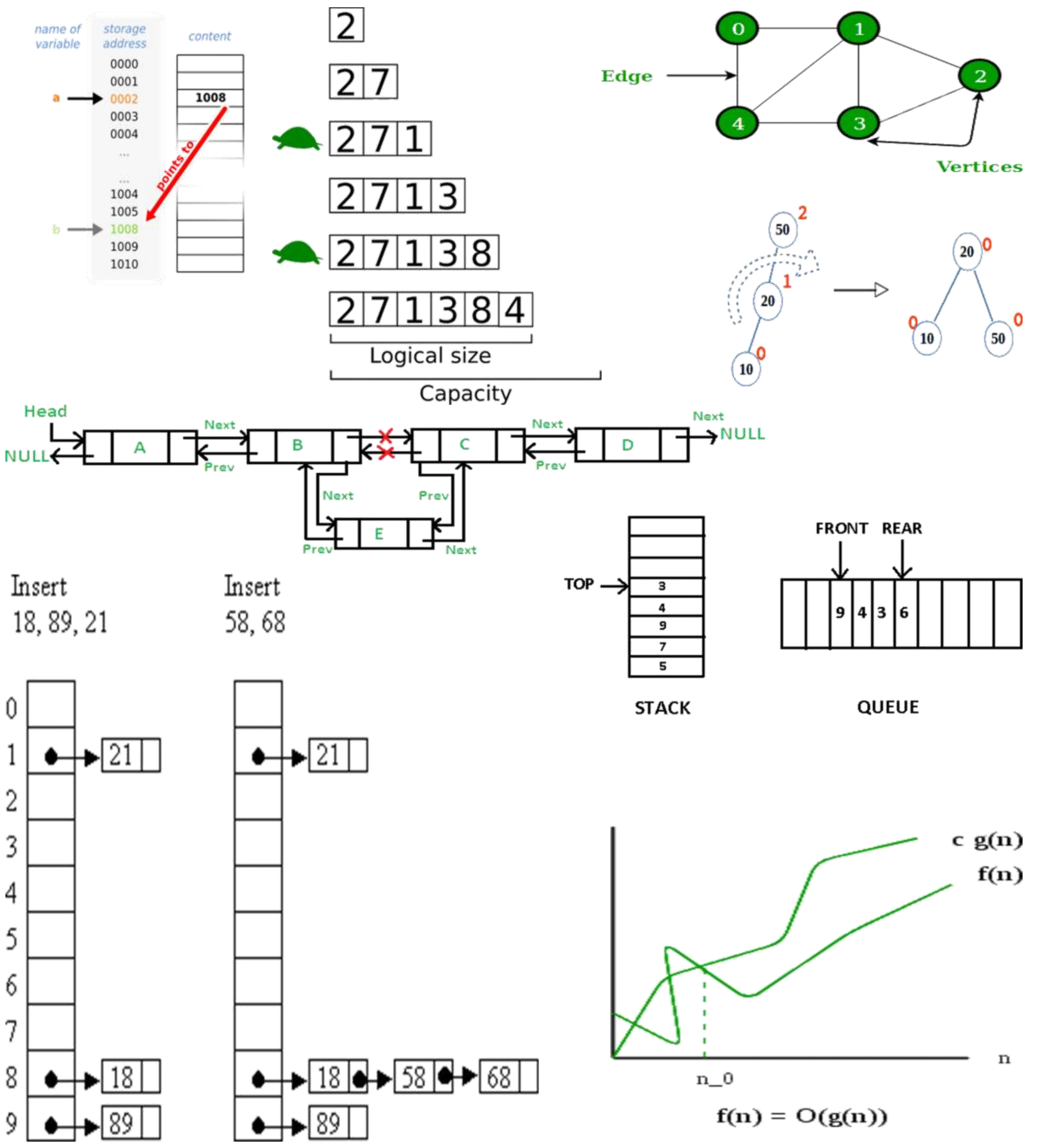
CS221 L Data Structures and Algorithms Lab Manual

Data structures (Ghulam Ishaq Khan Institute of Engineering Sciences and Technology)

CS221L Data Structures Lab

Faculty of Computer Science and Engineering



Ghulam Ishaq Khan Institute of Engineering Sciences and Technology, Topi.

Table of Contents

I. Programming Examples

II. WEEKLY BREAK DOWN

Week	**Contents/Topics**
Week 1	Pointers and Arrays (Static & Dynamic)
Week 2	Structures
Week 3	Linked Lists
Week 4	Stacks
Week 5	Queues
Week 6	Sorting I
Week 7	Sorting II
Week 8	Binary Search Trees
Week 9	AVL Trees
Week 10	Graphs
Week 11	Hashing
Week 12	Open Ended Lab

III. SYSTEM REQUIREMENTS

HARDWARE REQUIREMENT

- Core i3 or Above
- 2 GB RAM
- 20 GB HDD

SOFTWARE REQUIREMENT

- Windows 8x or above
- Dev C++ IDE / Visual Studio

IV. EVALUATION CRITERIA

Overall Grading Policy	
Assessment Items	**Percentage**
Lab Evaluation	30%
Project	10%
Midterm Exam	20%
Final Exam	40%

Rubric

Every lab task shall contain one or more tasks and every question shall be evaluated on the following criteria:

Running: 2
- Runs without exception: 1
- Interactive menu: 0.5
- Displays proper messages/operations clearly: 0.5

Completeness: 3
- Sub task 1: 1
- Sub task 2: 1
- Sub task 3: 1

Accuracy: 4
- Appropriate data structure: 1
- Efficient (Computational Complexity) algorithms: 1
- Passes all test cases: 1
- Viva: 1

Clarity 1 (No credit, if not relevant implementation):
- Indentation: 0.5
- Meaningful/relevant variable/function names: 0.5

*the grading policy is tentative and could be changed depending on the instructor suit.

V. INSTRUCTIONS FOR STUDENTS

- Students are expected to read respective lab contents before attending it
- Since the contents of a lab are covered in class before practicing in the lab hence, students are expected to read, understand and run all examples and sample problems before attending the lab
- Reach the lab on time. Late comers shall not be allowed to attend the lab
- Students need to maintain 100% attendance in lab if not a strict action will be taken.
- Wear your student card in the lab
- Mobile phones, USBs, and other electronic devices are strictly prohibited in the lab
- In case of hardware/power issues, immediately inform lab attendant and do not attempt to fix it by yourself
- In case of queries during lab task implementation, raise your hand, inform your instructor and wait for your turn

Lab 1 Pointers and Arrays (Static & Dynamic)

1.1. Introduction

In previous labs we have learned and practiced C++ variables and arrays. We know that every variable has an associated data type, name, address and value. For example

```
int x = 10;
```

Here, x is the name of a variable. The data type of x is integer and its value is 10. So, what is the address of variable x? Every variable is stored in memory at a specific location in memory. The starting address of the memory location where a variable is stored, is called the address of that variable. In C++ the address of a variable can be accessed using & operator. For example: int x = 10;

```
cout<<&x;
```

A pointer is a variable that can store memory address of another variable. We call them pointers because they "point" to another variable. Like any other variable, pointer variable has a data type, name, address and value. Unlike other variables the value of a pointer variable is address of another variable. Another important difference of pointer from normal variable is the size. The size of a variable depends upon the data type of that variable but every pointer variable has same size.

1.2. Pointers in C++

1.2.1. Pointer declaration

In C++ pointers are declared as:

```
DataType * Pointer;
```

- **DataType** can be an int, float, char, or anyother user defined data type
- * pronounced as "star" is an operator that tells the compiler that the variable declared is of type pointer
- **Pointer** is the name of pointer variable it can be any valid variable name in C++

In Example 1-1, ptrX is a pointer variable of type int. So, it can store the address of an integer type variable. We stored address of variable x in pointer variable ptrX. So, the output of &x and ptrX is same 0x24fe3c (i.e. the address of variable x).

Example 1-1 Pointer declaration

Code

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

int main()
{
        int x = 10;
        int *ptrX;
        ptrX = &x;

        cout<<"x = "<<x<<endl;
        cout<<"&x = "<<&x<<endl;
        cout<<"ptrX = "<<ptrX<<endl;
        cout<<"&ptrX = "<<&ptrX<<endl;

        return 0;
}
```

Output

```
x = 10
&x = 0x24fe3c
ptrX = 0x24fe3c
&ptrX = 0x24fe30
```

1.2.2. The Asterisk Operator (*)

There are two roles of asterisk operator in pointers: declaration and de-referencing.

1.2.3. Asterisk for Pointer Declaration

The asterisk operator is used for declaration of a pointer that differentiates it from other variables as explained in previous section. Please note that if you use * anywhere else in your code (after pointer declaration) before the pointer name then it is evaluated as de-referencing operator by the compiler.

1.2.4. Asterisk for De-Referencing

The asterisk operator is used as a de-reference operator which allows the compiler to jump to the memory location specified by the value of a pointer (which is the address of another variable) and get the value. In Example 1-2, x is a variable and ptrX is a pointer. The address of variable x is stored in ptrX. *ptrX gives us the value of variable x.

Example 1-2 Access the value of a variable using pointer

Code

```
#include<iostream>
```

```
using namespace std;

int main()
{
	int x = 10;
	int *ptrX;
	ptrX = &x;

	cout<<"x = "<<x<<endl;
	cout<<"*ptrX = "<<*ptrX<<endl<<endl;

	cout<<"&x = "<<&x<<endl;
	cout<<"ptrX = "<<ptrX<<endl;
	return 0;
}
```

Output

```
x = 10
*ptrX = 10

&x = 0x24fe34
ptrX = 0x24fe34
```

As we know that ptrX points towards variable x. If we update the value of x then *ptrX shows us the updated value and vice versa Example 1-3.

Example 1-3 Update the value of a variable using pointer

Code

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

int main()
{
	int x = 10;
	int *ptrX = &x;

	cout<<"x = "<<x<<endl;
	cout<<"*ptrX = "<<*ptrX<<endl<<endl;

	x = 20;
	cout<<"After updating the value of x"<<endl;
	cout<<"x = "<<x<<endl;
	cout<<"*ptrX = "<<*ptrX<<endl<<endl;
```

```
        cout<<"After updating the value of *ptrX"<<endl;
        *ptrX = 30;
        cout<<"x = "<<x<<endl;
        cout<<"*ptrX = "<<*ptrX<<endl<<endl;
        return 0;
}
```

Output

```
x = 10
*ptrX = 10

After updating the value of x
x = 20
*ptrX = 20

After updating the value of *ptrX
x = 30
*ptrX = 30
```

1.2.5. Pointer Arithmetic

Pointer arithmetic is different from regular variable (int, float etc.) arithmetic. When we increment a pointer by 1, it increments by the size of its data type. Moreover, only addition and subtraction are allowed in pointers. Let us examine pointer arithmetic by an Example 1-4. Pointer ptrX is pointing towards an int type variable x and pointer ptrY is pointing towards a double type variable y. The address of x and value of ptrX in this example is 0x24fe30. The output of ptrX+1 is 0x24fe34, not 0x24fe31, why? Because, when we increment a pointer variable by 1, it increments by the size of its data type. In this case data type of ptrX is int and size of int is 4 bytes. So, ptrX+1 gives the output 0x24fe34. We can see that initial value of ptrY is 0x24fe20 and ptrY+1 gives 0x24fe28 output because of its data type double that has size 8 bytes, in this case. Addition, subtraction and decrement operations also work the same way for pointers. Students should try it.

Example 1-4 Pointer arithmetic

Code

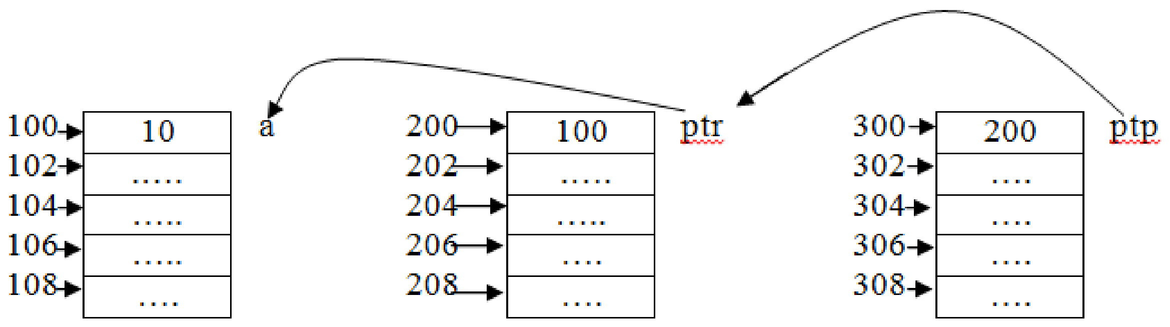
```
#include<iostream>
using namespace std;

int main()
{
        int x = 10;
        double y = 20;
        int *ptrX = &x;
        double *ptrY = &y;

        cout<<"ptrX   = "<<ptrX<<endl;
        cout<<"ptrX+1 = "<<ptrX+1<<endl;
        cout<<"ptrX+2 = "<<ptrX+2<<endl;
        cout<<endl;

        cout<<"ptrY   = "<<ptrY<<endl;
        cout<<"ptrY+1 = "<<ptrY+1<<endl;
        cout<<"ptrY+2 = "<<ptrY+2<<endl;
        cout<<endl;

        cout<<"Size of int = "<<sizeof(int)<<endl;
        cout<<"Size of double = "<<sizeof(double)<<endl;
        return 0;
}
```

Output

```
ptrX     = 0x24fe30
ptrX+1 = 0x24fe34
ptrX+2 = 0x24fe38

ptrY     = 0x24fe20
ptrY+1 = 0x24fe28
ptrY+2 = 0x24fe30

Size of int = 4
Size of double = 8
```

1.2.6. Pointer to Pointer (double pointer)

Pointers in C/C++ can point to a variable of any data type as well as other pointers too. We can store the address of a pointer in another pointer. This is called pointer to pointer access method. Consider the following graphical explanation.



Variable a is an integer variable which is stored at memory location 100 and its value is 10. ptr is an integer type pointer which is stored at 200 and it points to memory location 100. Pointer variable ptp is stored at 300 and points to a memory location 200 i.e. points to another pointer ptr. Please note that *ptr means the value of variable a (10), *ptp means value of ptr, which is the address of variable a (100). Few important points to note here are:

- `cout<<ptp;`
  Shows the address 200 (value of double pointer ptp)
- `cout<<*ptp;`
  results in 100 (value stored at address 200 - dereferencing)
- `cout<<**ptp;`
  results in 10 (value stored at address 100 – two level dereferencing)

Students are suggested to experiment pointers and double pointers by displaying addresses and values using cout statements as shown in Example 1-5.

Example 1-5 Pointer to pointers (double pointers)

Code

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

int main()
{
      int    a   = 10;
      int *ptr   = &a;
      int **ptp  = &ptr;

      cout<<"a       = "<<a<<endl;
      cout<<"&a      = "<<&a<<endl;
      cout<<"ptr     = "<<ptr<<endl;
      cout<<"&ptr    = "<<&ptr<<endl;
      cout<<"ptp     = "<<ptp<<endl;
      cout<<"&ptp    = "<<&ptp<<endl;
      cout<<endl;

      cout<<"Value of a can be accessed"<<endl;
      cout<<"Using variable a            = "<<a<<endl;
      cout<<"Using single pointer *ptr   = "<<*ptr<<endl;
      cout<<"Using double pointer **ptp = "<<**ptp<<endl;
      cout<<endl;

      cout<<"ptr is pointer of a because it has stored the address of
a"<<endl;
      cout<<"&a    = "<<&a<<endl;
      cout<<"ptr   = "<<ptr<<endl;
      cout<<endl;

      cout<<"ptp is a pointer of ptr because it has stored the address of
ptr"<<endl;
      cout<<"&ptr    = "<<&ptr<<endl;
      cout<<"ptp     = "<<ptp<<endl;
      cout<<endl;

      cout<<"You can access both addresses and values of all relevant
variables using ptp"<<endl;
```

```
      cout<<"Address of ptp using &ptp                   = "<<&ptp<<endl;
      cout<<"Value of ptp (address of ptr) using ptp = "<<ptp<<endl;
      cout<<"Value of ptr (address of a) using *ptp  = "<<*ptp<<endl;
      cout<<"Value of a using **ptp                    = "<<**ptp<<endl;
      return 0;
}
```

Output

```
a       = 10
&a      = 0x24fe3c
ptr     = 0x24fe3c
&ptr    = 0x24fe30
ptp     = 0x24fe30
&ptp    = 0x24fe28

Value of a can be accessed
Using variable a           = 10
Using single pointer *ptr  = 10
Using double pointer **ptp = 10

ptr is pointer of a because it has stored the address of a
&a    = 0x24fe3c
ptr   = 0x24fe3c

ptp is a pointer of ptr because it has stored the address of ptr
&ptr    = 0x24fe30
ptp     = 0x24fe30

You can access both addresses and values of all relevant variables using
ptp
Address of ptp using &ptp              = 0x24fe28
Value of ptp (address of ptr) using ptp = 0x24fe30
Value of ptr (address of a) using *ptp  = 0x24fe3c
Value of a using **ptp                  = 10
```

1.2.7. Pointers and Functions

We have covered functions in our previous labs. We know that there are two ways to pass arguments to a function: call by value and call by reference. Call by value copies the value of an argument into the formal parameter of the subroutine. Any changes made to the parameter have no effect on the original argument passed to the function. In call by reference, the address of an argument is copied into the parameter. The address is used to access the actual argument used in the call. This means that changes made to the parameter affect the original variable passed as argument. There is a third way, call by pointer. Like call by reference, original variable passed to function gets updated. The syntax of all three function calls is explained in Example 1-6.

Example 1-6 Call value, call by reference and call by pointer

Code

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

void callByValue(int x){
	x = 10;
}

void callByReference(int &x){
	x = 20;
}

void callByPointer(int *x){
	*x = 30;
}

int main()
{
	int x = 5;
	cout<<" Initially"<<endl;
	cout<<"x = "<<x<<endl;
	cout<<endl;

	callByValue(x);
	cout<<"After call by value"<<endl;
	cout<<"x = "<<x<<endl;
	cout<<endl;

	callByReference(x);
	cout<<"After call by reference"<<endl;
	cout<<"x = "<<x<<endl;
	cout<<endl;

	callByPointer(&x);
	cout<<"After call by pointer"<<endl;
	cout<<"x = "<<x<<endl;
	return 0;
}
```

Output

```
 Initially
x = 5

After call by value
x = 5

After call by reference
x = 20

After call by pointer
x = 30
```

1.3. Pointers and Arrays

There are two important concepts to remember while studying the relationship between pointers and arrays:

- Array elements are always stored in contiguous memory locations.
- Array name is a pointer to the first element in the array but mind it; array name is a constant pointer on which normal rules of pointers are not applied.

The following program illustrates these two concepts

Example 1-7 Arrays

Code

```
#include <iostream>
using namespace std;
int main()
{
    int numbers[5], sum = 0;
    cout << "Enter 5 numbers: ";

    //  Storing 5 number entered by user in an array
    //  Finding the sum of numbers entered
    for (int i = 0; i < 5; ++i)
    {
        cin >> numbers[i];
        sum += numbers[i];
    }

    cout << "Sum = " << sum << endl;

    return 0;
}
```

Output

```
 Enter 5 numbers: 1
 5
 2
 4
 3
 Sum = 15
```

Example 1-8 Array and Pointer

Code

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

int main () {
   // an array with 5 elements.
   double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
   double *p;

   p = balance;

   // output each array element's value
   cout << "Array values using pointer " << endl;

   for ( int i = 0; i < 5; i++ ) {
      cout << "*(p + " << i << ") : ";
      cout << *(p + i) << endl;
   }
   cout << "Array values using balance as address " << endl;

   for ( int i = 0; i < 5; i++ ) {
      cout << "*(balance + " << i << ") : ";
      cout << *(balance + i) << endl;
   }

   return 0;
}
```

Output

```
Array values using pointer
*(p + 0) : 1000
*(p + 1) : 2
*(p + 2) : 3.4
*(p + 3) : 17
*(p + 4) : 50
Array values using balance as address
*(balance + 0) : 1000
*(balance + 1) : 2
*(balance + 2) : 3.4
*(balance + 3) : 17
*(balance + 4) : 50
```

Obviously, a question arises as to which of the above two methods should be used when? Accessing array elements by pointers is always faster than accessing them subscripts. However, from the point of view of convenience in programming subscript method is used.

We already know that on mentioning the name of the array we get its base address. Thus by saying *num we would be able to refer to the zeroth element of the array, that is 1. One can easily see that *num and *(num+0) both refer to 1. Similarly, by saying *(num+1) we can refer to the first element that is 2. What the C++ compiler does internally is that when we say num[i] it converts the expression to *(num+i).

Thus, all the following notations are same:

```
num[i]
*(num+i)
*(i+num)
```

```
i[num] (This notation may appear strange but amazingly it gives the
sameresult. Don't understand it just as a concept, try it out in your
code)
```

1.4. Dynamic Arrays

So far, we have studied static arrays. The size of static arrays cannot be changed on run time. Therefore, dynamic array is used. The static declaration of an array requires the size of the array to be known in advance. What if the actual size of the list exceeds its expected size? Solution?

1.4.1. Declaration

The syntax to declare dynamic arrays is:

```
pointer = new type
pointer = new type [number_of_elements]

int * foo;
foo = new int [5];
```

1.5. Example Problems

Problem Statement
Write a C++ program that manages a list of values using the concept of dynamic memory allocation. Your program must have following operations (implemented using a separate function): 1. Insert an element   a. Beginning   b. End   c. Given index (optional/not mandatory) 2. Find an element 3. Show all elements   a. From first to last   b. From last to first (optional/not mandatory) 4. Delete an Element   a. You will prompt user to enter the number to be deleted   b. Use search method to find the index   c. Perform delete operation
Solution

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

class List{
      private:
            int *students;
            int used; // stores used size of array
            int maxSize;

            // Max size of list
          int getSize()
      {
            return sizeof(students)/sizeof(students[0]);
            }

    public:
      List()
      {
            cout<<"Enter size of the list:";
            cin>>this->maxSize;
            students = new int[this->maxSize];
            this->used    = 0;
            }

            // add new element at the begining
            void insertStart(int student)
            {
                  insert(student, 0);
            }

            // add new element at the end
            void insertEnd(int student)
            {
                  insert(student, this->used);
            }

            // add new element at a specific location
            void insert(int student, int index)
            {
                  if(!this->validIndex(index))
                  {
                        return;
                  }
                  if(this->used < this->maxSize)
                  {
                        int i;
                        // shift each element (right) by one location
                        for(i=this->used; i > index; i--)
                        {
                              this->students[i] = this->students[i-1];
                        }
                        this->students[index] = student;
                        this->used++;
                  }else{
                        cerr<<"Memory full"<<endl;
                  }
            }

            /*
             * Find an element
             * return index if element found otherwise return -1
             */
      int find(int student)
      {
            int N = this->used;
            int index = -1;

            for(int i=0; i < N; i++)
            {
                  if(this->students[i] == student)
                  {
                        index = i;
                        break;
                        }
                  }
                  return index;
            } //find ends

            /*
             * Display all elements start to end
             */
      void showAll(){
            if(this->used == 0)
            {
                  cout<<"List is empty"<<endl;
                  return;
                  }
            int N = this->used;
            for(int i=0; i < N; i++)
            {
                  cout<<"["<<i<<"] => "<<this->students[i]<<endl;
                  }
            } //showAll ends

            /*
             * Display all elements end to start
             */
      void showAllReverse(){
            if(this->used == 0)
            {
                  cout<<"List is empty"<<endl;
                  return;
                  }
            int N = this->used;
            for(int i=N-1; i >= 0; i--)
            {
                  cout<<"["<<i<<"] => "<<this->students[i]<<endl;
                  }
            } //showAll ends

            /*
             * display
             * index
             */
      int display(int index)
      {
            if(index > this->used)
            {
                  cout<<"Invalid index"<<endl;
                  }else{
                        cout<<"["<<index<<"]          =>          "<<this-
>students[index]<<endl;
                  }
            }

            /*
```

```
                * remove a student
                *
                */
        void remove(int student)
        {
                        int index = this->find(student);
                        if(index >= 0){
                                this->display(index);
                                int confirm = 0;
                                cout<<"Do you really want to delete it?"<<endl;
                                cout<<"1. Yes"<<endl;
                                cout<<"Anyother number. No"<<endl;
                                cout<<"Enter your choice: ";
                                cin>>confirm;
                                if(confirm == 1)
                                {
                                        for(int i=index; i < this->used-1; i++)
                                        {
                                                this->students[i]       =       this-
>students[i+1];
                                        }
                                        this->used--;
                                        cout<<"student delete successfuly"<<endl;
                                }

                        }else{
                                cout<<"student not found"<<endl;
                        }
                }

                bool validIndex(int index)
                {
                        if(index > this->used)
                        {
                                cout<<"Index must be less than or equal to "<<this-
>used<<endl;
                                return false;
                        }

                        if(index < 0)
                        {
                                cout<<"Index cannot be negative"<<endl;
                                return false;
                        }

                        return true;
                }

}; // List class ends here

int showMenu();
void printShashkaLine(char, int);

int main()
{
        List students;
        int choice;
        bool again = true;
        int student;
        int index;
        while(again){
                choice = showMenu();
                switch(choice)
                {
                        case 0: //exit program
                                again = false;
                                break;

                        case 1: // Insert start
                                cout<<"Enter student: ";
                                cin>>student;
                                students.insertStart(student);
                                break;

                        case 2: // Insert End
                                cout<<"Enter student: ";
                                cin>>student;
                                students.insertEnd(student);
                                break;

                        case 3: // Insert at given index
                                cout<<"Enter index: ";
                                cin>>index;

                                if(!students.validIndex(index))
                                {
                                        break;
                                }
                                cout<<"Enter student: ";
                                cin>>student;
                                students.insert(student, index);
                                break;

                        case 4: // find a student
                                cout<<"Enter student: ";
                                cin>>student;
                                index = students.find(student);
                                if(index >= 0){
                                        students.display(index);
                                }else{
                                        cout<<"student not found"<<endl;
                                }
                                break;

                        case 5: //show all
                                cout<<"All students (start to end)"<<endl;
                                students.showAll();
                                break;

                        case 6: //show all
                                cout<<"All students (end to start)"<<endl;
                                students.showAllReverse();
                                break;

                        case 7: // Remove a student
                                cout<<"Enter student: ";
                                cin>>student;
                                students.remove(student);
                                break;

                        default:
```

```
                        cerr<<"Invalid choice"<<endl;
                        cout<<"Kindly   select   a   valid   number   from
menu"<<endl;
                        break;
            }
            system("pause");
            system("cls");
      }
      cout<<"Thank you for using our program"<<endl;
}

int showMenu()
{
      int choice;
      int numbers = 0;
      printShashkaLine('=', 50);
      cout<<"\t\t My Program Menu"<<endl;
      printShashkaLine('=', 50);

      cout<<numbers++<<". Exit Program"<<endl;

      cout<<numbers++<<". Insert a student at the beginning"<<endl;
      cout<<numbers++<<". Insert a student at the end"<<endl;
      cout<<numbers++<<". Insert a student at a specific index"<<endl;

      cout<<numbers++<<". Find a student"<<endl;

      cout<<numbers++<<". Show all students (start to end)"<<endl;
      cout<<numbers++<<". Show all students (End to start)"<<endl;

      cout<<numbers++<<". Remove a student"<<endl;

      printShashkaLine('=', 50);
      cout<<"Enter your choice: ";
      cin>>choice;
      return choice;
}

void printShashkaLine(char element, int noOfTimes)
{
      for(int i=0; i < noOfTimes; i++)
      {
            cout<<element;
      }
      cout<<endl;
}
```

Lab 2 **Structures**

2.1. Introduction

There are many instances in programming where we need more than one variable in order to represent something. For example, to represent yourself, you might want to store your name, your birthday, your height, your weight, or any other number of characteristics about yourself. Structures are a way of storing many different values in variables of potentially different types under the same name. This makes it a more modular program, which is easier to modify because its design makes things more compact. Structs are generally useful whenever a lot of data needs to be grouped together--for instance, they can be used to hold records from a database or to store information about contacts in an address book. In the contacts example, a **struct** could be used that would hold all the information about a single contact--name, address, phone number, and so forth.

As structure is a group of data elements grouped together under one name. These data elements, known as members, can have different types and different lengths. Data structures are declared in C++ using the following syntax:

```
struct structure_name {
member_type1 member_name1;
member_type2 member_name2;
member_type3 member_name3;
.
.
} object_names;
```

where structure_name is a name for the structure type, object_name can be a set of valid identifiers for objects that have the type of this structure. Within braces { } there is a list with the data members, each one is specified with a type and a valid identifier as its name.
The first thing we must know is that a data structure creates a new type: Once a data structure is declared, a new type with the identifier specified as structure_name is created and can be used in the rest of the program as if it was any other type. For example:

```
struct product{
      int weight;
      float price;
};

product apple;
product banana, melon;
```

It is important to clearly differentiate between what is the structure type name, and what is an object (variable) that has this structure type. We can instantiate many objects (i.e. variables, like apple, banana and melon) from a single structure type (product).

Once we have declared our three objects of a determined structure type (apple, banana and melon) we can operate directly with their members. To do that we use a dot (.) inserted between the object name and the member name. For example, we could operate with any of these elements as if they were standard variables of their respective types:

```
apple.weight
apple.price
banana.weight
banana.price
melon.weight
melon.price
```

Each one of these has the data type corresponding to the member they refer to: apple.weight, banana.weight and melon.weight are of type int, while apple.price, banana.price and melon.price are of type float.
Here is an example program:

Example 2-1 Structures

Code

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

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

};

int main()
{
      //2 possible ways to declare a structure variable
      struct Books book1;
      Books book2;

      strcpy(book1.title, "Data Structures using C++");
      strcpy(book1.author, "DS Malik");
      strcpy(book1.subject, "Programming" );
      book1.book_id = 1234;

      strcpy(book2.title, "Data Structures & Algorithm Analysis in C++");
      strcpy(book2.author, "Mark Allen Weiss");
      strcpy(book2.subject, "Programming" );
      book2.book_id = 5678;

      cout<<"Book1 title : " << book1.title<<endl;
      cout<<"Book1 author : " << book1.author<<endl;
      cout<<"Book1 subject : " << book1.subject<<endl;
      cout<<"Book1 id: " << book1.book_id<<endl;
      cout<<endl;
      cout<<"Book2 title : " << book2.title<<endl;
      cout<<"Book2 author : " << book2.author<<endl;
      cout<<"Book2 subject : " << book2.subject<<endl;
      cout<<"Book2 id: " << book2.book_id<<endl;

}
```

Output

```
Book1 title : Data Structures using C++
Book1 author : DS Malik
Book1 subject : Programming
Book1 id: 1234

Book2 title : Data Structures & Algorithm Analysis in C++
Book2 author : Mark Allen Weiss
Book2 subject : Programming
Book2 id: 5678
```

2.2. Structures as Function Arguments

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

Example 2-2 Structure and functions

Code

```
//structure name passing as an argument to a function

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

void printBook(struct Books book);

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

};

int main()
{
      //2 possible ways to declare a structure variable
      struct Books book1;
      Books book2;

      strcpy(book1.title, "Data Structures using C++");
      strcpy(book1.author, "DS Malik");
      strcpy(book1.subject, "Programming" );
      book1.book_id = 1234;

      strcpy(book2.title, "Data Structures & Algorithm Analysis in C++");
      strcpy(book2.author, "Mark Allen Weiss");
      strcpy(book2.subject, "Programming" );
      book2.book_id = 5678;

      cout<<"Book1: "<<endl;
      printBook(book1);

      cout<<endl<<"Book2: "<<endl;
      printBook(book2);



}

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

Output

```
 Book1 title : Data Structures using C++
 Book1 author : DS Malik
 Book1 subject : Programming
 Book1 id: 1234

 Book2 title : Data Structures & Algorithm Analysis in C++
 Book2 author : Mark Allen Weiss
 Book2 subject : Programming
 Book2 id: 5678
```

2.3. Nesting structure

When a structure contains another structure, it is called nested structure. For example, we have two structures named Address and Employee. To make Address nested to Employee, we have to define Address structure before and outside Employee structure and create an object of Address structure inside Employee structure.
Syntax for structure within structure or nested structure

```
struct structure1
{
       - - - - - - - - - -
       - - - - - - - - - -
};

struct structure2
{
       - - - - - - - - - -
       - - - - - - - - - -
```

```
                        struct structure1 obj;
                };
```

Example 2-3 Nested structures

Code

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

struct movies{
      string title;
      int year;
};

struct friends{
      string name;
      string email;
      movies fav_mov;
}frnd1, frnd2;

friends *pfriends = &frnd2;

int main()
{
      frnd1.name = "Charlie";
      frnd1.fav_mov.title = "Harry Potter";
      frnd1.fav_mov.year = 2001;

      cout<<"Name: "<<frnd1.name;
      cout<<endl<<"Fav mov: "<<frnd1.fav_mov.title<<endl;
      cout<<"Movie year: "<<frnd1.fav_mov.year<<endl<<endl;

      pfriends->name = "Maria";
      pfriends->fav_mov.title = "Tangled";
      pfriends->fav_mov.year = 2010;
      pfriends->email = "abc@gmail.com";

      cout<<"Name: "<<pfriends->name;
      cout<<endl<<"Fav mov: "<<     pfriends->fav_mov.title  <<endl;
      cout<<"Movie Year: "<<pfriends->fav_mov.year <<endl;
      cout<<"Email: "<<pfriends->email<<endl;

}
```

Output

```
Name: Charlie
Fav mov: Harry Potter
Movie year: 2001

Name: Maria
Fav mov: Tangled
Movie Year: 2010
Email: abc@gmail.com
```

2.4. Pointers to Structure

As we have learnt a memory location can be accessed by a pointer variable. In the similar way a structure is also accessed by its pointer. The syntax for structure pointer is same as for the ordinary pointer variable. In general, the structure pointer is defined by the statement
struct-type *sptr;
Where **struct-type** refers to structure type specifier, and **sptr** is a variable that points to the structure. It must be ensured that the pointer definition must preceed the structure declaration. For example, a pointer to struct employee may be defined by the statement
struct employee *sptr;
In other words, the variable **sptr** can hold the address value for a structure of type **struct** employee.

We can create several pointers using a single declaration, as follows:

```
struct employee *sptr1, *sptr2, *sptr3;

We have a structure:
struct employee{
char name[30];
int  age;
float salary;
};
```

We define its pointer and variable as following:

```
struct    employee *sptr1, emp1;
```

A pointer to a structure must be initialized before it can be used anywhere in program. The address of a structure variable can be obtained by applying the address operator & to the variable. For example, the statement

```
sptr1 = &emp1;
```

2.4.1. Accessing Structure Members Using Arrow Operator

The members of a structure can be accessed using an arrow operator. The arrow operator -> (consisting of minus sign (-) followed by the greater than (>) symbol). Using the arrow operator, a structure member can be accessed by the expression

```
sptr1->member-name
```

Where sptr holds the address of a structure variable, and member-name is the name of a member belonging to the structure. For example, the values held by the members belonging to the structure emp1 are given by the expression:

```
sptr1->name
sptr1->age
sptr1->salary
```

Example 2-4 Pointer to structures

Code

```
// pointers to structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
  string title;
  int year;
};

int main ()
{
  string mystr;

  movies_t amovie;
  movies_t * pmovie;
  pmovie = &amovie;

  cout << "Enter title: ";
  getline (cin, pmovie->title);
  cout << "Enter year: ";
  getline (cin, mystr);
  (stringstream) mystr >> pmovie->year;

  cout << "\nYou have entered:\n";
  cout << pmovie->title;
  cout << " (" << pmovie->year << ")\n";

  return 0;
}
```

Output

```
Enter title: Invasion of the body snatchers
Enter year: 1978

You have entered:
Invasion of the body snatchers (1978)
```

2.5. Example Problems

Example 2-5 Structures Example Problem

Problem Statement

GIKI needs a student management system. A student has a name and registration number. Design a C++ program that can store a list of students and then display students' details on console.

Your program must:

Use struct for storing student details.

Have a function to take the students' data from the user

Have a function to display the students' data on console screen

Solution

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

struct Student{
	string name;
	int regNo;
};

void initialize(Student students[], int SIZE);
void display(Student students[], int SIZE);

int main()
{
	const int SIZE = 3;
	Student students[SIZE];
	initialize(students, SIZE);
	display(students, SIZE);
	return 0;
}

void initialize(Student students[], int SIZE)
{
	cout<<"Read Data"<<endl;
	for(int i=0; i < SIZE; i++)
	{
		cout<<"-------------------------------"<<endl;
		cout<<"\tStudent "<<i+1<<endl;
		cout<<"-------------------------------"<<endl<<endl;
		cout<<"Enter Reg. No.: ";
		cin>>students[i].regNo;
		cin.ignore();
		cout<<"Enter name: ";
		getline(cin, students[i].name);
		cout<<endl;
	}
}

void display(Student students[], int SIZE)
{
	cout<<"------------------------------"<<endl;
	cout<<"\tList of students "<<endl;
	cout<<"------------------------------"<<endl;
	cout<<"Reg. No\t\tName"<<endl;
	cout<<"------------------------------"<<endl;
	for(int i=0; i < SIZE; i++)
	{
		cout<<students[i].regNo<<"\t\t"<<students[i].name<<endl;
	}
}
```

Output

```
Read Data
------------------------------
        Student 1
------------------------------

Enter Reg. No.: 01
Enter name: Ali

------------------------------
        Student 2
------------------------------

Enter Reg. No.: 02
Enter name: Ahmad

------------------------------
        Student 3
------------------------------

Enter Reg. No.: 03
Enter name: Talha

------------------------------
        List of students
------------------------------
Reg. No         Name
------------------------------
1               Ali
2               Ahmad
3               Talha

 ------------------------------
```

Lab 3 **Linked Lists**

3.1. Introduction

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.

3.1.1. Why Linked List?

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

1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.

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

For example, in a system, if we maintain a sorted list of IDs in an array id[].

```
id[] = [1000, 1010, 1050, 2000, 2040]
```

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

3.1.2. Advantages over arrays

1) Dynamic size

2) Ease of insertion/deletion

3.1.3. Drawbacks

1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation. Read about it here.

2) Extra memory space for a pointer is required with each element of the list.

3) Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

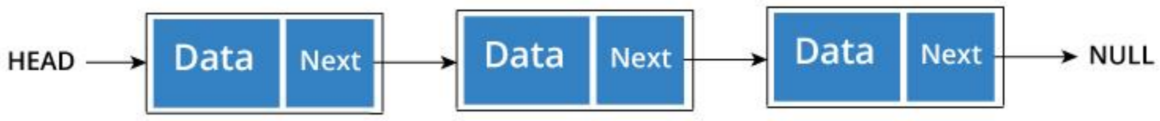
3.2. Types of Linked List

There are three common types of Linked List.

Singly Linked List
Doubly Linked List
Circular Linked List

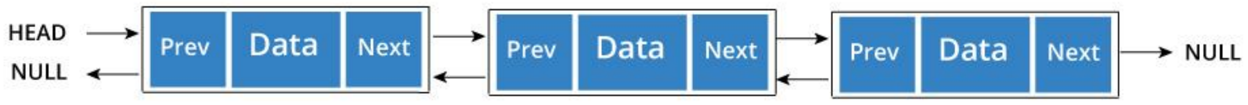
Singly Linked List

It is the most common. Each node has data and a pointer to the next node.



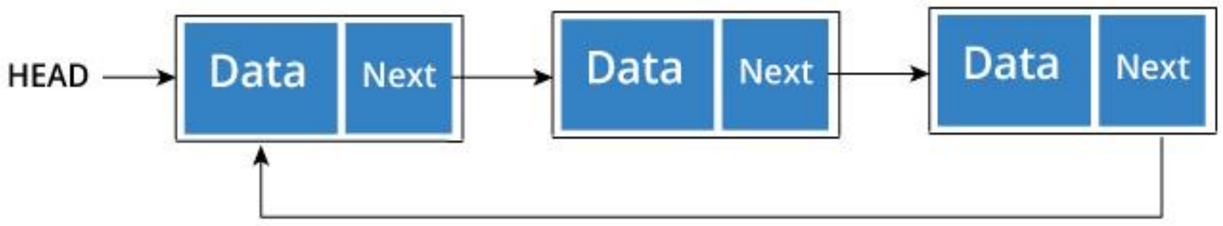
Doubly Linked List

We add a pointer to the previous node in a doubly linked list. Thus, we can go in either direction: forward or backward.



Circular Linked List

A circular linked list is a variation of linked list in which the last element is linked to the first element. This forms a circular loop.



A circular linked list can be either singly linked or doubly linked.

- for singly linked list, next pointer of last item points to the first item
- In doubly linked list, prev pointer of first item points to last item as well.

3.3. Single Linked List Operations

Now that you have got an understanding of the basic concepts behind linked list and their types, it’s time to dive into the common operations that can be performed.

Two important points to remember:

- head points to the first node of the linked list
- next pointer of last node is NULL, so if next of current node is NULL, we have reached end of linked list.

In all of the examples, we will assume that the linked list has three nodes 1 --->2 --->3 with node structure as below:

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

3.3.1. How to traverse a linked list

Displaying the contents of a linked list is very simple. We keep moving the temp node to the next one and display its contents.

When temp is NULL, we know that we have reached the end of linked list so we get out of the while loop.

```
struct node *temp = head;
printf("\n\nList elements are - \n");
while(temp != NULL)
{
     printf("%d --->",temp->data);
     temp = temp->next;
}
```

The output of this program will be:

```
List elements are -
1 --->2 --->3 --->
```

3.3.2. How to add elements to linked list

You can add elements to either beginning, middle or end of linked list.

Add to beginning

- Allocate memory for new node
- Store data

- Change next of new node to point to head
- Change head to point to recently created node

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = head;
head = newNode;
```

Add to end

- Allocate memory for new node
- Store data
- Traverse to last node
- Change next of last node to recently created node

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = NULL;
struct node *temp = head;
while(temp->next != NULL){
  temp = temp->next;
}
temp->next = newNode;
```

Add to middle

- Allocate memory and store data for new node
- Traverse to node just before the required position of new node
- Change next pointers to include new node in between

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
struct node *temp = head;
for(int i=2; i < position; i++) {
    if(temp->next != NULL) {
        temp = temp->next;
    }
}
newNode->next = temp->next;
temp->next = newNode;
```

3.3.3. How to delete from a linked list

You can delete either from beginning, end or from a position.

Delete from beginning

- Point head to the second node

```
head = head->next;
```

Delete from end

- Traverse to second last element
- Change its next pointer to null

```
struct node* temp = head;
while(temp->next->next!=NULL){
  temp = temp->next;
}
temp->next = NULL;
```

Delete from middle

- Traverse to element before the element to be deleted
- Change next pointers to exclude the node from the chain

```
for(int i=2; i< position; i++) {
    if(temp->next!=NULL) {
        temp = temp->next;
    }
}
temp->next = temp->next->next;
```

3.3.4. Complete program for linked list operations

Here is the complete program for all the linked list operations we learnt till now. Lots of edge cases have been left out to make the program short.

Example 3-1 Linked Lists Implementation

Code

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

struct Node{
      int value;
      Node *next;

      Node(){
            this->next = NULL;
      }
};

class SingleLinkedList{
      private:
            Node *head;

      public:
            SingleLinkedList(){   //initially set the empty list
parameters
                  head = new Node;
            }


            void printList(){

                  cout<<endl<<"Linked List"<<endl<<endl;
                  cout<<"[head] -> ";
                  for(Node *tmp=head->next; tmp != NULL; tmp = tmp->next)
                  {
                        cout<<"["<<tmp->value<<"]"<<" -> ";
                  }
                  cout<<"null"<<endl<<endl;
            }


            void addStart(int v){

                  Node *tmp = new Node;   //create new node to be added
                  tmp->value = v;         //assign a value

                  tmp->next = head->next; //new node next pointer (tmp)
is now pointing to the same as head
                  head->next = tmp;      //head next pointer is updated
to a new node

            }

            void addEnd(int v){

                  Node *tmp = new Node;   //create new node
                  tmp->value = v;         //set the value

                  Node *n = new Node;
                  for(n = head; n->next != NULL ; n = n->next);
                  n->next = tmp;
                  tmp->next = NULL;
            }

            void addAfter(Node *prevNode, int val){

                  Node *tmp = new Node;
                  tmp->value =  val;

                  tmp->next=prevNode->next;
                  prevNode->next = tmp;

            }


            Node* find(int v, bool findPrev = false){

                  //pointer to the node that we wana find
                  Node *prevNode = NULL;
                  bool found = false;

                  for(prevNode = head; prevNode->next != NULL ; prevNode
= prevNode->next){
```

```
                        //if value is found
                        if(prevNode->next->value == v){
                              found = true;
                              break;
                        }

                  }

                  if(found){
                        if(findPrev)
                              return prevNode;
                        else
                              return prevNode->next;
                  }
                  else
                        return NULL;

            }

      bool remove(int value){

            Node *prevN;
            Node *tmp;

            //check for value existance
            if(find(value) != NULL){

                  //find previous node
                  prevN = find(value,true);

                  tmp = prevN->next;
                  prevN->next = prevN->next->next;
                  delete tmp;
                  return true;
            }
            else
            return false;
      }




};
int main(){
      SingleLinkedList list;
      int choice;
      int value;
      Node* node;
      bool again = true;
      while(again)
      {
            cout<<"========== Menu =========="<<endl;
            cout<<"0. Exit"<<endl;
            cout<<"1. Display"<<endl;
            cout<<"2. addStart"<<endl;
            cout<<"3. addEnd"<<endl;
            cout<<"4. addAfter"<<endl;
            cout<<"5. Find Value"<<endl;
      cout<<"6. Remove Value"<<endl;
            cout<<"Enter choice: ";
            cin>>choice;

            switch(choice)
            {
                  case 0:  //exit loop
                        again = false;
                        break;
                  case 1: // display list
                        list.printList();
                        break;

                  case 2: // add start
                        cout<<"Enter the value that you want to add: ";
                        cin>>value;
                        list.addStart(value);
                        break;

                  case 3: // add end
                        cout<<"Enter the value that you want to add: ";
                        cin>>value;
                        list.addEnd(value);
                        break;

                  case 4: // add after


                        cout<<"Enter the value after which you want to
add: ";
                        cin>>value;
                        node = list.find(value); // find previous pointer

                        if(node == NULL)
                        {
                              cout<<"Value "<<value<<" not found!"<<endl;
                        }else{
                              cout<<"Enter the value that you want to
enter after "<<value<<": ";
                              cin>>value;
                              list.addAfter(node, value);
                        }
                        break;

                  case 5: // find value
                        cout<<"Enter the value that you want to find: ";
                        cin>>value;

                        if(list.find(value) != NULL)
                        {
                              cout<<"Value exists in the list"<<endl;
                        }else{
                              cout<<"Value does not exist in the
list"<<endl;
                        }
                        break;

                  case 6: // remove
                        cout<<"Enter the value that you want to remove:
";
                        cin>>value;

                        if(list.remove(value))
                        {
                              cout<<value<<" removed successfully"<<endl;
                        }else{
                              cout<<value<<" does not exist in the
list"<<endl;
                        }
                        break;
            }
            system("pause");
            system("CLS");
      }
}
```

Output

```
 ========== Menu ==========
 0. Exit
 1. Display
 2. addStart
 3. addEnd
 4. addAfter
 5. Find Value
 6. Remove Value
 Enter choice: 1

 Linked List

 [head] -> null
```

3.4. Example Problem

Example 3-2 Problem, Doubly Linked List Implementation

Problem Statement

Implement doubly linked list

Solution

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

struct Node
{
      int value;
      Node *next, *previous;  //now we have 2 pointers

      Node()
      {
            this->next = NULL;
            this->previous = NULL;
      }
};

class doublyLinkedList
{
      private:
            Node *head;
            Node *tail;


      public:
            //initially set the empty list parameters
            doublyLinkedList()
            {
                  head = new Node;
                  tail = NULL;
            }

            void printList()
            {
                  cout<<endl<<"Linked List"<<endl<<endl;
                  cout<<"[head] <-> ";
                  for(Node *tmp=head->next; tmp != NULL; tmp = tmp->next)
                  {
                        cout<<"["<<tmp->value<<"]"<<" <-> ";
                  }
                  cout<<"null"<<endl<<endl;
            }


            void addStart(int v)
            {
                  Node *tmp = new Node;   //create new node to be added
                  tmp->value    = v;         //assign a value
                  tmp->previous = head;   // pointing to head at
previously

                  if(tail == NULL)
                  {
                        tail = tmp;
                  }
                  tmp->next = head->next; //new node next pointer (tmp)
is now pointing to the same as head
                  head->next = tmp;      //head next pointer is updated
to a new node

            }

            void addEnd(int v)
            {
                  Node *tmp = new Node;   //create new node
                  tmp->value = v;         //set the value
                  tmp->previous = tail;   //tail is now the 2nd last node
so previously pointing to tail

                  if(tail == NULL)        //checking for the empty list
                  {
                        tail = tmp;          //new node is also the tail
in case of only one node in the list
                        head->next = tmp;
                        tmp->previous = head;    //new node is previously
is pointing towards head
                  }

                  else
                  {
                        tail->next = tmp;  //tail next pointer is updated
now which was null
                        tail = tail->next;  //set the new tail to tmp
i.e. new node
                  }

            }
            void addAfter(Node *prevNode, int val)
            {
                  Node *tmp = new Node;
                  tmp->value =  val;

                  //check for the end node
                  if(prevNode->next == NULL)
                  {
                        prevNode->next = tmp;
                        tmp->next = NULL;
                        tail = tmp;
                  }

                  else
                  {
                        prevNode->next->previous = tmp;
                        tmp->next=prevNode->next;
                        prevNode->next = tmp;
                  }
                  tmp->previous = prevNode;
            }

            Node* find(int v){

                  //pointer to the node that we wana find
                  Node *tmp = NULL;
                  bool found = false;

                  for(tmp = head->next; tmp != NULL ; tmp = tmp->next){

                        //if value is found
                        if(tmp->value == v){
                              found = true;
                              break;
                        }
                  }

                  if(found)
                        return tmp;
                  else
                        return NULL;

            }




            bool remove(int value){

            Node *tmp ;   //tmp node for traversing through the list
            Node *delNode;  //node to be deleted
            delNode = find(value);   //find the pointer of the deleted
node
```

```
             //check for value existance
            if( delNode != NULL){

                  for(tmp = head; tmp != NULL; tmp = tmp->next){
//traversing through the list

                        //if we 've reached to the previous node
                        if(tmp == delNode->previous){

                              //if tail is gonna delete
                              if(delNode->next == NULL){
                                    tmp->next = NULL;
                                    tail = tmp;
                              }

                              else {
                                    tmp->next = delNode->next;
                                    delNode->next->previous = tmp;

                              }

                        delete delNode;
                        break;
                        return true;
                        }
                  }
            }
            else
            return false;
      }





            void reversePrint(){

                        Node *tmp = new Node;
                        tmp = tail;
                        while(tmp != NULL){
                              cout<<tmp->value<<" ";
                              tmp = tmp->previous;
                        }
                        cout<<endl;
                  }
};
int main(){
      doublyLinkedList list;
      int choice;
      int value;
      Node* node;
      bool again = true;
      while(again)
      {
            cout<<"========== Menu =========="<<endl;
            cout<<"0. Exit"<<endl;
            cout<<"1. Display"<<endl;
            cout<<"2. addStart"<<endl;
            cout<<"3. addEnd"<<endl;
            cout<<"4. addAfter"<<endl;
            cout<<"5. Find Value"<<endl;
      cout<<"6. Remove Value"<<endl;
            cout<<"Enter choice: ";
            cin>>choice;

            switch(choice)
            {
                  case 0:  //exit loop
                        again = false;
                        break;
                  case 1: // display list
                        list.printList();
                        break;

                  case 2: // add start
                        cout<<"Enter the value that you want to add: ";
                        cin>>value;
                        list.addStart(value);
                        break;

                  case 3: // add end
                        cout<<"Enter the value that you want to add: ";
                        cin>>value;
                        list.addEnd(value);
                        break;

                  case 4: // add after


                        cout<<"Enter the value after which you want to
add: ";
                        cin>>value;
                        node = list.find(value); // find previous pointer

                        if(node == NULL)
                        {
                              cout<<"Value "<<value<<" not found!"<<endl;
                        }else{
                              cout<<"Enter the value that you want to
enter after "<<value<<": ";
                              cin>>value;
                              list.addAfter(node, value);
                        }
                        break;

                  case 5: // find value
                        cout<<"Enter the value that you want to find: ";
                        cin>>value;

                        if(list.find(value) != NULL)
                        {
                              cout<<"Value exists in the list"<<endl;
                        }else{
                              cout<<"Value does not exist in the
list"<<endl;
                        }
                        break;

                  case 6: // remove
                        cout<<"Enter the value that you want to remove:
";
```

```
                        cin>>value;

                        if(list.remove(value))
                        {
                              cout<<value<<" removed successfully"<<endl;
                        }else{
                              cout<<value<<" does not exist in the 
list"<<endl;
                        }
                        break;
            }
            system("pause");
            system("CLS");
      }
}
```

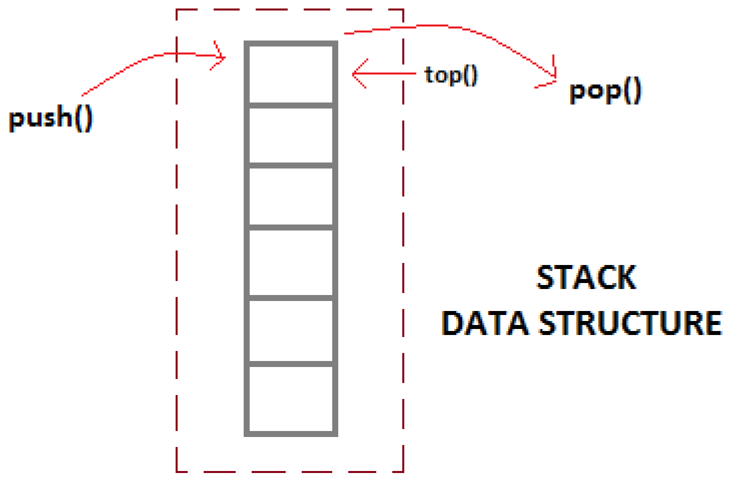
Output

```
 ========== Menu ==========
 0. Exit
 1. Display
 2. addStart
 3. addEnd
 4. addAfter
 5. Find Value
 6. Remove Value
 Enter choice: 1

 Linked List

 [head] -> null
```

Lab 4 Stacks

4.1. What is Stack Data Structure?

Stack is an abstract data type with a bounded(predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the **top** of the stack and the only element that can be removed is the element that is at the top of the stack, just like a pile of objects.



4.2. Basic features of Stack

1. Stack is an **ordered list** of **similar data type**.
2. Stack is a **LIFO**(Last in First out) structure or we can say **FILO**(First in Last out).
3. `push()` function is used to insert new elements into the Stack and `pop()` function is used to remove an element from the stack. Both insertion and removal are allowed at only one end of Stack called **Top**.
4. Stack is said to be in **Overflow** state when it is completely full and is said to be in **Underflow** state if it is completely empty.

4.3. Applications of Stack

The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.

There are other uses also like:

1. Parsing
2. Expression Conversion(Infix to Postfix, Postfix to Prefix etc)

4.4. Implementation of Stack Data Structure

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

4.4.1. Algorithm for PUSH operation

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

4.4.2. Algorithm for POP operation

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

Below we have a simple C++ program implementing stack data structure while following the object-oriented programming concepts.

Example 4-1 Stack Implementation

Code

```
/* Below program is written in C++ language */

# include<iostream>

using namespace std;

class Stack
{
    int top;
    public:
    int a[10];  //Maximum size of Stack
    Stack()
    {
        top = -1;
    }

    // declaring all the function
    void push(int x);
    int pop();
    void isEmpty();
};

// function to insert data into stack
void Stack::push(int x)
{
    if(top >= 10)
    {
        cout << "Stack Overflow \n";
    }
    else
    {
        a[++top] = x;
        cout << "Element Inserted \n";
    }
}

// function to remove data from the top of the stack
int Stack::pop()
{
    if(top < 0)
    {
        cout << "Stack Underflow \n";
        return 0;
    }
    else
    {
        int d = a[top--];
        return d;
    }
}

// function to check if stack is empty
void Stack::isEmpty()
{
    if(top < 0)
    {
        cout << "Stack is empty \n";
    }
    else
    {
        cout << "Stack is not empty \n";
    }
}

// main function
int main() {

    Stack s1;
    s1.push(10);
    s1.push(100);
    /*
        preform whatever operation you want on the stack
    */
}
```

Output

```
Element Inserted
Element Inserted
```

4.4.3. Analysis of Stack Operations

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

```
Push Operation: O(1)
Pop Operation: O(1)
Top Operation: O(1)
Search Operation: O(n)
```

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

4.5. Example Problem

Example 4-2 Stacks Example Problem

Problem Statement
Implement the stack using linked lists. The stack must have push, pop and top operations as follows: 1. Push operation must add a character in the stack 2. Pop operation must remove the last character that was pushed in the stack 3. Top operation must return the value of the last character that was inserted in the stack Please note that all operations (push, pop and top) must be implemented as separate functions (methods).
Solution

```
#include <iostream>
using namespace std;
struct Node {
   int data;
   struct Node *next;
};

class Stack{
      public:
      Node *top;
      int size;

      Stack()
      {
            this->top  = NULL;
            this->size = 0;
      }

      void push(int val) {
         struct Node* newnode = new Node();
         newnode->data = val;
         newnode->next = this->top;
         this->top = newnode;
      }

      void pop() {
         if(this->top==NULL)
            cout<<"Stack is empty"<<endl;
         else {
            cout<<"The popped element is "<< this->top->data <<endl;
            this->top = this->top->next;
         }
      }

      void display() {
         struct Node* ptr;
         if(this->top==NULL)
            cout<<"stack is empty";
         else {
            ptr = this->top;
            cout<<"Stack elements are: ";
            while (ptr != NULL) {
               cout<< ptr->data <<" ";
               ptr = ptr->next;
            }
         }
         cout<<endl;
      }
};

int main() {
   int ch, val;
   Stack stack;
   cout<<"1) Push in stack"<<endl;
   cout<<"2) Pop from stack"<<endl;
   cout<<"3) Display stack"<<endl;
   cout<<"4) Exit"<<endl;
   do {
      cout<<"Enter choice: "<<endl;
      cin>>ch;
      switch(ch) {
         case 1: {
            cout<<"Enter value to be pushed:"<<endl;
            cin>>val;
            stack.push(val);
            break;
         }
         case 2: {
            stack.pop();
            break;
         }
         case 3: {
            stack.display();
            break;
         }
         case 4: {
            cout<<"Exit"<<endl;
            break;
         }
         default: {
            cout<<"Invalid Choice"<<endl;
         }
      }
   }while(ch!=4);
      return 0;
}
```

Output

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

```
Enter value to be pushed:
11
Enter choice:
3
Stack elements are: 11
Enter choice:
2
The popped element is 11
Enter choice:
3
stack is empty
Enter choice:
```

Lab 5 Queues

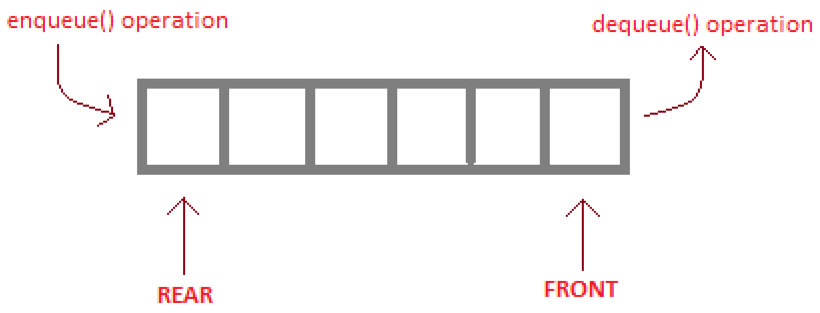
5.1. What is a Queue Data Structure?

Queue is also an abstract data type or a linear data structure, just like stack data structures, in which the first element is inserted from one end called the **REAR**(also called **tail**), and the removal of existing element takes place from the other end called as **FRONT**(also called **head**).

This makes queue as **FIFO**(First in First Out) data structure, which means that element inserted first will be removed first.

Which is exactly how queue system works in real world. If you go to a ticket counter to buy movie tickets, and are first in the queue, then you will be the first one to get the tickets. Right? Same is the case with Queue data structure. Data inserted first, will leave the queue first.

The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.



enqueue( ) is the operation for adding an element into Queue.

dequeue( ) is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE

5.2. Basic features of Queue

1. Like stack, queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO(First in First Out) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
4. `peek( )` function is oftenly used to return the value of first element without dequeuing it.

5.3. Applications of Queue

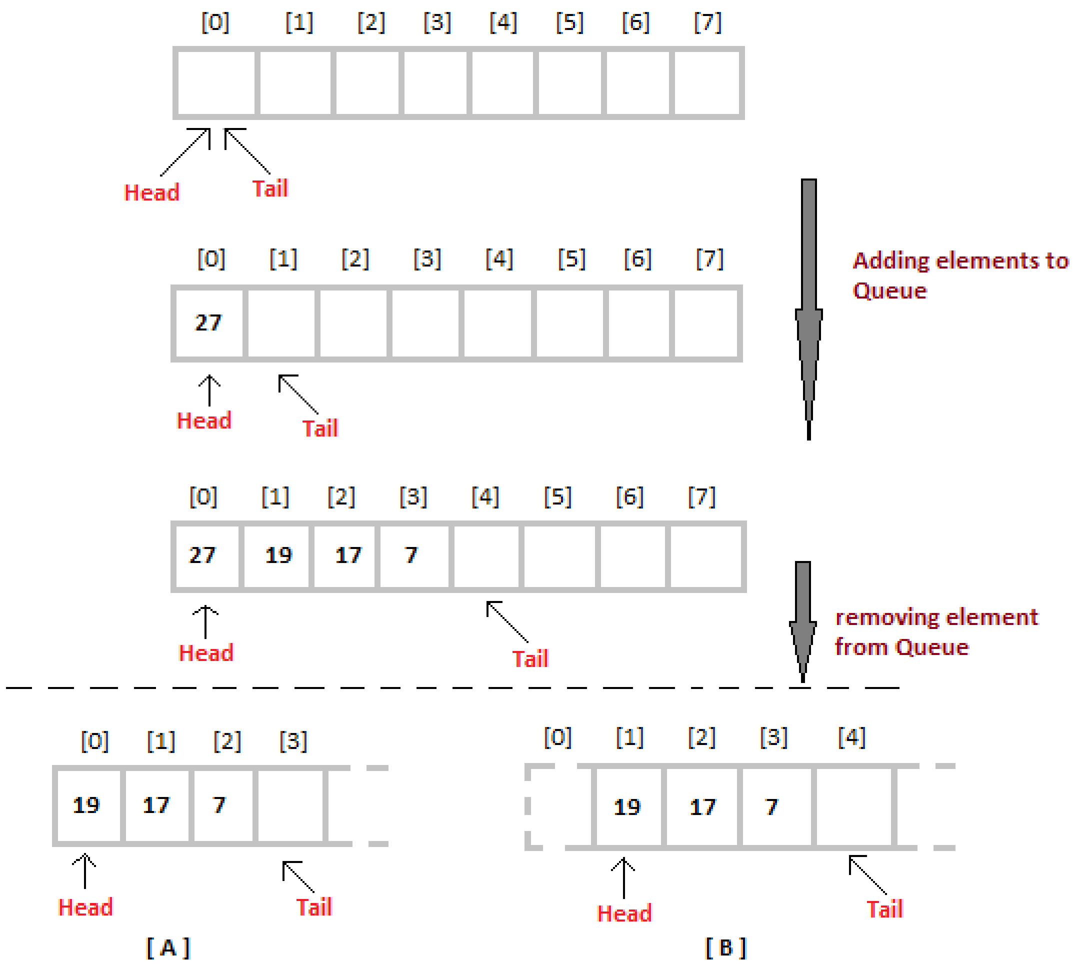
Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.

5.4. Implementation of Queue Data Structure

Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array.

Initially the **head**(FRONT) and the **tail**(REAR) of the queue points at the first index of the array (starting the index of array from `0`). As we add elements to the queue, the **tail** keeps on moving ahead, always pointing to the position where the next element will be inserted, while the **head** remains at the first index.



When we remove an element from Queue, we can follow two possible approaches (mentioned [A] and [B] in above diagram). In [A] approach, we remove the element at **head** position, and then one by one shift all the other elements in forward position.

In approach [B] we remove the element from **head** position and then move **head** to the next position.

In approach [A] there is an **overhead of shifting the elements one position forward** every time we remove the first element.

In approach [B] there is no such overhead, but whenever we move head one position ahead, after removal of first element, the **size on Queue is reduced by one space** each time.

5.4.1. Algorithm for ENQUEUE operation

1. Check if the queue is full or not.
2. If the queue is full, then print overflow error and exit the program.
3. If the queue is not full, then increment the tail and add the element.

5.4.2. Algorithm for DEQUEUE operation

1. Check if the queue is empty or not.
2. If the queue is empty, then print underflow error and exit the program.
3. If the queue is not empty, then print the element at the head and increment the head.

Code

```
/* Below program is written in C++ language */

#include<iostream>

using namespace std;

#define SIZE 10

class Queue
{
    int a[SIZE];
    int rear;   //same as tail
    int front;  //same as head

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

    //declaring enqueue, dequeue and display functions
    void enqueue(int x);
    int dequeue();
    void display();
};

// function enqueue - to add data to queue
void Queue :: enqueue(int x)
{
    if(front == -1) {
        front++;
    }
    if( rear == SIZE-1)
    {
        cout << "Queue is full";
    }
    else
    {
        a[++rear] = x;
    }
}

// function dequeue - to remove data from queue
int Queue :: dequeue()
{
    return a[++front];   // following approach [B], explained
above
}

// function to display the queue elements
void Queue :: display()
{
    int i;
    for( i = front; i <= rear; i++)
    {
        cout << a[i] << endl;
    }
}

// the main function
int main()
{
    Queue q;
    q.enqueue(10);
    q.enqueue(100);
    q.enqueue(1000);
    q.enqueue(1001);
    q.enqueue(1002);
    q.dequeue();
    q.enqueue(1003);
    q.dequeue();
    q.dequeue();
    q.enqueue(1004);
    q.display();

    return 0;
}
```

Output

```
1001
1002
1003
1004
```

To implement approach [A], you simply need to change the `dequeue` method, and include a `for` loop which will shift all the remaining elements by one position.

```
return a[0];        //returning first element
for (i = 0; i < tail-1; i++)        //shifting all other elements
{
    a[i] = a[i+1];
    tail--;
}
```

5.4.3. Complexity Analysis of Queue Operations

Just like Stack, in case of a Queue too, we know exactly, on which position new element will be added and from where an element will be removed, hence both these operations requires a single step.

```
Enqueue: O(1)
Dequeue: O(1)
Size: O(1)
```

5.5. Example Problem

Example 5-1 Queues Example Problem

Problem Statement

Implement the Queue. The queue must have enqueue, dequeue, display and front (read top) operations as follows:

1. Enqueue operation must add a string in the queue
2. Dequeue operation must remove the first string that was enqueued
3. Front operation must return the first string that was enqueued
4. Display operation must show all strings of the queue

Please note that all operations (Enqueue, Dequeue and Front) must be implemented as separate functions (methods).

Solution

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

using namespace std;

struct strng{
	string d;
	struct strng *next;
};

strng *input(){
	strng *data;
	data = new strng;

	cin.ignore();
	cout << "enter input : ";
	getline(cin,data->d);
	data->next = new strng;

	return data;
}

void enqueue(strng *entry,strng *bottom){
	entry=input();
	entry->next=bottom->next;
	bottom->next=entry;
}

void dequeue(strng *entry,strng *bottom,strng *top){
	strng *prev;
	prev = new strng;
	prev = bottom;

	for(entry=prev->next;entry!=NULL;entry=entry->next){
		if(entry->next==top){
			prev->next=entry->next;
			delete(entry);
			break;
		}
		prev=prev->next;
	}
}

void first(strng *entry,strng *bottom,strng *top){
	for(entry=bottom;entry!=NULL;entry=entry->next){
		if(entry->next==top){
			cout << entry->d;
			break;
		}
	}
}

void display(strng *bottom){
	if(bottom->next==NULL)return;
	display(bottom->next);
```

```
        cout << bottom->d << endl;
}

void menu(){
        cout << "\n\n\n\n\n";
        cout << "1. enqueue\n";
        cout << "2. dequeue\n";
        cout << "3. show top\n";
        cout << "4. show all\n";
        cout << "0. terminate\n";
}

void simulation(strng *top,strng *bottom,strng *entry){
        short int command;
        command=-1;

        do{
                menu();
                cin >> command;
                if(command==1){
                        system("CLS");
                        enqueue(entry,bottom);
                }else if(command==2){
                        system("CLS");
                        dequeue(entry,bottom,top);
                }else if(command==3){
                        system("CLS");
                        first(entry,bottom,top);
                }else if(command==4){
                        system("CLS");
                        display(bottom);
                }else if(command==0){
                        system("CLS");
                }else{
                        system("CLS");
                        cout << "incorrect input";
                }
        }while(command!=0);
}

int main()
{
        strng *top;
        top = new strng;
        top->next = NULL;

        strng *bottom;
        bottom = new strng;
        bottom->next = top;

        strng *entry;
        entry = new strng;

        simulation(top,bottom,entry);

        return 0;
}
```

Output

```
 1. enqueue
 2. dequeue
 3. show top
 4. show all
 0. terminate
```

Bibliography:

1. Data Structure Class Slides

Lab 6 **Sorting I**

6.1. Bubble Sort

Bubble sort is a sorting technique in which each pair of adjacent elements are compared, if they are in wrong order we swap them. This algorithm is named as bubble sort because, same as like bubbles the smaller or lighter elements comes up (at start) and bigger or heavier elements goes down (at end). Below I have shared a program for bubble sort in C++ which sorts a list of numbers in ascending order.

Example 6-1 Bubble Sort Implementation

Code

```
#include<iostream>

using namespace std;

int main()
{
	int a[50],n,i,j,temp;
	cout<<"Enter the size of array: ";
	cin>>n;
	cout<<"Enter the array elements: ";

	for(i=0;i<n;++i)
		cin>>a[i];

	for(i=1;i<n;++i)
	{
		for(j=0;j<(n-i);++j)
			if(a[j]>a[j+1])
			{
				temp=a[j];
				a[j]=a[j+1];
				a[j+1]=temp;
			}
	}

	cout<<"Array after bubble sort:";
	for(i=0;i<n;++i)
		cout<<" "<<a[i];

	return 0;
}
```

Output

```
Enter the size of array: 5
Enter the array elements: 4
7
1
0
8
Array after bubble sort: 0 1 4 7 8
```

6.1.1. Bubble Sort using linked list

Example 6-2 Bubble Sort Implementation using Linked Lists

Code

```
#include<iostream>
using namespace std;
struct node{
	int data;
	node *next;
} *head=NULL,*tail=NULL;
void putdata(int x)
{
	if(head==NULL)
	{
		node *newnode=new node;
		newnode->data=x;
		head=newnode;
		tail=newnode;
		head->next=NULL;
	}
	else
	{
		node *newnode=new node;
		newnode->data=x;
		tail->next=newnode;
		tail=newnode;
		tail->next=NULL;
	}
}
void sorting()
{
        int temp;
        node *current=head;
        node *traverse;
        while(current!=NULL)
        {
                traverse=head;
                while(traverse!=NULL)
                {
                        if(current->data<traverse->data)
                        {
                                temp=current->data;
                                current->data=traverse->data;
                                traverse->data=temp;

                        }
                        traverse=traverse->next;
                }
                current=current->next;
        }
}
void display()
{
        node *temp=head;
        while(temp!=NULL)
        {
                cout<<temp->data<<" ";
                temp=temp->next;
        }
}
int main()
{

        putdata(3);
        putdata(6);
        putdata(9);
        putdata(2);
        putdata(1);
        cout<<"before traversing"<<endl;
        display();
        cout<<"after traversing"<<endl;
        sorting();
        display();
```

```
}
```

Output

```
before traversing
3 6 9 2 1 after traversing
1 2 3 6 9
```

6.2. Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.
1) The subarray which is already sorted.
2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.
Following example explains the above steps:

```
arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]
// and place it at beginning
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 12 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 22 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 25 64
```

Example 6-3 Selection Sort Implementation

Code

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

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
        if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}

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

// Driver program to test above functions
int main()
{
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    cout << "Sorted array: \n";
    printArray(arr, n);
    return 0;
}
```

Output

```
Sorted array:
11 12 22 25 64
```

6.3. Insertion Sort

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands. in other words we sort cards using insertion sort mechanism. For this technique, we pick up one element from the data set and shift the data elements to make a place to insert back the picked up element into the data set.

6.3.1. The complexity of Insertion Sort Technique

- Time Complexity: O(n) for best case, O(n2) for average and worst case
- Space Complexity: O(1)

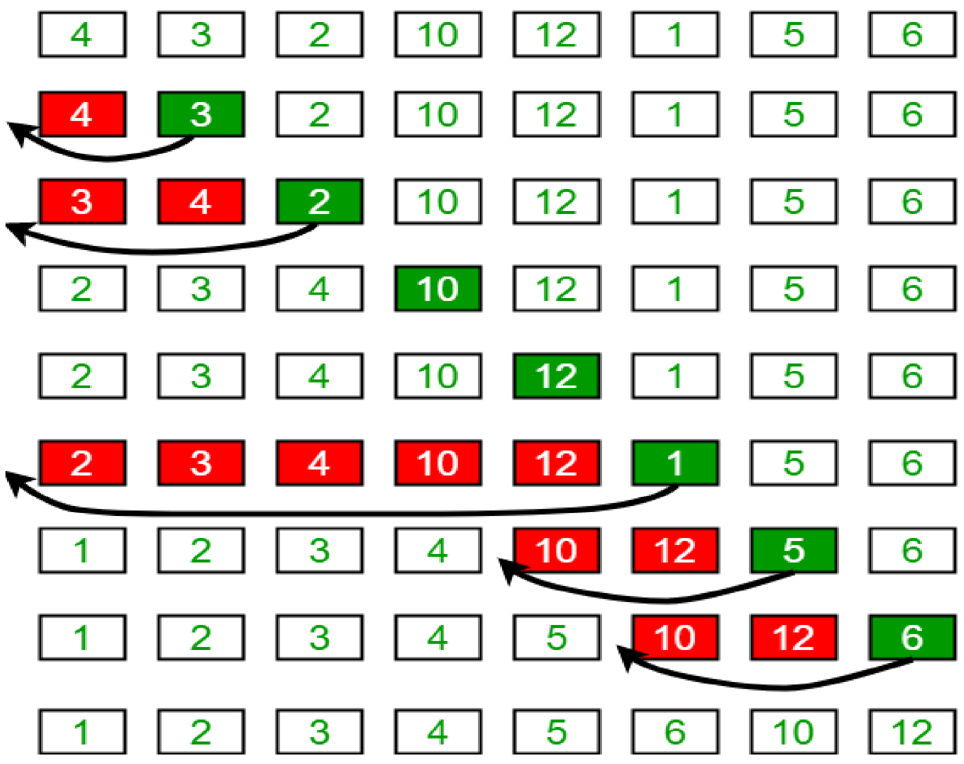
```
Input − The unsorted list: 9 45 23 71 80 55
Output − Array after Sorting: 9 23 45 55 71 80
```

6.3.2. Algorithm

```
insertionSort(array, size)
Input: An array of data, and the total number in the array
Output: The sorted Array
Begin
   for i := 1 to size-1 do
      key := array[i]
      j := i
      while j > 0 AND array[j-1] > key do
         array[j] := array[j-1];
         j := j – 1
      done
      array[j] := key
   done
End
```

Example:

Insertion Sort Execution Example



Example 6-4 Insertion Sort Implementation

Code

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

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

        /* Move elements of arr[0..i-1], that are
        greater than key, to one position ahead
        of their current position */
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

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

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

    insertionSort(arr, n);
    printArray(arr, n);

    return 0;
}
```

Output

```
 5 6 11 12 13
```

6.4. Example Problem

Example 6-5 Sorting I Example Problem

Problem Statement

Design a C++ program to maintain a list of books (using arrays). Your program must have the options to sort book by id and by name. You may use bubble sort or selection sort for sorting operations.

Solution

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

struct Book{
	int id;
	string name;
};

void printArray(Book A[], int N)
{
	cout<<"==========================="<<endl;
	for(int i=0; i<N; i++)
	{
		cout<<A[i].id<<" \t"<<A[i].name<<endl;
	}
	cout<<endl;
}

void swap(Book *a, Book *b){
	Book temp = *a;
	*a = *b;
	*b = temp;
}

void bubbleSortById(Book A[], int N)
{
	int n =N; // N is the size of the array;
	for (int i = 0; i < N; i++){

		int swapped = 0;

		for (int j = 1; j < n; j++) {

			if (A[j].id < A[j-1].id) {
				swap(&A[j-1] , &A[j]);
				swapped = 1;
			}//end if

		} //end inner for
		n = n-1;
		if (swapped == 0)
		{
			break;
		}
```

```
      } //end inner for
}

void bubbleSortByName(Book A[], int N)
{
      int n =N; // N is the size of the array;
      for (int i = 0; i < N; i++){

            int swapped = 0;

            for (int j = 1; j < n; j++) {

                  if (A[j].name < A[j-1].name) {
                        swap(&A[j-1] , &A[j]);
                        swapped = 1;
                  }//end if

            } //end inner for
            n = n-1;
            if (swapped == 0)
            {
                  break;
            }
      } //end inner for
}


int main()
{
      const int N = 5;
      Book A[N];

      A[0].id = 10;
      A[0].name = "Book D";

      A[1].id = 5;
      A[1].name = "Book E";

      A[2].id = 1;
      A[2].name = "Book B";

      A[3].id = 3;
      A[3].name = "Book C";

      A[4].id = 2;
      A[4].name = "Book A";

      cout<<"Before Bubble Sort by Id"<<endl;
      printArray(A, N);

      bubbleSortById(A, N);
      cout<<endl<<endl;
      cout<<"After Bubble Sort by Id"<<endl;
      printArray(A, N);

      bubbleSortByName(A, N);
      cout<<endl<<endl;
      cout<<"After Bubble Sort by Name"<<endl;
      printArray(A, N);
      return 0;
}
```

Output

```
10       Book D
5        Book E
1        Book B
3        Book C
2        Book A



After Bubble Sort by Id
===========================
1        Book B
2        Book A
3        Book C
5        Book E
10       Book D



After Bubble Sort by Name
===========================
2        Book A
1        Book B
3        Book C
10       Book D
5        Book E
```

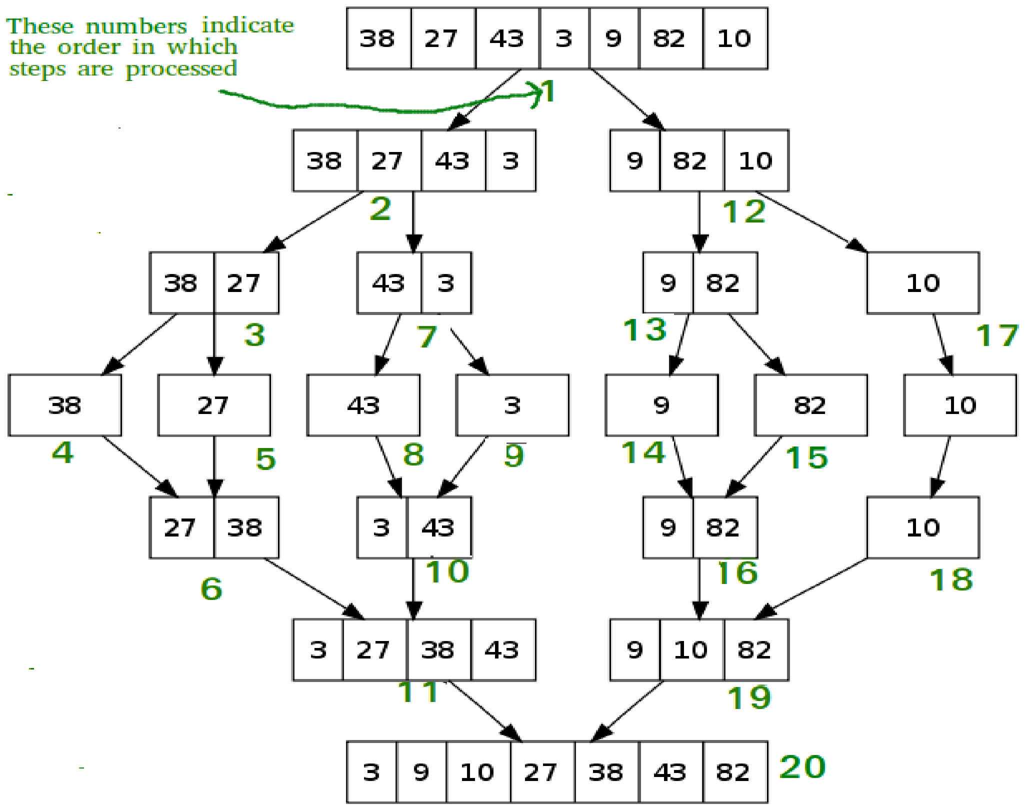
Lab 7 **Sorting II**

7.1. Merge Sort

Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = (l+r)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
```

The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.



Example 7-1 Merge Sort Implementation

Code

```
/* C++ program for Merge Sort */
#include<cstdlib>
#include<iostream>
using namespace std;

// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
      int i, j, k;
      int n1 = m - l + 1;
      int n2 = r - m;

      /* create temp arrays */
      int L[n1], R[n2];

      /* Copy data to temp arrays L[] and R[] */
      for (i = 0; i < n1; i++)
            L[i] = arr[l + i];
      for (j = 0; j < n2; j++)
            R[j] = arr[m + 1+ j];

      /* Merge the temp arrays back into arr[l..r]*/
      i = 0; // Initial index of first subarray
      j = 0; // Initial index of second subarray
      k = l; // Initial index of merged subarray
      while (i < n1 && j < n2)
      {
            if (L[i] <= R[j])
            {
                  arr[k] = L[i];
                  i++;
            }
            else
            {
                  arr[k] = R[j];
                  j++;
            }
            k++;
      }

      /* Copy the remaining elements of L[], if there
      are any */
      while (i < n1)
      {
            arr[k] = L[i];
            i++;
            k++;
      }

      /* Copy the remaining elements of R[], if there
      are any */
      while (j < n2)
      {
            arr[k] = R[j];
            j++;
            k++;
      }
}

/* l is for left index and r is right index of the
sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
      if (l < r)
      {
            // Same as (l+r)/2, but avoids overflow for
            // large l and h
            int m = l+(r-l)/2;

            // Sort first and second halves
            mergeSort(arr, l, m);
            mergeSort(arr, m+1, r);

            merge(arr, l, m, r);
      }
}

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

      cout<<endl;
}
```

```
/* Driver program to test above functions */
int main()
{
	int arr[] = {12, 11, 13, 5, 6, 7};
	int arr_size = sizeof(arr)/sizeof(arr[0]);

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

	mergeSort(arr, 0, arr_size - 1);

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

Output

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

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

7.1.1. Time Complexity:

Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

T(n) = 2T(n/2) + \Theta(n)

The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is \Theta(nLogn).

Time complexity of Merge Sort is \Theta(nLogn) in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and take linear time to merge two halves.

- Auxiliary Space: O(n)
- Algorithmic Paradigm: Divide and Conquer
- Sorting In Place: No in a typical implementation
- Stable: Yes

7.1.2. Applications of Merge Sort

- Merge Sort is useful for sorting linked lists in O(nLogn) time.In the case of linked lists, the case is different mainly due to the difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike an array, in the linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists. In arrays, we can do random access as elements are contiguous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i*4). Unlike arrays, we can not do random access in the linked list. Quick Sort requires a lot of this kind of access. In linked list to access i'th index, we have to travel each and every node from the head to i'th node as we don't have a continuous block of memory. Therefore, the overhead increases for quicksort. Merge sort accesses data sequentially and the need of random access is low.

- Inversion Count Problem
- Used in External Sorting

7.2. QuickSort

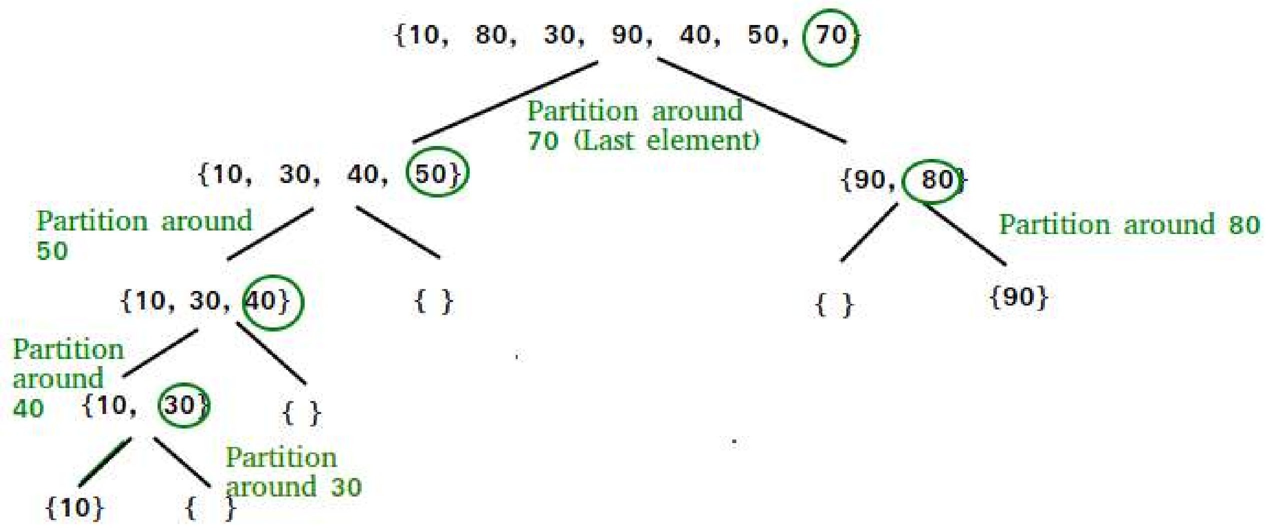
Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

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

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

7.2.1. Pseudo Code for recursive QuickSort function

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```



Partition Algorithm

There can be many ways to do partition, following pseudo code adopts the method given in CLRS book. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

7.2.2. Pseudo code for partition()

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

```
      i = (low - 1)  // Index of smaller element

      for (j = low; j <= high- 1; j++)
      {
          // If current element is smaller than or
          // equal to pivot
          if (arr[j] <= pivot)
          {
              i++;    // increment index of smaller element
              swap arr[i] and arr[j]
          }
      }
      swap arr[i + 1] and arr[high])
      return (i + 1)
  }
```

7.2.3. Illustration of partition()

```
arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes:  0   1   2   3   4   5   6

low = 0, high =  6, pivot = arr[h] = 70
Initialize index of smaller element, i = -1

Traverse elements from j = low to high-1
j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 0
arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j
                                     // are same

j = 1 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 1
arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

j = 3 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 2
arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped
j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
i = 3
arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

We come out of loop because j is now equal to high-1.
Finally we place pivot at correct position by swapping
arr[i+1] and arr[high] (or pivot)
arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

Now 70 is at its correct place. All elements smaller than
70 are before it and all elements greater than 70 are after
it.
```

Example 7-2 Quick Sort Implementation

Code

```
/* C++ implementation of QuickSort */
#include <bits/stdc++.h>
using namespace std;

// A utility function to swap two elements
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* This function takes last element as pivot, places
the pivot element at its correct position in sorted
array, and places all smaller (smaller than pivot)
to left of pivot and all greater elements to right
of pivot */
int partition (int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i = (low - 1); // Index of smaller element

    for (int j = low; j <= high - 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partition(arr, low, high);

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

/* Function to print an array */
```

```
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver Code
int main()
{
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, n - 1);
    cout << "Sorted array: \n";
    printArray(arr, n);
    return 0;
}
```

Output

```
Sorted array:
1 5 7 8 9 10
```

7.2.4. Analysis of QuickSort

Time taken by QuickSort in general can be written as following.

T(n) = T(k) + T(n-k-1) + \theta(n)

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.
The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

Worst Case: The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

T(n) = T(0) + T(n-1) + \theta(n)
which is equivalent to
T(n) = T(n-1) + \theta(n)

The solution of above recurrence is \theta(n2).

Best Case: The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

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

The solution of above recurrence is \theta(nLogn). It can be solved using case 2 of Master Theorem.

Average Case:
To do average case analysis, we need to consider all possible permutation of array and calculate time taken by every permutation which doesn't look easy.
We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case.

T(n) = T(n/9) + T(9n/10) + \theta(n)

Solution of above recurrence is also O(nLogn)

Although the worst case time complexity of QuickSort is O(n2) which is more than many other sorting algorithms like Merge Sort and Heap Sort, QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

7.3. Example Problem

Example 7-3 Sorting II Example Problem

Problem Statement

Design a C++ program to maintain a list of books (using Linked List). Your program must have the options to sort book by id and by name. The program must sort using merge sort or quick sort algorithm.

Solution

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

struct list{

	int id;
	string name;
	string getID();
```

```
};


void quickSort_id(list arr[], int low, int high)
{
      if (low < high)
      {

      int pivot = arr[high].id;
      int i = (low - 1);

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

            if (arr[j].id <= pivot)
            {
                  i++;
                  list t = arr[j];
                  arr[j] = arr[i];
                  arr[i] = t;
            }
      }

      list t = arr[i+1];
                  arr[i+1] = arr[high];
                  arr[high] = t;
      int pi = i + 1;

            quickSort_id(arr, low, pi - 1);
            quickSort_id(arr, pi + 1, high);
      }
}


void printArray(list arr[], int size)
{

      cout<<"Shelf\n\n";
      cout<<"ID | Name\n";
      cout<<"------------------------\n";
      for(int i = 0;i < size;i++){
            cout<<arr[i].id<<"     | ";
            cout<<arr[i].name;
            cout<<"\n------------------------\n";
      }

}


void quickSort_name(list arr[], int left, int right)
{


      if (left < right)
      {

      string val = arr[right].name;
        list temp;

        int j = right;
        int i = left - 1;
        while (true)
        {
            // cout<<"running"<<endl;
            while (arr[++i].name < val);

            while (arr[--j].name > val ){
                if(j == left)
                    break;
            }

            if(i >= j)
                break;

            temp=arr[i];
            arr[i]=arr[j];
            arr[j]=temp;
        }

        temp=arr[i];
        arr[i]=arr[right];
        arr[right]=temp;



      int pi = i;

            quickSort_name(arr, left, pi - 1);
            quickSort_name(arr, pi + 1, right);
      }
}


int main()
{
      list *arr = new list[6];


      arr[0].id=4;
      arr[1].id=7;
      arr[2].id=8;
      arr[3].id=9;
      arr[4].id=1;
      arr[5].id=5;

      arr[0].name = "Living To Tell";
      arr[1].name = "Eleven Minute";
      arr[2].name = "The Present";
      arr[3].name = "The Namesake";
      arr[4].name = "Go-Getter";
      arr[5].name = "Flash Boys";

      int n = 6;

      cout<<"\n***BEFORE SORT***\n"<<endl;
      printArray(arr,6);

      int option;
      cout<<"choose\n1) Sort by ID\n2)Sort by name\n";
      cin>>option;
      switch (option)
      {


      case 1: {
            quickSort_id(arr, 0, n-1);
          cout<<"\nSorted array by ID: \n"<<endl;
          printArray(arr, 6);
            break;
      }

      case 2: {
            quickSort_name(arr, 0, n-1);
            cout<<"\nSorted array by name: \n"<<endl;
            printArray(arr, 6);
            break;
      }



      }
}
```

Output

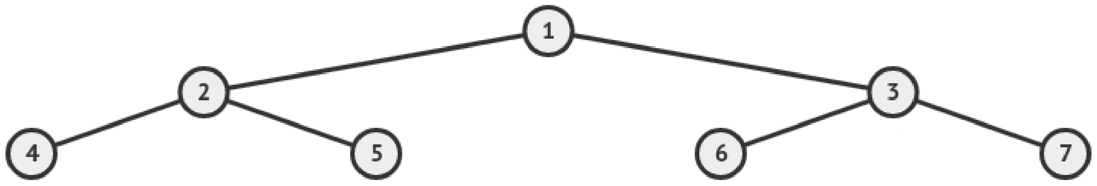
```
***BEFORE SORT***

Shelf

ID | Name
------------------------
4     | Living To Tell
------------------------
7     | Eleven Minute
------------------------
8     | The Present
------------------------
9     | The Namesake
------------------------
1     | Go-Getter
------------------------
5     | Flash Boys
------------------------
choose
1) Sort by ID
2)Sort by name
```

Lab 8 **Binary Search Trees**

8.1. Binary Tree

A **binary tree** is a hierarchical data structure whose behavior is similar to a tree, as it contains root and leaves (a node that has no child). The *root* of a binary tree is the topmost node. Each node can have at most two children, which are referred to as the *left child* and the *right child*. A node that has at least one child becomes a *parent* of its child. A node that has no child is a *leaf*. In this tutorial, you will be learning about the Binary tree data structures, its principles, and strategies in applying this data structures to various applications.

Take a look at the following binary tree:



From the preceding binary tree diagram, we can conclude the following:

- The root of the tree is the node of element **1** since it's the topmost node
- The children of element **1** are element **2** and element **3**
- The parent of elements **2** and **3** is **1**
- There are four leaves in the tree, and they are element **4**, element **5**, element **6**, and element **7** since they have no child

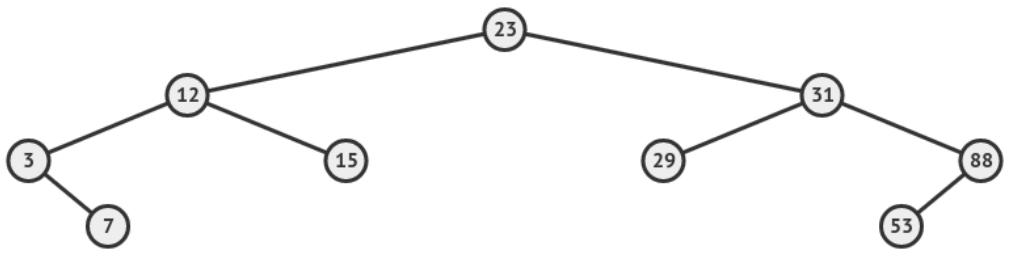
This hierarchical data structure is usually used to store information that forms a hierarchy, such as a file system of a computer.

8.2. Building a binary search tree ADT

A **binary search tree** (**BST**) is a sorted binary tree, where we can easily search for any key using the binary search algorithm. To sort the BST, it has to have the following properties:

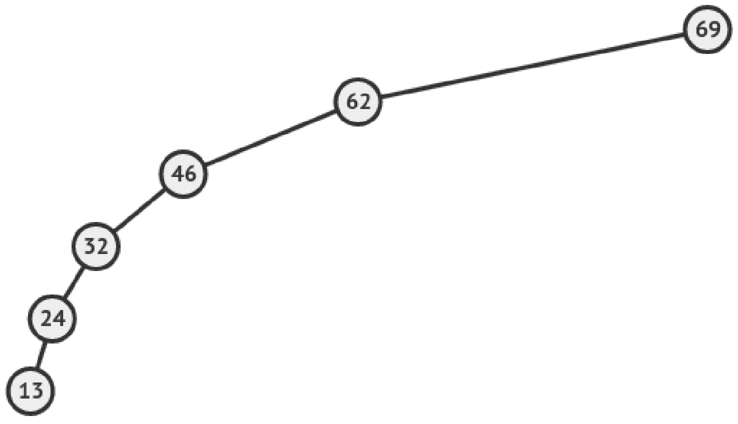
- The node's left subtree contains only a key that's smaller than the node's key
- The node's right subtree contains only a key that's greater than the node's key
- You cannot duplicate the node's key value

By having the preceding properties, we can easily search for a key value as well as find the maximum or minimum key value. Suppose we have the following BST:

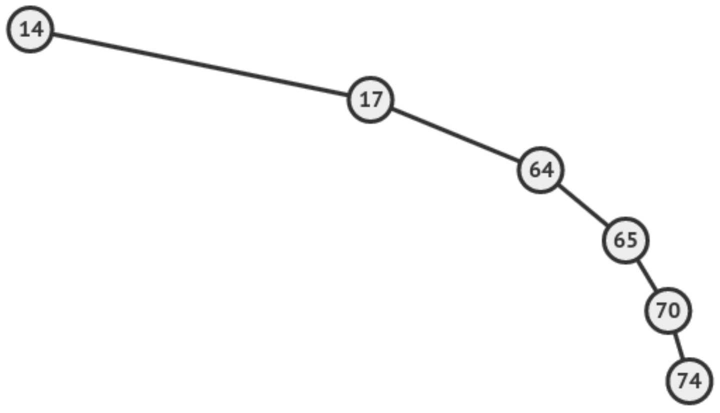


As we can see in the preceding tree diagram, it has been sorted since all of the keys in the root's left subtree are smaller than the root's key, and all of the keys in the root's right subtree are greater than the root's key. The preceding BST is a balanced BST since it has a balanced left and right subtree. We also can define the preceding BST as a balanced BST since both the left and right subtrees have an equal *height* (we are going to discuss this further in the upcoming section).

However, since we have to put the greater new key in the right subtree and the smaller new key in the left subtree, we might find an unbalanced BST, called a **skewed left** or a **skewed right BST**. Please see the following diagram:



The preceding image is a sample of a *skewed left BST*, since there's no right subtree. Also, we can find a BST that has no left subtree, which is called a *skewed right BST*, as shown in the following diagram:



As we can see in the two skewed BST diagrams, the height of the BST becomes taller since the height equals to *N – 1* (where *N* is the total keys in the BST), which is five. Comparing this with the balanced BST, the root's height is only three.

To create a BST in C++, we need to modify our `TreeNode` class in the preceding binary tree discussion, *Building a binary tree ADT*. We need to add the `Parent` properties so that we can track the parent of each node. It will make things easier for us when we traverse the tree. The class should be as follows:

```
class BSTNode
{
public:
    int Key;
    BSTNode * Left;
    BSTNode * Right;
    BSTNode * Parent;
};
```

There are several basic operations which BST usually has, and they are as follows:

- `Insert()` is used to add a new node to the current BST. If it's the first time we have added a node, the node we inserted will be a `root` node.
- `PrintTreeInOrder()` is used to print all of the keys in the BST, sorted from the smallest key to the greatest key.
- `Search()` is used to find a given key in the BST. If the key exists it returns `TRUE`, otherwise it returns `FALSE`.
- `FindMin()` and `FindMax()` are used to find the minimum key and the maximum key that exist in the BST.
- `Successor()` and `Predecessor()` are used to find the successor and predecessor of a given key. We are going to discuss these later in the upcoming section.
- `Remove()` is used to remove a given key from BST.

Now, let's discuss these BST operations further.

8.2.1. Inserting a new key into a BST

Inserting a key into the BST is actually adding a new node based on the behavior of the BST. Each time we want to insert a key, we have to compare it with the `root` node (if there's no root beforehand, the inserted key becomes a root) and check whether it's smaller or greater than the root's key. If the given key is greater than the currently selected node's key, then go to the right subtree. Otherwise, go to the left subtree if the given key is smaller than the currently selected node's key. Keep checking this until there's a node with no child so that we can add a new node there.

Insertion of elements into the BST is a reflection of the definition of BST. If there is no tree already, the insert creates a one node tree. Remember that each sub-tree of a node is a binary tree itself. If the tree is exists, then we proceed like this:

1. If the tree or (sub-tree) does not exist, create a node and insert the value in there.
2. If the value we are inserting is less than the root of the tree (or sub-tree) we move to the left sub-tree and start at step 1 again
3. If the value is greater than the root of the tree (or sub-tree) we move to the right sub-tree and start at step 1 again
4. If the value is equal to the root of the tree or (sub-tree) we do nothing because the value is already there! In this case we return.

```
struct treeNode* Insert( ElementType X, struct treeNode * T )
        {
if( T == NULL )
            {
                /* Create and return a one-node tree */
T = malloc( sizeof( struct treeNode ) );
if( T == NULL )
cout<< "Out of space!!!" ;
                else
                {
T->Element = X;
T->Left = T->Right = NULL;
                }
            }
            else
if( X < T->Element )
T->Left = Insert( X, T->Left );
            else
if( X > T->Element )
T->Right = Insert( X, T->Right );
            /* Else X is in the tree already; we'll do nothing */

                 return T;  /* Do not forget this line!! */
        }
```

The left and right are pointers to the corresponding children of the node.

8.2.2. Traversing a BST

There are three ways to traverse a binary search tree and a simple binary tree.

1. Pre-order traversal
2. In-order traversal
3. Post-order traversal

Pre-order traversal

In the pre-order traversal the parent node is visited before its children. In this type of traversal the parent node is visited followed by visiting its left sub-tree and right sub-tree. For example pre-order traversal of the tree given above will be:

8, 3, 1, 6, 4, 7, 10, 14, 13

The following recursive function implements pre-order traversal.

```
void preOrderTraverse (struct treeNode * root){
     if(root==NULL)
           return;
     cout<<root->data;
     preOrderTraverse (root->left);
     preOrderTraverse (root->right);
}
```

In-order traversal

In the in-order traversal, the left sub-tree is visited first, then the node itself is visited followed by visiting the right sub-tree. In-order traversal of the above tree will be like:

1, 3, 4, 6, 7, 8, 10, 13,14

The following function implements in-order traversal.

```
Void inOrderTraverse (struct treeNode * root){
     if(root==NULL)
           return;
     inOrderTraverse (root->left);
cout<<root->data;

     inOrderTraverse (root->right);
}
```

Post-order traversal

In the post-order traversal, first the left sub-tree of a node is visited, then the right sub-tree is visited and in the end the node itself is visited. Following is the post-order traversal of the above tree.

1, 4,7, 6, 3, 13, 14, 10, 8

The function implementing it is given below.

```
void postOrderTraverse (struct treeNode * root){
     if(root==NULL)
           return;
     postOrderTraverse (root->left);
     postOrderTraverse (root->right);
     cout<<root->data;
}
```

Example 8-1 Binary Search Tree

Code

```
// C program to demonstrate insert operation in binary search tree
#include<stdio.h>
#include<stdlib.h>

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

// A utility function to create a new BST node
struct node *newNode(int item)
{
    struct node *temp =  (struct node *)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to do inorder traversal of BST
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d \n", root->key);
        inorder(root->right);
    }
}

/* A utility function to insert a new node with given key in BST */
struct node* insert(struct node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);
    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left  = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}

// Driver Program to test above functions
int main()
{
    /* Let us create following BST
              50
           /     \
          30      70
         /  \    /  \
       20   40  60   80 */
    struct node *root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    // print inoder traversal of the BST
    inorder(root);

    return 0;
}
```

Output

```
20
30
40
50
60
70
80
```

8.3. Example Problem

Example 8-2 BST Example Problem

Problem Statement

Implement all basic operations (Insert, Traversing, Deletion) on BST using classes

Solution

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

```
struct TreeNode
{
      int value;
      TreeNode *left;
      TreeNode *right;
      TreeNode(){
            this->left = NULL;
            this->right =NULL;
      }
};
class BST
{
      private:
            TreeNode *root;

            //void destroySubTree(TreeNode *);

            void deleteNode(int num, TreeNode *&nodePtr)
            {
                  if(num <  nodePtr->value)
                        deleteNode(num, nodePtr->left);
                  else if(num > nodePtr->value)
                        deleteNode(num, nodePtr->right);
                  else
                        makeDeletion(nodePtr);
            }

            void makeDeletion(TreeNode *&nodePtr)
            {
                  TreeNode *tempNodePtr;

                  if(nodePtr == NULL)
                        cout<<"Cannot delete empty node.\n";

                  //in case of one child i.e. left
                  else if(nodePtr->right == NULL)
                  {
                        tempNodePtr = nodePtr;
                        nodePtr = nodePtr->left;
                        delete tempNodePtr;
                  }
                  //in case of one child i.e. right
                  else if(nodePtr->left == NULL)
                  {
                        tempNodePtr = nodePtr;
                        nodePtr = nodePtr->right;
                        delete tempNodePtr;
                  }
                  //if the node has two children.
                  else
                  {
                        tempNodePtr = nodePtr->right;
                        while(tempNodePtr->left != NULL)
                              tempNodePtr = tempNodePtr->left;

                        //reattach the left sub tree
                        tempNodePtr->left = nodePtr->left;

                        tempNodePtr = nodePtr;
                        //reattch the right subtree
                        nodePtr = nodePtr->right;
                        delete tempNodePtr;

                }
        }
        void displayInOrder(TreeNode *nodePtr)
        {
                if(nodePtr)
                {
                        displayInOrder(nodePtr->left);
                        cout<<nodePtr->value<<endl;
                        displayInOrder(nodePtr->right);
                }
        }
        void displayPreOrder(TreeNode *nodePtr)
        {
                if(nodePtr)
                {
                        cout<<nodePtr->value<<endl;
                        displayPreOrder(nodePtr->left);
                        displayPreOrder(nodePtr->right);
                }
        }


        void displayPostOrder(TreeNode *nodePtr)
        {
                if(nodePtr)
                {
                        displayPostOrder(nodePtr->left);
                        displayPostOrder(nodePtr->right);
                        cout<<nodePtr->value<<endl;
                }
        }


public:
        BST()
        {
                root = NULL;
        }
        ~BST()
        {
        //      destroySubTree(root);
        }

        void insertNode(int num)
        {
                TreeNode *newNode = new TreeNode; //Pointer to a new node
                TreeNode  *nodePtr;  //pointer to traverse the tree
                //create a new node


                newNode->value = num;
                if(root == NULL) //is the tree empty?
                {
                        root = newNode;
                }
                else
                {
                        nodePtr = root;
                        while(nodePtr != NULL)
                        {
                              if(num < nodePtr->value)
                              {
                                    if(nodePtr ->left != NULL)
                                          nodePtr = nodePtr->left;
                                    else
                                    {
                                          nodePtr->left = newNode;
                                          break;
                                    }
                              }
                              else if(num > nodePtr->value)
                              {
                                    if(nodePtr->right)
                                          nodePtr = nodePtr->right;
                                    else
                                    {
                                          nodePtr->right = newNode;
                                          break;
                                    }
                              }
                              else
                              {
                                    cout<<"Duplicate   value   found   in
tree.\n";
                                    break;
                              }
                        }
                  }
            }



            //Member function to search in the tree
            bool searchNode(int num)
            {
                  TreeNode *nodePtr = root;

                  while(nodePtr)
                  {
                        if(nodePtr->value == num)
                              return true;
                        else if(num < nodePtr->value)
                              nodePtr = nodePtr->left;
                        else
                              nodePtr = nodePtr->right;
                  }
                  return false;
            }

            //DELETION FUNCTION
            void remove(int num)
            {
                  deleteNode(num, root);
            }



            //TRAVERSING FUNCTIONS
            void showNodesInOrder(void){
                  displayInOrder(root);
            }
            void showNodesPreOrder(void){
                  displayPreOrder(root);
            }
            void showNodesPostOrder(void){
                  displayPostOrder(root);
            }

};

int main()
{
      BST tree;
      cout<<"Inseting nodes..";
      tree.insertNode(5);
      tree.insertNode(8);
      tree.insertNode(3);
      tree.insertNode(12);
      tree.insertNode(9);
      tree.insertNode(7);
      cout<<endl<<"Insertion done:";

       cout<<"InOrder traversal: \n";
      tree.showNodesInOrder();
      cout<<"\nPreOrder traversal:\n";
      tree.showNodesPreOrder();
      cout<<"\nPostorder traversal:\n";
      tree.showNodesPostOrder();

      if(tree.searchNode(3))
            cout<<"\n3 is found in the tree.\n";
      else
            cout<<"3 is not found in the tree.\n";

      cout<<"Deleting 8...\n";
      tree.remove(8);
      cout<<"Deleting 12...\n";
      tree.remove(12);

      cout<<"Now here are the nodes:\n";
      tree.showNodesInOrder();
}
```

Output

```
Inseting nodes..
Insertion done:InOrder traversal:
3
5
7
8
9
12

PreOrder traversal:
5
3
```

```
8
7
12
9

Postorder traversal:
3
7
9
12
8
5

3 is found in the tree.
Deleting 8...
Deleting 12...
Now here are the nodes:
3
5
7
9
```

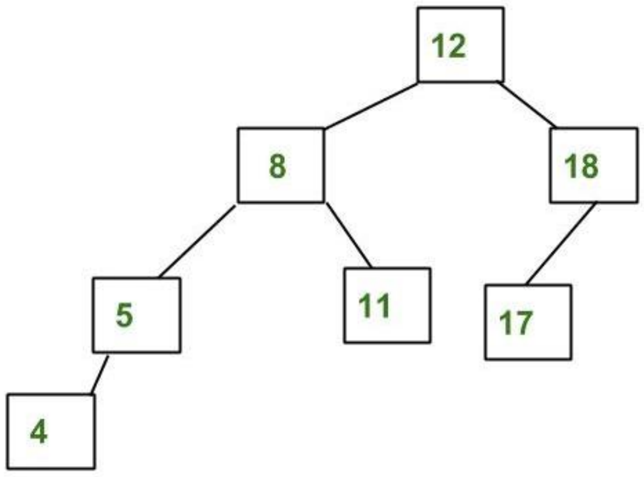
1. Data Structure Class Slides

Lab 9 **AVL Trees**

9.1. Introduction

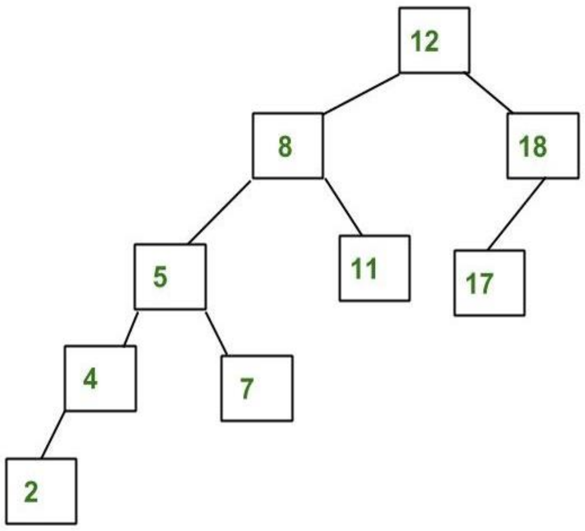
AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

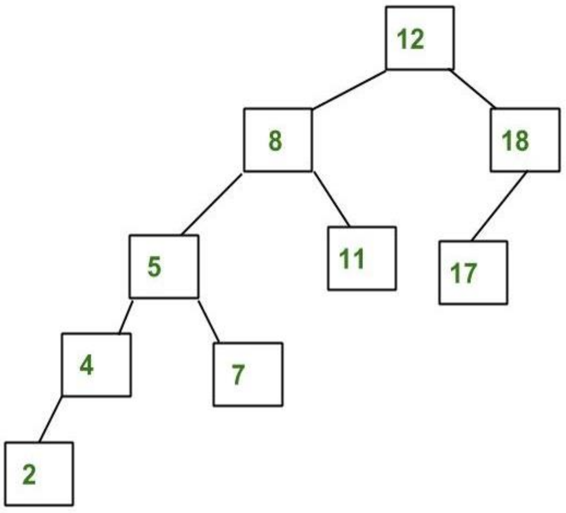
An Example Tree that is an AVL Tree



The above tree is AVL because differences between heights of left and right subtrees for every node is less than or equal to 1.

An Example Tree that is NOT an AVL Tree





The above tree is not AVL because differences between heights of left and right subtrees for 8 and 18 is greater than 1.

9.1.1. Why AVL Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree.

9.1.2. Insertion

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

1) Left Rotation

2) Right Rotation

```
T1, T2 and T3 are subtrees of the tree
rooted with y (on the left side) or x (on
the right side)
     y                               x
    / \     Right Rotation          /  \
   x   T3   - - - - - - - >        T1   y
  / \       < - - - - - - -            / \
 T1  T2     Left Rotation        T2  T3
Keys in both of the above trees follow the
following order
 keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)
So BST property is not violated anywhere.
```

Steps to follow for insertion

Let the newly inserted node be w

1) Perform standard BST insert for w.

2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.

3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways.

Following are the possible 4 arrangements:

a) y is left child of z and x is left child of y (Left Left Case)

b) y is left child of z and x is right child of y (Left Right Case)

c) y is right child of z and x is right child of y (Right Right Case)

d) y is right child of z and x is left child of y (Right Left Case)

Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to re-balance the subtree rooted with z and the complete tree becomes balanced as the height of subtree (After appropriate rotations) rooted with z becomes same as it was before insertion.

a) Left Left Case

```
T1, T2, T3 and T4 are subtrees.
       z                                      y
      / \                                  /      \
     y   T4      Right Rotate (z)        x          z
    / \          - - - - - - - - ->    /   \      /   \
   x   T3                             T1   T2   T3    T4
  / \
 T1   T2
```

b) Left Right Case

```
        z                               z                           x
       /   \                          /  \                         / \
     y      T4    Left Rotate (y)    x    T4    Right Rotate(z)   y     z
   /   \            - - - - ->      / \             - - - - - - ->  /\   /\
T1       x                         y    T3                     T1  T2 T3  T4
        /  \                      /   \
      T2    T3                  T1    T2
```

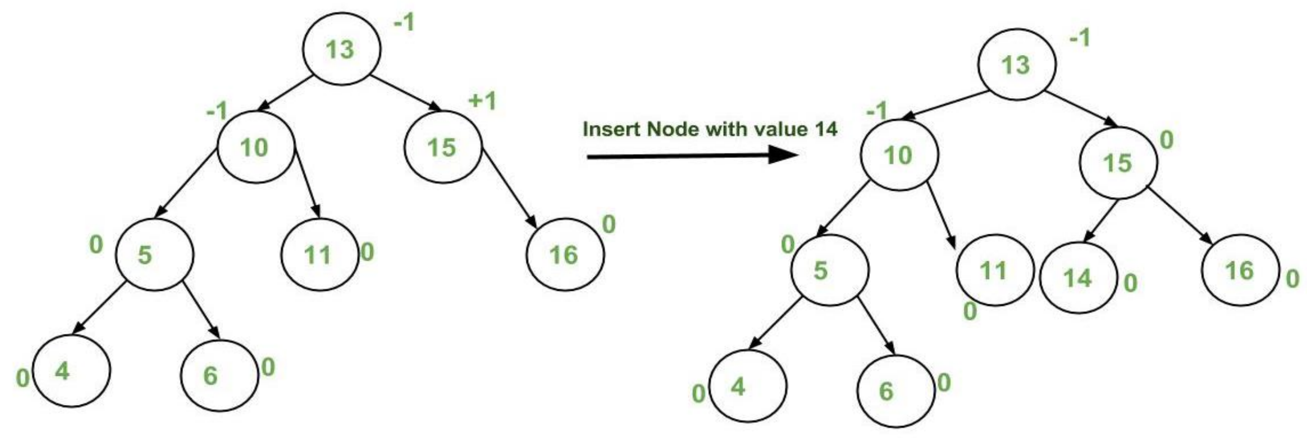
c) Right Right Case

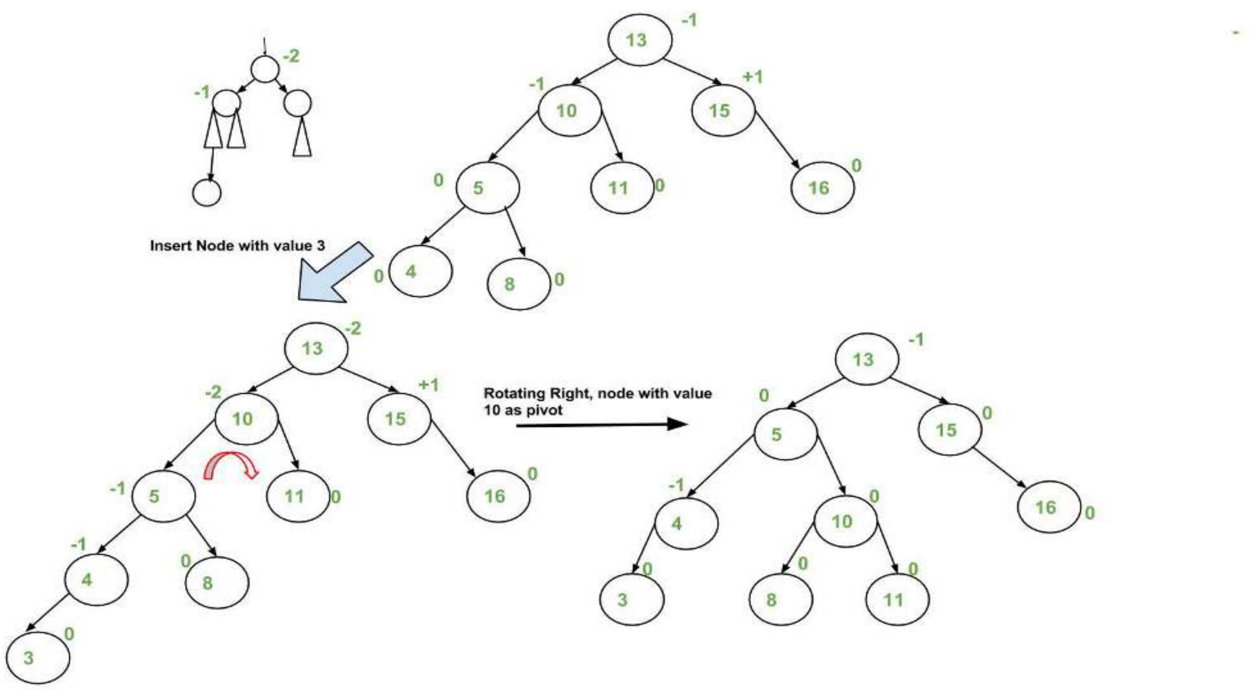
```
      z                                y
     / \                              /  \
   T1   y     Left Rotate(z)   z      x
       / \   - - - - - - - - ->      / \    / \
      T2   x                        T1  T2 T3  T4
          /  \
         T3  T4
```

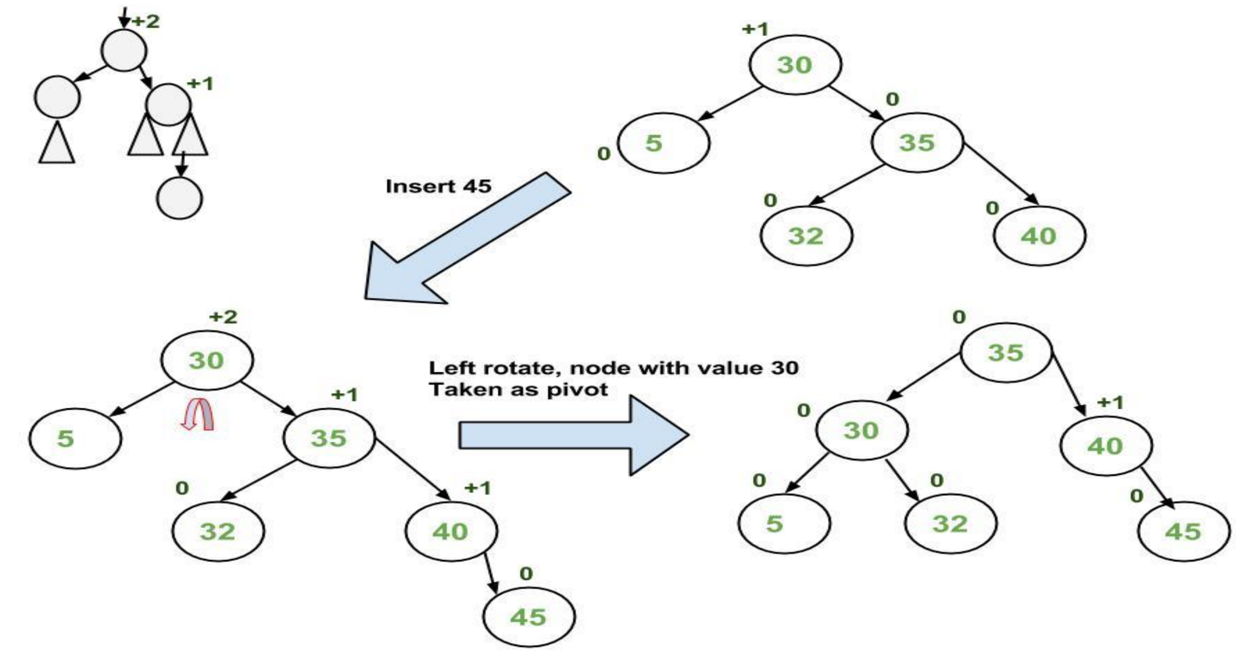
d) Right Left Case

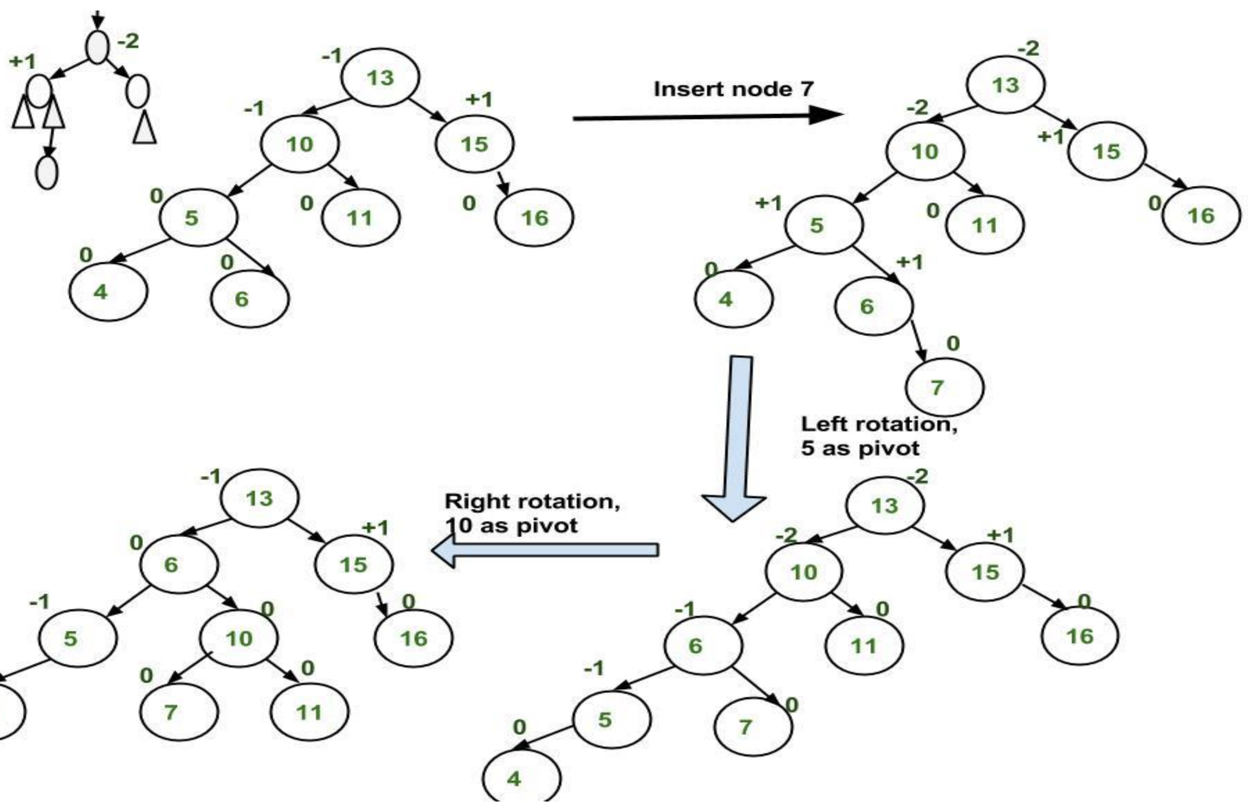
```
     z                              z                            x
    / \                            / \                          /  \
  T1   y      Right Rotate (y)    T1   x     Left Rotate(z)    z     y
   /    \     - - - - - - - - - ->    / \   - - - - - - - - - ->   / \   / \
  x       T4                        T2   y                    T1  T2  T3  T4
 / \                                    / \
T2   T3                               T3   T4
```

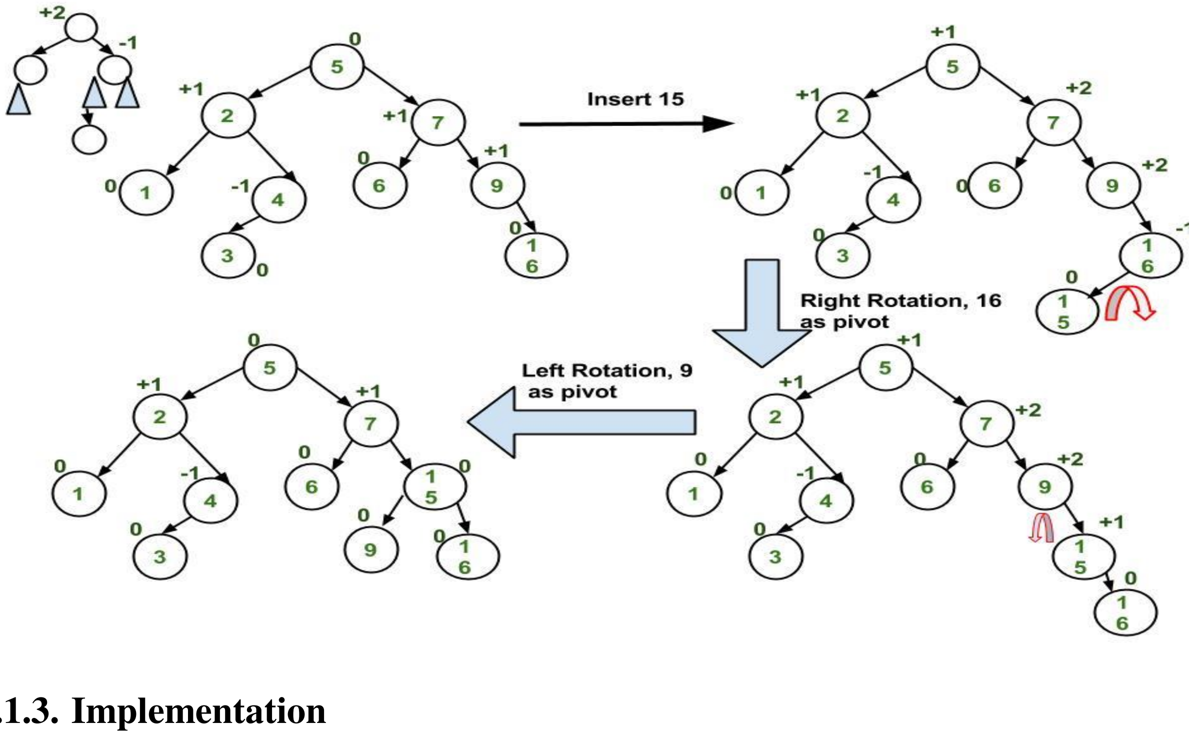
Insertion Examples:




Insert Node with value 3
Rotating Right, node with value 10 as pivot
13
10
15
5
11
16
4
8
3
-2
-1
+1
0


Insert 45
Left rotate, node with value 30 Taken as pivot
30
5
35
32
40
45
+2
+1
0





9.1.3. Implementation

Following is the implementation for AVL Tree Insertion. The following implementation uses the recursive BST insert to insert a new node. In the recursive BST insert, after insertion, we get pointers to all ancestors one by one in a bottom-up manner. So we don't need parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the newly inserted node.

1) Perform the normal BST insertion.

2) The current node must be one of the ancestors of the newly inserted node. Update the height of the current node.
3) Get the balance factor (left subtree height – right subtree height) of the current node.
4) If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or left Right case. To check whether it is left left case or not, compare the newly inserted key with the key in left subtree root.
5) If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right-Left case. To check whether it is Right Right case or not, compare the newly inserted key with the key in right subtree root.

Example 9-1 AVL Trees

Code

```
// C++ program to insert a node in AVL tree
#include<bits/stdc++.h>
using namespace std;

// An AVL tree node
class Node
{
    public:
    int key;
    Node *left;
    Node *right;
    int height;
};

// A utility function to get maximum
// of two integers
int max(int a, int b);

// A utility function to get the
// height of the tree
int height(Node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}

// A utility function to get maximum
// of two integers
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Helper function that allocates a
   new node with the given key and
   NULL left and right pointers. */
Node* newNode(int key)
{
    Node* node = new Node();
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // new node is initially
                      // added at leaf
    return(node);
}

// A utility function to right
// rotate subtree rooted with y
// See the diagram given above.
Node *rightRotate(Node *y)
{
    Node *x = y->left;
    Node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left),
                    height(y->right)) + 1;
    x->height = max(height(x->left),
                    height(x->right)) + 1;

    // Return new root
    return x;
}

// A utility function to left
// rotate subtree rooted with x
// See the diagram given above.
Node *leftRotate(Node *x)
{
    Node *y = x->right;
    Node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left),
                    height(x->right)) + 1;
    y->height = max(height(y->left),
                    height(y->right)) + 1;

    // Return new root
    return y;
}

// Get Balance factor of node N
int getBalance(Node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}
```

```
// Recursive function to insert a key
// in the subtree rooted with node and
// returns the new root of the subtree.
Node* insert(Node* node, int key)
{
    /* 1. Perform the normal BST insertion */
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Equal keys are not allowed in BST
        return node;

    /* 2. Update height of this ancestor node */
    node->height = 1 + max(height(node->left),
                        height(node->right));

    /* 3. Get the balance factor of this ancestor
        node to check whether this node became
        unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then
    // there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    /* return the (unchanged) node pointer */
    return node;
}

// A utility function to print preorder
// traversal of the tree.
// The function also prints height
// of every node
void preOrder(Node *root)
{
    if(root != NULL)
    {
        cout << root->key << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}

// Driver Code
int main()
{
    Node *root = NULL;

    /* Constructing tree given in
    the above figure */
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    /* The constructed AVL Tree would be
                30
            / \
            20 40
            / \ \
        10 25 50
    */
    cout << "Preorder traversal of the "
            "constructed AVL tree is \n";
    preOrder(root);

    return 0;
}
```

Output

```
Preorder traversal of the constructed AVL tree is
30 20 10 25 40 50
```

9.2. Example Problem

Example 9-2 AVL Trees Example Problem

Problem Statement

An object can have multiple attributes and methods. For example, a student may have registration number, name and CGPA, etc. However, we can build a tree on one of the attributes. You are required to implement AVL Tree to maintain the records of the students.

The tree must be developed on registration number attribute. In order traversal of an AVL tree shows the data in sorted order. Implement the in-order traversal.

Solution

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

struct student{
      int reg;
      string name;
      float gpa;
      student *left;
      student *right;
}*root;



void inorder(student *tree){
      if(tree==NULL){
            return;
      }
      inorder(tree->left);
      cout<<tree->reg<<",";
      inorder(tree->right);
}

int height(student *temp){
      int h=-1;
      if(temp != NULL){
            int l_height = height (temp->left);
            int r_height = height (temp->right);
            int max_height;
            if(l_height>r_height){
                  max_height = l_height;
            }
            else{
                  max_height = r_height;
            }
          h = max_height+1;
      }
      return h;
}

int diff(student *temp){
      int difference;
      int left;
      int right;

      left = height(temp->left);
      right = height(temp->right);
      difference = left-right;

      return difference;
}

student *ll_rotation(student *parent){
      cout<<"LL rotation"<<endl;
      student *temp;
      temp = parent->left;
      parent->left = temp->right;
      temp->right = parent;
      return temp;
}

student *rr_rotation(student *parent){
      cout<<"LL rotation"<<endl;
      student *temp;
      temp = parent->right;
      parent->right = temp->left;
      temp->left = parent;
      return temp;
}

student *rl_rotation(student *parent){
      cout<<"RL rotation"<<endl;
      student *temp;
      temp = parent->right;
      parent->right = ll_rotation(temp);
      return rr_rotation(parent);
}

student *lr_rotation(student *parent){
      cout<<"LR rotation"<<endl;
      student *temp;
      temp = parent->left;
      parent->left = rr_rotation(temp);
      return ll_rotation(parent);
}

student *balance(student *temp){
      int bal_factor = 0;
      bal_factor = diff(temp);

      cout<<"\nBalance factor: "<<bal_factor;

      if(bal_factor > 1){
            cout<<"\nBalancing..."<<endl;
            cout<<"Diff in balance at left is: "<<diff(temp->left)<<endl;

            if(diff(temp->left) > 0){
                  temp = ll_rotation(temp);
            }
            else{
                  temp = lr_rotation(temp);
            }
            return temp;
      }

      else if(bal_factor < -1){
            cout<<"\nBalancing..."<<endl;
            cout<<"Diff in balance at left is: "<<diff(temp->right)<<endl;

            if(diff(temp->right) > 0){
                  temp = rl_rotation(temp);
            }
            else{
                  temp = rr_rotation(temp);
            }
            return temp;
      }
      else{
            cout<<"\nThe tree is balance"<<endl;
            return temp;
      }

}

student *insert(student *root, int val){
      if(root==NULL){
            student *root = new student;
            root->reg = val;
            root->left = NULL;
            root->right = NULL;
            return root;
      }

      else{
            if(val < root->reg){
                  cout<<"\nInserting\n";
                  root->left = insert (root->left, val);
                  cout<<"Check Balance\n";
                  root = balance(root);
                  return root;
            }
            else{
                  cout<<"\nInserting\n";
                  root->right = insert (root->right, val);
                  cout<<"Check Balance\n";
                  root = balance(root);
                  return root;
            }
      }
}

void display(student *ptr, int level)
{
    int i;
    if (ptr!=NULL)
    {
        display(ptr->right, level + 1);
        printf("\n");
        if (ptr == root)
        cout<<"Root -> ";
        for (i = 0; i < level && ptr != root; i++)
            cout<<"        ";
        cout<<ptr->reg;
        display(ptr->left, level + 1);
    }
}

int main(){
      int choice;
      while(choice !=5){
            cout<<"\n1. Enter student in tree"<<endl;
            cout<<"2. Inorder Traversal of tree"<<endl;
            cout<<"3. Display"<<endl;
            cout<<"4. Height of tree"<<endl;
            cin>>choice;
            cout<<endl;

            switch(choice){
                  case 1:
                        int val;
                        cout<<"Enter   reg   number   of   student   to
insert"<<endl;
                        cin>>val;
                        root = insert(root, val);
                        break;

                  case 2:
                        cout<<"Inorder traversal is: "<<endl;
                        inorder(root);
                        cout<<endl;
                        break;

                  case 3:
                      display(root, 1);
                        break;

                  case 4:
                        int he = height(root);
                        cout<<"The height of tree is: "<<he<<endl;
                        break;
            }
      }
}
```

Output

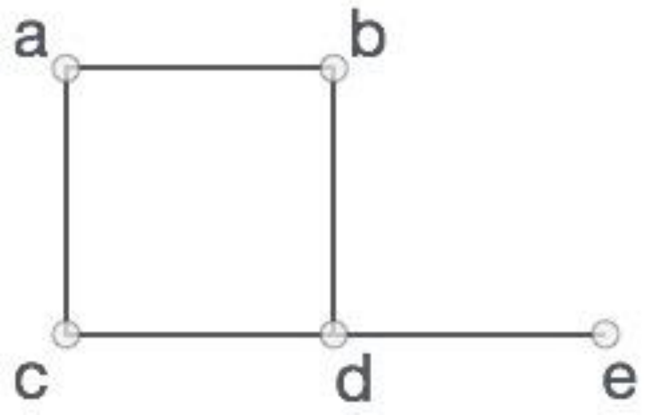
```
 1. Enter student in tree
 2. Inorder Traversal of tree
 3. Display
 4. Height of tree
```

1. Data Structure Class Slides

Lab 10 Graphs

10.1. Introduction

Graph is a data structure. A graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices.

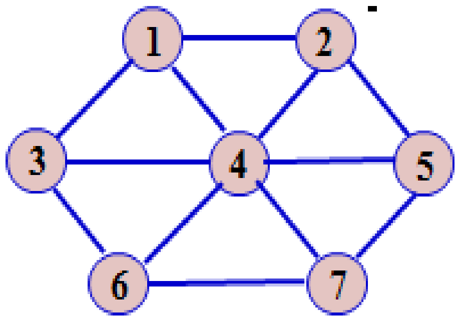


In the above graph,
V = {a, b, c, d, e}
E = {ab, ac, bd, cd, de}

10.2. Types of Graphs

10.2.1. Undirected Graphs

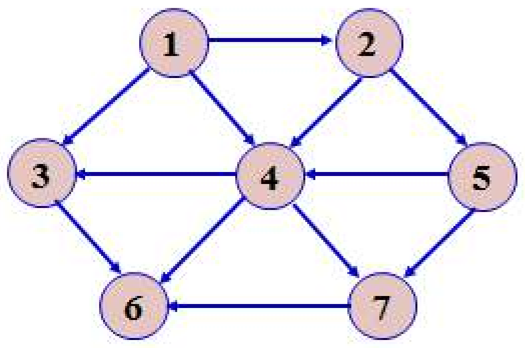
An undirected graph is graph, i.e., a set of objects (called vertices or nodes) that are connected, where all the edges are bidirectional.



The graph given above is an undirected graph. We can say that if the vertices represent cities then the edges as roads connecting those cities. Two-way roads can be represented as undirected graphs.

10.2.2. Directed Graphs

A graph where the edges point in a direction is called a directed graph.



The graph given above is a directed graph. We can say that if the vertices represent cities then the edges as roads connecting those cities. One-way roads can be represented as directed graphs where arrow points the direction of traffic movement.

10.3. Graph Representation

There are two main ways to represent graphs.

10.3.1. Adjacency matrix representation

In adjacency matrix representation, two-dimensional matrix of size *n* x *n* is used where n is the number of vertices in the graph.

For each pair of vertices (*u*, *v*),

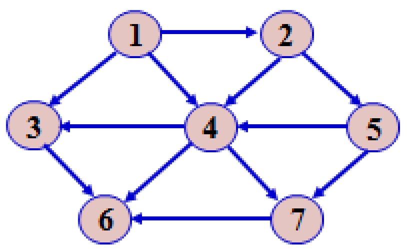
- $a[u][v] = 0$; if there is no edge between *u* and *v*
- $a[u][v] = 1$; if the edge exists between u and *v*
- $a[u][v] = w$; If the edge has a weight *w* associated with it

Use either a very large or a very small weight as a sentinel to indicate nonexistent edges.
For instance, if we were looking for the cheapest airplane route

- $a[u][v] = \infty$; if there is no edge between *u* and *v*

If we were looking, for some strange reason, for the most expensive airplane route

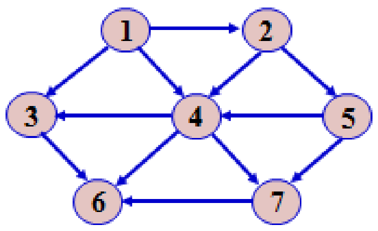
- $a[u][v] = -\infty$ or 0; if there is no edge between *u* and *v*

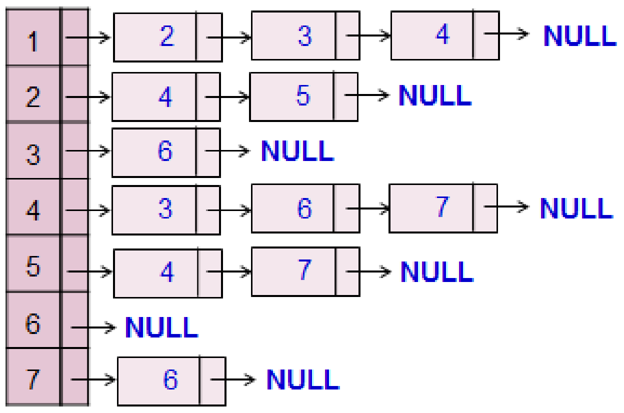


	1	2	3	4	5	6	7
1	0	1	1	1	0	0	0
2	0	0	0	1	1	0	0
3	0	0	0	0	0	1	0
4	0	0	1	0	0	1	1
5	0	0	0	1	0	0	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0

10.3.2. Adjacency list representation

If the graph is not dense, in other words, if the graph is *sparse*, a better solution to represent matrix is an *adjacency list* representation. Adjacency lists are the standard way to represent graphs. For each vertex, we keep a list of all adjacent vertices. If the edges have weights, then this additional information is also stored in the cells. The space requirement is then $O(|E| + |V|)$. Undirected graphs can be similarly represented; each edge (u, v) appears in two lists, so the space usage essentially doubles. A common requirement in graph algorithms is to find all vertices adjacent to some given vertex v, this can be done, in time proportional to the number of such vertices found, by a simple scan down the appropriate adjacency list.





10.4. Graph Operations

10.4.1. Append a new node

```
void Append ( int name )
     {
     node *ptr=new node;
     ptr->name=name;
     ptr->EHead=new edge;
     ptr->EHead->nextEdge=NULL;
     ptr->ETail= ptr-> EHead;
     ptr->next=NULL;
     tail->next=ptr;
     tail=ptr;
     }
```

10.4.2. Add a new edge

```
edge *AddEdge (int nextNode, int cost, edge *ETail )
     {
     edge *E=new edge;
     E->cost=cost;
     E->nextNode=nextNode;
     E->nextEdge=NULL;
     ETail->nextEdge=E;
     ETail=E;
     return ETail;
     }
```

Code

```
/*
Source: http://www.geeksforgeeks.org/graph-and-its-representations/
*/
#include<iostream>
#include<cstdlib>
using namespace std;

//struct for an adjacency list node
struct AdjListNode{
    int data;
    AdjListNode *next;
};

//struct for an adjacency list
struct AdjList{
    AdjListNode *head;  //pointer to head node of list
};

//struct for a graph. A graph as an array of adjacency lists
//Size of array will be V (total vertices)
struct Graph{
    int V;
    AdjList *arr;
};
```

```
AdjListNode *newAdjListNode(int);
Graph *createGraph(int);
void addEdge(Graph*,int,int);
void printGraph(Graph*);

int main(){
//create a new graph
    int totalVertices=4;
    Graph *graph;
    graph=createGraph(totalVertices);
    //connect edges
    addEdge(graph,0,1);
    addEdge(graph,0,2);
    addEdge(graph,0,3);
    addEdge(graph,1,3);
    addEdge(graph,2,3);
    /*
    addEdge(graph,0,1);
    addEdge(graph,0,4);
    addEdge(graph,1,2);
    addEdge(graph,1,3);
    addEdge(graph,1,4);
    addEdge(graph,2,3);
    addEdge(graph,3,4);
    */
    //print the adjacency list representation of graph
    printGraph(graph);
}

//create a new node
AdjListNode* newAdjListNode(int data){
    AdjListNode *nptr=new AdjListNode;
    nptr->data=data;
    nptr->next=NULL;
    return nptr;
}

//function to create a graph of V - vertices
Graph* createGraph(int V){
    Graph *graph=new Graph;
    graph->V=V;
    //create an array of adjacency list. size of array - V
    graph->arr=new AdjList[V];
    //initialize with NULL (e.g root=NULL)
    for(int i=0;i<V;i++){
        graph->arr[i].head=NULL;
    }
    return graph;
}

//add an edge to an undirected Graph
void addEdge(Graph *graph,int src,int dest){
    //Add an edge from src to dest. A new node added to the adjacency
list of src
    //node added at beginning
    AdjListNode *nptr=newAdjListNode(dest);
    nptr->next=graph->arr[src].head;
    graph->arr[src].head=nptr;
    //connect from dest to src (since undirected)
    nptr=newAdjListNode(src);
    nptr->next=graph->arr[dest].head;
    graph->arr[dest].head=nptr;
}

//function to print the graph
void printGraph(Graph* graph){
//loop over each adjacent list
    for(int i=0;i<graph->V;i++){
        AdjListNode *root=graph->arr[i].head;
        cout<<"Adjacency list of vertex "<<i<<endl;
        //loop over each node in list
        while(root!=NULL){
            cout<<root->data<<" -> ";
            root=root->next;
        }
        cout<<endl;
    }
}
```

Output

```
Adjacency list of vertex 0
3 -> 2 -> 1 ->
Adjacency list of vertex 1
3 -> 0 ->
Adjacency list of vertex 2
3 -> 0 ->
Adjacency list of vertex 3
2 -> 1 -> 0 ->
```

10.5. Graph Traversal

Graph traversal refers to the problem of visiting all the nodes in a graph in a particular manner. Tree traversal is a special case of graph traversal. In contrast to tree traversal, in general:

- In graph traversal, each node may have to be visited more than once,
- A root-like node that connects to all other nodes might not exist

10.5.1. Breadth First Search

In graph theory, breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it fulfills its objective (i.e., visit all the nodes or reach the destination node, if any). From the standpoint of the algorithm, all child nodes obtained by expanding a node are added to a FIFO (i.e., First In, First Out) queue. In typical implementations, nodes that have not yet been examined for their neighbors are placed in some container (such as a queue or linked list) and have special color (such as white) or status (such as unprocessed, not visited etc) then once examined are placed in another container and marked as processed, visited and etc.

BFS Algorithm

```
1. Enqueue the root node.
2. Dequeue a node and examine it.
3. If the element sought is found in this node, quit the search and
   return a result.
4. Otherwise enqueue any successors (the direct child nodes) that have
   not yet been discovered.
5. If the queue is empty, every node on the graph has been examined –
   quit the search and return "not found".
6. If the queue is not empty, repeat from Step 2.
```

10.5.2. Depth First Search

Formally, DFS is an uninformed search that progresses by expanding the first child node of the search tree (or graph) that appears, and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hasn't finished exploring. In a non-recursive implementation, all freshly expanded nodes are added to a stack for exploration.

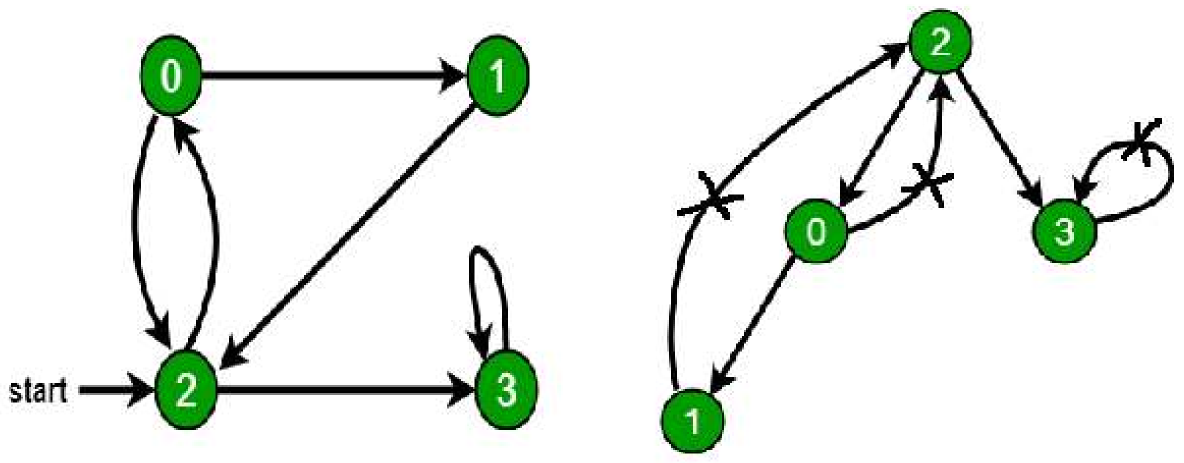
DFS Pseudocode

```
procedure DFS(G, s)
     for each vertex u in G
            u.status = notVisited/white
            u.pi = NULL
            time = 0;
     for each vertex u in G
            if u.status = notVisited/white
                   DFS_visit(G, u)

procedure DFS_visit(G, u)
     u.status = visited / grey
          time = time + 1;
        u.d = time;
     for each v adjacent to u do
          if v.status = notVisited/white
                 v.pi = u;
                 DFS_visit(G, v)
     u.status = processed/ black
```

10.6. Example Problem

Example 10-1 Graphs Example Problem

Statement



Implement the graph given in the above figure in C++ and show depth first search traversal.

Code

```
// C++ program to print DFS traversal from
// a given vertex in a given graph
#include<iostream>
#include<list>
using namespace std;

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

      // Pointer to an array containing
      // adjacency lists
      list<int> *adj;

      // A recursive function used by DFS
      void DFSUtil(int v, bool visited[]);
public:
      Graph(int V); // Constructor

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

      // DFS traversal of the vertices
      // reachable from v
      void DFS(int v);
};

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

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

```
void Graph::DFSUtil(int v, bool visited[])
{
	// Mark the current node as visited and
	// print it
	visited[v] = true;
	cout << v << " ";

	// Recur for all the vertices adjacent
	// to this vertex
	list<int>::iterator i;
	for (i = adj[v].begin(); i != adj[v].end(); ++i)
		if (!visited[*i])
			DFSUtil(*i, visited);
}

// DFS traversal of the vertices reachable from v.
// It uses recursive DFSUtil()
void Graph::DFS(int v)
{
	// Mark all the vertices as not visited
	bool *visited = new bool[V];
	for (int i = 0; i < V; i++)
		visited[i] = false;

	// Call the recursive helper function
	// to print DFS traversal
	DFSUtil(v, visited);
}

// Driver code
int main()
{
	// Create a graph given in the above diagram
	Graph g(4);
	g.addEdge(0, 1);
	g.addEdge(0, 2);
	g.addEdge(1, 2);
	g.addEdge(2, 0);
	g.addEdge(2, 3);
	g.addEdge(3, 3);

	cout << "Following is Depth First Traversal"
			" (starting from vertex 2) \n";
	g.DFS(2);

	return 0;
}
```

Output

```
Following is Depth First Traversal (starting from vertex 2)
2 0 1 3
```

Bibliography:

1. Data Structure Class Slides

Lab 11 **Hashing**

11.1. Introduction

Searching is dominant operation on any data structure. Most of the cases for inserting, deleting, and updating all operations require searchin. So, search operation of a data structure determines it's time complexity. If we take any data structure the best time complexity for searching is O (log n) in AVL trees. In most of the cases it will take O (n) time. To solve this searching problem hashing concept is introduced that takes O (1) time for searching.

In hashing, large keys are converted into small keys by using hash functions. The values are then stored in a data structure called hash table. The idea of hashing is to distribute entries (key/value pairs) uniformly across an array. Each element is assigned a key (converted key). By using that key you can access the element in O(1) time. Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted.

11.2. Implementation

Hashing is implemented in two steps:

1 An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.
2 The element is stored in the hash table where it can be quickly retrieved using hashed key.

```
hash = hashfunc(key)
index = hash % array_size
```

In this method, the hash is independent of the array size and it is then reduced to an index (a number between 0 and array_size − 1) by using the modulo operator (%).

11.2.1. Hash function

A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes. To achieve a good hashing mechanism, it is important to have a good hash function with the following basic requirements:

1. Easy to compute: It should be easy to compute and must not become an algorithm.
2. Uniform distribution: It should provide a uniform distribution across the hash table and should not result in clustering.

3. Less collisions: Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.

Note: Irrespective of how good a hash function is, collisions are bound to occur. Therefore, to maintain the performance of a hash table, it is important to manage collisions through various collision resolution techniques.

11.2.2. Hash Table

A hash table is a data structure that is used to store keys/value pairs. It uses a hash function to compute an index into an array in which an element will be inserted or searched. By using a good hash function, hashing can work well. Under reasonable assumptions, the average time required to search for an element in a hash table is O(1).
Let us consider string S. You are required to count the frequency of all the characters in this string.

```
string S = "ababcd"
```

The simplest way to do this is to iterate over all the possible characters and count their frequency one by one. The time complexity of this approach is O(26*N) where N is the size of the string and there are 26 possible characters.

Let us apply hashing to this problem. Take an array frequency of size 26 and hash the 26 characters with indices of the array by using the hash function. Then, iterate over the string and increase the value in the frequency at the corresponding index for each character. The complexity of this approach is O(N) where N is the size of the string.

```
int Frequency[26];

int hashFunc(char c)
{
    return (c - 'a');
}

void countFre(string S)
{
    for(int i = 0;i < S.length();++i)
    {
        int index = hashFunc(S[i]);
        Frequency[index]++;
    }
    for(int i = 0;i < 26;++i)
        cout << (char)(i+'a') << ' ' << Frequency[i] << endl;
}
```

11.3. Example Problem

Example 11-1 Hashing Example Problem

Statement

Implement a menu driven program to insert, search and delete elements in a hash table using C++.

Code

```
#include<iostream>
#include<cstdlib>
#include<string>
#include<cstdio>
using namespace std;
const int T_S = 200;
class HashTableEntry {
   public:
      int k;
      int v;
      HashTableEntry(int k, int v) {
         this->k= k;
         this->v = v;
      }
};
class HashMapTable {
   private:
      HashTableEntry **t;
   public:
      HashMapTable() {
         t = new HashTableEntry * [T_S];
         for (int i = 0; i< T_S; i++) {
            t[i] = NULL;
         }
      }
      int HashFunc(int k) {
         return k % T_S;
      }
      void Insert(int k, int v) {
         int h = HashFunc(k);
         while (t[h] != NULL && t[h]->k != k) {
            h = HashFunc(h + 1);
         }
         if (t[h] != NULL)
            delete t[h];
         t[h] = new HashTableEntry(k, v);
      }
      int SearchKey(int k) {
         int h = HashFunc(k);
         while (t[h] != NULL && t[h]->k != k) {
            h = HashFunc(h + 1);
         }
         if (t[h] == NULL)
            return -1;
         else
            return t[h]->v;
      }
      void Remove(int k) {
         int h = HashFunc(k);
         while (t[h] != NULL) {
            if (t[h]->k == k)
               break;
            h = HashFunc(h + 1);
         }
         if (t[h] == NULL) {
            cout<<"No Element found at key "<<k<<endl;
            return;
         } else {
            delete t[h];
         }
         cout<<"Element Deleted"<<endl;
      }
      ~HashMapTable() {
         for (int i = 0; i < T_S; i++) {
            if (t[i] != NULL)
               delete t[i];
               delete[] t;
         }
      }
};
int main() {
   HashMapTable hash;
   int k, v;
   int c;
   while (1) {
      cout<<"1.Insert element into the table"<<endl;
      cout<<"2.Search element from the key"<<endl;
      cout<<"3.Delete element at a key"<<endl;
      cout<<"4.Exit"<<endl;
      cout<<"Enter your choice: ";
      cin>>c;
      switch(c) {
         case 1:
            cout<<"Enter element to be inserted: ";
            cin>>v;
            cout<<"Enter key at which element to be inserted: ";
            cin>>k;
            hash.Insert(k, v);
         break;
         case 2:
            cout<<"Enter key of the element to be searched: ";
            cin>>k;
            if (hash.SearchKey(k) == -1) {
               cout<<"No element found at key "<<k<<endl;
               continue;
            } else {
               cout<<"Element at key "<<k<<" : ";
               cout<<hash.SearchKey(k)<<endl;
            }
         break;
         case 3:
            cout<<"Enter key of the element to be deleted: ";
            cin>>k;
            hash.Remove(k);
         break;
         case 4:
            exit(1);
         default:
            cout<<"\nEnter correct option\n";
      }
   }
   return 0;
}
```

Output

```
1.Insert element into the table
2.Search element from the key
3.Delete element at a key
4.Exit
Enter your choice: 1
Enter element to be inserted: 10
```

Lab 12 **Open Ended Lab**

In this lab, your instructor will give you one or more problems. There could be multiple possible solutions to that problem. You will be required to solve those problems in your lab without consultation with your instructor (and fellows).

12.1. Rubric

Rubric may vary according to the problem statement. A tentative rubric for the open-ended lab is as follows:

General Rubric[1]:

Running: 2

- Runs without exception: 1
- Interactive menu: 0.5
- Displays proper messages/operations clearly: 0.5

Completeness: 3

- Sub task 1: 1
- Sub task 2: 1
- Sub task 3: 1

Accuracy: 4

- Appropriate data structure: 1
- Efficient (Computational Complexity) algorithms: 1
- Passes all test cases: 1
- Viva: 1

Clarity 1 (No credit, if not relevant implementation):

- Indentation: 0.5
- Meaningful/relevant variable/function names: 0.5

[1] This is a general rubric, actual rubric may vary in every lab according to the contents

Example 12-1 Open Ended Lab Sample Problem

Problem Statement (Maximum Time: 3 hours)

Design and implement an efficient C++ program for a general store management system. The program must have the option to display list of items along with price and available quantities. A user should be able to add items to the cart, remove items from the cart and display detailed bill (sorted by item name). Please note that whenever a user adds an item to the cart the quantity of that item reduces from the store.

Rubric (10 Points)

- Running: 3
    * Runs without exception: 1
    * Interactive menu: 1
    * Displays proper messages/operations clearly: 1

- Completeness: 3
    * Display items: 0.5
    * Add to cart: 1
    * Remove from cart: 0.5
    * Display bill: 0.5
    * Sorted by name: 0.5

- Accuracy: 3
    * Apropirate data structure: 1
    * Efficient (Computational Complexity) algorithms: 1
    * Passes all test cases: 1

- Clarity 1 (No credit, if not relevant implementation):
    * Indentation: 0.5
    * Meaningful/relevant variable/function names: 0.5

Solution

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

struct items
{
    string item_name;
    float price;
    int Quantity;
    items *next;
};

struct cart
{
    string name_of_product;
    float price_of_product;
    int quantity_of_product;
    cart *next;
}*chead=NULL;

float find_price(items *head,string item)
{
    for(items *ptr=head; ptr!=NULL; ptr=ptr->next)
    {
        if (ptr->item_name==item)
        {
            return ptr->price;
        }
    }
}

void Add_to_cart(items *head,string name_of_item, int quantity)
{

    if (chead==NULL)
    {
        chead = new cart;
        chead->name_of_product=name_of_item;
        chead->quantity_of_product=quantity;
        chead->price_of_product=find_price(head,name_of_item);
        chead->next=NULL;
        for(items *ptr=head; ptr!=NULL; ptr=ptr->next)
        {
            if (ptr->item_name==name_of_item)
            {
                ptr->Quantity=(ptr->Quantity) - quantity;
            }
        }
        cout<<"Item added to cart!"<<endl;
    }
    else
    {
        cart *ptr1=chead;
        for(; ptr1->next!=NULL; ptr1=ptr1->next){}
        cart *newnode = new cart;
        newnode->name_of_product=name_of_item;
        newnode->quantity_of_product=quantity;
        newnode->price_of_product=find_price(head,name_of_item);
        ptr1->next=newnode;
        newnode->next=NULL;
        for(items *ptr=head; ptr!=NULL; ptr=ptr->next)
        {
            if (ptr->item_name==name_of_item)
            {
                ptr->Quantity=(ptr->Quantity) - quantity;
            }
        }
        cout<<"Item added to cart!"<<endl;
    }
}

void Delete_from_cart(items *head,string name_of_item)
{
    int q;
    if (chead==NULL)
    {
        cout<<"Your cart is already empty!"<<endl;
    }
    else
    {
        for(cart *ptr2=chead; ptr2->next!=NULL; ptr2=ptr2->next)
        {
            if (ptr2->next->name_of_product==name_of_item)
            {
                q=ptr2->quantity_of_product;
                ptr2->next=ptr2->next->next;
            }
        }
        for(items *ptr=head; ptr!=NULL; ptr=ptr->next)
        {
            if (ptr->item_name==name_of_item)
            {
                ptr->Quantity=(ptr->Quantity) + q;
            }
        }
        cout<<"Item deleted from cart!"<<endl;
    }
}

void Display_items(items *head)
{
    cout<<"=====:List of Available Items:====="<<endl;
    for(items *ptr=head; ptr!=NULL; ptr=ptr->next)
    {
        cout<<"Item  name:  "<<ptr->item_name<<"\t"<<"Quantity  available: 
"<<ptr->Quantity<<"\t"<<"Price of 1 piece: "<<ptr->price<<endl;
    }
}

void Display_cart()
{
    float total_bill=0;
    for(cart *ptr=chead; ptr!=NULL; ptr=ptr->next)
    {
        cout<<"Item:   "<<ptr->name_of_product<<"\t"<<"Quantity:   "<<ptr-
>quantity_of_product<<"\t"<<"Price:    "<<(ptr->quantity_of_product)*(ptr-
>price_of_product);
        total_bill+=(ptr->quantity_of_product)*(ptr->price_of_product);
        cout<<endl;
    }
    cout<<"Your total bill is: Rs "<<total_bill<<endl;
```

```
}
int main()
{
    string name_of_item;
    int quantity,choice;
    items *head= new items;
    head->next=NULL;

    //cart *chead= NULL;
    //chead=NULL;

    head->item_name="tissues";
    head->price=200;
    head->Quantity=5;

    head->next=new items;
    head->next->item_name="lays";
    head->next->price=20;
    head->next->Quantity=50;

    head->next->next=new items;
    head->next->next->item_name="cornetto";
    head->next->next->price=45;
    head->next->next->Quantity=75;

    head->next->next->next=new items;
    head->next->next->next->item_name="milk";
    head->next->next->next->price=100;
    head->next->next->next->Quantity=44;
    head->next->next->next->next=NULL;

    while(true){

    cout<<"=============:STORE MENU:============="<<endl;
    cout<<"Press 1 to See list of available items"<<endl;
    cout<<"Press 2 to Add an item to your cart"<<endl;
    cout<<"Press 3 to Delete an item from your cart"<<endl;
    cout<<"Press 4 to View your cart"<<endl;
    cin>>choice;

    switch(choice)
    {
        case 1:
            Display_items(head);
            break;
        case 2:
            cout<<"Enter name of item: ";
            cin>>name_of_item;
            cout<<"Enter quantity: ";
            cin>>quantity;
            Add_to_cart(head,name_of_item,quantity);
            break;
        case 3:
            cout<<"Enter name of item: ";
            cin>>name_of_item;
            cout<<"Enter quantity: ";
            cin>>quantity;
            Delete_from_cart(head,name_of_item);
            break;
        case 4:
            Display_cart();
```

```
                break;
            default:
                cout<<"Invalid choice!";
                break;
        }
    }
}
```