The University of Saskatchewan

Saskatoon, Canada

Department of Computer Science

CMPT 280– Intermediate Data Structures and Algorithms

Assignment 7

Date Due: April 1, 2022, 6:00pm

Total Marks: 45

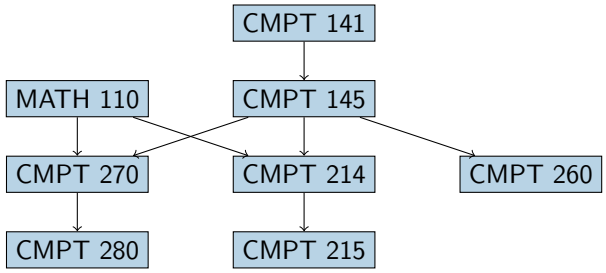
General Instructions

- Assignments must be submitted using Canvas.
- Programs must be written in Java.
- VERY IMPORTANT: Canvas is very fragile when it comes to submitting multiple files. We insist that you package all of the files for all questions for the entire assignment into a ZIP archive file. This can be done with a feature built into the Windows explorer (Windows), or with the zip terminal command (LINUX and Mac). We cannot accept any other archive formats. This means no tar, no gzip, no 7zip, no RAR. Non-zip archives will not be graded. We will not grade assignments if these submission instructions are not followed.

1 Background

1.1 Topological Sort of a Directed Graph

Imagine a directed graph is used to represent prerequisite relationships between university courses. There is a node in the graph for each course, and a directed edge from course A to course B if course A is a prerequisite for course B. Here's what such a graph would look like for some of our courses might look like:



Note that this is a graph and not a tree, because CMPT 270 has more than one "parent". But, there are no cycles in this graph.

If there are no cycles in a directed graph, we can perform an operation called a *topological sort*. A topological sort of a graph produces a linear ordering of the nodes such that if there is a directed edge from node A to node B, then node A appears before node B in the ordering. In the specific case of the course prerequisite graph, a topological sort produces an ordering of the nodes such that no course appears before any of its prerequisites – precisely the thing one needs to know in order to take courses in the right order!

Note that there is not necessarily a unique answer for topological sort. For example, in the graph pictured above, we could take MATH 110 and CMPT 141 in either order (because neither has prerequisites), so both of the following sequences would be valid topological sorts:

- 141, 110, 145, 270, 214, 260, 280, 215
- 110, 141, 145, 270, 214, 260, 280, 215

Also note that 280 and 215 could be taken in either order as long as they are taken after both 270 and 214, resulting in more possible topological orderings:

- 141, 110, 145, 270, 214, 260, 215, 280
- 110, 141, 145, 270, 214, 260, 215, 280

The basic algorithm for topological sort, and a variation that is relevant to the specific problem we will want to solve, is presented with Question 2.

Finally, it is worth noting that you cannot perform a topological sort on a graph that has a cycle. If there is a cycle, then it is not possible to satisfy the prerequisites for any node in the cycle because each course in the cycle is a prerequisite of itself.

Question 1 (30 points):

For this question you will be implementing a k-D tree. We begin with introducing some algorithms that you will need. Then we will present what you must do.

Helper Algorithms for Implementing k-dimensional Trees

As we saw in class, in order to build a k-D tree we need to be able to find the median of a set of elements efficiently. The "j-th smallest element" algorithm will do this for us. If we have an array of n elements, then finding the $n/2$-smallest element is the same as finding the median.

Below is a version of the j-th smallest element algorithm that operates on a subarray of an array specified by offsets *left* and *right* (inclusive). It places at offset j (where $left \leq j \leq right$) the element that belongs at offset j if the subarray were sorted. Moreover, all of the elements in the subarray smaller than that belonging at offset j are placed between offsets *left* and $j-1$ and all of the elements in the subarray larger than that element are placed between offsets $j+1$ and *right*, but there is no guarantee on the ordering of any of these elements! The only element guaranteed to be in its sorted position is the one that belongs at offset j. Thus, if we want to find the median element of a subarray of the array `list` bounded by offsets *left* and *right*, we can call

```
jSmallest(list, left, right, (left+right)/2)
```

The offset $(left + right)/2$ (integer division!) is always the element in the middle of the subarray between offsets *left* and *right* because the average of two numbers is always equal to the number halfway in between them.

The j-smallest algorithm is presented in its entirety on the next page.

```
Algorithm jSmallest(list, left, right, j)
   list - array of comparable elements
   left - offset of start of subarray for which we want the median element
   right - offset of end of subarray for which we want the median element
   j - we want to find the element that belongs at array index j
   To find the median of the subarray between array indices 'left' and 'right',
   pass in j = (right+left)/2.

   Precondition: left <= j <= right
   Precondition: all elements in 'list' are unique (things get messy otherwise!)
   Postcondition: the element x that belongs at offset j, if the subarray were
                  sorted, is at offset j.  Elements in the subarray
                  smaller than x are to the left of offset j and the
                  elements in the subarray larger than x are to the right
                  of offset j.

   if( right > left )
      // Partition the subarray using the last element, list[right], as a pivot.
      // The index of the pivot after partitioning is returned.
      // This is exactly the same partition algorithm used by quicksort.
      pivotIndex := partition(list, left, right)

      // If the pivotIndex is equal to j, then we found the j-th smallest
      // element and it is in the right place!  Yay!

      // If the position j is smaller than the pivot index, we know that
      // the j-th smallest element must be between left, and pivotIndex-1, so
      // recursively look for the j-th smallest element in that subarray:
      if j < pivotIndex
            jSmallest(list, left, pivotIndex-1, j)

      // Otherwise, the position j must be larger than the pivotIndex,
      // so the j-th smallest element must be between pivotIndex+1 and right.
      else if j > pivotIndex
            jSmallest(list, pivotIndex+1, right, j)

      // Otherwise, the pivot ended up at list[j], and the pivot *is* the
      // j-th smallest element and we're done.
```

Notice that there is nothing returned by `jSmallest`, rather, it is the postcondition that is important. The postcondition is simply that the element of the subarray specified by `left` and `right` that belongs at index j if the subarray were sorted is placed at index j and that elements between $left$ and $j-1$ are smaller than the j-th smallest element and the elements between $j+1$ and $right$ are larger than the j-th smallest element. There are no guarantees on ordering of the elements within these parts of the subarray except that they are smaller and larger than the the element at index j, respectively. *This means that if you invoke this algorithm with $j = (right + left)/2$ then you will end up with the median element in the median position of the subarray, all smaller elements to its left (though unordered) and all larger elements to its right (though unordered), which is just what you need to implement the tree-building algorithm!*

NOTE: for this algorithm to work on arrays of `NDPoint280` objects you will need an additional parameter d that specifies which dimension (coordinate) of the points is to be used to compare points.

An advantage of making this algorithm operate on subarrays is that you can use it to build the k-d tree without using any additional storage — your input is just one array of `NDPoint280` objects and you can do all the work without any additional arrays — just work with the correct subarrays.

You may have noticed that `jSmallest` uses the `partition` algorithm partition the elements of the subarray using a pivot. The pseudocode for the `partition` algorithm used by the `jSmallest` algorithm is given below. Note that in your implementation, you will, again, need to add a parameter d to denote which dimension of the n-dimensional points should be used for comparison of `NDPoint280` objects.

```
// Partition a subarray using its last element as a pivot.
Algorithm  partition(list, left, right)
list - array of comparable elements
left - lower limit on subarray to be partitioned
right - upper limit on subarray to be partitioned
Precondition: all elements in 'list' are unique (things get messy otherwise!)
Postcondition: all elements smaller than the pivot appear in the leftmost
               part of the subarray, then the pivot element, followed by
               the elements larger than the pivot.  There is no guarantee
               about the ordering of the elements before and after the
               pivot.
returns the offset at which the pivot element ended up


pivot = list[right]

swapOffset = left
for i = left to right-1
   if( list[i] <= pivot )
      swap list[i] and list[swapOffset]
      swapOffset = swapOffset + 1

swap list[right] and list[swapOffset]
return swapOffset;   // return the offset where the pivot ended up
```

Algorithm for Building the Tree

An algorithm for building a k-d tree from a set of k-dimensional points is given below. It is slightly more detailed than the version given in the lecture slides. It uses the `jSmallest` algorithm presented above.

```
Algorithm kdtree (pointArray, left, right, int depth)
pointArray - array of k-dimensional points
left - offset of start of subarray from which to build a kd-tree
right - offset of end of subarray from which to build a kd-tree
depth - the current depth in the partially built tree - note that the root
        of a tree has depth 0 and the $k$ dimensions of the points
        are numbered 0 through k-1.

if the specified subarray of pointArray is empty
    return null;
else
    // Select axis based on depth so that axis cycles through all
    // valid values. (k is the dimensionality of the tree)
    d = depth mod k;
    medianOffset = (left+right)/2

    // Put the median element in the correct position
    // This call assumes you have added the dimension d parameter
```

```
    // to jSmallest as described earlier.
    jSmallest(pointArray, left, right, d, medianOffset)

    // Create node and construct subtrees
    node = a new kD-tree node
    node.item = pointArray[medianOffset]
    node.leftChild = kdtree(pointArray, left, medianOffet-1, depth+1);
    node.rightChild = kdtree(pointArray medianOffset+1, right, depth+1);
    return node;
```

Your Tasks

Implementing the k-D Tree

Implement a k-D tree. You **must** use the `NDPoint280` class provided in the `lib280.base` package of `lib280-asn6` to represent your k-dimensional points. **You must design and implement both a node class (`KDNode280.java`) and a tree class (`KDTree280.java`).** Other than specific instructions given in this question, the design of these classes is up to you. You may use as much or as little of lib280 as you think is appropriate. You'll be graded in the **actual** appropriateness of your choices. You should aim to make the class fit into lib280 and its hierarchy of data structures, but you should not force things by extending classes inappropriately. You may use whatever private/protected methods you deem necessary.

A portion of the marks for this question will be awarded for the design/modularity/style of the implementation of your class. A portion of the marks for this question will be awarded for acceptable inline and javadoc commenting.

Your k-D tree ADT must support the following operations:

- Construct a new (balanced) k-D tree from a set of k-dimensional points (it must work for *any* $k > 0$).
- Perform a range search: given a pair of points $(a_1, a_2, \ldots a_k)$ and $(b_1, b_2, \ldots, b_k)$, $a_i <= b_i$ for all $i = 1 \ldots k$, return all of the points $(c_1, c_2, \ldots, c_k)$ such that $a_1 \leq c_1 \leq b_1, a_2 \leq c_2 \leq b_2, \ldots, a_k \leq c_k \leq b_k$.

*Note: your tree does **not** have to have operations that insert or remove individual NDPoints.*

In addition, you should write a test program that demonstrates the correctness of your tree. The test program should consist of two parts:

1. Show that your class can correctly build a k-D tree from a set of points. For k=2, display the set of k-dimensional points that you used as input (use between 8 and 12 elements), followed by a graphical representation of the built tree (similar to the `toStringByLevel()` output in the trees we've done previously). Do this again for one other value of k, between 3 and 5 (your choice).
2. For the second of the two trees you displayed in part 1, perform at least three range searches. For each search, display the query range, execute the range search, and then display the list of points in the tree that were found to be in range. A sample test program output is given below.

Implementation and Debugging Strategy

In order to implement the tree-building algorithm `kdtree` (shown in pseudocode, above) you first need to implement `jSmallest` which, in turn requires `partition`. It is **strongly** suggested that you implement and thoroughly test `partition` before trying to implement `jSmallest`. In turn, throughly

test `jSmallest` before you implement `kdtree`. If you don't do this, I can tell you from experience that it will be a nightmare to debug. You need to be sure that each algorithm is correct before implementing the algorithms that depend on it, otherwise, if you run into a bug it will be very hard to determine in which method in the chain of dependent methods the bug is occurring. This is a fundamental principle that is crucial to designing complex software systems. Make sure each piece is correct before relying on it later.

Keep in mind that the algorithms presented above as abstract pseudocode do not necessarily translate line-for-line into Java code. Likely much of the pseudocode you've seen up to this point *does* translate in a line-by-line fashion, but you need to get used to pseudocode that doesn't, as this is the entire point of using pseudocode, such that it is language independent and omits implementation details while still conveying what the algorithm is supposed to do.

Sample Output

```
Input 2D points:
(5.0, 2.0)
(9.0, 10.0)
(11.0, 1.0)
(4.0, 3.0)
(2.0, 12.0)
(3.0, 7.0)
(1.0, 5.0)

The 2D lib280.tree built from these points is:

          3: (9.0, 10.0)
     2: (5.0, 2.0)
          3: (11.0, 1.0)
1: (4.0, 3.0)
          3: (2.0, 12.0)
     2: (3.0, 7.0)
          3: (1.0, 5.0)
Input 3D points:
(1.0, 12.0, 0.0)
(18.0, 1.0, 2.0)
(2.0, 13.0, 16.0)
(7.0, 3.0, 3.0)
(3.0, 7.0, 5.0)
(16.0, 4.0, 4.0)
(4.0, 6.0, 1.0)
(5.0, 5.0, 17.0)

               4: (5.0, 5.0, 17.0)
          3: (16.0, 4.0, 4.0)
               4: -
     2: (7.0, 3.0, 3.0)
          3: (18.0, 1.0, 2.0)
1: (4.0, 6.0, 1.0)
          3: (2.0, 13.0, 16.0)
     2: (1.0, 12.0, 0.0)
          3: (3.0, 7.0, 5.0)
```

```
Looking for points between (0.0, 1.0, 0.0) and (4.0, 6.0, 3.0).
Found:
(4.0, 6.0, 1.0)

Looking for points between (0.0, 1.0, 0.0) and (8.0, 7.0, 4.0).
Found:
(7.0, 3.0, 3.0)
(4.0, 6.0, 1.0)

Looking for points between (0.0, 1.0, 0.0) and (17.0, 9.0, 10.0).
Found:
(16.0, 4.0, 4.0)
(7.0, 3.0, 3.0)
(3.0, 7.0, 5.0)
(4.0, 6.0, 1.0)
```

Question 2 (15 points):

In this question we will once again be looking at a problem related to quests in video games. In many roleplaying video games, one completes quests to gain *experience points*. When one's charater gains enough experience points, their character advances and grows in power, or gains new abilities. Very often there are quests that can only be attempted after one or more prerequisite quests have been completed. We can represent this as a *quest prerequisite graph*, much like the course prerequisite graph in Section 1.1, above, in which each quest is represented by a node, and there is a directed edge from node A to node B if node A's quest is a prerequisite to node B's quest. In this question you will work with just such a graph, and implement a *topological sort* algorithm that will output an ordering in which all of the quests in the graph can be completed without violating prerequisites.

We can perform the topological sort of the graph using the following algorithm:

```
Algorithm TopologicalSort(G)
G is a directed graph.

L = Empty list that will contain the result of the topological sort
S = set of nodes in G with no incoming edges (i.e. indegree 0)
while S is non-empty do
    remove a node n from S
    add n to the end of the list L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m now has no incoming edges (i.e. indegree 0) then
            insert m into S

if the graph has any edges in it then
    throw exception (the graph had at least one cycle!!!)
else
    return L (a topologically sorted order)
```

Keeping in mind that each node of the graph `G` stores information about one quest, we can say that `S` stores the set of quests that are "doable" or "available" (those for which all prerequisites have already been completed) at any particular moment. Being a set, it stores these quests in no particular order. But... we are greedy. When we remove a node from S at the top of the while loop, we always want to remove the quest in S that has the biggest experience point (XP) reward. In this way, we always do the available quest with the largest experience reward first so as to gain experience more quickly. But to do this we have to change this algorithm a little. Fortunately, it's a very easy change. All we have to do is **change S from a set of graph nodes to a heap of quests that are keyed on the quest's XP value**. Then we are guaranteed to always remove from S the available quest with the highest possible experience point reward (because it will always be at the top of the heap!)[1] The modified algorithm is:

[1]Note: although the modified algorithm will get us XP faster than if we used the original algorithm, it will NOT guarantee that we gain the **most** possible XP with the fewest number of quests. To illustrate this, suppose we have Quests 1, 2, 3, and 4 which are wroth 50, 500, 500, and 5000 XP respectively, but, quest 1 is a prerequisite to quest 4. Initially, the available quests are 1, 2, and 3 (4 cannot be done without doing 1 first). Following the algorithm with the "set" replaced by a heap, we would end up with quests 2 and 3 being first in the topological order, because they are both worth more experience than quest 1. Then we would have to do quest 1, and finally quest 4. If we did the quests in this order, over the first 3 quests we would gain 1050 experience. But we **could** have done quest 1 first, and **then** quest 4 immediately, giving us 5050 experience after just two quests. But we didn't realize this because we didn't look ahead... our algorithm is "greedy" in this respect and we took the **available** quest with the best reward first. This is fine and this is what we want to do here, I just don't want you to expect you will always get the "optimum" experience gain from this algorithm.

```
Algorithm TopologicalSort(G)
G is a directed graph.

L = Empty list that will contain the result of the topological sort
H = heap of quests (compared by experience value) whose corresponding
    nodes in G have no incoming edges,

while H is non-empty do
    remove a quest n from H
    add quest n to the end of the list L
    for each graph node m such that there is an edge e from n's graph node to m do
        remove edge e from the graph
        if m now has no incoming edges then
            insert m's quest into H

if the graph has any edges left in it then
    throw exception (the graph had at least one cycle!!!)
else
    return L (a topologically sorted order)
```

You will be provided with the following files as part of an IntelliJ module called `QuestPrerequisites-Template`:

Quest.java A class for storing information about a quest. Note that it is different from the `QuestLogEntry` class from Assignment 4. It contains mostly accessors and mutators for protected instance variables. It also has a `toString()` method and a `compareTo` methods that allow quests to be compared by their experience values. **You may not modify this file.**

QuestVertex.java A class to be used as the vertex class in a graph. It defines a graph node that can store a reference to a quest. You can use the `quest()` method of this graph node to obtain the quest associated with this node. **You may not modify this class.**

QuestProgression.java This class contains a few static methods that together form a program for doing a topological sorting of quests in a "quest prerequisite graph". It contains the following static methods:

- `readQuestFile`**:** Reads a data file containing information about quests and quest prerequisites and builds a quest prerequisite graph. This method is finished, and you don't have to do anything to it. Note that the graph that is created has nodes that are of type `QuestVertex`.
- `hasNoIncomingEdges`**:** An incomplete method that determines whether a given node in a given graph has no incoming edges, i.e., has indegree zero.
- `questProgression`**:** An incomplete method that performs a topological sort of a given quest prerequisite graph.
- `main`**:** The main program that loads the quest data, constructs the graph, performs the topological sort, and displays the result. This method is finished and you don't have to do anything to it.

quests16.txt A data file containing information on 16 quests and their prerequisites which is used by `main()` as the program input.

Your Tasks

You must complete the implementation of the `hasNoIncomingEdges` and `questProgression` methods. It is up to you to inspect the methods available in the `GraphMatrixRep280` class (and it's superclasses!) and use them to solve the problem. Because you are not implementing methods within the graph class, you can only access and modify the graph via its public interface.

Implementation Hints

This program relies on a correspondence between graph nodes and quests. Note that in the graph ADTs, nodes are referred to by integers, starting from 1. Note also that each `Quest` instance contains an ID (you can see this in `Quest.java`). The quest prerequisite graph is constructed by the `readQuestFile` method such that a quest with a particular ID k is always associated with node k of the graph. Thus, if you have a quest object, you can find its corresponding graph node by looking up the node in the graph with the quest's ID. If you have a node ID, you can find its corresponding quest by looking up the node object (of type `QuestVertex`) using its ID, and call the `quest()` method of the resulting vertex object to obtain the quest stored there.

Sample output is shown below.

Sample Output

If you implement the unfinished methods correctly, the output of the program when given quests16.txt as input will be:

```
13, Discover Peppermint Butler's Secrets, Candy Kingdom, XP: 14550
7, Find the Ice King's Wizard Eye, Ice Kingdom, XP: 12000
12, Rescue Marceline from the Nightosphere, The Nightosphere, XP: 25000
10, Win Wizard Battle, Wizard Battle Arena, XP: 29000
9, Rescue Wildberry Princess from the Ice King, Ice Kingdom, XP: 4200
3, Defeat Goliad, Candy Kingdom, XP: 2578
5, Atone for Shoko's Sins, Finn's Treehouse, XP: 2700
4, Gain Approximate Knowledge of Many Things, Demon Cat Lair, XP: 1000
8, Get some pickles from Prismo, Prismo's Home, XP: 150
6, Make an Amazing Sandwich, Finn's Treehouse, XP: 1900
16, Watch what Beemo Does When He Is Alone, Finn's Treehouse, XP: 70
1, Steal a Treat from the Donut Witch's Garden, Blue Plains, XP: 250
14, Defeat the Ice King's Penguin Army, Candy Kingdom, XP: 50000
11, Eat Marceline's Fries, Marceline's House, XP: 50
15, Discover the Ice King's Secret Past, The Past, XP: 3299
2, Recover the Stolen Items from the Door Lord, The Sandylands, XP: 700
```

Another test you can do to check whether your program is working is to make a copy of `quests16.txt` and remove all of the ordered pairs starting on line 18 of the file. If you do this, then the topological sort should result in a list of quests sorted in descending order by their experience value.

2 Files Provided

lib280-asn7: A copy of lib280 which includes:

- solutions to assignment 6;
- the `NDPoint280` class in the `lib280.base` package for representing n-dimensional points for question 1;
- the `GraphMatrixRep280` which you'll use in question 2;
- `QuestPrerequisites-Template` — an IntelliJ module with templates for question 2.

3 What to Hand In

Hand in a ZIP archive containing **only** the following files:

KDNode280.java: The node class for your k-D tree from Question 1.
KDTree280.java: Your k-D tree class for Question 1.
a7q1.txt/doc/pdf: The console output from your test program for question 1, cut and paste from the IntelliJ console window.
QuestProgression.java Your completed `QuestProgression` class from question 2.