Java Data Structures

Interview Guide

corejava interview questions.com

DATA STRUCTURES

Java data structures interview questions will come up in your interview. At it's nexus being a programmer is about handling data, transforming and outputting it from one format to another, which is why it is so important to understand what's going on under the hood and how it can affect your application. It never fails to confound me how many people can't tell what collection is used to back an ArrayList (hint: it's in the name). From the interviewer's perspective, questions on data structures reveal a lot of information about the candidate. They show if the candidate has a core understanding of how java works. Even better it provides a platform to lead to a wide ranging set of questions on design, algorithms and a whole lot more. In day to day java programming, you're going to spend most of your time with ArrayLists, LinkedLists and HashMaps. As a result it makes sense to review collections and algorithms you haven't looked at for a while as, in my experience, companies love asking this sort of question.

- **Q: What is the Java Collection Framework?**
- **Q: What Collection types are there in Java?**
- **Q: What are the basic interface types?**
- **Q: What are the differences between List, Set and Map?**

This is the bread and butter stuff that everyone who uses Java should know. The collections framework is simply the set of collection types that are included as part of core java. There are three types of collection you will deal with in Java; Set and List (both of which implement Collection) and Maps (which implement Map and technically aren't part of core Collections). Each type of collection exhibits certain characteristics which define it's usage. The key features of a collection are:

- Elements can be added or removed
- The collection will have a size that can be queried
- The collection may or may not contain duplicates
- It may or may not provide ordering
- It may or may not provide positional access (e.g. get me item at location 6)

The behaviour of these methods varies based on implementation (as you would expect from an interface). What happens on if you call *remove()* on a collection for an object that doesn't exist? It depends on the implementation.

The collection types

Set: A set has no duplicates and no guaranteed order. Because of this, it does not provide positional access. Implements Collection. Example implementations include TreeSet and HashSet.

List: A list may or may not contain duplicates and also guarantees order, allowing positional access. Implements Collection. Example implementations include ArrayList and LinkedList.

Map: A map is slightly different as it contains key-value pairs as opposed to specific object. The key of a map may not contain duplicates. A map has no guaranteed order and no positional access. Does not implement Collection. Example implementations include HashMap and TreeMap.

But what does this mean with regards to interviews? You're almost certainly going to get a question about the differences between the types so make an effort to learn them. More importantly the smart interviewee understands what the implication of these features are. Why would you choose one over the other? What are the implications?

Which to choose?

In reality, whether you choose a Set, List or Map will depend on the structure of your data. If you won't have duplicates and you don't need order, then your choice will lie with a set. If you need to enshrine order into your data, then you will choose List. If you have key-value pairs, usually if you want to associate two different objects or an object has an obvious identifier, then you will want to choose a map.

Examples

- A collection of colours would be best put into a set. There is no ordering to the elements, and there are no duplicates; You can't have two copies of "Red"!
- The batting order of a cricket team would be good to put in a list; you want to retain the order of the objects.
- A collection of web sessions would be best in a map; the unique session ID would make a good key/reference to the real object.

Understanding why you choose a collection is vitally important for giving the best impression in interviews. I've had candidates come in and tell me they just use ArrayList and HashMap because that's just the default implementation they choose to use. In reality, this maybe isn't a bad thing. Most times we need to use

collections there are no major latency or access requirements. It doesn't matter. However, people who are asking you these questions in interviews want to know that you understand how things work under the covers.

Q: What are the main concerns when choosing a collection?

When you're handling a collection, you care about speed, specifically

- Speed of access
- Speed of adding a new element
- Speed of removing an element
- Speed of iteration

On top of this, it isn't just a case of "which is faster". There is also consistency. Some implementations will guarantee the speed of access, whereas others will have variable speed. How that speed is determined depends on the implementation. The important thing to understand is the speed of the relative collection implementations to each other, and how that influences your selection.

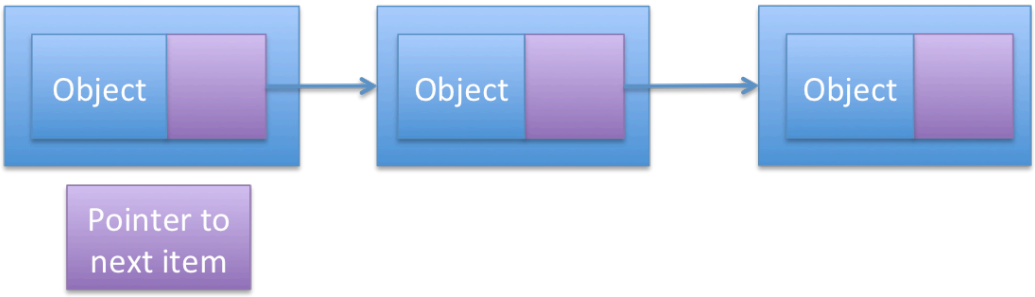
Collection implementations

- **Q: Why would I choose LinkedList over an ArrayList?**
- **Q: Which is faster, TreeSet or HashSet?**
- **Q: How does a LinkedList work?**

The collection implementations tend to be based on either a *Linked* implementation, a *Tree* implementation or a *Hash* implementation. There are entire textbooks based on this, but what you need to know is how it affects your collection choices.

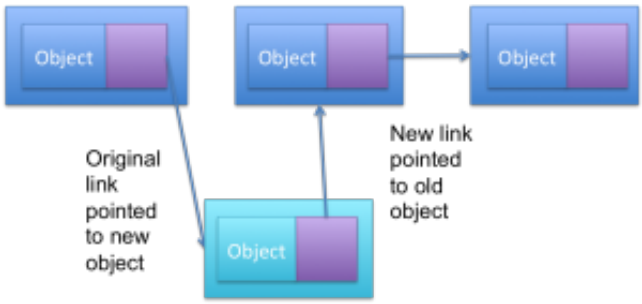
Linked implementation

Example: LinkedList, LinkedHashSet Under the covers, each item is held in a "node". Each node has a pointer to the next item in the collection like so.



What does this mean for speed?

When adding an item into a linked connection it's quick, irrelevant of where you're adding the node. If you have a list of 3 items and want to add a new one in position 1, the operation is simply to point position 2 to the new item, and the new item to the old position 2.



That's pretty fast! No copying collections, no moving things, no reordering. The same is true for removals; simply point node 0 to node 2 and you've removed node 1.

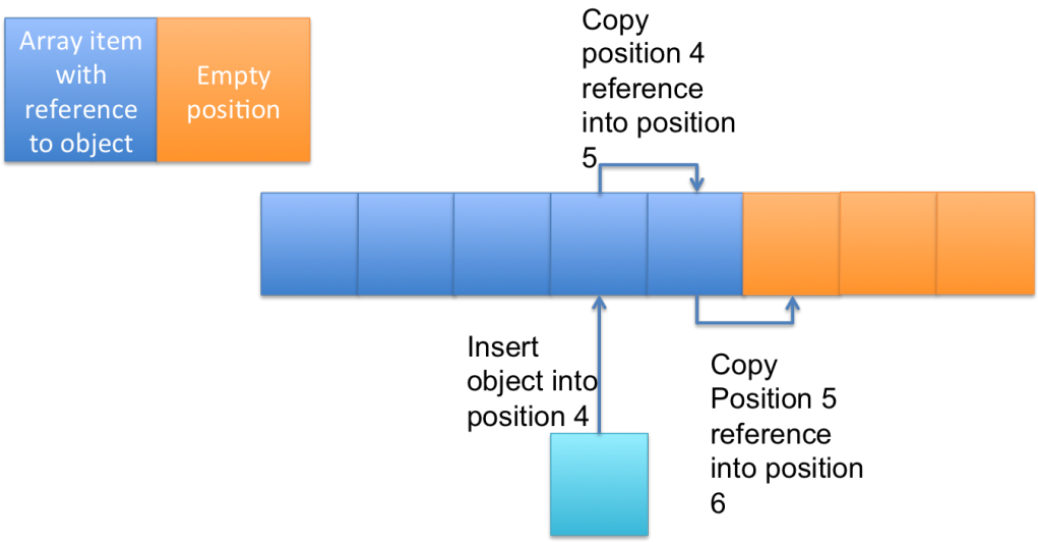
Conversely when accessing objects it is relatively slow. If you have a list of 1000 items, and you want item 999, you need to traverse every single node to get to that item. This also means that the access speed is not consistent. It's quicker to access element 1 than it is element 1000. Bear in mind that when adding a new object you need to first traverse to the node before, so this can have some impact on speed.

So linked collections such as LinkedList are great when you need fast addition/removal and access time is less of a concern. **If you're going to be adding/remove items from your collection a lot, then this is the option for you**.

Array implementation

Example: ArrayList. ArrayList is the only Array based implementation in the collection classes, but is often used to compare to LinkedList so it's important to understand. As alluded to previously, ArrayLists are backed by an Array. This has interesting implications.

When adding an item into the middle of an ArrayList, the structure needs to copy all of the items to shift them down the Array. This can be slow, and is not guaranteed time (depending on how many elements need to be copied.



The same applies when removing objects; all object references after it have to be copied forward one space. There is an even worse case. On creating an ArrayList it starts with a fixed length Array (whose length you can set in the constructor, or the default is 10). If the capacity needs to increase over this, the collection has to create a **brand new array** of greater length and copy all the references to it which can be really slow.

Where ArrayList succeeds is access speed; as the array is in contiguous memory it means that if you request object 999 of 1000, you can calculate exactly where in memory the reference is with no need to traverse the full collection. This gives constant time access which is **fast**.

So, ArrayLists make a great choice if you have a set of data that is **unlikely to be modified significantly, and you need speedy read access.**

Hash implementation

Examples: HashSet, HashMap. The HashMap is a complicated beast in itself and is a good choice for an interview question as it is easy to add extending questions to.

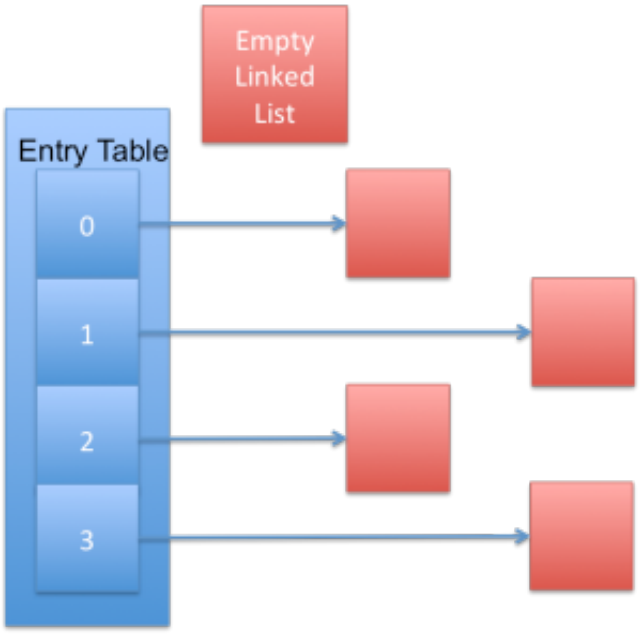
Q: How does a HashMap work?

Q: Why does the hashcode of the key matter?

Q: What happens if two objects have the same HashCode?

Q: How does the Map select the correct object if there is a clash?

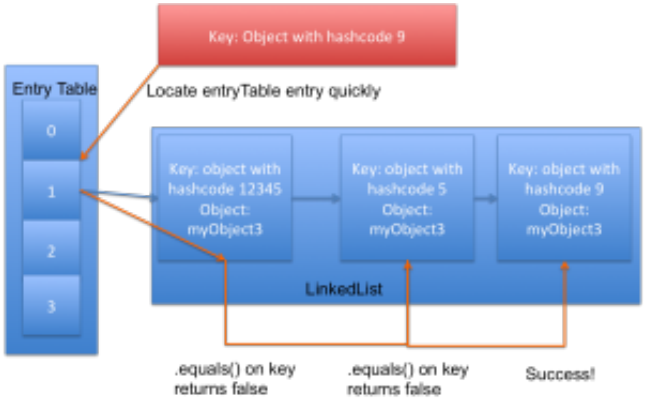
Interestingly HashSet is backed by HashMap, so for the purpose of this I will just discuss the latter. A HashMap works like this: under the covers a HashMap is an array of references (called the Entry Table, which defaults to 16 entries) to a group of LinkedLists (called Buckets) where HashEntries are stored. A HashEntry is an object containing the key and associated object.



All instances of Object have a method, hashCode(), which returns an int. The hashcode is usually a value derived from the properties of an object. The hashcode returned from an object should be consistent and equal objects must return the same hashcode. This property can then be used to determine which bucket to put it in based on the index of the entry table; for example, in the above example with 4 spaces in the entry table, if I have a key in with HashCode 123456, I can do 12345 % 4 = 1, so I would place my object in bucket 1.

This makes object retrieval very quick. In my example, if I try to retrieve the object for my key then all I need to do is the same mod calculation to find the bucket and it's the first entry. Exceptionally quick.

This can get more complicated. What if I have two more objects, HashCode of "5" and "9"? They both have a (modulus 4) of 1 so would both be put into bucket 1 too. Now when I use my "9" key to retrieve my object the HashMap has to iterate through a LinkedList, calling .equals() on the keys of each HashEntry.



This is slow! Fortunately, this shouldn't happen often. Although collisions will naturally occur, the intention is that objects will be spread evenly across the entry table with minimal collisions. This relies on a well implemented hashcode function. If the hashcode is poor (e.g., always return 0) then you will hit collisions.

To further try and reduce the chance of this, each HashMap has a load factor (which defaults to 0.75). If the entry table is more full than the load factor (e.g. if more than 75% of the entry table has entries) then, similar to the ArrayList, the HashMap will double the size of the entry table and reorder the entries. This is known as rehashing.

Tree implementation

Examples: TreeMap, TreeSet. The Tree implementation is completely different to any of the data structures we've considered before and is considerably more complicated. It is based on a Red Black tree, a complicated algorithm which you're unlikely to get asked about (if you're really keen then watch http://www.csanimated.com/animation.php?t=Red-black_tree). Values are stored in an ordered way, determined either by the natural ordering of the object or using a **Comparator** passed in as part of the constructor. This means on all implementations the collection will re-sort itself to maintain this natural order. When retrieving items it is necessary to navigate the tree, potentially traversing a

large number of nodes for sizeable collections. The important thing to note from your perspective: **this is slower than HashMap.** Considerably slower. If speed is the goal then always use a HashMap or HashSet. However, if ordering is important then TreeMap/TreeSet should be your collection of choice.

Multithreading and collections

Collections and Threading are intrinsically linked; many of the core problems related to Threading are regarding shared data access. How can we access data from multiple threads in a fast yet safe fashion?

How do I create a Thread Safe collection?

The easiest way to make something threadsafe if to make it immutable. An immutable object is one whose state cannot be altered. If it cannot change then it is safe for all threads to read it without any problems. This can be achieved in Java using the Collections.unmodifiable methods.

```
Collections.unmodifiableCollection
Collections.unmodifiableSet
Collections.unmodifiableSortedSet
Collections.unmodifiableList
Collections.unmodifiableCollection
Collections.unmodifiableMap
Collections.unmodifiableSortedMap
```

All of these collections act as a wrapper around a standard collection and guarantee access safety; any attempts to modify the collection will result in an UnsupportedOperationException. However, this is only a *view* on the data. If everyone is accessing via the unmodifiable version, then this is thread safe; however the underlying collection could still change. For example:

```
LinkedList<String> words = new LinkedList<>();
words.add("Hello");
words.add("There");
List<String> unmodifiableList =
Collections.unmodifiableList(words);
System.out.println(unmodifiableList);
```

```
words.add("Cheating");
System.out.println(unmodifiableList);
//result:
//[Hello, There]
//[Hello, There, Cheating]
```

As a result this isn't really immutable. Such things as a truly ImmutableList do exist in third party libraries such as Google Guava, but not in the core language. It's also important to note that immutable collections don't normally satisfy the needs of our programs; often data needs to be written and read, so we need a bigger solution.

In Java 5, the java.util.concurrent package was introduced with the specific aim of making life easier for dealing with collections in a thread safe way.

Q: Tell me about the java.util.concurrent package that was introduced in Java 5. Why is it important? Why does it matter?

The guys who write Java are smarter than you or me (or they should be). Instead of everyone having to write their own implementation to ensure thread safety, making sure the right things were locked in the right places without destroying performance, java.util.concurrent provides a number of collection types which guarantee thread safety with the minimum performance impact.

It also introduced a bunch of other cool Thread related features, which will be covered in the Threading chapter.

The queues

- ConcurrentLinkedDeque
- ConcurrentLinkedQueue
- LinkedBlockingQueue
- LinkedBlockingDeque
- LinkedTransferQueue
- PriorityBlockingQueue
- ArrayBlockingQueue

That's a lot of queues (and double ended queues, a deque)! The key things to note:

- A Blocking queue will block a consumer/producer when the queue is empty/full. This is a necessary mechanism to allow a producer/consumer pattern. The

BlockingQueue interface has put() and take() methods to allow for this.
- A Transfer queue is a blocking queue, with extra methods to allow a consumer to wait for a message to be consumed.
- A Priority queue will order elements based on their natural ordering or using a comparator passed in at construction time.

All of the implementations use Compare and Set (CAS) operations, not synchronized blocks to ensure thread safety. This ensures no risk of deadlock and Threads do not get stuck waiting, and is also significantly faster. All collections based on CAS operations, such as the above, have weakly consistent iterators. This means that they will not throw ConcurrentModificationExceptions but they make no guarantee to reflect the correct state; from the documentation:

The returned iterator is a "weakly consistent" iterator that will never throw ConcurrentModificationException and guarantees to traverse elements as they existed upon construction of the iterator, and may (but is not guaranteed to) reflect any modifications subsequent to construction.

The CopyOnWrites

Q: How can ConcurrentModificationException be avoided when iterating a collection?

- CopyOnWriteArrayList
- CopyOnWriteArraySet

I'm a big fan of well named Classes, and this is a great example. By copying the entire collection on every write it guarantees immutability; if I have an iterator looping through a collection then it can guarantee that the collection will not be modified as it is being traversed as all modifications will result in a copy of the data being created. This can be an expensive operation so these collections lend themselves better to data that has frequent access but infrequent update.

An iterator which relies on the data being copied, such as in this case, is known as a fail safe iterator.

ConcurrentHashMap

Q: How is ConcurrentHashMap implemented?

ConcurrentHashMap is a highly efficient, thread safe implementation of HashMap. As discussed earlier, a HashMap is made up of segments, a number of lists (defaulting to 16) which contain the data which are accessed based on HashCode.

In ConcurrentHashMap read operations are generally not blocking which allows for it to be very speedy; writes only block by segment. This means that the other 15 segments can be read without issue if an update is being written.

Q: Should I use a HashTable or a ConcurrentHashMap?

HashTable is a synchronized implementation of HashMap but it is from an old version of Java and should no longer be used. All operations are synchronized in HashTable which means it's performance is very poor relative to ConcurrentHashMap.

Q: Are collections such as ConccurrentLinkedQueue and ConcurrentHashMap completely Thread safe?

The answer is both yes and no. The implementation is completely ThreadSafe and can be used as a replacement for their non Thread safe counterparts. However, it is very possible to write code which access the collection in a non Thread safe manner.

```
Map<String, String> map = new ConcurrentHashMap<>();
map.put("a", "1");
map.put("b", "2");
//....Set off a bunch of other threads....
if (!map.containsKey("c"))
    map.put("c", "3");
else
    map.get("c");
```

The contains -> put and contains -> get operations are not atomic. Between checking for existence and putting/getting anything can happen in the collection. The collection is still thread safe, our access simply isn't. We need to ensure any operations where we check state and then execute are done in an atomic or synchronized way. The above code could be rectified by making use of the putIfAbsent method to create atomicity; however if for example you wanted to perform an action to a list based on it's size you may need to lock the collection to ensure it happens as a single operation.