VIGNAN'S

INSTITUTE OF INFORMATION TECHNOLOGY (Autonomous)

VISAKHAPATNAM

ADVANCED DATA STURUCTURES LAB RECORD

Department of Computer Science and Engineering

II B. Tech II Sem VR-20

NAME:

REG.NO:

YEAR:_______________SEM:

VIGNAN'S

INSTITUTE OF INFORMATION TECHNOLOGY (Autonomous)

VISAKHAPATNAM

<u>CERTIFICATE</u>

This is to certify that this is the bonafied record of the work done in………………………..Laboratory by Mr./Ms…………………………bearing Reg no/Rol l no. …………………of.......................course during…………...

Total number of
Experiments held……………….

Total number of
Experiments held………………..

LAB-IN-CHARGE

HEAD OF THE DEPARTMENT

INDEX

S.No	Date	Name Of Experiment/Program	Page No.	Remarks
1				
2				
3				
4				
5				
6				
7				
8				

EXPERIMENT-1

1a:

AIM: Implementation of static hashing(Use Linear probing for collision resolution)

Program:

Implementation of Static Hashing

```
#include<stdio.h>
#include<stdlib.h>
struct item
{
    int key;
    int value;
};

struct hashtable_item
{

    int flag;
    struct item *data;
};

struct hashtable_item *array;
int size = 0;
int max = 10;
void init_array()
{
    int i;
    for (i = 0; i < max; i++)
    {
array[i].flag = 0;
array[i].data = NULL;
    }
}
int hashcode(int key)
{
    return (key % max);
}

void insert(int key, int value)
{
    int index = hashcode(key);
    int i = index;
    struct item *new_item = (struct item*) malloc(sizeof(struct item));
```

```
    new_item->key = key;
    new_item->value = value;
    while (array[i].flag == 1)
    {

if (array[i].data->key == key)
        {
 printf("\n Key already exists, hence updating its value \n");
 array[i].data->value = value;
 return;

}

i = (i + 1) % max;
if (i == index)
        {
 printf("\n Hash table is full, cannot insert any more item \n");
 return;
}

    }

    array[i].flag = 1;
    array[i].data = new_item;
    size++;
    printf("\n Key (%d) has been inserted \n", key);

}

void remove_element(int key)
{
    int index = hashcode(key);
    int  i = index;
    while (array[i].flag != 0)
    {

if (array[i].flag == 1  &&  array[i].data->key == key )
        {

 // case when data key matches the given key
 array[i].flag =  2;
 array[i].data = NULL;
 size--;
 printf("\n Key (%d) has been removed \n", key);
 return;
```

```
}
i = (i + 1) % max;
if (i == index)
        {
 break;
}

    }

    printf("\n This key does not exist \n");

}

void display()
{
    int i;
    for (i = 0; i < max; i++)
    {
struct item *current = (struct item*) array[i].data;

if (current == NULL)
        {
    printf("\n Array[%d] has no elements \n", i);
}
else
        {
    printf("\n Array[%d] has elements -: \n  %d (key) and %d(value) ", i, current-
>key, current->value);
}
    }

}

int size_of_hashtable()
{
    return size;
}

void main()
{
int choice, key, value, n, c;
clrscr();

array = (struct hashtable_item*) malloc(max * sizeof(struct hashtable_item*));
init_array();
```

```
do {
 printf(" \n1.Inserting item in the Hashtable"
                              "\n2.Removing item from the Hashtable"
                              "\n3.Check the size of Hashtable"
                              "\n4.Display Hashtable"
        "\n\n Please enter your choice-:");

 scanf("%d", &choice);

 switch(choice)
                {

 case 1:

       printf("Inserting element in Hashtable\n");
       printf("Enter key and value-:\t");
       scanf("%d %d", &key, &value);
       insert(key, value);

       break;

 case 2:

       printf("Deleting in Hashtable \n Enter the key to delete-:");
       scanf("%d", &key);
       remove_element(key);

       break;

 case 3:

       n = size_of_hashtable();
       printf("Size of Hashtable is-:%d\n", n);

       break;

 case 4:

       display();

       break;

 default:

        printf("Wrong Input\n");
```

```
 }

 printf("\n Do you want to continue-:(press 1 for yes)\t");
 scanf("%d", &c);

}while(c == 1);

getch();

}
```

Output :

```
1. Inserting item in the Hashtable
2. Removing item from the Hashtable
3. Check the size of Hashtable
4. Display Hashtable

Please enter your choice-: 3
Size of Hashtable is-: 0

Do you want to continue-:(press 1 for yes) 1

1. Inserting item in the Hashtable
2. Removing item from the Hashtable
3. Check the size of Hashtable
4. Display Hashtable

Please enter your choice-: 1
Inserting element in Hashtable
Enter key and value-:   12          10

 Key (12) has been inserted

Do you want to continue-:(press 1 for yes) 1

1. Inserting item in the Hashtable
2. Removing item from the Hashtable
3. Check the size of Hashtable
4. Display Hashtable

Please enter your choice-: 1
Inserting element in Hash table
Enter key and value-:   122         4
```

Key (122) has been inserted

Do you want to continue-:(press 1 for yes) 1

1. Inserting item in the Hashtable
2. Removing item from the Hashtable
3. Check the size of Hashtable
4. Display Hashtable

Please enter your choice-: 3
Size of Hashtable is-: 2

Do you want to continue-:(press 1 for yes) 1

1. Inserting item in the Hashtable
2. Removing item from the Hashtable
3. Check the size of Hashtable
4. Display Hashtable

Please enter your choice-: 4
Array[0] has no elements

Array[1] has no elements

Array[2] has elements-:
12 (key) and 10 (value)

Array[3] has elements-:
122(key) and 5(value)

Array[4] has no elements

Array[5] has no elements

Array[6] has no elements

Array[7] has no elements

Array[8] has no elements

Array[9] has no elements

Do you want to continue-:(press 1 for yes) 1

1. Inserting item in the Hashtable

2. Removing item from the Hashtable
3. Check the size of Hashtable
4. Display Hashtable

Please enter your choice-: 2
Deleting in Hashtable
Enter the key to delete-: 122

Key (122) has been removed

Do you want to continue-:(press 1 for yes) 1

1. Inserting item in the Hashtable
2. Removing item from the Hashtable
3. Check the size of Hashtable
4. Display Hashtable

Please enter your choice-: 2
Deleting in Hashtable
Enter the key to delete-: 56

This key does not exist

Do you want to continue-:(press 1 for yes) 2

1b:

AIM: Implement Huffman coding

Program:

Implementation of Huffman Coding

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_TREE_HT 50
struct MinHNode {
  char item;
  unsigned freq;
  struct MinHNode *left, *right;
};

struct MinHeap {
  unsigned size;
  unsigned capacity;
  struct MinHNode **array;
};
```

```
struct MinHNode *newNode(char item, unsigned freq) {
  struct MinHNode *temp = (struct MinHNode *)malloc(sizeof(struct MinHNode));

  temp->left = temp->right = NULL;
  temp->item = item;
  temp->freq = freq;

  return temp;
}

struct MinHeap *createMinH(unsigned capacity) {
  struct MinHeap *minHeap = (struct MinHeap *)malloc(sizeof(struct MinHeap));

  minHeap->size = 0;

  minHeap->capacity = capacity;

  minHeap->array = (struct MinHNode **)malloc(minHeap->capacity *
sizeof(struct MinHNode *));
  return minHeap;
}

void swapMinHNode(struct MinHNode **a, struct MinHNode **b) {
  struct MinHNode *t = *a;
  *a = *b;
  *b = t;
}

void minHeapify(struct MinHeap *minHeap, int idx) {
  int smallest = idx;
  int left = 2 * idx + 1;
  int right = 2 * idx + 2;

  if (left < minHeap->size && minHeap->array[left]->freq < minHeap-
>array[smallest]->freq)
    smallest = left;

  if (right < minHeap->size && minHeap->array[right]->freq < minHeap-
>array[smallest]->freq)
    smallest = right;

  if (smallest != idx) {
    swapMinHNode(&minHeap->array[smallest], &minHeap->array[idx]);
    minHeapify(minHeap, smallest);
  }
```

```
}

int checkSizeOne(struct MinHeap *minHeap) {
  return (minHeap->size == 1);
}

struct MinHNode *extractMin(struct MinHeap *minHeap) {
  struct MinHNode *temp = minHeap->array[0];
  minHeap->array[0] = minHeap->array[minHeap->size - 1];

  --minHeap->size;
  minHeapify(minHeap, 0);

  return temp;
}

void insertMinHeap(struct MinHeap *minHeap, struct MinHNode *minHeapNode)
{
  ++minHeap->size;
  int i = minHeap->size - 1;

  while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {
    minHeap->array[i] = minHeap->array[(i - 1) / 2];
    i = (i - 1) / 2;
  }
  minHeap->array[i] = minHeapNode;
}

void buildMinHeap(struct MinHeap *minHeap) {
  int n = minHeap->size - 1;
  int i;

  for (i = (n - 1) / 2; i >= 0; --i)
    minHeapify(minHeap, i);
}

int isLeaf(struct MinHNode *root) {
  return !(root->left) && !(root->right);
}

struct MinHeap *createAndBuildMinHeap(char item[], int freq[], int size) {
  struct MinHeap *minHeap = createMinH(size);

  for (int i = 0; i < size; ++i)
    minHeap->array[i] = newNode(item[i], freq[i]);
```

```
  minHeap->size = size;
  buildMinHeap(minHeap);

  return minHeap;
}

struct MinHNode *buildHuffmanTree(char item[], int freq[], int size) {
  struct MinHNode *left, *right, *top;
  struct MinHeap *minHeap = createAndBuildMinHeap(item, freq, size);

  while (!checkSizeOne(minHeap)) {
    left = extractMin(minHeap);
    right = extractMin(minHeap);

    top = newNode('$', left->freq + right->freq);

    top->left = left;
    top->right = right;

    insertMinHeap(minHeap, top);
  }
  return extractMin(minHeap);
}

void printHCodes(struct MinHNode *root, int arr[], int top) {
  if (root->left) {
    arr[top] = 0;
    printHCodes(root->left, arr, top + 1);
  }
  if (root->right) {
    arr[top] = 1;
    printHCodes(root->right, arr, top + 1);
  }
  if (isLeaf(root)) {
    printf("  %c   | ", root->item);
    printArray(arr, top);
  }
}

void HuffmanCodes(char item[], int freq[], int size) {
  struct MinHNode *root = buildHuffmanTree(item, freq, size);

  int arr[MAX_TREE_HT], top = 0;

  printHCodes(root, arr, top);
}
```

```
void printArray(int arr[], int n) {
  int i;
  for (i = 0; i < n; ++i)
    printf("%d", arr[i]);

  printf("\n");
}

int main() {
  char arr[] = {'A', 'B', 'C', 'D'};
  int freq[] = {5, 1, 6, 3};

  int size = sizeof(arr) / sizeof(arr[0]);

  printf(" Char | Huffman code ");
  HuffmanCodes(arr, freq, size);
}
```

Output :

```
 Char | Huffman code
  C   | 0
  B   | 100
  D   | 101
  A   | 11
```

EXPERIMENT-2

AIM: Write a program to implement AVL tree operations

Program:

AVL Tree Implementation

```
#include<stdio.h>

typedef struct node
{
int data;
struct node *left,*right;
int ht;
}node;

node *insert(node *,int);
node *Delete(node *,int);
void preorder(node *);
void inorder(node *);
int height( node *);
node *rotateright(node *);
node *rotateleft(node *);
node *RR(node *);
node *LL(node *);
node *LR(node *);
node *RL(node *);
int BF(node *);

int main()
{
node *root=NULL;
int x,n,i,op;
do
{
printf("\n1)Create:");
printf("\n2)Insert:");
printf("\n3)Delete:");
printf("\n4)Print:");
printf("\n5)Quit:");
printf("\n\nEnter Your Choice:");
scanf("%d",&op);
switch(op)
{
```

```
case 1: printf("\nEnter no. of elements:");
scanf("%d",&n);
printf("\nEnter tree data:");
root=NULL;
for(i=0;i<n;i++)
{
scanf("%d",&x);
root=insert(root,x);
}
break;
case 2: printf("\nEnter a data:");
scanf("%d",&x);
root=insert(root,x);
break;
case 3: printf("\nEnter a data:");
scanf("%d",&x);
root=Delete(root,x);
break;
case 4: printf("\nPreorder sequence:\n");
preorder(root);
printf("\n\nInorder sequence:\n");
inorder(root);
printf("\n");
break;
}
}while(op!=5);
return 0;


}

node * insert(node *T,int x)
{
if(T==NULL)
{
T=(node*)malloc(sizeof(node));
T->data=x;
T->left=NULL;
T->right=NULL;
}
else
if(x > T->data)
{
T->right=insert(T->right,x);
if(BF(T)==-2)
if(x>T->right->data)
```

```
T=RR(T);
else
T=RL(T);
}
else
if(x<T->data)
{
T->left=insert(T->left,x);
if(BF(T)==2)
if(x < T->left->data)
T=LL(T);
else
T=LR(T);
}
T->ht=height(T);
return(T);
}

node * Delete(node *T,int x)
{
node *p;
if(T==NULL)
{
return NULL;
}
else
if(x > T->data)
{
T->right=Delete(T->right,x);
if(BF(T)==2)
if(BF(T->left)>=0)
T=LL(T);
else
T=LR(T);
}
else
if(x<T->data)
{
T->left=Delete(T->left,x);
if(BF(T)==-2)
if(BF(T->right)<=0)
T=RR(T);
else
T=RL(T);
}
else
```

```
{

if(T->right!=NULL)
{
p=T->right;
while(p->left!= NULL)


p=p->left;
T->data=p->data;
T->right=Delete(T->right,p->data);
if(BF(T)==2)
if(BF(T->left)>=0)
T=LL(T);
else
T=LR(T);
}
else
return(T->left);
}
T->ht=height(T);
return(T);
}

int height(node *T)
{
int lh,rh;
if(T==NULL)
return(0);
if(T->left==NULL)
lh=0;
else
lh=1+T->left->ht;
if(T->right==NULL)
rh=0;
else
rh=1+T->right->ht;
if(lh>rh)
return(lh);

return(rh);
}

node * rotateright(node *x)
{
node *y;
```

```
y=x->left;
x->left=y->right;
y->right=x;
x->ht=height(x);
y->ht=height(y);
return(y);
}

node * rotateleft(node *x)
{
node *y;
y=x->right;
x->right=y->left;
y->left=x;
x->ht=height(x);
y->ht=height(y);
return(y);
}

node * RR(node *T)
{
T=rotateleft(T);
return(T);
}

node * LL(node *T)
{
T=rotateright(T);
return(T);
}

node * LR(node *T)
{
T->left=rotateleft(T->left);
T=rotateright(T);
return(T);
}

node * RL(node *T)
{
T->right=rotateright(T->right);
T=rotateleft(T);
return(T);
}

int BF(node *T)
```

```
{
int lh,rh;
if(T==NULL)
return(0);

if(T->left==NULL)
lh=0;
else
lh=1+T->left->ht;

if(T->right==NULL)
rh=0;
else
rh=1+T->right->ht;

return(lh-rh);
}

void preorder(node *T)
{
if(T!=NULL)
{
printf("%d(Bf=%d)",T->data,BF(T));
preorder(T->left);
preorder(T->right);
}
}

void inorder(node *T)
{
if(T!=NULL)
{
inorder(T->left);
printf("%d(Bf=%d)",T->data,BF(T));
inorder(T->right);
}
}
```

Output :

1)Create:
2)Insert:
3)Delete:
4)Print:
5)Quit:

Enter Your Choice:1

Enter no. of elements:4

Enter tree data:7 12 4 9

1) Create:
2) Insert:
3) Delete:
4) Print:
5) Quit:

Enter Your Choice:4

Preorder sequence:
7(Bf=-1)4(Bf=0)12(Bf=1)9(Bf=0)

Inorder sequence:
4(Bf=0)7(Bf=-1)9(Bf=0)12(Bf=1)

1)Create:
2)Insert:
3)Delete:
4) Print:
5) Quit:

Enter Your Choice:3

Enter a data:7

1)Create:
2)Insert:
3)Delete:
4) Print:
5) Quit:

Enter Your Choice:4

Preorder sequence:
9(Bf=0)4(Bf=0)12(Bf=0)

Inorder sequence:
4(Bf=0)9(Bf=0)12(Bf=0)

1)Create:
2)Insert:
3)Delete:

4)Print:
5)Quit:

Enter Your Choice:5

EXPERIMENT-3

AIM: Write a program to implement Red-Black tree operations

Program:

Red-Black Tree Implementation

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

enum nodeColor {
  RED,
  BLACK
};

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

struct rbNode *root = NULL;
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;
}

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;
}

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--;
    }
  }
}

void inorderTraversal(struct rbNode *node) {
  if (node) {
    inorderTraversal(node->link[0]);
    printf("%d ", node->data);
    inorderTraversal(node->link[1]);
  }
  return;
}

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:23

1. Insertion	2. Deletion
3. Traverse	4. Exit
Enter your choice:1
Enter the element to insert:12

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

Enter the element to insert:45

1. Insertion 2. Deletion
3. Traverse 4. Exit
Enter your choice:1
Enter the element to insert:60

1. Insertion 2. Deletion
3. Traverse 4. Exit
Enter your choice:1
Enter the element to insert:78

1. Insertion 2. Deletion
3. Traverse 4. Exit
Enter your choice:3
12 23 45 60 78

1. Insertion 2. Deletion
3. Traverse 4. Exit
Enter your choice:2
Enter the element to delete:45

1. Insertion 2. Deletion
3. Traverse 4. Exit
Enter your choice:3
12 23 60 78

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

EXPERIMENT-4

AIM: Write a program to implement binomial queues

Program:

Implementation of Bionomial Queues

```
#include <stdio.h>
#define MAX 50
void   insert();
void   delete();
void display();
int queue_array[MAX];
int rear = - 1;
int front = - 1;
main()
{
   int choice;
   while (1)
   {
      printf("1.Insert element to queue \n");
      printf("2.Delete element from queue \n");
      printf("3.Display all elements of queue \n");
      printf("4.Quit \n");
      printf("Enter your choice : ");
      scanf("%d",  &choice);
      switch (choice)
      {
         case 1:
         insert();
         break;
         case 2:
         delete();
         break;
         case 3:
         display();
         break;
         case 4:
         exit(1);
         default:
         printf("Wrong choice \n");
      }
   }
}
```

```
void insert()
{
   int add_item;
   if (rear == MAX - 1)
   printf("Queue Overflow \n");
   else
   {
      if (front == - 1)
      /*If queue is initially empty */
      front = 0;
      printf("Inset the element in queue : ");
      scanf("%d", &add_item);
      rear = rear + 1;
      queue_array[rear] = add_item;
   }
}

void delete()
{
   if (front == - 1 || front > rear)
   {
      printf("Queue Underflow \n");
      return ;
   }
   else
   {
      printf("Element deleted from queue is : %d\n", queue_array[front]);
      front = front + 1;
   }
}

void display()
{
   int i;
   if (front == - 1)
      printf("Queue is empty \n");
   else
   {
      printf("Queue is : \n");
      for (i = front; i <= rear; i++)
         printf("%d ", queue_array[i]);
      printf("\n");
   }
}
```

Output :

```
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 10
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 15
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 20
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 30
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 2
Element deleted from queue is : 10
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 3
Queue is :
15 20 30
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 4
```

EXPERIMENT-5

AIM: Write a program to implement Heap sort

Program:

implementation of Heap Sort

```
#include <stdio.h>
void swap(int* a, int* b)
{

int temp = *a;
*a = *b;
*b = temp;
}

void heapify(int arr[], int N, int i)
{
int largest = i;
int left = 2 * i + 1;
int right = 2 * i + 2;
if (left < N && arr[left] > arr[largest])
        largest = left;
if (right < N && arr[right] > arr[largest])
        largest = right;
if (largest != i) {

swap(&arr[i], &arr[largest]);
heapify(arr, N, largest);
}
}

void heapSort(int arr[], int N)
{
for (int i = N / 2 - 1; i >= 0; i--)

 heapify(arr, N, i);
for (int i = N - 1; i >= 0; i--) {

 swap(&arr[0], &arr[i]);
 heapify(arr, i, 0);
}
}
```

```
void printArray(int arr[], int N)
{
for (int i = 0; i < N; i++)
printf("%d ", arr[i]);
printf("\n");
}

int main()
{
int arr[] = { 12, 11, 13, 5, 6, 7 };
int N = sizeof(arr) / sizeof(arr[0]);
heapSort(arr, N);
printf("Sorted array is\n");
printArray(arr, N);
}
```

Output :

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

EXPERIMENT-6

AIM: Write a program to implement B-tree

Program:

Implementation of B-Tree

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

#define MAX 4
#define MIN 2

struct btreeNode {
      int val[MAX + 1], count;
      struct btreeNode *link[MAX + 1];
};

struct btreeNode *root;
struct btreeNode * createNode(int val, struct btreeNode *child) {
      struct btreeNode *newNode;
      newNode = (struct btreeNode *)malloc(sizeof(struct btreeNode));
      newNode->val[1] = val;
      newNode->count = 1;
      newNode->link[0] = root;
      newNode->link[1] = child;
      return newNode;
}

void addValToNode(int val, int pos, struct btreeNode *node,
                  struct btreeNode *child) {
      int j = node->count;
      while (j > pos) {
            node->val[j + 1] = node->val[j];
            node->link[j + 1] = node->link[j];
            j--;
      }
      node->val[j + 1] = val;
      node->link[j + 1] = child;
      node->count++;
}

void splitNode (int val, int *pval, int pos, struct btreeNode *node,
   struct btreeNode *child, struct btreeNode **newNode) {
```

```
    int median, j;

    if (pos > MIN)
        median = MIN + 1;
    else
        median = MIN;

    *newNode = (struct btreeNode *)malloc(sizeof(struct btreeNode));
    j = median + 1;
    while (j <= MAX) {
        (*newNode)->val[j - median] = node->val[j];
        (*newNode)->link[j - median] = node->link[j];
        j++;
    }
    node->count = median;
    (*newNode)->count = MAX - median;

    if (pos <= MIN) {
        addValToNode(val, pos, node, child);
    } else {
        addValToNode(val, pos - median, *newNode, child);
    }
    *pval = node->val[node->count];
    (*newNode)->link[0] = node->link[node->count];
    node->count--;
}

int setValueInNode(int val, int *pval,
  struct btreeNode *node, struct btreeNode **child) {

    int pos;
    if (!node) {
        *pval = val;
        *child = NULL;
        return 1;
    }

    if (val < node->val[1]) {
        pos = 0;
    } else {
        for (pos = node->count;
            (val < node->val[pos] && pos > 1); pos--);
        if (val == node->val[pos]) {
            printf("Duplicates not allowed\n");
            return 0;
        }
```

```
        }
        if (setValueInNode(val, pval, node->link[pos], child)) {
                if (node->count < MAX) {
                        addValToNode(*pval, pos, node, *child);
                } else {
                        splitNode(*pval, pval, pos, node, *child, child);
                        return 1;
                }
        }
        return 0;
}

void insertion(int val) {
        int flag, i;
        struct btreeNode *child;

        flag = setValueInNode(val, &i, root, &child);
        if (flag)
                root = createNode(i, child);
}

void copySuccessor(struct btreeNode *myNode, int pos) {
        struct btreeNode *dummy;
        dummy = myNode->link[pos];

        for (;dummy->link[0] != NULL;)
                dummy = dummy->link[0];
        myNode->val[pos] = dummy->val[1];

}

void removeVal(struct btreeNode *myNode, int pos) {
        int i = pos + 1;
        while (i <= myNode->count) {
                myNode->val[i - 1] = myNode->val[i];
                myNode->link[i - 1] = myNode->link[i];
                i++;
        }
        myNode->count--;
}

void doRightShift(struct btreeNode *myNode, int pos) {
        struct btreeNode *x = myNode->link[pos];
        int j = x->count;

        while (j > 0) {
```

```
            x->val[j + 1] = x->val[j];
            x->link[j + 1] = x->link[j];
        }
        x->val[1] = myNode->val[pos];
        x->link[1] = x->link[0];
        x->count++;

        x = myNode->link[pos - 1];
        myNode->val[pos] = x->val[x->count];
        myNode->link[pos] = x->link[x->count];
        x->count--;
        return;
}

void doLeftShift(struct btreeNode *myNode, int pos) {
        int j = 1;
        struct btreeNode *x = myNode->link[pos - 1];

        x->count++;
        x->val[x->count] = myNode->val[pos];
        x->link[x->count] = myNode->link[pos]->link[0];

        x = myNode->link[pos];
        myNode->val[pos] = x->val[1];
        x->link[0] = x->link[1];
        x->count--;

        while (j <= x->count) {
            x->val[j] = x->val[j + 1];
            x->link[j] = x->link[j + 1];
            j++;
        }
        return;
}

void mergeNodes(struct btreeNode *myNode, int pos) {
        int j = 1;
        struct btreeNode *x1 = myNode->link[pos], *x2 = myNode->link[pos - 1];

        x2->count++;
        x2->val[x2->count] = myNode->val[pos];
        x2->link[x2->count] = myNode->link[0];

        while (j <= x1->count) {
            x2->count++;
            x2->val[x2->count] = x1->val[j];
```

```
		x2->link[x2->count] = x1->link[j];
		j++;
	}

	j = pos;
	while (j < myNode->count) {
		myNode->val[j] = myNode->val[j + 1];
		myNode->link[j] = myNode->link[j + 1];
		j++;
	}
	myNode->count--;
	free(x1);
}

void adjustNode(struct btreeNode *myNode, int pos) {
	if (!pos) {
		if (myNode->link[1]->count > MIN) {
			doLeftShift(myNode, 1);
		} else {
			mergeNodes(myNode, 1);
		}
	} else {
		if (myNode->count != pos) {
			if(myNode->link[pos - 1]->count > MIN) {
				doRightShift(myNode, pos);
			} else {
				if (myNode->link[pos + 1]->count > MIN) {
					doLeftShift(myNode, pos + 1);
				} else {
					mergeNodes(myNode, pos);
				}
			}
		} else {
			if (myNode->link[pos - 1]->count > MIN)
				doRightShift(myNode, pos);
			else
				mergeNodes(myNode, pos);
		}
	}
}

int delValFromNode(int val, struct btreeNode *myNode) {
	int pos, flag = 0;
	if (myNode) {
		if (val < myNode->val[1]) {
			pos = 0;
```

```
                flag = 0;
            } else {
                for (pos = myNode->count;
                    (val < myNode->val[pos] && pos > 1); pos--);
                 if (val == myNode->val[pos]) {
                    flag = 1;
                } else {
                    flag = 0;
                }
            }
            if (flag) {
                if (myNode->link[pos - 1]) {
                    copySuccessor(myNode, pos);
                    flag = delValFromNode(myNode->val[pos], myNode-
>link[pos]);
                    if (flag == 0) {
                        printf("Given data is not present in B-Tree\n");
                    }
                } else {
                    removeVal(myNode, pos);
                }
            } else {
                flag = delValFromNode(val, myNode->link[pos]);
            }
            if (myNode->link[pos]) {
                if (myNode->link[pos]->count < MIN)
                    adjustNode(myNode, pos);
            }
        }
        return flag;
    }

  void deletion(int val, struct btreeNode *myNode) {
        struct btreeNode *tmp;
        if (!delValFromNode(val, myNode)) {
            printf("Given value is not present in B-Tree\n");
            return;
        } else {
            if (myNode->count == 0) {
                tmp = myNode;
                myNode = myNode->link[0];
                free(tmp);
            }
        }
        root = myNode;
        return;
```

```
}

void searching(int val, int *pos, struct btreeNode *myNode) {
    if (!myNode) {
        return;
    }

    if (val < myNode->val[1]) {
        *pos = 0;
    } else {
        for (*pos = myNode->count;
            (val < myNode->val[*pos] && *pos > 1); (*pos)--);
        if (val == myNode->val[*pos]) {
            printf("Given data %d is present in B-Tree", val);
            return;
        }
    }
    searching(val, pos, myNode->link[*pos]);
    return;
}

void traversal(struct btreeNode *myNode) {
    int i;
    if (myNode) {
        for (i = 0; i < myNode->count; i++) {
            traversal(myNode->link[i]);
            printf("%d ", myNode->val[i + 1]);
        }
        traversal(myNode->link[i]);
    }
}

int main() {
    int val, ch;
    while (1) {
        printf("1. Insertion\t2. Deletion\n");
        printf("3. Searching\t4. Traversal\n");
        printf("5. Exit\nEnter your choice:");
        scanf("%d", &ch);
        switch (ch) {
            case 1:
                printf("Enter your input:");
                scanf("%d", &val);
                insertion(val);
                break;
            case 2:
```

```
                printf("Enter the element to delete:");
                scanf("%d", &val);
                deletion(val, root);
                break;
            case 3:
                printf("Enter the element to search:");
                scanf("%d", &val);
                searching(val, &ch, root);
                break;
            case 4:
                traversal(root);
                break;
            case 5:
                exit(0);
            default:
                printf("U have entered wrong option!!\n");
                break;
        }
        printf("\n");
    }
}
```

Output :

```
1. Insertion 2. Deletion
 3. Searching 4. Traversal
 5. Exit
 Enter your choice:1
 Enter your input:70

 1. Insertion 2. Deletion
 3. Searching 4. Traversal
 5. Exit
 Enter your choice:1
 Enter your input:17

 1. Insertion 2. Deletion
 3. Searching 4. Traversal
 5. Exit
 Enter your choice:1
 Enter your input:67

 1. Insertion 2. Deletion
 3. Searching 4. Traversal
 5. Exit
 Enter your choice:1
```

Enter your input:89

1. Insertion 2. Deletion
3. Searching 4. Traversal
5. Exit
Enter your choice:4
17 67 70 89

1. Insertion 2. Deletion
3. Searching 4. Traversal
5. Exit
Enter your choice:3
Enter the element to search:70
Given data 70 is present in B-Tree

1. Insertion 2. Deletion
3. Searching 4. Traversal
5. Exit
Enter your choice:2
Enter the element to delete:17

1. Insertion 2. Deletion
3. Searching 4. Traversal
5. Exit
Enter your choice:4
67 70 89

1. Insertion 2. Deletion
3. Searching 4. Traversal
5. Exit
Enter your choice:5

EXPERIMENT-7

AIM: Write a program to implement B+ Trees

Program:

Implementation of B+ trees

```
#include<stdio.h>
#include<conio.h>
#define Macro 4
struct node
{
int n;
int keys[Macro - 1];
struct node *p[Macro];
}*root=NULL;
enum KeyStatus { dupl,SearchFailure,Success,insrit,LessKeys };
void insert(int x);
void display(struct node *root,int);
void delete(int x);
enum KeyStatus ins(struct node *r, int x, int* y, struct node** u);
int searchPos(int x,int *key_arr, int n);
enum KeyStatus del(struct node *r, int x);
void main()
{
int k;
int op;
for(;;)
{
printf("\n1.Insert");
printf("\n2.Delete");
printf("\n3.Display");
printf("\n4.Quit");
printf("\nEnter the option : ");
scanf("%d",&op);
switch(op)
{
case 1:
printf("\nEnter the element : ");
scanf("%d",&k);
insert(k);
break;
case 2:
printf("\nEnter the element : ");
```

```
scanf("%d",&k);
delete(k);
break;
case 3:
printf("\nB+ tree is :\n");
display(root,0);
break;
case 4:
exit(1);
default:
printf("\nInvalid Option.");
break;
}
}
}
void insert(int k)
{
struct node *newnode;
int upK;
enum KeyStatus val;
val = ins(root, k, &upK, &newnode);
if (val == dupl)
printf("\nElement already present.");
if (val == insrit)
{
struct node *uproot = root;
root=malloc(sizeof(struct node));
root -> n = 1;
root -> keys[0] = upK;
root -> p[0] = uproot;
root->p[1] = newnode;
}
}
enum KeyStatus ins(struct node *ptr, int k, int *upK,struct node **newnode)
{
struct node *nptr, *lptr;
int  pos, i, n,splitPos;
int nK, lK;
enum KeyStatus val;
if (ptr == NULL)
{
*newnode = NULL;
*upK = k;
return insrit;
}
n = ptr -> n;
```

```
pos = searchPos(k, ptr->keys, n);
if (pos < n && k == ptr -> keys[pos])
return dupl;
val = ins(ptr->p[pos], k, &nK, &nptr);
if (val != insrit)
return val;
if (n < Macro - 1)
{
pos = searchPos(nK, ptr->keys, n);
for (i=n; i>pos; i--)
{
ptr -> keys[i] = ptr -> keys[i-1];
ptr -> p[i+1] = ptr -> p[i];
}
ptr -> keys[pos] = nK;
ptr -> p[pos+1] = nptr;
++ptr -> n;
return Success;
}
if (pos == Macro - 1)
{
lK = nK;
lptr = nptr;
}
else
{
lK = ptr -> keys[Macro-2];
lptr = ptr -> p[Macro-1];
for (i = Macro - 2; i > pos; i--)
{
ptr->keys[i] = ptr->keys[i-1];
ptr->p[i+1] = ptr->p[i];
}
ptr -> keys[pos] = nK;
ptr -> p[pos+1] = nptr;
}
splitPos = (Macro - 1)/2;
(*upK) = ptr->keys[splitPos];
(*newnode)=malloc(sizeof(struct node));
ptr -> n = splitPos;
(*newnode)->n = Macro - 1 - splitPos;
for (i=0; i < (*newnode)->n; i++)
{
(*newnode)->p[i] = ptr->p[i + splitPos + 1];
if(i < (*newnode)->n - 1)
(*newnode)->keys[i] = ptr->keys[i + splitPos + 1];
```

```
else
(*newnode)->keys[i] = lK;
}
(*newnode)->p[(*newnode)->n] = lptr;
return insert;
}
void display(struct node *ptr, int bl)
{
if (ptr)
{
int i;
for(i=1;i<=bl;i++)
printf(" ");
for (i=0; i < ptr->n; i++)
printf("%d ",ptr->keys[i]);
printf("\n");
for (i=0; i <= ptr->n; i++)
display(ptr->p[i], bl + 10);
}
}
int searchPos(int k, int *k_arr, int n)
{
int pos = 0;
while (pos < n && k > k_arr[pos])
pos++;
return pos;
}
void delete(int key)
{
struct node *uproot;
enum KeyStatus value;
value = del(root,key);
switch (value)
{
case SearchFailure:
printf("Key %d is not available\n",key);
break;
case LessKeys:
uproot = root;
root = root->p[0];
free(uproot);
break;
}
}
enum KeyStatus del(struct node *ptr, int k)
{
```

```
int pos, i, piv, n ,min;
int *k_arr;
enum KeyStatus  val;
struct node **p,*lptr,*rptr;
if (ptr == NULL)
return SearchFailure;
n = ptr -> n;
k_arr = ptr -> keys;
p = ptr->p;
min = (Macro - 1)/2;
pos = searchPos(k, k_arr, n);
if (p[0] == NULL)
{
if (pos == n || k < k_arr[pos])
return SearchFailure;
for (i=pos+1; i < n; i++)
{
k_arr[i-1] = k_arr[i];
p[i] = p[i+1];
}
return --ptr->n >= (ptr==root ? 1 : min) ? Success : LessKeys;
}
if (pos < n && k == k_arr[pos])
{
struct node *qp = p[pos], *qp1;
int nkey;
for(;;)
{
nkey = qp->n;
qp1 = qp->p[nkey];
if (qp1 == NULL)
break;
qp = qp1;
}
k_arr[pos] = qp -> keys[nkey-1];
qp -> keys[nkey - 1] = k;
}
val = del(p[pos], k);
if (val != LessKeys)
return val;
if (pos > 0 && p[pos-1]->n > min)
{
piv = pos - 1;
lptr =  p[piv];
rptr = p[pos];
rptr -> p[rptr->n + 1] = rptr->p[rptr->n];
```

```
for (i=rptr->n; i>0; i--)
{
rptr->keys[i] = rptr->keys[i-1];
rptr->p[i] = rptr->p[i-1];
}
rptr->n++;
rptr->keys[0] = k_arr[piv];
rptr->p[0] = lptr->p[lptr->n];
k_arr[piv] = lptr->keys[--lptr->n];
return Success;
}
if (pos > min)
{
piv = pos; lptr = p[piv];
rptr = p[piv+1];
lptr->keys[lptr->n] = k_arr[piv];
lptr->p[lptr->n + 1] = rptr->p[0];
k_arr[piv] = rptr->keys[0];
lptr->n++;
rptr->n--;
for (i=0; i < rptr->n; i++)
{
rptr->keys[i] = rptr->keys[i+1];
rptr->p[i] = rptr->p[i+1];
}
rptr->p[rptr->n] = rptr->p[rptr->n + 1];
return Success;
}
if(pos == n)
piv = pos-1;
else
piv   =   pos;
lptr = p[piv];
rptr = p[piv+1];
lptr->keys[lptr->n] = k_arr[piv];
lptr->p[lptr->n + 1] = rptr->p[0];
for (i=0; i < rptr->n; i++)
{
lptr->keys[lptr->n + 1 + i] = rptr->keys[i];
lptr->p[lptr->n + 2 + i] = rptr->p[i+1];
}
lptr->n = lptr->n + rptr->n +1;
free(rptr);
for (i=pos+1; i < n; i++)
{
k_arr[i-1] = k_arr[i];
```

```
p[i] = p[i+1];
}
return --ptr->n >= (ptr == root ? 1 : min) ? Success : LessKeys;
}
```

Output :

```
1.Insert
2.Delete
3.Display
4.Quit
Enter the option : 1

Enter the element : 23

1.Insert
2.Delete
3.Display
4.Quit
Enter the option : 1

Enter the element : 44

1.Insert
2.Delete
3.Display
4.Quit
Enter the option : 1

Enter the element : 78

1.Insert
2.Delete
3.Display
4.Quit
Enter the option : 1

Enter the element : 90

1.Insert
2.Delete
3.Display
4.Quit
Enter the option : 2

Enter the element : 44
```

1.Insert
2.Delete
3.Display
4.Quit
Enter the option : 3

B+ tree is :
23 78 90

1.Insert
2.Delete
3.Display
4.Quit
Enter the option : 4

EXPERIMENT-8

AIM: Construct tries for the implementation of English Dictionary and perform searching of a word in dictionary

Program:

Implementation of Tries

```
#include  <stdio.h>
#include <stdlib.h>
#include <string.h>
#define N 26

typedef struct TrieNode TrieNode;

struct TrieNode {
   char data;
   TrieNode* children[N];
   int is_leaf;
};

TrieNode* make_trienode(char data) {
   TrieNode* node = (TrieNode*) calloc (1, sizeof(TrieNode));
   for (int i=0; i<N; i++)
      node->children[i] = NULL;
   node->is_leaf = 0;
   node->data = data;
   return node;
}

void free_trienode(TrieNode* node) {
   for(int i=0; i<N; i++) {
      if (node->children[i] != NULL) {
         free_trienode(node->children[i]);
      }
      else {
         continue;
      }
   }
   free(node);
}

TrieNode* insert_trie(TrieNode* root, char* word) {
   TrieNode* temp = root;
```

```
    for (int i=0; word[i] != '\0'; i++) {
        int idx = (int) word[i] - 'a';
        if (temp->children[idx] == NULL) {
            temp->children[idx] = make_trienode(word[i]);
        }
        else {
        }
        temp = temp->children[idx];
    }
    temp->is_leaf = 1;
    return root;
}

int search_trie(TrieNode* root, char* word)
{
    TrieNode* temp = root;

    for(int i=0; word[i]!='\0'; i++)
    {
        int position = word[i] - 'a';
        if (temp->children[position] == NULL)
            return 0;
        temp = temp->children[position];
    }
    if (temp != NULL && temp->is_leaf == 1)
        return 1;
    return 0;
}

void print_trie(TrieNode* root) {
    if (!root)
        return;
    TrieNode* temp = root;
    printf("%c -> ", temp->data);
    for (int i=0; i<N; i++) {
        print_trie(temp->children[i]);
    }
}

void print_search(TrieNode* root, char* word) {
    printf("Searching for %s: ", word);
    if (search_trie(root, word) == 0)
        printf("Not Found\n");
    else
        printf("Found!\n");
}
```

```
int main() {
    TrieNode* root = make_trienode('\0');
    root = insert_trie(root, "hello");
    root = insert_trie(root,  "hi");
    root = insert_trie(root, "teabag");
    root = insert_trie(root, "teacan");
    print_search(root, "tea");
    print_search(root, "teabag");
    print_search(root, "teacan");
    print_search(root, "hi");
    print_search(root, "hey");
    print_search(root, "hello");
    print_trie(root);
    free_trienode(root);
    return 0;
}
```

Output :

```
Searching for tea: Not Found
Searching for teabag: Found!
Searching for teacan: Found!
Searching for hi: Found!
Searching for hey: Not Found
Searching for hello: Found!
 -> h -> e -> l -> l -> o -> i -> t -> e -> a -> b -> a -> g -> c -> a -> n ->
```