Tables, Priority Queues, Heaps

• Table ADT
  – purpose, implementations

• Priority Queue ADT
  – variation on Table ADT

• Heaps
  – purpose, implementation
  – heapsort
Table ADT

• A table in generic terms has M columns and N rows
  – each row contains a separate record
  – each column contains a different component, or field, of the same record

• Each table, or set of data, is also generally sorted, or accessed, by a key record component
  – a single set of data can be organized into several different tables, sorted according to different keys

• Another common terms is a dictionary, whose entries are records, inserted and accessed according to a key value
  – key may be a field in the record or not
  – may also be used as frontends for data base access
The ADT table, or dictionary

- Uses a search key to identify its items
- Its items are records that contain several pieces of data

<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>
ADT Table – Operations

• A simple and obvious set of operations can be used for a wide range of program activities
  – Create and Destroy Table instance
  – Determine the number of items including zero
  – Insert an item in a table using a key value
  – Delete an item with a given key value
  – Retrieve an item with a given key value
  – Retrieve the items in the table (sorted or unsorted)

• Entries with identical key values maybe forbidden, but can be handled with a little imagination
The ADT Table

- **void tableInsert(ItemType& item):**
  - store item under its key

- **boolean tableDelete(KeyType key_value):**
  - delete item with key == key_value, if present

- **ItemType* tableRetrieve(KeyType key_value):**
  - return pointer to item with key==key_value

- **void traverseTable(Functor visitor):**
  - Functor: a function-object, much like a fn pointer
  - visitor is executed for each node in 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
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

\[ \text{Figure 11-4} \text{ The data members for a binary search tree implementation of the ADT table for the data in Figure 11-1} \]
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
• Sorted pointer-based implementation
  – No data shifts
  – Insertions, deletions, and retrievals each require a sequential search

![Diagram](image)

*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

- **Unsorted array based**
  - Insertion: $O(1)$
  - Deletion: $O(n)$
  - Retrieval: $O(n)$
  - Traversal: $O(n)$

- **Unsorted pointer based**
  - Insertion: $O(1)$
  - Deletion: $O(n)$
  - Retrieval: $O(n)$
  - Traversal: $O(n)$

- **Sorted array based**
  - Insertion: $O(n)$
  - Deletion: $O(n)$
  - Retrieval: $O(\log n)$
  - Traversal: $O(n)$

- **Sorted pointer based**
  - Insertion: $O(n)$
  - Deletion: $O(n)$
  - Retrieval: $O(n)$
  - Traversal: $O(n)$

- **Binary search tree**
  - Insertion: $O(\log n)$
  - Deletion: $O(\log n)$
  - Retrieval: $O(\log n)$
  - Traversal: $O(n)$

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

• Frequent insertions and infrequent traversals in no particular order
  – Unsorted linear implementation

• Frequent retrievals
  – Sorted array-based implementation
    • Binary search
  – Balanced binary search tree

• Frequent retrievals, insertions, deletions, traversals
  – Binary search tree (preferably balanced)
Generalized Data Set Management

• Problem of managing a set of data items occurs many times in many contexts
  – arbitrary set of data represented by an arbitrary key value within the set

• Strict separation of the set of data from the key helps with abstraction and generalization

• Data Set
  – class or structure defined in application terms

• Container class
  – STL terminology
  – holds key and data set items
Keyed Base Class

• Create base class for associating *key* with an arbitrary item
• Maintains key outside the item fields
• Rows of Table are derived classes of this class
• Inserting item in Table creates instance of derived class and stores it under key

```cpp
#include <string>
using namespace std;
typedef stringKeyType;

class KeyedItem
{
public:
    KeyedItem() {}
    KeyedItem(const KeyType& keyValue)
        : searchKey(keyValue) {}
    KeyType getKey() const {
        return searchKey;
    }
private:
    KeyType searchKey;
};
```
Table Item Class

- Create table of cities indexed by city name
- Might create *struct* for each city
  - name, popu., country
- Or, might derive this class from KeyedItem
- Delegates chosen key to base class storage

```cpp
class City : public KeyedItem
{
public:
  City() : KeyedItem() {}
  City(const string& name, const string& ctry, const int& num)
    : KeyedItem(name), country(ctry), pop(num) {}
  string cityName() const;
  int getPopulation() const;
  void setPopulation(int newPop);
private:
  // city's name is search-key value
  string country;
  int pop;
};
```
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`
Priority Queue

• Binary Search Tree is an excellent data structure, but not always
  – simple in concept and implementation
  – BST supports many useful operations well
    • insert, delete, deleteMax, deleteMin, search, searchMax, searchMin, sort
  – efficient average case behavior $T(n) = O(\log n)$
• However, BST is not good in all respects for all purposes
  – brittle with respect to balance
  – worst case $T(n) = O(n)$
• Balanced Trees are possible but more complex
Priority Queue

- Priority Queue semantics are useful when items are added to the set in arbitrary order, but are removed in either ascending or descending priority order
  - priority can have a flexible definition
  - any property of the set elements imposing a total order on the set members
  - If only a partial order is imposed (multiple items with equal priority) a secondary tiebreaking rule can be used to create a total order
Priority Queue

• The deletion operation for a priority queue is different from the one for a table
  – general ‘delete’ operation is not supported
  – item removed is the one having the highest priority value

• Priority queues do not have retrieval and traversal operations
ADT Priority Queue

<table>
<thead>
<tr>
<th>PriorityQueue</th>
</tr>
</thead>
<tbody>
<tr>
<td>items</td>
</tr>
<tr>
<td>createPriorityQueue()</td>
</tr>
<tr>
<td>destroyPriorityQueue()</td>
</tr>
<tr>
<td>pqIsEmpty()</td>
</tr>
<tr>
<td>pqInsert()</td>
</tr>
<tr>
<td>pqDelete()</td>
</tr>
</tbody>
</table>

**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: Possible Implementations

- 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
The ADT Priority Queue: Heap Implementation

• A heap is a complete binary tree
  – that is empty, OR
  – whose root contains a search key >= the search key in each of its children, and whose root has heaps as its subtrees

• Heap is the best approach because it is the most efficient for the specific PQ semantics

• Heap provides a partially ordered tree
  – avoids brittleness of BST and has lower overhead than balanced search trees
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

• A maximum, binary, heap H is a complete binary tree satisfying the *heap-ordered* tree property:
  – *Complete*: Every level complete, except possibly the last, and all leaves are as far left as possible
  – *Heap Ordered*: Priority of any node is $\geq$ priority of all its descendants
    – maximum element of set is thus at root
• A minimum heap ensures that all nodes have priority values $\leq$ all its descendants
  – minimum element at root
Heap – ADT

<table>
<thead>
<tr>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>items</strong></td>
</tr>
</tbody>
</table>

| createHeap() |
| destroyHeap() |
| heapIsEmpty() |
| heapInsert() |
| heapDelete() |

*Figure 11-10  UML diagram for the class Heap*
Heap – Implementation

• Considering typical heap operations, for example, insert into heap
• Result must be a complete tree satisfying the heap property that all nodes are \( \geq \) descendants
• Two step insert process works well
  – insert the new item in the next “open” slot for keeping \( H \) a complete binary tree
  – restructure \( H \) to make it satisfy the heap-ordered property
• Two step remove
  – client code save root value for use
  – Replace root with “last” node in level-order
  – Restructure \( H \) to migrate/percolate new root to the correct tree location
Heap – Implementation

- Traversal of the inserted node to its proper place requires at most $O(\log n)$ operations -- since the height of a complete binary tree is $O(\log n)$
• **Deletion** is similar
  – always deletes the root of the tree, left with two disjoint subtrees
  – place item in last node in the root
  – out of place item in root node should percolate down to its proper position
  – $O(\log n)$
Heap – Implementation

• Data structure suitable for heap implementation must
  – support efficient determination of where next and last
    slots in a complete tree are located for insert and delete,
    respectively
  – support efficient percolation of misplaced nodes
• Percolation down is simple using standard child
  references and comparison of parent to child values
• Percolation up is almost as simple, but requires a
  parent reference at each node
• Knowing the last occupied and next open slots under
  different data structures is more subtle under some
  data structures than others
Heap – Implementation

• Pointer based heaps require two child and one parent pointer at each node
  – can use additional state information to track location of next and last complete tree slots
• Array based heap implementation simplifies parent and child references by making them calculated
  – lowers space overhead
  – not clear execution time would be lower
    • array index calculation vs. pointer access
• Similarly, location of the next and last slots for the complete tree can be calculated from the number of nodes in the tree, which is simple to track
Heap – Array Implementation

• In an array representation of a binary tree T
  – Root of T is at A[0]
  – parent of a node A[i] is at A[(i-1)/2]
  – for n>1, A[i] is a leaf iff 2i>n
  – in a heap with n elements the last element of the complete binary tree is at A[n-1] and the next element (element n+1) will be added at A[n]
Heap – Array Implementation

• An array-based representation is attractive
  – need to know the heap’s maximum size
• Constant MAX_HEAP
• Data members
  – items: an array of heap items
  – size: an integer equal to the current number of items in the heap
Heap – Array Implementation

- heapDelete operation with arrays
- Step 1: Return the item in the root
  - rootItem = items[0]

Figure 11-12a  Disjoint heaps
(a)
Heap – Array Implementation

• 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](image) A semiheap (b)
Heap – Array Implementation

• Step 3: Transform the semi-heap 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
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 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 : $O(n^2)$
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.