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

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

  ```java
  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.

  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.
  ```
Selecting an Implementation

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

\[ \begin{array}{cccc}
\text{size} & \text{head} & \text{tail} & \text{front} \\
0 & 1 & \vdots & \vdots \\
\end{array} \]

Figure 11.2

The data fields for two sorted linear implementations of the ADT table for the data in Figure 11.1: a) array based; b) reference based

Selecting an Implementation

- A binary search implementation
  - A nonlinear implementation

\[ \begin{array}{cccc}
\text{size} & \text{head} & \text{tail} & \text{front} \\
0 & 1 & \vdots & \vdots \\
\end{array} \]

Figure 11.3

The data fields for a binary search tree implementation of the ADT table for the data in Figure 11.1

Scenario A: Insertion and Traversal in No Particular Order

- 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 11.4

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

Scenario A: Insertion and Traversal in No Particular Order

- An unsorted order is efficient
  - Both array based and reference based
  - \text{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 11.3

The data fields for a binary search tree implementation of the ADT table for the data in Figure 11.1

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

Figure 11.4

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

Summary of Tables

- 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
Summary of Tables

<table>
<thead>
<tr>
<th>Implementation</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(log n)</td>
</tr>
<tr>
<td>Sorted pointer based</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(log 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(log n)</td>
</tr>
</tbody>
</table>

Figure 11.6
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 (Continued)

- Pseudocode for the operations of the ADT priority queue
  createPQueue() throws PQueueException // Creates an empty priority queue.
  pqIsEmpty() // Determines whether a priority queue is empty.
  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
  – 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

Pseudocode for the operations of the ADT heap

createHeap()
// Creates an empty heap.

isEmpty()
// 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 11.8
A heap with its array representation

Heaps: heapDelete

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

Figure 11.9a
a) Disjoint heaps

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

Figure 11.9b
b) A semiheap

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

Figure 11.11
Recursive calls to heapRebuild

Heaps: heapDelete

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

Figure 11.10
Deletion for 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 11.12
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

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

Heapsort

- 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

- 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 algorithm is as good in the average case as quicksort
  - Heapsort has an advantage over mergesort in that it does not require a second array