**Experiment 6**

**Aim: Implementation of Fibonacci and binary search.**

BINARY SEARCH

**Algorithm:**

1 : Find the middle element of array. using ,

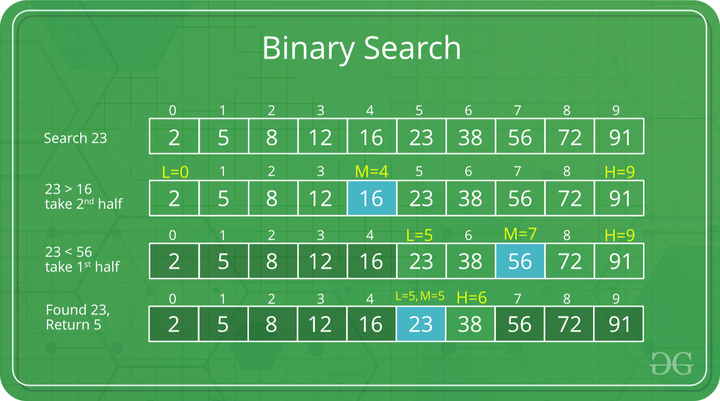
middle = initial\_value + end\_value / 2 ;

2 : If middle = element, return ‘element found’ and index.

3 : if middle > element, call the function with end\_value = middle - 1 .

4 : if middle < element, call the function with start\_value = middle + 1 .

5 : exit.

**Example:**

**Code:**

#include <stdio.h>

int main()

{

int search, mid;

int left = 0;

int right = 9;

printf("Binary Search (Shasvat Shah 60004220126)");

int arr[10] = {1,2,3,4,5,6,7,8,9,10};

printf("\nEnter Element to be searched through the array : ");

scanf("%d",&search);

while(left<=right){

mid = (left + right)/2;

if(arr[mid] == search){

printf("Element found at location : %d",mid);

break;

}

else if(arr[mid] > search){

right = mid -1;

}

else{

left = mid +1;

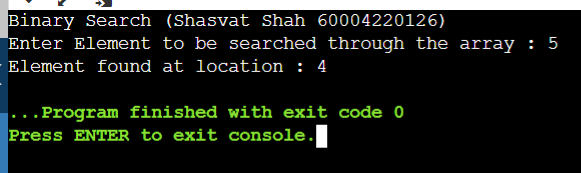
}

}

return 0;

}

**Output:**



Fibbonacci Search

**Algorithm:**

Let the searched element be x.

The idea is to first find the smallest Fibonacci number that is greater than or equal to the length of the given array. Let the found Fibonacci number be fib (m’th Fibonacci number). We use (m-2)’th Fibonacci number as the index (If it is a valid index). Let (m-2)’th Fibonacci Number be i, we compare arr[i] with x, if x is same, we return i. Else if x is greater, we recur for subarray after i, else we recur for subarray before i.

Below is the complete algorithm

Let arr[0..n-1] be the input array and the element to be searched be x.

1. Find the smallest Fibonacci Number greater than or equal to n. Let this number be fibM [m’th Fibonacci Number]. Let the two Fibonacci numbers preceding it be fibMm1 [(m-1)’th Fibonacci Number] and fibMm2 [(m-2)’th Fibonacci Number].

2. While the array has elements to be inspected:

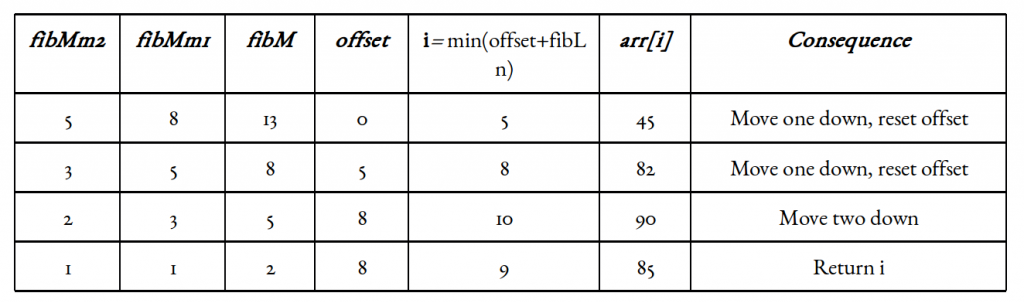
1. Compare x with the last element of the range covered by fibMm2

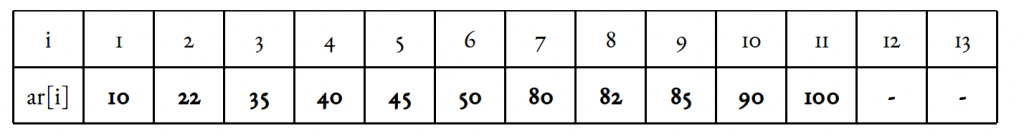
2. If x matches, return index

3. Else If x is less than the element, move the three Fibonacci variables two Fibonacci down, indicating elimination of approximately rear two-third of the remaining array.

4. Else x is greater than the element, move the three Fibonacci variables one Fibonacci down. Reset offset to index. Together these indicate the elimination of approximately front one-third of the remaining array.

3. Since there might be a single element remaining for comparison, check if fibMm1 is 1. If Yes, compare x with that remaining element. If match, return index.

**Example:**



**Code:**

// C program for Fibonacci Search

#include <stdio.h>

int min(int x, int y) {

return (x <= y) ? x : y;

}

int fibosearch(int arr[], int x, int n)

{

int fib2 = 0;

int fib1 = 1;

int fibk = fib2 + fib1;

while (fibk < n) {

fib2 = fib1;

fib1 = fibk;

fibk = fib2 + fib1;

}

int offset = -1;

while (fibk > 1) {

int i = min(offset + fib2, n - 1);

if (arr[i] < x) {

fibk = fib1;

fib1 = fib2;

fib2 = fibk - fib1;

offset = i;

}

else if (arr[i] > x) {

fibk = fib2;

fib1 = fib1 - fib2;

fib2 = fibk - fib1;

}

else

return i;

}

if (fib1 && arr[offset + 1] == x)

return offset + 1;

return -1;

}

int main(void)

{

int arr[]

= { 110, 22, 375, 42, 85};

int n = 5;

int x = 235;

int ind = fibosearch(arr, x, n);

if(ind>=0)

printf("Found at index: %d",ind);

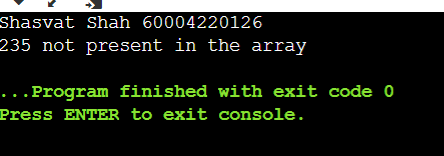
else

printf("%d not present in the array",x);

return 0;

}

**Ouput:**



**Conclusion** : Hereby, we have implemneted the searching techniques.

**Experiment 7**

Q1) Implementation of Insertion and Selection Sort.

Insertion Sort

**Algorithm**:

1. If the element is the first one, it is already sorted.
2. Move to next element
3. Compare the current element with all elements in the sorted array
4. If the element in the sorted array is smaller than the current element, iterate to the next element. Otherwise, shift all the greater element in the array by one position towards the right
5. Insert the value at the correct position
6. Repeat until the complete list is sorted

**Example:**

**Code:**

#include <stdio.h>

#include <conio.h>

void ins(int a[], int n)

{

int no, i, j;

for (i = 1; i < n; i++)

{

no = a[i];

for (j = i - 1; j >= 0; j--)

{

if (a[j] > no)

a[j + 1] = a[j];

else

break;

}

a[j + 1] = no;

}

}

void main()

{

int a[100], n, r;

clrscr();

printf("Enter the number of elements: \n");

scanf("%d", &n);

printf("Enter the elements: \n");

for (r = 0; r < n; r++)

scanf("%d", &a[r]);

ins(a, n);

printf("The sorted elements are as follows: \n");

for (r = 0; r < n; r++)

{

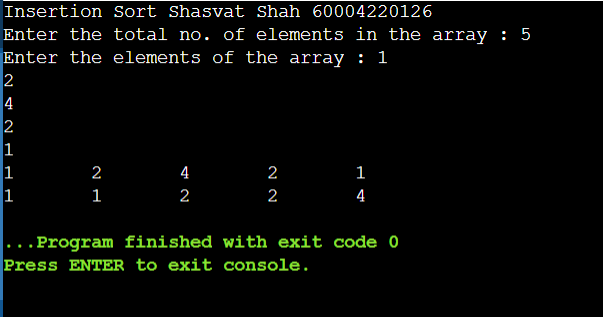
printf("%d\n", a[r]);

}

getch();

}

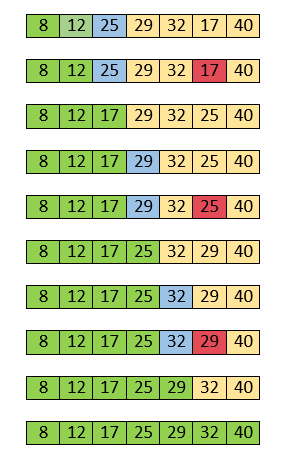
**Output:**



Selection Sort

**Algorithm:**

1. Set min to the first location
2. Search the minimum element in the array
3. swap the first location with the minimum value in the array
4. assign the second element as min.
5. Repeat the process until we get a sorted array.

**Example:**

**Code:**

#include <stdio.h>

#include <conio.h>

void select(int a[], int n)

{

int i, r, j, min, index;

for (i = 0; i < n - 1; i++)

{

min = a[i];

index = i;

for (j = i + 1; j < n; j++)

{

if (min > a[j])

{

index = j;

min = a[j];

}

}

a[index] = a[i];

a[i] = min;

}

printf("Sorted elements are as follows:\n");

for (r = 0; r < n; r++)

{

printf("%d\n", a[r]);

}

}

void main()

{

int a[100], n, r;

clrscr();

printf("Enter the number of elements: \n");

scanf("%d", &n);

printf("Enter the elements: \n");

for (r = 0; r < n; r++)

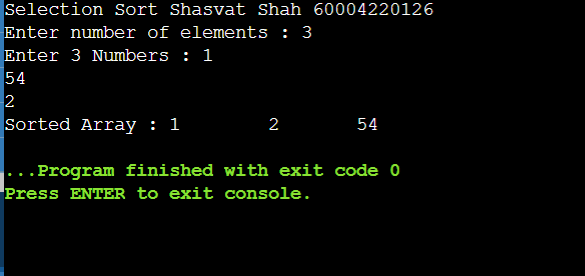
scanf("%d", &a[r]);

select(a, n);

getch();

}

**Output:**

****

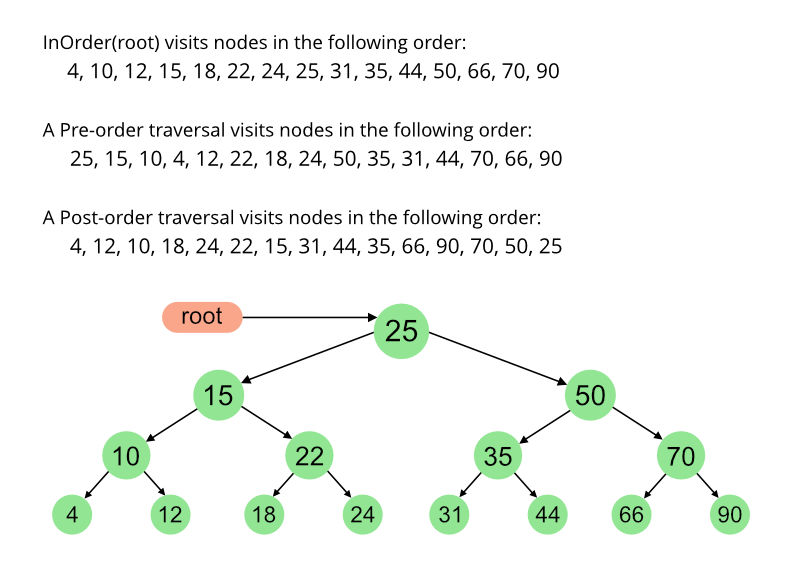
**Conclusion** : Hereby, we have implemneted the searching techniques.

**Experiment 8**

**Aim: Implementation of BST**

**Algorithm:**

1. If node == NULL
2. return createNode(data)
3. if (data < node->data)
4. node->left = insert(node->left, data);
5. else if (data > node->data)
6. node->right = insert(node->right, data);
7. return node;

**Example:**

**Code:**

#include <stdio.h>

#include <stdlib.h>

typedef struct node

{

int data;

struct node \*left, \*right;

} node;

void btree(int a[], int n);

node \*create\_tree(int a[], int n);

void inorder(struct node \*root);

void main()

{

int n, i;

int a[100];

printf("Enter number of nodes: \n");

scanf("%d", &n);

printf("Enter elements: \n");

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

{

scanf("%d", &a[i]);

}

btree(a, n);

}

void btree(int a[], int n)

{

node \*root;

root = create\_tree(a, n);

printf("\nInorder: ");

inorder(root);

printf("\nPostorder: ");

posorder(root);

printf("\nPreorder: ");

preorder(root);

}

node \*create\_tree(int a[], int n)

{

node \*p, \*prev, \*ptr, \*root;

int i, flag;

root = (node \*)malloc(sizeof(node));

root->data = a[0];

root->left = NULL;

root->right = NULL;

for (i = 1; i < n; i++)

{

ptr = (node \*)malloc(sizeof(node));

ptr->data = a[i];

ptr->left = NULL;

ptr->right = NULL;

p = root;

while (p != NULL)

{

prev = p;

if (a[i] < p->data)

{

p = p->left;

flag = 1;

}

else

{

p = p->right;

flag = 0;

}

}

if (flag == 1)

prev->left = ptr;

else

prev->right = ptr;

}

return (root);

}

void inorder(node \*root)

{

if (root != NULL)

{

inorder(root->left);

printf("%d\t", root->data);

inorder(root->right);

}

}

void preorder(node \*root)

{

if (root != NULL)

{

printf("%d\t", root->data);

preorder(root->left);

preorder(root->right);

}

}

void posorder(node \*root)

{

if (root != NULL)

{

posorder(root->left);

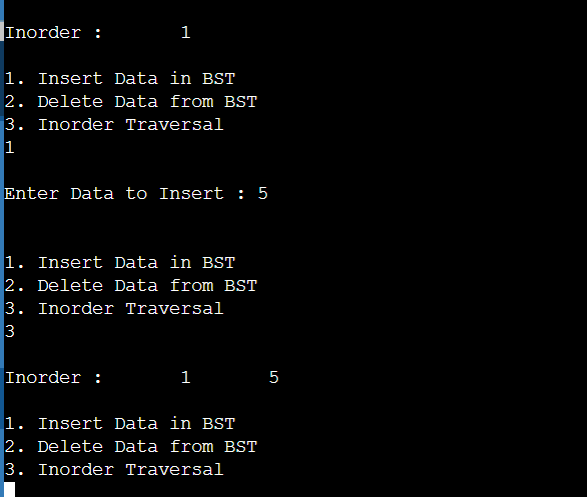
posorder(root->right);

printf("%d\t", root->data);

}

}

**Output:**

****

**Conclusion** : Hereby, we have implemneted the Binary Searching Technique.

**Experiment 9**

**Aim: Implementation of BFS DFS**

**BFS**

**Algorithm:**

A standard BFS implementation puts each vertex of the graph into one of two categories:

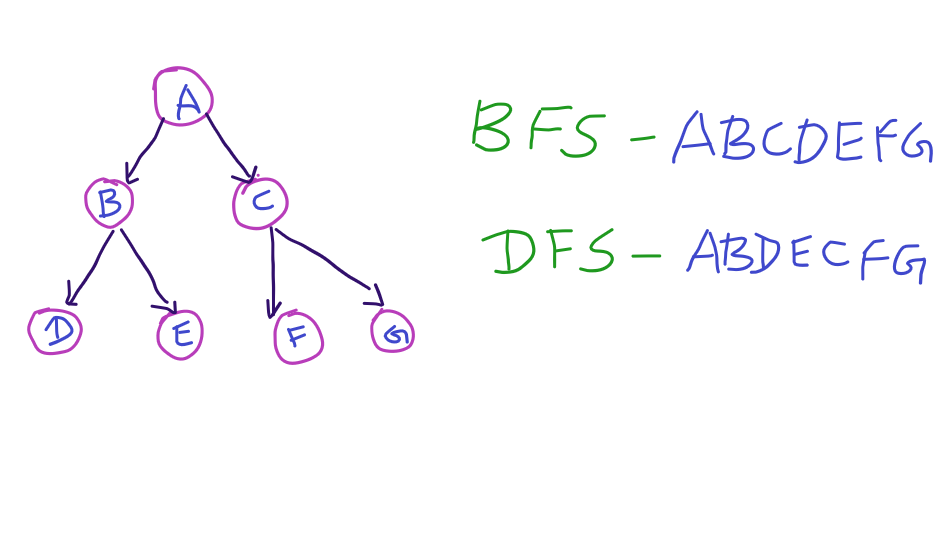
1.Visited

2.Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles The algorithm works as follows:

* 1.Start by putting any one of the graph's vertices at the back of a queue.
* 2.Take the front item of the queue and add it to the visited list.
* 3.Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.Keep repeating steps 2 and 3 until the queue is empty.
* The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

**Example:**



**Code:**

#include <stdio.h>

#include <stdlib.h>

#define SIZE 40

struct queue {

int items[SIZE];

int front;

int rear;

};

struct queue\* createQueue();

void enqueue(struct queue\* q, int);

int dequeue(struct queue\* q);

void display(struct queue\* q);

int isEmpty(struct queue\* q);

void printQueue(struct queue\* q);

struct node {

int vertex;

struct node\* next;

};

struct node\* createNode(int);

struct Graph {

int numVertices;

struct node\*\* adjLists;

int\* visited;

};

// BFS algorithm

void bfs(struct Graph\* graph, int startVertex) {

struct queue\* q = createQueue();

graph->visited[startVertex] = 1;

enqueue(q, startVertex);

while (!isEmpty(q)) {

printQueue(q);

int currentVertex = dequeue(q);

printf("Visited %d\n", currentVertex);

struct node\* temp = graph->adjLists[currentVertex];

while (temp) {

int adjVertex = temp->vertex;

if (graph->visited[adjVertex] == 0) {

graph->visited[adjVertex] = 1;

enqueue(q, adjVertex);

}

temp = temp->next;

}

}

}

// Creating a node

struct node\* createNode(int v) {

struct node\* newNode = malloc(sizeof(struct node));

newNode->vertex = v;

newNode->next = NULL;

return newNode;

}

// Creating a graph

struct Graph\* createGraph(int vertices) {

struct Graph\* graph = malloc(sizeof(struct Graph));

graph->numVertices = vertices;

graph->adjLists = malloc(vertices \* sizeof(struct node\*));

graph->visited = malloc(vertices \* sizeof(int));

int i;

for (i = 0; i < vertices; i++) {

graph->adjLists[i] = NULL;

graph->visited[i] = 0;

}

return graph;

}

// Add edge

void addEdge(struct Graph\* graph, int src, int dest) {

// Add edge from src to dest

struct node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src];

graph->adjLists[src] = newNode;

// Add edge from dest to src

newNode = createNode(src);

newNode->next = graph->adjLists[dest];

graph->adjLists[dest] = newNode;

}

// Create a queue

struct queue\* createQueue() {

struct queue\* q = malloc(sizeof(struct queue));

q->front = -1;

q->rear = -1;

return q;

}

// Check if the queue is empty

int isEmpty(struct queue\* q) {

if (q->rear == -1)

return 1;

else

return 0;

}

// Adding elements into queue

void enqueue(struct queue\* q, int value) {

if (q->rear == SIZE - 1)

printf("\nQueue is Full!!");

else {

if (q->front == -1)

q->front = 0;

q->rear++;

q->items[q->rear] = value;

}

}

// Removing elements from queue

int dequeue(struct queue\* q) {

int item;

if (isEmpty(q)) {

printf("Queue is empty");

item = -1;

} else {

item = q->items[q->front];

q->front++;

if (q->front > q->rear) {

printf("Resetting queue ");

q->front = q->rear = -1;

}

}

return item;

}

// Print the queue

void printQueue(struct queue\* q) {

int i = q->front;

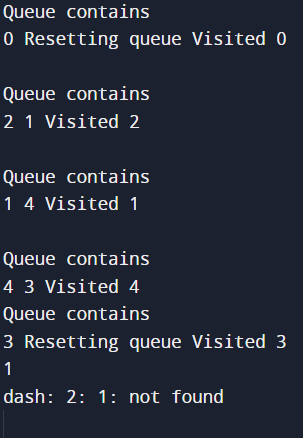
if (isEmpty(q)) {

printf("Queue is empty");

} else {

printf("\nQueue contains \n");

for (i = q->front; i < q->rear + 1; i++) {

**** printf("%d ", q->items[i]);

}

}

}

int main() {

struct Graph\* graph = createGraph(6);

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 2);

addEdge(graph, 1, 4);

addEdge(graph, 1, 3);

addEdge(graph, 2, 4);

addEdge(graph, 3, 4);

bfs(graph, 0);

return 0;

}

**Output:**

**DFS**

**Algorithm:**

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited

2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.

2. Take the top item of the stack and add it to the visited list.

3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.

4. Keep repeating steps 2 and 3 until the stack is empty.

**Code:**

#include <stdio.h>

#include <stdlib.h>

struct node {

int vertex;

struct node\* next;

};

struct node\* createNode(int v);

struct Graph {

int numVertices;

int\* visited;

// We need int\*\* to store a two dimensional array.

// Similary, we need struct node\*\* to store an array of Linked lists

struct node\*\* adjLists;

};

// DFS algo

void DFS(struct Graph\* graph, int vertex) {

struct node\* adjList = graph->adjLists[vertex];

struct node\* temp = adjList;

graph->visited[vertex] = 1;

printf("Visited %d \n", vertex);

while (temp != NULL) {

int connectedVertex = temp->vertex;

if (graph->visited[connectedVertex] == 0) {

DFS(graph, connectedVertex);

}

temp = temp->next;

}

}

// Create a node

struct node\* createNode(int v) {

struct node\* newNode = malloc(sizeof(struct node));

newNode->vertex = v;

newNode->next = NULL;

return newNode;

}

// Create graph

struct Graph\* createGraph(int vertices) {

struct Graph\* graph = malloc(sizeof(struct Graph));

graph->numVertices = vertices;

graph->adjLists = malloc(vertices \* sizeof(struct node\*));

graph->visited = malloc(vertices \* sizeof(int));

int i;

for (i = 0; i < vertices; i++) {

graph->adjLists[i] = NULL;

graph->visited[i] = 0;

}

return graph;

}

// Add edge

void addEdge(struct Graph\* graph, int src, int dest) {

// Add edge from src to dest

struct node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src];

graph->adjLists[src] = newNode;

// Add edge from dest to src

newNode = createNode(src);

newNode->next = graph->adjLists[dest];

graph->adjLists[dest] = newNode;

}

// Print the graph

void printGraph(struct Graph\* graph) {

int v;

for (v = 0; v < graph->numVertices; v++) {

struct node\* temp = graph->adjLists[v];

printf("\n Adjacency list of vertex %d\n ", v);

while (temp) {

printf("%d -> ", temp->vertex);

temp = temp->next;

}

printf("\n");

}

}

int main() {

struct Graph\* graph = createGraph(4);

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

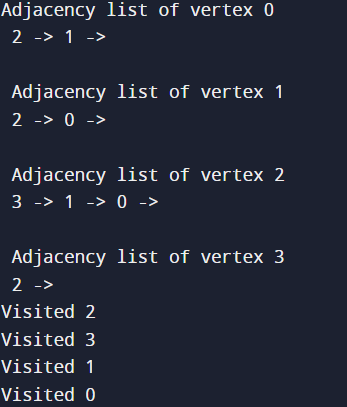
addEdge(graph, 1, 2);

addEdge(graph, 2, 3);

printGraph(graph);

DFS(graph, 2);

return 0;}

****

**Output:**

**Conclusion** : Hereby, we have implemneted the BFS & DFS.

**Experiment 10**

**AIM: IMPLEMENTATION OF HASHING FUNCTIONS WITH LINEAR PROBING COLLISION RESOLUTION TECHNIQUES**

**Algorithm to insert a value in linear probing**

Hashtable is an array of size = TABLE\_SIZE

Step 1: Read the value to be inserted, key

Step 2: let i = 0

Step 3: hkey = key% TABLE\_SIZE

Step 4 :compute the index at which the key has to be inserted in hash table

index = (hkey + i) % TABLE\_SIZE

Step 5: if there is no element at that index then insert the value at index and STOP

Step 6: If there is already an element at that index

step 4.1: i = i+1

step 7: if i < TABLE\_SIZE then go to step 4

**Algorithm to search a value in linear probing**

Hashtable is an array of size = TABLE\_SIZE

Step 1: Read the value to be searched, key

Step 2: let i = 0

Step 3: hkey = key% **TABLE\_SIZE**

Step 4: compute the index at which the key can be found

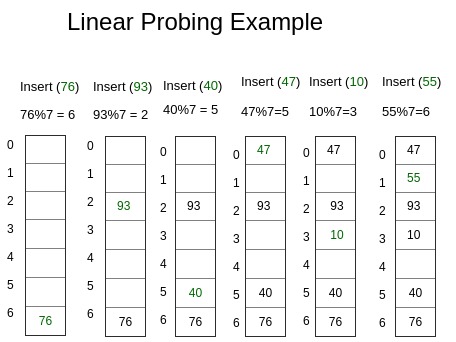
**index = (hkey+ i) % TABLE\_SIZE**

Step 5: if the element at that index is same as the search value then print element found and STOP

Step 6: else

step 4.1: i = i+1

step 7: if i < TABLE\_SIZE then go to step 4



**Example:**

**Code:**

#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();

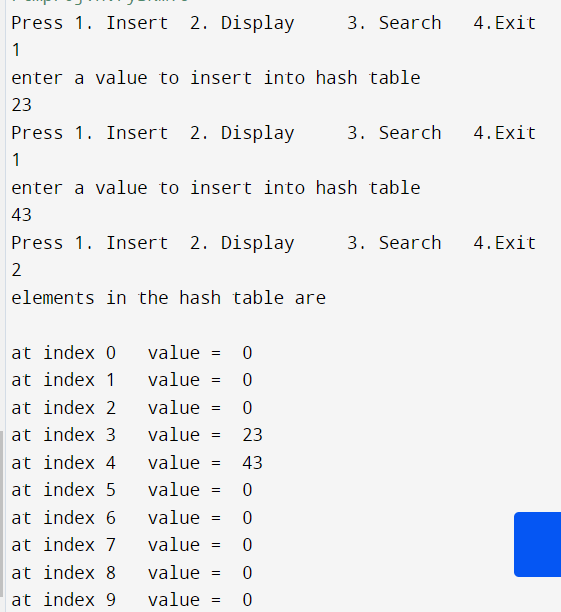
break;

case 4:exit(0);

}

}

}

**Output:**

**Conclusion** : Hereby, we have implemneted the Hashing techniques.