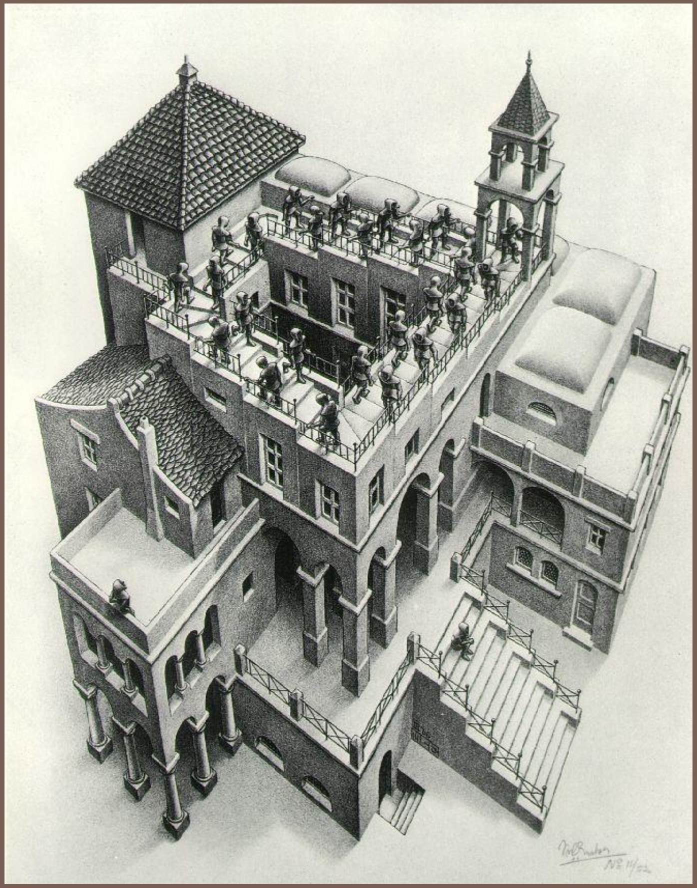


PRIORITY QUEUES AND HEAPS

Lecture 18

CS2110 Spring 2013

The Bag Interface

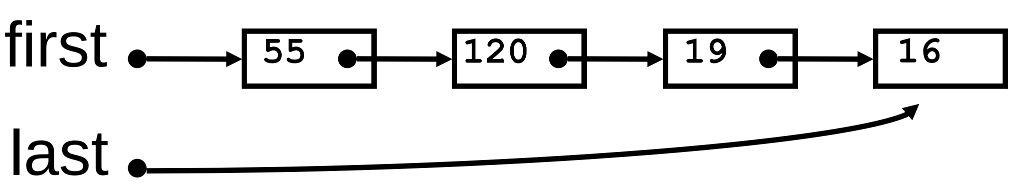
- A Bag:

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

Examples: Stack, Queue, PriorityQueue

Stacks and Queues as Lists

- 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

- 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

- 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
  - `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,...,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

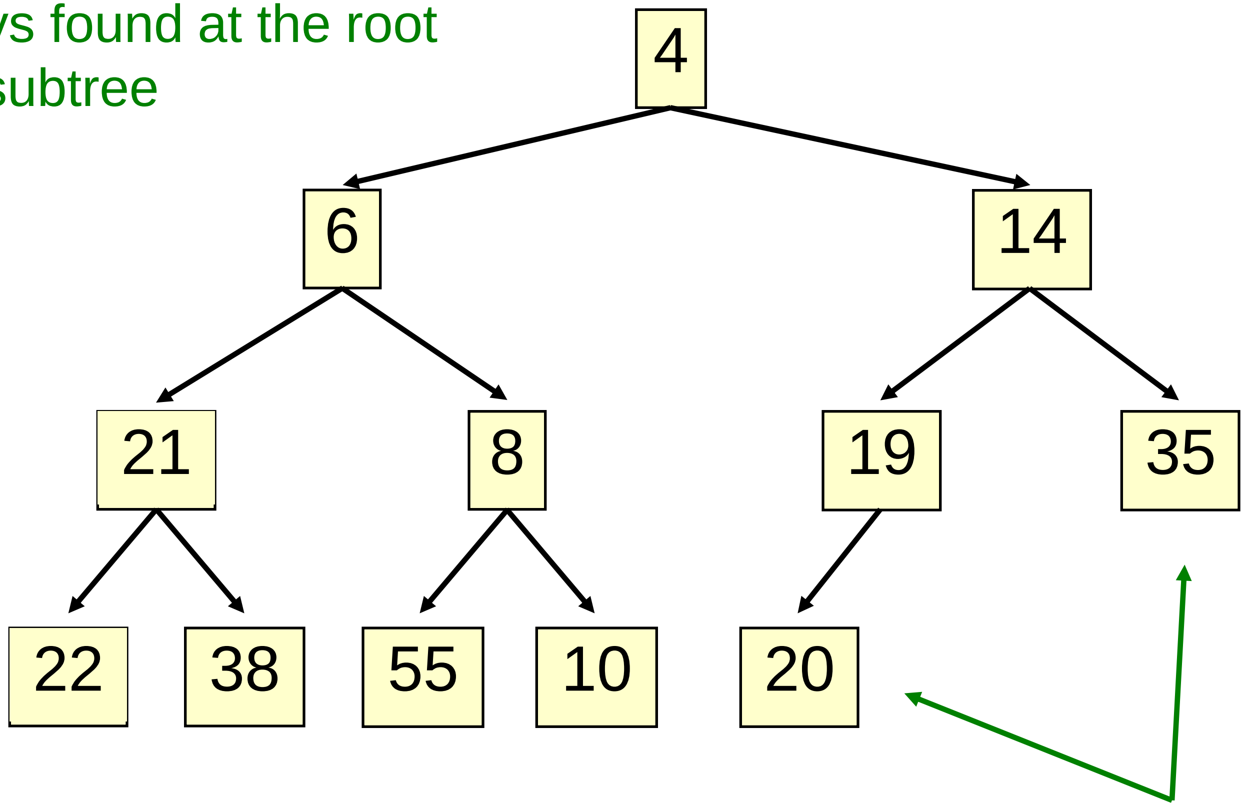
- Binary tree with data at each node
- Satisfies the *Heap Order Invariant*:

> 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)

Heaps

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



Note: 19, 20 < 35: we can often find
smaller elements deeper in the tree!

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

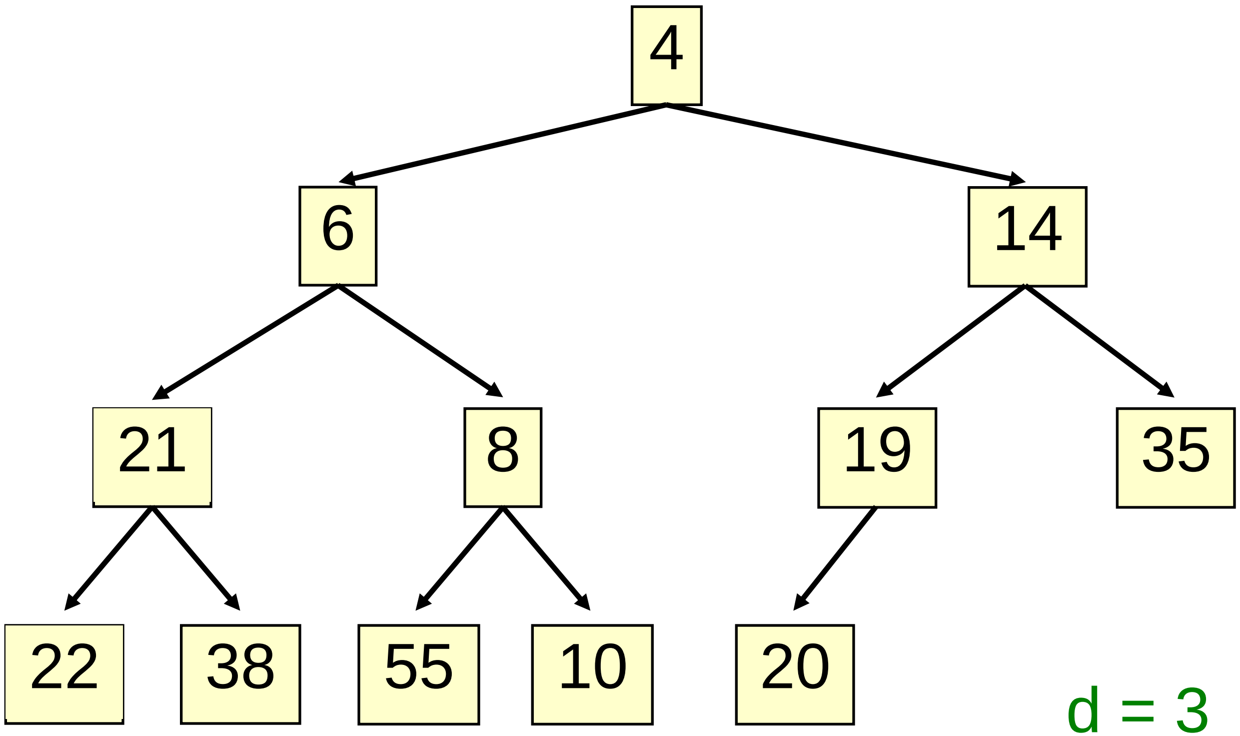
Balanced Heaps

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



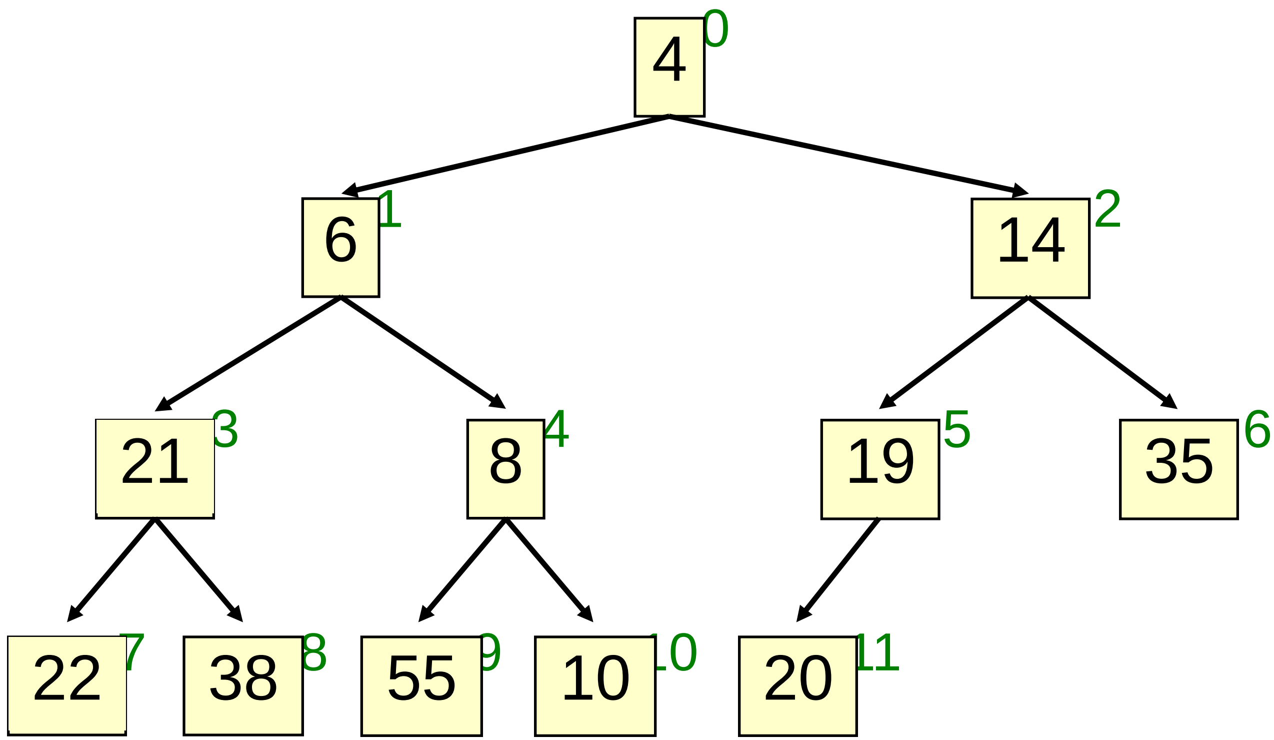
Store in an ArrayList or Vector

- 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

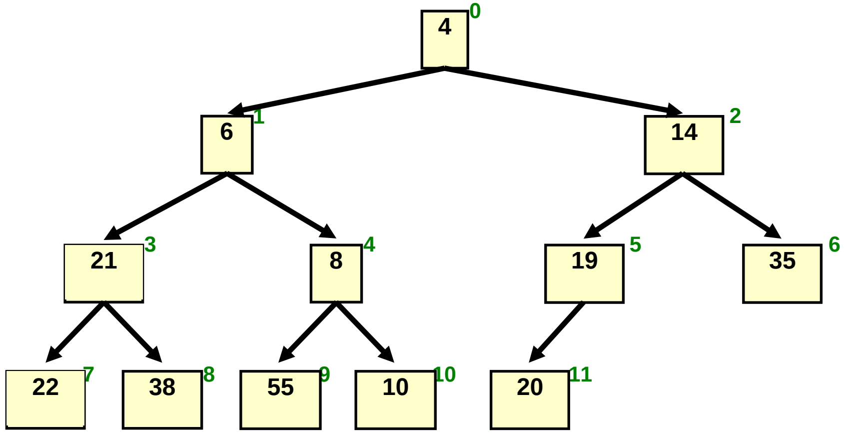
- The parent of node n is found at (n – 1)/2

Store in an ArrayList or Vector



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

Store in an ArrayList or Vector



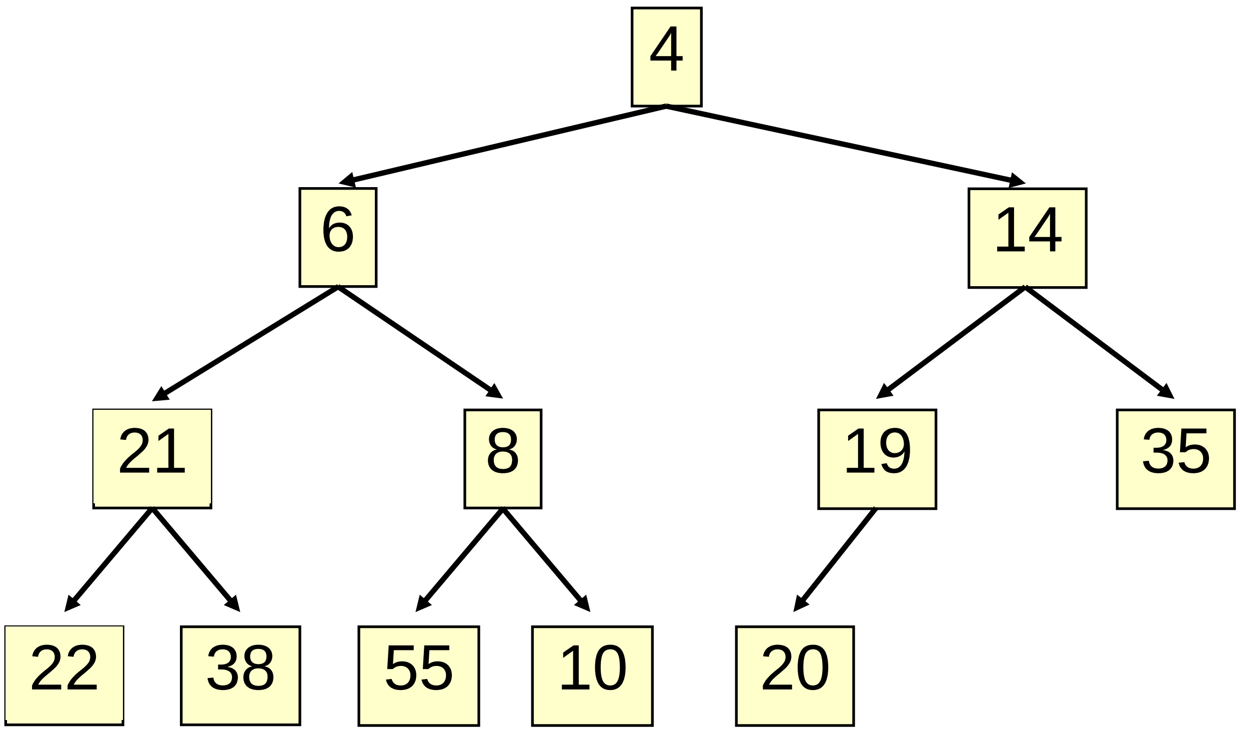
0	1	2	3	4	5	6	7	8	9	10	11
4	6	14	21	8	19	35	22	38	55	10	20

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

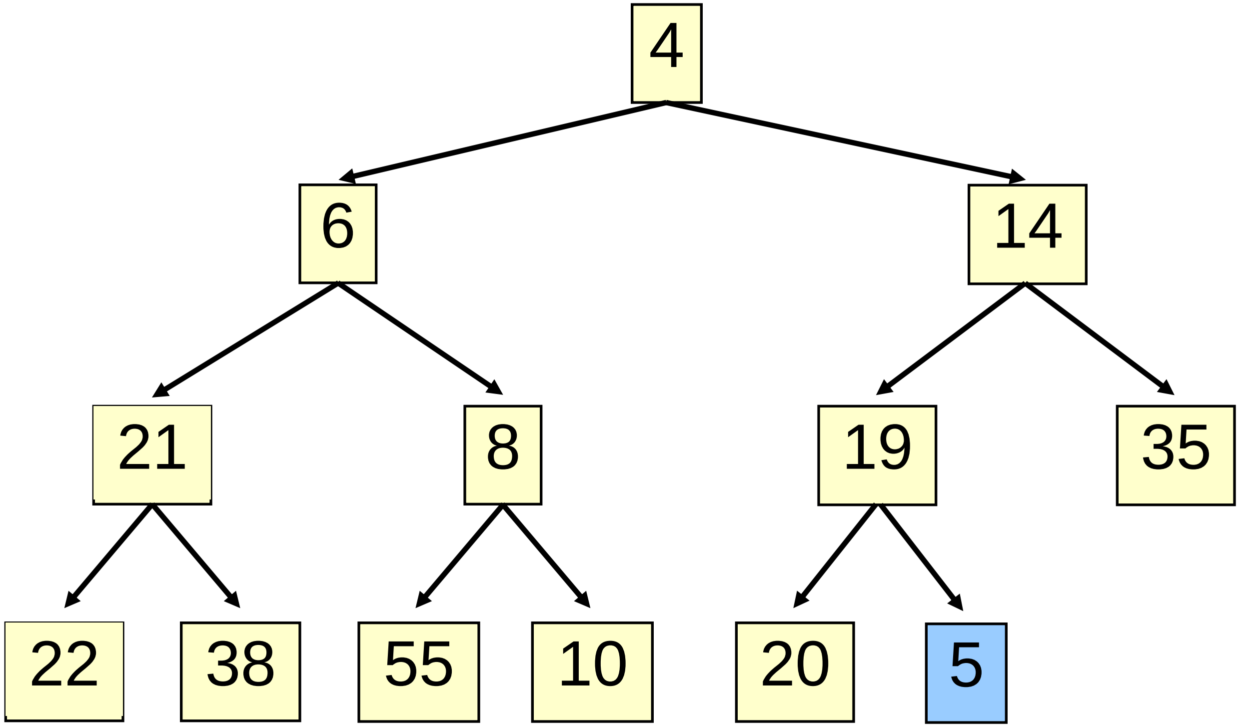
`insert()`

- 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!

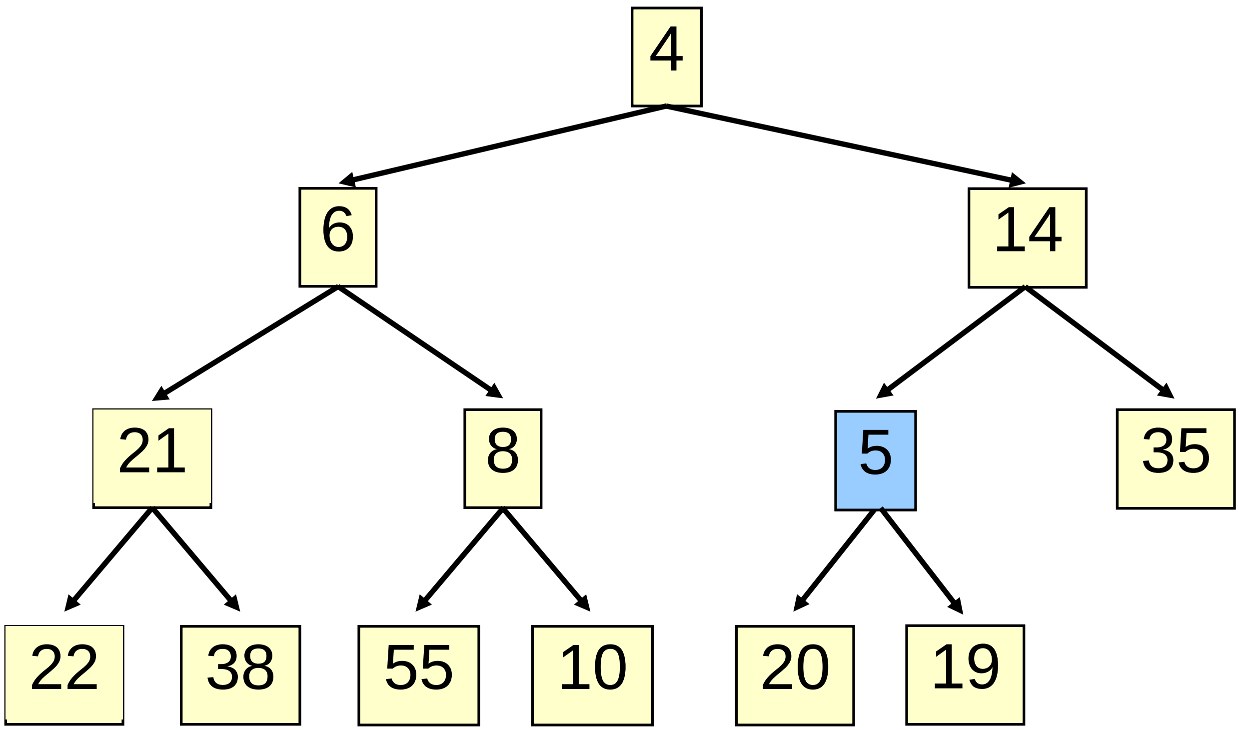
`insert()`



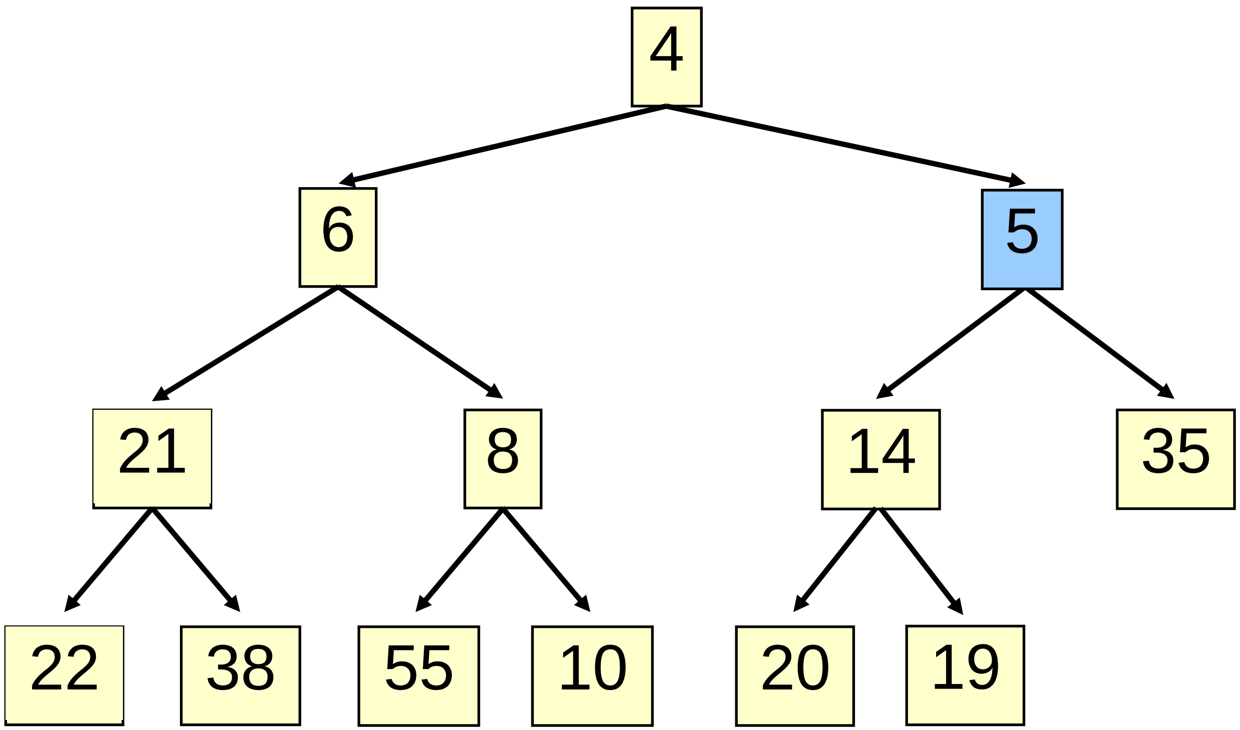
`insert()`



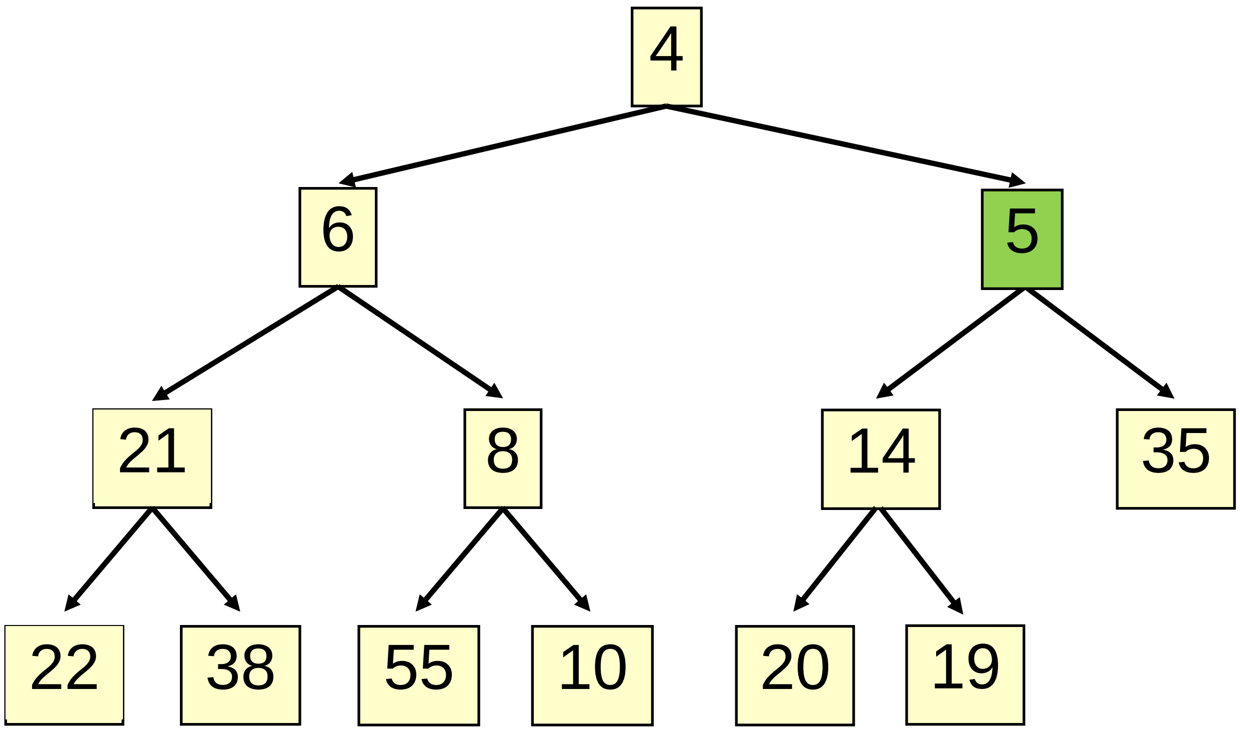
insert()



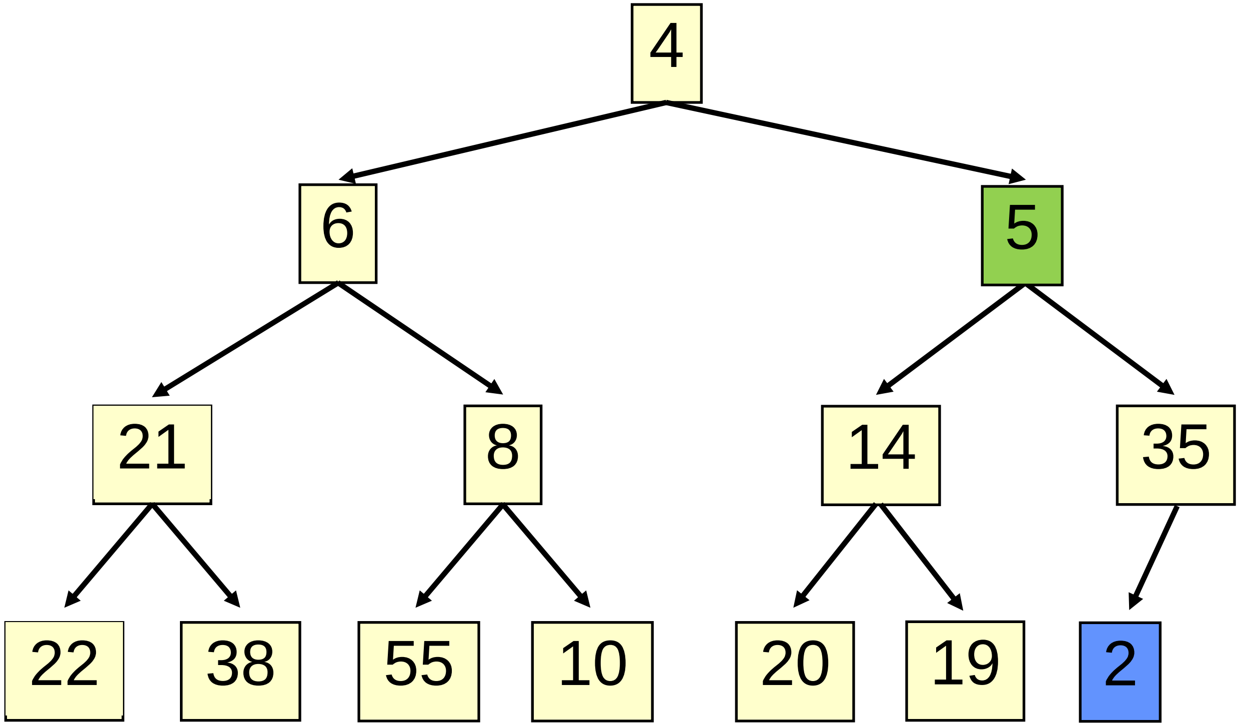
`insert()`



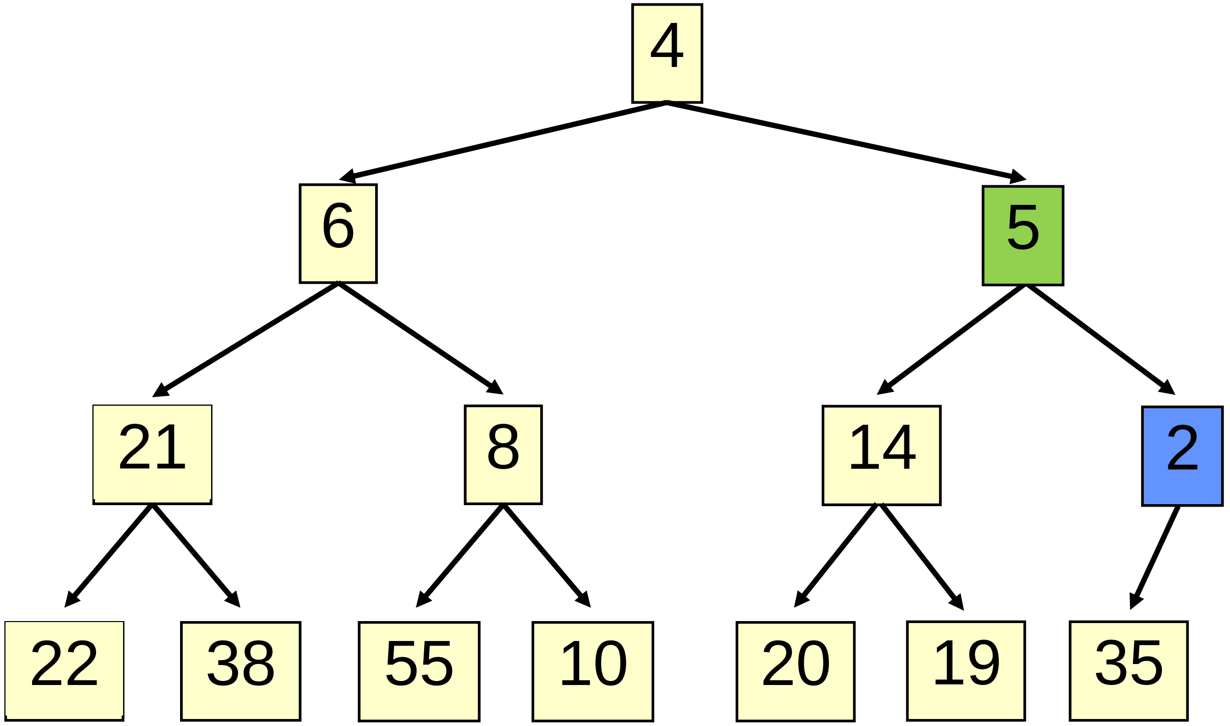
insert()



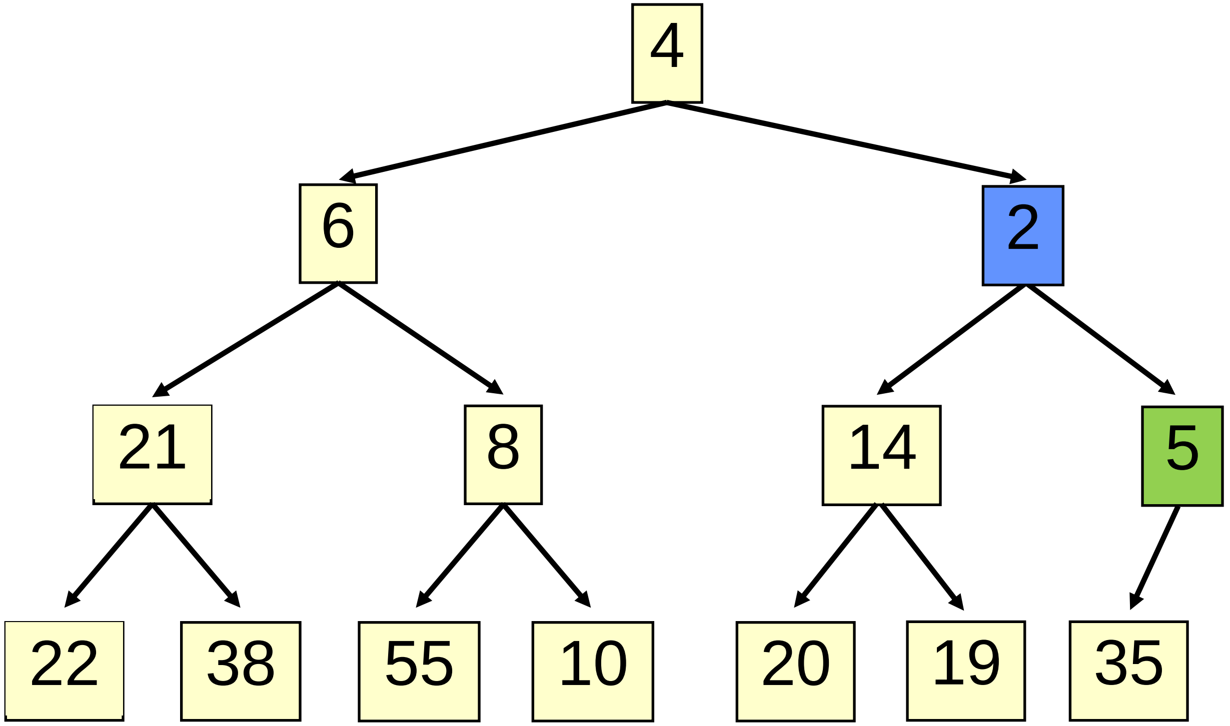
insert()



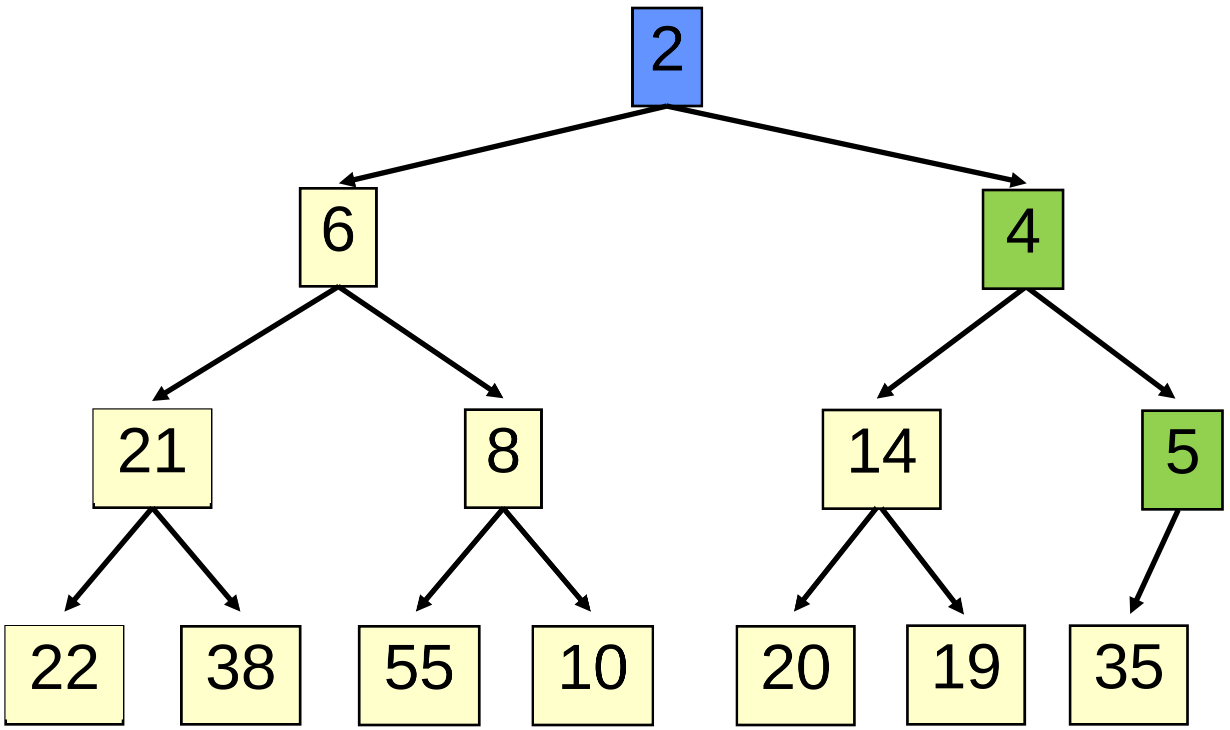
`insert()`



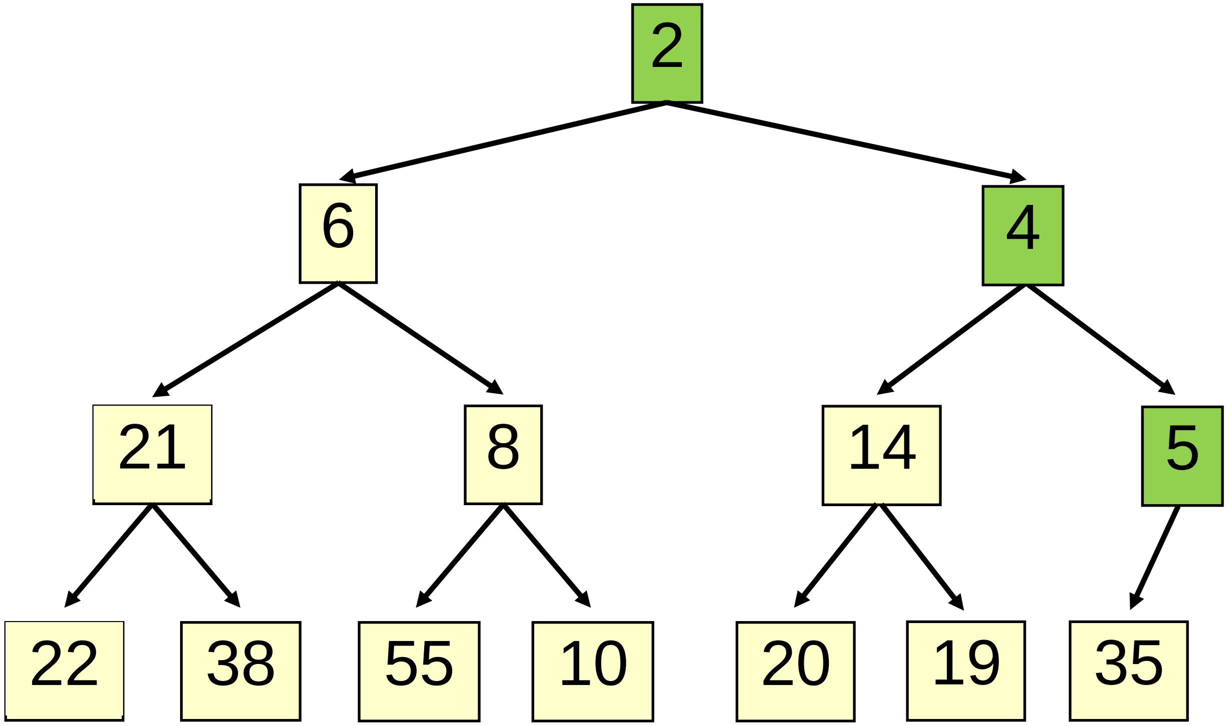
insert()



`insert()`



`insert()`



`insert()`

- 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

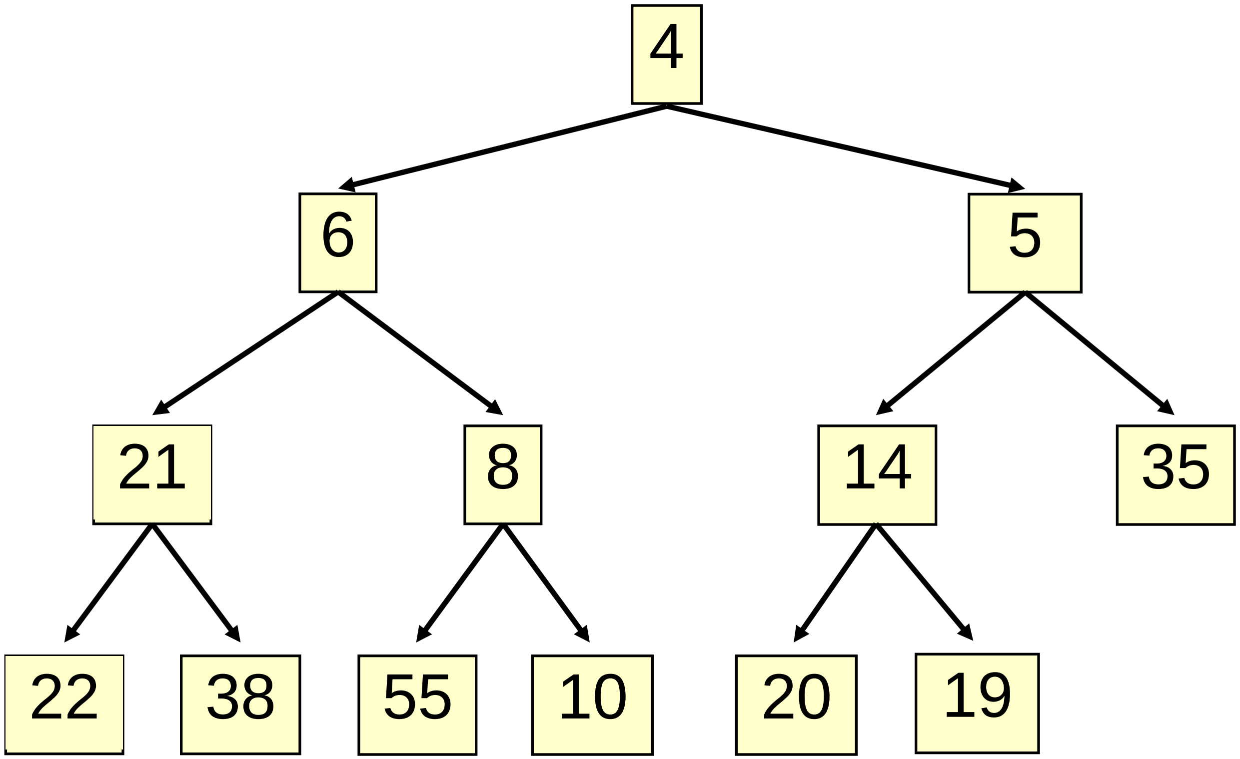
insert()


```
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()

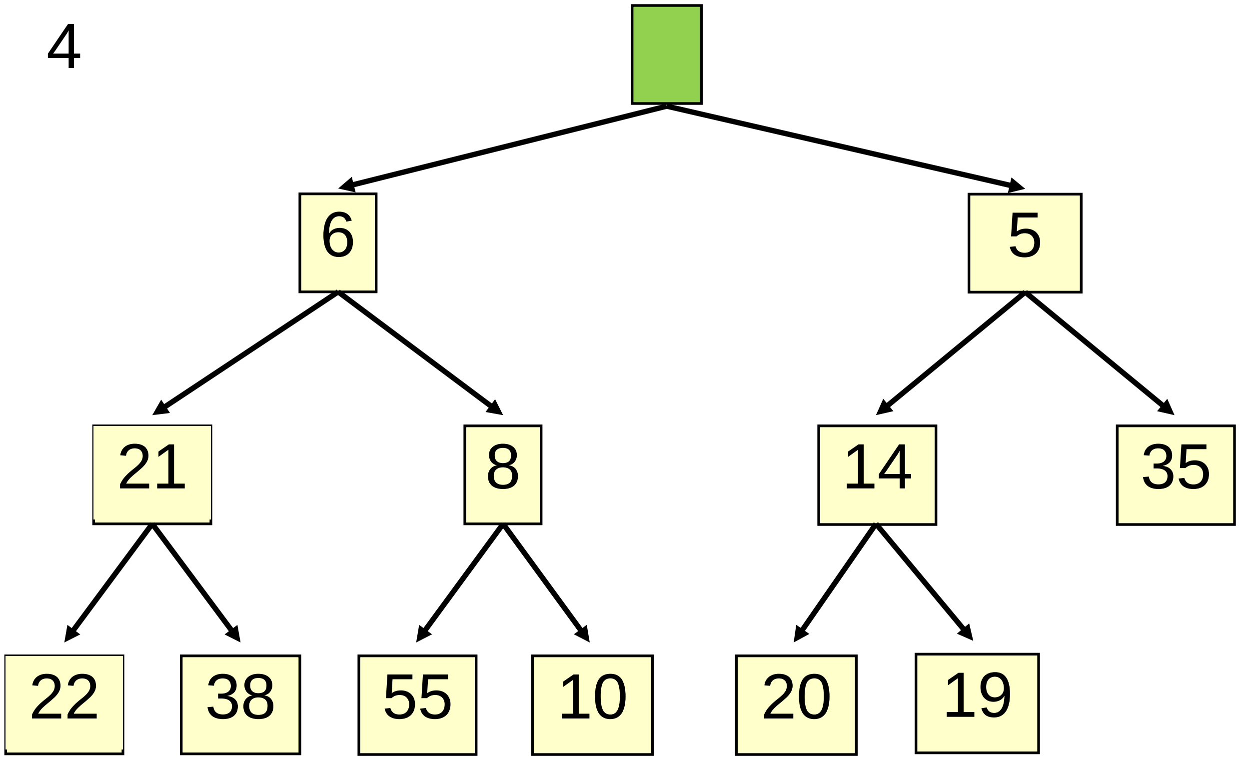
- Remove the least element – it is at the root
- This leaves a hole at the root – fill it in with the last element of the array
- If this violates heap order because the root element is too big, swap it down with the smaller of its children
- Continue swapping it down until it finds its rightful place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

`extract()`

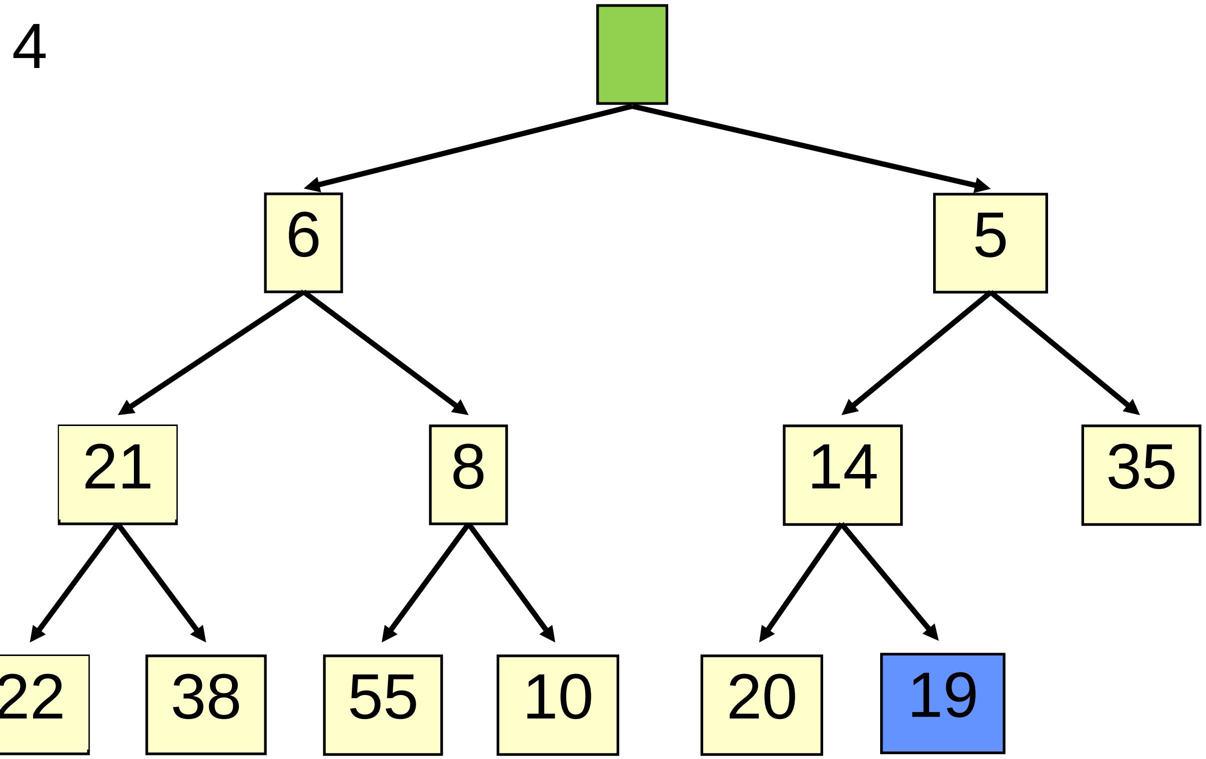


extract()

4

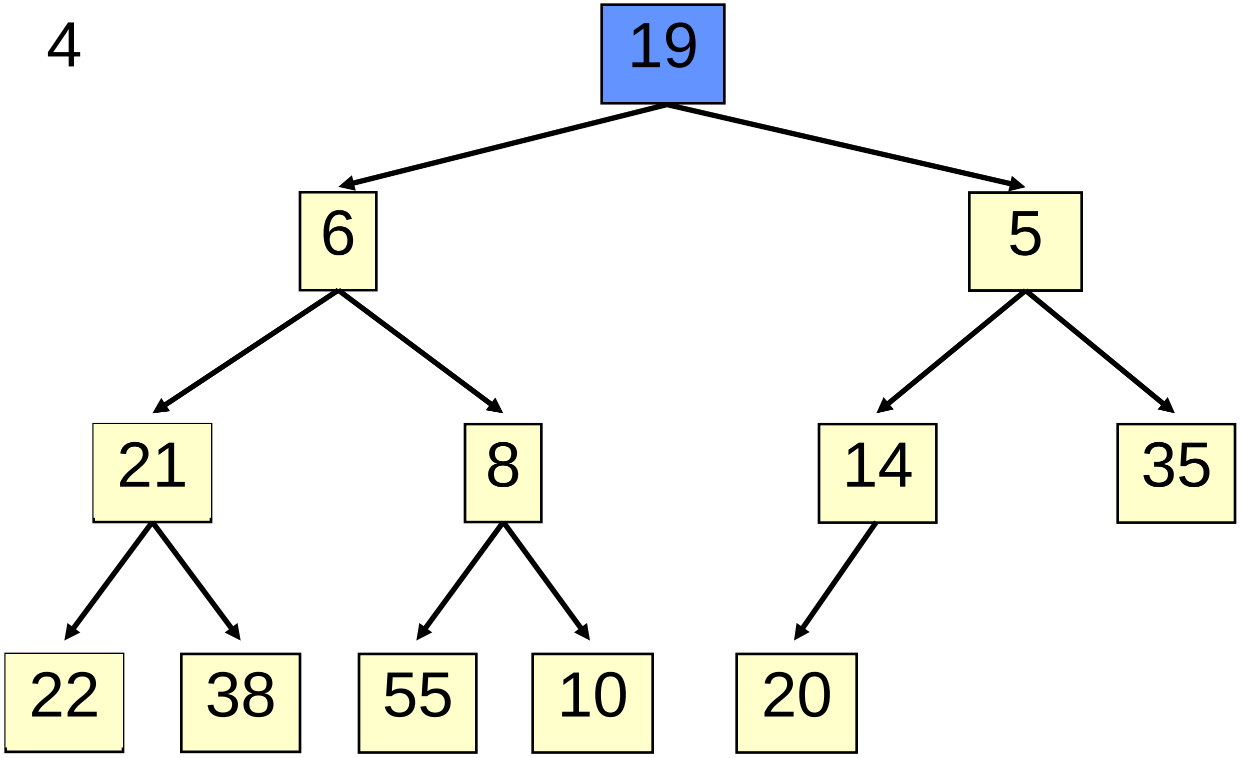


extract()



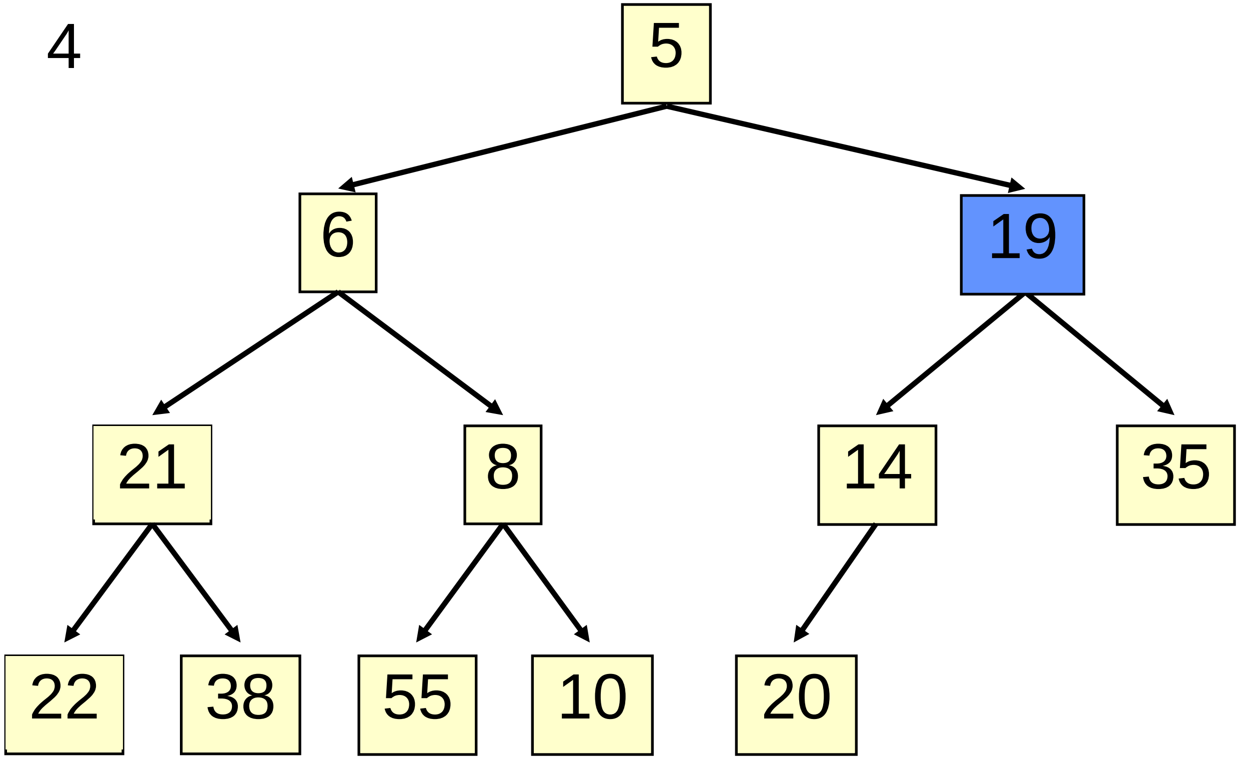
extract()

4

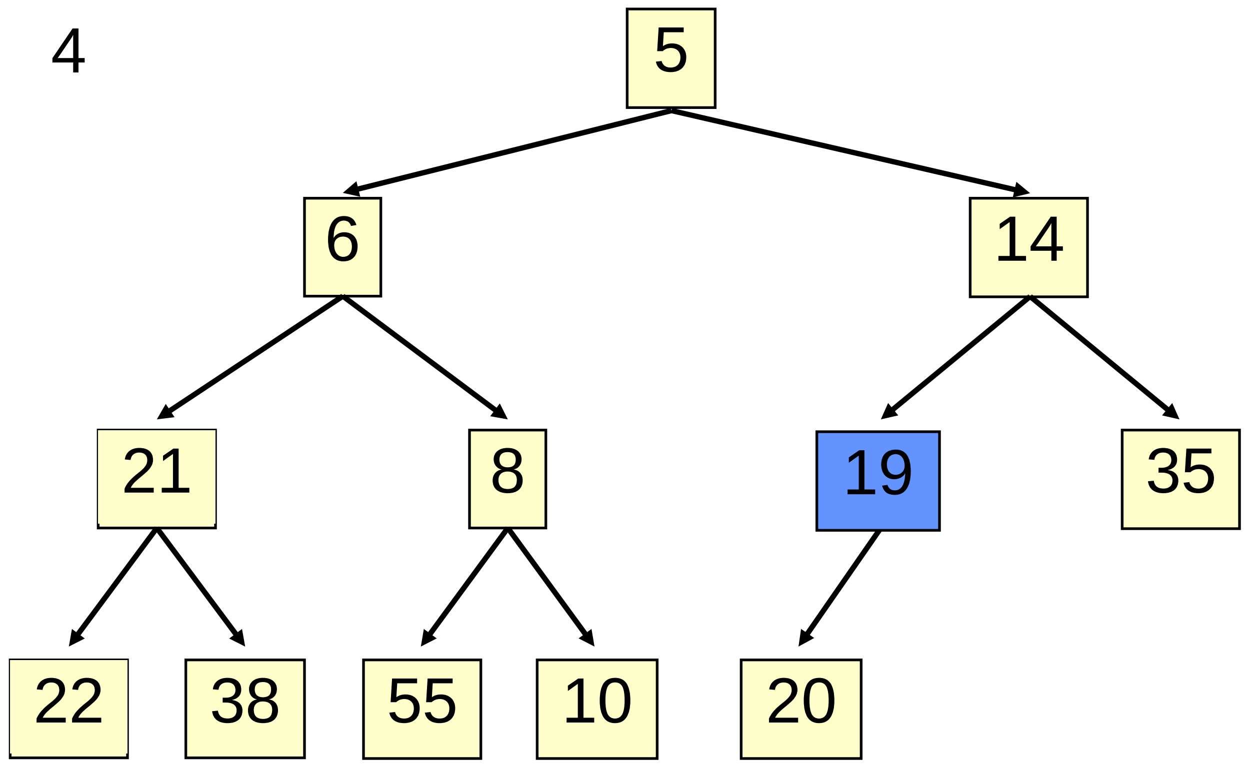


extract()

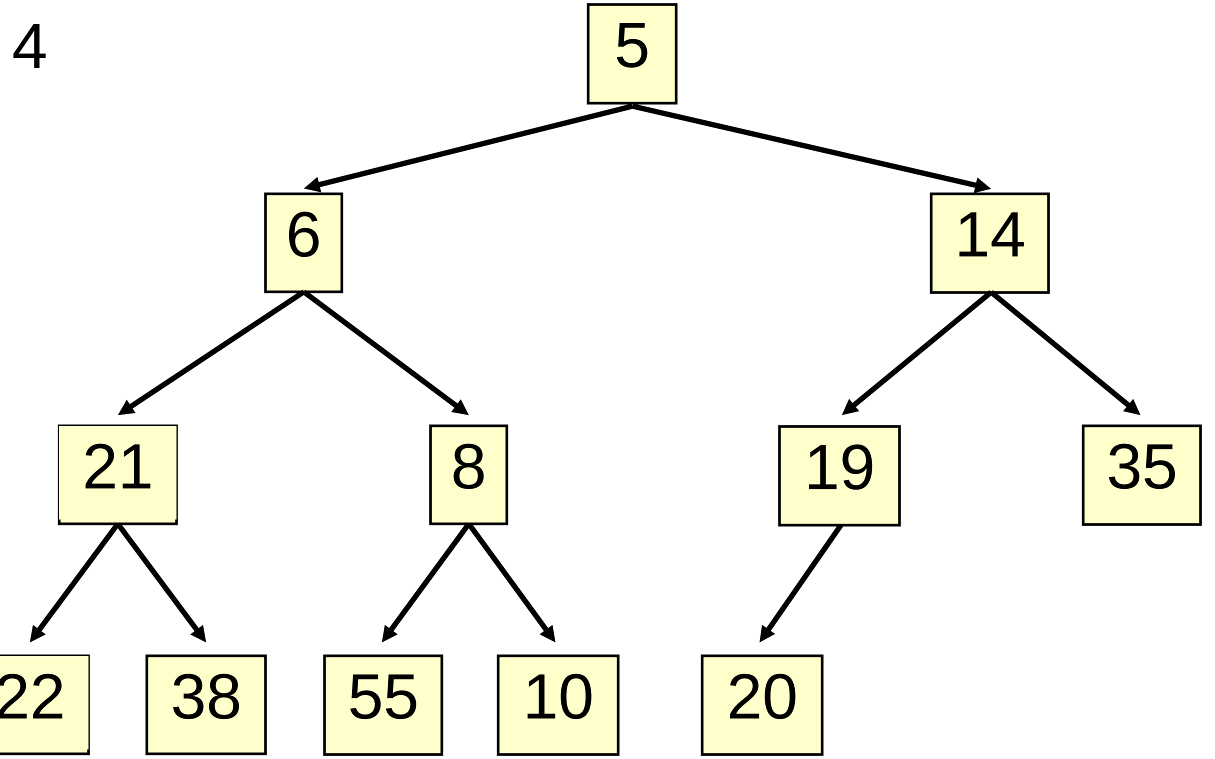
4



`extract()`

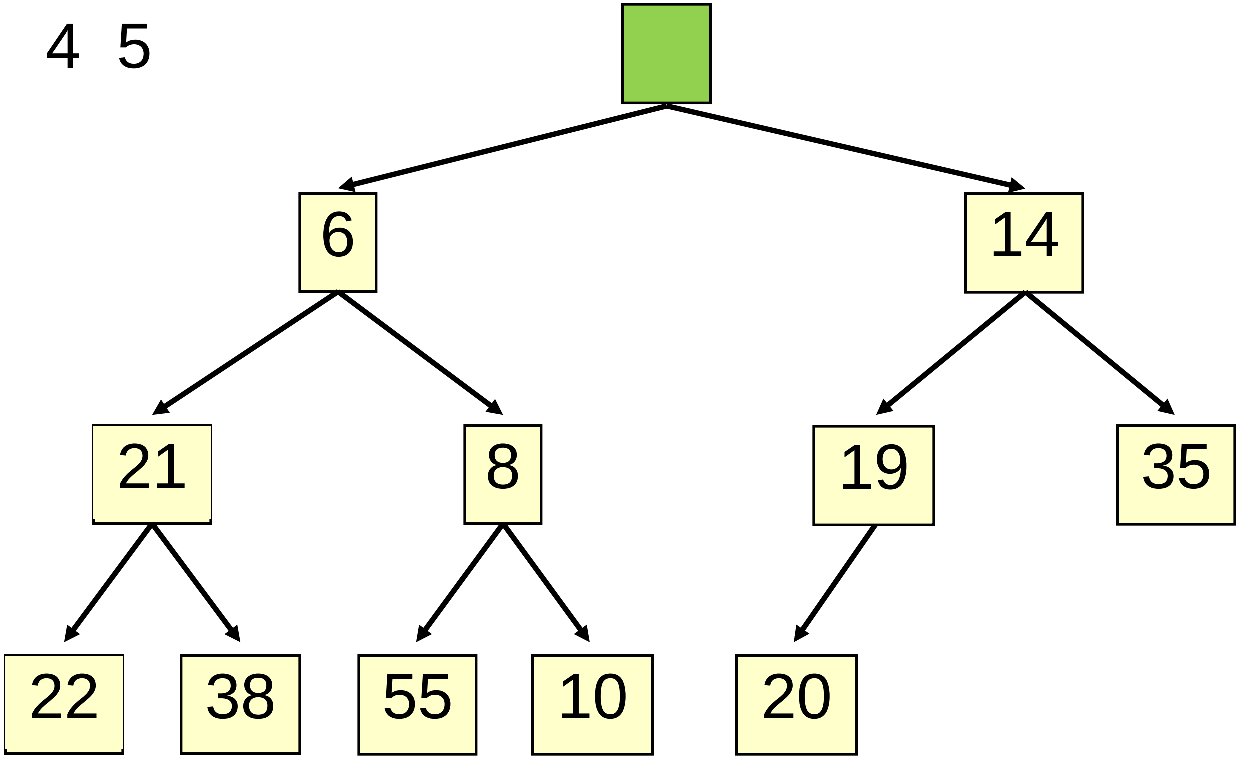


`extract()`



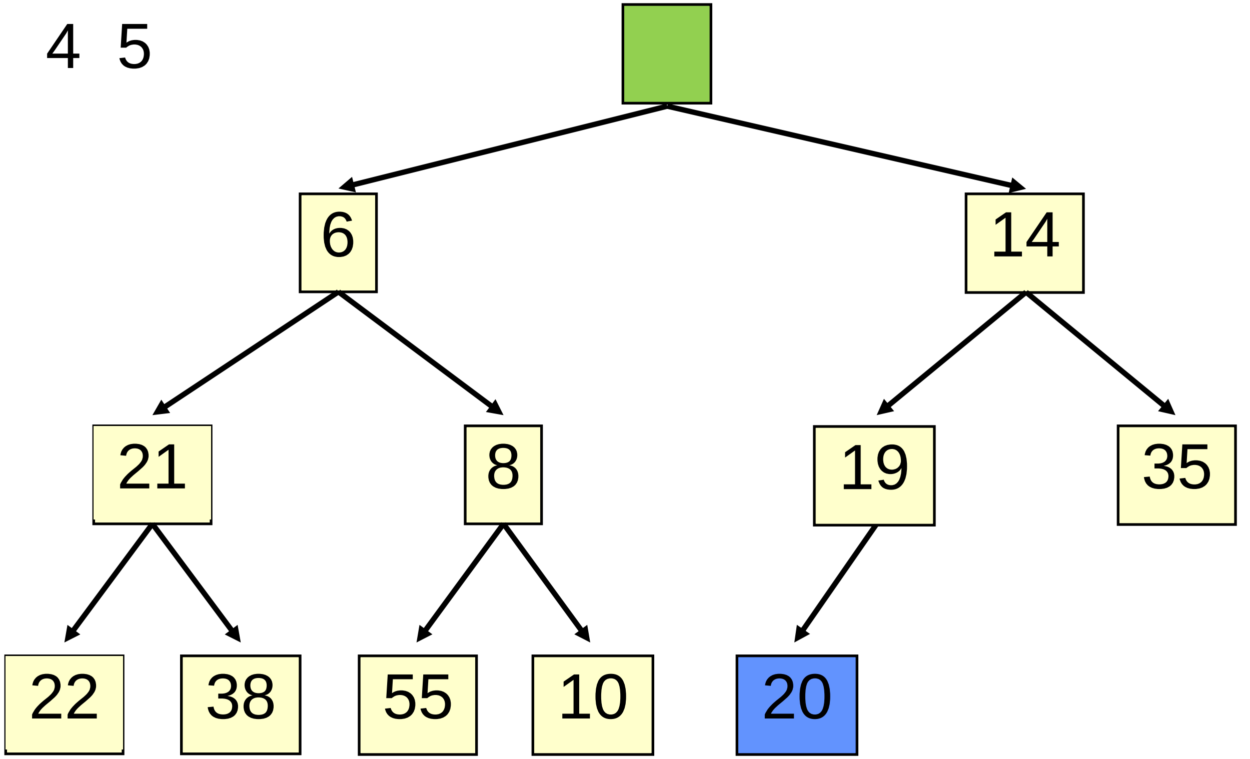
extract()

4 5



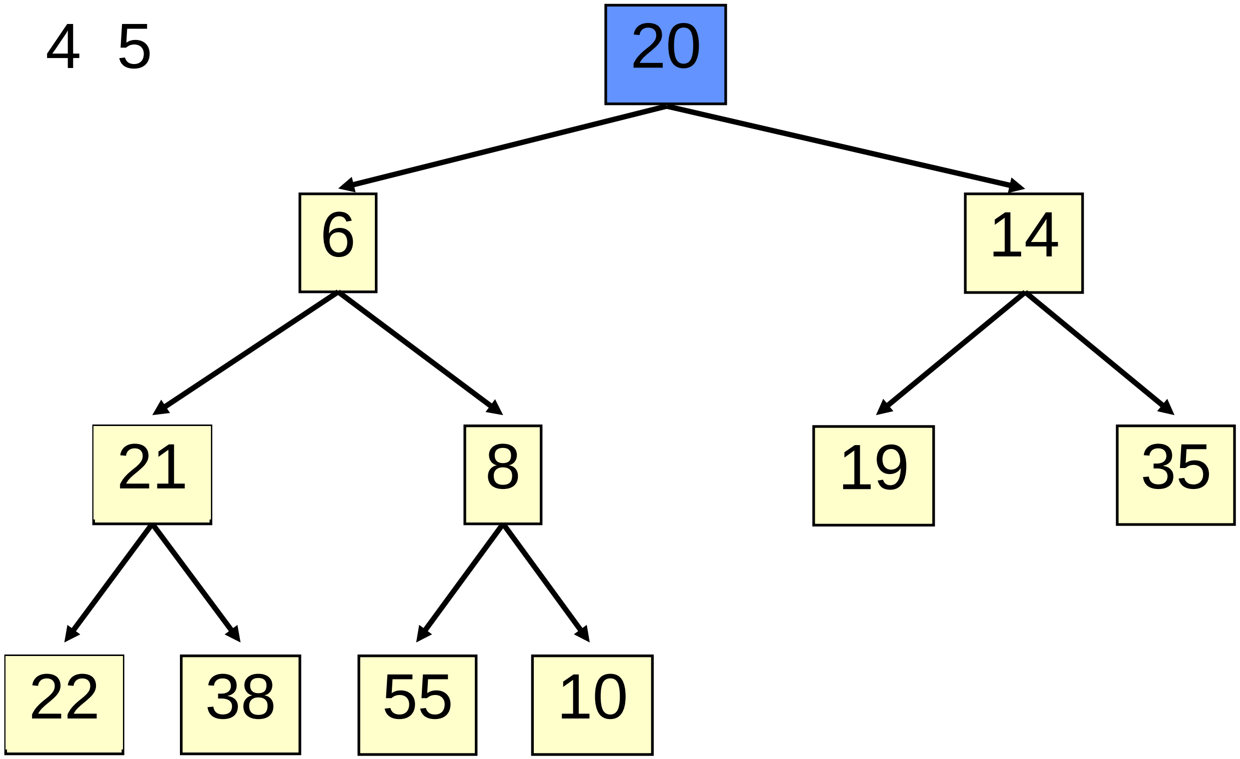
extract()

4 5



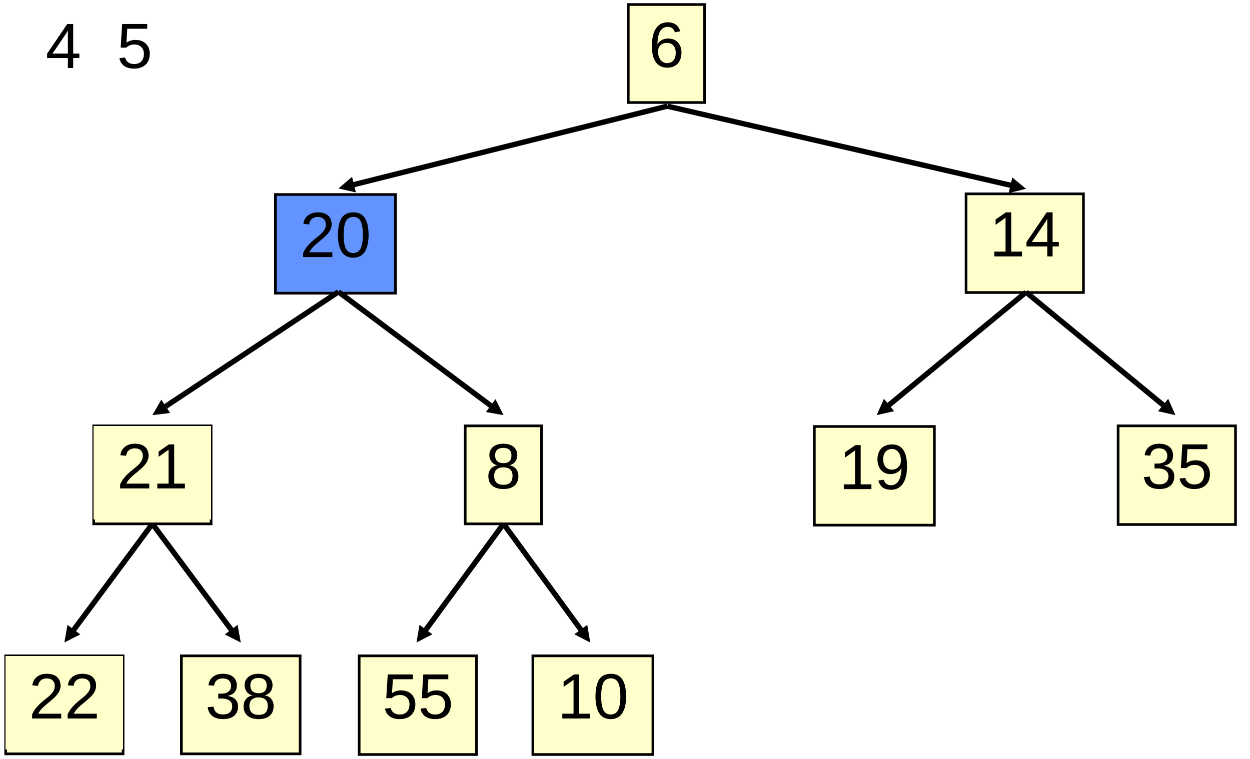
extract()

4 5



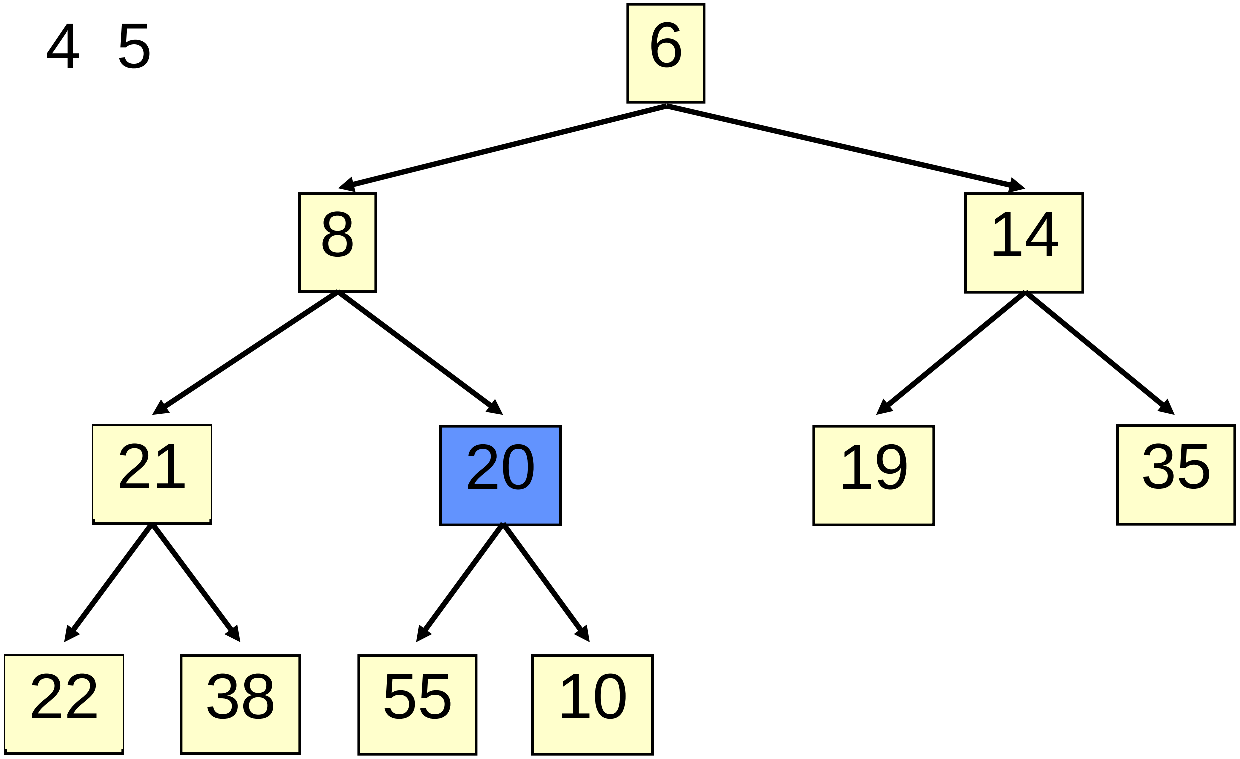
extract()

4 5



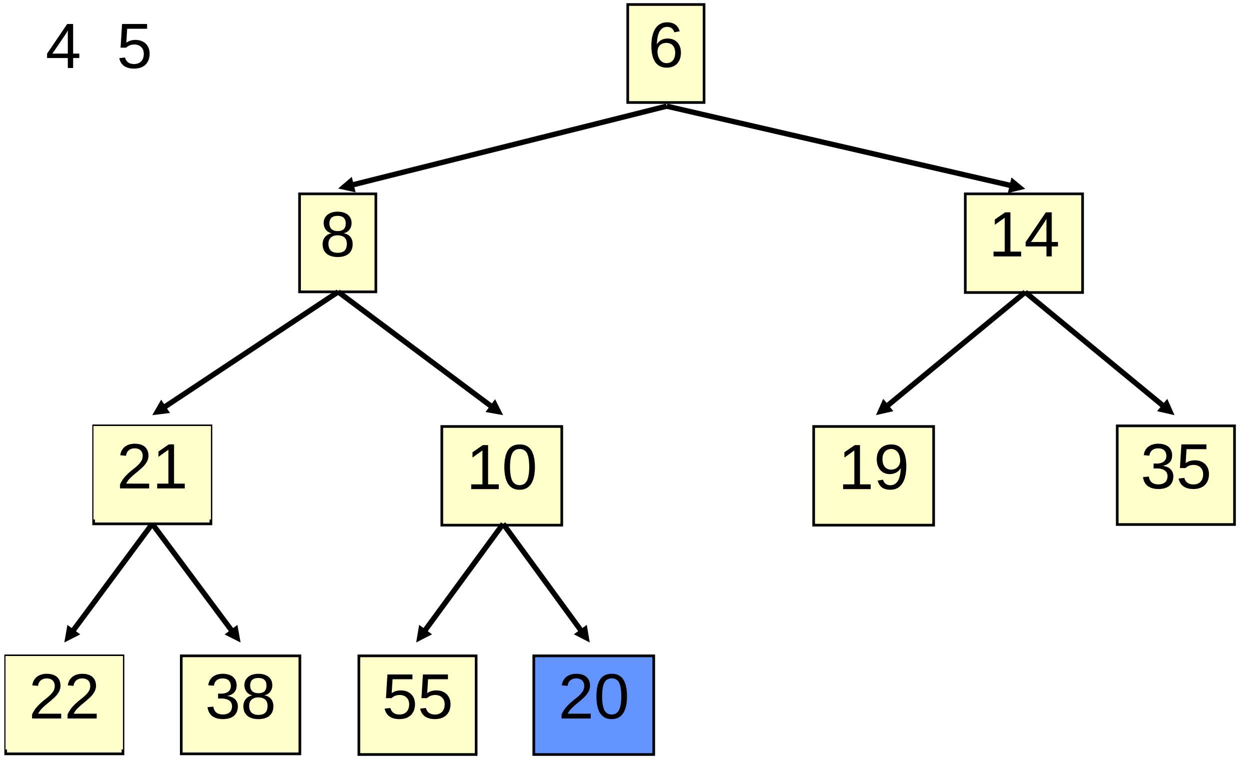
`extract()`

4 5



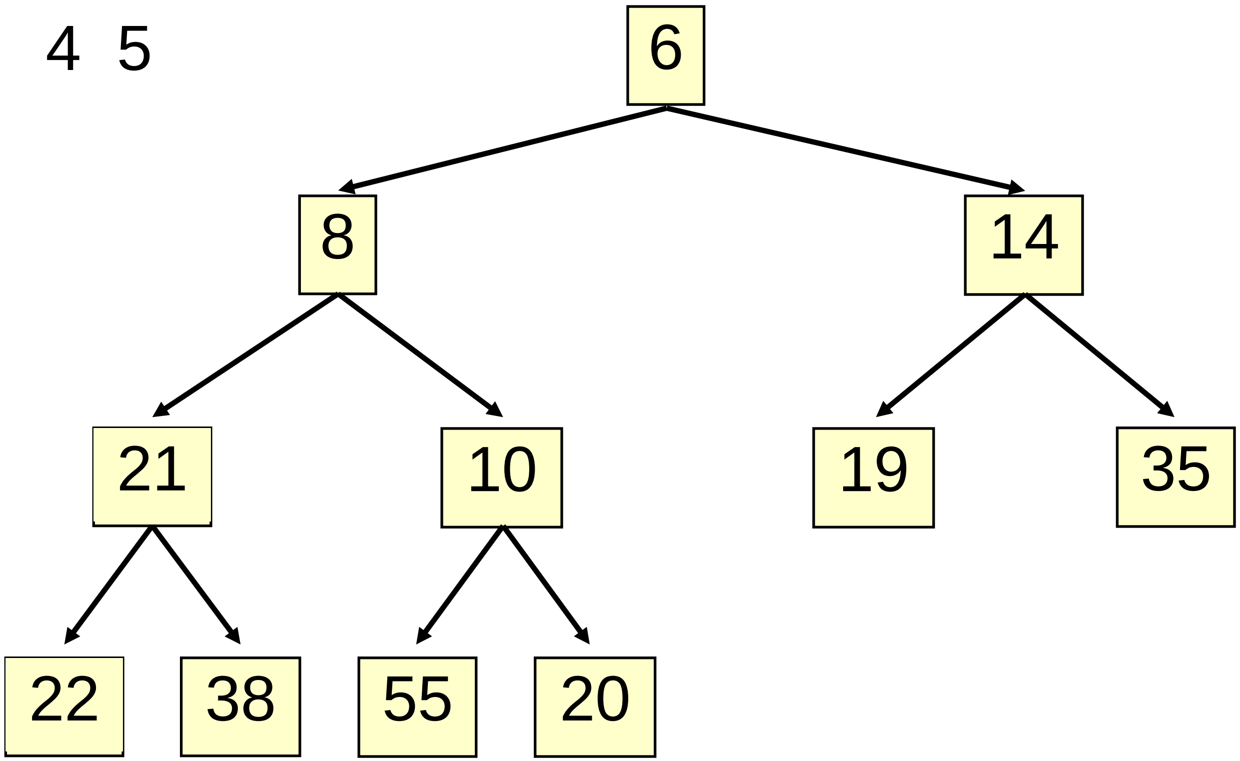
extract()

4 5



`extract()`

4 5



`extract()`

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

extract()


```
public E extract() {
  if (size() == 0) return null;
  E temp = elementAt(0);
  setElementAt(elementAt(size() - 1), 0);
  setSize(size() - 1);
  rotateDown(0);
  return temp;
}
private void rotateDown(int index) {
  int child = 2*(index + 1); //right child
  if (child >= size()
     || elementAt(child - 1).compareTo(elementAt(child)) < 0)
       child -= 1;
  if (child >= size()) return;
  if (elementAt(index).compareTo(elementAt(child)) <= 0)
     return;
  swap(index, child);
  rotateDown(child);
}
```

HeapSort

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

- 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!