CS 106B
Lecture 5: Stacks and Queues

Wednesday, April 11, 2018

Programming Abstractions
Spring 2018
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Chapter 5.2-5.3
Today's Topics

• Logistics:
  • LIFE due on Thursday!
  • Python to C++ session: 1:30PM-2:20PM Friday in Skilling Auditorium (will be recorded)
• Pair Programming: Rules and strategies
• Do we have to implement a Vector with an array?
  • We can implement the Vector Abstract Data Type any way we want!
• Efficiency of element manipulation
• Stacks
• Queues
In CS 106B, you may pair program with a partner from your section (SCPD: you can pair program with another SCPD student, but you must follow the rules below, which would necessitate a screen-sharing method):

1. Students must work on 100% of the assignment at the exact same time, together, in front of a single computer.

2. Students may **not** split the assignment into parts where one partner does some parts and the other does different parts.

3. All code-related LaIR help must have both partners present.
Real Life Cellular Automata?

A Lizard With Scales That Behave Like a Computer Simulation
Vector ("dynamic array") Review

• Pop quiz: what gives better speed, inserting and removing at the beginning or at the end of a Vector, and what is the Big O?

  Answer: At the end (no moving of elements necessary) O(1)

• Pop quiz: If a Vector has \(n\) elements, and we are going to insert somewhere into the Vector, what is the maximum number of elements that must be moved? Answer: \(n\)

• Pop quiz: If a Vector has \(n\) elements, and we are going to insert somewhere into the Vector, what is the minimum number of elements that must be moved? Answer: 0
• In the last class, we talked about how a Vector is implemented under the hood. And that may be important to the user -- the user should know about the speed of the operations so that he or she can make good choices about using the Vector.

• But -- is it necessary to implement a Vector with an array? We are going to switch gears a bit and talk about **Abstract Data Types (ADT)**, and the Vector actually counts as an ADT.

An abstract data type is a model that describes how data is manipulated **from the point of view of the user**. In other words, the user should get a set of functions and behaviors that are identical **regardless of how the model is implemented**.
Vector: Does the user care how it is implemented?

Consider the following program that uses a Vector:

```cpp
int main() {
    Vector<string> states;
    // manually add in alphabetic order
    states.add("California");
    states.insert(0,"Alaska");
    states.add("Texas");
    states.insert(3,"Utah");
    states.insert(2,"Nebraska");

    cout << "Originals: "
        << states.toString() << endl; // for testing

    // revolution happens
    states.remove(3); // Texas

    cout << "After removing Texas: "
        << states.toString() << endl;

    return 0;
}
```

From the perspective of the user, the Vector class needs to have certain behaviors, but the user isn't really concerned with how those behaviors are implemented under the hood.

This is the idea behind an ADT -- the Vector needs to have `add()`, `insert()`, and `remove()` functions, but the fact that it is an array under the hood is not relevant if you just want certain behaviors.

So...
We could imagine implementing the Stanford Library Vector using Moon Monkeys®, who keep all of our data on the Moon, and simply pass back the results of our functions as we need them.

```cpp
Vector<string> states;
states.add("California");
```

For these statements, we call up the Moon Monkeys, and say "we want a Vector that holds strings," and then we radio up to them, 'Please add "California" to the Vector.'

Is this efficient? No. Does it meet the requirements of our Vector ADT? Yes. That is the principle idea behind ADTs: the functional behavior is what matters, not the implementation.
A “stack” is another example of an abstract data type. A stack has the following behaviors / functions:

push(value) (or add(value)) - place an entity onto the top of the stack
pop() (or remove()) - remove an entity from the top of the stack and return it
top() (or peek()) - look at the entity at the top of the stack, but don’t remove it
isEmpty() - a boolean value, true if the stack is empty, false if it has at least one element. (note: a runtime error occurs if a pop() or top() operation is attempted on an empty stack.)

Why do we call it a "stack?" Because we model it using a stack of things:
The push, pop, and top operations are the only operations allowed by the stack ADT, and as such, **only** the top element is accessible. Therefore, a stack is a “Last-In-First-Out” (LIFO) structure: the last item in is the first one out of a stack.
Despite the stack’s limitations (and indeed, because of them), the stack is a very frequently used ADT. In fact, most computer architectures implement a stack at the very core of their instruction sets — both push and pop are assembly code instructions.

Stack operations are so useful that there is a stack built in to every program running on your PC — the stack is a memory block that gets used to store the state of memory when a function is called, and to restore it when a function returns.

Why are stacks used to when functions are called?

Let’s say we had a program like this:

```
main() {
    function1();
    return;
}

function1() {
    function2();
    return;
}

function2() {
    function3();
    return;
}
```

What happens to the state of the system as this program runs?
main calls function1, which calls function2, which calls function3.

Then, function3 returns, then function2 returns, then function1 returns, then main returns.

This is a LIFO pattern!
Stacks

What are some downsides to using a stack?

- No random access. You get the top, or nothing.
- No walking through the stack at all — you can only reach an element by popping all the elements higher up off first
- No searching through a stack.

What are some benefits to using a stack?

- Useful for lots of problems -- many real-world problems can be solved with a Last-In-First-Out model (we'll see one in a minute)
- Very easy to build one from an array such that access is guaranteed to be fast.
  - Where would you have the top of the stack if you built one using a Vector? Why would that be fast?
Reversing the words in a sentence

Let's build a program from scratch that reverses the words in a sentence.

Goal: reverse the words in a sentence that has no punctuation other than letters and spaces.

How might we do this?

1. Use a stack
2. Read characters in a string and place them in a new word.
3. When we get to a space, push that word onto the stack, and reset the word to be empty.
4. Repeat until we have put all the words into the stack.
5. Pop the words from the stack one at a time and print them out.
The following is a simple example of a program that uses a Stack. It simply creates a stack, pushes words onto the stack, then pops off the words and prints them.

```
#include <iostream>
#include "console.h"
#include "stack.h"

using namespace std;
const char SPACE = ' ';

int main() {
    string sentence = "hope is what defines humanity";
    string word;
    Stack<string> wordStack;

    cout << "Original sentence: " << sentence << endl;
    for (char c : sentence) {
        if (c == SPACE) {
            wordStack.push(word);
            word = ""; // reset
        } else {
            word += c;
        }
    }
    if (word != "") { // reset
        wordStack.push(word);
    }
    cout << "    New sentence: ";
    while (!wordStack.isEmpty()) {
        word = wordStack.pop();
        cout << word << SPACE;
    }
    cout << endl;
    return 0;
}
```

Output:
Original sentence: hope is what defines humanity
New sentence: humanity defines what is hope
When you were first learning algebraic expressions, your teacher probably gave you a problem like this, and said, "What is the result?"

\[ 5 \times 4 - \frac{8}{2} + 9 \]

The class got all sorts of different answers, because no one knew the order of operations yet (the correct answer is 25, by the way). Parenthesis become necessary as well (e.g., \( 10 / (8-3) \)).

This is a somewhat annoying problem — it would be nice if there were a better way to do arithmetic so we didn't have to worry about order of operations and parenthesis.

As it turns out, there is a "better" way! We can use a system of arithmetic called "postfix" notation — the expression above would become the following:

\[ 5 4 * 8 2 / - 9 + \] Wat?
Postfix Example

5 4 * 8 2 / − 9 +

Postfix notation* works like this: Operands (the numbers) come first, followed by an operator (+, −, *, /, etc.). When an operator is read in, it uses the previous operands to perform the calculation, depending on how many are needed (most of the time it is two).

So, to multiply 5 and 4 in postfix, the postfix is 5 4 *. To divide 8 by 2, it is 8 2 /

There is a simple and clever method using a stack to perform arithmetic on a postfix expression: (talk to your neighbor about how you might do this!)

Read the input and push numbers onto a stack until you reach an operator. When you see an operator, apply the operator to the two numbers that are popped from the stack. Push the resulting value back onto the stack. When the input is complete, the value left on the stack is the result.

*Postfix notation is also called "Reverse Polish Notation" (RPN) because in the 1920s a Polish logician named Jan Łukasiewicz invented "prefix" notation, and postfix is the opposite of postfix, and therefore so-called "Reverse Polish Notation"
// Postfix arithmetic, implementing +, -, *, /
#include <iostream>
#include "console.h"
#include "simpio.h"
#include "stack.h"

using namespace std;

const string OPERATORS = "+\-*x/";
const string SEPARATOR = " ";

// function prototypes
double parsePostfix(string expression);
string getNextToken(string &expression);
void performCalculation(Stack<double> &s, char op);
// Postfix arithmetic, implementing +, -, *, /
#include <iostream>
#include "console.h"
#include "simpio.h"
#include "stack.h"
using namespace std;

const string OPERATORS = "+-*x/";
const string SEPARATOR = " ";

// function prototypes
double parsePostfix(string expression);
string getNextToken(string &expression);
void performCalculation(Stack<double> &s, char op);
main():

```c++
int main() {
    string expression;
    double answer;
    do {
        expression = getline("Please enter a postfix expression (blank to quit): ");
        answer = parsePostfix(expression);
        cout << "The answer is: " << answer << endl << endl;
    } while (expression != "");
    return 0;
}
```
```
int main() {
    string expression;
    double answer;
    do {
        expression = getline("Please enter a postfix expression (blank to quit): ");
        if (expression == "") {
            break;
        }
        answer = parsePostfix(expression);
        cout << "The answer is: " << answer << endl << endl;
    } while (true);
    return 0;
}
```
string getNextToken(string &expression) {
    // pull out the substring up to the first space
    // and return the token, removing it from the expression

    string token;
    int sepLoc = expression.find(SEPARATOR);
    if (sepLoc != (int) string::npos) {
        token = expression.substr(0,sepLoc);
        expression = expression.substr(sepLoc+1,expression.size()-sepLoc);
        return token;
    }
    else {
        token = expression;
        expression = "";
        return token;
    }
}
string getNextToken(string &expression) {
    // pull out the substring up to the first space
    // and return the token, removing it from the expression

    string token;
    int sepLoc = expression.find(SEPARATOR);
    if (sepLoc != (int) string::npos) {
        token = expression.substr(0,sepLoc);
        expression = expression.substr(sepLoc+1,expression.size()-sepLoc);
        return token;
    } else {
        token = expression;
        expression = "";
        return token;
    }
}
double parsePostfix(string expression) {
    Stack<double> s;
    string nextToken;

    while (expression != "") {
        // gets the next token and removes it from expression
        nextToken = getNextToken(expression);
        if (OPERATORS.find(nextToken) == string::npos) {
            // we have a number
            double operand = stringToDouble(nextToken);
            s.push(operand);
        } else {
            // we have an operator
            char op = stringToChar(nextToken);
            performCalculation(s, op);
        }
    }
    return s.pop();
}
parsePostfix():

double parsePostfix(string expression) {
    Stack<double> s;
    string nextToken;

    while (expression != "") {
        // gets the next token and removes it from expression
        nextToken = getNextToken(expression);
        if (OPERATORS.find(nextToken) == string::npos) {
            // we have a number
            double operand = stringToDouble(nextToken);
            s.push(operand);
        } else {
            // we have an operator
            char op = stringToChar(nextToken);
            performCalculation(s, op);
        }
    }
    return s.pop();
}
void performCalculation(Stack<double> &s, char op) {
    double result;
    double operand2 = s.pop(); // LIFO!
    double operand1 = s.pop();
    switch (op) {
    case '+': result = operand1 + operand2;
        break;
    case '-': result = operand1 - operand2;
        break;
        // allow "*" or "x" for times
    case '*':
        case 'x': result = operand1 * operand2;
            break;
    case '/': result = operand1 / operand2;
        break;
    }
    s.push(result);
}
**Postfix Example Code**

**performCalculation():**

```c
void performCalculation(Stack<double> &s, char op) {
    double result;
    double operand2 = s.pop(); // LIFO!
    double operand1 = s.pop();
    switch (op) {
        case '+': result = operand1 + operand2;
                    break;
        case '-': result = operand1 - operand2;
                    break;
        // allow "*" or "x" for times
        case '*':
        case 'x': result = operand1 * operand2;
                    break;
        case '/': result = operand1 / operand2;
                    break;
    }
    s.push(result);
}
```

*remember LIFO behavior (subtraction and division are not commutative)*

the result simply gets pushed back on the stack
Please enter a postfix expression (blank to quit): 5 4 * 8 2 / - 9 +
The answer is: 25

Please enter a postfix expression (blank to quit): 1 2 3 4 + + +
The answer is: 10

Please enter a postfix expression (blank to quit): 1 2 3 4 -- -
The answer is: -2

Please enter a postfix expression (blank to quit): 1 2 + 3 * 6 + 2 3 + /
The answer is: 3

Please enter a postfix expression (blank to quit): 2 3 4 + * 6 -
The answer is: 8

Please enter a postfix expression (blank to quit):
World's First Programmable Desktop Computer

The HP 9100A Desktop Calculator: the world’s first programmable scientific desktop computer — really, the first desktop computer

(Wired, Dec. 2000)

- RPN (postfix)
- Special algorithm for trigonometric and logarithmic functions
- Cost $5000 in 1968 ($35,000 today)
The next ADT we are going to talk about is a "queue." A queue is similar to a stack, except that (much like a real queue/line), it follows a "First-In-First-Out" (FIFO) model:

- Insertion into a queue "enqueue()" is done at the back of the queue, and removal from a queue "dequeue()" is done at the front of the queue.
Like the stack, the queue Abstract Data Type can be implemented in many ways (we will talk about some later!). A queue must implement at least the following functions:

- **enqueue(value)** (or **add(value)**) - place an entity onto the **back** of the queue
- **dequeue()** (or **remove()**) - remove an entity from the **front** of the queue
- **front()** (or **peek()**) - look at the entity at the front of the queue, but don’t remove it
- **isEmpty()** - a boolean value, **true** if the queue is empty, **false** if it has at least one element.

(note: a runtime error occurs if a **dequeue()** or **front()** operation is attempted on an empty queue.)

Please look at the Stanford Library Queue reference for other functions (e.g., there is a **back()** function that is analogous to **front()** for the back of the queue -- but no removing the value!)

```cpp
Queue<int> q; // {}, empty queue
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.front() << endl; // -3 (q is {-3, 17})
cout << q.dequeue() << endl; // -3 (q is {17})
```
Queue Examples

There are many real world problems that are modeled well with a queue:

Jobs submitted to a printer go into a queue (although they can be deleted, so it breaks the model a bit)
Ticket counters, supermarkets, etc.
File server - files are doled out on a first-come-first served basis
Call centers (“your call will be handled by the next available agent”)
The LaIR is a queue!
Chris G’s research! Scheduling work between a CPU and a GPU is queue based.

The Scheduling Algorithm

Applications are moved from a main queue into a device sub-queue based on a set of rules (assuming one CPU and one GPU):

- Queue of Applications
  - Faster on GPU?
    - Yes: GPU Free?
      - Yes: Run on GPU
      - No: No starvation
    - No: Don’t know
  - No: Fair

CPU Free?
- Yes: Run on GPU unless CPU data is needed and CPU is free
- No: Estimate Finish time on both devices, and queue on device which will finish kernel first
- Run on CPU
Both the Stanford Stack and Queue classes have a `size()` function that returns the number of elements in the object.

What is the output of the following code?

```cpp
Queue<int> queue;
// produce: {1, 2, 3, 4, 5, 6}
for (int i = 1; i <= 6; i++) {
    queue.enqueue(i);
}
for (int i = 0; i < queue.size(); i++) {
    cout << queue.dequeue() << " ";
}
cout << queue.toString() << " size " << queue.size() << endl;
```

A. 1 2 3 4 5 6 {} size 0  
B. 1 2 3 {4,5,6} size 3  
C. 1 2 3 4 5 6 {1,2,3,4,5,6} size 6  
D. none of the above
Both the Stanford Stack and Queue classes have a `size()` function that returns the number of elements in the object. But, you must be careful using it!

What is the output of the following code?

```cpp
Queue<int> queue;
// produce: {1, 2, 3, 4, 5, 6}
for (int i = 1; i <= 6; i++) {
    queue.enqueue(i);
}
for (int i = 0; i < queue.size(); i++) {
    cout << queue.dequeue() << " ";
}
cout << queue.toString() << " size " << queue.size() << endl;
```

A. 1 2 3 4 5 6 {} size 0
B. 1 2 3 {4,5,6} size 3
C. 1 2 3 4 5 6 {1,2,3,4,5,6} size 6
D. none of the above
If you are going to empty a stack or queue, a very good programming idiom is the following:

```cpp
Queue<int> queue;
// produce: {1, 2, 3, 4, 5, 6}
for (int i = 1; i <= 6; i++) {
    queue.enqueue(i);
}
while (!queue.isEmpty()) {
    cout << queue.dequeue() << " ";
}
cout << queue.toString() << " size " << queue.size() << endl;
```

A. 1 2 3 4 5 6 {} size 0
B. 1 2 3 {4,5,6} size 3
C. 1 2 3 4 5 6 {1,2,3,4,5,6} size 6
D. none of the above
If you are going to go through a stack or queue once for the original values, a very good programming idiom is the following:

```cpp
int origQSize = queue.size();
for (int i=0; i < origQSize; i++) {
    int value = queue.dequeue();
    cout << value << " ";
    // re-enqueue even values
    if (value % 2 == 0) {
        queue.enqueue(value);
    }
}
```

Output:

```
1 2 3 4 5 6 {2, 4, 6} size 3
```
If you are going to go through a stack or queue once for the original values, a very good programming idiom is the following:

```cpp
int origQSize = queue.size();
for (int i=0; i < origQSize; i++) {
    int value = queue.dequeue();
    cout << value << " ";
    // re-enqueue even values
    if (value % 2 == 0) {
        queue.enqueue(value);
    }
}
```

Output:

```
1 2 3 4 5 6 {2, 4, 6} size 3
```
Recap

• **Vectors Review:**
  • Stored as arrays under the hood -- you should be cognizant of the ramifications of inserting or removing from the middle of the Vector.
  • But, the Vector *ADT* does not require an array! We could have Moon Monkeys.

• **Abstract Data Types:**
  • An ADT is a set of behaviors that the underlying code must produce, but how the underlying code is written does not affect the ADT behavior (but it might affect the speed!)

• **Stacks:**
  • Stacks are Last-In-First-Out (LIFO) and will have push(*value*), pop(), top(), and isEmpty().

• **Queues:**
  • Queues are First-In-First-Out (FIFO) and will have enqueue(*value*), dequeue(), front(), and isEmpty().
References and Advanced Reading

References:
• Stanford Stack reference: http://stanford.edu/~stepp/cppdoc/Stack-class.html
• Stanford Queue reference: stanford.edu/~stepp/cppdoc/Queue-class.html

Advanced Reading:
• Hewlett-Packard 9100A: http://en.wikipedia.org/wiki/Hewlett-Packard_9100A
• Reverse Polish Notation: http://en.wikipedia.org/wiki/Reverse_Polish_notation
• Standard queue library (not Stanford) reference: www.cplusplus.com/reference/queue/queue
• Chris G's dissertation (you will be one of few people to actually read it!)
• All about memory caching: http://www.hardwaresecrets.com/how-the-cache-memory-works/
What is wrong with the following code?

```cpp
int main() {
    for (int i=0; i < 10; i++) {
        if (i % 2 == 0) {
            cout << i << endl;
        }
    }
    return 0;
}
```

In Qt Creator, this is what happens when you compile:

```
Issues
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>![Issues Icon]</td>
<td>![Warning Icon]</td>
<td>![Error Icon]</td>
<td>![Tip Icon]</td>
<td>![Info Icon]</td>
</tr>
<tr>
<td>expected '}'</td>
<td>cool_project.cpp</td>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>to match this '{'</td>
<td>cool_project.cpp</td>
<td>15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

How does Qt Creator know that there are un-matched curly braces? A stack!
A Stack example algorithm: determine if a program has properly matched bracket symbols — parentheses, square brackets, and curly brackets: 

```
( ) [ ] { }
```

Algorithm: Think about it for a few minutes -- talk to your neighbor!

- Make an empty stack.
- Start reading characters.
- If the character is an opening symbol, push it onto the stack.
- If it is a closing symbol, then if the stack is empty, report an error. Otherwise, pop the stack.
- If the symbol popped is not the corresponding opening symbol, then report an error.
- At the end of the input, if the stack is not empty, report an error.

See code from lecture for full program.