CPU Scheduling Simulation

**Introduction:** In this assignment, you need to implement a priority queue \texttt{PQueue} class using a max-heap and to implement a MaxHeap classs in linked version (not array version). Each node in the max-heap contains a process. The Process class implements Comparable interface so that the comparison between nodes in max-heap can be made by calling \texttt{compareTo} method. To implement the \texttt{compareTo} method, process with larger priority number has higher priority. In case of tie, process with earlier arrival time should be picked.

This priority queue class \texttt{PQueue} is used for simulating the CPU round robin scheduling. This CPU scheduling is works as follows.

1. A system defines a unit time (time slice).
2. There is a priority queue holds the processes waiting for CPU time.
3. Each process has a priority level, time remaining to finish, arrival time.
4. The system assigns the next CPU time (time slice) to the highest priority process (if there is a tie, the one arriving first is picked) in the priority queue.
5. Each time a process get a time slice from CPU, its remaining time should be decremented by one.
6. In order to avoid starvation problem, the system will increment the priority for the lower priority processes. In this simulation, if the lower priority processes do not get any time slice for a certain amount of time (\texttt{timeToIncrementLevel}), its priority will be incremented by one until it reaches the highest allowable priority level.

There are 8 java source files in \texttt{~jhyeh/cs225/lab/lab5/files/}

\texttt{BTNode.java} defines a binary tree node as discussed in classes. Each node in max-heap is a \texttt{BTNode}. This class is finished.

\texttt{LinkedListQueue.java} defines the queue ADT in linked list version. This class is useful for level order traversal in a heap. You need to finish \texttt{enqueue} and \texttt{dequeue} methods in this class.

\texttt{CPUScheduling.java} simulates the round robin cpu scheduling algorithm. This class is finished.

\texttt{ProcessGenerator.java} randomly generates processes. This class is finished.

\texttt{Averager.java} keeps track of number of processes in the simulation and computes the average turn around time. This class is finished.

\texttt{Process.java} defines a process. You need to implement the \texttt{compareTo} method in this class.
MaxHeap.java defines a max-heap. You need to implement this class.

PQueue.java defines a priority queue. You need to implement this class.

Description Implement the PQueue, MaxHeap, and Process classes as follows.

// File PQueue.java
// This class using a max-heap to implement a priority queue.
// This priority queue is to simulate the round robin cpu scheduling.
import java.util.*;
public class PQueue
{
    private MaxHeap mHeap;

    // Constructor.
    public PQueue()
    {
        mHeap = new MaxHeap();
    }

    public PQueue(Process p)
    {
        mHeap = new MaxHeap(p);
    }

    // return true if the priority queue is empty.
    // otherwise, return false.
    public boolean isEmpty()
    {
        return true if the priority queue is empty.
        // otherwise, return false.
        public boolean isEmpty()
    }

    // Insert a Process p into the priority queue.
    // POST: The process p is inserted into the
    // priority queue.
    public void enPQueue(Process p)
    {
        // PRE: This Priority Queue should not be empty.
        // POST: Remove and return the process with the highest
        // priority (in case of tie, pick the one
        // with earliest arrival time)
        public void enPQueue(Process p)
    }

    // Return the number of processes currently in the priority queue.
    public int size()
    {
        // Increment the timeNotProcessed by one for all processes in
        // the priority queue. If a process’s timeNotProcessed exceeds
        // timeToIncrementLevel and if its original priority is less
        // than maxLevel, then increment its priority by 1 and
        public int size()
// reset the timeNotProcessed to 0.
// Finally, restructure the PQueue to make it a heap again.
public void update(int timeToIncrementLevel, int maxLevel)
{
}

// file: MaxHeap.java
// This file is to define a max heap data structure
import java.util.*;
public class MaxHeap
{
    private BTNode root;

    // Constructors
    public MaxHeap()
    {
    }

    public MaxHeap(Object o)
    {
        root = new BTNode(o, null, null, null);
    }

    // Return true if the heap is empty.
    // Otherwise, return false.
    public boolean isEmpty()
    {
    }

    // Return number of nodes in the heap
    public int sizeOfHeap()
    {
    }

    // return the height of the heap.
    public int heightOfHeap()
    {
    }

    // return true if the tree is full.
    // otherwise return false.
    public boolean isFull()
    {
    }

    // Return the first node having a null child in the heap. That is,
    // the first node has a null child in the level order sequence.
    // This method is called to located the correct position for insertion.
    // If the heap is an empty heap, then return null.
    public BTNode firstNodeWithNullChild()
    {
    }

    // Return the last node in the heap in the level order sequence.
    // This method is called to locate the node to be physically
    // deleted from the heap for removal.
    // If the heap is empty, return null.
    public BTNode lastNode()
    {
    }

    // PRE: both node’s left subtree and right subtree are heaps
// POST: push down the ‘node’ to its correct position
// and make the tree a heap.
public void maxHeapifyDown(BTNode node)

// PRE: All other nodes, except the node, are in their correct
// positions as a heap.
// POST: bubble up the node to its correct position
// and make the tree a heap.
public void maxHeapifyUp(BTNode node)

// PRE: the tree pointed by root is not necessary a heap
// POST: make the tree a heap
public void buildHeap()

// PRE: insert a new node containing the object o into the heap
// POST: the new object is inserted.
public void insert(Object o)

// PRE: remove the object in the root node from the heap.
// if the heap is empty, throw NoSuchElementException.
// POST: the object in the root is removed and returned.
public Object remove()

// public method to return a heap iterator
public Iterator heapItr()
{
    return new heapIterator();
}

// inner class to implement the Iterator interface
// return the nodes of the heap in a levelorder traversal
private class heapIterator implements Iterator
{
    private LinkedListQueue q;

    // Constructor
    public heapIterator()
    {
        q = new LinkedListQueue();
        if (root != null)
            q.enQueue(root);
    }

    // return true if there are more nodes in the heap.
    // otherwise, return false.
    public boolean hasNext()
    {
        // PRE: hasNext() return true.
    }

    // return the next object in the heap.
    // if the heap is empty, throw NoSuchElementException.
    // POST: return the next object.
    public Object next()
    {
        // PRE: next() return the next object.
    }

    // reset the iterator.
    public void remove()
    {
        // PRE: remove() return the next object.
    }
}
public Object next()
{
    // Do nothing for this method
    public void remove()
    {
    }
}

// File Process.java
// This Process class provides a process object.
public class Process implements Comparable
{
    private int arrivalTime;
    private int priority;
    private int timeRemaining;
    // The number of time slices the process needs before it can be finished.
    private int timeNotProcessed;
    // The number of unit time has past since last time it got a time slice.

    public Process(int arrival, int level, int timeRequire)
    {
        arrivalTime = arrival;
        priority = level;
        timeRemaining = timeRequire;
        timeNotProcessed = 0;
    }

    // return 1 if this process should be processed before p.
    // return -1 otherwise.
    // process with higher priority get processed first,
    // in case of tie, process with earlier arrival time
    // should be picked.
    public int compareTo(Object p)
    {
        // All other methods here
    }

    // return 1 if this process should be processed before p.
    // return -1 otherwise.
    // process with higher priority get processed first,
    // in case of tie, process with earlier arrival time
    // should be picked.
    public int compareTo(Object p)
    {
        // All other methods here
    }

    // All the files for the simulation are in
    // submit jhyeh cs225 5