Data Structures and Algorithms

Final Exam Solution to the exam on 30/12/2019

(This solution contains the full version of all programming questions. The solutions from students need not to contain the full code like this, students just write code directly for solving the question; students are not required to write code to main function, for creating graphs, and so on)

1)

Solution:

Array following sequence:

index	0	1	2	3	4	5	6	7	8	9
value	97	1	77	14	31	60	50	88	18	44

1.a) Perform Shellsort

With k = 3; Segment 1;

index	0	1	2	3	4	5	6	7	8	9
value	97			14			50			44

=> Sort Segment 1

index	0	1	2	3	4	5	6	7	8	9
step 1	97			14			50			44
step 2	14			97			50			44
step 3	14			50			97			44
step 4	14			44			50			97

With k = 3; Segment 2;

index	0	1	2	3	4	5	6	7	8	9
value		1			31			88		

=> Sort Segment 2

index	0	1	2	3	4	5	6	7	8	9
step 1		1			31			88		
step 2		1			31			88		
step 3		1			31			88		

With k = 3; Segment 3;

index	0	1	2	3	4	5	6	7	8	9
value			77			60			18	

=> Sort Segment 3

index	0	1	2	3	4	5	6	7	8	9
step 1			77			60			18	
step 2			60			77			18	
step 3			18			60			77	

=> Combine Segments 1, 2, 3

index	0	1	2	3	4	5	6	7	8	9
value	14	1	18	44	31	60	50	88	77	97

With k = 1; segment 1;

index	0	1	2	3	4	5	6	7	8	9
value	14	1	18	44	31	60	50	88	77	97

=> Sort Segment 1

index	0	1	2	3	4	5	6	7	8	9
step 1	14	1	18	44	31	60	50	88	77	97
step 2	1	14	18	44	31	60	50	88	77	97
step 3	1	14	18	44	31	60	50	88	77	97
step 4	1	14	18	44	31	60	50	88	77	97
step 5	1	14	18	31	44	60	50	88	77	97

step 6	1	14	18	31	44	60	50	88	77	97
step 7	1	14	18	31	44	50	60	88	77	97
step 8	1	14	18	31	44	50	60	88	77	97
step 9	1	14	18	31	44	50	60	77	88	97
step 10	1	14	18	31	44	50	60	77	88	97

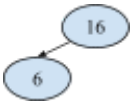
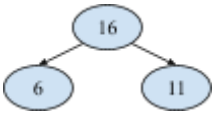
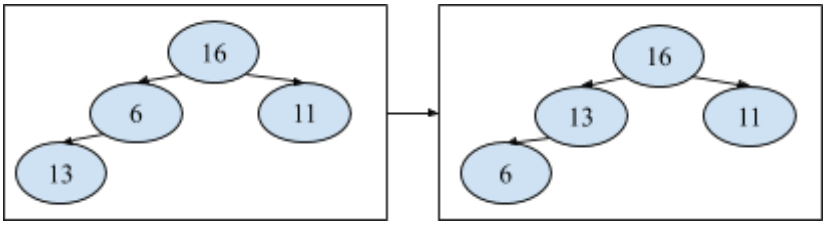
Final Result

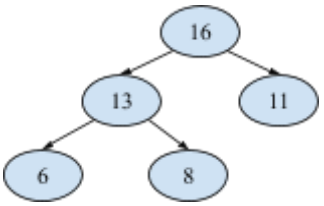
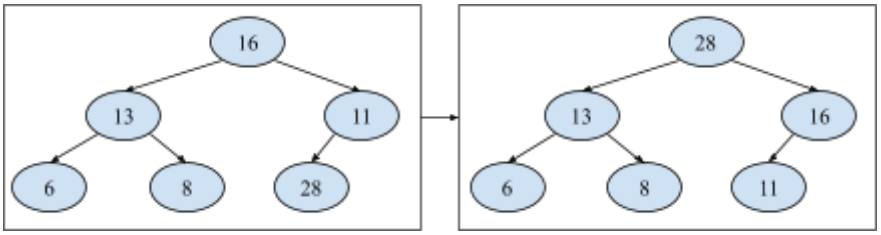
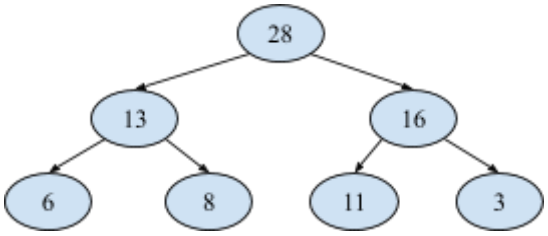
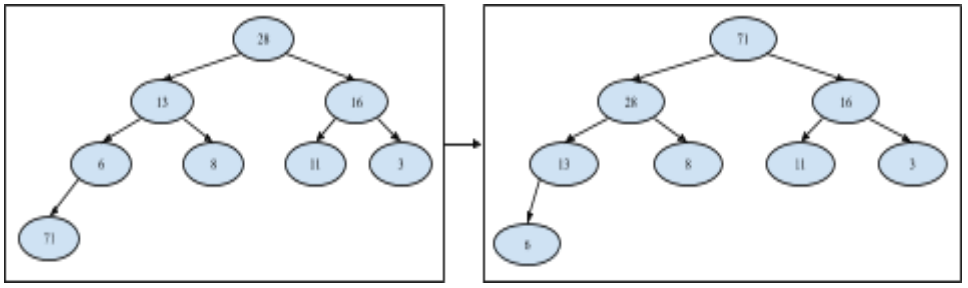
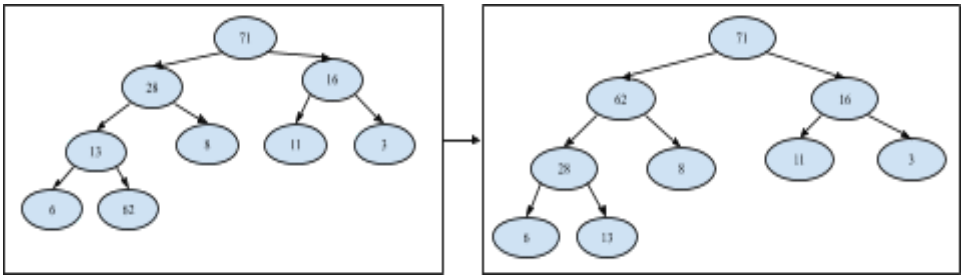
index	0	1	2	3	4	5	6	7	8	9
value	1	14	18	31	44	50	60	77	88	97

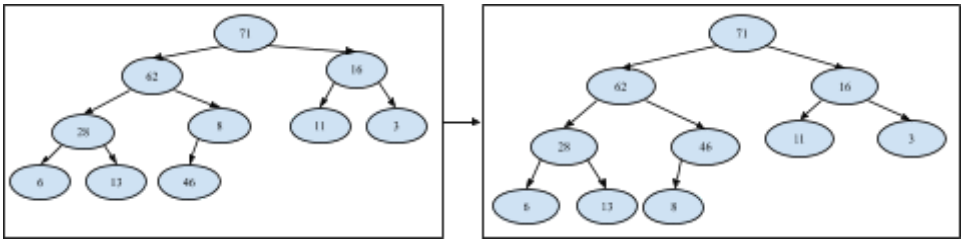
Note:

- Missing detailed steps: -50%.

1.b) Build maxHeap

Step	Build Tree
Insert 16	
Insert 6	
Insert 11	
Insert 13 and Up Heap	

Insert 8	
Insert 28 and Upheap	
Insert 3	
Insert 71 and Upheap	
Insert 62 and Upheap	

Insert 46 and Uphead	

Note:

- Tree or array solution is acceptable.
- Present detailed steps

1.c) Build hash table:

With h(k) = k MOD 9;
And use quadratic probing => hp(k, i) = (h(k) + i^2) mod 9

Value	83	46	20	30	40	58	38		
Index	0	1	2	3	4	5	6	7	8

Key (k)	addresses = h(k)	probe number (p)	addresses = h(k, p) (in case of collision)
20	2	0	
30	3	0	
40	4	0	
46	1	0	
38	2	0	
		1	3
		2	6
58	4	0	
		1	5
83	2	0	

		1	3
		2	6
		3	2
		4	0

Note:

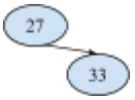
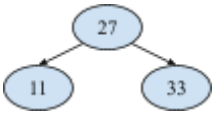
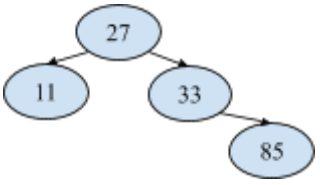
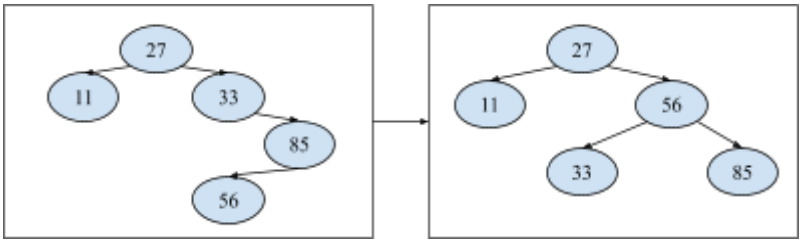
- Present detailed steps

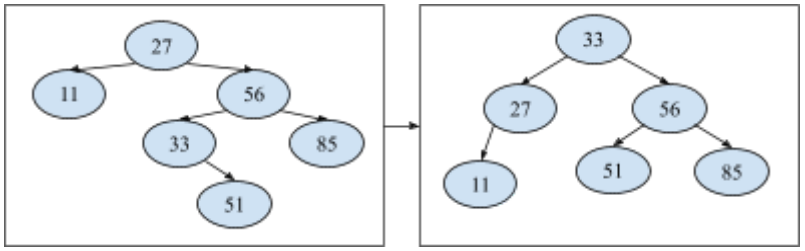
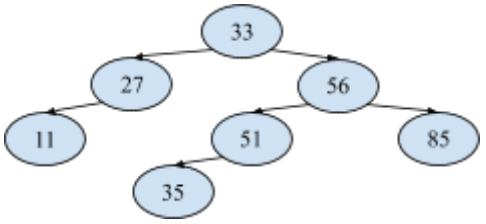
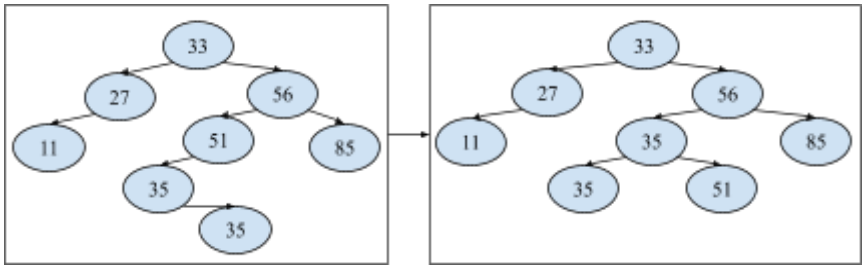
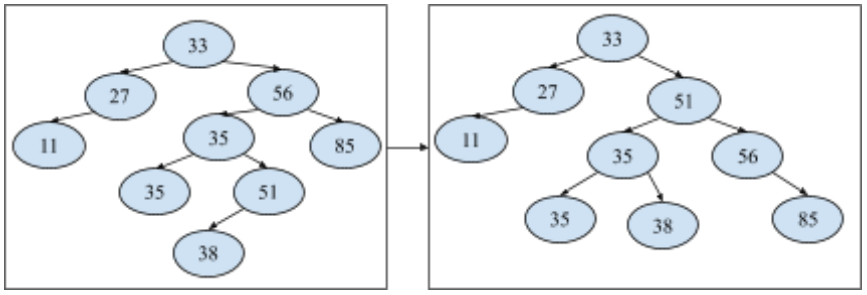
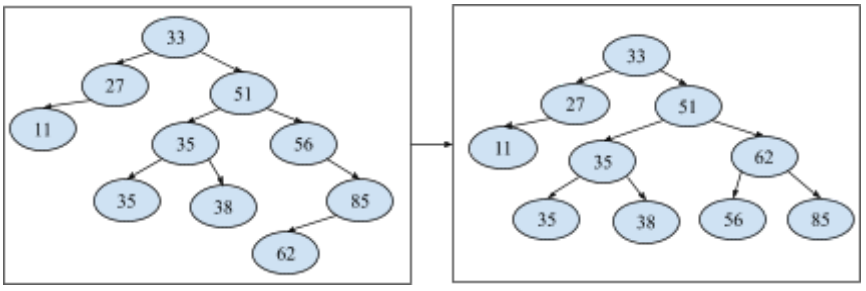
2)

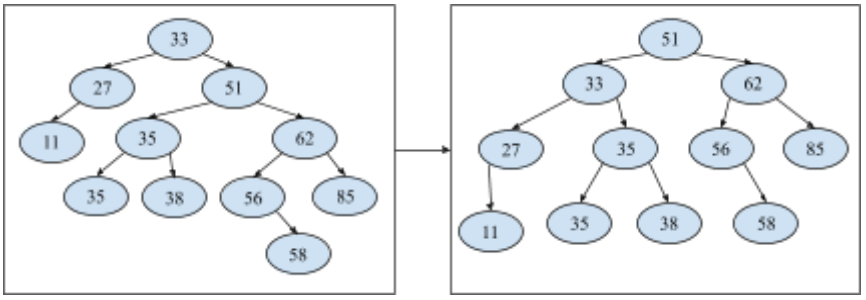
2.a)

Solution:

With Student ID: **1513656** we have: **xx = ef = 56**; **yy = ce = 35**;

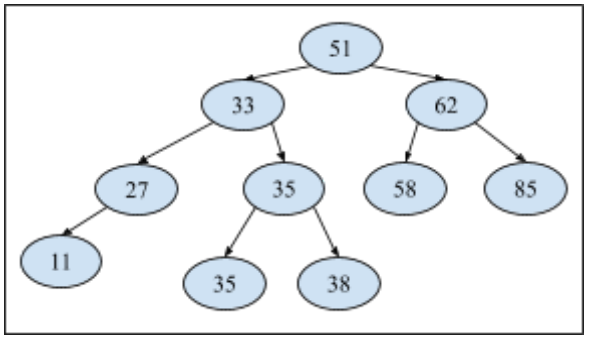
Step	Build Tree
Insert 27	
Insert 33	
Insert 11	
Insert 85	
Insert xx = 56 (In case LoR) Rotate Right at 85 then Rotate Left 33	

Insert 51 (In case LoR) Rotate Right at 56 then Rotate Left 27	
Insert 35	
Insert yy = 35 (In case LoL) Rotate Right at 51	
Insert 38 (In case RoL) Rotate Left at 35 (above) then Rotate Right at 56	
Insert 62 (In case LoR) Rotate Right at 85 then Rotate Left at 56	

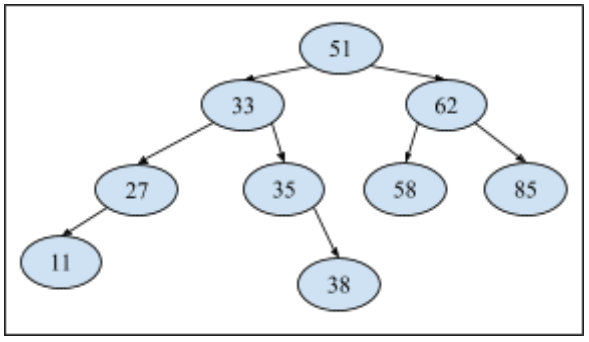
Insert 58 (In case RoR) Rotate Left at 33	

2.b)

Remove xx = 56



Remove yy = 35



3.

a & b.

Solution:

```
#include <iostream>
#include <iomanip>
#include <list>
#include <vector>
#include <string>
```

```cpp
#include <stack>
using namespace std;

// 3a.
class GraphNode
{
public:
   string vertex;
   list<string> adjVertex;
   list<int> adjWeight;
   int inDeg;
   int outDeg;
   bool isMarked;
   GraphNode(string vertex){
       this->vertex = vertex;
       this->inDeg = 0;
       this->outDeg = 0;
       this->isMarked = false;
   }
   void mark(){
       this->isMarked = true;
   }
   void unmark(){
       this->isMarked = false;
   }
   bool searchAdjvertex(string vertex_to){
       bool found = false;

       std::list<string>::const_iterator iterator;
       for (   iterator = adjVertex.begin();
               iterator != adjVertex.end();
               ++iterator) {
           string node_to = *iterator;
           if (node_to.compare(vertex_to) == 0){
               found= true;
               break;
           }
       }
       return found;
   }
```

```
    bool addAdjVertex(string vertex_to, int weight){
        bool found = this->searchAdjvertex(vertex_to);
        if(found)
            return false; //edge is existing
        //add new
        this->adjVertex.push_back(vertex_to);
        this->adjWeight.push_back(weight);
        this->outDeg++;
        return true;
    }
    void println(){
        cout << vertex << endl;
        cout << "\t" << "in degree =" << inDeg << endl;
        cout << "\t" << "out degree =" << outDeg << endl;
        cout << "\tto nodes:" << endl;
        cout << "\t\t";
        std::list<string>::const_iterator iterator;
        int idx = 0;

        std::list<int>::const_iterator iter_weight =
this->adjWeight.begin();
        for (   iterator = adjVertex.begin();
                iterator != adjVertex.end();
                ++iterator) {
            string node_to = *iterator;
            int w = *iter_weight;
            cout << node_to << " (w=" << w << ")" << " , ";
            iter_weight++;
        }
        cout << endl;
    }
};

// 3a.
class DiGraph
{
public:
    vector<GraphNode> node_list;
    void unmark(){
        for (int idx = 0; idx < this->node_list.size(); idx++) {
```

```
        GraphNode* pNode = (GraphNode*)(&(this->node_list[idx]));
        pNode->unmark();
    }
}

// 3b.
void dfs(string vertex_start){
    unmark();
    GraphNode* pStart = this->search(vertex_start);
    if(pStart == NULL){
        cout << vertex_start << " does not exist!" << endl;
        return;
    }
    //vertex_start exists
    stack<GraphNode*> fringe;
    //(0) push the start to fringe
    fringe.push(pStart);
    pStart->mark();
    while(!fringe.empty()){
        //(1) get the top node on the stack (called current)
        GraphNode* pCurrent = fringe.top();
        fringe.pop();
        //(2)process current node
        cout << pCurrent->vertex << " ";
        //(3)push all children of the current node to stack
        std::list<string>::const_iterator iterator;
        for (   iterator = pCurrent->adjVertex.begin();
                iterator != pCurrent->adjVertex.end();
                ++iterator) {
            string node_to = *iterator;
            GraphNode* pChild = this->search(node_to);
            if (pChild->isMarked == false){
                pChild->mark();
                fringe.push(pChild);
            }
        } //for
    }//while
    cout << endl;
}//dfs
```

```
/**
 * return NULL : not found
 */
GraphNode* search(string vertex){
    GraphNode* pNode = NULL;
    bool found = false;
    for (int idx = 0; idx < this->node_list.size(); idx++) {
        pNode = (GraphNode*)(&(this->node_list[idx]));
        if(pNode->vertex.compare(vertex) == 0){
            found = true;
            break;
        }
    }
    if(found)
        return pNode;
    else
        return NULL;
}
bool insert_vertex(string vertex){
    GraphNode* pNode = this->search(vertex);
    if(pNode != NULL)
        return false; //duplicate
    //insert
    GraphNode node(vertex);
    this->node_list.push_back(node);
    return true;
}
bool insert_edge(string vertex_from, string vertex_to, int weight){
    GraphNode* pNode_from = this->search(vertex_from);
    GraphNode* pNode_to = this->search(vertex_to);
    if ((pNode_from == NULL) || (pNode_to == NULL)){
        return false; //vertex is not found
    }
    bool added = pNode_from->addAdjVertex(vertex_to, weight);
    if(added){
        pNode_to->inDeg++;
        return true;
    }
    else{
        return false;
```

```
        }
    }
    /*
     print the graph
     */
    void println(){
        if (node_list.size() <= 0){
            cout << "The graph is empty!";
            return;
        }
        cout << "Num of nodes:" << this->node_list.size() << endl;
        for(int idx=0; idx < node_list.size(); idx++){
            GraphNode &node = node_list[idx];
            node.println();
        }

    }
};

int main(int argc, char** argv) {
    DiGraph graph;
    graph.insert_vertex("1");
    graph.insert_vertex("2");
    graph.insert_vertex("3");
    graph.insert_vertex("4");
    graph.insert_vertex("5");
    graph.insert_vertex("6");
    graph.insert_vertex("7");
    graph.insert_vertex("8");
    graph.insert_vertex("9");
    graph.insert_edge("1", "2", 8);
    graph.insert_edge("1", "3", 3);
    graph.insert_edge("1", "6", 13);
    graph.insert_edge("2", "3", 2);
    graph.insert_edge("2", "4", 1);
    graph.insert_edge("3", "2", 3);
    graph.insert_edge("3", "4", 9);
    graph.insert_edge("3", "5", 2);
    graph.insert_edge("4", "5", 4);
    graph.insert_edge("4", "8", 2);
```

```
    graph.insert_edge("4", "7", 6);
    graph.insert_edge("5", "1", 5);
    graph.insert_edge("5", "4", 6);
    graph.insert_edge("5", "6", 5);
    graph.insert_edge("5", "9", 4);
    graph.insert_edge("6", "7", 1);
    graph.insert_edge("6", "9", 7);
    graph.insert_edge("7", "5", 3);
    graph.insert_edge("7", "8", 4);
    graph.insert_edge("8", "9", 1);
    graph.insert_edge("9", "7", 5);
    graph.println();
    cout << "Visit nodes in graph:" << endl;
    graph.dfs("1");
    return 0;
}
```

Note:

- Must implement in C++ language. Prototype: -50%

4.

4.a)

Solution:

```
#include <iostream>
using namespace std;

class ProbDist
{
public:
    int nrows, ncols;
    float* pWeight;
    ProbDist(int N, int F){
        this->nrows = N;
        this->ncols = F;
        this->pWeight = new float[N*F];
        for (int k=0; k<N*F; k++) {
            this->pWeight[k] = 0;
        }
    }
    ~ProbDist(){
```

```
        if (pWeight != NULL) delete []pWeight;
    }
    int idx(int n, int f) {
        return n*this->ncols + f;
    }
    void set(int n, int f, float prob) {
        pWeight[idx(n, f)] = prob;
    }
    float get(int n, int f) {
        return pWeight[idx(n, f)];
    }
    void print() {
        for (int r=0; r<this->nrows; r++) {
            for (int c=0; c<this->ncols; c++) {
                cout << pWeight[idx(r, c)] << "\t";
            }
            cout << endl;
        }
    }
};

void print_arr(float* t, int size) {
    for(int s=0; s<size; s++) {
        cout << t[s] << endl;
    }
}

int main(int argc, char** argv) {
    int N = 3;
    int F = 3;
    ProbDist* x = new ProbDist(N, F);
    x->set(0, 0, 0.3);
    x->set(0, 1, 0.1);
    x->set(0, 2, 0.6);
    x->set(1, 0, 0.1);
    x->set(1, 1, 0.4);
    x->set(1, 2, 0.5);
    x->set(2, 0, 0.5);
    x->set(2, 1, 0.3);
    x->set(2, 2, 0.2);
    x->print();
```

```
    float* t = new float[N];
    t[0] = 0;
    t[1] = 2;
    t[2] = 1;
    print_arr(t, N);

    return 0;
}
```

4.b)

Solution:

```
class ProbDist
{
    ...
    float calculateCrossEntropy(float* T) {
        float result = 0;
        for(int row=0; row<this->nrows; row++) {
            int col = T[row];
            float prob = this->pWeight[idx(row, col)];
            result += log(prob);
        }
        return -result/this->nrows;
    }
};

...

int main(int argc, char** argv) {
    ...
    cout << "cross-entropy(X,T):" << endl;
    cout << x->calculateCrossEntropy(t) << endl;
    return 0;
}
```

Note:

- Must implement in C++ language. Prototype: -50%

5)

a & b.

Solution:

```
#include <iostream>
#include <vector>

using namespace std;

// 5a
class Task
{
public:
    Task(float cost)
    {
        this->cost = cost;
    }

    ~Task() = default;

    float value()
    {
        return this->cost;
    }

private:
    float cost;
    string name;
};

// 5a
class TaskPool
{
public:
    TaskPool() = default;
    TaskPool(vector<Task *> tasks)
    {
        //copy the data passed as parameters
        for (size_t i = 0; i < tasks.size(); i++)
        {
            this->data.push_back(tasks[i]);
        }
```

```
}

~TaskPool()
{
    while (!this->data.empty())
    {
        auto task = this->data.back();
        delete task;
        this->data.pop_back();
    }
}

// 5a
void reheapUp(int position)
{
    if (position > 0)
    {
        int parent = int((position - 1) / 2);
        if (data[position]->value() > data[parent]->value())
        {
            this->swap(position, parent);
            this->reheapUp(parent);
        }
    }
}

// 5a
void add(Task *task)
{
    this->data.push_back(task);
    this->reheapUp(this->data.size() - 1);
}

void buildHeap(vector<float> costs)
{
    //(1) copy the data passed as parameters
    for (size_t i = 0; i < costs.size(); i++)
    {
        this->data.push_back(new Task(costs[i]));
    }
```

```
        //(2) Build heap
        int walker = 1;
        while (walker < this->data.size())
        {
            this->reheapUp(walker);
            walker += 1;
        }
    }

    void display()
    {
        cout << "Max heap:" << endl;
        cout << "Item: ";
        for (size_t i = 0; i < this->data.size(); i++)
        {
            cout << this->data[i]->value() << " ";
        }
        cout << endl;
    }

    // 5b
    void reheapDown(int position)
    {
        int leftChild = position * 2 + 1;
        int rightChild = position * 2 + 2;
        int lastPosition = data.size() - 1;

        // cout << leftChild << " " << rightChild << " " << lastPosition
<< endl;
        if (leftChild <= lastPosition)
        {
            int largeChild = 0;

            if (rightChild <= lastPosition &&
                data[rightChild]->value() > data[leftChild]->value())
            {
                largeChild = rightChild;
            }
            else
```

```
            {
                largeChild = leftChild;
            }

            if (data[largeChild]->value() > data[position]->value())
            {
                this->swap(largeChild, position);
                this->reheapDown(largeChild);
            }
        }
    }

    // 5b
    Task *next_task()
    {
        Task *res = nullptr;

        if (!data.empty())
        {
            // get root
            res = data[0];

            // move the last node to root then reheap down
            data[0] = data.back();
            data.pop_back();

            this->reheapDown(0);
        }

        return res;
    }

private:
    void swap(int a, int b)
    {
        Task *temp = data[a];
        data[a] = data[b];
        data[b] = temp;
    }
```

```
    vector<Task *> data;
};

int main()
{
    TaskPool heap;
    vector<float> data{8, 19, 23, 32, 45, 56, 78};
    heap.buildHeap(data);
    heap.display();

    Task *next = heap.next_task();

    while (next)
    {
        heap.display();
        cout << next->value() << endl;
        delete next;
        next = heap.next_task();
    }

    return 0;
}
```

Note:

- 5a. Not using a max heap: -50%
- 5b. Implement reheap-down.
- Must implement in C++ language. Prototype: -50%

6)
a, b & c)
Solution:

```
#include <iostream>
#include <vector>
#include <list>

using namespace std;

typedef struct
{
    string name;
```

```
    long long nposts;
} Person;

// 6a
class FriendNode
{
public:
    Person vertex;
    list<FriendNode*> adjVertex;
    int degree;
    bool isMarked;
    // 6a
    FriendNode(string name, long long nposts)
    {
        this->vertex.name = name;
        this->vertex.nposts = nposts;
        this->degree = 0;
        this->isMarked = false;
    }

    void mark()
    {
        this->isMarked = true;
    }

    void unmark()
    {
        this->isMarked = false;
    }

    bool searchAdjVertex(string name)
    {
        bool found = false;

        for (auto const &node : adjVertex)
        {
            if (node->vertex.name.compare(name) == 0)
            {
                found = true;
                break;
```

```
            }
        }

        return found;
    }

    // 6a
    bool addAdjVertex(FriendNode* person)
    {
        bool found = this->searchAdjVertex(person->vertex.name);
        if (found)
            return false; //edge is existing
        //add new

        this->adjVertex.push_back(person);
        this->degree++;

        person->addAdjVertex(this);

        return true;
    }

    void println()
    {
        cout << vertex.name << endl;
        cout << "\t"
             << "degree =" << degree << endl;
        cout << "\tto nodes:" << endl;
        cout << "\t\t";

        for (auto const &node : adjVertex)
        {
            cout << node->vertex.name << " , ";
        }
        cout << endl;
    }
};

// 6b
```

```
bool query(list<FriendNode*> graph, vector<Person*> &ret, long long
nfriends)
{
   for (auto node : graph)
   {
       if (node->degree >= nfriends)
       {
           ret.push_back(&node->vertex);
       }
   }

   if (!ret.empty())
   {
       return true;
   }
   else
   {
       return false;
   }

}

template<typename T>
class AVLNode
{
public:
   AVLNode<T>(T d)
   {
       data = d;
       left = nullptr;
       right = nullptr;
   }

   T data;
   AVLNode* left;
   AVLNode* right;
};


class PostTree
```

```
{
public:

    PostTree(FriendNode* graphNode)
    {
        root = new AVLNode<Person*>(&graphNode->vertex);
    }

    ~PostTree()
    {
        delTree(root);
    }

    void addNode(AVLNode<Person*> *parent, FriendNode* graphNode, bool
isLeft)
    {
        if (isLeft)
        {
            parent->left = new AVLNode<Person*>(&graphNode->vertex);
        }
        else
        {
            parent->right = new AVLNode<Person*>(&graphNode->vertex);
        }
    }

    // 6c
    bool find(AVLNode<Person*> *pR, int npost_b, int npost_e,
vector<Person *> &ret)
    {
        if (pR)
        {
            if (pR->data->nposts >= npost_b)
                find(pR->left, npost_b, npost_e, ret);
            if (pR->data->nposts >= npost_b && pR->data->nposts <=
npost_e)
                ret.push_back(pR->data);
            if (pR->data->nposts <= npost_e)
                find(pR->right, npost_b, npost_e, ret);
        }
```

```
        if (!ret.empty())
            return true;
        else
            return false;
    }

    AVLNode<Person*> *root;

private:
    void delTree(AVLNode<Person*> *pR)
    {
        if (pR)
        {
                delTree(pR->left);
                delTree(pR->right);

                delete pR;
        }
    }
};

int main()
{
    FriendNode* a = new FriendNode("A", 5);
    FriendNode* b = new FriendNode("B", 10);
    FriendNode* c = new FriendNode("C", 4);
    FriendNode* d = new FriendNode("D", 11);
    FriendNode* e = new FriendNode("E", 15);
    FriendNode* f = new FriendNode("F", 25);
    FriendNode* g = new FriendNode("G", 12);
    FriendNode* h = new FriendNode("H", 30);
    FriendNode* i = new FriendNode("I", 14);

    list<FriendNode*> friendGraph{a, b, c, d, e, f, g, h, i};

    a->addAdjVertex(b);
    a->addAdjVertex(c);
    b->addAdjVertex(d);
    b->addAdjVertex(e);
```

```
c->addAdjVertex(e);
c->addAdjVertex(d);
d->addAdjVertex(f);
e->addAdjVertex(f);
f->addAdjVertex(g);
f->addAdjVertex(h);
g->addAdjVertex(i);
h->addAdjVertex(i);

vector<Person*> ret;
// B, C, D, E, F
cout << "Person has more than 3 friend:" << endl;
if (query(friendGraph, ret, 3))
{
    for(auto person : ret)
    {
        cout << person->name << endl;
    }
}

PostTree ptree(d);

ptree.addNode(ptree.root, a, true);
ptree.addNode(ptree.root, e, false);

ptree.addNode(ptree.root->left, c, true);
ptree.addNode(ptree.root->left, b, false);

ptree.addNode(ptree.root->right, g, true);
ptree.addNode(ptree.root->right, f, false);

ptree.addNode(ptree.root->right->left, i, false);
ptree.addNode(ptree.root->right->right, h, false);

ret.clear();
// B, D, G, I, E
cout << "Person has number of posts from 10 to 20:" << endl;
if (ptree.find(ptree.root, 10, 20, ret))
{
    for(auto person : ret)
```

```
        {
            cout << person->name << " " << person->nposts << endl;
        }
    }

    while(!friendGraph.empty())
    {
        auto node = friendGraph.back();
        delete node;
        friendGraph.pop_back();
    }

    return 0;
}
```

Note:

- 6a. Undirected graph or 2d array of True & False. Otherwise: -50%.
- 6b. Must implement in C++ language. Prototype: -50%.