1 Time complexity for containers

When deciding what container to use, we usually think about which of the following operations can be performed and how quickly those operations can be performed:

1. Insertion
2. Deletion
3. Lookup

It depends on the application which of these operations we actually care about however. Also, these operations are purposefully left vague; i.e. for insertion, does one mean they want to be able to insert an element at any given position or just at the end? The actual details of each of these operations determines which data structure we wish to use. Consider the following situations:

- We run a pizza parlor and we take orders from customers, representing each order with a timestamp, and all we wish to know is which order to prepare next. Then we care about (1.), (2.), and (3.): we wish to insert orders into the container (but we don’t worry about where), remove the oldest order, and find the oldest order. In this case a max heap (implemented in the standard library as `std::priority_queue`) is the data structure we should use. A max heap has $O(\log(N))$ insertion, $O(\log(N))$ deletion, and $O(1)$ lookup (all of these operations apply to the max element only).

- We wish to store a dictionary of words with definitions, and we want to be able to quickly find the definition of a given word. Additionally, we wish to be able to insert words into the dictionary as new words are invented. Then we only care about (1.) and (3.). In this case a hash map would be ideal, since it allows for $O(1)$ lookup (and also $O(1)$ insertion and $O(1)$ deletion in the average case). This is implemented in the standard library as `std::unordered_map`.

- Consider the previous case but we also want to be able to determine the next word and the previous word in terms of lexicographical ordering. We still only want to do (1.) and (3.), but now we not only want to lookup a given word, but also we wish to lookup the next and previous words. In this case we would use a binary search tree (specifically a red-black tree would likely be used). This container has $O(\log(N))$ lookup of a general element, $O(1)$ lookup of the next/previous element, and $O(\log(N))$ insertion.

- We are seismologists and we wish to store the locations of earthquakes using logitude and latitude coordinates. Then we want (1.) and (3.). In this case we don’t necessarily care about looking up elements based on any specific criteria, and there is no obvious ordering by which we would like to store values. Here, an array would likely be used, since it allows for $O(1)$ insertion (at the end with preallocation) and $O(1)$ lookup (of an element given a
position). Arrays are familiar to use in the standard library under the name std::vector or std::array.

Unfortunately, there are other factors one needs to consider when deciding upon a data structure. For instance, arrays have \(O(1)\) insertion/deletion at the end (with preallocation) and \(O(1)\) random access (one needs to know the index however), but have \(O(N)\) insertion/deletion for a general position. On the other hand, a list is \(O(1)\) for all of these operations, although it takes \(O(N)\) time to actually find an element one wishes to access. If one never knows the index of an element one wishes to access, which would give the array an advantage, why would one ever use an array over a list? (For it to make sense to even be choosing between these two data structures, we need to assume the data cannot be ordered reasonably or hashed, since in that case we should just use a binary search tree or a hash map). To answer this question, one has to know a bit about how computers actually work. We know that data in a program is stored in the RAM of a computer, but this is only part of the truth. The processor itself has some memory called the cache, which the processor uses to store data as it is performing calculations. It is extremely fast for the processor to access the cache, and slower for the processor to obtain data from the RAM. When an memory is accessed from RAM during the execution of a program and sent to the cache for calculations, surrounding memory is by default also accessed and sent to the cache; thus, because arrays consist of consecutive memory, multiple pieces of data in an array are sent to the cache whenever a single element is accessed. This optimization can speed up certain operations immensely; indeed, consider the following:

```cpp
1 int main() {
2 3 std::vector<double> x = fun();
4 double sum = 0;
5 for (size_t i = 0, sz = x.size(); i < sz; ++i) {
6    sum += x[i];
7 } 
8 }
```

Lastly, when deciding upon a data structure, one needs to think about where the data is stored. It is faster to access stack data than it is to access heap data. Accordingly, it is faster to access data in an std::array than in a std::vector. Of course, there are tradeoffs: an std::array has a fixed length whereas an std::vector has a variable length, etc.

### 2 Summary of containers in the standard library

When listing the time complexities below, we assume for insertion/deletion that we have the position of the element we wish to remove/delete.

1. List-based: data is stored in “nodes,” which are linked together via pointers.
   (a) std::list: doubly-linked list. Pointers exist in both directions between nodes.
   (b) std::forward_list: singly-linked list. Pointers only exist in one direction between nodes.
   (c) Time complexity: \(O(1)\) insertion, \(O(1)\) deletion, \(O(N)\) lookup.

2. Random-access containers: data is stored consecutively in memory.
3. Associative containers: consist of key-value pairs, where each key can only exist once in the collection

(a) **std::set**: data is stored in sorted order by keys. There cannot be copies of data. Keys are the same as the values.
(b) **std::map**: data is stored in sorted order by keys. There cannot be copies of data. Keys are distinct from values.
(c) **std::multiset**: data is stored in sorted order by keys. There can be copies of data. Keys are the same as the values.
(d) **std::multimap**: data is stored in sorted order by keys. There can be copies of data. Keys are distinct from values.
(e) **std::unordered_set**: data is stored by applying a hashing function to keys. There cannot be copies of data. Keys are the same as values.
(f) **std::unordered_map**: data is stored by applying a hashing function to keys. There cannot be copies of data. Keys are distinct from values.
(g) **std::unordered_multiset**: data is stored by applying a hashing function to keys. There can be copies of data. Keys are the same as values.
(h) **std::unordered_multimap**: data is stored by applying a hashing function to keys. There can be copies of data. Keys are distinct from values.

(i) Time complexity: $O(\log(N))$ lookup, insertion, and deletion for the ordered versions. $O(1)$ lookup, insertion, and deletion for the unordered versions. This is just the general case however; if one wishes to lookup the next or previous element starting from a given element, this can be done in $O(1)$ time for the ordered versions, but it is difficult to do for the unordered versions.

4. Others

(a) **std::priority_queue**: allows for quick access to the largest element of the container. Data is usually stored in an **std::vector** or an **std::deque**. $O(1)$ access to the first element. Insertion is $O(\log(N))$ and deletion of the largest element is $O(\log(N))$.
(b) **std::stack**: allows for access, deletion, and insertion at the end of the container; this is called last-in-first-out or LIFO. Access, insertion, and deletion are $O(1)$ because we can only inquire about the top element. Implemented as an **std::vector** usually.
(c) **std::queue**: allows for access and deletion of the top element, and insertion at the bottom; this is called first-in-first-out. Access, insertion, and deletion are $O(1)$. Implemented as an **std::deque** usually.
(d) `std::pair`: stores two pieces of data that can be of different types.

(e) `std: :: initializer_list`: useful for initializing certain data structures, range-based for loops, etc. A lightweight array.