Adaptable Priority Queues

Entry and Priority Queue ADTs

- An entry stores a (key, value) pair
- Entry ADT methods:
  - get_key(): returns the key associated with this entry
  - get_value(): returns the value paired with the key associated with this entry

- Priority Queue ADT:
  - insert(k, x): inserts an entry with key k and value x
  - remove_min(): removes and returns the entry with smallest key
  - min(): returns, but does not remove, an entry with smallest key
  - size(), isEmpty()
Example

- Online trading system where orders to purchase and sell a stock are stored in two priority queues (one for sell orders and one for buy orders) as \((p,s)\) entries:
  - The key, \(p\), of an order is the price.
  - The value, \(s\), for an entry is the number of shares.
  - A buy order \((p,s)\) is executed when a sell order \((p',s')\) with price \(p' < p\) is added (the execution is complete if \(s' > s\)).
  - A sell order \((p,s)\) is executed when a buy order \((p',s')\) with price \(p' > p\) is added (the execution is complete if \(s' > s\)).

- What if someone wishes to cancel their order before it executes?
- What if someone wishes to update the price or number of shares for their order?

Methods of the Adaptable Priority Queue ADT

- \texttt{remove}(e): Remove from \(P\) and return entry \(e\).
- \texttt{replaceKey}(e,k): Replace with \(k\) and return the key of entry \(e\) of \(P\); an error condition occurs if \(k\) is invalid (that is, \(k\) cannot be compared with other keys).
- \texttt{replaceValue}(e,v): Replace with \(v\) and return the value of entry \(e\) of \(P\).
Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>Output</th>
<th>$P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert(5,A)</td>
<td>$e_1$</td>
<td>(5,A)</td>
</tr>
<tr>
<td>insert(3,B)</td>
<td>$e_2$</td>
<td>(3,B),(5,A)</td>
</tr>
<tr>
<td>insert(7,C)</td>
<td>$e_3$</td>
<td>(3,B),(5,A),(7,C)</td>
</tr>
<tr>
<td>min()</td>
<td>$e_2$</td>
<td>(3,B),(5,A),(7,C)</td>
</tr>
<tr>
<td>key($e_2$)</td>
<td>3</td>
<td>(3,B),(5,A),(7,C)</td>
</tr>
<tr>
<td>remove($e_1$)</td>
<td>$e_1$</td>
<td>(3,B),(7,C)</td>
</tr>
<tr>
<td>replaceKey($e_2$,9)</td>
<td>3</td>
<td>(7,C),(9,B)</td>
</tr>
<tr>
<td>replaceValue($e_3$,D)</td>
<td>C</td>
<td>(7,D),(9,B)</td>
</tr>
<tr>
<td>remove($e_2$)</td>
<td>$e_2$</td>
<td>(7,D)</td>
</tr>
</tbody>
</table>

Locating Entries

- In order to implement the operations remove(e), replaceKey(e,k), and replaceValue(e,v), we need fast ways of locating an entry e in a priority queue.

- We can always just search the entire data structure to find an entry e, but there are better ways for locating entries.
Location-Aware Entries

- A location-aware entry identifies and tracks the location of its (key, value) object within a data structure
- Intuitive notion:
  - Coat claim check
  - Valet claim ticket
  - Reservation number
- Main idea:
  - Since entries are created and returned from the data structure itself, it can return location-aware entries, thereby making future updates easier

List Implementation

- A location-aware list entry is an object storing
  - key
  - value
  - position (or rank) of the item in the list
- In turn, the position (or array cell) stores the entry
- Back pointers (or ranks) are updated during swaps
Heap Implementation

- A location-aware heap entry is an object storing:
  - key
  - value
  - position of the entry in the underlying heap
- In turn, each heap position stores an entry
- Back pointers are updated during entry swaps

Performance

- Improved times thanks to location-aware entries are highlighted in red

<table>
<thead>
<tr>
<th>Method</th>
<th>Unsorted List</th>
<th>Sorted List</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>size, isEmpty</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insert</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>min</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>removeMin</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>replaceKey</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>replaceValue</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
Java Implementation

```java
/** An implementation of an adaptable priority queue using an array-based heap. */
public class HeapAdaptablePriorityQueue<K,V> extends HeapPriorityQueue<K,V>
    implements AdaptablePriorityQueue<K,V> {

    // -------------- nested AdaptablePQEntry class --------------
    /** Extension of the PQEntry to include location information. */
    protected static class AdaptablePQEntry<K,V> extends PQEntry<K,V> {
        private int index; // entry’s current index within the heap
        public AdaptablePQEntry(K key, V value, int j) {
            super(key, value); // this sets the key and value
            index = j; // this sets the new field
        }
        public int getIndex() { return index; }
        public void setIndex(int j) { index = j; }
    } // end of nested AdaptablePQEntry class

    /** Creates an empty adaptable priority queue using natural ordering of keys. */
    public HeapAdaptablePriorityQueue() { super(); }

    /** Creates an empty adaptable priority queue using the given comparator. */
    public HeapAdaptablePriorityQueue(Comparator<K> comp) { super(comp); }

    // protected utilities
    /** Validates an entry to ensure it is location-aware. */
    protected AdaptablePQEntry<K,V> validate(Entry<K,V> entry)
        throws IllegalArgumentException {
        if (!(entry instanceof AdaptablePQEntry))
            throw new IllegalArgumentException("Invalid entry");
        AdaptablePQEntry<K,V> locator = (AdaptablePQEntry<K,V>) entry; // safe
        int j = locator.getIndex();
        if (j >= heap.size() || heap.get(j) != locator)
            throw new IllegalArgumentException("Invalid entry");
        return locator;
    }

    /** Exchanges the entries at indices i and j of the array list. */
    protected void swap(int i, int j) {
        super.swap(i,j); // perform the swap
        ((AdaptablePQEntry<K,V>) heap.get(i)).setIndex(j); // reset entry’s index
        ((AdaptablePQEntry<K,V>) heap.get(j)).setIndex(i); // reset entry’s index
    }

    // --------------- end of class HeapAdaptablePriorityQueue
}
```

© 2014 Goodrich, Tamassia, Goldwasser  Adaptable Priority Queues 11

Java Implementation, 2

```java
// protected utilities
/** Validates an entry to ensure it is location-aware. */
protected AdaptablePQEntry<K,V> validate(Entry<K,V> entry)
    throws IllegalArgumentException {
    if (!(entry instanceof AdaptablePQEntry))
        throw new IllegalArgumentException("Invalid entry");
    AdaptablePQEntry<K,V> locator = (AdaptablePQEntry<K,V>) entry; // safe
    int j = locator.getIndex();
    if (j >= heap.size() || heap.get(j) != locator)
        throw new IllegalArgumentException("Invalid entry");
    return locator;
}

/** Exchanges the entries at indices i and j of the array list. */
protected void swap(int i, int j) {
    super.swap(i,j); // perform the swap
    ((AdaptablePQEntry<K,V>) heap.get(i)).setIndex(j); // reset entry’s index
    ((AdaptablePQEntry<K,V>) heap.get(j)).setIndex(i); // reset entry’s index
}
```

© 2014 Goodrich, Tamassia, Goldwasser  Adaptable Priority Queues 12
Java Implementation, 3

```java
/* Restores the heap property by moving the entry at index j upward/downward. */
protected void bubble(int j) {
    if (j < size() && compare(heap.get(j), heap.get(parent(j))) < 0)
        heap[j] = heap[parent(j)];
    else
        downheap(j);  // although it might not need to move
}
```

Java Implementation, 4

```java
/** Removes the given entry from the priority queue. */
public void remove(Entry<K,V> entry) throws IllegalArgumentException {
    int j = locator.indexOf(entry);
    if (j == heap.size() - 1)  // entry is at last position
        heap.remove(heap.size() - 1);
    else {
        swap(j, heap.size() - 1);  // swap entry to last position
        heap.remove(heap.size() - 1);  // then remove it
        bubble(j);  // and fix entry displaced by the swap
    }
}
```

/** Replaces the key of an entry. */
public void replaceKey(Entry<K,V> entry, K key) throws IllegalArgumentException {
    locator.setKey(key);  // method inherited from PQEntry
    bubble(locator.indexOf(entry));  // with new key, may need to move entry
}

/** Replaces the value of an entry. */
public void replaceValue(Entry<K,V> entry, V value) throws IllegalArgumentException {
    locator.setValue(value);  // method inherited from PQEntry
```