Working with Strings in Java

the Flesch Readability Score

$$Fleschscore = 206.835 - 1.015(words/sentences) - 84.6(syllables/words)$$

Basics of Strings

- Srtring are objects (not primitive).
- They are represented in the **heap**(not in the menmory).
- Variables refer to them have references.
- Strings are represented as arrays of chars.
- Strings are **immutable**

String append - multiple objects

```
String text = new String("1");
String text2 = text.concat("2"); (append things on)
Strng text2 = text + "2";
// text2 = "12";
// text2 refers to a whole new object, a new address in the heap.
// Make no change to the first object.
```

Interned String - one object

```
// Create a noe string object
String text = new String("1");
// text2 and text3 refer to the same interned string
```

```
String text2 = "1";
String text3 = "1";
```

Comapring Strings - .equals vs. **==**

```
String text = new String("1");
String text2 = new String("1");
text.equals(text2); // true; compares characters
text == text2; // false; compares variable calues(references)
```

Working with Strings built-in method

```
public static boolean hasLetter(String word, char letter) {
    for (int i = 0; i < word.length(); i++){
        if (word.charAt(i) == letter) {
            return true;
        }
    }
    return false;
}
```

for-each loop

```
public static boolean hasLetter(String word, char letter) {
    // toCharArray returns the chars in a String, as a char[]
    // c gets a COPY of each value in cArray
    for (char c : word.toCharArray()){
        if (c == letter) {
            return true;
        }
    }
    return flase;
}
```

replace

```
// Not working.
public static String replace(String word, char gone, char here) {
    char[] cArray = word.toCharArray();
    // c gets a COPY of each value in cArray
    for (char c : cArray){
        if (c == gone) {
            // Only changes the copy of that char
            c = here;
        }
    }
    return newString(cArray);
}
```

```
// Working.
public static String replace(String word, char gone, char here) {
    char[] cArray = word.toCharArray();
    char[] cArratMod = new char[cArray.length];
    int i = 0;
    for (char c : cArray){
        if (c == gone) {
            cArrayMod[i] = here;
        } else {
            cArrayMod[i] = c;
        }
        i++;
    }
    return newString(cArrayMod);
}
```

indexOf

```
String text = "Can you hear me?";
int index = text.indexOf("he"); // index is 8;
```

qualsIgnorCase, toLowerCase, toUpperCase ...

Regular Expressions

A pattern : characters are basic units

split method

```
String text = "Can you hear me?";
String[] words = text.split(" ");  // retrun an array of strings.
// " " this single space is a regular expression. It mathches single spaces.
```

3 ways to combine

- Repetition

```
String[] words = text.split(" +"); // mathces 1 or more spaces in a row
```

- Concatenation

```
String[] words = text.split("it");
```

- Alternation

```
String[] words = text.split("it+");  // "it", "itt", "ittt"...
String[] words = text.split("it*");  // zero or more: "i","it","itt"...
```

```
String[] words = text.split("it|st"); // OR
String[] words = text.split("[123]"); // anything in the set. "1", "2", "3"
String[] words = text.split("[1-3]"); // indicates a range
String[] words = text.split("[^a-z123 ]"); // NOT any chars in this set
```

Measuring Performance - Big O

Asymptotic Analysis

- **Count operations instead of time**
  - **Task**: Search for the letter 'a' in the word "San Diego".
  - **Linear Search**
    - **Operation**: basic unit that doesn't change as the input changes

```
// Finding 'x' in "San Diego" : 9 iterations, 29 operations.
public static boolean hasLetter (String word, char letter) {
    for (int i = 0; i < word.length(); i++) {
        if (word.chatAt(i) == letter) {
            return true;
        }
    }
    return false;
}
```

- **Foucus on how performance scales**
  - **Motivation**: If input is *twice* as big, how many *more operations* do we need?

```
// Constant Time
if (word.charAt(i) == letter) {
    return truel;
}
```

```
// Linear Time
// 1 (count) + (1(i) + 3n + 1(n))
int count = 0;
fot (int i = 0; i < word.length(); i++) {
    count++;
}
```

- **Go beyond input size** (how does the algorithm react to different sorts of input)

The big O Class

captures the **rate of growth** of two functions

$$f(n) = O(g(n))$$

```
f(n) and g(n) grow in same way as their input grows (up to constants)
```

- f(n) = O(g(n)) means there are constants N and c so that for each n > N, f(n) <= C g(n)
- Big-O only applies as n gets **large**.

With Consecutive Operation

- Runtimes are independent
- in a for loop: there will be n loop iterations, each iteration will take constant time.

```
// O(n)
for (int i = 0; i < vals.length; i++) {
    if (vals[i] < vals.[minIndex]) {
        minIndex = i;
    }
}
```

- O(1) + O(n) + O(1) + O(n) = O(n) - a Linear Algorithm

With Nested Operation

```
public static int maxDifference (int[] vlas){
    int max = 0;
    for (int i = 0; i < vals.length; i++) {
        for (int j = 0; j < vals.length; j++) {
            if (vals[i] - vals[j] > max) {
                max = vals[i] - vals[j];
            }
        }
    }
    return max;
}
```

- Count from the inside out
- O(n^2)

$$\because \log_m n = \frac{\log_a n}{\log_a m}$$

$$\therefore \log_{10} n = x log_2 n$$

$$x = \frac{1}{log_2 10}$$

Worst, Best and Average Cases

- Best Case - O(1)
- **Worst Case** - missing or in the end
- Average Case

Analyzing Search Algorithms

/	Linear Search	Binary Search
Best Case	O(1)	O(1)
Worst Case	O(n)	O(logn)

Analyzing Sorting Algorithms

/	Selection Sort	Insertion Sort	Merge Sort	Quick Sort
Best Case	O(n^2)	O(n)	O(nlogn)	O(nlogn)
Worst Case	O(n^2)	O(n^2)	O(nlogn)	O(n^2)

- Selection Sort: performance depends only on the size of the array

Merge Sort - recursion - O(nlogn)

- if list has one element, return
- **Devide** list in half - O(logn)
- **Sort** first half
- **Sort** second half
- **Merge** sorted lists - O(n) work to merge all the lists on one level

Measuring Performance - Benchmarking

Benchmarking

Times might not be consistant

- JVM is an abstraction, Operating System is an abstraction for hardware components
- Compiler makes choices that affect performances

Using Java Time

```
long startTime = System.nanoTime();
selectionSort(array);
long endTime = System.nanoTime();
double estTime = (endTime - startTime) / 1000000000.0
```

Abstraction, Interfaces, and Linked Lists

Abstraction

Hiding irrelevant details to focus on the essential features needed to understand and use a thing.

Abstraction Barrier

- User: *Behavior* specified
- *Implementation* specified

Data Abstraction

- **User of libraries** : Abstract Data Type(ADT) - interfaces/ abstract classes ; No implementation

  <<interface>> List

  add(Object)

  size() etc.

- **Library Developer** : Data Structure - Specific implementation of functionality

  ArrayList

Linked List vs. Array

Two ways to implement the same functionality!

- ADT : <<Interface>> List
- Data Structures : LinkedList, ArrayList

ArrayList

- Implements the List interface using an array
- Can access elements in constant time
- Takes O(n) to add an element to the front of an ArrayList : **ADT specifies funtionality, but not efficiency**
- Quick access to element given a particular index O(1)

(Doubly, Singly) LikedList

Implementation:

```
MyLinkedList object;
ListNode objects;
// sentinal / dummt nodes
```

- Takes O(n) to get an element at a particular index in a LinkedList : **Only have access to HEAD and TAIL of list**
- Takes O(1) to insert an element into the head of a LinkedList.

Random Acess

```
int[] a;
List b;
// Access number `n-th` element of the array/list
a[n];  // constant time
// The time complexity of List.get() depends on its implementation
//     - ArrayList.get(): O(1)
//     - LinkedList.get(): O(N)

// Abstract data type,
// list -> linked list

// 1.
// Assume you have list: b = 1 -> 2 -> 3 -> ... -> n -> n + 1 -> ... -> N
// b.get(n) will require you loop through the head of the list,
// keep forward until you find the n-th element.
// Or we can put it in this way:
// 2.
public int get(int n) {
  ListNode n = this.head;
  for (int i = 0; i < n; ++i) {
    n = n.next();
  }
  return n;
}
```

Generics and Exceptions

```
class ListNode<E> {
    // pointers to ListNode objects
    ListNode<E> next;
    ListNode<E> prev;
    // Stores data
    E data;
}
```

- A parameterized type. Our ListNode is "generic".
- Checked exceptions must be declared. (NPE is unchecked)

```
public class RememberLast<T> {
    private T lastElement;
    private int numElements;

    public RememberLast() {
        numElements = 0;
        lastElement = null;
    }

    // Must return a T
    public T add(T element) //throws NullPointerException
    {
        // Throw exceptions to indicate fatal problems
        if (element == null) {
            throw new NullPointertException ("RememberLast object cannot store null pointers.")
        }
        T prevLast = lastElement;
        lastElement = element;
        numElements++;
        return prevLast;
    }
    ...
}

RememberLast<Integer> rInt = new RememberLast<Integer>();
RememberLast<String> rStr = new RememberLast<String>();
rInt.add(3);
rStr.add("Happy");
```

Java Code for a Linked List

```
class ListNode<E> {
  // ListNode : recursive data type
  ListNode<E> next;
  ListNode<E> prev;
  E data;

  // Constructor . No type parameter in the constructor header.
  public ListNode(E theData) {
    this.data = theData;
  }
}

public class MyLinkedList<E> {
  private ListNode<E> head = new ListNode<E>(null);
  private ListNode<E> tail = new ListNode<E>(null);
  private int size = 0;

  // Empty list. Has zero data nodes, but two sentinel nodes.
  public MyLinkedList() {
    // Need to link the sentinel nodes to each other.
    head.next = tail;
    tail.prev = head;
  }
}

// 1. Constructor is defined to initialize member variables.
// 2. 具体需要初始化何种成员变量，视情况而定
// 3. Constructor 是否需要接受参数，视情况而定

// OR:
public class MyLinkedList<E> {
  private ListNode<E> head;
  private ListNode<E> tail;
  private int size;

  // Empty list. Has zero data nodes, but two sentinel nodes.
  public MyLinkedList() {
    size = 0;
    head = new ListNode<E>(null);
```

```
    tail = new ListNode<E>(null);
    // Need to link the sentinel nodes to each other.
    head.next = tail;
    tail.prev = head;
  }
}
```

Testing and Correctness

- Black Box Testing : Only tests through the interface
- Clear Box Testing : Tests which know about the implementation
- Unit Testing!

JUnit in Java

- Code to setup tests
  - setup run before each test to initialize variables and objects
  - setupClasee is run only once before the class to initialize objects
- Code to perform tests
  - assertQuals
  - fail
- Code to cleanup tests
  - tearDown

Testing Linked List's "Get" Method

```
// retireves the element at the index indicated or, if the
// index is invalid, throw an IndexOutOfBoundsException
public E get(int index);
```

- Corner Case(empty list): test get(0) in an empty list

```
// in testGet (in JUnit @Test)
try {
    emptyList.get(0);
    fail("Check out of bounds";)
}
catch (IndexOutOfBoundsException e) {
}
```

- Courner Case(negative index): Test get(-1) from a list with 1 element

```
// in testGet (in JUnit @Test)
try {
    shortList.get(-1);
    fail("Check out of bounds";)
}
catch (IndexOutOfBoundsException e) {
}
```

- Standard Use Case: Test get(0) from a list with 1 element

```
// in testGet (in JUnit @Test)
assertEquals("Check first", (Integer)65. shortList.get(0));
```

- Standard Use Case(get more than the first element): Test get(1) from a list with 2 elements

Markov Text Gerneration

- **Stage 1: Train** Build model based on data
- **Stage 2: Generate**

Implementation

- Class Design

  <<interface>>

  MarkovTextGenerator

  train(String)

  retrain(String)

  generateText(int)

- List<WordNode> wordList

Trees

Trees

- Trees are dynamic data structure : easy to add and remove
- Different organizations, different trees

Defining trees

Terminology

- Parent / Child
- Root Node: the only node has no parent node
- Leaf Node: nodes without children

What defines a tree?

- Single root
- Each node can have only one parent(except the root)
- No cycles in a tree (Cycle: two different paths between a pair of nodes)

Binary Trees

Generic Tree : Any Parent can have any number of children

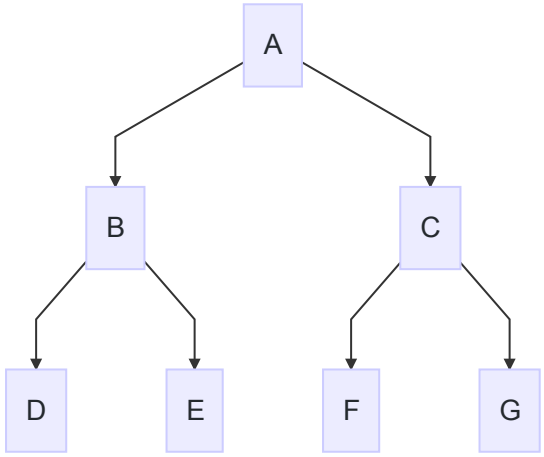
Binary Tree : Any Parent can have **at most** two children

- A tree just needs a root node (like head in LinkedList)
- Each node needs:
  - A value
  - A parent
  - A left child
  - A right child

```
public class BinaryTree<E> {
  TreeNode<E> root;

  public class TreeNode<E> {
    private E value;
    private TreeNode<E> parent;
    private TreeNode<E> left;
    private TreeNode<E> right;

    public TreeNode(E val, TreeNode<E> par) {
      value = val;
      parent = par;
      left = null;
      right = null;
    }

    public TreeNode<E> addLeftChild(E val) {
```

```
        left = new TreeNode<E>(val, this);
        return left;
      }
    }
  }
```

Traversal



Depth First Traversal : Pre-Order Traversals

Maze Traversal

- Go until hit a dead end, then retrace steps and try again.

- **Depth First Traversals**

Pre-Order Traversal : recursive process

- `ABDECFG`
- Visit yourself
- Then visit all your left subtree
- Then visit all your right subtree

```
private void preOrder(TreeNode<E> node) {
    if (node != null) {
      node.visit();
      preOrder(node.getLeftChild());
      preOrder(node.getRightChild());
    }
}

public void preOrder() {
    preOrder(root);
}
```

Post-Order

- `DEBFGCA`
- Visit all your left subtree
- Visit all your right subtree
- Visit yourself

In-Order

- `DBEAFCG`
- Visit all your left subtree
- Visit yourself

- Visit all your right subtree

Breadth First Traversal : Post-Order, In-Order, Level-Order

Social Network

- Look at all of my friends first, then find my friends' friends.
- **Breadth First Traversals**

Level-Order

- `ABCDEFG`
- Keep a list and keep adding to it and removing from start.*(Like a queue)*
- **FIFO: First-In, First-Out**

```
private void levelOrder() {
    Queue< TreeNode<E> > q = new LinkedList< TreeNode<E> >();
    q.add(root);
    while(!q.isEmpty()) {
      TreeNode<E> curr = q.remove();
      if(curr != null) {
        curr.visit();
        q.add(curr.getLeftNode());
        q.add(curr.getRightNode());
      }
    }
  }
}
```

Binary Search Tree - Recursion OR Iteration

Sorted arrays are good for search, but bad for insertion/ removal

Search

- Binary Tree
- Left subtrees are less than parent
- Right subtrees are greater than parent

Insert

- No rule that BSTs will be *full/ balanced* trees.
- Inserting a node means making it a child of an existing node

Deletion

- If leaf node: delete parent's link
- If only one child: hoist child
- If has multiple children
  - Find smallest value in right subtree : left reference is null
  - Replace deleted element with samllest right subtree value
  - Delete right subtree duplicate

HOW TO IMPLEMENT BST IS OFTEN-ASKED IN INTERVIEWS

Run Time Analysis of BSTs

- Structure of a BST depends on the order of insertion
- isWord(String wordToFind)
  - Start a root
  - Compare word to current node
    - If current node is null, return false
    - If wordToFind is less than word at current node, continue searching in left subtree
    - If wordToFind is greater than word at current node, continue searching in right subtree

    - If word at current node is eauql to wordToFind, return true
- **Best Case: O(1)**
- **Worst Case: O(n)** : Linear BST
- Performance also depends on the actual structure of the BST

Balanced BST

- Max Distance until Leaf : a.k.a. height
- |LeftHeight - RightHeight| <= 1
- Height around log(n)

\	Best Case	Average	Worst Case
Linked List	O(1)	O(n)	O(n)
BST	O(1)	O(log n)	O(n)*
Balanced BST**	O(1)	O(log n)	O(log n)

**Especially when insert to BST in order!*

***Use TreeSet in Java API to keep balanced*

Trie - reTRIEval

- Use the key to navigate the search through the structure
- Not all nodes represent words
- Nodes can have more than 2 children

Performance

- Finding a word depends on the height(i.e. length of the longest word)

Implementation

```
class TrieNode {
    private boolean isWord;
    HashMap<Character, TrieNode> children;
    private String text; //optional
}
```

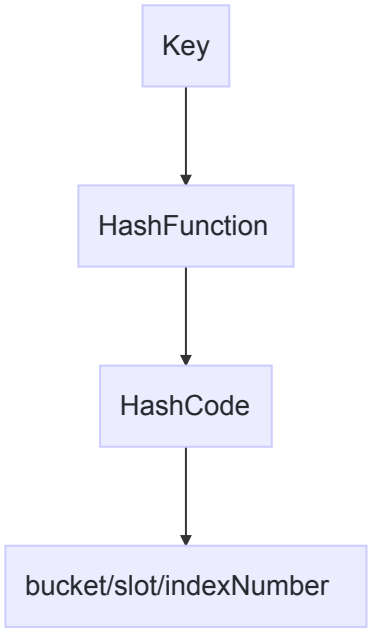
Autocomplete

- Find the stem
- Do a level order traversal from here

Hash Maps / Tables

Hash Tables

- Average:O(1) lookup; insert, and remove
- Condiser resizing costs and no data ordering



Modular Arithmetic

- Hashing an element into an index

Key	Hash Function	Hash Code
3	3 mod 5	3
11	11 mod 5	1
'a'	97 mod 5	2
"Hi"	(72+105) mode 5	2

**Key mod N(#elements in array) is a common hash function*

- A key part of creating hash functions is trying to minimize collisions

Challenge 1: Collisions in Hash Tables

Solution 1: Linear Probing : Insert

- Idea: Just out it in the next open spot
- Must check subsequent positions due to *linear probing*
- Linear Probing can struggle as the hash table starts getting full

Random Probing

- An alternative with tradeoffs
- Jump random # of steps instead of just the next

Solution2: Seperate Chaining

- Idea: Just keep a list at each spot
- Still have drawbacks

Challenge 2: Resizing

- **Rule of thumb: "Too full" is ~70%**
- When a hash table gets too full, the best thing to do is resize it
- Requires you create a new table, new hash function and reinsert everything!
- Resize cost

Challenge 3: Ordering data

- No ordering within the structure

Hash Set & Hash Map

Hash Set

- Tell you if an item is in the set or not

```
add(E e);
contains(Object o);
```

Hash Map

- Stores both a key and some data associated with the key (value)

```
get(Object key);
put(K key, V value);
```

Edit Distance

The number of modifications you need to make to one string to turn it into another.

tree grows too large:

- Dynamic programming -> O(k^2)
- Pruning: Restrict the path to only valid words