Data Structures and Sorting Implementations

Sumner Evans

25 August 2023

Colorado School of Mines

Why Do We Care?

Data structures allow us to hide implementation details of how our data is stored in memory. This is convenient for programmers because it allows us to interact with our data via *abstractions*.

However, without understanding some of the implementation details, we might misanalyse our algorithmic complexities when using these data structures.

This can not only lead to you getting a bad grade on your algorithms homework assignments or projects, but it can also lead to you writing code which you *think* is efficient, but is actually very slow.

Why Do We Care?

Data structures allow us to hide implementation details of how our data is stored in memory. This is convenient for programmers because it allows us to interact with our data via *abstractions*.

However, without understanding some of the implementation details, we might misanalyse our algorithmic complexities when using these data structures.

This can not only lead to you getting a bad grade on your algorithms homework assignments or projects, but it can also lead to you writing code which you *think* is efficient, but is actually very slow.

Why Do We Care?

Data structures allow us to hide implementation details of how our data is stored in memory. This is convenient for programmers because it allows us to interact with our data via *abstractions*.

However, without understanding some of the implementation details, we might misanalyse our algorithmic complexities when using these data structures.

This can not only lead to you getting a bad grade on your algorithms homework assignments or projects, but it can also lead to you writing code which you *think* is efficient, but is actually very slow.

Lists

A list is a data structure which supports at least the following operations:

- `append(x)` or `push(x)`: Add the element `x` to the end of the list.
- `at(i)` or the `[i]` operator: Get the element at index i.
- `insert(i, x)`: Insert the element `x` at index i.

What is the complexity of `append(x)`? `at(i)`? `insert(i, x)`?

This is a trick question! The answer depends on the list data structure implementation you are using.

Lists

A list is a data structure which supports at least the following operations:

- `append(x)` or `push(x)`: Add the element `x` to the end of the list.
- `at(i)` or the `[i]` operator: Get the element at index i.
- `insert(i, x)`: Insert the element `x` at index i.

What is the complexity of `append(x)`? `at(i)`? `insert(i, x)`?

This is a trick question! The answer depends on the list data structure implementation you are using.

Lists

A list is a data structure which supports at least the following operations:

- `append(x)` or `push(x)`: Add the element `x` to the end of the list.
- `at(i)` or the `[i]` operator: Get the element at index i.
- `insert(i, x)`: Insert the element `x` at index i.

What is the complexity of `append(x)`? `at(i)`? `insert(i, x)`?

This is a trick question! The answer depends on the list data structure implementation you are using.

List Implementations in C++

In C++ there are four main list implementations:

- `std::vector`
- `std::list`
- `std::forward_list`
- `std::deque`

Let's look at cppreference.com to see what the documentation says about the complexity of the various operations.

Question

With your neighbour, write down one real-world situation where you would use each of the four list implementations.

List Implementations in C++

In C++ there are four main list implementations:

- `std::vector`
- `std::list`
- `std::forward_list`
- `std::deque`

Let's look at cppreference.com to see what the documentation says about the complexity of the various operations.

Question

With your neighbour, write down one real-world situation where you would use each of the four list implementations.

Queues, Stacks, and Deques

In languages that have queues and stacks in their standard libraries, they are usually implemented as wrappers around a list data structure.

Stacks can be implemented with a list since appending and removing from the end of a list is $\mathcal{O}(1)$.

Queues are generally implemented with a deque since removal from the front and insertion at the back of a queue are both $\mathcal{O}(1)$ operations.

Queues, Stacks, and Deques

In languages that have queues and stacks in their standard libraries, they are usually implemented as wrappers around a list data structure.

Stacks can be implemented with a list since appending and removing from the end of a list is $\mathcal{O}(1)$.

Queues are generally implemented with a deque since removal from the front and insertion at the back of a queue are both $\mathcal{O}(1)$ operations.

Queues, Stacks, and Deques

In languages that have queues and stacks in their standard libraries, they are usually implemented as wrappers around a list data structure.

Stacks can be implemented with a list since appending and removing from the end of a list is $\mathcal{O}(1)$.

Queues are generally implemented with a deque since removal from the front and insertion at the back of a queue are both $\mathcal{O}(1)$ operations.

What about {MY_FAVOURITE_LANGUAGE}?

- Python has `list` (dynamically sized array) and `collections.deque` (doubly-linked list).
- Rust has `std::vec::Vec` (dynamically sized array), `std::collections::VecDeque` (ring buffer), and `std::collections::LinkedList` (doubly-linked list).
- JavaScript has `Array` and `TypedArray` (both dynamically sized arrays).
- Java has `ArrayList` (dynamically sized array), `LinkedList` (doubly-linked list), and `Vector` (synchronized dynamically sized array).
- Go has slices (dynamically sized arrays).

Dictionaries (Maps)

Generally, a dictionary is a data structure which supports at least the following operations:

- `get(k)` or `find(k)`: Get the value associated with the key `k`.
- `set(k, v)`: Set the value associated with the key `k` to `v`.
- `remove(k)`: Remove the key `k` from the dictionary.
- `contains(k)`: Return whether or not the dictionary contains the key `k`.

Dictionary Implementations in C++

In C++ there are two main dictionary implementations:

- `std::map`
- `std::unordered_map`

Let's look at cppreference.com to see what the documentation says about the complexity of the various operations.

Question

With your neighbour, write down one real-world situation where you would use each of the dictionary implementations.

Dictionary Implementations in C++

In C++ there are two main dictionary implementations:

- `std::map`
- `std::unordered_map`

Let's look at cppreference.com to see what the documentation says about the complexity of the various operations.

Question

With your neighbour, write down one real-world situation where you would use each of the dictionary implementations.

What about {MY_FAVOURITE_LANGUAGE}?

- Python has `dict` (hash map).
- Rust has `std::collections::hash_map::HashMap` (hash map), `std::collections::BTreeMap` (B-Tree).
- JavaScript has `Map` (hash map).
- Java has `HashMap` (hash map), `TreeMap` (Red-Black Tree), and `LinkedHashMap` (hash map with predictable iteration).
- Go has `map` (hash map).

Sets

Sets are just dictionaries without a value associated with each key (only the key matters).

For each of the dictionary data structure implementations in each of the languages we've discussed except Go, there is a corresponding set data structure implementation with equivalent performance characteristics.

- `std::set` and `std::unordered_set` in C++.
- `set` in Python.
- `std::collections::hash_set::HashSet` and `std::collections::BTreeSet` in Rust.
- `Set` in JavaScript.
- `HashSet`, `TreeSet`, and `LinkedHashSet` in Java.

Sets

Sets are just dictionaries without a value associated with each key (only the key matters).

For each of the dictionary data structure implementations in each of the languages we've discussed except Go, there is a corresponding set data structure implementation with equivalent performance characteristics.

- `std::set` and `std::unordered_set` in C++.
- `set` in Python.
- `std::collections::hash_set::HashSet` and `std::collections::BTreeSet` in Rust.
- `Set` in JavaScript.
- `HashSet`, `TreeSet`, and `LinkedHashSet` in Java.

Sorting

Most languages provide built-in sort functions. In general, you will need to read your programming language's documentation to find out what complexity guarantees the sorting functions provide.

- C++: `std::sort`, `std::stable_sort`, and `std::qsort`.
- Python: `sorted` and `list.sort` use Timsort.
- Rust: `Vec::sort` and `slice::sort_unstable`.
- Go: `sort.Sort` and `sort.Stable`. Or with generics: `slices.Sort` and `slices.SortStableFunc`.

Note

Sorting algorithms are like crypto algorithms: you generally should not implement them yourself.

Sorting

Most languages provide built-in sort functions. In general, you will need to read your programming language's documentation to find out what complexity guarantees the sorting functions provide.

- C++: `std::sort`, `std::stable_sort`, and `std::qsort`.
- Python: `sorted` and `list.sort` use Timsort.
- Rust: `Vec::sort` and `slice::sort_unstable`.
- Go: `sort.Sort` and `sort.Stable`. Or with generics: `slices.Sort` and `slices.SortStableFunc`.

Note

Sorting algorithms are like crypto algorithms: you generally should not implement them yourself.