Theoritical Assignment 3

Sahil Dhull

160607

Ques 1:

Given: Binary Min Heap H of size n, number x, +ve integer k
Logic: Initialize a counter=0, traverse the heap staring from top, check if the node has value less than x. If yes increment counter by 1 and check its right and left subtree by calling the function. If no, then simply return.
So basically traverse the heap depth wise till 1st element ¿ x, then go another branch and traverse it depth wise and so on.
But there must be another check, otherwise it will take more time. So inside the if condition, we must check if counter has already reached the value k, because that means we have already found k elements smaller than x (not necessarily k smallest elements, but this ensures that the kth smallest element will also be ¡ x), so no need to search anymore. Also after traversing left branch of any subtree, we must again check if counter has already reached k, as in that case, we don't need to traverse the right subtree (because we have found k smaller elements implying kth smallest element is smaller than x).
Algorithm:

```
check (node){
        if (value{node} < x) {
                //increment counter
                if (counter >= k)
                        return;
                check (left_subtree{node});
                if (counter >= k)
                        return;
                check (right_subtree{node});
        }
}

main{
        counter=0;
        check (H); // H is the heap
        if (counter >= k)
                print "x is not smaller than kth smallest element"
        // in this case more than k elements are smaller than x, so kth
        // smallest element is smaller than x
        else
                print "x is smaller than kth smallest element"
        // in this case the algorithm is not able to find the k nodes
        // whose value is smaller than x, i.e. kth smallest element must
        // be greater than x
}
```

Time Complexity:
The function will be called k times and each execution takes O(1) or constant time.
So the time complexity is O(k).

Ques 2:

1. To find if the graph is acyclic or not, the DFS (Depth First Search) is altered in a way that whenever a visited node belongs to adjacency list of some other node (other than from which it was accessed), cycle exists, so exit the loop. For this we create a flag and set its initial value to zero. whenever the av=bove condition arises, set its value=1 and then break the loop.
   Algorithm:

```
//initialization
index=0;
count=1//to keep track of different components
flag=0;//used to identify if cycle exists or not
for all v in V
        mark[v]=unvisited;
for all v in V{
        if(flag==1) break;
        if (mark[v]==unvisited){
                DFS(G,v);
                count++; // component number
        }
}
DFS(G,v){
        mark[v]=visited;
        for all w in ADJ(v){
                if(mark[w]==visited){
                        flag=1;
                        break;
                }
                parent[w]=v;
                DFS(G,w);
                if(flag==1) break;
        }
}
if(flag==1) "graph is not acyclic";
else "graph is cyclic";
```

   Time Complexity:
   Time complexity is O(|V|) since maximum n edges can be explored without setting flag to 0. In maximum case it is possible to traverse all |V|without flag=1, but as soon as the all vertices are traversed either the loop ends or flag becomes 1, leading to end of running.

2. For a graph to be tree, it must not contain any cycle and it must be connected.
   So along with keeping check on flag, this time both flag and count will be checked, as soon as flag becomes 1 or count becomes 2, graph does not remain a tree.
   Algorithm:

```
//initialization
index=0;
count=1//to keep track of different components
flag=0;//used to identify if cycle exists or not
for all v in V
        mark[v]=unvisited;
for all v in V{
        if(flag==1 || count==2) break;
        if (mark[v]==unvisited){
                DFS(G,v);
```

```
                count++; // component number
        }
}
DFS(G,v){
        mark[v]=visited;
        for all w in ADJ(v){
                if(mark[w]==visited){
                        flag=1;
                        break;
                }
                parent[w]=v;
                DFS(G,w);
                if(flag==1||count==2) break;
        }
}
if(flag==1||count>=2) "graph is not tree";
else "graph is a tree";
```

Time Complexity:
In this case also, time complexity is O(|V|). Here also in maximum case it is possible to traverse all |V|without flag=1, but as soon as the all vertices are traversed either the loop ends or flag becomes 1 or count becomes 2, leading to end of running. Moreover as soon as any component is finished, loop is exited.

Ques 3:

Given: A dictionary of words D, and task is to reach from A to B in minimum allowed edits.
Reasoning: The problem is similar to finding the shortest path, where nodes of the graph are words in dictionary and for any X and Y, if X and Y can be reached from each other using allowed edits, then there exists edge between the node X and Y in the graph.
So first we need to form a graph from the words in dictionary, then search for the shortest path from A to B (corresponds to printing path with minimum number of allowed edits). For shortest path, BFS (Breadth First Search) Algorithm can be modified to capture the distances between the nodes.
Algorithm:

1. Form the graph G from words in the dictionary.
   d= |D|

```
//Dictionary D stored in array D
for (i from 1 to d){
        for (j from 1 to d and not equal to i){
                //compare the strings D[i] and D[j]
                //Form edge between them if they lie
                //within range of allowed edits
        }
}
```

2. Now that the graph is formed. Need to find the shortest path from A to B.
   If A and B belong to different components, then algorithm will return "Not possible"
   V denotes the set of nodes in teh graph G, and v is any node among them.

```
Q=NULL; // define empty queue Q
for all v in V except A
        distance[v]=infinity; // initial distance of all nodes=infinity
```

```
distance[A]=0;
for all v in V
        mark[v]=unvisited;
enqueue(Q,A); // A is the starting vertes, so enqueued to Q
while(!isEmpty(Q)){ //iterate till Q becomes empty
        v=dequeue(Q); // store the dequeued node in v
        mark[v]=visited;
        for all w in ADF(v){ // ADJ(v) denotes Adjacency list of v
                if(mark[w]==unvisited){
                        mark[w]=visited;
                        distance[w]=distance[v]+1;
                        parent[w]=v; // using this, the path can be printed
                        enqueue(Q,w);
                }
        }
}
//define array arr to store the path
i=distance[B];
v=B;
if(distance[B]==infintiy)
        print "Not possible";
else{
        do{
                arr[i]=v;
                i--;
                v=parent[v];
        }while(i>-1);
        for i from 0 to distance[B]
                print arr[i]; // This prints one of shortest paths from A to B
}
```

Time Complexity Analysis:
There are 2 steps to the algorithm: first is forming of the graph and second is finding and printing of shortest path.
The first step involves 2 loops, each one iterating d times. Inside the loops, time is O(c) for comparisions and assignments among max c length words. So the total time for graph formation is $O(c \cdot d^2)$. The second step takes time that is taken for BFS plus some time for assigning the parents in array (which is much smaller as compared to time for BFS). Now since we are using adjacency list for BFS, time will be O(V+E) (because in max case either all nodes are traversed or all edges are traversed).
V=d and E<$d^2/2$ implies time for 2nd step is $O(d^2)$. So total time will be $O(c \cdot d^2)$.

Worst Case Time Complexity:
In the worst case also, time complexity remains the same because the time taken for forming the graph remains the same (beacuse both loops will have to iterate d times and check for c characters in the string). So the total time for graph formation is $O(c \cdot d^2)$.
In the second step, BFS in max case will take $O(d^2)$ time.
So worst case time complexity remains $O(c \cdot d^2)$.

Ques 4:

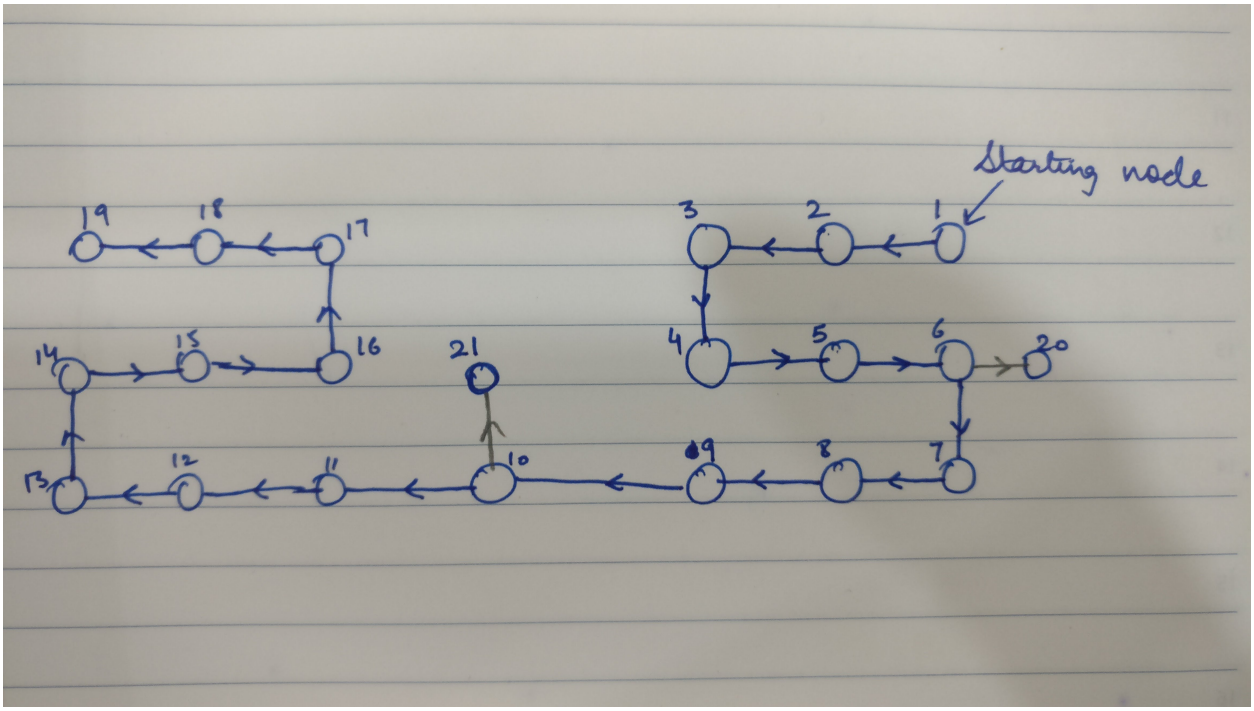
Integers given: 42,23,34,52,46,33
Hash function: h(g)=g mod10
Collision resolution technique: Linear Probing
To insert the values into the hash table in the given order, the following constraints must be followed:

- 42 must come before 52 because if 52 would have come earlier, then it would have occupied the place of 42 i.e. the index 2
- 23 must come before 33 because if 33 would have come earlier, then it would have occupied the place of 23 i.e. the index 3
- 23 and 34 must come before 52, because had they not come 52 would have gone to index 3 or 4 due to linear probing.
- Similarly 34,52,46 must come before 33 in order to have its index=7 in linear probing.

So the constraints are (23,34,42) $\rightarrow$ 52 and (23,34,52,46) $\rightarrow$ 33
but 42 $\rightarrow$ 52 $\Rightarrow$ (23,34,42,52,46) $\rightarrow$ 33
Therefore 33 must come at the end.
Now in the remaining 5 values, we have to take care of only one condition i.e. 23, 34 and 42 precede 52. 2 cases possible: Case1: 52 is at 4th place or 52 is at 5th place. When 52 is at 4th place implies 46 is at 5th place (beacuse 23,34,42 will occupy first 3 places). So number of ways= 3! = 6
Case2: 52 is at 5th place, number of ways = 4! = 24
So total number of ways are= 24+6 = 30.

Ques 5:

Maximum Possible height of DFS spanning tree=18 (Considering height of root as 0)



The top right node is the starting node for dfs tree to have maximum possible height. The edges (drawn with blue pen) are a part of maximum height branch whereas other branch edges are drawn with pencil. And back edges have not been drawn.
Explanation: If the DFS tree would have started from node 20, then in the best case tree height would have 17 because to maximize the height, edge 1 or 7 must be excluded from main branch.
If the DFS tree would have started from node 21 or 10, then only half of the graph nodes would contribute to the maximum height branch.
And if it would have stared from node 9 or 11, then also nearly half of the edges would have contributed.

Either that would be 1-8,20 or 10-19,21.
If the DFS tree would have started from some edge like 4 or 5 or 6 or 14 or 15 or 16, then also max attainable height=17.
And there are many possible DFS spanning tree that can be formed with height = 18.
So maximum possible height of DFS spanning tree = 18.

Ques 6:

Reasoning: In the ques it is required that John takes all the bridges exactly once. This problem is related to the problem of Eulerian Walk, where bridges are edges and islands are nodes. Eulerian Walk is a walk that visits every edge exactly once. But if rather than eulerian walk, there is eulerian cycle, then also John wins because then also he travels every bridge (edge). So basic condition on graphs remains that all edges must be in same component and zero or 2 nodes of odd degrees can be there (in eulerian walk, there must be 2 vertices of odd degrees i.e. the end vertices and in eulerian cycle all vertices must be even degree). So first we need to check if above conditions are satisfied or not.
Algorithm:

- Check if above conditions are satisfied or not. If they are not satisfied, print John can't win. If they are satisfied, find if eulerian walk is there or eulerian cycle is there (can be found with the help of no. of odd degree edges).
- (a) Case of eulerian cycle:
    1. Choose any node. Write a cycle containing this node. Let the cycle be a $\rightarrow$ b $\rightarrow$ c $\rightarrow$ a .
    2. Now take any in between node that has other connected nodes (outside of cycle), lets say b, write a cycle containing b like b $\rightarrow$ d $\rightarrow$ e $\rightarrow$ b .
    3. Add this cycle to the middle of upper cycle (where b was present) making it a $\rightarrow$ (b $\rightarrow$ d $\rightarrow$ e $\rightarrow$ b) $\rightarrow$ c $\rightarrow$ a .
    4. Repeat the steps 2-3 until whole graph is covered in the cycle and this cycle will be eulerian cycle.
- (b) Case of Eulerian walk:
    1. Take the 2 edges with odd degrees as starting and ending nodes for the walk. Let them be a and b.
    2. Write a path from a to b. Lets say a $\rightarrow$ d $\rightarrow$ c $\rightarrow$ b .
    3. Now like in case of eulerian cycle,take any middle vertex that has other connected nodes also. Write a cycle that contains that vertex, say d. Let the cycle be d $\rightarrow$ e $\rightarrow$ f $\rightarrow$ d .
    4. Add this cycle in place of that vertex (d) to make the path as a $\rightarrow$ (d $\rightarrow$ e $\rightarrow$ f $\rightarrow$ d) $\rightarrow$ c $\rightarrow$ b .
    5. Repeat the steps 3-4 to obtain a path that contains all nodes of the graph, this path is eulerian walk.

So the above algorithm finds a path that John needs to take to win.