Data Structures

Heaps and Priority Queues

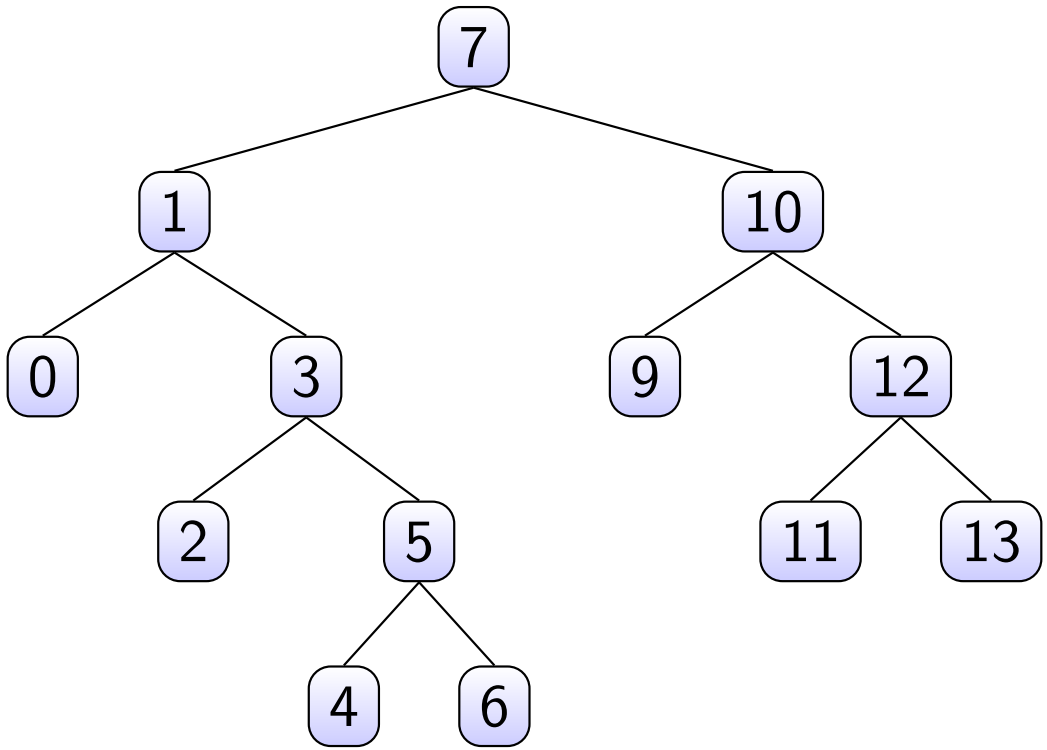
CS284

Heaps

Priority Queues

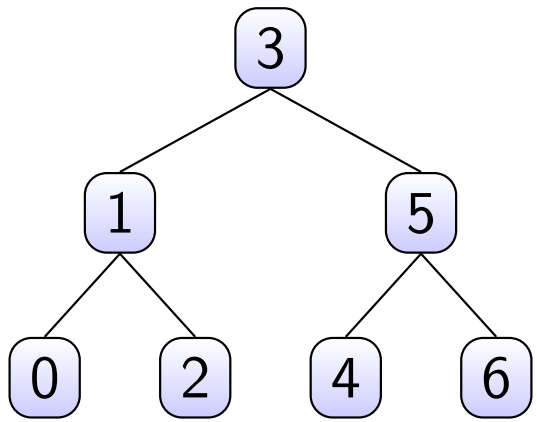
Full, Perfect, and Complete Binary Trees (cont.)

A full binary tree is a binary tree where all nodes have either 2 children or 0 children (the leaf nodes)



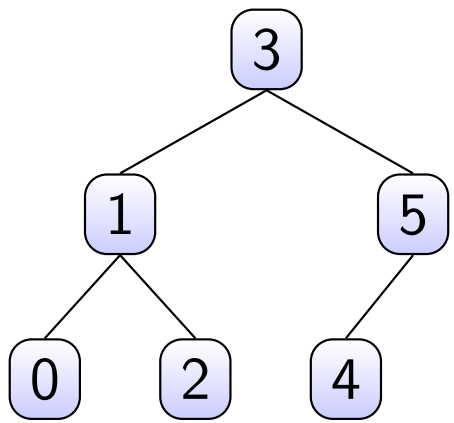
Full, Perfect, and Complete Binary Trees (cont.)

- A perfect binary tree is
  1. a full binary tree of height n
  2. all leaves have the same depth
- Item 2 is equivalent to requiring that the tree have exactly $2^n - 1$ nodes
- In this case, $n = 3$ and $2^n - 1 = 7$



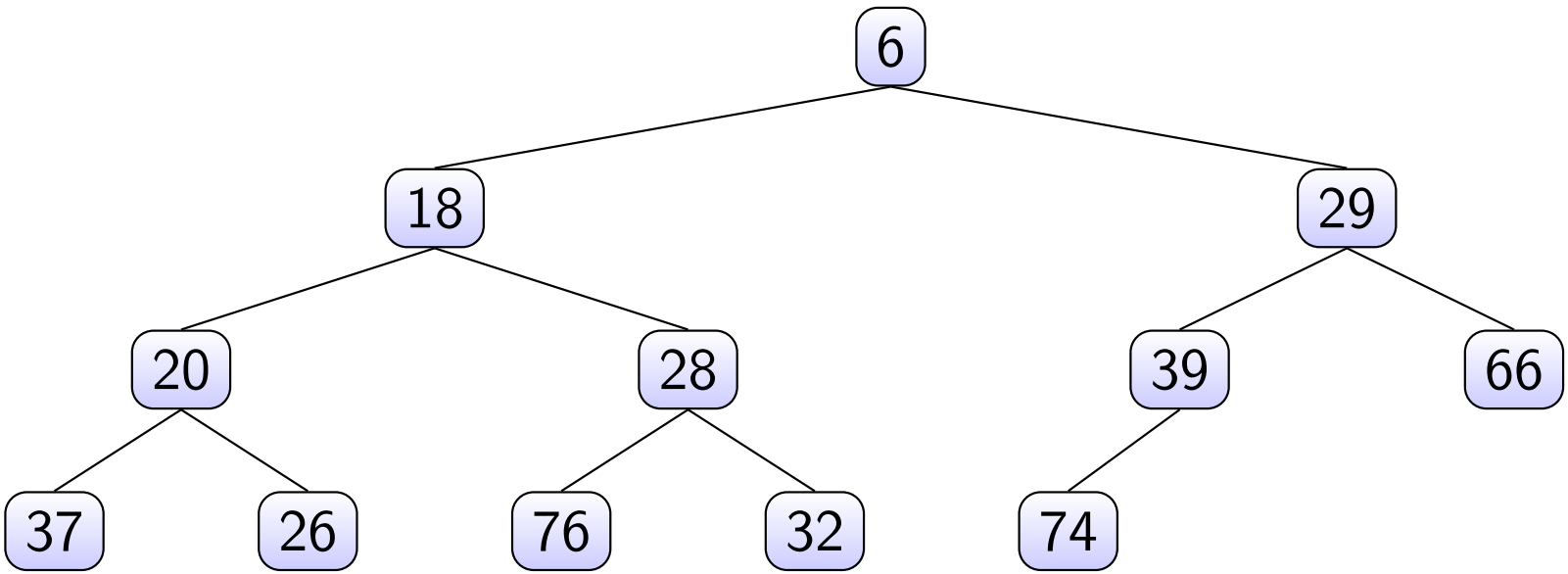
Full, Perfect, and Complete Binary Trees (cont.)

- A complete binary tree is a perfect binary tree through level $n-1$ with some extra leaf nodes at level n (the tree height), all toward the left



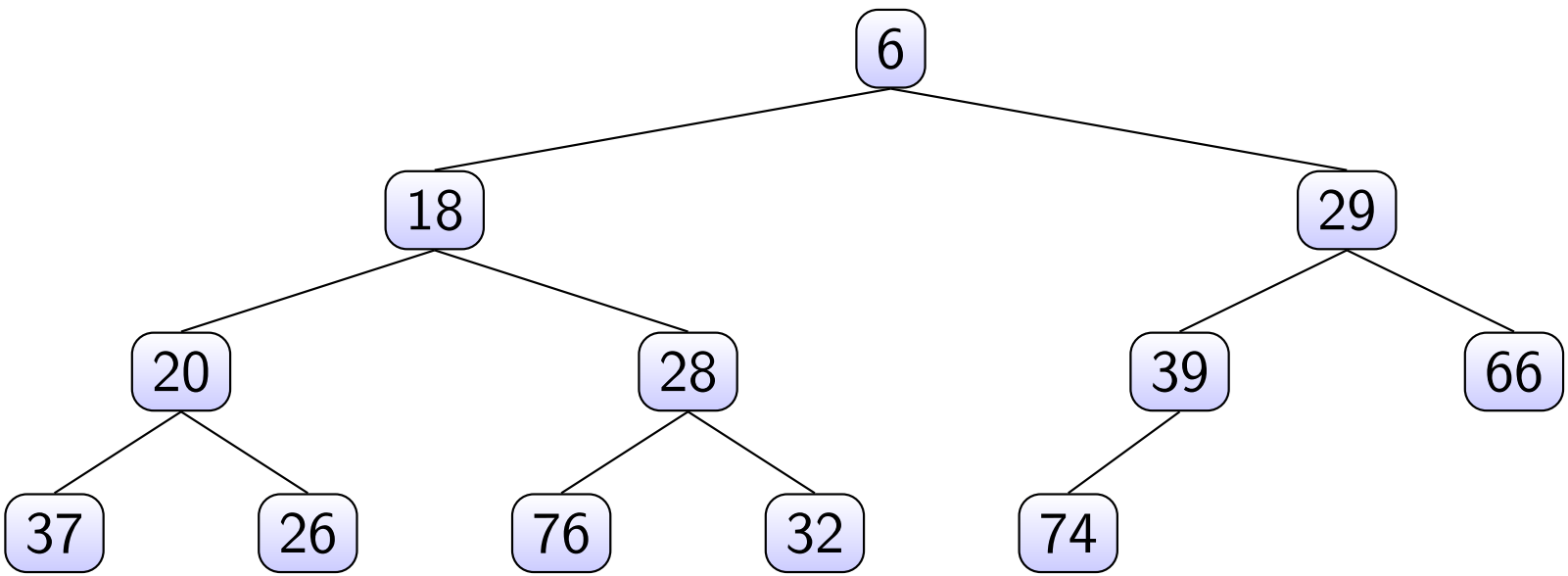
Heaps and Priority Queues

- A heap is a complete binary tree with the following properties
  - The value in the root is the smallest item in the tree
  - Every subtree is a heap

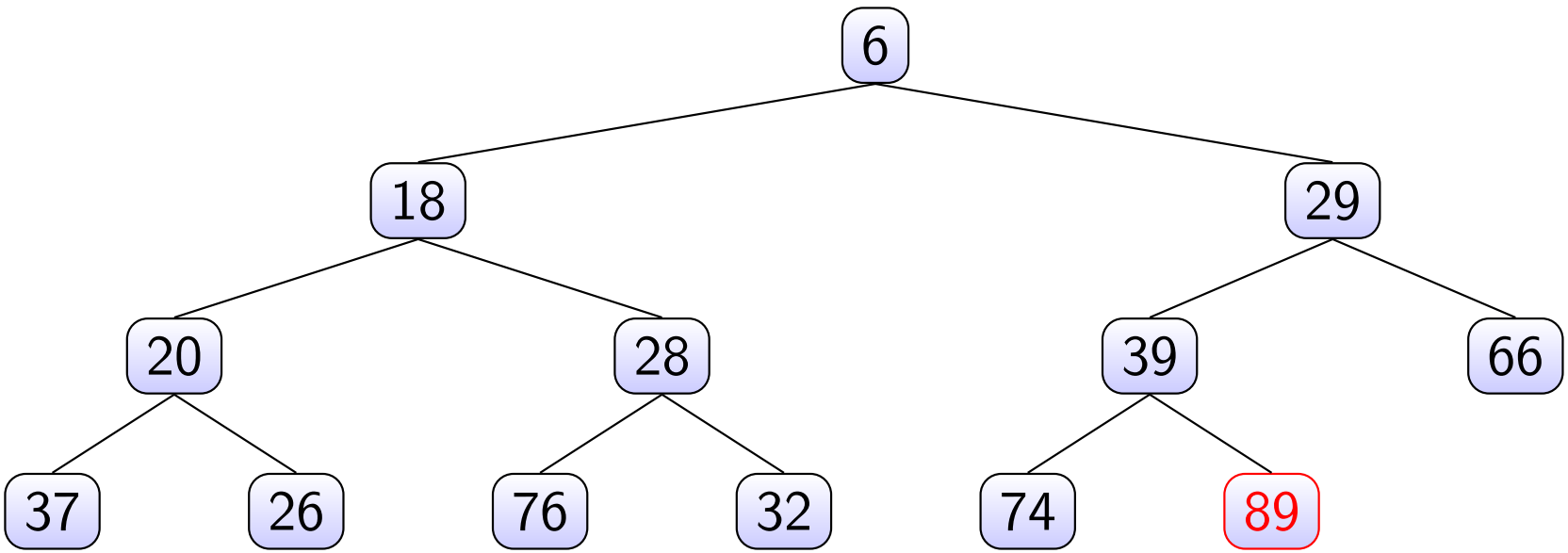


Inserting an Item e into a Heap

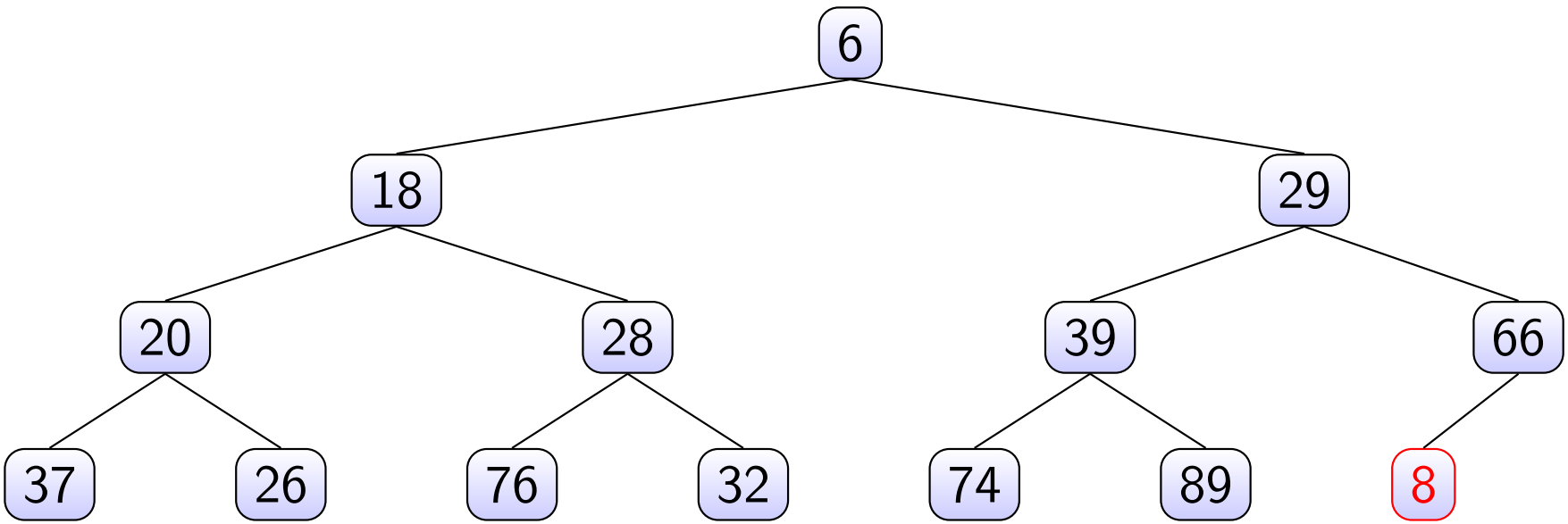
```
Insert e in the next position at the bottom of the Heap
while (e not at the root and is smaller than its root){
     Swap e with its parent, moving e up the heap
}
```



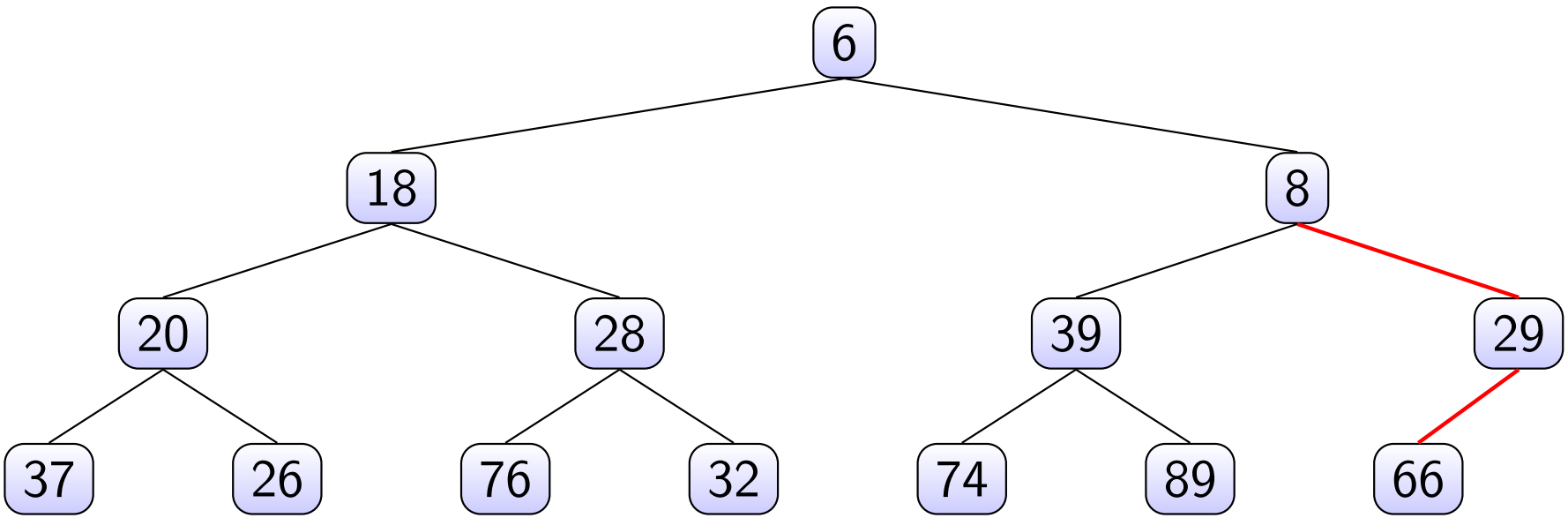
Example 1 – Insert 89



Example 2 – Insert 8



Example 2 – Insert 8 (cont.)



Performance

```
Insert e in the next position at the bottom of the Heap
while (e not at the root and is smaller than its root){
     Swap e with its parent, moving e up the heap
}
```

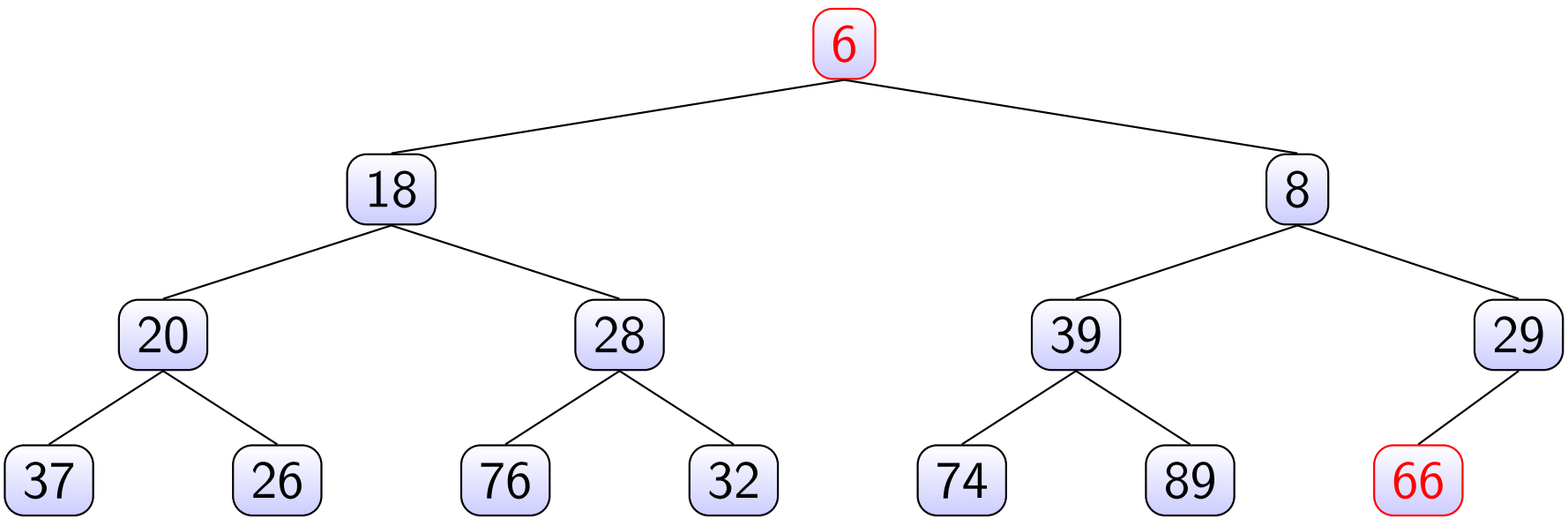
Performance

```
Insert e in the next position at the bottom of the Heap
while (e not at the root and is smaller than its root){
     Swap e with its parent, moving e up the heap
}
```

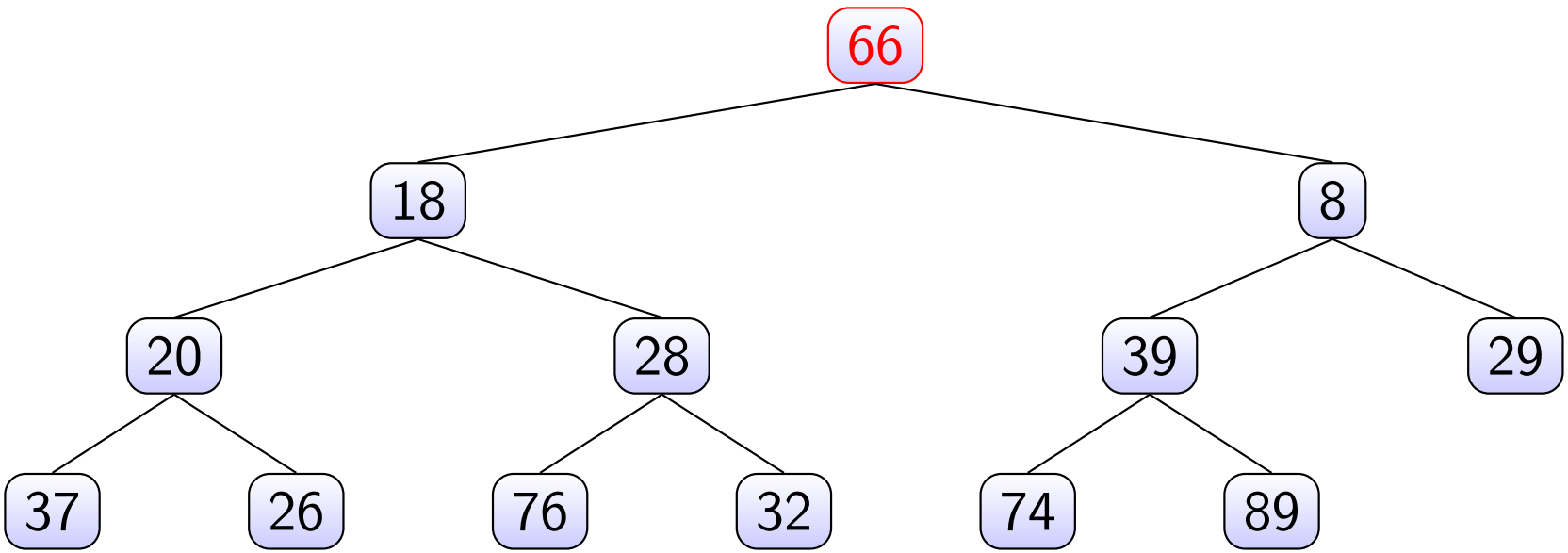
- Insertion is $\mathcal{O}(\log n)$

Removing the Item in the Root of a Heap

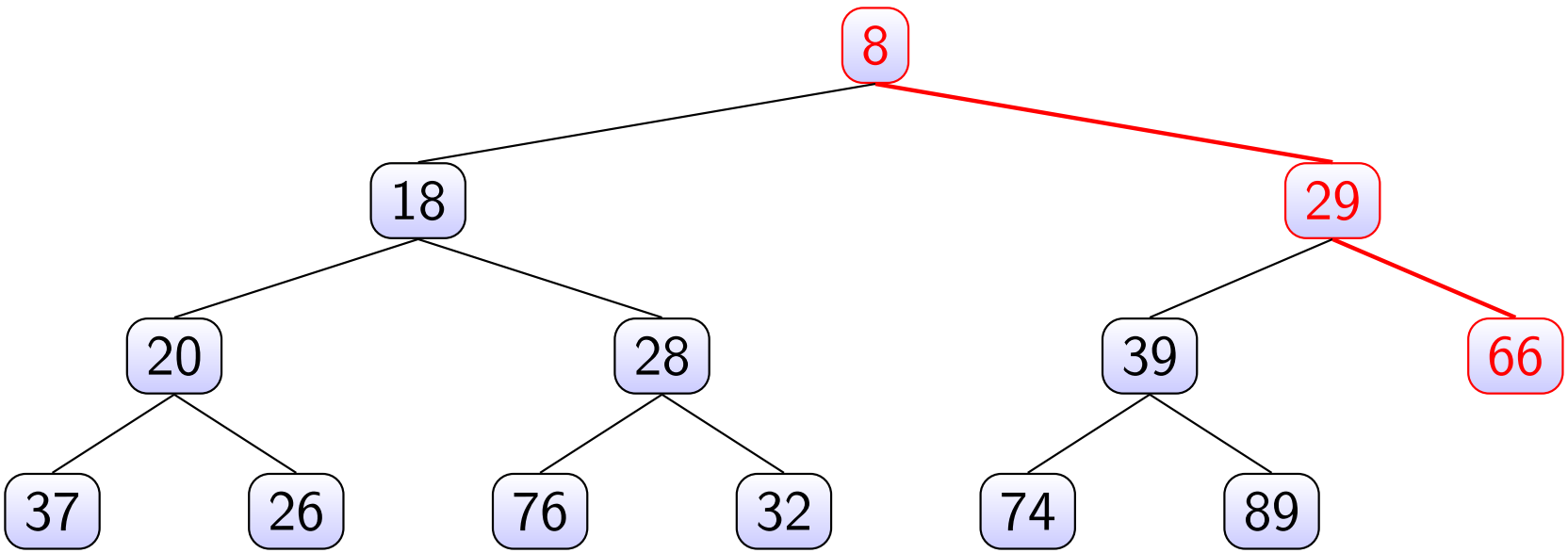
```
Remove the item in the root by replacing it with the last item
while (item LIH has children and it is larger than than either
    Swap item LIH with its smaller child, moving LIH down the
}
```



Removing the Item in the Root of a Heap



Removing the Item in the Root of a Heap



Performance

```
Remove the item in the root by replacing it with the last item
while (item LIH has children and it is larger than than either
    Swap item LIH with its smaller child, moving LIH down the
}
```

Performance

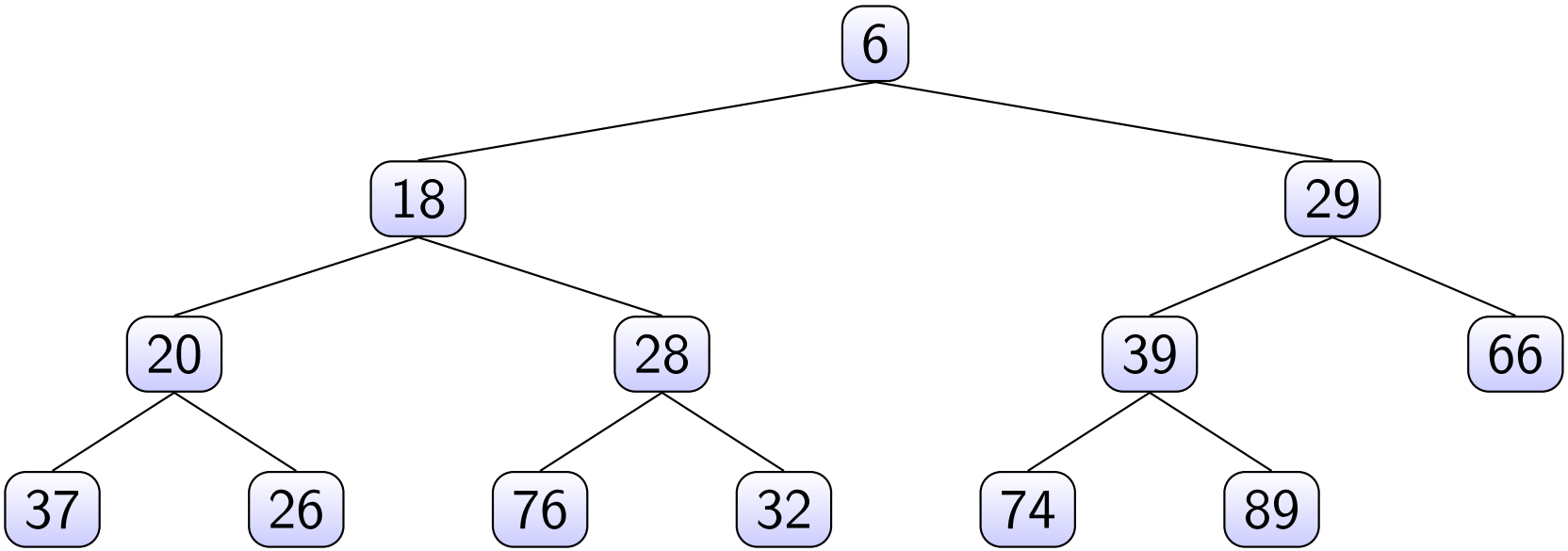
```
Remove the item in the root by replacing it with the last item
while (item LIH has children and it is larger than than either
    Swap item LIH with its smaller child, moving LIH down the
}
```

- Insertion is $\mathcal{O}(\log n)$

Implementing a Heap

- Because a heap is a complete binary tree, it can be implemented efficiently using an array rather than a linked data structure

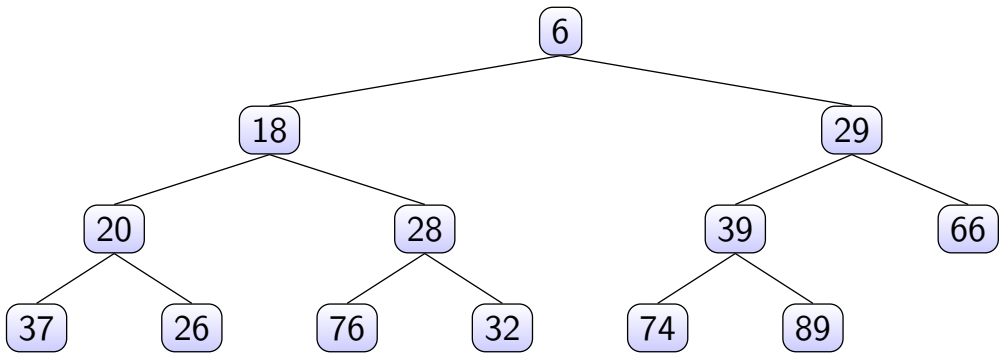
Implementing a Heap (cont.)



0	1	2	3	4	5	6	7	8	9	10	11	12
6	18	29	20	28	39	66	37	26	76	32	74	89

Node at p: left child at $2p + 1$, right child at $2p + 2$ and parent at $(p - 1)/2$.

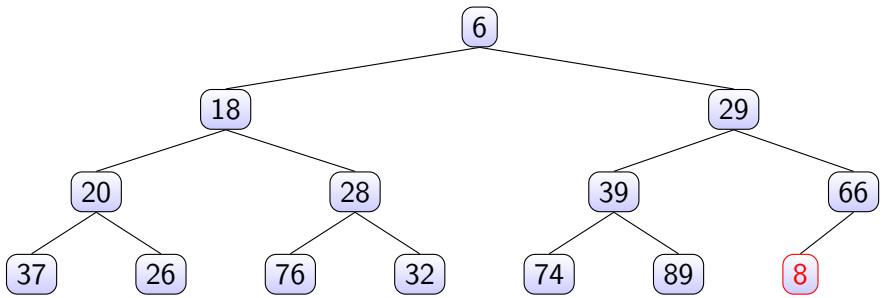
Inserting e into a Heap Implemented as an ArrayList




```
Insert e at the end of Array
child = table.size()
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13
6	18	29	20	28	39	66	37	26	76	32	74	89	

Inserting `e` into a Heap Implemented as an ArrayList

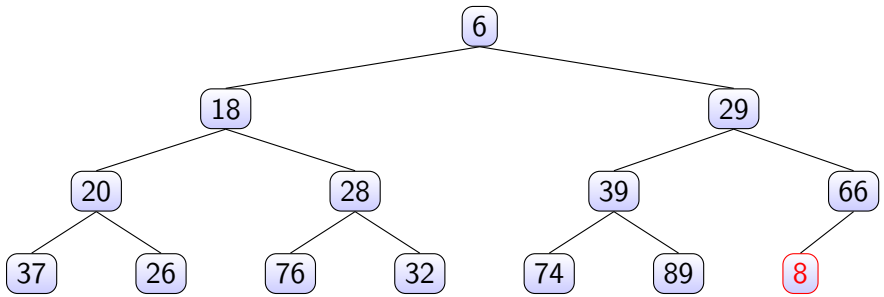



```
Insert e at the end of the Arr
child = table.size()
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13
6	18	29	20	28	39	66	37	26	76	32	74	89	8

`child=13`

Inserting `e` into a Heap Implemented as an ArrayList

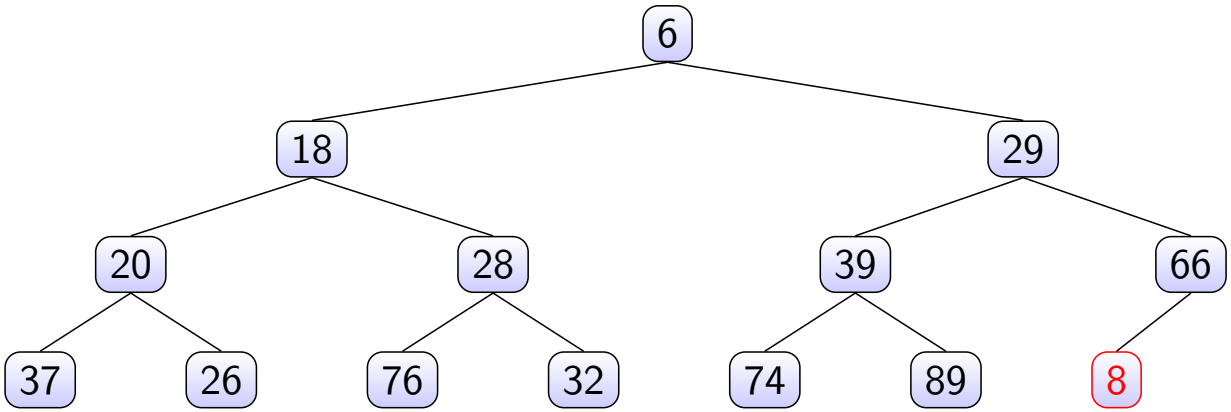



```
parent = (child-1)/2
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13
6	18	29	20	28	39	66	37	26	76	32	74	89	8

```
child=13
parent=6
```

Inserting into a Heap Implemented as an ArrayList

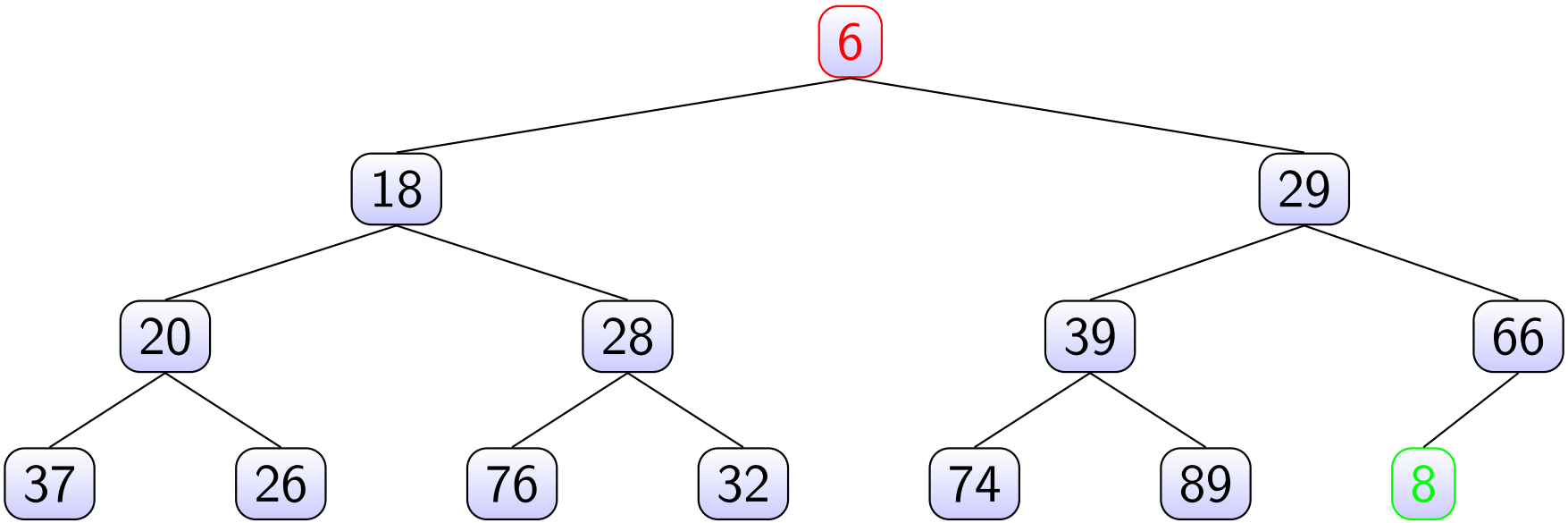



```
while  (parent>=0 and table[parent]>table[child]) {
  swap(table[parent],table[child])
  child  = parent
  parent = (child-1)/2
}
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13
6	18	29	20	28	39	66	37	26	76	32	74	89	8

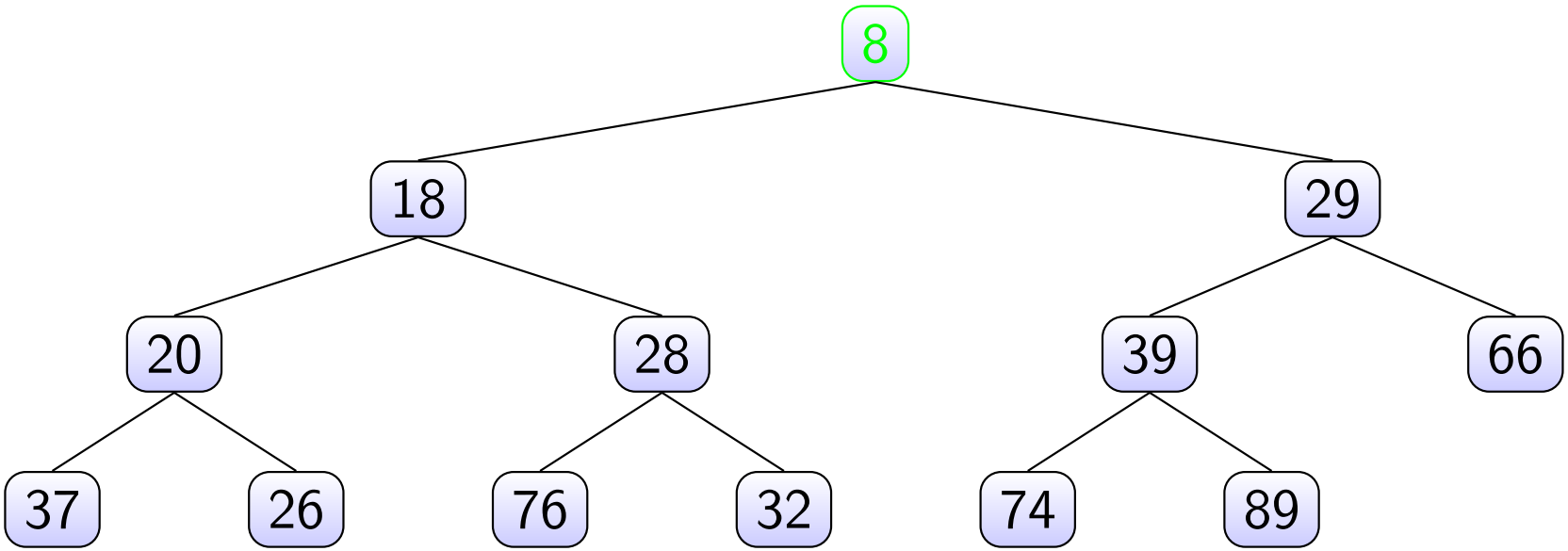
parent=6

Removal from a Heap



Remove and place last element in root

Removal from a Heap



"Trickle" root down

Removal from a Heap Implemented as an ArrayList

```
Remove the last element le (i.e. the one at size()-1)
table[0] = le
parent   = 0
while (true) {
    leftChild  = (2 * parent) + 1;
    rightChild = leftChild + 1;
    if (leftChild >= table.size())  // bottom of tree
        { Break out of loop }
    minChild = leftChild;           // get minChild
    if (rightChild < table.size() and table[rightChild] < table[leftCh
        { minChild = rightChild; }
    if (table[parent] > table[minChild]) {  // swap if necessary
          swap(table[parent],table[minChild]);
          parent = minChild; }
    else
        { Break out of loop }
}
```

Merging Heaps

- How would you merge two heaps H_1 and H_2?
- Remove each element from H_1 and insert it into H_2
- $\mathcal{O}(n\ logn)$ where n is the sum of the sizes of H_1 and H_2
- Faster methods are possible if one changes the structure used for representing heaps
- These are called mergeable heaps. Eg.
  - leftist heap
  - binomial heap

Summary of Performance of the Heap

- `remove` traces a path from the root to a leaf
- `insert` traces a path from a leaf to the root
- This requires at most h steps where h is the height of the tree
- The largest full tree of height h has $2^h - 1$ nodes
- The smallest complete tree of height h has $2^{(h-1)}$ nodes
- Both `insert` and `remove` are $\mathcal{O}(\log n)$

Heaps

Priority Queues

Priority Queues

- The heap is used to implement a special kind of queue called a priority queue
- The heap is not very useful as an ADT on its own
  - We will not create a Heap interface or code a class that implements it
  - Instead, we will incorporate its algorithms when we implement a priority queue class and heapsort
- Sometimes a FIFO queue may not be the best way to implement a waiting line
- A priority queue is a data structure in which only the highest-priority item is accessible (as opposed to the first item entered)

`PriorityQueue<E>` Class

- Java provides this class; it implements the `Queue<E>` interface

```
  public interface Queue<E> extends Collection<E> {

// Insert item at  rear of a queue
boolean offer(E item)

// Remove entry from the front of the queue; returns null is the queue
E poll()

// Return element at the front of the queue without removing it; retur
E peek()
}
```

- Only sample shown above

Using a Heap as the Basis of a Priority Queue

- In a priority queue, just like a heap, the smallest item always is removed first
- Because heap insertion and removal is $\mathcal{O}(\log n)$, a heap can be the basis of a very efficient implementation of a priority queue
- `java.util.PriorityQueue` uses an `Object[] array`
- We will use an `ArrayList` for our custom priority queue, `KWPriorityQueue`

Design of a KWPriorityQueue Class

Data Field

```
ArrayList<E> theData
Comparator<E> comparator
```

Constructor

```
public KWPriorityQueue()
public KWPriorityQueue(int cap, Comparator<E> comp)
```

Method

```
private int compare(E left, R right)
public void swap(int i, int j)
```

plus all those methods from the interface

Design of a KWPriorityQueue Class

```
import java.util.*;

public class KWPriorityQueue<E> extends AbstractQueue<E>
                  implements Queue<E> {

  private ArrayList<E> theData;

  /** An optional reference to a Comparator object. */
  Comparator<E> comparator = null;

  public KWPriorityQueue() {
      theData = new ArrayList<E>();
}
```

`compare` method

- If data field `comparator` references a `Comparator<E>` object, method `compare` delegates the task to the object's compare method
- If comparator is **`null`**, it will delegate to method `compareTo`

```
private int compare(E left, E right) {
  if (comparator != null) { // A Comparator is defined.
    return comparator.compare(left, right);
  } else {                  // Use left's compareTo method.
    return ((Comparable<E>) left).compareTo(right);
  }
}
```

`offer` Method

```
@Override
public boolean offer(E item) {
  // Add the item to the heap.
  theData.add(item);
  // child is newly inserted item.
  int child = theData.size() - 1;
  int parent = (child - 1) / 2; // Find child's parent.
  // Reheap
  while (parent >= 0 && compare(theData.get(parent), theData.get(child
      swap(parent, child);
      child = parent;
      parent = (child - 1) / 2;
  }
  return true;
}
```

`poll` Method

```
@Override
public E poll() {
    if (isEmpty()) {
        return null;
    }
    // Save the top of the heap.
    E result = theData.get(0);
    // If only one item then remove it.
    if (theData.size() == 1) {
        theData.remove(0);
        return result;
    }

    /* Remove the last item from the ArrayList and place it into the f
    theData.set(0, theData.remove(theData.size() - 1));
    // continued
```

`poll` Method

```
// The parent starts at the top.
int parent = 0;
while (true) {
        int leftChild = 2 * parent + 1;
        if (leftChild >= theData.size()) {
           break; // Out of heap.
        }
        int rightChild = leftChild + 1;
        // Assume leftChild is smaller.
        int minChild = leftChild;
        // See whether rightChild is smaller.
        if (rightChild < theData.size()
              && compare(theData.get(leftChild), theData.get(rightChil
            minChild = rightChild;
        }
```

`poll` Method

```
        // assert: minChild is the index of the smaller child.
        // Move smaller child up heap if necessary.
        if (compare(theData.get(parent),
                    theData.get(minChild)) > 0) {
            swap(parent, minChild);
            parent = minChild;
        }
      else { // Heap property is restored.
            break;
        }
    }
    return result;
}
```

Other Methods

- The `iterator` and `size` methods are implemented via delegation to the corresponding ArrayList methods
- Method `isEmpty` tests whether the result of calling method size is 0 and is inherited from class `AbstractCollection`
- The implementations of methods `peek` and `remove` are left as exercises
- We now look at the `compare` method

Setting the Comparator

```
public KWPriorityQueue(int cap, Comparator<E> comp) {
    if (cap < 1) {
        throw new IllegalArgumentException();
    }
    theData = new ArrayList<E>(cap + 1);
    comparator = comp;
}
```

Comparator Example - `PrintDocuments`

- The class `PrintDocument` is used to define documents to be printed on a printer
- We want to order documents by a value that is a function of both size and time submitted
- In the client program, use

```
Queue printQueue =
    new PriorityQueue(new ComparePrintDocuments());
```

- `ComparePrintDocuments` implements the interface `Comparator`

PrintDocuments Example (cont.)

```
import java.util.Comparator;

public class ComparePrintDocuments
        implements Comparator<PrintDocument> {

    /** Weight factor for size. */
    private static final double P1 = 0.8;
    /** Weight factor for time. */
    private static final double P2 = 0.2;

    @Override
    public int compare(PrintDocument left, PrintDocument right) {
     return Double.compare(orderValue(left), orderValue(right));
    }

    private double orderValue(PrintDocument pd) {
     return P1 * pd.getSize() + P2 * pd.getTimeStamp();
    }
}
```