Unit #2: Priority Queues

CPSC 221: Algorithms and Data Structures

Will Evans and Jan Manuch

2016W1

Unit Outline

- Rooted Trees, Briefly
- Priority Queue ADT
- Heaps
  - Implementing Priority Queue ADT
  - Focus on Create: Heapify
  - Brief introduction to *d*-Heaps

Learning Goals

- Provide examples of appropriate applications for priority queues and heaps
- Manipulate data in heaps
- Describe and apply the Heapify algorithm, and analyze its complexity

Rooted Trees

- Family Trees
- Organization Charts
- Classification trees (a.k.a. keys)
  - What kind of flower is this?
  - Is this mushroom poisonous?
- File directory structure
  - folders, subfolders in Windows
  - directories, subdirectories in UNIX
- Non-recursive call graphs

Tree Terminology

root:

leaf:

child:

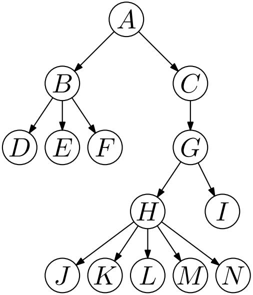
parent:

sibling:

ancestor:

descendent:

subtree:



Tree Terminology Reference

root: the single node with no parent

leaf: a node with no children

child: a node pointed to by me

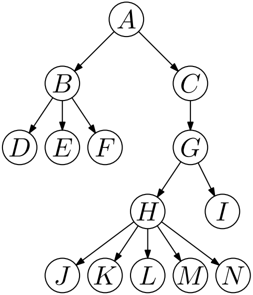
parent: the node that points to me

sibling: another child of my parent

ancestor: my parent or my parent's ancestor

descendent: my child or my child's descendent

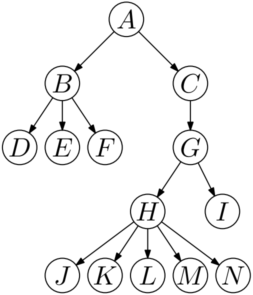
subtree: a node and its descendents



More Tree Terminology

depth: Number of edges on path from root to node

depth of H?

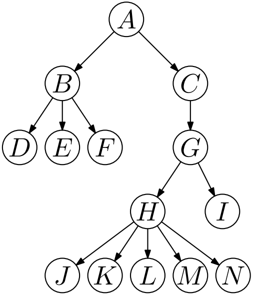


More Tree Terminology

height: Number of edges on longest path from node to descendent or, for whole tree, from root to leaf

height of tree?

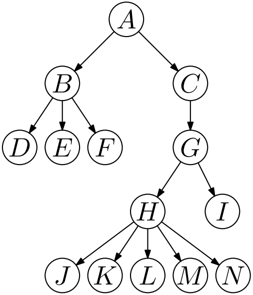
height of *G*?



More Tree Terminology

(downward) degree: Number of children of a node

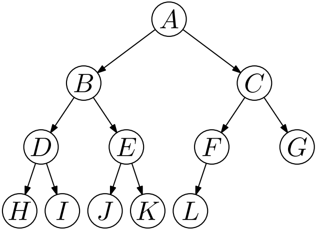
degree of B?



One More Tree Terminology Slide

binary: each node has degree at most 2

d-**ary:** degree at most *d*

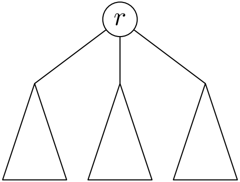


complete: as many nodes as possible for its height (each row filled in)

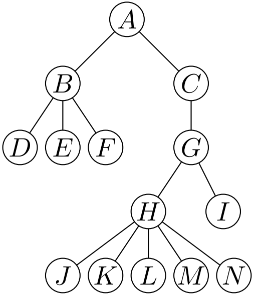
nearly complete: each row except the last one is filled in, all nodes in the last row are as far left as possible

Longest Path

Find the longest *undirected* path in a tree



Longest Path Example

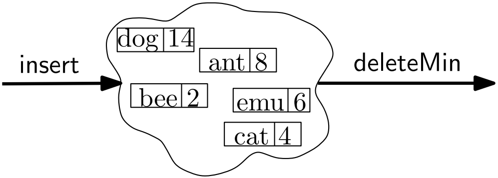


Back to Queues

- Applications
  - ordering CPU jobs
  - simulating events
  - picking the next search site
- But we don't want FIFO ...
  - *short* jobs should go first
  - *earliest* (simulated time) events should go first
  - *most promising* sites should be searched first

Priority Queue ADT

- Priority Queue operations
  - create
  - destroy
  - insert
  - deleteMin
  - is_empty



- Priority Queue property: For two elements in the queue, x and y, if x has a lower priority value than y, x will be deleted before y.

Applications of the Priority Q

- Hold jobs for a printer in order of length
- Store packets on network routers in order of urgency
- Simulate events
- Select symbols for compression
- Sort numbers
- Anything *greedy*: an algorithm that makes the "locally best choice" at each step

Priority Q Data Structures

- Unsorted list
  - insert time:
  - deleteMin time:
- Sorted list
  - insert time:
  - deleteMin time:

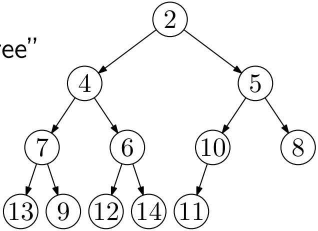
Binary Heap Priority Q Data Structure

Heap-order property: parent's key $\leq$ children's keys.

- minimum is always at the top

Structure property: "nearly complete tree"

- depth is always O(log n)
- next open location always known

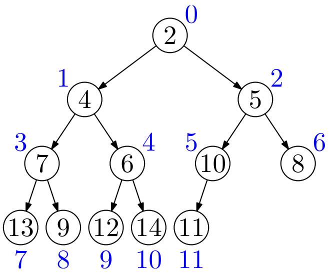


WARNING: This has NO SIMILARITY to the "heap" you hear about when people say "things you create with `new` go on the heap".

Nifty Storage Trick

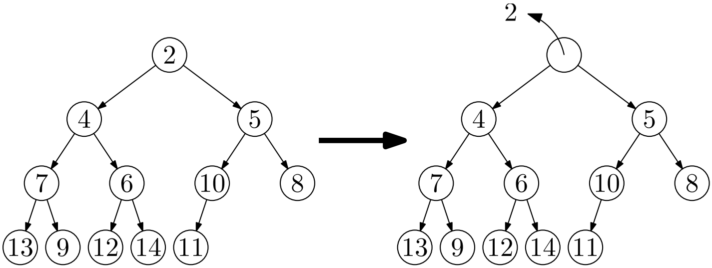
Navigation using indices:

- left_child(i) =
- right_child(i) =
- parent(i) =
- root =
- next free position =



0	1	2	3	4	5	6	7	8	9	10	11	12
2	4	5	7	6	10	8	13	9	12	14	11	

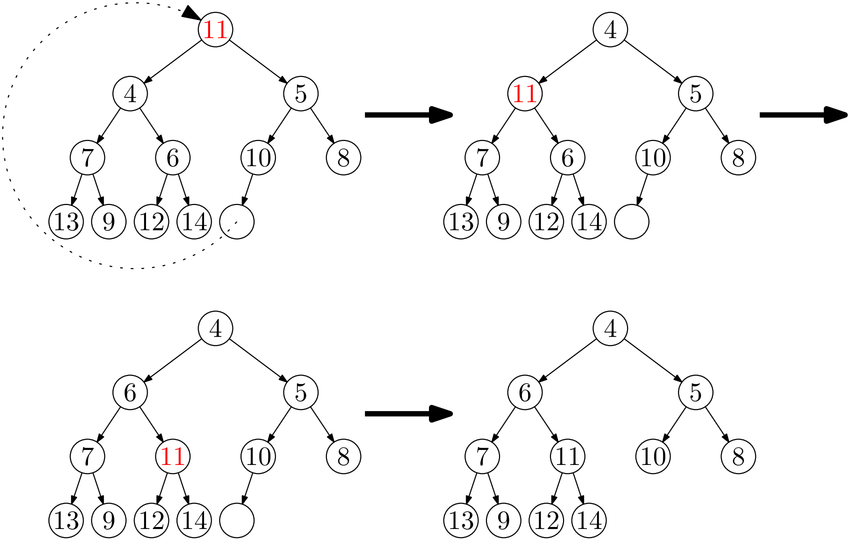
DeleteMin



Invariants violated! No longer “nearly complete”

Swap (Heapify) Down

Move last element to root then swap it down to its proper position.



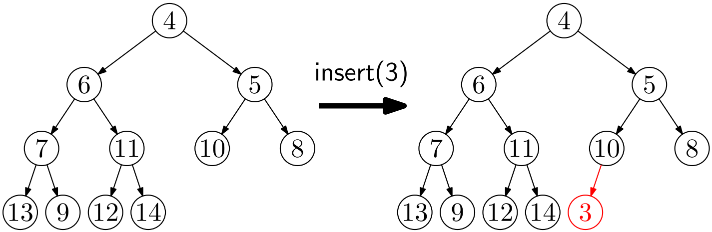
DeleteMin Code

```
int deleteMin() {
  assert(!isEmpty());
  int returnVal = Heap[0];
  Heap[0] = Heap[n-1];
  n--;
  swapDown(0);
  return returnVal;
}
```

Runtime:

```
void swapDown(int i) {
  int s = i;
  int left = i * 2 + 1;
  int right = left + 1;
  if( left < n &&
      Heap[left] < Heap[s] )
    s = left;
  if( right < n &&
      Heap[right] < Heap[s] )
    s = right;
  if( s != i ) {
    int tmp = Heap[i];
    Heap[i] = Heap[s];
    Heap[s] = tmp;
    swapDown(s);
  }
}
```

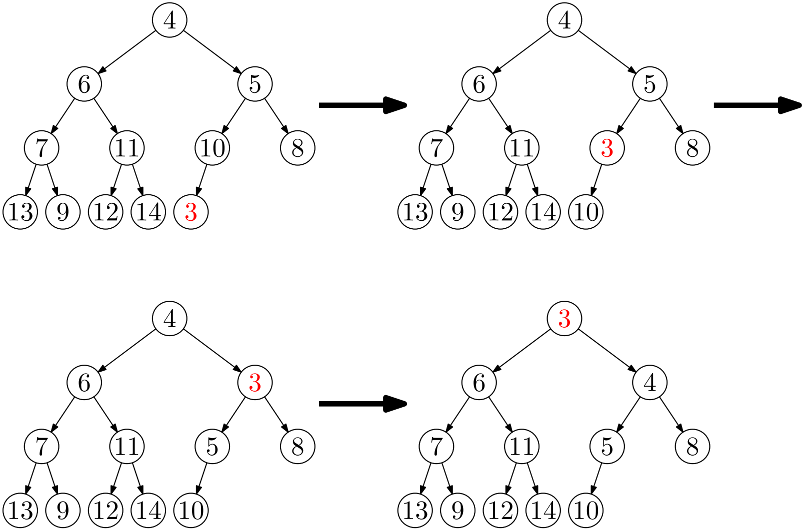
Insert



Invariant violated! Child has smaller key than parent.

Swap (Heapify) Up

Put new element last then swap it up to its proper position.



Insert Code

```
void insert(int x) {
  assert(!isFull());
  Heap[n] = x;
  n++;
  swapUp(n-1);
}
```

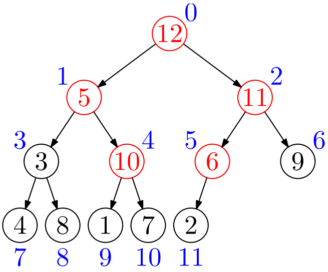
```
void swapUp(int i) {
  if( i == 0 ) return;
  int p = (i - 1)/2;
  if( Heap[i] < Heap[p] ) {
    int tmp = Heap[i];
    Heap[i] = Heap[p];
    Heap[p] = tmp;
    swapUp(p);
  }
}
```

Runtime:

Heapify: Build a Heap from a non-Heap Array

1. Start with the input array.

12	5	11	3	10	6	9	4	8	1	7	2

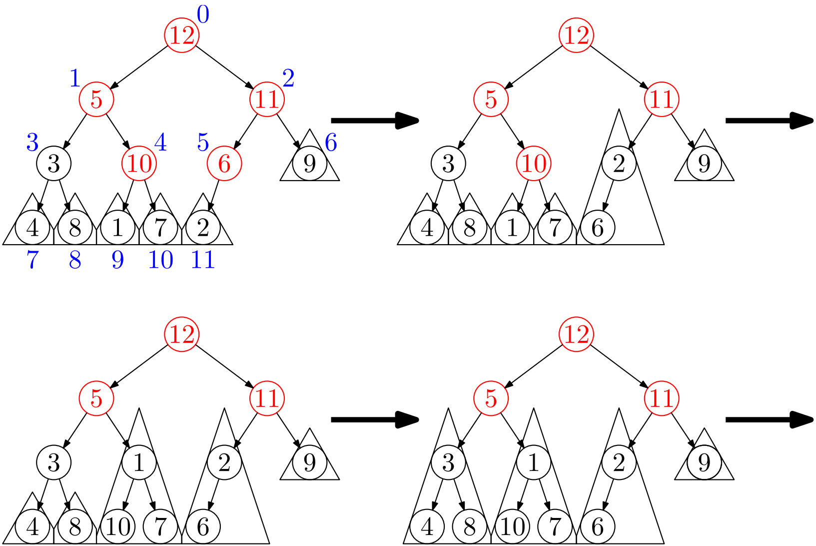


Invariant violated!

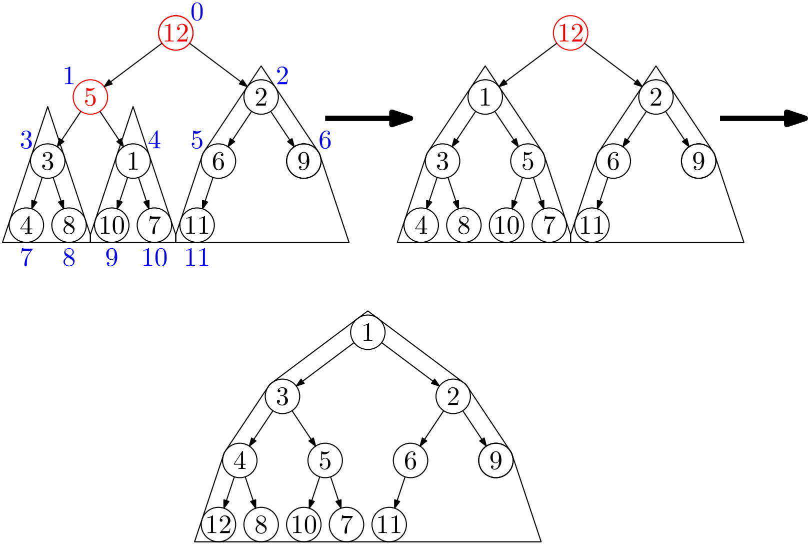
2. Fix the heap-order property bottom up. Use `swapDown`.

```
for( i=n/2-1; i >=0; i-- ) swapDown(i);
```

Heapify Example...

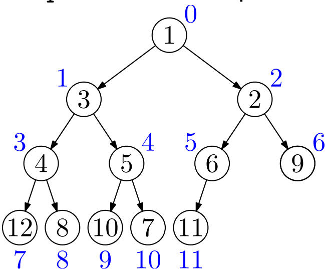


Heapify Example



Heapify Runtime

`swapDown` on a heap of height h takes at most ____________ steps.

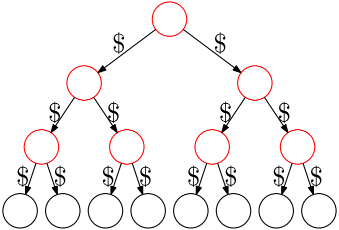


Let H be the height of the heap.

`swapDown` is called	once	on heap of height	H
	≤ 2 times	on heap of height	$H-1$
	≤ 4 times	on heap of height	$H-2$
	$\vdots$		
	$\leq 2^{H-1}$ times	on heap of height	1

Total # steps $\leq \sum_{h=1}^{H} h2^{H-h} = 2^H \sum_{h=1}^{H} h/2^h \leq 2^{H+1} = O(n)$

Heapify Runtime: Charging Scheme



Possible violations. How much time to fix them?
Place a dollar on each edge of the heap. One dollar pays for one step of `swapDown`. By induction, we can show that when `swapDown` is called on a node v, both children of v have a path (the rightmost path) to a leaf that is uncharged. The edges on the left child's rightmost path plus the edge to the left child pay for the steps of `swapDown` at v. The edges on the right child's rightmost path plus the edge to the right child form the uncharged path available to the parent of v.

Thinking about Binary Heaps

Observations

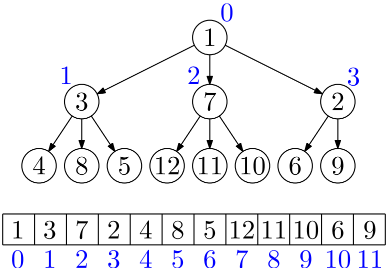
- finding a child/parent index is a multiply/divide by two
- deleteMin and insert access far-apart array locations
- deleteMin accesses all children of visited nodes
- insert accesses only parent of visited nodes
- insert is at least as common as deleteMin

Realities

- division and multiplication by powers of two are fast
- far-apart array accesses ruin cache performance
- with huge data sets, disk I/O dominates

Solution: *d*-Heaps

Nearly complete *d*-ary trees (representable by array) with Heap-order property.



Good choices for *d*:

- fit one set of children on a memory page/disk block
- fit one set of children in a cache line
- optimize performance based on ratio of inserts/deleteMins
- make *d* a power of two for efficiency

d-Heap Navigation

- jth-child(i) $=$
- parent(i) $=$
- root $=$
- next free position $=$