Experiment 1. Write a program to store k keys into an array of size n at the location computed using a hash function, loc = key % n, where k<=n and k takes values from [1 to m], m>n. To
handle the collisions use the following collision resolution techniques,

A. Linear probing
B. Quadratic probing

Code A: Linear Probing

```
#include <stdio.h>
#include<stdlib.h>
#define TABLE_SIZE 10

int h[TABLE_SIZE]={NULL};

void insert()
{

 int key,index,i,flag=0,hkey;
 printf("\nenter a value to insert into hash table\n");
 scanf("%d",&key);
 hkey=key%TABLE_SIZE;
 for(i=0;i<TABLE_SIZE;i++)
   {

    index=(hkey+i)%TABLE_SIZE;

    if(h[index] == NULL)
    {
       h[index]=key;
        break;
    }

   }

   if(i == TABLE_SIZE)

    printf("\nelement cannot be inserted\n");
}
void search()
{

 int key,index,i,flag=0,hkey;
```

```
 printf("\nenter search element\n");
 scanf("%d",&key);
 hkey=key%TABLE_SIZE;
 for(i=0;i<TABLE_SIZE; i++)
 {
   index=(hkey+i)%TABLE_SIZE;
   if(h[index]==key)
   {
     printf("value is found at index %d",index);
     break;
   }
  }
  if(i == TABLE_SIZE)
    printf("\n value is not found\n");
}
void display()
{

  int i;

  printf("\nelements in the hash table are \n");

  for(i=0;i< TABLE_SIZE; i++)

  printf("\nat index %d \t value =  %d",i,h[i]);

}
main()
{
    int opt,i;
    while(1)
    {
        printf("\nPress 1. Insert\t 2. Display \t3. Search \t4.Exit \n");
        scanf("%d",&opt);
        switch(opt)
        {
            case 1:
                insert();
                break;
            case 2:
                display();
                break;
            case 3:
                search();
```

```
            break;
        case 4:exit(0);
      }
    }
}
```

Output:

```
Press 1. Insert  2. Display     3. Search       4.Exit
1

enter a value to insert into hash table
5

Press 1. Insert  2. Display     3. Search       4.Exit
1

enter a value to insert into hash table
24

Press 1. Insert  2. Display     3. Search       4.Exit
1

enter a value to insert into hash table
33

Press 1. Insert  2. Display     3. Search       4.Exit
1

enter a value to insert into hash table
23
```

```
Press 1. Insert  2. Display     3. Search       4.Exit
2

elements in the hash table are

at index 0       value =  0
at index 1       value =  0
at index 2       value =  0
at index 3       value =  33
at index 4       value =  24
at index 5       value =  5
at index 6       value =  23
at index 7       value =  0
at index 8       value =  0
at index 9       value =  0
Press 1. Insert  2. Display     3. Search       4.Exit
3

enter search element
33
value is found at index 3
Press 1. Insert  2. Display     3. Search       4.Exit
4


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

B. Quadratic probing

Code:

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

// Size of Hash Table
#define SIZE 10

int hashTable[SIZE], c1, c2;

// Initialize the Hash Table with Invalid Value : -1
void init(){
   for (int i = 0;i < SIZE;i++){
      hashTable[i] = -1;
   }
}
```

```
// Display the Hash Table
void displayHashTable(){
    for (int i = 0;i < SIZE;i++){
        printf("| %d ", hashTable[i]);
    }
    printf("|\n");
}

/*
 * Formula: h(k, i) = [h'(k) + c1*i + c2*i*i] mod m
*/
void insertQuad(int key){
    int index = 0, m = SIZE;
    int hKey = key%m;
    for(int i = 0;i < SIZE;i++){
        index = (hKey + i*c1 + i*i*c2)%m;
        if (hashTable[index] == -1){
            hashTable[index] = key;
            return;
        }
    }
    printf("Key Cannot be Placed in Hash Table!\n");
}

// Search for Key
int searchQuad(int toFind){
    int index = 0, m = SIZE;
    int hKey = toFind%m;
    for (int i = 0;i < SIZE;i++){
        index = (hKey + i*c1 + i*i*c2)%m;
        if (hashTable[index] == toFind){
            return index;
        }
        else if (hashTable[index] == -1){
            return -1;
        }
    }
    return -1;
}

// Delete a Key
void deleteQuad(int toDelete){
    int index = searchQuad(toDelete);
```

```
    if (index == -1){
        printf("%d is not Present in the Hash Table, Cannot be Deleted!\n", toDelete);
    }
    else{
        hashTable[index] = -1;
    }
}

// Quadratic Probing
void quadraticProbing(){
    int choice, flag = -1;
    printf("Value of c1 and c2 Constants: ");
    scanf("%d %d", &c1, &c2);
    while(true){
        printf("1. Insert\t 2. Delete\t 3. Search\t 4. Display\n");
        printf("Choice: ");
        scanf("%d", &choice);
        switch(choice){
            case 1:{
                int key;
                printf("Insert Key to Insert: ");
                scanf("%d", &key);
                insertQuad(key);
                break;
            }
            case 2:{
                int toDelete;
                printf("Which Value to Delete: ");
                scanf("%d", &toDelete);
                deleteQuad(toDelete);
                break;
            }
            case 3:{
                int toFind;
                printf("Which Value to Find: ");
                scanf("%d", &toFind);
                int index = searchQuad(toFind);
                if(index == -1){
                    printf("Element Does not Exist in the Hash Table.\n");
                }
                else{
                    printf("%d is Present at %d Index in Hash Table.\n", toFind, index);
                }
                break;
```

```
            }
            case 4:{
                displayHashTable();
                break;
            }
            default:{
                flag = 1;
            }
        }
        if (flag == 1){
            break;
        }
        printf("\n");
    }
}

// The Main Function
int main(void){
    init();
    quadraticProbing();
    return 0;
}
```

Output:

```
Value of c1 and c2 Constants: 0 1
1. Insert        2. Delete       3. Search       4. Display
Choice: 1
Insert Key to Insert: 2

1. Insert        2. Delete       3. Search       4. Display
Choice: 1
Insert Key to Insert: 5

1. Insert        2. Delete       3. Search       4. Display
Choice: 1
Insert Key to Insert: 7

1. Insert        2. Delete       3. Search       4. Display
Choice: 1
Insert Key to Insert: 35

1. Insert        2. Delete       3. Search       4. Display
Choice: 1
Insert Key to Insert: 17

1. Insert        2. Delete       3. Search       4. Display
Choice: 4
| -1 | -1 | 2 | -1 | -1 | 5 | 35 | 7 | 17 | -1 |

1. Insert        2. Delete       3. Search       4. Display
Choice: 3
Which Value to Find: 35
35 is Present at 6 Index in Hash Table.

1. Insert        2. Delete       3. Search       4. Display
Choice: 
```

Exp.2 - Write a program to perform string matching using the Rabin-Karp algorithm.
Code:

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

// d is the number of characters in the input alphabet
#define d 256

void search(char pat[], char txt[], int q)
{
	int M = strlen(pat);
	int N = strlen(txt);
	int i, j;
	int p = 0; // hash value for pattern
	int t = 0; // hash value for txt
	int h = 1;

	// The value of h would be "pow(d, M-1)%q"
	for (i = 0; i < M - 1; i++)
		h = (h * d) % q;

	// Calculate the hash value of pattern and first
	// window of text
	for (i = 0; i < M; i++) {
		p = (d * p + pat[i]) % q;
		t = (d * t + txt[i]) % q;
	}

	// Slide the pattern over text one by one
	for (i = 0; i <= N - M; i++) {

		// Check the hash values of current window of text
		// and pattern. If the hash values match then only
		// check for characters one by one
		if (p == t) {
			/* Check for characters one by one */
			for (j = 0; j < M; j++) {
				if (txt[i + j] != pat[j])
					break;
			}

			// if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]
			if (j == M)
```

```
                printf("Pattern found at index %d \n", i);
        }

        // Calculate hash value for next window of text: Remove
        // leading digit, add trailing digit
        if (i < N - M) {
            t = (d * (t - txt[i] * h) + txt[i + M]) % q;

            // We might get negative value of t, converting it
            // to positive
            if (t < 0)
                t = (t + q);
        }
    }
}

/* Driver program to test above function */
int main()
{
    char txt[] = "This is the test program for Rabin-Karp algorithm";
    char pat[] = "Rabin-Karp";
    int q = 3; // A prime number
    search(pat, txt, q);
    return 0;
}
```

Output:

```
Pattern found at index 29


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

Exp.-3 Write a program to perform string matching using Finite Automata.
Code:

```
#include<stdio.h>
#include<string.h>
#define NO_OF_CHARS 256

int getNextState(char *pat, int M, int state, int x)
{
	// If the character c is same as next character
	// in pattern,then simply increment state
	if (state < M && x == pat[state])
		return state+1;

	// ns stores the result which is next state
	int ns, i;

	// ns finally contains the longest prefix
	// which is also suffix in "pat[0..state-1]c"

	// Start from the largest possible value
	// and stop when you find a prefix which
	// is also suffix
	for (ns = state; ns > 0; ns--)
	{
		if (pat[ns-1] == x)
		{
			for (i = 0; i < ns-1; i++)
				if (pat[i] != pat[state-ns+1+i])
					break;
			if (i == ns-1)
				return ns;
		}
	}

	return 0;
}

/* This function builds the TF table which represents4
	Finite Automata for a given pattern */
void computeTF(char *pat, int M, int TF[][NO_OF_CHARS])
{
	int state, x;
	for (state = 0; state <= M; ++state)
```

```
        for (x = 0; x < NO_OF_CHARS; ++x)
            TF[state][x] = getNextState(pat, M, state, x);
}

/* Prints all occurrences of pat in txt */
void search(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    int TF[M+1][NO_OF_CHARS];

    computeTF(pat, M, TF);

    // Process txt over FA.
    int i, state=0;
    for (i = 0; i < N; i++)
    {
        state = TF[state][txt[i]];
        if (state == M)
            printf ("\n Pattern found at index %d",
                                                    i-M+1);
    }
}

// Driver program to test above function
int main()
{
    char *txt = "AABAACAADAABAAABAA";
    char *pat = "AABA";
    search(pat, txt);
    return 0;
}
```

Output:

```
 Pattern found at index 0
 Pattern found at index 9
 Pattern found at index 13

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

Exp.- 4 – Write a program to perform string matching using Knuth-Morris-Pratt algorithm.
Code:

```
// C++ program for implementation of KMP pattern searching
// algorithm

#include <bits/stdc++.h>

void computeLPSArray(char* pat, int M, int* lps);

// Prints occurrences of pat[] in txt[]
void KMPSearch(char* pat, char* txt)
{
	int M = strlen(pat);
	int N = strlen(txt);

	// create lps[] that will hold the longest prefix suffix
	// values for pattern
	int lps[M];

	// Preprocess the pattern (calculate lps[] array)
	computeLPSArray(pat, M, lps);

	int i = 0; // index for txt[]
	int j = 0; // index for pat[]
	while ((N - i) >= (M - j)) {
		if (pat[j] == txt[i]) {
			j++;
			i++;
		}

		if (j == M) {
			printf("Found pattern at index %d ", i - j);
			j = lps[j - 1];
		}

		// mismatch after j matches
		else if (i < N && pat[j] != txt[i]) {
			// Do not match lps[0..lps[j-1]] characters,
			// they will match anyway
			if (j != 0)
				j = lps[j - 1];
			else
				i = i + 1;
```

```
                }
        }
}

// Fills lps[] for given pattern pat[0..M-1]
void computeLPSArray(char* pat, int M, int* lps)
{
        // length of the previous longest prefix suffix
        int len = 0;

        lps[0] = 0; // lps[0] is always 0

        // the loop calculates lps[i] for i = 1 to M-1
        int i = 1;
        while (i < M) {
                if (pat[i] == pat[len]) {
                        len++;
                        lps[i] = len;
                        i++;
                }
                else // (pat[i] != pat[len])
                {
                        // This is tricky. Consider the example.
                        // AAACAAAA and i = 7. The idea is similar
                        // to search step.
                        if (len != 0) {
                                len = lps[len - 1];

                                // Also, note that we do not increment
                                // i here
                        }
                        else // if (len == 0)
                        {
                                lps[i] = 0;
                                i++;
                        }
                }
        }
}

// Driver code
int main()
{
        char txt[] = "ABABDABACDABABCABAB";
```

```
	char pat[] = "ABABCABAB";
	KMPSearch(pat, txt);
	return 0;
}
```

Output:

```
Found pattern at index 10

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

Exp.-5 – Write a program to perform string matching using Boyer-Moore algorithm.
Code:

```
/* C Program for Bad Character Heuristic of Boyer
Moore String Matching Algorithm */
# include <limits.h>
# include <string.h>
# include <stdio.h>

# define NO_OF_CHARS 256

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

// The preprocessing function for Boyer Moore's
// bad character heuristic
void badCharHeuristic( char *str, int size, int badchar[NO_OF_CHARS])
{
	int i;

	// Initialize all occurrences as -1
	for (i = 0; i < NO_OF_CHARS; i++)
		badchar[i] = -1;

	// Fill the actual value of last occurrence
	// of a character
	for (i = 0; i < size; i++)
		badchar[(int) str[i]] = i;
}

/* A pattern searching function that uses Bad
Character Heuristic of Boyer Moore Algorithm */
void search( char *txt, char *pat)
{
	int m = strlen(pat);
	int n = strlen(txt);

	int badchar[NO_OF_CHARS];

	/* Fill the bad character array by calling
	the preprocessing function badCharHeuristic()
	for given pattern */
	badCharHeuristic(pat, m, badchar);
```

```
    int s = 0; // s is shift of the pattern with
                      // respect to text
    while(s <= (n - m))
    {
        int j = m-1;

        /* Keep reducing index j of pattern while
        characters of pattern and text are
        matching at this shift s */
        while(j >= 0 && pat[j] == txt[s+j])
            j--;

        /* If the pattern is present at current
        shift, then index j will become -1 after
        the above loop */
        if (j < 0)
        {
            printf("\n pattern occurs at shift = %d", s);

            /* Shift the pattern so that the next
            character in text aligns with the last
            occurrence of it in pattern.
            The condition s+m < n is necessary for
            the case when pattern occurs at the end
            of text */
            s += (s+m < n)? m-badchar[txt[s+m]] : 1;

        }

        else
            /* Shift the pattern so that the bad character
            in text aligns with the last occurrence of
            it in pattern. The max function is used to
            make sure that we get a positive shift.
            We may get a negative shift if the last
            occurrence of bad character in pattern
            is on the right side of the current
            character. */
            s += max(1, j - badchar[txt[s+j]]);
    }
}

/* Driver program to test above function */
int main()
```

```
{
	char txt[] = "ABAAABCD";
	char pat[] = "ABC";
	search(txt, pat);
	return 0;
}
```

Output:

```
pattern occurs at shift = 4

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

Experiment 6. Write a program for Binary Search Tree to implement following operations:

- **a. Insertion**
- **b. Deletion**
- **i. Delete node with only child**
- **ii. Delete node with both children**
- **c. Finding an element**
- **d. Finding Min element**
- **e. Finding Max element**
- **f. Left child of the given node**
- **g. Right child of the given node**
- **h. Finding the number of nodes, leaves nodes, full nodes, ancestors, descendants.**

Code:

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

struct node
{
	int info;
	struct node *lchild;
	struct node *rchild;
}*root;



void find(int item,struct node **par,struct node **loc)
{
	struct node *ptr,*ptrsave;

	if(root==NULL)  /*tree empty*/
	{
		*loc=NULL;
		*par=NULL;
		return;
	}
	if(item==root->info) /*item is at root*/
	{
		*loc=root;
		*par=NULL;
		return;
	}
	/*Initialize ptr and ptrsave*/
	if(item<root->info)
```

```
            ptr=root->lchild;
    else
            ptr=root->rchild;
    ptrsave=root;

    while(ptr!=NULL)
    {
            if(item==ptr->info)
            {       *loc=ptr;
                    *par=ptrsave;
                    return;
            }
            ptrsave=ptr;
            if(item<ptr->info)
                    ptr=ptr->lchild;
            else
                    ptr=ptr->rchild;
     }/*End of while */
     *loc=NULL;   /*item not found*/
     *par=ptrsave;
}/*End of find()*/

void insert(int item)
{       struct node *tmp,*parent,*location;
    find(item,&parent,&location);
    if(location!=NULL)
    {
            printf("Item already present");
            return;
    }

    tmp=(struct node *)malloc(sizeof(struct node));
    tmp->info=item;
    tmp->lchild=NULL;
    tmp->rchild=NULL;

    if(parent==NULL)
            root=tmp;
    else
            if(item<parent->info)
                    parent->lchild=tmp;
            else
                    parent->rchild=tmp;
}/*End of insert()*/
```

```
void case_a(struct node *par,struct node *loc )
{
	if(par==NULL) /*item to be deleted is root node*/
		root=NULL;
	else
		if(loc==par->lchild)
			par->lchild=NULL;
		else
			par->rchild=NULL;
}/*End of case_a()*/

void case_b(struct node *par,struct node *loc)
{
	struct node *child;

	/*Initialize child*/
	if(loc->lchild!=NULL) /*item to be deleted has lchild */
		child=loc->lchild;
	else		/*item to be deleted has rchild */
		child=loc->rchild;

	if(par==NULL )   /*Item to be deleted is root node*/
		root=child;
	else
		if( loc==par->lchild)   /*item is lchild of its parent*/
			par->lchild=child;
		else		/*item is rchild of its parent*/
			par->rchild=child;
}/*End of case_b()*/

void case_c(struct node *par,struct node *loc)
{
	struct node *ptr,*ptrsave,*suc,*parsuc;

	/*Find inorder successor and its parent*/
	ptrsave=loc;
	ptr=loc->rchild;
	while(ptr->lchild!=NULL)
	{
		ptrsave=ptr;
		ptr=ptr->lchild;
	}
```

```
        suc=ptr;
        parsuc=ptrsave;

        if(suc->lchild==NULL && suc->rchild==NULL)
                case_a(parsuc,suc);
        else
                case_b(parsuc,suc);

        if(par==NULL) /*if item to be deleted is root node */
                root=suc;
        else
                if(loc==par->lchild)
                        par->lchild=suc;
                else
                        par->rchild=suc;

        suc->lchild=loc->lchild;
        suc->rchild=loc->rchild;
}/*End of case_c()*/
int del(int item)
{
        struct node *parent,*location;
        if(root==NULL)
        {
                printf("Tree empty");
                return 0;
        }

        find(item,&parent,&location);
        if(location==NULL)
        {
                printf("Item not present in tree");
                return 0;
        }

        if(location->lchild==NULL && location->rchild==NULL)
                case_a(parent,location);
        if(location->lchild!=NULL && location->rchild==NULL)
                case_b(parent,location);
        if(location->lchild==NULL && location->rchild!=NULL)
                case_b(parent,location);
        if(location->lchild!=NULL && location->rchild!=NULL)
                case_c(parent,location);
        free(location);
```

```
}/*End of del()*/

int preorder(struct node *ptr)
{
	if(root==NULL)
	{
		printf("Tree is empty");
		return 0;
	}
	if(ptr!=NULL)
	{
		printf("%d  ",ptr->info);
		preorder(ptr->lchild);
		preorder(ptr->rchild);
	}
}/*End of preorder()*/

void inorder(struct node *ptr)
{
	if(root==NULL)
	{
		printf("Tree is empty");
		return;
	}
	if(ptr!=NULL)
	{
		inorder(ptr->lchild);
		printf("%d  ",ptr->info);
		inorder(ptr->rchild);
	}
}/*End of inorder()*/

void postorder(struct node *ptr)
{
	if(root==NULL)
	{
		printf("Tree is empty");
		return;
	}
	if(ptr!=NULL)
	{
		postorder(ptr->lchild);
		postorder(ptr->rchild);
		printf("%d  ",ptr->info);
```

```
	}
}/*End of postorder()*/

void display(struct node *ptr,int level)
{
	int i;
	if ( ptr!=NULL )
	{
		display(ptr->rchild, level+1);
		printf("\n");
		for (i = 0; i < level; i++)
			printf("    ");
		printf("%d", ptr->info);
		display(ptr->lchild, level+1);
	}/*End of if*/
}/*End of display()*/
main()
{
	int choice,num;
	root=NULL;
	while(1)
	{
		printf("\n");
		printf("1.Insert\n");
		printf("2.Delete\n");
		printf("3.Inorder Traversal\n");
		printf("4.Preorder Traversal\n");
		printf("5.Postorder Traversal\n");
		printf("6.Display\n");
		printf("7.Quit\n");
		printf("Enter your choice : ");
		scanf("%d",&choice);

		switch(choice)
		{
		 case 1:
			printf("Enter the number to be inserted : ");
			scanf("%d",&num);
			insert(num);
			break;
		 case 2:
			printf("Enter the number to be deleted : ");
			scanf("%d",&num);
			del(num);
```

```
                break;
            case 3:
                inorder(root);
                break;
            case 4:
                preorder(root);
                break;
            case 5:
                postorder(root);
                break;
            case 6:
                display(root,1);
                break;
            case 7:
        break;
            default:
                printf("Wrong choice\n");
            }/*End of switch */
        }/*End of while */
}/*End of main()*/
```

Output:

```
1.Insert
2.Delete
3.Inorder Traversal
4.Preorder Traversal
5.Postorder Traversal
6.Display
7.Quit
Enter your choice : 1
Enter the number to be inserted : 26

1.Insert
2.Delete
3.Inorder Traversal
4.Preorder Traversal
5.Postorder Traversal
6.Display
7.Quit
Enter your choice : 1
Enter the number to be inserted : 67

1.Insert
2.Delete
3.Inorder Traversal
4.Preorder Traversal
5.Postorder Traversal
6.Display
7.Quit
Enter your choice : 1
Enter the number to be inserted : 10
```

```
Enter the number to be inserted : 10

1.Insert
2.Delete
3.Inorder Traversal
4.Preorder Traversal
5.Postorder Traversal
6.Display
7.Quit
Enter your choice : 1
Enter the number to be inserted : 18

1.Insert
2.Delete
3.Inorder Traversal
4.Preorder Traversal
5.Postorder Traversal
6.Display
7.Quit
Enter your choice : 1
Enter the number to be inserted : 99

1.Insert
2.Delete
3.Inorder Traversal
4.Preorder Traversal
5.Postorder Traversal
6.Display
7.Quit
Enter your choice : 1
```

```
Enter your choice : 1
Enter the number to be inserted : 34

1.Insert
2.Delete
3.Inorder Traversal
4.Preorder Traversal
5.Postorder Traversal
6.Display
7.Quit
Enter your choice : 6

            99
        67
            34
    26
            18
        10
1.Insert
2.Delete
3.Inorder Traversal
4.Preorder Traversal
5.Postorder Traversal
6.Display
7.Quit
Enter your choice : 3
10  18  26  34  67  99
1.Insert
2.Delete
3.Inorder Traversal
4.Preorder Traversal
```

```
5.Postorder Traversal
6.Display
7.Quit
Enter your choice : 3
10   18   26   34   67   99
1.Insert
2.Delete
3.Inorder Traversal
4.Preorder Traversal
5.Postorder Traversal
6.Display
7.Quit
Enter your choice : 4
26   10   18   67   34   99
1.Insert
2.Delete
3.Inorder Traversal
4.Preorder Traversal
5.Postorder Traversal
6.Display
7.Quit
Enter your choice : 5
18   10   34   99   67   26
```

```
1.Insert
2.Delete
3.Inorder Traversal
4.Preorder Traversal
5.Postorder Traversal
6.Display
7.Quit
Enter your choice : 6

            99
        67
            34
    26
            18
        10
1.Insert
2.Delete
3.Inorder Traversal
4.Preorder Traversal
5.Postorder Traversal
6.Display
7.Quit
Enter your choice : 2
Enter the number to be deleted : 26
```

```
Enter the number to be deleted : 26

1.Insert
2.Delete
3.Inorder Traversal
4.Preorder Traversal
5.Postorder Traversal
6.Display
7.Quit
Enter your choice : 6

            99
        67
    34
            18
        10
```

Exp.-7 – Write a program to implement Inorder Threaded Binary Tree with insertion and deletion operation.

Code:

```
# include <stdio.h>
# include <malloc.h>
#define infinity 9999

typedef enum {
   thread, link
} boolean;
struct node *in_succ(struct node *p);
struct node *in_pred(struct node *p);

struct node {
   struct node *left_ptr;
   boolean left;
   int info;
   boolean right;
   struct node *right_ptr;
}*head = NULL;

int main() {
   int choice, num;
   insert_head();
   while (1) {
      printf("\n");
      printf("1.Insert\n");
      printf("2.Inorder Traversal\n");
      printf("3.Quit\n");
      printf("Enter your choice : ");
      scanf("%d", &choice);

      switch (choice) {
      case 1:
         printf("Enter the number to be inserted : ");
         scanf("%d", &num);
         insert(num);
         break;
      case 2:
         inorder();
         break;
      case 3:
```

```
            exit(0);
        default:
            printf("Wrong choice\n");
        }/*End of switch */
    }/*End of while */
}/*End of main()*/

int insert_head() {
    struct node *tmp;
    head = (struct node *) malloc(sizeof(struct node));
    head->info = infinity;
    head->left = thread;
    head->left_ptr = head;
    head->right = link;
    head->right_ptr = head;
}/*End of insert_head()*/

int find(int item, struct node **par, struct node **loc) {
    struct node *ptr, *ptrsave;
    if (head->left_ptr == head) /* If tree is empty*/
    {
        *loc = NULL;
        *par = head;
        return;
    }
    if (item == head->left_ptr->info) /* item is at head->left_ptr */
    {
        *loc = head->left_ptr;
        *par = head;
        return;
    }
    ptr = head->left_ptr;
    while (ptr != head) {
        ptrsave = ptr;
        if (item < ptr->info) {
            if (ptr->left == link)
                ptr = ptr->left_ptr;
            else
                break;
        } else if (item > ptr->info) {
            if (ptr->right == link)
                ptr = ptr->right_ptr;
            else
                break;
```

```
        }
        if (item == ptr->info) {
            *loc = ptr;
            *par = ptrsave;
            return;
        }
    }/*End of while*/
    *loc = NULL; /*item not found*/
    *par = ptrsave;
}/*End of find()*/
```

/* Creating threaded binary search tree */

```
int insert(int item) {
    struct node *tmp, *parent, *location;
    find(item, &parent, &location);

    if (location != NULL) {
        printf("Item already present");
        return;
    }

    tmp = (struct node *) malloc(sizeof(struct node));
    tmp->info = item;
    tmp->left = thread;
    tmp->right = thread;

    if (parent == head) /*tree is empty*/
    {
        head->left = link;
        head->left_ptr = tmp;
        tmp->left_ptr = head;
        tmp->right_ptr = head;
    } else if (item < parent->info) {
        tmp->left_ptr = parent->left_ptr;
        tmp->right_ptr = parent;
        parent->left = link;
        parent->left_ptr = tmp;
    } else {
        tmp->left_ptr = parent;
        tmp->right_ptr = parent->right_ptr;
        parent->right = link;
        parent->right_ptr = tmp;
    }
```

```
}/*End of insert()*/

/* Finding succeeder */

struct node *in_succ(struct node *ptr) {
    struct node *succ;
    if (ptr->right == thread)
        succ = ptr->right_ptr;
    else {
        ptr = ptr->right_ptr;
        while (ptr->left == link)
            ptr = ptr->left_ptr;
        succ = ptr;
    }
    return succ;
}/*End of in_succ()*/

/* Finding predecessor */

struct node *in_pred(struct node *ptr) {
    struct node *pred;
    if (ptr->left == thread)
        pred = ptr->left_ptr;
    else {
        ptr = ptr->left_ptr;
        while (ptr->right == link)
            ptr = ptr->right_ptr;
        pred = ptr;
    }
    return pred;
}/*End of in_pred()*/

/* Displaying all nodes */

inorder() {
    struct node *ptr;
    if (head->left_ptr == head) {
        printf("Tree is empty");
        return;
    }

    ptr = head->left_ptr;

    /*Find the leftmost node and traverse it */
```

```
    while (ptr->left == link)
        ptr = ptr->left_ptr;
    printf("%d ", ptr->info);

    while (1) {
        ptr = in_succ(ptr);
        if (ptr == head) /*If last node reached */
            break;
        printf("%d  ", ptr->info);
    } /*End of while*/
}/*End of inorder()*/
```

Output:

```
1.Insert
2.Inorder Traversal
3.Quit
Enter your choice : 1
Enter the number to be inserted : 20

1.Insert
2.Inorder Traversal
3.Quit
Enter your choice : 1
Enter the number to be inserted : 54

1.Insert
2.Inorder Traversal
3.Quit
Enter your choice : 2
20 54
1.Insert
2.Inorder Traversal
3.Quit
Enter your choice : 1
Enter the number to be inserted : 10

1.Insert
2.Inorder Traversal
3.Quit
Enter your choice : 2
10 20  54
1.Insert
2.Inorder Traversal
3.Quit
Enter your choice :
```

Experiment 7. Write a program to implement Inorder Threaded Binary Tree with insertion and deletion operation.

Experiment 8. Write a program to implement Preorder Threaded Binary Tree with insertion and deletion operation.

Experiment 9. Write a program to implement Postorder Threaded Binary Tree with insertion and deletion operations.

Code:

```
// C++ program for the above approach

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

// Structure of the
// node of a binary tree
struct Node {
	int data;
	struct Node *left, *right;

	Node(int data)
	{
		this->data = data;
		left = right = NULL;
	}
};

// Function to print all nodes of a
// binary tree in Preorder, Postorder
// and Inorder using only one stack
void allTraversal(Node* root)
{
	// Stores preorder traversal
	vector<int> pre;

	// Stores inorder traversal
	vector<int> post;

	// Stores postorder traversal
	vector<int> in;

	// Stores the nodes and the order
```

```
// in which they are currently visited
stack<pair<Node*, int> > s;

// Push root node of the tree
// into the stack
s.push(make_pair(root, 1));

// Traverse the stack while
// the stack is not empty
while (!s.empty()) {

        // Stores the top
        // element of the stack
        pair<Node*, int> p = s.top();

        // If the status of top node
        // of the stack is 1
        if (p.second == 1) {

                // Update the status
                // of top node
                s.top().second++;

                // Insert the current node
                // into preorder, pre[]
                pre.push_back(p.first->data);

                // If left child is not NULL
                if (p.first->left) {

                        // Insert the left subtree
                        // with status code 1
                        s.push(make_pair(
                                p.first->left, 1));
                }
        }

        // If the status of top node
        // of the stack is 2
        else if (p.second == 2) {

                // Update the status
                // of top node
                s.top().second++;
```

```
                // Insert the current node
                // in inorder, in[]
                in.push_back(p.first->data);

                // If right child is not NULL
                if (p.first->right) {

                        // Insert the right subtree into
                        // the stack with status code 1
                        s.push(make_pair(
                                p.first->right, 1));
                }
        }

        // If the status of top node
        // of the stack is 3
        else {

                // Push the current node
                // in post[]
                post.push_back(p.first->data);

                // Pop the top node
                s.pop();
        }
}

cout << "Preorder Traversal: ";
for (int i = 0; i < pre.size(); i++) {
        cout << pre[i] << " ";
}
cout << "\n";

// Printing Inorder
cout << "Inorder Traversal: ";

for (int i = 0; i < in.size(); i++) {
        cout << in[i] << " ";
}
cout << "\n";

// Printing Postorder
cout << "Postorder Traversal: ";
```

```
	for (int i = 0; i < post.size(); i++) {
		cout << post[i] << " ";
	}
	cout << "\n";
}

// Driver Code
int main()
{

	// Creating the root
	struct Node* root = new Node(1);
	root->left = new Node(2);
	root->right = new Node(3);
	root->left->left = new Node(4);
	root->left->right = new Node(5);
	root->right->left = new Node(6);
	root->right->right = new Node(7);

	// Function call
	allTraversal(root);

	return 0;
}
```

Output:

```
Preorder Traversal: 1 2 4 5 3 6 7
Inorder Traversal: 4 2 5 1 6 3 7
Postorder Traversal: 4 5 2 6 7 3 1


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

Exp.-10 – Write a program to transform BST into Threaded Binary Tree.

Code:

```
/* C++ program to convert a Binary Tree to Threaded Tree */
#include <bits/stdc++.h>
using namespace std;

/* Structure of a node in threaded binary tree */
struct Node {
	int key;
	Node *left, *right;

	// Used to indicate whether the right pointer is a
	// normal right pointer or a pointer to inorder
	// successor.
	bool isThreaded;
};

// Helper function to put the Nodes in inorder into queue
void populateQueue(Node* root, std::queue<Node*>* q)
{
	if (root == NULL)
		return;
	if (root->left)
		populateQueue(root->left, q);
	q->push(root);
	if (root->right)
		populateQueue(root->right, q);
}

// Function to traverse queue, and make tree threaded
void createThreadedUtil(Node* root, std::queue<Node*>* q)
{
	if (root == NULL)
		return;

	if (root->left)
		createThreadedUtil(root->left, q);
	q->pop();

	if (root->right)
		createThreadedUtil(root->right, q);
```

```
	// If right pointer is NULL, link it to the
	// inorder successor and set 'isThreaded' bit.
	else {
		root->right = q->front();
		root->isThreaded = true;
	}
}

// This function uses populateQueue() and
// createThreadedUtil() to convert a given binary tree
// to threaded tree.
void createThreaded(Node* root)
{
	// Create a queue to store inorder traversal
	std::queue<Node*> q;

	// Store inorder traversal in queue
	populateQueue(root, &q);

	// Link NULL right pointers to inorder successor
	createThreadedUtil(root, &q);
}

// A utility function to find leftmost node in a binary
// tree rooted with 'root'. This function is used in
// inOrder()
Node* leftMost(Node* root)
{
	while (root != NULL && root->left != NULL)
		root = root->left;
	return root;
}

// Function to do inorder traversal of a threaded binary
// tree
void inOrder(Node* root)
{
	if (root == NULL)
		return;

	// Find the leftmost node in Binary Tree
	Node* cur = leftMost(root);

	while (cur != NULL) {
```

```
		cout << cur->key << " ";

		// If this Node is a thread Node, then go to
		// inorder successor
		if (cur->isThreaded)
			cur = cur->right;

		else // Else go to the leftmost child in right
			// subtree
			cur = leftMost(cur->right);
	}
}

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

// Driver program to test above functions
int main()
{
	/*	 1
			/ \
		2 3
		/ \ / \
		4 5 6 7  */
	Node* root = newNode(1);
	root->left = newNode(2);
	root->right = newNode(3);
	root->left->left = newNode(4);
	root->left->right = newNode(5);
	root->right->left = newNode(6);
	root->right->right = newNode(7);

	createThreaded(root);

	cout << "Inorder traversal of created threaded tree "
			"is\n";
	inOrder(root);
```

```
	return 0;
}
```

Output:

```
Inorder traversal of created threaded tree is
4 2 5 1 6 3 7

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

Exp.- 12 – Write a program to implement Red-Black trees with insertion and deletion operation
for the given input data as Strings.
Code:

```
// Implementing Red-Black Tree in C

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

enum nodeColor {
  RED,
  BLACK
};

struct rbNode {
  int data, color;
  struct rbNode *link[2];
};

struct rbNode *root = NULL;

// Create a red-black tree
struct rbNode *createNode(int data) {
  struct rbNode *newnode;
  newnode = (struct rbNode *)malloc(sizeof(struct rbNode));
  newnode->data = data;
  newnode->color = RED;
  newnode->link[0] = newnode->link[1] = NULL;
  return newnode;
}

// Insert an node
void insertion(int data) {
  struct rbNode *stack[98], *ptr, *newnode, *xPtr, *yPtr;
  int dir[98], ht = 0, index;
  ptr = root;
  if (!root) {
    root = createNode(data);
    return;
  }

  stack[ht] = root;
  dir[ht++] = 0;
```

```
while (ptr != NULL) {
  if (ptr->data == data) {
    printf("Duplicates Not Allowed!!\n");
    return;
  }
  index = (data - ptr->data) > 0 ? 1 : 0;
  stack[ht] = ptr;
  ptr = ptr->link[index];
  dir[ht++] = index;
}
stack[ht - 1]->link[index] = newnode = createNode(data);
while ((ht >= 3) && (stack[ht - 1]->color == RED)) {
  if (dir[ht - 2] == 0) {
    yPtr = stack[ht - 2]->link[1];
    if (yPtr != NULL && yPtr->color == RED) {
      stack[ht - 2]->color = RED;
      stack[ht - 1]->color = yPtr->color = BLACK;
      ht = ht - 2;
    } else {
      if (dir[ht - 1] == 0) {
        yPtr = stack[ht - 1];
      } else {
        xPtr = stack[ht - 1];
        yPtr = xPtr->link[1];
        xPtr->link[1] = yPtr->link[0];
        yPtr->link[0] = xPtr;
        stack[ht - 2]->link[0] = yPtr;
      }
      xPtr = stack[ht - 2];
      xPtr->color = RED;
      yPtr->color = BLACK;
      xPtr->link[0] = yPtr->link[1];
      yPtr->link[1] = xPtr;
      if (xPtr == root) {
        root = yPtr;
      } else {
        stack[ht - 3]->link[dir[ht - 3]] = yPtr;
      }
      break;
    }
  } else {
    yPtr = stack[ht - 2]->link[0];
    if ((yPtr != NULL) && (yPtr->color == RED)) {
      stack[ht - 2]->color = RED;
```

```
        stack[ht - 1]->color = yPtr->color = BLACK;
        ht = ht - 2;
      } else {
        if (dir[ht - 1] == 1) {
          yPtr = stack[ht - 1];
        } else {
          xPtr = stack[ht - 1];
          yPtr = xPtr->link[0];
          xPtr->link[0] = yPtr->link[1];
          yPtr->link[1] = xPtr;
          stack[ht - 2]->link[1] = yPtr;
        }
        xPtr = stack[ht - 2];
        yPtr->color = BLACK;
        xPtr->color = RED;
        xPtr->link[1] = yPtr->link[0];
        yPtr->link[0] = xPtr;
        if (xPtr == root) {
          root = yPtr;
        } else {
          stack[ht - 3]->link[dir[ht - 3]] = yPtr;
        }
        break;
      }
    }
  }
  root->color = BLACK;
}

// Delete a node
void deletion(int data) {
  struct rbNode *stack[98], *ptr, *xPtr, *yPtr;
  struct rbNode *pPtr, *qPtr, *rPtr;
  int dir[98], ht = 0, diff, i;
  enum nodeColor color;

  if (!root) {
    printf("Tree not available\n");
    return;
  }

  ptr = root;
  while (ptr != NULL) {
    if ((data - ptr->data) == 0)
```

```
      break;
    diff = (data - ptr->data) > 0 ? 1 : 0;
    stack[ht] = ptr;
    dir[ht++] = diff;
    ptr = ptr->link[diff];
  }

  if (ptr->link[1] == NULL) {
    if ((ptr == root) && (ptr->link[0] == NULL)) {
      free(ptr);
      root = NULL;
    } else if (ptr == root) {
      root = ptr->link[0];
      free(ptr);
    } else {
      stack[ht - 1]->link[dir[ht - 1]] = ptr->link[0];
    }
  } else {
    xPtr = ptr->link[1];
    if (xPtr->link[0] == NULL) {
      xPtr->link[0] = ptr->link[0];
      color = xPtr->color;
      xPtr->color = ptr->color;
      ptr->color = color;

      if (ptr == root) {
        root = xPtr;
      } else {
        stack[ht - 1]->link[dir[ht - 1]] = xPtr;
      }

      dir[ht] = 1;
      stack[ht++] = xPtr;
    } else {
      i = ht++;
      while (1) {
        dir[ht] = 0;
        stack[ht++] = xPtr;
        yPtr = xPtr->link[0];
        if (!yPtr->link[0])
          break;
        xPtr = yPtr;
      }
```

```
        dir[i] = 1;
        stack[i] = yPtr;
        if (i > 0)
          stack[i - 1]->link[dir[i - 1]] = yPtr;

        yPtr->link[0] = ptr->link[0];

        xPtr->link[0] = yPtr->link[1];
        yPtr->link[1] = ptr->link[1];

        if (ptr == root) {
          root = yPtr;
        }

        color = yPtr->color;
        yPtr->color = ptr->color;
        ptr->color = color;
      }
    }

    if (ht < 1)
      return;

    if (ptr->color == BLACK) {
      while (1) {
        pPtr = stack[ht - 1]->link[dir[ht - 1]];
        if (pPtr && pPtr->color == RED) {
          pPtr->color = BLACK;
          break;
        }

        if (ht < 2)
          break;

        if (dir[ht - 2] == 0) {
          rPtr = stack[ht - 1]->link[1];

          if (!rPtr)
            break;

          if (rPtr->color == RED) {
            stack[ht - 1]->color = RED;
            rPtr->color = BLACK;
            stack[ht - 1]->link[1] = rPtr->link[0];
```

```
      rPtr->link[0] = stack[ht - 1];

      if (stack[ht - 1] == root) {
        root = rPtr;
      } else {
        stack[ht - 2]->link[dir[ht - 2]] = rPtr;
      }
      dir[ht] = 0;
      stack[ht] = stack[ht - 1];
      stack[ht - 1] = rPtr;
      ht++;

      rPtr = stack[ht - 1]->link[1];
    }

    if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
      (!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {
      rPtr->color = RED;
    } else {
      if (!rPtr->link[1] || rPtr->link[1]->color == BLACK) {
        qPtr = rPtr->link[0];
        rPtr->color = RED;
        qPtr->color = BLACK;
        rPtr->link[0] = qPtr->link[1];
        qPtr->link[1] = rPtr;
        rPtr = stack[ht - 1]->link[1] = qPtr;
      }
      rPtr->color = stack[ht - 1]->color;
      stack[ht - 1]->color = BLACK;
      rPtr->link[1]->color = BLACK;
      stack[ht - 1]->link[1] = rPtr->link[0];
      rPtr->link[0] = stack[ht - 1];
      if (stack[ht - 1] == root) {
        root = rPtr;
      } else {
        stack[ht - 2]->link[dir[ht - 2]] = rPtr;
      }
      break;
    }
  } else {
    rPtr = stack[ht - 1]->link[0];
    if (!rPtr)
      break;
```

```
  if (rPtr->color == RED) {
    stack[ht - 1]->color = RED;
    rPtr->color = BLACK;
    stack[ht - 1]->link[0] = rPtr->link[1];
    rPtr->link[1] = stack[ht - 1];

    if (stack[ht - 1] == root) {
      root = rPtr;
    } else {
      stack[ht - 2]->link[dir[ht - 2]] = rPtr;
    }
    dir[ht] = 1;
    stack[ht] = stack[ht - 1];
    stack[ht - 1] = rPtr;
    ht++;

    rPtr = stack[ht - 1]->link[0];
  }
  if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
    (!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {
    rPtr->color = RED;
  } else {
    if (!rPtr->link[0] || rPtr->link[0]->color == BLACK) {
      qPtr = rPtr->link[1];
      rPtr->color = RED;
      qPtr->color = BLACK;
      rPtr->link[1] = qPtr->link[0];
      qPtr->link[0] = rPtr;
      rPtr = stack[ht - 1]->link[0] = qPtr;
    }
    rPtr->color = stack[ht - 1]->color;
    stack[ht - 1]->color = BLACK;
    rPtr->link[0]->color = BLACK;
    stack[ht - 1]->link[0] = rPtr->link[1];
    rPtr->link[1] = stack[ht - 1];
    if (stack[ht - 1] == root) {
      root = rPtr;
    } else {
      stack[ht - 2]->link[dir[ht - 2]] = rPtr;
    }
    break;
  }
}
ht--;
```

```
    }
  }
}

// Print the inorder traversal of the tree
void inorderTraversal(struct rbNode *node) {
  if (node) {
    inorderTraversal(node->link[0]);
    printf("%d  ", node->data);
    inorderTraversal(node->link[1]);
  }
  return;
}

// Driver code
int main() {
  int ch, data;
  while (1) {
    printf("1. Insertion\t2. Deletion\n");
    printf("3. Traverse\t4. Exit");
    printf("\nEnter your choice:");
    scanf("%d", &ch);
    switch (ch) {
      case 1:
        printf("Enter the element to insert:");
        scanf("%d", &data);
        insertion(data);
        break;
      case 2:
        printf("Enter the element to delete:");
        scanf("%d", &data);
        deletion(data);
        break;
      case 3:
        inorderTraversal(root);
        printf("\n");
        break;
      case 4:
        exit(0);
      default:
        printf("Not available\n");
        break;
    }
    printf("\n");
```

```
  }
  return 0;
}
```

Output:

```
1. Insertion     2. Deletion
3. Traverse      4. Exit
Enter your choice:1
Enter the element to insert:20

1. Insertion     2. Deletion
3. Traverse      4. Exit
Enter your choice:1
Enter the element to insert:50

1. Insertion     2. Deletion
3. Traverse      4. Exit
Enter your choice:1
Enter the element to insert:5

1. Insertion     2. Deletion
3. Traverse      4. Exit
Enter your choice:3
5   20   50

1. Insertion     2. Deletion
3. Traverse      4. Exit
Enter your choice:2
Enter the element to delete:20

1. Insertion     2. Deletion
3. Traverse      4. Exit
Enter your choice:3
5   50

1. Insertion     2. Deletion
3. Traverse      4. Exit
Enter your choice:
```

Exp.- 13 – Write a program to transform BST into AVL trees and also count the number rotations performed.

Code:
```
#include <iostream>
#include <vector>

using namespace std;

class Node
{
  public:
    int value;

  Node * left_child;
  Node * right_child;

  Node()
  {
    value = 0;
    left_child = NULL;
    right_child = NULL;
  }

  Node(int v)
  {
    value = v;
    left_child = NULL;
    right_child = NULL;
  }
};

class binarySearchTree
{
  public:
    Node * root;

  binarySearchTree()
  {
    root = NULL;
  }

  int countNodes(Node *root)
  {
    if(root == NULL){
      return 0;
```

```
  }
  else
  {
    return 1 + countNodes(root->left_child) + countNodes(root->right_child);
  }
}

Node * insertNode(Node * root, Node * new_node)
{
  // inserting the node
  if (root == NULL)
  {
    root = new_node;
    return root;
  }

  if (new_node->value < root->value)
  {
    root->left_child = insertNode(root->left_child, new_node);
  }
  else if (new_node->value > root->value)
  {
    root->right_child = insertNode(root->right_child, new_node);
  }
  // node has been inserted

  return root;
}

void storeNodeValues(Node* root, vector<int> &node_values)
{
  if(root!=NULL)
  {
    // in-order traversal and inserting node values in an array
    storeNodeValues(root->left_child, node_values);
    node_values.push_back(root->value);
    storeNodeValues(root->right_child, node_values);
  }
  return;
}

/* Recursive function to construct binary tree */
Node* buildTreeFromArray(vector<int> &node_values)
{
  // base case
  if (node_values.size()==0)
```

```
    {
      return NULL;
    }

    // find the middle element and make it the root
    Node *root = new Node(node_values[node_values.size()/2]);

    // repeat for left_arr
    vector<int> left_arr;
    for(int i=0 ; i<node_values.size()/2 ; i++)
    {
      left_arr.push_back(node_values[i]);
    }
    root->left_child  = buildTreeFromArray(left_arr);

    // repeat for right_arr
    vector<int> right_arr;
    for(int i=(node_values.size()/2)+1 ; i<node_values.size() ; i++)
    {
     right_arr.push_back(node_values[i]);
    }
    root->right_child = buildTreeFromArray(right_arr);

    return root;
  }

  Node* convertBSTtoAVL(Node* root)
  {
    vector<int> node_values;
    storeNodeValues(root, node_values);
    root = buildTreeFromArray(node_values);
    return root;
  }
};

void prettyPrintTree(Node * r, int space)
{
  if (r == NULL)
  {
    return;
  }
  space += 10;
  prettyPrintTree(r->right_child, space);
  cout << endl;
  for (int i = 10; i < space; i++)
  {
```

```
    cout << " ";
  }
  cout << r->value << "\n";
  prettyPrintTree(r->left_child, space);
}

int main()
{
  // creating the BST
  binarySearchTree obj;

  Node * n1 = new Node(4);
  Node * n2 = new Node(3);
  Node * n3 = new Node(2);
  Node * n4 = new Node(1);
  Node * n5 = new Node(5);
  Node * n6 = new Node(6);
  Node * n7 = new Node(7);

  obj.root = obj.insertNode(obj.root, n1);
  obj.root = obj.insertNode(obj.root, n2);
  obj.root = obj.insertNode(obj.root, n3);
  obj.root = obj.insertNode(obj.root, n4);
  obj.root = obj.insertNode(obj.root, n5);
  obj.root = obj.insertNode(obj.root, n6);
  obj.root = obj.insertNode(obj.root, n7);

  cout << "Original BST:" << endl ;
  prettyPrintTree(obj.root, 1);

  obj.root = obj.convertBSTtoAVL(obj.root) ;

  cout << "\nBST after converting it to an AVL tree:" << endl ;
  prettyPrintTree(obj.root, 1);



  return 0 ;
}
```

Output:

```
Original BST:

                               7

                     6

           5

 4

           3

                     2

                               1

BST after converting it to an AVL tree:

                     7

           6

                     5

 4

                     3

           2

                     1
```