

Programming Abstractions

CS106B

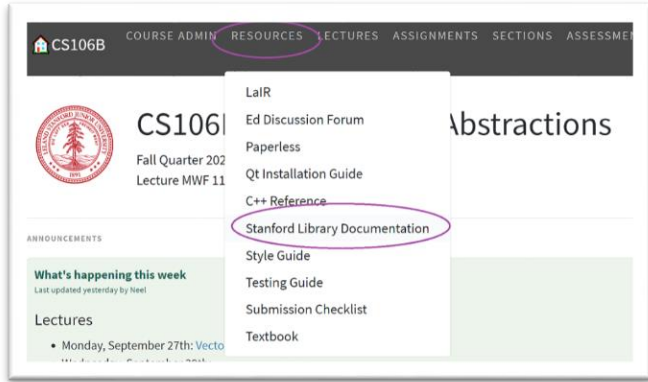
Cynthia Bailey Lee

Julie Zelenski

Today's Topics

Abstract Data Types

- Last time: What is an ADT? And two ADTs: Vector, Grid
- This time: More ADTs!
 - › Stack
 - › Queue
 - › Application of Stack
- Announcements:
 - › Assignment 1 due Friday
 - › *Do not use STL classes like vector, map, etc. for assignment 1 (or ever in this class). We also strongly recommend against using any Stanford library ADTs (except Vector, as directed). We carefully design these assignments to exercise certain skills learned in the class up to the date the assignment is released, so trust us, (a) you don't need them, (b) if you think you need them then you are missing a nice clean solution that takes a different approach (think topics we did cover in week 1, such as strings).*



Stacks

PRO TIP: TO VIEW DETAILS OF ANY OF OUR STANFORD LIBRARY IMPLEMENTATIONS OF ADTS, GO TO THE COURSE WEBSITE, RESOURCES TAB, STANFORD LIBRARY REFERENCE



New ADT: Stack

```
#include "stack.h"

Stack<string> recentCalls;
recentCalls.push("Neel");

recentCalls.push("Julie");

recentCalls.push("Esteban");

recentCalls.push("Minh");

while (!recentCalls.isEmpty()) {
    cout << recentCalls.pop() << " ";
}
```



New ADT: Stack

```
#include "stack.h"
```

```
Stack<string> recentCalls;  
recentCalls.push("Neel");
```

Neel

Julie

```
recentCalls
```

```
recentCalls
```

```
recentCalls
```

**“Why do I need Stack??
I could have done that with a Vector!”
—ADT skeptic**

```
while (!recentCalls.isEmpty()) {  
    cout << recentCalls.pop() << " ";  
}
```



Stack and Vector, side-by-side



```
Stack<string> recentCalls;
recentCalls.push("Neel");
recentCalls.push("Julie");
recentCalls.push("Esteban");
recentCalls.push("Minh");
```

```
while (!recentCalls.isEmpty()) {
    cout << recentCalls.pop() << " ";
}
```

0	1	2	3
Neel	Julie	Esteban	Minh

```
Vector<string> recentCalls;
recentCalls.add("Neel");
recentCalls.add("Julie");
recentCalls.add("Esteban");
recentCalls.add("Minh");
```

```
while (!recentCalls.isEmpty()) {
    string last = recentCalls[recentCalls.size() - 1];
    cout << last << " ";
    recentCalls.remove(recentCalls.size() - 1);
}
```

Stack and Vector, side-by-side



0	1	2	3
Neel	Julie	Esteban	Minh

```
Stack<string> recentCalls;
recentCalls.push("Neel");
recentCalls.push("Julie");
recentCalls.push("Esteban");
recentCalls.push("Minh");
```

```
while (!recentCalls.isEmpty()) {
    cout << recentCalls.pop() << " ";
}
```

```
Vector<string> recentCalls;
recentCalls.add("Neel");
recentCalls.add("Julie");
recentCalls.add("Esteban");
recentCalls.add("Minh");
```

```
while (!recentCalls.isEmpty()) {
    string last = recentCalls[recentCalls.size() - 1];
    cout << last << " ";
    recentCalls.remove(recentCalls.size() - 1);
}
```

This Vector code isn't terrible, but it is harder to read quickly, and is probably more error prone.

- You need to think carefully about which end of the Vector to use as the top of the stack (0^{th} or $\text{size}() - 1^{\text{th}}$), and performance impacts
- It would be easy to forget the "-1" when you print/remove $\text{size}() - 1^{\text{th}}$

Queues

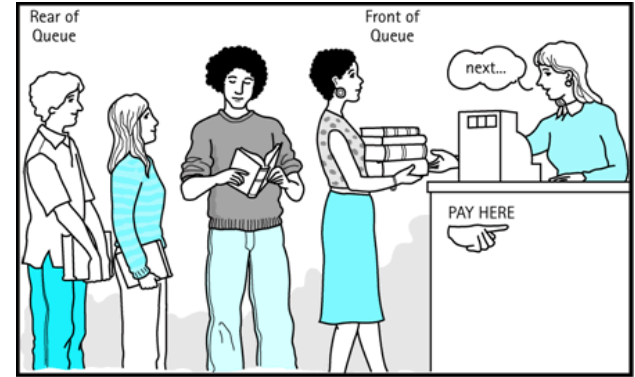
FIFO – FIRST IN, FIRST OUT
(OR “FIRST COME, FIRST
SERVE”)



Queues

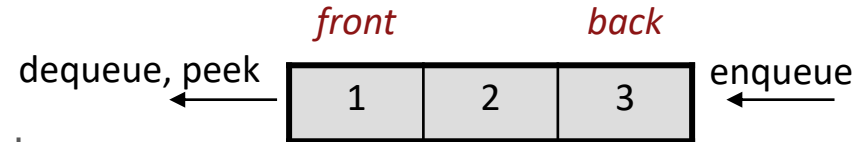
queue: First-In, First-Out ("FIFO")

- Elements stored in order they were added
- Can add only to the back, can only examine/remove frontmost element



queue operations

- enqueue: Add an element to the back
- dequeue: Remove the front element
- peek: Examine the front element



The Queue class

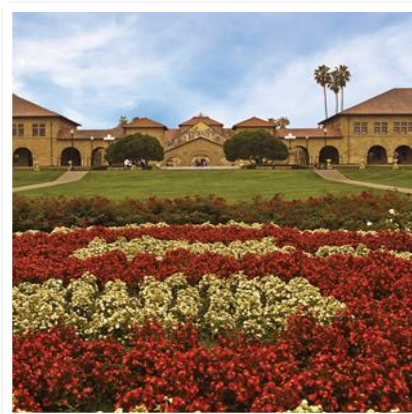
```
#include "queue.h"
```

<code>q.dequeue()</code>	removes front value and returns it; throws an error if queue is empty
<code>q.enqueue(value)</code>	places given value at back of queue
<code>q.isEmpty()</code>	returns true if queue has no elements
<code>q.peek()</code>	returns front value without removing; throws an error if queue is empty
<code>q.size()</code>	returns number of elements in queue

- `Queue<int> q; // {} front -> back`
- `q.enqueue(42); // {42}`
- `q.enqueue(-3); // {42, -3}`
- `q.enqueue(17); // {42, -3, 17}`
- `cout << q.dequeue() << endl; // 42 (q is {-3, 17})`
- `cout << q.peek() << endl; // -3 (q is {-3, 17})`
- `cout << q.dequeue() << endl; // -3 (q is {17})`

Application of Stacks

WE'VE SEEN ONE (BUFFERING
INPUT AND GIVING IT BACK IN
REVERSE—LIFO—ORDER).
WHAT ELSE ARE STACKS GOOD
FOR?



Operator Precedence and Syntax Trees

Ignoring operator precedence rules, how many distinct results are there to the following arithmetic expression?

- $3 * 3 + 3 * 3$

Reverse Polish Notation

Ambiguities don't exist in RPN

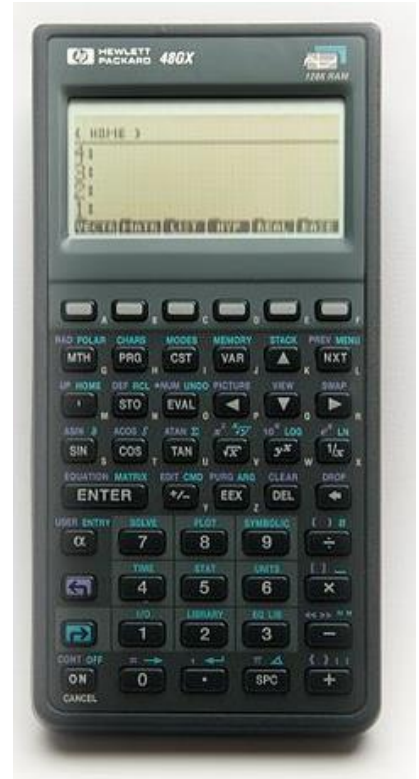
Also called “postfix” because the operator goes after the operands

Postfix (RPN):

- $43 * 43 * +$

Equivalent Infix:

- $(4*3) + (4*3)$

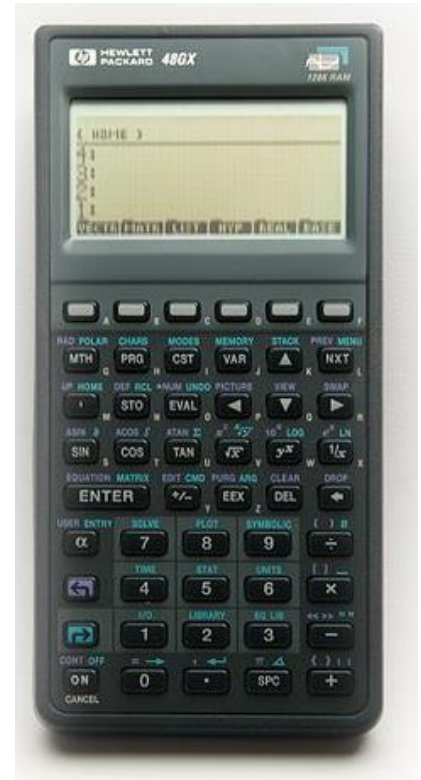


http://commons.wikimedia.org/wiki/File:Hewlett-Packard_48GX_Scientific_Graphing_Calculator.jpg

Reverse Polish Notation



#TBT: Me in 1991, I was 12 years old



ia.org/wiki/File:Hewlett-Packard_48GX_Scientific_Graphing_Calculator.jpg

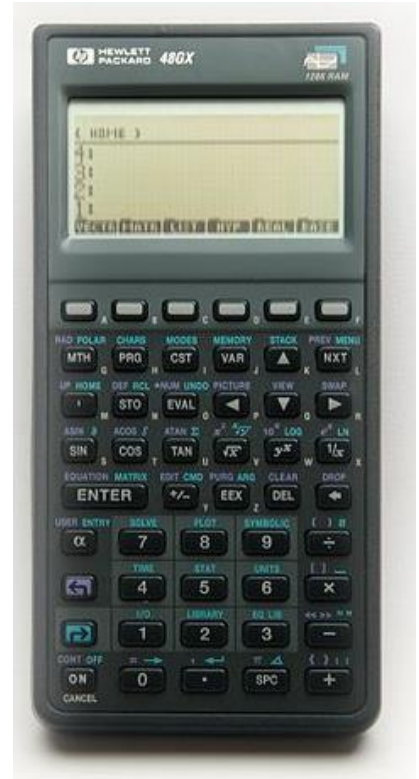
Reverse Polish Notation

This postfix expression:

- $43 * 725 * ++$

Is equivalent to this infix expression:

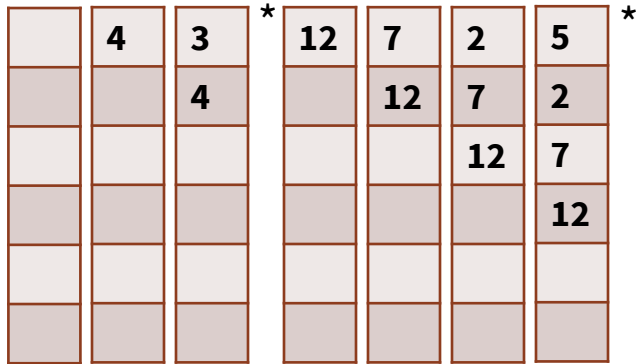
- A. $((4*3) + (7*2)) + 5$
- B. $(4*3) + ((7+2) + 5)$
- C. $(4*3) + (7 + (2*5))$
- D. Other/none/more than one



http://commons.wikimedia.org/wiki/File:Hewlett-Packard_48GX_Scientific_Graphing_Calculator.jpg

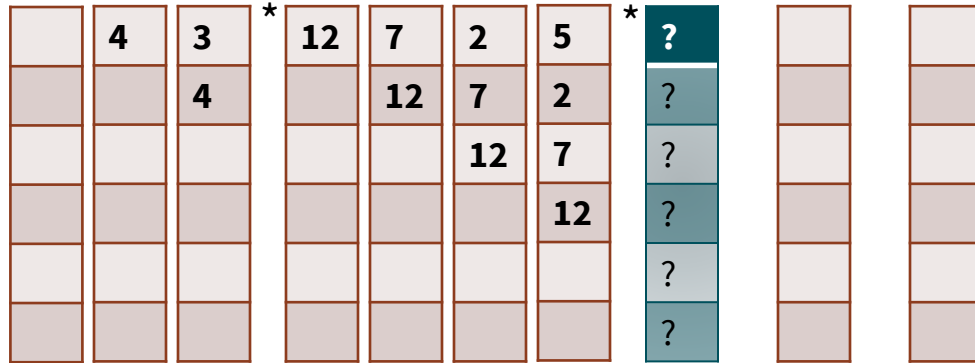
Stacks and RPN

- Evaluate this expression with the help of a stack
 - › Encounter a **number**? **PUSH** it
 - › Encounter an **operator**? **POP** two numbers and **PUSH** result
- $4\ 3\ *\ 7\ 2\ 5\ *\ +\ +$



Stacks and RPN

- Evaluate this expression with the help of a stack
 - Encounter a **number**? **PUSH** it
 - Encounter an **operator**? **POP** two numbers and **PUSH** result
- $4\ 3\ *\ 7\ 2\ 5\ * + +$



Contents of the stack, reading from top down:

(A) 7, 12

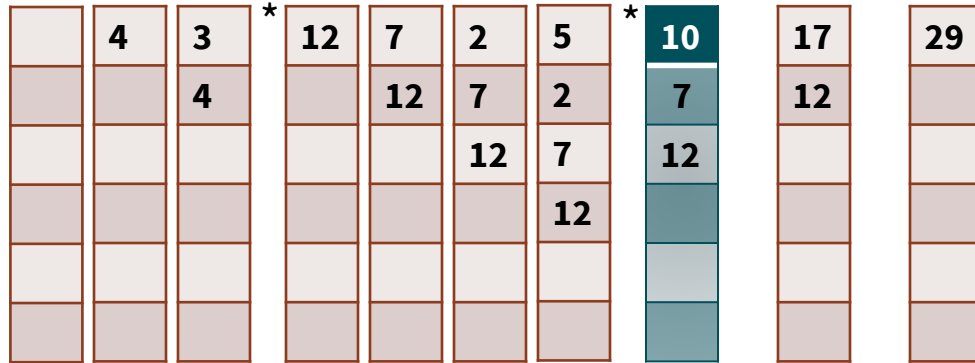
(B) 10, 7, 12

(C) 10, 5, 2, 7, 12

(D) Other

Stacks and RPN

- Evaluate this expression with the help of a stack
 - › Encounter a **number**? **PUSH** it
 - › Encounter an **operator**? **POP** two numbers and **PUSH** result
- $4\ 3\ *\ 7\ 2\ 5\ *\ +\ +$



- **Question:** what are some signs that an expression is badly formatted?

Final code of parser

```
bool calculate(const string& expression, int& result)
{
    Stack<int> memory;
    for (size_t i = 0; i < expression.length(); i++) {
        if (isdigit(expression[i])) {
            int value = charToInteger(expression[i]);
            memory.push(value);
        } else if (isSupportedOperator(expression[i]) && memory.size() >= 2) {
            int rhs = memory.pop(); // right-hand-side operand is 1st to pop
            int lhs = memory.pop(); // left-hand-side operand is 2nd to pop
            memory.push(applyOperator(lhs, expression[i], rhs));
        } else {
            // parse error on anything other than digit or operator
            return false;
        }
    }
    // final parse validity check
    if (memory.size() != 1) {
        return false;
    }
    result = memory.pop();
    return true;
}
```