# CS 211

# Computers and Programming

http://www.cs.cornell.edu/courses/cs211/2005su

## Lecture 15: Priority Queues and Heaps

# Announcements

- Assignment 4 Programming is up
- Assignment 4 Written coming soon
- Both due Tuesday at 10am
- Quiz Tomorrow on Complexity, Iterators, and ADTs
- Discussion of Java 5 Generics tomorrow
- Reading for today: Weiss 21.1 - 21.5

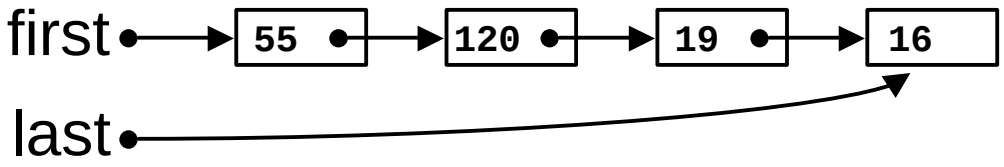
# The Bag Interface

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

Examples: Stack, Queue

# Stacks and Queues as Lists

- Stack (LIFO) implemented as list
  - **`put()`**, **`get()`** from front of list
- Queue (FIFO) implemented as list
  - **`put()`** on back of list, **`get()`** 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
- **`get()`** returns the element with the highest priority = least in the **`compareTo()`** ordering
- break ties arbitrarily

# 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

# Priority Queues

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

interface PriorityQueue<E extends Comparable>
   extends Bag<E> {}
```

# Priority Queues as Lists

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

- Maintain as ordered list
  - **`put()`** must search the list – O(n)
  - **`get()`** 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

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

# Heaps

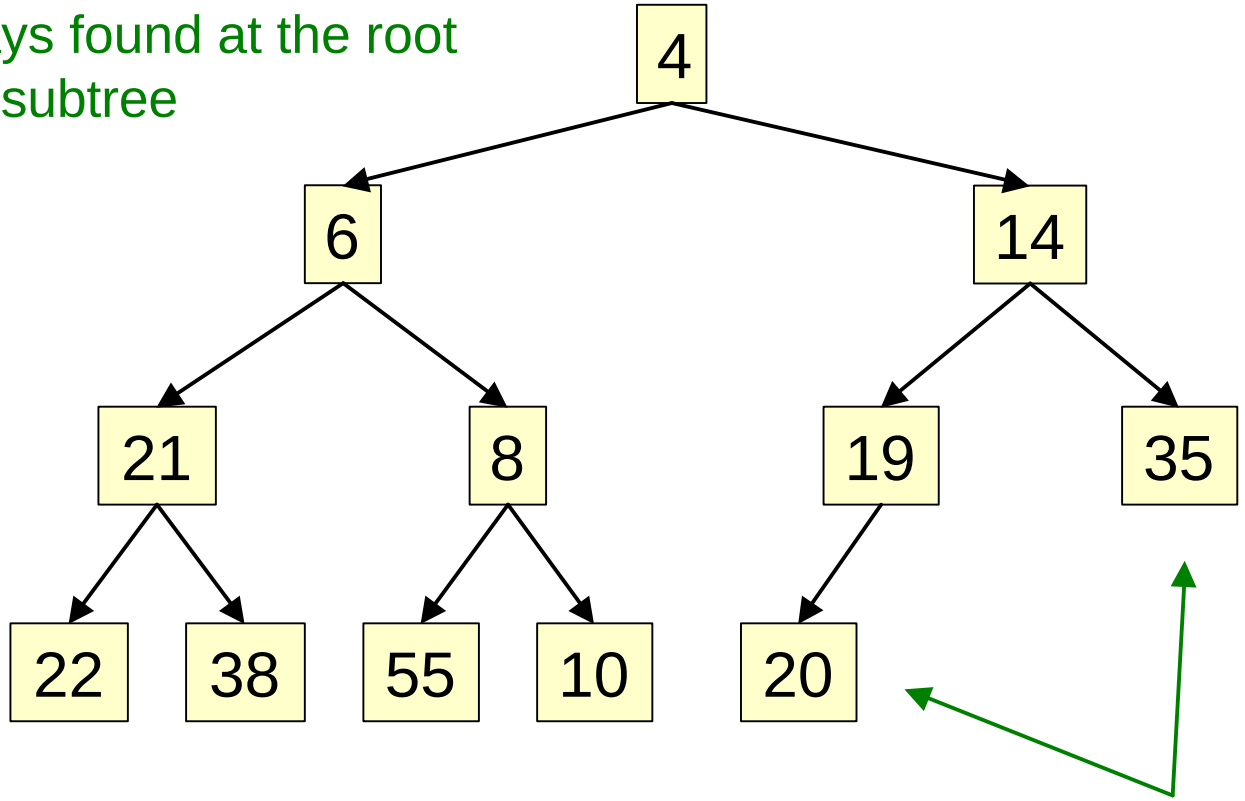
- A *heap* is a concrete data structure that can be used to implement priority queues
- Gives better complexity than either ordered or unordered list implementation:
  - **`put()`**, **`get()`** – O(log n)
  - **`isEmpty()`** – O(1)
- O(n log n) to process n elements
- Do not confuse with *heap memory*, where the Java virtual machine allocates space for objects – different usage!

# 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

least element of any subtree
is always found at the root
of that subtree




but it is possible to have
smaller elements deeper
in the tree!

# Examples of Heaps

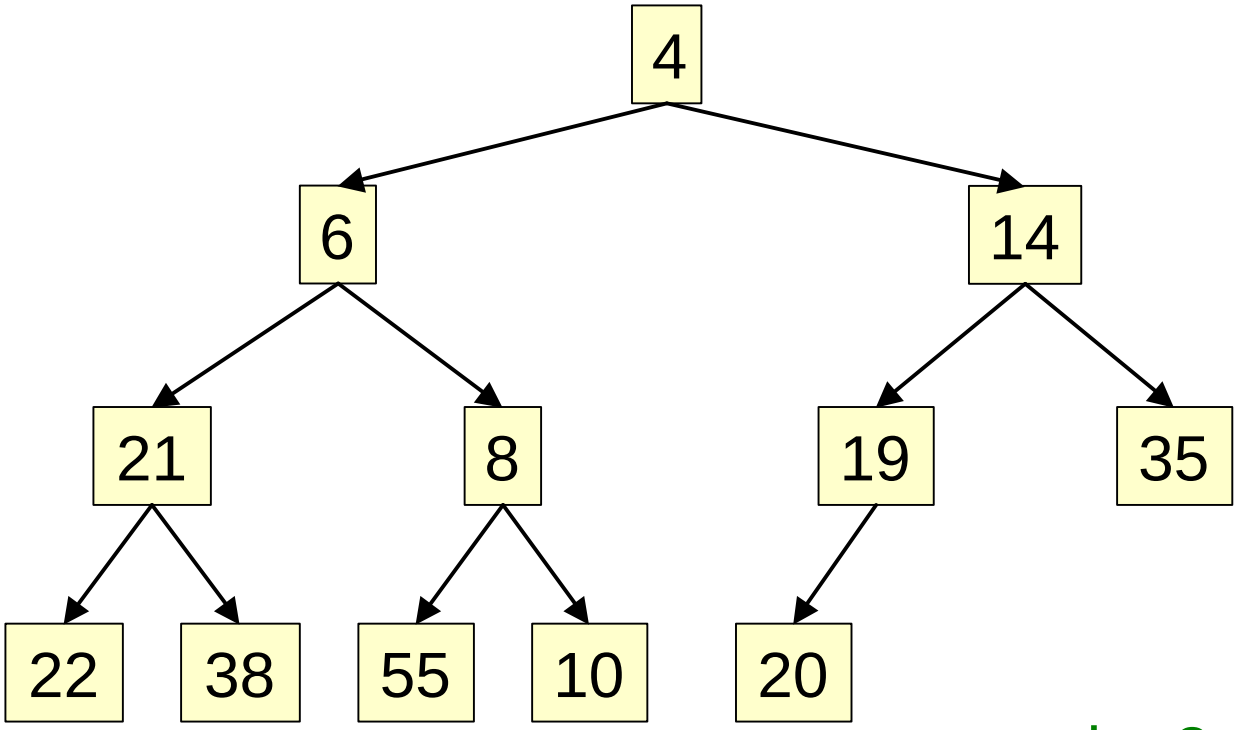
- Ages of people in family tree
  - parent is always older than children, but you can have an uncle who is younger than you
- Salaries of employees of a company
  - bosses generally make more than subordinates, but a VP in one subdivision may make less than a Project Supervisor in a different subdivision

# Balanced Heaps

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
2. All maximal paths of length d are to the left of those of length d – 1

# A Balanced Heap

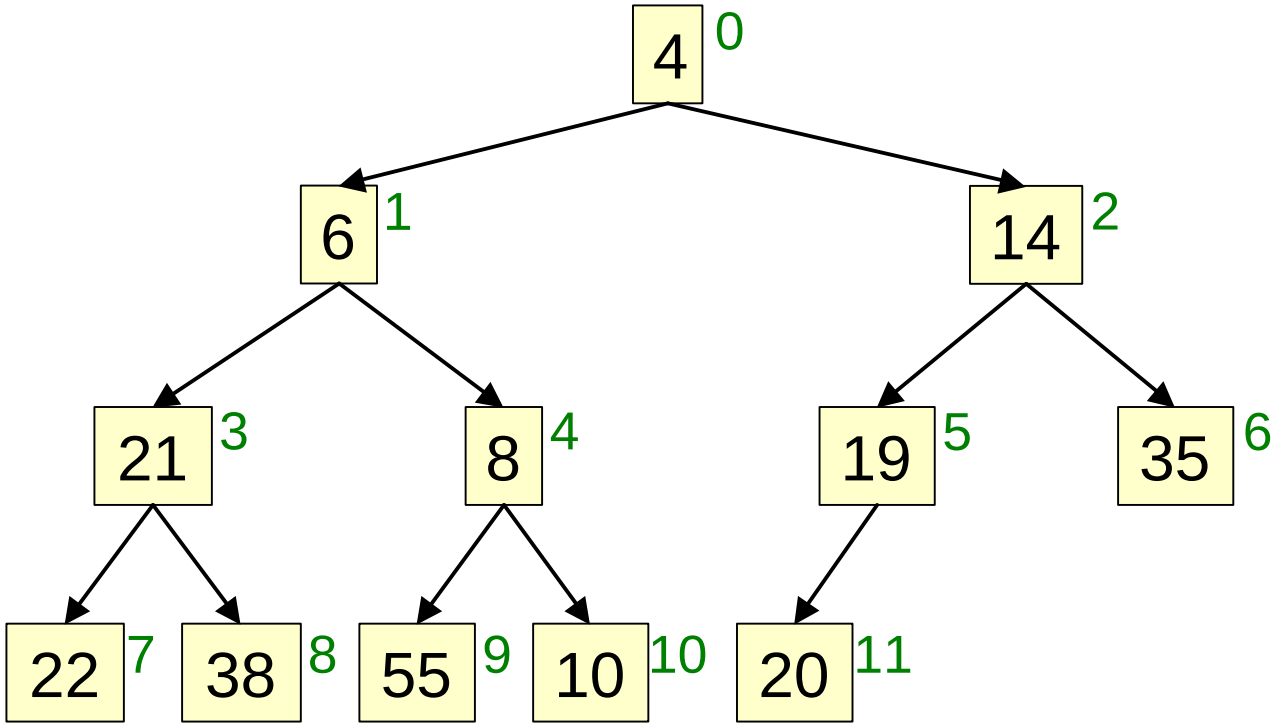



d = 3

# Store in an Array 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 Array or Vector

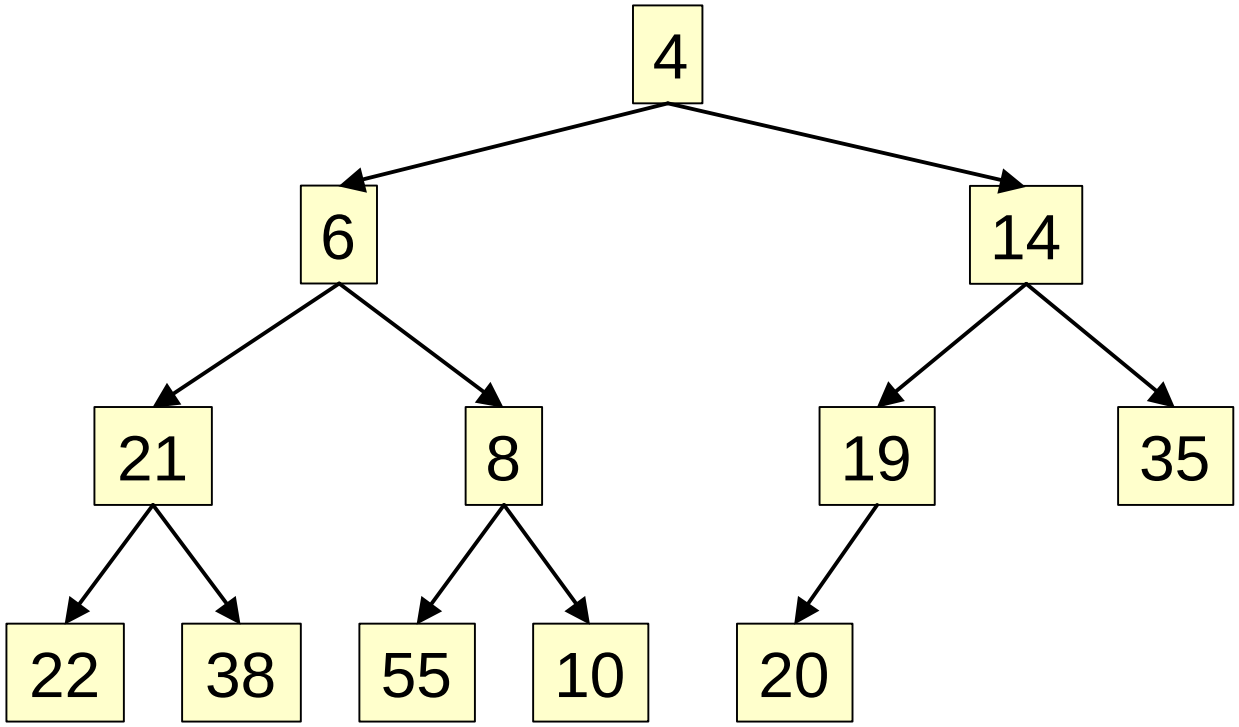



children of node n are found at 2n + 1 and 2n + 2

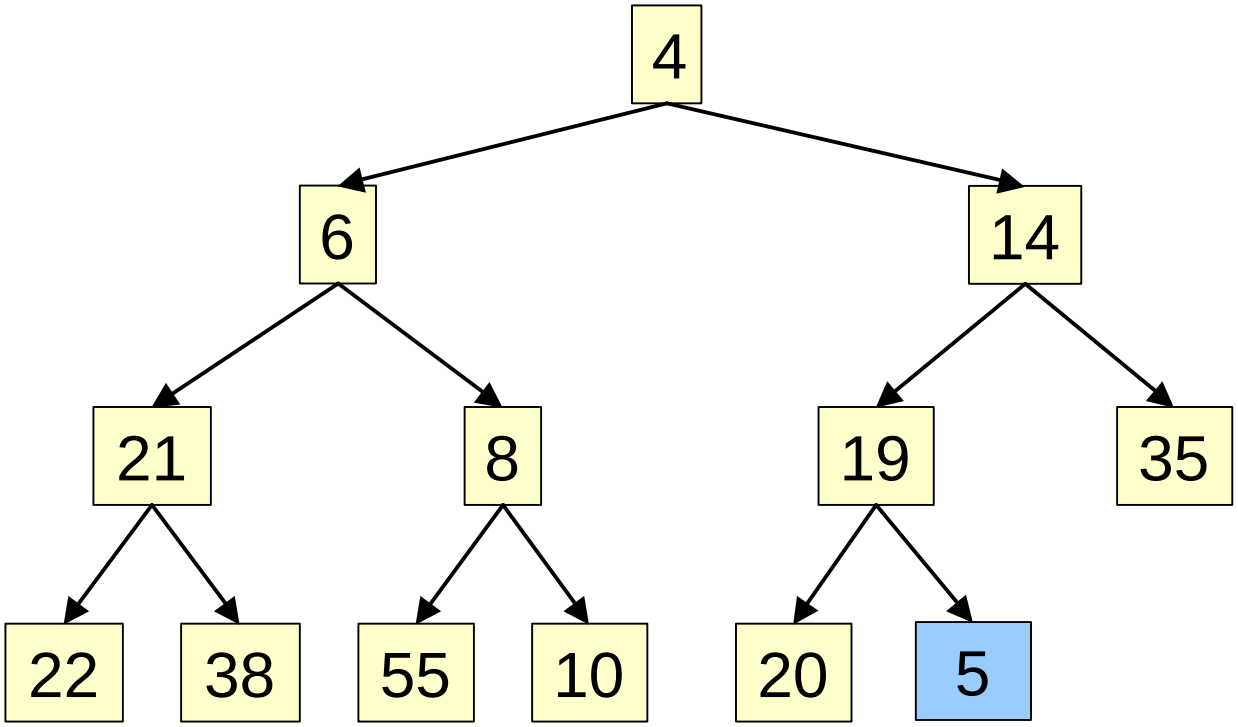
# `put()`

- 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!

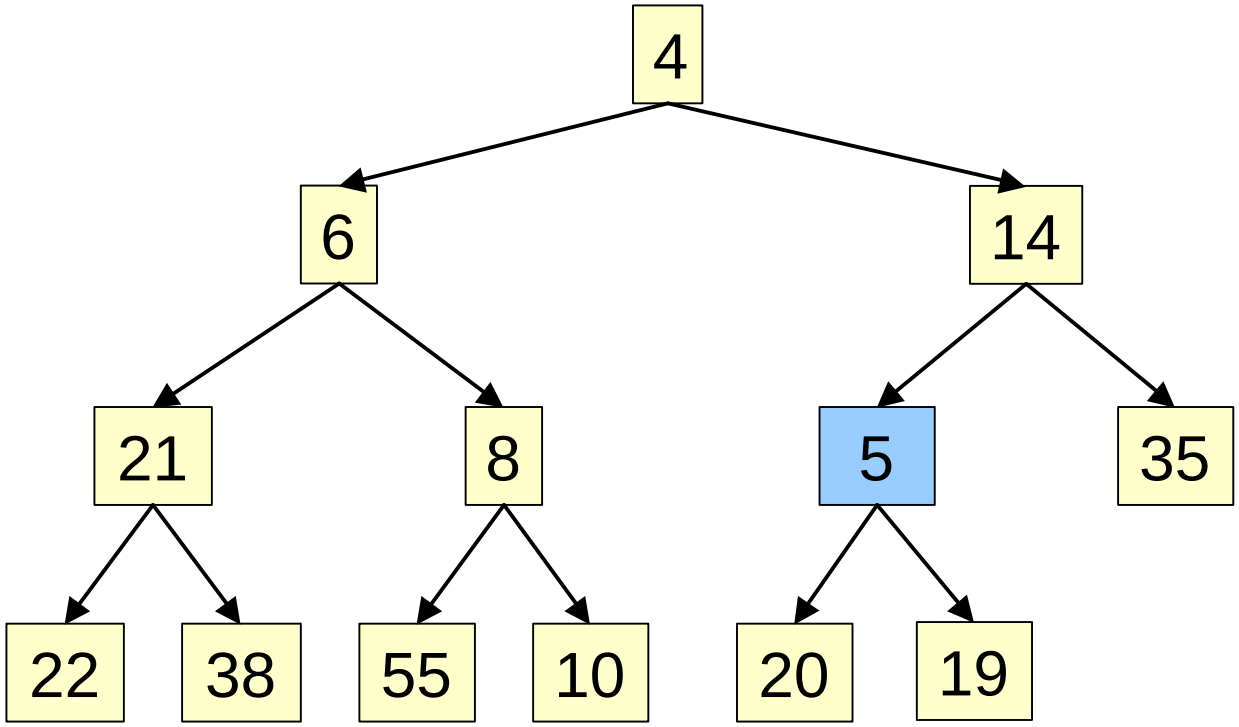
# put()



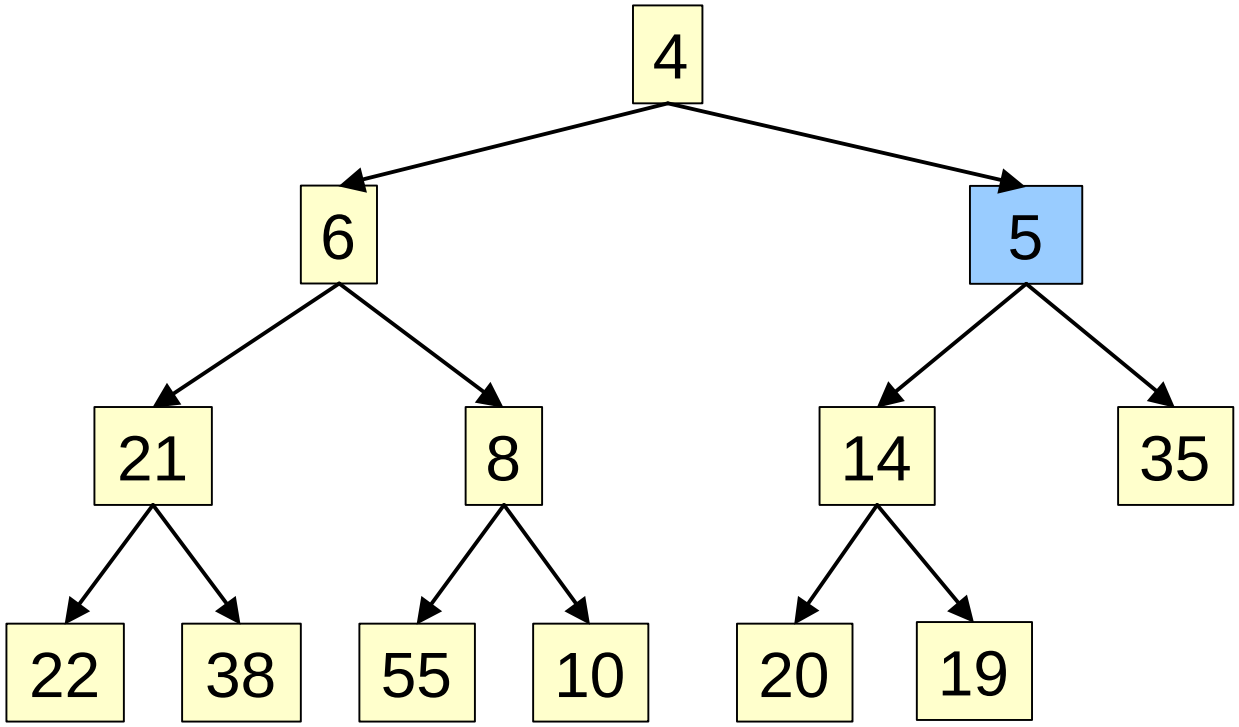
# put()



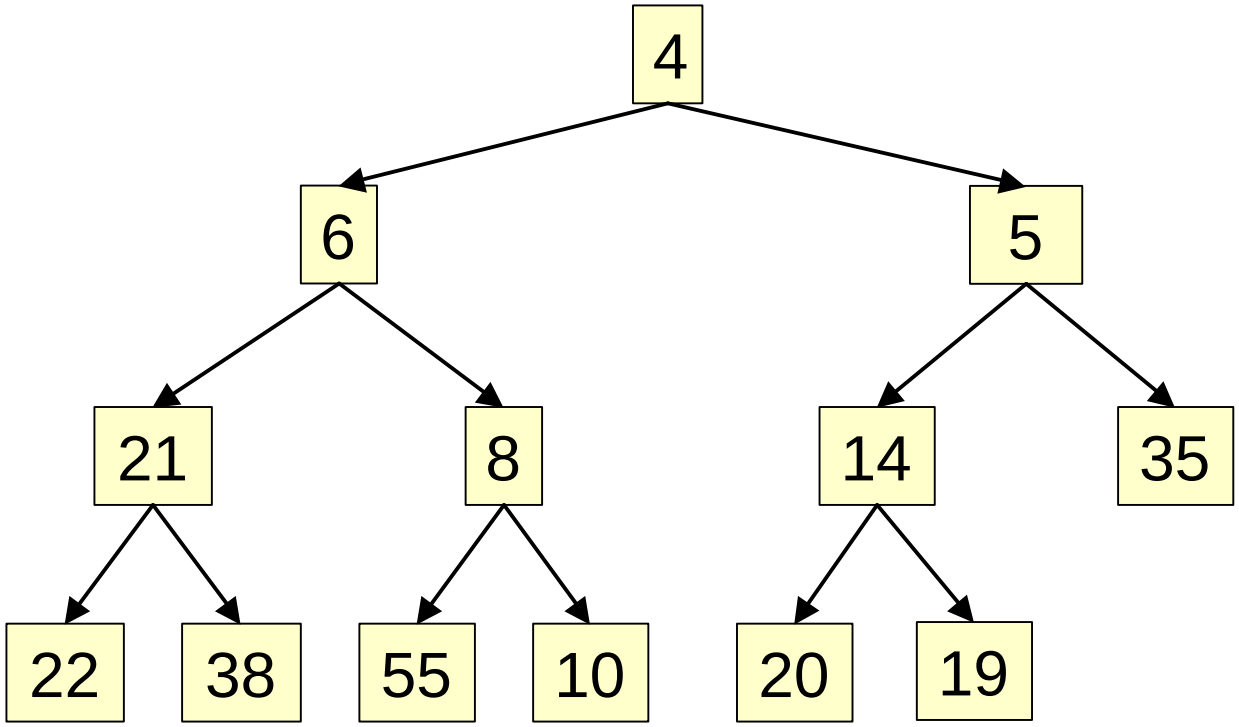
# put()



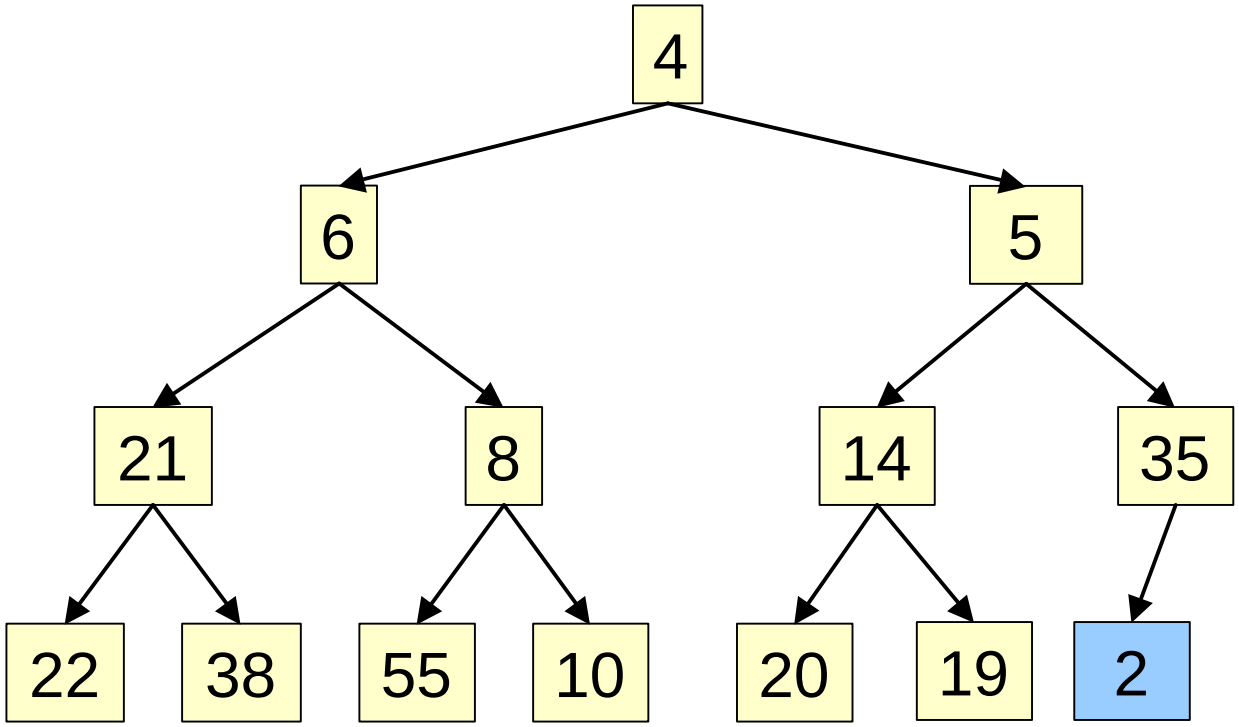
# put()



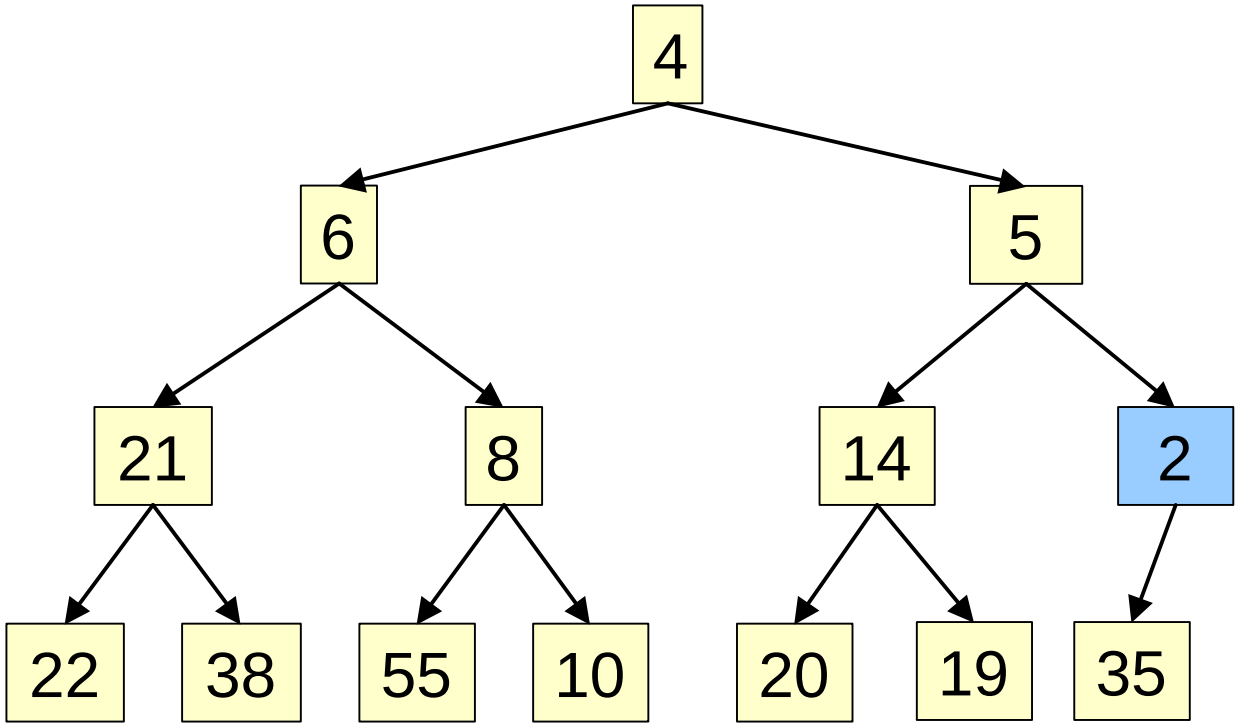
# put()



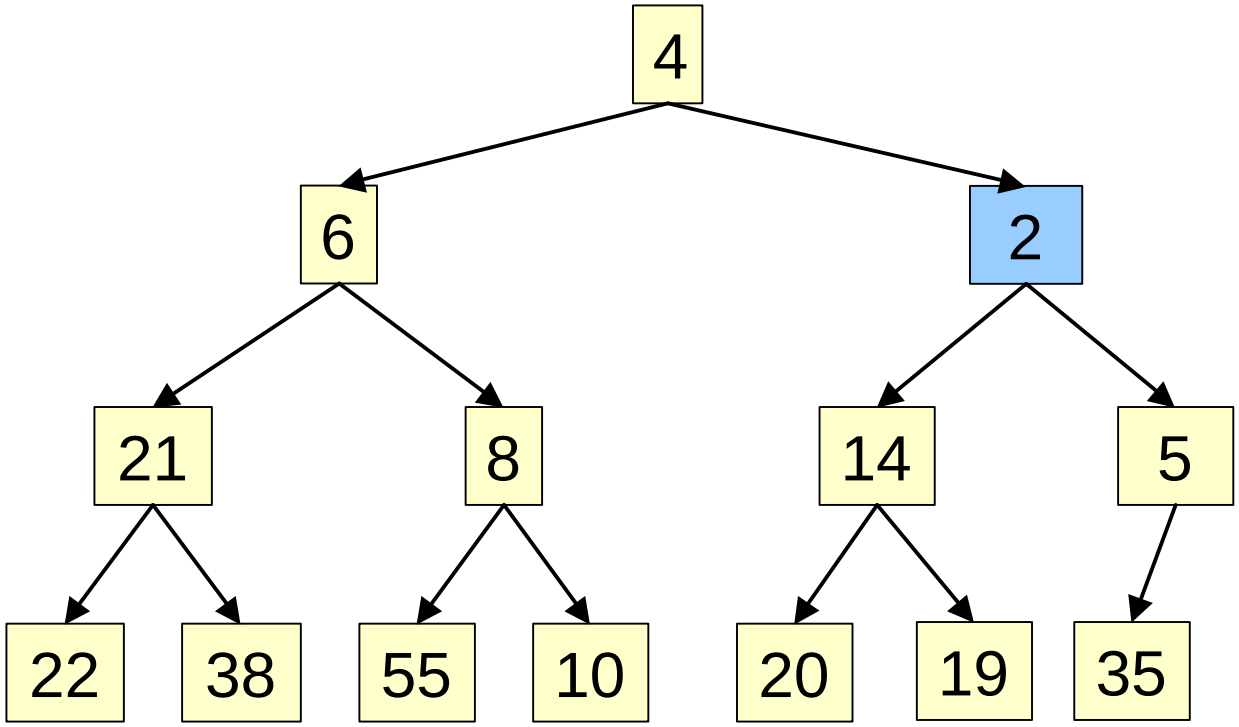
# put()



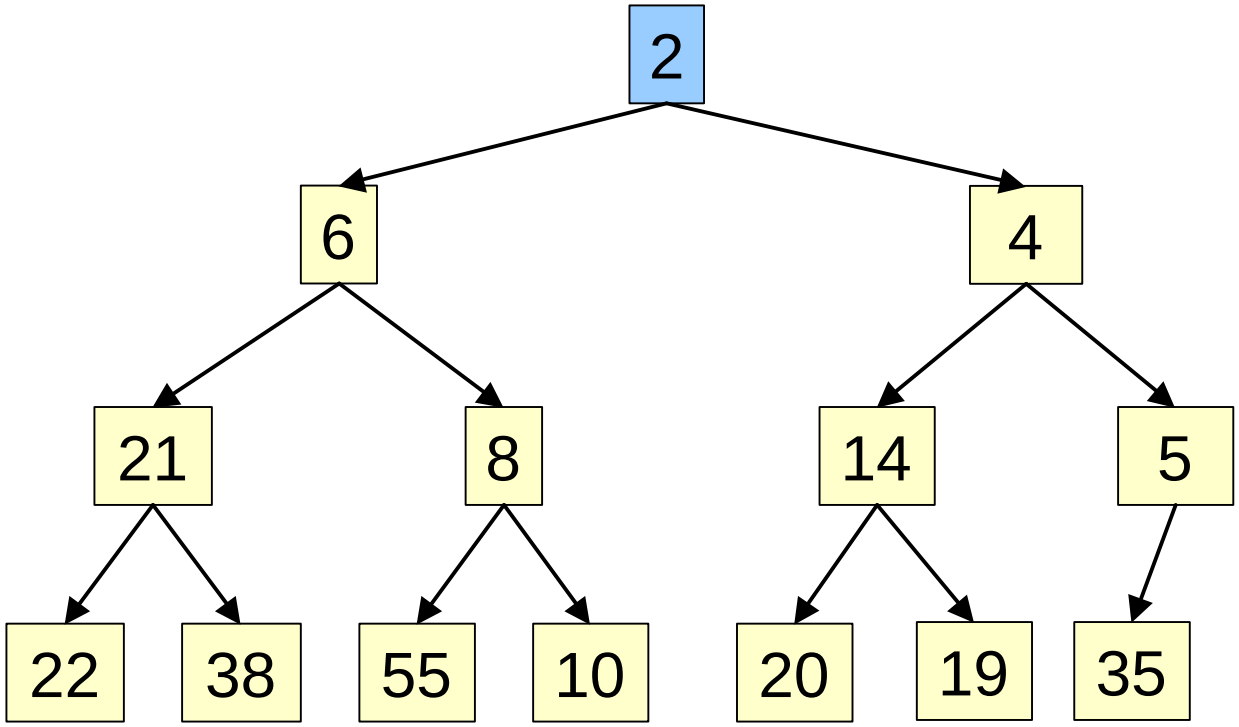
# put()



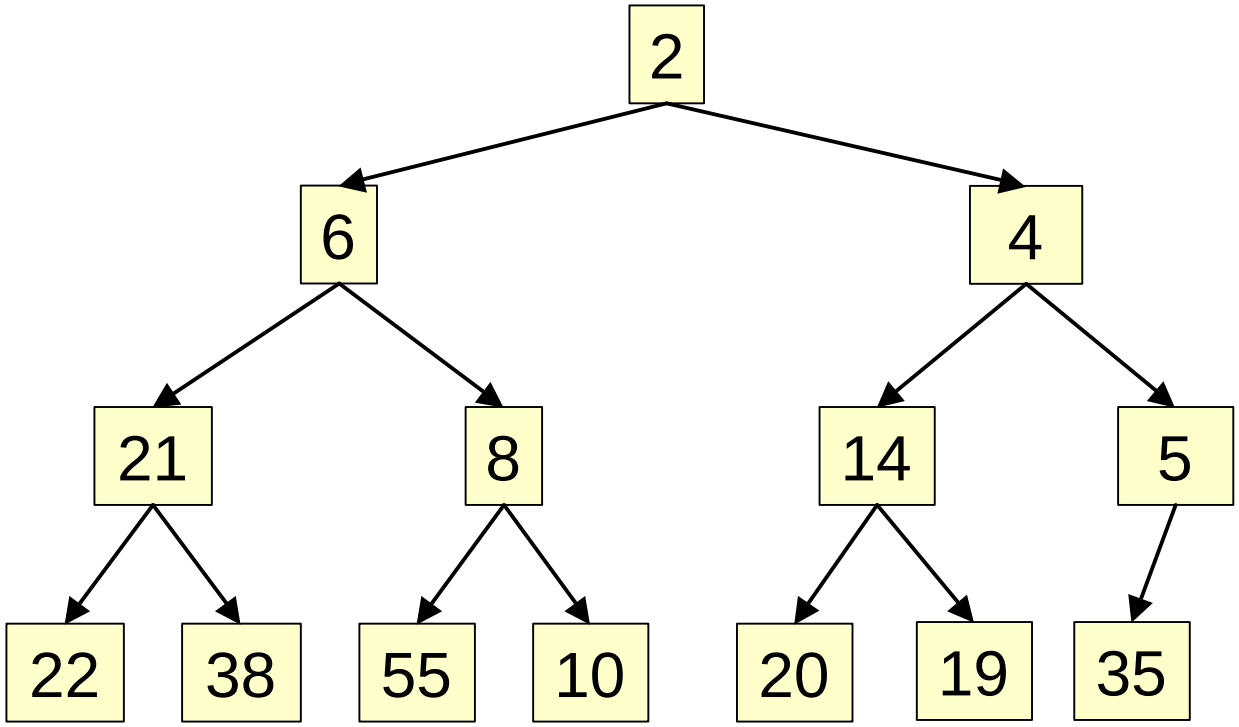
# put()



# put()



# put()



# `put()`

- 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

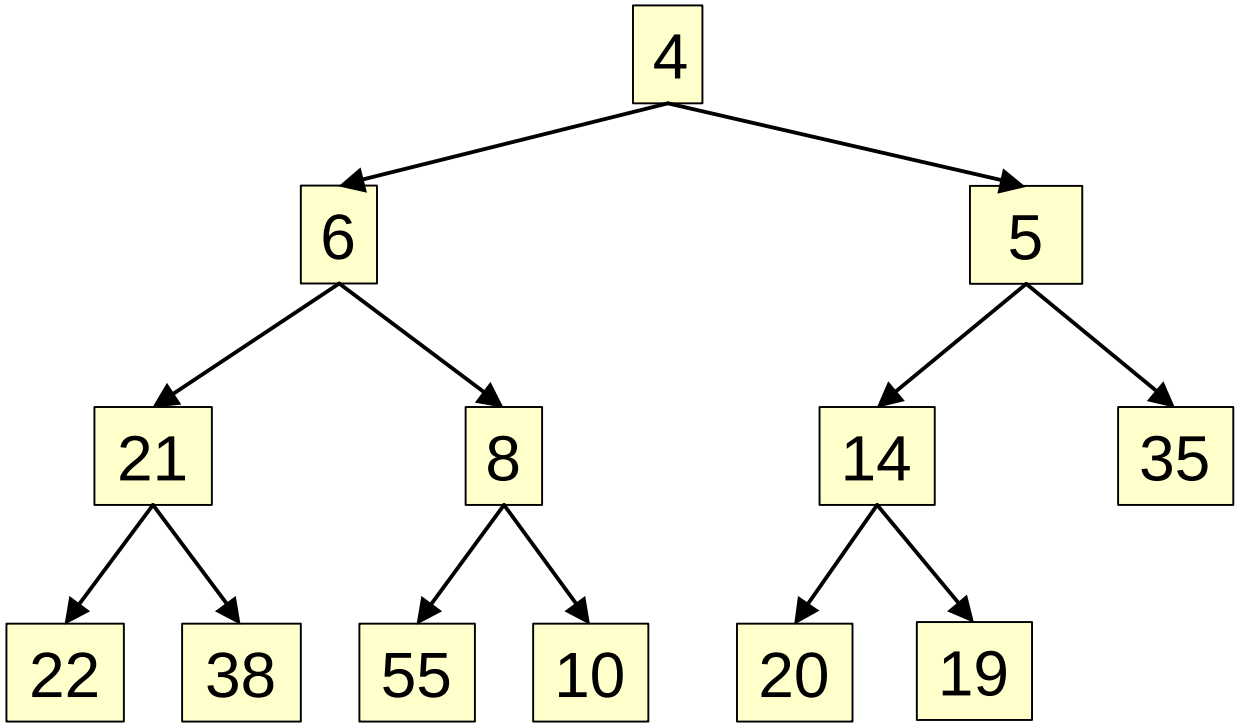
# put()

```
class PQ<E extends Comparable> extends java.util.Vector<E>
  implements PriorityQueue<E> {

  public void put(E obj) {
    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);
  }
```

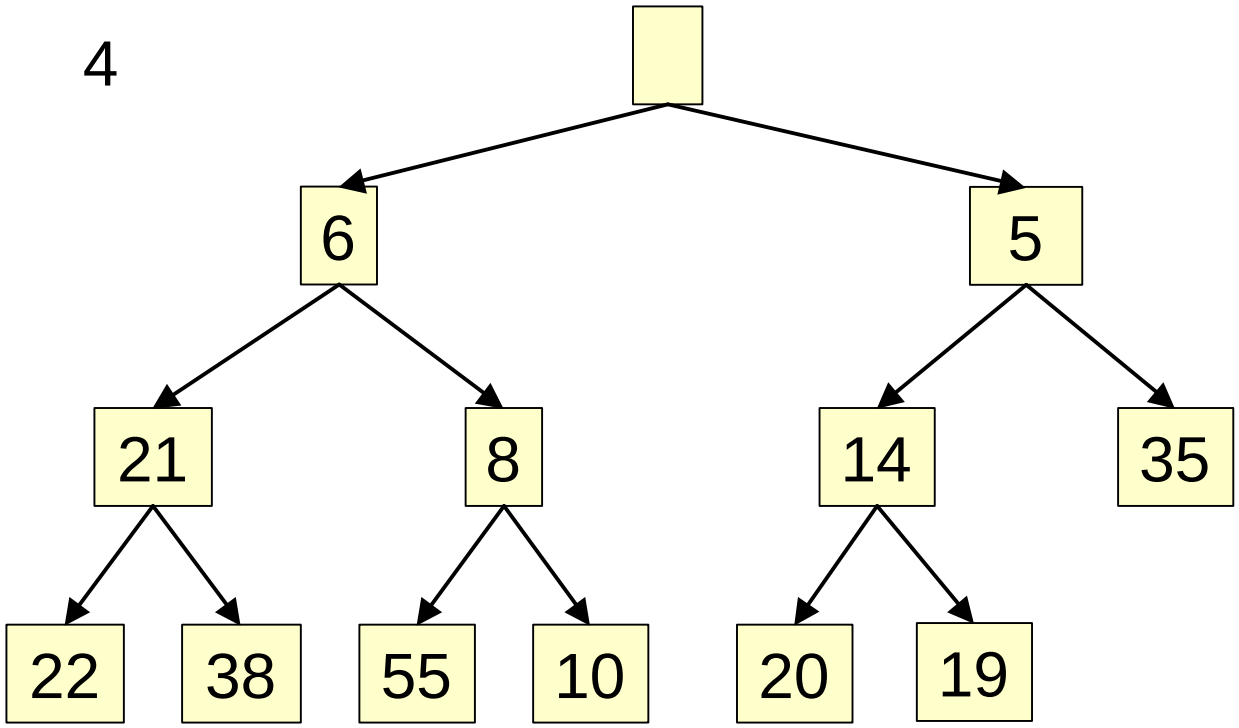
# `get()`

- 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!

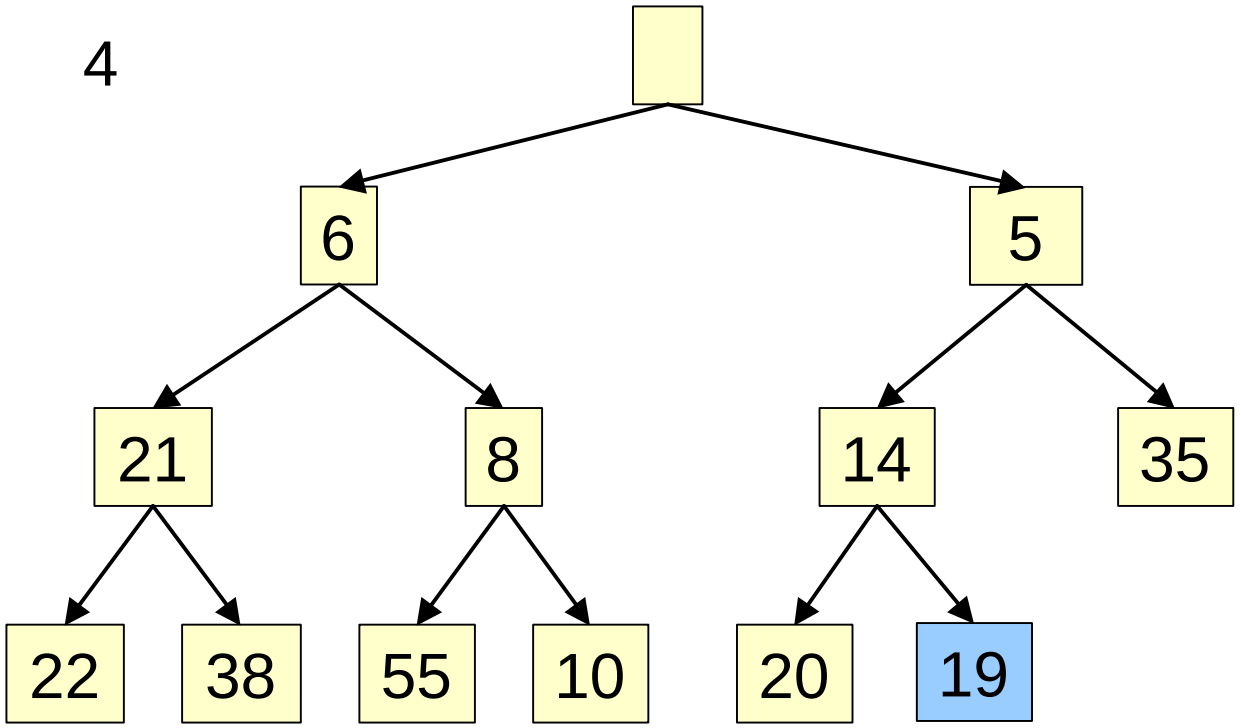
# get()



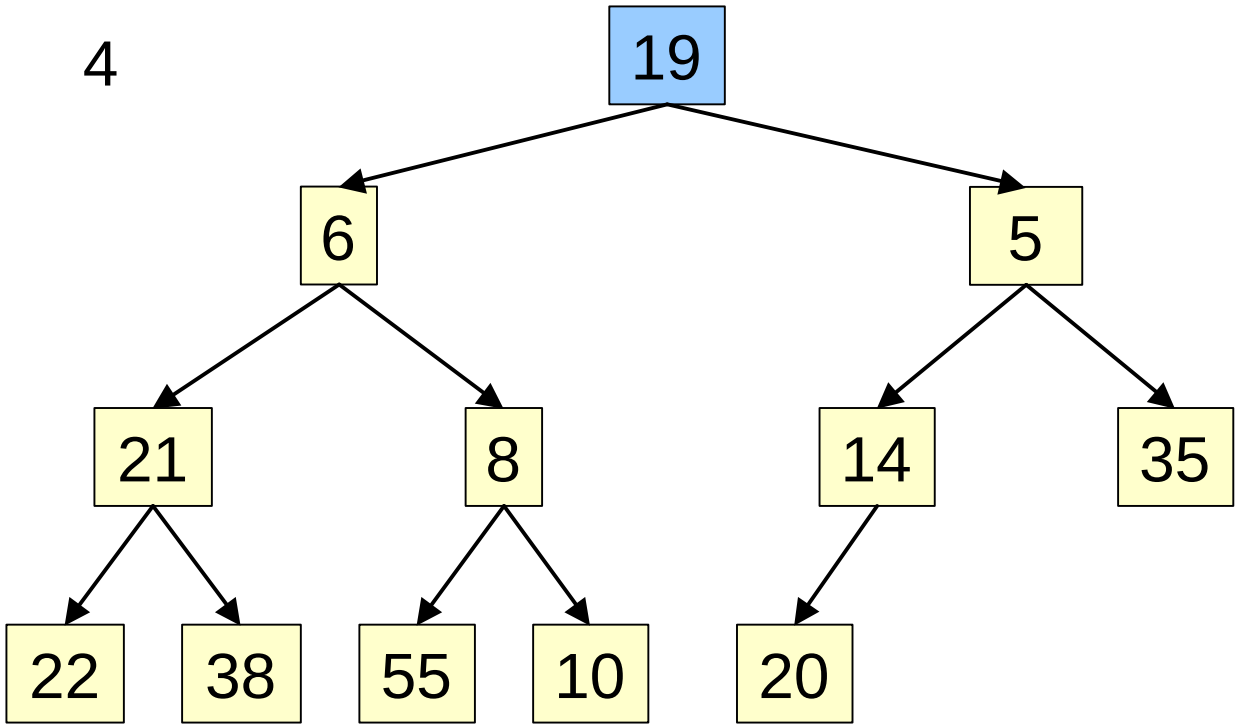
# get()



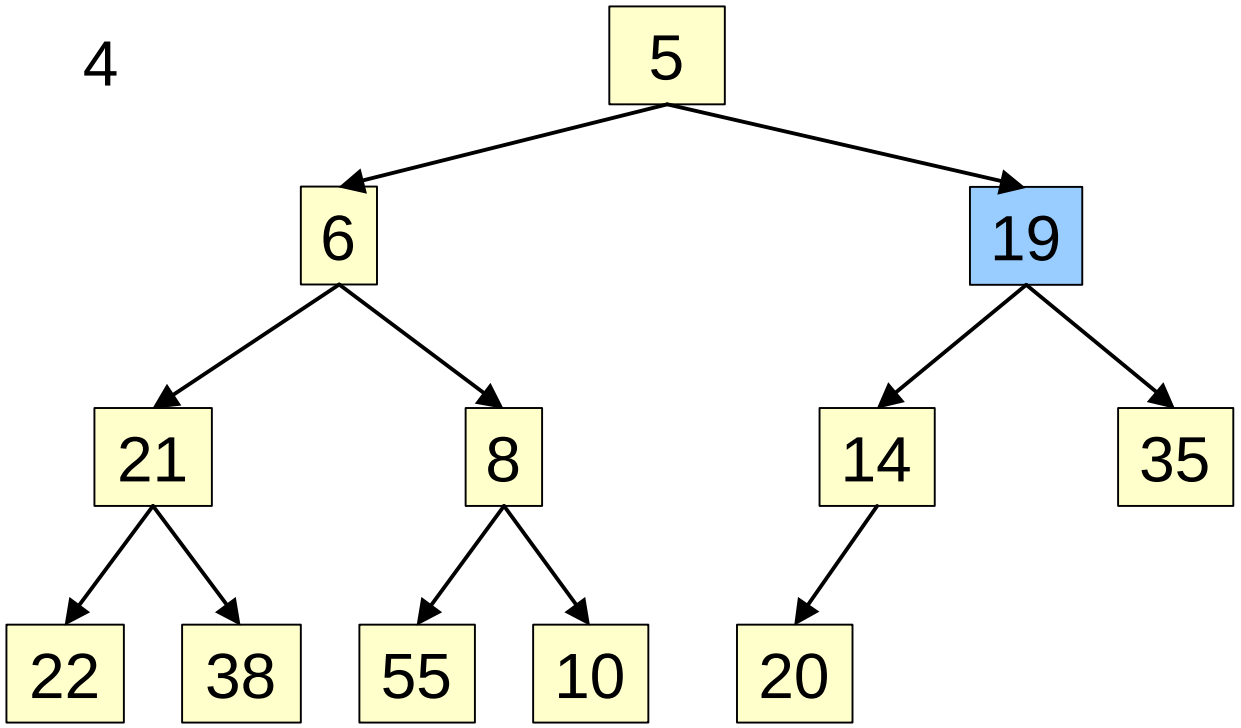
# get()



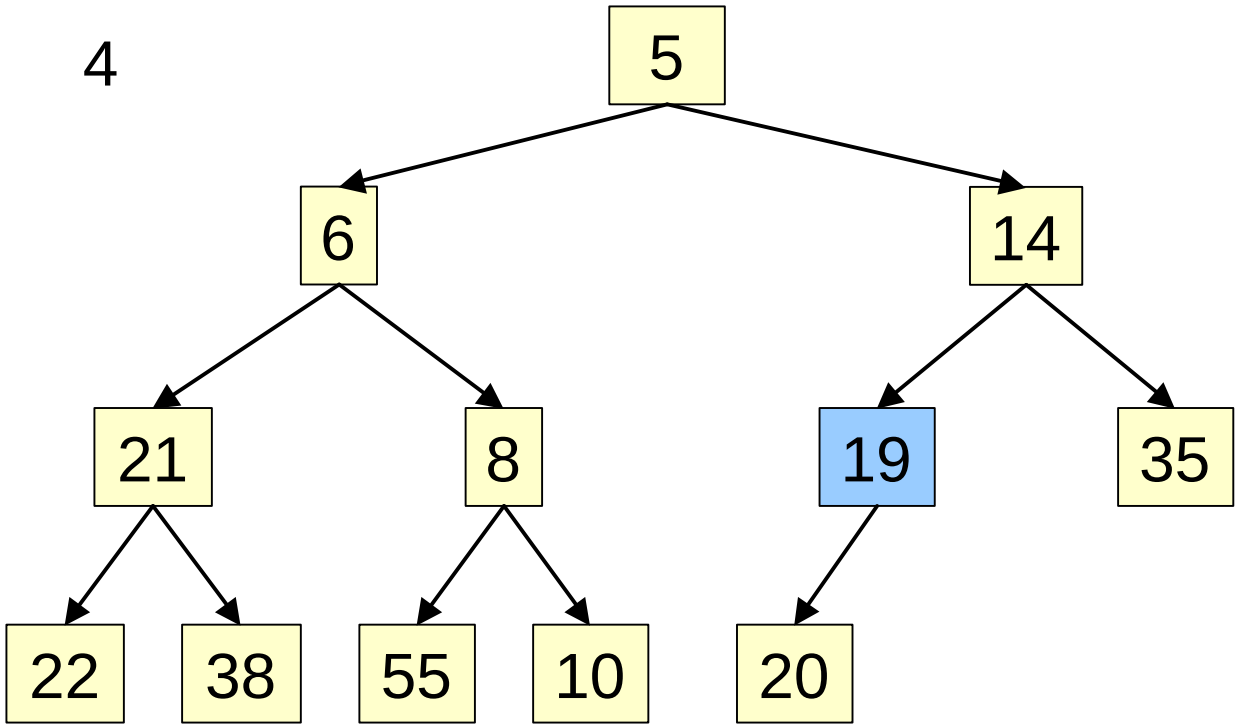
# get()



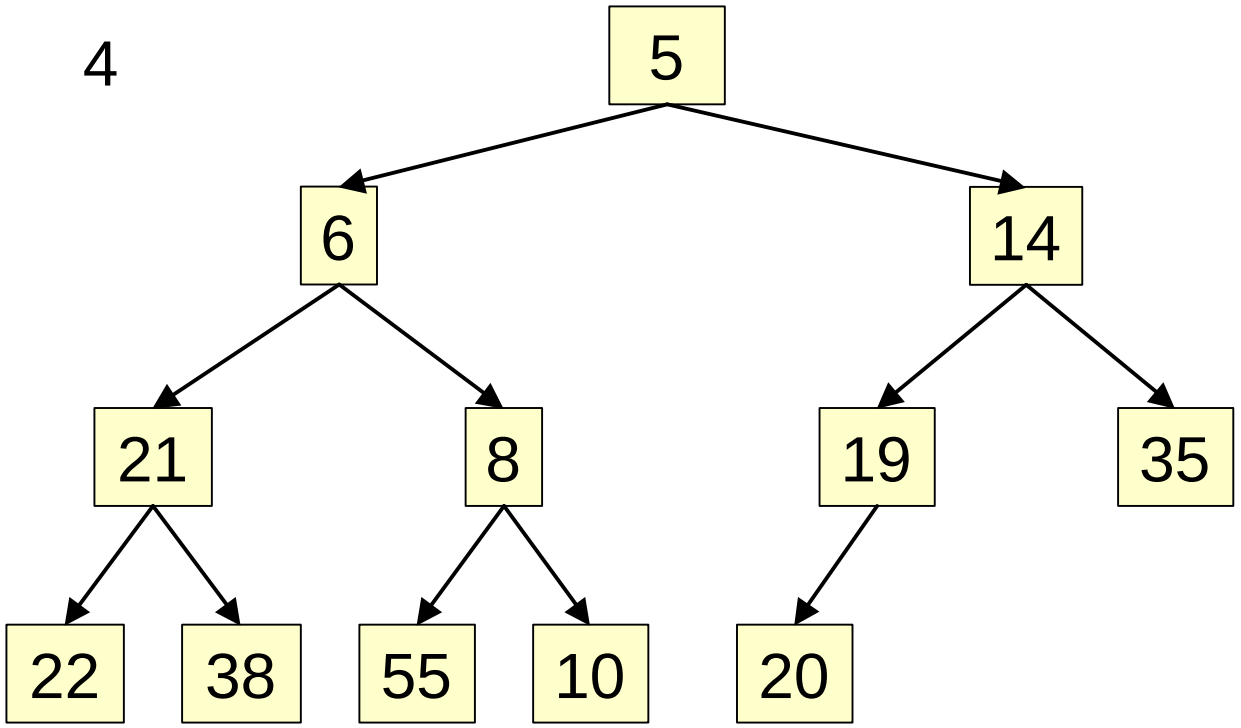
# get()



# get()

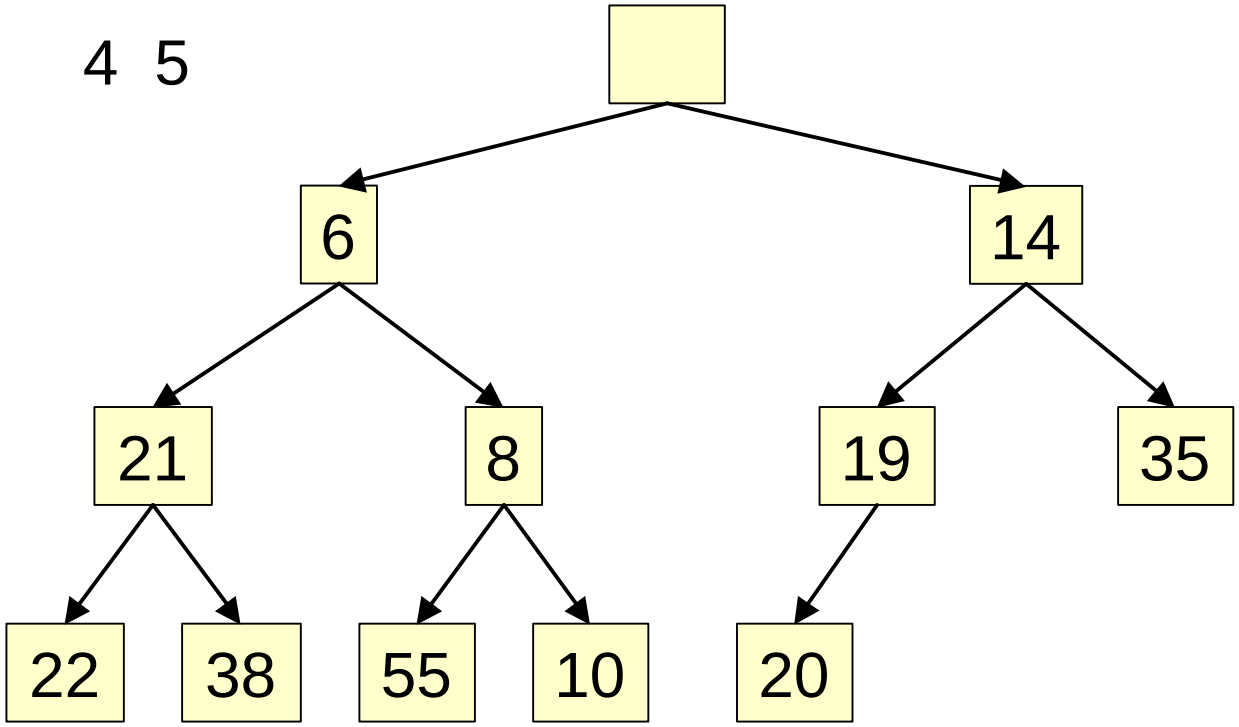


# get()

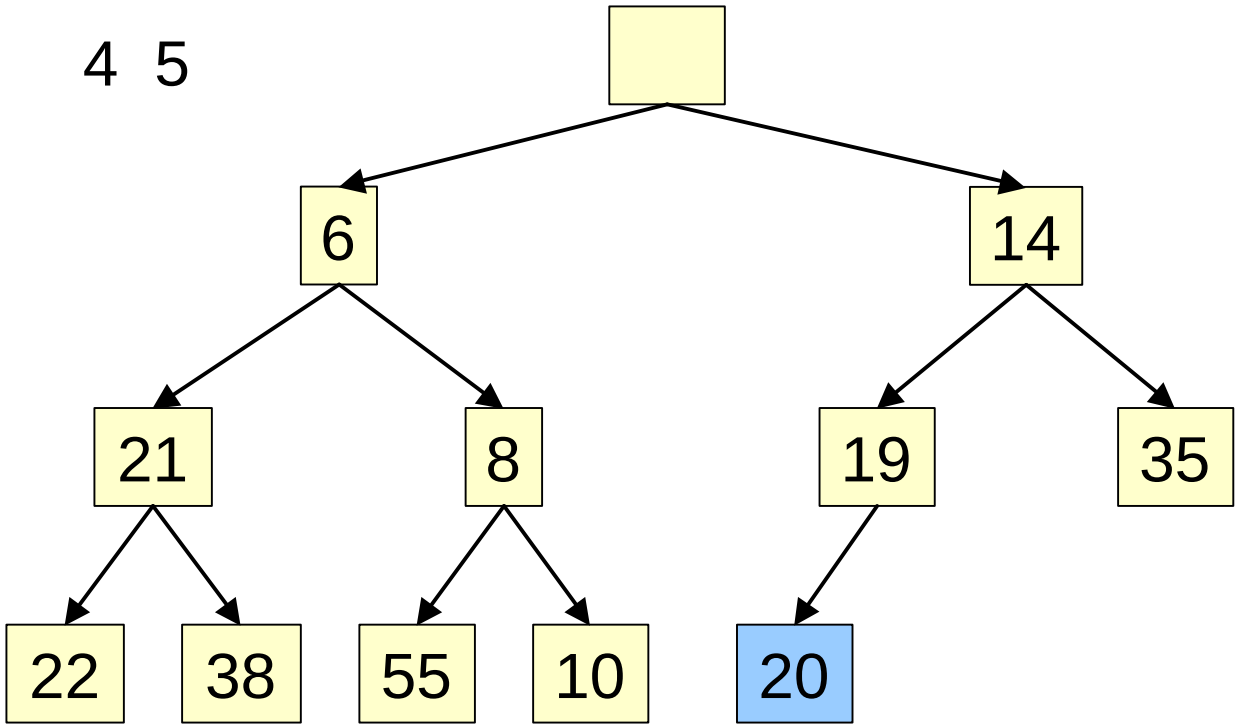


# get()

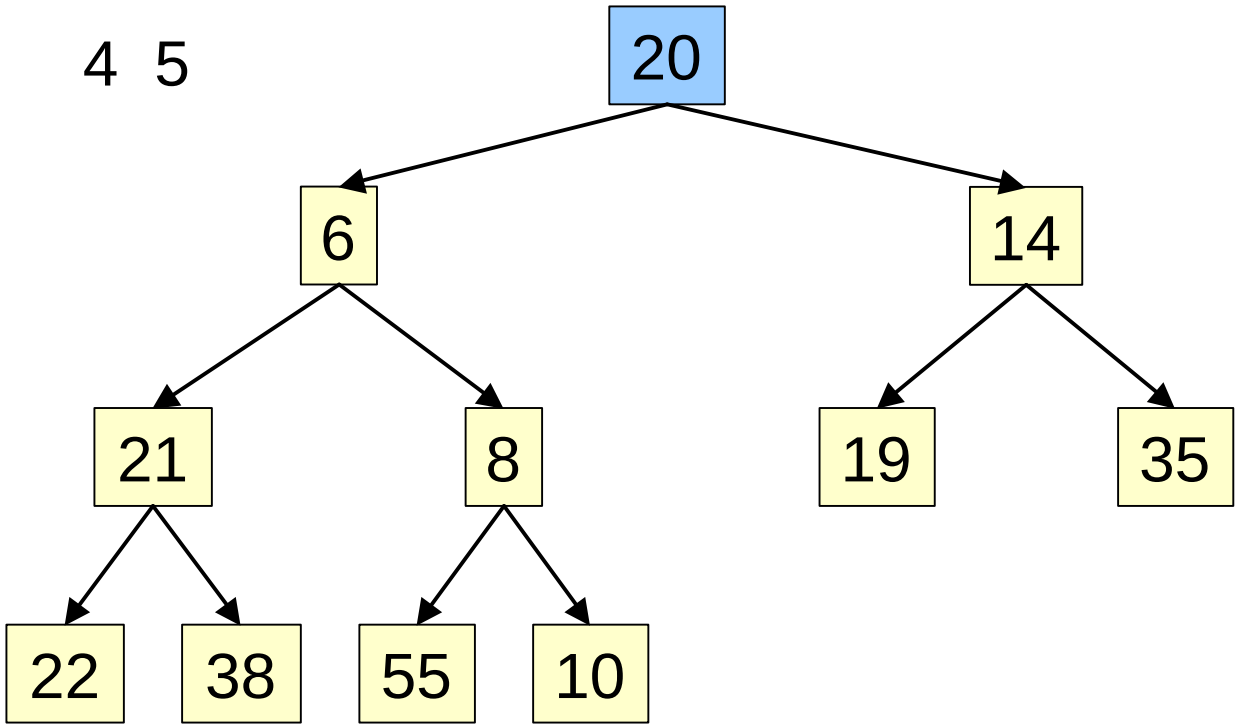
4 5



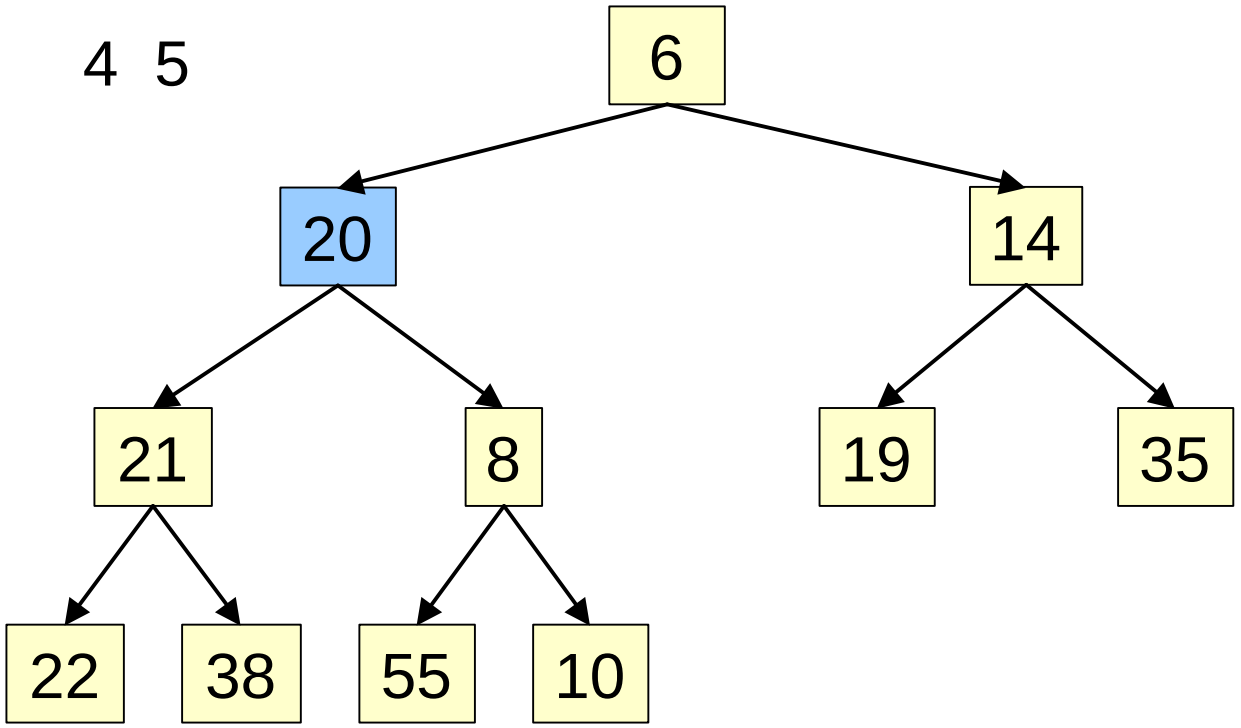
# get()



# get()

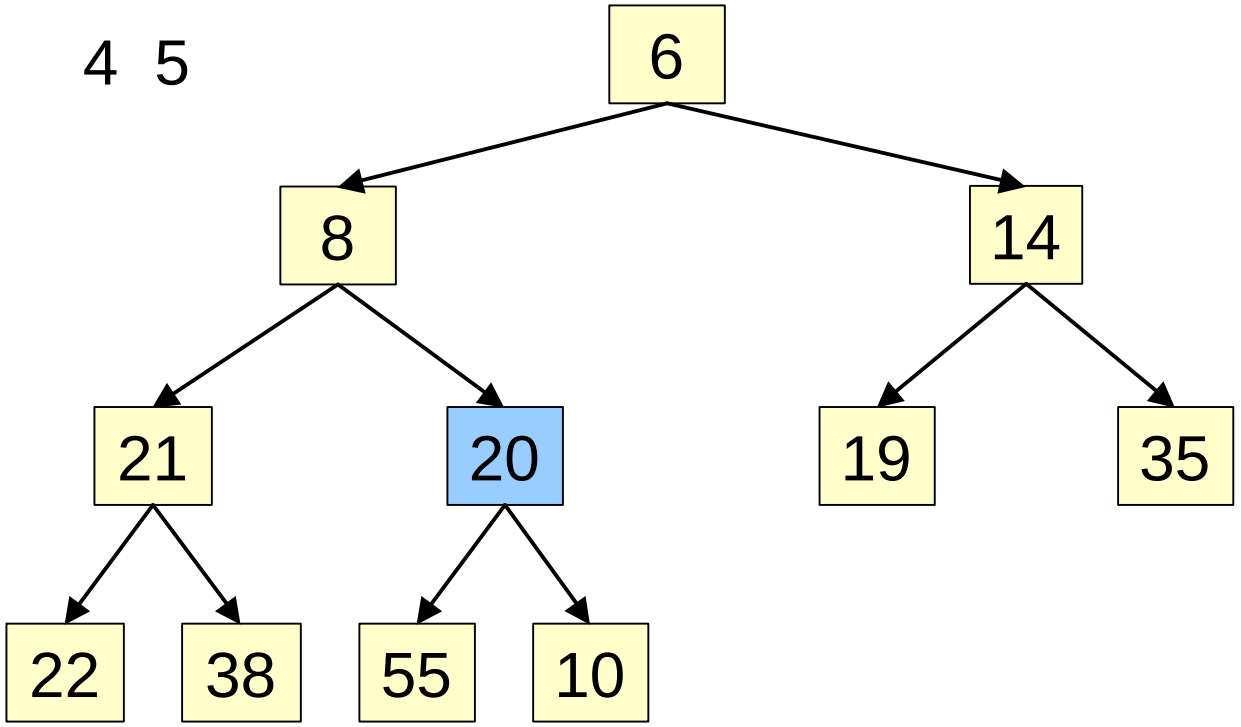


# get()

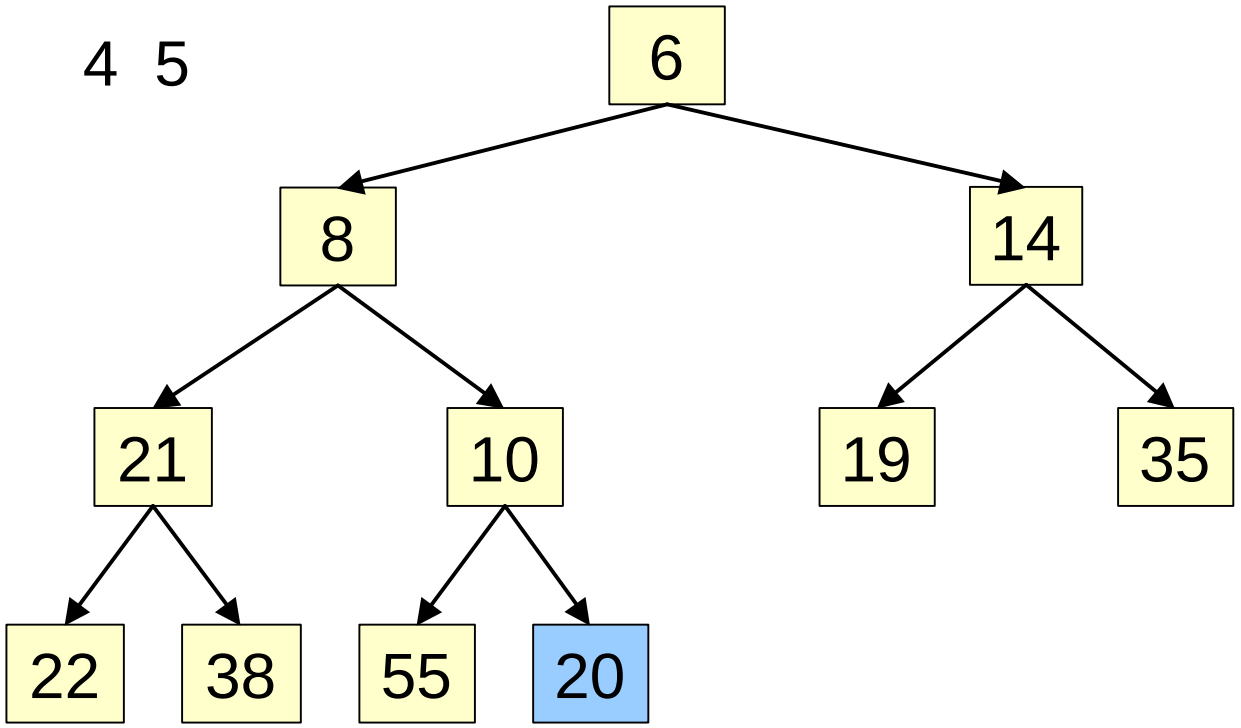


# get()

4 5

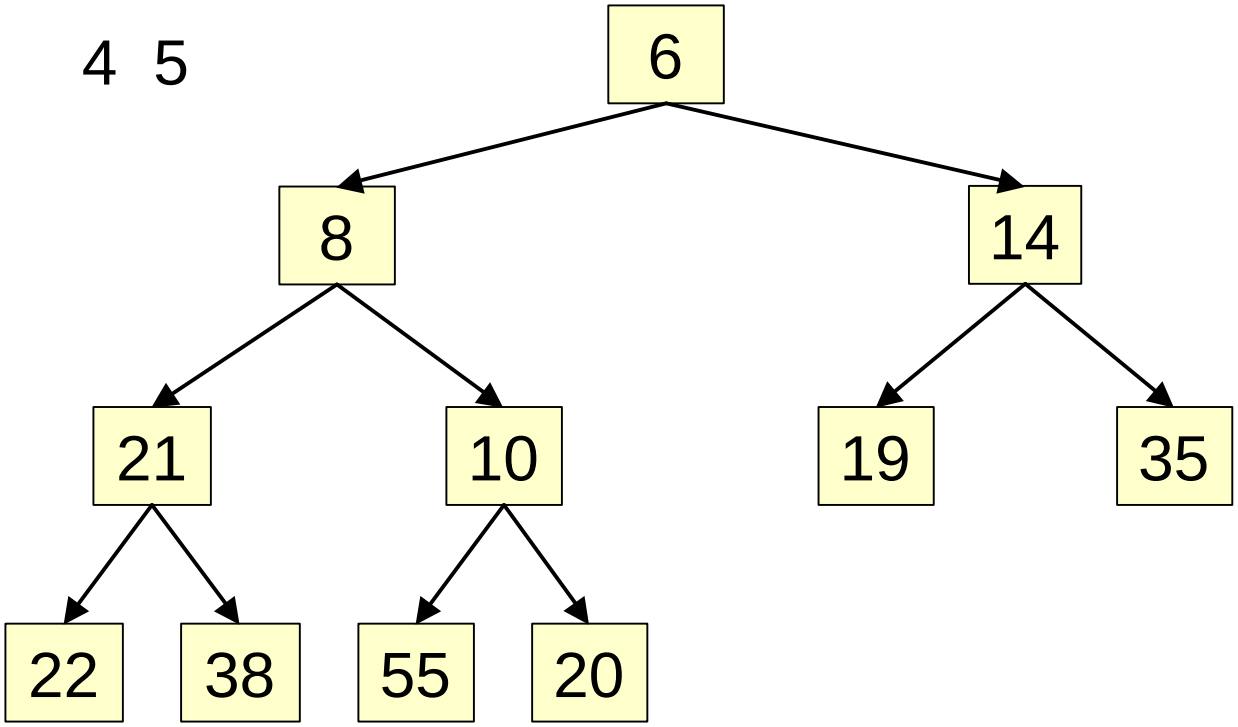


# get()



# get()

4 5



# `get()`

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

# get()

```
public E get() {
  if (isEmpty()) throw new NoSuchElementException();
  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,

1. Put all n elements into a heap – O(n log n)
2. Repeatedly get the min – O(n log n)

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

# Building the Heap

- We can actually do better than O(n log n) when building the heap:
  - First just put the data into a binary tree with the same balanced property as a heap
  - Then, starting at the bottom right of the tree, we'll visit each node in reverse level order. At each node, we'll call the same `rotateDown(i)` function that `get()` uses
  - You can show that this works, and is O(n). (Convince yourself that it works, and see the book for the running time analysis)