

Stacks

A *linear* data structure is one which is ordered. There are two special types with restricted access: a stack and a queue.

11.1 Stacks Basics

A *stack* is a data structure of ordered items such that items can be inserted and removed only at one end (called the *top*). It is also called a *LIFO* structure: last-in, first-out.

The standard (and usually only) modification operations are:

- **push**: add the element to the top of the stack
- **pop**: remove the top element from the stack and return it

If the stack is empty and one tries to remove an element, this is called *underflow*. Another common operation is called **peek**: this returns a reference to the top element on the stack (leaving the stack unchanged).

A simple stack algorithm could be used to *reverse* a word: push all the characters on the stack, then pop from the stack until it's empty.

$$\text{t h i s} \rightarrow \begin{array}{|c|} \hline \text{s} \\ \hline \text{i} \\ \hline \text{h} \\ \hline \text{t} \\ \hline \end{array} \rightarrow \text{s i h t}$$

11.2 Implementation

A stack is commonly and easily implemented using either an array or a linked list. In the latter case, the head points to the top of the stack: so addition/removal (push/pop) occurs at the head of the linked list.

11.3 Application: Balanced Brackets

A common application of stacks is the parsing and evaluation of arithmetic expressions. Indeed, compilers use a stack in *parsing* (checking the syntax of) programs.

Consider just the problem of checking the brackets/parentheses in an expression. Say $[(3+4)*(5-7)]/(8/4)$. The brackets here are okay: for each left bracket there is a matching right bracket. Actually, they match in a specific way: two pairs of matched brackets must either nest or be disjoint. You can have $[()]$ or $[]()$, but not $([])$

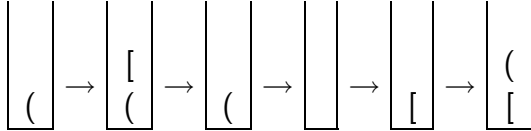
We can use a stack to store the unmatched brackets. The algorithm is as follows:

Scan the string from left to right, and for each char:

1. If a left bracket, push onto stack
2. If a right bracket, pop bracket from stack
(if not match or stack empty then fail)

At end of string, if stack empty and always matched, then accept.

For example, suppose the input is: $([])[()]$ Then the stack goes:



and then a bracket mismatch occurs.

11.4 Application: Evaluating Arithmetic Expressions

Consider the problem of evaluating the expression: $((3+8)-5)*(8/4)$. We assume for this that the brackets are compulsory: for each operation there is a surrounding bracket. If we do the evaluation by hand, we could:

repeatedly evaluate the first closing bracket and substitute

$$(((3+8)-5)*(8/4)) \rightarrow ((11-5)*(8/4)) \rightarrow (6*(8/4)) \rightarrow (6*2) \rightarrow 12$$

With two stacks, we can evaluate each subexpression when we reach the closing bracket:

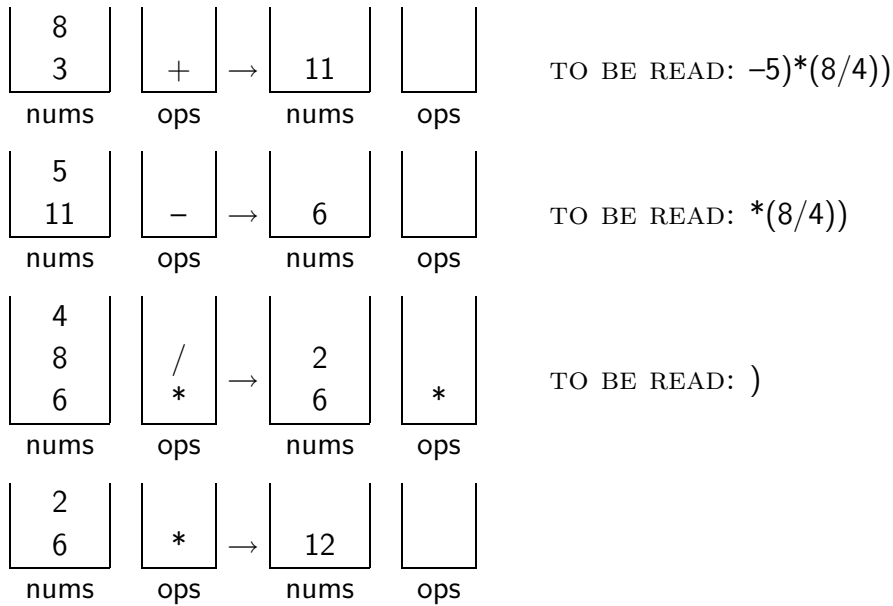
Algorithm (assuming brackets are correct!) is as follows:

Scan the string from left to right and for each char:

1. If a left bracket, do nothing
2. If a number, push onto *numberStack*
3. If an operator, push onto *operatorStack*
4. If a right bracket, do an evaluation:
 - a.) pop from the *operatorStack*
 - b.) pop two numbers from the *numberStack*
 - c.) perform the operation on these numbers (in the right order)
 - d.) push the result back on the *numberStack*

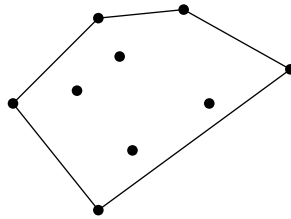
At end of string, the single value on the *numberStack* is the answer.

The above example $((3+8)-5)*(8/4)$: at the right brackets



11.5 Application: Convex Hulls

The *convex hull* of a set of points in the plane is a polygon. One might think of the points as being nails sticking out of a wooden board: then the convex hull is the shape formed by a tight rubber band that surrounds all the nails. For example, the highest, lowest, leftmost and rightmost points are on the convex hull. It is a basic building block of several graphics algorithms.



One algorithm to compute the convex hull is Graham's scan. It is an application of a stack. Let 0 be the leftmost point (which is guaranteed to be in the convex hull) and number the remaining points by angle from 0 going counterclockwise: 1, 2, ..., n - 1. Let nth be 0 again.

GRAHAM SCAN

1. Sort points by angle from 0
2. Push 0 and 1. Set $i=2$
3. While $i \leq n$ do:
 - If i makes left turn w.r.t. top 2 items on stack
 - then { push i ; $i++$ }
 - else { pop and discard }

We do not attempt to prove that the algorithm works. The running time: Each time the while loop is executed, a point is either stacked or discarded. Since a point is looked at only once, the loop is executed at most $2n$ times. There is a constant-time method for checking, given three points in order, whether the angle is a left or a right turn. This gives an $O(n)$ time algorithm, apart from the initial sort which takes time $O(n \log n)$.

One day I'll add an example.

11.6 Sample Code: ArrayStack and Balanced

Here is code for an array-based stack, and a balanced brackets tester.

```
// ArrayStack.h
// wdg 2009
#ifndef ARRAYSTACK_H
#define ARRAYSTACK_H

const int MAX_STACK = 100;
typedef int StackType;
const StackType ERROR = -1;

class ArrayStack
{
public:
    ArrayStack( );           // constructor
    void push(StackType item); // pushes object onto stack
    bool isEmpty() const;    // returns whether stack is empty or not
    StackType pop();         // pops top element from stack
    StackType peek() const;  // returns value of data on top of stack
    void dump() const;       // outputs representation on stdout

private:
    StackType arr[MAX_STACK]; // stores data
    int count;                // number of elements in stack
    // note that valid data always in 0..count-1
};
#endif
```

```
// ArrayStack.cpp
// minimal array implementation of Stack; no test for overflow
```

```

// wdg 2009
#include <iostream>
using namespace std;
#include "ArrayStack.h"

// constructor
// arbitrarily sets upper limit
ArrayStack::ArrayStack( ) : count(0)
{
    // arr not initialized
}

// pushes object onto stack
// @post: reference to item placed at right-end of array
void ArrayStack::push( StackType item )
{
    arr[count++] = item;    // count incremented after array access
}

//@ returns whether stack is empty or not
bool ArrayStack::isEmpty( ) const
{
    return (count==0);
}

// pops top element from stack
// @returns: previous top element or ERROR if problem
// @post: the top element is removed
StackType ArrayStack::pop( )
{
    if( isEmpty() )
        return ERROR;
    else
        return arr[--count];    // count decremented before array access
}

// @returns reference to data on top of stack
// or ERROR (constant in header) if stack is empty
StackType ArrayStack::peek( ) const

```

```

{
    if( isEmpty() )
        return ERROR;
    else
        return arr[count-1];
}

// outputs representation
void ArrayStack::dump() const
{
    if( isEmpty() )
        cout << "-empty-" << endl;
    else {
        for(int x=0; x<count; x++)
            cout << arr[x] << " ";
        cout << endl;
    }
}

```

```

// Balanced.cpp
// a hurried implementation of balanced brackets
#include <iostream>
using namespace std;
#include "ArrayStack.h"

bool test(char *argg); // uses C-strings

int main(int argc, char *argv[])
{
    if( argc==1)
        cout << "Add string on command-line! (in quotes)" << endl;
    else
        cout << boolalpha << test( argv[1] ) << endl;
    return 0;
}

bool test(char *argg) // accepts balanced strings of ()[]{}
{
    ArrayStack S;
    char D;

```

```

while ( *argg ) {
    switch( *argg ) {

        case '[': case '{': case '(':
            S.push( *argg );
            break;

        case ']':
            if( S.isEmpty() )
                return false;
            D = S.pop();
            if( D!='[' )
                return false;
            break;

        case '}':
            if( S.isEmpty() )
                return false;
            D = S.pop();
            if( D!='{' )
                return false;
            break;

        case ')':
            if( S.isEmpty() )
                return false;
            D = S.pop();
            if( D!='(' )
                return false;
            break;

        default:
            return false;
    } // end switch
    argg++;
} // end while

return S.isEmpty(); // return true if reach here with empty stack
}

```