Ghi chú của một coder

Vũ Anh

Tháng 01 năm 2018

Mục lục

Phần I

Khoa học máy tính

Chương 1

Data Structure and Algorithm

View online http://magizbox.com/training/danda/site/

1.1 Introduction

Algorithms + Data Structures = Programs
In computer science, a data structure is a particular way of organizing data in a computer so that it can be used efficiently. Data structures can implement one or more particular abstract data types (ADT), which specify the operations that can be performed on a data structure and the computional complexity of those operations. In comparison, a data structure is a concrete implementation of the specification provided by an ADT.
In mathematics and computer science, an algorithm is a self-contained step-by-step set of operations to be performed. Algorithms perform calculation, data processing, and/or automated reasoning tasks.
Software engineering is the study of ways in which to create large and complex computer applications and that generally involve many programmers and designers. At the heart of software engineering is with the overall design of the applications and on the creation of a design that is based on the needs and requirements of end users. While software engineering involves the full life cycle of a software project, is includes many different components - specification, requirements gathering, design, verification, coding, testing, quality assurance, user acceptance testing, production, and ongoing maintenance.
Having an in-depth understanding on every component of software engineering is not mandatory, however, it is important to understand that the subject of data structures and algorithms is concerned with the coding phase. The use of data structures and algorithms is the nuts-and-blots used by programmers to store and manipulate data.
This article, along with the other examples in this section focuses on the essentials of data structures and algorithms. Attempts will be made to understand how they work, which structure or algorithm is best in a particular situation in an easy to understand environment.

Data Structures and Algorithms - Defined A data structure is an arrangement of data in a computer's memory or even disk storage. An example of several common data structures are arrays, linked lists, queues, stacks, binary trees, and hash tables. Algorithms, on the other hand, are used to manipulate the data contained in these data structures as in searching and sorting.
Many algorithms apply directly to a specific data structures. When working with certain data structures you need to know how to insert new data, search for a specified item, and deleting a specific item.
Commonly used algorithms include are useful for:
Searching for a particular data item (or record). Sorting the data. There are many ways to sort data. Simple sorting, Advanced sorting Iterating through all the items in a data structure. (Visiting each item in turn so as to display it or perform some other action on these items)

1.1.1 Greedy Algorithm

Greedy Algorithms An algorithm is designed to achieve optimum solution for a given problem. In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen.
Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally greedy algorithms do not provide globally optimized solutions.
Counting Coins This problem is to count to a desired value by choosing the least possible coins and the greedy approach forces the algorithm to pick the largest possible coin. If we are provided coins of 1, 2, 5 and 10 and we are asked to count 18 then the greedy procedure will be
Select one 10 coin, the remaining count is 8
Then select one 5 coin, the remaining count is 3
Then select one 2 coin, the remaining count is 1
And finally, the selection of one 1 coins solves the problem
Though, it seems to be working fine, for this count we need to pick only 4 coins. But if we slightly change the problem then the same approach may not be able to produce the same optimum result.
For the currency system, where we have coins of 1, 7, 10 value, counting coins for value 18 will be absolutely optimum but for count like 15, it may use more coins than necessary. For example, the greedy approach will use 10 + 1 + 1 + 1 + 1 + 1, total 6 coins. Whereas the same problem could be solved by using only 3 coins (7 + 7 + 1)
Hence, we may conclude that the greedy approach picks an immediate optimized solution and may fail where global optimization is a major concern.
Examples Most networking algorithms use the greedy approach. Here is a list of few of them
Travelling Salesman Problem Prim's Minimal Spanning Tree Algorithm Kruskal's Minimal Spanning Tree Algorithm Dijkstra's Minimal Spanning Tree Algorithm Graph - Map Coloring Graph - Vertex Cover Knapsack Problem Job Scheduling Problem

1.1.2 Divide and Conquer

In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the subproblems into even smaller sub-problems, we may eventually reach a stage where no more division is possible. Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.
Broadly, we can understand divide-and-conquer approach in a three-step process.
Divide/Break This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in nature but still represent some part of the actual problem.
Conquer/Solve This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.
Merge/Combine When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively and conquer merge steps works so close that they appear as one.
Examples The following computer algorithms are based on divide-and-conquer programming approach
Merge Sort Quick Sort Binary Search Strassen's Matrix Multiplication Closest pair (points) There are various ways available to solve any computer problem, but the mentioned are a good example of divide and conquer approach.

1.1.3 Dynamic Programming

Dynamic programming approach is similar to divide and conquer in breaking down the problem into smaller and yet smaller possible sub-problems. But unlike, divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.
Dynamic programming is used where we have problems, which can be divided into similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best solution.
So we can say that
The problem should be able to be divided into smaller overlapping sub-problem. An optimum solution can be achieved by using an optimum solution of smaller sub-problems. Dynamic algorithms use memorization. Comparison In contrast to greedy algorithms, where local optimization is addressed, dynamic algorithms are motivated for an overall optimization of the problem.
In contrast to divide and conquer algorithms, where solutions are combined to achieve an overall solution, dynamic algorithms use the output of a smaller sub-problem and then try to optimize a bigger sub-problem. Dynamic algorithms use memorization to remember the output of already solved sub-problems.

Example The following computer problems can be solved using dynamic programming approach
Fibonacci number series Knapsack problem Tower of Hanoi All pair shortest path by Floyd-Warshall Shortest path by Dijkstra Project scheduling Dynamic programming can be used in both top-down and bottom-up manner. And of course, most of the times, referring to the previous solution output is cheaper than recomputing in terms of CPU cycles.

1.1.4 7 Steps to Solve Algorithm Problems

Today, I viewed the video "7 Steps to Solve Algorithm Problems" by Gayle Laakmann McDowell - the author of Cracking the Coding Interview book. In this video, Gayle describe her method for solve algorithms problems which consists 7 steps: listen carefully, example, brute force, optimize, walk through your algorithms, code and test. In this article, I will summary these steps base on what I learned from this video.
Step 1: Listen carefully Every single detail in a question is necessary to solve it.
The first step is to listen carefully to the problem. So, generally speaking every single detail in a question is necessary to solve that problem - either to solve it all or to solve it optimally. So if there's some detail you haven't used in the question in your algorithm so far think about how you can put that to use because it might be necessary to solve the problem optimally.
Let me give you an example.
You have two arrays, sorted and distinct How did you find the number of elements in common between the two arrays? A lot of people solve this problem and they'll get kind of stuck for awhile and what they'll do is they'll be solving the problem and they'll know the arrays are sorted but they haven't actually used the fact that it's sorted.
This sorting detail - it's not necessary just to find an algorithm but it is necessary to solve the problem optimally.
So remember every single detail in the problem and make sure you use it.
Step 2: Example Make example big, no special cases
The second piece is to come up with a good example, so the last problem that I gave two arrays sorted and distinct compute the number of elements in common, most people's examples look like this.
too small and special case A: 1, 5, 15, 20 B: 2, 5, 13, 30 Yes technically if it's a problem but it's not very useful.
As soon as you glance at this example you notice that there's only one element common and you know exactly what it is and it's obvious because this example is so small and it's actually kind of a special case.
A better example is something like this
larger and avoid special cases A: 1, 5, 15, 20, 30, 37 B: 2, 5, 13, 30, 32, 35, 37, 42 It's much larger and you've avoided some special cases. One of the easiest ways of improving your performance on algorithm questions is just make your examples larger and really avoid special cases.
Step 3: Brute force Better to have a brute force than nothing at all
The third step is to come up with a brute force algorithm. Now I'm not saying you need to go out of your way to come up with something slow, I'm really just saying, hey if the first thing you have is only something really really slow and terrible that's okay. It is so much better to start off with something slow then

to start off with nothing at all. So it's fine if your first algorithm is slow and terrible whatever. However, and this is very very very important, I'm not saying to code the brute force. I'm saying just state your brute force algorithm, state its runtime, and then immediately go to optimizing.
A good chunk of the time on algorithm interview question will often be spent on optimizations. So that's step 4 and spend some good time on it.
Step 4: Optimize The fourth step is optimize and spend some good time on it.
Step 5: Walk through your algorithms Know exactly what you're going to do before coding
what variables data structures? how, why, why do they change? what is the structure of your code Then once you have an optimal algorithm or you're ready to start coding take a step back and just make sure you know exactly what you're going to do in your code.
So many people code prematurely when they aren't really really comfortable with what they're about to do and it ends in disaster. An eighty percent understanding of what you're about to write is really not enough for a whiteboard especially. So take a moment and walk through your algorithm and make sure you know exactly what you're about to do.
Step 6: Code Use space wisely, coding style matters, modularize
Step 6 is to start coding and I'm gonna go into this in a bit of detail. So a couple things to keep in mind particularly when you're coding on a whiteboard. The first couple tips are kind of whiteboard specific but try to write your lines straight. I'm not gonna be judging you on your handwriting and things like that but when people start writing their lines and sharp angles they start to lose track over whether this if statement under this for loop or not. The second thing is use your board space wisely. If you don't need stuff up on the board anymore just erase it. Try to write in this top left corner etc.
Basically give yourself as much space as you possibly can to write your code. If you do run out of space though, it's ok to use arrows, that's fine, I'm really not gonna be judging you on this kind of stuff. So more important things.
Coding style matters (consistent braces, consistent variable naming, consistence spaces, descriptive variables)
Coding style matters even on a whiteboard but on a computer as well, so that means things like braces, naming conventions, or using camel case or underscores, things like that. Those kind of style things absolutely matter. I'm not that concerned over which style you pick, I don't care if you write braces on the same line or the next line but I do care a lot that you have a style and you stick to it. So be consistent in your style. When it comes to variable names, yeah I know it's an annoying to write long variable names on a whiteboard but descriptive variable names are important to good style. So one compromise here is write the good descriptive variable name first and then just ask your interviewer, hey is it okay if I abbreviate this the next time. So that'll be a nice little compromise - you'd show that you care about good variable names but you also don't waste a lot of time.
Modularize (before. not after)
Last thing I want to talk about is modularization. Modularize your code up front and just any little conceptual chunks of code, push that off to another function. So suppose you have three steps in your algorithm - process the first string, process the second string, and then compare the results. Don't start writing these for loops that walk through each string in the very beginning.

Instead write this overall function that wraps these three steps. So step one, step two, step three, and then start drilling in and going into all the details there. Remember any conceptual chunks of code push those off to other functions, don't write them in line.
Step 7: Test Analyse: think about each line, double check things that look weired/risky (for-loop that decrement, math)
Use test cases (smaller test-cases first (faster to run, you will problably be more through, edge cases, big test cases)
Then once you're done with the coding you have to start testing your code. One of the mistakes a lot of people do here is they take their big example from step 2 and throw that in as a test case. The problem with that is it's very large so it will take you a long time to run through but also you just used that to develop your code, so if here's an oversight there, the problem will probably repeat itself here.
What's a better step to do, what's a better process to do, is just walk through your code line by line and just think about each line up front not with the test case but just consider, is it really doing the right thing?
Double check anything that looks weird, so for loops that decrement instead of increment and any math at all is a really common place for errors. Just think, look at your code analytically and think what are the most likely places for errors to be and double-check those.
Start with small rather than big
Then once you start with actual test cases start with small test cases rather than big ones. Small test cases work pretty much as effectively as big test cases but they are so much faster to run through, and in fact because they're faster people tend to be much more thorough so you're much more likely to actually find bugs with small test cases than big test cases. So start with small test cases then go in to edge cases after that and then if you have time maybe throw in some big test cases. A couple last techniques with testing. The first one is make sure that when you're testing you're really thinking about what you're doing. A lot of people when they're testing they're just walking through their code almost like they're a bot, and they only look to see if things made sense at the very end when they look at their output. It's much better to really think as you're testing, this way you find the bug as soon as it happens rather than six lines later at the very bottom.
Test your code not your algorithm
The second thing is when you're testing make sure that you're actually testing your code and not your algorithm. An amazing number of people will just take their example and like just walk through it again as though they're just walking through their algorithm but they're never even looking at their code, they're not looking at the exact calculations their code actually did. So make sure that you're really testing your code.
Find bugs
Then the last thing is when you find in a bug, don't panic. Just really think about what caused the bug. A lot of times people will panic and just try to make the first fix that fixes it for that output but they haven't really given it some thought and then they're in a much worse position because if you make the wrong fix to your code, the thing that just fixed the output but didn't fix a real bug you've not fixed the actual bug, you've made your code more complex, and you potentially introduced a brand new bug and you're in a much worse

position. It's much better to just when you find the bug, it's ok, it's not that big of a deal to have a bug it's very normal just really think through what the actual bug, where the actual plug came from.
Remember
think as you test (don't be a bot) test your code, not your algorithm think before you fix bugs. Don't panic! (wrong fixes are worse than no fix) Suggested Reading 7 Steps to Solve Algorithm Problems. Gayle Laakmann McDowell

1.2 Data Structures

1.2.1 Array

Arrays An array is an aggregate data structure that is designed to store a group of objects of the same or different types. Arrays can hold primitives as well as references. The array is the most efficient data structure for storing and accessing a sequence of objects.
Here is the list of most important array features you must know (i.e. be able to program)
copying and cloning insertion and deletion searching and sorting You already know that the Java language has only two data types, primitives and references. Which one is an array? Is it primitive? An array is not a primitive data type - it has a field (and only one), called length. Formally speaking, an array is a reference type, though you cannot find such a class in the Java APIs. Therefore, you deal with arrays as you deal with references. One of the major diffeences between refeences and primituives is that you cannot copy arrays by assigning one to another:
int[] a = 9, 5, 4; int[] b = a; The assignment operator creates an alias to the object, like in the picture below
Since these two references a and b refer to the same object, comparing them with the double equal sign "==" will always return true. In the next code example, int [] a = 1,2,3; int [] b = 1,2,3; a and b refer to two different objects (though with identical contents). Comparing them with the double equal sign will return false. How would you compare two objects with identical contents? In short, using the equals method. For array comparison, the Java APIs provides the Arrays class. The Arrays class The java.util.Arrays class is a convenience class for various array manipulations, like comparison, searching, printing, sorting and others. Basically, this class is a set of static methods that are all useful for working with arrays. The code below demonstrates a proper invocation of equals:
int[] a = 1,2,3; int[] b = 1,2,3; if(Arrays.equals(a, b)) System.out.println("arrays with identical contents"); Another commonly used method is toString() which takes care of of printing
int[] a = 1,2,3; System.out.println(Arrays.toString(a)); Here is the example of sorting
int[] a = 3,2,1; Arrays.sort(a); System.out.println(Arrays.toString(a)); In addition to that, the class has other utility methods for supporting operations over multidimensional arrays.
Copying arrays There are four ways to copy arrays
using a loop structure using Arrays.copyOf() using System.arraycopy() using clone() The first way is very well known to you

int[] a = 1, 2, 3; int[] b = new int[a.length]; for(int i= 0; i < a.length; i++) b[i] = a[i]; The next choice is to use Arrays.copyOf()
int[] a = 1, 2, 3; int[] b = Arrays.copyOf(a, a.length); The second parameter specifies the length of the new array, which could either less or equal or bigger than the original length.
The most efficient copying data between arrays is provided by System.arraycopy() method. The method requires five arguments. Here is its signature
public static void arraycopy(Object source, int srcIndex, Object destination, int destIndex, int length) The method copies length elements from a source array starting with the index srcIndex to a new array destination at the index destIndex.The above code example can be rewritten as it follows
int[] a = 1, 2, 3; int[] b = new int[a.length]; System.arraycopy(a, 0, b, 0, 3) And the last copying choice is the use of cloning. Cloning involves creating a new array of the same size and type and copying all the old elements into the new array. The clone() method is defined in the Object class and its invocation is demonstrated by this code segment
int[] a = 1, 2, 3; int[] b = (int[]) a.clone(); Note, that casting (int[]) is the must. Examine the code in ArrayCopyPrimitives.java for further details.
Insertion and Deletion Arrays in Java have no methods and only one immutable field length. Once an array is created, its length is fixed and cannot be changed. What do you do to resize the array? You allocate the array with a different size and copy the contents of the old array to the new array. This code example demonstrates deletion from an array of primitives
public char[] delete(char[] data, int pos) if(pos >= 0 pos < data.length) char[] tmp = new char[data.length-1]; System.arraycopy(data, 0, tmp, 0, pos); System.arraycopy(data, pos+1, tmp, pos, data.length-pos-1); return tmp; else return data; The first arraycopy copies the elements from index 0 to index pos-1, the second arraycopy copies the elements from index pos+1 to data.length. Examine the code in ArrayDemo.java for further details.
The ArrayList class The java.util.ArrayList class supports an idea of a dynamic array - an array that grows and shrinks on demand to accomodate the number of elements in the array. Below is a list of commonly used methods
add(object) - adds to the end add(index, object) - inserts at the index set(index, object) - replaces at the index get(index) - returns the element at that index remove(index) - deletes the element at that index size() - returns the number of elements The following code example will give you a heads up into how some of them are used.
/* ADD */ ArrayList<Integer> num = new ArrayList<Integer>(); for(int i = 0; i < 10; i++) num.add(i); System.out.println(num);
/* REMOVE even integers */ for(int i = 0; i < num.size(); i++) if(num.get(i)System.out.println(num);
Copying arrays of objects This topic is more complex for understanding.. Let us start with a simple loop structure
Object[] obj1 = new Integer(10), new StringBuffer("foobar"), new Double(12.95); Object[] obj2 = new Object[obj1.length]; for(int i = 0; i ‹ obj1.length; i++) obj2[i] = obj1[i]; At the first glance we might think that all data is copied. In reality, the internal data is shared between two arrays. The figure below illustrates the inner structure
The assignment operator obj2[i] = obj1[i] is a crucial part of understanding the concept. You cannot copy references by assigning one to another. The assignment creates an alias rather than a copy. Let us trace down changes in the above

picture after execution the following statements
obj1[0] = new Integer(5);
and ((StringBuffer) obj1[1]).append('s');
As you see, obj1[0] and obj2[0] now refer to different objects. However, obj1[1] and obj2[1] refer to the same object (which is "foobars"). Since both arrays shares the data, you must be quite careful when you modify your data, because it might lead to unexpected effects.
The same behavior will take place again, if we use Arrays.copuyOf(), System.arraycopy() and clone(). Examine the code example ArrayCopyReferences.java for further details.
Multi-dimensional arrays In many practical application there is a need to use two- or multi-dimensional arrays. A two-dimensional array can be thought of as a table of rows and columns. This creates a table of 2 rows and 4 columns:
int[][] ar1 = new int[2][4]; You can create and initialize an array by using nested curcly braces. For example, this creates a table of 3 rows and 2 columns:
int[][] ar2 = 1,2,3,4,5,6; Generally speaking, a two-dimensional array is not exactly a table - each row in such array can have a different length. Consider this code fragment
Object[][] obj = new Integer(1),new Integer(2), new Integer(10), "bozo", new Double(1.95); The accompanying picture sheds a bit of light on internal representation
From the picture you clearly see that a two-dimensional array in Java is an array of arrays. The array obj has two elements obj[0] and obj[1] that are arrays of length 2 and 3 respectively.
Cloning 2D arrays The procedure is even more confusing and less expected. Consider the following code segment
Object[][] obj = new Integer(1),new Integer(2), new Integer(10), "bozo", new Double(1.95);
Object[][] twin = (Object[][]) obj.clone(); The procedure of clonig 2d arrays is virtually the same as cloning an array of references. Unfortunately, built-in clone() method does not actualy clone each row, but rather creates references to them Here is a graphical interpretation of the above code
Let us change the value of obj[1][1]
obj[1][1] = "xyz"; This assignment effects the value of twin[1][1] as well
Such a copy is called a "shallow" copy. The default behavior of clone() is to return a shallow copy of the object. If we want a "deep" copy instead, we must provide our own implementation by overriding Object's clone() method.
The idea of a "deep" copy is simple - it makes a distinct copy of each of the object's fields, recursing through the entire object. A deep copy is thus a completely separate object from the original; changes to it don't affect the original, and vise versa. In relevance to the above code, here is a deep clone graphically
Further, making a complete deep copy is not always needed. Consider an array of immutable objects. As we know, immutable objects cannot be modified, allowing clients to share the same instance without interfering with each other. In this case there is no need to clone them, which leads to the following picture
Always in this course we will create data structures of immutable objets, therefore implementing the clone method will require copying a structure (a shape) and sharing its internal data. We will discuss these issues later on in the course.
Challenges "Arrays: Left Rotation". hackerrank. 2016 References "Array Data Structure". Victor S.Adamchik, CMU. 2009

1.2.2 Linked List

A linked list is a sequence of data structures, which are connected together via links.
Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.
Link Each link of a linked list can store a data called an element. Next Each link of a linked list contains a link to the next link called Next. LinkedList A Linked List contains the connection link to the first link called First. Representation Linked list can be visualized as a chain of nodes, where every node points to the next node.
As per the above illustration, following are the important points to be considered.
Linked List contains a link element called first. Each link carries a data field(s) and a link field called next. Each link is linked with its next link using its next link. Last link carries a link as null to mark the end of the list. Types of Linked List Following are the various types of linked list.
Simple Linked List Item navigation is forward only. Doubly Linked List Items can be navigated forward and backward. Circular Linked List Last item contains link of the first element as next and the first element has a link to the last element as previous. Basic Operations Following are the basic operations supported by a list.
Insertion Adds an element at the beginning of the list. Deletion Deletes an element at the beginning of the list. Display Displays the complete list. Search Searches an element using the given key. Delete Deletes an element using the given key. Insertion Operation Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.
Imagine that we are inserting a node B (NewNode), between A (LeftNode) and C (RightNode). Then point B.next to C
NewNode.next > RightNode; It should look like this
Now, the next node at the left should point to the new node.
LeftNode.next > NewNode;
This will put the new node in the middle of the two. The new list should look like this
Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.
Deletion Operation Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.
The left (previous) node of the target node now should point to the next node of the target node
LeftNode.next > TargetNode.next;
This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.
TargetNode.next > NULL;

We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.
Reverse Operation This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node
We have to make sure that the last node is not the lost node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.
Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.
We'll make the head node point to the new first node by using the temp node. The linked list is now reversed.

1.2.3 Stack and Queue

An array is a random access data structure, where each element can be accessed directly and in constant time. A typical illustration of random access is a book - each page of the book can be open independently of others. Random access is critical to many algorithms, for example binary search.
A linked list is a sequential access data structure, where each element can be accesed only in particular order. A typical illustration of sequential access is a roll of paper or tape - all prior material must be unrolled in order to get to data you want.
In this note we consider a subcase of sequential data structures, so-called limited access data sturctures.
Stacks A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle. In the pushdown stacks only two operations are allowed: push the item into the stack, and pop the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. push adds an item to the top of the stack, pop removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top. A stack is a recursive data structure. Here is a structural definition of a Stack:
a stack is either empty or it consistes of a top and the rest which is a stack;
Applications The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack. Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack. Backtracking. This is a process when you need to access the most recent data element in a series of elements. Think of a labyrinth or maze - how do you find a way from an entrance to an exit? Once you reach a dead end, you must backtrack. But backtrack to where? to the previous choice point. Therefore, at each choice point you store on a stack all possible choices. Then backtracking simply means popping a next choice from the stack.
Language processing: space for parameters and local variables is created internally using a stack. compiler's syntax check for matching braces is implemented by using stack. support for recursion Implementation In the standard library of

classes, the data type stack is an adapter class, meaning that a stack is built on top of other data structures. The underlying structure for a stack could be an array, a vector, an ArrayList, a linked list, or any other collection. Regardless of the type of the underlying data structure, a Stack must implement the same functionality. This is achieved by providing a unique interface:
public interface StackInterface<AnyType> public void push(AnyType e);
public AnyType pop();
public AnyType peek();
public boolean isEmpty(); The following picture demonstrates the idea of implementation by composition.
Another implementation requirement (in addition to the above interface) is that all stack operations must run in constant time O(1). Constant time means that there is some constant k such that an operation takes k nanoseconds of computational time regardless of the stack size.
Array-based implementation
In an array-based implementation we maintain the following fields: an array A of a default size (1), the variable top that refers to the top element in the stack and the capacity that refers to the array size. The variable top changes from -1 to capacity - 1. We say that a stack is empty when top = -1, and the stack is full when top = capacity-1. In a fixed-size stack abstraction, the capacity stays unchanged, therefore when top reaches capacity, the stack object throws an exception. See ArrayStack.java for a complete implementation of the stack class.
In a dynamic stack abstraction when top reaches capacity, we double up the stack siz
Linked List-based implementation
Linked List-based implementation provides the best (from the efficiency point of view) dynamic stack implementation. See ListStack.java for a complete implementation of the stack class.
Queues A queue is a container of objects (a linear collection) that are inserted and removed according to the first-in first-out (FIFO) principle. An excellent example of a queue is a line of students in the food court of the UC. New additions to a line made to the back of the queue, while removal (or serving) happens in the front. In the queue only two operations are allowed enqueue and dequeue. Enqueue means to insert an item into the back of the queue, dequeue means removing the front item. The picture demonstrates the FIFO access. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.
Implementation In the standard library of classes, the data type queue is an adapter class, meaning that a queue is built on top of other data structures. The underlying structure for a queue could be an array, a Vector, an ArrayList, a LinkedList, or any other collection. Regardless of the type of the underlying data structure, a queue must implement the same functionality. This is achieved by providing a unique interface.
interface QueueInterface‹AnyType> public boolean isEmpty();
public AnyType getFront();
public AnyType dequeue();
public void enqueue(AnyType e);

public void clear(); Each of the above basic operations must run at constant time O(1). The following picture demonstrates the idea of implementation by composition.
Circular Queue Given an array A of a default size (1) with two references back and front, originally set to -1 and 0 respectively. Each time we insert (enqueue) a new item, we increase the back index; when we remove (dequeue) an item - we increase the front index. Here is a picture that illustrates the model after a few steps:
As you see from the picture, the queue logically moves in the array from left to right. After several moves back reaches the end, leaving no space for adding new elements
However, there is a free space before the front index. We shall use that space for enqueueing new items, i.e. the next entry will be stored at index 0, then 1, until front. Such a model is called a wrap around queue or a circular queue
Finally, when back reaches front, the queue is full. There are two choices to handle a full queue:a) throw an exception; b) double the array size.
The circular queue implementation is done by using the modulo operator (denoted
See ArrayQueue.java for a complete implementation of a circular queue.
Applications The simplest two search techniques are known as Depth-First Search(DFS) and Breadth-First Search (BFS). These two searches are described by looking at how the search tree (representing all the possible paths from the start) will be traversed.
Deapth-First Search with a Stack In depth-first search we go down a path until we get to a dead end; then we backtrack or back up (by popping a stack) to get an alternative path.
Create a stack Create a new choice point Push the choice point onto the stack while (not found and stack is not empty) Pop the stack Find all possible choices after the last one tried Push these choices onto the stack Return Breadth-First Search with a Queue In breadth-first search we explore all the nearest possibilities by finding all possible successors and enqueue them to a queue.
Create a queue Create a new choice point Enqueue the choice point onto the queue while (not found and queue is not empty) Dequeue the queue Find all possible choices after the last one tried Enqueue these choices onto the queue Return We will see more on search techniques later in the course.
Arithmetic Expression Evaluation An important application of stacks is in parsing. For example, a compiler must parse arithmetic expressions written using infix notation:
1 + ((2 + 3) * 4 + 5)*6 We break the problem of parsing infix expressions into two stages. First, we convert from infix to a different representation called postfix. Then we parse the postfix expression, which is a somewhat easier problem than directly parsing infix.
Converting from Infix to Postfix. Typically, we deal with expressions in infix notation
2 + 5 where the operators (e.g. +, *) are written between the operands (e.q, 2 and 5). Writing the operators after the operands gives a postfix expression 2 and 5 are called operands, and the '+' is operator. The above arithmetic expression is called infix, since the operator is in between operands. The expression
2 5 + Writing the operators before the operands gives a prefix expression

+2 5 Suppose you want to compute the cost of your shopping trip. To do so, you add a list of numbers and multiply them by the local sales tax (7.25
70 + 150 * 1.0725 Depending on the calculator, the answer would be either 235.95 or 230.875. To avoid this confusion we shall use a postfix notation
70 150 + 1.0725 * Postfix has the nice property that parentheses are unnecessary. Now, we describe how to convert from infix to postfix.
Read in the tokens one at a time If a token is an integer, write it into the output If a token is an operator, push it to the stack, if the stack is empty. If the stack is not empty, you pop entries with higher or equal priority and only then you push that 1. token to the stack. If a token is a left parentheses '(', push it to the stack If a token is a right parentheses ')', you pop entries until you meet '('. When you finish reading the string, you pop up all tokens which are left there. Arithmetic precedence is in increasing order: '+', '-', '*', '/'; Example. Suppose we have an infix expression:2+(4+3*2+1)/3. We read the string by characters. '2' - send to the output. '+' - push on the stack. '(' - push on the stack. '4' - send to the output. '+' - push on the stack. '3' - send to the output. '*' - push on the stack. '2' - send to the output. Evaluating a Postfix Expression. We describe how to parse and evaluate a postfix expression.
We read the tokens in one at a time. If it is an integer, push it on the stack If it is a binary operator, pop the top two elements from the stack, apply the operator, and push the result back on the stack. Consider the following postfix expression
5 9 3 + 4 2 * * 7 + * Here is a chain of operations
Stack Operations Output ———————————— push(5); 5 push(9); 5 9 push(3); 5 9 3 push(pop() + pop()) 5 12 push(4); 5 12 4 push(2); 5 12 4 2 push(pop() * pop()) 5 12 8 push(pop() * pop()) 5 96 push(7) 5 96 7 push(pop() + pop()) 5 103 push(pop() * pop()) 515 Note, that division is not a commutative operation, so 2/3 is not the same as 3/2.
Challenges Stacks: Balanced Brackets Queues: A Tale of Two Stacks References "Stacks and Queues". Victor S.Adamchik, CMU. 2009

1.2.4 Tree

Binary Tree Fundamentally, a binary tree is composed of nodes connected by edges (with further restrictions discussed below). Some binary tree, tt, is either empty or consists of a single root element with two distinct binary tree child elements known as the left subtree and the right subtree of tt. As the name binary suggests, a node in a binary tree has a maximum of 22 children.
The following diagrams depict two different binary trees:
Here are the basic facts and terms to know about binary trees:
The convention for binary tree diagrams is that the root is at the top, and the subtrees branch down from it. A node's left and right subtrees are referred to as children, and that node can be referred to as the parent of those subtrees. A non-root node with no children is called a leaf. Some node aa is an ancestor of some node bb if bb is located in a left or right subtree whose root node is aa. This means that the root node of binary tree tt is the ancestor of all other nodes in the tree. If some node aa is an ancestor of some node bb, then the path from aa to bb is the sequence of nodes starting with aa, moving down the ancestral chain of children, and ending with bb. The depth (or level) of some node aa is its distance (i.e., number of edges)

from the tree's root node. Simply put, the height of a tree is the number of edges between the root node and its furthest leaf. More technically put, it's 1+max(height(leftSubtree),height(rightSubtree))1+max(height(leftSubtree),height(rightSubtree))) (i.e., one more than the maximum of the heights of its left and right subtrees). Any node has a height of 11, and the height of an empty subtree is 11. Because the height of each node is 1+1+ the maximum height of its subtrees and an empty subtree's height is 11, the height of a single-element tree or leaf node is 00. Let's apply some of the terms we learned above to the binary tree on the right:

The root node is AA.

The respective left and right children of AA are BB and EE. The left child of BB is CC. The respective left and right children of EE are FF and DD.

Nodes CC, FF, and DD are leaves (i.e., each node is a leaf).

The root is the ancestor of all other nodes, BB is an ancestor of CC, and EE is an ancestor of FF and DD.

The path between AA and CC is ABCABC. The path between AA and FF is AEFAEF. The path between AA and DD is A EDED.

The depth of root node AA is 00. The depth of nodes BB and EE is 11. The depth of nodes CC, FF, and DD, is 22.

The height of the tree, height(t)height(t), is 22. We calculate this recursively as height(t)=1+(max(height(root.leftChild),height(root.rightChild)))height(t)=1+(max(height(root.leftChild),

Because this is long and complicated when expanded, we'll break it down using an image of a slightly simpler version of tt whose height is still 22:

Binary Search Tree A Binary Search Tree (BST), tt, is a binary tree that is either empty or satisfies the following three conditions:

Each element in the left subtree of tt is less than or equal to the root element of tt (i.e., max(leftTree(t).value)t.valuemax(leftTree(t).value)t.valuemax(leftTree(t).value)t.valuemax(leftTree(t).v

Each element in the right subtree of tt is greater than the root element of tt (i.e., max(rightTree(t).value)t.valuemax(rightTree(t).value)t.valuemax(rightTree(t).value)t.valuemax(rightTree(t).

Both leftTree(t)leftTree(t) and rightTree(t)rightTree(t) are BSTs.

You can essentially think of it as a regular binary tree where for each node parent having a leftChildleftChild and rightChildrightChild, leftChild.valueparent.valuerightChild.valueleftChild.valu

In the first diagram at the top of this article, the binary tree of integers on the left side is a binary search tree.

Advantages and Drawbacks Searching for elements is very fast. We know that each node has a maximum of two children and we know that the items are always in the left subtree and the » items are always in the right subtree. To search for an element, we simply need to compare the value we want against the value stored in the root node of the current subtree and work our way down the appropriate child subtrees until we either find the value we're looking for or we hit null (i.e., an empty subtree) which indicates the item is not in the BST. Inserting or searching for a node in a balanced tree is O(logn)O(logn) because you're discarding half of the possible values each time you go left or right.

It can easily become unbalanced. Depending on the insertion order, the tree can very easily become unbalanced (which makes for longer search times). For example, if we create a new tree where the sequence of inserted nodes is 213456213456, we end up with the following unbalanced tree:

Observe that the root's left subtree only has one node, whereas the root's right subtree has four nodes. For this reason, inserting or searching for a node in an unbalanced tree is O(n)O(n).

Challenges "Trees: Is This a Binary Search Tree?". hackerrank. 2016 References "Binary Trees and Binary Search Trees". AllisonP, hackerrank. 2016

1.2.5 Binary Search Tree

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties
The left sub-tree of a node has a key less than or equal to its parent node's key. The right sub-tree of a node has a key greater than to its parent node's key. Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as
$left_subtree(keys)node(key)right_subtree(keys)RepresentationBSTisacollectionofnodesarrangedinawaywhe$
Following is a pictorial representation of BST
We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.
Basic Operations Following are the basic operations of a tree
Search Searches an element in a tree. Insert Inserts an element in a tree. Pre-order Traversal Traverses a tree in a pre-order manner. In-order Traversal Traverses a tree in an in-order manner. Post-order Traversal Traverses a tree in a post-order manner. Node Define a node having some data, references to its left and right child nodes.
struct node int data; struct node *leftChild; struct node *rightChild; ; Search Operation Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.
Algorithm
struct node* search(int data) struct node *current = root; printf("Visiting elements: ");
while(current->data != data)
if(current != NULL) printf("
//go to left tree if(current->data > data) current = current->leftChild; //else go to right tree else current = current->rightChild;
//not found if(current == NULL) return NULL; return current; Insert Operation Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.
Algorithm
void insert(int data) struct node *tempNode = (struct node*) malloc(sizeof(struct node)); struct node *current; struct node *parent;
tempNode->data = data; tempNode->leftChild = NULL; tempNode->rightChild = NULL;
//if tree is empty if(root == NULL) root = tempNode; else current = root; parent = NULL;
while(1) parent = current;
//go to left of the tree if(data < parent->data) current = current->leftChild; //insert to the left
if(current == NULL) parent->leftChild = tempNode; return; //go to right of the tree else current = current->rightChild;

//insert to the right if(current == NULL) parent->rightChild = tempNode; return;

1.3 Heaps

A heap is just what it sounds like — a pile of values organized into a binary tree-like structure adhering to some ordering property. When we add elements to a heap, we fill this tree-like structure from left to right, level by level. This makes heaps really easy to implement in an array, where the value for some index ii's left child is located at index 2i+12i+1 and the value for its right child is at index 2i+22i+2 (using zero-indexing). Here are the two most fundamental heap operations:
add: Insert an element into the heap. You may also see this referred to as push.
poll: Retrieve and remove the root element of the heap. You may also see this referred to as pop. Max Heap This type heap orders the maximum value at the root.
When we add the values 12341234 to a Max heap, it looks like this:
When we poll the same Max heap until it's empty, it looks like this:
Min Heap This type of heap orders the minimum value at the root.
When we add the values 12341234 to a Min heap, it looks like this:
When we poll the same Min heap until it's empty, it looks like this:
Applications The heap data structure has many applications.
Heapsort: One of the best sorting methods being in-place and with no quadratic worst-case scenarios. Selection algorithms: A heap allows access to the min or max element in constant time, and other selections (such as median or kth-element) can be done in sub-linear time on data that is in a heap. Graph algorithms: By using heaps as internal traversal data structures, run time will be reduced by polynomial order. Examples of such problems are Prim's minimal-spanning-tree algorithm and Dijkstra's shortest-path algorithm. Priority Queue: A priority queue is an abstract concept like "a list" or "a map"; just as a list can be implemented with a linked list or an array, a priority queue can be implemented with a heap or a variety of other methods. Order statistics: The Heap data structure can be used to efficiently find the kth smallest (or largest) element in an array. Challenges "Heaps: Find the Running Median". hackerrank. 2016 References "Heaps". AllisonP, hackerrank. 2016 "Heap (data structure)". wikipedia. 2016

1.4 Sort

1.4.1 Introduction

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.
The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios

Telephone Directory The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily. Dictionary The dictionary stores words in an alphabetical order so that searching of any word becomes easy. In-place Sorting and Not-in-place Sorting Sorting algorithms may require some extra space for comparison and temporary storage of few data elements. These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself. This is called in-place sorting. Bubble sort is an example of in-place sorting.
However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted. Sorting which uses equal or more space is called not-in-place sorting. Merge-sort is an example of not-in-place sorting.
Stable and Not Stable Sorting If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called stable sorting.
If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called unstable sorting.
Stability of an algorithm matters when we wish to maintain the sequence of original elements, like in a tuple for example.
Adaptive and Non-Adaptive Sorting Algorithm A sorting algorithm is said to be adaptive, if it takes advantage of already 'sorted' elements in the list that is to be sorted. That is, while sorting if the source list has some element already sorted, adaptive algorithms will take this into account and will try not to re-order them.
A non-adaptive algorithm is one which does not take into account the elements which are already sorted. They try to force every single element to be re-ordered to confirm their sortedness.
Important Terms Some terms are generally coined while discussing sorting techniques, here is a brief introduction to them
Increasing Order
A sequence of values is said to be in increasing order, if the successive element is greater than the previous one. For example, 1, 3, 4, 6, 8, 9 are in increasing order, as every next element is greater than the previous element.
Decreasing Order
A sequence of values is said to be in decreasing order, if the successive element is less than the current one. For example, 9, 8, 6, 4, 3, 1 are in decreasing order, as every next element is less than the previous element.
Non-Increasing Order
A sequence of values is said to be in non-increasing order, if the successive element is less than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. For example, 9, 8, 6, 3, 3, 1 are in non-increasing order, as every next element is less than or equal to (in case of 3) but not greater than any previous element.
Non-Decreasing Order
A sequence of values is said to be in non-decreasing order, if the successive element is greater than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. For example, 1, 3, 3, 6, 8, 9 are in non-decreasing order, as every next element is greater than or equal to (in case of 3) but not less than the previous one.

1.4.2 Bubble Sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2)(n2) where n is the number of items.
How Bubble Sort Works? We take an unsorted array for our example. Bubble sort takes O(n2)(n2) time so we're keeping it short and precise.
Bubble sort starts with very first two elements, comparing them to check which one is greater.
In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.
We find that 27 is smaller than 33 and these two values must be swapped.
The new array should look like this
Next we compare 33 and 35. We find that both are in already sorted positions. Then we move to the next two values, 35 and 10.
We know then that 10 is smaller 35. Hence they are not sorted.
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this
Notice that after each iteration, at least one value moves at the end.
And when there's no swap required, bubble sorts learns that an array is completely sorted.
Now we should look into some practical aspects of bubble sort.
Algorithm We assume list is an array of n elements. We further assume that swap function swaps the values of the given array elements.
begin BubbleSort(list)
for all elements of list if list[i] > list[i+1] swap(list[i], list[i+1]) end if end for return list
end BubbleSort Pseudocode We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.
To ease-out the issue, we use one flag variable swapped which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.
Pseudocode of BubbleSort algorithm can be written as follows
procedure bubbleSort(list : array of items)
loop = list.count;
for i = 0 to loop-1 do: swapped = false
for j = 0 to loop-1 do:
/* compare the adjacent elements */ if list[j] > list[j+1] then /* swap them */ swap(list[j], list[j+1]) swapped = true end if
end for
/*if no number was swapped that means array is sorted now, break the loop.*/ if(not swapped) then break end if
end for

end procedure return list Implementation One more issue we did not address in our original algorithm and its improvised pseudocode, is that, after every iteration the highest values settles down at the end of the array. Hence, the next iteration need not include already sorted elements. For this purpose, in our implementation, we restrict the inner loop to avoid already sorted values.

1.4.3 Insertion Sort

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.
The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of (n2), where n is the number of items.
How Insertion Sort Works? We take an unsorted array for our example.
Insertion sort compares the first two elements.
It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.
Insertion sort moves ahead and compares 33 with 27.
And finds that 33 is not in the correct position.
It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.
By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.
These values are not in a sorted order.
So we swap them.
However, swapping makes 27 and 10 unsorted.
Hence, we swap them too.
Again we find 14 and 10 in an unsorted order.
We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.
This process goes on until all the unsorted values are covered in a sorted sub-list.
Now we shall see some programming aspects of insertion sort.
Algorithm Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.
Step 1 If it is the first element, it is already sorted. return 1; Step 2 Pick next element Step 3 Compare with all elements in the sorted sub-list Step 4 Shift all the elements in the sorted sub-list that is greater than the value to be sorted Step 5 Insert the value Step 6 Repeat until list is sorted Pseudocode procedure insertionSort(A : array of items) int holePosition int valueToInsert
for i = 1 to length(A) inclusive do:
/* select value to be inserted */ valueToInsert = A[i] holePosition = i
/*locate hole position for the element to be inserted */
while holePosition > 0 and A[holePosition-1] > valueToInsert do: A[holePosition] = A[holePosition-1] holePosition = holePosition -1 end while
/* insert the number at hole position */ A[holePosition] = valueToInsert
end for

end procedure

1.4.4 Selection Sort

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.
The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.
This algorithm is not suitable for large data sets as its average and worst case complexities are of O(n2)(n2), where nn is the number of items.
How Selection Sort Works? Consider the following depicted array as an example.
For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.
So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.
For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.
We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.
After two iterations, two least values are positioned at the beginning in a sorted manner.
The same process is applied to the rest of the items in the array.
Following is a pictorial depiction of the entire sorting process
Now, let us learn some programming aspects of selection sort.
Algorithm Step 1 Set MIN to location 0 Step 2 Search the minimum element in the list Step 3 Swap with value at location MIN Step 4 Increment MIN to point to next element Step 5 Repeat until list is sorted Pseudocode procedure selection sort list : array of items n : size of list
for i = 1 to n - 1 /* set current element as minimum*/ min = i
/* check the element to be minimum */
for j = i+1 to n if list[j] < list[min] then min = j; end if end for
/* swap the minimum element with the current element*/ if indexMin != i then swap list[min] and list[i] end if
end for
end procedure

1.4.5 Merge Sort

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(nlogn)(nlogn), it is one of the most respected algorithms.
Merge sort first divides the array into equal halves and then combines them in a sorted manner.
How Merge Sort Works? To understand merge sort, we take an unsorted array as the following

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.
This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.
We further divide these arrays and we achieve atomic value which can no more be divided.
Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.
We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.
In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.
After the final merging, the list should look like this
Now we should learn some programming aspects of merge sorting.
Algorithm Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.
Step 1 if it is only one element in the list it is already sorted, return. Step 2 divide the list recursively into two halves until it can no more be divided. Step 3 merge the smaller lists into new list in sorted order. Pseudocode We shall now see the pseudocodes for merge sort functions. As our algorithms point out two main functions divide merge.
Merge sort works with recursion and we shall see our implementation in the same way.
procedure mergesort(var a as array) if (n == 1) return a
var l1 as array = a[0] ... a[n/2] var l2 as array = a[n/2+1] ... a[n]
l1 = mergesort(l1) l2 = mergesort(l2)
return merge(l1, l2) end procedure
procedure merge(var a as array, var b as array)
var c as array
while (a and b have elements) if (a[0] > b[0]) add b[0] to the end of c remove b[0] from b else add a[0] to the end of c remove a[0] from a end if end while
while (a has elements) add a[0] to the end of c remove a[0] from a end while
while (b has elements) add b[0] to the end of c remove b[0] from b end while
return c
end procedure

1.4.6 Shell Sort

Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.
This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as interval. This interval is calculated based on Knuth's formula as
Knuth's Formula h=h3+1h=h3+1

where
hh is interval with initial value 1 This algorithm is quite efficient for medium-sized data sets as its average and worst case complexity are of O(n)(n), where nn is the number of items.
How Shell Sort Works? Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are 35, 14, 33, 19, 42, 27 and 10, 44
We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this
Then, we take interval of 2 and this gap generates two sub-lists - 14, 27, 35, 42, 19, 10, 33, 44
We compare and swap the values, if required, in the original array. After this step, the array should look like this
Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array.
Following is the step-by-step depiction
We see that it required only four swaps to sort the rest of the array.
Algorithm Following is the algorithm for shell sort.
Step 1 Initialize the value of h Step 2 Divide the list into smaller sub-list of equal interval h Step 3 Sort these sub-lists using insertion sort Step 3 Repeat until complete list is sorted Pseudocode Following is the pseudocode for shell sort.
procedure shellSort() A : array of items
/* calculate interval*/ while interval < A.length /3 do: interval = interval * 3 + 1 end while
while interval > 0 do:
for outer = interval; outer < A.length; outer ++ do:
/* select value to be inserted */ valueToInsert = A[outer] inner = outer;
/*shift element towards right*/ while inner > interval -1 A[inner - interval] >= valueToInsert do: A[inner] = A[inner - interval] inner = inner - interval end while
/* insert the number at hole position */ A[inner] = valueToInsert
end for
/* calculate interval*/ interval = (interval -1) /3;
end while
end procedure

1.4.7 Quick Sort

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.
Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of (n2), where n is the number of items.

Partition in Quick Sort Following animated representation explains how to find the pivot value in an array.
The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.
Quick Sort Pivot Algorithm Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.
Step 1 Choose the highest index value has pivot Step 2 Take two variables to point left and right of the list excluding pivot Step 3 left points to the low index Step 4 right points to the high Step 5 while value at left is less than pivot move right Step 6 while value at right is greater than pivot move left Step 7 if both step 5 and step 6 does not match swap left and right Step 8 if left right, the point where they met is new pivot Quick Sort Pivot Pseudocode The pseudocode for the above algorithm can be derived as
function partitionFunc(left, right, pivot) leftPointer = left rightPointer = right - 1
while True do while A[++leftPointer] < pivot do //do-nothing end while
while rightPointer > 0 A[–rightPointer] > pivot do //do-nothing end while
if leftPointer >= rightPointer break else swap leftPointer,rightPointer end if
end while
swap leftPointer,right return leftPointer
end function Quick Sort Algorithm Using pivot algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows
Step 1 Make the right-most index value pivot Step 2 partition the array using pivot value Step 3 quicksort left partition recursively Step 4 quicksort right partition recursively Quick Sort Pseudocode To get more into it, let see the pseudocode for quick sort algorithm
procedure quickSort(left, right)
if right-left <= 0 return else pivot = A[right] partition = partitionFunc(left, right, pivot) quickSort(left,partition-1) quickSort(partition+1,right) end if
end procedure

1.5 Search

1.5.1 Linear Search

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.
Algorithm Linear Search (Array A, Value x)
Step 1: Set i to 1 Step 2: if i > n then go to step 7 Step 3: if A[i] = x then go to step 6 Step 4: Set i to i + 1 Step 5: Go to Step 2 Step 6: Print Element x Found at index i and go to step 8 Step 7: Print element not found Step 8: Exit
Pseudocode procedure $\text{linear}_s earch(list, value)$
for each item in the list
if match item == value
return the item's location
end if

end for
end procedure

1.5.2 Binary Search

Binary search is a fast search algorithm with run-time complexity of (log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.
Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.
How Binary Search Works? For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.
First, we shall determine half of the array by using this formula
mid = low + (high - low) / 2 Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.
Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.
We change our low to mid + 1 and find the new mid value again.
low = mid + 1 mid = low + (high - low) / 2 Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.
The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.
Hence, we calculate the mid again. This time it is 5.
We compare the value stored at location 5 with our target value. We find that it is a match.
We conclude that the target value 31 is stored at location 5.
Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.
Pseudocode The pseudocode of binary search algorithms should look like this
Procedure $binary_search A sorted array n size of array x value to be searched$
Set lowerBound = 1 Set upperBound = n
while x not found if upperBound < lowerBound EXIT: x does not exists.
set midPoint = lowerBound + (upperBound - lowerBound) / 2
if A[midPoint] < x set lowerBound = midPoint + 1
if A[midPoint] > x set upperBound = midPoint - 1
if A[midPoint] = x EXIT: x found at location midPoint
end while
end procedure

1.5.3 Interpolation Search

Interpolation search is an improved variant of binary search. This search algorithm works on the probing position of the required value. For this algorithm to work properly, the data collection should be in a sorted form and equally distributed.
Binary search has a huge advantage of time complexity over linear search. Linear search has worst-case complexity of (n) whereas binary search has (log n).
There are cases where the location of target data may be known in advance. For example, in case of a telephone directory, if we want to search the telephone number of Morphius. Here, linear search and even binary search will seem slow as we can directly jump to memory space where the names start from 'M' are stored.
Positioning in Binary Search In binary search, if the desired data is not found then the rest of the list is divided in two parts, lower and higher. The search is carried out in either of them.
Even when the data is sorted, binary search does not take advantage to probe the position of the desired data.
Position Probing in Interpolation Search Interpolation search finds a particular item by computing the probe position. Initially, the probe position is the position of the middle most item of the collection.
If a match occurs, then the index of the item is returned. To split the list into two parts, we use the following method
mid = Lo + ((Hi - Lo) / (A[Hi] - A[Lo])) * (X - A[Lo]) where
A = list Lo = Lowest index of the list Hi = Highest index of the list A[n] = Value stored at index n in the list If the middle item is greater than the item, then the probe position is again calculated in the sub-array to the right of the middle item. Otherwise, the item is searched in the subarray to the left of the middle item. This process continues on the sub-array as well until the size of subarray reduces to zero.
Runtime complexity of interpolation search algorithm is O(log(logn))(log(logn)) as compared to O(logn)(logn) of BST in favorable situations.
Algorithm As it is an improvisation of the existing BST algorithm, we are mentioning the steps to search the 'target' data value index, using position probing
Step 1 Start searching data from middle of the list. Step 2 If it is a match, return the index of the item, and exit. Step 3 If it is not a match, probe position. Step 4 Divide the list using probing formula and find the new midle. Step 5 If data is greater than middle, search in higher sub-list. Step 6 If data is smaller than middle, search in lower sub-list. Step 7 Repeat until match. Pseudocode
A Array list N Size of A X Target Value
Procedure Interpolation$_S earch$()
Set Lo 0 Set Mid -1 Set Hi N-1
While X does not match
if Lo equals to Hi OR A[Lo] equals to A[Hi] EXIT: Failure, Target not found end if
Set Mid = Lo + ((Hi - Lo) / (A[Hi] - A[Lo])) * (X - A[Lo])
if A[Mid] = X EXIT: Success, Target found at Mid else if A[Mid] $<$ X Set Lo to Mid+1 else if A[Mid] $>$ X Set Hi to Mid-1 end if end if
End While
End Procedure

1.5.4 Hash Table

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.
Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.
Hashing Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.
(1,20) (2,70) (42,80) (4,25) (12,44) (14,32) (17,11) (13,78) (37,98) Sr. No. Key Hash Array Index 1 1 1 2 2 2 3 42 42 4 4 4 5 12 12 6 14 4 7 17 7 8 13 3 9 37 7 Linear Probing As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.
Sr. No. Key Hash Array Index After Linear Probing, Array Index 1 1 1 2 2 2 3 42 42 4 4 4 5 12 12 6 14 14 7 17 17 8 13 13 9 37 37 Basic Operations Following are the basic primary operations of a hash table.
Search Searches an element in a hash table. Insert inserts an element in a hash table. delete Deletes an element from a hash table. DataItem Define a data item having some data and key, based on which the search is to be conducted in a hash table.
struct DataItem int data; int key; ; Hash Method Define a hashing method to compute the hash code of the key of the data item.
int hashCode(int key) return key Search Operation Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code.
Example
struct DataItem *search(int key) //get the hash int hashIndex = hashCode(key);
//move in array until an empty while(hashArray[hashIndex] != NULL)
if(hashArray[hashIndex]->key == key) return hashArray[hashIndex];
//go to next cell ++hashIndex;
//wrap around the table hashIndex
return NULL; Insert Operation Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code.
Example
void insert(int key,int data) struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem)); item->data = data; item->key = key;
//get the hash int hashIndex = hashCode(key);
//move in array until an empty or deleted cell while(hashArray[hashIndex] != NULL hashArray[hashIndex]->key != -1) //go to next cell ++hashIndex;
//wrap around the table hashIndex

hashArray[hashIndex] = item; Delete Operation Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.
Example
struct DataItem* delete(struct DataItem* item) int key = item->key;
//get the hash int hashIndex = hashCode(key);
//move in array until an empty while(hashArray[hashIndex] !=NULL)
if(hashArray[hashIndex]->key == key) struct DataItem* temp = hashArray[hashIndex];
//assign a dummy item at deleted position hashArray[hashIndex] = dummyItem; return temp;
//go to next cell ++hashIndex;
//wrap around the table hashIndex
return NULL;

1.6 Graph

1.6.1 Graph Data Structure

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.
Formally, a graph is a pair of sets (V,E)(V,E), where VV is the set of vertices and EE is the set of edges, connecting the pairs of vertices. Take a look at the following graph
In the above graph,
V=a,b,c,d,eV=a,b,c,d,e
E=ab,ac,bd,cd,deE=ab,ac,bd,cd,de
Definitions Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms Vertex Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on. Edge Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0. Adjacency Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on. Path Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.
Basic Operations Following are basic primary operations of a Graph

Add Vertex Adds a vertex to the graph. Add Edge Adds an edge between the two vertices of the graph. Display Vertex Displays a vertex of the graph.

1.6.2 Depth First Traversal

Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.
As in the example given above, DFS algorithm traverses from A to B to C to D first then to E, then to F and lastly to G. It employs the following rules.
Rule 1 Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack. Rule 2 If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.) Rule 3 Repeat Rule 1 and Rule 2 until the stack is empty. Algorithms Step Traversal Description 1. Initialize the stack. 2. Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order. 3. Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both Sand D are adjacent to A but we are concerned for unvisited nodes only. 4. Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited. However, we shall again choose in an alphabetical order. 5. We choose B, mark it as visited and put onto the stack. Here Bdoes not have any unvisited adjacent node. So, we pop Bfrom the stack. 6. We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find D to be on the top of the stack. 7. Only unvisited adjacent node is from D is C now. So we visit C, mark it as visited and put it onto the stack.

1.6.3 Breadth First Traversal

Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.
As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.
Rule 1 Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it * in a queue. Rule 2 If no adjacent vertex is found, remove the first vertex from the queue. Rule 3 Repeat Rule 1 and Rule 2 until the queue is empty. Algorithms Step Traversal Description 1. Initialize the stack. 2. Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order. 3. Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both Sand D are adjacent to A but we are concerned for unvisited nodes only. 4. Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited. However, we shall again choose in an alphabetical order. 5. We choose B, mark it as visited and put onto the stack. Here Bdoes not have any unvisited adjacent node. So, we pop Bfrom the stack. 6. We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find D to be on the top of the stack. 7. Only unvisited adjacent

node is from D is C now. So we visit C, mark it as visited and put it onto the stack. At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over

1.7 String

String manipulation is a basic operation of many algorithms and utilities such as data validation, text parsing, file conversions and others. The Java APIs contain three classes that are used to work with character data:
Character – A class whose instances can hold a single character value. String – An immutable class for working with multiple characters. StringBuffer and StringBuilder – Mutable classes for working with multiple characters. The String and StringBuffer classes are two you will use the most in your programming assignments. You use the String class in situations when you want to prohibit data modification; otherwise you use the StringBuffer class.
The String class In Java Strings can be created in two different ways. Either using a new operator
String demo1 = new String("This is a string");
char[] demo2 = 's','t','r','i','n','g'; String str = new String(demo2); or using a string literal
String demo3 = "This is a string"; The example below demonstrates differences between these initializations
String s1 = new String("Fester"); String s2 = new String("Fester"); String s3 = "Fester"; String s4 = "Fester"; Then
s1 == s2 returns false s1 == s3 returns false s3 == s4 returns true Because of the importance strings in real life, Java stores (at compile time) all strings in a special internal table as long as you create your strings using a string literal String s3 = "Fester". This process is called canonicalization - it replaces multiple string objects with a single object. This is why in the above example s3 and s4 refer to the same object. Also note that creating strings like s3 and s4 is more efficient. Review the code example StringOptimization.java that demonstrates time comparisons between these two ways of string creation.
Here are some important facts you must know about strings:
1. A string is not an array of characters. Therefore, to access a particular character in a string, you have to use the charAt() method. In this code snippet we get the fourth character which is 't':
String str = "on the edge of history"; char ch = str.charAt(3); 2. The toString() method is used when we need a string representation of an object.
The method is defined in the Object class. For most important classes that you create, you will want to override toString() and provide your own string representation.
3. Comparing strings content using == is the most common mistake beginners do. You compare the content using either equals() or compareTo() methods.
Basic String methods The String class contains an enormous amount of useful methods for string manipulation. The following table presents the most common String methods:
str.charAt(k) returns a char at position k in str. str.substring(k) returns a substring from index k to the end of str s.substring(k, n) returns a substring from

index k to index n-1 of str str.indexOf(s) returns an index of the first occurrence of String s in str str.indexOf(s, k) returns an index of String s starting an index k in str str.startsWith(s) returns true if str starts with s str.startsWith(s, k) returns true if str starts with s at index k str.equals(s) returns true if the two strings have equal values str.equalsIgnoreCase(s) same as above ignoring case str.compareTo(s) compares two strings s.compareToIgnoreCase(t) same as above ignoring case Examine the code in BasicStringDemo.java for further details.

The StringBuffer class In many cases when you deal with strings you will use methods available in the companion StringBuffer class. This mutable class is used when you want to modify the contents of the string. It provides an efficient approach to dealing with strings, especially for large dynamic string data. StringBuffer is similar to ArrayList in a way that the memory allocated to an object is automatically expanded to take up additional data.

Here is an example of reversing a string using string concatenation
public static String reverse1(String s) String str = "";
for(int i = s.length() - 1; i>=0; i–) str += s.charAt(i);
return str; and using a StringBuffer's append
public static String revers2(String s) StringBuffer sb = new StringBuffer();
for(int i = s.length() - 1; i>=0; i–) sb.append(s.charAt(i));
return sb.toString(); Another way to reverse a string is to convert a String object into a StringBuffer object, use the reverse method, and then convert it back to a string:
public static String reverse3(String s) return new StringBuffer(s).reverse().toString();

The performance difference between these two classes is that StringBuffer is faster than String when performing concatenations. Each time a concatenation occurs, a new string is created, causing excessive system resource consumption. Review the code example StringOverhead.java that demonstrates time comparisons of concatenation on Strings and StringBuffer.

StringTokenizer This class (from java.util package) allows you to break a string into tokens (substrings). Each token is a group of characters that are separated by delimiters, such as an empty space, a semicolon, and so on. So, a token is a maximal sequence of consecutive characters that are not delimiters. Here is an example of the use of the tokenizer (an empty space is a default delimiter):
String s = "Nothing is as easy as it looks"; StringTokenizer st = new StringTokenizer(s); while (st.hasMoreTokens()) String token = st.nextToken(); System.out.println("Token [" + token + "]"); Here, hasMoreTokens() method checks if there are more tokens available from the string, and nextToken() method returns the next token from the string tokenizer.

The set of delimiters (the characters that separate tokens) may be specified in the second argument of StringTokenizer. In the following example, StringTokenizer has a set of two delimiters: an empty space and an underscore:
String s = "Every$_s$olution$_b$reedsnewproblems"; $StringTokenizerst = newStringTokenizer(s, "_"); while(st.h$
$handcolumnspecifiestheregularexpressionconstructs, whiletheright-handcolumndescribestheconditionsu$

Character Classes
[abc] a, b, or c (simple class) $[^abc]Anycharacterexcepta, b, orc(negation)[a-zA-Z]athroughz, orAthroughZ, inclusive(range)[a-d[m-p]]athroughd, ormthroughp : [a-dm-p](union)[a-z[def]]d, e, orf(intersection)[a-z[^bc]]athroughz, exceptforbandc : [ad-z](subtraction)[a-z[^m-p]]athroughz, andnotmthroughp : [a-lq-z](subtraction)$
$danydigitfrom0to9$

$wanywordcharacter(a-z, A-Z, 0-9and_)$
$Wanynon-wordcharacter$
$sanywhitespacecharacter?appearingonceornotatall*appearingzeroormoretimes+$
$appearingoneormoretimesTheJavaStringclasshasseveralmethodsthatallowyoutoperformanoperationusin$
The matches() method The matches("regex") method returns true or false depending whether the string can be matched entirely by the regular expression "regex". For example,
"abc".matches("abc") returns True, but
"abc".matches("bc") returns False. In the following code examples we match all strings that start with any number of dots (denoted by *), followed by "abc" and end with one or more underscores (denoted by +).
String regex = ".*"+"abc"+"$_+$";
"..abc$_{\text{".matches(regex);}}$
"abc$_{\text{".matches(regex);}}$
"abc".$matches(regex); ThereplaceAll()methodThemethodreplaceAll("regex","replacement")replaceseach$
$lettersfromagivenstring$
String str = "Nothing 2is as <> easy AS it $+_{=}looks$!"; $str = str.replaceAll("[^{a}-zA-$
$Z]","");Thepattern"[a-zA-Z]"describesallletters(inupperandlowercases).Nextwenegatethispattern,toge$
$letters"[^{a}-zA-Z]"$.
In the next example, we replace a sequence of characters by "-"
String str = "aabfooaaaabfooabfoob"; str = str.replaceAll("a*b", "-"); The star "" in the pattern "ab" denotes that character "a" may be repeated zero or more times. The output: "-foo-foo-foo-";
The split() method The split("regex") splits the string at each "regex" match and returns an array of strings where each element is a part of the original string between two "regex" matches.
In the following example we break a sentence into words, using an empty space as a delimiter:
String s = "Nothing is as easy as it looks"; String[] st = s.split(" "); Tokens are stored in in an array of strings and could be be easily accessible using array indexes. In the next code example, we choose two delimiters: either an empty space or an underscore:
String s = "Every$_{s}olution_{b}reedsnewproblems"; $String[]st = s.split("_|");Whatifastringcontainsseveralund$
$", thatdenotesarepetitivepattern$
String s = "Every$_{s}olution_{b}^{reeds_{n}ew_{p}roblems";String[]st=s.split("_+");It'simportanttoobservethatsplit()mightreturnsemptytokens.Inth$
String[] st = "Tomorrow".split("r"); we have three tokens, where the second token is empty string. That is so because split() returns tokens between two "regex" matches.
One of the widely use of split() is to break a given text file into words. This could be easily done by means of the metacharacter "" (any non-word character), which allows you to perform a "whole words only" search using a regular expression. A "word character" is either an alphabet character (a-z and A-Z) or a digit (0-9) or a underscore.
"Let's go, Steelers!!!".split("
W"); returns the following array of tokens
[Let, s, go, Steelers] Examine the code in Split.java for further details.
Pattern matching Pattern matching in Java is based on use of two classes
Pattern - compiled representation of a regular expression. Matcher - an engine that performs match operations. A typical invocation is the following, first we

create a pattern
String seq = "CCCAA"; Pattern p = Pattern.compile("C*A*"); In this example we match all substrings that start with any number of Cs followed by any number of As. Then we create a Matcher object that can match any string against our pattern
Matcher m = p.matcher(seq); Finally, we do actual matching
boolean res = m.matches(); The Matcher class has another widely used method, called find(), that finds next substring that matches a given pattern. In the following example we cound the number of matches "ACC"
String seq = "CGTATCCCACAGCACCACACCCAACAACCCA"; Pattern p = Pattern.compile("A1C2"); Matcher m = p.matcher(seq); int count = 0; while(m.find()) count++; System.out.println("there are " + count + " ACC"); Examine the code example Matching.java for further details.
Pattern matching in Computational Biology
The DNA (the genetic blueprint) of any species is composed of about 4 billion ACGT nucleotides. DNA forms a double helix that has two strands of DNA binding and twisting together. In pattern matching problems we ignore the fact that DNA forms a double helix, and think of it only as a single strand. The other strand is complimentary. Knowing one strand allows uniquely determine the other one. Thus, DNA is essentially a linear molecule that looks like a string composed out of only four characters A, C, G, and T:
CGTATCCCACAGCACCACACCCAACAACCC Each nucleotides (also called a base) strongly binds to no more than two other bases. These links provides a linear model of DNA strand. The particular order of ACGT nucleotides is extremely important. Different orders generate humans, animals, corn, and other organisms. The size of the genome (a genome is all the DNA in an organism) does not necessarily correlate with the complexity of the organism it belongs to. Humans have less than a third as many genes as were expected.
Pattern matching in computational biology arises from the need to know characteristics of DNA sequences, such as
find the best way to align two sequences. find any common subsequences determine how well a sequence fits into a given model. Comparing various DNA sequencesn provide many uses. Current scientific theories suggest that very similar DNA sequences have a common ancestor. The more similar two sequences are, the more recently they evolved from a single ansestor. With such knowledge, for example, we can reconstruct a phylogenetic tree (known as a "tree of life".) that shows how long ago various organisms diverged and which species are closely related.
Challenges Strings: Making Anagrams References "Strings". Victor S.Adamchik, CMU. 2009

1.7.1 Tries

Introduction There are many algorithms and data structures to index and search strings inside a text, some of them are included in the standard libraries, but not all of them; the trie data structure is a good example of one that isn't.
Let word be a single string and let dictionary be a large set of words. If we have a dictionary, and we need to know if a single word is inside of the dictionary the tries are a data structure that can help us. But you may be asking yourself, "Why use tries if set and hash tables can do the same?"

There are two main reasons:
The tries can insert and find strings in O(L)O(L) time (where L represent the length of a single word). This is much faster than set , but is it a bit faster than a hash table. The set and the hash tables can only find in a dictionary words that match exactly with the single word that we are finding; the trie allow us to find words that have a single character different, a prefix in common, a character missing, etc.
The tries can be useful in TopCoder problems, but also have a great amount of applications in software engineering. For example, consider a web browser. Do you know how the web browser can auto complete your text or show you many possibilities of the text that you could be writing? Yes, with the trie you can do it very fast. Do you know how an orthographic corrector can check that every word that you type is in a dictionary? Again a trie. You can also use a trie for suggested corrections of the words that are present in the text but not in the dictionary.
What is a Tree? You may read about how wonderful the tries are, but maybe you don't know yet what the tries are and why the tries have this name. The word trie is an infix of the word "retrieval" because the trie can find a single word in a dictionary with only a prefix of the word. The main idea of the trie data structure consists of the following parts:
The trie is a tree where each vertex represents a single word or a prefix. The root represents an empty string (""), the vertexes that are direct sons of the root represent prefixes of length 1, the vertexes that are 2 edges of distance from the root represent prefixes of length 2, the vertexes that are 3 edges of distance from the root represent prefixes of length 3 and so on. In other words, a vertex that are k edges of distance of the root have an associated prefix of length k. Let v and w be two vertexes of the trie, and assume that v is a direct father of w, then v must have an associated prefix of w. The next figure shows a trie with the words "tree", "trie", "algo", "assoc", "all", and "also."
Note that every vertex of the tree does not store entire prefixes or entire words. The idea is that the program should remember the word that represents each vertex while lower in the tree.
Coding a Trie The tries can be implemented in many ways, some of them can be used to find a set of words in the dictionary where every word can be a little different than the target word, and other implementations of the tries can provide us with only words that match exactly with the target word. The implementation of the trie that will be exposed here will consist only of finding words that match exactly and counting the words that have some prefix. This implementation will be pseudo code because different coders can use different programming languages.
We will code these 4 functions:
addWord. This function will add a single string word to the dictionary. countPreffixes. This function will count the number of words in the dictionary that have a string prefix as prefix. countWords. This function will count the number of words in the dictionary that match exactly with a given string word. Our trie will only support letters of the English alphabet. We need to also code a structure with some fields that indicate the values stored in each vertex. As we want to know the number of words that match with a given string, every vertex should have a field to indicate that this vertex represents a complete word or only a prefix (for simplicity, a complete word is considered also a prefix) and

how many words in the dictionary are represented by that prefix (there can be repeated words in the dictionary). This task can be done with only one integer field words.
Because we want to know the number of words that have like prefix a given string, we need another integer field prefixes that indicates how many words have the prefix of the vertex. Also, each vertex must have references to all his possible sons (26 references). Knowing all these details, our structure should have the following members:
structure Trie integer words; integer prefixes; reference edges[26]; And we also need the following functions:
initialize(vertex) addWord(vertex, word); integer countPrefixes(vertex, prefix); integer countWords(vertex, word); First of all, we have to initialize the vertexes with the following function:
initialize(vertex) vertex.words=0 vertex.prefixes=0 for i=0 to 26 edges[i]=NoEdge
The addWord function consists of two parameters, the vertex of the insertion and the word that will be added. The idea is that when a string word is added to a vertex vertex, we will add word to the corresponding branch of vertex cutting the leftmost character of word. If the needed branch does not exist, we will have to create it. All the TopCoder languages can simulate the process of cutting a character in constant time instead of creating a copy of the original string or moving the other characters.
addWord(vertex, word) if isEmpty(word) vertex.words=vertex.words+1 else vertex.prefixes=vertex.prefixes+1 k=firstCharacter(word) if(notExists(edges[k])) edges[k]=createEdge() initialize(edges[k]) cutLeftmostCharacter(word) addWord(edges[k], word) The functions countWords and countPrefixes are very similar. If we are finding an empty string we only have to return the number of words/prefixes associated with the vertex. If we are finding a non-empty string, we should to find in the corresponding branch of the tree, but if the branch does not exist, we have to return 0.
countWords(vertex, word) k=firstCharacter(word) if isEmpty(word) return vertex.words else if notExists(edges[k]) return 0 else cutLeftmostCharacter(word) return countWords(edges[k], word);
countPrefixes(vertex, prefix) k=firstCharacter(prefix) if isEmpty(word) return vertex.prefixes else if notExists(edges[k]) return 0 else cutLeftmostCharacter(prefix) return countWords(edges[k], prefix) Complexity Analysis In the introduction you may read that the complexity of finding and inserting a trie is linear, but we have not done the analysis yet. In the insertion and finding notice that lowering a single level in the tree is done in constant time, and every time that the program lowers a single level in the tree, a single character is cut from the string; we can conclude that every function lowers L levels on the tree and every time that the function lowers a level on the tree, it is done in constant time, then the insertion and finding of a word in a trie can be done in O(L) time. The memory used in the tries depends on the methods to store the edges and how many words have prefixes in common.
Other Kinds of Tries We used the tries to store words with lowercase letters, but the tries can be used to store many other things. We can use bits or bytes instead of lowercase letters and every data type can be stored in the tree, not only strings. Let flow your imagination using tries! For example, suppose that you want to find a word in a dictionary but a single letter was deleted from the word. You can modify the countWords function:

countWords(vertex, word, missingLetters) k=firstCharacter(word) if isEmpty(word) return vertex.word else if notExists(edges[k]) and missingLetters=0 return 0 else if notExists(edges[k]) cutLeftmostCharacter(word) return countWords(vertex, word, missingLetters-1) Here we cut a character but we don't go lower in the tree else We are adding the two possibilities: the first character has been deleted plus the first character is present r=countWords(vertex, word, missingLetters-1) cutLeftmostCharacter(word) r=r+countWords(edges[k], word, missingLetters) return r The complexity of this function may be larger than the original, but it is faster than checking all the subsets of characters of a word.
Challenges "Tries: Contacts". hackerrank. 2016 References "Using Tries – Topcoder". Topcoder.com. N.p., 2016. Web. 11 Oct. 2016.

1.7.2 Suffix Array and suffix tree

A suffix tree T is a natural improvement over trie used in pattern matching problem, the one defined over a set of substrings of a string s. The idea is very simple here. Such a trie can have a long paths without branches. If we only can reduce these long paths into one jump, we will reduce the size of the trie significantly, so this is a great first step in improving the complexity of operations on such a tree. This reduced trie defined over a subset of suffixes of a string s is called a suffix tree of s
For better understanding, let's consider the suffix tree T for a string s = abakan. A word abakan has 6 suffixes abakan , bakan, akan, kan, an, n and its suffix tree looks like this:
There is a famous algorithm by Ukkonen for building suffix tree for s in linear time in terms of the length of s. However, because it may look quite complicated at first sight, many people are discouraged to learn how it works. Fortunately, there is a great, I mean an excellent, description of Ukkonen's algorithm given on StackOverflow. Please refer to it for better understanding what a suffix tree is and how to build it in linear time.
Suffix trees can solve many complicated problems, because it contain so many information about the string itself. Fo example, in order to know how many times a pattern P occurs in s, it is sufficient to find P in T and return the size of a subtree corresponding to its node. Another well known application is finding the number of distinct substrings of s, and it can be solved easily with suffix tree, while the problem looks very complicated at first sight.
The post I linked from StackOverflow is so great, that you simple must read it. After that, you will be able to identify problems solvable with suffix trees easily. If you want to know more about when to use a suffix tree, you should read this paper about the applications of suffix trees.
Suffix Array Suffix array is a very nice array based structure. Basically, it is a lexicographically sorted array of suffixes of a string s. For example, let's consider a string s = abakan. A word abakan has 6 suffixes abakan , bakan, akan, kan, an, n and its suffix tree looks like this:
Of course, in order to reduce space, we do not store the exact suffixes. It is sufficient to store their indices.
Suffix arrays, especially combined with LCP table (which stands for Longest Common Prefix of neighboring suffixes table), are very very useful for solving many problems. I recommend reading this nice programming oriented paper

about suffix arrays, their applications and related problems by Stanford University.
Suffix arrays can be build easily in O(nlog2n)O(nlog2n) time, where n is the length of s, using the algorithm proposed in the paper from the previous paragraph. This time can be improved to O(nlogn)O(nlogn) using linear time sorting algorithm.
However, there is so extraordinary, cool and simple linear time algorithm for building suffix arrays by Kärkkäinen and Sanders, that reading it is a pure pleasure and you cannot miss it.
Correspondence between suffix tree and suffix array
It is also worth to mention, that a suffix array can be constructed directly from a suffix tree in linear time using DFS traversal. Suffix tree can be also constructed from the suffix array and LCP table as described here.

1.7.3 Knuth-Morris-Pratt Algorithm

The problem:
given a (short) pattern and a (long) text, both strings, determine whether the pattern appears somewhere in the text.
We'll go through the Knuth-Morris-Pratt (KMP) algorithm, which can be thought of as an efficient way to build these automata. I also have some working C++ source code which might help you understand the algorithm better.
First let's look at a naive solution.
suppose the text is in an array: char T[n]T[n] and the pattern is in another array: char P[m]P[m]. One simple method is just to try each possible position the pattern could appear in the text.
Naive string matching:
for (i=0; T[i] != ”; i++) for (j=0; T[i+j] != ” P[j] != ” T[i+j]==P[j]; j++) ; if (P[j] == ”) found a match There are two nested loops; the inner one takes O(m) iterations and the outer one takes O(n) iterations so the total time is the product, O(mn). This is slow; we'd like to speed it up.
In practice this works pretty well – not usually as bad as this O(mn) worst case analysis. This is because the inner loop usually finds a mismatch quickly and move on to the next position without going through all m steps. But this method still can take O(mn) for some inputs. In one bad example, all characters in T[] are "a"s, and P[] is all "a"'s except for one "b" at the end. Then it takes m comparisons each time to discover that you don't have a match, so mn overall.
Here's a more typical example. Each row represents an iteration of the outer loop, with each character in the row representing the result of a comparison (X if the comparison was unequal). Suppose we're looking for pattern "nano" in text "bananananobano".
0 1 2 3 4 5 6 7 8 9 10 11 T: b a n a n a n o b a n o
i=0: X i=1: X i=2: n a n X i=3: X i=4: n a n o i=5: X i=6: n X i=7: X i=8: X i=9: n X i=10: X
Some of these comparisons are wasted work! For instance, after iteration i=2, we know from the comparisons we've done that T[3]="a", so there is no point comparing it to "n" in iteration i=3. And we also know that T[4]="n", so there is no point making the same comparison in iteration i=4.
Skipping outer iterations The Knuth-Morris-Pratt idea is, in this sort of situation, after you've invested a lot of work making comparisons in the inner loop of

the code, you know a lot about what's in the text. Specifically, if you've found a partial match of j characters starting at position i, you know what's in positions T[i]...T[i+j-1]. You can use this knowledge to save work in two ways. First, you can skip some iterations for which no match is possible. Try overlapping the partial match you've found with the new match you want to find:
i=2: n a n i=3: n a n o Here the two placements of the pattern conflict with each other – we know from the i=2 iteration that T[3] and T[4] are "a" and "n", so they can't be the "n" and "a" that the i=3 iteration is looking for. We can keep skipping positions until we find one that doesn't conflict:
i=2: n a n i=4: n a n o Here the two "n"'s coincide. Define the overlap of two strings x and y to be the longest word that's a suffix of x and a prefix of y. Here the overlap of "nan" and "nano" is just "n". (We don't allow the overlap to be all of x or y, so it's not "nan"). In general the value of i we want to skip to is the one corresponding to the largest overlap with the current partial match:
String matching with skipped iterations:
i=0; while (i<n) for (j=0; T[i+j] != " P[j] != " T[i+j]==P[j]; j++) ; if (P[j] == ") found a match; i = i + max(1, j-overlap(P[0..j-1],P[0..m])); Skipping inner iterations The other optimization that can be done is to skip some iterations in the inner loop. Let's look at the same example, in which we skipped from i=2 to i=4:
i=2: n a n i=4: n a n o In this example, the "n" that overlaps has already been tested by the i=2 iteration. There's no need to test it again in the i=4 iteration. In general, if we have a nontrivial overlap with the last partial match, we can avoid testing a number of characters equal to the length of the overlap. This change produces (a version of) the KMP algorithm:
KMP, version 1:
i=0; o=0; while (i<n) for (j=o; T[i+j] != " P[j] != " T[i+j]==P[j]; j++) ; if (P[j] == ") found a match; o = overlap(P[0..j-1],P[0..m]); i = i + max(1, j-o); The only remaining detail is how to compute the overlap function. This is a function only of j, and not of the characters in T[], so we can compute it once in a preprocessing stage before we get to this part of the algorithm. First let's see how fast this algorithm is.
KMP time analysis We still have an outer loop and an inner loop, so it looks like the time might still be O(mn). But we can count it a different way to see that it's actually always less than that. The idea is that every time through the inner loop, we do one comparison T[i+j]==P[j]. We can count the total time of the algorithm by counting how many comparisons we perform. We split the comparisons into two groups: those that return true, and those that return false. If a comparison returns true, we've determined the value of T[i+j]. Then in future iterations, as long as there is a nontrivial overlap involving T[i+j], we'll skip past that overlap and not make a comparison with that position again. So each position of T[] is only involved in one true comparison, and there can be n such comparisons total. On the other hand, there is at most one false comparison per iteration of the outer loop, so there can also only be n of those. As a result we see that this part of the KMP algorithm makes at most 2n comparisons and takes time O(n).
KMP and finite automata If we look just at what happens to j during the algorithm above, it's sort of like a finite automaton. At each step j is set either to j+1 (in the inner loop, after a match) or to the overlap o (after a mismatch). At each step the value of o is just a function of j and doesn't depend on other infor-

mation like the characters in T[]. So we can draw something like an automaton, with arrows connecting values of j and labeled with matches and mismatches.
The difference between this and the automata we are used to is that it has only two arrows out of each circle, instead of one per character. But we can still simulate it just like any other automaton, by placing a marker on the start state (j=0) and moving it around the arrows. Whenever we get a matching character in T[] we move on to the next character of the text. But whenever we get a mismatch we look at the same character in the next step, except for the case of a mismatch in the state j=0.
So in this example (the same as the one above) the automaton goes through the sequence of states:
j=0 mismatch T[0] != "n" j=0 mismatch T[1] != "n" j=0 match T[2] == "n" j=1 match T[3] == "a" j=2 match T[4] == "n" j=3 mismatch T[5] != "o" j=1 match T[5] == "a" j=2 match T[6] == "n" j=3 match T[7] == "o" j=4 found match j=0 mismatch T[8] != "n" j=0 mismatch T[9] != "n" j=0 match T[10] == "n" j=1 mismatch T[11] != "a" j=0 mismatch T[11] != "n" This is essentially the same sequence of comparisons done by the KMP pseudocode above. So this automaton provides an equivalent definition of the KMP algorithm. As one student pointed out in lecture, the one transition in this automaton that may not be clear is the one from j=4 to j=0. In general, there should be a transition from j=m to some smaller value of j, which should happen on any character (there are no more matches to test before making this transition). If we want to find all occurrences of the pattern, we should be able to find an occurrence even if it overlaps another one. So for instance if the pattern were "nana", we should find both occurrences of it in the text "nanana". So the transition from j=m should go to the next longest position that can match, which is simply j=overlap(pattern,pattern). In this case overlap("nano","nano") is empty (all suffixes of "nano" use the letter "o", and no prefix does) so we go to j=0.
Alternate version of KMP The automaton above can be translated back into pseudo-code, looking a little different from the pseudo-code we saw before but performing the same comparisons.
KMP, version 2:
j = 0; for (i = 0; i < n; i++) for (;;) // loop until break if (T[i] == P[j]) // matches? j++; // yes, move on to next state if (j == m) // maybe that was the last state found a match; j = overlap[j]; break; else if (j == 0) break; // no match in state j=0, give up else j = overlap[j]; // try shorter partial match The code inside each iteration of the outer loop is essentially the same as the function match from the C++ implementation I've made available. One advantage of this version of the code is that it tests characters one by one, rather than performing random access in the T[] array, so (as in the implementation) it can be made to work for stream-based input rather than having to read the whole text into memory first. The overlap[j] array stores the values of overlap(pattern[0..j-1],pattern), which we still need to show how to compute.
Since this algorithm performs the same comparisons as the other version of KMP, it takes the same amount of time, O(n). One way of proving this bound directly is to note, first, that there is one true comparison (in which T[i]==P[j]) per iteration of the outer loop, since we break out of the inner loop when this happens. So there are n of these total. Each of these comparisons results in increasing j by one. Each iteration of the inner loop in which we don't break out of the loop results in executing the statement j=overlap[j], which decreases

j. Since j can only decrease as many times as it's increased, the total number of times this happens is also O(n).
Computing the overlap function Recall that we defined the overlap of two strings x and y to be the longest word that's a suffix of x and a prefix of y. The missing component of the KMP algorithm is a computation of this overlap function: we need to know overlap(P[0..j-1],P) for each value of j>0. Once we've computed these values we can store them in an array and look them up when we need them. To compute these overlap functions, we need to know for strings x and y not just the longest word that's a suffix of x and a prefix of y, but all such words. The key fact to notice here is that if w is a suffix of x and a prefix of y, and it's not the longest such word, then it's also a suffix of overlap(x,y). (This follows simply from the fact that it's a suffix of x that is shorter than overlap(x,y) itself.) So we can list all words that are suffixes of x and prefixes of y by the following loop:
while (x != empty) x = overlap(x,y); output x; Now let's make another definition: say that shorten(x) is the prefix of x with one fewer character. The next simple observation to make is that shorten(overlap(x,y)) is still a prefix of y, but is also a suffix of shorten(x). So we can find overlap(x,y) by adding one more character to some word that's a suffix of shorten(x) and a prefix of y. We can just find all such words using the loop above, and return the first one for which adding one more character produces a valid overlap:
Overlap computation:
z = overlap(shorten(x),y) while (last char of x != y[length(z)]) if (z = empty) return overlap(x,y) = empty else z = overlap(z,y) return overlap(x,y) = z
So this gives us a recursive algorithm for computing the overlap function in general. If we apply this algorithm for x=some prefix of the pattern, and y=the pattern itself, we see that all recursive calls have similar arguments. So if we store each value as we compute it, we can look it up instead of computing it again. (This simple idea of storing results instead of recomputing them is known as dynamic programming; we discussed it somewhat in the first lecture and will see it in more detail next time.) So replacing x by P[0..j-1] and y by P[0..m-1] in the pseudocode above and replacing recursive calls by lookups of previously computed values gives us a routine for the problem we're trying to solve, of computing these particular overlap values. The following pseudocode is taken (with some names changed) from the initialization code of the C++ implementation I've made available. The value in overlap[0] is just a flag to make the rest of the loop simpler. The code inside the for loop is the part that computes each overlap value.
KMP overlap computation:
overlap[0] = -1; for (int i = 0; pattern[i] != ''; i++) overlap[i + 1] = overlap[i] + 1; while (overlap[i + 1] > 0 pattern[i] != pattern[overlap[i + 1] - 1]) overlap[i + 1] = overlap[overlap[i + 1] - 1] + 1; return overlap; Let's finish by analyzing the time taken by this part of the KMP algorithm. The outer loop executes m times. Each iteration of the inner loop decreases the value of the formula overlap[i+1], and this formula's value only increases by one when we move from one iteration of the outer loop to the next. Since the number of decreases is at most the number of increases, the inner loop also has at most m iterations, and the total time for the algorithm is O(m). The entire KMP algorithm consists of this overlap computation followed by the main part of the algorithm in which we scan the text (using the overlap values to speed up the scan). The first part

takes O(m) and the second part takes O(n) time, so the total time is O(m+n).

Chương 2

Object Oriented Programming

View online http://magizbox.com/training/object$_o$riented$_p$rogramming/site/
Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. A feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated (objects have a notion of "this" or "self"). In OOP, computer programs are designed by making them out of objects that interact with one another. There is significant diversity of OOP languages, but the most popular ones are class-based, meaning that objects are instances of classes, which typically also determine their type.
Many of the most widely used programming languages (such as C++, Java, Python etc.) are multi-paradigm programming languages that support object-oriented programming to a greater or lesser degree, typically in combination with imperative, procedural programming. Significant object-oriented languages include Java, C++, C, Python, PHP, Ruby, Perl, Delphi, Objective-C, Swift, Scala, Common Lisp, and Smalltalk.

2.1 OOP

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which are data structures that contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. A distinguishing feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated (objects have a notion of "this" or "self"). In OO programming, computer programs are designed by making them out of objects that interact with one another.[1][2] There is significant diversity in object-oriented programming, but most popular languages are class-based, meaning that objects are instances of classes, which typically also determines their type. 1. A First Look
Procedural vs Object Oriented 1
Procedural Approach

Focus is on procedures All data is shared: no protection More difficult to modify Hard to manage complexity Advantages of Object Orientation
People think in terms of object OO models map to reality OO models are: Easy to develop Easy to understand. 2. Principles encapsulation, inheritance, abstraction, polymorphism 2
Fundamental Principles of OOP In order for a programming language to be object-oriented, it has to enable working with classes and objects as well as the implementation and use of the fundamental object-oriented principles and concepts: inheritance, abstraction, encapsulation and polymorphism.
2.1 Encapsulation 3 4 5
Encapsulation is the packing of data and functions into a single component. The features of encapsulation are supported using classes in most object-oriented programming languages, although other alternatives also exist. It allows selective hiding of properties and methods in an object by building an impenetrable wall to protect the code from accidental corruption.
What it do? We will learn to hide unnecessary details in our classes and provide a clear and simple interface for working with them.
Example: A popular example you'll hear for encapsulation is driving a car. Do you need to know exactly how every aspect of a car works (engine, carburettor, alternator, and so on)? No - you need to know how to use the steering wheel, brakes, accelerator, and so on.
2.2 Inheritance 6 7
Inheritance is when an object or class is based on another object (prototypal inheritance) or class (class-based inheritance), using the same implementation (inheriting from an object or class) specifying implementation to maintain the same behavior (realizing an interface; inheriting behavior).
inherit everything, add data or functionality, override functions, super
What it do? We will explain how class hierarchies improve code readability and enable the reuse of functionality.
Example: A real-world example of inheritance is genetic inheritance. We all receive genes from both our parents that then define who we are. We share qualities of both our parents, and yet at the same time are different from them.
Example: we might classify different kinds of vehicles according to the inheritance hierarchy. Moving down the hierarchy, each kind of vehicle is both more specialized than its parent (and all of its ancestors) and more general than its children (and all of its descendants). A wheeled vehicle inherits properties common to all vehicles (it holds one or more people and carries them from place to place) but has an additional property that makes it more specialized (it has wheels). A car inherits properties common to all wheeled vehicles, but has additional, more specialized properties (four wheels, an engine, a body, and so forth). The inheritance relationship can be viewed as an is-a relationship. In this relationship, the objects become more specialized the lower in the hierarchy you go.
Look at the image above you will get a point.8 Yes, the derived class can access base class properties and still the derived class has its own properties.
2.3 Abstraction
In computer science, abstraction is a technique for managing complexity of computer systems. It works by establishing a level of complexity on which a person interacts with the system, suppressing the more complex details below the current level. The programmer works with an idealized interface (usually well

defined) and can add additional levels of functionality that would otherwise be too complex to handle.
What it do? We will learn how to work through abstractions: to deal with objects considering their important characteristics and ignore all other details.
Example: You'll never buy a "device", but always buy something more specific : iPhone, Samsung Galaxy, Nokia 3310... Here, iPhone, Samsung Galaxy and Nokia 3310 are concrete things, device is abstract.
2.4 Polymorphism 9
Polymorphism is the provision of a single interface to entities of different types. A polymorphic type is one whose operations can also be applied to values of some other type, or types.
What it do? We will explain how to work in the same manner with different objects, which define a specific implementation of some abstract behavior.
Example: All animal can speak, but dogs woof, cats meow, and ducks quack
There are two types of polymorphism
Overloading (compile time polymorphism): methods have the same name but different parameters. Overriding (run time polymorphism): the implementation given in base class is replaced with that in sub class.
Example 10: Let us Consider Car example for discussing the polymorphism. Take any brand like Ford, Honda, Toyota, BMW, Benz etc., Everything is of type Car. But each have their own advanced features and more advanced technology involved in their move behavior.
3. Concepts Learn Object Oriented Programming though Mario Game
[embed]https://www.youtube.com/watch?v=HBbzYKMfx5Y[/embed]
How Mario get 1up
3.1. Object 11
Objects are key to understanding object-oriented technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle. In mario world, Mario is an object.
Goomba is an object. Koopa is also an object. Even a coin and a pile are objects
Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in fields (variables in some programming languages) and exposes its behavior through methods (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication. Hiding internal state and requiring all interaction to be performed through an object's methods is known as data encapsulation — a fundamental principle of object-oriented programming. In Mario world, Mario has some fields like position (which indicate where Mario stands), state (which indicate whether Mario alive), and some methods like walk , fire or jump.
Goomba has some fields like position (which indicate where Goomba stands), state (which indicate whether Goomba die), and direction (which indicate the direction Goomba moves). Goomba has move method, and jumped$_o$nmethod(whichoccurswhenitisjumpedonby
Mario Objects, real scene
3.2 Class 12
In the real world, you'll often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that

your bicycle is an instance of the class of objects known as bicycles. A class is the blueprint from which individual objects are created.
In Mario world, each coin object come from Coin class, and every Koomba come from Koomba class
3.3. Inheritance 13
Inheritance is a mechanism in OOP to design two or more entities that are different but share many common features.
Feature common to all classes are defined in the superclass The classes that inherit common features from the superclass are called subclasses In Mario World, Goomba and Koopa is in
AND MANY, MANY MORE
3.4. Association, Aggregation and Composition
13
Association:
Whenever two objects are related with each other the relationshiop is called association between object
Aggregation:
Aggregation is specialized from of association. In aggregation objects have their own life-cycle but there is ownership and child object can not belongs to another parent object. But this is only an ownership not the life-cycle control of child control through parent object.
Example: Student and Teacher, Person and address
Composition
Composition is again specialize form of aggregation and we can call this as ‘life and death“ relationship. It is a strong type of aggregation. Child object does not have their life-cycle and if parent object is deleted, all child object will also be deleted.
Example: House and room
3.5 Polymorphism 13
Polymorphism indicates the meaning of "many forms"
Polynorphism present a method that can have many definitions. Polymorphism is related to "over loading" and "over ridding".
Overloading indicates a method can have different definitions by defining different type of parameters.
[code] getPrice(): void getPrice(string name): void [/code]
3.6 Abstraction 13
Abstraction is the process of modelling only relevant features
Hide unnecessary details which are irrelevant for current purpose. Reduces complexity and aids understanding.
Abstraction provides the freedom to defer implementation decisions by avoiding commitments to details.
3.7 Interface 13
An interface is a contract consisting of group of related function prototypes whose usage is defined but whose implementation is not:
An interface definition specifies the interface’s member functions, called methods, their return types, the number and types of parameters and what they must do.
These is no implementation associated with an interface.
4. Coupling and Cohesion 13

4.1 Coupling Coupling defines how dependent one object on another object (that is uses).
Coupling is a measure of strength of connection between any two system components. The more any one components knows about other components, the tighter (worse) the coupling is between those components.
4.2 Cohesion Cohesion defines how narrowly defined an object is. Functional cohesion refers measures how strongly objects are related.
Cohesion is a measure of how logically related the parts of an individual components are to each other, and to the overall components. The more logically related the parts of components are to each other higher (better) the cohesion of that components.
4.3 Object Oriented Design Low coupling and tight cohesion is good object oriented design.
Challenge Object Task 1: With boiler plate code, make an gif image (32x32) Mario fire ball and jump to get coins
5. NEXT Design Principles Design Patterns

2.2 UML

The Unified Modeling Language (UML) is a general-purpose, developmental, modeling language in the field of software engineering, that is intended to provide a standard way to visualize the design of a system.
http://www.yuml.me/ Use UML with IntellIJ: UML Designer Architecture 1
Design of a system consists of classes, interfaces and collaboration. UML provides class diagram, object diagram to support this. Implementation defines the components assembled together to make a complete physical system. UML component diagram is used to support implementation perspective. Process defines the flow of the system. So the same elements as used in Design are also used to support this perspective. Deployment represents the physical nodes of the system that forms the hardware. UML deployment diagram is used to support this perspective. Modelling Types 2
Diagrams Usecase Diagram 3 4 5
A use case diagram at its simplest is a representation of a user's interaction with the system that shows the relationship between the user and the different use cases in which the user is involved.
A use case diagram can identify the different types of users of a system and the different use cases and will often be accompanied by other types of diagrams as well.
Use case diagrams depict:
Use cases. A use case describes a sequence of actions that provide something of measurable value to an actor and is drawn as a horizontal ellipse. (example) Actors. An actor is a person, organization, or external system that plays a role in one or more interactions with your system. Actors are drawn as stick figures. (example) Associations. Associations between actors and use cases are indicated in use case diagrams by solid lines. An association exists whenever an actor is involved with an interaction described by a use case. Associations are modeled as lines connecting use cases and actors to one another, with an optional arrowhead on one end of the line. The arrowhead is often used to indicating the direction of the initial invocation of the relationship or to indicate the primary actor within

the use case. The arrowheads are typically confused with data flow and as a result I avoid their use. (example) Extend: Extend is a directed relationship that specifies how and when the behavior defined in usually supplementary (optional) extending use case can be inserted into the behavior defined in the extended use case. (example) Include is a directed relationship between two use cases which is used to show that behavior of the included use case (the addition) is inserted into the behavior of the including (the base) use case. (example) System boundary boxes (optional). You can draw a rectangle around the use cases, called the system boundary box, to indicates the scope of your system. Anything within the box represents functionality that is in scope and anything outside the box is not. System boundary boxes are rarely used, although on occasion I have used them to identify which use cases will be delivered in each major release of a system. (example) Packages (optional). Packages are UML constructs that enable you to organize model elements (such as use cases) into groups. Packages are depicted as file folders and can be used on any of the UML diagrams, including both use case diagrams and class diagrams. I use packages only when my diagrams become unwieldy, which generally implies they cannot be printed on a single page, to organize a large diagram into smaller ones. (example) Class Diagram 6

In software engineering, a class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

3.3.1 UML Association 9 10

Association

Association is reference based relationship between two classes. Here a class A holds a class level reference to class B. Association can be represented by a line between these classes with an arrow indicating the navigation direction. In case arrow is on the both sides, association has bidirectional navigation.

Aggregation

Aggregation (shared aggregation) is a "weak" form of aggregation when part instance is independent of the composite:

the same (shared) part could be included in several composites, and if composite is deleted, shared parts may still exist. Shared aggregation is shown as binary association decorated with a hollow diamond as a terminal adornment at the aggregate end of the association line. The diamond should be noticeably smaller than the diamond notation for N-ary associations. Shared aggregation is shown as binary association decorated with a hollow diamond.

Composition

Composition (composite aggregation) is a "strong" form of aggregation. Composition requirements/features listed in UML specification are:

it is a whole/part relationship, it is binary association part could be included in at most one composite (whole) at a time, and if a composite (whole) is deleted, all of its composite parts are "normally" deleted with it. Note, that UML does not define how, when and specific order in which parts of the composite are created. Also, in some cases a part can be removed from a composite before the composite is deleted, and so is not necessarily deleted as part of the composite.

Aggregation vs Composition

11

Sequence Diagram 7

A Sequence diagram is an interaction diagram that shows how processes operate with one another and in what order. It is a construct of a Message Sequence Chart.
A sequence diagram shows object interactions arranged in time sequence.
It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. Sequence diagrams are typically associated with use case realizations in the Logical View of the system under development. Sequence diagrams are sometimes called event diagrams or event scenarios.
Activity Diagram 8
Activity diagrams are graphical representations of workflows of stepwise activities and actions with support for choice, iteration and concurrency. In the Unified Modeling Language, activity diagrams are intended to model both computational and organizational processes (i.e. workflows). Activity diagrams show the overall flow of control.
UML - Architecture
UML - Modeling Types
UML - Use Case Diagrams
Use Case Diagram
UML Association Between Actor and Use Case
Class diagram
Sequence diagram
Activity diagram
Aggregation
UML Class Diagram: Association, Aggregation and Composition
Lecture Notes on Object-Oriented Programming: Object Oriented Aggregation

2.3 SOLID

SOLID Principles In computer programming, SOLID (single responsibility, open-closed, Liskov substitution, interface segregation and dependency inversion) is a mnemonic acronym introduced by Michael Feathers for the "first five principles" named by Robert C. Martin in the early 2000s that stands for five basic principles of object-oriented programming and design. The intention is that these principles, when applied together, will make it more likely that a programmer will create a system that is easy to maintain and extend over time. The principles of SOLID are guidelines that can be applied while working on software to remove code smells by providing a framework through which the programmer may refactor the software's source code until it is both legible and extensible. It is part of an overall strategy of agile and Adaptive Software Development.
"Dependency Management is an issue that most of us have faced. Whenever we bring up on our screens a nasty batch of tangled legacy code, we are experiencing the results of poor dependency management. Poor dependency management leads to code that is hard to change, fragile, and non-reusable."
Uncle Bob talk about several different design smells in the PPP book, all relating to dependency management. On the other hand, when dependencies are well managed, the code remains flexible, robust, and reusable. So dependency

management, and therefore these principles, are at the foundation of the -ilities that software developers desire.
SRP - Single Responsibility A class should have one, and only one, reason to change.
A class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)
Original Paper
OCP - Open/Closed You should be able to extend a classes behavior, without modifying it.
Software entities ... should be open for extension, but closed for modification."
Original Paper
LSP - Liskov Substitution Derived classes must be substitutable for their base classes.
Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program. See also design by contract.
Original Paper
ISP - Interface Segregation Make fine grained interfaces that are client specific.
Many client-specific interfaces are better than one general-purpose interface.
Original Paper
DIP - Dependency Inversion Depend on abstractions, not on concretions.
One should "depend upon abstractions, not concretions."
Original Paper
References The Principles of OOD

2.4 Design Patterns

Design Patterns
Creational design patterns These design patterns are all about class instantiation. This pattern can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.
Structural design patterns These design patterns are all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.
Behavioral design patterns These design patterns are all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.
Design Pattern QA Examples of GoF Design Patterns in Java's core libraries Dependency Injection vs Factory Pattern What is Inversion of Control? What is so bad about singletons? What is the basic difference between Factory and Abstract Factory Patterns? When would you use the Builder Pattern? What is the difference between Builder Design pattern and Factory Design pattern? How do the Proxy, Decorator, Adapter, and Bridge Patterns differ? Abstract Factory Pattern Creates an instance of several families of classes Intuitive 1
Volkswagen Transparent Factory in Dresden

What is it? 2 The abstract factory pattern provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes.
In normal usage, the client software creates a concrete implementation of the abstract factory and then uses the generic interface of the factory to create the concrete objects that are part of the theme. The client doesn't know (or care) which concrete objects it gets from each of these internal factories, since it uses only the generic interfaces of their products.
This pattern separates the details of implementation of a set of objects from their general usage and relies on object composition, as object creation is implemented in methods exposed in the factory interface.
Design
Example Code
The most interesting factories in the world Abstract factory pattern Observer Pattern Intuitive
Definition 1 The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. It is mainly used to implement distributed event handling systems. The Observer pattern is also a key part in the familiar model–view–controller (MVC) architectural pattern. The observer pattern is implemented in numerous programming libraries and systems, including almost all GUI toolkits. Structure 2
Subject
knows its observers. Any number of Observer objects may observe a subject. provides an interface for attaching and detaching Observer objects Observer
defines an updating interface for objects that should be notified of changes in a subject. ConcreteSubject
stores state of interest to ConcreteObserver objects. sends a notification to its observers when its state changes. ConcreteObserver
maintains a reference to a ConcreteSubject object. stores state that should stay consistent with the subject's. implements the Observer updating interface to keep its state consistent with the subject's. Examples Example 1: Blog Manager Application
In this application, each user is an Observer, each blog is a Subject. When a blog post a new article (state change), user get an update. When users get update, they update their articles.
[code lang="java"] Blog sportBlog = new Blog("SPORT"); User user1 = new User("Fan1"); User user2 = new User("Fan2");
sportBlog.attach(user1); sportBlog.attach(user2);
sportBlog.post(new Article("football")); sportBlog.post(new Article("swimming"));
user1.getArticles(); user2.getArticles();
sportBlog.detach(user1);‘ [/code]
Real Implementations Broadcast Receiver 3 4 on Android
More Articles http://javapapers.com/design-patterns/observer-design-pattern/
Comparison Observer/Observable pattern vs Publisher/Subscriber pattern 5 Observer/Observable pattern is mostly implemented in a synchronous way, i.e. the observable calls the appropriate method of all its observers when some event occurs. The Publisher/Subscriber pattern is mostly implemented in an asynchronous way (using message queue). In the Observer/Observable pattern, the

observers are aware of the observable. Whereas, in Publisher/Subscriber, publishers and subscribers don't need to know each other. They simply communicate with the help of message queues. Observer pattern
Broadcast Receiver
Design Patterns: Elements of Reusable Object-Oriented Software
Which design patterns are used on Android?
stackoverflow, Difference between Observer, Pub/Sub, and Data Binding

Chương 3

Database

View online http://magizbox.com/training/computer$_s$*cience*/*site*/*database*/

3.1 Introduction

Relational DBMS: Oracle, MySQL, SQLite
Key-value Stores: Redis, Memcached
Document stores: MongoDB
Graph: Neo4j
Wide column stores: Cassandra, HBase
Design and Modeling (a.k.a Data Definition) 1.1 Schema A database schema of a database system is its structure described in a formal language supported by the database management system (DBMS) and refers to the organization of data as a blueprint of how a database is constructed (divided into database tables in the case of Relational Databases). The formal definition of database schema is a set of formulas (sentences) called integrity constraints imposed on a database. These integrity constraints ensure compatibility between parts of the schema. All constraints are expressible in the same language. A database can be considered a structure in realization of the database language. The states of a created conceptual schema are transformed into an explicit mapping, the database schema. This describes how real world entities are modeled in the database.
1.1.1 Type In computer science and computer programming, a data type or simply type is a classification identifying one of various types of data, such as real, integer or Boolean, that determines the possible values for that type; the operations that can be done on values of that type; the meaning of the data; and the way values of that type can be stored.
TEXT, INT, ENUM, TIMESTAMP
1.2 Cardinality (a.k.a Relationship) Foreign key, Primary key
1.2 Indexing A database index is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain the index data structure. Indexes are used to quickly locate data without having to search every row in a database table every time a database table is accessed. Indexes can be created using one or more columns

of a database table, providing the basis for both rapid random lookups and efficient access of ordered records. Why Indexing is important?
Indexing in MySQL
CREATE INDEX NameIndex ON Employee (name) SELECT * FROM Employee WHERE name = 'Ashish' 2. Data Manipulation Create - Read - Update - Delete Create or add new entries Read, retrieve, search, or view existing entries * Update or edit existing entries * Delete/deactivate existing entries /* create */
CREATE TABLE Guests (id INT(6) UNSIGNED AUTO$_INCREMENTPRIMARYKEY, firstnameVAR$
$create(insert)*/INSERTINTOGuests(firstname, lastname, email)VALUES('John','Doe','john@exam$
$read*/SELECT*FROMGuestsWHEREid = 1/*update*/UPDATEGuestsSETlastname ='$
$Doe'WHEREid = 1/*delete*/DELETEFROMGuestsWHEREid = 1$‘3.$DataRetrieveTransaction3.1Da$
Get user id, user name and number of post of this user
SELECT user.id, user.name, COUNT(post.*) AS posts FROM user LEFT OUTER JOIN post ON post.owner$_id = user.idGROUPBYuser.id$‘$Selectuserwhoonlyorderonetime.$
SELECT name, COUNT(name) AS c FROM orders GROUP BY name HAVING c = 1; Calculate the longest period (in days) that the company has gone without a hiring or firing anyone.
SELECT x.date, MIN(y.date) y$_date, DATEDIFF(MIN(y.date), x.date)daysFROM(SELECThiredatedat$
$x.dateGROUPBYx.dateORDERBYdaysDESCLIMIT1; DataRetrieveAPI$
API Description get get single item Get dog by id
Dog.get(1) find find items
@see collection.find()
Find dog name "Max"
Dog.find("name": "Max") sort sort items
@see cursor.sort
Get 10 older dogs
Dog.find().sort("age", limit: 10) aggregate sum, min, max items
@see collection.aggregate
Get sum of dogs' age
Dog.find().aggregate("sum$_age$" : sum: "age") 3.2 Transaction A transaction symbolizes a unit of work performed within a database management system (or similar system) against a database, and treated in a coherent and reliable way independent of other transactions. A transaction generally represents any change in database. Example: Transfer $900 from Account$
Bob to Alice
start transaction select balance from Account where Account$_Number =' Bob'; selectbalancefromAccountwhe$
$Alice'; updateAccountsetbalance = balance-900hereAccount_Number =' Bob'; updateAccountsetbalance =$
$balance+900hereAccount_Number =' Alice'; commit; //ifallsqlqueriessuccedrollback; //ifanyofSqlqueries$
In computer science, ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee that database transactions are processed reliably. In the context of databases, a single logical operation on the data is called a transaction.
For example, a transfer of funds from one bank account to another, even involving multiple changes such as debiting one account and crediting another, is a single transaction. ![][16]
4. Backup and Restore Sometimes it is desired to bring a database back to a previous state (for many reasons, e.g., cases when the database is found corrupted due to a software error, or if it has been updated with erroneous data). To achieve this a backup operation is done occasionally or continuously, where each desired database state (i.e., the values of its data and their embedding in

database's data structures) is kept within dedicated backup files (many techniques exist to do this effectively). When this state is needed, i.e., when it is decided by a database administrator to bring the database back to this state (e.g., by specifying this state by a desired point in time when the database was in this state), these files are utilized to restore that state.
5. Migration In software engineering, schema migration (also database migration, database change management) refers to the management of incremental, reversible changes to relational database schemas. A schema migration is performed on a database whenever it is necessary to update or revert that database's schema to some newer or older version. Example: Android Migration by droid-migrate
droid-migrate init -d my$_{d}atabasedroid-migrategenerateupdroid-migrategeneratedownExample$: $DatabaseSeedingwithLaravel$
6. Active record pattern | Object-Relational Mapping (ORM) Object-relational mapping in computer science is a programming technique for converting data between incompatible type systems in object-oriented programming languages. This creates, in effect, a "virtual object database" that can be used from within the programming language. There are both free and commercial packages available that perform object-relational mapping, although some programmers opt to create their own ORM tools.
Example
php
$employee = newEmployee()$;employee->setName("Joe"); $employee- > save()$; $Android$
public class User @DatabaseField(id = true) String username; @DatabaseField String password; @DatabaseField String email; @DatabaseField String alias; public User() Implementations
Android: [ormlite-android] PHP: [Eloquent]

3.2 SQL

SQL SELECT * FROM WORLD
INSERT INTO
SELECT * FROM girls

3.3 MySQL

MySQL
MySQL is an open-source relational database management system (RDBMS); in July 2013, it was the world's second most widely used RDBMS, and the most widely used open-source client–server model RDBMS. It is named after co-founder Michael Widenius's daughter, My. The SQL abbreviation stands for Structured Query Language. The MySQL development project has made its source code available under the terms of the GNU General Public License, as well as under a variety of proprietary agreements. MySQL was owned and sponsored by a single for-profit firm, the Swedish company MySQL AB, now owned by Oracle Corporation. For proprietary use, several paid editions are available, and offer additional functionality.

MySQL: Docker Docker Run docker pull mysql docker run -d -p 3306:3306 –env MYSQL$_{R}OOT_{P}ASSWORD = docker - -envMYSQL_{D}ATABASE = docker - -envMYSQL_{U}SER = docker - -envMYSQL_{P}ASSWORD = docker mysqlNote : OnWindows, viewyour0.0.0.0IPbyrunningbelowcommandline(oryoucanturnonKitema
Docker Compoose Step 1: Clone Docker Project
git clone https://github.com/magizbox/docker-mysql.git mv docker-mysql mysql
Step 2: Docker Compose
version: "2"
services: mysql: build: ./mysql/. ports: - 3306:3306 environment: - MYSQL$_{R}OOT_{P}ASSWORD = docker−MYSQL_{D}ATABASE = docker−MYSQL_{U}SER = docker−MYSQL_{P}ASSWORD = dockervolumes : −./data/mysql : /var/lib/mysqlDockerFoldermysql/configmy.cnfDockerfileVerifydoc machinelsNAMEACTIVEDRIVERSTATEURLSWARMdefault∗virtualboxRunningtcp : //192.168.99.100 : 2376Youcanaddphpmyadmintoseemysqlworks
phpmyadmin: image: phpmyadmin/phpmyadmin links: - mysql environment: - PMA$_{A}RBITRARY = 1ports : −80 : 80Seeitworks
Go to localhost Login with Server=mysql, Username=docker, Password=docker

3.4 Redis

Redis is an open source (BSD licensed), in-memory data structure store, used as database, cache and message broker. 1
It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs and geospatial indexes with radius queries. Redis has built-in replication, Lua scripting, LRU eviction, transactions and different levels of on-disk persistence, and provides high availability via Redis Sentinel and automatic partitioning with Redis Cluster.
Redis: Client Python Client pipy/redis
Installation
pip install redis Usage
import redis r = redis.StrictRedis(host='localhost', port=6379, db=0) r.set('foo', 'bar') -> True
r.get('foo') -> 'bar'
r.delete('foo')
after delete r.get('foo') -> None Java Client https://redislabs.com/redis-java
Redis: Docker Docker Run docker run -d -p 6379:6379 redis Docker Compose
version: "2"
services: redis: image: redis ports: - 6379:6379 Redis.io

3.5 MongoDB

MongoDB is an open-source document database that provides high performance, high availability, and automatic scaling.
MongoDB provides high performance data persistence. In particular,
Support for embedded data models reduces I/O activity on database system. Indexes support faster queries and can include keys from embedded documents and arrays. MongoDB is 1 in the Document Store according to db-engines

Client Mongo Shell The mongo shell is an interactive JavaScript interface to MongoDB and is a component of the MongoDB package. You can use the mongo shell to query and update data as well as perform administrative operations.
Start Mongo
Once you have installed and have started MongoDB, connect the mongo shell to your running MongoDB instance. Ensure that MongoDB is running before attempting to launch the mongo shell.
mongo Interact with mongo via shell
Show list database > show dbs
Create or use a database > use <database$_name$ >> $usetestexample$
List collection > show collections
Create or use a collection > db.<collection$_name$ >> $db.new_collectionexample$
Read document > db.new$_c$$ollection.find$()
Insert new document > db.new$_c$$ollection.insertOne$("$a$" : "$b$")
Update document > db.new$_c$$ollection.update$("$a$" : "$b$", set: "a": "bcd")
Remove document > db.new$_c$$ollection.remove$("$a$" : "$b$")$PyMongo-PythonClientPyMongoisaPythondistr$
Installation We recommend using pip to install pymongo on all platforms:
pip install pymongo Usage import pymongo create connection client = pymongo.MongoClient('127.0.0.1', 27017) -> MongoClient(host=['127.0.0.1:27017'], document$_c$$lass = dict, tz_aware = False, connect = True$)
create database db = client.db$_test$$- > Database(MongoClient(host = ['127.0.0.1 :$ 27017'], $document_class = dict, tz_aware = False, connect = True), u'db_test')$
create collection (collection is the same with table in SQL) collection = db.new$_c$$ollection$
insert document to collection (document is the same with rows in SQL) db.collection.insert$_one$("c" : "d")$- ><$ $pymongo.results.InsertOneResultat0x7f7eab3c9f00 >$
read document of collection db.new$_c$$ollection.find_one$("$c$" : "$d$")$- > u'_id' : ObjectId('589a8195f23e627a973c$
update documents of collection db.new$_c$$ollection.update$("$c$" : "$d$","set": "c": "def") -> u'n': 1, u'nModified': 1, u'ok': 1.0, 'updatedExisting': True
remove document of collection db.new$_c$$ollection.remove$("$c$" : "$def$")$- > u'n' : 1, u'ok' : 1.0DockerDockerRu$
docker run -p 27017:27017 mongo:latest

Chương 4

Hệ điều hành

Những phần mềm không thể thiếu
* Trình duyệt Google Chrome (với các extensions Scihub, Mendeley Desktop, Adblock) * Adblock extension * Terminal (Oh-my-zsh) * IDE Pycharm để code python * Quản lý phiên bản code Git * Bộ gõ ibus-unikey trong Ubuntu hoặc unikey (Windows) (Ctrl-Space để chuyển đổi ngôn ngữ) * CUDA (lập trình trên GPU)
Xem thông tin hệ thống
Phiên bản 'ubuntu 16.04'

```
sudo apt-get install sysstat
```

Xem hoạt động (
"' mpstat -A "'
CPU của mình có bao nhiều core, bao nhiêu siblibngs
"' cat /proc/cpuinfo
processor : 23 vendor$_id$: $GenuineIntelcpufamily$: 6$model$: 62$modelname$: $Intel(R)Xeon(R)CPUE5-2430v2@2.50GHzstepping$: 4$microcode$: 0$x428cpuMHz$: 1599.707$cachesize$: 15360$KBphysicalid$: 1$siblings$: 12$coreid$: 5$cpucores$: 6$apicid$: 43$initialapicid$: 43fpu : $yesfpu_exception$: $yescpuidlevel$: 13wp : $yesflags$: $fpuvmedepsetscmsrpaemcecx8apicsepmtrrpgemcacmovpatpse36clflushdtsacpimmxfxsrssesse
5005.20$clflushsize$: 64$cache_alignment$: 64$addresssizes$: 46$bitsphysical, 48bitsvirtualpowermanagement$:
"'
Kết quả cho thấy cpu của 6 core và 12 siblings
Xem chương trình nào tốn ram
ps aux | awk 'print 2,4, 11'|$sort - k2rn|head - n20$
https://www.garron.me/en/go2linux/how-find-which-process-eating-ram-memory-linux.html

Chương 5

Ubuntu

Chuyện terminal
Terminal là một câu chuyện muôn thuở của bất kì ông coder nào thích customize, đẹp, tiện (và bug kinh hoàng). Hiện tại mình đang thấy combo này khá ổn Terminal (Ubuntu) (Color: Black on white, Build-in schemes: Tango) + zsh + oh-my-zsh (fishy-custom theme). Những features hay ho
* Làm việc tốt trên cả Terminal (white background) và embedded terminal của Pycharm (black background) * Hiển thị folder dạng ngắn (chỉ ký tự đầu tiên) * Hiển thị brach của git ở bên phải
![Imgur](https://i.imgur.com/q53vQdH.png)
Chuyện bộ gõ
Làm sao để khởi động lại ibus, thỉnh thoảng lại chết bất đắc kì tử [1]"'$ibus - daemonibusrestart$"'
Chuyện lỗi login loop
Phiên bản: 'ubuntu 16.04'
27/12/2017: Lại dính lỗi không thể login. Lần này thì lại phải xóa bạn KDE đi. Kể cũng hơn buồn. Nhưng nhất quyết phải enable được tính năng Windows Spreading (hay đại loại thế). Hóa ra khi ubuntu bị lỗi không có lancher hay toolbar là do bạn unity plugin chưa được enable. Oài. Sao người hiền lành như mình suốt ngày bị mấy lỗi vớ vẩn thế không biết.
20/11/2017: Hôm nay đen thật, dính lỗi login loop. Fix mãi mới được. Thôi cũng kệ. Cảm giác bạn KDE này đỡ bị lỗi ibus-unikey hơn bạn GNOME. Hôm nay cũng đổi bạn zsh theme. Chọn mãi chẳng được bạn nào ổn ổn, nhưng không thể chịu được kiểu suggest lỗi nữa rồi. Đôi khi thấy default vẫn là tốt nhất.
21/11/2017: Sau một ngày trải nghiệm KDE, cảm giác giao diện mượt hơn GNOME. Khi overview windows với nhiều màn hình tốt và trực quan hơn. Đặc biệt là không bị lỗi ibus nữa. Đổi terminal cũng cảm giác ổn ổn. Không bị lỗi suggest nữa.
[1] : $https://askubuntu.com/questions/389903/ibus-doesnt-seem-to-restart$
Cần cài đặt ngay
"' sudo apt install gnome-tweak-tool "'
Cài đặt Workspace

Chương 6

Networking

View online http://magizbox.com/training/computer$_s$*cience*/*site*/*networking*/
TCP/IP TCP/IP is the protocol that has run the Internet for 30 years.
How TCP/IP works
Read More
Happy 30th Anniversary, Internet and TCP/IP!!! P2P Peer-to-peer (P2P) computing or networking is a distributed application architecture that partitions tasks or workloads between peers. Peers are equally privileged, equipotent participants in the application. They are said to form a peer-to-peer network of nodes.
Peers make a portion of their resources, such as processing power, disk storage or network bandwidth, directly available to other network participants, without the need for central coordination by servers or stable hosts.[1] Peers are both suppliers and consumers of resources, in contrast to the traditional client-server model in which the consumption and supply of resources is divided. Emerging collaborative P2P systems are going beyond the era of peers doing similar things while sharing resources, and are looking for diverse peers that can bring in unique resources and capabilities to a virtual community thereby empowering it to engage in greater tasks beyond those that can be accomplished by individual peers, yet that are beneficial to all the peers.
bridge vs NAT When you create a new virtual machine, you have one of many options when it comes to choosing your network connectivity. Two common options are to use either bridged networking or network address translation (NAT). So, what exactly does that look like? Take a look at the figure below.
NAT: In this diagram, the vertical line next to the firewall represents the production network and you can see that 192.168.1.1 is the IP address of the company's firewall that connects them to the Internet. There is also a virtual host with three virtual machines running inside it. The big red circle represents the virtual adapter to which NAT-based virtual machines connect (172.16.1.1). You can see that there are two such virtual machines with IP addresses of 172.16.1.2 and 172.16.1.3. When you configure a virtual machine as using NAT, it doesn't see the production network directly. In fact, all traffic coming from the virtual machine will share the VM host's IP address. Behind the scenes, traffic from the virtual machines is routed on the virtual host and sent out via the host's physical adapter and, eventually, to the Internet.
bridge: The third virtual machine (192.168.1.3) is configured in "bridged" mode

which basically means that the virtual network adapter in that virtual machine is bridged to the production network and that virtual machine operates as if it exists directly on the production network. In fact, this virtual machine won't even be able to see the two NAT-based virtual machines since they're on different networks.
Read more: NAT vs. bridged network: A simple diagram

Chương 7

UX - UI

View online http://magizbox.com/training/computer$_s$*cience*/*site*/*ux*/
1. Design Principles UI Design Do's and Don'ts Android Design Principles
2. Design Trends 2.1 Material Design 1 components
We challenged ourselves to create a visual language for our users that synthesizes the classic principles of good design with the innovation and possibility of technology and science. This is material design. This spec is a living document that will be updated as we continue to develop the tenets and specifics of material design.
Tools
materialpalette.com Icon: fa2png UI Components
Data Binding Transclusion Directive - Fragments
Messaging Intent Android 1
Intents are asynchronous messages which allow application components to request functionality from other Android components. Intents allow you to interact with components from the same applications as well as with components contributed by other applications. For example, an activity can start an external activity for taking a picture.
Intents are objects of the android.content.Intent type. Your code can send them to the Android system defining the components you are targeting. For example, via the startActivity() method you can define that the intent should be used to start an activity.
An intent can contain data via a Bundle. This data can be used by the receiving component.
Style Theme Android Development: Explaining Styles and Themes, https://m.youtube.com//watch?v=MXp
Responsive Design Support Multi Screen 2
Intent Android
Support Multi Screen

Chương 8

Service-Oriented Architecture

View online http://magizbox.com/training/computer$_s$cience/site/software$_a$rchitecture/
A service-oriented architecture (SOA) is an architectural pattern in computer software design in which application components provide services to other components via a communications protocol, typically over a network. The principles of service-orientation are independent of any vendor, product or technology. 2
Generally accepted view 1 Boundaries are explicit Services are autonomous Services share schema and contract, not class Service compatibility is based on policy Microservices In computing, microservices is a software architecture style in which complex applications are composed of small, independent processes communicating with each other using language-agnostic APIs. These services are small building blocks, highly decoupled and focussed on doing a small task, facilitating a modular approach to system-building. One of concepts which integrates microservices as a software architecture style is dew computing. 1
Properties 2 Each running in its own process Communicating with lightweight mechanisms, often an HTTP resource API Build around business capabilities Independently deployable fully automated deployment Maybe in a different programming language and use different data storage technologies Monolith vs Microservice Monolith Microservice Simplicity Partial Deployment Consistency Availability Inter-module refactoring Preserve Modularity Multiple Platforms Benefits 4 Their small size enables developers to be most productive. It's easy to comprehend and test each service. You can correctly handle failure of any dependent service. They reduce impact of correlated failures. Web Service RESTful API
REST Client Sense (Beta)
A JSON aware developer console to ElasticSearch.
API Document and Client Generator http://swagger.io/swagger-editor/
API Client CRUD Pet
API Client Method URL Body Return Body Method GET /pets [Pet] PetApi.list()
POST /pets/ Pet Pet PetApi.create(pet) GET /pets/pet$_i$d$PetPetApi.get(pet_id)PUT/pets/pet_idPetPetPetA$
CRUD Store
GET /stores StoreApi.list() ... ... Relationships
Many to many

Example [https://api.facebook.com/method/links.getStats?urls=Microservices
Slide 11/42, Micro-servies
Martin Fowler, Microservices, youtube
Rick E. Osowski, Microservices in action, Part 1: Introduction to microservices, IBM developerworks

Chương 9

License

View online http://magizbox.com/training/computer$_s$*cience*/*site*/*licenses*/
Licenses More Licenses
More Open Source Licenses Choose A License Top 20 Open Source Licenses

Chương 10

Semantic Web

View online http://magizbox.com/training/semantic$_web/site/$
The Semantic Web is an extension of the Web through standards by the World Wide Web Consortium (W3C). The standards promote common data formats and exchange protocols on the Web, most fundamentally the Resource Description Framework (RDF).
According to the W3C, "The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries". The term was coined by Tim Berners-Lee for a web of data that can be processed by machines. While its critics have questioned its feasibility, proponents argue that applications in industry, biology and human sciences research have already proven the validity of the original concept.

10.1 Web 3.0

Tim Berners-Lee has described the semantic web as a component of "Web 3.0".
People keep asking what Web 3.0 is. I think maybe when you've got an overlay of scalable vector graphics – everything rippling and folding and looking misty – on Web 2.0 and access to a semantic Web integrated across a huge space of data, you'll have access to an unbelievable data resource ...
—Tim Berners-Lee, 2006
"Semantic Web" is sometimes used as a synonym for "Web 3.0", though the definition of each term varies.

10.2 RDF

10.3 SPARQL

SPARQL (pronounced "sparkle", a recursive acronym for SPARQL Protocol and RDF Query Language) is an RDF query language, that is, a semantic query language for databases, able to retrieve and manipulate data stored in Resource Description Framework (RDF) format. It was made a standard by the RDF Data Access Working Group (DAWG) of the World Wide Web Consortium, and is recognized as one of the key technologies of the semantic web. On 15 January

2008, SPARQL 1.0 became an official W3C Recommendation, and SPARQL 1.1 in March, 2013.
SPARQL allows for a query to consist of triple patterns, conjunctions, disjunctions, and optional patterns.
A SPARQL query
Anatomy of a query
SPARQL has four query forms. These query forms use the solutions from pattern matching to form result sets or RDF graphs. The query forms are:
SELECT Returns all, or a subset of, the variables bound in a query pattern match. CONSTRUCT Returns an RDF graph constructed by substituting variables in a set of triple templates. ASK Returns a boolean indicating whether a query pattern matches or not. DESCRIBE Returns an RDF graph that describes the resources found. Example
Query Result Data filename: ex008.rq
PREFIX ab: <http://learningsparql.com/ns/addressbook>
SELECT ?person WHERE ?person ab:homeTel "(229) 276-5135" Offline query example GET CRAIG EMAILS PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns> PREFIX owl: <http://www.w3.org/2002/07/owl> PREFIX xsd: <http://www.w3.org/2001/XMLSchema> PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema> PREFIX : <http://www.semanticweb.org/lananh/ontologies/2016/10/untitled-ontology-3>
SELECT ?craigEmail WHERE :craig :email ?craigEmail . Online query example PREFIX ab: <http://learningsparql.com/ns/addressbook>
SELECT ?craigEmail WHERE ab:craig ab:email ?craigEmail . Query in dbpedia.org Example
SELECT * WHERE ?a ?b ?c . LIMIT 20

Tài liệu tham khảo

Ghi chú