Chapter 11:
Tables and Priority Queues

Data Abstraction & Problem Solving with C++
Fifth Edition
by Frank M. Carrano
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

![Figure 11-1](image)
An ordinary table of cities

<table>
<thead>
<tr>
<th>City</th>
<th>Country</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>Athens</td>
<td>Greece</td>
<td>2,500,000</td>
</tr>
<tr>
<td>Barcelona</td>
<td>Spain</td>
<td>1,800,000</td>
</tr>
<tr>
<td>Cairo</td>
<td>Egypt</td>
<td>9,500,000</td>
</tr>
<tr>
<td>London</td>
<td>England</td>
<td>9,400,000</td>
</tr>
<tr>
<td>New York</td>
<td>U.S.A.</td>
<td>7,300,000</td>
</tr>
<tr>
<td>Paris</td>
<td>France</td>
<td>2,200,000</td>
</tr>
<tr>
<td>Rome</td>
<td>Italy</td>
<td>2,800,000</td>
</tr>
<tr>
<td>Toronto</td>
<td>Canada</td>
<td>3,200,000</td>
</tr>
<tr>
<td>Venice</td>
<td>Italy</td>
<td>300,000</td>
</tr>
</tbody>
</table>
The ADT Table

- Various sets of table operations are possible

*Figure 11-2* UML diagram for class *Table*

<table>
<thead>
<tr>
<th>Table</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>items</em></td>
</tr>
<tr>
<td><code>createTable()</code></td>
</tr>
<tr>
<td><code>destroyTable()</code></td>
</tr>
<tr>
<td><code>tableIsEmpty()</code></td>
</tr>
<tr>
<td><code>tableLength()</code></td>
</tr>
<tr>
<td><code>tableInsert()</code></td>
</tr>
<tr>
<td><code>tableDelete()</code></td>
</tr>
<tr>
<td><code>tableRetrieve()</code></td>
</tr>
<tr>
<td><code>traverseTable()</code></td>
</tr>
</tbody>
</table>
The ADT Table

- Our table assumes distinct search keys
  - Other tables could allow duplicate search keys
- The `traverseTable` operation visits table items in a specified order
  - One common order is by sorted search key
  - A client-defined `visit` function is supplied as an argument to the traversal
    - Called once for each item in the table
The ADT Table

• **KeyedItem class**
  – Same as for the ADT binary search tree
  – Contains an item’s search key and a method for accessing the search-key data field
  – Prevents modification of the search-key value once an item is created

• Has only a constructor for initializing the search key
  – Also true of a class that extends KeyedItem
Selecting an Implementation

• Linear implementations: Four categories
  – Unsorted: array based or pointer based
  – Sorted (by search key): array based or pointer based

Figure 11-3 The data members for two sorted linear implementations of the ADT table for the data
in Figure 11-1: (a) array based; (b) pointer based
Selecting an Implementation

• Nonlinear implementations
  – Binary search tree implementation
  • Offers several advantages over linear implementations

*Figure 11-4* The data members for a binary search tree implementation of the ADT table for the data in Figure 11-1
Selecting an Implementation

• 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?
    • Are frequently used operations efficient given a particular implementation?
Comparing Linear Implementations

- Unsorted array-based implementation
  - Insertion is made efficiently after the last table item in an array
  - Deletion usually requires shifting data
  - Retrieval requires a sequential search

*Figure 11-5a*  Insertion for unsorted linear implementations: array based
Comparing Linear Implementations

- Sorted array-based implementation
  - Both insertions and deletions require shifting data
  - Retrieval can use an efficient binary search

*Figure 11-6a* Insertion for sorted linear implementations: array based
Comparing Linear Implementations

- Unsorted pointer-based implementation
  - No data shifts
  - Insertion is made efficiently at the beginning of a linked list
  - Deletion requires a sequential search
  - Retrieval requires a sequential search

*Figure 11-5b* Insertion for unsorted linear implementations: pointer based
Comparing Linear Implementations

- Sorted pointer-based implementation
  - No data shifts
  - Insertions, deletions, and retrievals each require a sequential search

*Figure 11-6b* Insertion for sorted linear implementations: pointer based
Selecting an Implementation

• **Linear**
  – Easy to understand conceptually
  – May be appropriate for small tables or unsorted tables with few deletions

• **Nonlinear**
  – Is usually a better choice than a linear implementation
  – A balanced binary search tree
    • Increases the efficiency of the table operations
Selecting an Implementation

<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 11-7* The average-case order of the ADT table operations for various implementations
Selecting an Implementation for a Particular Application

A. Frequent insertions and infrequent traversals in no particular order
   - Unsorted linear implementation

B. Frequent retrievals
   - Sorted array-based implementation
     - Binary search
   - Balanced binary search tree

C. Frequent retrievals, insertions, deletions, traversals
   - Binary search tree (preferably balanced)
A Sorted Array-Based Implementation of the ADT Table

- Default constructor and virtual destructor
- Copy constructor supplied by the compiler
- Has a `typedef` declaration for a “visit” function
- Public methods are virtual
- Protected methods: `setSize`, `setItem`, and `position`
A Binary Search Tree
Implementation of the ADT Table

• **Reuses** `BinarySearchTree`
  – An instance is a private data member
• Default constructor and virtual destructor
• Copy constructor supplied by the compiler
• Public methods are virtual
• Protected method: `setSize`
The ADT Priority Queue: A Variation of the ADT Table

- Orders items by a priority value
- The deletion operation for a priority queue is different from the one for a table
  - The item removed is the one having the highest priority value
- Priority queues do not have retrieval and traversal operations
The ADT Priority Queue: A Variation of the ADT Table

![UML diagram for the class PriorityQueue](Image)

**Figure 11-8** UML diagram for the class *PriorityQueue*
The ADT Priority Queue: 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
    - items[size - 1] has the highest priority

*Figure 11-9a*  An array-based implementation of the ADT priority queue
The ADT Priority Queue: Possible Implementations

• Sorted linear implementations (continued)
  – Pointer-based implementation
    • Maintains the items sorted in descending order of priority value
    • Item having the highest priority is at beginning of linked list

*Figure 11-9b* A pointer-based implementation of the ADT priority queue
The ADT Priority Queue: A Variation of the ADT Table

- Binary search tree implementation
  - Appropriate for any priority queue
  - Largest item is rightmost and has at most one child

*Figure 11-9c* A binary search tree implementation of the ADT priority queue
Heaps

• A heap is a complete binary tree
  – That is empty
  
  *or*

  – Whose root contains a search key $\geq$ the search key in each of its children, *and*

  – Whose root has heaps as its subtrees
Heaps

• Note:
  – The search key in each heap node is \( \geq \) the search keys in each of the node’s children
  – The search keys of a node’s children have no required relationship
Heaps

- The heap is a complete binary tree
  - An array-based representation is attractive
  - But need to know the heap’s maximum size

*Figure 11-11*
A heap with its array representation
Heaps

Figure 11-10 UML diagram for the class Heap
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: An Array-based Implementation of a Heap

- **Constant** MAX_HEAP
- **Data members**
  - `items`: an array of heap items
  - `size`: an integer equal to the current number of items in the heap
Heaps: heapDelete

Strategy

• Step 1: Return the item in the root
  - rootItem = items[0]
  - Results in disjoint heaps

Figure 11-12a  Disjoint heaps
Heaps: heapDelete

- Step 2: Copy the item from the last node into the root: items[0] = items[size-1]
- Step 3: Remove the last node: --size
  - Results in a semiheap

Figure 11-12b  A semiheap
Heaps: `heapDelete`

- Step 3: Transform the semiheap back into a heap
  - Use the recursive algorithm `heapRebuild`
    - The root value trickles down the tree until it is not out of place
      - If the root has a smaller search key than the larger of the search keys of its children, swap the item in the root with that of the larger child
Heaps: heapDelete

Figure 11-13 Deletion from a heap

- **Efficiency**
  - `heapDelete` is $O(\log n)$
Heaps: `heapInsert`

- **Strategy**
  - Insert `newItem` into the bottom of the tree
  - `newItem` trickles up to an appropriate spot in the tree

![Figure 11-15 Insertion into a heap](image)

- **Efficiency:** $O(\log n)$
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

• One implementation
  – Has an instance of the Heap class as a private data member
  – Methods call analogous heap operations
A Heap Implementation of the ADT Priority Queue

– Disadvantage
  • Requires the knowledge of the priority queue’s maximum size

– Advantage
  • A heap is always balanced

• Another implementation
  – 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
  – Transform the array into a heap
  – Remove the heap's root (the largest element) by exchanging it with the heap’s last element
  – Transforms the resulting semiheap back into a heap
Heapsort

Figure 11-17  Transforming the array anArray into a heap
Heapsort

• Compared to mergesort
  – Both heapsort and mergesort are $O(n \times \log n)$ in both the worst and average cases
  – However, heapsort does not require a second array

• Compared to quicksort
  – Quicksort is $O(n \times \log n)$ in the average case
  – It is generally the preferred sorting method, even though it has poor worst-case efficiency: $O(n^2)$
The STL priority_queue

- The STL class priority_queue is an adaptor container
  - The default underlying container is a vector
  - Allows for insertion, but not iteration through its elements
    - Only the top element can be accessed
  - Priority is determined by either
    - The default comparison operator < , or
    - A comparison function object
The STL Heap Algorithms

• Heap algorithms for containers
  - make_heap
    • Turns an existing container into a heap
  - sort_heap
    • Turns the heap back into the original container
  - push_heap
    • Adds an item to a heap
  - pop_heap
    • Removes the first item from a heap
      – Has the highest priority
Summary

• The ADT table supports value-oriented operations
• The linear implementations (array based and pointer based) of a table are adequate only in limited situations
  – When the table is small
  – For certain operations
• A nonlinear pointer based (binary search tree) implementation of the ADT table provides the best aspects of the two linear implementations
  – Dynamic growth
  – Insertions/deletions without extensive data movement
  – Efficient searches
Summary

• A priority queue is a variation of the ADT table
  – Its operations allow you to retrieve and remove the item with the largest priority value

• 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
Summary

• Heapsort, like mergesort, has good worst-case and average-case behaviors, but neither sort is as good as quicksort in the average case

• Heapsort has an advantage over mergesort in that it does not require a second array