Stacks & Queues
and
Asymptotic Analysis

CSE 373
Data Structures & Algorithms
Ruth Anderson
Spring 2008

Today’s Outline

• Admin: Office hours, etc.
• Stacks and Queues
• Asymptotic analysis

Office Hours, etc.

Ruth Anderson (in CSE 360)
M 12:30-1:30, T 1:30-2:30, or by appointment

Tian Sang (in CSE 220)
W & Th 4:30-5:30pm

Devy Pranowo (in CSE 218)
W 1:30-2:30pm

Eric McCambridge (in CSE 218)
Th 1:30-2:30

Project 1 – Sound Blaster!

Play your favorite song in reverse!

Aim:
1. Implement stack ADT two different ways
2. Use to reverse a sound file

Due: Thurs, April 10, 2008
Electronic: at 11:59pm
Hardcopy: in lecture at 11:30am on Friday April 11.

Stacks & Queues

First Example: Queue ADT

• Queue operations
  create
destroy
enqueue
dequeue
is_empty

G enqueue F E D C B dequeue A
Circular Array Queue Data Structure

1. enqueue(Object x) {
   Q[back] = x;
   back = (back + 1) % size;
}

2. dequeue() {
   x = Q[front];
   front = (front + 1) % size;
   return x;
}

How to test for empty list?
How to find K-th element in the queue?
What is complexity of these operations?
Limitations of this structure?

Linked List Queue Data Structure

1. void enqueue(Object x) {
   if (is_empty())
      front = back = new Node(x);
   else
      back->next = new Node(x);
      back = back->next;
}

2. bool is_empty() {
   return front == null;
}

3. Object dequeue() {
   assert(!is_empty);
   return_data = front->data;
   temp = front;
   front = front->next;
   delete temp;
   return return_data;
}

How test for empty list?
How to find K-th element in the queue?
What is complexity of these operations?
Limitations of this structure?

Circular Array vs. Linked List

Second Example: Stack ADT

- Stack operations
  - create
  - destroy
  - push
  - pop
  - top
  - is_empty

Stacks in Practice

- Function call stack
- Removing recursion
- Balancing symbols (parentheses)
- Evaluating Reverse Polish Notation

Asymptotic Analysis
Comparing Two Algorithms

What we want

- Rough Estimate
- Ignores Details

Big-O Analysis

- Ignores “details”

Analysis of Algorithms

- Efficiency measure
  - how long the program runs time complexity
  - how much memory it uses space complexity
  - For today, we’ll focus on time complexity only

Why analyze at all?

Asymptotic Analysis

- Complexity as a function of input size \( n \)
  - \( T(n) = 4n + 5 \)
  - \( T(n) = 0.5n \log n - 2n + 7 \)
  - \( T(n) = 2^n + n^3 + 3n \)

- What happens as \( n \) grows?

Why Asymptotic Analysis?

- Most algorithms are fast for small \( n \)
  - Time difference too small to be noticeable
  - External things dominate (OS, disk I/O, …)

- BUT \( n \) is often large in practice
  - Databases, internet, graphics, …

- Time difference really shows up as \( n \) grows!
**Big-O: Common Names**

- constant: \( O(1) \)
- logarithmic: \( O(\log n) \)
- linear: \( O(n) \)
- quadratic: \( O(n^2) \)
- cubic: \( O(n^3) \)
- polynomial: \( O(n^k) \) (\( k \) is a constant)
- exponential: \( O(c^n) \) (\( c \) is a constant > 1)

**Exercise**

```java
bool ArrayFind(int array[], int n, int key){
    // Insert your algorithm here
}
```

What algorithm would you choose to implement this code snippet?

**Analyzing Code**

**Basic Java operations**
- Constant time
- Sum of times
- Larger branch plus test
- Sum of iterations
- Cost of function body
- Solve recurrence relation

**Solving Recurrence Relations**

1. Determine the recurrence relation. What is the base case(s)?
2. “Expand” the original relation to find an equivalent general expression in terms of the number of expansions.
3. Find a closed-form expression by setting the number of expansions to a value which reduces the problem to a base case