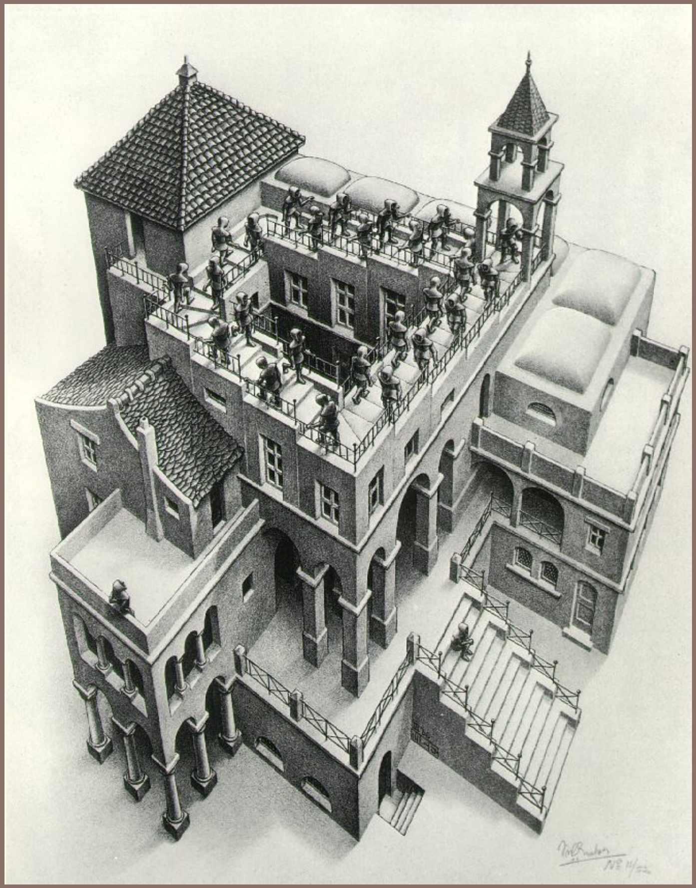


PRIORITY QUEUES AND HEAPS

Lecture 16

CS2110 Spring 2015

Readings and Homework

Read Chapter 26 "A Heap Implementation" to learn about heaps

Exercise: Salespeople often make matrices that show all the great features of their product that the competitor's product lacks. Try this for a heap versus a BST. First, try and sell someone on a BST: List some desirable properties of a BST that a heap lacks. Now be the heap salesperson: List some good things about heaps that a BST lacks. Can you think of situations where you would favor one over the other?

With ZipUltra heaps, you've got it made in the shade my friend!

Cool data structures you now know about

- Linked lists –singly linked, doubly linked, circular
- Binary search trees
- BST-like tree for A4 (BlockTree)
- Example of how one changes a data structure to make for efficiency purposes:

  In A4 a Shape (consisting of 1,000 Blocks?) gets moved around, rather than change the position field in each Block, have a field of Shape that gives the displacement for all Blocks.

Interface Bag (not In Java Collections)


```
interface Bag<E>
       implements Iterable {
   void add(E obj);
   boolean contains(E obj);
   boolean remove(E obj);
   int size();
   boolean isEmpty();
   Iterator<E> iterator()
}
```

Also called multiset

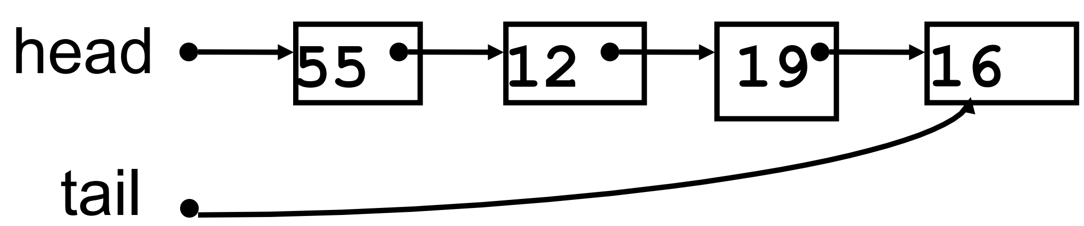
Like a set except that a value can be in it more than once. Example: a bag of coins

Refinements of Bag: Stack, Queue, PriorityQueue

Stacks and queues are restricted lists

- Stack (LIFO) implemented as list
  - `add()`, `remove()` from front of list
- Queue (FIFO) implemented as list
  - `add()` on back of list, `remove()` from front of list
- These operations are O(1)

Both efficiently implementable using a singly linked list with head and tail



Priority queue

- **`Bag`** in which data items are **`Comparable`**
- Smaller elements (determined by **`compareTo()`**) have higher priority
- **`remove()`** return the element with the highest priority = least in the **`compareTo()`** ordering
- break ties arbitrarily

Examples of Priority Queues

Scheduling jobs to run on a computer
default priority = arrival time
priority can be changed by operator

Scheduling events to be processed by an event handler
priority = time of occurrence

Airline check-in
first class, business class, coach
FIFO within each class

Tasks that you have to carry out. You determine priority

java.util.PriorityQueue<E>


```
boolean add(E e) {...} //insert an element
void clear() {...} //remove all elements
E peek() {...} //return min element without removing
E poll() {...} //remove and return min element
boolean contains(E e)
boolean remove(E e)
int size() {...}
Iterator<E> iterator()   //an iterator over the priority queue
```

Priority queues as lists

- Maintain as unordered list
  - `add()` put new element at front – O(1)
  - **poll()** must search the list – O(n)
  - **peek()** must search the list – O(n)

- Maintain as ordered list
  - `add()` must search the list – O(n)
  - **poll()** must search the list – O(n)
  - **peek()** O(1)

Can we do better?

Important Special Case

- Fixed number of priority levels 0,...,p – 1
- FIFO within each level
- Example: airline check-in

- **`add()`** – insert in appropriate queue – O(1)
- **`poll()`** – must find a nonempty queue – O(p)

Heap

- A *heap* is a concrete data structure that can be used to implement priority queues
- Gives better complexity than either ordered or unordered list implementation:
  - **`add()`**: O(log n)
  - **`poll()`**: O(log n)
- O(n log n) to process n elements
- Do not confuse with *heap memory*, where the Java virtual machine allocates space for objects – different usage of the word *heap*

Heap

- Binary tree with data at each node
- Satisfies the *Heap Order Invariant*:

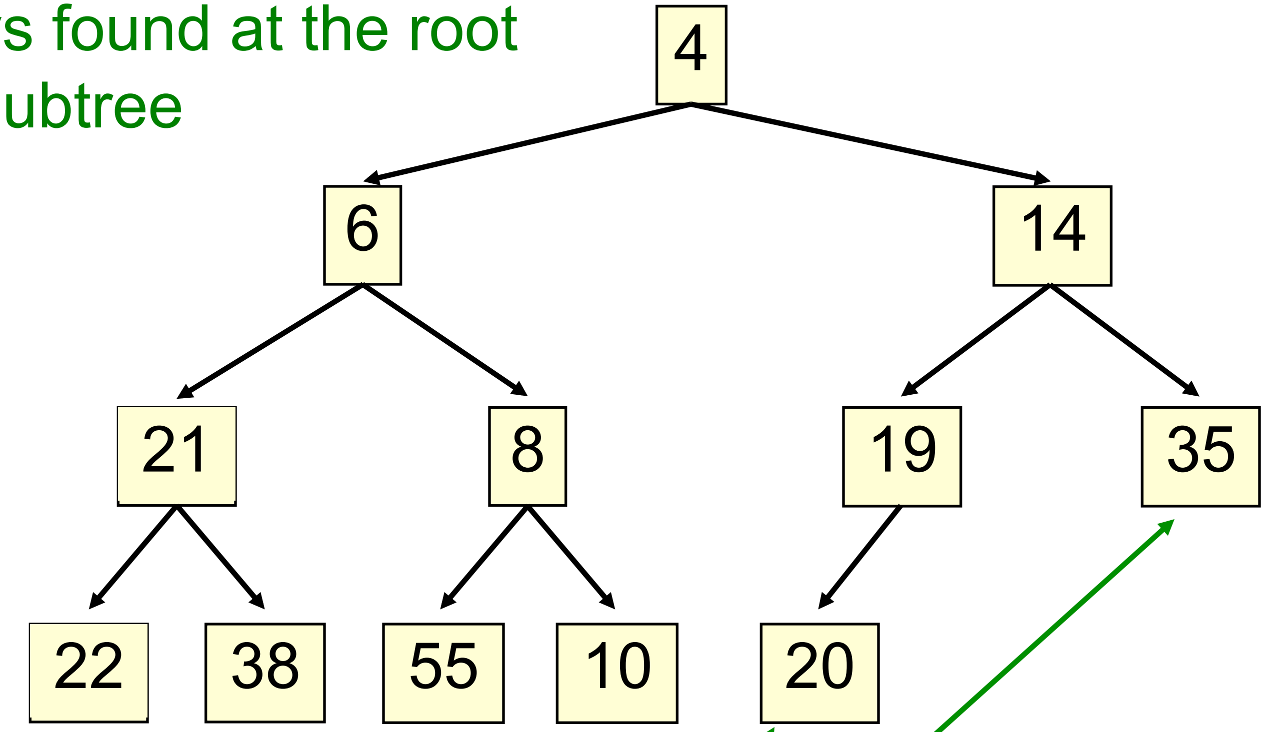
> 1. The least (highest priority) element of any subtree is at the root of that subtree.

- Binary tree is complete (no holes)

> 2. Every level (except last) completely filled. Nodes on bottom level are as far left as possible.

Heap

Smallest element in any subtree is always found at the root of that subtree



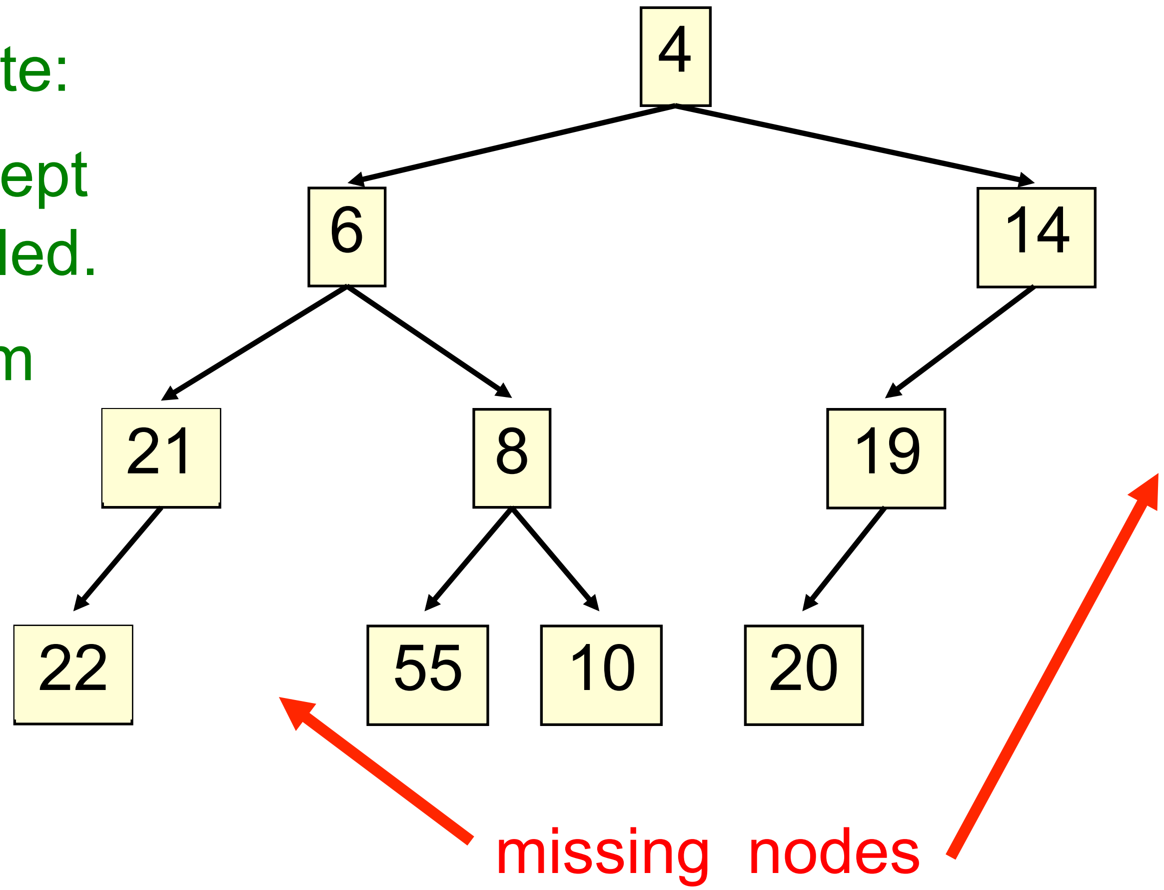
Note: 19, 20 < 35: Smaller elements can be deeper in the tree!

Not a heap —has two holes

Should be complete:

* Every level (except last) completely filled.

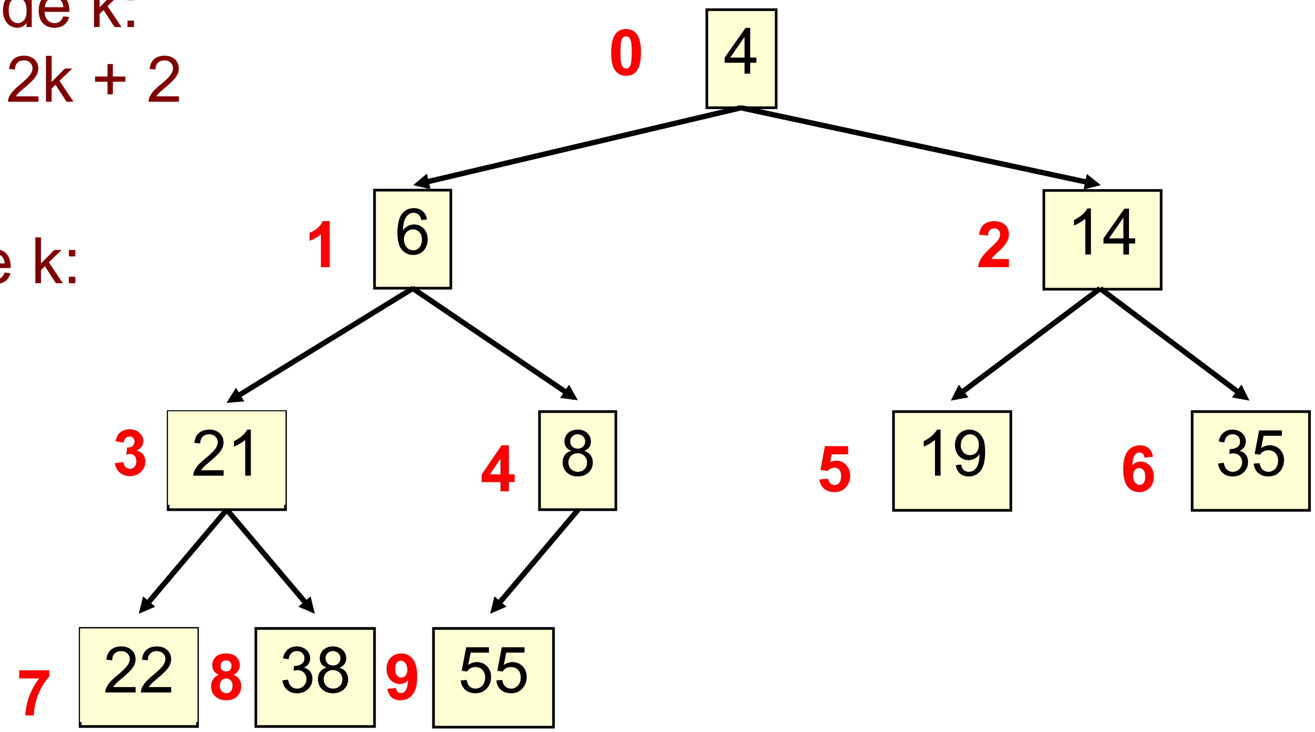
* Nodes on bottom level are as far left as possible.



Heap: number nodes as shown

children of node k:
at 2k + 1 and 2k + 2

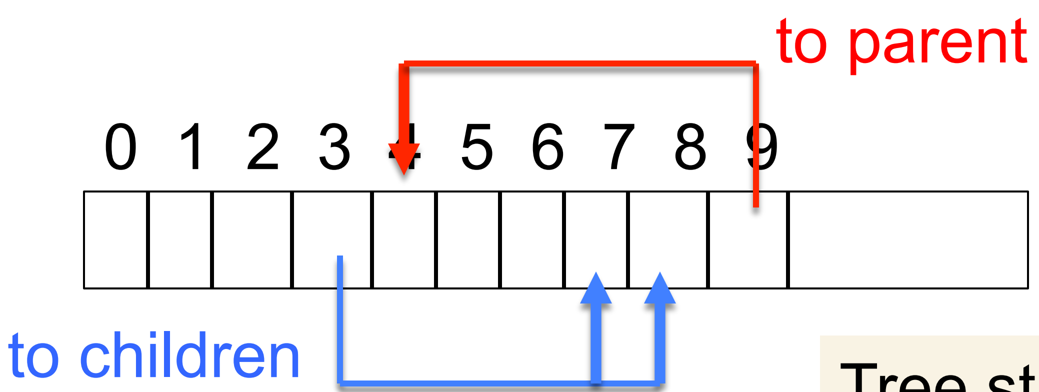
parent of node k:
at (k-1) / 2



Remember, tree has no holes

We illustrate using an array b (could also be ArrayList or Vector)

- Heap nodes in b in order, going across each level from left to right, top to bottom
- Children b[k] are b[2k + 1] and b[2k + 2]
- Parent of b[k] b[(k – 1)/2]

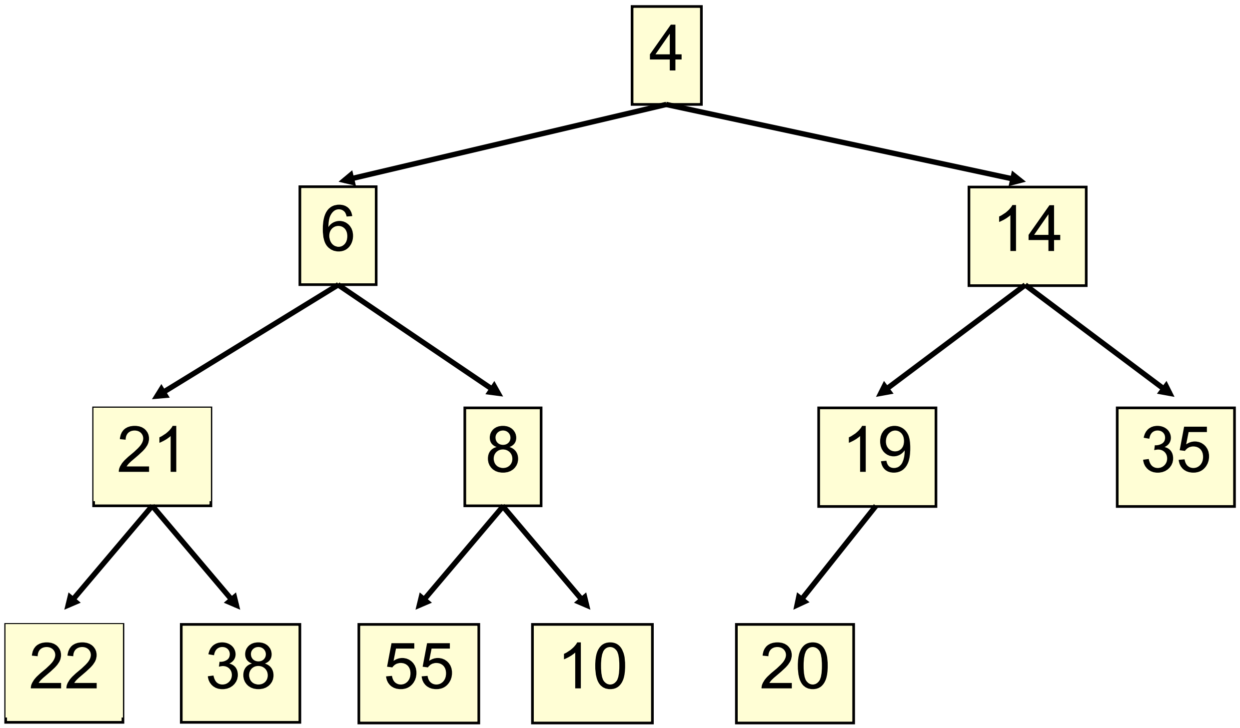


Tree structure is implicit.
No need for explicit links!

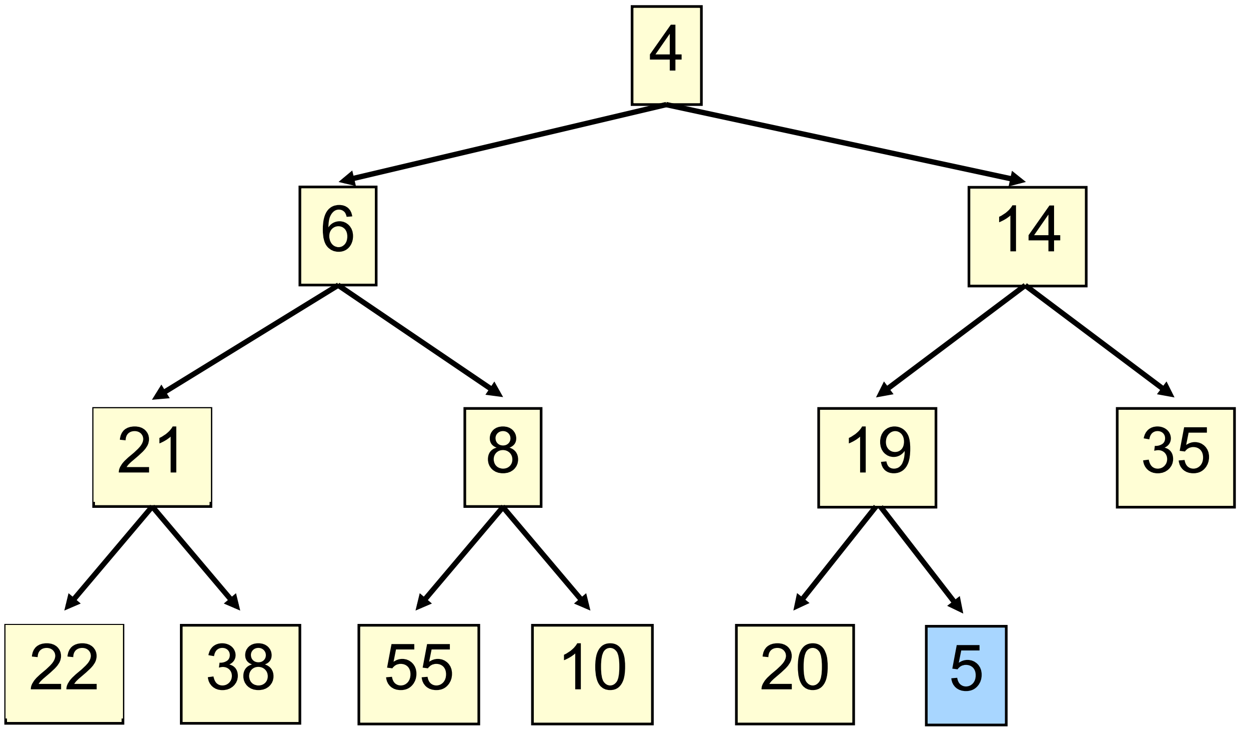
`add(e)`

- Add e at the end of the array
- If this violates heap order because it is smaller than its parent, swap it with its parent
- Continue swapping it up until it finds its rightful place
- The heap invariant is maintained!

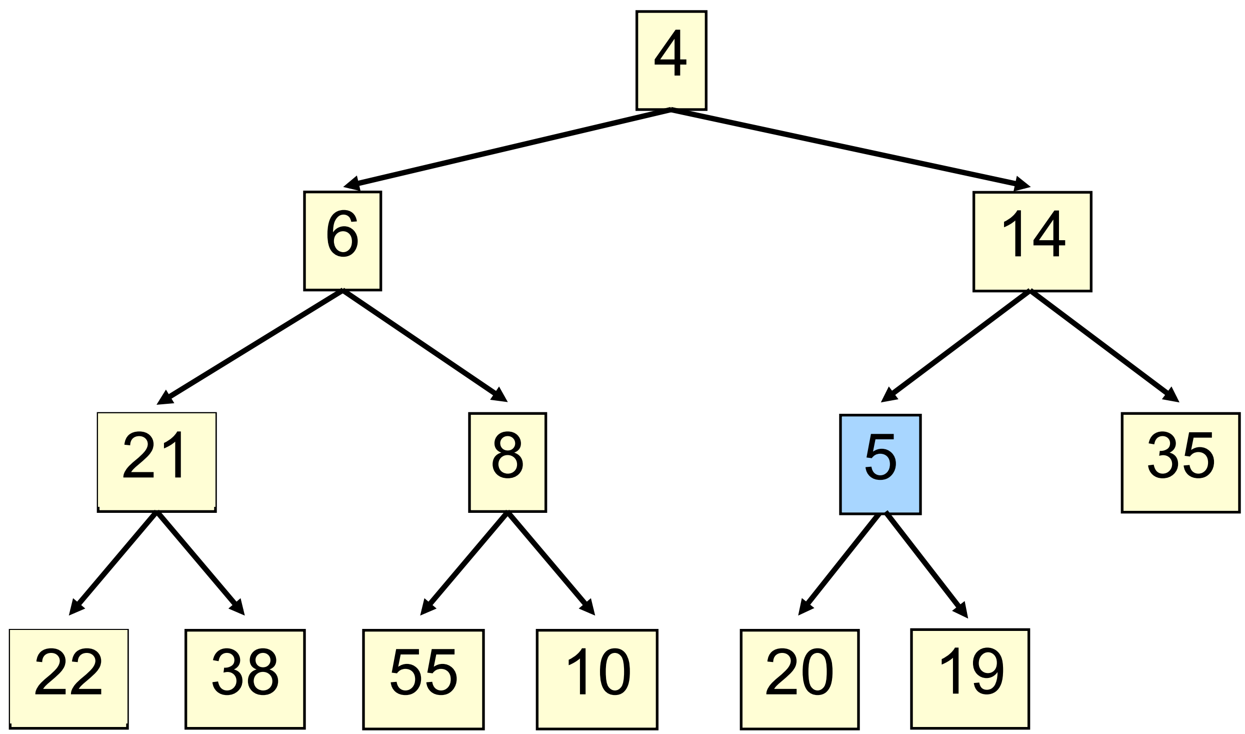
`add()`



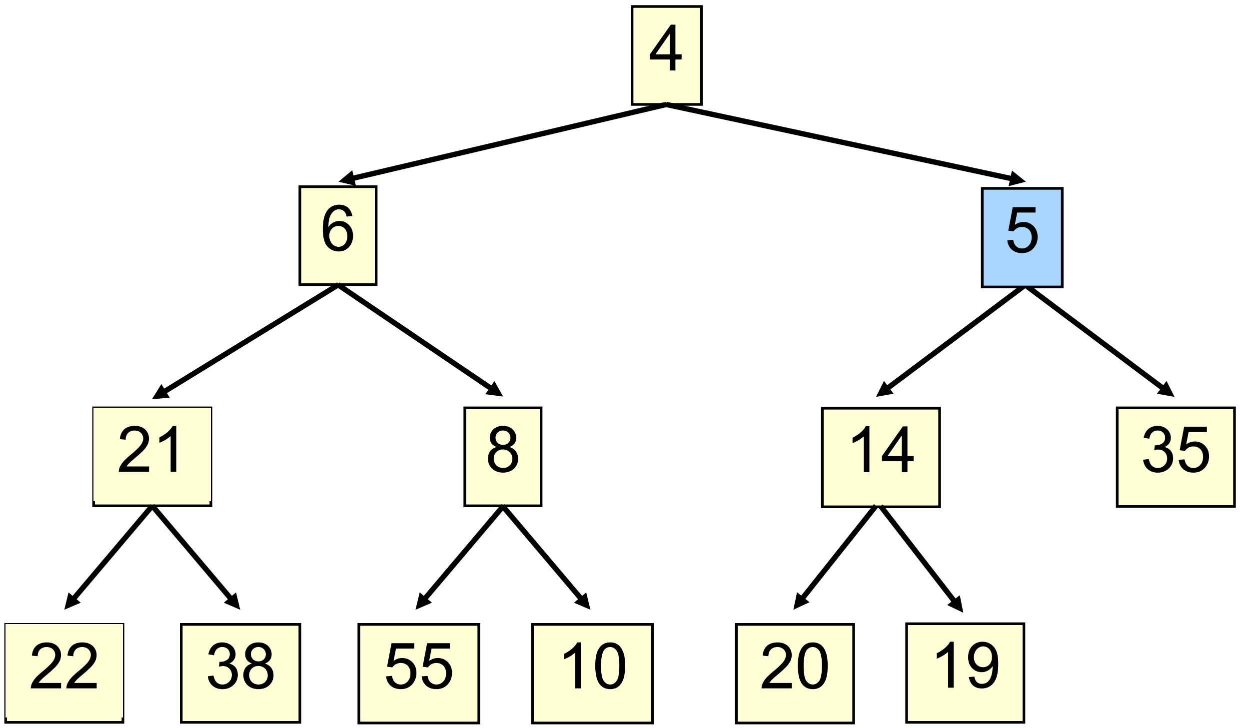
`add()`



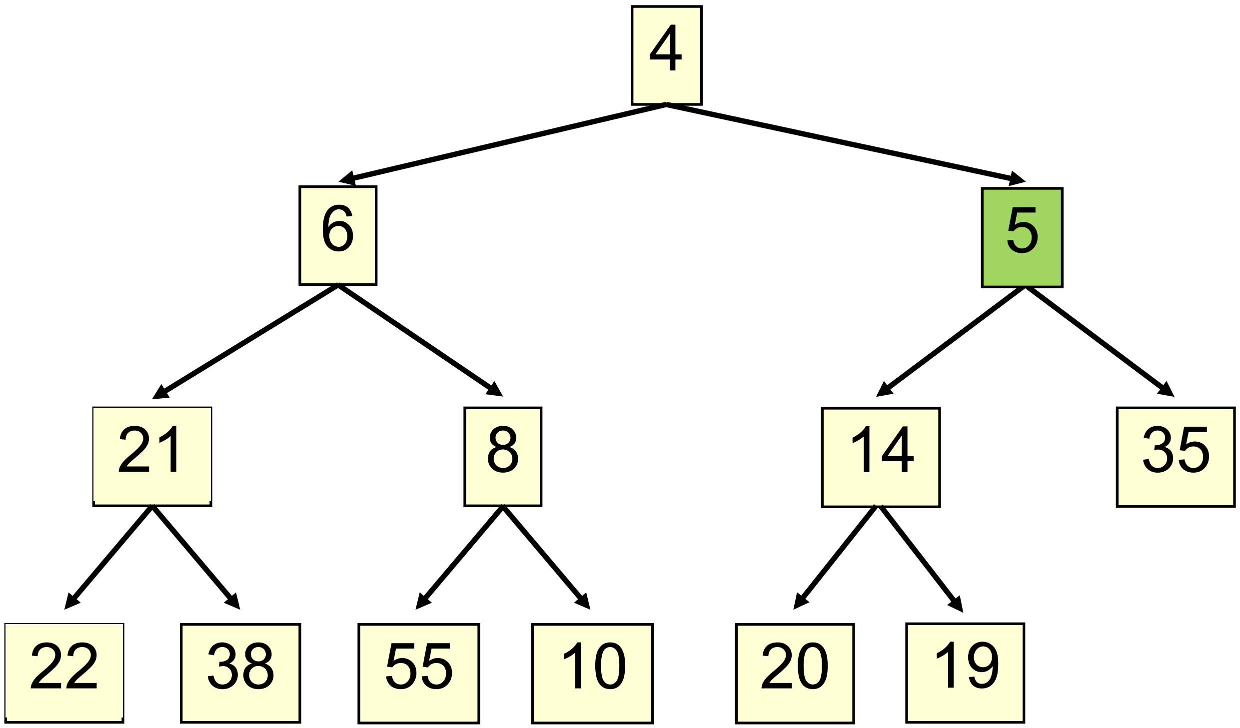
add()



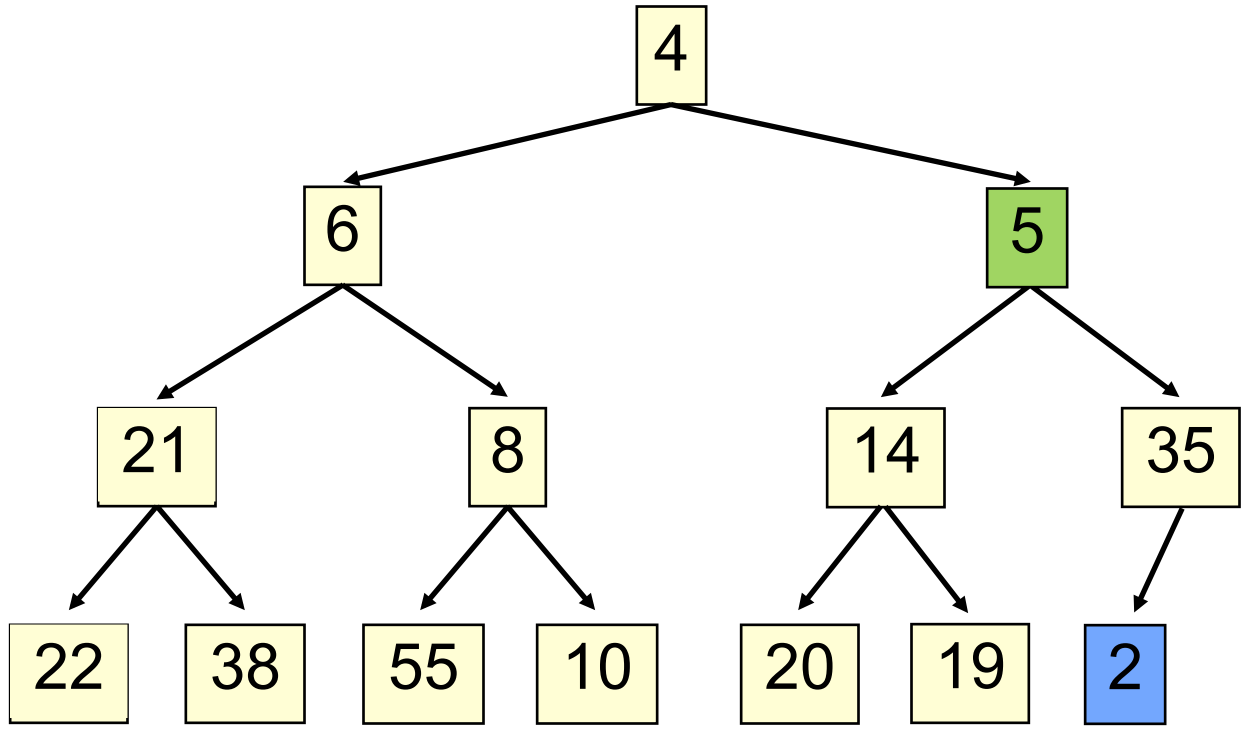
add()



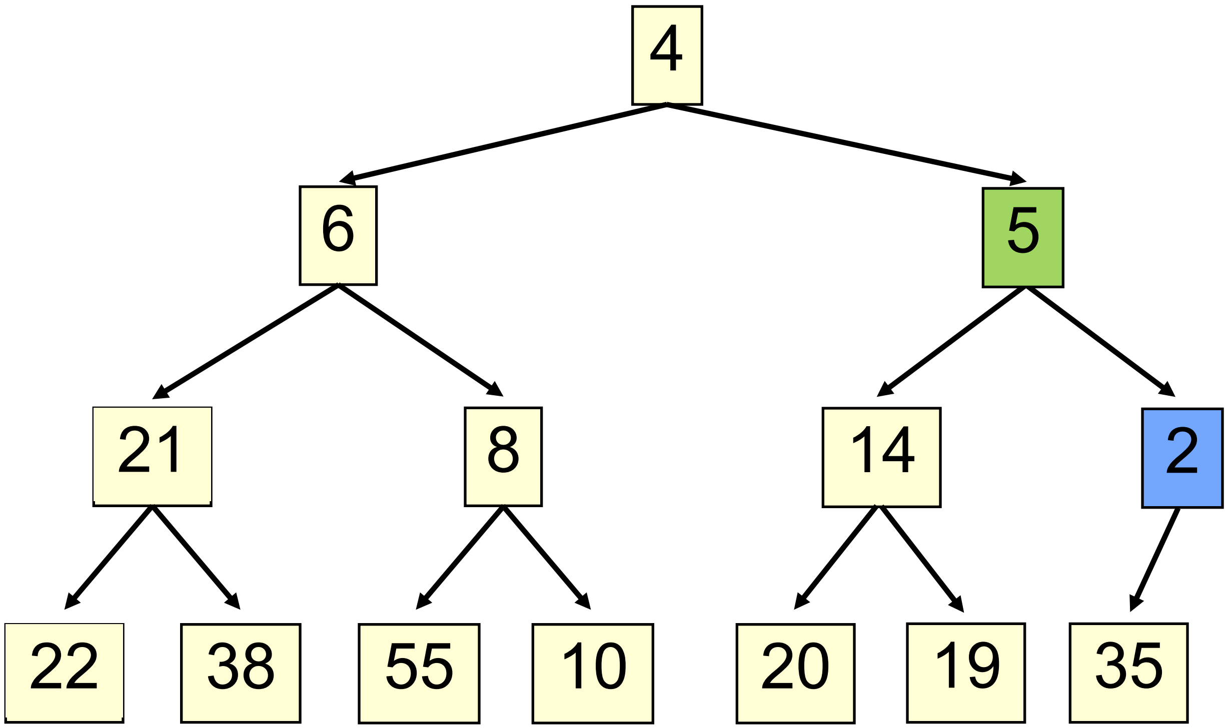
add()



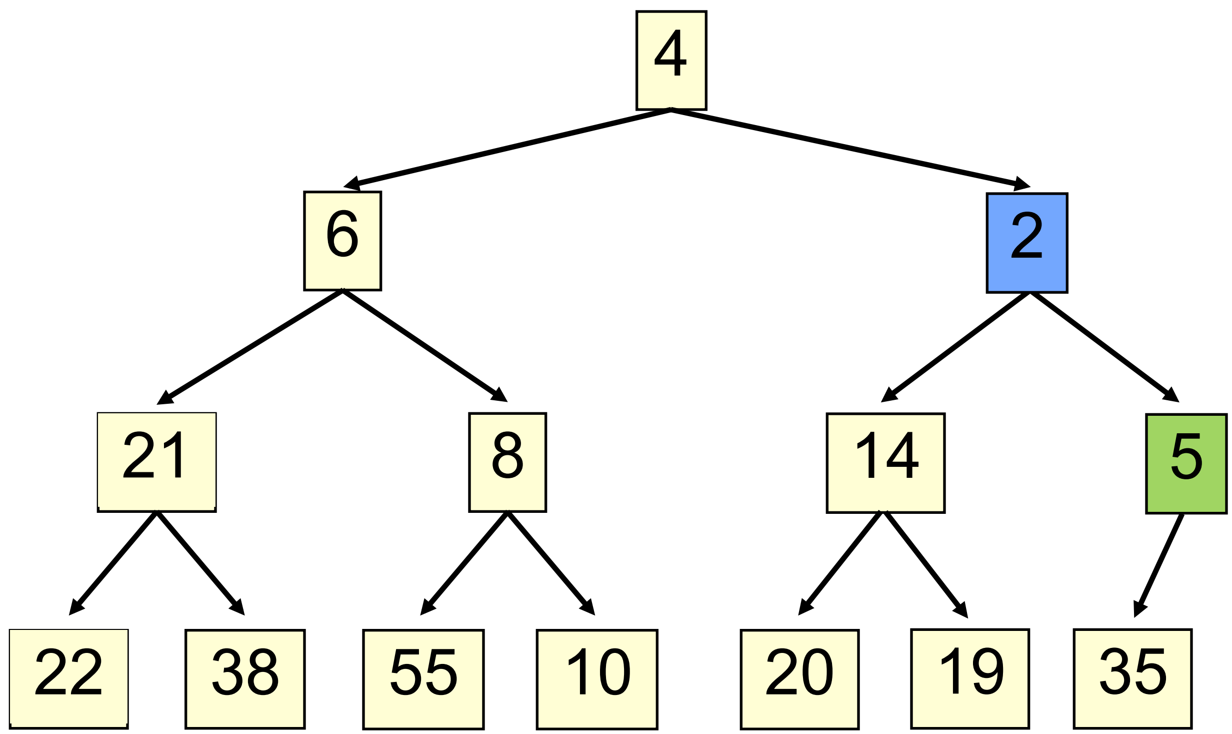
add()



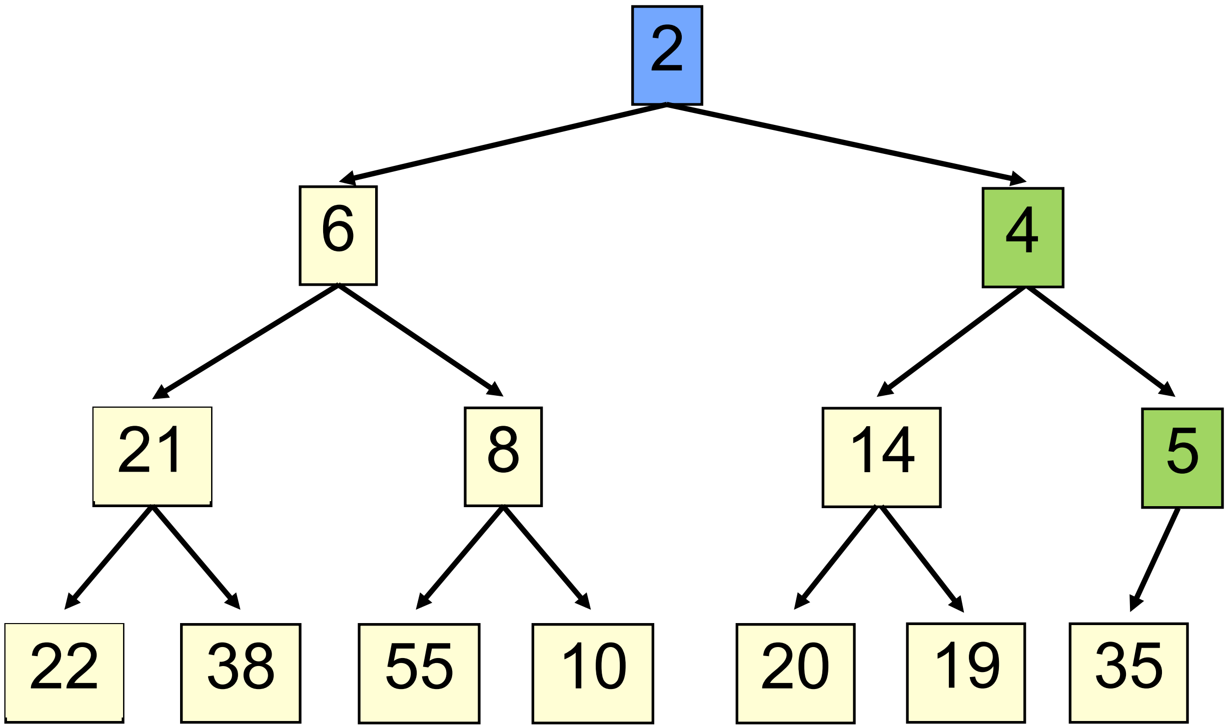
add()



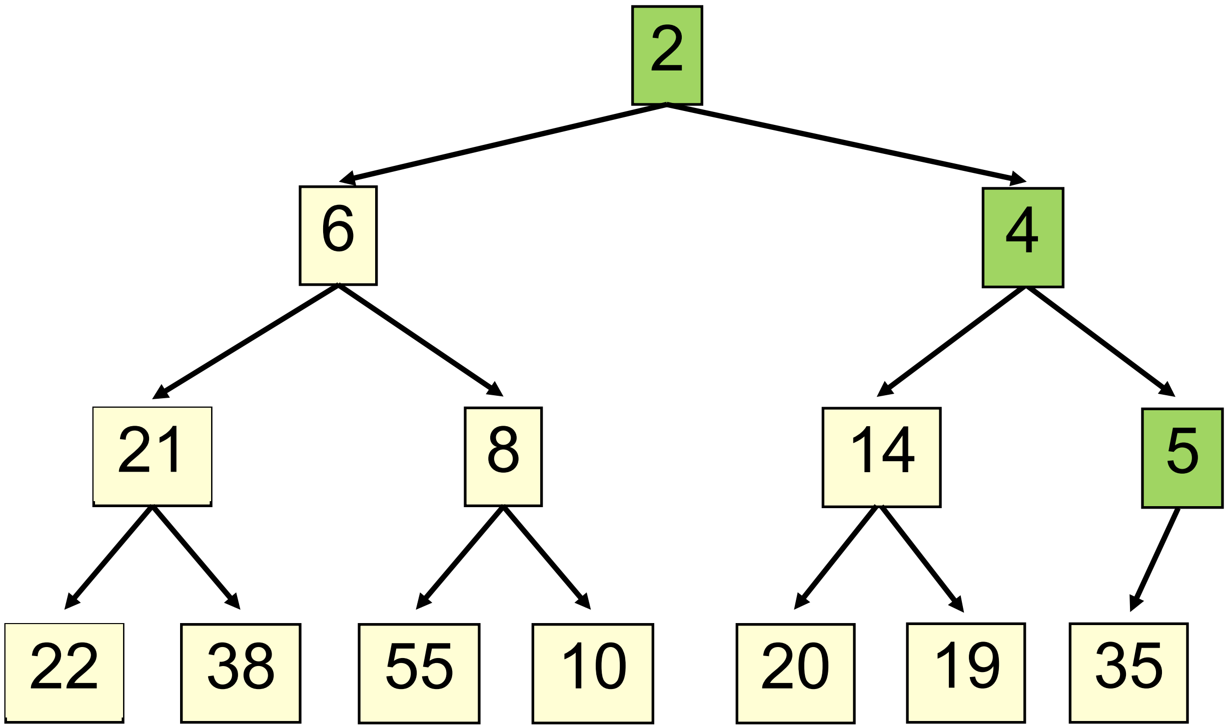
`add()`



add()



add()



`add()` to a tree of size n

- Time is O(log n), since the tree is balanced
  - size of tree is exponential as a function of depth
  - depth of tree is logarithmic as a function of size

add() --assuming there is space


```
/** An instance of a heap */
Class Heap<E>{
  E[] b= new E[50];  //heap is b[0..n-1]
  int n= 0;          // heap invariant is true

  /** Add e to the heap */
  public void add(E e) {
    b[n]= e;
    n= b + 1;
    bubbleUp(n - 1); // given on next slide
  }
}
```

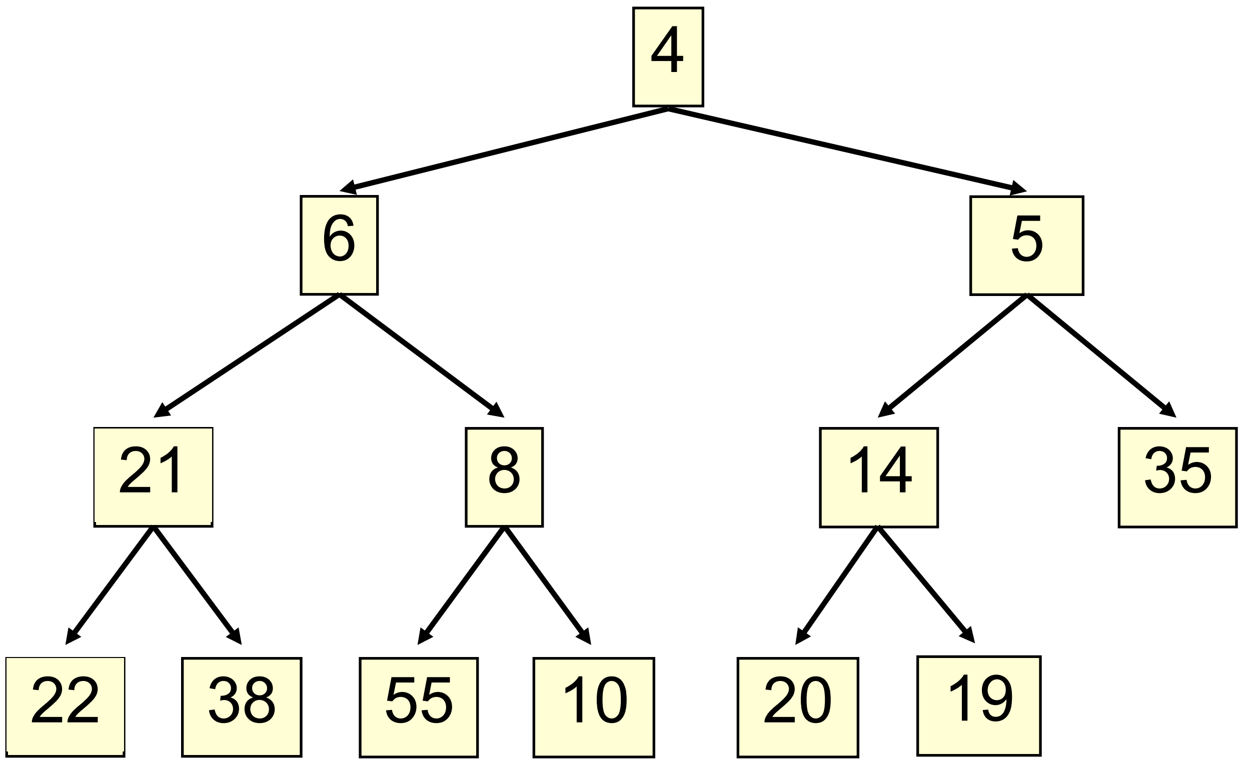
`add()`. Remember, heap is in b[0..n-1]


```
class Heap<E> {
   /** Bubble element #k up to its position.
     *  Precondition: heap inv true except maybe for element k */
   private void bubbleUp(int k) {
      int p= (k-1)/2;   // p is the parent of k
      // inv: p is k's parent  and
      // Every element satisfies the heap property
      // except perhaps k (might be smaller than its parent)
     while (k>0  &&  b[k].compareTo(b[p]) < 0) {
          Swap b[k] and b[p];
          k= p;
          p= (k-1)/2;
   }
}
```

`poll()`

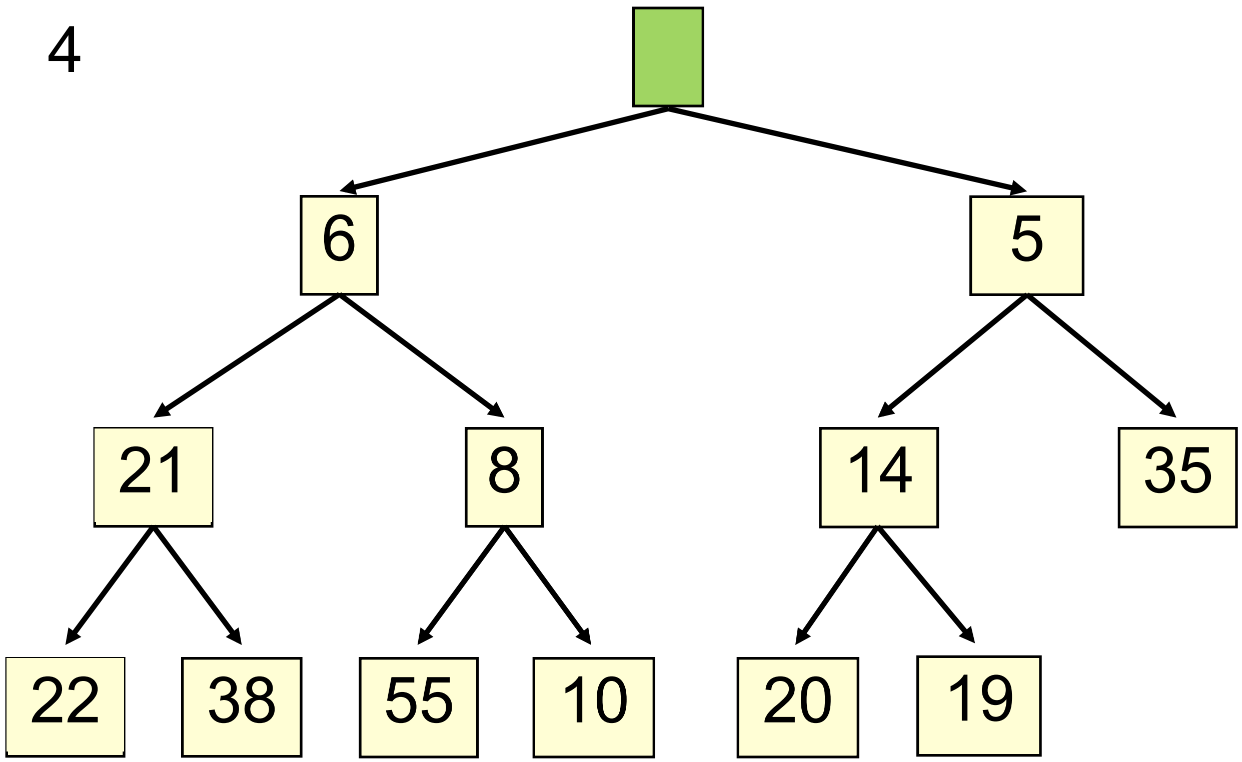
- Remove the least element and return it – (at the root)
- This leaves a hole at the root – fill it in with the last element of the array
- If this violates heap order because the root element is too big, swap it down with the smaller of its children
- Continue swapping it down until it finds its rightful place
- The heap invariant is maintained!

poll()



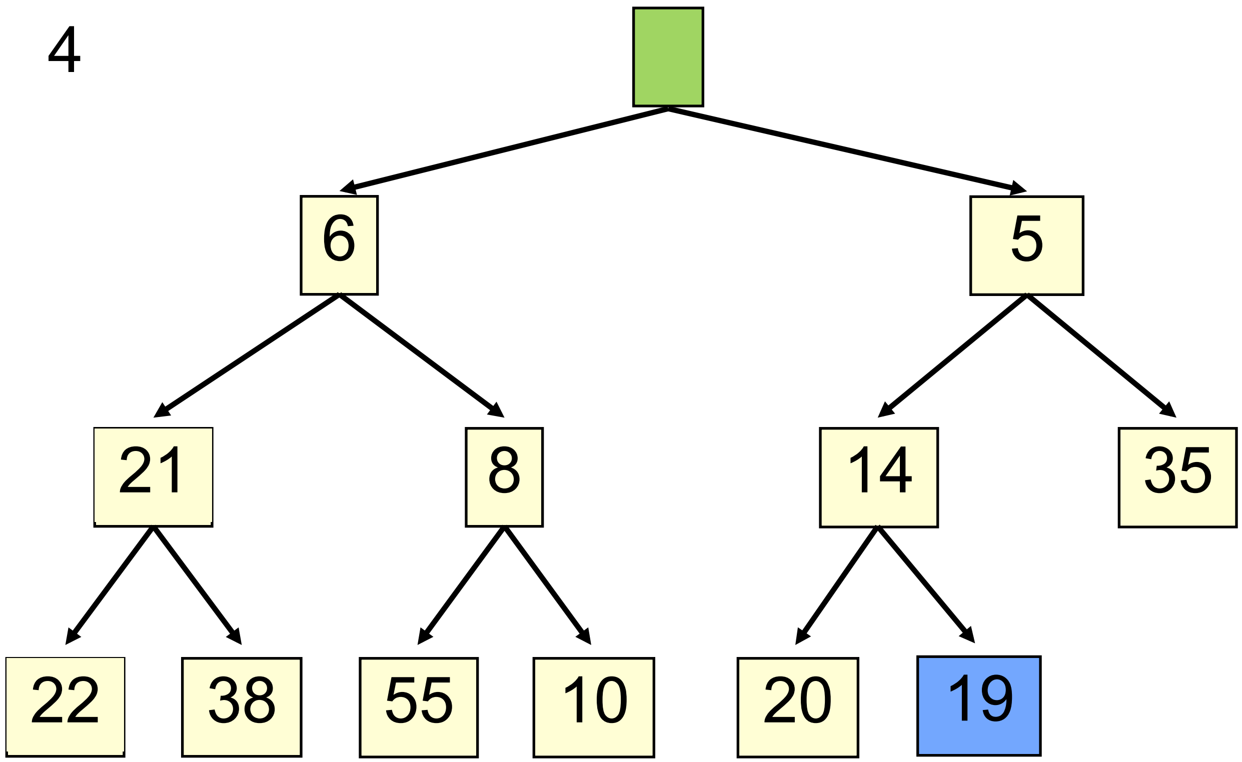
`poll()`

4



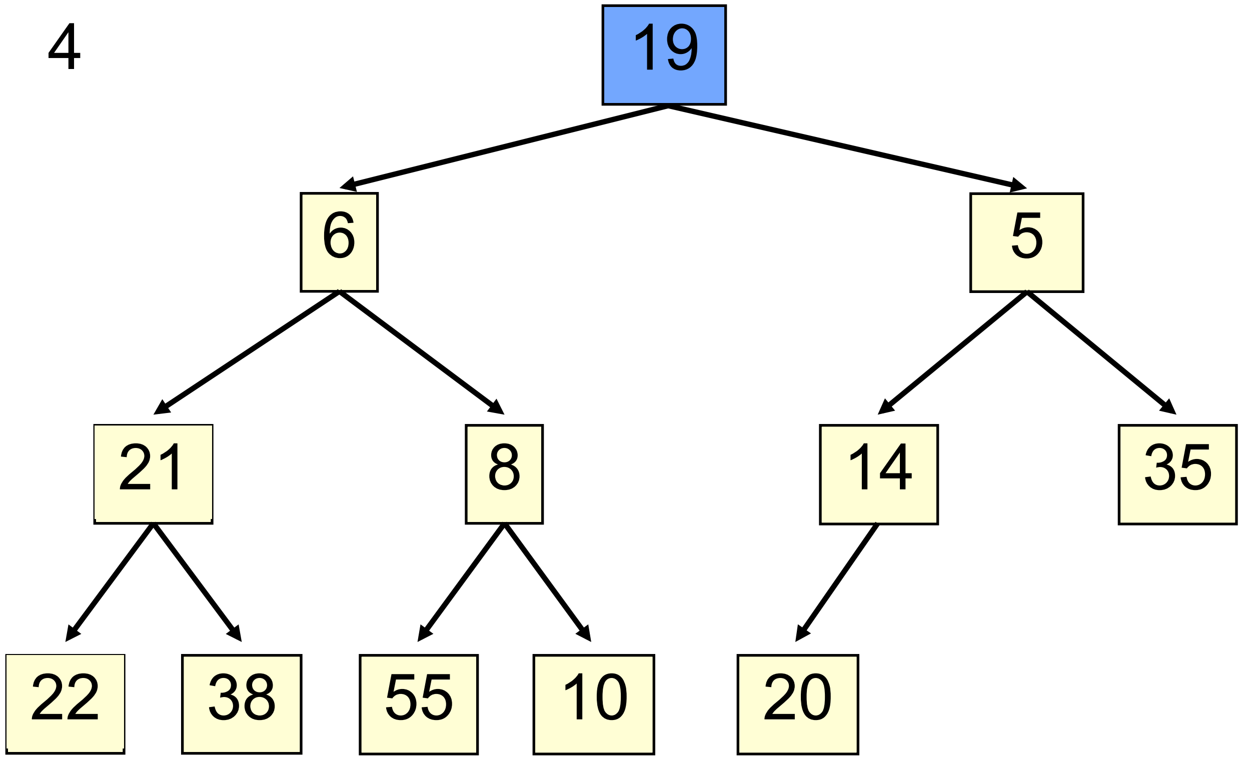
poll()

4



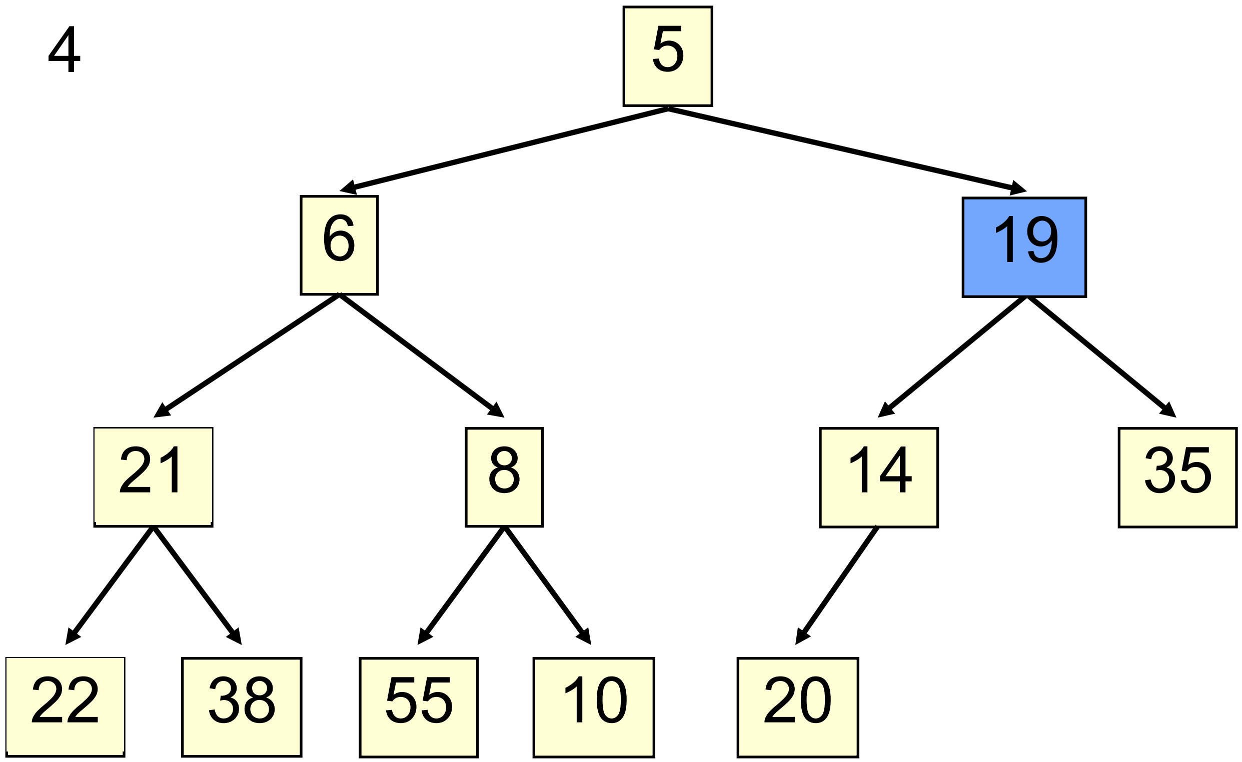
poll()

4



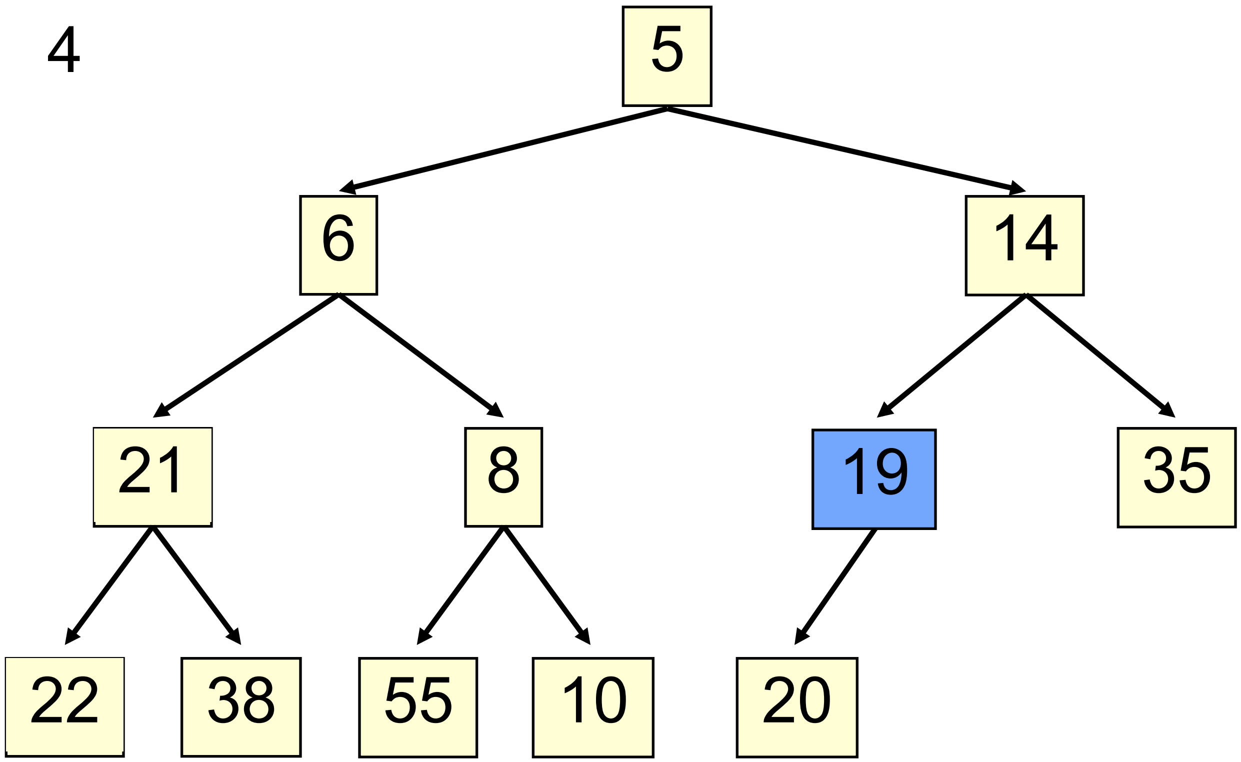
poll()

4



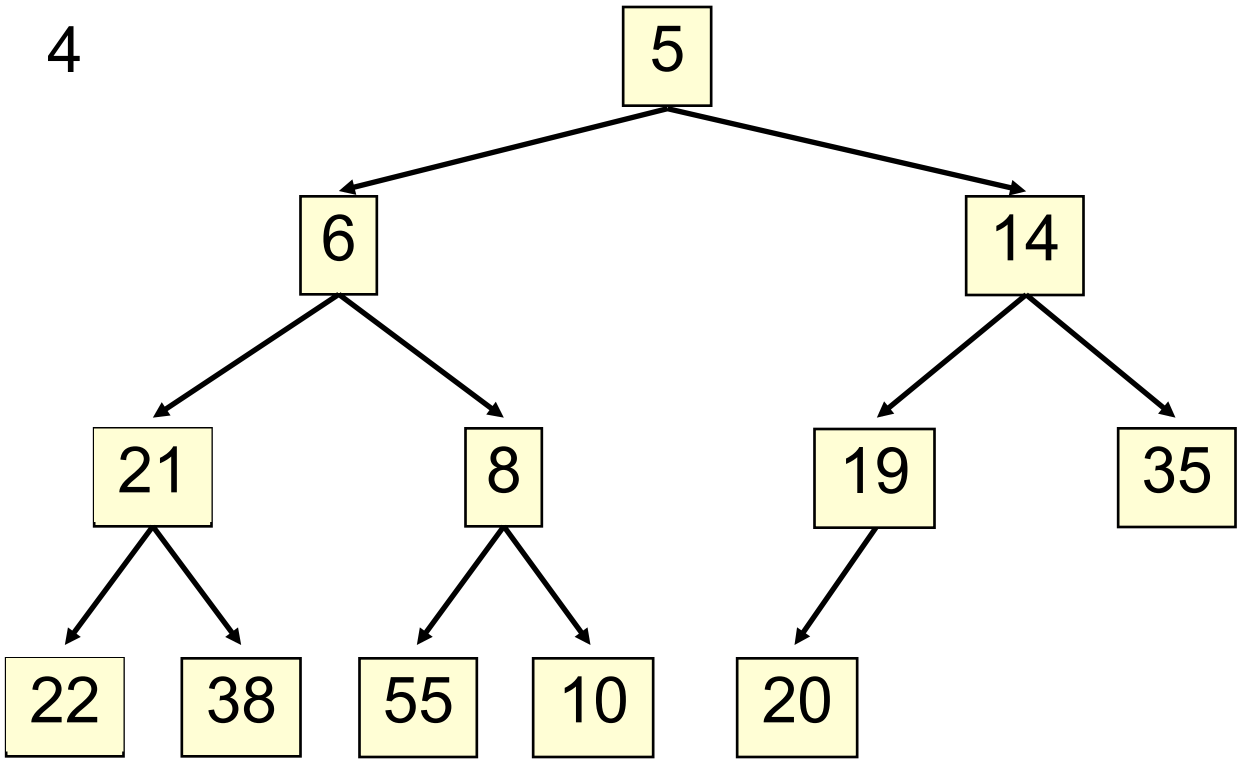
poll()

4



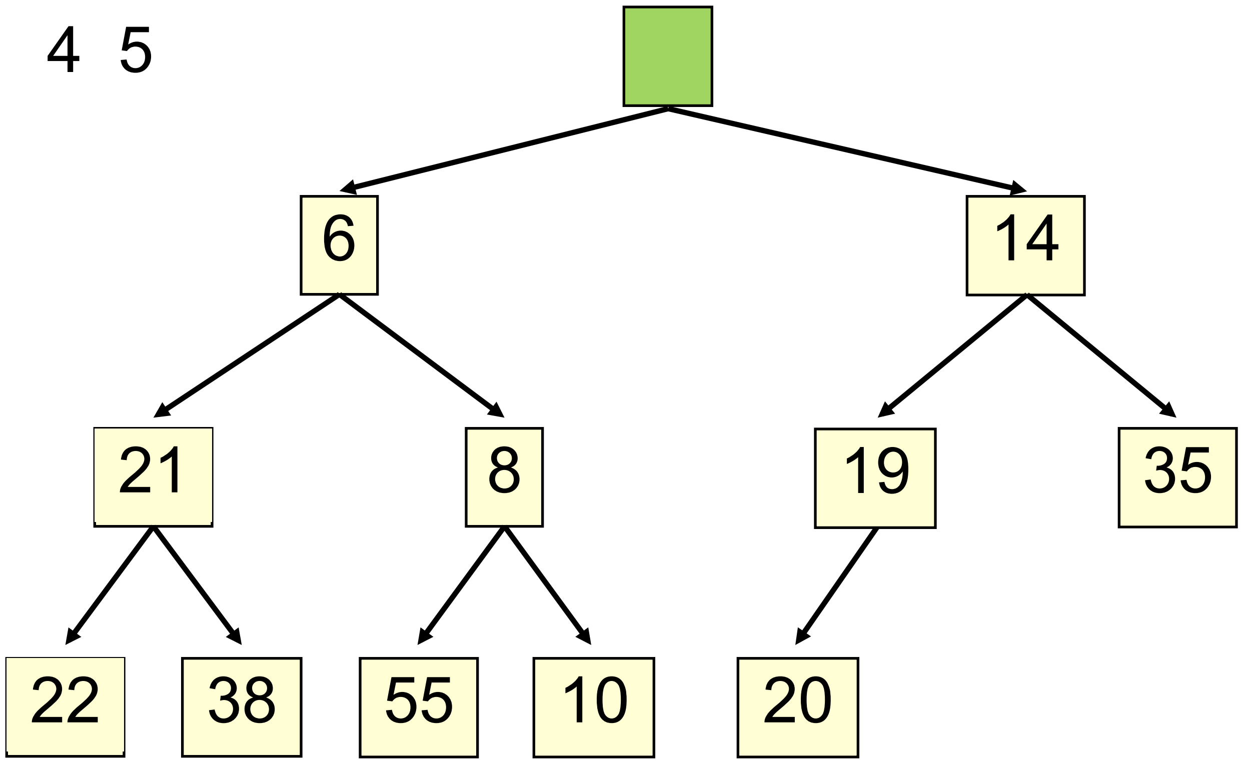
poll()

4

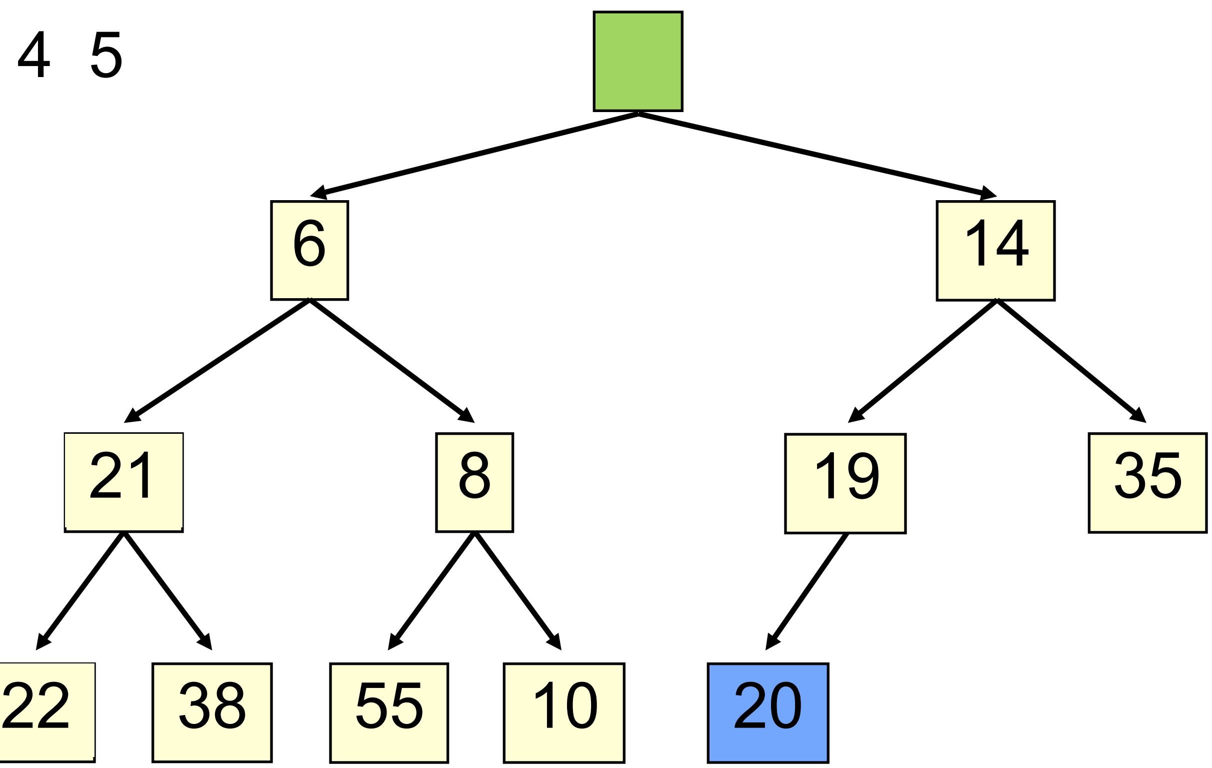


poll()

4 5

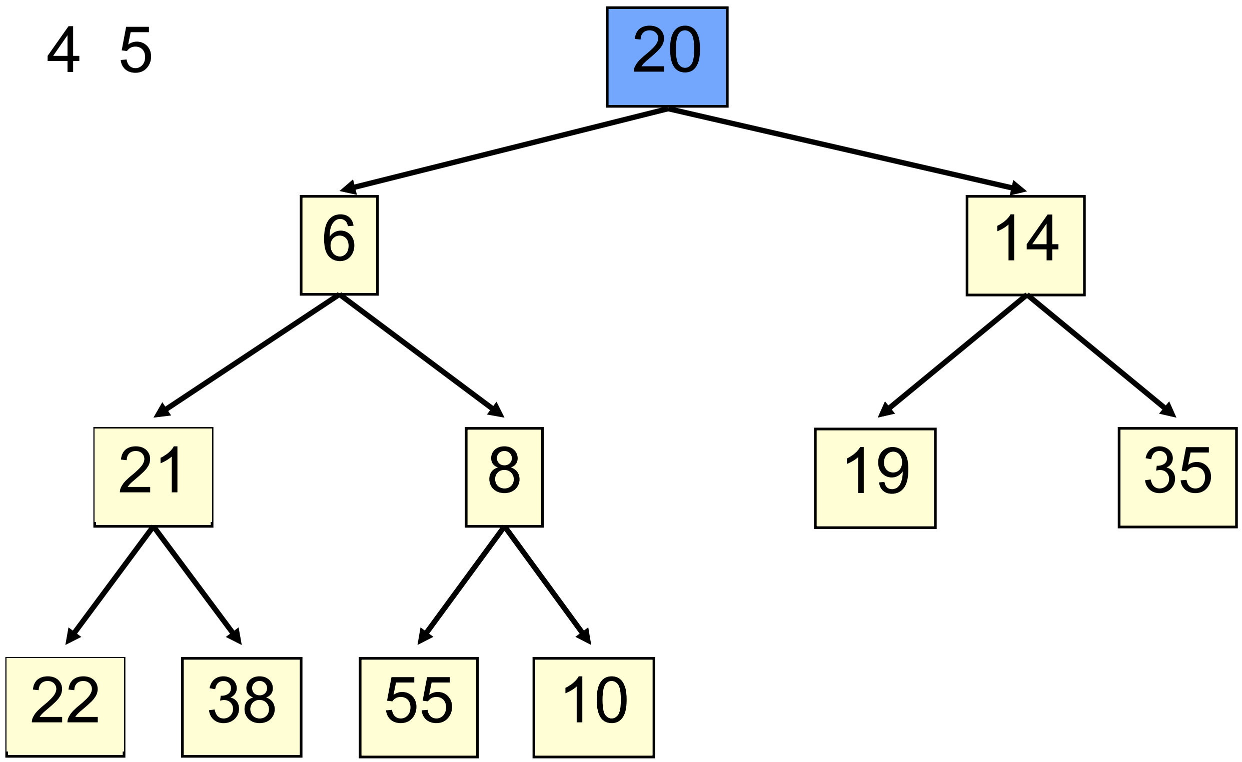


`poll()`



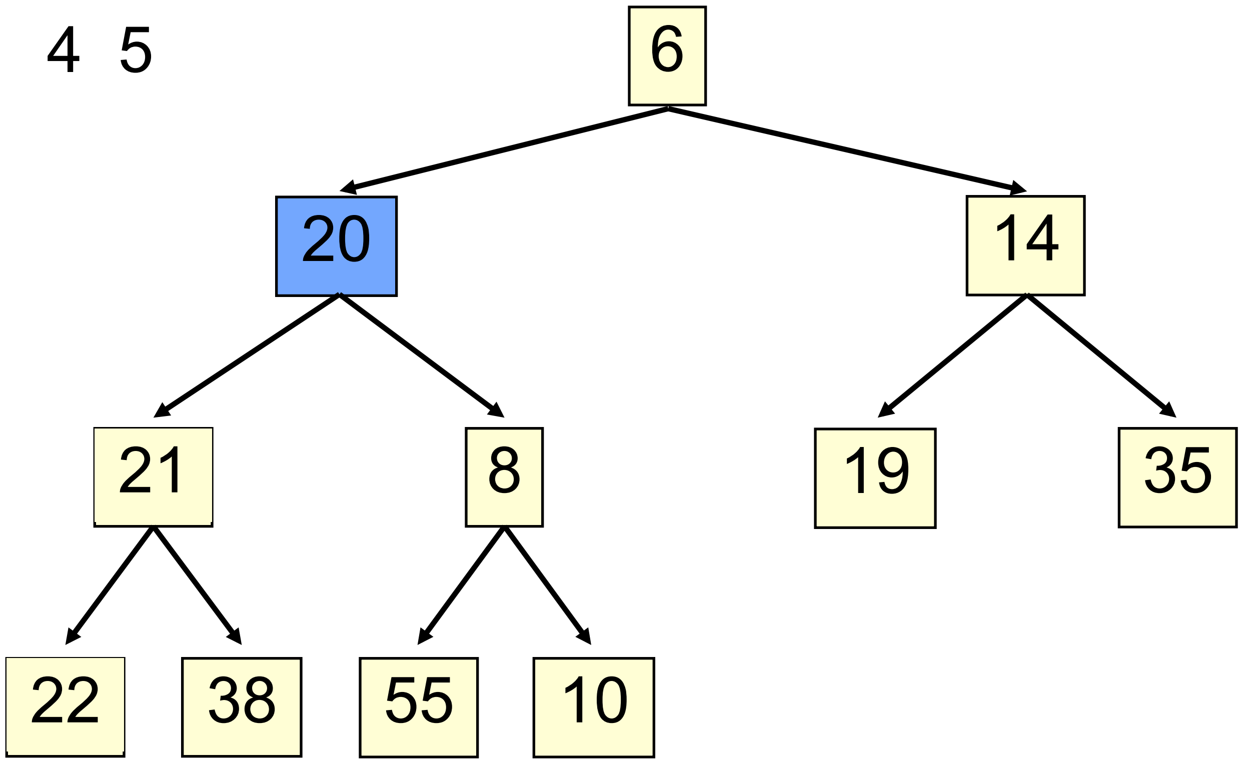
poll()

4 5



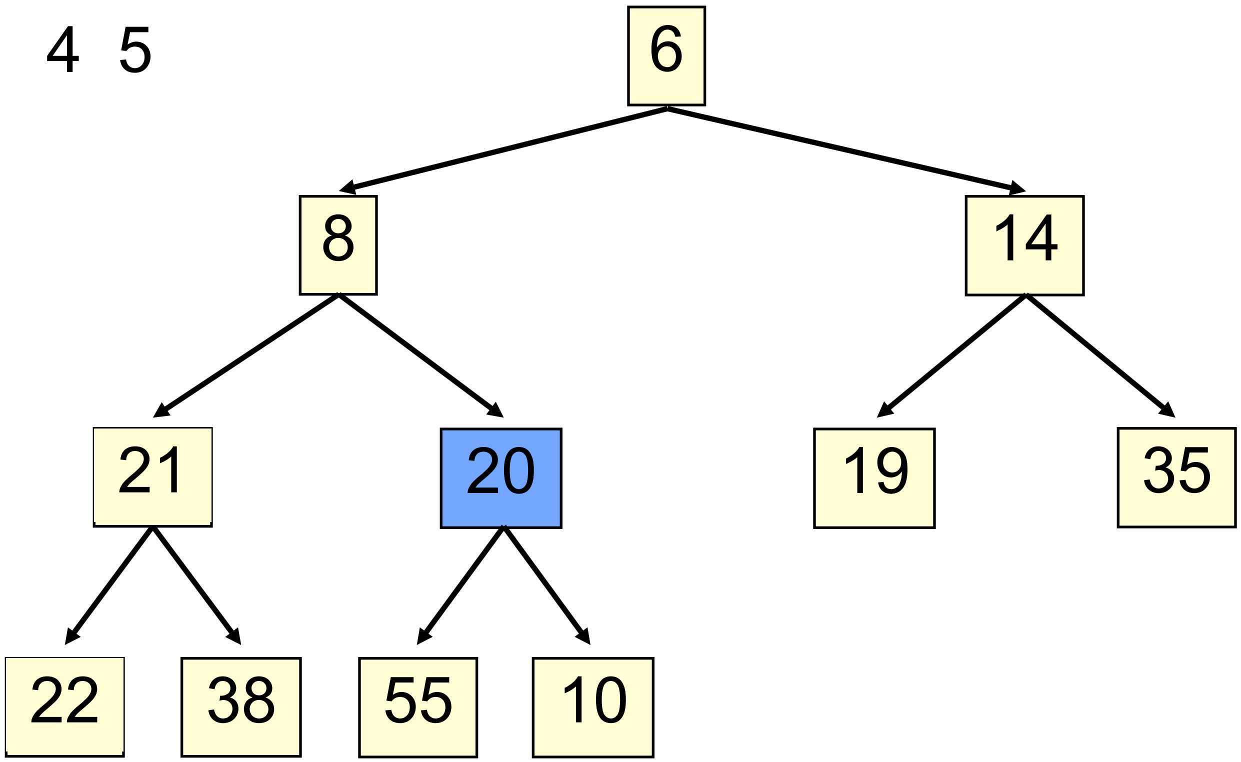
`poll()`

4 5



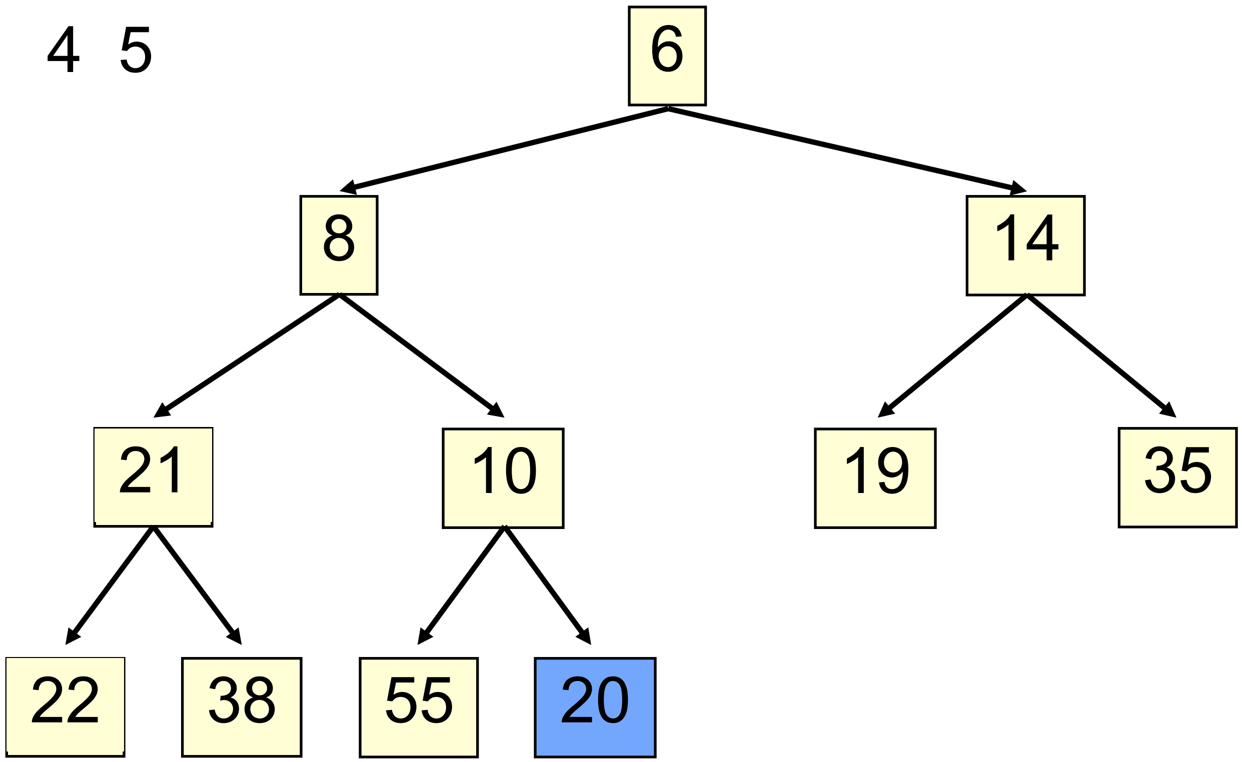
poll()

4 5



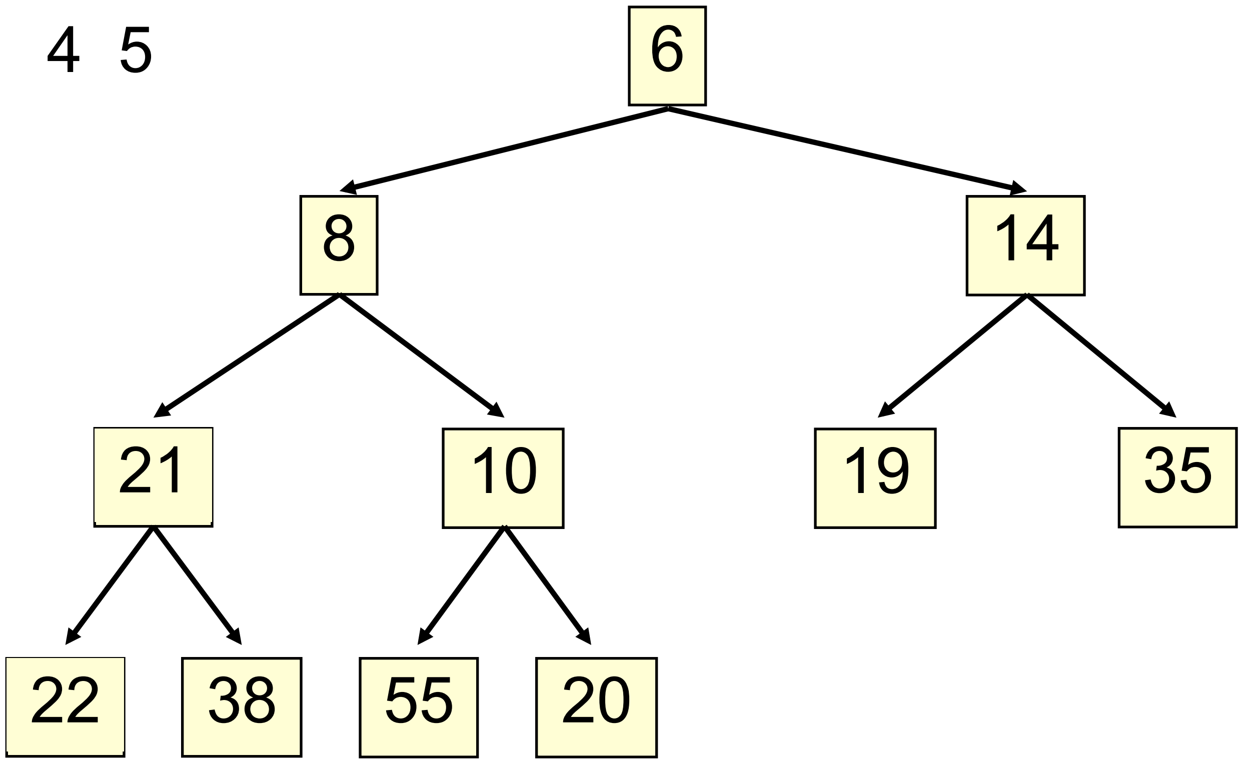
`poll()`

4 5



`poll()`

4 5



`poll()`

Time is O(log n), since the tree is balanced

`poll()`. Remember, heap is in b[0..n-1]


```
/** Remove and return the smallest element
     (return null if list is empty) */
public E poll() {
    if (n == 0) return null;
    E val=  b[0];     // smallest value is at root
    b[0]= b[n-1];     // move last element to root
    n= n - 1;
    bubbleDown(0);
    return val;
}
```

```
/** Bubble root down to its heap position.
   Pre: b[0..n-1] is a heap except: b[0] may be > than a child */
private void bubbleDown() {
   int k= 0;
  // Set c to smaller of k's children
  int c= 2*k + 2;     // k's right child
  if (c >= n || b[c-1].compareTo(b[c]) < 0)  c= c-1;

  // inv: b[0..n-1] is a heap except: b[k] may be > than a child.
  //       Also, b[c] is b[k]'s smallest child
  while (c < n   &&  b[k].compareTo(b[c]) > 0) {
    Swap b[k] and b[c];
    k= c;
    c= 2*k + 2;      // k's right child
    if (c >= n || b[c-1].compareTo(b[c]) < 0)  c= c-1;
  }
}
```

HeapSort(b, n) —Sort b[0..n-1]

Whet your appetite –use heap to get exactly n log n in-place sorting algorithm. 2 steps, each is O(n log n)

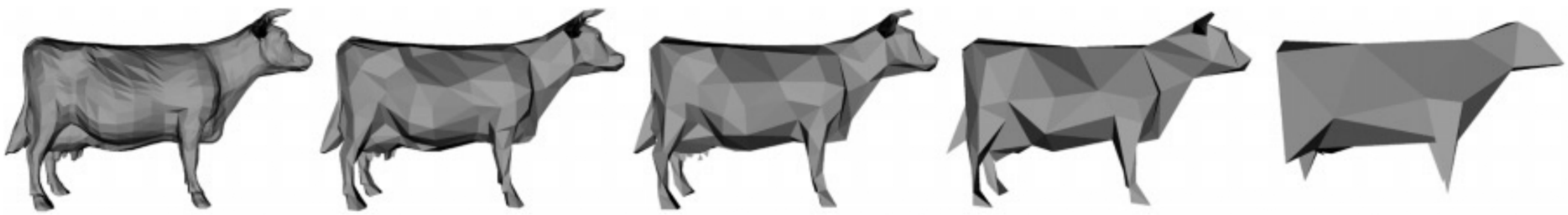
1. Make b[0..n-1] into a max-heap (in place)

2. for (k= n-1; k > 0; k= k-1) {
       b[k]= poll –i.e. take max element out of heap.
   }

We'll post this algorithm on course website

A max-heap has max value at root

Many uses of priority queues & heaps



Surface simplification [Garland and Heckbert 1997]

- Mesh compression: quadric error mesh simplification
- Event-driven simulation: customers in a line
- Collision detection: "next time of contact" for colliding bodies
- Data compression: Huffman coding
- Graph searching: Dijkstra's algorithm, Prim's algorithm
- AI Path Planning: A* search
- Statistics: maintain largest M values in a sequence
- Operating systems: load balancing, interrupt handling
- Discrete optimization: bin packing, scheduling
- Spam filtering: Bayesian spam filter