CSE332: Data Abstractions

Lecture 5: Binary Heaps, Continued

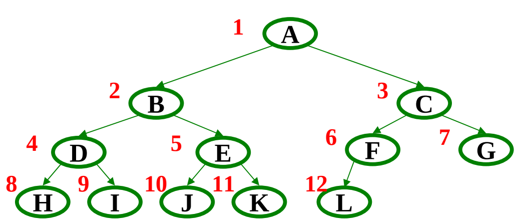
Dan Grossman

Spring 2010

Review

- Priority Queue ADT: **`insert`** comparable object, **`deleteMin`**
- Binary heap data structure: Complete binary tree where each node has priority value greater than its parent
- *O*(height-of-tree)=*O*(**`log`** *n*) **`insert`** and **`deleteMin`** operations
  - **`insert`**: put at new last position in tree and percolate-up
  - **`deleteMin`**: remove root, put last element at root and percolate-down
- But: tracking the "last position" is painful and we can do better

Array Representation of Binary Trees



From node **`i`**:

left child: **`i*2`**
right child: **`i*2+1`**
parent: **`i/2`**

(wasting index 0 is convenient)

implicit (array) implementation:

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Judging the array implementation

Plusses:

- Non-data space: just index 0 and unused space on right
  - In conventional tree representation, one edge per node (except for root), so *n*-1 wasted space (like linked lists)
  - Array would waste more space if tree were not complete
- For reasons you learn in CSE351 / CSE378, multiplying and dividing by 2 is very fast
- Last used position is just index **`size`**

Minuses:

- Same might-by-empty or might-get-full problems we saw with stacks and queues (resize by doubling as necessary)

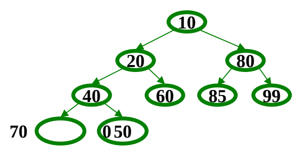
Plusses outweigh minuses: "this is how people do it"

Pseudocode: insert

Note this pseudocode inserts ints, not useful data with priorities

```
void insert(int val) {
  if(size==arr.length-1)
    resize();
  size++;
  i=percolateUp(size,val);
  arr[i] = val;
}
```

```
int percolateUp(int hole,
                int val) {
  while(hole > 1 &&
        val < arr[hole/2])
    arr[hole] = arr[hole/2];
    hole = hole / 2;
  }
  return hole;
}
```



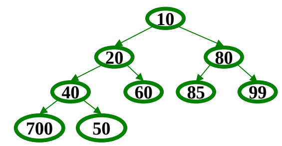
	10	20	80	40	60	85	99	700	50				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Pseudocode: deleteMin

Note this pseudocode deletes ints, not useful data with priorities

```
int deleteMin() {
  if(isEmpty()) throw…
  ans = arr[1];
  hole = percolateDown
           (1,arr[size]);
  arr[hole] = arr[size];
  size--;
  return ans;
}
```

```
int percolateDown(int hole,
                  int val) {
 while(2*hole <= size) {
  left  = 2*hole;
  right = left + 1;
  if(arr[left] < arr[right]
     || right > size)
    target = left;
  else
    target = right;
  if(arr[target] < val) {
    arr[hole] = arr[target];
    hole = target;
  } else
      break;
 }
 return hole;
}
```

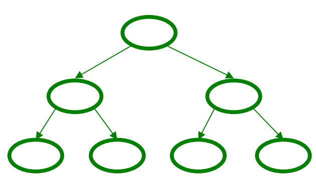


	10	20	80	40	60	85	99	700	50				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Example

1. insert: 16, 32, 4, 69, 105, 43, 2
2. deleteMin

0	1	2	3	4	5	6	7



Other operations

- **`decreaseKey`**: given pointer to object in priority queue (e.g., its array index), lower its priority value by *p*
  - Change priority and percolate up
- **`increaseKey`**: given pointer to object in priority queue (e.g., its array index), raise its priority value by *p*
  - Change priority and percolate down
- **`remove`**: given pointer to object, take it out of the queue
  - **`decreaseKey`** with $p = \infty$, then **`deleteMin`**

Running time for all these operations?

Build Heap

- Suppose you started with *n* items to put in a new priority queue
  - Call this the **`buildHeap`** operation
- **`create`**, followed by *n* **`insert`**s works
  - Only choice if ADT doesn't provide **`buildHeap`** explicitly
  - $O(n \log n)$
- Why would an ADT provide this unnecessary operation?
  - Convenience
  - Efficiency: an $O(n)$ algorithm called Floyd's Method
  - Common issue in ADT design: how many specialized operations

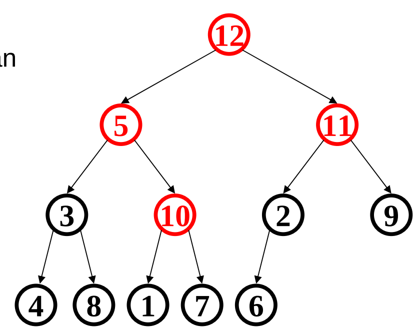
Floyd's Method

1. Use *n* items to make any complete tree you want
   - That is, put them in array indices 1,…,*n*
2. Treat it as a heap by fixing the heap-order property
   - Bottom-up: leaves are already in heap order, work up toward the root one level at a time

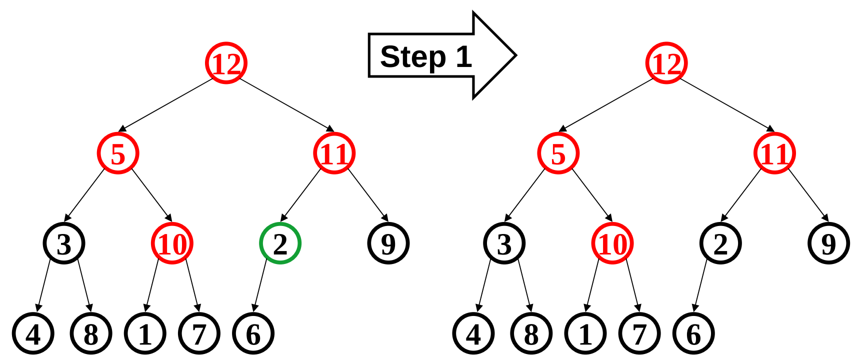
```
void buildHeap() {
  for(i = size/2; i>0; i--) {
    val  = arr[i];
    hole = percolateDown(i,val);
    arr[hole] = val;
  }
}
```


Example

- In tree form for readability
  - Red for node not less than descendants
    - heap-order problem
  - Notice no leaves are red
  - Check/fix each non-leaf bottom-up (6 steps here)

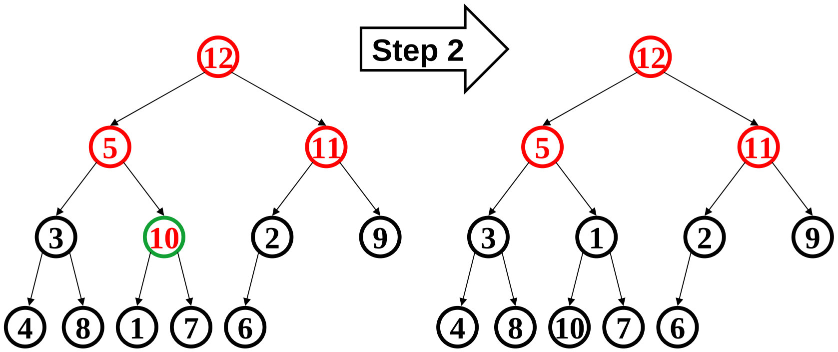


Example



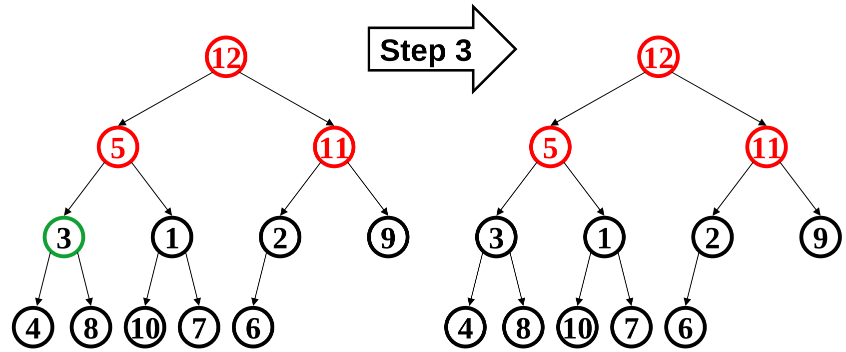
- Happens to already be less than children (er, child)

Example



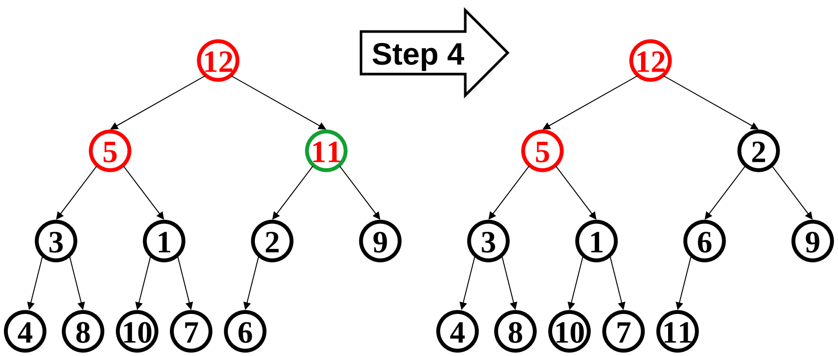
- Percolate down (notice that moves 1 up)

Example



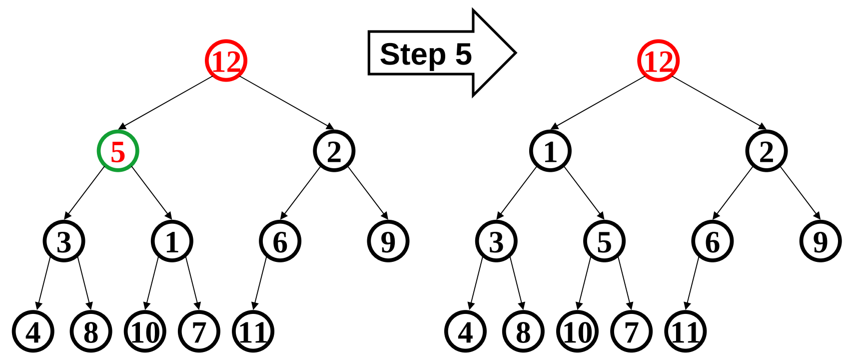
- Another nothing-to-do step

Example

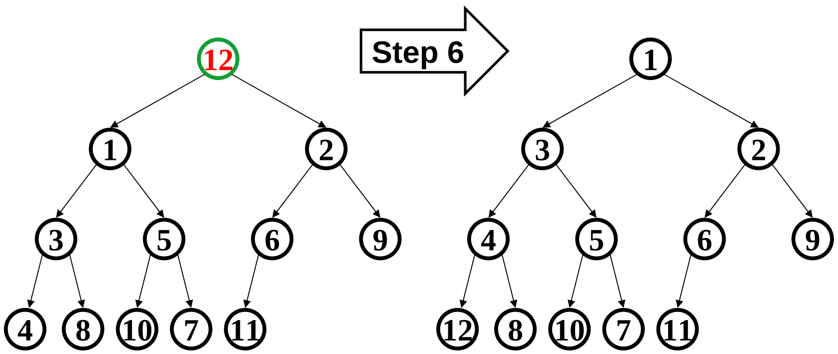


- Percolate down as necessary (steps 4a and 4b)

Example



Example



But is it right?

- "Seems to work"
  - Let's *prove* it restores the heap property (correctness)
  - Then let's *prove* its running time (efficiency)

```
void buildHeap() {
  for(i = size/2; i>0; i--) {
    val  = arr[i];
    hole = percolateDown(i,val);
    arr[hole] = val;
  }
}
```


Correctness

```
void buildHeap() {
  for(i = size/2; i>0; i--) {
    val  = arr[i];
    hole = percolateDown(i,val);
    arr[hole] = val;
  }
}
```

Loop Invariant: For all **j**>**i**, **arr[j]** is less than its children

- True initially: If **j > size/2**, then **j** is a leaf
  - Otherwise its left child would be at position > **size**
- True after one more iteration: loop body and **percolateDown** make **arr[i]** less than children without breaking the property for any descendants

So after the loop finishes, all nodes are less than their children

Efficiency

```
void buildHeap() {
  for(i = size/2; i>0; i--) {
    val  = arr[i];
    hole = percolateDown(i,val);
    arr[hole] = val;
  }
}
```

Easy argument: **buildHeap** is *O*(*n* **log** *n*) where *n* is **size**

- **size/2** loop iterations
- Each iteration does one **percolateDown**, each is *O*(**log** *n*)

This is correct, but there is a more precise ("tighter") analysis of the algorithm…

Efficiency

```
void buildHeap() {
  for(i = size/2; i>0; i--) {
    val  = arr[i];
    hole = percolateDown(i,val);
    arr[hole] = val;
  }
}
```

Better argument: **buildHeap** is $O(n)$ where n is **size**

- **size/2** total loop iterations: $O(n)$
- 1/2 the loop iterations percolate at most 1 step
- 1/4 the loop iterations percolate at most 2 steps
- 1/8 the loop iterations percolate at most 3 steps
- …
- ((1/2) + (2/4) + (3/8) + (4/16) + (5/32) + …) < 2 (page 4 of Weiss)
  - So at most **2(size/2)** *total* percolate steps: $O(n)$

Lessons from **buildHeap**

- Without **buildHeap**, our ADT already let clients implement their own in $\theta(n$ **log** $n)$ worst case
  - Worst case is inserting lower priority values later
- By providing a specialized operation internally (with access to the data structure), we can do $O(n)$ worst case
  - Intuition: Most data is near a leaf, so better to percolate down
- Can analyze this algorithm for:
  - Correctness: Non-trivial inductive proof using loop invariant
  - Efficiency:
    - First analysis easily proved it was O(n **log** n)
    - A "tighter" analysis shows same algorithm is $O(n)$

What we're skipping (see text if curious)

- *d*-heaps: have *d* children instead of 2
  - Makes heaps shallower, useful for heaps too big for memory
  - The same issue arises for balanced binary search trees and we *will* study "B-Trees"
- Different data structures for priority queues that support a logarithmic time **merge** operation (impossible with binary heaps)
  - **merge:** given two priority queues, make one priority queue
  - How might you merge binary heaps:
    - If one heap is much smaller than the other?
    - If both are about the same size?