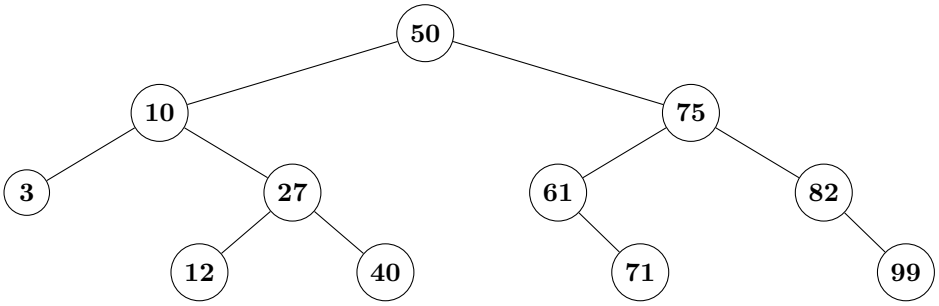
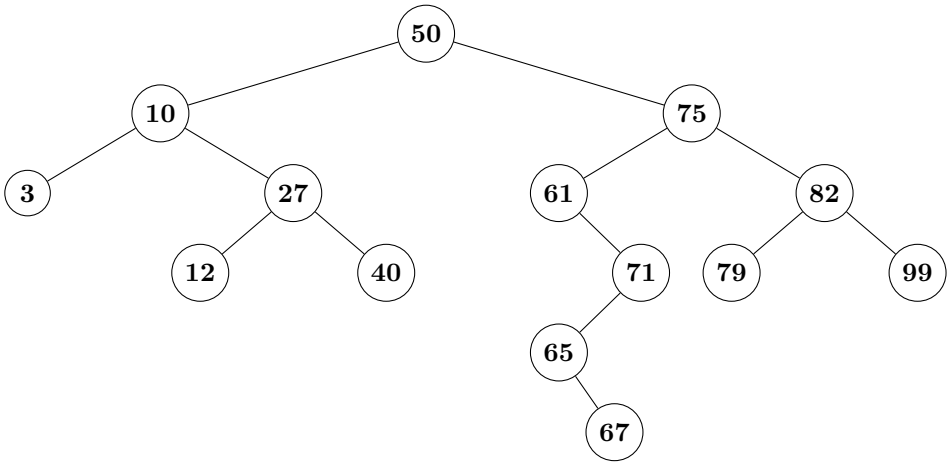
1 Theory

1. (*15 points*) Binary Search Trees: Given the following Binary Search Tree:

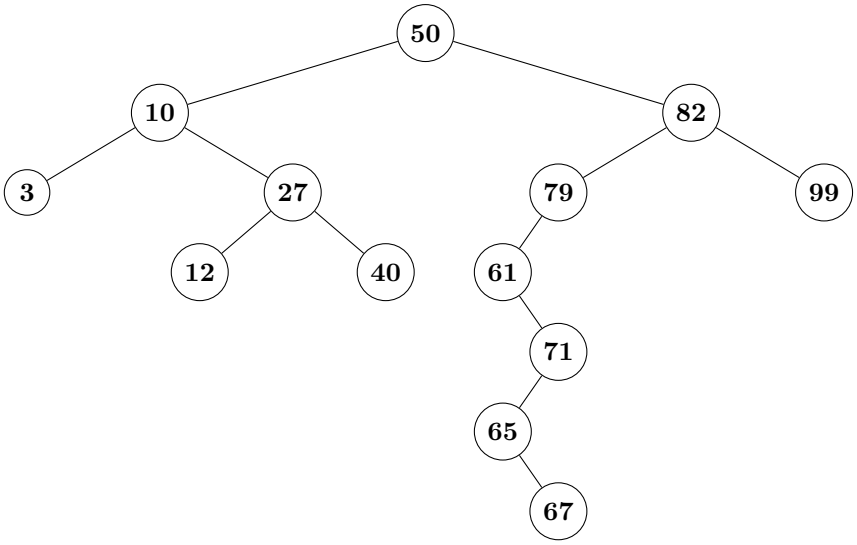


   (a) Display the order of node visits for a post-order traversal of the tree.
   Solution: LRV: 3, 12, 40, 27, 10, 71, 61, 99, 82, 75, 50.

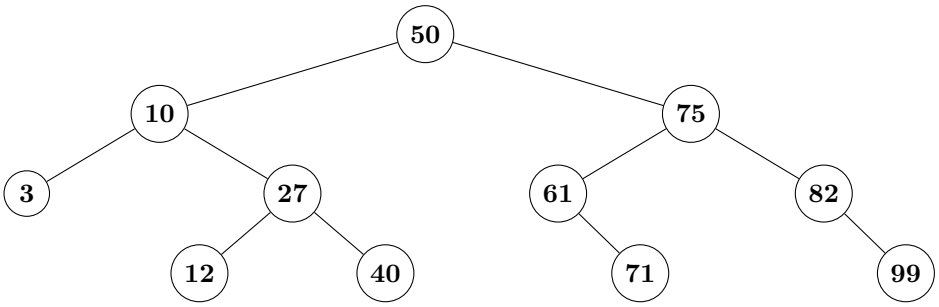
   (b) Draw the tree after 65, 67 and 79 have been added, in that order.
   Solution:



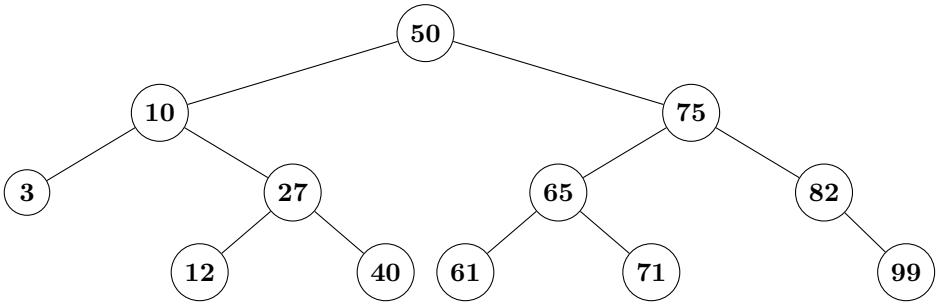
   (c) Draw the tree after 75 has been removed (by merging with the successor node), this is from the resulting tree of the previous exercise.
   Solution:



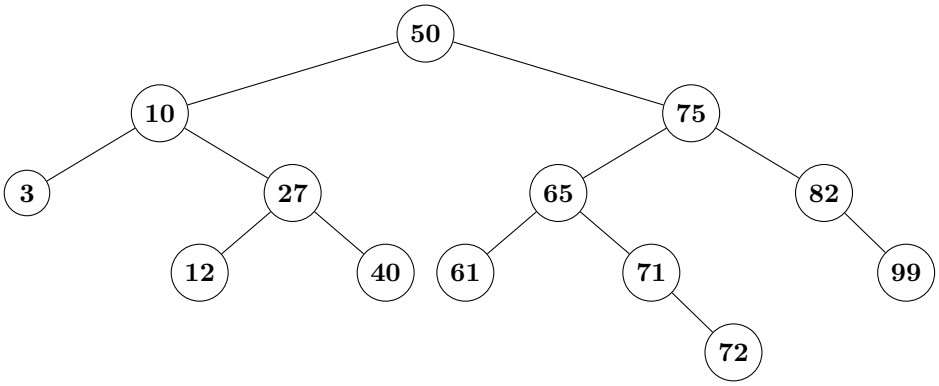
2. (*15 points*) AVL Trees: Given the following AVL Tree:



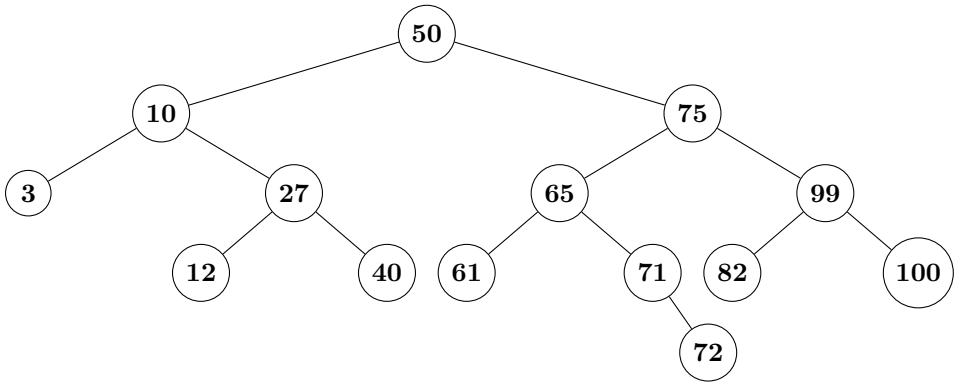
(a) Draw the tree after 65 has been added
Solution:



(b) Draw the tree after 72 has been added, this is from the resulting tree of the previous exercise.
Solution:



(c) Draw the tree after 100 has been added, this is from the resulting tree of the previous exercise.
Solution:



3. (*15 points*) Red-Black Trees: What is the criterion for a binary search tree to be a Red-Black tree?

   Solution:

   A red-black tree is a binary search tree with one extra bit of storage per node: its color, which can be either RED or BLACK. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced.

   A red-black tree is a binary tree that satisfies the following red-black properties:

   (a) Every node is either red or black.

   (b) The root is black.

   (c) Every leaf (NIL) is black.

   (d) If a node is red, then both its children are black.

   (e) For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

4. (*15 Points*) Asymptotic Analysis:

   (a) State the precise mathematical definitions for $O(g(n))$, $\Omega(g(n))$, and $\Theta(g(n))$.

   Solution:

   $f(n)$ is $O(g(n))$ if there exist positive numbers c and N such that $f(n) \leq cg(n)$ for all $n \geq N$.
   $f(n)$ is $\Omega(g(n))$ if there exist positive numbers c and N such that $f(n) \geq cg(n)$ for all $n \geq N$.
   $f(n)$ is $\Theta(g(n))$ if there exist positive numbers c_1, c_2, and N such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq N$.

   (b) Show that $n^{1.1} + n\lg(n)$ is $\Theta(n^{1.1})$.

   Solution:

$$\lim_{n\to\infty} \frac{n^{1.1} + n\lg(n)}{n^{1.1}} = 1 + \lim_{n\to\infty} \frac{\lg(n)}{n^{.1}} = 1 + \lim_{n\to\infty} \frac{\frac{1}{n\ln(2)}}{.1n^{-.9}} = 1 + \lim_{n\to\infty} \frac{1}{.1n^{.1}\ln(2)} = 1 \neq 0, \infty$$

5. (*15 Points*) Sorts: Of the sorts we have discussed in class: bubble, insertion, selection, merge, quick, comb, Shell, radix, count, heap, and bucket.

   (a) Which sorts in the average case have complexity, $O(n^2)$?
   Solution: Bubble, insertion, and selection.

   (b) Which sorts in the average case have complexity, $O(n\lg(n))$?
   Solution: Merge, quick, comb, heap, and Shell

   (c) Which sorts in the average case have linear complexity?
   Solution: Radix, count, and bucket.

   (d) Which sort is a modification of the bubble sort?
   Solution: Comb

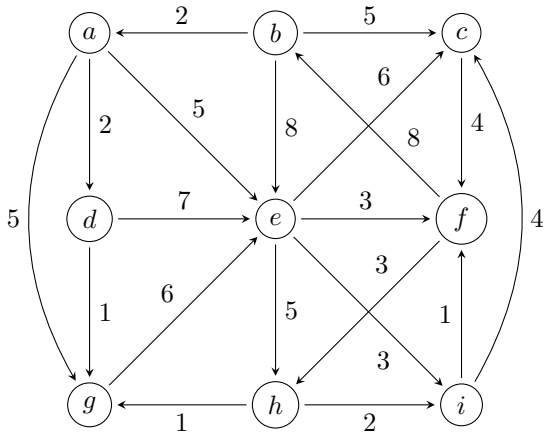
   (e) Which sort is a modification of the insertion sort?
   Solution: Shell

   (f) Which sort is a modification of the selection sort?
   Solution: Heap

   (g) Which sorts have restrictions on the data stored in the array and what are those restrictions?
   Solution:
   radix — non-negative integer data.
   count — positive integer data.
   bucket — floating point data in [0, 1).

6. (*20 Points*) Graphs:

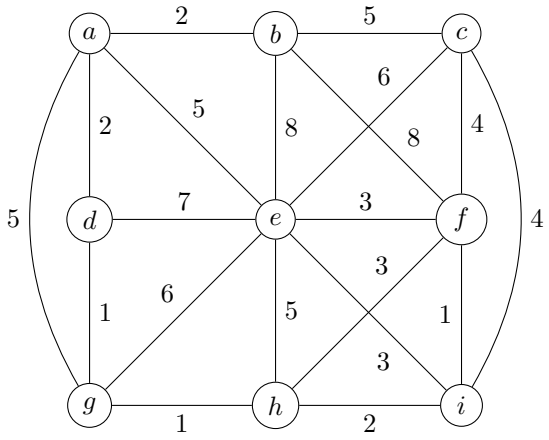
   (a) Go through all the steps of Dijkstra's Algorithm to determine the shortest path from vertex a to all other vertices. Create a chart of the steps as we did in class and is represented in the text.



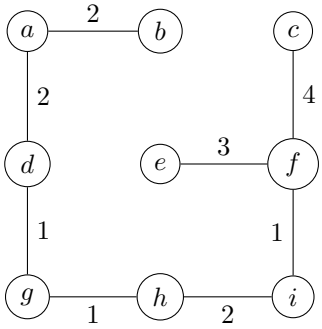
Solution:

Active Vertex		a	d	g	e	f	i	h	c
a	0								
b	∞	∞	∞	∞	∞	16	16	16	16
c	∞	∞	∞	∞	11	11	11	11	
d	∞	2							
e	∞	5	5	5					
f	∞	∞	∞	∞	8				
g	∞	5	3						
h	∞	∞	∞	∞	10	10	10		
i	∞	∞	∞	∞	8	8			

(b) Go through all the steps of Kruskal's Algorithm to determine the minimal spanning tree of the following graph.



Solution:



7. (*15 Points*) Hash Tables: Given a hash table of size 10 that is storing integers, a hash function that hashes an integer to itself, an empty flag of -1, a removed flag of -2 and linear probing.

   (a) Draw the hash table after the following numbers have been added in this order: 123, 432, 543, 555, 129, 662, 714, 975, 114.

   Solution:

0	1	2	3	4	5	6	7	8	9
114	-1	432	123	543	555	662	714	975	129

   (b) Then from this table draw the hash table after the removal of 543 and 662.

   Solution:

0	1	2	3	4	5	6	7	8	9
114	-1	432	123	-2	555	-2	714	975	129

   (c) Then from this table draw the hash table after the addition, in this order, of 747 and 111.

   Solution:

0	1	2	3	4	5	6	7	8	9
114	747	432	123	111	555	-2	714	975	129

2 Coding

1. (*35 Points*) Trees: Given the following specifications for the templated AVL tree with external node class implementation. Write the `LeftRotation`, `RightRotation`, and `Balance` functions.

```
template<class T> class AVLTreeNode {
public:
    T value;
    int height = 1;  //  Height of the subtree below the node.
    AVLTreeNode *left;
    AVLTreeNode *right;
    AVLTreeNode() {
        left = nullptr;
        right = nullptr;
    }
};

template<class T> class AVLTree: public BinaryTree<T, AVLTreeNode<T>> {
protected:
    void insert(AVLTreeNode<T>*&, AVLTreeNode<T>*&);
    void deleteNode(T, AVLTreeNode<T>*&);
    void LeftRotation(AVLTreeNode<T>*&);
    void RightRotation(AVLTreeNode<T>*&);
    int getBalanceFactor(AVLTreeNode<T>*);
    int getHeight(AVLTreeNode<T>*);
    void Balance(AVLTreeNode<T>*&);
    bool isBalanced(AVLTreeNode<T>*);
    void IndentBlock(int);
    void PrintTreeHB(AVLTreeNode<T>*, int, int);
public:
    AVLTree() {}
    ~AVLTree() {    }
    void insert(T);
    void remove(T);
    virtual bool find(const T&);
    virtual AVLTreeNode<T>* findNode(const T&);
    bool isBalanced();
    void PrintTreeHB(int Indent = 4, int Level = 0);
};

template<class T, class Node = TreeNode<T>> class BinaryTree {
protected:
    Node *root;
    void destroySubTree(Node*);
    void InOrderRec(Node*, void (*fct)(T&)) const;
    void PreOrderRec(Node*, void (*fct)(T&)) const;
    void PostOrderRec(Node*, void (*fct)(T&)) const;
    void IndentBlock(int num);
    void PrintTreeRec(Node *t, int Indent, int Level);
    int CountNodes(Node *nodePtr);
    int height(Node *nodePtr);
public:
    BinaryTree() {root = nullptr;}
    virtual ~BinaryTree() {destroySubTree(root);}
    virtual void insert(T) = 0;
    virtual void remove(T) = 0;
    void InOrder(void (*fct)(T&)) const {InOrderRec(root, fct);}
    void PreOrder(void (*fct)(T&)) const {PreOrderRec(root, fct);}
    void PostOrder(void (*fct)(T&)) const {PostOrderRec(root, fct);}
    int CountNodes();
    int height();
    void PrintTree(int Indent = 4, int Level = 0);
};
```

Solution:

```
template<class T> void AVLTree<T>::LeftRotation(AVLTreeNode<T> *&nodePtr)
    {
    AVLTreeNode<T> *R = nodePtr->right;
    AVLTreeNode<T> *temp = R->left;
    R->left = nodePtr;
    nodePtr->right = temp;
    nodePtr->height = max(getHeight(nodePtr->left), getHeight(nodePtr->
       right)) + 1;
    R->height = max(getHeight(R->left), getHeight(R->right)) + 1;
    nodePtr = R;
}

template<class T> void AVLTree<T>::RightRotation(AVLTreeNode<T> *&nodePtr)
     {
    AVLTreeNode<T> *L = nodePtr->left;
    AVLTreeNode<T> *temp = L->right;
    L->right = nodePtr;
    nodePtr->left = temp;
    nodePtr->height = max(getHeight(nodePtr->left), getHeight(nodePtr->
       right)) + 1;
    L->height = max(getHeight(L->left), getHeight(L->right)) + 1;
    nodePtr = L;
}

template<class T> void AVLTree<T>::Balance(AVLTreeNode<T> *&nodePtr) {
    if (!nodePtr)
        return;

    // Rebalance if needed.
    int balanceFactor = getBalanceFactor(nodePtr);

    // Rebalance if needed.
    // Left heavy.
    if (balanceFactor > 1) {
        if (getBalanceFactor(nodePtr->left) > 0)
            RightRotation(nodePtr);
        else {
            LeftRotation(nodePtr->left);
            RightRotation(nodePtr);
        }
    }

    // Right heavy.
    if (balanceFactor < -1) {
        if (getBalanceFactor(nodePtr->right) > 0) {
            RightRotation(nodePtr->right);
            LeftRotation(nodePtr);
        } else
            LeftRotation(nodePtr);
    }
}
```

2. (*35 Points*) Graphs: Given the following code framework, code any two of the following. All implementations are to be templated.

   (a) Depth first search of the given graph. This will print out a list of edges to the console in order of the edge traversal for this algorithm. The graph is stored as a ListOfLists object in adjacency list form (not an adjacency matrix).

   `void depthFirstSearch(ListOfLists<T> G)`

   (b) Breadth first search of the given graph. This will print out a list of edges to the console in order of the edge traversal for this algorithm. The graph is stored as a ListOfLists object in adjacency list form (not an adjacency matrix).

   `void breadthFirstSearch(ListOfLists<T> G)`

   (c) Dijkstra's algorithm to find the shortest path. This will return a vector of weighted nodes that will store the node name and the minimal distance from the start node to the node itself. The graph is stored as a list (vector) of weighted edge objects. The parameter start is the initial vertex to start from.

   `vector<wnode<T>> DijkstraAlgorithm(vector<wedge<T>> G, T start)`

   (d) Ford's algorithm to find the shortest path. This will return a vector of weighted nodes that will store the node name and the minimal distance from the start node to the node itself. The graph is stored as a list (vector) of weighted edge objects. The parameter start is the initial vertex to start from.

   `vector<wnode<T>> FordAlgorithm(vector<wedge<T>> G, T start)`

   (e) Kruskal's algorithm to find the minimal spanning tree. This will return a vector of weighted edges that will store the edge list for the construction of the minimal spanning tree. The graph is stored as a list (vector) of weighted edge objects.

   `vector<wedge<T>> KruskalAlgorithm(const vector<wedge<T>> &G)`

```
template<class T>
class edge {
public:
    T f, t;

    edge(T from, T to) {
        f = from;
        t = to;
    }
};

template<class T>
class wnode {
public:
    T name;
    double weight;

    wnode(T t, double w = 0) {
        name = t;
        weight = w;
    }

    friend ostream& operator <<(ostream &strm, const wnode &obj) {
        strm << obj.name << " : " << obj.weight;
        return strm;
    }

};

template<class T>
```

```
class wedge {
public:
    T from, to;
    double weight;

    wedge(T f, T t, double w = 0) {
        from = f;
        to = t;
        weight = w;
    }

    bool operator<(const wedge &rhs) {
        return weight < rhs.weight;
    }

    bool operator>(const wedge &rhs) {
        return weight > rhs.weight;
    }

    bool operator==(const wedge &rhs) {
        return (weight == rhs.weight) && (from == rhs.from) &&
        (to == rhs.to);
    }

    friend ostream& operator <<(ostream &strm, const wedge &obj) {
        strm << obj.from << " -> " << obj.to << " : " << obj.weight;
        return strm;
    }
};
```

You may also assume that you have the `ListOfLists` structure we used in class, specification is below.

```
template<class T>
class ListOfLists {
protected:
    vector<vector<T>> list;

public:
    ListOfLists(int rows = 0, int cols = 0);
    virtual ~ListOfLists();

    int size();
    void addRow();
    void addRows(int rows = 1, int cols = 0);
    void push_back(vector<T>);
    vector<T>& operator[](const int&);
};
```

You may also assume that you have the following functions at your disposal without creating them.

- `int findVertexPos(ListOfLists<T> G, T v)` returns the position of the vertex v in the ListOfLists structure for graph G.
- `int getWnodePos(const vector<wnode<T>> &nodes, T node)` returns the position of the weighted node *node* in the list of weighted nodes *nodes*.
- `int findMinWnodePos(const vector<wnode<T>> &nodes)` returns the position of the minimum weighted node in the list of weighted nodes *nodes*.
- `bool detectCycles(ListOfLists<T> G)` that will return true if there is a cycle in the graph and false otherwise.

Solution:

```
template<class T>
void depthFirstSearch(ListOfLists<T> G) {
    vector<int> num;
    vector<edge<T>> E;
    int count = 1;

    for (int i = 0; i < G.size(); i++)
        num.push_back(0);

    while (find(num.begin(), num.end(), 0) < num.end()) {
        int pos = find(num.begin(), num.end(), 0) - num.begin();
        DFS(G, num, pos, count, E);
    }

    for (edge<T> e : E)
        cout << e.f << " - " << e.t << endl;
}

template<class T>
void DFS(ListOfLists<T> G, vector<int> &num, int pos, int &count,
        vector<edge<T>> &E) {
    vector<T> Adj = G[pos];
    num[pos] = count++;

    for (int i = 1; i < Adj.size(); i++) {
        T vert = Adj[i];

        int vPos = findVertexPos(G, vert);
        if (vPos >= 0 && num[vPos] == 0) {
            E.push_back( { G[pos][0], vert });
            DFS(G, num, vPos, count, E);
        }
    }
}

////////////////////////////////////////////////////

template<class T>
void breadthFirstSearch(ListOfLists<T> G) {
    vector<int> num;
    vector<T> V;
    vector<edge<T>> E;
    deque<T> queue;
    int count = 1;

    for (int i = 0; i < G.size(); i++) {
        num.push_back(0);
        V.push_back(G[i][0]);
    }

    while (find(num.begin(), num.end(), 0) < num.end()) {
        int pos = find(num.begin(), num.end(), 0) - num.begin();
        num[pos] = count++;
        queue.push_back(G[pos][0]);
        while (!queue.empty()) {
            T vert = queue.front();
            queue.pop_front();
            int vPos = findVertexPos(G, vert);
            vector<T> Adj = G[vPos];
            for (int i = 1; i < Adj.size(); i++) {
                int AdjvPos = findVertexPos(G, Adj[i]);
                if (num[AdjvPos] == 0) {
                    num[AdjvPos] = count++;
                    queue.push_back(Adj[i]);
                    E.push_back( { vert, Adj[i] });
```

```
                    }
                }
            }
        }

        for (edge<T> e : E)
            cout << e.f << " - " << e.t << endl;
    }

    ////////////////////////////////////////////////////

    template<class T>
    void DijkstraInit(const vector<wedge<T>> &G, vector<wnode<T>> &nodes) {
        double max = 1E100; // DBL_MAX;

        for (int i = 0; i < G.size(); i++) {
            wedge<T> e = G[i];
            bool newnode = true;
            for (int j = 0; j < nodes.size(); j++) {
                if (e.from == nodes[j].name)
                    newnode = false;
            }
            if (newnode)
                nodes.push_back( { e.from, max });

            newnode = true;
            for (int j = 0; j < nodes.size(); j++) {
                if (e.to == nodes[j].name)
                    newnode = false;
            }
            if (newnode)
                nodes.push_back( { e.to, max });
        }
    }

    template<class T>
    vector<wnode<T>> DijkstraAlgorithm(vector<wedge<T>> G, T start) {
        vector<wnode<T>> nodes;
        vector<wnode<T>> tobechecked;
        int upos, tbcpos;

        DijkstraInit(G, nodes);
        nodes[getWnodePos(nodes, start)].weight = 0;
        for (int i = 0; i < nodes.size(); i++)
            tobechecked.push_back(nodes[i]);

        while (!tobechecked.empty()) {
            tbcpos = findMinWnodePos(tobechecked);
            wnode<T> v = tobechecked[tbcpos];
            tobechecked.erase(tobechecked.begin() + tbcpos);
            v = nodes[getWnodePos(nodes, v.name)];

            for (int j = 0; j < G.size(); j++)
                if (v.name == G[j].from)
                    for (int i = 0; i < tobechecked.size(); i++)
                        if ((tbcpos = getWnodePos(tobechecked, G[j].to)) !=
                            -1) {
                            upos = getWnodePos(nodes, G[j].to);
                            if (nodes[upos].weight > v.weight + G[j].weight) {
                                nodes[upos].weight = v.weight + G[j].weight;
                                tobechecked[tbcpos].weight = nodes[upos].
                                    weight;
                            }
                        }
        }
```

```
    return nodes;
}

////////////////////////////////////////////////////

template<class T>
vector<wnode<T>> FordAlgorithm(vector<wedge<T>> G, T start) {
    vector<wnode<T>> nodes;
    int upos, vpos;

    DijkstraInit(G, nodes);
    nodes[getWnodePos(nodes, start)].weight = 0;

    bool finished = false;
    while (!finished) {
        finished = true;

        for (int j = 0; j < G.size(); j++) {
            upos = getWnodePos(nodes, G[j].to);
            vpos = getWnodePos(nodes, G[j].from);
            if (nodes[upos].weight > nodes[vpos].weight + G[j].weight) {
                nodes[upos].weight = nodes[vpos].weight + G[j].weight;
                finished = false;
            }
        }
    }

    return nodes;
}

////////////////////////////////////////////////////

template<class T>
vector<wedge<T>> KruskalAlgorithm(const vector<wedge<T>> &G) {
    vector<wedge<T>> MST;
    vector<wedge<T>> MST_test;
    vector<wedge<T>> H = G;
    sort(H.begin(), H.end());
    int vcount = vertexList(H).size();

    for (int i = 0; i < H.size() && MST.size() < vcount - 1; i++) {
        wedge<T> e = H[i];
        MST_test = MST;
        MST_test.push_back(e);

        if (!detectCycles(MST_test)) {
            MST = MST_test;
        }
    }
    return MST;
}
```

3. (*30 Points*) Hash Tables: Given the following specification to a templated array-based hash table with hash function pointer `hf`, empty position flag `empty`, removed position flag `removed`, and linear probe. Write the constructor, destructor, probe, insert, remove, find, and rehash functions.

```
template<class T>
class HashTable {
protected:
    int size = 0;
    T *tab = nullptr;
    int (*hf)(T&);
    T empty;
    T removed;

    int probe(int pos);

public:
    HashTable(int sz, int (*hashfct)(T&), T e, T r);
    virtual ~HashTable();
    void insert(T);
    void remove(T);
    bool find(T);
    void rehash(int sz);
    void print();
};
```

The following example produces the following output.

```
int hf(int &val) {
    return val;
}

int main() {
    srand(time(0));

    HashTable<int> table(10, hf, -1, -2);

    table.insert(437543);
    table.insert(3284);
    table.insert(234);
    table.insert(11111);
    table.insert(1103);

    table.print();
    cout << endl;
    cout << table.find(3284) << endl;
    cout << table.find(123456) << endl;
    cout << table.find(1103) << endl;

    table.remove(1103);
    cout << table.find(1103) << endl;
    cout << endl;

    cout << table.find(22) << endl;
    cout << endl;

    table.print();
    cout << endl;

    table.rehash(7);
    table.print();
    cout << endl;

    cout << table.find(3284) << endl;
    cout << table.find(123456) << endl;
    cout << table.find(1103) << endl;

    return 0;
}
```

```
0: -1
1: 11111
2: -1
3: 437543
4: 3284
5: 234
6: 1103
7: -1
8: -1
9: -1

1
0
1
0

0

0: -1
1: 11111
2: -1
3: 437543
4: 3284
5: 234
6: -2
7: -1
8: -1
9: -1

0: -1
1: 437543
2: 11111
3: 3284
4: 234
5: -1
6: -1

1
0
0
```

Solution:

```
#ifndef HASHTABLE_H_
#define HASHTABLE_H_
#include <iostream>
using namespace std;

template<class T>
class HashTable {
protected:
    int size = 0;
    T *tab = nullptr;
    int (*hf)(T&);
    T empty;
    T removed;
    int probe(int pos);

public:
    HashTable(int sz, int (*hashfct)(T&), T e, T r);
    virtual ~HashTable();
    void insert(T);
    void remove(T);
    bool find(T);
    void rehash(int sz);
    void print();
};

template<class T>
HashTable<T>::HashTable(int sz, int (*hashfct)(T&), T e, T r) {
    tab = new T[sz];
    size = sz;
    hf = hashfct;
    empty = e;
    removed = r;

    for (int i = 0; i < size; i++)
        tab[i] = empty;
}

template<class T>
HashTable<T>::~HashTable() {
    delete[] tab;
}

template<class T>
int HashTable<T>::probe(int pos) {
    // Linear probe.
    return (pos + 1) % size;
}

template<class T>
void HashTable<T>::insert(T item) {
    int pos = hf(item) % size;
    int initpos = pos;

    // If position is open probe loop is not done.
    while (tab[pos] != empty && tab[pos] != removed) {
        pos = probe(pos);
        if (pos == initpos)
            return;
    }

    tab[pos] = item;
}
```

```
template<class T>
void HashTable<T>::remove(T item) {
    int pos = hf(item) % size;
    int initpos = pos;

    // If item is in first guess, probe loop is not done.
    while (tab[pos] != item) {
        pos = probe(pos);
        if (pos == initpos || tab[pos] == empty)
            return;
    }
    tab[pos] = removed;
}

template<class T>
bool HashTable<T>::find(T item) {
    int pos = hf(item) % size;
    int initpos = pos;

    // If item is in first guess, probe loop is not done.
    while (tab[pos] != item) {
        pos = probe(pos);
        if (pos == initpos || tab[pos] == empty)
            return false;
    }

    return true;
}

template<class T>
void HashTable<T>::rehash(int sz) {
    HashTable<T> newtable(sz, hf, empty, removed);
    for (int i = 0; i < size; i++) {
        if (tab[i] != empty && tab[i] != removed) {
            T item = tab[i];
            newtable.insert(item);
        }
    }

    delete[] tab;
    tab = newtable.tab;
    size = newtable.size;
    newtable.tab = nullptr;
}

template<class T>
void HashTable<T>::print() {
    for (int i = 0; i < size; i++)
        cout << i << ": " << tab[i] << endl;
}

#endif /* HASHTABLE_H_ */
```