Data Structures for Competitive Programming

This document will start with a high level overview of categories of data structures that are built into C++, Python and Java, talking about the general capabilities of these data structures and the run-times of each of their individual operations. In general, the goal of these built in data structures is to take operations that one would normally run a for loop through a list for (run time O(n) where the list size is n), and avoid that loop due to how the data is organized to achieve better run times. In competitive programming if $n = 10^5$, we can't repeat an O(n) operation n times because for this value of n, $n^2 = 10^{10}$, and for now (as of 2024), that's too many simple operations for a regular computer to carry out in a couple seconds. But, if we can change those O(n) operations to be O(1), O(lg n) or even in some cases $O(\lg^2 n)$, then the resulting program will run fast enough.

For the purposes of this lecture, it's assumed that students have used dynamically sized lists in one of these three languages (C++ = vectors, Python = lists, Java = ArrayList)

Stacks

A stack is a last in, first out data structure, pretty much exactly like a stack of books. You can only add something to the top of a stack or remove what is at the top of the stack. In addition, it's important to be able to check what's at the top of the stack without removing it and what the size of the stack is. All of these operations run in O(1) time for the built in implementations in all three languages.

Stacks are naturally useful for things like tracking matching parentheses or anything with a matching nesting structure. While there is rarely a problem where setting up a stack is the only thing you do, they come in handy as parts of solutions in many contest problems (and this is true of all the built in data structures in this lecture).

Here is a chart outlining these operations, run times (where n is the size of the stack) method names in the three languages:

Operation	Run Time	C++ (stack)	Java (ArrayDeque)	Python (list)
push (add top)	O(1)	push()	push()	append()
pop (del top)	O(1)	pop()	pop()	pop()
top	O(1)	top()	peekLast()	s[-1]
size	O(1)	size()	size()	len(s)

Note: For Python, just use a list as a stack. Thus, for top and size, instead of methods, just use the usual list techniques. For Java, use the class ArrayDeque instead of Stack. In C++ pop is void but in the other two languages the corresponding method returns the appropriate value that got removed.

Queues

A queue is a first in, first out data structure, pretty much exactly like a line at a grocery store (or anywhere else). You can only add something to the back of a queue or remove what is at the front of the queue. In addition, it's important to be able to check what's at the front of the queue without removing it and what the size of the queue is. All of these operations run in O(1) time for the built in implementations in all three languages.

Queues are useful for simulating lines and any other similar process.

Here is a chart outlining these operations, run times (where n is the size of the queue) method names in the three languages:

Operation	Run Time	C++ (queue)	Java (ArrayDeque)	Python (queue)
add back	O(1)	push_back()	offer()	append()
remove front	O(1)	pop_front()	poll()	popleft()
front	O(1)	front()	peekFirst()	q[0]
size	O(1)	size()	size()	len(q)

Note: For Python, use have to do the following import:

```
from collections import deque
```

In C++ pop_front() is void but in the other two languages the corresponding method returns the appropriate value that got removed.

PriorityQueue

A priority queue allows you to add items to it and remove either the minimum or maximum item from it (depending on the language). Java and Python implementations do minimum queues (remove minimum) while C++'s implementation is of a maximum queue (remove maximum). The run time of each of these operations is O(lg n), when there are n items in the Priority Queue.

Priority Queues are very useful for simulations where things are getting added but the thing you want to remove is always the "most important." (Think of people waiting in a line but when there is room in the club, the most important celebrity gets in, not the person who has waited the longest.) Often times, these tend to be useful in implementing greedy algorithms where we want to maintain a list of active objects, but when the time comes, we want to "pull the best one" from those active objects. Here is a chart outlining these operations, run times (where n is the size of the priority queue) method names in the three languages:

Operation	Run Time	C++	Java	Python
Class Name		priority_queue	PriorityQueue	heapq
add	O(lg n)	push()	offer()	heappush()
del min/max	O(lg n)	pop()	remove()	heappop()
ret min/max	O(1)	top()	peek()	h[0]
size	O(1)	size()	size()	len(h)

Note: For Python, use have to do the following import:

```
import heapq
```

Also in Python, a heap is just a list that you pass into the heappush and heappop methods. When you call those methods you must call them as heapq.heappush(). In C++ pop() is void but in the other two languages the corresponding method returns the appropriate value that got removed.

Unordered Sets

An unordered set is a collection of objects without duplicates. You can add items to a set, delete items from a set, search for items in a set and get the set's size. (You can also iterate through the items in a set efficiently.) The expected run time for the single operations is O(1). If you add an item to a set that's already in the set, no change is made to the set.

Sets are useful for many tasks, essentially when duplicates must be ignored or when you need to just have a list of each unique item you've seen in a list that might have duplicates.

Here is a chart outlining these operations, run times (where n is the size of the set) method names in the three languages:

Operation	Run Time	C++ (unordered_set)	Java (HashSet)	Python (set)
add	O(1)	insert()	add()	add()
remove	O(1)	erase()	remove()	remove()
find	O(1)	find()	contains()	In
size	O(1)	size()	size()	len(s)

Unordered Maps

An unordered map is just like an unordered set except that each item in the set (called a key) is mapped to a corresponding value. It's like a look up table or a dictionary or an array that you can index with any type of item, not just integers from 0 to sizd-1. In Python, these are called dictionaries and are built in without a class. In C++ and Java, the corresponding classes are unordered_map and HashMap.

Maps are extremely useful any time we need to make a look up table or define a function of some sort with an arbitrary domain. A quick common example is a shortest distance problem where each of the possible locations are cities. We want to create a map mapping each city name to a unique identifier from 0 to n – 1, where there are n cities so that we can then use the indexes 0 to n – 1 to refer to those cities elsewhere in our algorithm. We can store the number to city list in a regular list (index 0 stores city name 0) and we store the reverse look up in an unordered map. Here are some operations we would like to do with a map:

add an entry, remove an entry, look up the associated value for a key, get the size of the map

Here is a chart outlining these operations, run times (where n is the size of the map) method names in the three languages:

Operation	Run Time	C++ (unordered_map)	Java (HashMap)	Python
add	O(1)	insert()	put()	m[x] = y
remove	O(1)	erase()	remove()	del m[x]
get value	O(1)	m[x]	get()	m[x]
size	O(1)	size()	size()	len(m)

Ordered Sets

Note: Python does not provide an ordered set, only C++ and Java do. This is a drawback to using Python in competitive programming exclusively.

In addition to all of the regular set operations, since an ordered set is "ordered" you can iterate through the elements in order, and more importantly, for a particular value, you can find the next larger or next smaller element in the set. You can also retrieve the minimum and maximum elements easily. All of this functionality can be extremely useful for solving many problems.

We'll look at the relevant methods below with their run times:

Operation	Run Time	C++ (set)	Java (TreeSet)
add	O(lg n)	insert()	add()
ceiling/higher	O(lg n)	lower_bound(), upper_bound()	ceiling(), higher()
find	O(lg n)	find()	contains()
first/last	O(lg n)	begin(), end()	first(), last()
floor/lower	O(lg n)		floor(), lower()
pollFirst/pollLast	O(lg n)		pollFirst(), pollLast()
remove	O(lg n)	erase()	remove()
size	O(1)	size()	size()

In C++ one must utilize pointers to fully make use of these methods as many of the methods return pointers to the appropriate elements. In Java, pointers are not accessible to the user so these methods tend to be a little easier to use. Also, C++ doesn't have equivalent methods for floor, lower, pollFirst or pollLast. The latter two methods are easy to duplicate by getting a pointer to the appropriate item and then calling the erase method.

Ordered Maps

Note: Python does not provide an ordered map, only C++ and Java do. This is a drawback to using Python in competitive programming exclusively.

An ordered map provides the same functionality as an ordered set, but in addition to all the above items for dealing with keys, associated values are stored with each key and can be retrieved.

In C++, we can think of an ordered map as an ordered set of pairs where the first item is the key and the second item is the matching value.

In Java, we get the combination of the functionality of both the HashMap and the TreeSet.

On the following page is a list of methods provided by each language for their ordered map:

Operation	Run Time	C++ (map)	Java (TreeMap)
ceilingKey/higherKey	O(lg n)	lower_bound(), upper_bound()	ceilingKey(), higherKey()
containsKey	O(lg n)	find()	containsKey()
firstKey/lastKey	O(lg n)	begin(), end()	firstKey(), lastKey()
floorKey/lowerKey	O(lg n)		floorKey(), lowerKey()
get		m[x]	get()
put	O(lg n)	insert()	put()
remove	O(lg n)	erase()	remove()
size	O(1)	size()	size()

Just like the ordered set, C++ doesn't have an equivalent method to Java's floorKey or lowerKey.

Note to the Reader

Ultimately, there have been many details left out of these notes because having detailed examples with each method call for all three languages would have yielded a very unwieldy document. The key is for users to get used to carefully reading the documentation in your desired language and then practice writing some sample code to get use to the proper way to use these built in data structures. Beyond the method calls there are key syntactic idiosyncrasies that only get discovered once you get your hands dirty and try things out. This overview is meant to provide the capabilities and corresponding run-times of each of these built in tools, which would normally take hundreds of lines to write from scratch, but are ready at your fingertips for your use.

I do have separate notes for Java (covers much of this in detail) and C++ (just for sets and maps), and can provide those notes upon request. I do believe that the true power of these built in data structures isn't fully appreciated until one uses them several times to implement solutions to problems. Both the full adoption and familiarity with these data structures takes time, but is well worth the investment.