EECS 560 Fall 2021 Final Exam

Name: ________________________ **KU ID:**________________________

The exam is closed-book and closed-notes.

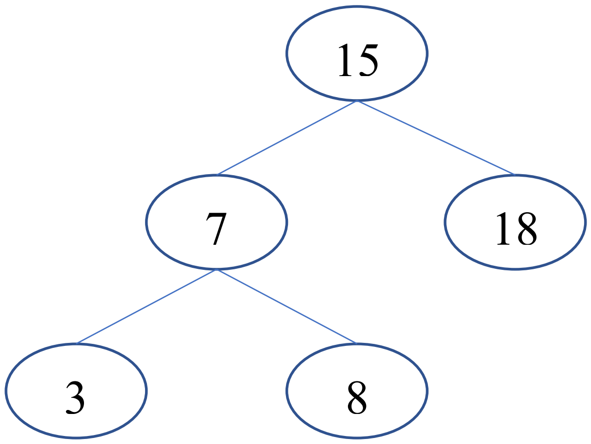
(1) Describe two essential requirements for hash function design. (10pts)

A: The main objectives of designing a hash function:
1) evenly distribute the records (5pts)
2) easy to compute (associate with a smaller constant even in O(1) time) (5pts)

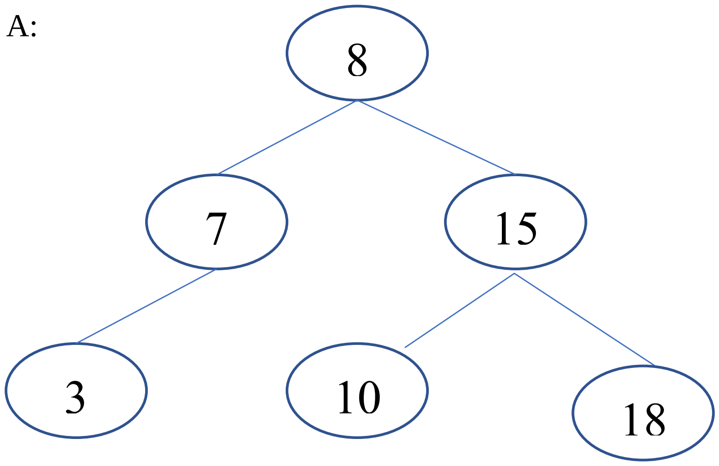
(2) Recall AVL tree.
What is the AVL property? (2pts)

A: The AVL property: for each node, the height of its left subtree and the height of its right subtree cannot differ by more than 1 (2pts)

Draw the resulted AVL tree after inserting 10 into the following AVL tree. (8pts)



A:



(8pts)

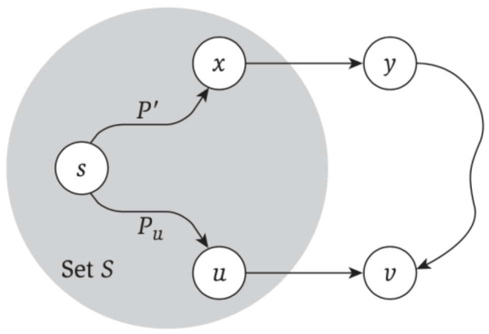
(3) Give the pseudocode of the Dijkstra's algorithm for finding the shortest path from an edge-weighted graph that contains no negative edge. Let the source node be s, and your algorithm should find the shortest path between s and all other nodes in the graph. (10pts)

Pseudocode of the Dijkstra's algorithm:

```
void Graph::dijkstra( Vertex s )
{
    for each Vertex v
    {                                                   2pts
        v.dist = INFINITY;
        v.known = false;
    }
    s.dist = 0;
    while( there is an unknown distance vertex ) {
        Vertex v = smallest unknown distance vertex;
        v.known = true;                                         3pts
        for each Vertex w adjacent to v {
            if( !w.known )
            {
                DistType cvw = cost of edge from v to w;
                if( v.dist + cvw < w.dist )
                {
                    // Update w                                 5pts
                    decrease( w.dist to v.dist + cvw );
                    w.path = v;
                }
        }
}
```

(4) Prove that the Dijkstra's algorithm is correct when assuming no negative edge (7pts). And give a counter example that the algorithm will not work if the graph contains negative edges (3pts).

a) Dijkstra's algorithm is correct when assuming no negative edge (7pts)



a) Consider the above example

1. The set S (gray area) represents the set of vertices that have been visited

2. Except the edges (u, v) and (x, y), all curvy lines can be considered as some paths (not necessarily an edge)

3. The Dijkstra's algorithm simply says that, if $dv = du + lu,v$ is smaller than $dy = dx + lx,y$, then

4. We should include v into S and dv will guarantee to be the shortest path length between s and v

5. If we do not allow negative edge (and path), then the claim will hold . Recall that $dy > dv$

6. If there exists no negative path, then P must be positive (a slight misuse of $P\ y,v$ to indicate the length of the path)

7. if follows that $dy + P\ y,v > dv$ because vertices x and y are chosen arbitrarily, it follows that dv is the shortest path in all cases

Note:
4 pts for lines 5 to 7.
3 pts for 1 to 4

b.Dijkstra’s algorithm will not work for negative edges: If we allow negative edges or paths (note that we do not allow negative cycle), we can simply make up a counter-example to disprove the claim (3pts)

1. let dy be 10 and dv be 7
2. let the path P have a length of -5
3. The shortest path between s and v goes through y (with a total length of 5), but not going through u; as a result, dv should be 5 instead of 7

Explanation : If the graph has negative edge costs, then Dijkstra’s algorithm does not work. The problem is that once a vertex, *u*, is declared *known*, it is possible that from some other *unknown* vertex, *v*, there is a path back to *u* that is very negative. In such a case, taking a path from *s* to *v* back to *u* is better than going from *s* to *u* without using *v* .*A possible solution* is to add a constant value to each edge cost, thus removing negative edges, calculate a shortest path on the new graph, and then use that result on the original graph. The naive implementation of this strategy does not work because paths with many edges become more weighty than paths with few edges.

(5) Write the pseudocode (NOT the C++ code; it doesn't need to be runnable) for the forest-based implementation of disjoint sets with union-by-size and path compression (8pts). Argue that the union$(r1, r2)$ operation can be done in $O(1)$ time (where $r1, r2$ are the roots of the trees to be merged.) (2pts).

VARIABLE DEFINITIONS: // define the variables (e.g., arrays) need to be used here

//array declaration

```
    vector<int> s;
```

METHOD IMPLEMENTATIONS:

init(S): (1pts) // initialize the disjoint set; where S contains all elements to be stored

//Construct the disjoint sets object.
//num of elements is the initial number of disjoint sets.
//initallizes parent element of all nodes/disjoint set to -1

```
 DisjSets::DisjSets( int numElements ) :
    S{ numElements, - 1 }
```

find(x): // find the root of an arbitrary element x (3pts)

```
find(x):

   if (s[x]<0)
       return x;
  else:
   return find( s[ x ] );
```

returns the root of arbitrary element

union$(r1, r2)$ // merge the two trees whose roots are $r1, r2$, respectively (2pts)

//Union two sets by weight (weight is no. of elements in set)
//Assume r1 and r 2 are distinct roots of two disjoint sets
//in Union by size all nodes are stored with size

//To implement this, we need to keep track of the size of each tree. Since we are really just using an array, we can have the array entry of each root contain the negative Size of its tree.

//Thus, initially the array representation of the tree is all −1s.

//When a union is performed, check the sizes; the new size is the sum of the old.

Union by size(r1, r2):

```
if (r1!= r2) :
        if (|s[r1]| < |s[r2]|)   //compare the size of roots
                s[ r1 ] = r2;       //merge with r1 as new root
                       s.[r2]=r1                     //make r2 to represent r1 as parent
                     s[r1]+=s[r2]      //update size
              else :
                      s[ root2 ] = root1;   //merge with r as new root
                       s[r1]=r2                     //make r1 to represent r2  as parent
                 s[r1]+=s[r2]    //update size
```

union(x, y): // merge the two trees that contains x, y, respectively; x, y are arbitrary elements (2pts)

```
 r1=find(x).     //find set containing x
 r2=find(y).     ////find set containing y

if (r1!= r2) :
        if (|s[r1]| < |s[r2]|)   //compare the size of roots
                s[ root1 ] = root2;       //merge with r1 as new root
                       s[r2]=r1                      //make r2 to represent r1 as parent
                     s[r1]+=s[r2]      //update size
              else :
                      s[ root2 ] = root1;   //merge with r as new root
                       s[r1]=r2                     //make r1 to represent r2  as parent
                 s[r1]+=s[r2]    //update size
```

Note:
4 pts for unionize methods
3 pts for path compression
1pt for initialization

Time complexity analysis of **union**$(r1, r2)$: (2pts)

Time complexity for unionizing two roots is. O(1) because we simply attach one root toother tree which is constant operation.

(6) Continuing from the above question, prove that the find(x) operation is in $O(\log n)$ with union-by-size. Note that x is an arbitrary element contained in the disjoint set. (10pts)

A:

When unions are done by size , each merge will double the size of the tree. Since a tree cannot be larger than n, it follows that the height of three is at most $O(\log n)$ (note that each merge will increase the height of the smaller tree by 1 because its root is attached to a new root) in this case, the find() operation can be done in $O(\log n)$ time on average

Proposition: if unions are done by size, then the find() operation can be performed in $O(\log n)$ on average.

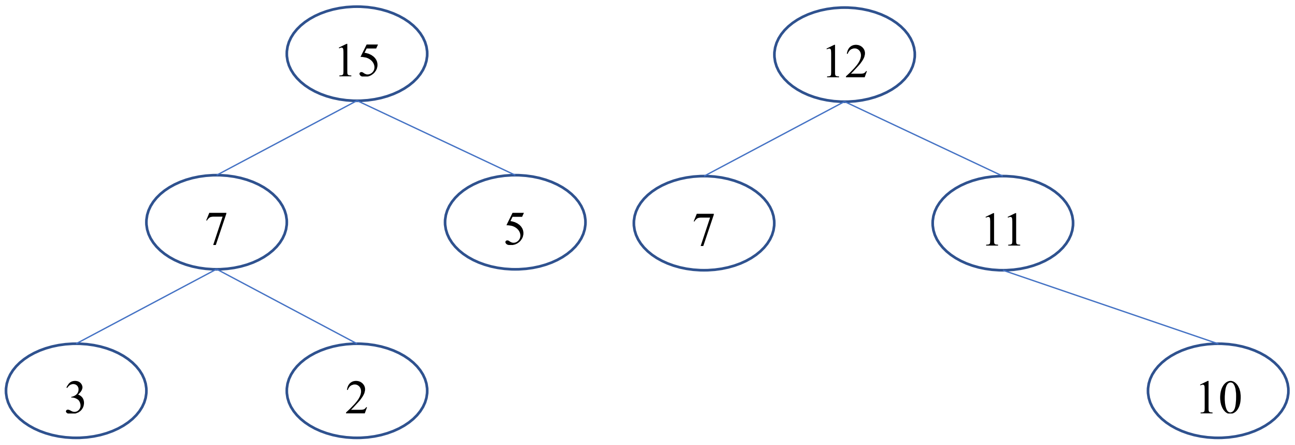
Proof:

1. Initially a set of disjoint sets. (1pt)
2. Because we always merge the smaller set with larger, each merge will at least double the size of the tree (2pts)
3. it follows that the height of three is at most $O(\log n)$ (note that each merge will increase the height of the smaller tree by 1 because its root is attached to a new root) (4pts)
4. In this case, the find() operation can be done in $O(\log n)$ time on average (3pts)

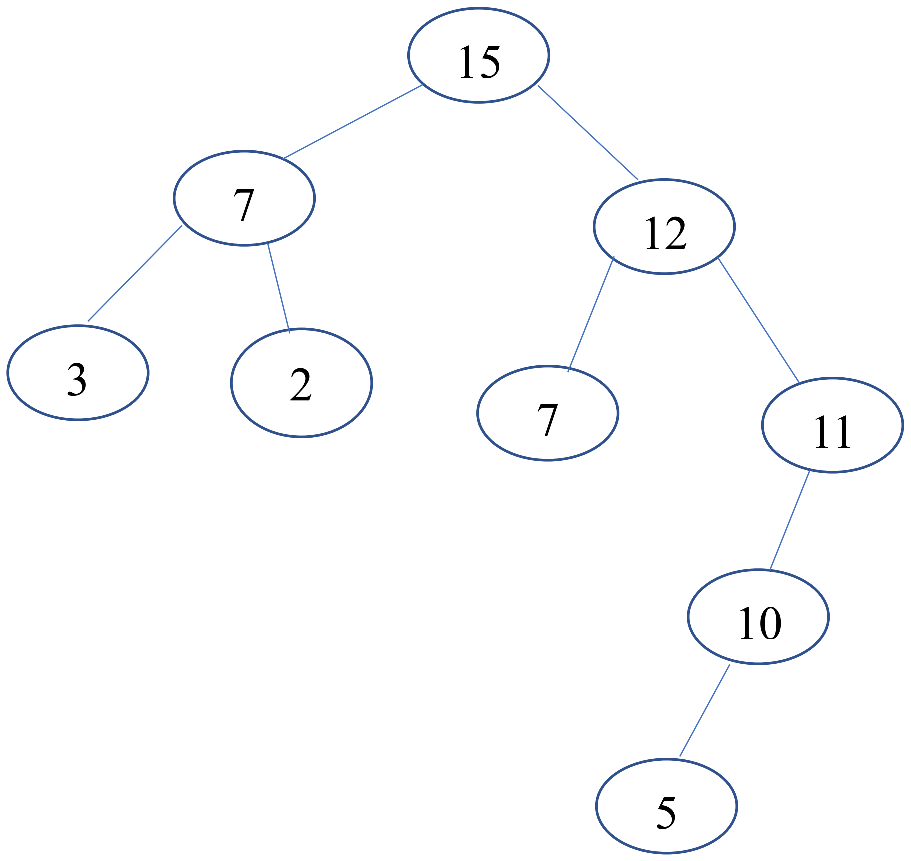
Note:
3pts for points 1 and 2
7pts for points 3 and 4.

(7) Recall the leftist heap. What is the definition of "null path length"? (2pts) Draw the resulted leftist heap from merging the following two leftist heaps (8pts).

We define the null path length, npl(X), of any node X to be the length of the shortest path from X to a node without two children. (2pts)



(8pts)



(8) Recall the adjacency matrix and adjacency list implementation of the graph data structure. What are the time/space complexity of both implementations w.r.t. the following aspects: storage required, adding a new node, adding a new edge, deciding whether an edge exists between two nodes, and enumerating all neighbors of a given node. Use $|V|$ for the number of vertices in the graph, $|E|$ for the number of edges in the graph, and d for the average degree of the graph. (10pts)

STORAGE REQUIRED: (2pts)

Adjacency matrix: O (V^2) 1pt
Adjacency list: O (|V| + |E|) 1pt

ADDING A NEW NODE: (2pts)

Adjacency matrix: O (V^2) 1pt
Adjacency list: O (V) or O (1) 1pt

ADDING A NEW EDGE: (2pts)

Adjacency matrix: O (1) 1pt
Adjacency list: O (1) 1pt

DECIDING WHETHER AN EDGE EXISTS BETWEEN TWO NODES: (2pts)

Adjacency matrix: O (1) 1pt
Adjacency list: O (V) 1pt

ENUMERATING ALL NEIGHBORS OF A GIVEN NODE: (2pts)

Adjacency matrix: O (V) 1pt
Adjacency list: O (d) 1pt

(9) What are “Big Five” defined in the context of C++ object interface? (10pts)

Destructor: reclaim allocated memory to the object. (2pts)
Copy constructor: initialize the object from another object. (2pts)
Move constructor: initialize the object from rvalue. (2pts)
Copy assignment: overwrite the object using another object. (2pts)
Move assignment: overwrite the object using rvalue. (2pts)

(10) Write the pseudocode for depth-first search (DFS) and bread-first search (BFS) algorithms. Let s be the source node. Your algorithms should print out the lists of visited nodes. (5pts for DFS, 5pts for BFS)

DFS(s, g) (5pts)

```
    DFS(s, g):
      let S be stack   1pt
      S.push( s )
      mark s as visited. 1pt
      Print s
      while ( S is not empty):
         v  =  S.top( ) 1pt
         S.pop( )
        for all neighbours w of v in Graph g: 1pt
            if w is not visited :
                     S.push( w )
                    mark w as visited 1pt

or
    DFS(s, g):
        mark s as visited 1pt
        Print s 1pt
        for all neighbours w of s in Graph g: 1pt
            if w is not visited: 1pt
                DFS(g, w) 1pt
```

BFS(s, g) (5pts)

```
BFS (s, g)
      let Q be queue.__1pt
      Q.enqueue( s )
      mark s as visited. 1pt
      Print s
      while (Q is not empty)
          v = Q.dequeue() 1pt
          for all neighbours w of v in Graph g
               if w is not visited  1pt
                        Q.enqueue( w )   1pt
                        mark w as visited.
```

(11) Write the C++ code (expected to be runnable, while minor glitches in syntax are acceptable) for the top(), enqueue(), dequeue() operations and the destructor function of a queue data structure. The data structure is implemented using singly linked list. Note that your implementation should not cause any memory leak nor leaving any uncollected memory blocks upon the termination of the program. (20pts)

Hint:

- Read the comments in the code carefully.
- When a new element is added, you should use new to allocate memory space for it.
- When destructing the data structure, you should make sure the above allocated memory is collected

DEFINITIONS:

```
struct NodeType     {
     int data;            // assuming only store integer type
     NodeType *next;      // pointer to the next element
};

class QueueList     {
  public:
     QueueList()     {     // constructor function
          head = tail = nullptr;
          queue_size = 0;
     }
     ~QueueList();        // destructor function
     NodeType & top(void);
     void enqueue(const NodeType & d);
     void dequeue(void);
  private:
     NodeType *head;      // it does point to real data
     NodeType *tail;      // it does point to real data
     int queue_size;      // the number of elements in the queue
};
```

IMPLEMANTATIONS:

```
#include <cstddef>
#include <iostream>
struct NodeType        {
            int data;                   // assuming only store integer type
            NodeType *next;       // pointer to the next element
};

class QueueList          {
public:
            QueueList()    {          // constructor function
                        head = tail = nullptr;
                        queue_size = 0;
            }
            ~QueueList();             // destructor function
    NodeType& top(void);
            void enqueue(const NodeType& d);
            void dequeue(void);
  private:
            NodeType *head;       // it does point to real data
            NodeType *tail;         // it does point to real data
            int queue_size;          // the number of elements in the queue
};

// returns the first element in the queue
NodeType& QueueList::top(void)// (5pts)
{
    if(head == nullptr)//2pts
    {
        printf("queue is empty");
    }
    else
        return *head; //3pts

}


// add the element d into the queue
void QueueList::enqueue(const NodeType& d) //(5pts)
{
        struct NodeType* nt = new NodeType; // 1pt
        nt->data = d.data;
        if (head == nullptr && tail == nullptr)
        {
            head = tail = nt; //1pt
            queue_size++; //1pt
            return;
        }
        //add newnode in tail->next
        tail->next = nt;//1pt
        //make the new node as the tail node
        tail = nt; //1pt
        queue_size++; //1pt
}

// remove the first element from the queue
void QueueList::dequeue(void)           //(5pts)
{

        if (head == nullptr) //1pt
            prinf("queue is empty");
            return;
        NodeType * temp = head;
        head = head->next; //1pt

        if (head == nullptr) //1pt
            tail = nullptr;
        queue_size--; //1pt
        delete (temp); //1pt
}

// the destructor function
QueueList::~QueueList()         //(5pts)
{

    while (head != nullptr) //1pt
    {
        NodeType *temp=head; //1pt
        head=head->next; //1pt
        delete temp; //1pt
    }
    tail= nullptr;
    queue_size=0; //1pt
}
```