The Stack and Queue Types

Lecture 10

Hartmut Kaiser

hkaiser@cct.lsu.edu

http://www.cct.lsu.edu/~hkaiser/spring_2015/csc1254.html
Programming Principle of the Day

• Do the simplest thing that could possibly work
  • A good question to ask one’s self when programming is “What is the simplest thing that could possibly work?”
  • This helps keep us on the path towards simplicity in the design.

Abstract

- This lecture will focus on two other sequential data types, the stack and the queue. We will use stacks to implement conversion between ‘normal expressions’ and the equivalent reverse polish notation.
Introduction to Stacks

• A stack is a last-in-first-out (LIFO) data structure
• Limited access vector (or list)

• Main operations:
  • Adding an item
    • Referred to as pushing it onto the stack
  • Removing an item
    • Referred to as popping it from the stack
Introduction to Stacks

• Definition:
  • An ordered collection of data items
  • Can be accessed at only one end (the top)

• Operations:
  • Construct a stack (usually empty)
  • Check if it is empty
  • push: add an element to the top
  • top: retrieve the top element
  • pop: remove the top element
  • size: returns number of elements in stack
Introduction to Stacks

• Useful for
  • Reversing a sequence
  • Managing a series of undo-actions
  • Tracking history when browsing the web
  • Function call hierarchy is implemented with a stack
Push

- *Push* means place a new data element at the top of the stack
Push (cont.)

- **Push** means place a new data element at the top of the stack
Push (cont.)

- *Push* means place a new data element at the top of the stack
Push (cont.)

- **Push** means place a new data element at the top of the stack

![Diagram of stack with elements: 3, 11, 5, 17]
Pop

- *Pop* means take a data element off the top of the stack
Pop (cont.)

- **Pop** means take a data element off the top of the stack
Pop (cont.)

- *Pop* means take a data element off the top of the stack
Pop (cont.)

- *Pop* means take a data element off the top of the stack
Top

- *Top* means retrieve the top of the stack without removing it.
Top (cont.)

- \textit{Top} means retrieve the top of the stack without removing it.
Top (cont.)

- **Top** means retrieve the top of the stack without removing it.
Linked-List Stack

- Stacks can also be implemented with a linked list
- The front node is the top of the stack
- In fact, there is `std::stack<>` which is an adaptor usable with different types of underlying containers (`std::list` is one possibility)
- `std::stack<T>`: implements a stack of “T”s

```cpp
t T top();
void push(T const&);
void pop();
std::stack<T>::size_type size() const;
bool empty();
```
Linked-List Stack (cont.)

- To pop, we remove the node at the front of the linked list, and return the element to the client...
Linked-List Stack (cont.)

- To pop, we remove the node at the front of the linked list, and return the element to the client...
Linked-List Stack (cont.)

- To push, we place the new element in a node and insert it at the front of the linked list...
Linked-List Stack (cont.)

- To push, we place a new element in a node and insert it at the front of the linked list...
Example: Implementing a Stack

• Implementing a stack on top of a list is trivial
  • Live coding
**Application of Stacks**

Consider the arithmetic statement in the assignment:

\[ x = a \times b + c \]

Compiler must generate machine instructions:

- LOAD \( a \)
- MULT \( b \)
- ADD \( c \)
- STORE \( x \)

Note: this is "infix" notation. The operators are between the operands.
Reverse Polish Notation

• Postfix Notation
  • Most compilers convert an expression in *infix* notation to *postfix*
  • The operators are written after the operands
  • So \( a * b + c \) becomes \( a b * c + \)
  • Advantage:
    • Expressions can be written without parentheses
## Postfix and Prefix Examples

<table>
<thead>
<tr>
<th>Infix</th>
<th>RPN (Postfix)</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>A + B</td>
<td>A B +</td>
<td>+ A B</td>
</tr>
<tr>
<td>A * B + C</td>
<td>A B * C +</td>
<td>+ * A B C</td>
</tr>
<tr>
<td>A * (B + C)</td>
<td>A B C + *</td>
<td>* A + B C</td>
</tr>
<tr>
<td>A – (B – (C – D))</td>
<td>A B C D – – –</td>
<td>– A – B – C – D</td>
</tr>
<tr>
<td>A – B – C – D</td>
<td>A B – C – D –</td>
<td>– – – A B C D</td>
</tr>
</tbody>
</table>

Prefix: Operators come before the operands
Evaluating RPN Expressions

"By hand" (Underlining technique):

1. Scan the expression from left to right to find an operator.
2. Locate ("underline") the last two preceding operands and combine them using this operator.
3. Repeat until the end of the expression is reached.

Example:

\[ 2 \ 3 \ 4 \ + \ 5 \ 6 \ - \ - \ * \]
Evaluating RPN Expressions

\[
2 \ 3 \ 4 \ + \ 5 \ 6 \ - \ - \ * \\
\rightarrow \ 2 \ 3 \ 4 \ + \ 5 \ 6 \ - \ - \ * \\
\rightarrow \ 2 \ 7 \ 5 \ 6 \ - \ - \ * \\
\rightarrow \ 2 \ 7 \ 5 \ 6 \ - \ - \ * \\
\rightarrow \ 2 \ 7 \ -1 \ - \ * \\
\rightarrow \ 2 \ 7 \ -1 \ - \ * \\
\rightarrow \ 2 \ 8 \ * \\
\rightarrow \ 2 \ 8 \ * \\
\rightarrow \ 16
\]

\[
2 \ * \ ((3 \ + \ 4) - (5 - 6))
\]
Evaluating RPN Expressions

• By using a stack algorithm
  • Initialize an empty stack

• Repeat the following until the end of the expression is encountered
  • Get the next token (const, var, operator) in the expression
  • Operand – push onto stack
  • Operator – do the following
    • Pop 2 values from stack
    • Apply operator to the two values
    • Push resulting value back onto stack

• When end of expression encountered, value of expression is the (only) number left in stack

Note: if only 1 value on stack, this is an invalid RPN expression
Evaluating of Postfix

- Note the changing status of the stack
Converting between Notations

• *By hand:* Represent infix expression as an *expression tree*:

\[ A \times B + C \]

\[ A \times (B + C) \]

\[ ((A + B) \times C) \div (D - E) \]
Converting RPN (Postfix)

- Traverse the tree in *Left-Right-Parent* order (*post-order*) to get **RPN**:
Converting to Prefix

• Traverse tree in *Parent-Left-Right* order (*pre-order*) to get *prefix*:

```
          /
         /  *
        /    
       /     
      /      
    /        
   / A B C - D E
```

```
+ * + A B C - D E
```
Converting to Infix

- Traverse tree in *Left-Parent-Right* order (*in-order*) to get *infix* (must insert parenthesis)

\[((A + B)\times C)/(D - E)\]
Another RPN Conversion Method

*By hand*: "Fully parenthesize-move-erase" method:

1. Fully parenthesize the expression.
2. Replace each right parenthesis by the corresponding operator.
3. Erase all left parentheses.

\[
\begin{align*}
A \times B + C & \rightarrow ((A \times B) + C) \\
& \rightarrow (A \times (B + C)) \\
A \times (B + C) & \rightarrow (A \times (B + C)) \\
& \rightarrow (A \times (B + C)) \\
& \rightarrow A \times (B + C)
\end{align*}
\]
Introduction to Queue

• A queue is a first-in-first-out (FIFO) data structure
• Limited access vector (or list)

• Main operations:
  • Adding an item
    • Referred to as pushing it onto the queue (enqueue an item, adding to the end)
  • Removing an item
    • Referred to as popping it from the queue (dequeue an item, remove from the front)
Introduction to Queues

• Definition:
  • An ordered collection of data items
  • Can be read at only one end (the front) and written to only at the other end (the back)

• Operations:
  • Construct a queue (usually empty)
  • Check if it is empty
  • push: add an element to the back
  • front: retrieve the front element
  • pop: remove the front element
  • size: returns number of elements in queue
Introduction to Queues

• Useful for
  • All kind of simulations (traffic, supermarket, etc.)
  • Computers use queues for scheduling
    • Handling keyboard events
    • Handling mouse events
    • Scrolling the screen
    • Printer spooler
  • C++ I/O streams are queues
    • Even if we only access one end, the other end is managed by the operating system
Example: Queue of Strings

```cpp
void manage_queue()
{
    queue<string> waiting_strings; // queue of 'waiting' strings
    while (true) {
        cout << "?> "; // ask for next line of text
        string response;
        getline(cin, response);

        if (response.empty()) break;
        if (response == "next") { // try to dequeue
            if (waiting_strings.empty())
                cout << "No one waiting!" << endl;
            else {
                cout << waiting_strings.front() << endl;
                waiting_strings.pop();
            }
        }
        else { // enqueue the line read
            waiting_strings.push(response);
        }
    }
}
```
Example: Implementing a Queue

• Implementing a queue on top of a list is equally trivial
  • Live coding