Stacks

Saikrishna Arcot
M. Hudachek-Buswell

July 18, 2020

Stack

- A stack is a data structure where insertions and deletions follow the LIFO (last-in first-out) scheme. In other words, the last object you added onto the stack is the first object returned by the stack.

Stack

- A stack is a data structure where insertions and deletions follow the LIFO (last-in first-out) scheme. In other words, the last object you added onto the stack is the first object returned by the stack.
- Using a stack, you access only the object at the top of the stack; you cannot access, add to, or remove from any other point in the stack.

Stack

- A stack is a data structure where insertions and deletions follow the LIFO (last-in first-out) scheme. In other words, the last object you added onto the stack is the first object returned by the stack.
- Using a stack, you access only the object at the top of the stack; you cannot access, add to, or remove from any other point in the stack.
- Stacks are an abstraction, or general idea, of how the data structure functions, and they do not have specific implementations. Therefore, a stack is an Abstract Data Type (ADT) with more than one implementation.

Stack

- Main stack operations:

Stack

- Main stack operations:
  - `push(object)`: inserts an element

Stack

- Main stack operations:
  - `push(object)`: inserts an element
  - `object pop()`: removes and returns the last inserted element

Stack

- Main stack operations:
  - `push(object)`: inserts an element
  - `object pop()`: removes and returns the last inserted element
- Auxillary stack operations:

Stack

- Main stack operations:
  - `push(object)`: inserts an element
  - `object pop()`: removes and returns the last inserted element
- Auxillary stack operations:
  - `object top()`: returns the last inserted element without removing it

Stack

- Main stack operations:
  - `push(object)`: inserts an element
  - `object pop()`: removes and returns the last inserted element
- Auxillary stack operations:
  - `object top()`: returns the last inserted element without removing it
  - `integer size()`: returns the number of elements stored

Stack

- Main stack operations:
  - `push(object)`: inserts an element
  - `object pop()`: removes and returns the last inserted element
- Auxillary stack operations:
  - `object top()`: returns the last inserted element without removing it
  - `integer size()`: returns the number of elements stored
  - `boolean isEmpty()`: indicates whether no elements are stored

Example

Operation	Return value	Stack
`push(5)`	-	(5)
`push(3)`	-	(3, 5)
`size()`	2	(3, 5)
`pop()`	3	(5)
`isEmpty()`	`false`	(5)
`pop()`	5	()
`isEmpty()`	`true`	()
`pop()`	`null`	()
`push(7)`	-	(7)
`push(9)`	-	(9, 7)
`top()`	9	(9, 7)
`push(4)`	-	(4, 9, 7)
`size()`	3	(4, 9, 7)

Array-backed Stack

- A simple way of implementing a stack uses an array.

Array-backed Stack

- A simple way of implementing a stack uses an array.
- We add elements from left to right and remove elements from right to left.

Array-backed Stack

- A simple way of implementing a stack uses an array.
- We add elements from left to right and remove elements from right to left.
- A variable keeps track of the index of the top element. (Alternatively, a variable representing the size of the stack could also be used to determine where to add the next item.)

Array-backed Stack

- A simple way of implementing a stack uses an array.
- We add elements from left to right and remove elements from right to left.
- A variable keeps track of the index of the top element. (Alternatively, a variable representing the size of the stack could also be used to determine where to add the next item.)
- The array storing the stack elements may become full. A `push()` operation may then throw an implementation-defined exception or resize the backing array. This behavior is not defined or forced by the ADT.

Array-backed Stack

- Example of an array implementation for Stack ADT where the top of the stack is index 3, the size of the stack is 4, and if you "pop" the top of the stack data 2 will come off the stack.

1	3	3	2		
0	1	2	3	4	5

Array-backed Stack

```
**procedure** PUSH($o$)
    $arr[size] \leftarrow o$
    $size \leftarrow size + 1$
**end procedure**
**procedure** POP
    $size \leftarrow size - 1$
    $item \leftarrow arr[size]$
    $arr[size] \leftarrow \texttt{NULL}$
    **return** $item$
**end procedure**
```

Linked List-backed Stack

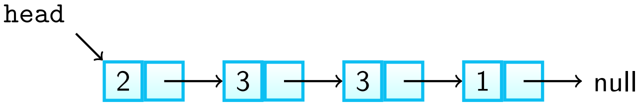
- A linked list could also be used as the backing data structure of a stack.

Linked List-backed Stack

- A linked list could also be used as the backing data structure of a stack.
- Elements would be added and removed to the same end of the list. For a stack, this usually means that the most efficient way to do this would be to add to and remove from the front of a linked list, as this would be $O(1)$ in most cases.

Linked List-backed Stack

- Example of a linked list implementation for Stack ADT where the top of the stack is at the head, the size of the stack is 4, and if you "pop" the top of the stack data 2 will come off the stack.



Performance and Limitations

- Performance

Performance and Limitations

- Performance
    - Let n be the number of elements in the stack.

Performance and Limitations

- Performance
  - Let n be the number of elements in the stack.
  - The space used is $O(n)$.

Performance and Limitations

- Performance
  - Let n be the number of elements in the stack.
  - The space used is $O(n)$.
  - Each of the operations, `push(obj)`, `pop()`, `isEmpty()`, `top()`, and `size()`, run in time $O(1)$.

Performance and Limitations

- Performance
  - Let n be the number of elements in the stack.
  - The space used is $O(n)$.
  - Each of the operations, `push(obj)`, `pop()`, `isEmpty()`, `top()`, and `size()`, run in time $O(1)$.
- Limitations

Performance and Limitations

- Performance
  - Let n be the number of elements in the stack.
  - The space used is $O(n)$.
  - Each of the operations, `push(obj)`, `pop()`, `isEmpty()`, `top()`, and `size()`, run in time $O(1)$.
- Limitations
  - For an array-backed stack, the initial maximum size of the stack must be defined *a priori*.

Performance and Limitations

- Performance
  - Let n be the number of elements in the stack.
  - The space used is $O(n)$.
  - Each of the operations, `push(obj)`, `pop()`, `isEmpty()`, `top()`, and `size()`, run in time $O(1)$.
- Limitations
  - For an array-backed stack, the initial maximum size of the stack must be defined *a priori*.
  - For an array-backed stack, if the backing array is to resized when it becomes full, this would then be an $O(n)$ operation (but the `push()` operation itself would be considered amortized $O(1)$).

Performance and Limitations

- Performance
  - Let n be the number of elements in the stack.
  - The space used is $O(n)$.
  - Each of the operations, `push(obj)`, `pop()`, `isEmpty()`, `top()`, and `size()`, run in time $O(1)$.
- Limitations
  - For an array-backed stack, the initial maximum size of the stack must be defined *a priori*.
  - For an array-backed stack, if the backing array is to resized when it becomes full, this would then be an $O(n)$ operation (but the `push()` operation itself would be considered amortized $O(1)$).
  - Linked list-backed stacks do not have either of the above two limitations.

Applications of Stacks

- Direct applications

Applications of Stacks

- Direct applications
    - Page-visited history in a Web browser

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in many programming languages

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in many programming languages
- Indirect applications

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in many programming languages
- Indirect applications
  - Auxillary data structure for algorithms

Applications of Stacks

- Direct applications
    - Page-visited history in a Web browser
    - Undo sequence in a text editor
    - Chain of method calls in many programming languages
- Indirect applications
    - Auxillary data structure for algorithms
    - Component of other data structures

Activation Stack

- Many programming languages (including Java) keep track of the chain of active methods with a stack.

Activation Stack

- Many programming languages (including Java) keep track of the chain of active methods with a stack.
- When a method is called, an **activation frame** is pushed onto the stack, containing:

Activation Stack

- Many programming languages (including Java) keep track of the chain of active methods with a stack.
- When a method is called, an **activation frame** is pushed onto the stack, containing:
    - Local variables and return value

Activation Stack

- Many programming languages (including Java) keep track of the chain of active methods with a stack.
- When a method is called, an **activation frame** is pushed onto the stack, containing:
    - Local variables and return value
    - Program counter, keeping track of the statement being executed

Activation Stack

- Many programming languages (including Java) keep track of the chain of active methods with a stack.
- When a method is called, an **activation frame** is pushed onto the stack, containing:
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a method ends, its frame is popped from the stack and control is passed to the method on top of the stack.

Activation Stack

- Many programming languages (including Java) keep track of the chain of active methods with a stack.
- When a method is called, an **activation frame** is pushed onto the stack, containing:
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a method ends, its frame is popped from the stack and control is passed to the method on top of the stack.
- This stack also allows for recursion to work correctly.

Activation Stack

- Many programming languages (including Java) keep track of the chain of active methods with a stack.
- When a method is called, an **activation frame** is pushed onto the stack, containing:
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a method ends, its frame is popped from the stack and control is passed to the method on top of the stack.
- This stack also allows for recursion to work correctly.
- This is known in general as the **activation stack**.

Parentheses Matching

- Each `(`, `{`, or `[` must be paired with a matching `)`, `}`, or `]`.

Parentheses Matching

- Each `(`, `{`, or `[` must be paired with a matching `)`, `}`, or `]`.
- This can be done with a stack.

Parentheses Matching

- Each `(`, `{`, or `[` must be paired with a matching `)`, `}`, or `]`.
- This can be done with a stack.
  - Correct: `( )(( )){([( )])}`

Parentheses Matching

- Each `(`, `{`, or `[` must be paired with a matching `)`, `}`, or `]`.
- This can be done with a stack.
  - Correct: `( )(( )){([( )])}`
  - Incorrect: `((( )(( )){([( )])}`

Parentheses Matching

- Each `(`, `{`, or `[` must be paired with a matching `)`, `}`, or `]`.
- This can be done with a stack.
  - Correct: `( )(( )){([( )])}`
  - Incorrect: `((( )(( )){([( )])}`
  - Incorrect: `)(( )){([( )])}`