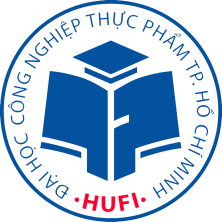
**BỘ CÔNG THƯƠNG**

**TRƯỜNG ĐẠI HỌC CÔNG NGHIỆP THỰC PHẨM TP.HCM**

**KHOA CÔNG NGHỆ THÔNG TIN**





**ĐỒ ÁN MÔN HỌC**

**CÀI ĐẶT MÔ PHỎNG CẤU TRÚC DỮ LIỆU CÂY**

Giảng viên hướng dẫn: **TS. Trần Văn Thọ**

Sinh viên thực hiện:

1. ĐỖ TRƯỜNG DŨ

**MSSV:2033172022**

1. NGUYỄN CÔNG DŨNG

**MSSV:2033172024**

# 

# LỜI CẢM ƠN

Đầu tiên, chúng em xin gửi lời cảm ơn chân thành đến thầy Vũ Đức Thịnh, là giảng viên hướng dẫn chúng em thực hiện đề tài. Thầy đã tận tình chỉ bảo, giúp đỡ nhóm rất nhiều trong suốt thời gian thực hiện và cũng là người gợi ý nhiều khái niệm, hướng đi mới giúp cho đề tài thực hiện một cách trơn tru, không bị lạc hướng.

Ngoài ra, nhóm cũng xin gửi lời cảm ơn đến các thầy cô trong khoa Công nghệ thông tin đã giảng dạy và tạo điều kiện cho nhóm hoàn thành được đề tài.

Mặc dù đã cố gắng hoàn thành đề tài một cách tốt nhất có thể. Tuy nhiên sẽ không tránh khỏi những sai sót. Chúng em rất mong nhận được sự cảm thông, góp ý đến từ các thầy cô để giúp cho chúng em ngày càng hoàn thiện hơn.

# NHẬN XÉT (Của giảng viên hướng dẫn)

# NHẬN XÉT (Của giảng viên phản biện)

**TRƯỜNG ĐẠI HỌC CÔNG NGHIỆP CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM**

**THỰC PHẨM TP.HCM Độc lập – Tự do – Hạnh phúc**

**KHOA CÔNG NGHỆ THÔNG TIN**

TP.Hồ Chí Minh, ngày … tháng… năm …

# ĐỀ CƯƠNG CHI TIẾT ĐỒ ÁN CHUYÊN NGÀNH

1. **Tên đề tài:** Cài đặt mô phỏng Cấu trúc dữ liệu Cây
2. **Thông tin giảng viên hướng dẫn:**

* Họ tên: ThS. Trần Văn Thọ
* Email: [thotv@hufi.edu.vn](mailto:thotv@hufi.edu.vn) ĐTDĐ: 0989.282.536

1. **Mục đích:**

* Trình bày tóm tắt phần lý thuyết về các loại cấu trúc dữ liệu cây (nhị phân, nhị phân tìm kiếm, nhị phân tìm kiếm cân bằng (AVL), Cây B-Tree, Cây đỏ đen), có ví dụ minh họa.
* Cài đặt mô phỏng trực quan bằng đồ họa các thao tác trên từng loại cây, cho phép người dùng thay đổi dữ liệu input.
* Ngôn ngữ lập trình: C++ hoặc C#
* Xây dựng giao diện đồ họa: trực quan, thân thiện, dễ sử dụng.

1. **Yêu cầu:**

* Siêng năng, chịu khó, có khả năng tự học và làm việc theo nhóm.
* Khả năng đọc tài liệu tiếng anh.
* Khả năng tìm hiểu các mô hình hệ thống, các ứng dụng có sẵn, khả năng sử dụng lại tài nguyên cần thiết.
* Khả năng phân tích lý thuyết và đưa ra mô hình hệ thống.
* Trình bày nội dung văn bản đúng biểu mẫu và đúng cấu trúc theo quy định.

1. **Môi trường thực hiện: sinh viên tự chọn môi tường thực hiện**

* Ngôn ngữ lập trình: C++ hoặc C#
* Phần mềm sử dụng: Microsoft Studio
* Môi trường minh hoạ: Windows Form (hoặc Web)

1. **Thời gian thực hiện**

* 8 tuần (Từ ngày … tháng … năm … đến ngày … tháng … năm …)

1. **Phân bổ tiến trình công việc hàng tuần**

|  |  |  |
| --- | --- | --- |
| **Tuần** | **Công việc thực hiện** | **Kết quả đạt được** |
| 01 | − Xác định rõ mục tiêu và yêu cầu của đề tài.  − GVHD hoặc nhóm trưởng phân chia công việc cụ thể từng thành viên.  − Tìm hiểu qui trình xây dựng phần mềm.  − Khảo sát các phần mềm hướng dẫn học tập tương tự.  − Xác định yêu cầu chức năng, xác định môi trường triển khai.  − Trình bày qui trình xử lý của hệ thống phần mềm. − Vẽ mô hình qui trình nghiệp vụ BPM, usecase (nếu có) | * Hiểu rõ mục tiêu và yêu cầu |
| 02 | − Phân tích và thiết kế dữ liệu.  − Trình bày phần lý thuyết theo yêu cầu. | * Biết được cách phân tích và thiết kế dữ liệu |
| 03 | − Tìm hiểu về cây nhị phân. | * Hiểu về cây nhị phân và các node. |
| 04 | − Tìm hiều về cây nhị phân tìm kiếm. | * Hiểu được cây nhị phân tìm kiếm. |
| 05 | − Tìm hiểu về cây nhị phân tìm kiếm cân bằng (AVL). | * Hiểu được cây nhị phân tìm kiếm cân bình (AVL). |
| 06 | − Tìm hiểu về cây nhị phân tìm kiếm cân bằng (AVL). (tt)  − Tìm hiểu về cây B-Tree. | * Hiểu được cây nhị phân tìm kiếm cân bình (AVL) và cài đặt được cây B-Tree. |
| 07 | − Tìm hiểu về cây đỏ đen. | * Hiểu được về cây đỏ đen |
| 08 | − Viết báo cáo và hoàn chỉnh đề tài.  − Thiết kế slide báo cáo trên Powerpoint | * Hiểu hoàn toàn đề tài và báo cáo slide   . |

Tài liệu tham khảo:

[1]

TP.HCM, Ngày … Tháng … Năm …

*Trưởng bộ môn* *Giảng viên hướng dẫn*

Trần Văn Thọ

# MỤC LỤC

MỤC LỤC

LỜI CẢM ƠN2

NHẬN XÉT3

ĐỀ CƯƠNG CHI TIẾT CHUYÊN NGÀNH 5

MỤC LỤC 9

TÓM TẮT ĐỒ ÁN 12

DANH MỤC MỘT SỐ KHÁI NIỆM 13

LỜI MỞ ĐẦU 14

CHƯƠNG 1 : TRÌNH BÀY VỀ CẤU TRÚC DỮ LIỆU CÂY15

1. Cây nhị phân 15
   1. Khái niệm và ví dụ15
   2. Biểu diễn cây nhị phân16
   3. Các thao tác trên cây nhị phân16
      1. Khởi tạo cây nhị phân rỗng16
      2. Tạo nút chứa giá trị x17
      3. Kiểm tra cây nhị phân rỗng 17
      4. Duyệt theo thứ tự giống trước (NLR) 18
      5. Duyệt theo thứ tự giống giữa (LNR) 18
      6. Duyệt theo thứ tự giống sau (LRN) 19
2. Cây nhị phân tìm kiếm 19
   1. Khái niệm và ví dụ19
   2. Biểu diễn cây nhị phân tìm kiếm 19
   3. Các thao tác trên cây nhị phân20
      1. Tìm kiếm phần tử x trong cây20
         1. Viết hàm dạng đệ quy20
         2. Viết hàm dạng không đệ quy 21
      2. Thêm một phần tử x trong cây21
         1. Viết hàm dạng đệ quy22
         2. Viết hàm dạng không đệ quy 23
      3. Tạo cây từ mảng a có sẵn dữ liệu 24
      4. Tạo cây tự động chứ n số nguyên 25
      5. Xóa một phần tử x trong cây27
      6. Xóa toàn bộ cây29
3. Cây nhị phân tìm kiếm cân bằng30
   1. Khái niệm và ví dụ30
   2. Cây AVL (AVL Tree)30
      1. Định nghĩa và ví dụ30
      2. Các trường hợp cây bị mất cân bằng31
         1. Cây T lệch về bên trái31
4. Cây T lệch về bên trái của cây con trái (LL: Left - Left) 31
5. Cây T lệch về bên phải của cây con trái (LR: Left – Right) 32
6. Cây con trái của T cân bằng (LB: Left – Balance) 32
   * + 1. Cây T lệch về bên phải32
7. Cây T lệch về bên trái của cây con phải (RL: Right - Left) 33
8. Cây T lệch về bên phải của cây con phải (RR :Right– Right) 33
9. Cây con trái của T cân bằng (RB: Right – Balance) 34
   * 1. Chỉ số cân bằng của một nút34
        1. Định nghĩa và ví dụ35
     2. Các thao tác trên cây AVL36
        1. Cân bằng lại cây 36
10. Trường hợp 1 : 36

− Cân bằng lại LL (Left – Left) 37

− Cân bằng lại LR (Left – Right) 38

− Cân bằng lại LB (Left – Blance) 39

1. Trường hợp 2 : 40

− Cân bằng lại RL (Right – Left) 41

− Cân bằng lại RB (Right-Balance) 41

− Cân bằng lại khi bị mất cân bằng 42

* + - 1. Xóa một phần tử x trong cây 43
      2. Tạo cây nhị phân cân bằng AVL 48

1. Cây B-Tree51
   1. Khái niệm và ví dụ51
   2. Các phép toán 52
      1. Phép toán tìm kiếm 52
      2. Duyệt cây53
      3. Thêm node mới54
      4. Tách node 56
      5. Tách node 57
2. Cây đỏ đen 57
   1. Khái niệm và ví dụ57

5.2 Cấu trúc dữ liệu của cây đỏ đen 57

* 1. Các phép toán 58
     1. Phép chèn58

5.3.2 Phép xóa60

CHƯƠNG 2

GỬI FILE ĐÍNH KÈM

KẾT LUẬN62

* 1. Đánh giá đề tài 62
  2. Những công việc đã làm được62
  3. Ưu điểm của hệ thống trong đề tài63
  4. Khuyết điểm của hệ thống trong đề tài63
  5. Hướng phát triển của đề tài63

TÀI LIỆU THAM KHẢO

# TÓM TẮT ĐỒ ÁN

***Tên đề tài***: Cài đặt mô phỏng Cấu trúc dữ liệu Cây

***Tóm tắt*:**

Trình bày tóm tắt phần lý thuyết về các loại cấu trúc dữ liệu cây (nhị phân, nhị phân tìm kiếm, nhị phân tìm kiếm cân bằng (AVL), Cây B-Tree, Cây đỏ đen), có ví dụ minh họa.

Cài đặt mô phỏng trực quan bằng đồ họa các thao tác trên từng loại cây, cho phép người dùng thay đổi dữ liệu input.

# DANH MỤC CÁC KHÁI NIỆM LIÊN QUAN

|  |  |
| --- | --- |
| **TÊN** | **ĐỊNH NGHĨA** |
| Bậc của một nút | Là số cây con của nút đó (nút lá có bậc bằng 0). |
| Bậc của một cây | Là bậc lớn nhất của các nút trong cây (số cây con tối đa của một nút trong cây). Cây có bậc n thì gọi là cây n - phân |
| Nút gốc | Không có nút cha. |
| Không có nút cha. | Không có nút con hay nút có bậc bằng 0. |
| Nút trong (nút nhánh) | Không phải nút lá và nút gốc hay nút có bậc khác 0. |
| Nút trước và nút sau của một nút | Nút T được gọi là nút trước của nút S nếu cây con có gốc là T chứa cây con có gốc là S. Khi đó, nút S được gọi là nút sau của nút T |
| Nút cha và nút con của một nút | Nút B được gọi là nút cha của nút C nếu nút B là nút trước của nút C và mức của nút C lớn hơn mức của nút B là 1 mức. Khi đó, nút C được gọi là nút con của nút B |
| Nút anh em (nút đồng cấp) | Các nút có cùng một nút cha. |
| Chiều dài đường đi của một nút | Là số đỉnh (số nút) tính từ nút gốc để đi đến nút đó. Như vậy, chiều dài đường đi của nút gốc luôn luôn bằng 1, chiều dài đường đi tới một nút bằng chiều dài đường đi tới nút cha nó cộng thêm 1. |
| Độ dài đường đi của một cây | Được định nghĩa là tổng các độ dài đường đi của tất cả các nút của cây. |
| Mức của một nút | Là độ dài đường đi từ gốc đến nút đó. |

# LỜI MỞ ĐẦU

“Cấu trúc dữ liệu và giải thuật” là môn học chính yếu và là kiến thức nền tảng của chuyên ngành Công nghệ Thông tin. Nhằm xây dựng một giáo trình vừa đảm bảo tính chuẩn mực của sách giáo khoa, vừa đáp ứng nhu cầu thực hành của sinh viên. Chúng tôi đã tham khảo nhiều tài liệu có giá trị của nhiều tác giả trong và ngoài nước nhằm cung cấp kiến thức về môn học “Cấu trúc dữ liệu và giải thuật” một cách có hệ thống, nhiều vấn đề được minh họa trực quan và hướng dẫn theo từng bước lập trình cụ thể cho sinh viên.

Cây là một cấu trúc rất quan trọng và được dùng rất nhiều trong các giải thuật. Trong chương này ta sẽ tìm hiểu các khái niệm cơ bản về cây, các phép toán quan trọng trên cây, biểu diễn cây trên máy tính. Cây có nhiều ứng dụng trong đời sống hàng ngày, chẳng hạn như tổ chức các quan hệ họ hàng trong gia phả, mục lục của một sách, … Chúng ta đã tìm hiểu về giải thuật tìm kiếm tuyến tính và tìm kiếm nhị phân. Đối với giải thuật tìm kiếm tuyến tính ta có thể đưa vào danh sách liên kết, nhưng giải thuật tìm kiếm nhị phân thì không thể. Trong phần này, chúng ta xét đến cây nhị phân tìm kiếm, mà việc tìm kiếm nhị phân có thể thực hiện trên đó, là một dạng của danh sách liên kết

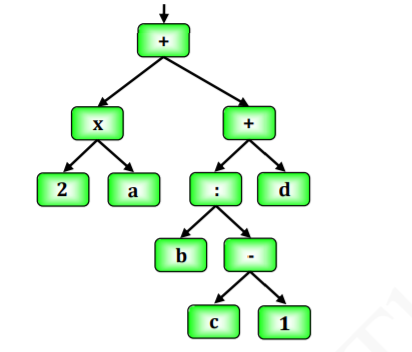
# CHƯƠNG 1: TRÌNH BÀY VỀ CẤU TRÚC DỮ LIỆU CÂY

## 1. Cây nhị phân .

### 1.1. Khái niệm và ví dụ .

Cây nhị phân (tiếng Anh: binary tree) là một cấu trúc dữ liệu cây mỗi nút có nhiều nhất hai nút con , được gọi là *con trái* (*left child*) và *con phải* (*right child*). Một định nghĩa đệ quy chỉ sử dụng các khái niệm lý thuyết tập hợp là cây nhị phân không trống là một tuple (*L*, *S*, *R*), với *L* và *R* là các cây nhị phân hay tập hợp rỗng và *S* là tập đơn (singleton set)

Cây nhị phân biểu diễn biểu thức (2 × a) + [b: (c – 1) + d] như sau:



Hình 1.1 Ví dụ cây nhị phân biểu diễn biểu thức

### 1.2. Biểu diễn cây nhị phân

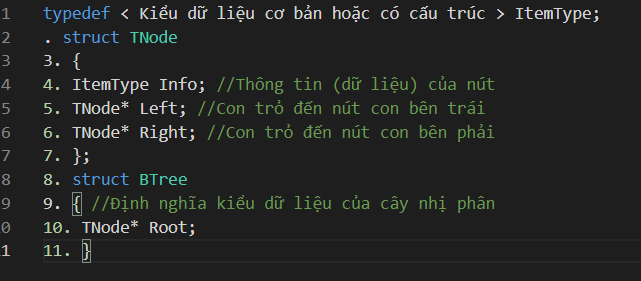
## Để biểu diễn cây nhị phân trong bộ nhớ máy tính chúng ta có thể sử dụng danh sách có hai mối liên kết để quản lý địa chỉ của hai nút gốc cây con (cây con trái và cây con phải). Như vậy cấu trúc dữ liệu của cây nhị phân tương tự như cấu trúc dữ liệu của danh sách liên kết đôi nhưng về cách thức liên kết thì khác nhau.

## Mỗi nút (phần tử) của cây nhị phân ứng với một biến động gồm ba thành phần:

## − Thông tin (dữ liệu) lưu trữ tại nút: Info.

## − Ðịa chỉ nút gốc của cây con trái trong bộ nhớ: Left.

## − Ðịa chỉ nút gốc của cây con phải trong bộ nhớ: Right

Code : 

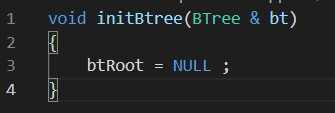
Hình 1.2 : Biểu diễn cây nhị phân

### 1.3. Các thao tác trên cây nhị phân

*1.3.1 Khởi tạo cây nhị phân rỗng*

Việc khởi tạo cây nhị phân chỉ đơn giản là khởi gán cho con trỏ quản lý địa chỉ nút gốc một giá trị NULL. Hàm khởi tạo cây nhị phân như sau:

Code :

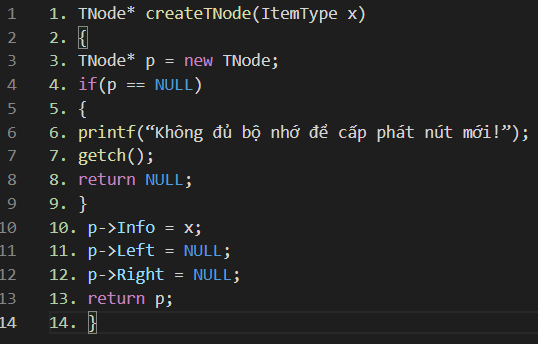


Hình 1.3.1 Khởi tạo nhị phân rỗng .

*1.3.2 Tạo nút chứa giá trị x*

Hàm tạo mới một nút có thành phần dữ liệu là x, hàm trả về con trỏ trỏ tới địa chỉ của nút mới tạo. Nếu không đủ bộ nhớ để tạo thì hàm sẽ trả về con trỏ NULL.

Code:

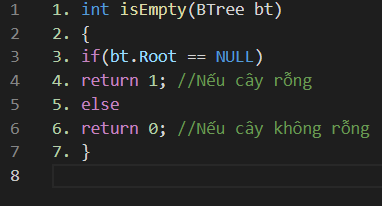


Hình 1.4.1 Tạo nút chứa giá trị x

*1.3.3 Kiểm tra cây nhị phân rỗng .*

Kiểm tra cây nhị phân đã có phần tử nào hay chưa? Nghĩa là kiểm tra con trỏ Root có bằng NULL hay không? Hàm trả sẽ về 1 nếu cây nhị phân chưa có phần tử nào (rỗng), ngược lại (không rỗng) thì hàm sẽ trả về 0

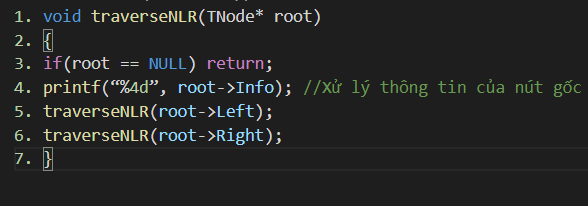
Code:

  
Hình 1.5.1 Kiểm tra cây nhị phân rỗng

*1.3.4 Duyệt theo thứ tự gốc trước (Node -Left-Right)*

Kiểu duyệt này trước tiên thăm nút gốc, sau đó thăm các nút của cây con trái, rồi cuối cùng thăm các nút của cây con phải.

Code :

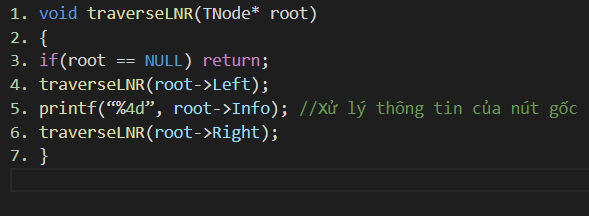


Hình 1.6.1 Duyệt theo thứ tự gốc trước (Node -Left-Right)

*1.3.5 Duyệt theo thứ tự gốc giữa (Left -Node -Right)*

Kiểu duyệt này trước tiên thăm các nút của cây con trái, sau đó thăm nút gốc, rồi cuối cùng thăm các nút của cây con phải

Code :

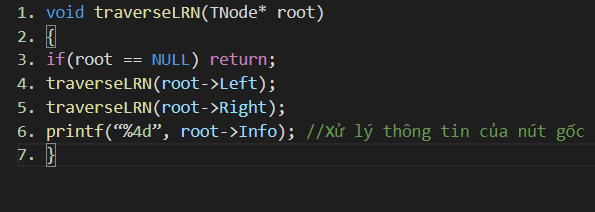


Hình 1.7.1 Duyệt theo thứ tự gốc giữa (Left -Node -Right)

*1.3.6 Duyệt theo thứ tự nút gốc sau (Left - Right - Node)*

Kiểu duyệt này trước tiên thăm các nút của cây con trái, sau đó thăm các nút của cây con phải, rồi cuối cùng thăm nút gốc

Code :

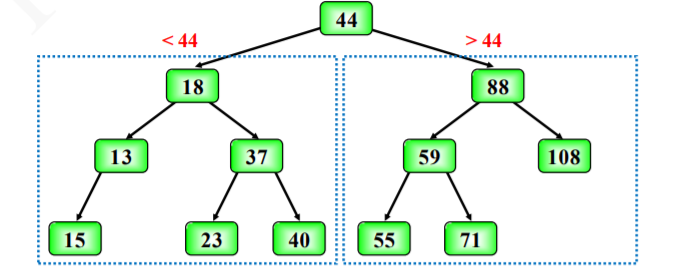


Hình 1.8.1 Duyệt theo thứ tự nút gốc sau (Left - Right - Node)

## 2. Cây nhị phân tìm kiếm

### 2.1. Khái niệm và ví dụ .

Cây nhị phân tìm kiếm là cây nhị phân có thành phần khóa của mỗi nút lớn hơn thành phần khóa của tất cả các nút trong cây con trái và nhỏ hơn thành phần khóa của tất cả các nút trong cây con phải của nó.



### 2.2. Biểu diễn cây nhị phân tìm kiếm

Khóa nhận diện (Info) của các nút trong cây nhị phân tìm kiếm đôi một khác nhau (không có hiện tượng trùng khóa). Nút ở bên trái nhất là nút có giá trị khóa nhận diện nhỏ nhất và nút ở bên phải nhất là nút có giá trị khóa nhận diện lớn nhất trong cây nhị phân tìm kiếm. Trong một cây nhị phân tìm kiếm thứ tự duyệt cây Left - Node - Right là thứ tự duyệt theo sự tăng dần các giá trị của Info trong các nút và thứ tự duyệt cây Right - Node - Left là thứ tự duyệt theo sự giảm dần các giá trị của Key trong các nút. Nhờ ràng buộc về khóa trên Cây nhị phân tìm kiếm, việc tìm kiếm trở nên có định hướng. Hơn nữa, do cấu trúc cây, việc tìm kiếm trở nên nhanh đáng kể. Nếu số nút trên cây là n thì chi phí tìm kiếm trung bình chỉ khoảng log2n. Trong thực tế, khi xét đến cây nhị phân chủ yếu ta xét Cây nhị phân tìm kiếm

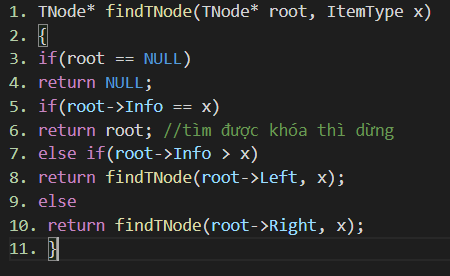
### 2.3. Các thao tác trên cây nhị phân tìm kiếm

*2.3.1 Tìm kiếm phần tử x trong cây*

Để thực hiện thao tác này chúng ta sẽ vận dụng giải thuật tìm kiếm nhị phân: Do đặc điểm của cây nhị phân tìm kiếm thì tại một nút, nếu Info của nút này khác với x thì x chỉ có thể tìm thấy như sau: − Hoặc trên cây con trái của nút này nếu x nhỏ hơn Info của nó. − Hoặc trên cây con phải của nút này nếu x lớn hơn Info của nó.

2.3.1.1 Viết hàm dạng đệ quy

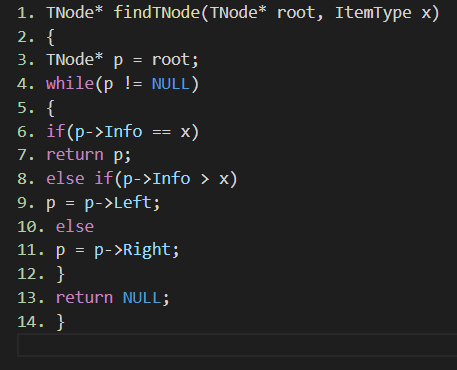
Code:



Hình 2.3.1.1 Dạng đệ quy

2.3.1.2 Viết hàm dạng không đệ quy

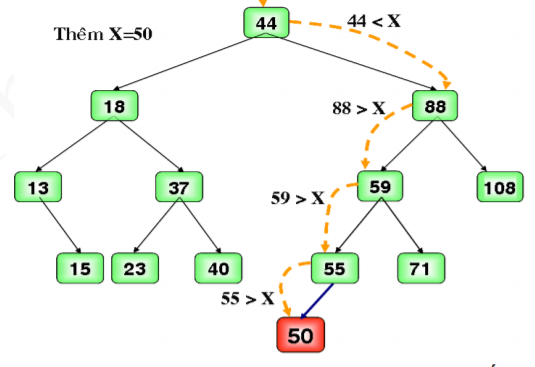
Code :



Hình 2.3.1.2 Dạng không đệ quy

*2.3.2 Thêm một phần tử x vào cây*

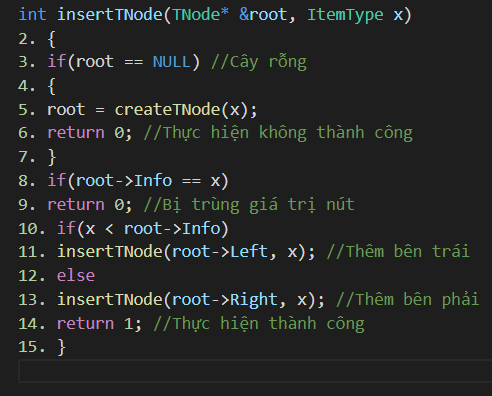
Giả sử cần thêm một nút có thành phần dữ liệu (Info) là x vào trong cây nhị phân tìm kiếm sao cho sau khi thêm cây vẫn là một cây nhị phân tìm kiếm. Trong thao tác này trước hết phải tìm kiếm vị trí cho phép thêm, sau đó mới tiến hành thêm nút mới vào cây (do vậy giải thuật còn được gọi là giải thuật tìm kiếm và thêm vào cây).



Hình 2.3.2 : Ví dụ thêm một nút vào cây nhị phân tìm kiếm

2.3.2.1 Viết hàm dạng đệ quy

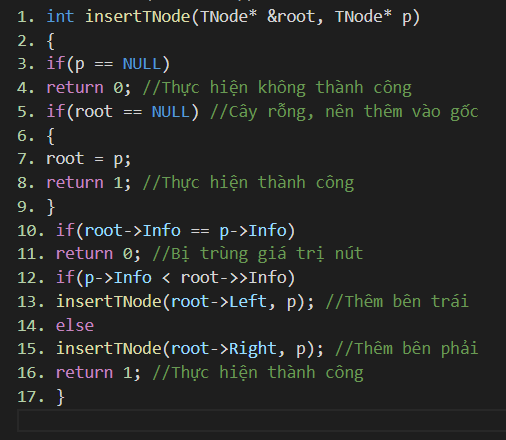
Code :



Hình 1.3.2.1.1 Viết hàm dạng đệ quy

Hoặc nếu truyền vào hàm là con trỏ nút thì hàm được định nghĩa lại như sau

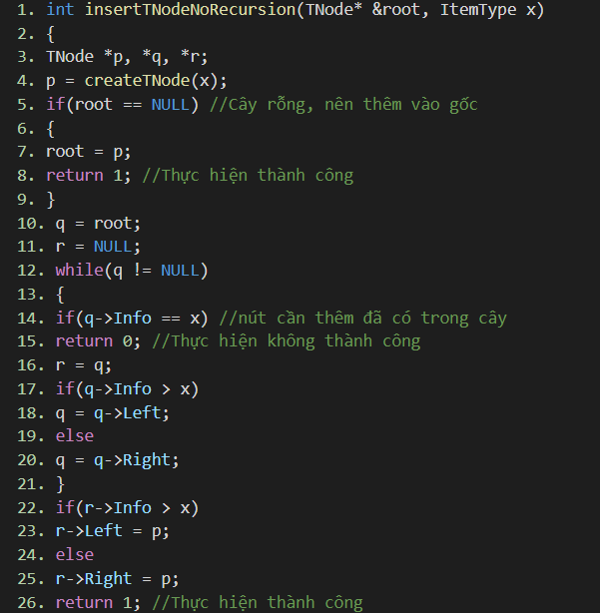
Code :



Hình 1.3.2.1.2 Viết hàm dạng đệ quy

2.3.2.2 Viết hàm không đệ quy

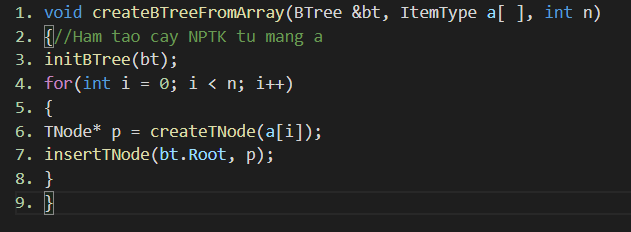
Code :



Hình 2.3.2..2 Viết hàm không đệ quy

*2.3.3 Tạo cây từ mảng a có sẵn dữ liệu*

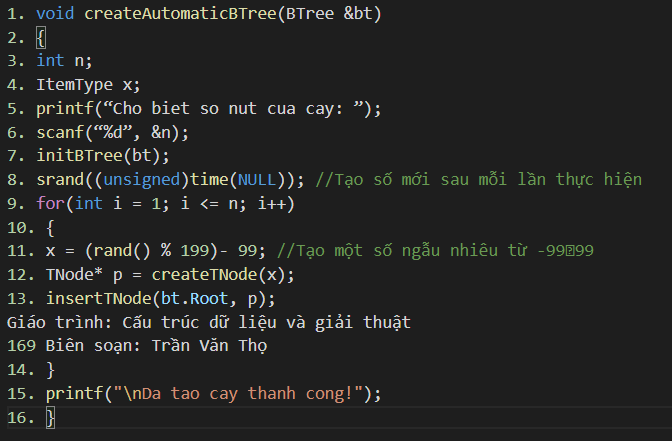
Giả sử ta đã có một mảng a chứa n phần tử, ta có thể tạo một cây nhị phân tìm kiếm từ mảng này.

Code : 

*2.3.4 Tạo cây tự động chứa n số nguyên*

Tạo một cây nhị phân tìm kiếm bằng cách phát sinh ngẫu nhiên n số nguyên và lần lược thêm chúng vào cây.

Code :

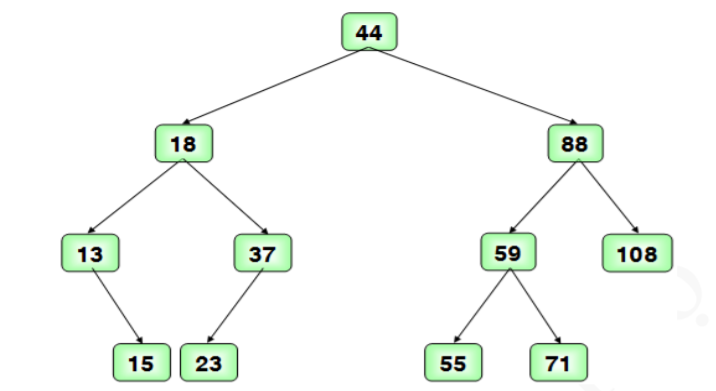


Hình 2.3.4 Tạo cây tự động chứa n số nguyên

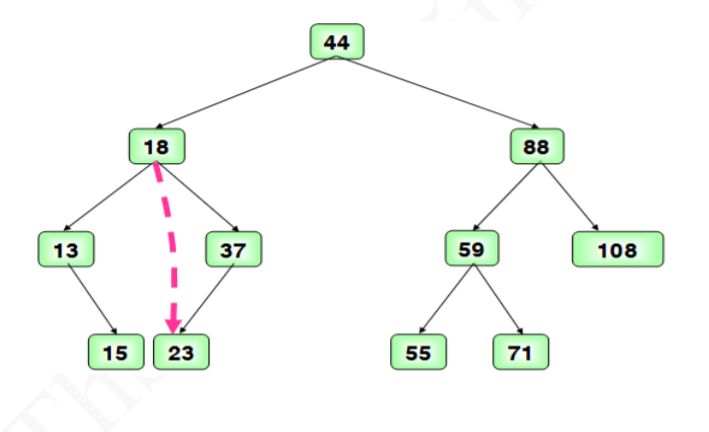
*2.3.5 Xóa một phần tử x trong cây*

Giả sử cần hủy nút có thành phần dữ liệu (Info) là x ra khỏi cây nhị phân tìm kiếm. Điều đầu tiên trong thao tác này là phải tìm kiếm địa chỉ của nút cần hủy là DelTNode, sau đó mới tiến hành hủy nút có địa chỉ là DelTNode này nếu tìm thấy (do vậy giải thuật này còn được gọi là giải thuật tìm kiếm và loại bỏ trên cây). Quá trình tìm kiếm đã trình bày ở trên, phần này chỉ trình bày thao tác hủy khi tìm thấy nút có địa chỉ DelTNode (DelTNode→Info == x) và trong quá trình tìm kiếm sẽ phải giữ địa chỉ nút cha của nút cần hủy là FatherDelTNode.

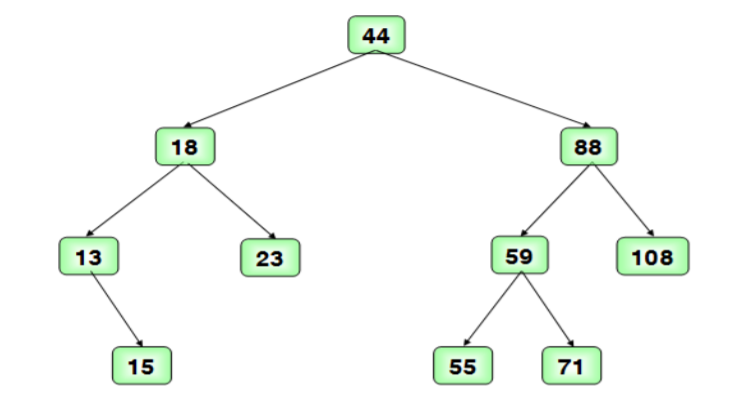
Ví dụ : Minh họa cho trường hợp phần tử bị xóa một cây có một cây con



Hình 2.3.5.1 Giả sử một cây nhị phân

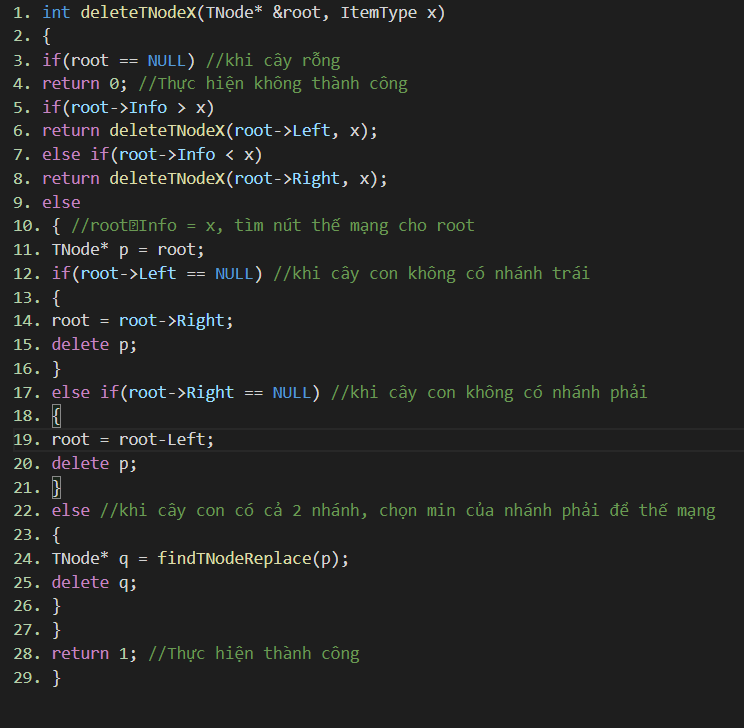


Hình 2.3.5.2 Xóa nút có giá trị x = 37 trong cây

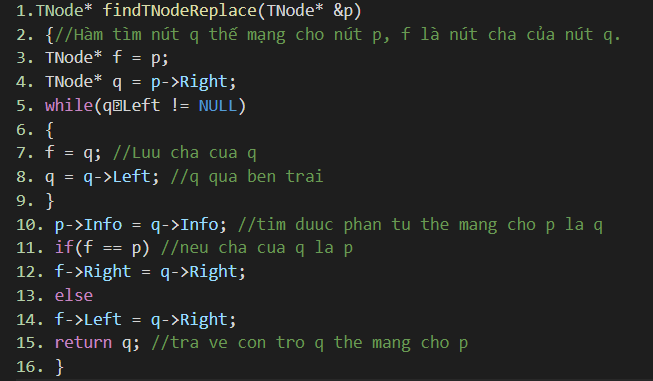


Hình 2.3.5.3 cây nhị phân tìm kiếm sau khi xóa nút

Code :



Hình 2.3.5.4 Tìm phần tử p có khóa x

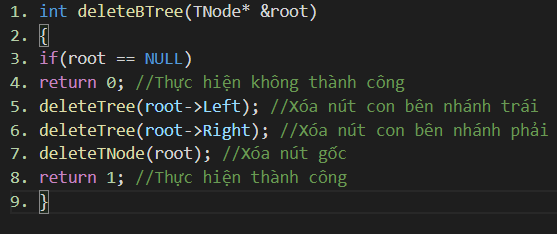


Hình 2.3.5.5 Tìm nút con thế mạng (phải viết trước hàm deleteTNodeX):

*2.3.6 Xóa toàn bộ cây*

Thao tác chỉ đơn giản là việc thực hiện nhiều lần thao tác hủy một nút trên cây nhị phân tìm kiếm cho đến khi cây trở thành cây rỗng

Code :



Hình 2.3.6 Xóa toàn bộ cây

Nhận xét :

Tất cả các thao tác tìm kiếm, xóa, thêm trên cây nhị phân tìm kiếm đều có độ phức tạp trung bình O(log2(n)).

Cây nhị phân tìm kiếm là một cấu trúc dữ liệu cho phép tìm kiếm hiệu quả hơn hẳn danh sách liên kết, mà các thao tác không quá phức tạp. Cây nhị phân tìm kiếm là một trong những cấu trúc dữ liệu động thông dụng nhất. Tuy nhiên trong một số trường hợp nó suy biến thành danh sách liên kết

## 3. Cây nhị phân tìm kiếm cân bằng

### 3.1. Khái niệm và ví dụ .

Theo Adelson - Velskii và Landis đưa ra định nghĩa về cây cân bằng tương đối như sau:

− Cây cân bằng tương đối là một cây nhị phân thỏa mãn điều kiện là đối với mỗi nút của cây thì chiều cao của cây con trái và chiều cao của cây con phải của nút đó hơn kém nhau không quá một. Cây cân bằng tương đối còn được gọi là cây AVL (AVL Tree).

− Cây cân bằng hoàn toàn: Cây cân bằng hoàn toàn là một cây nhị phân thỏa mãn điều kiện là đối với mỗi nút của cây thì số nút ở cây con trái và số nút ở cây con phải của nút đó hơn kém nhau không quá một.

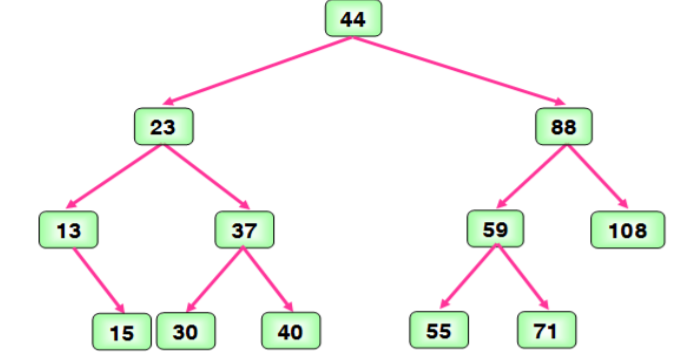
Như vậy, một cây cân bằng hoàn toàn chắc chắn là một cây cân bằng tương đối. Cân cân bằng hoàn toàn có n nút có chiều cao h = log2(n). Đây là lý do cho phép bảo đảm khả năng tìm kiếm nhanh trên cấu trúc dữ liệu này. Nhược điểm cây cân bằng hoàn toàn là một cấu trúc kém ổn định nên trong thực tế không thể sử dụng.

### 3.2 Cây AVL ( AVL Tree)

*3.2.1 Định nghĩa và ví dụ*

Cây nhị phân tìm kiếm cân bằng (AVL) là cây nhị phân tìm kiếm mà tại mỗi nút của nó chiều cao của cây con bên trái và chiều cao của cây con bên phải chênh lệch nhau không quá một

Ví dụ



### 

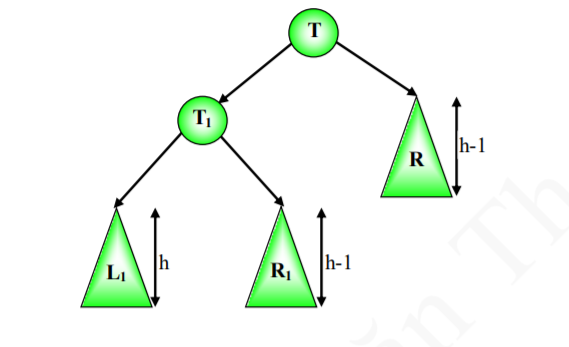
Hình 3.2.1 Ví dụ minh họa là một cây AVL

*3.2.2 Các trường hợp cây bị mất cân bằng*

*3.2.2.1 Trường hợp 1 : Cây T lệch về bên trái (có 3 khả năng).*

1. Cây T lệch về bên trái của cây con trái (LL: Left - Left):

− Chiều cao cây con phải R của cây T là (h - 1). − Chiều cao cây con trái T1 của cây T là (h + 1), trong đó chiều cao của cây con trái L1 của cây T1 là h và chiều cao của cây con phải R1 của cây T1 là (h - 1).

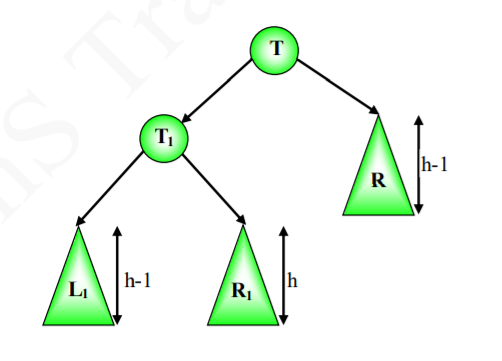


Hình 3.2.2.1 a. Minh họa cây AVL lệch trái LL

1. Cây T lệch về bên phải của cây con trái (LR: Left - Right):

− Chiều cao cây con phải R của cây T là (h - 1).

− Chiều cao cây con trái T1 của cây T là (h + 1), trong đó chiều cao của cây con trái L1 của cây T1 là (h - 1) và chiều cao của cây con phải R1 của cây T1 là h

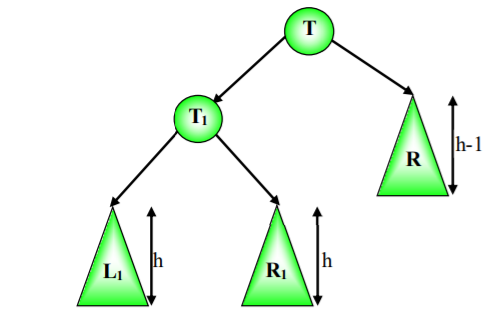


Hình 3.2.2.1 b. Minh họa cây AVL lệch trái LR

1. Cây con trái của T cân bằng (LB: Left - Balance):

− Chiều cao cây con phải R của cây T là (h - 1)

− Chiều cao cây con trái T1 của cây T là (h + 1), trong đó chiều cao của cây con trái L1 và cây con phải R1 của cây T1 đều là h



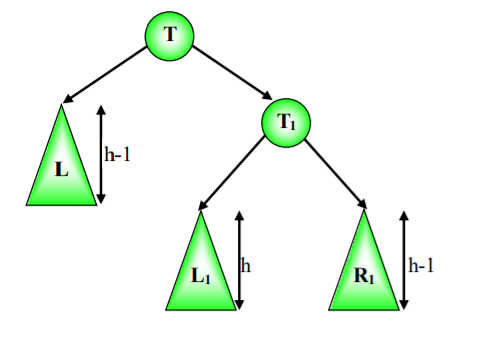
Hình 3.2.2.1 b. Minh họa cây AVL lệch trái LB

* + - 1. Trường hợp 2: Cây T lệch về bên phải (có 3 khả năng).

1. Cây T lệch về bên trái của cây con phải (RL: Right - Left):

− Chiều cao cây con trái L của cây T là (h - 1).

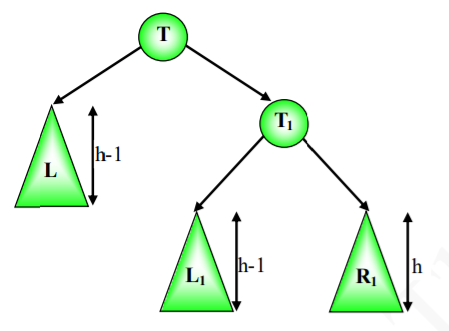
− Chiều cao cây con phải T1 của cây T là (h + 1), trong đó chiều cao của cây con trái L1 của cây T1 là h và chiều cao của cây con phải R1 của cây T1 là (h – 1)



Hình 3.2.2.2 a. Minh họa cây AVL lệch trái RL

1. Cây T lệch về bên phải của cây con phải (RR: Right – Right):

− Chiều cao cây con trái L của cây T là (h - 1)  
− Chiều cao cây con phải T1 của cây T là (h + 1), trong đó chiều cao của cây con trái L1 của cây T1 là (h - 1) và chiều cao của cây con phải R1 của cây T1 là h.

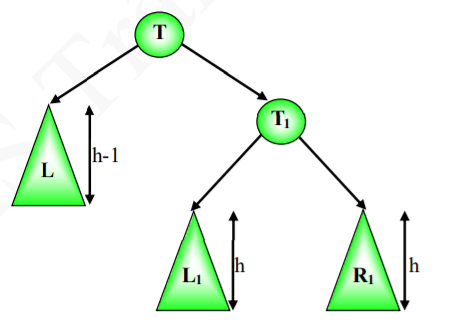


Hình 3.2.2.2 b. Minh họa cây AVL lệch trái RR

1. Cây con phải của T cân bằng (RB: Right - Balance):

− Chiều cao cây con trái L của cây T là (h - 1).

− Chiều cao cây con phải T1 của cây T là (h + 1), trong đó chiều cao của cây con trái L1 và cây con phải R1 của cây T1 đều là h.



Hình 3.2.2.2 c. Minh họa cây AVL lệch trái RB

* + 1. *Chỉ số cân bằng của một nút*

3.2.3.1 Định nghĩa và ví dụ

Chỉ số cân bằng của một nút là hiệu của chiều cao cây con bên phải và cây con bên trái của nó.

Đối với cây cân bằng thì chỉ số cân bằng (CSCB) của mỗi nút chỉ có thể mang một trong ba giá trị sau:

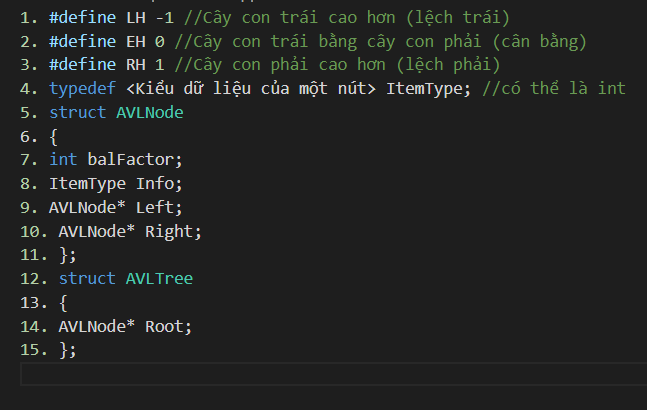
− CSCB(p) = 0 ⇔ Độ cao cây con trái (p) = Độ cao cây con phải (p).

− CSCB(p) = -1 ⇔ Độ cao cây con trái (p) > Độ cao cây con phải (p).

− CSCB(p) = 1 ⇔ Độ cao cây con trái (p) < Độ cao cây con phải (p). Chúng ta ký hiệu như sau: − p→balFactor = CSCB(p)  
 − hL: Độ cao cây con trái.

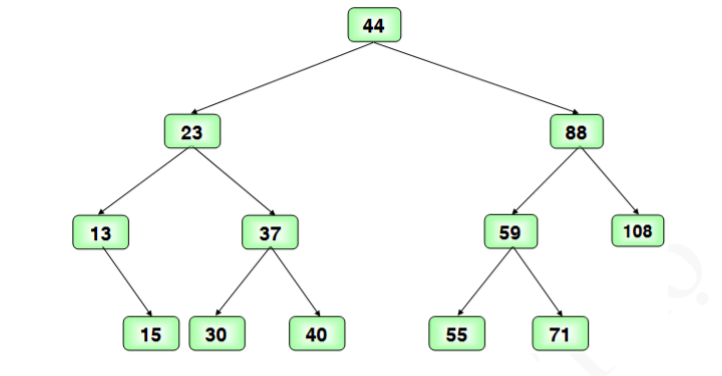
− hR: Độ cao cây con phải

Code :



Hình 3.2.3.1.1 : Chỉ số cân bằng một nút

Ví dụ :



Hình 3.2.3.1.2 : Minh họa cây AVL có nút mất cân bằng lệch trái

Xét nút p có giá trị bằng 88, thì hệ số cân bằng của nút p: CSCB(p) = -1.

* + 1. *Các thao tác trên cây AVL*

Khi thêm hay xóa một nút trên cây, có thể làm cho cây mất tính cân bằng, khi ấy sẽ phải tiến hành cân bằng lại cây. Có khả năng cây bị mất cân bằng khi thay đổi chiều cao:

− Thêm bên trái -> Lệch sang nhánh trái.

− Thêm bên phải ◊ Lệch sang nhánh phải.

− Hủy bên phải -> Lệch sang nhánh trái.

− Hủy bên trái -> Lệch sang nhánh phải. Vì vậy việc cân bằng lại cây là tìm cách bố trí lại cây sao cho chiều cao của hai cây con được cân đối với nhau:

− Kéo nhánh cao bù cho nhánh thấp.

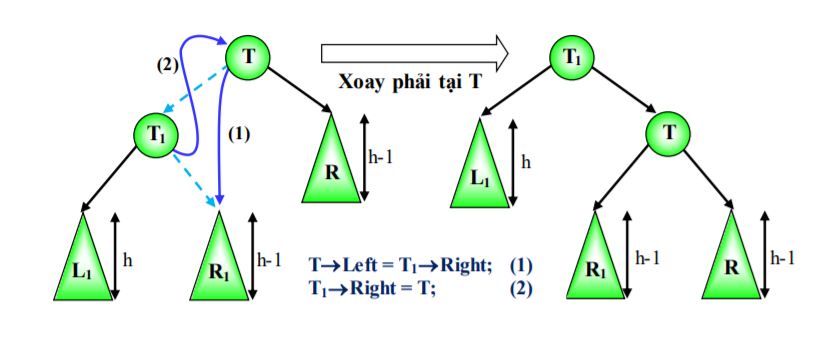
− Phải bảo đảm cây vẫn là nhị phân tìm kiếm.

3.2.4.1 Cân bằng lại cây

Tại nút T không cân bằng. Ta phải giải quyết hai trường hợp sau:

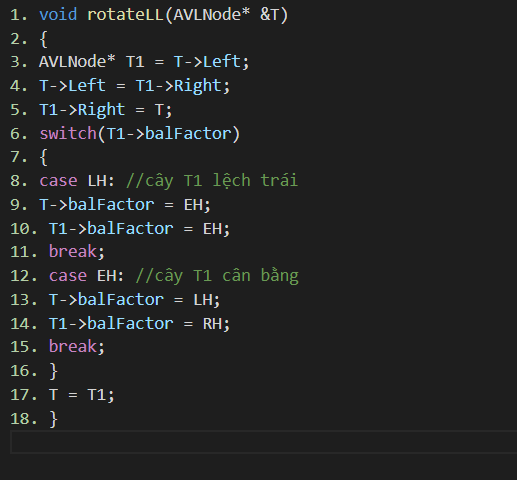
1. Trường hợp 1:

− Cân bằng lại LL (Left - Left):



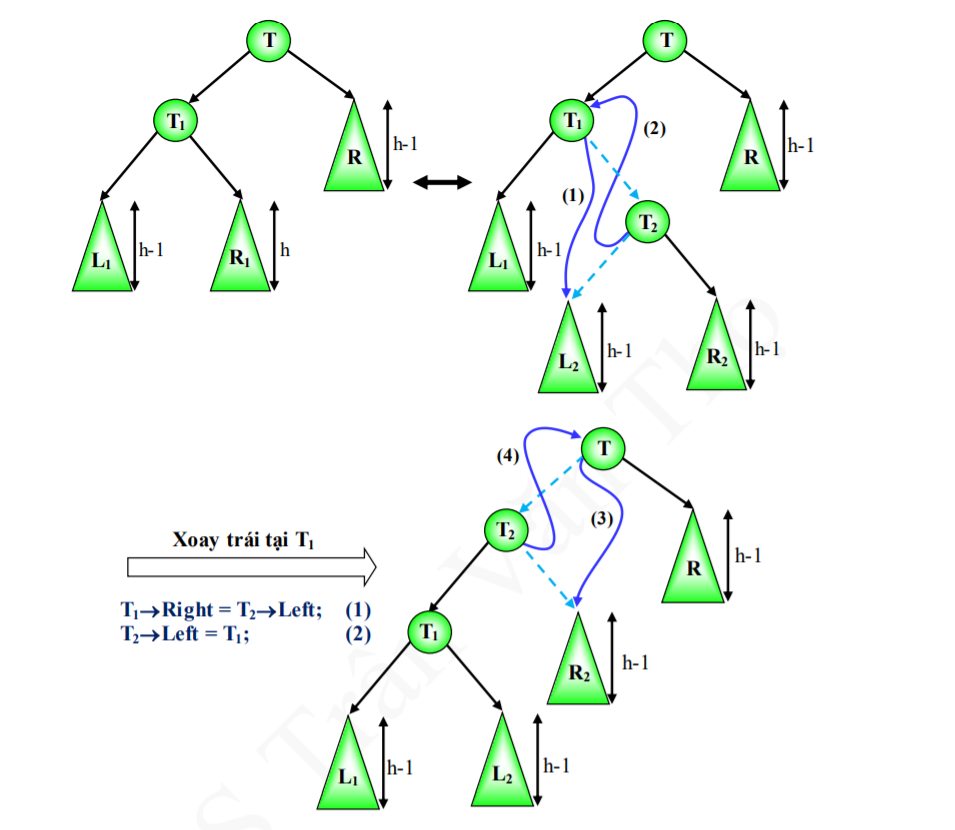
Hình 3.2.4.1 a.1 Cân bằng lại LL cho cây AVL

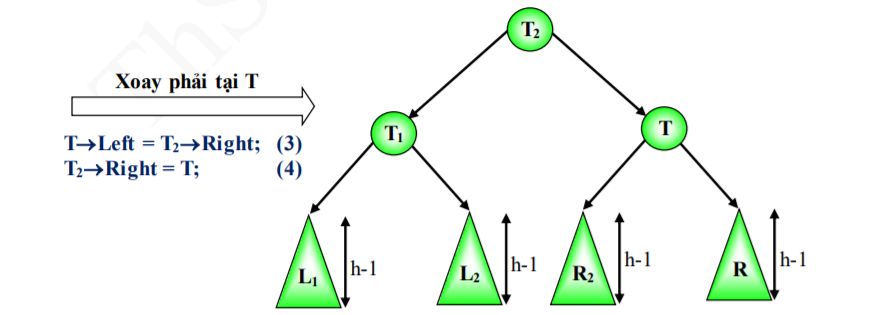
Code :



Hình 3.2.4.1 a.2 Cài đặt

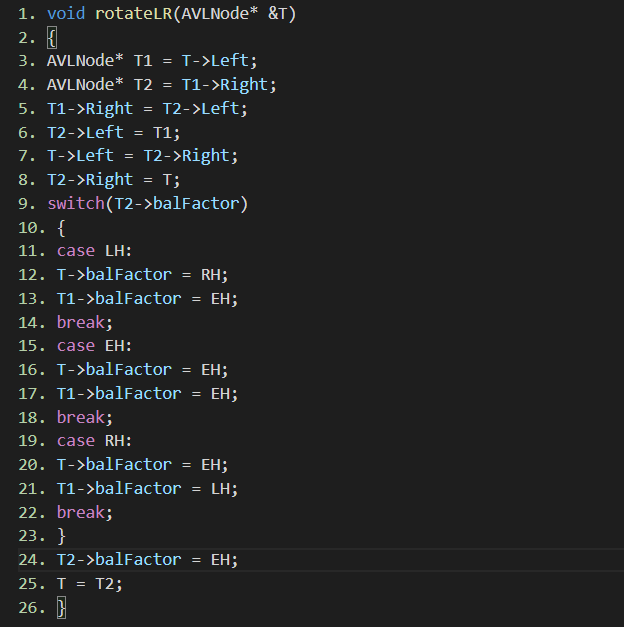
− Cân bằng lại LR (Left - Right):





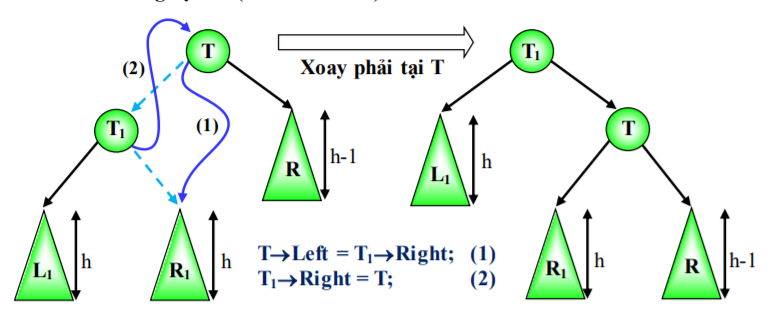
Hình 3.2.4.1 a.1 Cân bằng lại LR cho cây AVL

Code :



Hình 3.2.4.1 a.2 Cài đặt

− Cân bằng lại LB (Left - Balance):

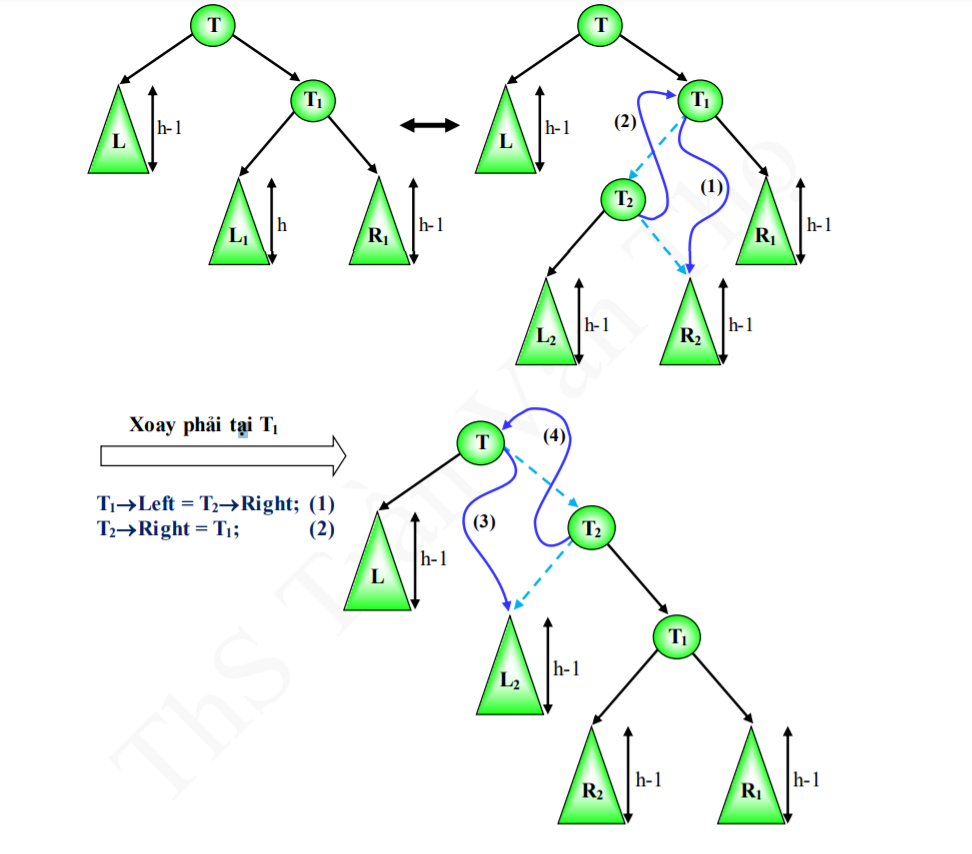


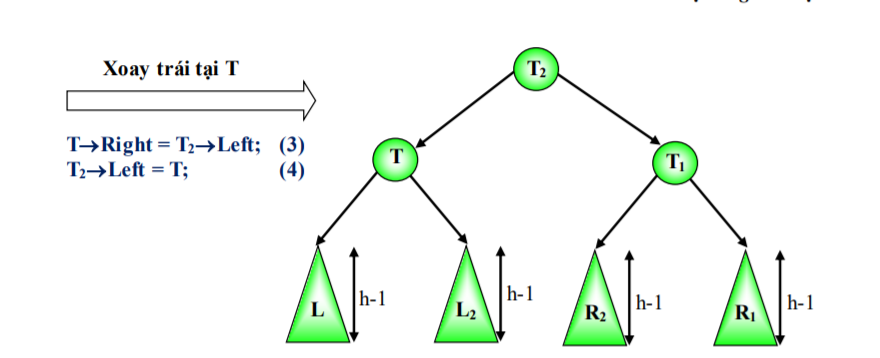
Hình 3.2.4.1 a.1 Cân bằng lại LB cho cây AVL

Code : Việc cân bằng lại cây trong trường hợp LB hoàn toàn giống như LL

1. Trường hợp 2:

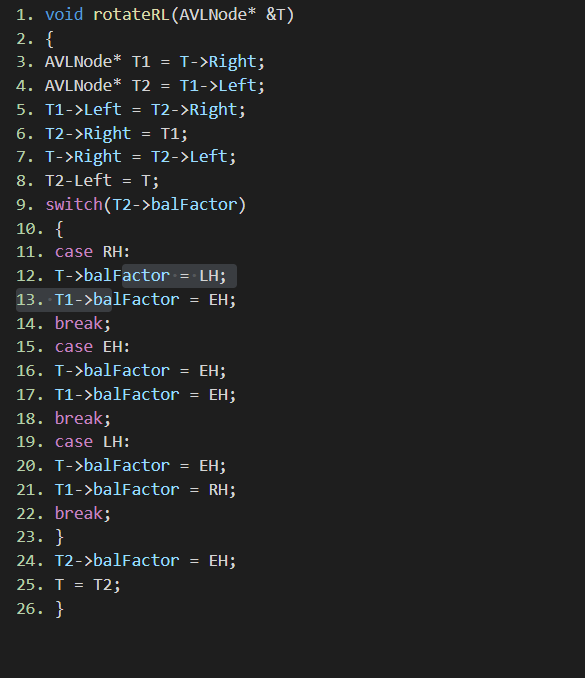
− Cân bằng lại RL (Right - Left):





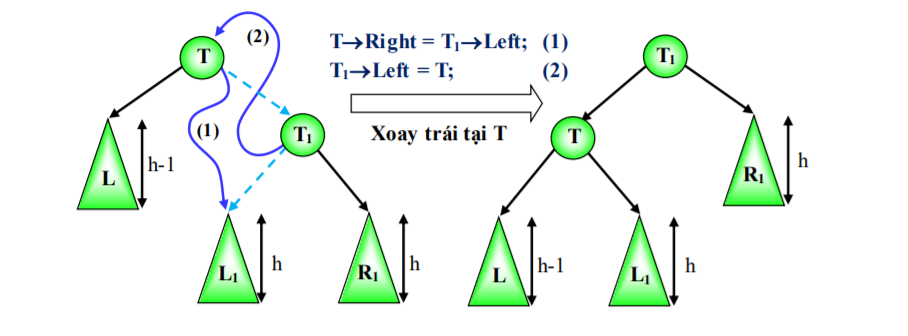
Hình 3.2.4.1 .b 1 Cân bằng lại RL cho cây AVL

Code :



Hình 3.2.4.1 .b 2 Cài đặt

− Cân bằng lại RB (Right - Balance):

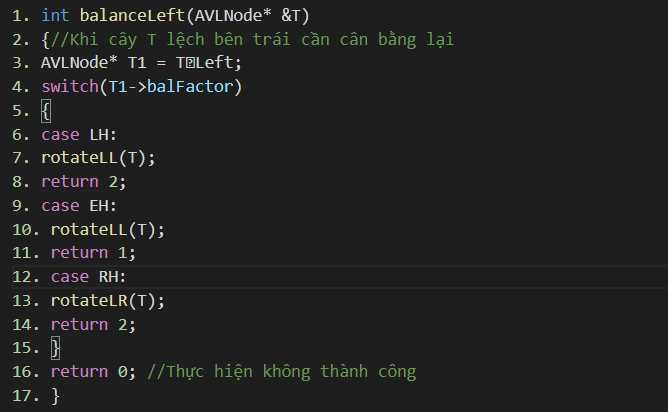


Hình 3.2.4.1 .b 1 Cân bằng lại RB cho cây AVL

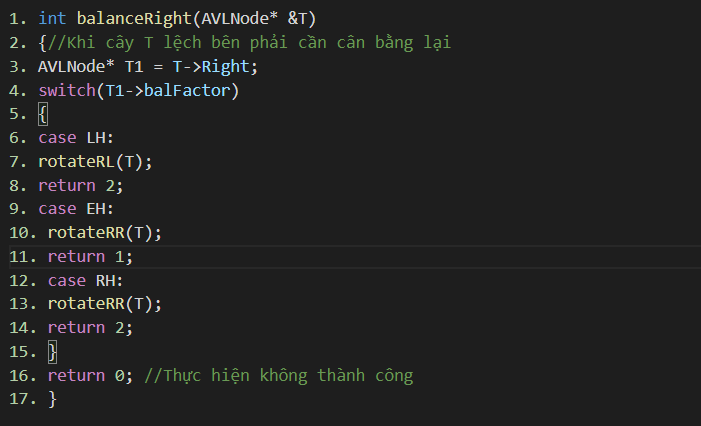
Code : Việc cân bằng lại cây trong trường hợp RB hoàn toàn giống như RR

1. Cân bằng lại cây AVL khi bị mất cân bằng:

Code :



Hình 3.2.4.1 .c 1 Khi cây T lệch bên trái cần cân bằng lại



Hình 3.2.4.1 .c 2 Khi cây T lệch bên phải cần cân bằng lại

* + - 1. Xóa một phần từ x trong cây

− Các trường hợp loại bỏ một phần tử trên cây cân bằng vẫn được giải quyết như khi loại bỏ trên cây nhị phân tìm kiếm.

− Nếu cây giảm chiều cao:

+ Lần ngược về gốc để phát hiện nút bị mất cân bằng.

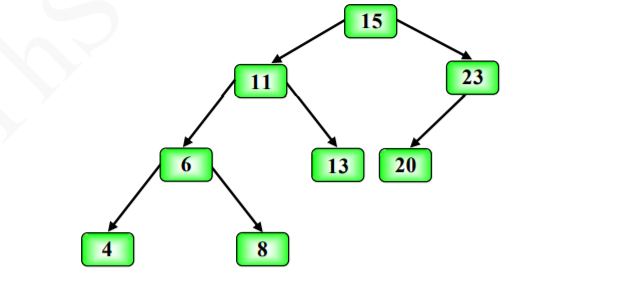
+ Tiến hành cân bằng lại nút đó bằng thao tác cân bằng thích hợp.

+ Tiếp tục lần ngược lên nút cha.

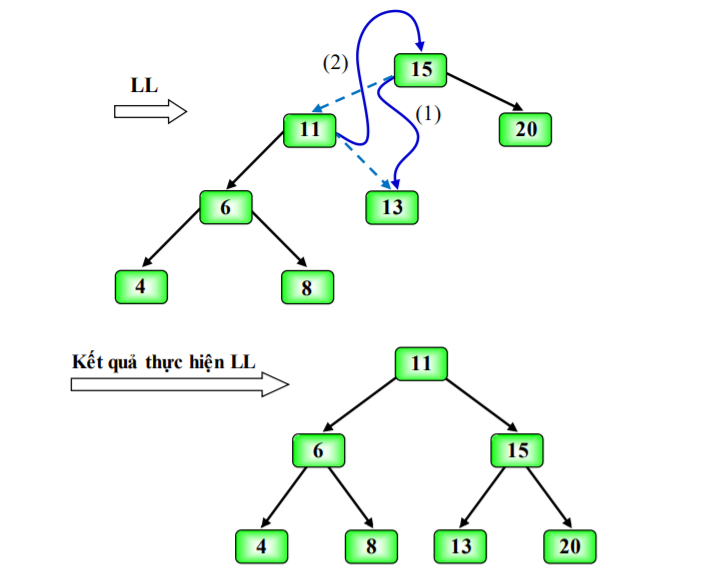
− Việc cân bằng lại có thể lan truyền lên tận gốc.

− Các trường hợp cân bằng lại giải quyết giống như thêm vào cây.

Ví dụ :

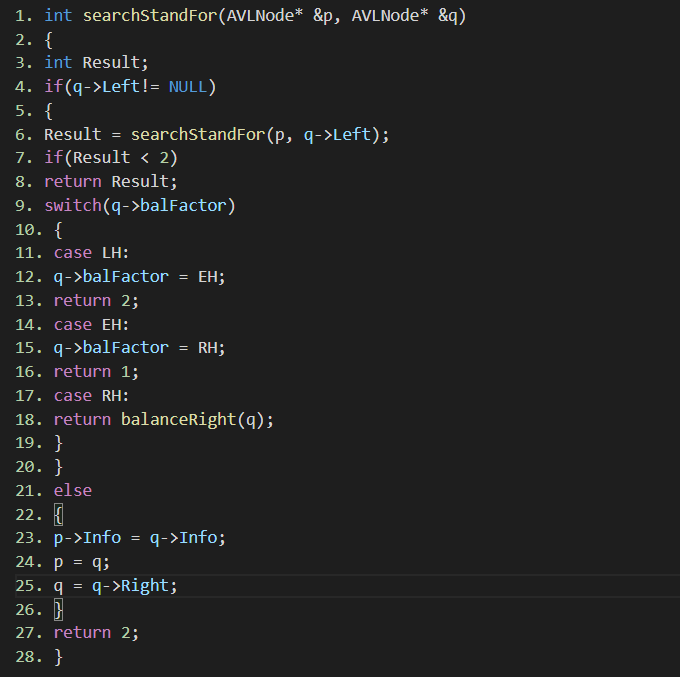


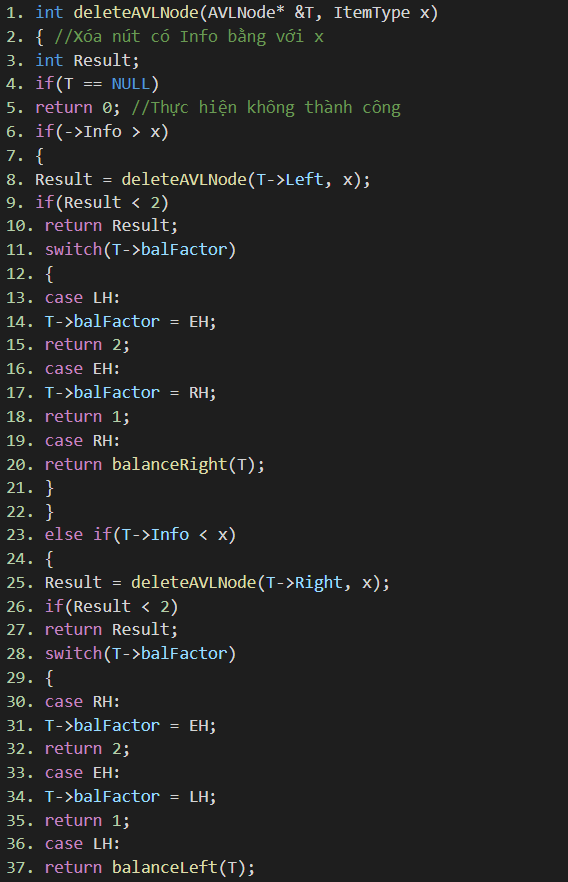
Hình 3.2.4.2.1 Hủy phần tử có giá trị 23

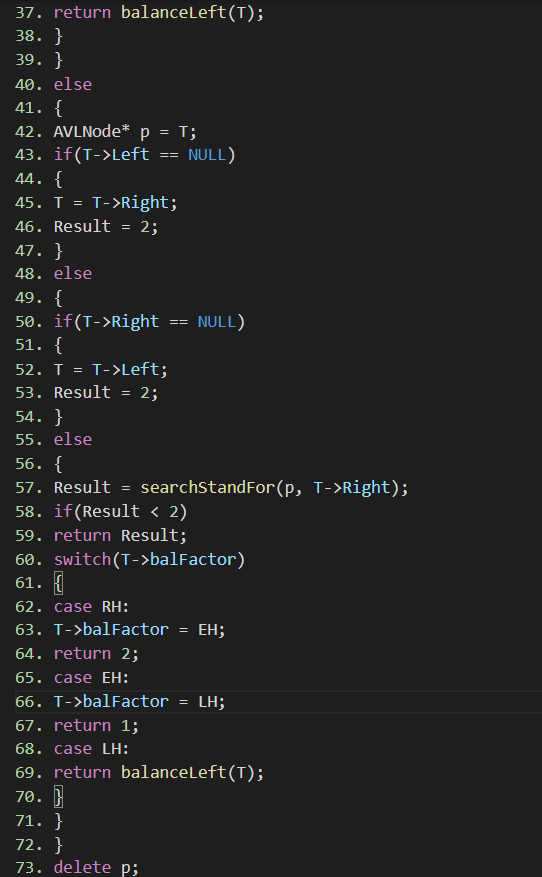


Hình 3.2.4.2.2 Xóa một nút từ cây AVL

Code







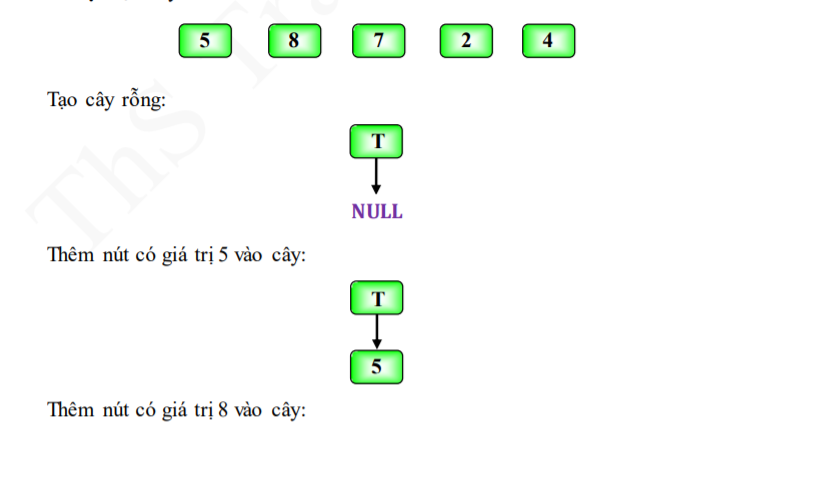


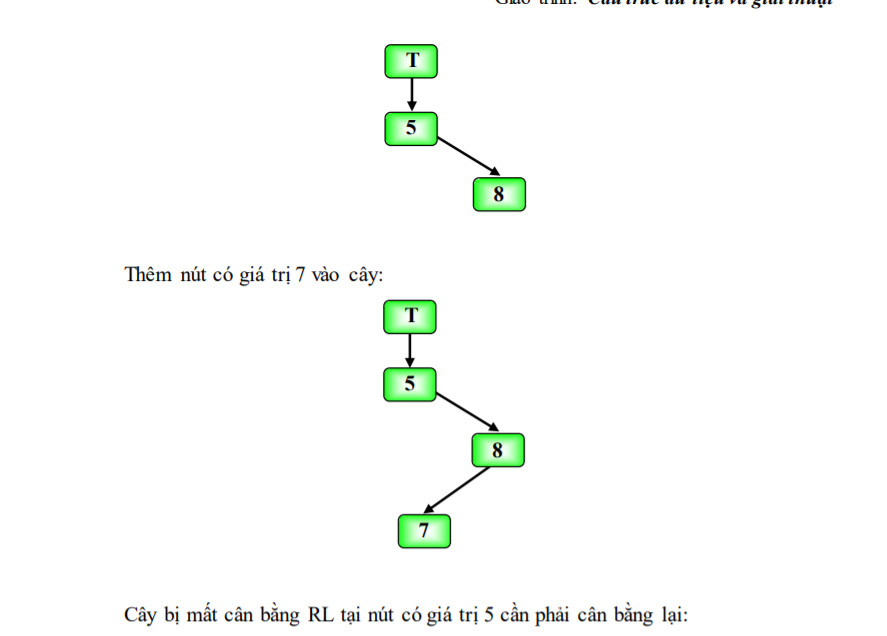
Hình 3.2.4.2.1 Cài đặt

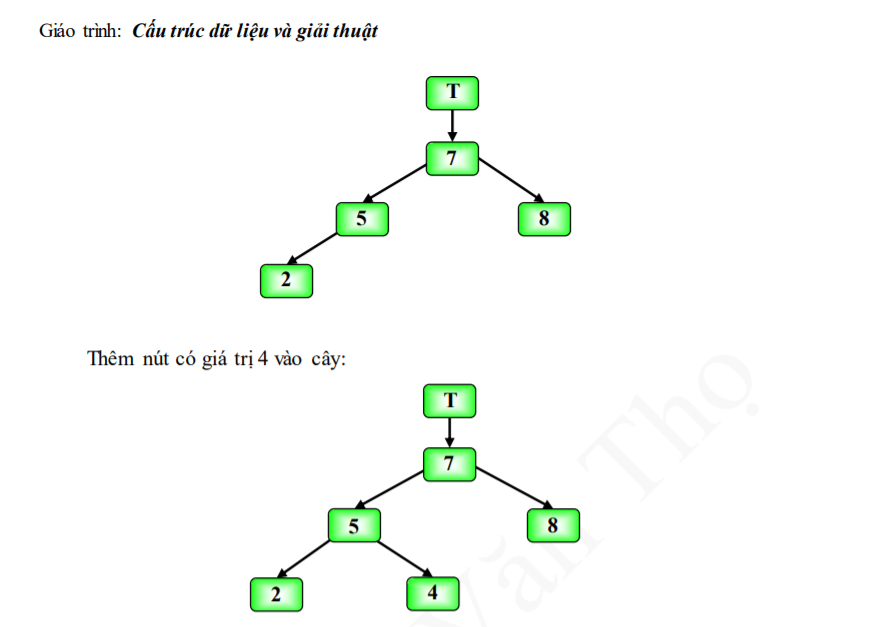
* + - 1. Tạo cây nhị phân cân bằng AVL

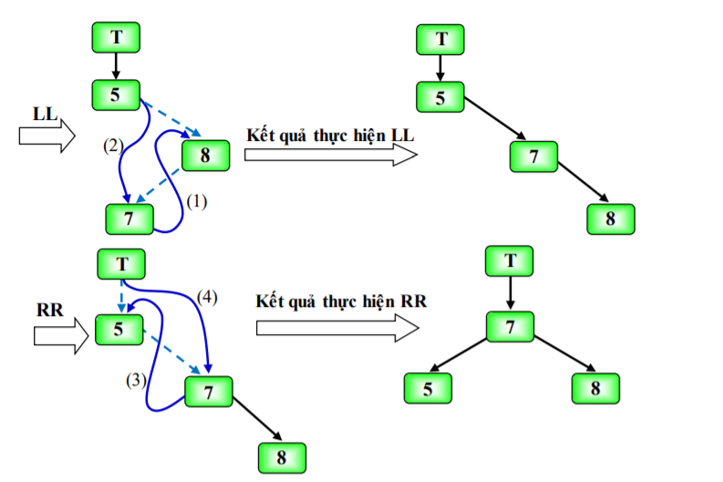
Việc tạo cây nhị phân cân bằng AVL là quá trình thêm lần lượt các phần tử (nút) vào cây, quá trình này sẽ dẫn đến việc mất cân bằng và phải cân bằng lại cây.

1. Ví dụ: Tạo cây AVL từ danh sách sau:



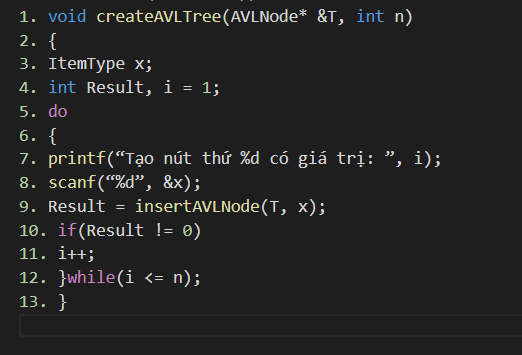






Hình 3.4.2.3.a Minh họa về cách tạo cây AVL

Code :



Hình 3.4.2.3.b Cài đặt

## 4. Cây B-Tree

### 4.1. Khái niệm và ví dụ .

B-tree là cấu trúc dữ liệu phù hợp cho việc lưu trữ ngoài do R.Bayer và E.M.McCreight đưa ra năm 1972.

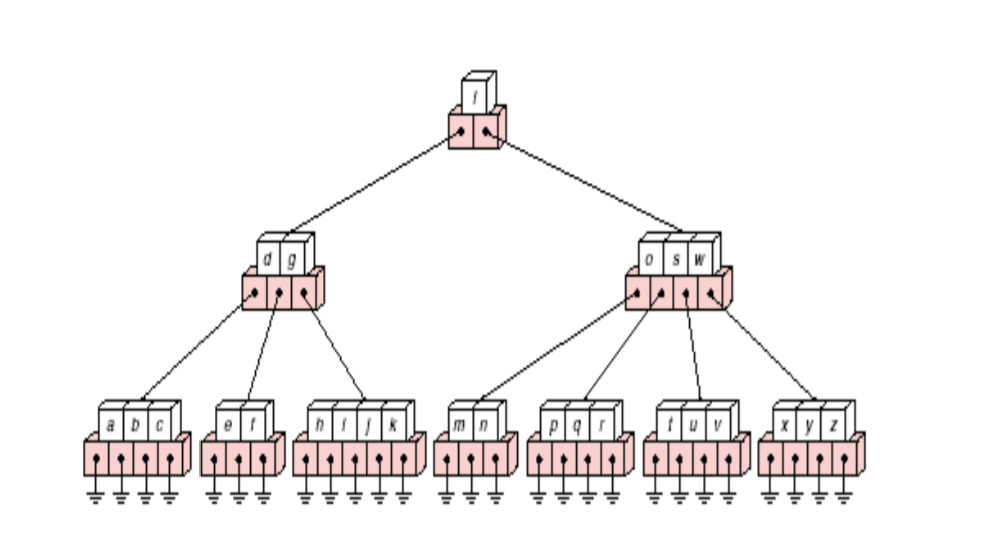
Một B-tree bậc m là cây nhiều nhánh tìm kiếm thỏa các điều kiện sau:

- Tất cả các node lá cùng mức.

-Tất cả các node trung gian (trừ node gốc) có nhiều nhất m cây con và có ít nhất m/2 cây con (khác rỗng).

- Mỗi node hoặc là node lá hoặc có k+1 cây con (k là số khoá của node này).

- Node gốc có nhiều nhất m cây con hoặc có thể có 2 cây con (Node gốc có 1 khoá và không phải là node lá) hoặc không chứa cây con nào(node gốc có 1 khoá và cũng là node lá).



Hình 4.1 a B-tree bậc 5 có 3 mức

Code :

typedef struct

{

int count; // số khoá của node hiện hành

int Key[Order-1];// mảng lưu trữ các khoá của node

int \*Branch[Order]; /\* các con trỏ chỉ đến các

cây con, Order-Bậc của cây\*/

} BNode; //Kiểu dữ liệu của node

typedef struct BNode \*pBNode // con trỏ node

pBNode Root // con tro node goc

**4.2 Các phép toán**

*4.2.1 Phép toán tìm kiếm*

Trả về vị trí nhỏ nhất của khóa trong nút hiện tại bắt đầu lớn hơn hay bằng k.

Code

int nodesearch (pBNode current, int k)

{

int i;

for(i=0;i<current->count && current->key[i] < k; i++);

return i;

}

Hình 4.2.1 a Ví dụ

Tìm khóa k trên B-Tree. Con trỏ current xuất phát từ gốc và đi xuống các nhánh cây con phù hợp để tìm khóa k có trong một nút current hay không.

Nếu có khóa k tại nút current trên cây:

-Biến found trả về giá trị TRUE

-Hàm search() trả về con trỏ chỉ nút current có chứa khóa k

-Biến position trả về vị trí của khóa k có trong nút current này

Nếu không có khóa k trên cây:

- Lúc này current=NULL và q (nút cha của current) chỉ nút lá có thể thêm khóa k vào nút này được.

- Biến found trả về giá trị FALSE

- Hàm search() trả về con trỏ q là nút lá có thêm nút k vào

- Biến position trả về vị trí có thể chèn khóa k vào nút lá q này

Code :

Hình 4.2.1 b Thực hiện phép toán

*4.2.2 Duyệt cây*

Code

pBNode search(int k, int &position, int &found)

{

int i;

pBNode current, q;

q = NULL; current = Root;

while (current !=NULL){

i = nodesearch (current, k);

if(i< current->count && k == current->key[i]) {

found = TRUE;

position = i; // vi trí tìm thấy khóa k

return(current);

}

q = current; current = current ->Branch[i];

}

found = FALSE; position = i;

return q;

}

Hình 4.2.2 Thực hiện phép duyệt cây

*4.2.3 Thêm node mới*

Quá trình thêm một khoá mới (newkey) vào Btree có thể được mô tả như sau:

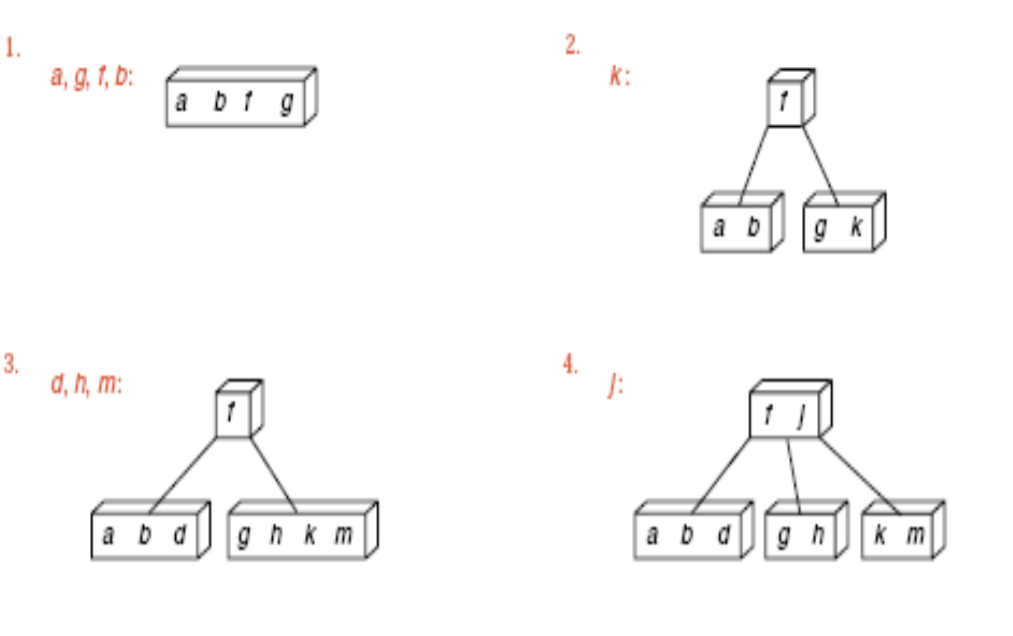
-Tìm node newkey nếu có trên cây thì kết thúc công việc này tại node lá (không thêm vào nữa)

-Thêm newkey vào node lá, nếu chưa đầy thì thực hiện thêm vào và kết thúc Node đầy là node có số khoá = (bậc của cây)-1

Khi node được thêm vào bị đầy, node này sẽ được tách thành 2 node cùng mức, khoá median sẽ được đưa vào node mới

- Khi tách node, khoá median sẽ được dời lên node cha, quá trình này có thể lan truyền đến node gốc

- Trong trường hợp node gốc bị đầy, node gốc sẽ bị tách và dẫn đến việc tăng trưởng chiều cao của cây.



Hình 4.2.3 a Mô tả

Khi thêm một khóa vào B-Tree chúng ta có thể viết như sau:

if(Root== NULL)

Root = makeroot(k);

else

{

s = search(k, &position, &timthay)

if (s!=NULL)

cout<<"Khong them vao duoc";

else

insert (s,k,position)

}

Hình 4.2.3 b

-Thêm khóa k vào vị trí position của nút lá s (s và position do phép toán search() trả về)

- Nếu nút lá s chưa đầy: gọi phép toán insnode để chèn khóa k vào nút s

- Nếu nút lá s đã đầy: tách nút lá này thành 2 nút nửa trái và nửa phải

Code :

Hình 4.2.3 c Mã code chi tiết thêm node

*4.2.4 Tách node*

-Tách node đầy (current). current là nút đầy bị tách, sau khi tách xong nút current chỉ còn lại một nửa số khóa bên trái.

-extra\_entry, extra\_branch và position là khóa mới, nhánh cây con và vị trí chèn vào nút current

-Nút right\_half là nút nửa phải có được sau lần tách, nút right\_half chiếm một nửa số khóa bên phải

-Median là khóa ngay chính giữa sẻ được chèn vào nút cha

Code :

void insert (pBNode s, int k, int position)

void insert (pBNode s, int f, int position)

{

pBNode current, right\_half, P;

pBNode extra\_branch;

int pos, extra\_entry, median;

current = s;

extra\_entry = f;

extra\_branch = NULL;

pos = position;

p = father (current);

while (p != NULL && current -> count == Order){

split(current, extra\_entry, extra\_branch, position, right\_half, median);

current = p;

extra\_entry = median;

extra\_branch = right\_half;

pos = nodesearch (p, median);

p = father (current);

}

if(current - > count+1 < Order){

insnode (current, extra\_entry, extra\_branch, pos);

return;

}

split (current, extra\_entry, extra\_branch, pos, right\_half, median);

Root = makeroot (median);

Root -> Branch[0] = current;

Root -> Branch[1] = right\_half;

}

void split (pBNode current, int extra\_entry, pBNode

extra\_branch, int position, pBNode &right\_half, int

&median)

{

pBNode p;

p = new pBNode;

if(position > Order/2)

{

copy(current, Order/2+1, Order – 2, p);

insnode (P , extra\_entry, extra\_branch, position- Order/2 -1);

current->numtrees = Order/2+1;

median = current -> key[Order/2];

right\_half = p ;

return;

}

if (position == Order/2){

copy(current, Order/2, Order-2, p);

current->numtrees = Order/2+1;

current -> Branch[0] = extra\_branch;

median = current -> key[Order/2];

right\_half = p;

return;

}

if (position < Order/2){

copy(current, Order/2, Order-2, p);

current->numtrees = Order/2;

median = current -> key[Order/2- 1];

insnode(current, extra\_entry, extra\_branch, position);

right\_half = p;

return;

}

}

Hình 4.2.4 Mô tả chi tiết tách node

*4.2.5 Thêm vào node lá*

Code :

void insnode (pBNode current, int extra\_entry,

pBNode extra\_branch, int position)

{

int i;

for(i = current->count; i >= position+1; i--){

current -> Branch[i+1] = current -> Branch[i];

current -> key[i] = current -> key[i - 1];

}

current -> key[position] = extra\_entry;

current -> Branch[position + 1] = extra\_branch;

current -> count +=1;

}

Hình 4.2.5 Mô tả chi tiết thêm vào node lá

## 5. Cây đỏ đen

### 5.1. Khái niệm và ví dụ .

là một dạng cây tìm kiếm nhị phân tự cân bằng, một cấu trúc dữ liệu được sử dụng trong khoa học máy tính. Cấu trúc ban đầu của nó được đưa ra vào năm 1972 bởi Rudolf Bayer. Ông gọi chúng là các "B-cây cân bằng" còn tên hiện nay được đưa ra từ 1978 bởi Leo J. Guibas và Robert Sedgewick. Nó là cấu trúc phức tạp nhưng cho kết quả tốt về thời gian trong trường hợp xấu nhất. Các phép toán trên chúng như tìm kiếm (search), chèn (insert), và xóa (delete) trong thời gian O (log n), trong đó n là số các phần tử của cây.

### 5.2. Cấu trúc dữ liệu của cây đỏ đen

* Thông tin lưu trữ tại Node (key).
* Địa chỉ Node gốc của cây con bên trái (\* pLeft).
* Địa chỉ Node gốc của cây con bên phải (\* pRight).
* Địa chỉ của Node cha (\* pParent).
* Thuộc tính màu của Node (color).

Khai báo theo cấu trúc dữ liệu :

/\* khai báo thuộc tính màu cho Node \*/

enum Color {RED, BLACK};

/\* Khai báo cấu trúc Node \*/

struct Node

{

int data;

bool color;

Node \*left, \*right, \*parent;

// Constructor

Node(int data)

{

this->data = data;

left = right = parent = NULL;

this->color = RED;

}

};

### 5.3. Các tính chất và phép toán

Mỗi nút của cây đỏ-đen có thuộc tính "màu" nhận một trong hai giá trị "đỏ" hoặc "đen". Ngoài ra:

1. Một nút hoặc là đỏ hoặc là đen.
2. Gốc là đen.
3. Tất cả các lá (NULL) là đen.
4. Cả hai con của mọi nút đỏ là đen. (và suy ra mọi nút đỏ có nút cha là đen)
5. Tất cả các đường đi từ một nút bất kỳ tới các lá có số nút đen bằng nhau.

5.3.1 Phép chèn

Phép chèn bắt đầu bằng việc bổ sung một nút như trong cây tìm kiếm nhị phân bình thường và gán cho nó màu đỏ

 Nhãn **N** sẽ dùng để chỉ nút đang chèn vào, **P** chỉ nút cha của **N**, **G** chỉ ông của **N**, và **U** chỉ chú bác của **N**. Nhớ rằng , giữa các trường hợp, vai trò và nhãn của các nút có thể thay đổi còn trong cùng một trường hợp thì không.

Mỗi trường hợp được giới thiệu bằng một đoạn mã **C**. Nút chú bác và nút ông dễ dàng xác định nhờ các hàm sau:

struct node \*grandparent(struct node \*n) {

return n->parent->parent;

}

struct node \*uncle(struct node \*n) {

if (n->parent == grandparent(n)->left)

return grandparent(n)->right;

else

return grandparent(n)->left;

}

**Trường hợp 1:** Nút mới thêm **N** ở tại gốc

void insert\_case1(struct node \*n) {

if (n->parent == NULL)

n->color = BLACK;

else

insert\_case2(n);

}

**Trường hợp 2:** Nút cha **P** của nút mới thêm là đen

void insert\_case2(struct node \*n) {

if (n->parent->color == BLACK)

return; /\* Tree is still valid \*/

else

insert\_case3(n);

}

**Trường hợp 3:** Cả cha **P** và bác **U** là đỏ, thì có thể đổi cả hai thành đen còn **G** thành đỏ

void insert\_case3(struct node \*n) {

if (uncle(n)!= NULL && uncle(n)->color == RED) {

n->parent->color = BLACK;

uncle(n)->color = BLACK;

grandparent(n)->color = RED;

insert\_case1(grandparent(n));

}

else

insert\_case4(n);

}

* + 1. Phép xóa

Trong cây tìm kiếm nhị phân bình thường khi xóa một nút có cả hai con (không là *lá null*), ta tìm phần tử lớn nhất trong cây con trái hoặc phần tử nhỏ nhất trong cây con phải, chuyển giá trị của nó vào nút đang muốn xóa .

Chúng ta sẽ sử dụng hàm sau tìm người anh em của *N*

struct node \*sibling(struct node \*n) {

if (n = = n->parent->left)

return n->parent->right;

else

return n->parent->left;

}

 Để thuận tiện các đoạn code trong mục này quy ước rằng các "lá null" được biểu diễn bằng các đối tượng nút thực sự khác biệt một chút với NULL (code trong *phép chèn* có biểu diễn không như vậy).

void delete\_one\_child(struct node \*n) {

/\* Giả thiết: n có ít nhất một nút con null \*/

struct node \*child = is\_leaf(n->right) ? n->left: n->right;

replace\_node(n, child);

if (n->color == BLACK) {

if (child->color == RED)

child->color = BLACK;

else

delete\_case1(child);

}

free(n);

}

**Trường hợp 1:** **N** là gốc mới. Trong trường hợp này chúng ta dừng lại.

void delete\_case1(struct node \*n) {

if (n->parent == NULL)

return;

else

delete\_case2(n);

}

**Trường hợp 2:** **S** là đỏ. Trong trường hợp này tráo đổi màu của **P** và **S**, và sau đó quay trái  tai **P**

void delete\_case2(struct node \*n) {

if (sibling(n)->color == RED) {

n->parent->color = RED;

sibling(n)->color = BLACK;

if (n == n->parent->left)

rotate\_left(n->parent);

else

rotate\_right(n->parent);

}

delete\_case3(n);

}

**Trường hợp 3:** **P**, **S**, và các con của **S** là đen.

void delete\_case3(struct node \*n) {

if (n->parent->color == BLACK &&

sibling(n)->color == BLACK &&

sibling(n)->left->color == BLACK &&

sibling(n)->right->color == BLACK)

{

sibling(n)->color = RED;

delete\_case1(n->parent);

}

else

delete\_case4(n);

}

# KẾT LUẬN

## 

## 1.Đánh giá đề tài

## 1.1.Những công việc đã làm được

- Hiểu thêm được các loại cây nhị phân có từng ví dụ minh họa

- Nâng cao kĩ năng làm việc theo nhóm.

## 1.2.Ưu điểm của hệ thống trong đề tài

- Xây dựng được chương trình “tìm kiếm nhị phân” sử dụng các thuật toán tìm kiếm được các node trên cây nhị phân .

- Nâng cao kĩ năng làm việc theo nhóm giữa các thành viên

- Chương trình xử lí nhanh tương đối chính xác

## 1.3.Khuyết điểm của hệ thống trong đề tài

- Mặc dù rất cố gắng nhưng trong thời gian ngắn , kinh nghiệm còn hạn chế nên kết quả còn nhiều thiếu sót cần tiếp tục được hoàn thiện để có thể giải được các yêu cầu phức tạp hơn.

- Chương trình còn nhiều lỗi như vấn đề xử lí , thiếu sót thông tin mô tả thuật toán chưa được chi tiết ,…….

## 1.4.Hướng phát triển của đề tài

- Xây dựng hoàn thiện các chức năng giúp người sử dụng dễ dàng hơn, các phương pháp duyệt tối ưu, hiệu quả hơn.

Trên đây là kết quả đạt được cũng như còn một số tồn tại, hướng phát triển của đề tài.

Nhóm em xin chân thành cảm ơn sự quan tâm , giúp đỡ, nhiệt tình của thầy Ths.Trần Văn Thọ giúp em hoàn thành đề tài này .