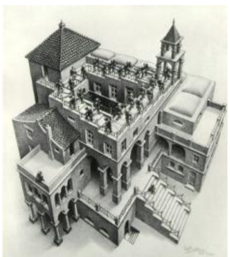
CS/ENGRD 2110
Object-Oriented Programming
and Data Structures

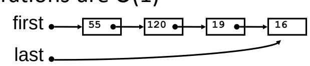


Spring 2011
Thorsten Joachims

Lecture 17: Heaps and Priority Queues

Stacks and Queues as Lists

- Stack (LIFO) implemented as list
  - insert (i.e. push) to, extract (i.e. pop) from front of list
- Queue (FIFO) implemented as list
  - insert (i.e. add) on back of list, extract (i.e. poll) from front of list
- All operations are O(1)



Priority Queue

- ADT Definition
  - data items are **`Comparable`**
  - *lesser* elements (as determined by **`compareTo()`**) have *higher* priority
  - **`extract()`** returns the element with the highest priority
    - i.e. least in the **`compareTo()`** ordering
  - break ties arbitrarily
    - alternatively could break ties FIFO, but lets keep it simple

Priority Queue Examples

- 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>

```
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 (i.e. queue)
  - **`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 (and small) number of p priority levels
  - Queue within each level
  - Example: airline check-in
- insert() – insert in appropriate queue – O(1)
- extract() – must find a nonempty queue – O(p)

Heaps

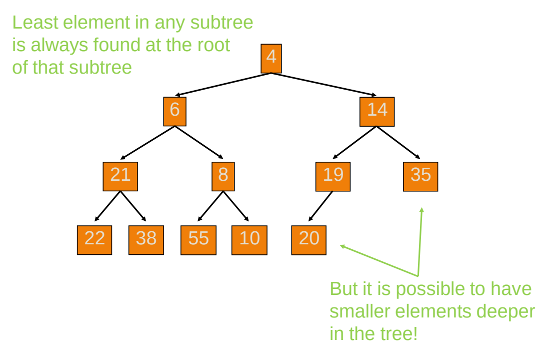
- A *heap* is a concrete data structure that can be used to implement priority queues
- Gives better complexity than either ordered or unordered list implementation:
  - insert(): O(log n)
  - extract(): O(log n)
  - → O(n log n) to process n elements

NOTE: Do not confuse with heap memory, where the Java virtual machine allocates space for objects – different usage of the word heap

Heap Invariant

- 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.



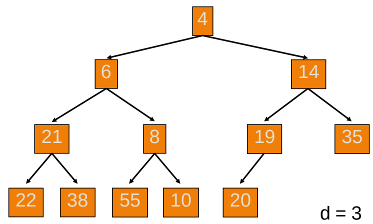
Examples of Heaps

- 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

- Two restrictions:
  - 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 2d nodes
  - All maximal paths of length d are to the left of those of length d – 1

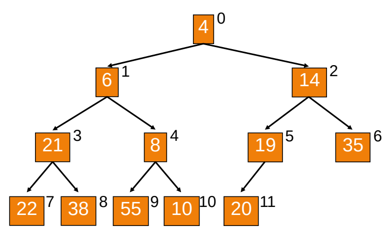
A Balanced Heap



Store in an ArrayList

- 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



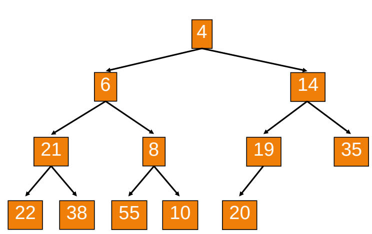
children of node n are found at 2n + 1 and 2n + 2

insert()

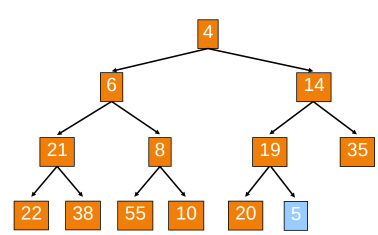
- Put the new element at the end of the array
- If this violates heap order because it is smaller than its parent, swap it with its parent
- Continue swapping it up until it finds its rightful place

→ The heap invariant is maintained!

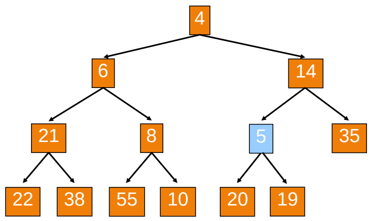
insert() Example



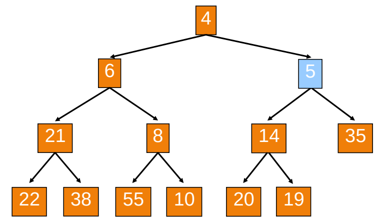
insert() Example



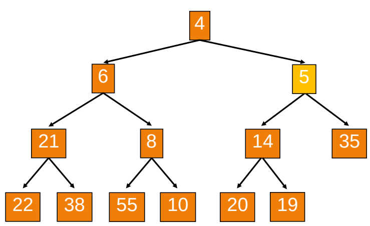
insert() Example



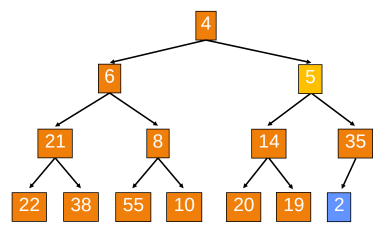
insert() Example



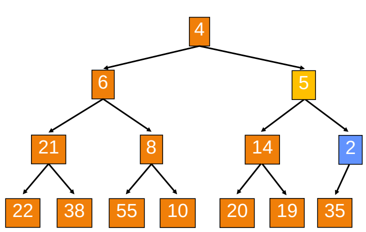
insert() Example



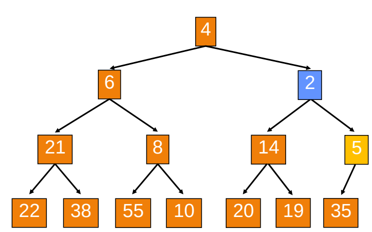
insert() Example



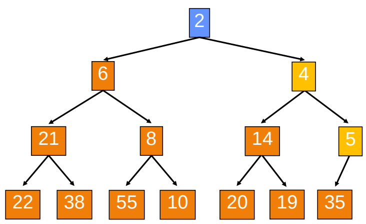
insert() Example



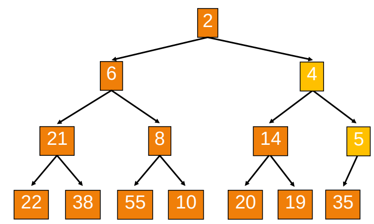
insert() Example



insert() Example



insert() Example



Analysis of insert()

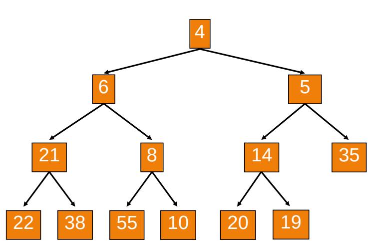
- Time is O(log n), since the tree is balanced
  - At most log(d) swaps up the tree before invariant is restored
  - size of tree is exponential as a function of depth d ⬄ depth of tree is logarithmic as a function of size n
  - Each insertion is finished after at most d <= log(n) swaps

extract()

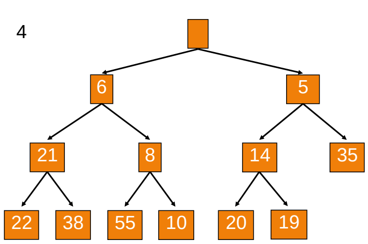
- Remove the least element – it is at the root
- This leaves a hole at the root – fill it in with the last element of the array
- If this violates heap order because the root element is too big, swap it down with the smaller of its children
- Continue swapping it down until it finds its rightful place

→ The heap invariant is maintained!

extract() Example

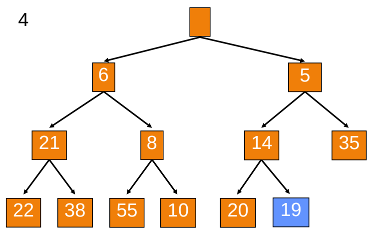


extract() Example



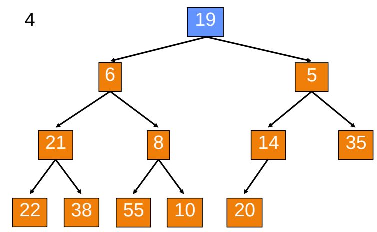
extract() Example

4



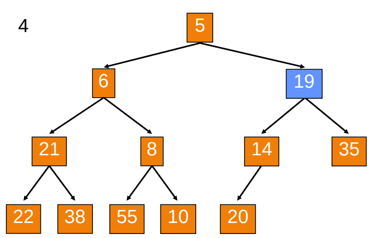
extract() Example

4



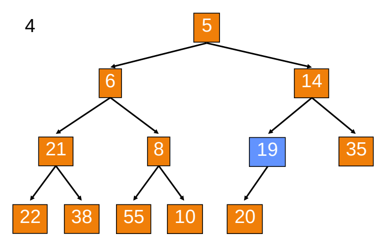
extract() Example

4



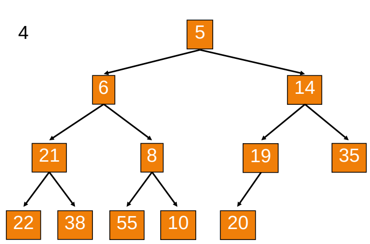
extract() Example

4



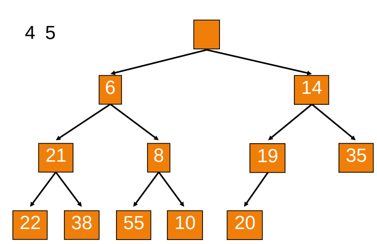
extract() Example

4



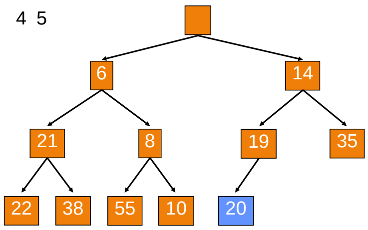
extract() Example

4 5



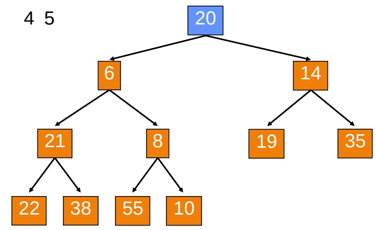
extract() Example

4 5



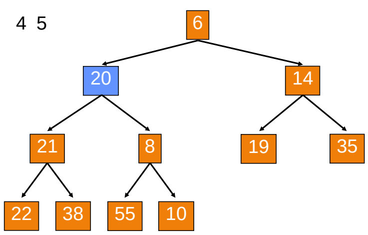
extract() Example

4 5



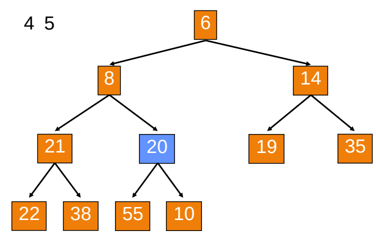
extract() Example

4 5



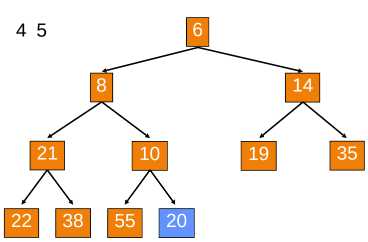
extract() Example

4 5



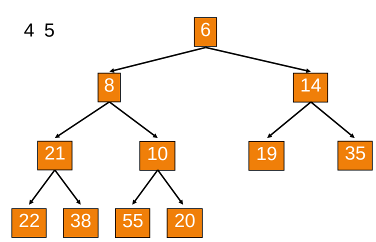
extract() Example

4 5



extract() Example

4 5



Analysis of extract()

- Time is O(log n), since the tree is balanced
  - At most log(d) swaps down towards the leaves of the tree before invariant is restored
  - size of tree is exponential as a function of depth d ⇔ depth of tree is logarithmic as a function of size n
  - Each extraction is finished after at most d <= log(n) swaps

HeapSort

- Given a Comparable[] array of length n
- Put all n elements into a heap – O(n log n)
- Repeatedly get the min and sequentially put into new array – O(n log n)