Week 13: Lecture 1, 2, 3, 8 Review

- Arrays, Objects, Classes
  - Array operations, functions, method overloading, strings, pass by value, immutability
- OOP Thinking, Inheritance, Polymorphism
  - OOP concepts, equals, method overriding
- Abstract Classes, Interfaces
  - Defining and implementing interfaces, extending abstract classes, comparable, comparator
- Developing Efficient Algorithms
  - Time/space complexity, basic sorting algorithms, logarithmic-time algorithms

Algorithm Complexity

- Short answer:

Consider this algorithm algo.

```
1.  public boolean algo(int[] a) {
2.      for (int i = 0; i < a.length; i++) {
3.          for (int j = i + 1; j < a.length; j++) {
4.              if (a[i] == a[j]) {
5.                  return true;
6.              }
7.          }
8.      }
9.      return false;
10. }
```

What is the time complexity of this algorithm?

Answer:

Algorithm Complexity

- Complete the blank:

The worst-case time complexity of the following algorithm is O( ).

```
1.  public int algo(int[] a, int k) {
2.      int n = a.length;
3.      for (int i = 0; i < n; i++) {
4.          if (a[i] == k) {
5.              return i;
6.          }
7.      }
8.      return -1;
9.  }
```

Lab Group 1 Q1, Q2, Q3, Q4, Q5, Q6

- Complete the class so that the Book class is **immutable**:

```
1  public class Book {
2      private String title;
3      Q1 String[] authors;
4
5      public Book(String title, Q2 authors) {
6          Q3 .title = title;
7          this.authors = new String[ Q4 ];
8          System.arraycopy(authors, 0, this.authors, 0, authors.length);
9      }
10
11     public Book(Book other) {
12         Q5 (other.title, other.authors);
13     }
14 }
```

- In line 11-13, we have an implementation of a/an Q6

Lab Group 1 Q1, Q2, Q3, Q4, Q5, Q6

- Complete the class so that the Book class is **immutable**:

```
1  public class Book {
2      private String title;
3      Q1 String[] authors;
4
5      public Book(String title, Q2 authors) {
6          Q3 .title = title;
7          this.authors = new String[ Q4 ];
8          System.arraycopy(authors, 0, this.authors, 0, authors.length);
9      }
10
11     public Book(Book other) {
12         Q5 (other.title, other.authors);
13     }
14 }
```

- In line 11-13, we have an implementation of a/an Q6

Lab Group 2 Q1, Q2, Q3, Q4, Q5, Q6

- Complete the class so that the Person class is **immutable**:

```
1  public class Person {
2      private String name;
3      Q1 Date dateOfBirth;
4
5      public Person(String name, Date dateOfBirth) {
6          Q2 .name = name;
7          this.dateOfBirth = Q3 (dateOfBirth.getTime());
8      }
9
10     Q4 Date getDateOfBirth() {
11         return Q5 (dateOfBirth.getTime());
12     }
13 }
```

- In line 11, we use a technique called Q6 to return a fresh instance of an object instead of a reference to the instance variable

Lab Group 2 Q1, Q2, Q3, Q4, Q5, Q6

Lab Group 2 Q1, Q2, Q3, Q4, Q5, Q6

- Complete the class so that the Person class is **immutable**:

```
1  public class Person {
2      private String name;
3      Q1 Date dateOfBirth;
4
5      public Person(String name, Date dateOfBirth) {
6          Q2 .name = name;
7          this.dateOfBirth = Q3 (dateOfBirth.getTime());
8      }
9
10     Q4 Date getDateOfBirth() {
11         return Q5 (dateOfBirth.getTime());
12     }
13 }
```

Q1: private
Q2: this
Q3: Date
Q4: public
Q5: Date

not imm because String is immutable

- In line 11, we use a technique called Q6 to return a fresh instance of an object instead of a reference to the instance variable

Q6: defensive copying

Lab Group 2 Q1, Q2, Q3, Q4, Q5, Q6

Lab Group 1 Q7, Q8, Q9

- Complete the classes:

```
1  class Employee {
2      void Q7 {
3          System.out.println("Employee is working");
4      }
5  }
6  class Manager Q8 Employee {
7      @Override
8      void work() {
9          Q9.work();
10         System.out.println("Manager is overseeing");
11     }
12 }
13 public class Program {
14     public static void main(String[] args) {
15         Manager manager = new Manager();
16         manager.work();
17     }
18 }
```

- Output:

```
Employee is working
Manager is overseeing
```

Lab Group 1 Q7, Q8, Q9

- Complete the classes:

```
1  class Employee {
2      void Q7 {
3          System.out.println("Employee is working");
4      }
5  }
6  class Manager Q8 Employee {
7      @Override
8      void work() {
9          Q9.work();
10         System.out.println("Manager is overseeing");
11     }
12 }
13 public class Program {
14     public static void main(String[] args) {
15         Manager manager = new Manager();
16         manager.work();
17     }
18 }
```

- Output:

```
Employee is working
Manager is overseeing
```

Lab Group 2 Q7, Q8, Q9

- Complete the classes:

```
1  class Person {
2      Q7 introduce() {
3          System.out.println("I am a person.");
4      }
5  }
6  class Student Q8 Person {
7      @Override
8      void introduce() {
9          Q9 ;
10         System.out.println("I am a student.");
11     }
12 }
13 public class Program {
14     public static void main(String[] args) {
15         Student student = new Student();
16         student.introduce();
17     }
18 }
```

Lab Group 2 Q7, Q8, Q9

Lab Group 2 Q7, Q8, Q9

- Complete the classes:

```
1  class Person {
2      Q7 introduce() {
3          System.out.println("I am a person.");
4      }
5  }
6  class Student Q8 Person {
7      @Override
8      void introduce() {
9          Q9;
10         System.out.println("I am a student.");
11     }
12 }
13 public class Program {
14     public static void main(String[] args) {
15         Student student = new Student();
16         student.introduce();
17     }
18 }
```

Q7: void

Q8: extends

Q9: super.introduce()

- Output:

```
I am a person.
I am a student.
```

Lab Group 2 Q7, Q8, Q9

- Output:

```
I am a person.
I am a student.
```

Lab Group 1 Q10, Q11, Q12

- Complete the class:

```
1  public class Book {
2      private String title;
3      private String author;
4
5      public Book(String title, String author) {
6          this.title = title;
7          this.author = author;
8      }
9
10     @Override
11     public boolean equals(Object obj) {
12         if (obj == null) return false;
13         if (this == obj) return true;
14         if (!(obj instanceof Book)) return false;
15         Book book = (Book) obj;
16         return this.title.equals(book.title) && this.author.equals(book.author);
17     }
18 }
```

Lab Group 1 Q10, Q11, Q12

Lab Group 1 Q10, Q11, Q12

- Complete the class:

```
1  public class Book {
2      private String title;
3      private String author;
4
5      public Book(String title, String author) {
6          this.title = title;
7          this.author = author;
8      }
9
10     @Override
11     public boolean equals( Q10 ) {
12         if (obj == null) return false;
13         if (this == obj) return true;
14         if (!(obj instanceof Book)) return Q11
15         Book book = (Book) obj;
16         return Q12
17     }
18 }
```

Object obj

false;

this.title.equals(book.title) &&

Lab Group 1 Q10, Q11, Q12

```
1  public class Book {
2      private String title;
3      private String author;
4
5      public Book(String title, String author) {
6          this.title = title;
7          this.author = author;
8      }
9
10     @Override
11     public boolean equals(Object obj) {
12         if (obj == null) return false;
13         if (this == obj) return true;
14         if (!(obj instanceof Book)) return false;
15         Book book = (Book) obj;
16         return this.title.equals(book.title) && this.author.equals(book.author);
17     }
18 }
```

Q10

Q11

Q12

Lab Group 2 Q10, Q11, Q12

- Complete the class:

```
1  public class Employee {
2      private int id;
3      private String dept;
4
5      public Employee(int id, String dept) {
6          this.id = id;
7          this.dept = dept;
8      }
9
10     @Override
11     public boolean equals(Object that) {
12         if (that == null) return Q10
13         if (this == that) return true;
14         if (!( Q11 )) return false;
15         Employee thatEmp = (Employee) that;
16         return Q12
17     }
18 }
```

Lab Group 2 Q10, Q11, Q12

```java
public class Employee {
    private int id;
    private String dept;

    public Employee(int id, String dept) {
        this.id = id;
        this.dept = dept;
    }

    @Override
    public boolean equals(Object that) {
        if (that == null) return false;
        if (this == that) return true;
        if (!(that instanceof Employee)) return false;
        Employee thatEmp = (Employee) that;
        return this.id == thatEmp.id && this.dept.equals(thatEmp.dept);
    }
}
```

Lab Group 2 Q10, Q11, Q12

```java
public class Employee {
    private int id;
    private String dept;

    public Employee(int id, String dept) {
        this.id = id;
        this.dept = dept;
    }

    @Override
    public boolean equals(Object that) {
        if (that == null) return false;
        if (this == that) return true;
        if (!(that instanceof Employee)) return false;
        Employee thatEmp = (Employee) that;
        return this.id == thatEmp.id && this.dept.equals(thatEmp.dept);
    }
}
```

Lab Group 1 Q13, Q14, Q15

- Complete the class:

```
1  public class Employee implement Q13 {
2      private double salary;
3
4      @Override
5      public int compareTo(Object obj) {
6          Employee that = (Employee) obj;
7          if (this.salary > that.salary)
8              return 1;
9          if ( Q14 )
10             return 0;
11         return Q15
12     }
13 }
```

Lab Group 1 Q13, Q14, Q15

Lab Group 1 Q13, Q14, Q15

- Complete the class:

Comparable

```
1  public class Employee implement Q13 {
2      private double salary;
3
4      @Override
5      public int compareTo(Object obj) {
6          Employee that = (Employee) obj;
7          if (this.salary > that.salary)
8              return 1;
9          if ( Q14 )
10             return 0;
11         return Q15
12     }
13 }
```

this.salary == that.salary

-1

Lab Group 1 Q13, Q14, Q15

Lab Group 2 Q13, Q14, Q15

- Complete the class:

```
1  public class Person implements Comparable {
2      private int age;
3
4      @Override
5      public int compareTo( Q13 ) {
6          Person person = (Person) other;
7          if (this.age > person.age)
8              return 1;
9          if ( Q14 )
10             return -1;
11         return Q15
12     }
13 }
```

Lab Group 2 Q13, Q14, Q15

Lab Group 2 Q13, Q14, Q15

- Complete the class:

```
1  public class Person implements Comparable {
2      private int age;
3
4      @Override
5      public int compareTo( Q13 ) {
6          Person person = (Person) other;
7          if (this.age > person.age)
8              return 1;
9          if ( Q14 )
10             return -1;
11         return Q15
12     }
13 }
```

Object other

this.age < person.age

0;

Lab Group 2 Q13, Q14, Q15

Generics

CPT204 Advanced Object-Oriented Programming

Lecture 4 Generics

What are Generics?

- ***Generics*** is the capability to ***parameterize types***
  - With this capability, you can define a class or a method with generic types that can be substituted using concrete types by the compiler
  - You may define a generic stack class that stores the elements of a generic type
    - From this generic class, you may create:
      - a stack object for holding Strings
      - a stack object for holding numbers

    Strings and numbers are concrete types that replace the generic type

Why Generics?

- The key benefit of generics is to enable errors to be detected at compile time rather than at runtime
  - A generic class or method permits you to specify allowable types of objects that the class or method may work with
    - We still do **code reuse**, e.g., write a single implementation for a special kind of data structure, like a single implementation of a generic stack and its standard methods
  - Most important advantage: If you attempt to use the class or method with an incompatible object, a **compile error** occurs

W4 - Sample Questions on Generic

- Conceptual & Programming
- Example 1

Explain the following Java code using plain English.

```
public class Foobar< T > { }
```

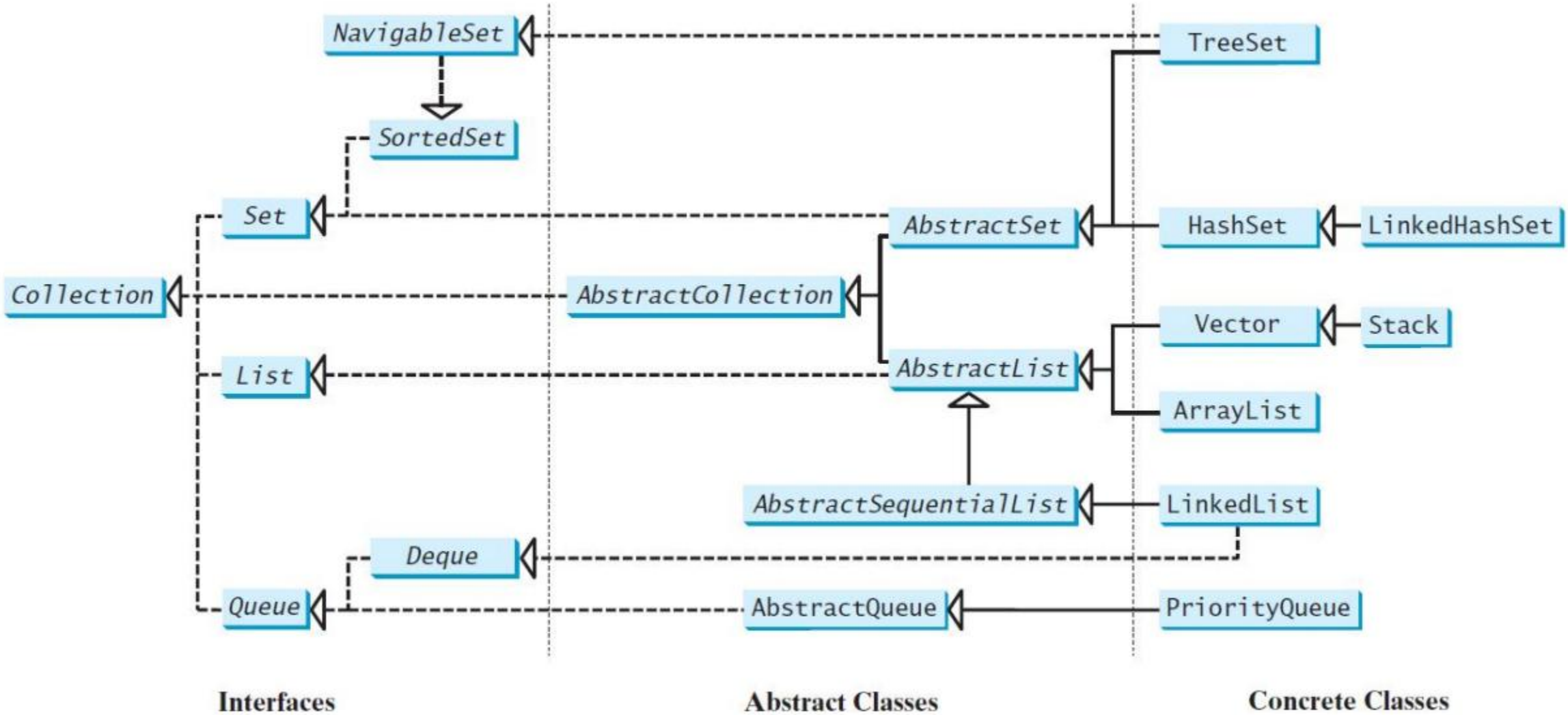
The code declares a [ ] class named Foobar with a single type parameter T.

- Example 2
  - Write Java code that declares a generic public class named Foobar with a single type parameter T.

Lists, Stacks, Queues, and Priority Queues

CPT204 Advanced Object-Oriented Programming

Lecture 5 Lists, Stacks, Queues, and Priority Queues


NavigableSet
SortedSet
Set
Collection
List
Deque
Queue
AbstractSet
AbstractCollection
AbstractList
AbstractSequentialList
AbstractQueue
TreeSet
HashSet
LinkedHashSet
Vector
Stack
ArrayList
LinkedList
PriorityQueue
Interfaces
Abstract Classes
Concrete Classes

Iterators

- Each collection is **`Iterable`**
  - ***Iterator*** is a classic design pattern for walking through a data structure without having to expose the details of how data is stored in the data structure
    - Also used in for-each loops:

```
for(String element: collection)
        System.out.print(element + " ");
```

- The **`Collection`** interface extends the **`Iterable`** interface
  - You can obtain a collection **`Iterator`** object to traverse all the elements in the collection with the **`iterator()`** method in the **`Iterable`** interface which returns an instance of **`Iterator`**
    - The **`Iterable`** interface defines the **`iterator`** method, which returns an **`Iterator`**

W5 - Sample Questions on Lists, Stacks, Queues, and Priority Queues

- Conceptual & Programming
- Example 1

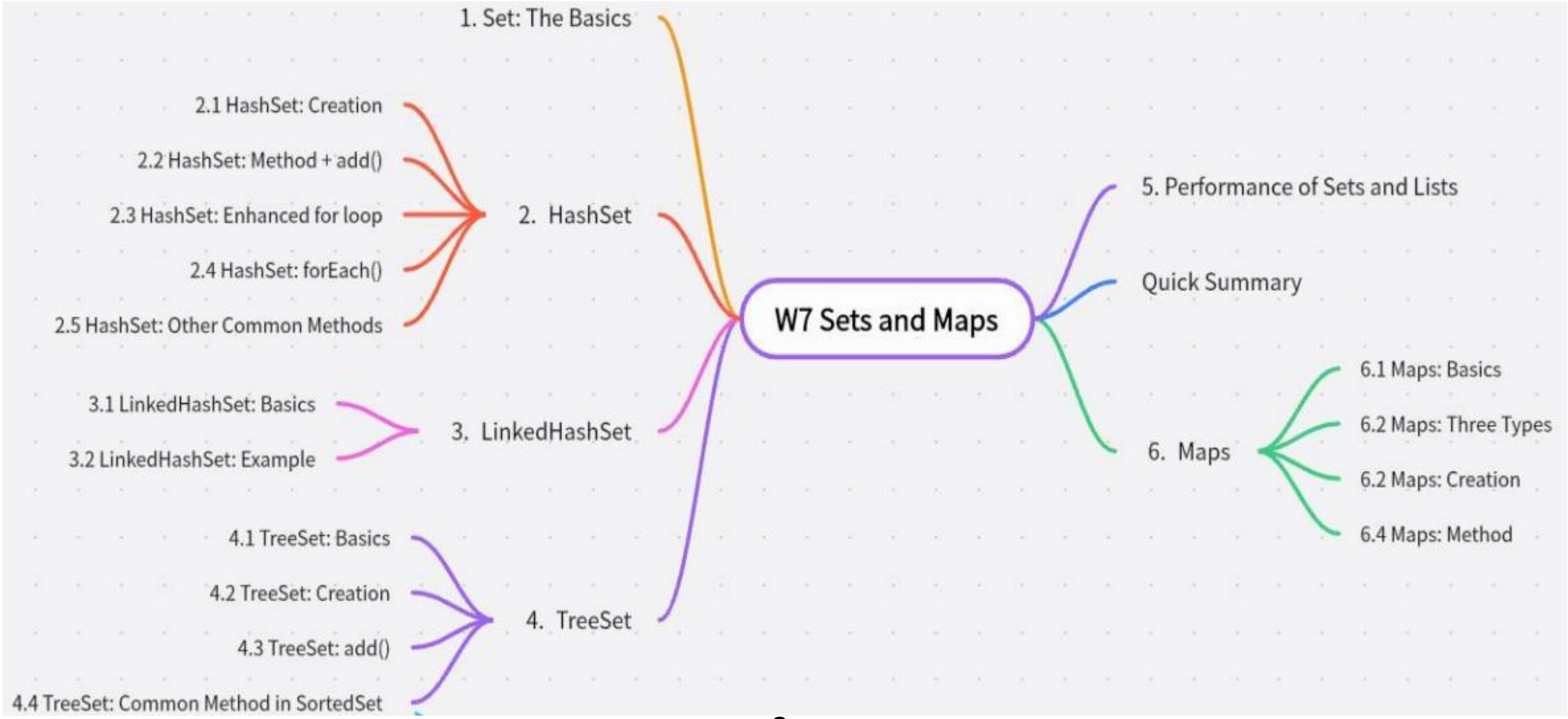
In a [ ], elements are assigned priorities and the element with the highest priority is removed first.

- Example 2
  - Write Java code that (1) declares a priority queue of String type; (2) adds “A”, “a”, “1” into the priority queue; (3) output the 3 Strings.
- Just because we use priority queue as examples above, doesn't mean other containers are not important.

Sets and Maps

CPT 204 - Advanced OO Programming

Content


W7 Sets and Maps
1. Set: The Basics
2. HashSet
2.1 HashSet: Creation
2.2 HashSet: Method + add()
2.3 HashSet: Enhanced for loop
2.4 HashSet: forEach()
2.5 HashSet: Other Common Methods
3. LinkedHashSet
3.1 LinkedHashSet: Basics
3.2 LinkedHashSet: Example
4. TreeSet
4.1 TreeSet: Basics
4.2 TreeSet: Creation
4.3 TreeSet: add()
4.4 TreeSet: Common Method in SortedSet
5. Performance of Sets and Lists
Quick Summary
6. Maps
6.1 Maps: Basics
6.2 Maps: Three Types
6.2 Maps: Creation
6.4 Maps: Method

Set: The Basics

- **Set interface** is a sub-interface of **Collection**
- It extends the **Collection,** but does not introduce new methods or constants.
- However, the **Set interface stipulates** that an instance of **Set contains no duplicate elements**
  - That is, **no two elements e1** and **e2** can be in the set such that **e1.equals(e2)** is true

W6 - Sets and Maps

- Conceptual & Programming
- Example 1

In a List, the indexes are integers. However, in a Map, the keys can be

- Example 2
  - Write a Java statement that (1) declares a Hash Set of String type; (2) adds “A”, “a”, “1” into the hash set; (3) output the 3 Strings.
- Just because we used hash set as examples above, doesn't mean other containers are not important

Revision W9-12

CPT 2024/25

Week 9 - Part 1 Sorting (Process, Time Complexity and Codes)

- **Bubble Sort:**
  - Process: Each pass compares adjacent elements (list[i] and list[i+1]); swaps if list[i] > list[i+1], moving the largest element to the end. Repeats until no swaps or all passes done, with needNextPass to stop early.
  - Time Complexity: Best O(n), Worst $O(n^2)$.
  - Code: The outer loop (passes) and inner loop (comparisons/swaps) - page 3 and 5.
- **Merge Sort:**
  - Process: Recursively splits array into halves until single elements (base case), then merges by comparing elements with pointers (current1, current2, current3) into a sorted array.
  - Time Complexity: O(n log n).
  - Code: Recursive calls and merge logic (page 9-13).
- **Quick Sort:**
  - Process: Picks a pivot, partitions using low and high pointers (moves low right if ≤ pivot, high left if > pivot, swaps, repeats), replaces pivot, and do it recursively on subarrays.
  - Time Complexity: Best/Average O(n log n), Worst $O(n^2)$.
  - Code: Partition logic and recursive calls - Pages 17-19 (partition on Page 17, quickSort on Page 19-20)
- **Heap Sort:**
  - Process: Builds a max heap, repeatedly removes the root, places it at the array's end, and reorganizes the heap for next removal
  - Time Complexity: O(n log n)
  - Code: Pages 34-37 (add on Page 34, remove on Pages 35-36, heapSort on Page 37)

Week 9 - Part 2 Life-Long Learning and EDI (Concepts and Pracrtices)

- **Life-long Learning:**
  - Definition and importance
  - Ways of recording and reflecting the learning process (e.g., reflective journal/log, etc)

- **EDI:**
  - Definition and importance
  - Practical ways

No need to recite everything, just understand the concepts/terms

E.g., _____principle ensures that all developers, regardless of gender, race, or background, have equal access to career opportunities and can contribute to software development

Week 10 - P1 Unweighted Graphs

- **Terminologies**
  - Weighted v.s. unweighted, directed v.s. undirected, loops, parallel edegs, etc

- **Graph Representation**
  - Represent vertices using arrays, lists, or objects (e.g., City class)
  - Represent edges using edge arrays (e.g., int[][] edges), edge objects (e.g., Edge class), adjacency matrices (1 for edge, 0 for none), adjacency vertex lists (list of neighbor indices), or adjacency edge lists (list of Edge objects)

- **DFS**
  - DFS Process: Start at a vertex, mark it as visited, explore each unvisited neighbor recursively as far as possible before backtracking, building a DFS spanning tree.
  - Time Complexity: O(|V | + |E |) (visiting each vertex and edge once).
  - Code: Pseudocode - Page 28, Recursive traversal in dfs methods - Page 32 (dfs and private dfs helper method).

- **BFS**
  - BFS Process: Start at a vertex, visit all its neighbors first, then visit their unvisited neighbors level by level using a queue, building a BFS spanning tree.
  - Time Complexity: O(|V | + |E |) (visiting each vertex and edge once).
  - Code: Pseudocode - Page 35, Queue-based traversal in bfs method - UnweightedGraph.java, line 207-235.

Week 10 - P2 Weighted Graphs

- **Similar but not the same graph representation** (page 41, e.g., , when using adjacency list for edges, weightedEdge class extending Edge class with new data field 'weight')

- **Minimum Spanning Tree (MST) - Prim's Algorithm:**
  - MST Process: Start with a vertex in set T, iteratively add the vertex v not in T with the smallest edge weight to a vertex u in T, updating costs and parents, until all vertices are included.
  - Time Complexity: $O(n^3)$ *(as implemented in the code provided in the class, not necessarily the standard answer)*
  - Code: pseudocode - Page 46, implementation - Page 49

- **Single Source Shortest Path Algorithm - Dijkstra's algorithm**
  - Shortest Path Process: Start at a source vertex, iteratively select the unvisited vertex with the smallest path cost from the source, update costs to neighbors (cost[v] = cost[u] + w(u,v)), until all vertices are processed
  - Time Complexity: $O(n^3)$ *(as implemented in the code provided in the class, not necessarily the standard answer)*
  - Code: pseudocode - Page 53, implementation - Page 58

- **Prim v.s. Dijkstra**
  - Prim's Goal: Builds a tree connecting all vertices with the smallest total edge weight (e.g., cheapest network).
  - Dijkstra's Goal: Finds the shortest path from one vertex (source) to all others.
  - Cost Difference: Prim's cost[v] is the smallest edge weight from a vertex to the tree (w(u,v)); Dijkstra's cost[v] is the shortest path from the source to a vertex (cost[u] + w(u,v)) - Page 56

Week 11 - Binary Search Tree

- **BST Basics**
  - A binary search tree (BST) is a binary tree where each node's left subtree has values less than the node, and the right subtree has values greater, with no duplicates by default.
  - BST properties (left < node < right, no duplicates) - Page 5

- **BST Representation:**
  - Represent a BST using linked nodes, where each TreeNode has an element, a left child, and a right child.
  - Node structure in TreeNode class - Page 4

- **Insertion in BST**
  - Insertion Process: Start at the root; if empty, set the new node as root; otherwise, traverse using current and parent pointers (go left if element < current, right if greater) until current is null, then insert as a child of parent.
  - Time Complexity: O(h), where h is the tree height (h is O(log n) if balanced, O(n) if unbalanced, e.g., a linked list)
  - Code: Insertion operation - Pages 6-26

- **Deletion in BST**
  - Deletion Process: Locate the node to delete and its parent; handle two cases: (1) if no left child, connect parent to the right child; (2) if a left child exists, replace with the rightmost node in the left subtree, then adjust the subtree.
  - Time Complexity: O(h), where h is the tree height (h is O(log n) if balanced, O(n) if unbalanced.
  - Code Focus: Deletion cases - Pages 42-43

Week 11 - Binary Search Tree (cont')

- **Tree Traversals:**
  - Traversal Process:
    - Inorder: Left, Node, Right
    - Postorder: Left, Right, Node
    - Preorder: Node, Left, Right
    - Breadth-First: Level by level, left to right
    - Depth-First: Branch by branch, left to right
  - Time Complexity: O(n) for all traversals.
  - Using iterator for Traversal: Use an InorderIterator to store elements in a list via inorder traversal, allowing flexible processing (e.g., uppercase printing)
  - Code: Inorder, postorder methods - Page 31, Preorder see BST.java - line 110-128. Implement iterator - Pages 37-38

- **Huffman Coding:**
  - Huffman Coding Process: Build a Huffman tree using a greedy algorithm: create leaf nodes for characters with frequencies, repeatedly combine the two smallest-weight trees into a new tree (weight = sum of children), assign 0/1 to left/right edges, and generate codes by paths to leaves.
  - Given the figure below, the Huffman coding of the word 'Mississippi' is _____. (Page 50)

Week 12 - Part 1 AVL Tree

- ***The lecture note has been updated***
  - Add page numbers
  - Page 32, ~~O(logn)~~ -> log n
  - Page 38, the XOR result of value2

- **AVL Tree Basics:**
  - AVL trees are self-balancing BSTs where the balance factor (height of right subtree minus left subtree) of each node is -1, 0, or 1, ensuring the height is approximately O(log n)

- **Balancing AVL Trees:**
  - Balancing Process: After insertion/deletion, if a node's balance factor becomes ±2, rebalance using one of four rotations: LL (right rotation at A if A and its left child are left-heavy), RR (left rotation at A if A and its right child are right-heavy), LR (left rotation at B, then right at A), RL (right rotation at B, then left at A).
  - Focus: Understand rotation process, step-by-step illustration, Page 6-13, implementation, Page 23-26

- **AVL Tree Operations:**
  - Operations Process: Extend BST operations (insert, search, delete) by updating heights and rebalancing after each operation to maintain the AVL property.
  - Time Complexity: O(log n) for search, insertion, and deletion (due to balanced height)
  - Focus: Rebalancing logic (the balancePath()) - Pages 21-22

Week 12 - Part 2 Hashing

- **Hash Table Basics:**
  - A hash table stores key-value pairs by mapping keys to indices in able using a hash function; the hash function converts a key into an integer, which is then compressed to an index within the table size (via modulo operation)

- **Collision Handling Process:**
  - Open Addressing: Find an open position using probing: linear (check key+1, key+2,...), quadratic (check key+j^2,...), or double hashing (use secondary hash h'(key))
  - Separate Chaining: Store colliding keys in a bucket (e.g., LinkedList) at the same index.

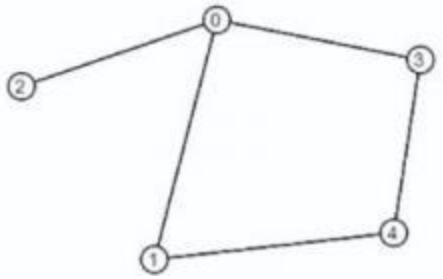
- **Load Factor and Rehashing:**
  - Load factor $\lambda = n/N$ (elements/table size); keep $\lambda < 0.5$ (open addressing) or 0.9 (separate chaining). If exceeded, double the table size and rehash entries..

- **Hash Table Implementation:**
  - Use MyMap interface for map operations (e.g., put, get), implemented by MyHashMap with, hash functions, rehashing logic, etc.
  - Code: Pages 51-55 (MyMap interface on Page 51-52, MyHashMap on Pages 53-55)

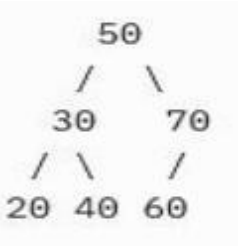
- **MyHashMap vs. MyHashSet:**
  - MyHashMap uses MyMap for key-value pairs with entrySet traversal; MyHashSet uses Collection with direct iterator() for elements.

Sample Questions

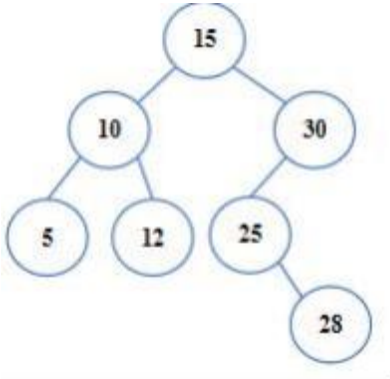
- **Week 9**
  - The time complexity of Merge Sort is _____ across all cases due to its balanced splitting and merging.
  - In Merge Sort, the merge function uses three pointers: current1 for the first subarray, current2 for the second subarray, and current3 for the temporary array. It compares list1[current1] and list2[current2]; if list1[current1] < list2[current2], it places _____ into temp[current3], then increments _____.
- **Week 10**
  - Breadth-First Search (BFS) visits vertices level by level using a _____ to process neighbors before moving to the next level.
  - Use DFS to search the figure on the right from vertex 2, the search order will be 2, _____ (use comma to seperate the elements)

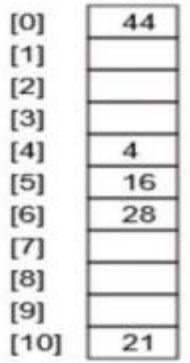


- **Week 11**
  - In a binary search tree, the value of any node in the _____ subtree is less than the node's value, and in the _____ subtree, it is greater.
  - To delete node 50 from the BST on the right, the rightmost node in its left subtree is _____, and after replacement, the left child of the element at node 50's position becomes _____.



- **Week 12**
  - Similar to what we have tested in CW2
    - The pre/post/inorder after inserting certain elements
    - Give an AVL tree, figure out the balance factor (R-L), imbalance type/rotation, etc
    - Give a hash table, find the index when adding certain values using different methods





Prepare the Final Exam

- **General Information:**
  - The final exam will be an open-booked, two-hour exam.
  - It will be conducted on Learning Mall using the Safe Exam Browser (SEB)
  - The exam includes 30 fill-in-the-blank questions (2 for each), and 2 coding questions (20 for each).
  - You are allowed to bring lecture notes, codes (in .java) and paper materials (e.g., your notes)
  - Java IDE (e.g., IntelliJ) and draft paper will be provided

- **Practice:**
  - Past exams in school Library
  - All the lab questions
  - The codes uploaded every week
  - Mock Exam using SEB (Get yourselves familiar with the SEB, announcement will be made later)

- My office hour would be adjusted during the reading week and exam weeks (at SD423)
  - Reading week and Exam week - Every Wednesday afternoon (2-5 pm)

CW3 Reminder

- **For project presentation:** *"Explain how the graph algorithm works to calculate the shortest path using **another test case** that is different from the ones required in the report"* --- In the video, run one (or more if you have) test case(s) that is/are different from the 3 test cases given in the task sheet (i.e., use different input).

- **In video recording**, make sure both you and your teammate talk, and appear on camera when talking (not necessarily at the same time though)

- **Submission:** May 18th; the cut-off date is May 25th, 5% penalty per day from 19th to 25th.

- **4 files required:** A Word for report, a Zip for all your code filles, a PPT and a Video (within 8 minutes; in mp4) for your presentation

- Double check the **Report Requirement Section** in task sheet before submission