Basic Data Structures

CS211
Fall 2000

Limitations of Runtime Analysis

- Big-O can hide a large constant
  - Example: Selection
  - Example: small problems
- The problem you want to solve may not be the worst case
  - Example: Simplex method for linear programming
- Your program may not be run often enough to make analysis worthwhile
  - Example: one-shot vs. every day
- You may be analyzing and improving the wrong part of the program
  - Common situation
  - Should use *profiling* tools

2

Why Bother with Runtime Analysis?

- Computers are so fast these days that we can do whatever we want using just simple algorithms and data structures, can't we?
- Well…not really; data-structure/algorithm improvements can be a very *big* win
- Scenario:
  - A runs in n^2 msec
  - A' runs in $n^2/10$ msec
  - B runs in 10 n log n msec
- Problem of size $n=10^3$
  - A: 10^3 sec ≈ 17 minutes
  - A': 10^2 sec ≈ 1.7 minutes
  - B: 10^2 sec ≈ 1.7 minutes
- Problem of size $n=10^6$
  - A: 10^9 sec ≈ 30 years
  - A': 10^8 sec ≈ 3 years
  - B: 2×10^5 sec ≈ 2 days

1 day = 86,400 sec ≈ 10^5 sec
1,000 days ≈ 3 years

3

Strategies for Programming Problems

Goal: Make it easier to solve programming problems

- Basic Data Structures
  - I recognize this; I can use this well-known data structure
  - Examples: Stack, Queue, Priority Queue, Hashtable, Binary Search Tree
- Algorithm Design Methods
  - I can design an algorithm to solve this
  - Examples: Divide & Conquer, Greedy, Dynamic Programming
- Problem Reductions
  - I can change this problem into another with a known solution
  - Or, I can show that a reasonable algorithm is most-likely impossible
  - Examples: reduction to network flow, NP-complete problems

4

Recall: Use of Objects Encourages…

- Abstraction
  - Avoid details
  - Distill down to fundamental parts
- Encapsulation
  - Information hiding
  - Our code depends on the available operations, but *not* on how they are implemented

Why use these ideas?
- *Basically, because they seem to help*
- *Result in clean interfaces, easier modification, portable code*

5

Abstract Data Types (ADTs)

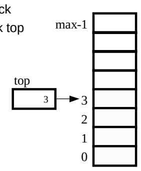
- A method for achieving abstraction for data structures and algorithms
- ADT = model + operations
- Describes what each operation does, but not how it does it
- An ADT is independent of its implementation
- In Java, an *interface* corresponds well to an ADT
  - The interface describes the operations, 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 ( );
}
```

6

Array Implementation of Stack

```
class StackArray implements Stack {
    Object [ ] s;                              // Holds the stack
    int top;                                   // Index of stack top
    public StackArray(int max)                 // Constructor
        {s = new Object [max]; top = -1;}
    public void push (Object item) {s [++top] = item;}
    public Object pop ( ) {return s [top – –];}
    public Object peek ( ) {return s [top];}
    public boolean isEmpty( ) {return top == -1;}
    public void makeEmpty( ) {top = -1;}
    }
// Better for garbage collection if makeEmpty( ) also cleared the array
```



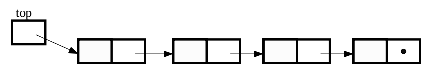
O(1) worst-case time for each operation

Linked List Implementation of Stack

```
class StackLinked implements Stack {
    class Node {Object data; Node next; // An inner class
        Node (Object d, Node n)          // Constructor for Node
            {data = d; next = n;}
        }
    Node top;                            // Top Node of stack
    public StackLinked ( ) {top = null;} // Constructor
    public void push (Object item) {top = new Node(item,top);}
    public Object pop ( ) {
        Object temp = top.data; top = top.next; return temp;}
    public boolean isEmpty ( ) {return top == null;}
    public void makeEmpty ( ) {top = null;}
}
```

O(1) worst-case time for each operation

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

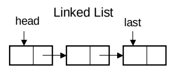


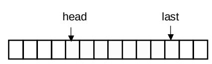
ADT Queue

Operations:
- void enQueue (Object x);
- Object deQueue ( );
- Object peek ( )
- boolean isEmpty ( );
- void makeEmpty ( );

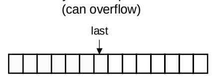
▲ Text uses getFront( ) instead of peek( )

Possible implementations





Array with wraparound
(can overflow)



Array with head always at A[0]
(deQueue( ) becomes expensive)
(can overflow)

ADT Dictionary

Operations:

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

Think of
key = word; value = definition

- Uses:
  - Symbol tables
  - Wide use within other algorithms

Array implementations

	sorted	unsorted
insert	O(n)	O(1)
update	O(log n)	O(n)
find	O(log n)	O(n)
remove	O(n)	O(n)
makeEmpty	O(1) or O(n)	O(1) or O(n)

- Update( ) and find ( ) use Binary Search
- Can use Binary Search for remove( ), but then find( ) doesn't work
- Can use special "removed" mark to make find( ) work again

Multiple Dictionary Implementations

	insert	**update**	**find**	**remove**	
Hashtable	O(1)	O(1)	O(1)	O(1)	expected
Binary Search Tree (BST)	O(log n)	O(log n)	O(log n)	O(log n)	expected (sort of)
Balanced Tree	O(log n)	O(log n)	O(log n)	O(log n)	worst-case

ADT Priority Queue

Operations:
- void insert (Object x);
- Object getMax ( );
- boolean isEmpty ( );

Uses:
- Job scheduler
- Event-driven simulation
- Wide use within other algorithms

	insert	**getMax**
Linked List	O(1)	O(n)
Unsorted Array	O(1)	O(n)
Sorted Array	O(n)	O(1)
Heap	O(log n)	O(log n)