Chapter 9
Stacks and Queues

9.1 Introduction to Stacks

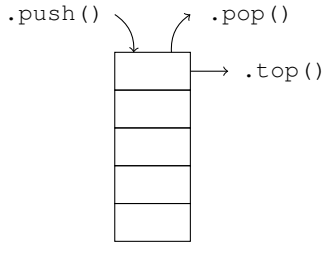
A **stack** is a one-ended linear data structure that supports last-in, first-out (LIFO) data access. When you retrieve elements out of a stack, you remove the elements in the opposite order in which they were inserted.

For a real-life example of how a stack works, consider a stack of books on a table. When you add a book to the stack, you add it to the very top of the pile. When you retrieve a book from the stack, you must remove the book on top — the one you most recently added. You cannot access the book at the very bottom of the stack without removing all the other books on top of it.

A stack works in a similar manner. When you call `.push()`, you add an element to the top of the stack. When you call `.top()`, you retrieve the most recently added element out of the stack. When you call `.pop()`, you remove the most recently added element from the stack.

The interface of a stack:

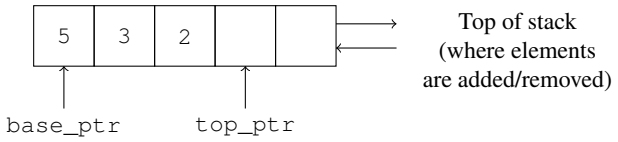
Function	Behavior
`.push(val)`	Adds `val` to the top of the stack
`.pop()`	Removes top element from the stack
`.top()`	Returns a reference to the top element in the stack
`.size()`	Returns the number of elements in the stack
`.empty()`	Returns whether the stack is empty



Stacks have numerous practical use cases in programming. For instance, text editors use a stack's behavior to implement "undo" buttons that allow you to revert changes. Compilers also use stacks to check for matching parentheses in code (e.g., making sure every `'{'` has a matching `'}'` in the right place). Stacks also play an important role in recursion (by keeping track of the data of previous function calls) and graph-searching algorithms (such as a depth-first search, which will be covered in chapter 19).

※ 9.1.1 *Implementing a Stack Using an Array*

How would you implement a stack container in memory? One strategy would be to use an array as the underlying container and keep a pointer to the first empty position past the last element:



If you want to push an element into the stack, you would add the element to the position pointed to by `top_ptr` and then increment the `top_ptr` pointer (allocating new space if needed). If no reallocation is needed, the time complexity of a single `.push()` is $\Theta(1)$. If reallocation is involved, all n elements have to be copied over, so the time complexity would be $\Theta(n)$, where n is the size of the stack.

If you want to pop an element off the stack, you would just need to decrement `top_ptr`. This takes $\Theta(1)$ time.

If you want to retrieve the top element, you would just need to dereference `top_ptr - 1`. This takes $\Theta(1)$ time. Note that we could check whether an element actually exists in the stack beforehand, but this check is omitted by the STL's `std::stack<>` implementation. This is for the sake of efficiency; the `.top()` operation would be slower if a validity check were made every time it is called. By omitting the validity check, we leave it to the programmer to ensure that they are not calling `.top()` on an empty stack.

If you want to compute the size of the stack, you could return `top_ptr - base_ptr`. Because elements in an array are contiguous in memory, the difference between the two pointers is equal to the number of elements in the array. This operation takes $\Theta(1)$ time.

If you want to check whether the stack is empty, you could check if `base_ptr` and `top_ptr` are equal. Because `top_ptr` points to the first available position in the stack, if `top_ptr` points to the first position, the first position must be empty. This check also takes $\Theta(1)$ time.

In summary, if you implement a stack using an array as the underlying container, you can achieve the following time complexities:

Method	Implementation	Complexity
`.push(val)`	Add new element at the position pointed to by `top_ptr`, then increment `top_ptr`	$\Theta(1)$ if no reallocation, $\Theta(n)$ if reallocation involved[1]
`.pop()`	Decrement `top_ptr`	$\Theta(1)$
`.top()`	Dereference `top_ptr - 1`	$\Theta(1)$
`.size()`	Calculate `top_ptr - base_ptr`	$\Theta(1)$
`.empty()`	Check if `base_ptr == top_ptr`	$\Theta(1)$

The code for the array-based stack implementation is shown below:

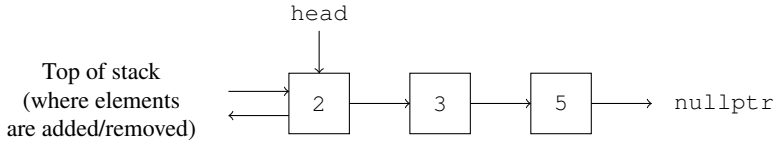
```
1   template <typename T>
2   class Stack {
3     T* data;                // pointer to array
4     T* top_ptr;             // pointer to first open position
5     size_t capacity;        // array capacity
6   public:
7     Stack(size_t num = 4);  // array capacity defaults to 4 if not specified
8     ~Stack();
9     void push(T val);
10    void pop();
11    T& top();
12    size_t size();
13    bool empty();
14  };
15
16  template <typename T>
17  Stack<T>::Stack(size_t num) : capacity{num} {
18    data = (num ? new T[num] : nullptr);          // allocate array memory
19    top_ptr = data;                               // set top_ptr to top of data
20  } // Stack()
21
22  template <typename T>
23  Stack<T>::~Stack() {
24    delete[] data;
25  } // ~Stack()
```

[1] Using amortized analysis, `.push()` is an amortized $\Theta(1)$ operation. This concept will be discussed in chapter 12.

```
27  template <typename T>
28  void Stack<T>::push(T val) {
29    if (top_ptr - data >= capacity) {             // if current array is full
30      T* temp = new T[capacity * 2];              // double capacity of array
31      for (size_t i = 0; i < capacity; ++i) {     // copy elements over
32        temp[i] = data[i];
33      } // for i
34      top_ptr = &temp[capacity];                  // reset top_ptr to new array
35      capacity *= 2;                              // update capacity value
36      std::swap(temp, data);                      // set data to new array
37      delete[] temp;                              // deallocate old array
38    } // if
39    *top_ptr++ = val;                             // writes new value to top_ptr
40  } // push()
41
42  template <typename T>
43  void Stack<T>::pop() {
44    --top_ptr;
45  } // pop()
46
47  template <typename T>
48  T& Stack<T>::top() {
49    return *(top_ptr - 1);
50  } // top()
51
52  template <typename T>
53  size_t Stack<T>::size() {
54    return top_ptr - data;
55  } // size()
56
57  template <typename T>
58  bool Stack<T>::empty() {
59    return data == top_ptr;
60  } // empty()
```

※ 9.1.2 *Implementing a Stack Using a Linked List*

We could also use a linked list to implement our stack. A singly-linked list is sufficient for implementing a stack — there is no need to store `prev` pointers, since we do not need them for any stack operations. In our linked list, we will treat the `head` as the top of the stack. This is because it is more efficient to add and remove elements from the head than it is to add and remove elements from the tail.



In this case, if we want to push an element into the stack, we would insert it at the beginning of the list. The amount of work required to insert an element at the beginning of a list does not depend on the size of the list, so the time complexity of pushing an element is $\Theta(1)$.

If we want to pop an element off the stack, we would delete the element at the `head` pointer. This requires us to set the `head` pointer to `head->next` and deallocate the old `head`. Since we are only moving some pointers around, this takes $\Theta(1)$ time.

If we want to retrieve the top element, we would have to return a reference to the data stored in the `head` node. Since we already have access to the `head` node, we can access its data in $\Theta(1)$ time.

If we want to return the size of the stack, we could count the number of nodes each time `size` is called; this would result in a $\Theta(n)$ operation. However, we could also store an additional member variable that stores the size. If we keep track of the size internally, we would not need to traverse the list each time we want the size of the stack; we could just check the value of our size variable. Even though this approach adds a bit more work to other operations (insertions and deletions would have to modify size), it allows us to retrieve the size of the list in $\Theta(1)$ time.

If we want to check if the stack is empty, we could just check to see if `head == nullptr` (or check if `size == 0` variable if we have a separate `size` variable). This check takes $\Theta(1)$ time.

In summary, if you implement a stack using a list as the underlying container, you can achieve the following time complexities:

Method	Implementation	Complexity
`.push(val)`	Insert `val` to the front of list	$\Theta(1)$
`.pop()`	Delete node at the front of list	$\Theta(1)$
`.top()`	Return reference to data of `head` node	$\Theta(1)$
`.size()`	Track `size` internally as a member variable	$\Theta(1)$
`.empty()`	Check if `head` is `nullptr`	$\Theta(1)$

The code for the list-based stack implementation is shown below:

```
1   template <typename T>
2   class Stack {
3     struct Node {
4       T val;
5       Node* next;
6       Node(T val_in) : val{val_in}, next{nullptr} {}
7     };
8     Node* head;
9     size_t sz;
10  public:
11    Stack();
12    ~Stack();
13    void push(T val);
14    void pop();
15    T& top();
16    size_t size();
17    bool empty();
18  };
19
20  template <typename T>
21  Stack<T>::Stack() : head{nullptr}, sz{0} {}
22
23  template <typename T>
24  Stack<T>::~Stack() {
25    Node* temp;
26    while (head != nullptr) {                    // destructor goes through
27      temp = head->next;                         // list and deallocates
28      delete head;                               // all the nodes
29      head = temp;
30    } // while
31  } // ~Stack()
32
33  template <typename T>
34  void Stack<T>::push(T val) {
35    Node* new_node = new Node{val};              // insert node at head
36    new_node->next = head;
37    head = new_node;
38    ++sz;
39  } // push()
40
41  template <typename T>
42  void Stack<T>::pop() {                         // remove node at head
43    Node* temp = head;
44    head = temp->next;
45    --sz;
46    delete temp;
47  } // pop()
48
49  template <typename T>
50  T& Stack<T>::top() {
51    return head->val;
52  } // top()
53
54  template <typename T>
55  size_t Stack<T>::size() {
56    return sz;
57  } // size()
58
59  template <typename T>
60  bool Stack<T>::empty() {
61    return head == nullptr;
62  } // empty()
```

Which stack implementation is better, the one that uses an array or the one that uses a linked list? It turns out that both containers support all stack operations in $\Theta(1)$ time. However, if you were to time these implementations, you would find that the array implementation is slightly faster than the linked list implementation. This is because, even though both implementations run in $\Theta(1)$ time, the constant factor of the array implementation is smaller. The linked list implementation also has higher memory overhead compared to the array (since additional pointers need to be stored with each element).

9.2 The STL Stack Container

The C++ standard template library provides a pre-implemented stack container for you in the `<stack>` library. To use this stack in your program, you will need to `#include <stack>` at the top of your code file and declare an object of type `std::stack<>`. A `std::stack<>` supports the following operations:

Function	Behavior
`.push(val)`	Adds `val` to the top of the stack
`.pop()`	Removes top element from the stack (undefined behavior if empty)
`.top()`	Returns a reference to the top element in the stack (undefined behavior if empty)
`.size()`	Returns the number of elements in the stack
`.empty()`	Checks if the stack is empty

In C++, the `.top()` operation returns a reference to the top element in the stack but does not remove it. The `.pop()` operation removes the top element but does not return it. This was designed with speed in mind; in cases where you only want to do one of these operations, there is no point in doing both. If you want to retrieve and remove the top element in a stack, you must call both `.top()` and `.pop()`.

The following code provides an example that utilizes a `std::stack<>`:

```
1  std::stack<int32_t> s;              // initializes a stack with variable name 's'
2  s.push(5);                          // pushes 5 onto the stack
3  s.push(3);                          // pushes 3 onto the stack
4  int32_t x = s.top();                // x stores the value 3
5  s.pop();                            // 3 is removed from the stack
6  s.top() = 4;                        // the top element is changed from 5 to 4
7  s.pop();                            // 4 is removed from the stack
8  std::cout << s.size() << '\n';      // stack is empty, so this prints out 0
```

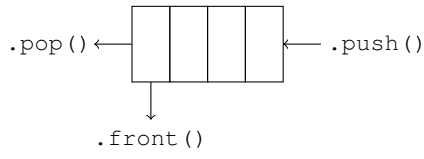
9.3 Introduction to Queues

A **queue** is linear data structure that supports first-in, first-out (FIFO) data access. When you retrieve elements out of a queue, you remove the elements in the same order in which you inserted them.

For a real-life example of how a queue works, think of an office hours queue where students wait in line for help. When a student adds themselves to the queue, they put themselves at the end of the line. If an instructor is available, they get the student who has been in the queue the longest. A queue works the same on data. When you *enqueue* an element, you add the element to the *back* of the queue. When you *dequeue* an element, you remove the element at the *front* of the queue. In C++, elements are enqueued using `.push()` and dequeued using `.pop()`.

The interface of a queue:

Function	Behavior
`.push(val)`	Adds `val` to the back of the queue
`.pop()`	Removes top element from the front of the queue
`.front()`	Returns a reference to the element at the front of the queue
`.size()`	Returns the number of elements in the queue
`.empty()`	Checks if the queue is empty



It is important to remember that stacks use `.top()` to retrieve the next element, while queues use `.front()` to retrieve the next element. The other method names of the two containers are identical.

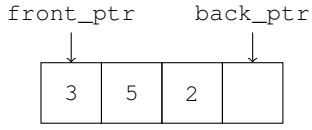
※ 9.3.1 *Implementing a Queue Using an Array (Circular Buffer)*

Like with stack, a queue can be implemented using an array or a linked list as the underlying structure. However, the implementation of a queue is slightly more complicated, since operations need to be supported on *both* ends of the underlying container. One method for implementing a queue using an array is to use a **circular buffer**. In a circular buffer, we allow elements to loop off the end of the array to support efficient deletions and insertions on opposing ends of the queue. To track the order of elements in a circular buffer, two pointers are used: `front_ptr`, which points to the element at the front of the queue, and `back_ptr`, which points to the first open position at the back of the queue.

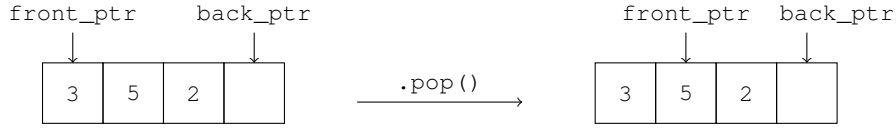
Consider the following array, which represents our circular buffer:

Here, the value `1` is at the front of the queue (and is next to be popped off), and the value `3` is at the back of the queue (the element most recently added to the queue). We know this because of the positions of `front_ptr` and `back_ptr`, which indicate the two ends of our queue.

Because a queue allows elements to be modified at both ends of the container, we have to be a bit more careful when storing our elements. With an array-based stack implementation, we knew that if `top_ptr` reached the end of the array, the stack was full and we needed to allocate more space. This assumption cannot be made with a queue, however, since the first element is not guaranteed to be in the leftmost position of the array! For instance, if we popped an element off the above queue, `1` would get removed and the first element would be located at the *second* position of the array. This is where the circular nature of the circular buffer comes in. Every time an element is pushed in, we add the element to the position pointed to by `back_ptr` and increment `back_ptr` by one. However, if `back_ptr` ends up off the end of the array, we *wrap around* back to the beginning of the array and check if the position is available. This process is illustrated below:

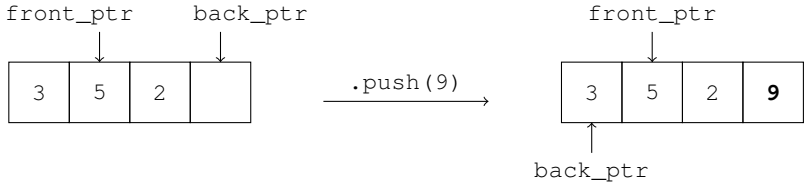


Suppose we call `.pop()`, which removes the element at the front of the queue. Since `front_ptr` is a pointer that points to the element at the front of the queue, we can "remove" the front element by moving `front_ptr` forward one position.

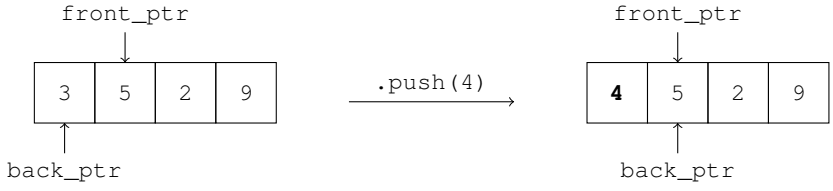


Notice here that the value `3` was not physically deleted from the array. This is because our queue, if implemented correctly, has no way of accessing this `3` since it is no longer a valid entry in our queue (which only looks at the values between the front and back pointers). As a result, there is no need to do the additional work of clearing this value out, as the queue should never access it anyway. We will only reset this value when need to access index 0 of our array again.

Now, suppose we push the value `9` to the back of the queue by making a call to `.push(9)`. Since `back_ptr` points to the first open position in the queue, we can write `9` to this location and increment `back_ptr` by one position. Since a circular buffer wraps around, `back_ptr` ends up pointing to index 0 of the array.



If were to push the value `4` into the queue (by calling `.push(4)`), we would write `4` to the value pointed to by `back_ptr` and increment `back_ptr` forward one position. This is shown below.



With this procedure, `front_ptr` will always point to the element at the front of the queue, and `back_ptr` will always point to the next available position in the array. If incrementing `back_ptr` causes it to point to the same position as `front_ptr`, that means the array is filled to capacity, and reallocation would be needed to support any more elements. During reallocation, the elements in the array are copied to a larger array from front to back, so that the front element ends up at the first position of the new array. For example, if reallocation were done on the above array, the elements would be copied to a new array in the order [5, 2, 9, 4], since 5 is the element at the front.

The operations for a circular buffer, along with their complexities, are shown below:

Method	Implementation	Complexity
`.push(val)`	Insert `val` at position pointed to by `back_ptr` and increment `back_ptr`, wrapping around to the beginning of the array if necessary. If `back_ptr` is ever incremented to the position of `front_ptr`, a larger array is allocated and elements are copied over in order from front to back.	$\Theta(1)$ if no reallocation, $\Theta(n)$ if reallocation involved[2]
`.pop()`	Increment `front_ptr`	$\Theta(1)$
`.front()`	Dereference `front_ptr` and return its value	$\Theta(1)$
`.size()`	If the address of `back_ptr` is larger than that of `front_ptr`, return the difference between the two pointers; otherwise, add the capacity of the array to the difference (i.e., return array capacity + `back_ptr` - `front_ptr`)	$\Theta(1)$
`.empty()`	Check if `back_ptr` == `front_ptr`	$\Theta(1)$

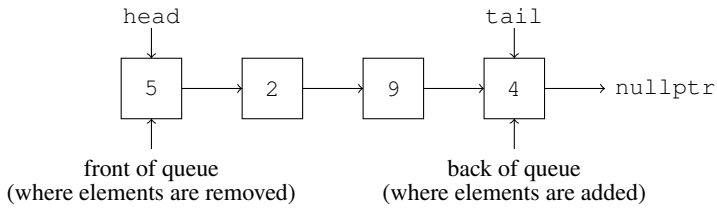
[2]Using amortized analysis, `.push()` is an amortized $\Theta(1)$ operation. This concept will be discussed in chapter 12.

The following code implements a queue using a circular buffer:

```
template <typename T>
class Queue {
  T* data;
  T* front_ptr;
  T* back_ptr;
  size_t capacity;
public:
  // array capacity defaults to 4 if not specified
  Queue(size_t num = 4);
  ~Queue();
  void push(T val);
  void pop();
  T& front();
  size_t size();
  bool empty();
};

template <typename T>
Queue<T>::Queue(size_t num) : capacity{num} {
  data = (num ? new T[num] : nullptr);
  front_ptr = back_ptr = data;
} // Queue()

template <typename T>
Queue<T>::~Queue() {
  delete[] data;
} // ~Queue()

template <typename T>
void Queue<T>::push(T val) {
  *back_ptr++ = val;                          // write val to back_ptr
  if (back_ptr >= data + capacity) {          // loop around if off end
    back_ptr = data;
  } // if
  if (back_ptr == front_ptr) {                // if back hits front, reallocate
    T* temp = new T[capacity * 2];            // double capacity of array
    for (size_t i = 0; i < capacity; ++i) {
      temp[i] = *front_ptr++;                 // write data starting from front
      if (front_ptr == data + capacity) {     // loop around if off end
        front_ptr = data;
      } // if
    } // for i
    front_ptr = temp;                         // reset front_ptr to new array
    back_ptr = &temp[capacity];               // reset back_ptr to new array
    capacity *= 2;                            // update capacity
    std::swap(temp, data);                    // set data to new array
    delete[] temp;                            // deallocate old array
  } // if
} // push()

template <typename T>
void Queue<T>::pop() {
  ++front_ptr;
  if (front_ptr >= data + capacity) {         // loop around if off end
    front_ptr = data;
  } // if
} // pop()

template <typename T>
T& Queue<T>::front() {
  return *front_ptr;
} // front()

template <typename T>
size_t Queue<T>::size() {
  if (back_ptr >= front_ptr) {
    return back_ptr - front_ptr;
  } // if
  else {
    return capacity + back_ptr - front_ptr;
  } // else
} // size()

template <typename T>
bool Queue<T>::empty() {
  return back_ptr == front_ptr;
} // empty()
```

※ 9.3.2 *Implementing a Queue Using a Linked List*

A queue can also be implemented with a linked list as the underlying container. The process is similar to implementing a stack with a linked list; the only major difference is that elements are added to the *tail* of the linked list rather than the head:



When `.push()` is called, an element is added to the back of the linked list. With a tail pointer, this can be done in $\Theta(1)$ time. When `.pop()` is called, the element at the front of the linked list is removed. The operations for a list-based queue are shown below with their complexities:

Method	Implementation	Complexity
`.push(val)`	Inserts `val` to the back of the list	$\Theta(1)$ if tail pointer, $\Theta(n)$ if no tail pointer
`.pop()`	Delete `head` node of the list	$\Theta(1)$
`.front()`	Return reference to data in `head` node	$\Theta(1)$
`.size()`	Track and update size internally when other methods modify the list, or count nodes each time	$\Theta(1)$ if size stored internally, $\Theta(n)$ if nodes counted every time
`.empty()`	Check if `head == nullptr`	$\Theta(1)$

The following code implements aqueue using a *singly-linked list*:

```
1   template <typename T>
2   class Queue {
3     struct Node {
4       T val;
5       Node* next;
6       Node(T val_in) : val{val_in}, next{nullptr} {}
7     };
8     Node* head;
9     Node* tail;
10    size_t sz;
11  public:
12    Queue();
13    ~Queue();
14    void push(T val);
15    void pop();
16    T& front();
17    size_t size();
18    bool empty();
19  };
20
21  template <typename T>
22  Queue<T>::Queue() : head{nullptr}, tail{nullptr}, sz{0} {}
23
24  template <typename T>
25  Queue<T>::~Queue() {
26    Node* temp;
27    while (head != nullptr) {
28      temp = head->next;
29      delete head;
30      head = temp;
31    }
32  }
33
34  template <typename T>
35  void Queue<T>::push(T val) {
36    // insert node at tail
37    Node* new_node = new Node{val};
38    if (tail != nullptr) {
39      tail->next = new_node;
40    } // if
41    else {
42      head = new_node;
43    } // else
44    tail = new_node;
45    ++sz;
46  } // push()
```

```
48  template <typename T>
49  void Queue<T>::pop() {
50    // remove node at head
51    Node* temp = head;
52    head = temp->next;
53    --sz;
54    delete temp;
55    if (head == nullptr) {
56      tail = nullptr;
57    } // if
58  } // pop()
59
60  template <typename T>
61  T& Queue<T>::front() {
62    return head->val;
63  } // front()
64
65  template <typename T>
66  size_t Queue<T>::size() {
67    return sz;
68  } // size()
69
70  template <typename T>
71  bool Queue<T>::empty() {
72    return head == nullptr;
73  } // empty()
```

9.4 The STL Queue Container

The C++ standard template library provides a pre-implemented queue container for you in the `<queue>` library. To use this queue in your program, you will need to `#include <queue>` at the top of your code file and declare an object of type `std::queue<>`. A `std::queue<>` supports the following operations:

Function	Behavior
`.push(val)`	Adds `val` to the back of the queue
`.pop()`	Removes the element at the front of the queue (undefined behavior if empty)
`.front()`	Returns a reference to the element at the front of the queue (undefined behavior if empty)
`.size()`	Returns the number of elements in the queue
`.empty()`	Checks if the queue is empty

Similar to a `std::stack<>`, a `std::queue<>` separates element retrieval and removal into two different methods. A call to `.front()` returns the element at the front of the queue, but does not remove it. In contrast, a call to `.pop()` removes the element at the front, but does not return it. The following code goes through an example that utilizes a `std::queue<>`:

```
1  std::queue<int32_t> q;              // initializes a queue with variable name 'q'
2  q.push(5);                          // pushes 5 onto the queue
3  q.push(3);                          // pushes 3 onto the queue
4  int x = q.front();                  // x now stores the value 5
5  q.pop();                            // 5 is removed from the queue
6  q.front() = 4;                      // the front element is changed from 3 to 4
7  q.pop();                            // 4 is now removed from the queue
8  std::cout << q.size() << '\n';      // queue is empty, so this prints out 0
```

9.5 Solving Problems Using Stacks and Queues

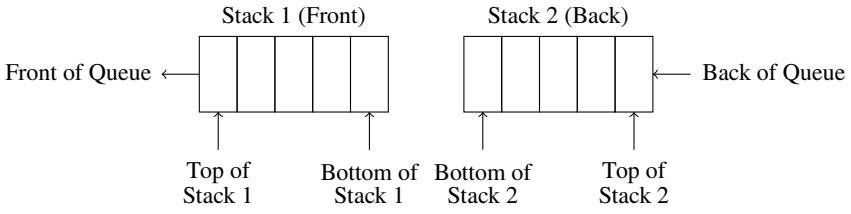
Stacks and queues can be quite useful in solving several different types of programming problems. In this section, we will explore some problems that utilize these containers.

※ 9.5.1 *Implementing a Queue Using Two Stacks*

Example 9.1 You are given two stacks, each of which supports `.top()`, `.push()`, `.pop()`, and `.size()`. If you are given no other containers, how can you use the two stacks to implement a queue that supports `.front()`, `.push()`, `.pop()`, and `.size()`?

The thing to notice about a stack is that it is a *one-ended* container. Since insertion and removal are both done on the same end, there is no way to access elements at the bottom of a stack. On the other hand, a queue must manage data on both ends. This is why a single stack cannot be used to simulate a queue, since there is only one place to insert and remove data.

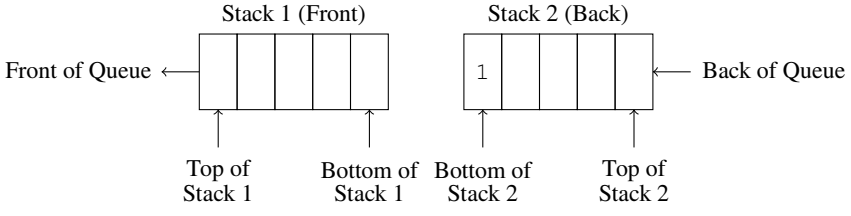
However, this changes if you are given two stacks instead of one. With two stacks, there are now two places where data can be added or removed: the top of stack 1, and the top of stack 2. As a result, we can treat one stack as the "front" of our queue (where elements are removed), and the other stack as the "back" of our queue (where elements are inserted). An illustration of this idea is shown below:



A summary of these operations is shown below, where `stackFront` is the front stack and `stackBack` is the back stack:

```
.push(x): stackBack.push(x);
.front(): return stackFront.top();
.pop()  : stackFront.pop();
.size() : return stackFront.size() + stackBack.size();
```

However, this implementation has a slight problem. Suppose we push 1 into our queue and immediately try to pop it. Using the operations above, we would push 1 into our back stack and then pop an element out of our front stack.



However, there is nothing to pop since our front stack is empty! In fact, our queue's `.pop()` and `.front()` methods do not work if all the elements are in the back stack.

To solve this, we will need to move *all* the elements from the back stack to the front stack if `.pop()` or `.front()` is ever invoked while the front stack is empty. We need to remove every element from the back stack because the next element we want to retrieve from our queue is at the bottom of this back stack, and it cannot be accessed before all the elements on top of it are removed first.

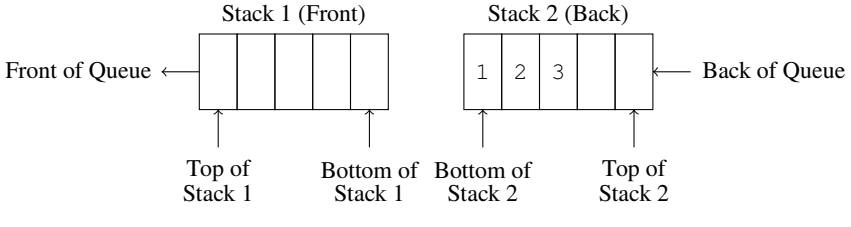
```
QueueWithTwoStacks::front() {
  if (stackFront.empty()) {
    while (!stackBack.empty()) {
      stackFront.push(stackBack.top());
      stackBack.pop();
    } // while
  } // if
  return stackFront.top();
} // front()
```

```
QueueWithTwoStacks::pop() {
  if (stackFront.empty()) {
    while (!stackBack.empty()) {
      stackFront.push(stackBack.top());
      stackBack.pop();
    } // while
  } // if
  stackFront.pop();
} // pop()
```

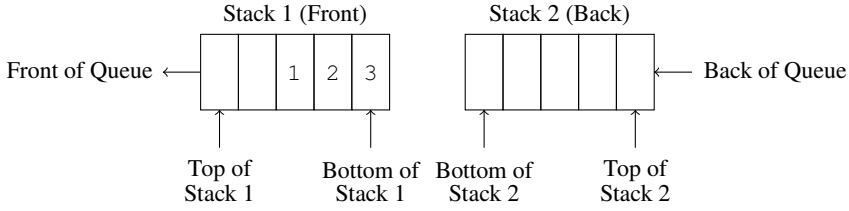
Let's look at how this process works with the following example:

```
1   QueueWithTwoStacks<int> q;
2   q.push(1);
3   q.push(2);
4   q.push(3);
5   q.pop();
6   q.push(4);
7   q.push(5);
8   q.pop();
9   q.pop();
10  q.pop();
```

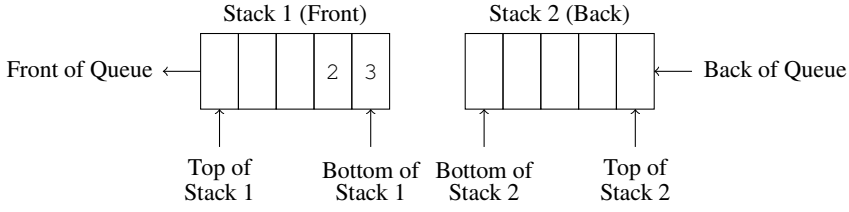
First, we push 1, 2, and 3 to the back of our queue. This is done by pushing these three elements into our back stack:



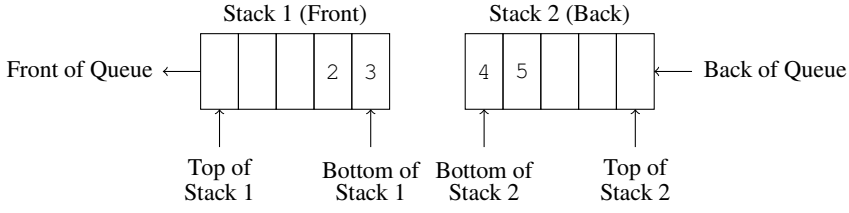
Next, we pop out an element (which should be 1, since it has been in our queue the longest). Since the front stack is empty, we first transfer all the elements from the back stack into the front stack:



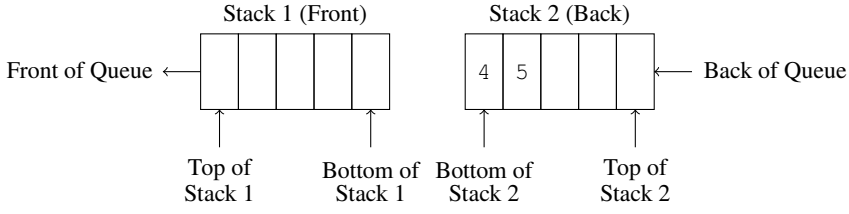
We then pop out the next element in the queue by popping from the front stack:



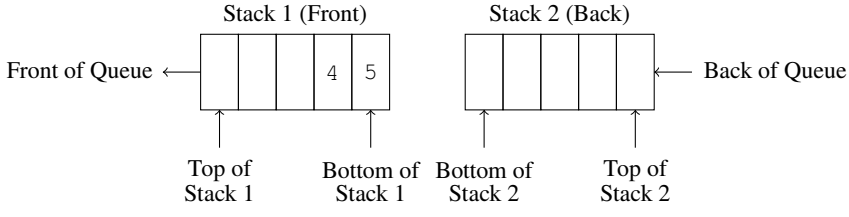
We then push 4 and 5 into our queue, which is added to the back stack:

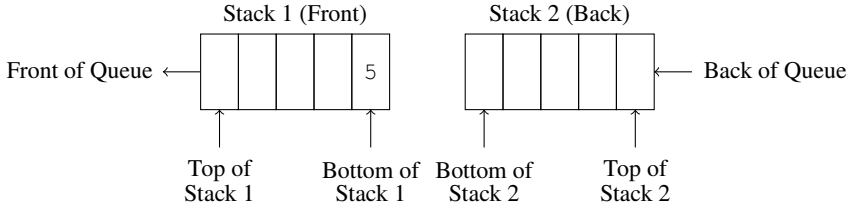


We then pop out three elements from the queue. Since the front stack is not empty, we can just pop out 2 and 3 without moving anything:



However, for our third pop, the front queue is empty. Thus, we will have to transfer all the elements from the back stack to the front stack before popping out the next element (which is the value 4 in this case):





What's the worst-case time complexity of this queue? Here, `.push()` and `.size()` both take $\Theta(1)$ time, but `.front()` and `.pop()` both take worst-case $\Theta(n)$ time, where n is the size of the queue. This is because these functions may be responsible for moving all the elements from the back stack to the front stack. It may seem that this queue is pretty terrible, since `.front()` and `.pop()` should not be taking linear time! However, looks can be deceiving — even though the worst-case time complexity of a single `.front()` or `.pop()` call may be $\Theta(n)$, this can only happen if the front stack is empty. Using something known as amortized analysis, we can prove that this worst-case scenario happens so rarely that both `.front()` and `.pop()` can essentially be treated as $\Theta(1)$ operations. This concept will be covered in a later chapter.

※ 9.5.2 *Sorting a Stack*

Example 9.2 Suppose you are given a stack of values, and you are told to sort this stack using only the supported stack operations of `.top()`, `.push()`, `.pop()`, and `.size()`. You are given an additional auxiliary stack (which you may use to help you sort the input stack), but you are not allowed to use any additional containers. How can you solve this problem?

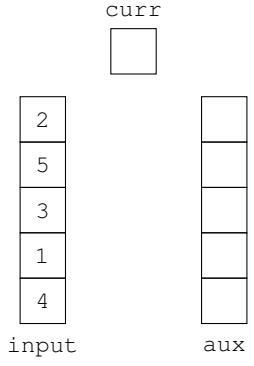
The idea here is to utilize our auxiliary stack to help keep our values sorted. Even though our input stack is not sorted, we can push elements into the auxiliary stack in such a way to keep its elements sorted. This is done by making sure that bigger values are always sent to the bottom of the auxiliary stack. This process is summarized below:

1. Save the value at the top of the input stack in a variable (we will call this value the current element).
2. Since we want larger items to be sent to the bottom of the auxiliary stack, move all elements in the auxiliary stack that are smaller than the current element back into the input stack.
3. Once the auxiliary stack only contains elements larger than the current element, push the current element into the auxiliary stack.

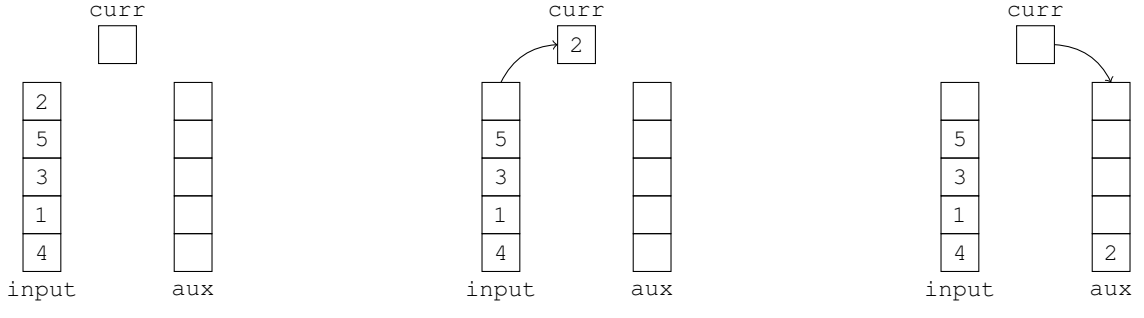
This algorithm is illustrated in the code below:

```
1  while (!input.empty()) {
2    curr = input.top();
3    input.pop();
4    while (!aux.empty() && aux.top() < curr) {
5      input.push(aux.top());
6      aux.pop();
7    } // while
8    aux.push(curr);
9  } // while
```

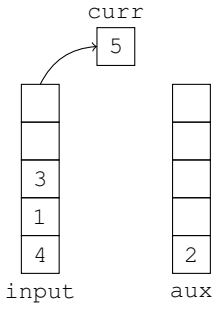
Let's look at this algorithm in action. In the figure below, `input` represents the input stack, `aux` represents the auxiliary stack, and `curr` represents the current element.



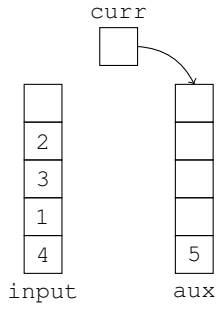
First, we look at the element at the top of the input stack, 2. Since the auxiliary stack is empty, we can immediately push 2 to into our auxiliary stack without any problems.



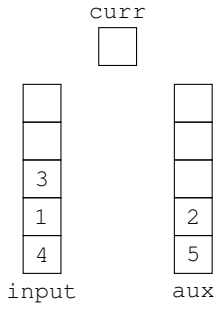
The next element to be considered is 5, which is at the top of the input stack. We want to insert 5 into `aux` so that it is in the correct sorted position relative to the other values in `aux`. To do this, we will first move all elements smaller than 5 back into `input`, push 5 into `aux`, and reinsert the elements we moved back into `aux`. In this case, we would move 2 back to `input`, add 5 into `aux`, and readd 2 into `aux`.



Set 5 (top of input stack) as current value

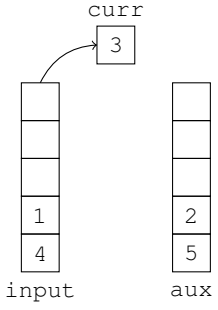


Move all `aux` values less than 5 into `input` and add 5 to `aux`

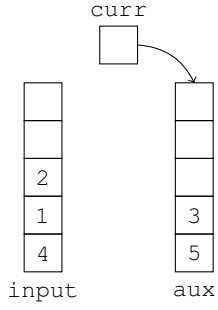


Readd previously removed elements back into auxiliary stack

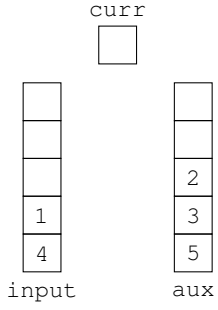
The next element to be considered is 3. To insert 3 into correct sorted position, we first move 2 from the auxiliary stack to the input stack. Then, we insert 3 into the auxiliary stack. Lastly, we add 2 back into the auxiliary stack.



Set 3 (top of input stack) as current value

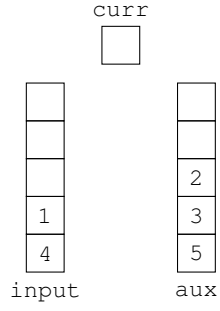


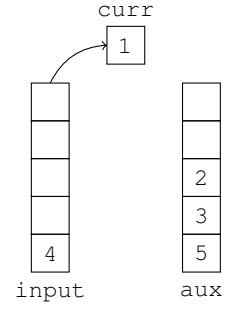
Move all `aux` values less than 3 into `input` and add 3 to `aux`

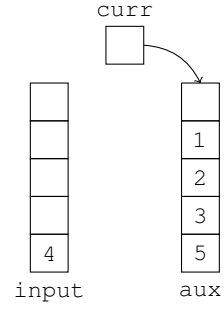


Readd previously removed elements back into auxiliary stack

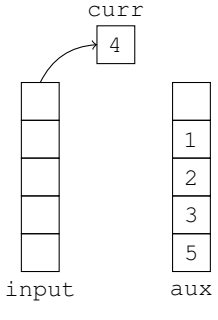
The next element to be considered is 1. Since 1 is already smaller than all the elements in `aux`, we can just push 1 into `aux` directly.



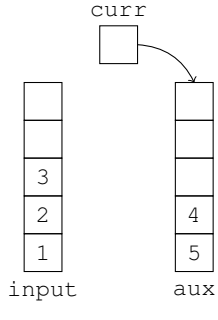




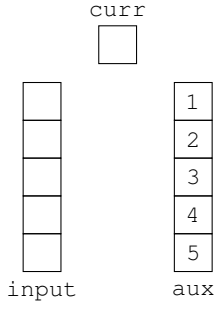
The next element to be considered is 4. To insert 4 into correct sorted position, we first move 1, 2, and 3 from the auxiliary stack to the input stack. Then, we insert 4 into the auxiliary stack. Lastly, we add 1, 2, and 3 back into the auxiliary stack.



Set 4 (top of input stack) as current value



Move all `aux` values less than 4 into `input` and add 4 to `aux`



Readd previously removed elements back into auxiliary stack

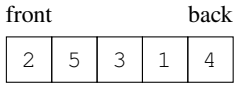
After this algorithm runs to completion, the values in `aux` will be sorted. The worst-case time complexity for this algorithm is $\Theta(n^2)$, when the input stack is already sorted. When this happens, you will have to move all the elements out of `aux` whenever you want to push the current element in. For instance, to push the second value into `aux`, you would need to move out the first element and later readd it in. To push the third value into `aux`, you would need to move out the first *and* second elements and later readd them in. To push the n^{th} element into `aux`, you would need to move out all $n-1$ elements before it and later readd them in. The total number of elements moved in this case would be $1 + 2 + \ldots + (n-1)$, which is $\Theta(n^2)$. The auxiliary space used by this algorithm is $\Theta(n)$, since an additional stack of size n was used to solve this problem.

※ 9.5.3 *Sorting a Queue*

Example 9.3 Suppose you are given a queue of values, and you are told to sort this queue using only the supported queue operations of `.front()`, `.push()`, `.pop()`, and `.size()`. Using no additional containers, how can you solve this problem?

This is similar to the previous problem, but this time you are asked to sort a queue rather than a stack. This difference in container actually makes quite a difference. Unlike a stack, a queue is not single-ended — if you insert elements into a stack, the element at the bottom of the stack cannot be retrieved without first removing and storing the elements above it elsewhere (think about a stack of books: to remove the book at the bottom of the stack, you must first move all the books on top of it). This was why we needed an auxiliary stack for the previous sorting problem. However, to reorder elements in a queue, there is no need to keep track of an auxiliary container. Because data in a queue is inserted and removed on different ends, all elements can eventually be accessed using $\Theta(1)$ auxiliary space by just popping each element off the front of the queue and repushing it onto the back.

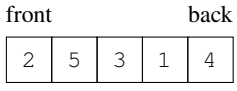
Suppose we are given the following unsorted queue:



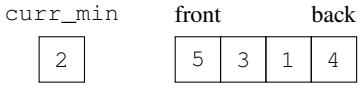
Assuming that this queue only supports `.push()`, `.pop()`, `.front()`, and `.size()`, we can devise the following algorithm to sort it:

1. Make multiple passes through the queue by popping elements off the front and reinserting them on the back. During each pass, keep track of the smallest value you have seen so far. If you encounter an element that is the smallest you have seen so far, do not push it to the back until a smaller element is found or the pass completes.
2. At the end of each pass, push the minimum value found to the back of the queue. This value is now in sorted order and should be ignored for all future passes.
3. Keep on repeating steps 1 and 2 until the queue becomes sorted.

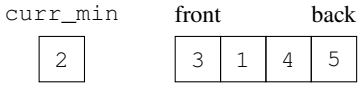
Let's look at this algorithm in action:



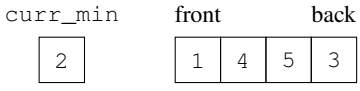
Store the first value in a separate variable that keeps track of the smallest value you have seen on the first pass. After doing so, pop this value off the queue.



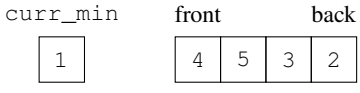
Retrieve the next value from the queue and compare it with the current minimum value encountered. In this case, 5 is not less than 2, so pop it off and push it to the back.



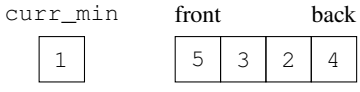
The next value in the queue, 3, is also not less than 2, so it also gets popped off and pushed to the back.



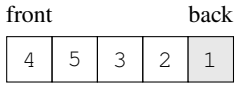
The next value in the queue is 1, which is less than 2. Because we found a new smallest element, we push 2 to the back of the queue and replace `curr_min` with 1.



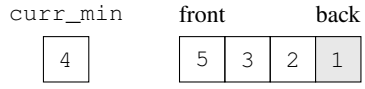
The next value in the queue is 4, which is not less than 1. It gets popped off and pushed to the back.



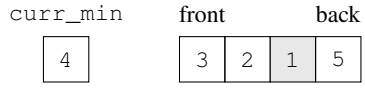
We have now completed our first pass of the queue. Since `curr_min` is 1 at the end of this first pass, 1 must be the smallest element in the queue. We push 1 to the back of the queue and keep its position fixed for the remainder of the algorithm.



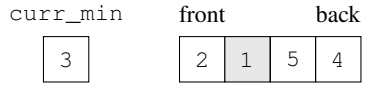
Now, let's start our second pass. Initialize `curr_min` to the first value encountered on the second pass, which in this case is 4.



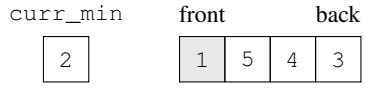
The next value in the queue, 5, is not less than 4. It gets popped off and pushed to the back.



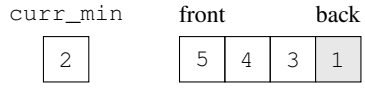
The next value in the queue, 3, is less than 4. Thus, we push 4 to the back and replace `curr_min` with 3.



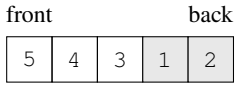
The next value in the queue, 2, is less than 3. Thus, we push 3 to the back and replace `curr_min` with 2.



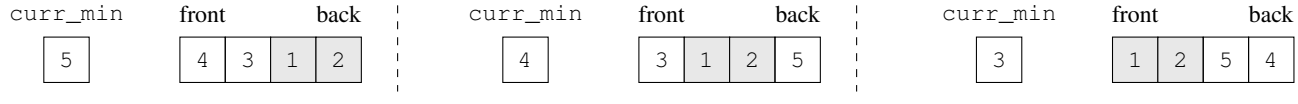
The next value in the queue, 1, has already been considered. Thus, we ignore it, and it gets popped off and pushed to the back.

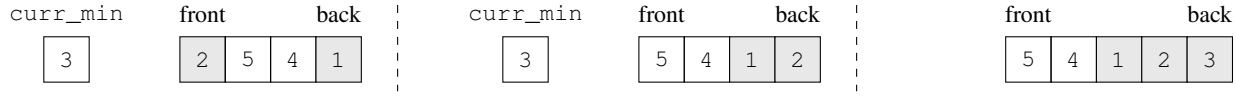


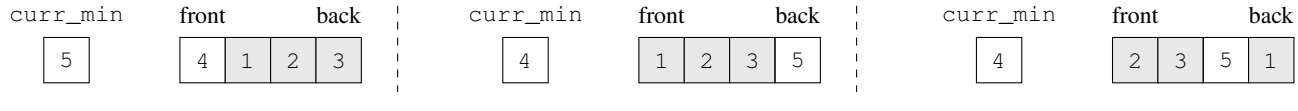
We have finished our second pass of the queue. Since the value of `curr_min` is 2 at the end of this second pass, 2 must be the second smallest value in the queue. We push it to the back and fix its position.

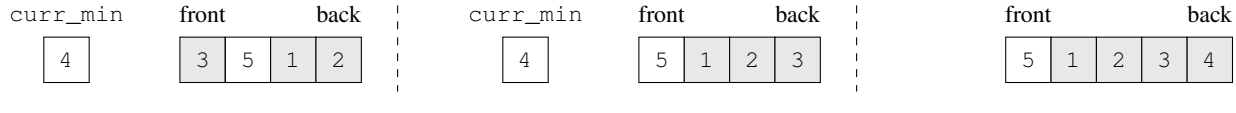


We can continue this process, which is shown below (read from left to right first, then top to bottom). On the third pass, 3 gets fixed in position. Then, on the fourth pass, 4 gets fixed in position.

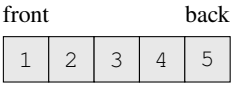








At this point, 5 is the last element that hasn't been fixed in position. Because of this, we know that 5 must be the largest element in the queue; we can just pop it off and push it to the end.



Our queue is now sorted. The time complexity of this algorithm is $\Theta(n^2)$. This is because we complete approximately n passes of the queue, where each pass makes n comparisons. As a result, the number of elements we check throughout the lifetime of the algorithm is $\Theta(n \times n)$, or $\Theta(n^2)$. The auxiliary space used by this algorithm is $\Theta(1)$. This is because we did the sorting in-place, using just the input queue itself to complete our algorithm. We did not have to allocate additional memory to solve this problem!

※ 9.5.4 *Evaluating Reverse Polish Notation*

Example 9.4 *Reverse Polish notation* is a mathematical notation in which operators follow their operands. For example, if we wanted to add together 1 and 2, reverse Polish notation would express this addition as "1 2 +" instead of "1 + 2". As another example, the expression "(3 - 4) * 5" is written as "3 4 - 5 *" in reverse Polish notation: the "3 4 -" indicates that 3 and 4 should be subtracted, and combining it with "5 *" indicates the this result should be multiplied by 5.

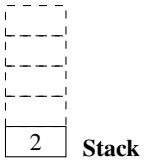
Suppose you are given a valid arithmetic expression in reverse Polish notation (in the form of a vector of strings), with the following valid operators: `"+"`, `"-"`, `"*"`, and `"/"`. Write a function that evaluates the expression's result. You may assume that the expression always evaluates to a solution, and there will never be a situation where you will divide by zero.

Example: Given `["2", "3", "*", "4", "5", "*", "+"]`, you would return 26, since that is the result of (2 * 3) + (4 * 5).

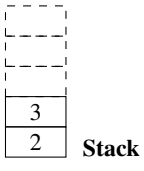
In this problem, each operator acts upon the results of the operands that precede it (for example, the "+" in "2 3 * 4 5 * +" multiplies the result of "2 * 3" with the result of "4 * 5"). Thus, whenever we encounter an operator in our input, we would need to identify the values of the most recent expressions encountered so far (e.g., to apply the "+" operator in our example, we would need to know the values of "2 3 *" and "4 5 *"). A stack would therefore be a good container choice for solving this problem, since it provides this LIFO behavior for identifying these values.

To solve this problem, we will iterate over the input values. Every time we encounter a number, we would push it into a stack. Every time we encounter an operator symbol, we would take out the two values at the top of the stack, apply the operator on these values, and push the result back into the stack. Once we reach the end of the input, the remaining value left in the stack must be the solution of our original expression.

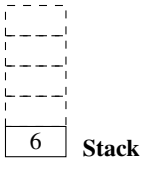
Consider our example input of `["2", "3", "*", "4", "5", "*", "+"]`. The first value is the number 2, so we push it onto a stack.



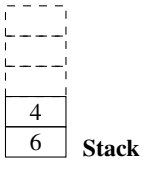
The next value is the number 3, so we push it onto the stack.



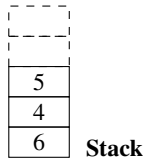
The next value is the operator "*", which indicates that we should multiply the results of the previous two expressions. Thus, we will take out the top two values in our stack (2 and 3), multiply them together, and push in the result of 6 back onto the stack.



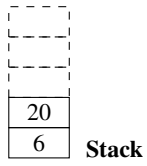
The next value is the number 4, so we push it onto the stack.



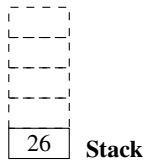
The next value is the number 5, so we push it onto the stack.



The next value is the operator "$\star$", which indicates that we should multiply the results of the previous two expressions. Thus, we will take out the top two values in our stack (4 and 5), multiply them together, and push in the result of 20 back onto the stack.



The next value is the operator "+", which indicates that we should add the results of the previous two expressions. Thus, we will take out the top two values in our stack (6 and 20), add them together, and push in the result of 26 back onto the stack.



There are no more values to process in our input, so the remaining value in the stack, 26, must be our final solution.

This solution to the problem is implemented in the code below:

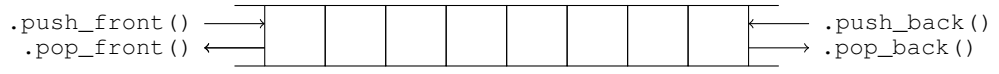
```
1   // helper function that applies an operator on two values and returns the result
2   int32_t apply_operation(int32_t a, int32_t b, char op) {
3     switch (op) {
4       case '+': return a + b;
5       case '-': return a - b;
6       case '*': return a * b;
7       case '/': return a / b;
8       default: throw std::invalid_argument{"Invalid operation"};
9     } // switch
10  } // apply_operation()
11
12  int32_t evaluate_reverse_polish_notation(std::vector<std::string>& tokens) {
13    std::stack<int32_t> values;
14    for (const std::string& token : tokens) {
15      if ((token == "+" || token == "-" || token == "*" || token == "/")) {
16        int32_t value1 = values.top();
17        values.pop();
18
19        int32_t value2 = values.top();
20        values.pop();
21
22        values.push(apply_operation(value2, value1, token[0]));
23      } // if
24      else {
25        values.push(std::stoi(token));  // stoi() converts string to integer
26      } // else
27    } // for token
28    return values.top();
29  } // evaluate_reverse_polish_notation()
```

Since the algorithm traverses over each value in the input vector once, its time complexity is $\Theta(n)$, where n is the size of the input vector. The auxiliary space used by the algorithm is also $\Theta(n)$, due to the additional stack that we allocated to build up our solution (whose size depends on the number of values we have to process).

There are certainly more problems that can be solved using stacks and queues. In fact, you will see these containers again when graphs and searching algorithms are covered in chapter 19. You have probably already encountered several use cases where stacks may be useful throughout your programming career: they play an important role in storing data during recursive calls, and they can be used by compilers and text editors to check the validity of parentheses. A queue is also a very important container type in programming. Just as a preview for what is to come, queues can be used to discover the shortest path between two nodes of a graph (known as a breadth-first search). They also play a vital role in operating systems, allowing your computer to manage CPU and disk usage for different system processes (which is beyond the scope of this class).

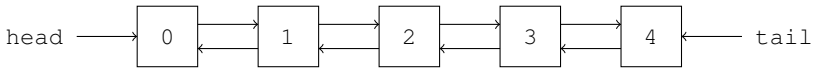
9.6 The STL Deque Container

A **deque** (pronounced as "deck"), or a **d**ouble-**e**nded **que**ue, is an STL container that supports both the functionality of a stack and a queue. Unlike a vector, where efficient insertion and deletion is only supported at the back of the container, a deque supports efficient insertion and deletion operations on *both* ends of the container.



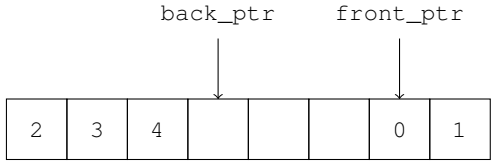
According to the C++ standard, a deque must support $\Theta(1)$ time insertion and removal from both the front (`.push_front()` and `.pop_front()`) and the back (`.push_back()` and `.pop_back()`). Deques must also support $\Theta(1)$ random access using `operator[]`, and insertion and deletion from either end of a deque must *never* invalidate pointers or references to the rest of the elements in the container.

How can a deque be implemented in memory? Since deques must support constant time insertion and deletion from both ends, a linked list implementation may seem tempting at first (as lists support constant time insertion and deletion, assuming that a `tail` pointer is included).



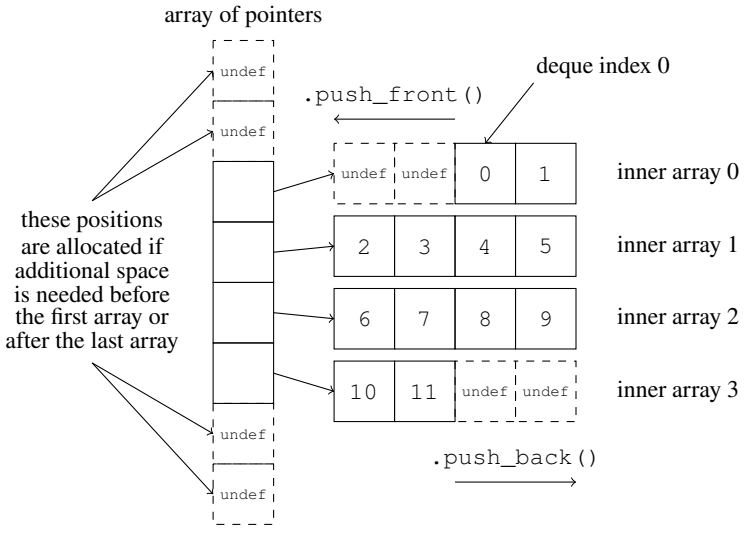
However, this approach does not fully satisfy the C++ standard. This is because a linked list deque cannot support `operator[]` in $\Theta(1)$ time. As mentioned previously, lists do not support random access; if you want to retrieve the n^{th} element of a list, you would have to iterate through all $n-1$ elements before it. A list is therefore unsuitable for implementing a deque that supports random access.

If linked lists do not work because they cannot provide random access, what about an array? Even though arrays cannot support constant time insertions or deletions from the front, we can remove this problem by using a *circular array* instead. Just like with our circular buffer queue implementation, we will use two pointers to keep track of the front and back of our deque. This allows us to remove and insert elements to the front of our deque without having to shift elements over in our underlying array.



It turns out that the circular array approach does not satisfy the C++ standard either. Even though an array provides random access in constant time, pointer invalidation now becomes a concern. Recall that a deque cannot invalidate any pointers if values are inserted or deleted from the front or back of the container. However, the circular array approach would occasionally require reallocation, which would cause all elements to move to a new location in memory (and thus would invalidate existing pointers).

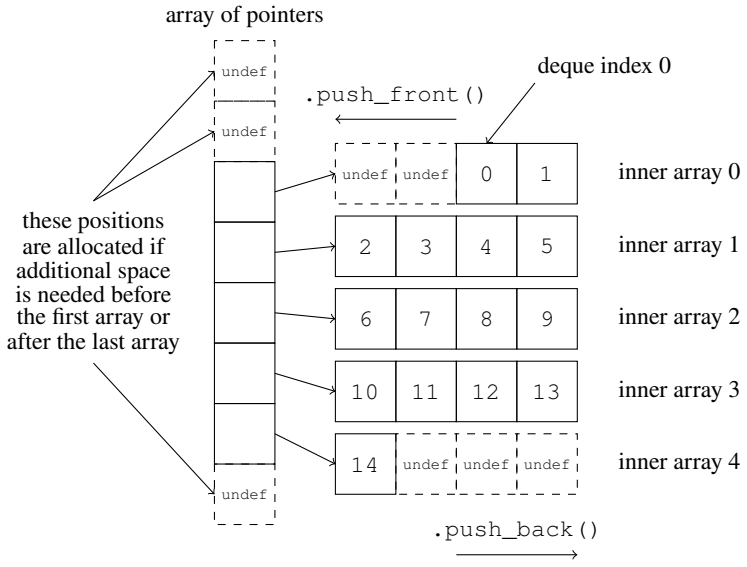
It turns out that the complex functionality of deques make them a bit more difficult to implement. We will discuss one common implementation in this section.[3] Behind the scenes, a deque is implemented as an array of pointers to other arrays, where each inner array has a constant size (usually a power of two). This is shown in the illustration below:



The outer array stores pointers to constant-size inner arrays, and the inner arrays store the deque's data. The behavior of the inner arrays change depending on whether the array is at the front or back of the deque. For the last array in the deque, data is added toward the back of the array (e.g., if `12` were pushed to the back of the deque, it would be added to the position directly after `11` in inner array 3). For the first array in the deque, data is added toward the front of the array (e.g., pushing `12` to the front of the deque would add it to the position directly before `0`).

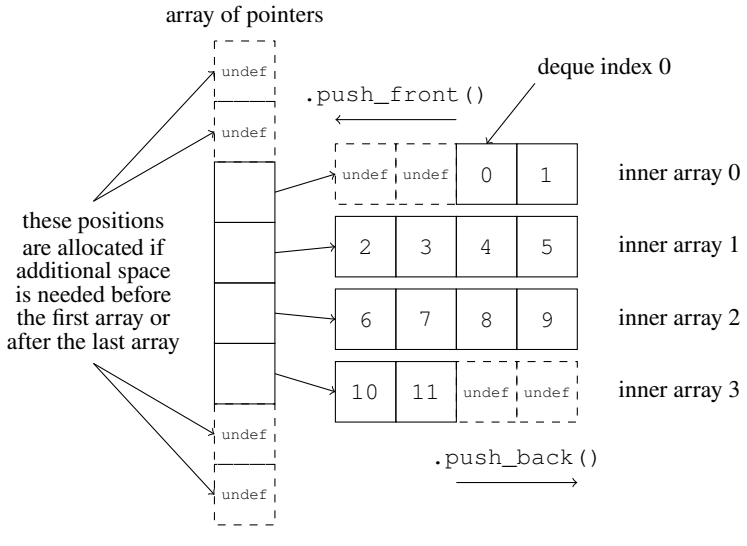
[3] The implementation of a deque may be different across libraries, as long as the implementation adheres to the C++ standards for a deque. The implementation covered in this section is used by the GCC standard library implementation.

If either the inner array at the beginning or end is fully filled, a new inner array is allocated at the next available position of the outer array. For instance, the deque would look like this if `12`, `13`, and `14` were pushed to the back:



If the entire outer array fills up (i.e., there is no space to allocate new inner arrays at either end), then the outer array is reallocated to a larger capacity. This is very similar to how a vector behaves when its underlying array fills up. However, unlike a vector, the additional space from reallocation is split between both ends of the deque's outer array. This allows insertion to be a constant time operation on both ends of the deque.

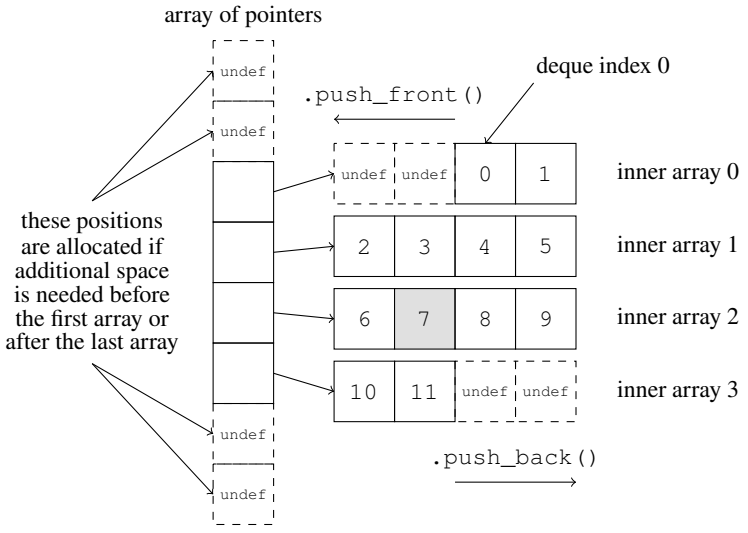
This implementation of a deque solves both of the problems we had when using a circular array and a linked list. Pointer invalidation is no longer an issue when inserting or deleting elements from either end of the container: since the inner arrays do not get reallocated, elements in the deque stay at the same memory location throughout its lifetime (assuming that they do not get deleted or moved). This layout in memory also allows the deque to support $\Theta(1)$ random access, which was not possible with a linked list. Given an index, we can quickly find the corresponding element in the deque using some simple arithmetic. For instance, suppose we wanted to retrieve the element at index 7 of the following deque:



To accomplish this, we first add the index we want (in this case, 7) to the number of unused slots in the first inner array: **7 + 2 = 9**. Then,

- Dividing this result by the capacity of each inner array gives us the array that the target value is in.
  - In this case, we get **9 / 4 = 2** using integer division (decimals are truncated). This means the element at index 7 is located within inner array 2.
- Taking the modulo of the original result with the capacity of the inner array gives us the position of the element within its inner array.
  - In this case, we get **9 % 4 = 1**. This means the element at index 7 is located at index 1 of its inner array.

Thus, the element at index 7 of the provided deque can be found at index 1 of inner array 2 (using zero indexing):



The STL provides an implementation of a deque for you, defined as a `std::deque<>`. To use a `std::deque<>`, you will need to include the `<deque>` library at the beginning of your code. A summary of deque operations is shown below:

```
template <typename T>
std::deque<T>();
```
Default constructor for deque that holds elements of type `T`, creates an empty deque without any elements.

```
// initializes an empty deque of integers
std::deque<int32_t> d1;
```

```
template <typename T>
std::deque<T>(std::initializer_list<T> init);
```
Initializes the deque with the contents of the initializer list.

```
// initializes d2 and d3 to have contents of {1, 2, 3}
std::deque<int32_t> d2{1, 2, 3};
std::deque<int32_t> d3 = {1, 2, 3};
```

```
template <typename T>
std::deque<T>(const std::deque<T>& other);
```
Copy constructor, copies the contents of `other` into the constructed deque.

```
// initializes d4 with the contents of d3, or {1, 2, 3}
std::deque<int32_t> d4{d3};
```

```
template <typename T>
std::deque<T>(size_t sz);
```
Creates a deque of `sz` elements, where each element is value initialized; size equal to `sz`.

```
// initializes d5 as a deque with size 5, each element initialized to 0
std::deque<int32_t> d5(5);
```

```
template <typename T>
std::deque<T>(size_t sz, T& val);
```
Creates a deque of `sz` elements, where each element is initialized to a value of `val`; size equal to `sz`.

```
// initializes d6 as a deque with size 5, each element initialized to 1
std::deque<int32_t> d6(5, 1);
```

```
template <typename T, typename InputIterator>
std::deque<T>(InputIterator begin_iter, InputIterator end_iter);
```
Creates a deque with all elements in the iterator range `[begin_iter, end_iter)` — inclusive begin but exclusive end. Both `begin_iter` and `end_iter` are input iterators.

```
// initializes d7 with the first two elements of d3, or {1, 2}
std::deque<int32_t> d7(d3.begin(), d3.begin() + 2);
```

```
template <typename T>
size_t std::deque<T>::size();
```
Returns the number of elements in the deque.

```
template <typename T>
bool std::deque<T>::empty();
```
Returns whether the deque is empty.

```
template <typename T>
T& std::deque<T>::front()
```
Returns a reference to the first element (undefined behavior if deque is empty).

```
template <typename T>
T& std::deque<T>::back()
```
Returns a reference to the last element (undefined behavior if deque is empty).

```
template <typename T>
void std::deque<T>::push_back(const T& val);
```
Pushes `val` to the back of the deque, increasing size by 1.

```
template <typename T, typename... Args>
T& std::deque<T>::emplace_back(Args&&... args);
```
Inserts a new element at the back of the deque, right after the current last element. The new element is constructed in place using arguments for its constructor (`args`). Returns a reference to the emplaced element since C++17.

```
template <typename T>
void std::deque<T>::pop_back();
```
Removes the last element in the deque, reducing size by 1 (undefined behavior if deque is empty).

```
template <typename T>
void std::deque<T>::push_front(const T& val);
```
Pushes `val` to the front of the deque, increasing size by 1.

```
template <typename T, typename... Args>
T& std::deque<T>::emplace_front(Args&&... args);
```
Inserts a new element at the front of the deque, right before the current first element. The new element is constructed in place using arguments for its constructor (`args`). Returns a reference to the emplaced element since C++17.

```
template <typename T>
void std::deque<T>::pop_front();
```
Removes the first element in the deque, reducing size by 1 (undefined behavior if deque is empty).

```
template <typename T>
iterator std::deque<T>::insert(iterator position, const T& val);
```
Inserts `val` directly before the element pointed to by the iterator `pos` and returns an iterator to the newly added element.

```
template <typename T>
iterator std::deque<T>::insert(iterator position, size_t n, const T& val);
```
Inserts `n` copies of `val` directly before the element pointed to by the iterator `pos` and returns an iterator to the first new element added.

```
template <typename T, typename InputIterator>
iterator std::deque<T>::insert(iterator position, InputIterator first, InputIterator last);
```
Inserts a copy of all elements in the iterator range `[first, last)` directly before the iterator `pos` and returns an iterator to the first new element added.

```
template <typename T>
iterator std::deque<T>::insert(iterator position, std::initializer_list<T> init);
```
Inserts the elements in the initializer list into the deque directly before the iterator `pos` and returns an iterator to the first new element added.

```
template <typename T, typename... Args>
iterator std::deque<T>::emplace(const_iterator pos, Args&&... args);
```
Inserts a new element directly before the element at position `pos`. The new element is constructed in place using arguments for its constructor (`args`). An iterator pointing to the emplaced object is returned.

```
template <typename T>
iterator std::deque<T>::erase(iterator pos);
```
Erases the element pointed to by the iterator `pos` and returns an iterator to the element following the one that was erased.

```
template <typename T>
iterator std::deque<T>::erase(iterator first, iterator last);
```
Erases all elements in the iterator range `[first, last)` and returns an iterator to the element following the last element that was erased.

```
template <typename T>
void std::deque<T>::clear();
```
Removes all elements in the container, leaving it with a size of 0.

```
template <typename T>
T& std::deque<T>::operator[](size_t n);
```
Returns a reference to the element at index `n` of the deque.

The iterator operations for a deque are similar to those of a vector, where `.begin()` returns an iterator to the beginning of the deque, and `.end()` returns an iterator one past the end of the deque. Deques also support reverse and constant iterators, which can be retrieved using `.rbegin()`, `.rend()`, `.cbegin()`, and `.cend()`.

Function	Behavior
`.begin()`	Returns a random access iterator to the first element in the deque
`.end()`	Returns a random access iterator to the position one past the last element in the deque
`.cbegin()`	Returns a *constant* random access iterator to the first element in the deque
`.cend()`	Returns a *constant* random access iterator to the position one past the last element in the deque
`.rbegin()`	Returns a *reverse* iterator to the last element in the deque
`.rend()`	Returns a *reverse* iterator to the position one before the first element in the deque
`.crbegin()`	Returns a constant reverse iterator to the last element in the deque
`.crend()`	Returns a constant reverse iterator to the position one before the first element in the deque

Deques are similar to vectors in that insertions and deletions from the middle of the container require elements to be shifted after the modification point (to open up a space for insertion, or to bridge a gap for deletion). However, in contrast to a vector, a deque's reallocation may exhibit better overall performance. In a vector, reallocation forces all data in the original array to be copied to a new, larger array. For a deque, reallocation is only done on the outer array of pointers, so only the contents of this outer array are copied during reallocation. This is because the actual data resides in the inner arrays, which have fixed capacity and do not need to be reallocated.

However, even if reallocation may be more efficient for a deque, `operator[]` is faster for a vector. Despite the fact that `operator[]` is a $\Theta(1)$ operation for both vectors and deques, the constant term for a deque is larger. This is because a vector only needs to perform a simple addition to access a value at any index (i.e., `arr[i] == *(arr + i)`), but deques require a bit more math to get the memory address of the element we want. In addition, elements in a deque are not always contiguous in memory (as shown by the illustrations in this section). As a result, iteration through a deque is typically slower than iteration through a vector. This is because vectors are better at exploiting caching: a phenomenon that allows elements in a sequence to be accessed faster if they are closer together in memory.

9.7 Container Adaptors

Vectors, lists, and deques fall into a category of containers known as *sequence containers*, whose data can be sequentially accessed in an established order. On the other hand, stacks and queues (and priority queues, which will be covered in the next chapter) belong to a class of containers known as **container adaptors**. Container adaptors are not full container classes on their own, but rather interfaces that are adapted for specific needs by modifying or restricting the functionality of a standard container. In the STL, container adaptors do not support iteration.

At the beginning of this chapter, we looked at how stacks and queues can be implemented using arrays and linked lists. However, by default, the STL implementations of stacks and queues use neither approach. Instead, they are implemented using a deque as the underlying container. When you initialize a `std::stack<>` in your program, you are actually getting a `std::deque<>` that limits data access to only one end. Similarly, if you initialize a `std::queue<>`, you are actually getting a `std::deque<>` that enforces insertion and deletion on different ends. In other words, both the STL stack and queue are built upon an underlying deque that is "adapted" to support only the operations that suit the interface of the desired container.

Thus, if you want to use a stack or a queue in your program, there is functionally no difference between initializing a stack or queue and initializing a deque, as a deque can be used to emulate the functionality of both stacks and queues. If you want a stack, you can initialize a deque and only use `.push_back()` and `.pop_back()`. If you want a queue, you can initialize a deque and only use `.push_back()` and `.pop_front()`. However, it is generally good practice to explicitly declare the container type that matches your needs (i.e., either a stack or a queue, rather than a deque), as it improves the readability of code and reduces the likelihood of mistakes (such as inserting or removing data from the wrong end); a deque should only be chosen if you need double-ended queue behavior that is not supported by a regular stack or queue.

A deque is not the only underlying container that can be used. You can also initialize a stack or a queue using a different underlying container, such as a vector or a list. The underlying container of a stack must support `.push_back()` and `.pop_back()`, while the underlying container of a queue must support `.push_back()` and `.pop_front()`. A table of possible underlying containers is shown below:

	STL Stack (`std::stack<>`)	STL Queue (`std::queue<>`)
Default Underlying Container	`std::deque<>` that only permits `.push_back()` and `.pop_back()`	`std::deque<>` that only permits `.push_back()` and `.pop_front()`
Optional Underlying Container (must be explicitly specified)	`std::list<>` that only permits `.push_back()` and `.pop_back()` **— OR —** `std::vector<>` that only permits `.push_back()` and `.pop_back()`	`std::list<>` that only permits `.push_back()` and `.pop_front()`

Notice that a `std::queue<>` cannot be initialized with a `std::vector<>` as the underlying container. This is because vectors do not support `.pop_front()` (and thus cannot provide efficient insertions/removals at different ends of the container).

To explicitly specify the underlying container for a stack or a queue, you will need to include the container type directly after the type of the data. For example, the following line of code initializes a stack that uses a vector as its underlying container:

```
std::stack<int32_t, std::vector<int32_t>> s;
```

Similarly, the following code initializes a queue that uses a list as its underlying container:

```
std::queue<int32_t, std::list<int32_t>> q;
```

If you do not explicitly specify the underlying container, a `std::deque<>` will be used by default. In this class, there is really no reason to ever use anything other than a deque as the underlying container of a stack or a queue. For stacks and queues, a deque is the ideal underlying container for most use cases — hence why the STL uses it as the default!