Trees

Some basic stack application demos with recursive equivalents
1. Reversing string B A T M A N using a stack. (See DSO sec 7.1)

2. Reversing string B A T M A N using a recursive procedure, visualizing the stack of activation records. Each activation record stores one character. (See

3. Testing string ( ) () () {} [] for properly nested parenthesis using a stack. (See DSO section 7.2, quoted below.)

At this point, we did low-tech dramatizations:
Recursive reverse string printing function

```cpp
void print(char *ch) {
    // print c
    // print (st)
    // Letting st store the rest of s,
    // save first char of s in local variable c
    } else
        if (s != "") do nothing:
    // Pseudocode:
    // post: the reversal of s has been printed
    // pre: pc points to a c-string s
    void print(char *ch) {
```
Let's look at the 2-stack algorithm from DS0 chapter 7.
computes the INTEGER whose value is signified by that digit character.

\[ \text{Char Value} = \text{MCHARR} - \text{CH} \]

\[
\text{isdigit} (\text{MCHARR}) == \text{false} \text{ so } \text{MCHARR} = \text{CH} \]

and

\[ \text{cin} >> \text{MCHARR} \text{ or } \text{MCHARR} = \text{but}[0] \]

\[ \text{char but}[\text{SIZE}] \]

A practical tip: How to evaluate a decimal digit? Suppose..
The purpose is to illustrate a fundamental use of stacks.

expression as input, and evaluates the arithmetic expression.

Basic calculator program that takes a fully parenthesized arithmetic

FILE: calc.cxx
\{ 
\{ 
    default: return '0', 
    you finish it! //
    case ',': return ',',)
\} (c) switch(c)

If not, the return value is 0, i.e., '0', //
the matching right parentheses corresponding to c. //
The return value is the char value representing //
the return value is the char value representing //

then
POST: if c equals any of ',',),', or ']' then //
PRE://
char RightProtocol(char c)

"matches right parentheses R." //
"matches right parentheses R." //
parentheis...RightProtocol(R} == R) is a test if left parentheses l //
(RightProtocol(C)}==0) is a test for C equalling a let //
Another practical tip: A parentheses conversion function,
DOiT (E) ;

} //you finish it!
count >> endl ; //INY: Leans digits of index scale printed.

} count >> (1/10) ;

for ( int i = 0 ; i < strlen ( E ) ; i++ )

} count >> E ;

} count >> "You typed" ;

} return 1 ; /*Input failed, bail out*/

) ( i ) if ( cin.getline ( E , MaxLen + 1 ) )

} count >> "Type a fully parenthesized arithmetic expression." ;

} char E[MaxLen+1] ;

) int main()

) void DoIT ( char *E ) ;

) static const int MaxLen = 100 ;

) using namespace std ;

) #include <iostream>

) #include <fstream>

) TIP: How to begin...(compiled OK 4/6/06, 14:00)

) University at Albany Computer Science Dept.
return 0;
{
struct ValAndPosNode
{
    int pos; // index of last char. of that
    double value; // value of some subexpression.
}

struct ValAndPosNode* next; // addr on next node or NULL if
ValAndPos data;
}

struct ValAndPosNode
{
    double expression in MEXPR;
    int expr; // index of last char. of that
    double value; // value of some subexpression.
}

Classic C-style: Stack impl. with a Linked List.
{
    return ret;
    delete pt;
    VPstackhead = pt->next;
    VPandpos ret = pt->data;
    VPandposNode *pt = VPstackHEAD;
    assert (VPstackHEAD);
}

VPandpos VPpop()
{
    VPstackHEAD = pt;
    pt->next = VPstackHEAD;
    pt->data = vp;
    VPandposNode *pt = new VPandposNode;
}

void VPpush(VPandpos vp)
    VPandposNode *VPstackHEAD; //GLOBAL
char E[EXPRSIZE]; //GLOBAL
#define EXPRSIZE 100

University at Albany Computer Science Dept.
0

6

×

7

}
Data Structure diagram of Linked List implementation of this ValAndPos stack.

Operator stacks after processing the expression $(3 + 4) \times (2 - 6)$. 

MyExpr

- $[0]$
- $[10]$
- $[6]$
- $[8]$
- $[6]$
- $[8]$
- $[2.0]$
- $[7.0]$
- $[6.0]$

ValAndPos

VPStackHEAD

6 in position 10.
void VPPush(ValAndPos vp)

    ValAndPosNode *pt = new ValAndPosNode;

    pt->value = 3.0;
    pt->spos = 2;
    pt->epos = 2;

    //call VPPush with a Value Argument!
    VPPush(TMP);

    CALLER of VPPush

    "new ValAndPosNode!"

    ValAndPosNode *pt = ...;

    } // end VPPush(ValAndPos vp)
of stack

(\textit{local}) automatic

\textbf{Vppush}

\texttt{void VPPush(ValAndPos \texttt{vp})}

\begin{verbatim}
ValAndPosNode *pt = new ValAndPosNode;
pt->data = \texttt{vp};
pt->next = VPStackHEAD;
\end{verbatim}
```c
int VPPush(ValAndPos vp)
{
    ValAndPosNode *pt = VPStackHEAD;
    VPStackHEAD = pt;
    VPStackHEAD = VPStackHEAD * ValAndPos(vp);
    return pt;
}
```
```cpp
{
    {
        \n        you write
    }
}

while (E[i] = \'\0\', i = 0) \n    // index of input queue end.
int i = 0; \n    // for std::stack

stack>charAndpos> operations; // or std::stack
stack>valAndpos> ValPosStack; // or std::stack

void DoExpression(char * E) \n    ...
```

```cpp
#include "charAndpos.h"
#include "valAndpos.h"
#include "stack2.h" from DOS, or #include <stack>
```

Main and Switch Style:
{ 
    VALPOSStack.push(TMP);
    TMP.epochs = 2; // Your program would generate the position
    TMP.spots = 2; // Your program would generate the position
    TMP.value = 3.0; // Your program would compute the value!
    VALADPOS TMP;
    //
    stack<char ADPOS> operations; // or std::stack
    stack<VALADPOS> VALADPosStack; // or std::stack
}

void DoExpression (char * E) {
    ... // Include "CHARDOS.h"
    #include "VALADPOS.h"
    ... // Include <stack> or #include <stack2.h> from DOS, or #include <stack>
    How to push a VALADPos record on this stack:
{  
    // Print a report for debugging.
    print (":" from the ValPortQueue") >> [
    "\" >> TMPSIDES >> "\" >> TMP.REGION
    ",\" position=\" >> TMPLONG
    cout >> \" we popped (value=\" >> TMPLONG
    ValPortQueue.pop();
    TMP = ValPortQueue.top();
    ValPortQueue.TOP
    ...
    ...
    // stack<char>operations;  
    stack<char>ValPortQueue;
}

void DoExpression (char * E)
{
    //CharAndPos.h
    #include "CharAndPos.h"
    #include "ValAndPos.h"
    #include "stack2.h"  
    // OS from DOS, or #include <stack>
    How to pop and use a ValAndPos record on this stack:
    University at Albany Computer Science Dept.
}

20
process given earlier: compare the operation of the 2-stacks algorithm (see the two stacks example)

Why the 2-stacks algorithm works:

\textbf{DSO 7.2} (WITH description of the "by hand")
(false iff checks false)
check toks matches operator
operators.push(result);
operator on operands and operands;
calculate result of
}
or
check operator is +,-,* or /; check operators is ( );
operator = operators.pop();
operator = operators.pop();
operator = operators.pop();
}
else if (is-integer toks)
operators.push(tok);
else if (is-jet toks)
operators.push(tok);
else if (is-operator toks)
operators.push(tok);
else if (is-number toks)
operators.push(tok);
while (tok = next toks)
( )
Each subexpression corresponds to a number stack entry. When two numbers are popped, combined, and pushed, the new stack entry corresponds to the combination of the two old stack entries.
CORRESPONDING to that number.

2. Store in each number stack structure a pointer to root of the subtree pushed in the number stack.

1. Finish building each subtree as soon as the CORRESPONDING number is

IDEAS:

Part 2 of Project 4 GOAL: Build the expression tree.
\((3 + 4) \times 2\)
(Print specificed results at each step)

evaluates the expression tree value evaluated by post-order traversal.

Finally program a TEST whether the SAVED value (from your partl algoritum)

test: and print that tree in post-order form.

4. Evaluate and print that tree in in-order form.

3. Print that tree in in-order form.

2. Print that tree in pre-order form.

1. Count the nodes in the expression tree.

Printed AND SAVED for the TEST, do a tree traversals:

Once the expression tree is built, and the value from the 2-stacks algorithm is
Recursive tree traversal algorithms... Really Easy!

University at Albany Computer Science Dept.
Parentheses!

Prefix and Postfix notations can express every expression without any parentheses. Suppose each operator has its own fixed number of operands, the problems:

**Postfix:** Each operator is written *after* its operand subexpressions.

**Infix:** Each operand is written *between* its operand subexpressions.

**Prefix notation:** Each operator is written *before* its operand subexpressions.

Expressions (as strings) are very smart. Computer scientists know two alternatives to infix notation for writing...
expression in postfix form.

are used by recursive evaluation of an expression tree, you will write the

3. If you write the constants and variables, and the operations, in the order they

(subject to store and retrieve subexpression values (solves problem (2)).

expression solved by push, access and removal by pop

2. A stack of subexpression values (solves Problem (1)).

exactly matches the order of data access when the computer evaluates the

1. The left-to-right order of atoms (constants and variables) and operators

What's cool about Postfix:
Suppose we write down the constants and operations in the order that we use them when we evaluate this expression. First we take 6 + 9 = 15 and then 9, and then add them.

Next we write 3, then divide 6 / 3 = 2, and then add them. First we take 6 - 4 = 2, and then 2.

Here's the result: The expression tree for this postfix notation for this expression:

\[
( ( 4 - 6 ) * ( 3 / ( 6 + 9 ) ) )
\]
We can verify the result is 10 by calculating directly from the expression tree.

We can also verify this from the expression by using elementary school methods:

\[
\begin{align*}
\text{Postfix: } & 6 & 9 & + & 3 & \div & 6 & 4 & - \\
& 6 + 9 &= 15 \\
& 15 \div 3 &= 5 \\
& 6 - 4 &= 2 \\
& 5 \times 2 &= 10
\end{align*}
\]
Example of an expression and its Parse Tree:

```
A = (B = (C - (D * (E + +)) + (F * G)))
```

Details continued on the next 2 frames.
The top level operator is multiplication (*).

(F*G)
The top level operator is subtraction (-).

Expression: `(C-(D*(E++)))`

Diagram:
```
   E
     ↓
  ++
   (++)
     ↓
   *  
   (++)
     ↓
  (D*(E++))
     ↓
(C-(D*(E++)))
```