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

Our next ordered ADT it uses “LIFO” order (last-in, first-out)
New ADT: Stack

```cpp
#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");
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\langle string\rangle recentCalls;
recentCalls.push("Neel");
recentCalls.push("Julie");
recentCalls.push("Esteban");
recentCalls.push("Minh");

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

Vector\langle string\rangle 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

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

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

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 (0th or size()-1th), and performance impacts
- It would be easy to forget the “-1” when you print/remove size()-1th
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

```c++
#include "queue.h"

Queue<int> q;                  // {}   front
q.enqueue(42);                 // {42}
q.enqueue(-3);                 // {42, -3}
q.enqueue(17);                 // {42, -3, 17}

```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>q.dequeue()</code></td>
<td>Removes front value and returns it; throws an error if queue is empty</td>
</tr>
<tr>
<td><code>q.enqueue(value)</code></td>
<td>Places given value at back of queue</td>
</tr>
<tr>
<td><code>q.isEmpty()</code></td>
<td>Returns true if queue has no elements</td>
</tr>
<tr>
<td><code>q.peek()</code></td>
<td>Returns front value without removing; throws an error if queue is empty</td>
</tr>
<tr>
<td><code>q.size()</code></td>
<td>Returns number of elements in queue</td>
</tr>
</tbody>
</table>

- `q.dequeue()` removes front value and returns it; throws an error if queue is empty.
- `q.enqueue(value)` places given value at back of queue.
- `q.isEmpty()` returns true if queue has no elements.
- `q.peek()` returns front value without removing; throws an error if queue is empty.
- `q.size()` returns number of elements in queue.
# The Queue class

```cpp
#include "queue.h"

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})
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>q.dequeue()</code></td>
<td>Removes front value and returns it; throws an error if queue is empty</td>
</tr>
<tr>
<td><code>q.enqueue(value)</code></td>
<td>Places given value at back of queue</td>
</tr>
<tr>
<td><code>q.isEmpty()</code></td>
<td>Returns true if queue has no elements</td>
</tr>
<tr>
<td><code>q.peek()</code></td>
<td>Returns front value without removing; throws an error if queue is empty</td>
</tr>
<tr>
<td><code>q.size()</code></td>
<td>Returns number of elements in queue</td>
</tr>
</tbody>
</table>

As usual, for more information check course website!
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?

- $3 \times 3 + 3 \times 3$

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):
- 4 3 * 4 3 * +

Equivalent Infix:
- (4*3) + (4*3)
Reverse Polish Notation

Ambiguities don’t exist in RPN. Also called “postfix” because the operator goes after the operands.

Postfix (RPN):
- \[ 4 \ 3 \ * \ 4 \ 3 \ * \ + \]

Equivalent Infix:
- \[(4*3) + (4*3)\]

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

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
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

- \(4 \times 3 \times 2 + 5++\)

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
- \( 4 \times 3 \times 2 + 5 + + \)

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