CPSC 221:
Algorithms and Data Structures
ADTs, Stacks, and Queues

Alan J. Hu
(Slides borrowed from Steve Wolfman)
Be sure to check course webpage!
http://www.ugrad.cs.ubc.ca/~cs221

Lab 1 available very soon!

- Instructions for Lab 1 will be posted on course webpage very soon:

  http://www.ugrad.cs.ubc.ca/~cs221

- Labs start on Monday.
- Read instructions and do any pre-labs before your lab section.

Today's Outline

- Abstract Data Types and Data Structures
- Queues
- Stacks
- Abstract Data Types vs. Data Structures

What is an Abstract Data Type?

Abstract Data Type (ADT) -

Formally:

Mathematical description of an object and the set of operations on the object

In Practice:

The interface of a data structure, without any information about the implementation

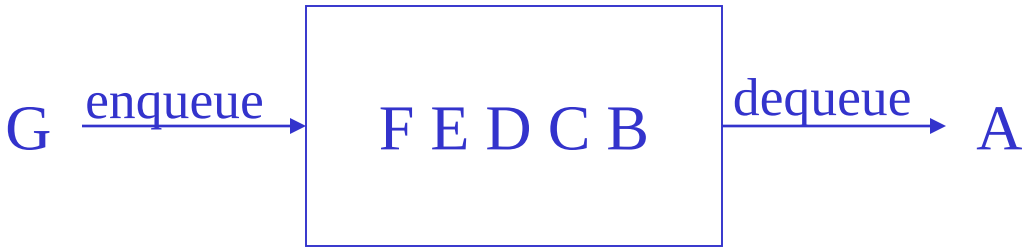
Data Structures

- 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

- Don’t get too obsessed with this distinction.
- Let’s look at some examples…

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

Why is it called a “queue”?

Applications of Queues

- Hold jobs for a printer
- Store packets on network routers
- Make waitlists fair
- Breadth first search
- Etc. etc. etc.
- Basically, any time you need to hold a bunch of stuff for a bit, where you want to keep them in order.

Abstract Queue Example

enqueue R

enqueue O

dequeue

enqueue T

enqueue A

enqueue T

dequeue

dequeue

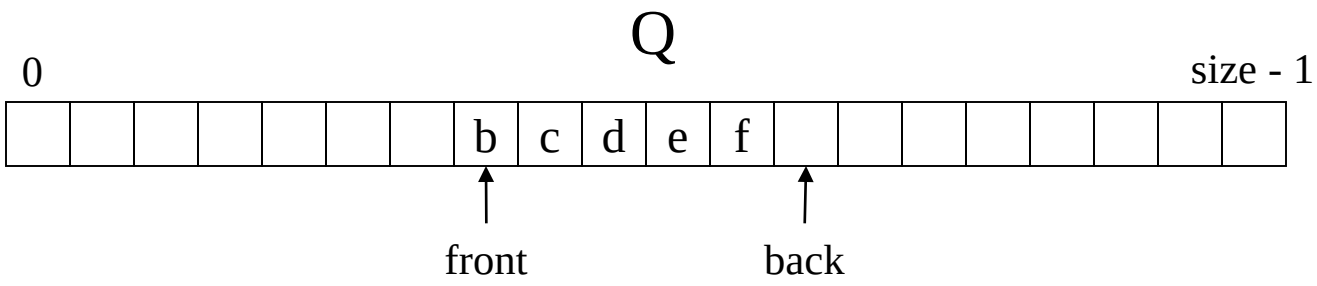
enqueue E

dequeue

Implementing Queues

- Many different ways to do this!
- What would you do?

Circular Array Q Data Structure



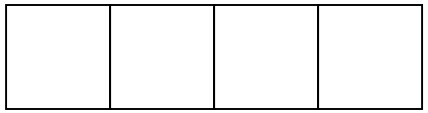

```
void enqueue(Object x) {
   Q[back] = x
   back = (back + 1) % size
}
Object dequeue() {
   x = Q[front]
   front = (front + 1) % size
   return x
}
```

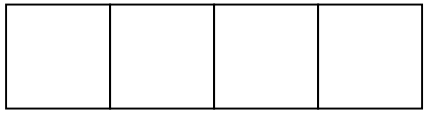
```
bool is_empty() {
   return (front == back)
}

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

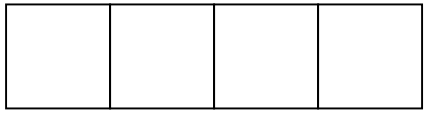
This is *pseudocode*. Do not correct my semicolons ☺

Circular Array Q Example

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



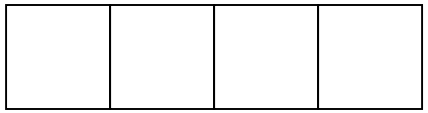


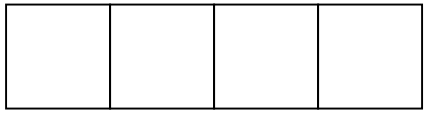


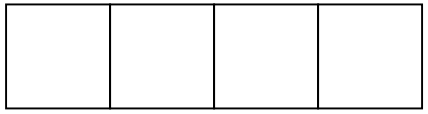
What are the final contents of the array?

Circular Array Q Example

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



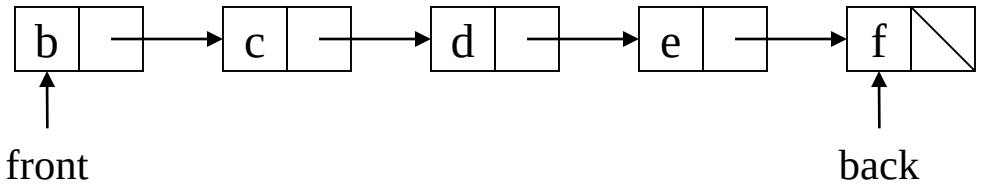




Assuming we can distinguish full and empty (could add a boolean)…

What are the final contents of the array?

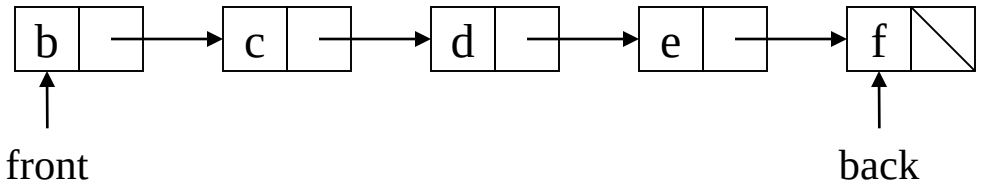
Linked List Q Data Structure




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

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

Linked List Q Data Structure




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

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

What's with the red text?

Circular Array vs. Linked List

- Which is better? Why?

Circular Array vs. Linked List

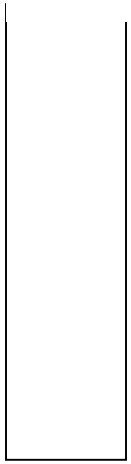
- Which is better? Why?

They both have plusses and minuses!

In general, many different data structures can implement an ADT, each with different trade-offs. You must pick the best for your needs.

Stack ADT

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



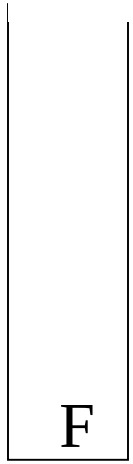
- Stack property: if x is pushed before y is pushed, then x will be popped after y is popped

LIFO: Last In First Out

Stack ADT

push F

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



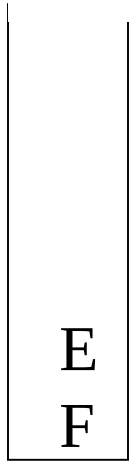
- Stack property: if x is pushed before y is pushed, then x will be popped after y is popped

LIFO: Last In First Out

Stack ADT

push E

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



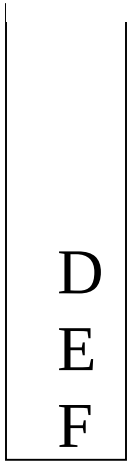
- Stack property: if x is pushed before y is pushed, then x will be popped after y is popped

LIFO: Last In First Out

Stack ADT

push D

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



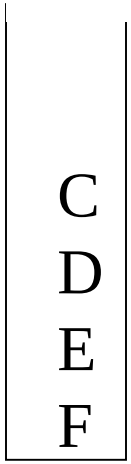
- Stack property: if x is pushed before y is pushed, then x will be popped after y is popped

LIFO: Last In First Out

Stack ADT

push C

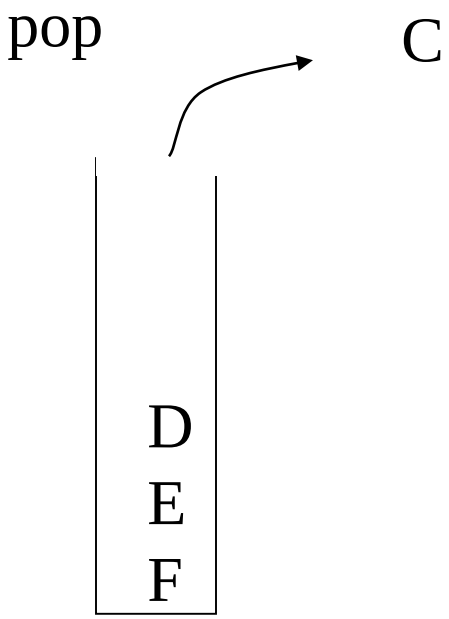
- Stack operations
  - create
  - destroy
  - push
  - pop
  - top
  - is_empty



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

LIFO: Last In First Out

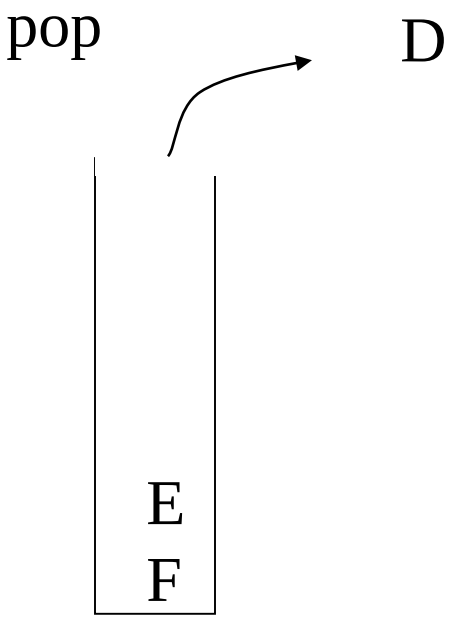
Stack ADT



- 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

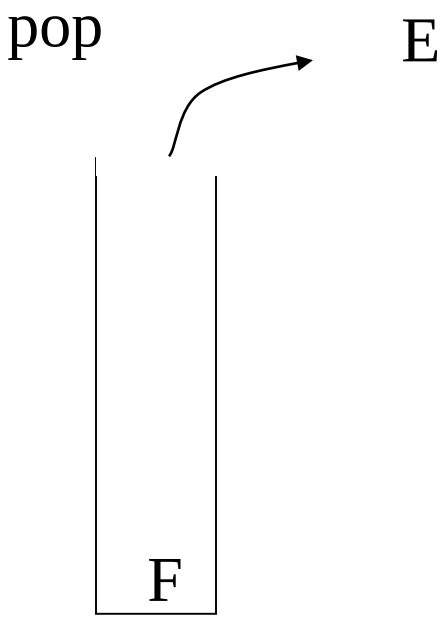
Stack ADT



- 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

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

Why use a stack?

Can you think of anything in real life where you want LIFO instead of FIFO?

Why use a stack?

Can you think of anything in real life where you want LIFO instead of FIFO?

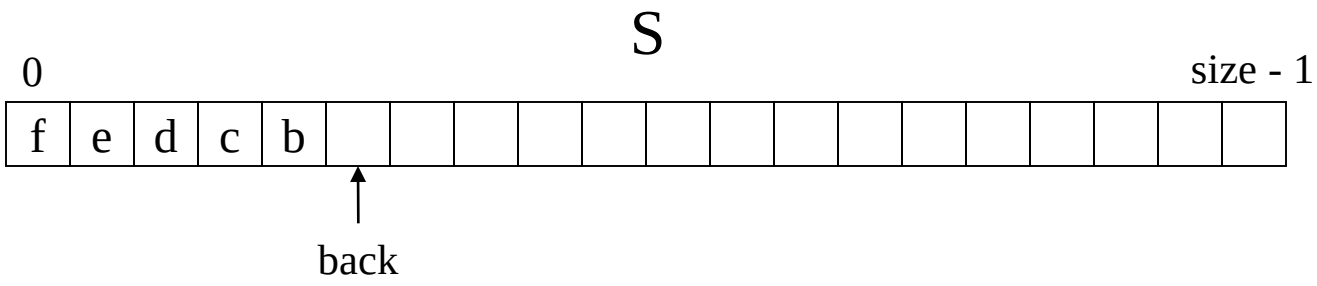
Handling interruptions?

Reversing the order of things?

Stacks in Practice

- Function call stack
- Removing recursion
- Balancing symbols (parentheses)
- Depth first search

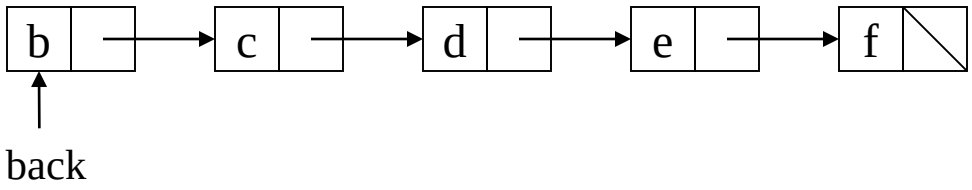
Array Stack Data Structure




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

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

Linked List Stack Data Structure




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

```
Object pop() {
   assert(!is_empty())
   return_data = back->data
   temp = back
   back = back->next
   delete temp
   return return_data
}
bool is_empty() {
   return back == null
}
```

Data structures you should already know (a bit)

- Arrays
- Linked lists
- Trees
- Queues
- Stacks

Abstract Data Types vs. Data Structures

- As mentioned before, ADT tells you what operations are available, but does not say anything about how implemented.
- Data structure consists of algorithms and memory layout to implement the ADT.
- Algorithms are language-independent. How does this map onto code?

ADTs vs. Data Structures in Code Implementation

- Theoretically
  - abstract base class (or Java interface) 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)

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

"Dictionary" ADT

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

CS 221 ADT Presentation Algorithm

- Present an ADT
- Motivate with some applications
- Repeat a bunch of times:
  - 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

Coming Up

- Asymptotic Analysis