Section 4

Data Structures & Algorithms

CSE225

Assignment 3

Submitted By:

Name: Md. Misbah Khan
ID: 2132089642

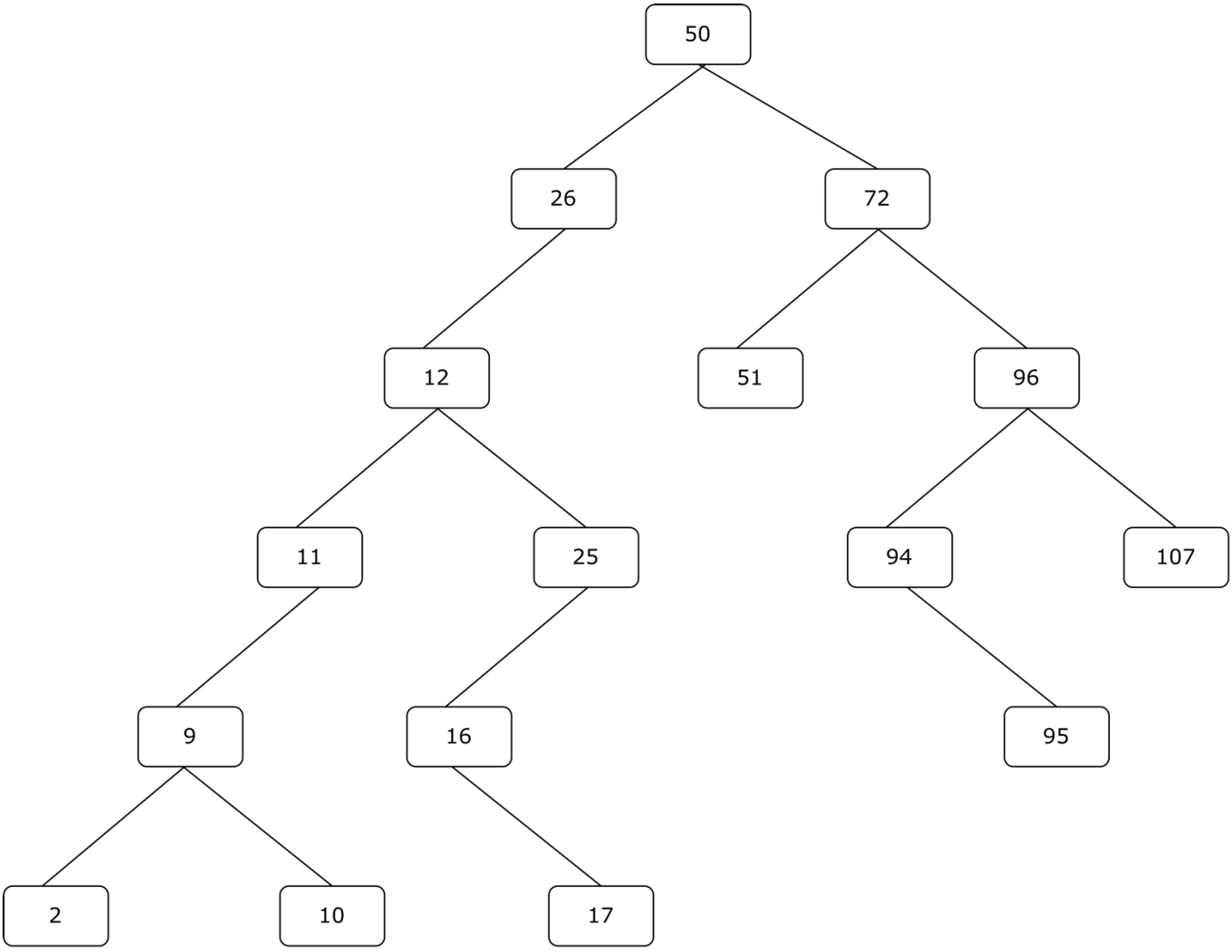
Submitted To:
Rifat Ahmed Hassan

Submitted On:
22 November 2023

Answer of Question 1: Binary search tree using these elements are:

Given elements: 50 72 96 94 107 26 12 11 9 2 10 25 51 16 17 95.

Now, in a binary search tree, first number here(50) will be root, then we will put immediate large number than 50 on right subtree, and immediate small number on right sub tree, we will keep following this process until all number are in include in the tree. Binary Search Tree using them:



Answer of Question 2(i):

a. The height of the tree is 5
b. Nodes on Level 3 are 11, 29, 62

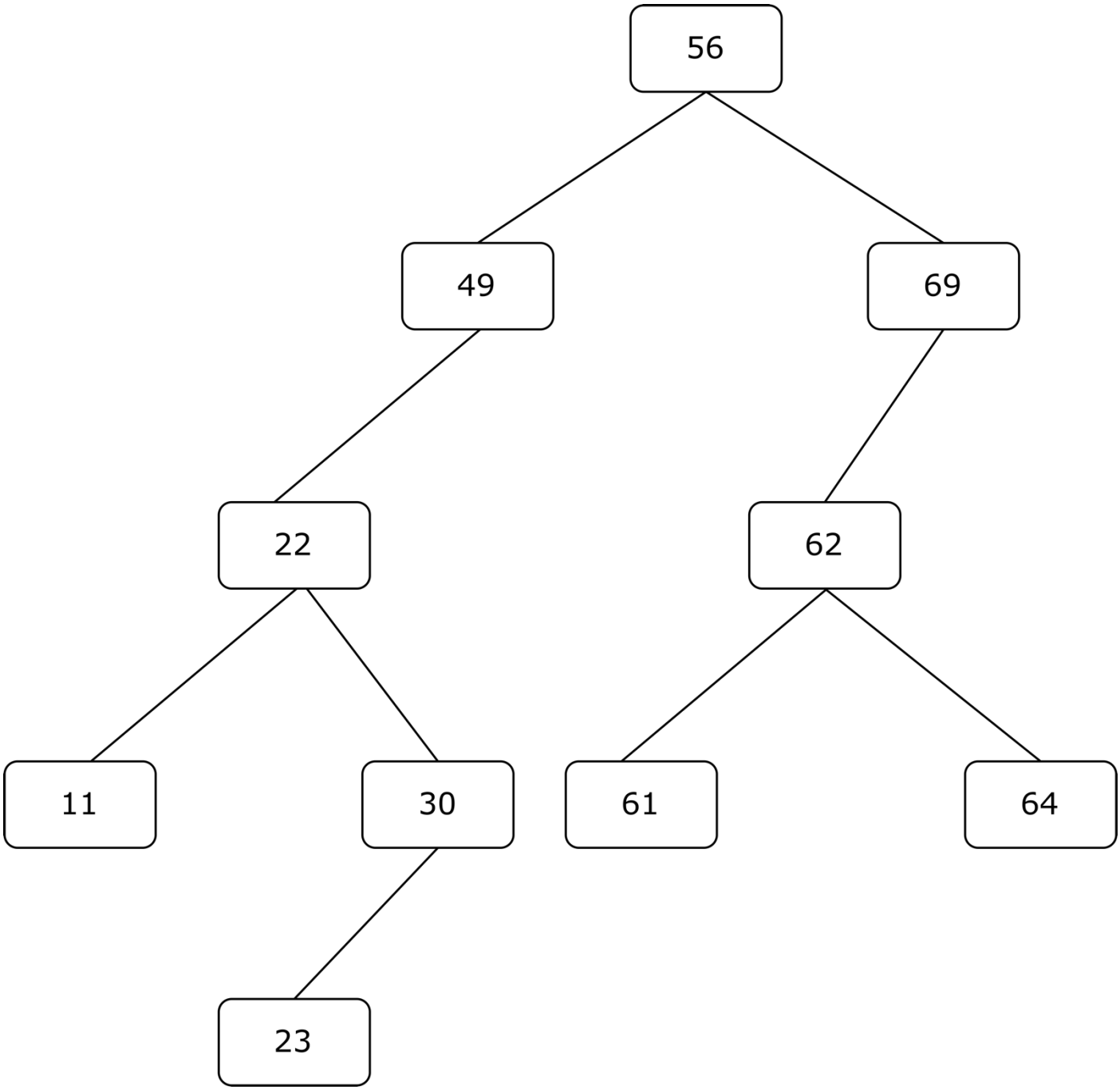
Answer of Question 2(ii):

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

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

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

Answer of Question 2(iii): The tree will look like this after the deletion of 29, 59, and 47:



Answer of Question 3(a): Here is the Extended Binary Search Tree ADT to include the member function **LeafCount** that returns the number of leaf nodes in the tree, we use the same .h and .cpp file from our slide, and modify to add the function.

::::::::::binarysearchtree.h::::::::::

```
#ifndef BINARYSEARCHTREE_H_INCLUDED
#define BINARYSEARCHTREE_H_INCLUDED
typedef char ItemType;
struct TreeNode
{
```

```
    ItemType info;
    TreeNode *left, *right;
};
enum OrderType
{
    PRE_ORDER,
    IN_ORDER,
    POST_ORDER
};
class TreeType
{
public:
    TreeType();
    ~TreeType();
    void MakeEmpty();
    bool IsEmpty();
    bool IsFull();
    int LengthIs();
    void RetrieveItem(ItemType &item, bool &found);
    void InsertItem(ItemType item);
    void DeleteItem(ItemType item);
    void ResetTree(OrderType order);
    void GetNextItem(ItemType &item, OrderType order, bool &finished);
    void Print();
    int LeafCount();

private:
    TreeNode *root;
};
#endif // BINARYSEARCHTREE_H_INCLUDED
```

::::::::::binarysearchtree.cpp::::::::::

```
#include "binarysearchtree.h"
#include <new>
TreeType::TreeType()
{
    root = NULL;
}
bool TreeType::IsEmpty()
{
    return root == NULL;
}
bool TreeType::IsFull()
{
```

```
    TreeNode *location;
    try
    {
        location = new TreeNode;
        delete location;
        return false;
    }
    catch (std::bad_alloc exception)
    {
        return true;
    }
}
int CountLeaves(TreeNode *node);

int TreeType::LeafCount()
{
    return CountLeaves(root);
}

int CountLeaves(TreeNode *node)
{
    if (node == NULL)
        return 0;
    if (node->left == NULL && node->right == NULL)
        return 1; //Node is a leaf
    return CountLeaves(node->left) + CountLeaves(node->right);
}
```

Answer of Question 3(b): Here is the extended Binary Search Tree ADT to include the member function **SingleParentCount** that returns the number of nodes in the tree that have only one child. we use the same .h and .cpp file from our slide, and modify to add the function.

::::::::::binarysearchtree.h::::::::::

```
#ifndef BINARYSEARCHTREE_H_INCLUDED
#define BINARYSEARCHTREE_H_INCLUDED
typedef char ItemType;
struct TreeNode
{
    ItemType info;
```

```
    TreeNode *left, *right;
};
enum OrderType
{
    PRE_ORDER,
    IN_ORDER,
    POST_ORDER
};
class TreeType
{
public:
    TreeType();
    ~TreeType();
    void MakeEmpty();
    bool IsEmpty();
    bool IsFull();
    int LengthIs();
    void RetrieveItem(ItemType &item, bool &found);
    void InsertItem(ItemType item);
    void DeleteItem(ItemType item);
    void ResetTree(OrderType order);
    void GetNextItem(ItemType &item, OrderType order, bool &finished);
    void Print();
    int LeafCount();
    int SingleParentCount();

private:
    TreeNode *root;
};
#endif // BINARYSEARCHTREE_H_INCLUDED
```

::::::::::binarysearchtree.cpp::::::::::

```
#include "binarysearchtree.h"
#include <new>
TreeType::TreeType()
{
    root = NULL;
}
bool TreeType::IsEmpty()
{
    return root == NULL;
}
bool TreeType::IsFull()
{
```

```
    TreeNode *location;
    try
    {
        location = new TreeNode;
        delete location;
        return false;
    }
    catch (std::bad_alloc exception)
    {
        return true;
    }
}
int CountLeaves(TreeNode *node);

int TreeType::LeafCount()
{
    return CountLeaves(root);
}

int CountLeaves(TreeNode *node)
{
    if (node == NULL)
        return 0;
    if (node->left == NULL && node->right == NULL)
        return 1;
    return CountLeaves(node->left) + CountLeaves(node->right);
}

int CountSingleParents(TreeNode *node);

int TreeType::SingleParentCount()
{
    return CountSingleParents(root);
}

int CountSingleParents(TreeNode *node)
{
    if (node == NULL)
        return 0;

    int count = 0;


    if ((node->left == NULL && node->right != NULL) || (node->left != NULL &&
node->right == NULL))
        count++;
```

```
    count += CountSingleParents(node->left);
    count += CountSingleParents(node->right);

    return count;
}
```

Answer of Question 4(i): Given values:

66 47 87 90 126 140 145 153 177 285 393 395 467 566 620 735

Table Size: 20

Hash Function: key % tableSize

Now the calculation for hashing:

Hashing	Remark
66 % 20 = 6	insert at 6
47 % 20 = 7	insert at 7
87 % 20 = 7	Collision occurs, so insert at 8
90 % 20 = 10	insert at 10
126 % 20 = 6	Collision occurs, so insert at 9
140 % 20 = 0	insert at 0
145 % 20 = 5	insert at 5
153 % 20 = 13	insert at 13
77 % 20 = 17	insert at 17
285 % 20 = 5	Collision occurs, so insert at 11
393 % 20 = 13	Collision occurs, so insert at 14
395 % 20 = 15	insert at 15
467 % 20 = 7	Collision occurs, so insert at 12
566 % 20 = 6	Collision occurs, so insert at 16
620 % 20 = 0	Collision occurs, so insert at 1
735 % 20 = 15	Collision occurs, so insert at 18

Hash Table	
0	140
1	620
2	
3	
4	
5	145
6	66
7	47
8	87
9	126
10	90
11	285
12	467
13	153
14	393
15	395
16	566
17	17
18	735
19	

Answer of Question 4(ii): Given values:
66 47 87 90 126 140 145 153 177 285 393 395 467 566 620 735
Table Size: 20
Hash Function: key % tableSize
Rehash Function: (key + 3) % tableSize
Now the calculation for hashing:

Hashing	Remark	Rehashing
66 % 20 = 6	insert at 6	Not applicable
47 % 20 = 7	insert at 7	Not applicable
87 % 20 = 7	Collision occurs	(87 + 3) % 20 = 10, insert at 10
90 % 20 = 10	Collision occurs	(90 + 3) % 20 = 13, insert at 13
126 % 20 = 6	Collision occurs	(126 + 3) % 20 = 9, insert at 9
140 % 20 = 0	insert at 0	Not applicable
145 % 20 = 5	insert at 5	Not applicable
153 % 20 = 13	Collision occurs	(153 + 3) % 20 = 16, insert at 16
177 % 20 = 17	insert at 17	Not applicable
285 % 20 = 5	Collision occurs	(285 + 3) % 20 = 8
393 % 20 = 13	Collision occurs	(393 + 3) % 20 = 16, Occupied, Rehash Again: (396+3) % 20 = 19, insert at 19
395 % 20 = 15	insert at 15	Not applicable
467 % 20 = 7	Collision occurs	(467 + 3) % 20 = 10, Occupied, Rehash Again: (470 + 3) % 20 = 13, Occupied, Rehash Again: (473 + 3) % 20 = 16, Occupied, Rehash Again: (476 + 3) % 20 = 19, Occupied, Rehash Again: (479 + 3) % 20 = 2, insert at 2
566 % 20 = 6	Collision occurs	(566 + 3) % 20 = 9, Occupied, Rehash Again: (566 + 6) % 20 = 12, insert at 12
620 % 20 = 0	Collision occurs	(620 + 3) % 20 = 3, insert at 3
735 % 20 = 15	Collision occurs	(735 + 3) % 20 = 18, insert at 18

Hash Table	
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

Answer of Question 4(iii): Given elements:
66 47 87 90 126 140 145 153 177 285 393 395 467 566 620 735

66 % 10 = 6
47 % 10 = 7
87 % 10 = 7
90 % 10 = 0
126 % 10 = 6
140 % 10 = 0
145 % 10 = 5
153 % 10 = 3
177 % 10 = 7
285 % 10 = 5
393 % 10 = 3
195 % 10 = 5
467 % 10 = 7
566 % 10 = 6
620 % 10 = 0
735 % 10 = 5

Now the table,

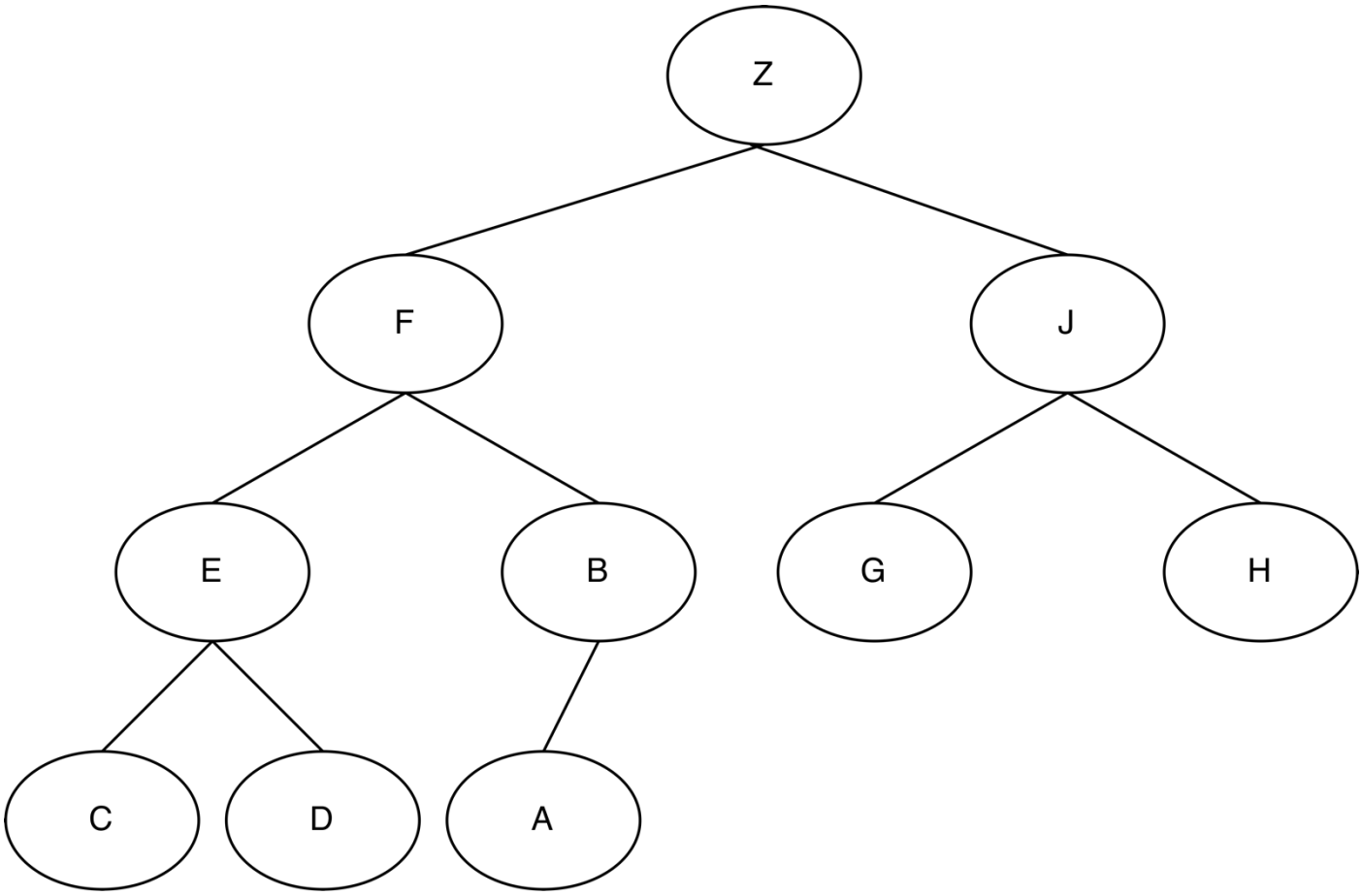
0	90	→	620	→		→	
1		→		→		→	
2		→		→		→	
3	153	→	393	→		→	
4		→		→		→	
5	145	→	285	→	395	→	735
6	66	→	126	→	566	→	
7	47	→	87	→	177	→	467
8		→		→		→	
9		→		→		→	

<u>Answer of Question 4(iii):</u>

For chaining, each element is inserted into hash table. We connect to override collision if any collision occurs.

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								

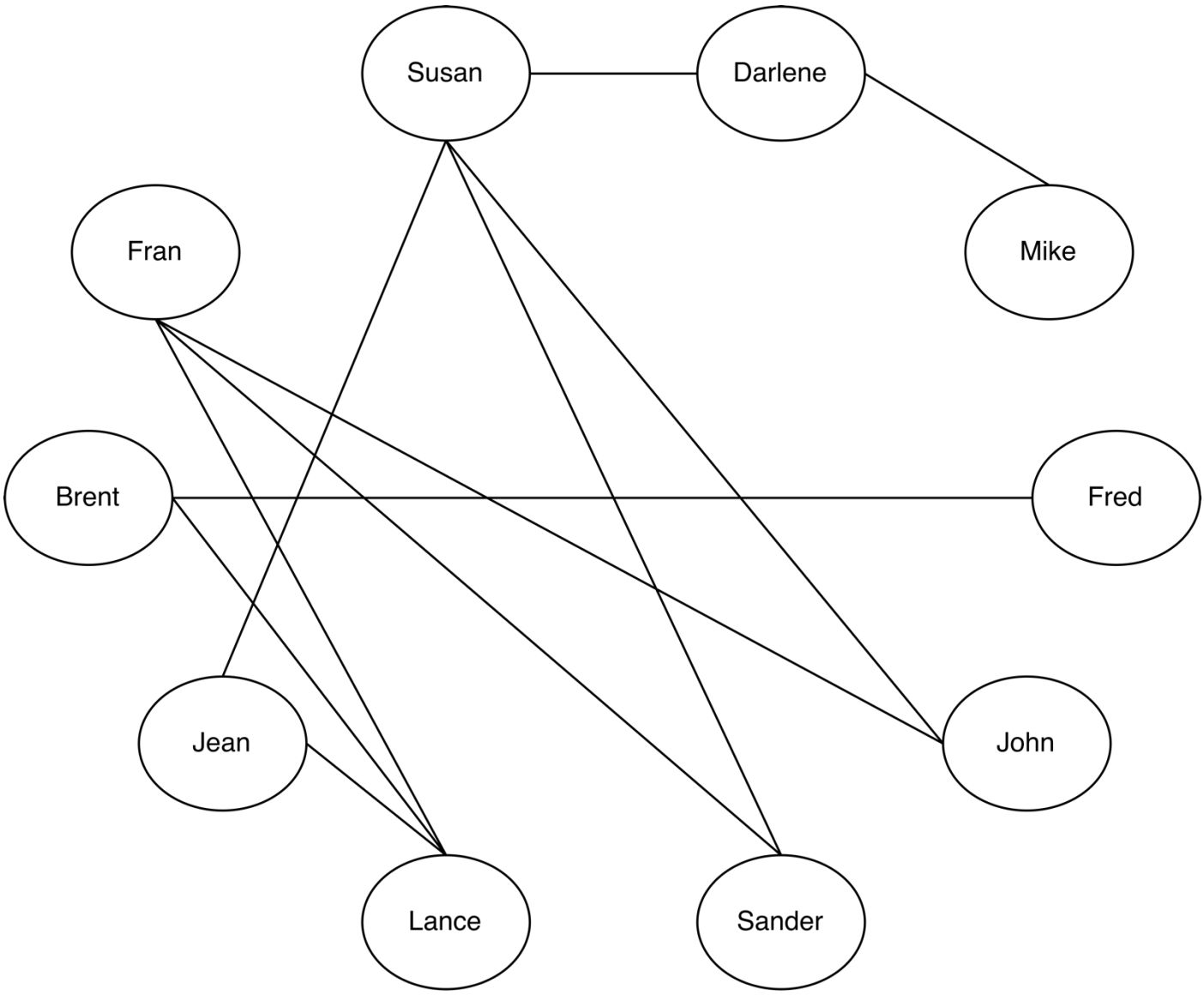
Answer of Question 5: Heaps are actually complete binary tree. So, the given table can be converted into a tree like this:



So, according to the tree, the values on 7-9 positions are:

7	C
8	D
9	A

Answer of Question 6(i): Here is the picture of EmployeeGraph:



Answer of Question 6(ii): Here is the picture of EmployeeGraph, implemented as an adjacency matrix:

0	Brent
1	Darlene
2	Fran
3	Fred
4	Jean
5	John
6	Lance
7	Mike
8	Sander
9	Susan

Vertices

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[0]	0	0	0	1	0	0	1	0	0	0
[1]	0	0	0	0	0	0	0	1	0	1
[2]	0	0	0	0	0	1	1	0	1	0
[3]	1	0	0	0	0	0	0	0	0	0
[4]	0	0	0	0	0	0	1	0	0	1
[5]	0	0	1	0	0	0	0	0	0	1
[6]	1	0	1	0	1	0	0	0	0	0
[7]	0	1	0	0	0	0	0	0	0	0
[8]	0	0	1	0	0	0	0	0	0	1
[9]	0	0	0	0	1	1	0	0	1	0

Edges

Answer of Question 6(iii):

Using the adjacency matrix for EmployeeGraph, the path from Susan to Lance is:

a) Breadth-first strategy: Susan is connected to Jean and then Jean is connected to Lance. So, Susan > Jean > Lance
b) Depth-first strategy: Susan is connected to John, John is connected to Fran, Fran is connected to Lance. So, Susan > John > Fran > Lance

Answer of Question 6(iv): "is senior to" phrases best describes the relationship represented by the edges between the vertices in EmployeeGraph.

"works for" cannot best describe the relationship as one person cannot work for many people at a time, but in the graph, it can be seen that Fran is connected to 3 other people.

"is the supervisor" is also not possible, because one person cannot be supervised by 3 person.

"works with" is not suitable because Susan is connected to Jean and Jean is connected to Lance, so it could be said that Susan, Jean, and Lance work with each other, but again Lance is working with Brent and Fran at the same time, So it might be possible but not the best option. So, we left with one option, "is senior to". Which describes every connected vertices perfectly

Answer of Question 7(a): Here is the declaration of the **EdgeExists** function:

:::::::::: graphtype.h::::::::::

```
#ifndef GRAPHTYPE_H_INCLUDED
#define GRAPHTYPE_H_INCLUDED

#include "stacktype.h"
#include "quetype.h"

template<class VertexType>
class GraphType {
    public:
        GraphType();
        GraphType(int maxV);
        ~GraphType();
        void MakeEmpty();
        bool IsEmpty();
        bool IsFull();
        void AddVertex(VertexType);
        void AddEdge(VertexType, VertexType, int);
        int WeightIs(VertexType, VertexType);
        void GetToVertices(VertexType, QueType<VertexType> &);
        void ClearMarks();
        void MarkVertex(VertexType);
        bool IsMarked(VertexType);
        bool EdgeExists(VertexType, VertexType); //Declaration in header file

    private:
        int numVertices;
        int maxVertices;
        VertexType *vertices;
        int **edges;
        bool *marks;
};

#endif // GRAPHTYPE_H_INCLUDED
```

:::::::::: graphtype.cpp::::::::::

```
#include <iostream>
#include "graphtype.h"
#include "stacktype.cpp"
#include "quetype.cpp"

using namespace std;

const int NULL_EDGE = 0;

template <class VertexType>
GraphType<VertexType>::GraphType()
{
    numVertices = 0;
    maxVertices = 50;
    vertices = new VertexType[50];
    edges = new int *[50];
    for (int i = 0; i < 50; i++)
        edges[i] = new int[50];
    marks = new bool[50];
}

template <class VertexType>
GraphType<VertexType>::GraphType(int maxV)
{
    numVertices = 0;
    maxVertices = maxV;
    vertices = new VertexType[maxV];
    edges = new int *[maxV];
    for (int i = 0; i < maxV; i++)
        edges[i] = new int[maxV];
    marks = new bool[maxV];
}

template <class VertexType>
GraphType<VertexType>::~GraphType()
{
    delete[] vertices;
    delete[] marks;
    for (int i = 0; i < maxVertices; i++)
        delete[] edges[i];
    delete[] edges;
}

template <class VertexType>
void GraphType<VertexType>::MakeEmpty()
```

```
{
    numVertices = 0;
}

template <class VertexType>
bool GraphType<VertexType>::IsEmpty()
{
    return (numVertices == 0);
}

template <class VertexType>
bool GraphType<VertexType>::IsFull()
{
    return (numVertices == maxVertices);
}

template <class VertexType>
void GraphType<VertexType>::AddVertex(VertexType vertex)
{
    vertices[numVertices] = vertex;
    for (int index = 0; index < numVertices; index++)
    {
        edges[numVertices][index] = NULL_EDGE;
        edges[index][numVertices] = NULL_EDGE;
    }
    numVertices++;
}

template <class VertexType>
int IndexIs(VertexType *vertices, VertexType vertex)
{
    int index = 0;
    while (!(vertex == vertices[index]))
        index++;
    return index;
}

template <class VertexType>
void GraphType<VertexType>::ClearMarks()
{
    for (int i = 0; i < maxVertices; i++)
        marks[i] = false;
}

template <class VertexType>
void GraphType<VertexType>::MarkVertex(VertexType vertex)
{
```

```
    int index = IndexIs(vertices, vertex);
    marks[index] = true;
}

template <class VertexType>
bool GraphType<VertexType>::IsMarked(VertexType vertex)
{
    int index = IndexIs(vertices, vertex);
    return marks[index];
}

template <class VertexType>
void GraphType<VertexType>::AddEdge(VertexType fromVertex, VertexType toVertex,
int weight)
{
    int row = IndexIs(vertices, fromVertex);
    int col = IndexIs(vertices, toVertex);
    edges[row][col] = weight;
}

template <class VertexType>
int GraphType<VertexType>::WeightIs(VertexType fromVertex, VertexType toVertex)
{
    int row = IndexIs(vertices, fromVertex);
    int col = IndexIs(vertices, toVertex);
    return edges[row][col];
}

template <class VertexType>
void GraphType<VertexType>::GetToVertices(VertexType vertex,
QueType<VertexType> &adjVertices)
{
    int fromIndex, toIndex;
    fromIndex = IndexIs(vertices, vertex);
    for (toIndex = 0; toIndex < numVertices; toIndex++)
        if (edges[fromIndex][toIndex] != NULL_EDGE)
            adjVertices.Enqueue(vertices[toIndex]);
}

/* Added  EdgeExists function which determines whether two vertices are
connected by an edge */

template <class VertexType>
bool GraphType<VertexType>:: EdgeExists(VertexType fromVertex, VertexType
toVertex)
{
    int row = IndexIs(vertices, fromVertex);
```

```
    int col = IndexIs(vertices, toVertex);
    if (edges[row][col] == 1)
        return true;
    else
        return false;
}
```

Answer of Question 7(b): Using the adjacency matrix implementation developed in the chapter and the declaration from part (a), here is the implementation of the body of the function:

:::::::::: main.cpp::::::::::

```
#include <iostream>
#include "graphtype.cpp"

using namespace std;
int main()
{
    GraphType<char> graph;

    graph.AddVertex('A');
    graph.AddVertex('B');
    graph.AddVertex('C');
    graph.AddVertex('D');
    graph.AddVertex('E');
    graph.AddVertex('F');
    graph.AddVertex('G');
    graph.AddVertex('H');

    graph.AddEdge('A', 'B', 1);
    graph.AddEdge('H', 'A', 1);
    graph.AddEdge('H', 'B', 1);
    graph.AddEdge('G', 'H', 1);
    graph.AddEdge('G', 'D', 1);
    graph.AddEdge('E', 'G', 1);
    graph.AddEdge('D', 'E', 1);
    graph.AddEdge('D', 'F', 1);
    graph.AddEdge('D', 'C', 1);
    graph.AddEdge('F', 'C', 1);

    if (graph.EdgeExists('A', 'B'))
        cout << "yes" << endl;
    else
        cout << "no" << endl;
    return 0;
}
```

Answer of Question 8:

a) False. A binary search of a sorted set of elements in an array is not always faster than a sequential search of the elements.
b) False. A binary search is an O(log N) algorithm.
c) True.
d) True.
e) False. When hashing is used, increasing the size of the array doesn't always reduce the number of collisions.
f) False. If we use buckets in a hashing scheme, we have to worry about collision resolution.
g) False. If we use chaining in a hashing scheme, we have to worry about collision resolution.
h) True.