COSC 2011 Section N

Tuesday, April 3 2001

Overview

- Priority Queues
  - Quick Review of definitions
  - Quick Review of ADT
  - Compositions & Comparators
  - Implementation
    - Sequence
- Mid-term Test Questions

Priority Queues: Compositions (1)

- Composition Objects:
  - A single object $e$ that is a composition of two (or more) other objects.
  - Each element in the priority queue is essentially a pair:
    - Each element has a key
  - Can create a composition of each key and element:

Priority Queues: Compositions (2)

- Implementing the Composition Concept:
  - Define the Item class:
    ```java
    public class Item{
        private Object key, elem;
        public Item(Object k, Object e){
            key = k;
            elem = e;
        }
        public Object key(){return key;}
        public Object element(){return element;}
        public void setKey(Object k){key = k;}
        public void setElem(Object e){elem = e;}
    }
    ```

Priority Queues: Comparators (1)

- How do we Compare keys?
  - Keys are Objects so they may be different types!
  - Use different priority queue for each key type and each possible way of comparing keys of such types.
    - Different priority queue for integer keys, strings...
    - Not very general.
    - Too much similar code!
Priority Queues: Comparators (2)

- Alternative Strategy:
  - Requires keys to be able to compare themselves to one another.
  - Have a general priority queue class that stores instances of a key class that implements a \textit{Comparable} interface.
    - Encapsulates all the usual comparison methods.

Priority Queues: Comparators (3)

- Problem with the \textit{Comparable}:
  - May be cases where we are asking “too much” of the keys.
    - Keys may not know how to be compared!
    - $4 \leq 11$ using integer keys.
    - $11 \leq 4$ using String keys.
  - Do not rely on keys to provide comparison rules!

Priority Queues: Comparators (4)

- Use \textit{comparator} objects:
  - External to the keys
  - Supply comparison rules.
  - Given to the priority queue during construction
  - Can be changed if necessary.

  - When the priority queue needs to compare keys, it uses the \textit{comparator} object.

Priority Queues: Comparators (5)

- Comparator ADT Methods:
  - The comparator ADT includes:
    - \texttt{isLessThan}(a, b)
    - \texttt{isLessThanOrEqualTo}(a, b)
    - \texttt{isEqualTo}(a, b)
    - \texttt{isGreaterThan}(a, b)
    - \texttt{isGreaterThanOrEqualTo}(a, b)
    - \texttt{isComparable}(a)
Priority Queues: Sequence Implementation (6)

Implementation with an Unsorted Sequence

- Let's try to implement a priority queue with an unsorted sequence S.
- The elements of S are a composition of two elements, k, the key, and e, the element.
- We can implement insertLast() by using insertLast() on the sequence. This takes \( O(1) \) time.

![Sequence Implementation Diagram](image)

- However, because we always insert at the end, irrespectively of the key value, our sequence is not ordered.

Priority Queues: Sequence Implementation (7)

Implementation with an Unsorted Sequence (contd.)

- Thus, for methods such as minElement(), minKey(), and removeMin(), we need to look at all the elements of S. The worst case time complexity for these methods is \( O(n) \).

![Sequence Implementation Diagram](image)

- Performance summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertLast</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>minKey, minElement</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>removeMin</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>

Priority Queues: Sequence Implementation (8)

Implementation with a Sorted Sequence

- Another implementation uses a sequence S, sorted by increasing keys.
- minElement(), minKey(), and removeMin() take \( O(1) \) time.

![Sequence Implementation Diagram](image)

- However, to implement insertLast(), we must now scan through the entire sequence in the worst case. Thus, insertLast() runs in \( O(n) \) time.

Priority Queues: Sorting (1)

Selection Sort

- Selection Sort is a variation of PriorityQueueSort that uses an unsorted sequence to implement the priority queue P.
- Phase 1: the insertion of an item into P takes \( O(1) \) time.
- Phase 2: removing an item from P takes time proportional to the current number of elements in P.

<table>
<thead>
<tr>
<th>Sequence S</th>
<th>Priority Queue P</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 4, 3)</td>
<td>(7, 8, 5)</td>
</tr>
<tr>
<td>(2, 3, 6)</td>
<td>(7, 8, 5)</td>
</tr>
<tr>
<td>(2, 3, 4)</td>
<td>(7, 8, 5)</td>
</tr>
<tr>
<td>(2, 3, 4)</td>
<td>(7, 8, 5)</td>
</tr>
<tr>
<td>(2, 3, 4)</td>
<td>(7, 8, 5)</td>
</tr>
<tr>
<td>(2, 3, 4)</td>
<td>(7, 8, 5)</td>
</tr>
</tbody>
</table>
Priority Queues: Sorting (2)

Selection Sort (cont.)

- As you can tell, a bottleneck occurs in Phase 2. The first removeMinElement operation takes $O(n)$, the second $O(n-1)$, etc. until the last removal takes only $O(1)$ time.
- The total time needed for phase 2 is:

$$O(n + (n - 1) + ... + 2 + 1) = O\left(\sum_{i=1}^{n} i\right)$$

- By a well-known fact:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

- The total time complexity of phase 2 is then $O(n^2)$. Thus, the time complexity of the algorithm is $O(n^2)$.

Priority Queues: Sorting (3)

Insertion Sort

- Insertion sort is the sort that results when we perform a PriorityQueueSort implementing the priority queue with a sorted sequence.

<table>
<thead>
<tr>
<th>Input</th>
<th>Sequence $S$</th>
<th>Priority Queue $P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(4, 8, 2, 5, 3, 9)</td>
<td>(7)</td>
</tr>
<tr>
<td>(b)</td>
<td>(8, 2, 5, 3, 9)</td>
<td>(4, 7)</td>
</tr>
<tr>
<td>(c)</td>
<td>(2, 5, 3, 9)</td>
<td>(4, 7, 8)</td>
</tr>
<tr>
<td>(d)</td>
<td>(5, 3, 9)</td>
<td>(2, 4, 7, 8)</td>
</tr>
<tr>
<td>(e)</td>
<td>(3, 5, 9)</td>
<td>(3, 4, 5, 7, 8)</td>
</tr>
<tr>
<td>(f)</td>
<td>(2)</td>
<td>(2, 3, 4, 5, 7, 8, 9)</td>
</tr>
<tr>
<td>(g)</td>
<td>()</td>
<td>()</td>
</tr>
</tbody>
</table>

Phase 2:

- We improve phase 2 to $O(n)$.
- However, phase 1 now becomes the bottleneck for the running time. The first insertItem takes $O(1)$ time, the second one $O(2)$, until the last operation takes $O(n)$ time, for a total of $O(n^2)$ time.
- Selection-sort and insertion-sort both take $O(n^2)$ time.
- Selection-sort will *always* execute a number of operations proportional to $n^2$, no matter what is the input sequence.
- The running time of insertion sort varies depending on the input sequence.
- Neither is a good sorting method, except for small sequences.
- We have yet to see the ultimate priority queue....

Priority Queues: Sorting (5)

- By now, you’ve seen a little bit of sorting, so let us tell you a little more about it.
- Sorting is essential because efficient searching in a database can be performed only if the records are sorted.
- It is estimated that about 20% of all the computing time worldwide is devoted to sorting.
- We shall see that there is a trade-off between the “simplicity” and efficiency of sorting algorithms.
- The elementary sorting algorithms you’ve just seen, though easy to understand and implement, take $O(n^2)$ time (unusable for large values of $n$).
- More sophisticated algorithms take $O(n \log n)$ time.
- Comparison of Keys: do we base comparison upon the entire key or upon parts of the key?
- Space Efficiency: *in-place* sorting vs. use of auxiliary structures.
- Stability: a stable sorting algorithm preserves the initial relative order of equal keys.