CSC115 Handbook: JAVA In a Nutshell

The mother of all cheat sheets

Written in 2014 by students: Carl Masri, Jakob Roberts, John McKay, Nathan Burrell

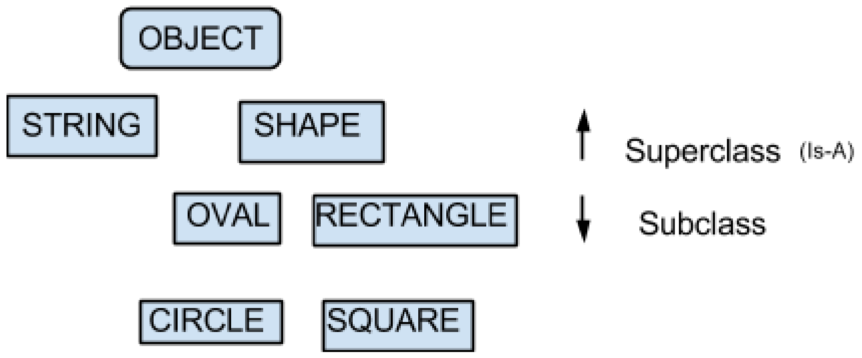
Table of Contents:

Roughly follows the final exam objectives sheet with some added information.

Java Objects and Heirarchy (Inheritance)

Relationships:

Is-A: All classes in Java inherit functionality from the Object class



Each class has an **IS-A** relationship with its **superclass**.
(eg. EVERY class in Java is an Object!)

Has-A: Each class has a Has-A relationship with its member variables.

Writing Subclasses for Abstract classes:

```
public abstract class Shape{
        public abstract double      getArea();
        protected abstract String getShapeName();
}
public class Rectangle extends Shape{
        public double getArea(){
                return length*width;
        }
        public     String getShapeName(){
                return "Rectangle";
        }
}
```

Explain the difference between an Overridden method and an Abstract method:
Overridden methods are methods that implement code for the abstract method (this code is contained within a different class that **extends** the superclass). In the example above, the Rectangle class's method getShapeName() overrides the abstract getShapeName() method in the Shape class. (Note: An abstract class can't provide an implementation for the method in that class that you're currently in BUT subclasses must provide an implementation or else the class itself becomes abstract! Alternatively, a class containing an abstract method must be declared as abstract)

Explain the difference between an interface and an abstract class:

- An interface is a way of defining expected behaviour without specifying any implementation details.
- When Implementing the is-A relationship, an interface can be considered "something" instead of just being classified as a reference to an object
- An interface has NO implementation, whereas an abstract class may have some implemented code. Any class that extends the abstract class must implement the code for these "unwritten" methods.

Classes <u>implement</u> Interfaces. Ex - public class Rectangle implements Box {
Classes <u>extend</u> Abstract Classes. Ex - public class Rectangle extends Shape {

Explain the relation between the Object class and other classes:
All classes are subclasses of the Object class.

Explain when a .equals() method must be used instead of an operator:
.equals() methods must be used to compare objects, as using the == operator will only compare the object references. The == operator is used to compare primitive types (int, double, float, long, boolean).

Write a .equals() method:
theObject is the type of object calling its .equals() method

```
public boolean equals(Object o){
        theObject x = (theObject) o;
        if (this.val1 != x.val1)
                return false;
        if (this.val2 != x.val2)
                return false;
        etc.
        else return true;
}
```

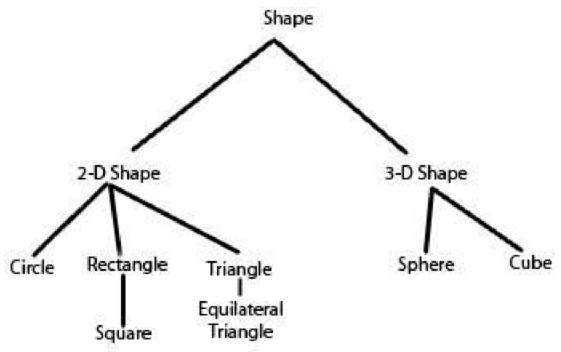
Protected Data Type: (not the same as private!)
Protected is a data type is similar to *private*, except this data is available to **all** the classes <u>in the inheritance hierarchy</u> (it is unable to be used by the"outside world" but **can** be used by subclasses. Private and protected are basically the same, <u>unless</u> inheritance is included).

i.e. protected data is available in: Shape, 2-D Shape, 3-D Shape, etc...

Understanding hierarchy processes

Suppose the only toString() method is given as:

```
public String toString(){
String s = getShapeName();
s+= " at position (" + xPos + "," + yPos + ") area: "
+ getArea();
        return s;
}
```



The process when s.toString(); is run is as follows:

- *Look inside the class type which **s** is*
    - s is a Circle → look inside the circle class
- There is no toString() method in Circle.
    - Move up the hierarchy
        - a Circle **is a** 2-D Shape → look inside the 2-D Shape class
- There is no toString method there.
    - Move up the hierarchy
        - a 2-D Shape **is a** Shape → look inside the Shape class
- There is a **toString** method in Shape
    - Execute the **toString()** method in Shape

Data Types:

- ADT:
    - no concrete information or any details about it but it is an idea, some examples are lists, queues, trees, etc.…
- Data Structure
    - An exact definition of the method names and details of the structure
    - Need to know all of the details in order to implement because it is implemented in a concrete way.
- Instance vs Static:
    - Instance:
        - Associated with a particular instance of an object. In order to invoke an instance method, you have to have an instance of an object.
        - eg: Student s = **new** Student("Jason");
        - Then s.getName(); where the "s" is the instance which references the student with the name "Jason"
    - Static:
        - Removed the need to instantiate objects because there is only one of the method that exists. Such as: int x = fib(7); where fib is a static method.

Generics:

List theList = new ArrayList();   // An example not using generics

List<String> stringList = new LinkedList<String>();
List<Student> studentList = new LinkedList<Student>();   // These all use generics
List<int> 13 = **NO PRIMITIVE TYPES IN GENERIC FORM! Use this instead:**
List<Integer> integerList = new LinkedList<Integer>();

****CANNOT MAKE A GENERIC TYPE ARRAY!****

```
public class Box<T> {
        private T x;
        public void add(T x) {
                this.x = x;
        }
        public T get() {
                return x;
        }

        public static void main(String[] args) {
                Box<Integer> integerBox = new Box<Integer>();
                integerBox.add(new Integer(10));
                System.out.printf("Integer Value: ", integerBox.get());
        }
}
```

The **Box class** is “delimited” by the inclusion of **<T>** ; that is a **Box** can be of any type.

The **Box class** is filled with objects of the generic type **T** which are initialized to a specified type at runtime.

When **Box<Integer>** is run, the main method runs through the **Box class** replacing all the generic **T objects** with **Integers**

The output of this code is:
Integer Value :10

Queues, Priority Queues, and Stacks

Definition of Queue ADT:
A queue is a linear storage of values, where only the last can be accessed (first in, first out).
The common methods in a queue are:

Enqueue - adds element to queue
Dequeue - remove and return *first* element in the queue
isEmpty - returns true if the queue is empty

Ex:
enqueue (“lol”)   →   [lol, stack, what]   →   dequeue   →   lol
enqueue (“stack”)   dequeue   →   stack
enqueue (“what”)

Definition of Priority Queue ADT: *(is NOT a heap but can be implemented with one)*
Priority Queues work like queues and stacks, but each element has a priority associated with it. The common methods in a priority queue are:

- insert - adds element to queue
- removeMax - remove and return the highest priority item from the queue
- isEmpty - returns true if the queue is empty

LL is best implementation

Priority Queue ADT**	**Unsorted Array***	**Sorted Array***	**Unsorted LL**	**Sorted LL**
insert	O(1)	O(N)	O(1)	O(N)
removeMax	O(N)	O(1)***	O(N)	O(1)
isEmpty	O(1)	O(1)	O(1)	O(1)

* - Array cannot get full (Does not count growAndCopyArray())
** - Note that this is a priority queue BigO table. The regular queue BigO table is below
*** - If the array is sorted properly (max must be at the end of the array)

Definition of Stack ADT:
A stack is a linear storage of values, where only the first can be accessed (last in, first out). The common methods in a stack are:

Push - pushes an element on top of the stack
Pop - remove and return value of the top element in the stack
isEmpty - checks if the stack is empty
Peek - Looks at the top element in the stack without removing

Write array based and linked-list based implementations for methods that check if a Queue is empty, that enqueue to and dequeue items from a Queue: (pg 415 - 424 in text)
LinkedList Implementation: (using generics)

```
public void enqueue(T element) {
        theList.addBack(element);
}

public T dequeue() {
        if (theList.size() != 0) {
                return theList.removeFront();
        } else {
                throw new QueueEmptyException();
        }
}

public boolean isEmpty() {
        return (theList.size() == 0);
}
```

Write array based and linked-list based implementations for methods that check if a Stack is empty, that push to and pop items from a Stack: (pg 365 - 369, 372 in textbook)
Linked List Implementation (using generics):

```
public boolean isEmpty()  {
            return theList.size() == 0;
 }

public void push(T element)  {
            theList.addBack(element);
 }

public T pop() throws StackEmptyException  {
            if (theList.size() == 0)  {
                        throw new StackEmptyException();
             }
             return theList.removeBack();
 }
```

Applications of Queue and Stack ADT:
Queues are used when you want the first item in the queue first (eg call center).
Stacks are used when you want the last data value inserted into the list (eg browser back button)

Queue Methods: isEmpty(), enqueue(item), dequeue()
Stack Methods: isEmpty(), push(item), pop()

Indicate and Defend the running time of each of the above Stack and Queue methods.

Queue ADT	**Array**	**LinkedList**
enqueue(item)	O(1)	O(1) *
dequeue()	O(N) **	O(1)
isEmpty()	O(1)	O(1)

* - if the list has a tail reference, O(N) otherwise
** - because the queue has to be shifted over after a dequeue operation

Stack ADT	**Array**	**Linked List**
push(item)	O(1) *	O(1) **
pop()	O(1) *	O(1) **
isEmpty()	O(1)	O(1)

* - if adding to / removing from the back of an array
** - adding to / removing from the front of a singly linked list

<u>Exceptions</u>

Write exception classes:

```
public class HeapFullException extends RuntimeException {
}
```

Call methods of exception classes:
Exception classes generally only have constructor methods. They are called when the exception is thrown.

Throwing exceptions:

```
public static void main(String[] args) throws SomeTypeOfException {
```

*Exceptions must be thrown in each successive method as well. For example, if the exception is first thrown in some method, lets call it MethodOne throws Exception, and MethodOne is called from another method called MethodTwo (which is in turn called by main), MethodOne, MethodTwo <u>and</u> main must ALL throw the exception.

Try / Catch Exceptions:

```
try {
        // Some code that may throw a SomeTypeOfException
} catch (SomeTypeOfException e) {
        //Some code that responds to the exception
        //Usually tries to resolve the issue which caused the exception
}
```

<u>Binary Trees, Binary Search Trees, and Heaps</u>

Binary Tree vs. Binary Search Tree vs. Heap:
Both Binary Search Trees and Heaps are types of Binary Tree.

- A <u>binary tree</u> is a tree such that for all nodes in the tree, each node has *at most* two children.
- A <u>Binary Search Tree</u> is a tree where for all nodes **n** in the tree:
    - the value at n is <u>greater</u> than all nodes in the *left* subtree AND
    - the value at n is <u>less</u> than all nodes of n in the *right* subtree
- A <u>Heap</u> is sorted from top to bottom where each node is <u>bigger **or** smaller</u> than both its children depending on it being a <u>max or min</u> heap respectively.

Node: an element in the tree
Leaf: a node with no children
Internal Node: a node that has parents and children
Root: “special” internal node which is the top and has no parent (can be a leaf if only root in tree)
Height of Tree: the number of nodes on the path from the root to the deepest node.

Depth of Node: the number of nodes on the path from the node to the root

Min Heap: Where the top-most element is the smallest element in the heap.
parent <= children
Max Heap: Where the top-most element is the biggest element in the heap.
parent >= children
How to find in a heap: *(using arrays, the entry at location 0 is left empty)*
Parent: i/2
Left Child: i*2
Right Child: i*2 +1

Determine the height of a specified binary tree: ***KNOW THIS***

For a **full** binary tree with N nodes, $\mathbf{N = 2^h - 1}$. Alternatively, $\mathbf{h = \log_2(N+1) \sim \log_2(N)}$.

The number of NODES on the path from the deepest node to the root is the height.
If the tree is empty, height = 0.
If the tree contains ONLY one node, the height is 1.
Basically, just count the levels in the tree, starting at 1 (the root). If there is no root, the height is zero.

Binary Search Tree Height Algorithm (Recursive):
```
class BinarySearchTree{
        TreeNode root;
        public int height(){
                return height(root);

        int height(TreeNode n){
                if (n==null) return 0;
                        int lh  height(n.left);
                        int rh = height(n.right);
                if (lh>rh) return 1+lh;
                else return 1+rh;
}
```

Explain the difference between full, complete and balanced binary trees:
Full:
- If the tree is empty, it is a full binary tree of height = 0; If the tree is not empty and has a height H > 0, it is a full binary tree if the subtrees of its root are both full binary trees of height (H-1). In other words, each parent node has TWO children.

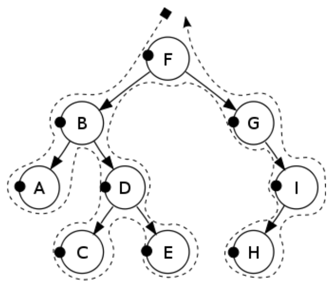
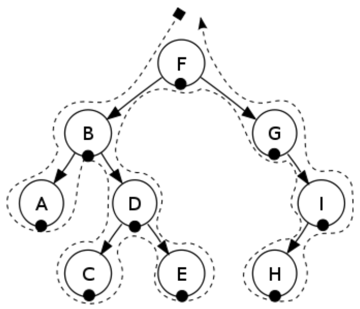
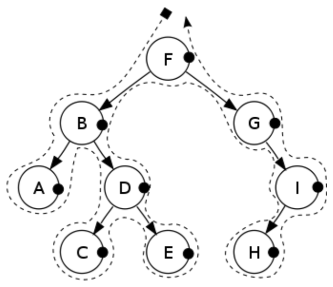
Complete: A binary tree of height H is complete if:
- All nodes at depth H-2 and above have 2 children.
- When a node at depth H-1 has children, all nodes to its left at the same level have 2 children.
- When a node at depth H-1 has only one child, it is a left child.
- If the tree is full, it is also complete.
- If the tree is not full, all the nodes on the last level are “pushed” as far left as possible.

Balanced: (We don’t really need to know this, but hey why not)

- A binary tree in which the depth of the left and right subtrees of every node differ by 1 or less, although in general it is a binary tree where no leaf is much farther away from the root than any other leaf.

Be able to traverse binary trees in pre-order, in-order, post-order: (pseudo-code)

```
traverse(TreeNode n){
        if(n==null) return;
        System.out.println(n.item); // If pre-order
        traverse (n.left);
        System.out.println(n.item); // If in-order
        traverse (n.right);
        System.out.println(n.item); // if post-order
```

Pre-Order	In-Order (or Symmetric)	Post-Order
		
Output: F, B, A, D, C, E, G, I, H	Output: A, B, C, D, E, F, G, H, I	Output: A, C, E, D, B, H, I, G, F

Indicate and defend the running time of each of above traversal methods:
All have the same running time: **O(N)**
Defence: Worst case scenario, each node in the tree must be traversed.

Analyze array and reference based implementations of a heap, including methods for: creating, inserting & deleting. In particular, be able to explain why the big O running times of the heap insert and delete methods are O(log N). (Hint: the argument should be based on the relationship between the number of nodes in a full tree and its height):
Min/Max heaps are *complete binary trees*. As such, the height of the tree with respect to n is N = 2^h - 1. When using an array implementation of (for example), a max heap, inserting values into the heap and deleting values from the heap is O(logN). The reason for this lies in the heap being stored as an array. Arrays in Java can be accessed in constant time. So, as long as we know where the item in the array is supposed to be inserted or deleted from, we can just jump to that spot and insert/delete as needed. Although inserting and deleting the value can be completed in constant time, we're not finished. Max and min heaps require the nodes to be in a certain order. This means that once the node has been inserted/deleted, the tree has to be "re-shaped" - nodes need to be bubbled up/down until their correct placement has been found. The number of operations required is dependent on the number of levels the node must rise/fall to satisfy the heap property. Worst case scenario, the node would have to be moved from the very bottom of the tree to the very top (or vice versa, taking 'tree height' operations). The height of a full binary tree is h = $\log_2(N+1)$ (complete is similar). Because BigO ignores constants, the worst-case time is O(logN).

Indicate and defend the running time of each of above Tree methods:

Heaps	Array	Object Reference (e.g.Node-based)
Creation	O(1)	O(1)
Insertion	O(logN) *	O(N)
Deletion (deleteMin or deleteMax)	O(logN) *	O(N)

* - Defence for these can be found just above this table

Full Binary Search Tree Implementation:

```
public class BinarySearchTree{
        TreeNoderoot;
        public BinarySearchTree(){
                root = null;
        }
        public boolean      find (int element){
                TreeNode p = root;
                for (;;){
                        if (p == null)
                                return false;
                        if (element < p.element)
                                p = p.left;
                        else if (element > p.element)
                                p = p.right;
                        else
                                return true;
                }
        }
        public boolean insert (int element){
                TreeNode n = new TreeNode(element);
                if (root == null){
                        root = n;
                        return true;
                }
                TreeNode p = root;
                for (;;){
                        if (element == p.element)
                                return false;
                        if (element < p.element){
                                if (p.left == null){
                                        p.left = n;
                                        return true;
                                }else{
                                        p = p.left;
                                }
                        }else{
                                if (p.right == null){
                                        p.right = n;
                                        return true;
                                }else{
                                        p = p.right;
                                }
                        }
                }
        }
```

```
        private void doToString (TreeNode node, int indent,StringBuilder
                output)
                {
                        if (node!= null)
                        {
                                doToString(node.right, indent+2,
                output);
                                for(int i=0; i<(indent*2); i++)
                                        output.append(" ");
                                output.append(node.element);
                                output.append("\n");
                                doToString(node.left, indent+2,
                output);
                        }
                }

        public String toString(){
                StringBuilder s = new StringBuilder("");
                doToString(root, s);
                return s.toString();
        }
        // Recursive height functionality from above
        public int height(){
                return height(root);
        }

        public int height(TreeNode n){
                if (n==null) return 0;
                        int lh = height(n.left);
                        int rh = height(n.right);
                if (lh>rh) return 1+lh;
                else return 1+rh;
        }
        public static void main (String[] args){
                BinarySearchTree t = new BinarySearchTree();
                if (args.length == 0){
                }else{
                        for (int i = 0; i < args.length; i++){
                                t.insert(Integer.parseInt(args[i]));

                        }
                }
                System.out.println("Tree height is : " + t.height());
                System.out.println(t);
        }
```

	}

Linked Lists And Resizable Arrays:

***Remember, an array is a special type of object and uses a reference! Also has a length variable.

Trace and write linked lists, including singly and doubly linked lists and circular lists:
Nope.

**Be able to create linked lists and insert into, delete from and traverse them.
Determine the big O of each operation.**

Linked Lists	Insertion	Deletion / Find	Traversal
Singly	O(N)*	O(N)	O(N)
Doubly	O(1)**	O(N)	O(N)
Circular (singly linked with a reference to the “tail”)	O(1) (addFront and addBack)	O(N)	O(N)

*addFront is O(1)
**addBack is O(N) if there is no tail reference

Printing Linked Lists: (example)

```
Node p = head;
while(p!=null){
        System.out.println(p.value);
        p=p.next;
}
```

List ADT	addFront	addBack	get	remove
Array	O(N)	O(N)*	O(1)	O(N)
Linked List	O(1)	O(1)**	O(N)	O(N)

* If the array is not allowed to get full (has to call growAndCopyArray()). If the array size is declared at maximum size then this is O(1).
** Has to have a tail reference

Singly Linked List Code:

Keep in mind: Double Linked Lists also have a tail reference, and an addBack(item) method.

```java
import java.util.*;
public class SLL<T> implements GeneralList<T>, Iterable<T>
{
        private Node<T> head;
        private int count;
        public SLL(){
                head=null;
                count=0;
        }
        public void addFront (T x){
                head=new Node<T>(x,head);
                count++;
        }
        public void addBack (T x){
                Node<T> newNode=new Node<T>(x);
                count++;
                //when it is empty
                if(head==null){
                        head=newNode;
                        return;
                }
                Node<T> temp=head;
                while(temp.next!=null){
                        temp=temp.next;
                }
                temp.next=newNode;
        }
        public int size(){
                return count;
        }
        public T  get (int pos){
                if(head==null)
                        return null;//throw exception
                Node<T> temp=head;
                for (int i=0; i<pos; i++){
                        temp=temp.next;
                }
                return temp.item;
        }
        public void clear(){
                head=null;
                count=0;
        }
```

Page 1

```java
public void remove (T value){
        if(head==null) return;
        Node<T> current=head;
        Node<T> previous=head;
        while(current!=null)  {
                if (current.item.equals(value))  {
                        if (current==head){
                                head=current.next;
                                previous=current.next;
                        }else{
                                previous.next=current.next;
                        }
                        current=current.next;
                        count--;

                }else{
                        previous=current;
                        current=current.next;
                }
        }
}

//create an instance of SLLIterator and return it
public Iterator<T> iterator()
{
        return new SLLIterator();
}
```

Page 2

```java
//SLLIterator class implements java.util.Iterator interface
//example of an inner class
private class SLLIterator implements Iterator<T>    {
        private Node<T> nextNode;
        public SLLIterator()  {
                this.nextNode = SLL.this.head;
        }
        public boolean hasNext(){
                return nextNode != null;
        }
        public T next(){
                if (this.hasNext()){
                        T returnVal = nextNode.item;
                        nextNode = nextNode.next;
                        return returnVal;
                }
                throw new NoSuchElementException();
        }
        public void remove(){
                throw new UnsupportedOperationException();
        }
}
public String toString(){
        StringBuilder output=new StringBuilder("{");
        Node temp=head;
        while(temp!=null){
                output.append(temp.item);
                if(temp.next!=null)
                        output.append(',');
                temp=temp.next;
        }
        output.append('}');
        return output.toString();
}

public static void main(String[] args){
        SLL<Integer> list=new SLL<Integer>();
        int[] input={1,2,3,4};
        //test addBack
        for (int i=0; i<input.length; i++){
                list.addBack(input[i]);
        }

        System.out.println(list);
        Iterator<Integer> it = list.iterator();
        while(it.hasNext()){
                System.out.println(it.next());
        }
        System.out.println("\n");
}
```

Page 3

```java
        // Another inner class
        private class Node<T>
        {
                public T item;
                public Node<T> next;

                public Node(){
                        item=null;
                        next=null;
                }
                public Node(T n)     {
                        item=n;
                        next=null;
                }
                public Node(T n, Node<T> nextNode ){
                        item=n;
                        next=nextNode;
                }
                public T getItem(){
                        return item;
                }
                public void setItem(T newItem){
                        item=newItem;
                }
                public Node<T> getNext(){
                        return next;
                }
                public void setNext(Node<T> nextNode){
                        next=nextNode;
                }
        }
}
```

Page 4

Be able to create resizable arrays and insert into, delete from and display them. Determine the big O of each operation.

```java
        public static void growAndCopyArray(){
                Song[] newOne = new Song[storage.length*2];
                for (int i = 0;i<songCount;i++)
                {
                        newOne[i] = storage[i];
                }
        storage = newOne;
    }
```

BigO Runtime for growAndCopy = **O(N)**
Insert and Delete (for arrays) are also **O(N)** if all values must be moved after insertion/deletion, (such as in a sorted list)

Efficiency

Searches:

- Binary Search: **O(logN)**
    - start in center so length/2
    - if value you are searching for is greater than this middle value, then it can't be in the bottom half of the list, so continue splitting in half!
    - each comparison to n by splitting in half reduces the size of the searchable addresses by a factor of 2
        - say N=1,000,000, one comparison brings you down to 500,000 etc…

```
int binarySearch(int[] n, int val){
    int checkval;
    int low=0;
    int high=n.length-1;

    while(low <= high){
            int mid  (low + high)/2 // problem here, low+high can hit neg.
            /* Possible fixes:
             * int mid = low + ((high-low) /2);
             * int mid = (low + high) >>> 1;
             */
            intmidVal = a[mid];

            if (midVal < key){
                    low= mid + 1;
            }else if (midVal > key){
                    high = mid - 1;
            }else{
                    return mid; //key found
            }
    }
    return -1; //key not found
}
```

Sorting:

- Bubblesort = **O(N^2)**
- Insertion sort/priority queue sorting = **O(N^2)**
    - Linked list: **O(N^2)**
    - Heap: **O(N*logN)**
- Quicksort = **O(N^2**)

<u>Hashing</u>

Hash function:
- Maps from keys to integers.
- Goal is to be constant time *with respect to* the number of elements in the list, but is not usually constant time with respect to the length of the key.
- Hashing works best when the keys are small values with respect to the total number of things in the list.

How it works:
Inserting:
insert ("Jason")→hash("Jason")→returns hash value: 1→"Jason" is stored at index 1 in the array
insert ("Jane")→hash("Jane")→returns hash value: 7→"Jane" is stored at index 7 in the array
Finding:
find("Jane")→hash("Jane")→returns hash value: 7→look in index 7 for "Jane"→is "Jane" at index?

A good hash function distributes keys <u>uniformly</u> across the range of hash values.

However, hashing has a problem. It's almost guaranteed that two keys are going to map to the same hash value. This is called a **collision**. So, how does hashing deal with collisions?
Two ways:
Open Addressing (Linear Probing): When a hash value is calculated for a key and that space is already taken within the array, go to the next available index down the array. Of course, the worst case for this would be that the entire array would have to be traversed in order to find an available slot.
Separate Chaining: Each slot in the array is a *reference* to a linked list. Keys that map to the same hash value (index) are then chained along the list at the index.

The load factor: **λ = N / Size** (N = # elements in the array, Size = length of the array)
For open addressing: **λ <= 0.5**
For separate chaining: **λ <= a "small" constant C***
*<u>Constant time for separate chaining</u> is expected when the length of the linked lists are constant with respect to the number of elements in the array.

BigO Hash Table:

	Array	**Linked**	**Binary Search Tree**	**Balanced BST**	**HASHING***
insert	O(N)	O(1)	O(N) (Expect logN)	O(logN)	EXPECT O(1)
find	O(logN)	O(N)	O(N) (Expect logN)	O(logN)	EXPECT O(1)

*Hashing expects constant time, but that is only under some very specific assumptions, the actual BigO runtime is O(N). Or infinity if your hash function really sucks.

Basic Scanner code in from file: (examples)

```
import java.util.*        // for scanner
import java.io.*          // for file, filenotfoundexception

File inFile = new File("songs.txt");
        Scanner sc = new Scanner(inFile);
and
        public Song (Scanner s)
        {
                name = s.nextLine();
                artist = s.nextLine();
                album = s.nextLine();
                filename = s.nextLine();
                playCount = 0;
                rating = 0;
        }
```

Grow and copy code: (example)

```
public static void growAndCopyArray(){
        Song[] newOne = new Song[storage.length*2];
        for (int i = 0;i<songCount;i++)
        {
                newOne[i] = storage[i];
        }
        storage = newOne;
}
```

Recursion

A recursive method is one that calls itself.

. In order for a problem to be solved recursively:

1. There must be at least one base case; a part of the problem that solves without recursion
2. Each recursive call must make progress towards the base case
3. Believe it works

Summing everything in a linked list using recursion: (example)

```
int sum() {
        return sum(head);
}
int sum(Node n) {
        if ( n == null)
                return 0;
        if ( n.next == null)
                return n.value;
        else
                return n.value + sum(n.next); //keeps going and going!
```

Tracing recursion:

```
public class MoreRecusion{
        public static void CountDownByTwo(int n){
                if(n>=0){
                        System.out.println(n);
                        CountDownByTwo(n-2);
                }
        }
        public static void main (string[] args){
                CountDownByTwo(5);
                CountDownByTwo(4);
        }
}
```

BOXTRACE

main	**CountDownByTwo(n)**

CountDownByTwo(5)
CountDownByTwo(4)
 	
output
5
3
1
4
2
0
	n = 5
	System.out.println(5);
	→ n = 3
	→ System.out.println(3);
	→→ n = 1
	→→ System.out.println(1);
	→→→ n = -1
	←——————
	n = 4
	System.out.println(4);
	→ n = 2
	→ System.out.println(2);
	→→ n = 0
	→→ System.out.println(0);
	→→→ n = -2
	←——————

Big O Notation (detailed)

- **Big O Runtime:** Worst case scenario for the runtime of a program.
- We are interested in the growth rates of our algorithms. As the number of elements/input increases, how does the number of operations increase?

Big O is useful when the input (N) is large

We don't care when N is small

$125\ N^3 + 2700\ N^2 + \ldots$

only care about the biggest power which is the fastest growing term in relation to N

→ that would be $\mathbf{N^3}$

When counting operations we only care about <u>non constant</u> operations
(typically loops or recursion)
Some examples:

```
for (int i=0;i<N;i++){
        int x;                          is O(N)
        x = 4*i + N;
        System.out.println(x);
}
```

```
for(int i=1;i<N;i++){                   Two loops, each O(N)
        for(int j=0;j<N;j++){           O(N)*O(N) is O(N^2)
                //constant stuff
        }
}
```

```
for(int i=0;i<N;i++){           // O(N)
        for(int j=0;j<i;j++){   // Summation, i=1 to N from i → (n(n+1))/2
                //constant
        }                       is O(N^2)  due to (n(n+1))/2 having a biggest term N^2
}
```

<u>**CompareTo implementation**</u>

In short:
this.compareTo(that)
returns:

- a negative int if this < that
- 0 if this == that
- a positive int if this > that

<u>**Fibbonacci code:**</u>

<u>Recursive</u>

```
static int fib (int n){
        if (n==0||n==1)
                return 1;
        else
                return fib(n-2) + fib(n-1);
}  /// BUT DO NOT DO THIS, is O(2^N) and gets big REALLY FAST!!!
```

<u>Iterative</u>

```
static int fib (int n) {
        int fib = 0;
        int a = 1;
        for (int i=0; i<n; i++) {               // This is O(N)
                fib = fib + a;
                a = fib;
        }
        return fib;
}
```

Harmonic code:

```
    public static double harmonic (int n) {
        if (n == 1)
            return 1.0;
        else
            return 1.0/(double)n + harmonic (n-1);
    }
```

Factorial code:

```
    public static int factorial (int n) {
        for (int i = 1; i <= n; i++) {
            result *=i;
        }
        return result;
    }
```

Post-Fix Math Expressions:

```
    private double evaluate(TreeNode node)throws InvalidExprException{
        if(isNum(node.item)){
            return convert(node.item);
        }
        if(isOperator(node.item)){
            double l = evaluate(node.left);
            double r = evaluate(node.right);
            char c = node.item.charAt(0);
            switch(c){
                case '+':
                    return l + r;
                case '-':
                    return l - r;
                case '*':
                    return l * r;
                case '/':
                    return l / r;
                default:
                    throw new InvalidExprException("Invalid Expression.");
            }
        }
        throw new InvalidExprException("Invalid Expression.");
    }
```

Recursive implementation of insert for a Binary Search Tree:

```
    private STreeNode insert(STreeNode node, String str){
            if(isEmpty()){
                root = node;
            }else{
                if(0 > str.compareTo(node.item)){
                    if(node.left == null){
                        node.left = new STreeNode(str);
                    }else{
                        insert(node.left, str);
                    }
                }else if(0 < str.compareTo(node.item)){
                    if(node.right == null){
                        node.right = new STreeNode(str);
                    }else{
                        insert(node.right, str);
                    }
                }else{
                    throw new TreeException("Duplicate string, cannot add to Tree.");
                }
            }return null;
```

```
        }
```

Heap Bubble Down Method:

```
private void bubbleDown()   {
        int index = 1;
      // Finding smallest child
        while(hasLeft(index))  {
                int smaller = leftChild(index);
            // If parent has a right child, this checks to see if it is smaller
            // than the left child. The smaller is chosen.
                if (hasRight(index))  {
                        if (storage[leftChild(index)].compareTo(storage[rightChild(index)]) > 0)  {
                                smaller = rightChild(index);
                        }
                }

            // If the parent is greater than the smallest child, the two
            // are swapped.
                if (storage[index].compareTo(storage[smaller]) > 0)  {
                        swapElement(index, smaller);
                }
            // Updates index.
                index = smaller;
        }
}
```

Heap Bubble Up Method:

```
private void bubbleUp()  {
        int index = currentSize;
     // While the current element has a parent and it is larger,
     // the child is swapped with the parent.
        while (hasParent(index) && (storage[parent(index)].compareTo(storage[index]) > 0))  {
                swapElement(index, parent(index));
                index = parent(index);
        }
  }
```

RemoveMin:

Check if the heap is empty (if it is, throw an exception). Store the root node to return it later. Place the last node in the heap structure at the root. Call bubbledown method. Return the stored root.

Insert:

Check if the heap is full (if it is, throw an exception). Check if the heap is empty (if it is, add the element into the array at position 1). If the heap isn't empty, add the new element to the first non-empty spot in the array (skipping the first spot, which is always empty). Call the bubbleup method.