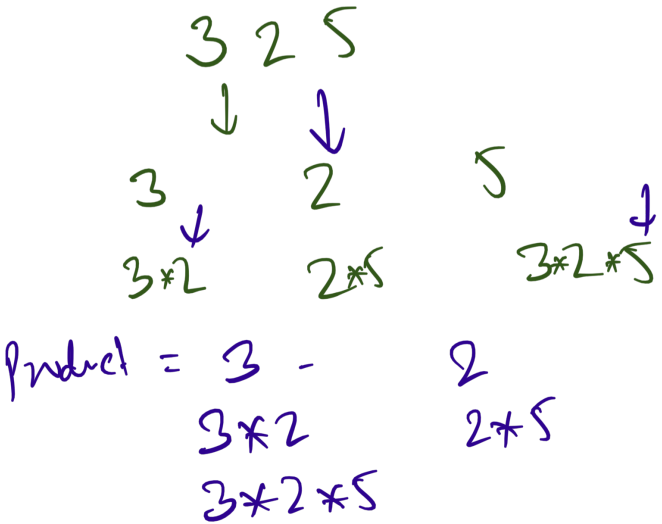
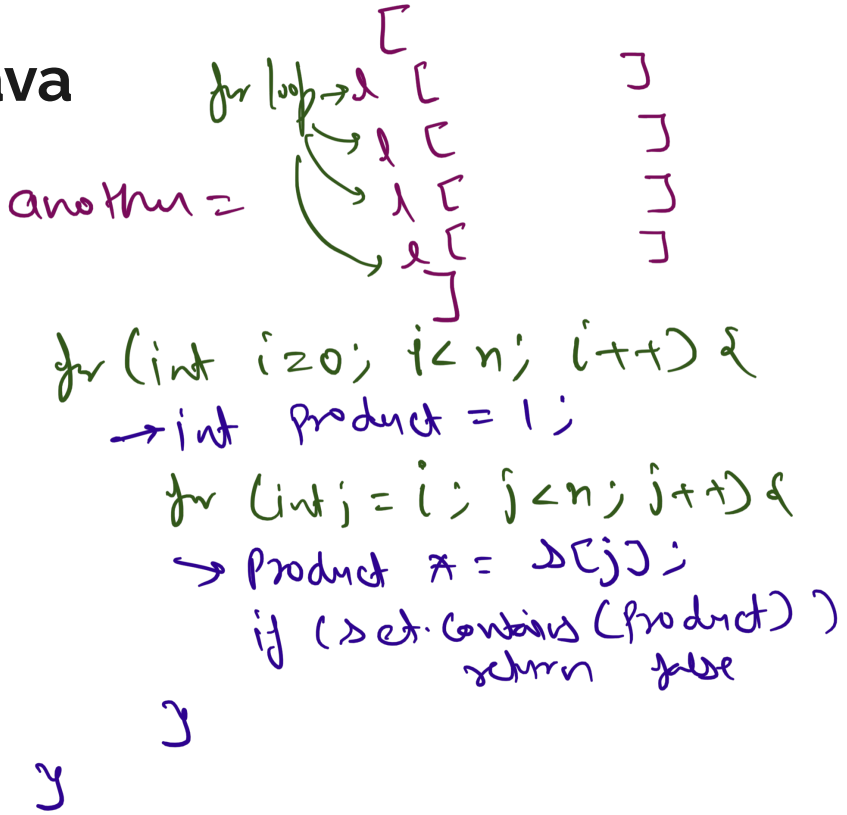
Hashing & HashMap Basics

HashSet and HashMap in Java





hashCode and equals contract

- During the execution of the application, if hashCode() is invoked more than once on the same Object then it must consistently return the same Integer value
- If two Objects are equal, according to the equals(Object) method, then hashCode() method must produce the same Integer on each of the two Objects.
- If two Objects are unequal, according to the equals(Object) method, It is not necessary the Integer value produced by hashCode() method on each of the two Objects will be distinct.

Count Distinct Elements

Frequency of Elements in an Array

Pair with given sum in an Unsorted Array

Zero Sum Subarray

Practice Problems

1. Subarray with given Sum.
2. Intersection of Two Arrays.
3. Union of Two Arrays.
4. Find the largest subarray with zero sum.
5. Count distinct elements in every window of size k.

Hashing Problems - I

Largest Subarray with zero sum

Longest Consecutive Subsequence

Find any quadruple whose sum is equal to the given sum.

Practice Problems

1. Longest Substring without repeat.
2. https://www.interviewbit.com/problems/anagrams/
3. https://www.interviewbit.com/problems/colorful-number/
4. https://www.interviewbit.com/courses/programming/hashing

Internal Working of HashMap

Hashing

- The process of assigning a unique value to an object or attribute using an algorithm, which enables quicker access, is known as hashing.
- It’s necessary to write the hashCode() method properly for better performance of HashMap.
- hashCode in Java is a function that returns the hashcode value of an object on calling. It returns an integer or a 4 bytes value which is generated by the hashing algorithm.

hashCode and equals contract

- During the execution of the application, if hashCode() is invoked more than once on the same Object then it must consistently return the same Integer value
- If two Objects are equal, according to the equals(Object) method, then hashCode() method must produce the same Integer on each of the two Objects.
- If two Objects are unequal, according to the equals(Object) method, It is not necessary the Integer value produced by hashCode() method on each of the two Objects will be distinct.

Hashing of Strings

```
int hash = 7;
for (int i = 0; i < strlen; i++) {
    hash = hash*31 + charAt(i);
}
```

Hashing of Custom Class Objects

```java
class Boy {
    String name;
    int age;

    List<Integer> marks;

    @Override
    public int hashCode() {
        return Objects.hash(age, marks, name);
    }
```

```java
public static int hashCode(Object[] a) {
    if (a == null)
        return 0;

    int result = 1;

    for (Object element : a)
        result = 31 * result + (element == null ? 0 : element.hashCode());

    return result;
}
```

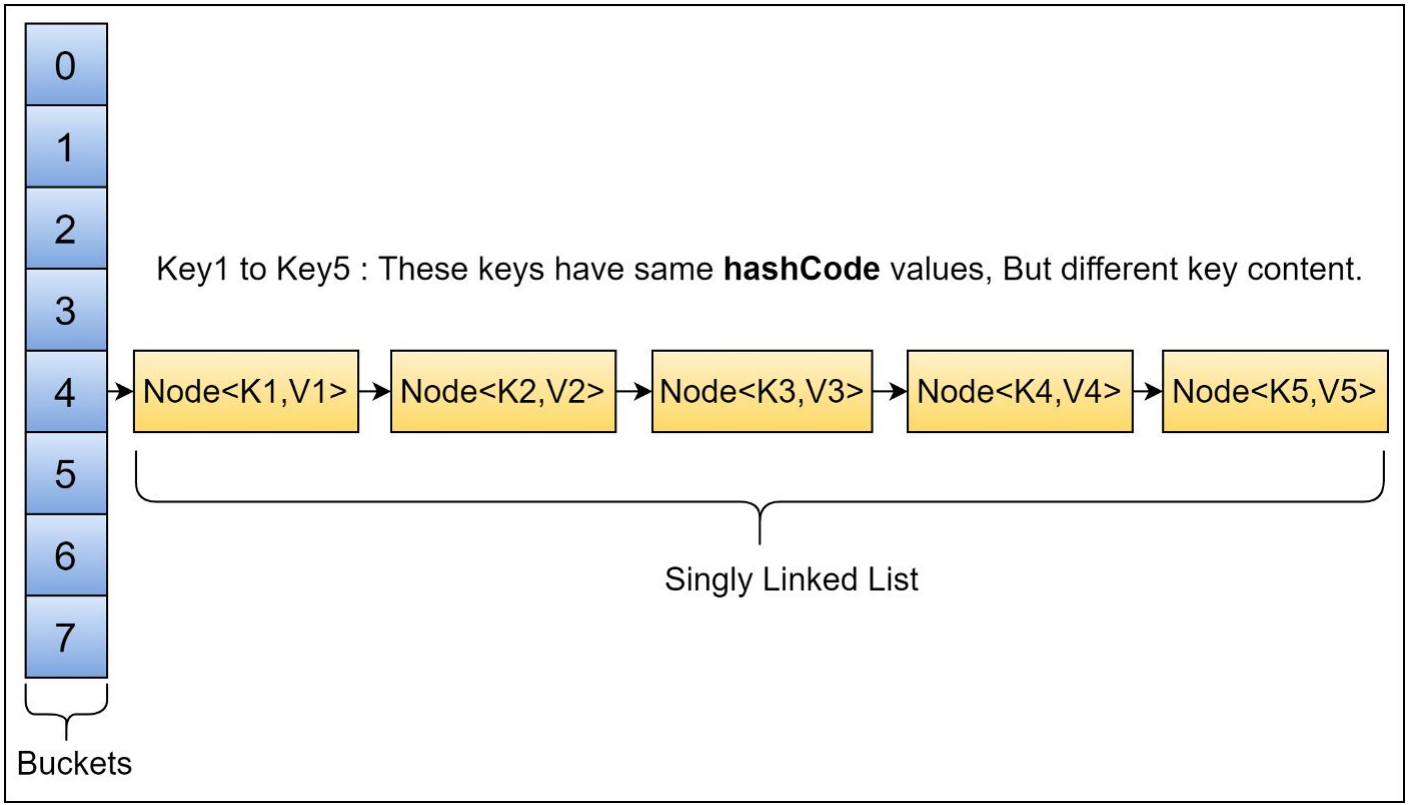
Calculating Index from Hashcode

index = hashCode(key) & (n-1).

Where n is equal to the number of buckets.

Internal Java HashMap

1. HashMap uses its static inner class Node<K,V> for storing map entries. That means each entry in hashMap is a Node. Internally HashMap uses a hashCode of the key Object and this hashCode is further used by the hash function to find the index of the bucket where the new entry can be added.
2. HashMap uses multiple buckets and each bucket points to a Singly Linked List where the entries (nodes) are stored.
3. Once the bucket is identified by the hash function using hashcode, then hashCode is used to check if there is already a key with the same hashCode or not in the bucket(singly linked list).
4. If there already exists a key with the same hashCode, then the equals() method is used on the keys. If the equals method returns true, that means there is already a node with the same key and hence the value against that key is overwritten in the entry(node), otherwise, a new node is created and added to this Singly Linked List of that bucket.
5. If there is no key with the same hashCode in the bucket found by the hash function then the new Node is added to the bucket found.


0
1
2
3
4
5
6
7
Buckets
Key1 to Key5 : These keys have same **hashCode** values, But different key content.
Node<K1,V1>
Node<K2,V2>
Node<K3,V3>
Node<K4,V4>
Node<K5,V5>
Singly Linked List

Load factor & Rehashing

Whenever the number of entries in the hashmap crosses the threshold value then the bucket size of the hashmap is doubled and **rehashing** is performed and all already existing entries(nodes) of the map are copied and new entries are added to this increased hashmap.

```
Threshold value = Bucket size * Load factor
```

Eg. If the bucket size is 16 and the load factor is 0.75 then the threshold value is 12.

```
1         Map<String, Long> myPhoneBook = new HashMap<>();
2         //Bucket size 16 and LD=0.75
3         Map<String, Long> myPhoneBook = new HashMap<>(64);
4         //Bucket size 64 and LD=0.75
5         Map<String, Long> myPhoneBook = new HashMap<>(128, 0.90f);
6         //Bucket size 16 and LD=0.9
7         Map<String, Long> myPhoneBook = new HashMap<>(parentsPhoneBook);
8         //Bucket copy the parentsPhoneBook to myPhoneBook
```

Time Complexity

Before java 8, singly-linked lists were used for storing the nodes. But this implementation has changed to self-balancing BST after a thresold is crossed (static final int TREEIFY_THRESHOLD = 8;). The motive behind this change is that HashMap buckets normally use linked lists, but for the linked lists the worst-case time is O(n) for lookup.

- In a fairly distributed hashMap where the entries go to all the buckets in such a scenario, the hashMap has O(1) time for search, insertion, and deletion operations.
- In the worst case, where all the entries go to the same bucket and the singly linked list stores these entries, O (n) time is required for operations like search, insert, and delete.
- In a case where the threshold for converting this linked list to a self-balancing binary search tree(i.e. AVL/Red black) is used then for the operations, search, insert and delete O(log(n)) is required as AVL/Red Black tree has a max length of log(n) in the worst case.

Linked List Basics

Linked List

A linked list is a linear data structure that includes a series of connected nodes. Here, each node stores the data and the address of the next node. For example,

Linked List vs Array

ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

Traverse in a Linked List

Insert in a Linked List

Delete in a Linked List

Find the Middle Element in a Linked List

Types of Linked Lists

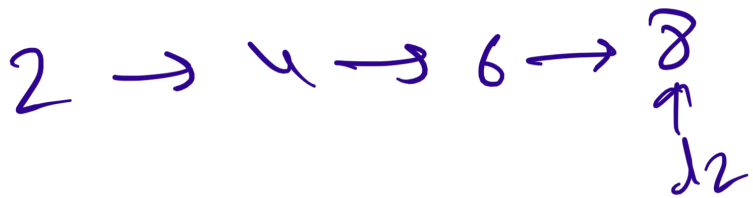
1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

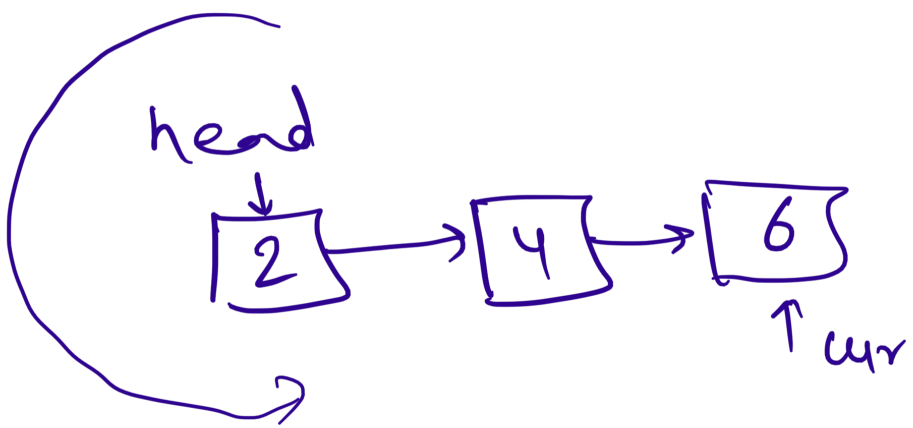
Delete an element whose pointer is given in a Linked List.

Practice Problems

1. Find the Kth Element from last in a Linked List.
2. Remove duplicates from a Sorted Linked List.
3. Sort a Linked List using Bubble sort.
4. Find the intersection of Two sorted Linked List.
    - First linked list: 1->2->3->4->6
    - Second linked list be 2->4->6->8,
    - Output: 2->4->6.
5. Check if a Singly Linked List is a Palindrome.


1 → 2 → 3 → 4 → 6 → X


2 → 4 → 6 → 8
12


head
2
4
6
curr