



## Standard ADTs

Lecture 14  
CS211 – Summer 2008

## Announcements

Assignment 4 posted; due Wed 6pm

- Recurrences...
- Mystery Question!

2

## 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();  
}
```

3

## 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

4

## 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*

5

## 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

6

## 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

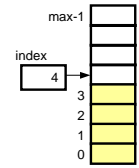
7

## 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.

8

## Detour: What about ArrayList?

- List implemented using array . . . Does it impose fixed size limit?

- What is the cost of adding to end of an ArrayList?

9

## Detour: What about ArrayList?

- List implemented using array . . . Does it impose fixed size limit?

- What is the cost of adding to end of an ArrayList?
  - Cost is  $O(1)$  using *amortized* analysis.

10

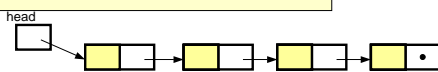
## Linked List Implementation of Stack

```
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

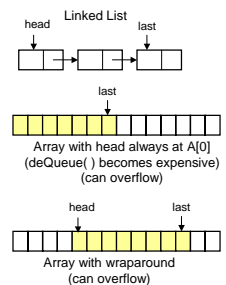


11

## Queue Implementations

- Recall: operations are *enqueue*, *dequeue*, *peek*,...
- Possible implementations

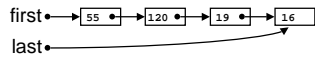
- For linked-list
  - All operations are  $O(1)$
- For array with head at  $A[0]$ 
  - *dequeue* takes time  $O(n)$
  - Other ops are  $O(1)$
  - Can overflow
- For array with wraparound
  - All operations are  $O(1)$
  - Can overflow



12

## Stacks and Queues as Lists

- Stack (LIFO) implemented as list
  - `put()`, `get()` from front of list
- Queue (FIFO) implemented as list
  - `put()` on back of list, `get()` from front of list



13

## 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?

14

## Goal: Design a Dictionary

- Operations

```

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

What building block data structure could we use to implement a dictionary?

Cost of each operation?

15

## 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

16

## Hashing

- Idea: "convert" an arbitrary key into a bounded integer
- Use a *hash function*  $h$ 
  - $U$  is the universe of keys
  - $h: U \rightarrow [0, \dots, m-1]$  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

17

## A Hashing Example

- A very simple hash function: count # of letters in string
- How do we resolve collisions?
  - use *chaining*: each table position is the head of a list
  - for any particular problem, this *might* work terribly

- 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

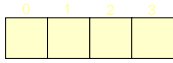
- In practice, using a good hash function, we can assume each position is equally likely

18

### Hash table example

$h(\text{"january"})=105$     $h(\text{"april"})=160$     $h(\text{"july"})=123$   
 $h(\text{"february"})=623$     $h(\text{"may"})=147$     $h(\text{"august"})=137$   
 $h(\text{"march"})=139$     $h(\text{"june"})=108$     $h(\text{"september"})=109$

$m=4$   
 $n=0$   
 $\lambda=n/m=0$   
 $\lambda_0=0.7$

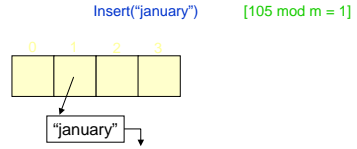


19

### Hash table example

$h(\text{"january"})=105$     $h(\text{"april"})=160$     $h(\text{"july"})=123$   
 $h(\text{"february"})=623$     $h(\text{"may"})=147$     $h(\text{"august"})=137$   
 $h(\text{"march"})=139$     $h(\text{"june"})=108$     $h(\text{"september"})=109$

$m=4$   
 $n=1$   
 $\lambda=n/m=0.25$   
 $\lambda_0=0.7$

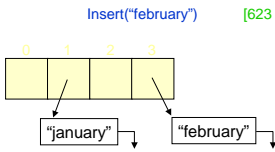


20

### Hash table example

$h(\text{"january"})=105$     $h(\text{"april"})=160$     $h(\text{"july"})=123$   
 $h(\text{"february"})=623$     $h(\text{"may"})=147$     $h(\text{"august"})=137$   
 $h(\text{"march"})=139$     $h(\text{"june"})=108$     $h(\text{"september"})=109$

$m=4$   
 $n=2$   
 $\lambda=n/m=0.5$   
 $\lambda_0=0.7$

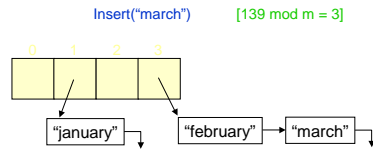


21

### Hash table example

$h(\text{"january"})=105$     $h(\text{"april"})=160$     $h(\text{"july"})=123$   
 $h(\text{"february"})=623$     $h(\text{"may"})=147$     $h(\text{"august"})=137$   
 $h(\text{"march"})=139$     $h(\text{"june"})=108$     $h(\text{"september"})=109$

$m=4$   
 $n=3$   
 $\lambda=n/m=0.75$   
 $\lambda_0=0.7$

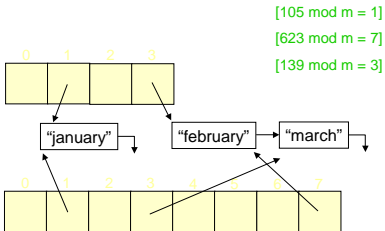


22

### Hash table example

$h(\text{"january"})=105$     $h(\text{"april"})=160$     $h(\text{"july"})=123$   
 $h(\text{"february"})=623$     $h(\text{"may"})=147$     $h(\text{"august"})=137$   
 $h(\text{"march"})=139$     $h(\text{"june"})=108$     $h(\text{"september"})=109$

$m=8$   
 $n=3$   
 $\lambda=n/m=0.375$   
 $\lambda_0=0.7$

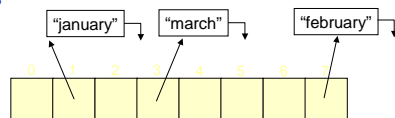


23

### Hash table example

$h(\text{"january"})=105$     $h(\text{"april"})=160$     $h(\text{"july"})=123$   
 $h(\text{"february"})=623$     $h(\text{"may"})=147$     $h(\text{"august"})=137$   
 $h(\text{"march"})=139$     $h(\text{"june"})=108$     $h(\text{"september"})=109$

$m=8$   
 $n=3$   
 $\lambda=n/m=0.375$   
 $\lambda_0=0.7$

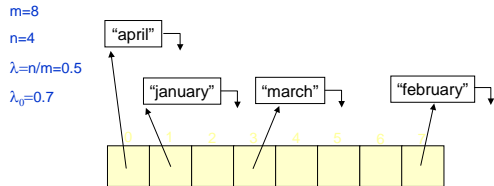


24

### Hash table example

$h(\text{"january"})=105$     $h(\text{"april"})=160$     $h(\text{"july"})=123$   
 $h(\text{"february"})=623$     $h(\text{"may"})=147$     $h(\text{"august"})=137$   
 $h(\text{"march"})=139$     $h(\text{"june"})=108$     $h(\text{"september"})=109$

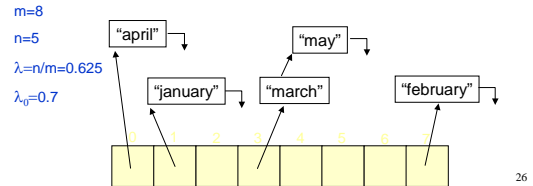
Insert("april")    $[160 \bmod m = 0]$



### Hash table example

$h(\text{"january"})=105$     $h(\text{"april"})=160$     $h(\text{"july"})=123$   
 $h(\text{"february"})=623$     $h(\text{"may"})=147$     $h(\text{"august"})=137$   
 $h(\text{"march"})=139$     $h(\text{"june"})=108$     $h(\text{"september"})=109$

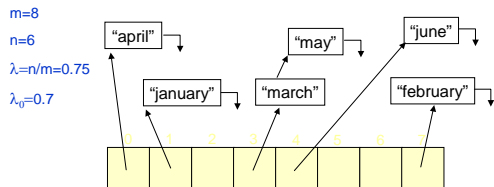
Insert("may")    $[147 \bmod m = 3]$



### Hash table example

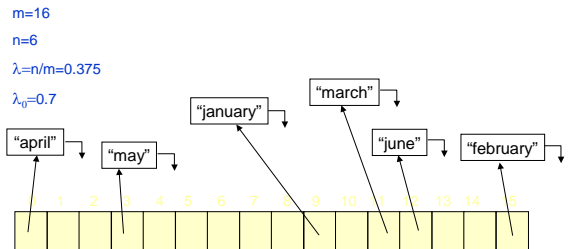
$h(\text{"january"})=105$     $h(\text{"april"})=160$     $h(\text{"july"})=123$   
 $h(\text{"february"})=623$     $h(\text{"may"})=147$     $h(\text{"august"})=137$   
 $h(\text{"march"})=139$     $h(\text{"june"})=108$     $h(\text{"september"})=109$

Insert("june")    $[108 \bmod m = 4]$



### Hash table example

$h(\text{"january"})=105$     $h(\text{"april"})=160$     $h(\text{"july"})=123$   
 $h(\text{"february"})=623$     $h(\text{"may"})=147$     $h(\text{"august"})=137$   
 $h(\text{"march"})=139$     $h(\text{"june"})=108$     $h(\text{"september"})=109$



### 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)
- $U$  = the expected # of probes for an *unsuccessful* search
  - $U$  = average number of items per table position =  $n/m = \lambda$ .
- $S$  = expected number of probes for a *successful* search
  - $S = 1 + \lambda/2 = O(\lambda)$

29

### Table Doubling

- We know each operation takes time  $O(\lambda)$  where  $\lambda = n/m$ 
  - But isn't  $\lambda = O(n)$ ?
- What's the deal here? It's still linear time!
- A solution: Table Doubling:
  - Set a bound for  $\lambda$  (call it  $\lambda_0$ )
  - Whenever  $\lambda$  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

30

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

31

## 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  $O(1)$ 
  - 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

32

## 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;  $39 * \text{previous} + \text{next value}$

33

## `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

34

## 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

35

## Detour: Data Locality

- Reading / writing memory usually faster for local accesses.
  - Locality == related data (often accessed at same time) exists in small neighborhood of memory.

- Analogy: grocery shopping

- Supermarket vs. Butcher + Bakery + Vegetable Stand + ...
- Supermarket organization

- Reason has to do with *caches* on modern processors and memory prefetching.

- Computer architecture class for details.

36

## Locality and Hashtable Chaining

Note: hashtable with chaining not guaranteed to have any locality. Alternative?

37

## Linear & Quadratic Probing

- All data is stored directly within the hash table array
- Linear Probing
  - Probe at  $h(X)$ , then at  $h(X) + 1, h(X) + 2, \dots, h(X) + i$
  - 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  $h(X) + 1, h(X) + 4, h(X) + 9, \dots, h(X) + i^2$
  - Works well when
    - $\lambda < 0.5$
    - Table size is prime

38

## Hashtable Pitfalls

- Good hash function is required
  - Test to check how many collisions you have in practice!
- Watch the load factor ( $\lambda$ ), especially for Linear & Quadratic Probing

39

## Dictionary Implementations

- Ordered Array
  - Better than unordered array because Binary Search can be used
- Unordered Linked-List
  - Ordering doesn't help
- Binary Tree
  - All operations are  $O(\log n)$  IF data not inserted in sorted order
- Hashtables
  - $O(1)$  expected time for Dictionary operations

40