**Programming Assignment 2 Report**

**By: Tai Dao**

**BST Design and Concepts**

**Node Class**

I made a Node class to represent each node in the binary search tree. This node class typically contains pointers to the left child, right child, and parent node with getters and setters for each. These pointers typically point to nothing until the tree is built up. It also contains a key which is an object called Process.

I made this class implement compareTo for whenever I needed to do comparisons between one node and another. This compareTo compares based on the key’s (The key is the Process) priority.

**Tree Class**

I’ve implemented everything based on CLRS. The only thing different is that I added a method called

**public** **void** **processInsert**(**Process** p) {

**Node** **z** = **new** Node(p);

treeInsert(**this**, z);

}

This is needed because I can’t just insert a Process into a tree with treeInsert. I needed to first create a Node with its key set as the Process that was to be inserted. Then I could call treeInsert().

Another method I modified was treeSearch. It wasn’t mentioned in the pseudocode in CLRS, but I needed to add some return nulls for when there is no leftChild / rightChild. This is for error handling, for the case that the google is searching for a priority # that doesn’t exist.

**Process Class**

In my design I have one Object called Process. When this object is created with the constructor Process(int currentPID) a random priority from 0-9999 is assigned and a processID is assigned based on the currentPID. The currentPID is tracked in main and starts from 0. It implements comparable in order to make it possible to compare each process to each other based on priority. It has getters for the processID & priority and a setter for priority.

It also has another constructor Process() which is utilized to create a Process with the lowest possible priority of 0 and a null process PID.

**BstTester**

20 processes are created with random priority and inserted into the binary search tree on start. Then a menu with options appear to test the Tree. Everything else is pretty self-explanatory.

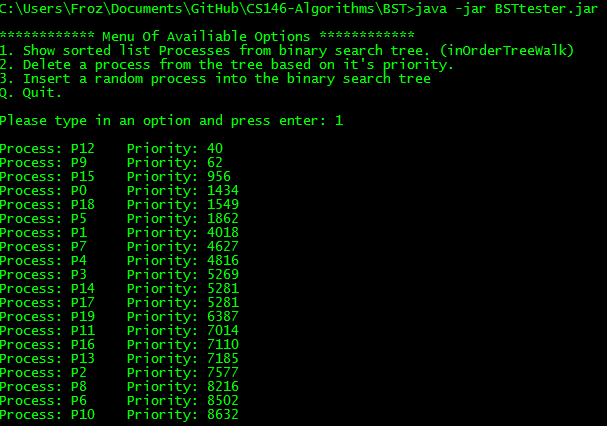
The menu allows the user to display a sorted list of processes in the BST. It also allows the user to enter in the priority of a process that they want deleted from the BST. This calls a treeSearch to find the Node to delete then calls treeDelete to delete that Node.

Search wasn’t required for this assignment. I just thought it was implied in order for me to test treeDelete I needed search.

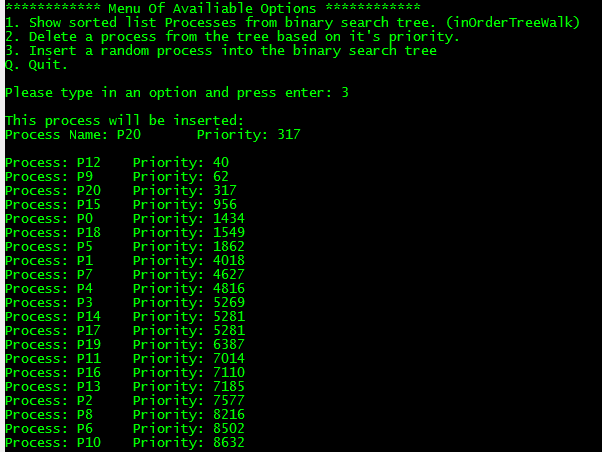
Lastly this menu allows the user to insert a random process into the BST.

**Screenshots of BST Application**

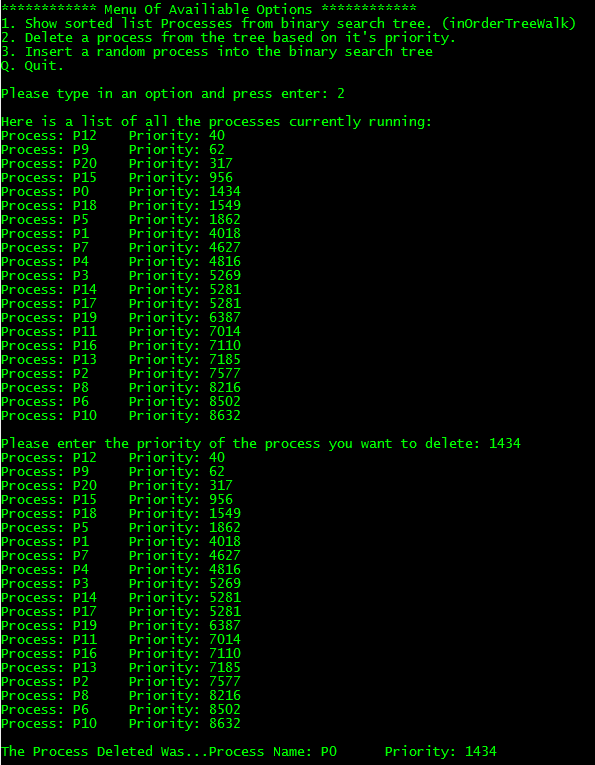
1. Showing initial bst in sorted order…



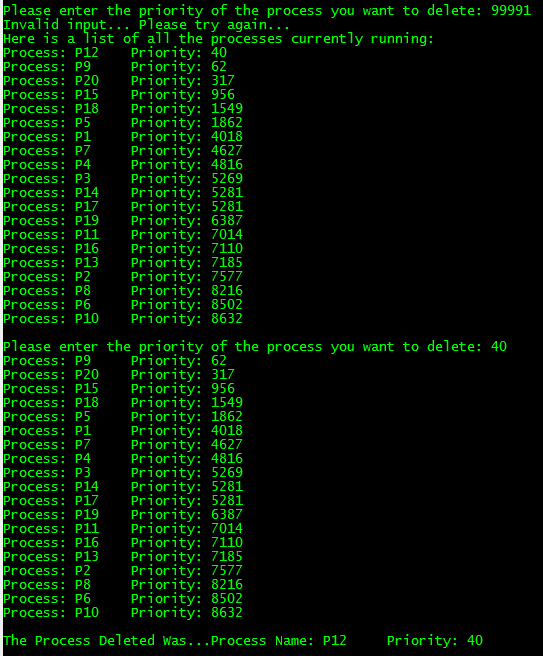
2. Inserting a random process into the BST



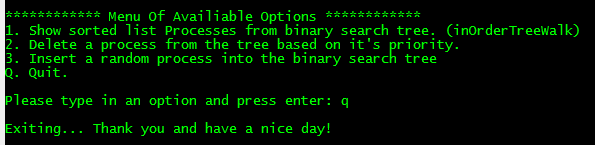
3. Deleting a process from the tree based on its priority



4) Deleting a process where the priority entered does not exist in the tree and trying again.



5) Quitting



**Procedures to run code (BST)**

Unzipping (Windows)

To unzip files go to https://ninite.com/. And checkmark the box next to "7-Zip". If you don't have "Java 8" checkmark the box next to Java 8 too. Click on "Get Your Ninite" and run the downloaded file to install the required applications.

After 7-Zip is finished installing right click on the zip file you want to unzip. Then select Extract To "\Dao-PA2.zip". A new folder called "Dao-PA2" will be created in the same directory as the zip file.

Running the application (Windows)

Java 8 is required. If you haven't installed it from the previous step with ninite. Go back to ninite.com and include Java 8 for the "Get Your Ninite.”

Navigate into the "BST" folder inside the folder you extracted with 7-Zip. While holding 'Shift'... Right-click on a blank space inside the folder, then click "Open command window here"

Now type in the following below without quotes:

| 'java - jar BSTtester.jar'

Then press Enter to run the application.

**Problems Encountered & Lessons Learned (BST)**

Well at first I was unsure of how you would insert a Process with treeInsert when its parameters were (Tree t, and Node z). I ended up solving this problem by creating a new processInsert method that creates a new Node(Process p). Then it sets that node’s key to the Process p. After that, treeInsert could be called with that Node as the parameter, solving my problem.

Initially for my treeSearch I was using Process as the parameter for key. Then, I realized I wanted Process’s priority as the key not the process itself because I wanted to search based on priority.

inOrderTreeWalk was confusing for me to understand at first. Then I realized starting at the root, it will keep seeking the leftChild until there is no more child first. This is always the minimum value in the tree and gets printed! Then it will continue on with the right child and keeps walking left to find the next minimum value… This keep repeating itself until there are no more nodes to walk.

**Chained Hash Table Design and Concepts**

In my design an arraylist of LinkedList Processes with 11 elements was created in the tester, it is called **table**. In order to initialize the table I created a constructor in my ChainedHash class.

**The Constructor** **ChainedHash**(**LinkedList**<Process>[] table

This constructor was used to initialize every element in the array called **table** to empty LinkedLists.

For my hash function I used process.getPriority() % table.length. This function gets used by hashInsert, and hashSearch.

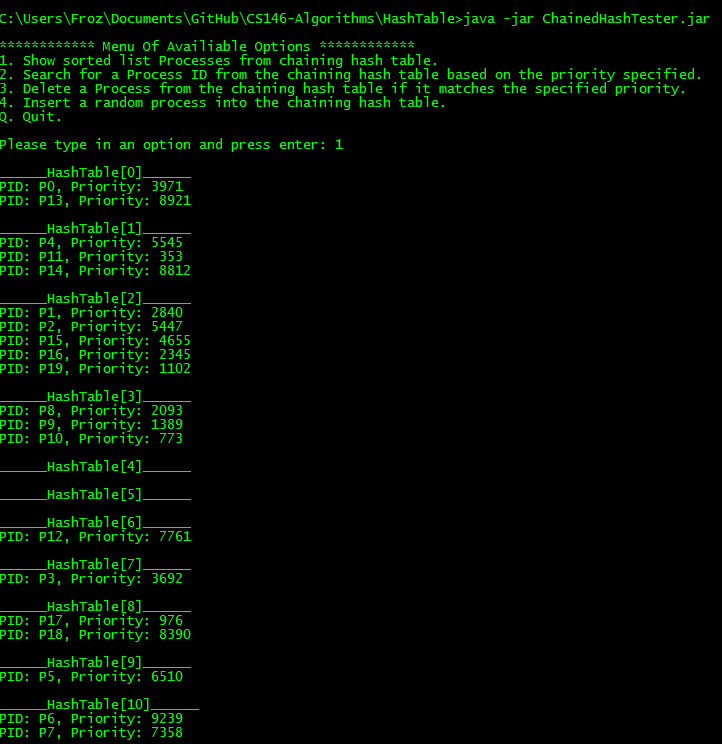
In order to fit the requirements the table.length would result in 11 because the **table** in the tester has 11 elements. If I wanted to do a hash with 13 elements I could just create a new table with 13 elements instead.

**public** **Process** **hashSearch** (**LinkedList**<Process>[] table, **int** priority)

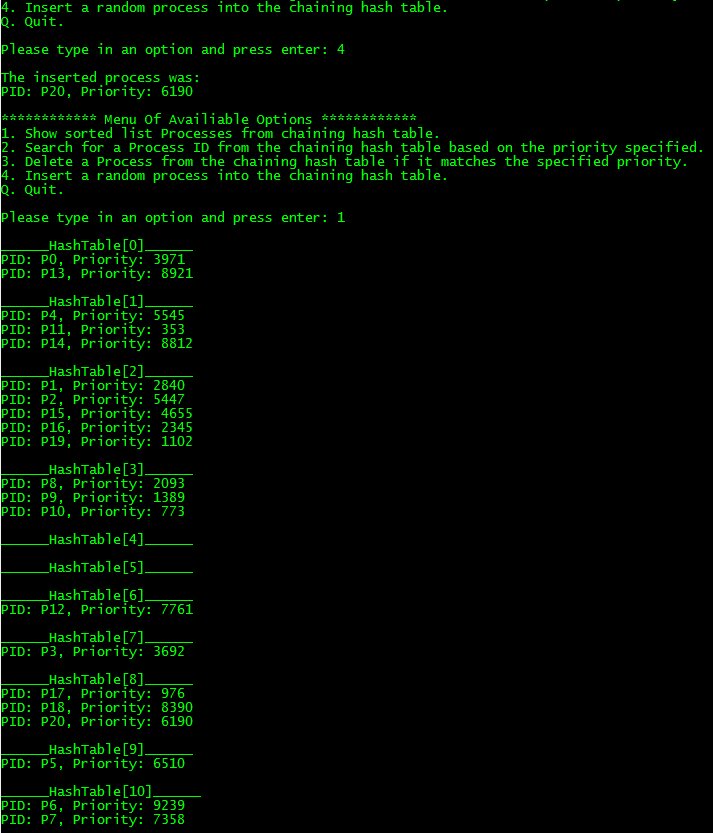
For my search I designed it so that it would return null of there didn’t exist any Process’s with the priority entered. If a Process with that priority was found it would return the process and exit for loop.

**Screenshots of ChainedHashTable Application**

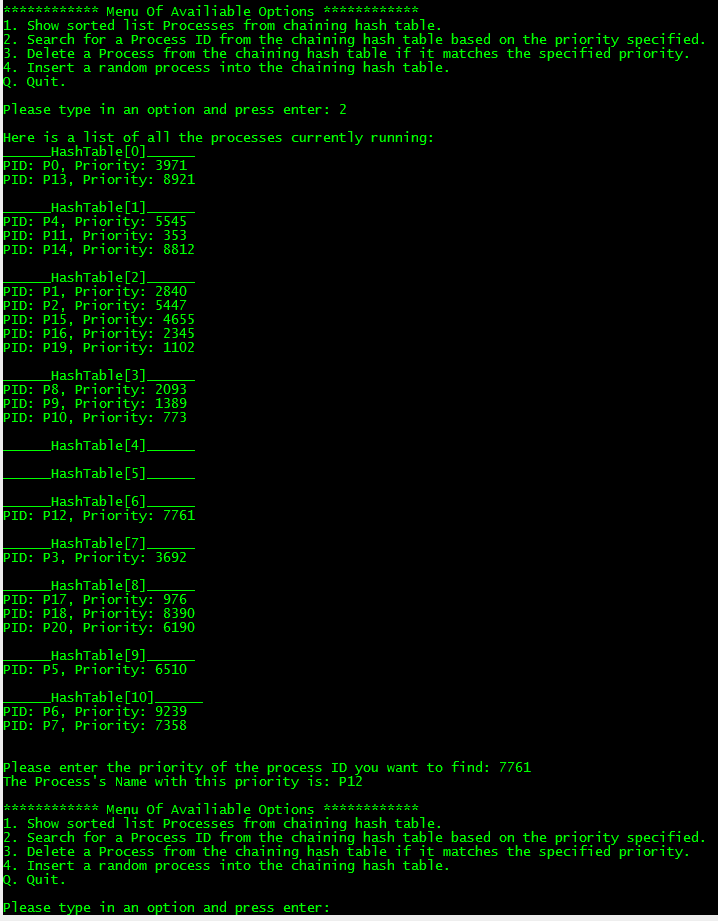
1. Showing initial hash table…



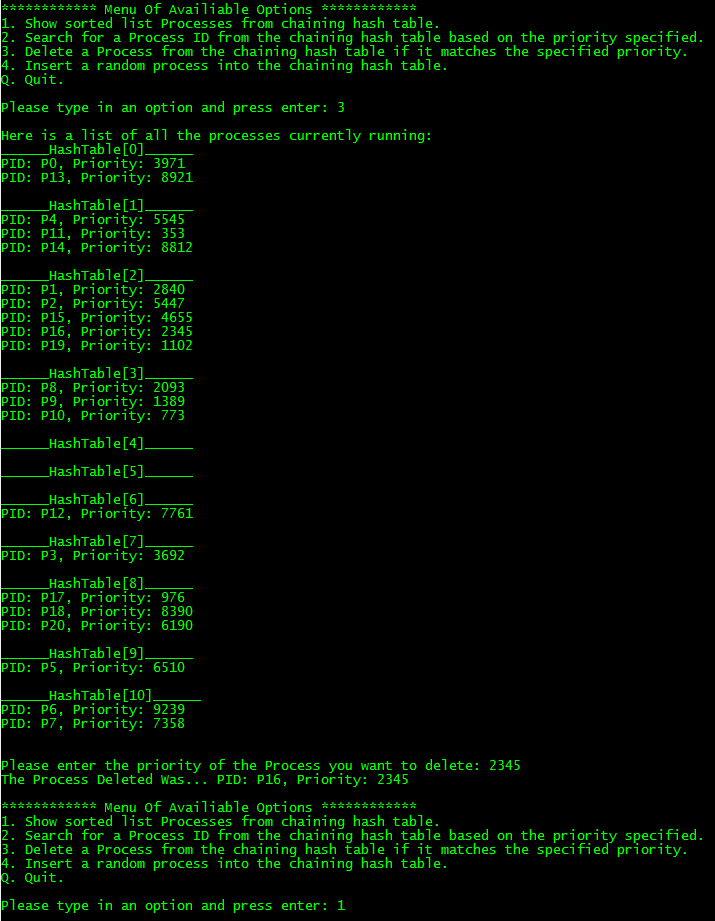
2. Inserting a random process into the chaining hash table



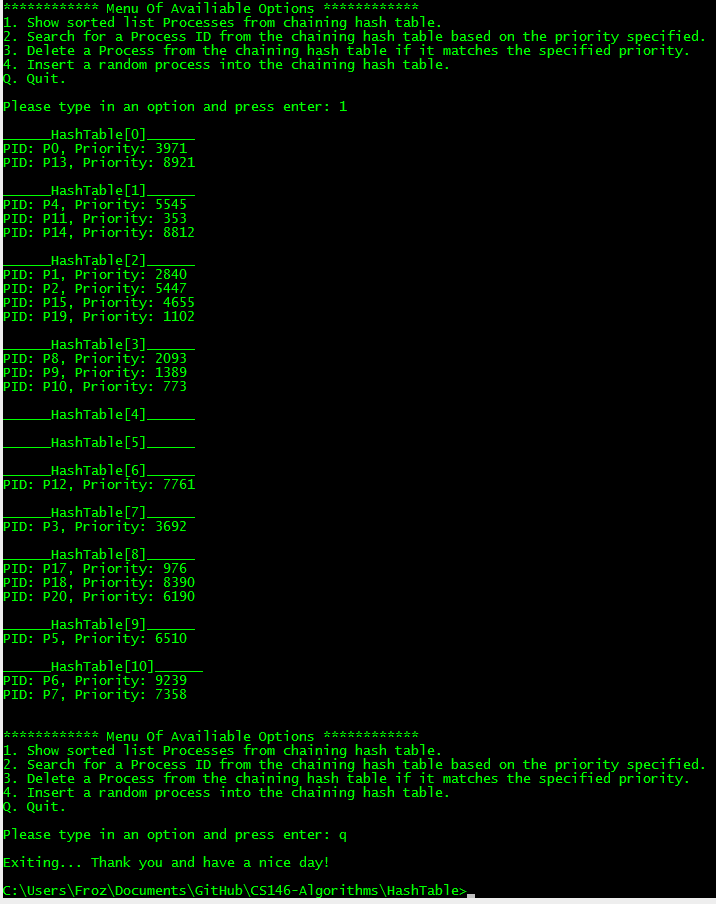
3. Searching for a process name when given a priority code.



4. Delete a process from the chaining hash table (part 1 of 2)…



4. Delete a process from the chaining hash table (part 2 of 2) viewing hash table after deletion.



**Procedures to run code (ChainedHashTable)**

Unzipping (Windows)

To unzip files go to https://ninite.com/. And checkmark the box next to "7-Zip". If you don't have "Java 8" checkmark the box next to Java 8 too. Click on "Get Your Ninite" and run the downloaded file to install the required applications.

After 7-Zip is finished installing right click on the zip file you want to unzip. Then select Extract To "\Dao-PA2.zip". A new folder called "Dao-PA2" will be created in the same directory as the zip file.

Running the application (Windows)

Java 8 is required. If you haven't installed it from the previous step with ninite. Go back to ninite.com and include Java 8 for the "Get Your Ninite.”

Navigate into the "HashTable" folder inside the folder you extracted with 7-Zip. While holding 'Shift'... Right-click on a blank space inside the folder, then click "Open command window here"

Now type in the following below without quotes:

| 'java - jar ChainedHashTester.jar'

Then press Enter to run the application.

**Problems encountered & Lessons Learned (ChainedHashTable)**

Most of my problems were to figure out how to use the generic java.util.LinkedList. I had to view the documentation on oracles to figure out the methods I wanted to use and to figure out whether it was a singly or doubly linked list. Fortunately, it was a doubly linked list.

I also initially made LinkedList<Process>[] **table** as an attribute of the ChainedHash class. However I made it in the tester in order to conform to the pseudocode. All chained hash functions take parameters table and an object x in the pseudocode. I made the **table** in tester so I could pass it to the ChainedHash functions. I can see why the book chose to accept **table** as a parameter. It’s so the programmer can choose to insert any sized table of their choosing. The hash function will adapt depending on the size of the table.

From the CLRS textbook I learned that the doubly linked list is preferred because deletion is fast. Unlike single linked lists, deletion in a doubly linked list doesn’t have to first find the Process then update the next attribute of the Process’s predecessor.