

Stacks



The Stack ADT

- ◆ Set of objects in which the location an item is inserted and deleted is pre-specified
- ◆ Stacks
 - Insert in order
 - Delete most recent item inserted
 - LIFO - last in, first out

Stacks

2

The Stack ADT

- ◆ Examples of stacks
 - Cafeteria tray dispenser
 - Coin dispenser in your car
 - Balancing braces
 - Recognizing strings in a language
 - Evaluating postfix expressions
 - Converting infix to postfix
 - Undo sequence in a text editor
 - Saving local variables when one function calls another, and this one calls another, and so on.

Stacks

3

The Stack ADT

- ◆ Main stack operations:
 - **push**(object *o*): inserts element *o*
 - **pop**(): removes and returns the last inserted element
- ◆ Auxiliary stack operations:
 - **top**(): returns a reference to the last inserted element without removing it
 - **size**(): returns the number of elements stored
 - **isEmpty**(): returns true if the stack is empty, else false

Stacks

4

Exceptions

- ◆ Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception
- ◆ Exceptions are said to be “thrown” by an operation that cannot be executed
- ◆ In the **Stack** ADT, operations **pop** and **top** cannot be performed if the stack is empty
- ◆ Attempting the execution of **pop** or **top** on an empty stack throws an **EmptyStackException**

Stacks

5

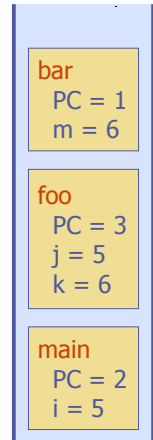
C++ Run-time Stack

- ◆ The C++ run-time system keeps track of the chain of active functions with a stack
- ◆ When a function is called, the run-time system pushes on the stack a frame containing
 - Local variables and return value
 - Program counter, keeping track of the statement being executed
- ◆ When a function returns, its frame is popped from the stack and control is passed to the method on top of the stack

```
main() {
    int i = 5;
    foo(i);
}

foo(int j) {
    int k;
    k = j+1;
    bar(k);
}

bar(int m) {
    ...
}
```



Stacks

6

Array-based Stack

- ◆ A simple way of implementing the Stack ADT uses an array
- ◆ We push (add) elements from left to right
- ◆ A variable keeps track of the index of the last item pushed

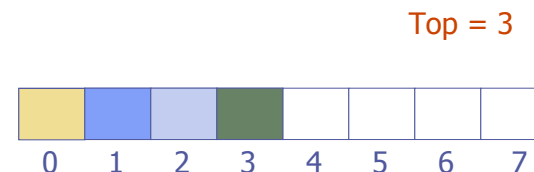


Stacks

7

Array-based Stack

- ◆ We pop (remove) elements from right to left



Stacks

8

Stack Data Structure

```
class Stack {  
private:  
    objectType stack[MAX_STACK_SIZE];  
    int top;  
public:  
    functions for stack manipulation  
    constructor sets top to -1  
};
```

Stacks

9

Stack Implementation- Push

- ◆ The array storing the stack elements may become full
- ◆ A push operation will then throw a `FullStackException`
 - Limitation of the array-based implementation

```
void push ( objectType o ) {  
    if ( top + 1 == MAX_STACK_SIZE )  
        throw FullStackException  
    else  
        S[top++] = o;
```

Stacks

10

Stack Implementation- Pop

- ◆ In class exercise - write pop function
 - Array may be empty when pop
 - A pop operation will then throw an `EmptyStackException`

Stacks

11

Performance and Limitations

- ◆ Performance
 - Let n be the number of elements in the stack
 - The space used is $O(n)$
 - Each operation runs in time $O(1)$
- ◆ Limitations
 - The maximum size of the stack must be defined *a priori*, and cannot be changed
 - Trying to push a new element into a full stack causes an implementation-specific exception

Stacks

12

Stack Application - Infix to Postfix Conversion

- ◆ Stack can be used to convert infix mathematical expressions to postfix mathematical expressions

Stacks

13

Stack Application - Infix to Postfix Conversion

- ◆ Algorithm
 - Process infix expression one item at a time
 - Operand - write to output
 - Operator - pop and write to output until an entry of lower priority is found (don't pop parentheses) then push
 - Left parentheses - push
 - Right parentheses - pop stack and write to output until left parentheses is found
 - When done processing expression, pop remaining items and write to output
 - NOTE - parentheses are not written to the output

Stacks

14

Stack Application - Infix to Postfix Conversion

$$a + b * c - (d * e + f) * g$$

Rule	Stack	Output
Operand - write to output		a
	+	a
	+	ab
	+*	ab
	+*	abc
	-	abc*+
	-(abc*+
	-(abc*+d
	-(*	abc*+d
.	-(*	abc*+de

Stacks

15

Stack Application - Infix to Postfix Conversion

$$a + b * c - (d * e + f) * g$$

Rule	Stack	Output
When done processing expression, pop remaining items and write to output	-(+	abc*+de*
	-(+	abc*+de*f
	-	abc*+de*f+
	-*	abc*+de*f+
	-*	abc*+de*f+g
.		abc*+de*f+g*-

Stacks

16

Stack Application - Evaluating Postfix Expressions

- ◆ You may assume I give you a valid postfix expression
- ◆ Algorithm
 - Process postfix expression one item at a time
 - Operand - push
 - Operator - pop 2 times, evaluate expression (second_pop operator first_pop), push result onto stack

Stacks

17

Stack Application - Evaluating Postfix Expressions

$6 * (5 + ((2 + 3) * 8) + 3) => 6 5 2 3 + 8 * + 3 + *$

Current Symbol	Stack
6	6
5	6 5
2	6 5 2
3	6 5 2 3
+	6 5 5

Stacks

18

Stack Application - Evaluating Postfix Expressions

$6 * (5 + ((2 + 3) * 8) + 3) => 6 5 2 3 + 8 * + 3 + *$

Current Symbol	Stack
8	6 5 5 8
*	6 5 40
+	6 45
3	6 45 3
+	6 48
*	288

Stacks

19

Other Stack Applications

- ◆ Balanced brace problem
 - Push every left brace
 - When you find a right brace, pop and compare. If no matching left brace then error
 - If stack doesn't end up empty then error
- ◆ Path problem
 - Take a path and return in the reverse order

Stacks

20

Growable Array-based Stack

- ◆ In a push operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one
- ◆ How large should the new array be?
 - incremental strategy: increase the size by a constant c
 - doubling strategy: double the size

```
Algorithm push(o)
if  $t = S.length - 1$  then
   $A \leftarrow$  new array of
    size ...
  for  $i \leftarrow 0$  to  $t$  do
     $A[i] \leftarrow S[i]$ 
   $S \leftarrow A$ 
 $t \leftarrow t + 1$ 
 $S[t] \leftarrow o$ 
```

Stacks

21

Linked List Based Stack

- ◆ Using a linked list can remove the size restrictions of an array
- ◆ Head will be referred to as the top
- ◆ Top initially points to NULL
- ◆ All operations are done at the top
 - Push = Insert at head/top
 - Pop = Remove from head/top

Stacks

22

Linked List Based Stack

```
bool isEmpty ( ) {
    if ( top == NULL )
        return true;
    else
        return false;
}

Node* getTop ( ) {
    return top;
}

void push ( Node* newTop ) {
    newTop->next = top;
    top = newTop;
}
```

Stacks

23

Linked List Based Stack

- ◆ In class exercise - Write the pop function
 - Think about memory leaks
 - ◆ Do you delete the node or not?
 - Use getTop () if you want to use the node
 - Use pop if you just want to remove the node
 - ◆ Delete the node, don't return a reference

Stacks

24

Linked List Based Stack

Stacks

- ◆ Often the array implementation is used since the stack usually never grows very large even when there is a large number of operations