**ASSIGNMENT NO: 1 DATE:28.11.2023**

**Problem statement:**

Consider a sequence S of n where n ≥ 7 and implement Heap sort on S.

**Theory:**

Heap sort is one of the most efficient in-place, unstable sorting algorithm that requires only O(nlogn) operations regardless of the order of the input to sort the sequence. The main idea behind Heap sort is to maintain a Max heap (Min heap in case of descending sorting order) of size n as an almost complete binary tree of n nodes such that the content of each node is less than or equal to the content of its parent node. Now that the root has the maximum value, we have to remove it from the heap. Again, we have to build the max heap and remove the root element and repeat the procedure until the whole sequence is sorted in ascending order.

**Pseudo code:**

The pseudo code for heap sort is as follows:

Heapsort(a,n)

{ // a[1:n] is the input sequence containing n elements to be sorted using Heap sort

Heapify(a,n)

for i=n to 2

{ // exchanging the value of the root element with the last element

t=a[i]

a[1]=t

Adjust(a,1,i-1) // Adjusting the tree to maintain the max heap property

}

}

The pseudo code for Heapify function is as follows:

Heapify(a,n)

{ // Readjust the elements in a[1:n] to form a heap

for i=n/2 to 1

{

Adjust(a,i,n)

}

}

The pseudo code for Adjust functions is as follows:

Adjust(a,i,n)

{ // The complete binary trees with roots 2i and 2i+1 are combined with node i

// to form a heap rooted at i

// No node has an address greater than n or less than 1

j=2i

item=a[i]

while(j≤n)

{

if((j<n) and (a[j]<a[j+1]))

{

j=j+1 // Compare left and right child and let j be the bigger child

}

if(item≥a[j])

{

Break from the loop // A position for item is found

}

a[j/2] =a[j]

j=2j

}

a[j/2]= item

}

**Assumptions:**

The following assumption have been made during this assignment

* We are assuming that the given input sequence is of integer type.
* We are assuming that the given input sequence contains only positive integer.

**Analysis:**

In this approach inside the Heapsort function, we have two more functions Heapify to build the max heap and adjust to adjust the heap after root deletion. Though the call of Heapify requires only O(n) operations, adjust possibly requires O(logn) operations for each invocation as the worst-case run time of Adjust is also proportional to the height of the tree. Therefore, if there are n elements in a heap, deleting the maximum can be done in O(logn) time. So for n elements it can be done in O(nlogn) time, hence the time complexity for Heap sort is in O(nlogn) .

**Dry run:**

Let us consider a sequence S of 7 elements as follows: [56 12 88 15 66 23 37]

Now initially we have to build the max heap so that the root node will have the highest value among all the nodes.

After making the max heap we have the sequence as: [88 66 56 37 12 23 15]

Now to have the sequence in sorted order we have to replace the root node with the last element and have to continue this procedure with all the nodes until we have the desired sequence. As of now we have the root node 88 and last node 15 so we will substitute it after substituting we have the sequence as: [15 66 56 37 12 23 88].

Now we have to adjust the remaining sequence i.e. [15 66 56 37 12 23]. After adjusting the sequence will become [15 56 23 37 12 66 88].

In the next iteration the sequence will be [12 37 23 15 56 66 88]. And in next consecutive iterations we sequence will update as follows:

[ 15 23 12 37 56 66 88]

[ 12 15 23 37 56 66 88]

[12 15 23 37 56 66 88]

Now after all iterations we finally have the sorted sequence [12 15 23 37 56 66 88].

**Code:**

#include <stdio.h>

#include <stdlib.h>

void heapsort(int \*, int); // Funtion prototype for Heap sort

void heapify(int \*, int); // Function prototype for heapify

void adjust(int \*, int, int); // Function prototype for adjust which will adjust the heap

int main()

{

int \*a, n;

d o

{

printf("How many elements to be inserted?: ");

scanf("%d", &n);

if (n <= 0)

{

printf("\nEnter a valid number!!!..\n");

}

} while (n <= 0);

// Dynamic memory allocation for input array

a = (int \*)calloc(n, sizeof(int));

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

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

{

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

}

heapsort(a, n- 1); // Calling the function for heap sort

printf("\n\narray after heapsort:\n\n");

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

{ // printing the sorted array

printf("\t%d", a[i]);

}

printf("\n");

free(a); // Free the input array

return 0;

}

void heapsort(int \*a, int n)

{ // Function definition for heapsort

heapify(a, n); // Calling the function to build the initial max heap

for (int i = n; i >= 1; i--)

{ // placing the root value in it’s appropriate position

int t = a[i];

a[i] = a[0];

a[0] = t;

adjust(a, 0, i- 1); // Calling the function to adjust the heap

}

}

void heapify(int \*a, int n)

{ // Function definition to build the heap

for (int i = n / 2; i >= 0; i--)

{

adjust(a, i, n);

}

}

void adjust(int \*a, int i, int n)

{ // Function definition to adjust the heap

int j = 2 \* i;

int item = a[i];

while (j <= n)

{

if ((j < n) && (a[j] < a[j + 1]))

{ // Compare left and right child and let j be the bigger child

j = j + 1;

}

if (item >= a[j])

{

break; // A position for item is found

}

a[j / 2] = a[j];

j = 2 \* j;

}

a[j / 2] = item;

}

**Output:**

How many elements to be inserted?: 7

Enter the elements: 56 12 88 15 66 23 37

array after heapsort:

12 15 23 37 56 66 88

How many elements to be inserted?: 10

Enter the elements: 25 98 47 19 87 23 66 12 50 33

array after heapsort:

12 19 23 25 33 47 50 66 87 98

**Discussion:**

* Heap sort has a time complexity of O(nlogn) where n is the number of elements to be sorted for all the cases i.e. best, worst, average case.
* Heap sort involves building and maintaining heap which makes this algorithm very costly.
* Heap sort is an unstable algorithm which means it can change the relative order of equal terms.

**ASSIGNMENT NO: 2 DATE:12.12.2023**

**Problem statement:**

Consider a graph G(V,E) use breadth first search (BFS) algorithm to check whether the given graph is connected or not also find the number of components in the graph using the said algorithm.

**Theory:**

A graph G(V,E) is said to be undirected if the graph G is defined abstractly as an ordered pair (V,E) where V is a set and E is a set of multisets of two elements from V. An undirected graph can be represented geometrically as a set of marked points V with a set of lines E between the points.

A graph G(V,E) is said to be unweighted if it’s edges don’t have any weight associated with it.

A graph G(V,E) is said to be connected if for every pair of distinct vertices vi,vj in G there is a path and if not then the graph G is said to be disconnected.

We will solve the given problem using Breadth First Search (BFS) algorithm.

BFS algorithm is one of the simplest algorithm for searching a graph and archetype for many important graph algorithms. Given a graph G(V,E) and a source vertex s breadth first search systematically explores the edges of G to discover every vertex that is reachable from s. It computes the distance (smallest number of edges) from s to each reachable vertex. In this algorithm we use queue data structure to keep track of next visiting vertex and while we deque a vertex from the queue we increase a count variable by 1, after all the vertices are explored if the value of count is equal to the value of total number of vertices, then we can say that our graph is connected else our graph is disconnected.

Now to find the number of components in the graph we will take the help of the distance array used in BFS which is keeping track of distances from the source vertex to other vertices. This array is initialized with -1 which indicates that all the vertices are unvisited now after the first BFS call we will check whether the array contains -1 or not if it contains -1 then it means that vertex or vertices are unvisited hence the number of component is increased by 1 then we will call the BFS function once again but this time with an unvisited source node after that we will check the distance array once again and if there are no -1 we will conclude that all the components and vertices are discovered.

**Pseudo code:**

The algorithm for BFS(Breadth first search) is as follows:

Algo\_BFS(G(V,E),s)

{ // G(V,E) is the connected graph given as input where V is the set of vertices and E is

// set of edges.

// s is the source vertex

n =|V|

for(i=1 to n)

{ // initializing the distance array d and parent array p to -1

d[i]= -1

parent[i]= -1

}

d[s]= 0 // initializing the source index of distance array to 0

Q 🡨 an empty queue

count 🡨 0

enqueue(Q,s) // inserting the source to queue

While(Not empty (Q))

{

v 🡨dequeue(Q) // dequeue a element from queue and storing it in v

count= count+1

for all u neighbour(v) // finding neighbours of v and storing in u

{

if( d[u] <0 )

{

d[u] = d[v] +1

parent[u]= v

enqueue(Q,u)

}

}

}

if(count == n)

{

print graph is connected

else

print graph is disconnected

}

}

**Assumptions:**

The following assumptions have been made during this assignment

* We are assuming that the given input sequence is of integer type.
* We are assuming that we will only call the BFS function until all the components are detected. Once all the components are detected we will not call BFS function any further.

**Analysis:**

The main idea behind BFS algorithm is to start with a vertex if it is unvisited mark it as visited then put all it’s adjacent vertices into a queue. We have to repeat this step until the queue is empty. Now this algorithm uses queue data structure to store the adjacent vertices. As queue follows first in first out methodology it helps BFS for the breadth wise traversing hence the Breadth First Search. So now the time complexity for operations on queue is in O(1) , and we have to do that for all the vertices so the total time complexity for queue operations is in O(V). We also have to find the adjacent neighbours by traversing through the graph, time complexity for this task is also is in O(V) so the time complexity for the BFS algorithm is in O(V2) given that we are taking the input graph in adjacency matrix format.

However if we use adjacency list format instead of matrix then the time complexity will be in O(V + E) because the time complexity for queue operation will remain the same but the time complexity for finding the neighbours will be in O(E) as now instead of scanning through the matrix now we are scanning the adjacency list. In this assignment we are using adjacency matrix representation.

**Dry run:**

Consider the following disconnected graph with 2 components.

Now we will begin the procedure with source vertex 1. Now initially all the parent array contains only -1. But as we begin our procedure the distance array cell for vertex 1 will be set to 0 then as the procedure advances neighbour vertices 2 and 3 will get into the queue and their respective distance array cell will be updated according to our algorithm. As vertices 4 and 5 are part of another component they will be not marked as visited in the first call as we have chosen vertex 1 as source node which belongs from another component. So, after the first call the distance array will be: [0 1 1 -1 -1].

Now we will choose vertex number 4 as the new source vertex which will make the vertex 5 visited. So now after the second call the distance array will be: [0 1 1 0 1]. Now we don not have any -1 in the distance array hence we can conclude that all vertices have been visited and as it requires 2 BFS calls for visiting all the vertices we can say that the number of components in the given input graph is 2.

**Code:**

#include <stdlib.h>

#include <stdio.h>

int bfs(int \*\*, int, int, int \*); // Function prototype for bfs

int main()

{

    int \*\*a, n, s, d, i, j, bfs\_counter = 0;

    do

    { // Taking input for the graph in adjacency matrix format

  printf("enter the no of vertices in the graph: ");

        scanf("%d", &n);

    } while (n <= 0);

    a = (int \*\*)calloc(n, sizeof(int \*)); // Dynamically allocating memory

    int \*dist = (int \*)calloc(n, sizeof(int));

    for (i = 0; i < n; i++)

    {

        a[i] = (int \*)calloc(n, sizeof(int));

        dist[i] = -1;

    }

    printf("\nenter the edges:\n");

    do

    {

        printf("\n\tenter the vertex pair for which there is an edge: ");

        scanf("%d%d", &s, &d);

        a[s - 1][d - 1] = 1;

        a[d - 1][s - 1] = 1;

        printf("\n do  you want to enter more no edges:[0/1] ");

        scanf("%d", &s);

    } while (s != 0);

    printf("\nthe input graph is:\n");

    for (i = 0; i < n; i++)

    {

        printf("\n");

        for (j = 0; j < n; j++)

        {

            printf("%d", a[i][j]);

            printf("\t");

        }

    }

    printf("\nplease enter the source node: ");

    scanf("%d", &s);

    s = s - 1;

    int count = bfs(a, n, s, dist); // calling the bfs function

    bfs\_counter++;

    if (count == n)

    {

        printf("\n graph is connected, there is only 1 component in the given graph\n");

        printf("\nthe distance array:\t");

        for (i = 0; i < n; i++)

        {

            printf("\t%d\t", dist[i]);

        }

        printf("\n");

    }

    else

    {

        printf("\n graph is not connected, there is more than 1 component in the given graph\n");

        printf("\nthe distance array:\t");

        for (i = 0; i < n; i++)

        {

            printf("\t%d\t", dist[i]);

        }

        printf("\nDo you want to apply bfs again?[0/1]:\t");

        scanf("%d", &d);

        while (d != 0)

        {

            printf("\nEnter the source vertex:\t");

            scanf("%d", &s);

            bfs(a, n, s - 1, dist);

            bfs\_counter++;

            printf("\nthe distance array:\t");

            for (i = 0; i < n; i++)

            {

                printf("\t%d\t", dist[i]);

            }

            printf("\nDo you want to apply bfs again?[0/1]:\t");

            scanf("%d", &d);

        }

    }

    printf("\nNumber of components detected: %d\n", bfs\_counter++);

    free(a);

    free(dist);

    return 0;

}

// Function definition for bfs

int bfs(int \*\*a, int n, int s, int \*d)

{

    int \*parent = (int \*)calloc(n, sizeof(int));

    int \*q = (int \*)calloc(n, sizeof(int));

    int i, count = 0, k, u, v, front = -1, rear = -1;

    for (i = 0; i < n; i++)

    {

        parent[i] = -1;

    }

    d[s] = 0; // set the value to 0 for source vertex in distance array

    q[++rear] = s; // enqueue the source vertex

    while (front != rear)

    {

        v = q[++front]; // dequeue a vertex from the queue

        count++;

        for (u = 0; u < n; u++) // scanning through the vertices

        {

            if (a[v][u] == 1) // finding the adjacent vertex

            {

                if (d[u] < 0) // if vertex is unvisited mark it visited and // update the distance array

                {

                    d[u] = d[v] + 1;

                    parent[u] = v;

                    q[++rear] = u;

                }

            }

        }

    }

    free(q);

    free(parent);

    return count;

}

**Output:**

enter the no of vertices in the graph: 5

enter the edges:

enter the vertex pair for which there is an edge: 1 2

do you want to enter more no edges:[0/1] 1

enter the vertex pair for which there is an edge: 2 3

do you want to enter more no edges:[0/1] 1

enter the vertex pair for which there is an edge: 1 3

do you want to enter more no edges:[0/1] 1

enter the vertex pair for which there is an edge: 4 5

do you want to enter more no edges:[0/1] 0

the input graph is:

0 1 1 0 0

1 0 1 0 0

1 1 0 0 0

0 0 0 0 1

0 0 0 1 0

please enter the source node: 1

graph is not connected, there is more than 1 component in the given graph

the distance array: 0 1 1 -1 -1

Do you want to apply bfs again?[0/1]: 1

Enter the source vertex: 4

the distance array: 0 1 1 0 1

Do you want to apply bfs again?[0/1]: 0

Number of components detected: 2

**Discussion:**

* Time complexity for BFS depends on the graph input format for adjacency matrix it is in O(V2) and for adjacency list it is O(V+E)
* As the name suggests BFS traverses the graph level wise hence it visited the neighbours first which are in the same level.