CS61B Summer 2006
Instructor: Erin Korber
Lectures 16: 24 July

1 Stacks

- "LIFO" - last in, first out
- Can access only the top item in the stack
- `push` - put an item on the stack
- `pop` - take an item off the stack
- Sometimes also use `peek` - look at the thing on top of the stack without removing it
- ```
  public interface Stack {
    public Object pop();
    public void push(Object o);
  }
  ```
- Can be implemented easily as a singly linked list - `insertFront, removeFront` are `push, pop`
- The call stack is (obviously) an example of a stack.
- Can also be implemented as an array
  - Have to have a max size bound, or be willing to resize the array if needed (this is slow!)
  - Bottom of stack at index 0
  - Maintain a "current size" variable so we know where to go to push/pop elements.
  - Code for this implementation:

    ```
    public ArrayStack implements Stack {

       private Object[] theArray;
       private int currSize = 0;
    ```
    ```

```
    public Object pop() {
       Object item = theArray[currSize-1];
       currSize--;
       return item;
    }

    public void push(Object item) {
       if (theArray.length = currSize) {
          newArray = new Object[currSize * 2];
          for (int i = 0; i<theArray.length; i++) {
             newArray[i] = theArray[i];
          }
          theArray = newArray;
       }
       theArray[currSize] = item;
       currSize++;
    }
 }
```

2 Queues

- "FIFO" - first in, first out
- Can only add items at the front, remove them from the back
- `enqueue` - put an item at the back of the queue
- `dequeue` - remove an item from the front of the queue
- ```
  public interface Queue {
    public Object dequeue();
    public void enqueue(Object o);
  }
  ```
- Can be implemented as a singly linked list with a tail pointer - `insertBack, removeFront` are `enqueue, dequeue`
- Example: printer queues
- Queues can also be implemented as an array!
  ```

- Have to have a max size bound, or be willing to resize the array if needed (this is slow!)
- Could slide everything over one every time we remove something from the queue, but this is slow.
- Better: use a "circular buffer" implementation
  * Keep two indices, for the first and last items in the queue, which "circle back" to 0 after falling off the end of the array.
  * Code for this implementation:

```
public ArrayQueue implements Queue {
   private Object[] theArray;
   private int frontIndex = 0;
   private int rearIndex = 0;
   private int currSize = 0;

   public Object dequeue() {
     if (currSize = 0) {
        System.out.println(``empty queue'');
        return null;
     } else {
        Object item = theArray[frontIndex];
        frontIndex = (frontIndex + 1) % theArray.length;
        currSize--;
        return item;
     }
   }

   public void enqueue(Object item) {
     if (theArray.length == currSize) {
       resize();
     }
     theArray[(rearIndex + 1) % theArray.length] = item;
     rearIndex = rearIndex + 1;
     currSize++;
    }

   public void resize() {
     //elided
   }
}
```

3 Priority Queues

- Items have a key and associated value
- Can access only the item with the highest priority, which is generally the *lowest* key.
- ```
  public interface PriorityQueue {
      public boolean isEmpty();
      public void insert(KeyValPair p);
      public KeyValPair seeMin();
      public KeyValPair removeMin();
  }
  ```

4 Binary Heaps

We can implement a priority queue using a binary heap, which is a *complete* binary tree which satisfies the *heap order property.* A complete binary tree is a binary tree in which every row is full, except possibly the bottom row, which is filled from left to right. The heap order property states that no child has a key less than its parent's key. Note that any subtree of a binary heap is also a binary heap.

We can implement a binary heap in a node-and-reference way, like the binary trees that we already have. However, the completeness property makes an array-based implementation (without storing explicit child references) possible - we store the root at index 1. If a node's index is i, then its children will be at 2i and 2i+1.

Let's look at how we can implement the priority queue operations with a binary heap.

- `seeMin()` - the heap order property guarantees that the entry with the minimum key is always at the top of the heap, so we can just return the key-value pair at the root.
- `insert(KeyValPair p)` - Let the key of p be k and the value of p be v. We place the new entry p in the bottom level of the tree, at the first free spot from the left. If the bottom level is full, start a new level with x at the far left. (So in an array-based implementation, we place x in the first free location in the array.)
  ```

Of course, doing this may cause us to violate the heap-order property. We correct this by having the entry "bubble" up the tree until the heap-order property is satisfied. More precisely, we compare k with its parent's key; if k is less, we exchange p with its parent and repeat the procedure with p's new parent. For instance, if we insert an entry whose key is 2:

```
          2                   2                   2                   2
         / \                 / \                 / \                 / \
        /   \               /   \               /   \               /   \
       5     3             5     3             5     3             2     3
      / \   / \    =>     / \   / \    =>     / \   / \    =>     / \   / \
     9   6 11  4         9   6 11  4         9   2 11  4         9   5 11  4
    / \ /               / \ / \             / \ / \             / \ / \
  17 10 8             17 10 8  2          17 10 8  6          17 10 8  6
```

As this example illustrates, a heap can contain several entries with the same key.

When we finish, is the heap-order property satisfied? Yes, if the heap-order property was satisfied before the insertion. Let's look (see diagram below) at a typical exchange of p with a parent x during the insertion operation. Since the heap-order property was satisfied before the insertion, we know that x $\leq$ s (where s is p's sibling), x $\leq$ l, and x $\leq$ r (where l and r are p's children). We only swap if p ¡ x, which implies that p ¡ s; after the swap, p is the parent of s. After the swap, p is the parent of l and r. All other relationships in the subtree rooted at p are maintained, so after the swap, the tree rooted at p has the heap-order property.

```
     x          p
    / \        / \
   s   p  =>  s   x
  /\   /\    /\   /\
 l  r       l  r
```

- `KeyValPair removeMin()` - If the heap is empty, return null or throw an exception. Otherwise, begin by removing the entry at the root node and saving it for the return value. This leaves a hole at the root. We fill the hoel with the last entry in the tree (which we call "x"), so that the tree is still complete.

It is unlikely that x has the minimum key. Fortunately, both subtrees rooted at the root's children are heaps, and thus the new mimimum key is one of these two children. We bubble x down the heap as follows: if x has a child whose key is smaller, swap x with the child having the minimum key. Next, compare x with its new children; if x still violates the heap-order property, again swap x with the child with the minimum key. Continue until x is less than or equal to its children, or reaches a leaf.

Consider running removeMin() on our original tree.

```
        2                  8                  3                  3
       / \                / \                / \                / \
      /   \              /   \              /   \              /   \
     5     3            5     3            5     8            5     4
    / \   / \    =>    / \   / \    =>    / \   / \    =>    / \   / \
   9   6 11  4        9   6 11  4        9   6 11  4        9   6 11  8
  / \ /              / \                / \                / \
17 10 8            17 10              17 10              17 10
```

Above, the entry bubbled all the way to a leaf. This is not always the case, as the example below shows.

```
     1                  4                  2
    / \                / \                / \
   /   \              /   \              /   \
  2     3     =>     2     3     =>     4     3
 / \   / \          / \   /            / \   /
9   6 11  4        9   6 11           9   6 11
```

5 Heapsort

We can use heaps for another way to sort items - simply put all of them into a heap, then remove them one by one - since we are always removing the smallest, we can get a sorted list out easily.