## Come up and say hello!



## CPSC 221: Algorithms and Data Structures Lecture #0: Introduction

Steve Wolfman

2014W1



## Fibonacci

0 +  = 

1, 1, 2, 3, 5, 8, 13, 21, …

Applications, in order of importance:

- Fun for CSists
- Brief appearance in Da Vinci Code
- Endlessly abundant in nature:
  http://www.youtube.com/user/Vihart?feature=g-u#p/u/1/ahXIMUkSXX0



## Fibonacci

0 +  = 

Definition:

$$Fib(n) = \begin{cases} 1 & if\ n = 0\ or\ n = 1 \\ Fib(n-1) + Fib(n-2) & otherwise \end{cases}$$



## Fibonacci

0 +  = 

Easy to implement as recursive code, right?

Let's try it.

(Then, we'll open up a big can of data structures and algorithms on that problem.)



## Fibonacci

0 +  = 

(Exact) Approximation:

$$Fib(n) = \left\lfloor \frac{\varphi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor$$

Where (golden ratio): $\varphi = \frac{1+\sqrt{5}}{2}$

But how long to raise phi to the n power?

What algorithm?

## Today's Outline

- Administrative Cruft
- Overview of the Course
- Queues
- Stacks



## Course Information

- Your Instructor: Steve Wolfman
  ICCS 239
  wolf@cs.ubc.ca
  Office hours: see website
- Other Instructor: Kendra Cooper (OHs: see website)
- Tas and their office hours: see website, more may be posted!
- Texts: Epp Discrete Mathematics, Koffman C++
  (But… feel free to get alternate texts or versions)



## Course Policies

- No late work; may be flexible with advance notice
  (why? so we can post solutions/discussion quickly!)
- Programming projects (~3?) due 9PM, usually Fridays
- Quizzes (~5?) online on Fridays (due time TBD)
- Written corrections of quizzes due 5PM the following Wed
- Grading
  - labs: 10%
  - quizzes/written assns: 15%
  - programming projects: 15%
  - midterms: 25%
  - final: 30%
  - your best of the above: 5%

Must pass the final to pass the course.



## Collaboration

**READ** the collaboration policy on the website.
You have **LOTS** of freedom to collaborate!
Use it to learn and have fun while doing it!

**Don't violate the collaboration policy.** There's no point in doing so, and the penalties are so severe that just thinking about them causes babies to cry*.



*Almost anything causes babies to cry, actually, but the cheating penalties really are severe.

## Course Mechanics

- 221 Web page: www.ugrad.cs.ubc.ca/~cs221
- 221 Piazza site at piazza.com
- Quizzes and grades on Connect (sorry!)
- Labs are in ICCS X350 and X251
  - use the "Linux" logon to the computers
- All programming projects graded on UNIX/g++



## Today's Outline

- Administrative Cruft
- Overview of the Course
- Queues
- Stacks

## Observation

- All programs manipulate data
  - programs *process, store, display, gather*
  - data can be *numbers, images, sound (*information!*)*
- Each program must decide how to store and manipulate data
- Choice influences program at every level
  - execution speed
  - memory requirements
  - maintenance (debugging, extending, etc.)

How you structure your data *matters* to every program you create!



## Goals of the Course

- Become familiar with some of the fundamental data structures and algorithms in computer science
- Improve ability to solve problems abstractly
  - data structures and algorithms are the building blocks
- Improve ability to analyze your algorithms
  - prove correctness
  - gauge, compare, and improve time and space complexity
- Become modestly skilled with C++ and UNIX, but this is ***largely on your own***!



## What is an Abstract Data Type?

Abstract Data Type (ADT) – a mathematical description of an object and the set of operations on the object.

Maybe more usefully: a description of how a data structure works.. which could be implemented by many different actual data structures.



## Data Structures as Algorithms

- Algorithm
  - A high level, language independent description of a step-by-step process for solving a problem
- Data Structure
  - A set of algorithms which implement an ADT
    (well… and a bit more like the state of the structure)



## Why so many data structures?

Ideal data structure:

fast, elegant, memory efficient

Generates tensions:

- time *vs.* space
- performance *vs.* elegance
- generality *vs.* simplicity
- one operation’s performance *vs.* another’s
- serial performance *vs.* parallel performance

“Dictionary” or “Map” ADT

- list
- binary search tree
- AVL tree
- Splay tree
- B+ tree
- Red-Black tree
- hash table
- concurrent hash table
- …



## Code Implementation

- Theoretically
  - abstract base class describes ADT
  - inherited implementations implement data structures
  - can change data structures transparently (to client code)
- Practice
  - different implementations sometimes suggest different interfaces (generality *vs.* simplicity)
  - performance of a data structure may influence form of client code (time *vs.* space, one operation *vs.* another)

# ADT Presentation Algorithm

- Present an ADT
- Motivate with some applications
- Repeat until browned entirely through
  - develop a data structure for the ADT
  - analyze its properties
    - efficiency
    - correctness
    - limitations
    - ease of programming
- Contrast data structure's strengths and weaknesses
  - understand when to use each one

19

# Today's Outline

- Administrative Cruft
- Overview of the Course
- Queues
- Stacks

20

# Queue ADT

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

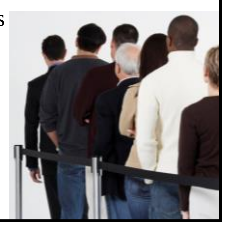

- Queue property:
  if x is enqueued before y is enqueued,
  then x will be dequeued before y is dequeued.

  FIFO: First In First Out

21

# Applications of the Q

- Store people waiting to deposit their paycheques at a bank (*historical note*: people used to do this!)
- Hold jobs for a printer
- Store packets on network routers
- Hold memory "freelists"
- Make UBC's waitlists fair!
- Breadth first search



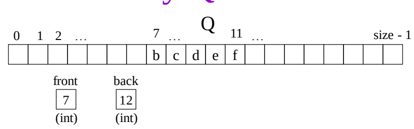
# Abstract Q Example

enqueue R
enqueue O
dequeue
enqueue T
enqueue A
enqueue T
dequeue
dequeue
enqueue E
dequeue

In order, what letters are dequeued?
(Can we tell, just from the ADT?)

23

# Circular Array Q Data Structure



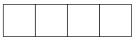

```
void enqueue(char x) {
   Q[back] = x
   back = (back + 1) % size
}
char dequeue() {
   x = Q[front]
   front = (front + 1) % size
   return x
}
```

```
bool is_empty() {
   return (front == back)
}
bool is_full() {
   return front ==
        (back + 1) % size
}
```

This is *pseudocode*. Do not correct my semicolons ☺
But.. is there anything *else* wrong?

24

## Circular Array Q Example



enqueue R
enqueue O
dequeue
enqueue T
enqueue A
enqueue T
dequeue
dequeue
enqueue E
dequeue

What are the final contents of the array?



## Circular Array Q Example

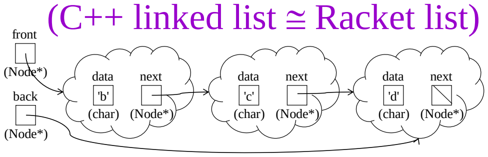
enqueue R
enqueue O
dequeue
enqueue T
enqueue A
enqueue T
dequeue
dequeue
enqueue E
dequeue

Assuming we can distinguish full and empty (could add a boolean)…

What are the final contents of the array?



## Linked List Q Data Structure (C++ linked list ≅ Racket list)



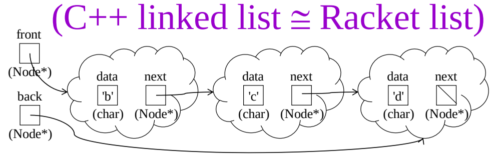

```
void enqueue(char x) {
  if (is_empty()) {
    front = back = new Node(x);
  } else {
    back->next = new Node(x);
    back = back->next;
  }
}
```

```
char dequeue() {
  assert(!is_empty());
  char result = front->data;
  Node * temp = front;
  front = front->next;
  delete temp;
  return result;
}
bool is_empty() {
  return front == NULL;
```

This is *not* pseudocode.
Let's draw a *memory diagram* of how it works!



## Linked List Q Data Structure (C++ linked list ≅ Racket list)




```
void enqueue(char x) {
  if (is_empty()) {
    front = back = new Node(x);
  } else {
    back->next = new Node(x);
    back = back->next;
  }
}
```

```
char dequeue() {
  assert(!is_empty());
  char result = front->data;
  Node * temp = front;
  front = front->next;
  delete temp;
  return result;
}
bool is_empty() {
  return front == NULL;
}
```

What's with the red code?
Manual memory management!
Tip: "a `delete` for every `new`"



## Circular Array vs. Linked List

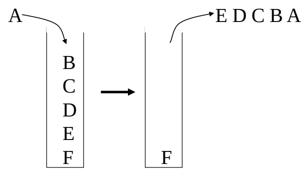


## Today's Outline

- Administrative Cruft
- Overview of the Course
- Queues
- Stacks

# Stack ADT

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




- Stack property: if x is pushed before y is pushed, then x will be popped after y is popped

LIFO: Last In First Out



# Stacks in Practice

- Store pancakes on your plate (does it bother you that you eat them in the opposite order they were put down?)
- Function call stack
- Implementing/removing recursion
- Balancing symbols (parentheses)
- Evaluating Reverse Polish Notation
- Depth first search

# Example Stolen from Alan Hu ☺ “Call Stack” and Recursion

```
int fib(int n) {
1. if (n <= 2) return 1;
2. int a = fib(n-1);
3. int b = fib(n-2);
4. return a+b;
}
```

Suppose we call `fib(4)`:

This is our stack. It’s a stack of program points and variable values.

Line 1, n=4
(location of `fib(4)` call *goes here*)



# “Call Stack” and Recursion

```
int fib(int n) {
1. if (n <= 2) return 1;
2. int a = fib(n-1);
3. int b = fib(n-2);
4. return a+b;
}
```

Suppose we call `fib(4)`:

Making a function call *pushes* a new “frame” onto the stack.

Line 2, n=4, a = *fib(3)*
(location of `fib(4)` call *goes here*)



# “Call Stack” and Recursion

```
int fib(int n) {
1. if (n <= 2) return 1;
2. int a = fib(n-1);
3. int b = fib(n-2);
4. return a+b;
}
```

Suppose we call `fib(4)`:

Making a function call *pushes* a new “frame” onto the stack.

Line 1, n=3
Line 2, n=4, a = *fib(3)*
(location of `fib(4)` call *goes here*)



# “Call Stack” and Recursion

```
int fib(int n) {
1. if (n <= 2) return 1;
2. int a = fib(n-1);
3. int b = fib(n-2);
4. return a+b;
}
```

Suppose we call `fib(4)`:

Line 2, n=3, a = *fib(2)*
Line 2, n=4, a = *fib(3)*
(location of `fib(4)` call *goes here*)

## “Call Stack” and Recursion

```
int fib(int n) {
1. if (n <= 2) return 1;
2. int a = fib(n-1);
3. int b = fib(n-2);
4. return a+b;
}
```

Suppose we call `fib(4)`:

Line 1, n=2
Line 2, n=3, a = *fib(2)*
Line 2, n=4, a = *fib(3)*
(location of `fib(4)` call *goes here*) 37

## “Call Stack” and Recursion

```
int fib(int n) {
1. if (n <= 2) return 1;
2. int a = fib(n-1);
3. int b = fib(n-2);
4. return a+b;
}
```

Suppose we call `fib(4)`:

Returning from a call *pops* the old “frame” off the stack.

Line 1, n=2, return 1
Line 2, n=3, a = *fib(2)*
Line 2, n=4, a = *fib(3)*
(location of `fib(4)` call *goes here*) 38

## “Call Stack” and Recursion

```
int fib(int n) {
1. if (n <= 2) return 1;
2. int a = fib(n-1);
3. int b = fib(n-2);
4. return a+b;
}
```

Suppose we call `fib(4)`:

Returning from a call *pops* the old “frame” off the stack.

Line 2, n=3, a = 1
Line 2, n=4, a = *fib(3)*
(location of `fib(4)` call *goes here*) 39

## “Call Stack” and Recursion

```
int fib(int n) {
1. if (n <= 2) return 1;
2. int a = fib(n-1);
3. int b = fib(n-2);
4. return a+b;
}
```

Suppose we call `fib(4)`:

Line 3, n=3, a = 1, b = *fib(1)*
Line 2, n=4, a = *fib(3)*
(location of `fib(4)` call *goes here*) 40

## “Call Stack” and Recursion

```
int fib(int n) {
1. if (n <= 2) return 1;
2. int a = fib(n-1);
3. int b = fib(n-2);
4. return a+b;
}
```

Suppose we call `fib(4)`:

Line 1, n=1
Line 3, n=3, a = 1, b = *fib(1)*
Line 2, n=4, a = *fib(3)*
(location of `fib(4)` call *goes here*) 41

## “Call Stack” and Recursion

```
int fib(int n) {
1. if (n <= 2) return 1;
2. int a = fib(n-1);
3. int b = fib(n-2);
4. return a+b;
}
```

Suppose we call `fib(4)`:

Line 1, n=1, return 1
Line 3, n=3, a = 1, b = *fib(1)*
Line 2, n=4, a = *fib(3)*
(location of `fib(4)` call *goes here*) 42

## "Call Stack" and Recursion

```
int fib(int n) {
1. if (n <= 2) return 1;
2. int a = fib(n-1);
3. int b = fib(n-2);
4. return a+b;
}
```

Suppose we call `fib(4)`:

Line 3, n=3, a = 1, b = 1
Line 2, n=4, a = *fib(3)*
(location of `fib(4)` call *goes here*) 43

## "Call Stack" and Recursion

```
int fib(int n) {
1. if (n <= 2) return 1;
2. int a = fib(n-1);
3. int b = fib(n-2);
4. return a+b;
}
```

Suppose we call `fib(4)`:

Line 4, n=3, a = 1, b = 1, return 2
Line 2, n=4, a = *fib(3)*
(location of `fib(4)` call *goes here*) 44

## "Call Stack" and Recursion

```
int fib(int n) {
1. if (n <= 2) return 1;
2. int a = fib(n-1);
3. int b = fib(n-2);
4. return a+b;
}
```

Suppose we call `fib(4)`:

Line 3, n=4, a = 2, b = *fib(2)*
(location of `fib(4)` call *goes here*) 45

## "Call Stack" and Recursion

```
int fib(int n) {
1. if (n <= 2) return 1;
2. int a = fib(n-1);
3. int b = fib(n-2);
4. return a+b;
}
```

Suppose we call `fib(4)`:

Line 1, n=2
Line 3, n=4, a = 2, b = *fib(2)*
(location of `fib(4)` call *goes here*) 46

## "Call Stack" and Recursion

```
int fib(int n) {
1. if (n <= 2) return 1;
2. int a = fib(n-1);
3. int b = fib(n-2);
4. return a+b;
}
```

Suppose we call `fib(4)`:

Line 1, n=2, return 1
Line 3, n=4, a = 2, b = *fib(2)*
(location of `fib(4)` call *goes here*) 47

## "Call Stack" and Recursion

```
int fib(int n) {
1. if (n <= 2) return 1;
2. int a = fib(n-1);
3. int b = fib(n-2);
4. return a+b;
}
```

Suppose we call `fib(4)`:

Line 3, n=4, a = 2, b = 1
(location of `fib(4)` call *goes here*) 48

# "Call Stack" and Recursion

```
int fib(int n) {
1.  if (n <= 2) return 1;
2.  int a = fib(n-1);
3.  int b = fib(n-2);
4.  return a+b;
}
```

Suppose we call `fib(4)`:

Line 3, n=4, a = 2, b = 1, return 3

(location of `fib(4)` call *goes here*)



# "Call Stack" and Recursion

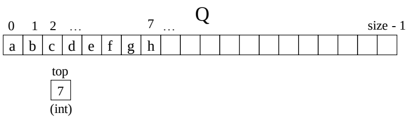
```
int fib(int n) {
1.  if (n <= 2) return 1;
2.  int a = fib(n-1);
3.  int b = fib(n-2);
4.  return a+b;
}
```

Suppose we call `fib(4)`:

(code that called `fib(4)` resumes w/value 3)



# Array Stack Data Structure



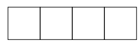

```
void push(char x) {
   assert(!is_full())
   S[top] = x
   top++
}
char top() {
   assert(!is_empty())
   return S[top - 1]
}
```

```
char pop() {
   assert(!is_empty())
   top--
   return S[top]
}
bool is_empty() {
   return top == 0
}
bool is_full() {
   return top == size
}
```



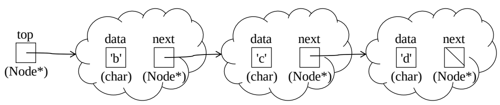
# Let's Trace How It Works

push B
pop
push K
push C
push A
pop
pop
pop





# Linked List Stack Data Structure




```
void push(char x) {
   temp = top;
   top = new Node(x);
   top->next = temp;
}
char top() {
   assert(!is_empty())
   return top->data;
}
```

```
char pop() {
   assert(!is_empty())
   char return_data = top->data;
   temp = top;
   top = top->next;
   delete temp;
   return return_data;
}
bool is_empty() {
   return top == nullptr;
}
```



# Let's Trace How It Works

push B
pop
push K
push C
push A
pop
pop
pop

## Data structures you should already know (a bit)

- Arrays
- Linked lists
- Trees
- Queues
- Stacks

55

## To Do

- Check out the web page and Piazza
- Download and read over Lab 1 materials
- Begin working through Chapters P and 1 of Koffman and Wolfgang (C++ background)
- Read 4.5-4.7, 5, and 6 (except 6.4) of Koffman and Wolfgang (linked lists, stacks, queues)
- ALWAYS when reading the text:
  - DO the exercises!
  - ASK questions!

56

## Coming Up

- Asymptotic Analysis
- Quiz/Written Assignment 1
- Programming Project 1

57