Thomas Payne

CSCI 2270

5.4.2018

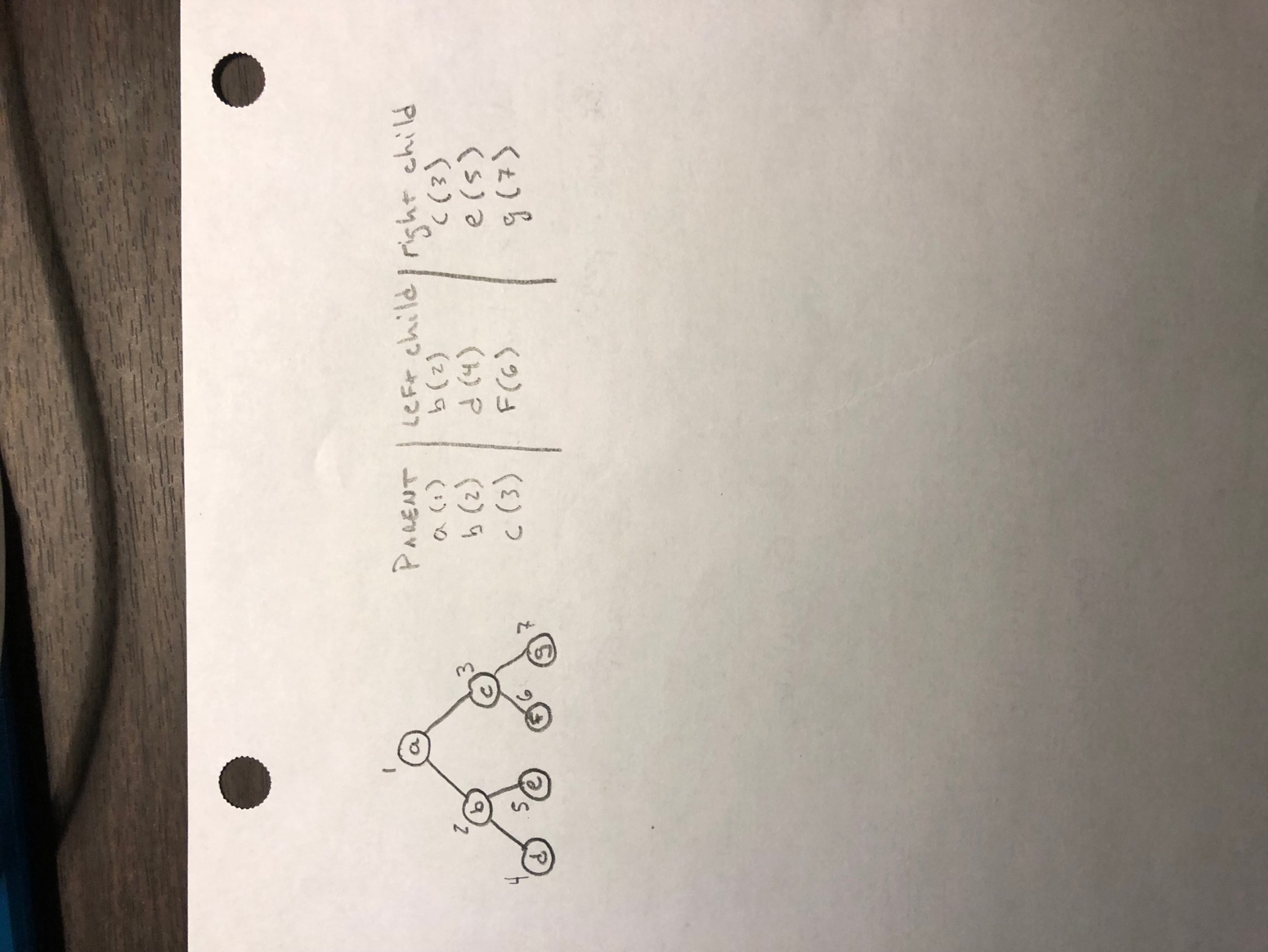
Final Project Analysis

**Purpose:**

The purpose of this project is to analyze and compare the enqueue and dequeue time of a set of data using three different minimum priority queue implementations. The three data structures used is an array implementation of a binary heap, a singly linked list, and a standard library priority queue. We are evaluating the run time speed to enqueue and dequeue n number of elements 500 times. Enqueuing and dequeuing 500 times for each set of elements allows us to form an accurate average and determine trends in consistency for each algorithm.

**Procedure:** We will be measuring the runtime differences via Chrono, returned as a float value in seconds.

**Array implementation of a binary heap:**

The enqueue function takes three parameters, a string name and two integer values, primary priority and secondary priority respectfully. There are also two integer values, last index (initialized to 0) and max index (initialized to size of data set) that are stored as private variables to check the size of the array. Initially, the implementation checks if the array is not full, if this is the case, we create a new item of struct priorityItem, which holds a string name, primary and secondary priority. The first item is implemented at the 1st index in the array, not the 0th index. This is because in future steps we will find the child and parent items via multiplication and division, and multiplying / dividing by 0 gives you 0, which would return incorrect child / parent values. We then set a Boolean variable that allows to check the validity of the tree once a new element is added. We set a ‘current’ variable to the last index (initially at 1 for the first element) that keeps track of where we are in the array. While the current index is greater than 1, and our Boolean is unchanged check the following cases:

First Case: The current primary priority is less than the parent primary priority. If this is true, we swap the values and change the current index to that of the parent. This allows us to re-enter the loop with a new current index, fixing violations from the bottom to top, pushing the smaller items to the top.

To further explain how parent / children indices are related:

The left child index is related to the parent by [parentindex \* 2].

The right child index is related to the parent by [parentindex \* 2+1].

The parent index is either child index / 2. The function takes the floor of this operation.

Second Case: The parent primary priority and the child primary priority are the same.

Check: Check the secondary priorities of the parent and the children. If a child has lower values than the parent, swap the values. If not, change the bool, that small instance of the tree is valid.

If the first and second cases are not executed, change the bool indicating a partially ordered binary heap. The purpose of the while loop is to fix any violations (children are of smaller value than parent), it will run until the tree has no violations.

The dequeue function preforms a very similar task as far as reorganizing the tree to fix any violations. We must always reorganize the tree because we are dequeuing at the very top. Initially, we check if the tree is empty, if not, we set the root of the tree to the item at the last index and the item at the last index to null. After we decrement the last index to indicate we are removing an item. We then create a left child and right child variables that current \* 2 and current index \* 2 + 1. While the current index < max index we check the following cases:

First Case: Check if the leftChild > lastIndex. This would there is no children. If this is the case, we stop.

Second Case:

Check: Check if the rightChild > last index. This means there is only one child, as the right child does not exist, so the smaller child must be the left child.

Check: To find the smaller child we check the primary priorities of the left and right children and pick the lowest. If they have the same primary priorities, check the secondary priorities.

Check: Once found the smaller child, check if the current index primary priority is greater than the smaller child primary priority. If so, there is a violation and we need to swap their values again, and reset the current index to equal the smaller child index so when the current index is passed back in, resolutions start at the parent.

Check: if the above check is not executed then the the current index and smaller child have the same primary priorities, we resolve this by checking the secondary priorities and swap the values if needed.

Again, this while loop will run until the tree is resolved of violations, and will return the dequeue item.

**Linked List implementation:**

The enqueue function essentially creates a new node with the provided name, primary priority and secondary values. This function then adds to a linked list, sorting while going based on the priority values. The following is checked to make sure we insert in the right place:

Check: If the head if null, nothing is in the list, reset head to the new node.

Check: If the new node is less than the head, reset the head to the new node.

Check: if the new node primary priority is equal to the head primary priority.

Check secondary priorities and insert where necessary.

Check: Create a temp node to traverse through the list in a while loop (this checks in between cases)

Check: if the new node primary priority < temp -> next -> primary priority.

Swap if so. Otherwise, check if they are equal. If they are check the secondary priorities.

Check: If temp -> next == null, you are at the end, insert new node at the tail.

The dequeue function simply removes the node at the head, and resets the head to head -> next.

**STL implementation:**

To implement the standard library, I created a struct patient which holds variables for the values in my data set, all initialized to empty string and 0. I began by initializing a priority queue passing in three parameters, patient, a vector of my primary priorities, and my compare priority. My compare parameter resorts to another function, compare, which takes in two patients and accesses their primary and secondary priorities. It then returns that of the highest priority. The STL implementation automatically creates a priority queue give these priorities, and the top priority for dequeue can be easily accessed by the .pop() method.

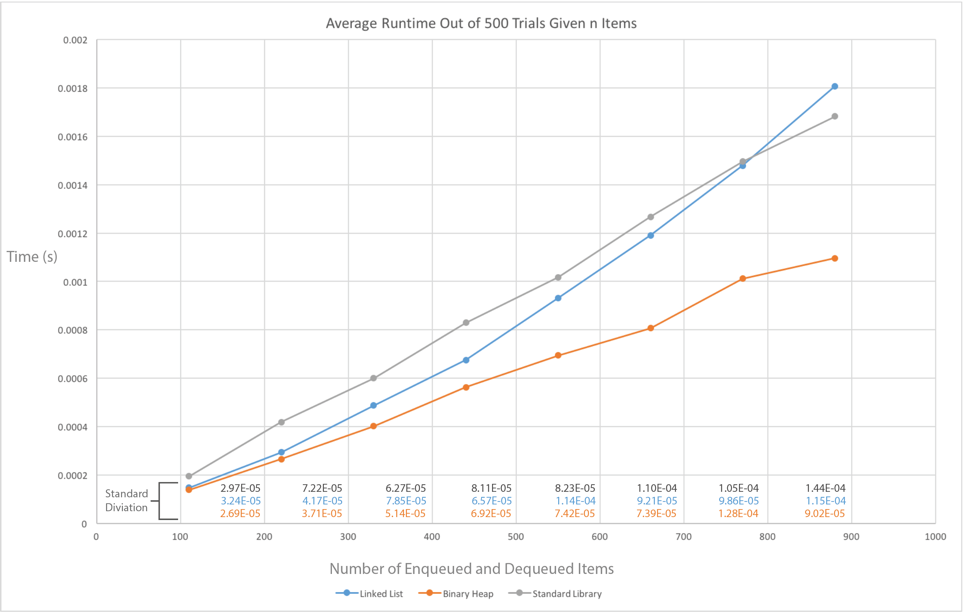
**Data:**

The data set used in theses implementation is a .csv file with 880 lines of data in it. Each line of data is separated by a string patient, an integer priority, and a second integer treatment. For all implementations the data is parsed by line. Each line is parsed into the three values (string, int, int) and passed into my enqueue functions. Some of the primary priorities are the same, so my enqueue and dequeue functions looks at the treatment (secondary priority) value to determine who is the top priority. Running the algorithms with a larger data set ie. 10000 patients would most likely give me even more accurate data, as well as allowing me to more easily determine the curve of the data. This curve could be compared to the known runtime equation, so I could see how closely my algorithm preformed next to the ideal case.

**Results:**

I gathered my data by averaging the runtime of n items, 500 times. Running each algorithm 500 times for each data set allows me to gather rather accurate data to determine the the proper trends between the three algorithms. The data depicts that the most efficient algorithm for enqueuing and dequeuing n elements was the array implementation of a binary heap, then the Linked List implementation, followed by the Standard Library implementation.

A few things I noticed:

* The difference in runtime performance becomes more obvious as the data set becomes larger. For instance, at n = 880 the binary heap executed in ~.0011s, and the linked list executed in ~.0018s. This means the binary heap is approximately 39% more efficient than the linked list.
* It seems the linked list implementation begins to get less efficient as the data set increases. This makes sense as finding where to enqueue the item, at worst case is O(n), where as worst case for enqueuing an item in a binary heap is O(logn).
* Though I am not sure the optimal runtime on the STL Implementation, it seems to be linear, quite possibly O(n).
* The standard deviations varied tremendously for each implementation, with each n sized data set. This suggest the algorithms are not very consistent from data set to data set, rather unimportant, but worth noting.
* When the item count surpasses ~780 the STL algorithm surpasses the linked list. The linked list seems to look more like an exponential function as n gets large. This suggest that with the larger data the algorithm efficiency would be binary heap > STL > linked list.