WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Cpt S 122 – Data Structures

Data Structures
Stacks

Nirmalya Roy

School of Electrical Engineering and Computer Science
Washington State University

Topics

- Stacks
  - `push, pop, printstack, isEmpty`
- Stacks Applications
  - Function calls, balancing symbols
  - Infix to postfix, postfix evaluation

Stacks

- A stack can be implemented as a constrained version of a linked list.
- New nodes can be added to a stack and removed from a stack *only* at the *top*.
- For this reason, a stack is referred to as a last-in, first-out (LIFO) data structure.
- A stack is referenced via a pointer to the top element of the stack.
- The link member in the last node of the stack is set to `NULL` to indicate the bottom of the stack.

Stacks (Cont.)

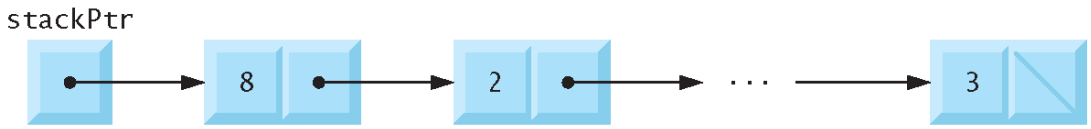


Fig. 12.7 | Stack graphical representation.

- A stack with several `nodes`
  - `stackPtr` points to the stack's top element.
- Stacks and linked lists are represented identically.
- The difference between stacks and linked lists is that insertions and deletions may occur *anywhere* in a linked list, but *only* at the *top* of a stack.

Stacks (Cont.)

- The primary functions used to manipulate a stack are `push` and `pop`.
- Function `push` creates a new node and places it on top of the stack.
- Function `pop` *removes* a node from the *top* of the stack, *frees* the memory that was allocated to the popped node and *returns the popped value*.

Stacks Example

```
1  // Fig. 12.8: fig12_08.c
2  // A simple stack program
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  // self-referential structure
7  struct stackNode {
8     int data; // define data as an int
9     struct stackNode *nextPtr; // stackNode pointer
10 }; // end structure stackNode
11
12 typedef struct stackNode StackNode; // synonym for struct stackNode
13 typedef StackNode *StackNodePtr; // synonym for StackNode*
14
15 // prototypes
16 void push( StackNodePtr *topPtr, int info );
17 int pop( StackNodePtr *topPtr );
18 int isEmpty( StackNodePtr topPtr );
19 void printStack( StackNodePtr currentPtr );
20 void instructions( void );
21
22 // function main begins program execution
23 int main( void )
24 {
```

Fig. 12.8 | A simple stack program. (Part 1 of 7.)

Stacks Example

```
25      StackNodePtr stackPtr = NULL; // points to stack top
26      unsigned int choice; // user's menu choice
27      int value; // int input by user
28
29      instructions(); // display the menu
30      printf( "%s", "? " );
31      scanf( "%u", &choice );
32
33      // while user does not enter 3
34      while ( choice != 3 ) {
35
36         switch ( choice ) {
37            // push value onto stack
38            case 1:
39               printf( "%s", "Enter an integer: " );
40               scanf( "%d", &value );
41               push( &stackPtr, value );
42               printStack( stackPtr );
43               break;
44            // pop value off stack
45            case 2:
46               // if stack is not empty
47               if ( !isEmpty( stackPtr ) ) {
48                  printf( "The popped value is %d.\n", pop( &stackPtr ) );
49               } // end if
```

Fig. 12.8 | A simple stack program. (Part 2 of 7.)

Stacks Example

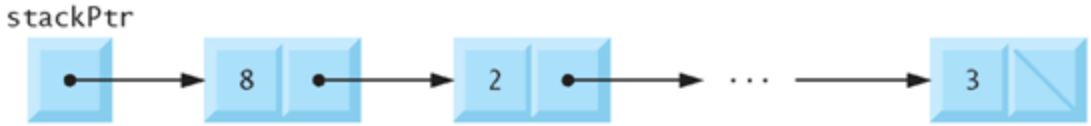
```
50
51             printStack( stackPtr );
52             break;
53          default:
54             puts( "Invalid choice.\n" );
55             instructions();
56             break;
57       } // end switch
58
59       printf( "%s", "? " );
60       scanf( "%u", &choice );
61    } // end while
62
63    puts( "End of run." );
64 } // end main
65
66 // display program instructions to user
67 void instructions( void )
68 {
69    puts( "Enter choice:\n"
70       "1 to push a value on the stack\n"
71       "2 to pop a value off the stack\n"
72       "3 to end program" );
73 } // end function instructions
74
```

Fig. 12.8 | A simple stack program. (Part 3 of 7.)

Function `push`

```
75  // insert a node at the stack top
76  void push( StackNodePtr *topPtr, int info )
77  {
78     StackNodePtr newPtr; // pointer to new node
79
80     newPtr = malloc( sizeof( StackNode ) );
81
82     // insert the node at stack top
83     if ( newPtr != NULL ) {
84        newPtr->data = info;
85        newPtr->nextPtr = *topPtr;
86        *topPtr = newPtr;
87     } // end if
88     else { // no space available
89        printf( "%d not inserted. No memory available.\n", info );
90     } // end else
91  } // end function push
92
```

Fig. 12.8 | A simple stack program. (Part 4 of 7.)



Function `push`

- Function `push` places a new node at the top of the stack.
- The function consists of three steps:
  - Create a new node by calling `malloc` and assign the location of the allocated memory to `newPtr`.
  - Assign to `newPtr->data` the value to be placed on the stack and assign `*topPtr` (the stack top pointer) to `newPtr->nextPtr`
    - the link member of `newPtr` now points to the previous top node.
  - Assign `newPtr` to `*topPtr`
    - `*topPtr` now points to the new stack top.

push operation

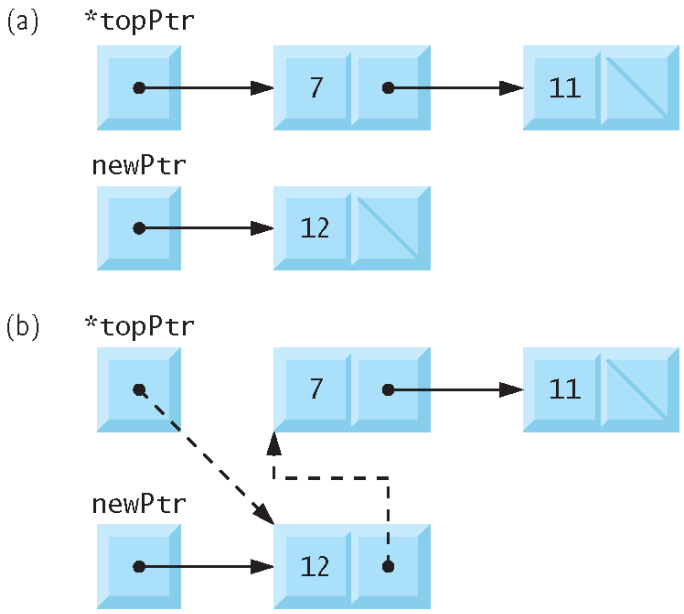


Fig. 12.10 | push operation.

Function pop

```
93  // remove a node from the stack top
94  int pop( StackNodePtr *topPtr )
95  {
96     StackNodePtr tempPtr; // temporary node pointer
97     int popValue; // node value
98
99     tempPtr = *topPtr;
100    popValue = ( *topPtr )->data;
101    *topPtr = ( *topPtr )->nextPtr;
102    free( tempPtr );
103    return popValue;
104 } // end function pop
105
```

Fig. 12.8 | A simple stack program. (Part 5 of 7.)

Function `pop`

- Function `pop` removes a node from the top of the stack.
- Function `main` determines if the stack is empty before calling `pop`.
- The `pop` operation consists of five steps:
  - Assign `*topPtr` to `tempPtr`; `tempPtr` will be used to free the unneeded memory.
  - Assign `(*topPtr)->data` to `popValue` to *save* the value in the top node.
  - Assign `(*topPtr)->nextPtr` to `*topPtr` so `*topPtr` contains *address of the new top node*.
  - *Free the memory* pointed to by `tempPtr`.
  - *Return* *`popValue`* to the caller.

pop operation

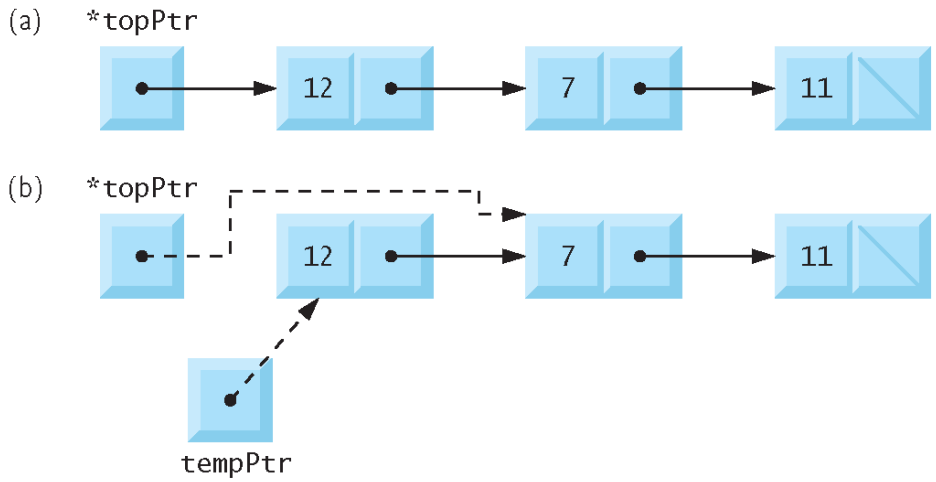


Fig. 12.11 | pop operation.

Function `printstack`

```
106 // print the stack
107 void printStack( StackNodePtr currentPtr )
108 {
109    // if stack is empty
110    if ( currentPtr == NULL ) {
111       puts( "The stack is empty.\n" );
112    } // end if
113    else {
114       puts( "The stack is:" );
115
116       // while not the end of the stack
117       while ( currentPtr != NULL ) {
118          printf( "%d --> ", currentPtr->data );
119          currentPtr = currentPtr->nextPtr;
120       } // end while
121
122       puts( "NULL\n" );
123    } // end else
124 } // end function printList
125
```

Fig. 12.8 | A simple stack program. (Part 6 of 7.)

Function `isEmpty`

```
126 // return 1 if the stack is empty, 0 otherwise
127 int isEmpty( StackNodePtr topPtr )
128 {
129    return topPtr == NULL;
130 } // end function isEmpty
```

Fig. 12.8 | A simple stack program. (Part 7 of 7.)

Output

```
Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 1
Enter an integer: 5
The stack is:
5 --> NULL

? 1
Enter an integer: 6
The stack is:
6 --> 5 --> NULL

? 1
Enter an integer: 4
The stack is:
4 --> 6 --> 5 --> NULL
```

Fig. 12.9 | Sample output from the program of Fig. 12.8. (Part 1 of 2.)

Output

```
? 2
The popped value is 4.
The stack is:
6 --> 5 --> NULL

? 2
The popped value is 6.
The stack is:
5 --> NULL

? 2
The popped value is 5.
The stack is empty.

? 2
The stack is empty.

? 4
Invalid choice.

Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 3
End of run.
```

Fig. 12.9 | Sample output from the program of Fig. 12.8. (Part 2 of 2.)

Applications of Stacks

- Stacks have many interesting applications.
- For example, whenever a *function call* is made, the called function must know how to *return* to its caller, so the *return address* is pushed onto a stack.
- If a series of function calls occurs, the successive return values are pushed onto the stack in *last-in, first-out order* so that each function can return to its caller.

Applications of Stacks (Cont.)

- Stacks support recursive function calls in the same manner as conventional nonrecursive calls.
- Stacks contain the space created for *automatic variables* on each invocation of a function.
- When the function returns to its caller, the space for that function's automatic variables is popped off the stack, and these variables no longer are known to the program.
- Stacks are used by compilers in the process of evaluating expressions and generating machine-language code.

Applications of Stacks (Cont.)

- Balancing symbols
  - Compiler checks for program syntax errors
  - Every right brace, bracket, and parenthesis must correspond to its left counterpart
  - The sequence [()] is legal, but [(]) is wrong
- Infix to Postfix Conversion
- Postfix Expressions Evaluations

Balancing Symbols

- Balancing symbols: ((()())(()))

```
stack<char> s;
while not end of file or input {
  read character c
  if (c == '(') then
    s.push(c)
  if (c == ')') then
    if (s.empty()) then
      error
    else
      s.pop();
}
if (!s.empty()) then
  error
else
  okay
```

- Make an empty stack
- Read characters until end of file
- If a character is an opening symbol, push it onto to the stack
- If it is a closing symbol, then if the stack is empty report an error, otherwise pop the stack
- If the symbol popped is not the corresponding opening symbol, then report an error
- At the EOF, the stack is not empty report an error

Infix to Postfix Conversion

- Infix to Postfix Conversion
  - Use a stack to convert an expression in standard form (infix) into postfix
  - Example: Infix expression: ((1 * 2) + 3) + (4 * 5)
  - Postfix expression: 1 2 * 3 + 4 5 * +

Infix to Postfix Conversion (Cont.)

- Steps:
  - When an operand is read, place it onto the output
  - Operators are not immediately output, so save them somewhere else which is stack
    - If a left parenthesis is encountered, stack it
  - Start with an initially empty stack
    - If we see a right parenthesis, pop the stack, writing symbols until we encounter a left parenthesis which is popped but not output
    - If we see any other symbols of higher priority inside stack, then we pop the entries from the stack until we find an entry of lower priority
      - When popping is done, we the push the operator onto the stack
  - Finally, if we read the end of input, pop the stack until it is empty, writing symbols onto the output

Infix to Postfix Conversion (Cont.)

- Convert the infix expression
  - Infix: a + b * c + (d * e + f) * g

  - Postfix: a b c * + d e * f + g * +

  - Note. We never remove a ‘(‘ from the stack except when processing a ‘)’

Evaluation of Postfix Expressions

- Evaluation of Postfix expressions
  - Infix expression: ((1 * 2) + 3) + (4 * 5)
  - Postfix expression: 1 2 * 3 + 4 5 * +
    - Unambiguous (no need for parenthesis)
    - Infix needs parenthesis or else implicit precedence specification to avoid ambiguity
      - E.g. a + b * c can be (a + b) * c or a + (b * c)
    - Postfix expression evaluation uses stack
      - E.g. Evaluate 1 2 * 3 + 4 5 * +
  - Rule of postfix expression evaluation
    - When a number/operand is seen push it onto the stack
    - When a operator is seen, the operator is applied to the two numbers (symbols) that are popped from the stack, and
    - Result is pushed onto the stack

Exercise: Infix-to-Postfix Converter

- Write a C program that converts an ordinary infix arithmetic expression (assume a valid expression is entered) with a single-digit integers such as

  (6 + 2) * 5 – 8 / 4

  to a postfix expression. The postfix version of preceding infix expression is

  6 2 + 5 * 8 4 / -

- Solve our textbook Deitel & Deitel Exercise 12.12 (Infix-to-Postfix converter) problem.

Exercise: Postfix Evaluation

- Write a C program that evaluates a postfix expression (assume it is valid) such as

  6 2 + 5 * 8 4 / -

- Solve our textbook Deitel & Deitel Exercise 12.13 (Postfix Evaluator) problem.