CSC 212: Data Structures and Abstractions

06: Stacks, Queues, Deques

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

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

- 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 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

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)


```
#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; }
};
```



```
#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;
}
```


Example application

- Fully parenthesized infix expressions
  - ✓ infix expression: operators are placed between two operands
  - ✓ fully parenthesized: every operator has explicit 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 left parenthesis, ignore it
  - ✓ if operand, push it onto values stack
  - ✓ if operator, push it onto operators stack
  - ✓ if right parenthesis:
    - 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 result back onto values stack

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

algorithm runs in $\Theta(n)$ time

Practice

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

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

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();
}
```


Practice

- Design a $\Theta(n)$ algorithm using a stack to verify if the following string has balanced brackets 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

Queues

Queues

- First-in-first-out
  - ✓ a **queue** is a linear data structure that follows the (FIFO) principle
  - ✓ the first element added to the queue is the first one to be removed

`all core queue operations run in Θ(1) time (assuming proper implementations)`

- Main operations
  - ✓ **enqueue**: (**push**) add element to the end of the queue
  - ✓ **dequeue**: (**pop**) remove element from the front of the queue

- Applications
  - ✓ scheduling tasks in operating systems, managing requests in web servers, implementing breadth-first search (BFS) in graph algorithms, etc.

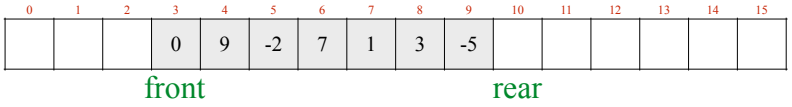
Implementation

- Using arrays
  - ✓ `enqueue` and `dequeue` at different ends of the array
  - ✓ array can be either fixed-length or a dynamic array
- Considerations
  - ✓ throw an error when calling dequeue on an empty queue
  - ✓ throw an error when calling enqueue on a full queue
    - applicable to fixed-size array implementations

Implementation

- Array-based (standard)
  - ✓ enqueue at the end — $\Theta(1)$ cost (amortized if using a dynamic array)
  - ✓ dequeue at front — $\Theta(n)$ cost
- Array-based (alternative)
  - ✓ enqueue at front — $\Theta(n)$ cost
  - ✓ dequeue at end — $\Theta(1)$ cost
- Circular array
  - ✓ enqueue at end — $\Theta(1)$ cost (amortized if using a dynamic array)
  - ✓ dequeue at front — $\Theta(1)$ cost
  - ✓ **efficient approach**, as it eliminates the need for shifting elements
    - (*) requires handling wrap-around at array boundaries

Circular array



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


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

// class implementing a Queue of integers
// fixed-length array (not a dynamic array)
class Queue {
    private:
        // array to store queue elements
        int *array;
        // index of the next available position in the queue
        size_t back_;
        // index of the first element in the queue
        size_t front_;
        // current number of elements in the queue
        size_t length;
        // maximum number of elements queue can hold
        size_t capacity;

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

        // adds an element to the end of the queue
        void push(int val);
        // removes the first element from the queue
        void pop();
        // check if queue is empty
        bool empty() const { return length == 0; }
        // returns the number of elements in the queue
        size_t size() const { return length; }
        // returns reference to the top element
        int& front();
        // returns reference to the back element
        int& back();
};
```

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

Queue::Queue(size_t cap) : back_(0), front_(0), length(0), capacity(cap) {
    if (cap < 1) {
        throw std::invalid_argument("Can't create an empty queue");
    }
    array = new int[capacity];
}

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

void Queue::push(int val) {
    if (length == capacity) {
        throw std::out_of_range("Queue is full");
    } else {
        array[back_] = val;
        back_ = (back_ + 1) % capacity;
        length ++;
    }
}

void Queue::pop() {
    if (length == 0) {
        throw std::out_of_range("Queue is empty");
    } else {
        front_ = (front_ + 1) % capacity;
        length --;
    }
}

int& Queue::front() {
    if (length == 0) {
        throw std::out_of_range("Queue is empty");
    } else {
        return array[front_];
    }
}

int& Queue::back() {
    if (length == 0) {
        throw std::out_of_range("Queue is empty");
    } else {
        return array[(back_ + capacity - 1) % capacity];
    }
}
```


Practice

- What is the output of this code?

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

int main() {
    Queue q1(50), q2(50);

    q1.push(-1);
    q2.push(-1);
    q1.push(100);
    q2.push(q1.front());
    q1.pop();
    q1.push(200);
    q1.push(300);
    q2.push(q1.front());
    q1.pop();
    q2.push(q1.front());
    q1.pop();

    q1.push(q2.front());
    q2.pop();
    q1.push(q2.front());
    q2.pop();

    while (!q1.empty()) {
        std::cout << q1.front() << " ";
        q1.pop();
    }
    std::cout << "\n---" << std::endl;
    while (!q2.empty()) {
        std::cout << q2.back() << " ";
        q2.pop();
    }
    std::cout << std::endl;

    return 0;
}
```


Practice

- Write a function that modifies a queue of elements by replacing every element with two copies of itself
  - ✓ e.g., [a, b, c] becomes [a, a, b, b, c, c]

Practice

- Write an algorithm to reverse the order of elements of a queue (hint: can use a separate stack)

- Write an algorithm that accepts a queue of elements and appends the queue's contents to itself in reverse order (hint: can use a separate stack)
  - ✓ for example: [a, b, c] becomes [a, b, c, c, b, a]

Practice

- Design an algorithm to:
  - load a number of audio files (songs)
  - play them in a continuous loop

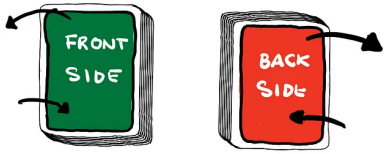
Deques

Deques

- Double-ended queue
  - a **deque** ("deck") allows insertion and removal of elements from both ends
  - combines the capabilities of stacks and queues

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

- Main operations
  - **push_front, push_back**
  - **pop_front, pop_back**
- Applications
  - task scheduling, undo/redo functionality, web browser history (forward/backward), sliding window problems, palindrome checking, etc.



Implementation

- Using arrays
  - array can be either fixed-length or a dynamic array
- Considerations
  - throw an error when calling "pop" on an empty deque
  - throw an error when calling "push" on a full deque
- Circular array
  - use a circular array to allow efficient operations at both ends
  - $\Theta(1)$ cost for all operations
    - `push_back` has amortized cost if using an efficient dynamic array

STL examples (standard template library)

```cpp
#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;
}
```

```cpp
#include <iostream>
#include <queue>
#include <string>

int main() {
    std::queue<std::string> q;

    q.push("Alice");
    q.push("Bob");
    q.push("Charlie");

    std::cout << "Front: " << q.front() << std::endl;
    std::cout << "Back: " << q.back() << std::endl;
    std::cout << "Size: " << q.size() << std::endl;

    q.pop();
    std::cout << "After pop, front: " << q.front() << std::endl;

    while (!q.empty()) {
        std::cout << q.front() << " ";
        q.pop();
    }

    return 0;
}
```

```cpp
#include <iostream>
#include <deque>
#include <utility>

int main() {
    std::deque<std::pair<std::string, int>> dq;

    dq.push_back({"Alice", 25});
    dq.push_back({"Bob", 30});
    dq.push_front({"Charlie", 22});

    std::cout << "Front: " << dq.front().first << ", " << dq.front().second << "\n";
    std::cout << "Back: " << dq.back().first << ", " << dq.back().second << "\n";
    std::cout << "Size: " << dq.size() << "\n";

    dq.pop_front();
    std::cout << "After pop_front: " << dq.front().first << "\n";

    while (!dq.empty()) {
        std::cout << dq.front().first << " ";
        dq.pop_front();
    }

    return 0;
}
```

OBS: random access is supported in std::deque, unlike in std::stack or std::queue