MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
A Constituent Unit of MAHE, Manipal

DEPARTMENT OF INFORMATION & COMMUNICATION TECHNOLOGY

III SEMESTER B.TECH. (IT/CCE)

DATA STRUCTURES – ICT 2162

LAB MANUAL

CONTENTS

Course Objectives

- Learn to implement some useful data structures
- To strengthen the ability to identify and apply the suitable data structure for the given real world problem
- Learn to implement sorting and searching techniques

Course Outcomes

At the end of this course, students will be able to

- Identify appropriate data structure
- Interpret the working of searching and sorting techniques
- Demonstrate the working of linear and non-linear data structure
- Make use of preliminary structures to implement various applications

Evaluation plan

Split up of 60 marks for Regular Lab Evaluation
Total of 3 regular evaluations which will be carried out. Each evaluation is for 20 marks, which will have the following split up: Record: 6 Marks Viva = 12 Marks Execution Check: 2 marks Total = 20 Marks
End Semester Lab evaluation: 40 marks (Duration 2 hrs)
Program Write up: 15 Marks Program Execution: 25 Marks Total: 15+25 =40 Marks

INSTRUCTIONS TO THE STUDENTS

Pre- Lab Session Instructions

1. Students should carry the Lab Manual Book and the required stationary to every lab session
2. Be on time and follow the institution dress code
3. Must Sign in the log register provided
4. Make sure to occupy the allotted seat and answer the attendance
5. Adhere to the rules and maintain the decorum

In- Lab Session Instructions

- Follow the instructions on the allotted exercises
- Show the program and results to the instructors on completion of experiments
- On receiving approval from the instructor, copy the program and results in the Lab record
- Prescribed textbooks and class notes can be kept ready for reference if required

General Instructions for the exercises in Lab

- Implement the given exercise individually and not in a group.
- The programs should meet the following criteria:
    - Programs should be interactive with appropriate prompt messages, error messages if any, and descriptive messages for outputs.
    - Programs should perform input validation (Data type, range error, etc.) and give appropriate error messages and suggest corrective actions.
    - Comments should be used to give the statement of the problem and every function should indicate the purpose of the function, inputs and outputs.
    - Statements within the program should be properly indented.
    - Use meaningful names for variables and functions.
    - Make use of constants and type definitions wherever needed.
- Plagiarism (copying from others) is strictly prohibited and would invite severe penalty in evaluation.
- The exercises for each week are divided under three sets:
    - Solved exercise
    - Lab exercises - to be completed during lab hours

  - Additional Exercises - to be completed outside the lab or in the lab to enhance the skill

- In case a student misses a lab class, he/ she must ensure that the experiment is completed during the repetition class with the permission of the faculty concerned but credit will be given only to one day's experiment(s).
- Students missing out lab on genuine reasons like conference, sport or activities assigned by the department or institute will have to take **prior permission** from the HOD to attend **additional lab** (with other batch) and complete it **before** the student goes on leave. The student could be awarded marks for the write up for that day provided he submits it during the **immediate** next lab.
- Students who fall sick should get permission from the HOD for evaluating the lab records. However, the attendance will not be given for that lab.
- Students will be evaluated only by the faculty with whom they are registered even though they carry out additional experiment in other batch.
- Presence of the student during the lab end semester exams is mandatory even if the student assumes he has scored enough to pass the examination
- Minimum attendance of 75% is mandatory to write the final exam.
- If the student loses his book, he/she will have to rewrite all the lab details in the lab record.
- Questions for lab tests and examination are not necessarily limited to the questions in the manual, but may involve some variations and/or combinations of the questions.
- A sample note preparation is given as a model for observation.

THE STUDENTS SHOULD NOT

- Bring mobile phones or any other electronic gadgets to the lab.
- Go out of the lab without permission.

SAMPLE LAB OBSERVATION NOTE PREPARATION USING

Introduction

This lab course provides an introduction to computer programming using the C++ language. In this lab, the student will be able to write, see and debug their first program.

Running a sample C++ program

Let's look at C++ program implementation in steps by writing, storing, compiling and executing a sample program

- Create a directory with section followed by roll number (to be unique); e.g. A21.
- As per the instructions given by the lab teacher, create *InchToCm.cpp* program.
    - Open a new notepad file and type the given program
- Save the file with name and extension as "*InchToCm.cpp*" into the respective directory created.

Sample Program (*InchToCm.cpp*)**:**

```
// InchToCm.cpp
// this program inputs a real number inches and outputs its centimeter equivalent
// (also a real number)
#include <iostream.h>
#include<conio.h>
void main() // int main()
{
        float centimeters, inches;
        cout<< "This program converts inches to centimeters" <<endl;
        cout<< "Enter a number ";
        cin>> inches;
        centimeters = inches * 2.54;
        cout<< inches << " inches is equivalent to " << centimeters
                <<" centimeters" <<endl;
```

```
getch(); // return 0;
} // end main
```

- Run the program as per the instructions given by the lab teacher.
    - Compile the saved program and run it either by using keyboard short cuts or through the menu.

PROGRAM STRUCTURE AND PARTS

Comments

The first line of the file is:

// InchToCm

This line is a comment. Let's add a comment above the name of the program that contains the student's name.

- Edit the file. **Add the student's name on the top line** so that the first two lines of the file now look like:

// student's name

// InchToCm

Comments informs the people what the program/line is intended to do. The compiler ignores these lines.

Preprocessor Directives

After the initial comments, the student should be able to see the following line:

#include <iostream.h>

This is called a preprocessor directive. It tells the compiler to do something. Preprocessor directives always start with a # sign. In this case, the preprocessor directive includes the information in the file in stream as part of the program. Most of the programs will almost always have at least one include file. These header files are stored in a library that shall be learnt more in the subsequent labs.

The function main()

The next non-blank line *void* ***main()***specifies the name of a function as **main**. There must be exactly one function named *main* in each C++ program and this is where program execution starts. The *void* before *main()* indicates the function is not returning any value and also to indicate empty argument list to the function. Essentially functions are units of C++ code that do a particular task. Large programs will have many functions just as large organizations have many functions. Small programs, like smaller organizations, have fewer functions. The parentheses following the words *void main* contains a list of arguments to the function. In the present case, there is no *argument*. Arguments to functions indicate to the function what objects are provided to the function to perform any task. The curly braces (**{**) on the next line and on the last line (**}**)of the program determine the beginning and ending of the function.

Variable Declarations

The line after the opening curly brace, *float centimeters, inches;* is called a variable declaration. This line tells the compiler to reserve two places in memory with adequate size for a real number (the *float* keyword indicates the variable as a real number). The memory locations will have the names *inches* and *centimeters* associated with them. The programs often have many different variables of many different types.

EXECUTABLE STATEMENTS

Output and Input

The statements following the variable declaration up to the closing curly brace are executable statements. The executable statements are statements that will be executed when the program run. *cout*, the output stream operator (<<) tells the compiler to generate instructions that will display information on the screen when the program run, and *cin*, the input stream operator (>>) reads information from the keyboard when the program run.

Assignment Statements

The statement *centimeters = inches * 2.54;*is an assignment statement. It calculates what is on the right hand side of the equation (in this case *inches * 2.54*) and stores it in the memory location that has the name specified on the left hand side of the equation (in this case, *centimeters*). So *centimeters = inches * 2.54* takes whatever was read into the memory location *inches*, multiplies it by 2.54, and stores the result in *centimeters*. The next statement outputs the result of the calculation.

Return Statement

The last statement of this program, *return 0;* returns the program control back to the operating system. The value 0 indicates that the program ended normally. The last line of every main function written should be **return 0**; for **int main()**; this is indicated alternatively to **void main()** which may have a simple **return** statement

Syntax

Syntax is the way that a language must be phrased in order for it to be understandable. The general form of a C++ program is given below:

```
// program name
// other comments like what program does and student's name
# include <appropriate files>
void main()
{
Variable declarations;
Executable statements:
} // end main
```

LAB NO: 1 **Date:**

STRINGS AND CLASS CONCEPTS

Objectives:

In this lab students should be able to:

- Understand the basics of strings.
- Write and execute the programs using class concept.

Introduction to Strings and Classes:

Strings:

- A string is an array of characters. Any group of characters (except double quote sign) defined between double quotations is a constant string.
- Character strings are often used to build meaningful and readable programs.

The common operations performed on strings are

- Reading and writing strings
- Combining strings together
- Copying one string to another
- Comparing strings with one another
- Extracting a portion of a string

Declaration:

Syntax: *char* string_name[size];

- The size determines the number of characters in the string_name.

Classes:

A class is a way to bind the data and its associated functions together. It allows the data and functions to be hidden, if necessary, from external use. When defining a class we are creating a new Abstract Data Type (ADT) that can be treated like any other built-in data type. Generally, a class definition has two parts:

1. Class declaration
2. Class function definitions

General form of a class declaration is:

```
class class_name
{
  private:
      variable_declarations;
      function_declarations;
  public:
     variable_declarations;
     function_declarations;
};
```

The class members that are defined as private can be accessed only from within the class. On the other hand, public members can be accessed from outside the class also. The variables declared inside the class are called Data members and functions that are defined inside the class are called Member Functions.

Once a class has been declared, we can create the variables of that type by using the class name. For example:

Class_name cn;

Creates variable cn of type class_name. These class variables are called as objects. The main() cannot contain statements that access the data members directly. Instead they are to be accessed through the objects of the class. For example: cn.function1(), cn.item, cn.name etc.

To define the member function:

<u>Inside the class definition:</u>

```
class class_name{
int a, b;
public:

void get()
{
 executable statements;
}
};
```

<u>Outside the class definition:</u>

```
class class_name{
int a, b;
public:
void get();
};
void class_name::get()
{
  executable statements;
}
```

Solved exercise

Code snippet to read a string

```
void  main()
{
        const int MAX = 80; //max characters in string
        char str[MAX];              //string variable str
        cout<< "Enter a string: ";
        cin>>str;                   //put string in str
        cout<< "You entered: " <<str<<endl; //display string from str
}
```

Lab exercises

1. Write a program to perform following string operations without using string handling functions:
   a.) length of the string
   b.) string concatenation
   c.) string comparison
   d.) to insert a sub string
   e.) to delete a substring
2. Write a C++ program to define a class **student** with the data members to store name, roll no and grade of the student. Also write the member functions to read, display, and sort student information according to the roll number of the student. All the member functions will have array of objects as arguments.

Additional Questions:

1. Define a class **time** with data members hour, min, sec .Write the user defined functions to (i) Add (ii) To find difference between two objects of class time. Functions take two time objects as argument and return time object. Also write the display and read function.

LAB NO: 2 **Date:**

STACKS

Objectives:

In this lab students should be able to:

- Understand the concept of stacks.
- Implement the concept of stacks.
- Write and execute the application programs for stacks

Introduction:

The stack and the queue are data types that support insertion and deletion operations with well-defined semantics. Stack deletion deletes the element in the stack that was inserted the last, while a queue deletion deletes the element in the queue that was inserted the earliest. For this reason, the stack is often referred to as a LIFO (Last In First Out) datatype.

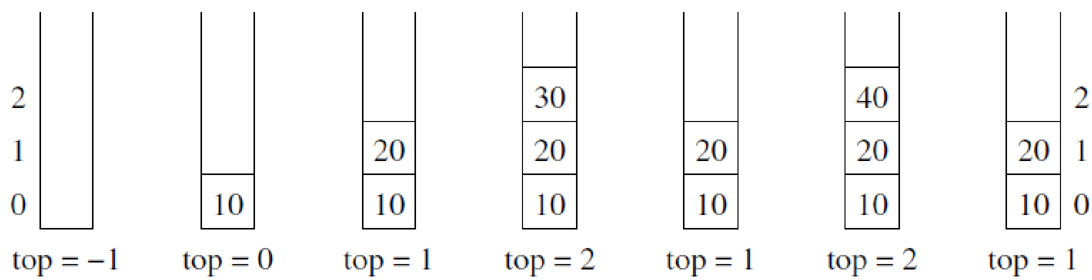


Figure: Stack Operations

Stack Implementation

Stacks and queues can be implemented using either arrays or linked lists. Although the burden of a correct stack or queue implementation appears to rest on deletion rather than insertion, it is convenient in actual implementations of these data types to place restrictions on the insertion operation as well. For example, in an array implementation of a stack, elements are inserted in a left-to-right order. A stack deletion simply deletes the right most element. A simple array implementation of a stack class is shown below:

```
class Stack {
    public:
    Stack(){ top = -1}; // Initialize the stack beyond the initial index for
empty stack
```

```
        void push();
        int pop();
        void display();
    private:
        int s[10]; // Size of the stack is currently ten
        int item; // The element that is added into the stack
        int top; // highest position in array that contains an element
};
```

Solved Exercise

[Implement the stack using arrays.**]**

```
#define STACK_SIZE 10
    class Stack {
     public:
       Stack(){ top = -1};
       void push();
       int pop();
       void display();
     private:
       int s[10];
       int item;
       int top;
    };
    void Stack:: push()
    {
      if(top = = STACK_SIZE-1)// Check if the index is beyond the stack size
      {
       cout<<"Stack Overflow \n";
       return;
    }
      top=top+1; //Increment the index on adding an element
      s[top]=item;// Add the element into the stack (array) as pointed by the index
    }

    int Stack::pop()
    {
      if(top = = -1) //Check if the stack index is beyond the initial index
```

```
  {
   cout<<"Empty Stack \n";
   return -1; //Return a value -1 if the stack is empty
 }
  return s[top - -]; // Return the topmost element
}
void Stack::display()
{
 int i;
 if(top==-1)
{
  cout<<"Empty Stack \n";
  return;
 }
 cout<< "Contents of stack\n";
 for(i=0;i<=top;i++)
      cout<<s[i];
}
```

Lab Exercise:

1. Write a program to implement stack concept.
2. Write a program to check whether a given string is a palindrome or not using stacks.
3. Write a program to convert a given decimal number to a number in any base using stack.

Additional Exercise

1. Write a program to implement Multiple stack using arrays.
2. Write a program to check for matching parenthesis in a given expression.

LAB NO: 3 **Date:**

QUEUES AND SPARSE MATRICES

Objectives:

In this lab students should be able to:

- Understand the concept of queues and sparse matrices.
- Implement the concept of queues and sparse matrices.
- Write and execute the application programs for sparse matrices

Introduction:

Queues are data types that support insertion and deletion operations with well-defined semantics. The queue is an FIFO (First In First out) data type. A dequeue (double ended queue) combines the stack and the queue by supporting both types of deletions. An array implementation of a queue is a bit trickier than that of a stack. Insertions can be made in a left-to-right fashion as with a stack. However, deletions must now be made from the left.

Consider a simple example of an array of size 5 into which the integers 10, 20, 30, 40 and 50 are inserted What if we are now required to insert the integer 60. On one hand, it appears that we are out of room as there is no more place to the right of 50. On the other hand, there are three locations available to the left of 40.

Figure: Queue Example

Solved Exercise:

[Implement Queue using arrays]

```
              #define q_size 20 // Size of the queue is initialized to 20
     class queue
              {
                 int front, rear;
                 int q[20]; // Size of the array used to implement queue is 20
               public:
                      void insertq(int item); // Add element into the queue
```

```
        int delq();              // Delete an element from the queue
        void display();          // Display elements of the queue
        queue()
         {
           front=0;// initialize the index which is used for  removing the
                   //element to the first inserted element
           rear=-1;// initialize the index which is used for adding an
                   //element to the queue beyond the initial index of zero
        }
};

void queue::insertq(int item)
{
        if(rear==q_size-1) //Check if the index reaches the size of
                           // the array storing the queue elements
        {
        cout<<"Queue overflow \n";
        return;
 }
rear=rear+1;     // Increment the index to store new element
q[rear]=item;    // Add the element/item  at the specified index
}
int queue::delq()
{
        if(front>rear) return -1;//Criteria for checking if the queue  is
                                 //empty.Return a value of -1 on queue empty
        return q[front++];// Return the contents of the queue which is the
                          //one which is to be deleted to the invoking function
}
void queue:: display()
{
int i;
if(front>rear) // Criteria to check if the queue is empty
{
        cout<<"Empty queue\n";
```

```
        return;
    }
cout<<"Contents:"
for(i=front;i<=rear;i++)
cout<<q[i];
}
```

Lab Exercise:

1. Write a program to implement queue concept.
2. Write a program to find the fast transpose of a sparse matrix represented using array of objects.

Additional Questions:

1. Write a program to find the transpose of a sparse matrix represented using array of objects.
2. Write a program to implement the circular queue using arrays.

LAB NO.4 **Date:**

APPLICATIONS OF STACKS

Objectives:

In this lab students should be able to:

- Write and execute the application programs for stacks

Introduction to the arithmetic expression conversion techniques:

Infix to postfix conversion: There is an algorithm to convert an infix expression into a postfix expression. It uses a stack; but in this case, the stack is used to hold operators rather than numbers. The purpose of the stack is to reverse the order of the operators in the expression. It also serves as a storage structure, since no operator can be printed until both of its operands have appeared.

Example: A * B + C becomes A B * C +

The order in which the operators appear is not reversed. When the '+' is read, it has lower precedence than the '*', so the '*' must be printed first.

We will show this in a table with three columns. The first will show the symbol currently being read. The second will show what is on the stack and the third will show the current contents of the postfix string. The stack will be written from left to right with the 'bottom' of the stack to the left.

	current symbol	operator stack content	postfix string
1	A		A
2	*	*	A
3	B	*	A B
4	+	+	A B * {pop and print the '*' before pushing the '+'}
5	C	+	A B * C
6	'\0'		A B * C +

Lab Exercise:

1. Write a program to input an infix expression and convert into its equivalent post fix form and display. Operands can be single character.
2. Write a program to evaluate a postfix expression. The input to the program is a postfix expression.
3. Write a program that converts a post fix expression to a fully parenthesized infix expression.
4. Write a program to input an infix expression and convert into its equivalent prefix form and display. Operands can be single character.
5. Write a program to evaluate prefix expression. The input to the program is a prefix expression.

Additional exercise:

1. Write a program that converts a prefix expression to a fully parenthesized infix expression.
2. Write a program to convert prefix expression to postfix.
3. Write a program to implement queue data structure using stack.

LAB NO. 5 **Date:**

LINKED LIST

Objectives:

In this lab, student will be able to:

- Understand and implement the concept of Linked list.
- Implement the applications of Linked List.

Introduction:

The linked list is an alternative to the array when a collection of objects is to be stored. The linked list is implemented using pointers. Thus, an element (or node) of a linked list contains the actual data to be stored *and* a pointer to the next node. Recall that a pointer is simply the address in memory of the next node. Thus a key difference from arrays is that a linked list does not have to be stored contiguously in memory.

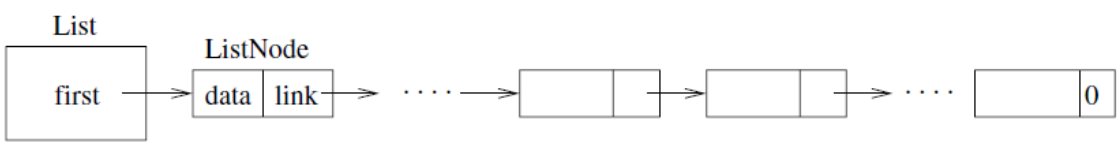


Figure: Structure of a Linked List

The code fragment below defines a linked list data structure, which is illustrated in Figure

```
Class ListNode {
        int data;
        ListNode *link;
        }
```

A chain is a linked list where each node contains a pointer to the next node in the list. The last node in the list contains a null (or zero) pointer.

Solved exercise

[Understand the concept of linked list and implement it]

Write a menu driven program to perform the following operations on linked list.

i) Create a list. ii) Display the list**.** iii) Delete the list

```
list *head=NULL;
list *tail=NULL;
class list
{
public:
     list();
     void insert();// to the end of the linked list
     void print();
     void delete();// deletes the first node
private:
    int val;     // A value that is stored in each list object
    list *next; //  next is a pointer to a list object
};
list::list(){
}
void main()
{
     clrscr();
     list l1;
     while(1)
     {
               cout<<"1. Insert    2.  Print    3.Delete    4.Exit\n";
               int ch;
               cin>>ch;
               switch(ch)
               {
               case 1: l1.insert();
                         break;
               case 2: l1.print(); break;
               case 3: l1.del();
                         break;
               default: exit(0);
               }
     }
}
```

```
void list:: insert()
{
     list * temp=new list; // temp is a pointer to a new list object
    temp→next =NULL;//Initialize the temp’s next object pointer to NULL
    cout<<”value:”;
    cin>>temp→val;//Accept the integer data and store it in the temp object
    if(head!=NULL && tail!=NULL)
    {       //If the linked list already exists
            tail→next=temp;  // Add a list object to the end of the linked list
            tail=temp;          // Rename the last node from temp to tail
    }
    else
    {      //If the linked list does not exist then head and tail is NULL
            tail=head=temp;//Both the object head and tail point to temp
    }
     return head;
}
void list::print()
{
    list *h=head; // h is holding the same address as pointed by head
    if(h==NULL)
    cout<<”List is empty”;
    while(h!=NULL)
    {
            cout<<”→”<<h→val<<endl;
            h=h→next; //Points to the next list object
    }
    return;
}
void list::del()
{
    list *temp=new list; // temp is a pointer to list object
    temp=head; // temp stores the address of head (first list object)
    if (head==NULL) {cout<<”Deletion not possible”; return;}
    if (head!=NULL) head=head→next; // head points to the next object
```

```
        if (head==NULL) tail=NULL;
        cout<<"element deleted\n"<< temp→next;
        delete(temp);
    }
```

Lab exercises

1. Write a menu driven program to perform the following operations on linked list.
   i) Insert an element in the beginning of the list
   ii) Insert an element at the end of the list
   iii) Insert an element before another element in the existing list
   iv) Insert an element after another element in the existing list
   v) Delete a given element from the list
   vi) Print the list

Additional Questions:

1. Write recursive functions for i) Creating a linked list ii) Traversing a linked list.
2. Let X = (x_1, x_2….x_n) and Y = (y_1, y_2….y_n) be 2 linked lists. Assume that, in each list, the nodes are in non-decreasing order of the data field values. Write an algorithm to merge two lists to obtain a new linked list Z in which the nodes are also in the non-decreasing order. Following the merge, X and Y do not exist as individual lists. Each node initially in X or Y is now in Z. Do not use additional nodes.
3. Let list1 = (x_1, x_2…..x_n) and list2= (y_1, y_2…..y_m). Write a function to merge list1 and list2 to obtain list3 = (x_1, y_1, x_2, y_2….x_m,y_m,x_{m+1}…x_n) for m<=n; and list3=(x_1, y_1,x_2,y_2…..x_n, y_n, x_{n+1}….x_m) for m>n.
4. Write a program to implement stack & queue using Singly linked lists.
5. Write the program which performs the following functions:
   - Reverse the list
   - Sort the list
   - Delete every alternate node in the list
   - Insert an element in a sorted list such that the order is maintained.

LAB NO: 6 **Date:**

DOUBLY LINKED LIST

Objectives:

In this lab, student will be able to:

- Write and execute programs on doubly linked list and applications of singly linked list.

Introduction to Doubly Linked List:

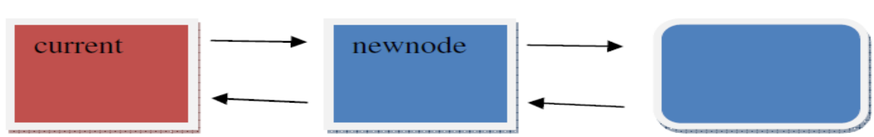
A doubly linked list is a list that contains links to next and previous nodes. Unlike singly linked lists where traversal are only one way, doubly linked lists allow traversals in both ways. A generic doubly linked list node can be designed as:

```
class dnode
    {
      int info;
      dnode *prev;
      dnode *next;
    } ;
```

The design of the node allows flexibility of storing any data type as the linked list data.

Inserting to a Doubly Linked Lists

Suppose a new node, *new node* needs to be inserted after the node *current,*



The following code can then be written

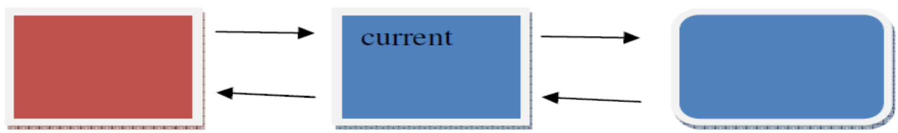
newnode➔ next = current➔next;   current➔next = newnode;

newnode➔prev = current; (newnode➔next)➔prev = newnode;

Doubly linked lists (DLL) are also widely used in many applications that deals with dynamic memory allocation and deallocation.

Deleting a Node from a Doubly Linked Lists

Suppose a new node, **current** needs to be deleted



The following code can then be written

node* N = current➔prev

N➔ next = current➔next;

(N➔next)➔prev = N;

free(current);

Solved exercise

[Write a program to create and display doubly linked list using the header node]

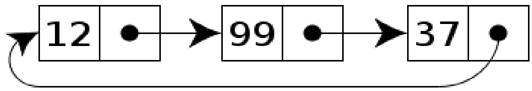
```
class dnode{
int info;  // Integer Value stored into  the object of type dnode
dnode *rlink; // rlink is a pointer to an object of type dnode
dnode *llink;
public:
   *dnode insert_front(dnode *header);
   *dnode del_front(dnode* header);
};

dnode* dnode::insert_front(dnode *header)
{
dnode *temp, *cur;
```

```
temp=new dnode; // Allocate space to store a dnode object
cout<< "Enter the element to be inserted\n";
cin>> temp→info; // Request the value to be entered from the user
cur = header→rlink; //obtain the address of first node after the header node
//store it in object cur. This node follows the node temp on insertion
header→rlink= temp; //insert temp between header & first (cur) node
temp→llink = header; // Connect links to the left and right of temp.
temp→rlink = cur;
cur→llink= temp; // cur's left pointer points to temp
return header;
}
dnode * dnode::del_front(dnode *header)
{
dnode *cur, *next;
if(header→rlink = = header)  //empty list
        {
        cout<< "Empty\n";
        return header;
        }
cur=header→rlink; //obtain first node after header
next=cur→rlink  //obtain the second node
header→rlink= next;
next→llink = header;
cout<<"Deleted Item is:" << cur→info;
delete (cur);
return header;
}
```

Introduction to polynomials and circular list

Circular List:

In the last node of a list, the link field often contains a null reference, a special value used to indicate the lack of further nodes. A less common convention is to make it point to the first node of the list; in that case the list is said to be 'circular' or 'circularly linked'; otherwise it is said to be 'open' or 'linear'.



In the case of a circular doubly linked list, the only change that occurs is that the end, or "tail", of the said list is linked back to the front, or "head", of the list and vice versa.

Polynomials:

A polynomial is an expression that contains more than two terms. A term is made up of coefficient and exponent. An example of polynomial is

$P(x) = 4x^3+6x^2+7x+9$

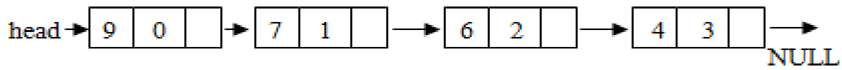
A polynomial thus may be represented using arrays or linked lists. Array representation assumes that the exponents of the given expression are arranged from 0 to the highest value (degree), which is represented by the subscript of the array beginning with 0. The coefficients of the respective exponent are placed at an appropriate index in the array. The array representation for the above polynomial expression is given below:

arr	9	7	6	4	(coefficients)
	0	1	2	3	(exponents)

A polynomial may also be represented using a linked list. A structure may be defined such that it contains two parts- one is the coefficient and second is the corresponding exponent. The class definition may be given as shown below:

```
class polynomial
{
   int coefficient;
   int exponent;
   polynomial *next;
};
```

Thus the above polynomial may be represented using linked list as shown below:



Sample Program:

[Write a program to add 2 polynomials represented as linked lists]

```
class node
{
     int coef;
     int exp;
     node *next;
   public:
          void display(node *);
          node *attach(int, int, node *);
          node *read_poly(node *);
          node *poly_add(node *, node *);
};

node display(node *first)
{
          node *temp;
          if(first==NULL) // Polynomial linked list does not exist
          {
                    cout<<"Polynomial does not exist\n";
                    return first;
          }
          temp=first;//Use temp for moving to next polynomial, so that the first pointer is not
                         // changed
          while(temp!=NULL)
          {
                    if(temp→coef<0)
                              // Positive value of coeff
                              cout<<temp→coef<<"x^" << temp→exp;
                    else
                              cout<< "+" <<temp→coef<< "x^" << temp→exp;
```

```
                temp=temp→next; // Next polynomial
        }
}
node *node ::attach(intcoef, intexp, node *first) // to the end of the list
{
node *temp=new node;
node *cur;
temp→coef = coef;
temp→exp=exp;
temp→next=NULL;
if(first==NULL) return temp; // temp has the address of the first node created
cur=first;
while(cur→next!=NULL) // Move along till the last node
cur=cur→next;
cur→next=temp; // attach the new node temp at the end of the linked list
return first;
}

node *node::read_poly(node *first)// read values for the coeff and exp until a value for
                                  //the coeff is given as 999
{
int coef, exp,i=0;
cout<< "Enter -999 to end the polynomial";
while(1)
 {
        cout<<"enter the term" <<i++ <<endl;
        cout<<"Coef=";
        cin>>coef;
        if(coef==-999)break;
        cout<<"Power of x=";
        cin>>exp;
        first=attach(coef, exp, first);
 }
return first;
}
```

```
node *node::poly_add(node *a, node*b)
{
node *c=NULL;
int coef, com;
while(a!=NULL && b!=NULL)
{
        if(a→exp == b→exp)//Both the linked list have the same exponential value
                com=0;
        else if(a→exp> b→exp)//a linked list had higher exponent as compared to b's
                                                //linked list
                com=1;
        else
                com= -1;

        switch(com)
        {
        case 0: coef=a→coef+b→coef;
                if(coef!=0)
                  c=attach(coef, a→exp,c);
                a=a→next; // Botha's and b's pointer have been moved forward
                b=b→next;
                break;
        case 1: c=attach(a→coef, a→exp, c);
                a=a→next;// Only'sa's linked list is added to the resultant linked list c and
                                // hence only a's pointer is moved forward
                break;
        default: c=attach(b→coef, b→exp, c);
                 b=b→next;
        }
    }
     while(a!=NULL)
        {
         c=attach(a→coef, a→exp, c);
         a=a→next;
```

```
        }
      while(b!=NULL)
      {
        c=attach(b→coef, b→exp, c);
        b=b→bext;
        }
  return c;
  }

  void main()
  {
   node *poly1=NULL, *poly2=NULL, *poly3=NULL, p;
   cout<<"Enter the first polynomial\n";
   poly1=p.read_poly(poly1);
   cout<<"Enter the second polynomial\n";
   poly2=p.read_poly(poly2);
   poly3=p.poly_add(poly1, poly2);
   cout<<"After adddtion\n";
   p.display(poly3);
   getch();
  }
```

Lab exercises

1. Write a menu driven program to perform the following on a doubly linked list
   - i.) Insert an element at the rear end of the list
   - ii.) Delete an element from the rear end of the list
   - iii.) Insert an element at a given position of the list
   - iv.) Delete an element from a given position of the list
   - v.) Insert an element after another element
   - vi.) Insert an element before another element
   - vii.) Print the list
2. Write a program to add two polynomials using doubly linked list.

Additional Questions:

1. Write a program to implement union and intersection of two doubly linked lists.

2. Write a program to implement addition of two long positive integer numbers.
3. Write a menu driven program to:
    i) Insert an element into a doubly linked circular list
    ii) Delete an element from a doubly linked circular list.
4. Write a program to add 2 polynomials using circular doubly linked list with head node.
5. Write a program to multiply two polynomials using circular doubly linked list with header node
6. Write a program to concatenate two doubly linked lists X1 and X2. After concatenation X1 is a pointer to first node of the resulting lists.

LAB NO: 7 **Date:**

TREES

Objectives:

In this lab, student will be able to:

- Understand and implement the concept of binary trees.
- Implement the traversal techniques and few applications based on traversal techniques of binary trees

Introduction:

Any tree can be represented as a binary tree. In fact binary trees are an important type of tree structure that occurs very often. The chief characteristics of a binary tree are the stipulation that the degree of any given node must not exceed two. A binary tree may have zero nodes. To define a binary tree formally- “A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called left subtree and right subtree.

Inorder Traversal: Informally, inorder traversal calls for moving down the tree toward the left until you can go no further. Then you “visit” the node, move one node to the right and continue. If you cannot move to the right, go back one more node. The pseudocode for inorder traversal is as given below:

```
void inorder(node *root)
{
  if(root==NULL) return;
  inroder(root→llink);
  cout<< root→info;
  inroder(root→rlink);
}
```

Preorder Traversal:

```
void preorder(node *root)
{
 if(root==NULL)return;
 cout<< root→info;
```

```
  preorder(root→llink);
  preorder(root→rlink);
}
```

Post Order Traversal:

```
void postorder(node *root)
{
  if(root==NULL)return;
  postorder(root→llink);
  postorder(root→rlink);
  cout<< root→info;
}
```

BINARY SEARCH TREES

Introduction:

A binary search tree is a rooted binary tree, whose internal nodes store a key (and optionally, an associated value) and each have two distinguished sub-trees, commonly denoted *left* and *right*. The tree additionally satisfies the binary search tree property, which states that the key in each node must be greater than all keys stored in the left sub-tree, and smaller than all keys in right sub-tree.

The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient; they are also easy to code.

Insertion:

Insertion begins as a search would begin; if the key is not equal to that of the root, we search the left or right subtrees as before. Eventually, we will reach an external node and add the new key-value pair (here encoded as a record 'newNode') as its right or left child, depending on the node's key. In other words, we examine the root and recursively insert the new node to the left subtree if its key is less than that of the root, or the right subtree if its key is greater than or equal to the root.

```
void insert(Node* root, int data)
 {if (!root)
            root = new Node(data);
elseif (data < root→data)
```

```
                insert(root→left, data);
        elseif (data > root→data)
                insert(root→right, data);
        }
```

Deletion:

There are three possible cases to consider:

- Deleting a node with no children: simply remove the node from the tree.
- Deleting a node with one child: remove the node and replace it with its child.
- Deleting a node with two children: call the node to be deleted *N*. Do not delete *N*. Instead, choose either its in-order successor node or its in-order predecessor node, *R*. Copy the value of *R* to *N*, then recursively call delete on *R* until reaching one of the first two cases. If you choose in-order successor of a node, as right sub tree is not NIL (Our present case is node has 2 children), then its in-order successor is node with least value in its right sub tree, which will have at a maximum of one sub tree, so deleting it would fall in one of first two cases.

Lab exercises

1. Write user defined functions to perform the following operations on binary trees.
   i.) Iteratively create a binary tree
   ii.) In order traversal (Iterative)
   iii.) Post order traversal (Iterative)
   iv.) Preorder traversal(Iterative)
   v.) Count the number of leaf nodes in a binary tree
2. Write a program to insert an element into a binary search tree.
3. Write a program to delete an element from a binary search tree.
4. Write a program to search for a given element in a binary search tree.

Additional exercise:

1. Write a program to check for equality of two trees.
2. Write a program to check if one tree is the mirror image of another tree.
3. Write a program to copy one tree to another.
4. Write a recursive function to i) Create a binary tree and ii) print a binary tree
5. Write a program to perform the following:

   i.) Print the parent of the given element
   ii.) Print the depth of a tree
   iii.) Print the ancestors of a given node

6. Write a program to implement level order traversal on binary search tree
7. Write a program to create a tree for a postfix expression and evaluate it.

LAB NO: 8 **Date:**

SORTING TECHNIQUES

Objective:

In this lab students should be able to:

- Understand the concept of Sorting.
- Implement different types of sorting techniques

Introduction:

A **sorting algorithm** is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) which require input data to be in sorted lists; it is also often useful for canonicalizing data and for producing human-readable output.

Quick Sort:

Quicksort is a divide and conquer algorithm which relies on a *partition* operation: to partition an array. An element called a *pivot* is selected. All elements smaller than the pivot is moved before it; all greater elements are moved after it. This can be done efficiently in linear time and in-place. The lesser and greater sublists are then recursively sorted. This yields average time complexity of O(n log n), with low overhead, and thus this is a popular algorithm. Efficient implementations of quicksort (with in-place partitioning) are typically unstable sorts and somewhat complex, but are among the fastest sorting algorithms in practice.

The steps are:

1. Pick an element, called a **pivot**, from the array.
2. Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

The base case of the recursion is arrays of size zero or one, which can never be sorted

Lab Exercise:

1. Write a program to sort a given list of elements using
   i. Bubble sort
   ii. Selection sort
   iii. Insertion sort
   iv. Quick sort
   v. Heap sort
   vi. Radix sort
   vii. Merge sort
2. Write program to perform the following:
   i. Linear Search
   ii. Binary Search

REFERENCES

1. Ellis Horowitz, Sartaj Sahni, Dinesh Mehta, “Fundamentals of Data Structures in C++”, (2e), Galgotia Publications, Reprint 2004.
2. Mark Allen Weiss, “Data Structures and Algorithm Analysis in C++”, (2e), Pearson Education, 2005.
3. Micheal T Goodrich, Roberto Tamassia, David Mount, “Data Structures and Algorithms in C++”, (2e), John Wiley & Sons, 2011.

C++ QUICK REFERENCE

PREPROCESSOR

```
                        // Comment to end of line
                        /* Multi-line comment */
#include <stdio.h>      // Insert standard header file
#include "myfile.h"     // Insert file in current directory
#define X some text     // Replace X with some text
#define F(a,b) a+b      // Replace F(1,2) with 1+2
#define X \
        some text       // Line continuation
#undef X                // Remove definition
#if defined(X)          // Condional compilation (#ifdef X)
#else                   // Optional (#ifndef X or #if !defined(X))
#endif                  // Required after #if, #ifdef
```

LITERALS

```
255, 0377, 0xff              // Integers (decimal, octal, hex)
2147463647L, 0x7fffffffl     // Long (32-bit) integers
123.0, 1.23e2                // double (real) numbers
‘a’, ‘\141’, ‘\x61’          // Character (literal, octal, hex)
‘\n’, ‘\\’, ‘\’’, ‘\”’,      // Newline, backslash, single quote, double quote
“string\n”                   // Array of characters ending with newline and \0
“hello” “world”              // Concatenated strings
true, false                  // bool constants 1 and 0
```

DECLARATIONS

```
int x;                  // Declare x to be an integer (value undefined)
int x=255;              // Declare and initialize x to 255
short s; long 1;        // Usually 16 or 32 bit integer (int may be either)
char c= ‘a’;            // Usually 8 bit character
unsigned char u=255; signed char m=-1; // char might be either
unsigned long x=0xffffffffL; // short, int, long are signed
float f; double d;      // Single or double precision real (never unsigned)
bool b=true;            // true or false, may also use int (1 or 0)
```

```
int a, b, c;                  // Multiple declarations
int a[10];                    // Array of 10 ints (a[0] through a[9])
int a[]={0,1,2};              // Initialized array (or a[3]={0,1,2}; )
int a[2][3]={{1,2,3},{4,5,6}; // Array of array of ints
char s[]= “hello”;            // String (6 elements including ‘\0’)
int* p;                       // p is a pointer to (address of) int
char* s= “hello”;             // s points to unnamed array containing "hello"
void* p=NULL;                 // Address of untyped memory (NULL is 0)
int& r=x;                     // r is a reference to (alias of) int x
enum weekend {SAT, SUN};// weekend is a type with values SAT and SUN
enum weekend day;             // day is a variable of type weekend
enum weekend {SAT=0,SUN=1}; // Explicit representation as int
enum {SAT,SUN} day;           // Anonymous enum
typedef String char*;         // String s; means char* s;
constint c=3;                 // Constants must be initialized, cannot assign
constint* p=a;                // Contents of p (elements of a) are constant
int* const p=a;               // p (but not contents) are constant
constint* const p=a;          // Both p and its contents are constant
r=x;                          // cr cannot be assigned to change x
```

STORAGE CLASSES

```
int x;                        // Auto (memory exists only while in scope)
staticint x;                  // Global lifetime even if local scope
externint x;                  // Information only, declared elsewhere
```

STATEMENTS

```
x=y;                          // Every expression is a statement
int x;                        // Declarations are statements
;                             // Empty statement
{                             // A block is a single statement
int x;                        // Scope of x is from declaration to end of
block
a;                            // In C, declarations must precede statements
}
if (x) a;                     // If x is true (not 0), evaluate a
```

```
else if (y) b;          // If not x and y (optional, may be repeated)
else c;                 // If not x and not y (optional)
while (x) a;            // Repeat 0 or more times while x is true
for (x; y; z) a;        // Equivalent to:  x; while(y) {a; z;}
do a; while (x);                // Equivalent to:  a; while(x) a;
switch (x) {            // x must be int
        case X1: a;     // If x == X1 (must be a const), jump here
        case X2: b;     // Else if x == X2, jump here
        default: c;     // Else jump here (optional)
}
break;                  // Jump out of while, do, for loop, or switch
continue;               // Jump to bottom of while, do, or for loop
return x;                       // Return x from function to caller
try { a; }
catch (T t) { b; }              // If a throws T, then jump here
catch (...)  { c; }             // If a throws something else, jump here
```

FUNCTIONS

```
int f(int x, int);              // f is a function taking 2 ints and returning int
void f();                       // f is a procedure taking no arguments
void f(int a=0);                // f() is equivalent to f(0)
f();                    // Default return type is int
inline f();             // Optimize for speed
f( ) { statements; }    // Function definition (must be global)
```

Function parameters and return values may be of any type. A function must either be declared or defined before it is used. It may be declared first and defined later. Every program consists of a set of global variable declarations and a set of function definitions (possibly in separate files), one of which must be:

int main() { statements... }   or

int main(intargc, char* argv[]) { statements... }

argv is an array of argc strings from the command line. By convention, main returns status 0 if successful, 1 or higher for errors.

EXPRESSIONS

Operators are grouped by precedence, highest first. Unary operators and assignment evaluate right to left. All others are left to right. Precedence does not affect order of evaluation which is undefined. There are no runtime checks for arrays out of bounds, invalid pointers etc.

T::X	// Name X defined in class T
N::X	// Name X defined in namespace N
::X	// Global name X
t.x	// Member x of struct or class t
p → x	// Member x of struct or class pointed to by p
a[i]	// i'th element of array a
f(x, y)	// Call to function f with arguments x and y
T(x, y)	// Object of class T initialized with x and y
x++	// Add 1 to x, evaluates to original x (postfix)
x--	// Subtract 1 from x, evaluates to original x
sizeof x	// Number of bytes used to represent object x
sizeof(T)	// Number of bytes to represent type T
++x	// Add 1 to x, evaluates to new value (prefix)
--x	// Subtract 1 from x, evaluates to new value
~x	// Bitwise complement of x
!x	// true if x is 0, else false (1 or 0 in C)
-x	// Unary minus
+x	// Unary plus (default)
&x	// Address of x
*p	// Contents of address p (*&x equals x)
x * y	// Multiply
x / y	// Divide (integers round toward 0)
x % y	// Modulo (result has sign of x)
x + y	// Add, or &x[y]
x – y	// Subtract, or number of elements from *x to *y
x << y	// x shifted y bits to left (x * pow(2, y))
x >> y	// x shifted y bits to right (x / pow(2, y))
x < y	// Less than
x <= y	// Less than or equal to
x > y	// Greater than

```
x >= y          // Greater than or equal to
x == y          // Equals
x != y          // Not equals
x& y            // Bitwise and (3 & 6 is 2)
x ^ y           // Bitwise exclusive or (3 ^ 6 is 5)
x | y           // Bitwise or (3 | 6 is 7)
x&& y           // x and then y (evaluates y only if x (not 0))
x || r          // x or else y (evaluates y only if x is false(0))
x = y           // Assign y to x, returns new value of x
x += y          // x = x + y, also -=  *=  /=  <<=  >>=  &=  |=  ^=
x ?y : z        // y if x is true (nonzero), else z
throw x         // Throw exception, aborts if not caught
                // evaluates x and y, returns y (seldom used)
```

IOSTREAM.H, IOSTREAM

```
cin>> x >> y;               // Read words x and y (any type) from stdin
cout<< “x=” << 3 <<endl;    // Write line to stdout
cerr « x « y « flush;       // Write to stderr and flush
c = cin.get();              // c = getchar();
cin.get(c);                 // Read char
cin.getline(s, n, ‘\n’);    // Read line into char s[n] to ‘\n’, (default)
if (cin)                    // Good state (not EOY)?
                            // To read/write any type T:
```

<u>STRING (Variable sized character array)</u>

```
string s1, s2= “hello”;     //Create strings
sl.size(), s2.size();       // Number of characters: 0, 5
sl += s2 +  ‘  ’  + “world”;  // Concatenation
sl == “hello world”;        // Comparison, also <, >, !=, etc.
s1[0];                      // ‘h’
sl.substr(m, n);            // Substring of size n starting at sl[m]
sl.c_str();                 // Convert to const char*
getline(cin, s);            // Read line ending in ‘\n’
```

```
asin(x); acos(x); atan(x);          // Inverses
atan2(y, x);                    // atan(y/x)
sinh(x); cosh(x); tanh(x);          // Hyperbolic
exp(x); log(x); log10(x);           // e to the x, log base e, log base 10
pow(x, y); sqrt(x);             // x to the y, square root
ceil(x); floor(x);                  // Round up or down (as a double)
fabs(x); fmod(x, y);            // Absolute value, x mod y
```