

CSC 212: Data Structures and Abstractions

08: Queues

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Fall 2025



Stacks

- Consider a stack implemented by an efficient dynamic array
 - what is the computational cost of insertion and deletion at end?

Push	$O(1)$ amortized
Pop	$O(1)$

- what is the computational cost of insertion and deletion at the beginning?

both operations require
shifting elements

Push	$O(n)$
Pop	$O(n)$

2

Practice

- Design an algorithm using a single stack to verify if the following string has balanced brackets or not
 - consider the following characters: `()`, `{}`, `[]`

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

3

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

Main operations

- ✓ **enqueue**: add element to the end of the queue
- ✓ **dequeue**: 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.



5

Implementation

Using arrays

- ✓ **enqueue** and **dequeue** at different ends of the array
- ✓ array can be either fixed-length or a dynamic array

Considerations

- ✓ underflow: throw an error when calling dequeue on an empty queue
- ✓ overflow: throw an error when calling enqueue on a full queue

6

Implementation

Array-based (standard)

- ✓ enqueue at the end — $\Theta(1)$ cost (amortized if using a dynamic array)
- ✓ dequeue from the beginning — $\Theta(n)$ cost

Array-based (alternative)

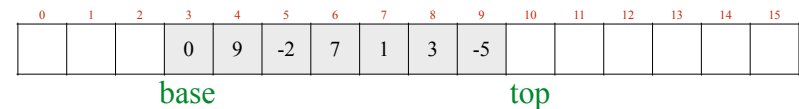
- ✓ enqueue at the beginning — $\Theta(n)$ cost
- ✓ dequeue from the end — $\Theta(1)$ cost

Circular array

- ✓ enqueue at the end — $\Theta(1)$ cost (amortized if using a dynamic array)
- ✓ dequeue from the beginning — $\Theta(1)$ cost
- ✓ more efficient approach, as it eliminates the need for shifting elements
 - (*) requires handling wrap-around at array boundaries

7

Circular array



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

8

```

#pragma once
#include <cstdint>

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

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

    // adds an element to the end of the queue
    void enqueue(int val);
    // removes the first element from the queue
    int dequeue();
    // 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; }
};

```

9

```

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

Queue::Queue(size_t cap) : top(0), base(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::enqueue(int val) {
    if (length == capacity) {
        throw std::out_of_range("Queue is full");
    } else {
        array[top] = val;
        top = (top + 1) % capacity;
        length ++;
    }
}

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

```

10

Practice

- What is the output of this code?

```

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

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

    q1.enqueue(100);
    q2.enqueue(q1.dequeue());
    q1.enqueue(200);
    q1.enqueue(300);
    q2.enqueue(q1.dequeue());
    q2.enqueue(q1.dequeue());

    q1.enqueue(q2.dequeue());
    q1.enqueue(q2.dequeue());

    while (!q1.empty()) {
        std::cout << q1.dequeue() << std::endl;
    }
    std::cout << "___" << std::endl;
    while (!q2.empty()) {
        std::cout << q2.dequeue() << std::endl;
    }

    return 0;
}

```

11

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]

12

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]

13

Practice

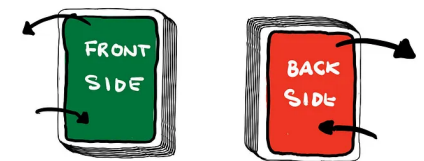
- Design an algorithm to:
 - ✓ load a number of audio files (songs)
 - ✓ play them in a continuous loop

14

Dequeues

Dequeues

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



16

Implementation

- Using arrays
 - ✓ array can be either fixed-length or a dynamic array
- Considerations
 - ✓ underflow: throw an error when calling “remove” on an empty queue
 - ✓ overflow: throw an error when calling “insert” on a full queue
- 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

17

STL (standard template library)

Code duplication

- How to modify the code below to avoid duplication?

```
#include <iostream>

int add_int(int a, int b) {
    return a + b;
}

double add_double(double a, double b) {
    return a + b;
}

int main() {
    std::cout << "Sum (int): " << add_int(5, 3) << "\n";
    std::cout << "Sum (double): " << add_double(2.5, 1.7) << "\n";

    return 0;
}
```

19

Function templates

```
#include <iostream>

template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    int sumI = add<int>(5, 3);
    double sumD = add<double>(2.5, 1.7);

    std::cout << "Sum (int): " << sumI << "\n";
    std::cout << "Sum (double): " << sumD << "\n";

    return 0;
}
```

Template functions/classes allow writing **generic code** that can work with different data types without the need to write separate code for each type. The compiler generates the appropriate instantiation based on the data type specified to the function/class.

20

Class templates

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


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

    void push(int val);
    void pop();
    int top();
};

template<typename T>
class Stack {
private:
    T *array;
    size_t capacity;
    size_t top;

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

    void push(T val);
    void pop();
    T top();
};
```



21

```
#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;
}
```

```
#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;
}
```

```
#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;
}
```