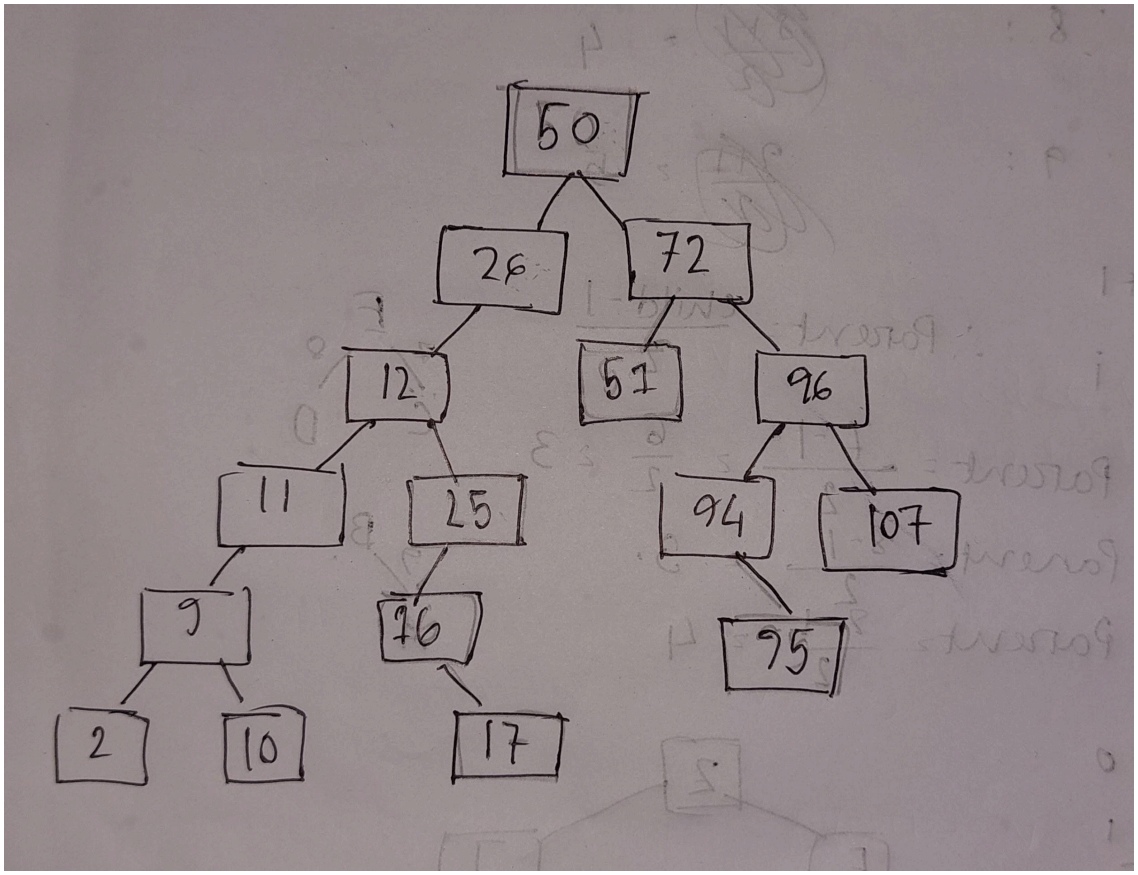
Homework -3

Course: CSE225
Section : 18
Submitted to : Rifat Ahmed Hassan [ RIH ]

Submitted by : Mushtasin Ahmed
ID : 2322175642

Homework 3

**Q1. Draw the binary search tree whose elements are inserted in the following order:**
**50 72 96 94 107 26 12 11 9 2 10 25 51 16 17 95**



**Q2.**

i) Height of the tree = 4

max(Height of left , Height of right)+1

There are 3 nodes on level 3 (since root is at level 0)

ii) Show the order in which the nodes in the tree are processed by

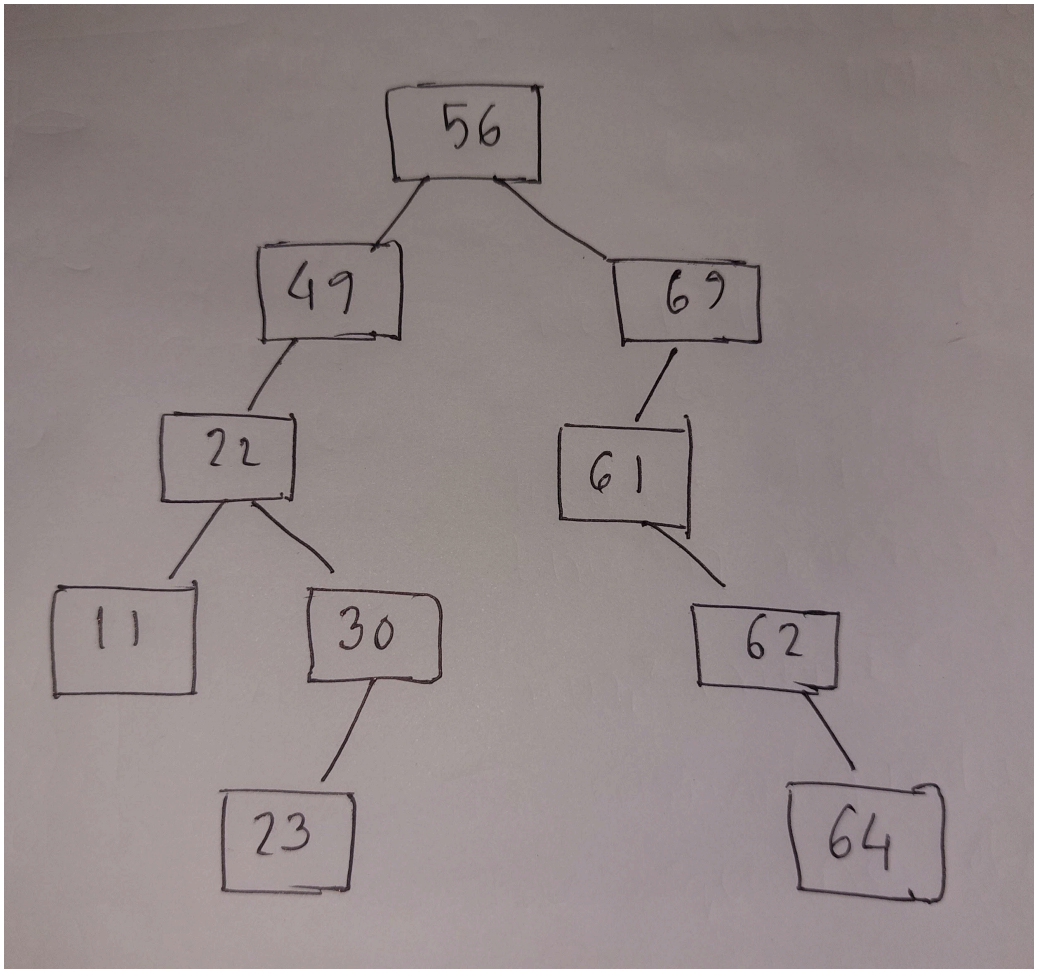
a. Inorder: 11, 23, 22, 29, 30, 47, 49, 56, 61, 62, 64, 59, 69

b. Postorder: 23, 11, 30, 29, 22, 49, 47, 61, 64, 62, 59, 69, 56

c. Preorder: 56, 47, 22, 11, 23, 29, 30, 49, 69, 59, 62, 61, 64

iii) Delete 29, 59, and 47

After deletion the binary tree would look like this :



**Q3.**

a. Extend the Binary Search Tree ADT to include the member function LeafCount that returns the number of leaf nodes in the tree

```
int BinarySearchTree::CountLeaf( Node* currentNode){
    if(currentNode==NULL){
        return 0;
    }
    if(currentNode->left == NULL && currentNode->right== NULL){
        return 1;
    }else {
        //Traverse through the left sub tree and right subtree
            return
CountLeaf(currentNode->left)+CountLeaf(currentNode->right);
}
```

B. Extend the Binary Search Tree ADT to include the member function SingleParentCount that returns the number of nodes in the tree that have only one child

```
int BinarySearchTree::SingleParentCount(Node*
currentNode){
    if(currentNode == NULL ){
        return 0;
    }
    if(currentNode->right==NULL &&
        currentNode->left !=NULL){
   return 1+SingleParentCount(currentNode->left);
    }else if(currentNode->right != NULL &&
currentNode->left==NULL){
   return 1+SingleParentCount(currentNode->right);
    }
    //if the tree has both childs
    else return SingleParentCount(currentNode->left)+
SingleParentCount(currentNode->right);
}
```

Q4. i. Store the values into a hash table with 20 positions, using the division method of hashing (mod operator) and the linear probing method of resolving collisions.
66 47 87 90 126 140 145 153 177 285 393 395 467 566 620 735

| Index | Hash Value | |
|---|---|---|
| 0 | 140 | |
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | 145 | |
| 6 | 66 | |
| 7 | 47 | |
| 8 | 87 | 7+i , i=1 |
| 9 | 126 | 6+i , i=3 |
| 10 | 90 | |
| 11 | 285 | 5+i , i=6 |
| 12 | 467 | 7+1 |
| 13 | 153 | |
| 14 | 393 | 13+i ,i=1 |
| 15 | 395 | |
| 16 | 566 | |

| | | |
|---|---|---|
| 17 | 177 | |
| 18 | 735 | |
| 19 | | |

**B. If collision Occurs we shall use the rehashing function**

Table Size = 20

| Key | Hash ( Key%20) | (Key+3)%20 | Final Position |
|---|---|---|---|
| 66 | 6 | | 6 |
| 47 | 7 | | 7 |
| 87 | 7 | 10 | 10 |
| 90 | 10 | 13 | 13 |
| 126 | 6 | 9 | 9 |
| 140 | 0 | | 0 |
| 145 | 5 | | 5 |
| 153 | 13 | | 13 |
| 177 | 17 | | 17 |
| 285 | 5 | 8 | 8 |
| 393 | 13 | 16 | 16 |
| 395 | 15 | | 15 |
| 467 | 7 | 10 | 10 |
| 566 | 6 | 9 | 9 |

| | | | |
|---|---|---|---|
| 620 | 0 | 3 | 3 |
| 735 | 15 | 18 | 18 |

Final Hash Table

| Key | Value |
|---|---|
| 0 | 140 |
| 1 | |
| 2 | 467 |
| 3 | 620 |
| 4 | |
| 5 | 145 |
| 6 | 66 |
| 7 | 47 |
| 8 | 285 |
| 9 | 126 |
| 10 | 87 |
| 11 | |
| 12 | 566 |
| 13 | 90 |
| 14 | |
| 15 | 395 |

| | |
|---|---|
| 16 | 153 |
| 17 | 177 |
| 18 | 735 |
| 19 | 393 |

**C. Use 10 buckets**

Since it has 10 buckets , so each buckets have 10 slots

We shall Fill the bucket sequentially if collision appears

| Key | Bucket Number (KEY % 10) |
|---|---|
| 66 | 6 |
| 47 | 7 |
| 87 | 7 |
| 90 | 0 |
| 126 | 6 |
| 140 | 0 |
| 145 | 5 |
| 153 | 3 |
| 177 | 7 |
| 285 | 5 |
| 393 | 3 |
| 395 | 5 |
| 467 | 7 |

| | |
|---|---|
| 566 | 6 |
| 620 | 0 |
| 735 | 5 |

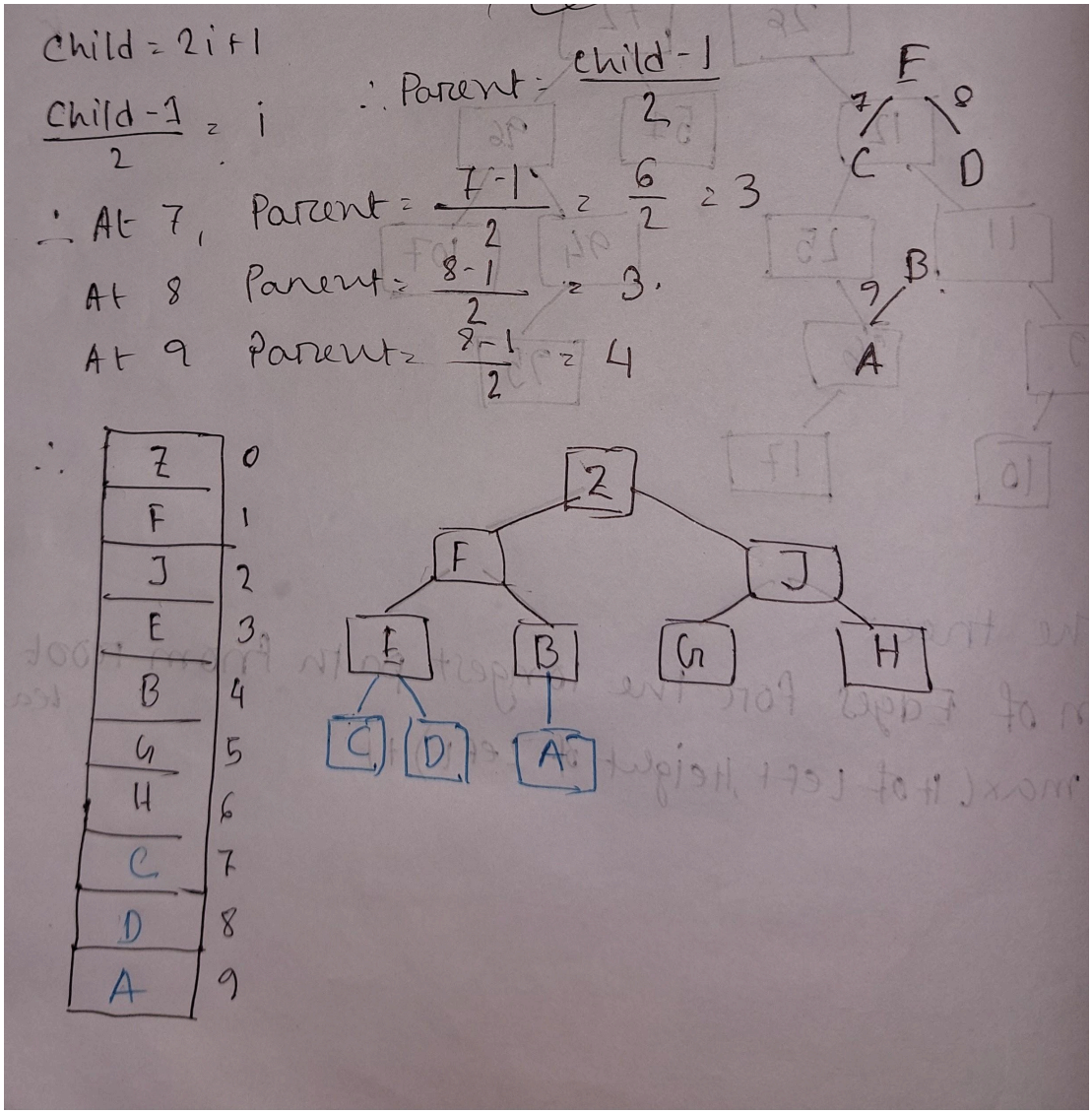
Final Buckets :

| Bucket Number | Bucket Number (KEY % 10) |
|---|---|
| 0 | 90,140 |
| 1 | 620 |
| 2 | |
| 3 | 153,393 |
| 4 | |
| 5 | 145,285,395 |
| 6 | 66,126,735 |
| 7 | 47,87,177 |
| 8 | 467,566 |
| 9 | |

## IV . Storing in Chains

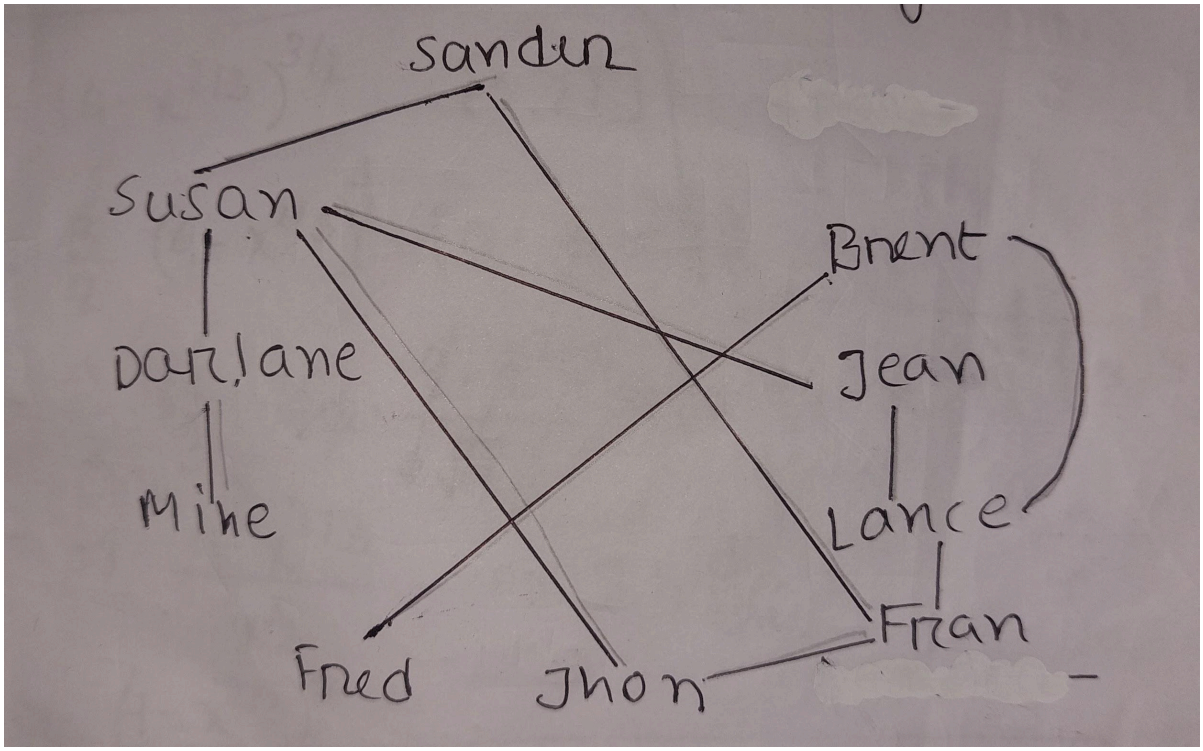
| Key | Chain ( Key % 10 ) |
|---|---|
| 0 | 90 -> 140 -> 620 |
| 1 | |
| 2 | |
| 3 | 153->393 |
| 4 | |
| 5 | 145 - 285->395-> 735 |
| 6 | 66 -> 126-> 566 |
| 7 | 47 -> 87 -> 177 ->  467 |
| 8 | |
| 9 | |

Q5. Currently, the priority queue holds 10 elements as shown. What values might be stored in array positions 7–9 so that the properties of a heap are satisfied?

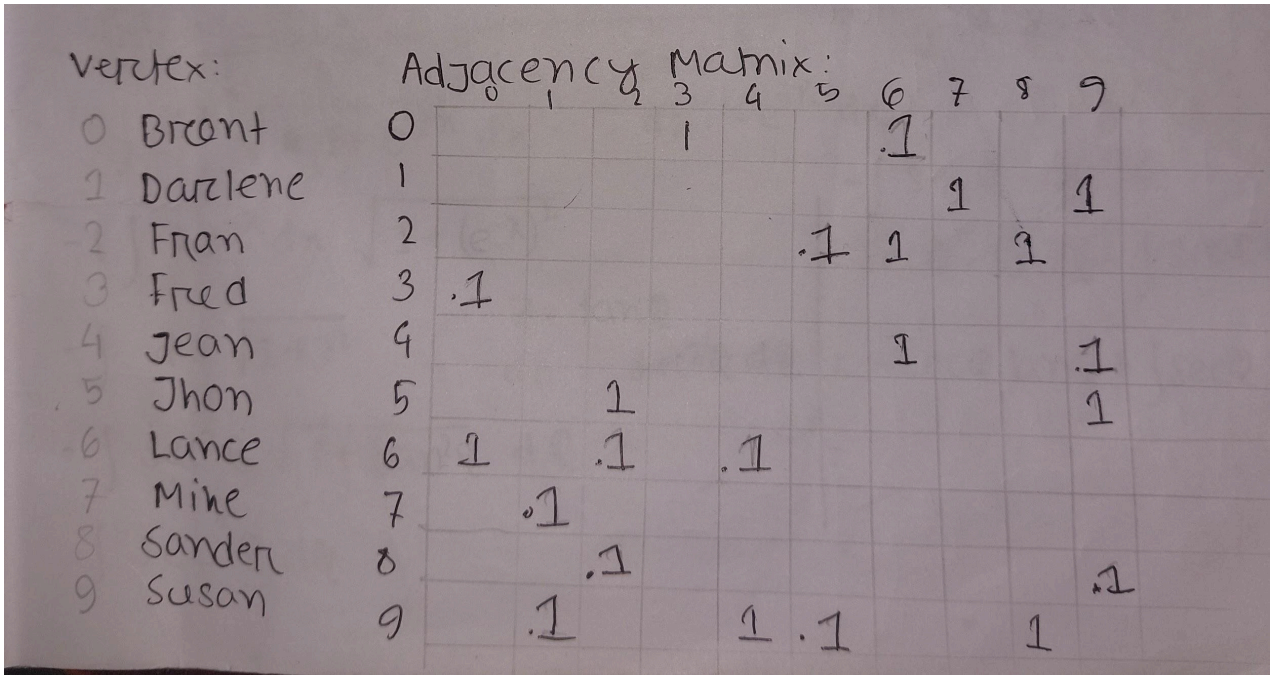


Q6.

i) Draw the Picture of Graph




ii) Draw the Graph as adjacency Matrix




The blank spots represent 0 . Where there is no 1 there are 0's

iii)

**Breadth-First Search (BFS): Susan → Lance**

In BFS, we shall explore all the nodes at the current level before moving to the next level. We use a queue for traversal.
We shall use a visited array as well to mark the visited nodes

Steps:

Start from Susan (vertex 9).
Mark Susan as visited and enqueue it: Queue = [Susan].

Explore Susan's neighbors:
Susan (9) connects to Darlene (1), Jean (4), John (5), Sander (8).
Enqueue them: Queue = [Darlene, Jean, John, Sander].

Dequeue Darlene (1):
Darlene connects to Mike (7). Enqueue Mike: Queue = [Jean, John, Sander, Mike].

Dequeue Jean (4):
Jean connects to Lance (6) and Susan (9).
Lance is the destination, so stop here.

Path Found:
Susan → Jean → Lance

**Depth-First Search (DFS): Susan → Lance**

In DFS, we explore as deep as possible along a branch before backtracking. We use a stack (or recursion) for traversal.

Steps:

Start from Susan (vertex 9).

Mark Susan as visited. Explore neighbors:
Susan connects to Darlene (1), Jean (4), John (5), Sander (8).

Visit Darlene (1):
Darlene connects to Mike (7). Explore Mike:
Mike has no unexplored neighbors. Backtrack to Susan.

Visit Jean (4):
Jean connects to Lance (6).
Lance is the destination. Stop here.

Path Found:
Susan → Jean → Lance

IV ) **D) “Works With”**
Since the Graph is mostly undirected and does not have a tree like structure so the relationship can be described as

Q7. a. Implement a edge exists function

Declaration of Function:

```
template<class VertexType>
bool GraphType<VertexType>::
EdgeExists(VertexType fromVertex, VertexType toVertex) const;
```

Function Implementation :

```
template<class VertexType>
bool GraphType<VertexType>::
EdgeExists(VertexType fromVertex, VertexType toVertex){
    //checking for valid Vertex
    if (fromVertex < 0 || fromVertex >= numVertices || toVertex < 0 ||
toVertex >= numVertices) {
        return false;
    }
    //Checking for edge
    if(vertices[fromVertex][toVertex]==1){
        return true;
    }else return false;

}
```

Q8.

a. A binary search of a sorted set of elements in an array is always faster than a sequential search of the elements

TRUE

b. A binary search is an O(N log2N) algorithm.

False

- Binary search is an O(logN)

c. A binary search of elements in an array requires that the elements be sorted from smallest to largest.

TRUE

d. When a hash function is used to determine the placement of elements in an array, the order in which the elements are added does not affect the resulting array.

FALSE

- The order in which elements are added does effect for linear probing

e. When hashing is used, increasing the size of the array always reduces the number of collisions.

TRUE

- Though it reduces only the probability but collisions still might occur

f. If we use buckets in a hashing scheme, we do not have to worry about collision resolution.

FALSE

- If we use buckets in a hashing scheme, we still need to handle collisions by managing multiple elements within each bucket

g. If we use chaining in a hashing scheme, we do not have to worry about collision resolution.

FALSE

- If we use chaining in a hashing scheme , we still need to handle collisions by adding elements to the chain

h. The goal of a successful hashing scheme is an O(1) search.
TRUE