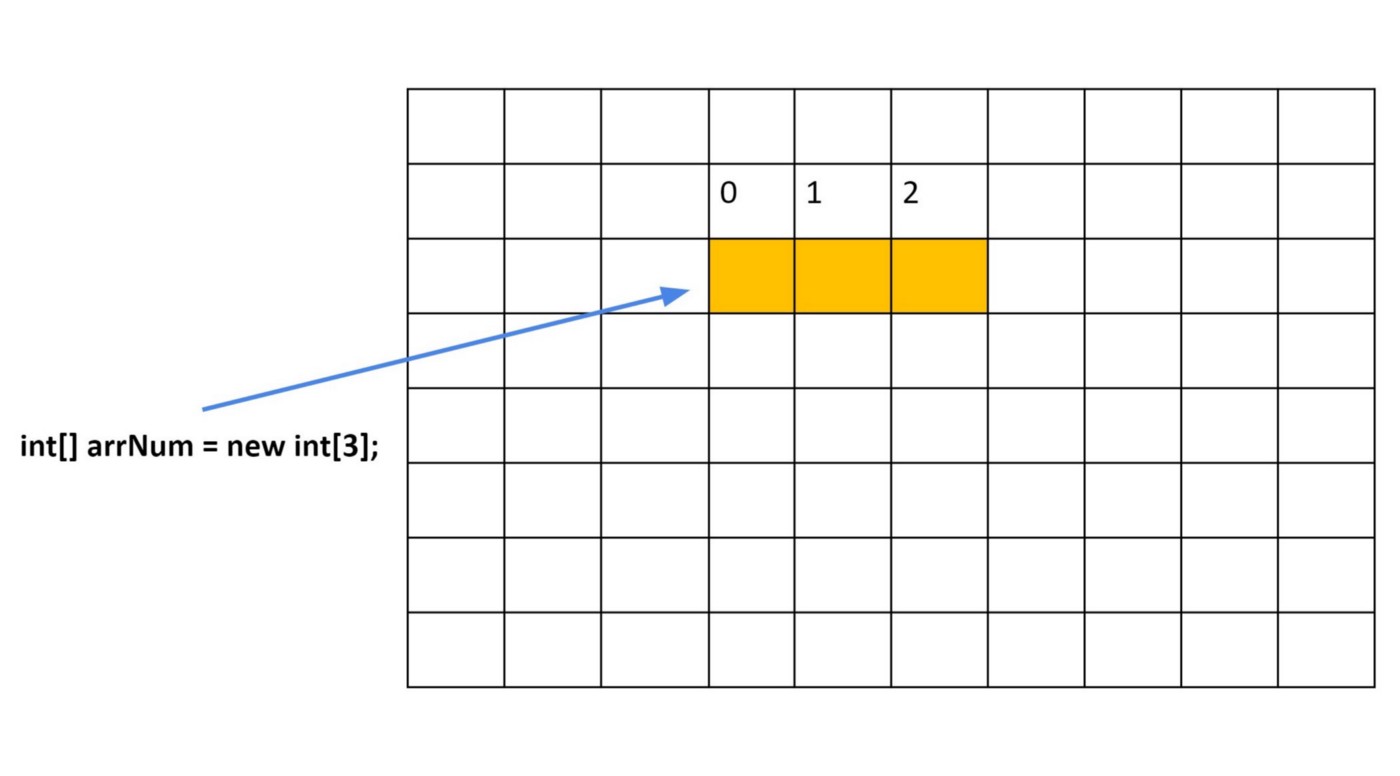
This article is about the main data structures with high-level coding examples in Java. We will discuss  
**- Array in Java  
- Singly-Linked list  
- Binary Tree  
- Hash table**

**Array in Java**

The array is a fixed-size data structure with random access. Fixed-size means that while creating an array we need to specify its size(number of elements array can hold).

We created an array of type int with size 3 in the below example.  
int[] arrNum = new int[3];  
If we will imagine memory as the canvas it will look like this:



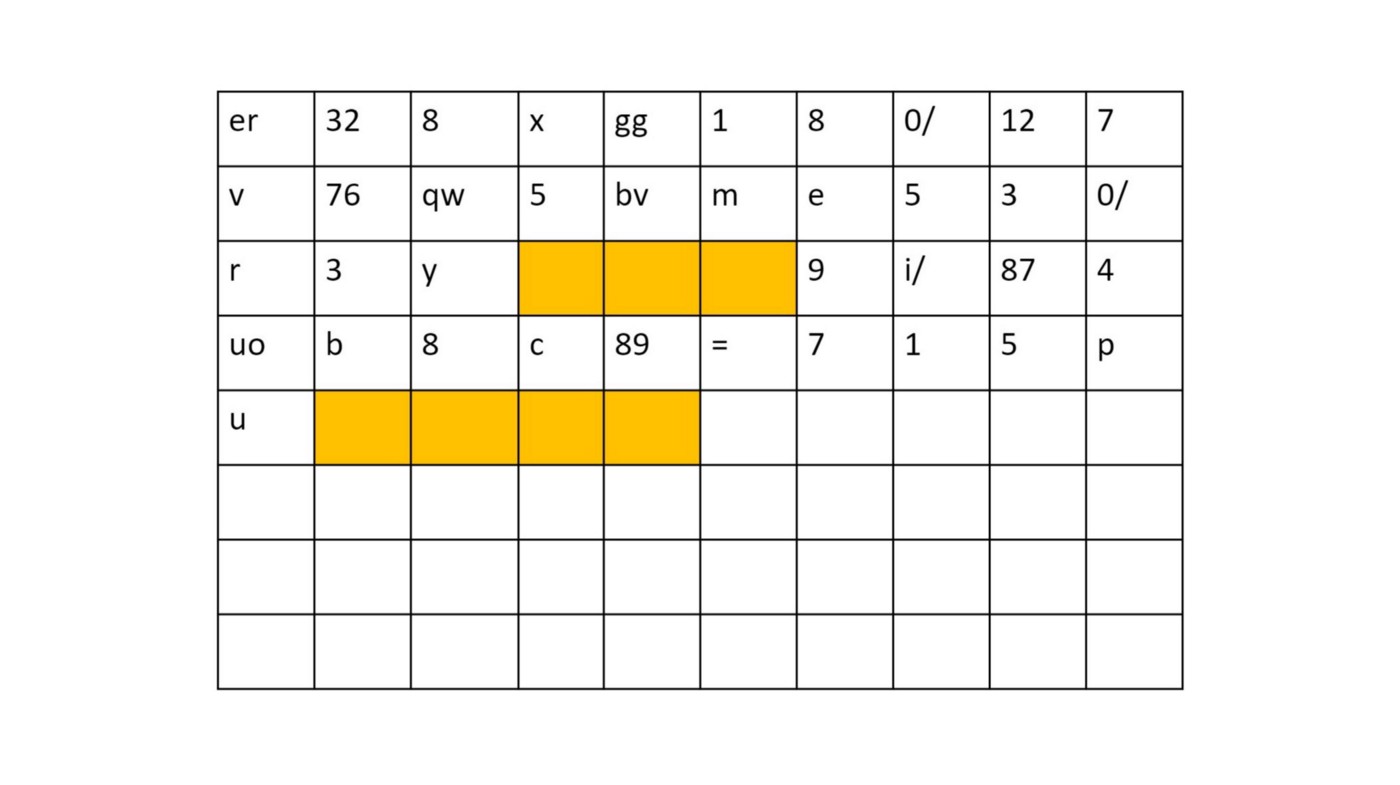
Java will allocate space for 3 elements and the name of the array(reference) will point there. You see, all elements of the array are stored one by one after each other. So that’s why we have random access with arrays. Each element has an index that starts from 0. Another thing to notice here, each element of the array will have a default value in the beginning. For numeric data is 0, for objects is null and for boolean is false.

Let’s discuss the advantages and disadvantages of the array. The first advantage is random access. If we know the correct index we can read or update elements value with one step using the index.

arrNum[0] = 7; // assign to first element value 7

In another hand, the biggest disadvantage is fixed size. For example, I need an array in my program and I’m not quite sure how many elements it should have. I will create an array with the approximate size and let’s say in the middle of my program I need to put more elements than the array size. I will need to resize my array, resizing is expensive because I need to create a new array with a bigger size and then copy all elements from an existing array. Creating a new array with a different size is the only way to resize an array.

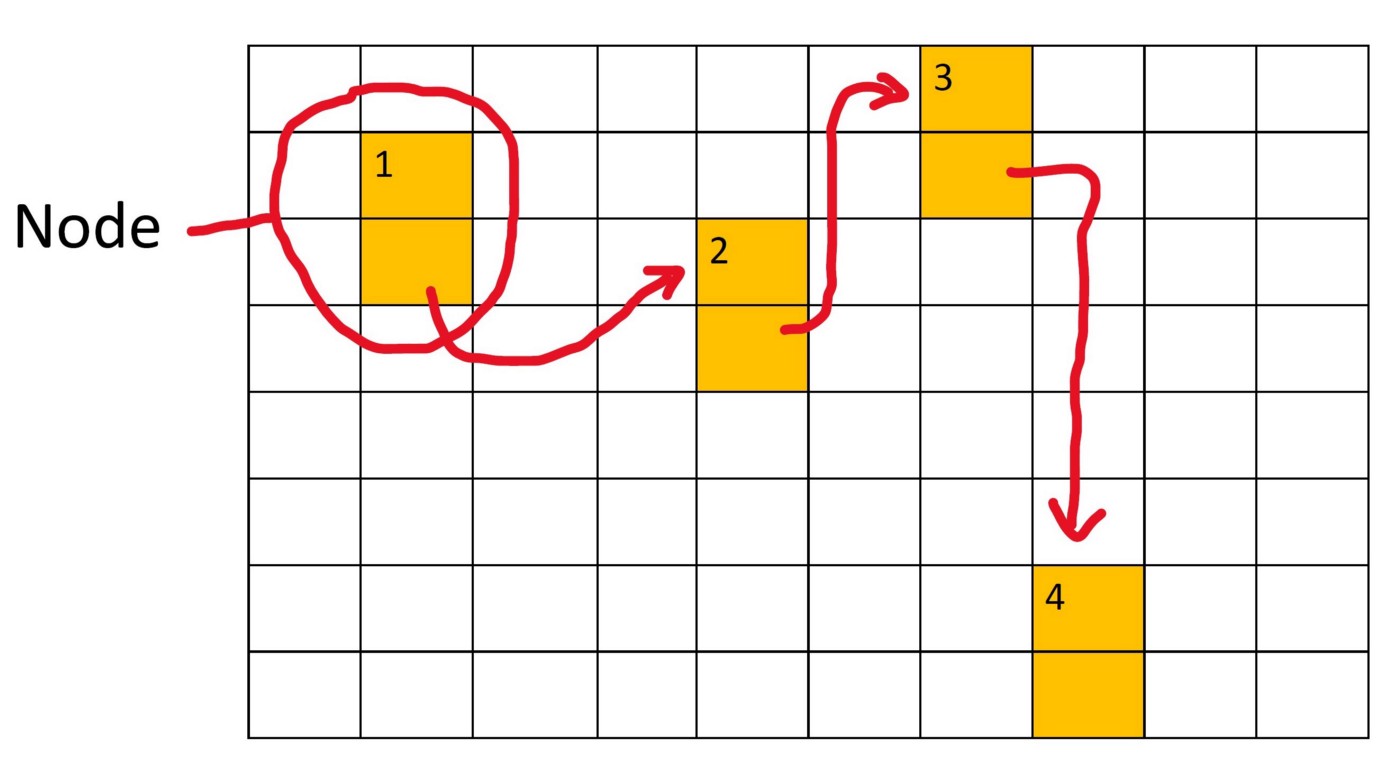
The question is why we don’t just add extra space to the array we created in the first place. This is not that simple. We allocated specific space for our array in the memory however memory is used by other programs as well. No one will guarantee that after our array we will have space to add extra space to the existing one.



The above picture is showing that there is no available space after the array, we created initially. So we will go ahead and create a new array with a bigger size and copy existing elements there. That’s how we resize the array(the shrinking array is similar).

**Linked List**

It would be great if we could have a collection of elements with dynamic size so we don’t need to worry about resizing. The linked list is a dynamic size collection. The way linked list achieved its dynamism is by storing its elements all over the memory(not together as an array) and linking them one to another.



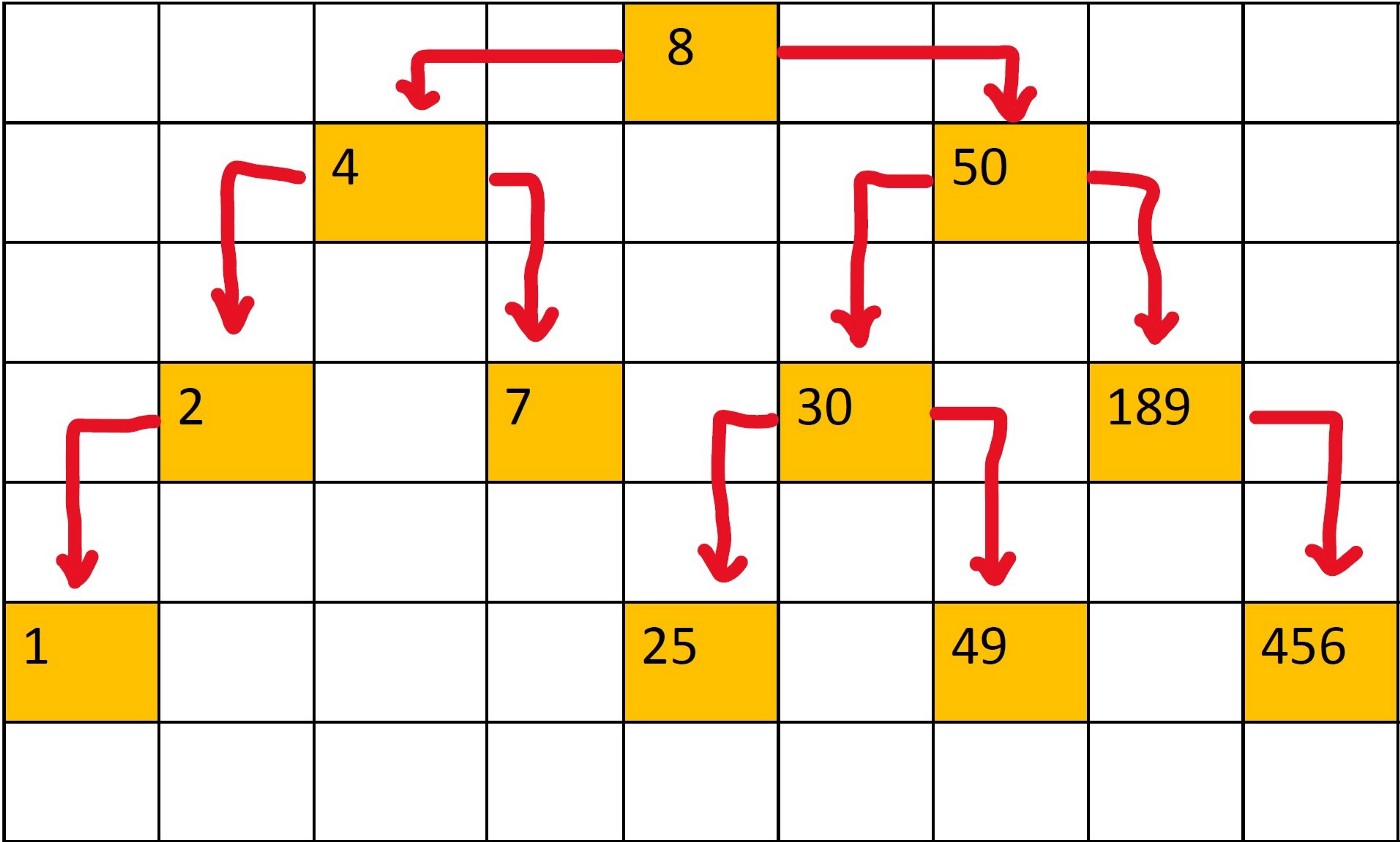
As you see in the above example the elements(nodes) of the linked list are all in different places. The first element of the link list will know where to find the second one and the second one will know where is the third one, and so on. Since we don’t need to worry about putting them together and we don’t need to worry about size because the next element can be located anywhere. Similarly, because the elements are not located together, we are losing random access. There is no random access in the linked list.

We refer to the element as a node. A node usually represented as an object. The node will have two properties. The first one will hold the value of the node and the second one will hold a reference to the next node.

The above snippet is just an example. We would create a class with APIs to manipulate with the list(ex: java.util.LinkedList).

The linked list has a dynamic size. It’s the main advantage of it. The main disadvantage — there is no random access. In order to get some value, we need to travel from the beginning to a specific element. There is no way I can in one step get a specific element if it's not the first one.

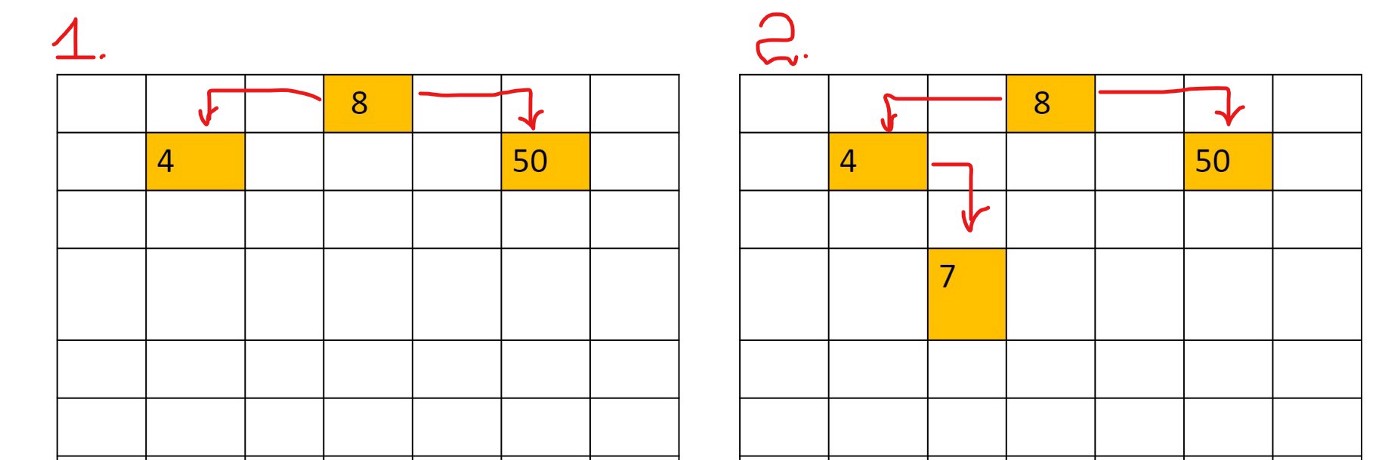
**Binary Tree**



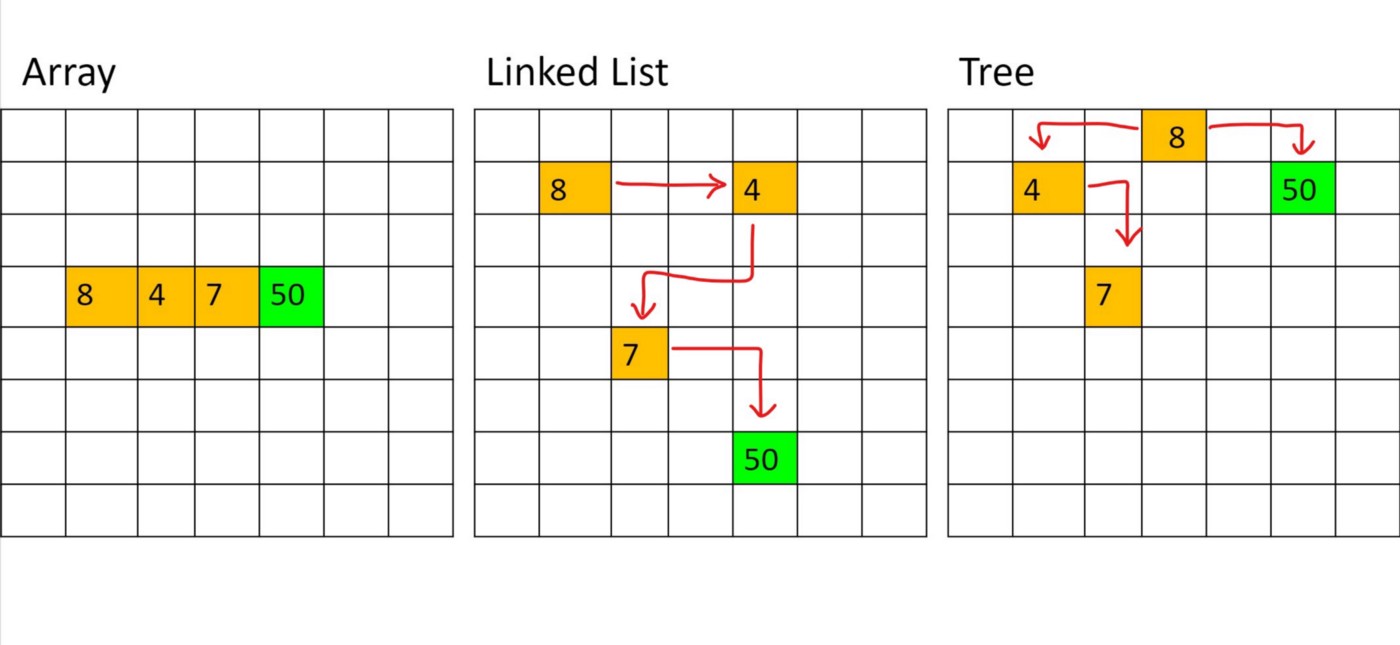
The tree data structure is used in many places. Few examples would be HTML where DOM is structured as a tree, JSON, and XML using a tree structure as well. We want to specifically focus on a binary tree where each node will have at most two child nodes.

The binary tree data structure is also based on nodes and links between them. In the tree node, we will have two links ***left child***and ***right child***so each node can have links to two other nodes. The first node that starts the chain is the ***root***node. See the above example.

Why do we need to complicate our lives and have this tree structure when we have already linked list? Well, the main idea of the binary tree data structure is “***how we store the data”***. It means we need to follow some rules while storing the data. The rule says smaller values should be stored on the left side of the current node and bigger values should be stored on the right side of the current node. Since data itself has a structure in our tree because while inserting we follow this rule while getting/accessing a specific node we can gain significant efficiency.



In the above example, we see how we inserted the number 7 into our tree. In this case, we assume that the value of the node of the tree is an integer. In the table, we had a tree with three nodes. Why we inserted a node with value 7 on the right side of the node with value 4? We do insert smaller values to the left and greater values to the right side:  
1. We compare the root node value with the new node value. 7 < 8 so 7 should go to the left side.  
2. Left side of the root node is already occupied by a node with value 4. We need to go deeper. We compare 4 and 7 now. 7 > 4 so node with value 7 will go to the right side. That’s it. If there would be more nodes we would keep repeating these steps.



In the above picture, we have identical data with three different data structures we have discussed so far. Now, we want to create a function that will check if the number 50 exists in each data structure.

1. For array, it appears that an element with value 50 is the last element and we need 4 steps to get it and make sure it’s there. Basically, it’s a worst-case scenario and it’s taking O(n) time complexity.
2. For the linked list, it’s exactly the same as for the array we need to iterate over each node, and the node with value 50 is last so it will take 4 steps. Similarly worst-case is O(n).
3. For Tree, it will take only 2 steps to get the correct node. Because while inserting we follow the above rule and while reading by using this rule we can access certain values faster. However, the worst-case scenario is still O(n). For example, if we fill our tree with sorted numbers, it will look like a linked list.

The binary tree is really good for searching if non-sorted data is inserted. Another thing to note, the tree is a recursive data structure and using recursion to traverse the tree will work well.

**Hash Table**

If I would be asked to describe a hash table in one sentence. **The data itself will give us clue where to store it and in the future where to find it.**

The above picture describes the hash table really well. Once your online order is ready they will put your meal above your name's first letter. When you will come to pick it up you don’t need to keep looking at each meal you will right away look into the meal which is above your letter. This is a hash table! Your name decided where to go. Since you know your name, you know where to look right away.

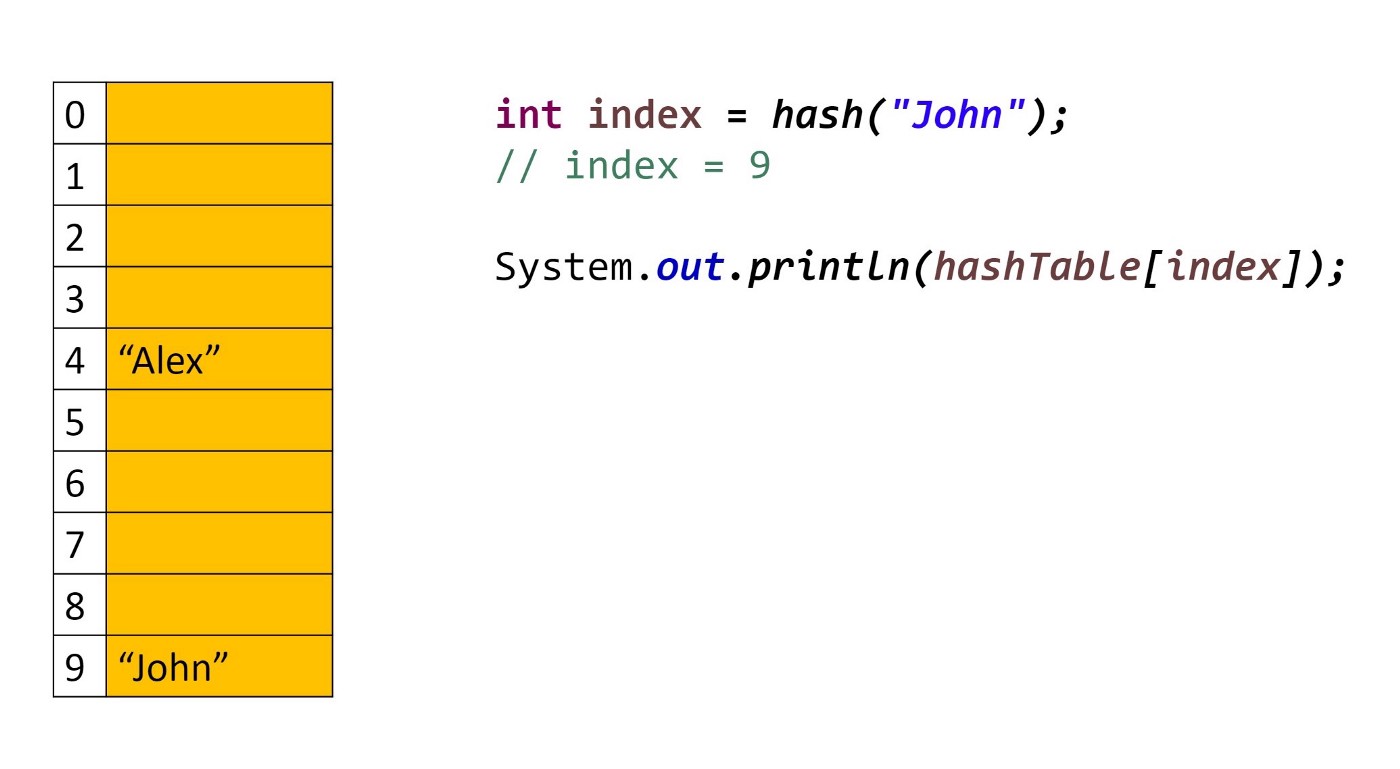
Let’s take a look at our hash table which will hold String as a value. We will need three main parts:

1. Hash function.
2. An array as an underlying data structure for the hash table.
3. Linked list to handle collisions.

**Hash function.** The main idea for our hash function will take a single data as an argument and generate hash code. This hash code will be used as an index in our underlying array data structure. How to define a hash function? Really no limit to the number of possible hash functions. A good hash function should:  
- Use only the data being hashed  
- Use all data being hashed  
- Be consistent. If we run the same data via our hash function a million times it should generate the same hash code for each time.

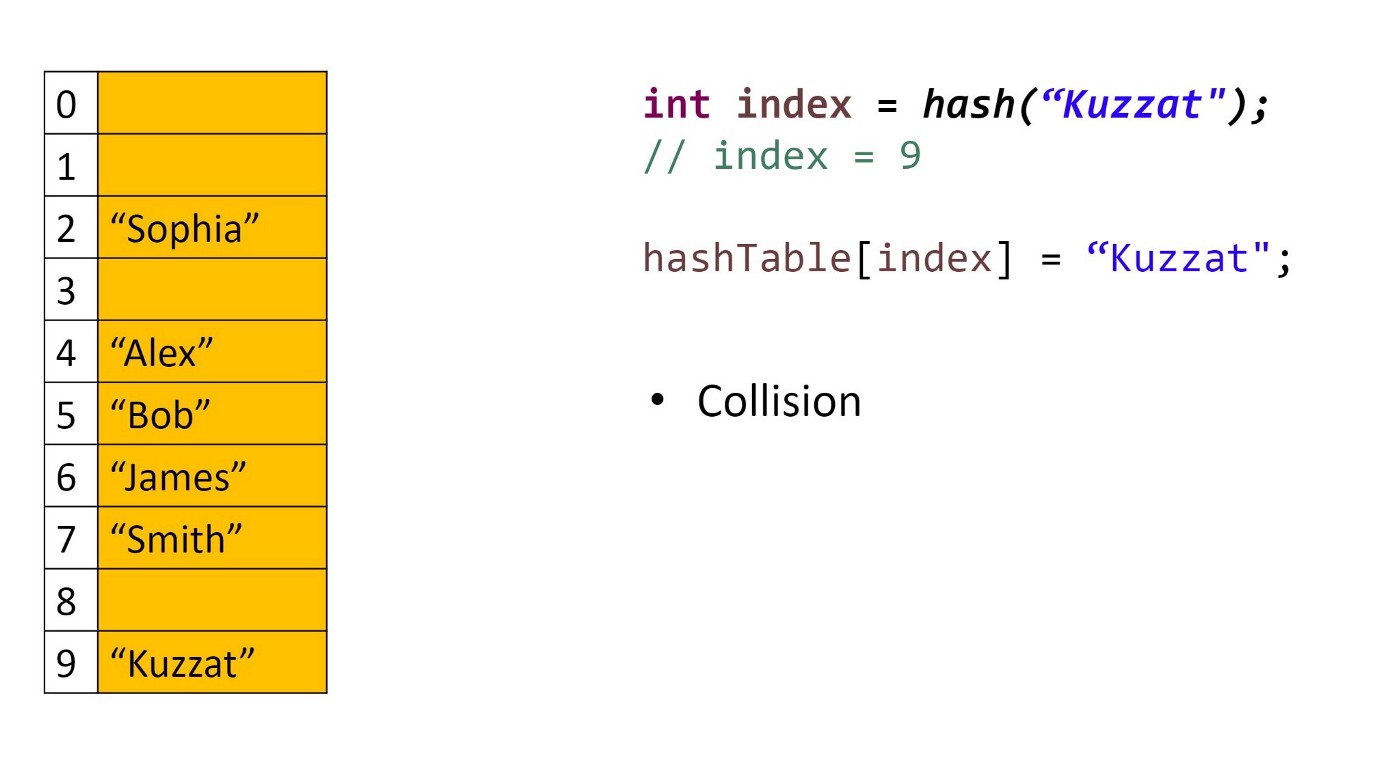
In the above code snippet, we have a hash function that will take a String as an argument and generate hash code. This is a very simple hash code we are just taking each char’s number representation in the string and adding it together. In the end, we are using the remainder operator to get an index that will fit into our array.

**An array as an underlying data structure for the hash table**. Our hash function will generate a hash code. We use this hash as an index to store data in our array.



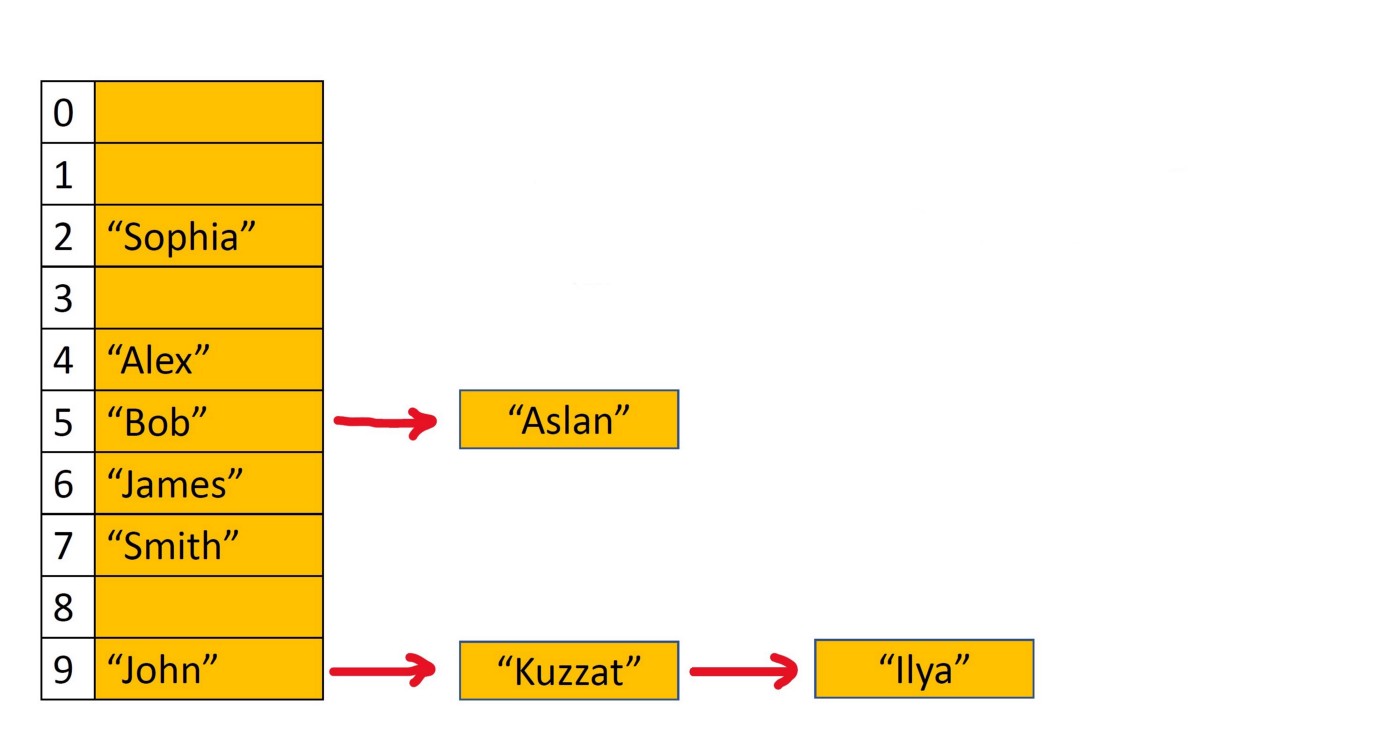
On the left side is an array that we are using as a hash table

**Linked list to handle collisions.**



When two different data will have the same hashcode then we have a collision here.

As you can see in the above example String “Kuzzat” will also have hashcode 9 and will try to directly put it in our array then we simply overriding the old value which was under index 9 — “John”. We could increase HASH\_MAX, it will solve this problem but there is no guarantee that next time two different data will not have the same hashcode even though the value of HASH\_MAX (array size) is huge.



John, Kuzzar, and Ilya generated the same hashcode

Each element of the array will be a singly-linked list to resolve the collision problem. Elements that have the same hashcode will be stored in the element of an array as a linked list.

Hash Tables combine the random-access ability of an array with the dynamism of a linked list. Assuming we define our hash table well, the insertion, deletion, and lookup can start to tend toward constant time. The trade-off is that hash tables are not great at ordering or sorting because data will decide where to go.