Stacks: Applications

Atul Gupta

Using Stacks: Parentheses Matching

- Consider the following expression

[a + b *({c/d(1-n) + e/f (1+n)})/(g – sqrt[(b**2) – 4*a*c]/2)]

Parentheses Matching

- Each "(", "{", or "[" must be paired with a matching ")", "}", or "]"
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

```
bool ParenthesesMatch(char[ ] exp) {
    char next_char, popped_char;
    stack S;
    bool valid = true;
     while (not_empty(exp)) {
        next_char  = get_next(exp);
        if ((next_char == '(') or (next_char =='{) or (next_char == '[')) then
            push (S, next_char);
        if ((next_char == ')') or (next_char =='}) or (next_char == ']')) then {
            If (stackEmpty(S)) then
                    return (valid = false);
            else  {
                    popped_char = pop (S);
                    if (! match (next_char, popped_char) )
                            return (valid = false);
              }
        }
    }
    If (not stackEmpty(S)) then
        valid = false;
    return valid;
}
```

Algorithm *ParentheseMatch (exp):*

Input: *An array exp of n characters, each of which is either a parenthesis, a variable, an arithmetic operator, or a number*

Output: **true** *if and only if all the parentheses in exp match*

Lets write the Code

Evaluating Expressions

Infix	Prefix	Postfix
a + b	+ a b	a b +
a + b * c	+ a * b c	a b c * +
(a + b) * (c - d)	* + a b - c d	a b + c d - *
b * b - 4 * a * c		b b * 4 a c * * -
40 - 3 * 5 + 1		

Infix Expression Evaluation

Exp:
(((2 * 5) - (1 * 2)) / (11 - 9))

Input Symbol	Stack (from bottom to top)	Operation
(		
(		
(		
2	2	
*	2 *	
5	2 * 5	
)	10	2 * 5 = 10 and push
-	10 -	
(	10 -	
1	10 - 1	
*	10 - 1 *	
2	10 - 1 * 2	
)	10 - 2	1 * 2 = 2 & Push
)	8	10 - 2 = 8 & Push
/	8 /	
(	8 /	
11	8 / 11	
-	8 / 11 -	
9	8 / 11 - 9	
)	8 / 2	11 - 9 = 2 & Push
)	4	8 / 2 = 4 & Push
New line	Empty	Pop & Print

Infix Expression Evaluation

- Analysis: Five types of input characters
  - Opening bracket
  - Numbers
  - Operators
  - Closing bracket
  - New line character
- Data structure requirement: A character stack

Infix Expression Evaluation

An Algorithm

1. Read one input character at a time
2. switch (input)
   - case (Opening bracket) : Go to step (1)
   - case (Number) : Push into stack and then Go to step (1)
   - case (Operator) : Push into stack and then Go to step (1)
   - case (Closing bracket) : op2 = pop (s)
     op = pop (s)
     op1 = pop (s)
     result = computer (op1 op op2)
     Push result into stack and Go to step (1)
   - case (new line character) : Pop from stack and print the answer
3. STOP

This algorithm doesn't handle errors in the input, although careful analysis of parenthesis or lack of parenthesis could point to such error determination.

Converting Expressions

Example: Converting Infix to Postfix

1. Create an empty stack and an empty postfix output string/stream
2. Scan the infix input string/stream left to right
3. If the current input token is an operand, simply append it to the output string (note the examples above that the operands remain in the same order)
4. If the current input token is an operator, pop off all operators that have equal or higher precedence and append them to the output string; push the operator onto the stack. The order of popping is the order in the output.
5. If the current input token is '(', push it onto the stack
6. If the current input token is ')', pop off all operators and append them to the output string until a '(' is popped; discard the '('.
7. If the end of the input string is found, pop all operators and append them to the output string.

Postfix Expression Evaluation

Steps:

1. Scan the expression left to right
2. push values (constant) or variables (operands)
3. When an operator is found, computer the result by applying the operation to the preceding two operands (by popping two operands from the stack)
4. Push the result back to stack
5. When finished, print the stack

A formal Algorithm

An Algorithm to evaluate Postfix expressions

1. Read one input character at a time
2. switch (input)
   - case (Operand)
   - case (Number) : Push input into stack and then Go to step (1)
   - case (Operator) : op2 = pop (s)
     op1 = pop (s)
     result = computer (op1 Operator op2)
     Push result back into stack and Go to step (1)
   - case (new line character) : Pop from stack and print the answer
3. STOP

This algorithm doesn't handle errors in the input, although careful analysis of input and stack status could point to such error determination.

Stacks: Applications

- Parenthesis Matching
- HTML tag mapping
- Expression Evaluations
- Implementing backtracking
- Maximum Span
- procedure call, System Programming
- and others…

Stack: C implementation using Array

```
#define MAX 10
typedef struct stack {
        int arr[MAX];
        int top;
} STACK;
void createStack (STACK *);
int stackEmpty ( STACK *);
int stackFull ( STACK *);
void push (STACK *, int item);
int pop (STACK *);
```

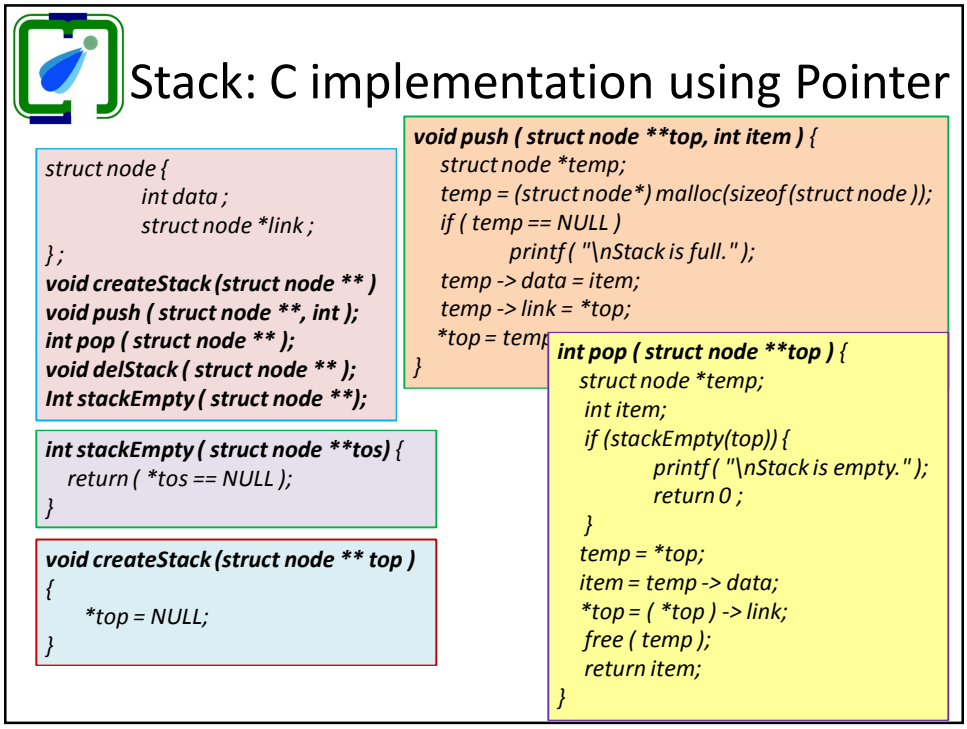
```
void createStack (STACK * s ) {
    s->top = -1;
}
```

```
void push ( STACK *s, int item ) {
   if ( stackFull(s)) {
       printf ( "\nStack is full." );
       return;
   }
   s->top++ ;
   s-> arr[s ->top] = item;
}
```

```
int stackFull ( STACK *s) {
   if ( s->top == MAX - 1 )
       return (1);
   else return (0);
}
```

```
int stackEmpty ( STACK *s) {
   if ( s->top == -1 )
       return (1);
   else return (0);
}
```

```
int pop( struct stack *s ) {
   int item;
   if ( stackEmpty(s)) {
      printf ( "\nStack is empty." );
      return (-1);
   }
   item = s ->arr[s -> top];
   s -> top--;
   return item;
}
```



Stack: C implementation using Pointer

```
struct node {
        int data ;
        struct node *link ;
} ;
void createStack (struct node ** )
void push ( struct node **, int );
int pop ( struct node ** );
void delStack ( struct node ** );
Int stackEmpty ( struct node **);
```

```
void push ( struct node **top, int item ) {
    struct node *temp;
    temp = (struct node*) malloc(sizeof (struct node ));
    if ( temp == NULL )
        printf ( "\nStack is full." );
    temp -> data = item;
    temp -> link = *top;
    *top = temp
}
```

```
int stackEmpty ( struct node **tos) {
    return ( *tos == NULL );
}
```

```
void createStack (struct node ** top )
{
    *top = NULL;
}
```

```
int pop ( struct node **top ) {
    struct node *temp;
    int item;
    if (stackEmpty(top)) {
        printf ( "\nStack is empty." );
        return 0 ;
    }
    temp = *top;
    item = temp -> data;
    *top = ( *top ) -> link;
    free ( temp );
    return item;
}
```