CSC 212: Data Structures and Abstractions

06: Stacks

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Spring 2026

THINK BIG WE DO℠

Stacks, queues, and deques

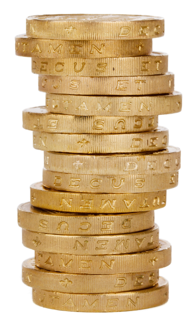
- Fundamental data structures for collections
  - store and manage collections of elements with specific **access patterns**
  - data is manipulated in controlled, predictable order
  - used in various applications, including algorithm design, data processing, and system design
- Why using specialized data structures?
  - clear, restricted interfaces prevent misuse and express algorithmic purpose
  - enforced access patterns reduce programming mistakes
  - core operations are $\Theta(1)$
    - constant-time operations compared to possible linear-time in general containers (e.g., vector insert at front)
- 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.

all core **stack** operations run in $\Theta(1)$ time



Implementation

- Using arrays
  - ✓ `push` and `pop` at the end of the array (easier and efficient)
  - ✓ array can be either fixed-length or a dynamic array
- Considerations
  - ✓ throw an error when calling pop on an empty stack
  - ✓ throw an error when calling push on a full stack
    - applicable to fixed-size array implementations

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

size

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

Stacks

- Consider a stack implemented by an efficient **dynamic array**
  - ✓ what is the cost of implementing push/pop at end?

Push	O(1) amortized
Pop	O(1)

  - ✓ what is the cost of implementing push/pop at front?

both operations require shifting elements

Push	O(n)
Pop	O(n)


```cpp
#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 capacity;
        // current number of elements in stack
        size_t size;

    public:
        // IMPORTANT: need to add copy constructor and
        // overload assignment operator
        // this class is NOT safe to copy as written
        Stack(size_t);
        ~Stack();

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



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

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

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

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

void Stack::pop() {
    if (size == 0) {
        throw std::out_of_range("Stack is empty");
    } else {
        size --;
    }
}

int& Stack::top() {
    if (size == 0) {
        throw std::out_of_range("Stack is empty");
    } else {
        return array[size - 1];
    }
}
```

Practice

- What is the output of this code?

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

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

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

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

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

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

    return 0;
}
```


Practice

- Design a $\Theta(n)$ algorithm using a stack to verify if the following string has <u>balanced brackets</u> or not
  - ✓ consider these as valid brackets: `()`, `{}`, `[]`

```
"int foo(int x) {
    return (x > 0 ? new int[x]{x}[0] : x * (2));
}"
```

- push opening brackets
- on closing bracket:
  - check stack not empty
  - check matching type
- at end: stack must be empty

Example application

- Fully parenthesized infix expressions
  - ✓ infix expression: operators are placed between two operands
  - ✓ fully parenthesized: every operator has <u>explicit</u> parentheses around its operands.
  - ✓ operator precedence and associativity don't matter
  - ✓ parentheses dictate exact computation order

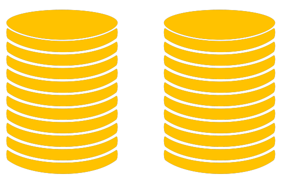
$$((5 + ((10 - 4) * (3 + 2))) + 25)$$

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

final result will be in the values stack

$$(5 + (10 - 4))$$



values operators

`algorithm runs in` $\Theta(n)$ `time`

Practice

- Trace the 2-stack algorithm with the following expression
  - ✓ assume operands are non-negative integers

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

Stack class (STL)

```
#include <iostream>
#include <stack>
using namespace std;

int main() {
    stack<int> s;

    s.push(10);
    s.push(20);
    s.push(30);

    cout << "Top: " << s.top() << endl;
    cout << "Size: " << s.size() << endl;

    s.pop();
    cout << "After pop, top: " << s.top() << endl;

    while (!s.empty()) {
        cout << s.top() << " ";
        s.pop();
    }

    return 0;
}
```


Solution (using STL class)

```
// simplified implementation of eval where we assume that operands are single-digit
// non-negative integers and the input expression is valid
int eval(const std::string& exp) {
    std::stack<int> operands;
    std::stack<char> operators;

    for (size_t i = 0 ; i < exp.length() ; ++i) {
        if (exp[i] == ' ' || exp[i] == '(') {
            continue;
        } else if (isdigit(exp[i])) {
            operands.push(exp[i] - '0');
        } else if (exp[i] == '+' || exp[i] == '-' || exp[i] == '*' || exp[i] == '/') {
            operators.push(exp[i]);
        } else if (exp[i] == ')') {
            int right = operands.top();
            operands.pop();
            int left = operands.top();
            operands.pop();
            char op = operators.top();
            operators.pop();
            switch (op) {
                case '+': operands.push(left + right); break;
                case '-': operands.push(left - right); break;
                case '*': operands.push(left * right); break;
                case '/': operands.push(left / right); break;
            }
        }
    }
    return operands.top();
}
```