Josue Andaluz

190250

CISC 160-XX

Final Project

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**

The book starts out by discussing the concept of a priority queue and how to implement it conceptually. Then, the first implementation is presented: using an array or Python sequence to represent the heap.

**Data Storage**: The book stores elements of a priority queue within a custom object (an \_Item) which wraps and provides almost no support for the 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 implementation, efficiency, and memory usage. **(PI 1.2/ABET[1])**

Searching and accessing an element is much faster, also its more efficient because the array can index in 0(1) time. The insertion of an element takes much longer since a new array has to be created every time thus having to resize the array. An array is a fixed size so if we need to more memory it is using more memory but if we do not have enough memory than it won’t function. So not too good on memory.

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 that implements a priority queue using a doubly linked list as the backing data structure for the heap. Implement all of the same public functions as the HeapPriorityQueue from the text (a constructor, add, min, remove\_min, is\_empty, and the \_\_len\_\_ magic method) with the same functionality. 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**. **(PI 1.1/ABET[1])**
2. What are the inherent benefits and drawbacks of this (linked list-based) backing representation? Discuss with respect to implementation, efficiency, and memory usage. **(PI 4.1/ABET[3])**

The implementation is a bit more difficult, since we have to implement more functions for example the index, append, set and pop functions. The efficiently drops due to all the moving around of the index which are in 0(n) time. But some benefits are a person can travel in both directions of the list as long as they keep track of the next and previous nodes. Also, the user can reverse the heap from min to max and vice versa if they desire. This does take up more memory due to the previous and next nodes being required.

1. How would this implementation be different if it were implemented with a singly-linked list? Discuss with respect to implementation, efficiency, and memory usage as compared to the doubly linked list. **(PI 1.1/ABET[1])**

Singly linked list would be different from a doubly linked list because we would not be able to move backwards. Only forward. But the lookups would still be done in 0(n) time. So it would be a little less efficient because we would not be able to move back and forth. As far as memory usage it would use less memory because we do not have to store one additional pointer the previous. The implication would be similar they are both linked list just one is able to move backwards and forwards and the other only moving forwards.

**Linked Tree-Based Heap Implementations**

Heaps are strongly structured binary trees. As we discussed in chapter 8, binary trees may be implemented with traditional lists or a more literal linked tree structure.

**Data Storage**: The book stores elements of a priority queue within a custom object (an \_Item) which wraps and provides almost no support for the 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 that implements a priority queue using a linked binary tree as the backing data structure for the heap. Implement all of the same public functions as the HeapPriorityQueue from the text (a constructor, add, min, remove\_min, is\_empty, and the \_\_len\_\_ magic method) with the same functionality. This will be done by **importing the BinaryNode object from Lab 04** 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**. **(PI 2.2/ABET[2])**
2. What are the inherent benefits and drawbacks of this (linked tree-based) backing representation? Discuss with respect to implementation, efficiency, and memory usage as compared to an array-based or linked list-based implementation. **(PI 1.2/ABET[1])**

The implication is not much different from the linked list, getting the minimum value is quick. The efficiency isn’t the best if the list or tree are large. As far as the memory usage this takes up more memory since it requires to use the left, right and parent nodes. But this is a bit more difficult than an array based.

**“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 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.

1. Implement an object called TwoDSequencePQ that implements a priority queue using a two-dimensional Python sequence as the backing data structure for the priority queue. Implement all of the same public functions as the HeapPriorityQueue from the text (a constructor, add, min, remove\_min, is\_empty, and the \_\_len\_\_ magic method) with the same functionality. 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 2.2/ABET[2])**
2. What are the inherent benefits and drawbacks of this (two-dimensional sequence) backing representation? Discuss with respect to implementation, efficiency, and memory usage as compared to an array-based or linked list-based implementation.

The implementation isn’t too difficult. Some benefits are removing is quick, and it’s done in constant time. Some draw backs are finding where to insert new elements can be time consuming. This can waste a lot of space when the array is filled up you might have to double it just to be on the safe side of having enough memory and this just waste memory.

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 4.1/ABET[3])**

The main linked list is responsible for maintaining both the priority and the head node for the linked list. This is done by altering the node object and information can be stored within the node.

1. Implement an object called LinkedLinkedPQ that implements a priority queue using a singly linked list that stores singly linked lists as the backing data structure for the priority queue. Implement all of the same public functions as the HeapPriorityQueue from the text (a constructor, add, min, remove\_min, is\_empty, and the \_\_len\_\_ magic method) with the same functionality. For this implementation, you must 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**.
2. What are the inherent benefits and drawbacks of this (essentially two-dimensional singly linked list) backing representation? Discuss with respect to implementation, efficiency, and memory usage as compared to an array-based or linked list-based implementation.

This is more difficult to implement in comparison to the twoDsequence The memory usage is more because the pointers to the next node have to be stored, but there is no wasted space. Some drawbacks are that you will not be able to move backwards and still have to search through every row and list. This is better when the information stored is not too large

**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 and this document in the final submission. Alter the cover page and answer questions 1, 3, 4, 6, 8, 9, and 11 within this book. For this lab, you may submit everything as a single zipped file if you wish.

**DEADLINE**: This assignment is due by 8:00 AM EST on Monday, December 9. 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.**