MỤC LỤC

[**I.** **BÀI TẬP TỔNG HỢP LINKED LIST** 2](#_Toc134651997)

[**1. BT tại lớp 1 – Singly linked list** 2](#_Toc134651998)

[**2. BT tại lớp 2– Doubly linked list** 4](#_Toc134652000)

[**3. BT tại lớp 3: Ứng dụng DSLK** 7](#_Toc134652002)

[**II.** **BÀI TẬP TỔNG HỢP STACK** 7](#_Toc134652003)

[**1. Cài đặt Stack theo 2 cách:** 7](#_Toc134652004)

[**Dựa vào mảng.** 8](#_Toc134652005)

[**Dựa vào danh sách liên kết.** 8](#_Toc134652006)

[**2. BT ứng dụng Stack** 8](#_Toc134652007)

[**III.** **BÀI TẬP TỔNG HỢP QUEUE** 10](#_Toc134652008)

[**1.** **Cài đặt Queue theo 2 cách:** 10](#_Toc134652009)

[**Dựa vào mảng.** 10](#_Toc134652010)

[**Dựa vào danh sách liên kết.** 10](#_Toc134652011)

[**2. BT ứng dụng Queue: Chọn 1 bài trong code.ptit, phân tích tính phù** 11](#_Toc134652012)

[**IV.** **BÀI TẬP TỔNG HỢP BINARY SEARCH TREE** 12](#_Toc134652013)

[**1. Cài đặt các thao tác trên cây nhị phân tìm kiếm** 12](#_Toc134652014)

[**2. BT ứng dụng Binary Search Tree: Chọn 1 bài trong code.ptit, phân tích tính** 14](#_Toc134652015)

[**V.** **BÀI TẬP TỔNG HỢP AVL** 15](#_Toc134652016)

[**1. Cài đặt các thao tác trên cây nhị phân tìm kiếm tự cân bằng** 15](#_Toc134652017)

[**2. BT ứng dụng AVL: Chọn 1 bài trong code.ptit, phân tích tính phù hợp và** 16](#_Toc134652018)

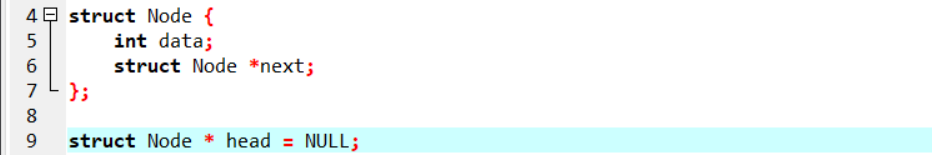
**BÁO CÁO TỔNG HỢP CTDL**

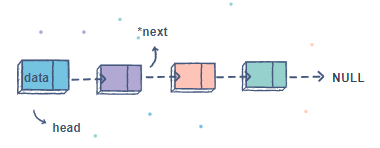
1. **BÀI TẬP TỔNG HỢP LINKED LIST**

## **1. BT tại lớp 1 – Singly linked list**

* **Tạo ds liên kết đơn**

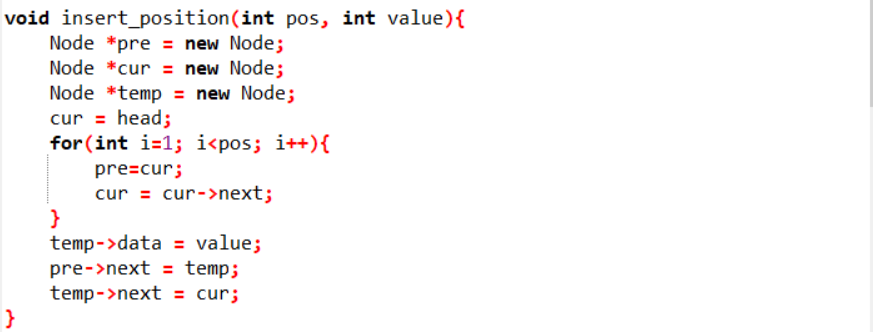
Định nghĩa ra 1 danh sách liên kết đơn bao gồm 2 thành là data chứa giá trị của node hiện tại và con trỏ next để trỏ tới phần tử node tiếp theo.

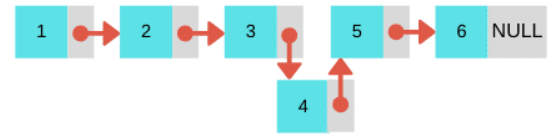
****

****

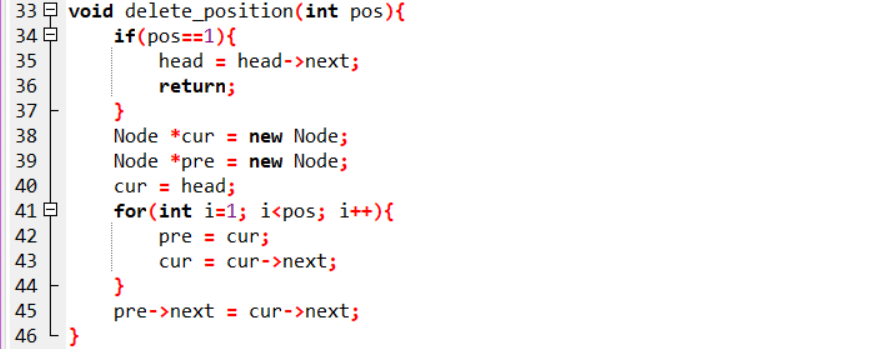
* **Thêm 1 phần tử vào ds lk đơn tại vị trí x**

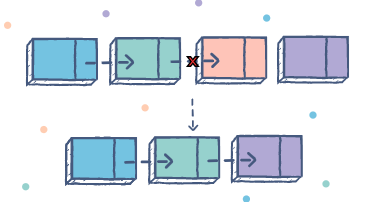
Cách thêm 1 phần tử vào dslk đơn tại vị trí x là dùng 2 con trỏ cur lấy ra phân từ hiện tại và 1 con trỏ pre lấy ra phần từ ngay phía trước. Khi tìm ra được vị trí cần thêm node mới chúng ta sẽ tạo thêm 1 node mới temp. Sau đó cho con trỏ next của pre trỏ đến node mới này và con trỏ next của phần tử mới này trỏ đến node cur.

****

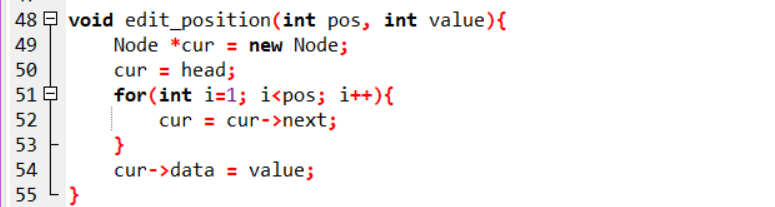
****

* **Xóa 1 phần tử từ ds lk đơn tại vị trí x**

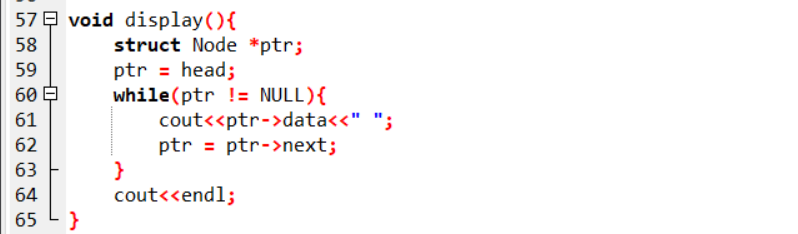
****

****

* **Sửa giá trị cho phần tử tại vị trí x trong dslk đơn**

****

* **Duyệt ds liên kết đơn**

****

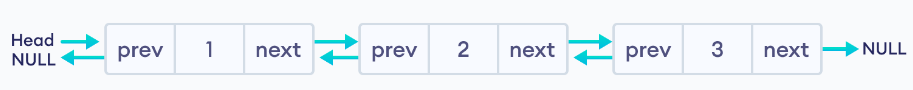
* **Link code full:** [*https://github.com/namlk173/C-/blob/datastructure\_and\_algorithm/LINKED\_LIST.cpp*](https://github.com/namlk173/C-/blob/datastructure_and_algorithm/LINKED_LIST.cpp)

## **2. BT tại lớp 2– Doubly linked list**

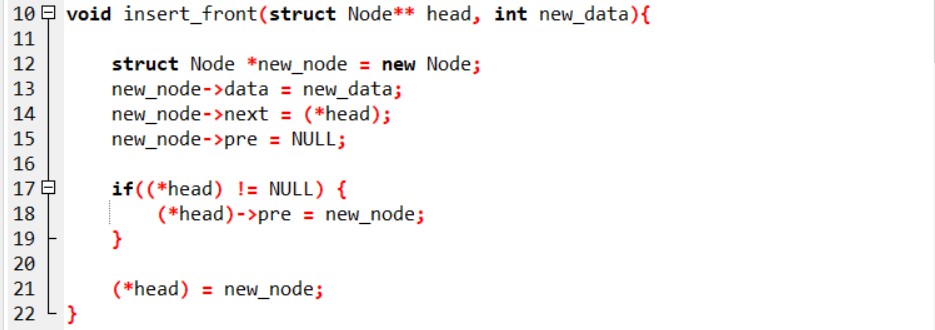
* **Tạo ds liên kết kép**

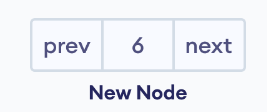
****

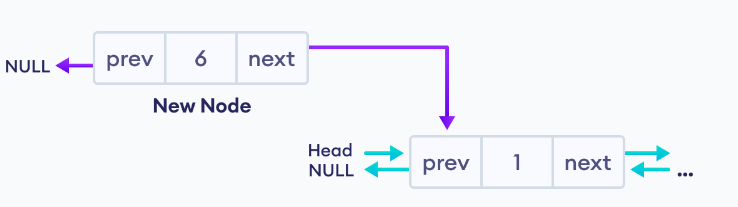
****

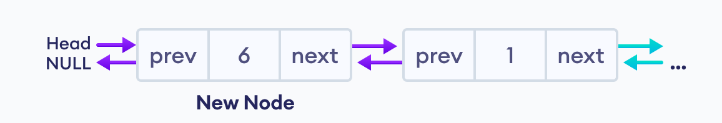
****

* **Thêm 1 phần tử vào đầu ds lk kép**

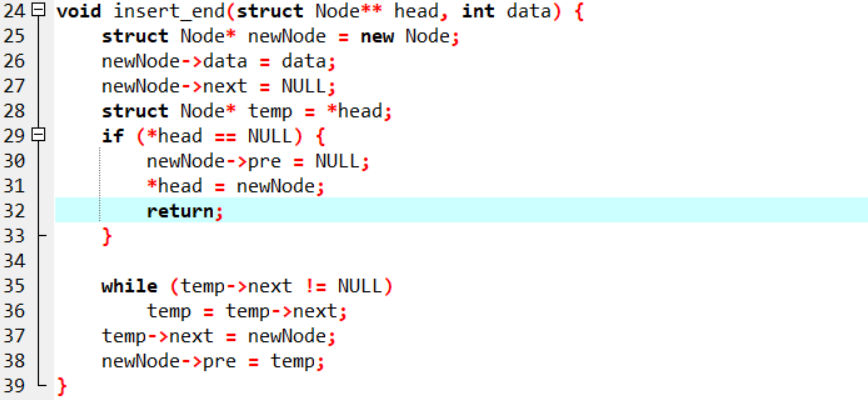
****

****

****

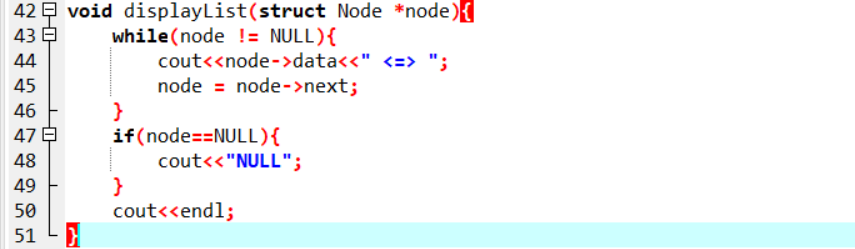
****

* **Thêm 1 phần tử vào cuối ds lk kép**

****

****

* **Duyệt ds liên kết kép**

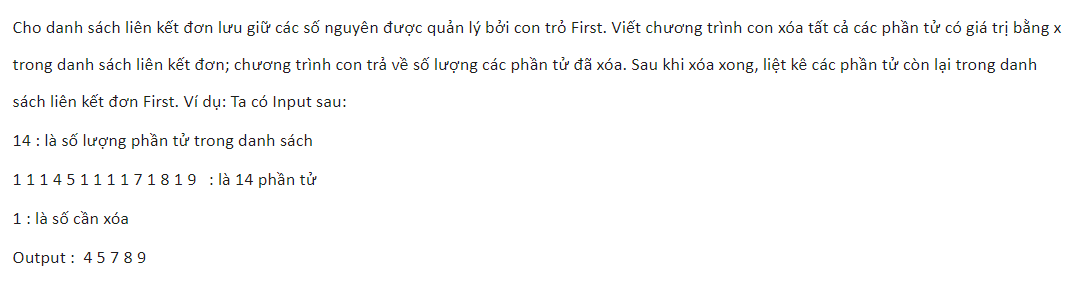
****

* **Link code full:** [***https://github.com/namlk173/C-/blob/datastructure\_and\_algorithm/Doubly\_List.cpp***](https://github.com/namlk173/C-/blob/datastructure_and_algorithm/Doubly_List.cpp)

## **3. BT tại lớp 3: Ứng dụng DSLK**

* **Chọn 1 bài trong code.ptit: Phân tích tính phù hợp và ứng dụng DSLK vào giải quyết bài toán đó.**

**Đề bài: CTDL\_005 – Xóa dữ liệu trong danh sách liên kết đơn.**

****

**Phân tích bài toán:** Hoàn toàn có thể sử dụng mảng 1 chiều để cho bài toàn được thực hiện dễ dàng hơn. Tuy nhiên chúng ta sẽ sử dụng Danh sách liên kết đơn để thực hiện bài toán.

Cần sử dụng các thao tác trên danh sách liên kết đơn để thực hiện bài toán đó là.

* Thêm 1 phần tử vào cuối danh sách để lưu các giá trị,
* Duyệt danh sách liên kết để đưa ra các phần tử còn lại của danh sách,
* Xóa 1 phần tử bất kỳ trong danh sách có giá trị bằng với giá trị đưa vào.

**Chuyển hóa sang code:**

**Link code:** [*https://github.com/namlk173/C-/blob/datastructure\_and\_algorithm/DSLK\_Don.cpp*](https://github.com/namlk173/C-/blob/datastructure_and_algorithm/DSLK_Don.cpp)

1. **BÀI TẬP TỔNG HỢP STACK**

## **1. Cài đặt Stack theo 2 cách:**

Thao tác trên Stack: Theo nguyên tắc LIFO phần tử vào sau sẽ được lấy ra trước. ó hai thao tác cơ bản để tạo nên cơ chế vào sau ra trước (LIFO) của stack đó là đưa phần tử vào ngăn xếp (push) và lấy phần tử ra khỏi ngăn xếp. Hai thao tác push và pop đều thực hiện chung tại một vị trí trên ngăn xếp sẽ tạo nên cơ chế LIFO.

* Đưa dữ liệu vào Stack
* Lấy dữ liệu ra khỏi Stack



### **Dựa vào mảng.**

* **Link code full:** [***https://github.com/namlk173/C-/blob/datastructure\_and\_algorithm/Stack.cpp***](https://github.com/namlk173/C-/blob/datastructure_and_algorithm/Stack.cpp)

### **Dựa vào danh sách liên kết.**

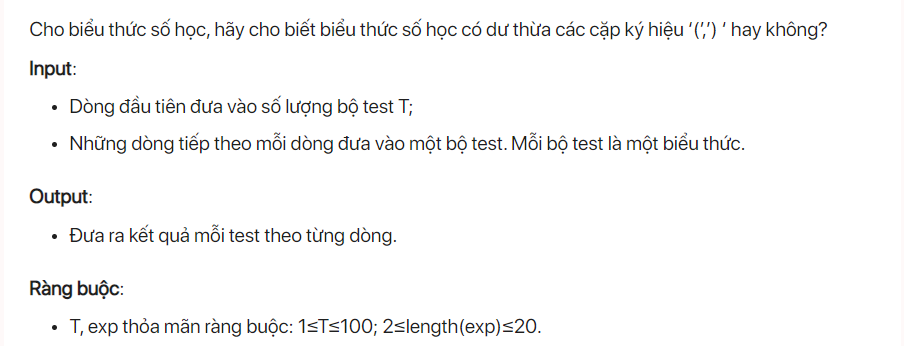
* **Link code full:** [***https://github.com/namlk173/C-/blob/datastructure\_and\_algorithm/Stack\_using\_linked\_list.cpp***](https://github.com/namlk173/C-/blob/datastructure_and_algorithm/Stack_using_linked_list.cpp)

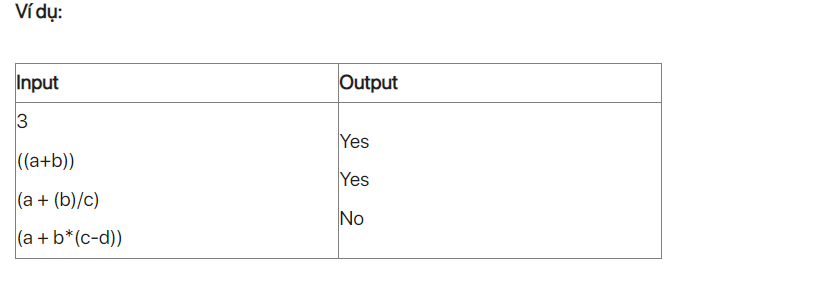
## **2. BT ứng dụng Stack**

**Chọn 1 bài trong code.ptit: Phân tích tính phù hợp và ứng dụng Stack**

**vào giải quyết bài toán đó.**

**Đề bài: DSA07003 – Kiểm tra biểu thức số học.**

****

****

**Phân tích:** Ý tưởng là sử dụng ngăn xếp, Đối với bất kỳ biểu thức con nào của biểu thức, nếu chúng ta có thể chọn bất kỳ biểu thức con nào của biểu thức được bao quanh bởi (), thì chúng ta lại để lại ( ) như một phần của chuỗi, thừa dấu ngoặc.

Thực hiện theo các bước được đề cập dưới đây để thực hiện phương pháp:

Chúng ta lặp qua biểu thức đã cho và cho từng ký tự trong biểu thức nếu ký tự là một dấu ngoặc đơn mở '(' hoặc bất kỳ toán tử hoặc toán hạng nào, chúng ta sẽ đẩy ký tự đó vào ngăn xếp.

Nếu ký tự nằm trong dấu ngoặc đơn đóng ')', thì bật các ký tự từ ngăn xếp cho đến khi tìm thấy dấu ngoặc đơn mở khớp '('.

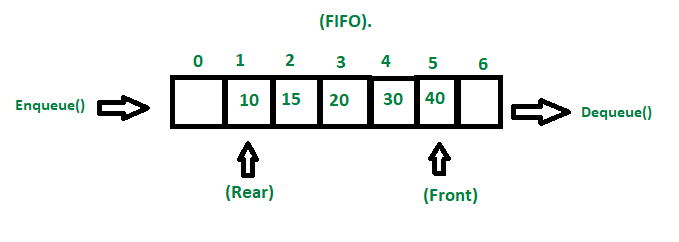
Bây giờ để dự phòng, hai điều kiện sẽ phát sinh trong khi xuất hiện.

Nếu cửa sổ bật lên ngay lập tức chạm vào dấu ngoặc đơn mở '(', thì chúng ta đã tìm thấy dấu ngoặc đơn trùng lặp. Ví dụ: (((a+b))+c) có dấu ngoặc đơn trùng lặp xung quanh a+b. Khi chúng ta đến dấu “)” thứ hai sau a+b, chúng ta có “((” trong ngăn xếp. Vì đỉnh của ngăn xếp là dấu ngoặc mở, chúng ta kết luận rằng có các dấu ngoặc thừa.

Nếu pop ngay lập tức không chạm vào bất kỳ toán hạng nào (‘\*’, ‘+’, ‘/’, ‘-‘) thì nó cho biết sự hiện diện của các dấu ngoặc không mong muốn được bao quanh bởi biểu thức. Chẳng hạn, (a)+b chứa () không mong muốn xung quanh a, do đó nó là dư thừa.

**Chuyển hóa sang code:** [*https://github.com/namlk173/C-/blob/datastructure\_and\_algorithm/DSA07003-Kiem\_Tra\_Bieu\_Thuc\_So\_Hoc.cpp*](https://github.com/namlk173/C-/blob/datastructure_and_algorithm/DSA07003-Kiem_Tra_Bieu_Thuc_So_Hoc.cpp)

1. **BÀI TẬP TỔNG HỢP QUEUE**
2. **Cài đặt Queue theo 2 cách:** Queue được xây dựng trên nguyên tắc FIFO phần tử nào được đưa vào trước sẽ được lấy ra trước. Hàng đợi được xây dựng dựa vào hai thao tác cơ bản: đưa phần tử vào hàng đợi (push) và lấy phần tử ra khỏi hàng đợi (pop). Hai thao tác push và pop phối hợp với nhau để tạo nên cơ chế FIFO của hàng đợi.



### **Dựa vào mảng.**

* **Link code full:** [***https://github.com/namlk173/C-/blob/datastructure\_and\_algorithm/Queue\_Array.cpp***](https://github.com/namlk173/C-/blob/datastructure_and_algorithm/Queue_Array.cpp)

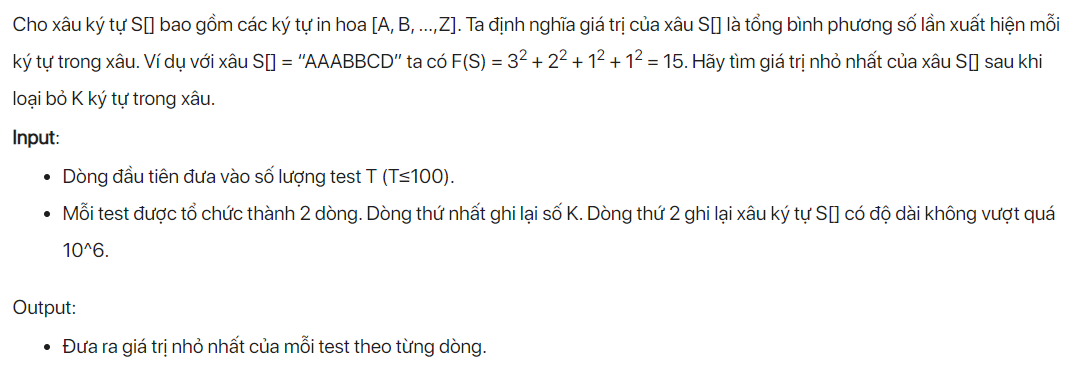
### **Dựa vào danh sách liên kết.**

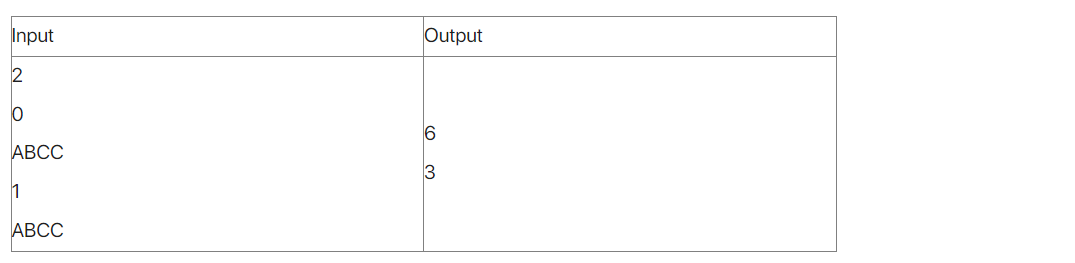
* **Link code full:** [***https://github.com/namlk173/C-/blob/datastructure\_and\_algorithm/Queue\_Linked\_List.cpp***](https://github.com/namlk173/C-/blob/datastructure_and_algorithm/Queue_Linked_List.cpp)

## **2. BT ứng dụng Queue: Chọn 1 bài trong code.ptit, phân tích tính phù**

**hợp và ứng dụng Queue vào giải quyết bài toán đó.**

**Đề bài: DSA08004 – Giá trị nhỏ nhất của xâu.**

****

****

**Phân tích bài toán:**

Đầu tiên chúng ta sẽ tính số lần xuất hiện của tất cả các kí tự trong xâu.

Chúng ta có thể thấy rằng biểu thức F(s) có cùng số mũ vì vậy phần tử càng có giá trị càng cao thì giá trị của biểu thức càng cao vì vậy chúng ta phải giảm giá trị của các phần tử là số lần xuất hiện của các phần tử có số lần xuất hiện nhiều hơn trước.

Sử dụng hàng đợi ưu tiên: Priority\_queue để lưu các giá trị là số lần xuất hiện của các kí tự trong xâu.

Mỗi lần lặp( mỗi lần bỏ 1 kí tự xuất hiện trrong xâu) Chúng ta sẽ lấy ra phần tử ở đầu queue( phần tử có giá trị lớn nhất) giảm giá trị xuống 1 đơn vị.

Sau đó lại push giá trị mới vào queue.

Sau khi lặp chúng ta sẽ tính giá trị của biểu thức dựa trên số lần xuất hiện còn lặp trong queue đây cũng chính là kết quả cuối cùng của bài toán.

**Chuyển hóa sang code:** [*https://github.com/namlk173/C-/blob/datastructure\_and\_algorithm/DSA08004-Gia\_Tri\_Nho\_Nhat\_Cua\_Xau.cpp*](https://github.com/namlk173/C-/blob/datastructure_and_algorithm/DSA08004-Gia_Tri_Nho_Nhat_Cua_Xau.cpp%20)

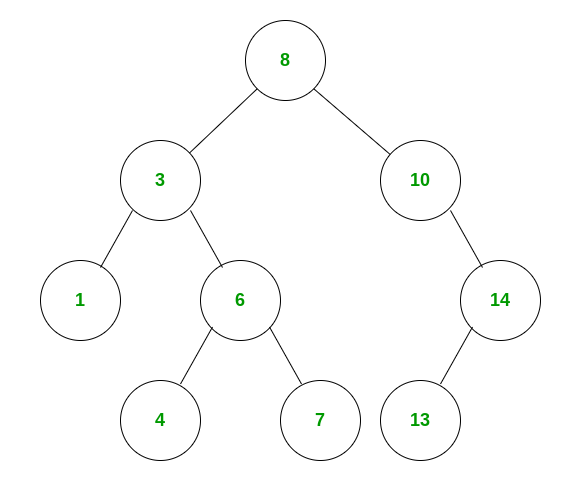
1. **BÀI TẬP TỔNG HỢP BINARY SEARCH TREE**

## **1. Cài đặt các thao tác trên cây nhị phân tìm kiếm**

* Tạo node gốc cho cây.
* Thêm vào node vào cây tìm kiếm.
* Loại bỏ node trên cây tìm kiếm.
* Tìm kiếm node trên cây.
* Xoay trái cây tìm kiếm
* Xoay phải cây tìm kiếm
* Duyệt cây theo thứ tự trước.
* Duyệt cây theo thứ tự giữa.
* Duyệt cây theo thứ tự sau.

**Cây nhị phân tìm kiếm là một cây nhị phân thỏa mãn hai điều kiện:**

* Hoặc là rỗng hoặc có một node gốc. Mỗi node gốc có tối đa hai cây con.
* Nội dung node gốc lớn hơn nội dung node con bên trái và nhỏ hơn nội dung node con bên phải. Hai cây con bên trái và bên phải cũng hình thành nên hai cây nhị phân tìm kiếm.



Biểu diễn cây nhị phân tìm kiếm cũng giống như biểu diễn của các cây nhị phân thông thường. Trong trường hợp biểu diễn liên tục ta sử dụng mảng. Trong trường hợp biểu diễn rời rạc ta sử dụng danh sách liên kết.

***struct node {***

***int data; //thông tin của node***

***struct node \*left; //con trỏ trỏ đến cây con trái***

***struct node \*right; // con trỏ trỏ đến cây con phải***

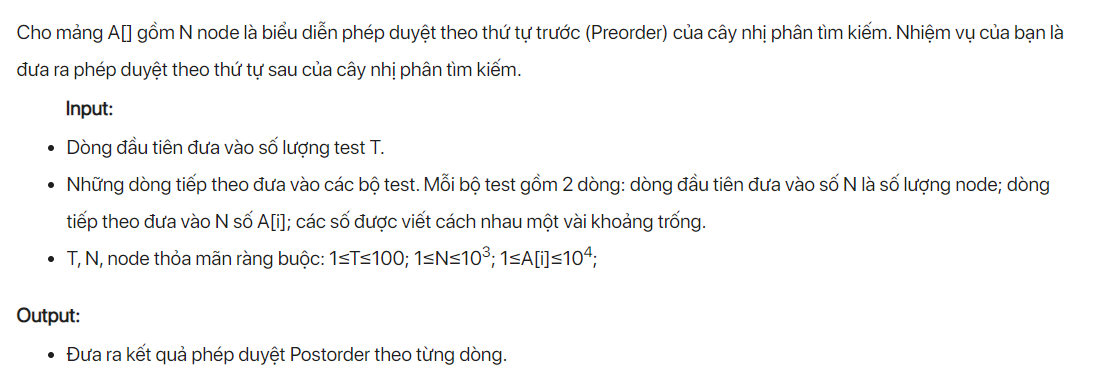
***} \*BST\_Tree;***

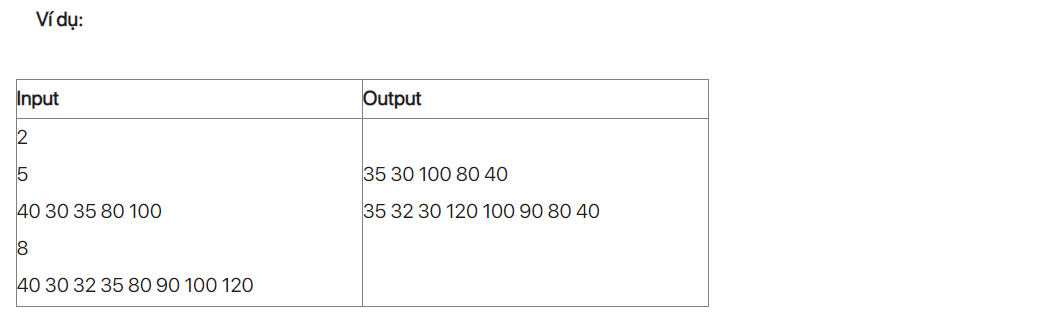
**Link code full:** [*https://github.com/namlk173/C-/blob/datastructure\_and\_algorithm/BINARY\_SEARCH\_TREE.cpp*](https://github.com/namlk173/C-/blob/datastructure_and_algorithm/BINARY_SEARCH_TREE.cpp)

## **2. BT ứng dụng Binary Search Tree: Chọn 1 bài trong code.ptit, phân tích tính**

**phù hợp và ứng dụng Binary Search Tree vào giải quyết bài toán đó.**

**Đề bài: DSA11017 – Duyệt cây nhị phân tìm kiếm 1.**

****

****

**Phân tích bài toán:**

Để giải quyết vấn đề, hãy làm theo ý tưởng dưới đây:

Sử dụng khái niệm đệ quy và lặp qua mảng các phần tử đã cho, chúng ta có thể tạo BST

Thực hiện theo các bước dưới đây để giải quyết vấn đề:

Tạo một Nút mới cho mọi giá trị trong mảng

Tạo BST bằng cách sử dụng các Nút mới này và chèn chúng theo quy tắc của BST

In thứ tự của BST

**Chuyển hóa sang code:** [*https://github.com/namlk173/C-/blob/datastructure\_and\_algorithm/Duyet\_Cay\_nhi\_Phan\_tim\_Kiem.cpp*](https://github.com/namlk173/C-/blob/datastructure_and_algorithm/Duyet_Cay_nhi_Phan_tim_Kiem.cpp)

1. **BÀI TẬP TỔNG HỢP AVL**

## **1. Cài đặt các thao tác trên cây nhị phân tìm kiếm tự cân bằng**

* Xoay trái cây AVL
* Xoay phải cây AVL
* Thêm node vào cây AVL
* Loại bỏ node trên cây AVL

Cây tìm nhị phân tìm kiếm cân bằng có tính chất độ cao của cây con bên trái và độ cao cây con bên phải luôn lệch nhau không quá 1.

Biểu diễn cây nhị phân tìm kiếm cân bằng cũng giống như biểu diễn cây nhị phân tìm kiếm thông thường. Tuy nhiên, để thuận tiện trong việc tính toán chỉ số cân bằng của các node trên cây ta đưa thêm thông tin độ cao mỗi node. Cây AVL được biểu diễn như sau:

***struct node {//định nghĩa cấu trúc node***

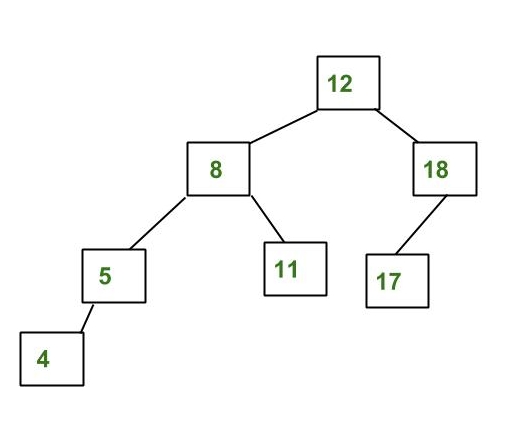
***int data;//thành phần dữ liệu của node***

***int height; //độ cao của node***

***struct node \*left;//thành phần con trỏ cây con trái***

***struct node \*right; //thành phần con trỏ cây con phải***

***} \*AVL\_Tree; //định nghĩa cây AVL***

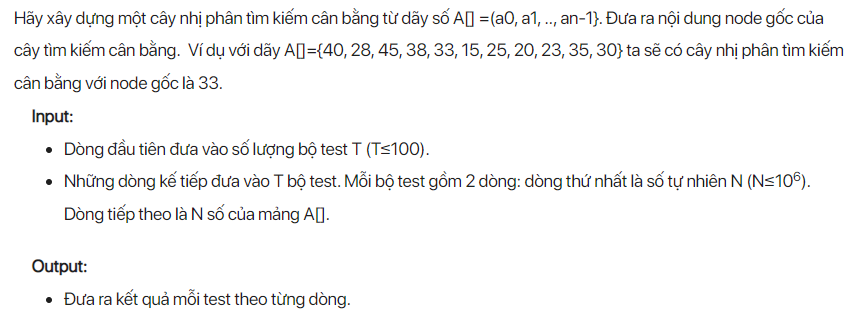


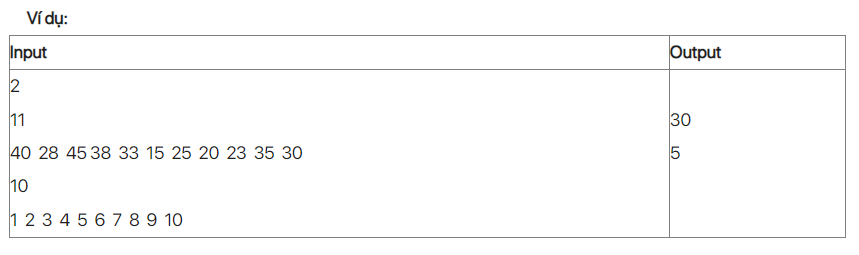
**Link code full:** [*https://github.com/namlk173/C-/blob/datastructure\_and\_algorithm/AVL-TREE.cpp*](https://github.com/namlk173/C-/blob/datastructure_and_algorithm/AVL-TREE.cpp)

## **2. BT ứng dụng AVL: Chọn 1 bài trong code.ptit, phân tích tính phù hợp và**

**ứng dụng AVL vào giải quyết bài toán đó.**

**Đề bài: DSA11024 – Cây Nhị Phân Tìm Kiếm Cân Bằng 1**

****

****

**Phân tích bài toán:**

**Chuyển hóa sang code: Chưa làm được**