CS 310: Priority Queues and Binary Heaps

Chris Kauffman

Week 14-2

Priority Queues

Queue

What operations does a queue support?

Priority: Number representing importance

- Convention lower is better priority
  Bring back life form. Priority One. All other priorities rescinded.
- Symmetric code if higher is better

Priority Queue (PQ): Supports 3 operations

- `void insert(T x,int p)`: Insert `x` with priority `p`
- `T findMin()`: Return the object with the best priority
- `void deleteMin()`: Remove the the object with the best priority

Priority

Explicit Priority

`insert(T x, int p)`

- Priority is explicitly `int p`
- Separate from data

Implicit Priority

`insert(Comparable<T> x)`

- `x` "knows" its own priority
- Comparisons dictated by `x.compareTo(y)`

Implicit is simpler for discussion: only one thing (`x`) to draw
Explicit usually uses a wrapper node of sorts

```
class PQElem<T> extends Comparable<PQElem>{
  int priority;    T elem;
  public int compareTo(PQElem<T> that){
    return this.priority - that.priority;
  }
}
```

Exercise: Design a PQ

Discuss

- How would you design `PriorityQueue` class?
- What underlying data structures would you use?
- Discuss with a neighbor
- Give rough idea of implementation
- Make it as efficient as possible in Big-O sense

Must Implement

- Constructor
- `void insert(T x)`: Insert `x`, knows its own priority, compare via `x.compareTo(y)`
- `T findMin()`: Return the object with the best priority
- `void deleteMin()`: Remove the the object with the best priority, re-arrange if needed

Exercise Results

Priority Queue implementations based on existing data structures

Data Structure	insert(x)	findMin()	deleteMin()	Notes
Sorted Array	O(N)	O(1)	O(1)	Keep min at high index
Sorted Linked List	O(N)	O(1)	O(1)	Min at head or tail
Binary Search Tree	O(height)	O(height)	O(height)	Min at left-most
AVL or Red-Black Tree	O(log N)	O(log N)	O(log N)	Height $\approx$ log N
Hash Table	O(N)	O(1)	O(1)	Tricky and pointless

Questions

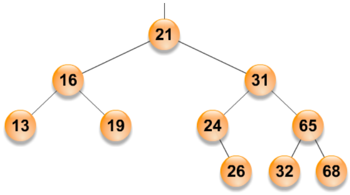
- Do priority queues implement more or less functionality than Binary Search Trees?
- Can we do better than these operational complexities?

Binary Heap: Sort of Sorted

- Most common way to build a PQ is using a new-ish data structure, the Binary Heap.
- Looks similar to a Binary Search Tree but maintains a different property

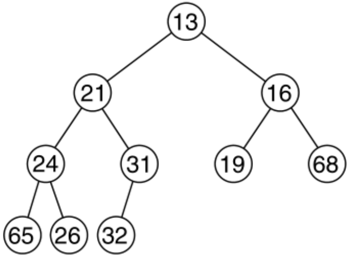
BST Property

A Node must be bigger than its left children and smaller than its right children

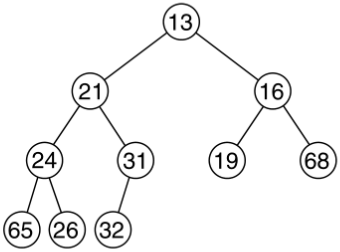


Binary Min-Heap Property

A Node must be smaller than its children



Heap and Not Heap

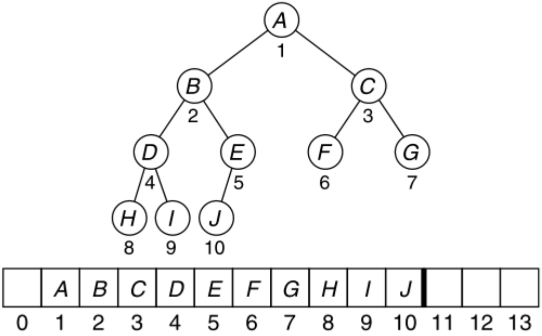


(a) (b)

Which of these is a min-heap and which is not?

Trees and Heaps in Arrays

- Mostly we have used trees of linked Nodes
- Can also put trees/heaps in an array



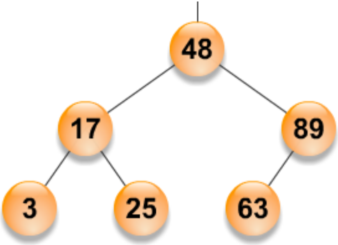
- Root is at 1 (discuss root at 0 later)
- `left(i) = 2*i`
- `right(i) = 2*i + 1`

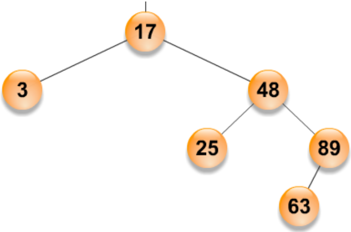
Balanced v. Unbalanced in Arrays

Find the array layout of these two trees

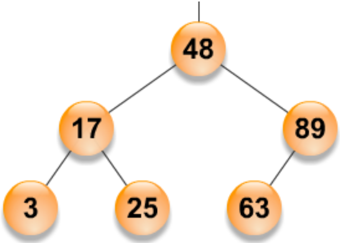
- Root is at 1
- `left(i) = 2*i`
- `right(i) = 2*i + 1`

Q: How big of array is required?

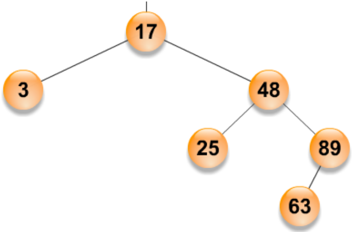




Balanced v. Unbalanced in Arrays



0	1	2	3	4	5	6
	48	17	89	3	25	63

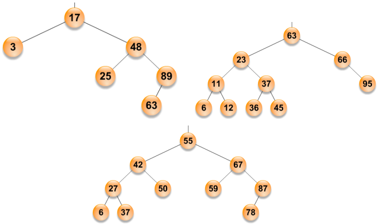


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	17	3	48			25	89							63

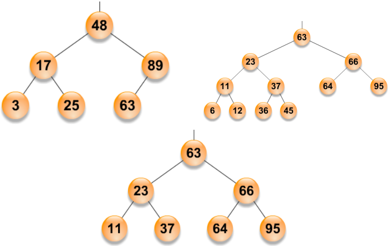
Complete Trees

- Only "missing" nodes in their bottom row (level set)
- Nodes in bottom row are as far left as possible

Not Complete (Why?)



Complete



- Complete trees don't waste space in arrays: no gaps
- Hard for general BSTs, easy for binary heaps. . .

Trees/Heaps in Array: Keep them Complete

- Storing in arrays: can cost space overhead
- If the tree is complete or nearly so, little wasted space

BSTs in arrays

- Hard to keep tree complete
- BST + balancing property makes it hard
- Rotations may not be constant time anymore
- Trees not usually laid out in arrays: linked nodes much more common

Binary Heaps in arrays

- Very easy to keep tree complete
- Heap Property is more loose, easier to maintain
- No rotations, no worries..
- Binary heaps almost always laid out in arrays

PQ Ops with Binary Heaps

- Use an internal `T array[]` of queue contents
- Maintaint min-heap order in `array`

Define

Tree-like ops for `array[]`

```
root()    => 1
left(i)   => i*2
right(i)  => i*2 + 1
parent(i) => i / 2
```

`T findMin()`

Super easy

```
return array[root()];
```

`insert(T x)`

Ensure heap is a complete tree

- Insert at next `array[size]`
- Increment size
- Percolate new element up

`deleteMin()`

Ensure heap is a complete tree

- Decrement size
- Replace root with last data
- Percolate root down

Demos of Binary Heaps

Not allowed on exams, but good for studying

Min Heap from David Galles @ Univ SanFran

- Visualize both heap and array version
- All ops supported

Max Heap from Steven Halim

- Good visuals
- No array
- Slow to load

Operations for Heaps

```
// Binary Heap, 1-indexed
public class BinaryHeapPQ<T>{
  private T [] array;
  private int size;

  // Helpers
  static int root(){
    return 1;
  }
  static int left(int i){
    return i*2;
  }
  static int right(int i){
    return i*2+1;
  }
  static int parent(int i){
    return i / 2;
  }
```

```
// Insert a data
public void insert(T x){
  size++;
  ensureCapacity(size+1);
  array[size] = x;
  percolateUp(size);
}
// Remove the minimum element
public void deleteMin(){
  array[root()] = array[size];
  array[size] = null;
  size--;
  percolateDown(root());
}
```

Percolate Up/Down

Up

```
void percolateUp(int xdx){
  while(xdx!=root()){
    T x = array[xdx];
    T p = array[parent(xdx)];
    if(doCompare(x,p) < 0){
      array[xdx] = p;
      array[parent(xdx)] = x;
      xdx = parent(xdx);
    }
    else{ break; }
  }
}
```

Down

```
void percolateDown(int xdx){
  while(true){
    T x = array[xdx];
    int cdx = left(xdx);
    // Determine which child
    // if any to swap with
    if(cdx > size){ break; } // No left, bottom
    if(right(xdx) < size &&  // Right valid
       doCompare(array[right(xdx)], array[cdx])
      cdx = right(xdx);      // Right smaller
    }
    T child = array[cdx];
    if(doCompare(child,x) < 0){ // child smaller
      array[cdx] = x;        // swap
      array[xdx] = child;
      xdx = cdx;             // reset index
    }
    else{ break; }
  }
}
```

PQ/Binary Heap Code

`BinaryHeapPQ.java`

- Code distribution today contains working heap
- `percolateUp()` and `percolateDown()` do most of the work
- Uses "root at index 1" convention

Text Book Binary Heap

- Weiss uses a different approach in percolate up/down
- Move a "hole" around rather than swapping
- Probably saves 1 comparison per loop iteration
- Have a look in `weiss/util/PriorityQueue.java`

Complexity of Binary Heap PQ methods?

```
T findMin();
void insert(T x); // x knows its priority
void deleteMin();
```

Give the complexity and justify for each

Height Again…

Efficiency of Binary Heap PQs

`findMin()` clearly $O(1)$

`deleteMin()` worst case height

`insert(x)` worst case height

Height of a Complete Binary Tree wrt number of nodes N?

- Guesses?
- Do some googling if you are feeling cowardly…

This is how rumors get started

Weiss's Assertion: `insert` is $O(1)$

On average the percolation [up] terminates early: It has been shown that 2.6 comparisons are required on average to perform the add, so the average add moves an element up 1.6 levels.

- Weiss 4th ed pg 815

Precise results on the number of comparisons and data movements used by heapsort in the best, worst, and average case are given in (Schaffer and Sedgewick).

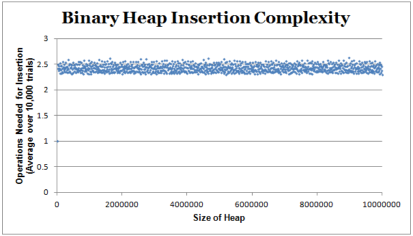
- pg 839
- Schaffer and R. Sedgewick, "The Analysis of Heapsort," Journal of

Algorithms 14 (1993), 76–100.

And how rumors resolve

Binary Heaps DO have $O(1)$ amortized insertion

Empirical Evaluation



Mathematical Proof

- Schaffer/Sedgewick (1991)
- $\rightarrow$ Carlsson (1987)
- $\rightarrow$ Porter/Simon (1974): "note pg. 15 in which the average case is stated to be upper-bound by lambda, which computes to 1.606691524"

Students in Fall 2013 did the above, edited Wikipedia to reflect this fact. You should have been there. It was awesome.

Summary of Binary Heaps

Op	Worst Case	Avg Case
`findMin()`	$O(1)$	$O(1)$
`insert(x)`	$O(\log N)$	$O(1)$
`deleteMin()`	$O(\log N)$	$O(\log N)$

- Notice: No `get(x)` method or `remove(x)` methods
- These would involve searching the whole binary heap/priority queue if they did existed: $O(N)$