UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

NGÔN NGỮ LẬP TRÌNH JAVA

Chương 3
CÁC LỚP TIỆN ÍCH TRONG JAVA (P3)
Collection

GVGD: ThS. Lê Thanh Trọng

NỘI DUNG

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

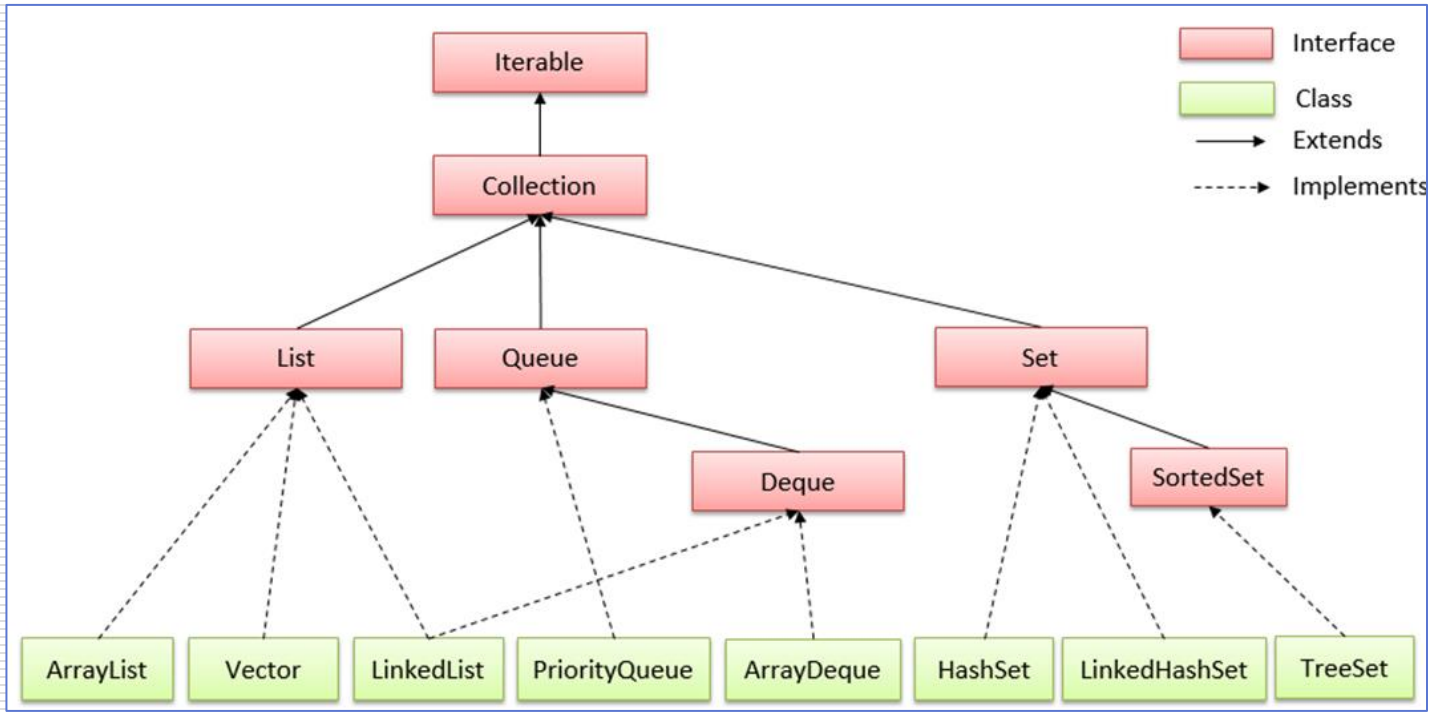
1. Giới thiệu Collection
2. List
   i. ArrayList
   ii. LinkedList
   iii. Vector
3. Set
   i. HashSet
   ii. LinkedHashSet
   iii. TreeSet
4. Queue
   i. PriorityQueue
   ii. ArrayDeque
5. Map Interface
   i. HashMap
   ii. LinkedHashMap
   iii. TreeMap
   iv. HashTable

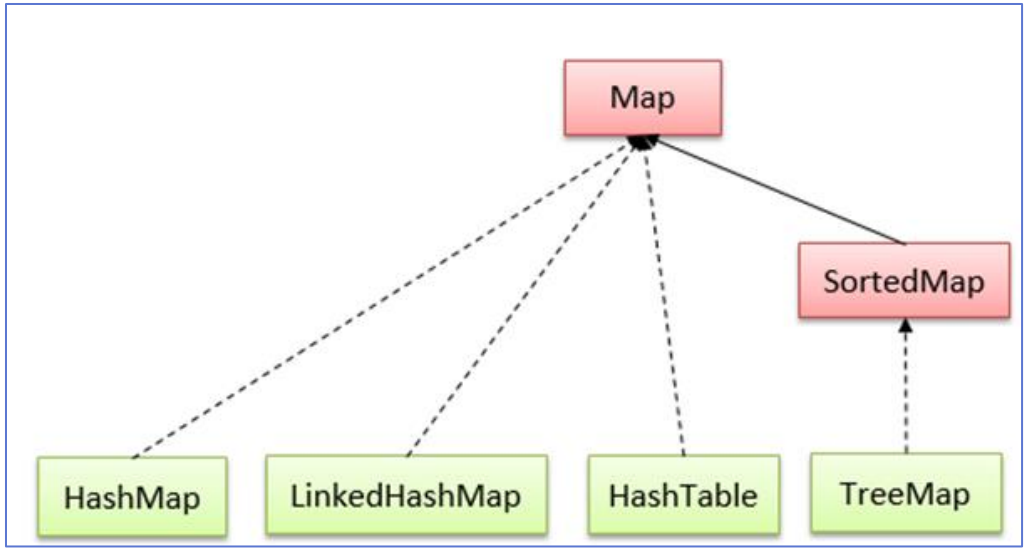
NỘI DUNG

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

1. **Giới thiệu Collection**
2. List
   i. ArrayList
   ii. LinkedList
   iii. Vector
3. Set
   i. HashSet
   ii. LinkedHashSet
   iii. TreeSet
4. Queue
   i. PriorityQueue
   ii. ArrayDeque
5. Map Interface
   i. HashMap
   ii. LinkedHashMap
   iii. TreeMap
   iv. HashTable

Tổ chức Collection trong Java





Tổ chức Collection trong Java

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

- 2 loại cơ bản của Java Collections:
  - **Collection** Interface
    - **List**: ArrayList, LinkedList, Vector
    - **Set**: HashSet, LinkedHashSet, TreeSet
    - **Queue**: PriorityQueue, ArrayDeque
  - **Map** Interface: TreeMap, HashMap, LinkedHashMap, HashTable
- Collection Interface:

Các lớp thuộc Collection

- **Iterable:** Chứa dữ liệu thành viên Iterator interface
- **Iterator**: Cung cấp phương tiện để lặp đi lặp lại các thành phần từ đầu đến cuối của một collection
- **Set:**
  - Không thể chứa 2 giá trị trùng lặp
  - Thường không có thứ tự
- **List**
  - Có thứ tự (đôi khi còn được gọi là một chuỗi)
  - Có thể chứa các phần tử trùng lặp
  - Được truy cập các phần tử hoặc chèn vào bằng chỉ số (index)
- **Queue** (hàng đợi)
  - Chứa và tổ chức nhiều phần tử trước khi xử lý
  - Cung cấp các thao tác bổ sung như chèn, lấy ra và kiểm tra
  - Queue có thể được sử dụng như là FIFO (first-in, first-out - vào trước, ra trước)

Các lớp thuộc Collection

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

- **Deque**
  - Kết hợp các tính năng của Stack và Queue
  - Có thể được sử dụng như là FIFO (first-in, first-out - vào trước, ra trước) và LIFO (last-in, first-out - vào sau, ra trước)
  - Tất cả các phần tử mới có thể được chèn vào, lấy ra và lấy ra ở cả hai đầu
- **Map**
  - Ánh xạ mỗi key tương trùng lặúng với một giá trị
  - Không thể chứa giá trị p
  - Mỗi key có thể ánh xạ đến nhiều nhất một giá trị
- **SortedSet**: Chứa các phần tử theo thứ tự tăng dần.
- **SortedMap**: Chứa các phần tử được sắp xếp theo thứ tự tăng dần theo key (dùng cho danh sách được sắp xếp "tự nhiên" như từ điển, danh bạ điện thoại,…)

Iterator interface

- public boolean **hasNext**(): Trả về true nếu iterator còn phần tử kế tiếp phần tử đang duyệt
- public object **next**(): Trả về phần tử hiện tại và di chuyển con trỏ trỏ tới phần tử tiếp theo
- public void **remove**(): Loại bỏ phần tử cuối được trả về bởi Iterator

Ví dụ Iterator

```
ArrayList<String> names = new ArrayList<>();

//Thêm danh sách các tên
names.add("Alice");
names.add("Bob");
names.add("Charlie");
names.add("David");
//Khai báo một biến kiểu Iterator (String) để duyệt các phần tử trong ArrayList
Iterator<String> iterator = names.iterator();
// Duyệt qua các phần tử
System.out.println("Các phần tử của ArrayList:");
while (iterator.hasNext()) {
    String name = iterator.next();
    System.out.println(name);
}
//Loại bỏ phần tử
iterator = names.iterator(); //Khởi động iterator
while (iterator.hasNext()) {
    String name = iterator.next();
    if (name.equals("Charlie")) {
        iterator.remove(); //Loại bỏ phần tử "Charlie"
    }
}
//Duyệt sau khi bỏ phần tử
System.out.println("Các phần tử còn lại: ");
for (String name : names) {
    System.out.println(name);
}
```

```
Các phần tử của ArrayList:
Alice
Bob
Charlie
David
Các phần tử còn lại:
Alice
Bob
David
```

Interface Collection

- public boolean **add**(Object element): Chèn một phần tử
- public boolean **addAll**(Collection c) : Chèn các phần tử collection được chỉ định vào collection gọi phương thức này
- public boolean **remove**(Object element): Xóa phần tử từ collection
- public boolean **removeAll**(Collection c): Xóa tất cả các phần tử của collection được chỉ định từ collection gọi phương thức này
- public boolean **retainAll**(Collection c): Xóa tất cả các thành phần từ collection gọi phương thức này ngoại trừ collection được chỉ định
- public int **size**(): Trả về tổng số số phần tử trong collection

Interface Collection

- public void **clear**(): Loại bỏ tổng số của phần tử khỏi collection.
- public boolean **contains**(Object element): Được sử dụng để tìm kiếm phần tử
- public boolean **containsAll**(Collection c): Tìm kiếm collection được chỉ định trong collection
- public Iterator **iterator**(): Trả về một iterator
- public Object[] **toArray**(): Chuyển đổi collection thành mảng (array)
- public boolean **isEmpty**(): Kiểm tra nếu collection trống
- public boolean **equals**(Object element): So sánh 2 collection
- public int **hashCode**(): Trả về số hashcode của collection

NỘI DUNG

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

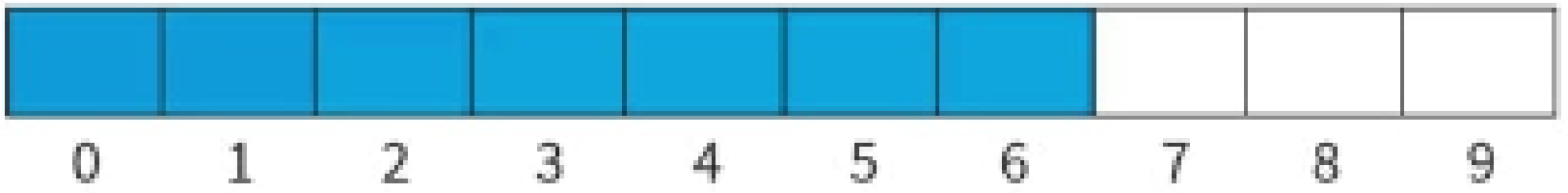
1. Giới thiệu Collection
2. List
   i. ArrayList
   ii. LinkedList
   iii. Vector
3. Set
   i. HashSet
   ii. LinkedHashSet
   iii. TreeSet
4. Queue
   i. PriorityQueue
   ii. ArrayDeque
5. Map Interface
   i. HashMap
   ii. LinkedHashMap
   iii. TreeMap
   iv. HashTable

List

- interface trong gói **java.util.*;**
- Quản lý danh sách các phần tử có thứ tự
- Cho phép chứa các phần tử trùng lặp
- Cho phép truy cập ngẫu nhiên đến các phần tử (thông qua index)
- Cung cấp các phương thức để thêm, xóa, truy cập và duyệt qua các phần tử trong danh sách
- Có thể thay đổi kích thước khi thực thi

Arraylist

- Từ JDK 1.2
- Mảng động, lưu các phần tử cùng kiểu
- Hỗ trợ truy cập ngẫu nhiên (index)
- Có khả năng tự động mở rộng kích thước
- Giúp việc thêm hoặc xóa phần tử khỏi danh sách trở nên linh hoạt



Arraylist

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

Ưu điểm:

- Dễ dàng thực hiện các thao tác như duyệt danh sách, tìm kiếm phần tử
- Có thể được sử dụng để chứa các đối tượng của các class khác nhau

Nhược điểm:

- Khi dung lượng của ArrayList vượt quá giới hạn khi đó hiệu suất của chương trình có thể bị ảnh hưởng
- Thêm phần tử vào cuối danh sách nhanh hơn so với thêm vào giữa danh sách

Sử dụng Arraylist

- import java.util.ArrayList;
- Khai báo
  public class **ArrayList<E> extends** AbstractList<E> **implements** List<E>, RandomAccess, Cloneable, Serializable
- **Constructors:**
  - ArrayList( )
  - ArrayList(Collection c): khởi tạo với các phần tử của c
  - ArrayList(int capacity): Khởi tạo với kích thước ban đầu

Các phương thức Arraylist

- void **add**(int index, Object element)
- boolean **add**(Object o): thêm vào cuối
- boolean **addAll**(Collection c)
- boolean **addAll**(int index, Collection c)
- void **clear**(): xóa toàn bộ các phần tử
- Object **clone**(): Tạo ra bản sao
- boolean **contains**(Object o)
- void **ensureCapacity**(int minCapacity)
- Object **get**(int index)
- int **indexOf**(Object o)

Các phương thức Arraylist

- int **lastIndexOf**(Object o)
- Object **remove**(int index)
- void **removeRange**(int fromIndex, int toIndex)
- Object **set**(int index, Object element)
- int **size**()
- Object[] **toArray**()
- Object[] **toArray**(Object[] a)
- void **trimToSize**()

Ví dụ Arraylist

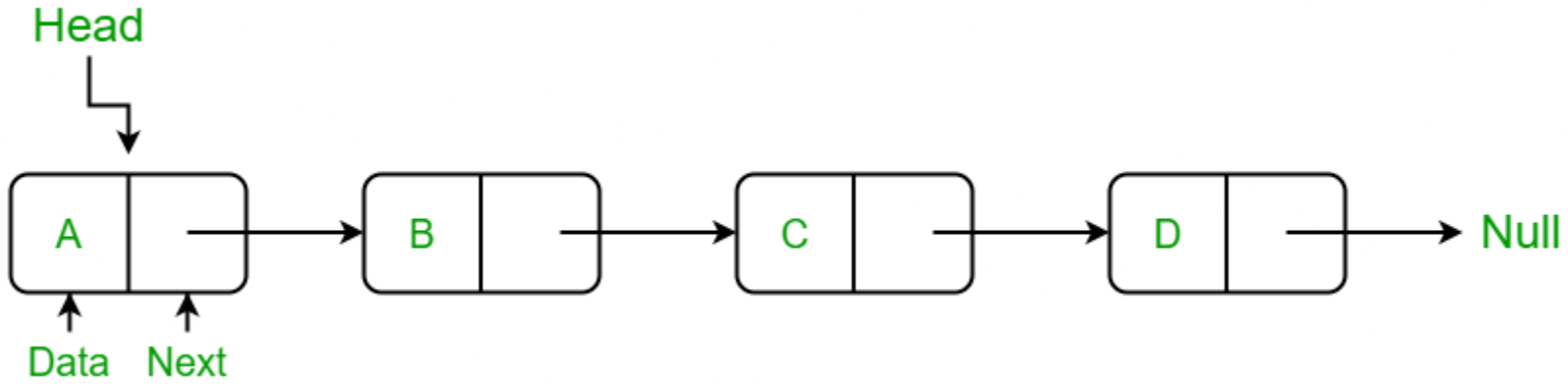
UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

```
public class ArrayListTest3 {
    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        System.out.println("Kích thước ban đầu: " + al.size());
        //Thêm các phần tử
        al.add("H");
        al.add("e");
        al.add("l");
        al.add("l");
        al.add("o");
        al.add(5, "!");
        System.out.println("Kích thước sau khi thêm: " +
        al.size());
        //Xuat các phần tử
        System.out.println("DS gồm: " + al);
        //Xóa phần tử
        al.remove("l");
        al.remove(2);
        System.out.println("Kích thước sau khi xóa: " + al.size());
        System.out.println("DS gồm: " + al);
    }
}
```

```
Kích thước ban đầu: 0
Kích thước sau khi thêm: 6
DS gồm: [H, e, l, l, o, !]
Kích thước sau khi xóa: 4
DS gồm: [H, e, o, !]
```

LinkedList

- ***import java.util.LinkedList;***
- Các phần tử được liên kết với nhau thông qua các địa chỉ bộ nhớ
- Không cần phải cấp phát toàn bộ một khối bộ nhớ liên tục
- Các phần tử có thể được chèn hoặc xóa bất kỳ lúc nào



LinkedList

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

- **Ưu điểm:**
  - Thêm hoặc xóa phần tử vào LinkedList nhanh chóng vì chúng chỉ tác động đến các phần tử bên cạnh (trường hợp thêm và xóa ở giữa danh sách)
  - Có thể lưu trữ số lượng phần tử chưa biết trước
  - Dễ thực hiện các thao tác như chèn hoặc xóa phần tử khỏi danh sách
- **Nhược điểm:**
  - Truy cập ngẫu nhiên đến các phần tử trong LinkedList chậm so với ArrayList
  - Do không được cấp phát liền mạch, LinkedList có thể dẫn đến việc lãng phí bộ nhớ và gây ra hiệu suất kém hơn nếu không được sử dụng đúng cách

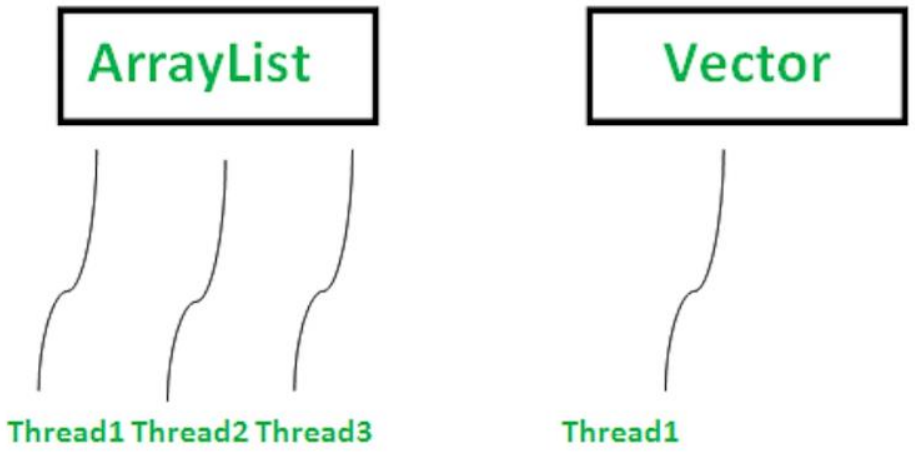
Ví dụ LinkedList

```
LinkedList<String> linkedList = new LinkedList<>();
// Thêm các phần tử vào LinkedList
linkedList.add("Apple");
linkedList.add("Banana");
linkedList.add("Orange");
linkedList.add("Grapes");
// Duyệt qua các phần tử bằng vòng lặp for
System.out.println("Duyệt qua các phần tử bằng vòng lặp for:");
for (int i = 0; i < linkedList.size(); i++) {
    System.out.println(linkedList.get(i));
}
// Duyệt qua các phần tử bằng Iterator và xóa nếu là "Banana"
Iterator<String> iterator = linkedList.iterator();
while (iterator.hasNext()) {
    if(iterator.next()=="Banana")
    iterator.remove();
}
//Duyệt sau khi xóa
System.out.println("Sau khi xoa Banana:");
for (int i = 0; i < linkedList.size(); i++) {
    System.out.println(linkedList.get(i));
}
```

```
Duyệt qua các phần tử bằng vòng lặp for:
Apple
Banana
Orange
Grapes
Sau khi xoa Banana:
Apple
Orange
Grapes
```

Vector

- ***import java.util.Vector;***
- impements giao diện List và duy trì thứ tự chèn của các phần tử
- Khác nhau giữa Vector và ArrayList
  - Vector là **synchronized**
  - Vector tăng 100% nghĩa là tăng gấp đôi kích thước hiện tại nếu số phần tử vượt quá khả năng chứa của nó (ArrayList chỉ 50%)
  - Duyệt dung Vector sử dụng Enumeration và Iterator (ArrayList dung Iterator)



Vector

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

Ưu điểm

- Cung cấp phương thức để tính toán độ lớn và hướng của một đối tượng trong không gian
- Có thể tùy biến, không gò bó về kích thước và dẫn tới tiết kiệm bộ nhớ
- Có thể được sử dụng để biểu diễn các phép toán đại số phức tạp

Nhược điểm

- Đồng bộ hóa nên có thể tốn chi phí và làm giảm hiệu suất (như so với các class không đồng bộ như ArrayList)
- Nếu chúng ta không cần đồng bộ hóa, nên sử dụng ArrayList
- Không thể kiểm soát cách đồng bộ hóa xảy ra trên Vector

Các phương thức trong Vector

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

void **addElement**(Object obj): Thêm phần tử vào cuối của vector

void **add**(int index, Object element): Chèn phần tử vào vị trí index trong vector

boolean **addAll**(Collection<? extends E> c): Nối tất cả các phần tử trong một tập được chỉ định vào cuối của vector

int **capacity**(): Trả về dung lượng hiện tại của vector

void **clear**(): Xóa tất cả các phần tử khỏi vector

boolean **contains**(Object o): Trả về true nếu vector chứa phần tử được chỉ định

E **elementAt**(int index): Trả về phần tử ở chỉ mục được chỉ định trong vector

Enumeration<E> **elements**(): Trả về một liệt kê của các phần tử trong vector

E **firstElement**(): Trả về thành phần đầu tiên (phần tử tại chỉ mục 0) của vector

E **get**(int index): Trả về phần tử ở vị trí được chỉ định trong vector

Các phương thức trong Vector

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

- int **indexOf**(Object o): Trả về chỉ mục của sự xuất hiện đầu tiên của phần tử được chỉ định trong vector, hoặc -1 nếu vector không chứa phần tử
- boolean **isEmpty**(): Trả về true nếu vector không chứa phần tử nào
- E **lastElement**(): Trả về thành phần cuối cùng của vector
- boolean **remove**(int index): Xóa phần tử ở vị trí được chỉ định trong vector
- boolean **removeElement**(Object obj): Xóa sự xuất hiện đầu tiên của phần tử được chỉ định khỏi vector
- void **removeAllElements**(): Xóa tất cả các thành phần khỏi vector và đặt kích thước của nó thành không
- void **removeElementAt**(int index): Xóa thành phần tại chỉ mục được chỉ định
- E **set**(int index, Object element): Thay thế phần tử ở vị trí được chỉ định trong vector bằng phần tử được chỉ định
- int **size**(): Trả về số lượng thành phần trong vector
- <T> T[] **toArray**(): Trả về một mảng chứa tất cả các phần tử trong vector
- void **trimToSize**(): Cắt dung lượng của vector để bằng kích thước hiện tại của vector

Ví dụ Vector

```
Vector<String> vector = new Vector<>();
// Thêm các phần tử
vector.add("Apple");
vector.add("Banana");
vector.add("Orange");
// Truy cập thông qua chỉ mục
System.out.println("Phần tử index 0: " + vector.get(0));
System.out.println("Phần tử index 1: " + vector.get(1));
System.out.println("Phần tử index 2: " + vector.get(2));
// Remove phần tử
vector.remove(2); // Removes "Orange"
// Duyệt các phần tử
System.out.println("Vector sau khi xóa:");
for (String element : vector) {
    System.out.println(element);
}
```

```
Phần tử index 0: Apple
Phần tử index 1: Banana
Phần tử index 2: Orange
Vector sau khi xóa:
Apple
Banana
```

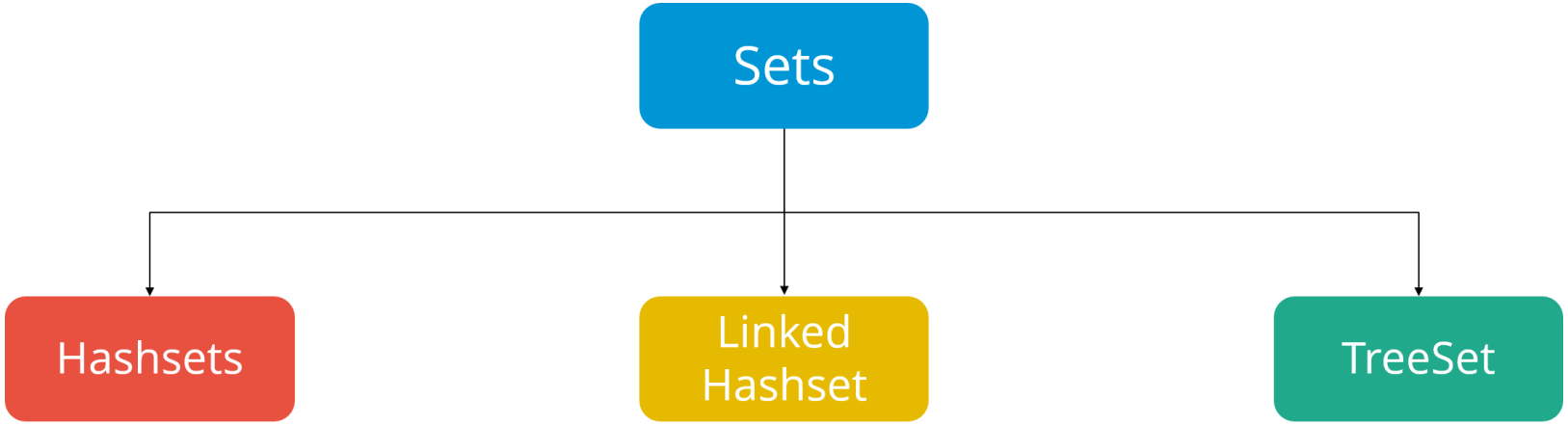
NỘI DUNG

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

1. Giới thiệu Collection
2. List
   i. ArrayList
   ii. LinkedList
   iii. Vector
3. **Set**
   i. **HashSet**
   ii. **LinkedHashSet**
   iii. **TreeSet**
4. Queue
   i. PriorityQueue
   ii. ArrayDeque
5. Map Interface
   i. HashMap
   ii. LinkedHashMap
   iii. TreeMap
   iv. HashTable

Set

- Lưu trữ các phần tử không trùng lặp
- Không có thứ tự cụ thể
- Khi thêm một phần tử vào Set, nếu phần tử ấy đã tồn tại trong Set rồi thì nó sẽ không được thêm vào
- Không cho phép truy cập ngẫu nhiên đến các phần tử



HashSet

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

- **Ưu điểm:**
  - Tốc độ truy xuất nhanh (sử dụng bảng băm để lưu trữ phần tử)
  - Hữu ích khi cần lưu trữ một danh sách các phần tử duy nhất
- **Nhược điểm:**
  - Không thể duy trì thứ tự của các phần tử trong danh sách
  - Không được đồng bộ hóa tự động (sử dụng ConcurrentHashMap)
  - Sử dụng một vòng lặp hoặc phương thức ***contains()*** để kiểm tra sự tồn tại của một phần tử

HashSet

```
HashSet<String> hashSet = new HashSet<>();
// Thêm phần tử vào HashSet
hashSet.add("Apple");
hashSet.add("Banana");
hashSet.add("Orange");
hashSet.add("Grapes");
// In ra tập hợp các phần tử trong HashSet
System.out.println("Tập hợp ban đầu: " + hashSet);
// Thử thêm một phần tử trùng lặp
hashSet.add("Apple");
// In ra tập hợp sau khi thêm phần tử trùng lặp
System.out.println("Tập hợp sau khi thêm phần tử trùng lặp: " + hashSet);
// Kiểm tra xem một phần tử có tồn tại trong HashSet hay không
boolean containsBanana = hashSet.contains("Banana");
System.out.println("Có phần tử 'Banana' trong tập hợp không? " + containsBanana);
// Xóa một phần tử khỏi HashSet
hashSet.remove("Orange");
// In ra tập hợp sau khi xóa
System.out.println("Tập hợp sau khi xóa phần tử 'Orange': " + hashSet);
// Lấy kích thước của HashSet
int size = hashSet.size();
System.out.println("Kích thước của tập hợp: " + size);
// Duyệt và in ra các phần tử trong HashSet
System.out.println("Duyệt các phần tử trong HashSet:");
for (String element : hashSet) {
    System.out.println(element);
}
```

```
Tập hợp ban đầu: [Apple, Grapes, Orange, Banana]
Tập hợp sau khi thêm phần tử trùng lặp: [Apple, Grapes, Orange, Banana]
Có phần tử 'Banana' trong tập hợp không? true
Tập hợp sau khi xóa phần tử 'Orange': [Apple, Grapes, Banana]
Kích thước của tập hợp: 3
Duyệt các phần tử trong HashSet:
Apple
Grapes
Banana
```

LinkedHashSet

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

- Có hỗ trợ duy trì thứ tự của các phần tử được thêm vào danh sách
- **Ưu điểm:**
  - Duy trì thứ tự được thêm vào danh sách
  - LinkedHashSet vẫn giữ lại tốc độ truy cập nhanh của HashSet, có thể thêm, xóa và kiểm tra sự tồn tại của phần tử nhanh chóng (độ phức tạp là hằng số)
- **Nhược điểm:**
  - Hiệu suất của LinkedHashSet không bằng HashSet (hơnTreeSet)
  - Tốn bộ nhớ hơn HashSet (sử dụng thêm một con trỏ cho mỗi phần tử để duy trì thứ tự chèn)
  - Không thích hợp cho các yêu cầu đòi hỏi hiệu suất cao (thêm/xóa/sửa, dùng HashMap hoặc ArrayList sẽ nhanh hơn)
  - Không hỗ trợ sắp xếp dữ liệu (khác với TreeSet)

Ví dụ với LinkedHashSet

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

```
Set<String> linkedHashSet = new LinkedHashSet<>();

// Thêm các phần tử vào LinkedHashSet
linkedHashSet.add("Apple");
linkedHashSet.add("Banana");
linkedHashSet.add("Orange");
linkedHashSet.add("Apple"); // Phần tử trùng lặp, sẽ không được thêm vào

// In ra các phần tử trong LinkedHashSet
System.out.println("Các phần tử trong LinkedHashSet:");
for (String element : linkedHashSet) {
    System.out.println(element);
}
```

```
Các phần tử trong LinkedHashSet:
Apple
Banana
Orange
```

TreeSet

- Lưu trữ các phần tử theo thứ tự tăng dần hoặc giảm dần
- Thừa hưởng những đặc điểm từ interface NavigableSet và class SortedSet
- TreeSet sử dụng TreeMap để lưu trữ các phần tử
- **Ưu điểm:**
  - Các phần tử được sắp xếp theo thứ tự
  - Việc tìm kiếm một phần tử trong TreeSet bằng cách sử dụng phương thức contains() có độ phức tạp thấp hơn so với các cấu trúc dữ liệu không được sắp xếp

TreeSet

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

Nhược điểm:

- Cần phải đồng bộ hóa quyền truy cập đồng thời vào TreeSet trong môi trường đa luồng
- Sử dụng bộ nhớ nhiều hơn so với một số cấu trúc dữ liệu khác (lưu trữ thêm thông tin về cây nhị phân)
- Thời gian thêm và xóa phần tử trong TreeSet có độ phức tạp cao hơn so với các cấu trúc dữ liệu không sắp xếp

Ví dụ TreeSet

```
TreeSet<String> treeSet = new TreeSet<>();

// Thêm các phần tử vào TreeSet
treeSet.add("Apple");
treeSet.add("Banana");
treeSet.add("Orange");
treeSet.add("Jack fruit"); // Phần tử trùng lặp, sẽ không được thêm vào

// In ra các phần tử trong TreeSet (đã được sắp xếp)
System.out.println("Các phần tử trong TreeSet:");
for (String element : treeSet) {
    System.out.println(element);
}
```

```
Các phần tử trong TreeSet:
Apple
Banana
Jack fruit
Orange
```

NỘI DUNG

1. Giới thiệu Collection
2. List
   i. ArrayList
   ii. LinkedList
   iii. Vector
3. Set
   i. HashSet
   ii. LinkedHashSet
   iii. TreeSet
4. **Queue**
   i. **PriorityQueue**
   ii. **ArrayDeque**
5. Map Interface
   i. HashMap
   ii. LinkedHashMap
   iii. TreeMap
   iv. HashTable

Queue

- Sử dụng để lưu trữ và quản lý các phần tử theo thứ tự First in, First out (FIFO)
- Các phần tử mới sẽ được thêm vào cuối hàng đợi và phần tử cũ sẽ được xóa khỏi đầu hàng đợi

Các phương thức Queue

- **add**(): Thêm một phần tử vào cuối hàng đợi
- **remove**(): Xóa phần tử đầu tiên trong hàng đợi
- **peek**(): Truy cập phần tử đầu tiên trong hàng đợi (mà không xóa)
- **poll**(): Lấy và xóa phần tử đầu tiên ra khỏi hàng đợi

PriorityQueue

- Mỗi phần tử có thể được sắp xếp theo thứ tự ưu tiên
- **Ưu điểm:**
  - Giải quyết vấn đề như tìm kiếm đường đi ngắn nhất trong đồ thị, triển khai các thuật toán như Dijkstra và A*
  - Sử dụng heap và cách tổ chức dữ liệu, có thể xử lý các tác vụ như việc thêm và xóa phần tử với độ phức tạp là O(log n)
- **Nhược điểm:**
  - Không cung cấp một phương thức cụ thể để kiểm tra xem một phần tử đó có tồn tại trong hàng đợi hay không (phải duyệt qua hàng đợi để kiểm tra thủ công)
  - Không thích hợp với các kiểu dữ liệu tùy chỉnh mà không có định nghĩa về sự so sánh

PriorityQueue

```
PriorityQueue<Integer> priorityQueue = new PriorityQueue<>();

// Thêm các phần tử vào PriorityQueue sử dụng phương thức offer()
priorityQueue.offer(30);
priorityQueue.offer(10);
priorityQueue.offer(20);

// Hiển thị PriorityQueue ban đầu
System.out.println("PriorityQueue ban đầu: " + priorityQueue);

// Lấy và hiển thị các phần tử từ PriorityQueue sử dụng phương thức poll()
System.out.println("Các phần tử từ PriorityQueue:");
while (!priorityQueue.isEmpty()) {
    System.out.println(priorityQueue.poll());
}
```

```
PriorityQueue ban đầu: [10, 30, 20]
Các phần tử từ PriorityQueue:
10
20
30
```

ArrayDeque

- Cho phép chúng ta có thể thêm một phần tử từ cả hai phía
- **Ưu điểm:**
  - Hiệu suất được xem là tốt nhất trong Collection (độ phức tạp O(1) để chèn, xóa và truy xuất) ArrayDeque không có giới hạn dung lượng
  - Tự động mở rộng khi chúng ta đạt giới hạn kích thước
- **Nhược điểm:**
  - Cần cân nhắc khi sử dụng trong môi trường đa luồng (không thread-safe)
  - Không thể thực hiện các thao tác liên quan đến index trên ArrayDeque

Ví dụ ArrayDeque

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

```
ArrayDeque<Integer> arrayDeque = new ArrayDeque<>();

// Thêm các phần tử vào ArrayDeque sử dụng phương thức add()
arrayDeque.add(10);
arrayDeque.add(20);
arrayDeque.add(30);

// Hiển thị ArrayDeque
System.out.println("ArrayDeque ban đầu: " + arrayDeque);

// Xóa phần tử đầu tiên sử dụng phương thức remove()
arrayDeque.remove();

// Hiển thị ArrayDeque sau khi xóa phần tử đầu tiên
System.out.println("ArrayDeque sau khi xóa phần tử đầu tiên: " + arrayDeque);

// Lấy phần tử đầu tiên sử dụng phương thức peek()
int firstElement = arrayDeque.peek();
System.out.println("Phần tử đầu tiên của ArrayDeque: " + firstElement);
```

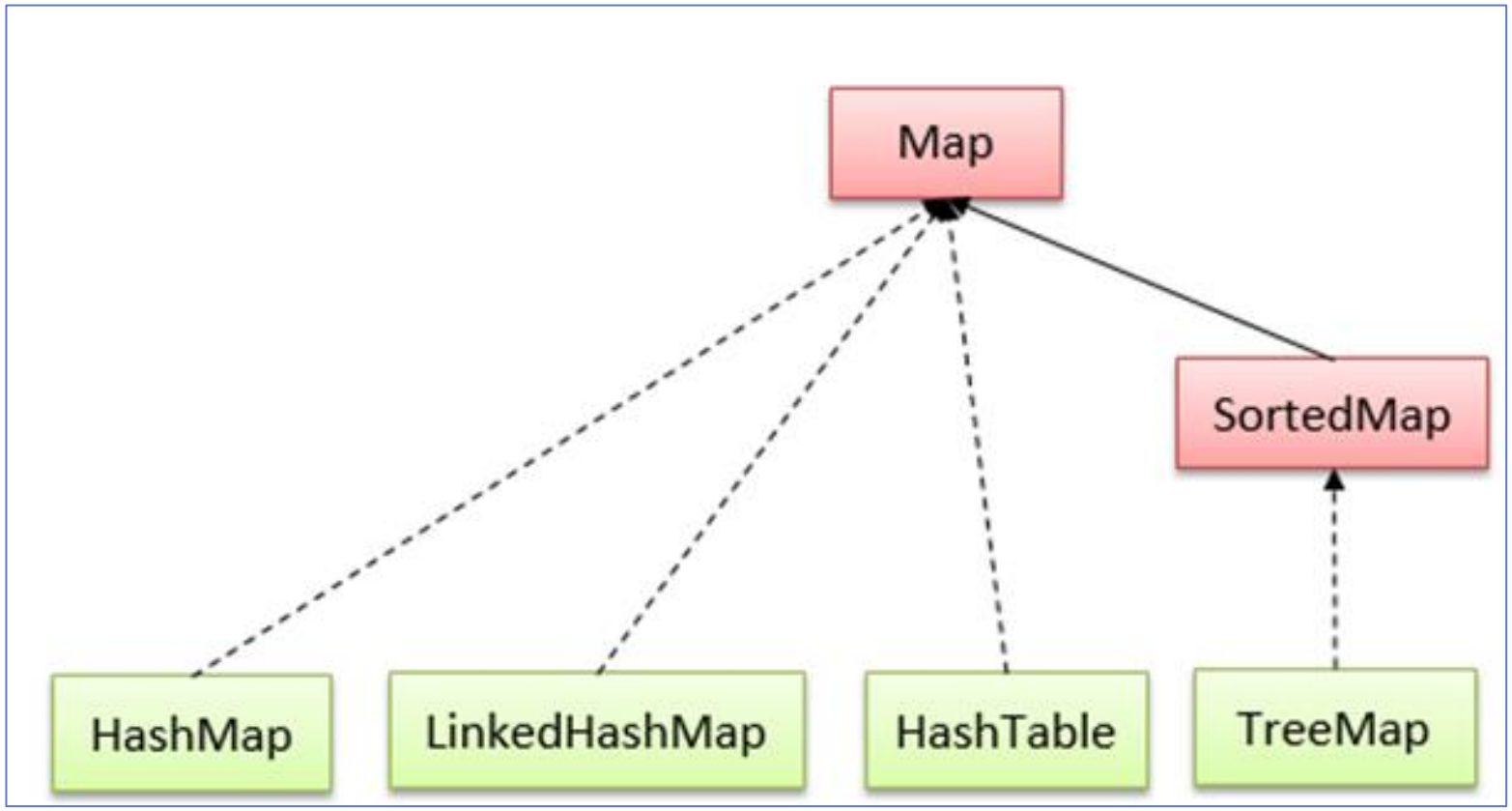
```
ArrayDeque ban đầu: [10, 20, 30]
ArrayDeque sau khi xóa phần tử đầu tiên: [20, 30]
Phần tử đầu tiên của ArrayDeque: 20
```

NỘI DUNG

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

1. Giới thiệu Collection
2. List
   i. ArrayList
   ii. LinkedList
   iii. Vector
3. Set
   i. HashSet
   ii. LinkedHashSet
   iii. TreeSet
4. Queue
   i. PriorityQueue
   ii. ArrayDeque
5. **Map Interface**
   i. **HashMap**
   ii. **LinkedHashMap**
   iii. **TreeMap**
   iv. **HashTable**

Map



Giới thiệu HashMap

- Từ JDK 1.2
- Dùng bảng băm và hiện thực giao diện Map
- Gồm các cặp (key, value)
- Cho phép key hay value nhận giá trị null, không đồng bộ (điểm khác với Hashtable)
- ***import java.util.HashMap;***
- Khai báo lớp:
  public class **HashMap**<K,V> ***extends*** **AbstractMap**<K,V>
  implements **Map**<K,V>, **Cloneable**, **Serializable**

Giới thiệu HashMap

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

- **Ưu điểm:**
  - Độ phức tạp của các thao tác (thêm, xóa, truy vấn) là hằng số
  - Phù hợp cho các bài toán có nhiều truy vấn và thao tác trên dữ liệu
  - Có thể lưu trữ null value và null key
- **Nhược điểm:**
  - Không lưu thứ tự của các phần tử
  - Không cho phép thêm các key trùng lặp (thêm một cặp key-value với key đã tồn tại nó sẽ ghi đè giá trị cũ)

HashMap

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

- **Constructors**:
  - HashMap( )
  - HashMap(Map m)
  - HashMap(int capacity)
  - HashMap(int capacity, float fillRatio)
- Phương thức:
  - void **clear**()
  - Object **clone**()
  - boolean **containsKey**(Object key)
  - boolean **containsValue**(Object value)
  - Set **entrySet**()

Các phương thức

- Object **get**(Object key)
- boolean **isEmpty**()
- Set **keySet**()
- Object **put**(Object key, Object value)
- void **putAll**(Map m): Thêm các phần tử m vào map, nếu trùng key sẽ thay thế giá trị
- Object **remove**(Object key)
- int **size**()
- Collection **values**()

Chương trình minh họa HashMap 1

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

```
public class MapTest {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();
        map.put("An", 10);
        map.put("Nam", 30);
        map.put("Chi", 20);
        System.out.println(map);
        map.put("Tai", 30);
        map.put("Tai", 33);
        System.out.println(map);
        map.remove("An");
        System.out.println(map);
        for(Map.Entry<String, Integer> e: map.entrySet())
            System.out.println(e.getKey() + "_" + e.getValue());
    }
}
```

```
{Chi=20, Nam=30, An=10}
{Chi=20, Nam=30, An=10, Tai=33}
{Chi=20, Nam=30, Tai=33}
Chi_20
Nam_30
Tai_33
```

Chương trình minh họa HashMap 2

```
Map <String, Float> MyMap = new HashMap <>();
MyMap.put("An", 9.1f);
MyMap.put("Châu", 7.5f);
MyMap.put("Nam", 8.3f);
System.out.print("Danh sách map ban đầu: ");
System.out.println(MyMap);
MyMap.put("Châu", 5.5f);
System.out.print("Danh sách map khi thêm Châu lần nữa: ");
System.out.println(MyMap);
System.out.println("Điểm của Châu là: " + MyMap.get("Châu"));
System.out.println("Danh sách có ai tên An không? " + MyMap.containsKey("An"));
System.out.print("Danh sách sau khi bỏ Châu là: ");
MyMap.remove("Châu");
System.out.println(MyMap);
```

```
Danh sách map ban đầu: {Châu=7.5, Nam=8.3, An=9.1}
Danh sách map khi thêm Châu lần nữa: {Châu=5.5, Nam=8.3, An=9.1}
Điểm của Châu là: 5.5
Danh sách có ai tên An không? true
Danh sách sau khi bỏ Châu là: {Nam=8.3, An=9.1}
```

LinkedHashMap

- Kế thừa tất cả tính năng của HashMap
- Bổ sung thêm các đặc điểm lưu thứ tự phần tử được chèn vào
- **Ưu điểm:**
  - Duyệt qua LinkedHashMap theo thứ tự chúng đã được thêm vào, giúp chúng ta truy xuất dữ liệu theo thứ tự
- **Nhược điểm:**
  - Không thích hợp cho việc sắp xếp theo key
  - Nếu thêm lại một phần tử có sẵn với key giống nhau, thứ tự của phần tử đó sẽ không thay đổi
  - LinkedHashMap là một cấu trúc dữ liệu phù hợp cho các tác vụ cần giữ nguyên thứ tự chèn và thao tác tra cứu nhanh nhưng cần xem xét các ưu điểm và nhược điểm trước khi sử dụng nó trong dự án.

Ví dụ LinkedHashMap

```
Map <String, Float> MyMap = new LinkedHashMap <>();
MyMap.put("An", 9.1f);
MyMap.put("Châu", 7.5f);
MyMap.put("Nam", 8.3f);
System.out.print("Danh sách map ban đầu: ");
System.out.println(MyMap);
MyMap.put("Châu", 8.5f);
System.out.print("Danh sách map khi thêm Châu lần nữa: ");
System.out.println(MyMap);
System.out.println("Điểm của Châu là: " + MyMap.get("Châu"));
System.out.println("Danh sách có ai tên An không? " + MyMap.containsKey("An"));
System.out.print("Danh sách sau khi bỏ Châu là: ");
MyMap.remove("Châu");
System.out.println(MyMap);
```

```
Danh sách map ban đầu: {An=9.1, Châu=7.5, Nam=8.3}
Danh sách map khi thêm Châu lần nữa: {An=9.1, Châu=8.5, Nam=8.3}
Điểm của Châu là: 8.5
Danh sách có ai tên An không? true
Danh sách sau khi bỏ Châu là: {An=9.1, Nam=8.3}
```

TreeMap

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

- **Ưu điểm:**
  - Tự động sắp xếp các phần tử theo thứ tự tăng dần của key, giúp cho việc truy xuất dữ liệu nhanh chóng
  - Các thao tác của TreeMap có hiệu suất cao
  - Có thể sử dụng các kiểu dữ liệu khác nhau cho key và value
- **Nhược điểm:**
  - Không cho phép giá trị null làm key (nếu không sẽ ném ra ngoại lệ NullPointerException)
  - Việc thêm, xóa hoặc sửa đổi các phần tử trong TreeMap sẽ mất nhiều thời gian hơn so với các cấu trúc dữ liệu khác

TreeMap

```
TreeMap<Integer,String> hm=new TreeMap<>();
hm.put(102,"Amit");
hm.put(100,"Ravi");
hm.put(101,"Vijay");
hm.put(103,"Rahul");
hm.put(100,"Trong");
```

```
100 Trong
101 Vijay
102 Amit
103 Rahul
```

```
for(Map.Entry<Integer, String> m:hm.entrySet())
 System.out.println(m.getKey()+" "+m.getValue());
```

Hashtable

- Sử dụng để lưu trữ cặp key-value
- Cung cấp các phương thức để thêm, truy xuất, cập nhật và xóa các phần tử theo key
- không cho phép bất kỳ key hoặc giá trị null
- Có đồng bộ nên chậm hơn HashMap
- Được duyệt bởi Enumerator và Iterator
- ConcurrentHashMap là lựa chọn thay thế tối ưu hơn

Hashtable

```
Hashtable<String, Integer> hashtable = new Hashtable<>();
hashtable.put("John", 25);
hashtable.put("Alice", 30);
hashtable.put("Bob", 28);
System.out.println("Age of Alice: " + hashtable.get("Alice"));
hashtable.remove("Bob");
for (String name : hashtable.keySet()) {
    System.out.println(name + " is " + hashtable.get(name) + " years old");
}
```

```
Age of Alice: 30
John is 25 years old
Alice is 30 years old
```

Ví dụ ConcurrentHashMap

```
ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();
// Thêm các phần tử
map.put("A", 1);
map.put("B", 2);
map.put("C", 3);
map.put("D", 4);
// Truy xuất các phần tử
System.out.println("Giá trị của B: " + map.get("B"));
map.forEach((key, value) -> System.out.println(key + ": " + value));
map.remove("B");
map.putIfAbsent("B", 3);
System.out.println(map);
```

```
Giá trị của B: 2
A: 1
B: 2
C: 3
D: 4
{A=1, B=3, C=3, D=4}
```

Tóm tắt bài học

- Enum dùng định nghĩa tập hợp các hằng số, là một kiểu đặc biệt của lớp trong Java, có thể chứa các trường dữ liệu, phương thức và constructor, có thể được định nghĩa bên trong hoặc bên ngoài một lớp
- Lớp System chứa một số trường dữ liệu và phương thức hữu ích, không cần khởi tạo khi sử dụng, cung cấp các thao tác trên luồng nhập, xuất, sao chép các phần tử của mảng, nạp các file dữ liệu và thư viện,…
- Từ JDK 8, các lớp tiện ích liên quan đến quản lý ngày, thời gian và zone gồm có LocalDate, LocalTime, LocalDateTime, ZonedDateTime thuộc package **java.time.***

Tóm tắt bài học

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

- 2 loại cơ bản của Java Collections:
  - Collection Interface
    - List: ArrayList, LinkedList, Vector
    - Set: HashSet, LinkedHashSet, TreeSet
    - Queue: PriorityQueue, ArrayDeque
  - Map Interface: TreeMap, HashMap, LinkedHashMap, HashTable

	Duplicates Allowed	Elements Ordered	Elements Sorted	Synchronized
ArrayList	YES	YES	NO	NO
LinkedList	YES	YES	NO	NO
Vector	YES	YES	NO	YES
HashSet	NO	NO	NO	NO
LinkedHashSet	NO	YES	NO	NO
TreeSet	NO	YES	YES	NO
HashMap	NO	NO	NO	NO
LinkedHashMap	NO	YES	NO	NO
HashTable	NO	NO	NO	YES
TreeMap	NO	YES	YES	NO