Array and String

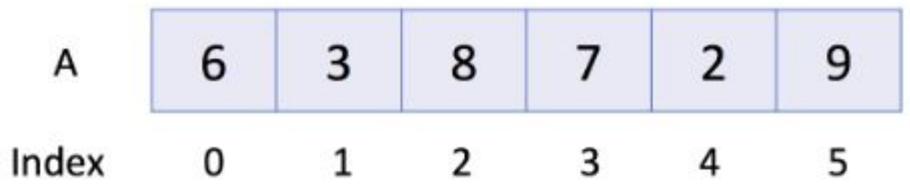
Introduction

1. Understand the differences between array and dynamic array;
2. Be familiar with basic operations in the array and dynamic array;
3. Understand multidimensional arrays and be able to use a two-dimensional array

Introduction to Array

An **array** is a basic **data structure to store a collection of elements sequentially**. But elements can be accessed randomly since each element in the array can be identified by an array index. Array has a **fixed capacity.**

An **array can have one or more dimensions**. Here we start with the one-dimensional array, which is also called the linear array. Here is an example:



Big-O

Time complexity:

1. Access: O(1)
2. Search: O(N)
3. Insertion: O(N)
4. Deletion: O(N)

Space complexity:
O(N)

Initialize

```
 public static void main(String[] args) {
        int[] a0 = new int[5];
       int[] a1 = {1, 2, 3};
}
```

Get Length and Access Element

<u>Get Length</u>

```
System.out.println("The size of a1 is: " + a1.length);
```

<u>Access Element</u>

```
System.out.println("The first element is: " + a1[0]);
```

Iterate all Elements

```
 for (int i = 0; i < a1.length; ++i) {
       System.out.print(" " + a1[i]);
}
for (int item: a1) {
       System.out.print(" " + item);
}
```

Modify Element and Sort

Modify Element

```
a1[0] = 4;
```

Sort

```
Arrays.sort(a1);
```

Dynamic Array

Programming languages offer built-in dynamic array which is still a random access list data structure but with variable size.

Big-O

Time Complexity:

1. Insert at last index - O(1) or If array copy operation is Considered then O(N)
2. Insert at given index - O(N)
3. Search by value - O(N)
4. Get by index - O(1)
5. Remove by value - O(N)
6. Remove by index - O(N)

Space Complexity:

O(N)

Initialize and cast an array to a ArrayList

1. initialize

```
List<Integer> v0 = new ArrayList<>();
List<Integer> v1;                              // v1 == null
```

2. cast an array to a ArrayLIst

```
Integer[] a = {0, 1, 2, 3, 4};
v1 = new ArrayList<>(Arrays.asList(a));
```

Actions

3. make a copy

```
List<Integer> v2 = v1;                    // another reference to v1
List<Integer> v3 = new ArrayList<>(v1);   // make an actual copy of v1
```

4. get length

```
System.out.println("The size of v1 is: " + v1.size());
```

5. access element

```
System.out.println("The first element in v1 is: " + v1.get(0));
```

Iterate the ArrayList

```
System.out.print("[Version 1] The contents of v1 are:");
    for (int i = 0; i < v1.size(); ++i) {
        System.out.print(" " + v1.get(i));
    }
    System.out.println();
    System.out.print("[Version 2] The contents of v1 are:");
    for (int item : v1) {
        System.out.print(" " + item);
    }
    System.out.println();
```

Modify element

6. modify element

```
v2.set(0, 5);       // modify v2 will actually modify v1
System.out.println("The first element in v1 is: " + v1.get(0));
v3.set(0, -1);
System.out.println("The first element in v1 is: " + v1.get(0));
```

7. sort

```
Collections.sort(v1);
```

Add and delete

8. add new element at the end of the ArrayList

```
v1.add(-1);

v1.add(1, 6);
```

9. delete the last element

```
v1.remove(v1.size() - 1);
```

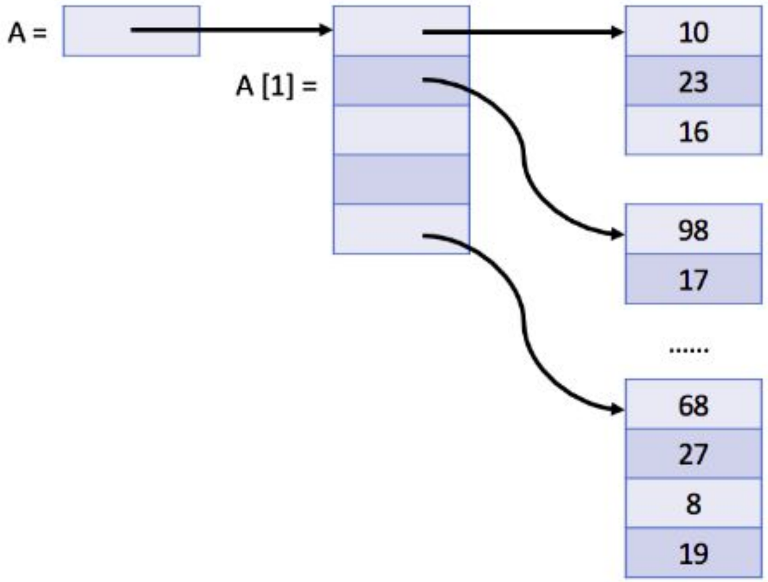
2D Array

Two-dimensional array consists of a sequence of elements. But the elements can be laid out in a rectangular grid rather than a line.

In Java, the two-dimensional array is actually a one-dimensional array which contains M elements, each of which is an array of N integers.

Similar to the one-dimensional dynamic array, we can also define a dynamic two-dimensional array. Actually, it can be just `a nested dynamic array`

Two-dimensional array



Introduction to String

A string is actually an array of **unicode characters.**

Immutable means that you can't change the content of the string once it's initialized.

In Java, since the string is immutable, concatenation works by first allocating enough space for the new string, copy the contents from the old string and append to the new string.

Mutable string - solution 1

If you did want your string to be mutable, you can convert it to a char array.

```
public class Main {
    public static void main(String[] args) {
        String s = "Hello World";
        char[] str = s.toCharArray();
        str[5] = ',';
        System.out.println(str);
    }
}
```

Mutable string - solution 2

If you have to concatenate strings often, it will be better to use some other data structures like StringBuilder. The below code runs in O(n) complexity.

```
public static void main(String[] args) {

    int n = 10000;

    StringBuilder str = new StringBuilder();

    for (int i = 0; i < n; i++) {

        str.append("hello");

    }

    String s = str.toString();

}
```

Compare Function

Can we use "==" to compare two strings?

If the answer is no (like Java), we may not use "==" to compare two strings. When we use "==", it actually compares whether these two objects are the same object.

1. Compared by 'equals'
2. Compared by 'compareTo'
3. equalsIgnoreCase
4. matches

Big O - String

String concatenation complexity in Java is quadratic O(n^2)

```
String aplusb = "a" + "b";
```

StringBuffer or StringBuilder is O(n)

Determine String Length

length() returns the total number of characters in our String.

isEmpty() returns true or false depending on whether our String is empty or not.

***isEmpty()* works faster than checking if the string's length is zero**.

Finding Characters and Substrings

indexOf(int ch) returns the first index position that matches the given character value

indexOf(int ch, int fromIndex) returns the first index that matches the given character value AFTER fromIndex

indexOf(String substring) returns the (first) starting position of substring in the String object it was called on

indexOf(String substring, int fromIndex) same as the previous method, but the search begins at fromIndex instead of 0

Extracting Substrings

substring(int startIndex) returns a String containing all the characters from startIndex (inclusive) to the end of our String. It behaves the same as substring(int startIndex, ourString.length()).

substring(int startIndex, int endIndex) returns a String containing all the characters from startIndex (inclusive) to endIndex (exclusive, i.e. the character at endIndex isn't returned)

Changing String Case

toLowerCase(): changes all upper case characters to lower case (ignores everything else)

toUpperCase(): changes all lower case characters to upper case (ignores everything else)

Removing Whitespace

trim()

""

Replacing Characters and Substrings

replace(char oldChar, char newChar) replaces all occurrences of oldChar with newChar.

replace(CharSequence target, CharSequence replacement) replaces all occurrences of target string with the replacement string (meaning that we can replace entire substrings instead of just characters).

replaceAll(String regex, String replacement) replaces all substrings that match the regex argument with the replacement string.

replaceFirst(String regex, String replacement) replaces only the first substring that matches the regex argument with the replacement string.

Splitting and Joining Strings

split(String regex) splits this string using a given regular expression and returns a character array.

split(String regex, int limit) is similar to the previous method, but only splits a limit number of times.

join(CharSequence delimiter, CharSequence... elements) on the other hand returns a String containing all of the elements we listed, joined by the delimiter.

join(CharSequence delimiter, Iterable<? extends CharSequence> elements) is a very complicated way of saying that we can use join() on things like lists, to combine all the elements into a String using the given delimiter.

Creating Character Arrays

toCharArray() a straightforward method signature.

```
String ourString = "These will all become separate characters";
System.out.println(Arrays.toString(ourString.toCharArray()));
```

Sources

https://stackabuse.com/common-string-operations-in-java/

https://www.baeldung.com/java-string-performance