Ordered Data Structures: Grids, Queues, and Stacks

What’s an example of “ordered data” that you’ve encountered in your life?
(Also grab a mask from the back desk if you don’t have one!)

https://pollev.com/cs106bpoll
Examples of ordered data
Roadmap

C++ basics
User/client

vectors + grids
stacks + queues
sets + maps

Core Tools

testing

Object-Oriented Programming

Midterm

arrays
dynamic memory management
linked data structures

Implementation

real-world algorithms
recursive
problem-solving

Life after CS106B!
Roadmap

C++ basics

User/client

- vectors + grids
- stacks + queues
- sets + maps

Implementation

Life after CS106B!
Roadmap

C++ basics

User/client

- vectors + grids
- stacks + queues
- sets + maps

Implementation

Life after CS106B!
“Map” of the “container store”

- vectors + grids
- stacks + queues
- sets + maps
“Map” of the “container store”

- Ordered data
  - Vectors + grids
  - Stacks + queues
  - Sets + maps
“Map” of the “container store”

Ordered data

- vectors + grids
- stacks + queues

Unordered data

sets + maps
“Map” of the “container store”

Ordered data

vectors + grids

stacks + queues

sets + maps
Today’s question

When is it appropriate to use different types of ordered data structures?
Today’s topics

1. Review
2. Grids
   [2.5 GridLocation + structs]
3. Queues
4. Stacks
Review
(vectors and pass-by-reference)
Containers, or Abstract Data Types, or Data Structures

- Containers are powerful abstractions that allow programmers to store data in predictable, organized ways.

- As the user, you get certain guarantees about the functionality of the container & the properties of the data inside that specific container.

- You can use ADTs **without understanding the underlying implementation**!
  - That’s abstraction!
Note: while we specifically use ADTs from the Stanford C++ libraries, these principles transcend language boundaries.
Vectors
Our first ADT: **Vectors**

- At a high level, a vector is an ordered collection of elements of the same type that can grow and shrink in size.

- Each element in the vector has a specific location, or index.
  - 0, 1, 2 ...

- All elements in a vector must be of the same type.

- Vectors are flexible when it comes to the number of elements they can store. You can easily add and remove elements, and vectors also know their current size.
## Stanford “Vector” vs STL “vector”

<table>
<thead>
<tr>
<th>What you want to do</th>
<th>Stanford <code>Vector&lt;int&gt;</code></th>
<th><code>std::vector&lt;int&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a new, empty vector</td>
<td><code>Vector&lt;int&gt; vec;</code></td>
<td><code>std::vector&lt;int&gt; vec;</code></td>
</tr>
<tr>
<td>Create a vector with n copies of 0</td>
<td><code>Vector&lt;int&gt; vec(n);</code></td>
<td><code>std::vector&lt;int&gt; vec(n);</code></td>
</tr>
<tr>
<td>Create a vector with n copies of a value k</td>
<td><code>Vector&lt;int&gt; vec(n, k);</code></td>
<td><code>std::vector&lt;int&gt; vec(n, k);</code></td>
</tr>
<tr>
<td>Add a value k to the end of a vector</td>
<td><code>vec.add(k);</code></td>
<td><code>vec.push_back(k);</code></td>
</tr>
<tr>
<td>Remove all elements of a vector</td>
<td><code>vec.clear();</code></td>
<td><code>vec.clear();</code></td>
</tr>
<tr>
<td>Get the element at index i</td>
<td><code>int k = vec[i];</code></td>
<td><code>int k = vec[i];</code></td>
</tr>
<tr>
<td>Check size of vector</td>
<td><code>vec.size();</code></td>
<td><code>vec.size();</code></td>
</tr>
<tr>
<td>Loop through vector by index i</td>
<td><code>for (int i = 0; i &lt; vec.size(); ++i) ...</code></td>
<td><code>for (std::size_t i = 0; i &lt; vec.size(); ++i) ...</code></td>
</tr>
<tr>
<td>Replace the element at index i</td>
<td><code>vec[i] = k;</code></td>
<td><code>vec[i] = k;</code></td>
</tr>
</tbody>
</table>

Credit: [CS106L](#)
What exactly is a reference?

- References look like this:

References have names and types, just like regular variables.

The type has an ampersand (&) after it to indicate it is a reference to that data type rather than the type itself.
Types you know in C++

- int
- char
- double
- string
- bool
- long
- Vector<int>
- Vector<char>

...
When we use references

- To allow helper functions to edit data structures in other functions
  - But why don’t we just return a copy of the data structure?

- To avoid making new copies of large data structures in memory
  - Passing data structures by reference makes your code more efficient!

- References also provide a workaround for **multiple return values**
  - Your function can take in multiple pieces of information by reference and modify them all. In this way you can "return" both a modified Vector and some auxiliary piece of information about how the structure was modified. This makes it as if your function is returning two updated pieces of information to the function that called it!
Definition

pass by value

When a parameter is passed into a function, the new variable stores a copy of the passed in value in memory

“Pass in a copy”
Definition

**pass by reference**

When a parameter is passed into a function, the new variable stores a *reference* to the passed in value, which allows you to directly edit the original value.

“Pass in the original under a different name”
In C++...

- By default, parameters are passed by value.
- You can choose to pass by reference in C++ by using references.
In C++...
- By default, parameters are passed by value.
- You can choose to pass by reference in C++ by using references.

In Python or Java?
In C++...

- By default, parameters are **passed by value**.
- You can **choose** to **pass by reference** in C++ by using references.

In Python or Java?

Seems like a straightforward question!
Parameters are passed by value by default. You can choose to pass by reference in C++ by using references.

In C++...

In Python or Java? Seems like a simple, straightforward question! WRONG.

“Java is only pass by value”

“Java passes references by value”

“Java object variables are simply references”
In C++...

- By default, parameters are **passed by value**.
- You can **choose to pass by reference** in C++ by using references.

In Python or Java?

Because of the way the languages are designed, **pass-by-value** and **pass-by-reference** mean slightly different things in C++ vs. Python vs. Java.
In C++...

- By default, parameters are **passed by value**.
- You can **choose** to **pass by reference** in C++ by using references.

In Python or Java?

By default, Python and Java treat some things as pass by value, others as pass by reference.
In C++...

- By default, parameters are **passed by value**.
- You can **choose** to **pass by reference** in C++ by using references.
- You **should** pass by value for primitives (int, string)*
- You **should** pass by reference for large data structures*

*in general

In Python or Java?

By default, Python and Java treat some things as pass by value, others as pass by reference.
Trace problem

[5-minute Ed workspace!]
Grids
What is a grid?

- A 2D array, defined with a particular width and height
What is a grid?

- A 2D array, defined with a particular width and height

We say array instead of vector here because the dimensions are established when the grid is created (but vectors can change their sizes).
What is a grid?

- A 2D array, defined with a particular width and height
- Useful for spreadsheets, game boards, etc.
What is a grid?

- A 2D array, defined with a particular width and height
- Useful for spreadsheets, game boards, etc.
- Three ways to declare a grid
  - `Grid<type> gridName;`
  - `Grid<type> gridName(numRows, numCols);`
  - `Grid<type> gridName = {{r0c0, r0c1, r0c2}, {r1c0, r1c1, r1c2},...};`
What is a grid?

- A 2D array, defined with a particular width and height
- Useful for spreadsheets, game boards, etc.

- Three ways to declare a grid
  - `Grid<type> gridName;`
  - `Grid<int> board;` 
  - `board.resize(3, 3);`
What is a grid?

- A 2D array, defined with a particular width and height
- Useful for spreadsheets, game boards, etc.

Three ways to declare a grid

- `Grid<type> gridName;`
- `Grid<int> board;`  
  `board.resize(3, 3);`

If you declare a board with no initialization, you must resize it or reassign it before using it. Resizing will fill it with default values for that type.
What is a grid?

- A 2D array, defined with a particular width and height
- Useful for spreadsheets, game boards, etc.
- Three ways to declare a grid
  - `Grid<type> gridName;`
  - `Grid<int> board;`  
    `board.resize(3, 3);`  
    `board[0][0] = 2;`  
    `board[1][0] = 6;`
What is a grid?

- A 2D array, defined with a particular width and height
- Useful for spreadsheets, game boards, etc.
- Three ways to declare a grid
  - `Grid<type> gridName(numRows, numCols);
  
  ```
  Grid<int> board(3, 3);
  board[0][0] = 2;
  board[1][0] = 6;
  ```
What is a grid?

- A 2D array, defined with a particular width and height
- Useful for spreadsheets, game boards, etc.
- Three ways to declare a grid
  - `Grid<type> gridName = {{r0c0, r0c1, r0c2}, {r1c0, r1c1, r1c2},...};`
  - `Grid<int> board = {{2,0,1}, {6,0,2}, {5,4,3}};`
What is a grid?

- A 2D array, defined with a particular width and height
- Useful for spreadsheets, game boards, etc.

- Three ways to declare a grid
  - `Grid<type> gridName;`
  - `Grid<type> gridName(numRows, numCols);`
  - `Grid<type> gridName = {{r0c0, r0c1, r0c2}, {r1c0, r1c1, r1c2},...};`

- Jenny, why can’t we use a combination of Vectors to simulate a 2D matrix?
What is a grid?

- A 2D array, defined with a particular width and height
- Useful for spreadsheets, game boards, etc.
- Three ways to declare a grid
  - `Grid<type> gridName;`
  - `Grid<type> gridName(numRows, numCols);`
  - `Grid<type> gridName = {{r0c0, r0c1, r0c2}, {r1c0, r1c1, r1c2},...};`
- Jenny, why can’t we use a combination of Vectors to simulate a 2D matrix?
  - You can! But a Grid is easier!
Grid methods

- The following methods are part of the grid collection and can be useful:
  - `grid.numRows()`: Returns the number of rows in the grid.
  - `grid.numCols()`: Returns the number of columns in the grid.
  - `grid[i][j]`: selects the element in the $i$th row and $j$th column.
  - `grid.resize(rows, cols)`: Changes the dimensions of the grid and re-initializes all entries to their default values.
  - `grid.inBounds(row, col)`: Returns `true` if the specified row, column position is in the grid, `false` otherwise.

- For the exhaustive list, check out the [Stanford Grid documentation](#).
Grid methods

- The following methods are part of the grid collection and can be useful:
  - `grid.numRows()`: Returns the number of rows in the grid.
  - `grid.numCols()`: Returns the number of columns in the grid.
  - `grid[i][j]`: selects the element in the $i$th row and $j$th column.
  - `grid.resize(rows, cols)`: Changes the dimensions of the grid and re-initializes all entries to their default values.
  - `grid.inBounds(row, col)`: Returns `true` if the specified row, column position is in the grid, `false` otherwise.

- For the exhaustive list, check out the [Stanford Grid documentation](https://www.stanford.edu).
How to traverse a Grid

```cpp
void printGrid(Grid<char>& grid) {
    for(int r = 0; r < grid.numRows(); r++) {
        for(int c = 0; c < grid.numCols(); c++) {
            cout << grid[r][c];
        }
        cout << endl;
    }
}
```
void printGrid(Grid<char>& grid) {
    for(int r = 0; r < grid.numRows(); r++) {
        for(int c = 0; c < grid.numCols(); c++) {
            cout << grid[r][c];
        }
        cout << endl;
    }
}

What is the output of this function called on the provided grid going to be?

pollev.com/cs106bpoll

A. yeehaw
B. yea
ehw
C. ye
eh
aw
D. None of the above
What is the output of printGrid going to be for this provided grid?

A
B
C
D
None of the above
How to traverse a Grid

```cpp
void printGrid(Grid<char>& grid) {
    for (int r = 0; r < grid.numRows(); r++) {
        for (int c = 0; c < grid.numCols(); c++) {
            cout << grid[r][c];
        }
        cout << endl;
    }
}
```

Output:

```
y
'e'
'e'
'h'
'a'
'w'
```
How to traverse a Grid

```cpp
void printGrid(Grid<char>& grid) {
    for(int r = 0; r < grid.numRows(); r++) {
        for(int c = 0; c < grid.numCols(); c++) {
            cout << grid[r][c];
        }
        cout << endl;
    }
}
```

Output:

Y
How to traverse a Grid

```cpp
void printGrid(Grid<char>& grid) {
    for(int r = 0; r < grid.numRows(); r++) {
        for(int c = 0; c < grid.numCols(); c++) {
            cout << grid[r][c];
        }
        cout << endl;
    }
}
```

Output:
```
ye
```
How to traverse a Grid

```cpp
void printGrid(Grid<char>& grid) {
    for(int r = 0; r < grid.numRows(); r++) {
        for(int c = 0; c < grid.numCols(); c++) {
            cout << grid[r][c];
        }
        cout << endl;
    }
}
```

Variables:
- \( r = 1 \)
- \( c = 0 \)

Output:
```
ye
 e
```

The grid is:
```
<table>
<thead>
<tr>
<th>y</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>h</td>
</tr>
<tr>
<td>a</td>
<td>w</td>
</tr>
</tbody>
</table>
```
How to traverse a Grid

```cpp
void printGrid(Grid<char>& grid) {
    for(int r = 0; r < grid.numRows(); r++) {
        for(int c = 0; c < grid.numCols(); c++) {
            cout << grid[r][c];
        }
        cout << endl;
    }
}
```

Variables:
- $r = 1$
- $c = 1$

Output:
```
ye
eh```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>'y'</td>
<td>'e'</td>
</tr>
<tr>
<td>'e'</td>
<td>'h'</td>
</tr>
<tr>
<td>'a'</td>
<td>'w'</td>
</tr>
</tbody>
</table>
How to traverse a Grid

```cpp
void printGrid(Grid<char>& grid) {
    for(int r = 0; r < grid.numRows(); r++) {
        for(int c = 0; c < grid.numCols(); c++) {
            cout << grid[r][c];
        }
        cout << endl;
    }
}
```

Variables:
- \( r = 2 \)
- \( c = 0 \)

Output:
- ye
- eh
- a
How to traverse a Grid

```cpp
void printGrid(Grid<char>& grid) {
    for(int r = 0; r < grid.numRows(); r++) {
        for(int c = 0; c < grid.numCols(); c++) {
            cout << grid[r][c];
        }
        cout << endl;
    }
}
```

Variables:
- r = 2
- c = 1

Output:
- ye
- eh
- aw
Common pitfalls when using Grids

- Don’t forget to specify what data type is stored in your grid

  NO: `Grid board;`
  YES: `Grid<char> board;`

- Like Vectors and other ADTs, Grids should be passed by reference when used as function parameters

- Watch your variable ordering with Grid indices! Rather than using `i` and `j` as indices to loop through a grid, it’s better to use `r` for rows and `c` for columns.
  - `[r][c]`

- Unlike in other languages, you can only access cells (not individual rows).
  `grid[0]` → doing this will cause an error!
Battleship uses a grid!
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Battleship uses a grid!
Battleship uses a grid!

What if we want to keep track of all cells where a ship is present?
Battleship uses a grid!

Jenny, wouldn’t it be nice to label each grid location as an index like we did with vectors?
Structs +
GridLocation
Definition

**struct**
A way to bundle different types of information in C++ – like creating a custom data structure.
The **GridLocation** struct

- A pre-defined struct in the Stanford C++ libraries that makes it more convenient to store Grid locations:

```cpp
struct GridLocation {
    int row;
    int col;
}
```
The **GridLocation** struct

- A pre-defined struct in the Stanford C++ libraries that makes it more convenient to store Grid locations:

```cpp
struct GridLocation {
    int row;
    int col;
}
```

**struct members**
(these can be different types)
The **GridLocation** struct

- A pre-defined struct in the Stanford C++ libraries that makes it more convenient to store Grid locations

- To declare a struct, you can either assign each of its members separately or assign it when it’s created:

```cpp
struct GridLocation {
    int row;
    int col;
}
```

```cpp
GridLocation origin = {0, 0};
GridLocation origin;
origin.row = 0;
origin.col = 0;
```
The **GridLocation** struct

- A pre-defined struct in the Stanford C++ libraries that makes it more convenient to store Grid locations.

- To declare a struct, you can either assign each of its members separately or assign it when it’s created:

  ```cpp
  GridLocation origin = {0, 0};
  GridLocation origin;
  origin.row = 0;
  origin.col = 0;
  ```

You can access members in a struct using the dot notation (no parentheses after the member name!)

```cpp
struct GridLocation {
  int row;
  int col;
}
```
Vector<GridLocation> shipCells;
GridLocation smallLeft = {0, 5};
GridLocation smallRight = {0, 6};
shipCells.add(smallLeft);
shipCells.add(smallRight);

VS.

Vector<int> rowIndices;
Vector<int> colIndices;
rowIndices.add(0);
rowIndices.add(0);
colIndices.add(5);
colIndices.add(6);

As an exercise on your own: Think about how you would answer the question “Is there a ship at (4, 3)?” for each of the different representations (with and without GridLocation structs).
The **GridLocation** struct

- A pre-defined struct in the Stanford C++ libraries that makes it more convenient to store Grid locations

- To declare a struct, you can either assign each of its members separately or assign it when it’s created:

```
GridLocation origin = {0, 0};  // or
GridLocation origin;           // then
origin.row = 0;
origin.col = 0;
```

**NOTE:** Grids are not made up of GridLocation structs! GridLocations are just a convenient way to store an index (single cell or a path of cells) within a Grid.
The **GridLocation** struct

- A pre-defined struct in the Stanford C++ libraries that makes it more convenient to store Grid locations
- To declare a struct, you can either assign each of its members separately or assign it when it’s created:

```cpp
struct GridLocation {
    int row;
    int col;
}
```

GridLocation origin = {0, 0};
GridLocation origin;
origin.row = 0;
origin.col = 0;

*We can use a GridLocation to access a particular cell in the grid: grid[origin]*
Announcements

● Assignment 1 is due Friday, July 1, at 11:59pm PDT.
  ○ If you didn’t get to attend YEAH on Monday, we highlight recommend watching the recorded session for getting started (will be posted soon)!
  ○ We also recorded an extra video on SimpleTest!

● Sections start tomorrow! Go to cs198.stanford.edu to double-check your assigned time.
  ○ If you missed the signup deadline by Sunday, please go to the CS198 website to manually sign up for a section with an available slot.

● Assignment 2 will be released by the end of the day on Friday.
Queues
What is a queue?

- Like a real queue/line!

- First person In is the First person Out (FIFO)
  - When you remove (dequeue) people from the queue, you remove them from the front of the line.

- Last person in is the last person served
  - When you insert (enqueue) people into a queue, you insert them at the back (the end of the line)
Queue methods

- A queue must implement at least the following functions:
  - `enqueue(value)` - place an entity onto the back of the queue
  - `dequeue()` - remove an entity from the front of the queue and return it
  - `peek()` - look at the entity at the front of the queue, but don’t remove it
  - `isEmpty()` - a boolean value, true if the queue is empty, false if it has at least one element.
    - note: if you try to dequeue() or peek() an empty queue, you will get a runtime error
- For the exhaustive list, check out the [Stanford Queue documentation](https://cs206.soe.ucsc.edu/).
Queue example

Queue<int> line; // {}, empty queue
line.enqueue(42); // {42}
line.enqueue(-3); // {42, -3}
line.enqueue(17); // {42, -3, 17}
cout << line.dequeue() << endl; // 42 (Line is {-3, 17})
cout << line.peek() << endl; // -3 (Line is {-3, 17})
cout << line.dequeue() << endl; // -3 (Line is {17})

// You can also create a queue using:
Queue<int> line = {42, -3, 17};
Stacks
What is a stack?

- Modeled like an actual stack (of pancakes)
- Only the top element in a stack is accessible.
  - The **Last item In** is the **First one Out**. (LIFO)
- The push, pop, and top operations are the only operations allowed by the stack ADT.
The **Last item In** is the **First one Out.** (LIFO)
Stack methods

- A stack is an abstract data type with the following behaviors/functions:
  - `push(value)` - place an entity onto the top of the stack
  - `pop()` - remove an entity from the top of the stack and return it
  - `peek()` - look at the entity at the top of the stack, but don’t remove it
  - `isEmpty()` - a boolean value, true if the stack is empty, false if it has at least one element. (Note: a runtime error occurs if a `pop()` or `peek()` operation is attempted on an empty stack.)
- For the exhaustive list, check out the Stanford Stack documentation.
Stack example

Stack<string> wordStack; // {}, empty stack
wordStack.push(“Kylie”);  // {“Kylie”}
wordStack.push(“Jenny”);  // {“Kylie”, “Jenny”}
wordStack.push(“Trip”);   // {“Kylie”, “Jenny”, “Trip”}
cout << wordStack.pop() << endl; // “Trip”
cout << wordStack.peek() << endl;  // “Jenny”
cout << wordStack.pop() << endl;   // “Jenny” (stack is {“Kylie”})

// You can also create a stack using:
Stack<string> wordStack = {“Kylie”, “Jenny”, “Trip”}; // the “top” is the rightmost element
The **First item In is the First one Out.**

(FIFO)

The **Last item In is the First one Out.**

(LIFO)

Queue

Stack
The **First item In** is the **First one Out.** (FIFO)

Queue
The **First item In is the First one Out.** (FIFO)

Queue
The **First item In** is the **First one Out**.

(FIFO)
The **First** item **In** is the **First** one **Out**. (FIFO)

Queue
The **First item In** is the **First one Out**.

*(FIFO)*

Queue
The **First item In** is the **First one Out.**

(FIFO)

Queue
The Last item In is the First one Out.
(LIFO)

Stack
The Last item In is the First one Out.
(LIFO)
The Last item In is the First one Out. (LIFO)

Stack
The **Last item In** is the **First one Out**. (LIFO)

**Stack**
The **Last item In** is the **First one Out**. (LIFO)

Stack
The Last item In is the First one Out.
(LIFO)
The **Last item In** is the **First one Out**.

(LIFO)
The Last item In is the First one Out. (LIFO)
Tradeoffs with queues and stacks (vs. other ADTs)

● What are some downsides to using a queue/stack?
  ○ No random access. You get the front/top, or nothing.
  ○ No side-effect-free traversal — you can only iterate over all elements in the structure by removing previous elements first.
  ○ No easy way to search through a queue/stack.

● What are some benefits?
  ○ Useful for lots of problems – many real-world problems can be solved with either a LIFO or FIFO model
  ○ Very easy to build one from an array such that access is guaranteed to be fast. (We'll talk more about arrays later in the quarter, and we'll talk about what "fast" access means later this week.)
    ■ Where would you have the top of the stack be if you build one using a Vector? Why would that be fast?
Queue + Stack patterns
Common patterns and pitfalls with stacks and queues

Idioms:

1. Emptying a stack/queue (both would empty one at a time)
Idiom 1: Emptying a queue/stack

Queue<int> queueIdiom1;

// produce: {1, 2, 3, 4, 5, 6}
for (int i = 1; i <= 6; i++) {
    queueIdiom1.enqueue(i);
}

while (!queueIdiom1.isEmpty()) {
    cout << queueIdiom1.dequeue() << " ";
}
cout << endl;

// prints: 1 2 3 4 5 6
Idiom 1: Emptying a queue/stack

Queue<int> queueIdiom1;

// produce: {1, 2, 3, 4, 5, 6}
for (int i = 1; i <= 6; i++) {
    queueIdiom1.enqueue(i);
}

while (!queueIdiom1.isEmpty()) {
    cout << queueIdiom1.dequeue() << " ";
}
cout << endl;

// prints: 1 2 3 4 5 6

Stack<int> stackIdiom1;

// produce: {1, 2, 3, 4, 5, 6}
for (int i = 1; i <= 6; i++) {
    stackIdiom1.push(i);
}

while (!stackIdiom1.isEmpty()) {
    cout << stackIdiom1.pop() << " ";
}
cout << endl;

// prints: 6 5 4 3 2 1
Common patterns and pitfalls with stacks and queues

Idioms:

1. Emptying a stack/queue
2. Iterating over and modifying a stack/queue ➔ only calculate the size once before looping
Idiom 2: Iterating over and modifying queue/stack

Queue<int> queueIdiom2 = {1, 2, 3, 4, 5, 6};

int origQSize = queueIdiom2.size();
for (int i = 0; i < origQSize; i++) {
    int value = queueIdiom2.dequeue();
    // re-enqueue even values
    if (value % 2 == 0) {
        queueIdiom2.enqueue(value);
    }
}
cout << queueIdiom2 << endl;

// prints: {2, 4, 6}
Idiom 2: Iterating over and modifying queue/stack

```cpp
Queue<int> queueIdiom2 = {1,2,3,4,5,6};
int origQSize = queueIdiom2.size();
for (int i = 0; i < origQSize; i++) {
    int value = queueIdiom2.dequeue();
    // re-enqueue even values
    if (value % 2 == 0) {
        queueIdiom2.enqueue(value);
    }
}
cout << queueIdiom2 << endl;
// prints: {2, 4, 6}
```

```cpp
Stack<int> stackIdiom2 = {1,2,3,4,5,6};
Stack<int> result;
int origSSize = stackIdiom2.size();
for (int i = 0; i < origSSize; i++) {
    int value = stackIdiom2.pop();
    // add even values to result
    if (value % 2 == 0) {
        result.push(value);
    }
}
cout << result << endl;
// prints: {6, 4, 2}
```
Common patterns and pitfalls with stacks and queues

Idioms:

1. Emptying a stack/queue
2. Iterating over and modifying a stack/queue ➞ only calculate the size once before looping

Common bugs:

- If you edit the ADT within a loop, don’t use `.size()` in the loop’s conditions! The size changes while the loop runs.
- Unlike with queues, you can’t iterate over a stack without destroying it ➞ think about when it might be beneficial to make a copy instead.
ADTs summary (so far)
Summary so far:

**Ordered data structures**

- Vectors (1D)
- Grids (2D)
- Stacks + Queues

Let's you access elements with indices

- **Vectors (1D)**
- **Grids (2D)**
  
- Easily able to search through all elements
- Can use the indices as a way of accessing specific cells
  - myVec[1]
  - myGrid[2][4]
Summary so far:

**Ordered data structures**

- **Vectors + Grids**
- **Stacks + Queues**

Let's you access to one element at a time (no indices)

**Queues (FIFO)**

**Stacks (LIFO)**

- Constrains the way you can insert and access data
  - Can only get top/first
  - No random access
- More efficient for solving specific LIFO/FIFO problems
### Ordered ADTs with accessible indices

**Types:**
- Vectors (1D)
- Grids (2D)

**Traits:**
- Easily able to search through all elements
- Can use the indices as a way of structuring the data

### Ordered ADTs where you can’t access elements by index

**Types:**
- Queues (FIFO)
- Stacks (LIFO)

**Traits:**
- Constrains the way you can insert and access data
- More efficient for solving specific LIFO/FIFO problems
Attendance ticket:

https://tinyurl.com/lec5cs106b

Please don’t send this link to students who are not here. It’s on your honor!
What ADT should we use?
For each of the tasks, pick which ADT is best suited for the task:

- The undo button in a text editor
- Jobs submitted to a printer that can also be cancelled
- LaIR requests
- Your browsing history
- Google spreadsheets
- Call centers (“your call will be handled by the next available agent”)
For each of the tasks, pick which ADT is best suited for the task:

- **Vectors**
  - The undo button in a text editor

- **Grids**
  - Jobs submitted to a printer that can also be cancelled
  - LaIR requests

- **Queues**
  - Your browsing history

- **Stacks**
  - Google spreadsheets
  - Call centers (“your call will be handled by the next available agent”)
For each of the tasks, pick which ADT is best suited for the task:

- The undo button in a text editor
- **Jobs submitted to a printer that can also be cancelled**
- LaIR requests
- Your browsing history
- Google spreadsheets
- Call centers (“your call will be handled by the next available agent”)
For each of the tasks, pick which ADT is best suited for the task:

- The undo button in a text editor
- Jobs submitted to a printer that can also be cancelled
- **LaIR requests**
- Your browsing history
- Google spreadsheets
- Call centers (“your call will be handled by the next available agent”)
For each of the tasks, pick which ADT is best suited for the task:

- The undo button in a text editor
- Jobs submitted to a printer that can also be cancelled

Vectors

- LaIR requests

Grids

- Your browsing history

Queues

- Google spreadsheets

Stacks

- Call centers (“your call will be handled by the next available agent”)

For each of the tasks, pick which ADT is best suited for the task:

- The undo button in a text editor

Vectors

- Jobs submitted to a printer that can also be cancelled

Grids

- LaIR requests

Queues

- Your browsing history

- Google spreadsheets

Stacks

- Call centers (“your call will be handled by the next available agent”)
For each of the tasks, pick which ADT is best suited for the task:

- The undo button in a text editor
- Jobs submitted to a printer that can also be cancelled
- LaIR requests
- Your browsing history
- Google spreadsheets
- Call centers ("your call will be handled by the next available agent")
What’s next?
Unordered ADTs: Sets and Maps
Nested data structures 😱