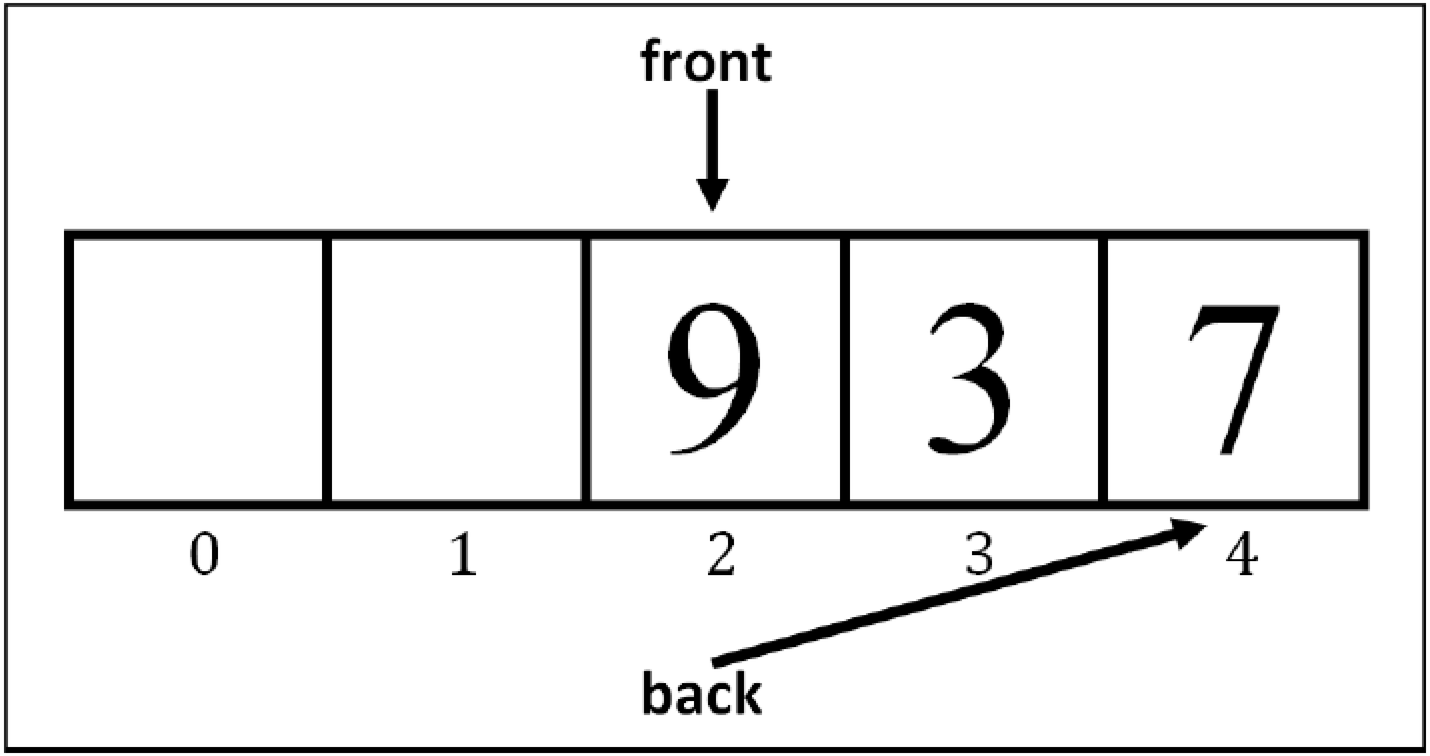
Admin

- Assignment 5 will be posted this week.
  - No late submissions
  - Code must compile
  - Interdependent code will be tested as such

CSC 115 Midterm 2 Review

Part B



Part C

```java
public static int sumSquares(int start, int end){
  // base case
  if (start == end){
     return start * start;
  }
  // recursive case
  else{
     return start * start + sumSquares(start + 1, end);
  }
}
```

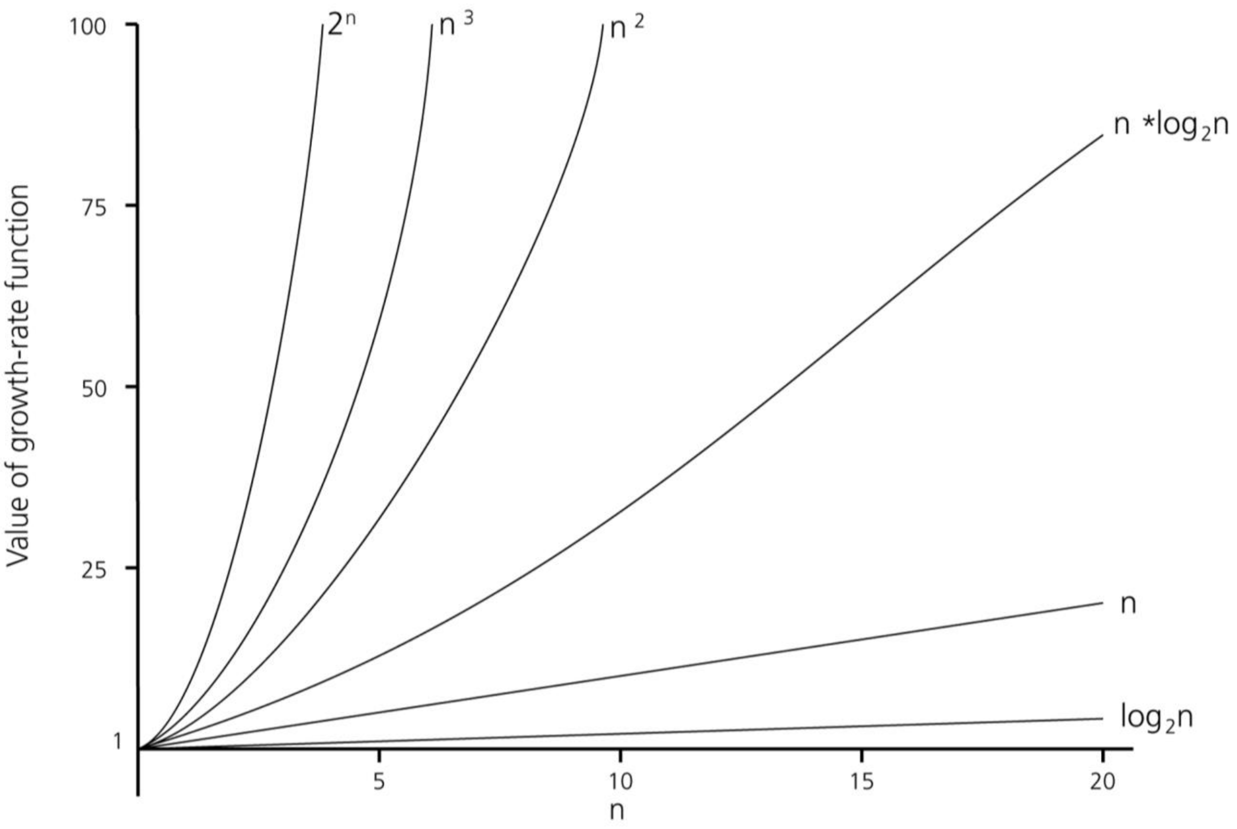
Tweak Part C

```java
public static int sumSquares(int start, int end){
  // base case
  if (start > end){
     return 0;
  }
  // recursive case
  else{
     return start * start + sumSquares(start + 1, end);
  }
}
```

Part C

```java
public static boolean palindrome(String input){

    Stack<Character> stack = new Stack<Character>();
    int half = input.length()/2;
    int beginPoppingHere = half;

    for (int i = 0; i < half; i++){
        stack.push(input.charAt(i));
    }
    if (input.length()% 2 == 1)beginPoppingHere = half + 1;

    for (int i = beginPoppingHere; i < input.length();i++){
        if (input.charAt(i)!= stack.pop()) return false;
    }
    return stack.isEmpty();
}
```

Big O notation (graphical)



7

Big O notation (tabular)

Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

8

A Comparison of Sorting Algorithms

	Worst case	Average case
Selection sort		
Bubble sort		
Insertion sort		
Mergesort		
Quicksort		
Radix sort		
Treesort		
Heapsort		

Figure 10-22

Approximate growth rates of time required for eight sorting algorithms

A Comparison of Sorting Algorithms

	Worst case	Average case
Selection sort	n^2	n^2
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Mergesort	$n * \log n$	$n * \log n$
Quicksort	n^2	$n * \log n$
Radix sort	n	n
Treesort	n^2	$n * \log n$
Heapsort	$n * \log n$	$n * \log n$

Figure 10-22

Approximate growth rates of time required for eight sorting algorithms

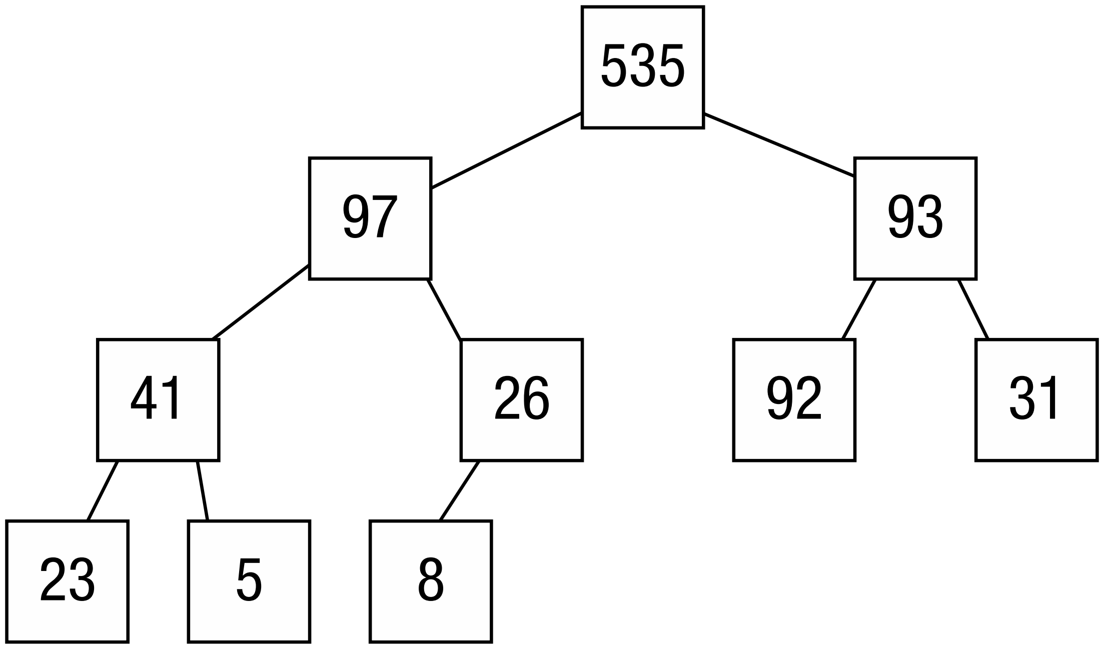
Summary

- Worst-case and average-case analyses
  - Worst-case analysis considers the maximum amount of work an algorithm requires on a problem of a given size
  - Average-case analysis considers the expected amount of work an algorithm requires on a problem of a given size
- Order-of-magnitude analysis can be used to choose an implementation for an abstract data type
- Selection sort, bubble sort, and insertion sort are all $O(n^2)$ algorithms
- Quicksort and mergesort are two very efficient sorting algorithms

Heapsort

- Recall:
  - A complete binary tree is one which may be stored in an array
- A heap is similar to a binary search tree, but with two big differences:
  - BSTs are sorted, while heaps are ordered more weakly
  - BSTs can be in different shapes, but heaps are always complete binary trees
- Ordering invariant for a maxheap:
  - Value of child nodes always less than value of parent node
  - However, there is no ordering amongst siblings!

A maxheap



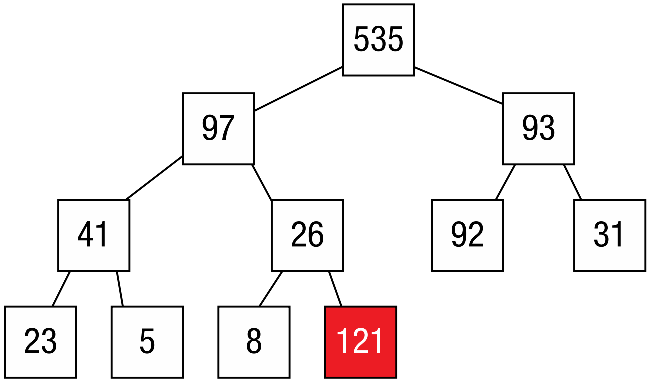
Array representation of the maxheap

535	97	93	41	26	92	31	23	5	8

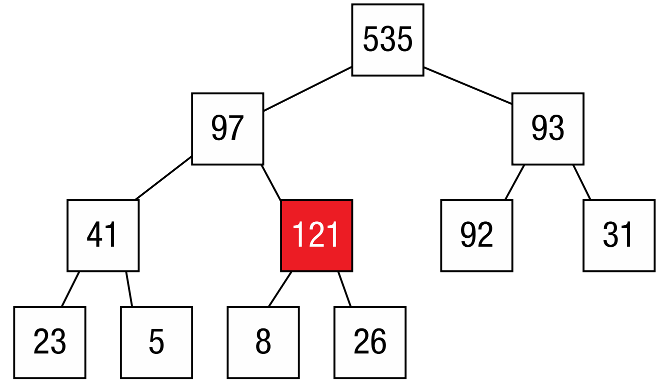
Heaprebuild (upwards)

- Building a heap is a recursive process
  - A new value is added to the array
  - Must re-establish maxheap property
  - To do so, determine how high the new value must float up towards the heap's root node
- One approach: **trickle up**
- Example: insert 121 into the previous heap

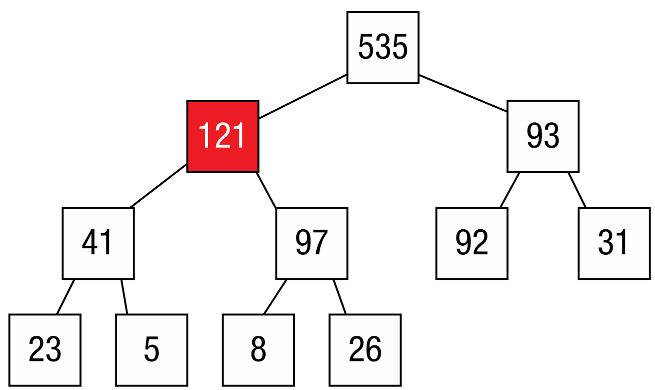
535	97	93	41	26	92	31	23	5	8	121



535	97	93	41	121	92	31	23	5	8	26

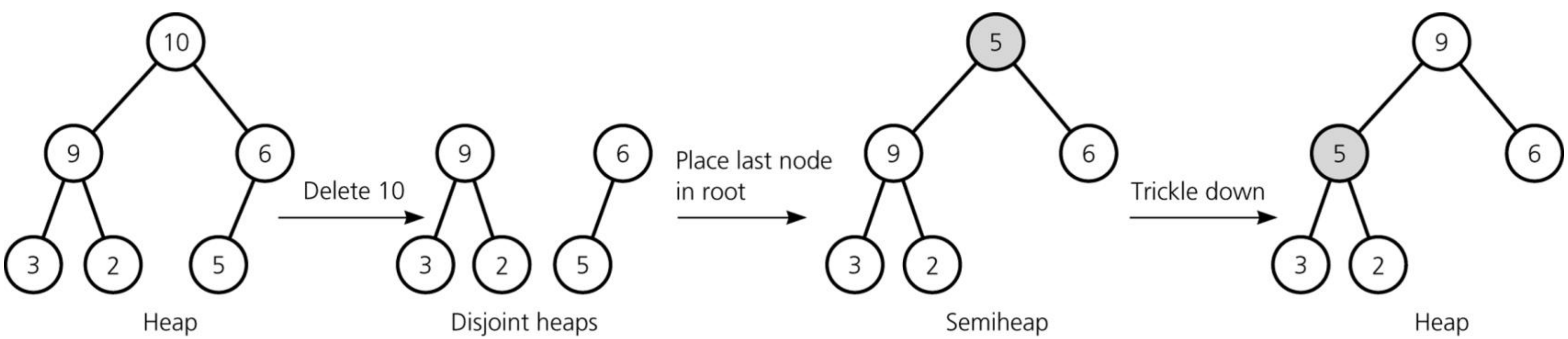


535	121	93	41	97	92	31	23	5	8	26



Heaprebuild (downwards)

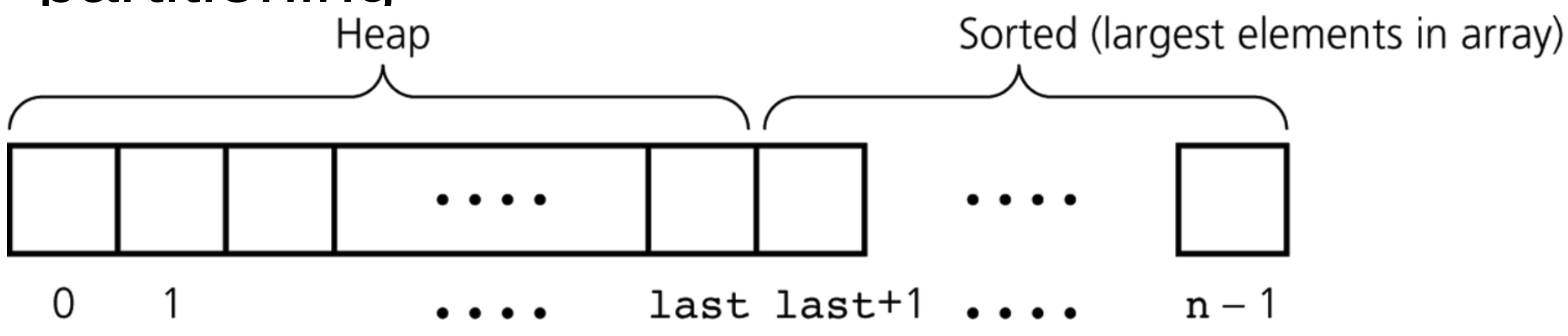
- Another approach: **trickle down**
  - Useful when deleting rather than inserting
  - Assumes two heaps formed at start of operation
  - Determine what value from the each of the smaller heaps should be root of the combined heap



```
heapRebuild(array, root, n)

    child = 2 * root + 1

    if ( child < n )
        rightChild = child + 1
        if (rightChild < n && array[child] < array[rightChild])
            child = rightChild

        if (array[root] < array[child])
             swap array[root] with array[child]
             heapRebuild(array, child, n)
```

Heapsort

- Intuition
  - Build an initial heap using the original array as storage
  - Once heap is built, move largest item in heap to beginning of a sorted region
- The second step performs a kind of partitioning



```
// Depends upon having heapRebuild

heapSort(array, n)

    for (i = n-1 down to 0)
        heapRebuild(array, i, n)

    last = n - 1

    for (step = 1 through n)
        swap array[0] and array[last]
        last--
        heapRebuild(array, 0, last)
```

Heapsort: analysis

- Number of moves?
- Number of compares?
- $O(n \log_2 n)$

Treesort

- This uses a BST
- Each item to be sorted is inserted into the BST
- Once all items are inserted, we perform an in-order traversal
- Upside:
  - Can get $O(n \log n)$ efficiency in average case
- Downside:
  - Worst case is $O(n^2)$

BST operation efficiency

- Theorem:
  - A full binary tree of height **$h \geq 0$** has **2^h-1 nodes**
- Theorem:
  - The **maximum number of nodes** that a binary tree of height h can have is **2^h-1**
- Theorem:
  - The **minimum height of a binary tree** with n nodes is **ceil[$\log_2(n+1)$]**

BST operation efficiency

- Given these theorems:
  – We can say something about the worst- and best-case heights of a particular binary search tree
  – And given that retrieve, insert and delete describes paths from the root (at worst) some leaf…
  – … this means our efficiencies will depend upon the length of this path (which is also the tree height)
- Operations (i.e., number of comparisons)
  – Retrieval: average O(log n); worst O(n)
  – Insertion: average O(log n); worst O(n)
  – Deletion: average O(log n); worst O(n)
- Note: Traversals will always be O(n) as one comparison is needed for visiting every node

Tables & Priority Queues

- Reading: Chapter 12

The ADT Table

- The ADT table, or dictionary
  - Uses a search key to identify its items
  - Its items are records that contain several pieces of data

City	Country	Population
Athens	Greece	2,500,000
Barcelona	Spain	1,800,000
Cairo	Egypt	9,500,000
London	England	9,400,000
New York	U.S.A.	7,300,000
Paris	France	2,200,000
Rome	Italy	2,800,000
Toronto	Canada	3,200,000
Venice	Italy	300,000

Figure 12-1

An ordinary table of cities

The ADT Table

- Operations of the ADT table
  - Create an empty table
  - Determine whether a table is empty
  - Determine the number of items in a table
  - Insert a new item into a table
  - Delete the item with a given search key from a table
  - Retrieve the item with a given search key from a table
  - Traverse the items in a table in sorted search-key order

The ADT Table

- Pseudocode for the operations of the ADT table

```
createTable()
// Creates an empty table.

tableIsEmpty()
// Determines whether a table is empty.

tableLength()
// Determines the number of items in a table.

tableInsert(newItem) throws TableException
// Inserts newItem into a table whose items have
// distinct search keys that differ from newItem's
// search key. Throws TableException if the
// insertion is not successful
```

The ADT Table

- Pseudocode for the operations of the ADT table (Continued)

```
tableDelete(searchKey)
// Deletes from a table the item whose search key
// equals searchKey. Returns false if no such item
// exists. Returns true if the deletion was
// successful.

tableRetrieve(searchKey)
// Returns the item in a table whose search key
// equals searchKey. Returns null if no such item
// exists.

tableTraverse()
// Traverses a table in sorted search-key order.
```

The ADT Table

- Value of the search key for an item must remain the same as long as the item is stored in the table
- `KeyedItem` class
  - Contains an item’s search key and a method for accessing the search-key data field
  - Prevents the search-key value from being modified once an item is created
- `TableInterface` interface
  - Defines the table operations

A Binary Search Tree Implementation of the ADT Table

- `TableBSTBased` class
  - Represents a nonlinear reference-based implementation of the ADT table
  - Uses a binary search tree to represent the items in the ADT table
    - Reuses the class `BinarySearchTree`

The ADT Priority Queue: A Variation of the ADT Table

- The ADT priority queue
  - Orders its items by a priority value
  - The first item removed is the one having the highest priority value
- Operations of the ADT priority queue
  - Create an empty priority queue
  - Determine whether a priority queue is empty
  - Insert a new item into a priority queue
  - Retrieve and then delete the item in a priority queue with the highest priority value

The ADT Priority Queue: A Variation of the ADT Table

- Pseudocode for the operations of the ADT priority queue

```
createPQueue()
// Creates an empty priority queue.

pqIsEmpty()
// Determines whether a priority queue is
// empty.
```

The ADT Priority Queue: A Variation of the ADT Table

- Pseudocode for the operations of the ADT priority queue (Continued)

```
pqInsert(newItem) throws PQueueException
// Inserts newItem into a priority queue.
// Throws PQueueException if priority queue is
// full.

pqDelete()
// Retrieves and then deletes the item in a
// priority queue with the highest priority
// value.
```

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 Priority Queue: A Variation of the ADT Table

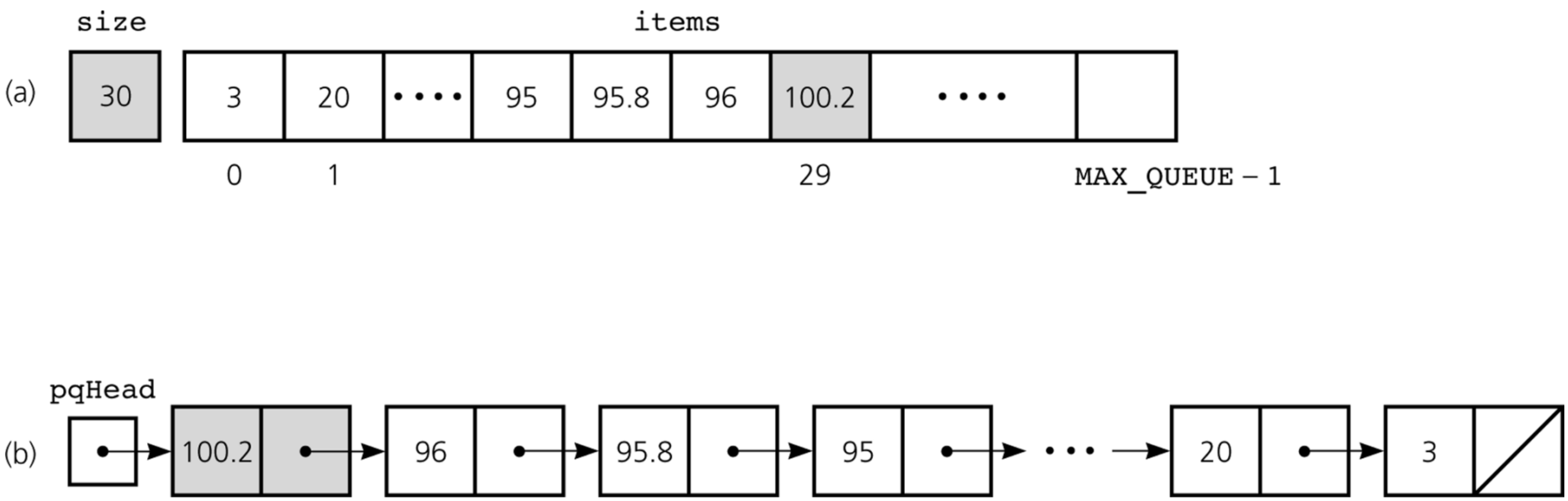


Figure 12-9a and 12-9b

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

The ADT Priority Queue

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

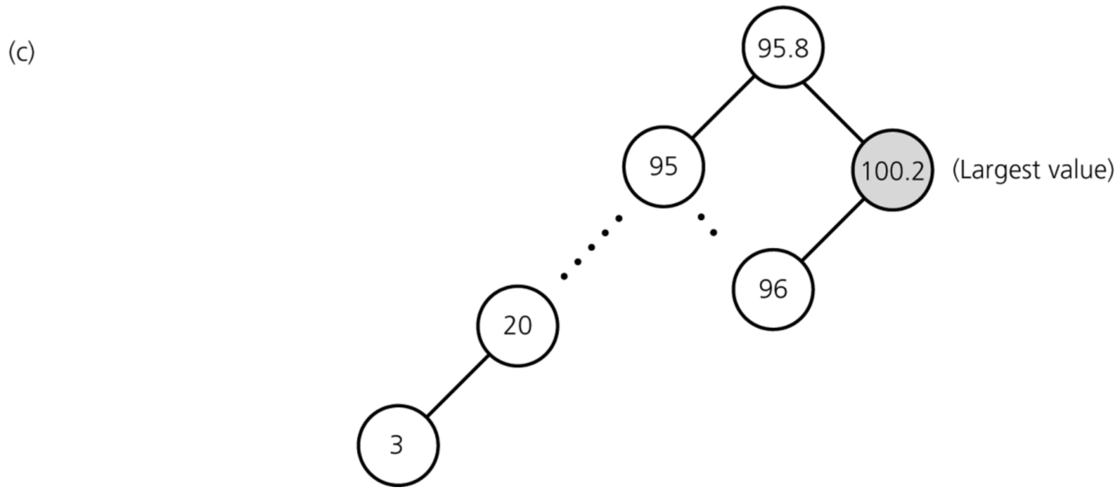
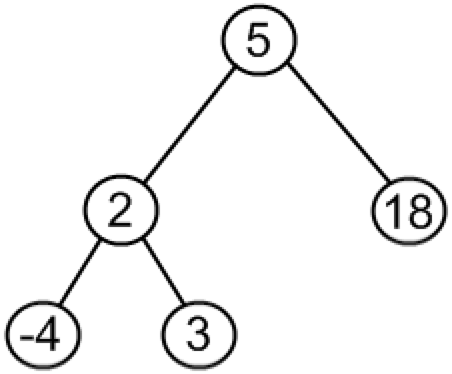


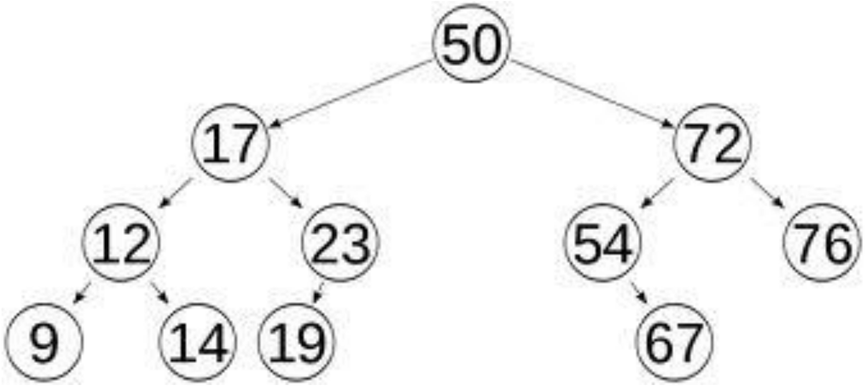
Figure 12-9c

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

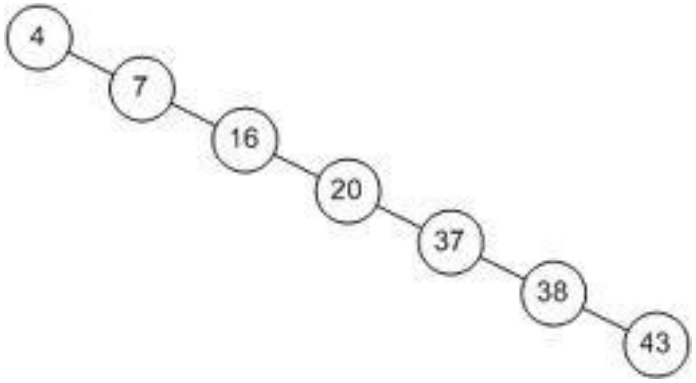
First:
Remember Binary Search Trees

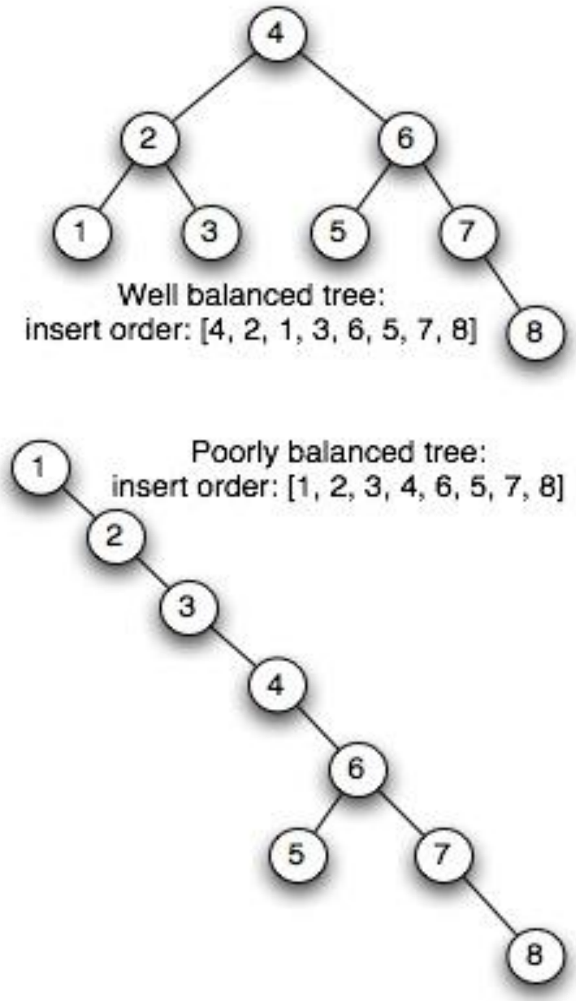
- For each node n:
  - n's value is greater than all values in its left subtree T_L
  - n's value is less than all values in its right subtree T_R
  - Both T_L and T_R are binary search trees





Problem with Binary Search Trees





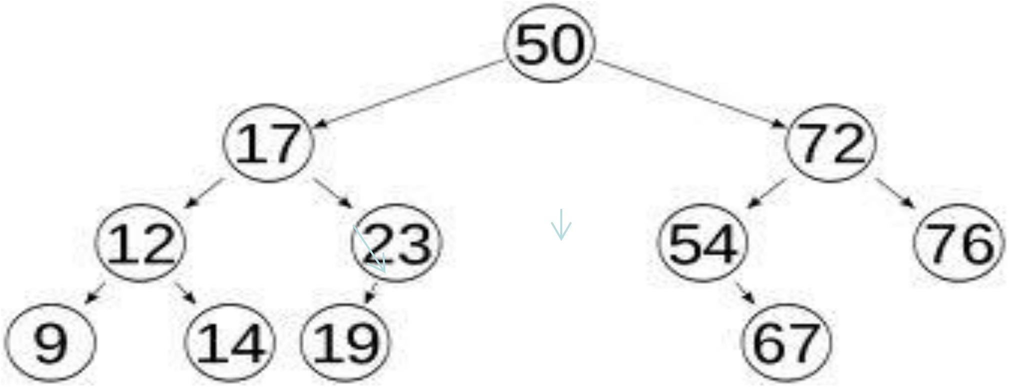
Heaps

- 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

Binary Search Tree



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

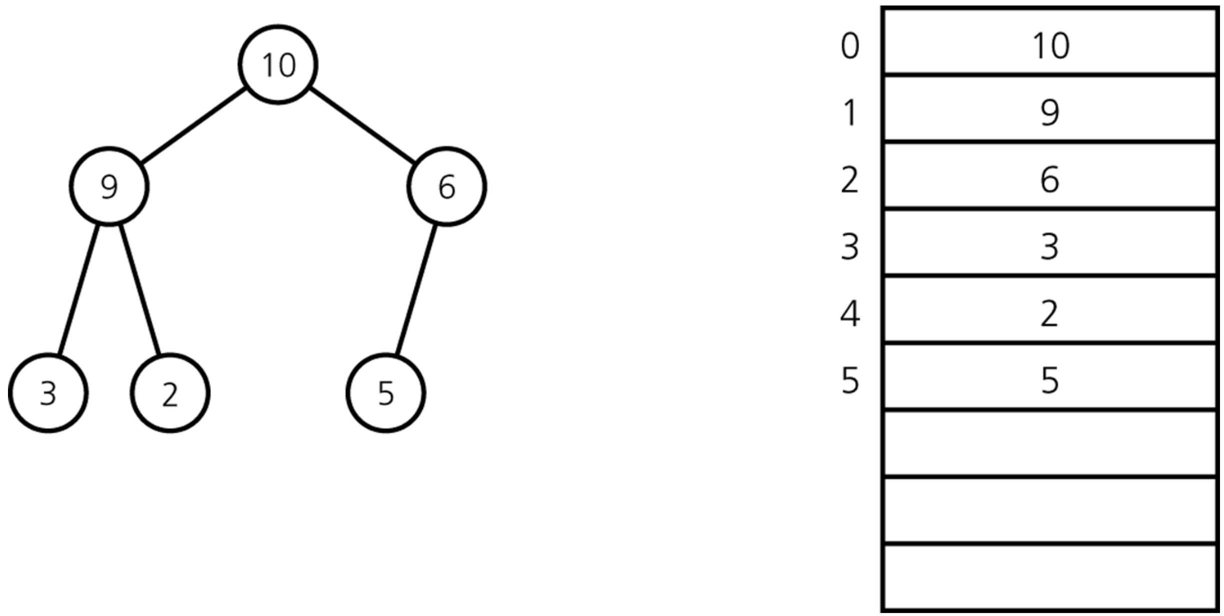
```
createHeap()
// Creates an empty heap.

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

heapInsert(newItem) throws HeapException
// Inserts newItem into a heap. Throws
// HeapException if heap is full.

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

Heaps: Array-based

- Data fields
  - `items`: an array of heap items
  - `size`: an integer equal to the number of items in the heap

Figure 12-11

A heap with its array representation



Heaps: `heapDelete`

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

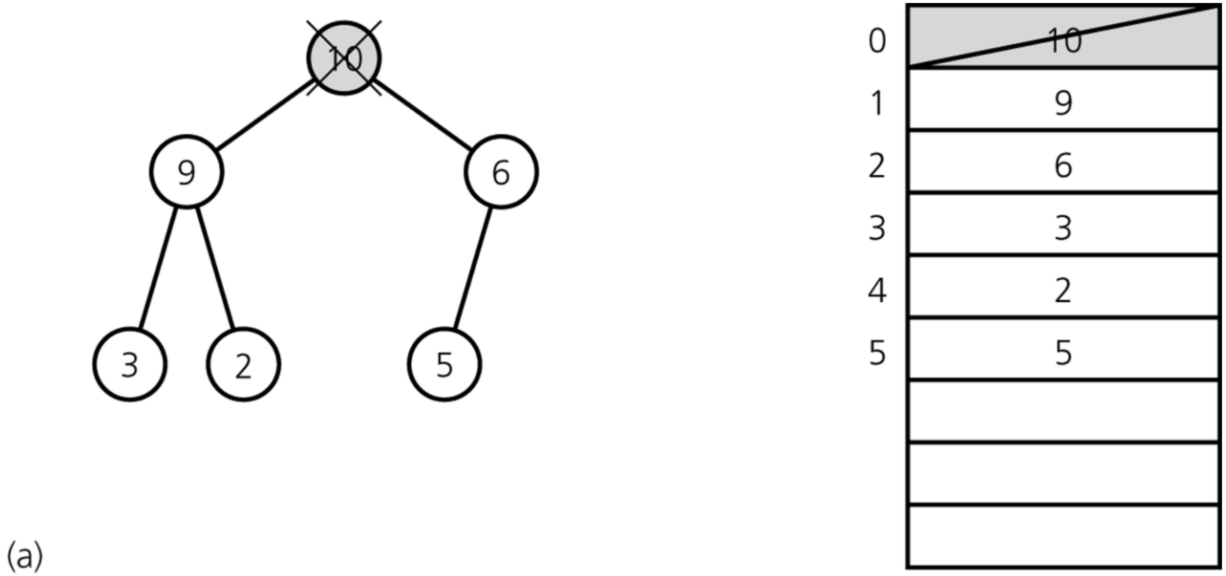
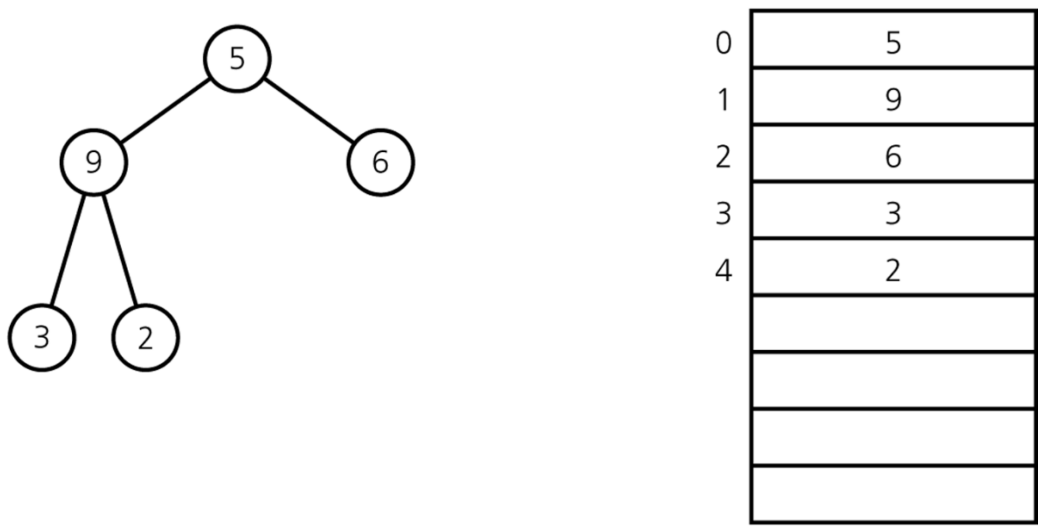


Figure 12-12a

a) Disjoint heaps

Heaps: `heapDelete`

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



(b)

Figure 12-12b

b) a semiheap

Heaps: `heapDelete`

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

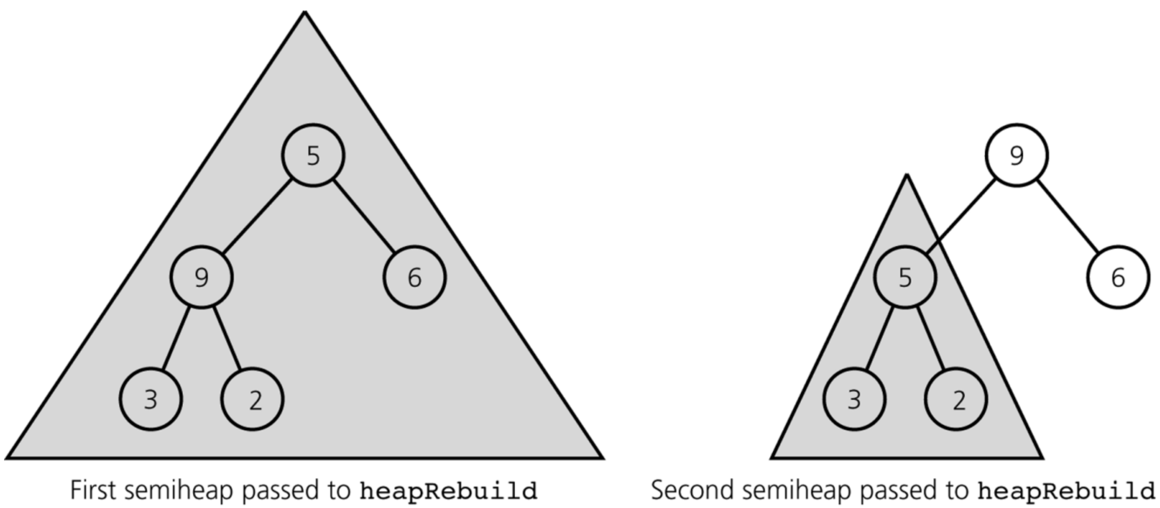


Figure 12-14

Recursive calls to ***heapRebuild***

Heaps: `heapDelete`

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

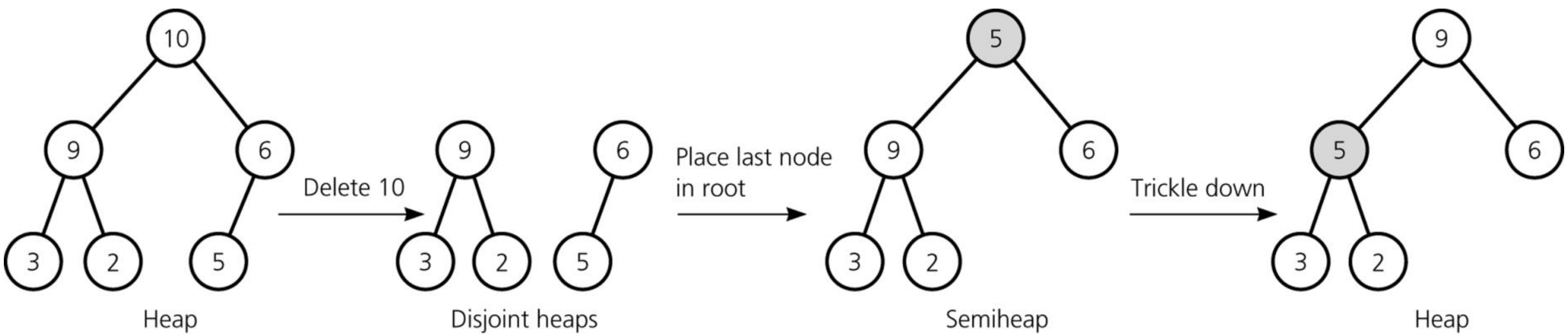
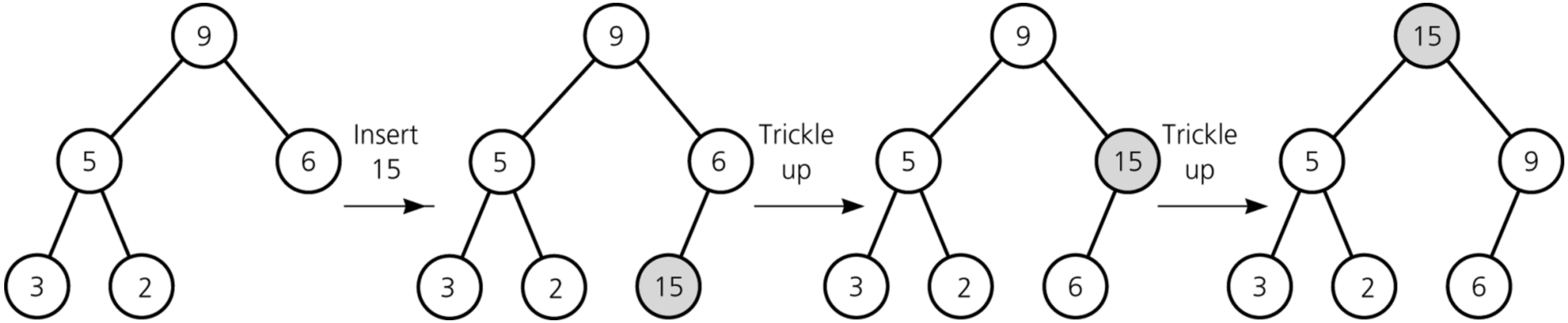


Figure 12-13

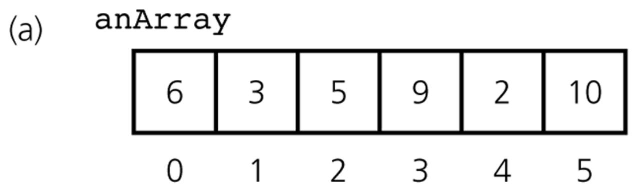
Deletion from a heap

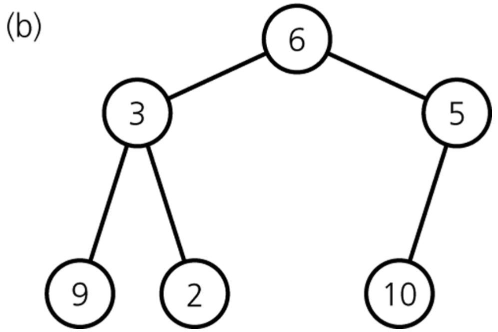
Heaps: `heapInsert`

- 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



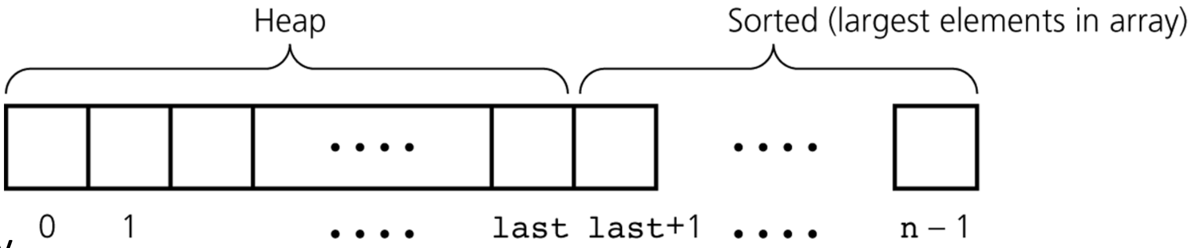
Heapsort





a) The initial contents of ***anArray;***

b) b) ***anArray's*** corresponding binary tree



Heapsort partitions an array into two regions

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

A Heap Implementation of the ADT Priority Queue

- A heap implementation of a priority queue
  - Disadvantage
    - Requires the knowledge of the priority queue’s maximum size
  - Advantage
    - A heap is always balanced
- Finite, distinct priority values
  - A heap of queues
    - Useful when a finite number of distinct priority values are used, which can result in many items having the same priority value

Summary

- The ADT table supports value-oriented operations
- The linear implementations (array based and reference based) of a table are adequate only in limited situations or for certain operations
- A nonlinear reference-based (binary search tree) implementation of the ADT table provides the best aspects of the two linear implementations
- A priority queue, a variation of the ADT table, has operations which allow you to retrieve and remove the item with the largest priority value

Summary

- A heap that uses an array-based representation of a complete binary tree is a good implementation of a priority queue when you know the maximum number of items that will be stored at any one time
- Efficiency
  - Heapsort, like mergesort, has good worst-case and average-case behaviors, but neither algorithms is as good in the average case as quicksort
  - Heapsort has an advantage over mergesort in that it does not require a second array