

Standard ADTs

Lecture 16
CS2110 – Fall 2008

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 clear();
}
```

2

Queues & Priority Queues

- ADT Queue
 - Operations:


```
void add(Object x);
Object poll();
Object peek();
boolean isEmpty();
void clear();
```
- 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 clear();
```
- Where used:
 - Job scheduler for OS
 - Event-driven simulation
 - Can be used for sorting
 - Wide use within other algorithms

3

Sets

- ADT Set
 - Operations:


```
void insert(Object element);
boolean contains(Object element);
void remove(Object element);
boolean isEmpty();
void clear();
```
- Where used:
 - Wide use within other algorithms
- Note: no duplicates allowed
 - A "set" with duplicates is sometimes called a *multiset* or *bag*

4

Dictionaries

- ADT Dictionary (aka Map)
 - Operations:


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

5

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

6

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 clear() { index = 0; }
}
```



O(1) worst-case time for each operation

Question: What can go wrong?

7

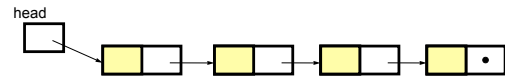
Linked List Implementation of Stack

```
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 clear() { head = null; }
}
```

O(1) worst-case time for each operation (but constant is larger)

Note that array implementation can overflow, but the linked list version cannot

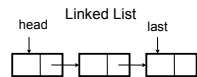


8

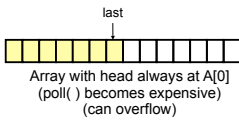
Queue Implementations

Possible implementations

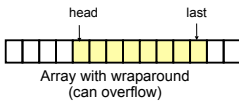
Recall: operations are **add**, **poll**, **peek**,...



For linked-list
• All operations are O(1)



For array with head at A[0]
• poll takes time O(n)
• Other ops are O(1)
• Can overflow



For array with wraparound
• All operations are O(1)
• Can overflow

9

A Queue From 2 Stacks

- Add pushes onto stack A
- Poll pops from stack B
- If B is empty, move all elements from stack A to stack B
- Some individual operations are costly, but still O(1) time per operations over the long run

10

Dealing with Overflow

- For array implementations of stacks and queues, use **table doubling**
- Check for overflow with each insert op
- If table will overflow,
 - Allocate a new table twice the size
 - Copy everything over
- The operations that cause overflow are expensive, but still constant time per operation over the long run (proof later)

11

Goal: Design a Dictionary (aka Map)

Operations

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

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

12

Hashing

- Idea: compute an array index via a *hash function* h
- U is the universe of keys
- $h: U \rightarrow [0, \dots, m-1]$ where m = hash table size
- Usually $|U|$ is much bigger than m , so *collisions* are possible (two elements with the same hash code)
- h should
 - be easy to compute
 - avoid collisions
 - have roughly equal probability for each table position

Typical situation:

U = all legal identifiers

Typical hash function:

h converts each letter to a number, then compute a function of these numbers

13

A Hashing Example

- Suppose each word below has the following hashCode

```

jan    7
feb    0
mar    5
apr    2
may    4
jun    7
jul    3
aug    7
sep    2
oct    5
    
```

- 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

14

Analysis for Hashing with Chaining

- Analyzed in terms of *load factor* $\lambda = n/m = (\text{items in table})/(\text{table size})$
- We count the expected number of *probes* (key comparisons)
- Goal: Determine expected number of probes for an *unsuccessful* search
- Expected number of probes for an *unsuccessful* search = average number of items per table position = $n/m = \lambda$
- Expected number of probes for a *successful* search = $1 + \lambda/2 = O(\lambda)$
- Worst case is $O(n)$

15

Table Doubling

- We know each operation takes time $O(\lambda)$ where $\lambda = n/m$
- So it gets worse as n gets large relative to m
- Table Doubling:
 - Set a bound for λ . (call it λ_0)
 - Whenever λ reaches this bound:
 - Create a new table twice as big
 - Then rehash all the data
 - As before, operations *usually* take time $O(1)$
 - But sometimes we copy the whole table

16

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	$n + n/2 + n/4 + \dots = 2n$

17

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 $O(\lambda_0)$ or $O(1)$, so total expected time to build entire table is $O(n)$
- Thus, expected time per operation is $O(1)$
- Disadvantages of table doubling:
 - Worst-case insertion time of $O(n)$ is definitely achieved (but rarely)
 - Thus, not appropriate for time critical operations

18

Java Hash Functions

- Most Java classes implement the `hashCode()` method
- `hashCode()` returns an int
- Java's `HashMap` class uses $h(X) = X.hashCode() \bmod m$
- `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:
 - $37^{\text{previous}} + \text{value of next character}$
 - Long Strings:
 - sample of 8 characters;
 - $39^{\text{previous}} + \text{next value}$

19

`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 (in the sense of `.equals(...)`) must have the same hash code
 - Two objects that are not equal should return different hash codes, but are not required to do so (i.e., collisions are allowed)

20

Hashtables in Java

- `java.util.HashMap`
- `java.util.HashSet`
- `java.util.Hashtable`
- Use chaining
- Initial (default) size = 101
- Load factor = $\lambda_0 = 0.75$
- Uses table doubling ($2^{\text{previous}+1}$)
- 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

21

Linear & Quadratic Probing

- These are techniques in which 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

22

Universal Hashing

- Choose a hash function at random from a large parameterized family of hash functions (e.g., $h(x) = ax + b$, where a and b are chosen at random)
- With high probability, it will be just as good as any custom-designed hash function you can come up with

23

`hashCode()` and `equals()`

- We mentioned that the hash codes of two equal objects must be equal — this is necessary for hashtable-based data structures such as `HashMap` and `HashSet` to work correctly
- In Java, this means if you override `Object.equals()`, you had better also override `Object.hashCode()`
- But how???

24

hashCode () and equals ()

```
class Identifier {
    String name;
    String type;

    public boolean equals(Object obj) {
        if (obj == null) return false;
        Identifier id;
        try {
            id = (Identifier)obj;
        } catch (ClassCastException cce) {
            return false;
        }
        return name.equals(id.name) && type.equals(id.type);
    }
}
```

25

hashCode () and equals ()

```
class Identifier {
    String name;
    String type;

    public boolean equals(Object obj) {
        if (obj == null) return false;
        Identifier id;
        try {
            id = (Identifier)obj;
        } catch (ClassCastException cce) {
            return false;
        }
        return name.equals(id.name) && type.equals(id.type);
    }

    public int hashCode() {
        return 37 * name.hashCode() + 113 * type.hashCode() + 42;
    }
}
```

26

hashCode () and equals ()

```
class TreeNode {
    TreeNode left, right;
    String datum;

    public boolean equals(Object obj) {
        if (obj == null || !(obj instanceof TreeNode)) return false;
        TreeNode t = (TreeNode)obj;
        boolean lEq = (left != null)?
            left.equals(t.left) : t.left == null;
        boolean rEq = (right != null)?
            right.equals(t.right) : t.right == null;
        return datum.equals(t.datum) && lEq && rEq;
    }
}
```

27

hashCode () and equals ()

```
class TreeNode {
    TreeNode left, right;
    String datum;

    public boolean equals(Object obj) {
        if (obj == null || !(obj instanceof TreeNode)) return false;
        TreeNode t = (TreeNode)obj;
        boolean lEq = (left != null)?
            left.equals(t.left) : t.left == null;
        boolean rEq = (right != null)?
            right.equals(t.right) : t.right == null;
        return datum.equals(t.datum) && lEq && rEq;
    }

    public int hashCode() {
        int lHC = (left != null)? left.hashCode() : 298;
        int rHC = (right != null)? right.hashCode() : 377;
        return 37 * datum.hashCode() + 611 * lHC - 43 * rHC;
    }
}
```

28

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

29