Justin Mendiguarin

Professor Ouyang

CSC 130

11/29/15

Assignment 4 - A Simple Simulator for Priority Based CPU Scheduling

**Design:**

Design and implement a Java class of binary heap that satisfies the following requirements:

* The binary heap has a size of 101, and is implemented using an array and a heapSize index.

**protected** **static** **final** **int** ***heapSize*** = 101; // 101

**private** **int** currentSize; // Size of heap at the moment.

**Private** BinaryHeapData *array*[]; // Heap array

* Each object in the heap is a process that contains the following data fields: (a) nice value (key), and (b) process ID (a sequence number).

**class** BinaryHeapData {

**private** **int** key = 0;

**private** **int** processID = 0;

**public** BinaryHeapData() {

}

**public** BinaryHeapData(**int** processID, **int** key) {

**this**.processID = processID;

**this**.key = key;

}

//**public** **void** setKey(**int** key) {

// **this**.key = key;

//}

**public** **int** getKey() {

**return** key;

}

//**public** **void** setProcessID(**int** processID) {

// **this**.processID = processID;

//}

**public** **int** getProcessID() {

**return** processID;

}

}

3. The process objects in the heap are organized based upon their nice values, i.e. the nice value of a process must be no greater than that of its children.

* The process objects in the heap are organized based upon their nice values, i.e. the nice value of a process must be no greater than that of its children. Every time you check for the children minimum of (2i and 2i+1). If there is a violation, which is that the parent is > the minimum child, we swap the two indexes. Then the new child must be checked if it has any children. If so, test again. If not, continue.

**private** **void** percolateDown(**int** i) {

**int** left = 2 \* i;

**int** right = 2 \* i + 1;

**if** (*array*[i] == **null** || i > currentSize || left > currentSize|| **this**.isEmpty()) {

**return**;

}

**if** (right <= currentSize) {

**if** (*array*[left].getKey() < *array*[right].getKey()) {

**if** (*array*[left].getKey() < *array*[i].getKey()) {

swap(left, i);

percolateDown(left);

}

} **else** {

**if** (*array*[right] != **null** && (*array*[i].getKey() > *array*[right].getKey())) {

swap(right, i);

percolateDown(right);

}

**else** **if** (*array*[left] != **null** && *array*[i].getKey() > *array*[left].getKey()) {

swap(left, i);

percolateDown(left);

}

}

} **else** **if** (left == currentSize) {

**if** (*array*[left].getKey() < *array*[i].getKey()) {

swap(left, i);

percolateDown(left);

}

} **else** {

**return**;

}

**return**;

}

**public** **void** swap(**int** i, **int** j) {

BinaryHeapData temp = **new** BinaryHeapData();

temp = *array*[j];

*array*[j] = *array*[i];

*array*[i] = temp;

}

* A process is inserted into the heap using heap structure property and heap order property.

**public** **void** insert(BinaryHeapData userNode) {

**if** (currentSize + 1 > ***heapSize***)

**return**;

currentSize++;

array[currentSize] = userNode;

percolateUp(currentSize);

}

**public** **void** percolateUp(**int** i) {

**int** parentIndex = i / 2;

**if** (parentIndex > 0) {

**if** (array[i].getKey() < array[parentIndex].getKey()) {

swap(parentIndex, i);

percolateUp(parentIndex);

}

}

**return**;

}

* A process with the minimum nice values is removed from the heap using heap structure property and heap order property.

**public** BinaryHeapData deleteMin() {

**if** (**this**.isEmpty()) {

**return** **new** BinaryHeapData(-1, -1);

} **else** {

**int** root = 1;

BinaryHeapData deleteMe = array[root];

array[root] = array[currentSize];

array[currentSize] = **null**;

currentSize--;

percolateDown(root);

**return** deleteMe;

}

}

Develop a driver that does the following:

* Create two instances of heap objects where the arrays are initially empty.

BinaryHeap heap1 = **new** BinaryHeap();

BinaryHeap heap2 = new BinaryHeap();

* Create 100 processes in such a manner that their IDs sequentially go from 1 to 100, and their nice values are between 0 and 39 (use a random generator). Then put them sequentially into the array of the first heap.

BinaryHeapData[] processArray = *createProcesses*();

heap1.putSequentially(processArray);

**public** **static** BinaryHeapData[] createProcesses() {

**int** randKey = -1;

Random rand = **new** Random();

BinaryHeapData[] localArray = **new** BinaryHeapData[***heapSize***];

**for** (**int** i = 1; i < ***heapSize***; i++) {

randKey = rand.nextInt((39 - 1) + 1) + 1;

localArray[i] = **new** BinaryHeapData(i, randKey);

}

**return** localArray;

}

**public** **void** putSequentially(BinaryHeapData [] processArray) {

**for** (**int** i = 1; i < ***heapSize***; i++) {

*array*[i] = processArray[i];

currentSize++;

}

}

* Use "buildHeap" algorithm to reorganize the first heap according to the heap order property. Binary Heap Order Property: start at the end. Take the size and divide by 2. This will give us the last non-leaf node. Work from the end back to the front. Start at n/2 and move towards 1.

**public** **void** buildHeap() {

**for** (**int** i = currentSize / 2; i > 0; i--) {

percolateDown(i);

}

}

* Perform "deleteMin" to remove a process from the first heap (then the process is running), then "insert" it into the second heap (the process returns to the waiting queue after running out of its time share). Keep doing this until the first heap becomes empty and the second heap is full.

**while** (!heap1.isEmpty()) {

BinaryHeapData deleted = heap1.deleteMin();

heap2.insert(deleted);

}

**public** BinaryHeapData deleteMin() {

**if** (**this**.isEmpty()) {

**return** **new** BinaryHeapData(-1, -1);

} **else** {

**int** root = 1;

BinaryHeapData deleteMe = array[root];

array[root] = array[currentSize];

array[currentSize] = **null**;

currentSize--;

percolateDown(root);

**return** deleteMe;

}

}

**public** **void** insert(BinaryHeapData userNode) {

**if** (currentSize + 1 > ***heapSize***)

**return**;

currentSize++;

array[currentSize] = userNode;

percolateUp(currentSize);

}