CMC UNIVERSITY

CHƯƠNG 2.
CÁC CẤU TRÚC DỮ LIỆU CƠ BẢN

CMC UNIVERSITY

Nội dung

- ✓ Mảng
- ✓ Danh sách
- ✓ Ngăn xếp
- ✓ Hàng đợi
- ✓ Bảng băm

CMC UNIVERSITY

MẢNG

Các khái niệm cơ bản

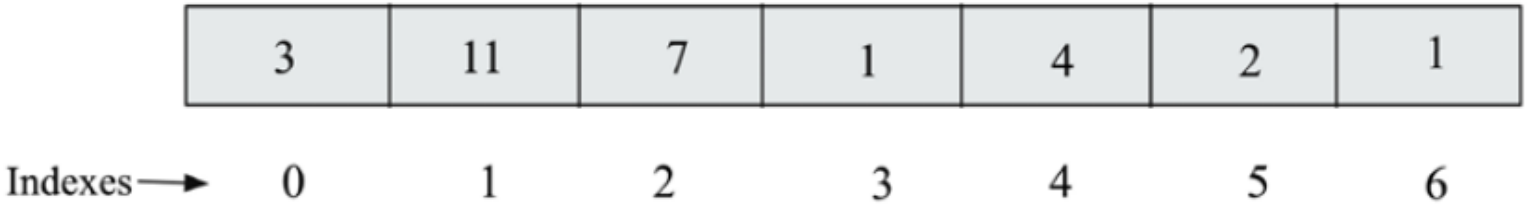
Mảng (Arrays)

- **Mảng** là một tập hợp có thứ tự gồm một số cố định các phần tử cùng kiểu, mỗi phần tử dữ liệu trong mảng được lưu trữ ở các vị trí bộ nhớ liền kề nhau.
- Ngoài giá trị, một phần từ của mảng còn được đặc trưng bởi chỉ số (index) thể hiện thứ tự của phần tử đó trong mảng..
- Vector là mảng một chiều, mỗi phần tử a_i ứng với một chỉ số i. Ma trận là mảng hai chiều, mỗi phần từ a_{ij} ứng với hai chỉ số i và j. Tương tự, có thể mở rộng mảng ba chiều, mảng bốn chiều, …
- Có thể tạo mảng, chèn một phần tử vào mảng, xóa một phần tử khỏi mảng, tìm kiếm các phần tử trong mảng, cập nhật một phần tử của mảng, v.v.

Các khái niệm cơ bản

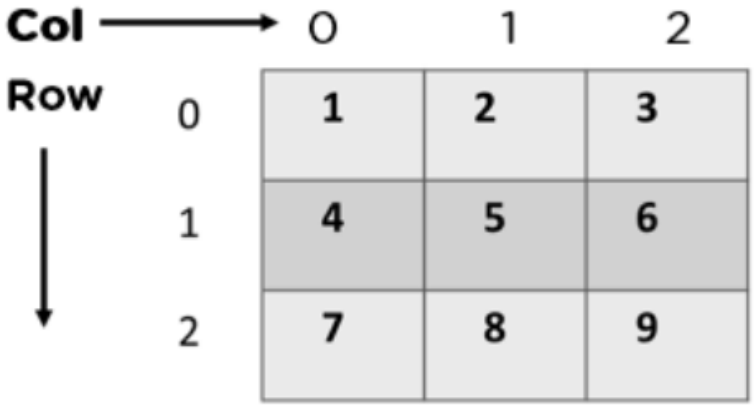
Ví dụ:

- Mảng một chiều:

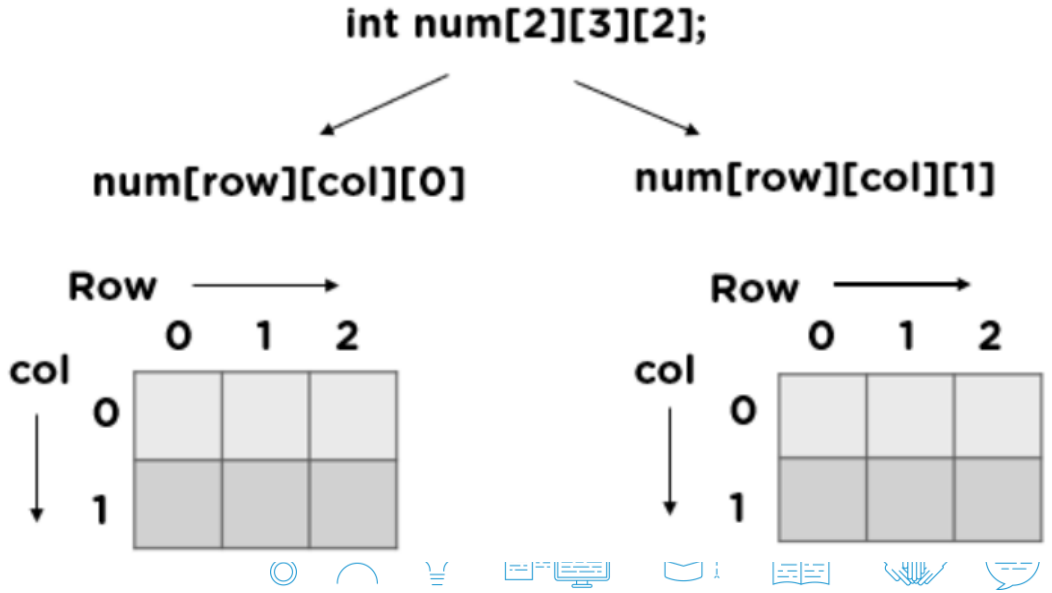


- Mảng nhiều chiều

Mảng hai chiều



Mảng ba chiều

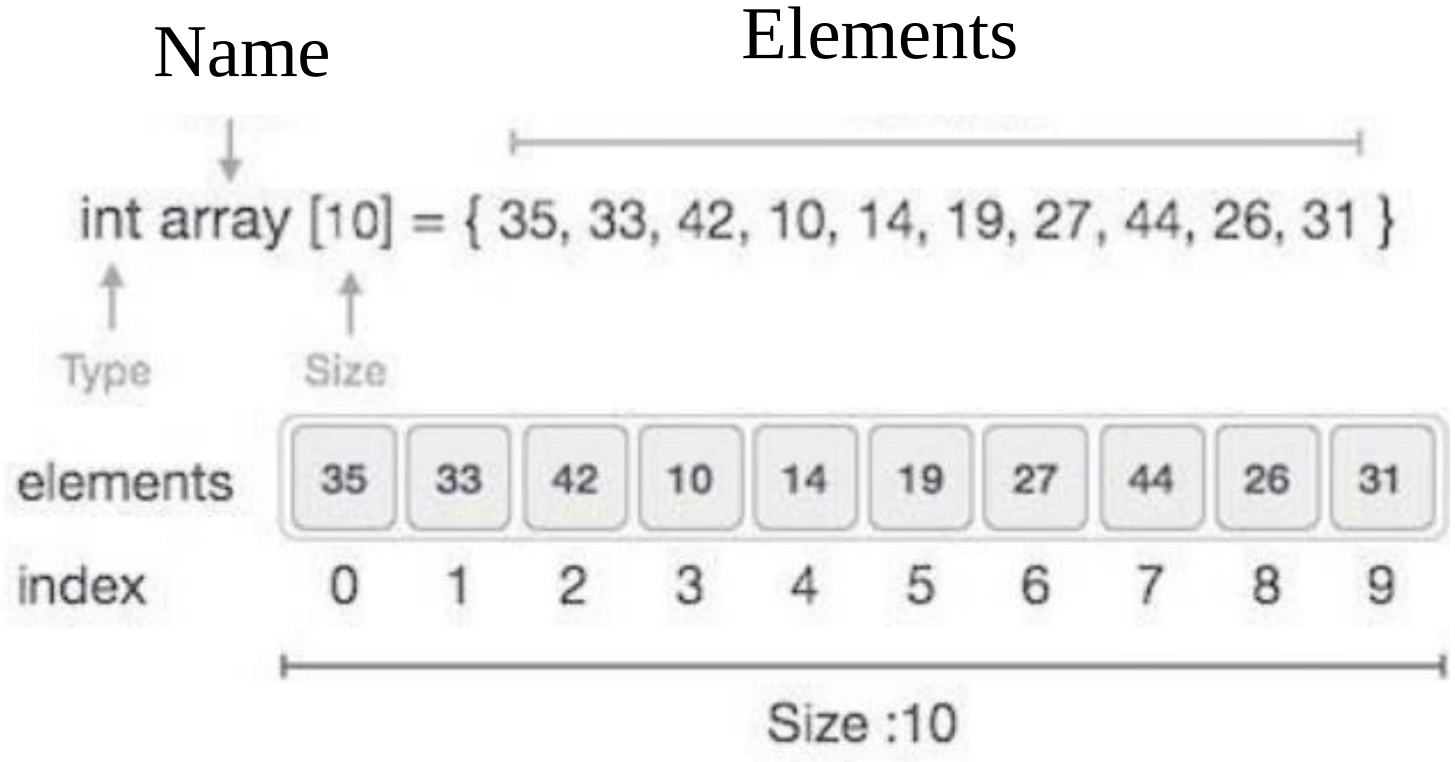


Biểu diễn mảng

Mảng có thể được khai báo theo nhiều cách trong các ngôn ngữ khác nhau:

Ví dụ:

- Chỉ số bắt đầu bởi 0
- Độ dài mảng là 10, tức là nó có thể lưu 10 phần tử
- Mỗi phần tử được truy cập qua chỉ số của nó. Ví dụ, phần tử có chỉ số 6 là 27.



Các phép toán cơ bản

Các phép toán cơ bản được hỗ trợ bởi mảng:

- ***Duyệt mảng (Traverse)*** – in ra tất cả các phần tử, mỗi phần tử một lần
- ***Chèn phần tử (Insertion)*** – Thêm một phần tử với chỉ số đã cho.
- ***Xóa phần tử (Deletion)*** – Xóa phần tử với chỉ số đã cho.
- ***Tìm kiếm (Search)*** – Tìm một phần tử với chỉ số hoặc giá trị cho trước.
- ***Cập nhật (Update)*** – Cập nhật phần tử tại chỉ số cho trước.

Cài đặt mảng

Khai báo mảng trong Java:

- Cách 1:

ElementType[ ] *arrayName* = {$initialValue_0$, $initialValue_1$, ..., $initialValue_{N-1}$};

Trong đó:

- *ElementType* có thể là kiểu dữ liệu cơ bản hoặc tên lớp Java và
- *arrayName* có thể là tên biến Java hợp lệ.
- $initialValue_i$: các giá trị ban đầu phải cùng kiểu dữ liệu của phần tử của mảng.

Ví dụ: **int**[ ] primes = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};

Cài đặt mảng

- Cách 2:

ElementType[ ] *arrayName* = **new** *ElementType*[length] ;

Trong đó:

- *ElementType:* có thể là kiểu dữ liệu cơ bản hoặc tên lớp Java và
- *arrayName:* là tên biến Java hợp lệ.
- *length:* là số nguyên dương biểu thị độ dài của mảng mới
- **new**: trả về một tham chiếu đến mảng mới và thông thường nó sẽ được gán cho một biến mảng.

Ví dụ: **double**[ ] measurements = **new double**[1000];

(khai báo một biến mảng *measurements* và gán cho nó một mảng mới gồm 1000 phần tử)

CMC UNIVERSITY

Cài đặt các phép toán cơ bản

Ví dụ: Tạo mảng các số nguyên có tên *a*, truy cập phần tử của mảng sử dụng chỉ số

```
int[] a = {10, 20, 30, 40, 50};
System.out.println("Các phần tử trong mảng:");
for (int i = 0; i < a.length; i++) {
   System.out.println("Phần tử tại chỉ số " + i + ": " + a[i]);
}
```

```
Scanner scanner = new Scanner(System.in);
System.out.print("Nhập chỉ số của phần tử cần truy cập: ");
int p = scanner.nextInt();
if (p >= 0 && p < a.length) {
    System.out.println("Phần tử tại chỉ số " + p + " " + a[p]);
} else {
        System.out.println("Chỉ số không hợp lệ ");
        }
```

CMC UNIVERSITY

Cài đặt các phép toán cơ bản

Ví dụ: chèn một phần tử có chỉ số p vào mảng *a*

```
if (p < 0 || p > a.length) {
    System.out.println("Chỉ số không hợp lệ");
} else {
    // Tạo mảng mới với kích thước lớn hơn 1
    int[] newA = new int[a.length + 1];
    // Sao chép các phần tử trước vị trí chèn
    for (int i = 0; i < p; i++) {
        newA[i] = a[i];
    }
    // Chèn phần tử mới
    newA[p] = value;
    // Sao chép các phần tử sau vị trí chèn
    for (int i = p; i < a.length; i++) {
        newA[i + 1] = a[i];
    }
    // In mảng sau khi chèn
    System.out.println("Mảng mới: " + Arrays.toString(newA));
}
```

Cài đặt các phép toán cơ bản

Ví dụ: Xóa phần tử có chỉ số p khỏi mảng và sắp xếp lại tất cả các phần tử của một mảng

```
if (p < 0 || p >= a.length) {
    System.out.println("Chỉ số không hợp lệ");
} else {
    // Tạo mảng mới với kích thước nhỏ hơn 1
    int[] newA = new int[a.length - 1];
    // Sao chép các phần tử trước và sau vị trí xóa
    for (int i = 0, j = 0; i < a.length; i++) {
        if (i != p) {
            newA[j++] = a[i];
        }
    }
    // In mảng sau khi xóa
    System.out.println("Mảng mới: " + Arrays.toString(newA));
}
```

Cài đặt các phép toán cơ bản

Ví dụ: cập nhật giá trị cho một phần tử có chỉ số p.

```
if (p < 0 || p >= a.length) {
            System.out.println("Chỉ số không hợp lệ ");
} else {
    // Nhập giá trị mới cho phần tử
    System.out.print("Nhập giá trị mới của phần tử: ");
    int newValue = scanner.nextInt();
    // Cập nhật giá trị cho phần tử tại chỉ số p
            a[p] = newValue;
    // In mảng sau khi cập nhật
    System.out.println("Mảng cập nhật: " + Arrays.toString(a));
        }
```

Các phép toán cơ bản

- Việc lưu trữ, duyệt và truy cập các phần tử mảng rất nhanh so với danh sách vì các phần tử có thể được truy cập ngẫu nhiên bằng cách sử dụng vị trí là các chỉ mục .
- Nếu dữ liệu được lưu trữ trong mảng lớn và hệ thống có bộ nhớ thấp thì cấu trúc dữ liệu mảng sẽ không phải là lựa chọn tốt để lưu trữ dữ liệu vì khó phân bổ một khối lớn vị trí bộ nhớ. Cấu trúc dữ liệu mảng còn có hạn chế về kích thước tĩnh phải được khai báo tại thời điểm tạo mảng.

CMC UNIVERSITY

Bài tập

- Tạo một mảng số nguyên với số phần tử được người dùng nhập vào.
- Chèn một phần tử có giá trị và vị trí được chỉ định vào mảng.
- Xóa một phần tử tại vị trí đã chỉ định khỏi mảng
- Xóa phần tử có giá trị được chỉ định khỏi mảng
- Xác định vị trí của phần tử với giá trị được chỉ định trong mảng

CMC UNIVERSITY

DANH SÁCH

Các khái niệm cơ bản

Danh sách (Lists)

- **Danh sách** là một tập hợp có thứ tự của một số biến động các phần tử. Một danh sách mà quan hệ lân cận giữa các phần tử được hiển thị ra thì gọi là danh sách tuyến tính (*linear list*).
- Ví dụ: tập hợp những người đến mua vé tàu là hình ảnh của một danh sách. Họ sẽ mua vé theo thứ tự xếp hàng.

(Vector là một trường hợp đặc biệt của danh sách tuyến tính, là hình ảnh của một danh sách tuyến tính tại một thời điểm)

- Danh sách tuyến tính là một danh sách hoặc rỗng (không có phần từ) hoặc có dạng $(a_1, a_2, ..., a_n)$, trong đó a_i là dữ liệu *nguyên tử*, a_1 là phần tử đầu tiên và a_n là phần tử cuối cùng. Với mỗi phần tử a_i, a_{i+1} là *phần tử sau*, a_{i-1} là *phần tử trước*, n là độ dài (kích cỡ) của danh sách, và giá trị của n có thể thay đổi.
- Mỗi phần tử trong danh sách thường là một bản ghi (gồm một hoặc nhiều trường).

Các khái niệm cơ bản

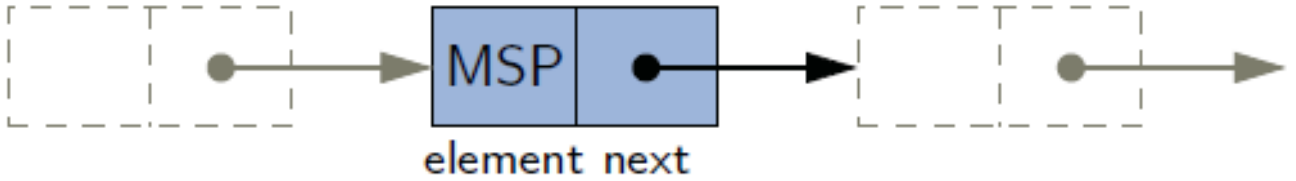
Ví dụ: danh mục điện thoại là một danh sách tuyến tính, mỗi phần tử ứng với một đơn vị thuê bao gồm các trường: tên đơn vị thuê bao, địa chỉ, số điện thoại.

Các phép toán cơ bản:

- Truy cập giá trị các phần tử trong danh sách
- Chèn phần tử vào danh sách
- Xóa phần tử khỏi danh sách
- Ghép hai hoặc nhiều danh sách
- Tách một danh sách thành nhiều danh sách
- Sao chép danh sách
- Cập nhật danh sách

Danh sách liên kết

- Danh sách liên kết (*linked list*), ở dạng đơn giản nhất, là một tập hợp các *nút* tạo thành một chuỗi tuyến tính.
- Trong danh sách liên kết đơn, mỗi nút lưu trữ một giá trị và tham chiếu đến một đối tượng là một phần tử của chuỗi, cũng như tham chiếu đến nút tiếp theo của danh sách (như hình dưới).

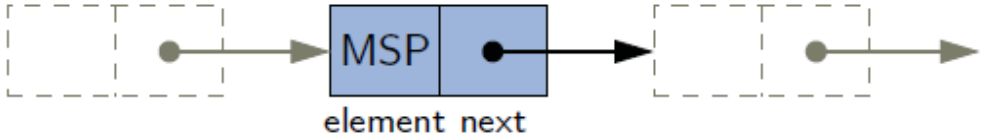


- Độ dài của danh sách có thể tăng hoặc giảm khi thực hiện chương trình
- Có nhiều loại danh sách liên kết: *danh sách liên kết đơn* (singly), *danh sách liến kết kép* (doubly), và *danh sách liên kết vòng* (circular linked list).

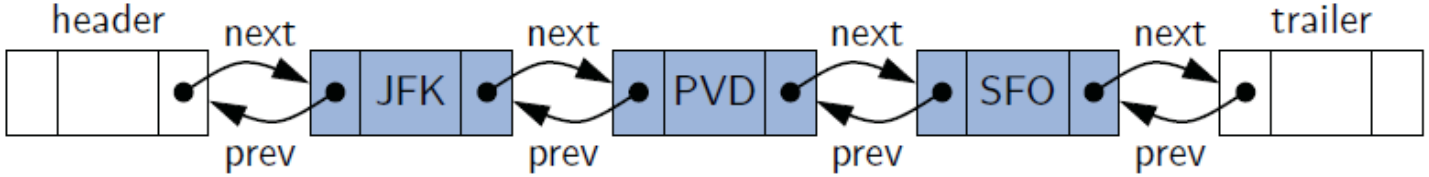
Các loại danh sách liên kết

Danh sách liên kết đơn:

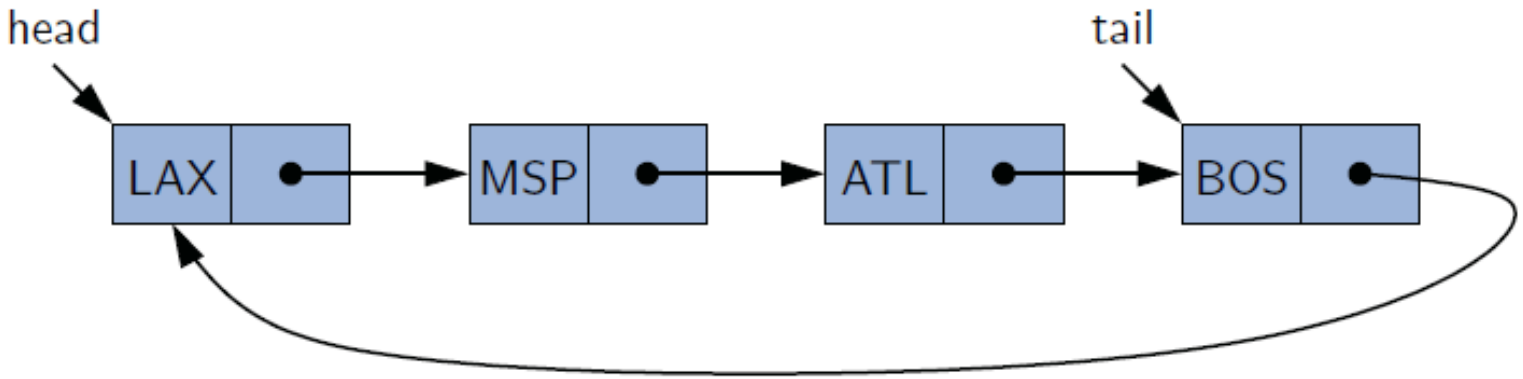
Mỗi nút chứa dữ liệu và địa chỉ (liên kết) đến nút tiếp theo.



Danh sách liên kết kép: bổ sung một tham chiếu đến nút trước cho mỗi nút → forward hoặc backward

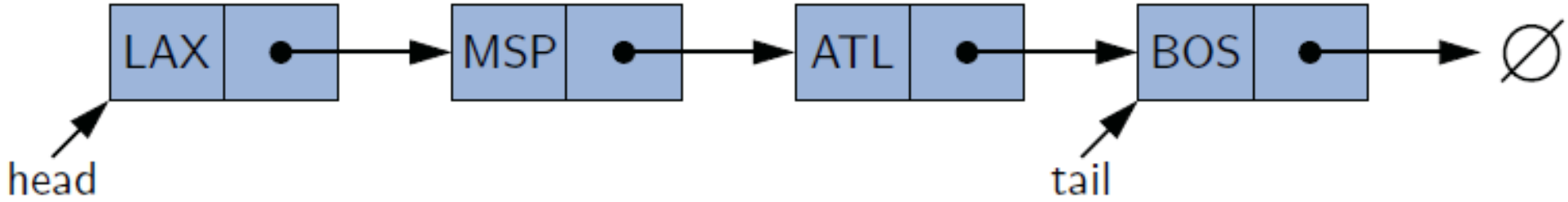


Danh sách liên kết vòng: là một biến thể của danh sách liên kết trong đó phần tử cuối cùng được liên kết với phần tử đầu tiên, tạo thành một vòng tròn.



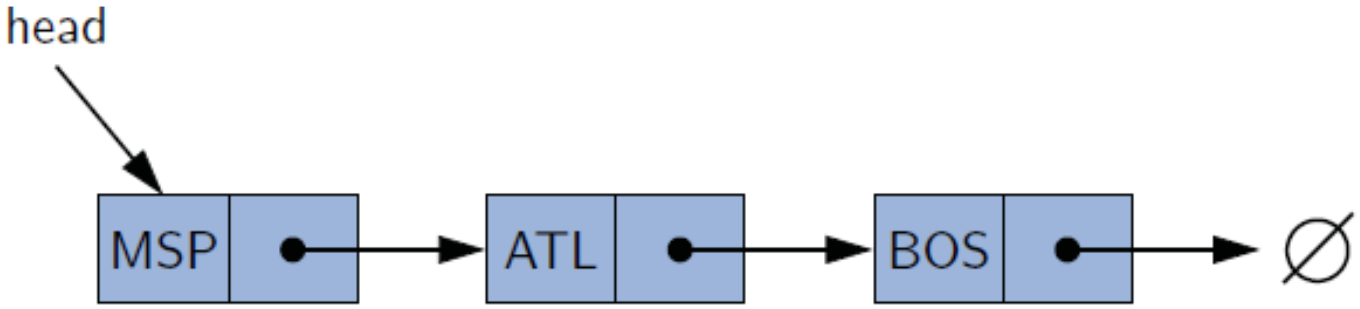
Biểu diễn danh sách liên kết

- Để truy cập các phần tử của danh sách liên kết, cần phải giữ tham chiếu đến nút đầu tiên của danh sách, được gọi là **đầu** (head)
- Có thể tìm phần tử cuối, hay gọi là **đuôi** (tail) của danh sách bằng cách duyệt bắt đầu từ nút đầu và di chuyển từ nút này sang nút khác theo tham chiếu tiếp theo của mỗi nút. Nút đuôi là nút có tham chiếu tiếp theo là *null*.



Biểu diễn danh sách liên kết

Ví dụ về danh sách liên kết đơn có các phần tử là chuỗi biểu thị mã sân bay. Danh sách có tham chiếu head đến nút đầu và nút đuôi tail đề cập đến nút cuối cùng của danh sách. Giá trị null được ký hiệu là Ø.




```
import java.util.LinkedList;
```

```
LinkedList<String> linkedList = new LinkedList<>();
// Thêm các phần tử vào LinkedList
linkedList.add("MSP");
linkedList.add("ATL");
linkedList.add("BOS");
// In LinkedList
System.out.println("LinkedList ban đầu: " + linkedList);
```

CMC UNIVERSITY

Biểu diễn danh sách liên kết

Để tổ chức danh sách móc nối, cần có những khả năng sau:

- Tồn tại những phương tiện để chia bộ nhớ thành các nút, tại mỗi nút có thể truy cập vào từng trường (ta chỉ xét trường hợp nút và trường có kích thước ấn định)
- Tồn tại một cơ chế để xác định được một nút đang sử dụng (gọi là nút bận) hoặc không sử dụng (gọi là nút trống)
- Tồn tại một cơ cấu như một kho chứa chỗ trống để cung cấp các nút trống khi có yêu cầu sử dụng và thu hồi lại các nút đó khi không cần dùng nữa.

CMC UNIVERSITY

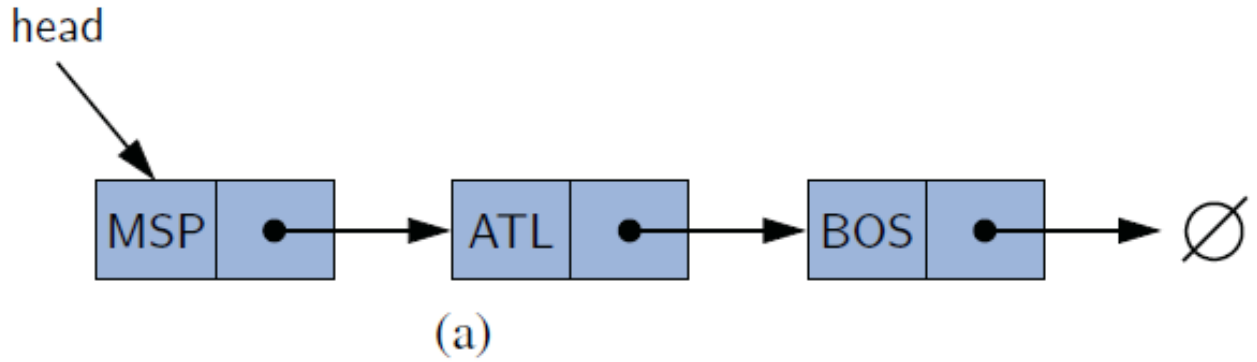
Các phép toán cơ bản

- ***Duyệt (Traversal)*** – truy cập mỗi phần tử của danh sách liên kết
- ***Chèn (Insertion)*** – thêm một phần tử mới vào danh sách liên kết
- ***Xóa (Deletion)*** – xóa phần tử khỏi danh sách
- ***Tìm kiếm (Search)*** – tìm một phần tử trong danh sách liên kết
- ***Sắp xếp (Sort)*** – sắp xếp các phần tử trong danh sách

Chèn phần tử vào danh sách

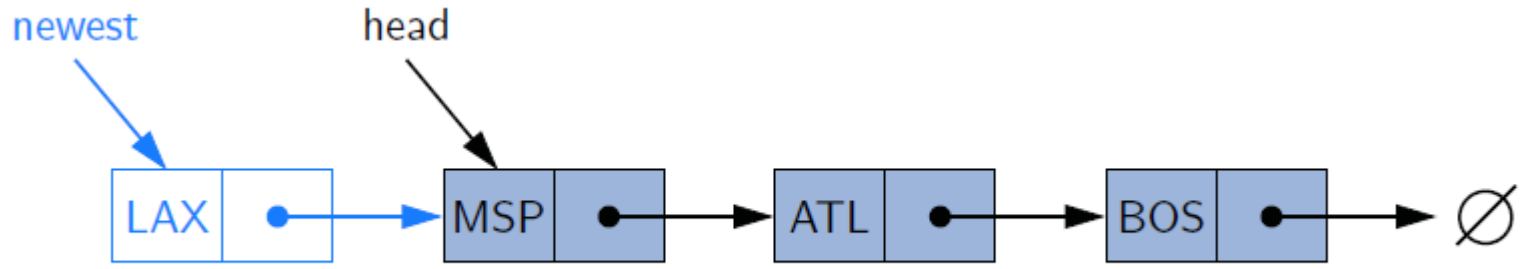
Chèn phần tử vào đầu danh sách

Ví dụ: chèn phần tử LAX vào đầu danh sách ở hình (a):



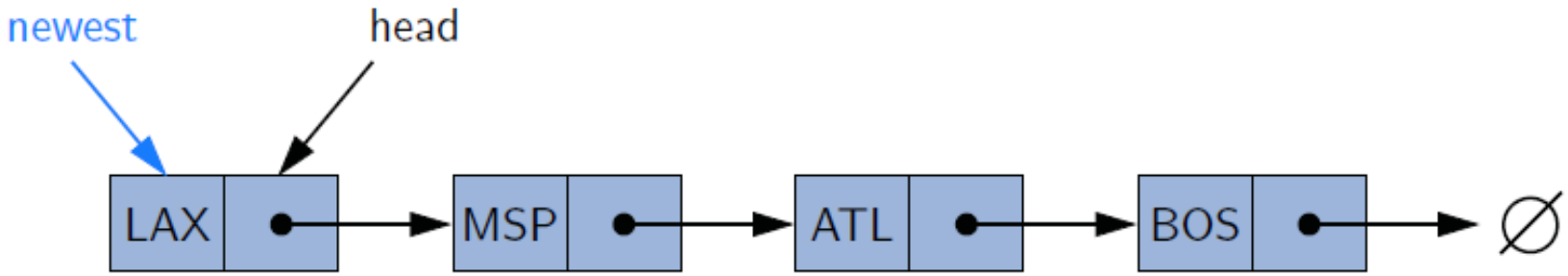
(a)

- Tạo nút mới newest có giá trị LAX, liên kết với nút đầu head



(b)

- Gán lại tham chiếu head là nút newest.



(c)

Chèn phần tử vào danh sách

Chèn phần tử vào đầu danh sách

Ví dụ: chèn phần tử LAX vào đầu danh sách ở hình (a):

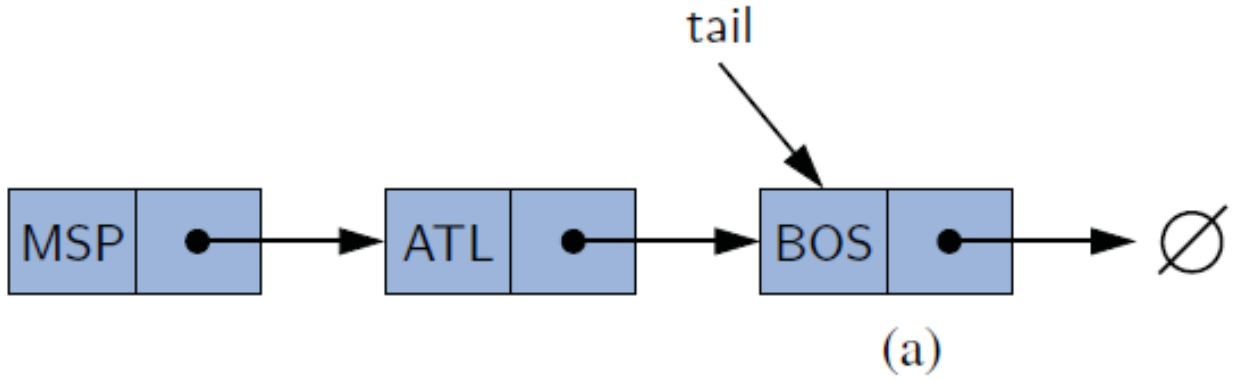
Algorithm addFirst(e):

```
newest = Node(e)   {create new node instance storing reference to element e}
newest.next = head   {set new node's next to reference the old head node}
head = newest   {set variable head to reference the new node}
size = size + 1   {increment the node count}
```

Chèn phần tử vào danh sách

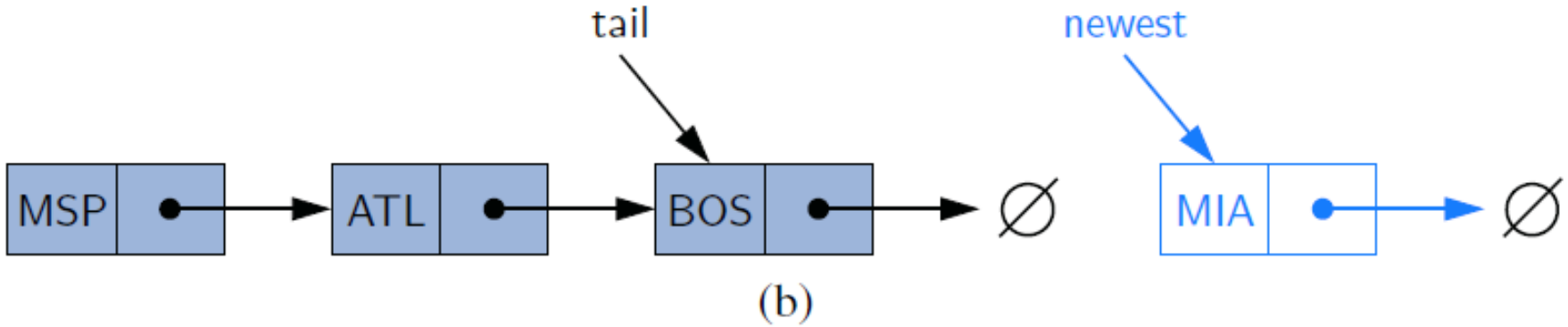
Chèn phần tử vào cuối danh sách

Ví dụ: chèn phần tử MIA vào cuối danh sách ở hình (a):



(a)

- Tạo nút mới newest có giá trị MIA

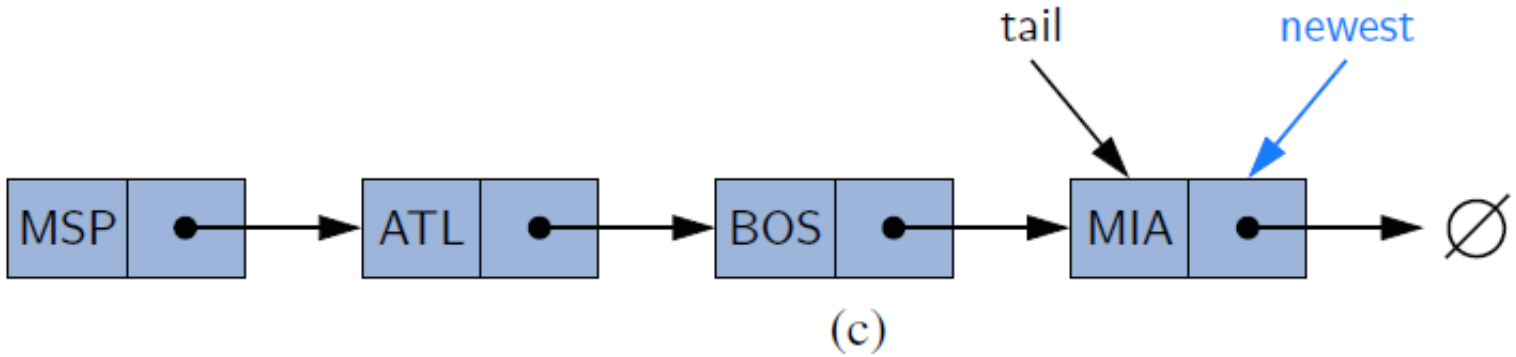


(b)

- Gán lại tham chiếu:

tail.next = newest;

tail = newest;



(c)

Chèn phần tử vào danh sách

Chèn phần tử vào cuối danh sách

Algorithm addLast(e):

```
newest = Node(e)      {create new node instance storing reference to element e}
newest.next = null    {set new node's next to reference the null object}
tail.next = newest    {make old tail node point to new node}
tail = newest         {set variable tail to reference the new node}
size = size + 1       {increment the node count}
```

Chèn phần tử vào danh sách

Chèn phần tử vào giữa danh sách

Ví dụ: chèn phần tử MON vào giữa danh sách ở hình (a): duyệt danh sách để đến vị trí cần chèn,

được tham chiếu bởi

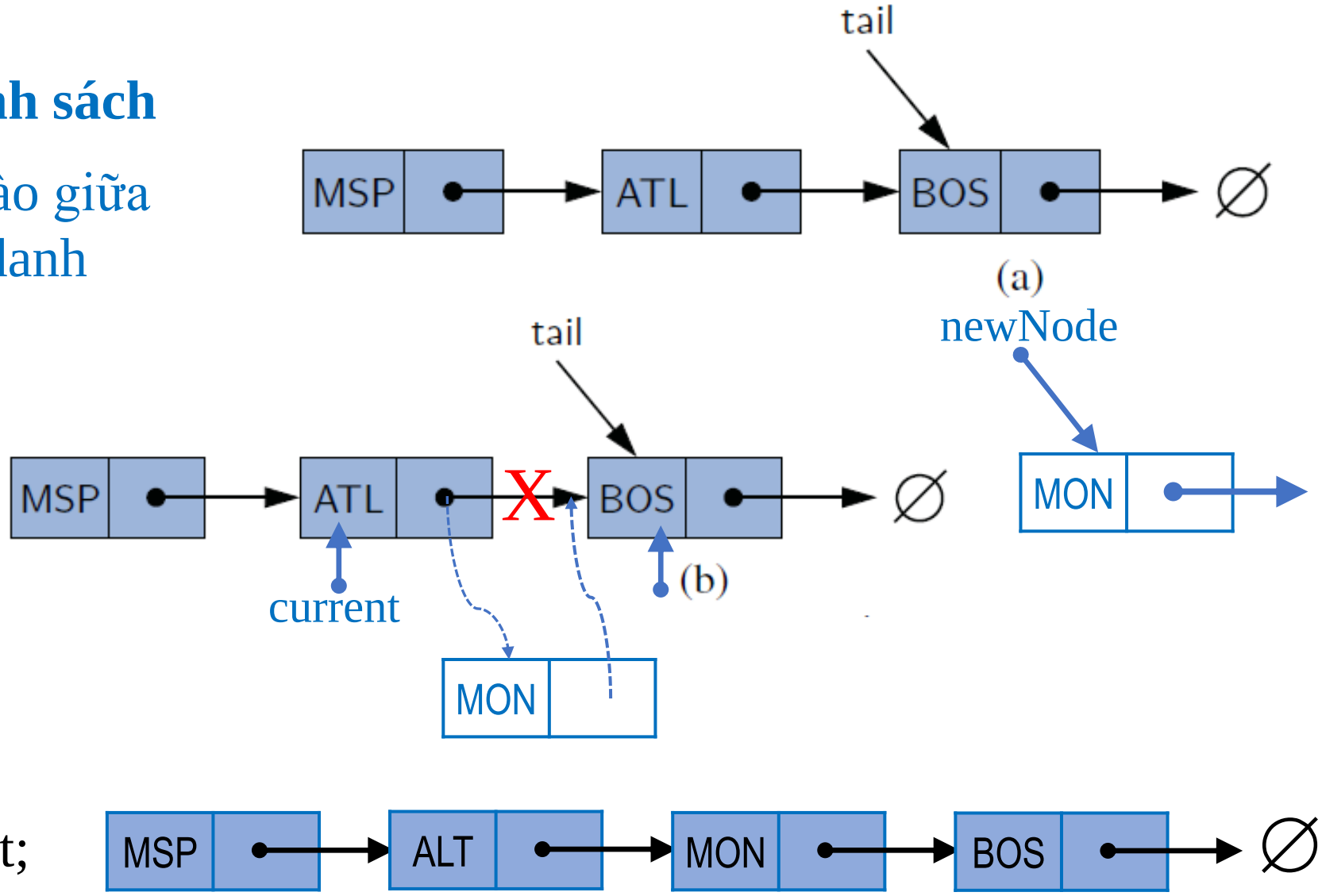
current

- Tạo nút mới newNode có giá trị MON

- Gán lại tham chiếu:

newNode.next = current.next;

current.next = newNode;



CMC UNIVERSITY

Chèn phần tử vào danh sách

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

```
class MyLinkedList {
    Node head;
    // Thêm phần tử vào đầu danh sách
    public void addFirst(String data) {
        Node newNode = new Node(data);
        newNode.next = head;
        head = newNode;
    }
```

```
// Thêm phần tử vào cuối danh sách
public void addLast(String 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;
    }
}
```

Chèn phần tử vào danh sách

```
public void addMidd(int p, String data) {
        Node newNode = new Node(data);
        if (p == 0) {
            newNode.next = head;
            head = newNode;
            return;
        }
        Node current = head;
        for (int i = 0; i < p - 1; i++) {
            if (current == null || current.next == null) {
                throw new IndexOutOfBoundsException("Chỉ số
                    vượt quá giới hạn của danh sách");
            }
            current = current.next;
        }
        newNode.next = current.next;
        current.next = newNode;
    }
```

Chèn phần tử vào danh sách

Tạo danh sách

```
public static void main(String[] args) {
        MyLinkedList linkedList = new MyLinkedList();
        // Thêm các phần tử vào danh sách liên kết
        linkedList.addLast("MSP");
        linkedList.addLast("ALT");
        linkedList.addLast("BOS");
        // In danh sách liên kết
        System.out.println("Danh sách liên kết:");
        linkedList.printList();
    }
```

Chèn phần tử vào danh sách

- Độ phức tạp thời gian trong trường hợp tồi nhất của thao tác chèn là O(1) khi chúng ta có thêm một con trỏ trỏ đến nút cuối cùng.
- Mặt khác, khi không có liên kết đến nút cuối cùng, độ phức tạp thời gian sẽ là O(n) vì phải duyệt qua danh sách để đến vị trí mong muốn và trong trường hợp tồi nhất, có thể phải duyệt qua tất cả n nút trong danh sách.

Duyệt danh sách

Ví dụ: Duyệt danh sách gồm các phần tử đã tạo ở ví dụ trên

```
// Duyệt danh sách và in các phần tử
    public void traverse() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " ");
            current = current.next;
        }
        System.out.println();
    }
```

CMC UNIVERSITY

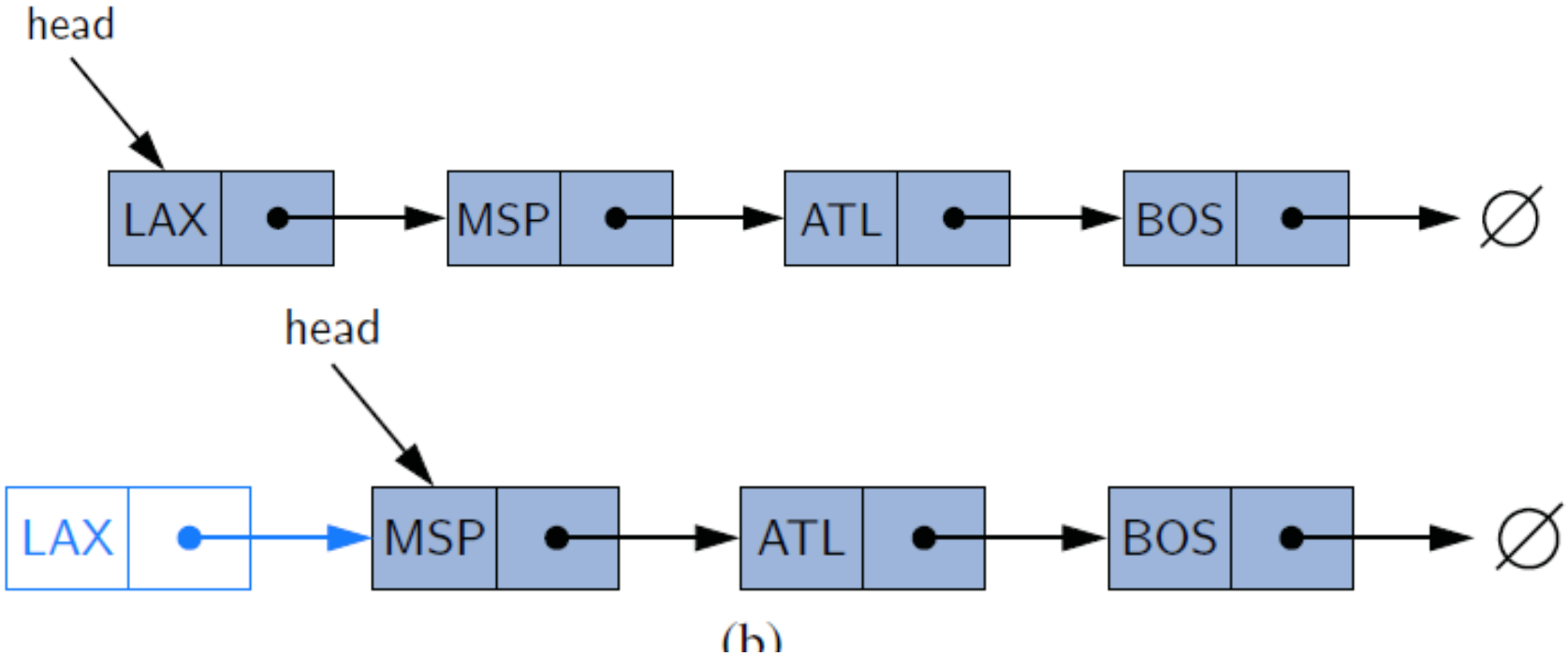
Tìm kiếm phần tử

Ví dụ: Tìm phần tử có dữ liệu data cho trước trong danh sách, nếu tìm thấy trả ra kết quả true, ngược lại trả ra false

```
// Tìm kiếm phần tử trong danh sách
    public boolean search(String data) {
        Node current = head;
        while (current != null) {
            if (current.data.equals(data)) {
                return true; // Tìm thấy phần tử
            }
            current = current.next;
        }
        return false; // Không tìm thấy phần tử
    }
```

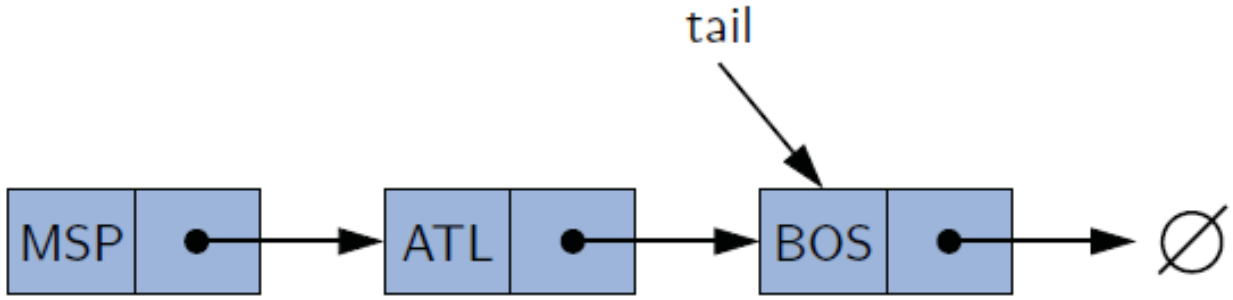
Xóa phần tử

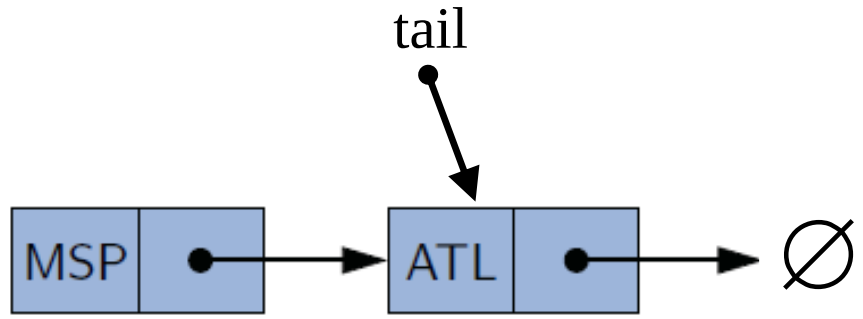
Xóa phần tử đầu danh sách



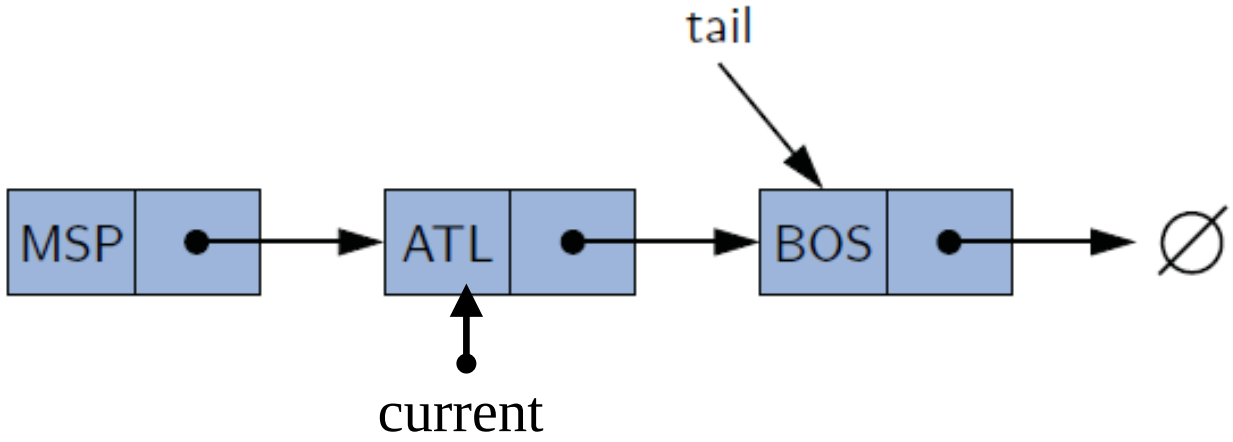
Xóa phần tử

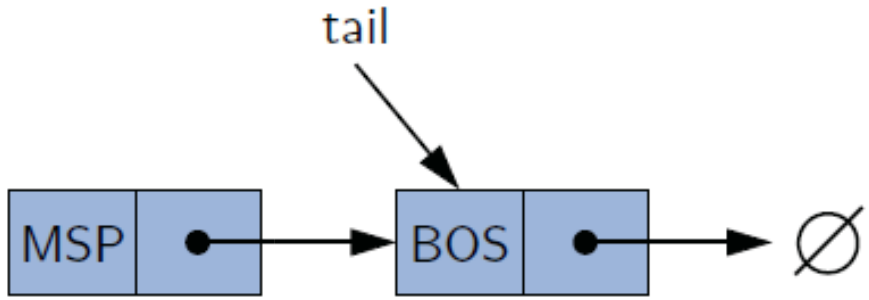
Xóa phần tử cuối danh sách





Xóa phần tử giữa danh sách





CMC UNIVERSITY

Xóa phần tử

Ví dụ: Xóa phần tử đầu danh sách

```
// Xóa phần tử đầu tiên
    public boolean removeFirst() {
        if (head == null) {
            return false; // Danh sách trống
        }
        head = head.next; // Cập nhật head để bỏ qua phần tử
            đầu tiên
        return true;
    }
```

Xóa phần tử

Ví dụ: Xóa phần tử cuối danh sách

```
// Xóa phần tử cuối cùng
public boolean removeLast() {
    if (head == null) {
        return false; // Danh sách trống
    }
    if (head.next == null) {
        head = null; // Danh sách chỉ có một phần tử
        return true;
    }
    Node current = head;
    while (current.next.next != null) {
        current = current.next;
    }
    current.next = null; // Bỏ qua phần tử cuối cùng
    return true;
}
```

Xóa phần tử

Ví dụ: Xóa phần tử có dữ liệu là data

```
public boolean removeData(String data) {
    if (head == null) {
        return false; // Danh sách trống
    }
    if (head.data.equals(data)) {
        head = head.next; // Xóa phần tử đầu tiên
        return true;
    }
    Node current = head;
    while (current.next != null && !current.next.data
        .equals(data)) {
        current = current.next;
    }
    if (current.next == null) {
        return false; // Không tìm thấy phần tử
    }
    current.next = current.next.next; // Bỏ qua ptu cần xóa
    return true;
}
```

CMC UNIVERSITY

Ứng dụng của danh sách liên kết

- Cấp phát bộ nhớ động
- Thực thi trong stack và queue
- Trong chức năng undo của phần mềm
- Bảng băm, đồ thị

CMC UNIVERSITY

NGĂN XẾP

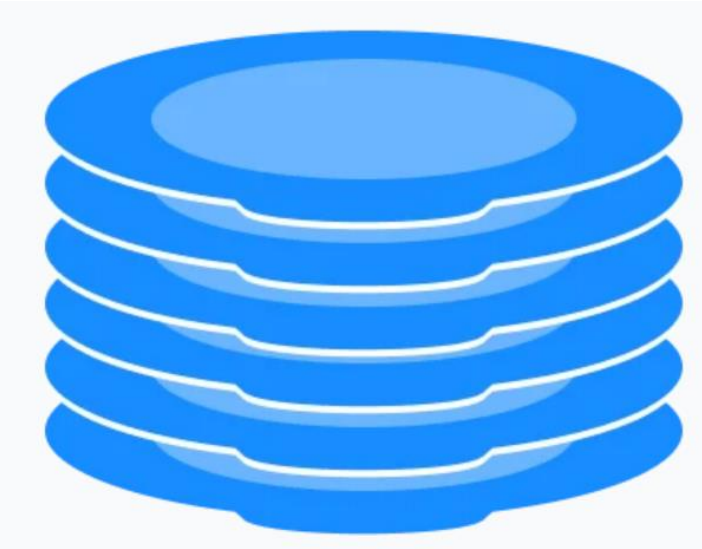
Ngăn xếp

- Ngăn xếp (stack) là một cấu trúc dữ liệu tuyến tính hoạt động theo nguyên tắc vào sau ra trước (**Last In First Out - LIFO**). Tức là phần tử được chèn vào sau cùng sẽ bị xóa đầu tiên
- Cấu trúc dữ liệu ngăn xếp có thể hình dung như các đĩa chồng lên nhau

Ngăn xếp lưu trữ dữ liệu theo một thứ tự cụ thể, và có các ràng buộc:

- Các phần tử dữ liệu trong stack chỉ được chèn vào cuối (push)
- Các phần tử dữ liệu trong stack chỉ được xóa từ cuối (pop)
- Chỉ phần tử dữ liệu cuối cùng có thể được đọc từ stack (peek)

Nếu muốn lấy đĩa ở dưới cùng thì phải lấy ra tất cả các đĩa ở trên

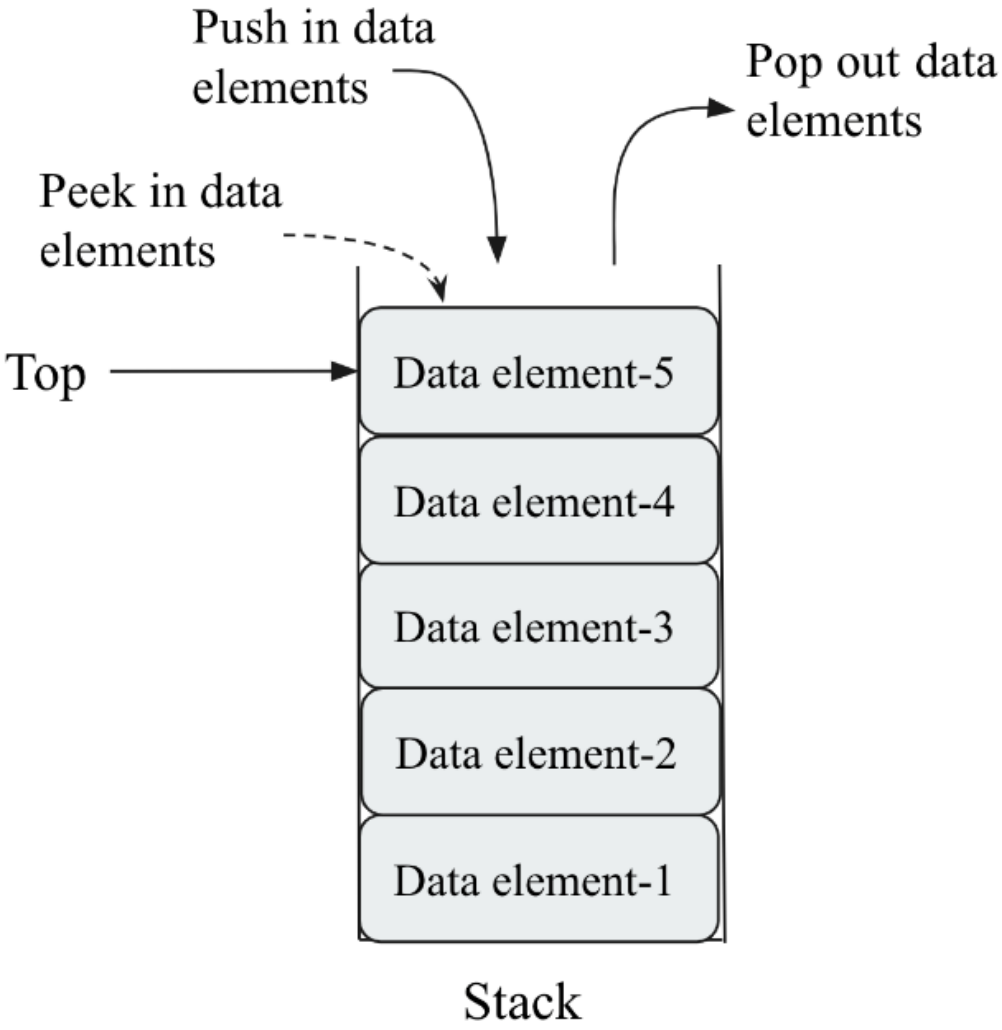


Stack representation similar to a pile of plate

Ngăn xếp

Nguyên tắc LIFO của Stack

- Trong lập trình, đẩy một phần tử và đỉnh (top) của stack được gọi là **push** và lấy phân tử khỏi stack gọi là **pop**.
- Tất cả các phép toán trong stack được thực hiện thông qua inter, which is generally named *Top*



Stack Push and Pop Operations

CMC UNIVERSITY

Các phép toán cơ bản trên Stack

- **push(e)**: Thêm phần tử e vào đỉnh stack
- **pop()**: Xóa phần tử ở đỉnh của stack và trả về phần tử đỉnh của stack (hoặc null nếu stack rỗng)
- **size**(): trả ra số phần tử của stack
- **isEmpty**(): trả ra giá trị Boolean chỉ ra stack có rỗng không
- **isFull**(): trả ra giá trị Boolean chỉ ra stack có đầy không
- **peek**: trả ra phần tử ở đỉnh stack nhưng không xóa nó (hoặc trả ra null nếu stack rỗng)

CMC UNIVERSITY

Các phép toán cơ bản trên Stack

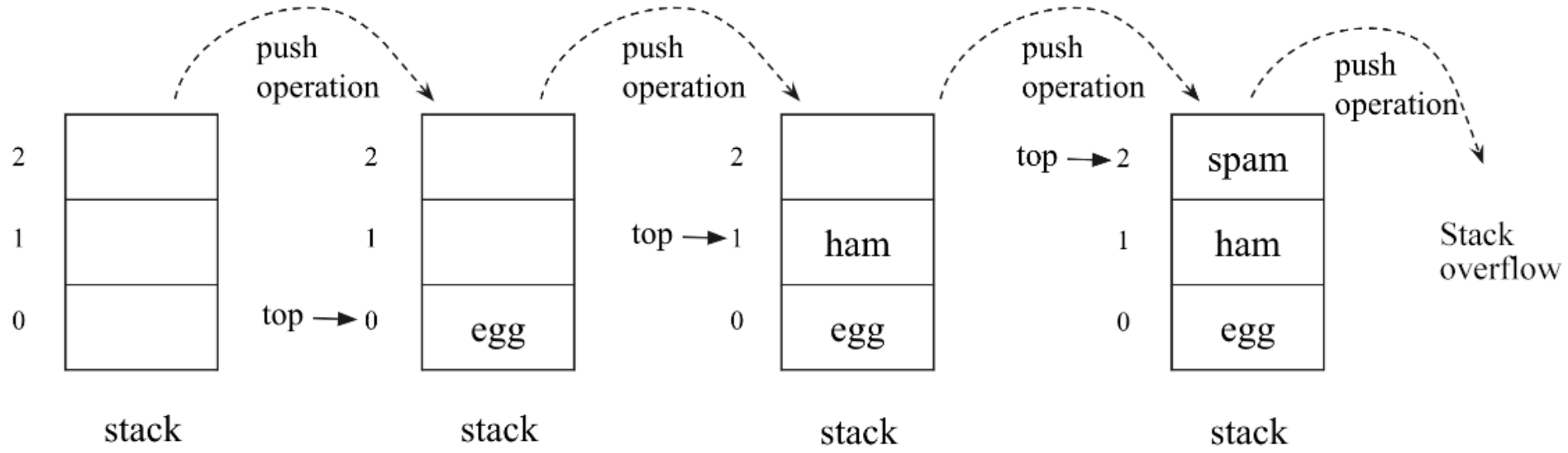
Bảng sau minh họa việc sử dụng các phép toán trong stack:

Stack operation	Size	Contents	Operation results
`stack()`	0	`[]`	Stack object created, which is empty.
`push "egg"`	1	`['egg']`	One item egg is added to the stack.
`push "ham"`	2	`['egg', 'ham']`	One more item, ham, is added to the stack.
`peek()`	2	`['egg', 'ham']`	The top element, ham, is returned.
`pop()`	1	`['egg']`	The ham item is popped off and returned. (This item was added last, so it is removed first.)
`pop()`	0	`[]`	The egg item is popped off and returned. (This is the first item added, so it is returned last.)

Lưu trữ stack bằng mảng

Khi lưu trữ stack bằng mảng (kích thước stack là cố định), cần phải kiểm tra stack đầy khi thực hiện thao tác push, vì push phần tử vào stack đầy sẽ tạo ra lỗi overflow. Tương tự, khi thực hiện pop với stack rỗng sẽ dẫn đến lỗi underflow.

- Hình sau minh họa phép toán push trên stack sử dụng mảng



CMC UNIVERSITY

Lưu trữ stack bằng mảng

```
public class ArrayStack {
    private int maxSize;
    private String[] stackArray;
    private int top;

    // Constructor để khởi tạo stack với kích thước cho trước
    public ArrayStack(int size) {
        this.maxSize = size;
        this.stackArray = new String[maxSize];
        this.top = -1; // stack rỗng
    }
```

CMC UNIVERSITY

Lưu trữ stack bằng mảng

```
// Thêm phần tử vào stack
public void push(String value) {
    if (isFull()) {
        System.out.println("Stack đầy");
    } else {
        stackArray[++top] = value;
    }
}
```

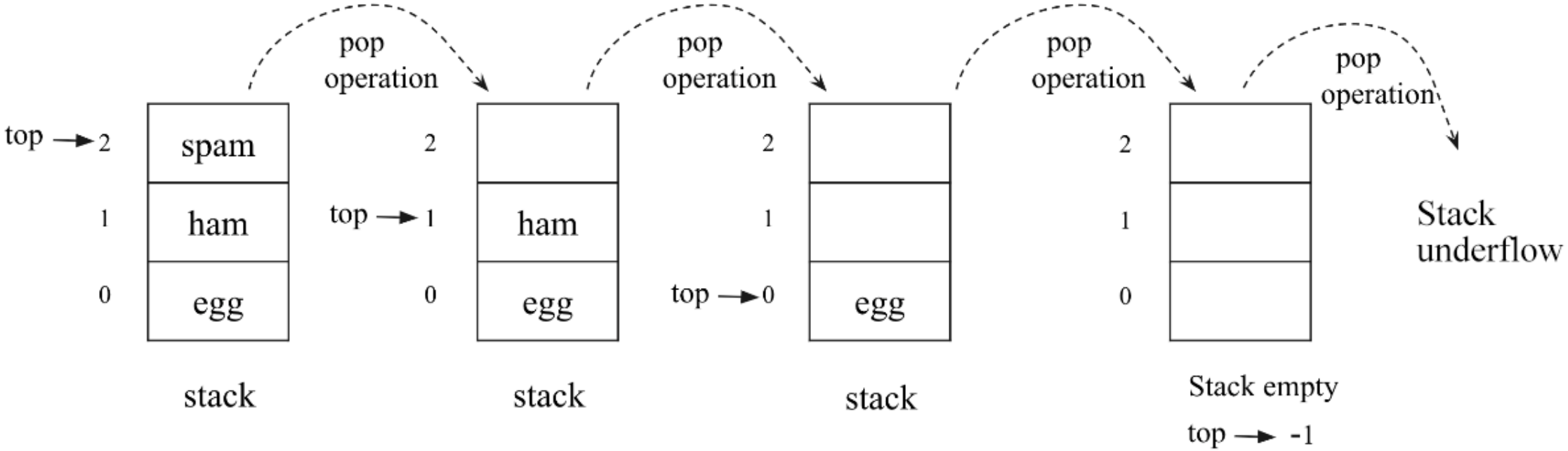
```
ArrayStack stack = new ArrayStack(10);
        // Thêm các phần tử vào stack
        stack.push("egg");
        stack.push("ham");
        stack.push("spam");
        // In stack
        System.out.println("Stack ban đầu:");
        stack.printStack();
```

The output:

['egg', 'ham', 'spam']
Stack Overflow
Stack Overflow

Lưu trữ stack bằng mảng

- Minh họa phép toán pop trên stack sử dụng mảng:



CMC UNIVERSITY

Lưu trữ stack bằng mảng

```
// Lấy và xóa phần tử đầu tiên khỏi stack
public String pop() {
    if (isEmpty()) {
        System.out.println("Stack rỗng");
        return null; // Trả về giá trị null để biểu thị lỗi
    } else {
        return stackArray[top--];
    }
}
```

```
String topElement = stack.pop();
topElement = stack.pop();
topElement = stack.pop();
topElement = stack.pop();
System.out.println("Stack sau khi xóa cuối cùng:");
stack.printStack();
```

Lưu trữ stack bằng mảng

- Minh họa phép toán peek trên stack sử dụng mảng:

```
// Lấy phần tử đầu tiên của stack mà không xóa nó
public String peek() {
    if (isEmpty()) {
        System.out.println("Stack rỗng");
        return null; // Trả về giá trị null để biểu thị lỗi
    } else {
        return stackArray[top];
    }
}
```

CMC UNIVERSITY

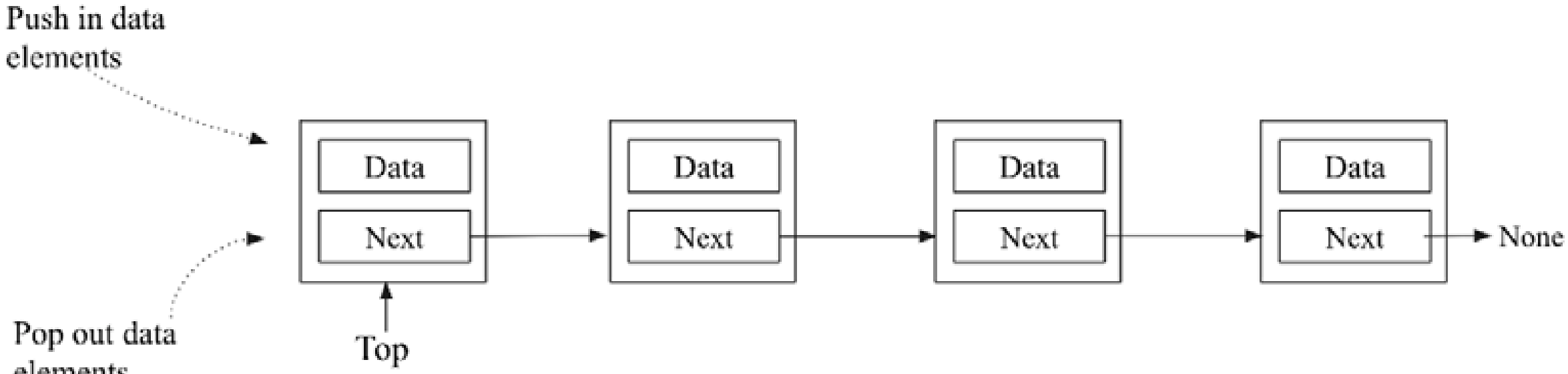
Lưu trữ stack bằng mảng

- Kiểm tra stack rỗng, đầy, tính số phần tử của stack

```
// Kiểm tra xem stack có rỗng không
public boolean isEmpty() {
    return (top == -1);
}
// Kiểm tra xem stack có đầy không
public boolean isFull() {
    return (top == maxSize - 1);
}
// Kích thước của stack
public int size() {
    return top + 1;
}
```

Lưu trữ stack bằng danh sách liên kết

Việc lưu trữ cấu trúc dữ liệu ngăn xếp bằng danh sách liên kết có thể được coi là danh sách liên kết chuẩn với một số ràng buộc, bao gồm các phần tử có thể được thêm hoặc xóa khỏi cuối danh sách (thao tác push và pop) thông qua tham chiếu top.



Stack using linked list

Lưu trữ stack bằng danh sách liên kết

Để triển khai một stack, cần các thông tin sau:

1. Nút ở đỉnh của stack, thực hiện push và pop thông qua nút này
2. Số lượng nút trong stack, bổ sung biến kích thước vào lớp Stack

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

```
public class LinkedListStack {
    private Node top;
    private int size;

    // Constructor để khởi tạo stack
    public LinkedListStack() {
        this.top = null;
        this.size = 0;
    }
```

Lưu trữ stack bằng danh sách liên kết

Push

Kiểm tra xem stack đã có một số mục chưa hay có rỗng không. Không cần kiểm tra điều kiện overflow vì độ dài của stack không cố định, không như cài đặt stack bằng mảng.

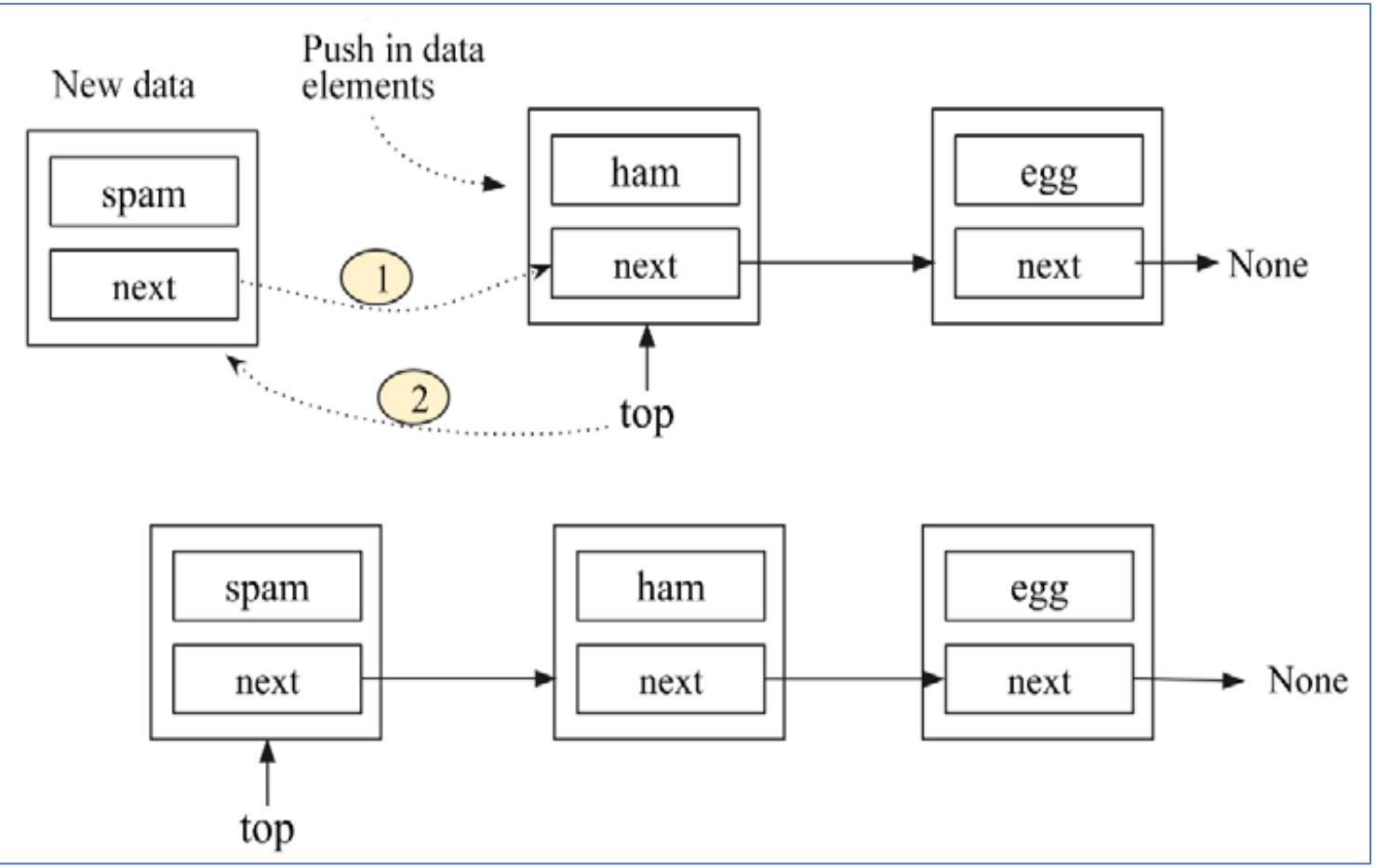
Nếu stack đã có một số phần tử, ta thực hiện các việc sau:

1. Nút mới phải có tham chiếu *next* đến nút là *top* trước đó

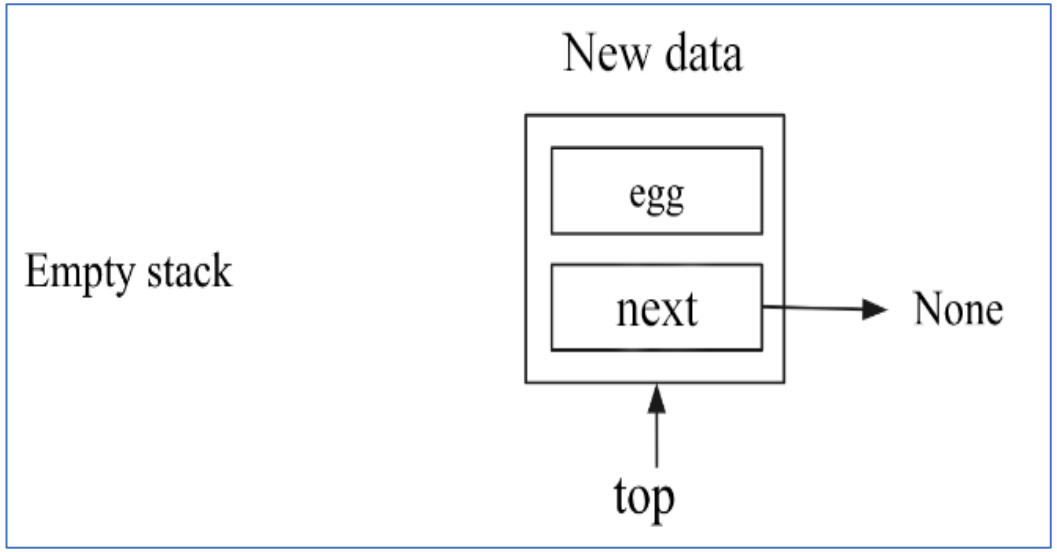
2. Đặt nút mời này tại đỉnh stack bằng cách tham chiếu *top* đến nút mới được thêm.

Lưu trữ stack bằng danh sách liên kết

Thực hiện phép toán push trên stack



Chèn phần tử "egg" vào stack rỗng



Lưu trữ stack bằng danh sách liên kết

```java
// Thêm phần tử vào stack
public void push(String value) {
    Node newNode = new Node(value);
    newNode.next = top;
    top = newNode;
    size++;
}
```

Tạo nút mới newNode và lưu dữ liệu cho nó, sau đó kiểm tra vị trí của *top*

```java
LinkedListStack stack = new LinkedListStack();
// Thêm các phần tử vào stack
stack.push("egg");
stack.push("ham");
stack.push("spam");
System.out.println("Stack ban đầu:");
stack.printStack();
System.out.println("Kích thước stack: " + stack.size());
```

Output:

spam
ham
egg

Lưu trữ stack bằng danh sách liên kết

Pop

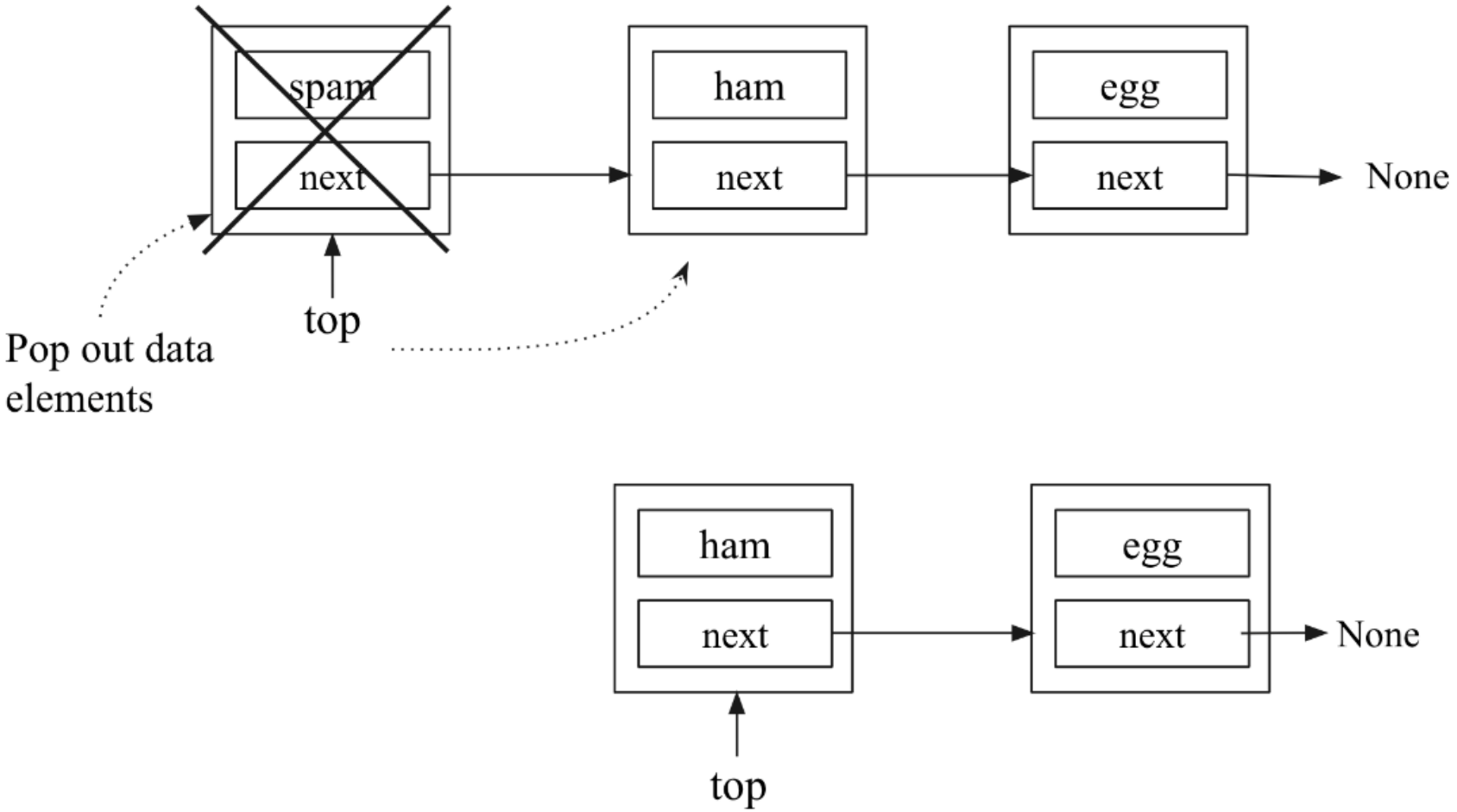
Phép toán này thực hiện đọc phần tử ở đỉnh của stack, và sau đó xóa khỏi stack. Phương thức pop trả ra phần tử ở đỉnh stack và trả ra None nếu stack rỗng.

Các bước thực hiện như sau:

1. Đầu tiên, kiểm tra stack rỗng, phép toán pop không thực hiện trên stack rỗng.
2. Nếu stack không rỗng, ta phải thay đổi tham chiếu top. Nút tiếp theo sẽ là *top*.

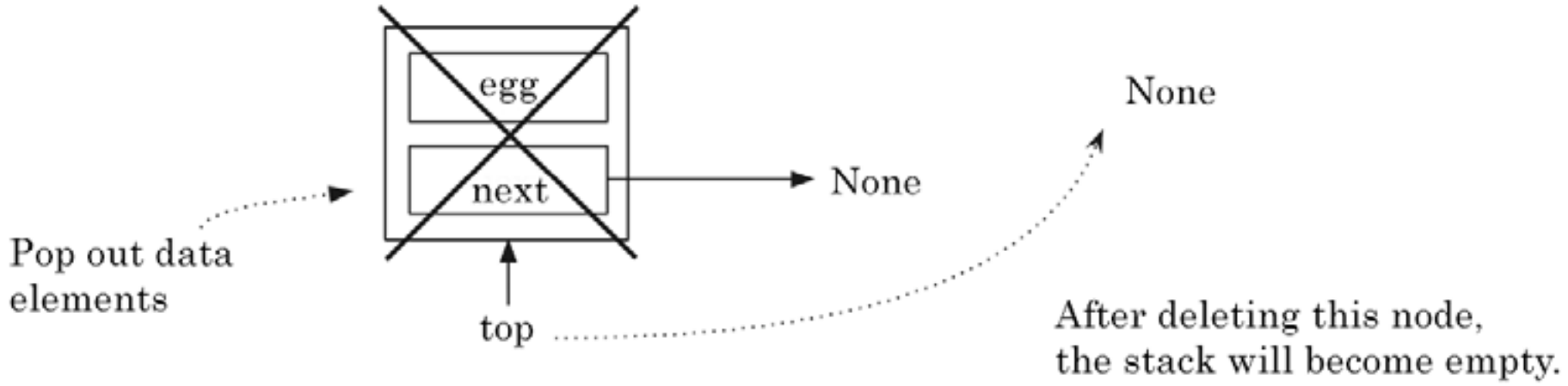
Lưu trữ stack bằng danh sách liên kết

Thực hiện phép toán *pop* trên stack



Lưu trữ stack bằng danh sách liên kết

3. Khi chỉ có 1 nút trong stack, stack sẽ trở thành rỗng sau khi thực hiện *pop*. Ta cần thay đổi *top* tham chiếu đến None



4. Xóa nút này và biến *top* chỉ đến None

5. Giảm kích thước stack đi 1 nếu stack không rỗng.

Lưu trữ stack bằng danh sách liên kết

```java
// Lấy và xóa phần tử đầu tiên khỏi stack
public String pop() {
    if (isEmpty()) {
        System.out.println("Stack rỗng");
        return null;
    } else {
        String value = top.data;
        top = top.next;
        size--;
        return value;
        }
}
```

```java
String removedElement = stack.pop();
System.out.println("Stack sau khi xoa phan tu:");
stack.printStack();
```

Output:

ham egg

Lưu trữ stack bằng danh sách liên kết

Peek operation

Phương thức này trả ra phần tử ở đỉnh stack và không xóa nó khỏi stack.

```
public String peek() {
    if (isEmpty()) {
        System.out.println("Stack rỗng");
        return null;
    } else {
        return top.data;
    }
}
```

```
String topElement = stack.peek();
System.out.println("Phần tử ở đỉnh stack: " + topElement
    );
```

Output:

spam

Ứng dụng của stacks

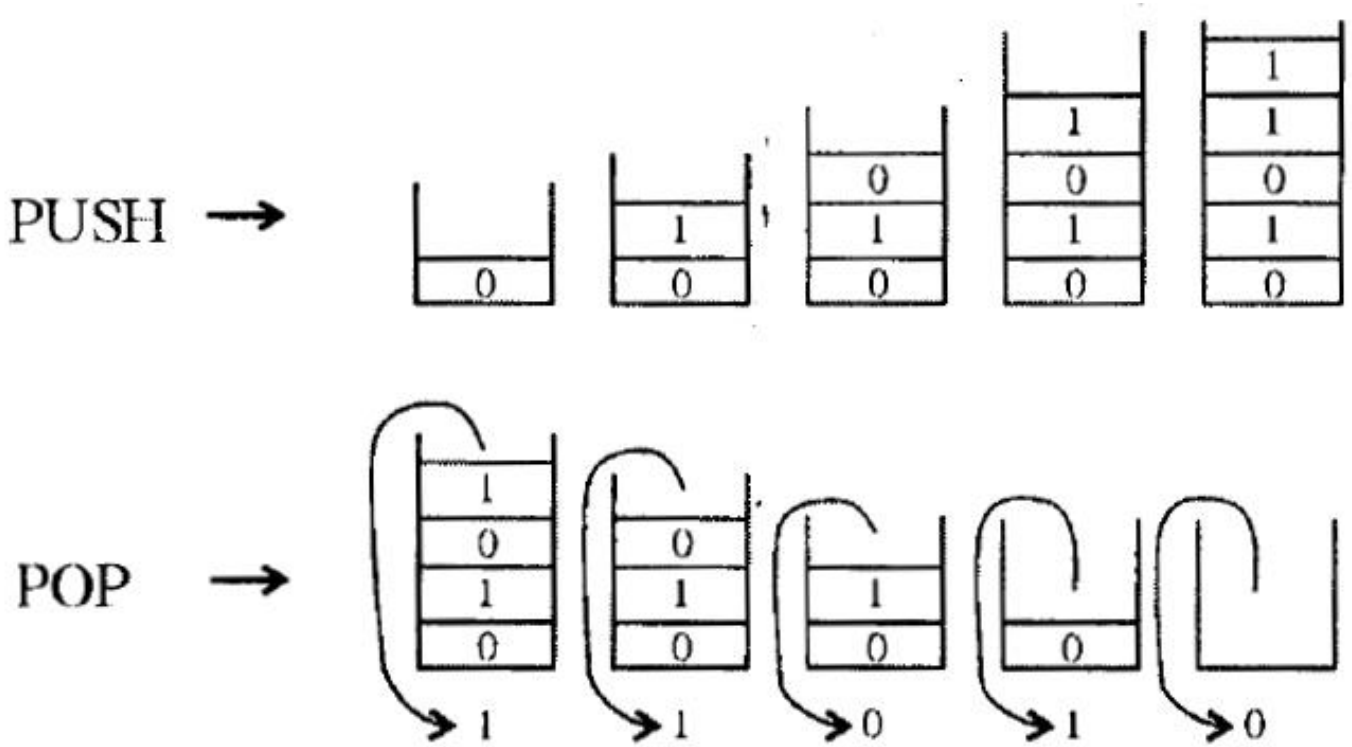
Độ phức tạp thời gian của Stack

Khi thực thi stack sử dụng mảng, push và pop lấy thời gian hằng số, i.e. O(1).

- ***Ứng dụng đổi cơ số***

Ví dụ:

$(26)_{10} \rightarrow (01011) \rightarrow (11010)_2$



CMC UNIVERSITY

Ứng dụng của stacks

- ***Kiểm tra dấu ngoặc của biểu thức***

Viết hàm *check_brackets* kiểm tra liệu một biểu thức đã cho có chứa các cặp dấu ngoặc ( ), [ ], hoặc { } cân bằng không, tức là, số lượng dấu ngoặc đóng có khớp với số lượng dấu ngoặc mở?

Hàm trên phân tích cú pháp từng ký tự trong biểu thức được truyền cho nó:

- Nếu nó nhận được một dấu ngoặc mở, nó sẽ được đẩy vào stack.
- Nếu nó nhận được dấu ngoặc đóng, sẽ lấy phần tử trên cùng ra khỏi ngăn xếp và so sánh hai dấu ngoặc để đảm bảo loại của chúng khớp nhau: (should match), [should match], { should match }. Nếu không, trả về False; nếu không, tiếp tục phân tích cú pháp.

Khi đến cuối biểu thức, cần thực hiện một lần kiểm tra cuối cùng. Nếu ngăn xếp rỗng thì có thể trả về True. Nhưng nếu ngăn xếp không trống, thì tức là có một ngoặc mở và không có ngoặc đóng phù hợp và sẽ trả về False.

Ứng dụng của stacks

- ***Biểu thức số học và ký pháp Balan (Polish notation)***

Cách viết biểu thức số học được gọi là ký hiệu. Một biểu thức số học có thể được viết với ba ký hiệu khác nhau nhưng tương đương nhau (tức là không làm thay đổi bản chất hoặc kết quả đầu ra của biểu thức): *trung tố (Infix), tiền tố (Prefix), hậu tố (Postfix)*

Expression No	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) * c	* + a b c	a b + c *
3	a * (b + c)	* a + b c	a b c + *
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) * (c + d)	* + a b + c d	a b + c d + *
6	((a + b) * c) - d	- * + a b c d	a b + c * d -

CMC UNIVERSITY

Ứng dụng của stacks

- ***Tính giá trị biểu thức hậu tố***

Độ ưu tiên và tính kết hợp xác định thứ tự đánh giá của một biểu thức: **, *, /, +, -

Thuật toán sau thực hiện tính giá trị biểu thức hậu tố:

Bước 1 – duyệt biểu thức từ trái sang phải

Bước 2 – nếu gặp toán hạng thì đẩy nó vào stack

Bước 3 – nếu gặp toán tử, lấy toán hạng từ stack và thực hiện phép toán

Bước 4 – lưu trữ đầu ra của bước 3, và đẩy lại vào stack

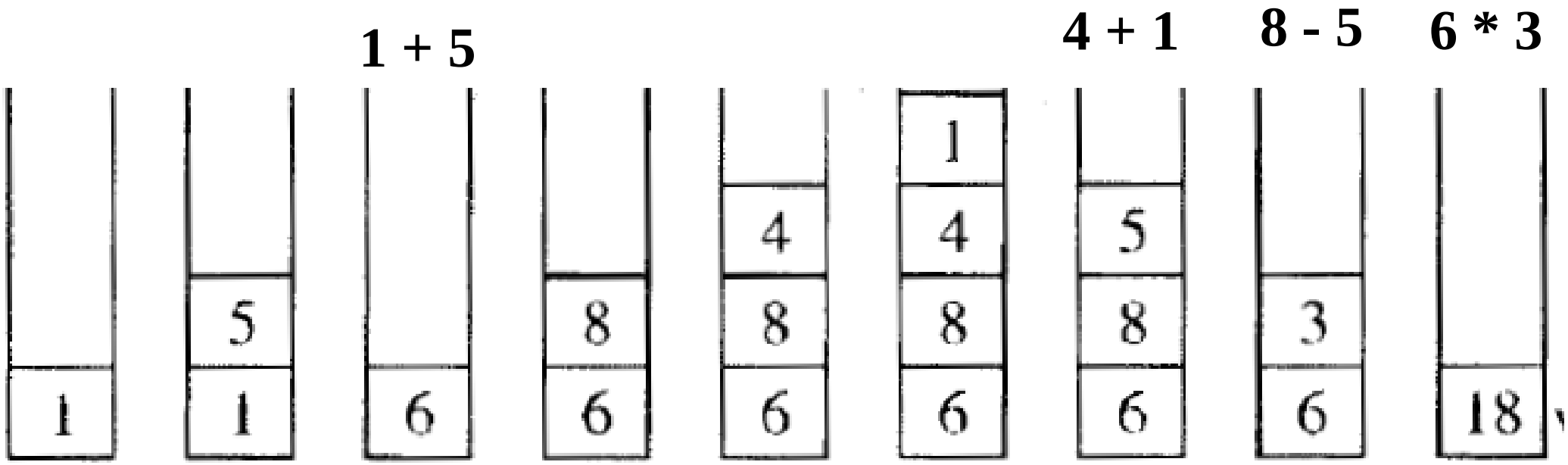
Bước 5 – duyệt biểu thức cho đến khi tất cả các toán hạng được sử dụng

Bước 6 – lấy phần tử khỏi stack và thực hiện phép toán

Ứng dụng của stacks

- ***Tính giá trị biểu thức hậu tố***

Ví dụ: (1 + 5) * (8 – (4 + 1)) = 1 5 + 8 4 1 + – *



CMC UNIVERSITY

Ứng dụng của stacks

- *Tính giá trị biểu thức hậu tố*

```
private boolean isOperator(String token) {
    return token.equals("+") || token.equals("-") ||
    token.equals("*") || token.equals("/");
}
```

```
private int performOperation(String operator, int operand1,
int operand2) {
    switch (operator) {
        case "+":
            return operand1 + operand2;
        case "-":
            return operand1 - operand2;
        case "*":
            return operand1 * operand2;
        case "/":
            if (operand2 == 0) {
             throw new ArithmeticException("Division by 0");
            }
            return operand1 / operand2;
        default:
            throw new IllegalArgumentException("Invalid
                operator");
      }
}
```

CMC UNIVERSITY

Ứng dụng của stacks

- *Tính giá trị biểu thức hậu tố*

```
public int evaluatePostfix(String expression) {
    Stack<Integer> stack = new Stack<>();
    String[] tokens = expression.split("\\s+");

    for (String token : tokens) {
        if (!isOperator(token)) {
            // Nếu token là toán hạng, đẩy nó vào stack
            stack.push(Integer.parseInt(token));
        } else {
            // Nếu token là toán tử, lấy các toán hạng từ
                stack và thực hiện phép toán
            int operand2 = stack.pop();
            int operand1 = stack.pop();
            int result = performOperation(token, operand1,
                operand2);
            stack.push(result);
        }
    }

    // Kết quả cuối cùng nằm ở đỉnh stack
    return stack.pop();
```

```
public static void main(String[] args) {
    PostfixEvaluator evaluator = new PostfixEvaluator();
    String expression = "3 4 + 2 * 7 /";
    int result = evaluator.evaluatePostfix(expression);
    System.out.println("Kết quả của biểu thức '" +
        expression + "' là: " + result);
}
```

CMC UNIVERSITY

Ứng dụng của stacks

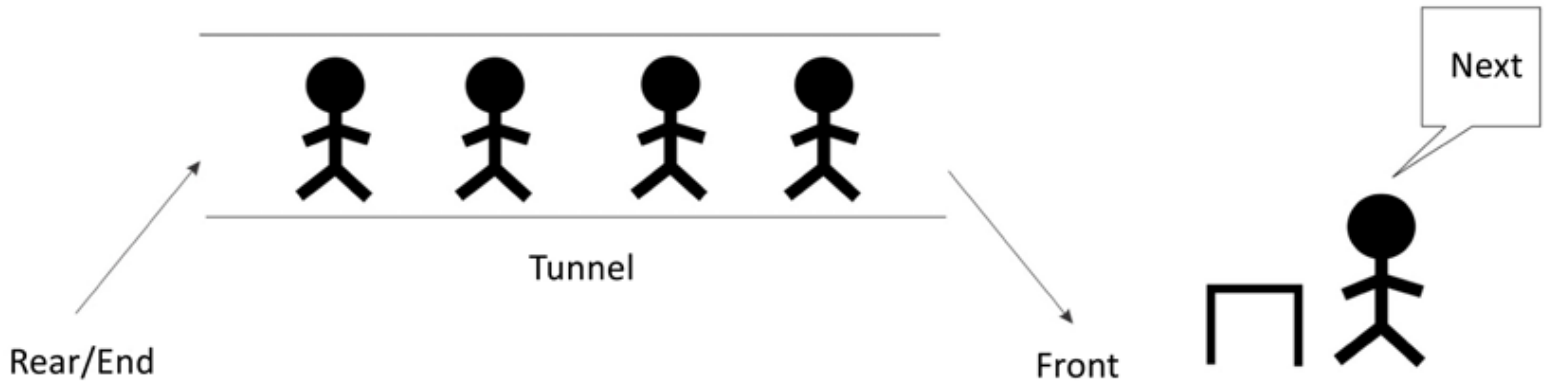
- **Đảo ngược từ** - Đặt tất cả các chữ cái vào stack và lấy chúng ra. Vì nguyên tắc LIFO của stack, ta sẽ nhận được các chữ cái theo thứ tự đảo ngược..
- **Trong trình biên dịch** - Trình biên dịch sử dụng stack để tính giá trị của các biểu thức bằng cách chuyển đổi biểu thức sang dạng tiền tố hoặc hậu tố.
- **Trong trình duyệt** - Nút **back** trong trình duyệt sẽ lưu tất cả các URL bạn đã truy cập trước đó vào một stack. Mỗi lần bạn truy cập một trang mới, nó sẽ thêm vào đầu stack. Khi bạn nhấn nút **back**, URL hiện tại sẽ bị xóa khỏi ngăn xếp và URL trước đó sẽ được truy cập.

CMC UNIVERSITY

QUEUES

Queues

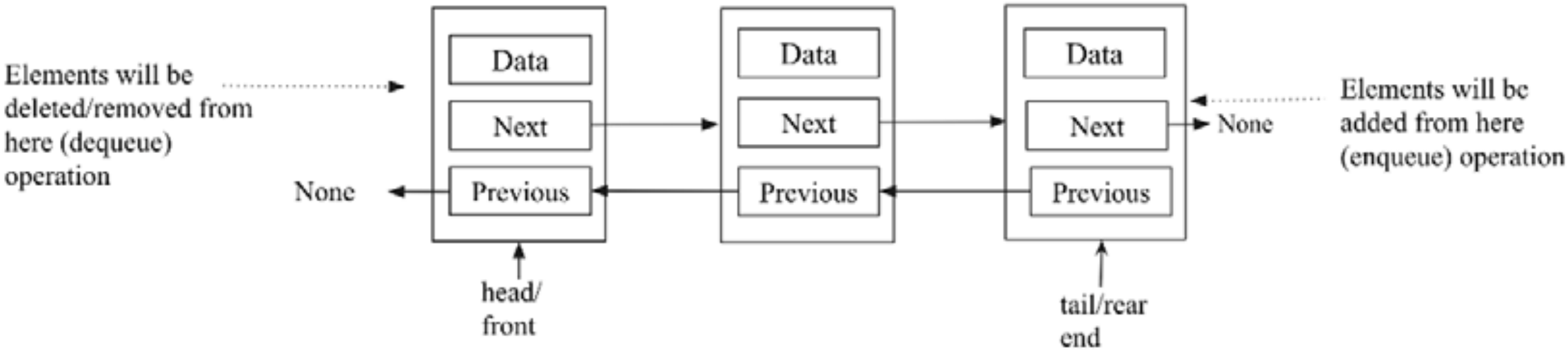
- **Hàng đợi** (queue) là tập hợp các đối tượng được chèn và xóa theo nguyên tắc ***vào trước ra trước*** (First In First Out - FIFO). Nghĩa là, các phần tử có thể được chèn vào bất kỳ lúc nào, nhưng chỉ phần tử ở trong hàng đợi lâu nhất mới có thể bị xóa tiếp theo.
- Queue là một cấu trúc dữ liệu hữu ích trong lập trình. Nó tương tự như việc xếp hàng mua vé bên ngoài rạp chiếu phim, nơi người đầu tiên vào hàng là người đầu tiên nhận được vé.



Queues

Hàng đợi là một danh sách các phần tử được lưu trữ theo thứ tự với các ràng buộc sau:

- Các phần tử dữ liệu chỉ được chèn từ một đầu, gọi là phía sau (rear)/đuôi (tail) của queue.
- Các phần tử dữ liệu chỉ được xóa từ đầu khác, gọi là phía trước (front)/đầu (head) của queue.
- Chỉ đọc được phần tử dữ liệu từ phía trước của queue.



FIFO Representation of Queue

- Trong thuật ngữ lập trình, việc đưa các mục vào queue được gọi là **enqueue** và xóa các mục khỏi queue được gọi là **dequeue**. Khi một phần tử được xếp vào queue, kích thước của queue sẽ tăng thêm 1 và khi loại bỏ một phần tử sẽ giảm số phần tử trong queue đi 1.

Queues

Các phép toán cơ bản trên Queue

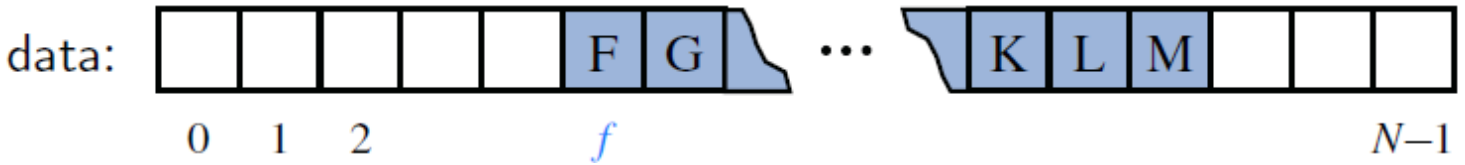
Queue là một đối tượng (ADT) cho phép các phép toán cơ bản sau:

- **enqueue (e)**: Thêm phần tử e vào cuối hàng đợi
- **dequeue()**: Xóa một phần tử khỏi đầu hàng đợi
- **isEmpty()**: Kiểm tra hàng đợi có rỗng không
- **isFull()**: Kiểm tra hàng đợi có đầy không
- **peek()**: Lấy giá trị của phần tử ở đầu hàng đợi và không xóa nó
- **size()**: trả ra số phần tử của hàng đợi

Lưu trữ queue bằng mảng

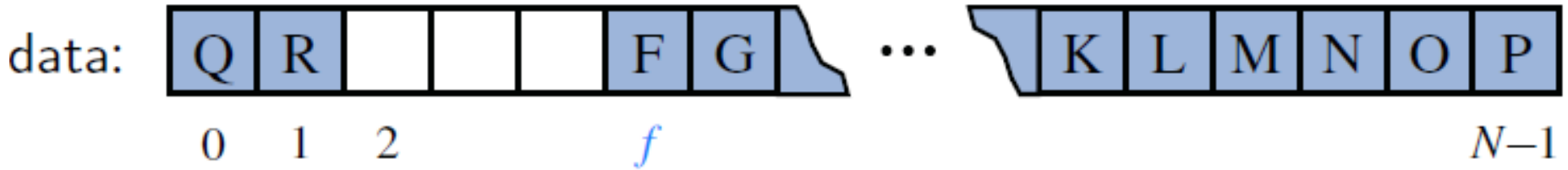
Giả sử khi các phần tử được chèn vào hàng đợi được lưu trữ bởi một mảng sao cho phần tử đầu tiên ở chỉ mục 0, phần tử thứ hai ở chỉ mục 1, v.v (như hình sau)

- Xóa phần tử tại chỉ mục 0, và dịch chuyển tất cả các phần tử còn lại sang trái 1 vị trí → thời gian thực hiện dequeue là O(n).
- Xóa phần tử tại chỉ mục 0, thay thế phần tử được loại bỏ khỏi hàng đợi trong mảng bằng một tham chiếu null và duy trì một biến f để biểu thị chỉ mục của phần tử hiện ở phía trước hàng đợi → thời gian thực hiện dequeue là O(1). Sau một số thao tác dequeue thì thu được cấu hình queue như hình sau:



Lưu trữ queue bằng mảng

- Giả sử mảng có độ dài cố định N, các phần tử mới được thêm vào hàng đợi ở phía "cuối" của hàng đợi hiện tại, cho đến chỉ số N −1 và tiếp tục ở chỉ số 0, sau đó là 1. Hình sau minh họa hàng đợi như vậy với phần tử đầu tiên F và phần tử cuối cùng R.



- Việc thực hiện cơ chế vòng tròn như vậy tương đối dễ dàng với toán tử *modulo* (được biểu thị bằng ký hiệu %) trong Java. Khi loại bỏ phần tử thì chỉ số của phần tử ở đầu front là:

f = (f + 1) % n

Ví dụ: mảng có độ dài 10, chỉ số của phần tử front là f = 7.

- Khi xóa phần tử chỉ số 7 thì f = (7 +1) % 10 = 8
- Xóa phần tử chỉ số 8, thì f = (8 + 1) % 10 = 9
- Xóa phần tử chỉ số 9, thì f = (9 +1) % 10 = 0

CMC UNIVERSITY

Lưu trữ queue bằng mảng

Thực thi queue trong Java theo nguyên tắc vòng tròn

Lớp queue gồm 3 biến: data (tham chiếu đến mảng); f (chỉ mục của phần tử đầu tiên của queue, giả sử queue không rỗng); sz (số phần tử hiện tại được lưu trữ trong queue)

```
/** Implementation of the queue ADT using a fixed-length array. */
public class ArrayQueue<E> implements Queue<E> {
  // instance variables
  private E[ ] data;                          // generic array used for storage
  private int f = 0;                          // index of the front element
  private int sz = 0;                         // current number of elements

  // constructors
  public ArrayQueue( ) {this(CAPACITY);} // constructs queue with default capacity
  public ArrayQueue(int capacity) {          // constructs queue with given capacity
    data = (E[ ]) new Object[capacity];      // safe cast; compiler may give warning
  }
```

Lưu trữ queue bằng mảng

```
// methods
/** Returns the number of elements in the queue. */
public int size( ) { return sz; }

/** Tests whether the queue is empty. */
public boolean isEmpty( ) { return (sz == 0); }

/** Inserts an element at the rear of the queue. */
public void enqueue(E e) throws IllegalStateException {
    if (sz == data.length) throw new IllegalStateException("Queue is full");
    int r = (f + sz) % data.length; // use modular arithmetic
    data[r] = e;
    sz++;
}
```

Lưu trữ queue bằng mảng

```
/** Returns, but does not remove, the first element of the queue (null if empty). */
public E first( ) {
  if (isEmpty( )) return null;
  return data[f];
}

/** Removes and returns the first element of the queue (null if empty). */
public E dequeue( ) {
  if (isEmpty( )) return null;
  E answer = data[f];
  data[f] = null;                    // dereference to help garbage collection
  f = (f + 1) % data.length;
  sz--;
  return answer;
}
```

CMC UNIVERSITY

Lưu trữ queue bằng mảng

Phân tích hiệu quả thực thi các phương thức trên queue

Mỗi phương thức trên queue sử dụng mảng thực thi trong thời gian O(1).

Độ phức tạp không gian là O(N), trong đó N là kích thước của mảng, được xác định tại thời điểm hàng đợi được tạo và không phụ thuộc vào số n < N của các phần tử thực sự có trong hàng đợi.

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
first	$O(1)$
enqueue	$O(1)$
dequeue	$O(1)$

CMC UNIVERSITY

Lưu trữ queue bằng danh sách liên kết

- Khi lưu trữ queue bằng danh sách liên kết, phía trước của hàng đợi được xem như đầu danh sách và phía sau của hàng đợi là phần cuối của danh sách.
- Thao tác thêm phần tử vào hàng đợi thực hiện ở cuối danh sách, xóa phần tử khỏi hàng đợi thực hiện ở đầu danh sách.

```
/** Realization of a FIFO queue as an adaptation of a SinglyLinkedList. */
public class LinkedQueue<E> implements Queue<E> {
  private SinglyLinkedList<E> list = new SinglyLinkedList<>();    // an empty  list
  public LinkedQueue( ) { }                    // new queue relies on the initially empty list
  public int size( ) { return list.size( ); }
  public boolean isEmpty( ) { return list.isEmpty( ); }
  public void enqueue(E element) { list.addLast(element); }
  public E first( ) { return list.first( ); }
  public E dequeue( ) { return list.removeFirst( ); }
}
```

CMC UNIVERSITY

Lưu trữ queue bằng danh sách liên kết

Phân tích hiệu quả của các phương thức khi sử dụng danh sách liên kết

- Mỗi phương thức của lớp LinkedQueue có thời gian chạy trường hợp tồi nhất là O(1)
- Vì mỗi nút lưu trữ một tham chiếu tiếp theo, nên ngoài tham chiếu phần tử, danh sách liên kết sử dụng nhiều không gian cho mỗi phần tử hơn là một mảng tham chiếu có kích thước phù hợp.
- Thực thi phương thức sử dụng danh sách liên kết có số lượng thao tác nguyên thủy lớn hơn sử dụng mảng.

Ví dụ: - thêm một phần tử vào hàng đợi dựa trên mảng bao gồm chủ yếu tính toán chỉ mục bằng số học mô-đun, lưu trữ phần tử trong ô của mảng và tăng bộ đếm kích thước.

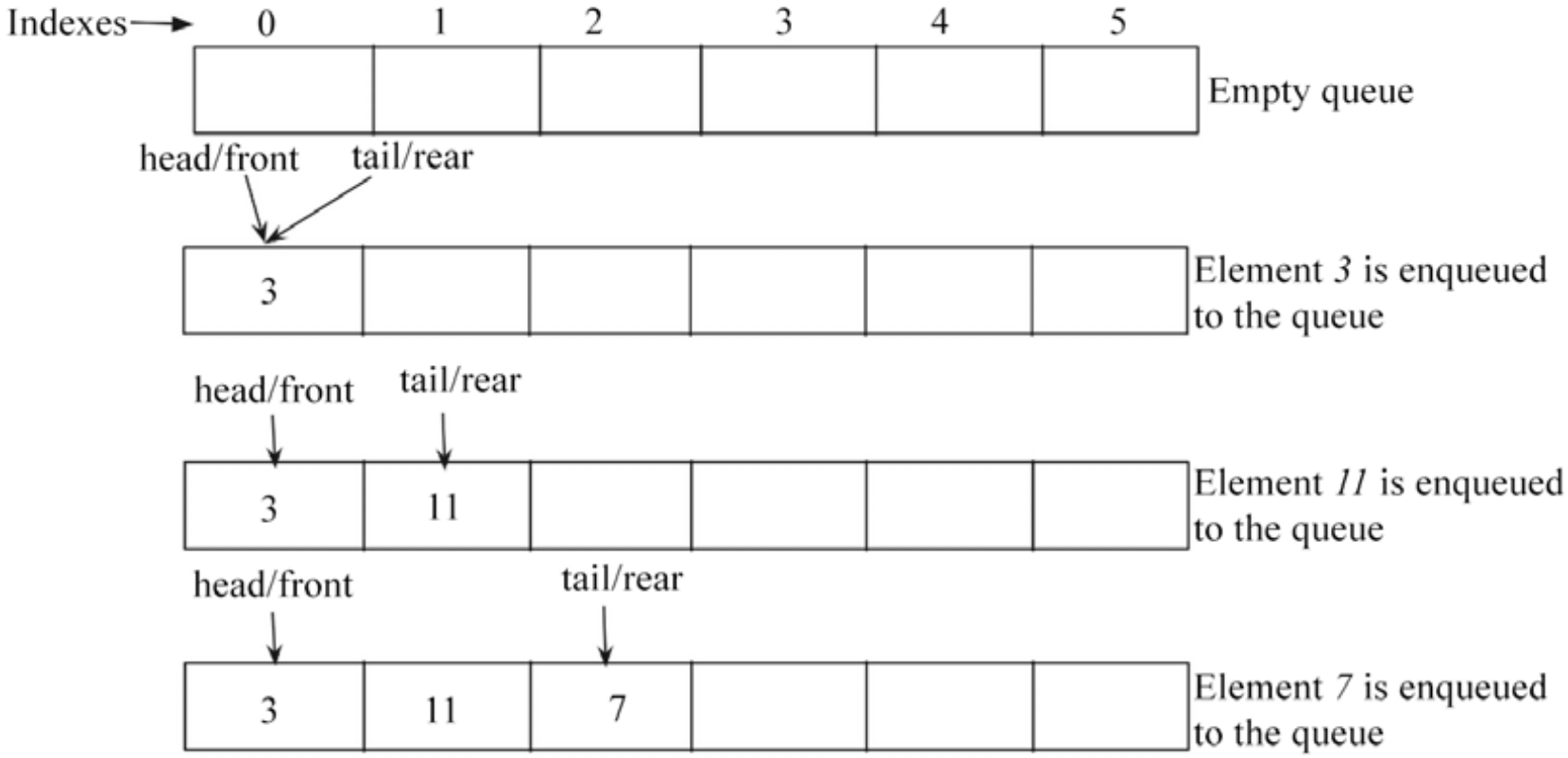
- thêm phần tử vào danh sách liên kết, bao gồm khởi tạo nút mới, liên kết lại nút hiện có với nút mới và tăng bộ đếm kích thước. Trong thực tế, điều này làm cho việc dùng danh sách liên kết tốn kém hơn dùng mảng

Lưu trữ queue bằng danh sách liên kết

Enqueue

Ví dụ: thêm phần tử vào queue được minh họa như hình sau:

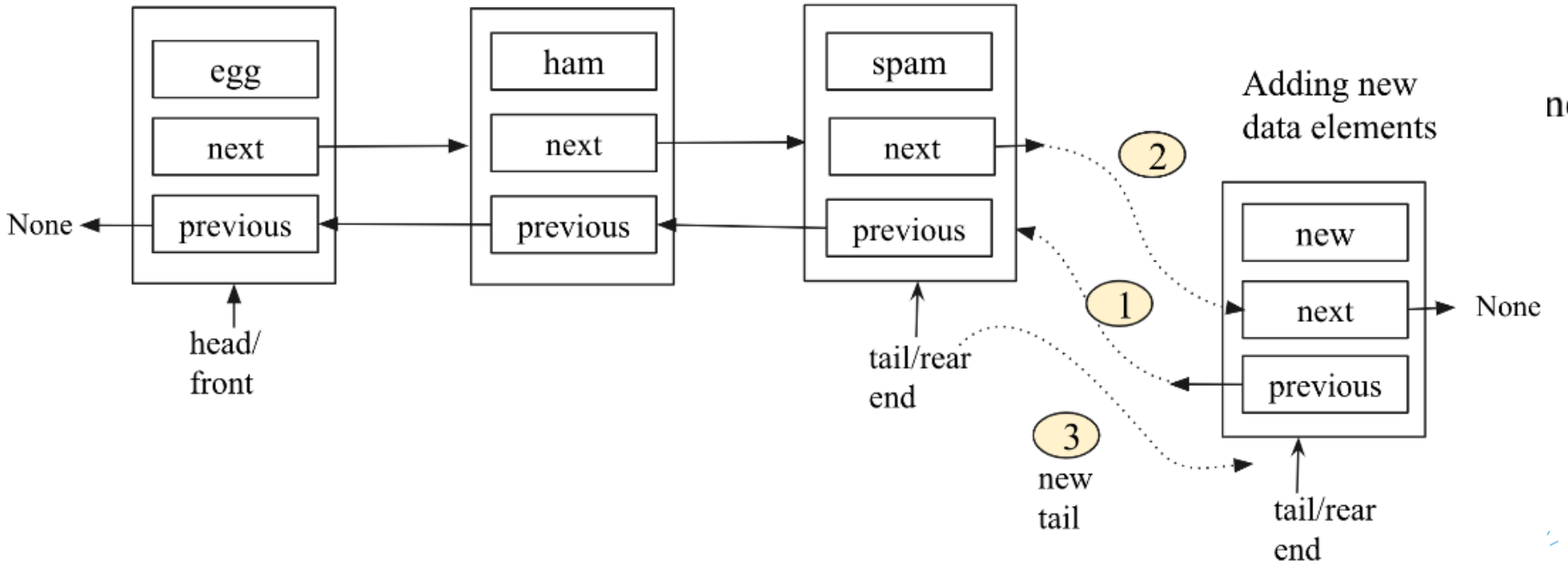
- Bắt đầu với danh sách rỗng
- Bổ sung phần tử 3 tại chỉ mục 0.
- Tiếp theo, thêm 11 tại chỉ mục 1, và di chuyển tham chiếu rear mỗi khi thêm phần tử

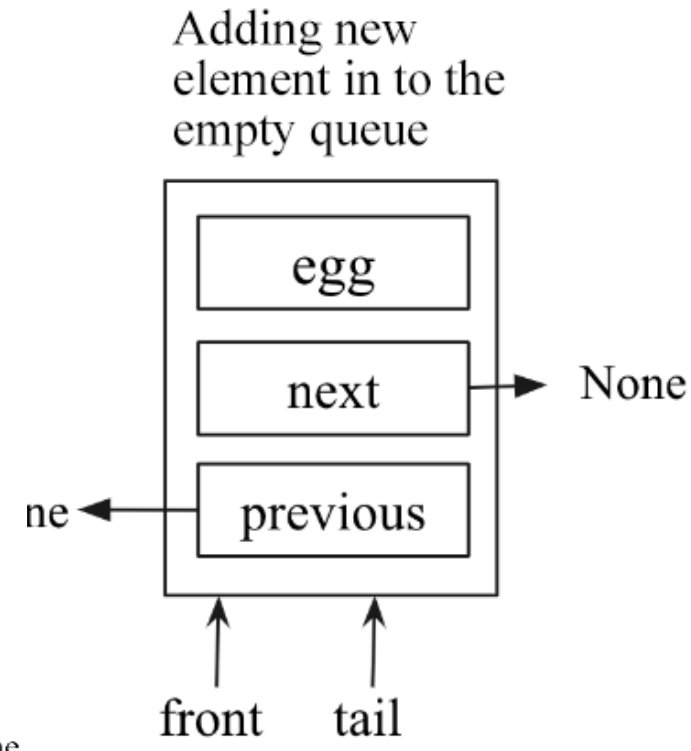


Lưu trữ queue bằng danh sách liên kết

Enqueue: Các phần tử được thêm vào đối tượng queue qua phương thức enqueue.

- Tạo một nút mới
- Nếu queue rỗng, nút mới trở thành nút đầu tiên của queue (như hình bên)
- Nếu không, nút mới được bổ sung vào phía cuối queue.

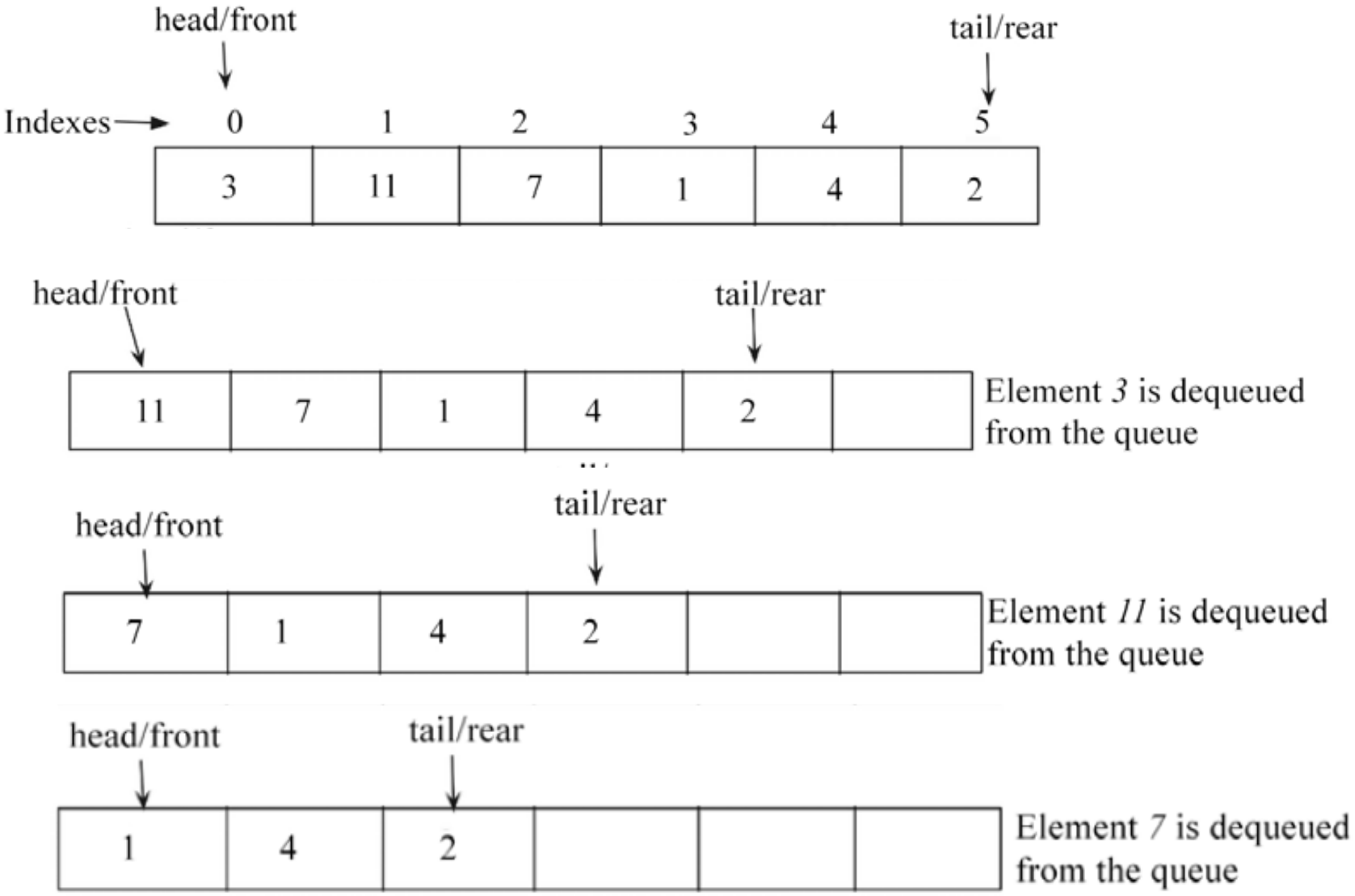




Lưu trữ queue bằng danh sách liên kết

Dequeue

Ví dụ: Thực hiện dequeue với queue sau:



Lưu trữ queue bằng danh sách liên kết

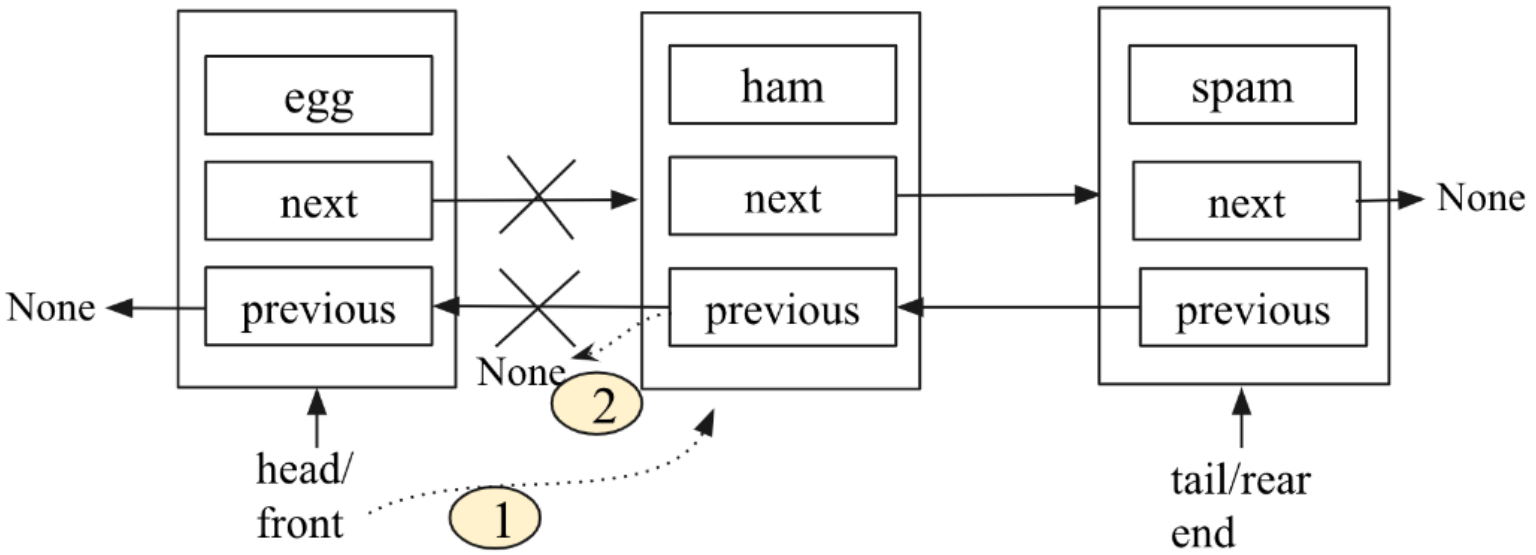
Dequeue

Phương thức này xóa nút tại phía trước queue.

- Kiểm tra xem phần tử xóa có là nút cuối của queue không?

  + Nếu nó là nút cuối, làm cho queue trở thành rỗng sau khi thực hiện dequeue

  + Nếu không, xóa phần tử đầu tiên bằng cách cập nhật tham chiếu *front/head* đến nút tiếp theo và tham chiếu previous của nút đầu mới chỉ đến *None*:



CMC UNIVERSITY

Ứng dụng của queue

- Lập lịch CPU, lập lịch ổ đĩa
- Khi dữ liệu được truyền không đồng bộ giữa hai tiến trình. Hàng đợi được sử dụng để đồng bộ hóa. Ví dụ : IO Buffers, pipes, file IO, etc
- Xử lý các ngắt trong hệ thống thời gian thực.
- Hệ thống điện thoại của trung tâm cuộc gọi sử dụng hàng đợi để giữ các cuộc gọi của người gọi theo thứ tự.

CMC UNIVERSITY

Các kiểu của queues

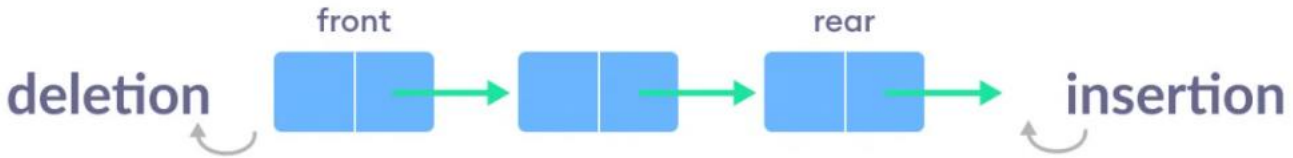
- Hàng đợi đơn (Simple Queue)
- Hàng đợi vòng (Circular Queue)
- Hàng đợi ưu tiên (Priority Queue)
- Hàng đợi hai đầu (Double Ended Queue - deque)

CMC UNIVERSITY

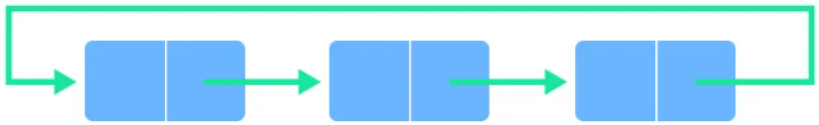
Các kiểu của queues

Hàng đợi đơn

- Hàng đợi đơn (Hình a) tuân thủ nghiêm ngặt quy tắc FIFO, chèn phần tử tại phía sau (rear) và thêm phần tử vào phía trước (front).



a) Simple Queue Representation



b) Circular Queue Representation

Hàng đợi vòng

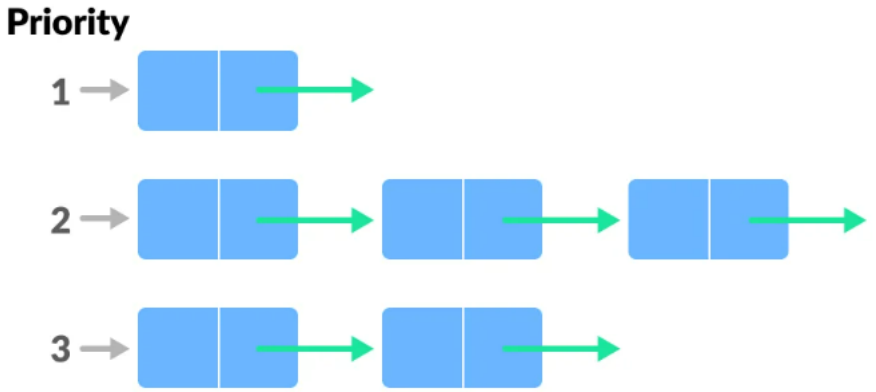
- Trong hàng đợi vòng, phần tử cuối cùng trỏ đến phần tử đầu tiên tạo thành liên kết vòng tròn
- Ưu điểm chính của hàng đợi vòng so với hàng đợi đơn là sử dụng bộ nhớ tốt hơn. Nếu vị trí cuối cùng đầy và vị trí đầu tiên trống, chúng ta có thể chèn một phần tử vào vị trí đầu tiên.

Các kiểu của queues

Hàng đợi ưu tiên

- Là một loại hàng đợi đặc biệt, trong đó mỗi phần tử được liên kết với một mức độ ưu tiên và được phục vụ theo mức độ ưu tiên của nó. Nếu các phần tử có cùng mức độ ưu tiên, chúng sẽ được sắp xếp theo thứ tự trong hàng đợi.

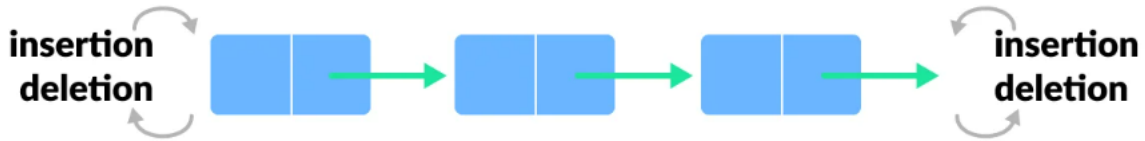
- Chèn phần tử dựa vào giá trị và xóa phần tử dựa vào độ ưu tiên.



Priority Queue Representation

Hàng đợi hai đầu

Trong hàng đợi hai đầu, việc chèn và loại bỏ các phần tử có thể được thực hiện từ phía trước hoặc phía sau. Do đó không tuân theo nguyên tắc FIFO.



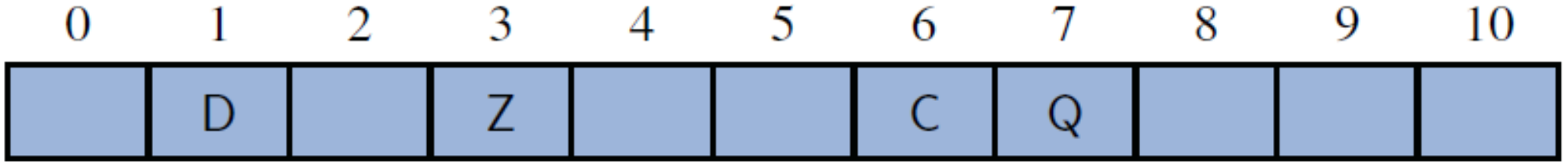
Deque Representation

CMC UNIVERSITY

BẢNG BĂM

Giới thiệu bảng băm

- Bảng băm (hash table) là một cấu trúc dữ liệu, trong đó các phần tử được truy cập bằng từ khóa mà không phải chỉ mục, không giống trong danh sách và mảng. Trong cấu trúc dữ liệu này, mục dữ liệu được lưu trữ trong các cặp (*khóa – giá trị)* (*key-value)* tương tự như với *map*.
- Ví dụ một bảng tra cứu có độ dài 11 cho *map* chứa các mục (1,D), (3,Z), (6,C) và (7,Q).



- Trong cách biểu diễn trên, ta lưu trữ giá trị được liên kết với khóa k tại chỉ mục k của bảng Các phép toán get, put và remove trên *map* cơ bản có thể được thực hiện trong thời gian trường hợp tồi nhất là O(1).

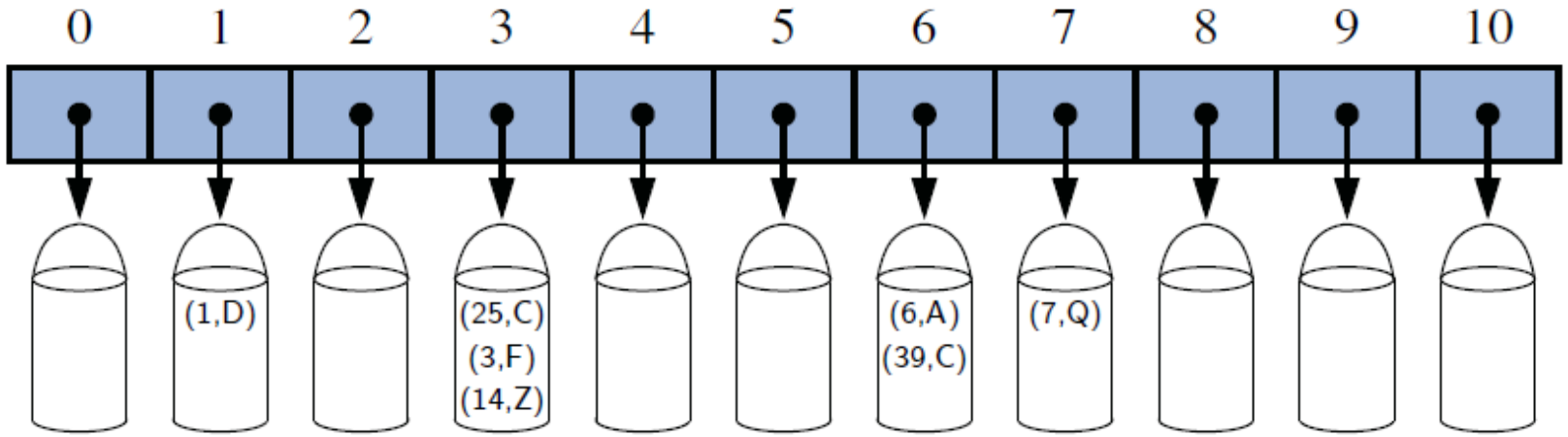
Giới thiệu bảng băm

- Có thể mở rộng cách biểu diễn trên vì:

  - Không cần dành một mảng có độ dài N trong trường hợp $N \gg n$.

  - Không yêu cầu khóa của *map* phải là số nguyên.

- Khái niệm mới cho bảng băm sử dụng *hàm băm h* (hashing function) để ánh xạ mỗi khóa k vào một số nguyên trong phạm vị [0, N-1], trong đó N là dung lượng của mảng các nhóm cho một bảng băm. Lý tưởng nhất là các khóa sẽ được phân phối tốt trong phạm vi từ 0 đến $N-1$ theo hàm băm. Tuy nhiên, có thể có nhiều khóa khác nhau được ánh xạ tới cùng một chỉ mục, như ví dụ trong hình sau, bảng như một *mảng nhóm* (bucket array):



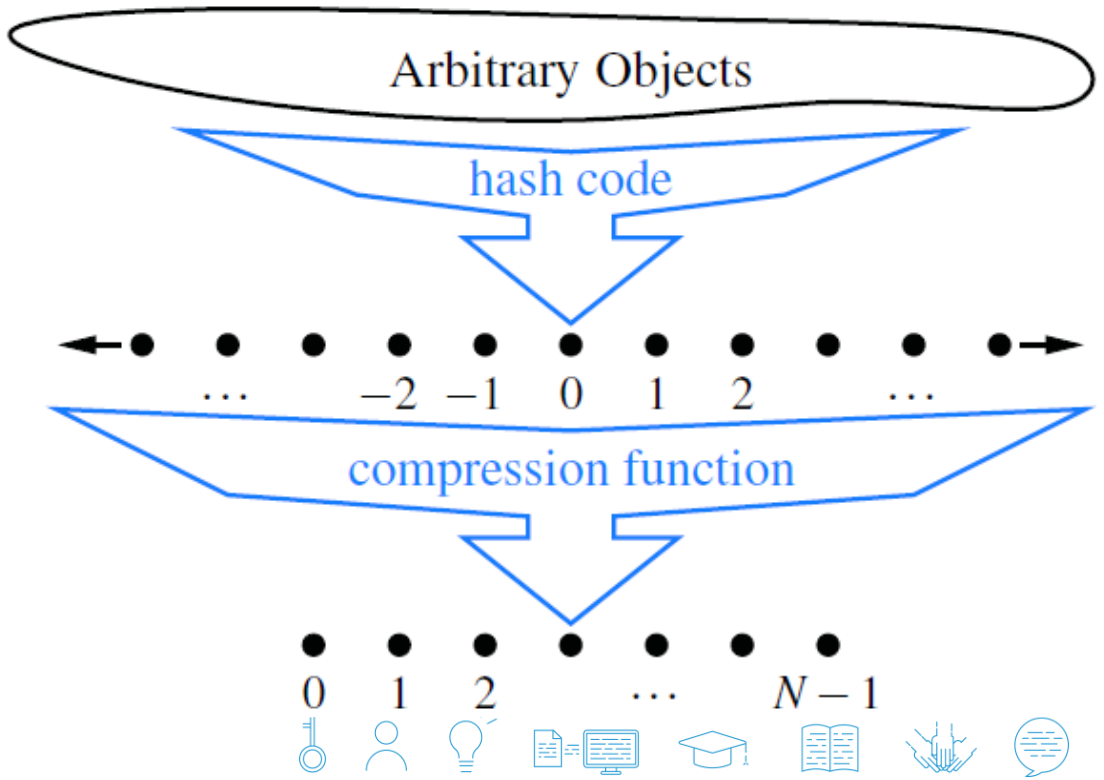
Hàm băm

- Với hàm băm h, giá trị hàm băm h(k) được sử dụng làm chỉ mục trong mảng nhóm A, thay vì khóa k. Tức là, lưu trữ mục nhập (k,v) trong nhóm (bucket) A[h(k)].

- Nếu có hai hoặc nhiều khóa có cùng giá trị băm thì hai mục nhập khác nhau sẽ được ánh xạ tới cùng một nhóm trong A. Trường hợp này được gọi là *va chạm* (xung đột).

- Hàm băm là "tốt" nếu nó ánh xạ các khóa trong *map* sao cho giảm thiểu các va chạm. Ngoài ra, cũng cần chọn hàm băm tính được nhanh và dễ tính toán.

- Việc đánh giá hàm băm h(k) thường bao gồm hai phần - *mã băm* (hash code) ánh xạ khóa k tới một số nguyên và *hàm nén* (compression function) ánh xạ mã băm tới một số nguyên trong một phạm vi của các chỉ số, [0,N −1], cho một bucket array như hình sau



CMC UNIVERSITY

Hàm băm

Ví dụ: Giả sử cần lưu trữ các chuỗi {“abcdef”, “bcdefa”, “cdefab” , “defabc”} trong bảng băm, sử dụng hàm băm: chỉ mục cho một chuỗi bằng tổng giá trị ASCII của các ký tự modulo 599.

(giá trị ASCII của a, b, c, d, e, f: 97, 98, 99, 100, 101, 102)

- Vì chỉ mục của tất cả các chuỗi đều giống nhau nên có thể tạo danh sách trên chỉ mục đó và chèn tất cả các chuỗi vào danh sách.

- Để truy cập vào một chuỗi chiếm thời gian O(n) (n là số chuỗi)

Index				
0				
1				
2	abcdef	bcdefa	cdefab	defabc
3				
4				
-				
-				
-				

Hàm băm

Sử dụng hàm băm khác: chỉ mục cho một chuỗi bằng tổng giá trị ASCII của các ký tự nhân với thứ tự tương ứng của chúng trong chuỗi, sau đó modulo với 2069 (số nguyên tố)

String	Hash function	Index
abcdef	(97*1* + *98*2 + 99*3* + *100*4 + 101*5* + *102*6)%2069	38
bcdefa	(98*1* + *99*2 + 100*3* + *101*4 + 102*5* + *97*6)%2069	23
cdefab	(99*1* + *100*2 + 101*3* + *102*4 + 97*5* + *98*6)%2069	14
defabc	(100*1* + *101*2 + 102*3* + *97*4 + 98*5* + *99*6)%2069	11

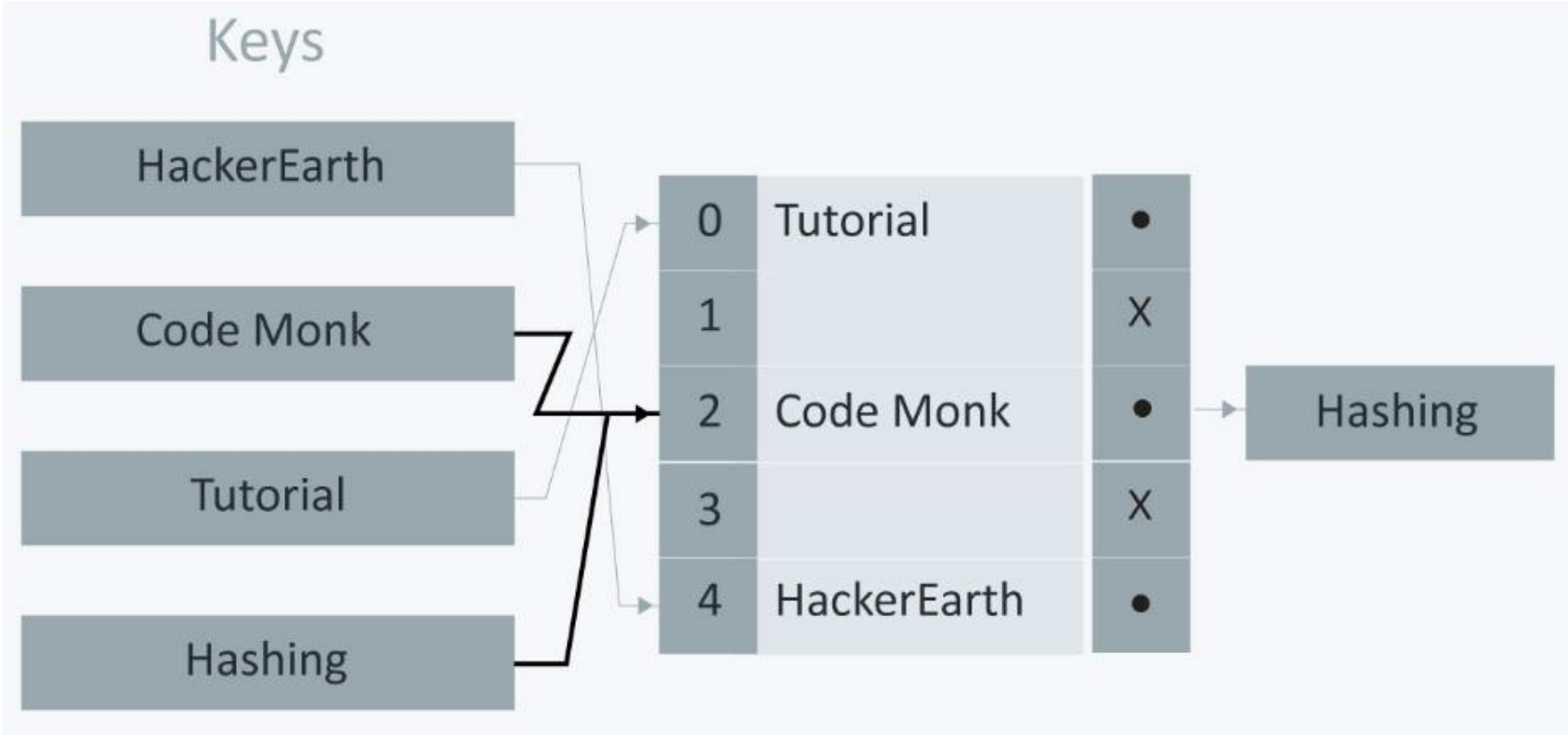
Index	
0	
1	
-	
-	
-	
11	defabc
12	
13	
14	cdefab
-	
-	
-	
23	bcdefa
-	
-	
-	
38	abcdef
-	
-	

Các kỹ thuật giải quyết xung đột

Chuỗi riêng biệt (băm mở)

Trong chuỗi riêng biệt, mỗi phần tử của bảng băm là một danh sách liên kết. Nếu có xung đột (tức là hai phần tử khác nhau có cùng giá trị băm) thì lưu trữ cả hai phần tử trong cùng một danh sách liên kết.

- Nếu các khóa phân bổ đồng đều thì chi phí trung bình của việc tìm kiếm chỉ phụ thuộc số lượng khóa trung bình trên mỗi danh sách liên kết.
- Trường hợp tồi nhất là khi tất cả các mục được chèn vào cùng một danh sách liên kết.



CMC UNIVERSITY

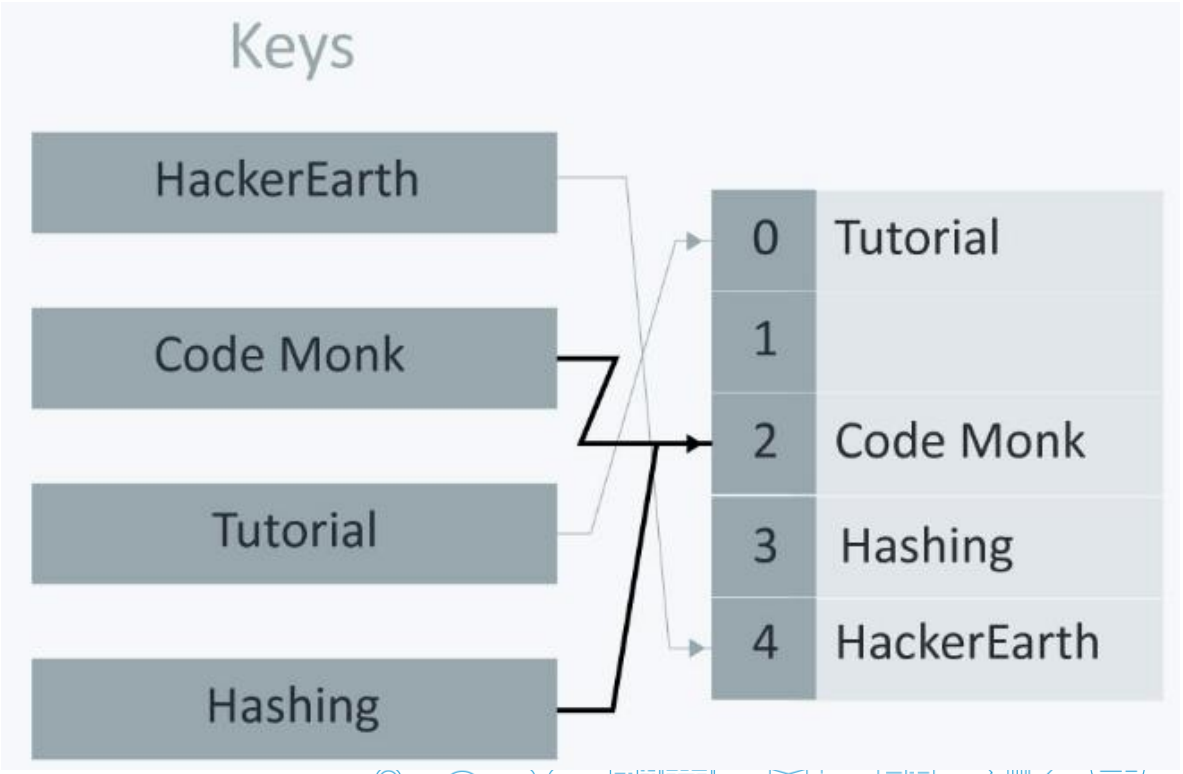
Các kỹ thuật giải quyết xung đột

Thăm dò tuyến tính (địa chỉ mở hoặc băm đóng)

- Khi chèn một mục mới, chỉ số băm của giá trị băm sẽ được tính toán và sau đó mảng sẽ được kiểm tra (bắt đầu bằng chỉ mục băm). Nếu vị trí ở chỉ mục băm không bị chiếm dụng thì bản ghi mục nhập sẽ được chèn vào vị trí ở chỉ mục băm, nếu không nó sẽ tiến hành theo trình tự thăm dò nào đó cho đến khi tìm thấy vị trí trống.

- Thăm dò tuyến tính là khi khoảng cách giữa các lần thăm dò liên tiếp được cố định (thường là 1).

```
index = index % hashTableSize
index = (index + 1) % hashTableSize
index = (index + 2) % hashTableSize
index = (index + 3) % hashTableSize
```



Các kỹ thuật giải quyết xung đột

Thăm dò bậc hai

- Thăm dò bậc hai tương tự thăm dò tuyến tính và sự khác biệt duy nhất là khoảng cách giữa các lần dò liên tiếp hoặc các khe vào. Ở đây, khi vị trí ở chỉ mục băm cho bản ghi mục nhập đã được sử dụng, phải duyệt ngang cho đến khi tìm thấy vị trí trống. Khoảng cách giữa các vị trí được tính bằng cách cộng giá trị liên tiếp của đa thức tùy ý vào chỉ mục băm ban đầu.
- Ví dụ: Giả sử rằng chỉ mục băm cho một mục nhập là chỉ mục và tại chỉ mục có một vị trí bị chiếm dụng. Trình tự thăm dò sẽ như sau:

index = index % hashTableSize

index = (index + 1^2) % hashTableSize

index = (index + 2^2) % hashTableSize

index = (index + 3^2) % hashTableSize

and so on...

CMC UNIVERSITY

Các kỹ thuật giải quyết xung đột

Băm kép

- Băm kép tương tự như thăm dò tuyến tính và điểm khác biệt là khoảng cách giữa các lần thăm dò liên tiếp được tính bằng cách sử dụng hai hàm băm..
- Ví dụ: Giả sử chỉ mục băm cho một bản ghi mục nhập là một chỉ mục được tính toán bởi một hàm băm và vị trí tại chỉ mục đó đã được chiếm giữ. Ta phải bắt đầu duyệt theo trình tự thăm dò cụ thể để tìm kiếm một vị trí trống. Trình tự thăm dò sẽ là:

```
index = (index + 1 * indexH) % hashTableSize;
index = (index + 2 * indexH) % hashTableSize;

and so on...
```

Trong đó, indexH là giá trị băm được tính bởi hàm băm khác

Truy xuất phần tử từ bảng băm

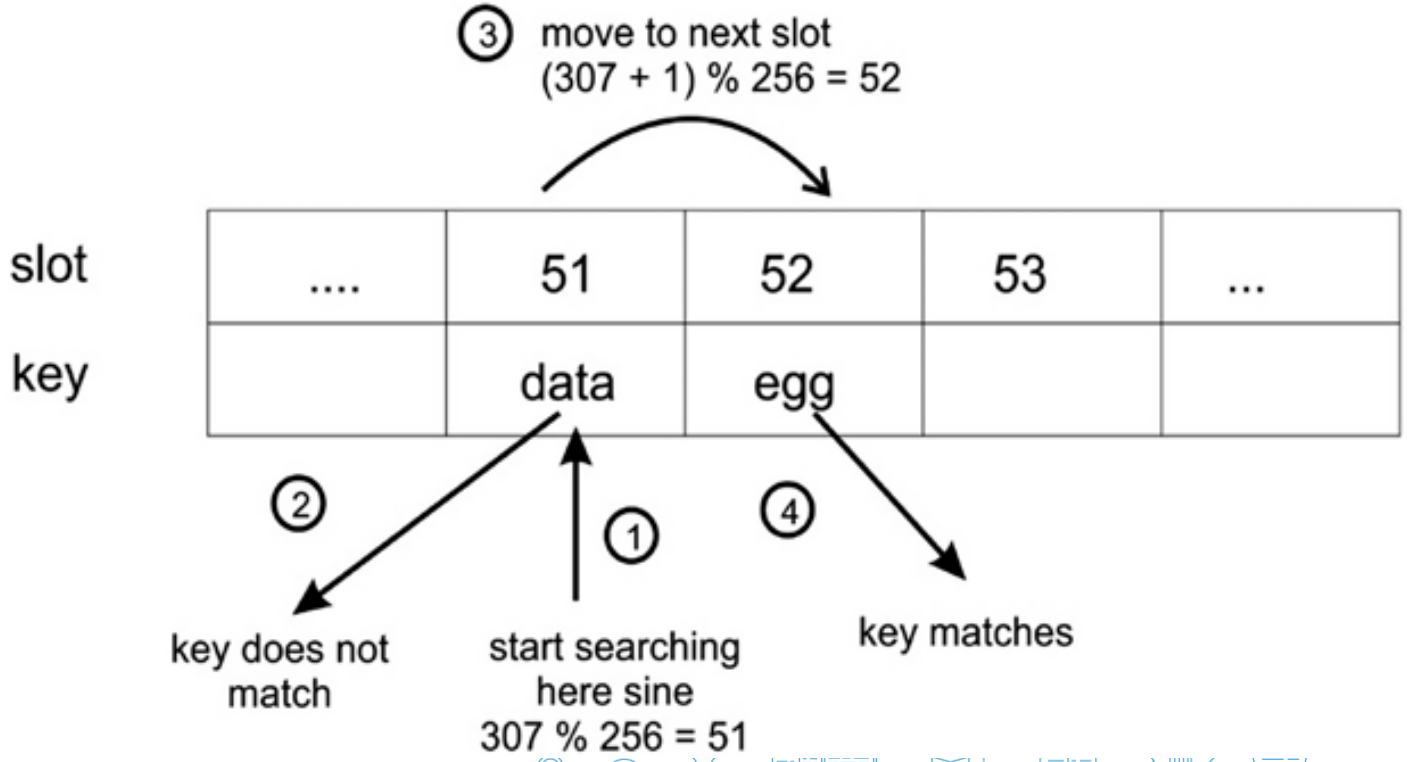
Truy xuất các phần tử từ bảng băm: giá trị được lưu trữ tương ứng với khóa sẽ được trả về

1. Tính giá trị băm cho chuỗi khóa đã cho, egg, được giá trị 51. Sau đó, so sánh khóa này với giá trị khóa được lưu ở vị trí 51, nhưng nó không khớp.

2. Vì khóa không khớp, nên tính giá trị băm mới.

3. Kiểm tra khóa tại vị trí giá trị băm mới tạo là 52; so sánh chuỗi khóa với giá trị khóa được lưu và nó khớp, như trong sơ đồ sau.

4. Giá trị khóa lưu trữ được trả về tương ứng với giá trị khóa này trong bảng băm.



CMC UNIVERSITY

Ứng dụng của bảng băm

- Hệ thống quản lý bộ nhớ
- Các ứng dụng trong mật mã
- Đánh chỉ mục cơ sở dữ liệu

CMC UNIVERSITY

MAPS

CMC UNIVERSITY

Giới thiệu map

- Map là một kiểu dữ liệu trừu tượng được thiết kế để lưu trữ và truy xuất các giá trị một cách hiệu quả dựa trên khóa tìm kiếm xác định duy nhất cho từng giá trị.
- Cụ thể, map lưu trữ các cặp *khóa* - *giá trị* (k,v), mà được gọi là các *mục* (entry), trong đó k là khóa và v là giá trị tương ứng của nó. Khóa phải là duy nhất để việc liên kết giữa các khóa với các giá trị được xác định như một ánh xạ.
- Ví dụ, web được xem như một map có các mục là các trang web. Khóa của một trang là URL của nó (ví dụ: http://datastructures.net/) và giá trị của nó là nội dung trang.
- Map còn được gọi là mảng kết hợp (associative arrays), vì khóa của mục đóng vai trò giống như một chỉ mục trên map, nó hỗ trợ map định vị mục liên quan một cách hiệu quả. Tuy nhiên, không giống như mảng chuẩn, khóa của map không nhất thiết phải là số và không chỉ định trực tiếp một vị trí trong cấu trúc

CMC UNIVERSITY

Các ứng dụng của map

Các ứng dụng phổ biến của map bao gồm:

- Hệ thống thông tin của trường đại học, dùng ID sinh viên làm khóa và được ánh xạ tới giá trị là hồ sơ của sinh viên đó (như tên, địa chỉ và điểm khóa học của sinh viên).
- Hệ thống tên miền (DNS) ánh xạ tên máy chủ, ví dụ www.wiley.com, tới địa chỉ giao thức Internet (IP), ví dụ 208.215.179.146.
- Trang web truyền thông xã hội thường lấy tên người dùng (không phải số) làm khóa có thể được ánh xạ một cách hiệu quả tới thông tin liên quan của một người dùng cụ thể.
- Thông tin khách hàng của công ty, với số tài khoản của khách hàng hoặc ID người dung là khóa và bản ghi chứa thông tin của khách hàng là giá trị. Map sẽ cho phép đại diện dịch vụ truy cập nhanh vào hồ sơ của khách hàng khi được cung cấp khóa.
- Hệ thống đồ họa máy tính có thể ánh xạ tên màu, chẳng hạn màu "ngọc lam", vào bộ ba số mô tả cách thể hiện RGB (đỏ-lục-xanh) của màu đó, như (64, 224, 208).

CMC UNIVERSITY

Các phép toán trên map

Map M hỗ trợ các phương thức sau:

size( ): trả về số lượng mục trong M.

isEmpty( ): trả ra giá trị boolean cho biết M có rỗng không.

get(k): trả ra giá trị v liên kết với khóa k, nếu mục này tồn tại; nếu không trả ra null.

put(k, v): Nếu M không có mục nào có khóa bằng k, thì thêm mục (k,v) vào M và trả ra null; nếu không, thay thế v cho giá trị của mục có khóa bằng k và trả ra giá trị cũ.

remove(k): Xóa khỏi M mục có khóa bằng k, và trả ra giá trị của nó; nếu M không có mục như vậy, thì trả ra null.

keySet( ): trả ra tập hợp có thể lặp lại gồm tất cả các khóa được lưu trong M.

values( ): trả ra tập hợp có thể lặp lại gồm tất cả giá trị của các mục lưu trong M (lặp nếu nhiều khóa ánh xạ vào cùng giá trị).

entrySet( ): trả ra tập hợp có thể lặp gồm tất cả các mục khóa-giá trị trong M.

CMC UNIVERSITY

Ví dụ

Ví dụ: các phép toán trên một map lưu trữ các mục ban đầu trống với các khóa là các số nguyên và giá trị là các ký tự đơn.

Method	*Return Value*	*Map*
isEmpty()	true	$\{\}$
put(5,A)	null	$\{(5,A)\}$
put(7,B)	null	$\{(5,A),(7,B)\}$
put(2,C)	null	$\{(5,A),(7,B),(2,C)\}$
put(8,D)	null	$\{(5,A),(7,B),(2,C),(8,D)\}$
put(2,E)	C	$\{(5,A),(7,B),(2,E),(8,D)\}$
get(7)	B	$\{(5,A),(7,B),(2,E),(8,D)\}$
get(4)	null	$\{(5,A),(7,B),(2,E),(8,D)\}$
get(2)	E	$\{(5,A),(7,B),(2,E),(8,D)\}$
size()	4	$\{(5,A),(7,B),(2,E),(8,D)\}$
remove(5)	A	$\{(7,B),(2,E),(8,D)\}$
remove(2)	E	$\{(7,B),(8,D)\}$
get(2)	null	$\{(7,B),(8,D)\}$
remove(2)	null	$\{(7,B),(8,D)\}$
isEmpty()	false	$\{(7,B),(8,D)\}$
entrySet()	$\{(7,B),(8,D)\}$	$\{(7,B),(8,D)\}$
keySet()	$\{7,8\}$	$\{(7,B),(8,D)\}$
values()	$\{B,D\}$	$\{(7,B),(8,D)\}$

CMC UNIVERSITY

Giao diện Java cho map

```
1  public interface Map<K,V> {
2    int size( );
3    boolean isEmpty( );
4    V get(K key);
5    V put(K key, V value);
6    V remove(K key);
7    Iterable<K> keySet( );
8    Iterable<V> values( );
9    Iterable<Entry<K,V>> entrySet( );
10 }
```


CMC UNIVERSITY
THANK YOU