CSE 20312
Stacks, Queues, Deques
Stacks
**Stacks: ADT**

**push(value)**
Insert an element at the top

**pop()**
Delete the element at the top

**top()**
Access element at the top

**empty()**
Whether or not there are any elements

Elements are added (**pushed**) and removed (**popped**) in last-in, first-out (**LIFO**) order.
Stacks: Use Cases

Call stack

Language parsing

Backtracking
Stacks: Interview Question

Problem 9.1: Design a stack that supports a max operation, which returns the maximum value stored in the stack:

All operations must be $O(1)$ time

You can use $O(n)$ space, in addition to what is required for the elements themselves

Use std::stack to implement solution called max_stack
template <typename T>
class max_stack {
    std::stack<T> data;
    std::stack<T> max_data;

public:
    bool empty() const { return data.empty(); }
    const T& top() const { return data.top(); }
    void push(const T& value) {
        data.push(value);
        if (max_data.empty() || value > max_data.top()) {
            max_data.push(value);
        } else {
            max_data.push(max_data.top());
        }
    }
    void pop() {
        data.pop();
        max_data.pop();
    }
    const T& max() const { return max_data.top(); }
};
**Stacks: Implementation**

**Array**

- Top of stack is back of array
- Array must be dynamic to prevent overflow

```
0 1 2 3 4
```

**Linked List**

- Top of stack is front of list

```
0 → 1 → 2 → 3 → 4
```

Top
template <typename T>
class vector_stack {
    std::vector<T> data;

public:
    bool empty() const { return data.empty(); }
    const T& top() const { return data.back(); }
    void push(const T& value) {
        data.push_back(value);
    }
    void pop() {
        data.pop_back();
    }
};
Queues
Queues: ADT

**push(value)**
Insert an element at back

**pop()**
Delete the element at the front

**front()**
Access element at the front

**back()**
Access element at the back

Elements are added (pushed) and removed (popped) in first-in, first-out (FIFO) order.
Queues: Use Cases

- Communication
- Scheduling
- Buffering

A UNIX pipe provides one-way communication between two processes on the same computer.
Queues: Interview Question

Problem 9.12: How would you implement a queue given two stacks and $O(1)$ additional storage?

Your implementation should be efficient: the time to do a sequence of $m$ combined enqueues and dequeues should be $O(m)$
template <typename T>
class stack_queue {
    std::stack<T> in, out;
    void in_to_out() {
        while (!in.empty()) {
            out.push(in.top());
            in.pop();
        }
    }

public:
    bool empty() const { return in.empty() && out.empty(); }
    const T& front() {
        if (out.empty()) in_to_out();
        return out.top();
    }
    const T& back() const { return in.top(); }
    void push(const T& value) {
        in.push(value);
    }
    void pop() {
        if (out.empty()) in_to_out();
        out.pop();
    }
};
Queues: Implementation

**Array**

Can’t efficiently push/pop at the front

Need to keep track of both front/back index

**Alternative:** start in the middle until we hit end and then resize or shift

**Linked List**

Can’t efficiently push or pop at the back with singly linked list

Need a **tail pointer** to push at back of list

Need **doubly-linked** list to pop efficiently at the back of the list
template <typename T>
class list_queue {
protected:
    struct Node { Node *next; Node *prev; T data; };  
    Node head;
    Node tail;
public:
    list_queue() : head({&tail, &tail}), tail({&head, &head}) {}  
    bool empty() const { return head.next == &tail; }  
    const T & front() { return head.next->data; };  
    const T & back() { return tail.prev->data; };  
    void push(const T &data) {  
        Node *node = new Node({&tail, tail.prev, data});  
        tail.prev->next = node;  
        tail.prev = node;  
    }
    void pop() {  
        Node *old_front = head.next;  
        Node *new_front = head.next->next;  
        new_front->prev = &head;  
        head.next = new_front;  
        delete old_front;  
    }
};
Deques
Deques: ADT

**push**_{front, back}**(value)**
Insert an element at front/back

**pop**_{front, back}**(value)**
Delete the element at the front/back

**front**(value)
Access element at the front

**back**(value)
Access element at the back

**empty**(value)
Whether or not queue has any elements
Deques: Implementation

In the STL, a deque is neither a dynamic array or a linked list:

Linked list has each element in a different block of memory, which is bad for locality and wastes space.

Dynamic array invalidates pointers when resizing and is difficult to get large blocks of memory.