Priority Queues, Sets, and Bags

Wellesley College CS230
Lecture 12
Monday, March 12
Handout #22

Exam 1 due Friday, March 16

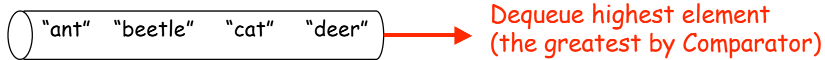
Overview of Today's Lecture

1. More on Priority Queues = ordered queues in which maximal element is dequeued.
2. Sets = conceptually unordered collections of elements without duplicates (but need order on elements to implement efficiently).
3. Bags = conceptually unordered collections of elements with duplicates

Priority Queues

A priority queue is a kind of queue in which deq()/front() return the highest-priority element, where priority is determined by a Comparator. If comparator is null, this indicates use of the natural Comparable ordering.

```
PQueue<String> pq = new PQueue<String>(); // Use null Comparator
pq.enq("beetle"); pq.enq("cat"); pq.enq("deer"); pq.enq("ant");
```




```
while (! pq.isEmpty()) System.out.print(pq.deq() + " ");
System.out.println();
```

```
deer cat beetle ant
```

Specifying a Priority Queue Comparator

The following method demonstrates how a pqueue depends on the Comparator:

```
public static void test (PQueue<String> pq) {
   pq.enq("beetle");  pq.enq("cat");  pq.enq("deer"); pq.enq("ant");
   while (!pq.isEmpty()) System.out.print(pq.deq() + " ");
   System.out.println();
}
```

A particular pqueue implementation that we'll study later.

1. PQueueTester.test(new PQueueVectorLH<String>());

2. PQueueTester.test(new PQueueVectorLH<String>(new LengthComparator()));

3. PQueueTester.test(new PQueueVectorLH<String>(
      new ComparatorInverter<String> (new ComparableComparator<String>())));

4. PQueueTester.test(new PQueueVectorLH<String>(
      new ComparatorInverter<String> (new LengthComparator())));

What are Priority Queues Good For?

1. They model situations where certain "to-do" items have a higher priority than others. E.g.:
- printer queue that prints shorter documents before longer ones;
- emergency room doctor who performs triage on patients
- student who works on assignment based on due date/importance

2. They can be use to sort elements:

```
public static <T> void pqsort (Comparator<T>, T[] elts) {




}
```

This is a good idea only if an efficient priority queue implementation is used. In particular, **heapsort** is an efficient sorting algorithm using the **heap** implementation of a pqueue (later this semester).

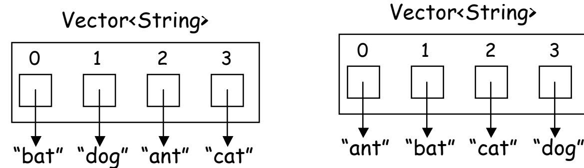
A Priority Queue (PQueue) Interface

```
public interface PQueue<T> extends Queue<T> {
  public Comparator<T> comparator();
    // Returns comparator used by this pqueue to determine priority.
    // Returns null if the natural Comparable ordering is being used.
  public void enq(T elt); // Add elt to this pqueue.
  public T deq(); // Remove and return highest-priority element from this pqueue.
  public T front(); // Return highest-priority element from this pqueue
                    // without removing it.
  public Iterator<T> iterator();
    // Return an iterator that yields this pqueue's elements from
    // highest priority to lowest priority.
  public PQueue<T> copy(); // Return a shallow copy of this pqueue
  public String toString(); // Return string listing elements of this pqueue from highest
       priority down.

  // Inherits the following methods with their usual behavior from Queue<T>:
  // boolean isEmpty(),  int size(); void clear();

  // A pqueue implementation should also provide two constructor methods:
  // 1. A nullary constructor method in which the Comparator is assumed null
  // 2. A unary constructor method explicitly supplying the  Comparator method
}
```

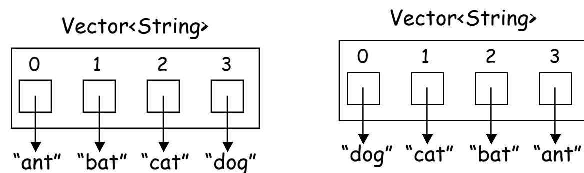
Implementing a PQueue as a Vector

Suppose we implement a PQueue as a Vector:

1. Should elements be unsorted or sorted? Why?



2. Should elements be sorted low-to-high or high-to-low? Why?



PQueueVectorLH: Constructor Methods

```
/** Represent a pqueue as a vector of elements from low to high priority */
public class PQueueVectorLH<T> implements PQueue<T> {
  // Instance Variables:
  private Vector<T> elts; // Stores sorted elements
  private Comparator<T> comp; // Remembers comparator
  // Constructor Methods:
  public PQueueVectorLH(Comparator<T> comp) {
    this.elts = new Vector<T>();
    this.comp = comp;
  }
 public PQueueVectorLH() {
    this(null); // null means to use "natural" Comparable ordering
  }

 // Instance Methods go here ...
}
```

PQueueVectorLH: Easy Instance Methods

```
// One brand new instance method:
public Comparator<T> comparator() { return comp; }

// Most instance methods like those in back-to-front queue:
public T deq () {
  if (elts.size() == 0)
    throw new RuntimeException("Attempt to deq() empty pqueue.");
  else return elts.remove(elts.size()-1);
}
public T front () {
  if (elts.size() == 0)
    throw new RuntimeException("Attempt to take front() of empty pqueue.");
  else return elts.get(elts.size()-1);
}
public boolean isEmpty() { return elts.size() == 0; }
public int size() { return elts.size(); }
public void clear() { elts = new Vector<T>(); }
// copy(), iterator(), and toString() are also like in back-to-front queue.
```

12-9

PQueueVectorLH: Enqueuing

```
public void enq (T elt) {
  // Here, use linear search for insertion.  Could use binary search instead, but
  // insertion will generally be linear anyway due to element shifting in vector.
  int i = elts.size() - 1;
  while ((i >= 0) && (compare(elt, elts.get(i)) < 0)) {
      // Loop invariant: all slots in elts at indices > i hold values greater than elt.
      i--;
    }
    // Invariant: i in range [-1..(elts.size() - 1)] is largest i such that all slots
    // with indices greater than i contain values greater than elt.
    elts.add(i+1,elt);
  }
```

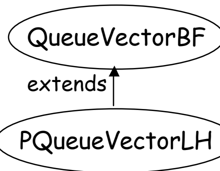
12-10

PQueueVectorLH: Comparing Elements

```
private int compare(T x, T y) {
  if (comp == null) { // Assume that x is an instance of Comparable.
    return ((Comparable) x).compareTo(y);
    // A ClassCastException will be thrown if this assumption is violated.
    // Note: the Java 1.5 compiler will issue an unchecked warning for the
    //  above line because it cannot prove that the downcast will succeed.
  } else {
    return comp.compare(x,y);
  }
}
```

```
[fturbak@puma queue] javac PQueueVectorLH.java
Note: PQueueVectorLH.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

[fturbak@puma queue] javac -Xlint:unchecked PQueueVectorLH.java
PQueueVectorLH.java:29: warning: [unchecked] unchecked call to
compareTo(T) as a member of the raw type java.lang.Comparable return
((Comparable) x).compareTo(y);
```

Using Inheritance to Simplify Implementation



Implements Queue<T> using vector of elements arranged from back to front

Implements PQueue<T> using vector of elements arranged from low to high priority. Can inherit the the following from QueueVectorBF:

```
public T deq();
public T front();
public boolean isEmpty();
public boolean size();
public boolean clear();
public Iterator<T> iterator();
```

PQueueVectorLH must still implement the following:

```
Constructor methods
public Comparator<T> comp();  // new to PQueue<T>
public void enq (T elt);  // inserts in sorted order
public PQueue<T> copy();  // has different return type
public String toString(); // has different name
```

Can Even Inherit toString()

```
public class QueueVectorBF<T> implements Queue<T> {

   public String toString() {
      StringBuffer buff = new StringBuffer();
      buff.append(name() + "[");
      … code for listing elements from front to back …
      buff.append("]");
      return buff.toString();
    }

   public String name() {// can be overridden by PQueueVectorLH.
      return "QueueVectorBF";
   }
}

public class PQueueVectorLH<T> extends QueueVectorBF<T>
   implements PQueue<T> {

   public String name() {// overrides name() in QueueVectorBF
      return "PQueueVectorLH";
   }
   … other methods go here …
}
```

12-13

Implementing Priority Queues as a Mutable List

We could instead implement a Priority Queue using a mutable list.

How should elements be arranged?

What is the code for enq(), deq(), front()?

12-14

Priority Queues Implementation Summary

Using vectors or lists, all priority queue operations can be implemented efficiently except for enq():

- Constructor methods, front(), deq(), size(), isEmpty(), clear() take constant time.
- copy(), toString(), and iterator() take time linear in number of elements (iterator() can be changed to constant time).
- enq() takes time linear in number of elements: BAD!!!

12-15

Mathematical Sets

In mathematics, a set is an unordered collection of elements w/o duplicates.

A set is written by enumerating its elements between braces. E.g., the empty set (with no elements) is written {}.

Insertion: inserts an element into a set if it's not already there:

insert("a", {}) = {"a"}
insert("b", {"a"}) = {"a", "b"} = {"b", "a"} // order is irrelevant
insert("a", {"a", "b"}) = {"a", "b"} = {"b", "a"}
insert("c", {"a","b"}) = {"a", "b", "c"} = {"a", "c", "b"} = {"b", "a", "c"}
= {"b", "c", "a"} = {"c", "a", "b"} = {"c", "b", "a"}

Deletion: removes an element if it's in the set:

delete("a", {}) = {}
delete("a", {"a", "b"}) = {"b"}
delete("b", {"a", "b"}) = {"a"}
delete("c", {"a", "b"}) = {"a", "b"}

Size (Cardinality): |S| is the number of elements in S.

|{}| = 0; |{"a"}| = 1; |{"a", "b"}| = 2; |{"a", "b", "c"}| = 3;

12-16

More Sets

Membership: x ∈ S is true if x is in the set S and false otherwise.

"a" ∈ {} = false; "a" ∈ {"a", "b"} = true; "c" ∈ {"a", "b"} = false;

Subset: S1 ⊆ S2 is true if all elements in S1 are also in S2.

{} ⊆ {"a", "b"} is true; {"a"} ⊆ {"a", "b"} is true; {"b"} ⊆ {"a", "b"} is true;
{"a", "b"} ⊆ {"a", "b"} is true; {"a", "b"} ⊆ {"a"} is false; {"a"} ⊆ {} is false;

Union: S1 ∪ S2 = all elements in at least one of S1 and S2.

{"a", "b", "d", "e"} ∪ {"b", "c", "d", "f"} = {"a", "b", "c", "d", "e", "f"}

Intersection: S1 ∩S2 = all elements in at both S1 and S2.

{"a", "b", "d", "e"} ∩ {"b", "c", "d", "f"} = {"b", "d"}

Difference: S1 - S2 or S1/S2 = all elements in S1 that are not in S2.

{"a", "b", "d", "e"} / {"b", "c", "d", "f"} = {"a", "e"}

12-17

Mathematical Bags (Multi-Sets)

In mathematics, a bag (multi-set) is an unordered collection of elements **with** duplicates.

A bag is written by enumerating its elements (with duplicates) between braces. E.g., the empty bag (with no elements) is written {}. The bag with one "a" is written {"a", "a"}. The bag with two "a"s is written {"a", "a"}.

Most set notions (insertion, deletion, membership, subset, union, intersection, difference, cardinality) generalize to bags as long as duplicates are accounted for. E.g:

insert("a", {"a", "b"}) = {"a" ,"a", "b"} = {"a", "b", "a"} = {"b", "a", "a"}

delete("a", {"a", "a", "a", "b", "c", "c"}) = {"a" ,"a", "b", "c", "c"}

{"a", "b"} ⊆ {"a", "a", "b"} = true; {"a", "a", "b"} ⊆ {"a", "b"} = false;

{"a", "a", "b", "d"} ∪ {"a", "d", "e"} = {"a", "a", "a", "b", "d", "d", "e"}

{"a", "a", "a", "b", "c"} ∩ {"a", "a", "b", "b"} = {"a", "a", "b"}

{"a", "a", "a", "b", "c"} / {"a", "a", "b", "b"} = {"a", "c"}

|{"a", "a", "a", "b", "c", "c"}| = 6

12-18

New Bag Operations

Multiplicity: we shall write #(x,B) (my nonstandard notation) for the number of occurrences of element x in bag B, a.k.a. the multiplicity of x in B.

Suppose B1 = {"a","a", "a", "b", "c", "c"}. Then:
#("a", B1) = 3; #("b", B1) = 1; #("c", B1) = 2; #("d", B1) = 0;

DeleteAll: delete(x,B) deletes one occurrence of x from B. Sometimes we want to delete all occurrences of x from B. For this we use deleteAll(x,B).

delete("a", {"a", "a", "a", "b", "c", "c"}) = {"a" ,"a", "b", "c", "c"}

deleteAll("a", {"a", "a", "a", "b", "c", "c"}) = {"b", "c", "c"}

deleteAll("d", {"a", "a", "a", "b", "c", "c"}) = {"a", "a", "a", "b", "c", "c"}

CountDistinct: |B| counts all element occurrences in B, including duplicates. Sometimes we want just the number of distinct elements. For this we use countDistinct(B).

|{"a", "a", "a", "b", "c", "c"}| = 6

countDistinct({"a", "a", "a", "b", "c", "c"}) = 3

Digression: Ordered Sets and Bags

Sets and bags are unordered collections of elements.

However, we can also have ordered notions of sets and bags, in which the order of elements matters.

The order might be specified by a Comparator or the Comparable compareTo() method, in which case elements would have a well-defined index. For instance, in the ordered bag

OB{"a", "a", "a", "b", "c", "c"}

"a"s are at indices 0, 1, 2; "b" is at index 3, and "c"s are at indices 4 & 5.

Since elements have a well-defined index, we could request the element at a given index or delete the element at an index from an ordered set/bag.

Inserting elements into or deleting elements from an ordered set/bag can shift the indices of other elements, similar to what happens in a vector.

We will not consider implementations of ordered sets or bags this semester.