CSC 212: Data Structures and Abstractions

07: Queues and Deques

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Spring 2026

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**: (**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.

all core **queue** operations run in $\Theta(1)$ time (assuming proper implementations)

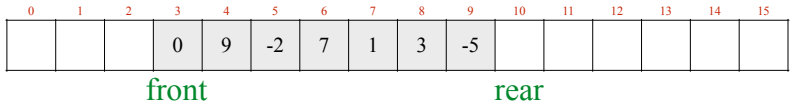
Implementation

- 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?

```
#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

Queue class (STL)

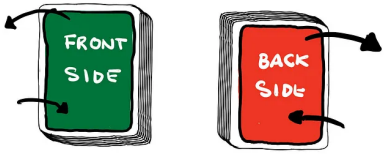
```cpp
#include <iostream>
#include <queue>
#include <string>

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;
}
```


Deques

Deques

- 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.

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



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

Deque class (STL)

```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