**Name:Vikas Mane**

**Roll No:B1921152**

**Class: BE-B**

Assignment 12

## **Title:** Design and implement parallel algorithm utilizing all resources available. For Binary Search for Sorted Array

Depth-First Search ( tree or an undirected graph ) OR Breadth-First Search (tree or an undirected graph) OR

Best-First Search that ( traversal of graph to reach a target in the shortest possible path)

**System Requirements:** C++ / Java

# Theory:

## In computer science, binary search, also known as half-interval search,logarithmic search,or binary chop,is a search algorithm that finds the position of a target value within a sorted array. Binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array. Even though the idea is simple, implementing binary search correctly requires attention to some subtleties about its exit conditions and midpoint calculation.

Binary search runs in logarithmic time in the worst case, making *O*(log *n*) comparisons,. Binary search is faster than linear search except for small arrays. However, the array must be sorted first to be able to apply binary search.

There are specialized data structures designed for fast searching, such as hash tables, that can be searched more efficiently than binary search. However, binary search can be used to solve a wider range of problems, such as finding the next-smallest or next-largest element in the array relative to the target even if it is absent from the array.

There are numerous variations of binary search. In particular, fractional cascading speeds up binary searches for the same value in multiple arrays. Fractional cascading efficiently solves a number of search problems in computational geometry and in numerous other fields. Exponential search extends binary search to unbounded lists. The binary search tree and B-tree data structures are based on binary search.

## **Objectiv**e**:** To study and implementation of parallel searching techniques.

**Input:** Sorted Array Numbers

## **Output:** The key found at respective position/ Not found message

**Result / Conclusion:**

Hence, after the execution of the assignment, parallel searching is being implemented successfully

**Program for Binary Search:**

#include<iostream> #include<stdlib.h> #include<omp.h> using namespace std;

int binary(int \*, int, int, int);

int binary(int \*a, int low, int high, int key)

{

int mid; mid=(low+high)/2;

int low1,low2,high1,high2,mid1,mid2,found=0,loc=-1;

#pragma omp parallel sections

{

#pragma omp section

{

low1=low; high1=mid; while(low1<=high1)

{

if(!(key>=a[low1] && key<=a[high1]))

{

low1=low1+high1; continue;

}

mid1=(low1+high1)/2; if(key==a[mid1])

{

found=1; loc=mid1; low1=high1+1;

}

else if(key>a[mid1])

{

low1=mid1+1;

}

else if(key<a[mid1]) high1=mid1-1;

}

}

#pragma omp section

{

low2=mid+1;

high2=high;

while(low2<=high2)

{

if(!(key>=a[low2] && key<=a[high2]))

{

low2=low2+high2; continue;

}

mid2=(low2+high2)/2; if(key==a[mid2])

{

found=1; loc=mid2; low2=high2+1;

}

else if(key>a[mid2])

{

low2=mid2+1;

}

else if(key<a[mid2]) high2=mid2-1;

}

}

}

return loc;

}

int main()

{

int \*a,i,n,key,loc=-1;

cout<<"\n Enter Total No of Elements=> "; cin>>n;

a=new int[n];

cout<<"\n Enter Elements=> "; for(i=0;i<n;i++)

cin>>a[i];

cout<<"\n Enter Key to Find=>"; cin>>key;

loc=binary(a,0,n-1,key);

if(loc==-1)

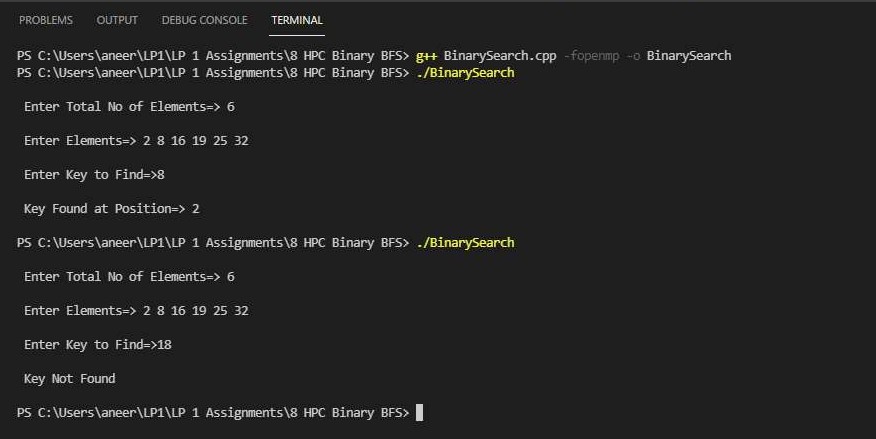
cout<<"\n Key Not Found\n\n"; else

cout<<"\n Key Found at Position=> "<<loc+1<<"\n\n";

return 0;

}

# Output for Binary Search:



**Program for BFS:**

#include<iostream> #include<stdlib.h> #include<queue> #include<omp.h> using namespace std;

class node

{

public:

node \*left, \*right; int data;

};

class Breadthfs

{

public:

node \*insert(node \*, int); void bfs(node \*);

};

node \*insert(node \*root, int data)

{

if(!root)

{

root=new node; root->left=NULL; root->right=NULL; root->data=data; return root;

}

queue<node \*> q; q.push(root); while(!q.empty())

{

node \*temp=q.front(); q.pop();

if(temp->left==NULL)

{

}

else

temp->left=new node; temp->left->left=NULL; temp->left->right=NULL; temp->left->data=data; return root;

q.push(temp->left);

if(temp->right==NULL)

{

}

else

{

temp->right=new node; temp->right->left=NULL; temp->right->right=NULL; temp->right->data=data; return root;

q.push(temp->right);

}

}

return NULL;

}

void bfs(node \*head)

{

queue<node\*> q; q.push(head); int qSize;

while (!q.empty())

{

qSize = q.size(); #pragma omp parallel for

for (int i = 0; i < qSize; i++)

{

node\* currNode; #pragma omp critical

{

currNode = q.front(); q.pop(); cout<<"\t"<<currNode->data;

}

#pragma omp critical

{

if(currNode->left)

q.push(currNode->left); if(currNode->right)

q.push(currNode->right);

}

}

}

}

int main(){

node \*root=NULL; int data;

char ans; do

{

cout<<"\n Enter Data=> "; cin>>data; root=insert(root,data);

cout<<"\n\t\t Press 'Y' to insert one more node: "; cin>>ans;

}while(ans=='y'||ans=='Y'); cout<<"\n";

cout<<"\n\t\t BFS Result: "; bfs(root);

cout<<"\n\n"; return 0;

}

# Output for BFS:

