In any desktop application, actions can be recorded in linked list and implement Undo and Redo function iterating from last.

Browser’s Next and Previous buttons can be programmed using linkedlist.

Linked Lists (paired with a hashtable) are really useful for LRU Caches.

**ArrayList vs LinkedList**

Let’s list down few notiable **differences between arraylist and linkedlist**.

* ArrayList is implemented with the concept of dynamic resizable array. While LinkedList is a doubly linked list implementation.
* ArrayList allows random access to it’s elements while LinkedList does not.
* LinkedList, also implements Queue interface which adds more methods than ArrayList, such as offer(), peek(), poll(), etc.
* While comparing to LinkedList, [ArrayList](https://howtodoinjava.com/java-arraylist/" \t "_blank) is slower in add and remove, but faster in get, because there is no need of resizing array and copying content to new array if [array](https://howtodoinjava.com/java-array/) gets full in LinkedList.
* LinkedList has more memory overhead than ArrayList because in ArrayList each index only holds actual object but in case of LinkedList each node holds both data and address of next and previous node.

**Java HashMap Features**

**Key notes on internal working of HashMap**

1. Data structure to store entry objects is an array named table of type Entry.
2. A particular index location in array is referred as bucket, because it can hold the first element of a linkedlist of entry objects.
3. Key object’s hashCode() is required to calculate the index location of Entry object.
4. Key object’s equals() method is used to maintain uniqueness of keys in map.
5. Value object’s hashCode() and equals() method are not used in HashMap’s get() and put() methods.
6. Hash code for null keys is always zero, and such entry object is always stored in zero index in Entry[].
7. Key’s hash code is used primarily in conjunction to its equals() method, for putting a key in map and then getting it back from map. So, our only focus point is these two methods. So if hash code of key object changes after we have put a key value pair in map, then its almost impossible to fetch the value object back from map. It is a case of memory leak.
8. For above basic reasoning, key objects are suggested to be **IMMUTABLE**. Immutability allows you to get same hash code every time, for a key object.

Equal objects must produce the same hash code as long as they are equal, however unequal objects need not produce distinct hash codes.” i.e.

1. Whenever a.equals(b) is true, then a.hashCode() must be same as b.hashCode().
2. Whenever a.equals(b) is false, then a.hashCode() may/may not be same as b.hashCode().

entify few differences between both version of maps so that we can decide which one to choose in which condition.

1. Multiple threads can add/remove key-value pairs from ConcurrentHashMap, while only one thread is allowed to make change in case of SynchronizedHashMap. This results higher degree of concurrency in ConcurrentHashMap.
2. No need to lock the map to read a value in ConcurrentHashMap. A retrieval operation will return the value inserted by the most recent completed insert operation. A lock is required for read operation too in SynchronizedHashMap.
3. ConcurrentHashMap doesn’t throw a ConcurrentModificationException if one thread tries to modify it while another is iterating over it. The iterator reflects the state of the map at the time of it’s creation. SynchronizedHashMap returns Iterator, which fails-fast on concurrent modification.
4. This is the main reason why immutable classes like String, Integer or other wrapper classes are a good key object candidate. and it is the answer to question why string is popular hashmap key in java?
5. The principal idea is that **once the number of items in a hash bucket grows beyond a certain threshold, that bucket will switch from using a linked list of entries to a balanced tree. In the case of high hash collisions, this will improve worst-case performance from O(n) to O(log n)**.
6. Basically when a bucket becomes too big (**currently: TREEIFY\_THRESHOLD = 8**), HashMap dynamically replaces it with an ad-hoc implementation of the treemap. This way rather than having pessimistic O(n) we get much better O(log n).

* HashMap cannot contain duplicate keys.
* HashMap allows multiple null values but only one null key.
* HashMap is an **unordered collection**. It does not guarantee any specific order of the elements.
* HashMap is **not thread-safe**. You must explicitly synchronize concurrent modifications to the HashMap. Or you can use **Collections.synchronizedMap(hashMap)** to get the synchronized version of HashMap.
* A value can be retrieved only using the associated key.
* HashMap stores only object references. So primitives must be used with their corresponding wrapper classes. Such as int
* will be stored as Integer.
* HashMap implements **Cloneable** and **Serializable** interfaces.
* All instances of Entry class are stored in an array declard as 'transient Entry[] table'. For each key-value to be stored in HashMap, a hash value is calculated using the key’s hash code. This hash value is used to calculate the **index** in the array for storing Entry object.
* In case of **collision**, where multiple keys are mapped to single index location, a **linked list** of formed to store all such key-value pairs which should go in single array index location.
* While retrieving the value by key, first index location is found using key’s hashcode. Then all elements are iterated in the linkedlist and correct value object is found by identifying the correct key using it’s **equals()** method.
* “Hash function should **return the same hash code each and every time** when the function is applied on same or equal objects. In other words, two equal objects must produce the same hash code consistently.”
* All objects in Java inherit a default implementation of hashCode() function defined in Object class. This function produces hash code by typically converting the internal address of the object into an integer, thus producing different hash codes for all different objects.
* Before going into put() method’s implementation, it is very important to learn that instances of Entry class are stored in an array. HashMap class defines this variable as:

Let’s note down the internal working of put method in hashmap.

1. First of all, the key object is checked for null. If the key is null, the value is stored in table[0] position. Because hashcode for null is always 0.
2. Then on next step, a hash value is calculated using the key’s hash code by calling its hashCode() method. This hash value is used to calculate the index in the array for storing Entry object. JDK designers well assumed that there might be some poorly What if we add the another value object with same key as entered before. Logically, it should replace the old value. How it is done? Well, after determining the index position of Entry object, while iterating over linkedist on calculated index, HashMap calls equals method on key object for each entry object.

All these entry objects in linkedlist will have similar hashcode but equals() method will test for true equality. If key.equals(k) will be true then both keys are treated as same key object. This will cause the replacing of value object inside entry object only.

**In this way, HashMap ensure the uniqueness of keys**.

1. Now indexFor(hash, table.length) function is called to calculate exact index position for storing the Entry object.