Data Structures

**Introduction**

Welcome to the third series of notes in our journey through Java. This will pretty much wrap up the intermediate details about Java, as well as provide you with useful knowledge applicable all programming languages. Data structures are conceptual structures that help us visualize complex algorithms for efficiency. In other words, we study many, many different solutions to different problems whilst maximizing efficiency. To begin, I will be introducing you to big O, recursion, the stack, and sorting algorithms. After that, I will be introducing you to the Collection Framework, which is a set of classes in Java that will implement these structures into our code. I will also be showing you how to implement all of these methods natively ie. manually written.

**Big O(h)**

We will begin by learning about what we called Big O/Big Oh. Big O is a notation that computer scientists use for measuring efficiency of an algorithm when compared to a given size of data. Think of it as a ratio between how much data you are given, and how the algorithm performs when you give it more data, or less data. Big O is really a very simple concept but is very difficult for beginners to grasp. There are at most, 7 levels of efficiency. Going from worst to best, we have: O(!n), O(, O(, O(nlogn), O(n), O(logn), 0(1). As you can see, every time we use the notation, we place the efficiency within parentheses following a capital letter O. You read the O as ‘order’, so O(!n) is read as “order of factorial n”. O(1) is what we call linear time. This means that no matter what size of data is given to the algorithm, it will always take the exact same amount of steps to solve. It should be noted that anytime logarithms are used in computer science, a base of 2 is assumed, as logarithms are almost always used for measuring binary trees. More in that when we look at binary search. I’ll try to explain big O more in depth as we continue forwards. Don’t be too nervous as it is actually quite straightforward.

**Recursion**

You may or may not have heard of recursion mentioned during your studies. One such example may be in Linux when we “recursively” delete files. In computer science the term recursion almost always means a function calling itself. I won’t go too in depth here, as I explain this in a bit more detail within my C programming notes, but essentially, when a function is called, a new stack frame is created in memory. A stack frame is a block of memory within the stack that contains local variable information, as well as the return address and parameters of the function. So let’s say that we have a method that calls itself. What would happen? Well, as I mentioned, a new stack frame is created for every function that we call, stack frames would continue to accumulate and occupy the stack. With no condition to stop this, we create an infinite loop, and we inevitably run out of stack space. This is called a stack overflow (aaaahh see, now you know what that refers to!) So lets say that we place a limit on the amount of times the function may be called. What happens when we reach the last call, and what happens with all the stacks we’ve created. Well, once we reach the “base condition” as we call it (the condition that results in the discontinuation of a recursive process), then each stack is popped ie. removed from memory and the return value is sent back to the calling function. The classic example of recursion is factorials. If you don’t remember what factorials are, it’s basically a way of taking the sigma of a set of numbers such that n is the last number in the set. So !5 (factorial 5) would be 1 + 2 + 3 + 4 + 5 = 15. Note in the following code that the base case is an if statement. The base case **must** be an if statement for recursive functions.

public class Main {

static int factorial(int n) {

if(n == 0) {

System.***out***.print(n);

return n;

}

System.***out***.print(n + " + ");

return n + *factorial*(n-1);

}

public static void main(String[] args) {

int result = Main.*factorial*(5);

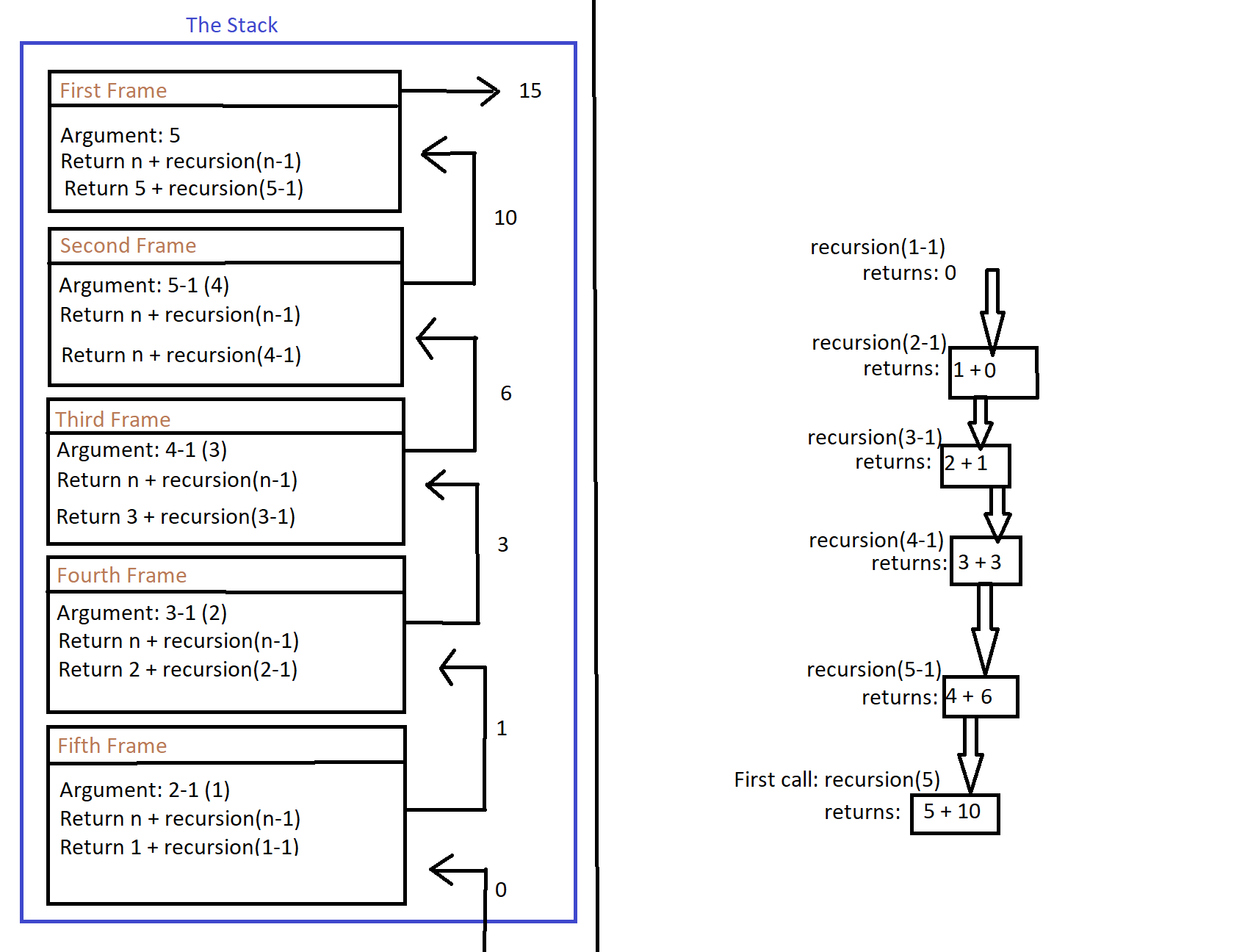
System.***out***.println(" = " + result);

}

}

Output: 5 + 4 + 3 + 2 + 1 + 0 = 15

So here we have a static function called recursion which returns an int. The parameter n acts as both our base case condition and as addition operand. First and foremost, forget about what this function is returning for a moment, and focus on each instance of stack frame as the function is called. So in main, factorial is called with 5 as an argument. A new stack frame is created where it stores the parameter 5. Then we call the function again in the return statement, this time with 5-1 which is 4. This process repeats until we meet the case condition ie. until n == 0. At this point, the infinite loop is broken, and we begin to pop stack frames and return the values held within each frame. So first, we return 0, then we return 0 + 1, then 2 + 1, 3 + 2, 4 + 3, and finally 5 + 4. In other words, 0 + 1 + 2 + 3 + 4 + 5. Here is a graphic I drew in paint attempting to demonstrate this.



I forgot to add the last frame but there would be another frame for the last call of the function on the left hand side where the 0 is coming from. Another popular, albeit more complex recursive function is the Fibonacci sequence where you begin with the numbers 1 and 2, and then the third digit is the sum of the 2 previous digits, so the third digit is 3, then 5, then 8, then 13, etc. Here is the code for the Fibonacci sequence:

static int fibonacci(int n) {

if(n == 0 || n == 1)

return n;

else

return (*fibonacci*(n-1) + *fibonacci*(n-2));

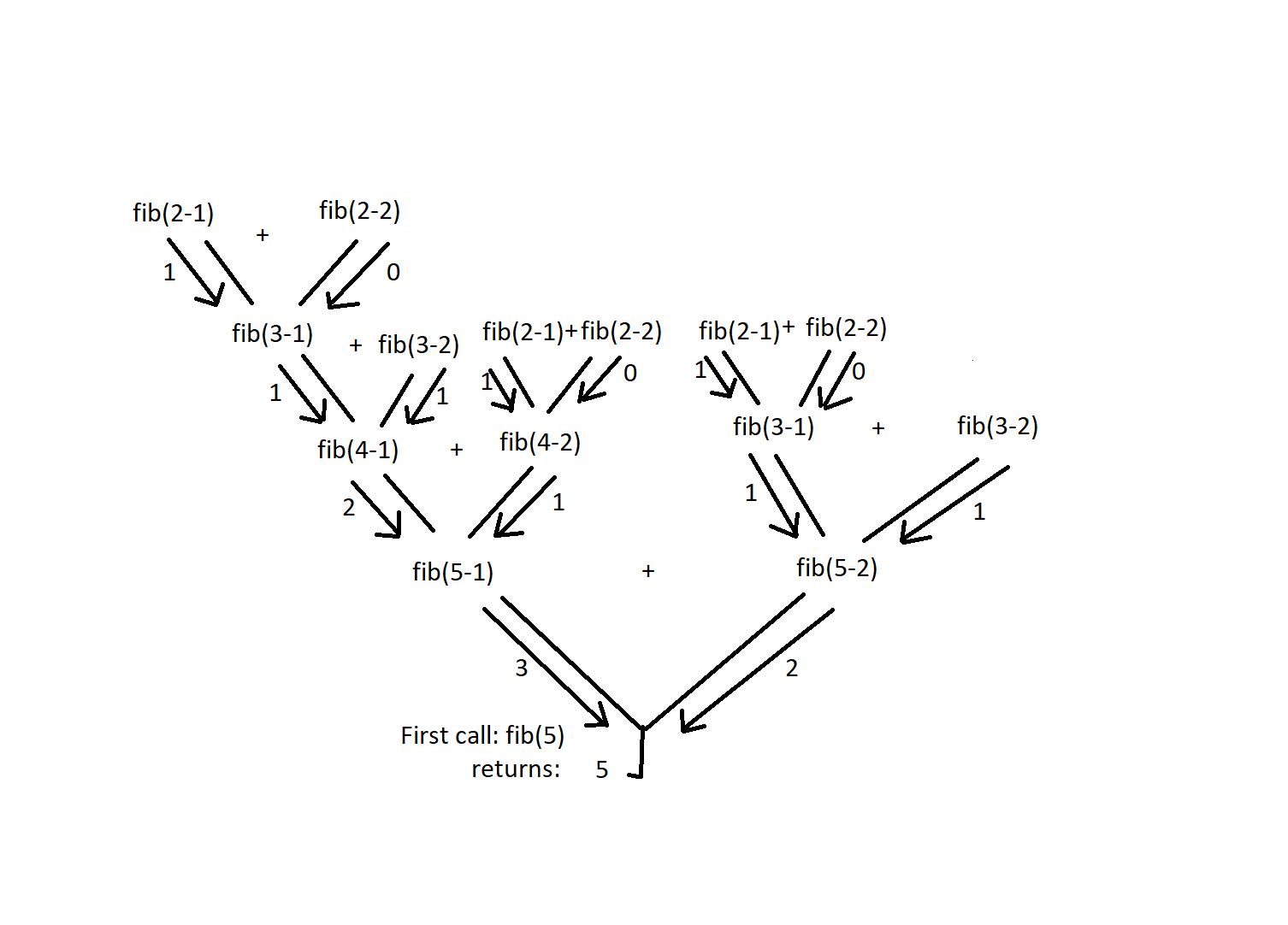
}

}

fibonacci(12);

Output: 144

This is a pretty simple piece of code although it is quite complex under the hood since we are now returning the return values of two functions which are returning the return values of two more functions, etc. (exponential). Here is the diagram of what’s happening (in paint once again, of course!)



<----------------------------------------------HALF WAY ------------------------------------------------->

**Native Linked List**

A linked list is a data structure that comes in 2 forms: A single linked list, and a doubly linked list. You can think of a linked list as a daisy chain of nodes which contain references to each other. Within a single linked list, each node contains 2 fields: A data field, and a next reference field. The data field contains the actual data that we want the linked list to contain. This is similar to an array since arrays are limited to one datatype. The primary difference between an array and a linked list, however, is that linked lists do not have indexes. This is a blessing and curse, as it makes data search inefficient, but allows for true dynamic sizing, unlike an array (or array list which is not truly dynamic). The next reference field is essentially a pointer to the next node in the list. Ex. Node\_1 -> Node\_2 -> Node\_3 -> NULL. The last node in the list, typically referred to as the tail of the list, or last node, always points to NULL so that we may break from the chain when our looping statement detects that there are no more nodes in the list. More importantly is the head of the linked list, or the first node in the list. If this node is NULL, the list is effectively empty, and we cannot traverse it. It is an important node to distinguish from the rest of the list because the only means to traverse a linked list is to begin at the head node and follow the sequential links contained in each node. In other words, the traversal/seek time for a linked list is always O(n), as we always start from the beginning, and continue until the data is found. A doubly linked list operates in the same manner, only with doubly linked lists, each node contains 3 fields: A previous reference, the data, and a next reference. While doubly linked lists require a bit more memory to create, their usefulness over single linked lists tends to outweigh this issue. Doubly linked lists allow for much easier deletion of nodes, whereas it is very hard to delete nodes that are not head or tail in single linked lists. Please also note that this section is specifically about native linked lists, meaning that they are native to are program, and not imported from the collection’s framework. We will also briefly be discussing collections LinkedList and Generic linked list. Here I will provide some sample code for a native single linked list. There are 2 classes, one used as a template for node objects, and one for the list itself, which contains methods to operate on the nodes. It may appear daunting at first, however it is quite simple after a few read throughs.

Ex:

//Class for linked list node objects

public class LLNode {

private int data;

private LLNode next;

public LLNode(int data) {

this.data = data;

this.next = null;

}

//Optional

public int getData() {

return this.data;

}

//Optional

public void setData(int data) {

this.data = data;

}

public void updateNext(LLNode next) {

this.next = next;

}

public LLNode getNext() {

return this.next;

}

}

//Class for actual instance of list

public class LList {

private LLNode head;

public LList() {

head = null;

}

public void push(int data) {

LLNode push = new LLNode(data);

push.updateNext(head);

head = push;

}

public void pop() {

head = head.getNext();

}

public void printList() {

LLNode temp = head;

while(temp != null) {

System.out.println(temp.getData());

temp = temp.getNext();

}

}

}

Okay, so I’m going to try and explain how these two classes coordinate, and it will be up to you to follow along to the best of your ability. The LLNode class is responsible for creating a linked list node. It contains the two fields that we discussed: data and a reference to the next Node object. In case I havn’t mentioned this yet, or if you simply forgot, when we have as one of our instance variables, an instance of the current class, this is called a self-referential class. Since objects can also be called references (as they are just pointers to memory addresses), you can think of a self-referential class as a class that uses itself as one of its data members. Anyways, I added two optional (but generally included) methods (a getter and a setter) for data retrieval, and data updating. updateNext() is the most notable out of all of these, as it is used for nearly all operations. Within out LList class, we create an instance variable called head which will always be the first node in our list. When we push a new node onto to list (add at the beginning), we need to create the node to be added, set its next reference to point to the head, and then update head to be the node we’re pushing. Now, the node that once was head is now in front of the added node so to speak, and head is set to this new node. If we wanted to delete a node, we could do this by pointing a node’s next field to the node after the one it currently points to. Or in other words, node1 would now point to node3 (the next reference of node2). In C, we would normally use free() to tell the system the node is no longer necessary, however in java, the node is deleted along with it’s data by the garbage collector.

Let’s discuss big O for single linked list. I mentioned earlier that the big O for searching the linked list was O(n). This is of course because we must always start at the head and traverse the entire list until the data is found. The average seek period will grow as the list grows. Push and pop operations (deleting at head or adding at head) are O(1), as head is always the first node and we can immediately operate on it. Some single linked lists keep track of tail as well, although it is not common. Deleting from tail is O(n) because single linked lists contain no previous reference, meaning that we have no way of updating the node previous to tail to point to NULL without going through the entire list. Alternatively, adding a node to the end of a single linked list that keeps track of the tail would be O(1) since we could simply point tail’s next reference to be the node being added, and set the node being added’s next reference to NULL. It should also be noted that linked lists are technically never sorted, although we will discuss a data structure which mimics a linked list and can be sorted, and therefore, can also be searched using binary search.

**Native Doubly Linked List**

The doubly linked list is very similar to the single linked list. As I mentioned the only difference is that a doubly linked list contains a reference to the previous node as well as the next. I wont bother to write out the code for doubly linked list since it is mostly the same. The methods used for updating references must now also update the previous reference as well. For instance, in order to delete a node from a doubly linked list, not only do we need to set the node’s next field to jump forwards a node, we also need to update the node after the one we’re deleting’s previous node. In other words: Node1 <=> Node3. The head of the doubly linked list will have it’s previous reference set to NULL, and tail will have it’s next reference as NULL. This gives us the ability to traverse the list backwards if you had to reverse a sequence. I will not discuss big O for doubly linked list either since it exactly the same, the only difference being that we can now remove the tail node with O(1) due to our ability to do the following: tail.getPrev().updateNext(null); tail = tail.getPrev(); I’m sure that I am doing you a disservice by not writing the code for doubly linked list here, but I am certain that a quick google search will be able to provide you with a sufficient example. More-over, I encourage you to draw out the flow of the doubly linked list to understand why and what things need to update their references.

**Generic Linked List**

As you know, a generic is a reference datatype which extends Object. Generic linked lists allow us to create linked lists of varying datatypes with the same linked list class. For example, you may want a linked list which is of type Integer, as well as a linked list of type String. The primary difference in the code is that our class name will end with the Generic signifier <T>, as well as all instances of the class, eg. private LLNode <T> next; The constructor will have a parameter of type T eg. public LLNode(T data) {...}

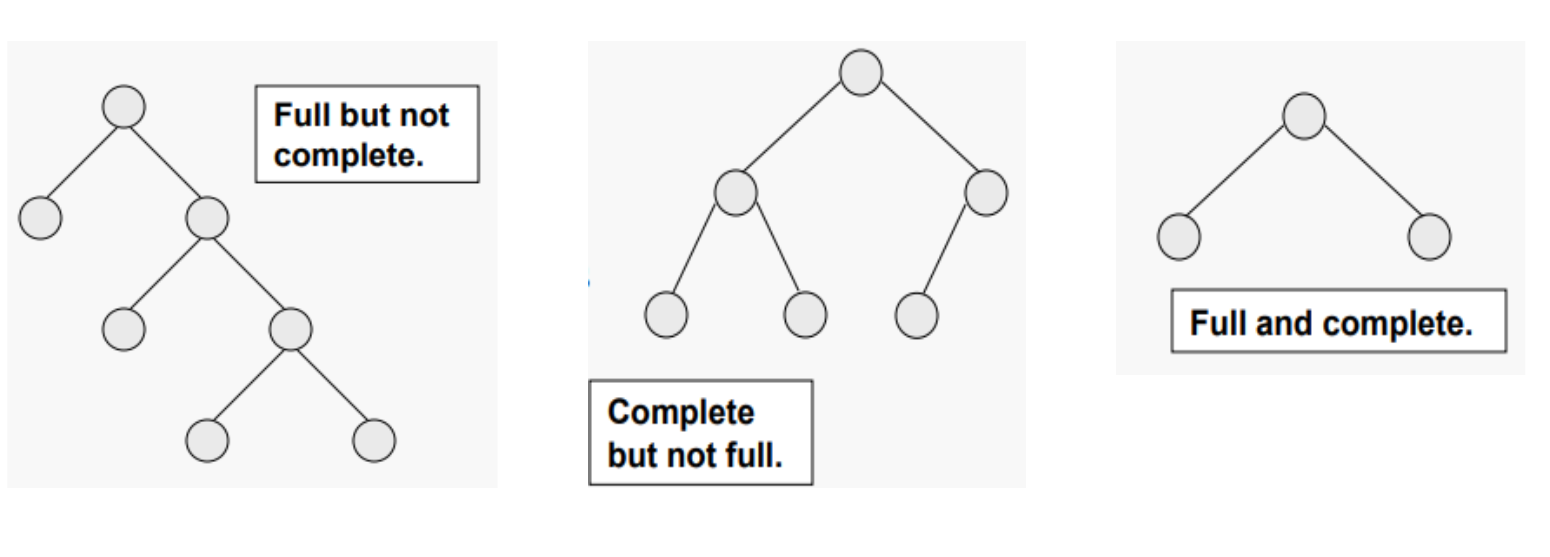
**Collection Linked List**

As you should know, there is a linked list class in the Java Collection framework. It is a subset/child of the List interface, as is ArrayList, Vector and Stack. This means that the collection implementation of a linked list contains many of the same methods as array list does. Some of it’s methods include: add(), addFirst(), addLast(), removeFirst(), removeLast(), an overriden toString for printing, etc. There is not really a major advantage of using Collection linked list over a native linked list. It is a matter of whether or not you’d prefer to be lazy with your typing, and whether or not you are okay using a more bloated class with unnecessary features in your code.

Trees

In the last section, I briefly mentioned a data structure which could mimic a linked list and still perform binary search due to it’s sorting capabilities. You are aware of trees to a degree already, since binary search uses this concept. While binary search uses the concept of trees, it is not a data structure in-and-of itself. Rather, it operates on a list using the logic of a tree, whereas a tree data structure stores data using it’s logic. When we think of a tree, you probably picture the trunk of the tree extending upwards and outwards as the branches multiply. In computer science, trees are almost always inverted, meaning that it’s probably easier to think of a tree as more like the roots of a tree, growing outwards as it extends downwards. What we call the root node refers to the top node in the tree. A parent node is any node which has children, and a child node is a node with a predecessor/parent. A leaf node is a child node with no successor ie. no children of it’s own. An internal/interior node is any node which is not the root, or a leaf node. Sibling nodes are two or more nodes which share a parent. The height of a tree is the maximum number of levels present in the tree, and the level of a node is its distance from the root.

**Binary Trees**

Now that we understand the concept of a tree and all of it’s terminology, we can begin to look at some data structures which are subsets of the tree concept. The first and most familiar is the binary tree. Annoyingly, the computer science community has coined terms for various states that a binary tree can exist within (I know, it’s a lot to memorize). The first term is the balanced tree. A balanced tree refers to any tree where no two nodes ever differ by more than one level. In other words, you may not have a node on level 1 and level 3 of the tree and say that it is balanced. A full tree is a tree which has only leaf nodes or exactly two child nodes. A complete tree is one where are levels except for possibly the last are completely full, and if the last level is not full, all left nodes must be filled. 

It took me a minute to understand why the second image is not full. The answer: the right node on the first level is not a leaf and does not contain two child nodes. Finally, we have what is called an almost complete tree, which is a complete tree which has its data inserted from left to right, level by level. The second image above is simultaneously a complete tree and an almost complete tree (it makes no sense, I know!). If for example, the left node with a missing sibling were actually a right node, this tree would no longer constitute as complete or almost complete.

In order to insert data in a binary tree structure, there is a very distinct pattern that must take place if we want to keep it ordered. When the list is still empty, the first node to be added permanently becomes the root node (unless deleted). When the second node is added, a comparison is made between the root node’s data, and the pending node’s data. If the pending node’s data is less than the root node’s data, it is placed to the left of the root, and if it is greater than the root node’s data, it is placed to the right. This check is done between every parent that the pending node comes in contact with, until eventually, it reaches a leaf node and becomes it’s child. I will now provide the code for a binary tree structure. Similar to linked list, we have 2 classes: one for the creation of node objects, and the other for the tree itself, containing all methods for operating on data.

public class BinaryTreeNode {

private int data;

private BinaryTreeNode left;

private BinaryTreeNode right;

public BinaryTreeNode(int data) {

this.data = data;

this.left = null;

this.right = null;

}

public BinaryTreeNode getLeft() {

return this.left;

}

public BinaryTreeNode getRight() {

return this.right;

}

public int getData() {

return this.data;

}

public void insert(int data) {

if(data < this.data) {

if(left == null)

left = new BinaryTreeNode(data);

else

left.insert(data);

}

else if(data > this.data) {

if(right == null)

right = new BinaryTreeNode(data);

else

right.insert(data);

}

else

System.out.println(“Duplicate items – not adding ” + data);

}

}

public class BinaryTree {

private BinaryTreeNode root;

public BinaryTree() {

this.root = null;

}

public void insert(int data) {

if(root == null)

root = new BinaryTreeNode(data);

else

root.insert(data);

}

public void displayInOrder() {

displayInOrder(root);

}

//Print order for in-order sequence

public void displayInOrder(BinaryTreeNode subRoot) {

if(subRoot == null)

return;

displayInOrder(subRoot.getLeft());

System.out.println(subRoot.getData() + “ ”);

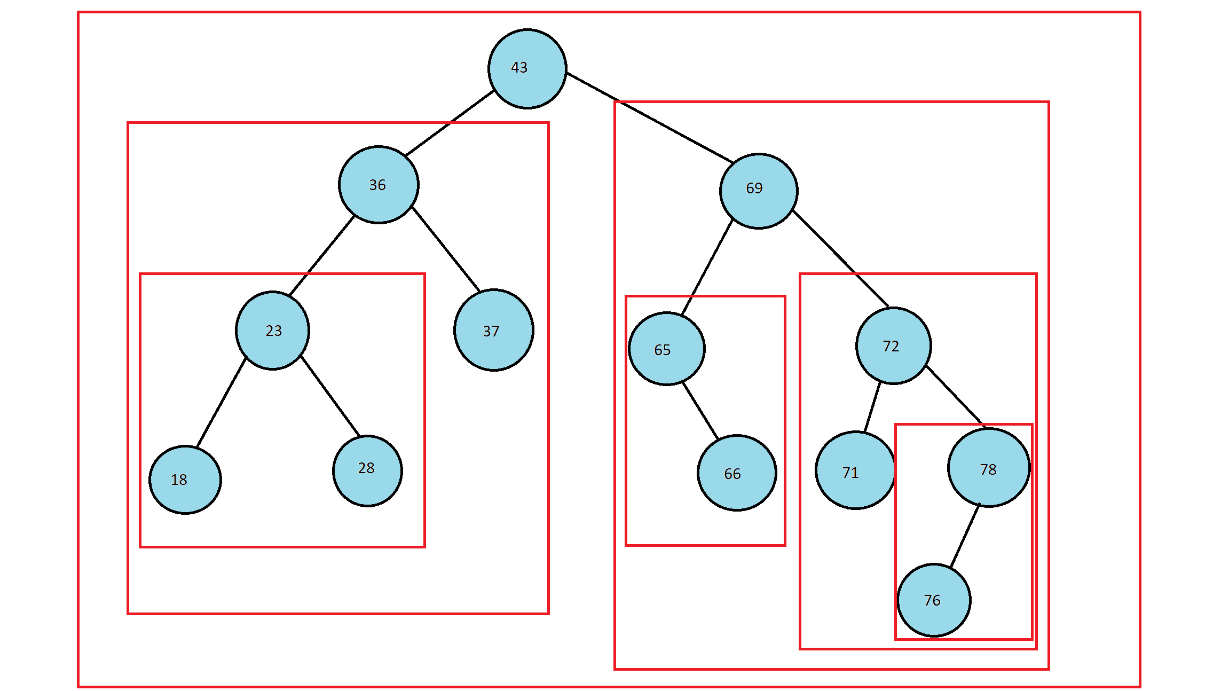
displayInOrder(subRoot.getRight());

}

}

Okay, so once again, this looks quite daunting... Binary tree is slightly more difficult than linked list so I will try to be a bit more in-depth to accomodate. Hopefully, after understanding how linked list works, the BinaryTreeNode class at getters and setters make sense, as they are simply for initializing each instance’s data field to be the user provided data. The left and right fields represent the possible children of the node, and are set to null initially. It is the insert method that may be mildly confusing, though it’s nothing you shouldn’t be able to handle. It may not look it, but this is a recursive function, as we must continuously call it for each parent node that the pending node comes in contact with. There are 3 possible cases: The pending node’s data is smaller than that of the comparative node (insert at the left of the node), the data is greater (insert at right), or the data is equivalent. Binary tree could possibly be categorized as a set, as it does not contain duplicate data. Hence we print a message telling the user that the program did not add the data if it already exists in the tree. As for the former two conditions, if the comparative node does have a left or right node yet, we can simply set this field equal to our new node being added. Otherwise, we call insert again, and attempt an insert with the child as the new comparative node (thus recursively calling the function). Another area in the code that may throw you off is the overloaded function displayInOrder(). First of all, there are 3 ways to retrieve a trees data, and in order is the most common method. I will be discussing all 3 methods shortly. The reason that we create an overloaded method is because once again, this is a recursive function, although the first call will begin at root, thus we have no parent node to give as a parameter. Instead, we call the unparameterized version, and pass the root node as the argument for the parameterized version to allow for polymorphism.

So those 3 ways of accessing the tree data I was mentioning... They are known as pre-order, in-order, and post-order accessing. They are recursive concepts and retrieve data according to subtrees. Pre-order accessing is when we first get the root, then left subtree, then right subtree. In-order is left subtree, then root, then right subtree. Finally, post-order is left subtree, then right subtree, then root. This concept can be confusing if you are a visual learner, which is why I recommend drawing boxes around the subtrees if visual learning is your style. Every time a node splits, it creates a new subtree, encapsulating all successors. Here is an image I drew in paint to visualize this:

As you can see, each red box indicates a tree/subtree. When doing pre, in, and post order searches, rather than confuse yourself staring at the nodes, separate the subtrees, and think of each one as a node. That way, you begin at the outermost tree, and think of the right and left subtrees as nodes, and recursively continue the pattern. If you have trouble remembering pre, in, and post order, perhaps this little visualization will help:

Pre-order In-order Post-order

1 2 3

2 3 1 3 1 2

Notice that if we look at the root node, pre is 1, in is 2, and post is 3. When we get to the second level, we always start at the left if possible. Hopefully that makes some sense. Here are the pre-, in-, and post-order sequences for the tree in the diagram above:

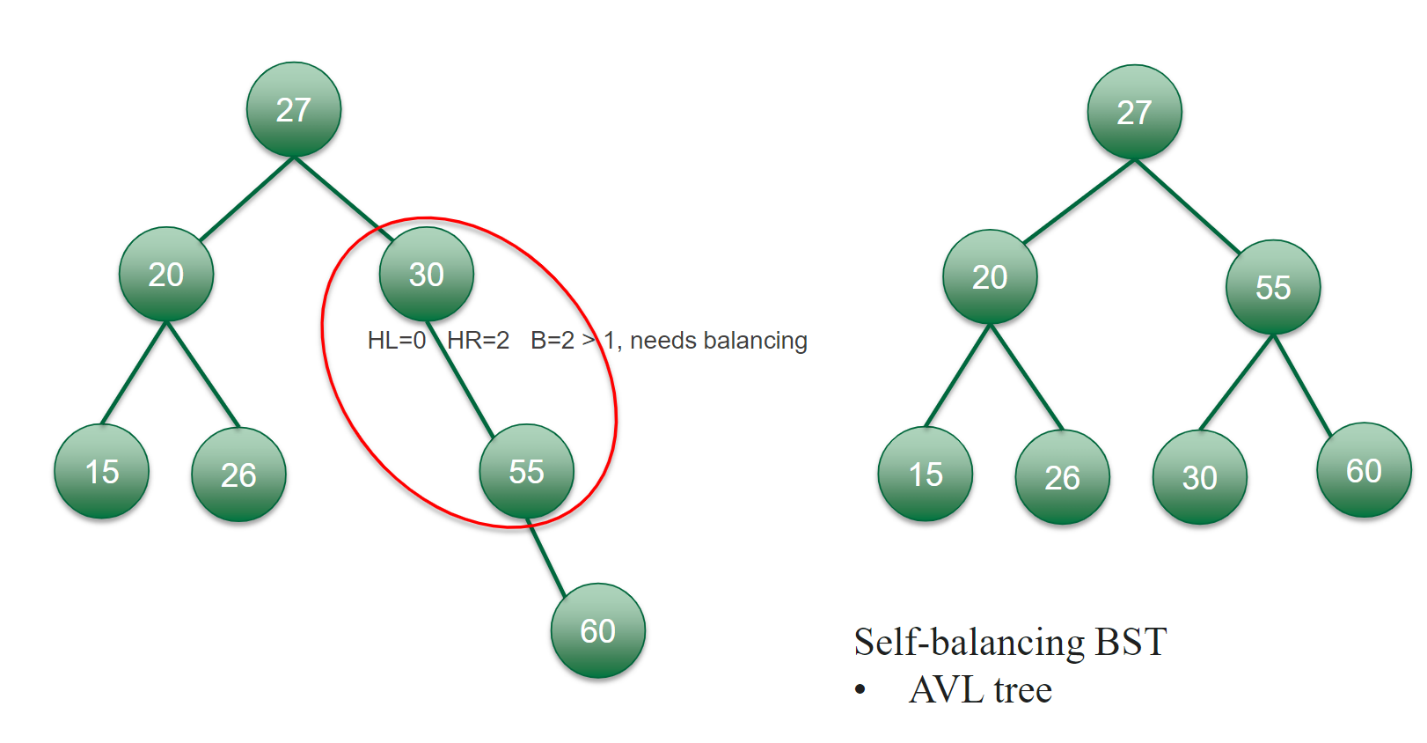
Pre-order: [ 43, 36, 23,18, 28, 37, 69, 65, 66, 72, 71, 78, 76 ]

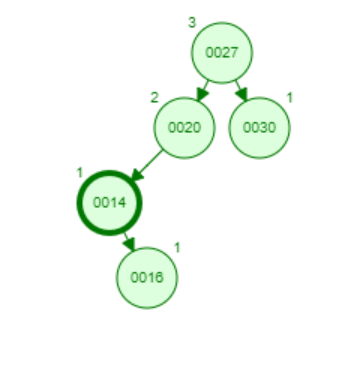
In-order: [ 18, 23, 28, 36, 37, 43, 65, 66, 69, 71, 72, 76, 78 ]

Post-order: [ 18, 28, 23, 37, 36, 66, 65, 71, 76, 78, 72, 69, 43 ]

In the code example that I provided, the printInOrder() function displays the array in order. For us to be able to display the tree in the other two orders, it is as simple as swapping the function calls appropriately.

Big O for binary trees is very good as you should be aware from binary search. We can guarantee a big O of O(logn) for all operations under the condition that the tree is a complete tree (otherwise, worst case is O(n)). How do we get this? Well, in a worst-case scenario, you can imagine a binary tree where all data that gets added is greater than the previous node that was added. For example, adding [ 1, 2, 3, 4, 5, 6 ] would create what’s called a right-heavy tree, where you basically just have nodes in sequence as a right arm. If you think about it, this would really just be a linked list, meaning our worst case is O(n). In terms of how we get O(logn), that’s a bit more mathematical but it’s not as bad as it seems at first. If you recall from the binary search section, we can get the maximum available node spaces if we know the height of the tree by doing the formula where h is height. With this knowledge, we want to obtain the minimum height that the tree could be given the maximum number of nodes (in a worst-case scenario). Given n\_MAX (we’ll call it n), we can plug it into our height formula: which, in big O, becomes or O(logn). But how can we get our tree to be a complete tree? We use a method called self-balancing tree (specifically we are talking about AVL balancing). The idea of a self-balancing tree is to maintain order, while balancing each node using rotations. Rotations come in four forms: a left-right rotation (LR), a right-left rotation(RL), a right rotation (RR), and a left rotation (LL). A rotation occurs after each node is added if the tree is imbalanced. This is calculated using the reverse-height, and by that, I mean starting at the added node, going upwards. Perhaps a better way to look at it is simply the number of left child nodes, and the number of right child nodes, that a given root/subtree root node has.

In this image, the right subtree with 30 as the subtree root has no nodes on the left ie. HL = 0, but it does have 2 nodes on the right side ie. HR = 2. Since the difference between 2 – 0 is greater than 1, this tree is imbalanced. In order to balance it, we must choose: RR, LL, RL, or LR rotation. We follow the flow of the imbalance starting at the root or subtree root (in our case 30), and we note the direction that we move in. In this case, we move right once to get to 55, then right again to get to 60 (meaning we have a RR imbalance). That means we must do a left rotation to fix the imbalance. A strange quirk about this whole thing is that in order to fix a LL imbalance, we do a RR rotation to counter-act, and vice versa, however, a RL imbalance is fixed with a RL rotation, and a LR imbalance is fixed with a LR rotation. It’s also important to note that a RL rotation is really just a RR rotation followed by a LL rotation, and vice versa for LR rotation. Okay so what does all this rotation stuff mean, how does it work? Well, as I said, we have a RR imbalance, so we fix it with a LL rotation. A LL rotation gets the node with the value that is in between the other values and puts the highest valued node on the left. Then 55 becomes root, and 30 will go on it’s left, while 60 stays on the right.



Another example is on the right here, where we find a LR imbalance, and as I mentioned, LR imbalances are fixed using an LR rotation. Therefor, we do a left rotation where 14 rotates to the left of 16 (since it’s the middle value out of 20 and 14), and then we do a RR rotation where 20 goes to the right of 16 thus balancing the tree.

**Deletion**

**Heap (The data structure)**

A heap (not to be confused with “the heap” which is memory-based), is a type of binary tree, but operates differently than the one we discussed previously. There are two types of heaps: Max-heap and Min-heap. In the case of max heap, each root for each subtree must be the highest number in that subtree, although the order of left vs right makes no difference. The same is true for min-heap, but smallest at the top and largest at the bottom. Heaps must be complete or nearly complete at all times. In order to add to a heap, we read the heap left to right, top to bottom, and fill the next available spot with the new node. When then perform what’s called a heap-up where we check each consecutive parent node and continuously swap positions with the added node while the parent node is larger (for min heap), or while the parent is smaller (for max heap). This is O(logn). In order to delete from max-heap what we do is select the node to be deleted, then we get the right-most, bottom-most node in the subtree of our delete node, and perform a heap-up, switching it with each consecutive parent node regardless of whether or not the parent is smaller/larger until we eventually replace our subtree root that we want to delete with it. Once the bottom-most, right most node is now the root for that subtree, we need to fix the heap by checking if the right node is <= to the new root node (heap can have duplicates unlike tree). We must now fix the heap if the new root is not the largest. In order to do this, we compare the new roots left node with the new roots right node. The new root swaps places with whichever is larger. This continues recursively going right to left until it either reaches the final level of the heap, or it is larger than the node it compares with.

You may hear the term array-based heap thrown around. We actually did this with binary tree too. Essentially, all we do to put our heap in array form is read the values from right to left, top to bottom into each consecutive index in the array. The reason this is important for heap is because while we may have holes when storing a binary tree in an array (if we don’t balance it), heap will always compactly fit into an array since it must always be balanced. This allows us to reliably perform arithmetic on the nodes to see where their respective indexes will lie. The formula to do this is for left children and for right children. To calculate the index of the root node’s left node, the formula would be 2 \* 0 + 1 = 1 since height of root is 0, and for it’s right child 2 \* 0 + 2 = 2. This makes sense because root will reside at index 0 always, it’s left node at 1, and it’s right node at 2.

Hashing

**What is it?**

You can think of hashing as converting a value into another form. Hashing algorithms come in many flavors, and they are basically mathematical computations to translate one piece of data into another. This is generally performed on a lot of data containing key-value pairs, although hashing is also a technique that can be used for security purposes. The drawback of hashing a large amount of data is that we may end up with the same hash value for 2 different keys. This is called a collision and it can be resolved in numerous ways. First, let’s look at some hashing methods.

Direct: Key is the address of the array that we’re storing our data in. No algorithmic manipulation is performed. This is bad because we would have a different index for every single key, however we would avoid collisions.

Subtraction: We subtract some constant from the key assuming that the key is >= to the constant.

Modulo Division: We divide the key by the array size and use the remainder as the index for the data in the array.

Digit Extraction: Selected digits are extracted from the key and used as the index.

Mid-square: Key is squared and the address is selection from the middle of the squared number. This can lead to overflow of the datatype.

Folding:

fold-shift – Key value of divided into 3 parts whos size matches the index size. Then the left and right parts are shifted and added with the middle part. 123456789 is divided into 123 – 456 – 789. Then they are added: 123 + 456 + 789 = 1368

fold boundary – Key value is divided into parts. Left and right parts are reversed and added to the middle. 123456789 is divided into 123-456-789. Then 123 -> 321, 789 -> 987. We add 321 + 456 + 987 = 1764

Rotation: Usually used in combination with other methods, we move a part of the end of the key the the front to get index.

Pseudorandom generation: Key is used as a seed in a random generator. The result is scaled using modulus division to fit index range of the array.

Please note that none of these are secure hashing algorithms that you would see on a password in a database. The aim of these hashing algorithms is to create as few collisions with the keys as possible, so data storage is convenient, fast, and easy. It should be noted that for keys which are strings, we typically use the ascii values in some form for the hash.

**Collision Resolution**

Here are some of the most popular methods to deal with collisions when they happen.

LinkedList table: A linked list table is when each entry in the array is actually a linked list. If a collision occurs, the duplicate’s value is stored in the linked list. Linked lists in this context are often referred to as buckets, and thus, each row in the table is a bucket.

Load factor: The of load factor is a formula: = n/L where α is the load factor, n is the number elements in the table, and L is the number of buckets in the table (the length). Ex. if the table is an array of length 1000, and it has 750 elements added to it, the load factor is 0.75. If the load factor is reached, the table is increased. This is basically an indirect way of dealing with collisions, as it is preventative, but does not solve the problem.

Open addressing: Resolves collisions in the prime area, rather than the overflow area

linear probe – Add one to the current index and try again until you find a spot. If no empty spots, reallocate to a larger table.

quadratic probe – increment to hashed index is the collision number squared. For a collision at index 3, check if 3^2 = 9 is free (make sure to take appropriate remainder as to not exceed bound of the array).

pseudorandom resolution – When a collision occurs, use the collision index as the seed for random number generator

key-offset – Calculate a new index as a function of the old index and original key.

Set Vs Map

**Set Interface (Collection Framework)**

We’re going back to collection once again to take a look at the Set interface. As I’ve already mentioned, Sets do not have duplicate values. It is an interface that has 3 primary implementations: HashSet, TreeSet, and LinkedHashSet.

HashSet: Stores elements in a hash table (unordered)

TreeSet: Implements SortedSet. Stores elements in order, either by elements natural order or by user-defined Comparator

LinkedHashSet: Maintains a linked list of the entries in the set in the order of insertion.

**Map Interface (Not part of Collection Framework)**

The map interface is related to Set but has distinct differences. It associates key=>value pairs. Keys must be unique, though values may repeat. Map also has 3 primary implementations: HashMap, TreeMap, and LinkedHashMap (just like set’s implementations).

HashMap: Stores elements in hash tables

TreeMap: Implements SortedMap. Stores elements in order in a tree, either by elements natural order, or by user-defined Comparator.

LinkedHashMap: Maintains the insertion order of keys

**Comparison**

|  |  |
| --- | --- |
| Set | Map |
| Contains only values | Contains key=>value pairs |
| Does not allow duplicates | Allows duplicate values but not keys |
| Interface that extends Collection Interface | A separate interface |
| Set uses Map internally | Map uses hashing technique to store key=>value pair |

Queue and Stack

**Stack**

I have grouped these two data structures together as they are pretty similar. In fact you can create a queue using 2 stacks, and you can create 2 stacks using 2 queues. A stack is a LIFO structure (Last in First Out). Think of a stack of anything. My favorite example is a stack of plates. To add a plate on top, we “push” the plate on top. To remove the plate, we “pop” (sometimes also called pull). You may only add to the top of the stack and remove from the top. An example of using the stack would be in a web browser when you use the back and forward arrows, you push and pop pages that you have visited (note that this requires a buffer). Another example might be if you had a mathematics formula and you wanted to figure out if the braces were correct eg. (a+(b+c \* [r/x])) -> valid if(s == true) { print('correct'}; ]")) -> invalid. Here is some code that uses the collection implementation of stack from the List interface:

static boolean braceCheck(String equation) {

Stack<Character> braces = new Stack<>();

for(int i = 0; i < equation.length(); i++) {

char c = equation.charAt(i);

if(c == '(' || c == '{' || c == '[')

braces.push(c);

if(c == ')') {

if(braces.peek() == '(')

braces.pop();

else return false;

}

else if(c == '}') {

if(braces.peek() == '{')

braces.pop();

else return false;

}

else if(c == ']') {

if(braces.peek() == '[')

braces.pop();

else return false;

}

}

return true;

}

public static void main(String[] args) {

System.***out***.println(*braceCheck*("(a+(b+c \* [r/x]))"));

System.***out***.println(*braceCheck*("if(s == true) { print('correct'}; ]"));

}

output: true, false

Basically, we create a stack called braces, iterate through the string that we pass to the function, push the current character onto the stack if it is an opening brace (‘(’, ‘{’, ‘[’) and if the character is a closing brace, we use the peek() method to look at the variable stored at the top of the stack and see if it is a match. If it is, we pop that opening brace from the stack, otherwise we return false.

The big O for stack is O(1) for push, O(1) for peek, and O(1) for pop, meaning it is very efficient no matter how big the stack is since we only ever operate on the first item. That being said, this algorithm isn’t practical for most things, so you probably wont use it all that often.

**Queue**

Queue is it’s own Interface in the Collection framework. There are two types of queues as far as I know: regular queues, and priority queues. Queues can be implemented in a variety of ways (Collection interface, ArrayList, LinkedList, circular buffer, etc.). The most common way of thinking about a queue is to imagine a line of people. If you’ve ever been to a hospital for some sort of operation, they will have a TV with names in a queue based on who registered first. By adding to the queue (called enqueue), the item gets sent to the end of the array, and by removing from the array (dequeue), the first item of the array gets removed. This leads to a glaring issue, which is that (at least for array-based queues,) the item that we dequeue is now empty and there is no way to reuse that empty index. Additionally, performing too many enqueue operations will eventually exceed the size of the array. If you use ArrayList, this is solved, as ArrayLists are dynamic, however, this is still very inefficient. Our only solution is to shift all elements to the left which takes O(n) complexity and is not very ideal. Sound like a raw deal? Well, maybe for array-based implementations, but for doubly linked lists, queues work wonderfully. Enqueue becomes O(1), dequeue becomes O(1), and searching becomes O(n) (although typically, you wouldn’t really use queue if you had to search a lot). Also note that queues are synchronous. Many of the tasks executed by your computer rely on a queue structure for execution. In other words, instructions are enqueued by the compiler for the CPU to execute, and then dequeued once execution is complete. Some languages, such as JavaScript, allow for asynchronous operations, meaning that if an instruction takes a long time to execute, the CPU might carry out the next instruction while it waits. For example, a large query may get sent to a database, and the CPU may execute successive instructions while it awaits the results.

**Priority Queue**

Priority queue is the same as queue, only it will insert elements into the queue based on the natural ordering of elements that Comparator provides. That means that enqueueing the letter ‘C’ would be inserted in front of ‘P’, and thus dequeuing returns C before P, even if it was enqueued after P. Insertion becomes O(logn) which is definitely not ideal, although we can efficiently implement a priority queue using a heap structure. Dequeuing would be like removing min/max from the top (O(1)) of the heap, and enqueuing would be like adding as you normally would to a heap (log(n)). This improves access and search times (normally O(n)), to be O(logn).