MTH 4300/4299: Programming and Computer Science II

Lecture 18: Stacks and Queues

Jaime Abbariao

▓▓ Stack Recap

A stack is a last-in, first-out (LIFO) container. Think of a pile of plates.

Operation	What it does
`push(x)`	Add `x` to the top
`pop()`	Remove the top element
`top()`	Look at the top element

The standard library already ships this container — we'll go straight to `std::stack`.

▓▓ std::stack

The standard library already provides a stack in `<stack>`. It's a container adaptor — it wraps `std::deque` by default and exposes only the stack operations.

```cpp
#include <stack>

std::stack<int> s;
s.push(1);
s.push(2);
std::cout << s.top() << std::endl;  // 2
s.pop();
std::cout << s.size() << std::endl; // 1
```

Same `push`, `pop`, `top`, `empty`, `size` — already written, tested, optimized. We'll use it for the problems next.

▓▓ Classic Stack Problems

A stack lets us defer work — remember something now, handle it when the matching piece arrives.

1. Balanced brackets — is `({[]})` well-formed?
2. Expression evaluation — compute `3 + 4 * 2` with precedence.
3. Next greater element — for each element, find the next larger value on its right.

▒▒▒ Problem 1: Balanced Brackets — Idea

Goal: given a string like `({[]})`, decide whether every opener has a matching closer in the right order.

Algorithm:

- Opener (`(`, `[`, `{`) → push it.
- Closer → stack top must be the matching opener; pop.
- Empty stack on a closer, or mismatch → fail.
- At the end, the stack must be empty.

```cpp
bool is_balanced(std::string s) {
  std::stack<char> stk;
  for (int i = 0; i < s.size(); i++) {
    char c = s[i];
    if (c == '(' || c == '[' || c == '{') {
      stk.push(c);
    } else if (c == ')' || c == ']' || c == '}') {
      if (stk.empty()) return false;
      char top = stk.top();
      stk.pop();
      if (c == ')' && top != '(') return false;
      if (c == ']' && top != '[') return false;
      if (c == '}' && top != '{') return false;
    }
  }
  return stk.empty();
}
```

▓▓ Problem 1: Balanced Brackets — Demo

```cpp
int main() {
  std::cout << is_balanced("({[]})") << std::endl;  // 1
  std::cout << is_balanced("({[})")  << std::endl;  // 0
  std::cout << is_balanced("(())")  << std::endl;  // 0
  return 0;
}
```

The stack remembers which opener is still waiting — and the most recent one must be matched first. That's LIFO.

▒▒▒ Problem 2: Evaluating Postfix — What Is Postfix?

Goal: evaluate an expression like `3 4 2 * +` → `11`.

Postfix (a.k.a. reverse Polish notation) puts each operator after its two operands:

- Infix: `3 + 4 * 2`
- Postfix: `3 4 2 * +`

No parentheses, no precedence rules — the order of operations is baked into the token order. A single stack walk evaluates it.

▓▓▓ Problem 2: Evaluating Postfix — Rules

Walk the postfix tokens with an operand stack:

- Number → push it.
- Operator → pop two operands, apply, push the result.

```
tokens: 3 4 2 * +
stack:  [3]
        [3, 4]
        [3, 4, 2]
        [3, 8]      (4 * 2)
        [11]        (3 + 8)
```

```cpp
bool is_number(std::string t) {
  return t != "+" && t != "-" && t != "*" && t != "/";
}

int apply(char op, int a, int b) {
  if (op == '+') return a + b;
  if (op == '-') return a - b;
  if (op == '*') return a * b;
  return a / b;
}
```

▓▓ Problem 2: Evaluating Postfix — Code

```
int eval_postfix(std::vector<std::string> tokens) {
  std::stack<int> stk;
  for (int i = 0; i < tokens.size(); i++) {
    std::string t = tokens[i];
    if (is_number(t)) {
      stk.push(std::stoi(t));
    } else {
      int b = stk.top(); stk.pop();
      int a = stk.top(); stk.pop();
      stk.push(apply(t[0], a, b));
    }
  }
  return stk.top();
}
```

Watch the order: first pop is `b` (right operand), second is `a`. Swapping breaks `-` and `/`.

▓▓ Problem 3: Next Greater Element — Idea

Goal: for each `a[i]`, find the nearest `j > i` with `a[j] > a[i]`. Return `-1` if none.

Brute force: O(n²). A monotonic stack does it in O(n).

Idea: keep a stack of indices whose answer isn't yet known, with values in decreasing order. When a new element appears, it's the answer for every smaller index on top of the stack.

▓▓ Problem 3: Next Greater Element — Code

```cpp
std::vector<int> next_greater(std::vector<int> a) {
  int n = a.size();
  std::vector<int> result(n, -1);
  std::stack<int> stk;  // indices, values decreasing
  for (int i = 0; i < n; i++) {
    while (!stk.empty() && a[stk.top()] < a[i]) {
      result[stk.top()] = a[i];
      stk.pop();
    }
    stk.push(i);
  }
  return result;
}
```

Each index is pushed and popped at most once → O(n).

▓▓ Problem 3: Next Greater Element — Trace

Input: `[2, 1, 3]`

```
i=0, a=2: stack empty → push 0.        stk=[0]
i=1, a=1: 2 > 1 → push 1.               stk=[0, 1]
i=2, a=3: pop 1, result[1]=3
          pop 0, result[0]=3
          push 2.                       stk=[2]

result = [3, 3, -1]
```

▓▓ Queue Concept

A queue is first-in, first-out (FIFO). Like a line at a store.

Operation	What it does
`enqueue(x)`	Add `x` to the back
`dequeue()`	Remove from the front
`front()`	Peek at the front

▓▓ Stack vs. Queue

	Stack	Queue
Order	LIFO	FIFO
Add	push (top)	enqueue (back)
Remove	pop (top)	dequeue (front)
Analogy	pile of plates	line at a store

std::queue

Standard-library queue in `<queue>` — another container adaptor over `std::deque`.

```cpp
#include <queue>

std::queue<int> q;
q.push(10);
q.push(20);
std::cout << q.front() << std::endl; // 10
std::cout << q.back()  << std::endl; // 20
q.pop();
std::cout << q.front() << std::endl; // 20
```

Naming: `push` adds to the back, `pop` removes from the front.

▒▒▒ std::deque — Sneak Peek

For the sliding-window problem coming up we'll want a double-ended queue.

`std::deque` supports all four in O(1):

- `push_front` / `push_back`
- `pop_front` / `pop_back`
- `front()` / `back()`

A deque is a superset of a queue — anything a queue can do, a deque can do.

Classic Queue Problems

Three problems where FIFO order is exactly what you want:

1. Task / job scheduling — handle requests in arrival order.
2. Producer–consumer buffering — decouple a fast producer from a slow consumer.
3. Sliding window maximum — max of every window of size `k`, in O(n).

The first two are systems patterns. The third is a classic algorithm — we'll code it.

▓▓ Queue Problem: Task Scheduling

A printer handles one page at a time. New jobs go to the back; the printer pulls from the front.

Same shape: CPU task queues, web-server request buffers, ticketing systems. The queue makes scheduling fair — first come, first served. A stack would starve old jobs.

▓▓ Queue Problem: Producer–Consumer

One side (the producer) generates items; another side (the consumer) handles them — at different speeds.

```
producer  →  [ item ] [ item ] [ item ]  →  consumer
              enqueue                        dequeue
```

Examples: keyboard input buffers, network packets, video streaming, message queues between services.

If the consumer falls behind the queue grows; if the producer pauses the consumer drains it. Neither side has to wait on the other directly.

▒▒▒ Queue Problem: Sliding Window Max — Goal

Goal: given array `a` and window size `k`, output the max of every window of `k` consecutive elements.

Brute force: O(nk). We can do it in O(n) using a deque.

Trick: keep a deque of indices with values in decreasing order. The front is always the index of the current window's max.

▓▓ Sliding Window Max — Invariants

For every new index `i`:

- Drop stale fronts: pop from the front while `dq.front() <= i - k` (fell out of the window).
- Drop dominated backs: pop from the back while `a[dq.back()] <= a[i]` — they can never be the max again.
- Push `i` at the back.
- Once `i >= k - 1`, record `a[dq.front()]` as the window's max.

▓▓ Sliding Window Max — Code

```cpp
std::vector<int> sliding_max(std::vector<int> a, int k) {
  std::deque<int> dq;
  std::vector<int> result;
  for (int i = 0; i < a.size(); i++) {
    while (!dq.empty() && dq.front() <= i - k) dq.pop_front();
    while (!dq.empty() && a[dq.back()] <= a[i]) dq.pop_back();
    dq.push_back(i);
    if (i >= k - 1) result.push_back(a[dq.front()]);
  }
  return result;
}
```

Each index enters and leaves the deque at most once → O(n) total.

▓▓ Sliding Window Max — Trace

Input: `a = [1, 3, 2, 5, 4]`, `k = 3`

```
i=0 (1): dq=[0]
i=1 (3): pop 0 (1<=3), push 1. dq=[1]
i=2 (2): push 2. dq=[1,2]       window [1,3,2] → 3
i=3 (5): pop 2, pop 1, push 3. dq=[3]
                                 window [3,2,5] → 5
i=4 (4): push 4. dq=[3,4]       window [2,5,4] → 5
```

Result: `[3, 5, 5]`.

Summary

- Stack = LIFO; Queue = FIFO.
- Use the STL: `std::stack`, `std::queue`, `std::deque` — all operations O(1).
- Stack problems: balanced brackets, postfix evaluation, next greater element (monotonic stack).
- Queue problems: scheduling, producer–consumer, sliding window max (deque).

Next lecture: trees and binary search trees.

▓▓ Lab: Queue with Two Stacks — Setup

Implement a queue using only two `std::stack` objects.

```cpp
#include <iostream>
#include <stack>

class TwoStackQueue {
public:
  void enqueue(int value);
  int dequeue();
  int front();
  bool empty();
private:
  std::stack<int> in_;   // new elements go here
  std::stack<int> out_;  // elements come out from here
};
```

Key insight: popping a whole stack into another stack reverses the order. LIFO + LIFO = FIFO.

Lab: Queue with Two Stacks — Method Stubs

```
void enqueue(int value) {
  // TODO
}

int dequeue() {
  // TODO
}

int front() {
  // TODO
}

bool empty() {
  // TODO
}
```

Hints:

- `enqueue` always pushes onto `in_`.
- `dequeue` / `front`: if `out_` is empty, transfer all of `in_` into `out_`, then use `out_`.
- Only transfer when `out_` is empty — otherwise order gets mixed up.

```cpp
int main() {
  TwoStackQueue q;
  q.enqueue(1);
  q.enqueue(2);
  q.enqueue(3);

  std::cout << q.front() << std::endl;
  std::cout << q.dequeue() << std::endl;

  q.enqueue(4);
  q.enqueue(5);

  while (!q.empty()) {
    std::cout << q.dequeue() << std::endl;
  }
}
```

Lab: Expected Output

```
1
1
2
3
4
5
```

`front()` shows `1` (first enqueued). Then `dequeue()` returns `1` and removes it. After enqueueing `4` and `5`, draining the queue gives `2, 3, 4, 5` in order.