Priority Queues

The ADT Priority Queue: A Variation of the ADT Table

- Consider how hospital assign a priority to each patient at ER.
- ADT PQ organizes objects according to their priorities.
- Definition of "Priority" depends on nature of the items. The queue.
- The first item removed is the one having the highest priority value

ADT Priority Queue

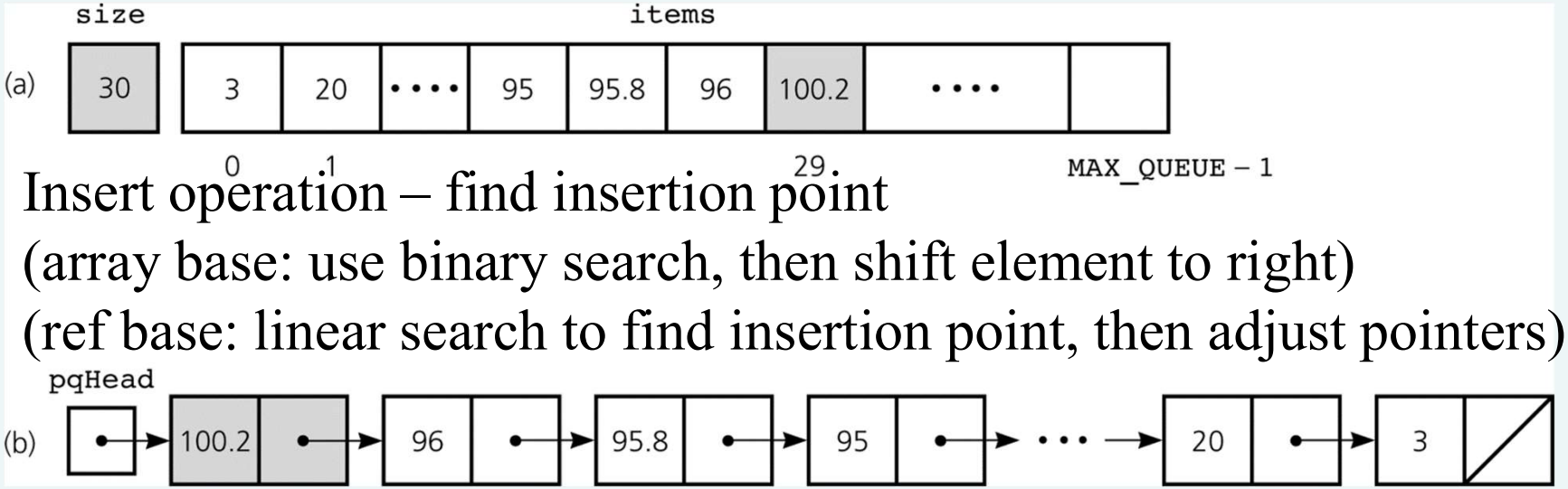
```
/** An interface for the ADT priority queue. */
public interface PriorityQueueInterface<T extends Comparable<? super T>>
{
  /** Adds a new entry to this priority queue.
     @param newEntry  An object to be added. */
  public void add(T newEntry);

  /** Removes and returns the entry having the highest priority.
     @return  Either the object having the highest priority or, if the
              priority queue is empty before the operation, null. */
  public T remove();

  /** Retrieves the entry having the highest priority.
     @return  Either the object having the highest priority or, if the
              priority queue is empty, null. */
  public T peek();

  /** Detects whether this priority queue is empty.
     @return  True if the priority queue is empty, or false otherwise. */
  public boolean isEmpty();

  /** Gets the size of this priority queue.
     @return  The number of entries currently in the priority queue. */
  public int getSize();

  /** Removes all entries from this priority queue. */
  public void clear();
} // end PriorityQueueInterface
```

The ADT Priority Queue: A Variation of the ADT Table

- Possible implementations
  - Sorted linear implementations
    - Appropriate if the number of items in the priority queue is small
      - Array-based implementation
      - Maintains the items sorted in ascending order of priority value
      - Reference-based implementation
      - Maintains the items sorted in descending order of priority value

The ADT Max Priority Queue:

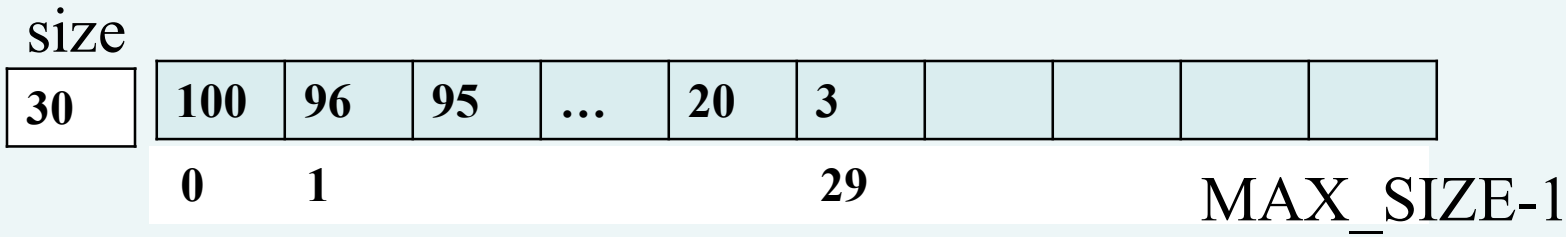
Remove operation
(array base from items[size-1] and Size--)
(ref based delete at pqHead)

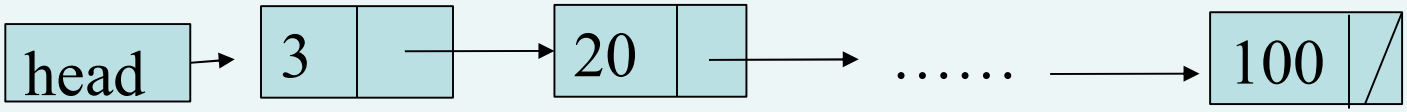


Insert operation – find insertion point
(array base: use binary search, then shift element to right)
(ref base: linear search to find insertion point, then adjust pointers)

Some implementations of the ADT priority queue: a) array based; b) reference based

The ADT Min Priority Queue:





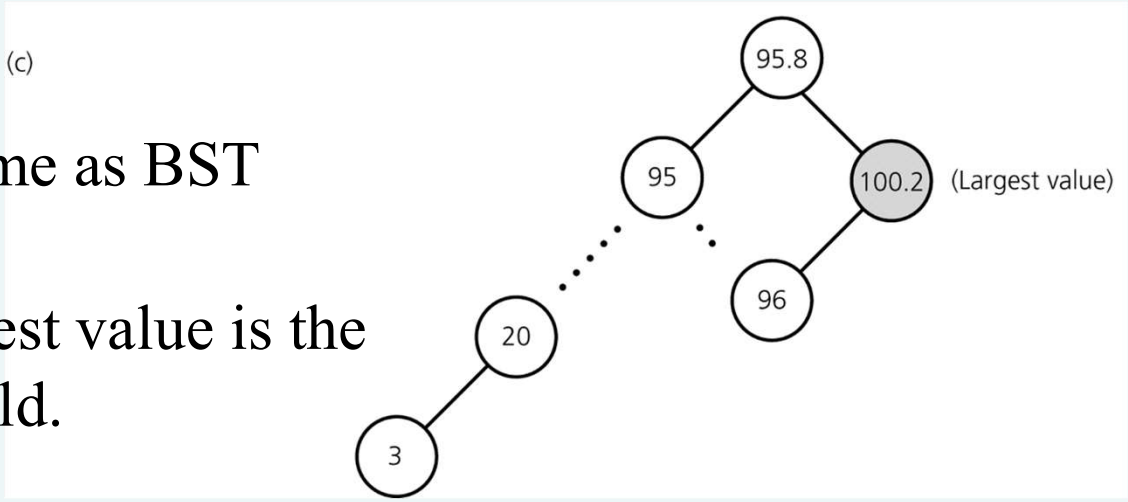
Some implementations of the ADT priority queue: a) array based; b) reference based

The ADT Priority Queue: A Variation of the ADT Table

- Possible implementations (Continued)
  - Binary search tree implementation
    - Appropriate for any priority queue

Insertion is same as BST

Deletion: largest value is the Rightmost child.



Some implementations of the ADT priority queue: c) binary search tree

Heaps

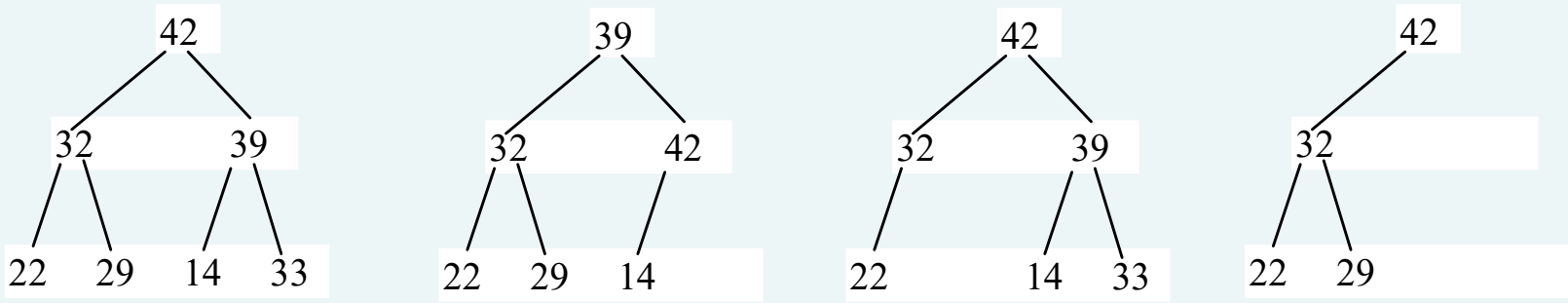
- Heap is a useful data structure for designing efficient sorting algorithms and priority queues.
- A heap is a complete binary tree
  - That is empty

  or

  - Whose root contains a search key greater than or equal to the search key in each of its children, and
  - Whose root has heaps as its subtrees

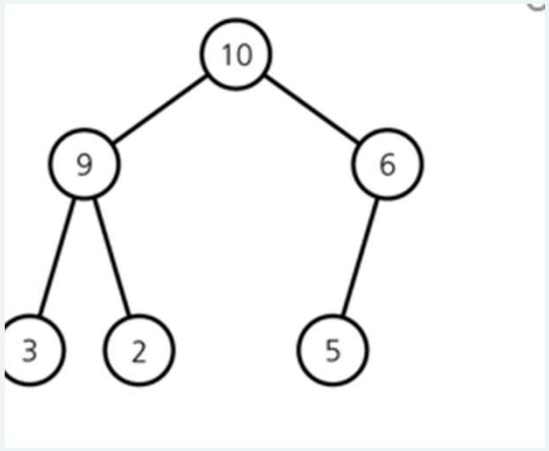
Complete Binary Tree

A binary tree is *complete* if every level of the tree is full except that the last level may not be full and all the leaves on the last level are placed left-most. For example, in the following figure, the binary trees in (a) and (b) are complete, but the binary trees in (c) and (d) are not complete. Further, the binary tree in (a) is a heap, but the binary tree in (b) is not a heap, because the root (39) is less than its right child (42).

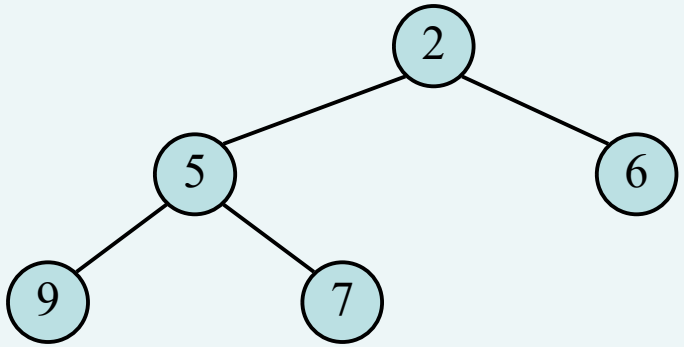


(a) (b) (c) (d)

Heaps



- Maxheap
  - A heap in which the root contains the item with the largest search key



- Minheap
  - A heap in which the root contains the item with the smallest search key

Heaps

- Pseudocode for the operations of the ADT heap

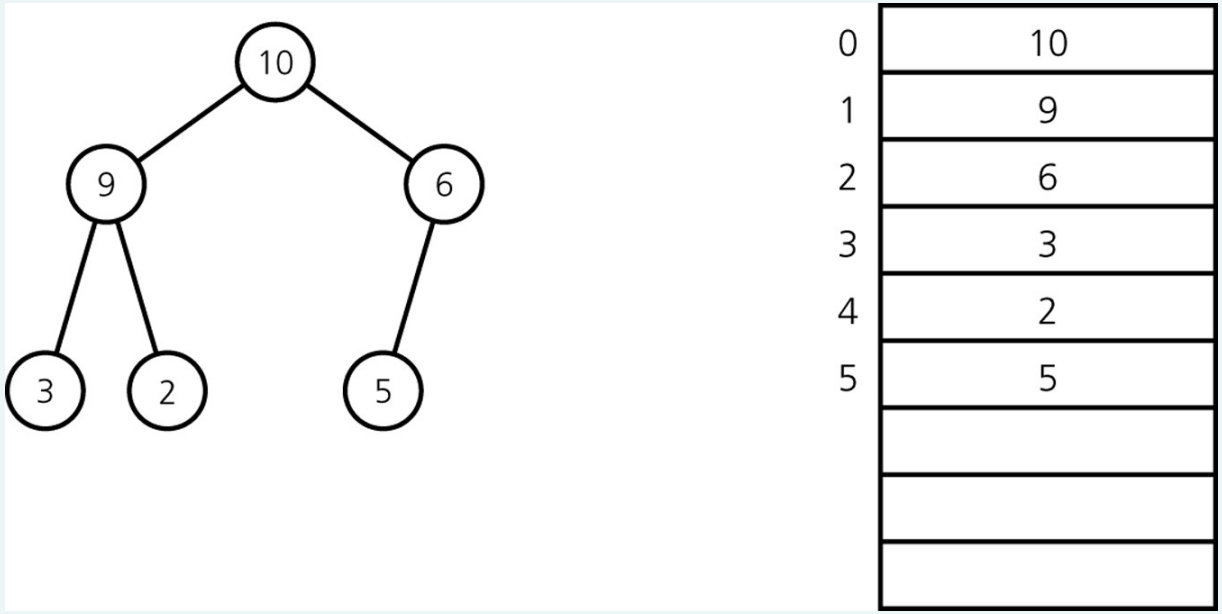
```
Heap()
// Creates an empty heap.

isEmpty()
// Determines whether a heap is empty.

add(newItem)
// Inserts newItem into a heap.

remove()
// Retrieves and then deletes a heap's root
// item. This item has the largest search key.
```

Heaps: An Array-based Implementation of a Heap

- Data fields
  - `List`: an ArrayList of heap items
  - private ArrayList<T> list; // array of list items

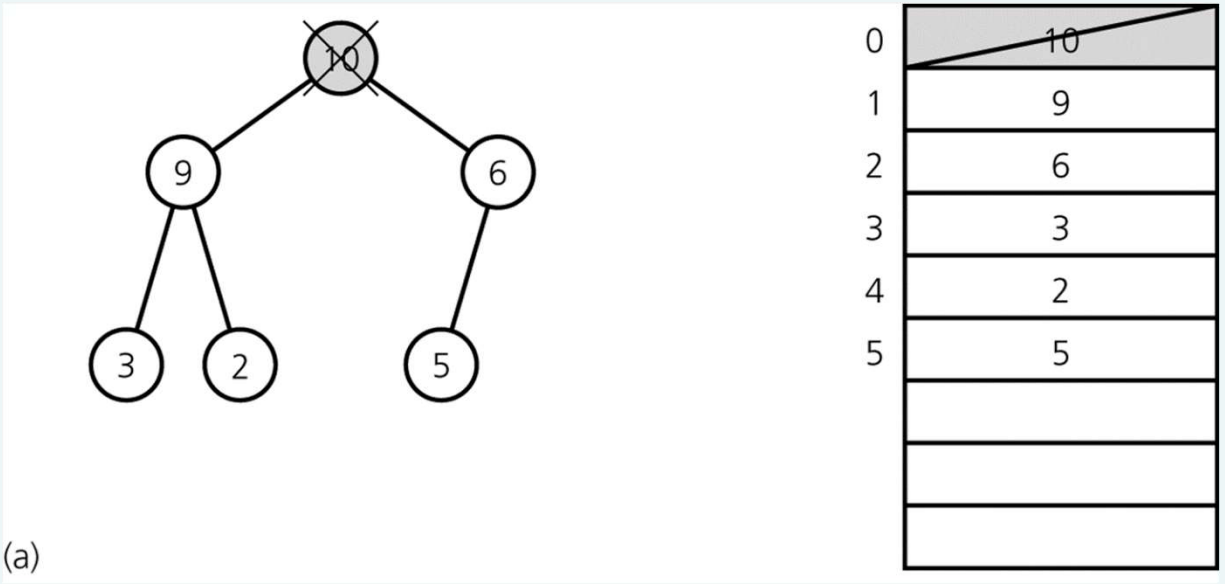
A heap with its array representation

Left child index: 2i+1
Right child index: 2i+2



Heaps: `heapDelete`

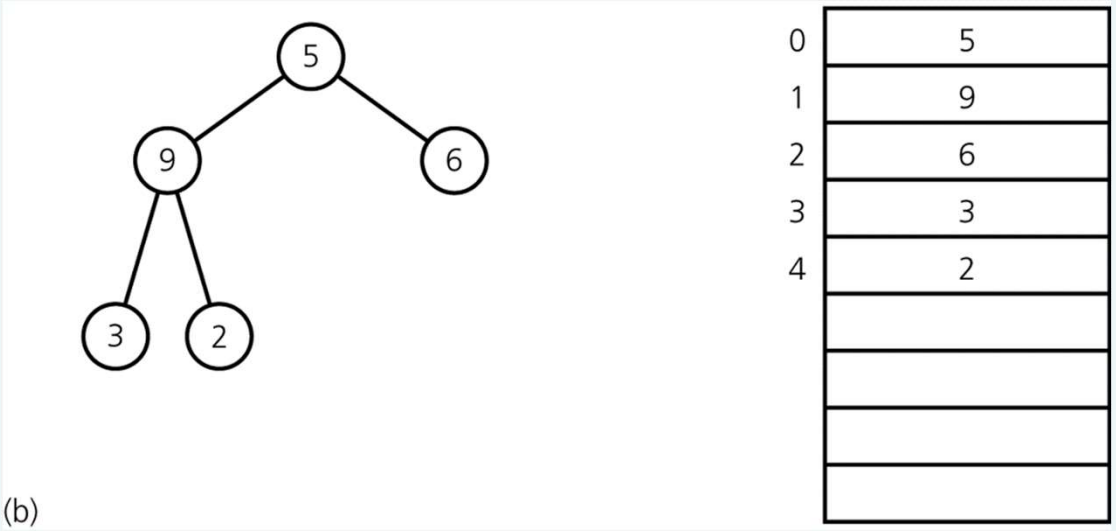
- Step 1: Return the item in the root
  - Results in disjoint heaps



a) Disjoint heaps

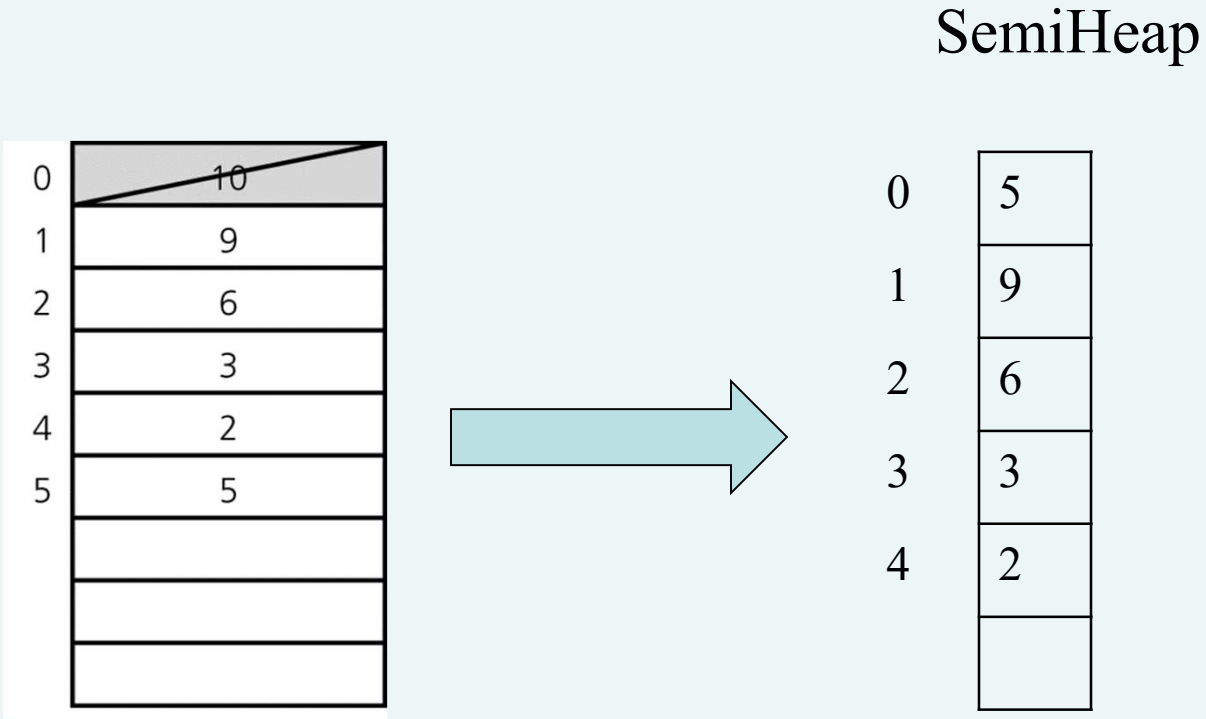
Heaps: `heapDelete`

- Step 2: Copy the item from the last node into the root
  - Results in a semiheap



b) a semiheap

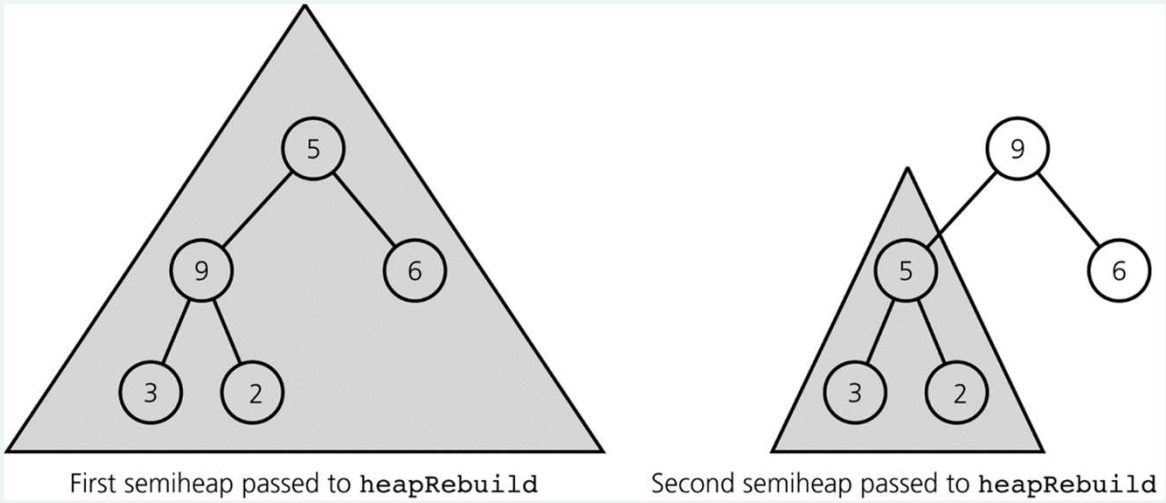
Heaps: `heapDelete`



Heaps: `heapDelete`

- Step 3: Transform the semiheap back into a heap
  - Performed by the recursive algorithm `heapRebuild`

5	9
9	5
6	6
3	3
2	2



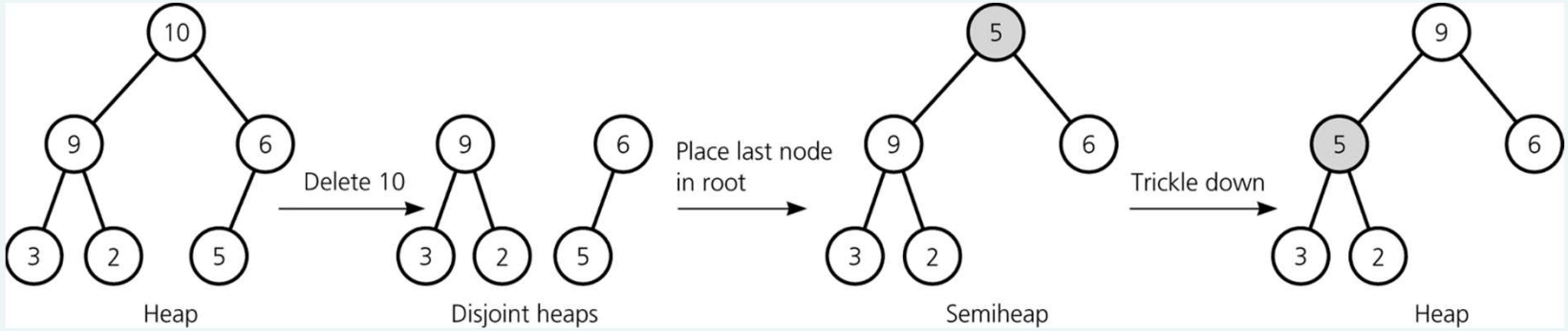
First semiheap passed to **heapRebuild**

Second semiheap passed to **heapRebuild**

Recursive calls to ***heapRebuild***

Heaps: `heapDelete`

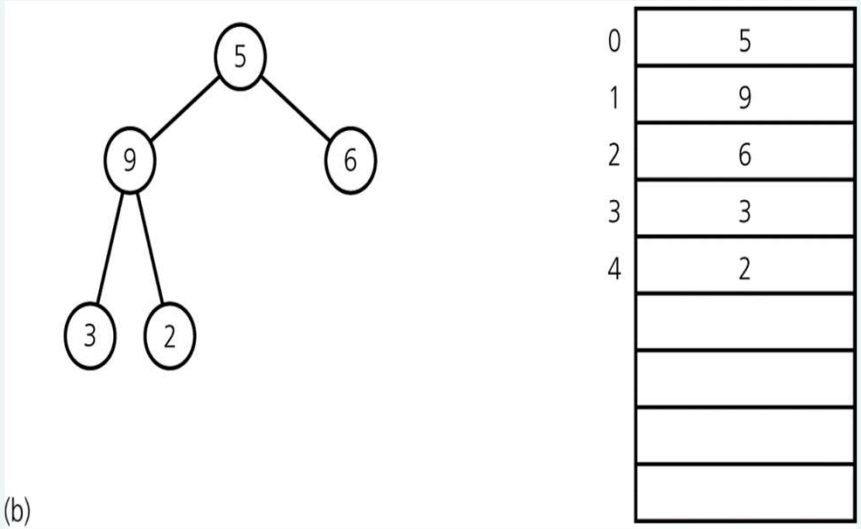
- Efficiency
  - `heapDelete` is O(log n)

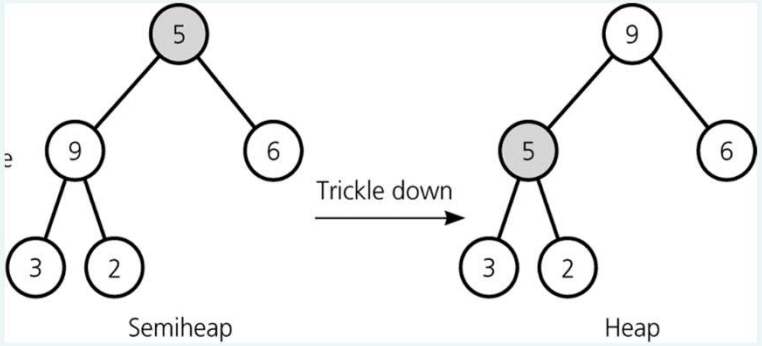


Deletion from a heap

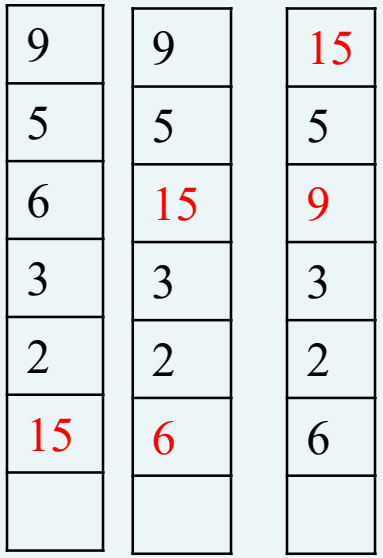
Heaps: `heapRebuild`

```
protected void heapRebuild(int root) {
      // if the root is not a leaf and the root's search key
      // is less than the larger of the search keys in the
      // root's children

      // index of root's left child, if any
      int child = 2 * root + 1;
      if (child < list.size() ) {
         // root is not a leaf, so it has a left child
         // index of root's right child, if any
         int rightChild = child + 1;

         // if root has a right child, find larger child
         if (( rightChild < list.size()) &&
              list.get(rightChild).compareTo(list.get(child)) > 0) {
            child = rightChild; // index of larger child
         }

         // if the root's value is smaller than the value in the
larger
         // child, swap values
         if (list.get(root).compareTo(list.get(child)) < 0) {
            T temp = list.get(root);
            list.set(root,list.get(child));
            list.set(child, temp);
            // transform the new subtree into a heap
            heapRebuild(child);
         } // end if
      } // end if
      // if root is a leaf, do nothing
   } // end rebuild
```

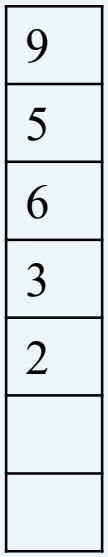


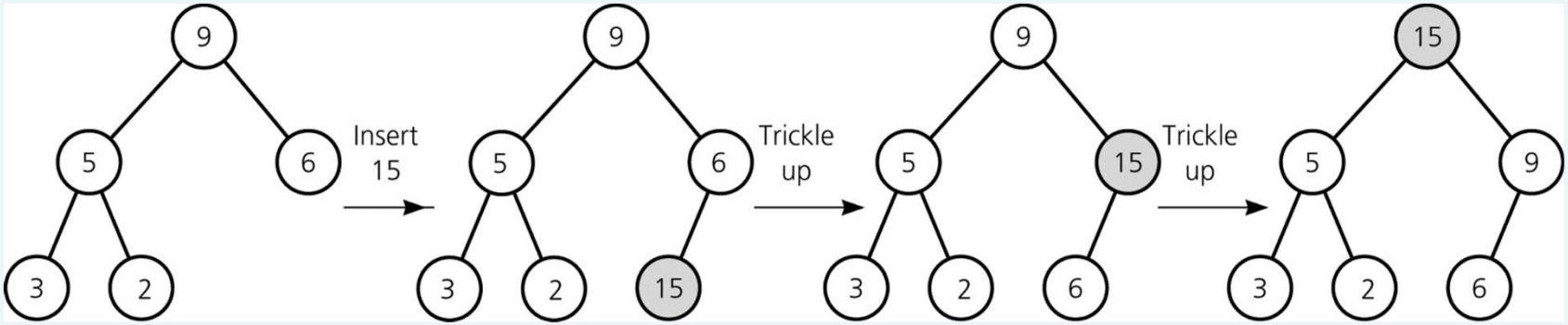


Heaps: `heapInsert`



- Strategy
  - Insert `newItem` into the bottom of the tree
  - Trickle new item up to appropriate spot in the tree
- Efficiency: O(log n)
- `Heap` class
  - Represents an array-based implementation of the ADT heap



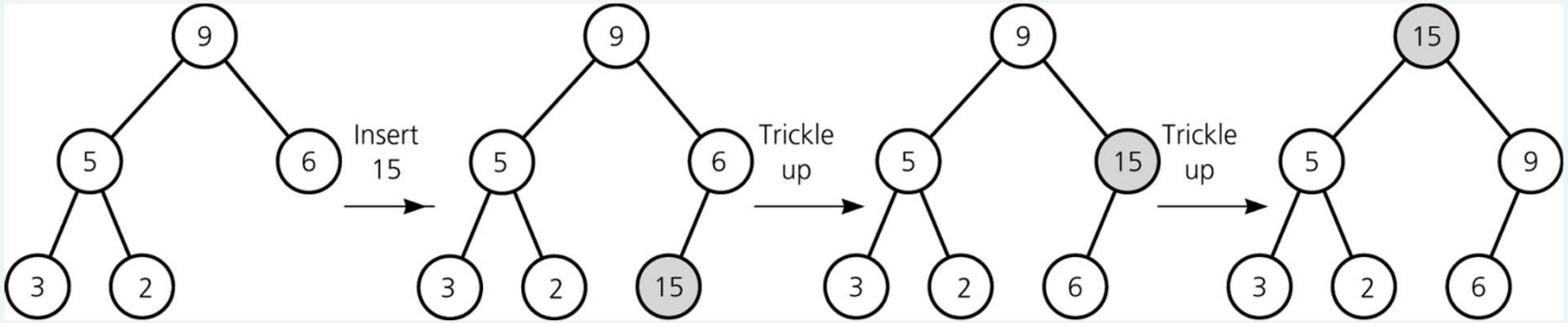


Insertion into a heap

Heaps: heapInsert

```
public void add(T newItem) {
    // adds an item into a heap
    // Precondition: newItem is the item to be added
    // Postcondition: newItem is added in proper
    //location in the heap

    // trickle new item up to its proper position
       list.add(newItem); // Append to the heap
    // The index of the last node
       int currentIndex = list.size() - 1;
       int parentIndex = (currentIndex - 1) / 2;
       while (currentIndex >= 0 &&
list.get(currentIndex).compareTo(list.get(parentInd
ex)) > 0 ) {

    // Swap list[currentIndex] and list[parentIndex]

  T temp = list.get(currentIndex);
  list.set(currentIndex, list.get(parentIndex));
   list.set(parentIndex, temp);
  currentIndex = parentIndex;
  parentIndex = (currentIndex - 1) / 2;
       } // end while
```

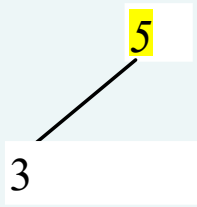
9	9	9	15
5	5	5	5
6	6	15	9
3	3	3	3
2	2	2	2
	15	6	6



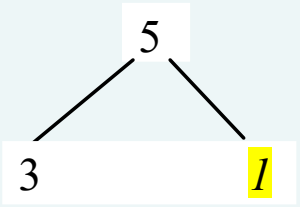
Adding Elements to the Heap

Adding 3, 5, 1, 19, 11, and 22 to a heap, initially empty

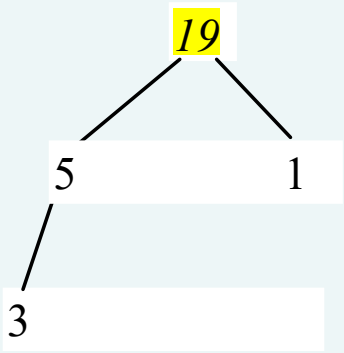
(a) After adding 3



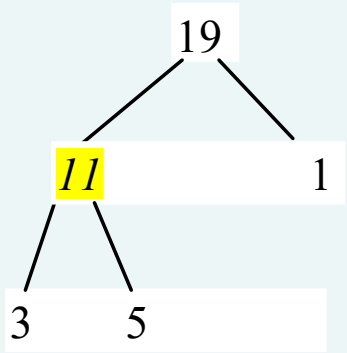
(b) After adding 5



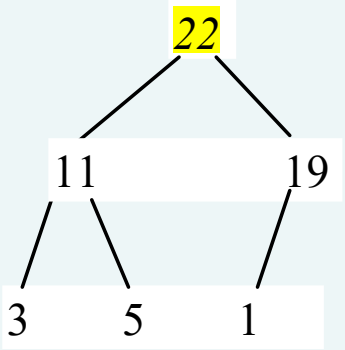
(c) After adding 1



(d) After adding 19



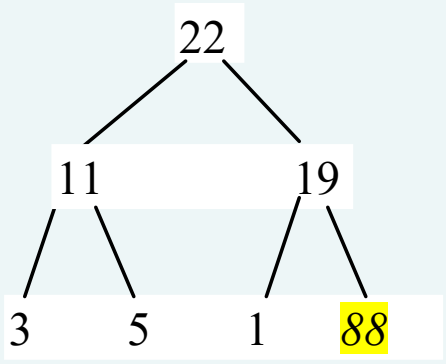
(e) After adding 11



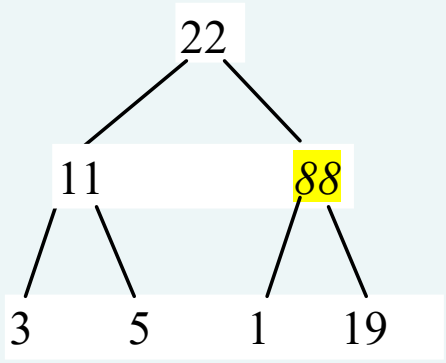
(f) After adding 22

Rebuild the heap after adding a new node

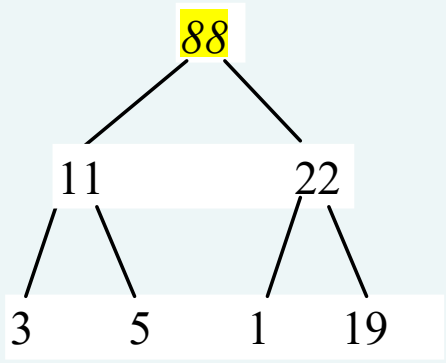
Adding 88 to the heap



(a) Add 88 to a heap



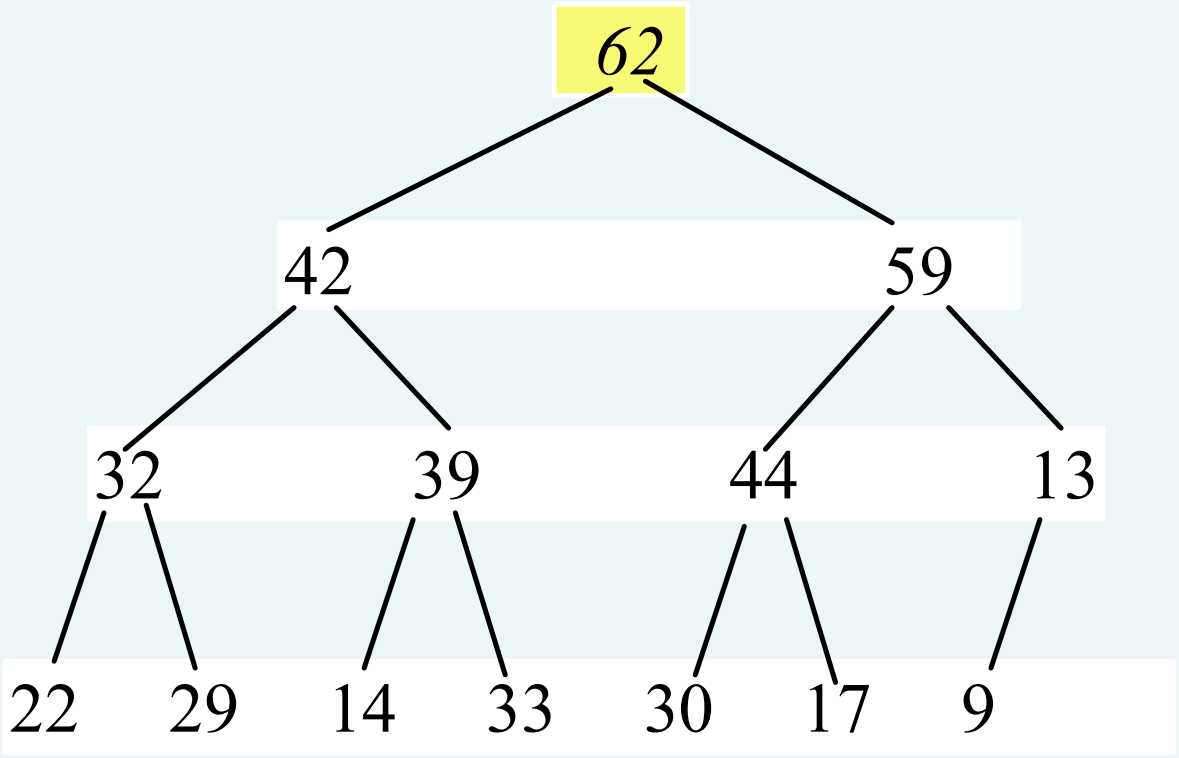
(b) After swapping 88 with 19



(b) After swapping 88 with 22

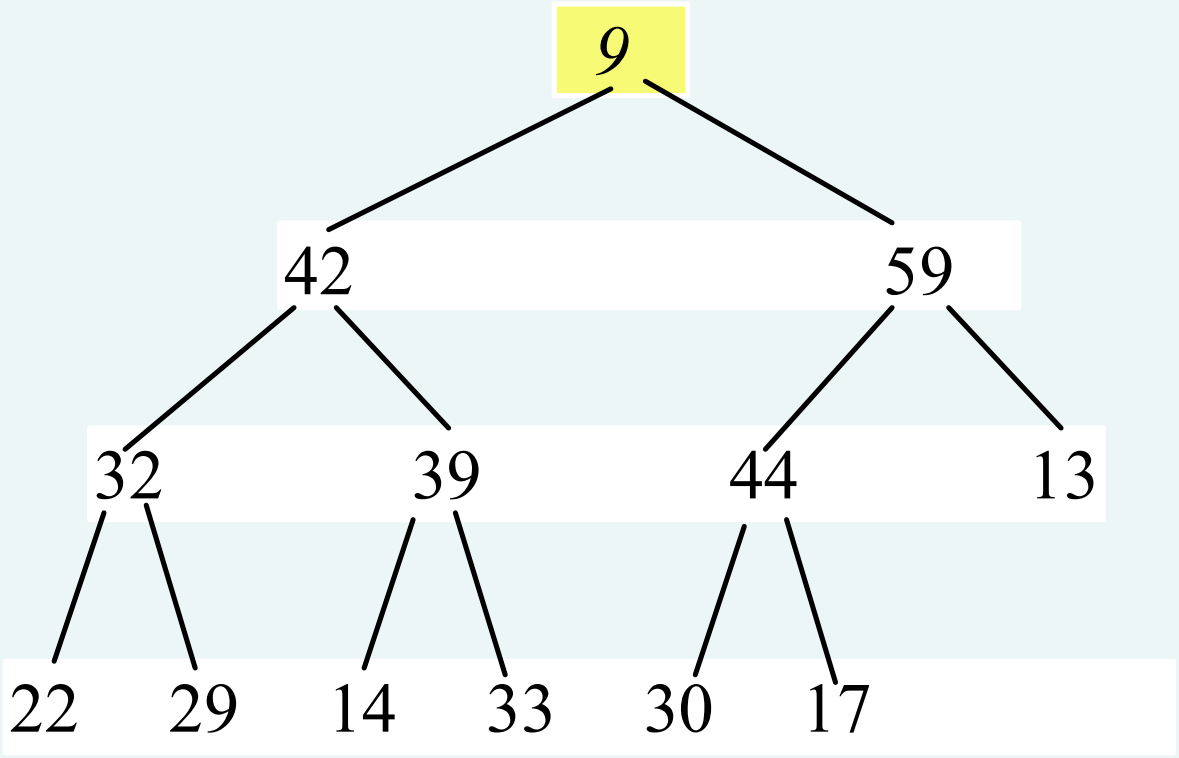
Removing the Root and Rebuild the Tree

Removing root 62 from the heap



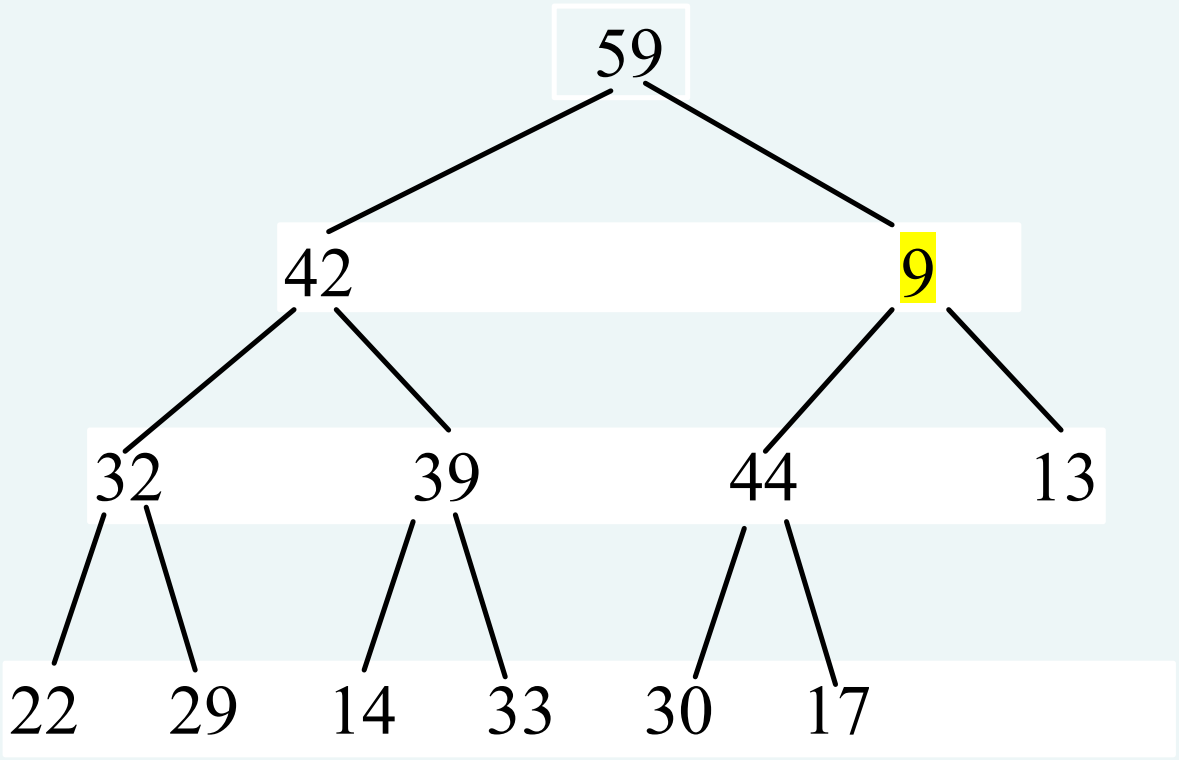
Removing the Root and Rebuild the Tree

Move 9 to root



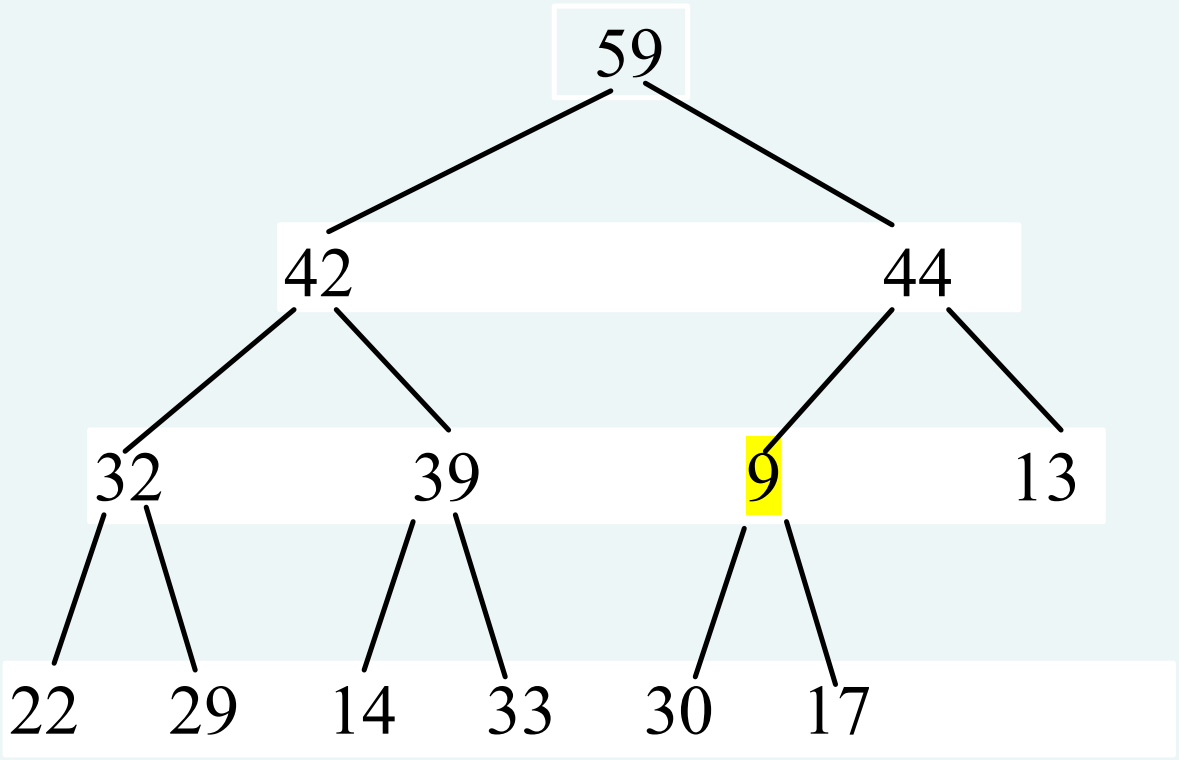
Removing the Root and Rebuild the Tree

Swap 9 with 59



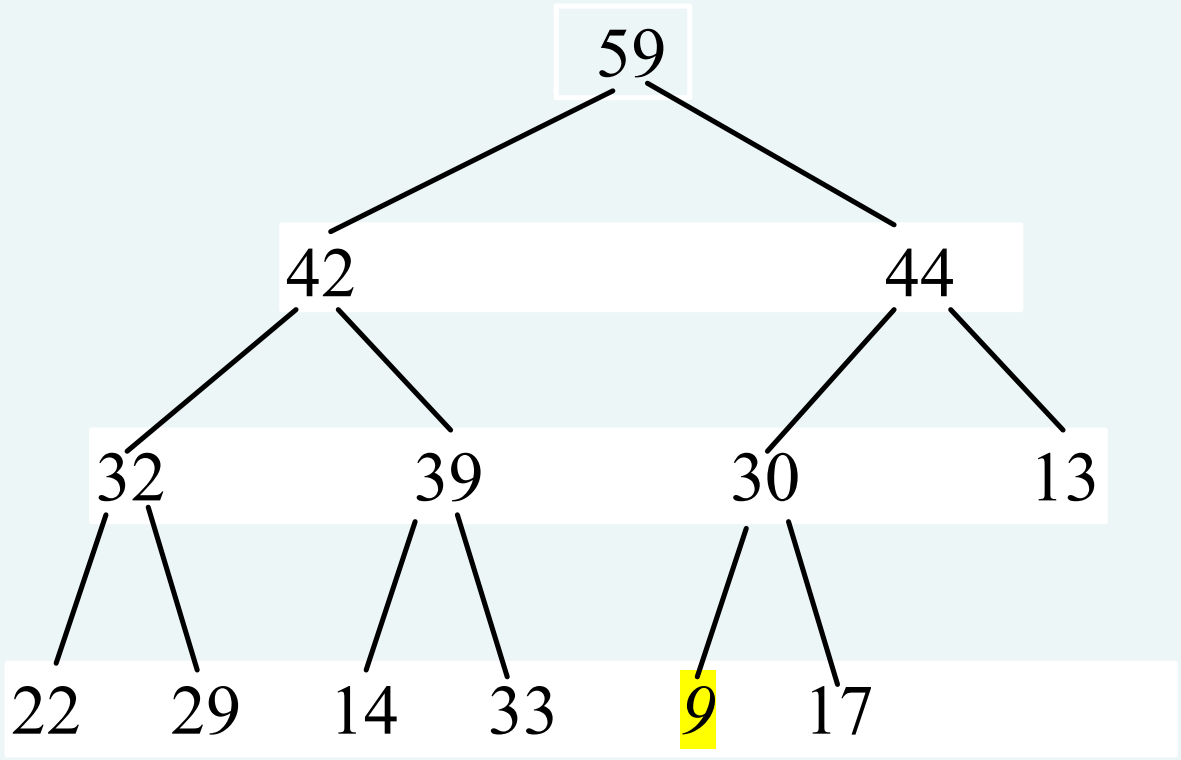
Removing the Root and Rebuild the Tree

Swap 9 with 44



Removing the Root and Rebuild the Tree

Swap 9 with 30



The Heap Class

- Pseudocode for the operations of the ADT heap

```
Heap()
// Creates an empty heap.

isEmpty()
// Determines whether a heap is empty.

add(newItem)
// Inserts newItem into a heap.

remove()
// Retrieves and then deletes a heap's root
// item. This item has the largest search key.
```

Heap

A Heap Implementation of the ADT Priority Queue

- Priority-queue operations and heap operations are analogous
  - The priority value in a priority-queue corresponds to a heap item's search key
- `PriorityQueue` class
  - Has an instance of the `Heap` class as its data field

Heapsort

- Strategy
  - Transforms the array into a heap
  - Removes from the heap element by element and add to your array.
- Efficiency
  - Compared to mergesort
    - Both heapsort and mergesort are O(n * log n) in both the worst and average cases
    - Advantage over mergesort
      - Heapsort does not require a second array
  - Compared to quicksort
    - Quicksort is the preferred sorting method

Heap Sort

HeapSort

Java Class Library: The Class `PriorityQueue`

Basic constructors and methods `PriorityQueue`

`add`
`offer`
`remove`
`poll`
`element`
`peek`
`isEmpty, clear, size`