Standard ADTs

Lecture 14
CS211 – Summer 2007

Announcements

- Assignment 2 regrades due today, 11:59 PM
- Assignment 3 due tomorrow, 11:59 PM

Abstract Data Types (ADTs)

- A method for achieving abstraction for data structures and algorithms
  - ADT = model + operations
  - Describes what each operation does, but not how it does it
  - An ADT is independent of its implementation
- In Java, an interface corresponds well to an ADT
  - The interface describes the operations, but says nothing at all about how they are implemented
- Example: Stack interface/ADT

```
public interface Stack {
  public void push(Object x);
  public Object pop();
  public Object peek();
  public boolean isEmpty();
  public void makeEmpty();
}
```


Queues & Priority Queues

- ADT Queue
  - Operations:
```
void enQueue(Object x);
Object deQueue();
Object peek();
boolean isEmpty();
void makeEmpty();
```
- Where used:
  - Simple job scheduler (e.g., print queue)
  - Wide use within other algorithms

- ADT PriorityQueue
  - Operations:
```
void insert(Object x);
Object getMax();
Object peekAtMax();
boolean isEmpty();
void makeEmpty();
```
- Where used:
  - Job scheduler for OS
  - Event-driven simulation
  - Can be used for sorting
  - Wide use within other algorithms

Sets

- ADT Set
  - Operations:
```
void insert(Object element);
boolean contains(Object element);
void remove(Object element);
boolean isEmpty();
void makeEmpty();
```
- Where used:
  - Wide use within other algorithms
- Note: no duplicates allowed
  - Like a set in mathematics
  - A “set” with duplicates is sometimes called a *multiset* or *bag*

Dictionaries

- ADT Dictionary
  - Operations:
```
void insert(Object key, Object value);
void update(Object key, Object value);
Object find(Object key);
void remove(Object key);
boolean isEmpty();
void makeEmpty();
```
- Think of: key = word; value = definition
- Where used:
  - Symbol tables
  - Wide use within other algorithms

Data Structure Building Blocks

- These are *implementation* “building blocks” that are often used to build more complicated data structures
  - Arrays
  - Linked Lists
    - Singly linked
    - Doubly linked
  - Binary Trees
  - Graphs
    - Adjacency matrix
    - Adjacency list

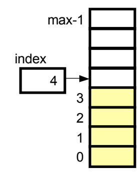
Array Implementation of Stack

```
class ArrayStack implements Stack {

   private Object[] array; //Array that holds the Stack
   private int index = 0; //First empty slot in Stack

   public ArrayStack (int maxSize)
      { array = new Object[maxSize]; }

   public void push(Object x) { array[index++] = x; }
   public Object pop() { return array[--index]; }
   public Object peek() { return array[index-1]; }
   public boolean isEmpty() { return index == 0; }
   public void makeEmpty() { index = 0; }
}
//Better for garbage collection if makeEmpty() also
   cleared the array
```



O(1) worst-case time for each operation

Caveat: This implementation imposes a fixed limit on stack size.

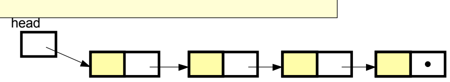
Linked List Implementation of Stack

```
class ListStack implements Stack {
   private Node head = null;   //Head of list that
                               //holds the Stack

   public void push(Object x) { head = new Node(x, head);
   }
   public Object pop() {
      Node temp = head;
      head = head.next;
      return temp.data;
   }
   public Object peek() { return head.data; }
   public boolean isEmpty() { return head == null; }
   public void makeEmpty() { head = null; }
}
```

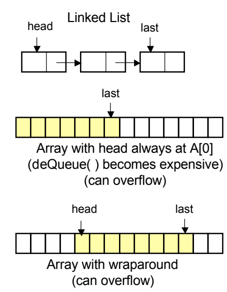
O(1) worst-case time for each operation

Unlike array implementation, this implementation doesn’t impose a limit on stack size



Queue Implementations

- Recall: operations are `enQueue`, `deQueue`, `peek`,…
  - For linked-list
    - All operations are O(1)
  - For array with head at A[0]
    - deQueue takes time O(n)
    - Other ops are O(1)
    - Can overflow
  - For array with wraparound
    - All operations are O(1)
    - Can overflow
- Possible implementations



Choosing an Implementation

- What operations do I need to perform on the data?
  - Insertion, deletion, searching, reset to initial state?
  - How often is each operation performed?
- How efficient do the operations need to be?
- Are there any additional constraints on the operations or on the data structure?
  - Can there be duplicates?
  - When extracting elements, does order matter?
- Is there a known upper bound on the amount of data? Or can it grow unboundedly large?

Goal: Design a *Dictionary*

- Operations

```
void insert(key, value)
void update(key, value)
Object find(key)
void remove(key)
boolean isEmpty()
void makeEmpty()
```

Array implementation: Using an array of (key,value) pairs

	Unsorted	Sorted
insert	O(1)	O(n)
update	O(n)	O(log n)
find	O(n)	O(log n)
remove	O(n)	O(n)

n is the number of items currently held in the dictionary

If the keys happen to be bounded integers, we can do better

Hashing

- Idea: "convert" an arbitrary key into a bounded integer
- Use a *hash function* h
  - U is the universe of keys
  - h: U → [0,…,m-1] where m = hash table size
- h should
  - Be easy to compute
  - Cause few *collisions*
  - Have equal probability for each table position

Typical situation:
U = all legal identifiers

Typical hash function:
h converts each letter to a number and we compute a function of these numbers

A Hashing Example

- A very simple hash function: count # of letters in string
- Suppose each word below has the following hashCode

january	7
february	8
march	5
april	5
may	3
june	4
july	4
august	6
september	9
october	7

- How do we resolve collisions?
  - use chaining: each table position is the head of a list
  - for any particular problem, this *might* work terribly
- In practice, using a good hash function, we can assume each position is equally likely

Analysis for Hashing with Chaining

- Analyzed in terms of *load factor*
  - λ = n/m = (items in table)/(table size)
- We count the expected number of *probes* (key comparisons)
- U = the expected # of probes for an *unsuccessful* search
  - U = average number of items per table position = n/m = λ
- S = expected number of probes for a *successful* search
  - S = 1 + λ/2 = O(λ)

Table Doubling

- We know each operation takes time O(λ) where λ=n/m
  - But isn't λ = O(n)?
- What's the deal here? It's still linear time!
- A solution: Table Doubling:
  - Set a bound for λ (call it λ_0)
  - Whenever λ reaches this bound we
    - Create a new table, twice as big and
    - Re-insert all the data
- Easy to see operations *usually* take time O(1)
  - But sometimes we copy the whole table

Analysis of Table Doubling

- Suppose we reach a state with n items in a table of size m and that we have just completed a table doubling

	Copying Work
Everything has just been copied	n inserts
Half were copied previously	n/2 inserts
Half of those were copied previously	n/4 inserts
…	…
Total work	n + n/2 + n/4 + … = 2n

Analysis of Table Doubling, Cont'd

- Total number of insert operations needed to reach current table
  - = copying work + initial insertions of items
  - = 2n + n = 3n inserts
- Each insert takes expected time O(λ_0) or O(1), so total expected time to build entire table is O(n)
- Thus, expected time per operation is O(1)
  - Cost of table doubling is *amortized* over many table inserts
- Disadvantages of table doubling:
  - Worst-case insertion time of O(n) definitely occurs (but rarely)
  - Thus, not appropriate for time critical operations

Java Hash Functions

- Most Java classes implement the `hashCode()` method
  - `hashCode()` returns an int
- Java's `HashMap` class uses h(X) = X.hashCode() mod m
- h(X) in detail:

```
int hash = X.hashCode();
int index = (hash & 0x7FFFFFFF) % m;
```

- What `hashCode()` returns:
  - Integer:
    - uses the int value
  - Float:
    - converts to a bit representation and treats it as an int
  - Short Strings:
    - 37*previous + value of next character
  - Long Strings:
    - sample of 8 characters; 39*previous + next value

`hashCode()` Requirements

- Contract for `hashCode()` method:
  - Whenever it is invoked in the same object, it must return the same result
  - Two objects that are equal must have the same hash code
  - Two objects that are not equal should ideally return different hash codes, but are not required to do so

Hashtables in Java

- `java.util.HashMap`
- `java.util.HashSet`
- `java.util.Hashtable`
- Use chaining
- Initial (default) size = 101
- Load factor = λ_0 = 0.75
- Uses table doubling (2*previous+1)

- A node in each chain looks like this:

hashCode	key	value	next

original hashCode (before mod m)
Allows faster rehashing and (possibly) faster key comparison

Linear & Quadratic Probing

- All data is stored directly within the hash table array
- Linear Probing
  - Probe at h(X), then at h(X) + 1, h(X) + 2, … h(X) + i
  - Leads to *primary clustering*
    - Long sequences of filled cells
- Quadratic Probing
  - Similar to Linear Probing in that data is stored within the table
  - Probe at h(X), then at h(X) + 1, h(X) + 4, h(X) + 9, … h(X) + i^2
  - Works well when
    - $\lambda < 0.5$
    - Table size is prime

Hashtable Pitfalls

- Good hash function is required
- Watch the load factor (λ), especially for Linear & Quadratic Probing

Dictionary Implementations

- Ordered Array
  - Better than unordered array because Binary Search can be used
- Unordered Linked-List
  - Ordering doesn't help
- Hashtables
  - O(1) expected time for Dictionary operations