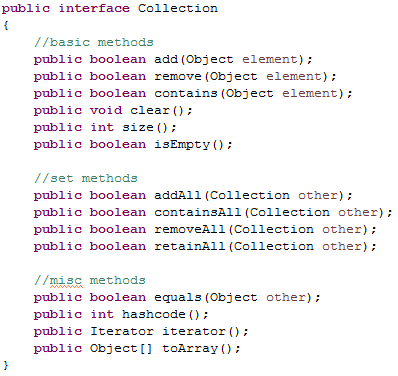
Midterm Study Guide

Miscellaneous

1. Name four of the primary methods that are part of the Collection<T> interface.

Add, remove, contains, size, isEmpty, clear

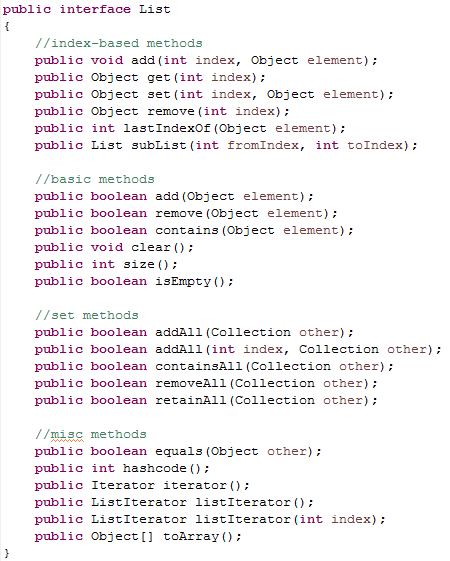


1. What is the relationship between the Collection<T> and List<T> interface?

List inherits from Collection

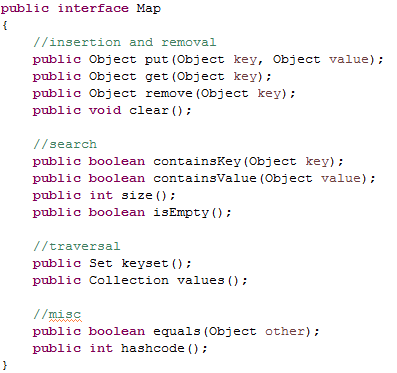
What methods are part of the List<T> interface, but not part of the Collection<T> interface?

Get, set, indexOf, add(index), remove(index),



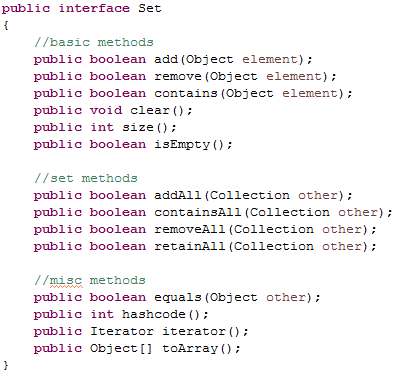
1. Name the primary two Set<T> classes included in Java's collection framework.

HashSet, TreeSet?



Name the two Map<K, V> classes.

HashMap, TreeMap



1. With respect to the Comparable<T> and Comparator<T> interfaces:
   1. What is the primary benefit of using the interfaces?

It allows you to compare objects that do not have a numeric value.

Things like addresses, people, or books.

If you can’t compare an object, you cannot sort it.

* 1. What methods are included in the interfaces?

compareTo() in Comparable & compare() in Comparator

* 1. Why would a developer prefer one interface over the other?

Comparator allows several sorts to happen sequentially and you can use separately in code segments

1. Alter the following class so that it uses the Comparable<T> interface. After your changes, you should be able to order books by page count:

public class Book implements Comparable<T>  
{  
 private String author;

private String title;

private int pageCount;  
  
 //constructors…  
  
 //methods…

public int compareTo(Book other) {

int authorSort = this.author.compareTo(other.author);

if (authorSort == 0) {

return this.title.compareTo(other.title);

} else {

return authorSort;

}

}

1. What is a wrapper class?

Each primitive type has an associated wrapper class that represents the primitive type when an object is required.

1. Name the eight wrapper classes provided in Java.lang.

|  |  |
| --- | --- |
| **Primitive** | **Wrapper Class** |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| boolean | Boolean |
| char | Character |

1. What is auto-boxing/unboxing?

Autoboxing is putting it in wrapper

Unboxing is returning it to primitive

1. What is type erasure? Be as clear as possible when answering this question.

public class DataStorage<T>  
{  
    private T data;  
  
    public DataStorage(T data)  
    {  
        this.data = data;  
    }  
  
    public T getData()  
    {  
        return data;  
    }

public T setData(T data)  
    {  
        this.data = data;  
    }  
}

When the compiler evaluates the two DataStorage declarations above, it will convert the generic DataStorage classes into two non-generic classes (with meaningless names), by replaced the generic type T in each class with the specified type given during instantiation

1. Write a simple generic class. Your class should have:
   1. A single generic type field
   2. A parameterized constructor that assigns your field.
   3. A getter/setter for the field
2. Inside of a generic class, you cannot directly instantiate a generic type. For example:

//create a new object of type T (this does not work!)

T newObject = new T();

Explain why this restriction makes sense.

Not all constructors take no parameters. Some would require you to pass in specific information before they could build your object, and when using a Generic it does not know what your object will need.

1. Change the following class header so that the generic type E also uses the interface *Printable*.

public class MyClass<E implements Printable>  
{  
 //fields, constructors, methods…  
}

1. Change the following class header so that the generic type V has *Shape* as a parent class.

public class Drawing<V extends Shape>  
{  
 //fields, constructors, methods…  
}

1. **True or false** A class can have more than one generic type.

## Algorithm Analysis

1. Why is it useful to use a simplified computer model when studying algorithms?

*For example, we assumed that the words size in our model was 32 bits, every operation takes the same amount of time to complete (one unit of work) and that we have infinite memory.*

1. What are the input and output for a runtime function T(n)?
2. Place each of the following runtime categories in order, from fastest to slowest.

1, log10n, n, n log10n, n2, n2(log10n), n3, 2n, 3n, n!

1. Which of the following are valid within big-oh notation?
   1. O(7n) – O(n)
   2. O(n + n) – O(n)
   3. O( ) – O(n)
   4. O(n logn) – O(n log10n)
   5. O(7n2) – O(n2)
2. For each of the following problems provide a solution in pseudocode.
   * Finding an element in an array of integers using sequential search. -- O(n)

algorithm indexOf(A[0...n-1], value)

//input: An ARRAY A of integers, an int value that might be in the array

//output: An int that is the index of a specified int in the ARRAY A

for i <-- 0 to n-1

if A[i] = value

return i

return -1

* + Given an array with the letters of the alphabet, print the vowels found in the array. – O(n)

algorithm printVowels(A[0...n-1])

//input: ARRAY A of letters

//output: prints vowels, no return value

vowels <-- [a, e, i, o, u]

for i <-- 1 to n - 1

if vowels.contains( A[i])

print A[i]

* + Finding an element in an array of integers using binary search. – O(log n)

algorithm Find X(A[0...n-1], X)

//input: A sorted List A of integers

//input: An integer X

//output: Boolean of whether X is found in the array

Low <-- 0

High <-- n-1

while Low <= High

SearchIndex <-- (High + Low) / 2

CompareValue <-- A[SearchIndex]

if CompareValue = X

return true

if CompareValue < X

High <-- SearchIndex -1

if CompareValue > X

Low <-- SearchIndex + 1

return false

* + Given two arrays of strings, call them A and B, return a new array with all string values in both A and B. O(A \* B)

algorithm Intersection(A[0...n-1], X)

//input: A List A of unique strings (no duplicates)

//input: A List B of unique strings (no duplicates)

//output: an array listing strings that were in both A and B

t1[] <-- new ARRAY

for i <-- 1 to n - 1

for j <-- 1 to n - 1

if A[i] = B[j]

t1.add(A[i])

return t1

1. For each of your answers to (5), offer a big-oh analysis. Justify your answers.

A good justification:

* Offers a runtime function T(n) and then gives the big-oh of T(n).
* An argument about the loops or control structures encountered in your solution.
* An argument using the principles we've covered in our class.

1. What do the following notations signify? Be as clear as possible when answering this question (you do not need to give the mathematical definitions).
   * f(n) = Ω(g(n)) – Omega is the tightest lower bound
   * f(n) = Ɵ(g(n)) – Theta is the tightest upper bound
2. For each of the following scenarios, state a problem where the estimation given might be useful
   * Best case scenario –

When you know you will be using it in that scenario like making a Stack or Queue with a List or Bubble sort on something almost sorted

* + Average case scenario

If only rarely will it need to do the Worst Case Scenario it can help you decide whether to use one algorithm that has a better worst case scenario but on average the scenario will run significantly longer than the algorithm you are proposing.

* + Worst case scenario

What is the maximum amount of work it will take to run the algorithm? What if you do a search and the element isn’t in the List, or you need to add at the beginning of an array. You need to know how it compares to a different algorithm to make a decision on which one to implement for your current use.

1. Which of the scenarios from (Question 8) are we primarily concerned with in algorithm analysis? Why?

## Lists, Stacks, Queues

1. Give a tight big-oh estimate of the following operations in a singly-linked list (with tail pointer):

|  |  |
| --- | --- |
| Operation | Big-oh |
| add(element) | 1 |
| add(index, element) | n |
| addFirst(element) | 1 |
| addSecond(element) | 1 |
| remove(element), remove(index) | n |
| removeFirst(element) | 1 |
| removeLast(element) | n |
| contains(element) | n |
| get(index), set(index) | n |

1. Give a tight big-oh estimate of the following operations in a doubly-linked list (with tail pointer):

|  |  |
| --- | --- |
| Operation | Big-oh |
| add(element) | 1 |
| add(index, element) | n |
| addFirst(element) | 1 |
| addSecond(element) | 1 |
| remove(element), remove(index) | n |
| removeFirst(element) | 1 |
| removeLast(element) | 1 |
| contains(element) | n |
| get(index), set(index) | n |

1. Give a tight big-oh estimate of the following operations in an array list:

|  |  |
| --- | --- |
| Operation | Big-oh |
| add(element) | 1 |
| add(index, element) | n |
| addFirst(element) | n |
| addSecond(element) | n |
| remove(element), remove(index) | n |
| removeFirst(element) | n |
| removeLast(element) | 1 |
| contains(element) | n |
| get(index), set(index) | 1 |

1. What is an amortized running time?

On average

Why is this an important concept when discussing the add() method in the array list class?

When the array gets full you’ll have to increase its size, which will be a lot of work to switch everything over to the new array, but you rarely have to do that work so overall its cost does not impact the run time.

## Iterators

1. What are iterators and why are they important? Powers for each loops. Allows you go to through every element in a list or collection
2. Name the two interfaces that are used with the iterator design pattern. Iterable and Iterator
3. Suppose you are given an array list called "myWords." Write a code segment that uses the array list iterator to print out each element in myWord.

*Note: You should not be using a for-each loop in your answer.*

1. Suppose you are given the following Color class. Finish the inner class below so that it becomes an iterator that prints out the individual color values in the class.

*Note: You do not need to write a remove() method.*

public class Color  
{  
 //fields  
 private int red;

private int green;

private int blue;

private int alpha;

//constructors…

//methods…

//here is an inner class

public class ColorIterator

{

}  
}

## Searching

1. What is the big-oh of the following search routines for the best, worst and average cases?
2. Linear search -- Best – O(1), Worst – O(n), Avg – O(n)
   1. Binary search -- Best – O(1), Worst – O(logn), Avg – O(logn)
   2. Interpolation search -- Best – O(1), Worst – O(n?), Avg – O(log(logn))
3. State the *repeated halving principle*.
4. What prerequisite must be true before applying the binary search or interpolation search algorithm on an input array?

Binary must be sorted

Interpolation must be sorted and values should be increasing at a consistent rate/pattern

1. Given the following array and search value:
   * Give the steps taken by the binary search algorithm during the search.
   * Give the output of the algorithm.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| -20 | -3 | -1 | 0 | 11 | 41 | 56 | 77 | 78 | 83 | 99 | 101 | 122 | 124 | 150 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

1. Search = 56

14+0 /2 = 7, 77 > 56, 6+0/2 = 3, 0 < 56, 6+4 /2 = 5, 41<56, 6+6/2 = 6, 56 =56, return true

1. Search = 150

14+0 /2 = 7,77<150, 14+8/2 = 11, 101<150, 12+14/2=13, 124<150, 14+14/2 = 14, 150 = 150 return true

1. Search = -10

14+0 /2 = 7, 77 > -10, 6+0/2 = 3, 0 > -10, 2+0 /2 = 1, -3 > -10, 0 +0 /2 = 0, -20 < -10 return false

1. In the recursive version of the binary search algorithm, is it possible to get a stack overflow error? Explain why or why not. Log of 1 million is only around 20 steps, so unless you have petabytes of information it would be hard to overflow.
2. What advantage does interpolation search have over binary search? It can find the index of the value at log(logn) time almost constant time.

What disadvantages? If the data is not increasing at a consistent slope it is unable to accurately determine the index of the element you are searching.

Hashing

1. Match each of the following terms with their corresponding definitions:

|  |  |
| --- | --- |
| Term | Definition |
| Hash Code | A number or string that is used to represent an object. This value represents a summarization of the object and is often called a “digest.” |
| Hash Table | An array that stores elements using Hash Codes. |
| Collision | This occurs when two elements would be placed in the same location in a Hash Table, given their hash codes. |
| Load Factor | The percent of used spots in a Hash Table which, if exceeded, results in a resize of the Hash Table. |
| Linear/Quadratic Probing | A collision resolution strategy where we look to adjacent spots in a Hash Table if a collision occurs. |
| Chaining | A collision resolution strategy where linked lists are used to hold elements at every position in a Hash Table. When a collision occurs, the new element is added to a linked list at the position determined by the element’s hash code. |

1. Given the following Hash Table and a load factor of 50%:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| B | A |  |  |  | C |  | D |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| B |  |  |  |  |  |  | F | H | A |  | G | E | C |  | D |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  | F |  | A |  | G | E | C | J | D | B | K |  |  |  |  |  |  | H |  |  |  |  |  |  | I |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |

Add each of the following elements to the Hash Table above:

|  |  |
| --- | --- |
| Element | Hash Code |
| A | 9 |
| B | 16 |
| C | 13 |
| D | 15 |
| E | 12 |
| F | 7 |
| G | 11 |
| H | 24 |
| I | 31 |
| J | 13 |
| K | 15 |

*Note: You should resolve any collisions using Linear Probing.*

*Note: You should show the steps of each add() operation. This includes how you arrived at a location for each new element as well as the state of the Hash Table after each call to add().*

1. Explain why the following rule exists: “if you override equals(), then you must also override hashcode().” If you override .equals your hashes could go in different buckets and the map won’t realize and keep both values.
2. Suppose you have the following Hash Table:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Giraffe** | **Koala** | **Bear** | **Antelope** | null | null | null | **Platypus** | null | null |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Assuming the element “Bear” has a hash code of 160, explain the steps involved when calling remove(“Bear”).

160 % 10 = 0, Giraffe != Bear, look at 1, Koala != Bear, look at 2, Bear = Bear, mark Bear as deleted

1. It is important not to replace an element in a Hash Table with a null value when remove() is called. For example, the method call from (4) should not result in the following table:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Giraffe** | **Koala** | null | **Antelope** | null | null | null | **Platypus** | null | null |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Explain why this is an important feature of a hash table class.

Since there is linear probing if 2 is null searching for Antelope might give a false negative if Antelopes hash number is 0, 1, or 2

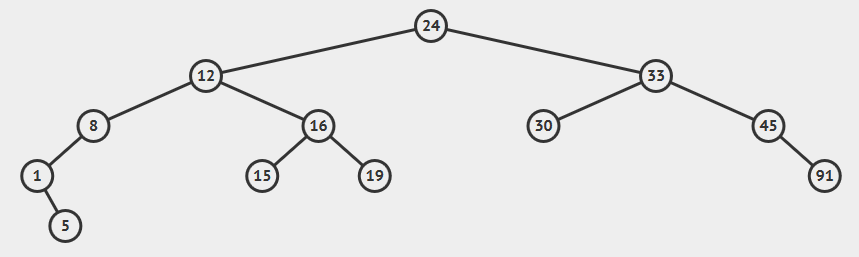
1. Give three strategies that help to prevent collisions in a Hash Table.

Hire mathematician to write hashCode, use a large prime number for table size, keep load factor low so table is not full

1. What are the advantages and disadvantages of a high load factor?
2. Conversely, what are the advantages and disadvantages of a low load factor?
3. Given a relatively small number of collisions in a hash table, give a big-oh estimate of the following operations on a hashing structure:
   1. insert(x):
   2. remove(x):
   3. contains(x):
   4. size():
4. Given repeated collisions in a hash table, give a big-oh estimate of the following operations on a hashing structure according to the worst-case scenario:
   1. insert(x):
   2. remove(x):
   3. contains(x):
   4. size():

## Binary Search Trees

1. For the following tree, identify the root, internal nodes and leaf nodes.



Root = 24

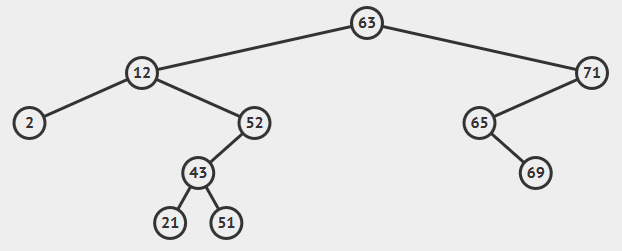
Leaf nodes = 5, 15, 19, 30, 91

Internal nodes = 1, 8, 12, 16, 33, 45

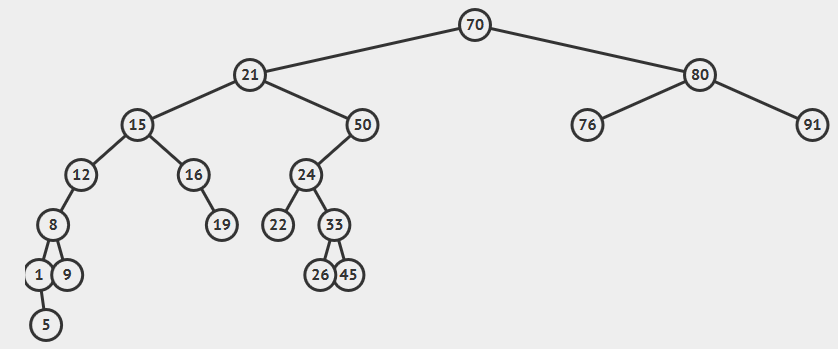
1. Give the depth and height of each node from (1).

Depth = 3 , height = 1

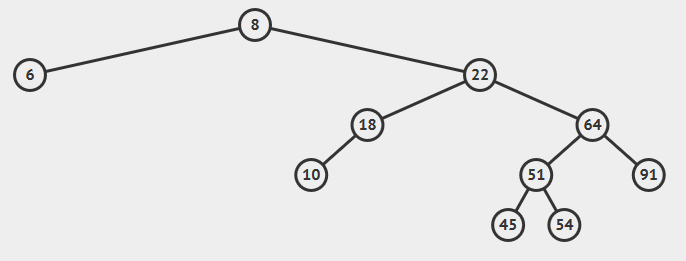
1. Why is the order of insertion important for a basic binary search tree?
2. Give the state of the binary search tree below after each of the following operations:
   1. add(64)
   2. add(55)
   3. add(54)
   4. add(65)
   5. remove(65)
   6. remove(63)



1. Given the basic binary search tree described in your book, is it possible to get a stack overflow error when calling the add() method? Explain why or why not.
2. For the following tree give the pre-order, in-order and post-order traversals.

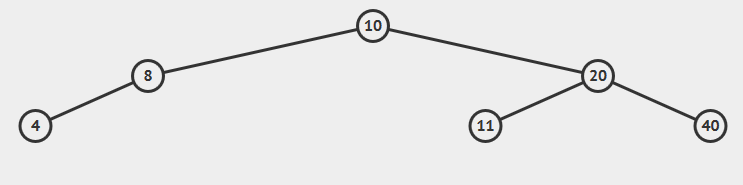


1. For the following tree, give the breadth-first (level-order) search on the tree using the algorithm we described in class. Show your work!



1. Show the following insertions in an AVL tree step-by-step. Be careful to perform the correct rotations during insertion: 40, 20, 10, 4, 8, 11.

*Note: your tree should look like the following after all insertions are complete.*

**

1. Show the following insertions in an AVL tree step-by-step. Be careful to perform the correct rotations during insertion: 22, 11, 33, 15, 40, 50, 36, 37.

*Note: your tree should look like the following after all insertions are complete.*

