CS221: Algorithms and Data Structures

Priority Queues and Heaps

Alan J. Hu

(Borrowing slides from Steve Wolfman)

Learning Goals

After this unit, you should be able to:

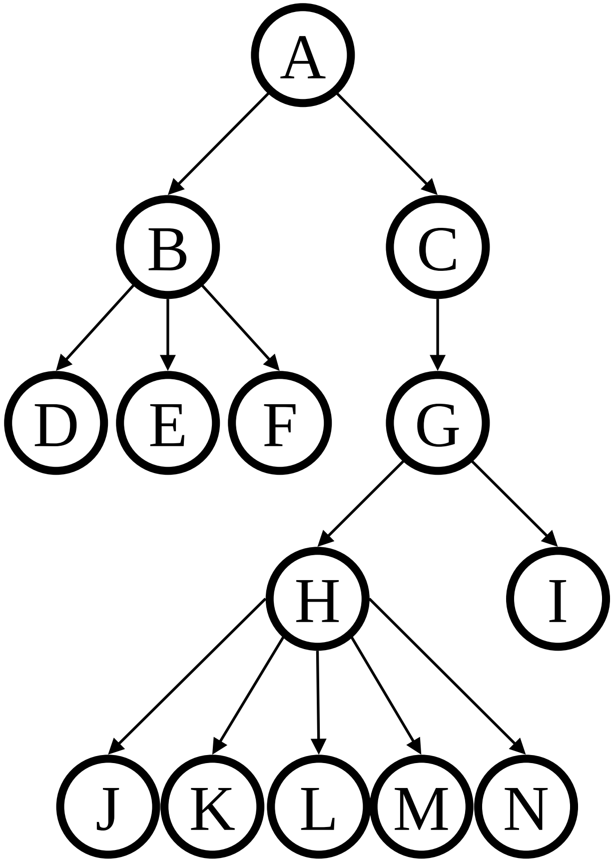
- Provide examples of appropriate applications for priority queues and heaps
- Manipulate data in heaps
- Describe and apply the Heapify algorithm, and analyze its complexity

Today's Outline

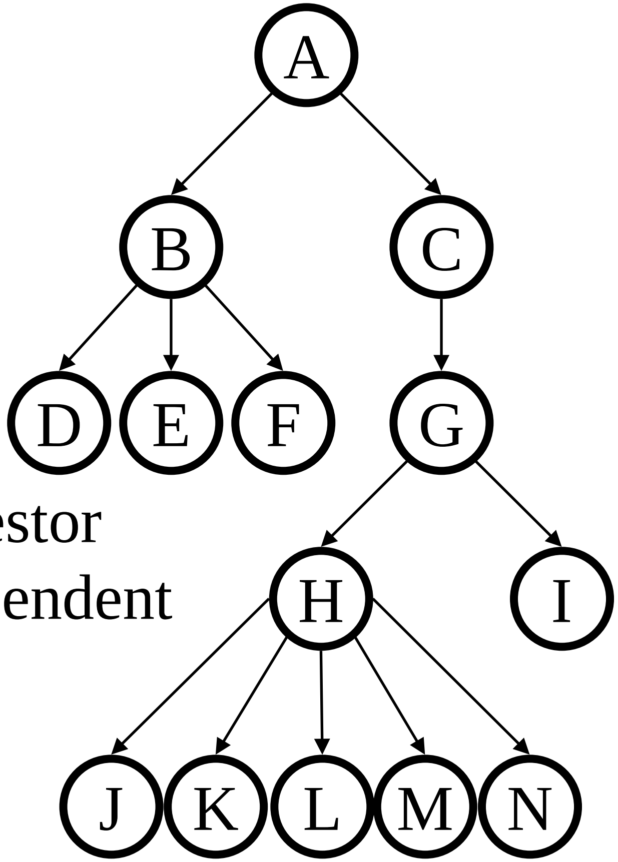
- Trees, Briefly
- Priority Queue ADT
- Heaps
  - Implementing Priority Queue ADT
  - Focus on Create: Heapify
  - Brief introduction to d-Heaps

Tree Terminology

root:
leaf:
child:
parent:
sibling:
ancestor:
descendent:
subtree:



Tree Terminology Reference



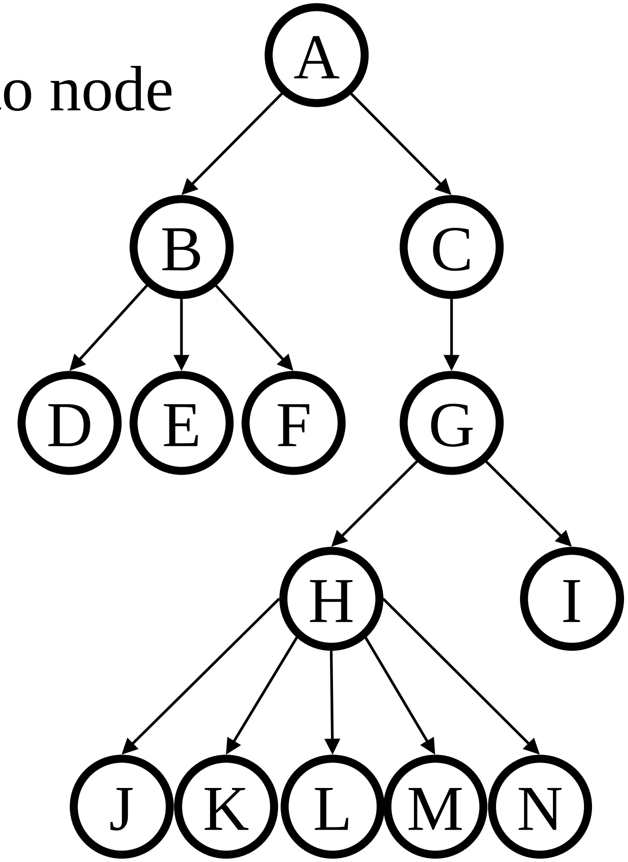
root: the single node with no parent
leaf: a node with no children
child: a node pointed to by me
parent: the node that points to me
sibling: another child of my parent
ancestor: my parent or my parent's ancestor
descendent: my child or my child's descendent
subtree: a node and its descendents

We sometimes use degenerate versions of these definitions that allow NULL as the empty tree. (This can be *very* handy for recursive base cases!)

More Tree Terminology

depth: # of edges along path from root to node

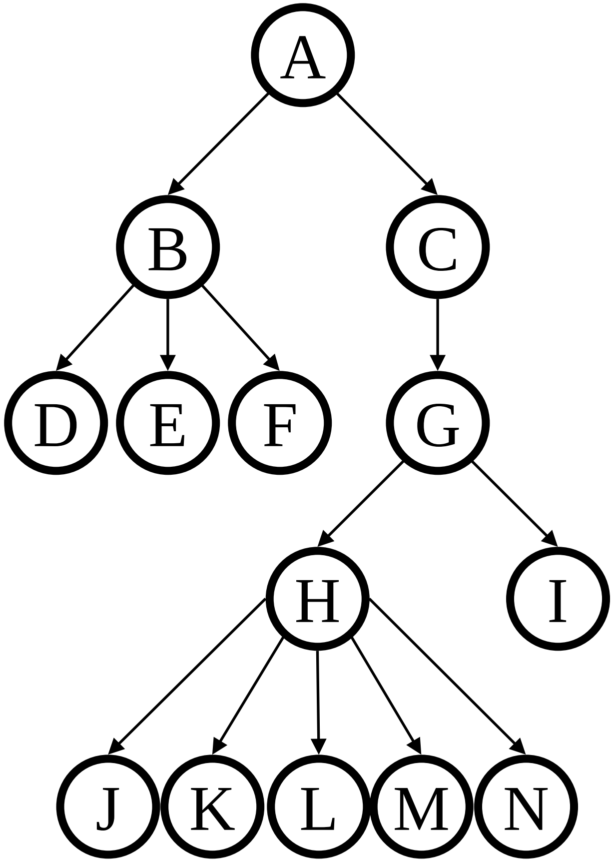
depth of H?



More Tree Terminology

height: # of edges along longest path from node to leaf or, for whole tree, from root to leaf

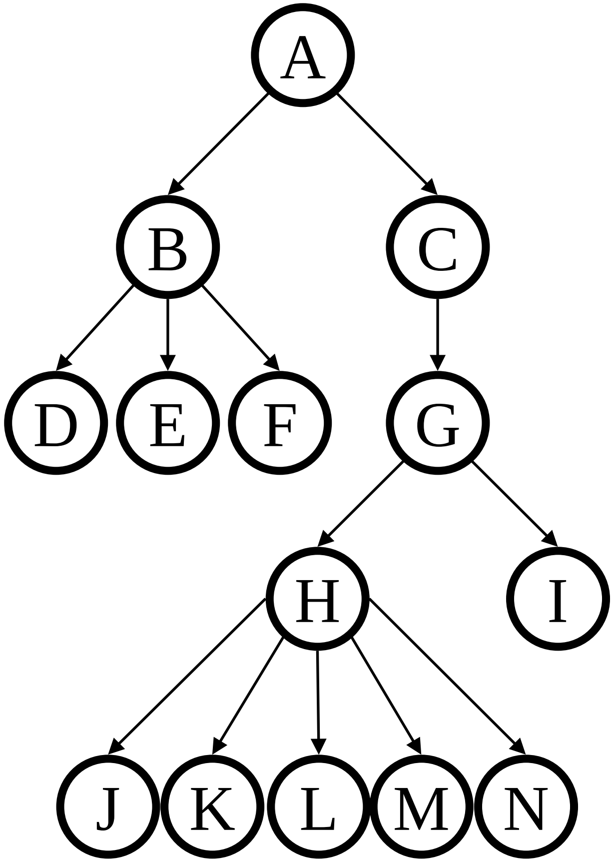
height of tree?



More Tree Terminology

degree: # of children of a node

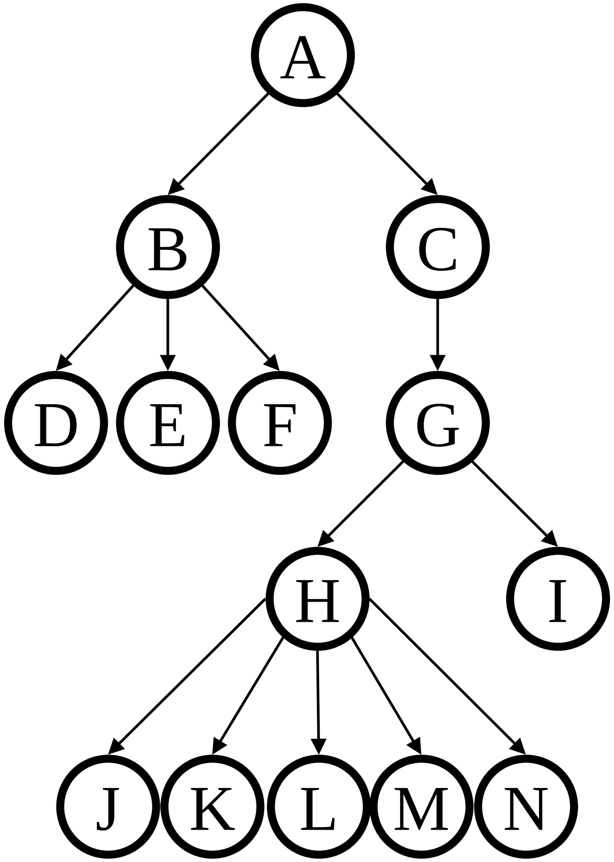
degree of B?



More Tree Terminology

branching factor: maximum degree of any node in the tree

2 for binary trees,
our usual concern;
5 for this weird tree

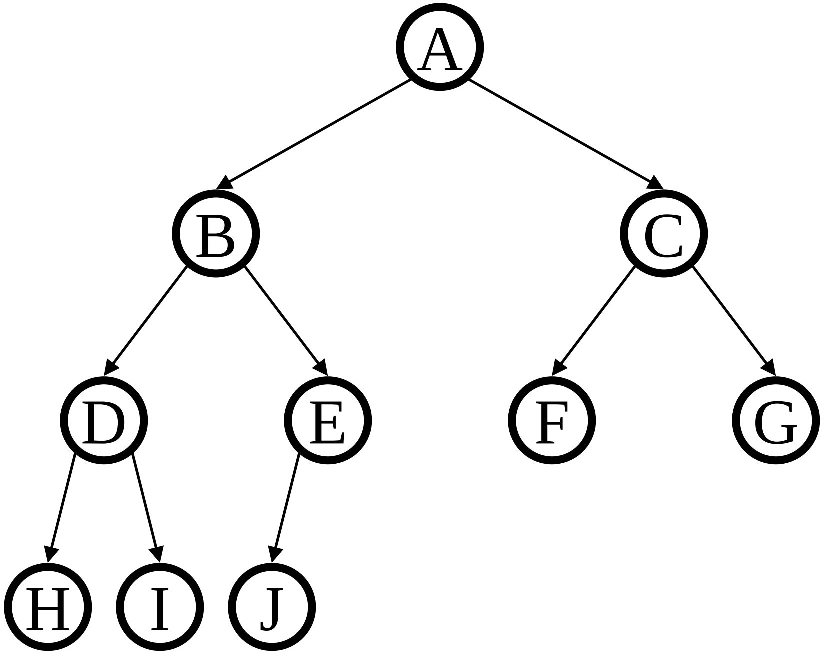


One More Tree Terminology Slide

binary: branching factor of 2 (each child has at most 2 children)

n-ary: branching factor of n

complete: "packed" binary tree;
as many nodes as
possible for its height



nearly complete: complete plus some nodes on the left at the bottom

Trees and (Structural) Recursion

A tree is either:

- the empty tree
- a root node and an ordered list of subtrees

Trees are a recursively defined structure, so it makes sense to operate on them recursively.

Today's Outline

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

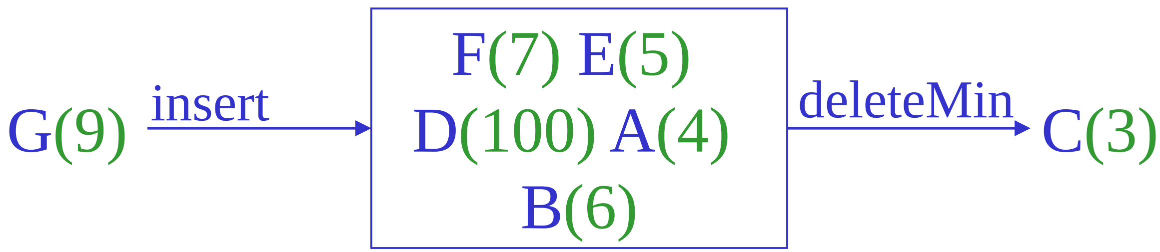
Back to Queues

- Some applications
  - ordering CPU jobs
  - simulating events
  - picking the next search site
- Problems?
  - short jobs **should go first**
  - earliest (simulated time) events **should go first**
  - most promising sites **should be searched first**

Remember ADTs?

Priority Queue ADT

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



- 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

Naïve Priority Q Data Structures

- Unsorted list:
  - *insert:*
  - *deleteMin:*
- Sorted list:
  - *insert:*
  - *deleteMin:*

a. O(lg n)
b. O(n)
c. O(n lg n)
d. $O(n^2)$
e. Something else

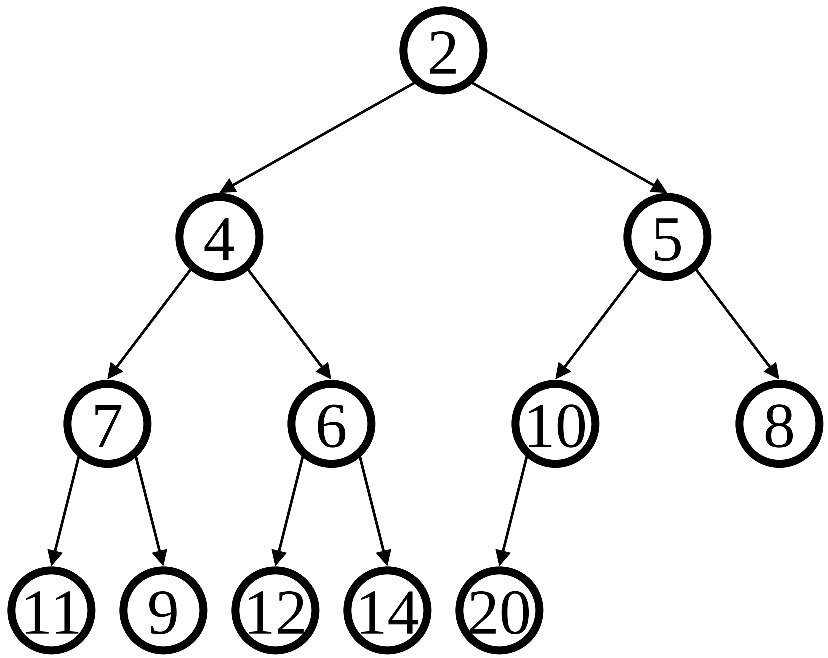
Today's Outline

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

Binary Heap Priority Q Data Structure

- Heap-order property
  - parent's key is less than or equal to children's keys
  - result: minimum is always at the top
- Structure property
  - "nearly complete tree"
  - result: depth is always O(log n); next open location always known

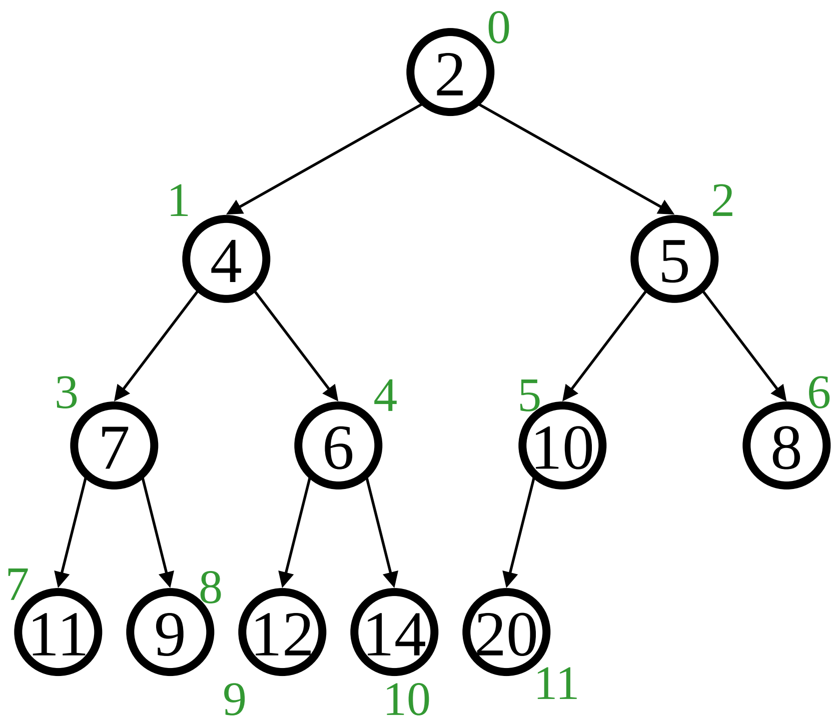
Look! Invariants!



WARNING: this has *NO SIMILARITY* to the "heap" you hear about when people say "objects you create with **new** go on the heap".

Nifty Storage Trick

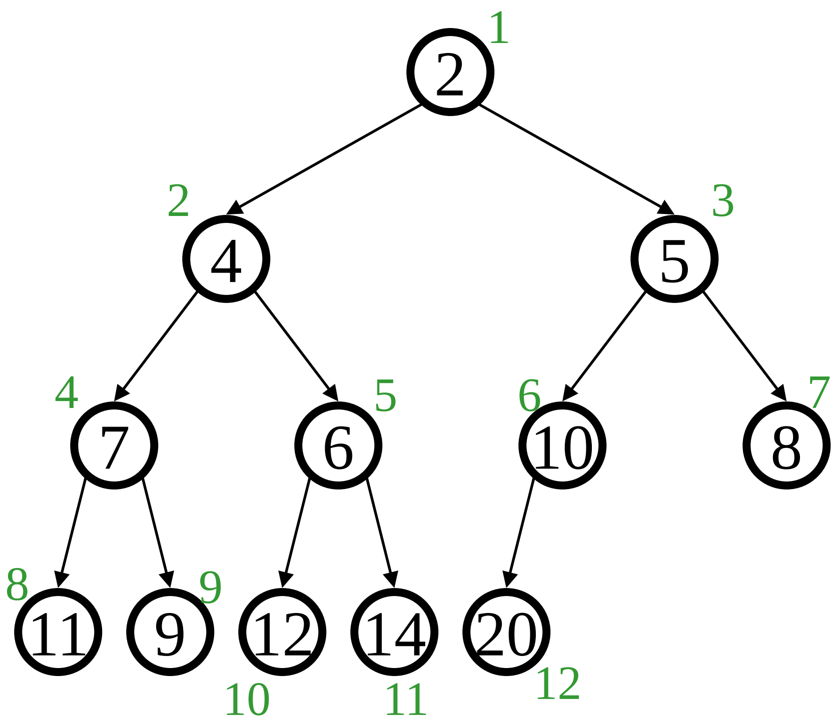
- Calculations:
  - child:
  - parent:
  - root:
  - next free:



0	1	2	3	4	5	6	7	8	9	10	11	
2	4	5	7	6	10	8	11	9	12	14	20	

(Aside: Steve numbers from 1.)

- Calculations:
  - child:
  - parent:
  - root:
  - next free:

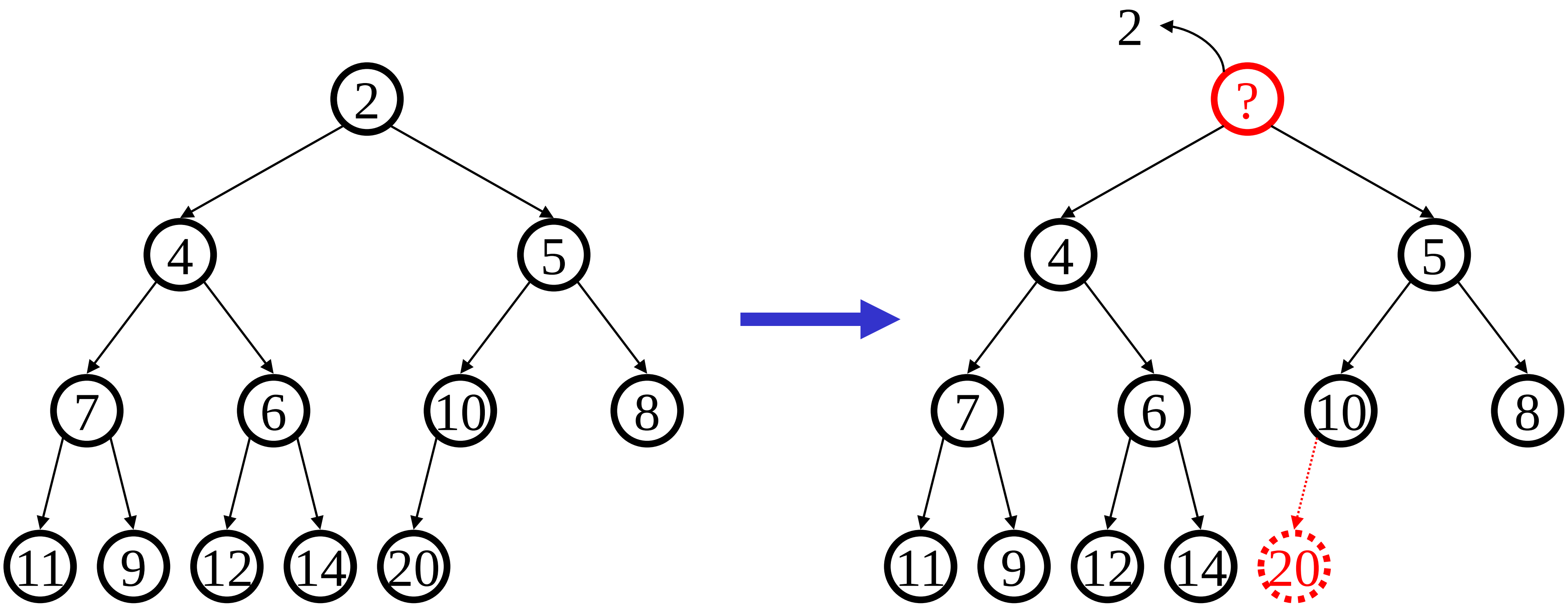


0	1	2	3	4	5	6	7	8	9	10	11	12	
	2	4	5	7	6	10	8	11	9	12	14	20	

Steve like to just skip using entry 0 in the array, so the root is at index 1.
For a binary heap, this makes the calculations slightly shorter.

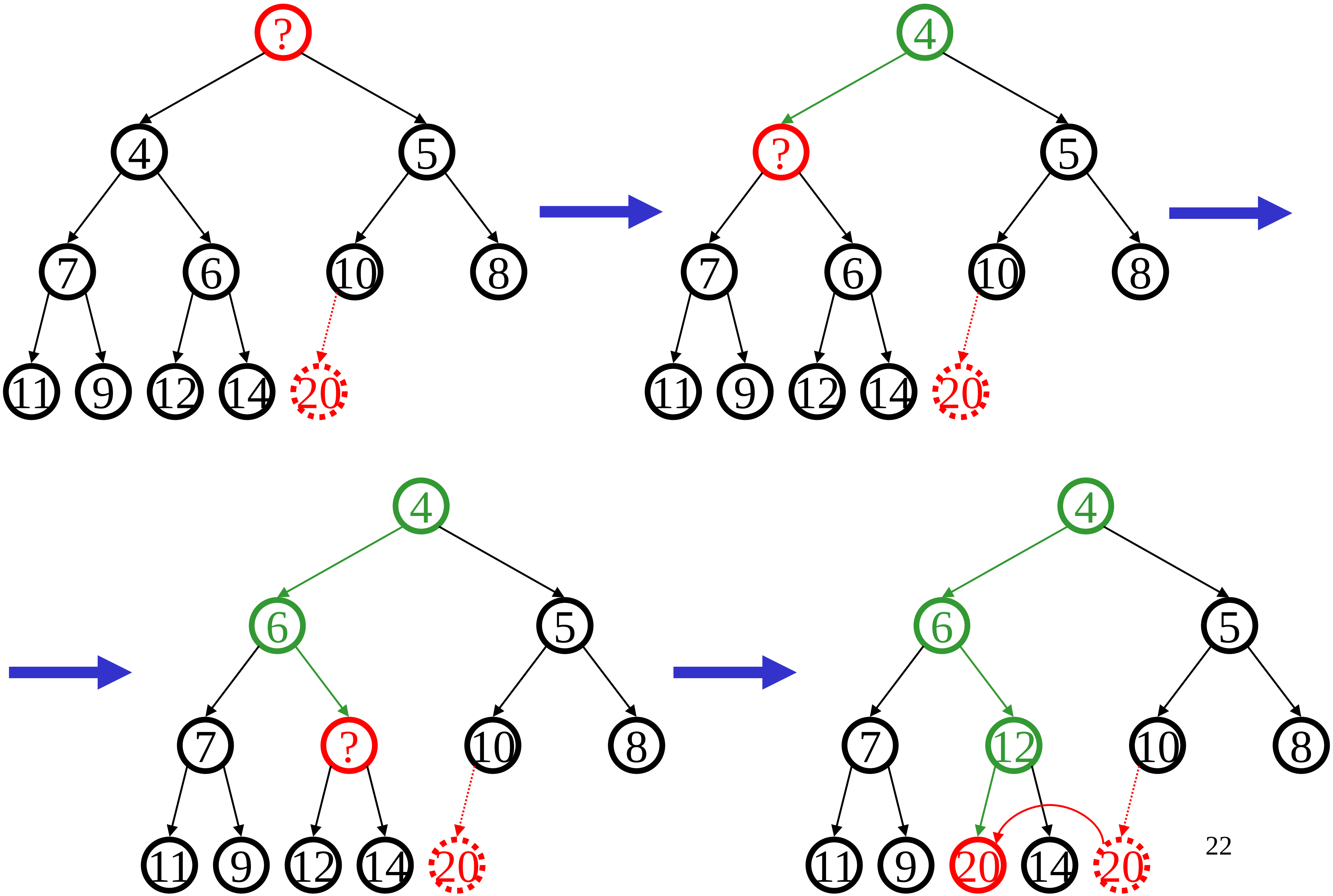
DeleteMin

`pqueue.deleteMin()`

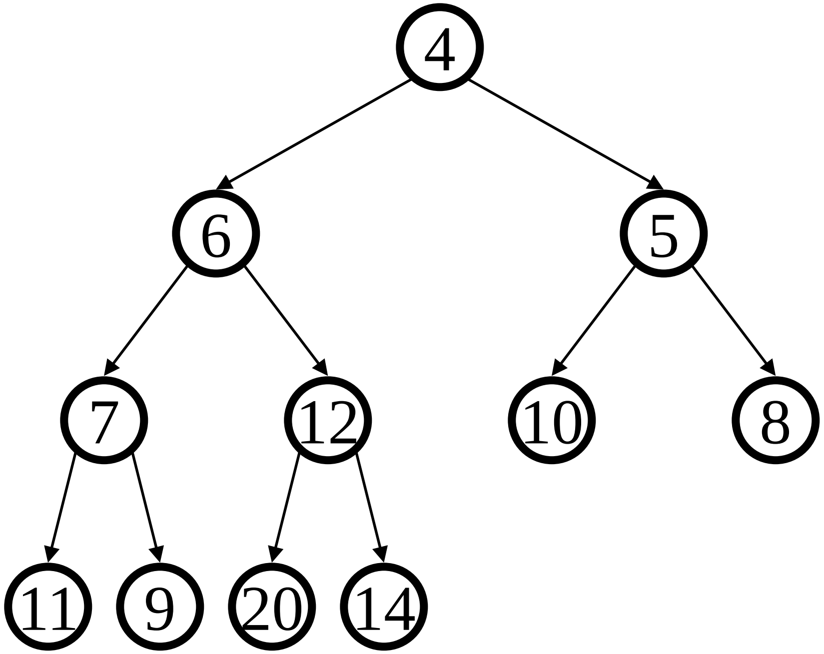


Invariants violated! DOOOM!!!

Percolate Down



Finally…



DeleteMin Code

```
Object deleteMin() {
  assert(!isEmpty());
  returnVal = Heap[0];
  size--;
  newPos =
    percolateDown(0,
        Heap[size]);
  Heap[newPos] =
    Heap[size];
  return returnVal;
}
```

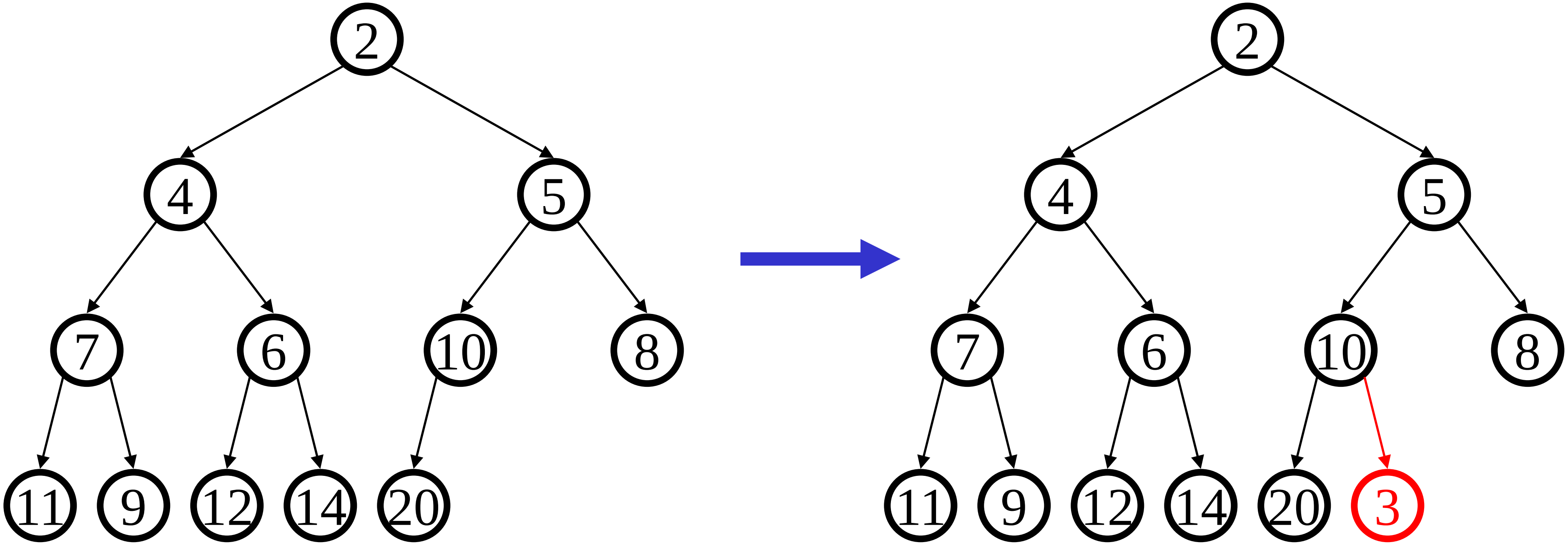
runtime:

```
int percolateDown(int hole,
                  Object val) {
while (2*hole+1 < size) {
    left = 2*hole + 1;
    right = left + 1;
    if (right < size &&
        Heap[right] < Heap[left])
      target = right;
    else
      target = left;

    if (Heap[target] < val) {
      Heap[hole] = Heap[target];
      hole = target;
    }
    else
      break;
  }
  return hole;
}
```

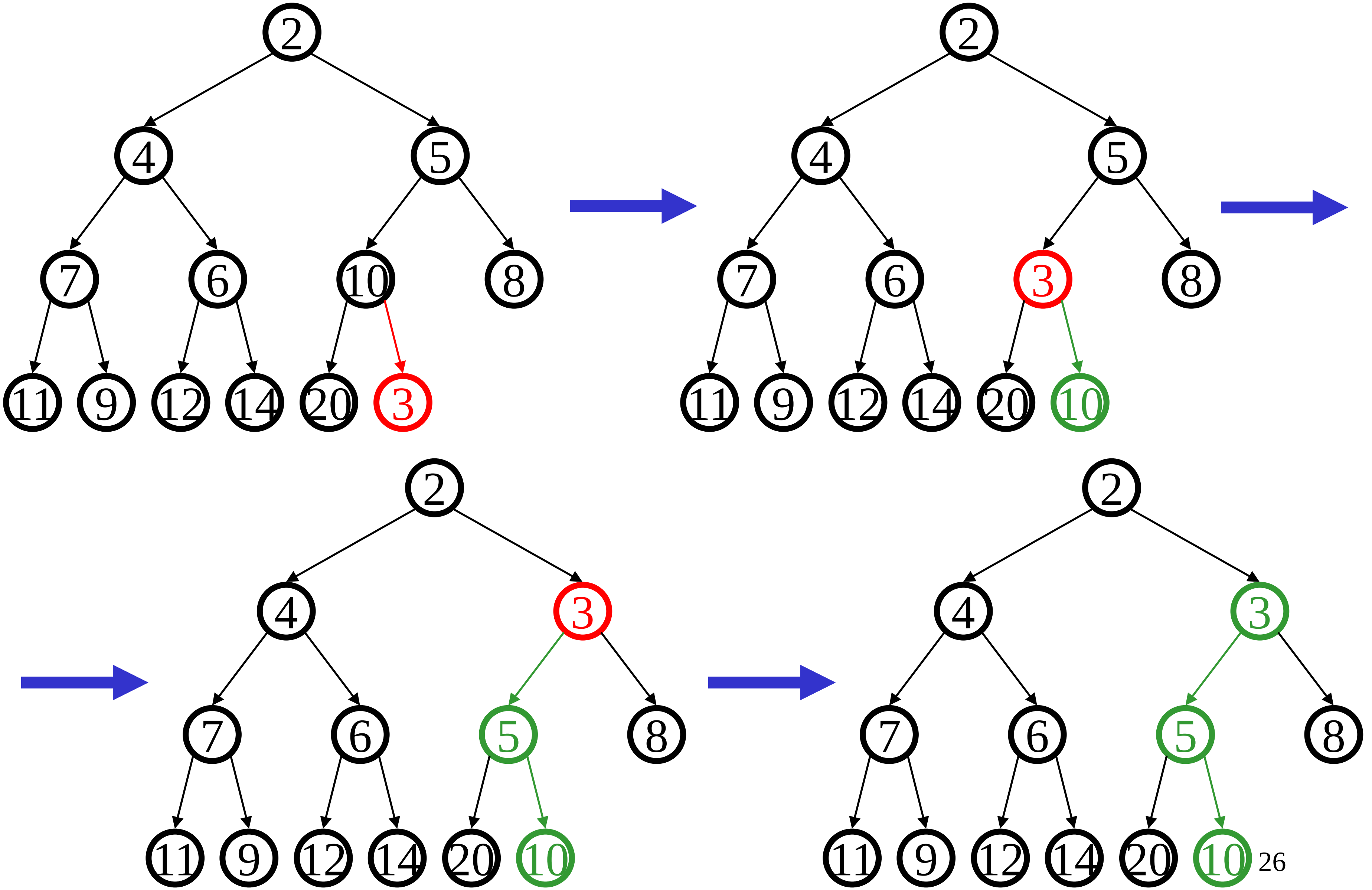
Insert

`pqueue.insert(3)`



Invariant violated! What will we do?

Percolate Up



Insert Code

```
void insert(Object o) {
  assert(!isFull());
  newPos =
    percolateUp(size,o);
  size++;
  Heap[newPos] = o;
}
```

```
int percolateUp(int hole,
                Object val) {
  while (hole > 0 &&
         val < Heap[(hole-1)/2])
    Heap[hole] = Heap[(hole-1)/2];
    hole = (hole-1)/2;
  }
  return hole;
}
```

runtime:

Today’s Outline

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

Closer Look at Creating Heaps

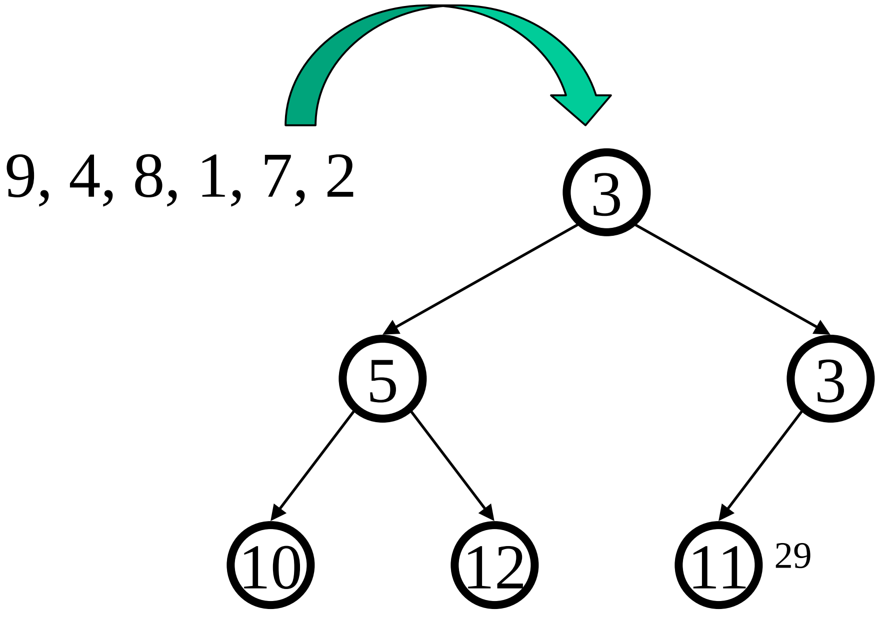
To create a heap given a list of items:

Create an empty heap.

For each item: insert into heap.

Time complexity?

a. O(lg n)
b. O(n)
c. O(n lg n)
d. O(n^2)
e. None of these

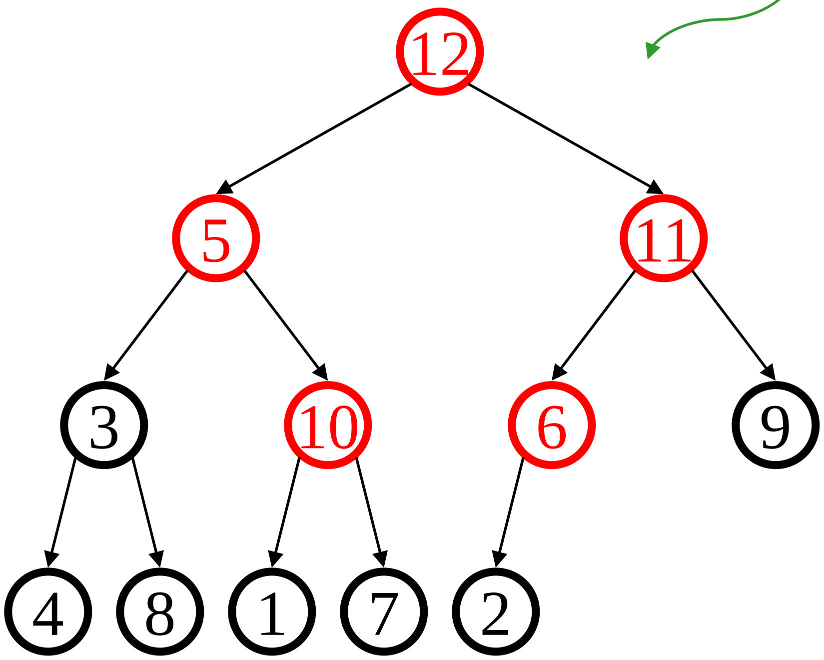


A Better BuildHeap

Floyd's Method. Thank you, Floyd.

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

pretend it's a heap and fix the heap-order property!



Invariant violated!

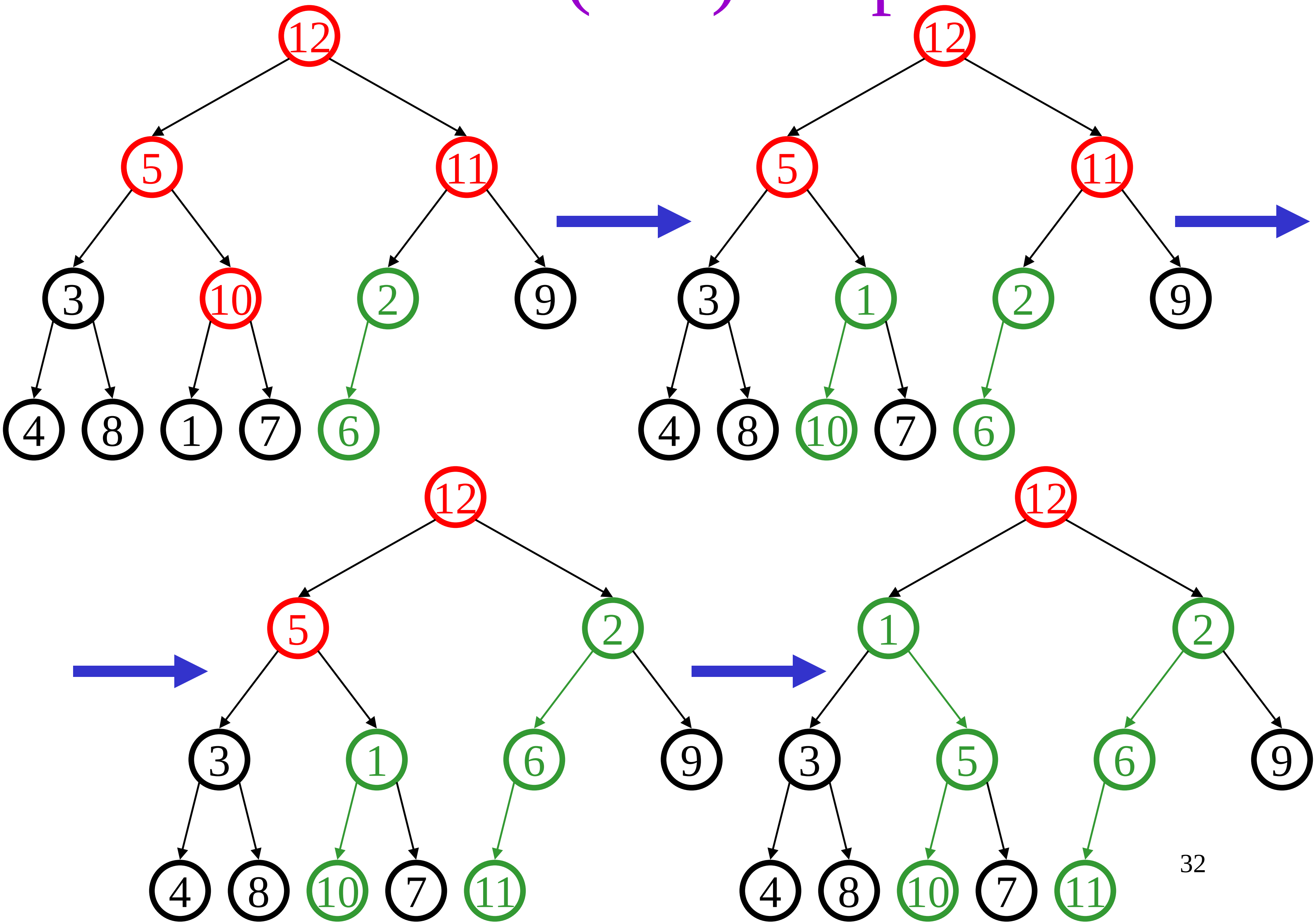
Where can the order invariant be violated in general?

a. Anywhere
b. Non-leaves
c. Non-roots

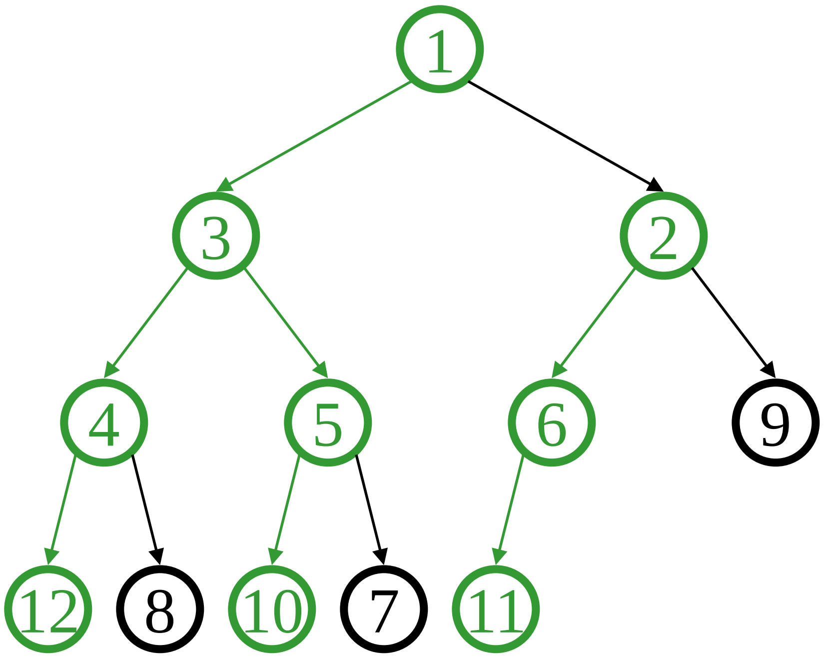
Alan’s Aside:

- I don’t really like the way Steve explains this.
- Heaps are recursive (mostly, except for structure):
  - A single node is a heap.
  - If parent value less than its child(ren), and child(ren) are heaps (except for “nearly complete” property).
- Think of enforcing the heap invariant from the bottom up!
  - Base Case: All nodes with no children are heaps already.
  - Inductive Case: My children are heaps. Percolate my value down, and that makes me a heap, too.

Build(this)Heap

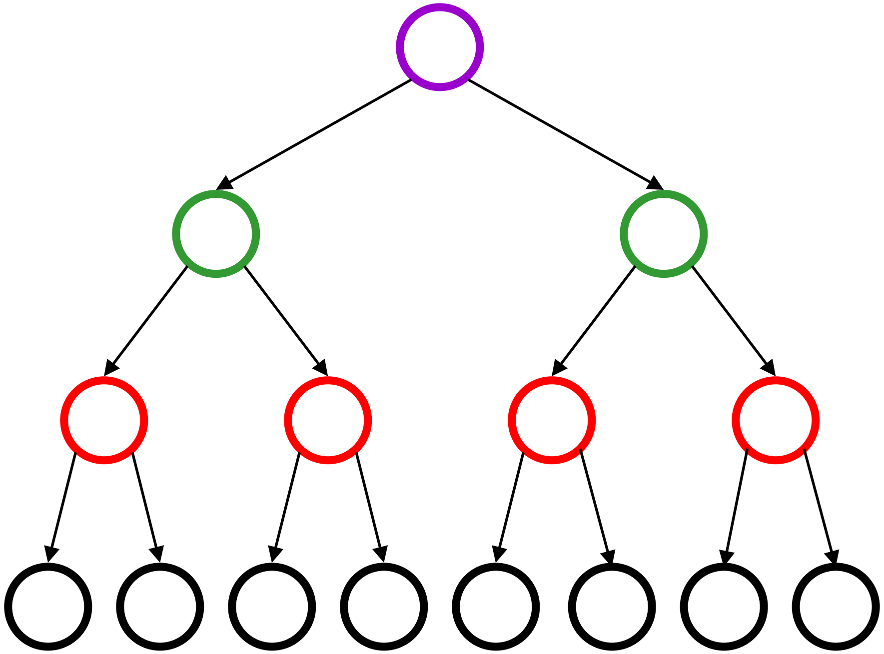


Finally…



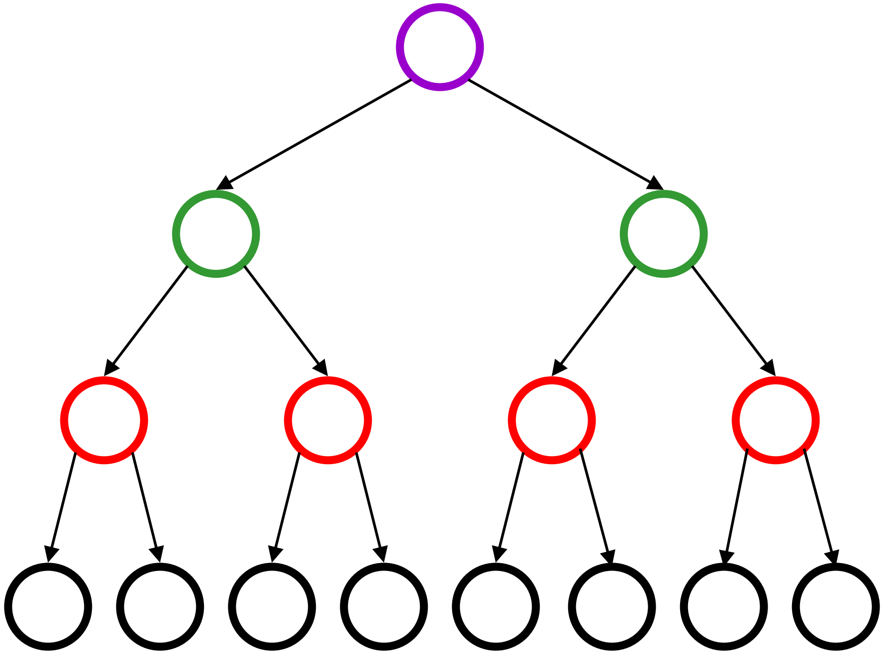
runtime:

Build(any)Heap



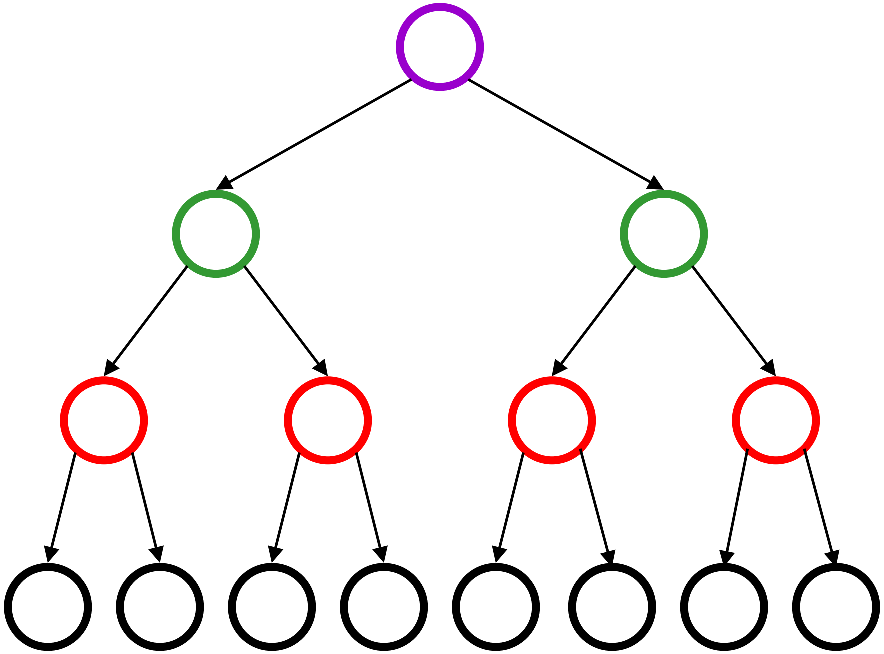
This is as many violations as we can get.
How do we fix them? Let's play colouring games!

Build(any)Heap



Alan's Aside: I like to think of this instead as "charging" edges in the tree for the cost of the moves. We can work out a scheme where each edge pays only once.
(A 1-1 correspondence!)

Build(any)Heap



Alan's Aside: The proof that this always works is inductive. The inductive step is that both of my subtrees have an uncharged path (rightmost) to the leaves. I charge my cost to my left child, and my right child provides the rightmost, uncharged path that I offer to my parent.

Alan's Aside

- Alternatively, we can do this with algebra.
- Consider a complete heap:
  - As we do percolate-down on bottom row, the cost is 0, each. There are roughly n/2 nodes on bottom row.
  - On next row up, the cost is 1, each. There are roughly n/4 nodes on second row.
  - On the kth row up, the cost is k-1 times n/(2^k) nodes on that row.
  - Therefore, run time is $\sum_{i=1}^{\log n}(i-1)\frac{n}{2^i} \leq \sum_{i=0}^{\infty} i\frac{n}{2^{i+1}} = \frac{n}{2}\sum_{i=0}^{\infty}\frac{i}{2^i} = n$

Alan’s Aside

- The last sum is tricky…
- Think of the 2s as 1+1; the 3s, as 1+1+1; etc.
- Now, add up a “layer” of 1s for the whole tree.
- Then, add up a layer of 1s for the part of the tree where the cost was 2 or more.
- Then, add up a layer of 1s for the part of the tree where the cost was 3 or more.
- Etc.

Alan's Aside

$$\sum_{i=0}^{\infty} \frac{i}{2^i} = \frac{0}{2^0} + \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \ldots$$

$$= \frac{1}{2^1} + \frac{1+1}{2^2} + \frac{1+1+1}{2^3} + \ldots$$

$$= \sum_{i=1}^{\infty} \frac{1}{2^i} + \sum_{i=2}^{\infty} \frac{1}{2^i} + \sum_{i=3}^{\infty} \frac{1}{2^i} + \ldots$$

Alan's Aside

$$\sum_{i=0}^{\infty}\frac{i}{2^i}=\sum_{i=1}^{\infty}\frac{1}{2^i}+\sum_{i=2}^{\infty}\frac{1}{2^i}+\sum_{i=3}^{\infty}\frac{1}{2^i}+\cdots$$

$$=\sum_{j=1}^{\infty}\left(\sum_{i=j}^{\infty}\frac{1}{2^i}\right)$$

$$=\sum_{j=1}^{\infty}\left(\frac{1}{2^{j-1}}\sum_{i=1}^{\infty}\frac{1}{2^i}\right)$$

$$=\sum_{j=1}^{\infty}\left(\frac{1}{2^{j-1}}\right)=2$$

Steve's Version of Alan's Aside

$$S=\sum_{i=0}^{\infty}\frac{i}{2^i}=\sum_{i=1}^{\infty}\frac{i}{2^i}=\frac{1}{2}+\sum_{i=2}^{\infty}\frac{i}{2^i}$$

$$=\frac{1}{2}+\frac{1}{2}\sum_{i=2}^{\infty}\frac{i}{2^{i-1}}=\frac{1}{2}+\frac{1}{2}\sum_{i=1}^{\infty}\frac{i+1}{2^i}$$

$$=\frac{1}{2}+\frac{1}{2}\left(\sum_{i=1}^{\infty}\frac{i}{2^i}+\sum_{i=1}^{\infty}\frac{1}{2^i}\right)$$

$$=\frac{1}{2}+\frac{1}{2}\left(\sum_{i=1}^{\infty}\frac{i}{2^i}+1\right)=\frac{1}{2}+\frac{1}{2}(S+1)$$

Today's Outline

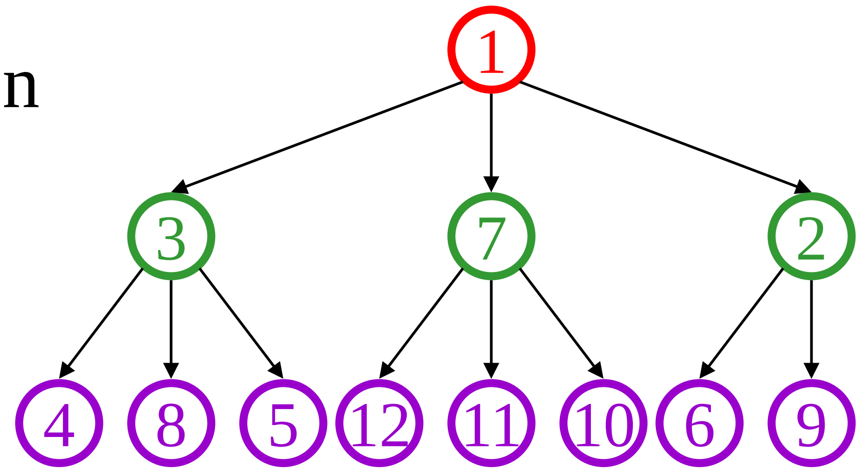
- Trees, Briefly
- Priority Queue ADT
- Heaps
  - Implementing Priority Queue ADT
  - Focus on Create: Heapify
  - Brief introduction to d-Heaps

Thinking about Binary Heaps

- Observations
  - finding a child/parent index is a multiply/divide by two
  - operations jump widely through the heap
  - deleteMins look at all (two) children of some nodes
  - inserts only care about parents of some nodes
  - inserts are at least as common as deleteMins
- Realities
  - division and multiplication by powers of two are **fast**
  - looking at one new piece of data sucks in a cache line
  - with **huge** data sets, disk accesses dominate

Solution: d-Heaps

- Nodes have (up to) *d* children
- Still representable by array
- Good choices for *d*:
  - optimize (non-asymptotic) performance based on ratio of inserts/removes
  - make *d* a power of two for efficiency
  - fit one set of children in a cache line
  - fit one set of children on a memory page/disk block



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

d-heap mnemonic:
d is for degree!

d-Heap calculations

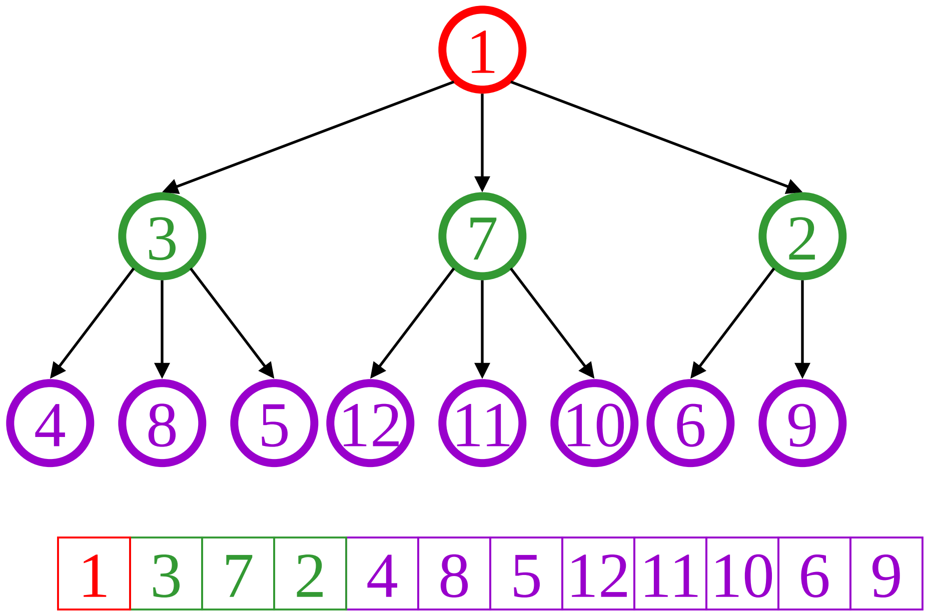
Calculations *in terms of d*:

– child:

– parent:

– root:

– next free:



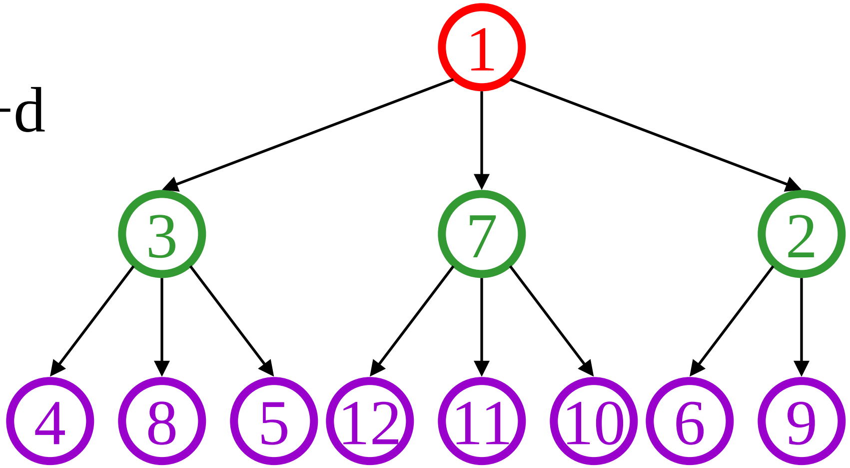
Alan’s Aside: Easier to work pattern if you count from zero!

d-heap mnemonic:
d is for degree!

d-Heap calculations

Calculations *in terms of d*:

- child: d*i+1 through d*i+d
- parent: floor((i-1)/d)
- root: 0
- next free: size



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

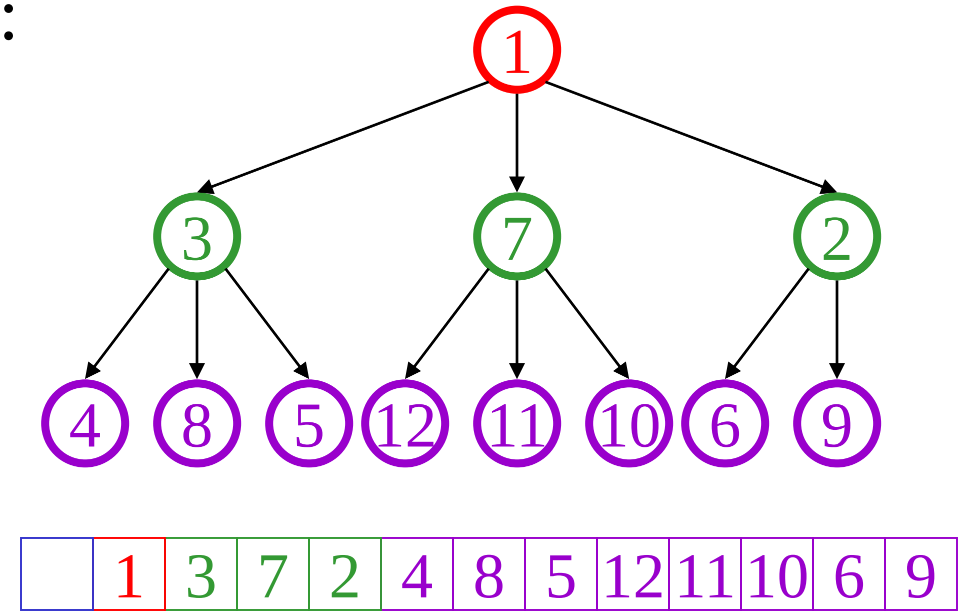
Alan's Aside: Easier to work pattern if you count from zero!

d-heap mnemonic:
d is for degree!

(Steve’s d-Heap calculations)

Calculations *in terms of d*:

- child:
- parent:
- root:
- next free:



d-heap mnemonic:
d is for degree!

(Steve's d-Heap calculations)

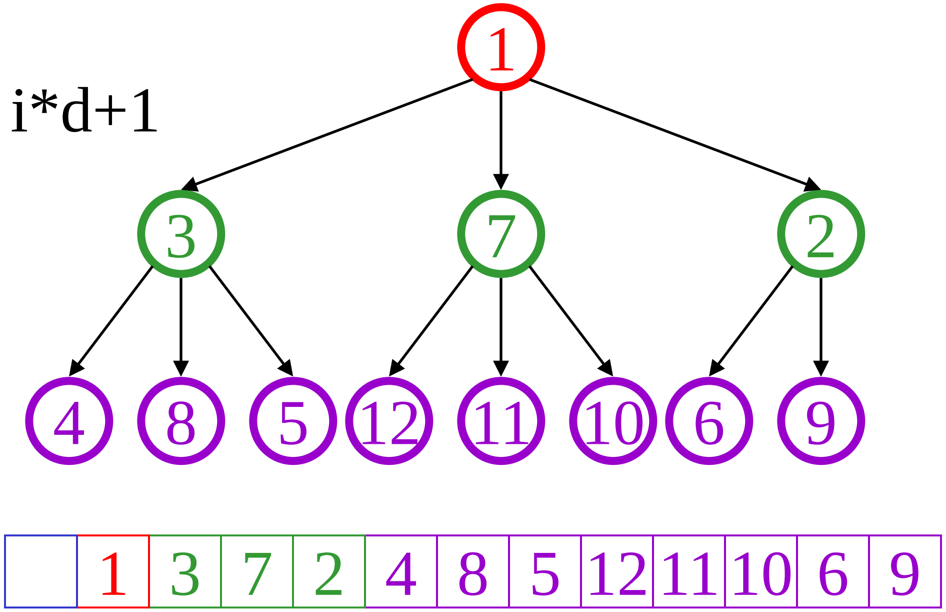
Calculations *in terms of d*:

– child: (i-1)*d+2 through i*d+1

– parent: floor((i-2)/d) + 1

– root: 1

– next free: size+1



d-heap mnemonic:
d is for degree!

Rest of Today’s Learning Goals

- Get comfortable with C++ pointers, understand the * and & operators.
- Draw diagrams to help understand code that manipulates pointers.

C++ Reference Parameters

- & in a formal parameter makes the parameter **another name for the argument that was passed in!**
  - (This is a totally different meaning of & from the "address of" operator (and also totally different from bitwise-AND).)
- **It's not a copy of the value of the argument,** the way normal parameter passing works.

C++ Reference Parameters

```
void swap(int x, int y) {
    int t = x;
    x = y;
    y = t;
}
…
int a=0; int b=1;
swap(a,b);
cout << a << ", " << b;
```

```
void swap(int &x, int &y) {
    int t = x;
    x = y;
    y = t;
}
…
int a=0; int b=1;
swap(a,b);
cout << a << ", " << b;
```

C++ Reference Parameters

```
void swap(int x, int y) {
    int t = x;
    x = y;
    y = t;
}
…
int a=0; int b=1;
swap(a,b);
cout << a << “, “ << b;
```

```
void swap(int &x, int &y) {
    int t = x;
    x = y;
    y = t;
}
…
int a=0; int b=1;
swap(a,b);
cout << a << “, “ << b;
```

Old-School C (and C++)

```
void swap(int *x, int *y) {
    int *t = x;
    x = y;
    y = t;
}
…
int a=0; int b=1;
swap(a,b);
cout << a << ", " << b;
```

```
void swap(int *x, int *y) {
    int t = *x;
    *x = *y;
    *y = t;
}
…
int a=0; int b=1;
swap(a,b);
cout << a << ", " << b;
```

Old-School C (and C++)

```
void swap(int *x, int *y) {
    int *t = x;
    x = y;
    y = t;
}
…
int a=0; int b=1;
swap(a,b);
cout << a << ", " << b;
```

```
void swap(int *x, int *y) {
    int t = *x;
    *x = *y;
    *y = t;
}
…
int a=0; int b=1;
swap(a,b);
cout << a << ", " << b;
```

Old-School C (and C++)

```
void swap(int *x, int *y) {
    int t = *x;
    *x = *y;
    *y = t;
}
…
int a=0; int b=1;
swap(&a,&b);
cout << a << “, “ << b;
```

```
void swap(int *x, int *y) {
    int t = *x;
    *x = *y;
    *y = t;
}
…
int a=0; int b=1;
swap(a,b);
cout << a << “, “ << b;
```

Old-School C (and C++)

```
void swap(int *x, int *y) {
    int t = *x;
    *x = *y;
    *y = t;
}
…
int a=0; int b=1;
swap(&a,&b);
cout << a << ", " << b;
```

```
void swap(int *x, int *y) {
    int t = *x;
    *x = *y;
    *y = t;
}
…
int a=0; int b=1;
swap(a,b);
cout << a << ", " << b;
```