CISC 160

Final

**Background**

Priority queues are versatile and complex data structures that can be implemented multiple ways. In this lab, you will do that very thing.

**List-Based Heap Implementations**

In class, we discussed the concept of a priority queue and how to implement it conceptually. Then we walked through the traditional implementation: using an array or Python sequence to represent the heap.

**Data Storage**: In class, our example used a custom object (an \_Item) to store the elements of the priority queue. This object wraps the data but provides almost no support for that stored data. For this section, you will be storing your key and value as an ordered pair (or tuple) of two values where the first element is the key and the second element is the value.

1. What are the inherent benefits and drawbacks of this (array-based) backing representation? Discuss with respect to ease of implementation, efficiency, and memory usage. (PI 1.2/ABET[1], PI 6.1/ABET[6]**)**
2. As we have discussed many times this semester, anything that can be implemented using an array can be implemented using a linked list. Implement an object called LinkedHeapPQ in a file LinkedHeapPQ.py that implements/extends the PriorityQueue\_Interface we designed in PriorityQueue\_Interface.py using a **doubly linked list** as the backing data structure for the **heap**.   
     
   Implement all of the public methods (a constructor, add, min, remove\_min, is\_empty, and the \_\_len\_\_ magic method) with the same functionality as with the ArrayHeap object. This means that the min and remove\_min methods will return a tuple of (key, value). This will be done by **importing the DNode object from the lab page** and manipulating the doubly linked list inside of the LinkedHeapPQ object.  
     
   You must store the head node of the list. You are also permitted to store the tail node and the length of the priority queue as a whole but **no other meta-data**. **NOTE:** Remember that the element of DNode is immutable once data is established. This means that swapping elements must be done by moving references and nodes, not through moving the internal data of the nodes.  
     
   (PI 1.1/ABET[1], PI 1.2/ABET[1], PI 2.2/ABET[2], PI 6.3/ABET[6]**)**
3. What are the inherent benefits and drawbacks of this (linked list-based) backing representation? Discuss with respect to ease of implementation, efficiency, and memory usage in general and as compared to the array-based implementation.  
   (PI 1.2/ABET[1], PI 6.1/ABET[6], PI 6.2/ABET[6])
4. How would this implementation be different if it were implemented with a singly-linked list? Discuss with respect to ease of implementation, efficiency, and memory usage as compared to the doubly linked list. (PI 1.1/ABET[1], PI 1.2/ABET[1], PI 2.2/ABET[2], PI 6.1/ABET[6])

**Linked Tree-Based Heap Implementations**

Heaps are strongly structured binary trees. As we discussed in Module 07, binary trees may be implemented with traditional lists or a more literal linked tree structure.

**Data Storage**: In class, our example used a custom object (an \_Item) to store the elements of the priority queue. This object wraps the data but provides almost no support for that stored data. For this section, you will be storing your key and value as an ordered pair (or tuple) of two values where the first element is the key and the second element is the value.

1. Implement an object called TreeHeapPQ in a file TreeHeapPQ.py that implements/extends the PriorityQueue\_Interface we designed in PriorityQueue\_Interface.py using a linked **binary tree** as the backing data structure for the **heap**.  
     
   Implement all of the public methods (a constructor, add, min, remove\_min, is\_empty, and the \_\_len\_\_ magic method) with the same functionality as with the ArrayHeap object. This means that the min and remove\_min methods will return a tuple of (key, value). This will be done by **importing the BinaryNode object from the lab page** and manipulating the binary tree inside of the TreeHeapPQ object.  
     
   You must store the root node of the list. You are also permitted to store length of the priority queue as a whole but **no other meta-data**. **NOTE:** Remember that the element of Binary\_Node is immutable once data is established. This means that swapping elements must be done by moving references and nodes, not through moving the internal data of the nodes.(PI 1.1/ABET[1], PI 1.2/ABET[1], PI 2.2/ABET[2], PI 6.3/ABET[6])
2. What are the inherent benefits and drawbacks of this (linked tree-based) backing representation? Discuss with respect to ease of implementation, efficiency, and memory usage in general and as compared to an array-based and linked list-based implementation. (PI 1.2/ABET[1], PI 6.1/ABET[6] , PI 6.2/ABET[6])

**“List of List” Implementations**

These implementations are logical however there are other implementations of a priority queue which are simpler to conceptually visualize. These can be thought of as “list of lists” implementations where the first list determines the priority level and the second list determines elements at that priority level. For example, consider the following diagram:

A close up of a clock

Description automatically generated

In this example, the highest priority would be 1 and there would be two elements in the priority queue at that priority level, a and b. There would also be an element in priority level 2, c, and an element in lowest (current) priority level, d.

**Data Storage**: Unlike the previous two sections, you do not necessarily need to store your data as a tuple. In questions 7 & 8, the index will represent your key. In questions 9, 10, & 11, you will store your information within the given structures as **you** specify.

1. Implement an object called TwoDSequencePQ in a file TwoDSequencePQ.py that implements/extends the PriorityQueue\_Interface we designed in PriorityQueue\_Interface.py using a **two-dimensional Python sequence** as the backing data structure for the **non-heaped priority queue**.  
     
   Implement all of the public methods (a constructor, add, min, remove\_min, is\_empty, and the \_\_len\_\_ magic method) with the same functionality as with the ArrayHeap object. This means that the min and remove\_min methods will return a tuple of (key, value). For this implementation, you will consider the index of the main list to be the priority level and then the list remaining as containing the elements at that priority level.  
     
   For example, if your two-dimensional array is named pq then pq[0] will represent all of the elements at priority 0, pq[85] will represent all of the elements at priority 85, and pq[85][2] will represent the third element (if it exists) at priority 85.  
     
   You must store the two-dimensional sequence. You are also permitted to store length of the priority queue as a whole but **no other meta-data**.  
     
   (PI 1.1/ABET[1], PI 1.2/ABET[1], PI 2.2/ABET[2], PI 6.3/ABET[6])
2. What are the inherent benefits and drawbacks of this (two-dimensional sequence) backing representation? Discuss with respect to ease of implementation, efficiency, and memory usage in general and as compared to an array-based, a linked list-based, and a binary tree-based implementation. (PI 1.2/ABET[1], PI 6.1/ABET[6] , PI 6.2/ABET[6])

The behavior can be accomplished with a linked list of linked lists. In this case, the main linked list is responsible for maintaining both the priority and the head node for the associated linked list. This can be accomplished multiple ways, including altering the node object, having multiple types of node objects, or getting creative with how information is stored within the node.

1. You are going to be asked to make a singly-linked list of singly-linked lists. The primary/first dimension of the singly linked list will consist of nodes which **MUST** contain the key of a given level and a reference to the head of the secondary/second dimension of the linked list. This secondary linked list contains a list of singly linked nodes which each contain a single value and a reference to the next value-containing node in the linked list. How are you designing your data structure to best facilitate this design? Be sure to explain how the data is internally stored within your linked list of linked lists. (PI 1.1/ABET[1], PI 6.1/ABET[6])
2. Implement an object called LinkedLinkedPQ in a file LinkedLinkedPQ.py that implements/extends the PriorityQueue\_Interface we designed in PriorityQueue\_Interface.py using a **singly linked list that stores singly linked lists** as the backing data structure for the **non-heaped priority queue**.  
     
   Implement all of the public methods (a constructor, add, min, remove\_min, is\_empty, and the \_\_len\_\_ magic method) with the same functionality as with the ArrayHeap object. This means that the min and remove\_min methods will return a tuple of (key, value). For this implementation, you will have to **use the Node object from the lab page** in some way, but the exact nature of this use depends on how you answered question 9.  
     
   Regardless, you must store the head to the main singly-linked list which contains the heads of all of the subsequent singly-linked lists. You are also permitted to store length of the priority queue as a whole but **no other meta-data**.   
     
   (PI 1.1/ABET[1], PI 1.2/ABET[1], PI 2.2/ABET[2], PI 6.3/ABET[6])
3. What are the inherent benefits and drawbacks of this (essentially two-dimensional singly linked list) backing representation? Discuss with respect to ease of implementation, efficiency, and memory usage in general and as compared to an array-based, a linked list-based, a binary tree-based, and a two dimensional dynamic sequence-based implementation. (PI 1.2/ABET[1], PI 6.1/ABET[6] , PI 6.2/ABET[6])

**Lab Requirements**

Download all source code files as instructed from this book. You may add any additional helper methods as needed. Helper functions should be named with an underscore at the front as they are not designed to be publicly accessible If you add a helper function, be sure to comment it in the same style as the other, included functions.

You may (and I recommend that you do add), the “testing structure” of if \_\_name\_\_ == ‘\_\_main\_\_’ in any code files you wish. You may also have separate testing files. If you do have any extra testing files, I recommend that you include them in submission. I may look at it for understanding of how you are approaching testing but you will not be explicitly graded on code included there. For the same reason, I recommend commenting your code. I will not take off for not having comments but I will if your code doesn’t work and I cannot understand what you did.

Remember: These requirements may not be all encompassing. Use your brain, your knowledge of the system, and any descriptions within the code as sanity checks and reminders to make a complete system.

**Submission:** Include ALL source files (LinkedHeapPQ.py, TreeHeapPQ.py, TwoDSequencePQ.py, and LinkedLinkedPQ.py, and any adjusted node files that you created to make the LinkedLinkedPQ object to work) and this document in the final submission. Answer questions 1, 3, 4, 6, 8, 9, and 11 within this book or a separate file with the answers for each question well labeled.

**Submit your files unzipped to Canvas**. Do not include any of the supporting files that I provided you, such as the any basic Node or Tester files. The original files will be provided and used during testing and your code **MUST** work with the original files.

**Grade**: This project has 110 points available but is out of 100 points. This will be handled as a 10% "fudge factor" on your grade meaning that the first 10% of mistakes do no affect your grade. **If you score more than 100% on the assignment, your grade will be capped at 100%.**

**DEADLINE**: This assignment is due by **8:00 AM ET on Monday, December 11**. Given that I will have less than a week to grade these**, THIS IS A HARD DEADLINE. NO LATE SUBMISSIONS WILL BE ACCEPTED. EVEN 1 SECOND LATE MEANS A ZERO.** I will not begin grading before the deadline, so it would be acceptable to submit a “draft” version early and then attempt to overwrite later. **DO NOT RISK SLOW SERVERS. GET YOUR FINAL IN EARLY.**