GH raisoni COLLEGE
Engineering and Management Pune

G H Raisoni College of Engineering & Management
An Empowered Autonomous Institute, Affiliated to Savitribai Phule Pune University
Approved by AICTE, New Delhi and Recognized by Govt. of Maharashtra, NAAC Accredited "A+" Grade
Navin Gat No.1200, Domkhel Road, Wagholi, Pune – 412 207 (India)
T: +91 - 9604787185/186 | E: ghrcem.pune@raisoni.net | W: www.ghrcemp.raisoni.net

`

Department of Computer Engineering

D-19

Lab Manual (2024-25)

Pattern-2023

Class: SY Computer Term: III

Data Structure and Algorithms
23UCOP2410

`

G H Raisoni College of Engineering and Management, Wagholi, Pune 412207

Department: Computer Engineering

Course Details

Subject: Data Structure and Algorithms (23UCOP2410)

CLASS: SY BTECH **DIVISION: A**

External Marks: 25

COURSE OUTCOME	
CO1	Describe the fundamentals of data structure and perform the basic operations on data structure.
CO2	Interpret and compare various searching, sorting algorithms for efficient access of data.
CO3	Demonstrate the role of data structure in structuring and manipulating data and implement using array or linked list representation
CO4	Implement linear and non-linear data structures to solve the given problems.

`

List of Experiments

Sr. No	List of Laboratory Assignments	Relevanc eto CO	Software Required
1.	• Implement Linear search and Binary Search for a given array.	CO2	Dev C++, Code Block
2	• Arrange the list of students according to roll numbers in ascending order using a) Bubble Sort b) Insertion sort c) Quick sort	CO2	Dev C++, Code Block
3	• Implement a sparse matrix with operations like initialize empty sparse matrix, insert an element, sort a sparse matrix on row-column, transpose a matrix, etc.	CO1	Dev C++, Code Block
4.	• Write functions to a) Add and delete the nodes in a linked list b) Compute total number of nodes in the linked list c) Display list in reverse order using recursion	CO3	Dev C++, Code Block
5	• Implement a data type to represent a Polynomial with the operations like create an empty polynomial, insert an entry into polynomial, add two polynomials and return the result as a polynomial, evaluate a polynomial, etc.	CO3	Dev C++, Code Block
6	• Implement Stack using a linked list. Use this stack to perform evaluation of a postfix expression.	CO1	Dev C++, Code Block
7	• Write a function which evaluates an infix expression, without converting it to postfix. The input string can have spaces, (,) and precedence of operators should be handled.	CO4	Dev C++, Code Block
8	• Implement Tower of Hanoi using Recursion.	CO4	Dev C++, Code Block
9	• Write a program to simulate deque with functions to add and delete elements from either end of the deque.	CO3	Dev C++, Code Block
10	• Beginning with an empty binary search tree, Construct binary search tree by inserting the values in the order given. After constructing a binary tree CO4 i. Insert new node ii. Find number of nodes in longest path iii. Minimum data value found in the tree	CO4	Dev C++, Code Block

`

	iv. Change a tree so that the roles of the left and right pointers are swapped at every node v. Search a value		
Content Beyond Syllabus			
11	Develop C program to build a MAX_HEAP with given input.	CO4	Dev C++, Code Block
12	Develop hash table to implement hashing.	CO1	Dev C++, Code Block

`

Experiment No. 1

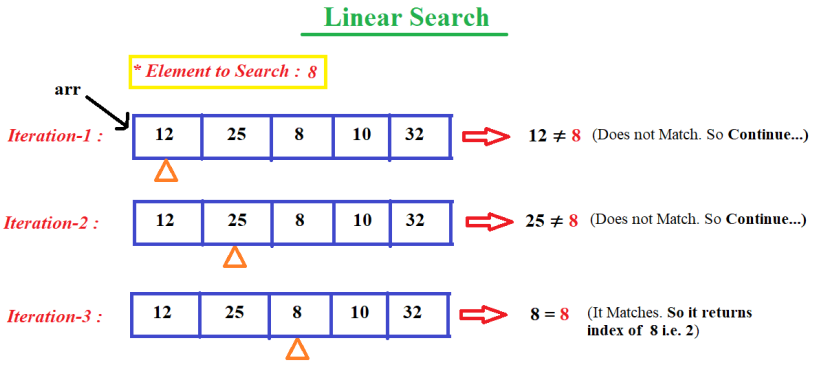
Aim:
Implement Linear Search and Binary Search for a given array.

Theory:
A linear search is also known as a sequential search that simply scans each element at a time. Assume that item is in an array in random order and we have to find an item. Then the only way to search for a target item is, to begin with, the first position and compares it to the target. If the item is at the same, we will return the position of the current item. Otherwise, we will move to the next position. If we arrive at the last position of an array and still cannot find the target, we return -1.
Following are the sequence of steps for linear search.

1. Start at the beginning of the array.
2. Compare the current element with the target value.
3. If the current element matches the target value, return the index of the current element.
4. If the current element does not match the target value, move to the next element in the array.
5. Repeat steps 2-4 until the end of the array is reached or the target value is found

Algorithm: (Linear Search)
LINEAR (A, SKEY) Here A is a Linear Array with N elements and KEY is a given item of information to search. This algorithm finds the location of KEY in A and if successful, it returns its location otherwise it returns -1 for unsuccessful.
1. Repeat for i = 0 to N-1
2. if(A[i] = KEY) return i [Successful Search]
[End of loop]
3. return -1 [Un-Successful]
4. Exit.

Example



`

C Code:

```
#include<stdio.h>
#include<conio.h>
void main ()
{
int a[10] = {10, 23, 40, 1, 2, 0, 14, 13, 50, 9};
int item, i, flag;
printf("\nEnter Item which is to be searched\n");
scanf("%d",&item);
for (i = 0; i< 10; i++)
{
if(a[i] == item)
{
flag = i+1;
break;
}
else
{
flag = 0;
}
}
if(flag != 0)
{
printf("\nItem found at location %d\n",flag);
}
else
{
printf("\nItem not found\n");
}
getch();
}
```

Output:

```
Enter Item which is to be searched
6

Item not found
Enter Item which is to be searched
9

Item found at location 10
```

`

Binary Search

A binary search is a search in which the middle element is calculated to check whether it is smaller or larger than the element which is to be searched. The main advantage of using binary search is that it does not scan each element in the list. Instead of scanning each element, it performs the searching to the half of the list. So, the binary search takes less time to search an element as compared to a linear search.

The one pre-requisite of binary search is that an array should be in sorted order, whereas the linear search works on both sorted and unsorted array. The binary search algorithm is based on the divide and conquers technique, which means that it will divide the array recursively.

There are three cases used in the binary search:

Case 1: data<a[mid] then left = mid+1.
Case 2: data>a[mid] then right=mid-1
Case 3: data = a[mid] // element is found

In the above case, 'a' is the name of the array, mid is the index of the element calculated recursively, data is the element that is to be searched, left denotes the left element of the array and right denotes the element that occur on the right side of the array.

The algorithm for binary search can be described in the following steps:

1. Set the lower bound low to the first index of the array and the upper bound high to the last index.
2. While low is less than or equal to high:
    I. Calculate the middle index mid as the average of low and high.
    II. Compare the element at index mid with the target value.
    III. If the element at index mid matches the target value, return the index mid.
    IV. If the element at index mid is less than the target value, set low to mid + 1.
    V. If the element at index mid is greater than the target value, set high to mid - 1.
3. If the target value is not found, return -1.

Algorithm: (Binary Search)

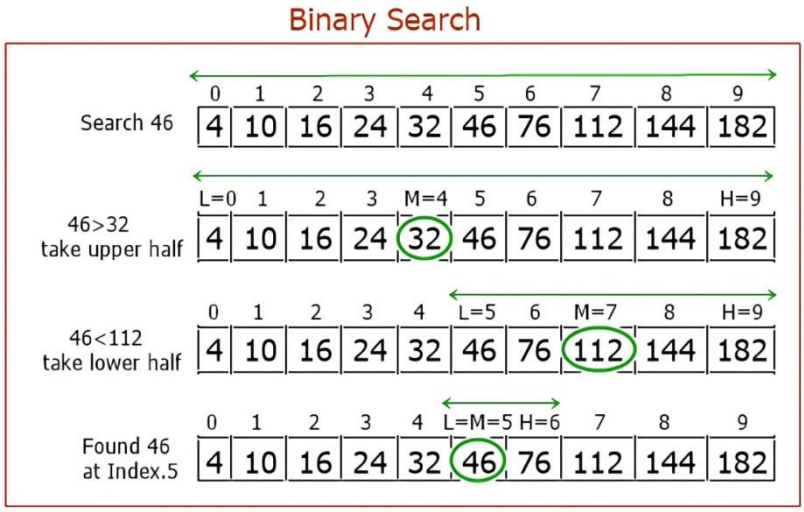
Here A is a sorted Linear Array with N elements and SKEY is a given item of information to search. This algorithm finds the location of SKEY in A and if successful, it returns its location otherwise it returns -1 for unsuccessful.

BinarySearch (A, SKEY)
1. [Initialize segment variables.] Set START=0, END=N-1 and
   MID=INT((START+END)/2).
2. Repeat Steps 3 and 4 while START ≤ END and A[MID]≠SKEY.
3. If SKEY< A[MID]
  Then Set END=MID-1.
    Else Set START=MID+1.
  [End of If Structure]
4. Set MID=INT((START +END)/2).
   [End of Step 2 loop.]
5. If A[MID]= SKEY

`

then Set LOC= MID
Else: Set LOC = -1 [End of IF structure.]
6. return LOC and Exit

Example



C Code

```
#include<stdio.h>
#include<conio.h>
void main()
{
int first, last, middle, size, i, key, list[100];
printf("Enter the size of the list: ");
scanf("%d",& size);
printf("Enter %d integer values in Ascending order\n", size);
for (i = 0; i < size; i++)
{
scanf("%d",&list[i]);
}
printf("Enter value to be search: ");
scanf("%d", &key);
first = 0;
last = size - 1;
middle = (first+last)/2;
while (first <= last)
{
if (list[middle] <key)
{
first = middle + 1;
}
else if (list[middle] == key)
```

`

```
{
printf("Element found at index %d.\n",middle);
break;
}
else
{
last = middle - 1;}
middle = (first + last)/2;
}
if (first > last)
{
printf("Element Not found in the list.");
}
}
```

Output

```
Enter the size of the list: 6
Enter 6 integer values in Ascending order
2 5 8 13 19 23
Enter value to be search: 19
Element found at index 4.
```

Conclusions: This program implements linear search and binary search. The time complexity of binary search is O(log n) in worst case and for linear search it is O(n).

`

Experiment No. 2

Aim: Arrange the list of students according to roll numbers in ascending order using
a) Bubble Sort
b) Insertion sort
c) Quick sort

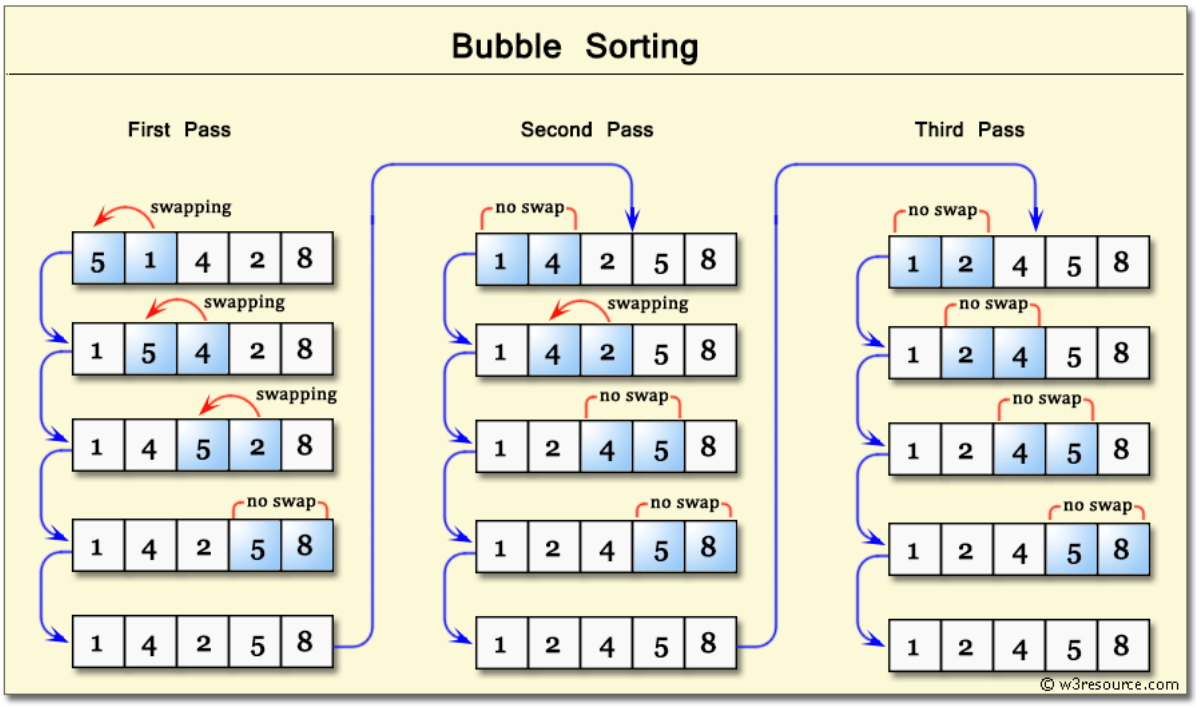
Theory:
Bubble sort

The technique we use is called "Bubble Sort" because the bigger value gradually bubbles their way up to the top of array like air bubble rising in water, while the small values sink to the bottom of array. This technique is to make several passes through the array. On each pass, successive pairs of elements are compared. If a pair is in increasing order (or the values are identical), we leave the values as they are. If a pair is in decreasing order, their values are swapped in the array.

Algorithm: (Bubble Sort)
BUBBLE (DATA, N) Here DATA is an Array with N elements. This algorithm sorts the elements in DATA.
1. for pass=1 to N-1.
2. for (i=0; i<= N-Pass; i++)
3. If DATA[i]>DATA [i+1],
    then: Interchange DATA[i] and DATA[i+1].
    [End of If Structure]
  [End of inner loop]
[End of Step 1 outer loop]
4. Exit.

`

Example



C Code

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
 int i, n, temp, j, arr[10];
 printf("Enter the maximum elements you want to store : ");
 scanf("%d", &n);
 printf("Enter the elements \n");
 for(i=0;i<n;i++)
 {
 scanf("%d", & arr[i]);
 }
for(i=0;i<n;i++)
 {
 for(j=0;j<n-1;j++)
 {
 if(arr[j]>arr[j+1])
 {
 temp = arr[j];
 arr[j] = arr[j+1];
 arr[j+1] = temp;
 }
```

```
 }
 }
 printf("The array sorted in ascending order is :\n");
 for(i=0;i<n;i++)
 printf("%d\t", arr[i]);
 getch();
 return 0;
}
```

Output

```
Enter the maximum elements you want to store : 8
Enter the elements
3
8
1
6
90
23
12
5
The array sorted in ascending order is :
1       3       5       6       8       12      23      90
```

Insertion sort

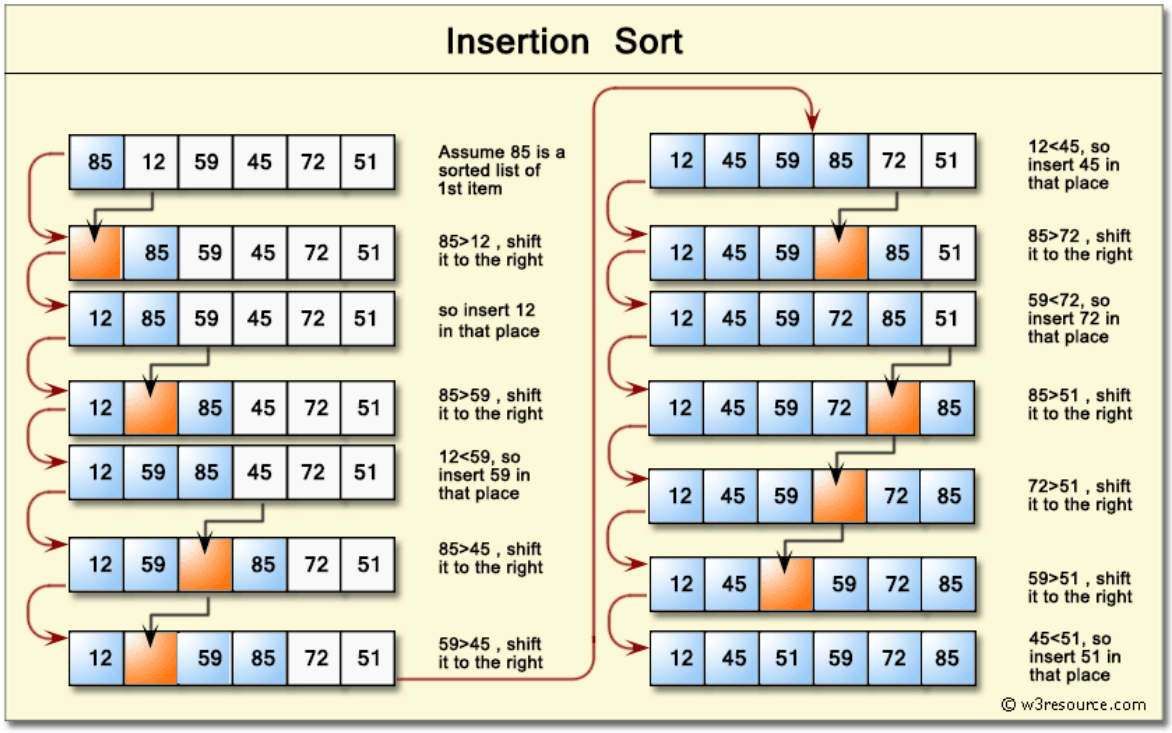
Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration. The array is virtually split into a sorted and an unsorted part. Elements from the unsorted part are picked and placed at the correct position in the sorted part. For example, the lower part of an array is maintained to be sorted. An element which is to be inserted in this sorted list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.

Algorithm: INSERTION-SORT (ARR, N)

Step 1: Repeat Steps 2 to 5 for K = 1 to N-1
Step 2: SET TEMP = ARR[K]
Step 3: SET J = K - 1
Step 4: Repeat while TEMP <=ARR[J] SET ARR[J + 1] = ARR[J] SET J = J - 1
        [END OF INNER LOOP]
Step 5: SET ARR[J + 1] = TEMP
        [END OF OUTER LOOP]
Step 6: EXIT

Example

`



C Code

```
#include<stdio.h>
#include<conio.h>
void main ()
{
 int i, j, k,temp;
 int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
 printf("\nprinting sorted elements...\n");
 for(k=1; k<10; k++)
 {
 temp = a[k];
 j= k-1;
 while(j>=0 && temp <= a[j])
 {
 a[j+1] = a[j];
 j = j-1;
 }
 a[j+1] = temp;
 }
 for(i=0;i<10;i++)
 {
 printf("\n%d\n",a[i]);
 }
```

`

```
 getch();
}
```

Output

Sorted elements...

7

9

10

12

23

23

34

44

78

101

QuickSort

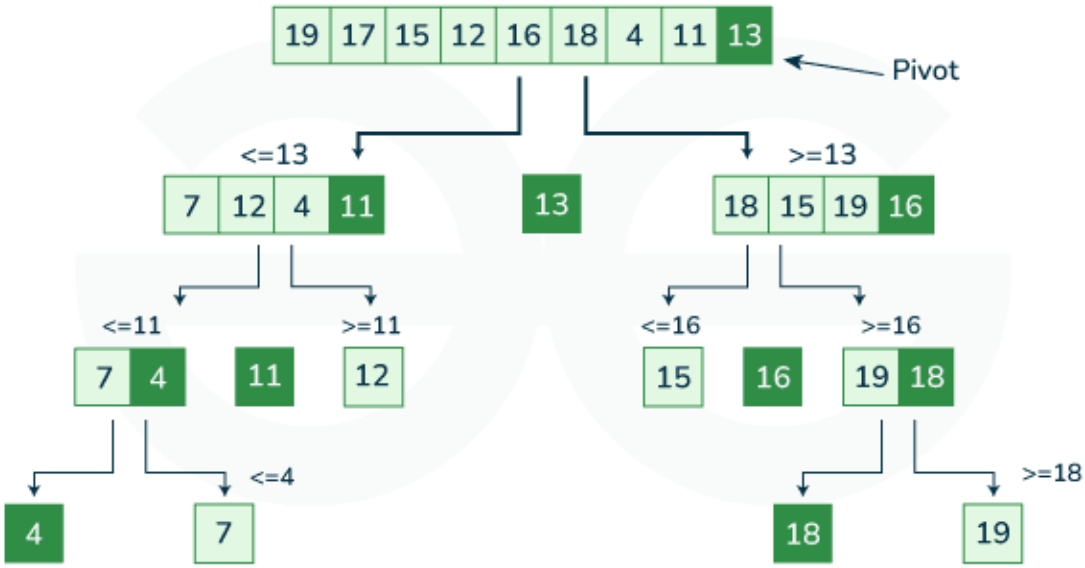
Quicksort is one of the best sorting algorithms developed by Tony Hoare in 1960 to sort the array. It follows the divide-and-conquer. The basic idea behind QuickSort is to select a "pivot" element from the array and partition the other elements into two sub-arrays according to whether they are less than or greater than the pivot.

.

Working: Quicksort algorithm is based on divide and conquers approach, where an array is divided into subarrays by selecting a pivot element (element selected from the array).

1. While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.
2. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
3. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

Example



// Quick sort implementation in C

```
#include <stdio.h>
// function to swap elements
void swap(int *a, int *b) {
  int t = *a;
  *a = *b;
  *b = t;
}
// function to find the partition position
int partition(int array[], int low, int high) {
  // select the rightmost element as pivot
  int pivot = array[high];
  // pointer for greater element
  int i = (low - 1);
  // traverse each element of the array
  // compare them with the pivot
  for (int j = low; j < high; j++) {
    if (array[j] <= pivot) {
      // if element smaller than pivot is found
      // swap it with the greater element pointed by i
      i++;
      // swap element at i with element at j
      swap(&array[i], &array[j]);
    }
  }
```

```
  // swap the pivot element with the greater element at i
  swap(&array[i + 1], &array[high]);
  // return the partition point
  return (i + 1);
}
void quickSort(int array[], int low, int high) {
  if (low < high) {
    // find the pivot element such that
    // elements smaller than pivot are on left of pivot
    // elements greater than pivot are on right of pivot
    int pi = partition(array, low, high);
    // recursive call on the left of pivot
    quickSort(array, low, pi - 1);
    // recursive call on the right of pivot
    quickSort(array, pi + 1, high);
  }
}

// function to print array elements
void printArray(int array[], int size) {
  for (int i = 0; i < size; ++i) {
    printf("%d  ", array[i]);
  }
  printf("\n");
}
// main function
int main() {
  int data[] = {8, 7, 2, 1, 0, 9, 6};
  int n = sizeof(data) / sizeof(data[0]);
  printf("Unsorted Array\n");
  printArray(data, n);
  // perform quicksort on data
  quickSort(data, 0, n - 1);
  printf("Sorted array in ascending order: \n");
  printArray(data, n);
}
```

Output

Unsorted Array
8 7 2 1 0 9 6
Sorted array in ascending order:
0 1 2 6 7 8 9

`

Conclusion: This program implements three sorting algorithm, Insertion sort, Bubble sort and Quick sort. The time complexity of insertion sort and bubble sort are O (n^2), in worst case. Quick sort algorithm follows divide and conquer approach, its time complexity is O (nlogn) in average case.

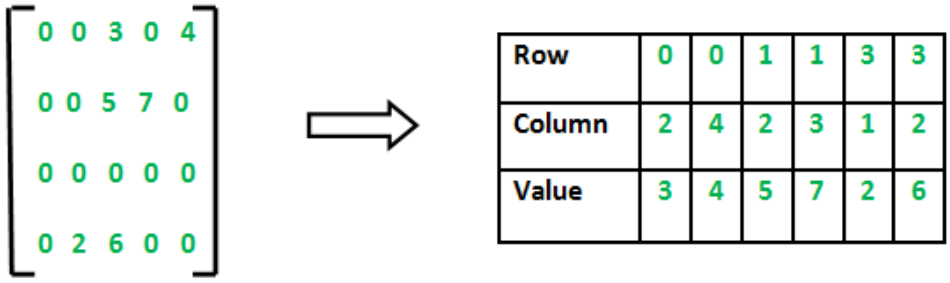
`

Experiment No. 3

Aim: Implement a sparse matrix with operations like initialize empty sparse matrix, insert an element, sort a sparse matrix on row-column, transpose a matrix, etc.

Theory:
A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have 0 values, then it is called a sparse matrix.
Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with triples- (Row, Column, value).

Sparse matrix representation



//C Code

```
#include<stdio.h>
 int main()
{
   // Assume 4x5 sparse matrix
   int sparseMatrix[4][5] =
   {
      {0 , 0 , 3 , 0 , 4 },
      {0 , 0 , 5 , 7 , 0 },
      {0 , 0 , 0 , 0 , 0 },
      {0 , 2 , 6 , 0 , 0 }
   };

   int size = 0;
   for (int i = 0; i < 4; i++)
      for (int j = 0; j < 5; j++)
         if (sparseMatrix[i][j] != 0)
            size++;
```

`

```
    // number of columns in compactMatrix (size) must be equal to number of non - zero
elements in sparseMatrix

    int compactMatrix[3][size];
     // Making of new matrix
    int k = 0;
    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 5; j++)
            if (sparseMatrix[i][j] != 0)
            {
                compactMatrix[0][k] = i;
                compactMatrix[1][k] = j;
                compactMatrix[2][k] = sparseMatrix[i][j];
                k++;
            }

    for (int i=0; i<3; i++)
    {
        for (int j=0; j<size; j++)
            printf("%d ", compactMatrix[i][j]);

        printf("\n");
    }
    return 0;
}
//Transpose of sparse matrix
#include<stdio.h>
#include<stdlib.h>
#define max 5
int main()
{
        int matrix[max][max];
        int spmatrix[max][3];
        int transposematrix[max][3];
        int i,j,k,row,col;
        printf("Enter the order of sparse matrix\n");
        scanf("%d%d",&row,&col);
        printf("Enter the element of the sparse matrix\n");
        for(i=0;i<row;i++)
        for(j=0;j<col;j++)
                scanf("%d",&matrix[i][j]);
        k=1;
```

`

```
        for(i=0;i<row;i++)
        for(j=0;j<col;j++)
        if(matrix[i][j]!=0)
        {
                spmatrix[k][0]=i;
                spmatrix[k][1]=j;
                spmatrix[k][2]=matrix[i][j];
                k++;
        }
        spmatrix[0][0]=row;
        spmatrix[0][1]=col;
        spmatrix[0][2]=k-1;

        printf("ELEMENTS OF THE SPARSE MATRIX\n");
        for(i=0;i<=spmatrix[0][2];i++)
        {
                for(j=0;j<3;j++)
                        printf("%d\t",spmatrix[i][j]);

                printf("\n");
        }

        transposematrix[0][0]=spmatrix[0][1];
        transposematrix[0][1]=spmatrix[0][0];
        transposematrix[0][2]=spmatrix[0][2];

        k=1;
        for(i=0;i<spmatrix[0][1];i++)
        for(j=0;j<=spmatrix[0][2];j++)
        if(spmatrix[j][1]==i){
                transposematrix[k][0]=spmatrix[j][1];
                transposematrix[k][1]=spmatrix[j][0];
                transposematrix[k][2]=spmatrix[j][2];
                k++;
                }

        printf("Transpose of the sparse matrix\n");
        for(i=0;i<=transposematrix[0][2];i++)
        {
        for(j=0;j<3;j++)
                printf("%d\t",transposematrix[i][j]);
        printf("\n");
        }
return 0;
}
```

`

Output

```
Row      0 0 1 1 3 3
Column 2 4 2 3 1 2
Value    3 4 5 7 2 6

Enter the order of sparse matrix
3
4
Enter the element of the sparse matrix
0
0
0
0
1
0
0
0
2
0
0
15
ELEMENTS OF THE SPARSE MATRIX
3       4       3
1       0       1
2       0       2
2       3       15
Transpose of the sparse matrix
4       3       3
0       1       1
0       2       2
3       2       15
```

Conclusion: The aim of this experiment is to implement sparse matrix with row- column representation. This representation allows us to store only non-zero elements, makes it space efficient.

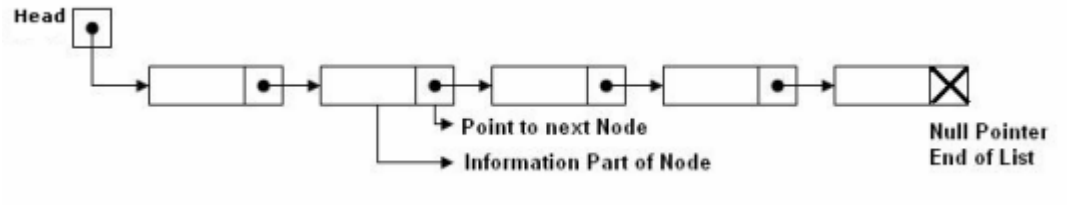
`

Experiment No. 4

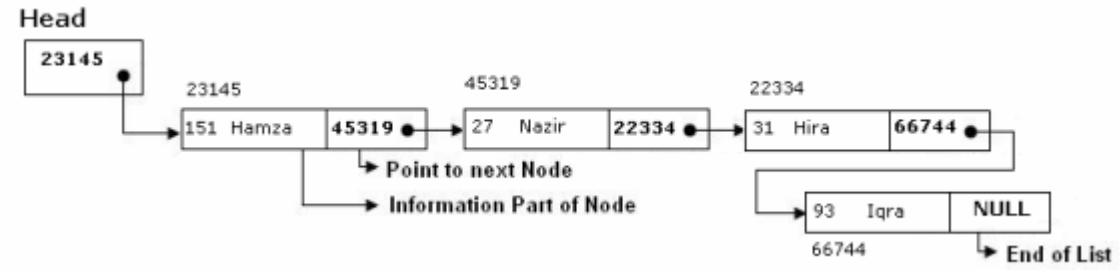
Aim:
Write functions to
a) Add and delete the nodes in a linked list
b) Compute total number of nodes in the linked list
c) Display list in reverse order using recursion

Theory:
Linked List: A linked list or one way list is a linear collection of data elements, called nodes, where the linear order is given by means of "pointers". Each node is divided into two parts.

- The first part contains the information of the element.
- The second part called the link field contains the address of the next node in the list.



Example



The **Head** is a special pointer variable which contains the address of the first node of the list. If there is no node available in the list then Head contains NULL value that means, List is empty. The left part of the each node represents the information part of the node, which may contain an entire record of data (e.g. ID, name, marks, age etc). the right part represents pointer/link to the next node. The next pointer of the last node is null pointer signal the end of the list.

Inserting a new node in list: The following algorithm inserts an ITEM into LIST.
Algorithm: INSERT(ITEM)
[This algorithm add newnodes at any position (Top, in Middle and at
End) in the List]
1. Create a NewNode node in memory
2. Set NewNode -> INFO =ITEM. [Copies new data into INFO of new node.]
3. Set NewNode -> NEXT = NULL. [Copies NULL in NEXT of new node.]
4. If HEAD=NULL, then HEAD=NewNode and return. [Add first node in list]
5. if NewNode-> INFO < HEAD->INFO
then Set NewNode->NEXT=HEAD and HEAD=NewNode and return

`

[Add node on top of existing list]
6. PrevNode = NULL, CurrNode=NULL;
7. for(CurrNode =HEAD; CurrNode != NULL; CurrNode = CurrNode ->NEXT)
{ if(NewNode->INFO <= CurrNode ->INFO)
{
break the loop
}
PrevNode = CurrNode;
} [end of loop]
[Insert after PREV node (in middle or at end) of the list]
8. Set NewNode->NEXT = PrevNode->NEXT and
9. Set PrevNode->NEXT= NewNode.
10.Exit.

Delete a node from list: The following algorithm deletes a node from any position in the LIST.
Algorithm: DELETE(ITEM)
LIST is a linked list in the memory. This algorithm deletes the node
where ITEM first appear in LIST, otherwise it writes "NOT FOUND"
1. if Head =NULL then write: "Empty List" and return [Check for Empty List]
2. if ITEM = Head -> info then: [Top node is to delete]
Set Head = Head -> next and return
3. Set PrevNode = NULL, CurrNode=NULL.
4. for(CurrNode =HEAD; CurrNode != NULL; CurrNode = CurrNode ->NEXT)
{ if (ITEM = CurrNode ->INFO) then:
{
break the loop
}
Set PrevNode = CurrNode;
} [end of loop]
5. if(CurrNode = NULL) then write : Item not found in the list and return
6. [delete the current node from the list]
Set PrevNode ->NEXT = CurrNode->NEXT
7. Exit.

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

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

void insertAtBeginning(struct Node** head, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
```

`

```
    newNode->next = *head;
    *head = newNode;
}

void insertAtEnd(struct Node** head, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    struct Node* temp = *head;
    newNode->data = value;
    newNode->next = NULL;

    if (*head == NULL) {
        *head = newNode;
        return;
    }

    while (temp->next != NULL) {
        temp = temp->next;
    }

    temp->next = newNode;
}

void insertAtPosition(struct Node** head, int value, int position) {
    if (position <= 0) {
        printf("Invalid position\n");
        return;
    }

    if (position == 1 || *head == NULL) {
        insertAtBeginning(head, value);
        return;
    }

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    struct Node* temp = *head;
    int count = 1;

    while (count < position - 1 && temp->next != NULL) {
        temp = temp->next;
        count++;
    }

    if (count < position - 1) {
        printf("Invalid position\n");
```

```
        return;
    }

    newNode->next = temp->next;
    temp->next = newNode;
}

void displayLinkedList(struct Node* head) {
    struct Node* temp = head;

    if (temp == NULL) {
        printf("Linked list is empty.\n");
        return;
    }

    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }

    printf("NULL\n");
}

int main() {
    struct Node* head = NULL;

    // Insertion at the beginning
    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 20);
    insertAtBeginning(&head, 30);

    printf("Linked list after insertion at the beginning: ");
    displayLinkedList(head);

    // Insertion at the end
    insertAtEnd(&head, 40);
    insertAtEnd(&head, 50);

    printf("Linked list after insertion at the end: ");
    displayLinkedList(head);

    // Insertion at a specific position
    insertAtPosition(&head, 25, 2);
    insertAtPosition(&head, 35, 4);
```

`

```
    printf("Linked list after insertion at specific positions: ");
    displayLinkedList(head);

    return 0;
}
```

Linked list after insertion at the beginning: 30 -> 20 -> 10 -> NULL
Linked list after insertion at the end: 30 -> 20 -> 10 -> 40 -> 50 -> NULL
Linked list after insertion at specific positions: 30 -> 25 -> 20 -> 35 -> 10 -> 40 -> 50 -> NULL

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

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

void insertAtBeginning(struct Node** head, int value) {
    // Create a new node
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

    // Set the value of the new node
    newNode->data = value;

    // Point the new node to the current head
    newNode->next = *head;

    // Update the head to point to the new node
    *head = newNode;
}

void deleteAtBeginning(struct Node** head) {
    if (*head == NULL) {
        printf("Linked list is already empty.\n");
        return;
    }

    struct Node* temp = *head;
    *head = (*head)->next;
    free(temp);
}

void deleteAtEnd(struct Node** head) {
    if (*head == NULL) {
```

`

```
        printf("Linked list is already empty.\n");
        return;
    }

    struct Node* temp = *head;
    struct Node* prev = NULL;

    while (temp->next != NULL) {
        prev = temp;
        temp = temp->next;
    }

    if (prev == NULL) {
        *head = NULL;
    } else {
        prev->next = NULL;
    }

    free(temp);
}

void deleteAtPosition(struct Node** head, int position) {
    if (*head == NULL) {
        printf("Linked list is already empty.\n");
        return;
    }

    struct Node* temp = *head;
    struct Node* prev = NULL;

    if (position == 1) {
        *head = temp->next;
        free(temp);
        return;
    }

    for (int i = 1; temp != NULL && i < position; i++) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("Invalid position.\n");
        return;
    }
```

`

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

void displayList(struct Node* head) {
    struct Node* temp = head;

    // Traverse and print the list
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    struct Node* head = NULL;

    // Insert elements at the beginning
    insertAtBeginning(&head, 3);
    insertAtBeginning(&head, 2);
    insertAtBeginning(&head, 1);

    // Display the original list
    printf("Linked List: ");
    displayList(head);

    // Delete node at the beginning
    deleteAtBeginning(&head);

    // Display the list after deletion at the beginning
    printf("Linked List after deletion at the beginning: ");
    displayList(head);

    // Delete node at the end
    deleteAtEnd(&head);

    // Display the list after deletion at the end
    printf("Linked List after deletion at the end: ");
    displayList(head);

    // Delete node at a specific position
    deleteAtPosition(&head, 1);
```

`

```
    // Display the list after deletion at a specific position
    printf("Linked List after deletion at a specific position: ");
    displayList(head);

    return 0;
}
```

Linked List: 5 4 1 2 3
Linked List after deletion at the beginning: 4 1 2 3
Linked List after deletion at the end: 4 1 2
Linked List after deletion at a specific position: 4 2

Reverse linked list

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

/* Link list node */
struct Node
{
        int data;
        struct Node* next;
};

/* Function to reverse the linked list */
static void reverse(struct Node** head_ref)
{
        struct Node* prev = NULL;
        struct Node* current = *head_ref;
        struct Node* next;
        while (current != NULL)
        {
                next = current->next;
                current->next = prev;
                prev = current;
                current = next;
        }
        *head_ref = prev;
}

/* Function to push a node */
void push(struct Node** head_ref, int new_data)
{
        /* allocate node */
        struct Node* new_node =
                        (struct Node*) malloc(sizeof(struct Node));
```

`

```
        /* put in the data */
        new_node->data = new_data;

        /* link the old list of the new node */
        new_node->next = (*head_ref);

        /* move the head to point to the new node */
        (*head_ref) = new_node;
}

/* Function to print linked list */
void printList(struct Node *head)
{
        struct Node *temp = head;
        while(temp != NULL)
        {
                printf("%d ", temp->data);
                temp = temp->next;
        }
}

/* Driver program to test above function*/
int main()
{
        /* Start with the empty list */
        struct Node* head = NULL;

        push(&head, 20);
        push(&head, 4);
        push(&head, 15);
        push(&head, 85);

        printf("Given linked list\n");
        printList(head);
        reverse(&head);
        printf("\nReversed Linked list \n");
        printList(head);
        getchar();
}
```

Given linked list
85 15 4 20
Reversed Linked list
20 4 15 85

`

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


/* Structure of a node */
struct node {
   int data;         // Data
   struct node *next; // Address
}*head;


void createList(int n);
int countNodes();
void displayList();


int main()
{
   int n, total;

   /*
    * Create a singly linked list of n nodes
    */
   printf("Enter the total number of nodes: ");
   scanf("%d", &n);
   createList(n);

   printf("\nData in the list \n");
   displayList();

   /* Count number of nodes in list */
   total = countNodes();

   printf("\nTotal number of nodes = %d\n", total);

   return 0;
}


/*
 * Create a list of n nodes
 */
void createList(int n)
{
```

`

```
struct node *newNode, *temp;
int data, i;

head = (struct node *)malloc(sizeof(struct node));

/*
 * If unable to allocate memory for head node
 */
if(head == NULL)
{
   printf("Unable to allocate memory.");
}
else
{
   /*
    * Read data of node from the user
    */
   printf("Enter the data of node 1: ");
   scanf("%d", &data);

   head->data = data; // Link data field with data
   head->next = NULL; // Link address field to NULL

   temp = head;

   /*
    * Create n nodes and adds to linked list
    */
   for(i=2; i<=n; i++)
   {
      newNode = (struct node *)malloc(sizeof(struct node));

      /* If memory is not allocated for newNode */
      if(newNode == NULL)
      {
         printf("Unable to allocate memory.");
         break;
      }
      else
      {
         printf("Enter the data of node %d: ", i);
         scanf("%d", &data);

         newNode->data = data; // Link the data field of newNode with data
         newNode->next = NULL; // Link the address field of newNode with NULL
```

`

```
            temp->next = newNode; // Link previous node i.e. temp to the newNode
            temp = temp->next;
         }
      }

      printf("SINGLY LINKED LIST CREATED SUCCESSFULLY\n");
   }
}


/*
 * Counts total number of nodes in the list
 */
int countNodes()
{
   int count = 0;
   struct node *temp;

   temp = head;

   while(temp != NULL)
   {
      count++;
      temp = temp->next;
   }

   return count;
}


/*
 * Displays the entire list
 */
void displayList()
{
   struct node *temp;

   /*
    * If the list is empty i.e. head = NULL
    */
   if(head == NULL)
   {
      printf("List is empty.");
   }
```

```
    else
    {
        temp = head;
        while(temp != NULL)
        {
            printf("Data = %d\n", temp->data); // Print data of current node
            temp = temp->next;                 // Move to next node
        }
    }
}
```

Output

```
Enter the total number of nodes: 4
Enter the data of node 1: 7
Enter the data of node 2: 9
Enter the data of node 3: 8
Enter the data of node 4: 5
SINGLY LINKED LIST CREATED SUCCESSFULLY

Data in the list
Data = 7
Data = 9
Data = 8
Data = 5

Total number of nodes = 4
```

//Function to reverse linked list

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

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

/* Function to reverse the linked list */
static void reverse(struct Node** head_ref)
{
    struct Node* prev   = NULL;
    struct Node* current = *head_ref;
    struct Node* next;
    while (current != NULL)
```

```
    {
        next  = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
}

/* Function to push a node */
void insert(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
            (struct Node*) malloc(sizeof(struct Node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list of the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print linked list */
void printList(struct Node *head)
{
    struct Node *temp = head;
    while(temp != NULL)
    {
        printf("%d  ", temp->data);
        temp = temp->next;
    }
}

/* Driver program to test above function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

     insert(&head, 22);
     insert(&head, 24);
```

`

```
    insert(&head, 15);
    insert(&head, 85);

    printf("Given linked list\n");
    printList(head);
    reverse(&head);
    printf("\nReversed Linked list \n");
    printList(head);
    getchar();
}
```

Output

Given linked list
85 15 24 22
Reversed Linked list
22 24 15 85

Conclusion: This program implements insertion and deletion in singly linked list in all the cases. The function for counting the number of nodes in a linked and reversing the list is also implemented.

`

Experiment No. 5

Aim: Implement a data type to represent a Polynomial with the operations like create an empty polynomial, insert an entry into polynomial, add two polynomials and return the result as a polynomial, evaluate a polynomial, etc.

Theory:

Polynomial addition in data structure

A fundamental mathematical operation, polynomial addition has numerous uses in various disciplines, particularly computer science and data structures. In this thorough investigation, we explore the details of polynomial addition in the context of data structures. Abstractly, polynomials do not merely exist; they also develop algorithms which make it possible to address real-world situations.

Linked lists and other similarities can be used to represent them in terms of data structures. It is an approach for combining two or more polynomials in which terms are sequenced in an organized way. It is needed for many applications, including scientific computing, computer graphics, signal processing, and cryptography.

Understanding Data Structures' Polynomial Representation

Understanding how polynomials are stored in data structures is crucial before diving into polynomial addition. One typical method is to record the coefficients of each phrase and their related exponents in arrays or linked lists.

The polynomial $3x^2+2x+5$ can be represented as an array [3, 2, 5] or a linked list with nodes representing each term.

Consider two polynomials:

$A(x)=4x^3+3x^2+2x+7$

$B(x)=2x^2+5x+1$

A and B's array representations would be [4,3,2,7] and [0,2,5,1], respectively. The coefficients of comparable terms must be added to add these polynomials.

Algorithm

Let's use the sample polynomials A and B from before to demonstrate the algorithm:

- Create an array C that is initially zero-filled and will be used to hold the outcome.
- Repeat each word from A and B in a single iteration.
- Corresponding terms' coefficients are added, and the result is stored in the array C.
- Copy the remaining terms to C if one polynomial has more terms than the other.
- The sum polynomial is represented by the resulting array C.

When A and B are subjected to this method, the sum polynomial results:

$C(x)=4x^3+5x^2+7x+8$

`

C++ Code

```
class Node:
    def __init__(self, coefficient, exponent):
        self.coefficient = coefficient
        self.exponent = exponent
        self.next = None
def add_polynomials(poly1, poly2):
    result_head = Node(0, 0)
    current_result = result_head
    while poly1 is not None or poly2 is not None:
        if poly1 is None:
            current_result.next = poly2
            break
        elif poly2 is None:
            current_result.next = poly1
            break
        if poly1.exponent > poly2.exponent:
            current_result.next = Node(poly1.coefficient, poly1.exponent)
            poly1 = poly1.next
        elif poly1.exponent < poly2.exponent:
            current_result.next = Node(poly2.coefficient, poly2.exponent)
            poly2 = poly2.next
        else:
            new_coefficient = poly1.coefficient + poly2.coefficient
            if new_coefficient != 0:
                current_result.next = Node(new_coefficient, poly1.exponent)
            poly1 = poly1.next
            poly2 = poly2.next
        current_result = current_result.next
    return result_head.next
def display_polynomial(poly):
    while poly is not None:
        print(f"{poly.coefficient}x^{poly.exponent}", end=" ")
        if poly.next is not None and poly.next.coefficient >= 0:
            print("+", end=" ")
        poly = poly.next
    print()
poly1 = Node(3, 2)
poly1.next = Node(2, 1)
poly1.next.next = Node(5, 0)
poly2 = Node(-1, 2)
poly2.next = Node(4, 1)
poly2.next.next = Node(-1, 0)
result = add_polynomials(poly1, poly2)
```

`

```
print("Polynomial 1:", end=" ")
display_polynomial(poly1)
print("Polynomial 2:", end=" ")
display_polynomial(poly2)
print("Sum:", end=" ")
display_polynomial(result)
```

Output:

```
Polynomial 1: 3x^2 + 2x^1 + 5x^0
Polynomial 2: -1x^2 + 4x^1 -1x^0
Sum: 2x^2 + 6x^1 + 4x^0


...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusion: Polynomial addition in the context of data structures has both academic and practical importance. A crucial talent for computer scientists and mathematicians, the manipulation and operation of polynomials are vital in many computing fields.
The representation of polynomials in data structures, the technique for adding polynomials, and concerns for efficiency optimization have all been covered in this investigation.

`

Experiment No. 6

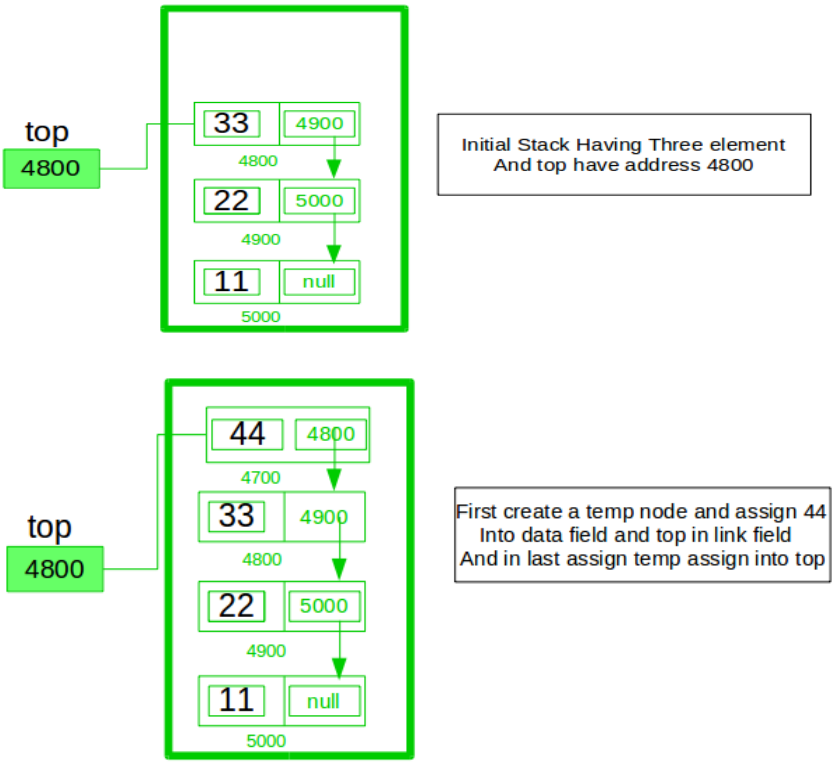
Aim: Implement Stack using a linked list. Use this stack to perform evaluation of a postfix expression

Theory:

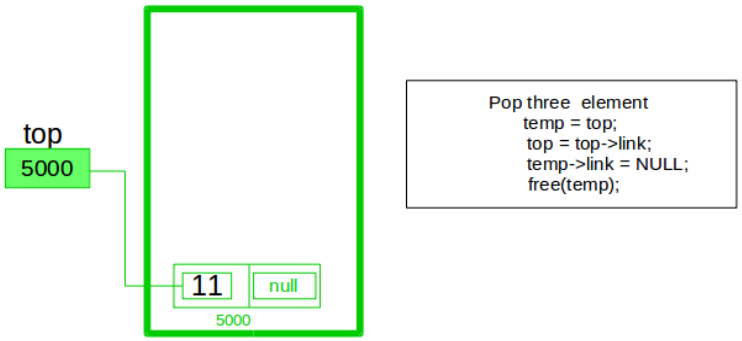
To implement a stack using the singly linked list concept, all the singly linked list operations should be performed based on Stack operations LIFO (last in first out) and with the help of that knowledge, we are going to implement a stack using a singly linked list.

So we need to follow a simple rule in the implementation of a stack which is last in first out and all the operations can be performed with the help of a top variable. Let us learn how to perform Pop, Push, Peek, and Display operations in the following article:

Example:



`



In the stack Implementation, a stack contains a top pointer, which is the "head" of the stack where pushing and popping items happens at the head of the list. The first node has a null in the link field and second node-link has the first node address in the link field and so on and the last node address is in the "top" pointer.
The main advantage of using a linked list over arrays is that it is possible to implement a stack that can shrink or grow as much as needed. Using an array will put a restriction on the maximum capacity of the array which can lead to stack overflow. Here each new node will be dynamically allocated. so overflow is not possible.
Stack Operations:

- push(): Insert a new element into the stack i.e just insert a new element at the beginning of the linked list.
- pop(): Return the top element of the Stack i.e simply delete the first element from the linked list.
- peek(): Return the top element.
- display(): Print all elements in Stack.

Push Operation:

- Initialise a node
- Update the value of that node by data i.e. node->data = data
- Now link this node to the top of the linked list
- And update top pointer to the current node

Pop Operation:

- First Check whether there is any node present in the linked list or not, if not then return
- Otherwise make pointer let say temp to the top node and move forward the top node by 1 step
- Now free this temp node

Peek Operation:

`

- Check if there is any node present or not, if not then return.
- Otherwise return the value of top node of the linked list

Display Operation:

- Take a temp node and initialize it with top pointer
- Now start traversing temp till it encounters NULL
- Simultaneously print the value of the temp node

// C++ code to Implement a stack

```
// using singly linked list
#include <bits/stdc++.h>
using namespace std;
// creating a linked list;
class Node {
public:
        int data;
        Node* link;
        // Constructor
        Node(int n)
        {
                this->data = n;
                this->link = NULL;
        }
};
class Stack {
        Node* top;
public:
        Stack() { top = NULL; }
        void push(int data)
        {
                // Create new node temp and allocate memory in heap
                Node* temp = new Node(data);
                // Check if stack (heap) is full.
                // Then inserting an element would
                // lead to stack overflow
                if (!temp) {
                        cout << "\nStack Overflow";
                        exit(1);
                }

                // Initialize data into temp data field
                temp->data = data;
```

`

```
        // Put top pointer reference into temp link
        temp->link = top;

        // Make temp as top of Stack
        top = temp;
}

// Utility function to check if
// the stack is empty or not
bool isEmpty()
{
        // If top is NULL it means that
        // there are no elements are in stack
        return top == NULL;
}

// Utility function to return top element in a stack
int peek()
{
        // If stack is not empty , return the top element
        if (!isEmpty())
                return top->data;
        else
                exit(1);
}

// Function to remove
// a key from given queue q
void pop()
{
        Node* temp;

        // Check for stack underflow
        if (top == NULL) {
                cout << "\nStack Underflow" << endl;
                exit(1);
        }
        else {

                // Assign top to temp
                temp = top;

                // Assign second node to top
                top = top->link;
```

`

```
                        // This will automatically destroy
                        // the link between first node and second node

                        // Release memory of top node
                        // i.e delete the node
                        free(temp);
                }
        }

        // Function to print all the
        // elements of the stack
        void display()
        {
                Node* temp;

                // Check for stack underflow
                if (top == NULL) {
                        cout << "\nStack Underflow";
                        exit(1);
                }
                else {
                        temp = top;
                        while (temp != NULL) {

                                // Print node data
                                cout << temp->data;

                                // Assign temp link to temp
                                temp = temp->link;
                                if (temp != NULL)
                                        cout << " -> ";
                        }
                }
        }
};

// Driven Program
int main()
{
        // Creating a stack
        Stack s;

        // Push the elements of stack
        s.push(11);
```

`

```
        s.push(22);
        s.push(33);
        s.push(44);

        // Display stack elements
        s.display();

        // Print top element of stack
        cout << "\nTop element is " << s.peek() << endl;

        // Delete top elements of stack
        s.pop();
        s.pop();

        // Display stack elements
        s.display();

        // Print top element of stack
        cout << "\nTop element is " << s.peek() << endl;

        return 0;
}
```

Output

```
44 -> 33 -> 22 -> 11
Top element is 44
22 -> 11
Top element is 22
```

Conclusion: Implementation of stack using linked list allows us to achieve all the functionality of stack also dynamic memory allocation, easy implementation, efficient use of memory and versatility.

`

Experiment No. 7

Aim: Write a function which evaluates an infix expression, without converting it to postfix. The input string can have spaces, (,) and precedence of operators should be handled.

Theory:

Infix expression

An infix expression is an expression in which operators (+, -, *, /) are written between the two operands. For example, consider the following expressions:

1. A + B
2. A + B - C
3. (A + B) + (C - D)

Here we have written '+' operator between the operands A and B, and the - operator in between the C and D operand.

Algorithm: 1. While there are still tokens to be read in,

1.1 Get the next token.

1.2 If the token is:

1.2.1 A number: push it onto the value stack.

1.2.2 A variable: get its value, and push onto the value stack.

1.2.3 A left parenthesis: push it onto the operator stack.

1.2.4 A right parenthesis:

1 While the thing on top of the operator stack is not a left parenthesis,

1 Pop the operator from the operator stack.

2 Pop the value stack twice, getting two operands.

3 Apply the operator to the operands, in the correct order.

4 Push the result onto the value stack.

2 Pop the left parenthesis from the operator stack, and discard it.

1.2.5 An operator (call it thisOp):

1 While the operator stack is not empty, and the top thing on the operator stack has the same or greater precedence as thisOp,

1 Pop the operator from the operator stack.

2 Pop the value stack twice, getting two operands.

3 Apply the operator to the operands, in the correct order.

4 Push the result onto the value stack.

2 Push this Op onto the operator stack.

2. While the operator stack is not empty,

1 Pop the operator from the operator stack.

2 Pop the value stack twice, getting two operands.

3 Apply the operator to the operands, in the correct order.

4 Push the result onto the value stack.

3. At this point the operator stack should be empty, and the value stack should have only one value in it, which is the final result.

`

C++ Code:

```
// expression where tokens are
// separated by space.
#include <bits/stdc++.h>
using namespace std;
 // Function to find precedence of
// operators.
int precedence(char op){
    if(op == '+'||op == '-')
    return 1;
    if(op == '*'||op == '/')
    return 2;
    return 0;
}
 // Function to perform arithmetic operations.
int applyOp(int a, int b, char op){
    switch(op){
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        case '/': return a / b;
    }
}
 // Function that returns value of
// expression after evaluation.
int evaluate(string tokens){
    int i;
         // stack to store integer values.
    stack <int> values;
         // stack to store operators.
    stack <char> ops;
         for(i = 0; i < tokens.length(); i++){
                 // Current token is a whitespace,
        // skip it.
        if(tokens[i] == ' ')
            continue;
         // Current token is an opening
        // brace, push it to 'ops'
        else if(tokens[i] == '('){
            ops.push(tokens[i]);
        }
                 // Current token is a number, push
        // it to stack for numbers.
        else if(isdigit(tokens[i])){
```

`

```
    int val = 0;
         // There may be more than one
     // digits in number.
     while(i < tokens.length() &&
               isdigit(tokens[i]))
     {
        val = (val*10) + (tokens[i]-'0');
        i++;
     }

     values.push(val);

     // right now the i points to
     // the character next to the digit,
     // since the for loop also increases
     // the i, we would skip one
     //  token position; we need to
     // decrease the value of i by 1 to
     // correct the offset.
       i--;
}

// Closing brace encountered, solve
// entire brace.
else if(tokens[i] == ')')
{
     while(!ops.empty() && ops.top() != '(')
     {
        int val2 = values.top();
        values.pop();

        int val1 = values.top();
        values.pop();

        char op = ops.top();
        ops.pop();

        values.push(applyOp(val1, val2, op));
     }

     // pop opening brace.
     if(!ops.empty())
       ops.pop();
}
```

`

```
        // Current token is an operator.
        else
        {
            // While top of 'ops' has same or greater
            // precedence to current token, which
            // is an operator. Apply operator on top
            // of 'ops' to top two elements in values stack.
            while(!ops.empty() && precedence(ops.top())
                        >= precedence(tokens[i])){
                int val2 = values.top();
                values.pop();

                int val1 = values.top();
                values.pop();

                char op = ops.top();
                ops.pop();

                values.push(applyOp(val1, val2, op));
            }

            // Push current token to 'ops'.
            ops.push(tokens[i]);
        }
    }

    // Entire expression has been parsed at this
    // point, apply remaining ops to remaining
    // values.
    while(!ops.empty()){
        int val2 = values.top();
        values.pop();

        int val1 = values.top();
        values.pop();

        char op = ops.top();
        ops.pop();

        values.push(applyOp(val1, val2, op));
    }

    // Top of 'values' contains result, return it.
    return values.top();
}
```

`

```
int main() {
   cout << evaluate("10 + 2 * 6") << "\n";
   cout << evaluate("100 * 2 + 12") << "\n";
   cout << evaluate("100 * ( 2 + 12 )") << "\n";
   cout << evaluate("100 * ( 2 + 12 ) / 14");
   return 0;
}
```

Output:

```
22
212
1400
100
```

Conclusion: Infix expressions are successfully evaluated without converting it into postfix expression.

`

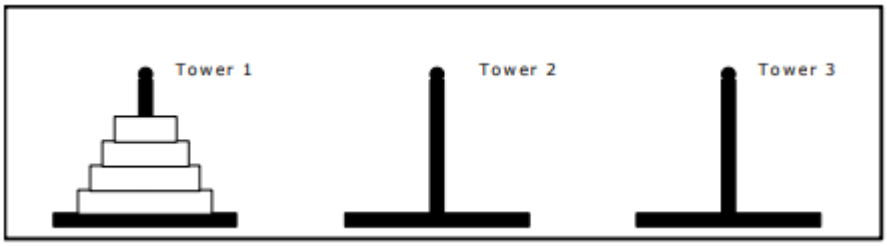
Experiment No. 8

Aim: Implement Tower of Hanoi using Recursion.

Theory:

In the game of Towers of Hanoi, there are three towers labeled 1, 2, and 3. The game starts with n disks on tower A. For simplicity, let n is 3. The disks are numbered from 1 to 3, and without loss of generality we may assume that the diameter of each disk is the same as its number. That is, disk 1 has diameter 1 (in some unit of measure), disk 2 has diameter 2, and disk 3 has diameter 3. All three disks start on tower A in the order 1, 2, 3. The objective of the game is to move all the disks in tower 1 to entire tower 3 using tower 2. That is, at no time can a larger disk be placed on a smaller disk

Following figure, illustrates the initial setup of towers of Hanoi.



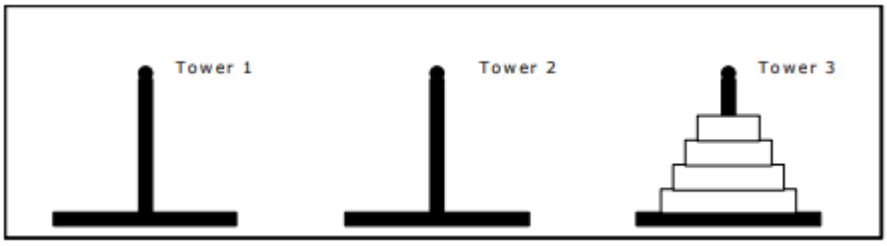
The rules to be followed in moving the disks from tower 1 tower 3 using tower 2 are as follows:

• Only one disk can be moved at a time.

• Only the top disc on any tower can be moved to any other tower.

• A larger disk cannot be placed on a smaller disk.

The figure below, illustrates the final setup of towers of Hanoi



C Code

```
#include <stdio.h>
#include <conio.h>
void towers_of_hanoi (int n, char *a, char *b, char *c);
int cnt=0;
int main (void)
```

`

```
{
int n;
printf("Enter number of discs: ");
scanf("%d",&n);
towers_of_hanoi (n, "Tower 1", "Tower 2", "Tower 3");
getch();
}
void towers_of_hanoi (int n, char *a, char *b, char *c)
{
if (n == 1)
{
 ++cnt;
printf ("\n%5d: Move disk 1 from %s to %s", cnt, a, c);
 return;
}
else
{
towers_of_hanoi (n-1, a, c, b);
 ++cnt;
printf ("\n%5d: Move disk %d from %s to %s", cnt, n, a, c);
towers_of_hanoi (n-1, b, a, c);
 return;
}
}
```

Output

```
Enter the number of discs: 4
1: Move disk 1 from tower 1 to tower 2.
2: Move disk 2 from tower 1 to tower 3.
3: Move disk 1 from tower 2 to tower 3.
4: Move disk 3 from tower 1 to tower 2.
5: Move disk 1 from tower 3 to tower 1.
6: Move disk 2 from tower 3 to tower 2.
7: Move disk 1 from tower 1 to tower 2.
8: Move disk 4 from tower 1 to tower 3.
9: Move disk 1 from tower 2 to tower 3.
10: Move disk 2 from tower 2 to tower 1.
11: Move disk 1 from tower 3 to tower 1.
12: Move disk 3 from tower 2 to tower 3.
13: Move disk 1 from tower 1 to tower 2.
```

`

14: Move disk 2 from tower 1 to tower 3.
15: Move disk 1 from tower 2 to tower 3.

Conclusion: This program implements tower of Hanoi problem using Recursion.

`

Experiment No. 9

Aim: Write a program to simulate deque with functions to add and delete elements from either end of the deque.

Theory:

Double-Ended Queue: A double-ended queue is an abstract data type similar to an simple queue, it allows you to insert and delete from both sides means items can be added or deleted from the front or rear end.

Algorithm for Insertion at rear end

Step -1: [Check for overflow] if(rear==MAX) Print("Queue is Overflow"); return;
Step-2: [Insert element] else
rear=rear+1;
q[rear]=no;
[Set rear and front pointer]
if rear=0
rear=1; if front=0
front=1; Step-3: return

Function of Insertion at rear end

```
void add_item_rear()
{
int num;
printf("\n Enter Item to insert : "); scanf("%d",&num); if(rear==MAX)
{
printf("\n Queue is Overflow"); return;
}
else
{
rear++; q[rear]=num; if(rear==0) rear=1; if(front==0) front=1;
}
}
```

Algorithm for Insertion at font end

Step-1 : [Check for the front position] if(front<=1)
Print ("Cannot add item at front end"); return;
Step-2 : [Insert at front] else
front=front-1; q[front]=no; Step-3 : Return

Function of Insertion at font end

```
void add_item_front()
{
int num;
printf("\n Enter item to insert:"); scanf("%d",&num);
```

```
if(front<=1)
{
printf("\n Cannot add item at front end"); return;
}
else
{
front--; q[front]=num;
}
}
```

Algorithm for Deletion from front end

Step-1 [Check for front pointer] if front=0
print(" Queue is Underflow"); return;
Step-2 [Perform deletion] else
no=q[front];
print("Deleted element is",no); [Set front and rear pointer]
if front=rear front=0; rear=0;
else front=front+1; Step-3 : Return

Function of Deletion from front end

```
void delete_item_front()
{
int num; if(front==0)
{
printf("\n Queue is Underflow\n"); return;
}
else
{
num=q[front];
printf("\n Deleted item is %d\n",num); if(front==rear)
{
front=0; rear=0;
}
else
{
front++;
}
}
}
```

Algorithm for Deletion from rear end

Step-1 : [Check for the rear pointer] if rear=0
print("Cannot delete value at rear end"); return;
Step-2: [perform deletion] else
no=q[rear];

`

[Check for the front and rear pointer] if front= rear
front=0; rear=0; else
rear=rear-1;
print(“Deleted element is”,no); Step-3 : Return

Function of Deletion from rear end

```
void delete_item_rear()
{
int num; if(rear==0)
{
printf("\n Cannot delete item at rear end\n"); return;
}
else
{
num=q[rear]; if(front==rear)
{
front=0; rear=0;
}
else
{
rear--;
printf("\n Deleted item is %d\n",num);
}
}
}
```

C++ Code :

```
#include<iostream>
//#include
//#include
using namespace std;
#define SIZE 5
class dequeue
{
int a[10],front,rear,count;
public:
dequeue();
void add_at_beg(int);
void add_at_end(int);
void delete_fr_front();
void delete_fr_rear();
void display();
};
dequeue::dequeue()
{
```

`

```
front=-1;
rear=-1;
count=0;
}
void dequeue::add_at_beg(int item)
{
int i;
if(front==-1)
{
front++;
rear++;
a[rear]=item;
count++;
}
else if(rear>=SIZE-1)
{

cout<<"\nInsertion is not possible,overflow!!!!";
}
else
{
for(i=count;i>=0;i--)
{
a[i]=a[i-1];
}
a[i]=item;
count++;
rear++;
}
}
void dequeue::add_at_end(int item)
{
if(front==-1)
{
front++;
rear++;
a[rear]=item;
count++;
}
else if(rear>=SIZE-1)
{
cout<<"\nInsertion is not possible,overflow!!!";
return;
}
else
```

`

```
{
a[++rear]=item;
}}
void dequeue::display()
{
for(int i=front;i<=rear;i++)
{

cout<<a[i]<<" "; }
}
void dequeue::delete_fr_front()
{
if(front==-1)
{
cout<<"Deletion is not possible:: Dequeue is empty";
return;
}
else
{
if(front==rear)
{
front=rear=-1;
return;
}
cout<<"The deleted element is "<<a[front];
front=front+1;
}}
void dequeue::delete_fr_rear()
{
if(front==-1)
{
cout<<"Deletion is not possible:Dequeue is empty";
return;
}
else
{
if(front==rear)
{
front=rear=-1;
}
cout<<"The deleted element is "<< a[rear];
rear=rear-1;
}}
```

`

```
int main()
{
int c,item;
dequeue d1;
do
{
cout<<"\n\n****DEQUEUE OPERATION****\n";
cout<<"\n1-Insert at beginning";
cout<<"\n2-Insert at end";
cout<<"\n3_Display";
cout<<"\n4_Deletion from front";
cout<<"\n5-Deletion from rear";
cout<<"\n6_Exit";
cout<<"\nEnter your choice<1-4>:";
cin>>c;
switch(c)
{
case 1:
cout<<"Enter the element to be inserted:";
cin>>item;
d1.add_at_beg(item);
break;
case 2:
cout<<"Enter the element to be inserted:";
cin>>item;
d1.add_at_end(item);
break;
case 3:
d1.display();
break;
case 4:
d1.delete_fr_front();
break;
case 5:
d1.delete_fr_rear();
break;
case 6:
exit(1);
break;
default:
cout<<"Invalid choice";
break;
}
}while(c!=7);
return 0;}
```

`

Output

```
****DEQUEUE OPERATION****
1-Insert at beginning
2-Insert at end
3_Display
4_Deletion from front
5-Deletion from rear
6_Exit
Enter your choice<1-4>:1
Enter the element to be inserted:45

****DEQUEUE OPERATION****
1-Insert at beginning
2-Insert at end
3_Display
4_Deletion from front
5-Deletion from rear
6_Exit
Enter your choice<1-4>:2
Enter the element to be inserted:46

****DEQUEUE OPERATION****
1-Insert at beginning
2-Insert at end
3_Display
4_Deletion from front
5-Deletion from rear
6_Exit
Enter your choice<1-4>:3
45 46

****DEQUEUE OPERATION****
1-Insert at beginning
2-Insert at end
3_Display
4_Deletion from front
5-Deletion from rear
6_Exit
Enter your choice<1-4>:4
The deleted element is 45

****DEQUEUE OPERATION****
1-Insert at beginning
```

`

2-Insert at end
3_Display
4_Deletion from front
5-Deletion from rear
6_Exit
Enter your choice<1-4>:3

****DEQUEUE OPERATION****
1-Insert at beginning
2-Insert at end
3_Display
4_Deletion from front
5-Deletion from rear
6_Exit
Enter your choice<1-4>:5
The deleted element is 0

****DEQUEUE OPERATION****
1-Insert at beginning
2-Insert at end
3_Display
4_Deletion from front
5-Deletion from rear
6_Exit
Enter your choice<1-4>:3
46
****DEQUEUE OPERATION****
1-Insert at beginning
2-Insert at end
3_Display
4_Deletion from front
5-Deletion from rear
6_Exit
Enter your choice<1-4>:

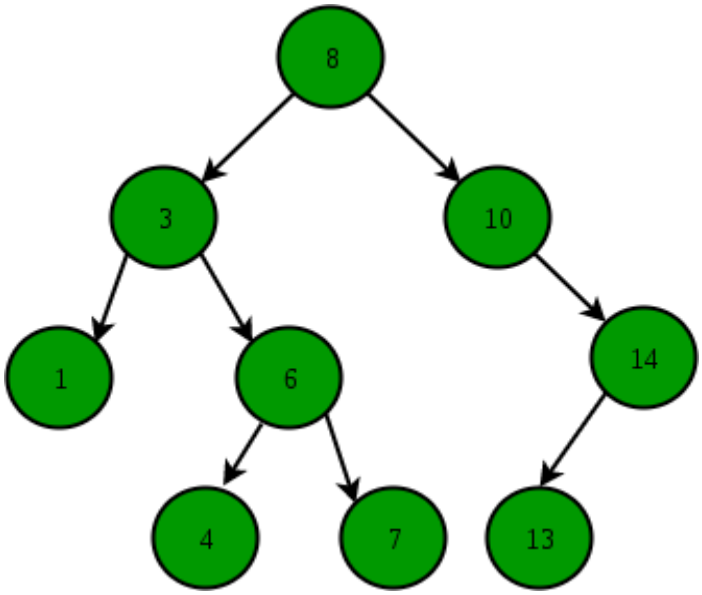
Conclusion: By this way, we can perform operations on double ended queue.

`

Experiment No. 10

Aim: Beginning with an empty binary search tree, Construct binary search tree by inserting the values in the order given. After constructing a binary tree -
i. Insert new node
ii. Find number of nodes in longest path
iii. Minimum data value found in the tree
iv. Change a tree so that the roles of the left and right pointers are swapped at every node
v. Search a value

Theory:
Binary Search Tree is a node-based binary tree data structure which has the following properties:
- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.



ALGORITHM:
Create and Insert:
1. Accept a random order of numbers from the user. The first number would be inserted at the root node.
2. Thereafter, every number is compared with the root node. If less than or equal to the data in the root node, proceed left of the BST, else proceed right.
3. Perform this till you reach a null pointer, the place where the present data is to be inserted.
4. Allocate a new node and assign the data in this node and allocate pointers appropriately.

`

Search in BST

Let's say we want to search for the number X, We start at the root. Then:

- We compare the value to be searched with the value of the root.
  If it's equal we are done with the search if it's smaller we know that we need to go to the left subtree because in a binary search tree all the elements in the left subtree are smaller and all the elements in the right subtree are larger.
- Repeat the above step till no more traversal is possible
- If at any iteration, key is found, return True. Else False.

C++ Code

```
#include<iostream>
using namespace std;
struct node
{
int data;
node *L;
node *R;
};
node *root,*temp;
int count,key;
class bst
{
public:
void create();
void insert(node*,node*);
void disin(node*);
void dispre(node*);
void dispost(node*);
void search(node*,int);
int height(node*);
void mirror(node*);
void min(node*);
bst()
{
root=NULL;
count=0;
}
};
void bst::create()
{
char ans;
do
```

`

```
{
temp=new node;
cout<<"Enter the data : ";
cin>>temp->data;
temp->L=NULL;
temp->R=NULL;
if(root==NULL)
{
root=temp;
}
else
insert(root,temp);
cout<<"Do you want to insert more value : "<<endl;
cin>>ans;
count++;
cout<<endl;
}while(ans=='y');
cout<<"The Total no.of nodes are:" <<count;
}
void bst::insert(node *root,node* temp)
{
if(temp->data>root->data)
{
if(root->R==NULL)
{
root->R=temp;
}
else
insert(root->R,temp);
}
else
{
if(root->L==NULL)
{
root->L=temp;
}
else
insert(root->L,temp);
}
}
void bst::disin(node *root)
{
if(root!=NULL)
{
disin(root->L);
```

`

```
cout<<root->data<<"\t";
disin(root->R);
count++;
}
}
void bst::dispre(node *root)
{
if(root!=NULL)
{
cout<<root->data<<"\t";
dispre(root->L);
dispre(root->R);
}
}
void bst::dispost(node *root)
{
if(root!=NULL)
{
dispost(root->L);
dispost(root->R);
cout<<root->data<<"\t";
}
}
void bst::search(node * root,int key)
{
int flag=0;
cout<<"\nEnter your key : "<<endl;
cin>>key;
temp=root;
while(temp!=NULL)
{
if(key==temp->data)
{
cout<<"KEY FOUND\n";
flag=1;
break;
} node *parent=temp; if(key>parent->data)
{
temp=temp->R;
}
else
{
temp=temp->L;
}
}
```

`

```
if(flag==0)
{
cout<< "KEY NOT FOUND " <<endl;
}
} int bst::height(node *root)
{ int hl,hr;
if(root==NULL)
{ return 0; }
else if(root->L==NULL && root->R==NULL)
{
return 0;
}
cout<<endl; hr=height(root->R);
hl=height(root->L);
if(hr>hl)
{
return(1+hr);
}
else
{
return(1+hl);
}
}
void bst::min(node *root)
{
temp=root;
cout<<endl; while(temp->L!=NULL)
{
temp=temp->L;
}
cout<<root->data;
}
void bst::mirror(node *root)
{
temp=root;
if(root!=NULL)
{
mirror(root->L);
mirror(root->R);
temp=root->L;
root->L=root->R;
root->R=temp;
}
}
int main()
```

`

```
{
bst t;
int ch;
char ans;
do
{
cout<<"\n1) Insert new node 2)number of nodes in longest path 3) minimum 4) mirror 5) search
6) inorder
7) preorder 8) postorder" <<endl;
cin>>ch;
switch(ch)
{
case 1:
t.create();
break;
case 2:
cout<<"\"n Number of nodes in longest path:"<<(1+(t.height(root)));
break;
case 3:
cout<<"\nThe min element is:";
t.min(root);
break;
case 4:
t.mirror(root);
cout<<"\nThe mirror of tree is: ";
t.disin(root);
break;
case 5:
t.search(root,key);
break;
case 6:
cout<<"\n***************INORDER**************" <<endl;
t.disin(root);
break;
case 7:
cout<<"\n***************PREORDER**************"<<endl;
t.dispre(root);
break;
case 8:
cout<<"\n*******************POSTORDER**************"<<endl;
t.dispost(root);
break;
}
cout<<"\nDo you want to continue :"; cin>>ans;
}while(ans=='y');
```

`

```
return 0;
}
```

Output:

```
1) Insert new node 2)number of nodes in longest path 3) minimum 4) mirror 5) search 6)
inorder 7) preorder 8) postorder
1
Enter the data : 3
Do you want to insert more value :
y
Enter the data : 4
Do you want to insert more value :
y
Enter the data : 1
Do you want to insert more value :
y
Enter the data : 6
Do you want to insert more value :
y
Enter the data : 9
Do you want to insert more value :
y
Enter the data : 7
Do you want to insert more value :
n
The Total no.of nodes are:6
Do you want to continue :y
1) Insert new node 2)number of nodes in longest path 3) minimum 4) mirror 5) search 6)
inorder 7) preorder 8) postorder
2
"n Number of nodes in longest path:5
Do you want to continue :y
1) Insert new node 2)number of nodes in longest path 3) minimum 4) mirror 5) search 6)
inorder 7) preorder 8) postorder
3
The min element is:
3
Do you want to continue :y
1) Insert new node 2)number of nodes in longest path 3) minimum 4) mirror 5) search 6)
inorder 7) preorder 8) postorder
4
The mirror of tree is: 9 7 6 4 3 1
Do you want to continue :y
1) Insert new node 2)number of nodes in longest path 3) minimum 4) mirror 5) search 6)
inorder 7) preorder 8) postorder
```

`

5
Enter your key :
1
KEY NOT FOUND
Do you want to continue :y
1) Insert new node 2)number of nodes in longest path 3) minimum 4) mirror 5) search 6) inorder 7) preorder 8) postorder
6
**********INORDER**********
9 7 6 4 3 1
Do you want to continue :y
1) Insert new node 2)number of nodes in longest path 3) minimum 4) mirror 5) search 6) inorder 7) preorder 8) postorder
7
**********PREORDER**********
3 4 6 9 7 1
Do you want to continue :y
1) Insert new node 2)number of nodes in longest path 3) minimum 4) mirror 5) search 6) inorder 7) preorder 8) postorder
8
**********POSTORDER**********
7 9 6 4 1 3
Do you want to continue :y
1) Insert new node 2)number of nodes in longest path 3) minimum 4) mirror 5) search 6) inorder 7) preorder 8) postorder

Conclusion: This program implements Binary Search Tree and various operations on BST.