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]**)**

The benefits of an array-based implementation are that an array is a concrete data structure. The ease of implementation for this is that it is easy to implement since you can use formulas to find where a child and parent is in the heap, and it is easy to switch the 2 points. It is easy to keep the heap order priority since arrays allow access to all the data and allows you to switch the parent and child data at will. It is also efficient to insert into the array since the implementation has a run time of O(logn) which is faster than O(n). It can be somewhat memory efficient since the memory is allocated at creation, however array-based implementations struggle with memory due to having to resize the array which requires much more memory than other data structures due to having to copy the array every time the array resizes.

1. 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]**)**
2. 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.

The linked-list backing implementation is harder to implement than an array based due to having to keep track of pointers and also the list needing to be iterated through to insert new data being O(n) compared to an arrays O(1). It can be efficient to use a doubly linked list since it doesn’t run into the same issues as a singly linked list due to it using both a previous and next pointer. A drawback of this type of implementation is adding and removing data from the list since the pointers must be changed rather than an index like an array. A linked-list implementation is more memory inefficient than an array-based implementation due to nodes being arbitrarily created in memory which causes a lot more memory usage than an array.  
(PI 1.2/ABET[1], PI 6.1/ABET[6], PI 6.2/ABET[6])

1. 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])

How the implementation of question 2 would be different if a singly linked list was used is that the logic behind the pointers and the way that the search for the parent node would change slightly. For switching the nodes, since there wouldn’t be a get\_previous() command there would need to be variables that saves the node before and after the parent and the new node. These variables would be used to change the get\_next() for the previous node and be able to point the get\_next() for the parent and new node when they switch positions. Also, the way it decides the parent node would be slightly different since it would need to dynamically store the previous node to the current node and the next so when it does get to the current node it is possible to call upon them. The memory usage would be worse since you would need to store more things in memory to be called upon rather than using the get\_previous() command but the ease of implementation would be worse than a doubly linked list since even though there are less pointers to change and keep track of, not being able to call the previous node in the list is a critical flaw due to heaps needing both the child and parent nodes to keep the heap order. However, the efficiency would be worse due to traversal being only in one direction it would be difficult to switch nodes. And in terms of memory usage, both a singly linked list and doubly linked lists are similar due to the nodes being created arbitrarily in memory.

**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])

The benefits of using a linked tree- based backing representation is in its efficiency with all of its operations having a efficiency of O(log n). Depending on the size of the tree this can be more efficient with traversal than a linked-list implementation but can fall behind array-based implementation at higher amounts of data. It is more efficient than an array with insertion at the beginning and middle of the list but can be memory inefficient due to having the same problems as linked lists where the new data is arbitrarily created in memory with takes up more space in memory. Also the ease of implementation can be low due to having to keep the tree sorted and having to keep track of all of the pointers that change when upheap bubbling occurs.

**“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 benefit of using the two-dimensional sequence is that the ease of implementation is better than the other methods of implementation since you don’t have to deal with switching pointers and finding where to insert and inserting the new values into the list is a lot easier. Also, the memory usage is better than using a linked-list or binary tree and similar to a array based method since it on creation it initializes memory rather than arbitrarily create it like linked-lists or binary trees. The efficiency is comparable to an array with searching through the sequence however a linked list or tree is better for insertion.

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. 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])

How I’m going to complete #10 is that since singly linked nodes can’t point forward to 2 separate nodes, they will need to be placed as the elements of nodes in self.\_head. This will make it so self.\_head can have the heads of the linked lists and be able to be a linked list itself. This allows me to store the list of lists in self.\_list and have head be the head of that list and still allows me to sort the lists and keep all of the heads in the main linked list.

1. 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])
2. 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])

The benefits of this backing representation is that it is efficient to add or delete data in linked lists due to the operation of the operations being O(1). The downsides however are in the way that the list needs to be traversed since singly linked lists cannot go to the previous node is it less efficient with traversal than using a doubly linked list or array-based implementation. It can be more difficult to implement due to having to keep track of pointers and the inability to traverse to the previous node in the list. It is also memory inefficient compared to an array or two-dimensional sequence implementation due to linked lists arbitrarily creating nodes in memory.

**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 12**. 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.**