1. Design { ~~Data Structures~~ / ~~Motivation of studying~~ / ~~Language~~

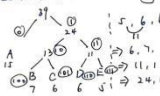
1.1 Abstraction
Huffman coding & priority queues
Information Hiding
Encapsulation
1.2 Efficiency in space & time
Static vs. dynamic data structures

1.1 How to develop good software : abstraction, information hiding, encapsulation.

Example 1 Huffman coding

A	15
B	7
C	6
D	6
E	5

⇒ priority queue



5, 6, 6, 7, 15
⇒ 6, 7, 11, 15
⇒ 11, 13, 15
⇒ 24, 15 ⇒ 39

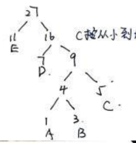
⇒ base on the generating order ⇒
A → 0. 1 bit → 15 bit (15×1)
B → 100. } 3 bits 21
C → 101. 18
D → 110. 18 (×3)
E → 111 15
↓
Total : 87

Example 2

A	1
B	3
C	5
D	7
E	11

4, 5, 7, 11
⇒ 7, 9, 11
⇒ 11, 16
⇒ 27 ABCDE order.

如何画Huffman tree ⇒ (按从小到大) ⇒ 编码 向左为 0，向右为 1.



E : 0
D : 10
A : 1100
B : 1101
C : ~~000~~ 111

Defination 1 : priority queue : data structure
— object : ~~insert x in to queue return the value of min queue~~
~~extract min queue~~
- insert (x, queue) → new queue.
- min (queue) → min
- extract_min (queue) → remove the min from queue and return this min number

Defination 2 : information hiding : only know abstraction

Defination 3 : signature / encapsulation 封装 (接口对接)

public void setMap(int x, int y)

signature 用于标识一个方法关键部分 (唯一)
eg: public int add (int a, int b) { ... }
⇒ add (int, int) 顺序重要.
public int add (String a, int b).
⇒ add (String, int)

1.2 efficiency in space & time. ← a Standard for what 数据结构 to choose
↓ memory usage
↳ 访问存储在数据结构的数据项次数.

1.3 Static and dynamic data Structure
dynamic ↓ 增大/缩小能存储的数据项适应需求
Static ↓ 创建时就固定 (数据可变，结构不能变).
★ Java 数组 从大小固定上 → 静态.
从内存分配上 → 动态 (运行时分配)
⇒ 混合类型.

2. Collection
getKey).
put (key, value).
remove (key)
containsKey(key)
size()
no duplicate key

keySet() → Set of all keys
values() → collection of all values.
entrySet() → set of all map(key, values) → map.getKey(); / map.getValue();
应用：文件 统计 <词，出现次数> → 最大频率的词

2.1 Programming with Linear collections → 元素按线性顺序排列，依次从头到尾访问支持
(Lists, Sets, Bags, Stacks, Queues, Priority Queues)

Collection<E> = Bag // most general.

① Lists (ArrayList / LinkedList) 有顺序
List<Integer> list = new ArrayList<>();
② Set 不允许重复元素 (HashSet / TreeSet). 无顺序 (Set 在 add 上 如果重复不添加) 理论上非线性集合 ppt上放这里!
Set<Integer> set = new HashSet<>();
③ Bag 允许重复元素 Multiset
Multiset<String> bag = HashMultiset.create();
→ add(value), remove(), contains(), findElement(),
size(), isEmpty(), iterator(), clear(), addAll(collection),
containsAll(collection), removeAll(collection),
④ Map (HashMap / TreeMap)
Map<String, Integer> map = new HashMap<>();
⑤ Stack 后进先出 有存 (ArrayStack / LinkedStack) (来自 List).
→ Java : push(Object item) → 任何类型对象
Object pop()
Object peek()
→ 官方方法 : push(value) 把元素放到栈顶
pop() : 将栈顶元素移除.并返回这个元素的值
peek() : 返回栈顶元素值不移除.
→ 应用: ① 处理 html / xml 检查文件
② 程序执行执行子方法.
③ 撤消功能：操作记录压入栈中 当用户撤消 从栈弹出最后一个操作
④ 表达式求值中 "(1+2)×3" 中括号入栈出栈
Stack<Integer> stack = new Stack<>();
⑥ Queue 先进先出 (有接口) 有存 (LinkedList, PriorityQueue) (见 2.6)
Queue<Integer> queue = new LinkedList<>();
⑦ Priority Queue 优先队列 (返回最小值)
PriorityQueue<Integer> pq = new PriorityQueue<>();

2.2. Programming with Hiearical collections (Trees, binary trees, general trees).
↳ (Java 中没有直接 Graph 的实现) { TreeSet<Integer> treeSet = new TreeSet<>(); / TreeMap<String, Integer> treeMap = new TreeMap<>();

2.3 Java API
java.util → collection / utility
java.io → input/output.
javax.swing / java.awt → GUI.

如何 import : import package.class0 # 特定类
import package.* # 所有类

• Java inteface = Abstract Data Type → ArrayList / LinkedList ←extend List<E> ←extend Collection<E>.
接口可继承并拥有父类所有方法
example : ~~public interface Set { ... public void add(item);~~
★ List<Task> tasks = new ArrayList<Task>();
编译时，编译器知道 task 是一个 List (接口类型)，只能调用 List 接口中定义的方法，而具体实现只能调用这个接口定义的方法.

```
public interface List<E> extends Collection<E>{
    public boolean add (E o){
        return ....
    }
    public E get(int index){
    }
}
```

↳ 任意数据类型 ↓ 但是一旦输入(比如字符串，即就不能改，在同一个类中 都是字符串).

How to declare? private Set<Person> friends = new HashSet<Person>(); (类)
private List<Shape> drawing = new ArrayList<Shape>();

Comparison : Array and List (List better, more function, no limits)
Student[] students = new Student[100];
List<Student> students = new ArrayList<Student>();
Student s = new Student("Lindy King", 12);
students.add(s);

2.4 Iterator(接口)

```
for (Task task : taskList) {
    textArea.append(task+"\n");
}
```

在编译后转换为 ⇒ taskList.iterator();

```
Iterator<Task> iter = taskList.iterator();
while (iter.hasNext()) {
    Task task = iter.next();
    textArea.append(task + "\n");
```

这个 public interface Iterator<E> {
    public boolean hasNext();
    public E next();
    public void remove()   必须 → throw 异常即可.
}

对比

construct a iterator

Iterable<T>
(public Iterator<T> iterator();
→ taskList 是 Iterable 我们需要构建一个 Iterator 迭代, 内部只包含这个方法, 目的只是创建实例)

2.5 Transform infix Expression (中缀表达式 2+3*5) into Postfix Expression (后缀表达式 235*+).

E1: char / Stack for operator

方式:
a → a
+ → a   +
b → ab   +
* → ab   *+
c → abc   *+
→ abc*   +   (遵从后进先出!)
→ abc*+

E2:
a   a
-    a   -
b   ab   -
+   ab-   +
     ab-   +
c   ab-c   +
     ab-c+

特例
a   a
^   a   ^
b   ab   ^
^   ab   ^^
c   abc   ^^
     abc^   ^
     abc^^

* 次符为运算符 ① 栈为空 ⇒ 入栈 ② 出栈所有符号直到栈空或者栈顶优先级低于当前优先级再让当前入栈 (幂 > "×,÷" > "+ -")

~~①栈运算符优先级 ≤ 当前运算符优先级 ⇒ 入栈~~

括号 > 所有运算符

~~②其他 ⇒ 先将栈顶弹出放到后缀表达式, 再把当前运算符入栈~~

对于幂运算由于它右结合遇到相同优先级直接入栈而不是弹出

* 左括号: 入栈

右括号: 从栈顶弹出运算符直到左括号为止; 二者都不保留在后缀表达式

① (a+n)*(b-8*m)

(		(
a	a	(
+	a	+(
n	an	+(
)	an+	
*	an+	*
(	an+	(*
b	an+b	(*
-	an+b	-(*
8	an+b8	-(*
*	an+b8	*-(*
m	an+b8m	*-(*
)	an+b8m*-	*
	an+b8m*-*	

② b/v^7

b	b	
/	b	/
v	bv	/
^	bv	^/
7	bv7	^/
	bv7^/	

③ {3+[d-7*(g+5)]/w}

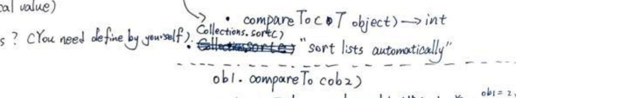
3d7	*-[+{
3d7g5	+(*-[+{
")" 3d7g5+	*-[+{
"]" 3d7g5+*-	/+{
3d7g5+*-w	/+{
"}" 3d7g5+*-w/+	

④ [(4+b] -2)

4b	+(
] 4b+(2	-
4b+(2-	⇒ 4b+2-

→ 左弹出自动移除!

a=2, b=4, c=3   "24+3/"

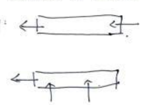


* 为什么后缀式子正确

a/b ⇒ ab/ 入栈 出栈后 a/b 填入计算

(a+b)/c ⇒ ab+c/

2.6 Two varities of Queues



add or remove with a given priority

offer(value) ⇒ boolean

poll() ⇒ value "remove and return value at front of queue or the queue is empty"

peek() ⇒ value "return head of queue or queue is empty"

remove() like poll, but throw exception when queue empty

element() like peek()

3. Iterators and Comparators

3.1 Why Iterators (Program can not get inside the Collection object) * Each Collection needs an Iterator class

Iterator<Face> ← List<Face>
Data[i]   Data,   see 2.4   * Iterators 是一个接口.

hasNext next
boolean value

* public class NumCreator (implements) Iterator<Integer>{ ... #
 ↳ 这里为什么不用 extend 原因在于 { implement + 接口 (实现) / extend + 类 (继承) }

}

* Iterable和Iterator<E> 都是接口, 都不是类

3.2 Sorting a collection in Natural order

— Strings (alphabetically)
— Integer / Double (ordered by numerical value)
— Complex questions: faces in crowds? (You need define by yourself)

a class implement Comparable<T> (接口)
— Objects from that class have natural ordering
 • compareTo(T object) → int
 • Collections.sort() "sort lists automatically"

ob1.compareTo(ob2)

⇒ { return -1 when ob1 排在 ob2 前 (ob1=2, ob2=3)
return 0 when ob1, ob2 相等 (ob1=2, ob2=2)
return 1 when ob1 排在 ob2 后 (ob1=3, ob2=2)

3.3 Comparator V.S. Comparable

Collections.sort(todoList) — Collection → "sort by natural order" * ~~todoList~~ values in todoList must be Comparable.

Collections.sort(~~todoList~~ data / todoList, object of Comparable) "sort according specific order"

Collection → Comparator: an object that can compare other objects.

• Comparator<T> 接口

public class Agent implements Comparable<~~Per~~ Agent>{
    public int compareTo (Agent other){
        if - - -
        else if - - -
        else - -
    }
}

(一样 / Vs / 不一样)

public class ~~Agent~~ AgentComparator implement Comparator<Agent>{
    public int compare (Agent a1, Agent a2){
        return (a1.length - a2.length)
    }

Collections.sort (crowd, new AgentComparator())

* A comparable class can implement multiple comparators(✓)
Java 支持多接口实现，因此一个类同时实现"Comparable"和多个"Comparator"接口

值得提醒使用下列:

public class ArrayList<E> extends AbstractList<E>{
    public ArrayList(){
        data = (E[]) new Object [Initial capacity].
    }

不能用 type variables as array constructors
只能用 Object[] / E[] 否则报错

抽象类特点：1. 不能被实例化
2. 可以包含抽象方法(没有方法体)和具体方法(有方法体)
3. 可以有成员变量和构造方法.
4. 需要被子类继承并实现其抽象方法

扩容：在 ArrayList 放不下时：

private void ensureCapacity(){
    if(count < data.length){
        return.
    }
    E[] newArray = (E[])(new Object[data.length * 2]).
}

(data.length * 2 ↔ data.length + Initial capacity)

4 Implement Collections

4.1 Exception

try { - - -
}
catch (<ExceptionType> e1){
    - - - -
}

(ExceptionType: 特别各种异常 → ArrayIndexOutOfBoundsException, ArithmeticException, NullPointerException)
- RuntimeExceptions → 不需要处理
- Others → 需要处理
- provides a local exit when goes wrong

if catch an exception, act! 当异常被处理后，继续执行代码!

if (A == B){
    throw new ExceptionType ();
}

4.2 Interface and Class (implements List<E>)

List<E> ↓ define constructors, methods.
ArrayList<E>

* List<E> ≥ extends Collection<E>. 都是接口 interface ⇒

由 List 实现 ↓ ArrayList<E>

重新(ArrayList定义方式)

interface	List<E>
↓ implements	
Abstract Class	AbstractList<E>
↓ extends	
Class	ArrayList<E>

4.3 Multiple Interators (多重迭代器)

↳ 允许在同一个集合同时进行遍历，
- 可以访问 ArrayList 私有 fields
- 因对 ArrayList 修改会发出 exception.
- 支持 remove()
- 每个 interator 维护当前位置.

4.4 Cost in ArrayList (Complexcity

get: O(1)
set: O(1).
remove: O(n)
add: 尾部添加 (平均 O(1), 溢出 → O(n))
    中间插入 O(n)

以下为问题解答：
1. remove() 没有必要一定实现 Interator 接口，可以抛出异常.
2. type parameter 保证 ArrayList 在编译时类型检查，确保指定类型.
3. ArrayList.remove(index) 索引元素
    ↑ Object o 列表第一次出现指定元素

5 Analyzing Costs

(We need both Algorithm and Program)
Algorithm → Big-O Notation
Program → System.currentTimeMills()

$O(\log n) < O(n^{\frac{1}{2}}) < O(n) < O(n\log n) < O(n^2) < O(2^n) < O(n!)$

focus on Loop.

Reading a line from user is more expensive 和 from file.
(from user → 输入设备，用户响应时间; from file → I/O 操作，文件读取速度)

6. Recursion

6.1 related to recursive problem: binary tree traversal

6.2 Iteration v.s. Recursion ⟶ slow speed and elegant (轻松去设计)

↓

efficient but not easy to design.

Every recursive function can be rewritten into iterative function.(包括NP-hard也行)

7. Linked Structures

7.1 Queue about Collection Inplemetiting

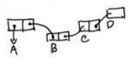
add O(1) remove O(1)

7.2 How to insert fast in array? ⇒ Linked Structures

We know to ensure items sorted to use binary search.

But if arrays stored in contiguous chunks of memory (连续的内存块), 插入元素很慢, 需要移动大量元素来保持有序性, 因此我们需要

为每个元素都存储在独立内存位置, 避免插入导致整体元素移动.



insert/remove by changing links.

Example: A ⟶ C ⟶ H ⟶ M ⟶ Y (inserting G between C and H)

现插入 G 无法找到"C"点, 如何做?

① 将 C → P 创建作为新链接.

② 不管"C" 创建 G → H

7.3 Memory allocation 内存分配

reference / pointer 引用/指针

memory from heap 必须回收 (自动 ⟵ garbage collector).

using new create object will allocate heap memory

copy address ≠ copy chunck of memory

stack ⟶ high address

↕ ⟶ free memory

heap.

static data

code ⟶ low address

```
public class LinkedNode<E>{..
    - get()
      set()
      setNext()
}
```

7.4 Linked Node and Linkedlist (extend ⟶ Abstractlist)

LinkedNode<S> color = new L...Node<S>(red, null) ⇒ red.

color.next = new LinkedNode<S>(blue, null) ⇒ red → blue

color = new LinkedNode<S>(green, color) ⇒ green → red → blue.

color.next.next = new LinkedNode<String>(black, color.next.next) ⇒ green → red → black → blue.

black → blue

printList: for (rest != null, rest = rest.next)
print (rest).

回 7-1 PPT 第6页.

在实现之前编写测试用例进行黑箱测试 (Test-Drieven Development 测试驱动开发).

7.5 Linked Collection Cost.

Linked list: get/set: O(n) 从头找.
insert: 指定节点 O(1). / 其余 O(n).
remove: 指定节点 O(1) / 其余 O(n).

Ordered Linked Set: get/remove/insert O(n).

Stack / Queue with LinkList:

8. Sorting Algorithm.

8.1 ArraySet Algorithm Comparison. (ArraySet ⟶ 顺序不重要, 不允许重复.)

unordered	ordered	
O(n)	O(logn)	contains( ).
O(n)	O(n)	add( )
O(n)	O(n)	remove( )

↓ 需要将位置所有向前/向后移动导致

8.2 Different Sorting Algorithms.

	Selection	Bubble	Insertion	Quick Sort	Merge Sort
worst	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n\log n)$
Avg	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n\log n)$	$O(n\log n)$
space usage	原状	原状	原状	原状	非原状

Insertion ⟵
9 6 4 8 1 5 3.
6 9 4 8 1 5 3
4 6 9 8 1 5 3
- - - - - -

Explanation (Quick Sort $O(n^2)$):

在每次分割总选到最小值/最大值

要进行 O(n) 操作将所有移动到一边

才确定一个位置, 有 n 个

$\Rightarrow O(n) \cdot n = O(n^2)$

8.3 Reflection Upon QuickSort.

pivot → 枢轴

multi-pivot quicksort advantages:

— 减少递归深度.

— 既避平均时间复杂度

— 减少交换次数.

8.4 All 整理

	ArrayList	Stack	Queue	Linklist (从头找)	ArraySet	SortedArraySet
add	O(n)	O(1)	O(1)	O(n)	O(n)	O(n)
remove	O(n).	O(1)	O(1)	O(n).	O(n).	O(n)
add n items						$O(n^2)$
Sort All						$O(n\log n)$
contain						$O(\log n)$.

9 Tree.
- Binary Tree.
- General Tree
- AVL Tree.

9.1 疑问问答:

树是一种分级数据结构, 二叉树的兄弟节点没有顺序要求.

↓

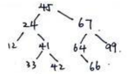
应用
- 文件系统.
- 数据库索引
- 表达式解析

递归调用之间关系可以表示为递归树

9.1 Binary Search Tree.

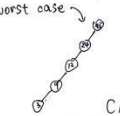
— A binary tree

— left child < parent < right chile.



O(logn) in insert, lookup, remove. (每次减半)

O(n) : worst case



(AVL : Adelson-Velskii and Landis).

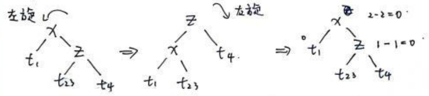
9.2 Balanced Search Tree. :使用左旋/右旋,保持树的高度较低且均匀,避免树退化成有序链表 O(n) 情况.

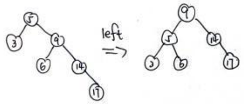
在最坏情况 insert, lookup, remove 都是 O(logn) → 保证所有节点在 (-1), (0), (1)

↳ 平衡因子: |左子树高度 − 右子树高度| ≤ 1

— 平衡搜索树是一种二叉搜索树.

左旋

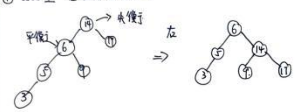




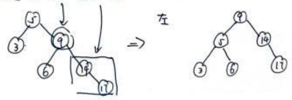
AVL.

	Avg	Worst
Search	O(logn)	
Insert		
Delet		
Space	O(n)	O(n)

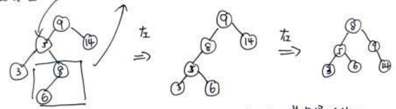
① LL型 左孩子左子树.



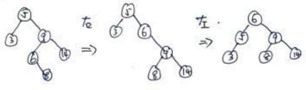
③ RR型 右孩子右子树. 不指偏向



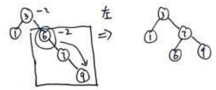
② LR型 左孩子 右子树.

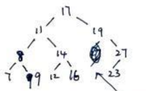


④ RL型



插入节点后 如果有多个父节点失衡 则调整离插入节点最近的.

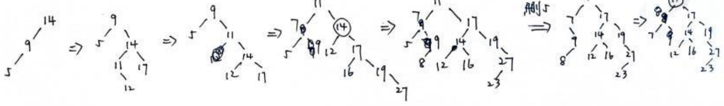




删除从下往上依次调整

检查!

平衡二叉树构建: 14, 9, 5, 17, 11, 12, 7, 19, 16, 27, 8, 23



10. Hash Table:

O(1) in avg insert, lookup, remove (通过哈希函数计算键的索引) [没有冲突下].

使用T来存储键值对, 容量为N { 太大浪费空间 / 太小意味冲突

H(string key) = (int) 0~N−1, store key and info at T[H(key)]

H() must return same for one key.

eg: 哈希表.

Hash Code	name/number
0	/
1	/
2	Corea/46532, Davis/43534

常见哈希函数: ① 字符全小写 用ASCII码相加取模运算得到 →

avoids collisions / must be O(1) / 把key尽量均匀分布在列表中

② mid-square method: k = 1234 计算 $k^2 = 1522756$. 取中间4位 2275 作为哈希值

碰撞: 12, 22, 32, H(n) = n % 10

12 (22) 碰撞 解决方法: 1. 线性探测 linear probing (hash(k) + i) % N (但容易导致主集群现象) (i: 探测次数)

2. 二次探测 Quadratic probing (hash(k) + $c_1 i + c_2 i^2$) % N (次集群现象).

3. 双重哈希 [hash1(k) + i * hash2(k)] % N

(若哈希表满 ① 扩容 ② 删去小的)

11 Graph.

H is spanning subgraph (spans all vertice G)

H is subgraph G (vetice set is subset of G, edge set is subset of G)

cycle: 

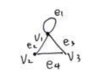
disconnected components

multiple edges, loop > 排除 → simplegraph

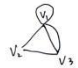
s → t walk → trail (所有边不同) → path (所有点、边不同)

closed ($V_s = V_t$)

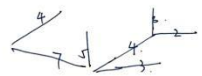
Length = |E|



	V_1	V_2	V_3
e_1	2	0	0
e_2	1	1	0
e_3	1	0	1
e_4	0	1	1



	V_1	V_2	V_3
V_1	1	1	2
V_2	1	0	1
V_3	2	1	0



---- 插入/移除新元素到最小堆的算法 ***O(1)~ O(logn)***