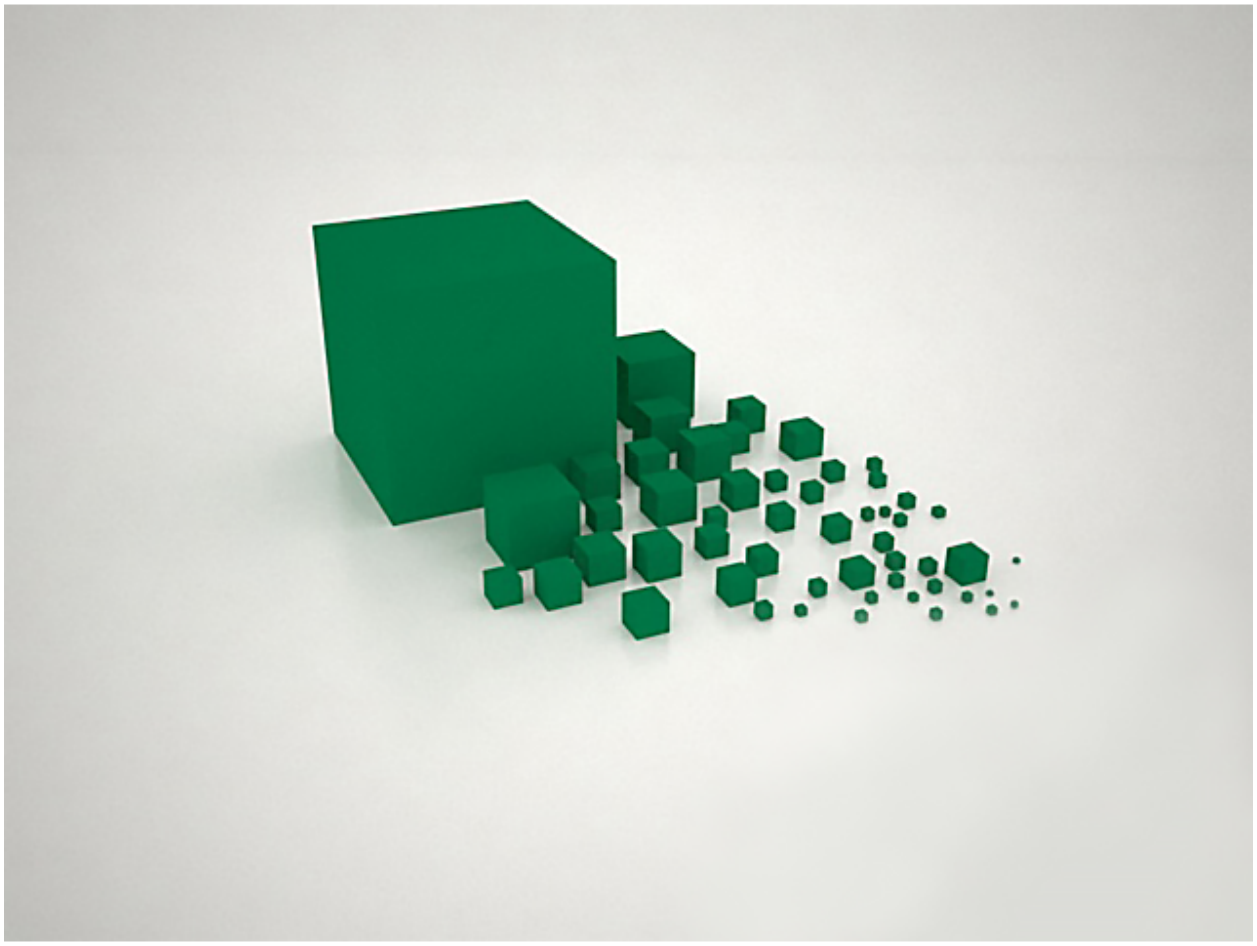


Data Structures & Algorithms with Java:

The Building Blocks for Better Algorithms and Dynamic Programming

By Chad Jordan – December 16th 2007

Please Note: This guide is a continuation from my two previous guides on *Understanding Programming with C++, and An Intro to Game Logic in C++*. Those documents are vital to the foundations of programming and directly correlate to the content covered in this guide. If you are new to algorithms and programming in general, I recommend that you cover my previous two guides first as this guide is designed for experienced programmers.

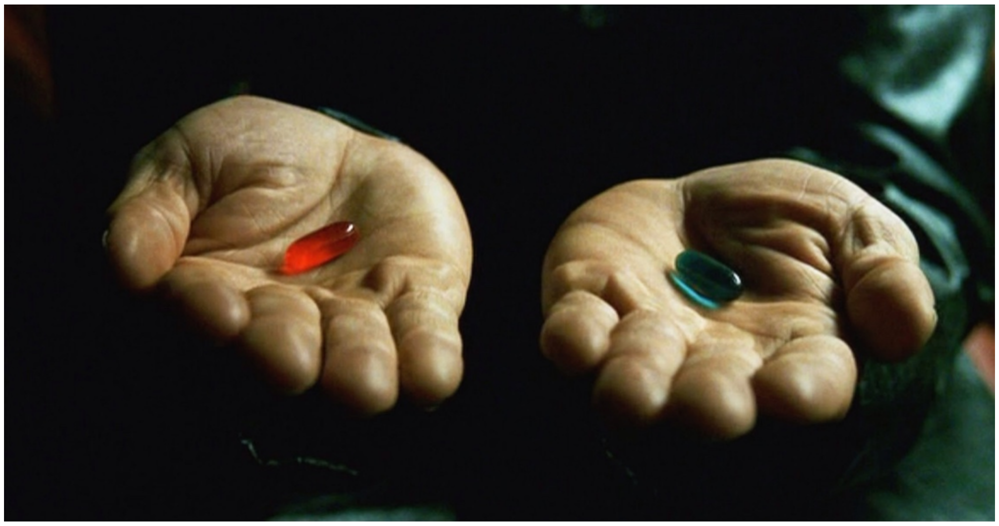
In this guide, you will learn:

1) A fundamental explanation of the Java programming language
2) How to design, analyze, and compare algorithms to solve complex problems
3) Data analysis for space and time in computational complexity
4) Exception handling, Merging and Mergesort structures
5) A deeper dive into Abstract Data Types *(ADT)*, and OOP concepts using Java
6) Explanations on hash tables, searching and sorting with radices, and graph theory

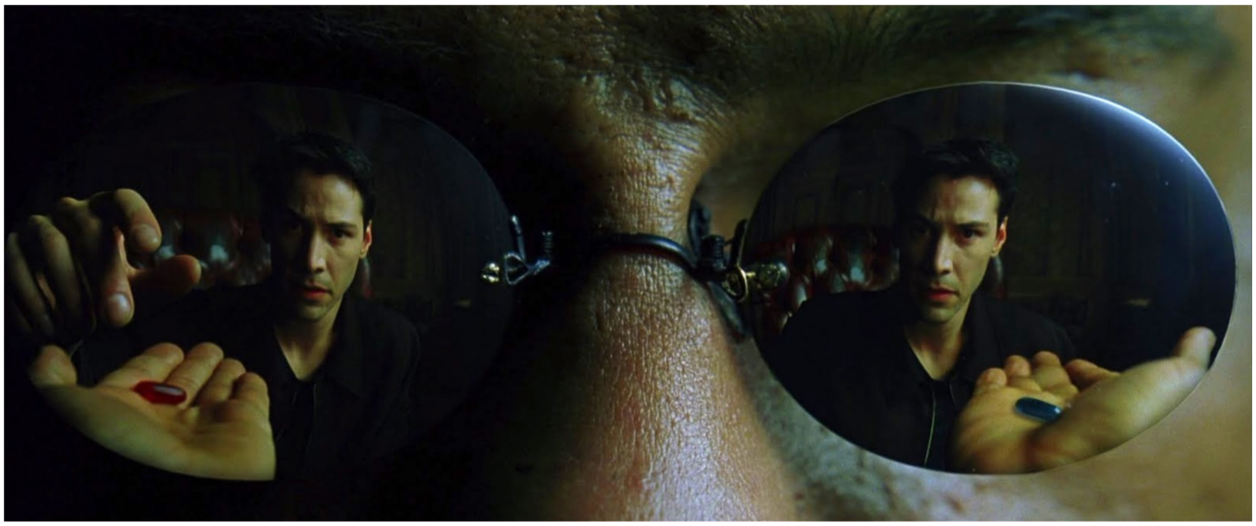
Introduction

"Do I really need a Data Structures & Algorithms course?" With the accelerated rising of online forums, we see programming discussion groups regarding the importance, and relevance of Data Structures & Algorithms as a required subject in the field of Computer Science. I've witnessed numerous online users asking, *"do I really need to take a Data Structures & Algorithms course?"* When I see these questions, my initial thought goes back to the film, *The Matrix* when Morpheus asks Neo,

"do you want to take the red pill or the blue pill?"



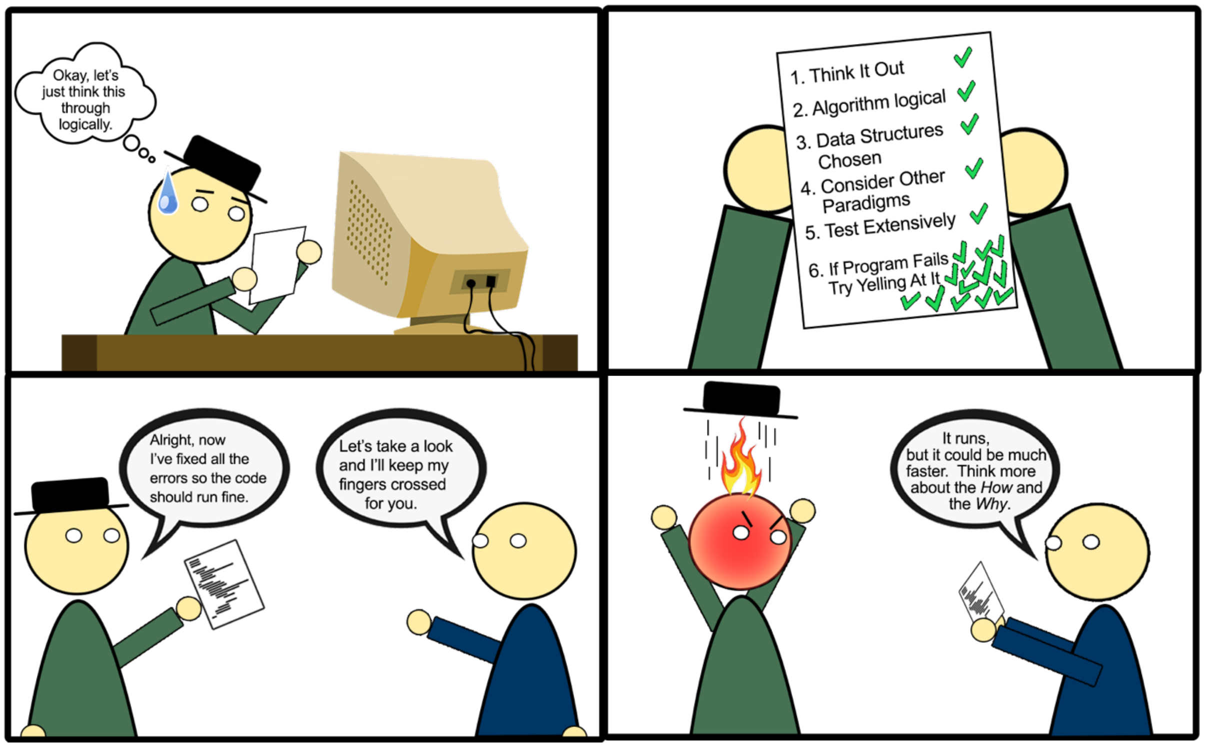
The blue pill lets you stay where you are and you can believe whatever you want to believe, or you can take the red pill and be shown just how deep the rabbit hole goes. Your introductory computer science courses will most likely give you nearly all of the essential tools for learning algorithms and writing good data structures, but a Data Structures & Algorithms course will significantly mature your problem-solving skills, and prepare you for even more in-depth skills in software development and machine learning.


Since you are clearly a more experienced and confident programmer and have decided to read further in this guide, you have chosen to take the red pill.

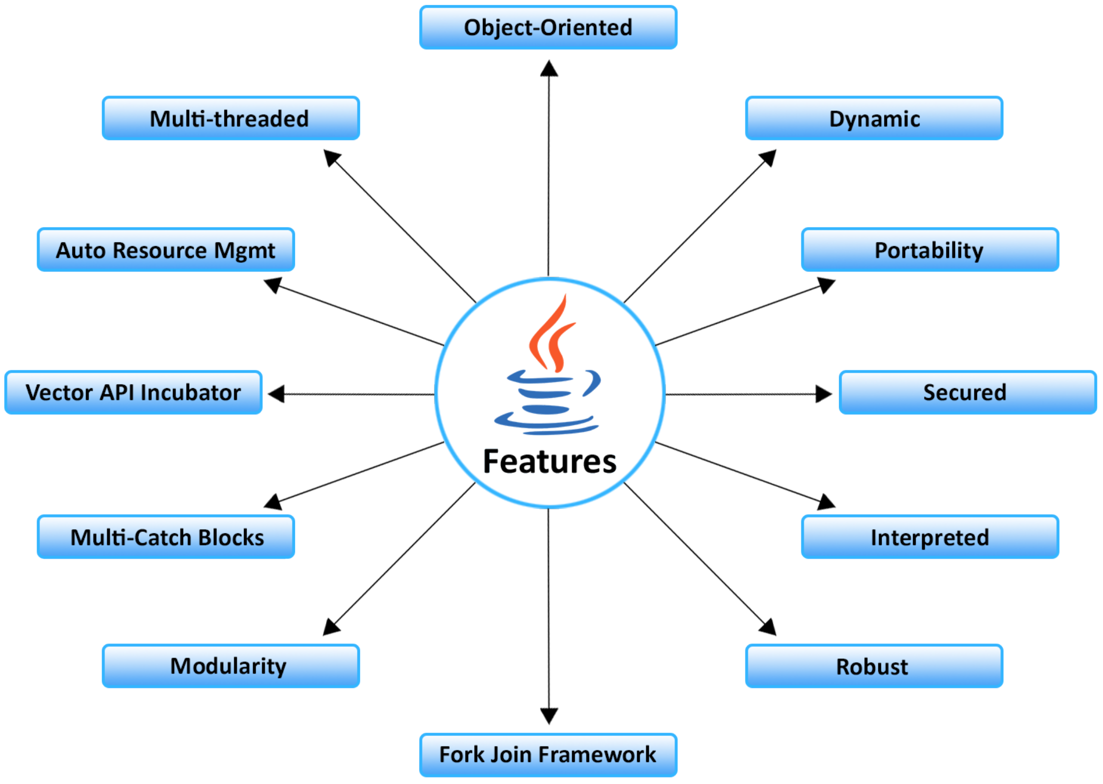
Speaking from the standpoint as a Computer Science major who has taken a Data Structures & Algorithms course, I can confidently elaborate on this frequently growing question that continues to sprout up online. Not all four-year universities offer the option *(or requirement)* to take a Data Structures & Algorithms course but they should. If they do not, you should know that this is one red flag that differentiates a subpar department from a better department that is doing its due diligence to give the proper foundations to their CS students. If your intro courses were anything like mine, the first intro course would have given you the fundamental concepts of building your own algorithms using Pseudo-Code, learning and implementing conditional statements, logical expressions, selection control structures, loops, functions, arrays, data abstraction, recursion, etc. After successfully completing the first introductory course, you would take an *'Intro to Computer Science II'* course which, *if taught properly*, would take an aggressive leap from procedural programming with C++ into object-oriented programming with C++. This second introductory course is considerably more in depth with next-level programming where the emphasis is on **Abstract Data Types** *(ADT)*, with object-oriented concepts and implementation using lists, stacks, queues, pushes, pops, binary trees, simulation, recursion, artificial intelligence, and an introduction to software engineering. These topics are absolutely crucial to learn prior to taking a data structures and algorithms course, so the next question is, why does there need to be a course in Data Structures & Algorithms when you just finished learning all of the essentials? Programmers and other inquiring minds would likely be shocked to see how many people ask this question within the field of computer science. This is most likely due to what first appears like repetitious learning, but this is far from the case. Taking a Data Structures & Algorithms course requires a considerable increase in understanding how to write better code that runs faster and more efficiently. There's also a much bigger jump into discrete mathematics, probability, Boolean algebra, and graph theory.

The next question is, 'What is Java, and why do we use it?' Java is a high-level, object-oriented programming language known for its platform independence, robustness, and portability. It was developed by James Gosling and his team at Sun Microsystems *(later acquired by Oracle Corporation)* and first released in 1995. Java has since become one of the most popular programming languages for a wide range of applications, from web development to mobile apps and enterprise-level systems. Java is often referred to as a "write once, run anywhere" *(WORA)* language. This is because Java code is compiled into an intermediate bytecode format that can run on any platform with a Java Virtual Machine *(JVM)*. The JVM acts as a runtime environment that translates bytecode into machine code specific to the underlying operating system. Java is also an object-oriented programming *(OOP)* language making it easier to structure code and build complex applications. One of the bigger methodologies that separated my Data Structures & Algorithms course from my introductory comp-sci courses was the **analysis of time and space**. IE, time *(quicker response)* and space *(minimal memory usage)*. In previous courses as long as any assignments we did met all required tasks and compiled with no errors or warnings, we were good, and we received full credit for each assignment. In Data Structures & Algorithms, you must still meet those requirements, and any code that resulted in correct output but did not run within time and memory constraints would not receive full credit. These performance and conformance restraints really developed critical thinking skills.

Metacognition in Data Structures & Algorithms:


Okay, let's just think this through logically.
1. Think It Out
2. Algorithm logical
3. Data Structures Chosen
4. Consider Other Paradigms
5. Test Extensively
6. If Program Fails Try Yelling At It
Alright, now I've fixed all the errors so the code should run fine.
Let's take a look and I'll keep my fingers crossed for you.
It runs, but it could be much faster. Think more about the How and the Why.

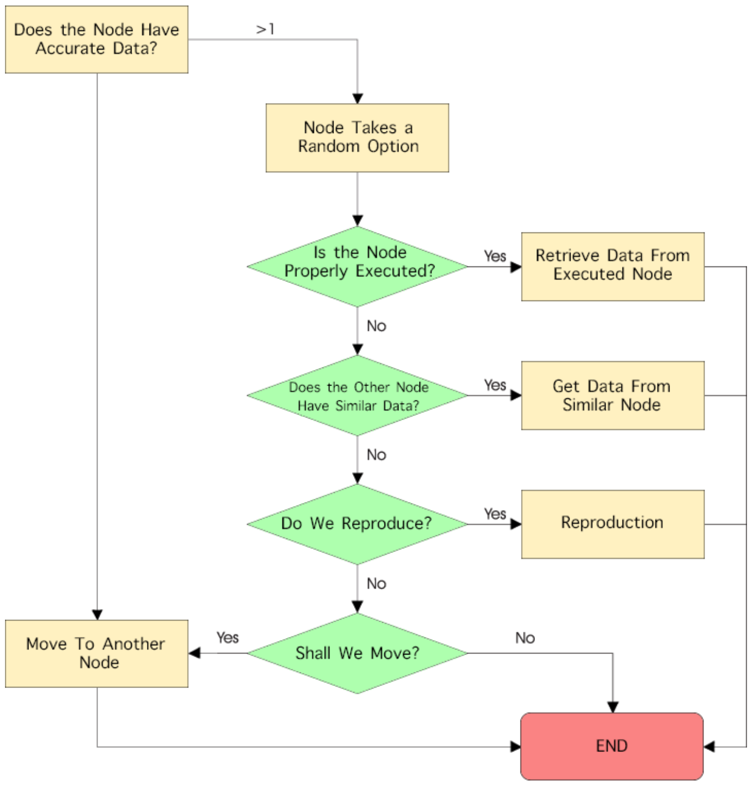
If you read my first technical guide on ***Understanding Programming in C++*** then you would have seen these two characters that I created known as *The Programmer and The Compiler*. Data Structures & Algorithms with Java is more than just writing code that works, it's about learning how to write more efficient code that runs better. Everything in Java is object-oriented, which allows for encapsulation, inheritance, and polymorphism. Java also uses automatic memory management through a *garbage collector*. This feature frees developers from managing memory explicitly, helping to prevent common programming errors like memory leaks and making the language more reliable. Java emphasizes strong typing and includes various checks at compile-time and runtime to catch errors early in the development process. Additionally, exception handling enables better error management and resilience giving it a means of great robustness and reliability. Another feature is Java's multi-threading support. This allows developers to create concurrent applications to make use of modern multi-core processors and improve performance. Here are several key features that Java provides as a programming language:



These are only some of the features that Java provides, and there's a great deal more related to Java's standard library providing better built-in functions and classes for more efficient tasks related to file I/O, networking, and data manipulation. Java also has great security that incorporates a security manager and sandboxing to protect against unauthorized access with malicious code execution. These features make Java a great option for both beginners and experienced developers. Java remains one of the most sought-after programming languages in the market, and especially on the enterprise software development side of the tech industry. As mentioned at the beginning of this document, I will be delving even deeper than my previous two documents that only scratched the surface of C++ programming. If you are already on a much more intermediate level of using programming languages and matured problem-solving skills, then it's time to jump deeper with Data Structures & Algorithms with Java.

Formulating Better Algorithms

As programmers, our greatest focus is to learn how to formulate better algorithms to build greater software. This fundamental concept in computer science should remain the cornerstone of every great implementation of code into your projects. Better scalability equals a better methodologies, better performance, better efficiency, and greater possibilities that were perhaps not previously conceptualized in the systematic thought process. From my previous documents with C++ programming, we already know that you begin by understanding/identifying that a problem exists, followed by breaking down the problem similar to my diagram on the left. Our thought process must align to the type of algorithms that we create or follow. However, coming up with a successful algorithm is only part of the process. Even with a solid algorithm in place, we must consider what we do to take the appropriate steps to choose the right data structures. We know that selecting appropriate data structures can have a significant impact on the efficiency of your algorithm. It is paramount to understand the strengths and weaknesses of different data structures *(arrays, lists, trees, hash tables, etc.)* and use the ones that best match the requirements of your problem. Ergo, it is important to understand that it all depends on the type of problems we are trying to solve. There is no *One-Size-Fits-All* answer to every algorithmic approach. However, we can break down some commonly used algorithms with data structures that are frequently applied to software development use cases:


Does the Node Have Accurate Data?
>1
Node Takes a Random Option
Is the Node Properly Executed?
Yes
Retrieve Data From Executed Node
No
Does the Other Node Have Similar Data?
Yes
Get Data From Similar Node
No
Do We Reproduce?
Yes
Reproduction
No
Move To Another Node
Yes
Shall We Move?
No
END

1. **Sorting Algorithms:**
   <u>*Quick Sort*</u>*:* A comparison-based sorting algorithm that uses a divide-and-conquer approach. Efficient for sorting elements in an array or list.
   <u>*Merge Sort*</u>*:* Similar to a quick sort, merge sort is known for its stability and predictable performance characteristics.
   <u>*Bubble Sort*</u>*:* Primarily used for educational purposes and for demonstrating basic sorting concepts due to its straightforward implementation.

2. **Searching Algorithms:**
   <u>*Binary Search*</u>*:* Widely used for swiftly locating a specific element in a sorted array or list.
   <u>*Linear Search*</u>: One of the simplest searching algorithms, linear searches are for small unordered lists in an array.

3. **Data Structures:**
   Arrays: Fundamental for storing collections of data.
   Linked Lists: Useful for dynamic data structures.
   Disjoint Sets (Union-find): keeps track of elements partitioned in non-overlapping subsets.
   Binary Trees & AVL Trees: Efficient for hierarchical data.
   Hash Tables: Excellent for fast key-value lookups.
   Graphs: For modeling complex relationships.

4. **Dynamic Programming:**
   Used to solve problems by breaking them into subproblems. It's particularly useful for optimization problems.

5. **Graph Algorithms:**
   Dijkstra's Algorithm: Finds the shortest path in a weighted graph.
   Breadth-First Search (BFS) and Depth-First Search (DFS): Fundamental for graph traversal and for exploring trees.

6. **String Matching Algorithms:**
   Knuth-Morris-Pratt (KMP) and Rabin-Karp: Linear and hash algorithms used to efficiently find occurrences of a substring within a larger string.

7. **Numerical Algorithms:**
   Newton-Raphson: Also referred to as The Newton Method, this algorithm is an iterative numerical technique for finding approximate roots of real-valued functions.
   FFT (Fast Fourier Transform): For efficient computing of the Discrete Fourier Transform (DFT) and its inverse, signal processing, and data analysis.

8. **Machine Learning Algorithms:**
   Linear Regression, Decision Trees, Neural Networks and more for various machine learning tasks.

9. **Compression Algorithms:**
   LZW, Huffman Coding, and Run-Length Encoding: Three different data compression algorithms used to reduce the size of data for efficient storage or transmission.

10. **Cryptography Algorithms:**
    AES, RSA, and SHA: are cryptographic algorithms used for data handling of information security, including encryption, decryption, digital signatures and data integrity verification.

11. **Pattern Matching:**
    Regular Expressions: Commonly abbreviated as 'Regex' are powerful and flexible patterns used for searching, matching and manipulating strings in text.

12. **Database Algorithms:**
    <u>*Binary Tree, Hash Indexing, and Query Optimization*</u>: For efficient database operations and management with hierarchical structures, data retrieval and indexing cost-based analysis.

We see that these examples provide insight into specific areas that programmers can use to resolve certain tasks in software development. These are also only several examples compared to the multitude of algorithms that are available for numerous other areas in computer science. Formulating better algorithms is a complex and iterative process requiring an array of critical-thinking skills, systematic reason, experience and creativity. Remember, you can improve your algorithmic design thought process by following a set of standard rules:

1. **Understand the Problem:**
   Before you can design an algorithm, you need a deeper understanding of the problem you're trying to solve. Break the problem down into essential components and consider any potential constraints or requirements by dividing the problem into smaller subproblems to outline your algorithm's logic.

2. **Analyze Time and Space Complexity:**
   Strive for algorithms that are both time-efficient *(they execute quickly)* and space efficient *(they use minimal memory)*.

3. **Use Pseudocode:**
   Before diving into the code, write pseudocode to outline the logic of your algorithm. A step by step procedure helps you to understand how the program should behave.

4. **Choose the Right Data Structures:**
   Selecting the appropriate data structures is crucial to algorithm design. The choice of data structures will greatly impact the efficiency in our algorithm.

5. **Consider Different Paradigms:**
   Exploring different algorithmic paradigms such as greedy algorithms, dynamic programming, backtracking, and divide-and-conquer.

6. **Optimize and Refine:**
   Once you have a functional algorithm, continuously find ways to optimize it. Making small changes to the algorithm or data structures can lead to significant performance improvements.

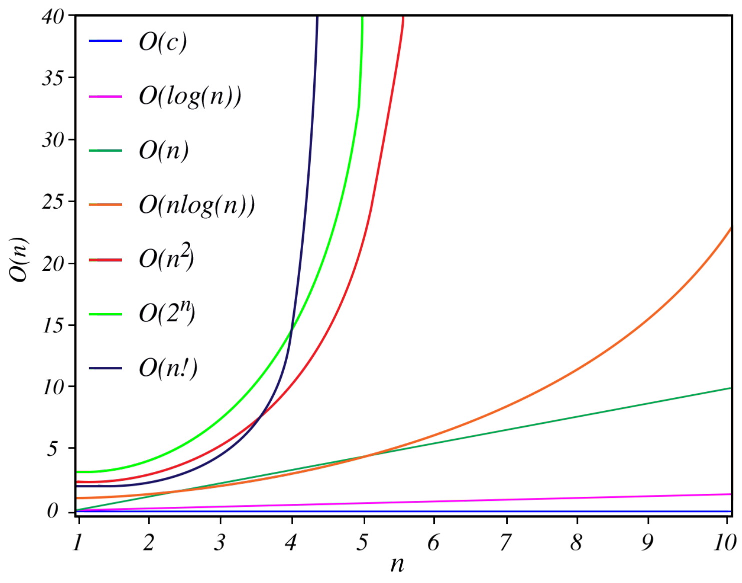
7. **Test Extensively:**
   Thoroughly test your algorithm with various inputs, including edge cases and worst-case scenarios, to ensure it behaves correctly and efficiently.

8. **Learn from Mistakes:**
   Expect to make mistakes and encounter challenges along the way. Learning from these experiences is an essential part of becoming a better algorithm designer.

While these processes are already an essential part of designing, analyzing, and comparing algorithms, we can jump deeper into understanding how to analyze more efficient algorithms with computational complexity.

Computational Complexity

Logically speaking, the same problem can frequently be solved with algorithms that differ in efficiency. The differences between the algorithms may be immaterial for processing a small number of data items, but these differences grow proportionally with the amount of data. To compare the efficiency of algorithms, a measure of the degree of difficulty of an algorithm called *Computational Complexity* was developed by Juris Hartmanis and Richard E. Stearns. Computational complexity indicates how much effort is needed to apply an algorithm or how costly it is. This cost can be measured in various ways and the particular context determines its meaning. This guide emphasizes the two efficiency criteria: *time and space*. The factor of time is far more important than that of space, so efficiency considerations usually focus on the amount of time elapsed when processing data. However, the most inefficient algorithm run on a Cray computer can execute much faster than the most efficient algorithm run on a PC, so runtime is always system-dependent. For example, to compare a hundred algorithms, all of them would have to be run on the same machine. Furthermore, the results of runtime tests depend on the language in which a given algorithm is written even if the tests are performed on the same machine. If programs are compiled, they execute much faster than when they are interpreted. A program written in C or Pascal may be 20 times faster than the same program encoded in BASIC or LISP.



To evaluate an algorithm's efficiency, real-time units such as microseconds and nanoseconds should not be used. Instead, logical units that express a relationship between the size n of a file or an array and the amount of time t required to process the data that should be used. If there is a linear relationship between the size n and the time t, that is, $t_1 = cn_1$, then an increase of data by a factor of 5 results in the increase of the execution time by the same factor, if $n_2 = 5\,n_1$, then $t_2 = 5\,t_1$. Similarly, if $t_1 = log_2 n$, then doubling n increases t by only one unit of time. Therefore, if $t_2 = log_2\,(2n)$, then $t_2 = t_1 + 1$. A function expressing the relationship between n and t is usually much more complex, and calculating such a function is important only in regard to large bodies of data; any terms which do not substantially change the function's magnitude should be eliminated from the function. The resulting function only gives an approximate measure of efficiency of the original function. However, this approximation is sufficiency close to the original, especially for a function that processes large quantities of data. The measure of efficiency is called *asymptotic complexity* and is used when disregarding certain terms of a function to express the efficiency of an algorithm or when calculating a function is difficult or impossible and only approximations can be found. To illustrate the first case, consider the following formula:

$$f(n) = n^2 + 100n + log_{10}n + 1000$$

For small values of n, the last term, 1000, is the largest. When n equals 10, the second (100n) and last (1000) terms are on equal footing with the other terms making the same contribution to the function value. When n reaches the value of 100, the first and second terms make the same contribution to the result. But when n becomes larger than 100, the contribution of the second term becomes less significant. Ergo, for larger values of n, due to the quadratic growth of the first term (n^2), the value of the function f depends mainly on the value of this first time. The following data table demonstrates this.

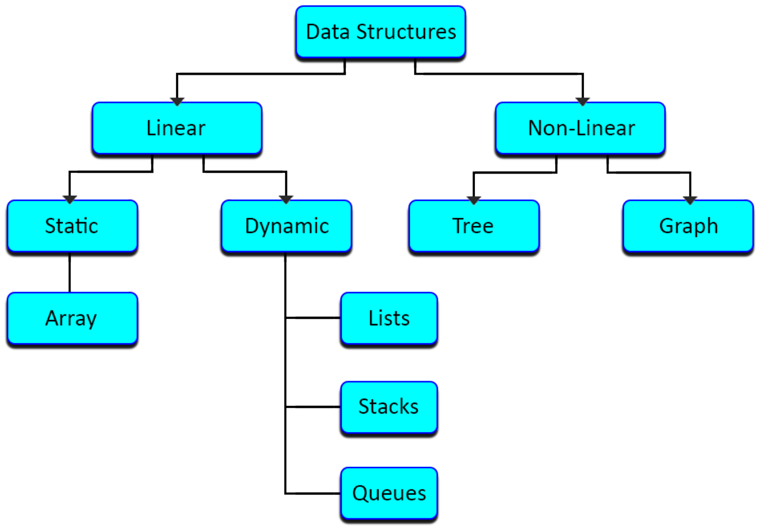
The growth rate of all terms of function $f(n)= n^2+100n + log_{10} n + 1000$

n	$f(n)$	n^2		$100n$		$log_{10} n$		1000	
	Value	**Value**	**%**	**Value**	**%**	**Value**	**%**	**Value**	**%**
1	1,101	1	0.1	100	9.1	0	0.0	1,000	90.82
10	2,101	100	4.76	1,000	47.6	1	0.05	1,000	47.62
100	21,002	10,000	47.6	10,000	47.6	2	0.991	1,000	4.76
1,000	1,101,003	1,000,000	90.8	100,000	9.1	3	0.0003	1,000	0.09
10,000	101,001,004	100,000,000	99.0	1,000,000	0.99	4	0.0	1,000	0.001
100,000	10,010,001,005	10,000,000,000	99.9	10,000,000	0.099	5	0.0	1,000	0.00

The most commonly used notation for specifying *asymptotic complexity*, that is, for estimating the rate of function growth, is the big-O notation introduced in 1894 by Paul Bachmann. Given two positive-valued functions f and g, consider the following definition:
$f(n)$ is $O(g(n))$ if there exist positive numbers c and N such that $f(n) \leq cg(n)$ for all $n \geq N$. This definition reads: f is big-O of g if there is a positive number c such that f is not larger than cg for sufficiently large ns, for all ns larger than some number N. The relationship between f and g can be expressed by stating either that $g(n)$ is an upper bound on the value of $f(n)$ or that f grows at most as fast as g. The next question is, what is the practical significance of the data listed above? It's all related to the same function $g(n) = n^2$ and to the same $f(n)$. For a fixed g, an infinite number of pairs of cs and Ns can be identified for helping to solve for Big-O notation and the result is checking efficiency of time and space for better algorithms.

In my previous two guides I briefly discussed and provided examples of **Abstract Data Types** *(ADT)* such as lists and stacks, but Java programming courses such as Data Structures & Algorithms tend to utilize them even more as well as introducing hash tables, concurrency, and graph algorithms to the mix of algorithmic complexity.

A Deeper Dive into ADTs



As an experienced programmer, you know that data structures come in multiple forms within their own hierarchical structure. These organization of elements fall under **linear** and **non-linear** data structures. In linear data structures, data elements are organized in a sequential manner, where each element has a predecessor and a successor, except for the first and last elements. Examples of linear data structures include arrays, lists, stacks, and queues. In non-linear data structures, data elements are not organized sequentially. Instead, they can have multiple predecessors and successors, forming complex relationships. Examples of non-linear data structures include trees and graphs. Per my above diagram, we see that stacks are linear data structures and we use stacks to store and retrieve data. In most cases when this is done, the stack is very useful when doing so in reverse order. One application of the stack is in matching delimiters in a program. This is an important example because matching delimiters is part of any compiler. No program is considered correct if the delimiters are mismatched. In Java programs, we have the following delimiters: Parenthesis **'('** and **')'**, square brackets **'['** and **']'**, curly brackets **'{'** and **'}'** and comment delimiters **'/*'** and **'*\'**. Here are some examples of Java statements that used delimiters properly:

```
a = b + (c – d) ∗ (e – f);
g[10] = h[i[9]] + (j + k) ∗ l;
while (m < (n[8] + o)) { p = 7; /∗ initialize p ∗/ r = 6; }
```

and these examples are statements in which mismatching occurs:

```
a = b + (c – d) ∗ (e – f));
g[10] = h[i[9]] + j + k) ∗ l;
while (m < (n[8] + o) { p = 7; /∗ initialize p ∗/ r = 6; }
```

Do you see what's happening here? The best way to explain this is a particular delimiter can be separated from its match by other delimiters; that is, delimiters can be nested. Therefore, a particular delimiter is matced up **only after all the delimiters following it and preceding its match have been matched**. For example, in the condition of the loop

```
while (m < (n[8] + o))
```

the first opening parenthesis must be matched with the last closing parenthesis, but this is done only after the second opening parenthesis is matched with the next to last closing parenthesis; this, in turn, is done after the opening square bracket is matched with the closing bracket. The delimiter matching algorithm reads a character from a Java program and stores it on a stack if it is an opening delimiter. If a closing delimiter is found, the delimiter is compared to a delimiter popped off the stack. If they match, processing continues. If not, processing discontinues by signaling an error. The processing of the Java program ends successfully after the end of the program is reached and the stack is empty. Here is the algorithm for further reference and understanding:

```
delimiterMaching(file)
     read character ch from file;
     while not end of file
          if ch is '(','[', or '{'
               push(ch);
          else if ch is '/'
               read the next character;
               if this character is '*'
                    push(ch);
          else ch = the character read in;
               continue; // go to the beginning of the loop;
          else if ch is ')',']', or '}'
               if ch and popped off delimiter do not match
               failure;
          else if ch is '*'
               read the next character;
               if this character is '/' and popped off delimiter is not '/'
                    failure;
               else ch = the character read in;
                    push back the popped off delimiter;
                    continue;
       // else ignore other characters;
          read next character ch from file;
       if stack is empty
          success;
       else failure;
```

Per the above algorithm, you would get the following result when applying this statement:
s = t[5] + u / (v * (w + y));

Stack	Nonblank Character Read	Input Left
empty		s = t[5] + u / (v * (w + y));
empty	s	= t[5] + u / (v * (w + y));
empty	=	t[5] + u / (v * (w + y));
empty	t	[5] + u / (v * (w + y));
[	[	5] + u / (v * (w + y));
[	5	] + u / (v * (w + y));
empty	]	+ u / (v * (w + y));
empty	+	u / (v * (w + y));
empty	u	/ (v * (w + y));
empty	/	(v * (w + y));
(	(	v * (w + y));
(	v	* (w + y));
(	*	(w + y));
(		
(	(	w + y));
(		
(	w	+ y));
(		
(	+	y));
(		
(	y	));
(	)	);
empty	)	;
empty	;	

The way this works is the first column shows the contents of the stack at the end of the loop before the next character is input from the program file. The first line shows the initial situation in the file and on the stack. Variable *ch* is initialized to the first character of the file, letter *s*, and in the first iteration of the loop, the character is simply ignored. This situation is shown in the second row. Then, the next character, 'equal sign' is read. It is also ignored and so is the letter *t*. After reading the left bracket, the bracket is pushed onto the stack so that the stack now has one element, the left bracket. Reading digit 5 does not change the stack, but after the right bracket becomes the value of *ch*, the topmost element is popped off the stack and compared with *ch*. Because the popped off element *(left bracket)* matches *ch (right bracket)*, the processing of input continues. After reading and discarding the letter *u*, a slash is read and the algorithm checks whether it is part of the comment delimiter by reading the next character, a left parenthesis. Because the character read in is not an asterisk, the slash is not the beginning of a comment, so *ch* is set to left parenthesis. In the next iteration, this parenthesis is pushed onto the stack and processing continues. After reading the last character, a semicolon, the loop is exited and the stack is checked. Because it is empty with no unmatched delimiters left, success is reached.

This now brings me to the subject of **First-Class ADTs**. As we know, ADTs help us manage the complexity of creating client programs that address the needs of increasingly more complicated applications by building increasingly powerful layers of abstraction. Throughout this process, it is often natural to want to use the data types in our programs in the same way that we use primitive types such as `int` or `float`. At this point in my guide, we consider the pitfalls that arise when we try to do so. The next question is, *what is a first-class abstract data type (ADT)?* A first-class data type *(also referred to as a first-class object)* is a concept in programming languages that denotes the level of support and flexibility a programming language provides for a particular data type or entity. First-class data types can be used the same way that we use primitive data types which means they also follow the same characteristics:

- Storage
- Assignment
- Passing as Arguments
- Return Value
- Equality
- Operations
- Encapsulation

If a first-class data type is accessed only through an interface, it is a first-class ADT. As a generalization, **<u>Java does not support first-class data types</u>** because its primitive *(built-in)* data-types are fundamentally different from its class *(user-defined)* data types. Java also provides direct language support for the *String* type, making it different from both primitive types and other class types. First, arithmetic operators such as + and * are defined for primitive data types (and + is defined for the String type), but we cannot arrange to write $a + b$ when a and b are objects of a user-defined type. Second, we can define methods for class types and extend them, but we cannot do either for primitive types. Third, the meaning of $a = b$ depends on whether or not a and b are primitive types: if they are primitive a gets a copy of the value of b; if not, a gets a copy of a reference to b. The same holds true of method parameters and return values. As with other definitions related to data types, we cannot be precise in defining the concept of first-class types without straying into deep issues relating to semantics of operations. It is one thing to expect that we are able to write $a = b$ when a and b are objects from a user-defined class, but it is quite another thing to precisely specify what we mean by that statement. Ideally, we would envision all data types having some universal set of well-defined methods. An example is the convention that all Java objects have a `toString` method. In practice, each data type is characterized by its own set of methods. This difference between data types in itself militates against a precise definition of the concept of first-class data types, because it implies that we should provide definitions for every operation that is defined for built-in types, which we rarely do. Most often, only a few crucial operations are of real importance to us, and we try to use those operations for our own data types in the same way as we do for built-in types.

To illustrate this, we consider an ADT for the complex-number abstraction. Our goal is to be able to write programs that perform algebraic operations on complex numbers using operations defined in the ADT. We would like to declare and initialize complex numbers and use arithmetic operations on complex numbers to perform various computations involving them. As mentioned above, we will not be able to write clients that use the arithmetic operators * and + on complex numbers; we will still have to define and use appropriate methods for these operations. Regardless, it is natural to want to compute with complex numbers in much the same way as we compute with real numbers or integers. We now briefly consider a few mathematical properties of complex numbers. The number of $i = \sqrt{-1}$ is an imaginary number. Although, $\sqrt{-1}$ is meaningless as a real number, we name it i and perform algebraic manipulations with i, replacing i^2 with -1 whenever it appears. A complex number consists of two parts, real and imaginary—complex numbers can be written in the form $a + bi$ where a and b are real complex numbers. To multiply complex numbers, we apply the usual algebraic rules, replacing i^2 with –1 whenever it appears. For example:

$$(a + bi)(c + di) = ac + bci + adi + bdi^2 = (ac - bd) + (ad + bc)i$$

The real or imaginary parts might cancel out (have the value 0) when we perform a complex multiplication. For example:

$$(1 - i)(1 - i) = 1 - i - i \; + \; i^2 = \; -2i,$$

$$(1 + i)^4 = 4i^2 = \; -4,$$

$$(1 + i)^8 = 16$$

Scaling the preceding equation by dividing through by $16 = (\sqrt{2})^8$, we find that

$$\mathrm{s}\left(\frac{1}{\sqrt{2}} + \frac{i}{\sqrt{2}}\right)^8 = 1$$

In general, there are many complex numbers that evaluate to 1 when raised to a power. These are the complex roots of unity. For each N, there are exactly N complex numbers z with $z^N = 1$. The formulas

$$cos\left(\frac{2\pi k}{N}\right) + i \; sin\left(\frac{2\pi k}{N}\right)$$

For $k = 0$, 1, …., $N - 1$ are easily shown to have this property. For example, taking $k = 1$ and $N = 8$ in this formula gives the particular eighth root of unity that we just discovered. As an example of a client, consider the task of writing a program that computes each of the Nth roots of unity for any given N and checks the computation by raising each of them to the Nth power.

So, what does all of this mean? As we continue our deep dive into ADTs we consider this process as part of the **complex roots of unity**. Let's look at some code to better illustrate the above mathematical formulas using a complex number driver *(roots of unity)*. The following program performs a computation on complex numbers using an ADT that allows it to compute directly with the abstraction of interest by using objects of *Complex* type. This code will check the ADT implementation by computing the powers of the roots of unity.

```
public class RootsOfUnity {
      public static void main(String[] args) {
            int N = Integer.parseInt(args[0]);
            Out.println(N + "roots of unity”);
            for (int k = 0; k < N; k++) {
                  double x = Math.cos(2.0 ∗ Math.PI ∗ k/N),
                  double y = Math.sin(2.0 ∗ Math.PI ∗ k/N);
                  Complex t = new Complex(x, y);
                  Out.println(k + ": ” + t);
                  Complex z = (Complex) t.clone();
                  for (int j = 0; j < N − 1; j++) z.mult(t);
                  Out.println(" ” + z);
            }
      }
}
```

With an appropriate `toString` method this code will print the following table:

```
0   1.000   0.000  1.000  0.000
1   0.707   0.707  1.000  0.000
2   0.000   1.000  1.000  0.000
3  -0.707   0.707  1.000  0.000
4  -1.000   0.000  1.000  0.000
5  -0.707  -0.707  1.000  0.000
6   0.000  -1.000  1.000  0.000
7   0.707  -0.707  1.000  0.000
```

This table gives the output that would be produced when invoked with '`a.out 8`' in a Linux *(Vim)* terminal with an implementation of the overloaded `toString` method.

Next, we could consider how to arrange and multiply two complex numbers. Ideally, we would want to write expressions like $a = b * c$; where a, b, and c are all of type *Complex*, but as I mentioned earlier, **Java does not support this style of programming**. One idea is to try to mimic this style by writing a static method that takes two `Complex` objects as parameters and returns a `Complex`, so that we can write:

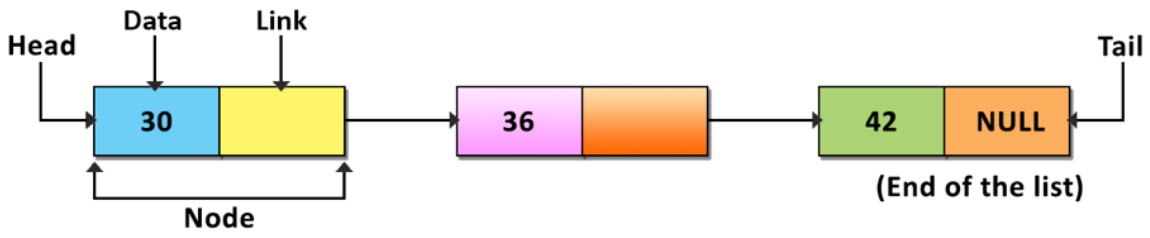
$a = Complex.mult(b, c);$ Another approach is to use a single-parameter class method `mult` that we use to multiply a `Complex` object by the given parameter. This approach mimics the use of expressions like a *= b with primitive types. It's essential to strike a balance when using complex objects as parameters. Overuse of complex objects can lead to overly complex and tightly coupled code, which can be challenging to maintain. Proper class design and careful consideration of the relationships between objects are crucial to reaping the benefits of using complex objects as parameters in Java.

As a programmer who's written data structures and formulated illuminating algorithms to light the path, you understand the important advantages that linked lists offer over linear data structures. We know that unlike arrays, linked lists are dynamic data structures, resizable at runtime that allow easy implementation for insertion and deletion operations. Let's consider a few complex ADTs in Java and how they are implemented and utilized in the Java programming language. A singly linked list is a dynamic data structure where each element *(node)* contains a value and a reference to the next element in the list. Here is an implemented example of a singly linked list:

```java
public class SinglyLinkedList<T> {
    private Node<T> head;

    private static class Node<T> {
        T data;
        Node<T> next;

        Node(T data) {
            this.data = data;
            this.next = null;
        }
    }

    public void add(T data) {
        Node<T> newNode = new Node<>(data);
        if (head == null) {
            head = newNode;
        } else {
            Node<T> current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newNode;
        }
    }

    public void display() {
        Node<T> current = head;
        while (current != null) {
            System.out.print(current.data + " -> ");
            current = current.next;
        }
        System.out.println("null");
    }
}
```

So, what is happening in this above code example? Let's take a look at a visual representation to better understand this process.

- The SinglyLinkedList class represents the ADT and contains a reference to the head of the list.
- The Node inner class defines the elements of the list, which include the data and a reference to the next node.
- The add method appends a new element to the end of the list.
- The display method is used to print the elements of the list for demonstration.



This collection of nodes are kept in random memory, and a node has two fields:

1. Saved data at a specific address
2. A pointer to the next node in the memory

As we can see, the null pointer is contained in the list's last node. Now, you've seen my example of a singly linked list, but what about a doubly linked list to maintain bidirectional links between nodes for easy traversal in both directions? We can create that as well with some slight modifications from our SinglyLinkedList!

```
public class DoublyLinkedList<T> {
    private Node<T> head;
    private Node<T> tail;

    private static class Node<T> {
        T data;
        Node<T> prev;
        Node<T> next;

        Node(T data) {
            this.data = data;
            this.prev = null;
            this.next = null;
        }
    }

    public void addFirst(T data) {
        Node<T> newNode = new Node<>(data);
        if (isEmpty()) {
            head = newNode;
            tail = newNode;
        } else {
            newNode.next = head;
            head.prev = newNode;
            head = newNode;
        }
    }

    public void addLast(T data) {
        Node<T> newNode = new Node<>(data);
        if (isEmpty()) {
            head = newNode;
            tail = newNode;
        } else {
            newNode.prev = tail;
            tail.next = newNode;
            tail = newNode;
        }
    }

    public void remove(T data) {
        Node<T> current = head;
        while (current != null) {
            if (current.data.equals(data)) {
                if (current == head) {
                    head = current.next;
                    if (head != null) {
                        head.prev = null;
                    }
                } else if (current == tail) {
                    tail = current.prev;
                    tail.next = null;
                } else {
                    current.prev.next = current.next;
                    current.next.prev = current.prev;
                }
                return;
            }
            current = current.next;
        }
    }

    public void display() {
        Node<T> current = head;
        while (current != null) {
            System.out.print(current.data + " <-> ");
            current = current.next;
        }
        System.out.println("null");
    }
    public boolean isEmpty() {
        return head == null;
    }
}
```

Do you see what is happening in the code? Here are the details covering the doubly linked list that I just created:

- The DoublyLinkedList class represents the ADT for a doubly linked list.
- The Node inner class defines the elements of the list, including the data and references to the previous and next nodes.
- addFirst adds an element to the beginning of the list, updating the head.
- addLast adds an element to the end of the list, updating the tail.
- remove allows you to remove a specific element from the list by searching for its value and adjusting the previous and next references of adjacent nodes.
- display prints the elements of the list from the head to the tail.
- isEmpty checks if the list is empty by verifying if the head is null.

This custom doubly linked list ADT allows us to manage a list of elements with efficient insertion and deletion at both the beginning and end of the list. We can also implement a binary search tree *(BST)* in Java with the following code:

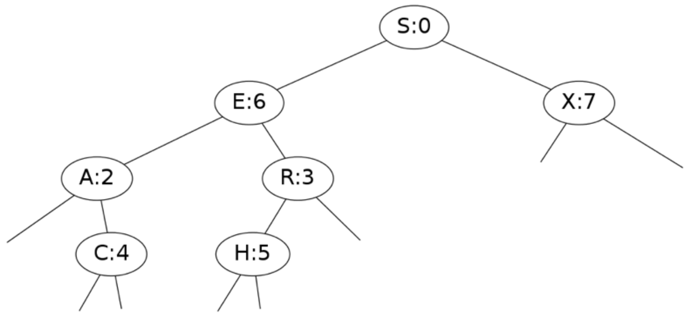
```java
public class BinarySearchTree<T extends Comparable<T>> {
    private TreeNode<T> root;

    private static class TreeNode<T> {
        T data;
        TreeNode<T> left;
        TreeNode<T> right;

        TreeNode(T data) {
            this.data = data;
            this.left = null;
            this.right = null;
        }
    }

    public void insert(T data) {
        root = insertRecursive(root, data);
    }

    private TreeNode<T> insertRecursive(TreeNode<T> root, T data) {
        if (root == null) {
            return new TreeNode<>(data);
        }

        if (data.compareTo(root.data) < 0) {
            root.left = insertRecursive(root.left, data);
        } else if (data.compareTo(root.data) > 0) {
            root.right = insertRecursive(root.right, data);
        }

        return root;
    }
    public void inOrderTraversal() {
        inOrderTraversalRecursive(root);
    }

    private void inOrderTraversalRecursive(TreeNode<T> root) {
        if (root != null) {
            inOrderTraversalRecursive(root.left);
            System.out.print(root.data + " ");
            inOrderTraversalRecursive(root.right);
        }
    }
}
```

Here's an explanation of how the above binary search tree works:

- The BinarySearchTree class represents the ADT and contains a reference to the root of the tree.
- The TreeNode inner class defines the elements of the tree, including the data and references to the left and right children.
- The insert method inserts a new element into the BST while maintaining the binary search tree property.
- The inOrderTraversal method performs an in-order traversal of the tree, which prints the elements in ascending order.

A visual representation from my lab notes about BSTs:

- S-node is root of tree
- E-node is parent of A-node (left) and R-node (right)
- E-node and all children nodes is a subtree
- E-node has key = E, value = 6
- All nodes in subtree rooted at A have keys smaller than E
- All nodes in subtree rooted at R have keys larger than E



Compared to what real-world assignments/projects require in software development for large-scale applications these are just simplified implementations from my COS 265 labs to illustrate the concepts of linked lists and binary search trees as ADTs in Java.

So, what has this deeper dive into ADTs taught us? In essence, we've come to know ADTs as the cornerstone of Java programming that promotes good software engineering practices, fosters code reusability, simplifies code development and maintenance, and enables efficient and reliable data management within your Java applications. They provide a structured and organized way to work with data, leading to more robust and maintainable software.

Union-Find Algorithms

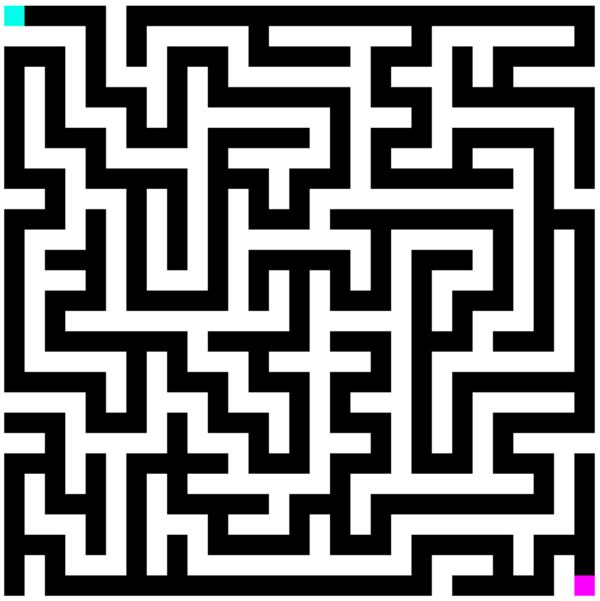
Suppose that we are given a sequence of pairs of integers, where each integer represents an object of some type and we are to interpret the pair p-q as meaning "p is connected to q." Given a set of N elements that support two operations:

1. **Connection command:** directly connect two elements with an edge
2. **Connection query:** is there a path connecting two elements?

We assume the relation "is connected to" to be transitive: If p is connected to q, and q is connected to r, then p is connected to r. Our goal is to write a program to filter out extraneous pairs from the set: When the program inputs a pair p-q, it should output the pair only if the pairs it has seen to that point do not imply that p is connected to q. If the previous pairs do imply that p is connected to q, then the program should ignore p-q and should proceed to input the next pair. The following integer example is a **dynamic connectivity example**. Given a sequence of pairs of intehers representing connections between objects *(left column)*, the task of a connectivity algorithm is to output those pairs that provide new connections *(center column)*. For example, the pair 2-9 is not part of the output because the connection 2-3-4-9 is implied by previous connections. The goal is to devise a program that can remember sufficient information about the pairs it has seen to be able to decide whether or not a new pair of objects is connected. Informally, we refer to the task of designing such a method as the dynamic connectivity problem. This problem arises in a large array of important applications. We can briefly consider some examples to indicate the fundamental nature of the problem. For example, these integers may represent computers in a large network, and the pairs might represent connections in a network. Then, the program may be used to determine whether we need to establish a new direct connection for p and q to be able to communicate or whether we could use existing connections to set up a communications path. In this type of application, we may need to process millions of points and billions of connections, or more. A problem like this will be nearly impossible to solve for such an application without a very efficient algorithm. Similarly, the integers may represent contact points in an electrical network, and the pairs may represent wires connecting the points. In this case, we could use our program to find a way to connect all the points without any extraneous connections *(if that's possible)*. There is no guarantee that the edges in the list will suffice to connect all the points—most certainly, it will be determined whether or not they could be a prime application of our program. Now, let's consider this program from a different perspective on a larger scale.

4-3	4-3	
4-9	4-9	
8-0	8-0	
3-8	3-8	
6-5	6-5	
2-3	2-3	
2-9		2-3-4-9
5-9	5-9	
5-0	5-0	
6-1	6-1	
5-7		5-7
0-2		0-8-4-3-2
5-7	5-7	

```
connect(4, 3)
connect(3, 8)
connect(6, 5)
connect(9, 4)
connect(2, 1)
isConnected(2, 9) // ?
isConnected(5, 7) // ?
connect(5, 0)
connect(7, 2)
connect(6, 1)
connect(1, 0)
isConnected(5, 7) // ?
```



The following maze is a representation of a larger connectivity example. The objects in a connectivity problem may represent connection points, and the pairs might be connections between them. As indicated in this example, they may represent wires connecting buildings in a city or components in a computer chip. This graphical representation makes it possible for a human to spot nodes that are not connected, but the algorithm has to work with only the pairs of integers that it's given. At closer observation of this larger example, can you see a path connecting the cyan and pink elements? Examining this figure gives us an appreciation of the difficulty of the connectivity problem: *How can we arrange to tell quickly whether any given two points in such a network are connected?* Still, another example arises in certain programming environments where it is possible to declare two variable names as equivalent. The problem is to be able to determine whether two given names are equivalent, after a sequence of such declarations. Applications such as the variable-name-equivalence problem described above require that we associate an integer with each distinct variable name.

The first step in the process of developing an efficient algorithm to solve a given problem is to *implement a simple algorithm that solves the problem*. If we need to solve a few particular instances of a problem that turn out to be easy, then the simple implementation may finish the job for us. If a more sophisticated algorithm is called for, then the simple implementation provides us with a correctness check for small cases and a baseline for evaluating performance characteristics. We always care about efficiency, but our primary concern in developing the first program that we write to solve a problem is to make sure that the program is a *correct* solution to the problem. The first idea that might come to mind is to somehow save all of the input pairs, then write a function to pass through them to try and discover whether the next pair of objects *is connected*. In this guide, I will use a different approach. First, the number of pairs may be sufficiently large to impede our preserving them all in memory in practical applications. Second, and more to the point, no simple method immediately suggests itself for determining whether two objects are connected from the set of all the connections, even if we could save them all. As a general rule of thought, we consider modeling the elements.
Applications involve manipulating elements of all types:

- pixels in a digital photo
- computers in a network
- friends in a social network
- transistors in a computer chip
- elements in a mathematical set

- variable names in a Fortran program
- metallic sites in a composite system
- modeling the elements

When programming, it is convenient to name elements *0* to *N-1*.

- use integers as array index
- suppress details not relevant to union-find

We model *"is connected to"* as an equivalence relation, which is reflexive, symmetric, and transitive.

Reflexive

p is connected to p

Symmetric

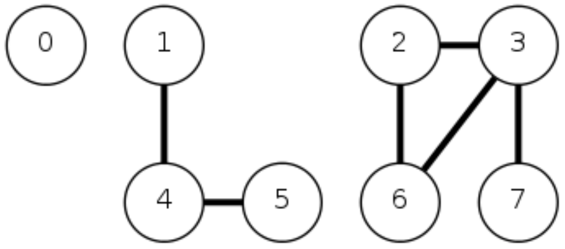
if p is connected to q, then q is connected to p

Transitive

if p is connected to q and q is connected to r, then p is connected to r

When considering connected exponents, we're only interested in the ***maximal set*** of elements that are mutually connected. Let's look at 3 disjoint sets per connected components.

$$\{0\}\ \{1, 4, 5\}\ \{2, 3, 6, 7\}$$



Example:

`union(p, q)`

replaces sets containing elements p and q with their union.

`Find(p)`

In which set is element `p`?

$$\{0\}\ \{1, 4, 5\}\ \{2, 3, 6, 7\} \quad \underset{\text{union(2,5)}}{\Rightarrow} \quad \{0\}\ \{1, 2, 3, 4, 5, 6, 7\}$$

```
find(5) != find(6)
union(2, 5)           // 3 disjoint sets -> 2 disjoint sets
find(5) == find(6)
```

Let's try a ***Quick-find*** solution to a connectivity problem. This program takes an integer N from the command line, reads a sequence of pairs of integers, interprets the pair p q to mean *"connect object p to object q,"* and prints the pairs that represent objects that are not yet connected. The program maintains the array id such that id[p] and id[q] are equal if and only if p and q are connected.

```
public class QuickF {
      public static void main(String[] args) {
            int N = Integer.parseInt(args[0]);
            int id[] = new int[N];
            for (int i = 0; i < N; i++) id[i] = i;
            for (In.init(); !In.empty();) {
                  int p = In.getInt(), q = In.getInt();
                  int t = id[p];
                  if (t == id[q]) continue;
                  for (int i = 0; i < N; i++)
                        if (id[i] == t) id[i] = id[q];
                  Out.println(" ” + p + " ” + q);
            }
      }
}
```

This method is simpler because for this particular example we are solving a less difficult problem, more efficiently because it does not require saving all of the pairs. They all use an array of integers—one corresponding to each object—to hold the requisite information to be able to implement ***union*** and ***find***. In the above example, I am using elementary arrays in their simplest form: I create an array that can hold N integers by writing `int id[] = new int[N];` then I refer to the ith integer in the array by writing `id[i], = new int[N] for 0 < i < 1000`.

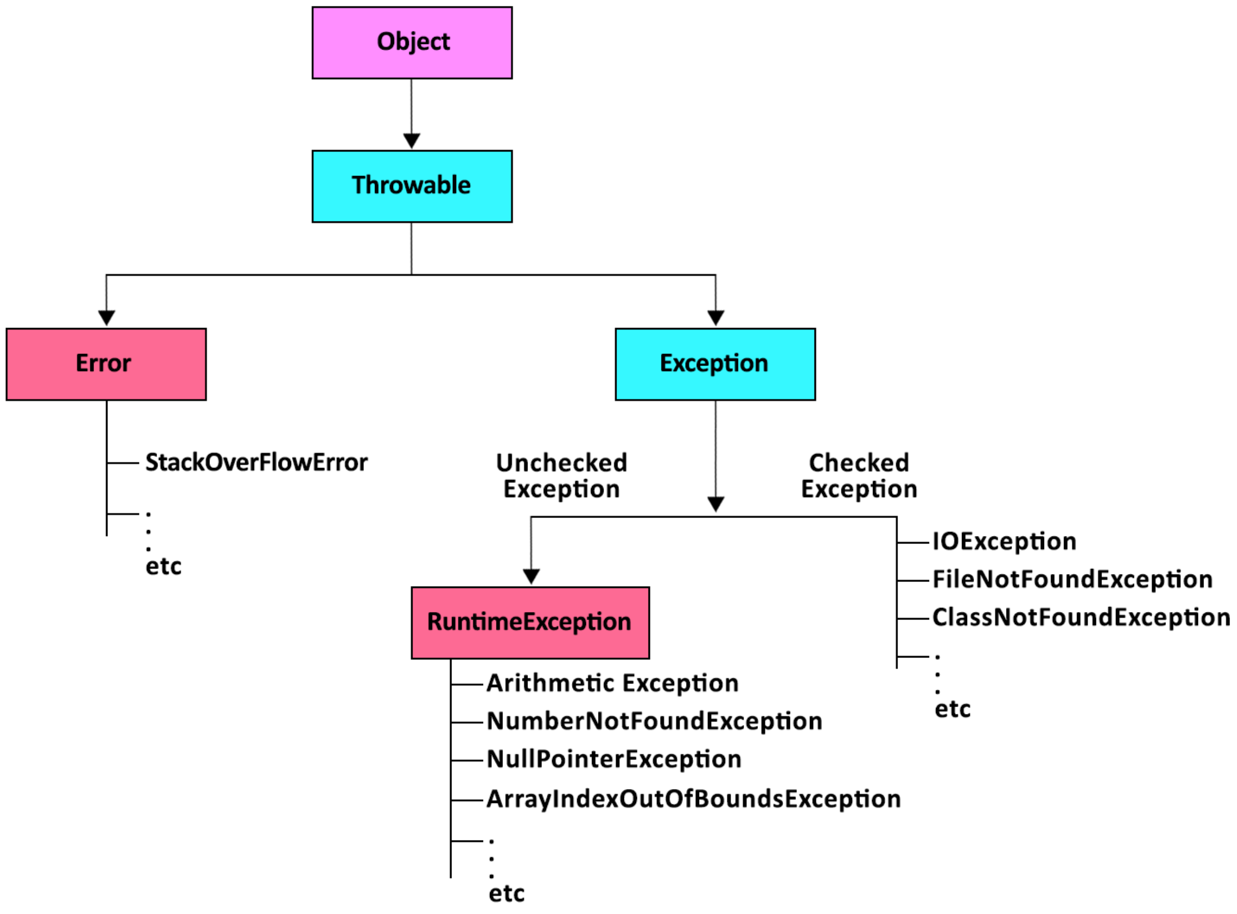
Now, if I replace the body of the `for` loop in the above QuickF function, by this code I could have a program that meets the same specifications but does less computation for the ***union*** operation at the expense of more computation for the ***find*** operation. The `for` loops and subsequent `if` statement in this code specify the necessary and sufficient conditions on the `id` array for p and q to be connected. The assignment statement `id[i] = j` implements the ***union*** operation.

```
int i, j, p = In.getInt(), q = In.getInt();
for (i = p; i != id[i]; i = id[i]);
for (j = q; j != id[j]; j = id[j]);
      if (i == j) continue;
      id[i] = j;
Out.println(" ” + p + " ” + q);
```

This is an implementation of the ***union*** and ***find*** operations that comprise the quick-union algorithm to solve the connectivity problem.

Exception Handling

Exception handling in Java has been a fundamental part of the language since its inception in 1995. Java introduced a robust exception-handling mechanism based on the concepts of try, catch, and throw to handle exceptional situations or errors that could occur during program execution. This approach was designed to provide a safer and more structured way to handle errors compared to some other programming languages that relied on error codes or other mechanisms. Exception handling has been a significant step towards writing more reliable and maintainable code. It allows developers to separate the normal flow of program execution from error handling, making it easier to identify and address issues in our code. This feature continues to be a key aspect of Java development. Here are the **Hierarchy of Exception** classes:



We all know exceptions are unexpected events that occur during the execution of a program. An exception can be the result of an error condition or simply an unanticipated input. In any case, in an object-oriented language, such as Java, exceptions can be thought of as being objects themselves. **Checked exceptions** are a type of exception that the compiler requires you to handle explicitly. These exceptions are associated with conditions that are generally outside the control of the program, and they typically represent errors or exceptional situations that might occur during the execution of your program. Checked exceptions are checked by the compiler at compile-time, and if not properly handled, they will result in a compilation error. **Unchecked exceptions**, also known as runtime exceptions, are exceptions that do not require explicit handling at compile time. Unlike checked exceptions, which the compiler enforces you to either catch or declare, unchecked exceptions are not checked by the compiler at compile time. Instead, they can occur at runtime and are typically the result of programming errors or unexpected conditions during program execution. In Java, exceptions are objects that are *thrown* by code that encounter some sort of unexpected condition. They can also be thrown by the Java run-time environment should it encounter an unexpected condition, like running out of object memory. A thrown exception is caught by other code that *"handles"* the exception somehow, or the program is terminated unexpectedly. For example, if we try to delete the tenth element from a sequence that has only five elements, the code may throw a `BoundaryViolationException`. This action could be done, for example, using the following code fragment:

```
if (insertIndex >= A.length) {
      throw new BoundaryViolationException("No element at index " +
insertIndex);
}
```

It is typically convenient to instantiate an exception object at the time the exception has to be thrown. Thus, a **throw** statement is generally written as follows:

```
throw new exception_type(param0, param1, ..., paramn-1);
```

where `exception_type` is the type of the exception and the param's form the list of parameters for a constructor for this exception. Exceptions are also thrown by the Java run-time environment itself. For example, the counterpart to the example above is `ArrayIndexOutOfBoundsException`. If we have a six-element array and ask for the ninth element, then this exception will be thrown by the Java run-time system. When it comes to the *throw clause*, it is appropriate to specify when a method is declared for the exceptions it might throw. This convention has both a functional and courteous purpose. For one, it lets users know what to expect. It also lets the Java compiler know which exceptions to prepare for. The following is an example of such a method definition:

```
public void goShopping() throws ShoppingListTooSmallException,
            OutOfMoneyException {

      //method body...

}
```

By specifying all the exceptions that might be thrown by a method, we prepare others to be able to handle all of the exceptional cases that might arise from using this method. Another benefit of declaring exceptions is that we do not need to catch those exceptions in our method. Sometimes this is appropriate in the case where other code is responsible for causing the circumstances leading up to the exception.

The following illustrates an exception that is "passed through":

```
public void getReadyForClass() throws ShoppingListTooSmallException,
            OutOfMoneyException {

      goShopping();  //no need to try or catch the exceptions

      makeCookiesForTA();
}
```

Do you see what's happening in this example? We pass the parameter `goshopping()` because it may *throw* therefore, `getReadyForClass()` will just pass these along. A function can declare that it throws as many exceptions as it likes. Such a listing can be simplified somewhat if all exceptions that can be thrown are subclasses of the same exception. In this case, we only have to declare that a method throws the appropriate superclass.

Java defines classes `Exception` and `Error` as subclasses of `Throwable`, which denotes any object that can be thrown and caught. Also, it defines class `RuntimeException` as a subclass of `Exception`. The `Error` class is used for abnormal conditions occurring in the run-time environment, such as running out of memory. Errors can be caught, but they probably should not be, because they usually signal problems that cannot be handled gracefully. An error message or a sudden program termination is about as much grace as we can expect. The `Exception` class is the root of the exception hierarchy. Specialized exceptions such as `BoundaryViolationException` should be defined by subclassing from either `Exception` or `RuntimeException`. Note that exceptions that are not subclasses of `RuntimeException` must be declared in the **throws** clause of any method that can throw them.

When an exception is thrown, it must be ***caught*** or the program will terminate. In any particular method, an exception in that method can be passed through to the calling method or it can be caught in that method. When an exception is caught, it can be analyzed and dealt with. The general methodology for dealing with exceptions is to "try" to execute some fragment of code that might throw an exception. If it does throw an exception, then that exception is caught by having the flow of control jump to a predefined catch block that contains the code dealing with the exception. The general syntax for a ***try-catch block*** in Java is as follows:

`try`
main_block_of_statements
`catch` *(exception_type$_1$ variable$_1$)*
block_of_statements$_1$
`catch` *(exception_type$_2$ variable$_2$)*
block_of_statements$_2$
`finally`
block_of_statements$_n$

where there must be at least one `catch` part, but the **finally** part is optional. Each `exception_type` is the type of some exception, and each variable is a valid Java variable name. The Java run-time environment begins performing a **try-catch** block such as this by executing the block of statements, *main_block_of_statements*. If this execution generates no exceptions, then the flow of control continues with the first statement after the last line of the entire **try-catch** block, unless it includes an optional **finally** part. The **finally** part, if it exists, is executed regardless of whether any exceptions are thrown or caught. Thus, in this case if no exception is thrown, execution progresses through the **try-catch** block, jumps to the **finally** part, and then continues with the first statement after the last line of the **try-catch** block.

If, on the other hand, the block, *main_block_of_statements*, generates an exception, then execution in the **try-catch** block terminates at that point and execution jumps to the catch block whose `exception_type` most closely matches the exception thrown. The variable for this catch statement references the exception object itself, which can be used in the block of the matching catch statement. Once execution of that catch block completes, control flow is passed to the optional finally block, if it exists, or immediately to the first statement after the last line of the entire try-catch block if there is no finally block. Otherwise, if there is no catch block matching the exception thrown, then control is passed to the optional finally block, if it exists, and then the exception is thrown back to the calling method.
Let's consider the following example code:

```
int index = Integer.MAX_VALUE;

try {
   string toBuy = shoppingList[index];
}
catch (ArrayIndexOutOfBoundsException aio obx) {
   system.out.println ("The index " +index+ " is outside
the array.");
}
```

Do you understand what this code is doing? Essentially, if this code does not **catch** a thrown exception, the flow of control will immediately exit the method and return to the code that called our method. At that point, the Java run-time environment will look again for a **catch** block. If there is no **catch** block in the code that called this method, the flow of control will jump to the code that called this, and so on. Eventually, if no code catches the exception, the Java run-time system *(the origin of our program's flow of control)* will catch the exception. At this point, an error message and a stack trace are printed to the screen and the program is terminated. The following is an actual run-time error message that I've received before:

```
java.lang.NullPointerException: Returned a null locator
at java.awt.Component.handleEvent(Component.java:621)
at java.awt.Component.postEvent(Component.java:542)
at java.awt.Component.postEvent(Component.java:539)
```

```
 at sun.awt.motif.MButtonPeer.action(MButtonPeer.java:27)
 at java.lang.Thread.run(Thread.java)
```

While this error message is not directly related to the above exception example, it still provides a great insight into the topic at hand. Once an exception is caught, there are several things a programmer might want to do. One possibility is to print out an error message and terminate the program. There are also some interesting cases in which the best way to handle an exception is to ignore it (this can be done by having an empty **catch** block). In this scenario, typically what we do is ignore the exceptions. For example, when the programmer does not care whether there was an exception or not. Another legitimate way of handling exceptions is to create and throw another exception, possibly one that specifies the exceptional condition more precisely. The following is an example of this approach:

```
catch (ArrayIndexOutOfBoundsException aio obx) {
throw new ShoppingListTooSmallException(
        "Product index is not in the shopping list");
}
```

Perhaps what we should consider as the best way to handle an exception *(although not always possible)* is to find the problem, fix it, and continue execution. As a recap on exception handling, we emphasize this fundamental concept because of these key features of better programming practices:

1. **Resource Management**
   Ensuring that files, network/database connections are properly closed, preventing resource leaks, and managing system resources more efficiently.

2. **Program Robustness**
   Handles vast amounts of inputs and conditions without failure, making software more reliable and stable.

3. **Error Resilience**
   Instead of crashing or abruptly terminating, the program can *catch* and manage these exceptions, ensuring a better user experience, and preventing data loss or corruption in output.

4. **Modularity and Code Organization**
   promotes cleaner code by separating error-handling logic from the main program flow.

5. **Customization**
   Custom exception classes represent application-specific errors, making it easier to differentiate between different error scenarios.

Merging & Mergesorts

In this section of my guide, I examine a family of sorting algorithms based on a complementary process, *merging*—combining two ordered files to make one larger ordered file. Merging is the basis for a straightforward ***divide-and-conquer*** sorting algorithm and for a bottom-up counterpart, both of which are fairly easy to implement. Selection and merging are complementary operations in the sense that the selection splits a file into two independent files to make one file. The contrast between these operations also becomes apparent when we apply the divide-and-conquer paradigm to create a sorting method. We can rearrange the file such that, when two parts are sorted, the whole file is ordered. Alternatively, we can break the file into two parts to be sorted and then combine the ordered parts to make the whole ordered file. In this portion of my guide, we will look at ***mergesort***, which is quicksort's complement in that it consists of two recursive calls followed by a merging procedure. One of mergesort's most attractive properties is that it sorts a file of N elements in time proportional to $N\,log\,N$, no matter what the input. As such, consider the following proposition:
Merge sort uses $\leq N \lg N$ compares to sort an array of length N.
Pf sketch: The number compares $C(N)$ to merge sort an array of length N satisfies the recurrence.

$$C(N) \leq \underbrace{C(\lceil N/2 \rceil)}_{\text{sort left half}} + \underbrace{C(\lfloor N/2 \rfloor)}_{\text{sort right half}} + \underbrace{N}_{\text{merge}} \text{ for } N > 1, \text{ with } C(1) = 0$$

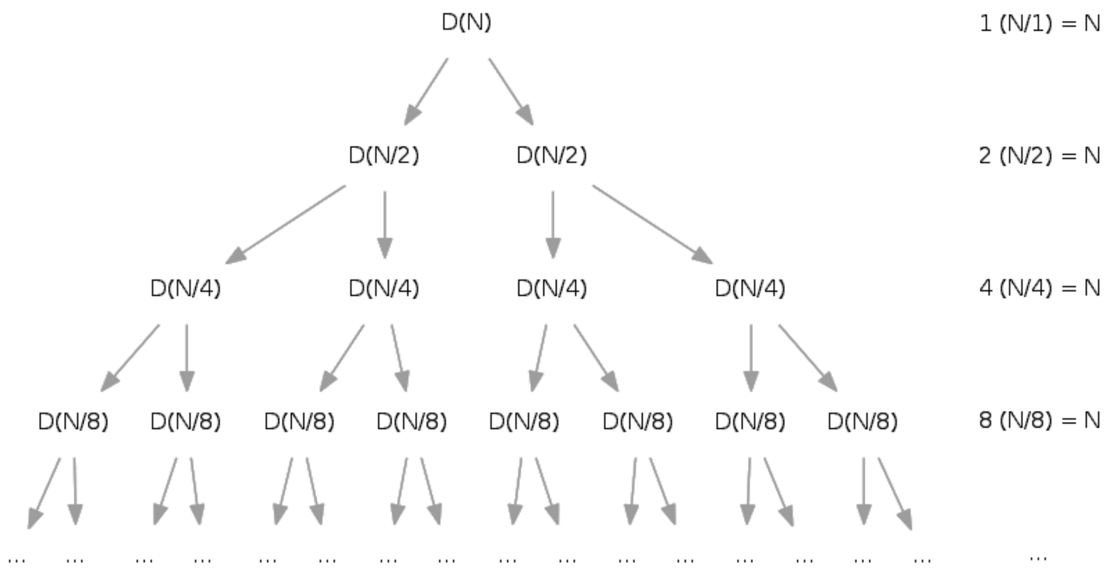
We solve for the recurrence when N is a power of 2 (result holds for all N, analysis cleaner in this case):

$$D(N) = 2D(N/2) + N, \text{ for } N > 1, \text{ with } D(1) = 0$$

We also consider for divide-and-conquer recurrence with the following proposition:

If $D(N)$ satisfies $D(N) = 2D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$

Pf by picture (assuming N is a power of 2):



Height of binary tree: $\lg N$
Recurrence: $D(N) = N \lg N$

We also consider the following proposition with a merge sort in a number of array accesses:
Merge sort uses $\leq 6N \lg N$ array accesses to sort an array of length N.
Pf sketch: The number of array accesses $A(N)$ satisfies the recurrence:

$$A(N) \leq A(\lceil N/2 \rceil) + A(\lfloor N/2 \rfloor) + 6N \text{ for } N > 1, \text{ with } A(1) = 0$$

Key point: any algorithm with the following structures takes $N \log N$ time

```
public static void linearithmic(int N) {
    if(N == 0) return;
    linearithmic(N/2);        // solve two problems
    linearithmic(N/2);        //   of half the size
    linear(N);                // do a linear amount of work
}
```

A guaranteed $N \log N$ running time can be a liability. For example, there are methods that can adapt to run in linear time in certain special situations, such as where there is a significant amount of order in the file, or when there are only a few distinct keys. In contrast, the running time of mergesort depends primarily on only the number of input keys and is virtually dismissive to their order. Mergesort is a stable sort and this feature tips the balance in its favor for applications where stability is the upmost importance. Competitive methods such as a quicksort or heapsort are not stable. Various techniques to make such methods stable tend to require extra space; whereas mergesort's extra space requirement thus becomes less significant if stability is a prime consideration.

To combine two ordered arrays a and b into an ordered array c, we use a for loop that puts an element into c at each iteration. If a is exhausted, the element comes from a; and if items remain in both, the smallest of the remaining elements in a and b goes to c. This implementation of **merging** assumes that the array c is disjoint *(does not overlap or share storage)* from a and b.

```
static void mergeAB(ITEM[] c, int cl,
                    ITEM[] a, int al, int ar,
                    ITEM[] b, int bl, int br)
    { int i = al, j = bl;
      for (int k = cl; k < cl + ar − al + br − bl + 1; k++) {
            if (i > ar) { c[k] = b[j++]; continue; }
            if (j > br) { c[k] = a[i++]; continue; }
            c[k] = less(a[i], b[j]) ? a[i++] : b[j++];
          }
    }
```

Once we have a merging procedure in place, it's not difficult to use that procedure as the basis for a recursive sorting procedure. To sort a given file, we divide it in half, recursively sort the two halves, and then merge them. A top-down mergesort is comparable to a top-down management style, where a manager gets an organization to take on a big task by dividing it into pieces to be solved independently by underlings. If each manager operates by simply dividing the given task in half, then putting together the solutions that the subordinates develop and passing the result up to a superior, the result is a process like **mergesort**. As an example of a top-down mergesort, this basic implementation is a prototypical *divide-and-conquer* recursive program. It sorts the array a[1],….,a[r] by dividing it into two parts a[1],….,a[m] and a[m+1],…..,a[r], sorting them independently *(via recursive calls)*, and merging the resulting ordered subfiles to produce the final ordered result.

```
static void mergeAB(ITEM[] a, int l, int r) {
      if (r <= l) return;
      int m = (r + l)/2;
      mergesort(a, l, m);
      mergesort(a, m + l, r);
      merge(a, l, m, r);
}
```

```
input:                 M E R G E S O R T E X A M P L E
                      /              / \              \
two halves:         M E R G E S O R  | |  T E X A M P L E
sort left half:     E E G M O R R S <==   T E X A M P L E
sort right half:    E E G M O R R S   ==> A E E L M P T X
merge results:      A E E E E G L M  | |  M O P R R S T X
                      \              \ /              /
merged:                A E E E E G L M M O P R R S T X
```

Every recursive program has a non-recursive analog that, although equivalent, may perform computations in a different order. As prototypes of the divide-and-conquer algorithm-design philosophy, nonrecursive implementations of mergesort are worth studying in detail. Consider the sequence of merges done by the recursive algorithm. A file size of 15 numerical objects is sorted by the following sequence of merges:

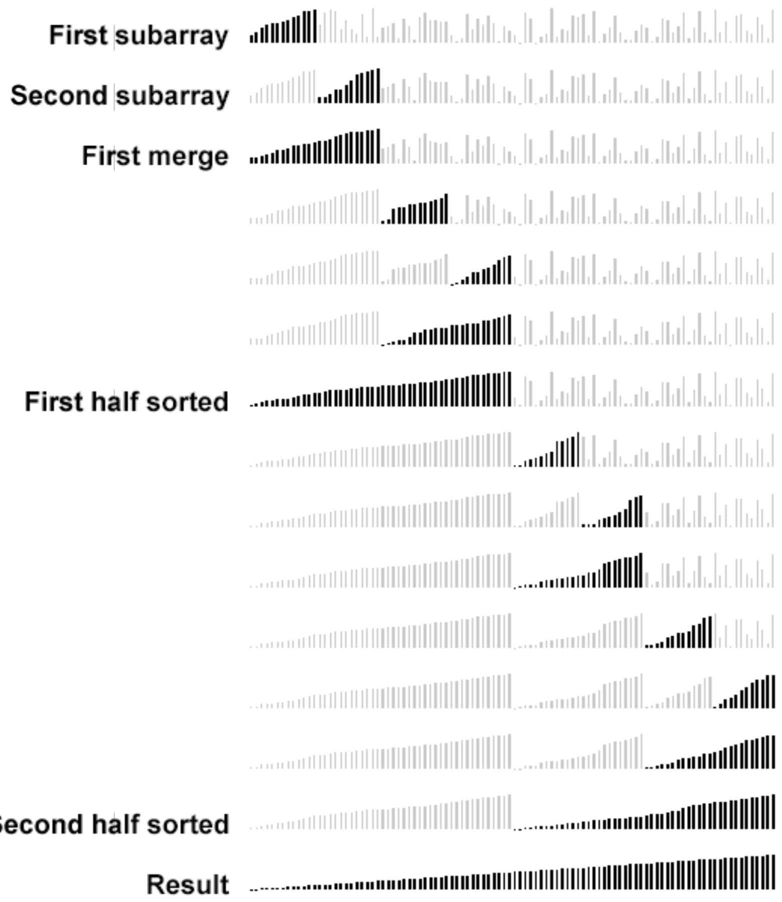
1-by-1	1-by-1	2-by-2	1-by-1	1-by-1	2-by-2	4-by-4
1-by-1	1-by-1	2-by-2	1-by-1	2-by-1	4-by-3	8-by-7

This order of the merges is determined by the recursive structure of the algorithm. However, the subfiles are processed independently, and the merges can be done in different sequences. A **bottom-up mergesort** consists of a sequence of passes over the whole file doing `m-by-m` mergers, doubling m on each pass. The final subfile is of size m only if the file size is an even multiple of m, so the final merge is an `m-by-r` merge, for some x less than or equal to m.

```
static int min(int A, int B) {
      return (A < B) ? A : B; }
static void mergesort(ITEM[] a, int l, int r) {
      if (r <= l) return;
      aux = new ITEM[a.length];
      for (int m = 1; m <= r – l; m = m+m)
            for (int i = l; i <= r – m; i += m+m)
                  merge(a, i, i+m, min(i+m+m–1, r));
}
```

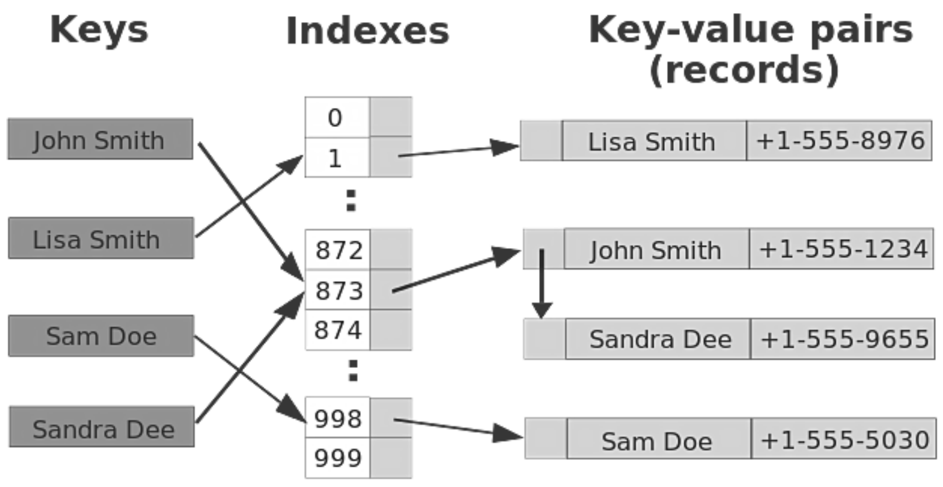
We can also use insertion sorts for small subarrays. Mergesorts can also have too much overhead for tiny subarrays. Ergo, we can create a **cutoff** to insertion sort for ≈ 10 items:

```
private static void sort(Comparable[] a, Comparable[] aux,
                                 int lo, int hi) {
    if(hi <= lo + CUTOFF - 1) {      // cut off to insertion sort
        Insertion.sort(a, lo, hi);
        return;
    }
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```



Hash Tables in Java

Hash tables in Java are versatile data structures that offer efficient key-value storage and retrieval, making them invaluable for a wide range of applications, from data storage and retrieval to algorithm optimization and performance improvement. Java provides the `HashMap` and `Hashtable` classes for implementing hash tables, and they are widely used in Java programming. Search algorithms are based on an abstract comparison operation, and search algorithms that use hashing consist of two separate parts. The first step is to compute a *hash function* that transforms the search key into a table address. Ergo, the second part of a hashing search is a *collision-resolution* process that deals with such keys. One of the collision-resolution methods that you will see in this guide use linked lists and as such immediately useful in dynamic situations where the number of search keys is difficult to predict in advance. The other two collision-resolution methods that you will see achieve fast search times on items that are stored within a fixed array. Hashing is a good example of a *time-space tradeoff*. If there were no memory limitation, then we could perform any search with only one memory access by simply using the key as a memory address, as in key-indexed search.



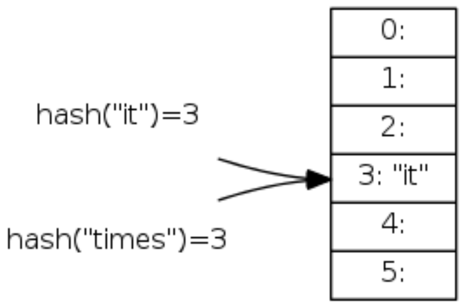
Hashing is a classical computer science problem: The various algorithms have been studied in depth and are widely used. It is not unreasonable to expect to support the *search* and *insert* symbol-table operations in *constant* time, independent of the size of the table.

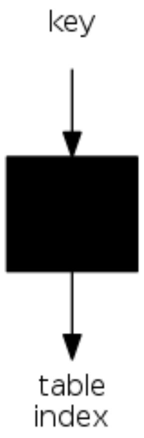
implementation	search*	insert*	delete*	search†	insert†	delete†	ordered	ops on keys
seq search (unordered list)	N	N	N	N	N	N		`equals()`
binary search (ordered array)	$\log N$	N	N	$\log N$	N	N	X	`compareTo()`
BST	N	N	N	$\log N$	$\log N$	$\sqrt{N}$	X	`compareTo()`
2-3 tree	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	X	`compareTo()`
LLRB	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	X	`compareTo()`

This expectation is the theoretical optimum performance for any symbol-table implementation, but hashing is not a *one-size-fits-all* solution for two primary reasons. First, the running time does depend on the length of the key, which can be a liability in practical applications with long keys. Second, hashing does not provide efficient implementations for other symbol-table operations, such as *select* or *sort*.

The first problem to consider is the computation of the **hash function**, which transforms keys into table addresses. This arithmetic computation is typically a straight forward process to implement. If we have a table that can hold M items, then we need a function that transforms keys into integers in the range of [0, $M - 1$]. The basic plan is to save items in a key-indexed table *(index is a function of the key)*

- Computing the hash function
- Equality test: Method for checking whether two keys are equal
- Collision resolution: Algorithm and data structure to handle two keys that hash to the same array index.



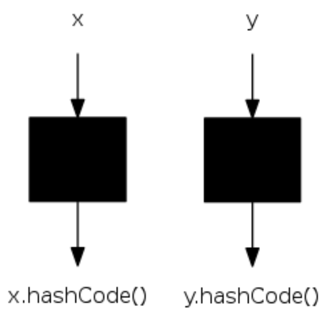


To compute the hash function, the ideal goal is to scramble the keys uniformly to produce a table index.

- Efficiently computable
- Each table index equally likely for each key
  (thoroughly researched problem, still problematic in practical applications)

All Java classes inherit a method `hashCode()`, returns a 32-bit `int`

- **Requirement:** `if x.equals(y), then x.hashCode() == y.hashCode()`
- **Highly desirable:** `if !x.equals(y), then x.hashCode() != y.hashCode()`
- **Default implementation:** Memory address of x
- **Legal *(but poor)* implementation:** Always return 17
- **Customized implementations:** Integer, Double, String, File, URL, Date, ...
- **User-defined types:** Users are on their own



Here is a Java library implementation of hash code:

```java
public final class Integer {
    private final int value;
    // ...
    public int hashCode() { return value; }
}
```

Integers

```java
public final class Boolean {
    private final boolean value;
    // ...
    public int hashCode() {
        if(value) return 1231;
        else      return 1237;
    }
}
```

Booleans

```java
public final class Double {
    private final double value;
    // ...
    public int hashCode() {
        long bits = doubleToLongBits(value);
        return (int) (bits ^ (bits >>> 32));
    }
}
```

Doubles

- convert to IEEE 64-bit representation
- xor most significant 32-bits with least significant 32-bits
- warning: -0.0 and +0.0 have different hash codes
    - 00000000000000000000000000000000b = 0.0
    - 10000000000000000000000000000000b = -0.0
    - 32 bit float

Treat string of length L as L-digit, base-31 number:

$$\begin{aligned} h &= s[0]\cdot 31^{L-1} + \ldots + s[L-3]\cdot 31^{2} + \\ &\quad + s[L-2]\cdot 31^{1} + s[L-1]\cdot 31^{0} \end{aligned}$$

```java
public final class String {
    private final char[] s;
    // ...
    public int hashCode() {
        int hash = 0;
        for(int i=0; i<length(); i++)
            hash = s[i] + (31 * hash);
        return hash;
    }
}
```

char	Unicode
...	...
a	97
b	98
c	99
...	...

Horner's method: only L multiples/adds to hash string of length L

```
String s = "call";
s.hashCode();         // 3045982 = 99*31^3 + 97*31^2 +
                      //             + 108*31^1 + 108*31^0
                      //           = 108 + 31*(108 + 31*(97 + 31*(99))
```

Implementing hash code for *strings* for performance optimization:

- Cache the hash value in an instance variable
- Return cached value

```
public final class String {
    private int hash = 0;                   // cache of hash code
    private final char[] s;
    // ...
    public int hashCode() {
        int h = hash, i;
        if(h != 0) return h;                // return cached value
        for(i = 0; i < length(); i++)
            h = s[i] + (31 * h);
        hash = h;                           // store cache of hash code
        return hash;
    }
}
```

In summary, to use hashing for an abstract symbol-table implementation, the final step is to extend the abstract type interface to include a `hash` operation that map keys into nonnegative integers less than M, the table size. Here is the implementation:

```
static int hash(double v, int M) {
      return (int) M*(v-s)/(t-s); }
```

completes the job for floating-point keys between the values s and t; for integer keys, we can simply return v % M. If M is not prime, the hash function might return

```
(int) (.616161 * (double) v) % M
```

or the result of a similar integer computation such as:

```
(16161 * v) % M
```

All of the above functions for string keys, are respectable approaches that typically spread out the keys and have served programmers well for years. Universal methods are a distinctive improvement for string keys that provides random hash values.

Understanding Radix in Java

In Java, "radix" typically refers to the base of a numbering system, such as binary (base 2), decimal (base 10), or hexadecimal (base 16). The term "radix" is often used when working with number representations other than the familiar base 10 decimal system. It's used in various contexts, including when converting numbers between different bases or specifying the base for formatting numbers. Here are some common uses for Radix in Java:

1. **Radix in Number Formatting:**
   When formatting numbers in Java, you can specify the radix to display the number in a different base. The `Integer.toString(int i, int radix)` and `Long.toString(long l, int radix)` methods allow you to convert an integer or long to a string representation in the specified radix. For example, to convert the decimal number 10 to binary, you would use:

   ```
   String binaryString = Integer.toString(10, 2);
   ```
   This sets the radix to 2, resulting in the string "1010."

2. **Parsing Numbers in Different Radices:**
   Java allows you to parse numbers from strings in different radices using methods like Integer.parseInt(String s, int radix) and Long.parseLong(String s, int radix). For example, to parse a binary string "1010" into an integer, you would use:

   ```
   int decimalValue = Integer.parseInt("1010", 2);
   ```
   This sets the radix to 2, indicating that the input string is in binary representation.

3. **Radix Constants:**
   Java provides constants for commonly used radices, such as Character.MIN_RADIX (which is typically 2 for binary) and Character.MAX_RADIX (which is typically 36, representing numbers using letters as well, e.g., hexadecimal). Here's an example that demonstrates converting a number from one radix to another in Java:

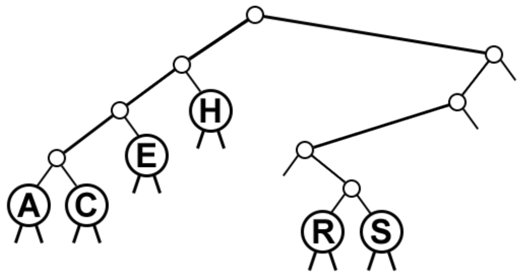
   ```
   public class RadixExample {
       public static void main(String[] args) {
           int decimalNumber = 10;
           String binaryString = Integer.toString(decimalNumber, 2);
           String hexadecimalString = Integer.toString(decimalNumber, 16);

           System.out.println("Decimal: " + decimalNumber);
           System.out.println("Binary: " + binaryString);
           System.out.println("Hexadecimal: " + hexadecimalString);
       }
   }
   ```

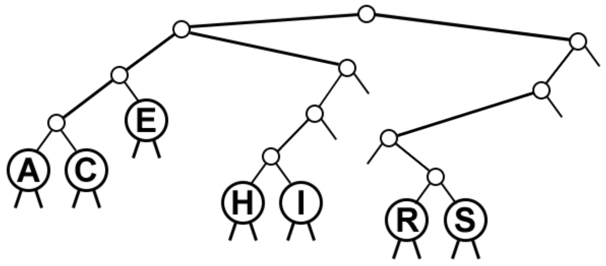
This code converts the decimal number 10 into its binary and hexadecimal representations using the Integer.toString() method with different radices.

When implementing radices, we can use tries for keys that are either a fixed number of bits or are variable-length bitstrings. Tries are a search tree that allows us to use the bits of the keys to guide the search. The idea is to store keys only at the bottom of the tree, in leaf nodes. The resulting data structure has a number of useful properties and serves as the basis for several effective search algorithms. In a trie, we keep the keys in the *leaves* of a binary tree. In a binary tree, a leaf is an internal node whole left and right links are both null. Keeping keys in leaves instead of internal nodes allows us to use the bits of the keys to guide the search.

In a successful search for the key H = 01000 in this sample trie *(top)*, we move left at the root *(since the first bit in the binary representation of the key is 0)*, then right *(since the second bit is 1)*, where we find H, which is the only key in the tree that begins with **01**. None of the keys in the trie begin with **101** or **11**; these bit patterns lead to the two null links in the trie that are in the non-leaf nodes. To insert **I** *(bottom)*, we need to add **three non-leaf** nodes:

- One corresponding to **01**, with a null link corresponding to **011**
- One corresponding to **010**, with a null link corresponding to **0101**
- One corresponding to 0100 with H = 01000 in a leaf on its left and I = 01001 in a leaf on its right





Each key in the trie is sorted in a leaf, on the path described by the leading bit pattern of the key. Conversely, each leaf contains the only key in the trie that begins with the bits defined by the path from the root to that leaf. Here is an implemented example of a **trie search**:

```
private ITEM searchR(Node h, KEY v, int d) {
      if (h == null) return null;
      if (h.l == null && h.r == null) {
            if (equals(v, h.item.key()))
                  return h.item; else return null; }
      if (bit(v, d) == 0)
           return searchR(h.l, v, d+1);
      else return searchR(h.r, v, d+1);
   }
ITEM search(KEY key) {
      return searchR(head, key, 0); }
```

This code should be fairly straight forward to you. This method uses bits of the key to control the branching on the way down the trie. Essentially it yields three possible outcomes:

1. If the search reaches a leaf *(with both links null)*, then that is the unique node in the trie that could contain the record with key v, so we test whether that node does in fact contain v *(search hit)* or some key whose leading bits match v *(search miss)*.
2. If the search reaches a null link, then the parent's other link must not be null, so there is some other key in the trie that differs from the search key in the corresponding bit, and we have a search miss.
3. The above code example assumes the final outcome, which is that the keys are distinct and *(if the keys may be of different lengths)* that no key is a prefix of another. The `item` member is not used in non-leaf nodes.

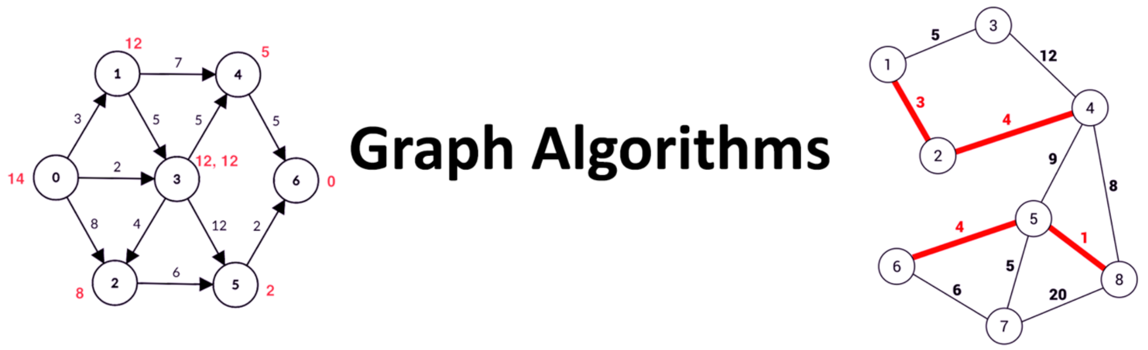
The next portion of understanding radices is the insertion of a key into a trie. However, we must first perform the search, as usual. If the search ends on a null link, we replace that link with a link to a new leaf containing the key. But if the search ends on a leaf, we need to continue down the trie, adding an internal node for every bit where the search key and the key that was found agree, ending with both keys in leaves as children of the internal node corresponding to the first bit position where they differ. Here is an example of a **trie insertion**:

```
Node split(Node p, Node q, int d) {
      Node t = new Node (null);
      KEY v = p.item.key(), w = q.item.key();
      switch(bit(v, d)*2 + bit(w, d)) {
            case 0: t.l = split(p, q, d+1); break;
            case 1: t.l = p; t.r = q; break;
            case 2: t.r = p; t.l = q; break;
            case 3: t.r = split(p, q, d+1); break;
      }
    return t;
}
private Node insertR(Node h, ITEM x, int d) {
      if (h == null)
         return new Node(x);
      if (h.l == null && h.r == null)
         return split(new Node(x), h, d);
      if (bit(x.key(), d) == 0)
           h.l = insertR(h.l, x, d+1);
      else h.r = insertR(h.r, x, d+1);
      return h;
}
void insert(ITEM x) {
      head = insertR(head, x, 0); }
```

Do you see what is happening in this code? We search, then distinguish the two cases that can occur for a search miss. If the miss was not on a leaf, then we replace the null link that caused us to detect the miss with a link to a new node. If the miss was on a leaf, then we use a method `split` to make one new internal node for each bit position where the search key and the key found agree, finishing with one internal node for the leftmost bit position where the keys differ. The `switch` statement in `split` converts the two bits that it is testing into a number to handle the four possible cases. If the bits are the same (case 00_2 = 0 or 11_2 = 3), then we continue splitting; if the bits are different (case 01_2 = 1 or 10_2 = 2), then we stop splitting. We do not access null links in leaves, and we do not store items in non-leaf nodes, so we could save space by using a pair of derived classes to define nodes as being one of these two types.

Graph Algorithms Visually Explained

In Data Structures & Algorithms, graphing algorithms are a pivotal learning point in computer science and carries over into preparing you for any potential steps that you may take into machine learning. This is one more example of where Data Structures & Algorithms is an important subject to study in the field of computer science. Graphs serve real-world applications such as modeling and capturing data with web pages and links, transportation and logistics, finance and fraud detection, biology and bioinformatics, and even game development. If you have relational objects/nodes, they can be represented using graphs.

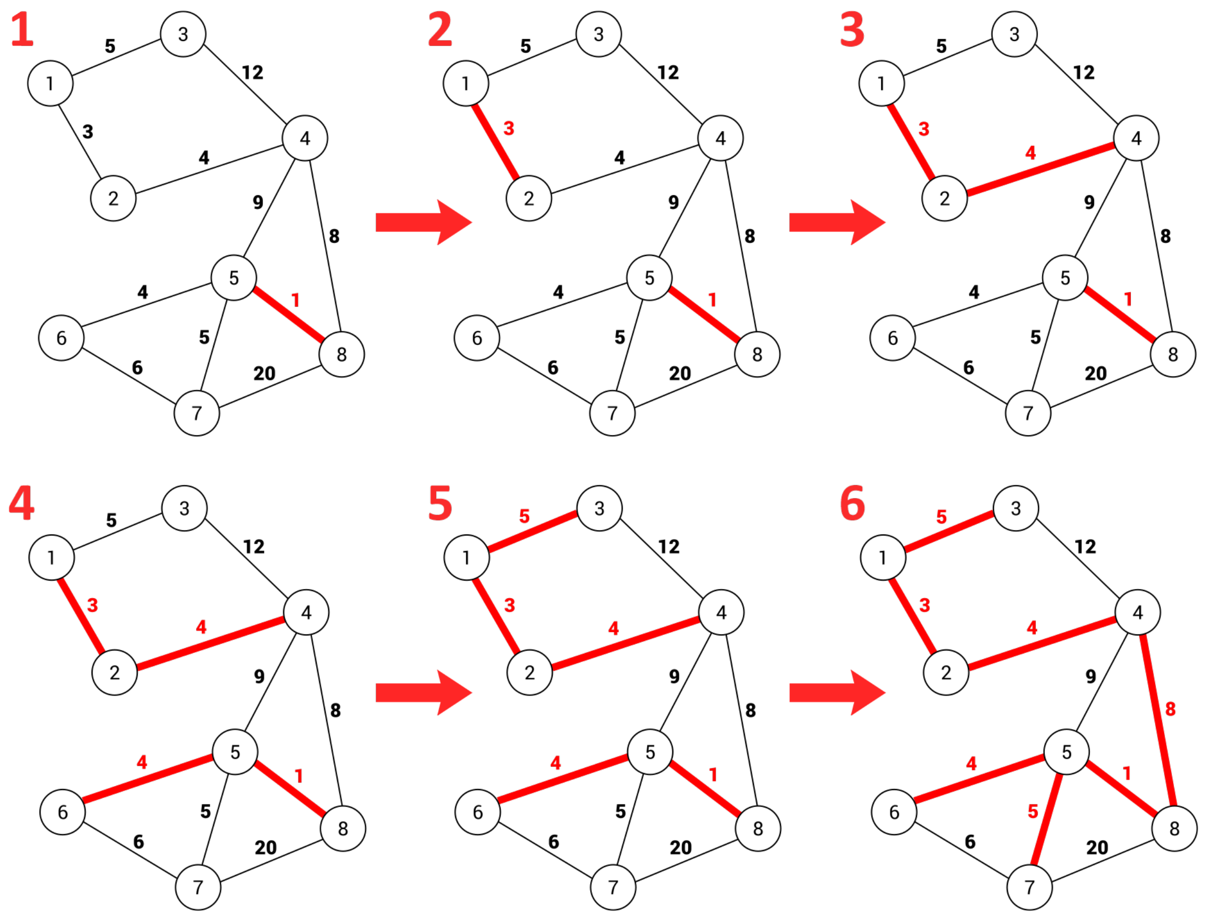


In computer science and mathematics, a **graph** is a data structure consisting of a set of vertices *(or nodes)* connected by a set of edges *(or arcs)*. Graph algorithms are fundamental in computer science and have wide-ranging applications including computer networking, recommendation systems, operations research, and scientific computing. It's important to understand that there are multiple types of graphs created in different formats *(even color-coded)* depending on the purpose they serve and the type of data they represent. Graph algorithms are used to perform various operations on graphs including:

- **Traversal** - systematically visiting the vertices *(nodes)* and edges of a graph. Depth-First Search *(DFS)* and Breadth-First Search *(BFS)*
- **Topological Sorting** - linearly order the vertices *(nodes)* of a directed acyclic graph *(DAG)*
- **Minimum Spanning Tree** - MSTs find applications in various fields, including network design, transportation, and clustering
- **Shortest Path** - used to find the shortest path between two vertices *(nodes)* in a graph
- **Connectivity** - refers to the property of determining whether there is a path or a connection between two vertices *(nodes)* in a graph
- **Matching** - a set of edges in a graph such that no two edges share a common vertex
- **Graph Coloring** - involves assigning colors to the vertices of a graph in such a way that no two adjacent vertices share the same color
- **Cluster Detection** - identifies groups of nodes *(vertices)* within a graph that are often referred to as clusters or communities; revealing hidden structures, relationships, and patterns in the data

We can assume that a vertex can have a name and can carry other associated information. Similarly, words like *arc*, *edge*, and *link* are all widely used by mathematicians to describe the abstraction embodying a connection between two vertices, but we consistently use *edge* when discussing graphs and *link* when discussing references in Java data structures.

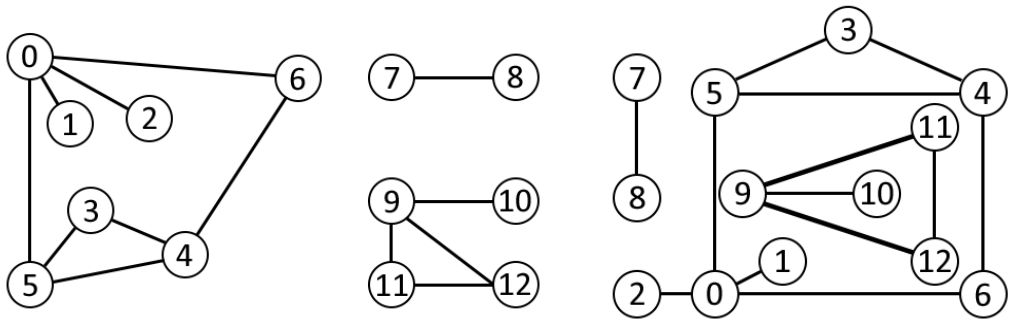
Example: When there is an edge connecting two vertices, we say that the vertices are *adjacent* to one another and that the edge is *incident* on both vertices. **The degree of a vertex is the number of edges incident on it.** We use the notation $v - w$ to represent an edge that connects v and w; the notation $w - v$ is an alternative way to represent the same edge.



Kruskal's Algorithm for finding Minimal Spanning Trees *(MST)*

A **subgraph** is a subset of a graph's edges *(and associated vertices)* that constitutes a graph. Many computational tasks involve identifying subgraphs of various types. If we identify a subset of a graph's vertices, we call that subset, together with all edges that connect two of its members, the **induced subgraph** associated with those vertices. Per the above example, we can draw a graph by marking points for the vertices and drawing lines connecting them for the edges. A drawing gives us intuition about the structure of a graph; but it's also true to say that this intuition can be misleading, because the graph is defined independently of its representation.

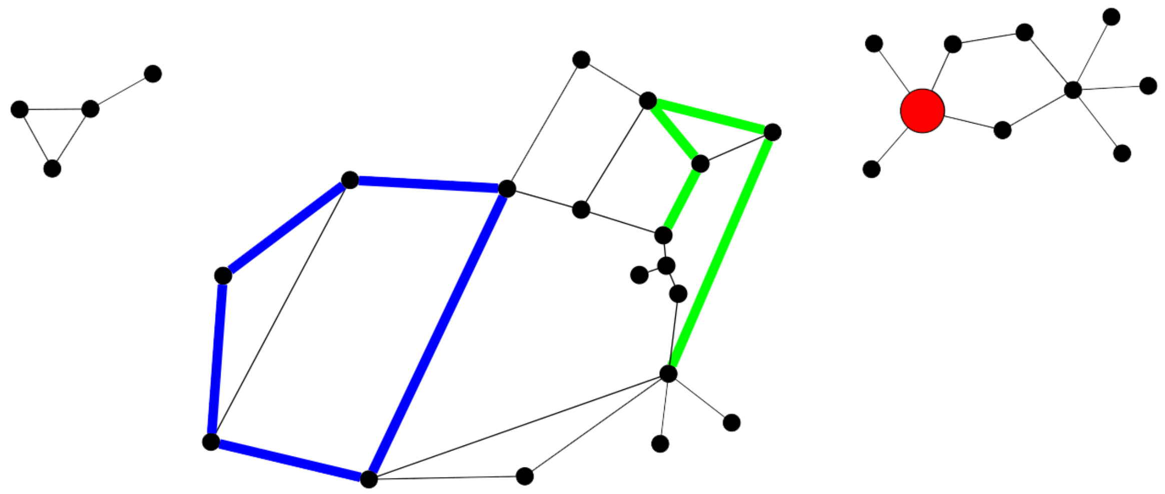
Here is an example of why this is. The two graph drawings shown here, and the list of edges represents the same graph because the graph is only its *(unordered)* set of vertices and its *(unordered)* set of edges *(pairs of vertices)*—nothing more. A **path** in a graph is a sequence of vertices in which each successive vertex *(after the first)* is adjacent to its predecessor in the path. In a **simple path**, the vertices and edges are distinct. A **cycle** is a path that is simple except that the first and final vertices are the same.



0-5	5-4	7-8
4-3	0-2	9-12
0-1	11-12	5-3
9-12	9-10	
6-4	0-6	

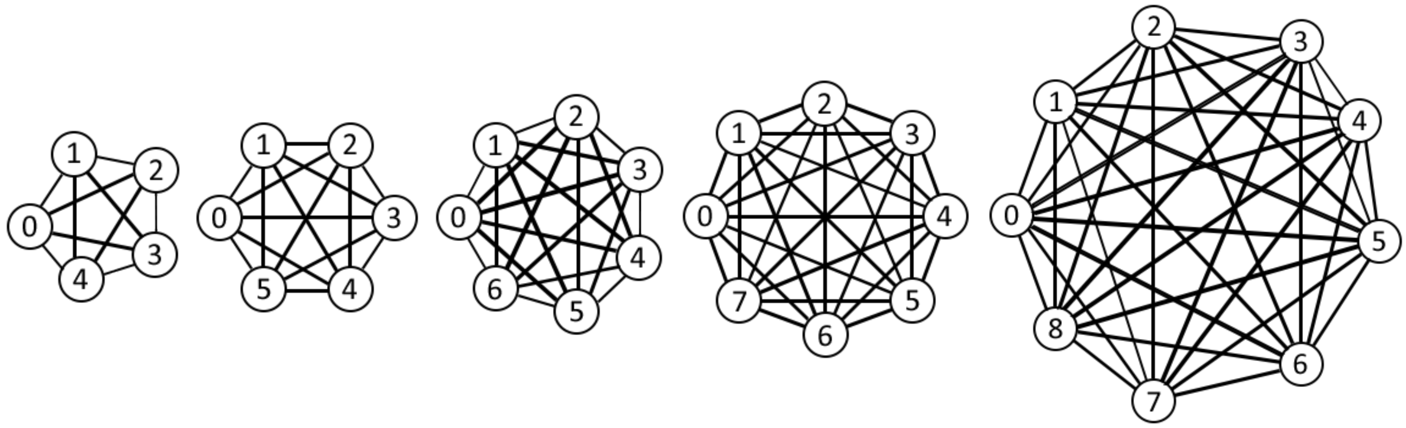
The possible vertex placements, edge-drawing styles, and aesthetic constraints on the drawing are boundless. A **planar graph** is one that can be drawn in the plane without any edges crossing. An acyclic connected graph is called a **tree**. A set of trees is called a **forest**. A **spanning tree** of a connected graph is a subgraph that contains all of that graph's vertices and is a single tree. A **spanning forest** of a graph is a subgraph that contains all of that graph's vertices and is a forest.

Graph Terminolgy

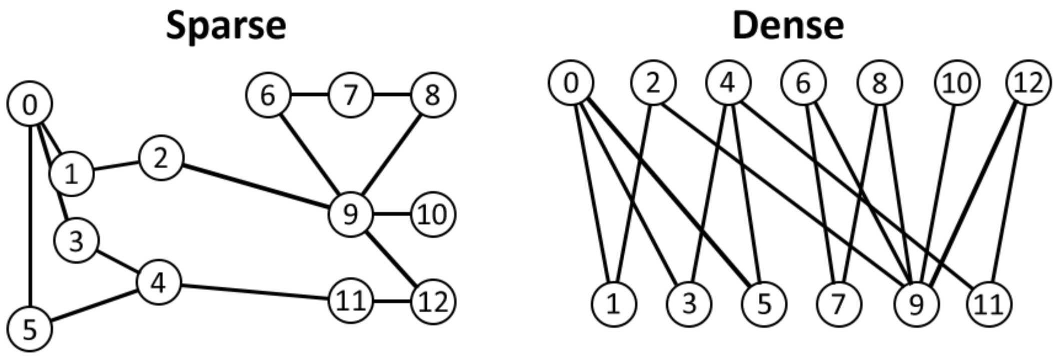


vertex: black dot; edge: line; path: green lines (len=4); cycle: blue lines (len=5); 3 connected components; red vertex has degree 4

However, it suffices to consider a graph simply as a set of edges, we see other representations that are particularly suitable as the basis for graph data structures here:



These five above graphs with all edges present are known as **complete graphs**. We define the counterpart of a graph G by starting with a complete graph that has the same set of vertices as the original graph and then removing the edges of G. The union of two graphs is the graph induced by the union of their sets and edges. The union of a graph and its counterpart is a complete graph. The above illustration demonstrates with every vertex connected to every other vertex, have 10, 15, 21, 28, and 36 edges *(left to right)*. Every graph with between 5 and 9 vertices *(there are more than 68 billion possible graphs)* is a subgraph of one of these graphs. Placing the vertices of a given graph on the plane and drawing them and the edges that connect them is known as **graph drawing**. All graphs that have V vertices are subgraphs of the complete graph that has V vertices. The total number of different graphs that have V vertices is $2^{V(V-1)/2}$ *(the number of different ways to choose a subset from the $V(V-1)/2$ possible edges)*. A complete graph is called a **clique**. **Sparse** and **dense** graphs are two fundamental subjects that describe the density of edges in a graph data structure.

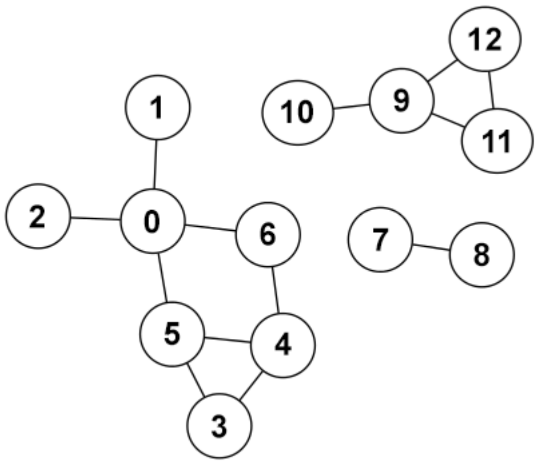


The primary difference between them results from the number of edges they contain relative to the number of vertices. **Understanding this difference is essential because it can influence the choice of data structures and algorithms used to represent and process these graphs proficiently.** Knowing whether a graph is sparse or dense is generally a key factor in selecting an efficient algorithm to process the graph. For example, we might develop one algorithm that takes V^2 steps and another that takes about $E \lg E$ steps. These formulas tell us that the second algorithm would be better for sparse graphs, whereas the first would be preferred for dense graphs. The two above examples of sparse and dense graphs are also referred to as **Bipartite graphs**. This means all edges in this graph connect odd-numbered vertices with even-numbered ones, so therefore it is bipartite. The graph on the right makes the property evident. These formulas tell us the second algorithm would be better for sparse graphs, whereas the first would be preferred for dense graphs. When analyzing graph algorithms, assume that V/E is bounded above by a small constant, so it's reduced expressions such as $V(V+E)$ to VE. As a part of graph representation we can implement the API.

GraphUndirected	
GraphUndirected(V : int)	// create an empty graph with V vertices
addEdge(v : int, w : int)	// add an edge v-w
adj(v : int) : Iterable<Integer>	// vertices adjacent to v
V() : int	// number of vertices
E() : int	// number of edges

```
// degree of vertex v in Graph G
public static int degree(Graph G, int V) {
    int degree = 0;
    for(int w : G.adj(v)) degree++;
    return degree;
}
```

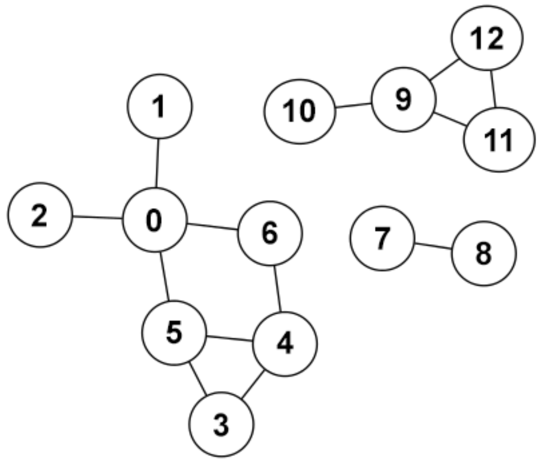
The following is a graph representation of an edge list. Maintain a list of edges as pairs of vertices.



```
edges = [(0,1), (0,2), (0,5),
         (0,6), (3,4), (3,5),
         (4,5), (4,6), (7,8),
         (9,10), (9,11),
         (9,12), (11,12)]
```

Note: 0, 1 implies 1, 0 is also an edge.

This graph example is how you represent an adjacency matrix by maintaining a two-dimensional V by V boolean array; for each edge $v - w$ in graph: adj [v] [w] = adj [w] [v] = true.



```
     0 1 2 3 4 5 6 7 8 9 10 11 12
 0:  . X X . . X X . . . .  .  .
 1:  X . . . . . . . . . .  .  .
 2:  X . . . . . . . . . .  .  .
 3:  . . . . X X . . . . .  .  .
 4:  . . . X . X X . . . .  .  .
 5:  X . . X X . . . . . .  .  .
 6:  X . . . X . . . . . .  .  .
 7:  . . . . . . . . X . .  .  .
 8:  . . . . . . . X . . .  .  .
 9:  . . . . . . . . . . X  X  X
10:  . . . . . . . . . X .  .  .
11:  . . . . . . . . . X .  .  X
12:  . . . . . . . . . X .  X  .
```

There are two entries for each edge and we can implement a DFS as follows:

```java
public class Graph {
    private final int V;
    private Bag<Integer>[] adj;     // adj lists

    public Graph(int V) {
        // create empty graph with V vertices
        this.V = V;
        adj = (Bag<Integer>[]) new Bag[V];
        for(int v = 0; v < V; v++)
            adj[v] = new Bag<Integer>();
    }

    public void addEdge(int v, int w) {
        // add edge v-w (parallel edges and self-loops allowed)
        adj[v].add(w);
        adj[w].add(v);
    }

    // iterator for vertices adjacent to v
    public Iterable<Integer> adj(int v) {
        return adj[v];
    }
}
```

Conclusion

This concludes my technical guide on Data Structures & Algorithms with Java. At nothing else, my hope is that this guide provided a detailed glimpse into the theoretical and conceptual methodologies of writing programs for software development in the Java programming language. I will say, after completing COS 265 - *Data Structures & Algorithms* at Taylor University, I couldn't be happier to get it behind me, but I'm also glad that my department required it for my unique computer science degree. There was a tremendous amount of work involved in that course and you can only really appreciate that type of discipline when you're facing the hurdle head on. Although, as mentioned in my previous guides, the information I provided was only scraping the surface, and the same is said for this guide as well. This guide also served as a 'reiteration-to-memory' for myself of what I had just completed, although extremely watered down with my notes, and examples from several labs. Obviously, the subject of Data Structures & Algorithms covers a considerable amount of more information and tasks within one semester of work. But for the growing number of people who question having to take this kind of course, or maybe are not sure of the variance in content with this kind of course, hopefully this guide provided the insight you were perhaps looking for. All diagrams, code samples, and notes were created and provided by me, pulled from my course and lab notes and converted digitally for the purpose of this guide. If you have any questions about this guide or any other general inquiries, you can email me at technologicguy@gmail.com

Resources Used:

- Drozdek, Adam. *Data Structures & Algorithms in Java* – 2001
- Kleinberg, Jon. *Algorithm Design* – 2005
- Oracle.com