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 (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

  Table Interface 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

- 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?

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

Scenario A: Insertion andTraversal in No Particular Order

- An unsorted order is 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

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

Scenario A: Insertion andTraversal 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 to a reference-based implementation
- Step 2
  - A reference-based implementation is superior to 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

- 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

<table>
<thead>
<tr>
<th></th>
<th>Insertion</th>
<th>Deletion</th>
<th>Retrieval</th>
<th>Traversal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted array based</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Unsorted pointer based</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Sorted array based</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Sorted pointer based</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Binary search tree</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

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

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

Pseudocode for the operations of the ADT priority queue

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

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

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

```java
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

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

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

Figure 12-12b
b) a semiheap

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

Figure 12-14
Recursive calls to `heapRebuild`

---

Heaps: `heapInsert`

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

Figure 12-13
Deletion from a heap

- 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

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

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);

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

Figure 12-16

Figure 12-18

Heapsort partitions an array into two regions

Table 12-18

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 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)
  - 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

The JFC PriorityQueue Class (continued)

• 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()