Lab Manual

CSC211-Data Structures & Algorithms.

CUI

Department of Computer Science
Islamabad Campus

Lab Contents:

Fundamentals of C++; Static List(Array Based), Singly Linked List, Doubly Linked list, Circular Linked list, Stack; Queue; Binary Search Tree, AVL Tree, Heap Tree as Priority Queue; Graph representation by Matrix, Graph traversals, Dijkastra Algorithm; Minimum Spanning Tree; Searching and Hashing Algorithms; Sorting Algorithms.

Student Outcomes (SO)

S.#	**Description**
2	Identify, formulate, research literature, and solve complex computing problems reaching substantiated conclusions using fundamental principles of mathematics, computing sciences, and relevant domain disciplines.
3	Design and evaluate solutions for complex computing problems, and design and evaluate systems, components, or processes that meet specified needs with appropriate consideration for public health and safety, cultural, societal, and environmental considerations.
4	Create, select, adapt and apply appropriate techniques, resources, and modern computing tools to complex computing activities, with an understanding of the limitations.
5	Function effectively as an individual and as a member or leader in diverse teams and in multi-disciplinary settings.

Intended Learning Outcomes

Sr.#	**Description**	**Blooms Taxonomy Learning Level**	**SO**
CLO-4	Implement data structures and algorithms.	*Applying*	2,3,4
CLO-5	Develop a project using appropriate data structures in a team environment.	*Creating*	2-5

Lab Assessment Policy

The lab work done by the student is evaluated using rubrics defined by the course instructor, viva-voce, project work/performance. Marks distribution is as follows:

Assignments	Lab Mid Term Exam	Lab Terminal Exam	Total
25	25	50	100

Note: Midterm and Final term exams must be computer based.

List of Labs

Lab 01

C++ Revision and Array List

Objective:

In this lab, students will get familiar with the new language C++ and its IDE with the help of simple programs that will clear the syntax of the language. Beside learning the basics of C++ like control statements, functions, arrays, etc., you will also learn the concept of Array List.

Activity Outcomes:

This lab teaches you the following topics:

- Basic syntax of C++
- Data types and operators in C++
- Control flow statements in C++
- Arrays and Functions

Instructor Note:

As a pre-lab activity, read fundamental concepts from the text book "C++ How to Program, Deitel, P. & Deitel, H., Prentice Hall, 2019".

1) Useful Concepts

C++, as we all know is an extension to C language and was developed by **Bjarne stroustrup** at bell labs. C++ is an intermediate level language, as it comprises a confirmation of both high level and low level language features.
Following features of C++ makes it a stronger language than C,

1. There is Stronger Type Checking in C++.
2. C++ supports and allows user defined operators (i.e Operator Overloading) and function overloading is also supported in it.
3. Exception Handling is there in C++.
4. Inline Functions in C++ instead of Macros in C language. Inline functions make complete function body act like Macro, safely.
5. Variables can be declared anywhere in the program in C++, but must be declared before they are used.

Installing GNU C/C++ Compiler

Install GCC at Windows you need to install MinGW. To install MinGW, go to the MinGW homepage, www.mingw.org, and follow the link to the MinGW download page. Download the latest version of the MinGW installation program which should be named MinGW-<version>.exe.

While installing MinGW, at a minimum, you must install gcc-core, gcc-g++, binutils, and the MinGW runtime, but you may wish to install more.

Add the bin subdirectory of your MinGW installation to your **PATH** environment variable so that you can specify these tools on the command line by their simple names.

When the installation is complete, you will be able to run gcc, g++, ar, ranlib, dlltool, and several other GNU tools from the Windows command line.

IDE for coding

The IDE we will be using is Dev-C++. Today we will clear our concepts regarding the syntax of C++ with the help of small programs.

Note: we use cin for input and cout for output/print.

At start of your program always use

```
#include<iostream>

using namespace std;
```

Shortcut Keys:

- Press F10 to compile.
- Press F9 to run the complied program.
- Press F11 to compile and run the program.
- For functions in C++ you will have to define them before writing the main method and the main methods always return 0;

2) Solved Lab Activites

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
Activity 1	***5 mins***	***Low***	***CLO-4***
Activity 2	***5 mins***	***Low***	***CLO-4***
Activity 3	***5 mins***	***Low***	***CLO-4***
Activity 4	***5 mins***	***Low***	***CLO-4***
Activity 5	***5 mins***	***Low***	***CLO-4***
Activity 6	***5 mins***	***Low***	***CLO-4***
Activity 7	***5 mins***	***Low***	***CLO-4***
Activity 8	***5 mins***	***Low***	***CLO-4***
Activity 9	***5 mins***	***Low***	***CLO-4***
Activity 10	***5 mins***	***Low***	***CLO-4***
Activity 11	***5 mins***	***Low***	***CLO-4***
Activity 12	***5 mins***	***Low***	***CLO-4***

Activity 1:

Single Line, Multi Line comments and basic Input/ Output in C++.

Solution:

```
// Single line Comment

/*
```

```
* Multiple line
* comment
*/
#include<iostream>
using namespace std;
int main()
{
cout<<"Hello World!";
return 0;
}
```

Output

The output will be “Hello World!”.

Activity 2:

Variables are containers for storing data values.
In C++, there are different types of variables (defined with different keywords), for example:

- **int** - stores integers (whole numbers), without decimals, such as 123 or -123
- **double** - stores floating point numbers, with decimals, such as 19.99 or -19.99
- **char** - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
- **string** - stores text, such as "Hello World". String values are surrounded by double quotes
- **bool** - stores values with two states: true or false

Solution:

```
#include <iostream>
using namespace std;
int main ()
{
  float a=5.5;               // initial value: 5
  int b(3);              // initial value: 3
  int c{2};              // initial value: 2
  float result;            // initial value undetermined

  a = a + b;
  result = a - c;
```

```
    cout << result;


    return 0;

  }
```

Output

Value present in result variable will be shown on Screen

Activity 3:

Using cin, extraction operator (>>) and cout (<<). cin iJava program to illustrate enhanced for loop

```
#include <iostream>

using namespace std;

int main ()

{

int x;
cout << "Type a number: "; // Type a number and press enter
cin >> x; // Get user input from the keyboard
cout << "Your number is: " << x; // Display the input value

return 0;

}
```

Output

Type a number: 100   //suppose user enters 100

Your number is: 100

Activity 4:

Write a program to use string data type in C++

```
#include <iostream>

#include <string>

using namespace std;

int main ()
```

```
{
  string mystring;
  mystring = "This is the initial string content";
  cout << mystring << endl;
  mystring = "This is a different string content";
  cout << mystring << endl;
  return 0;
}
```

Output

This is the initial string content

This is a different string content

Activity 5:

Write a program to use arithmetic operators in C++

```
#include <iostream>
using namespace std;
int main ()
{
  int a, b=3;
  a = b;
  a+=2;             // equivalent to a=a+2
  cout << a;
}
```

Output

5

Activity 6:

Write a program to use relational operators in C++

```
#include <iostream>
using namespace std;
int main ()
{
  int a,b,c;

  a=2;
  b=7;
  c = (a>b) ? a : b;

  cout << c << '\n';
}
```

Output

7

Activity 7:

Write a program to use if-else statement in C++

```
#include <iostream>
using namespace std;
int main ()
{
int x;
cin>>x;
if (x > 0)
  cout << "x is positive";
else if (x < 0)
  cout << "x is negative";
else
```

```
   cout << "x is 0";
 }
```

Output

The ouput depends upon the value of ‘x’ entered by the user, the output will be either x is positive or x is negative or x is 0.

Activity 8:

Write a program to use while loop in C++

```
 #include <iostream>
 using namespace std;
 int main ()
 {
   int n = 10;

   while (n>0) {
     cout << n << ", ";
     --n;
   }

   cout << "liftoff!\n";
 }
```

Output

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff!
```

Activity 9:

Write a program to use do-while loop in C++

```
#include <iostream>

#include <string>

using namespace std;


int main ()

{

  string str;

  do {

    cout << "Enter text: ";

    getline (cin,str);

    cout << "You entered: " << str << '\n';

  } while (str != "goodbye");

}
```

Output

Program repeatedly takes a string as input then displays it saying "You entered…." Until user types goodbye.

Activity 10:

Write a program to apply the switch statements.

```
#include <iostream>

using namespace std;

int main ()

{

int day = 4;
switch (day) {
  case 1:
    cout << "Monday";
    break;
  case 2:
    cout << "Tuesday";
    break;
  case 3:
    cout << "Wednesday";
    break;
  case 4:
    cout << "Thursday";
    break;
  case 5:
    cout << "Friday";
    break;
  case 6:
    cout << "Saturday";
    break;
  case 7:
    cout << "Sunday";
    break;

default:

cout<<"Enter valid day";
}


}
```

Output

Outputs "Thursday" as variable day is initialized to 4

Activity 11:

Write a program that will add two numbers in a function and call that function in main.

```
#include <iostream>

using namespace std;

int addition (int a, int b)

{

  int r;

  r=a+b;

  return r;

}


int main ()

{

  int z;

  z = addition (5,3);

  cout << "The result is " << z;

}
```

Output

The result is 8

Activity 12:

Write a program to use the concept of Arrays in C++

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

int foo [] = {16, 2, 77, 40, 12071};
int n, result=0;

int main ()
{
  for ( n=0 ; n<5 ; ++n )
  {
    result += foo[n];
  }
  cout << result;
  return 0;
}
```

Output

12206

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Write down a program that find $\sum X^2$, where input for X and starting and stopping value is entered by the user.

Lab Task 2

Write the computer program to apply the concepts of Array List. The array list will include the following functions:

1. *Insert the value at end of the list*
2. *Insert the value at start of the list*
3. *Insert the value after specific value*
4. *Insert the value before specific value*
5. *Display the array list*
6. *Delete the value from end of the list*
7. *Delete the value from start of the list*
8. *Delete specific value*

Lab Task 3

Using while loop apply liear search on the Array List you developed in Lab Task 2.

Lab 02

Singly Linked List

Objective:

The purpose of this lab session is to acquire skills in working with singly linked lists.
lab will give you an overview of C++ language.

Activity Outcomes:

This lab teaches you the following topics:

- Creation of singly linked list
- Insertion in singly linked list
- Deletion from singly linked list
- Traversal of all nodes

Instructor Note:

As a pre-lab activity, read Chapter 14 from the text book "A Common-Sense Guide to Data Structures and Algorithms, Jay Wengrow, Pragmatic Bookshelf, 2020.".

1) Useful Concepts

A *list* is a finite ordered set of elements of a certain type. The elements of the list are called *cells* or *nodes*. A list can be represented *statically*, using arrays or, more often, *dynamically*, by allocating and releasing memory as needed. In the case of static lists, the ordering is given implicitly by the one-dimension array. In the case of dynamic lists, the order of nodes is set by *pointers*. In this case, the cells are allocated dynamically in the heap of the program. Dynamic lists are typically called *linked lists*, and they can be singly- or doubly-linked.

The structure of a node may be:

```
struct Nodetype
  {
  int data; /* an optional field */
    ... /* other useful data fields */
  Nodetype *next=NULL; /* link to next node, assigned NULL so that should not point garbage*/
  };
Nodetype *first=NULL, *last=NULL;  /* first and last pointers are global and point first and last node */
```

A singly-linked list may be depicted as in Figure 1.1.

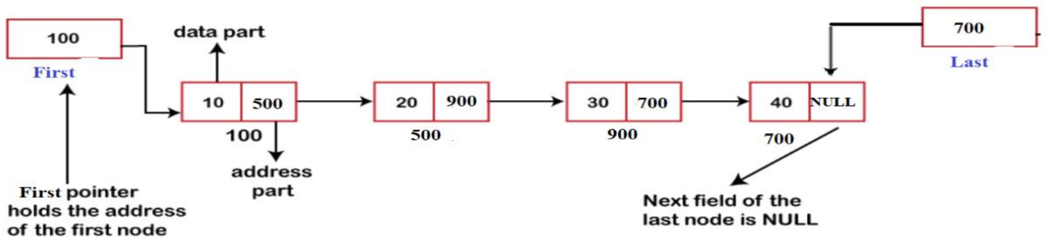

100
First
data part
10
500
100
address part
20
900
500
30
700
900
40
NULL
700
700
Last
First pointer holds the address of the first node
Next field of the last node is NULL

Figure 1.1.: A singly-linked list model.

2) Solved Lab Activites

Sr.No	***Allocated Time***	***Level of Complexity***	***CLO Mapping***
Activity 1	***10 mins***	***Medium***	***CLO-4***
Activity 2	***10 mins***	***Medium***	***CLO-4***
Activity 3	***5 mins***	***Medium***	***CLO-4***
Activity 4	***10 mins***	***Medium***	***CLO-4***
Activity 5	***5 mins***	***Medium***	***CLO-4***
Activity 6	***10 mins***	***Medium***	***CLO-4***

Activity 7	***5 mins***	***Medium***	***CLO-4***
Activity 8	***5 mins***	***Medium***	***CLO-4***

Activity 1:

Insertion at the end of the linked list

Consider the following steps which apply to dynamic lists:

```
Void insert_end()
{
//take the pointer to hold the address of nodetype record
Nodetype *p;
//allocate runtime memory for new record of nodetype using new operator
p = new Nodetype ;
cout<<"Enter the data in node:   ";
cin>>p->data;
/*  l i s t  i s   empty  */
i f  (  first == NULL )
/* p becomes first node and first and last pointer will point to same node */
     first=last = p;
else {
/* link the last pointer with newly created node (p) */
    last->next  = p;
    last = p;             /* assign p to last node */
     }
}
```

Output

Will insert the new node at the end of the linked list.

Activity 2:

Insertion at the start of the linked list

```
Consider the following steps which apply to dynamic lists:
```

```
Void insert_start()
{
//take the pointer to hold the address of nodetype record
Nodetype *p;
//allocate runtime memory for new record of nodetype using new operator
p = new Nodetype ;
cout<<"Enter the data in node:   ";
cin>>p->data;
i f  (  first == NULL )  /*  l i s t  i s   empty  */
/* p becomes first node and first and last pointer will point to same node */
        first=last = p;
else {
     p->next=first;   /* link the new node with first node */
   fisrt = p;             /* assign p to last node */
     }
}
```

Output

Will insert the new node at the end of the linked list.

Activity 3:

Accessing nodes of linked list

```
Nodetype search (int key){
Nodetype *p = NULL;
p =  f i r s t ;
while  (  p != NULL  && p->data != key)
{
p = p->next ;
}
return p;  /* if p is NULL then value not found  */
}
```

Output

Return the pointer of Node if the required element is found.

Activity 4:

Inserting after specific value

```
void insert_after (int key){
Nodetype *p = NULL;
p=search(key);
if(p==NULL)
cout<<"value not found";
else
{
   Notetype *Newnode;
   Newnode= new Notetype;
if(p==last)
    {
  last->next=p;
  last=p;
  }
else
{
 Newnode->next=p->next;
 p->next=Newnode;
}
 Cout<<"New node linked successfully";
}
```

Output

Will insert the new node after specified node

Activity 5:

Deleting first node from single linked list

```
Void delete_first()
{
Nodetype *p;
i f  (  f i r s t  == NULL )  /*  l i s t  i s  empty  */
    cout<<"\n Linked List is empty";
else
{  /*  non-empty  l i s t  */
    p =  f i r s t ;
    f i r s t  =  f i rst ->next ;
    delete (  p  ) ;  /*  free  up memory  */
}
}
```

Output

Will delete the first node of the singly linked list.

Activity 6:

Deleting last node from single linked list

```
Void delete_last()
{
NodeT *q,  *q1;
q1 = NULL;  /*  i n i t i a l i z e  */
q =  f i r s t ;
if(q==NULL)
{
    Cout<<"\n Linked List is empty";
}
else
{    /*  non-empty  l i s t  */
while (  q !=  last  )
{    /*  advance  towards  end  */
q1 = q;      /*q1 will follow the q pointer */
q = q->next ;
}
i f  (  q ==  f i r s t  )
{  /*  only  one  node  */
f i r s t  =  last = NULL;
}
else
{  /*  more  than  one  node  */
q1->next = NULL;
last = q1;
}
delete q; } }
```

Activity 7:

Deleting a node given by user

```
void remove_spec(int key)
{
Nodetype *q,  *q1;
q1 = NULL;  /*  i n i t i a l i z e  */
q =  f i r s t ;
/*  search  node  */
while (  q != NULL && q->data != key )
{
q1 = q;
q = q->next ;
}
i f  (  q == NULL )
{    cout<<" Not found  supplied  key";
}
else if(q==first && q==last)
{
  delete q;
  first=last=NULL;
}
else if (q==last)
{
 q1->next=NULL;
 last=q1;    /* make 2nd last node as last node */
 delete q;
}
else     /*  other  than  f i r s t  node and last */
{
q1->next = q->next ;
delete q; } }
```

Output

Will delete the user given node of the singly linked list.

Activity 8:

Complete deletion of single linked list

```
 while  (  f i r s t  != NULL )
 {
 p =  f i r s t ;
 f i r s t  =  f i rst ->next ;
 free (  p  ) ;
 }
 last = NULL;
```

Output

All the nodes of Linked List will be deleted.

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Add functionality to display the Link List in reverse (both using loops and using recursive approach).

Lab Task 2

Add a function that merges two linked lists passed as parameter into a third new linked list.

Lab Task 3

Add a function to find multiple occurrences of data element in the list..

Lab 03

Doubly Linked List

Objective:

This lab session is intended to help you develop the operations on doubly-linked lists.

Activity Outcomes:

This lab teaches you the following topics:

- Creation of doubly linked list
- Insertion in doubly linked list
- Deletion from doubly linked list
- Traversal of all nodes

Instructor Note

As a pre-lab activity, read Chapter 14 from the text book "A Common-Sense Guide to Data Structures and Algorithms, Jay Wengrow, Pragmatic Bookshelf, 2020.".

1) Useful Concepts

A *doubly-linked* list is a (dynamically allocated) list where the nodes feature two relationships: *successor* and *predecessor*. A model of such a list is given in figure 4.1. The type of a node in a doubly-linked list may be defined as follows:

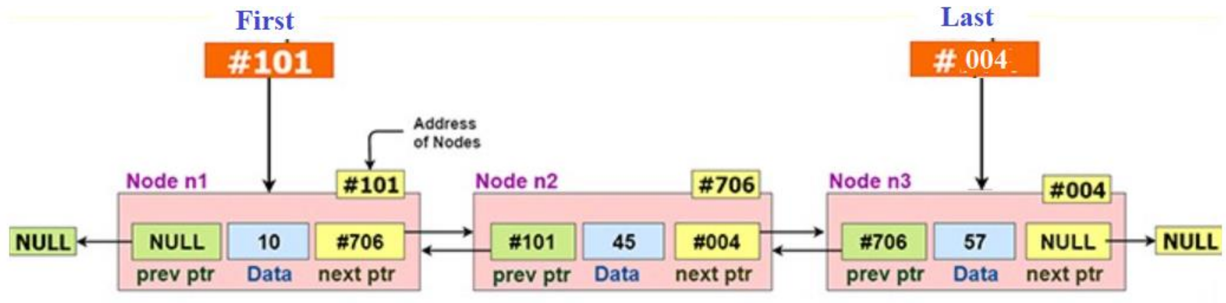


Figure: 4.1

struct *Nodetype*
{
Data members;
struct *Nodetype *next ;*   */* pointer to next node */*
struct *Nodetype *prev ;*   */* pointer to previous node */*
};
*Node *first=NULL, *last=NULL;*

2) Solved Lab Activites

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
Activity 1	*10 mins*	*Medium*	*CLO-4*
Activity 2	*10 mins*	*Medium*	*CLO-4*
Activity 3	*15 mins*	*Medium*	*CLO-4*

Activity 4	***15 mins***	***Medium***	***CLO-4***
Activity 5	***10 mins***	***Medium***	***CLO-4***

Activity 1:

Creation of Doubly linked list

```
Consider the following steps which apply to dynamic lists:
Void insert_end()
{
//take the pointer to hold the address of nodetype record
  Nodetype *p;
//allocate runtime memory for new record of nodetype using new operator
  p = new Nodetype ;
 cout<<"Enter the data in node:   ";
cin>>p->data;
i f  (  first == NULL )  /*  l i s t  i s   empty  */
/* p becomes first node and first and last pointer will point to same node */
   first=last = p;
else {
    last->next  = p;   /* link the last pointer with newly created node (p) */
    p->prev=last;    /* link the new node's previous pointer to last node */
    last = p;             /* assign p to last node */
     }
}
```

Activity 2:

Accessing nodes of Doubly linked list

```
Stating at a certain position (i.e. at a certain node) we can access a list:

In sequential forward direction

for  (  p = L->f i r s t ;  p != NULL;  p = p->next  )

{

/*  some  operation  o  current  cell  */

}

In sequential backward direction

for  (  p = L->l ast ;  p != NULL;  p->p->prev  )

{

/*  some  operation  o  current  cell  */

}
```

Activity 3:

Insertion in doubly linked list

```
We can insert a node before the first node in a list, after the last one, or at
a position specified by a given key:

before the first node:

i f  (  L->f i r s t  == NULL )

{  /*  empty  l i s t  */

L->f i r s t  = L->l ast = p;

p_>next = p->prev = NULL;

}

else

{  /*  nonempty  l i s t  */ p->next = L->f i r s t ; p->prev = NULL; L->f i
rst ->prev = p; L->f i r s t  = p;

}

after the last node:

i f  (  L->f i r s t  == NULL )

{  /*  empty  l i s t  */

L->f i r s t  = L->l ast = p;

p_>next = p->prev = NULL;
}

else

{  /*  nonempty  l i s t  */ p->next = NULL; p->prev = L->l ast ; L->last->next
= p; L->l ast = p;

}

After a node given by its key:

p->prev = q;

p->next = q->next ;

i f  (  q->next != NULL )  q->next->prev = p;q->next = p;

i f  (  L->l ast == q )  L->l ast = p;


Here we assumed that the node with the given key is present on list L and it we
found it and placed a pointer to it in variable q.
```

Activity 4:

Deleting a node from doubly linked list

When deleting a node we meet the following cases:

- ***Deleting the first node:***

```
p = L->f i r s t ;

L->f i r s t  = L->f i rst ->next ;  /*  nonempty  l i s t  assumed  */

free (  p  ) ;  /*  release  memory taken  by  node  */

i f  (  L->f i r s t  == NULL )

L->l ast == NULL;  /*  l i s t  became  empty  */

else

L->f i rst ->prev = NULL;
```

- ***Deleting the last node:***

```
p = L->last ;

L->last = L->last->prev ;  /*  nonempty  l i s t  assumed  */

i f  (  L->l ast == NULL )

L->f i r s t  = NULL;  /*  l i s t  became  empty  */

else
```

```
L->last->next = NULL;

free ( p ) ; /* release memory taken by node */
```

Deleting a node given by its key. We will assume that the node of key givenKey exists and it is pointed to by p (as a result of searching for it)

```
i f ( L->f i r s t == p && L->l ast == p )

{ /* l i s t has a single node */

L->f i r s t = NULL;

L->l ast = NULL; /* l i s t became empty */

free ( p ) ;

}

else

i f ( p == L->f i r s t )

{

/* deletion of f i r s t node */

 L->f i r s t = L->f i rst ->next ; L->f i rst ->prev = NULL;

free ( p ) ;

}

else

{

/* deletion of an inner node */

 p->next->prev = p->prev ; p->prev->next = p->next ;

free ( p ) ;

}
```

Activity 5:

Complete deletion of Doubly linked list

```
Deleting a list completely means deleting each of its nodes, one by one.

NodeT *p;

while ( L->f i r s t != NULL )

{

p = L->f i r s t ;
L->f i r s t  = L->f i rst ->next ;
free (  p  ) ;
}
L->l ast = NULL;
```

Output

All the nodes of Doubly Linked List will be deleted.

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Write a function which reverses the order of a doubly linked list.

Lab Task 2

Write a function which takes two values as input from the user and searches them in the list. If both the values are found, your task is to swap both the nodes in which these values are found. Note, that you are not supposed to swap values.

Lab Task 3

Write a function that takes a singly linked list as parameter and creates a doubly linked list for the same data present in the singly linkd list i.e. for user the singly linked list will be converted into doubly linked list.

Lab 04

Circular Linked List

Objective:

In this lab session we will enhance singly-linked lists with another feature: we'll make them circular. And, as was the case in the previous lab session, we will show how to implement creation, insertion, and removal of nodes.

Activity Outcomes:

This lab teaches you the following topics:

- Creation of circular linked list
- Insertion in circular linked list
- Deletion from circular linked list
- Traversal of all nodes

Instructor Note

- As a pre-lab activity, read Chapter 14 from the text book "A Common-Sense Guide to Data Structures and Algorithms, Jay Wengrow, Pragmatic Bookshelf, 2020.".

1) Useful Concepts

A circular singly linked list is a singly linked list which has the last element linked to the first element in the list. Being circular it really has no ends; then we’ll use only one pointer pNode to indicate one element in the list – the newest element. Figure 3.1 show a model of such a list. pNode

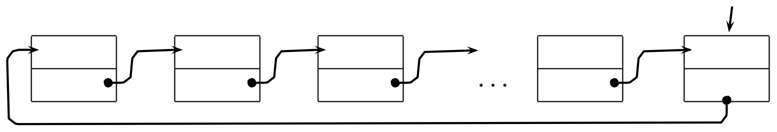


Figure 3.1.: A model of a circular singly-linked list.

The structure of a node may be:

```
struct nodetype
{
int key ; /* an optional f i e l d */
/* other useful data f i e l ds */
struct nodetype *next ;
/* l ink to next node */
};
```

2) Solved Lab Activites

Sr.No	***Allocated Time***	***Level of Complexity***	***CLO Mapping***
Activity 1	***10 mins***	***Medium***	***CLO-4***
Activity 2	***15 mins***	***Medium***	***CLO-4***
Activity 3	***15 mins***	***Medium***	***CLO-4***
Activity 4	***10 mins***	***Medium***	***CLO-4***
Activity 5	***10 mins***	***Medium***	***CLO-4***

Activity 1:

Creation of Circular linked list

Consider the following steps which apply to dynamic lists:

Initially, the list is empty. This can be coded by setting the pointers to the first and last cells to the special value

NULL, i.e. first = NULL, last = NULL.

Reserve space for a new node to be inserted in the list:

```
/*  reserve  space  */

p =  new nodetype ;
```

Then place the data into the node addressed by p. The implementation depends on the type of data the node holds. For primitive types, you can use an assignment such as ∗p = data.

Make the necessary connections:

```
p−>next = NULL;      /*  node  i s  appended  to  the  l i s t  */

i f  (  last  != NULL )  /*  l i s t  i s  not  empty  */

last−>next = p;

else

f i r s t  = p;  /*  f i r s t  node  */

last = p; }
```

Activity 2:

Accessing nodes of Circular linked list

```
NodeT *p;

p = pNode;

i f   (  p != NULL )

do

{

access current node and get data ;

p = p−>next ;

}

while  (  p != pNode ) ;
```

Another choice is to look for a key, say givenKey. Code for such list scan is given below:

```
NodeT *p;

p = pNode;
```

```
i f  (  p != NULL )
do
{
i f  (  p->key = givenKey )
{  /*  key  found  at  address  p  */
return p;
}
p = p->next ;
}
while  (  p != NULL ) ;
return NULL;  /*  not  found  */
}
```

Output

Return the index of Node if the required element is found.

Activity 3:

Insertion in circular linked list

```
Inserting Before a node with key givenKey
There are two steps to execute:
Find the node with key givenKey:
NodeT *p,  *q,  *q1;
q1 = NULL;  /*  i n i t i a l i z e  */
q = pNode;
do
{
q1 = q;
q = q->next ;
i f  (  q->key == givenKey )  break;
}
while (  q != pNode ) ;
Insert the node pointed to by p, and adjust links:
i f  (  q->key == givenKey )
{  /*  node  with  key  givenKey  has  address  q  */
q1->next = p;

p->next = q;

}

Insertion after a node with key

Again, there are two steps to execute:


Find the node with key givenKey:NodeT *p,  *q;

q = pNode;

do

{

i f  (  q->key == givenKey )  break;

q = q->next ;

}

while (  q != pNode ) ;

Insert the node pointed to by p, and adjust links:

i f  (  q->key == givenKey )

{  /*  node  with  key  givenKey  has  address  q  */

p->next = q->next ;

q->next = p;

}
```

Activity 4:

Deleting a node from circular linked list

Again there are two steps to take:

Find the node with key givenKey:

```
NodeT *p,  *q,  *q1;

q = pNode;

do

{

q1 = q;

q = q->next ;
i f  (  q->key == givenKey )  break;

}

while (  q != pNode ) ;

Delete the node pointed to by q. If that node is pNode then we adjust pNode to
point to its previous.

i f  (  q->key == givenKey )

{  /*  node  with  key  givenKey  has  address  q  */

i f  (  q == q->next  )

{

/*  l i s t  now empty  */

}

else

{

q1->next = q->next ;

i f  (  q == pNode )  pNode = q1;

}

free (  q  ) ;

}
```

Activity 5:

Complete deletion of Circular linked list

```
NodeT *p,  *p1;

p = pNode;

do

{

p1 = p;

p = p->next ;

free (  p1  ) ;

}

while  (  p != pNode ) ;

pNode = NULL;
```

Output

All the nodes of Circular Linked List will be deleted.

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Write a function that deletes all those nodes from a linked list which have even/odd numbered value in their data part.

Lab Task 2

Write a function that implements Josephus problem.

Lab Task 3

Write a function that deletes all even positioned nodes from a linked list. Last node should also be deleted if its position is even.

Lab 05

Stack

Objective:

This lab will introduce you the concept of Stack data structure

Activity Outcomes:

This lab teaches you the following topics:

- How to access the top of the stack
- How to push data onto the stack
- How to pop data from the stack

Instructor Note

As a pre-lab activity, read Chapter 09 from the text book “A Common-Sense Guide to Data Structures and Algorithms, Jay Wengrow, Pragmatic Bookshelf, 2020.”.

1) Useful Concepts

A *stack* is a ordered collection of elements in which the elements can be accessed from a sible end known as Top od Stack, it works in LIFO order. Its operations are:

push − push an element onto the top of the stack;

pop − pop an element from the top of the stack;

top − retrieve the element at the top of the stack;

delete − delete the whole stack. This can be done as explained in the previuos paragraph.

2) Solved Lab Activites

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
Activity 1	***15 mins***	***Medium***	***CLO-4***
Activity 2	***15 mins***	***Medium***	***CLO-4***
Activity 3	***15 mins***	***Medium***	***CLO-4***
Activity 4	***15 mins***	***Medium***	***CLO-4***

Activity 1:

Implement a Stack in Array, use class in the implementation so that Stack can be treated as an object and its operations as public interface for the user.

```
#include <iostream>

using namespace std;

class Stack

{

     private:

             int tos;
```

```
            char values[];
        public:
            Stack(int x)
            {
                tos = -1;
                values[x];
            }

            void push(char x)
            {
                if(tos==9)
                    cout<<"\nStack overflow condition";
                else
                    values[++tos] = x;
            }

            char pop()
            {
                if(tos == -1){

                    cout<<"\nStack Underflow condition";
                    return ' ';
                }
                else
                    return values[tos--];
            }
};
int main()
{
        Stack* myStack = new Stack(10);
        myStack->push('B');
        myStack->push('S');
```

```
      myStack->push('S');
      myStack->push('E');
      myStack->push('3');
      myStack->push('Z');

      cout<<myStack->pop();
      cout<<myStack->pop();
      cout<<myStack->pop();
      cout<<myStack->pop();
      cout<<myStack->pop();
      cout<<myStack->pop();
      cout<<myStack->pop();

 }
```

Output:

Z3ESSB

Stack Underflow Condition

Activity 2:

Implement Dynamic Stack.

```
 #include <iostream>
 #include <conio.h>
 using namespace std;

 struct Node
 {
        char data;
        Node* link;
```

```
};

class LinkedStack
{
      private:
              Node* head;
      public:
             LinkedStack()
             {
                head = NULL;
             }

void push(char e)
{

     Node* temp = new Node;
     temp->data = e;
     temp->link = head;
     head = temp;

}

char pop()
      {
           if(head == NULL)
             cout<<"\nStack is Empty .... Underflow Codition\n";
           else
           {
               Node* temp = head;
               char T = temp->data;
               head = temp->link;
```

```
                delete temp;
                return T;

                     }
               }
                      };
    int main()
    {
        LinkedStack myStack;
        myStack.push('H');
        myStack.push('e');
        myStack.push('l');
        myStack.push('l');
        myStack.push('o');

        cout<<myStack.pop();
        cout<<myStack.pop();
        cout<<myStack.pop();
        cout<<myStack.pop();
        cout<<myStack.pop();

        }
```

Output:

olleH

Activity 3:

Implement a Stack in Array, use class in the implementation so that Stack can be treated as an object and its operations as public interface for the user.

```
 main()
 {
    Stack s1;
```

```
   char str[]="I Love Programming";
   int len = strlen(str);
   for(int i=0; i<len; i++)
        s1.push(str[i]);
   for(i=0; i<len; i++)
      s1.pop();
}
```

Output:

gnimmargorP evoL I

Activity 4:

Using the stack check that the given expression has balanced paranthesis or not.

```
#include<iostream>
using namespace std;
bool isBalanced(string expr) {
   stack s;
   char ch;
   for (int i=0; i<expr.length(); i++) {
      if (expr[i]=='('||expr[i]=='['||expr[i]=='{') {
         s.push(expr[i]);
         continue;
      }
      if (s.empty())
         return false;
         switch (expr[i]) {
            case ')':
               ch = s.top();
               s.pop();
               if (ch=='{' || ch=='[')
                  return false;
                  break;
            case '}':
                    ch = s.top();
                    s.pop();
                    if (ch=='(' || ch=='[')
                        return false;
                        break;
                case ']':
                    ch = s.top();
                    s.pop();
                    if (ch =='(' || ch == '{')
                        return false;
                        break;
            }
        }
        return (s.empty()); //when stack is empty, return true
}
main() {
    string expr = "[{}(){()}]";
    if (isBalanced(expr))
        cout << "Balanced";
    else
        cout << "Not Balanced";
}
```

***Input**: exp = "[()]{}{[()()]()}"*
***Output**: Balanced*
***Input**: exp = "[(])"*
***Output**: Not Balanced*

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Write an application using Stack that checks whether the entered string of brackets is balanced, e.g.

[{( )}]   is Balanced while {[( )] } is not Balanced because the priority of the pranthesis is not maintained.

Lab Task 2

Use dynamic stack and implement Infix to Postfix conversion algorithm and test it for various inputs.

Lab Task 3

For a given postfix expression, use dynamic stack to evaluate a numerical result for given values of variables.

Lab 06

Queue

Objective:

This lab will introduce you the concept of Queue data structure

Activity Outcomes:

This lab teaches you the following topics:

- How to access the front and rear pointers
- How to enqueuer/insert the data
- How to dequeuer/ delete the data

Instructor Note

As a pre-lab activity, read Chapter 09 from the text book "A Common-Sense Guide to Data Structures and Algorithms, Jay Wengrow, Pragmatic Bookshelf, 2020.".

1) Useful Concepts

A *queue* is ordered collection of items which works according to FIFO (First In First Out) algorithm. Of the two ends of the queue, one is designated as the *front* − where elements are extracted (operation called *dequeue*), and another is the *rear*, where elements are inserted (operation called *enqueue*). A *queue* may be depicted as in Figure below:

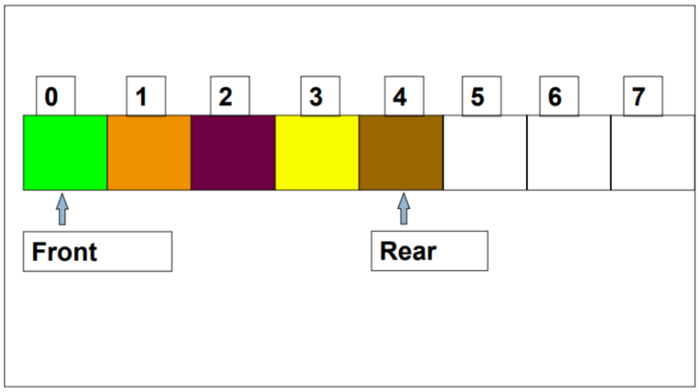


Fig 6.1: Queue with Front & Rear pointer

The main operations are:
Enqueue − place an element at the tail of the queue;
Dequeue − take out an element form the front of the queue;
Delete − delete the whole queue

2) Solved Lab Activites

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
Activity 1	***15 mins***	***Medium***	***CLO-4***
Activity 2	***15 mins***	***Medium***	***CLO-4***
Activity 3	***15 mins***	***Medium***	***CLO-4***
Activity 4	***15 mins***	***Medium***	***CLO-4***

Activity 1:

Implement an Array based FIFO Queue.

```
public class Que    //why class name is not queue
{
public :
int size; // default capacity
int *q;          // array that holds queue elements
int front;           // index of front of queue
int rear;                // index of rear of queue
Que()       //default constructor
{
   size = 10;           //default size
   q = new int[size];   // run time size allocation
   front=-1;     // initially front and rear are at -1
   rear=-1;
}

Que(int x)       //overloaded constructor
{
   size = x;            //user given size
   q = new int[size];   // run time size allocation
   front=-1;     // initially front and rear are at -1
   rear=-1;
}
bool is_empty()
{
   if (rear==-1)
         return true;
   else
         return false;
}
bool is_full()
```

```
{
      if (rear==SIZE-1)
                  return true;
      else
                  return false ;
}
Void display()
{
  if(is_empty())
{
 cout<<"\n Queue is empty….";
}
else{
  for(int i=front; i<=rear; i++)
      {
      cout<<"\n Value at index "<<i<< " is: "<<q[i];
}
void Enqueue(int x)
{
      if (is_full())
      {
                  cout<<"No space ";
      }
      else {
           if(is_empty())
           {
                front=rear=0;
            }
             else{
                      rear++;
                 }
                  q[rear]=x;
            }
    }
    int Dequeue()
    {
          if (is_empty())
          {
            cout<<"Queue is already empty";
            return -1;
          }
          else
          {
            int x= q[front];
          if(front==rear)
            {
              front=rear=-1;
        }else{
                      front++;
          }
          return x;
     }
    Void main ()
    {
    Que q1;          //Default constructor: object of que class with default size
    Que q2(5);        //Overloaded constructor: object of que class (q2) with size 5
    Que q3(15);
    q1.enqueue(5);
    q2.enqueue(7);
    q3.enqueue(6);
    int x= q2.dequeue();
    cout<<"Dequeued element from q3 is: "<<x;
return 0;
}
```

Activity 2:

Elements				**7**	**5**	**2**
Indexes	**0**	**1**	**2**	**3**	**4**	**5**

Front = 3 Rear = 5

What if we call Enqueue (3)? → Queue is full

but the queue is not full

Question:
What to do now? →

4	5	7	2				
0	1	2	3	4	5	6	7

Front=0 Rear = 3

5	**7**	**2**	**2**				
0	1	2	3	4	5	6	7

After Dequeue();
Front=0 Rear = 2
Value at index 3 is now garbage as it is repeated, so
after new equeue() new value will be at index 3.
Hence value 2 at index 3 will be overwritten.

Write the shift left method for dequeue operation.

```
Int deQue()
            {
                int temp;
                if(!isEmpty())
                {
                  temp = que[front];
                  if(front==rear)
                    front=rear=-1;          //make the queue empty
                  else
                 shift_left(front, rear);
                 rear--;
                 }
                 else
                 {
                     cout<<"\nSorry Q is Empty ";
                     temp = -1;
                 }
                 return temp;
             }

Void shift_left(int front, int rear)
{
   Int x=front;
   While(x+1!=rear)
   {
      Que[x]= Que[x+1];
   }
}
```

Activity 3:

*Consider the activity-1 where there was an issue of space and to resolve this we applied the shifting. The problem with shifting is that it increases the complexity. The solution of this is circular queue. Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. We apply the formula (**Rear=(Rear+1) mod Queue Size)** to move the rear to next index.*

Implement an Array based circular Queue that is generic in nature (use template classes), apply proper checks of Queue Full and Queue Empty before Enque and DeQue operations.

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

```
template <class T>
class Q
{
      private:
              T Data[10];
              int front;
              int rear;
              int count;
      public:
             Q() { front = rear = count =0;}

             bool isEmpty()  { return (count==0); }
             bool isFull()   { return (count==10); }
             void enQue(T);
             T deQue();

};

template <class T>
void Q <T> :: enQue(T x)
{

                  if (!isFull())
                  {
                      Data[rear] = x;
                      rear++;
                      rear = rear % 10;
                      count++;
                      }
                  else
                  {
                      cout<<"\nSorry Q is Full ";
```

```
                        }
                }
template <class T>

T Q<T> ::deQue()
            {
                T temp;
                if(!isEmpty())
                {
                   temp = Data[front];
                   front++;
                   front = front%10;
                   count--;
                }
                else
                {
                    cout<<"\nSorry Q is Empty ";
                    temp = -1;
                }
                return temp;
            }
int main()
{
     Q <int> q1;
    Q <char> q2;
    Q <string> q3;
     q1.enQue(4);
     q1.enQue(7);
     q1.enQue(9);

     cout<<"\n Dequeued value is: "<<q1.deQue();
    q2.enQue('A');
```

```
      q2.enQue('B');
      q2.enQue('C');
      cout<<"\n Dequeued value is: "<<q2.deQue();

      q3.enQue("I");
      q3.enQue("Love");
      q3.enQue("Programming");
      cout<<"\n Dequeued value is: "<<q3.deQue();
      cout<<"\n Dequeued value is: "<<q3.deQue();
      cout<<"\n Dequeued value is: "<<q3.deQue();
  }
```

Activity 4:

Double ended queue or simply called "Deque" is a generalized version of Queue. The difference between Queue and Deque is that it does not follow the FIFO (First In, First Out) approach. The second feature of Deque is that we can insert and remove elements from either front or rear ends.Some basic operations of dequeue are:

insert_at_beg(): inserts an item at the front of Dequeue.

insert_at_end(): inserts an item at the rear of Dequeue.

delete_fr_beg(): Deletes an item from front of Dequeue.

delete_fr_rear(): Deletes an item from rear of Dequeue.

Write down the C++ program to implement the Double Ended Queue.

```
#include<iostream>
using namespace std;
#define SIZE 10
class dequeue {
   int a[20],f,r;
   public:
      dequeue();
      void insert_at_beg(int);
      void insert_at_end(int);
```

```
    void delete_fr_front();
    void delete_fr_rear();
    void show();
};
dequeue::dequeue() {
  f=-1;
  r=-1;
}
void dequeue::insert_at_end(int i) {
  if(r>=SIZE-1) {
    cout<<"\n insertion is not possible, overflow!!!!";
  } else {
    if(f==-1) {
      f++;
      r++;
    } else {
      r=r+1;
    }
    a[r]=i;
    cout<<"\nInserted item is"<<a[r];
  }
}
void dequeue::insert_at_beg(int i) {
  if(f==-1) {
    f=0;
    a[++r]=i;
    cout<<"\n inserted element is:"<<i;
  } else if(f!=0) {
    a[--f]=i;
    cout<<"\n inserted element is:"<<i;
  } else {
    cout<<"\n insertion is not possible, overflow!!!";
  }
}
void dequeue::delete_fr_front() {
  if(f==-1) {
    cout<<"deletion is not possible::dequeue is empty";
    return;
  }
  else {
    cout<<"the deleted element is:"<<a[f];
    if(f==r) {
      f=r=-1;
      return;
    } else
      f=f+1;
    }
  }
  void dequeue::delete_fr_rear() {
    if(f==-1) {
      cout<<"deletion is not possible::dequeue is empty";
      return;
    }
    else {
      cout<<"the deleted element is:"<<a[r];
      if(f==r) {
        f=r=-1;
      } else
        r=r-1;
    }
  }
  void dequeue::show() {
    if(f==-1) {
      cout<<"Dequeue is empty";
    } else {
      for(int i=f;i<=r;i++) {
        cout<<a[i]<<" ";
      }
    }
  }
  int main() {
    int c,i;
    dequeue d;
    Do//perform switch opeartion {
    cout<<"\n 1.insert at beginning";
    cout<<"\n 2.insert at end";
    cout<<"\n 3.show";
    cout<<"\n 4.deletion from front";
    cout<<"\n 5.deletion from rear";
    cout<<"\n 6.exit";
    cout<<"\n enter your choice:";
    cin>>c;
    switch(c) {
      case 1:
        cout<<"enter the element to be inserted";
        cin>>i;
        d.insert_at_beg(i);
      break;
      case 2:
        cout<<"enter the element to be inserted";
        cin>>i;
        d.insert_at_end(i);
      break;
      case 3:
        d.show();
      break;
```

```
        case 4:
          d.delete_fr_front();
        break;
        case 5:
          d.delete_fr_rear();
        break;
        case 6:
          exit(1);
        break;
        default:
          cout<<"invalid choice";
        break;
      }
    } while(c!=7);
  }
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Implement the methods developed in Activity 1 for Dynmaic Queue i.e. Linked Implementation of the Queue.

Lab Task 2

Implement the ***Deque*** *in linked list.*

Lab 07

Binary Search Tree (BST)

Objective:

This lab will introduce you the concept and implementation of BST.

Activity Outcomes:

This lab teaches you the following topics:

- How to code BST as a special case of Binary Tree
- BST Traversals, PreOrder, InOrder and PostOrder
- Searching in BST

Instructor Note

As a pre-lab activity, read Chapter 15 from the text book "A Common-Sense Guide to Data Structures and Algorithms, Jay Wengrow, Pragmatic Bookshelf, 2020.".

1) Useful Concepts

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

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

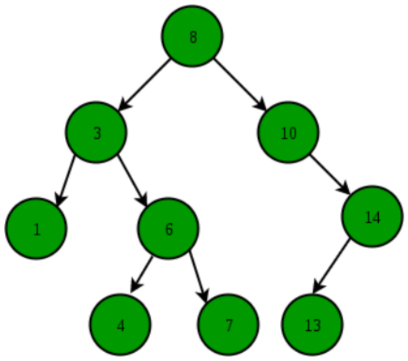


Fig 7.1: Binary Search Tree

The main operations are:
Insert in BST − place an element in the existing structure of BST.
Search in BST – check about presence of an elemnt in BST, if found returns number of comparisons done for a successful search.
Delete from BST − delete the element from BST
Display BST/ Traversal – displays all the nodes in one of the three possible traversals.

2) Solved Lab Activites

Sr.No	***Allocated Time***	***Level of Complexity***	***CLO Mapping***
Activity 1	***15 mins***	***Medium***	***CLO-4***
Activity 2	***10 mins***	***Medium***	***CLO-4***
Activity 3	***15 mins***	***Medium***	***CLO-4***
Activity 4	***05 mins***	***Medium***	***CLO-4***
Activity 5	***05 mins***	***Medium***	***CLO-4***
Activity 6	***10 mins***	***Medium***	***CLO-4***

Activity 1:

Iterative insertion of Binary Search Tree.

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

 struct bst{
     int id;
     bst *left=NULL;
     bst *right=NULL;
     };
 bst *root=NULL;
void insert(int a){

     bst *curr=new bst;
     curr->id=a;
     if (root==NULL){
            root=curr;
      }
      else{

            bst *p=root, *q;


            while(p!=NULL){
                  q=p;
                  if(curr->id>q->id){
                        p=p->right;
                   }

                   else{
                        p=p->left;
                   }
        }
        if(curr->id>q->id){
             q->right=curr;
        }
        else{
             q->left=curr;
        }
 }

}

void Inorder(struct bst *root) {
  if (root != NULL) {
    preorder(root->left);
    cout << root->id << " > ";
    preorder(root->right);
  }
}
```

Activity 2:

Write down Traversal code for the above activity, perform Pre order, Post order and In order Traversals.

```
void Inorder(struct bst *root) {
  if (root != NULL) {
    Inorder(root->left);
    cout << root->id << " > ";
    Inorder(root->right);
  }
}
void Preorder(struct bst *root) {
  if (root != NULL) {
    cout << root->id << " > ";
    preorder(root->left);
    preorder(root->right);
  }
}

void Postorder(struct bst *root) {
  if (root != NULL) {
    postorder(root->left);
    postorder(root->right);
    cout << root->id << " > ";
  }
}
```

Activity 3:

Write the recursive insertion method.

```
void insertroot(BST *curr)
{
root=curr;
}
void recursive_insert(BST *root, BST *curr)
{
    if(root==NULL)
      {
            insertroot(curr);
      }else{
          if(root->marks < curr->marks)
           {
                 if(root->right==NULL)
```

```
                    {
                    root->right=curr;
                    }
                    else{
                    recursive_insert (root->right, curr);
                          }
              }
        else
              {
                    if(root->left==NULL)
                          {
                                root->left=curr;
                          }
                    else
                          {
                                recursive_insert (root->left,curr);
                          }
        }     }
    }
```

Activity 4:

Write down the search method for BST.

```
BST * search(int key)
{
 BST *p=root;
 While(p!=NULL && p->data!=key)
 {
  If(key>p->data)
     P=p->right;
 Else
```

```
    P=p->left;
 }
 Return p;
 }
```

Activity 5:

Write down the method to find the minimum and maximum value from BST.

```
BST * min()
{
 BST *p=root, p2;
 While(p!=NULL)
 {
 P2=p;
 P=p->left;
 }
 Return p2;
 }
BST * max()
{
 BST *p=root, p2;
 While(p!=NULL)
 {
 P2=p;
 P=p->right;
 }
 Return p2;
 }
```

Activity 6:

Write down the method to count to number of nodes and find the sum of all nodes.

```
// count and sum are globally declared or may be diclared as static
void sum(BST *s)
      {
             if(s!=NULL)
            {
                    count=count+1;
                    add=add+s->data;
                    sum(s->left);
                    sum(s->right);
            }
      }
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Implement the methods for Tree Traversal with Right Branch Priority i.e. Pre Order (NRL), Post Order (RLN) and In Order (RNL).

Lab Task 2

Write down code to print and count Leaf nodes of a BST.

Lab Task 3

Introduce method to delete a Node from BST, keep in mind that there are 3 possiblities for deletion, Node without any child, Node with One child and Node with both the children.

Lab 08

AVL Tree

Objective:

By completing the AVL Tree Lab, you will be able to:

- Implement functions to rotate nodes and balance a Binary Search Tree.
- Insert nodes from a Binary Search Tree while maintaining tree balance.
- Remove nodes from a Binary Search Tree while maintaining tree balance.

Activity Outcomes:

This lab teaches you the following topics:

- Height of the Tree
- Rotate Left
- Rotate Right
- Insertion in AVL Tree
- Deletion in AVL Tree

Instructor Note

As a pre-lab activity, read Chapter 15 from the text book "A Common-Sense Guide to Data Structures and Algorithms, Jay Wengrow, Pragmatic Bookshelf, 2020.".

1) Useful Concepts

An AVL tree is a self-balancing binary search tree. In an AVL tree, the heights of the two subtrees of any node differ by at most one. Lookup, insertion, and deletion all take O(log n) time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

2) Solved Lab Activites

Sr.No	***Allocated Time***	***Level of Complexity***	***CLO Mapping***
Activity 1	***15 mins***	***Medium***	***CLO-4***
Activity 2	***05 mins***	***Medium***	***CLO-4***
Activity 3	***05 mins***	***Medium***	***CLO-4***
Activity 4	***05 mins***	***Medium***	***CLO-4***
Activity 5	***15 mins***	***Medium***	***CLO-4***
Activity 6	***15 mins***	***Medium***	***CLO-4***

Activity 1:

Height of the Tree

```
int height(node *T)
{
int lh,rh;
if(T==NULL)
return(0);
if(T->left==NULL)
lh=0;
else
lh=1+T->left->ht;
if(T->right==NULL)
rh=0;
else
rh=1+T->right->ht;
if(lh>rh)

return(lh);

return(rh);

}
```

Output

Will find the height of the tree.

Activity 2:

Rotate Right

```
node * rotateright(node *x)

{

node *y;

y=x->left;

x->left=y->right;

y->right=x;

x->ht=height(x);

y->ht=height(y);

return(y);

}
```

Output

Will apply the right rotation.

Activity 3:

Apply the left right rotation

```
node * LR(node *T)

{

T->left=rotateleft(T->left);

T=rotateright(T);
return(T);
}
```

Output

Will apply the left right rotation.

Activity 4:

Find the balance factor

```
int BF(node *T)
{
int lh,rh;
if(T==NULL)
return(0);
if(T->left==NULL)
lh=0;
else
lh=1+T->left->ht;
if(T->right==NULL)
rh=0;
else
rh=1+T->right->ht;

return(lh-rh);
}
```

Output

The method will return the balance factor.

Activity 5:

Insert the new node in AVL

```
node * insert(node *T, int x)    //T is head node
{
if(T==NULL)
{
T= new node;
T->data=x;
T->left=NULL;
T->right=NULL;
}
else
if(x > T->data) // insert in right subtree
{
T->right=insert(T->right,x);
if(BF(T)==-2)
if(x>T->right->data)
T=RR(T);
else
T=RL(T);
}
else
if(x<T->data)
{
T->left=insert(T->left,x);
if(BF(T)==2)
if(x < T->left->data)
T=LL(T);
else
T=LR(T);
}
T->ht=height(T);
return(T);
}
```

Output

The method will insert the new node in AVL Tree.

Activity 6:

Delete from AVL Tree

```
node * Delete(node *T,int x)
{
node *p;
if(T==NULL)
{
return NULL;
}
else
if(x > T->data) // insert in right subtree
{
T->right=Delete(T->right,x);
if(BF(T)==2)
if(BF(T->left)>=0)
T=LL(T);
else
T=LR(T);
}
else
if(x<T->data)
{
T->left=Delete(T->left,x);
if(BF(T)==-2) //Rebalance during windup
if(BF(T->right)<=0)
T=RR(T);
else
T=RL(T);
}
else
{
//data to be deleted is found
if(T->right!=NULL)
{ //delete its inorder succesor
p=T->right;
while(p->left!= NULL)
p=p->left;
T->data=p->data;
T->right=Delete(T->right,p->data);
if(BF(T)==2)//Rebalance during windup
if(BF(T->left)>=0)
T=LL(T);
else
T=LR(T);
}
else
return(T->left);
}
T->ht=height(T);
return(T); }
```

Output

The method will insert the new node in AVL Tree.

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Complete all types of rotations applied in AVL Tree.

Lab Task 2

Apply level order traversal of AVL Tree.

Lab 10

Heap Tree as Priority Queue

Objective:

This lab will provide you with the inside knowledge of another very important type of Queue, i.e. Priority Queues.

Activity Outcomes:

- Design Priority Queue Array based
- Design Priority Queue Dynamic
- Apply priority Queue in a real-life Example

Instructor Note

As a pre-lab activity, read Chapter 16 from the text book "A Common-Sense Guide to Data Structures and Algorithms, Jay Wengrow, Pragmatic Bookshelf, 2020.".

1) Useful Concepts

Till now you have studied that inn a Queue we go on with the first in first out system but what to do if you encounter such a system where this system partially fails, i.e. you require the same system of first out but the incoming queue is settled according to a defined system??
We have studied that there is a special type of Queue in which we can edit form both the ends, i.e. DEQUE.
For a better understanding of this topic consider a Hospital System where we first serve the patients who are brought in a critical condition. We leave all the patients with not so serious issues and the Doctor give the treatment at a priority to the one brought in critical condition. Hence, we can grasp an idea that every element comes with its own priority, and are served in accordance. This is the same idea we will implement for a Priority Queue.

2) Solved Lab Activites

Sr.No	***Allocated Time***	***Level of Complexity***	***CLO Mapping***
Activity 1	***20 mins***	***Medium***	***CLO-4***
Activity 2	***20 mins***	***Medium***	***CLO-4***

Activity 1:

Design an Array Based Priority Queue. Write the enqueue method.

```
using namespace std;


int H[50];

int size = -1;


// Function to return the index of the parent node of a given node

int parent(int i)

{


    return (i - 1) / 2;

}

// Function to shift up the node in order to maintain the heap property

void shiftUp(int i)
```

```
{
    while (i > 0 && H[parent(i)] < H[i]) {

        // Swap parent and current node
        swap(H[parent(i)], H[i]);

        // Update i to parent of i
        i = parent(i);
    }
}
void insert(int p)
{
    size = size + 1;
    H[size] = p;

    // Shift Up to maintain heap property
    shiftUp(size);
}
```

Output

The method will insert then new node in heap tree.

Activity 2:

Dequeue the element from queue

```
// Function to extract the element with maximum priority
int extractMax()
{
    int result = H[0];

    // Replace the value at the root
    // with the last leaf
    H[0] = H[size];
    size = size - 1;

    // Shift down the replaced element
    // to maintain the heap property
    shiftDown(0);
    return result;
}
// Function to shift down the node in
// order to maintain the heap property
void shiftDown(int i)
{
    int maxIndex = i;

    // Left Child
    int l = leftChild(i);

    if (l <= size && H[l] > H[maxIndex]) {
        maxIndex = l;
    }

    // Right Child
    int r = rightChild(i);

    if (r <= size && H[r] > H[maxIndex]) {
        maxIndex = r;
    }

    // If i not same as maxIndex
    if (i != maxIndex) {
        swap(H[i], H[maxIndex]);
        shiftDown(maxIndex);
    }
}

// Function to return the index of the left child of the given node
int leftChild(int i)
{

    return ((2 * i) + 1);
}

// Function to return the index of the right child of the given node
int rightChild(int i)
{

    return ((2 * i) + 2);
}
```

Output

Will method will delete the element of highest priority.

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Design a Hospital Registration system for patients using Priority Queue.

a. Assume a struct that you think is appropriate for such a system.

b. Take the priority factor, the condition of patient.

c. Design it using the above implemented system of Priority Queue.

Lab 11

Graph representation by Matrix, Graph traversals and Dijkastra Algorithm

Objective:

This lab will introduce you the concept and implementation of Graph.

Activity Outcomes:

- Get to know about Graph Data Structure.
- Use the operations on Graph Data Structure

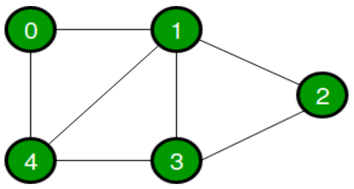
Instructor Note

As a pre-lab activity, read Chapter 18 from the text book "A Common-Sense Guide to Data Structures and Algorithms, Jay Wengrow, Pragmatic Bookshelf, 2020.".

1) Useful Concepts

Graph is a data structure that consists of 2 finite set as follows:

1. A finite set of vertices that are called nodes.
2. A finite set of order pair (u, v) called an edge. The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v.

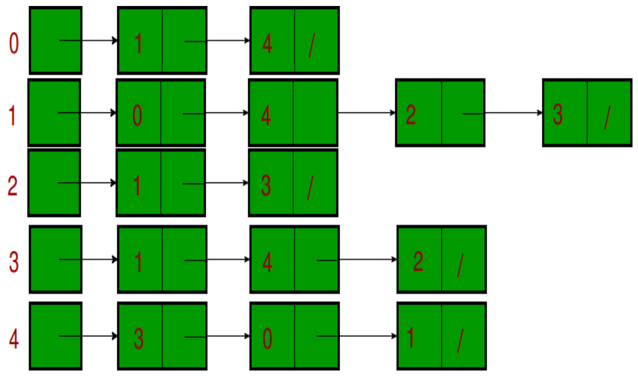


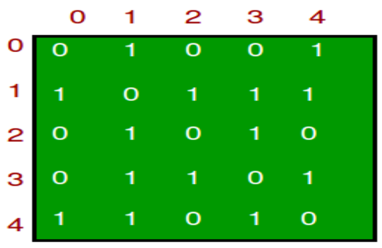
There are further two types of Graph:

1. Directed
2. Undirected

A Directed graph is such a in which the order pair matters as (u, v) pair
Hence, in a directed graph we will also have a direction on each edge which will be from (u, v) the first node (u) to second node (v). This also clears that we can go from u to v from this edge but not in the vice versa matter.
An Undirected graph is such a in which we have no order restriction in the edge pair. Thus, we can go and come back from the same edge, i.e. (u, v) is equal to (v, u).

So how to represent a Graph?
We can do it in two ways, i.e. using an Adjacency Matrix (2D array) and also an Adjacency List (Link List of Link List).





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

Adjacency List ⇧ Adjacency Matrix ⇧

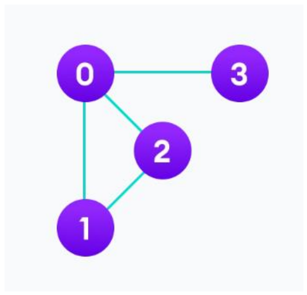
2) Solved Lab Activites

Sr.No	***Allocated Time***	***Level of Complexity***	***CLO Mapping***
Activity 1	***20 mins***	***Medium***	***CLO-4***
Activity 2	***20 mins***	***High***	***CLO-4***
Activity 3	***20 mins***	***High***	***CLO-4***

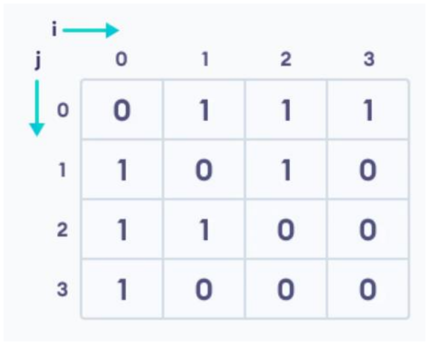
Activity 1:

Implements the Graph data structure by Adjacency Matrix.

Consider the following Graph:



The matrix of the above Graph will be:



j \ i →	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

```
#include <stdio.h>
#define V 4
// Initialize the matrix to zero
void init(int arr[][V]) {
  int i, j;
  for (i = 0; i < V; i++)
    for (j = 0; j < V; j++)
      arr[i][j] = 0;
}

// Add edges
void addEdge(int arr[][V], int i, int j) {
  arr[i][j] = 1;
  arr[j][i] = 1;
}


// Print the matrix
void printAdjMatrix(int arr[][V]) {
  int i, j;


  for (i = 0; i < V; i++) {
    cout<<I;
    for (j = 0; j < V; j++) {
      cout<< arr[i][j]);
    }
    Cout<<"\n");
  }
}


int main() {
  int adjMatrix[V][V];


  init(adjMatrix);
  addEdge(adjMatrix, 0, 1);
  addEdge(adjMatrix, 0, 2);
  addEdge(adjMatrix, 1, 2);
  addEdge(adjMatrix, 2, 0);
  addEdge(adjMatrix, 2, 3);


  printAdjMatrix(adjMatrix);
  return 0;
}
```

Activity 2:

Implement the Graph by Adjacency List.

```
#include<iostream>

using namespace std;

struct edge{
     int data;
     edge *next;
};

struct node{
     int data;
     edge *head;
     edge *tail;
     node *next;
}*head=NULL, *tail=NULL;

void insert_vertex(int data){
     node *temp=new node;
     temp->data=data;
     temp->head=NULL;
     temp->tail=NULL;

     if(head==NULL){
           temp->next=NULL;
           head=tail=temp;
     }
     else{
           tail->next=temp;
           tail=temp;
```

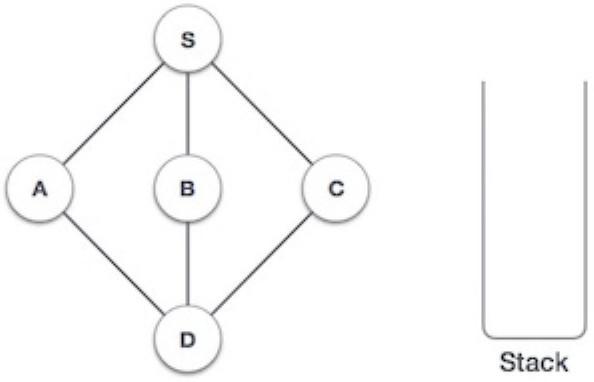
```
            tail->next=NULL;
        }
}

bool check_uname(int data){
        node *temp=head;
        while((temp->data!=data) && (temp!=tail)){
            temp=temp->next;
        }

        if(temp->data!=data){
            cout<<"Second Vertix Not Found "<<data<<endl;
            return false;
        }

        return true;
}

void add_edge(int vname, int uname){
        node *temp=head;
        while(temp->data!=vname && temp!=NULL){
            temp=temp->next;
        }

        if(temp==NULL){
            cout<<"Source Vertix Not Found"<<endl;
            return;
        }

        if(temp!=NULL && check_uname(uname)){
            edge *etemp=new edge;
            etemp->data=uname;
            if(temp->head==NULL){
                  etemp->next=NULL;
                  temp->head=temp->tail=etemp;
            }
            else{
                  temp->tail->next=etemp;
                  temp->tail=etemp;
                  etemp->next=NULL;
            }
      }
}

void display(){
      node *temp=head;
      edge *etem;

      while(temp!=NULL){
            etem=temp->head;
            cout<<temp->data<<" -> ";
            while (etem!=NULL){
                  cout<<etem->data<<" ";
                  etem=etem->next;
            }
            temp=temp->next;
            cout<<endl;
      }
}

void delete_edge(int vname, int uname){
      node *temp=head;
      while (temp!=NULL){
            if(temp->data==vname){
                break;
            }
            temp=temp->next;
        }

        if(temp==NULL){
            cout<<"Vertix Not Found"<<endl;
            return;
        }
        else{
            edge *emp=temp->head;

            if(emp->data==uname){
                if(temp->head==temp->tail){
                    temp->head=temp->tail=NULL;
                    return;
                }
                temp->head=temp->head->next;
                return;
            }

            while(emp!=NULL){
                if(emp->next->data==uname){
                    emp->next=emp->next->next;
                    return;
                }
                emp=emp->next;
            }
            if(emp==NULL){
                cout<<"Edge Not Found"<<endl;
            }
        }
```

```
}


int main(){
      insert_vertex(1);
      insert_vertex(2);
      insert_vertex(3);
      insert_vertex(4);
      insert_vertex(5);

      add_edge(1,2);
      add_edge(2,15);//exception handled if node if not found
      add_edge(1,4);
      add_edge(4,1);
      display();

      delete_edge(1,2);
      display();
}
```

Activity 3:

Implement the DFS Algorithm.

For our reference purpose, we shall follow our example and take this as our graph model



```
#include <stdlib.h>
```

```
#include <iostream>
using namespace std;
#define MAX 5

struct Vertex {
   char label;
   bool visited;
};
//stack variables
int stack[MAX];
int top = -1;
//graph variables
//array of vertices
struct Vertex* lstVertices[MAX];

//adjacency matrix
int adjMatrix[MAX][MAX];
//vertex count
int vertexCount = 0;
//stack functions
void push(int item) {
   stack[++top] = item;
}

int pop() {
   return stack[top--];
}

int peek() {
   return stack[top];
}
```

```
bool isStackEmpty() {
   return top == -1;
}

//graph functions

//add vertex to the vertex list
void addVertex(char label) {
   struct Vertex* vertex = (struct Vertex*) malloc(sizeof(struct Vertex));
   vertex->label = label;
   vertex->visited = false;
   lstVertices[vertexCount++] = vertex;
}

//add edge to edge array
void addEdge(int start,int end) {
   adjMatrix[start][end] = 1;
   adjMatrix[end][start] = 1;
}

//display the vertex
void displayVertex(int vertexIndex) {
   cout<<lstVertices[vertexIndex]->label;
}

//get the adjacent unvisited vertex
int getAdjUnvisitedVertex(int vertexIndex) {
   int i;

   for(i = 0; i < vertexCount; i++) {
      if(adjMatrix[vertexIndex][i] == 1 && lstVertices[i]->visited == false)
{
```

```
         return i;
      }
   }

   return -1;
}

void depthFirstSearch() {
   int i;

   //mark first node as visited
   lstVertices[0]->visited = true;

   //display the vertex
   displayVertex(0);

   //push vertex index in stack
   push(0);

   while(!isStackEmpty()) {
      //get the unvisited vertex of vertex which is at top of the stack
      int unvisitedVertex = getAdjUnvisitedVertex(peek());

      //no adjacent vertex found
      if(unvisitedVertex == -1) {
         pop();
      } else {
         lstVertices[unvisitedVertex]->visited = true;
         displayVertex(unvisitedVertex);
         push(unvisitedVertex);
      }
   }
   //stack is empty, search is complete, reset the visited flag
   for(i = 0;i < vertexCount;i++) {
      lstVertices[i]->visited = false;
   }
}

int main() {
   int i, j;

   for(i = 0; i < MAX; i++)    // set adjacency
     {
          for(j = 0; j < MAX; j++) // matrix to 0
         adjMatrix[i][j] = 0;
   }

   addVertex('S');   // 0
   addVertex('A');   // 1
   addVertex('B');   // 2
   addVertex('C');   // 3
   addVertex('D');   // 4

   addEdge(0, 1);    // S - A
   addEdge(0, 2);    // S - B
   addEdge(0, 3);    // S - C
   addEdge(1, 4);    // A - D
   addEdge(2, 4);    // B - D
   addEdge(3, 4);    // C - D

   cout<<"\n Depth First Search: ";
   depthFirstSearch();
```

```
    return 0;
}
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Implement the recursive method of DFS.

Lab Task 2

Implement the BFS.

Lab Task 3

Implement the Graph part of your project.

Lab 12

Minimum Spanning Tree

Objective:

This lab will introduce you the concept of minimum spanning trees.

Activity Outcomes:

Get to know about minimum spanning trees.

Instructor Note

As a pre-lab activity, read Chapter 9 from the text book "A Common-Sense Guide to Data Structures and Algorithms, Jay Wengrow, Pragmatic Bookshelf, 2020.".

1) Useful Concepts

It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.

We start from one vertex and keep adding edges with the lowest weight until we reach our goal.

The steps for implementing Prim's algorithm are as follows:

1. Initialize the minimum spanning tree with a vertex chosen at random.
2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
3. Keep repeating step 2 until we get a minimum spanning tree

2) Solved Lab Activites

Sr.No	***Allocated Time***	***Level of Complexity***	***CLO Mapping***
Activity 1	***1 hour***	***High***	***CLO-5***

Activity 1:

Implements the PRIMS algorithm to construct the MST.

```
#include <stdio.h>
#include<iostream>
using namespace std;
// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with minimum key value
int minKey(int key[], bool mstSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;
    return min_index;
}


// A utility function to print the constructed MST stored in parent[]
int printMST(int parent[], int graph[V][V])
{
    cout<<"\n Edge \tWeight: \n";
    for (int i = 1; i < V; i++)
        cout<<"\n"<< parent[i]<<" -> "<< i<<"   "<<graph[i][parent[i]];
}


// Function to construct and print MST for a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
    // Array to store constructed MST
    int parent[V];
    // Key values used to pick minimum weight edge in cut
    int key[V];
    // To represent set of vertices included in MST
    bool mstSet[V];

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.
    // Make key 0 so that this vertex is picked as first vertex.
    key[0] = 0;
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum key vertex from the
        // set of vertices not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of
        // the adjacent vertices of the picked vertex.
        // Consider only those vertices which are not
        // yet included in MST
        for (int v = 0; v < V; v++)

            // graph[u][v] is non zero only for adjacent vertices of m
            // mstSet[v] is false for vertices not yet included in MST
            // Update the key only if graph[u][v] is smaller than key[v]
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    // print the constructed MST
    printMST(parent, graph);
}
 // driver program to test above function
int main()
{
    int graph[V][V] = { { 0, 2, 0, 6, 0 },
                        { 2, 0, 3, 8, 5 },
                        { 0, 3, 0, 0, 7 },
                        { 6, 8, 0, 0, 9 },
                        { 0, 5, 7, 9, 0 } };

    // Print the solution
    primMST(graph);

    return 0;
}
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Implement the MST part of your project.

Lab 13

Searching and Hashing Algorithms

Objective:

In this lab we will be learning about various methods of searching in a given set of data and also discuss about time consumption of these algorithms. Then we will be working with retrieval of data in a linear time, i.e. O (1), i.e. we will be studying Hashing.
By completing this Lab, you will be able to:

- Apply the searching techniques.
- Apply the Hashing techniques.

Activity Outcomes:

This lab teaches you the following topics:

- Linear Search
- Binary Search
- Hashing

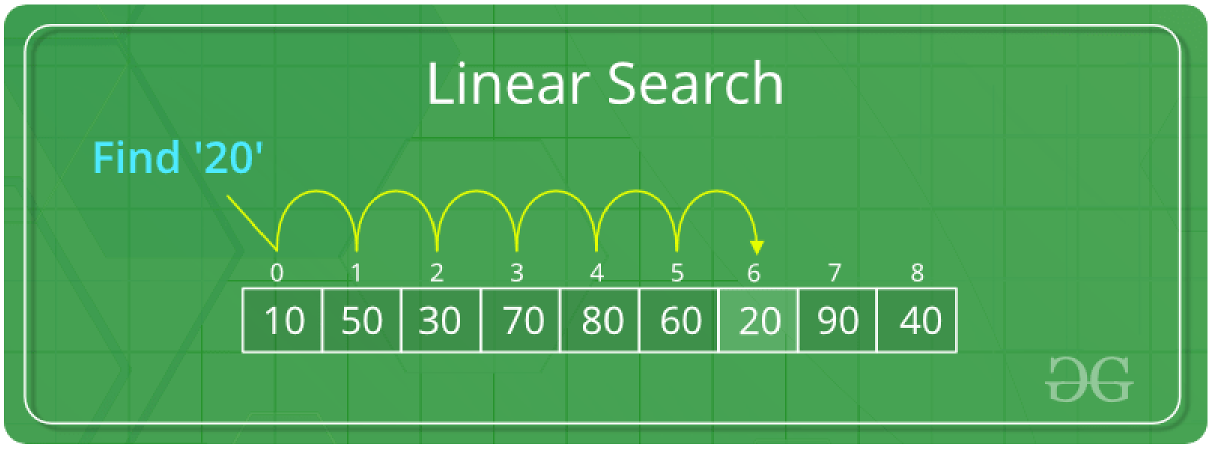
Instructor Note

As a pre-lab activity, read Chapter 2 &8 from the text book “A Common-Sense Guide to Data Structures and Algorithms, Jay Wengrow, Pragmatic Bookshelf, 2020.”.

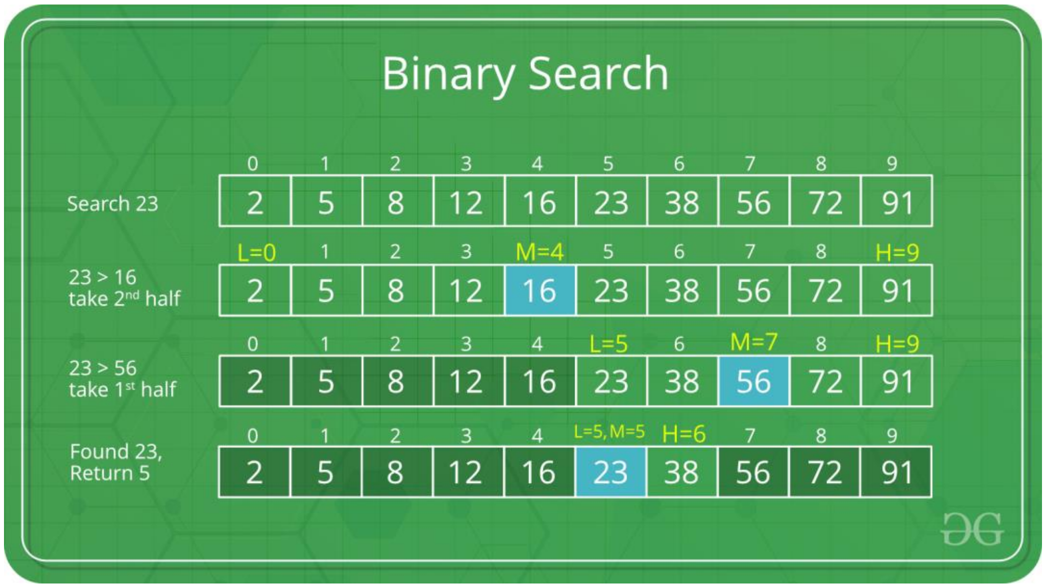
1) Useful Concepts

Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. Based on the type of search operation, these algorithms are generally classified into two categories:

1. Sequential Search: In this, the list or array is traversed sequentially and every element is checked. For example: Linear Search.
2. Interval Search: These algorithms are specifically designed for searching in sorted data-structures. These type of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half. For Example: Binary Search.



Linear Search: ⇧



Binary Search: ⇧

2) Solved Lab Activites

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
Activity 1	***05 mins***	***Low***	***CLO-4***
Activity 2	***05 mins***	***Low***	***CLO-4***
Activity 3	***10 mins***	***Low***	***CLO-4***
Activity 4	***10 mins***	***Low***	***CLO-4***
Activity 5	***15 mins***	***High***	***CLO-4***
Activity 6	***15 mins***	***High***	***CLO-4***

Activity 1:

Design an iterative Program for Linear Search.

```
int linear _search(int a[], int item,int size){//normal version
     for(int i=0; i<size; i++){
           if(item == a[i]){
                 return i;
           }
     }
     return -1;
//returning -1 if the key is not found as index can never be negative
}
```

Output

Will return the index where the value found else will return -1.

Activity 2:

Write recursive method to search the value by applying linear search algorithm

```
int rec_linear_search(int a[], int item , int size){//recursive approach
     if(size==1){
           return -1;    //if key is not found returning -1
     }
     else if (item == a[size-1]){
           return size-1;
     }
     else{
           return rec_l_search(a,item,size-1);
     }
}
```

Output

Will return the index where the value found else will return -1.

Activity 3:

Design a Program for Binary Search. (Note: The Input Array for this approach must be sorted)

```
int binary_search(int array[],int s,int e,int item){//normal approach
     int mid=(e+s)/2;
     if(item==array[mid]){
           return mid;
     }
     else if(item == array[s]){
            return s;
     }
     else if(item == array[e]){
           return e;
     }
```

```
      else{
             while(s<=e){
                 if(item==array[mid]){
                         return mid;
                         break;
                   }
                   else if (item == array[s]){
                         return s;
                   }
                   else if(item == array[e]){
                         return e;
                   }
                   else if(item<array[mid]){
                         mid=(s+e)/2;
                         e=mid-1;
                   }
                   else if(item>array[mid]){
                         mid=(s+e)/2;
                         s=mid+1;
                   }
                   else{
                         continue;
                   }
             }
      }
      return -1;//returning -1 if key not found
 }
```

Output

Will return the index where the value found else will return -1.

Activity 4:

Write the recursive method for Binary Search.

```
int recursive_ binary_search (int array[],int s,int e,int item){
 int mid=(e+s)/2;
     if(s>e){
           return -1;
     }
     if(item == array[mid]){
           return mid;
     }
     else if(item == array[s]){
           return s;
     }
     else if(item == array[e]){
           return e;
     }
     else if (item < array[mid]){
           return rec_merge(array , s, mid-1, item);
     }
     else if (item > array[mid]){
           return rec_merge(array , mid+1, e, item);
     }
     else {
           return -1;
     }
}
```

Output

Will return the index where the value found else will return -1.

Activity 5:

Implement the linear probing method.

```
#include <stdio.h>

#include<stdlib.h>

#include<iostream>

using namespace std;

#define TABLE_SIZE 10


int h[TABLE_SIZE]={NULL};


void insert()

{


 int key,index,i,flag=0,hkey;

 cout<<"\nenter a value to insert into hash table\n";

cin>>key;

 hkey=key%TABLE_SIZE;

 for(i=0;i<TABLE_SIZE;i++)

    {


     index=(hkey+i)%TABLE_SIZE;


     if(h[index] == NULL)

     {

        h[index]=key;

         break;

     }


    }


    if(i == TABLE_SIZE)
```

```
     cout<<"\n element cannot be inserted\n";
}
void search()
{

 int key,index,i,flag=0,hkey;
 cout<<"\nenter search element\n";
 cin>>key;
 hkey=key%TABLE_SIZE;
 for(i=0;i<TABLE_SIZE; i++)
 {
    index=(hkey+i)%TABLE_SIZE;
    if(h[index]==key)
    {
      cout<<"value is found at index"<<index;
      break;
    }
  }
  if(i == TABLE_SIZE)
    cout<<"\n value is not found\n";
}
void display()
{

  int i;

  cout<<"\nelements in the hash table are \n";

  for(i=0;i< TABLE_SIZE; i++)

  cout<<"\nat index "<<i<<" \t value =  "<<h[i];
```

```
}
main()
{
    int opt,i;
    while(1)
    {
        cout<<"\n Press 1. Insert\t 2. Display \t3. Search \t4.Exit \n";
        cin>>opt;
        switch(opt)
        {
            case 1:
                insert();
                break;
            case 2:
                display();
                break;
            case 3:
                search();
                break;
            case 4:exit(0);
        }
    }
}
```

Output

The method will insert, display and search the value from Hash Table.

Activity 6:

Insertion by chaining method.

```
#include<stdio.h>

#include<stdlib.h>

#include<iostream>

using namespace std;

#define size 7

struct node

{

    int data;

    struct node *next;

};


struct node *chain[size];


void init()

{

    int i;

    for(i = 0; i < size; i++)

        chain[i] = NULL;

}


void insert(int value)

{

    //create a newnode with value

       node *newNode = new node;

     newNode->data = value;

    newNode->next = NULL;


    //calculate hash key

    int key = value % size;
    //check if chain[key] is empty
    if(chain[key] == NULL)
        chain[key] = newNode;
    //collision
    else
    {
        //add the node at the end of chain[key].
        struct node *temp = chain[key];
        while(temp->next)
        {
            temp = temp->next;
        }

        temp->next = newNode;
    }
}

void print()
{
    int i;

    for(i = 0; i < size; i++)
    {
        struct node *temp = chain[i];
        cout<<"chain[i]-->",i);

              while(temp)
        {
            cout<<"\n "<<temp->data;
                  temp = temp->next;
        }
```

```
          Cout<<"NULL\n";
      }
  }
  int main()
  {
      //init array of list to NULL
      init();

      insert(7);
      insert(0);
      insert(3);
      insert(10);
      insert(4);
      insert(5);
      print();
     return 0;
  }
```

Output

Insertion of value by chaining method.

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Write the deletion method for chaining.

Lab Task 2

Apply searching in Project.

Lab 14

Sorting Algorithms

Objective:

In this lab we will be learning about various methods of sorting a given set of data and also discuss about time consumption of these algorithms.

Activity Outcomes:

This lab teaches you the following topics:

1. Get to study the various type of sorting.
2. Study about time required by each one of them.

Instructor Note

As a pre-lab activity, read Chapter 4-6 from the text book "A Common-Sense Guide to Data Structures and Algorithms, Jay Wengrow, Pragmatic Bookshelf, 2020.".

1) Useful Concepts

Sorting is nothing but arranging the data in ascending or descending order. The term sorting came into picture, as humans realized the importance of searching quickly.

There are so many things in our real life that we need to search for, like a particular record in database, roll numbers in merit list, a particular telephone number in telephone directory, a particular page in a book etc. All this would have been a mess if the data was kept unordered and unsorted, but fortunately the concept of sorting came into existence, making it easier for everyone to arrange data in an order, hence making it easier to search.

Sorting arranges data in a sequence which makes searching easier.

Since the beginning of the programming age, computer scientists have been working on solving the problem of sorting by coming up with various different algorithms to sort data.

The two main criteria to judge which algorithm is better than the other have been:

- Time taken to sort the given data.
- Memory Space required to do so.

There are many different techniques available for sorting, differentiated by their efficiency and space requirements. We will be studying the following:

- Bubble Sort
- Insertion Sort
- Selection Sort
- Merge Sort

Sr.No	***Allocated Time***	***Level of Complexity***	***CLO Mapping***
Activity 1	***10 mins***	***Low***	***CLO-3***
Activity 2	***15 mins***	***Low***	***CLO-3***
Activity 3	***15 mins***	***Low***	***CLO-3***

Activity 4	*20 mins*	*High*	*CLO-3*

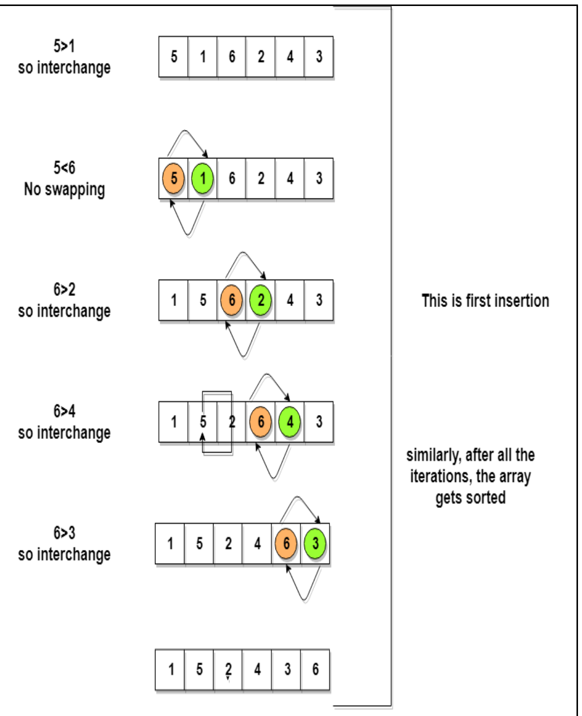
2) Solved Lab Activites

Activity 1:

Implementing the bubble sort Algorithm:

1. ***Starting with the first element (index = 0/node = 1), compare the current element with the next element of the array.***
2. ***If the current element is greater than the next element of the array, swap them.***
3. ***If the current element is less than the next element, move to the next element. Repeat Step 1.***

```
void bubbleSort(int arr[], int n) {

    int i, j, temp;

    for(i = 0; i < n; i++){

        for(j = 0; j < n-i-1; j++){

            if( arr[j] > arr[j+1]){

            // swap the elements

                temp = arr[j];

                arr[j] = arr[j+1];

                arr[j+1] = temp;

            }

        }

    }

}
```



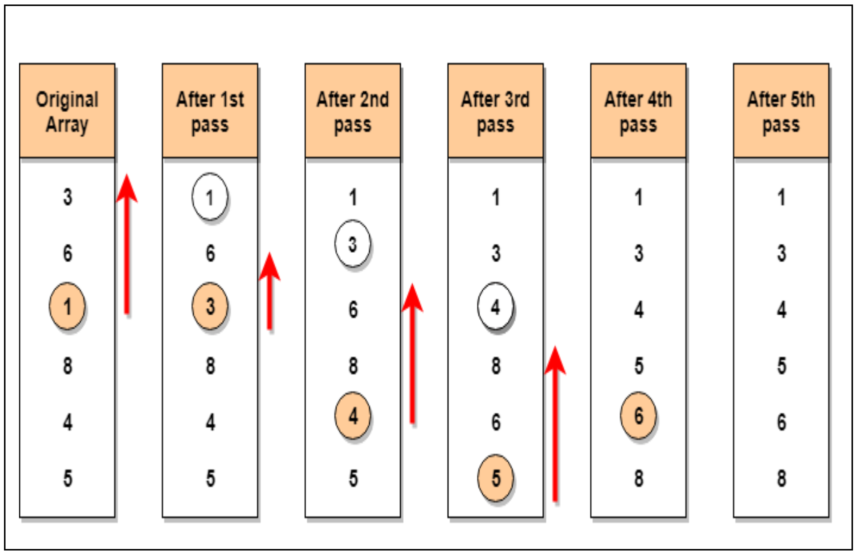
Output

Will Sort the array by bubble sort.

Activity 2:

Implementing the Selection Sort Algorithm:

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.***



```
int findMin_Index(int a[], int size, int startIndex){
     int i=startIndex;
     int min = i;
     for(i; i<size; i++){
            if(a[i] < a[min]){
                   min = i;
            }
     }
     return min;
}

void swap_Elems(int a[], int i, int j){
     int temp = a[i];
     a[i] = a[j];
     a[j] = temp;
}

void select_andSort(int a[], int size){
     int minIndex;
     for(int i=0; i<size; i++){
            minIndex = findMin_Index(a, size ,i);

            swap_Elems(a, i, minIndex);

      }

}
```

Output

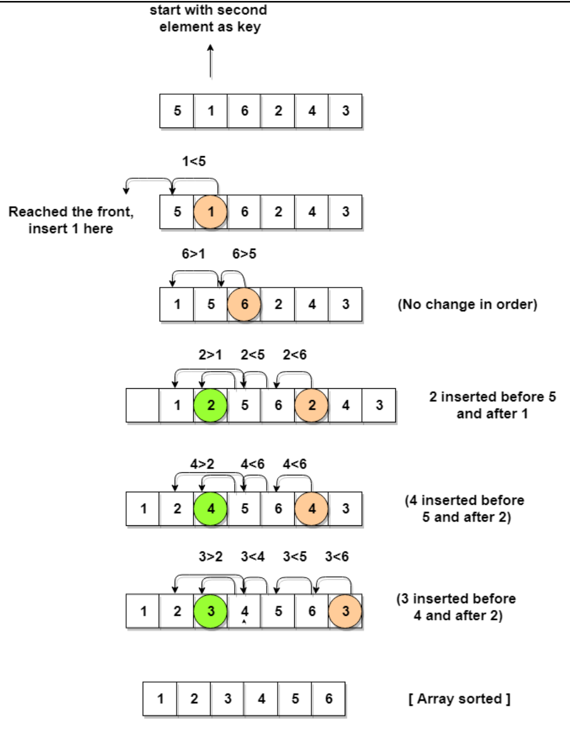
Will Sort the array by selection sort.

Activity 3:

Implementing the Insertion Sort Algorithm:

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.***

```
void    insertionSort(int    arr[],    int
length){

      int i, j, key;

      for (i = 1; i < length; i++){

            j = i;

 while (j > 0 && arr[j - 1] > arr[j]){

                  key = arr[j];

                  arr[j] = arr[j - 1];

                  arr[j - 1] = key;

                  j--;

            }

      }

}
```



start with second element as key
1<5
Reached the front, insert 1 here
6>1 6>5
(No change in order)
2>1 2<5 2<6
2 inserted before 5 and after 1
4>2 4<6 4<6
(4 inserted before 5 and after 2)
3>2 3<4 3<5 3<6
(3 inserted before 4 and after 2)
[Array sorted]

Output

Will Sort the array by insertion sort.

Activity 4:

Implementing the Merge Sort Algorithm:

1. ***Divide the problem into multiple small problems.***
2. ***Conquer the sub-problems by solving them. The idea is to break down the problem into atomic sub-problems, where they are actually solved.***
3. ***Combine the solutions of the sub-problems to find the solution of the actual problem***

```
Void mergesort(int list[], int first, int last) {
if( first < last )
      {
         mid = (first + last)/2;
        // Sort the 1st half of the list
        mergesort(list, first, mid);
        // Sort the 2nd half of the list
        mergesort(list, mid+1, last);
     end if
}}

Void merge(int list[], int first, int mid, int last) {
// Initialize the first and last indices of our subarrays
     firstA = first
     lastA = mid
     firstB = mid+1
     lastB = last

     index = firstA   // Index into our temp array

while (firstA <= lastA){
```

```
            tempArray[index] = list[firstA]
            firstA = firstA + 1
            index = index + 1
      }
while ( firstB <= lastB ){
            tempArray[index] = list[firstB]
            firstB = firstB + 1
            index = index + 1
      }
// Finally, we copy our temp array back into our original array
      index = first
      while (index <= last){
            list[index] = tempArray[index]
            index = index + 1
      }
}
```

Output

Will Sort the array by Merge sort.

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficulty and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Implement a recursive Bubble sort.

Lab Task 2

Try to improve the give algorithms for better time.

Lab Task 3

Implement the bubble sort for Linked list.

Lab Task 4

Implement the Selection sort for Linked list.

Lab Task 5

Implement the Insertion sort for Linked list.