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

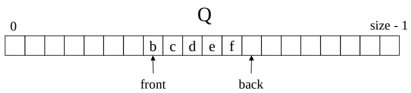
Due: Thurs, April 10, 2008
Electronic: at 11:59pm
Hardcopy: in lecture at 11:30am on Friday April 11.

Stacks & Queues

First Example: Queue ADT

- Queue operations
  - create
  - destroy
  - enqueue
  - dequeue
  - is_empty

Circular Array Queue Data Structure




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

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

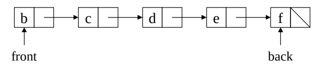
How test for empty list?

How to find K-th element in the queue?

What is complexity of these operations?

Limitations of this structure?

Linked List Queue Data Structure



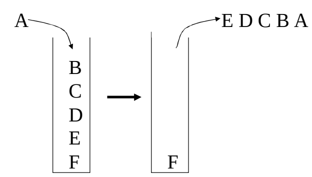

```
void enqueue(Object x) {
   if (is_empty())
        front = back = new Node(x)
   else
        back->next = new Node(x)
        back = back->next
}
bool is_empty() {
   return front == null
}
```

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


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

04/02/08 13

What we want

- Rough Estimate
- Ignores Details

04/02/08 14

Big-O Analysis

- Ignores "details"

04/02/08 15

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?*

04/02/08 16

Asymptotic Analysis

- Complexity as a function of input size n
  - $T(n) = 4n + 5$
  - $T(n) = 0.5\, n \log n - 2n + 7$
  - $T(n) = 2^n + n^3 + 3n$
- *What happens as n grows?*

04/02/08 17

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!

04/02/08 18

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

2	3	5	16	37	50	73	75	126

```
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
Consecutive statements	Sum of times
Conditionals	Larger branch plus test
Loops	Sum of iterations
Function calls	Cost of function body
Recursive functions	Solve recurrence relation

Analyze your code!

Linear Search Analysis

```
bool LinearArrayFind(int array[],
                     int n,
                     int key ) {
  for( int i = 0; i < n; i++ ) {
     if( array[i] == key )
            // Found it!
            return true;
  }
  return false;
}
```

Best Case:

Worst Case:

Binary Search Analysis

```
bool BinArrayFind( int array[], int low,
                   int high, int key ) {
  // The subarray is empty
  if( low > high ) return false;

  // Search this subarray recursively
  int mid = (high + low) / 2;
  if( key == array[mid] ) {
      return true;
  } else if( key < array[mid] ) {
      return BinArrayFind( array, low,
                           mid-1, key );
  } else {
      return BinArrayFind( array, mid+1,
                           high, key );
}
```

Best case:

Worst case:

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