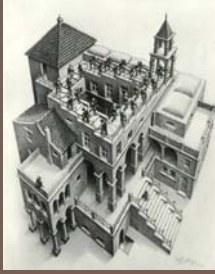


1




PRIORITY QUEUES AND HEAPS

Lecture 18
CS2110 Fall 2013

Readings and Homework

2

- Read Chapter 26 to learn about heaps
- 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!

The Bag Interface

3

- A Bag:

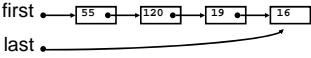
```
interface Bag<E> {
    void insert(E obj);
    E extract(); //extract some element
    boolean isEmpty();
}
```

Examples: Stack, Queue, PriorityQueue

Stacks and Queues as Lists

4

- Stack (LIFO) implemented as list
 - insert(), extract() from front of list
- Queue (FIFO) implemented as list
 - insert() on back of list, extract() from front of list
- All Bag operations are $O(1)$



Priority Queue

5

- A Bag in which data items are Comparable
- lesser elements (as determined by compareTo()) have higher priority
- extract() returns the element with the highest priority = least in the compareTo() ordering
- break ties arbitrarily

Priority Queue Examples

6

- 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

```

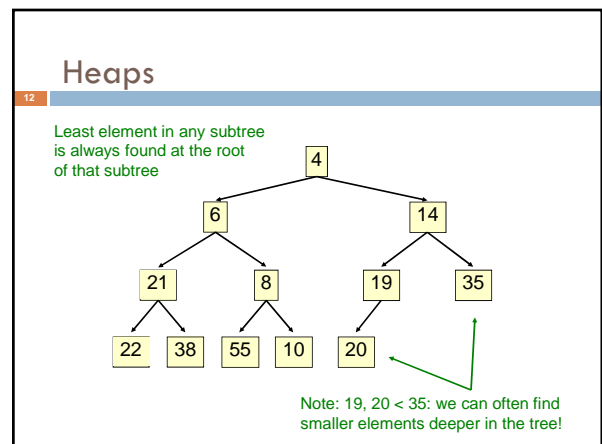
java.util.PriorityQueue<E>
7
boolean add(E e) {...} //insert an element (insert)
void clear() {...} //remove all elements
E peek() {...} //return min element without removing
// (null if empty)
E poll() {...} //remove min element (extract)
// (null if empty)
int size() {...}
    
```

- ### Priority Queues as Lists
- Maintain as **unordered list**
 - `insert()` puts new element at front - $O(1)$
 - `extract()` must search the list - $O(n)$
 - Maintain as **ordered list**
 - `insert()` must search the list - $O(n)$
 - `extract()` gets element at front - $O(1)$
 - In either case, $O(n^2)$ to process n elements
- Can we do better?

- ### Important Special Case
- Fixed number of priority levels $0, \dots, p - 1$
 - FIFO within each level
 - Example: airline check-in
 - `insert()` - insert in appropriate queue - $O(1)$
 - `extract()` - must find a nonempty queue - $O(p)$

- ### Heaps
- 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:
 - `insert()`: $O(\log n)$
 - `extract()`: $O(\log n)$
 - $O(n \log n)$ 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*

- ### Heaps
- Binary tree with data at each node
 - Satisfies the *Heap Order Invariant*:
- The least (highest priority) element of any subtree is found at the root of that subtree
- Size of the heap is "fixed" at n . (But can usually double n if heap fills up)



Examples of Heaps

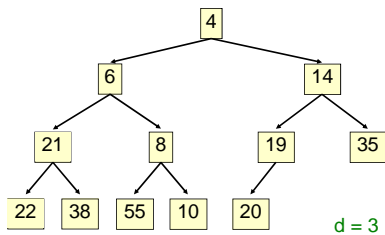
- Ages of people in family tree
 - parent is always older than children, but you can have an uncle who is younger than you
- Salaries of employees of a company
 - bosses generally make more than subordinates, but a VP in one subdivision may make less than a Project Supervisor in a different subdivision

Balanced Heaps

These add two restrictions:

1. Any node of depth $< d - 1$ has exactly 2 children, where d is the height of the tree
 - implies that any two maximal paths (path from a root to a leaf) are of length d or $d - 1$, and the tree has at least 2^d nodes
- All maximal paths of length d are to the left of those of length $d - 1$

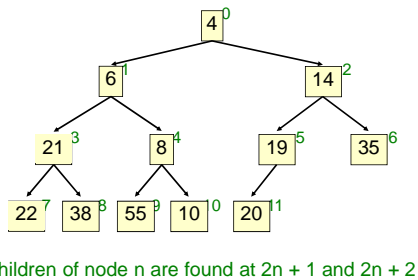
Example of a Balanced Heap



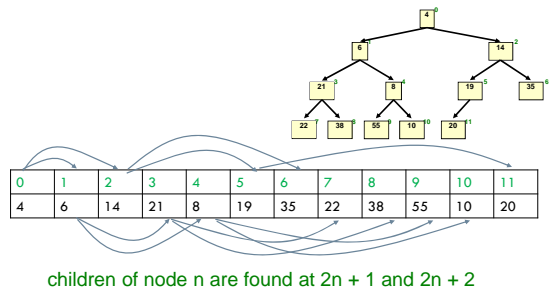
Store in an ArrayList or Vector

- Elements of the heap are stored in the array in order, going across each level from left to right, top to bottom
- The children of the node at array index n are found at $2n + 1$ and $2n + 2$
- The parent of node n is found at $(n - 1)/2$

Store in an ArrayList or Vector



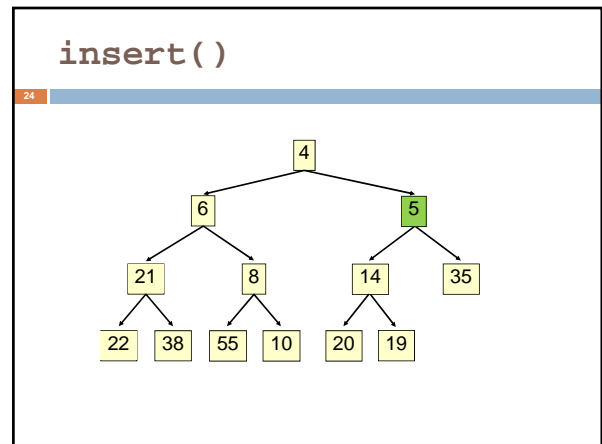
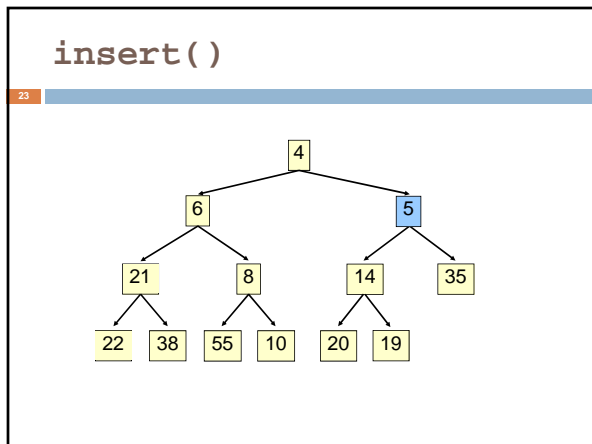
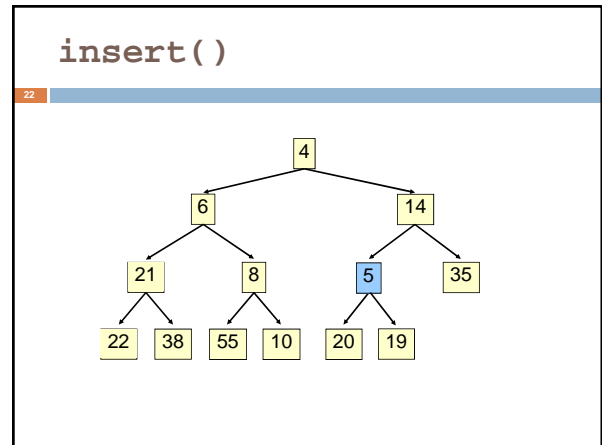
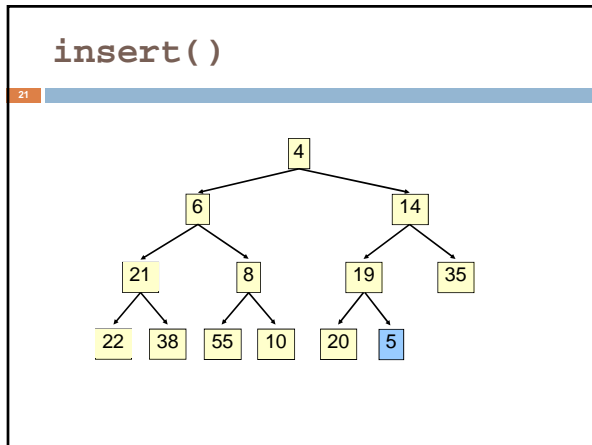
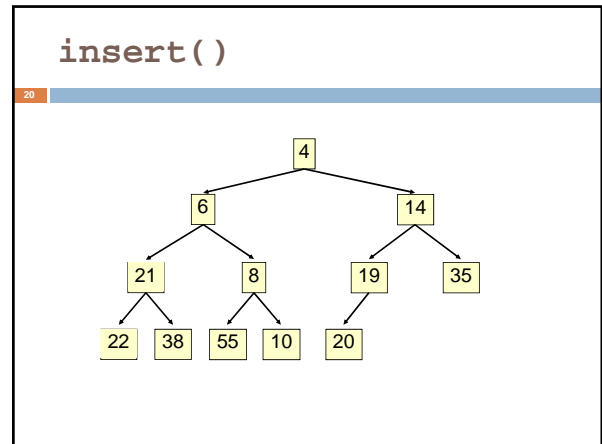
Store in an ArrayList or Vector

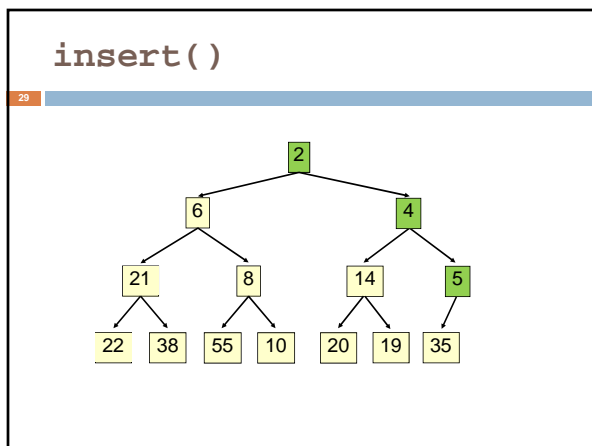
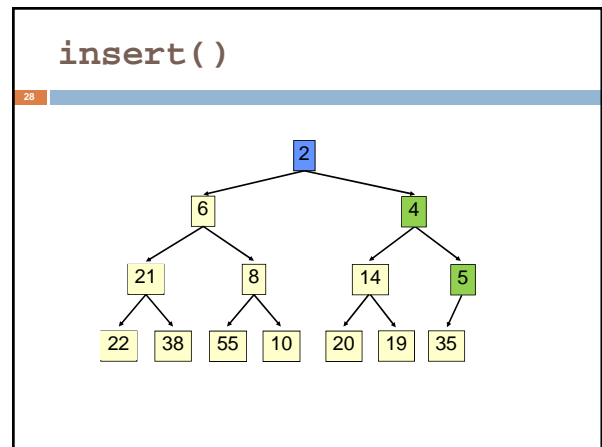
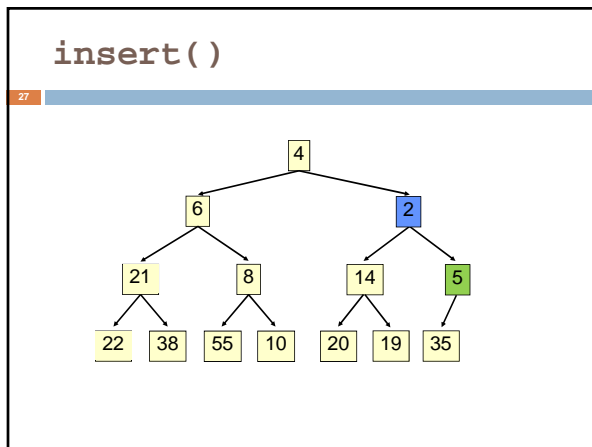
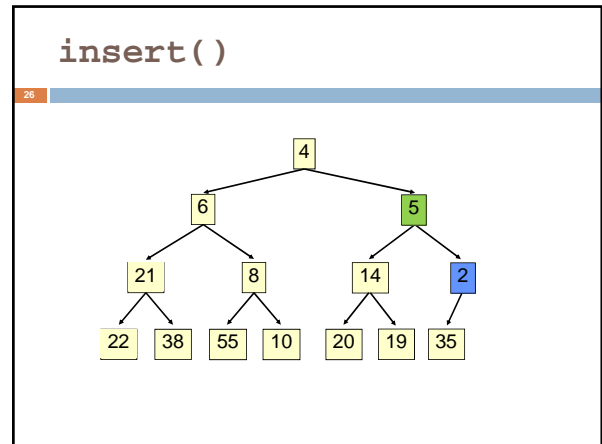
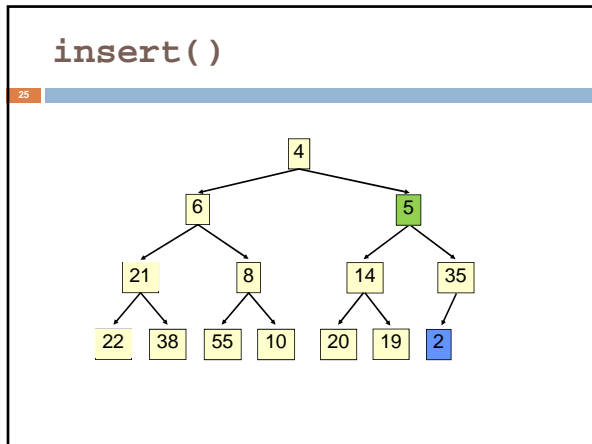


insert()

19

- Put the new element 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!





insert()

30

- 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

insert()

31

```
class PriorityQueue<E> extends java.util.Vector<E> {
    public void insert(E obj) {
        super.add(obj); //add new element to end of array
        rotateUp(size() - 1);
    }
    private void rotateUp(int index) {
        if (index == 0) return;
        int parent = (index - 1)/2;
        if (elementAt(parent).compareTo(elementAt(index)) <= 0)
            return;
        swap(index, parent);
        rotateUp(parent);
    }
}
```

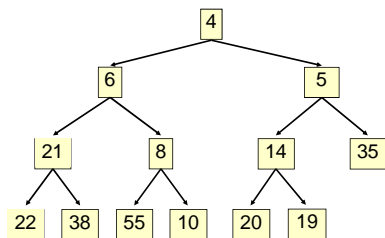
extract()

32

- Remove the least element – it is 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!

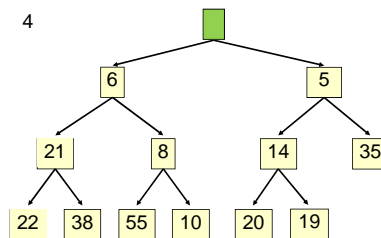
extract()

33



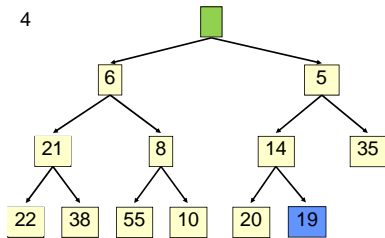
extract()

34



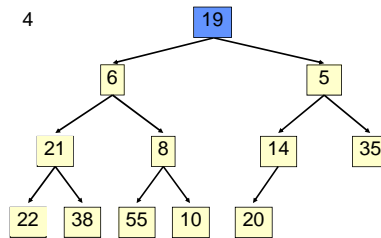
extract()

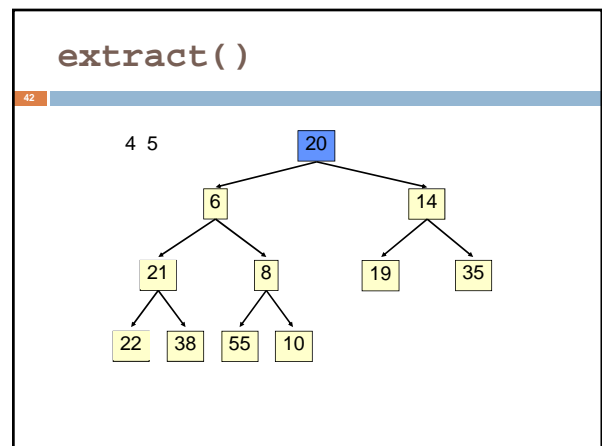
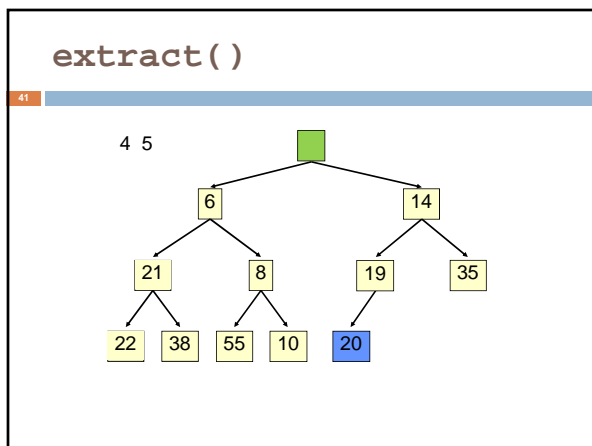
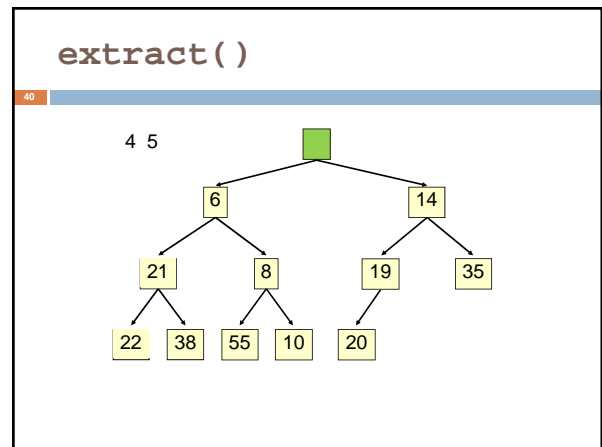
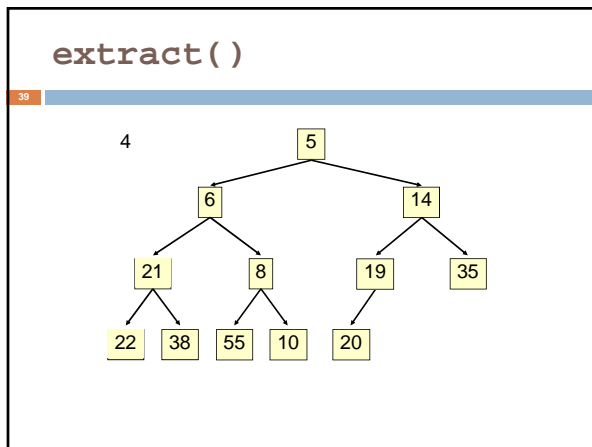
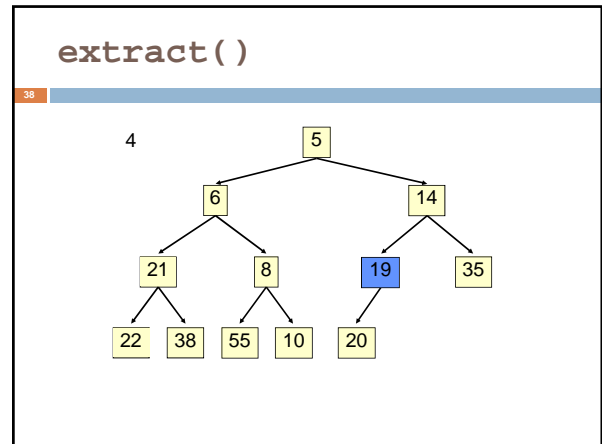
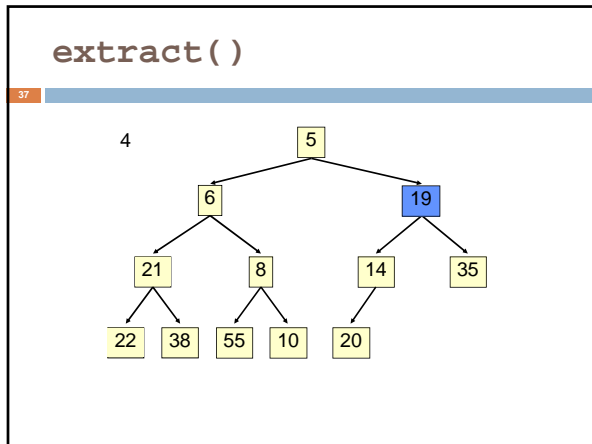
35

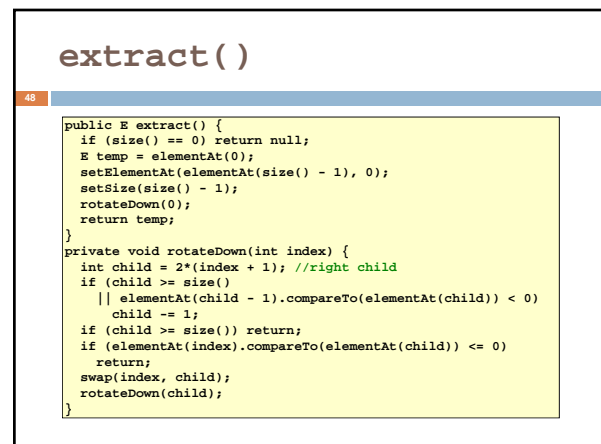
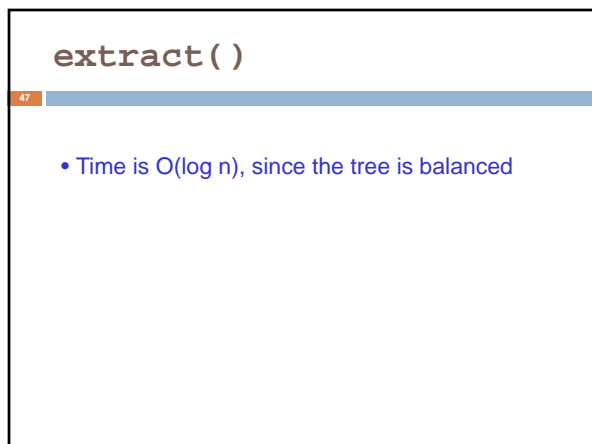
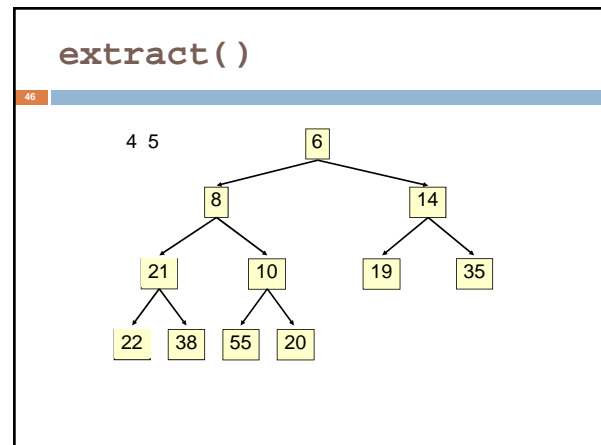
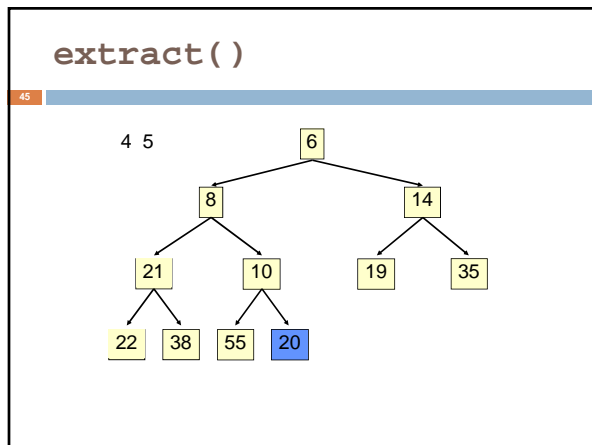
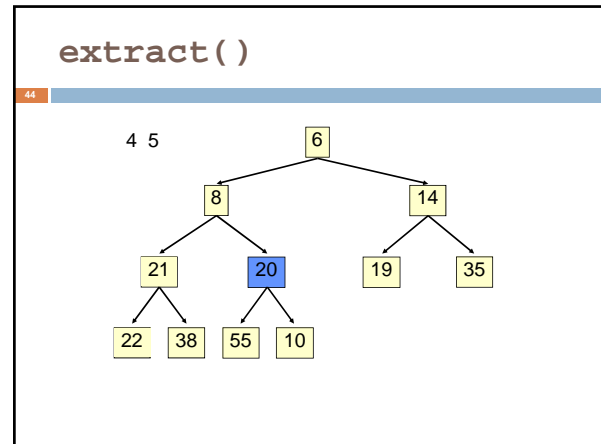
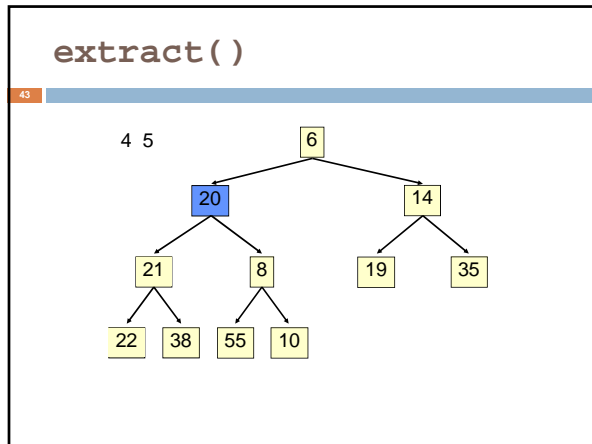


extract()

36







HeapSort

49

Given a `Comparable[]` array of length n ,

- Put all n elements into a heap – $O(n \log n)$
- Repeatedly get the min – $O(n \log n)$

```
public static void heapSort(Comparable[] a) {
    PriorityQueue<Comparable> pq = new PriorityQueue<Comparable>(a);
    for (int i = 0; i < a.length; i++) { a[i] = pq.extract(); }
}
```

PQ Application: Simulation

50

□ Example: Probabilistic model of bank-customer arrival times and transaction times, how many tellers are needed?

- Assume we have a way to generate random inter-arrival times
- Assume we have a way to generate transaction times
- Can simulate the bank to get some idea of how long customers must wait

Time-Driven Simulation

- Check at each *tick* to see if any event occurs

Event-Driven Simulation

- Advance clock to next event, skipping intervening *ticks*
- This uses a PQ!