Cosc 241
Programming and Problem Solving
Lecture 21 (13/5/2019)
Priority queues and HeapSort

Lech Szymanski
lechszym@cs.otago.ac.nz

Keywords: priority queue, HeapSort

UNIVERSITY of OTAGO
SAPERE AUDE
Te Whare Wānanga o Otāgo
NEW ZEALAND

Priority Queues

- In a *priority queue* each element is added with an associated *priority*.
- When an element is removed, it is the element with highest priority.
- If more than one element shares highest priority then the earliest arrival should be removed.
- So, if all elements have the same priority then it behaves as a queue, while if the elements are added in strictly increasing order of priority, then it behaves like a stack.

Priority Queue implementation

- A heap is the ideal backing structure for a priority queue,
- We just need to do a bit of bundling together of items with priorities, and then just use the basic heap operations.
- We suppose that priorities are supplied as integers.

Priority queue

```
public class PriorityQueue<T>{

   private ArrayHeap<QueueNode<T>> heap;
   private static int arrivalNumber = 0;

   public PriorityQueue() {
      heap = new ArrayHeap<QueueNode<T>>();
   }

   public void add(T item, int priority) {
      heap.add(new QueueNode<T>(item, priority));
   }

   public T removeNext() {
      return heap.remove().value;
   }

   private class ...
}
```

Priority queue node

```
private class QueueNode<T> implements Comparable<QueueNode<T>>

   private T value;
   private int priority;
   private int arrival;

   private QueueNode(T value, int priority) {
      this.value = value;
      this.priority = priority;
      this.arrival = arrivalNumber;
      arrivalNumber++;
   }

   public int compareTo(QueueNode<T> other) {
      if (this.priority < other.priority) return -1;
      if (this.priority > other.priority) return 1;
      return other.arrival - this.arrival;
   }

}
```

HeapSort

- *HeapSort* (1964) is an in place, comparison based, array sorting algorithm which has *guaranteed* worst case $O(n \log n)$ behaviour.
- The basic idea:
  - Organize the elements of the array into a heap structure.
  - Exchange the first (largest) element, and the last element.
  - Restore the heap structure (except for the final element).
  - Repeat last two steps until finished.

Organizing the heap

- There are two choices *top down* or *bottom up*.
- The first mimics our algorithms from the previous lecture, effectively treating a growing initial segment of the array as a heap and adding one element at a time, letting it float as high as necessary.
- The second imagines the tree structure already in place over the whole array, and fixes violations of the heap property beginning from the lowest non-leaf nodes and moving upwards.
- The first is easier conceptually but $O(n \log n)$.
- The second is actually $O(n)$.

Why are they different?

- In the top down version, where elements float up the heap, the elements from the larger parts of the heap float farthest.
- In particular each element of the bottom level (size $n/2$) might need to float to the top ($\log n$ away), requiring $O(n \log n)$ steps.
- In the bottom up version, the elements in the larger levels are sinking down, and have a shorter distance to travel.
- In fact, at most $n/2^i$ elements need to sink a distance i.
- So the total number of steps required is

$$O\left(\sum_{i=1}^{\infty} i \frac{n}{2^i}\right) = O(n)$$

- That's why!

Doing the sort

```
public static <T extends Comparable<T>> void sort(T[] a) {

   heapify(a);
   for(int i = a.length-1; i > 0; i--) {
      swap(a, 0, i);
      siftDown(a, 0, i);
   }

}
```

Heapifying (bottom up)

```java
private static <T extends Comparable<T>> void
                              heapify(T[] a) {
   for(int i = (a.length-1)/2; i >= 0; i--) {
      siftDown(a, i, a.length);
   }
}
```

Sifting down

```java
private static <T extends Comparable<T>> void
                              siftDown(T[] a, int s, int high) {
    while(true) {
      int largerChildIndex = getLargerChildIndex(a,s,high);
      if (largerChildIndex == NO_LARGER_CHILD) {
        break;
      }
      swap(a, s, largerChildIndex);
      s = largerChildIndex;
    }
}
```