CPSC 221
Basic Algorithms and Data Structures

May 13, 2015

Administrative stuff

Our TAs

Jason Chan
Devon Graham
Angad Kalra
Kamil Khan
Issam Hadj Laradji
Henry Li
Wei Luo
Kaitlyn Melton
Mushfiqur (Nasa) Rouf
Shu Yang
Sedigheh Zolaktaf
Zainab Zolaktaf

Administrative stuff

Labs

Collaboration: You will work in teams of two in the lab.

First lab is tomorrow. The lab will be posted this evening on UBC connect.

Read Chapters P and 1 of Objects, Abstraction, Data Structures and Design Using C++ before your first lab.

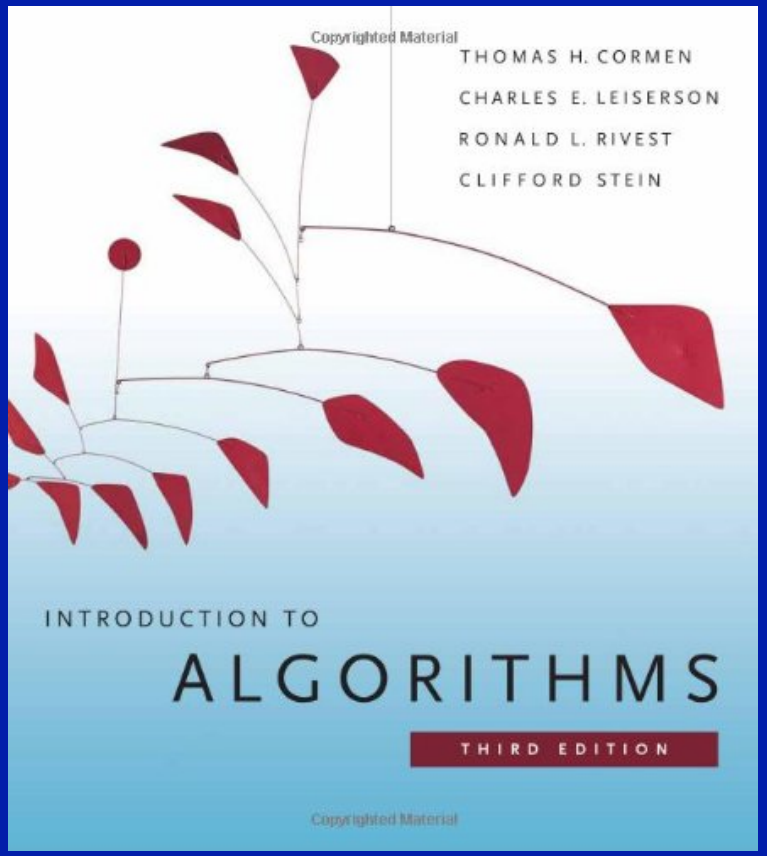
Administrative stuff

Language-independent algorithms text

Here's the gold standard:

Introduction to Algorithms
Cormen, Leiserson, Rivest
and Stein
MIT Press

There may be copies in
the CS reading room



Administrative stuff

Lecture slides

Published on UBC Connect in both PDF and PowerPoint format

Procedures and algorithms

There's a general commonly-used definition of an algorithm. Here's one version of many:

An algorithm is

- a finite procedure
- written in a fixed symbolic vocabulary
- governed by precise instructions
- moving in discrete steps, 1, 2, 3, ...
- whose execution requires no insight, cleverness, intuition, intelligence, or perspicuity
- and that sooner or later comes to an end

David Berlinski in The Advent of the Algorithm

Procedures and algorithms

There's a more specific, not commonly-used definition of an algorithm:

Practitioners (and many researchers) sometimes use the term more narrowly, to refer only to generative recursion. Or even more more narrowly than that, referring only to generative recursion that requires either a subtle correctness argument or a subtle termination argument.

That's because for most software systems, most of the recursion is structural. So for most software systems a large part of the focus is on data design - after that the basic recursive structure of the functions follows directly from the data.

In this narrow sense of the word algorithms are typically small but critical parts of real software systems that often require very specialized domain knowledge.

from CPSC 110 week 9 lecture notes

What's a data structure again?

- Data structure (street definition): how to organize your data to get the results you want, along with the supporting algorithms
- Any storage structure for data objects, along with the operations that are specific to the data structure
- There's a commonly used computing name for a combination of data structure and associated operations. You may have heard the term in CPSC 210 if not in CPSC 110...

Abstract data type

An abstract data type or ADT is a combination of

- a collection of data items
- a set of operations on those items

It's abstract because it lets us ignore the implementation details of how the data is organized or how the operations really work.

Abstraction

abstraction: treating something complex as if it were simpler, giving that simple thing a name, and throwing away (or postponing) the details.

"The most important concept in all of computer science is abstraction. Computer science deals with information and complexity. We make complexity manageable by judiciously reducing it when and where possible.

"I regret that I cannot recall who remarked that computation is the art of carefully throwing away information: given an overwhelming collection of data, you reduce it to a usable result by discarding most of its content."

Guy Steele

Abstract data type

An abstract data type or ADT is a combination of

- a collection of data items
- a set of operations on those items

Formally, an ADT is a mathematical description of an object and the set of operations on the object

In practice, an ADT is the interface of a data structure, without any information about the implementation

Abstract data type

You have lots of experience with ADTs. Every class definition with its associated method definitions that you wrote in CPSC 210 was an ADT.

Once you defined it, you or other programmers could work with the data contained in an object of that class using the object's methods. The interface between the programmer and the data/operations relieved the programmer of knowing anything about the inner details.

That's a really good thing. It makes programming easier. At least until your choice of ADT sucks all performance down a black hole. More about that after we look at some ADT examples.

Array and ArrayList

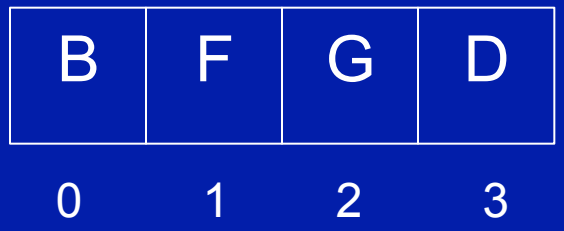
The array in Java is a very low-level ADT. Its logical model basically matches the underlying memory. And storing a value in an array requires that you know about array addressing. That's not much in the way of abstraction.

Arrays have fixed size. They can't grow or shrink.

You can't insert into or delete from the middle of an array.

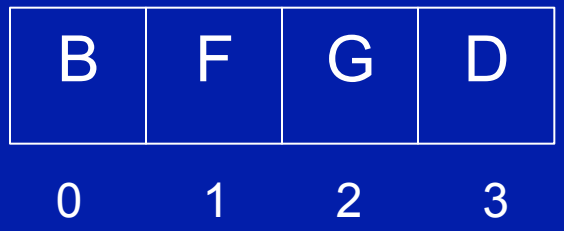
Within those limitations, they're very efficient.

Array and ArrayList



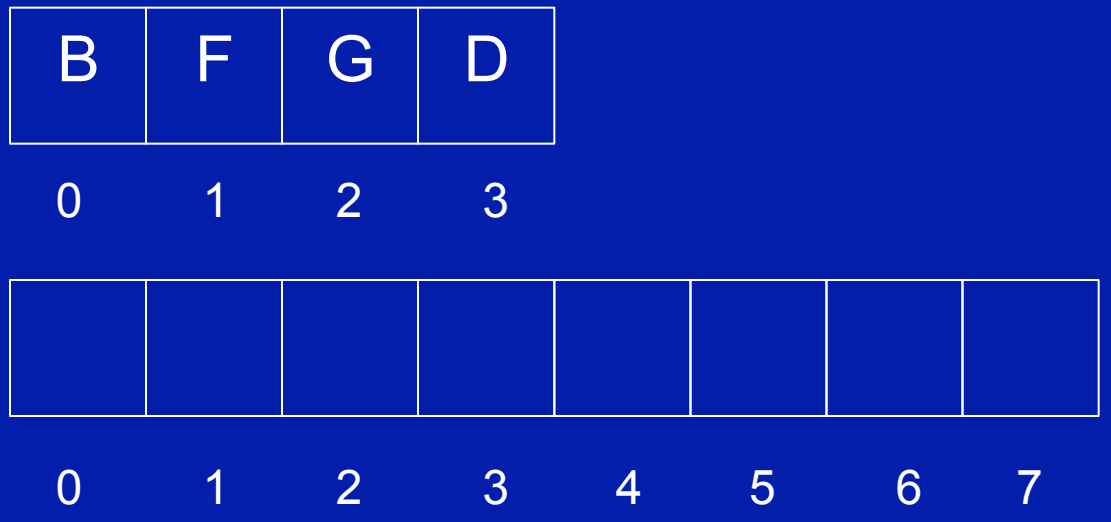
Java provides the ArrayList ADT for more flexibility and easier programming. You can add things and delete things anywhere. The ArrayList can grow in size. Great features for programmers, but there are hidden costs.

Array and ArrayList



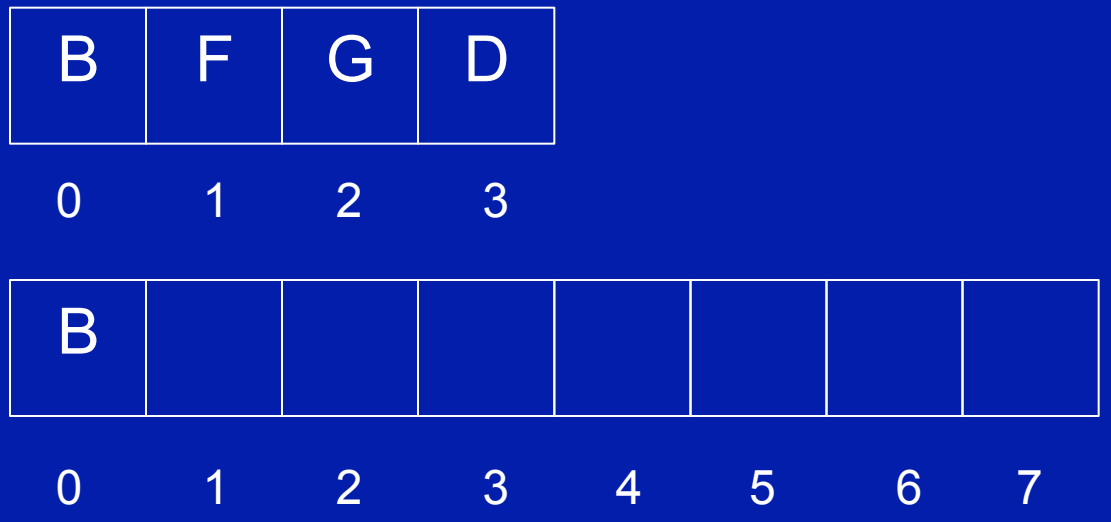
An ArrayList itself is implemented as a fixed-size array. What happens when you want to add to your ArrayList, but the fixed-size array is full and you want to add X?

Array and ArrayList



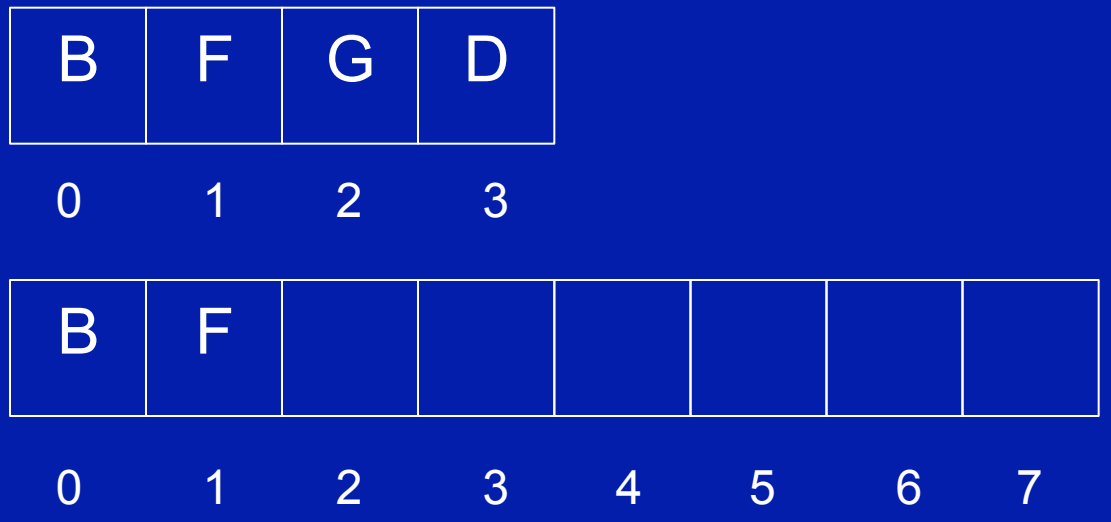
Make a new array that's larger than the original.

Array and ArrayList



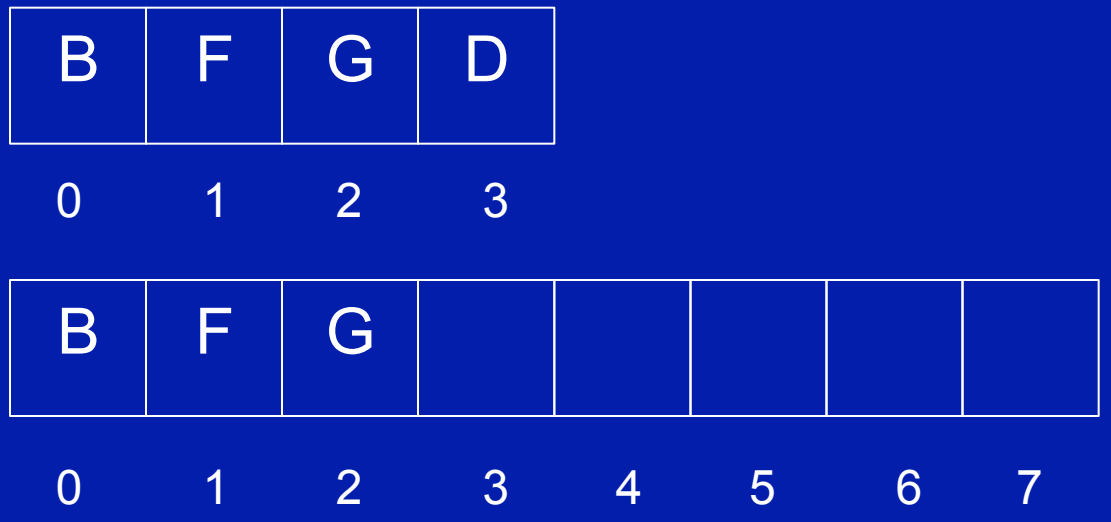
Make a new array that's larger than the original.
Now copy every element in the old array to the new array, one at a time.

Array and ArrayList



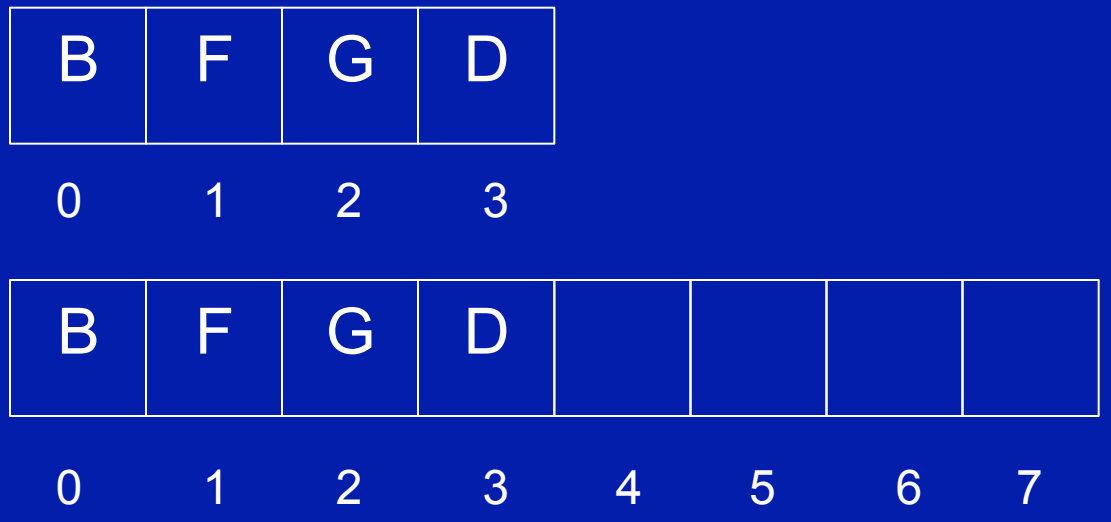
Make a new array that's larger than the original.
Now copy every element in the old array to the new array, one at a time.

Array and ArrayList



Make a new array that's larger than the original.
Now copy every element in the old array to the new array, one at a time.

Array and ArrayList



Make a new array that's larger than the original.
Now copy every element in the old array to the new array, one at a time.

Array and ArrayList

Make a new array that's larger than the original.
Now copy every element in the old array to the new array, one at a time.
Lose the old array so that it can be garbage collected.

Array and ArrayList

Make a new array that's larger than the original.
Now copy every element in the old array to the new array, one at a time.
Lose the old array so that it can be garbage collected.
Add the X to the new array.

Array and ArrayList

That's great flexibility for you as a programmer, but what's the cost here? No big deal if you're always working with small ArrayLists.

Array and ArrayList

But what if your ArrayList has 1,000,000 elements and it's full? Now you want to add just one more element. How much more memory is allocated? How much time does it cost?

Array and ArrayList

As an ArrayList gets really really big, adding just that one more element might result in an unexpected hiccup at run time: A sudden temporary performance loss at an inconvenient moment? A system crash when there's no more memory to be allocated?

Here are some definitely not real examples to illustrate the point (to the best of my knowledge, nothing here was actually written in Java)...

Array and ArrayList

Assume that Microsoft Word was written in Java using an ArrayList to store every character in that novel you've been writing for the past ten years.

Maybe every so often there's a slight slow down when the ArrayList is enlarged, but we don't really think much about word processor speed, so no harm done. But maybe once in a great while, when somebody is writing a really big document, Word just dies because there was no more memory to add to the ArrayList. (There goes your novel. You do backups frequently, right?)

But they're Microsoft programmers. They would have thought about how ArrayList really works ahead of time...

Array and ArrayList

You're playing World of Gears of Warcraft, and the folks at Amazingly Awesome Games built it using Java and ArrayLists(!?). As the game progresses, it hiccups once in awhile. That's annoying, and you make a mental note to complain about it online as soon as you're done with the game. But after several hours of continuous play (don't you ever read those health warnings?) your XBOX crashes and you lose your session.

You never buy an Amazingly Awesome Games title again. You're out a few dollars. But no serious damage to anyone or anything.

Array and ArrayList

But let's just say you're in charge of a major space science organization that's putting a science package on Mars. Your software developers have written the programs that control the descent to the surface of the Red Planet. They've chosen Java and ArrayLists, but they didn't really look into the performance issues that might be inherent in the ArrayList ADT.

Can you think of a time where a little performance glitch could be a serious problem?

Array and ArrayList

But let's just say you're in charge of a major space science organization that's putting a science package on Mars. Your software developers have written the programs that control the descent to the surface of the Red Planet. They've chosen Java and ArrayLists, but they didn't really look into the performance issues that might be inherent in the ArrayList ADT.

Can you think of a time where a little performance glitch could be a serious problem?

Well, a hiccup any time during that 7 minutes turns that Mars Lander into a $2,500,000,000 crater.

Array and ArrayList

Like I said, those are unrealistic exaggerated extremes to make a point.

The choices you make with respect to data structures and algorithms have to be informed choices. There are situations where the performance differences in the choices you make will have negligible impact. But there are going to be situations where your choices will have huge impact.

When your programming language provides you with conveniently packaged ADTs, take the time to understand how they do what they do, even though the point of the ADT is so that you don't have to know how it works.

Array and ArrayList

Arrays are our friends, but they have limitations.

They are of fixed size. “Growing” an array has costs.

On the other hand, a big array with few elements wastes space.

And here’s another limitation: homogeneity. What does that mean? All the data in an array has the same type. Why? So that the arithmetic with array indices is consistent.

Then how does an ArrayList in Java accommodate different data types?

Linked Lists

Linked lists are another way to represent a sequential collection of data with more flexibility than arrays.

That flexibility carries some cost in terms of memory used. For each data element you might store in an array, a list would need to use at least that same amount of memory, plus more to store the link to the next list element.

Linked Lists

A linked list is an abstract data type for representing data as a collection of linked nodes. Each node contains some information and at least one pointer to in an adjacent node.

Instead of having a single monolithic structure like an array in which adjacent items are physically next to each other, the ordering in a list is represented locally: any node knows only about the next node (via the pointer).

List elements may not be represented contiguously in memory, and they're likely not represented in the order that the elements occur in the list.

Linked Lists

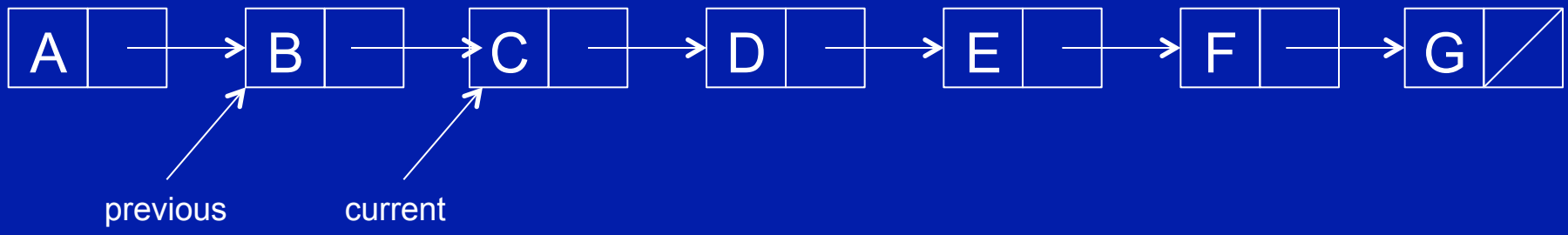
With a singly linked list, addition to the front of the list (i.e., the cons) is simple and efficient. It takes the same amount of time every time, no matter how big the list is. (That's called constant time.)

We showed you on Monday how to insert something into the middle of the list. What was our rough estimate of the time required to insert something in the middle of a list? The time to insert something in the middle of the list is, on average, going to be proportional to the size of the list, no?

Linked Lists

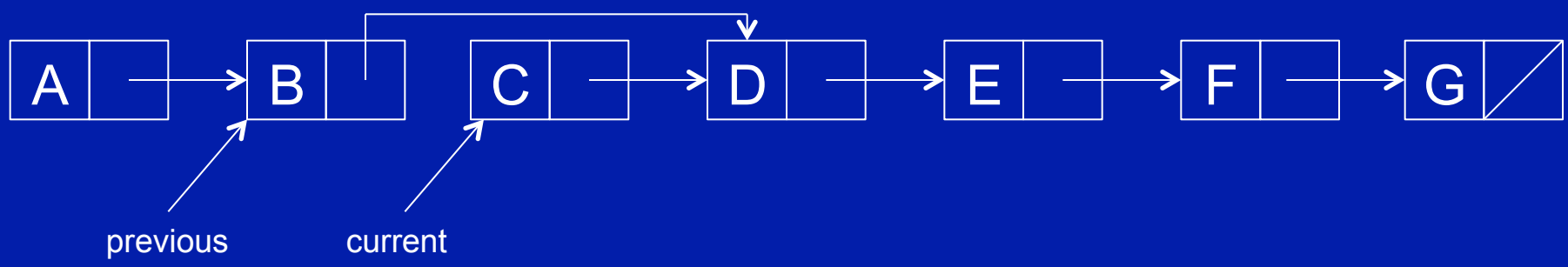
Deletion from a singly linked list is similarly straightforward.

Linked Lists



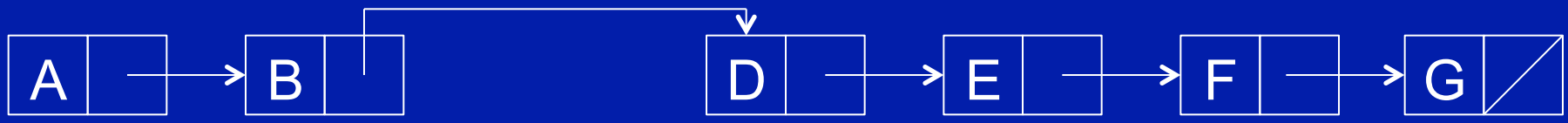
Deletion from a singly linked list is similarly straightforward. Assuming you know what the information is you want to delete (C, in this case), but don't know where it is, traverse list until you find the info to be deleted, while retaining a pointer to the previous node along the way,

Linked Lists



Deletion from a singly linked list is similarly straightforward. Assuming you know what the information is you want to delete (C, in this case), but don't know where it is, traverse list until you find the info to be deleted, while retaining a pointer to the previous node along the way, and copy the link leading away from the current node (C) into the link leading away from the previous node (B).

Linked Lists



Deletion from a singly linked list is similarly straightforward. Assuming you know what the information is you want to delete (C, in this case), but don't know where it is, traverse list until you find the info to be deleted, while retaining a pointer to the previous node along the way, and copy the link leading away from the current node (C) into the link leading away from the previous node (B). When the link node containing C is returned to free memory, the list looks like this.

Linked Lists

Your textbook says that singly linked lists have limitations:

- you can insert a node only after a node to which you already have a pointer
- you can remove a node only if you have a pointer to its predecessor node
- you can traverse only in one direction

The doubly linked list solves those problems by adding an additional link to each node, such that each node has a link to its predecessor and its successor in the sequence.

Stack

The stack is another container for a sequential collection of data.

The stack ADT permits data to be added or deleted only at one end of the sequence. It's easily implemented, and is great for storing "postponed obligations" or "postponed computations".

Real world analogies:

Spring-loaded dish dispenser in a cafeteria

Pez candy dispenser

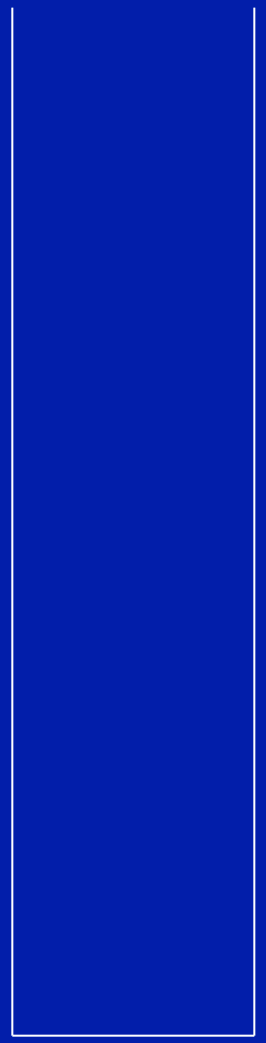
Stack

Only the “top” of a stack is accessible, so the stack ADT requires very few operations:

push(item)	-	add to top of stack
pop()	-	remove from top of stack
top()	-	return item at top without removing it
empty()	-	return true if stack empty else false
size()	-	return number of items on stack

Stack

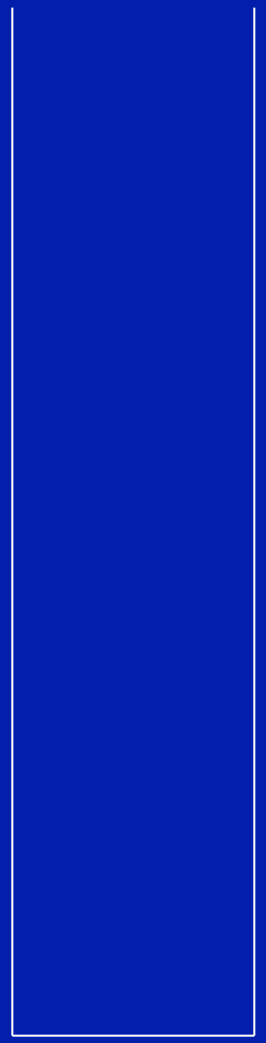
Here's a simple meaningless example of stack behaviour



Stack

Here's a simple meaningless example of stack behaviour

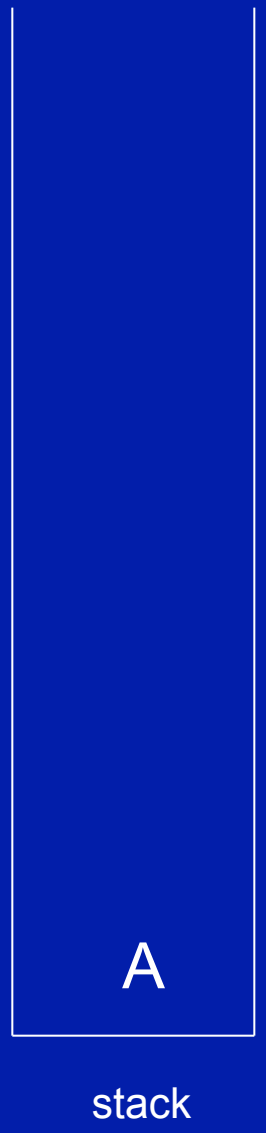
empty()
  true



Stack

Here's a simple meaningless example of stack behaviour

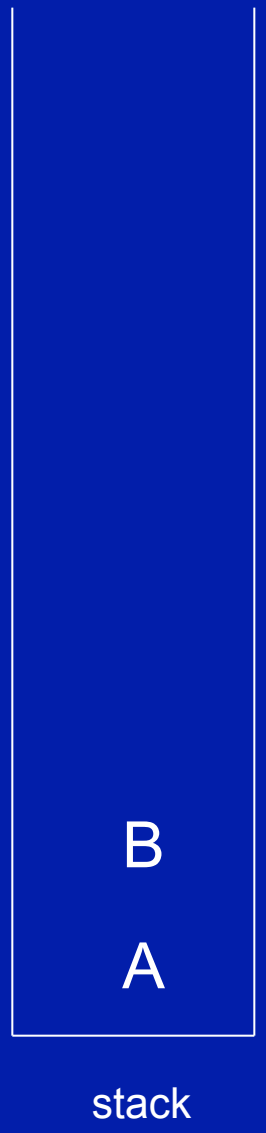
empty()
  true
push(A)



Stack

Here's a simple meaningless example of stack behaviour

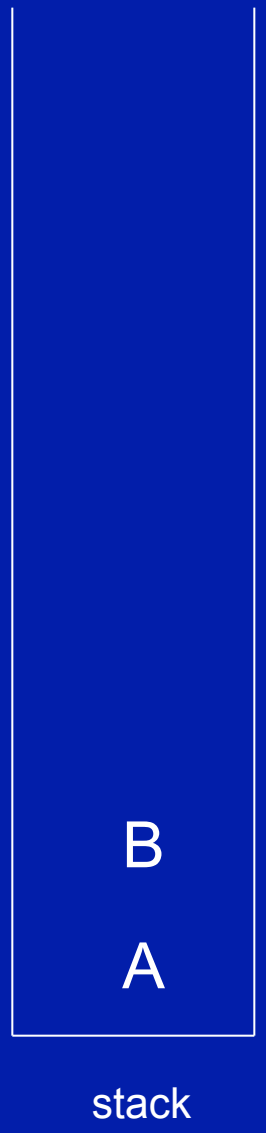
empty()
  true
push(A)
push(B)



Stack

Here's a simple meaningless example of stack behaviour

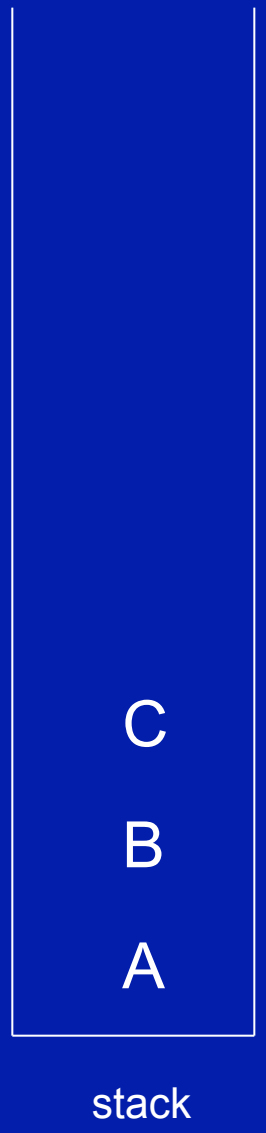
```
empty()
  true
push(A)
push(B)
top()
  B
```



Stack

Here's a simple meaningless example of stack behaviour

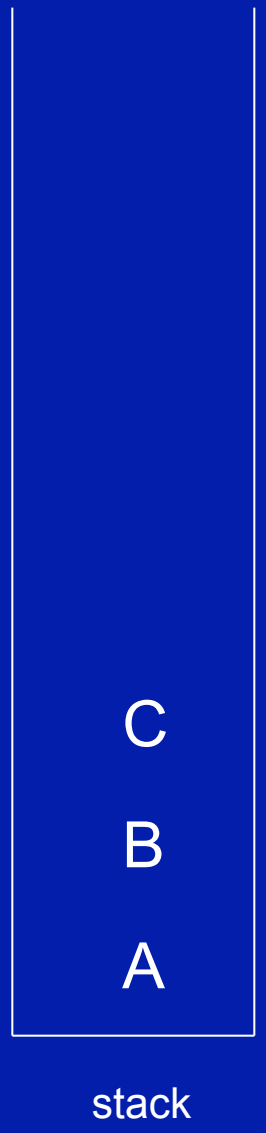
```
empty()
  true
push(A)
push(B)
top()
  B
push(C)
```



Stack

Here's a simple meaningless example of stack behaviour

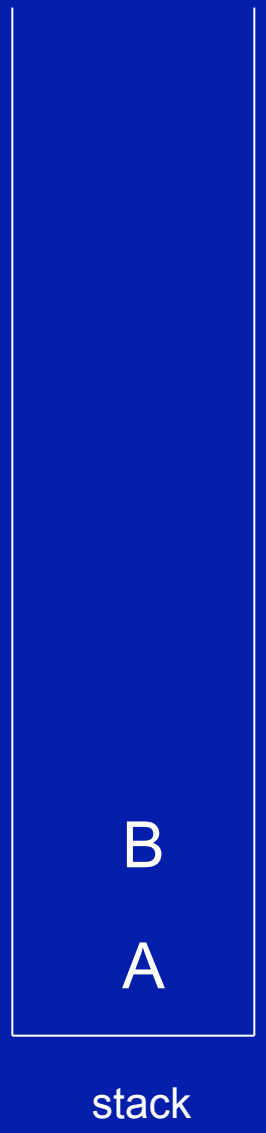
```
empty()
  true
push(A)
push(B)
top()
  B
push(C)
top()
  C
```



Stack

Here's a simple meaningless example of stack behaviour

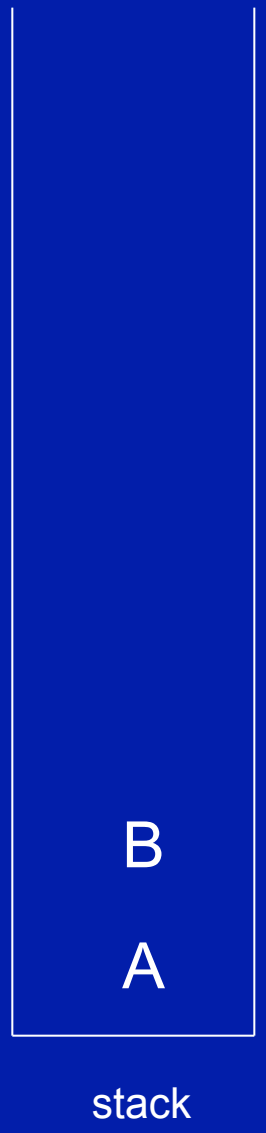
```
empty()
  true
push(A)
push(B)
top()
  B
push(C)
top()
  C
pop()
```



Stack

Here's a simple meaningless example of stack behaviour

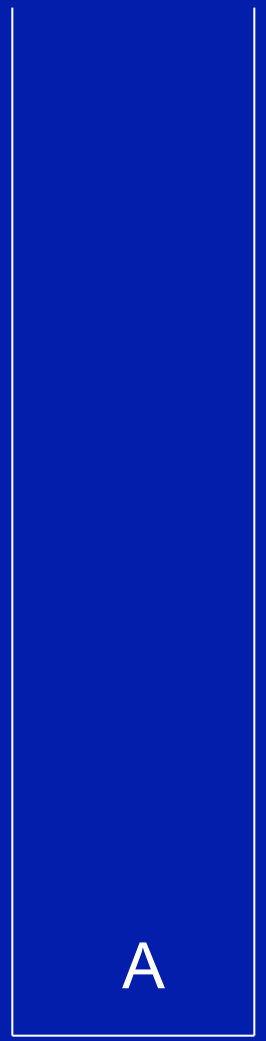
```
empty()
  true
push(A)
push(B)
top()
  B
push(C)
top()
  C
pop()
top()
  B
```



Stack

Here's a simple meaningless example of stack behaviour

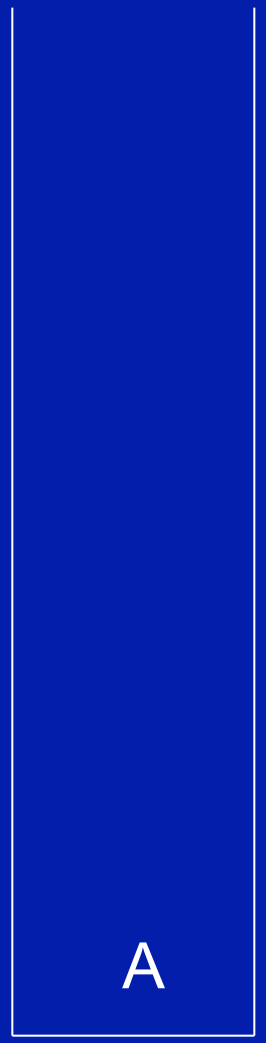
```
empty()
  true
push(A)
push(B)
top()
  B
push(C)
top()
  C
pop()
top()
  B
pop()
```



Stack

Here's a simple meaningless example of stack behaviour

```
empty()
  true
push(A)
push(B)
top()
  B
push(C)
top()
  C
pop()
top()
  B
pop()
size()
  1
```



Stack

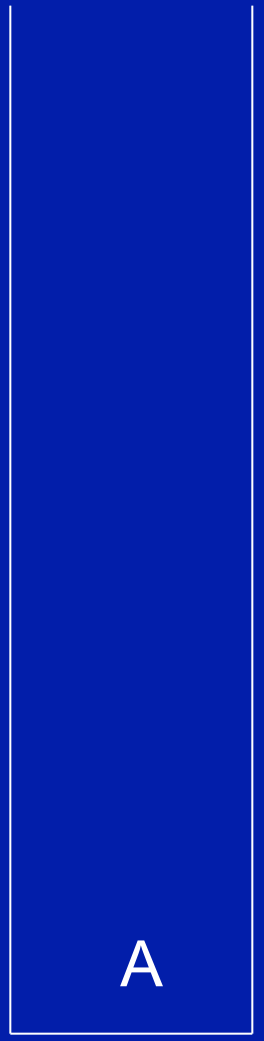
A stack reverses the order of arrival. This is called last-in-first-out or LIFO behaviour.

Stacks have been widely used in computing.

A compiler or interpreter may use a stack to parse expressions.

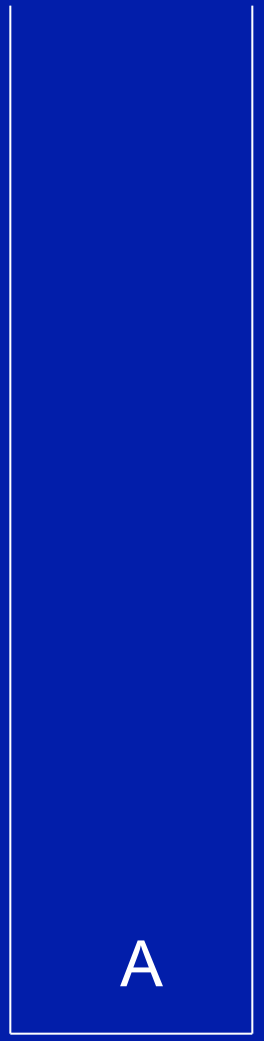
During program execution, a stack is used to keep track of procedure calls and parameters. It's how recursion is handled.

A web browser's 'back' button. Undo...



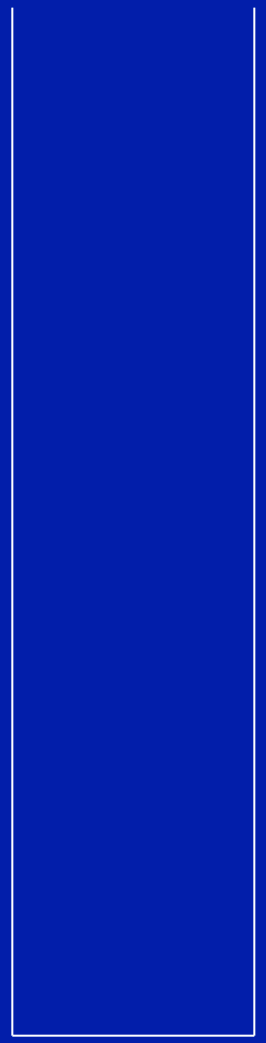
Stack

Around 1972, Hewlett Packard introduced the HP-35 scientific calculator ("the electronic slide rule"). It used a stack to evaluate arithmetic expressions, which in turn relied on the user to enter the expression in Reverse Polish Notation (RPN). RPN puts the operator after the operands, and through the correct ordering of operators the order of operands is preserved while the need for parentheses is eliminated.



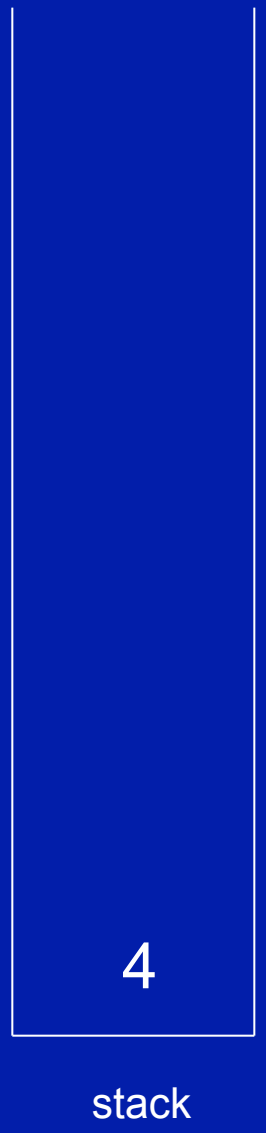
Stack

4 * (7 + 2) becomes
4 7 2 + *



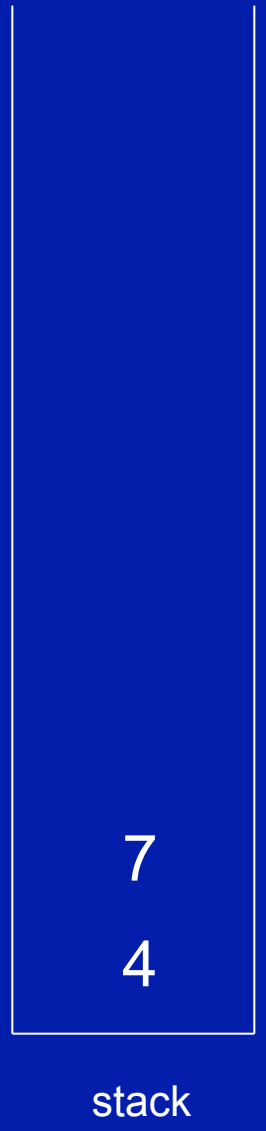
Stack

4 * (7 + 2) becomes
4 7 2 + *



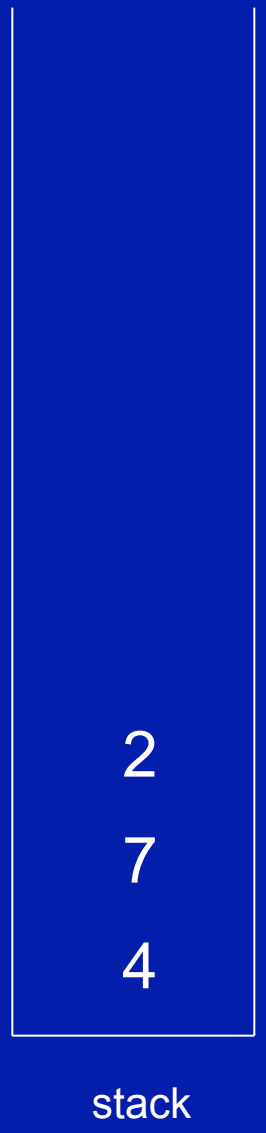
Stack

4 * (7 + 2) becomes
4 7 2 + *



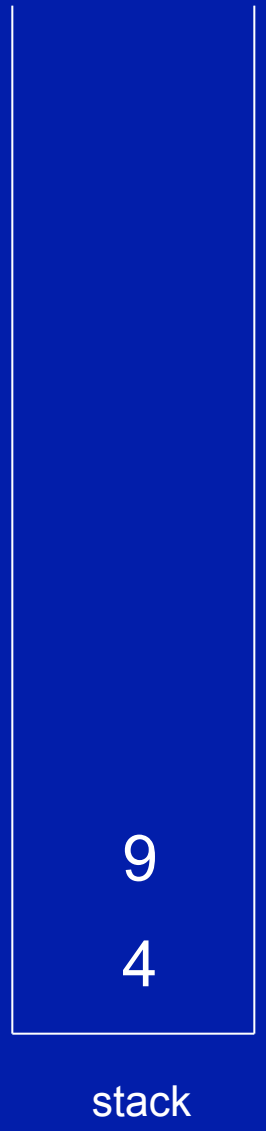
Stack

4 * (7 + 2) becomes
4 7 2 + *



Stack

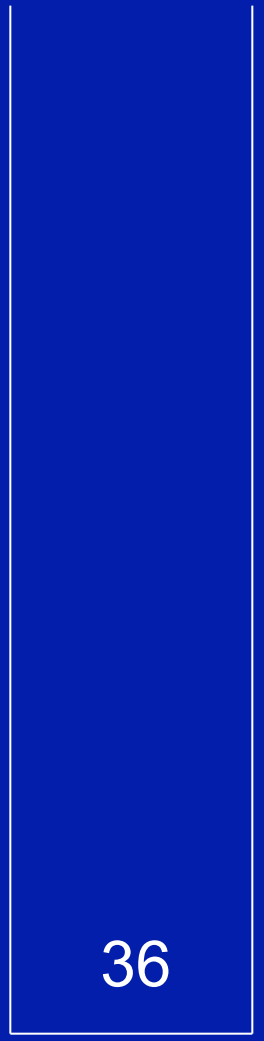
4 * (7 + 2) becomes
4 7 2 + *



Stack

4 * (7 + 2) becomes
4 7 2 + *

What is ((7 * 4) + 2) / (4 + 1) in RPN?

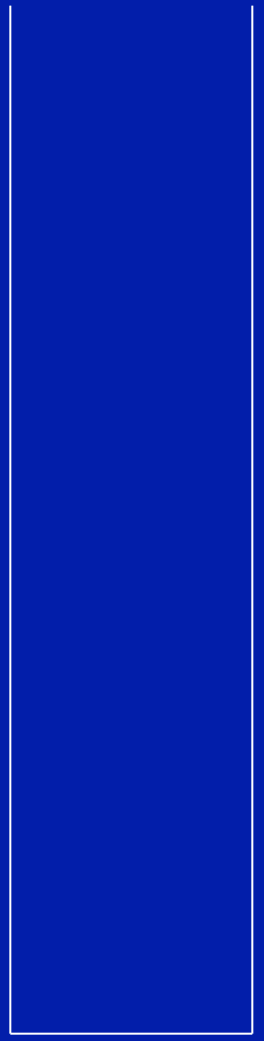


Stack

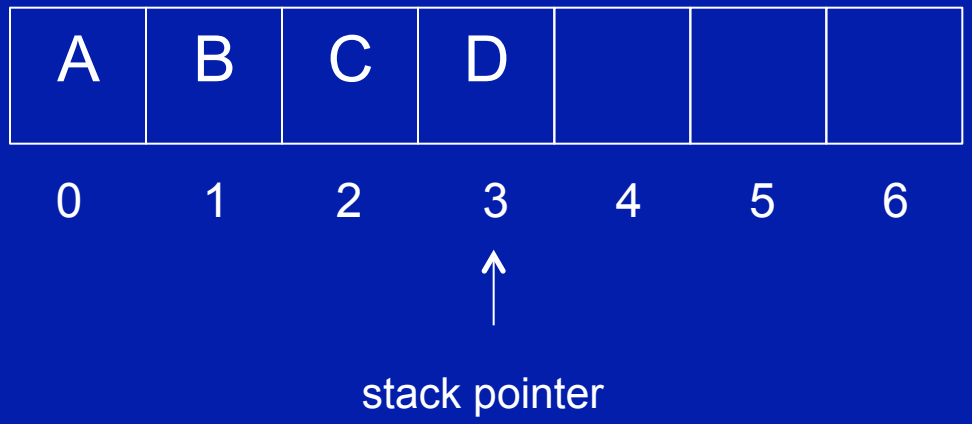
4 * (7 + 2) becomes
4 7 2 + *

What is ((7 * 4) + 2) / (4 + 1) in RPN?

There are still RPN calculators and people who use them.



Stack implementation



A stack can be implemented as a fixed array in memory.

What was the first thing pushed on the stack? What was the last thing? What will be the first thing popped off the stack?

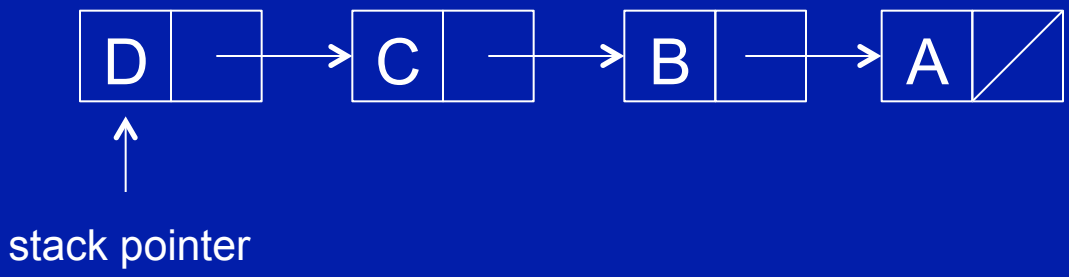
What are the details involved in pushing something on the stack? In popping something off the stack?

Stack operators revisited

Only the “top” of a stack is accessible, so the stack ADT requires very few operations:

push(item)	-	add to top of stack
pop()	-	remove from top of stack
top()	-	return item at top without removing it
empty()	-	return true if stack empty else false
size()	-	return number of items on stack

Linked Lists



A stack can also be implemented as a linked list.

Now what are the details involved in pushing something on the stack? In popping something off the stack?

Are there tradeoffs in choosing between an array implementation and a linked list implementation?