

Programming Abstractions

CS106B

Instructor: Cynthia Bailey

Head TA: Yasmine Alonso

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:
 - › Sections are starting this afternoon!
 - Check your assigned section or do late registration at cs198.stanford.edu
 - › Assignment 1 due Friday

pollev.com/cs106b



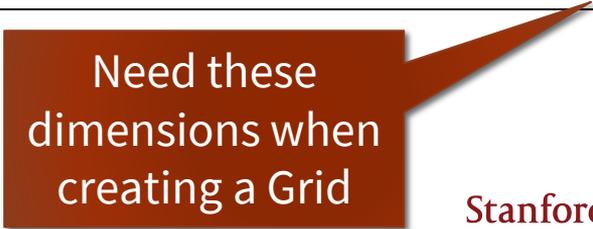
Grid

- ADT abstraction similar to an array of arrays (matrix)
- Many languages have a version of this
 - › (remember, ADTs are conceptual abstractions that are language-independent)
- In C++ we declare one like this:



Typo correction
from last time!

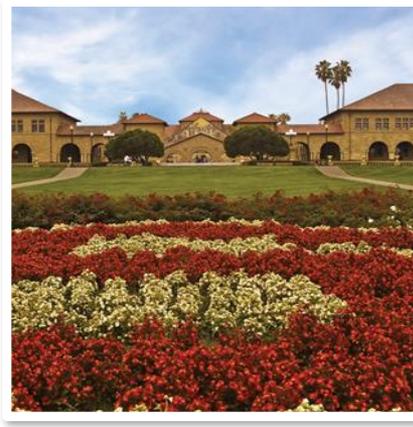
```
#include "grid.h"  
  
Grid<int> chessboard(8, 8);  
Grid<int> image(500, 1000);  
Grid<double> realMatrix(10, 10);
```



Need these
dimensions when
creating a Grid

Stacks

OUR NEXT ORDERED ADT
IT USES “LIFO” ORDER
(LAST-IN, FIRST-OUT)



New ADT: Stack

```
#include "stack.h"

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

recentCalls.push("Julie");

recentCalls.push("Esteban");

recentCalls.push("Minh");

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

Bottom of stack
(first to be added,
last to come out)



Top of stack (last to
be added, first to
come out)



New ADT: Stack

```
#include "stack.h"

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

recentCalls.push("Julie");

recentCalls.push("Esteban");

recentCalls.push("Minh");

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



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

Stack and Vector, side-by-side



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

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

0	1	2	3
Yasmine	Julie	Esteban	Minh

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

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

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

```
Vector<string> recentCalls;
recentCalls.add("Yasmine");
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”)



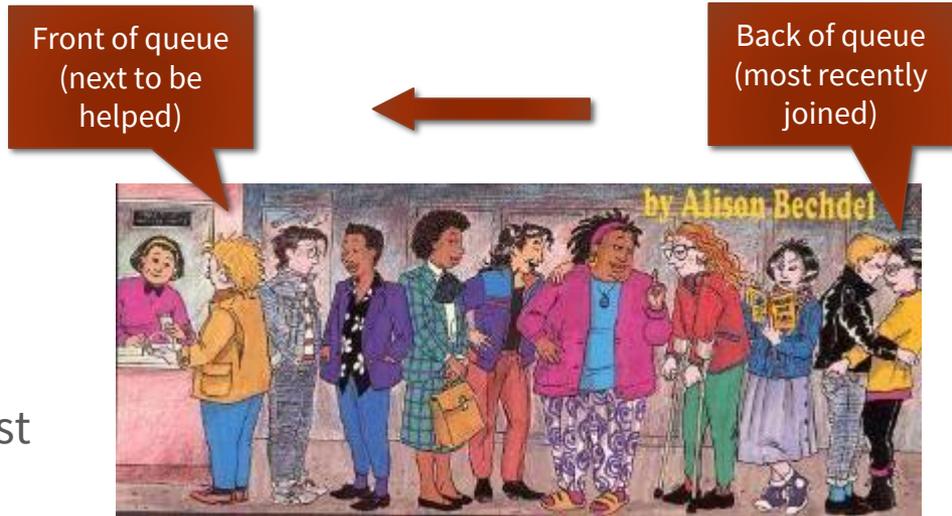
Queues

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"
```

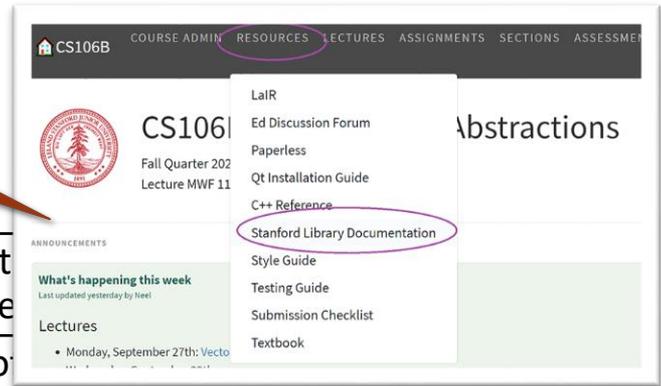
<code>q.dequeue()</code>	removes front value and returns it; throws an error if queue is empty
<code>q.enqueue(value)</code>	places given value at back of queue
<code>q.isEmpty()</code>	returns true if queue has no elements
<code>q.peek()</code>	returns front value without removing; throws an error if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

The Queue class

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

As usual, for more information check course website!

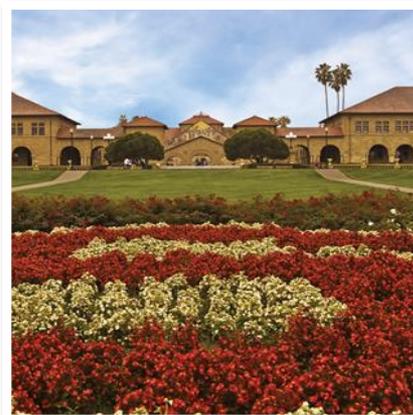


<code>q.dequeue()</code>	removes front value and returns it; throws an error if queue is empty
<code>q.enqueue(value)</code>	places given value at back of queue
<code>q.isEmpty()</code>	returns true if queue has no elements
<code>q.peek()</code>	returns front value without removing; throws an error if 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 NAMES AND GIVING
THEM BACK IN REVERSE).
WHAT ELSE ARE STACKS GOOD
FOR?



Operator Precedence and Syntax Trees

Ignoring operator precedence rules, what are all the distinct results for the following arithmetic expression?

- $3 * 3 + 3 * 3$

Go to
pollev.com/cs106b
to respond!



Reverse Polish Notation

Ambiguities don't exist in RPN

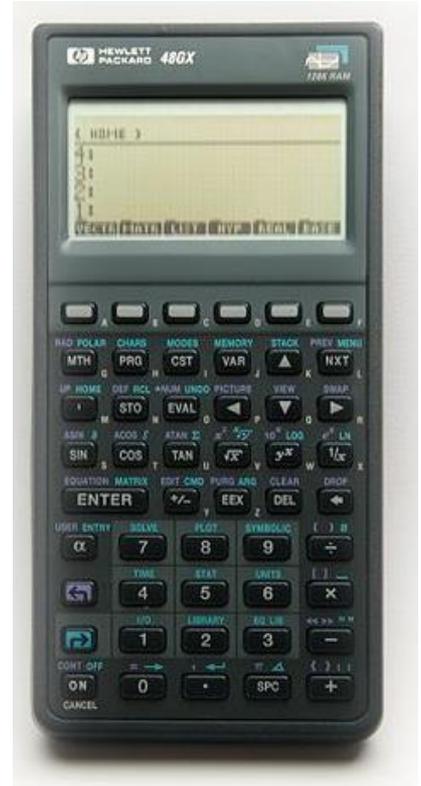
Also called “postfix” because the operator goes after the operands

Postfix (RPN):

- $43 * 43 * +$

Equivalent Infix:

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

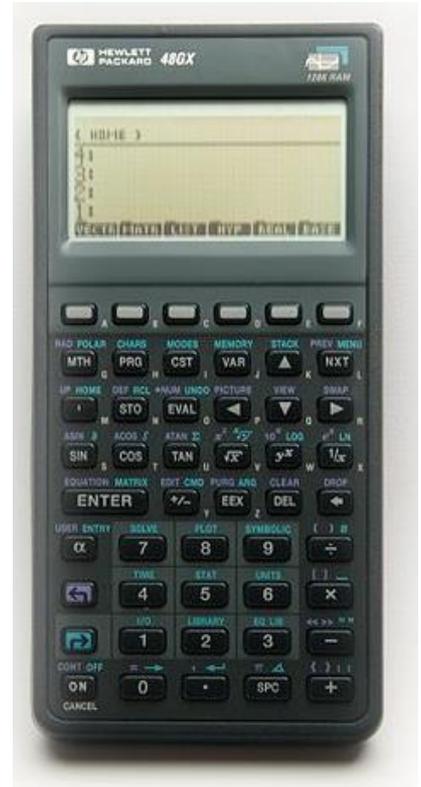


http://commons.wikimedia.org/wiki/File:Hewlett-Packard_48GX_Scientific_Graphing_Calculator.jpg

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

Go to
pollev.com/cs106b
to respond!

This postfix expression:

■ $4\ 3\ * \ 7\ 2\ 5\ * \ + \ +$

Is equivalent to this infix expression:

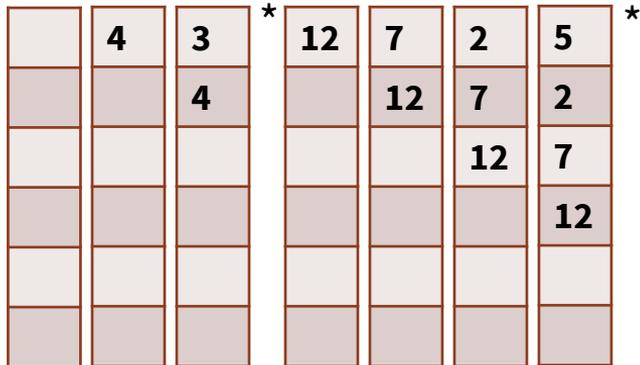
- A. $((4*3) + (7*2)) + 5$
- B. $(4*3) + ((7+2) + 5)$
- C. $(4*3) + (7 + (2*5))$
- D. Other/none/more than one



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
- $4\ 3\ *\ 7\ 2\ 5\ *\ +\ +$

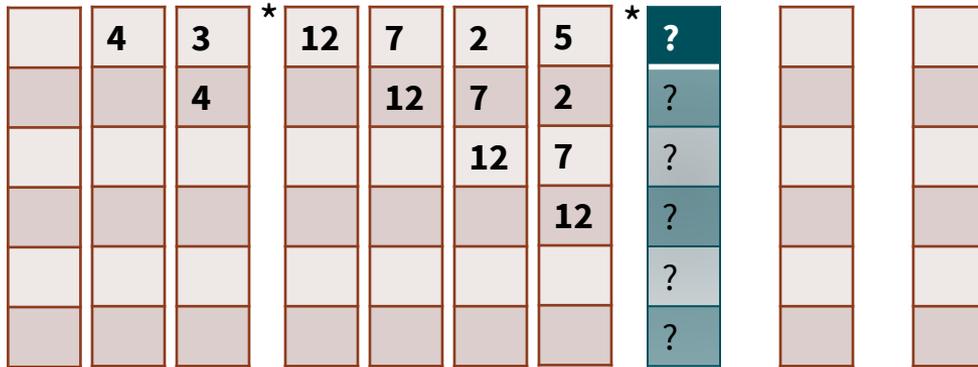


Stacks and RPN

Go to pollev.com/cs106b to respond!



- 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

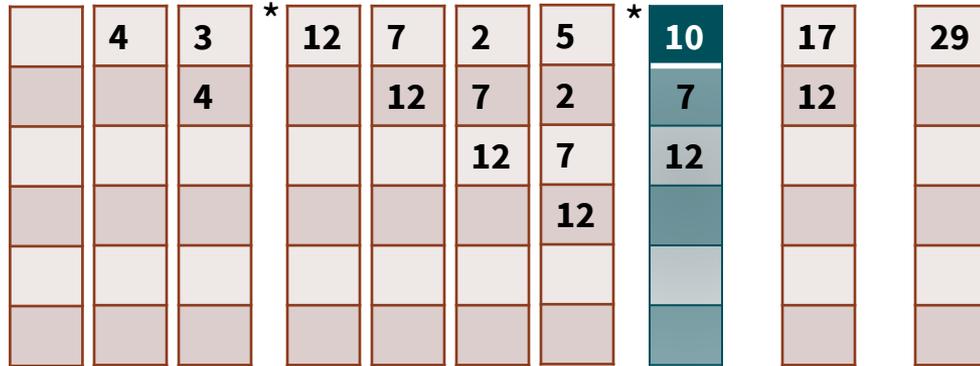
(B) 10, 7, 12

(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
- $4\ 3\ *\ 7\ 2\ 5\ *\ +\ +$



Final answer

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

Final code of parser

```
bool calculate(string expression, int& result)
{
    Stack<int> memory;
    // Examine each character of input, left to right
    for (char c : expression) {
        // if digit, store it
        if (isdigit(c)) {
            int value = charToInteger(c);
            memory.push(value);
        // if operator, perform operation
        } else if (isSupportedOperator(c) && memory.size() >= 2) {
            int rhs = memory.pop();
            int lhs = memory.pop();
            memory.push(applyOperator(lhs, c, rhs));
        // otherwise parse error
        } else {
            return false;
        }
    }
    // should be single number in memory, that's our answer
    if (memory.size() != 1) {
        return false;
    }
    result = memory.pop();

    return true;
}
```