(3-3) Basics of a Stack

Instructor - Andrew S. O'Fallon
CptS 122 (January 26, 2024)
Washington State University

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

What is a Stack?

- A finite sequence of nodes, where only the top node may be accessed
- Insertions (PUSHes) may only be made at the top and deletions (POPs) may only be made at the top
  - A stack is referred to as a last-in, first-out (LIFO) data structure
  - Consider a pile or "stack" of plates; as you unload your dishwasher, the most recent plate is placed on top of the last plate, etc.; as you need a plate, you grab one from the top of the stack
- A stack is a restricted or constrained list
- We will focus most of our attention on linked list implementations of stacks

The Function-Call Stack (1)

- Refer to D & D Section 6.11
- We are aware of the function call stack; it is LIFO
- Also known as the *program-execution* stack, *run-time* stack, *program* stack, or simply "the *stack*"
- Works behind the scenes – supports the function call/return *mechanism – LIFO*
  - Necessary to track sequence of called functions

The Function-Call Stack (2)

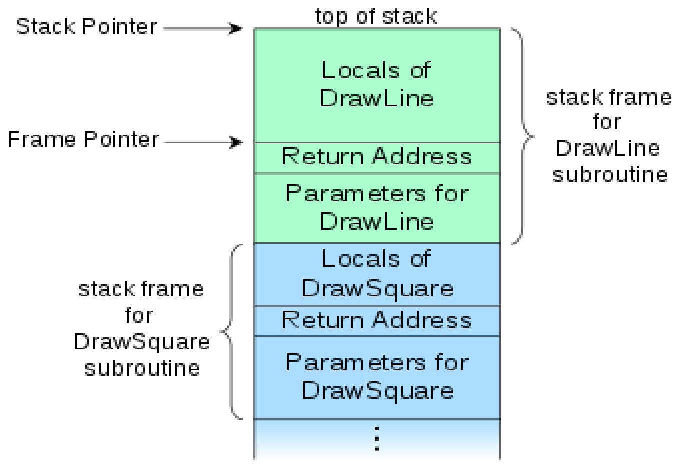
- Supports the *creation*, *maintenance*, and *destruction* of each called function's *local* variables
- Call stack memory is placed in RAM; monitored closely by CPU

The Function-Call Stack (3)

- When a function declares a variable, it is “pushed” onto the stack (dynamic memory is not though!)
- Parameters are also passed using the call stack

The Function-Call Stack (4)

- How to use the call stack when debugging in MS VS 2015: https://msdn.microsoft.com/en-us/library/a3694ts5.aspx
- Diagram of call stack - courtesy of https://en.wikipedia.org/wiki/Call_stack



Stack Frames (1)

- Each *called* function must eventually return control to the *calling* function

```
void function1(void) // calling function
{
        function2(); // called function
        // after executing function2(),
        // control returns back to function1()
}
```

- The system must track the *return address* that each called function needs to return control to the calling function – the *function-call* stack handles this info

Stack Frames (2)

- Each time a function *calls* another function, an entry is *pushed* to the stack
  - The entry is called the *stack frame* or *activation record*, which contains the return address required for the called function to return to the calling function
  - The entry also contains some other information discussed later

Stack Frames (3)

- If called function returns, instead of calling another function before returning, then the stack frame for the function call is *popped*, and control transfers to the *return address* in the stack frame
- The information required for the *called* function to return to its caller is always at the *top* of the call stack!

Stack Frames (4)

- If a called function makes a call to *another* function, then the *stack frame* for the new function is *pushed* to the top of the stack

Stack Frames and Local Variables (1)

- Local variables including parameters and variables declared by the function are reserved in the stack frame
  - The reason is these variables need to remain active if a function makes a call to another function and "go away" when the function *returns* to its caller

Stack Frames and Local Variables (2)

- Stack Overflow
  - If more function calls occur than can be handled by the finite amount of memory for the function call-stack, then an error called *stack overflow* occurs
  - There is high potential for this occurring with recursion, on problems that require a lot of recursive steps!

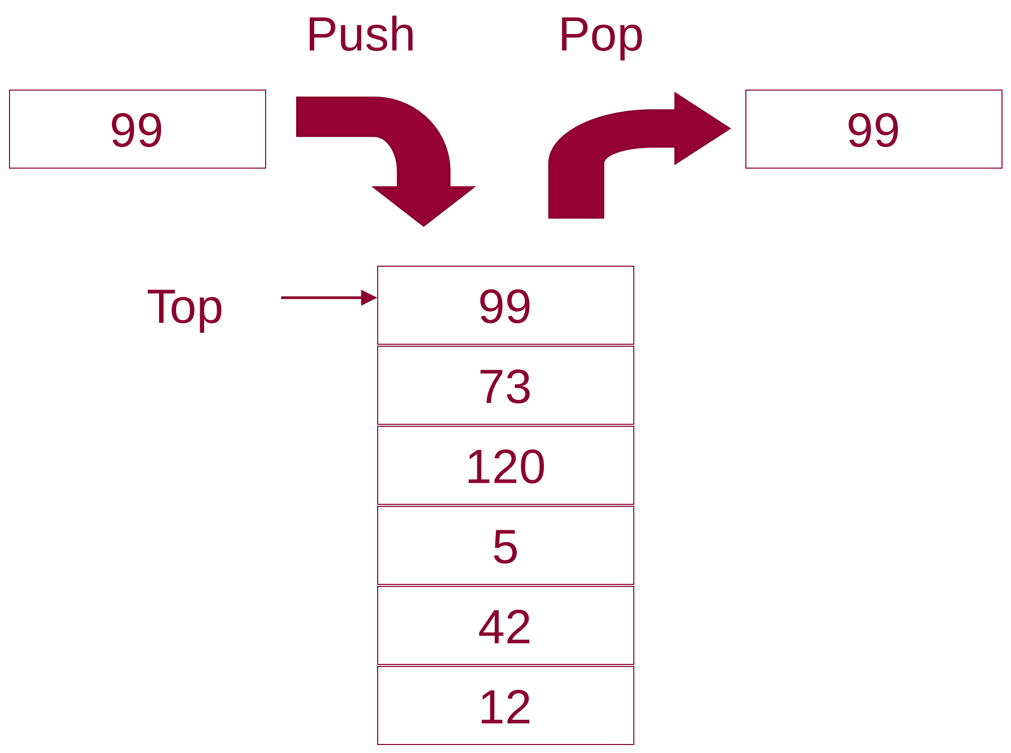
Video Explanation of Call Stack

- https://www.youtube.com/watch?v=Q2sFmqvpBe0

The Heap

- A region of memory that is not managed for you (unlike with the stack)
- We need to explicitly deallocate (free) the memory

Typical Representation of Stack of Integers



Struct StackNode

- For these examples, we'll use the following definition for `stackNode`:

```
typedef struct stackNode
{
    char data;
    // self-referential
    struct stackNode *pNext;
} StackNode;
```

Initializing a Stack (1)

- **InitStack (S)** Procedure to initialize the stack S to empty
- Our implementation:

```
void initStack (StackNode **pStack)
{
    // Recall: we must dereference a
    // pointer to retain changes
    *pStack = NULL;
}
```

Initializing a Stack (2)

- The `initStack()` function is elementary and is not always implemented
- We may instead initialize the pointer to the top of the stack with `NULL` within `main()`

```
int main (void)
{
    StackNode *pStack = NULL; // points to
                              // stack top
    …
}
```

Checking for Empty Stack (1)

- **StackIsEmpty (L) -> b:** Boolean function to return TRUE if S is empty
- Our implementation:

```
int isEmpty (StackNode *pStack)
{
    int status = 0; // False initially

    if (pStack == NULL) // The stack is empty
    {
        status = 1; // True
    }

    return status;
}
```

Checking for Empty Stack (2)

- Note: we could substitute the `int` return type with an enumerated type such as Boolean

```
typedef enum boolean
{
    FALSE, TRUE
} Boolean;
```

Checking for Empty Stack (3)

- Our implementation with `Boolean` defined:

```
Boolean isEmpty (StackNode *pStack)
{
    Boolean status = FALSE;

    if (pStack == NULL)
    {
        status = TRUE;
    }

    return status;
}
```

Printing Data in Stack (1)

- Our implementation:

```
void printStackIterative (StackNode *pStack)
{
    printf ("X -> ");
    while (!isEmpty (pStack))
    {
        printf ("%c -> ", pStack -> data);
        // Get to the next item
        pStack = pStack -> pNext;
    }
    printf ("NULL\n");
}
```

Printing Data in Stack (2)

- Another possible implementation using recursion:

```
void printStackRecursive (StackNode *pStack)
{
     if (!isEmpty (pStack)) // Recursive step
     {
             printf (“| %c |\n”, pStack -> data);
             printf (“   |  \n”); // Trying to imitate link
             printf (“   V  \n”);
             // Get to the next item
             pStack = pStack -> pNext;
             printStackRecursive (pStack);
     }
     else // Base case
     {
             printf (“NULL\n”);
     }
}
```

Inserting Data into a Stack

- **Push (S,e):** Procedure to insert a node with information e into S; in case S is empty, make a node containing e the only node in S and the current node
- Please consider these basic specifications for stack operations in the future; However, I will only show code from this point forward

Inserting Data onto Top of Stack w/o Error Checking (1)

- Our implementation:

```
void push (StackNode **pStack, char newData)
{
     StackNode *pMem = NULL;

     pMem = (StackNode *) malloc (sizeof (StackNode));
     // Initialize the dynamic memory
     pMem -> data = newData;
     pMem -> pNext = NULL;

     // Insert the new node onto top of stack
     pMem -> pNext = *pStack;
     *pStack = pMem;

}
```

- Does this look similar to insertAtFront () for a linked list? Yes!!!!!!

Inserting Data onto Top of Stack w/o Error Checking (2)

- Let’s define a new function which handles the dynamic allocation and initialization of a node:

```
StackNode * makeNode (char newData)
{
    StackNode *pMem = NULL;

    pMem = (StackNode *) malloc (sizeof (StackNode));
    // Initialize the dynamic memory
    pMem -> data = newData;
    pMem -> pNext = NULL;

    return pMem;
}
```

Inserting Data onto Top of Stack w/o Error Checking (3)

- Now we can reorganize our code and take advantage of the new function:

```
void push (StackNode **pStack, char newData)
{
    StackNode *pMem = NULL;

    pMem = makeNode (newData);

    // Insert the new node onto top of stack
    pMem -> pNext = *pStack;
    *pStack = pMem;
}
```

Inserting Data onto Top of Stack with Error Checking (1)

- Let’s modify our code so that we can check for dynamic memory allocation errors
- We’ll start with `makeNode():`

```
StackNode * makeNode (char newData)
{
     StackNode *pMem = NULL;

     pMem = (StackNode *) malloc (sizeof (StackNode));
     if (pMem != NULL)
     {
             // Initialize the dynamic memory
             pMem -> data = newData;
             pMem -> pNext = NULL;
     }
     // Otherwise no memory is available; could use else, but
     // it’s not necessary

     return pMem;
}
```

Inserting Data onto Top of Stack with Error Checking (2)

- Let's define a `Boolean` enumerated type as follows:

```
typedef enum boolean
{
      FALSE, TRUE
} Boolean; // To be used to indicate success of push ()
```

- Now let's add some error checking to `push():`

```
Boolean push (StackNode **pStack, char newData)
{
        StackNode *pMem = NULL;
        Boolean status = FALSE; // Assume can't insert a new node; out of memory

        pMem = makeNode (newData);

        if (pMem != NULL) // Memory was available
        {
                // Insert the new node onto top of stack
                pMem -> pNext = *pStack;
                *pStack = pMem;
                status = TRUE; // Successfully added a node to the stack!
        }

        return status;
}
```

Removing Data from Top of Stack (1)

- We will sometimes apply defensive design practices and ensure the stack is not empty; if we do not, then the precondition that must be satisfied is that the stack is not empty!
- This implementation of `pop()` checks for removal errors and doesn't return the data popped from the stack:

```
Boolean pop(StackNode **pStack)
{
      Boolean status = FALSE;
      StackNode *pTop = NULL;

      if (!isEmpty (*pStack)) // Stack is not empty; defensive design
      {
            pTop = *pStack; // Temp storage of top of stack
            *pStack = (*pStack)->pNext;
            free (pTop); // Remove the top node
            status = TRUE; // Successfully removed the top node
      }

      return status;
}
```

Removing Data from Top of Stack (2)

- This implementation of `pop()` returns the data removed from the top of the stack

```
char pop(StackNode **pStack)
{
      StackNode *pTop = NULL;
      character retData = '\0';

      if (!isEmpty (*pStack)) // Stack is not empty; defensive design
      {
            pTop = *pStack; // Temp storage of top of stack
            retData = (*pStack) -> data; // Keep data in top node
            *pStack = (*pStack) -> pNext;
            free (pTop); // Remove the top node
      }

      return retData;
}
```

Retrieving Data from Top of Stack w/o Deleting Nodes

- The `peek()` or `top()` function does not modify the stack; it just returns the data in the top of the stack (it "peeks" at the data)

```
char peek (StackNode *pStack)
{
     character retData = '\0';

     if (!isEmpty (pStack)) // Stack is not empty; defensive design
     {
          retData = pStack -> data;
     }

     return retData;
}
```

Stack Applications

- Reversing strings
- Checking for palindromes
- Searching for a path in a maze
- Tower of Hanoi
- Evaluating infix expressions
- Function call stacks
- Many others…

Closing Thoughts

- Can you build a driver program to test these functions?
- `push()` for a stack is essentially the same operation as `insertFront()` for a linked list…
- `pop()` is `deleteFront()` for a linked list
- If you know how to implement a linked list you should be able to implement a stack…
- You can implement a stack without using links; Hence, you can use an array as the underlying structure for the stack
- Continue to discuss why you would use a dynamic linked list instead of a dynamic linked stack and vice versa

Next Lecture…

- Queues

References

- P.J. Deitel & H.M. Deitel, *C: How to Program* (8th ed.), Prentice Hall, 2016
- J.R. Hanly & E.B. Koffman, *Problem Solving and Program Design in C (7th Ed.)*, Addison-Wesley, 2013

Collaborators

- Jack Hagemeister