Tables and Priority Queue

孫雅麗
台灣大學 資管所
The ADT Table

"Attributes", "Tuples"

<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>

- Appropriate for problems that manage data by **value**.
- Table implementations: **array, linked lists, and binary search trees**.
ADT priority queue

- To retrieve and delete efficiently the item with the largest value

- Implementation: heap – a variant of binary search tree
Introduction

- The ADT table, or dictionary, allows users to look up information.
- Search information based on a specified search key (a combination of one or more fields).
- Implementation choices:
  - Efficiency
  - Arrange the data to facilitate the search, insert and delete.
Pseudocode for ADT Table Operations

createTable()
// Creates an empty table.

destroyTable()
// Destroys a table.

tableIsEmpty():boolean {query}
// Determines whether a table is empty.

tableLength(): integer {query}
// Determines the number of items in a table.
ADT Table Operations (cont’d)

tableInsert(in newItem: TableItem Type) throw TableException

// Insert newItem into a table whose items have distinct
// search keys that differ from newItem's search key.

tableDelete(in searchKey: KeyType) throw TableException

// Deletes from a table the item whose search key equals
// searchKey. If no such item exists, throw exception.
ADT Table Operations

tableRetrieve(in searchKey: KeyType, out tableItem: TableItemType) throw TableException

// Retrieves into tableItem the item in a table whose search key equals searchKey.

traverseTable(in visit: FunctionType)

// Traverses a table in sorted search-key order and calls the function visit once for each item.
Implementation Choices of ADT Table (1/4)

**Efficiency!**

- **Unsorted**
  - array based vs. pointer based

- **Sorted (by search key)**
  - array based vs. pointer based
Unsorted Table: array-based vs. pointer-based

(a) Unsorted Table (Array-Based)

- **Size**: K + 1
- **Items**: Data, Data, ..., Data, New item, ?, ..., ?
- **Insert**, **Delete**, **Search**

(b) Unsorted Table (Pointer-Based)

- **Size**: K + 1
- **Head**
- **Insert**
- **Delete**
- **Search**

- **Time Complexity**: O(?)
Implementation Choices of ADT Table (2/4)

- Sorted (by search key)
  - array based vs. pointer based
Sorted Table: array-based vs. pointer-based

Figure 11-5

- Insert
- Delete

$O(\text{?})$
Sorted Table: array-based vs. pointer-based

<table>
<thead>
<tr>
<th>Size</th>
<th>Athens •••</th>
<th>Barcelona •••</th>
<th>•••</th>
<th>Venice •••</th>
<th>•••</th>
<th>•••</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 11-2**

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

- Search
- O(?)
Binary Search Tree
Implementation of ADT Table (3/4)

Figure 11-3
The data members for a binary search tree implementation of the ADT table for the data in Figure 11-1
Comparisons of the Average-case Order of the ADT Operations \((4/4)\)

<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(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 11-6*

The average-case order of the ADT table operations for various implementations
Discussions

- Select an appropriate implementation according to the application needs.
  - Insertion and traversal in no particular order
    → unsorted
  - Retrieval
    → sorted, binary search (array, pointer)
  - insertion, deletion, retrieval and traversal in sorted order
    → BST
Implementation Alternatives

See the sample codes in the textbook

- sorted array
- binary search tree
Design Rationale for Class KeyedItem

- It is important that the value of the search key remain the same as long as the item is sorted in the table.
- The search key value should NOT be modified directly by other class objects.
- Define a class containing the search key as a data member and a method for accessing it.
- Classes that extend KeyedItem will have ONLY the constructor available for initializing the search key.
#include <string>
Using namespace std;
typedef string KeyType; // as a structure
class KeyedItem
{
  public:
    KeyedItem() {};
    KeyedItem(const KeyType & keyValue): searchKey(KeyValue) {}; //init
    KeyType getKey() const //retrieve
    {
      return searchKey;
    } // getKey
  private:
    KeyType searchKey;
}; // end class
#include <string>
Using namespace std;

class city: public KeyedItem {
    public:
    city() {};
    city(const string & name, const string & cty, 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;
}; // end class
Priority Queue
Some applications want to retrieve the item with the highest priority value.

e.g., printer server, transaction processing server, emergency center, etc.

Each data item is given a priority value when entering the system.

The data items are organized to facilitate the “retrieval” (search) (e.g., $O(1)$), insertion and deletion operations (e.g., $O(\log N)$)
ADT Priority Queue Operations

createPQueue()
  // Creates an empty priority queue.
destroyPQueue()
  // Destroys a priority queue.
pqIsEmpty()
  // Determines whether a priority queue is empty.
pqInsert(in newItem:PQItemType) throw PQException
  // insert newItem to a priority queue.
pqDelete(out priorityItem:PQItemType) throw PQException
  // Retrieves into priorityItem and then deletes the item
  // with the highest priority value.
Possible Implementations of ADT PQueue

Index = size - 1

Sorted array

Sorted linked list

**Figure 11-7**

Some implementations of the ADT priority queue: (a) array based; (b) pointer based; (c) binary search tree
Implementations of ADT PQQueue

Binary search tree

Figure 11-7
Some implementations of the ADT priority queue: (a) array based; (b) pointer based; (c) binary search tree
Heap: An ADT

- **Definition: max heap**
  
  - A **max tree** is a tree in which the key value in each node is **no smaller than** the key values in its **children** (if any).

- **A max heap** is a **complete binary tree** that is also a max tree.
Example Max Heaps

Sample max heaps
Heap: An ADT

Definition: **min tree**
- A *min tree* is a tree in which the key value in each node is **no larger than** the key values in its **children** (if any).

A *min heap* is a **complete binary tree** that is also a min tree.
Example Min Heaps

Sample min heaps
ADT Heap - recursive definition

A **heap** is a **complete** binary tree

1. that is empty

or

2a. whose **root** contains a priority value greater than or equal to the priority value in each of its children, and

2b. Whose root has **heaps** as its **subtrees**
ADT Heap Operations

createHeap()
   // Creates an empty heap.
destroyHeap()
   // Destroys a heap.
heapIsEmpty(): boolean {query}
   // Determines whether a heap is empty.
heapInsert(in newItem:HeapItemType) throw HeapException
   // Adds newItem to a heap. make the resulting tree a heap - a complete binary tree

heapDelete(out rootItem:HeapItemType) throw HeapException
   // Removes and retrieves a heap's root item. make the resulting tree a heap - a complete binary tree
Heap: array representation

A heap is a **complete binary tree**

\[ P_{\text{root}} \geq P_{\text{left child}} \land P_{\text{right child}} \]

*Level ordering (numbering)*

*Figure 11-8*

A heap with its array representation
heapDelete: step 1 - remove root with the max value
heapDelete: step 2 - move the last one to the root

make it a complete binary tree!

heapDelete: step 3 - move the new node to the right position - trickle down

- Root : i
- LeftChild = 2i + 1
- RightChild = Left + 1
HeapDelete - summary

1. Step 1:
   largest priority value ?: rootItem = items[0]

2. Step 2: // make the resulting tree a heap - a complete binary tree
   Copy the item from the last node into the root
   items[0] = items[size-1]; --size;

3. Step 3:
   Trickle down (sinking)
   heapRebuild(items, 0, size);
Recursive heapRebuild()

Fig 11-14
heapRebuild ()

heapRebuild (inout items: ArrayType, in root: integer, in size: integer)

// Recursively trickle the item at index Root down to
// its proper position by swapping it with its larger
// child, if the child is larger than the item.
// If the item is at a leaf, nothing needs to be done.

if (the root is not a leaf)
{
    // root must have a left child
    child = 2 * root + 1  // left child index
heapRebuild () (cont’d)

if (the root has a right child)
{
    rightChild = child + 1  // right child index
    if (items[rightChild].getKey() > items[child].getKey())
        child = rightChild  // larger child index
}  // end if

// if the item in the root has a smaller search key than the search
// key of the item in the larger child, swap items
if (items[root].getKey() < items[child].getKey())
{
    Swap items[root] and items[child]
    // transform semiheap rooted at child into a heap
    rebuildHeap(items, child, size)
}  // end if
}  // end if

// else root is a leaf, so you are done

Efficiency ??
HeapInsert

- Step 1: make a new last node $\rightarrow$ a complete binary tree
- Step 2: POP UP!!
HeapInsert

Figure 11-12
Insertion into a heap

POP UP !!
heapInsert()

// Step 1 - insert newItem into the bottom of the tree
items[size] = newItem

// Step 2 - trickle new item UP to appropriate spot in the tree
place = size
parent = (place-1)/2
while ( (parent >= 0) and (items[place] > items[parent]) )
  //TRICK UP
  { Swap items[place] and items[parent]
    place = parent // go one level up again
    parent = (place-1)/2
  }
increment size
const int MAX_HEAP = maximum-size-of-heap;
#include “KeyedItem.h”

typedef KeyedItem HeapItemType;
class Heap
{public:
  heap (); // default constructor
    // copy constructor and destructor are supplied by
  the compiler
// heap operations:
virtual bool heapIsEmpty() const;
   // Determines whether a heap is empty.
   // Precondition: None.
   // Postcondition: Returns true if the heap is empty;
   // else returns false.

virtual void heapInsert(const heapItemType& newItem) throw (HeapException)
   // Inserts an Item into a heap.
   // Precondition: newItem is the item to be inserted.
   // Postcondition: If the heap was not full, newItem is
   // in its proper position; otherwise throw exception
Heap.h (cont’d)

virtual void heapDelete(heapItemType& rootItem) throw (HeapException)

  // Retrieves and deletes the item in the root of a heap.
  // This item has the largest search key in the heap.
  // Precondition: None.
  // Postcondition: If the heap was not empty, rootItem
  // is the retrieved item, the item is deleted from the
  // heap; otherwise throw exception
protected:
    void heapRebuild (int root);
    // Converts the semiheap rooted at index Root
    // into a heap.

private:
    heapItemType items[MAX_HEAP];  // array of heap items
    int size;  // number of heap items
};  // end class

// End of header file.
A Heap Implementation of ADT PQQueue (cont’d)

// priority queue operations:
virtual bool pqIsEmpty();
virtual void pqInsert(const PQueueItemType & newItem) throw(PQueueException);
virtual void pqDelete(PQueueItemType & priorityItem) throw(PQueueException);

private:
    heap h;
}; // end class

// End of header file.
A Heap Implementation of ADT PQueue

// ********************************************
// Implementation file PQ.cpp for the ADT priority queue.
// A heap represents the priority queue.
// ********************************************

#include "PQ.h"    // header file for priority queue

bool PriorityQueue ::pqIsEmpty()
{
  return h.heapIs Empty();
}  // end pqIsEmpty
A Heap Implementation of ADT
PQueue (cont’d)

```cpp
void PriorityQueue::PQueueInsert(const PQueueItemType &newItem) throw(PQueueException);
{
    try {
        h.heapInsert(newItem);
    } // end try
    catch (HeapException e){
        throw PQueueException("PQueueException: Priority queue full");
    } // end catch
} // end PQueueInsert
```
A Heap Implementation of ADT PQueue (cont’d)

void pqDelete(PQueueItemType & priorityItem)
   throw(PQueueException)
{
   try
   {
      h.heapDelete(priorityItem);
   } // end try
   catch (HeapException e)
   {
      throw PQueueException ("PQueueException: Priority queue empty");
   } // end catch
} // end PQueueDelete

// End of implementation file.
#include "Heap.h"  // header file for class Heap

Heap::Heap() : size(0)
{
}  // end default constructor

Heap::~Heap()
{
}  // end destructor

bool Heap::heapIsEmpty() const
{
    return bool(size == 0);
}  // end heapIsEmpty
void Heap::heapInsert(const HeapItemType& newItem)
    throw(HeapException)
    // Method: Inserts the new item after the last item in the
    // heap and trickles it up to its proper position. The
    // heap is full when it contains MAX_HEAP items.
{
    if (size >= MAX_HEAP)
        throw HeapException("HeapException: Heap full");
    // place the new item at the end of the heap
    items[size] = newItem;

    // trickle new item up to its proper position
    int place = size;
    int parent = (place - 1)/2;
ADT Heaps (cont’d)

while ( (parent >= 0) &&
        (items[place].getKey() > items[parent].getKey()) )
{
    // swap items[place] and items[parent]
    HeapItemType temp = items[parent];
    items[parent] = items[place];
    items[place] = temp;

    place = parent;
    parent = (place - 1)/2;
}
// end while

++size;
} // end heapInsert
void Heap::heapDelete(HeapItemType& rootItem) 
    throw(HeapException)
    // Method: Swaps the last item in the heap with the root
    // and trickles it down to its proper position.
    {
        if (heapIsEmpty())
            throw HeapException("HeapException: Heap empty");
        else
            { rootItem = items[0];
                items[0] = items[--size];
                heapRebuild(0);
            } // end if
    } // end heapDelete
void Heap::heapRebuild(int root)
{
    // if the root is not a leaf and the root's search key
    // is less than the larger of the search keys in the
    // root's children
    int child = 2 * root + 1;  // index of root's left
    // child, if any

    if ( child < size )
    {
        // root is not a leaf, so it has a left child at child
        int rightChild = child + 1;  // index of right child,
        // if any

        // if root has a right child, find larger child
        if ( (rightChild < size) &&
             (items[rightChild].getKey() > items[child].getKey()) )
            child = rightChild;  // index of larger child
    }
ADT Heaps

// if the root's value is smaller than the
// value in the larger child, swap values
if ( items[root].getKey() < items[child].getKey() )
{
    HeapItemType temp = items[root];
    items[root] = items[child];
    items[child] = temp;

    // transform the new subtree into a heap
    heapRebuild(child);
} // end if
} // end if

// if root is a leaf, do nothing
} // end heapRebuild

// End of implementation file.
Compare heap and binary search tree implementations

- If the maximum number of items is known, heap implementation is a better choice.
- Why?
  - Heap is a complete binary tree.
  - It is always balanced.
  - Average and worst-case: $O(\log)$

- Are there applications more appropriate to use binary search tree than heap implementation?
- Search and traverse in search-key order
The end. 😊
homework

- Page 647
- #3, #5, #6, #9 and #18
Heap Sort

See CHV2 chap09-2 Sorting
Figure 11-9
(a) Disjoint heaps; (b) a semiheap
Figure 11-9
(a) Disjoint heaps; (b) a semiheap
Discussion

- When an instance of class City is created, it is returned by the **inherited method** key.