Data and Analysis of Algorithms
Homework 3

1. Given n keys and arbitrary binary tree T with n nodes,
Add a key y to unique keys, not distinct keys, rest of tree is x.
Then compare x and y. If $y < x$, add y to the left subtree of x, and if $y > x$, add y to the right subtree of x.
Then let $y < x$ so that it nests in the left subtree of x. Then go to the child z of y and make the comparison again.
If $z < y$, add z to left subtree of y and if $z > y$, add z to right subtree of y
Then let $z > y$ so that z is in the right subtree. Repeat until the end of the tree is reached. Doing this will show that the key is in a true or false position. With this knowledge, we have only one place to add the new key in new tree n+1.
And, with that, it proves that there is a unique assignment of a key to the root.

Homework 3 continued

2. Inputs: queue, elements in queue, new priority value
   Output: A queue with a changed priority value element.

```
Function: ChangePriority(Q, x, v)
        hold = Q[x]
        Q[x] = v
        If v > hold Then IncreasePrio(x)
        Else, DecreasePrio(x)
        End IF
End Function

Function: IncreasePrio(Q, x)
        while x > 0 And Q[Parent(x)] < Q[x]
            hold = Q[Parent(x)]
            Q[Parent(x)] = Q[x]
            Q[x] = hold
            x = Parent[x]
        End while
End Function

Function: DecreasePrio(Q, x)
        a = x , b = 2x+1
        If b ≤ Size(Q) And Q[b] > Q[a] Then
            a = b, c = 2x+2
        End If
        If c ≤ Size(Q) and Q[c] > Q(a) Then
            a=c
        End If
        If x != a Then
        hold = Q[x], Q[x] = Q[a], Q[a] = hold
        DecreasePrio(Q, a)
        End If
End Function
```

3.

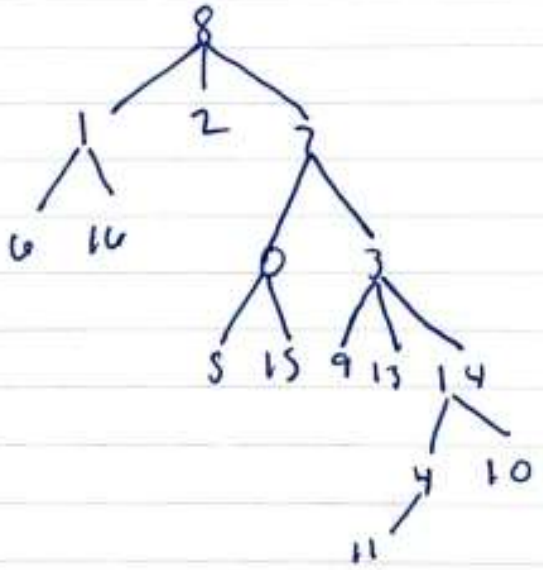
a)

12

b)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
7	8	8	7	14	0	1	8	-16	3	14	4	-1	3	3	0	1

12



c)

Find2(Parent[0:16], 4) → will give a result of 8

Since the sequence of parents is 4, 14, 3, 7, 8

The tree remains the same as part B

```
Input the # of vertices, then the sequence of edges, and end with neg integer.

Number of vertices = 6

i = 0
j = 2
[0, 2]

i = 2
j = 4
[2, 4]

i = 1
j = 3
[1, 3]

i = -1


V = {0,1,2,3,4,5}
E = {{0,2},{2,4},{1,3}}




        0       0       1       0       0       0
        0       0       0       1       0       0
        1       0       0       0       1       0
        0       1       0       0       0       0
        0       0       1       0       0       0
        0       0       0       0       0       0

0-->    2
1-->    3
2-->    0 4
3-->    1
4-->    2
5-->


Number of Components: 3
1)      0       2       4
2)      1       3
3)      5
```

```
Input the # of vertices, then the sequence of edges, and end with neg integer.

Number of vertices = 7

i = 0
j = 1
[0, 1]

i = 1
j = 2
[1, 2]

i = 2
j = 3
[2, 3]

i = 5
j = 6
[5, 6]

i = -1


V = {0,1,2,3,4,5,6}
E = {{0,1},{1,2},{2,3},{5,6}}




        0       1       0       0       0       0       0
        1       0       1       0       0       0       0
        0       1       0       1       0       0       0
        0       0       1       0       0       0       0
        0       0       0       0       0       0       0
        0       0       0       0       0       0       1
        0       0       0       0       0       1       0

0-->    1
1-->    0 2
2-->    1 3
3-->    2
4-->
5-->    6
6-->    5


Number of Components: 3
1)      0       1       2       3
2)      4
3)      5       6
```

```
Input the # of vertices, then the sequence of edges, and end with neg integer.

Number of vertices = 8

i = 0
j = 4
[0, 4]

i = 4
j = 7
[4, 7]

i = -1


V = {0,1,2,3,4,5,6,7}
E = {{0,4},{4,7}}




        0       0       0       0       1       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        1       0       0       0       0       0       0       1
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       1       0       0       0

0-->    4
1-->
2-->
3-->
4-->    0 7
5-->
6-->
7-->    4


Number of Components: 6
1)      0       4       7
2)      1
3)      2
4)      3
5)      5
6)      6
```

```
Input the # of vertices, then the sequence of edges, and end with neg integer.

Number of vertices = 5

i = 0
j = 1
[0, 1]

i = 1
j = 4
[1, 4]

i = 2
j = 3
[2, 3]

i = 3
j = 4
[3, 4]

i = 1
j = 3
[1, 3]

i = -1


V = {0,1,2,3,4}
E = {{0,1},{1,4},{2,3},{3,4},{1,3}}




        0       1       0       0       0
        1       0       0       1       1
        0       0       0       1       0
        0       1       1       0       1
        0       1       0       1       0

0-->    1
1-->    0 4 3
2-->    3
3-->    2 4 1
4-->    1 3


Number of Components: 1
1)      0       1       2       3       4
```

```cpp
#include <iostream>
#include <vector>
#include <map>
using namespace std;

class Node {
    public:
        int data;
        Node* next;
};

class Graph {
    public:
        Graph(int numVertices, map<int, map<int, int>> E) {

            _numVertices = numVertices;
            _numEdges = E.size();

            initializeAdjacencyList(); // Creates an array of linked lists, each linked list is filled with NULL value
            initializeAdjacencyMatrix();
            resetVisited(); // Creates an array of size _numVertices and fills all values as 'false' or '0'

            fillAdjacencyList(E); // Create the initial adjacency list, uses Edges to compute
        }

        ~Graph() { deleteEntireAdjacencyList(); }

        void edgeAddition(int i, int j) { addNode(i, j); addNode(j, i); _numEdges++; Components(); } // Calls append twice, flipping the parameters.

        void edgeDeletion(int i, int j) { removeNode(i, j); removeNode(j, i); _numEdges--; Components(); }

        void DFS(int vertex) {
            Node* head = adjacencyList[vertex];
            vector<int> adjacentVertices = Visit(head);
            int length = listLength(head);

            visited[vertex] = true;
            for (int i = 0; i < length; i++) {
                if (visited[adjacentVertices.at(i)] == false) {
                    DFS(adjacentVertices.at(i));
                }
            }
        }

        void Components() {// computes the vertex sets of the connected components of G. Involves calling DFS(G, v), v = 0, 1, ..n - 1.
            _numComponents = 0;
```

```
47              components = vector<vector<int>>();
48              fillAdjacencyMatrix();
49
50              for (int i = 0; i < _numVertices; i++) {
51                  DFS(i);
52                  if (_numComponents == 0) { components.push_back(getComponent());
                      _numComponents++; }
53                  else { if (!isRepeatedComponent()) { components.push_back
                      (getComponent()); _numComponents++; } }
54                  resetVisited();
55              }
56          }
57
58          void printAllComponents() {
59              cout << endl << endl << "Number of Components: " << _numComponents <<
                   endl;
60              for (int i = 0; i < _numComponents; i++) {
61                  vector<int> vertexSetOfComponent = components.at(i);
62                  cout << i + 1 << ")";
63                  for (int j = 0; j < vertexSetOfComponent.size(); j++) {
64                      cout << '\t' << vertexSetOfComponent.at(j);
65                  }
66                  cout << endl;
67              }
68          }
69
70          void printAdjacencyList() {
71              cout << endl << endl;
72              for (int i = 0; i < _numVertices; ++i) {
73                  Node* temp = *(adjacencyList + i);
74
75                  cout << i << "-->\t";
76
77                  while (temp != NULL) {
78                      cout << temp->data << " ";
79                      temp = temp->next;
80                  }
81                  cout << '\n';
82              }
83          }
84
85          void printAdjacencyMatrix() {
86              cout << endl << endl;
87              for (int i = 0; i < _numVertices; i++) {
88                  cout << endl;
89                  for (int j = 0; j < _numVertices; j++) {
90                      cout << '\t' << adjacencyMatrix[i][j];
91                  }
92              }
93          }
94
95          void printVisited() { for (int i = 0; i < _numVertices; i++) { cout <<
```

```
            visited[i] << endl; } }
 96
 97     protected: // Basically all helper methods that cannot be called from outside the Graph class.
 98
 99         void addNode(int i, int j) { // Appends vertex j to the end of linked list of vertex i
100         // 1. create and allocate node
101             Node* newNode = new Node;
102             Node* last = adjacencyList[i];
103             // 2. assign data to the node
104             newNode->data = j;
105             // 3. set next pointer of new node to null as its the last node
106             newNode->next = NULL;
107             // 4. if list is empty, new node becomes first node
108             if (adjacencyList[i] == NULL)
109             {
110                 adjacencyList[i] = newNode;
111                 return;
112             }
113             // 5. Else traverse till the last node
114             while (last->next != NULL)
115                 last = last->next;
116             // 6. Change the next of last node
117             last->next = newNode;
118             return;
119         }
120
121         void removeNode(int i, int j) {
122             // Store head node
123             Node* temp = adjacencyList[i];
124             Node* prev = new Node();
125
126             // If head node itself holds
127             // the key to be deleted
128             if (temp != NULL && temp->data == j)
129             {
130                 adjacencyList[i] = temp->next; // Changed head
131                 delete temp;            // free old head
132                 return;
133             }
134
135             // Else Search for the key to be deleted,
136             // keep track of the previous node as we
137             // need to change 'prev->next' */
138             else
139             {
140                 while (temp != NULL && temp->data != j)
141                 {
142                     prev = temp;
143                     temp = temp->next;
144                 }
```

```
145
146                 // If key was not present in linked list
147                 if (temp == NULL)
148                     return;
149
150                 // Unlink the node from linked list
151                 prev->next = temp->next;
152
153                 // Free memory
154                 delete temp;
155             }
156         }
157
158         vector<int> Visit(Node* head) {
159             vector<int> vec;
160             while (head != NULL) {
161                 vec.push_back(head->data);
162                 head = head->next;
163             }
164             return vec;
165         }
166
167         void resetVisited() { visited = new bool[_numVertices]; for (int i = 0; i ↵
               < _numVertices; i++) { visited[i] = false; } }
168
169         bool isRepeatedComponent() {
170             for (int i = 0; i < _numComponents; i++) {
171                 if (getComponent() == components.at(i)) { return true; }
172             }
173             return false;
174         }
175
176         vector<int> getComponent() {
177             vector<int> verticesInComponent;
178             for (int i = 0; i < _numVertices; i++) {
179                 if (visited[i] == 1) {
180                     verticesInComponent.push_back(i);
181                 }
182             }
183             return verticesInComponent;
184         }
185
186         int listLength(Node* head) {
187             if (head == NULL) { return 0; }
188             return 1 + listLength(head->next);
189         }
190
191         void initializeAdjacencyList() { adjacencyList = new Node *                ↵
              [_numVertices]; for (int i = 0; i < _numVertices; ++i) { adjacencyList  ↵
              [i] = NULL; } }
192
193         void fillAdjacencyList(map<int, map<int, int>> originalEdges) {
```

```
194             map<int, map<int, int> >::iterator itr;
195             map<int, int>::iterator ptr;
196
197             for (itr = originalEdges.begin(); itr != originalEdges.end(); itr++)
                  {
198                 for (ptr = itr->second.begin(); ptr != itr->second.end(); ptr++)
                      {
199                     edgeAddition(ptr->first, ptr->second);
200                 }
201             }
202         }
203
204         void deleteEntireAdjacencyList() {
205             for (int i = 0; i < _numVertices; i++) {
206                 Node* current = adjacencyList[i];
207                 Node* next;
208                 while (current != NULL) {
209                     next = current->next;
210                     free(current);
211                     current = next;
212                 }
213                 adjacencyList[i] = NULL;
214             }
215         }
216
217         void initializeAdjacencyMatrix() {
218             bool** matrix = new bool*[_numVertices];
219             for (int i = 0; i < _numVertices; i++) { matrix[i] = new bool
                  [_numVertices]; }
220             adjacencyMatrix = matrix;
221         }
222
223         void fillAdjacencyMatrix() {
224             for (int i = 0; i < _numVertices; i++) {
225                 vector<int> connectedComponents = Visit(adjacencyList[i]);
226                 for (int j = 0; j < _numVertices; j++) {
227                     adjacencyMatrix[i][j] = false;
228                     for (vector<int>::iterator itr = connectedComponents.begin();
                          itr != connectedComponents.end(); ++itr) {
229                         adjacencyMatrix[i][*itr] = true;
230                     }
231                 }
232             }
233         }
234
235     private:
236         int _numVertices;
237         int _numEdges;
238
239         vector<vector<int>> components;
240         int _numComponents;
241
```

```
242         Node** adjacencyList;
243
244         bool** adjacencyMatrix;
245         bool* visited;
246 };
247
248
249
250 void printVertices(int vertices) {
251     cout << "V = {";
252     for (int i = 0; i < vertices - 1; i++) { cout << i << ","; }
253     cout << vertices - 1 << "}" << endl;
254 }
255
256 void printEdges(map<int, map<int, int>> edgeMapping) {
257     int counter = edgeMapping.size();
258     cout << "E = {";
259     map<int, map<int, int> >::iterator itr;
260     map<int, int>::iterator ptr;
261     for (itr = edgeMapping.begin(); itr != edgeMapping.end(); itr++) {
262         for (ptr = itr->second.begin(); ptr != itr->second.end(); ptr++) {
263             cout << "{" << ptr->first << "," << ptr->second << "}";
264             if (counter != 1) { cout << ","; }
265             counter--;
266         }
267     }
268     cout << "}" << endl;
269 }
270
271 map<int, map<int,int>> addEdgeForE(map<int, map<int, int>> edgeMapping, int i,
      int j) {
272     int index = edgeMapping.size();
273     edgeMapping.insert(make_pair(index, map<int, int>()));
274     edgeMapping[index].insert(make_pair(i, j));
275     return edgeMapping;
276 }
277
278 int main() {
279     vector<int> V;
280     map<int, map<int, int>> E;
281
282     int numVertices, i, j = 0;
283     int loopCount = 0;
284
285     cout << "Input the # of vertices, then the sequence of edges, and end with
        neg integer." << endl << endl;
286
287     cout << "Number of vertices = ";
288     cin >> numVertices;
289     cout << endl;
290
291     while (true) {
```

```
292         if (loopCount % 2 == 0) {
293             cout << "i = ";
294             cin >> i;
295         }
296         else {
297             cout << "j = ";
298             cin >> j;
299         }
300         if ((i < 0 || j < 0) || ((i || j) > numVertices - 1)) { break; }
301         if (loopCount % 2 == 1) {
302             E = addEdgeForE(E, i, j);
303             cout << "[" << i << ", " << j << "]\n" << endl;
304         }
305         loopCount++;
306     }
307     cout << endl << endl;
308
309     printVertices(numVertices);
310     printEdges(E);
311
312     cout << endl;
313
314     Graph graph = Graph(numVertices, E);
315     graph.Components();
316
317     graph.printAdjacencyMatrix();
318     graph.printAdjacencyList();
319     graph.printAllComponents();
320
321     graph.~Graph();
322 }
```