Lecture 6 Feb 3, 2014

Goals:

stacks

- Implementation of stack

applications

- Postfix expression evaluation
- Convert infix to postfix

Stack Overview

Stack ADT

Basic operations of stack

- push, pop, top, isEmpty etc.

Implementations of stacks using

- Array
- Linked list

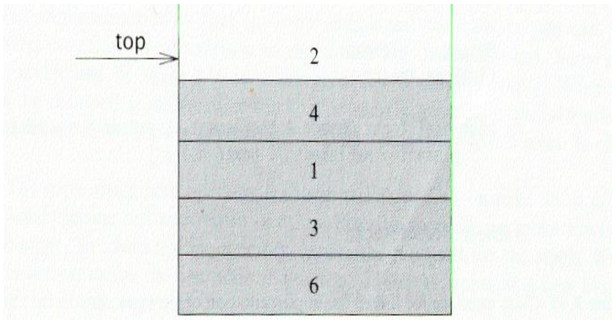
(recall the abstract data type vs. concrete data structures that implement them.)

Application to arithmetic expression evaluation

(both steps: converting to postfix, postfix expression evaluation)

Stack ADT

- A *stack* is a list in which insertion and deletion take place at the same end
  - This end is called *top*
  - The other end is called *bottom*



- Stacks are known as LIFO (Last In, First Out) lists.
  - The last element inserted will be the first to be retrieved

Push and Pop

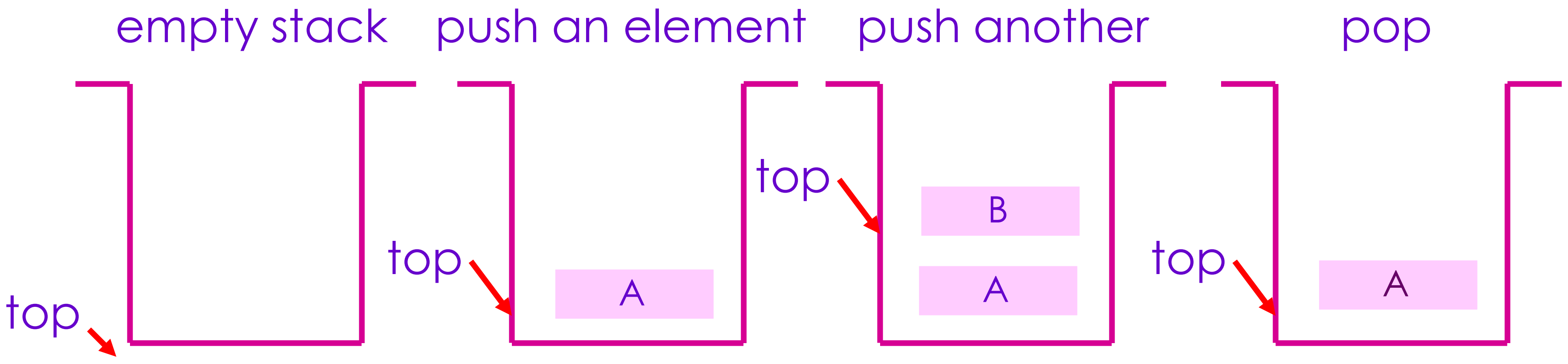
Primary operations: Push and Pop

Push

- Add an element to the top of the stack

Pop

- Remove the element at the top of the stack



Implementation of Stacks

Any list implementation could be used to implement a stack

- arrays (static: the size of stack is given initially)
- Linked lists (dynamic: never becomes full)

We will explore implementation based on array.

Stack class

```
class Stack {
public:
      Stack(int size = 10);                  //
constructor
      ~Stack() { delete [] values; }         //
destructor
      bool IsEmpty() { return top == -1; }
      bool IsFull() { return top == maxTop; }
      double Top();    // examine, without popping
      void Push(const double x);
      double Pop();
      void DisplayStack();
private:
      int maxTop;          // max stack size = size - 1
      int top;             // current top of stack
      double* values;      // element array
};
```

Stack class

Attributes of Stack

- maxTop: the max size of stack
- top: the index of the top element of stack
- values: point to an array which stores elements of stack

Operations of Stack

- IsEmpty: return true if stack is empty, return false otherwise
- IsFull: return true if stack is full, return false otherwise
- Top: return the element at the top of stack
- Push: add an element to the top of stack
- Pop: delete the element at the top of stack
- DisplayStack: print all the data in the stack

Create Stack

The constructor of Stack

- Allocate a stack array of size. By default, size = 10.
- Initially top is set to -1. It means the stack is empty.
- When the stack is full, top will have its maximum value, i.e. size – 1.

```
Stack::Stack(int size /*= 10*/) {
      values      =     new double[size];
      top         =     -1;
      maxTop      =     size - 1;
}
```

Although the constructor dynamically allocates the stack array, the stack is still static. The size is fixed

Push Stack

`void Push(const double x);`

Push an element onto the stack

Note top always represents the index of the top element. After pushing an element, increment top.

```
void Stack::Push(const double x) {
      if (IsFull()) // if stack is full, print error
       cout << "Error: the stack is full." << endl;
      else
            values[++top]     =     x;
}
```

Pop Stack

`double Pop()`

Pop and return the element at the top of the stack

Don’t forget to decrement top

```
double Stack::Pop() {
      if (IsEmpty()) { //if stack is empty, print error
       cout << "Error: the stack is empty." << endl;
            return -1;
      }
      else {
            return values[top--];
      }
}
```

Stack Top

`double Top()`

- Return the top element of the stack
- Unlike Pop, this function does not remove the top element

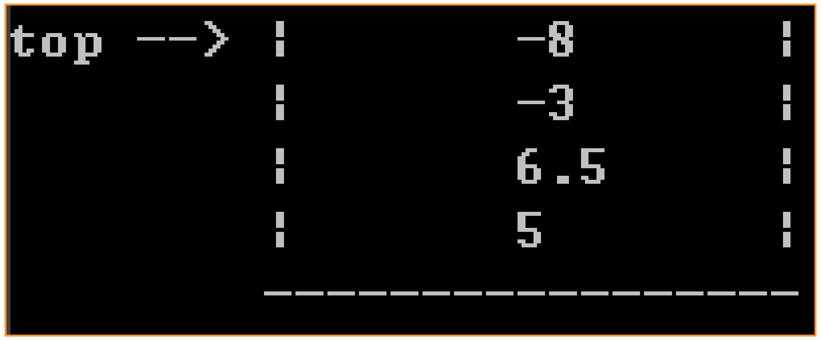
```
double Stack::Top() {
      if (IsEmpty()) {
       cout << "Error: the stack is empty." << endl;
            return -1;
      }
      else
            return values[top];
    }
```

Printing all the elements

`void DisplayStack()`

Print all the elements

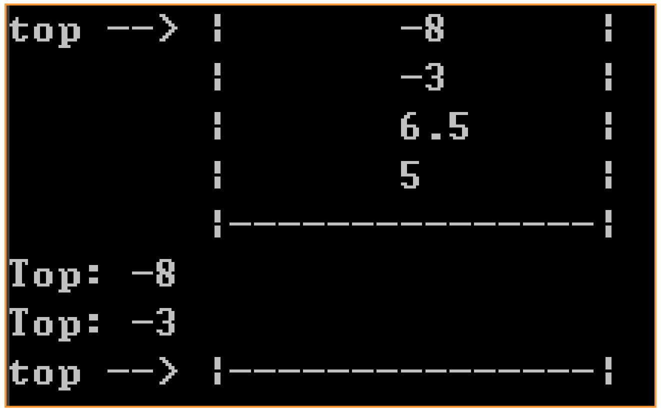
```
void Stack::DisplayStack() {
      cout << "top -->";
      for (int i = top; i >= 0; i--)
            cout << "\t|\t" << values[i] << "\t|" << endl;
      cout << "\t|---------------|" << endl;
}
```



```
top -->  |        -8        |
         |        -3        |
         |        6.5       |
         |        5         |
         -------------------
```

Using Stack

result



```
top --> |       -8       |
        |       -3       |
        |       6.5      |
        |       5        |
        |----------------|
Top: -8
Top: -3
top --> |----------------|
```

```
int main(void) {
        Stack stack(5);
        stack.Push(5.0);
        stack.Push(6.5);
        stack.Push(-3.0);
        stack.Push(-8.0);
        stack.DisplayStack();
        cout << "Top: " << stack.Top() << endl;

        stack.Pop();
        cout << "Top: " << stack.Top() << endl;
        while (!stack.IsEmpty()) stack.Pop();
        stack.DisplayStack();
        return 0;
}
```

Application 1: Balancing Symbols

To check that every right brace, bracket, and parentheses must correspond to its left counterpart

e.g. [( )]{ } is legal, but {[( ])} is illegal

Formal definition of balanced expressions:

A balanced expression over the symbols [ ] { } ( ) is a string using these characters that is recursively defined as follows:

(base) { }, [ ] and ( ) are balanced expression (rule 1)

(induction) if R is balanced, so are (R), [R] and {R}. If R and S are balanced, then so is R.S (concatenation) (rules 2a and 2b)

(exclusion) Any balanced expression can be obtained by applying the above rules a finite number of times. (rule 3)

You can check using above definition that
{ [ ] ( ) } [ ] is balanced.

Formal Proof:

1. [ ] is balanced (rule 1)
2. () is balanced (rule 1)
3. [] () is balanced (rule 2b applied to 1 & 2)
4. { [ ] () } is balanced (rule 2a applied to 3)
5. [ ] is balanced (rule 1)
6. {[ ] ( )} [ ] is balanced (rule 2b applied to 3 and 5)

How do you formally prove that some expression is NOT balanced?

Algorithm for balanced expression testing

To check that every right brace, bracket, and parentheses must correspond to its left counterpart

e.g. [( )]{ } is legal, but {[( ])} is illegal

Algorithm (returns true or false)

```
(1)  Make an empty stack. If the input string is of length 0,
     return false.
(2)  Read characters until end of file
     i.   If the character is an opening symbol, push it onto the stack
     ii.  If it is a closing symbol, then if the stack is empty, report false
     iii. Otherwise, pop the stack. If the symbol popped is not the
          corresponding opening symbol, then report false
(3)  At end of file, if the stack is not empty, report false
(4)  Report true
```

Application 2: Expression evaluation

Given an arithmetic expression such as:

x + y * (z + w)

(Given values assigned to x = 23, y = 12, z = 3 and w = -4)

What is the value of the expression?

Goal: Design a program that takes as input an arithmetic expression and evaluates it.

This task is an important part of compilers. As part of this evaluation, the program also needs to check if the given expression is correctly formed.

Example of bad expressions:

(3 + 12 * (5 – 3)

A + B * + C etc.

Postfix expression

Instead of writing the expression as A + B, we write the two operands, then the operator.

Example: a b c + *

It represents a*(b + c)

Question: What is the postfix form of the expression a+ b*c?

Postfix expression

Instead of writing the expression as A + B, we write the two operands, then the operator.

Example: a b c + *

It represents a*(b + c)

Question: What is the postfix form of the expression a+ b*c?

Answer: a b c * +

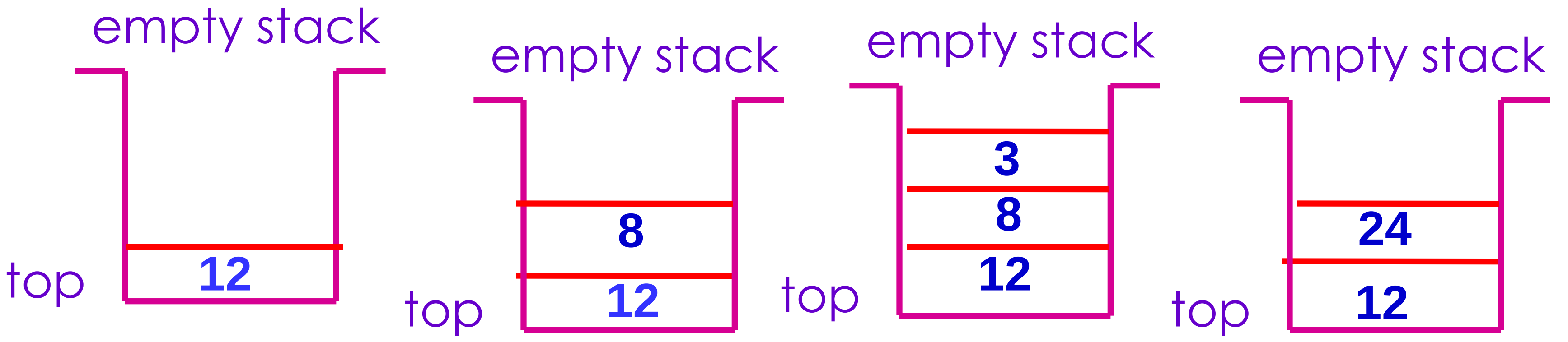
Algorithm for evaluating postfix expression

- Use a stack.
- Push operands on the stack.
- When you see an operator, pop off the top two elements of the stack, apply the operator, push the result back.
- At the end, there will be exactly one value left on the stack which is the final result.

Algorithm for evaluating postfix expression

- Use a stack.
- Push operands on the stack.
- When you see an operator, pop off the top two elements of the stack, apply the operator, push the result back.
- At the end, there will be exactly one value left on the stack which is the final result.

Example: 12 8 3 * +



Finally, the result 36 is pushed back on the stack.

Implementation of exp evaluation

token class:

```
class token {
 private: int op_type;
           double value;
  public:
     token(int x, int y) {
         op_type = x; op_value = y;
      }
     int get_op_type() {
        return op_type;
      }
      double get_value() {
        return value;
     }
     void set_op_type(int x) { op_type = x;
     }

     void set_value(double y) {value = y;
   }
 };
```

Op_type:

1 → +

2 → -

3 → *

4 → /

5 → **

6 → operand

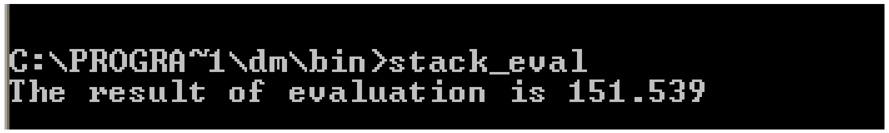
–1 → token represents end of expression

op_value: value of the operand.

Input

Look at the main program:

```
int main(void) {
  string str = "908 100 200+   23 19 * +/ 123 *";
  Expr ex(str, 0);
  double rslt = ex.eval();
  cout << "The result of evaluation is " << rslt << endl;
  return 0;
};
```



```
C:\PROGRA~1\dm\bin>stack_eval
The result of evaluation is 151.539
```

There must be a space between successive operands. There need not be a space when an operand follows an operator, and after an operator. There can be more than one space after any token, including the last.

Implementation of expression evaluation

```
    double eval() {
     // assumes that the postfix expression is correct
     // also unary minus is not allowed. Operands have to be integers
     // although the final result can be non-integral
         Stack st(MAX_SIZE);
         token tok = get_token();  //gets the next token
         while (tok.get_op_type() != -1) {
          if (tok.get_op_type() == 6)
            st.Push(tok.get_value());
          else {
            double opd2 = st.Pop();
            double opd1 = st.Pop();
            double op = apply(tok.get_op_type(), opd1, opd2);
            st.Push(op);
          }
          current++; tok = get_token();
         }
        double result = st.Pop(); return result;
      } // eval
  }; // end Expr
```

Code for get_token

```
     token get_token() {
      token tok( -1, 0);
      if (current > exp.length() - 1)
            return tok;
     while (exp[current] == ' ') current++;
      if (current > exp.length() - 1) return tok;
      if (exp[current] == '+') tok.set_op_type(1);
       else if (exp[current] == '-') tok.set_op_type(2);
       else if (exp[current] == '/') tok.set_op_type(4);
       else if (exp[current] == '*') {
             if (exp[current+1] != '*') tok.set_op_type(3);
                    else {tok.set_op_type(5); current++;}
             }
       else { // token is an operand
         double temp = 0.0;
         while (!(exp[current] == ' ') && !optr(exp[current])) {
                                temp= 10*temp+val(exp[current]); current++; }
                   if (optr(exp[current])) current--;
                     tok.set_op_type(6);
                     tok.set_value(temp);
             }
         return tok;
   } //end get_token
```

Converting infix to Postfix expression

Recall the postfix notation from last lecture.
Example: a b c + *

It represents a*(b + c)

What is the postfix form of the expression a + b*(c+d)?

Answer: a b c d + * +

Observation 1: The order of operands in infix and postfix are exactly the same.

Observation 2: There are no parentheses in the postfix notation.

Example :

(a) Infix: 2 + 3 – 4 Postfix: 2 3 + 4 –

(b) Infix: 2 + 3 * 4 Postfix: 2 3 4 * +

The operators of the same priority appear in the same order, operator with higher priority appears before the one with lower priority.

Rule: hold the operators in a stack, and when a new operator comes, push it on the stack if it has higher priority. Else, pop the stack off and move the result to the output until the stack is empty or an operator with a lower priority is reached. Then, push the new operator on the stack.

Applying the correct rules on when to pop

Assign a priority: (* and / have a higher priority than + and – etc.)

Recall: Suppose st.top() is + and next token is *, then * is pushed on the stack.

However, (behaves differently. When it enters the stack, it has the highest priority since it is pushed on top no matter what is on stack. However, once it is in the stack, it allows every symbol to be pushed on top. Thus, its in-stack-priority is lowest.

We have two functions, ISP (in-stack-priority) and ICP (incoming-priority).

In-stack and in-coming priorities

	icp	isp
+, –	1	1
*, /	2	2
**	3	3
(	4	0

Dealing with parentheses

An opening parenthesis is pushed on the stack (always). It is not removed until a matching right parenthesis is encountered. At that point, the stack is popped until the matching (is reached.

Example: (a + b * c + d)* a

Stack: (	Output: a
Stack: (+	Output: a
Stack: (+	Output: a b
Stack: (+ *	Output: a b
Stack: (+ *	Output: a b c
Stack: (+	Output: a b c * +
Stack: (+	Output: a b c * + d
Stack	Output: a b c * + d +
Stack *	Output: a b c * + d +
Stack *	Output: a b c * + d + a

```
string postfix() {
    Stack st(100);
    string str ="";
    token tok = get_token();
    string cur = tok.get_content();
    while (tok.get_op_type() != -1) {
        if (tok.get_value() == 1)
          str+= cur + " ";
        else if (cur == ")") {
             while (st.Top()!= "(")
                   str += st.Pop() +" ";
             string temp1 = st.Pop();
    }
```

```
  else if (!st.IsEmpty()) {
    string temp2 = st.Top();
    while (!st.IsEmpty() && icprio(cur) <= isprio(temp2)) {
      str+= temp2 + " ";
          string temp = st.Pop();
        if (!st.IsEmpty()) temp2 = st.Top();
  }
            }
            if (tok.get_value() != 1 && tok.get_content()!= ")") st.Push(cur);
            current++;
            tok = get_token();
            cur = tok.get_content();
        }
        while (!st.IsEmpty()) {
             str += st.Pop() + " ";
             cout << "string at this point is " << str << endl;
         }
        return str;
       }
```