CS 106X, Lecture 5
Big-O, Stacks and Queues

reading:

Programming Abstractions in C++, Chapter 5.2-5.3, 10

Plan For Today

- Recap: Vectors
- Big-O
- Stacks
- Announcements
- Queues

Plan For Today

- Recap: Vectors
- Big-O
- Stacks
- Announcements
- Queues

Vectors (Lists)

```
#include "vector.h"
```

- **vector** (aka **list**): a collection of elements with 0-based **indexes**
  - like a dynamically-resizing array (Java `ArrayList` or Python `list`)
  - Include the type of elements in the <> brackets

```
// initialize a vector containing 5 integers
//          index   0   1   2   3   4
Vector<int> nums {42, 17, -6,  0, 28};

Vector<string> names;          // {}
names.add("Nick");           // {"Nick"}
names.add("Zach");         // {"Nick", "Zach"}
names.insert(0, "Ed");     // {"Ed", "Nick", "Zach"}
```

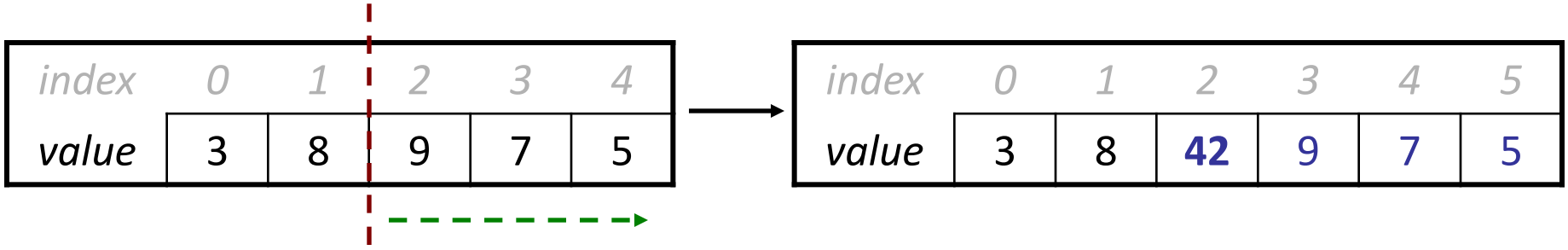
Vector members (5.1)

`v.add(value);` or `v += value;` or `v += v1, v2, ..., vN;`	appends value(s) at end of vector
`v.clear();`	removes all elements
`v[i]` or `v.get(i)`	returns the value at given index
`v.insert(i, value);`	inserts given value just before the given index, shifting subsequent values to the right
`v.isEmpty()`	returns `true` if the vector contains no elements
`v.remove(i);`	removes/returns value at given index, shifting subsequent values to the left
`v[i] = value;` or `v.set(i, value);`	replaces value at given index
`v.subList(start, length)`	returns new vector of sub-range of indexes
`v.size()`	returns the number of elements in vector
`v.toString()`	returns a string representation of the vector such as `"{3, 42, -7, 15}"`
`ostr << v`	prints v to given output stream (e.g. `cout << v`)

Vector insert/remove

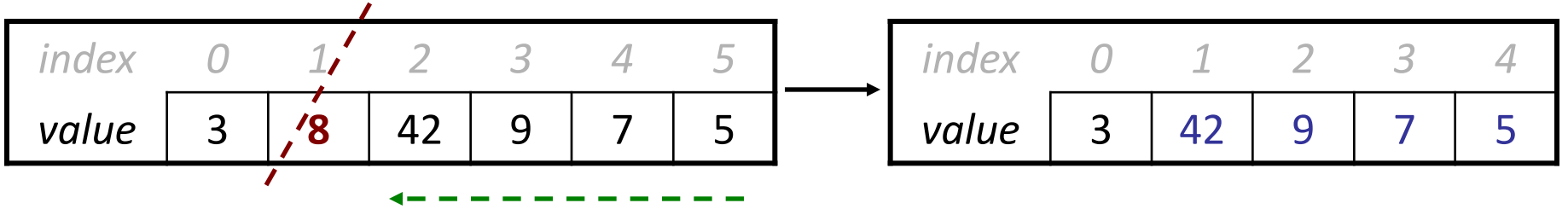
```
v.insert(2, 42);
```

- shift elements right to make room for the new element




```
v.remove(1);
```

- shift elements left to cover the space left by the removed element



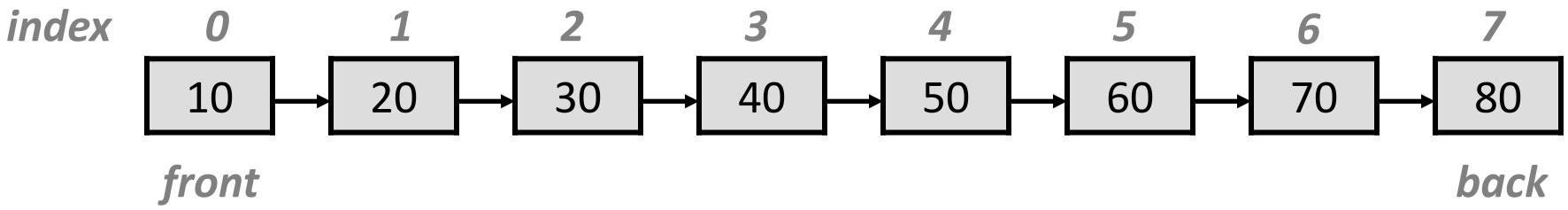
(These operations are slower the more elements they need to shift.)

LinkedList class

- Class **`LinkedList`** provides the same functionality as `Vector`.

```
LinkedList<int> list;
for (int i = 1; i <= 8; i++) {
    list.add(10 * i);   // {10, 20, 30, 40, 50, 60, 70, 80}
}
```

- **linked list:** Made of *nodes*, each storing a value and link to 'next' node.
  - Internally the list knows its **front** node only (sometimes **back** too), but can go to 'next' repeatedly to reach other nodes.



Plan For Today

- Recap: Vectors
- Big-O
- Stacks
- Announcements
- Queues

Algorithmic Efficiency

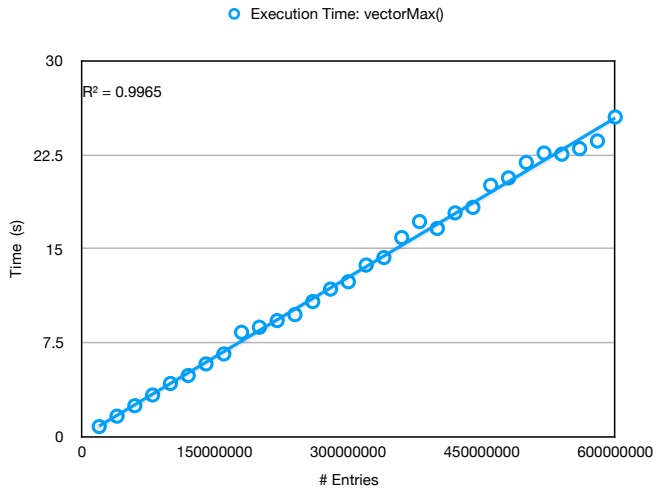
- We would like a way to measure how "fast" our code is.
  - **Seconds?** *This changes on each computer/chip!*
  - **Exact # Operations?** Hard to measure, maybe unnecessarily precise.
- To measure algorithmic efficiency, we need to narrow down why efficiency is important in the first place.
- What most significantly impacts an algorithm's speed? **Data size**
  - Most algorithms run fast when processing little data. But at Facebook, difference between 1 hr. and 1 day for machine learning matters!
  - Fun fact: Facebook scanned ~105TB of data per hour….6 years ago!
- Let's develop a way to measure an algorithm's speed that depends on the amount of data being processed.

Example: vectorMax

```
int vectorMax(const Vector<int>& v) {
    int currentMax = v[0];
    for (int element : v) {
        if (currentMax < element) {
            currentMax = element;
        }
    }

    return currentMax;
}
```

Exploring Runtime

- The runtime of the first implementation seems to be linearly proportional to the data size. So **2x more data means 2x longer to run.**



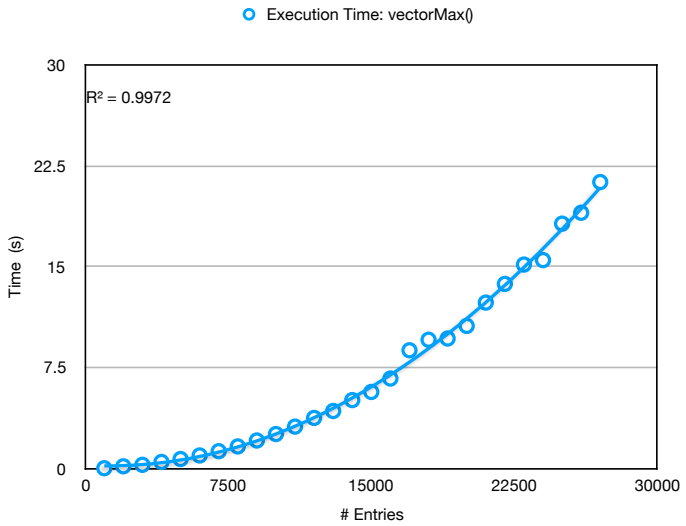

```
int vectorMax(const Vector<int>& v) {
    int currentMax = v[0];
    for (int element : v) {
        if (currentMax < element) {
            currentMax = element;
        }
    }

    return currentMax;
}
```

For each element, this loop will execute once. Doubling the data means that this loop will execute twice as many times.

We say this runtime is O(N): "on the order of N operations".

Exploring Runtime

- The runtime of the second implementation seems to be quadratically (^{2}) proportional to the data size. So **2x more data means 4x longer to run.**




```
int vectorMaxQuadratic(const Vector<int>& v) {
    Vector<int> copy(v);
    selectionSort(copy);
    return copy[copy.size() - 1];
}
```

https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Exploring Runtime

```
void selectionSort(Vector<int>& v) {
    for (int i = 0; i < v.size(); i++) {
        // walk across the array looking for the smallest value
        int smallestIndex = i;
        for (int j = i+1; j < v.size(); j++) {
            if (v[j] < v[smallestIndex]) {
                smallestIndex = j;
            }
        }
        // swap v[i] with v[smallestIndex]
        swap(v, i, smallestIndex);
    }
}
```

We say this runtime is $O(N^2)$.

For each element, the first loop will execute once. Additionally, for each element, *all inner loops* will execute once. Doubling the data means that the outer loop will execute twice as many times, *and* each inner loop will execute twice as many times, totaling 4x more time.

Exploring Runtime

```
int vectorMaxQuadratic(const Vector<int>& v) {
    Vector<int> copy(v);
    selectionSort(copy);

    // Paranoid check to see if it's correct
    int max = copy[copy.size() - 1];
    bool isCorrect = true;
    for (int i = 0; i < copy.size(); i++) {
        if (copy[i] > max) {
            isCorrect = false;
        }
    }

    if (isCorrect) {
        return max;
    } else {
        error("Internal error");
    }
}
```

We said this runtime is $O(N^2)$.

Due to this loop, this runtime is $O(N)$.

Technically the total runtime is $O(N^2 + N)$, but we say that this is just $O(N^2)$ because as N gets extremely large, this is the only term that matters.

Exploring Runtime

```
int vectorMaxQuadratic(const Vector<int>& v) {
    // Show the user we are doing something
    for (int i = 0; i < 100000000; i++) {
        cout << "Calculating..." << endl;
    }

    Vector<int> copy(v);
    selectionSort(copy);
    return copy[copy.size() - 1];
}
```

We say this runtime is constant O(1) relative to N.

We said this runtime is $O(N^2)$.

We could say the total runtime is $O(N^2 + 100M)$, but we don't care about constants (even large ones). If N = 100T, then it's insignificant! *We just care about how the algorithm performs relative to the data size.* So it's $O(N^2)$.

Constant Time

When an algorithm's time is *independent* of the number of elements in the container it holds, this is *constant time* complexity, or O(1). We love O(1) algorithms! Examples include (for efficiently designed data structures):

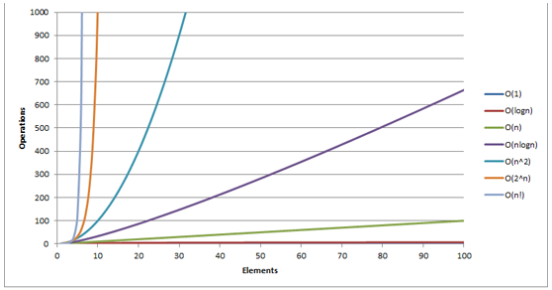
- Adding or removing from the *end* of a Vector.
- Pushing onto a stack or popping off a stack.
- Enqueuing or dequeuing from a queue.
- Other cool data structures we will cover soon (*hint:* one is a "hash table"!)

Runtime Rules

- If you have terms adding together, drop all but the **most dominant** term.
  - Code at the same indentation level adds
- We only care about terms that depend on the data size – constant terms (O(1)) are insignificant.
- Multiplying terms together *does* matter (e.g. selection sort).
  - Nested code multiplies
  - E.g. a loop over all the data, and inside that loop you loop again each time.
- Work from the innermost indented code out
- Realize that some code statements are more costly than others
  - It takes $O(N^2)$ time to call a function with runtime $O(N^2)$, even though calling that function is only one line of code

Complexity classes

- **complexity class**: A category of algorithm efficiency based on the algorithm's relationship to the input size "N".



Class	Big-Oh	If you double N, ...	Example
constant	O(1)	unchanged	10ms
logarithmic	$O(\log_2 N)$	increases slightly	175ms
linear	O(N)	doubles	3.2 sec
log-linear	$O(N \log_2 N)$	slightly more than doubles	11 sec
quadratic	$O(N^2)$	quadruples	1 min 42 sec
quad-linear	$O(N^2 \log_2 N)$	slightly more than quadruple	8 min
cubic	$O(N^3)$	multiplies by 8	55 min
...	...	...	...
exponential	$O(2^N)$	multiplies drastically	$5 * 10^{61}$ years
factorial	O(N!)	multiplies drastically	10^{200} years

What is the Big O?

```
int sum = 0;
for (int i = 1; i < 100000; i++) {
    for (int j = 1; j <= i; j++;) {
        for (int k = 1; k <= N; k++) {
            sum++;
        }
    }
}
Vector<int> v;
for (int x = 1; x <= N; x += 2) {
    v.insert(0, x);
}
cout << v << endl;
```

These loops are both O(1).

This loop is O(N).

O(1) * O(1) * O(N) = O(N)

This loop is O(N).

This statement is O(N).

O(N) * O(N) = O(N^2)

Total runtime = O(N) + O(N^2) = O(N^2).

More Examples

```
int nestedLoop1(int n){
        int result = 0;
        for (int i=0;i<n;i++){
                for (int j=0;j<n;j++){
                        for (int k=0;k<n;k++)
                                result++;
                }
        }
        return result;
}
```

What would the complexity be of a 3-nested loop?
Answer: n^3 (polynomial)
In real life, this comes up in 3D imaging, video, etc., and it is slow!
Graphics cards are built with hundreds or thousands of processors to tackle this problem!

More Examples

Practice: what is Big-O for this function?

$$20n^3 + 10n \log n + 5$$

More Examples

Practice: what is Big-O for this function?

$$20n^3 + 10n \log n + 5$$

Answer: O(n^3)

First, strip the constants: $n^3 + n \log n$

Then, find the biggest factor: n^3

More Examples

Practice: what is Big-O for this function?

$$2000 \log \mathrm{n} + 7n \log n + 5$$

void play(const Grid<double>& matrix)

More Examples

Practice: what is Big-O for this function?

$$2000 \log \mathrm{n} + 7n \log n + 5$$

Answer: O(n log n)

First, strip the constants: $\log n + n \log n$

Then, find the biggest factor: $n \log n$

Preparing for the Worst

```
void linearSearchVector(Vector<int> &vec, int numToFind){
    int numCompares = 0;
    bool answer = false;
    int n = vec.size();

    for (int i = 0; i < n; i++) {
        numCompares++;
        if (vec[i]==numToFind) {
            answer = true;
            break;
        }
    }
    cout << "Found? " << (answer ? "True" : "False") << ", "
         << "Number of compares: " << numCompares << endl << endl;
}
```

Preparing for the Worst

```
void linearSearchVector(Vector<int> &vec, int numToFind){
    int numCompares = 0;
    bool answer = false;
    int n = vec.size();

    for (int i = 0; i < n; i++) {
        numCompares++;
        if (vec[i]==numToFind) {
            answer = true;
            break;
        }
    }
    cout << "Found? " << (answer ? "True" : "False") << ", "
         << "Number of compares: " << numCompares << endl << endl;
}
```

Complexity: O(n) (linear, worst case)

You have to walk through the entire vector one element at a time.

Preparing for the Worst

```
void linearSearchVector(Vector<int> &vec, int numToFind){
    int numCompares = 0;
    bool answer = false;
    int n = vec.size();

    for (int i = 0; i < n; i++) {
        numCompares++;
        if (vec[i]==numToFind) {
            answer = true;
            break;
        }
    }
    cout << "Found? " << (answer ? "True" : "False") << ", "
         << "Number of compares: " << numCompares << endl << endl;
}
```

Best case? O(1)

Worst case? O(*n*)

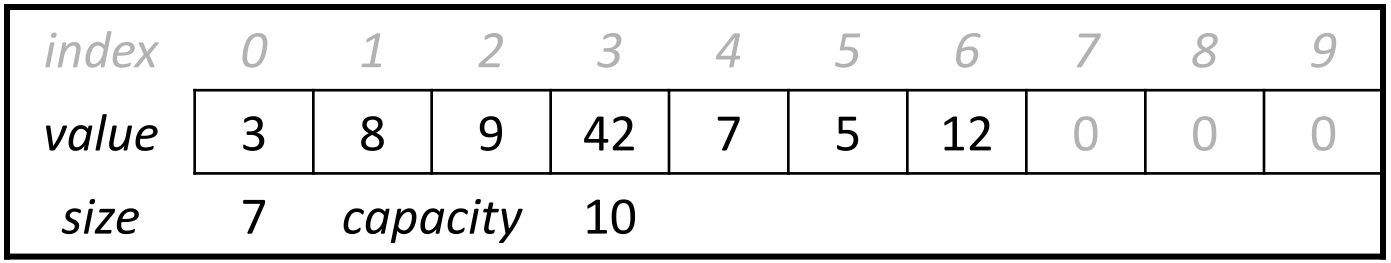
Complexity: O(n) (linear, worst case)

You have to walk through the entire vector one element at a time.

Preparing for the Worst

- In general, we always examine the *worst case* runtime.
- Sometimes, when explicitly mentioned, we examine the best case or average case. But we always assume we are discussing worst case unless explicitly mentioning otherwise.

Vector/LinkedList runtime



Member	Vector	LinkedList
`add(`***value***`);`	O(1)	O(1)
`get(`***i***`) or [`***i***`]`	**O(1)**	**O(N)***
`insert(`***i***`, `***value***`);`	**O(N)***	**O(N)***
`remove(`***i***`);`	**O(N)***	**O(N)***
`set(`***i***`, `***val***`) or [`***i***`]=`	**O(1)**	**O(N)***
`size(), isEmpty()`	O(1)	O(1)
`toString(), cout << `***v***	**O(N)**	**O(N)**

* *average-case* runtime;

Vector = O(1) at end, worst at front;

LinkedList = O(1) at front and end, worst in middle

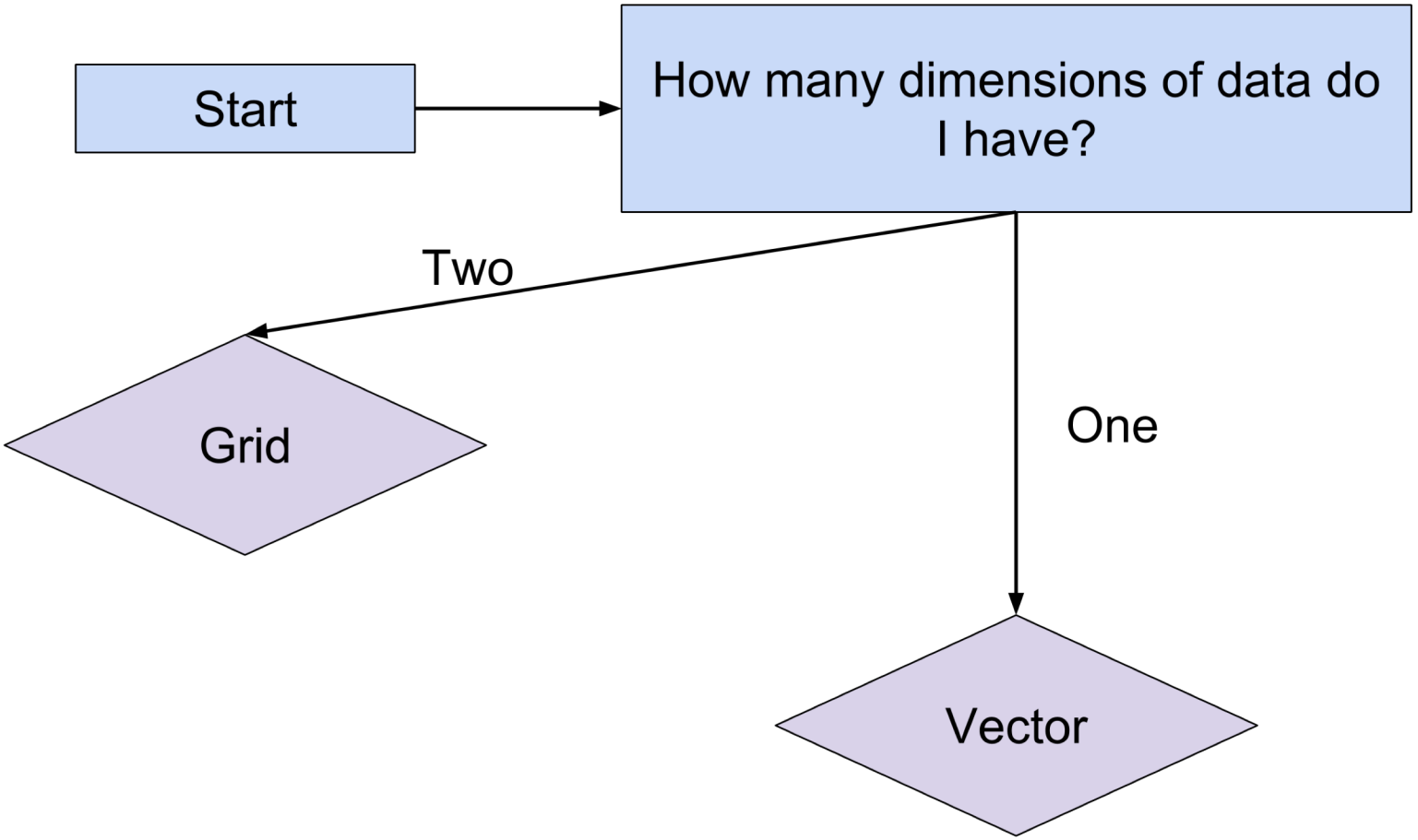
What is N?

- N represents the “data size”: this could mean:
  - The number of elements in a data structure
  - The number passed in to a function representing # times to loop
- It depends on what the algorithm is (but there is likely only 1 clear choice)

Plan For Today

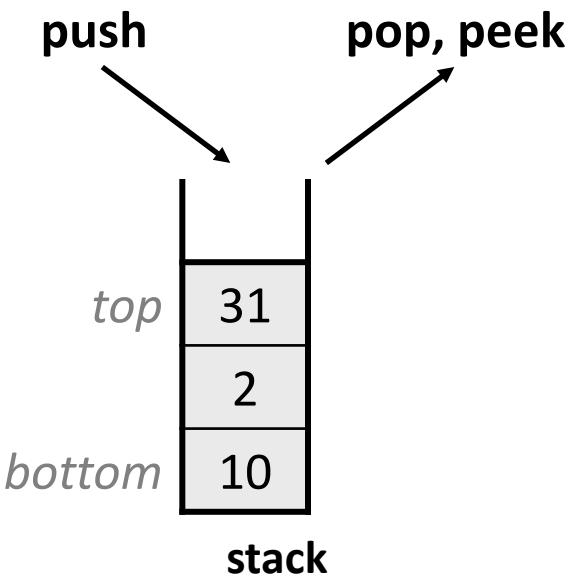
- Recap: Vectors
- Big-O
- Stacks
- Announcements
- Queues

ADTs – the Story so Far



A new ADT: the Stack

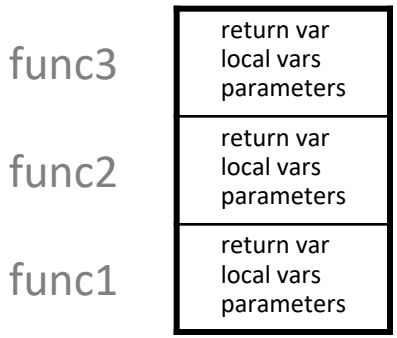
- A **Stack** is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of or ***popped*** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- Main operations:
  - **`push(value)`**: add an element to the top of the stack
  - **`pop()`**: remove and return the top element in the stack
  - **`peek()`**: return (but do not remove) the top element in the stack



Stacks in computer science

- Programming languages and compilers:
  - function calls are placed onto a stack *(call=push, return=pop)*
  - compilers use stacks to evaluate expressions

- Matching up related pairs of things:
  - find out whether a string is a palindrome
  - examine a file to see if its braces { } match
  - convert "infix" expressions to pre/postfix



- Sophisticated algorithms:
  - searching through a maze with "backtracking"
  - many programs use an "undo stack" of previous operations

The Stack class

```
#include "stack.h"
```

`s.isEmpty()`	O(1)	returns `true` if stack has no elements
`s.peek()`	O(1)	returns **top** value without removing it; throws an error if stack is empty
`s.pop()`	O(1)	removes **top** value and returns it; throws an error if stack is empty
`s.push(value);`	O(1)	places given value on **top** of stack
`s.size()`	O(1)	returns number of elements in stack

```
Stack<int> s;                          // {}      bottom -> top
s.push(42);                            // {42}
s.push(-3);                            // {42, -3}
s.push(17);                            // {42, -3, 17}

cout << s.pop() << endl;               // 17   (s is {42, -3})
cout << s.peek() << endl;              // -3   (s is {42, -3})
cout << s.pop() << endl;               // -3   (s is {42})
```

Stack limitations/idioms

- You cannot access a stack's elements by index.

```
Stack<int> s;
...
for (int i = 0; i < s.size(); i++) {
    do something with s[i];          // does not compile
}
```

- Instead, you pull elements out of the stack one at a time.

- **common idiom: Pop each element until the stack is empty.**

```
// process (and empty!) an entire stack
while (!s.isEmpty()) {
    do something with s.pop();
}
```

Stacks In The Real World

top

bottom

I'm heading home. By, friends!

Stacks In The Real World

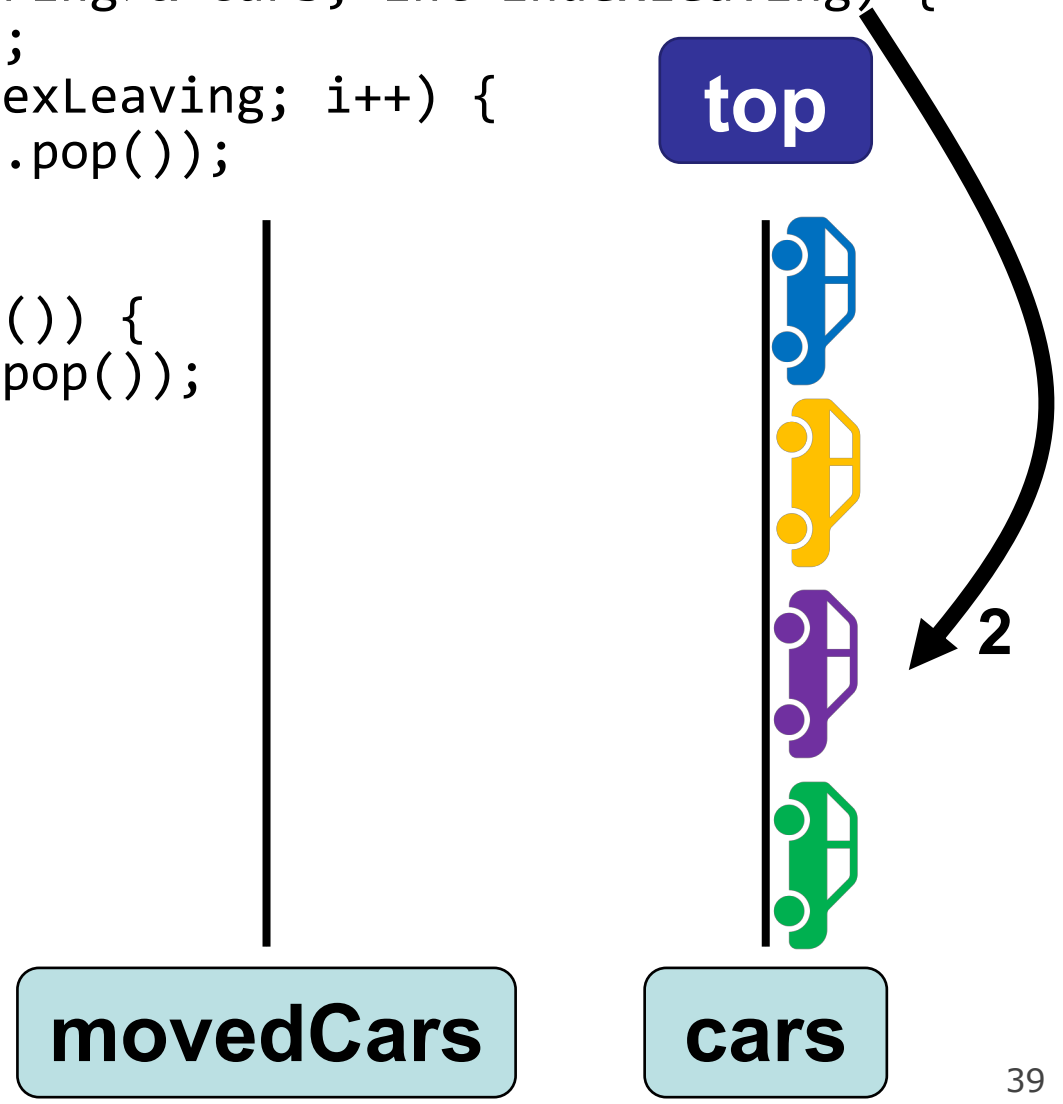
top

bottom

Challenge: given a stack of "cars" (strings), and the index of the car leaving, remove that car from the stack, but preserve the order otherwise.

Exercise solution

```
void leaveCarStack(Stack<string>& cars, int indexLeaving) {
    Stack<string> movedCars;
    for (int i = 0; i < indexLeaving; i++) {
        movedCars.push(cars.pop());
    }
    cars.pop();
    while (!movedCars.empty()) {
       cars.push(movedCars.pop());
    }
}
```



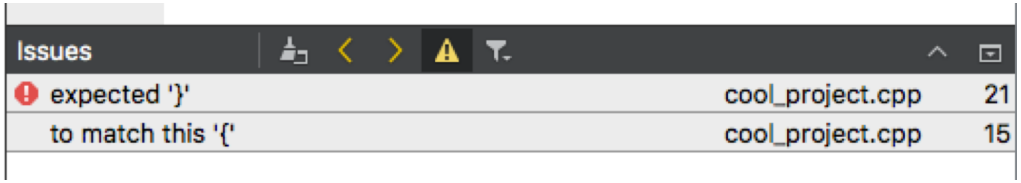
Stack exercise

checkBalance

- Write a function **`checkBalance`** that accepts a string of source code and checks whether the braces/parentheses are balanced.
  - Every (or { must be closed by a } or) in the opposite order.
  - Return the index at which an imbalance occurs, or -1 if balanced.
  - If any (or { are never closed, return the string's length.

  - Examples:

```
//     index    01234567890123456789012345678901234567890
checkBalance("if (a(4) > 9) { foo(a(2)); }") returns -1
checkBalance("for (i=0;i<a(3};i++) { foo(); )") returns 14
checkBalance("while (true) foo(); }{ ()") returns 20
checkBalance("if (x) {") returns 8
```

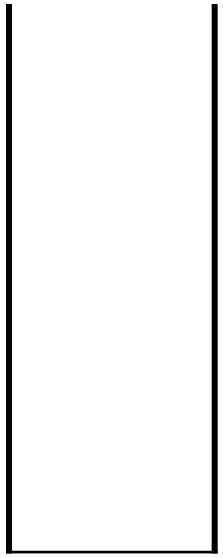


Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```

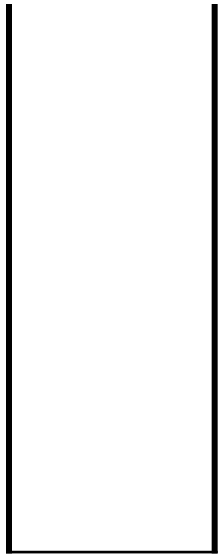
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
^
```



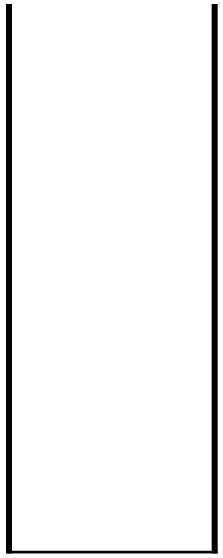
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
 ^
```



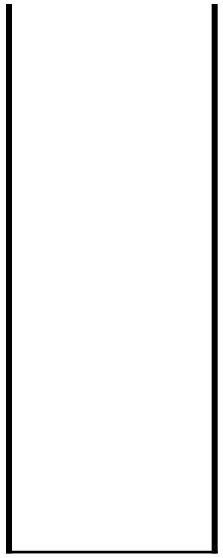
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
  ^
```



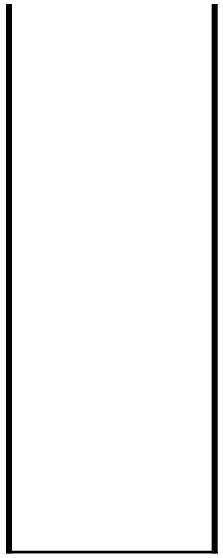
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
    ^
```



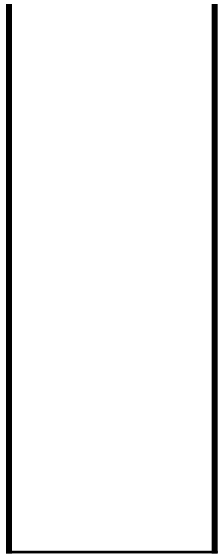
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
    ^
```



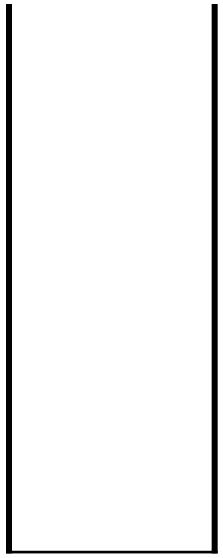
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
      ^
```



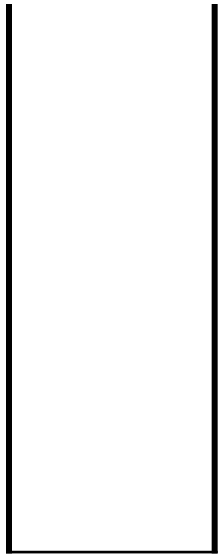
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
      ^
```



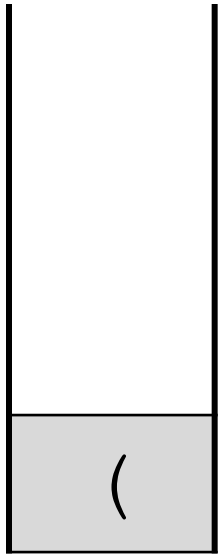
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
        ^
```



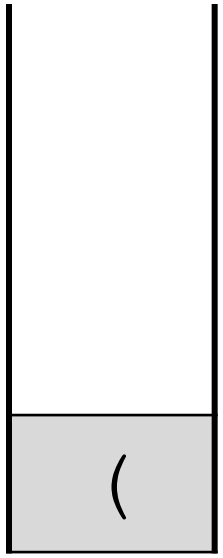
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
        ^
```



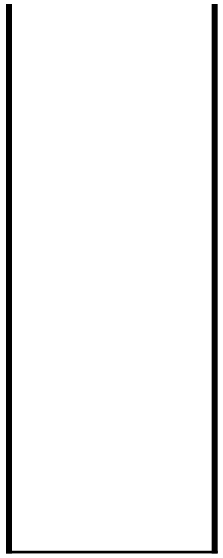
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
          ^
```



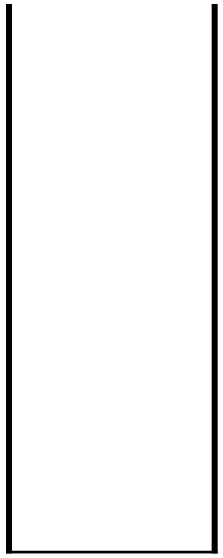
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
          ^
```



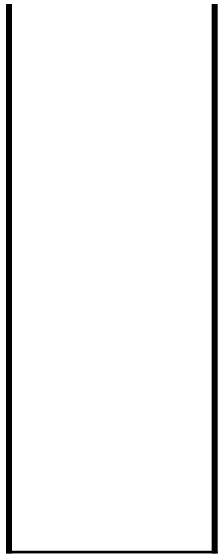
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
          ^
```



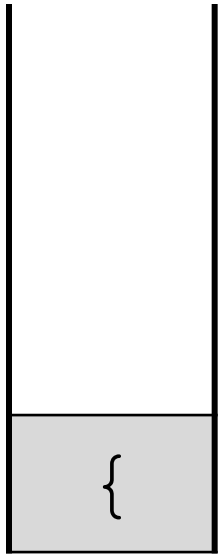
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
          ^
```



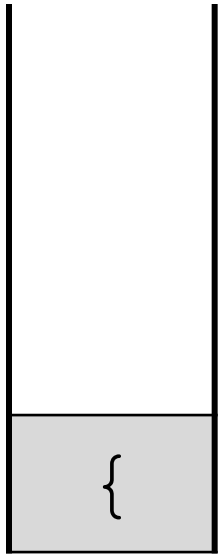
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
          ^
```



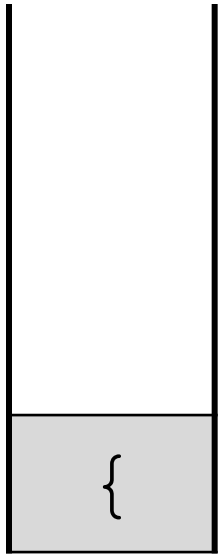
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
            ^
```



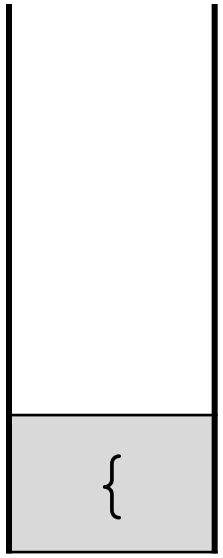
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
            ^
```



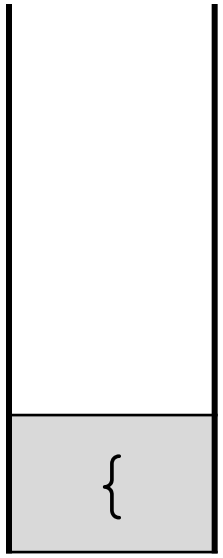
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
            ^
```



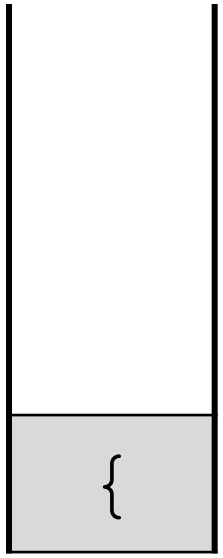
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
            ^
```



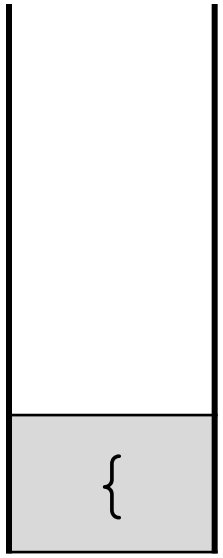
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
            ^
```



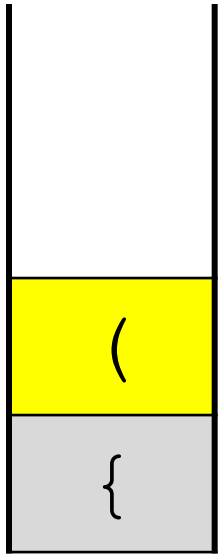
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
            ^
```



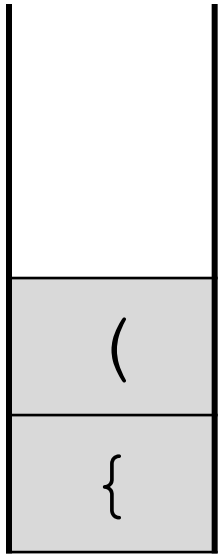
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
            ^
```



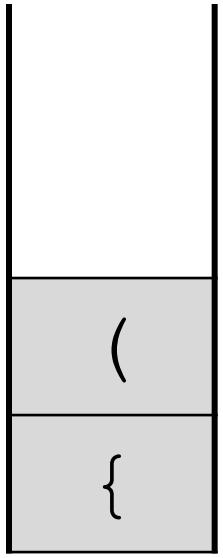
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                 ^
```



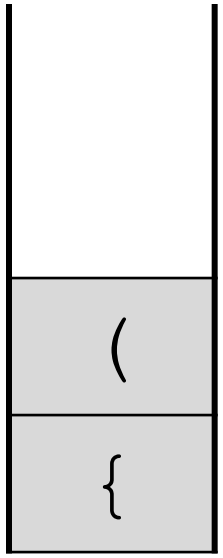
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                  ^
```



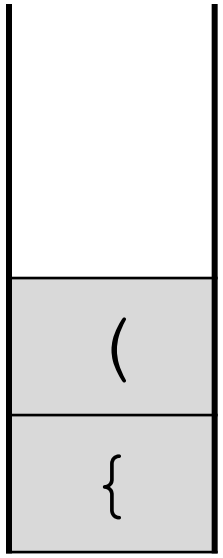
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                  ^
```



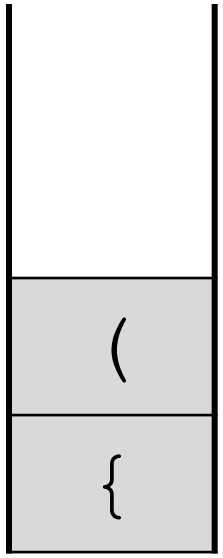
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                  ^
```



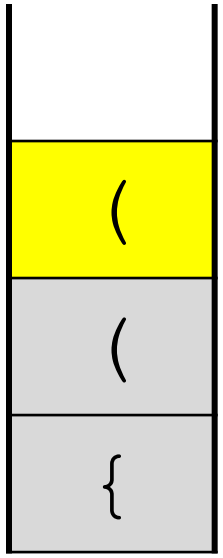
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                    ^
```



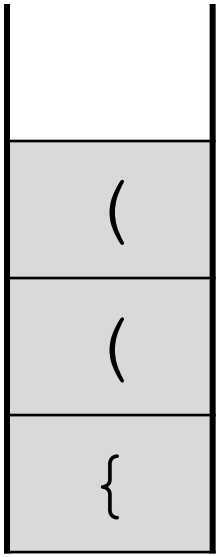
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                    ^
```



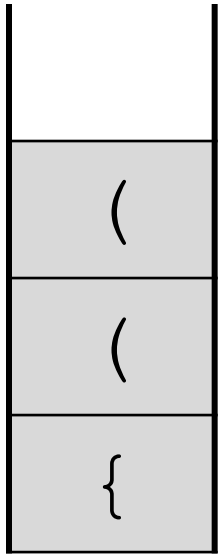
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                    ^
```



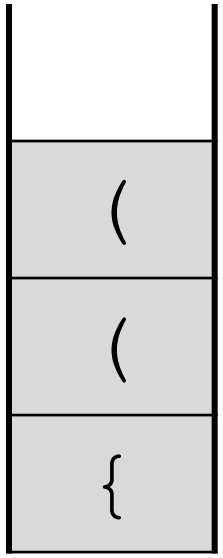
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                    ^
```



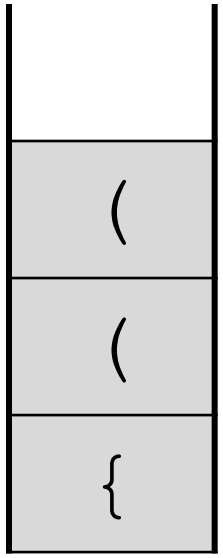
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                    ^
```



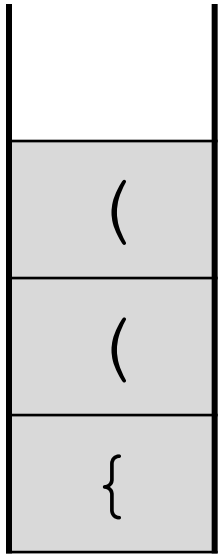
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                    ^
```



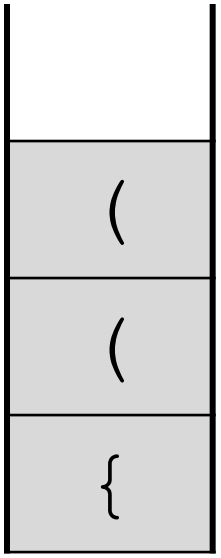
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                          ^
```



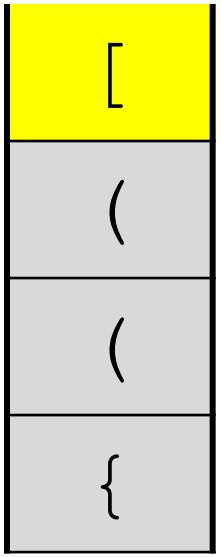
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                          ^
```



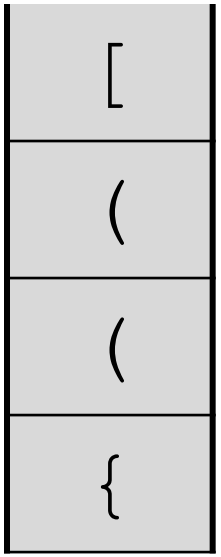
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                          ^
```



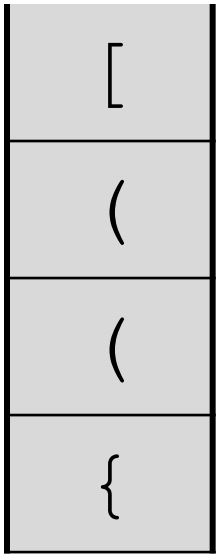
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                          ^
```



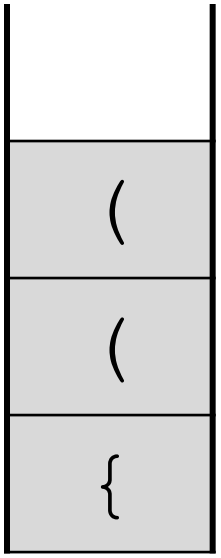
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                          ^
```



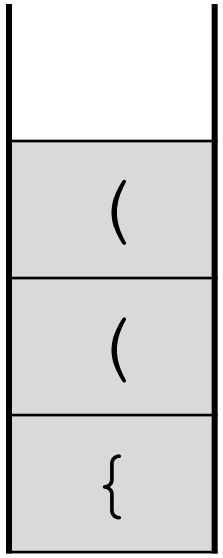
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                          ^
```



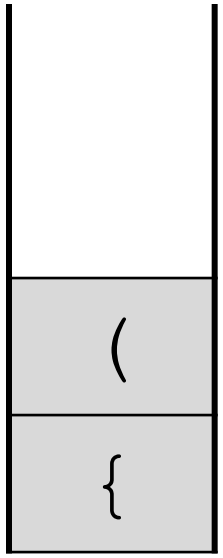
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                          ^
```



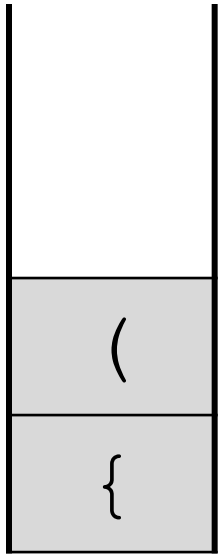
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                          ^
```



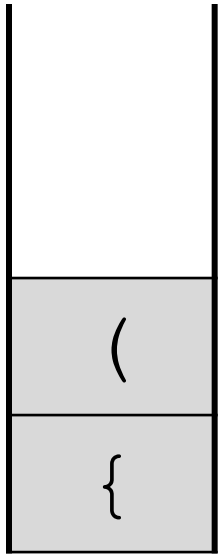
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                ^
```



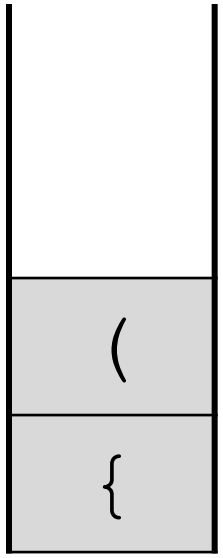
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                ^
```



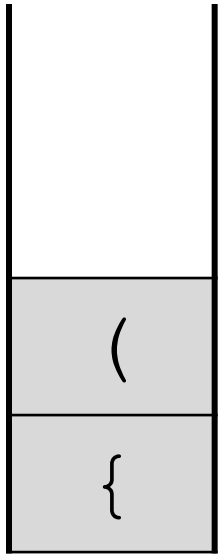
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                ^
```



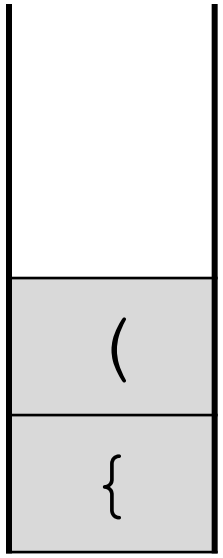
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                 ^
```



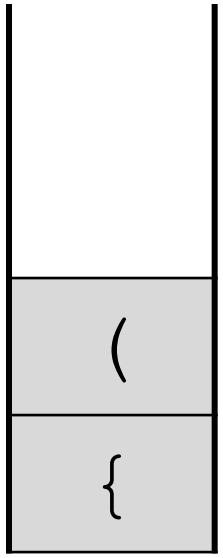
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                 ^
```



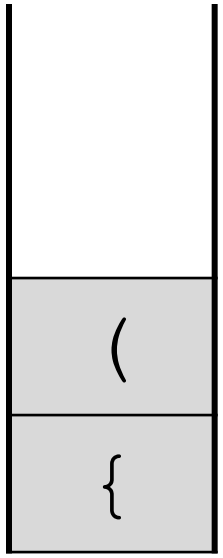
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                  ^
```



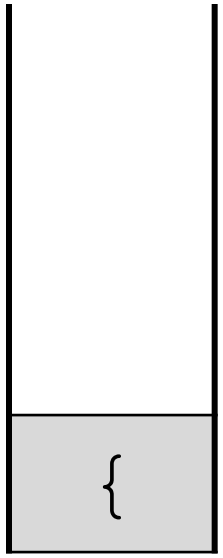
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                  ^
```



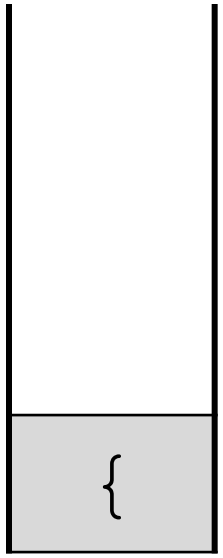
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                  ^
```



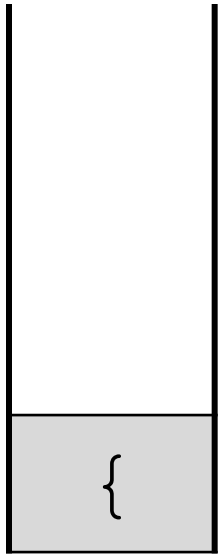
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                      ^
```



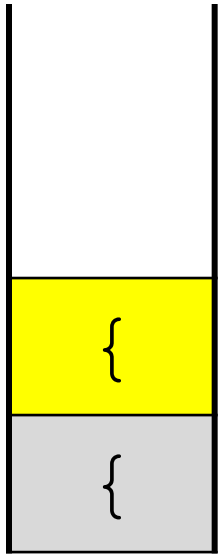
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                      ^
```



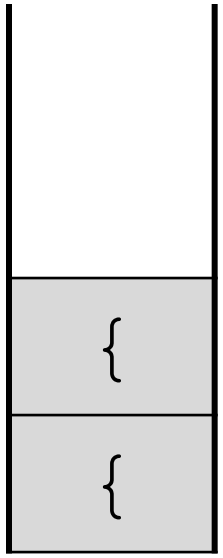
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                      ^
```



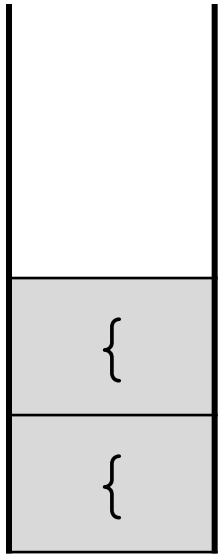
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                        ^
```



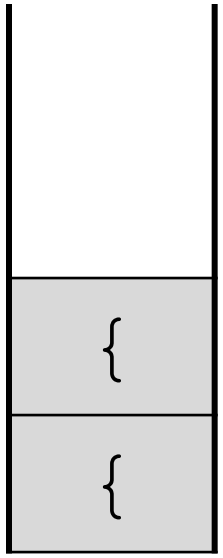
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                        ^
```



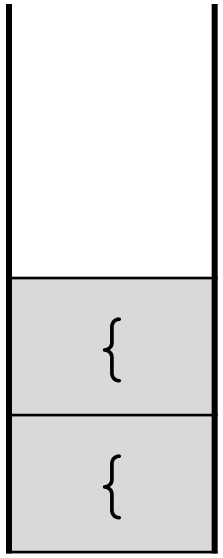
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                        ^
```



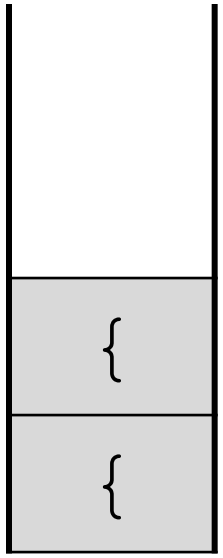
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                        ^
```



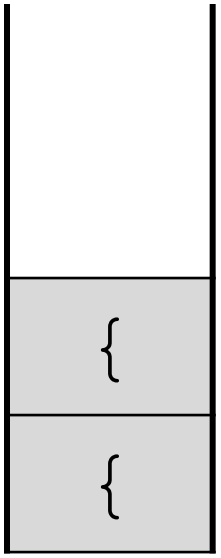
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                        ^
```



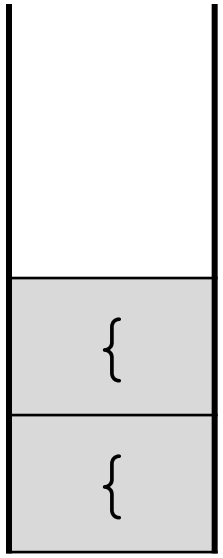
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                          ^
```



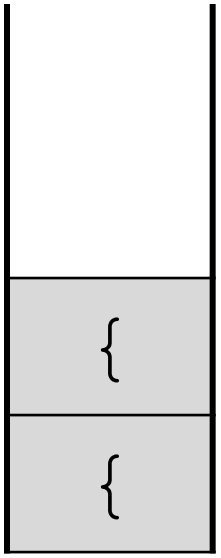
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                          ^
```



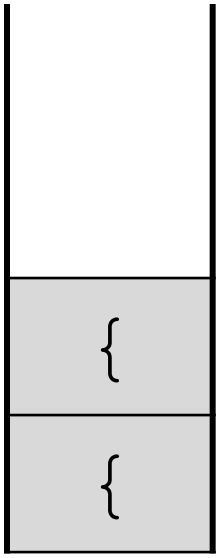
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                          ^
```



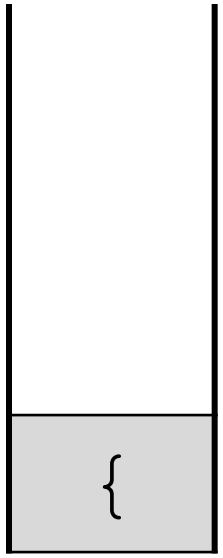
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                          ^
```



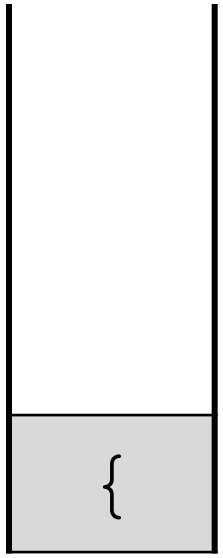
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                          ^
```



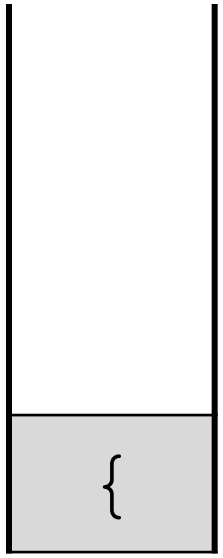
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                              ^
```



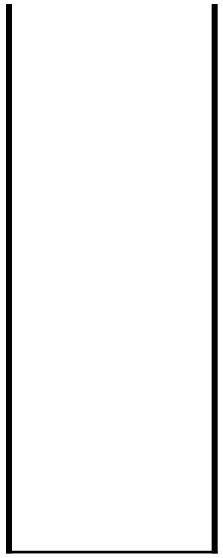
Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                                ^
```



Let's Be A Compiler!

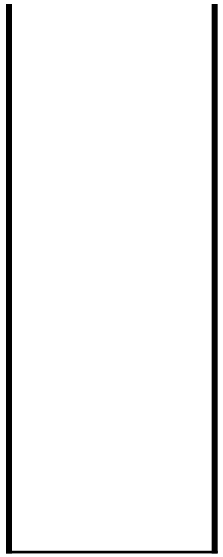
```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                                ^
```



Let's Be A Compiler!

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                      ^
```

Let's code this up!



Our Algorithm

- For each character:
  - If it's an open parenthesis of some sort, push it onto the stack.
  - If it's a close parenthesis of some sort:
    - If the stack is empty, report an error.
    - If the character doesn't pair with the character on top of the stack, report an error.
- At the end, return whether the stack is empty (nothing was left unmatched.)

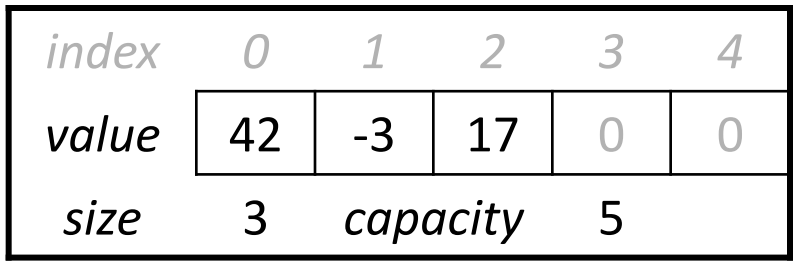
Exercise solution

```
int checkBalance(const string& line) {
    Stack<char> parens;
    for (int i = 0; i < line.length(); i++) {
        char c = line[i];
        if (c == '(' || c == '{' || c == '[') {
            parens.push(c);
        } else if (c == ')' || c == '}' || c == ']') {
            // If there is nothing to match, stop
            if (parens.isEmpty()) {
                return i;
            }
            char top = parens.pop();
            // If there is something, but not the right match, stop
            if ((top == '(' && c != ')') || (top == '{' && c != '}') ||
                    (top == '[' && c != ']')) {
                return i;
            }
        }
    }
    if (parens.isEmpty()) {
        return -1;              // balanced
    } else {
        return line.length();
    }
}
```

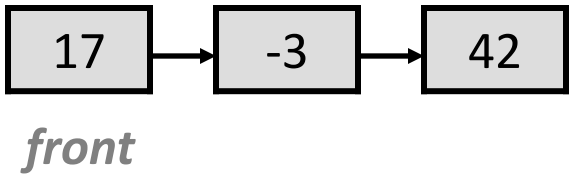
Stack implementation

- A stack is often implemented using an **array** or **Vector** internally.
  - "bottom" = index 0
  - "top" = index (size - 1) *(why not the other way around?)*

```
Stack<int> s;
s.push(42);
s.push(-3);
s.push(17);
```



- A stack can also be implemented using a **linked list**.
  - "top" = front



Plan For Today

- Recap: Vectors
- Big-O
- Stacks
- Announcements
- Queues

Announcements

- Sections assigned
  - Check the course website
  - Late signups are open until **Tues. 10/9 @ 5PM**
- Try submitting HW1 on Paperless!
  - We accept multiple submissions
  - Make sure you are able to submit and can submit successfully

HOPES

hopes
HUNTINGTON'S OUTREACH PROJECT FOR EDUCATION, AT STANFORD

HOPES is hiring
student researchers,
graphic designers,
and web developers.

The Huntington's Outreach Project for Education, at Stanford (HOPES) is an educational service project working to build a web resource on Huntington's disease (HD). Our mission is to make scientific information about HD more accessible to patients, their families, and the general public.

Apply by **Sunday, October 7th at 11:59 pm!** Please send a resume, letter of application, and unofficial transcript to HOPES project leader Cole Holderman (jcoleh@stanford.edu) with the subject line "YOUR LAST NAME - HOPES application." The letter should include a candid discussion of your qualifications, other time commitments, leadership skills, and reasons for interest in the position.

Student researchers: Please attach two writing samples that are science-related or research-based in nature.

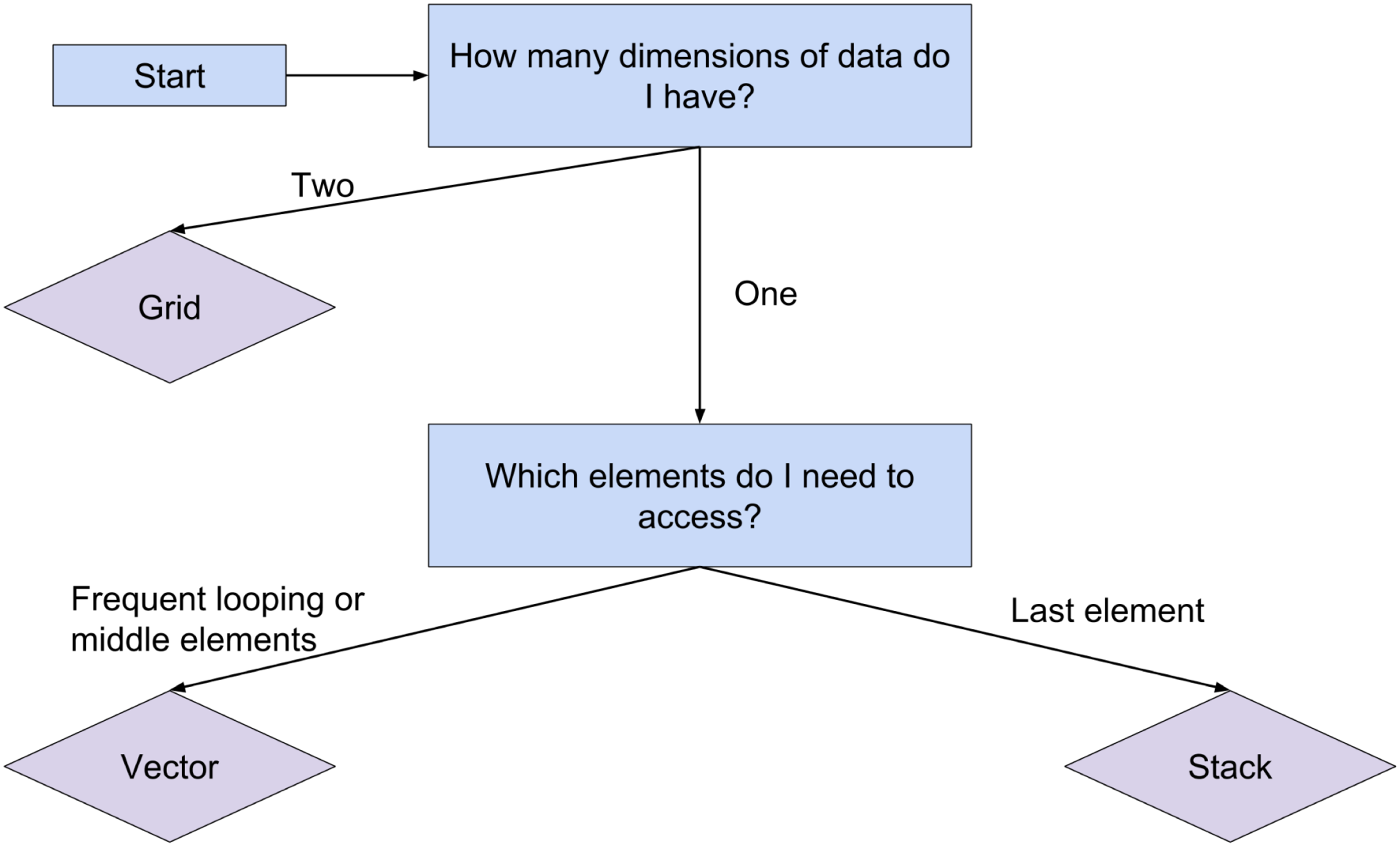
Graphic designers: Please send in 3 recent designs with a description about each (tools used, time spent, purpose/client)

Web developers: Please send links to any of your web-design work.
For more information, please visit hopes.stanford.edu or email the project leader.

Plan For Today

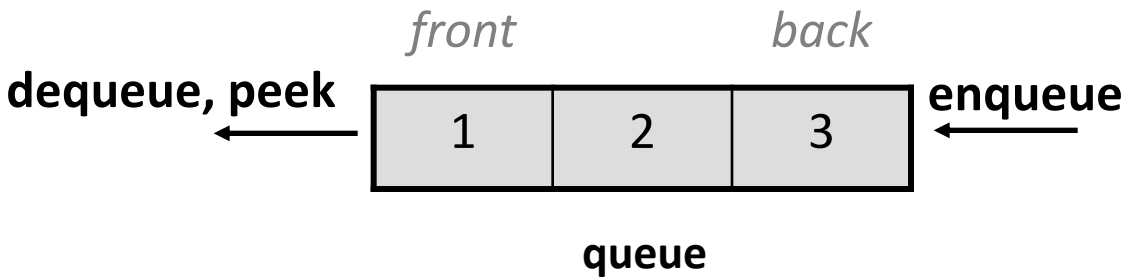
- Recap: Vectors
- Big-O
- Stacks
- Announcements
- Queues

ADTs – the Story so Far



Queues (5.3)

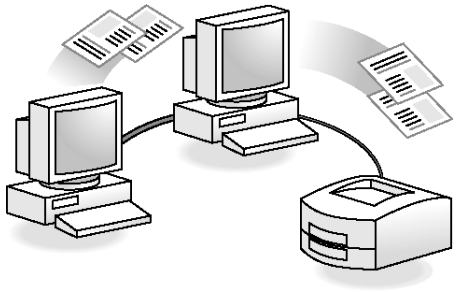
- **queue**: Retrieves elements in the order they were added.
  - First-In, First-Out ("FIFO")
  - Elements are stored in order of insertion but don't have indexes.
  - Can add only to the end of the queue, and can only examine/remove the front.



- basic queue operations:
  - **enqueue**: Add an element to the back.
  - **dequeue**: Remove the front element.
  - **peek**: Examine the front element.

Queues in computer science

- Operating systems:
  - queue of print jobs to send to the printer
  - queue of programs / processes to be run
  - queue of network data packets to send



- Programming:
  - modeling a line of customers or clients
  - storing a queue of computations to be performed in order

- Real world examples:
  - people on an escalator or waiting in a line
  - cars at a gas station (or on an assembly line)

The Queue class

```
#include "queue.h"
```

`q.dequeue()`	O(1)	removes **front** value and returns it; throws an error if queue is empty
`q.enqueue(value);`	O(1)	places given value at **back** of queue
`q.isEmpty()`	O(1)	returns `true` if queue has no elements
`q.peek()`	O(1)	returns **front** value without removing; throws an error if queue is empty
`q.size()`	O(1)	returns number of elements in queue

```
Queue<int> q;                          // {}    front -> back
q.enqueue(42);                         // {42}
q.enqueue(-3);                         // {42, -3}
q.enqueue(17);                         // {42, -3, 17}
cout << q.dequeue() << endl;           // 42    (q is {-3, 17})
cout << q.peek() << endl;              // -3    (q is {-3, 17})
cout << q.dequeue() << endl;           // -3    (q is {17})
```

Queue idioms

- As with stacks, must pull contents out of queue to view them.

```
// process (and destroy) an entire queue
while (!q.isEmpty()) {
    do something with q.dequeue();
}
```

  - another idiom: Examining each element exactly once.

```
int size = q.size();
for (int i = 0; i < size; i++) {
    do something with q.dequeue();
    (including possibly re-adding it to the queue)
}
```

    - Note the `size` variable, which is important; don't use `size()` directly.

Mixing stacks and queues

- We often mix stacks and queues to achieve certain effects.
  - Example: Reverse the order of the elements of a queue.

```
Queue<int> q;
q.enqueue(1);
q.enqueue(2);
q.enqueue(3);                   // q={1, 2, 3}

Stack<int> s;
while (!q.isEmpty()) {          // transfer queue to stack
    s.push(q.dequeue());        // q={} s={1, 2, 3}
}
while (!s.isEmpty()) {          // transfer stack to queue
    q.enqueue(s.pop());         // q={3, 2, 1} s={}
}
cout << q << endl;              // {3, 2, 1}
```

Queue exercises

- Write a function **doubleCopy** that accepts a queue of integers and replaces every element with two copies of itself.
  - `{1, 2, 3}`
    becomes
    `{1, 1, 2, 2, 3, 3}`

Exercise solutions

```
void doubleCopy(Queue<int>& q) {
    int size = q.size();
    for (int i = 0; i < size; i++) {
        int n = q.dequeue();
        q.enqueue(n);
        q.enqueue(n);
    }
}
```

Queue exercises

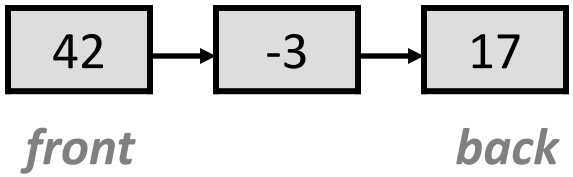
- Write a function **`mirror`** that accepts a queue of strings and appends the queue's contents to itself in reverse order.
  - `{"a", "b", "c"}`
    becomes
    `{"a", "b", "c", "c", "b", "a"}`

Exercise solutions

```
void mirror(Queue<string>& q) {
    Stack<string> s;
    int size = q.size();
    for (int i = 0; i < size; i++) {
        string str = q.dequeue();
        s.push(str);
        q.enqueue(str);
    }
    while (!s.isEmpty()) {
        q.enqueue(s.pop());
    }
}
```

Queue implementation

- A queue is often implemented using a **linked list** internally.
  - "front" = front
  - "back" = back




```
Queue<int> q;
q.enqueue(42);
q.enqueue(-3);
q.enqueue(17);
```

- A stack can also be implemented using a **circular** array/Vector.

```
q.dequeue();
q.dequeue();
q.enqueue(99);
q.enqueue(86);
q.enqueue(31);
```

index	*0*	*1*	*2*	*3*	*4*
value	31	*0*	17	99	86
size	4	*cap*	5	*front*	2

Plan For Today

- Recap: Vectors
- Big-O
- Stacks
- Announcements
- Queues
- Deques

Next time: Sets and Maps

Overflow

Exponential Time

There are a number of algorithms that have *exponential* behavior. If we don't like quadratic or polynomial behavior, we *really* don't like exponential behavior.

Example: what does the following beautiful recursive function do?

```
long mysteryFunc(int n) {
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    return mysteryFunc(n-1) + mysteryFunc(n-2);
}
```

This is the *fibonacci sequence!* 0, 1, 1, 2, 3, 5, 8, 13, 21 …

Exponential Time

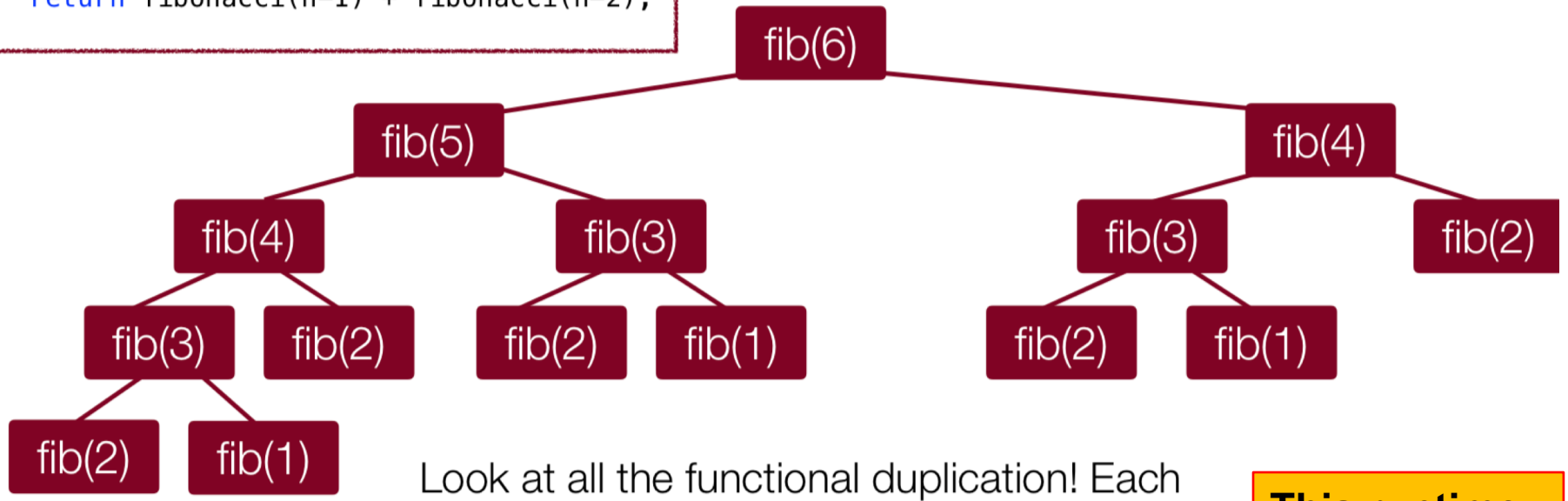
```
long fibonacci(int n) {
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Beautiful, but a flawed algorithm! Yes, it works, but why is it flawed? Let's look at the call tree for fib(6):

Exponential Time

```
long fibonacci(int n) {
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    return fibonacci(n-1) + fibonacci(n-2);
}
```

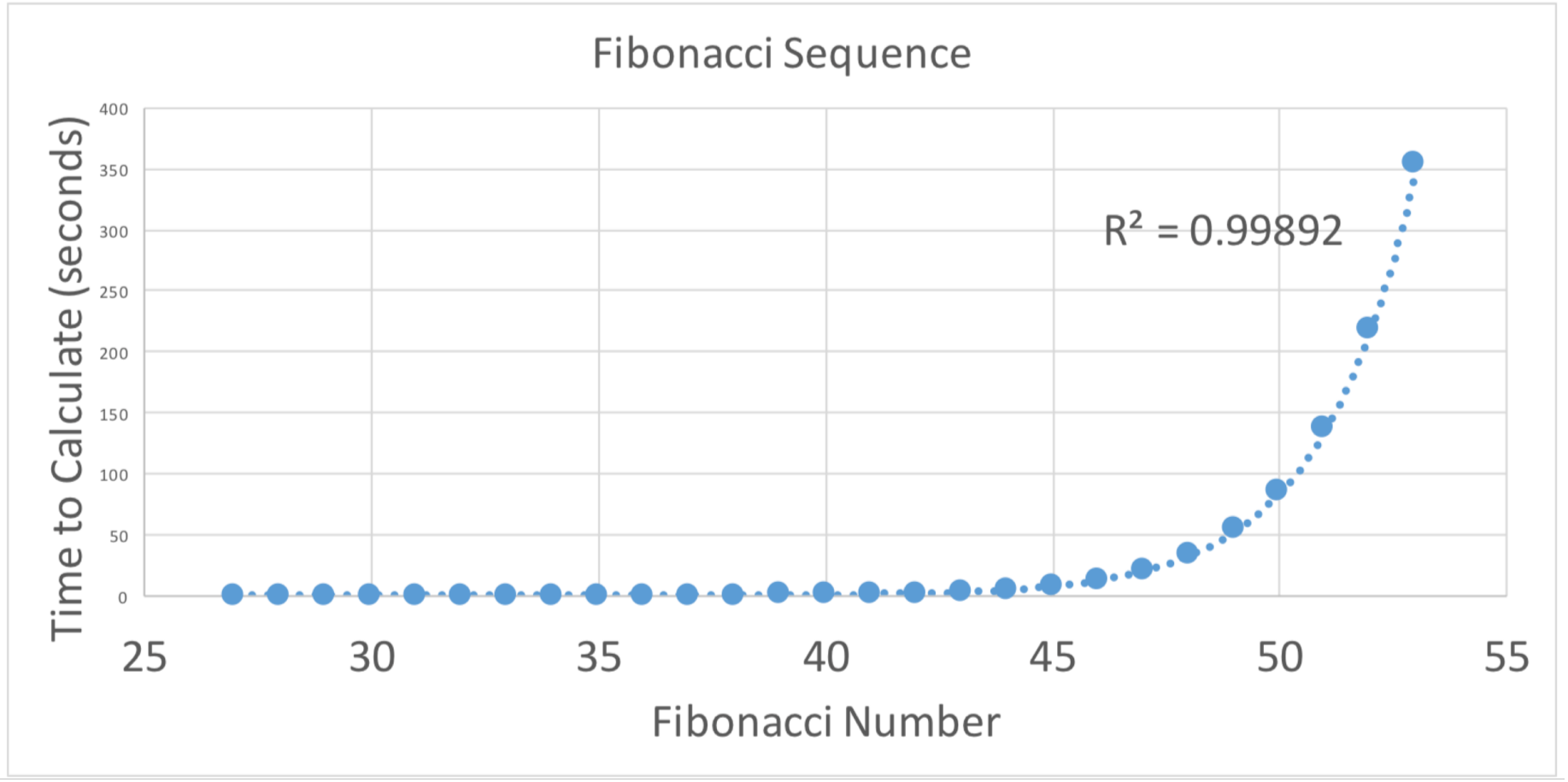
Beautiful, but a flawed algorithm! Yes, it works, but why is it flawed? Let's look at the call tree for fib(6):



Look at all the functional duplication! Each call (down to level 3) has to make two recursive calls, and many are duplicated!

This runtime is $O(2^N)$!

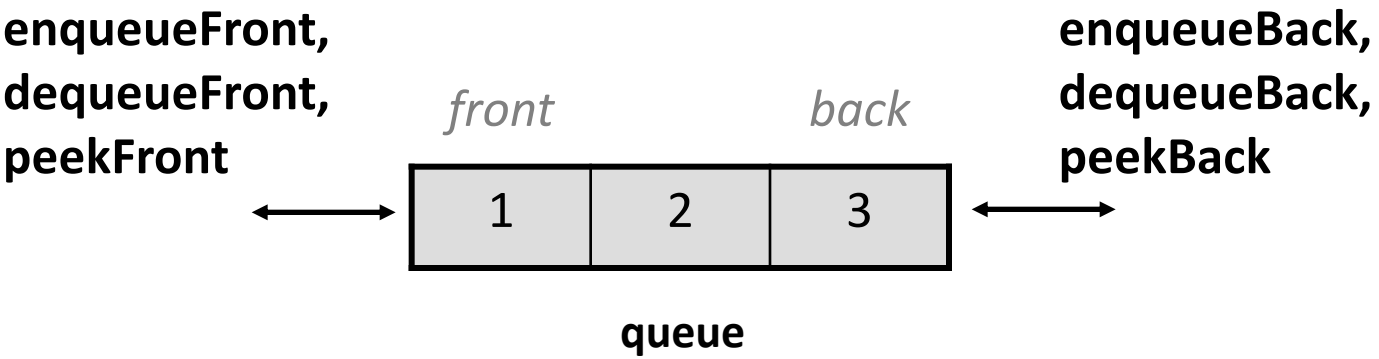
Exponential Time



Deques

- **deque**: Double-ended queue.
  - Can add/remove from either end.
  - Combines many of the benefits of stack and queue.
  - Often implemented using a linked list.

- basic deque operations:
  - **enqueueFront/Back; dequeueFront/Back; peekFront/Back**



The Deque class

```
#include "deque.h"
```

`d.dequeueBack()`	O(1)	removes **back** value and returns it; throws an error if queue is empty
`d.dequeueFront()`	O(1)	removes **front** value and returns it
`d.enqueueBack(value);`	O(1)	places given value at **back** of queue
`d.enqueueFront(value);`	O(1)	places given value at **front** of queue
`d.isEmpty()`	O(1)	returns `true` if queue has no elements
`d.peekBack()`	O(1)	returns **back** value without removing; throws an error if queue is empty
`d.peekFront()`	O(1)	returns **front** value without removing; throws an error if queue is empty
`d.size()`	O(1)	returns number of elements in queue
`d.toString()`	O(*N*)	returns e.g. `"{42, -3, 17}"`
`ostr << d`	O(*N*)	outputs deque to stream