Chapter 12

Tables and Priority Queues
CS102 Sections 51 and 52

Marc Smith and Jim Ten Eyck

Spring 2008

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

Selecting an Implementation

- Categories of linear implementations
  - Unsorted, array based
  - Unsorted, referenced based
  - Sorted (by search key), array based
  - Sorted (by search key), reference based

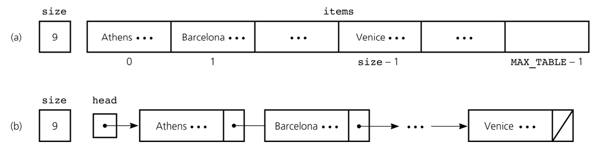
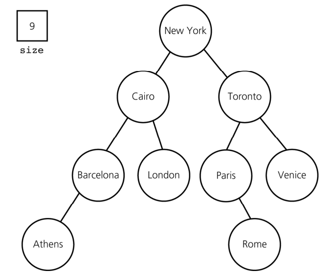


Figure 12-3
The data fields for two sorted linear implementations of the ADT table for the data in Figure 12-1: a) array based; b) reference based

Selecting an Implementation

- A binary search implementation
  - A nonlinear implementation

Figure 12-4
The data fields for a binary search tree implementation of the ADT table for the data in Figure 12-1



Selecting an Implementation

- The binary search tree implementation offers several advantages over linear implementations
- The requirements of a particular application influence the selection of an implementation
  - Questions to be considered about an application before choosing an implementation
    - What operations are needed?
    - How often is each operation required?

Scenario A: Insertion and Traversal in No Particular Order

- An unsorted order in efficient
  - Both array based and reference based `tableInsert` operation is O(1)
- Array based versus reference based
  - If a good estimate of the maximum possible size of the table is not available
    - Reference based implementation is preferred
  - If a good estimate of the maximum possible size of the table is available
    - The choice is mostly a matter of style

Scenario A: Insertion and Traversal in No Particular Order

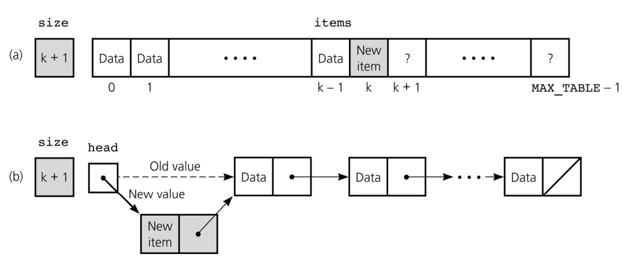


Figure 12-5
Insertion for unsorted linear implementations: a) array based; b) reference based

Scenario A: Insertion and Traversal in No Particular Order

- A binary search tree implementation is not appropriate
  - It does more work than the application requires
    - It orders the table items
  - The insertion operation is O(log n) in the average case

Scenario B: Retrieval

- Binary search
  - An array-based implementation
    - Binary search can be used if the array is sorted
  - A reference-based implementation
    - Binary search can be performed, but is too inefficient to be practical
- A binary search of an array is more efficient than a sequential search of a linked list
  - Binary search of an array
    - Worst case: $O(\log_2 n)$
  - Sequential search of a linked list
    - $O(n)$

Scenario B: Retrieval

- For frequent retrievals
  - If the table's maximum size is known
    - A sorted array-based implementation is appropriate
  - If the table's maximum size is not known
    - A binary search tree implementation is appropriate

Scenario C: Insertion, Deletion, Retrieval, and Traversal in Sorted Order

- Steps performed by both insertion and deletion
  - Step 1: Find the appropriate position in the table
  - Step 2: Insert into (or delete from) this position
- Step 1
  - An array-based implementation is superior than a reference-based implementation
- Step 2
  - A reference-based implementation is superior than an array-based implementation
    - A sorted array-based implementation shifts data during insertions and deletions

Scenario C: Insertion, Deletion, Retrieval, and Traversal in Sorted Order

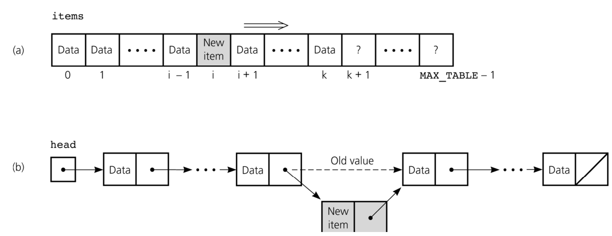


Figure 12-6

Insertion for sorted linear implementations: a) array based; b) reference based

Scenario C: Insertion, Deletion, Retrieval, and Traversal in Sorted Order

- Insertion and deletion operations
  - Both sorted linear implementations are comparable, but neither is suitable
    - `tableInsert` and `tableDelete` operations
      - Sorted array-based implementation is $O(n)$
      - Sorted reference-based implementation is $O(n)$
  - Binary search tree implementation is suitable
    - It combines the best features of the two linear implementations

A Sorted Array-Based Implementation of the ADT Table

- Linear implementations
  - Useful for many applications despite certain difficulties
- A binary search tree implementation
  - In general, can be a better choice than a linear implementation
- A balanced binary search tree implementation
  - Increases the efficiency of the ADT table operations

A Sorted Array-Based Implementation of the ADT Table

	Insertion	Deletion	Retrieval	Traversal
Unsorted array based	O(1)	O(n)	O(n)	O(n)
Unsorted pointer based	O(1)	O(n)	O(n)	O(n)
Sorted array based	O(n)	O(n)	O(log n)	O(n)
Sorted pointer based	O(n)	O(n)	O(n)	O(n)
Binary search tree	O(log n)	O(log n)	O(log n)	O(n)

Figure 12-7
The average-case order of the operations of the ADT table for various implementations

A Sorted Array-Based Implementation of the ADT Table

- Reasons for studying linear implementations
  - Perspective
  - Efficiency
  - Motivation
- `TableArrayBased` class
  - Provides an array-based implementation of the ADT table
  - Implements `TableInterface`

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

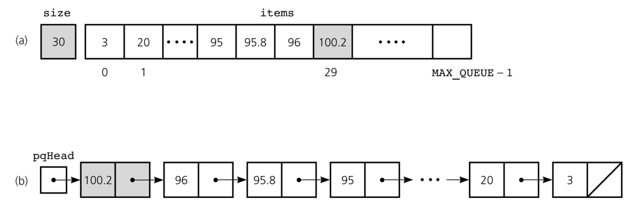


Figure 12-9a and 12-9b
Some implementations of the ADT priority queue: a) array based; b) reference based

The ADT Priority Queue: A Variation of the ADT Table

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

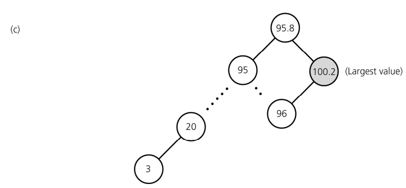


Figure 12-9c
Some implementations of the ADT priority queue: c) binary search tree

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

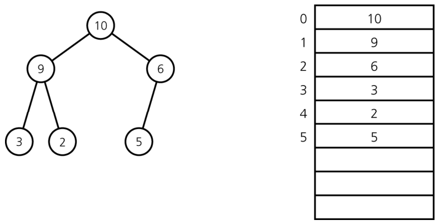
```
createHeap()
// Creates an empty heap.

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

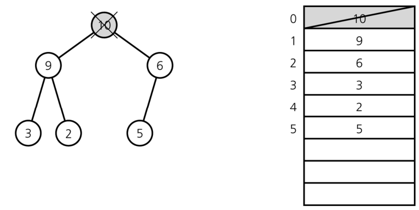


Figure 12-12a

a) Disjoint heaps

Heaps: `heapDelete`

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

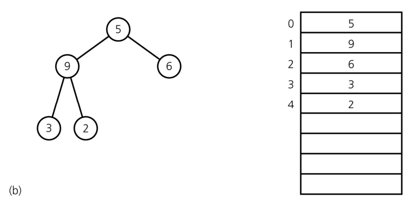


Figure 12-12b

b) a semiheap

Heaps: `heapDelete`

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

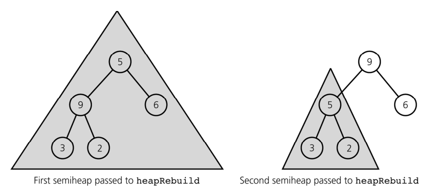


Figure 12-14

Recursive calls to ***heapRebuild***

Heaps: `heapDelete`

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

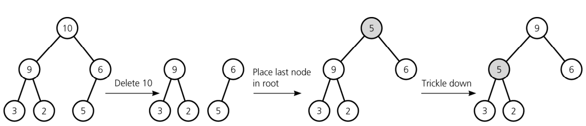


Figure 12-13

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

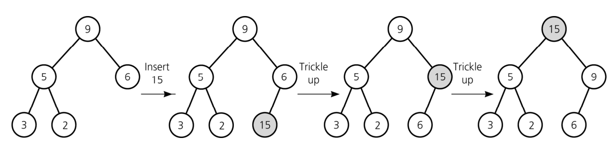


Figure 12-15

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
  - 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

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

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

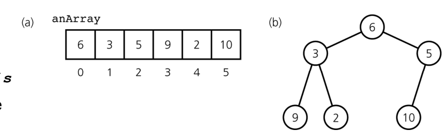
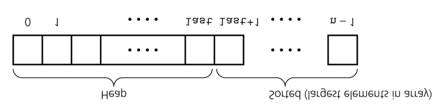


Figure 12-18
Heapsort partitions an array into two regions



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 `Set` Interface

- `Set` interface
  - Ordered collection
  - Stores single value entries
  - Does not allow for duplicate elements
- **`public interface`** `Set<T>` methods
  - **`boolean`** `add(T o)`
  - **`boolean`** `addAll(Collection<? `**`extends`**` T> c)`
  - **`void`** `clear()`
  - **`boolean`** `contains(Object o)`
  - **`boolean`** `isEmpty()`

The JFC `Set` Interface

- **`public interface`** `Set<T>` methods (continued)
  - `Iterator<T> iterator()`
  - **`boolean`** `remove(Object o)`
  - **`boolean`** `removeAll(Collection<?> c)`
  - **`boolean`** `retainAll(Collection<?> c)`
  - **`int`** `size()`

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)`
  - `PriorityQueue(`**`int`**` initialCapacity, Comparator<? `**`super`**` T> comparator)`
  - **`boolean`** `add(T o)`
  - **`void`** `clear()`
  - **`boolean`** `contains(Object o)`

The JFC `PriorityQueue` Class

- **`public Class`** `PriorityQueue<T>` methods (continued)
  - `Comparator<? `**`super`**` T> comparator()`
  - `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
- Tables and priority queues in JFC
  - `Map` interface
  - `Set` interface
  - `PriorityQueue` class