

# Lecture 18: Heaps & Heapsort

CS 62  
Spring 2013  
Kim Bruce & Kevin Coogan

## Exam Next Monday

- In class: 50 minutes
- Covers everything through Splay trees
- Studying essential
  - Form study groups
  - Do problems from sample exams (on web page)
  - Do problems from text
- No Quiz Friday

## Exam Topics

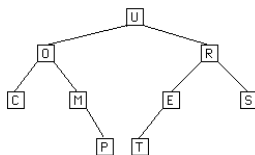
- Pre and post-conditions
- ArrayLists
- Java Graphics/GUI
- Analysis of Algorithms: Big-O
- Induction/Sorting
- Iterators
- Linked Lists
- Stacks
- Queues
- Trees
- Binary Search Trees/  
Splay trees

## Next Project: Darwin

- Learn JUnit in lab
- Darwin: Program creatures w/zombie-like behavior!
- Final version due in 1 1/2 weeks
  - Part 1 due Friday.
  - Contest

## Array Representation

- `data[0..n-1]` can hold values in trees
  - left subtree of node  $i$  in  $2*i+1$ , right in  $2*i+2$ ,
  - parent in  $(i-1)/2$



Indices: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14  
 data[]: U O R C M E S - - - P T - - -

## Min-Heap

- Min-Heap  $H$  is complete binary tree s.t.
  - $H$  is empty, or
  - Both of the following hold:
    - The value in root position is smallest value in  $H$
    - The left and right subtrees of  $H$  are also heaps.  
*Equivalent to saying parent  $\leq$  both left and right children*
- Excellent implementation for priority queue
  - Dequeue elements w/lowest priority values before higher

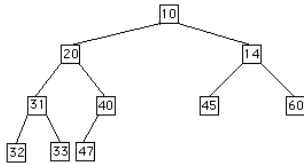
## PriorityQueue

```
public interface PriorityQueue<E extends Comparable<E>>
{
    /**
     * @pre !isEmpty()
     * @return The minimum value in the queue.
     */
    public E remove();
    public E getFirst();
    public void add(E value);
    public boolean isEmpty();
    public int size();
    public void clear();
}
```

## Implementations

- As regular queue (array or linked) where either keep in order or search for lowest to remove:
  - One of add or remove will be  $O(n)$
- Heap representation (in arraylist) is more efficient:  $O(\log n)$  for both add and remove.
  - Insert into heap:
    - Place in next free position,
    - "Percolate" it up.
  - Delete:
    - remove root,
    - move smallest child up to fill gaps, repeat

Insert 15:



IndexRange: 0 1 2 3 4 5 6 7 8 9 10  
 data: 10 20 14 31 40 45 60 32 33 47 -

IndexRange: 0 1 2 3 4 5 6 7 8 9 10  
 data: 10 20 14 31 40 45 60 32 33 47 15

IndexRange: 0 1 2 3 4 5 6 7 8 9 10  
 data: 10 20 14 31 15 45 60 32 33 47 40

IndexRange: 0 1 2 3 4 5 6 7 8 9 10  
 data: 10 15 14 31 20 45 60 32 33 47 40

See VectorHeap code

## Sorting with Trees

## Tree Sort

- Build Binary search tree (later)
- Do Inorder traversal, adding elts to array
  - Inorder traversal:  $O(n)$
  - Building tree:
    - $\log 1 + \log 2 + \dots + \log n = O(n \log n)$  in best (& average) case
    - $O(n^2)$  in worst case
- $O(n \log n)$  in best & average case
- $O(n^2)$  in worst case :- ( *What is worst case?*
- Heapsort is always better!

## Heapsort

- Make vector into a heap:
  - $n$  add operations =  $O(n \log n)$
- Remove elements in order
  - $n$  remove operations =  $O(n \log n)$
- Total:  $O(n \log n)$ 
  - If clever can make into heap in  $O(n)$
  - ... but still  $O(n \log n)$  total.

## Comparing Sorts

- Quicksort: fastest on average  $O(n \log n)$ , but worst case is  $O(n^2)$  & takes  $O(\log n)$  extra space
- Heapsort:  $O(n \log n)$  in average & worst case. No extra space.
  - Bit slower on average than quick & mergesorts.
- Mergesort:  $O(n \log n)$  in average and worst case.  $O(n)$  extra space.
  - Performs well on external files where not all fit in memory.

## Binary Search Trees

## BST

- A binary tree is a binary search tree iff
  - it is empty or
  - if the value of every node is both greater than or equal to every value in its left subtree and less than or equal to every value in its right subtree.

## Implementation

- Focus on trickiest methods:
  - add, get, & remove
  - protected methods: locate, predecessor, and removeTop

```
// @pre root and value are non-null
// @post returned: 1 - existing tree node with the desired value, or
//                2 - the node to which value should be added
protected BinaryTree<E> locate(BinaryTree<E> root, E value){
    E rootValue = root.value();
    BinaryTree<E> child;
    if (rootValue.equals(value)) return root; // found at root
    // look left if less-than, right if greater-than
    if (ordering.compare(rootValue,value) < 0) {
        child = root.right();
    } else {
        child = root.left();
    }
    // no child there: not in tree, return this node,
    // else keep searching
    if (child.isEmpty()) {
        return root;
    } else {
        return locate(child, value);
    }
}
```