CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Bài 03: Ngăn xếp - Hàng đợi

Nguyễn Thị Tâm
nguyenthitam.hus@gmail.com

Ngày 28 tháng 2 năm 2025

Nội dung

Nội dung

Kiểu dữ liệu nguyên thuỷ

Kiểu dữ liệu

Các kiểu dữ liệu được đặc trưng bởi

- Tập các giá trị
- Cách biểu diễn dữ liệu được sử dụng chung cho tất cả các giá trị
- Tập các phép toán có thể được thực hiện trên tất cả các giá trị này

Ví dụ các kiểu dữ liệu nguyên thuỷ trong Java

Bảng 1: Các kiểu dữ liệu cơ bản - Primitive data types

Kiểu	**Gt. mặc định**	**Kích thước**	**Khoảng giá trị**
char	0	2 bytes	0 .. 65535 (Unicode)
boolean	false	1 bit	false/true
short	0	2 bytes	-32768 .. 32767
int	0	4 bytes	-2.147.483.648 .. 2.147.483.647
long	0	8 bytes	$-2^{63}+1..2^{63}$
float	0.0	4 bytes	$+/-1.4023\times10^{-45}..3.4028\times10^{38}$
double	0.0	8 bytes	$+/-4.9406\times10^{-324}..1.7977\times10^{308}$

Kiểu dữ liệu trừu tượng

Kiểu dữ liệu trừu tượng (Abstract Data Type - ADT)

Kiểu dữ liệu trừu tượng bao gồm:

- Tập các giá trị
- Tập các phép toán có thể thực hiện được trên tất cả các giá trị này

Rõ ràng không có cách biểu diễn dữ liệu chung cho dữ liệu trừu tượng

Kiểu	Thành phần	Phép toán
Mảng	Các phần tử	khởi tạo, chèn
Danh sách	Các phần tử	chèn, xoá, tìm,..
Đồ thị	Đỉnh, cạnh	duyệt, tìm đường,...
Ngăn xếp	Các phần tử	thêm, lấy,...
Hàng đợi	Các phần tử	thêm, lấy,...
Cây	Gốc, lá, cành	duyệt, tìm kiếm,....

Giao diện trong Java - Java Interface

- Mỗi đối tượng tương tác với các đối tượng khác thông qua các phương thức công khai
  - Các phương thức này tạo thành giao diện lớp
  - Người lập trình không biết phương thức được cài đặt như thế nào, chỉ cần biết đặc tả vào/ra của phương thức => Tính đóng gói (encapsulation) của lập trình hướng đối tượng
- 2 đối tượng thuộc 2 lớp có thể có các hành vi trên giao diện (nguyên mẫu phương thức) giống nhau, nhưng cách thức thực hiện hành vi (cài đặt phương thức) khác nhau.

Giao diện trong Java - Java Interface

- Các đối tượng thuộc nhiều lớp khác nhau có thể chia sẻ 1 kiểu giao diện (interface) nào đó
  - Các đối tượng này có tất cả các thuộc tính tĩnh và các phương thức khai báo trong giao diện
  - Ngoài các phương thức đã khai báo trong giao diện này, mỗi lớp có thể có các phương thức riêng.
- Kiểu giao diện interface
  - Được khai báo trong một tệp có đuôi .java, tên tệp trùng với tên giao diện;
  - Được biên dịch thành tệp mã byte có đuôi .class như các lớp (class) thông thường;
  - Không có hàm dựng; có thể có các thuộc tính tĩnh, chứa một số bất kì các phương thức trừu tượng, tức là phương thức chỉ có khai báo nguyên mẫu, không định nghĩa thân hàm;
  - Từ Java 8, interface còn chứa các phương thức ngầm định default và các phương thức tĩnh, các phương thức này có cài đặt thân hàm.

Cách khai báo giao diện

Tên tệp: NameOfInterface.java

```
/* File name: <NameOfInterface>.java */
//import statements
public interface NameOfInterface {
// final, static fields
// method declarations
}
```

Giao diện trong Java - Java Interface

```
public interface Shape {
   double getPerimeter();
   double getArea();

   default int compareTo(Shape shape){
      if(getArea() > shape.getArea()) {
         return 1;
      }
      if(getArea() == shape.getArea()) {
         return 0;
      }
      return -1;
   }
}
```

Giao diện trong Java - Java Interface

- Giao diện là kiểu trừu tượng, không thể tạo đối tượng giao diện.

```
Shape shape = new Shape (); // Sai
Shape shape = new Square(); // Dung
```

- Lớp sử dụng (cài đặt) giao diện:
  - Một lớp được khai báo là cài đặt một giao diện bắt buộc phải cài đặt (implements) cụ thể tất cả các phương thức trừu tượng của giao diện. Việc cài đặt này được coi là ghi đè (override) lên các phương thức của giao diện.
  - Với các phương thức ngầm định (default), lớp cài đặt giao diện cũng có thể ghi đè lên phương thức này.
  - Lớp cài đặt giao diện có thể định nghĩa phương thức khác có chung nguyên mẫu với phương thức tĩnh của giao diện - đây không phải là thao tác ghi đè. Việc gọi phương thức tĩnh được thực hiện thông qua tên lớp hoặc tên giao diện.
  - Chương trình có thể dùng giao diện làm tên kiểu cho 1 biến đối tượng. Có thể ép kiểu giao diện về kiểu lớp thực tế của biến.
  - Một lớp có thể cài đặt nhiều giao diện.

Giao diện trong Java - Java Interface

Lớp cài đặt một hoặc nhiều giao diện

```
public class NameOfClass implements NameOfInterface1,
    NameOfInterface2 {
// Method implementation of Interface 1
// Method implementation of Interface 2
// other stuffs (fields, methods) of this class
}
```

Giao diện lớp

Ví dụ

```
public class Square implements Shape{
   private double length; // field
   public Square(){}
   public Square(double a){
      length = a;
   }
   public double getLength(){
      return length;
   }
   // Implements methods of Shape interface
   public double getPerimeter(){
      return 4*length;
   }
   public double getArea(){
      return length*length;
   }
}
```

Nội dung

Danh sách liên kết

- Một danh sách liên kết là một cấu trúc dữ liệu bao các phần tử (nút), trong đó, mỗi phần tử chứa một liên kết đến phần tử tiếp theo trong danh sách.
- Danh sách liên kết có thể là danh sách liên kết đơn (*singly linked list*), danh sách liên kết đôi (*doubly linked list*) hoặc danh sách liên kết vòng (*circular linked list*)
- Mỗi phần tử gồm hai phần: dữ liệu và một liên kết đến phần tử tiếp theo trong danh sách.
- Đầu của danh sách (*head*) và được duy trì trong một biến riêng biệt.
- Cuối của danh sách được chỉ định bằng giá trị **null**.

Danh sách liên kết đơn

- Mô tả danh sách liên kết đơn

```
class Node {
    int data;
    Node next;
}
public Node(int data) {
    this.data = data;
    this.next = null;
}
Node head = null;
```

Danh sách liên kết đơn

- Mô tả danh sách liên kết đơn với kiểu dữ liệu tổng quát

```
class LinkedList<E>{
    class Node{
        E data;
        Node next;
        public Node(E data){
              this.data = data;
              this.next = null;
        }
  }
}
```

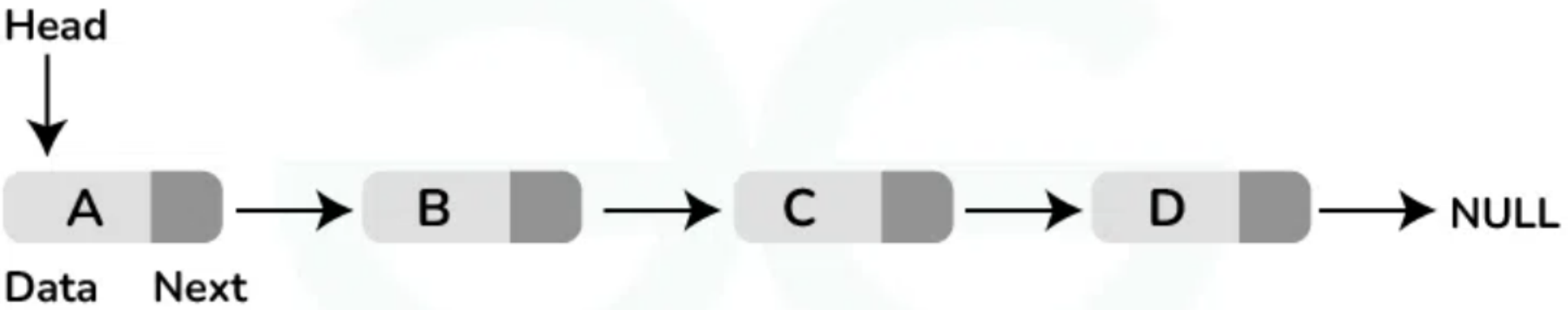
Danh sách liên kết đơn

Lấy độ dài của danh sách

```
Node head;
public int getLength()
{
    int length = 0;
    Node currentNode = head;
    while (currentNode != null){
        length++;
        currentNode = currentNode.next;
    }
    return length;
}
```

Danh sách liên kết đơn

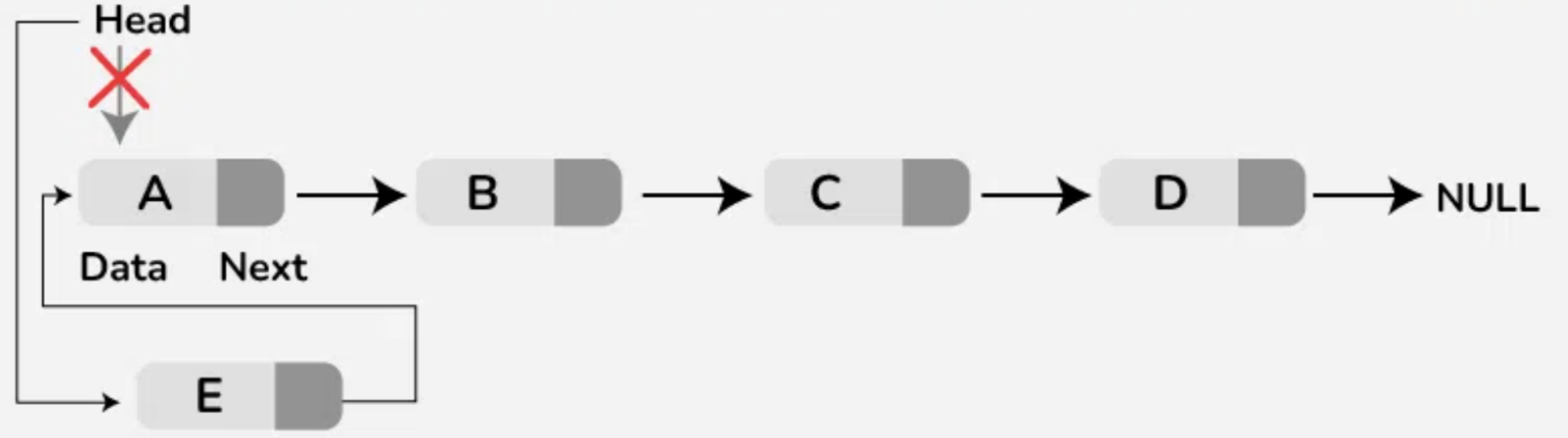
Chèn một phần tử vào đầu danh sách



- Tạo một nút mới: Tạo một nút mới chứa giá trị của phần tử cần chèn.
- Gắn liên kết: Đặt liên kết của nút mới để trỏ đến nút đầu tiên của danh sách hiện tại.
- Cập nhật nút đầu: Gán nút mới làm nút đầu tiên của danh sách.

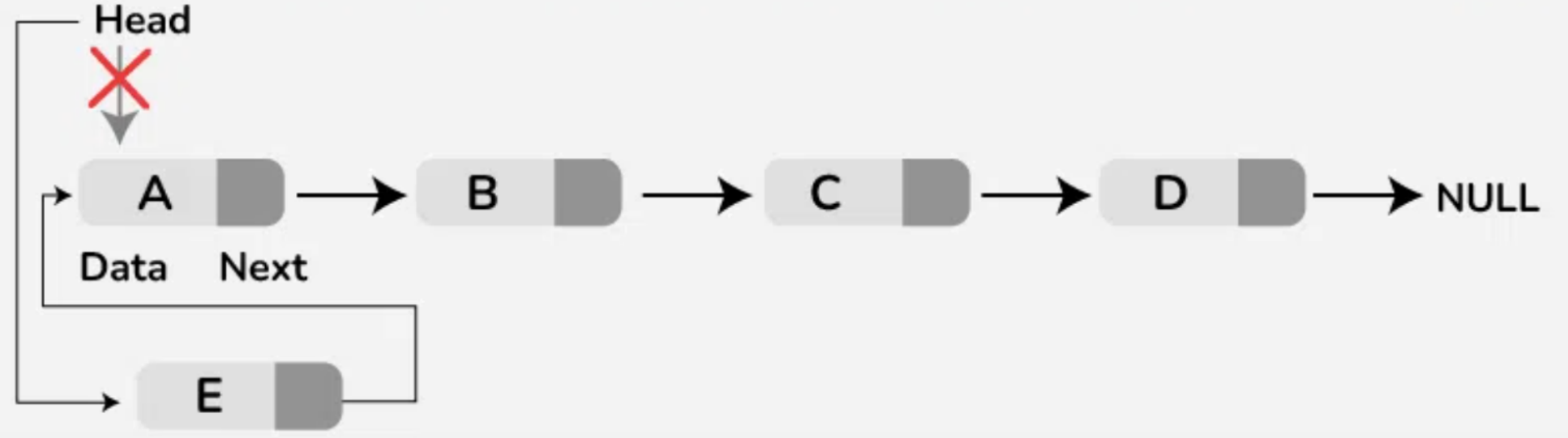
Danh sách liên kết đơn

Chèn một phần tử vào đầu danh sách



Danh sách liên kết đơn

Chèn một phần tử vào đầu danh sách

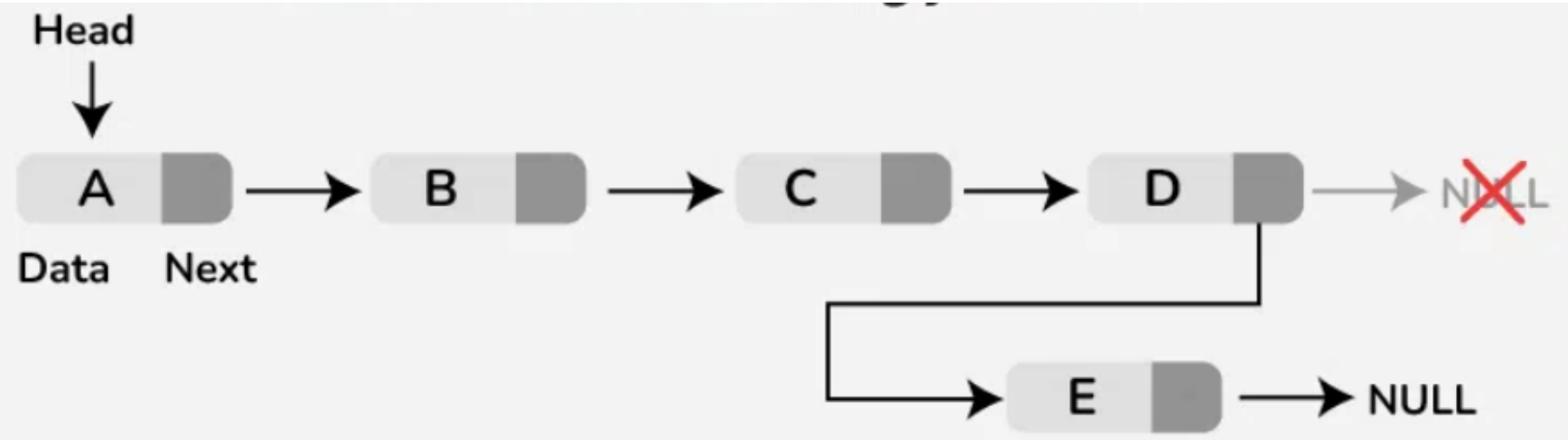



```
public void insertAtBeginning(E data) {
    Node newNode = new Node(data);
    newNode.next = head;
    head = newNode;
}
```

Danh sách liên kết đơn

Chèn phần tử vào cuối danh sách

- Tạo một nút mới: Tạo một nút mới chứa giá trị của phần tử cần chèn.
- Di chuyển đến cuối danh sách: Lặp qua danh sách cho đến khi đến được nút cuối cùng.
- Gắn liên kết: Đặt liên kết của nút cuối cùng của danh sách để trỏ đến nút mới.



Danh sách liên kết đơn

Chèn phần tử vào cuối danh sách

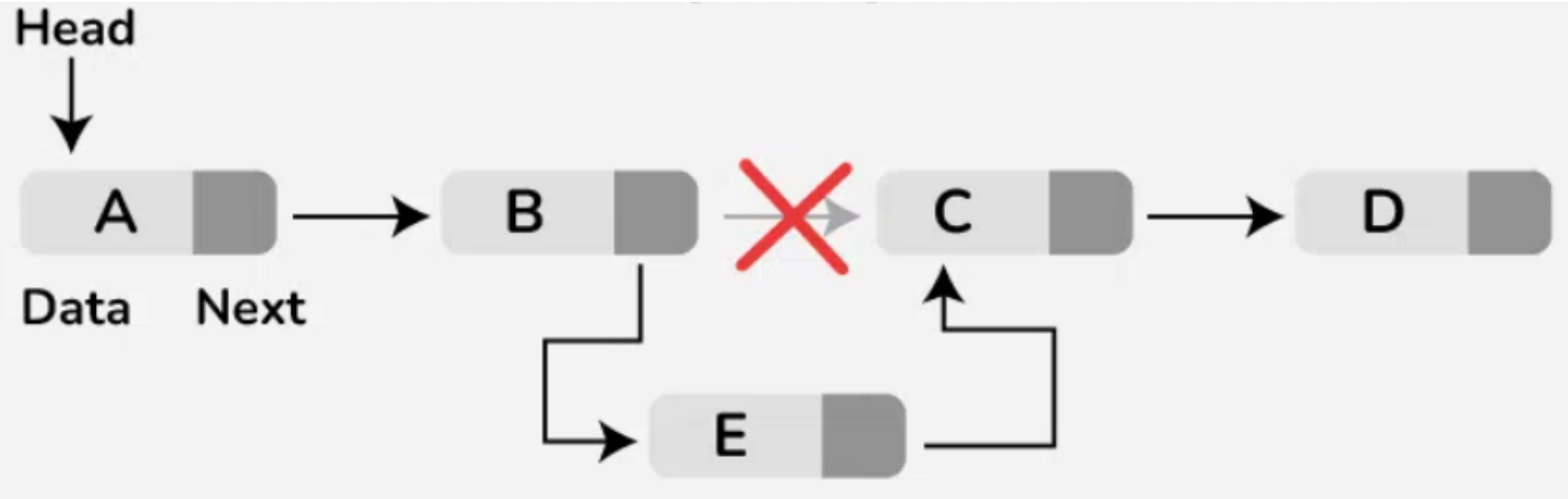
- Tạo một nút mới: Tạo một nút mới chứa giá trị của phần tử cần chèn.
- Di chuyển đến cuối danh sách: Lặp qua danh sách cho đến khi đến được nút cuối cùng.
- Gắn liên kết: Đặt liên kết của nút cuối cùng của danh sách để trỏ đến nút mới.

```
public void insertAtEnd(E data) {
    Node newNode = new Node(data);
    if (head == null) {
        head = newNode;
    } else {
        Node current = head;
        while (current.next != null) {
            current = current.next;
        }
        current.next = newNode;
    }
}
```

Danh sách liên kết đơn

Chèn phần tử vào vị trí bất kỳ

- Tạo một nút mới: Tạo một nút mới chứa giá trị của phần tử cần chèn.
- Tìm vị trí chèn: Di chuyển qua danh sách để đến vị trí trước vị trí cần chèn.
- Gắn liên kết: Đặt liên kết của nút mới để trỏ đến nút tiếp theo của nút hiện tại và gắn liên kết của nút hiện tại để trỏ đến nút mới.



Danh sách liên kết đơn

Chèn phần tử vào vị trí bất kỳ

```
public void insertAt(E data, int position) {
    Node newNode = new Node(data);
    if (position >= 0 and position <= countElement())
    {
        if (position == 0)
            insertAtBeginning(data);
        else
            if (position == getLength())
                insertAtEnd(data);
            else{
                Node current = head;
                int count = 0;
                while (current != null && count < position - 1) {
                    current = current.next;
                    count++; }
                newNode.next = current.next;
                current.next = newNode;
            }
```

Danh sách liên kết đơn

Chèn phần tử vào vị trí bất kỳ

```
        else
        {
            try {
                  throw new Exception("Invalid position");
                } catch (Exception e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
                }
        }
    }
```

Danh sách liên kết đơn

Xoá phần tử ở vị trí đầu tiên

- Kiểm tra xem danh sách có rỗng không. Nếu danh sách rỗng (không có phần tử nào), không có gì để xóa, chỉ cần trả về.
- Đặt nút đầu tiên của danh sách (head) để trỏ đến phần tử tiếp theo của nó (head.next).
- Giải phóng bộ nhớ của nút cũ (nút bị xóa).

```
public void deleteFirst() {
    if (head == null) {
        System.out.println("LinkedList is null");
        return;
    }
    head = head.next;
}
```

Danh sách liên kết đơn

Xoá phần tử ở vị trí cuối cùng

- Kiểm tra xem danh sách có rỗng không. Nếu danh sách rỗng (không có phần tử nào), không có gì để xóa, chỉ cần trả về.
- Nếu danh sách chỉ có một phần tử, xóa phần tử đó bằng cách đặt head trỏ đến null.
- Di chuyển đến phần tử cuối cùng của danh sách và phần tử trước nó.
- Đặt liên kết của phần tử trước phần tử cuối cùng để trỏ đến null, giải phóng phần tử cuối cùng.

Danh sách liên kết đơn (8)

Xoá phần tử ở vị trí cuối cùng

```
public void deleteLast() {
    if (head == null) {
        System.out.println("Linked List is null");
        return;
    }
    if (head.next == null) {
        head = null;
        return;
    }

    Node previous = null;
    Node current = head;
    while (current.next != null) {
        previous = current;
        current = current.next;
    }
    previous.next = null;
}
```

Danh sách liên kết đơn

Xoá phần tử ở vị trí bất kỳ

- Kiểm tra xem danh sách có rỗng không. Nếu danh sách rỗng (không có phần tử nào) hoặc vị trí cần xóa là vị trí đầu tiên, chỉ cần sử dụng phương thức deleteFirst() để xóa phần tử đầu tiên.
- Di chuyển qua danh sách để đến vị trí trước vị trí cần xóa.
- Xác định phần tử cần xóa và phần tử trước nó.
- Đặt liên kết của phần tử trước phần tử cần xóa để trỏ đến phần tử sau phần tử cần xóa.
- Giải phóng bộ nhớ của phần tử cần xóa.

Danh sách liên kết đơn

Xoá phần tử ở vị trí bất kỳ

```
    public void deleteAt(int position) {
        if (head == null || position == 0) {
            deleteFirst();
            return;
        }
        Node previous = null;
        Node current = head;
        int count = 0;
        while (current != null && count < position) {
            previous = current;
            current = current.next;
            count++;
        }
        if (current == null) {
            System.out.println("Position is invalid");
            return;
        }
        previous.next = current.next; current = null;
}
```

Danh sách liên kết đơn (11)

Một số bài tập

1. Viết chương trình để chèn một phần tử vào danh sách liên kết đã được sắp
2. Viếtchương trình để đảo ngược một danh sách liên kết
3. Viết chương trình để tìm phần tử lớn nhất/nhỏ nhất trong danh sách liên kết
4. Cho hai danh sách được sắp, viết chương trình để gộp 2 danh sách được sắp thành một danh sách được sắp
5. ....

Danh sách liên kết đơn (11)

So sánh danh sách liên kết đơn và mảng

	Danh sách liên kết đơn	Mảng
Kích thước	Thay đổi	Cố định
Indexing	O(n)	O(1)
Chèn tại vị trí đầu	O(1)	O(n)
Xoá tại vị trí đầu	O(1)	O(n)
Chèn tại vị trí cuối	O(n)	O(1)
Xoá tại vị trí cuối	O(n)	O(1)

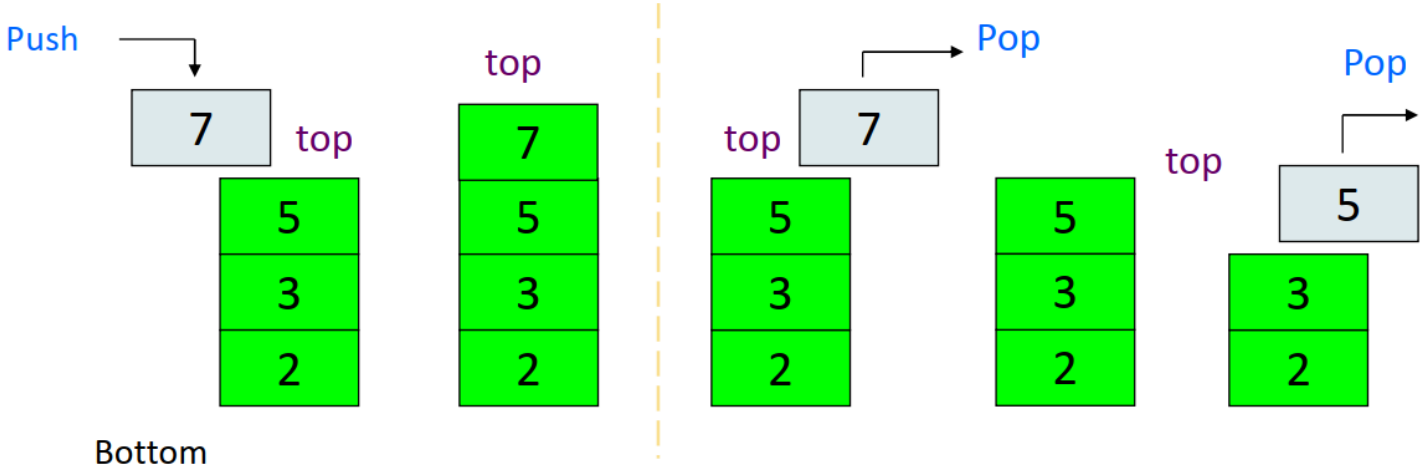
Nội dung

Ngăn xếp - stack

- Là cấu trúc hiệu quả để lưu trữ và lấy dữ liệu theo thứ tự vào sau, ra trước (*last in, first out* - LIFO)
- top: đỉnh stack
- bottom: đáy stack
- Ví dụ:
  - Thao tác back khi duyệt web
  - Thao tác undo khi soạn thảo
  - ...

Ngăn xếp - stack

- Các phương thức cơ bản của stack
  - push(): thêm một phần tử mới vào stack
  - pop(): lấy một phần tử ra khỏi stack
  - isEmpty(): kiểm tra xem stack có rỗng hay không
  - top(): trả về giá trị phần tử ở đỉnh stack
- Lưu trữ stack
  - Lưu trữ kế tiếp dùng mảng
  - Lưu trữ sử dụng danh sách liên kết



Cài đặt ngăn xếp bằng mảng

- Sử dụng cả phân bổ tài nguyên tĩnh và động

```
E[] stack;
```

- Các phần tử được thêm vào và loại bỏ ở cuối mảng

```
int top = 0;
```

Cài đặt ngăn xếp bằng mảng (1)

- Thêm phần tử

```
public void push(E element) {
    stack[top] = element;
    top++;
}
```

- Xoá phần tử

```
public E pop() {
    top--;
    return stack[top];
}
```

Cài đặt ngăn xếp bằng mảng (2)

Ưu điểm

Thao tác	**Độ phức tạp**
push()	O(1)
pop()	O(1)
size()	O(1)
isEmpty()	O(1)

Cài đặt ngăn xếp bằng mảng (3)

Hạn chế

- Cần phải dự đoán kích thước tối đa của mảng.
- Nếu không thể cung cấp kích thước tối đa, chúng ta có hai giải pháp:
  - Thay đổi kích thước của mảng
  - Chọn một cách triển khai khác ngoài mảng

Làm thế nào để thay đổi kích thước của mảng?

Thay đổi kích thước của mảng

- Cách 1: mỗi khi ngăn xếp đầy tăng kích thước lên 1
  - push(): tăng kích thước của ngăn xếp lên 1
  - pop(): giảm kích thước của ngăn xếp đi 1
- Cách 2 (*Repeated Doubling Technique*): Nếu mảng đầy, tạo một mảng mới có kích thước gấp đôi và sao chép các phần tử vào mảng mới. Điều này giúp giảm tần suất cần phải thay đổi kích thước của mảng và giúp việc mở rộng mảng hiệu quả hơn
  Yêu cầu hãy triển khai ngăn xếp bằng mảng theo cách mở rộng mảng thứ hai

Mở rộng mảng

```
public class DynamicArrayStack {
    protected int capacity;
    public static final int CAPACITY = 16; // power of 2
    public static int MINCAPACITY = 1 << 15; // power of 2
    protected int[] stack;
    protected int top = -1;

    public DynamicArrayStack() {
        this(CAPACITY); // default capacity
    }
    public DynamicArrayStack(int cap) {
        capacity = cap;
        stack = new int[capacity];
    }
    // xem trang tiep
```

Mở rộng mảng (1)

```
    public int size() {
        return (top + 1);
    }
    public boolean isEmpty() {
        return (top < 0);
    }
    public void push(int data) throws Exception {
        if (size() == capacity)
            expand();
        stack[++top] = data;
    }
    private void expand() {
        int length = size();
        int[] newstack = new int[length << 1];
        System.arraycopy(stack, 0, newstack, 0, length);
        stack = newstack;
        this.capacity = this.capacity << 1;
    }
// xem trang tiep
```

Mở rộng mảng (2)

```
private void shrink() {
    int length = top + 1;
    if (length <= MINCAPACITY || top << 2 >= length)
        return;
    length = length + (top << 1); // still means shrink to at 1/2 or
        less of the heap
    if (top < MINCAPACITY)
        length = MINCAPACITY;
    int[] newstack = new int[length];
    System.arraycopy(stack, 0, newstack, 0, top + 1);
    stack = newstack;
    this.capacity = length;
}
public int top() throws Exception {
    if (isEmpty())
        throw new Exception("Stack is empty.");
    return stack[top];
}
// xem trang tiep
```

Mở rộng mảng (3)

```
    public int pop() throws Exception {
        int data;
        if (isEmpty()) throw new Exception("Stack is empty.");
        data = stack[top];
        stack[top--] = Integer.MIN_VALUE;
        shrink();
        return data;
    }
    public String toString() {
        String s = "[";
        if (size() > 0)
            s += stack[0];
        if (size() > 1)
            for (int i = 1; i <= size() - 1; i++) {
                s += ", " + stack[i];
            }
        return s + "]";
    }
}
```

Cài đặt ngăn xếp bằng danh sách liên kết

- Lưu trữ trong danh sách liên kết

```
class Node<E>{
    E element;
    Node next;
}
Node stack = null;
```

- "top" của ngăn xếp là phần tử đầu của danh sách liên kết

Cài đặt ngăn xếp bằng danh sách liên kết

- Thêm phần tử vào danh sách

```
public void push(E element) {
    Node node = new Node();
    node.element = element;
    node.next = stack;
    stack = node;
}
```

Cài đặt ngăn xếp bằng danh sách liên kết (1)

- Xoá phần tử

```
public E pop() {
    E element = stack.element;
    stack = stack.next;
    return element;
}
```

Cài đặt ngăn xếp bằng danh sách liên kết (2)

```
public class LinkedListStack<E> implements Stack<E> {
    private class Node {
        E element;
        Node next;
    }
    private Node stack = null;
    public void push(E element) {
        ...
    }
    public E pop() {
        ...
    }
    public boolean isEmpty() {
        return stack == null;
    }
}
```

Ứng dụng của ngăn xếp

- Tính toán biểu thức: $A = b + c * d / e - f$
- **Biểu thức trung tố:** toán tử hai ngôi đứng giữ hai toán hạng, toán tử một ngôi đứng trước toán hạng
- **Biểu thức hậu tố:** toán tử đứng sau toán hạng
- Ví dụ
  - Trung tố: $a * (b - c) / d$
  - Hậu tố: $abc - *d /$
  - Tiền tố: $/ * a - bcd$

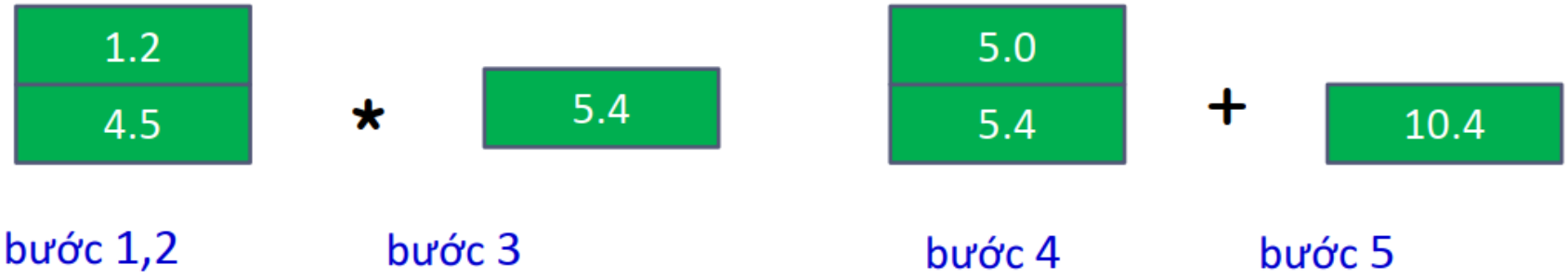
Tính giá trị biểu thức hậu tố

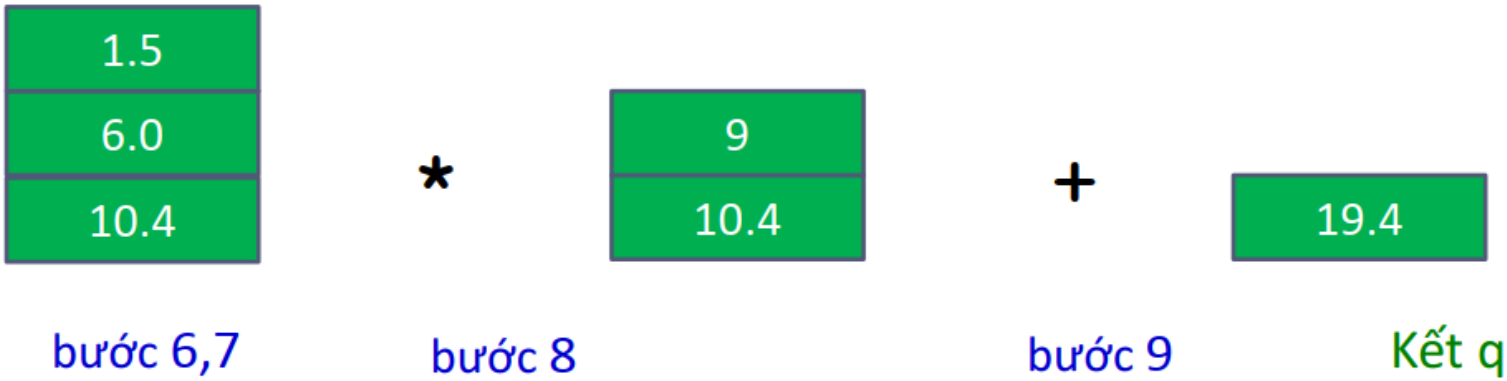
- Tính giá trị biểu thức hậu tố
  - Trung tố: (4.5 * 1.2) + 5.0 + (6.0 * 1.5)
  - Hậu tố: 4.5 1.2 * 5.0 + 6.0 1.5 * +
- Cách tính
  - Duyệt biểu thức hậu tố
    - Gặp toán hạng: đẩy vào stack
    - Gặp toán tử 1 ngôi: lấy ra 1 toán hạng trong stack, áp dụng toán tử lên toán hạng và đẩy kết quả trở lại stack
    - Gặp toán tử 2 ngôi: lấy 2 toán hạng ở đỉnh stack theo thứ tự, áp dụng toán tử lên 2 toán hạng đó, kết quả lại đẩy vào stack.
    - Kết thúc, đưa ra kết quả là giá trị ở đỉnh stack.

Tính giá trị biểu thức hậu tố (1)

Ví dụ minh hoạ

```
4.5 1.2 * 5.0 + 6.0 1.5 * +
```





Tính giá trị biểu thức hậu tố (2)

Ví dụ minh hoạ

$$2\ 4\ 9\ \sqrt{}\ -\ +\ 4\ 2\ \hat{}\ *$$

Bước	Mô tả	Trạng thái stack	Ghi chú
0	Stack rỗng	$\emptyset$	
1	Gặp 2 (toán hạng)	2	
2	4	$2, 4$	
3	9	$2, 4, 9$	
4	$\sqrt{}$ (toán tử 1 ngôi)	$2, 4, 3$	Thực hiện $\sqrt{9}$
5	$-$ (toán tử 2 ngôi)	$2, 1$	Thực hiện $4 - 3$
6	$+$ (toán tử 2 ngôi)	3	Thực hiện $2 + 1$
7	4	$3, 4$	
8	2	$3, 4, 2$	
9	^ (toán tử 2 ngôi)	$3, 16$	Thực hiện 4^2
10	$*$ (toán tử 2 ngôi)	48	Thực hiện $3 * 16$

Chuyển biểu thức trung tố sang biểu thức hậu tố

Ví dụ. Trung tố	Hậu tố
a*b*c*d*e*f	ab*c*d*e*f*
1 + (-5) / (6 * (7+8))	1 5 - 6 7 8 + * / +
(x/y – a*b) * ((b+x) – y) ^{y}	x y / a b * – b x + y y ^ – *
(x*y*z – x^2 / (y*2 – z^3) + 1/z) * (x – y)	xy*z*x2^y2*z3^ – / – 1z/+xy – *

Chuyển biểu thức trung tố sang biểu thức hậu tố (1)

Toán tử	Mức ưu tiên
$\wedge$ toán tử hai ngôi	6
!, $\sim$ (toán tử một ngôi giai thừa và đảo dấu)	6
abs, sin, cos, tan, exp, ln, lg, round, trunc, sqr, sqrt, acrtan	6
*, /, % (toán tử hai ngôi)	5
+, - (toán tử hai ngôi)	4
==, !=, <, > $\leq$, $\geq$ (toán tử quan hệ)	3
Not (toán tử logic)	2
&&, || (toán tử logic)	1
= (toán tử gán)	0

Chuyển biểu thức trung tố sang biểu thức hậu tố (3)

Thuật toán

Duyệt biểu thức trung tố lần lượt từ trái qua phải

1. **Gặp toán hạng**: viết sang biểu thức kết quả (là biểu thức hậu tố cần tìm)
2. **Gặp toán tử** (có độ ưu tiên nhỏ hơn 6):
   - Nếu **stack rỗng,** hoặc đỉnh stack là toán tử có độ ưu tiên nhỏ hơn, hoặc là '(': đẩy toán tử đang xét vào stack
   - **Ngược lại**: lấy các toán tử ở đỉnh stack có độ ưu tiên lớn hơn hoặc bằng toán tử đang xét lần lượt đưa vào biểu thức kết quả và đẩy toán tử đang xét vào stack
3. **Gặp toán tử** có độ ưu tiên 6, hoặc '**(**': đẩy vào stack
4. Gặp '**)**': lấy tất cả các toán tử trong stack cho đến khi gặp '**(**' đầu tiên, đưa sang biểu thức kết quả theo đúng thứ tự (không đưa '(, **)**' vào biểu thức kết quả), và đẩy 1 ký hiệu '**(**' ra khỏi stack.
5. Nếu duyệt hết biểu thức trung tố, lấy nốt những toán tử trong stack đưa sang biểu thức kết quả (theo đúng thứ tự).

Chuyển biểu thức trung tố sang biểu thức hậu tố (4)

Ví dụ

- Chuyển biểu thức sau sang biểu thức hậu tố

$$-3 + \frac{4^{5^a}}{2} - 7 \tag{1}$$

- Biểu thức trên được viết lại là

$$-3 + 4 \wedge (5 \wedge a) / 2 - 7$$

hoặc

$$-3 + 4 \wedge 5 \wedge a / 2 - 7$$

- Chú ý: dấu $-$ trong -3 là toán tử 1 ngôi có độ ưu tiên là 6

Chuyển biểu thức trung tố sang hậu tố

Ví dụ

$$-3 + 4\text{^}5\text{^}a/2 - 7$$

Bước	Mô tả	Trạng thái stack	Kết quả
0	Stack rỗng	Ø	Ø
1	Gặp − (toán tử 1 ngôi)	−	Ø
2	Gặp 3 (toán hạng), đưa sang biểu thức kết quả)	−	3
3	Gặp + (toán tử 2 ngôi), lấy − khỏi stack và đẩy + vào	+	3 −
4	Gặp 4 (toán hạng)	+	3 − 4
5	Gặp ^ (toán tử 2 ngôi bậc 6)	+, ^	3 − 4
6	Gặp 5	+, ^	3 − 4 5
7	Gặp ^	+, ^, ^	3 − 4 5
8	Gặp a (toán hạng)	+, ^, ^	3 − 4 5 a
9	Gặp / (toán tử 2 ngôi), lấy các toán tử có độ ưu tiên lớn hơn hoặc bằng ra khỏi stack, sau đó đẩy / vào	+,/	3 − 4 5 a ^ ^

Chuyển biểu thức trung tố sang biểu thức hậu tố (5)

Ví dụ

$$-3 + 4\hat{}5\hat{}a/2 - 7$$

Bước	Mô tả	Trạng thái stack	Kết quả
10	Gặp 2	$+,/$	$3\ -\ 4\ 5\ a\ \hat{}\ \hat{}\ 2$
11	Gặp $-$ (toán tử 2 ngôi), lấy $/$ và $+$ khỏi stack, rồi đẩy $-$ vào	$-$	$3\ -\ 4\ 5\ a\ \hat{}\ \hat{}\ 2\ /\ +$
12	Gặp 7 (toán hạng)	$-$	$3\ -\ 4\ 5\ a\ \hat{}\ \hat{}2\ /\ +\ 7$
	Đã duyệt song biểu thức, lấy nốt cac toán tử trong stack đưa sang biểu thức kết quả	$\emptyset$	$3\ -\ 4\ 5\ a\ \hat{}\ \hat{}2\ /\ +\ 7\ -$

Biểu thức kết quả: $3\ -\ 4\ 5\ a\ \hat{}\ \hat{}2\ /\ +\ 7\ -$

Nội dung

Hàng đợi

- Là cấu trúc hiệu quả để lưu trữ và lấy ra dữ liệu theo thứ tự vào trước ra trước (*first in, first out*- FIFO). Phần tử được thêm vào trước sẽ được ưu tiên lấy ra trước
- Việc thêm vào và lấy ra được diễn ra ở hai đầu khác nhau. Phần tử được lấy ra ở đầu hàng đợi (front), và được thêm vào ở cuối hàng đợi (rear)
  - Thêm một phần tử vào hàng đợi: `enqueue()`
  - Xoá một phần tử khỏi hàng đợi: `dequeue()`
  - Kiểm tra xem hàng đợi có rỗng hay không: `empty()`
  - Trả về giá trị phần tử ở đầu của hàng đợi, mà không huỷ nó: `front()`
- Ứng dụng
  - Trong hệ điều hành: hàng đợi xử lý sự kiện, quản lý tiến trình, xử lý các lệnh
  - Khử đệ quy
  - Lưu vết quá trình tìm kiếm, quay lui, vét cạn
- Lưu trữ hàng đợi
  - Dùng mảng
  - Dùng danh sách liên kết

Hàng đợi

```
public interface Queue<E> {
    public void enqueue(E element);
    public E dequeue();
    public boolean isEmpty();
    ...
}
```

Cài đặt hàng đợi bằng mảng

- Sử dụng mảng tĩnh hoặc động

```
E queue[n];
```

- Các phần tử được thêm vào cuối và xoá ở đầu
- Cần lưu trữ `front` và `rear`

```
int front = 0, rear = 0;
```

- Hoặc chỉ cần lưu trữ `front` và số lượng phần tử `count`

```
int front = 0, count = 0;
```

Cài đặt hàng đợi bằng mảng (1)

- Thêm phần tử

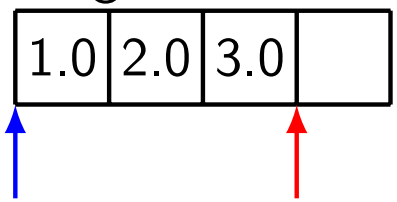
```
public void enqueue(E element) {
    if (count < n) {
        queue[front + count] = element;
        count++;
    }
}
```

- Xoá phần tử

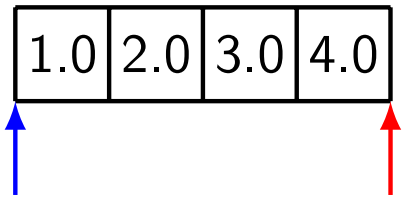
```
public E dequeue() {
    if (count > 0) {
        count--;
        return queue[front++];
    }
    return null;
}
```

Cài đặt hàng đợi bằng mảng (2)

- Triển khai theo cách trên không tốt? Tại sao?
- Xem xét một hàng đợi dưới đây, trong đó ↑ chỉ đầu hàng đợi, ↑ chỉ cuối hàng đợi



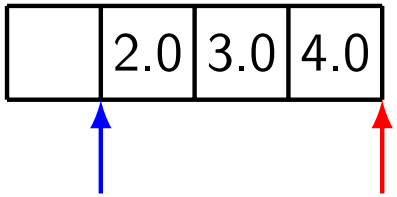
- Thêm phần tử 4.0 vào hàng đợi



Hàng đợi đầy

Cài đặt hàng đợi bằng mảng (3)

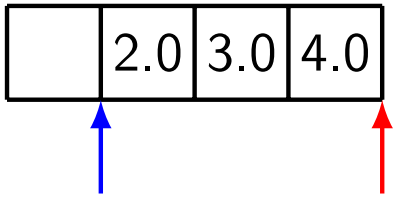
- Lấy phần tử 1.0 ra khỏi hàng đợi



- Điều gì xảy ra nếu thêm phần tử 5.0 vào hàng đợi

Cài đặt hàng đợi bằng mảng (3)

- Lấy phần tử 1.0 ra khỏi hàng đợi



- Điều gì xảy ra nếu thêm phần tử 5.0 vào hàng đợi
- Sửa đổi phương thức enqueue()

```
public void enqueue(E element) {
    if (count < n) {
        queue[(front + count) % n] = element;
        count++;
    }
}
```

Cài đặt hàng đợi bằng mảng (4)

- Sửa đổi phương thức dequeue()

```
public E dequeue() {
    if (count > 0) {
        E element = queue[front];
        count--;
        front++;
        if (front == n) {
            front = 0;
        }
        return element;
    }
    return null;
}
```

Cài đặt hàng đợi bằng mảng (5)

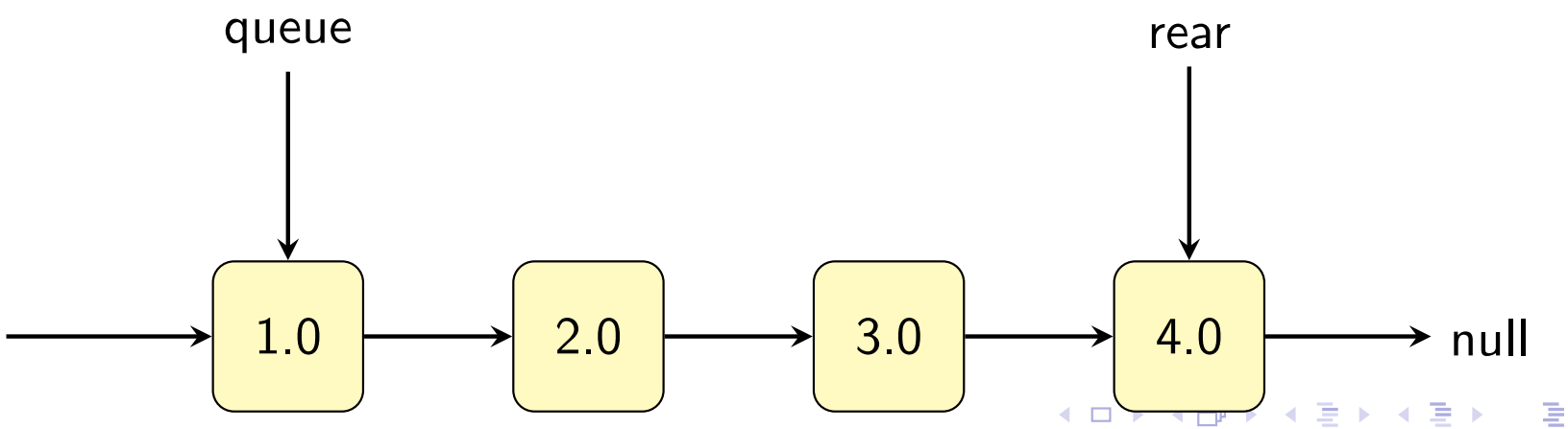
```
public class ArrayQueue<E> implements Queue<E> {
    private E[] queue;
    private int n;
    private int front = 0;
    private int count = 0;
    public ArrayQueue(int capacity) {
        n = capacity;
        queue = (E[]) new Object[capacity];
    }
    public void enqueue(E element) {...}
    public E dequeue() {...}
    public boolean isEmpty() {
        return count == 0;
    }
}
```

Cài đặt hàng đợi bằng danh sách liên kết

```
class Node {
    E element;
    Node next;
}
Node queue = null;
```

- Không cần theo dõi `front`
- Cần theo dõi `rear`

```
Node rear = null;
```



Cài đặt hàng đợi bằng danh sách liên kết (1)

Thêm phần tử

```
public void enqueue(E element) {
    Node node = new Node();
    node.element = element;
    node.next = null;
    if (rear != null) {
        rear.next = node;
    } else {
        queue = node;
    }
    rear = node;
}
```

Thêm phần tử không ảnh hưởng đến đầu của hàng đợi nếu hàng đợi không rỗng

Cài đặt hàng đợi bằng danh sách liên kết (2)

Xoá phần tử

```
public E dequeue() {
    if (queue != null) {
        Node node = queue;
        E element = node.element;
        queue = node.next;
        if (node == rear) {
            rear = null;
        }
        return element;
    }
    return null;
}
```

Xoá phần tử không ảnh hưởng đến đuôi của hàng đợi nếu hàng đợi không rỗng

Cài đặt hàng đợi bằng danh sách liên kết

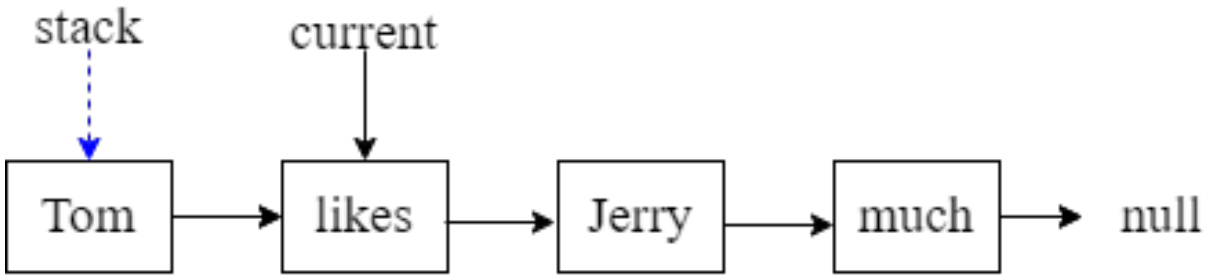
Xoá phần tử

```
public class LinkedListQueue<E> implements Queue<E> {
    private class Node {
        E element;
        Node next;
    }
    private Node queue = null;
    private Node rear = null;
    public void enqueue(E element) {
        ...
    }
    public E dequeue() {
        ...
    }
    public boolean isEmpty() {
        return queue == null;
    }
}
```

Iteration

Thách thức khi thiết kế: Làm thế nào để lặp qua các phần tử của ngăn xếp mà không cần cung cấp thông tin bên trong ngăn xếp.

- Biểu diễn bằng danh sách liên kết



- Biểu diễn bằng mảng

0	1	2	3	4	5
much	Jerry	likes	Tom	null	null
	i		top		

Tạo Stack cài đặt từ giao diện `java.lang.Iterable`

Iteration

- `Iterable`? Liên quan đến việc duyệt qua các phần tử trong một tập hợp dữ liệu (collection) như mảng, danh sách liên kết, hoặc tập hợp (Set). Iterable cung cấp một cách tiếp cận trừu tượng để truy cập từng phần tử trong tập hợp, mà không cần biết chi tiết cài đặt của tập hợp đó. Giao diện này chỉ chứa duy nhất phương thức `iterator()`

```
public interface Iterable<E> {
    Iterator<E> iterator();
}
```

- `Iterator`? được sử dụng để duyệt qua các phần tử trong một tập hợp dữ liệu một cách tuần tự. `Iterator` có thể được sử dụng với bất kỳ tập hợp nào triển khai giao diện `Iterable`.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

Iteration

```
Stack<String> stack = new ArrayStack<String>();
```

Lặp các phần tử của stack

```
while (!stack.isEmpty()) {
    String element = stack.pop();
    System.out.println(element);
}
```

Nếu Stack là Iterable

```
for (String element : stack) {
    System.out.println(element);
}
```

Stack API

```
public interface Stack<E> implements Iterable<E>{
    public void push(E element);
    public E pop();
    public boolean isEmpty();
}
```

Cài đặt ngăn xếp bằng danh sách liên kết

```
public class LinkedListStack<E> implements Stack<E> {
    private Node stack = null;
    public void push(E element) {...}
    public E pop() {...}
    public boolean isEmpty() {...}
    public Iterator<E> iterator() {
        return new LinkedListStackIterator();
    }
}
```

Linked List Stack Iterator

```
class LinkedListStackIterator implements Iterator<E> {
    private Node current = stack;
    public boolean hasNext() {
        return current != null;
    }
    public E next() {
        E element = current.element;
        current = current.next;
        return element;
    }
    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Cài đặt ngăn xếp bằng mảng

```
public class ArrayStack<E> implements Stack<E> {
    private E[] stack;
    private int top = 0;
    public ArrayStack(int capacity) {...}
    public void push(E element) {...}
    public E pop() {...}
    public boolean isEmpty() {...}
    public Iterator<E> iterator() {
        return new ArrayStackIterator();
    }
}
```

Array Stack Iterator

```
class ArrayStackIterator implements Iterator<E> {
    private int i = top;
    public boolean hasNext() {
        return (i > 0);
    }
    public E next() {
        return stack[--i];
    }
    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```