CSCI 136
Data Structures & Advanced Programming

Jeannie Albrecht

Lecture 26

April 23, 2014

Administrative Details

- Darwin lab today
  - Part 1 due next Monday Apr 28th
  - Part 2 due Monday May 5th
- Midterm 2 @ 1:00 next Wed (Apr 24th) in Wege
  - Sample exam posted on Handouts page later today
- Tentative review session
  - Tue 9:30pm-10:30pm in TCL 202 (?)

Midterm 2

- Stacks - Linear structure
- Queues - Linear structure
- Iterators - Think about vectors, lists, etc.
- Comparables and Ordered Structures
  - Comparators vs. Comparables
  - OrderedVector, OrderedList, OrderedArray
- Trees
  - BinaryTree class
  - Tree traversal and iterators
- Priority Queues and Heaps
  - Array/vector representation of PQ
  - Heap construction and maintenance
  - No skew heaps!
- Binary Search Trees (?) - TBD

Last Time

- Briefly talked about how to represent a tree using an array (or vector/list)
- Starting talking about Priority Queues

Today's Outline

- Continue discussing priority queues
- Discuss ways to implement priority queues using ordered structures and heaps

Recap: Priority Queues

- Name is misleading: they are not FIFO!
- Always dequeue object with highest priority regardless of when it was enqueued

```
public interface PriorityQueue<E extends
                                Comparable<E>> {
      public E getFirst();
      public E remove();
      public void add(E value);
      public boolean isEmpty();
      public int size();
      public void clear();
}
```

Recap: Heap

- We can implement PQs using a heap
  - **Partially** ordered binary tree
- A heap is a **complete** tree where:
  - Root holds smallest value (i.e., one with highest priority)
  - Left and right subtrees are also heaps
- So values descend in order (priority) from root to leaf, or ascend as you go up to root from leaf
- Invariant for nodes
  - node.value() <= node.left.value()
  - node.value() <=node.right.value()
- Several valid heaps for same data set (no unique representation)

Implementing Heaps

- VectorHeap
  - Use logical array representation of BT (like last class)
  - But use extensible vector instead of array (makes adding elements easier)
- Features
  - No gaps in array – why?
    - Because BT is complete!
  - Invariant
    - `data[i] <= data[2i+1]; data[i] <= data[2i+2]`
  - When elements are added and removed, do small amount of work to "re-heapify"

Insertion

- Example
  - Add 'Z' to our heap from last class
  - Now add 'A'
- How do we insert elements into the heap?
  - First, add new Object to end of vector
  - Then heapify: percolate Object up to correct position (if needed)
  - Let's look at the code (recursive and iterative)
- Cost?
  - O(log n)

Insertion (iterative)

```
protected Vector<E> data;

public void add(E value) {
        data.add(value);
        percolateUp(data.size()-1);
}

protected static int parent(int i) {
        return (i-1)/2;
}

protected void percolateUp(int leaf) {
        int parent = parent(leaf);  //find index of parent
        E value = data.get(leaf);   //get value of leaf node (the one we just added)
        //while the leaf's value is smaller than its parent…
        while (leaf > 0 && (value.compareTo(data.get(parent)) < 0)) {
            data.set(leaf,data.get(parent));  //set parent's value to leaf node index
            leaf = parent;          //update leaf index
            parent = parent(leaf);  //recompute parent index (ie, move up one level)
        }
        data.set(leaf,value);       //we've found the right index (leaf) so set value
}
```

Removal

- Example
- How do we remove elements from the heap?
  - First remove top (root) element
  - Replace with rightmost leaf (last element)
  - Push down until heap is valid again (by always swapping element with smallest/highest priority child)
- Cost?
  - O(log n)

Removal (iterative)

```
protected Vector<E> data;

public E remove() {
        E minVal = getFirst();
        data.set(0,data.get(data.size()-1)); //move last node to index 0
        data.setSize(data.size()-1); //explicitly set vector size
        if (data.size() > 1) pushDownRoot(0);
        return minVal;
}

public E getFirst() {
        return data.get(0);
}
```

Removal (iterative)

```
protected void pushDownRoot(int root) {
    int heapSize = data.size();
    E value = data.get(root);
    while (root < heapSize) { //can't move beyond end of vector
       int childpos = left(root);  //compute index of left child
       if (childpos < heapSize) { //not at a leaf yet
       //figure out if right or left child is smaller
           if ((right(root) < heapSize) &&
               ((data.get(childpos+1)).compareTo(data.get(childpos)) < 0)) {
                  childpos++;
           }
           // Assert: childpos indexes smaller of two children
           if ((data.get(childpos)).compareTo(value) < 0) { //root is bigger than child
              data.set(root,data.get(childpos));
              root = childpos;  //need to keep moving down tree
           } else { //found right location
              data.set(root,value);
              return;
           }
       } else { //at a leaf! insert and halt
           data.set(root,value);
           return;
       }
    }
}
```

VectorHeap Summary

- Can implement methods recursively or iteratively
- Add/remove are both O(log n)
- Data is not completely sorted
  - **Partial** order is maintained
  - Why?
    - We can't say anything about order of siblings

An Aside: Skew Heap

- What if heaps are not complete BTs?
- We can implement PQs using skew heaps instead of "regular" complete heaps
- Key differences:
  - Rather than use Vector as underlying data structure, use BT
  - Need a merge operation that merges two heaps together into one heap
- Details in book…(**not on midterm!!**)

VH Questions

- Why do we swap with the smallest child in removal/pushDownRoot?
- Why do we pick rightmost leaf?
- Why are they only O(log n)? (Aren't we adding and removing from a vector??)

Heapsort

- We can also use a priority queue (and a heap) as the underlying mechanism for sorting an array/vector of objects
- General idea:
  - Start with unsorted array/vector
  - Remove elements and insert into heap one at a time, running heapify on each step
  - When no elements are left in array, remove from heap and add to array in sorted order
- Example

Heapsort

- What is the runtime of heapsort?
  - n insertions at O(log n) each: O(n log n)
  - n removals at O(log n) each: O(n log n)
- So overall heapsort takes O(n log n)
- But this is usually 3-4x slower than QuickSort!
  - this == Lame

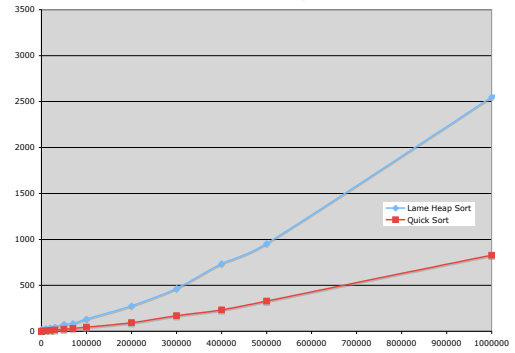
Lame Heap Sort

```
int v[] = ...;

VectorHeap<Integer> h = new VectorHeap<Integer>();
for (int i = 0; i < v.length; i++) {
  h.add(v[i]);
}

for (int i = 0; i < v.length; i++) {
  v[i] = h.remove();
}
```

Lame Heap Sort

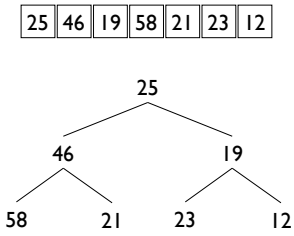


Can We Do Better?

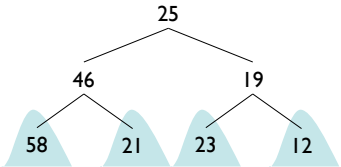
- Just make unordered array a heap *without* running heapify (percolateUp) on each addition
- Treat unsorted array as broken heap

- Leaves of tree are already heaps
- We just need to fix the rest…

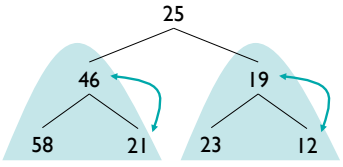
Better Heap Sort



Better Heap Sort

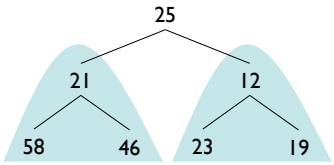


Better Heap Sort



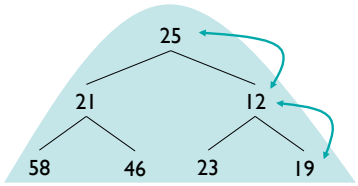
- We need to fix the subtrees rooted at 46 and 19
- Call pushDownRoot on 46, 19

Better Heap Sort



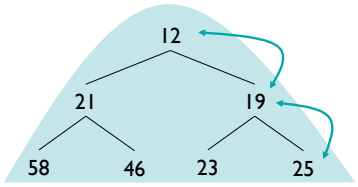
- Now subtrees at 21 and 12 are valid heaps

Better Heap Sort



- We need to fix the subtrees rooted at 25
- Call pushDownRoot on 25

Better Heap Sort



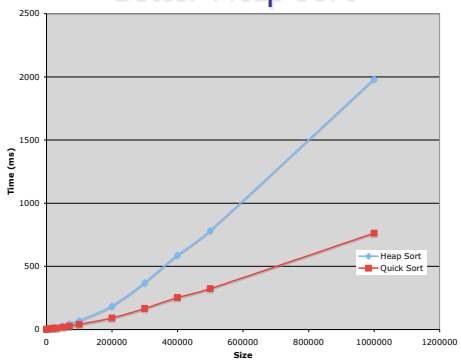
- Now we have a valid heap!

Better Heap Sort

```
class VectorHeap<E extends Comparable<E>>
   implements PriorityQueue<E> {

  protected Vector<E> data;
  ...
  public VectorHeap(E v[]) {
    data = new Vector<E>();
    for (int i = 0; i < v.length; i++) {
      data.add(v[i]);
    }
    for (int i = data.size()/2 - 1; i >= 0; i--) {
      //why data.size()/2??
      pushDownRoot(i);
    }
  }
  ...
}
```

Better Heap Sort



Better Heap Sort

- So still slower than QuickSort, but only 2x slower
- Cost?
  - You can create heap from array in O(n) time
    - Proof is left as an exercise…
    - (It's a little tricky)
  - But still need O(n log n) to remove
  - So total cost is still O(n log n)
- Any questions?

Why Heapsort?

- Heapsort is slower than Quicksort in general
- Any benefits to heapsort?
  - *Guaranteed* O(n log n) runtime
  - Constant space overhead
- Works well on mostly sorted data, unlike quicksort
- Good for incremental sorting

Tree Wrapup

- General Binary Trees
  - Express hierarchical relationships
  - "Ordering" is based on some external notion
    - i.e., ancestry, game boards, decisions, etc.
- Heap
  - Partially ordered (complete) binary tree based on priorities (highest priority node is root)
  - Node invariants: parent has higher priority than both children