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

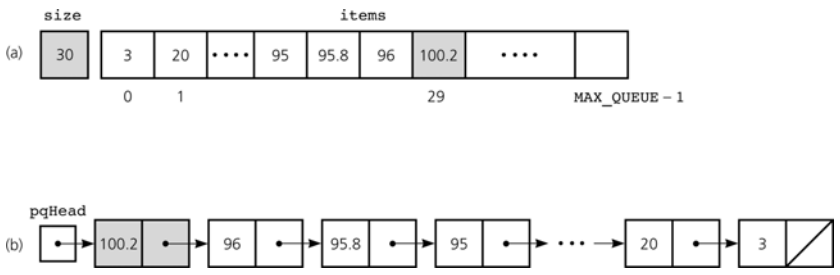


Figure 12-9a and 12-9b

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

The ADT Priority Queue: A Variation of the ADT Table

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

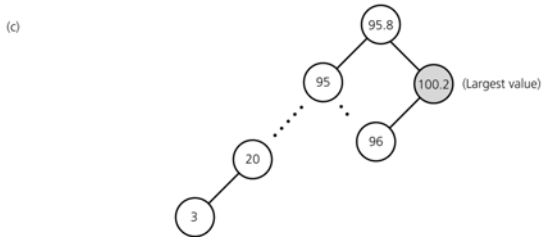


Figure 12-9c

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

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

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: An Array-based Implementation of a Heap

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

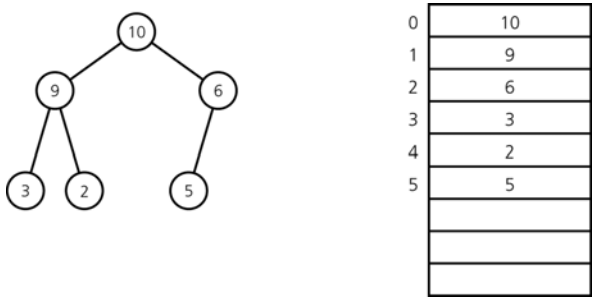


Figure 12-11

A heap with its array representation

Heaps: `heapDelete`

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

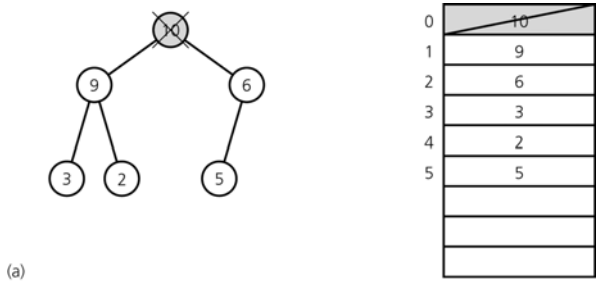


Figure 12-12a

a) Disjoint heaps

Heaps: `heapDelete`

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

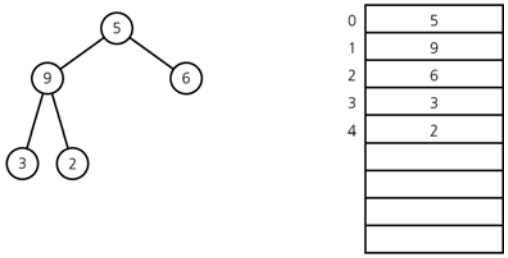


Figure 12-12b

b) a semiheap

Heaps: heapDelete

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

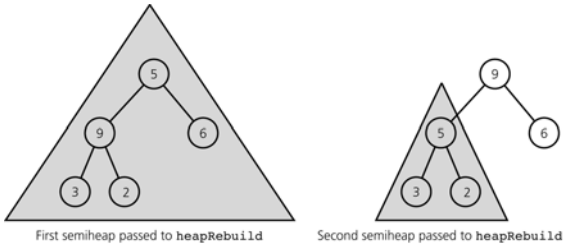


Figure 12-14

Recursive calls to ***heapRebuild***

Heaps: heapDelete

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

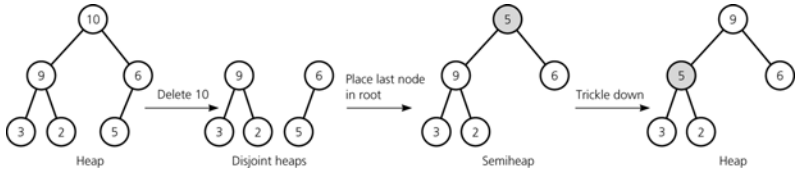


Figure 12-13

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

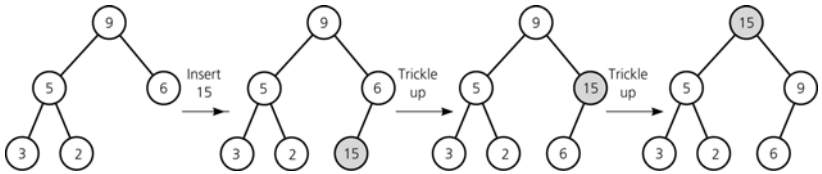


Figure 12-15

Insertion into a 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

A Heap Implementation of the ADT Priority Queue

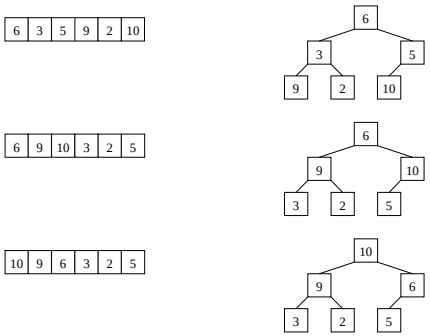
- A heap implementation of a priority queue
  - Disadvantage
    - Requires the knowledge of the priority queue's maximum size (when using the array-based implementation of a heap)
  - Advantage
    - A heap is always balanced
- Finite, distinct priority values (e.g. 1 – 10)
  - A heap of queues
    - Useful when a finite number of distinct priority values are used, which can result in many items having the same priority value

Heapsort

- Strategy
  - Transforms the array into a heap
  - Removes the heap's root (the largest element) by exchanging it with the heap's last element
  - Transforms the resulting semiheap back into a heap
- 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

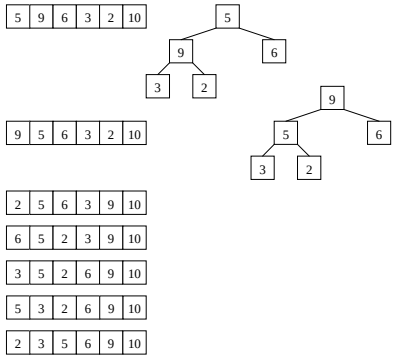
Heapsort

- Convert array to a heap



Heapsort

- Then repeatedly move largest element to the end maintaining the heap.



Heapsort

Figure 12-16
a) The initial contents of ***anArray***; b) ***anArray's*** corresponding binary tree

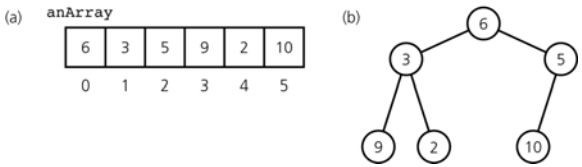
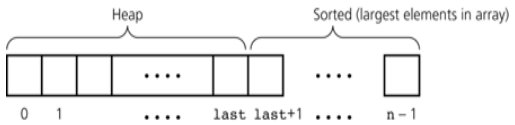


Figure 12-18
Heapsort partitions an array into two regions



Tables and Priority Queues in JFC: The JFC `Map` Interface

- `Map` interface
  - Provides the basis for numerous other implementations of different kinds of maps
- **`public interface`** `Map<K,V>` methods
  - **`void`** `clear()`
  - **`boolean`** `containsKey(Object key)`
  - **`boolean`** `containsValue(Object value)`
  - `Set<Map.Entry<K,V>> entrySet()`
  - `V get(Object key);`

Tables and Priority Queues in JFC: The JFC `Map` Interface

- **`public interface`** `Map<K,V>` methods (continued)
  - **`boolean`** `isEmpty()`
  - `Set<K> keySet()`
  - `V put(K key, V value)`
  - `V remove(Object key)`
  - `Collection<V> values()`

The JFC `PriorityQueue` Class

- `PriorityQueue` class
  - Has a single data-type parameter with ordered elements
  - Relies on the natural ordering of the elements
    - As provided by the `Comparable` interface or a `Comparator` object
  - Elements in queue are ordered in ascending order
- **`public Class`** `PriorityQueue<T>` methods
  - `PriorityQueue(`**`int`**` initialCapacity)`
  - **`boolean`** `add(T o)`
  - **`void`** `clear()`
  - **`boolean`** `contains(Object o)`

The JFC `PriorityQueue` Class

- **`public Class`** `PriorityQueue<T>`
  methods (continued)
  - `T element()`
  - `Iterator<T> iterator()`
  - **`boolean`** `offer(T o)`
  - `T peek()`
  - `T poll()`
  - **`boolean`** `remove(Object o)`
  - **`int`** `size()`

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