CSC 212: Data Structures and Abstractions

07: Stacks

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Fall 2025

Stacks, queues, deques

- Fundamental data structures for collections (stack, queue, deque)
  - store and manage collections of elements with specific access patterns
  - data is manipulated in controlled, predictable order
  - used in various applications, including algorithm design, data processing, and system design
- Why using specialized data structures?
  - clear, restricted interfaces prevent misuse and express algorithmic purpose
  - enforced access patterns reduce programming mistakes
  - optimized $\Theta(1)$ operations vs. linear-time overhead in general containers
- Available in many programming languages and libraries
  - STL C++: `std::stack`, `std::queue`, and `std::deque`
  - Python: `collections.deque` (more efficient than lists)
  - Java: `java.util` provides `Stack` and `Queue` interfaces, as well as `ArrayDeque` and `LinkedList`

Stacks

Stacks

- Last-in-first-out
  - a **stack** is a linear data structure that follows the (LIFO) principle
  - the last element added to the stack is the first one to be removed
- Main operations
  - **push**: add element to the top
  - **pop**: remove element from the top
- Applications
  - expression evaluation, backtracking algorithms, undo mechanisms in applications, browser history navigation, etc.

Implementation

- Using arrays
  - ✓ `push` and `pop` at the end of the array (easier and efficient)
  - ✓ array can be either <u>fixed-length</u> or a <u>dynamic array</u>
- Considerations
  - ✓ <u>underflow</u>: throw an error when calling pop on an empty stack
  - ✓ <u>overflow</u>: throw an error when calling push on a full stack

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	3	-5	0	9	-2	7	1								

top

https://www.cs.usfca.edu/~galles/visualization/StackArray.html


```cpp
#pragma once
#include <cstddef>

// class implementing a Stack of integers
// fixed-length array (not a dynamic array)
class Stack {
    private:
        // array to store stack elements
        int *array;
        // maximum number of elements stack can hold
        size_t length;
        // current number of elements in stack
        size_t top;

    public:
        // IMPORTANT: need to add copy constructor and
        // overload assignment operator
        Stack(size_t);
        ~Stack();

        // pushes an element onto the stack
        void push(int);
        // returns and removes the top element from the stack
        int pop();
        // check if stack is empty
        bool empty() const { return top == 0; }
};
```



```cpp
#include "stack.h"
#include <stdexcept>

Stack::Stack(size_t len) {
    if (len < 1) {
        throw std::invalid_argument("Can't create an empty stack");
    }
    length = len;
    array = new int[length];
    top = 0;
}

Stack::~Stack() {
    delete [] array;
}

void Stack::push(int value) {
    if (top == length) {
        throw std::out_of_range("Stack is full");
    } else {
        array[top] = value;
        top ++;
    }
}

int Stack::pop() {
    if (top == 0) {
        throw std::out_of_range("Stack is empty");
    } else {
        top --;
        return array[top];
    }
}
```

```cpp
class Stack {
    private:
        int *array;
        size_t length;
        size_t top;

    public:
        Stack(size_t);
        ~Stack();

        void push(int);
        int pop();
        bool empty();
};
```


Practice

- What is the output of this code?

```cpp
#include <iostream>
#include "stack.h"

int main() {
    Stack s1(10), s2(10);

    s1.push(100);
    s2.push(s1.pop());
    s1.push(200);
    s1.push(300);
    s2.push(s1.pop());
    s2.push(s1.pop());

    s1.push(s2.pop());
    s1.push(s2.pop());

    while (!s1.empty()) {
        std::cout << s1.pop() << std::endl;
    }

    while (!s2.empty()) {
        std::cout << s2.pop() << std::endl;
    }

    return 0;
}
```

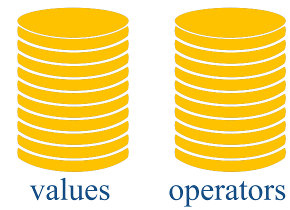
Example application

- Fully parenthesized infix expressions
  - infix expression: operators are placed between two operands
  - fully parenthesized: every operator and its operands are contained in parentheses
  - operator precedence and associativity don't matter
  - parentheses dictate exact computation order

$$((5 + ((10 - 4) * (3 + 2))) + 25)$$

Dijkstra's two-stack algorithm

- Create two stacks:
  - values (for operands) and operators (for operators)
- Process the expression from left to right, token by token:
  - if <u>left parenthesis</u>, ignore it
  - if <u>operand</u>, push it onto values stack
  - if <u>operator</u>, push it onto operators stack
  - if <u>right parenthesis</u>:
    - pop operator from operators stack
    - pop two elements from values stack
    - apply operator to those operands in the correct order
      - <result = second-popped operator first-popped>
    - push the <u>result</u> back onto values stack



Practice

- Trace the 2-stack algorithm with the following expression

$$((5 + ((10 - 4) * (3 + 2))) + 25)$$