Practical Workbook

CT-159

DATA STRUCTURES ALGORITHMS & APPLICATIONS

Name: ______________________________

Roll No: ______________________________

Batch: ______________________________

Department: ______________________________

Practical Workbook

DATA STRUCTURES ALGORITHMS & APPLICATIONS

(CT – 159)

Prepared by
Dr. Muhammad Kamran / Ms. Huma Tabassum
CS & IT

Approved by

Chairman

Department of Computer Science & Information Technology
NED University of Engineering & Technology

Table of Contents

Lab 01 - Arrays and Dynamic Memory Allocation

Objective:

The objective of this experiment to get familiar with

1. Arrays and its role in data structure
2. Storage of data in Row major order and column major order.
3. Safe Arrays
4. Jagged Array
5. Implementation of linear and binary searching techniques.

Introduction:

An array is a series of elements of the same data type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

int arr[5]

For example, five values of type **int** can be declared as an array without having to declare 5 different variables (each with its own identifier). These values can be accessed using the same identifier, with the proper index. **Multidimensional arrays** can be described as "arrays of arrays". For example, a bi-dimensional array can be imagined as a two-dimensional table made of elements, all of them of a same uniform data type.

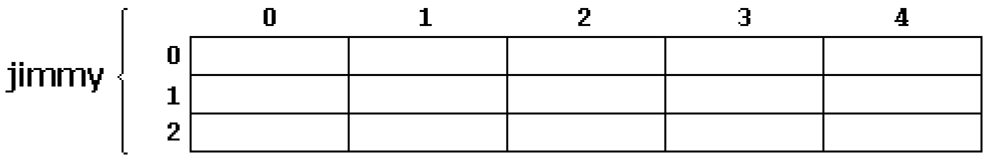


Figure 1.1 2D Arrays

Jimmy represents a bi-dimensional array of 3 per 5 elements of type int as shown in above Figure 1.1. The C++ syntax for this is: **int jimmy[3][5]**

In addition, for example, the way to reference the second element vertically and fourth horizontally in anexpression would be: **jimmy[1][3]**

Dynamic Memory Allocation for arrays:

Memory in your C++ program is divided into two parts

1. The stack − All variables declared inside the function will take up memory from the stack.
2. The heap − this is unused memory and can be used to allocate the memory dynamically during program execution.

A **dynamic array** is an array with a big improvement, that is, **automatic resizing**. One limitation of static arrays is that they're fixed size, meaning you need to specify the number of elements your array will hold ahead of time. A dynamic array expands as you add more elements. So you don't need to determine the size ahead of time.

Strengths:

1. **Fast lookups**. Just like static arrays, retrieving the element at a given index takes O(1)

time.
2. **Variable size.** You can add as many items as you want, and the dynamic array will expand to hold them.
3. **Cache-friendly.** Just like static arrays, dynamic arrays place items right next to each other in memory, making efficient use of caches.

Weaknesses:

1. **Slow worst-case appends.** Usually, adding a new element at the end of the dynamic array takes **O(1)** time. But if the dynamic array doesn't have any room for the new item, it'll need to expand, which takes **O(n)** time.
2. **Costly inserts and deletes.** Just like static arrays, elements are stored adjacent to each other. So adding or removing an item in the middle of the array requires "scooting over" other elements, which takes **O(n)** time.

Factors impacting the performance of Dynamic Arrays:

The array's initial size and its growth factor determine its performance. Note the following points:

1. If an array has a **small size** and a **small growth factor**, it will keep on **reallocating** memory more often. This will **reduce** the performance of the array.
2. If an array has a **large size** and a **large growth factor**, it will have a **huge chunk** of **unused** memory. Due to this, resize operations may take longer. This will reduce the performance of the array.

The new Keyword:

In C++, we can create a dynamic array using the **new keyword**. The number of items to be allocated is specified within a pair of square brackets. The type name should precede this. The requested number of items will be allocated.

Syntax:

```
int *ptr1 = new int;
int *ptr1 = new int[5];
int *array { new int[10]{}};
int *array { new int[10]{1,2,3,4,5,6,7,8,9,10}};
```

Resizing Arrays:

The length of a dynamic array is set during the allocation time.However, C++ doesn't have a built-in mechanism for resizing an array once it has been allocated. You can, however, overcome this challenge by allocating a new array dynamically, copying over the elements, then erasing the old array.

Dynamically Deleting Arrays:

A dynamic array should be deleted from the computer memory once its purpose is fulfilled. The delete statement can help you accomplish this. The released memory space can then be used to hold another set of data. However, even if you do not delete the dynamic array from the computer memory, it will be deleted automatically once the program terminates.

Syntax:

```
delete ptr;
delete[] array;
```

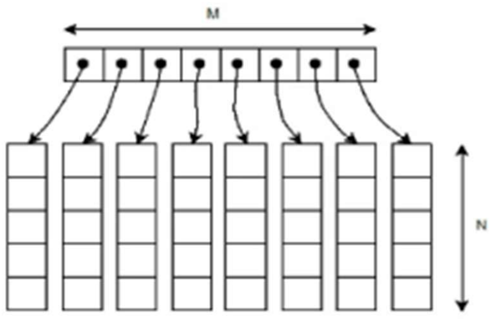
NOTE: To delete a dynamic array from the computer memory, you should use delete[], instead of delete. The [] instructs the CPU to delete multiple variables rather than one variable. The use of delete instead of delete[] when dealing with a dynamic array may result in problems. Examples of such problems include **memory leaks, data corruption, crashes,** etc.

Example: Single Dimensional Array:

```
#define N 10
int main(){
        // dynamically allocate memory of size 5 and assign values to allocated memory
        int *array{ new int[5]{ 10, 7, 15, 3, 11 } };
        // print the 1D array de allocate memory
}
```

Two Dimensional Array Using Array of Pointers:

We can dynamically create an array of pointers of size M and then dynamically allocate memory of size N for each row as shown below.



Example:

```
// Dynamically Allocate Memory for 2D Array in C++
int main(){
//Enter two dimensions N and M, Dynamically allocate memory to both
int** ary = new int*[N];
  for(int i = 0; i < N; ++i)
      ary[i] = new int[M];
//fill the arrays, print them, deallocate the memory
}
```

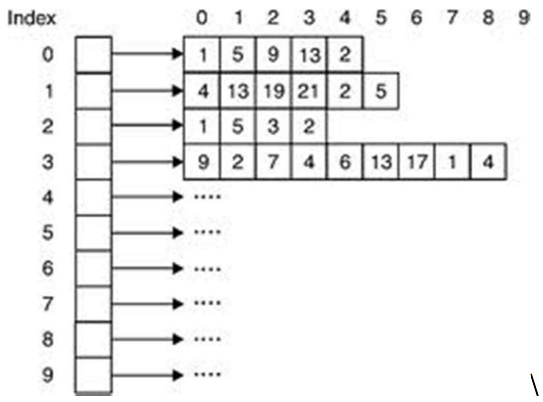
Safe Array:

In C++, there is **no check** to determine whether the **array index** is **out of bounds**. During program execution, an out-of- bound array index can cause **serious problems**. Also, recall that in C++ the array index starts at 0. Safe array solves the out-of-bound array index problem and allows the user to begin the array index starting at any integer, positive or negative. **"Safely"** in this context would mean that access to the array elements must not be **out of range**. ie. the position of the element must be **validated** prior to access. For example in the member function to allow the user to set a value of the array at a particular location:

```
void set(int pos, Element val){     //set method
        if (pos<0 || pos>=size)
                cout<<"Boundary Error\n";
        else Array[pos] = val;
}
```

Jagged Array

Jagged array is nothing but it is **an array of arrays** in which the member arrays can be in different sizes.



Example:

```
int **arr = new int*[3];
int Size[3], i,j,k;
for(i=0;i<3;i++){
        cout<<"Row "<<i+1<< " size: ";
        cin>>Size[i];
        arr[i] =new int[Size[i]]; }
for(i=0;i<3;i++){
        for(j=0;j<Size[i];j++){
                cout<<"Enter row " <<i+1<<" elements: ";
                cin>>*(*(arr + i) + j);  }
}// print the array elements  deallocate memory using delete[] operator
```

Linear Search:

Linear Search is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found, otherwise the search continues till the end of the data set. It is the easiest searching algorithm as shown in figure 1.4:

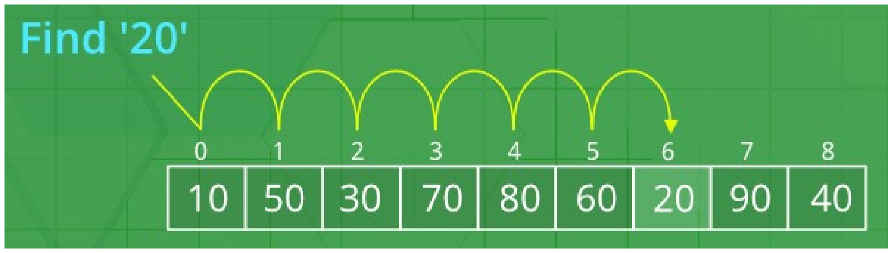


Figure 1.4 Linear Search

Algorithm:

1. Start from the leftmost element of arr[] and one by one compare x with each element of arr[],
2. If x matches with an element, return the index,
3. If x doesn’t match with any of the elements, return -1

Code:

```
#include <iostream>
using namespace std;
int search(int arr[], int N, int x){
    int i;
    for (i = 0; i < N; i++)
        if (arr[i] == x)
            return i;
    return -1;
}
int main(void){
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int N = sizeof(arr) / sizeof(arr[0]);
    // Function call
    int result = search(arr, N, x);
    (result == -1)
        ? cout << "Element is not present in array"
        : cout << "Element is present at index " << result;
    return 0;}
```

Binary Search:

Binary Search is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n) as shown in Figure 1.5.

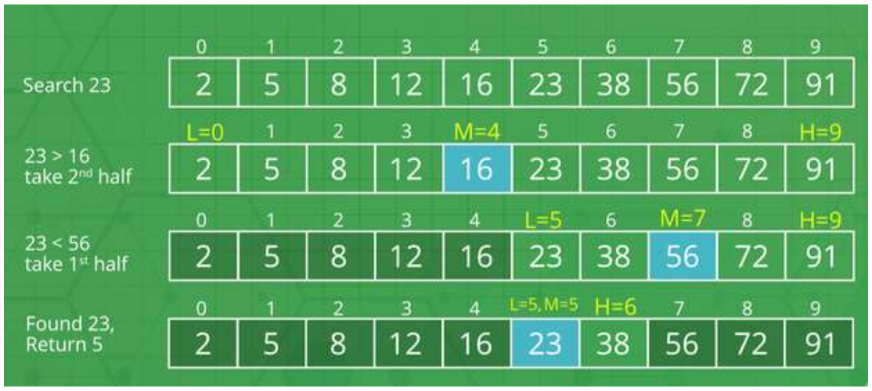


Figure 1.5 Binary Search

Algorithm:

1. Compare x with the middle element.
2. If x matches with the middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in the right half subarray after the mid element. So we recur for the right half.
4. Else (x is smaller) recur for the left half.

Code: (Iterative Approach)

```
using namespace std;
// A iterative binary search function. It returns
// location of x in given array arr[l..r] if present, otherwise -1
int binarySearch(int arr[], int l, int r, int x){
    while (l <= r) {
        int m = l + (r - l) / 2;
        // Check if x is present at mid
        if (arr[m] == x)
            return m;
        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;
        // If x is smaller, ignore right half
        else
            r = m - 1;
    }
    // if we reach here, then element was
    // not present
    return -1;
}
int main(void){
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1)
        ? cout << "Element is not present in array"
        : cout << "Element is present at index " << result;
    return 0;
}
```

Exercise

1. Write a C++ code to copy data of a 2D array in a 1D array using Column Major Order.

2. Write a program to calculate the CGPA of students of all subjects of a single semester. Assume all the courses have the same credit hour (let's assume 3 credit hours).

	Data Structure	**Programming for AI**	**Digital Logic Design**	**Probability & Statistics**	**Finance & Accounting**
Ali	3.66	3.33	4.0	3.0	2.66
Hiba	3.33	3.0	3.66	3.0	---
Asma	4.0	3.66	2.66	---	---
Zain	2.66	2.33	4.0	---	---
Faisal	3.33	3.66	4.0	3.0	3.33

3. Suppose you are planning a picnic. You want to store the names of the students of your class who have submitted their contribution money. Write a program that uses jagged array to store the students' names and then perform a search for your name.

4. Using the abstract data Type of a Matrix given below, write a program that
   1. Input a 4*3 matrix from user in 2D array
   2. Map this array in 1D array using Row major order
   3. Input second matrix of 3*4 in 2D array
   4. Map this array in 1D array using Row major order.
   5. Now perform matrix multiplication **on these 1D arrays**
   6. Save the result back in a 2D array.

Implement this question for any number of rows and columns using class "matrix".

```
#include<iostream>
using namespace std;
class matrix{
    int **p;
    int   r;
    int c;
    int   *rowmajor;
    int *multiply1D;
public:
  matrix(int row, int col);
  // Constructor

  void disp2D();
  // displays the elements of **p

  void dispRowMajor();
  // converts 2D into 1D using row major
//and displays the elements Row Major Order Matrix

  void Multiply_rowMajor(matrix & x);
  // Multiplies Matrices in row major order and
save the result in a 1D dynamic array

 void rowMajor_2D();
  // Maps the elements stored in row major order to
  // the 2D array and print the results
  ~matrix();
  // Destructor
}
```

```
void main()
{
       matrix a(4,3);
       matrix b(3,4);
       a.disp2D();
       a.dispRowMajor();
       b.disp2D();
       b.dispRowMajor();
       a.Multiply_rowMajor(b);
       a.rowMajor_2D();

}

matrix::matrix(int row,int col)
{
       r=row;
       c=col;
       p   =   new   int*[r];
       for(int  i=0;i<r;i++)
       {
              p[i]=new     int[c];
              for(int j=0;j<c;j++)
                     p[i][j]=(i+j);
       }
// CODE FOR STORING DATA FROM
// **P TO *rowmajor ROW MAJOR
}
```

5. Write a program that creates a 2D array of 5x5 values of type boolean. Suppose indices represent cities and that the value at row i, column j of a 2D array is true just in case i and j are direct neighbors and false otherwise. Use initializer list to instantiate and initialize your array to represent the following configuration: (* means "neighbors")

	Khi	hyd	jam	Mpk	kot
Khi		*			
Hyd	*		*		
Jam		*			*
Mpk		*	*		
Kot		*	*	*	

Write a method to check whether two cities have a common neighbor. For example, in the example above, 0 and 2, 3 and 4 are neighbors with 1 (so they have a common neighbor), whereas 0 and 1 have no common neighbors.

Lab 02- Sorting Techniques

Objective:

The objective of this lab is to implement basic sorting techniques: insertion sort, bubble sort and selection sort.

Introduction:

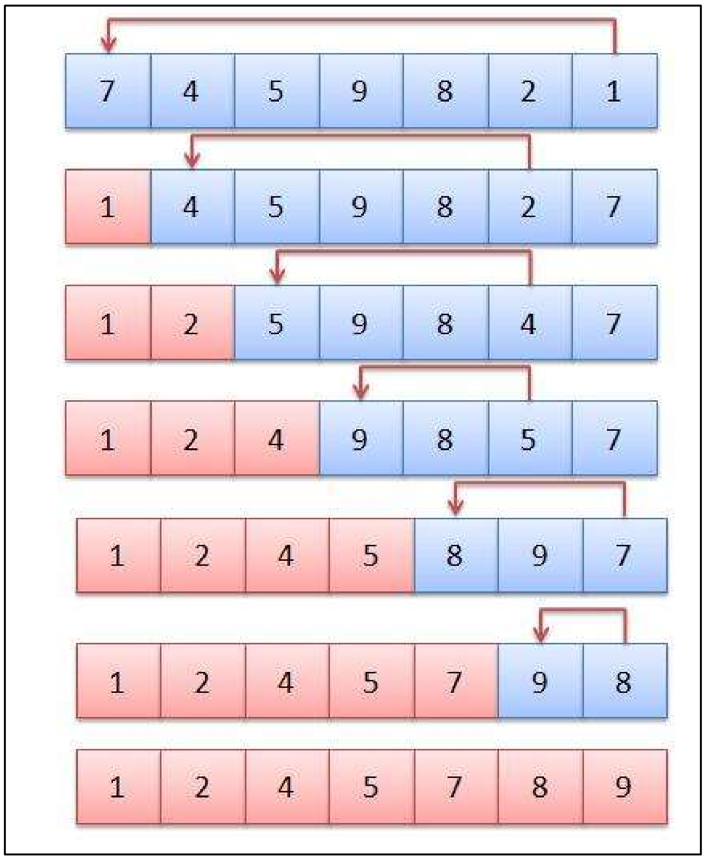
A Sorting Algorithm is used to rearrange a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.

Selection Sort:

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

- The subarray which already sorted.
- The remaining subarray was unsorted.

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



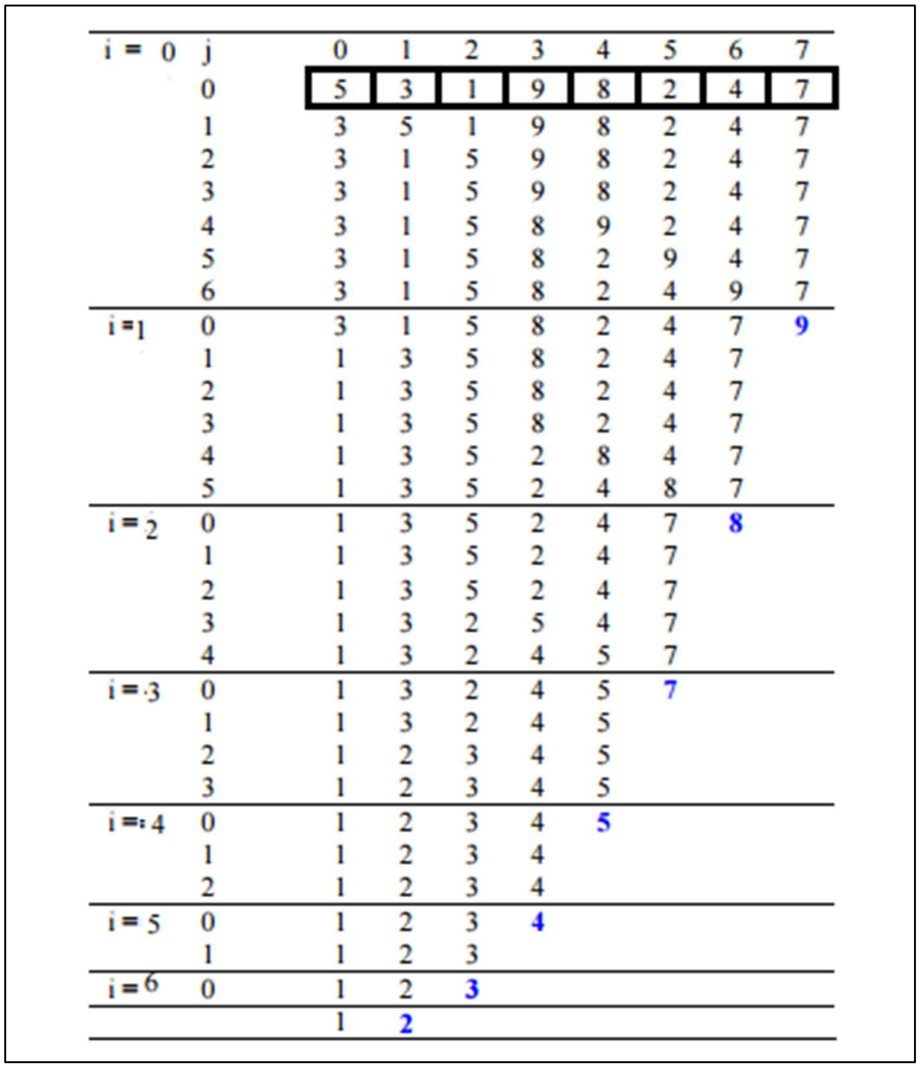
Example 01:

```
// C++ program for implementation of selection sort
#include <bits/stdc++.h>
using namespace std;
//Swap function
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
                if(min_idx!=i)
                        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;
}
```

Bubble Sort:

It is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high. Follow the below steps to solve the problem:

1. Run a nested for loop to traverse the input array using two variables i and j, such that 0 ≤ i < n-1 and 0 ≤ j < n-i-1
2. If arr[j] is greater than arr[j+1] then swap these adjacent elements, else move on
3. Print the sorted array



i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i = 1	0	3	1	5	8	2	4	7	**9**
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 2	0	1	3	5	2	4	7	**8**	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 3	0	1	3	2	4	5	**7**		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i = 4	0	1	2	3	4	**5**			
	1	1	2	3	4				
	2	1	2	3	4				
i = 5	0	1	2	3	**4**				
	1	1	2	3					
i = 6	0	1	2	**3**					
		1	**2**						

Example 02:

```
// C++ program for implementation of Bubble sort
#include <bits/stdc++.h>
using namespace std;
// A function to implement bubble sort
void bubbleSort(int arr[], int n){
        int i, j;
        for (i = 0; i < n - 1; i++)
                // Last i elements are already in place
                for (j = 0; j < n - i - 1; j++)
                        if (arr[j] > arr[j + 1])
                                swap(arr[j], arr[j + 1]);
}
// 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[] = { 5, 1, 4, 2, 8};
        int N = sizeof(arr) / sizeof(arr[0]);
        bubbleSort(arr, N);
        cout << "Sorted array: \n";
        printArray(arr, N);
        return 0;
}
```

Insertion sort

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

Characteristics of Insertion Sort:

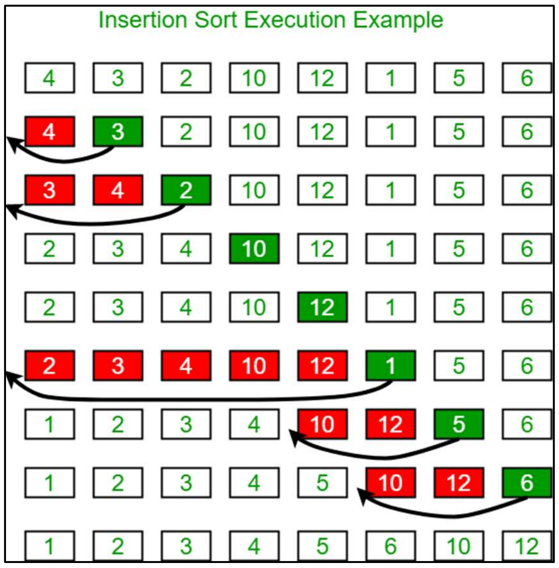
1. This algorithm is one of the simplest algorithm with simple implementation
2. Basically, Insertion sort is efficient for small data values
3. Insertion sort is adaptive in nature, i.e. it is appropriate for data sets which are already partially sorted.

To sort an array of size N in ascending order:

1. Iterate from arr[1] to arr[N] over the array.
2. Compare the current element (key) to its predecessor.
3. If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Example 03:

```
// C++ program for insertion sort
#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;
}
```



Exercise

1. If the array is already sorted, we don’t want to continue with the comparisons. This can be achieved with modified bubble sort. Update the code in example 02 to have a modified bubble sort function.

2. Create a Person class which has following attributes:

   First Name
   Last Name
   Birth Year
   Birth Month
   Birth Date

   Develop C++ solution such that day month and year are taken as input for *N* persons and perform Sorting based on year, month and day using Selection Sort.

3. Given an array **arr[ ]** of length **N** consisting cost of **N** toys and an integer **K** the amount with you. The task is to find maximum number of toys you can buy with **K** amount.

   Test Case:
   Input: N = 7, K = 50, arr[] = {1, 12, 5, 111, 200, 1000, 10}
   Output: 4
   Explanation: The costs of the toys. You can buy are 1, 12, 5 and 10.

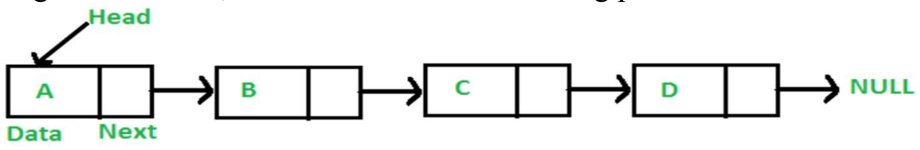
4. Create a single class Sort, which will provide the user the option to choose between all 4 sorting techniques. The class should have following capabilities:

   - ✓ Take an array and a string (indicating the user choice for sorting technique) as input and perform the desired sorting.
   - ✓ Should allow the user to perform analysis on a randomly generated array. The analysis provides number of comparisons and number of swaps performed for each technique.
   - ✓ After printing all the results, the class should highlight the best and worst techniques.

Lab 03 - Singly Linked List

Objective:
The objective of this lab is to implement Singly Linked List.

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.



Singly Linked List
A singly linked list is a collection of components, called **nodes**. Every node (except the last node) contains the address of the next node. Thus, every node in a linked list has two components: one to store the relevant information (that is, data) and one to store the address, called the **link**, of the next node in the list. The address of the first node in the list is stored in a separate location, called the **head** or **first**. Last node to list points to NULL.

Properties:

- ✓ Singly linked list can store data in non-contiguous locations.
- ✓ Insertion and deletion of values is efficient with respect to Time and Space Complexity.
- ✓ Dynamic structure (Memory Allocated at run-time).
- ✓ Nodes can only be accessed sequentially. That means, we cannot jump to a particular node directly.
- ✓ There is no way to go back from one node to previous one. Only forward traversal is possible.

Basic Functionality:

1. Insertion of new node in Singly Linked List is depicted in Figure 3.1

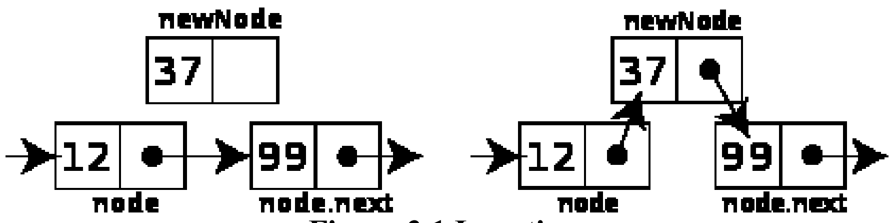


Figure 3.1 Insertion

2. Deletion of node from Singly Linked List is depicted in Figure 3.2

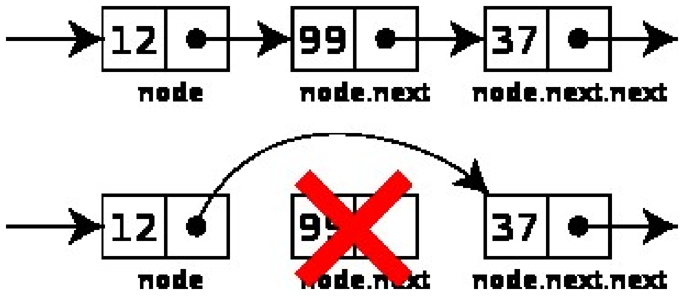


Figure 3.2 Deletion

3. Searching a node based on content. If the specified node is not present in the list an error message should be displayed.

Example 01: Implement a Singly Linked List Class

```
//Node Object
Class Node {//public members: key, data, next
  Node () {
     //initialize both key and data with zero while next pointer with NULL;
  }
  Node (int k, int d) {
     //assign the data members initialized in default constructor to these arguments.
  }
};

//SinglyLinkedList Object
Class SinglyLinkedList {
        // create head node as a public member Default Constructor
    SinglyLinkedList() {
      //initialize the head pointer will NULL;
}
        //Parameterized Constructor with node type 'n' pointer
    SinglyLinkedList (Node* n){
      //Assign the head pointer to value of pointer n;
    }
};
```

Add a Node at the End:

The new node is always added after the last node of the given Linked List. For example, if the given Linked List is 5->10->15->20->25 and we add an item 30 at the end, then the Linked List becomes 5->10->15->20->25->30. Since a Linked List is typically represented by the head of it, we have to traverse the list till the end and then change the next to last node to a new node.

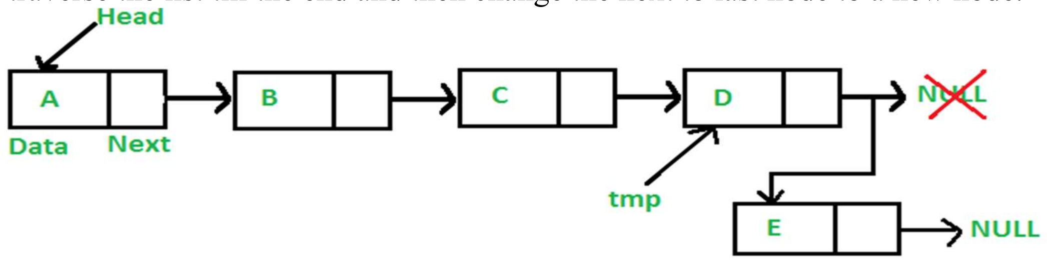


Figure 3.3 Adding a node at the end of a linked list

Check if the list has some node or not, i.e., if the head pointer is null or not. If it is null, access the head and assign node n to it. Otherwise, traverse through the list to find that the node whose next is Null, i.e. the last node in the list. Then, assign the node n to next pointer of the last node. Also, make sure the next pointer of node n (new last node) is null.

```
//Append Function
 Void appendNode (Node* n, int data){
     //create new node and assign data to it. check if the head pointer points to Null or not with a condition
     if (head  == NULL){
         //If it does, assign the head pointer the passed pointer 'n'. This will put the address of node n in the head pointer.}

       else{// Else traverse through the list to find the next pointer holding Null as address (last node)
            if (nodeExists(n->key) != NULL  ) {
               // Print an intimation that a node holding passed key already exists.
             }
           else{
                If (head != NULL){
                    // assign the head pointer to a new pointer of type Node (say, ptr).
                   //loop through the list using ptr until the next of any node contains null.
                   //when ptr->next = NULL. Then assign n node to ptr->next to append that node after the last
                }
            }
       }
}
```

Add a Node at the Front:

The new node is always added before the head of the given Linked List. And newly added node becomes the new head of the Linked List. For example, if the given Linked List is 10->15->20->25 and we add an item 5 at the front, then the Linked List becomes 5->10->15->20->25. Let us call the function that adds at the front of the list is push (). The push () must receive a pointer to the head pointer, because push must change the head pointer to point to the new node.

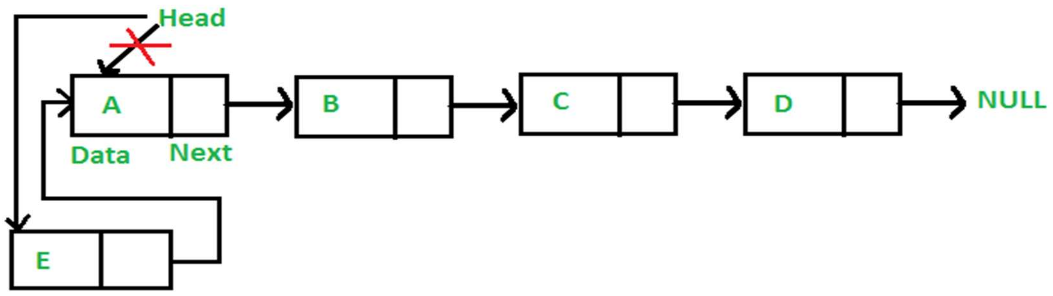


Figure 3.4 Adding a node to the start of a linked list

```
 //Prepending a Node.
 void prependNode(Node* n) {
            // checking the value of node's key if the node with that key already exists.
     if (nodeExists(n->key) != NULL  ) {
    // Print an intimation that a node holding passed key already exists.
     }
     else{
       // new node's next is pointing to the head i.e. address of first node.
       // since we have changed the head pointer's value from first node to the new node, now the
new                              .      // head will be pointing to address of new node.
     }
}
```

Insert Node After a Node:

For inserting a new node after a given node, create a void function named insertNodeAfter () carrying two arguments, one for the key of the node after which the insertion is to be done, the new node.

```
//Inserting a new node after some node.
Void insertAfterNode (key , new node){
//creating a node type pointer that calls nodeExists () and passes the key argument into it to
check if there exists a node with this key value
 Node * ptr = nodeExists(k);
 If(ptr == NULL){
//print a message saying no node exists with that key
}
Else {
//now the next pointer of new node will be holding the address kept in next pointer of ptr.
//assign the address of node to the next pointer of the previous node (ptr).
//print a message that anode is inserted;
}
```

Deleting a Node

To delete a node from the linked list, we need to do the following steps.

- ✓ Find the previous node of the node to be deleted.
- ✓ Change the next of the previous node to hold the node next to the node to be deleted.
- ✓ Free memory for the node to be deleted.

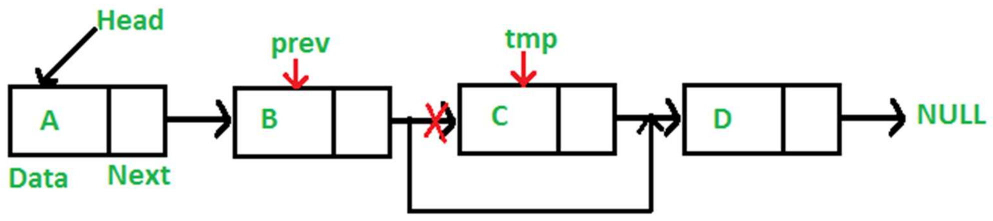


Figure 3.5 Deleting a node from a linked list

```
void deleteNode(key){
        // create a Node type pointer to hold head (say, temp)
        // create a Node type pointer to hold previous node (say, prev)
        // check if head contains the key
        if (temp!=NULL && temp->key==key){
                // assign next of temp to temp, which will unlink the node pointed by head
                // delete the temp node
                // return
         }
        Else{
             While(temp!=NULL && temp->key != key){
                  // traverse the list until temp is not NULL //and temp's key is same as the key.
                 // set prev to temp
                // set temp to temp's next pointer
              }
              If(temp == NULL){      // key not found.
                        // return
              }
                // unlink by setting temp's next to prev's next.
                // free memory by deleting temp
        }
}
```

Updating a Linked List

Updating Linked List or modifying Linked List means replacing the data of a particular node with the new data. Implement a function to modify a node's data when the key is given. First, check if the node exists using the helper function nodeExists.

```
void updateNode(key, new_data){
// call the nodeExists function and hold the return value in a Node type pointer (say, ptr)
     If(ptr!=NULL){
        // set ptr's data as new data
      }
      Else{
        // print a message sating node does not exist.
      }
}
```

Exercise

1. Implement a singly linked list class with the following functions:
    a) Insert a node at head
    b) Insert a node at tail/end/back
    c) Insert a node at any position
    d) Delete a node by value
    e) Delete head
    f) Delete tail
    g) Delete a node at any position.

2. Solve the following problem using a Singly Linked List. Given a singly linked list of characters, write a function to make word out of given letters in the list. Test Case:

```
Input:C->S->A->R->B->B->E->L->NULL,
Output:S->C->R->A->B->B->L->E->NULL
```

3. Use the class of SLL created by you during the lab task 1. Do the following:
    a) Reverse the linked list
    b) Sort the contents of linked list
    c) Find the duplicates in the linked list

Lab 04 - Doubly and Circular Linked List

Objective:

The objective of this lab is to implement Doubly Linked List.

Introduction:

A doubly linked list is a linked list in which every node has a next pointer and a back pointer. In other words, every node contains the address of the next node (last node points to null as next node), and every node contains the address of the previous node (first node points to null as the previous node) as shown in Figure 4.1. Insertion and Deletion of Node are depicted in Figure 4.1 and 4.2 respectively.

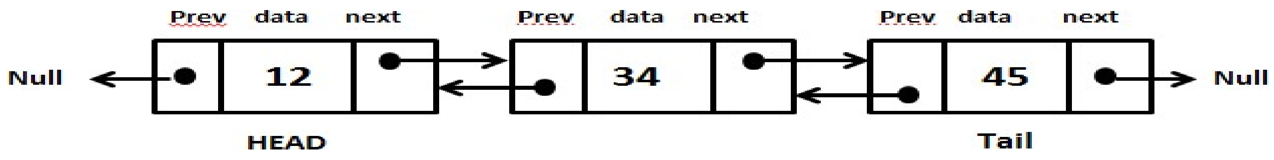


Figure 4.1 Doubly linked list

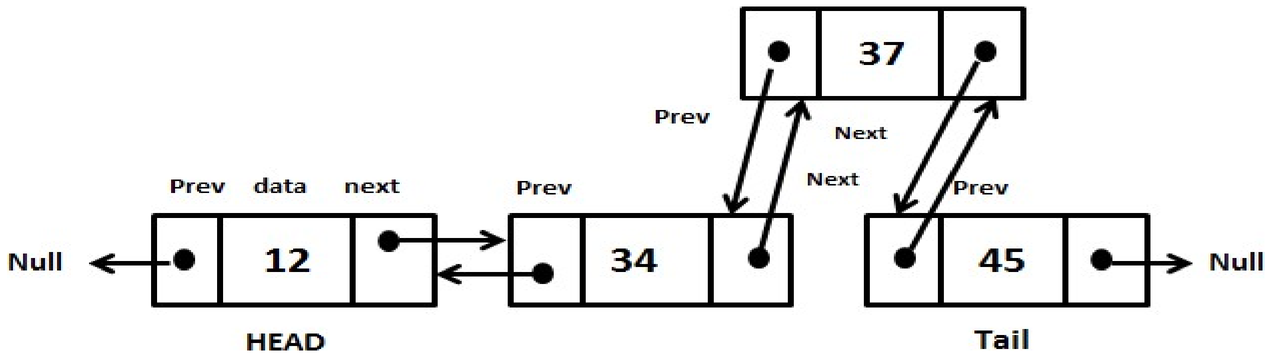


Figure 4.2 Insertion

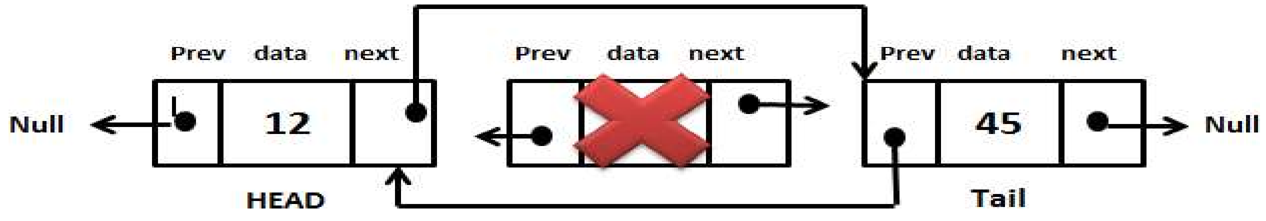


Figure 4.3 Deletion

```
class Node {
public:
  int key;
  int data;
  Node * next;
  Node * previous;
  Node() {
     key = 0;
     data = 0;
     next = NULL;
     previous = NULL;
  }
  Node(int k, int d) {
     key = k;
     data = d;
   }
};

class DoublyLinkedList {
  public:
    Node * head;
    DoublyLinkedList() {
       head = NULL;
     }
    DoublyLinkedList(Node * n) {
       head = n;
    }
   appendNode();
   prependNode();
   insertNodeAfter();
  deleteNodeByKey();
  updateNodeByKey();
};
```

Circular Linked List

In some situations, a *circular list* is needed in which nodes form a ring: the list is finite and each node has a successor. An example of such a situation is when several processes are using the same resource for the same amount of time, and we have to ensure that each process has a fair share of the resource. Therefore, all processes—let their numbers be 6, 5, 8, and 10, as in Figure—are put on a circular list accessible through the pointer current. After one node in the list is accessed and the process number is retrieved from the node to activate this process, current moves to the next node so that the next process can be activated the next time.

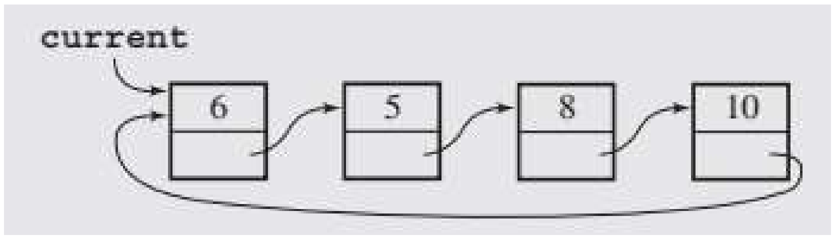


Figure 4.4 Current Pointer

```
class Node {
    public:
       int key;
       int data;
      Node * next;
      Node() {
         key = 0;
         data = 0;
         next = NULL;
      }
      Node(int k, int d) {
         key = k;
         data = d;
      }
};
class CircularLinkedList {
     public:
       Node * head;
       CircularLinkedList() {
       head = NULL;
       }
      appendNode();
      prependNode();
      insertNodeAfter();
      deleteNodeByKey();
      updateNodeByKey();
      print();
};
```

Exercise

1. Create a doubly link list and perform the mentioned tasks.
    a. Insert a new node at the end of the list.
    b. Insert a new node at the beginning of list.
    c. Insert a new node at given position.
    d. Delete any node.
    e. Print the complete doubly link list.

2. Create two doubly link lists, say L and M . List L should contain all even elements from 2 to 10 and list M should contain all odd elements from 1 to 9. Create a new list N by concatenating list L and M.

3. Using the above created list N, sort the contents of list N is descending order.

4. Create a circular link list and perform the mentioned tasks.
    a. Insert a new node at the end of the list.
    b. Insert a new node at the beginning of list.
    c. Insert a new node at given position.
    d. Delete any node.
    e. Print the complete doubly link list.

5. Break the above-created circular linked list into two halves.

Lab 05 - Stacks

Objective:

The objective of the lab is to develop understanding of the Stack data structure and its basic functions.

Introduction:

A stack is a linear data structure that can be accessed only at one of its ends for storing and retrieving data. Such a stack resembles a stack of trays in a cafeteria: new trays are put on the top of the stack and taken off the top. The last tray put on the stack is the first tray removed from the stack. For this reason, a stack is called an LIFO structure: last in/first out. A stack is defined in terms of operations that change its status and operations that check this status. The operations are as follows:

- ✓ clear()—Clear the stack.
- ✓ isEmpty()—Check to see if the stack is empty.
- ✓ push(el)—Put the element el on the top of the stack.
- ✓ pop()—Take the topmost element from the stack.
- ✓ topEl()—Return the topmost element in the stack without removing it.

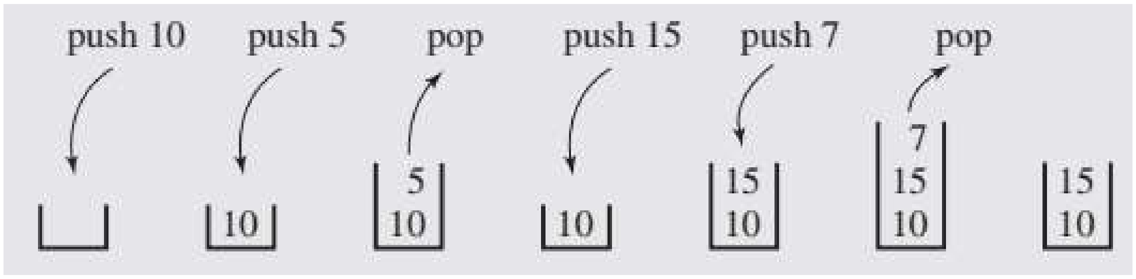


Figure 5.1 Push and Pop Operations in Stack

Implement Stack using array.

```
/* C++ program to implement basic stack operations */
#include <bits/stdc++.h>
using namespace std;
#define MAX 1000
class Stack {
        int top;
public:
        int a[MAX]; // Maximum size of Stack
        Stack() { top = -1; }
        bool push(int x);
        int pop();
        int peek();
        bool isEmpty();
};
```

```
bool Stack::push(int x){
	if (top >= (MAX - 1)) {
		cout << "Stack Overflow";
		return false;
	}
	else {
		a[++top] = x;
		cout << x << " pushed into stack\n";
		return true;
	}
}

int Stack::pop(){
	if (top < 0) {
		cout << "Stack Underflow";
		return 0;
	}
	else {
		int x = a[top--];
		return x;
	}
}

int Stack::peek(){
	if (top < 0) {
		cout << "Stack is Empty";
		return 0;
	}
	else {
		int x = a[top];
		return x;
	}
}

bool Stack::isEmpty(){
	return (top < 0);
}
```

```
// Driver program to test above functions
int main(){
        class Stack s;
        s.push(10);
        s.push(20);
        s.push(30);
        cout << s.pop() << " Popped from stack\n";
        //print top element of stack after poping
        cout << "Top element is : " << s.peek() << endl;
        //print all elements in stack :
        cout <<"Elements present in stack : ";
        while(!s.isEmpty()){
                // print top element in stack
                cout << s.peek() <<" ";
                // remove top element in stack
                s.pop();
        }
        return 0;
}
```

Implement Stack using Singly Linked list.

Implement all the methods given in the following class definition for Stack using linked list as the underlying data structure. Your implementation should work for the main function that follows.

```
#ifndef STACK_H
#define STACK_H
template<class KeyType>

class Stack{ //
 public:
// constructor , creates an empty stack
      Stack();
// returns true if Stack is full, otherwise return false
      bool IsFull();
//If number of elements in the Stack is zero return true, otherwise return false
       bool IsEmpty();

// If Stack is not full, insert item into the Stack. Must be an O(1) operation
     void Push(const KeyType item);

// If Stack is empty return 0 or NULL;
// else return appropriate item from the Stack. Must be an O(1) operation
      KeyType Pop();
private:
   SList <KeyType>* list; Node<KeyType>* top;
};
#endif
________________________________________________________________________________
#include "Stack.h"
#include "Stack.cpp"
#include <iostream>
using namespace std;
int main(){
   Stack<int> *st =new Stack<int>();
    if(st->IsEmpty())
       cout<<"Stack is currently empty"<<endl;
       st->Push(1);
       st->Push(2);
       st->Push(3);
    while (!st->IsEmpty()){
       int value=st->Pop(); cout<<value<<endl;
     }
  return 0;
}
```

Application of Stack

1. Delimiter Matching

One application of the stack is in matching delimiters in a program. This is an important example because delimiter matching is part of any compiler: No program is considered correct if the delimiters are mismatched. In C++ programs, we have the following delimiters: parentheses “(” and “)”, square brackets “[” and “]”, curly brackets “{” and “}”, and comment delimiters “/*” and “ */”.
The delimiter matching algorithm reads a character from a C++ program and stores it on a stack if it is an opening delimiter. If a closing delimiter is found, the delimiter is compared to a delimiter popped off the stack. If they match, processing continues; if not, processing discontinues by signaling an error. The processing of the C++ program ends successfully after the end of the program is reached and the stack is empty.

2. Postfix Expression

A postfix expression is an expression in which each operator follows its operands. The advantage of this form is that there is no need to group sub-expressions in parentheses or to consider operator precedence. Stack can be used to evaluate a postfix (or prefix) expression.

Exercise

1. Please write a program which performs the following tasks:
    a. Make a left to right scan of the postfix expression
    b. If the element is an operand push it on Stack
    c. If the element is operator, evaluate it using as operands the correct number from stack and pushing the result onto the stack

2. A palindrome is a word, phrase, number, or another sequence of characters that reads the same backward and forwards. Can you determine if a given string, s, is a palindrome? Write a Program using stack for checking whether a string is palindrome or not.

3. Write a program using stacks which takes an expression as input and determines whether the delimiters are matched or not.

Lab 06 - Queues

Objective:

The objective of the lab is to develop understanding of the Queue data structure and its basicfunctions.

Description:

A queue is also a linear data structure which works according to FIFO (First IN First Out) manner. Of the two ends of the queue, one is designated as the front − where elements are extracted (operation called dequeue), and another is the rear, where elements are inserted (operation called enqueue). A queue may be depicted as in Figure 6.1. Queue operations are similar to stack operations. The following operations are needed to properly manage a queue:

- ✓ clear()—Clear the queue.
- ✓ isEmpty()—Check to see if the queue is empty.
- ✓ enqueue(el)—Put the element el at the end of the queue.
- ✓ dequeue()—Take the first element from the queue.
- ✓ firstEl()—Return the first element in the queue without removing it
- ✓ lastEl()—Return the last element in the queue without removing it

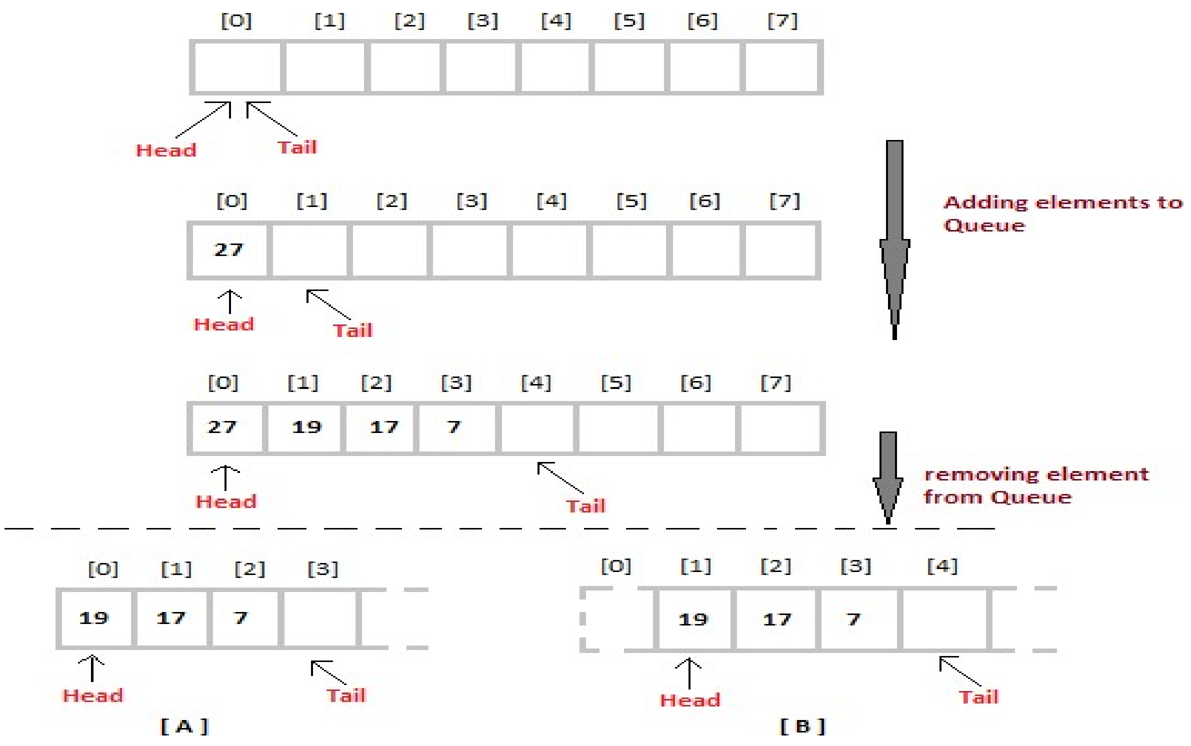


Figure 6.1 Queue and Operations on Queue

Example 01: Implementation of Queue using Array

```
using namespace std;
class Queue { // A class to represent a queue
  private:
    int front, rear, size;
    unsigned capacity;
    int* array;
   public:
      Queue(int siz){// constructor
      }
      void insert(int j){
      }
      int remove(){
      }
      int peek(){//equivalent to accessing first element
      }
      bool is Empty(){
      }
     bool isFull(){
     }
     int size(){
     }
};
```

Exercise

1. Write the definition of all the functions listed in Example 01.

2. Please implement the Generic Queue definition using singly linked list (you may use the Singly Linked List that you already developed in Lab # 3), you may also add any functions needed in the Singly Linked List definition given in Lab # 3. Your implementation should work for the main function given below.

```
#ifndef QUEUE_H
#define QUEUE_H
template<class DT> class Queue{
private:
    //include private variables according to the underlying data structure public:
    //contructor Queue();

    //puts element at the rear end of the Queue if it is not full. Must be O(1) void Put(DT
element);

   //if queue not empty then delete the element at front of the Queue. Must be O(1) DT Get();

   //return true if the Queue is empty and false if it is not bool IsEmpty();

   //return true if the Queue is full and false if it is not bool IsFull();
};
#endif

----------------------------------------------------------------------------------------------------------------

#include "Queue.h"
#include "Queue.cpp"
#include <iostream>
using namespace std;
int main(){
  Queue<int> *q =new Queue<int>();
  if(q->IsEmpty())
       cout<<"Queue is currently empty"<<endl;
  q->Put(1);
  q->Put(2);
  q->Put(3);
  while (!q->IsEmpty()){
      int value=q->Get(); cout<<value<<endl;
   }
return 0;
}
```

3. Give the C++ code to implement the Generic Queue using array. Please change the private datamembers as you are now using an array.

Lab 07 - Recursion

Objective:

The objective of the lab is to develop understanding of the recursion and its basic implementation.

Description:

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. Using a recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc. A recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems.

Properties of Recursion:

- Performing the same operations multiple times with different inputs.
- In every step, we try smaller inputs to make the problem smaller.
- Base condition is needed to stop the recursion otherwise infinite loop will occur.

Direct and indirect recursion

A function fun is called direct recursive if it calls the same function fun. A function fun is called indirect recursive if it calls another function say fun_new and fun_new calls fun directly or indirectly.

```
// An example of direct recursion
void directRecFun(){
   // Some code....
   directRecFun();
   // Some code...
}
// An example of indirect recursion
void indirectRecFun1(){
   // Some code...
   indirectRecFun2();
   // Some code...
}
void indirectRecFun2(){
   // Some code...
   indirectRecFun1();
   // Some code...
}
```

Example 01:

```
// A C++ program to demonstrate working of recursion
#include <bits/stdc++.h>
using namespace std;

void printFun(int test){
        if (test < 1)
                return;
        else {
                cout << test << " ";
                printFun(test - 1); // statement 2
                cout << test << " ";
                return;
        }
}
// Driver Code
int main(){
        int test = 3;
        printFun(test);
}
```

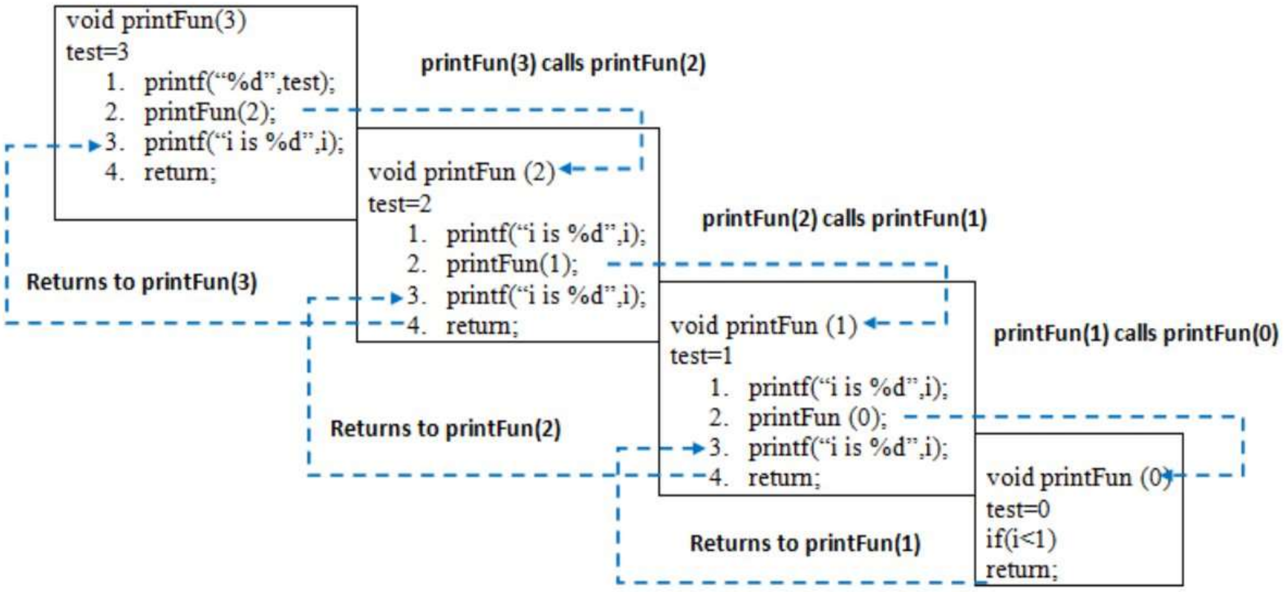


Figure 7.1 Recursion

Exercise

1. Write a program and recurrence relation to find the Fibonacci series of n where n>2.

2. Write a program and recurrence relation to find the Factorial of n where n>2.

3. Write a recursive function which will take input from the user until a special character (also selected by the user) is not entered. Then print all the input in reverse.
   Sample Input:
   Enter Special Character: !
   Enter Character: A
   Enter Character: B
   Enter Character: C
   Enter Character: !
   Sample Output: C B A

4. Write a recursive function which will raise a number (double) to a non-negative integer power n. The function receives the double value and integer as arguments.

5. Write a recursive method that for a positive integer n prints odd numbers
   a. between 1 and n
   b. between n and 1

Lab 08 - Advanced Sorting Techniques

Objective:

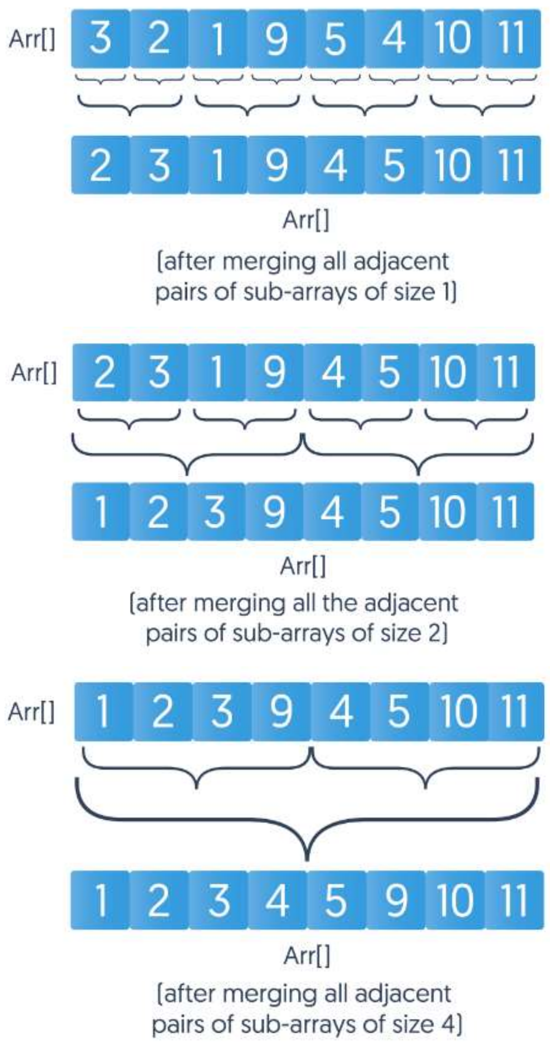
The objective of this lab is to implement two advanced sorting techniques: merge-sort and quicksort.

Description:

The sorting techniques described in Lab 2 were fairly simple. This lab will discuss how sorting can be done using other previously introduced data structures.

Merge-Sort:

Merge-sort algorithm uses the "divide and conquer" strategy which divides the problem into subproblems and solves those subproblems individually. These subproblems are then combined or merged to form a unified solution.



Arr[]
3 2 1 9 5 4 10 11
2 3 1 9 4 5 10 11
Arr[]
[after merging all adjacent pairs of sub-arrays of size 1]
Arr[]
2 3 1 9 4 5 10 11
1 2 3 9 4 5 10 11
Arr[]
[after merging all the adjacent pairs of sub-arrays of size 2]
Arr[]
1 2 3 9 4 5 10 11
1 2 3 4 5 9 10 11
Arr[]
[after merging all adjacent pairs of sub-arrays of size 4]

Figure 8.1 Working of Merge-Sort Algorithm

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

/* Function to merge the two haves arr[l..m] and arr[m+1..r] of array arr[] */
void merge(int arr[], int l, int m, int r);

// Utility function to find minimum of two integers
int min(int x, int y) { return (x<y)? x :y; }

/* Mergesort function to sort arr[0...n-1] */
void mergeSort(int arr[], int n) {
   int curr_size;  // For current size of subarrays to be merged. curr_size varies from 1 to n/2
   int left_start; // For picking starting index of left subarray to be merged

   // Merge subarrays in bottom-up manner.
   for (curr_size=1; curr_size<=n-1; curr_size = 2*curr_size) {
       // Pick starting point of different subarrays of current size
       for (left_start=0; left_start<n-1; left_start += 2*curr_size) {
           // Find ending point of left subarray. mid+1 is starting point of right
           int mid = min(left_start + curr_size - 1, n-1);
           int right_end = min(left_start + 2*curr_size - 1, n-1);
           // Merge Subarrays arr[left_start...mid] & arr[mid+1...right_end]
           merge(arr, left_start, mid, right_end);
       }
   }
}

/* Function to merge the two haves arr[l..m] and arr[m+1..r] of array arr[] */
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;
    j = 0;
    k = l;
    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++;
    }
}

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

/* Driver */
int main() {
    int arr[30], n;

    cout<<"Enter number of elements to be sorted (30 elements max.): ";
    cin>>n;
    cout<<"Enter "<<n<<" elements to be sorted:";
    for (int i = 0; i < n; i++) { cin>>arr[i];
    }
    mergeSort(arr, n);
     cout <<"\nSorted array is \n ";
    printArray(arr, n);
    return 0;
}
```

QuickSort:

Quicksort is an application of sorting using stacks. It also uses divide-and-conquer approach. In quicksort, a pivot element is selected and positioned in such a manner that all the elements smaller than the pivot element are to its left and all the elements greater than the pivot are to its right. The elements to the left and right of the pivot form two separate arrays and are sorted using this same approach. This continues until each subarray consists of a single element. The elements are merged to get the required sorted list. The pivot element can be the first, last, random, or median value in the given list.

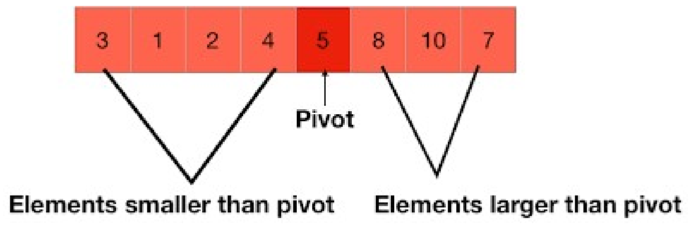


Figure 8.2 Division of elements after selection of pivot element

```
#include <iostream>
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 l, int h) {
    int x = arr[h];
    int i = (l - 1);
    for (int j = l; j <= h - 1; j++) {
        if (arr[j] <= x) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[h]);
    return (i + 1);
}

/* A[] --> Array to be sorted,
l --> Starting index,
h --> Ending index */
void quickSort(int arr[], int l, int h) {
    // Create an auxiliary stack
    int stack[h - l + 1];
    // initialize top of stack
    int top = -1;
    // push initial values of l and h to stack
    stack[++top] = l;
    stack[++top] = h;

    // Keep popping from stack while is not empty
    while (top >= 0) {
        // Pop h and l
        h = stack[top--];
        l = stack[top--];

    // Set pivot element at its correct position in sorted array
        int p = partition(arr, l, h);
// If there are elements on left side of pivot,
        // then push left side to stack
        if (p - 1 > l) {
            stack[++top] = l;
            stack[++top] = p - 1;
        }

        // If there are elements on right side of pivot,
        // then push right side to stack
        if (p + 1 < h) {
            stack[++top] = p + 1;
            stack[++top] = h;
        }
    }
}

// A utility function to print contents of arr
void printArr(int arr[], int n) {
    int i;
    for (i = 0; i < n; ++i)
        cout << arr[i] << " ";
}

/* Driver */
int main() {
    int arr[30], n;

    cout<<"Enter number of elements to be sorted (30 elements max.): ";
    cin>>n;
    cout<<"Enter "<<n<<" elements to be sorted:";
    for (int i = 0; i < n; i++) { cin>>arr[i];
    }

    quickSort(arr, 0, n-1);

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

Exercise

1. Write a program to implement a recursive version of merge-sort. Run it for some sample data.
2. Write a program to implement a recursive version of quicksort. Run it for some sample data.

Lab 09 - Binary Trees

Objective:

The objective of this lab is to implement a binary tree and apply traversal techniques.

Description:

Binary tree is a useful non-linear data structure to rapidly store and retrieve data. A binary tree is composed of parent nodes, or leaves, each of which stores data and also links to up to two other child nodes (leaves). As the name suggests, any parent node in a binary tree can have at most two child nodes.

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc.) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used methods for traversing trees:

InOrder(root) visits nodes in the following order:
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:
25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25

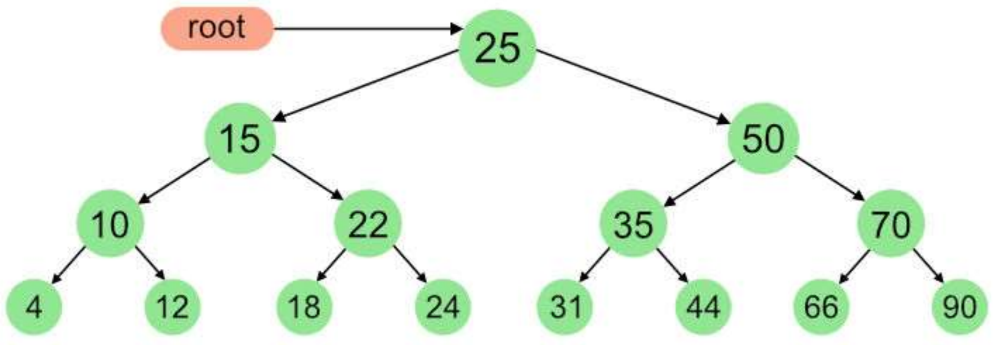


Figure 9.1 Binary Tree and Binary Tree Traversal

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

// A binary tree node has data, pointer to left child and a pointer to right child
struct Node {
    int data;
    struct Node *left, *right;
};

// Utility function to create a new tree node
Node* newNode(int data) {
    Node* temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

/* Given a binary tree, print its nodes in inorder*/
void printInorder(struct Node* node) {
    if (node == NULL)
        return;
     /* first recur on left child */
    printInorder(node->left);
     /* then print the data of node */
    cout << node->data << " ";
     /* now recur on right child */
    printInorder(node->right);
}

/* Driver code*/
int main()
{
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);


      // Function call
    cout << "\nInorder traversal of binary tree is \n";
    printInorder(root);
return 0;
}
```

Exercise:

1. Implement pre-order traversal using the above code.
2. Implement post-order traversal using the above code.
3. Modify the above program to create and traverse the binary tree based on user input.

Lab 10 - Binary Search Trees

Objective:

The objective of this lab is to apply search, insert, and delete operations on a binary search tree (BST).

Description:

Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers. The property that separates a binary search tree from a regular binary tree is that all nodes of left subtree are less than the root node, and all nodes of right subtree are more than the root node. In addition, both subtrees of each node are also BSTs.

- ✓ **Search:**

  If the value is below the root value, it cannot be in the right subtree; we need to only search in the left subtree and if the value is above the root value, it cannot be in the left subtree; we need to only search in the right subtree. If the value is not found, we eventually reach the left or right child of a leaf node which is NULL and it gets propagated and returned.

- ✓ **Insert:**

  Inserting a value in the correct position is similar to searching because we try to maintain the property of BST. We keep going to either right subtree or left subtree depending on the value and when we reach a point left or right subtree is null, we put the new node there.

- ✓ **Delete:**

  There are three cases for deleting a node from a binary search tree.

  - **Case a:**

    If the node to be deleted is the leaf node, simply delete the node from the tree.

  - **Case b:**

    If the node to be deleted has a single child node, first replace that node with its child node and then remove the child node from its original position.

  - **Case c:**

    If the node to be deleted has two children, first get the inorder successor of that node, then replace the node with the inorder successor, and finally remove the inorder successor from its original position.

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

struct nod {
  int info;
  struct nod *l;
  struct nod *r;
}*r;

class BST {
  public://functions declaration
  void search(nod *, int);
  void find(int, nod **, nod **);
  void insert(nod *, nod *);
  void del(int);
  void casea(nod *,nod *);
  void caseb(nod *,nod *);
  void casec(nod *,nod *);
  void show(nod *, int);
  BST() {
    r = NULL; }
};

void BST::find(int i, nod **par, nod **loc) {
  nod *ptr, *ptrsave;
  if (r == NULL) {
    *loc = NULL;
    *par = NULL;
    return;   }
  if (i == r->info) {
    *loc = r;
    *par = NULL;
    return;   }
  if (i < r->info)
    ptr = r->l;
  else
    ptr = r->r;
  ptrsave = r;
  while (ptr != NULL) {
    if (i == ptr->info) {
      *loc = ptr;
      *par = ptrsave;
      return;      }
    ptrsave = ptr;
    if (i < ptr->info)
      ptr = ptr->l;
    else
      ptr = ptr->r;
  }
  *loc = NULL;
  *par = ptrsave;
}
```

```
void BST::search(nod *root, int data) {
  int depth = 0;
  nod *temp = new nod;
  temp = root;
  while(temp != NULL) {
    depth++;
    if(temp->info == data) {
      cout<<"\nData found at depth: "<<depth<<endl;
      return;
    }
    else if(temp->info > data)
    temp = temp->l;
    else
    temp = temp->r;
  }
  cout<<"\n Data not found"<<endl;
  return;
}

void BST::insert(nod *tree, nod *newnode) {
  if (r == NULL) {
    r = new nod;
    r->info = newnode->info;
    r->l= NULL;
    r->r= NULL;
    cout<<"Root Node is Added"<<endl;
    return;
  }
  if (tree->info == newnode->info) {
    cout<<"Element already in the tree"<<endl;
    return; }
  if (tree->info > newnode->info) {
    if (tree->l != NULL)
      insert(tree->l, newnode);
    else {
      tree->l= newnode;
      (tree->l)->l = NULL;
      (tree->l)->r= NULL;
      cout<<"Node Added To Left"<<endl;
      return;
    }
  }
  else {
    if (tree->r != NULL)
      insert(tree->r, newnode)
    else {
      tree->r = newnode;
      (tree->r)->l= NULL;
      (tree->r)->r = NULL;
      cout<<"Node Added To Right"<<endl;
      return;
    }
  }
}
```

```
void BST::del(int i) {
  nod *par, *loc;
  if (r == NULL) {
    cout<<"Tree empty"<<endl;
    return;
  }
  find(i, &par, &loc);
  if (loc == NULL) {
    cout<<"Item not present in tree"<<endl;
    return;
  }
  if (loc->l == NULL && loc->r == NULL) {
    casea(par, loc);
    cout<<"item deleted"<<endl;
  }
  if (loc->l!= NULL && loc->r == NULL) {
    caseb(par, loc);
    cout<<"item deleted"<<endl;
  }
  if (loc->l== NULL && loc->r != NULL) {
    caseb(par, loc);
    cout<<"item deleted"<<endl;
  }
  if (loc->l != NULL && loc->r != NULL) {
    casec(par, loc);
    cout<<"item deleted"<<endl;
  }
  free(loc);
}

void BST::casea(nod *par, nod *loc) {
  if (par == NULL) {
  r= NULL;
  }
  else {
  if (loc == par->l)
  par->l = NULL;
  else
  par->r = NULL;
  }
}

void BST::caseb(nod *par, nod *loc) {
  nod *child;
  if (loc->l!= NULL)
    child = loc->l;
  else
    child = loc->r;
  if (par == NULL)
    r = child;
  else {
    if (loc == par->l)
      par->l = child;
    else
      par->r = child;
  }
}

void BST::casec(nod *par, nod *loc) {
  nod *ptr, *ptrsave, *suc, *parsuc;
  ptrsave = loc;
  ptr = loc->r;
  while (ptr->l!= NULL) {
    ptrsave = ptr;
    ptr = ptr->l;
  }
  suc = ptr;
  parsuc = ptrsave;
  if (suc->l == NULL && suc->r == NULL)
    casea(parsuc, suc);
  else caseb(parsuc, suc);
  if (par == NULL)
    r = suc;
  else {
    if (loc == par->l)
      par->l = suc;
    else
      par->r= suc;
  }
  suc->l = loc->l;
  suc->r= loc->r;
}

//print the tree
void BST::show(nod *ptr, int level) {
  int i;
  if (ptr != NULL) {
    show(ptr->r, level+1);
    cout<<endl;
    if (ptr == r)
      cout<<"Root->: ";
    else {
      for (i = 0;i < level;i++)
      cout<<" ";
    }
    cout<<ptr->info;
    show(ptr->l, level+1);
  }
}
```

```
int main() {
  int c, n,item;
  BST bst;
  nod *t;
  while (1) {
    cout<<"1.Insert Element "<<endl;
    cout<<"2.Delete Element "<<endl;
    cout<<"3.Search Element"<<endl;
    cout<<"4.Display the tree"<<endl;
    cout<<"5.Quit"<<endl;
    cout<<"Enter your choice : ";
    cin>>c;
    switch(c) {
      case 1:
        t = new nod;
         cout<<"Enter the number to be inserted: ";
        cin>>t->info;
        bst.insert(r, t);
        break;


      case 2:
        if (r == NULL) {
          cout<<"Tree is empty, nothing to delete"<<endl;
          continue; }
        cout<<"Enter the number to be deleted: ";
        cin>>n;
        bst.del(n);
        break;
      case 3:
        cout<<"Search:"<<endl;
        cin>>item;
        bst.search(r,item);
        break;
      case 4:
        cout<<"Display BST:"<<endl;
        bst.show(r,1);
        cout<<endl;
        break;
      case 5:
        exit(1);
      default:
        cout<<"Wrong choice!"<<endl;
    }
  }
}
```

Exercise:

1. Analyze the given code to find the difference between “find” and “search” methods.
2. Merge “find” and “search” methods into a single method by necessary modifications in the given code and execute it.

Lab 11 - Heap

Objective:

The objective of this lab is to apply operations on a heap.

Description:

Heap data structure is a complete binary tree that satisfies the heap property, where any given node is either

- ✓ Always greater than its child node(s) and the value of the root node is the largest among all other nodes. This property is also called max heap property. OR
- ✓ Always smaller than the child node(s) and the value of the root node is the smallest among all other nodes. This property is also called min heap property.

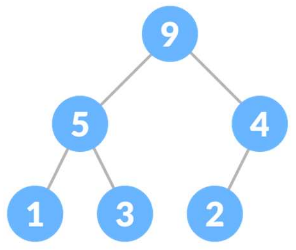


Figure 11.1 Max Heap

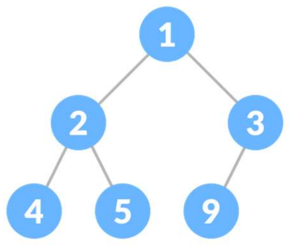


Figure 11.2 Min Heap

Some of the important operations performed on a heap are:

- ✓ **Heapify:**
  Heapify is the process of creating a heap data structure from a binary tree. It is used to create a Min-Heap or a Max-Heap.
- ✓ **Insert:**
  Insert the new element at the end of the tree. Then heapify it.
- ✓ **Delete:**
  Swap the element to be deleted with the last element. Remove the last element and heapify the tree.
- ✓ **Peek (find max/min):**
  Returns the maximum element from Max Heap or minimum element from Min Heap without deleting the node. (Note that the root is max value in max heap and vice versa).
- ✓ **Extract max/min:**
  Returns the node with maximum value after removing it from a Max Heap whereas Extract-Min returns the node with minimum after removing it from Min Heap.

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

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

void heapify(vector<int> &hT, int i) {
  int size = hT.size();
  int largest = i;
  int l = 2 * i + 1;
  int r = 2 * i + 2;
  if (l < size && hT[l] > hT[largest])
    largest = l;
  if (r < size && hT[r] > hT[largest])
    largest = r;
  if (largest != i) {
    swap(&hT[i], &hT[largest]);
    heapify(hT, largest);
  }
}

void insert(vector<int> &hT, int newNum) {
  int size = hT.size();
  if (size == 0)
    hT.push_back(newNum);
  else {
    hT.push_back(newNum);
    for (int i = size / 2 - 1; i >= 0; i--)
      heapify(hT, i);
  }
}
```

```
void deleteNode(vector<int> &hT, int num) {
  int size = hT.size();
  int i;
  for (i = 0; i < size; i++) {
    if (num == hT[i])
      break; }
  swap(&hT[i], &hT[size - 1]);
  hT.pop_back();
  for (int i = size / 2 - 1; i >= 0; i--)
    heapify(hT, i);
}

void printArray(vector<int> &hT) {
  for (int i = 0; i < hT.size(); ++i)
    cout << hT[i] << " ";
  cout << "\n";
}

int main() {
  vector<int> heapTree;
  insert(heapTree, 3);
    insert(heapTree, 4);
  insert(heapTree, 9);
  insert(heapTree, 5);
  insert(heapTree, 2);
    cout << "Max-Heap array: ";
    printArray(heapTree);
  deleteNode(heapTree, 4);
    cout << "After deleting an element: ";
    printArray(heapTree);
}
```

Exercise:

1. The above code is for Max Heap. Implement a Min Heap and then insert and delete elements.
2. Implement peek and extract operations for both max and min values.

Lab 12 - Hashing

Objective:

The objective of this lab is to implement hashing technique.

Description:

Hashing is a technique using which, we can map a large amount of data to a smaller table, called "hash table," using a "hash function." By using hashing, we can search the data more quickly and efficiently when compared to other searching techniques like linear and binary search.

The value is converted into a unique integer key or hash by using a hash function. It is used as an index to store the original element, which falls into the hash table. The element from the data array is stored in the hash table where it can be quickly retrieved using the hashed key.

```
hash = hash_func(key);
index = hash % array_size;
```

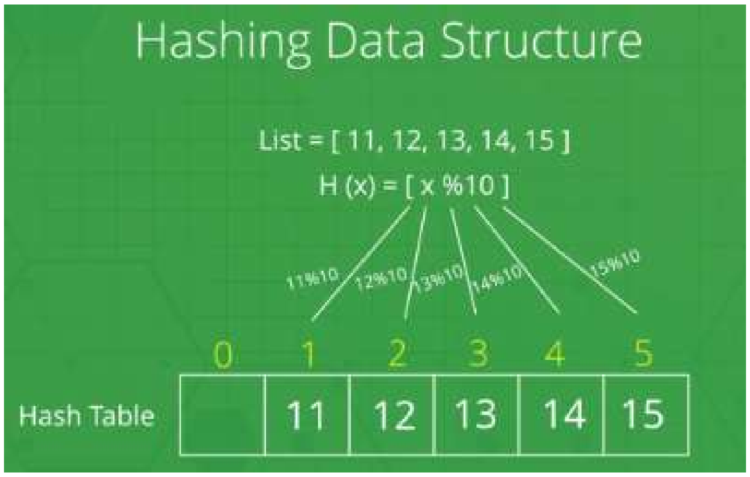


Figure 12.1 Hashing

It must be noted that there are high chances of collision as there could be 2 or more keys that compute the same hash code resulting in the same index of elements in the hash table. A collision cannot be avoided in case of hashing even if we have a large table size. We can prevent a collision by choosing the good hash function and the implementation method.

Collision Resolution Techniques:

- ✓ Separate Chaining (Open Hashing)
- ✓ Linear Probing (Open Addressing/Closed Hashing)
- ✓ Quadratic Probing
- ✓ Double Hashing

The following code implements Open Hashing which is a simple technique.

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

class HashMapTable {
// size of the hash table
int table_size;
// Pointer to an array containing the keys
list<int> *table;
public:
   // creating constructor of the above class containing all the methods
   HashMapTable(int key);
   // hash function to compute the index using table_size and key
   int hashFunction(int key) {
   return (key % table_size);
}
// inserting the key in the hash table
void insertElement(int key);
// deleting the key in the hash table
void deleteElement(int key);
// displaying the full hash table
void displayHashTable();
};

//creating the hash table with the given table size
HashMapTable::HashMapTable(int ts) {
   this->table_size = ts;
   table = new list<int>[table_size];
}

// insert function to push the keys in hash table
void HashMapTable::insertElement(int key) {
   int index = hashFunction(key);
   table[index].push_back(key);
}

// delete function to delete the element from the hash table
void HashMapTable::deleteElement(int key) {
   int index = hashFunction(key);
   // finding the key at the computed index
   list <int> :: iterator i;
   for (i = table[index].begin(); i != table[index].end(); i++) {
      if (*i == key)
      break;
   }
   // removing the key from hash table if found
   if (i != table[index].end())
      table[index].erase(i);
}
```

```
// display function to showcase the whole hash table
void HashMapTable::displayHashTable() {
   for (int i = 0; i<table_size; i++) {
   cout<<i;
   // traversing at the recent/ current index
   for (auto j : table[i])
      cout<< " ==> " << j;
   cout<<endl;
   }
}

// Main function
int main() {
   // array of all the keys to be inserted in hash table
   int arr[] = {20, 34, 56, 54, 76, 87};
   int n = sizeof(arr)/sizeof(arr[0]);
   // table_size of hash table as 6
   HashMapTable ht(6);
   for (int i = 0; i< n; i++)
      ht.insertElement(arr[i]);
   // displaying hash table
   ht.displayHashTable();
   cout<<endl;
   // deleting element 34 from the hash table
   ht.deleteElement(34);
   // displaying the final data of hash table
   ht.displayHashTable();
   return 0;
}
```

Exercise:

1. Implement a hashing algorithm using Linear Probing.
2. Implement a hashing algorithm using Quadratic Probing.
3. Implement a hashing algorithm using Double Hashing.

Lab 13 - Graph Traversal

Objective:

The objective of this lab is to implement graph traversal methods.

Description:

Graph is a non-linear data structure defined as a collection of vertices and edges. In other words, a graph data structure is a collection of nodes that have data and are connected to other nodes. Every relationship or connection is an edge from one node to another.

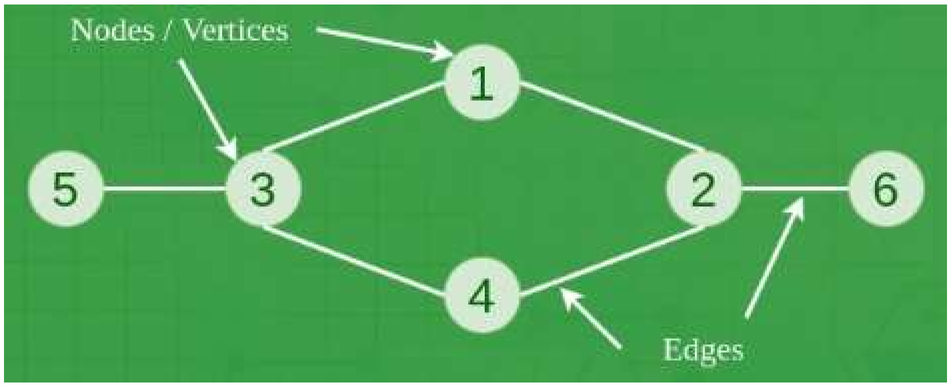
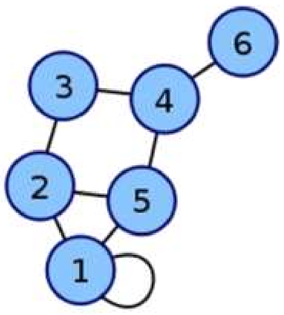


Figure 13.1 Representation of a Graph

Graphs can be represented using:

Adjacency Matrix:

An adjacency matrix is a means of representing which vertices (or nodes) of a graph are adjacent to which other vertices. Another matrix representation for a graph is the incidence matrix. Specifically, the adjacency matrix of a finite graph *g* on *n* vertices is the $n \times n$ matrix where the non-diagonal entry a_{ij} is the number of edges from vertex *i* to vertex *j*, and the diagonal entry a_{ii}, depending on the convention, is either once or twice the number of edges (loops) from vertex *i* to itself. Undirected graphs often use the latter convention of counting loops twice, whereas directed graphs typically use the former convention.



$$\begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Figure 13.2 Adjacency Matrix of a Graph

Adjacency List:

An adjacency list is the representation of all edges or arcs in a graph as a list. If the graph is undirected, every entry is a set (or multiset) of two nodes containing the two ends of the corresponding edge; if it is directed, every entry is a tuple of two nodes, one denoting the source node and the other denoting the destination node of the corresponding arc. Typically, adjacency lists are unordered.

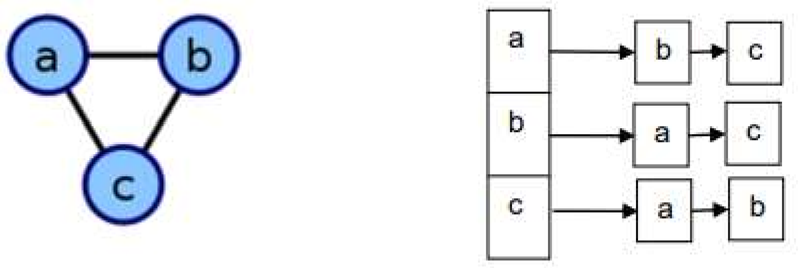


Figure 13.3 A representation of Adjacency List of a Graph

Graph traversal is mainly carried out using the following techniques.

✓ **Breadth-First Search (BFS):**

Breadth-first search (BFS) is a popular technique to traverse a graph. BFS starts with the root node or an arbitrary node and explores the neighbor nodes first, before moving to the next level neighbors. A standard BFS implementation puts each vertex of the graph into one of two categories: visited and not visited. The purpose is to mark each vertex as visited while avoiding cycles.

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node.

✓ **Depth-First Search (DFS):**

Depth-first search (DFS) is another technique used to traverse a graph. DFS starts with the root node or a start node and then explores the adjacent nodes of the current node by going deeper into the graph or a tree. This means that in DFS the nodes are explored depth-wise until a node with no children is encountered. Once the leaf node is reached, DFS backtracks and starts exploring some more nodes in a similar fashion.

The edges that lead us to unexplored nodes are called “discovery edges” while the edges leading to already visited nodes are called “block edges.”

```
// BFS algorithm in C++

#include <iostream>
#include <list>

using namespace std;

class Graph {
  int numVertices;
  list<int>* adjLists;
  bool* visited;

  public:
  Graph(int vertices);
  void addEdge(int src, int dest);
  void BFS(int startVertex);
};

// Create a graph with given vertices,
// and maintain an adjacency list
Graph::Graph(int vertices) {
  numVertices = vertices;
  adjLists = new list<int>[vertices];
}

// Add edges to the graph
void Graph::addEdge(int src, int dest) {
  adjLists[src].push_back(dest);
  adjLists[dest].push_back(src);
}
// BFS algorithm
void Graph::BFS(int startVertex) {
  visited = new bool[numVertices];
  for (int i = 0; i < numVertices; i++)
    visited[i] = false;

  list<int> queue;

  visited[startVertex] = true;
  queue.push_back(startVertex);

  list<int>::iterator i;

  while (!queue.empty()) {
    int currVertex = queue.front();
    cout << "Visited " << currVertex << " ";
    queue.pop_front();

    for (i = adjLists[currVertex].begin(); i != adjLists[currVertex].end(); ++i) {
      int adjVertex = *i;
      if (!visited[adjVertex]) {
        visited[adjVertex] = true;
        queue.push_back(adjVertex);
      }
    }
  }
}

int main() {
  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);

  g.BFS(2);

  return 0;
}
```

Exercise:

1. Implement Depth-First Search (DFS) algorithm.
2. Implement graph traversal methods using adjacency matrix instead of adjacency list.