN INSTITUTE OF ENGINEERING SCIENCES AND TECHNLOLOGY (GIKI)

CS221L - DATA STRUCTURES AND ALGORITHMS

CS221 Data Structures Lab (1 CH)

Pre-Requisite:CS112L

Instructor: **Engr. Amna Arooj**
Office # G35, GIK Institute, Ext. 2746
Email: *amna.arooj@giki.edu.pk*
Office Hours: 9:00am ~ 11:00 am

Lab Introduction

This lab aims to introduce the fundamental concept of data structures and to emphasize the importance of data structures in developing and implementing efficient algorithms. Efficient data structure provides basis for a good algorithm (code). This lab focuses on the most common data structures utilized in various computational problems. It will be taught that how these data structures work and their implementation in C++/C. Students will practice implementing them in a few programming tasks. This will help them understand the nuts and bolts of various data structures and enable to write efficient programs.

Lab Contents

Broadly, this lab will cover following contents: algorithm design, Abstract Data Types (ADTs), lists, stacks, queues, trees, Binary trees, B-trees, AVL tree, hashing, sorting, graph algorithms and other recent topics in data structures.

Mapping of CLOs and PLOs

Sr. No	Course Learning Outcomes[+]	PLOs*	Taxonomy level (Psychomotor Domain)
CLO 1	Implement primitive data structures, like linked lists, arrays, and trees to design solutions for a given problem.	PLO 3	**P3**
CLO 2	Map requirements of a computational problem to the various abstract data types.	PLO 3	**P3**
CLO 3	Design efficient solutions using the concepts of data structures for solving various real-world problems.	PLO 3	**P4**
	[+]Please add the prefix "Upon successful completion of this course, the student will be able to" [*]PLOs are for BS (CE) only		

CLO Assessment Mechanism

Assessment tools	CLO_1	CLO_2	CLO_3
Lab Tasks	**50%**	**50%**	**40%**
Midterm Exam	**50%**	**50%**	**-**
Open Ended Lab	**-**	**-**	**10%**
Final Exam	**-**	**-**	**50%**

Overall Grading Policy

Assessment Items	Percentage
Lab Evaluation	**30%**
Midterm Exam	**20%**
Open Ended Lab	**5%**
Final Exam	**30%**

Text and Reference Books

Text books:

- Lab Manual

- Introduction to Algorithms, Thomas H. Cormen et al, 2009.
- Data Structure using C++ D.S. Malik

Administrative Instruction

- According to institute policy, 100% attendance is *mandatory* to appear in the final examination.
- Assignments assigned must be submitted as per instructions mentioned in the assignments.
- For queries, kindly follow the office hours to avoid any inconvenience.

Computer Usage/Software tools

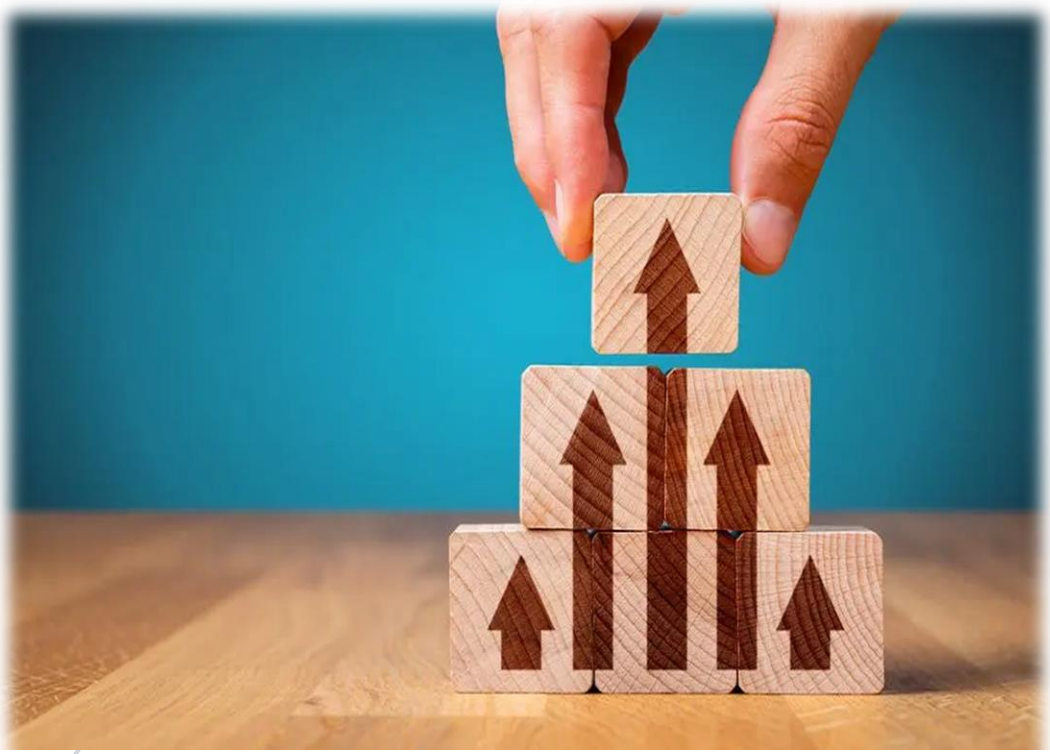
- Students are encouraged to solve some assigned homework and lab tasks using the available programming software, such as DevC, Visual Studio community version and Visual Studio code

Lecture Breakdown

Lab 1	Pointers Structures Arrays (static and Dynamic)
Lab 2	Abstract Data Types (ADT) List Data Structure Single Linked List
Lab 3	List Data Structure Double Linked List
Lab 4	Stacks o Array Implementation o Linked List Implementation
Lab 5	Queue o Array Implementation o Linked List Implementation
Lab 6	Sorting I o Bubble sort o Insertion sort o Selection sort
Lab 7	Sorting II o Merge sort o Quick sort
Lab 8	Binary Search Tree
Lab 9	AVL Trees
Lab 10	Graph Algorithms
Lab 11	Hash table and Hash Function in Data Structures
Lab 12	Open Ended Lab

1.

II) BENCHMARK REPORT



Engr. Amna Arooj
FACULTY OF COMPUTER SCIENCE AND ENGINEERING (FCSE)
GHULAM ISHAQ KHAN INSTITUTE OF ENGINEERING SCIENCES AND TECHNLOLOGY (GIKI)

Data Structures Lab (Comparison of different universities)

Labs	GIKI	UNF (USA)	UNSW(Australia)	CIT (India)	KFUPM (KSA)
Programming Language	C++	C++	C++	C++	C++
Lab 1	Structures, Arrays, Pointers	Introduction	Revision	Arrays	OOP Review
Lab 2	Abstract Data Types Linked Lists	Abstract Data Types	Linked Lists	Strings	Algorithms Analysis
Lab 3	Doubly Linked Lists	Arrays	ADT	Stacks	Linked Lists
Lab 4	Stacks	Lists	ADTs, benchmarking and profiling	Stack Applications	Stacks & Queues
Lab 5	Queues	Binary Trees	Trees	Stack Applications II	Recursion
Lab 6	Sorting I	B trees	Sorting I	Queues	Binary Trees and BST
Lab 7	Sorting II	Heaps	Sorting II	Linked Lists	Binary Heaps
Lab 8	Binary Search Trees	Tries	Graphs	Circular Linked Lists	AVL Trees
Lab 9	AVL Trees	Multiway trees	Graph Algorithms	Binary Search Trees	Graphs
Lab 10	Graphs	Space partitioning trees	Trees Balancing	Graphs	Graph Algorithms
Lab 11	Hashing	Hashes	Hashing	Hashes	Hashing
Lab 12	Open-Ended Lab	Graphs			Applications: Data Compression

Data Structures Topics comparison

Topic	GIKI	UNF (USA)	UNSW(Australia)	CIT (India)	KFUPM (KSA)
Structures	✓	✗	✗	✗	✗
Abstract Data Types	✓	✓	✓	✗	✗
Arrays	✓	✓	✗	✓	✗
Strings	✗	✗	✗	✓	✗
Pointers	✓	✗	✗	✗	✗
Recursion	✗	✗	✗	✗	✓
Linked Lists	✓	✓	✓	✓	✓
Doubly/Circular Linked Lists	✓	✗	✗	✓	✗
Stacks	✓	✓	✗	✓	✓
Queues	✓	✓	✗	✓	✓
Sorting I	✓	✗	✓	✗	✗
Sorting II	✓	✗	✓	✗	✗
Binary Search Trees	✓	✓	✗	✓	✓
B trees	✗	✓	✗	✗	✓
AVL Trees	✓	✓	✓	✗	✓
Space partitioning trees	✗	✓	✗	✗	✗
Heaps	✗	✓	✗	✗	✓
Graphs	✓	✓	✓	✓	✓
Hashes	✓	✓	✓	✓	✓
Open Ended Lab	✓	✗		✗	✗

III) LAB EVALUATION AND RUBRICS

Engr. Amna Arooj,
FACULTY OF COMPUTER SCIENCE AND ENGINEERING (FCSE)
GHULAM ISHAQ KHAN INSTITUTE OF ENGINEERING SCIENCES AND TECHNLOLOGY (GIKI)

Rubrics for Assessment of CS221-L Data Structure and Algorithms				
Criteria	**Total Marks**	**Excellent (2)**	**Average (1)**	**Poor (0)**
Accuracy	2	The solution does exactly what has been asked in the task including for tricky cases and gives the accurate/expected results for all parts.	The solution does exactly as asked in the task but does not handle exceptions/tricky input	The solution is poor, does not perform at all according to the task or gives absolutely wrong results even for standard input.
Clarity	2	The solution is clear, and the student is able to formulate and explain the solution coherently.	The solution is partially clear and the student can explain parts of the program	The student has not formulated the solution correctly or can not explain his/her solution at all.
Code	2	The code is coherently written and documented to the point that the reader can broadly understand the working of each major part.	The code is somewhat coherent, tabbing has not been used but has useful chunks written with broad documentation.	The code lacks readability, can not be explained without the coder and has little/no documentation.
Performance	2	The code is extremely fast and for standard medium sized input performs its processing quickly.	The code is somewhat fast, runs quickly for small inputs but may take long time for medium sized/large inputs.	The code is extremely slow and gets stuck even for small-sized inputs.
Completion Time	2	The final complete solution for all tasks was submitted before half of the class	The final complete solution for all tasks was submitted later than half of the class but before 80% of the size of class.	The final complete solution was submitted along with the last 20% of the class
Total	10	10	5	0

IV) WEEKLY CONTENTS BREAKDOWN

Engr. Amna Arooj
FACULTY OF COMPUTER SCIENCE AND ENGINEERING (FCSE)
GHULAM ISHAQ KHAN INSTITUTE OF ENGINEERING SCIENCES AND TECHNLOLOGY (GIKI)

Weekly Breakdown	
Week 1	Pointers Structures Arrays (static and Dynamic)
Week 2	Abstract Data Types (ADT) List Data Structure Single Linked List
Week 3	List Data Structure Double Linked List
Week 4	Stacks o Array Implementation o Linked List Implementation
Week 5	Queue o Array Implementation o Linked List Implementation
Week 6	Sorting I o Bubble sort o Insertion sort o Selection sort
Week 7	Mid term Exam
Week 8	Sorting II o Merge sort o Quick sort
Week 9	Binary Search Tree
Week 10	AVL Trees
Week 11	Graph Algorithms
Week 12	Hash Table and Hash Functions in Data structures
Week 13	Open Ended Lab
Week 14	Final Term Exam

Lab#01
Structures, Pointers, Arrays

Prepared by: Engr. Amna Arooj

FCSE, GIKI

Lab#01
Structures, Pointers, Arrays

1.1 C++ Structures

C/C++ arrays allow you to define variables that combine several data items of the same kind, but **structure** is another user defined data type which allows you to combine data items of different kinds.

Structures are used to represent a record, suppose you want to keep track of your books in a library. You might want to track the following attributes about each book −

Title

Author

Subject

Book ID

1.1.1 Defining a Structure

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member, for your program. The format of the struct statement is this −

```
struct [structure tag] {
  member definition;
  member definition;
  ...
  member definition;
} [one or more structure variables];
```

The **structure tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure −

```
struct Books {
  char  title[50];
  char  author[50];
  char  subject[100];
  int   book_id;
} book;
```

1.1.2 Accessing Structure Members

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use **struct** keyword to define variables of structure type. Following is the example to explain usage of structure.

```
#include <iostream>
#include <cstring>

using namespace std;

struct Books {
  char  title[50];
  char  author[50];
  char  subject[100];
  int   book_id;
};

int main() {
  struct Books Book1;       // Declare Book1 of type Book
  struct Books Book2;       // Declare Book2 of type Book

  // book 1 specification
  strcpy( Book1.title, "Learn C++ Programming");
  strcpy( Book1.author, "Chand Miyan");
  strcpy( Book1.subject, "C++ Programming");
  Book1.book_id = 6495407;

  // book 2 specification
  strcpy( Book2.title, "Telecom Billing");
  strcpy( Book2.author, "Yakit Singha");
  strcpy( Book2.subject, "Telecom");
  Book2.book_id = 6495700;

  // Print Book1 info
  cout << "Book 1 title : " << Book1.title <<endl;
  cout << "Book 1 author : " << Book1.author <<endl;
  cout << "Book 1 subject : " << Book1.subject <<endl;
  cout << "Book 1 id : " << Book1.book_id <<endl;

  // Print Book2 info
  cout << "Book 2 title : " << Book2.title <<endl;
  cout << "Book 2 author : " << Book2.author <<endl;
  cout << "Book 2 subject : " << Book2.subject <<endl;
  cout << "Book 2 id : " << Book2.book_id <<endl;

  return 0;
```

```
}
```

When the above code is compiled and executed, it produces the following result –

```
Book 1 title : Learn C++ Programming
Book 1 author : Chand Miyan
Book 1 subject : C++ Programming
Book 1 id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Yakit Singha
Book 2 subject : Telecom
Book 2 id : 6495700
```

1.1.3 Structures as Function Arguments

You can pass a structure as a function argument in very similar way as you pass any other variable or pointer. You would access structure variables in the similar way as you have accessed in the above example –

```
#include <iostream>
#include <cstring>

using namespace std;
void printBook( struct Books book );

struct Books {
  char  title[50];
  char  author[50];
  char  subject[100];
  int   book_id;
};

int main() {
  struct Books Book1;       // Declare Book1 of type Book
  struct Books Book2;       // Declare Book2 of type Book

  // book 1 specification
  strcpy( Book1.title, "Learn C++ Programming");
  strcpy( Book1.author, "Chand Miyan");
  strcpy( Book1.subject, "C++ Programming");
  Book1.book_id = 6495407;

  // book 2 specification
  strcpy( Book2.title, "Telecom Billing");
  strcpy( Book2.author, "Yakit Singha");
  strcpy( Book2.subject, "Telecom");
  Book2.book_id = 6495700;

  // Print Book1 info
```

```
   printBook( Book1 );

   // Print Book2 info
   printBook( Book2 );

   return 0;
}
void printBook( struct Books book ) {
   cout << "Book title : " << book.title <<endl;
   cout << "Book author : " << book.author <<endl;
   cout << "Book subject : " << book.subject <<endl;
   cout << "Book id : " << book.book_id <<endl;
}
```

When the above code is compiled and executed, it produces the following result –

```
Book title : Learn C++ Programming
Book author : Chand Miyan
Book subject : C++ Programming
Book id : 6495407
Book title : Telecom Billing
Book author : Yakit Singha
Book subject : Telecom
Book id : 6495700
```

1.1.4 Pointers to Structures

You can define pointers to structures in very similar way as you define pointer to any other variable as follows −

```
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the & operator before the structure's name as follows −

```
struct_pointer = &Book1;
```

To access the members of a structure using a pointer to that structure, you must use the -> operator as follows −

```
struct_pointer->title;
```

Let us re-write above example using structure pointer, hope this will be easy for you to understand the concept −

```
#include <iostream>
#include <cstring>

using namespace std;
void printBook( struct Books *book );

struct Books {
```

```
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};
int main() {
   struct Books Book1;        // Declare Book1 of type Book
   struct Books Book2;        // Declare Book2 of type Book

   // Book 1 specification
   strcpy( Book1.title, "Learn C++ Programming");
   strcpy( Book1.author, "Chand Miyan");
   strcpy( Book1.subject, "C++ Programming");
   Book1.book_id = 6495407;

   // Book 2 specification
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Yakit Singha");
   strcpy( Book2.subject, "Telecom");
   Book2.book_id = 6495700;

   // Print Book1 info, passing address of structure
   printBook( &Book1 );

   // Print Book1 info, passing address of structure
   printBook( &Book2 );

   return 0;
}

// This function accept pointer to structure as parameter.
void printBook( struct Books *book ) {
   cout << "Book title : " << book->title <<endl;
   cout << "Book author : " << book->author <<endl;
   cout << "Book subject : " << book->subject <<endl;
   cout << "Book id : " << book->book_id <<endl;
}
```

When the above code is compiled and executed, it produces the following result –

```
Book title : Learn C++ Programming
Book author : Chand Miyan
Book subject : C++ Programming
Book id : 6495407
Book title : Telecom Billing
Book author : Yakit Singha
Book subject : Telecom
Book id : 6495700
```

1.1.5 The typedef Keyword

There is an easier way to define structs or you could "alias" types you create. For example −

```
typedef struct {
  char  title[50];
  char  author[50];
  char  subject[100];
  int   book_id;
} Books;
```

Now, you can use *Books* directly to define variables of *Books* type without using struct keyword. Following is the example −

```
Books Book1, Book2;
```

You can use **typedef** keyword for non-structs as well as follows −

```
typedef long int *pint32;


pint32 x, y, z;
```

x, y and z are all pointers to long ints.

1.2 Introduction to Pointers in Data Structure

Pointers are the variables that are used to store the location of value present in the memory. A pointer to a location stores its memory address. The process of obtaining the value stored at a location being referenced by a pointer is known as dereferencing. It is the same as the index for a textbook where each page is referred by its page number present in the index. One can easily find the page using the location referred to there. Such pointers usage helps in the dynamic implementation of various data structures such as stack or list.

1.2.1 Why do We Need Pointers in Data Structure?

Optimization of our code and improving the time complexity of one algorithm. Using pointers helps reduce the time needed by an algorithm to copy data from one place to another. Since it used the memory locations directly, any change made to the value will be reflected at all the locations.

Example:

Call_by_value needs the value of arguments to be copied every time any operation needs to be performed.

Call_by_reference makes this task easier using its memory location to update the value at memory locations.

1.2.1.1 *Control Program Flow*

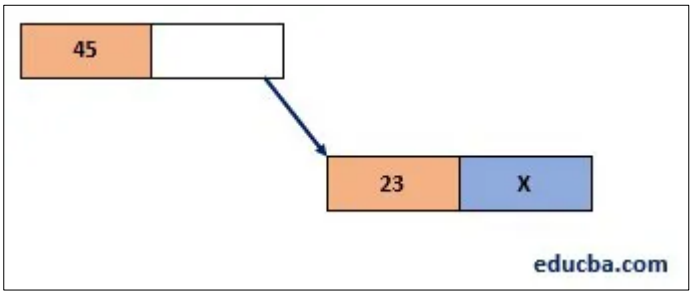
Another use of pointers is to control the program flow. This is implemented by control tables that use these pointers. These pointers are stored in a table to point to each subroutine's entry point to be executed one after the other. These pointers reference the addresses of the various procedures. This helps while working with a recursive procedure or traversal of algorithms where there is a need to store the calling step's location.

1.2.1.2 secondary data structures

secondary data structures such as linked lists or structures to point to the next memory locations in the list.

```
struct Node {
int data;
struct Node* next;
};
```

Example:



1.2.1.3 *Dynamic Memory Allocation*

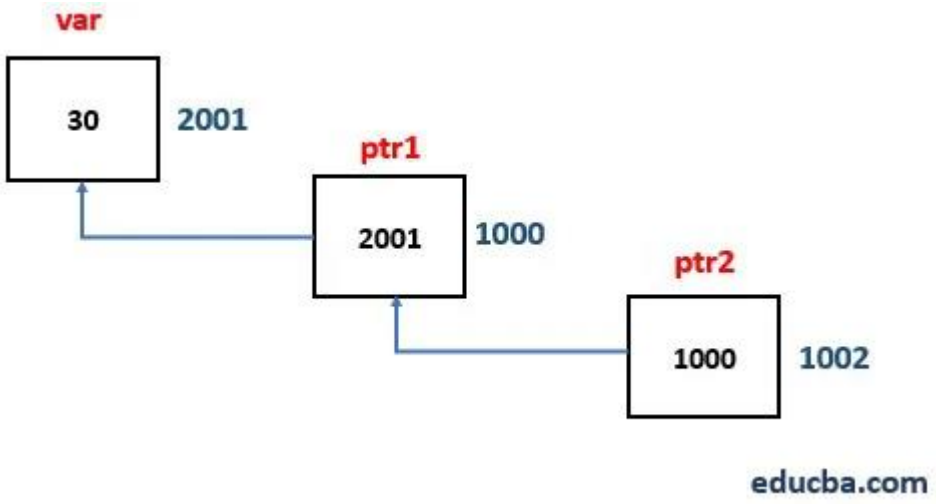
Many programming languages use dynamic memory allocations to allocate the memory for run-time variables. For such type of memory, allocations heap is used rather than the stack, which uses pointers. Here pointers hold the address of these dynamically generated data blocks or array of objects. Many structured or OOPs languages use a heap or free store to provide them with storage locations. The last Node in the linked list is denoted using a NULL pointer that indicates there is no element further in the list.

1.2.2 How do Pointers Work in Data Structure?

Pointers are kind of variables that store the address of a variable.

Defining a Pointer

Here we discuss defining a pointer in the data structure.



Syntax:

```
<datatype> *variable_name
```

Above depicts, vaiable_name is a pointer to a variable of the specified data type.

Example:

```
int *ptr1 – ptr1 references to a memory location
that holds data of int datatype.
int var = 30;
int *ptr1 = &var; // pointer to var
int **ptr2 = & ptr1; // pointer to pointer variable
ptr1
```

In the above example, '&' is used to denote the unary operator, which returns a variable's address.

And '*' is a unary operator that returns the value stored at an address specified by the pointer variable, thus if we need to get the value of variable referenced by a pointer, often called as dereferencing, we use:

```
print("%d", *ptr1) // prints 30

print("%d",**ptr2) // prints 30
```

We need to specify datatype- It helps to identify the number of bytes data stored in a variable; thus. Simultaneously, we increment a pointer variable, and it is incremented according to the size of this datatype only.

C Program on Pointers

Following is an example of creating pointers using C Program.

Example:

```
#include <stdio.h>
void pointerDemo()
{
int var1 = 30;
int *ptr1;
int **ptr2;
ptr1 = &var1;
ptr2 = &ptr1;
printf("Value at ptr1 = %p \n",ptr1);
printf("Value at var1 = %d \n",var1);
printf("Value of variable using *ptr1 = %d \n", *ptr1);
printf("Value at ptr2 = %p \n",ptr2);
printf("Value stored at *ptr2 = %d \n", *ptr2);
printf("Value of variable using  **ptr2 = %d \n", **ptr2);
}
int main()
{
pointerDemo();
return 0;
}
```

Output:

```
Value at ptr1 = 0x7fffc57e1c5c
Value at var1 = 30
Value of variable using *ptr1 = 30
Value at ptr2 = 0x7fffc57e1c60
Value stored at *ptr2 = -981590948
Value of variable using  **ptr2 = 30
```

Explanation: In the above program, we have used single and double dereferencing to display the value of the variable.

There are many types of pointers being **used in computer programming**:

NULL Pointer: Such type of pointer is used to indicate that this points to an invalid object. This type of pointer is often used to represent various conditions such as the end of a list.

VOID Pointer: This type of pointer can be used to point to the address of any type of variable, but the only limitation is that it cannot be dereferenced easily.

WILD Pointer: It is a type of pointer which doesn't hold the address of any variable.

Dangling Pointer: The type of pointers that don't refer to a valid object and are not specifically initialized to point a particular memory. For ex: int *ptr1 = malloc(sizeof(char))

Function pointer: This is a type of pointer to reference an executable code. It is mostly used in the recursive procedure that holds the address of the code that needs to be executed later.

1.2.3 Disadvantage Of Pointers

It can lead to many programming errors as it allows the program to access a variable that has not been defined yet. They can be easily manipulated as the number and made to point some void locations.

Thus to avoid such a situation, many programming languages have started using constructs.

Programming languages such as **JAVA has replaced** the concept of pointers with reference variables which can only be used to refer the address of a variable and cannot be manipulated as a number.

1.3 Dynamic Allocation of 2D Arrays in C++ (with code)

You already know that a 2D array array is an array of arrays. But it is defined at the compiled time. In this article, we learn about dynamically allocating a 2D array in C++ at run time. Let's dive deep to understand the dynamic world!

1.3.1 What is a Dynamic 2D Array in C++?

Can you have dynamic arrays in C++? Yes, C++ provides several ways to create dynamic arrays.

We have 2 types of memory: one which we have already before running the program called stack memory and the other is used for dynamic allocation called heap memory. Stack memory is used at compile time of the C++ program while heap memory is used when we use the 'new' operator somewhere in our program.

So, to create a variable or a data structure dynamically, we must use a 'new' operator. When we deal with the heap memory it always returns an address pointing to a created data structure and then we use a pointer to store that address and to access it from heap memory.

Why use heap memory or dynamic allocation? because heap memory is very efficient as it allows us to use only required memory whereas stack memory first comes and then assigns some part of memory to us.

A dynamic array in C++ is an array that can change its size during runtime. It is just like a normal 2D or multi-dimensional array but it is implemented using pointers and memory allocation functions such as new and delete. The main point is that the size of this array is not fixed at compile time.

These arrays are useful when we don't know the size during compilation or we have to change the size during runtime. But be careful while using them as they require extra careful management to avoid memory leaks.

1.3.2 Methods to Dynamically Allocate a 2D Array

Let's now learn about 3 different ways to dynamically allocate a simple 2D array in C++.

1.3.2.1 Single Pointer Method

In this method, a memory block of size M*N is allocated and then the memory blocks are accessed using pointer arithmetic. Below is the program for the same:

```
// C++ program to dynamically allocate
// the memory for 2D array in C++
// using new operator
#include<iostream>
using namespace std;

// Driver Code
int main()
{
        // Dimensions of the 2D array
        int m = 3, n = 4, c = 0;

        // Declare a memory block of
```

```
        // size m*n
        int* arr = new int[m * n];

        // Traverse the 2D array
        for (int i = 0; i < m; i++) {
                for (int j = 0; j < n; j++) {

                        // Assign values to
                        // the memory block
                        *(arr + i * n + j) = ++c;
                }
        }

        // Traverse the 2D array
        for (int i = 0; i < m; i++) {
                for (int j = 0; j < n; j++) {

                        // Print values of the
                        // memory block
                        cout << *(arr + i * n + j)
                                << " ";
                }
                cout << endl;
        }

        //Delete the array created
        delete[] arr;

        return 0;
}
```

Output:

```
1 2 3 4
5 6 7 8
9 10 11 12
```

First, let's understand this 'new int[m *n]' Here the new operator is used for dynamic allocation and we are creating an array of type int and contiguous allocation of size m * n. This will return us an address as explained above already and stored in a pointer.

Then we are just assigning the values to the address blocks in heap memory using the pointers syntax. 'arr + i * n + j' -> this is the address of the heap block and *(arr + i * n + j) this is the value of that heap block.

1.3.2.2 Using an Array of Pointer

Here an array of pointers is created and then to each memory block. Below is the C++ program to implement it:

```
// C++ program to dynamically allocate
// the memory for 2D array in C++
// using new operator
#include<iostream>
using namespace std;

// Driver Code
int main()
{
	// Dimensions of the array
	int m = 3, n = 4, c = 0;

	// Declare memory block of size M
	int** a = new int*[m];

	for (int i = 0; i < m; i++) {

		// Declare a memory block
		// of size n
		a[i] = new int[n];
	}

	// Traverse the 2D array
	for (int i = 0; i < m; i++) {
		for (int j = 0; j < n; j++) {

			// Assign values to the
			// memory blocks created
			a[i][j] = ++c;
		}
	}

	// Traverse the 2D array
	for (int i = 0; i < m; i++) {
		for (int j = 0; j < n; j++) {

			// Print the values of
			// memory blocks created
			cout << a[i][j] << " ";
		}
		cout << endl;
	}

	// Delete the array created
	for (int i = 0; i < m; i++) // To delete the inner
```

```
                                                        // arrays
            delete[] a[i];
      delete[] a; // To delete the outer array
                        // which contained the pointers
                        // of all the inner arrays

      return 0;
}
```

Output:

```
1 2 3 4
5 6 7 8
9 10 11 12
```

'int a' - > This means we are creating a pointer of type int and this pointer will store the address of an array (in this case) of int type pointers. This array of pointers will point to each row in an array in heap memory.

Practice Problems:

Lab Activity 1:

Write a C++ program of a structure variable to assign data to three members, i-e name, age and salary of a person and display it on screen.

Lab Activity 2: Write a structure to store the roll no., name, age (between 11 and 14) and address of students (more than 10). Store the information of the students.

1 - Write a function to print the names of all the students having age 14.

2 - Write another function to print the names of all the students having even roll no.

3 - Write another function to display the details of the student whose roll no is given (i.e. roll no. entered by the user).

Lab Activity 3:

In C/C++ language, arrays work on memory addresses so as pointers and hence they are strongly related. Design a C program which take an array of 6 elements from the user :and do the following

a) Declare the pointer and refernce it with the array address
b) Displayevery element of an array by using the pointer
c) Find the maximum value in arrayusi n g pointer

```
Enter values into an array:
6
89
56
3
12
5
Display array:
6 89 56 3 12 5
Maximum value is: 89
--------------------------------
Process exited after 11.56 seconds with return value 0
Press any key to continue . . .
```

Lab Activity 4:

Learning Objective: Pointers in function call

In this activity you will have to use pointers to perform the following tasks on randomly generated 1D array of type **int** having size of 20.

i. Write a function **int smallest = smallest_value(int *temp, int size)**, which receives an array as pointer and find out smallest value using pointer to array.

ii. Write a function **void traversal_back(int *temp, int sizeofarry)**, which receives an array as pointer and traverse array from last index to first index and print its value with its address as well.

Lab#02
Abstract Data Types
Single Linked list

Prepared by: Engr. Amna Arooj

FCSE, GIKI

Lab#02
Abstract Data types and Single Linked List

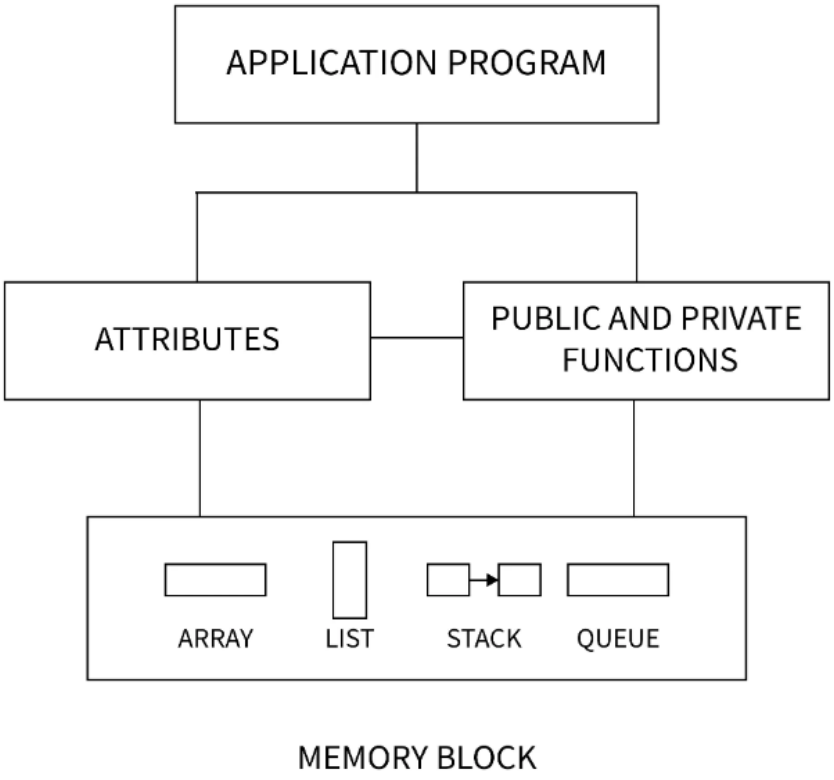
2.1 Introduction to ADT

Data types are used to define or classify the type of values a variable can store in it. Moreover, it also describes the possible operations allowed on those values. For example, the integer data type can store an integer value. Possible operations on an integer include addition, subtraction, multiplication, modulo.

Abstract data type (ADT) is a concept or model of a data type. An Abstract Data Type in data structure is a kind of a data type whose behavior is defined with the help of some attributes and some functions. Generally, we write these attributes and functions inside a class or a structure so that we can use an object of the class to use that particular abstract data type.

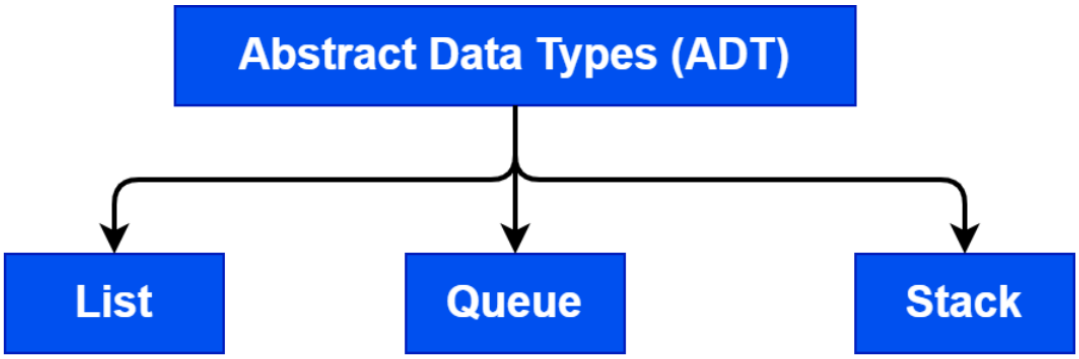
Examples of Abstract Data Type in Data Structure are list, stack, queue etc.

2.1.1 Abstract Data Type Model



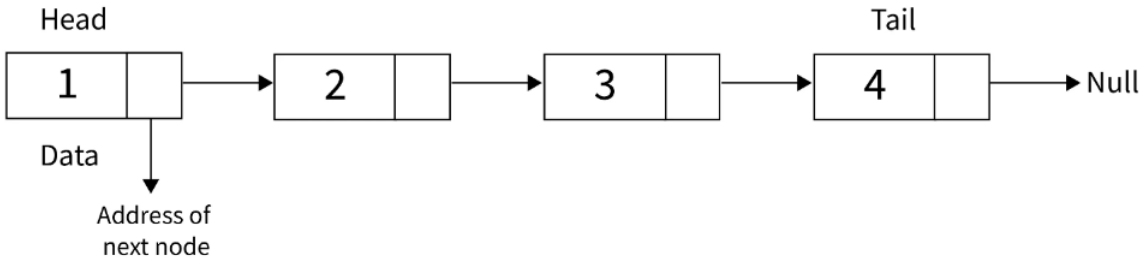
In every programming language, we implement ADTs using different methods and logic. Although, we can still perform all the associated operations which are defined for that ADT irrespective of language. For example, in C, ADTs are implemented mostly using structure. On the other hand, in C++ or JAVA, they're implemented using class. **However, operations are common in all languages.**

There are mainly three types of ADTs:



2.2 Introduction to List

A Linked List is a linear data structure consisting of connected nodes where each node has corresponding data and a pointer to the address of the next node. The first node of a linked list is called the **Head**, and it acts as an access point. On the other hand, the last node is called the **Tail**, and it marks the end of a linked list by pointing to a *NULL* value!



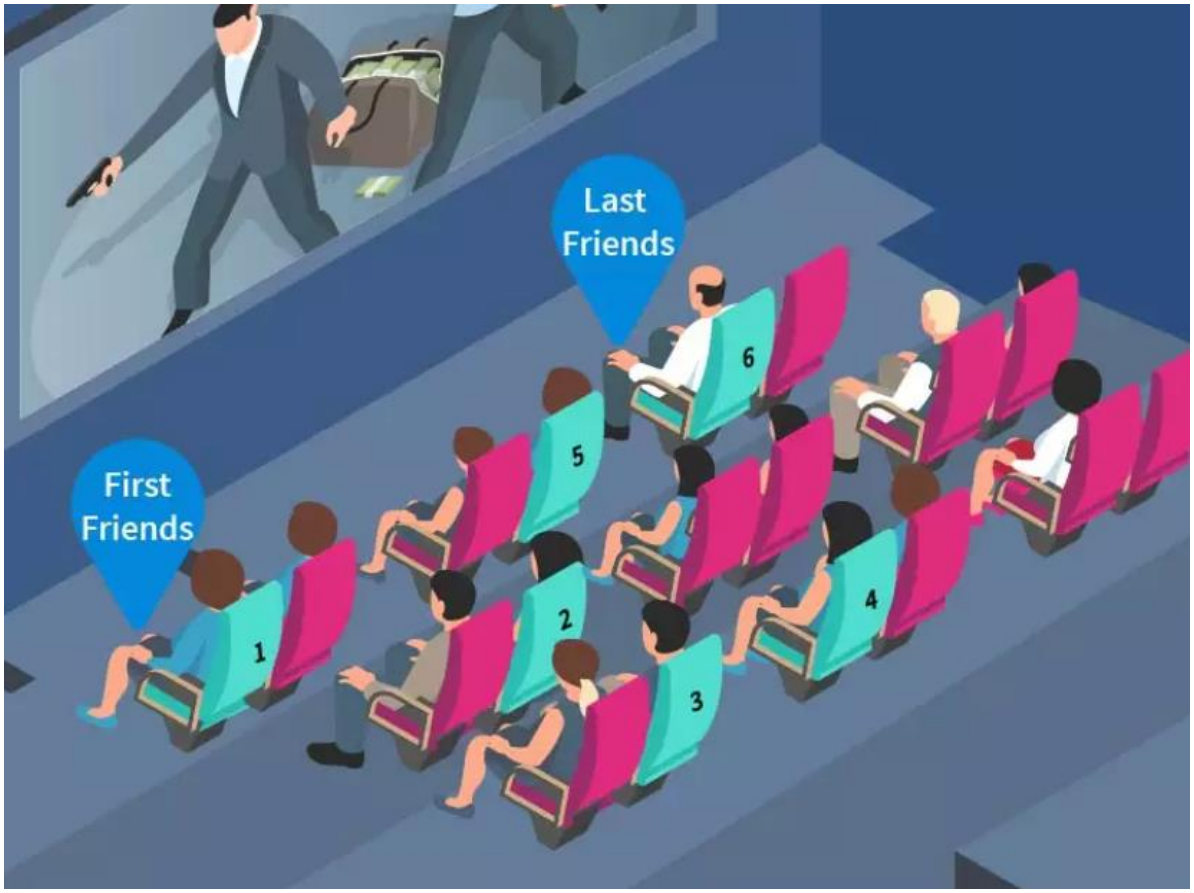
2.2.1 Why Do We Need a Linked List?

Imagine going to a movie theater along with a large group of friends, only to find out there's no way to book consecutive seats to accommodate all. However, there are plenty of disjoint empty seats for that particular screen. So, you all buy tickets and sit accordingly.

But there's a catch!

Since only you were buying popcorn for all, how would you distribute that to your friends? You can't remember where everyone sat, and it's dark inside the hall.

So, instead of you remembering where everyone sat, if everyone kept track of the next friend's seat, then such reference would help popcorn tubs to traverse up to a friend who's yet to get popcorn. This could be achieved by passing popcorn to the next friend starting from you!



See how the ***linked list*** came in handy while keeping track of all these randomly distributed disjoint seats?

In a computer, you can think of memory as a movie theatre, the virtual address space as the seats, and any particular memory address as a seat number.

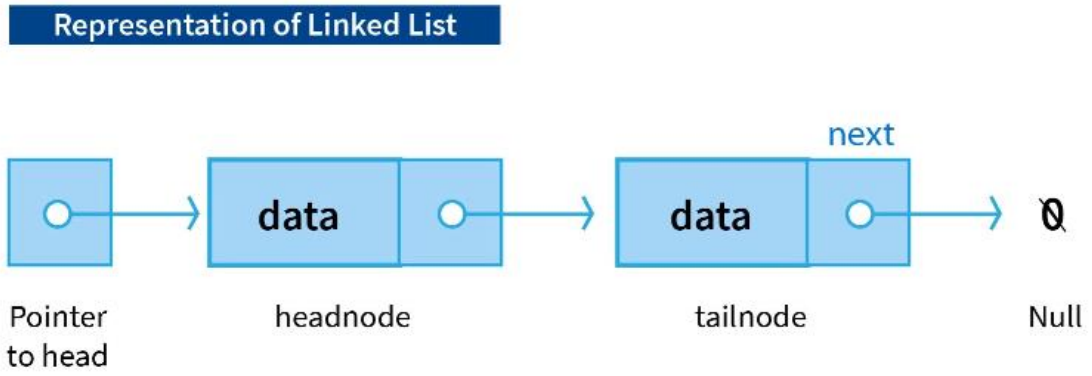
Now, if we had initiated booking in advance, we could have booked an entire row of consecutive seats much like an *array* does when we declare it with a fixed size in advance, like so:

int seats[24];

In our context of a movie theater, if more friends later decide to join the group to catch the movie. We cannot simply append them to the array of seats since there might be other audiences seating just after the last allocated seat.

So either we find another larger empty row of consecutive seats so that all friends can sit together (much like *dynamic array*), or we rely on an approach similar to the **linked list**, in that we simply insert new friends in vacant spots but still have references to them should we require to pass popcorn and drinks.

You see, arrays are suitable when we need fast access. Like who's seating at seat no. 13 can be answered in constant time! But arrays require you to declare a fixed size at the compile-time, due to which memory can be either wasted or fell short.



Whereas linked list shines when we need to modify existing data by insertion and deletion because it doesn't have a fixed size. So, our memory consumption is determined at run time as the linked list shrinks and grows dynamically in constant time.

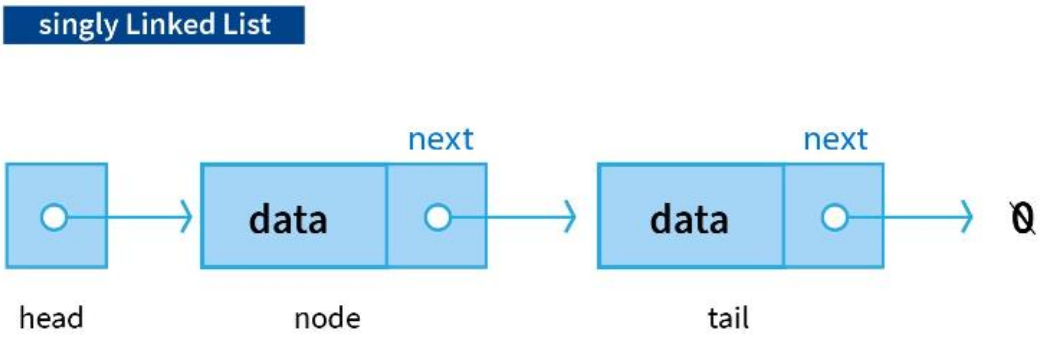
2.2.2 Types of Linked List

There are mainly three types of linked lists:

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

2.2.2.1 Singly Linked List

Here, we can only traverse in *one direction* (not the band) due to the linking of every node to its next node.



2.2.2.2How Can We Declare a Linked List?

A node which is both head and tail and in and of itself a linked list!

```
class Node {
        some_constructor(int value) {
                int data = value;
                Node* next = NULL;
        }
```

```
}
```

where data can be any *valid type* and not just *integer*, but next must be a *Node pointer*.

Let's declare the **head** of our linked list like so:

```
Node* head = new Node(77);
```

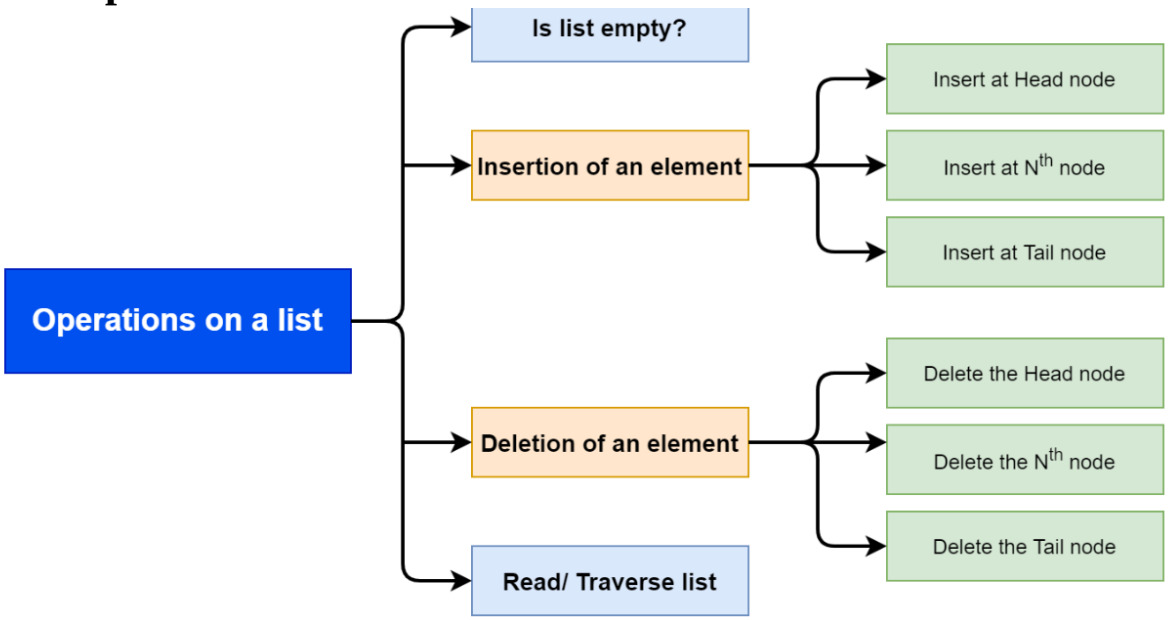
And that's it for a single node of a singly linked list

"But how can we extend it for a linked list of size k?"

```
Node* head = new Node(num[0]);
Node* curr = head;
for itr -> 1 to (k - 1) {
        curr.next = new Node(num[itr]);
        curr = curr.next;
```

where num could be an array of integers storing the data for building our respective linked list and itr could be any variable iterating over the indices of the said array.

2.2.3 Operations on Linked List



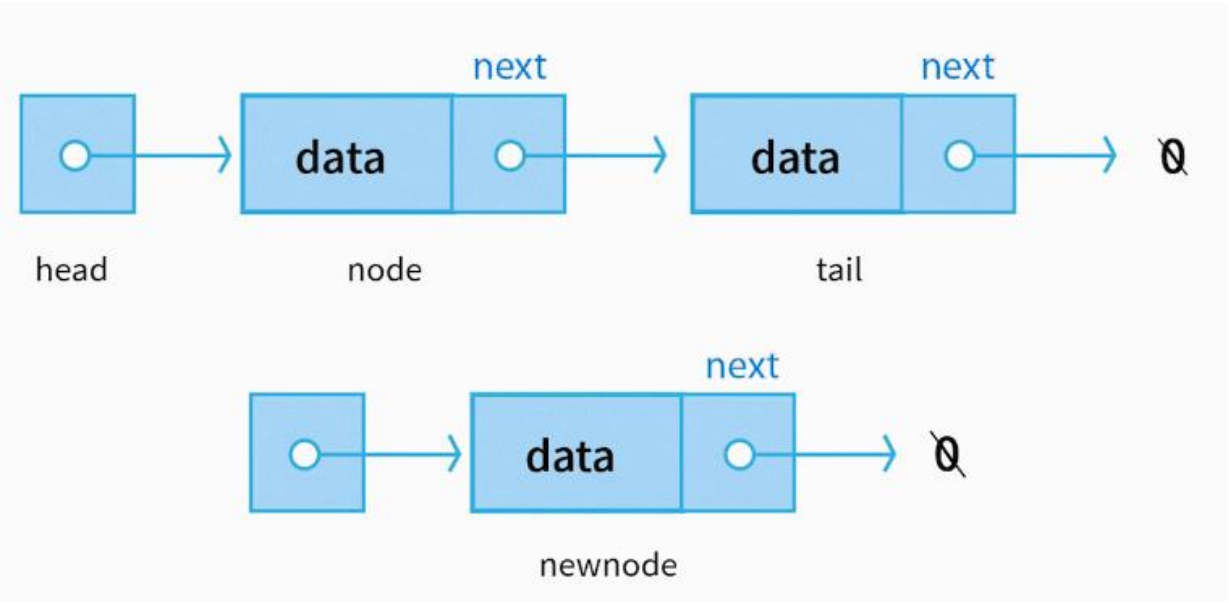
Some of the most essential operations defined in List ADT are listed below.

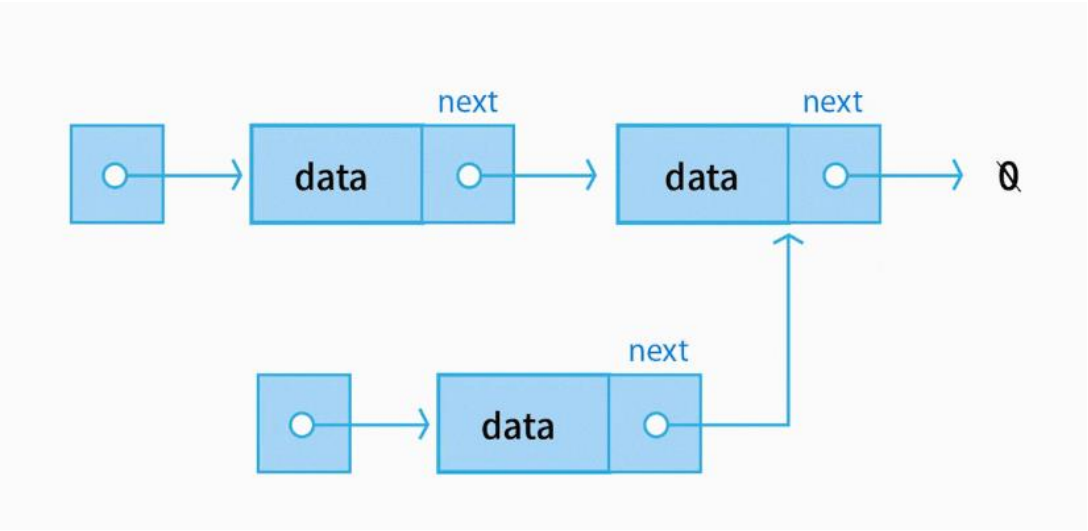
- **front():** returns the value of the node present at the front of the list.
- **back():** returns the value of the node present at the back of the list.
- **push_front(int val):** creates a pointer with value = val and keeps this pointer to the front of the linked list.
- **push_back(int val):** creates a pointer with value = val and keeps this pointer to the back of the linked list.
- **pop_front():** removes the front node from the list.
- **pop_back():** removes the last node from the list.
- **empty():** returns true if the list is empty, otherwise returns false.
- **size():** returns the number of nodes that are present in the list.

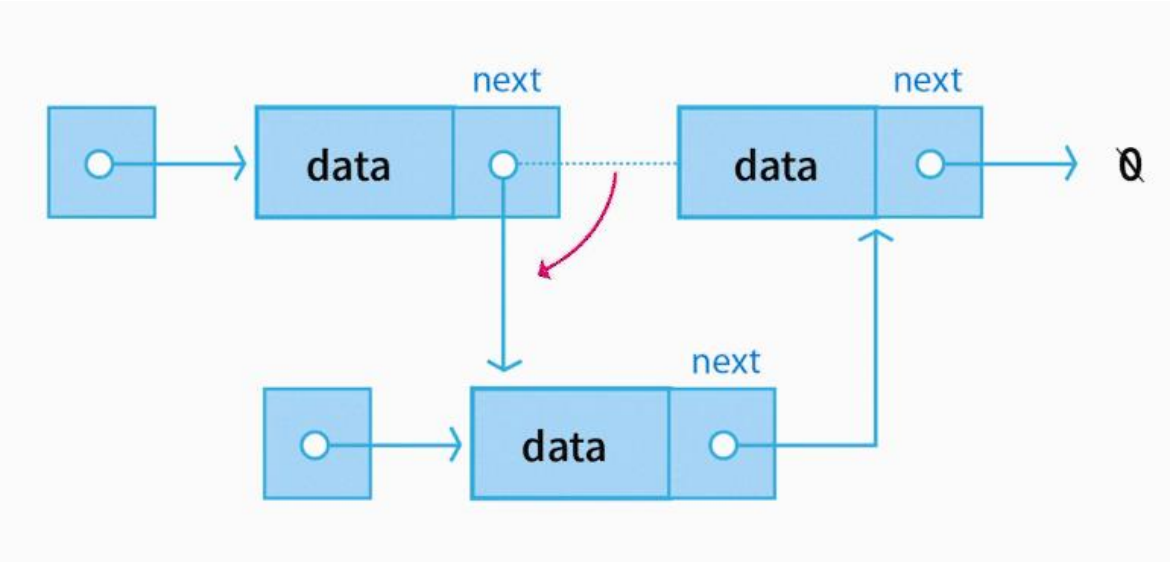
2.2.3.1 Insertion

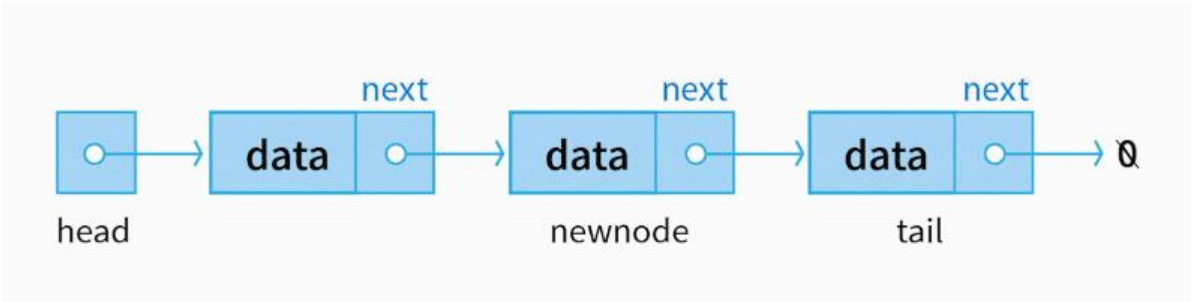
Inserting a new node at a given position is achieved by manipulating at most 2 next pointers like so; consider two pointers, prevNode, and newNode set newNode's next pointer to the prevNode's next pointer and set prevNode's next pointer to the newNode where prevNode denotes a pointer to the node after which newNode is to be inserted.
And since only a constant number of pointers gets changed irrespective of the size of the linked list, the complexity should have been O(1).

However, to get access at any given position, we have to traverse starting from the head till prevNode. Thus, **the worst case** for insertion could be expressed as; Traversal O(n) + Just the insertion O(1) = Insertion O(n). However, **the best case** would be to insert in the beginning and hence the time complexity becomes Ω(1).
Notice that the order of execution does matter in case of insertion. If we interchange the steps i.e. first set prevNode's next pointer to the newNode then we lose the reference to the remaining of the linked list previously occurring after the prevNode. Something to keep in mind!


next
next
data
data
head
node
tail
next
data
newnode


next
next
data
data
next
data

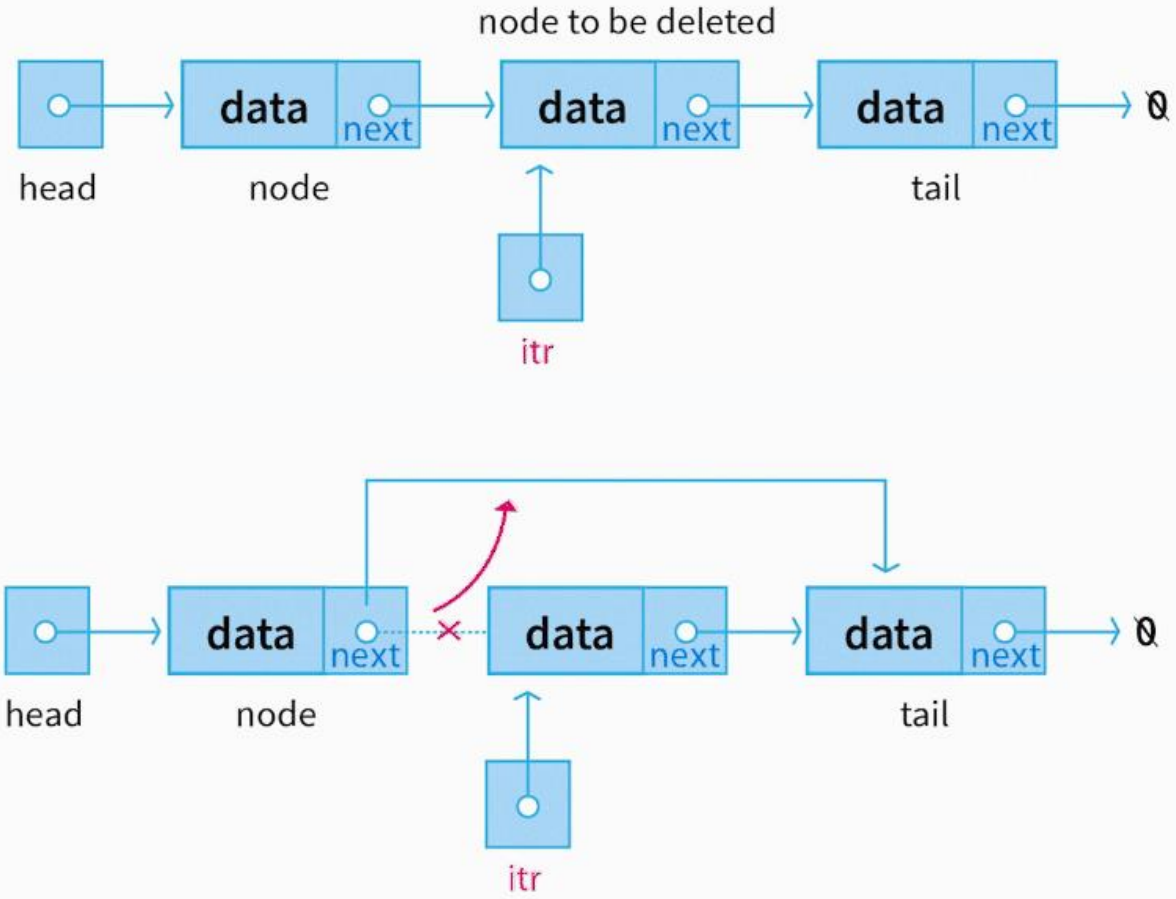

next
next
data
data
next
data

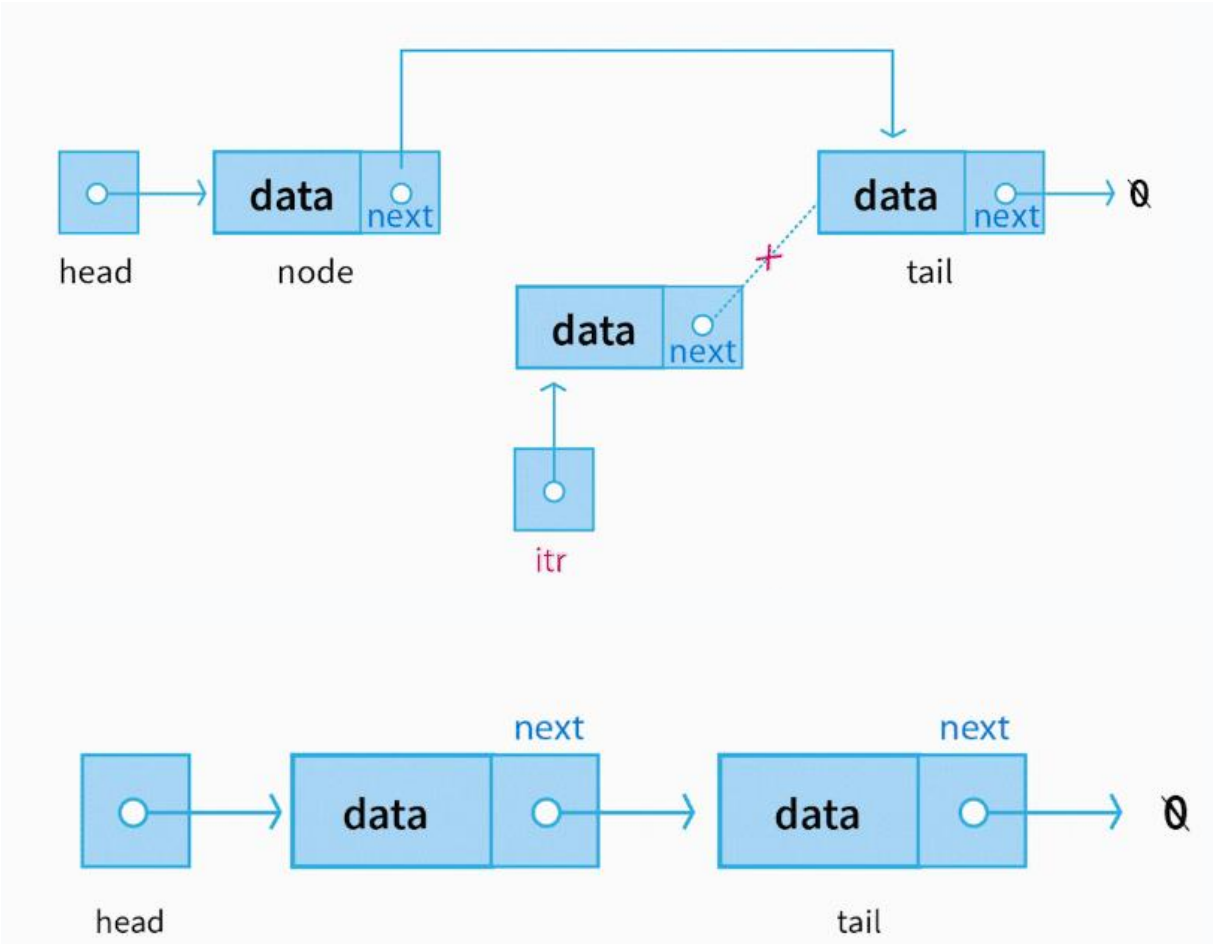


2.2.3.2 Deletion

We can delete a node located at a specified index by manipulating at most 2 next pointers like so; consider two pointers, prevNode, and targetNode set prevNode's next pointer to the targetNode's next pointer and set targetNode's next pointer to NULL where prevNode denotes a pointer to the node after which targetNode is to be deleted. Following the same reasoning as insertion, the best case and the worst case time complexity stands at Ω(1) and O(n) respectively.

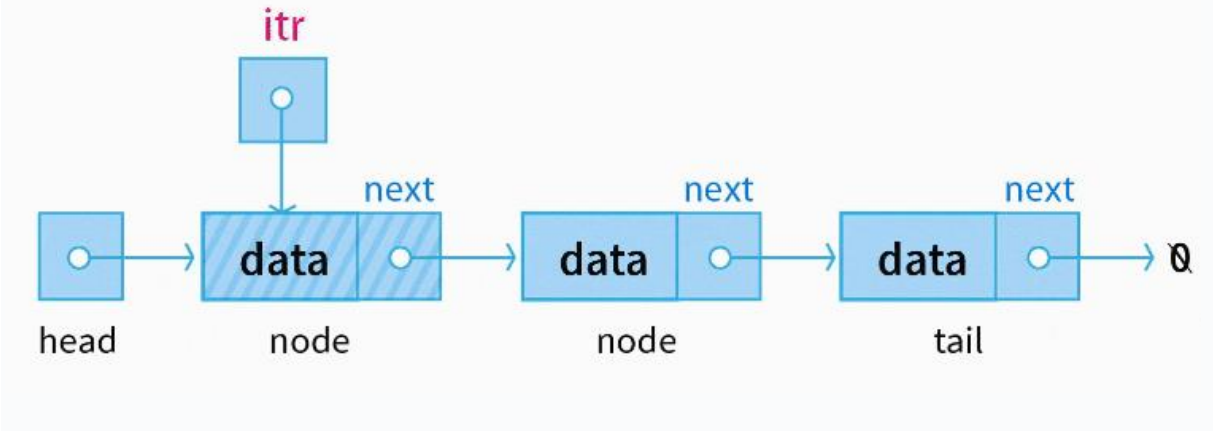
Again just like insertion, the order is important. It's important to realize that setting targetNode's next pointer to NULL as the first step would cost us the reference to the remaining of the linked list (if the targetNode wasn't the *Tail*).

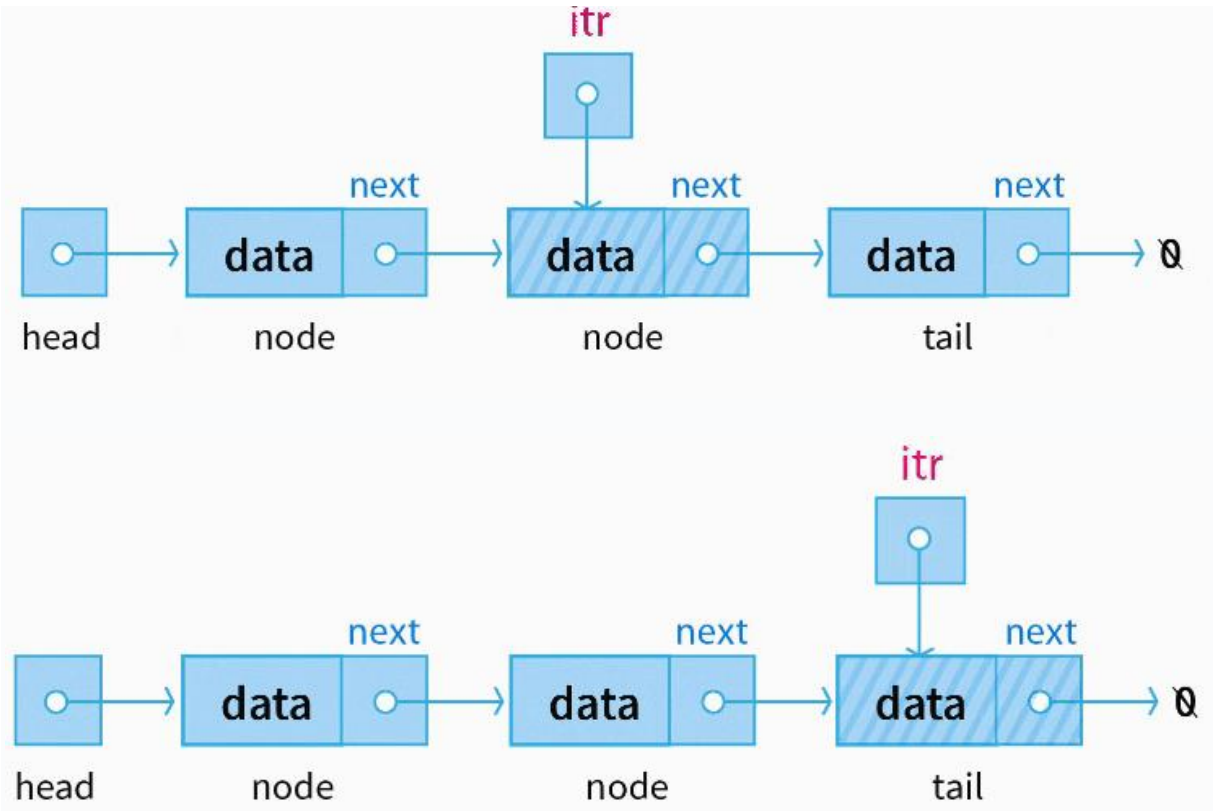




2.2.3.3 Traversal

We can traverse the entire linked list starting from the head node. If there are n nodes then the time complexity for traversal becomes O(n) as we hop through each and every node.





2.2.3.4 Search

Given a particular **key** (data), we need to search for a node whose data matches the key. In the best-case scenario, the head would be the node we are looking for, whereas in the worst-case the required node would be the tail. In terms of time complexity the best case, the average case, and the worst case of searching are denoted as Ω(1), Θ(n), and O(n) respectively.

2.2.4 Implementation of Linked List in C++

```
struct Node {
    // Blueprint for a node of a linked list
    int data;
    Node *next;
    Node() : data(0), next(nullptr) {}
    Node(int x) : data(x), next(nullptr) {}
    Node(int x, Node *next) : data(x), next(next) {}
};

class LinkedList {
public:
    // constructor for creating a linked list
    // and storing it in 'head'
    LinkedList() {}

    void traversal() {
```

```
        Node* itr = head;
        while (itr != nullptr) {
            cout << itr -> data << endl;
            itr = itr -> next;
        }
    }

    // 'newNode' - pointer to the node to be inserted
    // 'prevNode' - pointer to the node after which insertion occurs
    void insertion(Node* prevNode, Node* newNode) {
        newNode -> next = prevNode -> next;
        prevNode -> next = newNode;
    }

    // 'targetNode' - pointer to the node to be deleted
    // 'prevNode' - pointer to the node after which deletion occurs
    void deletion(Node* prevNode, Node* targetNode) {
        prevNode -> next = targetNode -> next;
        targetNode -> next = null;
    }

    bool search(int key) {
        Node* itr = head;
        while (itr != null) {
            if (itr -> data == key)
                return true;
            itr = itr -> next;
        }
        return false; // key not found!
    }
}
```

2.2.5 Time Complexity of Linked List

As you can clearly see that few linked list operations are more efficient than others. To understand, let's observe the dichotomy of linked lists and arrays in the context of time complexity for the aforementioned operations.

Operation	linked list	Array
Random access	O (n)	O (1)
Insertion and Deletion at the beginning	O (1)	O (n)
Insertion and Deletion at the end	O (n)	O (1)
Insertion and Deletion from random location	O (n)	O (n)

Time Complexity (Big O) Comparisons

2.2.6 Advantages / Disadvantages of Linked List

Dynamic in nature - Linked lists are dynamic in nature, and their size can be adjusted according to our requirements.

Ease of Insertion and Deletion - Insertion and deletion of a node in a linked list remain efficient as the nodes are stored in random locations, and we only need to update the next pointer of a constant number of nodes.

Memory efficient - Memory consumption of a linked list is efficient as its size can grow or shrink dynamically according to our requirements.

Implementation - Various advanced data structures can be implemented using a linked list vis-a-vis *stack*, *queue*, *graph*, *hash maps*, etc.

Disadvantages of Linked List

Memory usage - A node in a linked list occupies more memory than an element in an array as each node occupies at least two kinds of variables.

Accessing a node - If you want to access a node in a linked list, you have to traverse starting from the head. We cannot access any random nodes directly except for the head itself, since nodes don't share a linear order in the physical memory and are only referenced by pointers.

Traversing in reverse order - Traversing in reverse order is difficult. Although a doubly-linked list makes it easier but requires more memory to store those extra 'prev' pointers.

2.2.7 Applications of Linked List

It would have been futile if a linked list didn't have many applications beyond a mere movie theater use-case.

Here are some of the applications of a linked list:

- Linear data structures such as stack, queue, and non-linear data structures such as hash maps, graphs can be implemented using linked lists.
- Doubly linked lists are useful for building a forward-backward navigation button in a web browser or undo-redo feature in an editor.
- Circular Doubly linked lists can also be used for the implementation of advanced data structures like Fibonacci Heap.

Practice Problems:

Lab Activity 1:

Write a C++ function to count the number of nodes in a singly linked list.

You can take the following structure as a reference:

```
struct Node {
    int data;
    Node* next;
};
```

Your task is to implement the function with the following prototype:
int countNodes(Node* head);
The function should take the head of a singly linked list as an argument and return the total number of nodes present in the list.

Lab Activity 2:

Write a C++ function to check if a given singly linked list is empty. You can use question#01 structure as reference.

Your task is to implement the function with the following prototype:

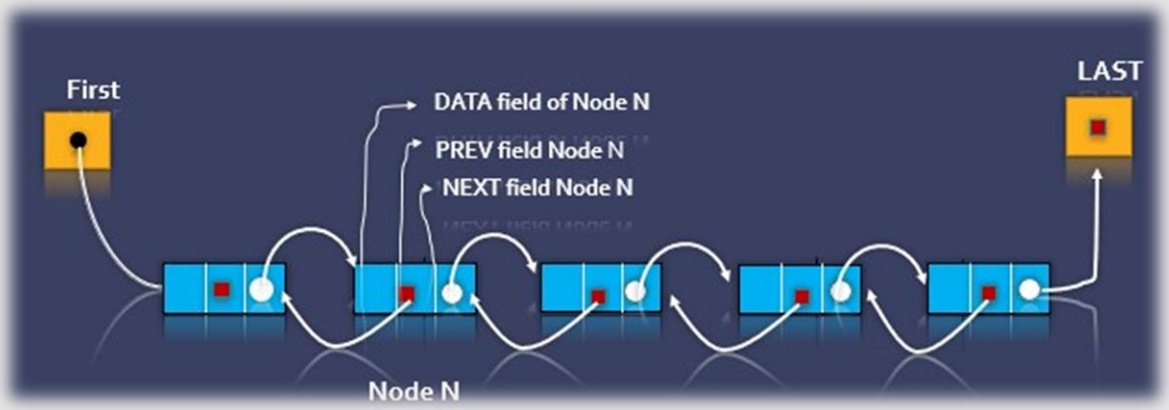
bool isListEmpty(Node* head);

The function should take the head of a singly linked list as an argument and return `true` if the list is empty (i.e., it has no nodes), and `false` otherwise.

Lab Activity 3:

Write a C++ function to delete a node with a given value from a singly linked list. Assume that the value to be deleted is always present in the list.

Lab#03
Double Linked List



Prepared by: Engr. Amna Arooj

FCSE, GIKI

Lab 03
Doubly Linked List

3.1 Doubly Linked List

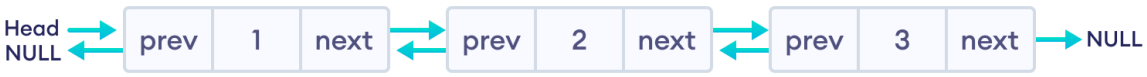
A doubly linked list is a type of linked list in which each node consists of 3 components:

- `*prev` - address of the previous node
- `data` - data item
- `*next` - address of next node

A doubly linked list node.

3.2 Representation of Doubly Linked List

Let's see how we can represent a doubly linked list on an algorithm/code. Suppose we have a doubly linked list:



Newly created doubly linked list

Here, the single node is represented as

```
struct node {
    int data;
    struct node *next;
    struct node *prev;
}
```

Each struct node has a data item, a pointer to the previous struct node, and a pointer to the next struct node.

Now we will create a simple doubly linked list with three items to understand how this works.

```
/* Initialize nodes */
struct node *head;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;

/* Allocate memory */
one = malloc(sizeof(struct node));
two = malloc(sizeof(struct node));
three = malloc(sizeof(struct node));

/* Assign data values */
one->data = 1;
two->data = 2;
three->data = 3;

/* Connect nodes */
one->next = two;
one->prev = NULL;

two->next = three;
two->prev = one;

three->next = NULL;
three->prev = two;

/* Save address of first node in head */
head = one;
```

In the above code, `one`, `two`, and `three` are the nodes with data items **1**, **2**, and **3** respectively.

For node one: `next` stores the address of `two` and `prev` stores `null` (there is no node before it)

For node two: `next` stores the address of `three` and `prev` stores the address of `one`

For node three: `next` stores `null` (there is no node after it) and `prev` stores the address of `two`.

Note: In the case of the head node, `prev` points to `null`, and in the case of the tail pointer, `next` points to null. Here, `one` is a head node and `three` is a tail node.

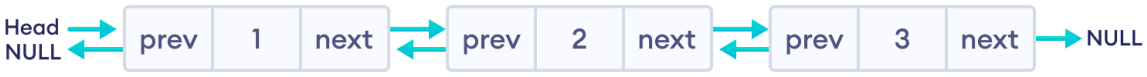
3.3 Insertion on a Doubly Linked List

Pushing a node to a doubly-linked list is similar to pushing a node to a linked list, but extra work is required to handle the pointer to the previous node.

We can insert elements at 3 different positions of a doubly-linked list:

- Insertion at the beginning
- Insertion in-between nodes
- Insertion at the End

Suppose we have a double-linked list with elements **1**, **2**, and **3**.



Original doubly linked list

3.3.1 Insertion at the Beginning

Let's add a node with value **6** at the beginning of the doubly linked list we made above.

1. Create a new node

allocate memory for `newNode`

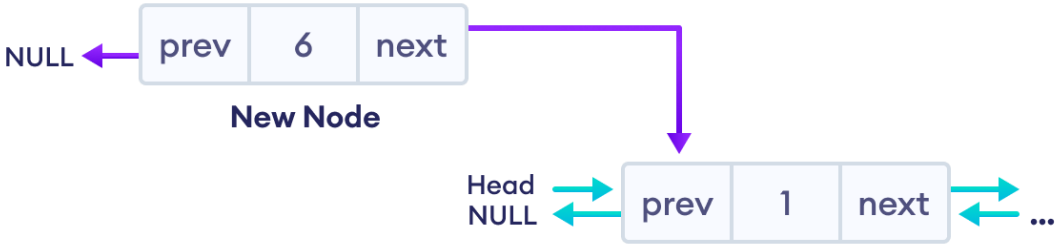
assign the data to `newNode`.

New node

2. Set prev and next pointers of new node

point `next` of `newNode` to the first node of the doubly linked list

point `prev` to `null`

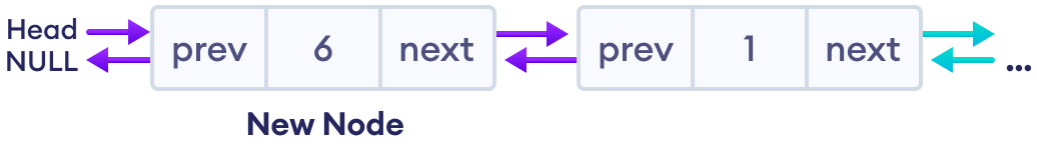


Reorganize the pointers (changes are denoted by purple arrows)

3. Make new node as head node

Point `prev` of the first node to `newNode` (now the previous `head` is the second node)

Point `head` to `newNode`



Reorganize the pointers

Code for Insertion at the Beginning

```
// insert node at the front
void insertFront(struct Node** head, int data) {


   // allocate memory for newNode
```

```
    struct Node* newNode = new Node;

    // assign data to newNode
    newNode->data = data;

    // point next of newNode to the first node of the doubly linked list
    newNode->next = (*head);

    // point prev to NULL
    newNode->prev = NULL;

    // point previous of the first node (now first node is the second node) to newNode
    if ((*head) != NULL)
        (*head)->prev = newNode;

    // head points to newNode
    (*head) = newNode;
}
```

3.3.2 Insertion in between two nodes

Let's add a node with value 6 after node with value 1 in the doubly linked list.

1. Create a new node

allocate memory for `newNode`

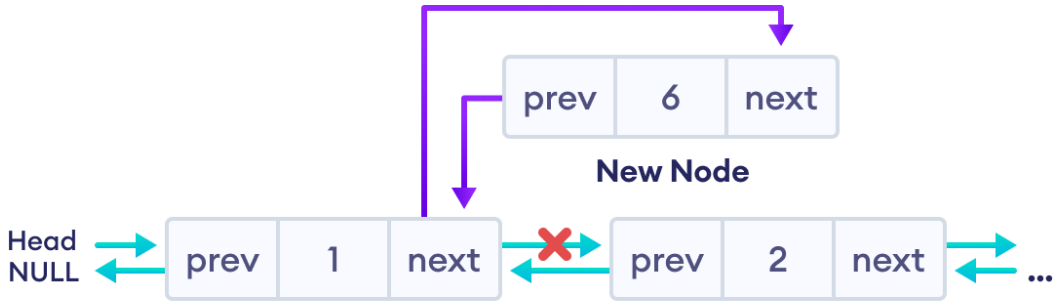
assign the data to `newNode`.

New node

2. Set the next pointer of new node and previous node

assign the value of `next` from previous node to the `next` of `newNode`

assign the address of `newNode` to the `next` of previous node

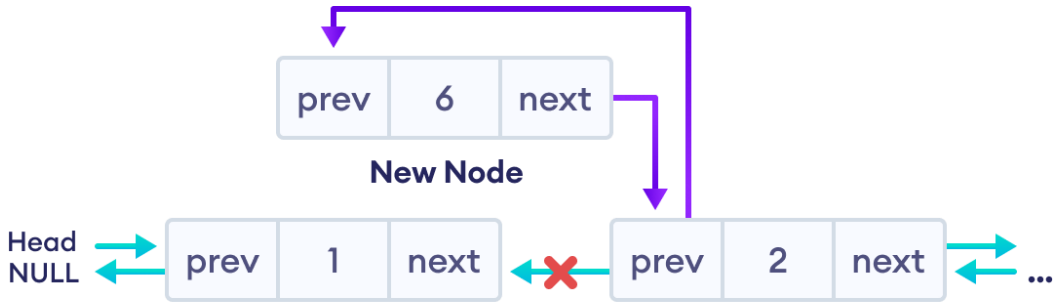


Reorganize the pointers

3. Set the prev pointer of new node and the next node

assign the value of `prev` of next node to the `prev` of `newNode`

assign the address of `newNode` to the `prev` of next node



Reorganize the pointers

The final doubly linked list is after this insertion is:

Final list

Code for Insertion in between two Nodes

```
// insert a node after a specific node
```

```
void insertAfter(struct Node* prev_node, int data) {

    // check if previous node is NULL
    if (prev_node == NULL) {
        cout << "previous node cannot be NULL";
        return;
    }

    // allocate memory for newNode
    struct Node* newNode = new Node;

    // assign data to newNode
    newNode->data = data;

    // set next of newNode to next of prev node
    newNode->next = prev_node->next;

    // set next of prev node to newNode
    prev_node->next = newNode;

    // set prev of newNode to the previous node
    newNode->prev = prev_node;

    // set prev of newNode's next to newNode
    if (newNode->next != NULL)
        newNode->next->prev = newNode;
}
```

3.3.3. Insertion at the End

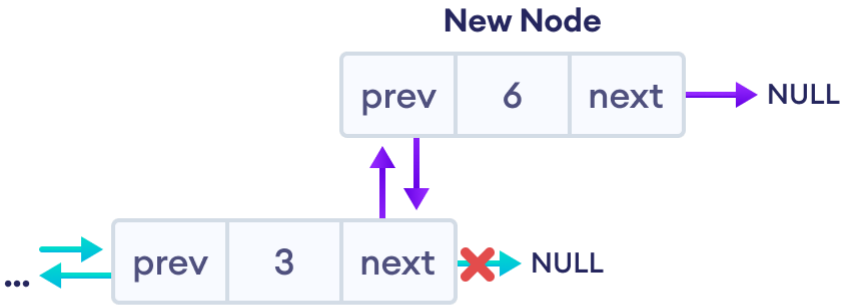
Let's add a node with value 6 at the end of the doubly linked list.

1. Create a new node

New node

2. Set prev and next pointers of new node and the previous node

If the linked list is empty, make the `newNode` as the head node. Otherwise, traverse to the end of the doubly linked list and



Reorganize the pointers

The final doubly linked list looks like this.

The final list

Code for Insertion at the End

```
// insert a newNode at the end of the list

void insertEnd(struct Node** head, int data) {
```

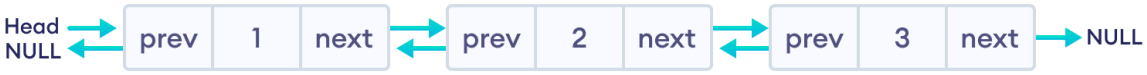
```
// allocate memory for node
struct Node* newNode = new Node;

// assign data to newNode
newNode->data = data;

// assign NULL to next of newNode
newNode->next = NULL;

// store the head node temporarily (for later use)
struct Node* temp = *head;

// if the linked list is empty, make the newNode as head node
if (*head == NULL) {
    newNode->prev = NULL;
    *head = newNode;
    return;
}

// if the linked list is not empty, traverse to the end of the linked list
while (temp->next != NULL)
    temp = temp->next;

// now, the last node of the linked list is temp

// point the next of the last node (temp) to newNode.
temp->next = newNode;

// assign prev of newNode to temp
newNode->prev = temp;
```

```
}
```

3.4 Deletion from a Doubly Linked List

Similar to insertion, we can also delete a node from **3** different positions of a doubly linked list.

Suppose we have a double-linked list with elements **1**, **2**, and **3**.

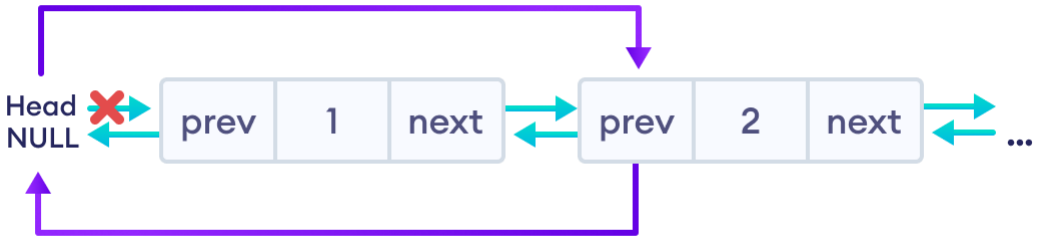


Original doubly linked list

3.4.1 Delete the First Node of Doubly Linked List

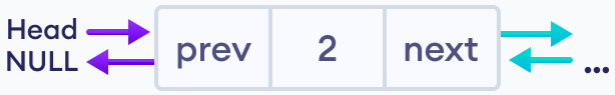
If the node to be deleted (i.e. `del_node`) is at the beginning

Reset value node after the del_node (i.e. node two)



Reorganize the pointers

Finally, free the memory of `del_node`. And, the linked will look like this



Free the space of the first node

Final list

Code for Deletion of the First Node

```
if (*head == del_node)
    *head = del_node->next;

if (del_node->prev != NULL)
    del_node->prev->next = del_node->next;

free(del);
```

3.4.2 Deletion of the Inner Node

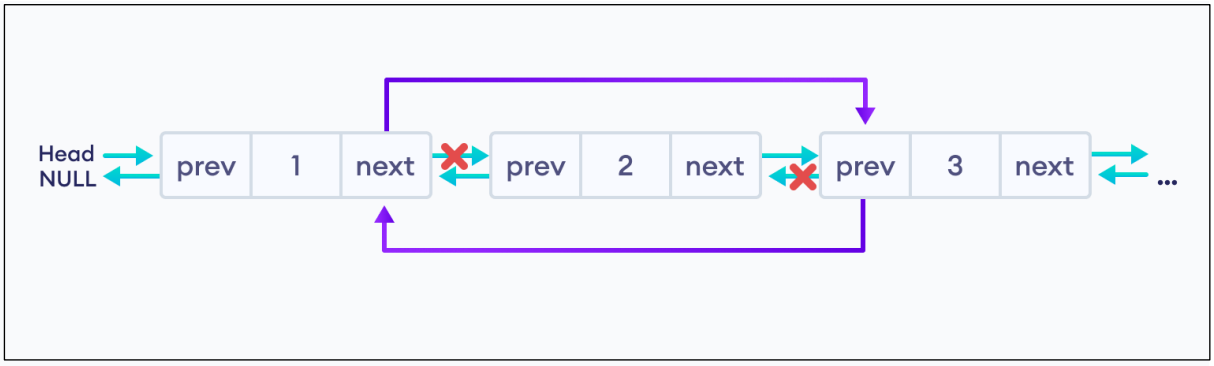
If `del_node` is an inner node (second node), we must have to reset the value of `next` and `prev` of the nodes before and after the `del_node`.

For the node before the del_node (i.e. first node)

Assign the value of `next` of `del_node` to the `next` of the `first` node.

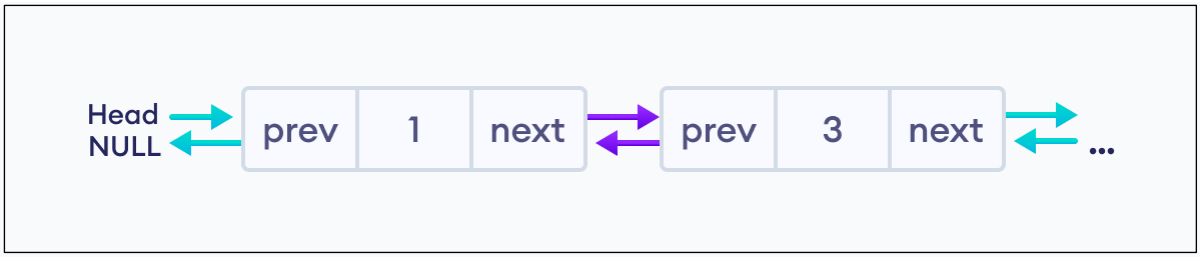
For the node after the del_node (i.e. third node)

Assign the value of `prev` of `del_node` to the `prev` of the `third` node.



Reorganize the pointers

Finally, we will free the memory of `del_node`. And, the final doubly linked list looks like this.



Final list

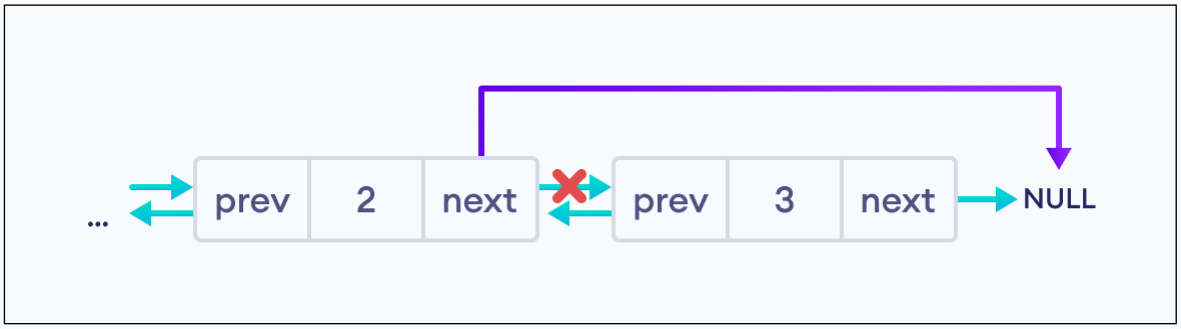
Code for Deletion of the Inner Node

```
if (del_node->next != NULL)
    del_node->next->prev = del_node->prev;


if (del_node->prev != NULL)
    del_node->prev->next = del_node->next;
```

3.3.3 Delete the Last Node of Doubly Linked List

In this case, we are deleting the last node with value **3** of the doubly linked list.

Here, we can simply delete the `del_node` and make the `next` of node before `del_node` point to `NULL`.



Reorganize the pointers

The final doubly linked list looks like this.

Final list

Code for Deletion of the Last Node

```
if (del_node->prev != NULL)
    del_node->prev->next = del_node->next;
```

Here, `del_node ->next` is `NULL` so `del_node->prev->next = NULL`.

Note: We can also solve this using the first condition (for the node before `del_node`) of the second case (Delete the inner node).

3.4 Doubly Linked List Code in C++

```
#include <iostream>
using namespace std;

// node creation
struct Node {
  int data;
  struct Node* next;
  struct Node* prev;
};

// insert node at the front
void insertFront(struct Node** head, int data) {
  // allocate memory for newNode
  struct Node* newNode = new Node;

  // assign data to newNode
  newNode->data = data;

  // make newNode as a head
  newNode->next = (*head);

  // assign null to prev
  newNode->prev = NULL;
```

```
  // previous of head (now head is the second node) is newNode
  if ((*head) != NULL)
    (*head)->prev = newNode;

  // head points to newNode
  (*head) = newNode;
}

// insert a node after a specific node
void insertAfter(struct Node* prev_node, int data) {
  // check if previous node is null
  if (prev_node == NULL) {
    cout << "previous node cannot be null";
    return;
  }

  // allocate memory for newNode
  struct Node* newNode = new Node;

  // assign data to newNode
  newNode->data = data;

  // set next of newNode to next of prev node
  newNode->next = prev_node->next;

  // set next of prev node to newNode
  prev_node->next = newNode;

  // set prev of newNode to the previous node
```

```
  newNode->prev = prev_node;

  // set prev of newNode's next to newNode
  if (newNode->next != NULL)
    newNode->next->prev = newNode;
}

// insert a newNode at the end of the list
void insertEnd(struct Node** head, int data) {
  // allocate memory for node
  struct Node* newNode = new Node;

  // assign data to newNode
  newNode->data = data;

  // assign null to next of newNode
  newNode->next = NULL;

  // store the head node temporarily (for later use)
  struct Node* temp = *head;

  // if the linked list is empty, make the newNode as head node
  if (*head == NULL) {
    newNode->prev = NULL;
    *head = newNode;
    return;
  }

  // if the linked list is not empty, traverse to the end of the linked list
  while (temp->next != NULL)
```

```
    temp = temp->next;

  // now, the last node of the linked list is temp

  // assign next of the last node (temp) to newNode
  temp->next = newNode;

  // assign prev of newNode to temp
  newNode->prev = temp;
}

// delete a node from the doubly linked list
void deleteNode(struct Node** head, struct Node* del_node) {
  // if head or del is null, deletion is not possible
  if (*head == NULL || del_node == NULL)
    return;

  // if del_node is the head node, point the head pointer to the next of del_node
  if (*head == del_node)
    *head = del_node->next;

  // if del_node is not at the last node, point the prev of node next to del_node to the previous
of del_node
  if (del_node->next != NULL)
    del_node->next->prev = del_node->prev;

  // if del_node is not the first node, point the next of the previous node to the next node of
del_node
  if (del_node->prev != NULL)
    del_node->prev->next = del_node->next;
```

```
  // free the memory of del_node
  free(del_node);
}

// print the doubly linked list
void displayList(struct Node* node) {
  struct Node* last;

  while (node != NULL) {
    cout << node->data << "->";
    last = node;
    node = node->next;
  }
  if (node == NULL)
    cout << "NULL\n";
}

int main() {
  // initialize an empty node
  struct Node* head = NULL;

  insertEnd(&head, 5);
  insertFront(&head, 1);
  insertFront(&head, 6);
  insertEnd(&head, 9);

  // insert 11 after head
  insertAfter(head, 11);

  // insert 15 after the seond node
```

```
 insertAfter(head->next, 15);


 displayList(head);


 // delete the last node
 deleteNode(&head, head->next->next->next->next->next);


 displayList(head);
}
```

3.5 Doubly Linked List Complexity

Doubly Linked List Complexity	Time Complexity	Space Complexity
Insertion Operation	O(1) or O(n)	O(1)
Deletion Operation	O(1)	O(1)

1. Complexity of Insertion Operation

The insertion operations that do not require traversal have the time complexity of `O(1)`.

And, insertion that requires traversal has time complexity of `O(n)`.

The space complexity is `O(1)`.

2. Complexity of Deletion Operation

All deletion operations run with time complexity of `O(1)`.

And, the space complexity is `O(1)`.

3.6 Doubly Linked List Applications

Redo and undo functionality in software.

Forward and backward navigation in browsers.

For navigation systems where forward and backward navigation is required.

3.7 Singly Linked List Vs Doubly Linked List

Singly Linked List	Doubly Linked List
Each node consists of a data value and a pointer to the next node.	Each node consists of a data value, a pointer to the next node, and a pointer to the previous node.
Traversal can occur in one way only (forward direction).	Traversal can occur in both ways.
It requires less space.	It requires more space because of an extra pointer.
It can be implemented on the stack.	It has multiple usages. It can be implemented on the stack, heap, and binary tree.

3.8 Practice Problems

Exercise 1

Design doubly linked list having basic functionality of adding an element at the end of the list and display all the entered list elements when user entered -1. Also implement the functionality for deletion in doubly linked list based upon the data (value in the list).

Exercise 2

Write a C++ program to check if a double linked list is a palindrome or not.
Test Data and Expected Output :

```
Original List:
1 2 3 4 5
Linked list is not a palindrome.

Original Singly List:
1 2 2 1
Linked list is a palindrome.

Original List:
```

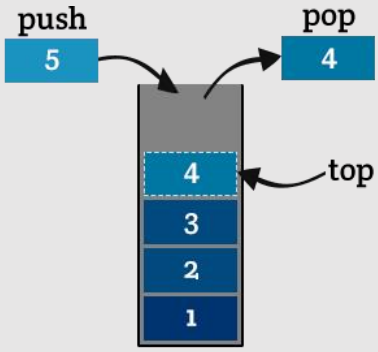
```
MADAM
Linked list is a palindrome.
```

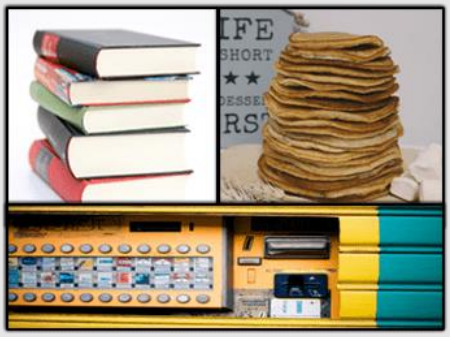
Exercise 3

Write a C++ program to remove duplicates from a double unsorted linked list.
Test Data and Expected Output :

```
Original List:
1 2 3 3 4
After removing duplicate elements from the said singly list:
1 2 3 4

Original List:
1 2 3 3 4 4
After removing duplicate elements from the said singly list:
1 2 3 4
```

Lab 04
Stack Data Structure





Prepared by: Engr. Amna Arooj

FCSE, GIKI

Lab# 04
Stack Data Structure in C++

4.1 Introduction to Stack in C++

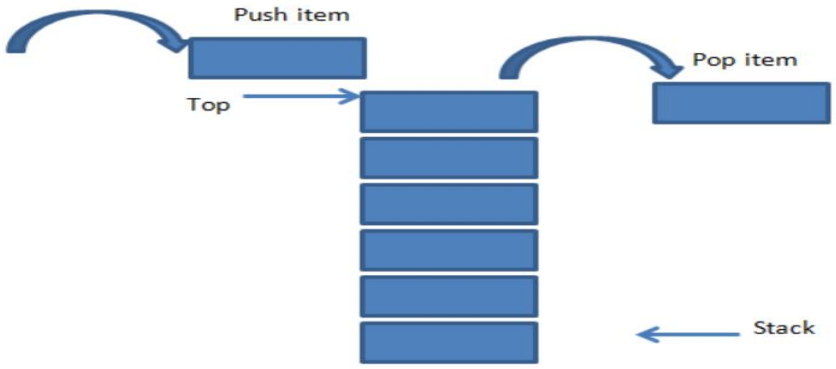
A stack is a linear data structure that contains a collection of elements in which the insertion of a new element and removal of an existing element takes place at the same end represented as the top of the stack.

To implement the stack, it is required to maintain the **pointer to the top of the stack**, which is the last element to be inserted because **we can access the elements only on the top of the stack.**

4.1.1 LIFO(Last In First Out):

Both insertion and removal are allowed at only one end of Stack called Top.The element that is inserted last will come out first. You can take a pile of plates kept on top of each other as a real-life example. The plate which we put last is on the top and since we remove the plate that is at the top, we can say that the plate that was put last comes out first.

Stack is said to be in **Overflow** state when it is completely full and is said to be in **Underflow** state if it is completely empty.



As shown above, there is a pile of plates stacked on top of each other. If we want to add another item to it, then we add it at the top of the stack as shown in the above figure (left-hand side). This operation of adding an item to stack is called "**Push**".

On the right side, we have shown an opposite operation i.e. we remove an item from the stack. This is also done from the same end i.e. the top of the stack. This operation is called "**Pop**".

As shown in the above figure, we see that push and pop are carried out from the same end. This makes the stack to follow LIFO order. The position or end from which the items are pushed in or popped out to/from the stack is called the "**Top of the stack**".

Initially, when there are no items in the stack, the top of the stack is set to -1. When we add an item to the stack, the top of the stack is incremented by 1 indicating that the item is added. As opposed to this, the top of the stack is decremented by 1 when an item is popped out of the stack.

4.2 Basic Operations

Following are the basic operations that are supported by the stack.

- **push –** Adds or pushes an element into the stack.
- **pop –** Removes or pops an element out of the stack.
- **peek –** Gets the top element of the stack but doesn't remove it.
- **isFull –** Tests if the stack is full.
- **isEmpty –** Tests if the stack is empty.

4.2.1 PUSH

This method allows us to add an element to the stack. Adding an element in a Stack in C++ occurs only at its top because of the LIFO policy.

In this process, the following steps are performed:

1. Check if the stack is **full** or not.
2. If the stack is full, then print error of overflow and exit the program.
3. If the stack is not full, then increment the top and add the element.

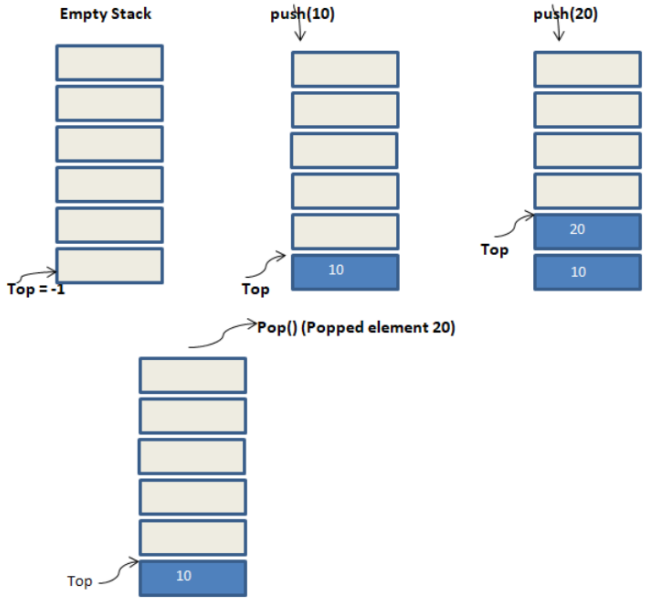
4.2.2 POP

This method allows us to remove an element from the top of the stack.

Dequeue operation consists of the following steps:

1. Check if the stack is empty or not.
2. If the stack is empty, then print error of underflow and exit the program.
3. If the stack is not empty, then print the element at the top and decrement the top.

4.2.3 Illustration



The above illustration shows the sequence of operations that are performed on the stack. Initially, the stack is empty. For an empty stack, the top of the stack is set to -1.

Next, we push the element 10 into the stack. We see that the top of the stack now points to element 10.

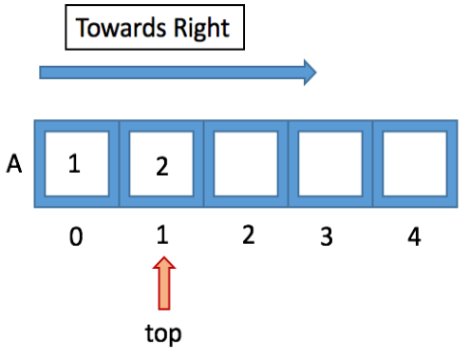
Next, we perform another push operation with element 20, as a result of which the top of the stack now points to 20. This state is the third figure.

Now in the last figure, we perform a pop () operation. As a result of the pop operation, the element pointed at the top of the stack is removed from the stack. Hence in the figure, we see that element 20 is removed from the stack. Thus the top of the stack now points to 10.

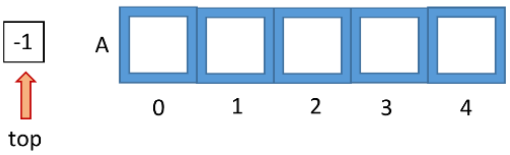
4.3 Array Implementation For Stack

Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size. Here we will implement Stack using array.

- When we implement stack uisng array we take the direction of the stack i.e the direction in which elements are inserted towards right.

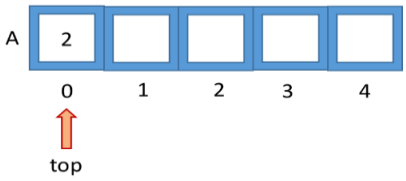


Lets take an example of an array of 5 elements to implement stack. So we define the size of the array using pre-processor directive `#deifne SIZE 5` & then we can create an array of 5 elements as `int A[SIZE];` Also we will declare a `top` variable to track the element at the top of the stack which will initially be -1 when the stack is empty i.e `int top=-1;`

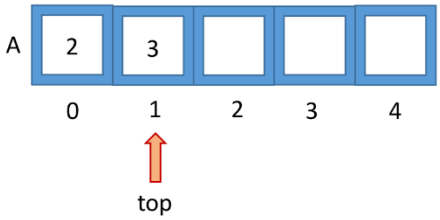


- `isempty()`- Now that we know that the stack is empty when `top` is equal to -1 we can use this to implement our `isempty()` function i.e. when `top == -1` retrun true else return false.
- `push()`- To push an element into the stack we will simply increment `top` by one and insert the element at that position. While inserting we have to take care of the condition when the array is full i.e. `when top == SIZE-1;`

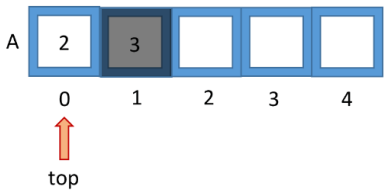
- `pop()`- To pop an element from the stack we will simple decrement `top` by one which will simply mean that the element is no longer the part of the stack. In this case we have to take care of the condition when the stack is empty i.e `top == -1` then we cannot perform the pop operation.
- `show_top()`- we will simply print the element at top if the stack is not empty.
- Lets say we have an stack with one element i.e 2 and we have to insert ot push an element 3 to this stack.



- then we will simply increment top by one and push the element at top.



- And to pop an element we will simply decrement top by one



Position of Top	Status of Stack
-1	Stack is Empty
0	Only one element in Stack
N-1	Stack is Full
N	Overflow state of Stack

Let us implement the stack data structure using C++.

```
#include <iostream>
using namespace std;

#define SIZE 5
int A[SIZE];
int top = -1;

bool isempty()
{
  if(top==-1)
  return true;
```

```
  else
  return false;
}

void push(int value)
{
  if(top==SIZE-1)
  {    cout<<"Stack is full!\n";
  }
   else
  {
    top++;
    A[top]=value;
  }
}

void pop()
{
 if(isempty())
  cout<<"Stack is empty!\n";
 else
  top--;
}

void show_top()
{
 if(isempty())
  cout<<"Stack is empty!\n";
 else
  cout<<"Element at top is: "<<A[top]<<"\n";

}

void displayStack()
{
  if(isempty())
 {
    cout<<"Stack is empty!\n";
 }
 else
 {
  for(int i=0 ; i<=top; i++)
    cout<<A[i]<<" ";
   cout<<"\n";

  }

}

int main()
{

 int choice, flag=1, value;
 while( flag == 1)
 {
 cout<<"\n1.PUSH 2.POP 3.SHOW_TOP 4.DISPLAY_STACK 5.EXIT\n";
 cin>>choice; switch (choice)
 {
 case 1: cout<<"Enter Value:\n";
```

```
          cin>>value;
          push(value);
          break;
 case 2: pop();
          break;
 case 3: show_top();
          break;
 case 4: displayStack();
          break;
 case 5: flag = 0;
          break;
 }
 }
  return 0;
}
```

Advantages of array implementation:

- Easy to implement.
- Memory is saved as pointers are not involved.

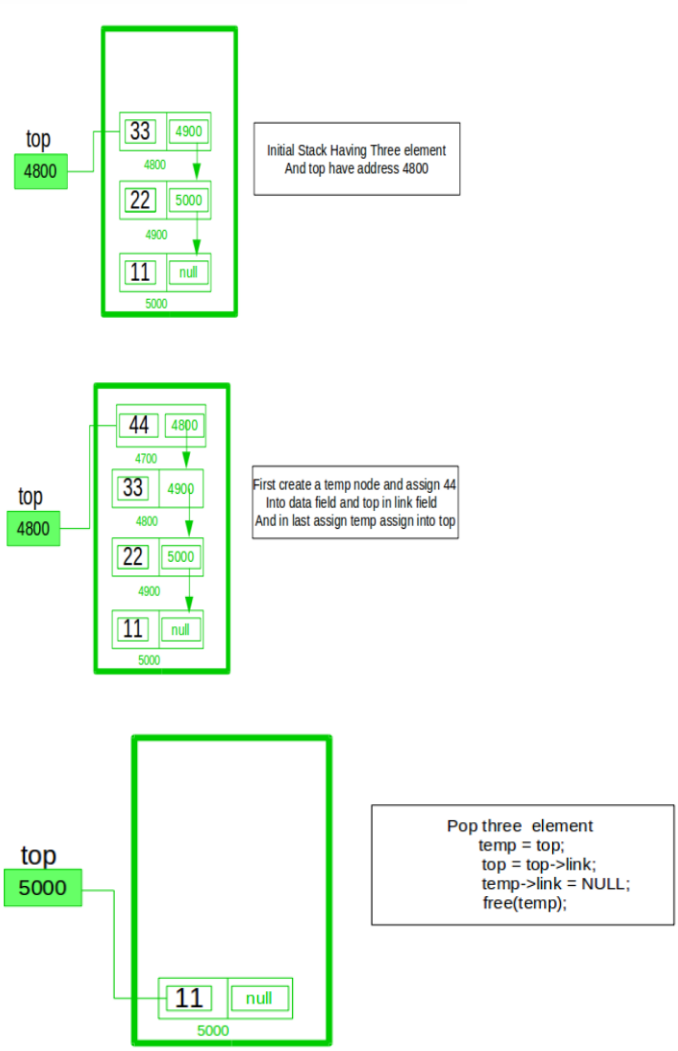
Disadvantages of array implementation:

- It is not dynamic i.e., it doesn't grow and shrink depending on needs at runtime. [But in case of dynamic sized arrays like vector in C++, list in Python, ArrayList in Java, stacks can grow and shrink with array implementation as well].
- The total size of the stack must be defined beforehand.

4.4 Linked List Implementation for Stack:

To implement a stack using the singly linked list concept, all the singly linked list operations should be performed based on Stack operations LIFO(last in first out) and with the help of that knowledge, we are going to implement a stack using a singly linked list.

So we need to follow a simple rule in the implementation of a stack which is **last in first out** and all the operations can be performed with the help of a top variable. Let us learn how to perform **Pop, Push, Peek, and Display** operations:



In the stack Implementation, a stack contains a top pointer. which is the “head” of the stack where pushing and popping items happens at the head of the list. The first node has a null in the link field and second node-link has the first node address in the link field and so on and the last node address is in the “top” pointer.

```
#include <iostream>
using namespace std;
struct Node {
  int data;
  struct Node *next;
};
struct Node* top = NULL;
void push(int val) {
  struct Node* newnode = (struct Node*) malloc(sizeof(struct Node));
  newnode->data = val;
```

```
  newnode->next = top;
  top = newnode;
}
void pop() {
  if(top==NULL)
  cout<<"Stack Underflow"<<endl;
  else {
    cout<<"The popped element is "<< top->data <<endl;
    top = top->next;
  }
}
void display() {
  struct Node* ptr;
  if(top==NULL)
  cout<<"stack is empty";
  else {
    ptr = top;
    cout<<"Stack elements are: ";
    while (ptr != NULL) {
      cout<< ptr->data <<" ";
      ptr = ptr->next;
    }
  }
  cout<<endl;
}
int main() {
  int ch, val;
  cout<<"1) Push in stack"<<endl;
  cout<<"2) Pop from stack"<<endl;
  cout<<"3) Display stack"<<endl;
  cout<<"4) Exit"<<endl;
  do {
    cout<<"Enter choice: "<<endl;
    cin>>ch;
    switch(ch) {
```

```
      case 1: {
        cout<<"Enter value to be pushed:"<<endl;
        cin>>val;
        push(val);
        break;
      }
      case 2: {
        pop();
        break;
      }
      case 3: {
        display();
        break;
      }
      case 4: {
        cout<<"Exit"<<endl;
        break;
      }
      default: {
        cout<<"Invalid Choice"<<endl;
      }
    }
  }while(ch!=4);
  return 0;
}
```

Output:

```
1) Push in stack
2) Pop from stack
3) Display stack
4) Exit

Enter choice: 1
Enter value to be pushed: 2
Enter choice: 1
Enter value to be pushed: 6
```

```
Enter choice: 1
Enter value to be pushed: 8
Enter choice: 1
Enter value to be pushed: 7
Enter choice: 2
The popped element is 7
Enter choice: 3
Stack elements are:8 6 2
Enter choice: 5
Invalid Choice
Enter choice: 4
Exit
```

Advantages of Linked List implementation:

- The linked list implementation of a stack can grow and shrink according to the needs at runtime.
- It is used in many virtual machines like JVM.

Disadvantages of Linked List implementation:

- Requires extra memory due to the involvement of pointers.
- Random accessing is not possible in stack.

4.5 Analysis of Stack Operations

Below mentioned are the time complexities for various operations that can be performed on the Stack data structure.

Push Operation : O(1)

Pop Operation : O(1)

Top Operation : O(1)

Search Operation : O(n)

The time complexities for `push()` and `pop()` functions are `O(1)` because we always have to insert or remove the data from the **top** of the stack, which is a one step process.

4.6 Applications Of Stack

Let us discuss the various applications of the stack data structure below.

- CD/DVD stand.
- Stack of books in a book shop.
- Call center systems.
- Undo and Redo mechanism in text editors.

- The history of a web browser is stored in the form of a stack.
- Call logs, E-mails, and Google photos in any gallery are also stored in form of a stack.
- YouTube downloads and Notifications are also shown in LIFO format(the latest appears first).
- Allocation of memory by an operating system while executing a process.
- A Stack can be used for evaluating expressions consisting of operands and operators.
- Stacks can be used for Backtracking, i.e., to check parenthesis matching in an expression.
- It can also be used to convert one form of expression to another form.
- It can be used for systematic Memory Management.

Practice Problems:

LAB Activity#01:

Write a C++ function to check if a given string of parentheses is balanced or not using a stack.Your task is to implement the function with the following prototype:

bool isBalanced(const std::string& parentheses);

The function should take a string containing only parentheses as an argument. It should return `true` if the parentheses in the string are balanced (i.e., every opening parenthesis has a corresponding closing parenthesis in the correct order), and `false` otherwise.

For example, if the input string is "(())", the function should return `true`. If the input string is "(()", the function should return `false`.

LAB Activity#02:

Write a C++ function to reverse a string using a stack.

Your task is to implement the function with the following prototype:

std::string reverseString(const std::string& input);

The function should take a string as input and return a new string that is the reverse of the input string using a stack.

For example, if the input string is "Hello", the function should return "olleH".

Lab#05
Queues Data Structure in C++

Prepared by: Engr. Amna Arooj

FCSE, GIKI

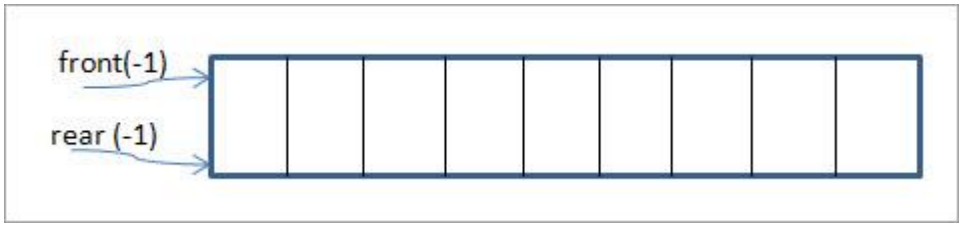
Lab# 05
Queue Data Structure In C++

5.1 Introduction To Queue In C++ With Illustration.

The queue is a basic data structure just like a stack. In contrast to stack that uses the LIFO approach, queue uses the FIFO (first in, first out) approach. With this approach, the first item that is added to the queue is the first item to be removed from the queue. Just like Stack, the queue is also a linear data structure.

In a real-world analogy, we can imagine a bus queue where the passengers wait for the bus in a queue or a line. The first passenger in the line enters the bus first as that passenger happens to be the one who had come first.

In software terms, the queue can be viewed as a set or collection of elements as shown below. The elements are arranged linearly.



We have two ends i.e. “front” and “rear” of the queue. When the queue is empty, then both the pointers are set to -1.

The “rear” end pointer is the place from where the elements are inserted in the queue. The operation of adding /inserting elements in the queue is called “enqueue”.

The “front” end pointer is the place from where the elements are removed from the queue. The operation to remove/delete elements from the queue is called “dequeue”.

When the rear pointer value is size-1, then we say that the queue is full. When the front is null, then the queue is empty.

5.2 Basic Operations

The queue data structure includes the following operations:

- **EnQueue:** Adds an item to the queue. Addition of an item to the queue is always done at the rear of the queue.
- **DeQueue:** Removes an item from the queue. An item is removed or de-queued always from the front of the queue.
- **isEmpty:** Checks if the queue is empty.
- **isFull:** Checks if the queue is full.
- **peek:** Gets an element at the front of the queue without removing it.

5.2.1 Enqueue

In this process, the following steps are performed:

- Check if the queue is full.
- If full, produce overflow error and exit.
- Else, increment 'rear'.
- Add an element to the location pointed by 'rear'.
- Return success.

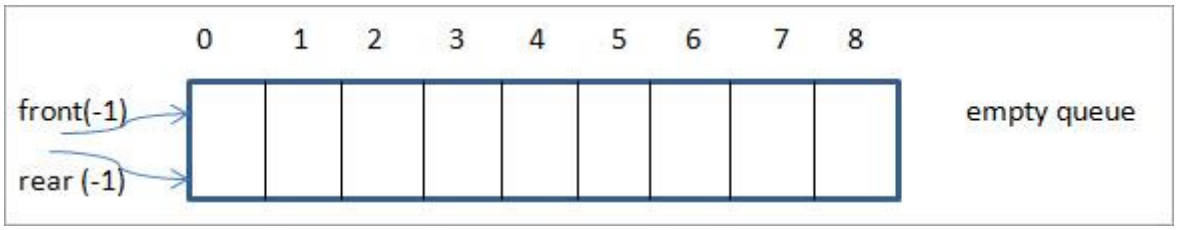
5.2.2 Dequeue

Dequeue operation consists of the following steps:

- Check if the queue is empty.
- If empty, display an underflow error and exit.
- Else, the access element is pointed out by 'front'.
- Increment the 'front' to point to the next accessible data.
- Return success.

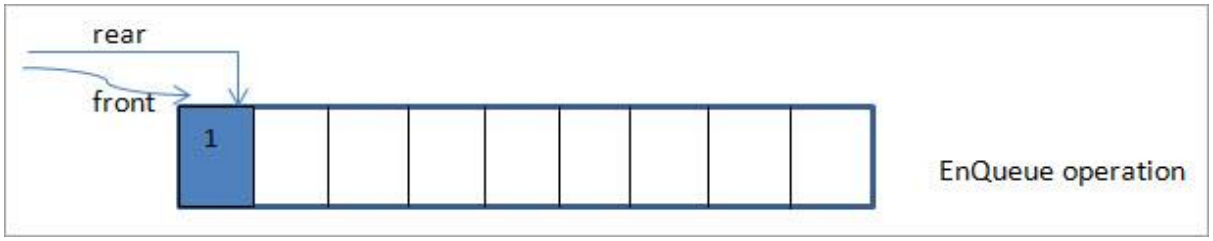
Next, we will see a detailed illustration of insertion and deletion operations in queue.

5.2.3 Illustration

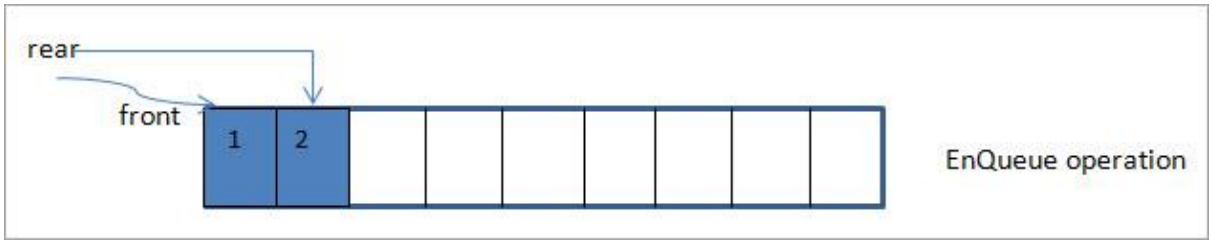


This is an empty queue and thus we have rear and empty set to -1.

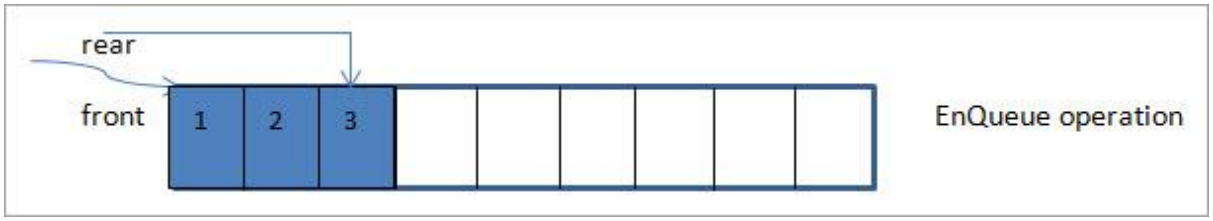
Next, we add 1 to the queue and as a result, the rear pointer moves ahead by one location.



In the next figure, we add element 2 to the queue by moving the rear pointer ahead by another increment.

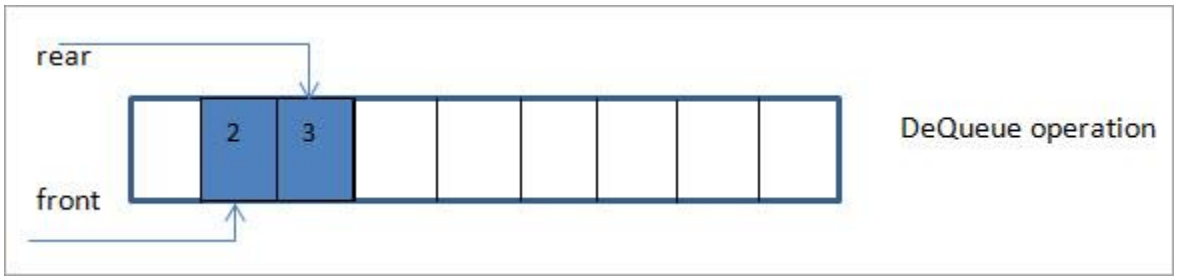


In the following figure, we add element 3 and move the rear pointer by 1.



At this point, the rear pointer has value 2 while the front pointer is at the 0th location.

Next, we delete the element pointed by the front pointer. As the front pointer is at 0, the element that is deleted is 1.



Thus the first element entered in the queue i.e. 1 happens to be the first element removed from the queue. As a result, after the first dequeue, the front pointer now will be moved ahead t0 the next location which is 1.

5.3 Array Implementation For Queue

Let us implement the queue data structure using C++.

```
#include <iostream>

#define MAX_SIZE 5

using namespace std;
```

```
class Queue {
private:
int myqueue[MAX_SIZE], front, rear;

public:
Queue(){
front = -1;
rear = -1;
   }

boolisFull(){
if(front == 0 && rear == MAX_SIZE - 1){
return true;
      }
return false;
   }

boolisEmpty(){
if(front == -1) return true;
else return false;
   }

void enQueue(int value){
if(isFull()){
cout << endl<< "Queue is full!!";
      } else {
if(front == -1) front = 0;
rear++;
myqueue[rear] = value;
cout << value << " ";
```

```
      }
   }
int deQueue(){
int value;
if(isEmpty()){
cout << "Queue is empty!!" << endl; return(-1); } else { value = myqueue[front]; if(front >= rear){      //only one element in queue
front = -1;
rear = -1;
          }
else {
front++;
          }
cout << endl << "Deleted => " << value << " from myqueue";
return(value);
      }
   }

   /* Function to display elements of Queue */
void displayQueue()
   {
int i;
if(isEmpty()) {
cout << endl << "Queue is Empty!!" << endl;
      }
else {
cout << endl << "Front = " << front;
cout << endl << "Queue elements : ";
for(i=front; i<=rear; i++)
cout << myqueue[i] << "\t";
cout << endl << "Rear = " << rear << endl;
```

```
    }
  }
};
int main()
{
   Queue myq;

myq.deQueue();      //deQueue

cout<<"Queue created:"<<endl; myq.enQueue(10); myq.enQueue(20);
myq.enQueue(30); myq.enQueue(40); myq.enQueue(50); //enqueue 60 => queue is full
myq.enQueue(60);

myq.displayQueue();

   //deQueue =>removes 10
myq.deQueue();

   //queue after dequeue
myq.displayQueue();

return 0;
}
```

Output:

```
Queue is empty!!
Queue created:
10   20   30   40   50
Queue is full!!
Front = 0
Queue elements : 10          20          30          40          50
Rear = 4

Deleted => 10 from myqueue
Front = 1
```

```
Queue elements: 20        30        40        50
Rear = 4
```

The above implementation shows the queue represented as an array. We specify the max_size for the array. We also define the enqueue and dequeue operations as well as the isFull and isEmpty operations.

5.4 Linked List Implementation for Queue:

Let us implement the queue in C++ using a linked list.

```
include <iostream>
using namespace std;
struct node {
int data;
struct node *next;
};
struct node* front = NULL;
struct node* rear = NULL;
struct node* temp;
void Insert(int val) {
if (rear == NULL) {
rear = new node;
rear->next = NULL;
rear->data = val;
front = rear;
     } else {
temp=new node;
rear->next = temp;
temp->data = val;
temp->next = NULL;
rear = temp;
   }
}
void Delete() {
```

```
temp = front;
if (front == NULL) {
cout<<"Queue is empty!!"<<endl; } else if (temp->next != NULL) {
temp = temp->next;
cout<<"Element deleted from queue is : "<<front->data<<endl;
free(front);
front = temp;
   } else {
cout<<"Element deleted from queue is : "<<front->data<<endl;
free(front);
front = NULL;
rear = NULL;
   }
}
void Display() {
temp = front;
if ((front == NULL) && (rear == NULL)) {
cout<<"Queue is empty"<<endl;
return;
   }
while (temp != NULL) {
cout<<temp->data<<" "; temp = temp->next;
   }
cout<<endl;
}
int main() {

cout<<"Queue Created:"<<endl;
Insert(10);
Insert(20);
Insert(30);
Insert(40);
Insert(50);
Display();
```

```
Delete();

cout<<"Queue after one deletion: "<<endl;

Display();


return 0;

}
```

Output:

```
Queue Created:
10      20      30      40      50
Element deleted from queue is: 10
Queue after one deletion:
20  30   40  50
```

5.5 Stack Vs. Queue

Stacks and queues are secondary data structures which can be used to store data. They can be programmed using the primary data structures like arrays and linked lists. Having discussed both the data structures in detail, it's time to discuss the main differences between these two data structures.

Stacks	Queues
Uses LIFO (Last in, First out) approach.	Uses FIFO (First in, First out) approach.
Items are added or deleted from only one end called "Top" of the stack.	Items are added from "Rear" end of the queue and are removed from the "front" of the queue.
The basic operations for the stack are "push" and "Pop".	The basic operations for a queue are "enqueue" and "dequeue".
We can do all operations on the stack by maintaining only one pointer to access the top of the stack.	In queues, we need to maintain two pointers, one to access the front of the queue and the second one to access the rear of the queue.
The stack is mostly used to solve recursive problems.	Queues are used to solve problems related to ordered processing.

5.6 Applications Of Queue

Let us discuss the various applications of the queue data structure below.

- The queue data structure is used in various CPU and disk scheduling. Here we have multiple tasks requiring CPU or disk at the same time. The CPU or disk time is scheduled for each task using a queue.

- The queue can also be used for print spooling wherein the number of print jobs is placed in a queue.
- Handling of interrupts in real-time systems is done by using a queue data structure. The interrupts are handled in the order they arrive.
- Breadth-first search in which the neighboring nodes of a tree are traversed before moving on to next level uses a queue for implementation.
- Call center phone systems use queues to hold the calls until they are answered by the service representatives.
- In general, we can say that the queue data structure is used whenever we require the resources or items to be serviced in the order they arrive i.e. First in, First Out.

Conclusion

The queue is a FIFO (First In, First Out) data structure that is mostly used in resources where scheduling is required. It has two pointers rear and front at two ends and these are used to insert an element and remove an element to/from the queue respectively.

Practice Problems:

Implement the queue that perform following functions:

1: Add a node (data member of type: string), Enqueue()

2: Delete a node, Dequeue()

3: Display the queue, Display()

In this activity modify above program by writing another function int LengthOfString(Queue *) “This function will return the number of characters in the string of nodes one by one and dequeue that node until queue is empty”.

Lab#06
Sorting Algorithms I

Prepared by: Engr. Amna Arooj

FCSE, GIKI

Lab#06
Sorting Algorithms I

6.1 Bubble Sort

Bubble Sort is a simple algorithm which is used to sort a given set of n elements provided in form of an array with n number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will **swap** both the elements, and then move on to compare the second and the third element, and so on.

6.1.1 How does Bubble Sort work?

If we have total n elements, then we need to repeat this process for n-1 times.

It is known as **bubble sort**, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

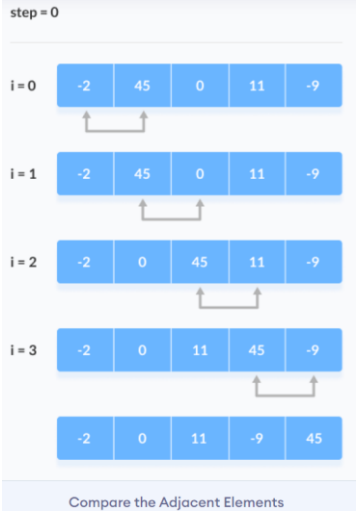
Sorting takes place by stepping through all the elements one-by-one and comparing it with the adjacent element and swapping them if required.

6.1.2 Implementing Bubble Sort Algorithm:

Suppose we are trying to sort the elements in ascending order.

1. First Iteration (Compare and Swap:

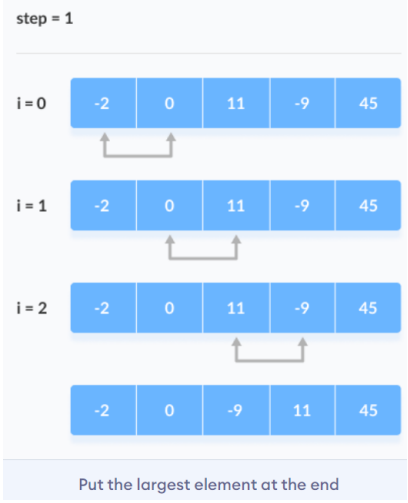
1. Starting from the first index, compare the first and the second elements.
2. If the first element is greater than the second element, they are swapped.
3. Now, compare the second and the third elements. Swap them if they are not in order.
4. The above process goes on until the last element.



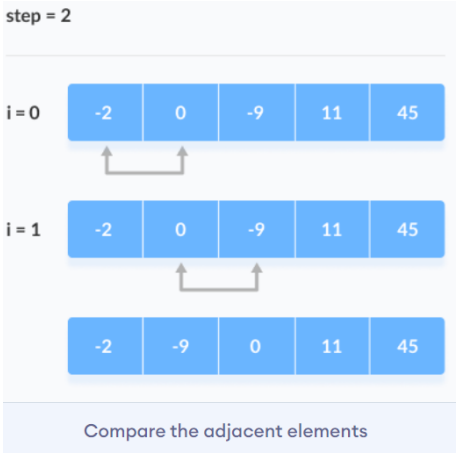
2. Remaining Iteration

The same process goes on for the remaining iterations.

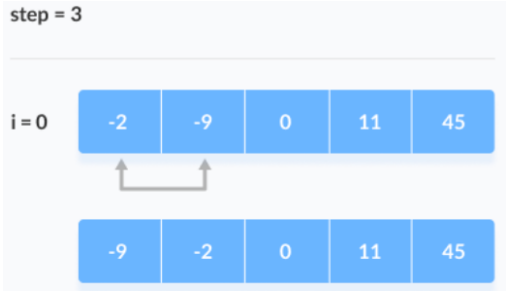
After each iteration, the largest element among the unsorted elements is placed at the end.



In each iteration, the comparison takes place up to the last unsorted element.



The array is sorted when all the unsorted elements are placed at their correct positions.



The array is sorted if all elements are kept in the right order.

6.1.2.1 Bubble Sort Algorithm

```
bubbleSort(array)
  for i <- 1 to indexOfLastUnsortedElement-1
    if leftElement > rightElement
      swap leftElement and rightElement
end bubbleSort
```

6.1.3 Bubble Sort Implementation in C++

Below is the Code implementation of Bubble Sort.

```
// Bubble sort in C++
#include <iostream>
using namespace std;
// perform bubble sort
void bubbleSort(int array[], int size) {

  // loop to access each array element
  for (int step = 0; step < size; ++step) {

    // loop to compare array elements
    for (int i = 0; i < size - step; ++i) {

      // compare two adjacent elements
      // change > to < to sort in descending order
      if (array[i] > array[i + 1]) {

        // swapping elements if elements
        // are not in the intended order
        int temp = array[i];
        array[i] = array[i + 1];
        array[i + 1] = temp;
      }
    }
  }
}
// print array
void printArray(int array[], int size) {
  for (int i = 0; i < size; ++i) {
    cout << "  " << array[i];
  }
  cout << "\n";
}
int main() {
  int data[] = {-2, 45, 0, 11, -9};

  // find array's length
  int size = sizeof(data) / sizeof(data[0]);
```

```
  bubbleSort(data, size);

  cout << "Sorted Array in Ascending Order:\n";
  printArray(data, size);
}
```

6.1.4 Complexity Analysis of Bubble Sort:

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

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is **O(n).**
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is **$O(n^2)$.**
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is **$O(n^2)$.**

Space Complexity	O(1)

6.1.5 Applications of Bubble Sort:

Bubble sort is used if

- complexity does not matter
- short and simple code is preferred

6.1.6 Advantages of Bubble Sort:

- Bubble sort is easy to understand and implement.
- It does not require any additional memory space.
- It is a stable sorting algorithm, meaning that elements with the same key value maintain their relative order in the sorted output.

6.1.7 Drawbacks of Bubble Sort:

- Bubble sort has a time complexity of $O(N^2)$ which makes it very slow for large data sets.

- Bubble sort is a comparison-based sorting algorithm, which means that it requires a comparison operator to determine the relative order of elements in the input data set. It can limit the efficiency of the algorithm in certain cases.

6.2 Insertion Sort – Data Structure and Algorithm C++

Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

Consider you have 10 cards out of a deck of cards in your hand. And they are sorted, or arranged in the ascending order of their numbers.

If I give you another card, and ask you to **insert** the card in just the right position, so that the cards in your hand are still sorted. What will you do?

Well, you will have to go through each card from the starting or the back and find the right position for the new card, comparing it's value with each card. Once you find the right position, you will **insert** the card there.

Similarly, if more new cards are provided to you, you can easily repeat the same process and insert the new cards and keep the cards sorted too.

This is exactly how **insertion sort** works. It starts from the index 1(not 0), and each index starting from index 1 is like a new card, that you have to place at the right position in the sorted subarray on the left.

6.2.1 How does Insertion Sort Work?

The simple steps of achieving the insertion sort are listed as follows -

Step 1 - If the element is the first element, assume that it is already sorted. Return 1.

Step2 - Pick the next element, and store it separately in a **key.**

Step3 - Now, compare the **key** with all elements in the sorted array.

Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

Step 5 - Insert the value.

Step 6 - Repeat until the array is sorted.

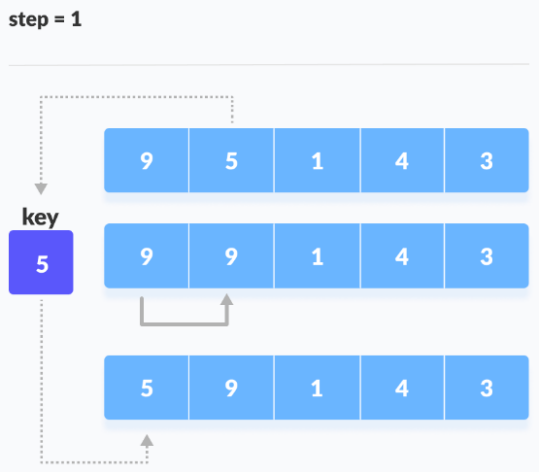
6.2.1.1 Illustration:

Suppose we need to sort the following array.

Initial array

1. The first element in the array is assumed to be sorted. Take the second element and store it separately in `key`.

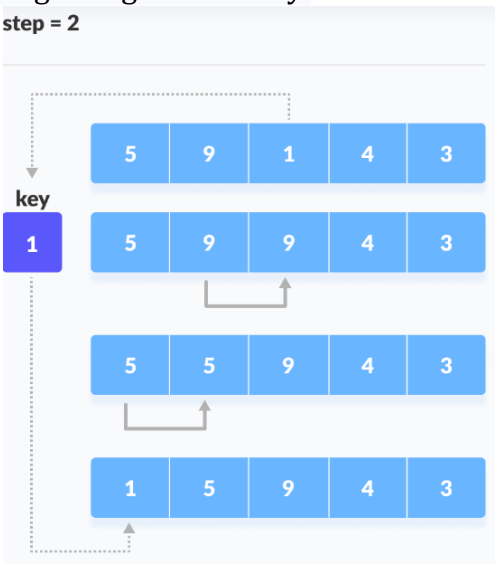
   Compare `key` with the first element. If the first element is greater than `key`, then `key` is placed in front of the first element.



If the first element is greater than key, then key is placed in front of the first element.

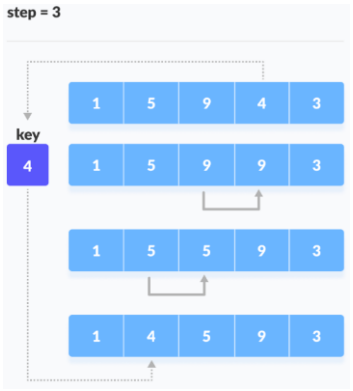
2. Now, the first two elements are sorted.

   Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.

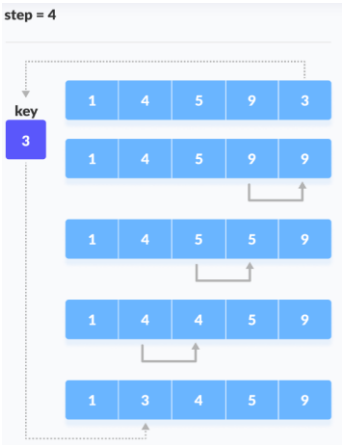


Place 1 at the beginning

3. Similarly, place every unsorted element at its correct position



Place 4 behind 1



Place 3 behind 1 and the array is sorted

6.2.2 Insertion Sort Algorithm

```
insertionSort(array)
  mark first element as sorted
  for each unsorted element X
    'extract' the element X
    for j <- lastSortedIndex down to 0
      if current element j > X
        move sorted element to the right by 1
    break loop and insert X here
end insertionSort
```

6.2.3 Insertion Sort Implementation in C++

```
// Insertion sort in C++

#include <iostream>
using namespace std;

// Function to print an array
void printArray(int array[], int size) {
  for (int i = 0; i < size; i++) {
```

```
    cout << array[i] << " ";
  }
  cout << endl;
}

void insertionSort(int array[], int size) {
  for (int step = 1; step < size; step++) {
    int key = array[step];
    int j = step - 1;

    // Compare key with each element on the left of it until an element smaller than
    // it is found.
    // For descending order, change key<array[j] to key>array[j].
    while (key < array[j] && j >= 0) {
      array[j + 1] = array[j];
      --j;
    }
    array[j + 1] = key;
  }
}

// Driver code
int main() {
  int data[] = {9, 5, 1, 4, 3};
  int size = sizeof(data) / sizeof(data[0]);
  insertionSort(data, size);
  cout << "Sorted array in ascending order:\n";
  printArray(data, size);
}
```

6.2.4 Complexity Analysis of Insertion Sort:

Time Complexity	
Best	O(n)
Worst	$O(n^2)$
Average	$O(n^2)$
Space Complexity	O(1)

6.2.5 Applications of Insertion Sort

The insertion sort is used when:

1. the array is has a small number of elements
2. there are only a few elements left to be sorted

6.2.6 Advantages of Insertion Sort:

1. It is efficient for smaller data sets, but very inefficient for larger lists.
2. Insertion Sort is adaptive, that means it reduces its total number of steps if a partially sorted array is provided as input, making it efficient.
3. It is better than Selection Sort and Bubble Sort algorithms.
4. Its space complexity is less. Like bubble Sort, insertion sort also requires a single additional memory space.
5. It is a **stable** sorting technique, as it does not change the relative order of elements which are equal.

6.2.7 Disadvantages of Insertion Sort:

1. Inefficient for large datasets
2. worst-case time complexity of O(n^2)

6.3 Selection Sort

Selection sort is conceptually the most simplest sorting algorithm. This algorithm will first find the **smallest** element in the array and swap it with the element in the **first** position, then it will find the **second smallest** element and swap it with the element in the **second** position, and it will keep on doing this until the entire array is sorted.

It is called selection sort because it repeatedly **selects** the next-smallest element and swaps it into the right place.

6.3.1 How Selection Sort Works?

Following are the steps involved in selection sort(for sorting a given array in ascending order):

1. Starting from the first element, we search the smallest element in the array, and replace it with the element in the first position.
2. We then move on to the second position, and look for smallest element present in the subarray, starting from index 1, till the last index.
3. We replace the element at the **second** position in the original array, or we can say at the first position in the subarray, with the second smallest element.
4. This is repeated, until the array is completely sorted.

6.3.1.1 Illustration:

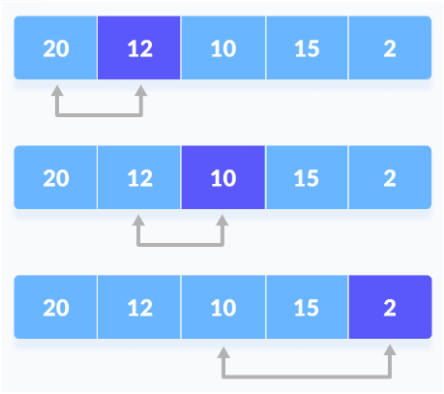
Let's consider below array:

1. Set the first element as `minimum`

2. Compare `minimum` with the second element. If the second element is smaller than `minimum`, assign the second element as `minimum`.

   Compare `minimum` with the third element. Again, if the third element is smaller,

then assign `minimum` to the third element otherwise do nothing. The process goes on until the last element.

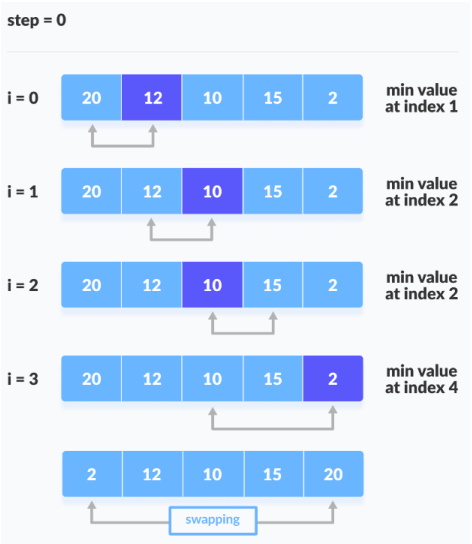


Compare minimum with the remaining elements

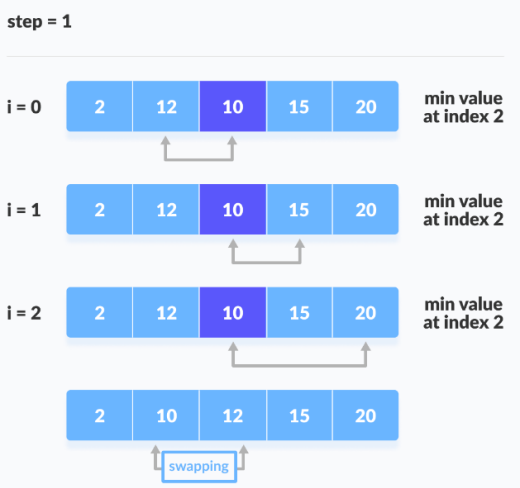
3. After each iteration, `minimum` is placed in the front of the unsorted list

Swap the first with minimum

4. For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.



The first iteration



The second iteration

The third iteration

The fourth iteration

6.3.2 Selection Sort Algorithm

```
selectionSort(array, size)
  repeat (size - 1) times
  set the first unsorted element as the minimum
  for each of the unsorted elements
    if element < currentMinimum
      set element as new minimum
```

```
  swap minimum with first unsorted position
end selectionSort
```

6.3.3 Insertion Sort Implementation in C++

```
// Selection sort in C++

#include <iostream>
using namespace std;

// function to swap the the position of two elements
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

// function to print an array
void printArray(int array[], int size) {
  for (int i = 0; i < size; i++) {
    cout << array[i] << " ";
  }
  cout << endl;
}

void selectionSort(int array[], int size) {
  for (int step = 0; step < size - 1; step++) {
    int min_idx = step;
    for (int i = step + 1; i < size; i++) {

      // To sort in descending order, change > to < in this line.
      // Select the minimum element in each loop.
      if (array[i] < array[min_idx])
        min_idx = i;
    }

    // put min at the correct position
    swap(&array[min_idx], &array[step]);
  }
}

// driver code
int main() {
  int data[] = {20, 12, 10, 15, 2};
  int size = sizeof(data) / sizeof(data[0]);
  selectionSort(data, size);
  cout << "Sorted array in Acsending Order:\n";
  printArray(data, size);
}
```

6.3.4 Complexity Analysis of Selection Sort

Selection Sort requires two nested for loops to complete itself, one for loop is in the function selectionSort, and inside the first loop we are making a call to another function indexOfMinimum, which has the second(inner) for loop.

Hence for a given input size of n, following will be the time and space complexity for selection sort algorithm:

Worst Case Time Complexity [Big-O]: **$O(n^2)$**

Best Case Time Complexity [Big-omega]: **$O(n^2)$**

Average Time Complexity [Big-theta]: **$O(n^2)$**

Space Complexity: **O(1)**

6.3.5 Selection Sort Applications

The selection sort is used when

1. a small list is to be sorted
2. cost of swapping does not matter
3. checking of all the elements is compulsory
4. cost of writing to a memory matters like in flash memory (number of writes/swaps is $O(n)$ as compared to $O(n^2)$ of bubble sort)

6.3.6 Advantages of Selection Sort

1. It can also be used on list structures that make add and remove efficient, such as a linked list. Just remove the smallest element of unsorted part and end at the end of sorted part.
2. The number of swaps reduced. **O(N)** swaps in all cases.
3. In-Place sort.

6.3.7 Disadvantages of Selection Sort

1. Time complexity in all cases is **$O(N^2)$**, no best case scenario.
2. It requires n-squared number of steps for sorting n elements.
3. It is not scalable.

Practice Problems:

Lab Activity #01:

Write a C++ function called `insertion_sort` that takes in a list of integers and sorts it using the insertion sort algorithm. The function should modify the original list in-place and return nothing.

For example, if the input list is [5, 2, 8, 12, 3], the function should modify the list to become [2, 3, 5, 8, 12].

Please provide the implementation of the `insertion_sort` function.

Lab Activity #02:

Write a C++ function called `selectionSort` that takes in an array of integers and sorts it using the selection sort algorithm. The function should modify the original array in-place and return nothing.

For example, if the input array is {9, 4, 7, 1, 3}, the function should modify the array to become {1, 3, 4, 7, 9}.

Please provide the implementation of the `selectionSort` function in C++.

Lab#07
Sorting Algorithms II

Prepared by: Engr. Amna Arooj

FCSE, GIKI

Lab#07
Sorting Algorithms II

7.1 Merge Sort

Merge sort is defined as a *sorting algorithm* that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

7.1.1 How does Merge Sort work?

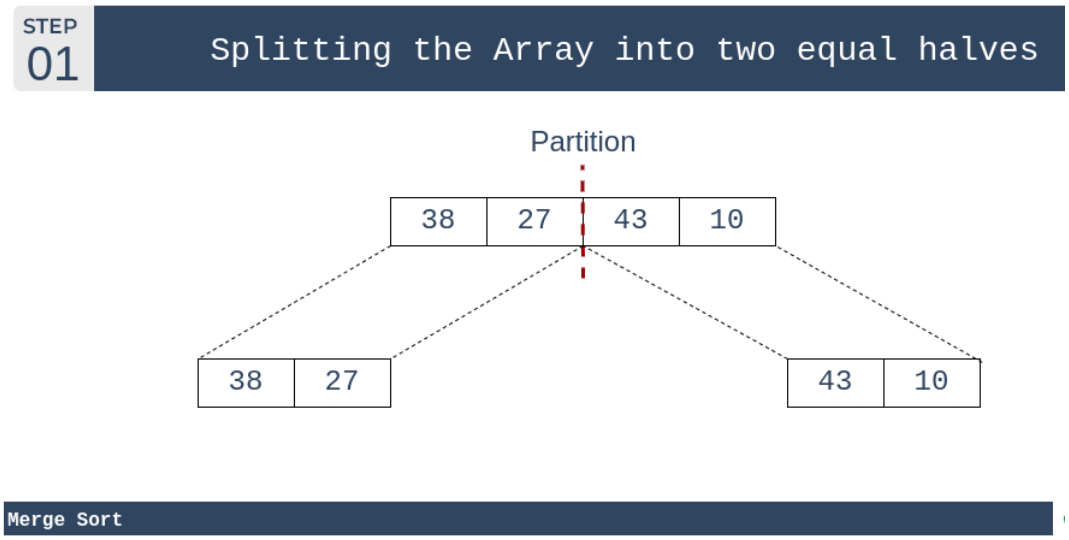
Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.

See the below illustration to understand the working of merge sort.

7.1.2 Illustration:

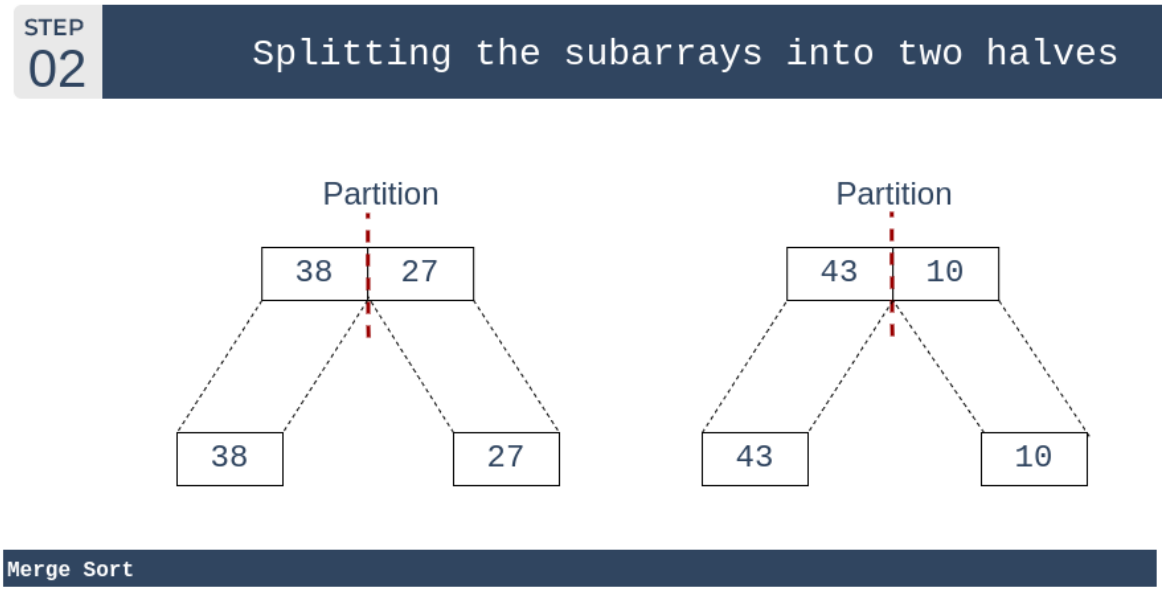
Lets consider an array ***arr[] = {38, 27, 43, 10}***

Initially divide the array into two equal halves:



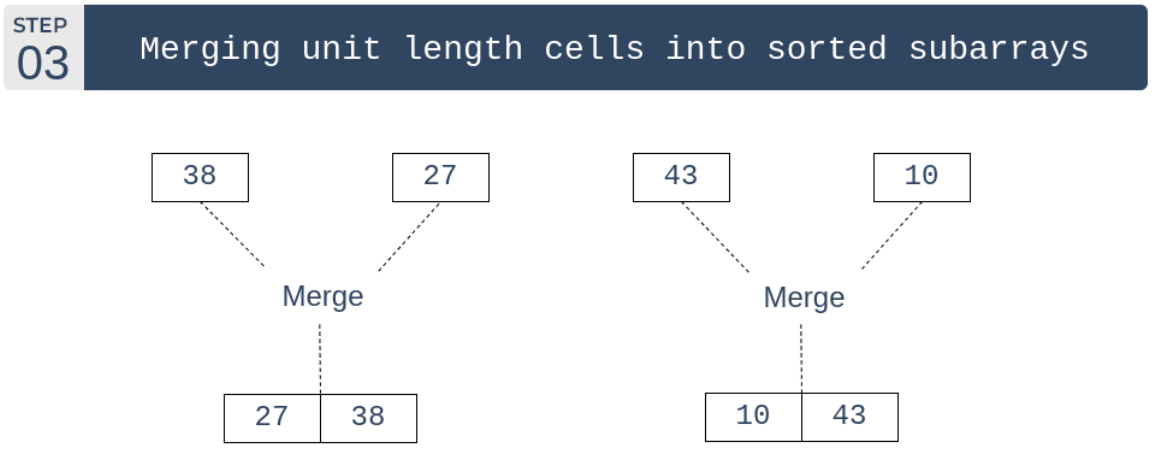
Merge Sort: Divide the array into two halves

- These subarrays are further divided into two halves. Now they become array of unit length that can no longer be divided and array of unit length are always sorted.



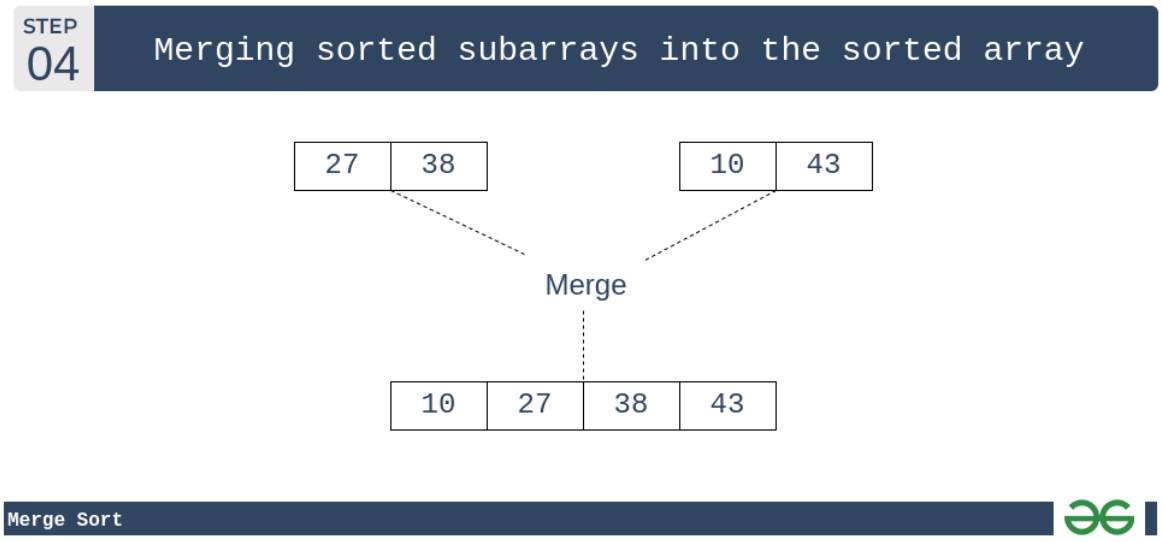
Merge Sort: Divide the subarrays into two halves (unit length subarrays here)

These sorted subarrays are merged together, and we get bigger sorted subarrays.



Merge Sort: Merge the unit length subarrys into sorted subarrays

This merging process is continued until the sorted array is built from the smaller subarrays.



Merge Sort: Merge the sorted subarrys to get the sorted array

The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}.

7.1.3 Merge Sort Implementation in C++

Below is the Code implementation of Merge Sort.

```
// C++ program for Merge Sort
#include <bits/stdc++.h>
using namespace std;

// Merges two subarrays of array[].
// First subarray is arr[begin..mid]
// Second subarray is arr[mid+1..end]
void merge(int array[], int const left, int const mid,
           int const right)
{
    int const subArrayOne = mid - left + 1;
    int const subArrayTwo = right - mid;
```

```
// Create temp arrays
auto *leftArray = new int[subArrayOne],
     *rightArray = new int[subArrayTwo];

// Copy data to temp arrays leftArray[] and rightArray[]
for (auto i = 0; i < subArrayOne; i++)
    leftArray[i] = array[left + i];
for (auto j = 0; j < subArrayTwo; j++)
    rightArray[j] = array[mid + 1 + j];

auto indexOfSubArrayOne = 0, indexOfSubArrayTwo = 0;
int indexOfMergedArray = left;

// Merge the temp arrays back into array[left..right]
while (indexOfSubArrayOne < subArrayOne
       && indexOfSubArrayTwo < subArrayTwo) {
    if (leftArray[indexOfSubArrayOne]
        <= rightArray[indexOfSubArrayTwo]) {
        array[indexOfMergedArray]
            = leftArray[indexOfSubArrayOne];
        indexOfSubArrayOne++;
    }
    else {
        array[indexOfMergedArray]
            = rightArray[indexOfSubArrayTwo];
        indexOfSubArrayTwo++;
    }
    indexOfMergedArray++;
}
```

```
    // Copy the remaining elements of
    // left[], if there are any
    while (indexOfSubArrayOne < subArrayOne) {
        array[indexOfMergedArray]
            = leftArray[indexOfSubArrayOne];
        indexOfSubArrayOne++;
        indexOfMergedArray++;
    }

    // Copy the remaining elements of
    // right[], if there are any
    while (indexOfSubArrayTwo < subArrayTwo) {
        array[indexOfMergedArray]
            = rightArray[indexOfSubArrayTwo];
        indexOfSubArrayTwo++;
        indexOfMergedArray++;
    }
    delete[] leftArray;
    delete[] rightArray;
}

// begin is for left index and end is right index
// of the sub-array of arr to be sorted
void mergeSort(int array[], int const begin, int const end)
{
    if (begin >= end)
        return;

    int mid = begin + (end - begin) / 2;
    mergeSort(array, begin, mid);
```

```
    mergeSort(array, mid + 1, end);
    merge(array, begin, mid, end);
}

// UTILITY FUNCTIONS
// Function to print an array
void printArray(int A[], int size)
{
    for (int i = 0; i < size; i++)
        cout << A[i] << " ";
    cout << endl;
}

// Driver code
int main()
{
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    int arr_size = sizeof(arr) / sizeof(arr[0]);

    cout << "Given array is \n";
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    cout << "\nSorted array is \n";
    printArray(arr, arr_size);
    return 0;
}

// This code is contributed by Mayank Tyagi
```

```
// This code was revised by Joshua Estes
```

Output

```
Given array is
12 11 13 5 6 7
Sorted array is
5 6 7 11 12 13
```

7.1.4 Complexity Analysis of Merge Sort:

Time Complexity: O(N log(N)), Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$T(n) = 2T(n/2) + \theta(n)$

The above recurrence can be solved either using the Recurrence Tree method or the Master method. It falls in case II of the Master Method and the solution of the recurrence is θ(Nlog(N)). The time complexity of Merge Sort isθ(Nlog(N)) in all 3 cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

Auxiliary Space: O(N), In merge sort all elements are copied into an auxiliary array. So N auxiliary space is required for merge sort.

7.1.5 Applications of Merge Sort:

- **Sorting large datasets:** Merge sort is particularly well-suited for sorting large datasets due to its guaranteed worst-case time complexity of O(n log n).
- **External sorting:** Merge sort is commonly used in external sorting, where the data to be sorted is too large to fit into memory.
- **Custom sorting:** Merge sort can be adapted to handle different input distributions, such as partially sorted, nearly sorted, or completely unsorted data.
- Inversion Count Problem

7.1.6 Advantages of Merge Sort:

Stability: Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.

Guaranteed worst-case performance: Merge sort has a worst-case time complexity of O(N logN), which means it performs well even on large datasets.

Parallelizable: Merge sort is a naturally parallelizable algorithm, which means it can be easily parallelized to take advantage of multiple processors or threads.

7.1.7 Drawbacks of Merge Sort:

- **Space complexity:** Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
- **Not in-place:** Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.
- **Not always optimal for small datasets:** For small datasets, Merge sort has a higher time complexity than some other sorting algorithms, such as insertion sort. This can result in slower performance for very small datasets.

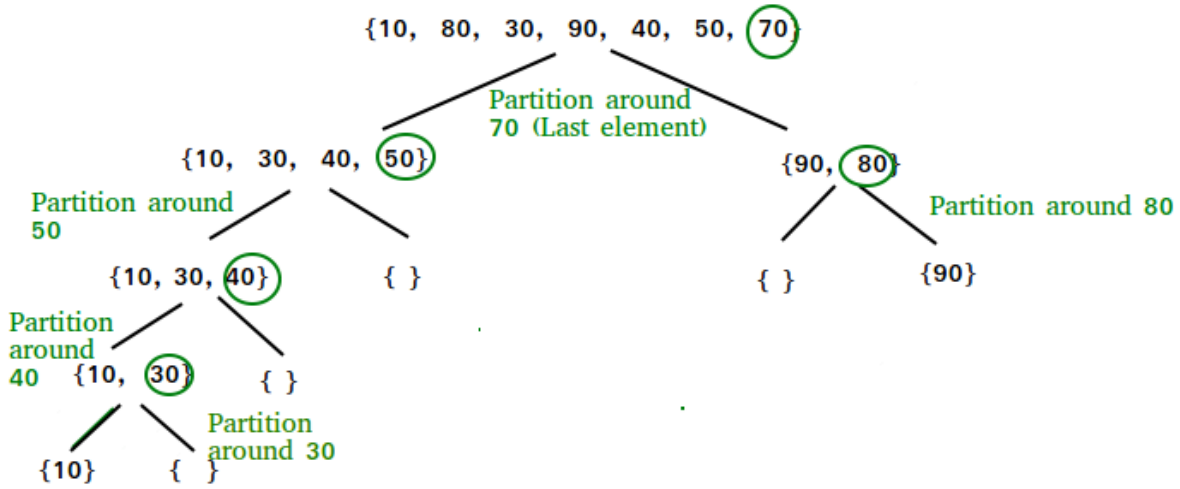
7.2 QuickSort – Data Structure and Algorithm C++

QuickSort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

7.2.1 How does QuickSort work?

The key process in ***quickSort*** is a ***partition()***. The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.

Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.



How Quicksort works

7.2.2 Choice of Pivot:

There are many different choices for picking pivots.

- Always pick the first element as a pivot.
- Always pick the last element as a pivot (implemented below)
- Pick a random element as a pivot.
- Pick the middle as the pivot.

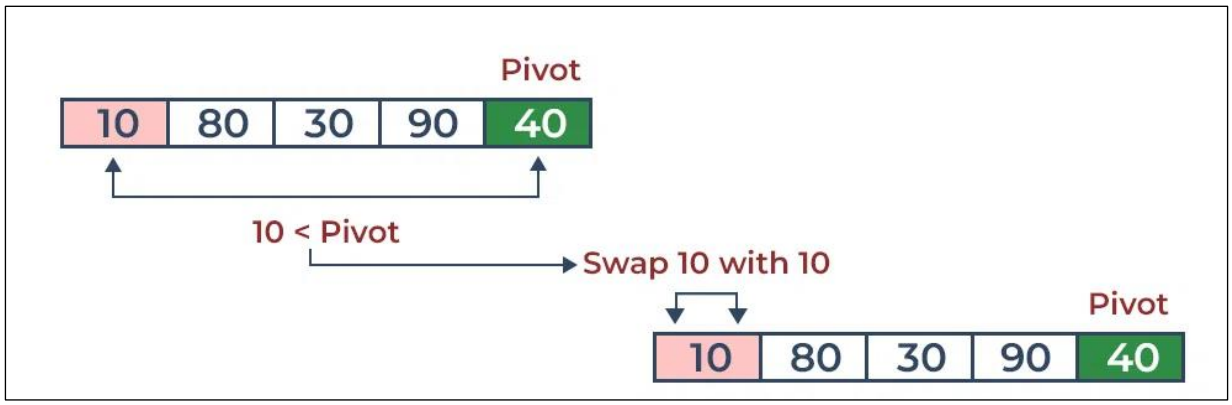
7.2.3 Partition Algorithm:

The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal) elements as ***i***. While traversing, if we find a smaller element, we swap the current element with arr[i]. Otherwise, we ignore the current element.

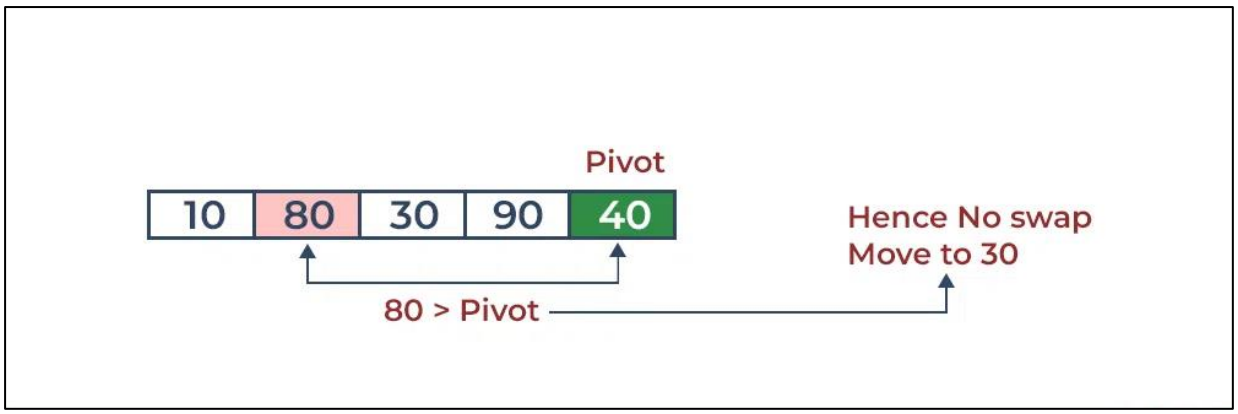
Let us understand the working of partition and the Quick Sort algorithm with the help of the following example:

Consider: arr[] = {10, 80, 30, 90, 40}.

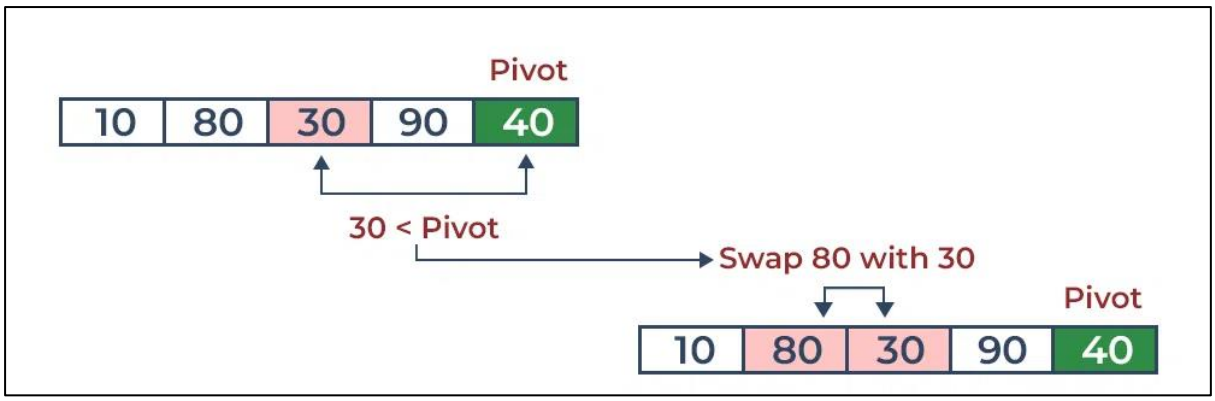
Compare 10 with the pivot and as it is less than pivot arrange it accordingly.



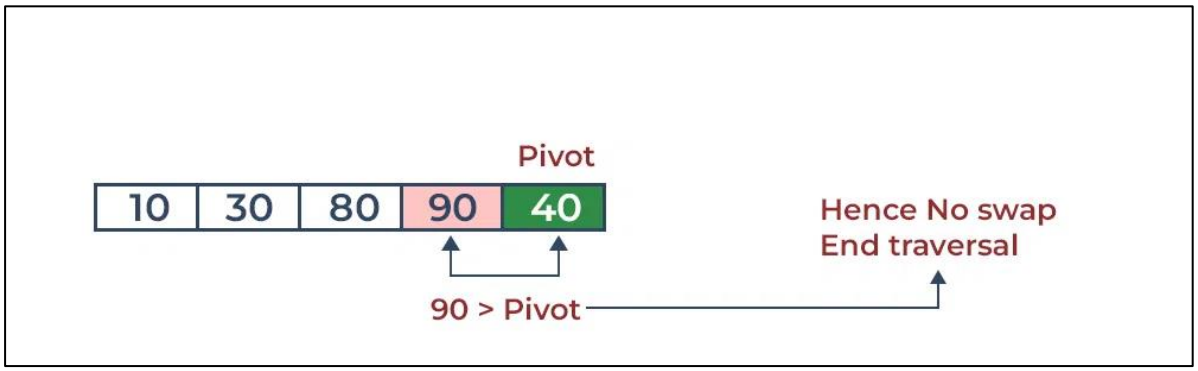
- Compare 80 with the pivot. It is greater than pivot.



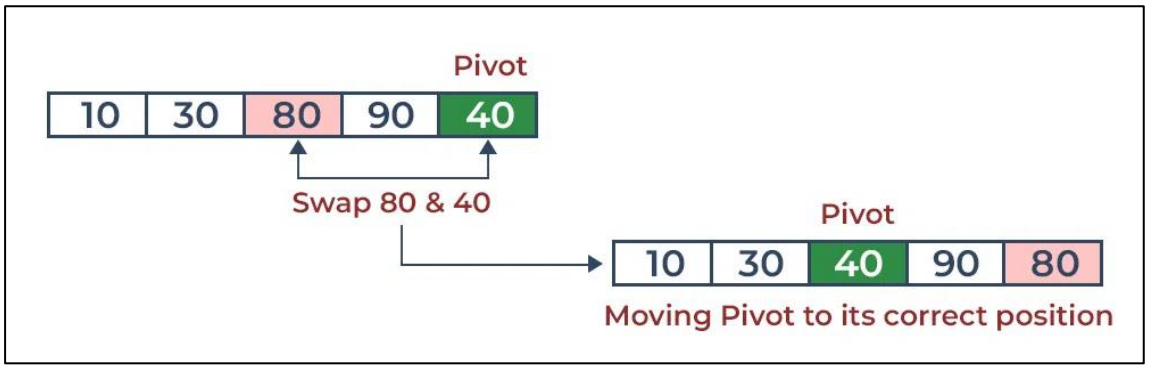
- Compare 30 with pivot. It is less than pivot so arrange it accordingly.



- Compare 90 with the pivot. It is greater than the pivot.



- Arrange the pivot in its correct position.

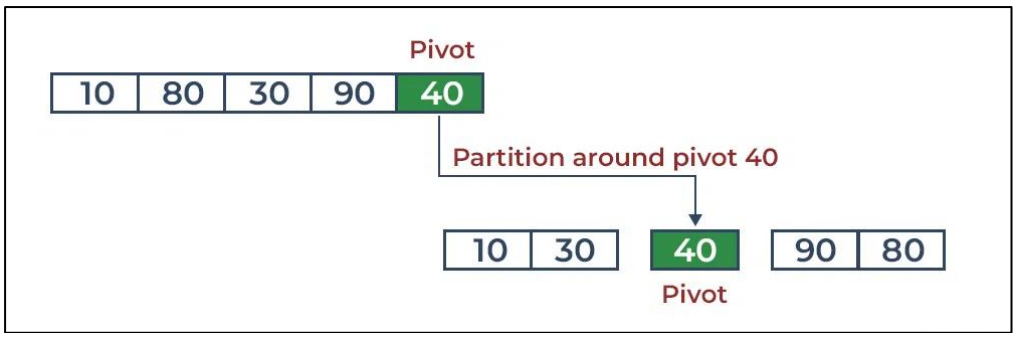


7.2.4 Illustration of Quicksort:

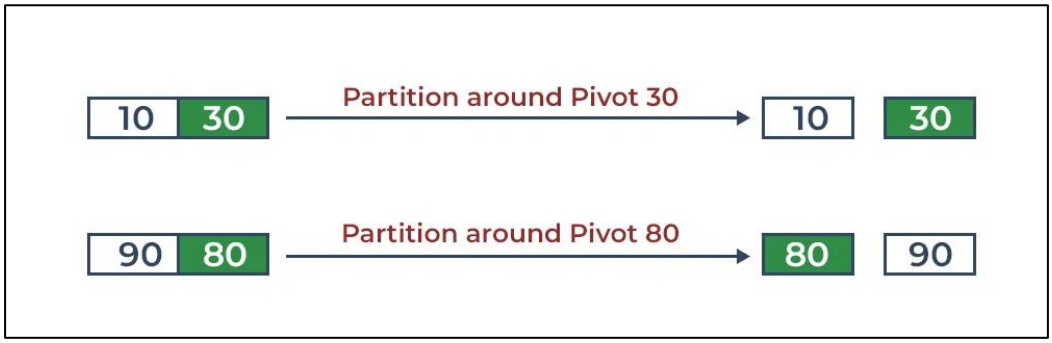
As the partition process is done recursively, it keeps on putting the pivot in its actual position in the sorted array. Repeatedly putting pivots in their actual position makes the array sorted.

Follow the below images to understand how the recursive implementation of the partition algorithm helps to sort the array.

Initial partition on the main array:



Partitioning of the subarrays:



7.2.5 Code implementation of the Quick Sort

Below is the implementation of the Quicksort:

```
// C++ code to implement quicksort

#include <bits/stdc++.h>
using namespace std;

// This function takes last element as pivot,
// places the pivot element at its correct position
// in sorted array, and places all smaller to left
// of pivot and all greater elements to right of pivot
int partition(int arr[], int low, int high)
{
    // Choosing the pivot
    int pivot = arr[high];

    // Index of smaller element and indicates
    // the right position of pivot found so far
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {

        // If current element is smaller than the pivot
        if (arr[j] < pivot) {
```

```
            // Increment index of smaller element
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}

// The main function that implements QuickSort
// arr[] --> Array to be sorted,
// low --> Starting index,
// high --> Ending index
void quickSort(int arr[], int low, int high)
{
    if (low < high) {

        // pi is partitioning index, arr[p]
        // is now at right place
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Driver Code
int main()
{
    int arr[] = { 10, 7, 8, 9, 1, 5 };
    int N = sizeof(arr) / sizeof(arr[0]);

    // Function call
    quickSort(arr, 0, N - 1);
    cout << "Sorted array: " << endl;
    for (int i = 0; i < N; i++)
        cout << arr[i] << " ";
    return 0;
}
```

Output

```
Sorted array:

1 5 7 8 9 10
```

7.2.6 Complexity Analysis of Quick Sort:

Time Complexity:

Best Case: $\omega(N * logN)$

Average Case: $\Theta(N * logN)$

Worst Case: $O(N^2)$

Auxiliary Space: O(1) as no extra space is used

7.2.7 Advantages of Quick Sort:

- It is a divide-and-conquer algorithm that makes it easier to solve problems.
- It is efficient on large data sets.
- It has a low overhead, as it only requires a small amount of memory to function.

7.2.8 Disadvantages of Quick Sort:

- It has a worst-case time complexity of $O(N^2)$, which occurs when the pivot is chosen poorly.
- It is not a good choice for small data sets.
- It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions).

Practice Problems:

Given a doubly linked list. Sort it using merge sort algorithm

List: 10->20->8-17->5->13->4

Sorted list: 4->5->8->10->13->17->20

Algorithm

1. If head is NULL or list contains only one element then return list

2. Create two lists by dividing original list into 2 parts

3. Sort first and second part of list

4. Merge both sorted list

Lab#08
Binary Search Tree Algorithm

Prepared by: Engr. Amna Arooj

FCSE, GIKI

Lab#08
Binary Search Tree (BST)

8.1 Binary Search Tree (BST)

To enhance the performance of binary tree, we use a special type of binary tree known as **Binary Search Tree**. Binary search tree mainly focuses on the search operation in a binary tree. Binary search tree can be defined as follows...

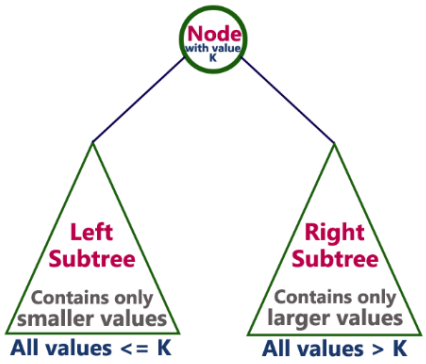
Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.

- It is called a binary tree because each tree node has a maximum of two children.
- It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.

8.1.1 Properties of BST:

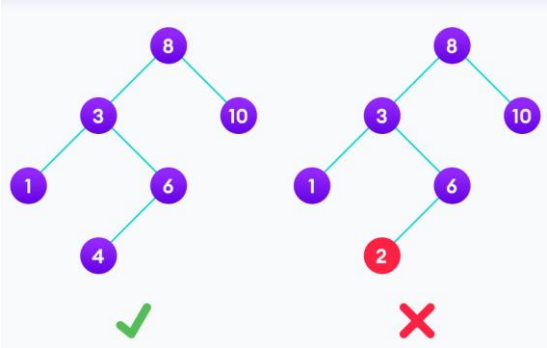
The properties that separate a binary search tree from a regular binary tree is

1. All nodes of left subtree are less than the root node (node with value K).
2. All nodes of right subtree are more than the root node(node with value K).
3. Both subtrees of each node are also BSTs i.e. they have the above two properties.



8.1.2 Example

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.



The binary tree on the right isn't a binary search tree because the right subtree of the node "3" contains a value smaller than it.

Note: Every binary search tree is a binary tree but every binary tree need not to be binary search tree.

8.2 Operations on a Binary Search Tree

The following operations are performed on a binary search tree...

1. **Search**
2. **Insertion**
3. **Deletion**

8.2.1 Search Operation in BST

In a binary search tree, the search operation is performed with **O(log n)** time complexity. The search operation is performed as follows...

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with the value of root node in the tree.

Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function

Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.

Step 5 - If search element is smaller, then continue the search process in left subtree.

Step 6- If search element is larger, then continue the search process in right subtree.

Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node

Step 8 - If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.

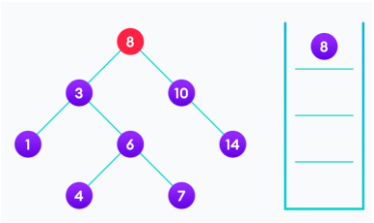
Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

8.2.1.1 Algorithm:

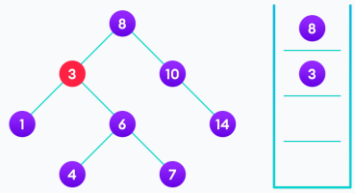
```
If root == NULL
   return NULL;
If number == root->data
   return root->data;
If number < root->data
   return search(root->left)
If number > root->data
   return search(root->right)
```

8.2.1.2 Illustration:

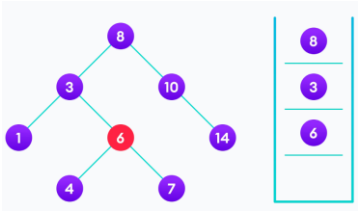
Let us try to visualize this with a diagram.



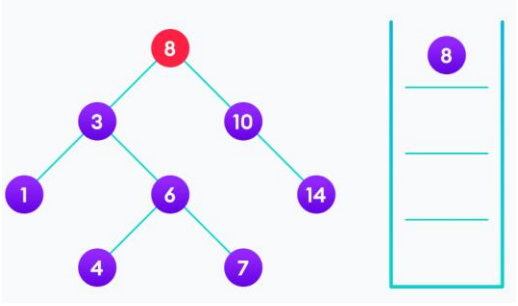
4 is not found so, traverse through the left subtree of 8



4 is not found so, traverse through the right subtree of 3



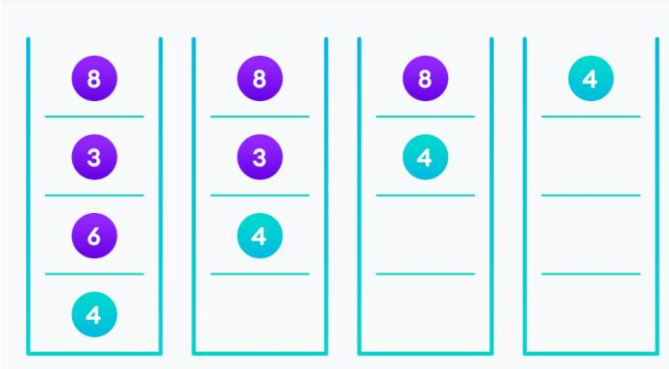
4 is not found so, traverse through the left subtree of 6



4 is found

If the value is found, we return the value so that it gets propagated in each recursion step as shown in the image below.

If you might have noticed, we have called return search(struct node*) four times. When we return either the new node or NULL, the value gets returned again and again until search(root) returns the final result.



If the value is found in any of the subtrees, it is propagated up so that in the end it is returned, otherwise null is returned.

If the value is not found, we eventually reach the left or right child of a leaf node which is NULL and it gets propagated and returned.

8.2.2 Insertion Operation in BST

In a binary search tree, the insertion operation is performed with **O(log n)** time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1 - Create a newNode with given value and set its **left** and **right** to **NULL**.

Step 2 - Check whether tree is Empty.

Step 3 - If the tree is **Empty**, then set **root** to **newNode**.

Step 4 - If the tree is **Not Empty**, then check whether the value of newNode is **smaller** or **larger** than the node (here it is root node).

Step 5 - If newNode is **smaller** than **or equal** to the node then move to its **left** child. If newNode is **larger** than the node then move to its **right** child.

Step 6- Repeat the above steps until we reach to the **leaf** node (i.e., reaches to NULL).

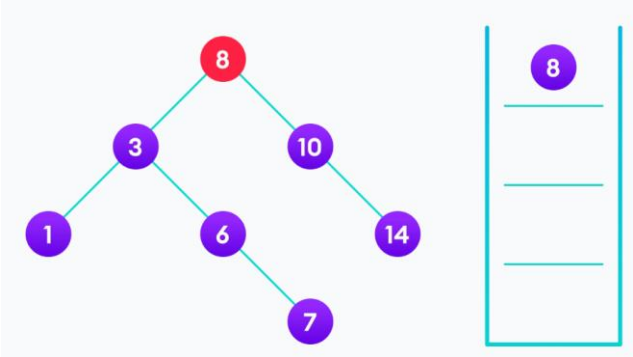
Step 7 - After reaching the leaf node, insert the newNode as **left child** if the newNode is **smaller or equal** to that leaf node or else insert it as **right child**.

8.2.2.1 Algorithm:

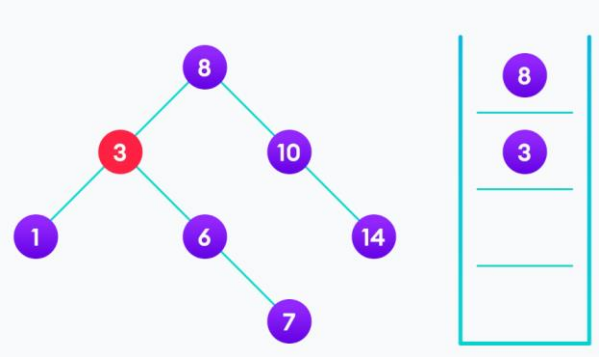
```
If node == NULL
   return createNode(data)
if (data < node->data)
   node->left  = insert(node->left, data);
else if (data > node->data)
   node->right = insert(node->right, data);
return node;
```

8.2.2.2 Illustration:

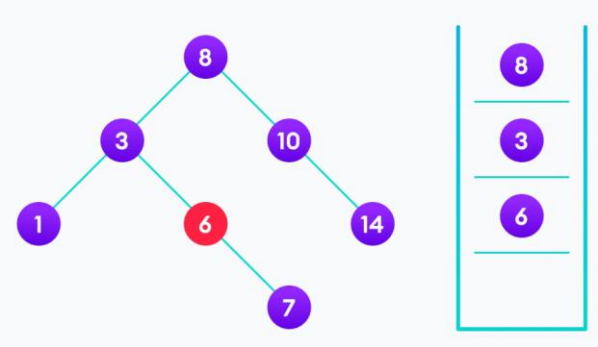
The algorithm isn't as simple as it looks. Let's try to visualize how we add a number to an existing BST.



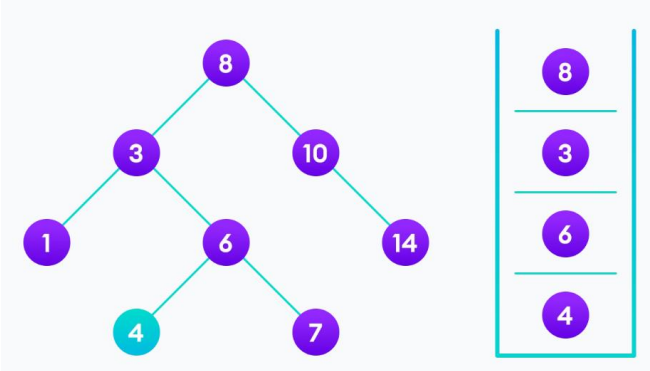
4<8 so, transverse through the left child of 8



4>3 so, transverse through the right child of 8



4<6 so, transverse through the left child of 6



Insert 4 as a left child of 6

We have attached the node but we still have to exit from the function without doing any damage to the rest of the tree. This is where the `return node;` at the end comes in handy. In the case of `NULL`, the newly created node is returned and attached to the parent node, otherwise the same node is returned without any change as we go up until we return to the root.

This makes sure that as we move back up the tree, the other node connections aren't changed.

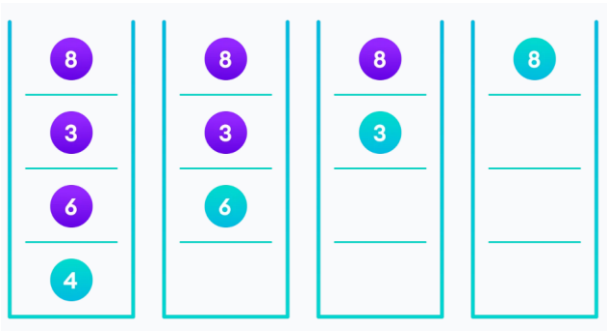


Image showing the importance of returning the root element at the end so that the elements don't lose their position during the upward recursion step.

8.2.3 Deletion Operation in BST

In a binary search tree, the deletion operation is performed with **O(log n)** time complexity. Deleting a node from Binary search tree includes following three cases...

- **Case 1: Deleting a Leaf node (A node with no children)**
- **Case 2: Deleting a node with one child**
- **Case 3: Deleting a node with two children**

Case 1: Deleting a leaf node

We use the following steps to delete a leaf node from BST...

- **Step 1 - Find** the node to be deleted using **search operation**
- **Step 2 -** Delete the node using **free** function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...

- **Step 1 - Find** the node to be deleted using **search operation**
- **Step 2 -** If it has only one child then create a link between its parent node and child node.
- **Step 3 -** Delete the node using **free** function and terminate the function.

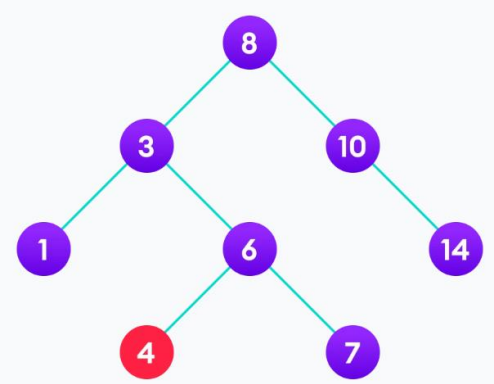
Case 3: Deleting a node with two children

We use the following steps to delete a node with two children from BST...

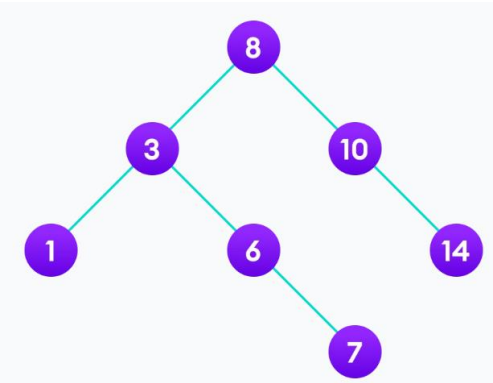
- **Step 1 - Find** the node to be deleted using **search operation**
- **Step 2 -** If it has two children, then find the **largest** node in its **left subtree** (OR) the **smallest** node in its **right subtree**.
- **Step 3 - Swap** both **deleting node** and node which is found in the above step.
- **Step 4 -** Then check whether deleting node came to **case 1** or **case 2** or else goto step 2
- **Step 5 -** If it comes to **case 1**, then delete using case 1 logic.
- **Step 6-** If it comes to **case 2**, then delete using case 2 logic.
- **Step 7 -** Repeat the same process until the node is deleted from the tree.

Case I:

In the first case, the node to be deleted is the leaf node. In such a case, simply delete the node from the tree.



4 is to be deleted

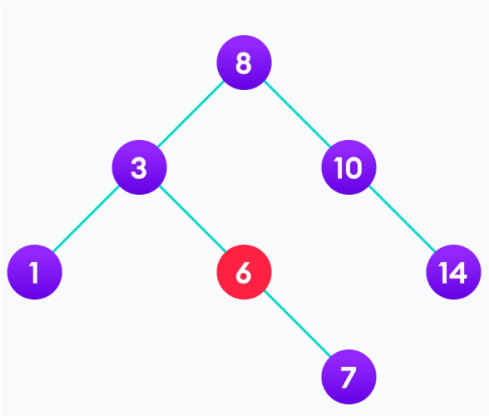


Delete the node

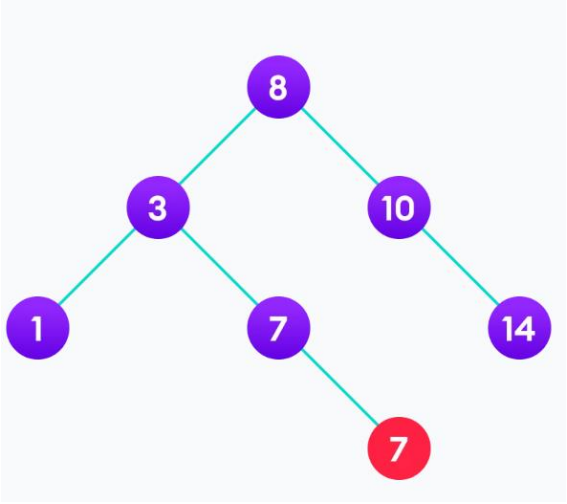
Case II

In the second case, the node to be deleted lies has a single child node. In such a case follow the steps below:

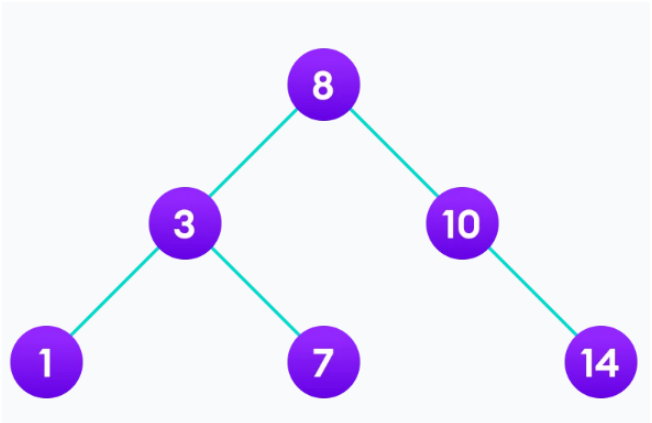
1. Replace that node with its child node.
2. Remove the child node from its original position.



6 is to be deleted



copy the value of its child to the node and delete the child

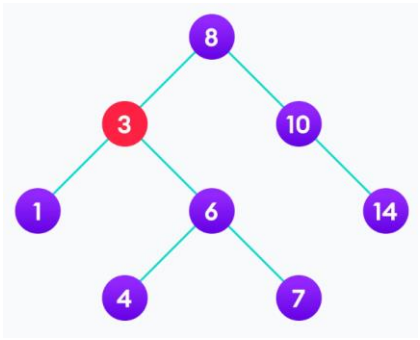


Final tree

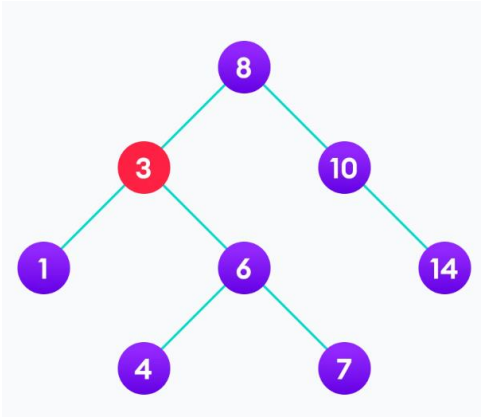
Case III

In the third case, the node to be deleted has two children. In such a case follow the steps below:

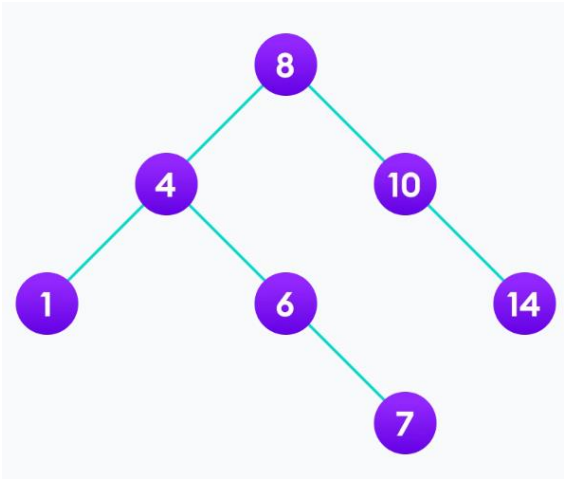
1. Get the inorder successor of that node.
2. Replace the node with the inorder successor.
3. Remove the inorder successor from its original position.



3 is to be deleted



Copy the value of the inorder successor (4) to the node



Delete the inorder successor

8.3 Implementation in C++

```
// Binary Search Tree operations in C++

#include <iostream>
using namespace std;

struct node {
  int key;
  struct node *left, *right;
};

// Create a node
struct node *newNode(int item) {
  struct node *temp = (struct node *)malloc(sizeof(struct node));
  temp->key = item;
  temp->left = temp->right = NULL;
  return temp;
}

// Inorder Traversal
void inorder(struct node *root) {
  if (root != NULL) {
    // Traverse left
    inorder(root->left);

    // Traverse root
    cout << root->key << " -> ";

    // Traverse right
    inorder(root->right);
  }
}

// Insert a node
struct node *insert(struct node *node, int key) {
  // Return a new node if the tree is empty
  if (node == NULL) return newNode(key);

  // Traverse to the right place and insert the node
  if (key < node->key)
    node->left = insert(node->left, key);
  else
    node->right = insert(node->right, key);

  return node;
}

// Find the inorder successor
```

```
struct node *minValueNode(struct node *node) {
  struct node *current = node;

  // Find the leftmost leaf
  while (current && current->left != NULL)
    current = current->left;

  return current;
}

// Deleting a node
struct node *deleteNode(struct node *root, int key) {
  // Return if the tree is empty
  if (root == NULL) return root;

  // Find the node to be deleted
  if (key < root->key)
    root->left = deleteNode(root->left, key);
  else if (key > root->key)
    root->right = deleteNode(root->right, key);
  else {
    // If the node is with only one child or no child
    if (root->left == NULL) {
      struct node *temp = root->right;
      free(root);
      return temp;
    } else if (root->right == NULL) {
      struct node *temp = root->left;
      free(root);
      return temp;
    }

    // If the node has two children
    struct node *temp = minValueNode(root->right);

    // Place the inorder successor in position of the node to be deleted
    root->key = temp->key;

    // Delete the inorder successor
    root->right = deleteNode(root->right, temp->key);
  }
  return root;
}

// Driver code
int main() {
  struct node *root = NULL;
  root = insert(root, 8);
  root = insert(root, 3);
  root = insert(root, 1);
```

```
  root = insert(root, 6);
  root = insert(root, 7);
  root = insert(root, 10);
  root = insert(root, 14);
  root = insert(root, 4);

  cout << "Inorder traversal: ";
  inorder(root);

  cout << "\nAfter deleting 10\n";
  root = deleteNode(root, 10);
  cout << "Inorder traversal: ";
  inorder(root);
}
```

8.4 Binary Search Tree Complexities

Time Complexity

Operation	Best Case Complexity	Average Case Complexity	Worst Case Complexity
Search	O(log n)	O(log n)	O(n)
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)

Here, `n` is the number of nodes in the tree.

Space Complexity

The space complexity for all the operations is `O(n)`.

8.5 Binary Search Tree Applications

- In multilevel indexing in the database
- For dynamic sorting
- For managing virtual memory areas in Unix kerne

Real-time Application of Binary Search tree:

- BSTs are used for indexing in databases.
- It is used to implement searching algorithms.
- BSTs are used to implement Huffman coding algorithm.
- It is also used to implement dictionaries.
- Used for data caching.
- Used in Priority queues.
- Used in spell checkers.

8.6 Advantages of Binary Search Tree:

- BST is fast in insertion and deletion when balanced. It is fast with a time complexity of O(log n).
- BST is also for fast searching, with a time complexity of O(log n) for most operations.
- BST is efficient. It is efficient because they only store the elements and do not require additional memory for pointers or other data structures.
- We can also do range queries – find keys between N and M (N <= M).

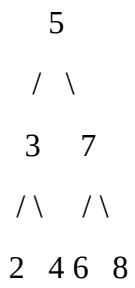
8.6.1 Disadvantages of Binary Search Tree:

1. The main disadvantage is that we should always implement a balanced binary search tree. Otherwise the cost of operations may not be logarithmic and degenerate into a linear search on an array.
2. They are not well-suited for data structures that need to be accessed randomly, since the time complexity for search, insert, and delete operations is O(log n), which is good for large data sets, but not as fast as some other data structures such as arrays or hash tables.
3. A BST can be imbalanced or degenerated which can increase the complexity.

Practice Problems:

Lab Activity #01:

Write a C++ program to implement a binary search tree (BST) given below and perform the following operations:



1. Insert a new node into the BST.
2. Search for a given value in the BST.
3. Delete a node from the BST.
4. Print the elements of the BST in ascending order.

You can use any preferred data structure or approach to implement the BST.

Lab#09
AVL Trees

Prepared by: Engr. Amna Arooj

FCSE, GIKI

9.1 AVL Tree

In this tutorial, you will learn what an avl tree is. Also, you will find working examples of various operations performed on an avl tree in C, C++, Java and Python.

AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.

AVL tree got its name after its inventor Georgy Adelson-Velsky and Landis.

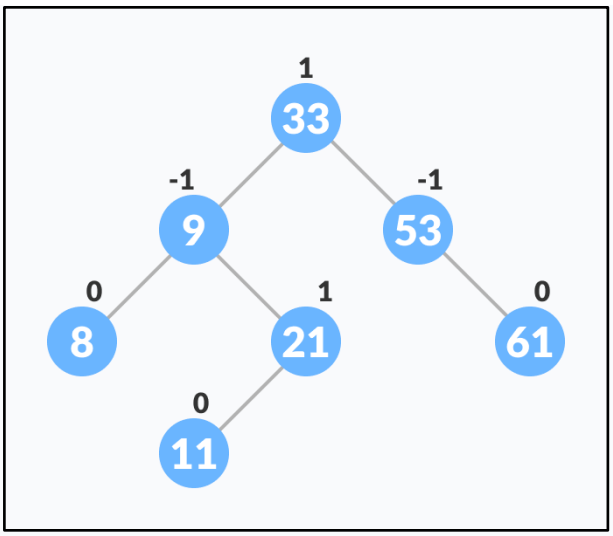
9.2 AVL Tree Balance Factor

Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.

Balance Factor = (Height of Left Subtree - Height of Right Subtree) or (Height of Right Subtree - Height of Left Subtree)

The self balancing property of an avl tree is maintained by the balance factor. The value of balance factor should always be -1, 0 or +1.

An example of a balanced avl tree is:



Avl tree

9.3 Operations on an AVL tree

Various operations that can be performed on an AVL tree are:

9.3.1 Rotating the subtrees in an AVL Tree

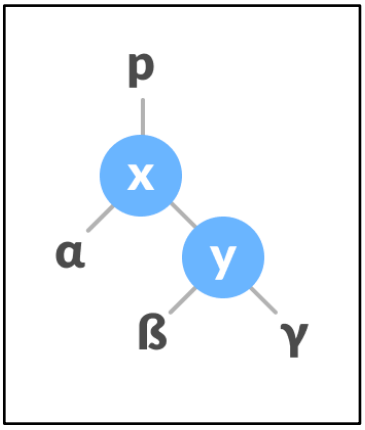
In rotation operation, the positions of the nodes of a subtree are interchanged.

There are two types of rotations:

Left Rotate

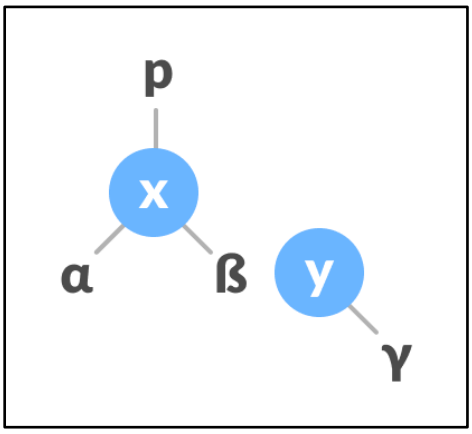
In left-rotation, the arrangement of the nodes on the right is transformed into the arrangements on the left node.

Algorithm



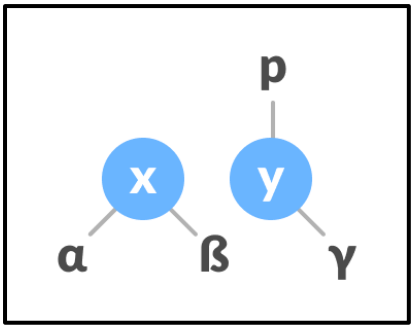
Left rotate

1. Let the initial tree be like above.
2. If `y` has a left subtree, assign `x` as the parent of the left subtree of `y`.



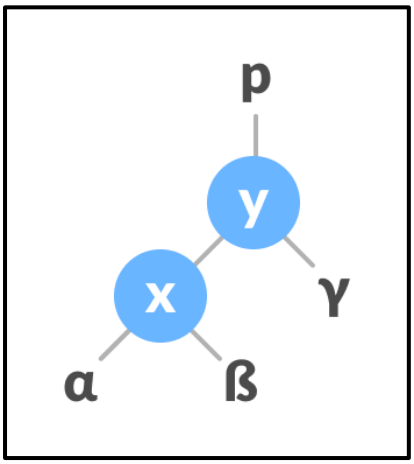
Assign x as the parent of the left subtree of y

3. If the parent of `x` is `NULL`, make `y` as the root of the tree.
4. Else if `x` is the left child of `p`, make `y` as the left child of `p`.
5. Else assign `y` as the right child of `p`.



Change the parent of x to that of y

6. Make y as the parent of x.

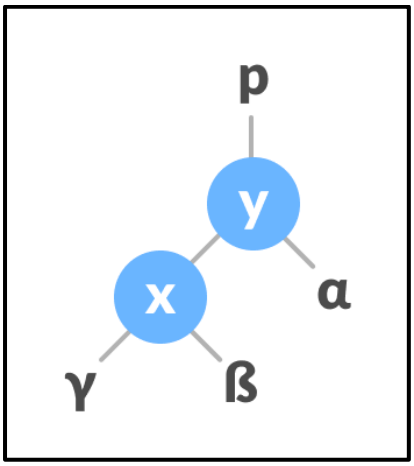


Assign y as the parent of x.

Right Rotate

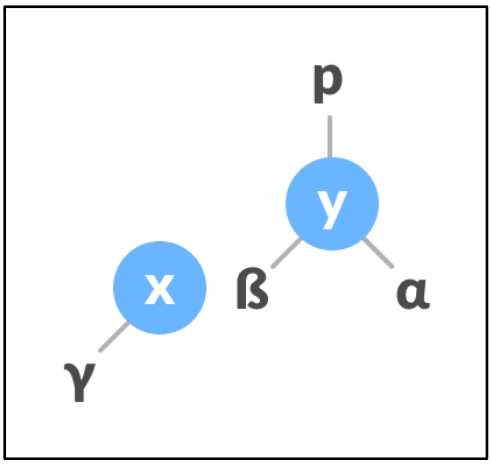
In left-rotation, the arrangement of the nodes on the left is transformed into the arrangements on the right node.

1. Let the initial tree be:



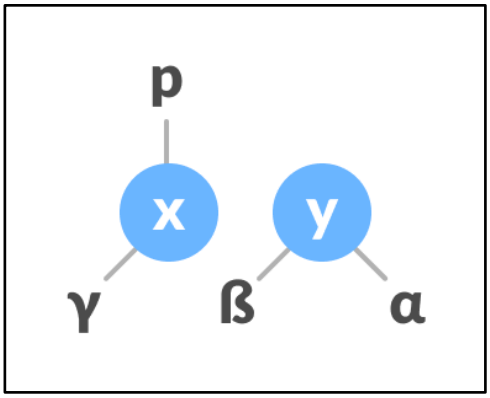
Initial tree

2. If x has a right subtree, assign y as the parent of the right subtree of x.



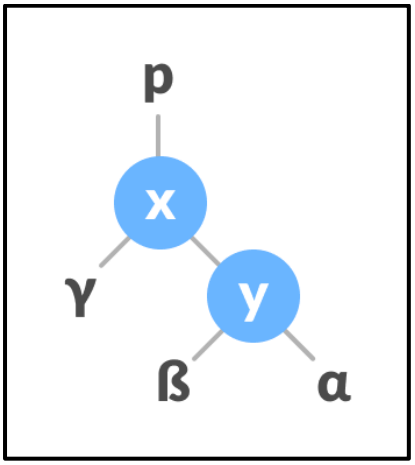
Assign y as the parent of the right subtree of x

3. If the parent of `y` is `NULL`, make `x` as the root of the tree.
4. Else if `y` is the right child of its parent `p`, make `x` as the right child of `p`.
5. Else assign `x` as the left child of `p`.



Assign the parent of y as the parent of x.

6. Make `x` as the parent of `y`.

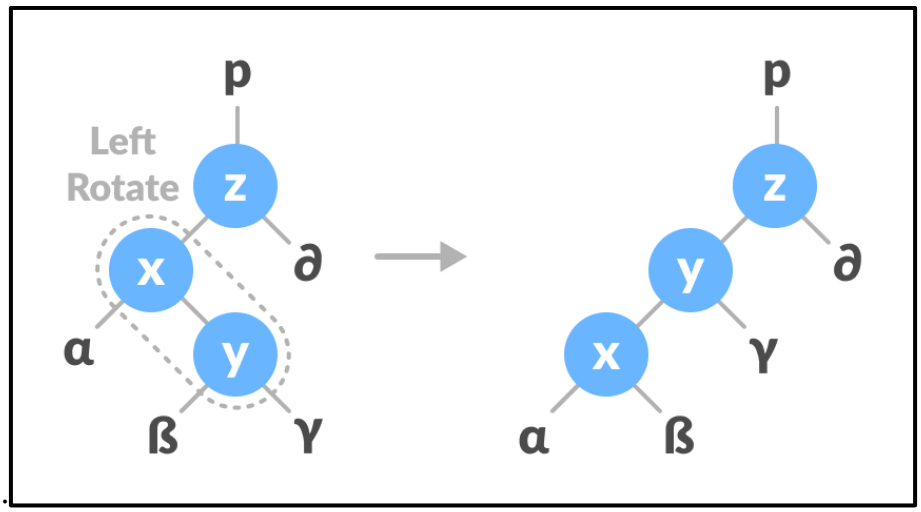


Assign x as the parent of y

Left-Right and Right-Left Rotate

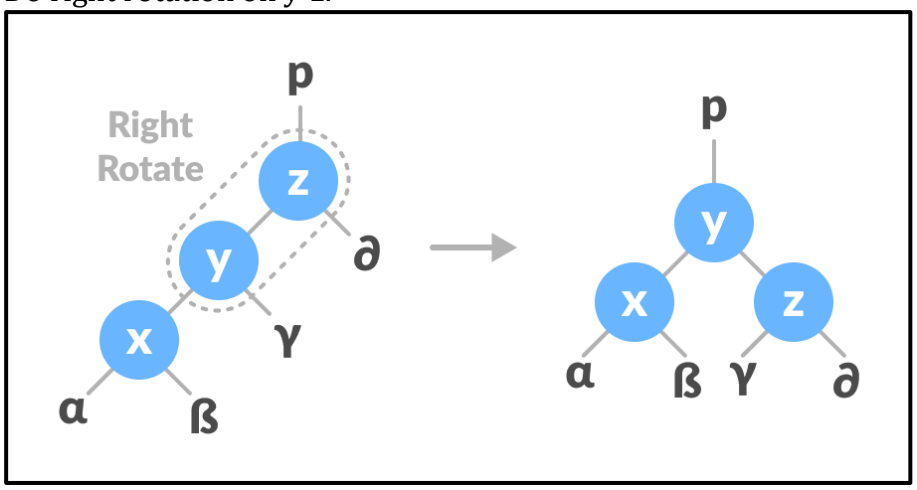
In left-right rotation, the arrangements are first shifted to the left and then to the right.

1. Do left rotation on x-y



Left rotate x-y

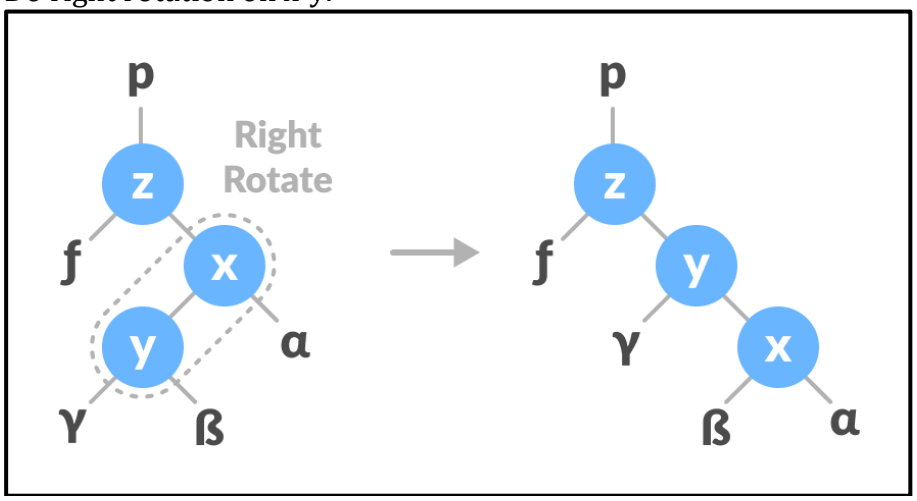
2. Do right rotation on y-z.



Right rotate z-y

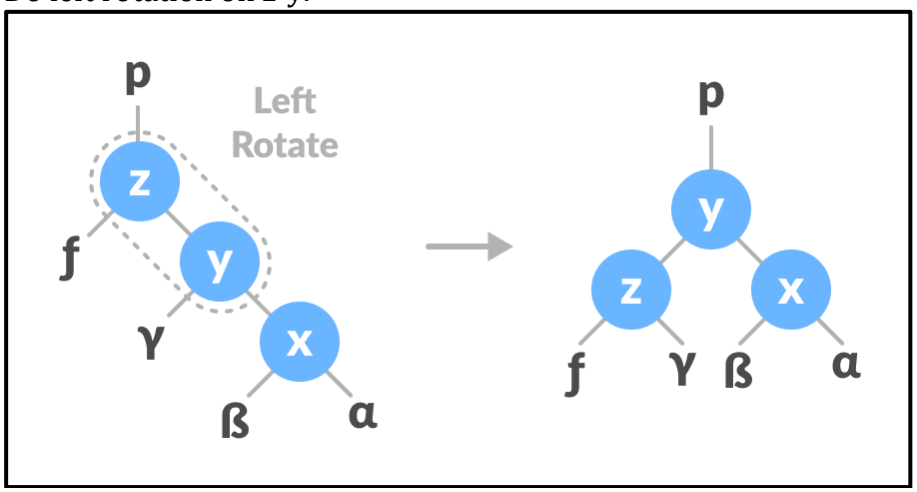
In right-left rotation, the arrangements are first shifted to the right and then to the left.

1. Do right rotation on x-y.



Right rotate x-y

2. Do left rotation on z-y.

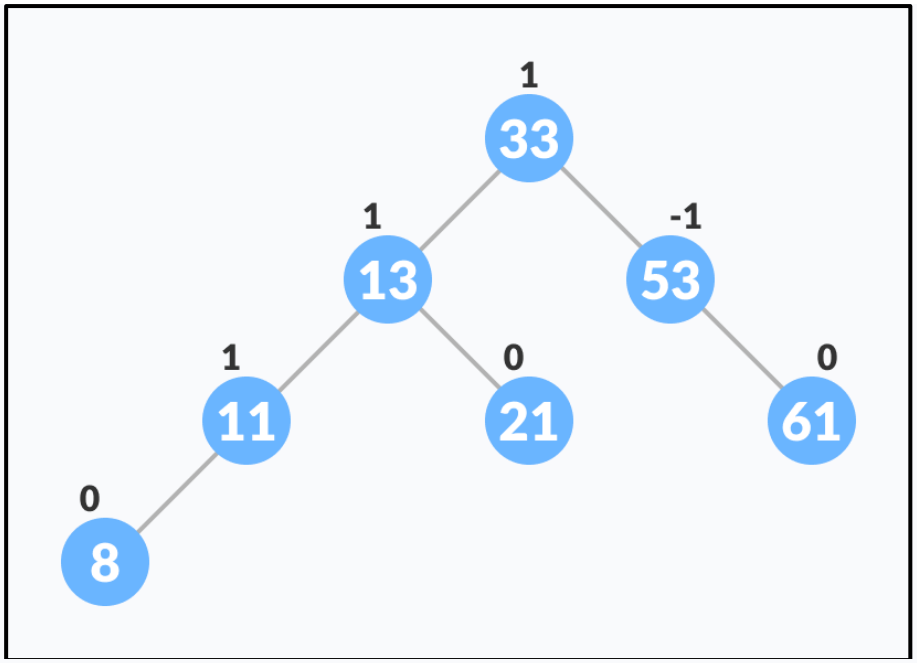


Left rotate z-y

9.3.2 Algorithm to insert a newNode

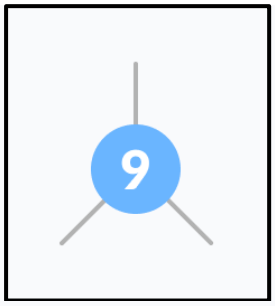
A `newNode` is always inserted as a leaf node with balance factor equal to 0.

a) Let the initial tree be:



Initial tree for insertion

Let the node to be inserted be:



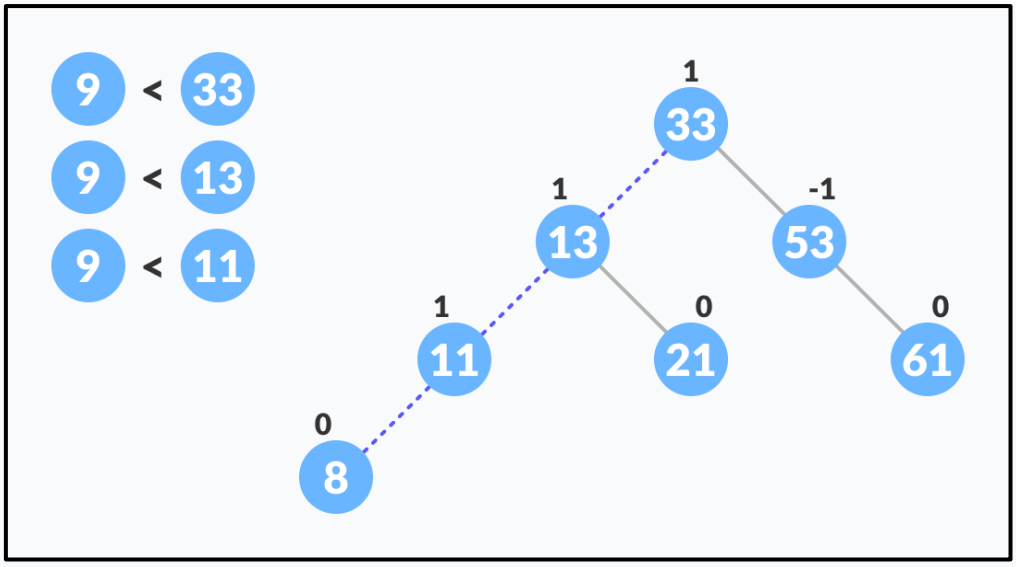
New node

b) Go to the appropriate leaf node to insert a `newNode` using the following recursive steps. Compare `newKey` with `rootKey` of the current tree.

a) If `newKey` < `rootKey`, call insertion algorithm on the left subtree of the current node until the leaf node is reached.

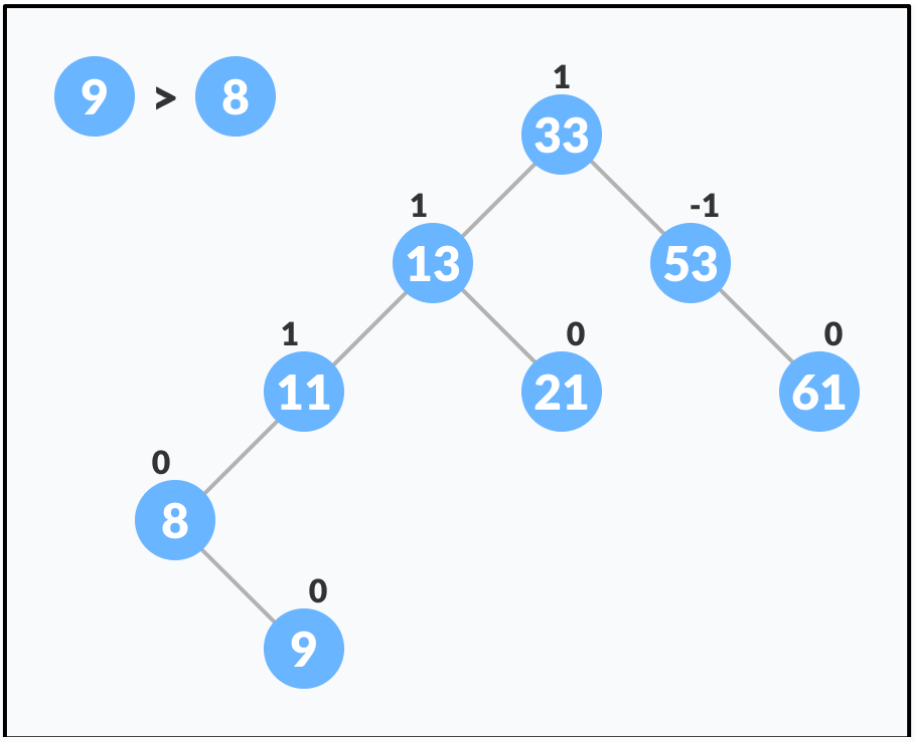
b) Else if `newKey` > `rootKey`, call insertion algorithm on the right subtree of current node until the leaf node is reached.

c) Else, return leafNode.



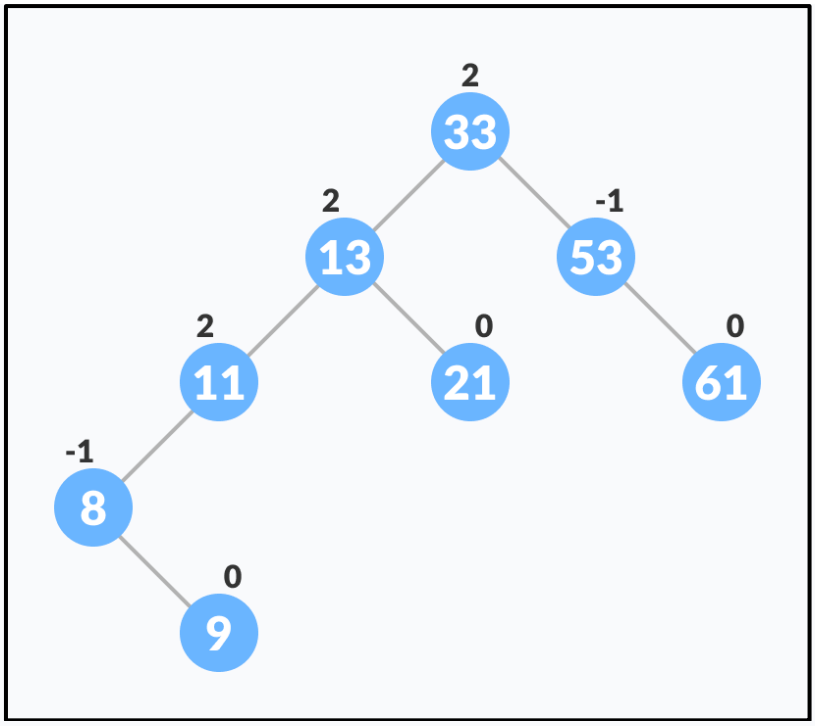
Finding the location to insert newNode

c) Compare `leafKey` obtained from the above steps with `newKey`:
a) If `newKey` < `leafKey`, make newNode as the `leftChild` of `leafNode`.
b) Else, make `newNode` as rightChild of `leafNode`.



Inserting the new node

d) Update `balanceFactor` of the nodes.

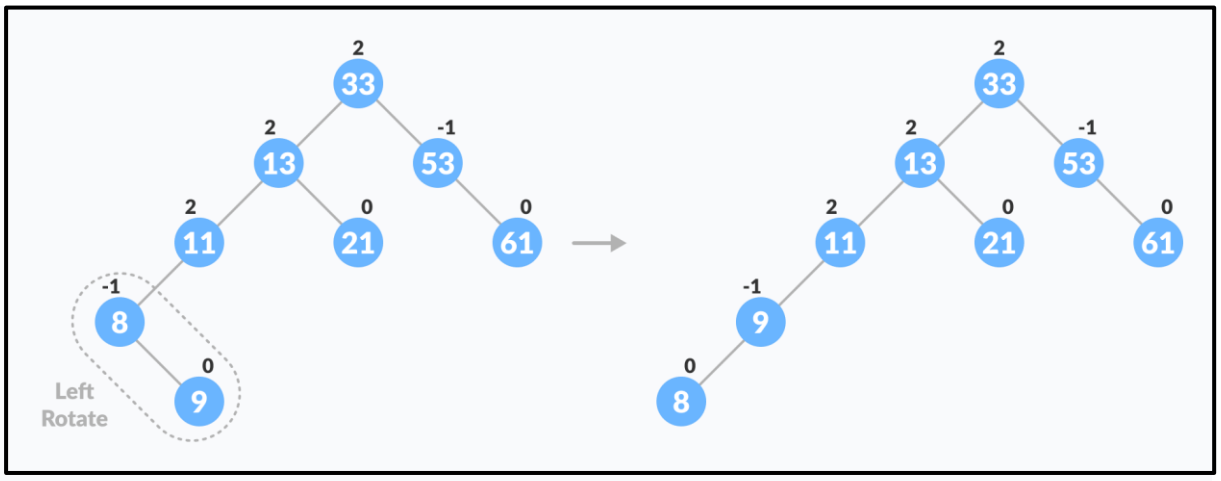


Updating the balance factor after insertion

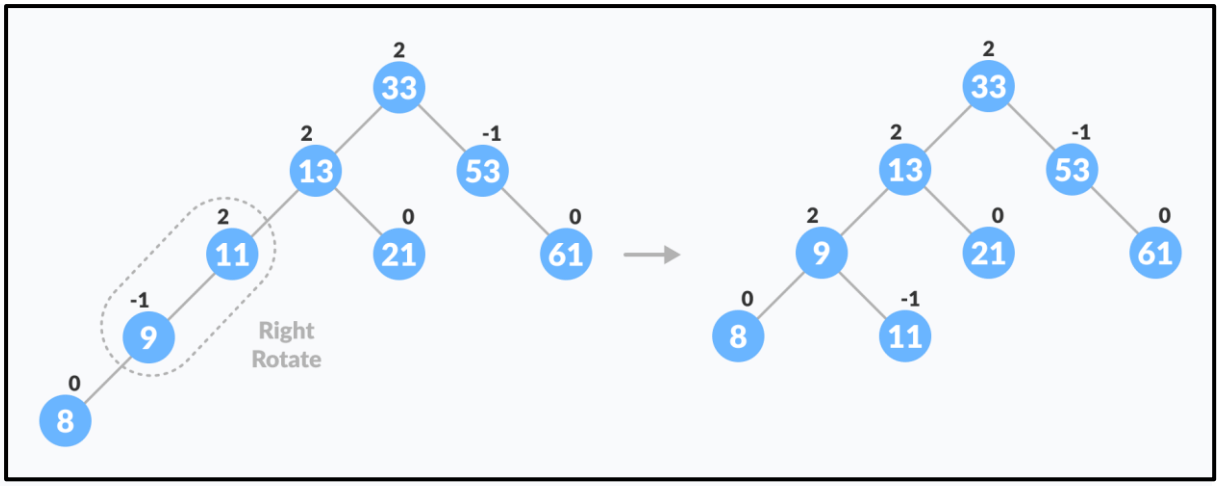
e) If the nodes are unbalanced, then rebalance the node.

a. If `balanceFactor` > 1, it means the height of the left subtree is greater than that of the right subtree. So, do a right rotation or left-right rotation

a. If `newNodeKey` < `leftChildKey` do right rotation.
b. Else, do left-right rotation.



Balancing the tree with rotation



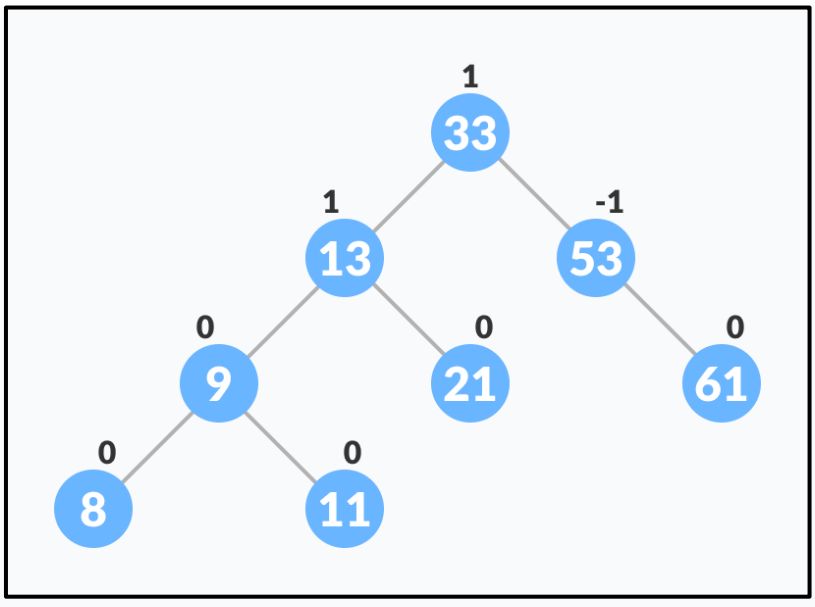
Balancing the tree with rotation

b. If `balanceFactor` < -1, it means the height of the right subtree is greater than that of the left subtree. So, do right rotation or right-left rotation

a. If `newNodeKey` > `rightChildKey` do left rotation.

b. Else, do right-left rotation

f) The final tree is:

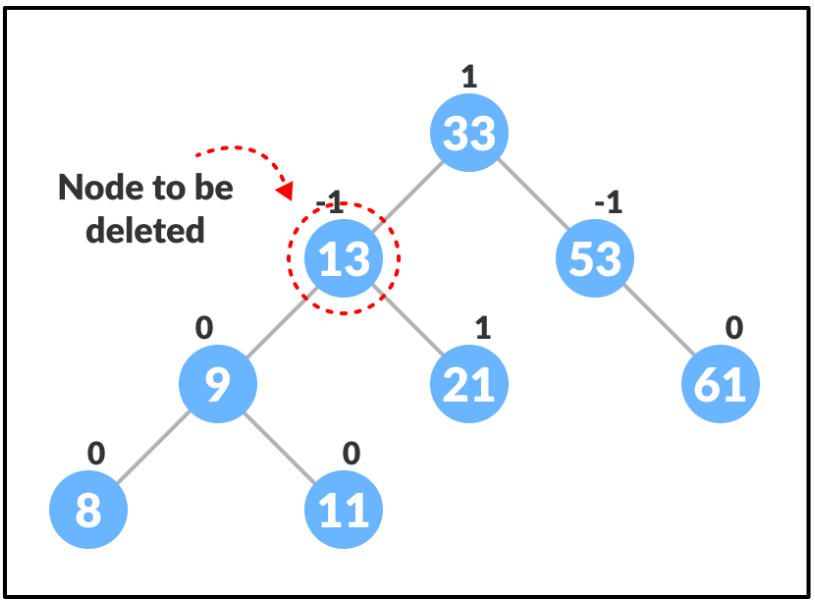


Final balanced tree

9.4 Algorithm to Delete a node

A node is always deleted as a leaf node. After deleting a node, the balance factors of the nodes get changed. In order to rebalance the balance factor, suitable rotations are performed.

1. Locate `nodeToBeDeleted` (recursion is used to find `nodeToBeDeleted` in the code used below).



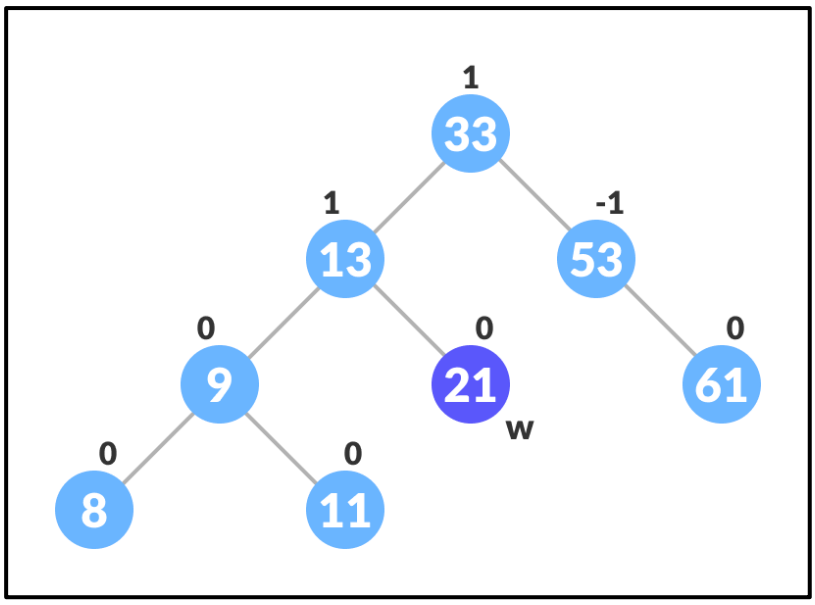
Locating the node to be deleted

2. There are three cases for deleting a node:

a) If `nodeToBeDeleted` is the leaf node (ie. does not have any child), then remove `nodeToBeDeleted`.

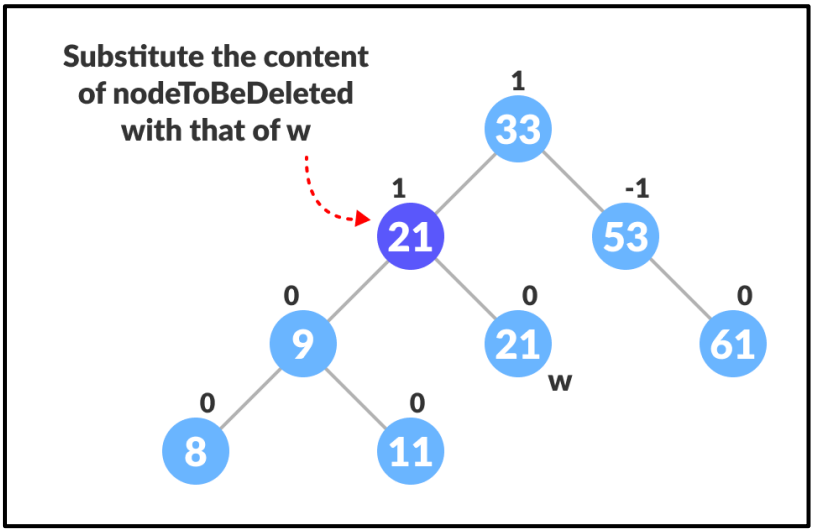
b) If `nodeToBeDeleted` has one child, then substitute the contents of `nodeToBeDeleted` with that of the child. Remove the child.

c) If `nodeToBeDeleted` has two children, find the inorder successor `w` of `nodeToBeDeleted` (ie. node with a minimum value of key in the right subtree).



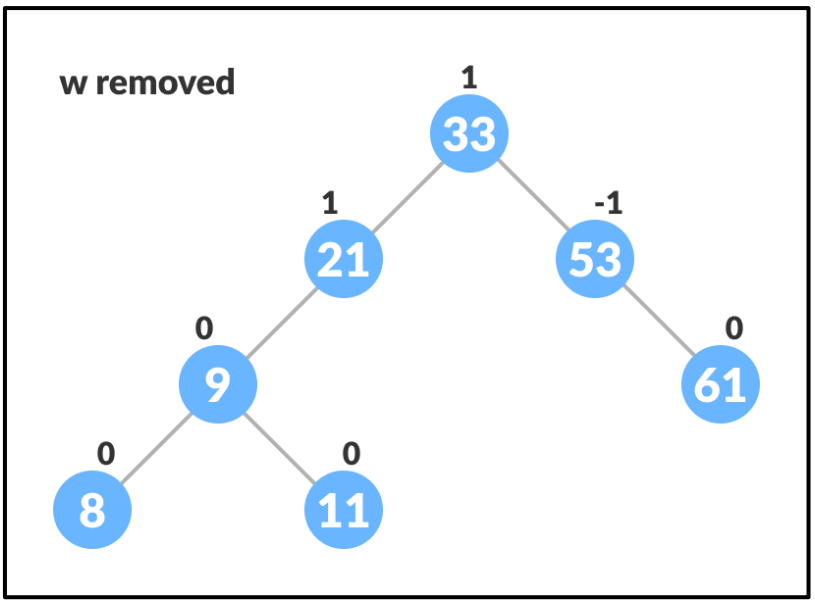
Finding the successor

a) Substitute the contents of `nodeToBeDeleted` with that of `w`.



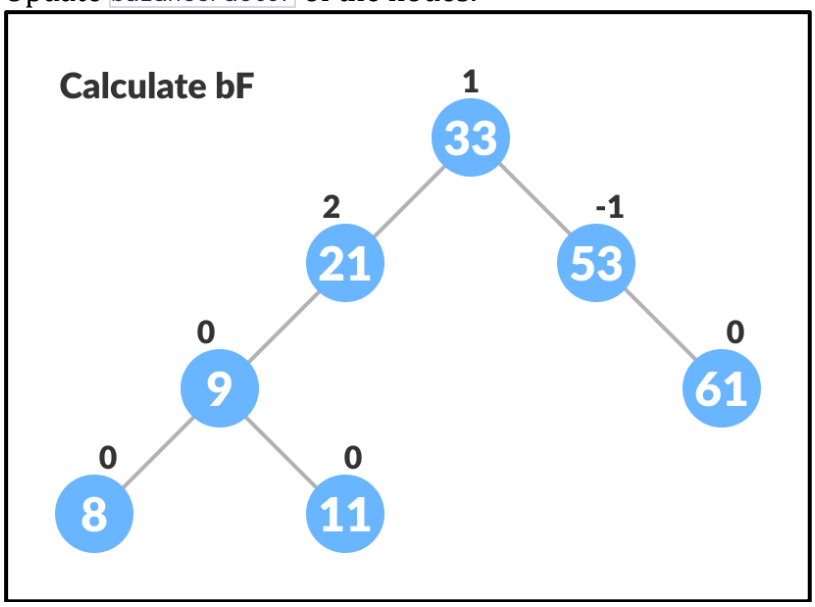
Substitute the node to be deleted

b) Remove the leaf node `w`.



Remove w

3. Update `balanceFactor` of the nodes.

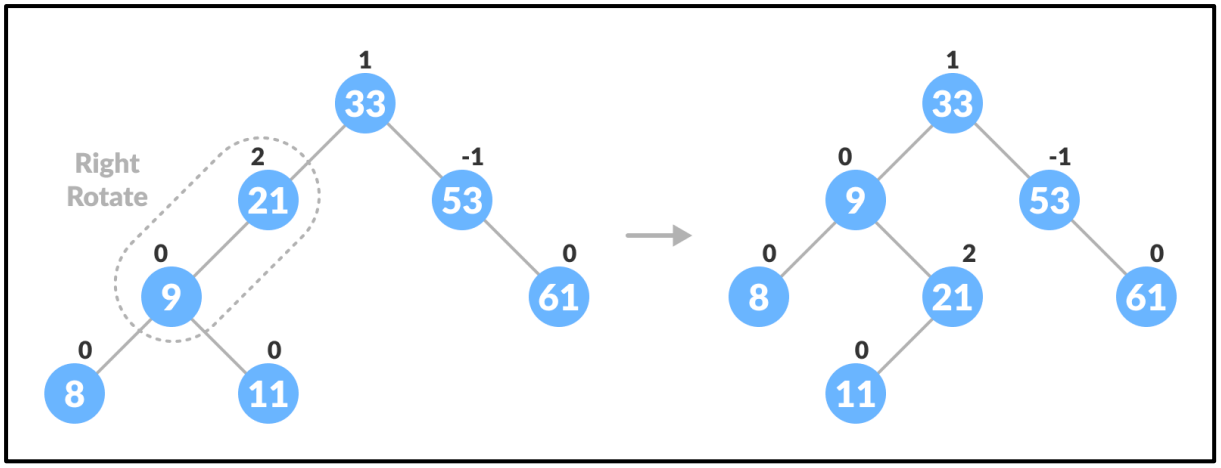


Update bf

4. Rebalance the tree if the balance factor of any of the nodes is not equal to -1, 0 or 1.

a) If `balanceFactor` of `currentNode` > 1,

i) If `balanceFactor` of `leftChild` >= 0, do right rotation.



Right-rotate for balancing the tree

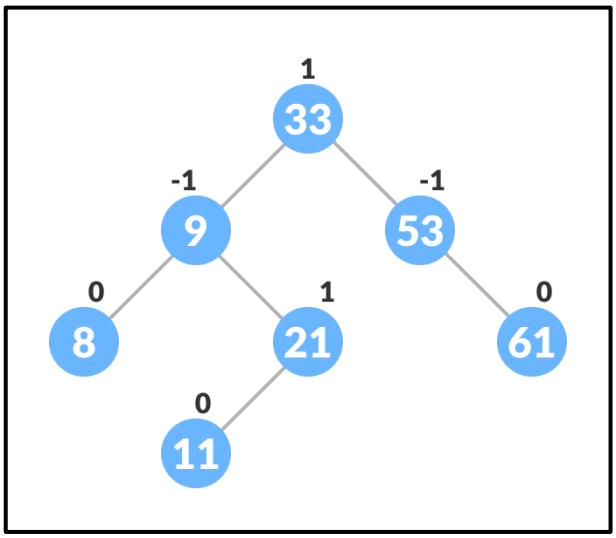
ii) Else do left-right rotation.

b) If `balanceFactor` of `currentNode` < -1,

a. If `balanceFactor` of `rightChild` <= 0, do left rotation.

b. Else do right-left rotation.

5. The final tree is:



Avl tree final

9.5 AVL Tree Implementation in C++

```
// AVL tree implementation in C++


#include <iostream>

using namespace std;
```

```
class Node {
  public:
  int key;
  Node *left;
  Node *right;
  int height;
};

int max(int a, int b);

// Calculate height
int height(Node *N) {
  if (N == NULL)
    return 0;
  return N->height;
}

int max(int a, int b) {
  return (a > b) ? a : b;
}

// New node creation
Node *newNode(int key) {
  Node *node = new Node();
  node->key = key;
  node->left = NULL;
  node->right = NULL;
  node->height = 1;
  return (node);
```

```
}

// Rotate right
Node *rightRotate(Node *y) {
  Node *x = y->left;
  Node *T2 = x->right;
  x->right = y;
  y->left = T2;
  y->height = max(height(y->left),
          height(y->right)) +
        1;
  x->height = max(height(x->left),
          height(x->right)) +
        1;
  return x;
}

// Rotate left
Node *leftRotate(Node *x) {
  Node *y = x->right;
  Node *T2 = y->left;
  y->left = x;
  x->right = T2;
  x->height = max(height(x->left),
          height(x->right)) +
        1;
  y->height = max(height(y->left),
          height(y->right)) +
        1;
  return y;
```

```
}

// Get the balance factor of each node
int getBalanceFactor(Node *N) {
  if (N == NULL)
    return 0;
  return height(N->left) -
      height(N->right);
}

// Insert a node
Node *insertNode(Node *node, int key) {
  // Find the correct postion and insert the node
  if (node == NULL)
    return (newNode(key));
  if (key < node->key)
    node->left = insertNode(node->left, key);
  else if (key > node->key)
    node->right = insertNode(node->right, key);
  else
    return node;

  // Update the balance factor of each node and
  // balance the tree
  node->height = 1 + max(height(node->left),
              height(node->right));
  int balanceFactor = getBalanceFactor(node);
  if (balanceFactor > 1) {
    if (key < node->left->key) {
      return rightRotate(node);
```

```
    } else if (key > node->left->key) {
      node->left = leftRotate(node->left);
      return rightRotate(node);
    }
  }
  if (balanceFactor < -1) {
    if (key > node->right->key) {
      return leftRotate(node);
    } else if (key < node->right->key) {
      node->right = rightRotate(node->right);
      return leftRotate(node);
    }
  }
  return node;
}

// Node with minimum value
Node *nodeWithMimumValue(Node *node) {
  Node *current = node;
  while (current->left != NULL)
    current = current->left;
  return current;
}

// Delete a node
Node *deleteNode(Node *root, int key) {
  // Find the node and delete it
  if (root == NULL)
    return root;
  if (key < root->key)
```

```
  root->left = deleteNode(root->left, key);
else if (key > root->key)
  root->right = deleteNode(root->right, key);
else {
  if ((root->left == NULL) ||
    (root->right == NULL)) {
    Node *temp = root->left ? root->left : root->right;
    if (temp == NULL) {
      temp = root;
      root = NULL;
    } else
      *root = *temp;
    free(temp);
  } else {
    Node *temp = nodeWithMimumValue(root->right);
    root->key = temp->key;
    root->right = deleteNode(root->right,
                temp->key);
  }
}

if (root == NULL)
  return root;

// Update the balance factor of each node and
// balance the tree
root->height = 1 + max(height(root->left),
            height(root->right));
int balanceFactor = getBalanceFactor(root);
if (balanceFactor > 1) {
```

```
    if (getBalanceFactor(root->left) >= 0) {
      return rightRotate(root);
    } else {
      root->left = leftRotate(root->left);
      return rightRotate(root);
    }
  }
  if (balanceFactor < -1) {
    if (getBalanceFactor(root->right) <= 0) {
      return leftRotate(root);
    } else {
      root->right = rightRotate(root->right);
      return leftRotate(root);
    }
  }
  return root;
}

// Print the tree
void printTree(Node *root, string indent, bool last) {
  if (root != nullptr) {
    cout << indent;
    if (last) {
      cout << "R----";
      indent += "   ";
    } else {
      cout << "L----";
      indent += "|  ";
    }
    cout << root->key << endl;
```

```
    printTree(root->left, indent, false);
    printTree(root->right, indent, true);
  }
}

int main() {
  Node *root = NULL;
  root = insertNode(root, 33);
  root = insertNode(root, 13);
  root = insertNode(root, 53);
  root = insertNode(root, 9);
  root = insertNode(root, 21);
  root = insertNode(root, 61);
  root = insertNode(root, 8);
  root = insertNode(root, 11);
  printTree(root, "", true);
  root = deleteNode(root, 13);
  cout << "After deleting " << endl;
  printTree(root, "", true);
}
```

9.6 Complexities of Different Operations on an AVL Tree

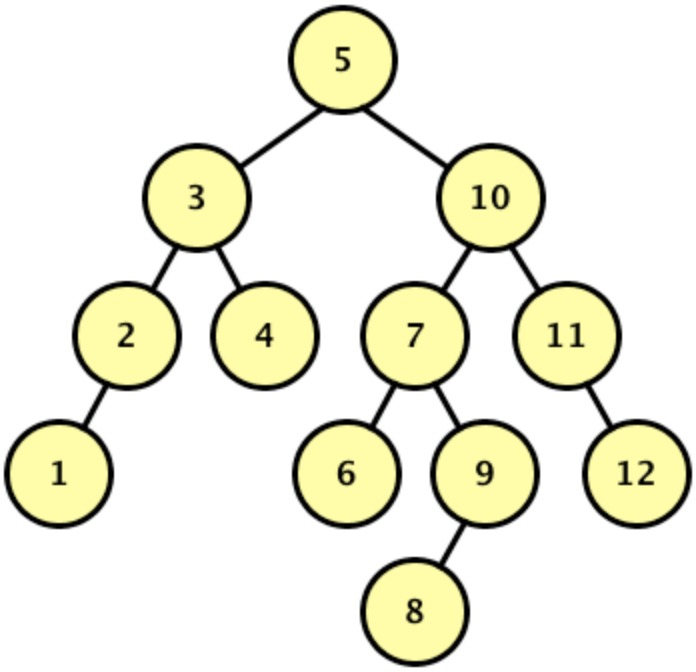
Insertion	Deletion	Search
O(log n)	O(log n)	O(log n)

9.7 AVL Tree Applications

- AVL trees are mostly used for in-memory sorts of sets and dictionaries.
- AVL trees are also used extensively in database applications in which insertions and deletions are fewer but there are frequent lookups for data required.
- It is used in applications that require improved searching apart from the database applications.

Practice Problems:

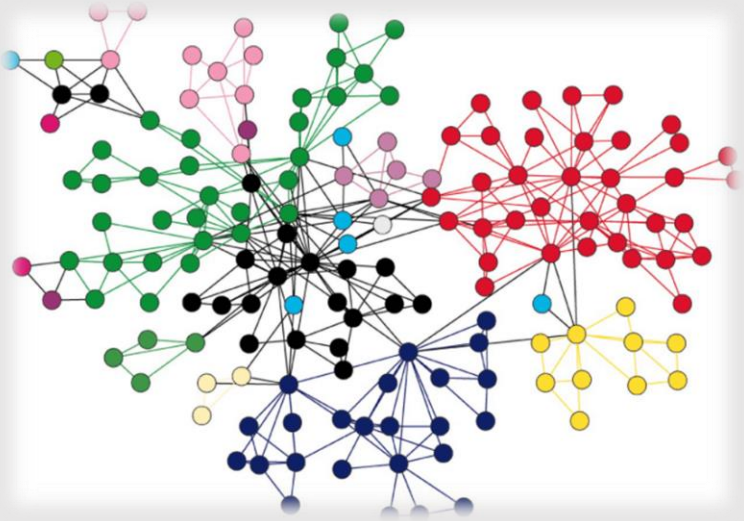
Given the following AVL Tree:



a) Draw and design the resulting **BST** after **5** is removed, but *before* any rebalancing takes place. Label each node in the resulting tree with its **balance factor**. Replace a node with both children using an appropriate value from the node's **left** child.

b) Now rebalance the tree that results from (a). Draw a new tree for each rotation that occurs when rebalancing the AVL Tree (you only need to draw one tree that results from an RL or LR rotation). You do not need to label these trees with balance factors.

Lab#10
Graph Algorithms



Prepared by: Engr. Amna Arooj

FCSE, GIKI

Lab#10
Graph Algorithms

10.1 Graph Data Structure

A graph data structure is a collection of nodes that have data and are connected to other nodes.

Let's try to understand this through an **example**. On facebook, everything is a node. That includes User, Photo, Album, Event, Group, Page, Comment, Story, Video, Link, Note...anything that has data is a node.

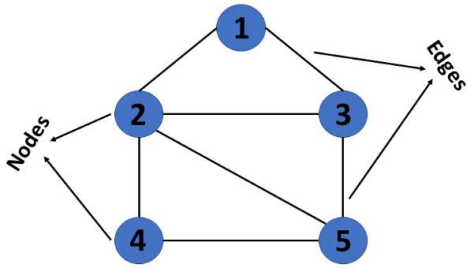
Every relationship is an edge from one node to another. Whether you post a photo, join a group, like a page, etc., a new edge is created for that relationship.

Example of graph data structure

All of facebook is then a collection of these nodes and edges. This is because facebook uses a graph data structure to store its data.

More precisely, a graph is a data structure (V, E) that consists of

- A collection of vertices V
- A collection of edges E, represented as ordered pairs of vertices (u,v)



This graph has a set of vertices V= { 1,2,3,4,5} and a set of edges E= { (1,2),(1,3),(2,3),(2,4),(2,5),(3,5),(4,50 }.

In the graph,

V = {0, 1, 2, 3}

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

G = {V, E}

10.2 Why Graph Algorithms are Important?

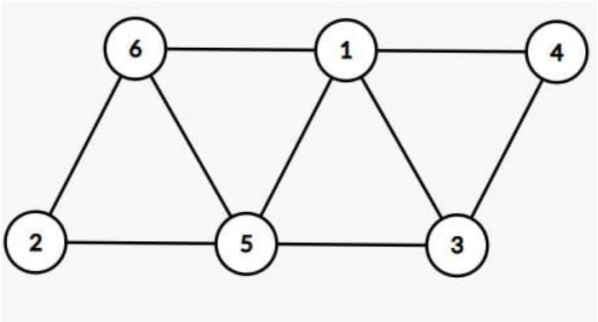
Graphs are very useful data structures which can be to model various problems. These algorithms have direct applications on Social Networking sites, State Machine modeling and many more.

10.3 An application in real life

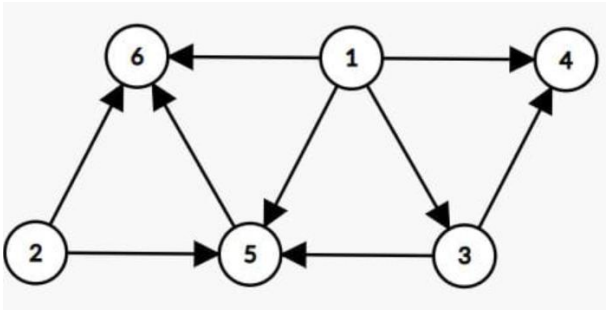
- **Google Maps**: link your journey from the start to the end.
- **Social Networks**: friends are connected with each other using an edge where each user represents a vertex.
- **Recommendation System**: relationship data between user's recommendations uses graphs for connection.

10.4 Type of graphs

- **Undirected**: All edges in an undirected graph are bidirectional and do not point in any particular direction.



- **Directed**: In a directed graph, all edges are unidirectional; they point in a single direction.



10.5 Operations on Graphs in Data Structures

The operations you perform on the graphs in data structures are listed below:

- Creating graphs
- Insert vertex
- Delete vertex
- Insert edge
- Delete edge

You will go over each operation in detail one by one:

10.5.1 Creating Graph OR Graph representation

We can create or represent a graph in several ways.

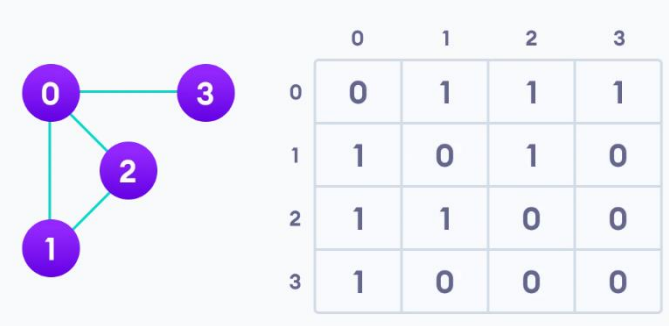
The following are the two most frequent ways of expressing a graph:

10.5.*2* Adjacency Matrix

An adjacency matrix is a 2D array of V x V vertices. Each row and column represent a vertex.

If the value of any element `a[i][j]` is 1, it represents that there is an edge connecting vertex i and vertex j.

The adjacency matrix for the graph we created below is



Graph adjacency matrix

Since it is an undirected graph, for edge (0,2), we also need to mark edge (2,0); making the adjacency matrix symmetric about the diagonal.

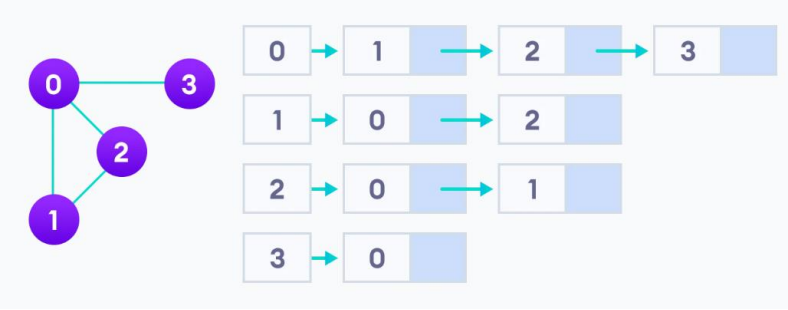
Edge lookup(checking if an edge exists between vertex A and vertex B) is extremely fast in adjacency matrix representation but we have to reserve space for every possible link between all vertices(V x V), so it requires more space.

10.5.*3* Adjacency List

An adjacency list represents a graph as an array of linked lists.

The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

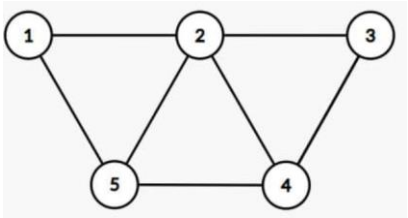
The adjacency list for the graph we made in the first example is as follows:



Adjacency list representation

An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a graph with millions of vertices, this can mean a lot of saved space.

C++ Code for Adjacency list representation of a graph




```
#include <iostream>
#include<list>
using namespace std;

class graph{
public:
   list<int> *adjlist;
   int n;
   graph(int v){
      adjlist=new list<int> [v];
      n=v;
   }

   void addedge(int u,int v,bool bi){
      adjlist[u].push_back(v);
      if(bi){
         adjlist[v].push_back(u);
      }
   }

   void print(){
      for(int i=0;i<n;i++){
         cout<<i<<"-->";
         for(auto it:adjlist[i]){
            cout<<it<<" ";
         }
         cout<<endl;
```

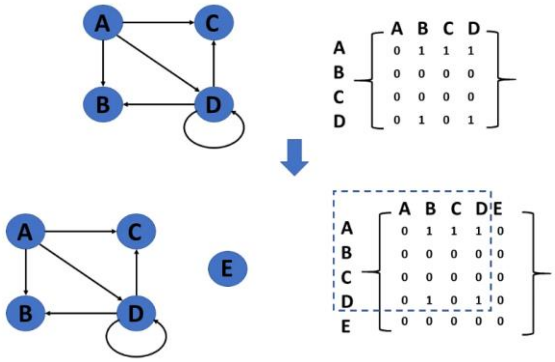
```
        }
        cout<<endl;
    }


};

int main() {
    graph g(5);
    g.addedge(1,2,true);
    g.addedge(4,2,true);
    g.addedge(1,3,true);
    g.addedge(4,3,true);
    g.addedge(1,4,true);

    g.print();
}
```

Output

```
1-->2 3 4
2-->1 4
3-->1 4
4-->2 3 1
```

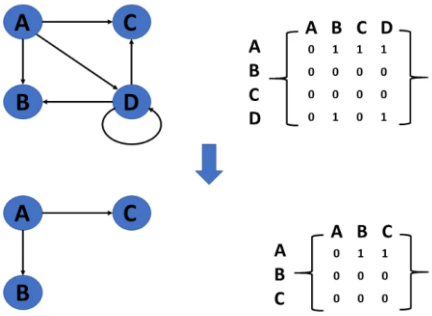
10.5.4 Insert Vertex

When you add a vertex that after introducing one or more vertices or nodes, the graph's size grows by one, increasing the matrix's size by one at the row and column levels.



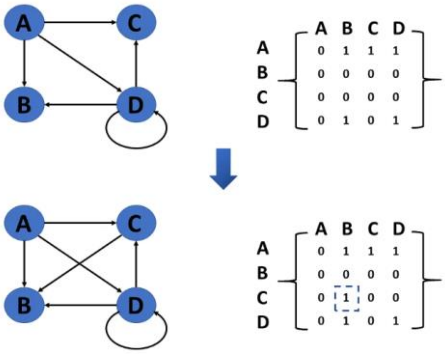
10.5.5 Delete Vertex

- Deleting a vertex refers to removing a specific node or vertex from a graph that has been saved.
- If a removed node appears in the graph, the matrix returns that node. If a deleted node does not appear in the graph, the matrix returns the node not available.



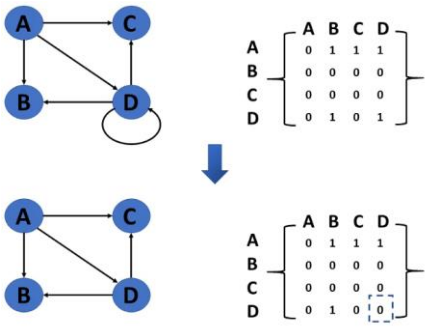
10.5.6 Insert Edge

Connecting two provided vertices can be used to add an edge to a graph.



10.5.7 Delete Edge

The connection between the vertices or nodes can be removed to delete an edge.



10.6 Graph Traversal Algorithm

The process of visiting or updating each vertex in a graph is known as graph traversal. The sequence in which they visit the vertices is used to classify such traversals. Graph traversal is a subset of tree traversal.

There are two techniques to implement a graph traversal algorithm:

- Breadth-first search
- Depth-first search

10.6.1 Breadth-First Search or BFS

BFS is a search technique for finding a node in a graph data structure that meets a set of criteria.

It begins at the root of the graph and investigates all nodes at the current depth level before moving on to nodes at the next depth level.

To maintain track of the child nodes that have been encountered but not yet inspected, more memory, generally you require a queue.

Algorithm of breadth-first search

Step 1: Consider the graph you want to navigate.

Step 2: Select any vertex in your graph, say v1, from which you want to traverse the graph.

Step 3: Examine any two data structures for traversing the graph.

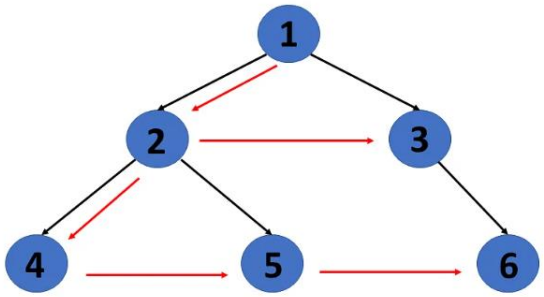
Visited array (size of the graph)

Queue data structure

Step 4: Starting from the vertex, you will add to the visited array, and afterward, you will v1's adjacent vertices to the queue data structure.

Step 5: Now, using the FIFO concept, you must remove the element from the queue, put it into the visited array, and then return to the queue to add the adjacent vertices of the removed element.

Step 6: Repeat step 5 until the queue is not empty and no vertex is left to be visited.



BFS	1	2	3	4	5	6

Space Complexity: O(n)

Worse Case Time Complexity: O(n)

Breadth First Search is complete on a finite set of nodes and optimal if the cost of moving from one node to another is constant.

10.6.*2* C++ code for BFS implementation

```
// Program to print BFS traversal from a given
// source vertex. BFS(int s) traverses vertices
// reachable from s.
#include<iostream>
#include <list>
```

```
using namespace std;

// This class represents a directed graph using
// adjacency list representation
class Graph
{
    int V;    // No. of vertices

    // Pointer to an array containing adjacency
    // lists
    list<int> *adj;
public:
    Graph(int V);  // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // prints BFS traversal from a given source s
    void BFS(int s);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent
    // vertices of a vertex
    list<int>::iterator i;
```

```
    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued
        // vertex s. If a adjacent has not been visited,
        // then mark it visited and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}

// Driver program to test methods of graph class
int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Breadth First Traversal "
         << "(starting from vertex 2) \n";
    g.BFS(2);

    return 0;
}
```

10.6.*3* Depth-First Search or DFS

DFS is a search technique for finding a node in a graph data structure that meets a set of criteria.

The depth-first search (DFS) algorithm traverses or explores data structures such as trees and graphs. The DFS algorithm begins at the root node and examines each branch as far as feasible before backtracking.

To maintain track of the child nodes that have been encountered but not yet inspected, more memory, generally a stack, is required.

Algorithm of depth-first search

Step 1: Consider the graph you want to navigate.

Step 2: Select any vertex in our graph, say v1, from which you want to begin traversing the graph.

Step 3: Examine any two data structures for traversing the graph.

Visited array (size of the graph)

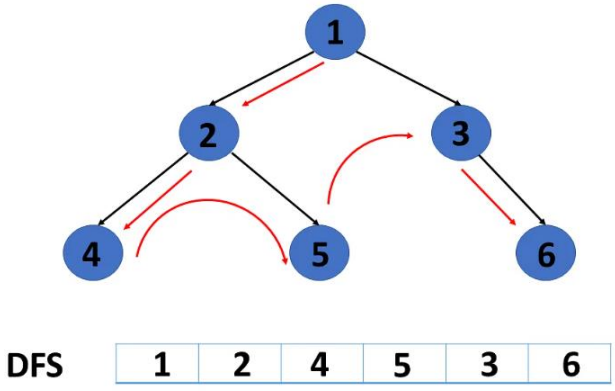
Stack data structure

Step 4: Insert v1 into the array's first block and push all the adjacent nodes or vertices of vertex v1 into the stack.

Step 5: Now, using the FIFO principle, pop the topmost element and put it into the visited array, pushing all of the popped element's nearby nodes into it.

Step 6: If the topmost element of the stack is already present in the array, discard it instead of inserting it into the visited array.

Step 7: Repeat step 6 until the stack data structure isn't empty.



10.6.*4* code for DFS implementation

Visualisation

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
using namespace std;

class Graph{
   int v;    // number of vertices
```

```
   // pointer to a vector containing adjacency lists
   vector < int > *adj;
public:
   Graph(int v);  // Constructor

   // function to add an edge to graph
   void add_edge(int v, int w);

   // prints dfs traversal from a given source `s`
   void dfs();
   void dfs_util(int s, vector < bool> &visited);
};

Graph::Graph(int v){
   this -> v = v;
   adj = new vector < int >[v];
}

void Graph::add_edge(int u, int v){
   adj[u].push_back(v); // add v to u's list
   adj[v].push_back(v);  // add u to v's list (remove this statement if the
graph is directed!)
}
void Graph::dfs(){
   // visited vector - to keep track of nodes visited during DFS
   vector < bool > visited(v, false);  // marking all nodes/vertices as not
visited
   for(int i = 0; i < v; i++)
       if(!visited[i])
           dfs_util(i, visited);
}
// notice the usage of call-by-reference here!
void Graph::dfs_util(int s, vector < bool > &visited){
   // mark the current node/vertex as visited
   visited[s] = true;
    // output it to the standard output (screen)
   cout << s << " ";
```

```
   // traverse its adjacency list and recursively call dfs_util for all of
its neighbours!
   // (only if the neighbour has not been visited yet!)
   for(vector < int > :: iterator itr = adj[s].begin(); itr !=
adj[s].end(); itr++)
       if(!visited[*itr])
           dfs_util(*itr, visited);
}

int main()
{
   // create a graph using the Graph class we defined above
   Graph g(4);
   g.add_edge(0, 1);
   g.add_edge(0, 2);
   g.add_edge(1, 2);
   g.add_edge(2, 0);
   g.add_edge(2, 3);
   g.add_edge(3, 3);

   cout << "Following is the Depth First Traversal of the provided graph"
        << "(starting from vertex 0): ";
   g.dfs();
   // output would be: 0 1 2 3
   return 0;
}
```

Space Complexity: O(n)

Worse Case Time Complexity: O(n) Depth First Search is complete on a finite set of nodes. I works better on shallow trees.

Implementation of DFS in C++

```
#include<iostream>
#include<vector>
#include<queue>

using namespace std;
```

```
struct Graph{
        int v;
        bool **adj;
        public:
                Graph(int vcount);
                void addEdge(int u,int v);
                void deleteEdge(int u,int v);
                vector<int> DFS(int s);
                void DFSUtil(int s,vector<int> &dfs,vector<bool> &visited);
};
Graph::Graph(int vcount){
        this->v = vcount;
        this->adj=new bool*[vcount];
        for(int i=0;i<vcount;i++)
                this->adj[i]=new bool[vcount];
        for(int i=0;i<vcount;i++)
                for(int j=0;j<vcount;j++)
                        adj[i][j]=false;
}

void Graph::addEdge(int u,int w){
        this->adj[u][w]=true;
        this->adj[w][u]=true;
}

void Graph::deleteEdge(int u,int w){
        this->adj[u][w]=false;
        this->adj[w][u]=false;
}

void Graph::DFSUtil(int s, vector<int> &dfs, vector<bool> &visited){
        visited[s]=true;
        dfs.push_back(s);
        for(int i=0;i<this->v;i++){
                if(this->adj[s][i]==true && visited[i]==false)
                        DFSUtil(i,dfs,visited);
        }
}
```

```
vector<int> Graph::DFS(int s){
        vector<bool> visited(this->v);
        vector<int> dfs;
        DFSUtil(s,dfs,visited);
        return dfs;
}
```

10.7 Difference between BFS and DFS

The following are the important differences between BFS and DFS –

Key	BFS	DFS
Definition	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
Data structure	BFS uses a Queue to find the shortest path.	DFS uses a Stack to find the shortest path.
Source	BFS is better when target is closer to Source.	DFS is better when target is far from source.
Suitability for decision tree	As BFS considers all neighbor so it is not suitable for decision tree used in puzzle games.	DFS is more suitable for decision tree. As with one decision, we need to traverse further to augment the decision. If we reach the conclusion, we won.
Speed	BFS is slower than DFS.	DFS is faster than BFS.
Time Complexity	Time Complexity of BFS = O(V+E) where V is vertices and E is edges.	Time Complexity of DFS is also O(V+E) where V is vertices and E is edges.

Memory	BFS requires more memory space.	DFS requires less memory space.
Tapping in loops	In BFS, there is no problem of trapping into finite loops.	In DFS, we may be trapped into infinite loops.
Principle	BFS is implemented using FIFO (First In First Out) principle.	DFS is implemented using LIFO (Last In First Out) principle.

Conclusion

Both BFS and DFS are graph traversal algorithms. The most significant difference between the two is that the BFS algorithm uses a Queue to find the shortest path, while the DFS algorithm uses a Stack to find the shortest path.

Practice Problems:

Implement the Breadth-First Search algorithm in C++ for a given graph represented as an adjacency list. Assume the graph has n nodes numbered from 0 to n-1. Write a function **bfs** that takes the adjacency list and the starting node as parameters and prints the order in which the nodes are visited during BFS.

For example, given the following adjacency list for a graph with 5 nodes:

0: 1 2

1: 0 2 3

2: 0 1 4

3: 1 4

4: 2 3

If the starting node is 0, the function **bfs** should print: 0, 1, 2, 3, 4.

Your task is to complete the implementation of the **bfs** function in C++.

Lab#11
Hash Table and Hash Function in Data Structures

Prepared by: Engr. Amna Arooj

FCSE, GIKI

Lab#11
Hash Table and Hash Function in Data Structures

11.1 Definition of C++ Hash Table

A Hash table is basically a data structure that is used to store the key value pair. In C++, a hash table uses the hash function to compute the index in an array at which the value needs to be stored or searched. This process of computing the index is called hashing. Values in a hash table are not stored in the sorted order and there are huge chances of collisions in the hash table which is generally solved by the chaining process (creation of a linked list having all the values and the keys associated with it).

11.2 Need for Hash data structure

Every day, the data on the internet is increasing multifold and it is always a struggle to store this data efficiently. In day-to-day programming, this amount of data might not be that big, but still, it needs to be stored, accessed, and processed easily and efficiently. A very common data structure that is used for such a purpose is the Array data structure.

Now the question arises if Array was already there, what was the need for a new data structure! The answer to this is in the word "**efficiency**". Though storing in Array takes O(1) time, searching in it takes at least O(log n) time. This time appears to be small, but for a large data set, it can cause a lot of problems and this, in turn, makes the Array data structure inefficient.

So now we are looking for a data structure that can store the data and search in it in constant time, i.e. in O(1) time. This is how Hashing data structure came into play. With the introduction of the Hash data structure, it is now possible to easily store data in constant time and retrieve them in constant time as well.

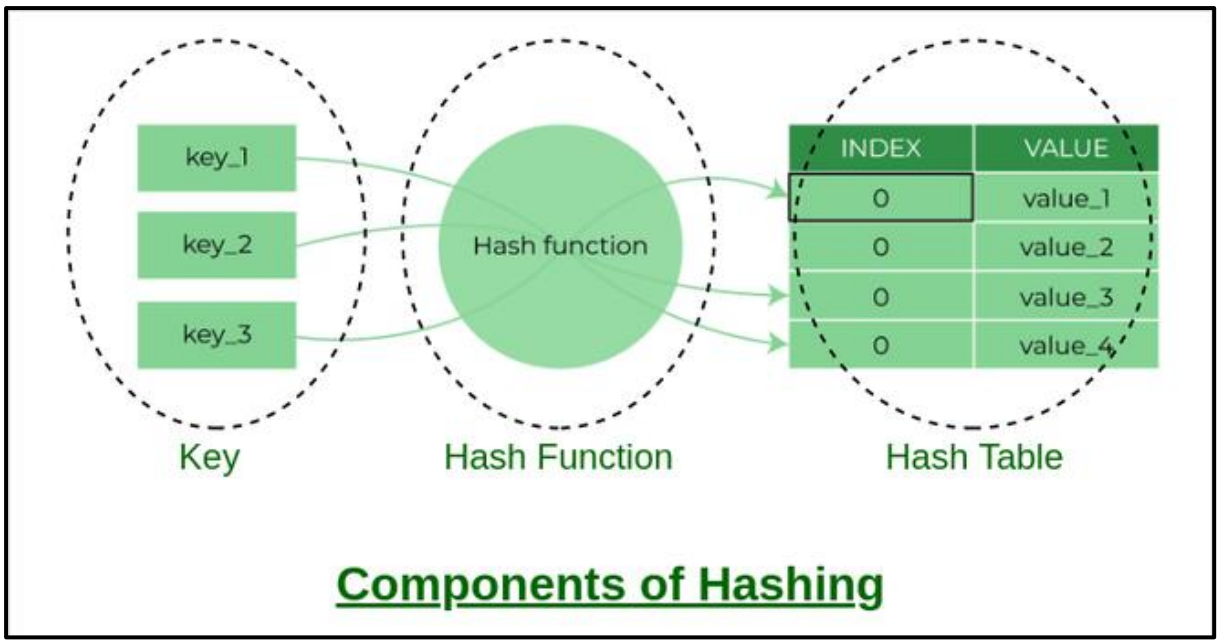
11.3 Components of Hashing

There are majorly three components of hashing:

a) **Key:** A **Key** can be anything string or integer which is fed as input in the hash function the technique that determines an index or location for storage of an item in a data structure.

b) Hash Function: The **hash function** receives the input key and returns the index of an element in an array called a hash table. The index is known as the **hash index**.

c) Hash Table: Hash table is a data structure that maps keys to values using a special function called a hash function. Hash stores the data in an associative manner in an array where each data value has its own unique index.



11.4 How does Hashing work?

Suppose we have a set of strings {“ab”, “cd”, “efg”} and we would like to store it in a table.

Our main objective here is to search or update the values stored in the table quickly in O(1) time and we are not concerned about the ordering of strings in the table. So the given set of strings can act as a key and the string itself will act as the value of the string but how to store the value corresponding to the key?

Step 1: We know that hash functions (which is some mathematical formula) are used to calculate the hash value which acts as the index of the data structure where the value will be stored.

Step 2: So, let’s assign

“a” = 1,

“b”=2, .. etc, to all alphabetical characters.

Step 3: Therefore, the numerical value by summation of all characters of the string:

“ab” = 1 + 2 = 3,

“cd” = 3 + 4 = 7 ,

“efg” = 5 + 6 + 7 = 18

Step 4: Now, assume that we have a table of size 7 to store these strings. The hash function that is used here is the sum of the characters in **key mod Table size**. We can compute the location of the string in the array by taking the **sum(string) mod 7**.

Step 5: So we will then store

“ab” in 3 mod 7 = 3,

“cd” in 7 mod 7 = 0, and

“efg” in 18 mod 7 = 4.

0	1	2	3	4	5	6
cd			ab	efg		

Mapping key with indices of array

The above technique enables us to calculate the location of a given string by using a simple hash function and rapidly find the value that is stored in that location. Therefore the idea of hashing seems like a great way to store (key, value) pairs of the data in a table.

Another Example of Hash Table Working

As discussed above, hash tables store the pointers to the actual data/ values. It uses the key to find the index at which the data/ value needs to be stored. Let us understand this with the help of the diagram given below:

Consider the hash table size be 10

Key	Index (using a hash function)	Data
12	12%10 = 2	23
10	10%10 = 0	34
6	6% 10 = 6	54
23	23 % 10 = 3	76
54	54 %10 = 4	75
81	81 %10 = 1	87

The element position in the hash table will be:

0	1	2	3	4	5	6	7	8	9
34	87	23	76	75		54			

As we can see above, there are high chances of collision as there could be 2 or more keys that compute the same hash code resulting in the same index of elements in the hash table. A collision cannot be avoided in case of hashing even if we have a large table size. We can prevent a collision by choosing the good hash function and the implementation method.

Though there are a lot of implementation techniques used for it like Linear probing, open hashing, etc. We will understand and implement the basic Open hashing technique also called separate chaining. In this technique, a linked list is used for the chaining of values. Every entry in the hash table is a linked list. So, when the new entry needs to be done, the index is computed using the key and table size. Once computed, it is inserted in the list corresponding to that index. When there are 2 or more values having the same hash value/ index, both entries are inserted corresponding to that index linked with each other. For example,

```
2 —> 12 ——>22 —->32
```

Where 2 is the index of the hash table retrieved using the hash function

12, 22, 32 are the data values that will be inserted linked with each other

11.5 C++ Implementation of Hash Table

Let us implement the hash table using the above described Open hashing or Separate technique:

Code:

```
#include <iostream>
#include <list>
using namespace std;
class HashMapTable
{
// size of the hash table
```

```
inttable_size;
// Pointer to an array containing the keys
list<int> *table;
public:
// creating constructor of the above class containing all
the methods
HashMapTable(int key);
// hash function to compute the index using table_size and
key
inthashFunction(int key) {
return (key % table_size);
}
// inserting the key in the hash table
void insertElement(int key);
// deleting the key in the hash table
void deleteElement(int key);
// displaying the full hash table
void displayHashTable();
};
//creating the hash table with the given table size
HashMapTable::HashMapTable(intts)
{
this->table_size = ts;
table = new list<int>[table_size];
}
// insert function to push the keys in hash table
void HashMapTable::insertElement(int key)
{
```

```
int index = hashFunction(key);
table[index].push_back(key);
}
// delete function to delete the element from the hash table
void HashMapTable::deleteElement(int key)
{
int index = hashFunction(key);
// finding the key at the computed index
list <int> :: iterator i;
for (i = table[index].begin(); i != table[index].end(); i++)
{
if (*i == key)
break;
}
// removing the key from hash table if found
if (i != table[index].end())
table[index].erase(i);
}
// display function to showcase the whole hash table
void HashMapTable::displayHashTable() {
for (inti = 0; i<table_size; i++) {
cout<<i;
// traversing at the recent/ current index
for (auto j : table[i])
cout<< " ==> " << j;
cout<<endl;
}
}
```

```
// Main function
intmain()
{
// array of all the keys to be inserted in hash table
intarr[] = {20, 34, 56, 54, 76, 87};
int n = sizeof(arr)/sizeof(arr[0]);
// table_size of hash table as 6
HashMapTableht(6);
for (inti = 0; i< n; i++)
ht.insertElement(arr[i]);
// deleting element 34 from the hash table
ht.deleteElement(34);
// displaying the final data of hash table
ht.displayHashTable();
return 0;
}
```

Output:

```
0 ==> 54
1
2 ==> 20 ==> 56
3 ==> 87
4 ==> 76
5
```

Explanation:

In the above code, an array is created for all the keys that need to be inserted in the has table. Class and constructors are created for hashMapTable to calculate the hash function using the

formula mentioned above. The list is created as the pointer to the array of key values. Specific functions are created for the insertion, deletion, and display of the hash table and called from the main method. While insertion, if 2 or more elements have the same index, they are inserted using the list one after the other.

11.6 What is a Hash function?

The hash function creates a mapping between key and value, this is done through the use of mathematical formulas known as hash functions. The result of the hash function is referred to as a hash value or hash. The hash value is a representation of the original string of characters but usually smaller than the original.

For example: Consider an array as a Map where the key is the index and the value is the value at that index. So for an array A if we have index i which will be treated as the key then we can find the value by simply looking at the value at A[i].
simply looking up A[i].

11.6.1 Types of Hash functions:

There are many hash functions that use numeric or alphanumeric keys. This article focuses on discussing different hash functions:

- Division Method.
- Mid Square Method.
- Folding Method.
- Multiplication Method

11.6.2 Properties of a Good hash function

A hash function that maps every item into its own unique slot is known as a perfect hash function. We can construct a perfect hash function if we know the items and the collection will never change but the problem is that there is no systematic way to construct a perfect hash function given an arbitrary collection of items. Fortunately, we will still gain performance efficiency even if the hash function isn't perfect. We can achieve a perfect hash function by increasing the size of the hash table so that every possible value can be accommodated. As a result, each item will have a unique slot. Although this approach is feasible for a small number of items, it is not practical when the number of possibilities is large.

So, We can construct our hash function to do the same but the things that we must be careful about while constructing our own hash function.

A good hash function should have the following properties:

- Efficiently computable.
- Should uniformly distribute the keys (Each table position is equally likely for each.
- Should minimize collisions.

- Should have a low load factor(number of items in the table divided by the size of the table).

Complexity of calculating hash value using the hash function

- Time complexity: O(n)
- Space complexity: O(1)

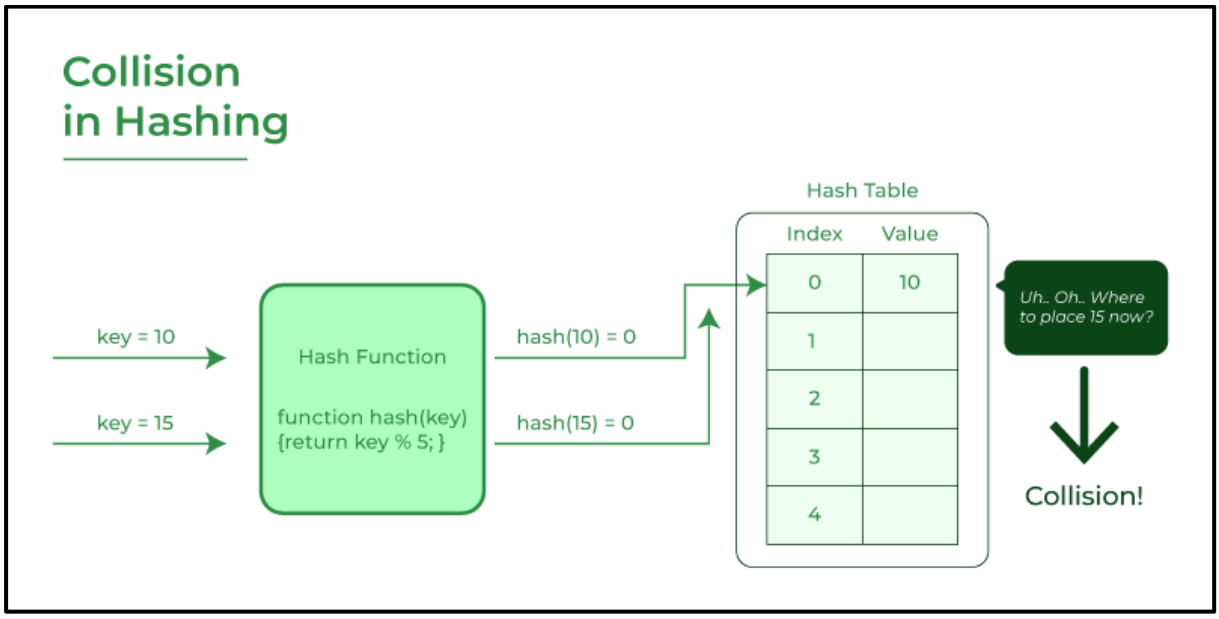
11.6.3 Problem with Hashing

If we consider the above example, the hash function we used is the sum of the letters, but if we examined the hash function closely then the problem can be easily visualized that for different strings same hash value is begin generated by the hash function.

For example: {“ab”, “ba”} both have the same hash value, and string {“cd”,”be”} also generate the same hash value, etc. This is known as **collision** and it creates problem in searching, insertion, deletion, and updating of value.

11.6.4 What is collision?

The hashing process generates a small number for a big key, so there is a possibility that two keys could produce the same value. The situation where the newly inserted key maps to an already occupied, and it must be handled using some collision handling technology.

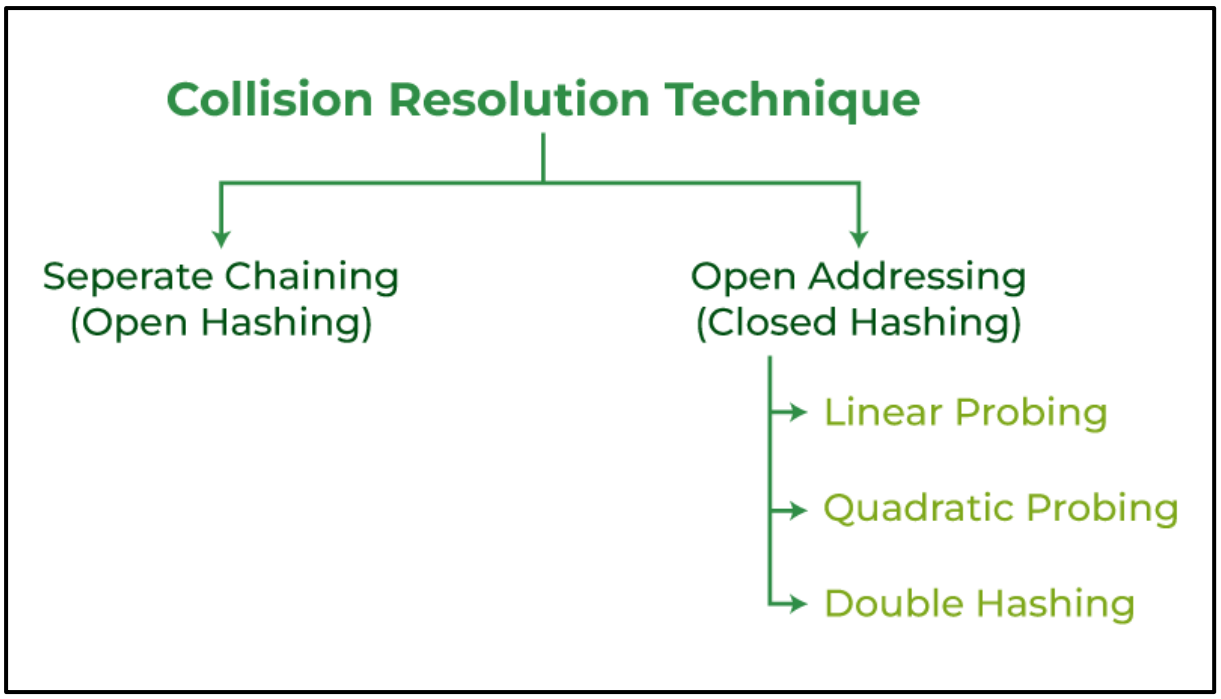


What is Collision in Hashing

11.6.5 How to handle Collisions?

There are mainly two methods to handle collision:

1. Separate Chaining:
2. Open Addressing:



Collision resolution technique

1) Separate Chaining

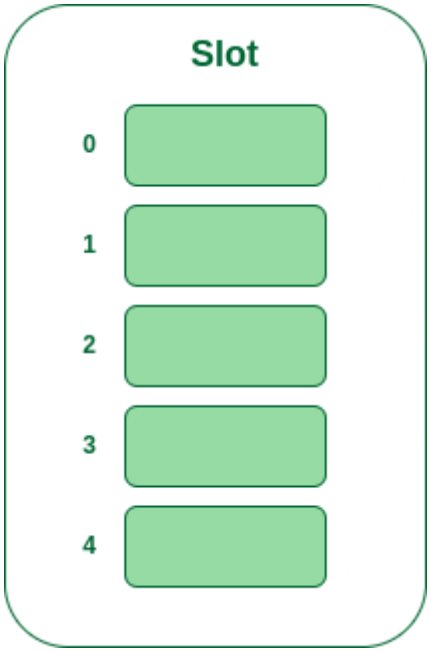
The idea is to make each cell of the hash table point to a linked list of records that have the same hash function value. Chaining is simple but requires additional memory outside the table.

Example: We have given a hash function and we have to insert some elements in the hash table using a separate chaining method for collision resolution technique.

```
Hash function = key % 5,

Elements = 12, 15, 22, 25 and 37.
```

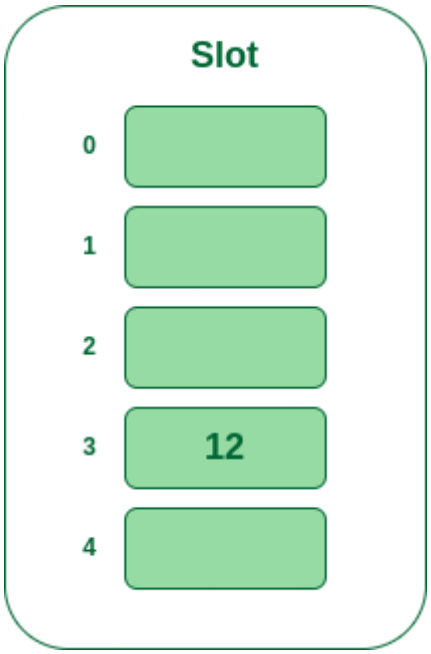
Let's see step by step approach to how to solve the above problem:

Step 1: First draw the empty hash table which will have a possible range of hash values from 0 to 4 according to the hash function provided.



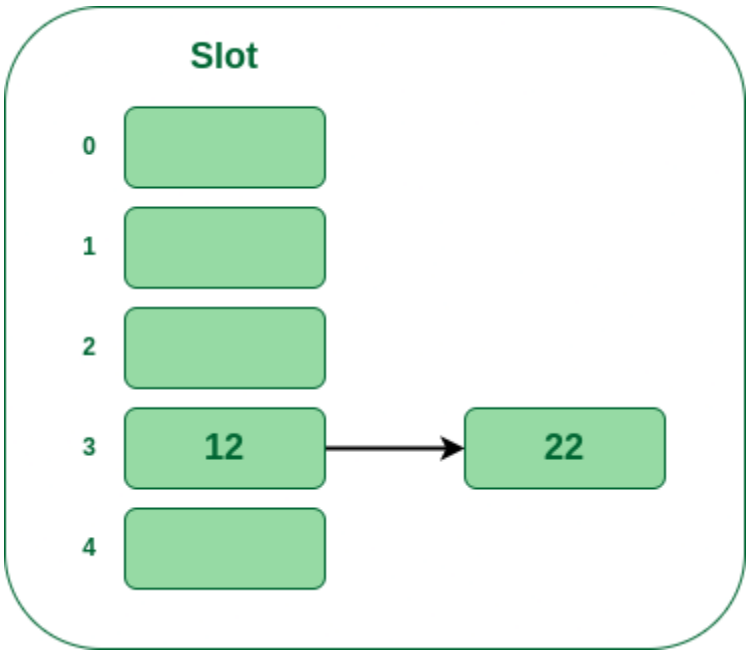
Hash table

Step 2: Now insert all the keys in the hash table one by one. The first key to be inserted is 12 which is mapped to bucket number 2 which is calculated by using the hash function 12%5=2.



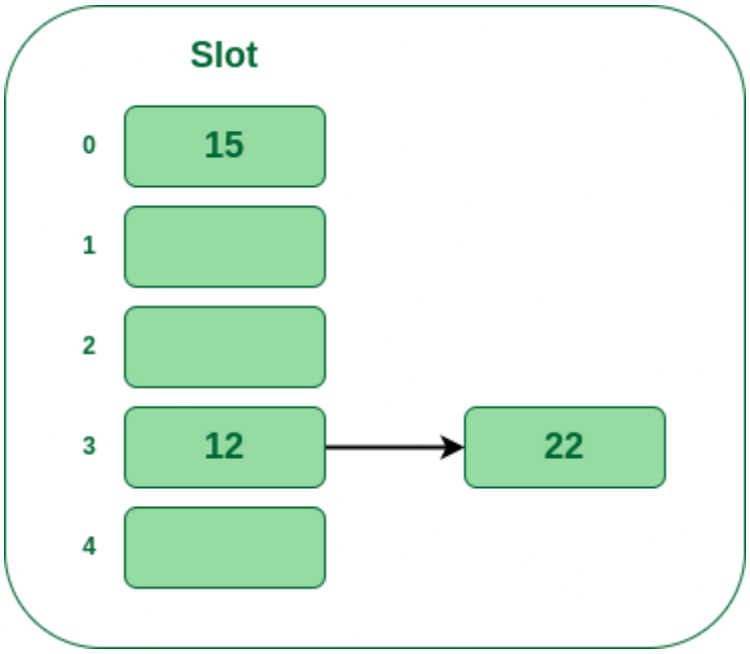
Insert 12 into hash table

Step 3: Now the next key is 22. It will map to bucket number 2 because 22%5=2. But bucket 2 is already occupied by key 12.



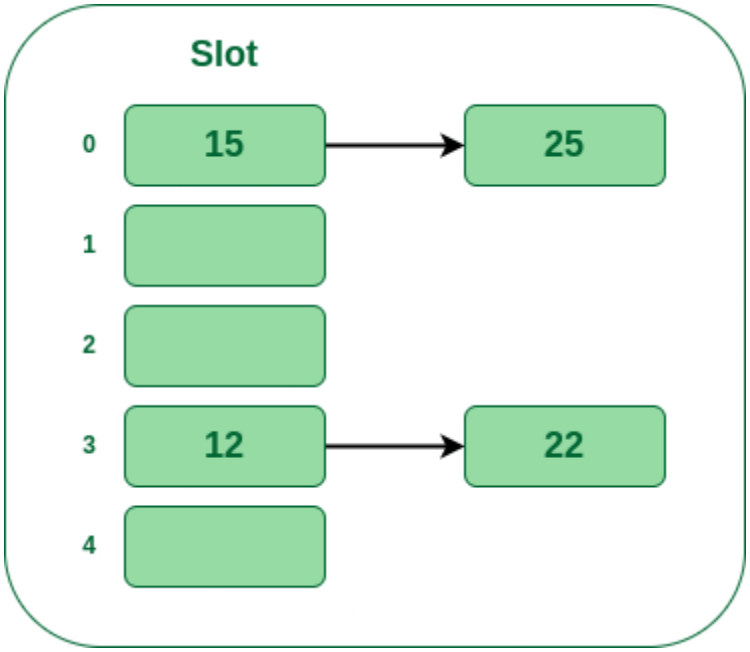
Insert 22 into hash table

Step 4: The next key is 15. It will map to slot number 0 because 15%5=0.



Insert 15 into hash table

Step 5: Now the next key is 25. Its bucket number will be 25%5=0. But bucket 0 is already occupied by key 25. So separate chaining method will again handle the collision by creating a linked list to bucket 0.



Insert 25 into hash table

Hence In this way, the separate chaining method is used as the collision resolution technique.

2) Open Addressing

In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we examine the table slots one by one until the desired element is found or it is clear that the element is not in the table.

2.a) Linear Probing

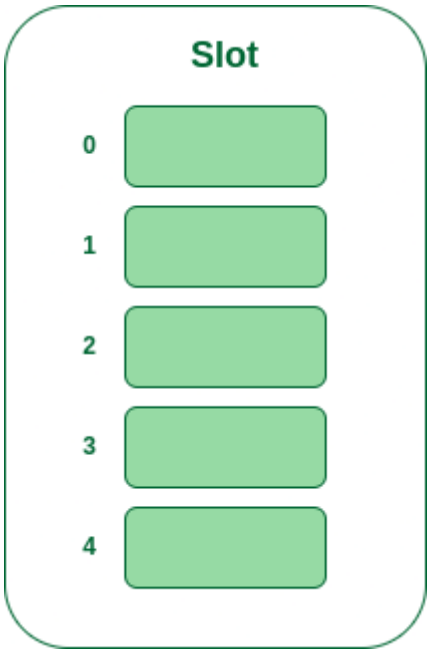
In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

Algorithm:

- *Calculate the hash key. i.e.* **key = data % size**
- *Check, if* **hashTable[key]** *is empty*
- *store the value directly by* **hashTable[key] = data**
- *If the hash index already has some value then*
- *check for next index using* **key = (key+1) % size**
- *Check, if the next index is available hashTable[key] then store the value. Otherwise try for next index.*
- *Do the above process till we find the space.*

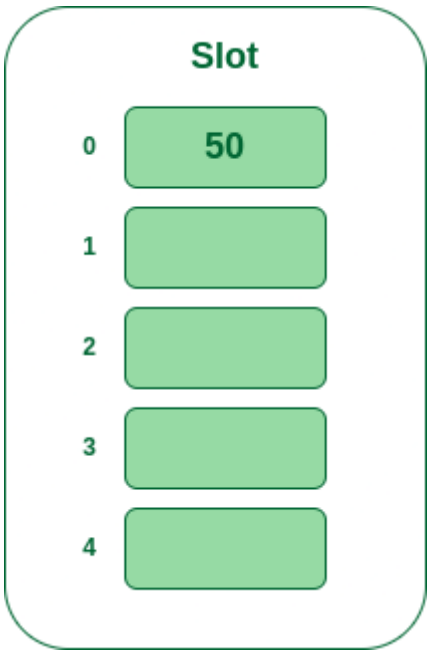
Example: Let us consider a simple hash function as “key mod 5” and a sequence of keys that are to be inserted are 50, 70, 76, 85, 93.

Step1: First draw the empty hash table which will have a possible range of hash values from 0 to 4 according to the hash function provided.



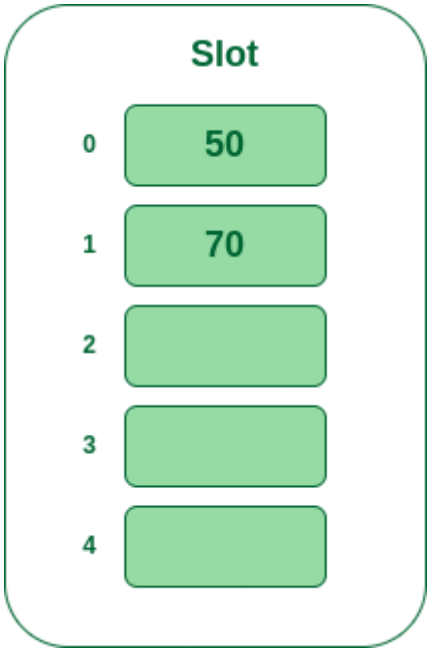
Hash table

Step 2: Now insert all the keys in the hash table one by one. The first key is 50. It will map to slot number 0 because 50%5=0. So insert it into slot number 0.



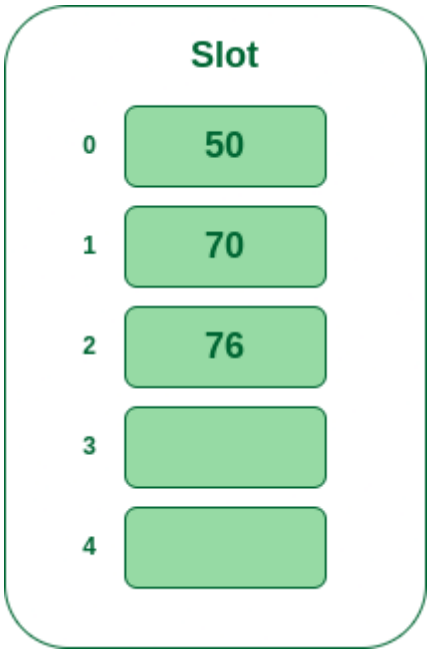
Insert 50 into hash table

Step 3: The next key is 70. It will map to slot number 0 because 70%5=0 but 50 is already at slot number 0 so, search for the next empty slot and insert it.



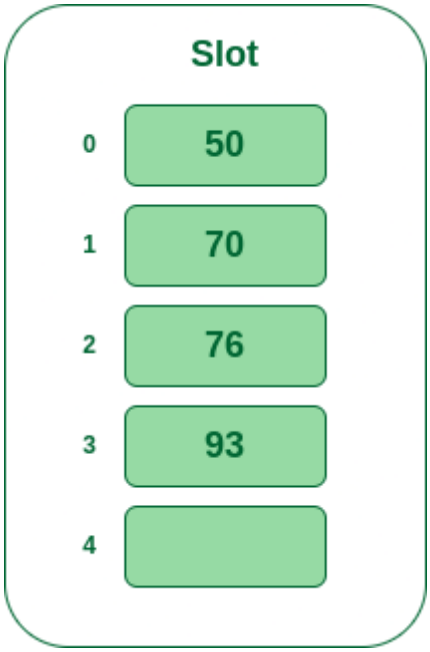
Insert 70 into hash table

Step 4: The next key is 76. It will map to slot number 1 because 76%5=1 but 70 is already at slot number 1 so, search for the next empty slot and insert it.



Insert 76 into hash table

Step 5: The next key is 93 It will map to slot number 3 because 93%5=3, So insert it into slot number 3.



Insert 93 into hash table

2.b) Quadratic Probing

Quadratic probing is an open addressing scheme in computer programming for resolving hash collisions in hash tables. Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.

An example sequence using quadratic probing is:

$H + 1^2, H + 2^2, H + 3^2, H + 4^2 \ldots\ldots\ldots\ldots\ldots\ldots H + k^2$

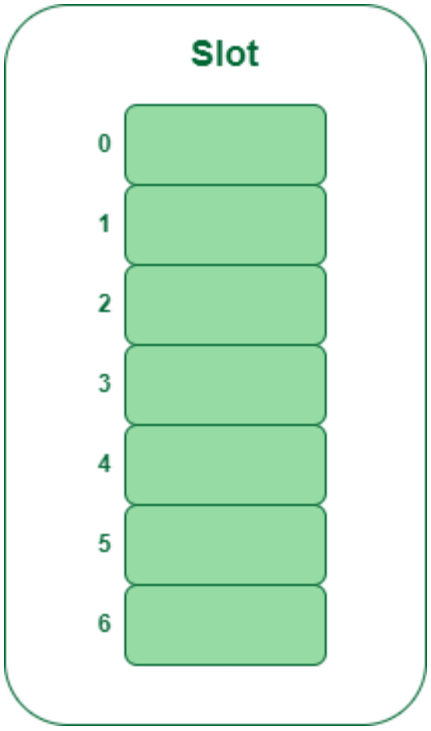
This method is also known as the mid-square method because in this method we look for i^2'th probe (slot) in i'th iteration and the value of i = 0, 1, . . . n – 1. We always start from the original hash location. If only the location is occupied then we check the other slots.

Let hash(x) be the slot index computed using the hash function and n be the size of the hash table.

- If the slot hash(x) % n is full, then we try (hash(x) + 1^2) % n.
- If (hash(x) + 1^2) % n is also full, then we try (hash(x) + 2^2) % n.
- If (hash(x) + 2^2) % n is also full, then we try (hash(x) + 3^2) % n.
- This process will be repeated for all the values of ***i*** until an empty slot is found

Example: Let us consider table Size = 7, hash function as Hash(x) = x % 7 and collision resolution strategy to be f(i) = i^2. Insert = 22, 30, and 50

Step 1: Create a table of size 7.

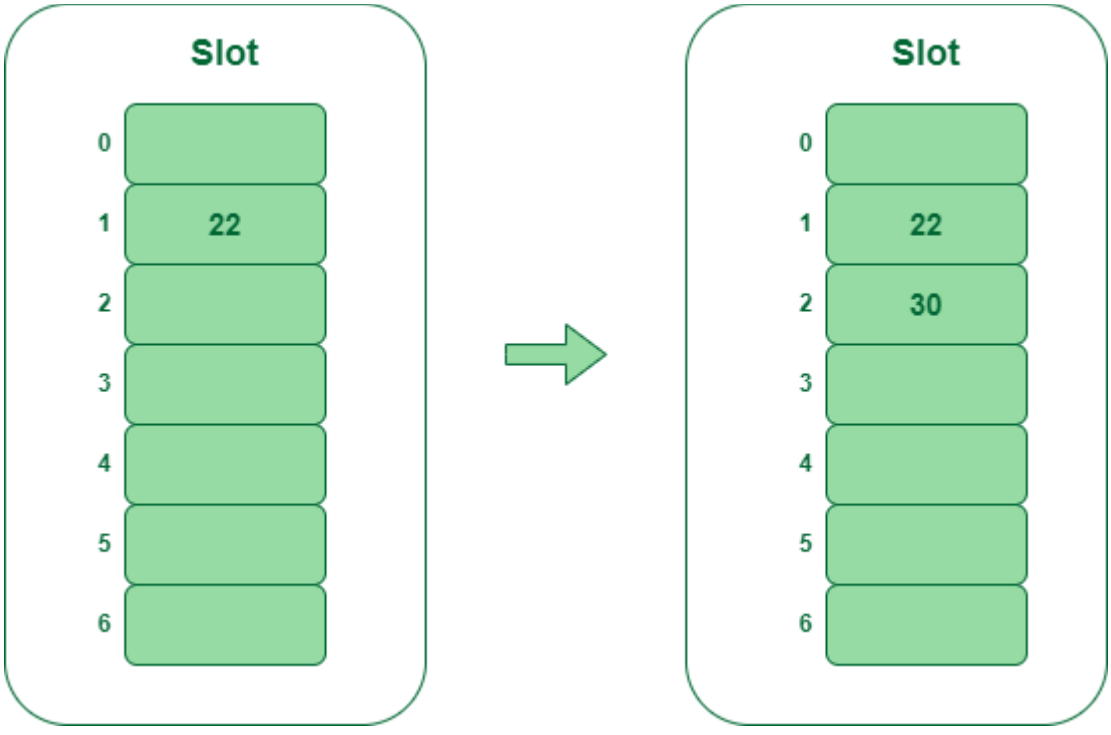


Hash table

Step 2 – Insert 22 and 30

Hash(22) = 22 % 7 = 1, Since the cell at index 1 is empty, we can easily insert 22 at slot 1.

Hash(30) = 30 % 7 = 2, Since the cell at index 2 is empty, we can easily insert 30 at slot 2.



Insert key 22 and 30 in the hash table

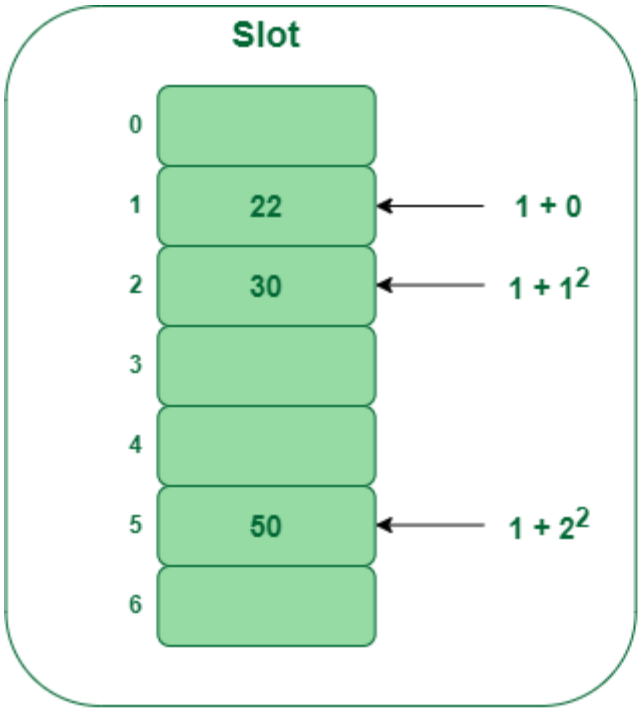
Step 3: Inserting 50

Hash(25) = 50 % 7 = 1

In our hash table slot 1 is already occupied. So, we will search for slot $1+1^2$, i.e. 1+1 = 2,

Again slot 2 is found occupied, so we will search for cell $1+2^2$, i.e.1+4 = 5,

Now, cell 5 is not occupied so we will place 50 in slot 5.



Insert key 50 in the hash table

2.c) Double Hashing

Double hashing is a collision resolving technique in Open Addressed Hash tables. Double hashing make use of two hash function,

The first hash function is **h1(k)** which takes the key and gives out a location on the hash table. But if the new location is not occupied or empty then we can easily place our key.

But in case the location is occupied (collision) we will use secondary hash-function **h2(k)** in combination with the first hash-function **h1(k)** to find the new location on the hash table.

This combination of hash functions is of the form

```
h(k, i) = (h1(k) + i * h2(k)) % n
```

where

i is a non-negative integer that indicates a collision number,

k = element/key which is being hashed

n = hash table size.

Complexity of the Double hashing algorithm:

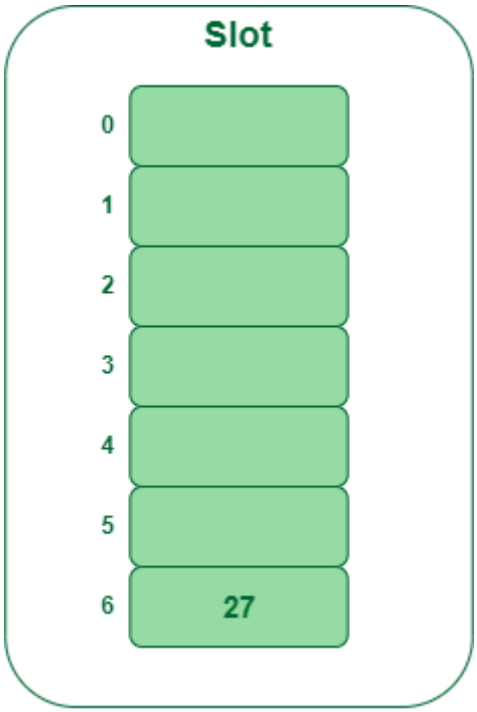
`Time complexity: O(n)`

Example:

Insert the keys 27, 43, 692, 72 into the Hash Table of size 7. where first hash-function is **h1 (k) = k mod 7** and second hash-function is **h2(k) = 1 + (k mod 5)**

Step 1: Insert 27

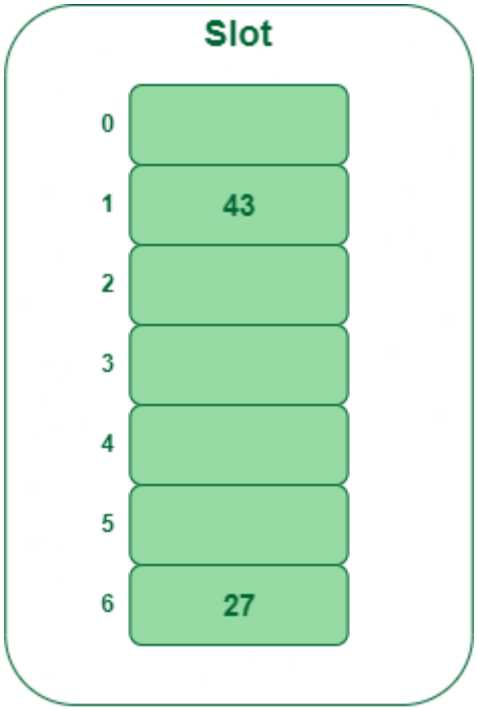
27 % 7 = 6, location 6 is empty so insert 27 into 6 slot.



Insert key 27 in the hash table

Step 2: Insert 43

43 % 7 = 1, location 1 is empty so insert 43 into 1 slot.



Insert key 43 in the hash table

Step 3: Insert 692

692 % 7 = 6, but location 6 is already being occupied and this is a collision

So we need to resolve this collision using double hashing.

h_{new} = [h1(692) + i * (h2(692)] % 7

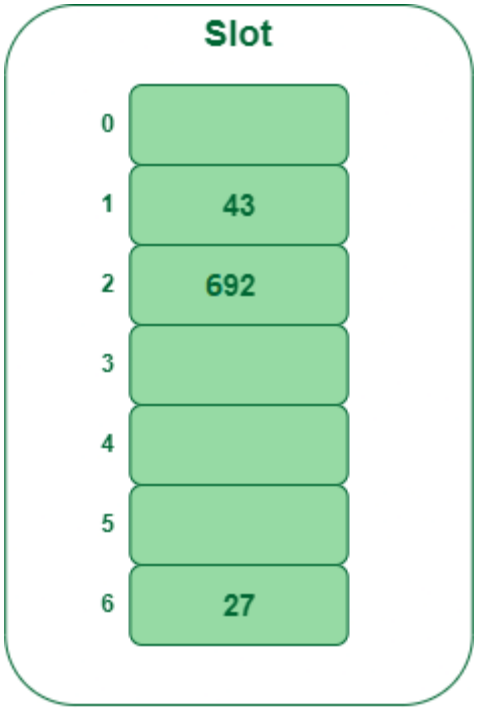
= [6 + 1 * (1 + 692 % 5)] % 7

= 9 % 7

= 2

Now, as 2 is an empty slot,

so we can insert 692 into 2nd slot.



Insert key 692 in the hash table

Step 4: Insert 72

72 % 7 = 2, but location 2 is already being occupied and this is a collision.

So we need to resolve this collision using double hashing.

h_{new} = [h1(72) + i * (h2(72)] % 7

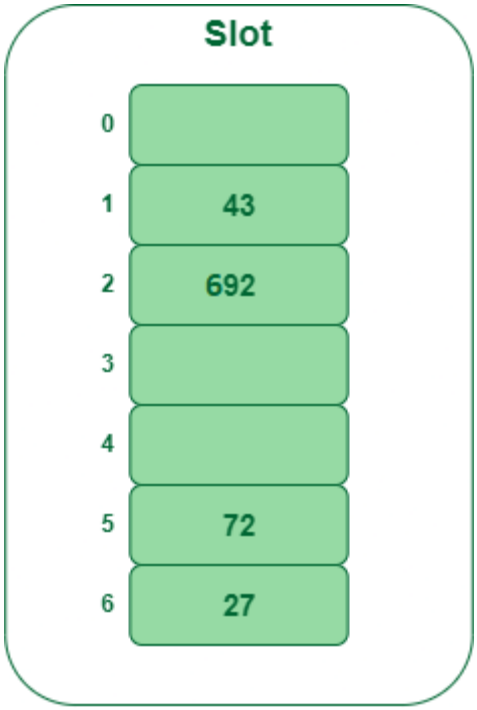
= [2 + 1 * (1 + 72 % 5)] % 7

= 5 % 7

= 5,

Now, as 5 is an empty slot,

so we can insert 72 into 5th slot.



Insert key 72 in the hash table

11.7 What is meant by Load Factor in Hashing?

The load factor of the hash table can be defined as the number of items the hash table contains divided by the size of the hash table. Load factor is the decisive parameter that is used when we want to rehash the previous hash function or want to add more elements to the existing hash table.

It helps us in determining the efficiency of the hash function i.e. it tells whether the hash function which we are using is distributing the keys uniformly or not in the hash table.

```
Load Factor = Total elements in hash table/ Size of hash table
```

11.8 What is Rehashing?

As the name suggests, rehashing means hashing again. Basically, when the load factor increases to more than its predefined value (the default value of the load factor is 0.75), the complexity increases. So to overcome this, the size of the array is increased (doubled) and all the values are hashed again and stored in the new double-sized array to maintain a low load factor and low complexity.

11.9 Applications of Hash Data structure

- Hash is used in databases for indexing.
- Hash is used in disk-based data structures.
- In some programming languages like Python, JavaScript hash is used to implement objects.

Real-Time Applications of Hash Data structure

- Hash is used for cache mapping for fast access to the data.
- Hash can be used for password verification.
- Hash is used in cryptography as a message digest.
- Rabin-Karp algorithm for pattern matching in a string.
- Calculating the number of different substrings of a string.

11.10 Advantages of Hash Data structure

- Hash provides better synchronization than other data structures.
- Hash tables are more efficient than search trees or other data structures
- Hash provides constant time for searching, insertion, and deletion operations on average.

11.11 Disadvantages of Hash Data structure

- Hash is inefficient when there are many collisions.
- Hash collisions are practically not avoided for a large set of possible keys.
- Hash does not allow null values.

Practice Problems:

Linear probing is a collision resolving technique in Open Addressed Hash tables. In this method, each cell of a hash table stores a single key–value pair. If a collision occurs by mapping a new key to a cell of the hash table that is already occupied by another key. This method searches the table for the following closest free location and inserts the new key there.

Using the given algorithm below to add some data into the hash table, design a C++ program to Implement Hash Tables with Linear Probing.

Algorithm for Insert:

```
Begin
  Declare Function Insert(int k, int v)
    int hash_val = HashFunc(k)
    intialize init = -1
    intialize delindex = -1
    while (hash_val != init and
    (ht[hash_val]==DelNode::getNode() or ht[hash_val]
      != NULL and ht[hash_val]->k != k))
      if (init == -1)
        init = hash_val
```

```
      if (ht[hash_val] == DelNode::getNode())
        delindex = hash_val
      hash_val = HashFunc(hash_val + 1)
    if (ht[hash_val] == NULL || hash_val == init)
      if(delindex != -1)
        ht[delindex] = new HashTable(k, v)
      else
        ht[hash_val] = new HashTable(k, v)
    if(init != hash_val)
      if (ht[hash_val] != DelNode::getNode())
        if (ht[hash_val] != NULL)
          if (ht[hash_val]->k== k)
            ht[hash_val]->v = v
      else
        ht[hash_val] = new HashTable(k, v)
End.
```

Lab#12
Open Ended Lab

Prepared by: Engr. Amna Arooj

FCSE, GIKI

Lab 12
Open Ended Lab

An open-ended lab is where students are given the freedom to develop their own experiments, instead of merely following the already set guidelines from a lab manual or elsewhere.

Making labs open-ended pushes students to think for themselves and think harder. The students here have to devise their own strategies and back them with explanations, theory and logical justification. This not only encourages students to come up with their experiments, but requires them to defend themselves and their experiment, if questioned.

Different lab classes may vary in their degrees of open-endedness. However, there are three general areas that can be made open-ended:

Concept

In order to make this stage open-ended, the teacher may give the students a Problem Statement with a purpose and not the procedure. The students would then have to come up with their own experiments to back the theory or fulfil the purpose. This helps boost confidence in students, as they can proudly say that they did the experiment on their own.

Table. Schwab/Herron Levels of Laboratory Openness

LEVEL	PROBLEM	WAYS & MEANS/METHODS	ANSWERS
0	Given	Given	Given
1	Given	Given	Open
2	Given	Open	Open
3	Open	Open	Open

We use Schwab and Heron’s scale of laboratory openness. In the first level the teacher posed a problem, which student then tried to answer by using different methods. In the second level the teacher posed a problem, but this time the students had to develop a method themselves. In the third level of inquiry the students had to both pose the problem and develop a suitable method.

Level

Our open-ended lab conforms to level 2 of Schwab/Herron Levels of Laboratory Openness.

I. REFERENCES

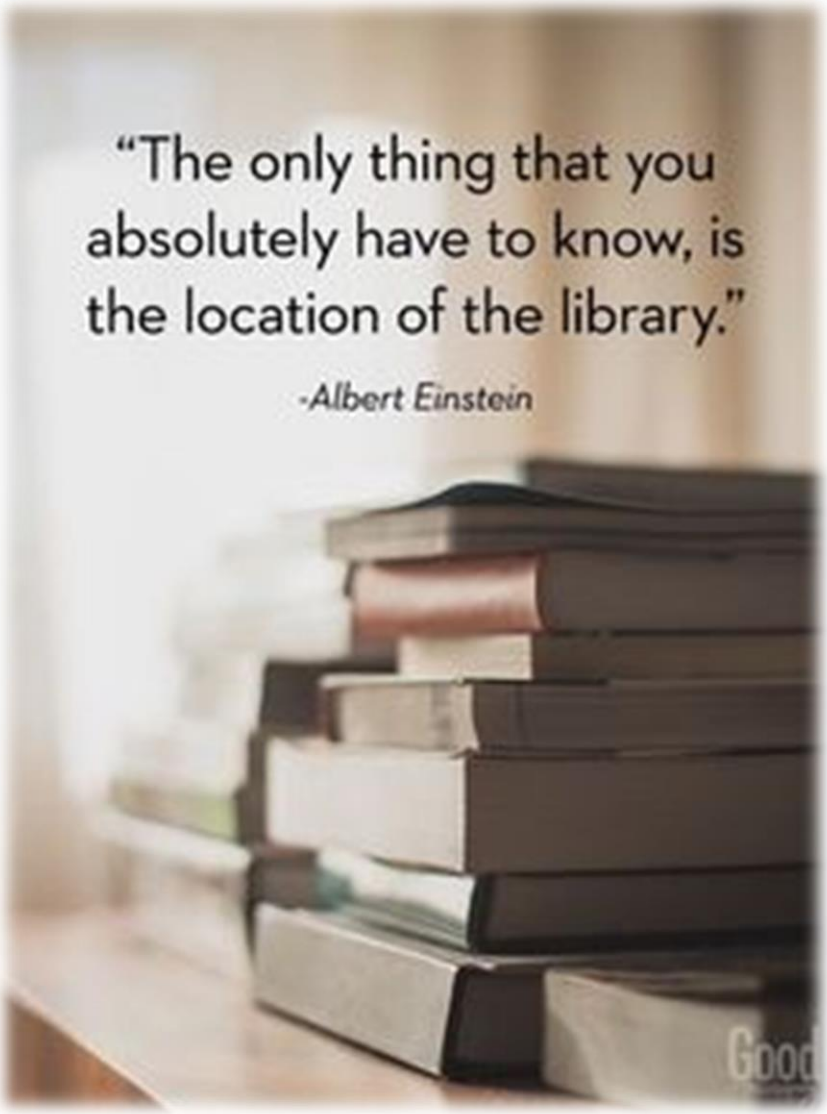
Text Books:

- Data Structures in C++ by D.S Malik
- Data Structure and Algorithm Analysis in C++ by Mark Allen Weiss
- Data Structures and Algorithms in C++ by Robert Lafore

Reference Books:

- Data Structures and Algorithms Made Easy: Data Structure and Algorithmic Puzzles by Narasimha Karumanchi 2011
- Easy Learning Data Structures & Algorithms C++ by Yng Hu, 2019

THE END


"The only thing that you absolutely have to know, is the location of the library."
-Albert Einstein