Course Code: CSE 1302
Course Title: Data Structure Lab

LABORATORY MANUAL

Department of Computer Science and Engineering
University of Liberal Arts Bangladesh
September 2023

Preface

This manual is designed to help you achieve the course outcomes of demonstrating various basic data structures and their operations, applying appropriate data structures for solving real-world problems, and developing applications using various data structures. In this manual, you will find a variety of experiments and exercises that will help you develop your skills in data structure. Data structure is a systematic way to organize data in order to use it efficiently. It is a fundamental concept in computer science and is widely used in real-world applications. This lab manual is designed to help you understand the principles of data structure and apply them to solve real-world problems.

The experiments in this manual cover a range of topics, including arrays, linked lists, stacks, queues, trees, and graphs. Each experiment is designed to help you develop your skills in data structure and apply them to real-world problems. The manual also includes exercises that will help you reinforce your understanding of the concepts covered in each experiment. We hope that this lab manual will be a valuable resource for you as you learn data structure. We encourage you to work through each experiment and exercise carefully and to seek help from your instructor or teaching assistant if you have any questions or difficulties. Good luck and happy programming!

Course Objectives:

The course aims to provide exposure to problem-solving through programming. It aims to train the student to the basic concepts of the C-programming language. This course involves a lab component which is designed to give the student hands-on experience with the concepts that develop problem solving and coding skills.

Course Outcome:

CO No.	CO Statement	Domain/level of learning Taxonomy	Assessment tools
1	Explain fundamental concepts of various Data Structures	Cognitive / L2, Affective / L2	Open Ended Lab / Final Term Exam
2	Apply the fundamental concepts of various Data Structures	Cognitive / L2, Affective / L2	Open Ended Lab / Final Term Exam

Text and Reference Materials

Text Book(s): Schaum's Outline of Programming with C, 2nd Edition, by Byron S Gottfried

Reference Material/Book(s):

Teach Yourself C, 3rd Edition, by Herbert Schildt

Online Resources:

http://www.tutorialspoint.com/cprogramming/

https://www.w3schools.in/c-tutorial/

Lab Instructions

- All students must arrive on time for the class.
- Just after entering and before class started, all students were required to submit their lab reports (if any).
- All computers must be handled with care.
- Use only the software needed for the course.
- No haste should be done during the class period.
- After a class period, all computers must be shut down.
- Perfect discipline should be maintained and useless talking should be avoided while performing LAB.
- Food and Drinks are not allowed in the LAB.
- LAB will be used for academic purposes only.

Contents

Introduction to Data Structure:

A data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently. A data structure is a special format for organizing and storing data. General data structure types include arrays, files, linked lists, stacks, queues, trees, graphs and so on.

Depending on the organization of the elements, data structures are classified into two types:

1) Linear Data structure: Elements are accessed in a sequential order but it is not compulsory to store all elements sequentially. Examples: Arrays, Linked lists, Stacks and queues.
2) Non-Linear data structure: Elements of this data structure are stored/accessed in a non-linear order. Examples: Trees and Graphs.

Objectives of Learning Data Structure:

- Be able to acquire the knowledge of basic data structures and their implementations.
- Be able to develop skills to apply appropriate data structures in problem solving.
- Have practical knowledge on the applications of data structures

Operations on Data Structures:

Different operations that can be performed on the various data structures are included below:

Traversing: It means to access each data item exactly once so that it can be processed. For example, to print the names of all the students in a class.

Searching: It is used to find the location of one or more data items that satisfy the given constraint. Such a data item may or may not be present in the given collection of data items. For example, to find the names of all the students who secured 100 marks in mathematics.

Inserting: It is used to add new data items to the given list of data items. For example, to add the details of a new student who has recently joined the course.

Deleting: It means to remove (delete) a particular data item from the given collection of data items. For example, to delete the name of a student who has left the course.

Sorting: Data items can be arranged in some order like ascending order or descending order depending on the type of application. For example, arranging the names of students in a class in an alphabetical order, or calculating the top three winners by arranging the participants' scores in descending order and then extracting the top three.

Merging: Lists of two sorted data items can be combined to form a single list of sorted data items.

Lab Task 1: Arrays

Objectives:

- Be able to declare and initialize arrays
- Be able to understand different kinds of array operations such as access an array element, insert an element to an array, delete an element from array and merge two arrays
- Have a basic idea about Pseudocode

Theory

Array: An array is a user-defined data type that stores related information together. All the information stored in an array belongs to the same data type.

Types of Arrays:

Arrays can be classified into two categories:

1. One dimensional array
2. Two dimensional arrays

Let's start with a discussion of one-dimensional array as follows:

Declaration of Arrays:

Arrays are declared using the following syntax:

type name[size];

The type can be either int, float, double, char, or any other valid data type. The number within brackets indicates the size of the array, i.e., the maximum number of elements that can be stored in the array. For example, if we write,

int marks[10];

then the statement declares marks to be an array containing 10 elements. In C, the array index starts from zero. The first element will be stored in marks[0], second element in marks[1], and so on. Therefore, the last element, that is the 10th element, will be stored in marks[9]. Note that 0, 1, 2, 3 written within square brackets are the subscripts. In the memory, the array will be stored as shown in Fig. 1.

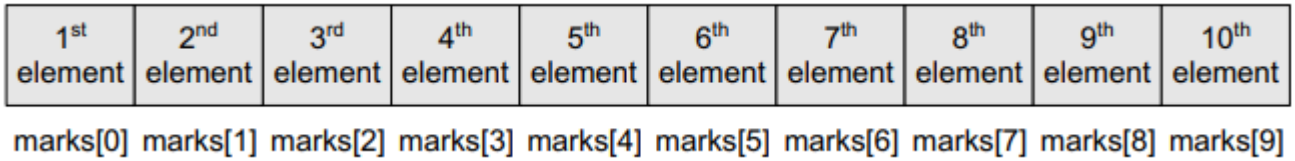


Fig. 1. Memory representation of an array of 10 elements

Operations on Array:

There are a number of operations that can be performed on arrays. These operations include:

- Traversing an array
- Inserting an element in an array

- Deleting an element from an array
- Merging two arrays

Traversing an Array:

Traversing an array means accessing each and every element of the array for a specific purpose.

C code to read and display *n* numbers using an array.

```
#include <stdio.h>
#include <conio.h>
int main()
{
int i, n, arr[20];

printf("\n Enter the number of elements in the array : ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\n arr[%d] = ", i);
scanf("%d",&arr[i]);
}
printf("\n The array elements are: ");
for(i=0;i<n;i++)
   printf("\t %d", arr[i]);

   return 0;
}
```

Output

Enter the number of elements in the array: 5

```
 arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
arr[4] = 5
The array elements are:          1      2      3      4      5
```

Inserting an Element to an Array

Pseudocode to Insert an Element in the Middle of an Array

```
 Step 1: [INITIALIZATION] SET I = N

 Step 2: Repeat Steps 3 and 4 while I >=
POS Step 3:    SET A[I + 1] = A[I]

         Step 4:
               SET I = I – 1
        [END OF LOOP]

 Step 5: SET N = N + 1
Step 6: SET A[POS] = VAL
Step 7: EXIT
```

C code to insert an element in the middle of an array

```
#include <stdio.h>
int main()
{
int i, n, num, pos, arr[10];
printf("\n Enter the number of elements in the array : ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\n arr[%d] = ", i);
scanf("%d", &arr[i]);
}
printf("\n Enter the number to be inserted : ");
scanf("%d", &num);
printf("\n Enter the position at which the number has to be added :");
scanf("%d", &pos);
for(int i=n-1;i>=pos;i--){
  arr[i+1] = arr[i];
}
arr[pos] = num;
n = n+1;
printf("\n The array after insertion of %d is : ", num);
for(i=0;i<n;i++){
  printf("\n arr[%d] = %d", i, arr[i]);
}
return 0;
}
```

Lab Task 2: Deleting an Element from an Array

Objectives:

Theory:

Pseudocode to delete an Element from the Middle of an Array

```
  Step 1: [INITIALIZATION] SET I = POS

  Step 2: Repeat Steps 3 and 4 while I <= N –
1 Step 3:        SET A[I] = A[I + 1]

         Step 4:          SET
       I = I + 1 [END OF
       LOOP]

  Step 5: SET N = N – 1
Step 6: EXIT
```

C code to delete an element from an array

```
#include <stdio.h>
int main()
{
int i, n, pos, arr[10];
printf("\n Enter the number of elements in the array : "); scanf("%d", &n);
for(i=0;i<n;i++)
{
  printf("\n arr[%d] = ", i);
  scanf("%d", &arr[i]);
}
printf("\nEnter the position from which the number has to be deleted : "); scanf("%d", &pos);
for(i=pos; i<n-1;i++){
   arr[i] = arr[i+1];
}
n--;
printf("\n The array after deletion is : ");
for(i=0;i<n;i++){
   printf("\n arr[%d] = %d", i, arr[i]);
}
return 0;
}
```

Merging two arrays:

Merging two arrays in a third array means first copying the contents of the first array into the third array and then copying the contents of the second array into the third array. Hence, the merged array contains the contents of the first array followed by the contents of the second array.

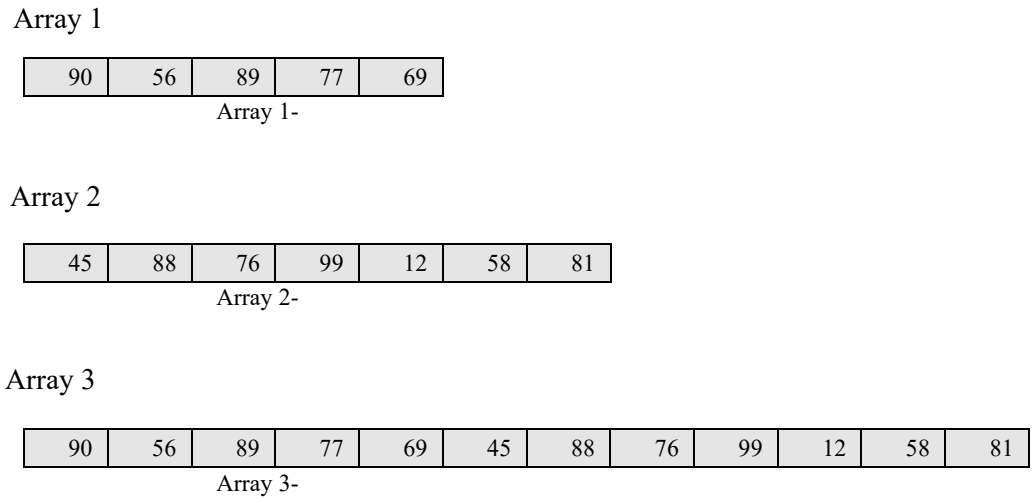


Fig. 2. Merging of two unsorted arrays

C code to merge two arrays:

```
#include <stdio.h>
int main()
{
int arr1[10], arr2[10], arr3[20];
int i, n1, n2, m, index=0;
printf("\n Enter the number of elements in array1 : ");
scanf("%d", &n1);
printf("\n\n Enter the elements of the first array");
for(i=0;i<n1;i++)
{
printf("\n arr1[%d] = ", i);
scanf("%d", &arr1[i]);
}
printf("\n Enter the number of elements in array2 : ");
scanf("%d", &n2);
printf("\n\n Enter the elements of the second array");
for(i=0;i<n2;i++)
{
```

```
printf("\n arr2[%d] = ", i);
scanf("%d", &arr2[i]);
}
m = n1+n2;
for(i=0;i<n1;i++)
{
arr3[index] = arr1[i]; index++;
}
for(i=0;i<n2;i++)
{
arr3[index] = arr2[i]; index++;
}
printf("\n\n The merged array is");
for(i=0;i<m;i++)
printf("\n arr[%d] = %d", i, arr3[i]);

return 0;
}
```

Two-dimensional arrays:

Till now, we have only discussed one-dimensional arrays. One-dimensional arrays are organized linearly in only one direction. But at times, we need to store data in the form of grids or tables. Here, the concept of single-dimension arrays is extended to incorporate two-dimensional data structures. A two-dimensional array is specified using two subscripts where the first subscript denotes the row and the second denotes the column. The C compiler treats a two-dimensional array as an array of one-dimensional arrays. Figure 3. Shows a two-dimensional array which can be viewed as an array of arrays.

Fig. 3. Two-dimensional array

Operations on two-dimensional arrays:

Transpose: Transpose of an $m\ x\ n$ matrix A is given as a $n\ x\ m$ matrix B, where $B_{i,j} = A_{i,j}$.

Sum: Two matrices that are compatible with each other can be added together, storing the result in the third matrix. Two matrices are said to be compatible when they have the same number of rows and columns. The elements of two matrices can be added by writing:

$$C_{i,j} = A_{i,j} + B_{i,j}$$

Declaring Two-dimensional Arrays:

Any array must be declared before being used. The declaration statement tells the compiler the name of the array, the data type of each element in the array, and the size of each dimension. A two-dimensional array is declared as:

data_type array_name[row_size][column_size];

int marks[3][5];

In the above statement, a two-dimensional array called marks has been declared that has 3 rows and 5 columns.

Accessing the Elements of Two-dimensional Arrays

C code to read and display a 3 x 3 matrix.

```
#include <stdio.h>
int main()
{
int i, j, mat[3][3];
printf("\n Enter the elements of the matrix ");
for(i=0;i<3;i++){
for(j=0;j<3;j++){
   scanf("%d",&mat[i][j]);
  }
}
printf("\n The elements of the matrix are ");
for(i=0;i<3;i++){
  printf("\n");
  for(j=0;j<3;j++){
      printf("\t %d",mat[i][j]);
  }
}

return 0;
}
```

C code to transpose a 3x3 matrix:

```
#include <stdio.h>
int main(){
int i, j, mat[3][3], transposed_mat[3][3];
printf("\n Enter the elements of the matrix ");
for(i=0;i<3;i++){
for(j=0;j<3;j++){
   scanf("%d", &mat[i][j]);
  }
}
printf("\n The elements of the matrix are ");
for(i=0;i<3;i++){
   printf("\n");
   for(j=0;j<3;j++){
   printf("\t %d", mat[i][j]);
   }
}
for(i=0;i<3;i++){
  for(j=0;j<3;j++){
    transposed_mat[i][j] = mat[j][i];
 }
}
printf("\n The elements of the transposed matrix are ");
for(i=0;i<3;i++){
  printf("\n");
  for(j=0;j<3;j++){
  printf("\t %d",transposed_mat[i][j]);
  }
}
return 0;
}
```

Lab Task 3: Searching Algorithms

Objectives

- To be able to implement Linear Search
- To be able to implement Binary Search
- To be able to implement Bubble Sort
- To be able to implement Selection Sort
- To be able to implement Insertion Sort

Theory

Searching Algorithms

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

Sequential Search: In this, the list or array is traversed sequentially and every element is checked. For example: Linear Search.

Interval Search: These algorithms are specifically designed for searching in sorted data-structures. This type of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half. For Example: Binary Search.

Linear Search

- A linear search scans one item at a time, without jumping to any item.
- The worst case complexity is O(n), sometimes known an O(n) search
- Time taken to search elements keeps increasing as the numbers of elements are increased.

Pseudocode for Linear Search

```
function linear_search (array, searchingForValue)
        for each item in the array
            if  item == searchingForValue
                return the item's location
            end if
         end for
end function
```

C Code to Implement Linear Search

```
#include <stdio.h>
int SearchItem(int arr[], int n, int x){
for (int i = 0; i < n; i++){
    if (arr[i] == x){
       return i;
    }
}
       return -1;
}

int main()
{
int n,x,i;
printf("Enter size of array: ");
scanf("%d", &n);
int arr[n];
printf("Enter Array Elements: ");
for(i=0;i<n;i++){
   scanf("%d", &arr[i]);
}
printf("Enter Item to search for: ");
scanf("%d", &x);

int result = SearchItem(arr, n, x);
if (result == -1){
printf("Value is not found in the array");
}
else{
printf("Value is found at location: %d", result);
}
return 0;
}
```

Binary Search Algorithm:

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, **the data collection should be in the sorted form.** Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub- array as well until the size of the subarray reduces to zero.

Pseudocode for Binary Search

```
// initially called with low = 0, high = N – 1
  BinarySearch(A[0..N-1], item, low, high) {

  while(low<=high){
       mid = (low + high) / 2
  if (A[mid]==item){
     return mid;
   }
  else if(A[mid]<item){
        low=mid+1
  }
  else{
       high=mid-1
  }

 }
```

C Code to Implement Binary Search

```
#include <stdio.h>
int binarySearch(int array[], int x, int low, int high) {
  while (low <= high) {
    int mid = low + (high - low) / 2;

    if (array[mid] == x)
      return mid;

    if (array[mid] < x)
      low = mid + 1;

    else
```

```
      high = mid - 1;
  }

  return -1;
}

int main() {
int n,x,i;
printf("Enter size of array: ");
scanf("%d", &n);
int arr[n];
printf("Enter Array Elements: ");
for(i=0;i<n;i++){
   scanf("%d", &arr[i]);
}
printf("Enter Item to search for: ");
scanf("%d", &x);

int result = binarySearch(arr, x, 0, n-1);

   if (result == -1){
    printf("Element is not found in the array");
  }
  else{
    printf("Element is found at index %d", result);
  }
  return 0;
   }
```

Lab Task 4 and 5: Sorting Algorithms

Objectives:

Theory:

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

Example:

Input: 7, 6, 1, 5, 3, 4, 9, 5

Output: 1, 3, 4, 5, 5, 6, 7, 9

Bubble Sort:

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst-case complexity are of $O(n^2)$ where n is the number of items.

Pseudocode for Bubble Sort

```
function BubbleSort(array)
        for all elements of array
           if array[i] > array[i+1]
              swap(list[i], list[i+1])
           end if
        end for
      return list
  end BubbleSort
```

C Code to Implement Bubble Sort

```
#include <stdio.h>
void BubbleSort(int arr[], int n){
int i, j;
for (i = 0; i < n-i-1; i++){
   for (j = 0; j < n-1; j++){
      if (arr[j] > arr[j+1]){
        int swap=arr[j];
        arr[j]=arr[j+1];
        arr[j+1]=swap;
      }
   }
  }
}
```

```
int main() {
int n,x;
printf("Enter size of array: ");
scanf("%d", &n);
int arr[n];
printf("Enter Array Elements: ");
for(int i=0;i<n;i++){
  scanf("%d", &arr[i]);
}

BubbleSort(arr, n);
printf("Sorted Elements: ");
for(int i=0;i<n;i++){
  printf("%d ", arr[i]);
}

 return 0;
}
```

Selection Sort

The Selection sort algorithm is based on the idea of finding the minimum or maximum element in an unsorted array and then putting it in its correct position in a sorted array.

- Concept for sorting in ascending order:
  - Locate smallest element in array. Exchange it with element in position 0
  - Locate next smallest element in array. Exchange it with element in position 1.
  - Continue until all elements are arranged in order

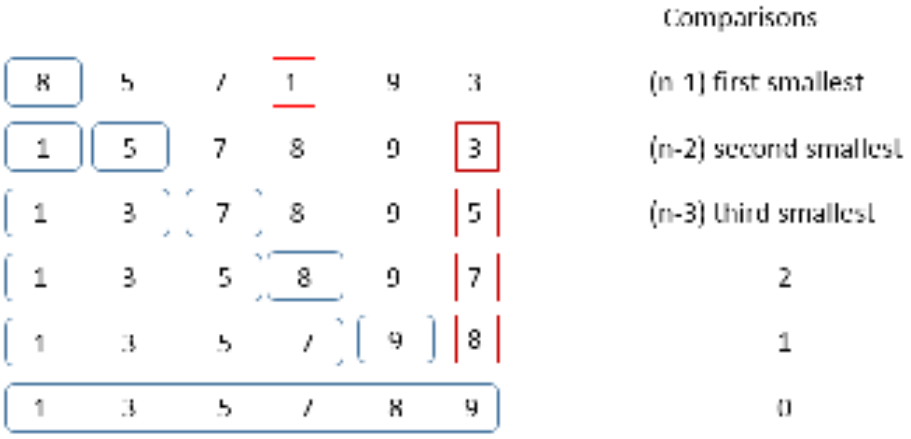


Fig. 4. Selection Sort

Pseudocode for Selection Sort

```
void selectionSort(int array[], int n)
 {
   int select, minIndex, minValue;

   for (select = 0; select < (n - 1); select++)
   { //select the location and find the minimum value
     minIndex = select;


 for(int i = select + 1; i < n; i++)
     { //start from the next of selected one to find minimum
         if (array[i] < array[minIndex])
       {
           minIndex = i;
       }
     }
     swap(array[minIndex], array[select])
   }
}
```

C Code to Implement Selection Sort

```
#include<stdio.h>
void selectionSort(int arr[], int n)
 {
   int select, minIndex, minValue;

   for (select = 0; select < (n - 1); select++)
   {
     minIndex = select;

     for(int i = select + 1; i < n; i++)
     {
         if (arr[i] < arr[minIndex])
       {
           minIndex = i;
       }
     }
     int tmp;
     tmp=arr[minIndex];
     arr[minIndex] = arr[select];
     arr[select] = tmp;
   }
 }
```

```
int main() {
int n,x;
printf("Enter size of array: ");
scanf("%d", &n);
int arr[n];
printf("Enter Array Elements: ");
for(int i=0;i<n;i++){
   scanf("%d", &arr[i]);
}

selectionSort(arr, n);
printf("Sorted Elements: ");
for(int i=0;i<n;i++){
   printf("%d ", arr[i]);
}

  return 0;
}
```

Insertion Sort

Insertion sort is based on the idea that one element from the input elements is consumed in each iteration to find its correct position i.e, the position to which it belongs in a sorted array. It iterates the input elements by growing the sorted array at each iteration. It compares the current element with the largest value in the sorted array. If the current element is greater, then it leaves the element in its place and moves on to the next element else it finds its correct position in the sorted array and moves it to that position. This is done by shifting all the elements, which are larger than the current element, in the sorted array to one position ahead.

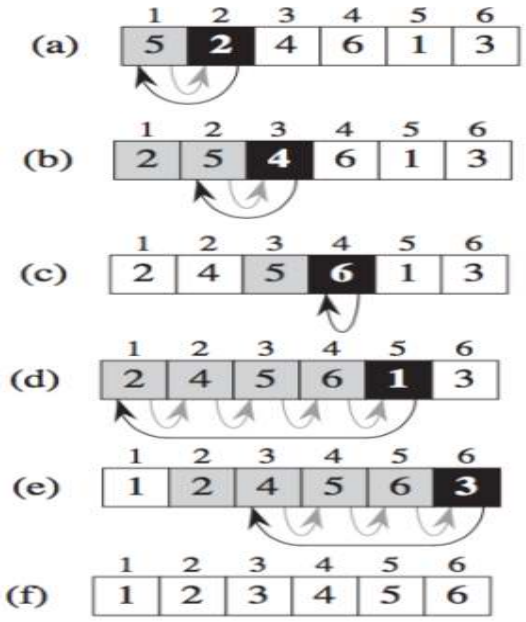


Fig. 5: Insertion Sort

Pseudocode for Insertion Sort

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted
           sequence A[1 . . j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

C Code to Implement Insertion Sort

```
#include<stdio.h>
void insertionSort(int arr[], int n){
  int i, key, j;
for (j = 1; j < n; j++){
    key = arr[j];
    i = j - 1;

while (i >= 0 && arr[i] > key){
      arr[i + 1] = arr[i];
      i = i - 1;
}
    arr[i + 1] = key;
  }
}

int main() {
int n,x;
printf("Enter size of array: ");
scanf("%d", &n);
int arr[n];
printf("Enter Array Elements: ");
for(int i=0;i<n;i++){
   scanf("%d", &arr[i]);
}

insertionSort(arr, n);
printf("Sorted Elements: ");
for(int i=0;i<n;i++){
   printf("%d ", arr[i]);
}
  return 0;
}
```

Lab Task 6, 7 & 8: Linked List and Doubly Linked

Objective:

- To be able to implement basic operations on a singly linked list and doubly linked list

Theory:

A linked list is a linear collection of specially designed data elements, called nodes, linked to one another by means of pointers. Each node is divided into two parts: the first part contains the information of the element, and the second part contains the address of the next node in the linked list. Address part of the node is also called linked or next filed. The following figure shows a typical example of node.

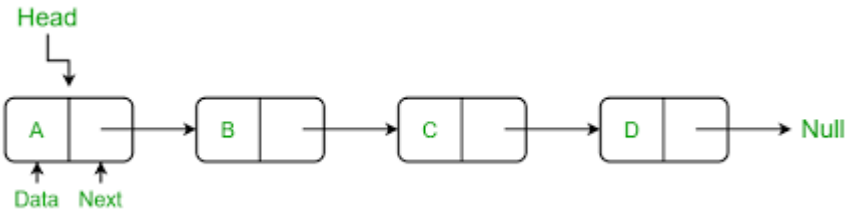


Fig. 6. Nodes in a Linked

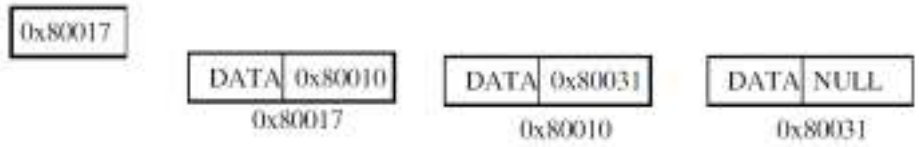


Fig. 7. Linked List

List

Fig. 7. Linked List Node Contains Data and Next Node Address

Representation of linked list

Suppose we want to store a list of integer numbers using linked list. Then it can be schematically represented as

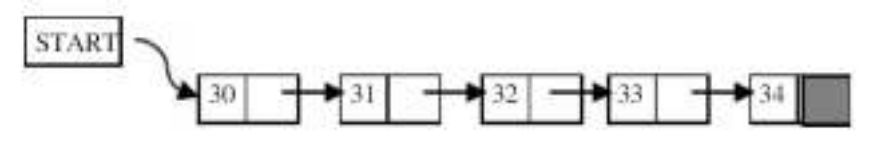


Fig. 8. Linked list representation of integers

The linear linked list can be represented in memory with the following declaration.

```
struct Node{
    int DATA;
    struct Node *Next;
};
```

Operation on linked list

The primitive operations performed on the linked list are as follows:

- Insertion
- Deletion
- Traversing
- Searching

Creation Operation is used to create a linked list. Once a linked list is created with one node, insertion operation can be used to add more elements in a node.

Insertion operation is used to insert a new node at any specified location in the linked list. A new node may be inserted.

(a) At the beginning of the linked list
(b) At the end of the linked list
(c) At any specified position in between in a linked list

Deletion operation is used to delete an item (or node) from the linked list. A node may be deleted from the

(a) Beginning of a linked list
(b) End of a linked list
(c) Specified location of the linked list

C code for inserting a node at the beginning, at the end, and at any position to the linked list:

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

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

struct node* head=NULL;
void insertAtBegining(int data){
struct node* newnode=(struct node*)malloc(sizeof(struct node*));
```

```
newnode->data = data;
newnode->next = head;
head = newnode;
}

void insertAnyPosition(int data,int pos){
struct node *tmp,*q;
int i;
q=head;
for(i=0;i<=pos-1;i++){
   q=q->next;
   if(q==NULL){
      return;

  }
}
tmp=malloc(sizeof (struct node));
tmp->next=q->next;
tmp->data=data;
q->next=tmp;
}

void traverseLinkedList(){
struct node* temp;
temp = head;
while(temp != NULL){
    printf("%d ",temp->data);
    temp=temp->next;
 }
}

void insertAtEnd(int num){
   struct node* newNode = malloc(sizeof(struct node));
   newNode->data  = num;
   newNode->next = NULL;
   if (head == NULL) {
       head = newNode;
   }
   else{
       struct node *tmp=head;
      while(tmp->next!=NULL){
         tmp=tmp->next;
      }
     tmp->next=newNode;
  }
}
```

```
int searchAvalue(int s){
struct node *tmp=head;
while(tmp!=NULL){
if(tmp->data==s){
   return 1;
 }
  tmp=tmp->next;
}
return -1;

}

int main()

{
insertAtBegining(2);
insertAtBegining(7);
insertAtBegining(8);
 insertAtEnd(5);
 insertAnyPosition(10,3);
 traverseLinkedList();

 int val = searchAvalue(7);
 if(val==-1){
     printf("\nValue is not present in the list");
      }
else{
    printf("\nValue is present in the list");
}
return 0;

}
C code for deleting a node at given position
void deleteNodeAtGivenPosition(int position){
    struct node *temp, *prev, *curr;
    if(position == 0){
        temp=head;
        head = head->next;
        free(temp);
    }
    else{
        prev = head;
        int count = 0;
        while(count<position-1){
            prev = prev->next;
            count++;
```

```
        }
        curr=prev->next;
        prev->next = curr->next;
        free(curr);

    }

}
```

C code for deleting a node at the beginning

```
void deleteAtBeginning(){
if(head==NULL){
   return;
} else{
struct node *tmp=head;
head=head->next;
free(tmp);
  }
}
```

C code for delete a node at the end

```
void deleteNodeAtEnd()
{
      struct node *curr, *tmp, *prev;
      curr=head;
      while(curr->next!=NULL)
      {
        prev=curr;
        curr=curr->next;
      }
      tmp=prev;
      prev->next=NULL;
      free(curr);
}
```

Lab Task 9: Queue

Objectives

- To be able to implement basic operations on normal queue
- To be able to implement basic operations on circular queue
- To be able to implement queue as an array
- To be able to implement a queue using Linked Lists

Theory:

A queue is logically a first in first out (FIFO or first come first serve) linear data structure. The concept of queue can be understood by our real life problems. For example a customer come and join in a queue to take the train ticket at the end (rear) and the ticket is issued from the front end of queue. That is, the customer who arrived first will receive the ticket first. It means the customers are serviced in the order in which they arrive at the service centre.

It is a homogeneous collection of elements in which new elements are added at one end called rear, and the existing elements are deleted from other end called front.

The basic operations that can be performed on queue are:

- Insert (or add) an element to the queue (enqueue)
- Delete (or remove) an element from a queue (dequeue)

Enqueue operation will insert (or add) an element to queue, at the rear end, by incrementing the array index. Dequeue operation will delete (or remove) from the front end by decrementing the array index and will assign the deleted value to a variable. Total number of elements present in the queue is front-rear+1, when implemented using arrays. Following figure will illustrate the basic operations on queue.

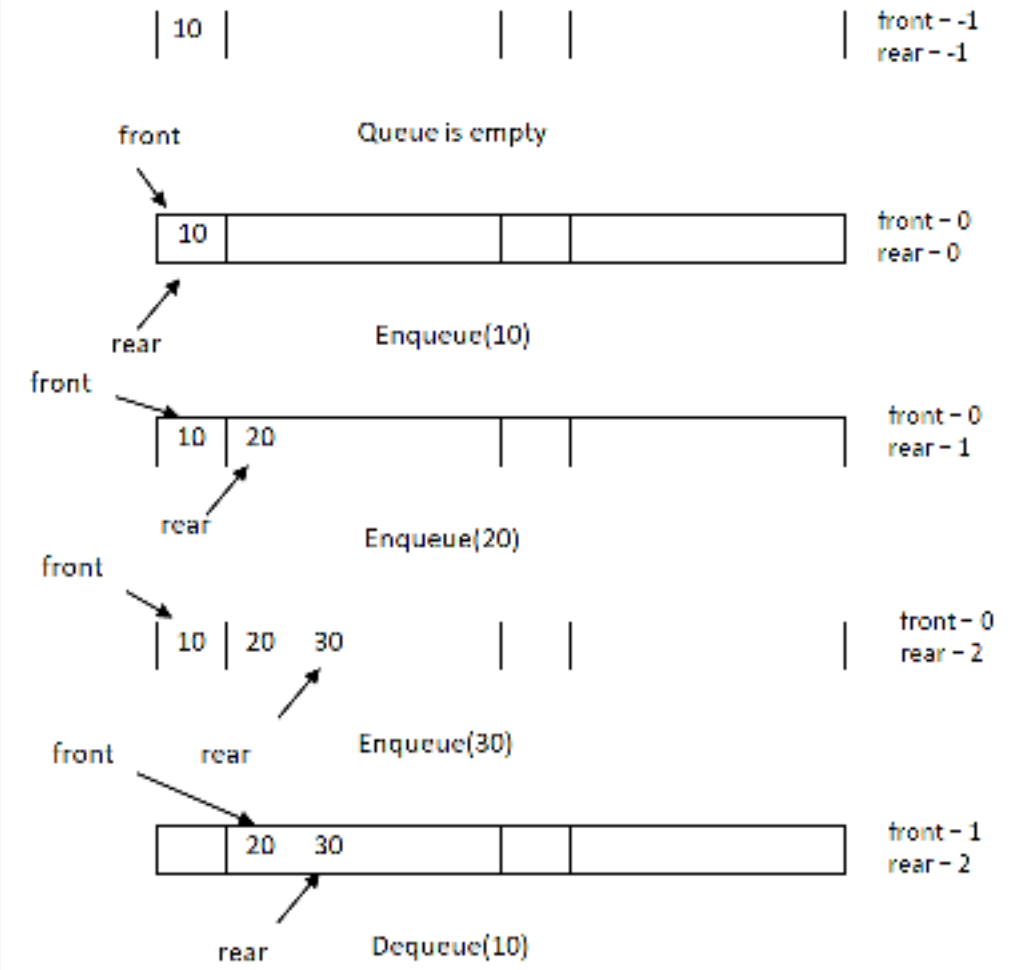


Fig. 9. Basic Operations on Queue

Pseudocode for enqueuing an element into the queue

1. Initialize front=-1 rear = -1
2. Input the value to be inserted and assign to variable “data”
3. If (rear = SIZE-1)
   (a) Display “Queue overflow”
   (b) Exit
4. Else
   (a) rear = rear +1
5. Q[rear] = data
6. Exit

Pseudocode for dequeuing an element from queue

1. If (rear== front)
   (a) Display “The queue is empty”
   (b) Exit
2. Else
   (a) Data = Q[front]

3. front = front + 1
4. Exit

C code for implementing Queue using arrays

```
#include <stdio.h>
#define SIZE 5
int front =-1;
int rear = -1;
int queue[SIZE];

void enqueue(int n){
   if(rear == SIZE - 1){
      printf("Sorry, the queue is full at the moment!");
   }
   else{
      rear++;
      queue[rear] = n;
   }
}

void dequeue(){
   int value;
   if(front == rear){
      printf("The queue is empty at now!");
   }
   else{
      front++;
      value=queue[front];
      printf("Dequeued value is: %d\n", value);
   }
}

void printQueue(){
   printf("Elements in the queue: ");
   for(int i=front+1;i<=rear;i++){
      printf("%d ", queue[i]);
   }
}

int main()
{
   int ch;
   printf("1: Enqueueing an element to queue\n");
   printf("2: Dequeueing an element from queue\n");
   printf("3: Display all the elements of queue\n");
   printf("4: Exit\n");
   do {
```

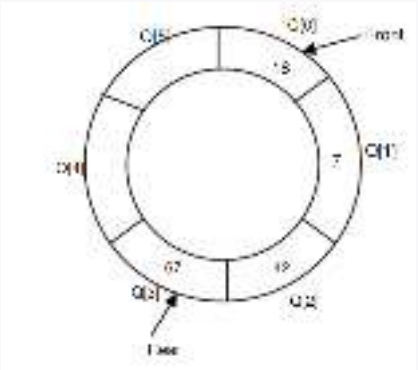
```
      printf("Enter your choice : ");
       scanf("%d", &ch);
       switch (ch) {
          case 1: {
             printf("Enter element to be enqueued:");
             int item;
             scanf("%d", &item);
             enqueue(item);
             break;
          }
          case 2: dequeue();
             break;
          case 3: printQueue();
             break;
          case 4: printf("Exit");
             break;
          default: printf("Invalid choice");
       }
   } while(ch!=4);
   return 0;
}
```

There are three major variations in a simple queue. They are:

1. Circular queue
2. Double ended queue (de-queue)
3. Priority queue

Circular Queue

In circular queues, the elements Q[0],Q[1],Q[2] .... Q[n – 1] is represented in a circular fashion with Q[1] following Q[n]. A circular queue is one in which the insertion of a new element is done at the very first location of the queue if the last location at the queue is full. Suppose Q is a queue array of 6 elements. Push and pop operation can be performed on circular. The following figures will illustrate the same.



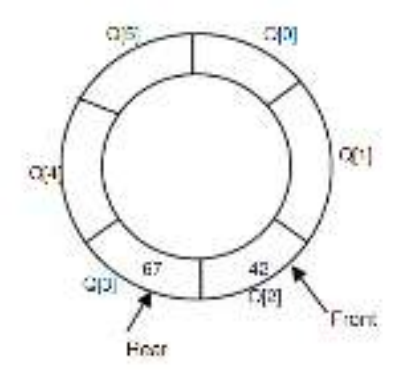


Fig. 10. A circular queue after enqueuing 18, 7, 42, 67.

Fig. 11. A circular queue after dequeuing 18, 7.

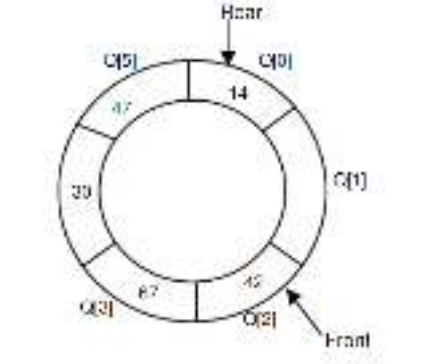


Fig. 12. A circular queue after enqueueing 30, 47, 14.

Fig. 10. A circular queue after enqueuing 18, 7, 42, 67.

Fig. 11. A circular queue after dequeuing 18, 7.

Fig. 12. A circular queue after enqueueing 30, 47, 14.

At any time the position of the element to be enqueued will be calculated by the relation Rear = (Rear + 1) % SIZE. After dequeuing an element from circular queue the position of the front end is calculated by the relation Front= (Front + 1) % SIZE. After locating the position of the new element to be inserted,
rear, compare it with front. If (rear = front), the queue is full and cannot be inserted anymore.

Pseudocode for enqueuing an element to the circular queue

1. Initialize FRONT = – 1; REAR = -1
2. If (REAR + 1) % SIZE==FRONT)
(a) Display “Queue is full”
(b) Exit
3. Else
   (a) Input the value to be inserted and assign to variable “DATA”
4. If (FRONT is equal to – 1)
   (a) FRONT = 0
5. REAR = (REAR + 1) % SIZE
6. Q[REAR] = DATA
7. Repeat steps 2 to 4 if we want to insert more elements
8. Exit

Deleting an element from a circular queue

1. If (FRONT == – 1)
   (a) Display “Queue is empty”
   (b) Exit
2. Else
   (a) DATA = Q[FRONT]
3. If (REAR is equal to FRONT)
   (a) FRONT = –1
   (b) REAR = –1
4. Else
   (a) FRONT = (FRONT +1) % SIZE
5. Repeat the steps 1, 2 and 3 if we want to delete more elements
6. Exit

C code for implementing a Circular queue

```
#include <stdio.h>
#define SIZE 5
int front =-1;
int rear = -1;
int queue[SIZE];

void enqueue(int number){
      if((rear+1)%SIZE==front){
         printf("Queue is full\n");
      }
      else{
         if(front==-1){
            front=0;
         }
      rear=(rear+1)%SIZE;
      queue[rear]=number;
    }
```

```
   }


void dequeue(){
        int element;
        if(front==-1){
           printf("Queue is empty\n");
        }
        else {
             element = queue[front];
             if (front == rear) {
                front = -1;
                rear = -1;
              }
             else{
                 front=(front+1)%SIZE;

             }
             printf("\n Deleted element -> %d \n", element);
        }
}

void printQueue(){
        int i;
        if(front<=rear){
        for(i=front;i<=rear;i++){
           printf("%d ", queue[i]);
        }
        }
        else{
           i=front;
           while(i<SIZE){
              printf("%d ", queue[i]);
              i++;
           }
           i=0;
           while(i<=rear){
              printf("%d ", queue[i]);
              i++;
           }

        }

   }
```

```
int main()
{
     int ch;
   printf("1: Enqueueing an element to queue\n");
   printf("2: Dequeueing an element from queue\n");
   printf("3: Display all the elements of queue\n");
   printf("4: Exit\n");
   do {
      printf("Enter your choice : ");
       scanf("%d", &ch);
       switch (ch) {
          case 1: {
             printf("Enter element to be enqueued:");
             int item;
             scanf("%d", &item);
             enqueue(item);
             break;
          }
          case 2: dequeue();
             break;
          case 3: printQueue();
             break;
          case 4: printf("Exit");
             break;
          default: printf("Invalid choice");
       }
   } while(ch!=4);


   return 0;
}
```

Week 10: Stack

Objectives:

- To be able to implement stack as an array
- To be able to implement stack as Linked Lists
- To be able to evaluate postfix expressions using a stack

Theory:

A stack is one of the most important and useful linear data structure in computer science. It is an ordered collection of items into which new data items may be added/inserted and from which items may be deleted at only one end, called the top of the stack. As all the addition and deletion in a stack is done from the top of the stack, the last added element will be first removed from the stack. That is why the stack is also called Last-in-First-out (LIFO).

Note that the most frequently accessible element in the stack is the top most elements, whereas the least accessible element is the bottom of the stack. The operation of the stack can be illustrated as in Fig. 13.

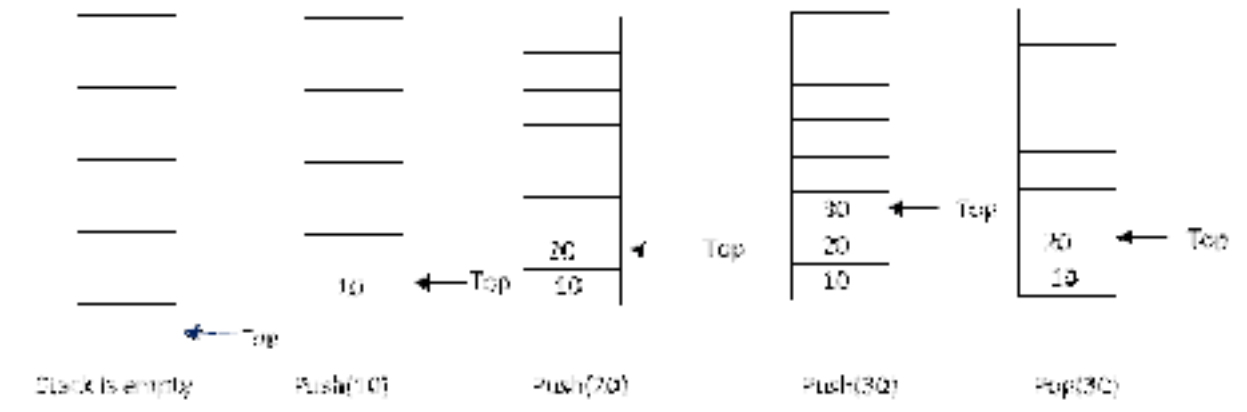


Fig. 13. Stack Operations

The insertion (or addition) operation is referred to as push and the deletion (or remove) operation as pop. A stack is said to be Empty or underflow, if the stack contains no elements. At this point the top of the stack is present at the bottom of the stack. And it is overflow when the stack becomes full, i.e., no other elements can be pushed onto the stack. At this point the top pointer is at the highest location of the stack.

OPERATIONS PERFORMED ON STACK

The primitive operations performed on the stack are as follows:

PUSH:

The process of adding (or inserting) a new element to the top of the stack iscalled PUSH operation. Pushing an element to a stack will add the new element at the top. After every push operation the top is incremented by one. If the array is full and no new element can be accommodated, then the stack overflow condition occurs.

POP:

The process of deleting (or removing) an element from the top of stack is called POP operation. After every pop operation the stack is decremented by one. If there is no element in the stack and the pop operation is performed then the stack underflow condition occurs.

STACK I MPLEMENTATION

Stack can be implemented in two ways:

1. Static implementation (using arrays)
2. Dynamic implementation (using linked lists)

Pseudocode for PUSH operation in STACK

1.If TOP = SIZE – 1, then:
  (a) Display “The stack is in overflow condition”
  (b) Exit
2.TOP = TOP + 1
3. STACK [TOP] = ITEM
4. Exit

Pseudocode for POP operation in STACK

1. If TOP < 0, then
  (a) Display “The Stack is empty”
  (b) Exit
2. Else remove the Top most element
3. DATA = STACK[TOP]
4. TOP = TOP – 1
5. Exit

C code for implementing Stack using arrays

```
#include <stdio.h>
int top =-1;
#define MAX 10
int arr[MAX];

void push(int element){
   if(top == MAX-1){
      printf("Stack is full");
   }
   else{
      top++;
      arr[top] = element;
      //printf("%d Added value in the stack:  \n", element);
   }
}
void pop(){
   int element;
   if(top==-1){
      printf("Stack is empty: ");
   }
```

```
   else{
      element = arr[top];
      top--;
      printf("Popped value: %d", element);
   }
}
void display(){
   printf("Elements in the stack:");
   for(int i=top;i>=0; i--){

      printf("%d ", arr[i]);
   }
}

int main()
{
   push(10);
   push(20);
   push(30);
   push(40);
   display();
   printf("\n");
   pop();
   printf("\n");
   pop();
   printf("\n");
   display();
   return 0;
}
```

C code for implementing Stack using Linked List

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
struct node *head = NULL;

void push(int val)
{
   struct node *newNode = malloc(sizeof(struct node));
   newNode->data = val;
   newNode->next = head;
   head = newNode;
}
```

```
void pop()
{
   struct node *temp;
   if(head == NULL)
      printf("Stack is Empty\n");
   else
   {
      printf("Poped element = %d\n", head->data);
      temp = head;
      head = head->next;
      free(temp);
   }
}

void display()
{
   struct node *temp = head;
   while(temp != NULL)
   {
       printf("%d ", temp->data);
       temp = temp->next;
   }

}

int main()
{
   push(100);
   push(200);
   push(300);
   printf("Elements in the Linked List: ");
   display();
   printf("\n");
   pop();
   printf("Showing the new linked list after the pop:");
   display();
   printf("\n");
   pop();
   printf("Showing the new linked list after the pop: ");
   display();
   return 0;
}
```

Lab Task 11 & 12: Binary Search Tree

Objectives:

- To be able to create and implement Binary Search Tree
- To understand the operations of the Binary Search Tree
- To be able to write application programs using the Binary Search Tree

Theory:

A Binary Search Tree is a binary tree, which is either empty or satisfies the following properties:

- Every node has a value and no two nodes have the same value (i.e., all the values are unique)
- If there exists a left child or left sub tree then its value is less than the value of the root
- The value(s) in the right child or right sub tree is larger than the value of the root node

All the nodes or sub trees of the left and right children follows above rules. The Fig. 14 shows a typical binary search tree. Here the root node information is 50. Note that the right sub tree node's value is greater than 50, and the left sub tree nodes value is less than 50. Again right child node of 25 has large values than 25 and left child node has small values than 25. Similarly right child node of 75 has large values than 75 and left child node has small values that 75 and so on.

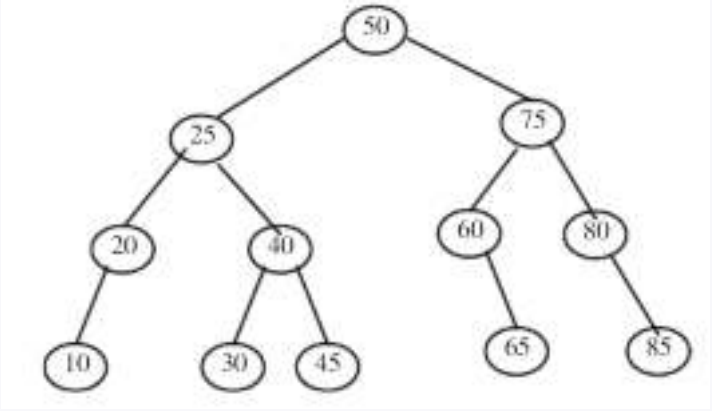


Fig. 14. A Binary Search Tree

The operations performed on binary tree can also be applied to Binary Search Tree (BST). The following operations are performed on BST :

- Inserting a node
- Searching a node
- Deleting a node

Another most commonly performed operation on BST is, traversal. The tree traversal algorithm (pre-order, post-order and in-order) are the standard way of traversing a binary search tree.

C code for inserting a node, delete a node and search for a node in the Binary Search Tree

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

```
#include<stdbool.h>

struct node{
   int data;
   struct node *left;
   struct node *right;
};

struct node *createNode(int n){
   struct node *newNode = malloc(sizeof(struct node));
   newNode->data=n;
   newNode->left = NULL;
   newNode->right = NULL;

   return newNode;
}

struct node *insert(struct node *root, int value){
   if(root == NULL){
      return createNode(value);
   }
   else if(root->data<value){
      root->right=insert(root->right, value);
   }
   else{
      root->left = insert(root->left, value);
   }

   return root;
}
void inOrder(struct node *root){
   if(root==NULL){
      return;
   }
   inOrder(root->left);
   printf("%d ", root->data);

   inOrder(root->right);
}

bool search(struct node *root, int searchingValue){
      bool found = false;
      if(root==NULL){
       return false;
      }
      else if(root->data==searchingValue){
```

```
        return true;
       }
       else if(root->data<searchingValue){
        found = search(root->right, searchingValue);
       }
       else{
        found = search(root->left, searchingValue);
       }

       return found;
}

struct node *minValueNode(struct node *root) {
  struct node *tmp = root;
  while (tmp->left != NULL)
    tmp = tmp->left;

  return tmp;
}

struct node *deleteNode(struct node *root, int key) {
   if (root == NULL)
      return root;
   if (key < root->data)
      root->left = deleteNode(root->left, key);
   else if (key > root->data)
      root->right = deleteNode(root->right, key);

   else {
        if (root->left == NULL) {
        struct node *temp = root->right;
        free(root);
        return temp;
    } else if (root->right == NULL) {
       struct node *temp = root->left;
       free(root);
       return temp;
    }
    struct node *temp = minValueNode(root->right);
    root->data = temp->data;
    root->right = deleteNode(root->right, temp->data);
  }
  return root;
}
```

```
int main()
{
  struct node *root = NULL;
  root = insert(root, 50);
  root = insert(root, 30);
  root = insert(root, 20);
  root = insert(root, 60);
  root = insert(root, 55);
  root = insert(root, 65);
  printf("Inorder traversal: ");
  inOrder(root);
  printf("\nAfter deleting:60 \n");
  root = deleteNode(root, 60);
  printf("Inorder traversal: ");
  inOrder(root);
  int node = search(root, 30);
  if(node == 0){
    printf("\nValue is not found in BST\n");

  } else{
       printf("\nValue is found in BST\n");
    }

    return 0;
}
```

References:

1. Data Structures Using C, Second Edition by Reema Thareja
2. Data Structure and Algorithmic Thinking with Python: Data Structure and Algorithmic Puzzles, 1st Edition by Narasimha Karumanchi
3. Principles of Data Structures Using C and C++ by Das, Vinu V

THANK YOU