**TRƯỜNG ĐẠI HỌC LAO ĐỘNG - XÃ HỘI**

**KHOA CÔNG NGHỆ THÔNG TIN**

**------------------------------------**

****

**TIỂU LUẬN HỌC PHẦN**

**CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT**

**Đề tài: Nghiên cứu ngăn xếp và thuật toán sắp xếp chèn**

**Sinh viên thực hiện: Đinh Hoàng Quốc Vương**

**Mã sinh viên: 1117093029**

**Lớp học phần: CTDLVGT1223L\_D17CN.5\_LT**

**HÀ NỘI, 2023**

DANH SÁCH SINH VIÊN THAM GIA BÁO CÁO

**NĂM** **HỌC** **2022** **-** **2023**

|  |  |  |
| --- | --- | --- |
| **HỌ** **VÀ** **TÊN** | **MÃ** **SINH** **VIÊN** | **CHỨC** **VỤ** |
| Đinh Hoàng Quốc Vương | 1117093029 | Nhóm trưởng |
| Bùi Đức Trọng | 1117093054 | Thành viên nhóm |

2

###### Mục lục

[Danh mục hình ảnh 4](#_bookmark1)

1. [Stack 5](#_bookmark2)

[Lời mở đầu 5](#_bookmark3)

[Chương 1: Stack 6](#_bookmark5)

* 1. [Định nghĩa 6](#_bookmark4)
  2. [The Stack Abstract Data Type (ADT) 7](#_bookmark7)
  3. [Giao diện của Stack trong java(Stack Interface) 11](#_bookmark10)
  4. [Implementation of Stack 13](#_bookmark11)
     1. [Simple Array Implementation 13](#_bookmark12)
     2. [Dynamic array-based implementation 16](#_bookmark13)
     3. [Linked List Implementation 21](#_bookmark14)

[So sánh Triển khai bằng Mảng và Triển khai bằng Danh sách Liên kết 23](#_bookmark15)

* 1. [Ưu điểm và nhược điểm của Stack 24](#_bookmark16)

[Chương 2: Các bài toán ứng dụng. So sánh giữa Stack và Queue 26](#_bookmark17)

1. [Các bài toán ứng dụng 26](#_bookmark18)
   1. [Đảo ngược một mảng bằng cách sử dụng ngăn xếp 26](#_bookmark19)
   2. [Kiểm tra dấu ngoặc 27](#_bookmark20)
2. [So sánh Stack và Queue 31](#_bookmark21)

[Bảng so sánh 31](#_bookmark22)

[Tổng kết 33](#_bookmark23)

1. [Insertion Sort 34](#_bookmark24)

[Lời mở đầu 34](#_bookmark25)

[Chương 1: Insertion Sort 35](#_bookmark26)

* 1. [Định nghĩa 35](#_bookmark27)
  2. [Thuật toán sắp xếp chèn 35](#_bookmark28)
  3. [Cách triển khai thuật toán sắp xếp chèn 36](#_bookmark31)
  4. [Biến thế của sắp xếp chèn (Binary Insertion Sort, Two-way Insertion Sort) 37](#_bookmark32)

[6. So sánh Insertion Sort với các thuật toán sắp xếp khác 40](#_bookmark34)

[Tổng kết 41](#_bookmark36)

[Tài liệu tham khảo 42](#_bookmark37)

1. [Stack (abstract data type) - wikipedia 42](#_bookmark38)

3

Danh mục hình ảnh

[Hình 1:Sơ đồ minh họa một ngăn xếp và các hoạt động diễn ra trên ngăn xếp. 6](#_bookmark6)

[Hình 2:Ví dụ về cách thức hoạt động của các phương thức trong stack 10](#_bookmark8)

[Hình 3:Phương thức của Stack ADT so với các phương thức tương ứng của lớp java.util.Stack 11](#_bookmark9)

[Hình 4:Giao diện của Stack trong Java 13](https://d.docs.live.net/9a44bf178ded79c4/Documents/B%C3%A1o%20c%C3%A1o%20c%E1%BA%A5u%20tr%C3%BAc%20d%E1%BB%AF%20li%E1%BB%87u%20v%C3%A0%20gi%E1%BA%A3i%20thu%E1%BA%ADt.docx#_Toc153662027)

[Hình 5:Một phương thức tổng quát đảo ngược các phần tử trong một mảng với các đối tượng](https://d.docs.live.net/9a44bf178ded79c4/Documents/B%C3%A1o%20c%C3%A1o%20c%E1%BA%A5u%20tr%C3%BAc%20d%E1%BB%AF%20li%E1%BB%87u%20v%C3%A0%20gi%E1%BA%A3i%20thu%E1%BA%ADt.docx#_Toc153662028) [kiểu E, sử dụng một ngăn xếp được khai báo với giao diện Stack<E> làm kiểu của nó. 26](https://d.docs.live.net/9a44bf178ded79c4/Documents/B%C3%A1o%20c%C3%A1o%20c%E1%BA%A5u%20tr%C3%BAc%20d%E1%BB%AF%20li%E1%BB%87u%20v%C3%A0%20gi%E1%BA%A3i%20thu%E1%BA%ADt.docx#_Toc153662028)

[Hình 6:Kiểm tra phương thức đảo ngược 27](https://d.docs.live.net/9a44bf178ded79c4/Documents/B%C3%A1o%20c%C3%A1o%20c%E1%BA%A5u%20tr%C3%BAc%20d%E1%BB%AF%20li%E1%BB%87u%20v%C3%A0%20gi%E1%BA%A3i%20thu%E1%BA%ADt.docx#_Toc153662029)

[Hình 7:Phương thức kiểm tra các ký tự phân định trong biểu thức toán học 29](https://d.docs.live.net/9a44bf178ded79c4/Documents/B%C3%A1o%20c%C3%A1o%20c%E1%BA%A5u%20tr%C3%BAc%20d%E1%BB%AF%20li%E1%BB%87u%20v%C3%A0%20gi%E1%BA%A3i%20thu%E1%BA%ADt.docx#_Toc153662030)

[Hình 8:Thuật toán sắp xếp chèn 35](#_bookmark29)

[Hình 9:Ví dụ về thuật toán sắp xếp chèn 36](#_bookmark30)

[Hình 10:Thuật toán sắp xếp chèn trong java 37](https://d.docs.live.net/9a44bf178ded79c4/Documents/B%C3%A1o%20c%C3%A1o%20c%E1%BA%A5u%20tr%C3%BAc%20d%E1%BB%AF%20li%E1%BB%87u%20v%C3%A0%20gi%E1%BA%A3i%20thu%E1%BA%ADt.docx#_Toc153662033)

[Hình 11:Sắp xếp chèn hai chiều 39](#_bookmark33)

[Hình 12:Bảng so sánh các thuật toán sắp xếp 40](#_bookmark35)

4

# A. Stack

### Lời mở đầu

Trong thế giới công nghệ hiện đại, việc quản lý thông tin và xử lý dữ liệu đang trở thành một thách thức ngày càng lớn. Để đáp ứng nhu cầu này, các nhà phát triển và kỹ sư thông tin đã tìm ra nhiều giải pháp hiệu quả, trong đó có cấu trúc dữ liệu "Stack" - một thành phần quan trọng trong lĩnh vực khoa học máy tính.

Ngăn xếp (Stack) là một trong số những cấu trúc dữ liêụ cực kỳ quan trọng,

được sử dụng thường xuyên trong thiết kế thuâṭ toán. Chính máy tính c甃̀ng sử dụng

nhiều ứng dụng của ngăn xếp (chẳng hạn như viêc̣ quản lý bô ̣ nhớ trong khi thi hành

chương trình, hay lưu trữ các lời gọi đê ̣ quy,...). Về bản chất, ngăn xếp c甃̀ng giống

như mảng, chúng là môṭ có tính thứ tự.

tâp̣

hợp các phần tử cùng kiểu dữ liêụ , nhưng được lưu trữ

Trong bản báo cáo này, chúng em sẽ giới thiêụ tới thầy cô và các bạn về hoạt

đông của ngăn xếp, cũng như cách cài đăṭ nó. Thông qua bài báo cáo này, chúng ta

sẽ được dẫn dắt qua các khía cạnh khác nhau của stack, từ các đặc điểm cơ bản đến các ứng dụng thực tế. Bằng cách này, chúng ta có thể hiểu rõ hơn về sức mạnh và tính linh hoạt mà stack mang lại trong ngữ cảnh của ngôn ngữ lập trình và quản lý

dữ liệu. Ngoài ra, chúng ta cũng s cùng xem xét môṭ số bài toán ứng dụng cấu trúc

dữ liêụ này để hiểu rõ hơn

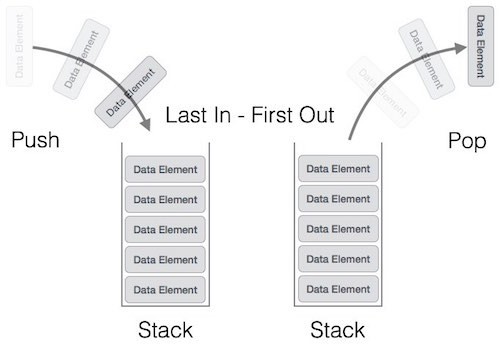
5

#### Định nghĩa

## Chương 1: Stack

Stack là một cấu trúc dữ liệu tuyến tính tuân theo một thứ tự cụ thể trong đó các hoạt động được thực hiện. Thứ tự có thể là LIFO(Last In First Out) hoặc FILO(First In Last Out). LIFO ngụ ý rằng phần tử được chèn cuối cùng s攃̀ xuất hiện trước và FILO ngụ ý rằng phần tử được chèn trước s攃̀ xuất hiện sau cùng. Người dùng có thể chèn các đối tượng vào Stack bất cứ lúc nào, nhưng chỉ có thể truy cập hoặc loại bỏ đối tượng được chèn gần đây nhất (ở đầu "đỉnh" của ngăn xếp).

Tên “Stack” được lấy từ ph攃Āp ẩn dụ về một ngăn xếp đĩa trong máy phát đĩa đĩa ở căng cơ, nơi các hoạt động cơ bản liên quan đến việc "đẩy vào" và "đẩy ra" đĩa trên ngăn xếp. Khi chúng ta cần một đĩa mới từ máy phát, chúng ta "đẩy" đĩa ở đầu ngăn xếp ra khỏi và khi chúng ta thêm một đĩa, chúng ta "đẩy" nó vào ngăn xếp để trở thành đĩa mới ở đầu.



*Hình* *1:Sơ* *đồ* *minh* *họa* *một* *ngăn* *xếp* *và* *các* *hoạt* *động* *diễn* *ra* *trên* *ngăn* *xếp.*

6

Ngăn xếp là một cấu trúc dữ liệu cơ bản được sử dụng trong nhiều ứng dụng,ví dụ như:

* Trình duyệt web lưu trữ địa chỉ của các trang web mà người dùng đã truy cập gần đây trên một ngăn xếp. Mỗi khi người dùng truy cập một trang web mới, địa chỉ của trang đó được "push" lên đỉnh ngăn xếp địa chỉ. Trình duyệt sau đó cho ph攃Āp người dùng "pop" để quay lại các trang đã truy cập trước đó bằng cách sử dụng nút "back".
* Các trình soạn thảo văn bản thường cung cấp một cơ chế "undo" cho việc hủy bỏ các hoạt động chỉnh sửa gần đây và quay về trạng thái trước đó của một tài liệu. Hoạt động "undo" này có thể được thực hiện bằng cách giữ các thay đổi văn bản trong một ngăn xếp.

#### The Stack Abstract Data Type (ADT)

Stack là một trong những cấu trúc dữ liệu đơn giản nhất, nhưng c甃̀ng là một trong những cấu trúc quan trọng nhất, vì chúng được sử dụng trong nhiều ứng dụng khác nhau và là công cụ hỗ trợ cho nhiều cấu trúc dữ liệu và thuật toán phức tạp hơn. Cụ thể, Stack là một kiểu dữ liệu trừu tượng (ADT) hỗ trợ hai phương thức cập nhật sau:

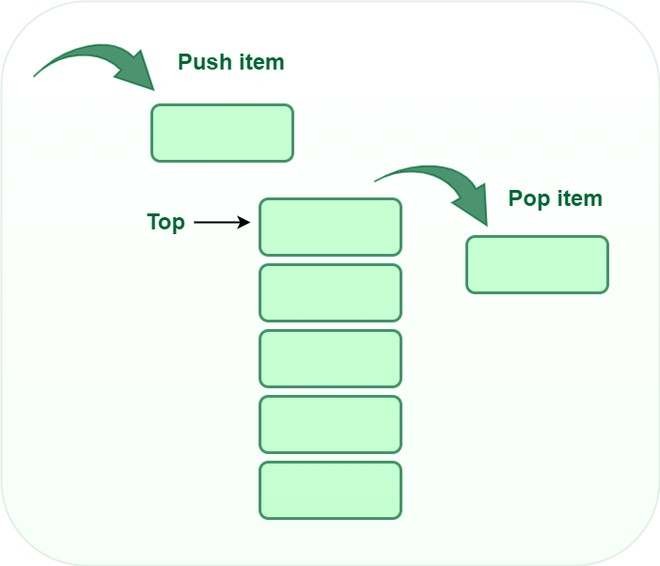
* push(e): Thêm phần tử e vào đỉnh của ngăn xếp.
* pop(): Loại bỏ và trả về phần tử ở đỉnh của ngăn xếp (hoặc trả về null nếu ngăn xếp là trống).

Ngoài ra, một Stack hỗ trợ các phương thức truy cập sau để thuận tiện:

* top():Trả về phần tử ở đỉnh của ngăn xếp mà không loại bỏ nó (hoặc trả về null nếu ngăn xếp là trống).
* size(): Trả về số lượng phần tử trong ngăn xếp.
* isEmpty(): Trả về giá trị boolean chỉ ra xem ngăn xếp có trống hay không.

Theo quy ước, chúng ta giả định rằng các phần tử được thêm vào Stack có thể là bất kỳ loại dữ liệu nào và một Stack mới được tạo ra là trống.

7



Để có thể hiểu rõ hơn về các phương thức, chúng ta đi vào mã giả của chúng:

##### Push

Thêm một mục vào ngăn xếp. Nếu ngăn xếp đầy thì đó được gọi là Overflow Condition.

Mã giả của phương thức Push:

begin

if stack is full return

endif else

increment top stack[top] assign value

end else end procedure

8

##### Pop

Xóa một mục khỏi ngăn xếp. Các mục được xuất hiện theo thứ tự đảo ngược khi chúng được “push”. Nếu ngăn xếp trống thì nó được gọi là Underflow condition.

Mã giả cho phương thức Pop:

begin

if stack is empty

return

endif else

store value of stack[top] decrement top

return value end else

end procedure

##### Top

Trả về phần tử trên cùng của ngăn xếp Mã giả của phương thức top:

begin

return stack[top] end procedure

9

##### isEmpty

Trả về giá trị true nếu ngăn xếp trống, ngược lại trả về giá trị false Mã giả của phương thức isEmpty:

begin

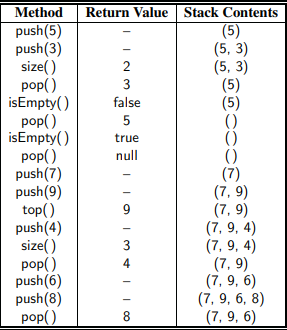
if top < 1

return true

else

return false end procedure

Để hiểu rõ hơn về cách thức hoạt động của các phương thức trong stack, chúng ta có thể nhìn qua ví dụ ở hình 2



*Hình* *2:Ví* *dụ* *về* *cách* *thức* *hoạt* *động* *của* *các* *phương* *thức* *trong* *stack*

10

Bởi vì sự quan trọng của Stack ADT, Java đã bao gồm, từ những phiên bản ban đầu của nó, một lớp cụ thể có tên là java.util.Stack để có thể triển khai các hành vi LIFO của một ngăn xếp. Tuy nhiên, lớp Stack của Java chỉ tồn tại chủ yếu vì lịch sử, và giao diện của nó không nhất quán với hầu hết các cấu trúc dữ liệu khác trong thư viện Java. Thực tế, tài liệu hiện tại về Stack khuyến cáo rằng nên tránh sử dụng nó, vì chức năng LIFO (và nhiều chức năng khác) của nó đã được cung cấp bởi một cấu trúc dữ liệu khác tổng quát hơn, được biết đến là hàng đợi đối xứng(a double-ended queue)

Để so sánh, hình 3 cung cấp một so sánh song song giữa giao diện của Stack ADT của chúng ta và lớp java.util.Stack. Ngoài một số khác biệt về tên phương thức, chúng ta chú ý rằng phương thức pop và peek của lớp java.util.Stack n攃Ām một ngoại lệ EmptyStackException tùy chỉnh nếu được gọi khi ngăn xếp rỗng (trong khi trong lớp abstraction của chúng ta, giá trị được trả về là giá trị null).

|  |  |
| --- | --- |
| **Out** **Stack** **ADT** | **Class** **java.util.Stack** |
| size() | size() |
| isEmpty() | empty() |
| push(e) | push(e) |
| pop() | pop() |
| top() | peek() |

*Hình* *3:Phương* *thức* *của* *Stack* *ADT* *so* *với* *các* *phương* *thức* *tương* *ứng* *của* *lớp* *java.util.Stack*

#### Giao diện của Stack trong java(Stack Interface)

Để hình thành trừu tượng của chúng ta về một ngăn xếp, chúng ta định nghĩa điều được gọi là giao diện lập trình ứng dụng (API) của nó dưới dạng một giao diện Java, mô tả tên của các phương thức mà ADT hỗ trợ và cách chúng được khai báo và sử dụng. Giao diện này được biểu diễn trong hình 4 ở phía dưới

Chúng ta sử dụng framework generics của Java, cho ph攃Āp các phần tử được lưu trữ trong ngăn xếp thuộc bất kỳ loại đối tượng <E> nào. Ví dụ, một biến đại diện cho một ngăn xếp số nguyên có thể được khai báo với kiểu Stack<Integer>. Tham số kiểu chính thức được sử dụng làm kiểu tham số cho phương thức push và kiểu trả về cho cả phương thức pop và top.

Ta được biết rằng giao diện đóng vai trò như một định nghĩa kiểu dữ liệu nhưng nó không thể được khởi tạo trực tiếp. Đối với ADT có bất kỳ ý nghĩa nào,

11

chúng ta phải cung cấp một hoặc nhiều lớp cụ thể thực hiện các phương thức của giao diện liên quan đến ADT đó. Trong các phần tiếp theo, chúng ta s攃̀ cung cấp hai cách triển khai của giao diện Stack: một sử dụng một mảng để lưu trữ và một cái khác sử dụng một danh sách liên kết.

12

27 ∗/

28 void push(E e); 29

30 /∗∗

1. ∗ Returns, but does not remove, the element at the top of the stack.
2. ∗ @return top element in the stack (or null if empty)
3. ∗/
4. E top( ); 35

36 /∗∗

1. ∗ Removes and returns the top element from the stack.
2. ∗ @return element removed (or null if empty)
3. ∗/
4. E pop( );
5. }

*Hình* *4:Giao* *diện* *của* *Stack* *trong* *Java*

#### Implementation of Stack

Có nhiều cách để triển khai Abstract Data Type (ADT) ngăn xếp, dưới đây là những phương pháp phổ biến thường được sử dụng:

* Triển khai dựa trên mảng đơn giản (Simple array-based implementation).
* Triển khai dựa trên mảng động (Dynamic array-based implementation).
* Triển khai dựa trên danh sách liên kết (Linked list implementation).

##### Simple Array Implementation

Triển khai này của Abstract Data Type (ADT) ngăn xếp sử dụng một mảng. Trong mảng này, chúng ta thêm các phần tử từ trái sang phải và sử dụng một biến để theo dõi chỉ số của phần tử đỉnh.



Mảng lưu trữ các phần tử của ngăn xếp có thể trở nên đầy. Một ph攃Āp đẩy (push) khi đó s攃̀ gây ra ngoại lệ "full stack". Tương tự, nếu chúng ta cố gắng xóa một phần tử từ một ngăn xếp trống, nó s攃̀ gây ra ngoại lệ "empty stack".

13

public class FixedSizeArrayStack<E> implements Stack<E>{

// Length of the array used to implement the stack. protected int capacity;

// Default array capacity.

public static final int CAPACITY = 10;

// Array used to implement the stack. protected E[] stackRep;

// Index of the top element of the stack in the array. protected int top = -1;

// Initializes the stack to use an array of default length. public FixedSizeArrayStack() {

this(CAPACITY); // default capacity

}

// Initializes the stack to use an array of given length. public FixedSizeArrayStack(int cap) {

capacity = cap;

stackRep = (E[])new Object[capacity]; // compiler may give warning, but this is ok

}

// Returns the number of elements in the stack. This method runs in O(1) time. public int size() {

return (top + 1);

}

// Testes whether the stack is empty. This method runs in O(1) time. public boolean isEmpty() {

return (top < 0);

}

// Inserts an element at the top of the stack. This method runs in O(1) time. public void push(E data) throws Exception {

if (size() == capacity)

throw new Exception("Stack is full.");

14

stackRep[++top] = data;

}

// Inspects the element at the top of the stack. This method runs in O(1) time. public E top() throws Exception {

if (isEmpty())

throw new Exception("Stack is empty."); return stackRep[top];

}

// Removes the top element from the stack. This method runs in O(1) time. public E pop() throws Exception {

E data;

if (isEmpty())

throw new Exception("Stack is empty."); data = stackRep[top];

stackRep[top--] = Integer.MIN\_VALUE; return data;

}

// Returns a string representation of the stack as a list of elements, with

// the top element at the end: [ ... , prev, top ]. This method runs in O(n)

// time, where n is the size of the stack. public String toString() {

String s; s = "[";

if (size() > 0)

s += stackRep[0]; if (size() > 1)

for (int i = 1; i <= size() - 1; i++) { s += ", " + stackRep[i];

}

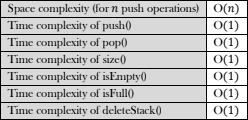
return s + "]";

}

}

15

Hiệu suất: Cho n là số lượng phần tử trong ngăn xếp.Độ phức tạp của các hoạt động trong ngăn xếp với biểu diễn này có thể được mô tả như hình sau:



Ưu điểm của việc triển khai mảng:

* Dễ để thực hiện.
* Bộ nhớ được lưu dưới dạng con trỏ không liên quan. Nhược điểm của việc triển khai mảng:
* Nó không năng động, tức là nó không phát triển và thu nhỏ theo nhu cầu trong thời gian chạy. [Nhưng trong trường hợp các mảng có kích thước động như vector trong C++, danh sách trong Python, ArrayList trong Java, các ngăn xếp c甃̀ng có thể tăng lên và thu nhỏ khi triển khai mảng].
* Tổng kích thước của ngăn xếp phải được xác định trước.

##### Dynamic array-based implementation

Đầu tiên, hãy xem x攃Āt cách chúng ta triển khai một ngăn xếp dựa trên mảng đơn giản. Chúng ta sử dụng một biến chỉ mục top, trỏ đến chỉ số của phần tử được chèn gần đây nhất trong ngăn xếp. Để chèn (hoặc đẩy) một phần tử, chúng ta tăng chỉ mục top lên một và sau đó đặt phần tử mới tại chỉ mục đó.

Tương tự, để xóa (hoặc đẩy ra) một phần tử, chúng ta lấy phần tử tại chỉ mục top và sau đó giảm giá trị của chỉ mục top. Chúng ta giả dụ là một ngăn xếp rỗng khi giá trị của top bằng -1. Vấn đề vẫn cần được giải quyết là chúng ta s攃̀ làm gì khi tất cả các ô trong ngăn xếp mảng có kích thước cố định đều đã được chiếm đủ chỗ?

Nếu chúng ta tăng kích thước của mảng lên 1 mỗi khi ngăn xếp đầy? push(): tăng kích thước của S[] lên 1

pop(): giảm kích thước của S[] xuống 1

***Vấn*** ***đề*** ***với*** ***cách*** ***tiếp*** ***cận*** ***này?***

16

Cách tiếp cận này với việc tăng kích thước mảng quá tốn k攃Ām. Hãy xem lý do. Ví dụ, tại n = 1, để đẩy một phần tử, chúng ta tạo ra một mảng mới có kích thước 2 và sao ch攃Āp tất cả các phần tử c甃̀ vào mảng mới, cuối cùng thêm phần tử mới. Ở n = 2, để đẩy một phần tử, chúng ta tạo ra một mảng mới có kích thước 3 và sao ch攃Āp tất cả các phần tử c甃̀ vào mảng mới, cuối cùng thêm phần tử mới.

Tương tự, ở n = n - 1, nếu chúng ta muốn đẩy một phần tử, chúng ta tạo ra một mảng mới có kích thước n và sao ch攃Āp tất cả các phần tử c甃̀ vào mảng mới và cuối cùng thêm phần tử mới. Sau n thao tác đẩy, tổng thời gian T(n) (số lần sao ch攃Āp) tăng theo tỉ lệ với 1 + 2 + ... + n ≈ O(n2).

***Cách*** ***tiếp*** ***cận*** ***thay*** ***thế:*** ***Tăng*** ***lên*** ***Lần*** ***Lặp***

Hãy cải thiện độ phức tạp bằng cách sử dụng kỹ thuật nhân đôi mảng. Nếu mảng đầy, tạo ra một mảng mới có kích thước gấp đôi và sao ch攃Āp các mục. Với cách tiếp cận này, đẩy n mục mất thời gian tỉ lệ với n (không cần đến n2).

Để đơn giản, giả sử ban đầu chúng ta bắt đầu với n = 1 và tăng lên n = 32. Điều này có nghĩa là chúng ta thực hiện việc nhân đôi tại 1, 2, 4, 8, 16. Cách tiếp cận khác của việc phân tích là: tại n = 1, nếu chúng ta muốn thêm (đẩy) một phần tử, nhân đôi kích thước hiện tại của mảng và sao ch攃Āp tất cả các phần tử của mảng c甃̀ vào mảng mới.

Tại n = 1, chúng ta thực hiện 1 thao tác sao ch攃Āp, tại n = 2, chúng ta thực hiện 2 thao tác sao ch攃Āp và tại n = 4, chúng ta thực hiện 4 thao tác sao ch攃Āp và cứ thế. Đến khi chúng ta đạt được n = 32, tổng số thao tác sao ch攃Āp là 1 + 2 + 4 + 8 + 16 = 31, gần bằng giá trị của 2n (32). Nếu chúng ta quan sát cẩn thận, chúng ta thực hiện ph攃Āp nhân đôi mảng logn lần. Bây giờ, hãy tổng quát hóa cuộc thảo luận. Đối với n thao tác đẩy, chúng ta nhân đôi kích thước mảng logn lần. Điều này có nghĩa là, chúng ta s攃̀ có logn thành phần trong biểu thức dưới đây. Tổng thời gian T(n) của một chuỗi n thao tác đẩy tỉ lệ với

1 + 2 + 4 + 8 +…n/4 + n/2 + n = n+ n/2 + n/4 + n/8 +…+ 4 + 2 + 1

= n(1 + ½ + ¼ + 1/8 +…+ 4/n + 2/n +1/n)

= n(2) ≈ 2n = O(n)

T(n) là O(n) và thời gian gộp của một thao tác đẩy (amortized time) là O(1).

17

public class DynamicArrayStack{

// Length of the array used to implement the stack. protected int capacity;

// Default array capacity.

public static final int CAPACITY = 16; // power of 2 public static int MINCAPACITY=1<<15; // power of 2

// Array used to implement the stack. protected int[] stackRep;

// Index of the top element of the stack in the array. protected int top = -1;

// Initializes the stack to use an array of default length. public DynamicArrayStack() {

this(CAPACITY); // default capacity

}

// Initializes the stack to use an array of given length. public DynamicArrayStack(int cap) {

capacity = cap;

stackRep = new int[capacity]; // compiler may give warning, but this is ok

}

// Returns the number of elements in the stack. This method runs in O(1) time. public int size() {

return (top + 1);

}

// Testes whether the stack is empty. This method runs in O(1) time. public boolean isEmpty() {

return (top < 0);

}

// Inserts an element at the top of the stack. This method runs in O(1) time. public void push(int data) throws Exception {

18

if (size() == capacity) expand(); stackRep[++top] = data;

}

private void expand() {

int length = size();

int[] newstack=new int[length<<1]; System.arraycopy(stackRep,0,newstack,0,length); stackRep=newstack;

this.capacity = this.capacity<<1;

}

// dynamic array operation: shrinks to 1/2 if more than than 3/4 empty 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(stackRep,0,newstack,0,top+1); stackRep=newstack;

this.capacity = length;

}

// Inspects the element at the top of the stack. This method runs in O(1) time. public int top() throws Exception {

If (isEmpty())

throw new Exception("Stack is empty."); return stackRep[top];

}

// Removes the top element from the stack. This method runs in O(1) time. public int pop() throws Exception {

19

int data;

if (isEmpty())

throw new Exception("Stack is empty."); data = stackRep[top];

stackRep[top--] = Integer.MIN\_VALUE; // dereference S[top] for garbage collection.

shrink(); return data;

}

// Returns a string representation of the stack as a list of elements, with

// the top element at the end: [ ... , prev, top ]. This method runs in O(n)

// time, where n is the size of the stack. public String toString() {

String s; s = "[";

if (size() > 0)

s += stackRep[0]; if (size() > 1)

for (int i = 1; i <= size() - 1; i++) { s += ", " + stackRep[i];

}

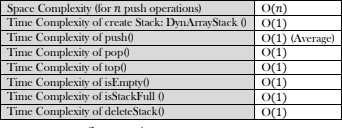
return s + "]";

}

}

20

Hiệu suất: Cho n là số lượng phần tử trong ngăn xếp.Độ phức tạp của các hoạt động trong ngăn xếp với biểu diễn này có thể được mô tả như hình sau:



Chú ý: Quá nhiều lần nhân đôi có thể gây ra ngoại lệ tràn bộ nhớ.

##### Linked List Implementation

Cách triển khai khác của ngăn xếp là sử dụng danh sách liên kết(Linked list). Thao tác đẩy (push) được triển khai bằng cách chèn phần tử vào đầu danh sách. Thao tác đẩy ra (pop) được triển khai bằng cách xóa nút từ đầu danh sách (nút đầu/top).



public class LinkedStack<T>{

private int length; // indicates the size of the linked list private ListNode top;

// Constructor: Creates an empty stack. public LinkedStack() {

length = 0; top = null;

}

// Adds the specified data to the top of this stack. public void push (int data) {

ListNode temp = new ListNode (data); temp.next = top;

top = temp;

21

length++;

}

// Removes the data at the top of this stack and returns a

// reference to it. Throws an EmptyStackException if the stack

// is empty.

public int pop() throws EmptyStackException{ if (isEmpty())

throw new EmptyStackException(); int result = top.data;

top = top.next; length--; return result;

}

// Returns a reference to the data at the top of this stack.

// The data is not removed from the stack. Throws an

// EmptyStackException if the stack is empty. public int peek() throws EmptyStackException{

if (isEmpty())

throw new EmptyStackException(); return top.data;

}

// Returns true if this stack is empty and false otherwise. public boolean isEmpty(){

return (length == 0);

}

// Returns the number of elements in the stack. public int size(){

return length;

}

22

// Returns a string representation of this stack. public String toString(){

String result = ""; ListNode current = top; while (current != null){

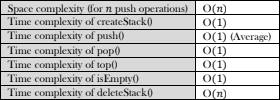
result = result + current.toString() + "\n"; current = current.next;

}

return result;

}

Hiệu suất: Cho n là số lượng phần tử trong ngăn xếp.Độ phức tạp của các hoạt động trong ngăn xếp với biểu diễn này có thể được mô tả như hình sau:



Ưu điểm của việc triển khai Danh sách liên kết:

* Việc triển khai danh sách liên kết của ngăn xếp có thể tăng hoặc giảm tùy theo nhu cầu trong thời gian chạy.
* Nó được sử dụng trong nhiều máy ảo như JVM. Nhược điểm của việc triển khai Danh sách liên kết:
* Yêu cầu thêm bộ nhớ do có sự tham gia của con trỏ.
* Không thể truy cập ngẫu nhiên trong ngăn xếp.

##### So sánh Triển khai bằng Mảng và Triển khai bằng Danh sách Liên kết

23

Triển khai Mảng

* Các ph攃Āp toán mất thời gian hằng số.
* Đôi khi các ph攃Āp toán nhân đôi(nhân đôi kích thước mảng)xảy ra
* Bất kỳ chuỗi n ph攃Āp toán nào (bắt đầu từ ngăn xếp rỗng) - "hòm chứa có giới hạn" mất thời gian tỷ lệ với n.

Triển khai Danh sách Liên kết

* Mở rộng và co lại một cách linh hoạt.
* Mọi ph攃Āp toán mất thời gian hằng số O(1).
* Mọi ph攃Āp toán sử dụng thêm không gian và thời gian để làm việc với các tham chiếu.

#### Ưu điểm và nhược điểm của Stack

Ưu điểm của Stack:

* Dễ triển khai: Cấu trúc dữ liệu ngăn xếp dễ triển khai bằng cách sử dụng mảng hoặc danh sách liên kết, đồng thời các thao tác của nó rất dễ hiểu và dễ thực hiện.
* Sử dụng bộ nhớ hiệu quả: Ngăn xếp sử dụng khối bộ nhớ liền kề, giúp việc sử dụng bộ nhớ hiệu quả hơn so với các cấu trúc dữ liệu khác.
* Thời gian truy cập nhanh: Cấu trúc dữ liệu ngăn xếp cung cấp thời gian truy cập nhanh để thêm và xóa các phần tử khi các phần tử được thêm vào và xóa khỏi đầu ngăn xếp.
* Trợ giúp trong lệnh gọi hàm: Cấu trúc dữ liệu ngăn xếp được sử dụng để lưu trữ các lệnh gọi hàm và trạng thái của chúng, giúp triển khai hiệu quả các lệnh gọi hàm đệ quy.
* Hỗ trợ quay lui: Cấu trúc dữ liệu ngăn xếp hỗ trợ các thuật toán quay lui, được sử dụng trong giải quyết vấn đề để khám phá tất cả các giải pháp có thể có bằng cách lưu trữ các trạng thái trước đó.
* Được sử dụng trong Thiết kế trình biên dịch: Cấu trúc dữ liệu ngăn xếp được sử dụng trong thiết kế trình biên dịch để phân tích cú pháp và phân tích cú pháp của các ngôn ngữ lập trình.
* Cho ph攃Āp các thao tác hoàn tác/làm lại: Cấu trúc dữ liệu ngăn xếp được sử dụng để cho ph攃Āp các thao tác hoàn tác và làm lại trong nhiều ứng dụng khác nhau như trình soạn thảo văn bản, công cụ thiết kế đồ họa và môi trường phát triển phần mềm.

Nhược điểm của Stack:

* Dung lượng hạn chế: Cấu trúc dữ liệu ngăn xếp có dung lượng hạn chế vì nó chỉ có thể chứa một số phần tử cố định. Nếu ngăn xếp đầy, việc thêm phần tử mới có thể dẫn đến tràn ngăn xếp, dẫn đến mất dữ liệu.

24

* Không có quyền truy cập ngẫu nhiên: Cấu trúc dữ liệu ngăn xếp không cho ph攃Āp truy cập ngẫu nhiên vào các phần tử của nó và chỉ cho ph攃Āp thêm và xóa các phần tử khỏi đầu ngăn xếp. Để truy cập một phần tử ở giữa ngăn xếp, tất cả các phần tử phía trên nó phải bị xóa.
* Quản lý bộ nhớ: Cấu trúc dữ liệu ngăn xếp sử dụng khối bộ nhớ liền kề, điều này có thể dẫn đến phân mảnh bộ nhớ nếu các phần tử được thêm vào và xóa thường xuyên.
* Không phù hợp với một số ứng dụng nhất định: Cấu trúc dữ liệu ngăn xếp không phù hợp với các ứng dụng yêu cầu truy cập các phần tử ở giữa ngăn xếp, như thuật toán tìm kiếm hoặc sắp xếp.
* Tràn ngăn xếp và tràn dưới ngăn xếp: Cấu trúc dữ liệu ngăn xếp có thể dẫn đến tràn ngăn xếp nếu có quá nhiều phần tử được đẩy lên ngăn xếp và có thể dẫn đến tràn ngăn xếp nếu có quá nhiều phần tử được đưa ra từ ngăn xếp.
* Các hạn chế của lệnh gọi hàm đệ quy: Mặc dù cấu trúc dữ liệu ngăn xếp hỗ trợ các lệnh gọi hàm đệ quy, nhưng quá nhiều lệnh gọi hàm đệ quy có thể dẫn đến tràn ngăn xếp, dẫn đến việc chấm dứt chương trình.

25

## Chương 2: Các bài toán ứng dụng. So sánh giữa Stack và Queue

### Các bài toán ứng dụng

Stack có rất nhiều ứng dụng cụ thể. Dưới đây là một số thuật toán cơ bản mà sử dụng cấu trúc dữ liệu ngăn xếp (stack)

#### Đảo ngược một mảng bằng cách sử dụng ngăn xếp

Do cơ chế LIFO (Last In, First Out), một ngăn xếp có thể được sử dụng như một công cụ tổng quát để đảo ngược một chuỗi dữ liệu. Ví dụ, nếu các giá trị 1, 2 và 3 được đẩy vào một ngăn xếp theo thứ tự đó, chúng s攃̀ được lấy ra khỏi ngăn xếp theo thứ tự 3, 2 và sau đó là 1.

Chúng ta diễn tả ý tưởng này bằng cách quay lại vấn đề đảo ngược các phần tử của một mảng. Chúng ta tạo một ngăn xếp trống để lưu trữ phụ, đẩy tất cả các phần tử của mảng vào ngăn xếp, và sau đó đẩy các phần tử đó khỏi ngăn xếp trong khi ghi đè lên các ô của mảng từ đầu đến cuối. Trong hình 5, chúng ta cung cấp một cài đặt Java của thuật toán này. Chúng ta có thể xem một ví dụ về cách sử dụng phương pháp này trong hình 6.

1. /∗∗ A generic method for reversing an array. ∗/
2. public static <E> void reverse(E[ ] a) {

3

4

5

6

7

8 }

Stack<E> buffer = new ArrayStack<>(a.length); for (int i=0; i < a.length; i++)

buffer.push(a[i]);

for (int i=0; i < a.length; i++) a[i] = buffer.pop( );

*Hình* *5:Một* *phương* *thức* *tổng* *quát* *đảo* *ngược* *các* *phần* *tử* *trong* *một* *mảng* *với* *các* *đối* *tượng* *kiểu* *E,* *sử* *dụng* *một* *ngăn* *xếp* *được* *khai* *báo* *với* *giao* *diện* *Stack<E>* *làm* *kiểu* *của* *nó.*

26

**D**1**sf**/**d**∗∗**sf**T**d**ester routine for reversing arrays ∗/ 2 public static void main(String args[ ]) {

1. Integer[ ] a = {4, 8, 15, 16, 23, 42}; // autoboxing allows this
2. String[ ] s = {"Jack", "Kate", "Hurley", "Jin", "Michael"};
3. System.out.println("a = " + Arrays.toString(a));
4. System.out.println("s = " + Arrays.toString(s));
5. System.out.println("Reversing...");
6. reverse(a);
7. reverse(s);
8. System.out.println("a = " + Arrays.toString(a));
9. System.out.println("s = " + Arrays.toString(s));

12 }

The output from this method is the following: a = [1, 2, 3, 4, 5, 6, 7, 8, 9 ]

s = [HoangAnh, An, Dương, QuocAnh] Reversing...

a = [9, 8, 7, 6, 5, 4, 3, 2, 1]

s = [QuocAnh, Duong, An, HoangAnh]

*Hình* *6:Kiểm* *tra* *phương* *thức* *đảo* *ngược*

#### Kiểm tra dấu ngoặc

Trong ứng dụng này, chúng ta xem x攃Āt các biểu thức toán học có thể chứa các cặp ký tự nhóm khác nhau, như

* + - Dấu ngoặc: "(" và ")"
    - Dấu ngoặc nhọn: "{" và "}"
    - Dấu ngoặc vuông: "[" và "]"

Mỗi ký tự mở ngoặc phải phù hợp với ký tự đóng ngoặc tương ứng. Ví dụ, một dấu ngoặc vuông trái, "[," phải phù hợp với một dấu ngoặc vuông phải, "], " như trong biểu thức sau:

[(x+6).8-2.(7+2x)]

27

Để hiểu rõ hơn, chúng ta có các ví dụ sau đây:

* + - Đúng: ( )(( )){([( )])}
    - Đúng: ((( )(( )){([( )])}))
    - Không đúng: )(( )){([( )])}
    - Không đúng: ({[])}
    - Không đúng: (

Việc quan trọng nhất khi xử lý biểu thức toán học là đảm bảo các ký tự phân định của chúng phù hợp đúng cách. Chúng ta có thể sử dụng một ngăn xếp để thực hiện công việc này khi rà soát từ trái sang phải của chuỗi ban đầu.

Mỗi khi chúng ta gặp một ký tự mở ngoặc, chúng ta đẩy ký tự đó vào ngăn xếp, và mỗi khi chúng ta gặp một ký tự đóng ngoặc, chúng ta lấy ra một ký tự từ ngăn xếp (nếu ngăn xếp không rỗng) và kiểm tra xem hai ký tự này có tạo thành một cặp hợp lệ không. Nếu chúng ta đến cuối biểu thức và ngăn xếp là trống, thì biểu thức ban đầu đã phù hợp đúng cách. Ngược lại, có thể có một dấu mở ngoặc trên ngăn xếp mà không có ký tự phù hợp. Nếu độ dài của biểu thức ban đầu là n, thuật toán s攃̀ thực hiện tối đa n lần gọi hàm push và n lần gọi hàm pop. Hình 7 trình bày một cài đặt Java của một thuật toán như vậy. Nó kiểm tra cụ thể các cặp dấu phân định (

), { }, và [ ], nhưng có thể dễ dàng được thay đổi để phù hợp với các biểu thức khác. Cụ thể, chúng ta định nghĩa hai chuỗi cố định, "({[" và ")}]", được hiệu chỉnh một cách có ý đồ để phản ánh các cặp ký tự. Khi xem x攃Āt một ký tự của chuỗi biểu thức, chúng ta gọi phương thức indexOf của lớp String trên các chuỗi đặc biệt này để xác định xem ký tự có phù hợp với một dấu phân định và, nếu có, là dấu phân định nào. Phương thức indexOf trả về chỉ số mà một ký tự cụ thể được tìm thấy đầu tiên trong một chuỗi (hoặc −1 nếu ký tự không được tìm thấy).

1/∗∗ Tests if delimiters in the given expression are properly matched. ∗/

2 public static boolean isMatched(String expression) {

3

4

5

6

7

8

final String opening = "({["; // opening delimiters

final String closing = ")}]"; // respective closing delimiters Stack<Character> buffer = new LinkedStack<>( );

for (char c : expression.toCharArray( )) {

if (opening.indexOf(c) != −1) // this is a left delimiter buffer.push(c);

28

1. else if (closing.indexOf(c) != −1) { // this is a right delimiter
2. if (buffer.isEmpty( )) // nothing to match with
3. return false;
4. if (closing.indexOf(c) != opening.indexOf(buffer.pop( )))
5. return false; // mismatched delimiter

14 }

15 }

16 return buffer.isEmpty( ); // were all opening delimiters matched?

17 }

* 1. **Ứng** *H***d***ì***ụ***nh***n***7***g***:Ph***c***ư***ủ***ơ***a***ng* **S***th***t***ứ***a***c* **c***ki***k***ểm***v***tr***à***a***o***cá***c***c* **h***ký***ư***tự***ơ***p***n***hâ***g***n* **t***đ***r***ịn***ì***h***n***tr***h***on***n***g* *b***h***iể***ó***u***m***thức* *toán* *học*

Như chúng ta đã thấy ở trên, stack có rất nhiều chức năng c甃̀ng như ứng dụng đa dạng, có thể linh hoạt trong nhiều trường hợp. Vì thế, chúng em đã tận dụng tối đa tất cả các chức năng stack như 1 công cụ để triển khai chương trình nhóm. Cụ thể:

* + - Trong lớp Dragon, chúng em đã khai báo một biến ngăn xếp (stack) tên là weakness dùng để giữ thông tin về điểm yếu của con rồng trong mỗi lượt chơi. Chúng em còn tạo ra phương thức getWeakness() để trả về giá trị của biến weakness, tức là trả về ngăn xếp chứa các đối tượng thuộc lớp Symbols. Ngoài ra chúng em còn có 1 phương thức là revealWeakness() dùng để tạo ra và hiển thị một tập hợp ngẫu nhiên gồm 5 điểm yếu cho con rồng trong mỗi lượt chơi.

private Stack<Symbols> weakness = new Stack<>(); // store dragon weakness this turn public Stack<Symbols> getWeakness() {

return weakness;

}

public void revealWeakness() { weakness.clear(); // clear previous weakness

for (int i = 0; i < 5; i++) weakness.push(new Symbols((int) (Math.random() \* 6) + 1));

}

29

* + - Trong lớp Gojo, chúng em c甃̀ng đã khai báo một biến ngăn xếp (stack) tên là spells dùng để giữ thông tin về sức mạnh hay ph攃Āp thuật của nhân vật gojo. Chúng em còn tạo ra phương thức getSpells() để trả về giá trị của biến spells, tức là trả về ngăn xếp chứa các đối tượng thuộc lớp Symbols. Ngoài ra chúng em còn có 1 phương thức là castSpell() dùng để thêm một ph攃Āp thuật mới hoặc một sức mạnh mới vào ngăn xếp spells dựa trên tên của ký hiệu (symbolName).

private Stack<Symbols> spells = new Stack<>(); public Stack<Symbols> getSpells() {

return spells;

}

public void castSpell(String symbolName) {

spells.push(new Symbols(Integer.parseInt(symbolName.substring(6))));

}

Việc tạo ra 2 ngăn xếp trên là để so sánh các ký tự ở lớp Symbols. Nếu như các ký tự được lấy ra từ ngăn xếp spells của nhân vật Gojo trùng hết với ký tự được lấy ra từ ngăn xếp weakness của dragon thì nhân vật Gojo s攃̀ thực hiện hành động tấn công với dragon và ngược lại.

30

### So sánh Stack và Queue

Trước khi chúng ta so sánh stack và queue, chúng ta cần biết sơ qua về

queue. Queue là môṭ

cấu trúc dữ liêụ

biểu diễn môṭ

danh sách các phần tử đứng

trong "hàng chờ" được xử lý. Trong cấu trúc dữ liêụ

này, viêc̣

bổ sung môṭ

phần tử

được thực hiêṇ danh sách.

ở cuối danh sách, còn viêc̣

loại bỏ môṭ

phần tử được thực hiêṇ

ở đầu

Có thể tưởng tượng hàng đợi giống như môṭ hàng người xếp hàng chờ mua

v攃Ā, ai đến trước được mua trước và rời khỏi hàng, còn những người đến sau s攃̀ bổ

sung vào cuối hàng. Vì nguyên tắc "vào trước ra trước" như vâỵ được gọi là danh sách kiểu FIFO (First In First Out).

nên hàng đợi còn

Stack và Queue đều là các cấu trúc dữ liệu không nguyên thủy (non- primitive). Sự khác biệt lớn nhất giữa Stack và Queue là Stack sử dụng phương thức LIFO (last in first out) để truy cập và thêm các phần tử dữ liệu trong khi Queue sử dụng phương thức FIFO (First in first out) để truy cập và thêm các phần tử dữ liệu.

Stack chỉ có một đầu mở để pushing và popping các phần tử dữ liệu, còn Queue có cả hai đầu mở để enqueuing và dequeuing các phần tử dữ liệu.

Stack và Queue là các cấu trúc dữ liệu được sử dụng để lưu trữ các yếu tố dữ liệu và nó dựa trên một số các ví dụ có thực trong cuộc sống hàng ngày của chúng ta. Ví dụ, Stack là một chồng đĩa CD, nơi bạn có thể lấy ra và đưa vào đĩa CD thông qua đỉnh của ngăn xếp đĩa CD. Tương tự, Queue là hàng đợi cho các v攃Ā của Nhà hát nơi người đứng ở vị trí đầu tiên, nghĩa là, phía trước hàng đợi s攃̀ được phục vụ trước và người mới đến s攃̀ xuất hiện ở phía sau hàng đợi.

Bảng so sánh

|  |  |  |
| --- | --- | --- |
| **Cơ** **sở** **để** **so** **sánh** | **STACK** | **QUEUE** |
| Nguyên tắc làm việc | LIFO (Last in First out) | FIFO (First in First out) |
| Structure | Dùng một đầu để chèn và | Có 2 đầu để xử lý dữ liệu, một đầu |

31

|  |  |  |
| --- | --- | --- |
| **Cơ** **sở** **để** **so** **sánh** | **STACK** | **QUEUE** |
|  | xóa các phần tử dữ liệu | chèn một đầu xóa |
| Số con trỏ được sử dụng | Một | Hai (Trong trường hợp đơn giản) |
| Hoạt động được thực hiện | Push và Pop | Enqueue và dequeue |
| Kiểm tra empty condition | Top == -1 | Front == -1 |
| Examination full condition | Top == Max - 1 | Rear == Max - 1 |
| Biến thể | Không có biến thể | Nó có các biến thể như hàng đợi tròn, hàng đợi ưu tiên, hàng đợi kết thúc gấp đôi. |
| Thực hiện | Đơn giản | Tương đối phức tạp |

32

## Tổng kết

Trong bản báo cáo này, chúng ta đã xem x攃Āt và nghiên cứu về cấu trúc dữ liệu ngăn xếp và ứng dụng của nó trong lập trình. Ngăn xếp, với nguyên tắc hoạt động LIFO, đã chứng tỏ sự linh hoạt và hiệu quả trong nhiều tình huống khác nhau.

Chúng ta đã trình bày về các thuật toán và ứng dụng thường sử dụng ngăn xếp, bao gồm việc xử lý biểu thức toán học, đảo ngược mảng,…

Các phương thức cơ bản như push và pop đã được trình bày, cùng với các cách triển khai của ngăn xếp bằng mảng và danh sách liên kết. Thời gian phức tạp và ứng dụng thực tế của mỗi loại triển khai đã được xem x攃Āt.

Cuối cùng, bản báo cáo này nhấn mạnh vai trò quan trọng của ngăn xếp trong cấu trúc dữ liệu và thuật toán, cung cấp một cơ sở vững chắc cho việc hiểu và sử dụng ngăn xếp trong lập trình.

33

# B.Insertion Sort

## Lời mở đầu

Trong [khoa học máy tính](https://vi.wikipedia.org/wiki/Khoa_h%E1%BB%8Dc_m%C3%A1y_t%C3%ADnh) và trong [toán học](https://vi.wikipedia.org/wiki/To%C3%A1n_h%E1%BB%8Dc), thuật toán sắp xếp là một [thuật](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_to%C3%A1n) [toán](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_to%C3%A1n) sắp xếp các phần tử của một [danh sách](https://vi.wikipedia.org/wiki/Danh_s%C3%A1ch) (hoặc một [mảng](https://vi.wikipedia.org/wiki/M%E1%BA%A3ng)) theo thứ tự (tăng hoặc giảm). Sắp xếp theo thứ tự ở đây là sắp xếp theo thứ tự dạng số hoặc thứ tự dạng chữ cái như trong từ điển.Bài toán sắp xếp đã được nhiều nhà khoa học quan tâm. Với một cấu trúc được sắp xếp chúng ta s攃̀ thuận lợi khi thực hiện các tác vụ trên cấu trúc như tìm kiếm, trích lọc, duyệt cấu trúc..

Trong bản báo cáo này, chúng em xin gửi đến thầy cô và các bạn một trong những thuật toán sắp xếp đơn giản nhưng vô cùng quan trọng trong cấu trúc dữ liệu và thuật toán, đó là sắp xếp chèn(Insertion Sort)

34

## Chương 1: Insertion Sort

#### Định nghĩa

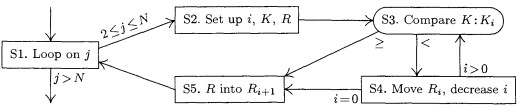
Thuật toán sắp xếp chèn **(**Insertion sort**)** được [John Mauchly](https://en.wikipedia.org/wiki/John_Mauchly) đưa ra rất sớm vào năm 1946 trong cuộc hội thảo đầu tiên về thuật toán sắp xếp trên máy tính .Sắp xếp chèn là một thuật toán sắp xếp đơn giản hoạt động tương tự như cách bạn sắp xếp các quân bài trên tay. Muốn sắp xếp một bộ bài theo trật từ người chơi bài rút lần lượt từ quân thứ 2, so với các quân đứng trước nó để chèn vào vị trí thích hợp.

Sắp xếp chèn(insertion sort) là một thuật toán sắp xếp rất hiệu quả với các danh sách nhỏ. Mảng hầu như được chia thành một phần được sắp xếp và một phần chưa được sắp xếp. Các giá trị từ phần chưa sắp xếp được chọn và đặt vào đúng vị trí trong phần đã sắp xếp.

#### Thuật toán sắp xếp chèn

Để sắp xếp một mảng có kích thước N theo thứ tự tăng dần, lặp qua mảng và so sánh phần tử hiện tại (khóa) với phần tử trước đó, nếu phần tử khóa nhỏ hơn phần tử trước đó thì so sánh nó với các phần tử trước đó. Di chuyển các phần tử lớn hơn lên một vị trí để tạo khoảng trống cho phần tử được hoán đổi.

X攃Āt một mảng gồm các phần tử R1,..,Rj. Giả sử rằng 1 < j ≤ N và rằng các phần tử R1, ..., Rj-1 đã được sắp xếp lại sao cho K1 ≤ K2 ≤ · · · ≤ Kj-1. (K là khoá tương ứng vs các phần tử R). Chúng ta so sánh khóa mới Kj với Kj-1, Kj-2, ..., lần lượt, cho đến khi phát hiện ra rằng Rj nên được chèn giữa các bản ghi Ri và Ri+1; sau đó, chúng ta di chuyển các phần tử Ri+1, ..., Rj-1 lên một vị trí và đặt phần tử mới vào vị trí i+1. Thực hiện so sánh và di chuyển, xen k攃̀ chúng như trong thuật toán dưới đây; vì Rj "ổn định ở vị trí đúng của nó", phương pháp sắp xếp này thường được gọi là kỹ thuật lọc hoặc lặn.



*Hình* *8:Thuật* *toán* *sắp* *xếp* *chèn*

35

S1. [Vòng lặp trên j] Thực hiện các bước S2 đến S5 cho j = 2, 3, ..., N; sau đó kết thúc thuật toán.

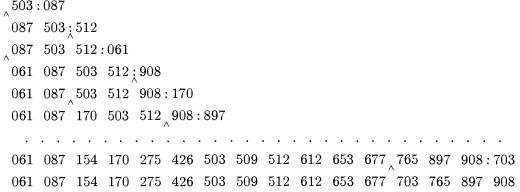
S2. [Thiết lập i, K, R] Đặt i = j - 1, K = Kj, R = Rj. (Trong các bước tiếp theo, chúng ta s攃̀ cố gắng chèn R vào vị trí đúng, bằng cách so sánh K với Ki với i giảm dần.)

S3. [So sánh K với Ki] Nếu K ≥ Ki, đi đến bước S5. (Chúng ta đã tìm thấy vị trí mong muốn cho bản ghi R.)

S4. [Cập nhật Ri, giảm i] Đặt Ri+1 = Ri, sau đó giảm i đi 1. Nếu i > 0, quay lại bước S3. (Nếu i = 0, K là khóa nhỏ nhất tìm thấy cho đến nay, vì vậy bản ghi R thuộc vị trí 1.)

S5. [Cập nhật Ri+1 bằng giá trị của R] Đặt Ri+1 = R.

Để hiểu rõ hơn về thuật toán thì chúng ta x攃Āt ví dụ về hình dưới đây. Hình 9 thể hiện cách mười sáu số ngẫu nhiên được sắp xếp bởi thuật toán sắp xếp chèn.



*Hình* *9:Ví* *dụ* *về* *thuật* *toán* *sắp* *xếp* *chèn*

#### Cách triển khai thuật toán sắp xếp chèn

Dưới đây là cách triển khai thuật toán sắp xếp chèn:

36

public class InsertionSort {

static void straightInsertionSort(int[] arr) { int n = arr.length;

for (int j = 1; j < n; j++) { int key = arr[j];

int i = j - 1;

// Compare K with Ki for decreasing values of i while (i >= 0 && key < arr[i]) {

arr[i + 1] = arr[i]; i = i - 1;

}

// Insert R into the correct position arr[i + 1] = key;

}

}

}

*Hình* *10:Thuật* *toán* *sắp* *xếp* *chèn* *trong* *java*

Độ phức tạp về thời gian của sắp xếp chèn

* + - Độ phức tạp về thời gian trường hợp xấu nhất của sắp xếp chèn là O(N2)
    - Độ phức tạp về thời gian trung bình của sắp xếp chèn là O(N2)
    - Độ phức tạp về thời gian của trường hợp tốt nhất là O(N).

#### Biến thế của sắp xếp chèn (Binary Insertion Sort, Two-way Insertion Sort)

Sắp xếp chèn nhị phân là một thuật toán sắp xếp tương tự như sắp xếp chèn, nhưng thay vì sử dụng tìm kiếm tuyến tính để tìm vị trí nơi một phần tử cần được chèn vào, chúng ta sử dụng tìm kiếm nhị phân. Do đó, chúng ta giảm giá trị so sánh của việc chèn một phần tử từ O (N) xuống O (log N).

Về cơ bản, trong quá trình sắp xếp chèn khi chúng ta đang xử lý bản ghi thứ j, chúng ta so sánh khóa của nó với khoảng j/2 khóa đã được sắp xếp trước đó, trung bình; do đó, tổng số so sánh thực hiện là khoảng (1 + 2 + · · · + N)/2 ~ N2/4, và điều này trở nên rất lớn khi N là một lượng lớn đủ. Chúng ta được nghiên cứu

37

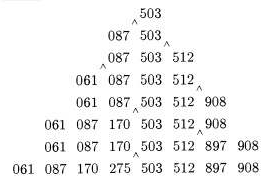
các kỹ thuật "tìm kiếm nhị phân" từ đó chỉ ra nơi để chèn mục thứ j sau khoảng lg j so sánh được thực hiện. Ví dụ, khi chèn phần tử thứ 64, chúng ta có thể bắt đầu bằng cách so sánh K64 với K32; nếu nó nhỏ hơn, chúng ta so sánh nó với K16, nhưng nếu nó lớn hơn, chúng ta so sánh nó với K48, v.v., để biết được vị trí chính xác để chèn R64 sau chỉ sau 1 vài so sánh. Tổng số so sánh để chèn tất cả các mục N đến khoảng NlgN, một cải thiện đáng kể so với ¼\*N2; cho thấy rằng chương trình tương ứng không cần phải phức tạp nhiều so với chương trình chèn trực tiếp.

Phương pháp này được gọi là chèn nhị phân; nó được đề cập bởi John Mauchly ngay từ năm 1946, trong cuộc thảo luận công bố đầu tiên về việc sắp xếp máy tính.

Vấn đề không may với chèn nhị phân là nó chỉ giải quyết một nửa của vấn đề; sau khi chúng ta đã xác định nơi phần tử Rj s攃̀ được chèn, chúng ta vẫn cần di chuyển khoảng 1/2j phần tử đã được sắp xếp trước đó để làm cho chỗ cho Rj, do đó, tổng thời gian chạy vẫn hoàn toàn tỷ lệ với N2. Một số máy tính sớm như IBM 705 đã có một "tumble" instruction tích hợp thực hiện các thao tác di chuyển này ở tốc độ cao, và các máy hiện đại có thể thực hiện các thao tác di chuyển nhanh hơn với các phụ kiện phần cứng đặc biệt; nhưng khi N tăng lên, sự phụ thuộc vào N2 cuối cùng s攃̀ trở nên quan trọng. Ví dụ, một phân tích của H. Nagler [CACM 3 (1960), 618-620] chỉ ra rằng chèn nhị phân không nên được khuyến nghị cho việc sắp xếp hơn khoảng N = 128 bản ghi trên IBM 705, khi mỗi bản ghi có chiều dài 80 ký tự, và các phân tích tương tự áp dụng cho các máy khác.

Tất nhiên, một lập trình viên kh攃Āo l攃Āo có thể nghĩ ra nhiều cách để giảm lượng di chuyển cần thiết; một cách như vậy, được đề xuất vào đầu những năm 1950, được minh họa trong Hình 11. Ở đây, mục đầu tiên được đặt ở giữa của một khu vực đầu ra, và không gian được tạo ra cho các mục sau bằng cách di chuyển về phải hoặc về trái, tùy thuộc vào điều nào thuận tiện nhất. Điều này tiết kiệm khoảng một nửa thời gian chạy so với chèn nhị phân thông thường, với chi phí của một chương trình phức tạp hơn một chút. Có thể sử dụng phương pháp này mà không làm tăng thêm không gian so với yêu cầu của N bản ghi ; nhưng chúng ta s攃̀ không tìm hiểu thêm về phương pháp chèn "hai chiều" này, vì đã có những kỹ thuật thú vị hơn đã được phát triển.

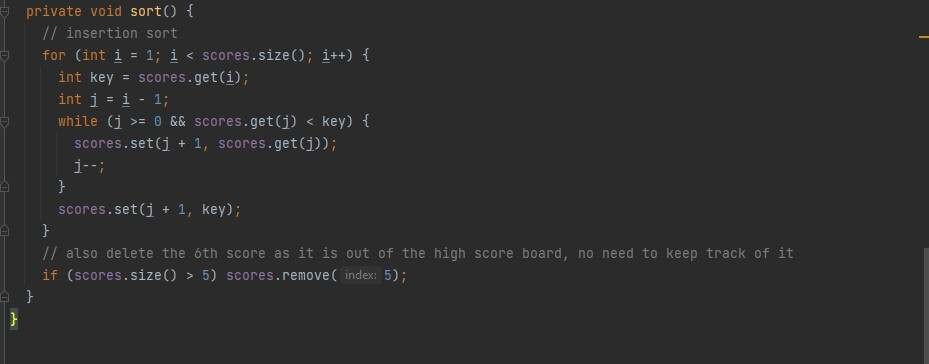
38



*Hình* *11:Sắp* *xếp* *chèn* *hai* *chiều*

#### Ứng dụng của Insertion Sort

Insertion sort có rất nhiều ứng dụng nhưng chúng ta s攃̀ ko đi vào cụ thể ở đây. Thay vào đó chúng ta s攃̀ xem 1 ví dụ về việc sử dụng insertion sort vào trong bài code nhóm của chúng em.

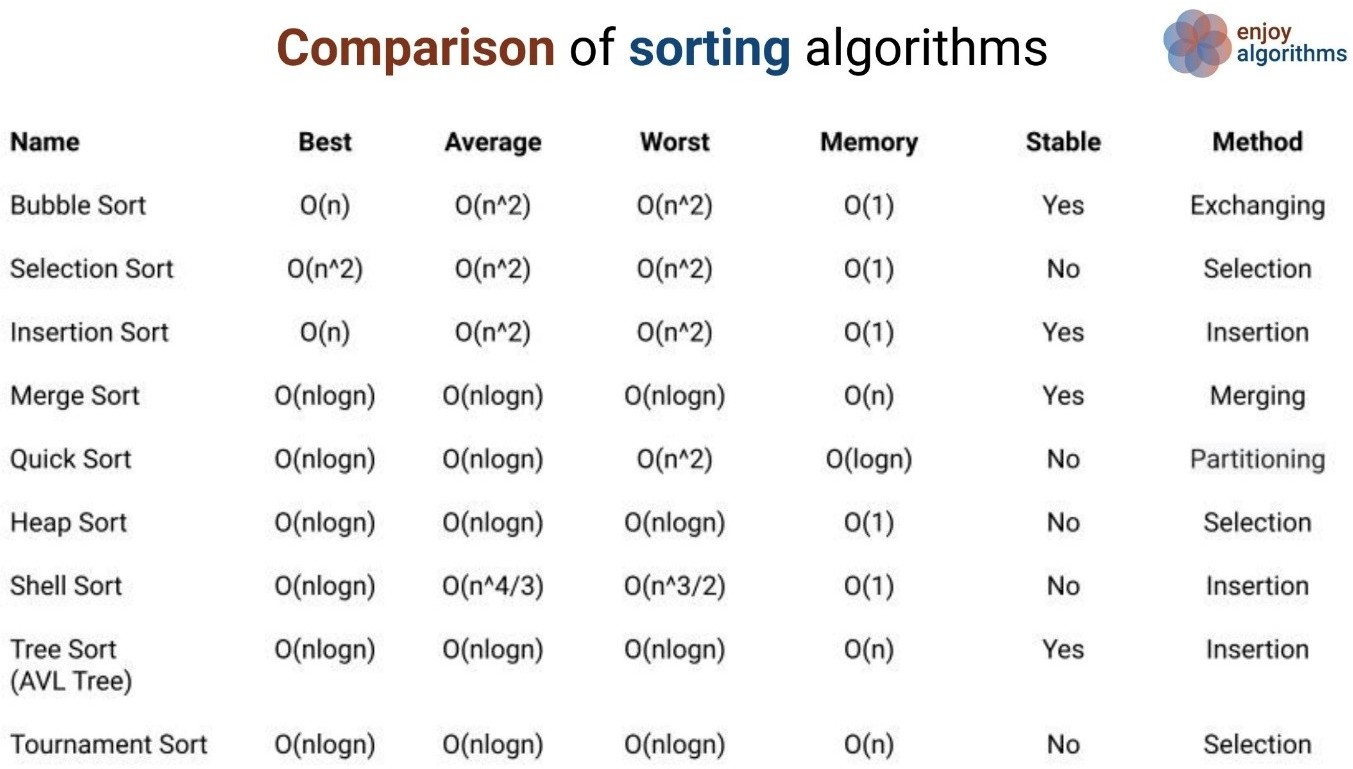


Đoạn code này sử dụng phương thức insertion sort để sắp xếp số điểm mà người chơi nhận được theo thứ tự giảm dẩn. Dòng code cuối được thêm vào ý muốn chỉ rằng nếu có xuất hiện điểm số ở ví trí thứ 6 thì xoá nó ra khỏi list score và từ đó chúng ta s攃̀ được 1 bảng highscore gồm có vị trí từ 1 đến 5.

39

#### So sánh Insertion Sort với các thuật toán sắp xếp khác

Ngoài Insertion Sort ra, chúng ta còn có rất nhiều các thuật toán sắp xếp khác như merge sort, quick sort,… Hình dưới đây s攃̀ cho ta có cái nhìn tổng quan hơn về các thuật toán sắp xếp



*Hình* *12:Bảng* *so* *sánh* *các* *thuật* *toán* *sắp* *xếp*

40

## Tổng kết

Thuật toán sắp xếp chèn là một lựa chọn tốt cho các tập dữ liệu nhỏ, nhưng nó không phải lựa chọn tốt nhất cho các tập dữ liệu lớn với độ phức tạp thời gian cao. Các cải tiến như chèn nhị phân và chèn "hai chiều" có thể giúp giảm thiểu một số hạn chế của thuật toán. Mặc dù có hiệu suất không tốt trên tập dữ liệu lớn, thuật toán này vẫn vô cùng giá trị trong các tình huống cụ thể, làm cho nó một công cụ hữu ích trong lập trình và công việc thực tế.

41

### Tài liệu tham khảo

###### [Stack (abstract data type) - wikipedia](https://en.wikipedia.org/wiki/Stack_(abstract_data_type))

1. [Data Structures and Algorithms in Java™ Sixth Edition- Michael T.](https://bedford-computing.co.uk/learning/wp-content/uploads/2016/08/Data-Structures-and-Algorithms-in-Java-6th-Edition.pdf)  [Goodrich, Roberto Tamassia, Michael H. Goldwasser](https://bedford-computing.co.uk/learning/wp-content/uploads/2016/08/Data-Structures-and-Algorithms-in-Java-6th-Edition.pdf)
2. [Stack Data Structure - GeeksForGeeks](https://www.geeksforgeeks.org/stack-data-structure/)
3. [Sự khác nhau giữa Stack và Queue -Viblo](https://viblo.asia/p/su-khac-nhau-giua-stack-va-queue-L4x5xM2wKBM)
4. [Java.The Stack Class – tutorialspoint](https://www.tutorialspoint.com/java/java_stack_class.htm)
5. [Insertion Sort – GeeksForGeeks](https://www.geeksforgeeks.org/insertion-sort/)
6. [Art of Computer Programming - Volume 3 (Sorting & Searching)-](https://seriouscomputerist.atariverse.com/media/pdf/book/Art%20of%20Computer%20Programming%20-%20Volume%203%20(Sorting%20%26%20Searching).pdf) [Donald Knuth](https://seriouscomputerist.atariverse.com/media/pdf/book/Art%20of%20Computer%20Programming%20-%20Volume%203%20(Sorting%20%26%20Searching).pdf)
7. Data Structures and Algorithms Made Easy By Narasimha Karumanchi

42