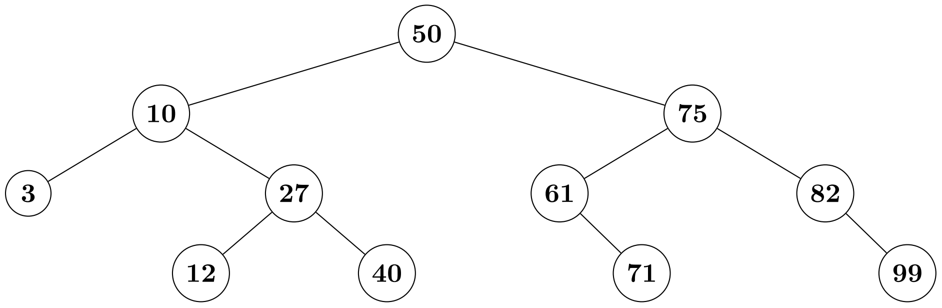
Name: ______________________________

Write all of your responses on these exam pages. If you need extra space please use the backs of the pages.

1 Theory

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

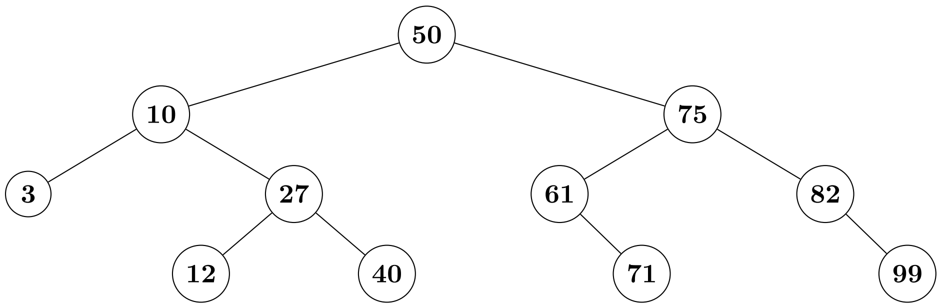


   (a) Display the order of node visits for a post-order traversal of the tree.

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

(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.

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



(a) Draw the tree after 65 has been added

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

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

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

4. (*15 Points*) Asymptotic Analysis:

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

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

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, $O(n^2)$?

   (b) Which sorts in the average case have complexity, $O(n \lg(n))$?

   (c) Which sorts in the average case have linear complexity?

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

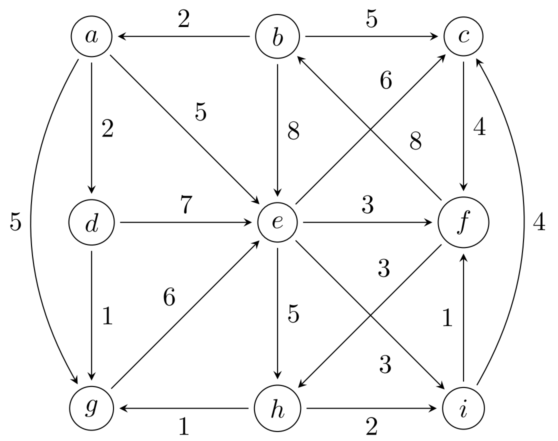
   (e) Which sort is a modification of the insertion sort?

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

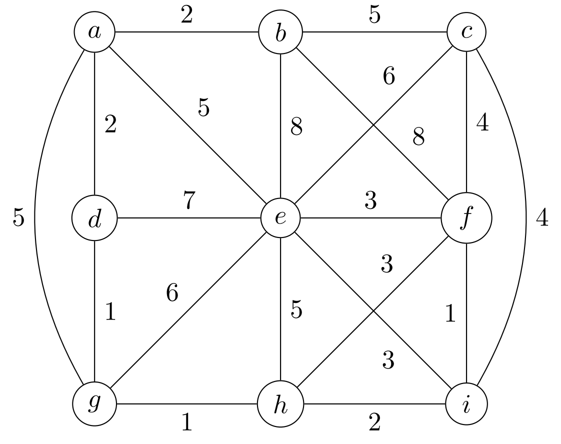
   (g) Which sorts have restrictions on the data stored in the array and what are those restrictions?

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.



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



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.

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

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

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);
};
```

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.

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
```