2/1/2022 Notes: OOP Applications

Outline

- Java wrapper classes
- Class string
- Exception Handling Classes and mechancism
- Classes for input/output from/to files

"student learning outcomes"

- Java wrapper classes
- use java string class methods
- Use exception handling
- Read write from files

Each type will have a primitive type will have a wrapper class. The wrapped primitive type will provide an interface to allow but better manipulation of the datatype.

For example: The integer class allows for casting of a string or integer to be an object of type integer.

```
parseInt(String, int);
///This will cast the string but with an included number system.

System.out.println("111",2);
//7

System.out.println("The maximum integer is "+Integer.MAX_VALUE);
//Proffessor scrolled down but big number
```

Calling Integer allows for use of the methods

Boxing: COnverting from primative type to warpper class

- type (int-> Integer)

```
//Automatic boxing
Integer[] intArray={1,2,3,};

//Automatic UNboxing
int x = intArray[0]
```

The main use for a warpper class is to better manipulate primative types

String class

- Class to manipulate text - String
- String Objects are immutable (cannot be changed once created)
- Wide set of methods to manipuate String objects (13 constructors and 40 methods)

String has lots of methods and constructors, I'm, not going to bother memorizing them.

Split(String): String[]

Returns array of the string between space.

```
+replaceFirst(String regex, String): String
all
split
match
```

regex: regular expression - general pattern in the string

Regular Expressions

- Used to describe a general pattern in a text
- Analyze text for specific patterns - validate user input for example

  phone number (ddd) ddd-dddd Social Secutriy Number ddd dd dddd

```
"java.*" * /stands for any zero or more characters

"\\d{3}-\\d{2}-\\{4}" // \d is a single digit {2} number of digits

"[$+#%]" // any of the included characters can be used
```

Regular expressions cheat sheet

```
"2+3-5".replaceFirst("[+-*/]","%");
//returns 2%3-5

String[] items =
"02/25/2021".split("/");
//returns items = {"02","25","2021"}

String[] tokens =
"Java,C?C#,C++".split("[].,?,#]");


"2+3-5".matches("\\d[+-]\\d[+-]\\d");
// returns true
```

```
"2+3-5".equals("\\d[+-]\\d[+-]\\d");
//returns false

"044-03-534".matches("\\d{3}-\\d{3}-\\d{3}")
```

Examples:

```
System.out.println("Hi,ABC, good".matches(".*ABC .*"));
//false
```

StringBuilder is a class that can create mutable classes.

Exceptions

- **Exceptions** - runtie error thrown by the program. - causes the program to stop immediately.
- **Handling an Exception** - Avoid immediate termination -infrom the user - continue program or exit with friendly messags.

Mechanisms for handling exceptions

- **Try Block** - chode block where the excpation is thrown
- **Catch** - code block to execute when exception is thown in try

Catch Block

- Like a method - called when an exception is thrown in the try block
- Never returns to the try block
- Has one parameter of type throwawble, which is the super clas of all exceptions.
    - IOexceptions and classnotfoundexception are general excpetions while other would be considered Runtime Exceptions

Knowing specific expceptions allows for more specific catch blocks to be executed

Class throwable has:

```
String message;
String getMessage();
String toString();
String printStacktrace();
```

Throwing Exceptions

- Exceptions are thown bt speciic methods or operations (nextInt(),/)
- Programmer can use throw to "throw" their own exceptions in the code.

```
throw new Exception("Something went Wrong");
```

Throwing new exception created can be used to check for specific entered data, as in a valid type but wrong range.

```
Student s = new Student();
processStudent(s);

processStudent(new Student());
//parameter would be considered an anaymous object
```

```
try {

    //...
    Exception e;
    e=new Exception("Under 30.");
    throw e;
}
catch(Exception ex) {
    System.out.println(ex.getMessage);
}
```

Creating New Exception Classes

- You can create your own exception classes
- Programmer-created excpetions must be derived from Java Exception classes
- Derived classes must have two constuctors at least

```
public class InvalidGPAExcpetion extendes Exception {

    public InvalidGPAException() {
        super("Invalid GPA Exception");
    }
    public void getMessage()
    {
        System.out.println("Gpa in invalid. GPA: "+getMessage());
    }
}
```

get notes from last chapter

2/3/2022 Notes: Exception Blocks continued

To Do

- Finish Assignment 1
- Check bookwork

Multiple Catch blocks

- Each Catch block is associated with a specific tyoe if exception (type of parameter e)
- A try block may throw exceptions of different types
- Multiple catch blocks - one for each type of exception thrown
- Order of caatch blocks matters
- From specific to general
- Follow the hierarchy of inheritance from sub classes to super classes

Catch-Declare Rule

- nextInt() throws an exception and does not handle it-(declare rule)
- The caller of nextInt() decides to handle the exceptino or not
- You can also create methods that throw exceptions and handle them(catch rule)
- **Declare rule** : use the clause 'throws'

```
public void safeDIvide(int a,int b) throws DivisionByZero
```

```
Int safeDivide(int a, int b)
    throws DizisionByZeroException
    {
        try{
        if(b==0)
            throw new DivisionByZeroException();
        else
            return (a/b);

        }
        catch(DivisionByZeroException e {
            return 0;
        })

    }
```

- Checked Exceptions - Exception for which Java enforces the rule catch or declare
- Unchecked Exception - Exception not checked by java or catched

I/O excpetions are consifered to be Checked exceptions

Finally Block

- A block after the try block and all its catch blocks
- The finally block is always executed whether an excpetion is thrown or not

```
igh University Spring 2022
CSE-017
57
Exception Handling public class FinallyDemo {
    public static void main(String[] args){
        try { exerciseMethod(0);}
        catch(Exception e){
        System.out.println("Caught in main."); }
 }
 public static void exerciseMethod(int n) throws Exception {
    try{
        if (n > 0)
            throw new Exception();
        else if (n < 0)
            throw new NegativeNumberException();
        else
            System.out.println("No Exception.");
        System.out.println("Still in exerciseMethod.");
    }
    catch(NegativeNumberException e) {
        System.out.println("Caught in exerciseMethod.");
    }
 finally{
    System.out.println("In finally block.");
    }
    System.out.println("After finally block.");
 }
}
//if n>0
//In finally block
//Caught in main
```

Summary

- Exception Handling - try - catch - throw - finally
- Deriving new exception classes
- Declare exception - throws
- Catch or declare rule - checked/unchecked exception

File I/O

- Accessing fiels on your hard disk or remotely
- Access file properties(size, location/folder/file..)

Class file

- Wraffer Class for files
- Allow access to file properties
- Does file exist....etc

 File Class

Reading/Writing to files

- open the file
- read the file
- close the file

Open Files for reading

- Create a Scanner object linked to a class File object- Class FIle object is linked to the file
- Constructor Scanner(File) throws a checked FileNotFoundException

```
File file = new File ("data.txt");
Scanner fileScanner =  new Scanner(file):
```

Open Files for writing

- Create a printwriter object linked to a class file object - Class File object is linked to the file
- PrintWriter(File) constructor throws a checked "FileNotFoundException"
    - in output means you cannont write to file

```
File file = new File("output.txt");
PrintWriter fileWrite = new PrintWriter(file);
```

Reading from a file

- Use a scanner methods nextInt(), nextDouble(), next(), nextLine()

Writing to file

- Use Print Writer methods print(), println(), printf()

Close Files

- close() method from class Scanner

```
filScanner.close();
```

- Close() method from class PrintWriter

```
fileWrite.CLose
```

```
//needed imports for file managemnet
import java.util.Scanner;
import java.io.File;
import java.io.PrintWriter;
import java.io.IOException;
```

```
try {
    studentCount = input.nextInt();
    studentList = new Student[studentCount];
    Scanner readFile = new Scanner(file);
    System.out.println("File opened successfully.");
    for(int i=0; i<studentCount; i++) {
        String fname, lname; int id; double gpa;
        fname = readFile.next();
        lname = readFile.next();
        id = readFile.nextInt();
        gpa = readFile.nextDouble();
        studentList[i] = new Student(fname + " " + lname, id, gpa);
        System.out.println("Student " + (i+1) + ": " +
studentList[i].toString());
    }
    readFile.close();
}
catch(InputMismatchException e) {
    System.out.println("Input Format Error."); System.exit(0);
}
catch(FileNotFoundException e) {
    System.out.println("Cannot open file \"students.txt\"");
}
```

2/8/2022 Abstract Classes and Interfaces

Outline

- Polymorphosm
- Dynamic Binding
- Abstract Classes
- Interfaces

Goals

- Explain polymorhpism and dynamic binding
- Create and extend abstract classes
- Use interfaces to model common behavior between classes and multim=ple inheritance

Polymorhism

- Every object of a derived class is an object of that base class
- Polymorhosm a variable of a super type that can refer to a sub type object

Dynamic Binding

- JVM (Java Virtual Machine) decided which method is invoked at runtime
- Variable have a devlared type and an actual type
- Methods are called on the actual type.

```
Object obj = new Circle();
System.out.print(obj.toString());
//Compiler looking to see if type Object has toString, then on runtime
calling circle toString.
```

```
public static void printArray(Object[] list)
{
    for(Object o: list)
        System.out.print(o.toString() + " ");
    System.out.println();
}
//This is considered to by a generic method
```

Object Casting

- An object of the sub class is an object of the super class
- An object of the super class is not an object of the sub class

Cannot cast up inheritanec normally

Up casting `Object obj = new Circle();`

Down Casting

```
Circle c = obj; //Error
Circle c = (Circle) obj; //down casting
```

- If the actual type of obj is not Circle, ClassCastExcpetion is thrown

- obj1 == obj2 compare the referances to obj1 and obj2
- obj1.equals(obj2) should compare the attributed of the two objects
- Method `equals()` in class Object compares references only
- Must override `equals()` in every class where you need to compare object attributes

```
public boolean equals(Object obj) {
    if (obj instanceOf Circle) {
        Circle c = (Circle) obj;
        return (radius == c.getRadius());
    }
    else
        return false;
}
```

Abstract Classes

- Abstract class - common behavior for related sub classes
- Interface - common behavior for classes not necessarily related
- When thhe super class is too general that you cannont instantiate it (create objects), the class is made abstact

Class Shape - abstact class

- creating a shape object does not make any sense (no real attributes)
- Class **Shape** - abstract methods getArea() and getPerimeter() but with no definiton
- But every subclass must have a getArea and getPermiter method

Constructors of the abstract class are set to protected since this would allow only the subclasses to create instances of the class

- Abstract classes cnanot be instntiated - but can be ised as a data type (polymorphism)
- Abstract methods make the class abstract bit the class can be abstract without abstract methods
- Abstract methods must be implemented in the sub class. If they are not, the subclass remains abstract
- A sub class can be abstract even if the super class is not(object and shape)

Interfaces

- **Interface** - class like constructor for defining common operations (behavior) for objects from unrelated classes
- Similar to abstract classes but contain behavior of objects of unrelated classes
- Examples: Comparable,Edible,Cloneable,Drawable,etc...
- Defined using the keyword interface instead of class

- Contrains only static constant, static methods, and abstract methods,
- A class **implements** an interface **not extends**

Deafult Definitions

- Abstract methods in an interface may have a deafult definition
- When an interface is implemented the deafult definition may be used or overridden

Interface- Comparable

- java.lang.Comparable: Interface to define the comparable feature between objects of any class
- The interface has only 1 abstract method compareTo()

```
public interface Comparable<E> {
 int compareTo(E obj);
}
```

`int compareTo()`

- returns 0 if the tow arguments are equal
- returns > 0 if the first argument comes after the second
- returns < 0 if the first argment comes before the second argument

2/10/2022 Interfaces and Abstract Classes continued

Interfaces-Cloneable

- Empty Interface to define the ability to be cloned for objects of any class (**marker interface**)
- The interface is empty and is only used to mark a class as having cloneable behavior

```
public interface Cloneable {
}
```

- Implementing the interface CLoneable consits in overriding the method `clone()` from class Object(Object clone())
- Many classes in Java API implement the interfaces Compareable and Cloneable

```
Object clone() {
    return this;
}
Object 01 = new Object();
Object 02 = 01.clone();
o2.a=15;
o1.a=35;
//Both would be equal to same place in memory
```

```

```

Deep copy/Shallow Copy

- **Shallow Copy** - copy of same object's location in memory

```
Object clone() {
    return this;
}
```

- **Deep Copy** - Different object with same attributes

```
Object clone() {
    return new Circle(this.getColor(),this.getRadius());
}
```

Implementing Multiple interfaces

Java allows for multiple interfaces therefore becoming an workaround for multiple inheritance.

2/15/2022 Recursion

- Recursion is a trchnique to solve iterative problems that are dificult to solve using loops
- Painting a surface
- Finding a file in a file hierarchy
- Drawing or traversnig a tree structure

Painting a surface would be divining the larger sections of the wall into a small section. Solving many small problems makes big problems eaiser.

File searching would mean looking through each path and going back through subsequent folders

Drawing or traversing a tree structure -File searching is also considered a tree-

Recursive method is a method that calls itself

Example: calculating factorial

```
n! = n * (n-1)!
5! = 5*4!
4! = 4*3!
3! = 3*2!
```

```
2! = 2*1!
1! = 1
```

```
int factorial(int n) {
    if(n==1||n==0)
        return 1;
    else
        return n*factorial(n-1);
}
```

In the recursive method the base case must exist. This is where the program will stop and return which would provide values to all other calls in the recursion.

Infinite recursion is when there is no base case, which without would cause a stack overflow.

Stack overflow - stack is full.

```
int power(int x, int n) {
    if(n==0)//base case
        x=1;
    else
        return power(x,n-1)*x;
}
```

Recursive binary search

Using divide and conquer, finding a key in an array of ordered numbers.

Continuously splits array upon sorted list to know if search value will exist.

iterative binary search

```
int binarySearch(int[] list, int key){
 int first, last, middle;
 first = 0;
 last = list.length-1;
 while (first <= last){
    middle = (last + first) / 2;
    if (key == list[middle]) return middle;
    else if (key < list[middle])
        last = middle - 1;
    else
        first = middle + 1;
 }
 return -1;
```

Recursive binary search

```
int binarySearch(int[] list, int key){
    int first = 0;
    int last = list.length-1;
    return binarySearch(list, first, last, key);
}


int binarySearch(int[] list,int first,int last,int key){
    if (first > last) return -1;
    else{
        int middle = (last + first) / 2;
        if (key == list[middle]) return middle;
        else if (key < list[middle])
            last = middle - 1;
        else
            first = middle + 1;
 return binarySearch(list, first, last, key);
 }
}
```

When extra parameters are needed in recursive call use helper method to assist

Recursino vs. iteration

- REcursion usually requires less code
- Recursion reflects the divide-and-conquer strategy for solving a problem
- Recursion requires consecutive calls to the same function (context switching - stack push/pop operations)
- Iterations are perferred by compilers
- Iterations may be more efficieint (Computationally)

Recursion is not always good

An example of this would be the fibonacci series

Used to model many real-life situations such as the rabbit population growth

Fibonacci series is F(n)=F(n-1)+F(n-2)

```
1 1 2 3 5 8 13
```

iterative fib

```
int fibonacci(int n){
    int f1=1, f2=1, f=0;
    if (n <= 2)
```

```
            return 1;
        else{
            while (n > 0){
                f = f1 + f2;
                f1 = f2;
                f2 = f;
                n = n -1;
            }
         }
     return f;
    }
```

recursive fib

```
int fibonacci(int n){
    if (n <= 2)
        return 1;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

This is a nice looking code but is very redundant even when just considering 5.

Although less lines of code may be less efficient.

Use recursion only when the problem is hard to solve using loops.

```
/users
    /Documents
        /Spring2022
            /eclpise-projects
                /HW2
                    /src
                        test.java
                    /bin
    /Desktop
    /Downloads
```

Using class file to access properties of files

Allowing us to pass a path of the file into the constructor

`File folder = new File(path) File[] <- folder.listFiles()`

```
public static String searchFile(String startPath,String fileName) {
        File folder = new File(startPath);
```

```
        if(!folder.isDirectory()) {
            return "";
        }
        else {
            File[] files = folder.listFiles();//return content of folder
            for(int i=0;i<files.length;i++) {
                if(files[i].isFile()) {
                    if(files[i].getName().equals(fileName)) {
                        return files[i].getAbsolutePath();
                    }

                }
                if(files[i].isDirectory()) {
                    //look for filename in sub folder of files
                    String out = searchFile(files[i].getAbsolutePath(),
fileName);
                    if(out.equals(""))
                        return out;
                }
            }

        }
        return "";
    }
}
```

2/17/2022 Recursion continued ALA 4 start

2/22/22 Generic types

What are generics?

- Genereics allow to specify a range of types allowable for a class, an interface, or a method.
- Used to create classes that hold data of different types.
- Used to create mehtods that accept parameters of different types
- interface `Compareable<E>` can be implemented for any reference type E

```
interface Comparable<E> {
    int compareTo(E obj);
}
```

- Generic Class - Class of type `<E>`
- E is the type parameter or generic type

- E can be replaced by any reference type String, Integer, or Student
- Primitive types are **not** allowed as generic type parameters(int,double,char,...)
- Can use any name for the generic type (between <>)

Generic Class - `java.util.ArrayList`

- array of objects of any type
- Array of any size
- The sie of the array may increase or decrease at runtime
- Like a wrapper class for Arrays
- A generic type can be defined for a class or an interface
- A concrete type must be specified when using the generic class/interface
- Either to create objects or use the class as a referance type.

Erasure of the Generic Type

- After complile time, E is removed and replaced with the raw type (Object)
- Old way of implementing generics: use type Object intstead of E
- Using array with elements of type Object would also work
- Using Generics improve software reliability and readablility
- Errors are detected at complile time

Restrictions on Generic types

- Restriction 1
  - cannot create instances of generic types
- Restriction 2
  - cannot create array of type generic
- Restriction 3
  - not allowed in static context
- Restriction 4
  - cannot create exception class

Multilple Generic Types

- A class/interface may have multiple type parameters (generic types)
- Class Pair<E1,E2>

2/24/22 Generics (Templates)

Generic Methods

- A method can be generic - parameters or return value are of type generic
- Printing arrays of different types `printArray()`
- Searching arrays of different types
- Sorting arrays of different types `java.util.Arrays.sort()`

Generic method to print arrays of any type E

```
public static <E> void printArray(E[] list) {
    System.out.print("[ ");
    for (int i=0; i<list.length; i++)
        System.out.print(list[i] + " ");
        System.out.println("]");
}
```

- Sorting arrays of different types `java.util.Arrrays.sort()`
- sort() needs to compare the elements (order them)
- Elements of the array must be compareable

```
public static <E extends Compareable<E>> void sort(E[] list) {

}
```

When using generic types `extends` will be used in every case

There are two different versions of the sort method

- `java.util.Array.sort()` is a generic method and is overlaoded(Compareable or Comparator)

Compareable

```
public static <E> void sort(E[] list)
public static <E> void sort(E[] list,int start, int end )
```

`Comparator<T>`

```
public static <E> void sort(E[] list, Comparator<?, Super E> c)
```

```
java.util.Comparator

public interface Comparator<T> {
    int compare(T obj1,T obj2);
}
```

- A generic class or interface used without specifting a concrete type, is raw type and will be replaces with Object `Stack stack = new Stack();` same thing `Stack<Object> = new Stack<>`
- Raw types are used for backward compartibility only
- Old Java Cersion of the interface Compareable is not generic

`int compareTo(Object obj)`

WilCard Generic Type

- Generic types can be restricted to specific types of groups of types
- `<E extends Compareable<E>>` restircts the type E
- Types of wildcards: ?, ? extends T, and ? Super T
- Unbounded wildcard ?
    - Equivalent to ? extends Object
- Bounded wildcard ? extends T
    - Generic Type must be T or a subtype of T
- Lower bound wildcard ? Super **T**
    - Generic type must be T or super type of T

3/1/22 Algorithim Analysis

No Coding just theory

- Algorithim Design- Finding a computational solution to a problem
- Often many solutions are possible. How to select a solution?
    - Use Algorithim Analysis
    - Example: Binary Seacrh and Linear Search
- Algorithim Analysis: Predict the performance of an algorithm before implementing it (coding)
- Determine the upper bound on the performance of the algorithm based on the problem size
- **Growth Rate** - how fast an algorithms execturion time (or memory space) increases as the input size incresases

- Worst Case analysis
- Theoretical approach independent of computer (Machines) and specific input
- **Big-O** notion - is a mathematical function for measuring algorithm time complexity (or space complexity) based on the input size
- Time complexity- Execution time as a function of the input size
- Space Complexity- Amount of memory space as a function of the input size

Big-O Notation

- Linear Search `Time(n) = (3n + 2).const=O(n)`
    - Time grows linearly with the input size (n)
    - O(n) - Order of n - Linear Algorithm
- Mutiplicative constants and non-dominating terms are ignored

Useful formulas

1+2+3...+n=n(n+1)/2 - O(n^2)

a^0+a^1+a^2...a^n = O(a^n)

2^0 + 2^1+ ... + 2^n = 2^(n+1)-1 ≅ O(2n)

Determining Big-O

```
for(int i=0;i<n;i++)
{
    k=k+5
}
```

Time complexity(3*n+1)*const = O(n)

```
for(int i=1; i<= n; i++){
    for(int j=1; j<= n; j++){
        k = k + i + j;
 }
}
```

Time Complexity: (1 + 3*n* + *3*n^2) * const = O(n^2)

```
for(int i = 1; i <= n; i++){
    for(int j = 1; j <= i; j++){
        k = k + i + j;
 }
}
```

Time Complexity: [1 + 3*n* + *3*(1+2+...+n)] * const = n + (n+1)n/2 = O(n^2) - Quadratic

---

```
for(int i = 1; i <= n; i++){
    k = k + 4;
}
for(int i = 1; i <=n; i++){
    for(int j=1; j<=20; j++){
     k = k + i + j;
    }
}
```

Time Complexity: = (1+ 3*n) * const + (1 +* 3n + 3 * 20 * n) * const = O(n)

```
if(list.contains(e)){
   System.out.print(e);
}
else{
   for(Object t: list){
   System.out.print(t);
   }
}
```

Time Complexity: (1+ n)*const or ((2* n)+ n)*const = O(n)

```
result=1;
for(int i=1; i<=n; i++)
 result = result * a;
```

Time complexity: (2 + 3 * n) * const = O(n)

```
result = a;
for(int i=1; i<=k; i++)
 result = result * result; c
```

Time complexity: (2 + 3 * log n) * const = O(log n)

```
int count = 1;
while(count < n)
 count = count * 2;
```

Time complexity: (1 + 2 * n/2) * const = O(n)

Analysis of Binary Search

After each iteration the array size is split into half

Eventually until n/(2^k)=1 since can no longet be split

then from there can be considered log n =k

Analysis of Selction sort

since there are nested loops depending on the size of each is considered O(n^2)

Analysis Recursive Fibonacci sequence

Time(n) = Time(n-1) + Time(n-2) Time(n) ~ 2 * Time(n-1) Time(n-1) = 2 * Time(n-2) Time(n) = 2 * 2 * Time(n-2) ... Time(n) = 2^k * Time(n-k) Time(n) = 2^(n-2) Time(2) Time(n) = 2^(n) * constant

Recursive Fibonacci: O(2n) - Exponential growth

Iterative

```
public static long fibonacci(long n) {
 long f0=0,f1=1, f2=1;
if(n == 1 || n == 2)
 return f1;
for(int i=3; i<=n; i++){
 f0=1;
 f1=f2;
 f2=f0+f1;
 }
 return f2;
}
```

Time Complexity: (8 + 5 * (n-3)) * const Iterative Fibonacci: O(n) - Linear growth

---

O(1) < O(logn) < O(n) < O(n logn) < O(n^2) < O(n^3) < O(2^n)

```
for (int i = n/2; i > 0; --i){ //n/2
    int x = n * 3;
 while(x > 1){  //log(3n)
    for (int j=0; j < 100; j+=2) //50
        System.out.prinltn("X: " + x);
    x = x / 2;
 }
}
```

Therefore time complexity is log(3n)

3/8/2022 Data Structures

- Data Structure - Collection of data organized in a specific way
- Arrays are the most commonly used data structure
- Choosing efficient data structures and algorithims-key ussyes in developing high-performance software
- You can write any program with any data structure other than arrays
- the program efficienty can be incresed if yoou choose the appropriate data strucutres
- Data Structure is a generic class with
    - Data collection storage
    - Methods to manipulate the data

CRUD

- Create
- Remove
- Update
- Delete

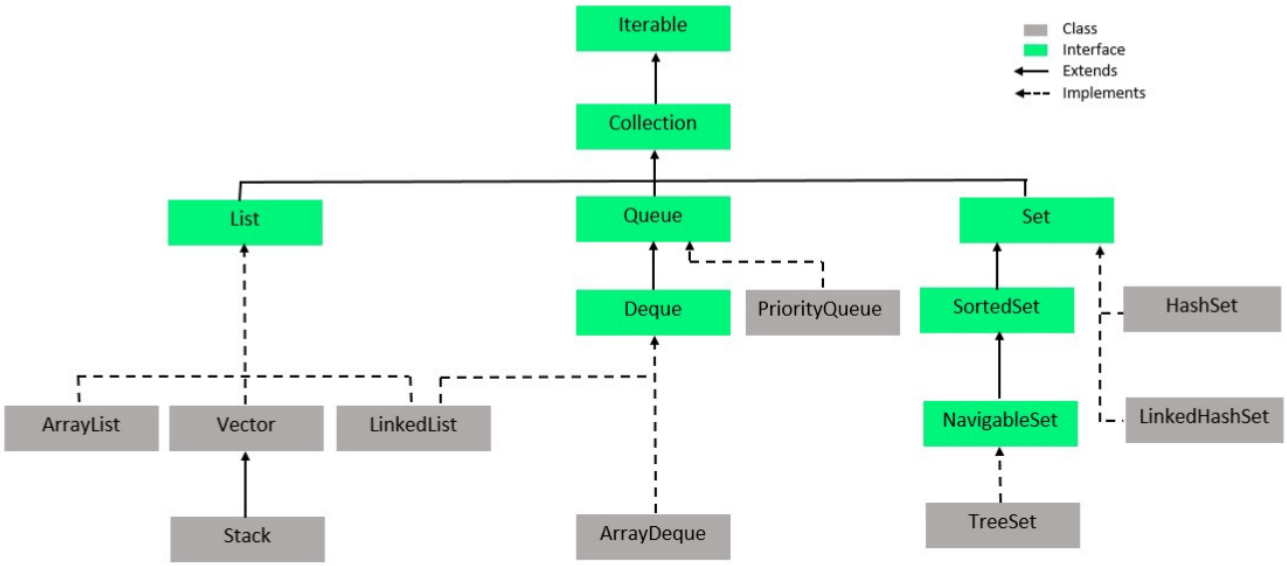
Java collection Framework

- Containers- Data Structures
- Iterators- objects to iterate through the containers's data items
- Algorithims- Utillity mehtods to manipulate containers

Containers

- List
    - ArrayList
    - LinkedList
- Stack
    - Stack
- Queue
    - Linked List
- Priority Queue
    - PritorityQueue

- Binary Tree
    - Hashset
- Hash Table
    - Hashmap



Since ArrayList stems from Collection

```
Collection<String> c1 = new ArrayList<String>();

//but in the future
Collection<String> c3 = (ArrayList<String>)(ArrayList<String>c1).clone();
//since clone does not exist in collection
//but clone will by deafult will return type obj so need to down cast back
to array list
```

ArrayList add() -union remove() -intersection remove all() -difference

Iterator

Iterator stems from interface-interface. Iterator only contains one method `iterator()`, allowing objects to be called with methods that exist in iterator type

List Iterator

Different type that allows for bidirection methods

Java Collection Framework

Contains many base methods that can be called staticlly on data structures.

Link to JavaDoc

Containers

- List: store ordered collection of elements
- Stack: stores elements that are processed in LIFO fashion (Last-In First-Out)
- Queue: stores elements that are processed in FIFO fashion (First-In First-Out)
- PriorityQueue: stores elements that are processed in the order defined by a priority

List

Array Based List ArrayList-Random Access to the elemends index to any element

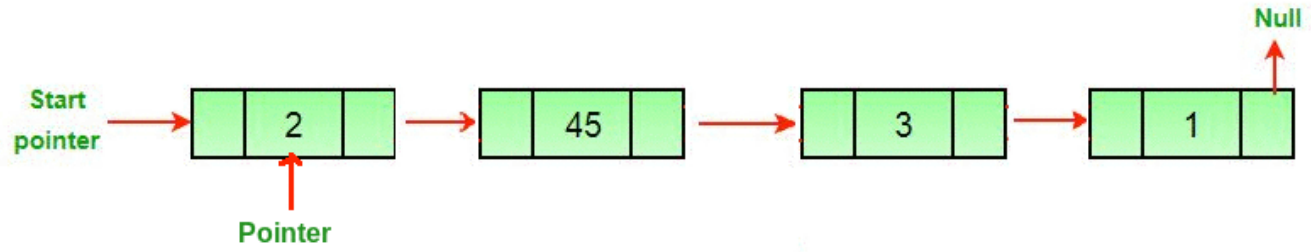
`add(obj)`-is added to end of the list

`add(int index,obj)`- is added at specific index

`remove(obj)`- removes val from list

`remove(int index)`- removes val at index

Linked List LinkedList-Sequential access only (first,last,next)



`addFirst(obj)`- head now becomes obj with referance to old head of list

`addLast(obj)`-tail now becomes obj with referance to old tail of list

`removeFirst()`-Head now becomes referance to second item

`removeLast()` -tail now becomes refeance to second-to-last item

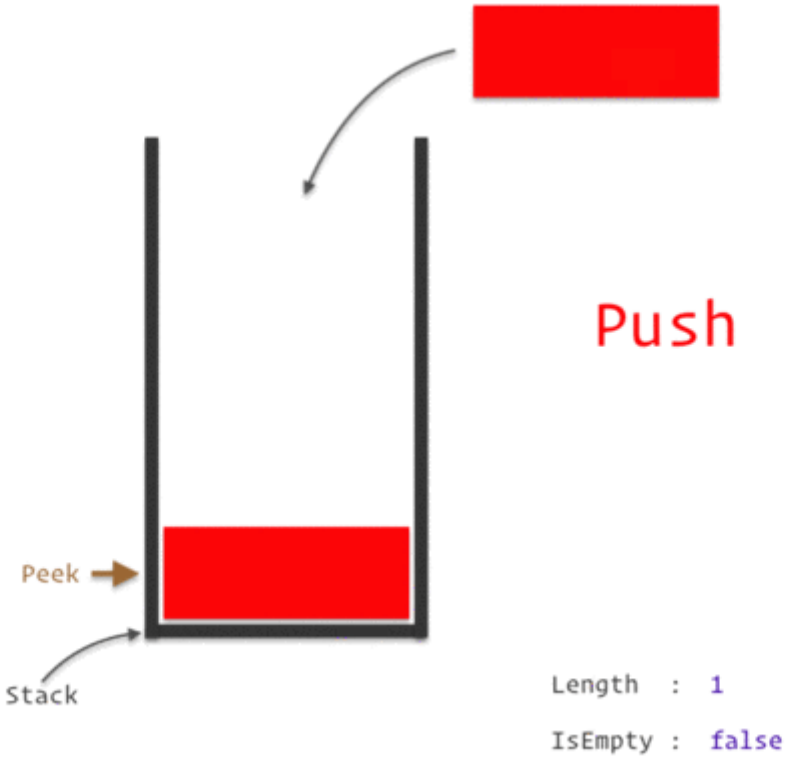
Summary

- ArrayList
  - Random access to any element
  - uses an array (contiguous memory space)
  - Size of the array can be adjusted at runtime
- LinkedList
  - Sequential access to the list elements
  - Uses as much memory as the number of elements in the list (more efficient in memory usage)

Stack

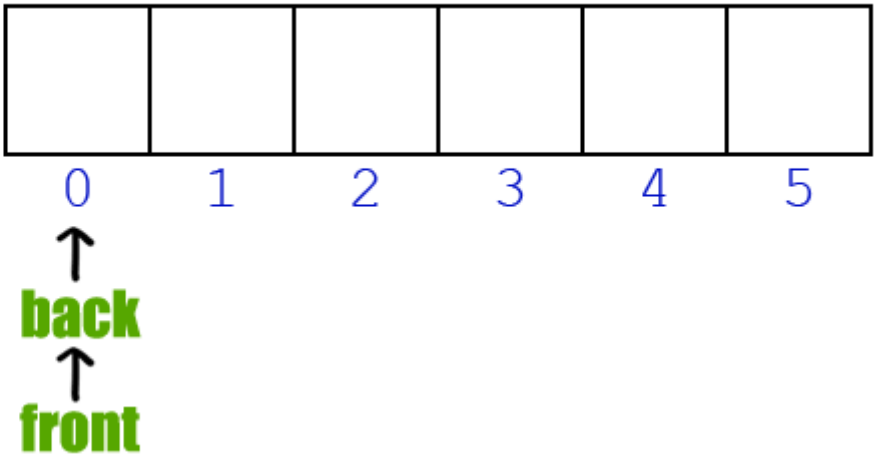
- LIFO structure - (last in first out)
- Acess to the top of the stack only

- Operations; `push()`,`pop()`,and `peek()`
- Used for tracking method calls and arithmrtic expression evaluation



Queue

- FIFO structure-(first in first out)
- Access at the front (or back) only
- Operations: `offer()`,`poll()`,and `peek()`
  - offer in back poll in front
- Used for job scheduling and many real-life problme modeling
- Implemened as a linked List in the Java API



Data Structures continued 3/10/2022

Priority Queue

- FIFO structure with priority
- Access at the front or back only
- Elements are inserted according to their priority
- Operations: offer(), poll(), and peek()
- Used for job scheduling and many real-life problem modeling too
- Priority queue uses the natural ordering (compareTo() from comparable) or a comparator (compare())

Will fill each added object in the sort of the given comparator, if none is given will be given normal compare of passed type.

Application of Data Strucutes

Infix and post fix is the method of using a stack to hold operations in pemdas

Infix: (1 + 5) * (8 - (4-1))

postfix: 1 5 + 8 4 1 - - *

With given postfix expression two numbers are taken and the next operation is taken

Ex 12 25 5 1 / / * 8 7 + -

Stack: 12 25 5 1 | pop(1) pop(5) push(5/1) Stack: 12 25 5 | pop(5) pop(25) push(25/1) Stack: 12 5 | pop(5) pop(12) push(12*5) Stack: 60 Stack: 60 8 7 | pop(7) pop(8) push(8+7) Stack: 60 15 | pop(15) pop(60) push(60-15) Stack: 45 | pop(45)

3/22/2222 Data Structures: Implementation

Outline

- Implementation of List
- Stack
- Queue
- Priority Queue

List

- Store data in order
- Commmon Operations on a List
    - Reteieve
    - add
    - remove
    - get number of elements

ArrayList `ArrayList<E>`

- Fixed array size when the list is constructed
- New larger array is created when the array is full

LinkedList `LinkedList<E>`

- Size not fixed
- Nodes are created when an Item is added
- Nodes are linked together to form a list

Array based List

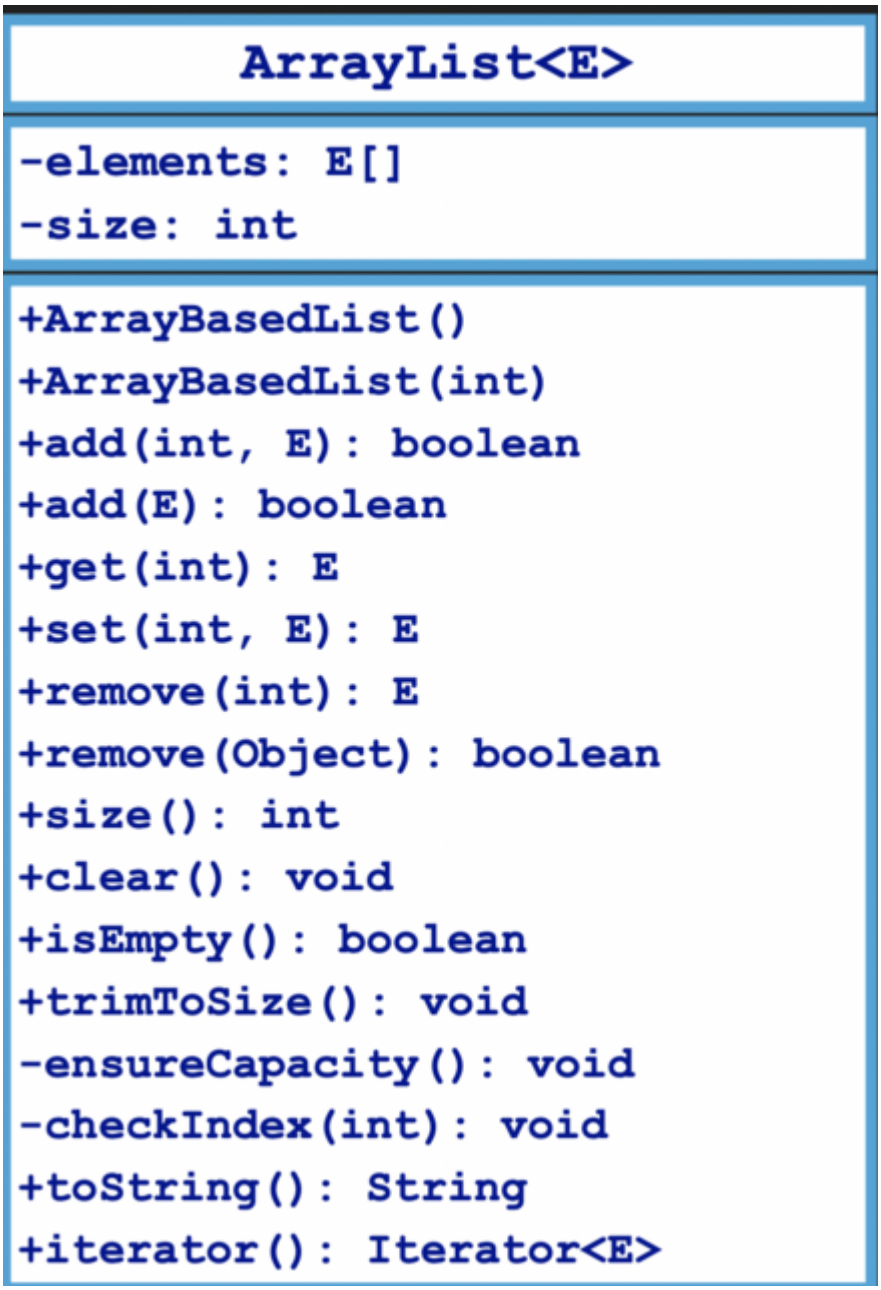
Inserting an element at a specific index

- will check if `size==capacity`, if larger then will create new array list of size 1.5x original size
- When adding at item to index will shift all later elements over.

I am learning data structures to understand what is going on behind the scenes and also be capapable of creating my own data structure when needed

- When I remove an element at a specific index
    - Shift all elements after the inex and decrease the size by 1

Now we are going to implement an Array Based List class



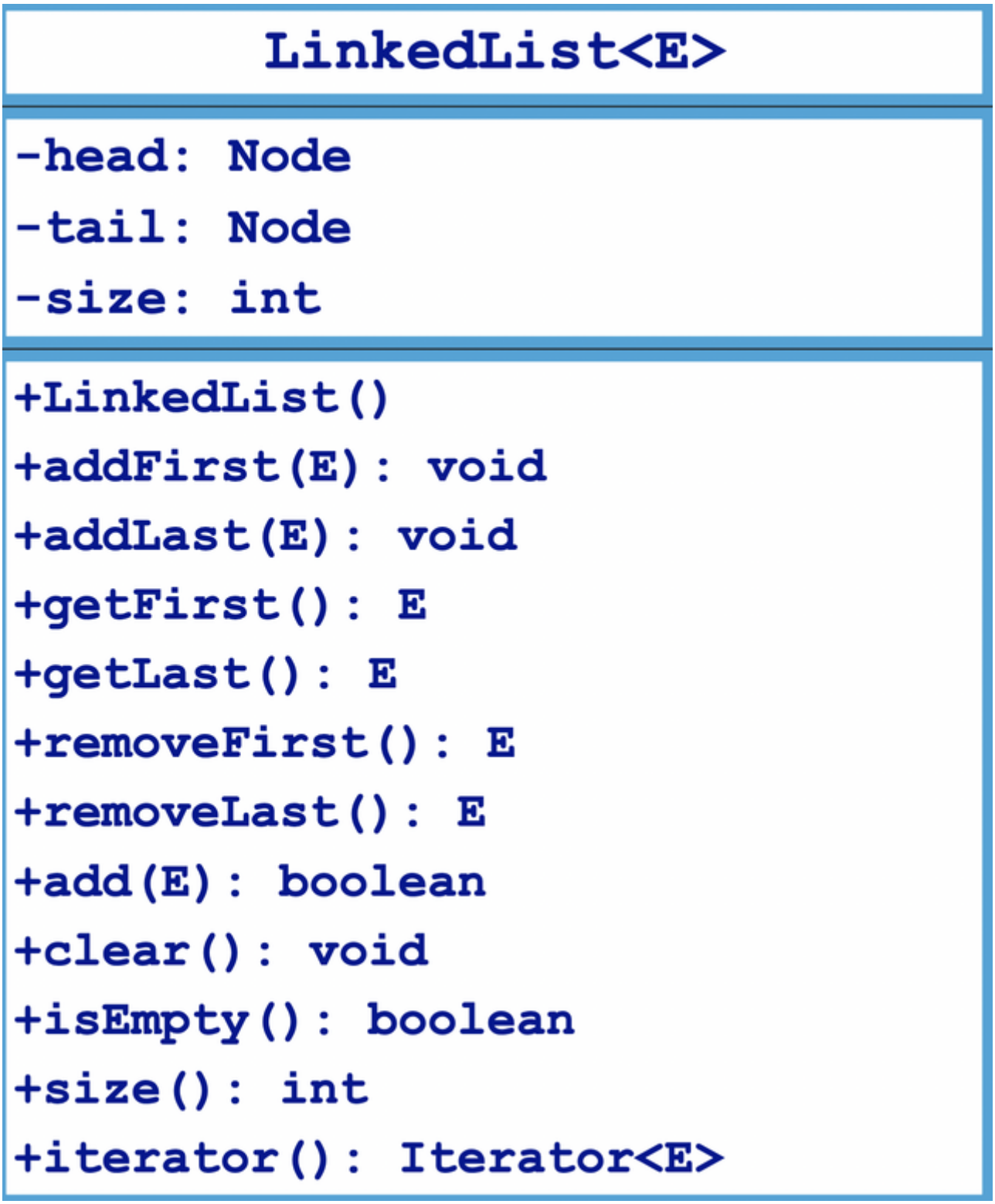
3/23/2022 Data Structure implementation continued

Linked List

- List implementation using linked nodes
    - Class node (inner class - inside LinkedList)

```
Class Node {
    E value;
    Node next;
    Node(E) {

    }
}
```



When initiating a linked list,

1. New node is created setting next to null
2. Tail and head are set to `&` of 1

How `addLast()` works

- Creates new node
- Sets link of node1 next to node2
- Set tail to `&` of head2

Traversal

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

`addFirst()`

- Create new node
- Set next of new node to head
- Set new node to head

`removeFirst()`

- Set head equal to next of current head
- remove head

Two variations of Linked List

Doubly Linked List

- Every node is linked to the previous and next node

Circularly linked list

- Last element is linked back to the first node

Doubly Linked List

This list improves time complexity of removelast() from O(n) to O(1)

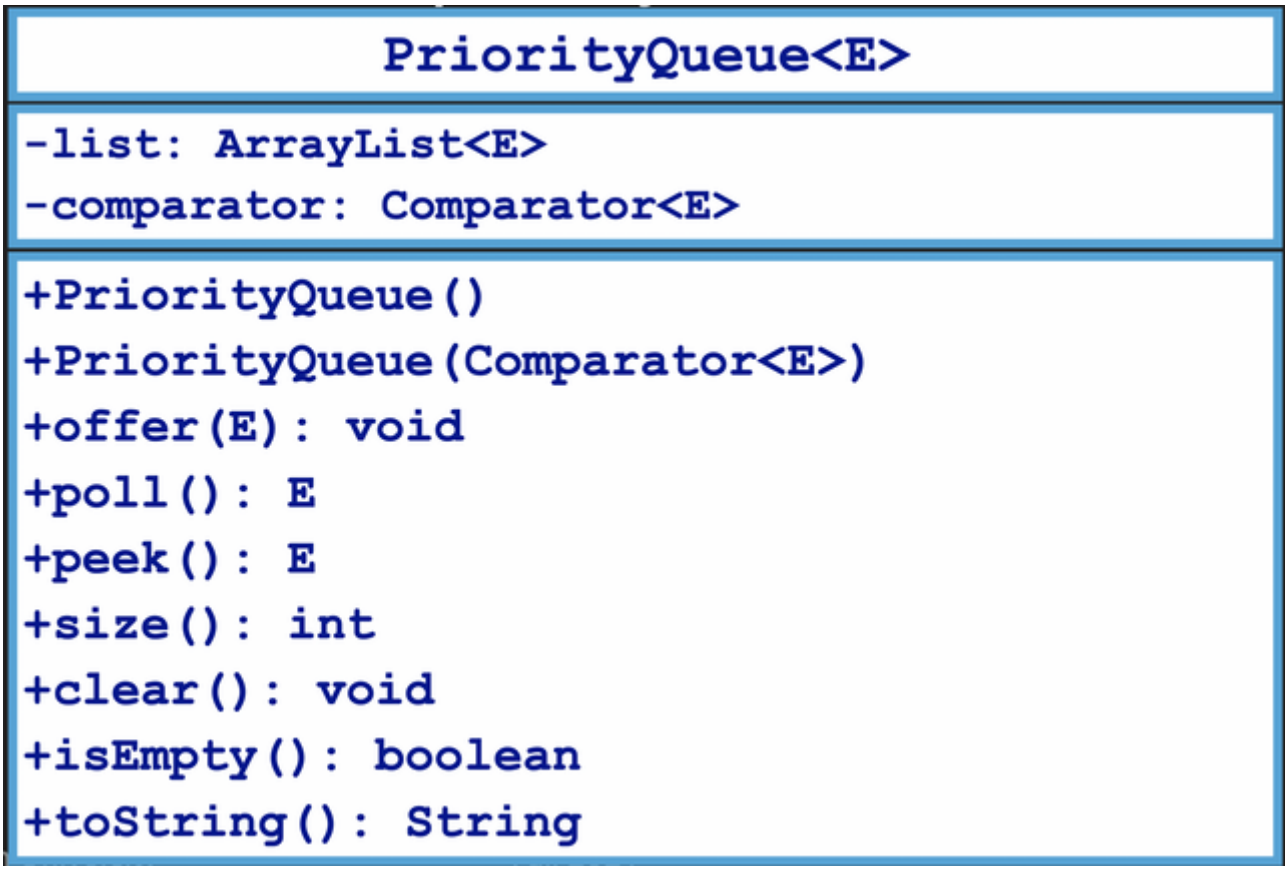
Stack and Queue

- **Stack** is implemented using an array basked list with access only at the end of the list
- **Queue** is implemented using a linked list with access at the head and the tail.

3/29/2022 Data Structure implementation continued

Time complexity of Queue operations are constant except toString().

Priority Queue



Review `Compareable<E>` `compareTo(E object)`

`Comparator<E>` `int compare(E o1,E o2)`

Comparator is used to compare object in some type of way other than the current ordering of the object

3/31/2022 Binary Trees (very tired today)

Properties

What is a binary tree

- Data is organized in a binary tree strucutre
- easy and efficient access and update in large collections of data
- Used for efficient search operations
- Wide range of applications: mathematical expressions, game strategies, decision tree, data compression

Properties

- set of elemtnes called nodes(verticies) interconnected with edges (arcs)
- The first node is called the root
- The root is connected to two binary trees (left and right)
- Every node has a parent (expect the root) may have no child, one child, or at most two chldren
- the root is the ancestor of all nodes

- **Path**- Sequence of connected nodes starting at any level of the tree
- **Length of a path** - number of nodes in the sequence -1 or numberof edges
- If there is a path from node P to node Q, Q is the descendant of P and P is an ancestor of Q
- **Depth of a node**- length of a path from root to the node
- **Height of a tree**- the depth of the deepest node +1
- **Leaf Node**- node as no children
- **Internal Node**-node that has at least one child
- **Full Binary Tree** - each node is a leaf or an internal node with exactly two children
- **Complete Binary Tree**- Every level is filled except the last level and the leaves on the last level are placed left most
- **Balanced Binary Tree** - for each node the height of the left subtree and the height of the right subtree differ by at must by 1

Traversal

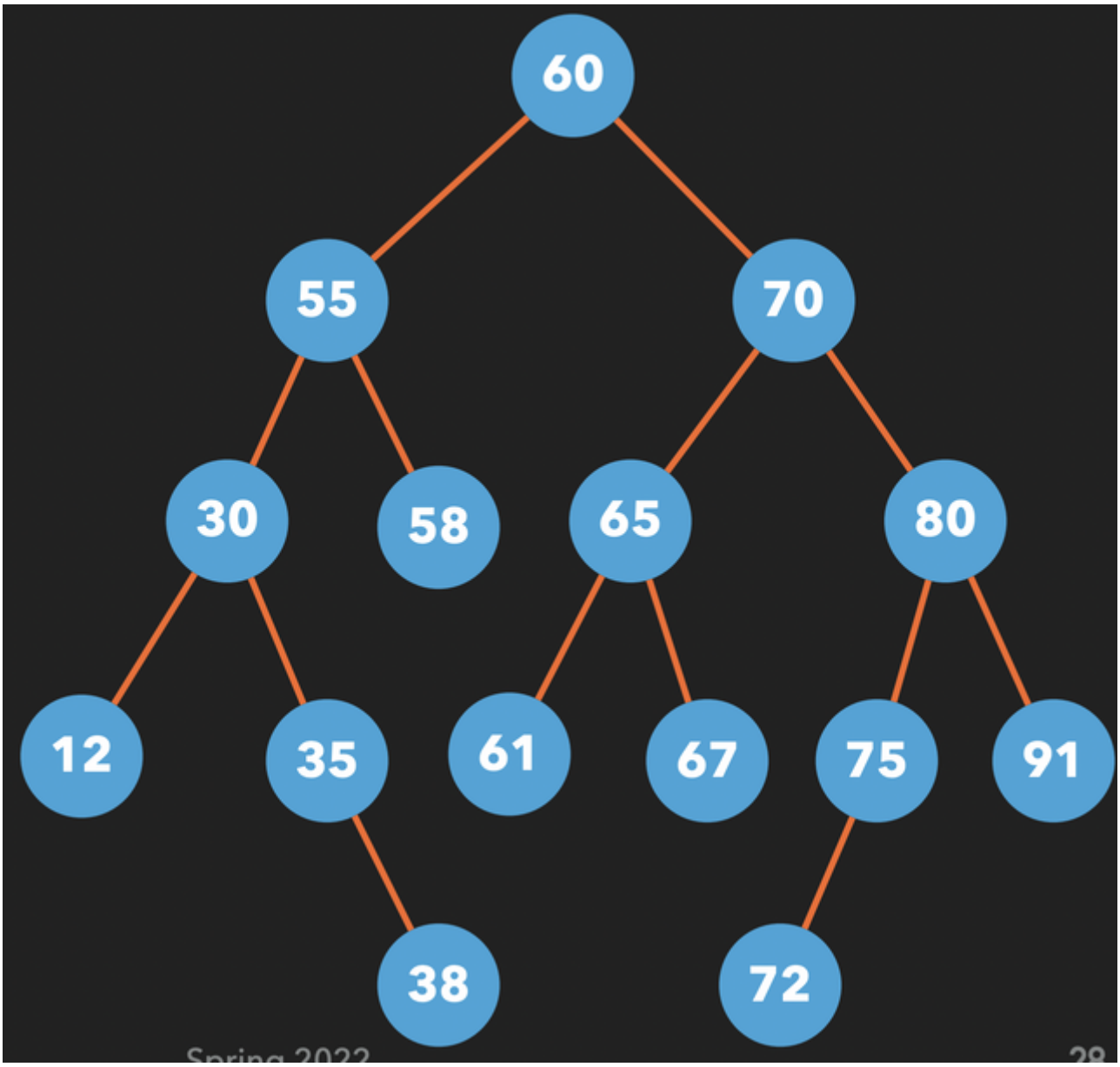
- Any process of visiting all of the nodes in a tree is called traversal
- Three common traversals
    - Preorder
    - Inorder
    - Postorder **Preorder Traversal**
- Any node is visitied before its chidlren
- V-L-R = Visit Node -Go left-Go right
- Visit the node
- Traverse to the left subtree
- traverse to the right subtree

inorder traversal

- Any node is visited after its left subtree and before its right subtree
- L-V-R = Go left-visit node-go right
- Traverse to left subtree
- Visit the node
- Traverse to the right subtree

Post order traversal

- Any node is visited after its left subtree and right subtree
- L-R-V = Go left-Go Right-Visit node
- Traverse left subtree
- traverse right subtree
- Visit node



Preorder: 60 55 30 12 35 38 58 70 65 61 67 80 75 72 91

Inorder: 12 30 35 38 55 58 60 62 65 67 70 72 75 80 91

post order: 12 38 35 30 58 55 61 67 65 72 75 91 80 70 60

Binary Search Tree (BST)

- **Special Binary tree**
  - BST has a root, a left subtree (L) and a right subtree (R)
  - The value of the root is greater than the value of every node in L
  - The value of the root is less that the value of every node in R
  - L and R are also BSTs
  - Used for efficient search in large data sets

Common operations on BST

- **Search** for a specific value in the BST
- **Add** a node to the BST while keeping the same properties
- **Remove** node from the BST while keeping best prooperties

- **Traverse** the BST(preorder,inorder,postorder)

search in prior example 35<60 (L)-> 35<55 (L)->35>30 (R) -> 35

Add uses similar algorithm to search just adding when reached null

Remove

- Node is leaf
  - simply set to null
- Node has one leaf
  - leaf becomes new node
- In case of (70)
  - Would find largest node in left subtree and replace with current node

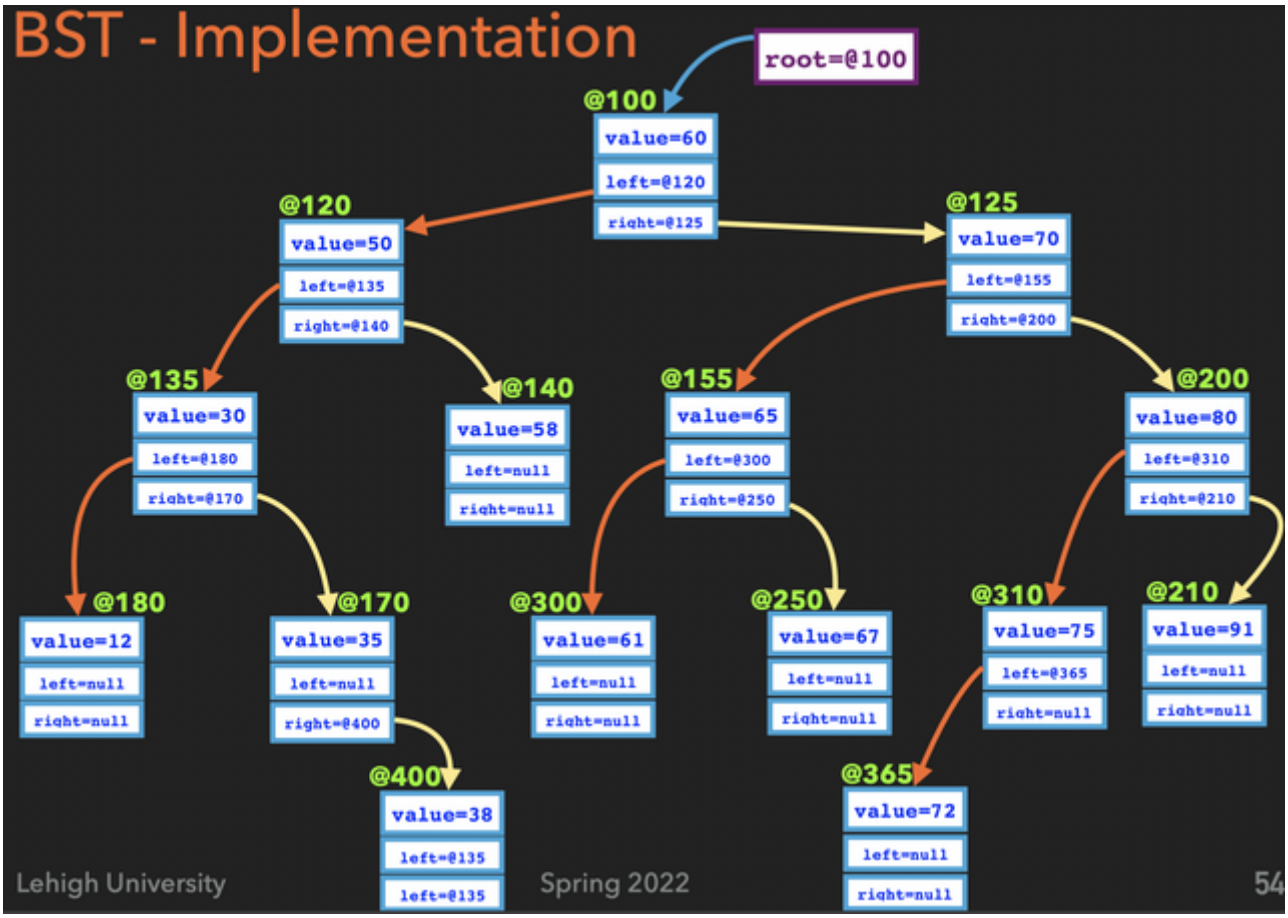
4/5/2022 Binary trees Exam thursday 4:25

BST Implementation

- Array Based BST
  - Not the best since will waste the most amount of space
- Linked BST

Array Based Nodes of the tree are stored in the array Children of the node follow the node

Linked Based Nodes of the tree are linked Every node has a value and two referances to the left and right child.



4/12/2022 Heaps

- Heap is a special binary tree
  - Complete binary tree
    - Meaning all levels excpet last are filled
  - Every node is greater than or equal to any of it children (Max Heap) [Min Heap: less than or equal]
  - Used for effiecient sorting

Heap operations

2 main operations **Adding a new node while maintaining properties Removing a node while keeping heap properties**

Heap implementations

IndexOf(Parent) = (IndexOf(current) - 1) / 2

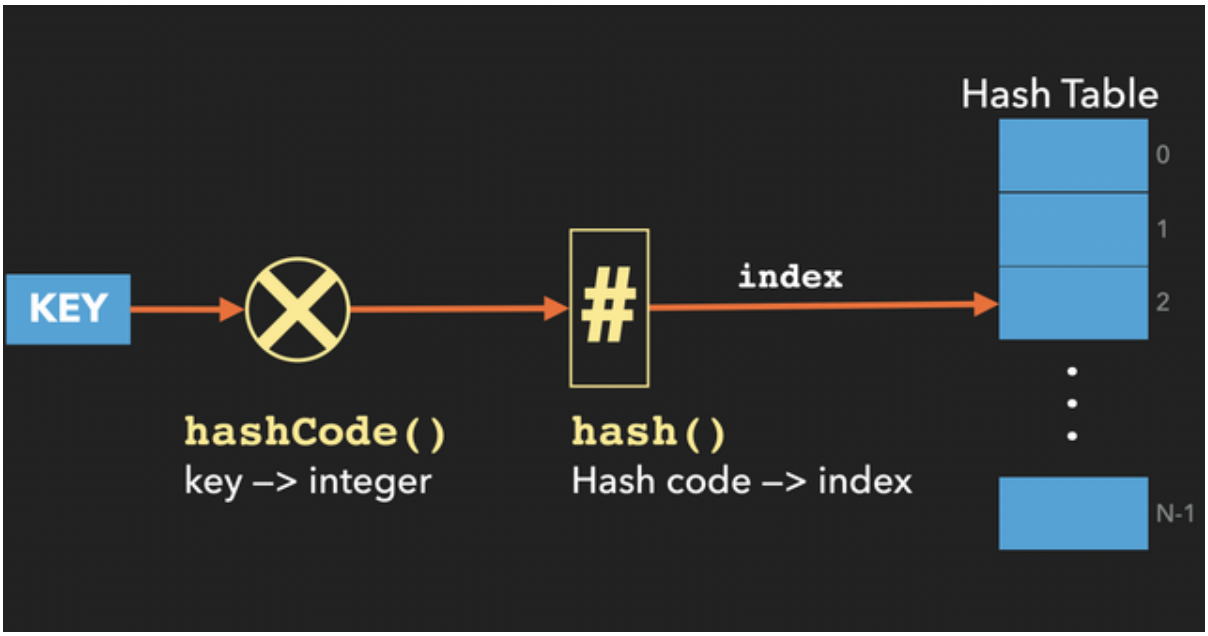
IndexOf(Left child) = 2 * IndexOf(current) + 1 IndexOf(Right child) = 2 * IndexOf(current) + 2

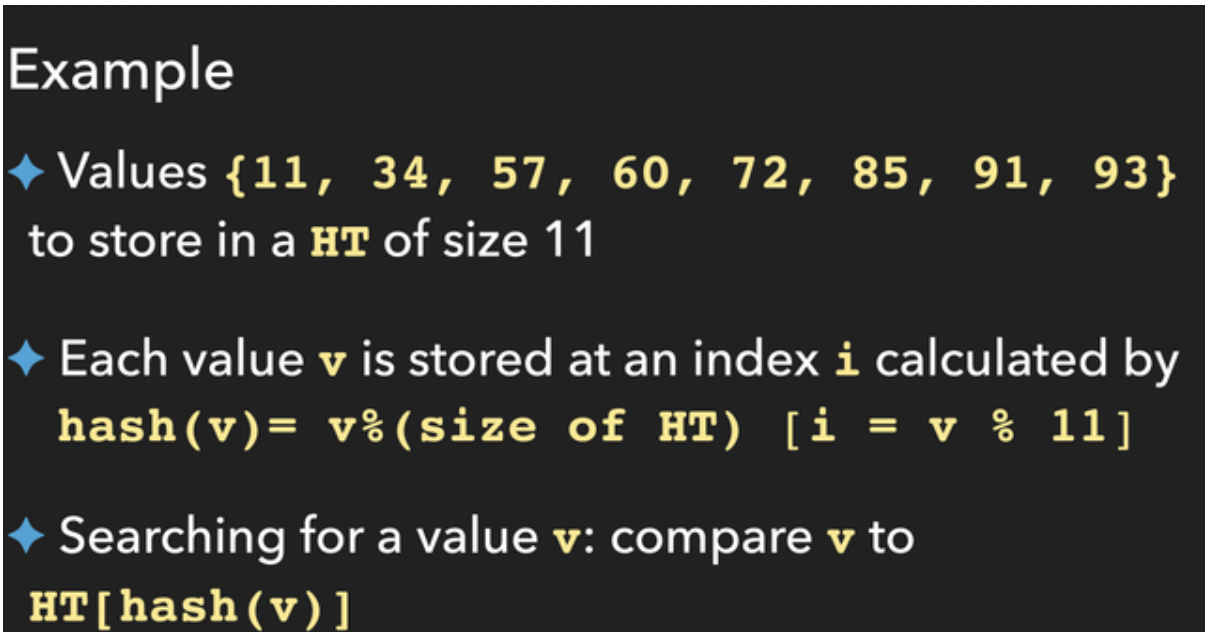
4/14/2022 Hash Tables

Hashtables preform searches in O(1)

- Hash tables use associatice access to data

- Associative memory: access data using the data itself instead of an address (index)
- Store the data in an array - Hash Table (HT)
- Access the element in HT using a hash function h() that returns an index in HT
- If the size of HT is N then `0<= hash() <= N-1`
- Searching for a value key is performed using one comparison with
    - HT[hash(hashcode(key))]
- adding data to the table
    - Apply hash to the index to find where the data should be added
- Retriving data from the table
    - apply hash to the datato find the index where it is in the table

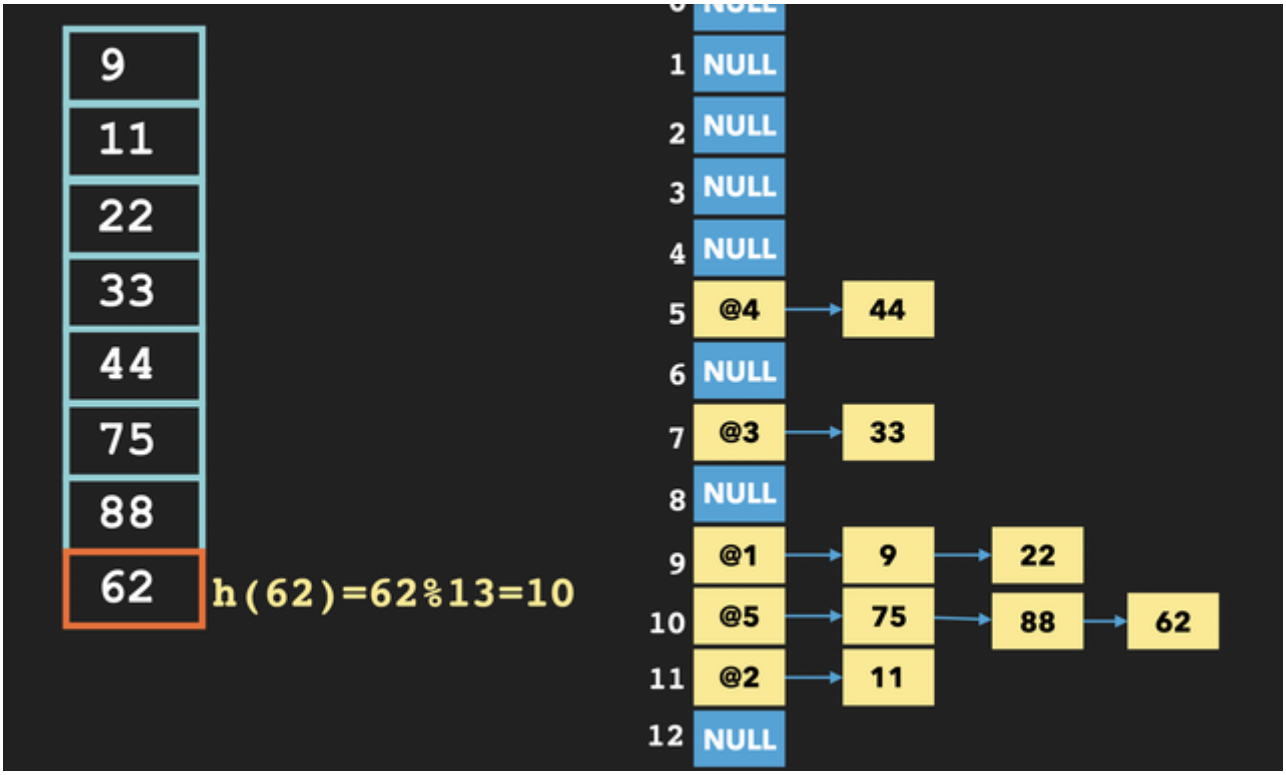




Collisions exist when two numbers are added with the same index

- Two ways to solve collisions
- Sperate chaining- collisions are stored outside the hash table in a list
- Open addressing collisions are stored in the hash table itslef at the next avalible index

Seperate chaining



Open Addressing

- Linear probing
  - Looking for the next avalible slot using leinear function till empty slowt is found
- Quadratic probing is done by squareing the index to find the next avalible index
- Double probing- is double by using another hashing function

The number of collusions may affect the performance of the serach operations could result in linear search if the number of collisions is high

Search, add, remove - O(1)

- Three factors may cause more collisions
- affect the constant time performance
  - HashCode function
  - Hash function
  - Size of the hash table

HashCode function

- hashCode() is simple for integers - return the integer itself
- hashcode() exists already in simple objects

Everytime equal() is overridded, hashCode() should be as well

Size of the Hash Table

- Choosing the size of the table
- A prime number larger than the size of the data set - may take time to find such number
- Bigger table - less collisions - waste of memory space
- Tradeoff - space vs. time - use power of 2 to simplify calculations

Load Factor - How full is the hash table?

- Load Factor = # of added elements / size of the HT
- High load factor results in more collisions - requires rehashing
- Rehashing - Increase the size of the table and rehash all the data to add it to the new table
- 0.5 < load factor < 0.9 (0.5 for probing and 0.9 for chaining)

4/19/2022 Implementation of Hashmap

Load factor of 0.5 is going to be used for PP3

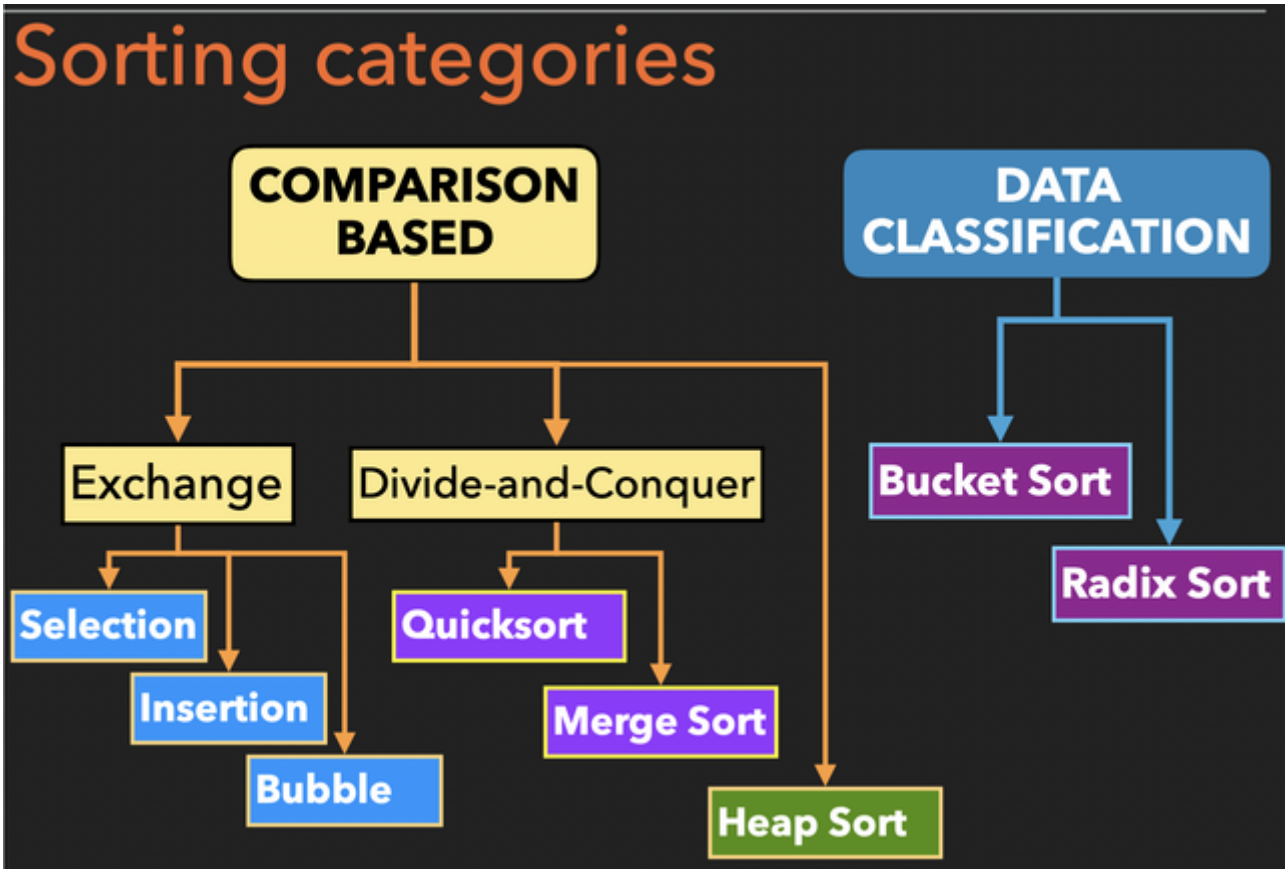
4/26/2022 Sorting algorithms

Sorting problem

- Given a list of data items arrange the list in an ascending or descenting order
- Commonly used in computer science to organize objects based on one specific criterion
- Allows using efficient binary search algorithm
- Sorting is more complex than searching

Sorting types Internal - Data is stored in memory

External - Data is in secondary memory (Hard disk:Large Files)



Selection Sort

Finds minimum in unselected portion of the list.

```
Algorithm selectionSort
 for every element i (N size of the list)
 find the smallest element from i to N-1
 swap element i with the smallest element
End
```

```
public static void selectionSort(int[] list) {
  int minIndex;
  for (int i=0; i<list.length-1; i++) {
    int min = list[i];
    minIndex = i;
    for (int j=i; j<list.length; j++){
      if (list[j] < min){
      min = list[j];
      minIndex = j;
      }
  }
  int temp = list[i];
  list[i] = list[minIndex];
  list[minIndex] = temp;
```

```
 }
}
```

Time Complexity

```
Iteration1(outer loop)
 (n)iterations(inner loop)
Iteration 2 (outer loop)
 (n-1) iterations (inner loop)
Iteration k (outer loop)
 (n-k+1) iterations (inner loop)
Iteration n-1 (outer loop)
 1 iteration (inner loop)
1 + 2 + … + (n) = n(n+1)/2
```

Time complexity: O(n^2) - Quadratic growth Space complexity: O(1) - no additional space (Space complexity is used to describe additional space within the memory)

Generic selection sort

```
public static <E extends Comparable<E>>
 void selectionSort(E[] list) {
    int minIndex;
    for (int i=0; i<list.length-1; i++) {
      E min = list[i];
      minIndex = i;
      for (int j=i; j<list.length; j++){
        if (list[j].compareTo(min) < 0){
          min = list[j];
          minIndex = j;
        }
      }
  E temp = list[i];
  list[i] = list[minIndex];
  list[minIndex] = temp;
  }
}
```

Insertion Sort

Iterating through list sorting individual item into sorted list.

```
Algorithm insertionSort
 for every element i
 insert element i in the sorted list
(0 to i)
```

```
 end for
End
```

```
public static void insertionSort(int[] list) {
 for (int i=1; i<list.length; i++) {
 //Insert element i in the sorted sub-list
 int currentVal = list[i];
 int j = i;
 while (j > 0 && currentVal < (list[j - 1]))
 {
 // Shift element (j-1) into element (j)
 list[j] = list[j - 1];
 j--;
 }
 // Insert currentVal at position j
 list[j] = currentVal;
 }
}
```

```
Iteration1(outer loop)
 (1)iteration(inner loop)
Iteration 2 (outer loop)
 (2) iterations (inner loop)
Iteration k (outer loop)
 (k) iterations (inner loop)
Iteration n-1 (outer loop)
 n-1 iterations (inner loop)
1 + 2 + … + (n-1) = n(n-1)/2
```

Time complexity: O(n^2) - Quadratic growth Space complexity: O(1)

Bubble Sort

```
Algorithm BubbleSort
 sorted = false
 last = N-1 (N size of the array)
 while (not sorted)
  sorted = true
  for i=0 to last-1
    if(list[i] > list[i+1])
      swap(list[i], list[i+1])
      sorted = false
    end if
  end for
  last = last - 1;;
 end while
End
```

```
public static void bubbleSort(int[] list) {
  boolean sorted = false;
  for (int k = 1; k < list.length && !sorted; k++) {
      sorted = true;
      for (int i = 0; i < list.length - k; i++) {
          if (list[i] > list[i + 1]) {
              // swap
              int temp = list[i];
              list[i] = list[i + 1];
              list[i + 1] = temp;
              sorted = false;
          }
      }
    }
}
```

```
Iteration1(outer loop)
 (n-1)iteration(inner loop) to push the max
Iteration 2 (outer loop)
 (n-2) iterations (inner loop)
Iteration k (outer loop)
 (n-k) iterations (inner loop)
Iteration n-1 (outer loop)
 1 iterations (inner loop)
1 + 2 + … + (n-1) = n(n-1)/2
```

Time complexity: O(n^2) - Quadratic growth Space complexity: O(1)

Merge Sort

Merge sort uses divide and conquer strategy, constantly halving the size of the array

```
Algorithm MergeSort (recursive)
 Split array in two halves
 MergeSort the first half
 MergeSort the second half
 Merge the two sorted halves
End
```

```
public static void mergeSort(int[] list) {
    if (list.length > 1) { // ==1: base case
        int[] firstHalf = new int[list.length / 2];
        int[] secondHalf = new int[list.length - list.length / 2];
        System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
```

```
        System.arraycopy(list, list.length / 2, secondHalf, 0, list.length
- list.length / 2);
        mergeSort(firstHalf);
        mergeSort(secondHalf);
        merge(firstHalf, secondHalf, list);
    }
}

public static void merge(int[] list1, int[] list2, int[] list) {
    int list1Index = 0;
    int list2Index = 0;
    int listIndex = 0;
    while (list1Index < list1.length && list2Index < list2.length) {
        if (list1[list1Index] < list2[list2Index]) list[listIndex++] =
list1[list1Index++];
        else list[listIndex++] = list2[list2Index++];
    }
    while (list1Index < list1.length) list[listIndex++] =
list1[list1Index++];
    while (list2Index < list2.length) list[listIndex++] =
list2[list2Index++];
}
```

```
Splitting the array in halves
  (log n) iterations (n/2^k = 1)
 Merging halves
  (n) iterations (worst case)
```

Time compelxity: O(n log n)-Log Linear

```
Splitting the array in halves
 n/2, n/4, n/8, …
 Total number of elements
 (n/2 + n/4 + n/8 + …) ≃ n
```

Space complexity: O(n)

Quick Sort

Quick sort is used to divide and conquer

- Divide two lists into partially stored lists using a pivot
    - part 1: all elements less than pivot

```
Algorithm QuickSort (recursive)
 Select a pivot
```

```
 Partition array in two parts
 Part1: Elements less than the pivot
 Part2: Elements greater than the pivot
 QuickSort(part1)
 QuickSort(part2)
End
```

```
public static void quickSort(int[] list) {
      quickSort(list, 0, list.length - 1);
  }

  public static void quickSort(int[] list,
          int first, int last) {
      if (last > first) {
          int pivotIndex = partition(list, first, last);
          quickSort(list, first, pivotIndex - 1);
          quickSort(list, pivotIndex + 1, last);
      }
  }

  public static int partition(int list[],
          int first, int last) {
      int pivot;
      int index, pivotIndex;
      pivot = list[first];// pivot is the first element
      pivotIndex = first;
      for (index = first + 1; index <= last; index++)
          if (list[index] < pivot) {
              pivotIndex++;
              swap(list, pivotIndex, index);
          }
      swap(list, first, pivotIndex);
      return pivotIndex;
  }

  public static void swap(int[] list, int i1, int i2) {
      int temp = list[i1];
      list[i1] = list[i2];
      list[i2] = temp;
  }
```

```
Partitioning the array in ~ halves
 (log n) iterations (average)
 Arranging elements around the pivot
 (n) iterations - worst case
```

Time complexity: average case O(n log n) - Log Linear

Space complexity

```
Partitioning the array not in halves
(n) iterations - worst case
Arranging elements around the pivot
(n) iterations - worst case
```

**Time complexity: O(n2)- worst case Space complexity: O(1) **

Heap Sort

Sort using heap data strucutre

- Data to be sorted is added to heap
- As data is removed from the heap it is removed in decending order

```
public static <E extends Comparable<E>>
void heapSort(E[] list) {
Heap<E> heap = new Heap<>();
 for(int i=0; i<list.length; i++){
  heap.add(list[i]);
 }
for (int i=list.length-1; i>=0; i--) {
  list[i] = heap.remove();
}
}
```

```
add() method - (O(log n))
 Trace path from a leaf to root
remove() method - (O(log n))
 Trace path from the root to a leaf
add() and remove() are called n times
to create the heap and to get the
sorted data - (O(n))
```

Heap Sort: Worst case O(n log n) - Log Linear

```
Heap with n nodes required to sort the
array list
```

Heap Sort space complexity: O(n)

Quadratic and log linear sorting algorithims 5/3/2022

**The ceiling of possibility for sorting algorihms that use comparables is (nlogn)

Bucket Sort

- For integers only
- Range of values to be sorted - [0,t]
- use (t+1) buckets
- An element equal to i is put in bucket i
- A bucket holds all the elemetns with the same value

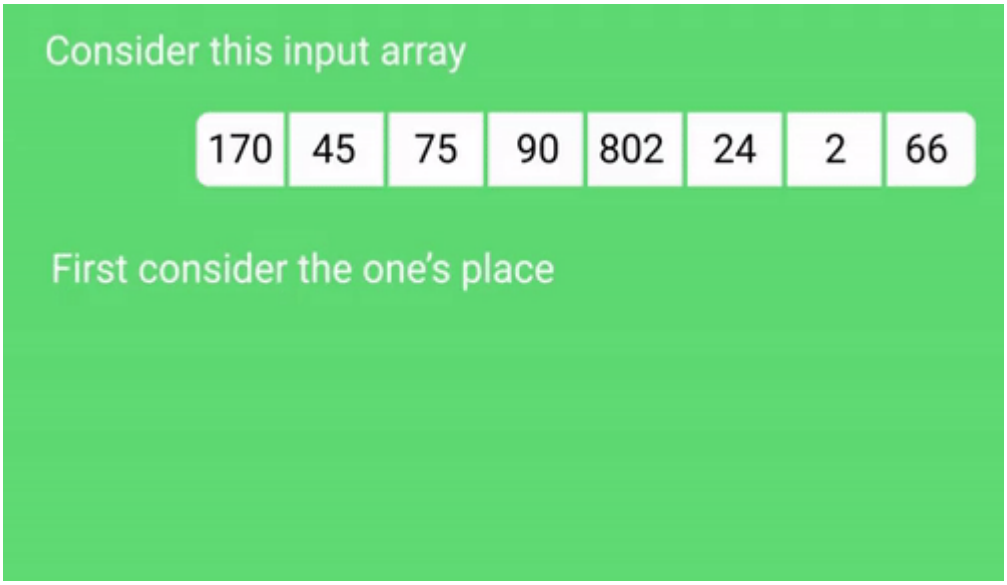
```
Algorithm bucketSort(list)
create t+1 buckets
 (t is the maximum value in the list)
for each value in list (n)
Assign value to bucket(value)
for each bucket i (0 to t)
 Assign the elements of bucket i to list
```

```
public static void bucketSort(int[] list) {
    int t = max(list);
    ArrayList < ArrayList < Integer >> buckets;
    buckets = new ArrayList < > (t + 1);
    for (int i = 0; i < t + 1; i++) buckets.add(new ArrayList < > ()); //
bucket i
    //Distribute the data on the buckets
    for (int i = 0; i < list.length; i++) {
        ArrayList < Integer > bucket = buckets.get(list[i]);
        bucket.add(list[i]);
    }
    // Move the data from the buckets back to the list
    int k = 0;
    for (int i = 0; i < buckets.size(); i++) {
        ArrayList < Integer > bucket = buckets.get(i);
        for (int j = 0; j < bucket.size(); j++) list[k++] = bucket.get(j);
    }
}
```

```
Creating the buckets - O(t)
 Distribute data on buckets - (O(n))
 Move data from buckets to list(O(n))
Bucket Sort:
 Time Complexity: O(n+t) - Linear
 Space Complexity: O(t)
Not desirable for large t
```

Radix Sort

- Bucket sort with fixed t buckets
- Radix-10 means 10 buckets (decimal numbers)
- Divide data into subgroups based on their radix positions
- Buck sort is applied on each radix position

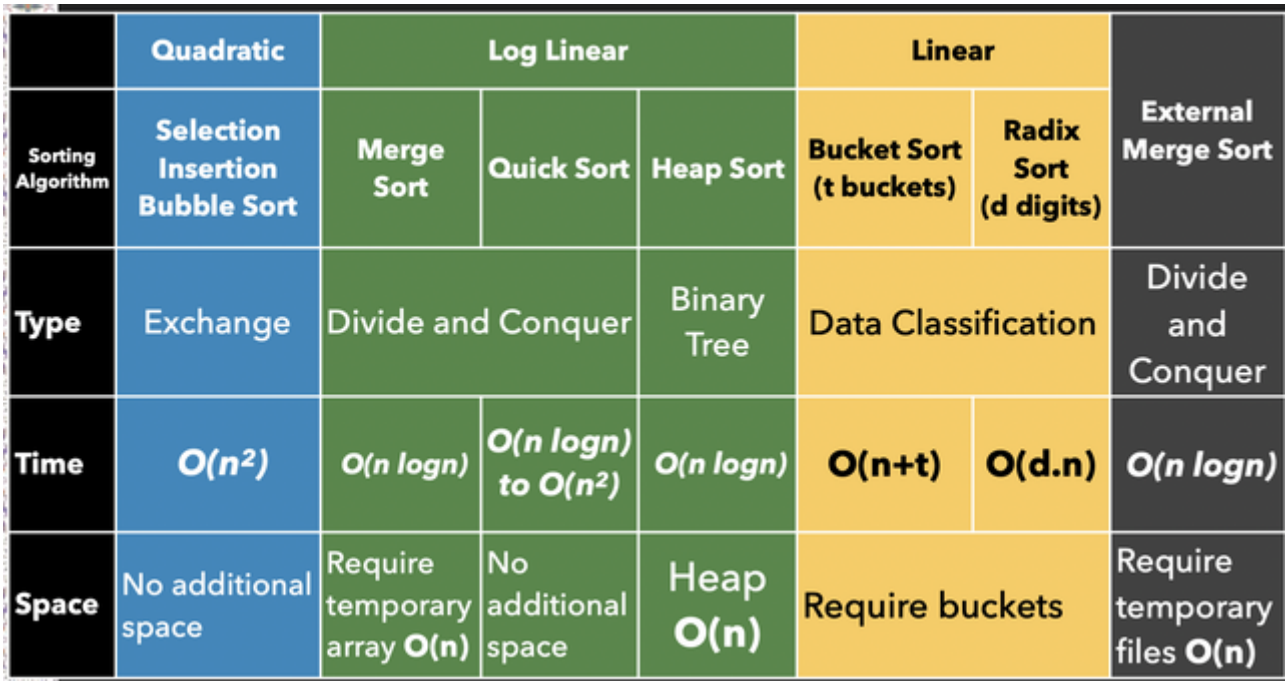



```
Algorithm radixSort(list)
 create 10 buckets
 max = number of digits of the largest value in list
 for each digit (0 to max-1){
 for each item in list
 Assign item to bucket number
 (item % 10(digit+1) / 10digit)
 for each bucket i (0 to 9)
 Assign the elements of bucket i to list
 clear all buckets
 }
End
```

```
 public static void radixSort(int[] list) {
        ArrayList<ArrayList<Integer>> buckets;
        buckets = new ArrayList<>(10); // 10 buckets
        Integer maxValue = max(list);
        int digits = maxValue.toString().length();
        for (int d = 0; d < digits; d++) {
            for (int j = 0; j < 10; j++) { // create buckets for iteration
d
                buckets.add(new ArrayList<>());
            }
            // Distribute the data on the buckets
            for (int j = 0; j < list.length; j++) {
                int digit = (list[j] % (int) (Math.pow(10, d + 1))) /
(int) (Math.pow(10, d));
                ArrayList<Integer> bucket = buckets.get(digit);
                bucket.add(list[j]);
```

```
            }
            // Move the data from the buckets back to the list
            int k = 0;
            for (int l = 0; l < 10; l++) {
                ArrayList<Integer> bucket = buckets.get(l);
                for (int j = 0; j < bucket.size(); j++)
                    list[k++] = bucket.get(j);
            }
            buckets.clear(); // for next iteration
        }
    }
```

```
Classifying the data into buckets- O(n)
 Classifying for each position- O(d)
Radix Sort:
 Time Complexity: O(d.n)
 Space complexity: O(1)
 d: maximum number of radix positions
```



	Quadratic	Log Linear			Linear		
Sorting Algorithm	Selection Insertion Bubble Sort	Merge Sort	Quick Sort	Heap Sort	Bucket Sort (t buckets)	Radix Sort (d digits)	External Merge Sort
Type	Exchange	Divide and Conquer		Binary Tree	Data Classification		Divide and Conquer
Time	$O(n^2)$	$O(n \log n)$	$O(n \log n)$ to $O(n^2)$	$O(n \log n)$	O(n+t)	O(d.n)	$O(n \log n)$
Space	No additional space	Require temporary array O(n)	No additional space	Heap O(n)	Require buckets		Require temporary files O(n)

Final Review

```
for (int i = n; i > 0; --i) n
 for (int j = n/20; j > 0; j /= 2) (log(n/20)+1)
  for(int k = 1; k < n; k *= 3) (log (n))
  int product = i * k + j * (k-2) + k;
```

Because Nested loops answer is n*(log(n/20)+1)*log(n)

Quick sort example

```
16 80 22 55 64 96 25 pivot=16
16* 80 22 55 64 96 25
16* 80 22 55 64 96 25 pivot=80
16* 80 22 55 64 25 96
16* 25 22 55 64 80* 96 pivot =25
16* 25 22 55 64 80* 96
16* 22 25* 55 64 80* 96
16* 22 25* 55* 64 80* 96 pivot =55
```

not radix or bucket or insertion or selection

Look at all binary tree traversal

Deleting node from binary tree moves largest value in right subtree to root

adding node in heap would add it to left most position on row and swap with roots till value is found

Heap as an array list starts from root to root children to their children from left to right

linear probing in hash table

```
34 29 53 44 120 39 45
capacity =4 lf=0.5 rehash when size=2

put(34)hash(34)=34%4=2,HT[2]=34,size=1
put(29):hash(29)=29%4=1, HT[1]=29 size=2
put(53): rehash - capacity=8 rehash when size=4
put(29):hash(29)=29%8=5,HT[5]=29
put(34):hash(34)=34%8=2,HT[2]=34
put(53):hash(53)=53%8=5,HT[6]=53
put(44):hash(44)=44%8=4 HT[4]=44
rehash since size=4, now size =16 and rehash when size=8
put(34):hash(34)=34%16=2,HT[2]=34
put(44):hash(44)=44%16=12, HT[12]=44
put(29): hash(29)=29%16=13, HT[13]=29
put(53): hash(120)=120%16, HT[8]=120
put(39): hash(39)=39%16=7 HT[7]=39
put(45): hash(45)=45%16=13, HT[14]=45
```