

TRƯỜNG ĐẠI HỌC GIAO THÔNG VẬN TẢI  
PHÂN HIỆU TẠI THÀNH PHỐ HỒ CHÍ MINH



**BÁO CÁO BÀI TẬP LỚN  
CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT**

Giảng viên hướng dẫn:

Trần Thị Dung

Sinh viên:

MSSV:

Võ Đoàn Hoàng Long

6051071067

Trần Huỳnh Lưu

6051071071

Hồ Ngọc Thống

6051071118

Lê Gia Minh

6051071072

Nguyễn Phi Thanh

6051071106

TRƯỜNG ĐẠI HỌC GIAO THÔNG VẬN TẢI  
PHÂN HIỆU TẠI THÀNH PHỐ HỒ CHÍ MINH



**BÁO CÁO BÀI TẬP LỚN  
LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG**

**Giảng viên hướng dẫn:**

**Trần Thị Dung**

**Sinh viên:**

**MSSV:**

Võ Đoàn Hoàng Long

6051071067

Trần Huỳnh Lưu

6051071071

Hồ Ngọc Thống

6051071118

Lê Gia Minh

6051071072

Nguyễn Phi Thanh

6051071106

## **LỜI CẢM ƠN**

Sau gần 3 học kỳ học tập và rèn luyện tại bộ môn Công nghệ thông tin Trường Đại học Giao thông Vận tải – Phân hiệu tại thành phố Hồ Chí Minh chúng em đã được trang bị các kiến thức cơ bản và nâng cao, cùng với các kỹ năng thực tế để có thể hoàn thành bài tập lớn của mình.

Sau một thời gian nỗ lực thực hiện thì đề tài cũng đã hoàn thành. Nhưng không sao tránh khỏi những sai sót do chúng em còn chưa có nhiều kinh nghiệm thực tế. Em kính mong nhận được sự góp ý và nhận xét từ quý thầy, cô để em có thể hoàn thiện và hoàn thành tốt hơn cho đề tài của mình.

Lời sau cùng em một lần nữa kính chúc quý thầy, cô bộ môn Công nghệ thông tin Trường Đại học Giao thông Vận tải - Phân hiệu tại thành phố Hồ Chí Minh thật nhiều sức khỏe và thành công.

*Tp. Hồ Chí Minh, ngày 25 tháng 11 năm 2020*

Sinh viên thực hiện

**Võ Đoàn Hoàng Long**

**Trần Huỳnh Lưu**

**Hồ Ngọc Thống**

**Lê Gia Minh**

**Nguyễn Phi Thanh**

## MỤC LỤC

<b>Chương 1. Cơ sở lý thuyết .....</b>	<b>1</b>
<b>    1. Lists, Stacks, Queues .....</b>	<b>2</b>
<b>        1.1 Kiểu dữ liệu trừu tượng (ADTs).....</b>	<b>1</b>
<b>        1.2 The List ADT .....</b>	<b>1</b>
<b>            1.2.1 Triển khai mảng danh sách .....</b>	<b>1</b>
<b>            1.2.2 Danh Sách Liên Kết Đơn.....</b>	<b>2</b>
<b>        1.3 Vector và List trong STL.....</b>	<b>2</b>
<b>            1.3.1 Iterators .....</b>	<b>3</b>
<b>            1.3.2 Ví dụ về tính năng xóa trên danh sách .....</b>	<b>2</b>
<b>            1.3.3 Const_iterators .....</b>	<b>5</b>
<b>        1.4 Triển khai vector .....</b>	<b>6</b>
<b>        1.5 Triển khai danh sách .....</b>	<b>7</b>
<b>        1.6 Ngăn xếp Stack .....</b>	<b>13</b>
<b>            1.6.1 Mô hình ngăn xếp .....</b>	<b>13</b>
<b>            1.6.2 Khai triển ngăn xếp Stack.....</b>	<b>14</b>
<b>            1.6.3 Các ứng dụng .....</b>	<b>14</b>
<b>        1.7 Hàng đợi ADT .....</b>	<b>18</b>
<b>    Tóm tắt.....</b>	<b>23</b>
<b>    2. Cây .....</b>	<b>23</b>
<b>        2.1 Khái quát cây .....</b>	<b>24</b>
<b>            2.1.1 Thực hiện cây .....</b>	<b>24</b>
<b>            2.1.2 Traversal cây với một số ứng dụng.....</b>	<b>24</b>

<b>2.2 Cây nhị phân .....</b>	<b>25</b>
<b>2.2.1 Thực hiện .....</b>	<b>25</b>
<b>2.2.2 Cây biểu thức .....</b>	<b>25</b>
<b>2.3 Cây tìm kiếm ADT - Cây tìm kiếm nhị phân .....</b>	<b>29</b>
<b>2.3.1 Chứa đựng .....</b>	<b>30</b>
<b>2.3.2 Tìm Min và Max .....</b>	<b>31</b>
<b>2.3.3 Chèn .....</b>	<b>32</b>
<b>2.3.4 Xóa .....</b>	<b>35</b>
<b>2.3.5 Hàm hủy và sao chép .....</b>	<b>37</b>
<b>2.3.6 Phân tích trường hợp trung bình .....</b>	<b>37</b>
<b>2.4 Cây AVL .....</b>	<b>41</b>
<b>2.4.1 Xoay đơn .....</b>	<b>44</b>
<b>2.4.2 Xoay kép .....</b>	<b>47</b>
<b>2.5 Phát họa cây .....</b>	<b>56</b>
<b>2.5.1 Một ý tưởng đơn giản .....</b>	<b>57</b>
<b>2.5.2 Phân bố .....</b>	<b>59</b>
<b>2.6 Duyệt cây.....</b>	<b>66</b>
<b>2.7 B - cây .....</b>	<b>68</b>
<b>2.8 Sets và Maps trong thư viện Chuẩn .....</b>	<b>75</b>
<b>2.8.1 Sets.....</b>	<b>75</b>
<b>2.8.2 Maps .....</b>	<b>76</b>
<b>2.8.3 Thực hiện của sets và maps .....</b>	<b>78</b>
<b>2.8.4 Ví dụ về sử dụng một số Maps.....</b>	<b>79</b>
<b>Tóm lại .....</b>	<b>85</b>

<b>3. Hàng đợi ưu tiên .....</b>	<b>87</b>
<b>3.1 Mô hình .....</b>	<b>87</b>
<b>3.2 Triển khai đơn giản .....</b>	<b>88</b>
<b>3.3 Đống nhị phân .....</b>	<b>89</b>
<b>3.3.1 Thuộc tính cấu trúc .....</b>	<b>89</b>
<b>3.3.2 Thuộc tính thứ tự đống .....</b>	<b>92</b>
<b>3.3.3 Hoạt động đống cơ bản .....</b>	<b>92</b>
<b>3.3.4 Hoạt động Heap khác .....</b>	<b>97</b>
<b>3.4 Các ứng dụng của hàng đợi ưu tiên .....</b>	<b>102</b>
<b>3.4.1 Các vấn đề lựa chọn .....</b>	<b>102</b>
<b>3.4.2 Mô phỏng sự kiện .....</b>	<b>104</b>
<b>3.5 d-Heaps .....</b>	<b>106</b>
<b>3.6 Hàng đống cánh tả .....</b>	<b>107</b>
<b>3.6.1 Thuộc tính đống cánh tả .....</b>	<b>107</b>
<b>3.6.2 Hoạt động của đống cánh tả .....</b>	<b>109</b>
<b>3.7 Đống xiên .....</b>	<b>116</b>
<b>3.8 Hàng đợi nhị thức .....</b>	<b>118</b>
<b>3.8.1 Cấu trúc hàng đợi nhị thức .....</b>	<b>118</b>
<b>3.8.2 Hoạt động hàng đợi nhị thức .....</b>	<b>119</b>
<b>3.8.3 Triển khai hàng đợi nhị thức .....</b>	<b>124</b>
<b>3.9 Hàng đợi Ưu tiên trong thư viện Chuẩn .....</b>	<b>132</b>
<b>Tóm lại .....</b>	<b>132</b>

<b>Chương 2. Nội dung đề tài .....</b>	<b>133</b>
1. Hàng đợi ưu tiên .....	133
2. Ý tưởng làm đề tài.....	133
3. Hướng dẫn cài đặt và sử dụng .....	134
<b>Chương 3. Kết luận .....</b>	<b>137</b>
1. Kết quả đạt được .....	137
2. Nhược điểm .....	137
3. Hướng phát triển .....	137
4. Công việc của mỗi thành viên và mức độ hoàn thành .....	137
<b>Tài liệu tham khảo.....</b>	<b>138</b>

# Chương 1. Cơ sở lý thuyết

## 1. Lists, Stacks, Queues

Ở phần này chúng ta đề cập đến 3 trong số các cấu trúc dữ liệu cơ bản, đơn giản nhất mà có thể chúng ta sẽ gặp nhiều trong các chương trình.

- Giới thiệu khái niệm về các kiểu dữ liệu trừu tượng (ADTs).
- Chỉ ra cách để tối ưu các thao tác trên danh sách.
- Giới thiệu cấu trúc dữ liệu ngăn xếp và công dụng của nó trong việc triển khai đệ quy.
- Giới thiệu cấu trúc dữ liệu hàng đợi và việc sử dụng nó trong hệ điều hành và trong thiết kế thuật toán.

### 1.1 Kiểu dữ liệu trừu tượng (ADTs)

Một kiểu dữ liệu trừu tượng là một mô hình toán học cùng với một tập hợp các phép toán (operator) trừu tượng được định nghĩa trên mô hình đó. Ví dụ tập hợp số nguyên cùng với các phép toán hợp, giao, hiệu là một kiểu dữ liệu trừu tượng.

### 1.2 The List ADT

Giả sử chúng ta có một danh sách có dạng tổng quát như sau: A0, A1, A2, A3....AN-1, kích thước của danh sách này là N và một danh sách rỗng sẽ có kích thước là 0.

Đối với bất kỳ danh sách nào ngoại trừ danh sách rỗng, Ai luôn theo sau Ai-1 ( $i < N$ ) và Ai-1 đứng trước Ai ( $i > 0$ ). Phần tử đầu tiên của danh sách là A0, phần tử cuối cùng là AN-1. Vị trí của phần tử Ai trong danh sách là i.

Một số các thao tác phổ biến thực hiện trên List ADT là: printList (xuất danh sách), makeEmpty (làm rỗng), find (trả về vị trí xuất hiện đầu tiên của đối tượng cần tìm), insert và remove (chèn, xóa tại vị trí bất kỳ) và findKth trả về phần tử ở một số vị trí (được chỉ định làm đối số).

#### 1.2.1 Triển khai mảng danh sách

Trước đây để quản lý một danh sách ta cần sử dụng mảng để quản lý. Vấn đề nghiêm trọng ở đây đó chính là ta cần phải ước tính về kích thước tối đa của mảng. Đối với **printList** thì việc triển khai mảng được thực hiện theo thời gian tuyến tính và **findKth** thì mất thời gian không đổi. Tuy nhiên việc chèn, xóa có thể mất nhiều thời gian và tốn kém. Khi chèn hay xóa một phần tử ta

có thể phải dịch toàn bộ phần tử của mảng về một vị trí để thực hiện thao tác.

### 1.2.2 Danh Sách Liên Kết Đơn

Để khắc phục những nhược điểm của mảng thì danh sách liên kết được sử dụng đến. Trong danh sách liên kết các phần tử không gần kề nhau mà chúng được liên kết với nhau bằng liên kết đến phần tử kế nhiệm trước đó.

Mỗi phần tử sẽ là một Node bao gồm Data (dữ liệu của node đó) và Next (con trỏ để liên kết các phần tử với nhau).

Thời gian xuất danh sách cũng như tìm kiếm tương tự như mảng. Tuy nhiên chúng ta không còn lo lắng về vấn đề kích thước lưu trữ. Xóa hay chèn phần tử vào danh sách giờ đây ta chỉ cần thay đổi con trỏ (Next) của node bất kì chứ không cần phải dịch hay tác động đến toàn bộ các phần tử trong danh sách như mảng.

## 1.3 Vector và List trong STL

C++ bao gồm các thư viện chuẩn (STL) của nó và việc triển khai các cấu trúc dữ liệu chung.

Danh sách ADT là một trong những cấu trúc dữ liệu thực hiện trong STL. Lợi thế của việc sử dụng vector là nó có thể lập chỉ mục trong thời gian không đổi. Bất lợi khi sử dụng vector là ở việc chèn và xóa phần tử.

Vector và danh sách đều không hiệu quả trong việc tìm kiếm. Danh sách thì không dễ lập chỉ mục. Tuy nhiên chúng đều có những phương pháp chung:

Int size() : trả về số phần tử có trong vùng chứa.

Void clear() : loại bỏ tất cả các phần tử ra khỏi vùng chứa.

Bool blank() : trả về True nếu vùng chứa rỗng và False nếu ngược lại.

Void push\_back() : thêm một phần tử vào cuối danh sách.

Void pop\_back() : xóa phần tử ở cuối danh sách.

Const Object &back() : trả về đối tượng cuối danh sách.

Const Object &front() : trả về đối tượng đầu danh sách.

Tuy nhiên một danh sách được liên kết kép cho phép các thay đổi hiệu quả ở phía trước, nhưng một vectơ thì không, hai phương pháp sau chỉ có sẵn cho danh sách:

`void push_front (const Object & x):` thêm x vào trước danh sách.

`void pop_front ():` loại bỏ đối tượng ở đầu danh sách.

### 1.3.1 Iterators

Iterator là một con trỏ được sử dụng để đại diện cho các biến được trả đến để thực hiện các thao tác thêm, sửa, xóa, ...

Cung cấp cách để truy xuất các thành phần của một đối tượng hợp nhất một cách tuần tự mà không cho thấy đại diện bên dưới của nó.

Danh sách kiểu `<string>::iterator`.

Danh sách kiểu vector `<int>::iterator`.

Có 3 vấn đề chính:

- Làm thế nào để có một trình lặp.
- Bản thân các trình vòng lặp có thể thực hiện những hoạt động nào.
- Liệt kê các phương pháp ADT.

#### Getting an Iterator

- `begin()` trả về một iterator đại diện cho vị trí của phần tử đầu tiên trong container.
- `end()` trả về một iterator đại diện cho vị trí đứng ngay sau phần tử cuối cùng trong container.

#### Iterator Methods

- Operator \* cereference và trả về giá trị bên trong container tại vị trí mà iterator được đặt.
- Operator ++ di chuyển iterator đến phần tử tiếp theo trong container.
- Operator -- ngược lại so với operator ++.
- Operator == và operator != dùng để so sánh vị trí tương đối của 2 phần tử đang được trả đến bởi 2 iterator.

- Operator = dùng để gán vị trí mà iterator trả đến.

### **Container Operations That Require Iterator**

- **Iterator insert (iterator pos, const Object & x)**: thêm x vào trước vị trí được cung cấp bởi vị trí trình lặp.
- **Iterator erase (iterator pos)**: xóa đối tượng tại vị trí được cung cấp trước.
- **Iterator erase (iterator start, iterator end)**: xóa các đối tượng từ vị trí bắt đầu đến vị trí kết thúc đã chọn.

#### **1.3.2 Ví dụ về tính năng xóa trên danh sách**

Xóa khỏi vùng chứa danh sách một phần tử (vị trí) hoặc một loạt các phần tử ([đầu tiên, cuối cùng]).

template <typename Container>

```
void removeEveryOtherItem( Container & lst )
{
    auto itr = lst.begin( );      // itr is a Container::iterator
    while( itr != lst.end( ) )
    {
        itr = lst.erase( itr );
        if( itr != lst.end( ) )
            ++itr;
    }
}
```

### **1.3.3 Const\_iterators**

Quy trình sau đây hoạt động cho cả vectơ và danh sách và chạy trong thời gian tuyến tính.

```
template <typename Container, typename Object>

void change (Container & c, const Object & newValue)

{

    typename Container :: iterator itr = c.begin ();

    while (itr != c.end ())

        * itr ++ = newValue;

}

void print( const list<int> & lst, ostream & out = cout )

{

    typename Container::iterator itr = lst.begin( );

    while( itr != lst.end( ) )

    {

        out << *itr << endl;

        *itr = 0; // This is fishy!!!

        ++itr;

    }

}
```

Các giải pháp được cung cấp bởi STL là mọi bộ sưu tập không chỉ chứa một trình lặp lồng nhau nhưng cũng là một kiểu lồng nhau `const_iterator`. Sự khác biệt chính giữa trình lặp và `const_iterator` là toán tử `*` đối với `const_iterator` trả về một tham chiếu không đổi, và do đó `*itr` cho một `const_iterator` không thể xuất hiện ở bên trái của câu lệnh gán.

Hơn nữa, trình biên dịch sẽ buộc bạn sử dụng `const_iterator` để duyệt qua một hằng số bộ sưu tập. Nó làm như vậy bằng cách cung cấp hai phiên bản bắt đầu và hai phiên bản kết thúc, như sau:

- `iterator begin( )`
- `const_iterator begin( ) const`
- `iterator end( )`
- `const_iterator end( ) const`

#### 1.4 Triển khai vector

- Mảng chỉ đơn giản là một biến con trỏ trỏ tới một khối bộ nhớ, kích thước mảng thực tế phải được duy trì bởi lập trình viên.
- Khối bộ nhớ có thể được cấp phát thông qua `new []` nhưng sau đó phải được giải phóng thông qua `delete []`.
- Không thể thay đổi kích thước khối bộ nhớ (nhưng một khối mới, có lẽ lớn hơn có thể lấy và khởi tạo với khối cũ, sau đó khối cũ có thể được giải phóng).

Để tránh sự mơ hồ với lớp thư viện, chúng ta sẽ đặt tên cho lớp mẫu là `Vector`.

Trước khi kiểm tra mã `Vector`, chúng ta phác thảo các chi tiết chính:

1. `Vector` sẽ duy trì mảng nguyên thủy (through qua một biến con trỏ tới khối bộ nhớ được cấp phát), dung lượng của mảng và số lượng mục hiện tại được lưu trữ trong `Vector`.
2. `Vector` sẽ triển khai Big-Five để cung cấp sao chép sâu ngữ nghĩa cho bản sao hàm tạo và toán tử `=`, và sẽ cung cấp hàm hủy để lấy lại mảng nguyên thủy.

3. Vector sẽ cung cấp một quy trình thay đổi kích thước sẽ thay đổi kích thước của Vector và quy trình `reserve` (dự trữ) sẽ thay đổi sức chứa của Vector. Dung lượng được thay đổi bằng cách lấy một khối mới bộ nhớ cho mảng nguyên thủy, sao chép khối cũ vào khối mới và cải tạo khối cũ.
4. Vector sẽ cung cấp một triển khai toán tử `[]`.
5. Vector sẽ cung cấp các quy trình cơ bản, chẳng hạn như `size`, `empty`, `clear`, `back`, `pop_back` và `push_back`. Quy trình `push_back` sẽ gọi `reserve` nếu kích thước và sức chứa là như nhau.
6. Vector sẽ cung cấp hỗ trợ cho các trình lặp kiểu lồng nhau và `const_iterator`, và phương thức bắt đầu và kết thúc được liên kết.

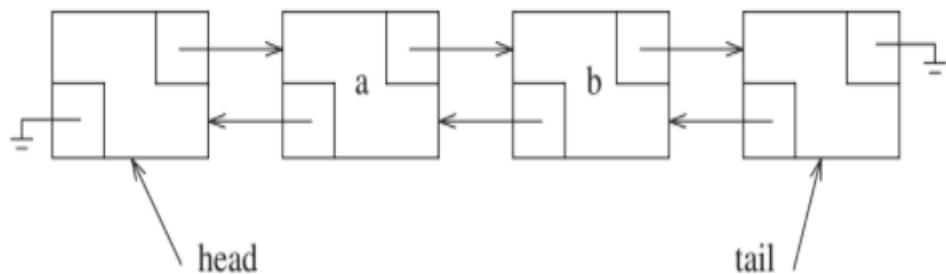
## 1.5. Triển khai danh sách

Khi xem xét thiết kế, chúng ta sẽ cần cung cấp bốn lớp:

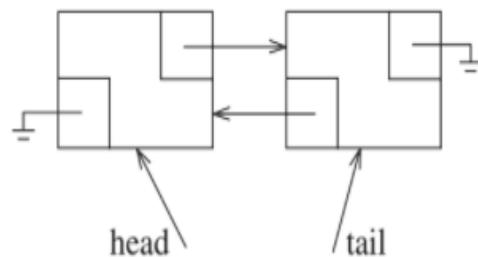
1. Bản thân lớp `List`, chứa các liên kết đến cả hai đầu, kích thước của danh sách và một loạt các phương thức.
2. Lớp `Node`, có thể là một lớp lồng nhau riêng tư. Một node chứa dữ liệu và con trỏ đến các node trước đó và tiếp theo, cùng với các hàm tạo thích hợp.
3. Lớp `const_iterator`, trừu tượng hóa khái niệm vị trí và là một lớp lồng nhau công khai. `Const_iterator` lưu trữ một con trỏ tới node "hiện tại" và cung cấp việc triển khai các hoạt động của trình lặp cơ bản, tất cả đều ở dạng toán tử được nạp chồng chẳng hạn như `=`, `==`, `!=`, và `++`.
4. Lớp trình lặp, tóm tắt khái niệm về một vị trí và là một lớp lồng nhau công khai. Trình vòng lặp có cùng chức năng với `const_iterator`, ngoại trừ toán tử `*` trả về một tham chiếu đến mục đang được xét, thay vì tham chiếu hằng số đến mục. Một vấn đề kỹ thuật quan trọng là một trình lặp có thể được sử dụng trong bất kỳ quy trình nào yêu cầu trình điều khiển `const_iterator`, nhưng không phải ngược lại. Nói cách khác, trình lặp IS-A `const_iterator`.

Bởi vì các lớp trình vòng lặp lưu trữ một con trỏ đến “node hiện tại” và điểm đánh dấu kết thúc là một vị trí hợp lệ, nên việc tạo một nút bổ sung ở cuối danh sách để đại diện cho điểm đánh dấu là rất hợp lý. Hơn nữa, chúng ta có thể tạo thêm một node ở đầu danh sách, đại diện một cách hợp lý cho điểm đánh dấu đầu. Những nút phụ này đôi khi được gọi là **sentinel nodes**, cụ thể, node ở phía trước đôi khi được gọi là **header node** (node đầu) và node ở cuối đôi khi được gọi là **tail node** (node đuôi).

Ưu điểm của việc sử dụng các node bổ sung này là chúng đơn giản hóa quá trình mã hóa bằng cách loại bỏ một loạt các trường hợp đặc biệt.



Hình 1.5.1 cho thấy danh sách được liên kết kép với các node đầu và đuôi.



Hình 1.5.2 cho thấy một danh sách trống.

```
1 template <typename Object>
2 class List
3 {
4     private:
5         struct Node
6             { /* See Figure 3.13 */ };
7
8     public:
9         class const_iterator
10            { /* See Figure 3.14 */ };
11
12        class iterator : public const_iterator
13            { /* See Figure 3.15 */ };
14
15    public:
16        List( )
17            { /* See Figure 3.16 */ }
18        List( const List & rhs )
19            { /* See Figure 3.16 */ }
20        ~List( )
21            { /* See Figure 3.16 */ }
22        List & operator= ( const List & rhs )
23            { /* See Figure 3.16 */ }
24        List( List && rhs )
```

Hình 1.5.3 Liệt kê lớp (Phần 1/4)

```
25     { /* See Figure 3.16 */ }
26     List & operator= ( List && rhs )
27     { /* See Figure 3.16 */ }
28
29     iterator begin( )
30     { return { head->next }; }
31     const_iterator begin( ) const
32     { return { head->next }; }
33     iterator end( )
34     { return { tail }; }
35     const_iterator end( ) const
36     { return { tail }; }
37
38     int size( ) const
39     { return theSize; }
40     bool empty( ) const
41     { return size( ) == 0; }
42
43     void clear( )
44     {
45         while( !empty( ) )
46             pop_front( );
47     }
```

Hình 1.5.4 Liệt kê lớp (Phần 2/4)

```

48     Object & front( )
49         { return *begin( ); }
50     const Object & front( ) const
51         { return *begin( ); }
52     Object & back( )
53         { return *--end( ); }
54     const Object & back( ) const
55         { return *--end( ); }
56     void push_front( const Object & x )
57         { insert( begin( ), x ); }
58     void push_front( Object && x )
59         { insert( begin( ), std::move( x ) ); }
60     void push_back( const Object & x )
61         { insert( end( ), x ); }
62     void push_back( Object && x )
63         { insert( end( ), std::move( x ) ); }
64     void pop_front( )
65         { erase( begin( ) ); }
66     void pop_back( )
67         { erase( --end( ) ); }
68

```

Hình 1.5.5 Liệt kê lớp (Phần 3/4)

```

69     iterator insert( iterator itr, const Object & x )
70         { /* See Figure 3.18 */ }
71     iterator insert( iterator itr, Object && x )
72         { /* See Figure 3.18 */ }
73
74     iterator erase( iterator itr )
75         { /* See Figure 3.20 */ }
76     iterator erase( iterator from, iterator to )
77         { /* See Figure 3.20 */ }
78
79     private:
80         int theSize;
81         Node *head;
82         Node *tail;
83
84     void init( )
85         { /* See Figure 3.16 */ }
86 };

```

Hình 1.5.6 Liệt kê lớp (Phần 3/4)

Vì không muốn thêm dữ liệu mới cũng như không có ý định thay đổi cách thức của phương thức hiện tại.

Tại dòng 28, 29, const\_iterator được lưu trữ dưới dạng thành viên dữ liệu đơn lẽ và con trỏ trả tới nút hiện tại. Thông thường điều này là riêng tư, trình lặp sẽ không có quyền truy cập.

Tuy nhiên để kế thừa và đồng thời không cho các lớp khác có quyền truy cập thì tại dòng 34 35 chúng ta thấy hàm tạo cho const\_iterator được sử dụng trong việc triển khai lớp Danh sách của begin và end. Giải pháp đưa ra là chúng ta sẽ khai báo một hàm bạn ở dòng 37 để cấp quyền truy cập lớp Danh sách cho các thành viên không công khai của const\_iterator.

Các phương thức nạp chồng toán tử :

- Operator ==
- Operator !=
- Operator ++
- Operator --
- Operator >
- Operator <

```
1     struct Node
2     {
3         Object data;
4         Node *prev;
5         Node *next;
6
7         Node( const Object & d = Object{ }, Node * p = nullptr,
8               Node * n = nullptr )
9             : data{ d }, prev{ p }, next{ n } { }
10
11        Node( Object && d, Node * p = nullptr, Node * n = nullptr )
12            : data{ std::move( d ) }, prev{ p }, next{ n } { }
13    };
```

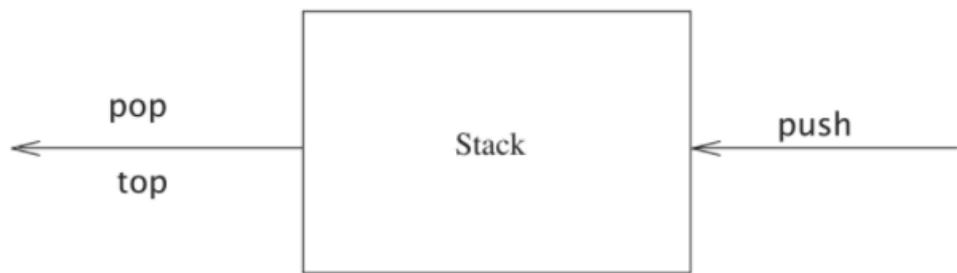
Hình 1.5.7 Lớp Node chồng nhau cho lớp Danh sách

## 1.6 Ngăn xếp Stack

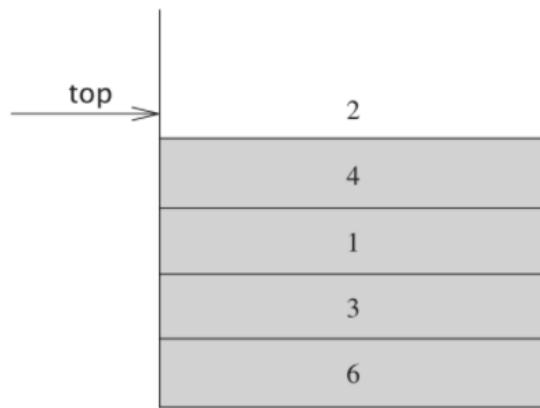
### 1.6.1 Mô Hình Ngăn Xếp

Ngăn xếp được gọi là danh sách LIFO (vào sau ra trước), các phần tử được thêm vào sau sẽ được kiểm tra trước.

Một số thao tác cơ bản của ngăn xếp như pop (lấy ra), push (thêm vào), insert (chèn), erase (xóa)...giống như danh sách thông thường.



Hình 1.6.1 Mô hình ngăn xếp Đầu vào cho ngăn xếp là bằng đầu ra đầy là bằng cửa sổ bật lên và trên cùng.



Hình 1.6.2 Mô hình ngăn xếp Chỉ có thể truy cập phần tử trên cùng

### **1.6.2 Khai triển ngăn xếp Stack**

Có 2 cách triển khai ngăn xếp phổ biến:

- Sử dụng cấu trúc để liên kết
- Sử dụng một m

#### **Triển khai danh sách liên kết của ngăn xếp**

Lần triển khai đầu tiên của ngăn xếp sử dụng một danh sách được liên kết đơn lẻ. Thực hiện việc thêm một phần tử vào đầu danh sách, lấy ra một phần tử ở cuối danh sách.

#### **Triển khai mảng của ngăn xếp**

Sử dụng back(), pop\_back(), push\_back() giống như vector để triển khai.

### **1.6.3 Các ứng dụng**

Sau đây là 3 trong số những ứng dụng quan trọng của ngăn xếp, trong đó ứng dụng thứ 3 sẽ cung cấp cho chúng ta cái nhìn sâu sắc về các chương trình được tổ chức.

#### **Cân bằng các biểu tượng**

Trình biên dịch kiểm tra các chương trình của bạn để tìm lỗi cú pháp, nhưng thường xuyên thiếu một ký hiệu (chẳng hạn như thiếu dấu ngoặc nhọn hoặc dấu khởi đầu chú thích) có thể khiến trình biên dịch đưa ra hàng trăm dòng chẩn đoán mà không xác định được lỗi thực sự.

Một công cụ hữu ích trong tình huống này là một chương trình kiểm tra xem mọi thứ có cân bằng hay không. Do đó, mọi dấu ngoặc nhọn, dấu ngoặc vuông và dấu ngoặc đơn phải tương ứng với phần đối vị bên trái của nó.

#### **Biểu thức hậu tố**

Giả sử chúng ta có một máy tính bỏ túi và muốn tính chi phí của một chuyến đi mua sắm. Để làm như vậy, chúng ta thêm một danh sách các số và nhân

kết quả với 1,06, điều này sẽ tính giá mua của một số mặt hàng đã thêm thuế bán hàng địa phương. Nếu các mặt hàng là 4,99, 5,99 và 6,99, thì cách nhập tự nhiên sẽ là dãy số  $4,99 + 5,99 + 6,99 * 1,06 = ?$

Hầu hết các máy tính bốn hàm đơn giản sẽ đưa ra câu trả lời đầu tiên, nhưng nhiều máy tính tiên tiến biết rằng phép nhân có mức độ ưu tiên cao hơn phép cộng. Mặt khác, một số mặt hàng chịu thuế và một số mặt hàng thì không, vì vậy nếu chỉ có mặt hàng đầu tiên và mặt hàng cuối cùng thực sự chịu thuế, thì trình tự  $4,99 * 1,06 + 5,99 + 6,99 * 1,06 =$  sẽ cho câu trả lời đúng (18,69) trên máy tính khoa học và câu trả lời sai (19,37) trên máy tính đơn giản.

Một chuỗi đánh giá điển hình cho ví dụ này có thể là nhân 4,99 và 1,06, lưu câu trả lời này dưới dạng A1. Sau đó, chúng tôi thêm 5,99 và A1, lưu kết quả trong A1. Chúng tôi nhân 6,99 và 1,06, lưu câu trả lời trong A2 và kết thúc bằng cách thêm A1 và A2, để lại câu trả lời cuối cùng câu trả lời trong A1. Chúng ta có thể viết chuỗi hoạt động này như sau:

4,99 1,06 \* 5,99 + 6,99 1,06 \*+

Ký hiệu này được gọi là hậu tố, hoặc ký hiệu Ba Lan đảo ngược

Cách dễ nhất để làm điều này là sử dụng ngăn xếp. Khi một số được thấy, nó được đẩy lên ngăn xếp, khi một toán tử được nhìn thấy, toán tử được áp dụng cho hai số (kí hiệu).

### ***Chuyển đổi biểu thức trung tố sang hậu tố***

Không chỉ có thể sử dụng ngăn xếp để đánh giá biểu thức hậu tố mà chúng ta còn có thể sử dụng ngăn xếp để chuyển đổi một biểu thức ở dạng trung tố (hay còn gọi là infix) thành hậu tố.

Giả sử chúng ta muốn chuyển đổi biểu thức infix  $a + b * c + (d * e + f) * g$  thành postfix. Một câu trả lời đúng là  $abc * + de * f + g * +$ . Khi một toán hạng được đọc, nó ngay lập tức được đặt vào đầu ra. Các toán tử không được xuất ra ngay lập tức, vì vậy chúng phải được lưu ở đâu đó. Việc cần làm là đặt các toán tử có được nhìn thấy, nhưng không được đặt trên đầu ra, vào ngăn xếp.

Chúng ta cũng sẽ xếp chồng các dấu ngoặc đơn bên trái khi chúng gặp phải. Chúng ta bắt đầu với một ngăn xếp trống ban đầu.

Nếu chúng ta thấy một dấu ngoặc phải, thì chúng ta bật ngăn xếp, viết các ký hiệu cho đến khi chúng ta gặp một dấu ngoặc trái (tương ứng), được xuất hiện nhưng không xuất hiện. Nếu chúng ta thấy bất kỳ biểu tượng nào khác (+, \*, (), thì chúng ta bật mục từ ngăn xếp cho đến khi chúng tôi tìm thấy mục nhập có mức độ ưu tiên thấp hơn. Ý tưởng của thuật toán này là khi một toán tử được nhìn thấy, nó sẽ được đặt trên ngăn xếp. Ngăn xếp đại diện cho các toán tử đang chờ xử lý.

Ví dụ minh họa:

a + b \* c + ( d \* e + f ) \* g

Để xem thuật toán này hoạt động như thế nào, chúng tôi sẽ chuyển đổi biểu thức infix dài ở trên thành dạng hậu tố của nó. Đầu tiên, ký hiệu **a** được đọc, vì vậy nó được chuyển đến đầu ra. Sau đó **+** được đọc và đẩy lên ngăn xếp, **b** tiếp theo được đọc và chuyển đến đầu ra.



Tiếp theo, dấu **\*** được đọc. Mục trên cùng trên ngăn xếp toán tử có mức độ ưu tiên thấp hơn **\***, vì vậy không có gì là đầu ra và **\*** được đưa vào ngăn xếp. Tiếp theo, **c** là đọc và xuất. Đến lúc này, chúng ta có :



Biểu tượng tiếp theo là dấu +. Kiểm tra ngăn xếp, chúng ta thấy rằng chúng ta sẽ bật dấu \* và đặt nó trên đầu ra, bật dấu + khác, không có mức độ ưu tiên thấp hơn nhưng bằng nhau, trên ngăn xếp và sau đó đẩy dấu +.



Ký hiệu tiếp theo được đọc là **a** (. Được ưu tiên cao nhất, ký hiệu này được đặt trên ngăn xếp. Sau đó **d** là đọc và xuất.



Chúng ta tiếp tục bằng cách đọc dấu \*. Vì dấu ngoặc đơn mở không bị xóa trừ khi đóng ngoặc đang được xử lý, không có đầu ra. Tiếp theo, **e** được đọc và xuất.



Ký hiệu tiếp theo được đọc là dấu +. Chúng tôi bật và xuất \* rồi nhấn +. Sau đó, chúng tôi đọc và đầu ra **f**.



Bây giờ chúng ta đọc **a**), vì vậy ngăn xếp được làm trống trở lại dấu (. Chúng ta xuất ra a +.



Chúng ta đọc dấu \* tiếp theo; nó được đẩy vào ngăn xếp, sau đó **g** được đọc và xuất ra.



Đầu vào bây giờ trống, vì vậy chúng tôi bật và xuất các ký hiệu từ ngăn xếp cho đến khi nó trống.



## 1.7 Hàng đợi ADT

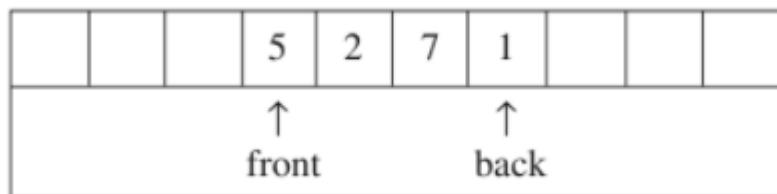
Giống như hàng đợi ngăn xếp là danh sách. Với một hàng đợi, tuy nhiên, việc chèn được thực hiện ở một đầu trong khi việc xóa được thực hiện ở đầu kia.

### Mô hình hàng đợi

Các hoạt động cơ bản trên hàng đợi là enqueue chèn một phần tử vào cuối danh sách (được gọi là phía sau) và dequeue xóa (và trả về) phần tử ở đầu danh sách (được gọi là phía trước). Hình 1.7.1 cho thấy mô hình trừu tượng của một hàng đợi.

## Triển khai hàng đợi theo mảng

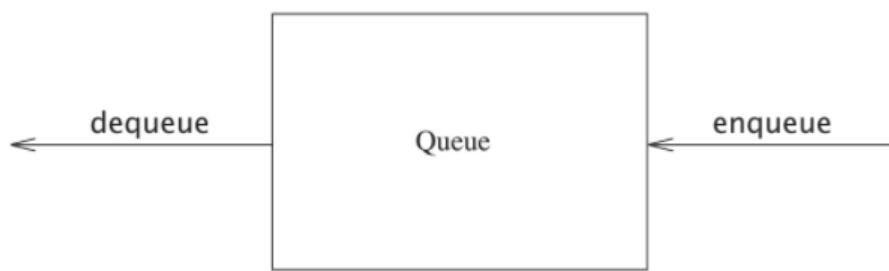
Như với ngăn xếp, bất kỳ triển khai danh sách nào cũng hợp pháp đối với hàng đợi. Giống như ngăn xếp, cả danh sách liên kết và triển khai mảng đều cung cấp thời gian chạy O (1) nhanh cho mọi hoạt động. Việc triển khai danh sách liên kết là đơn giản và được để lại như một bài tập. Nay giờ chúng ta sẽ thảo luận về việc triển khai mảng các hàng đợi. Đối với mỗi cấu trúc dữ liệu hàng đợi, chúng tôi giữ một mảng theArray và các vị trí phía trước và phía sau đại diện cho các phần cuối của hàng đợi. Chúng tôi cũng theo dõi số lượng phần tử thực sự có trong hàng đợi currentSize. Bảng sau đây cho thấy một hàng đợi ở một số trạng thái trung gian.



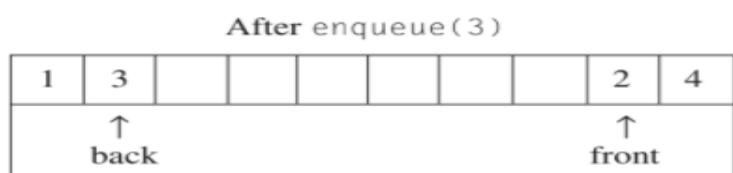
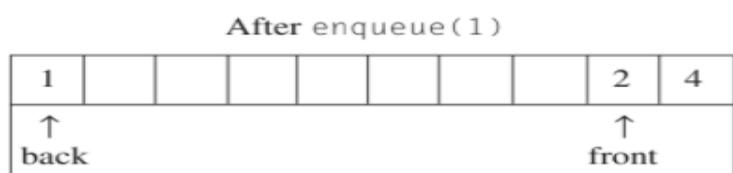
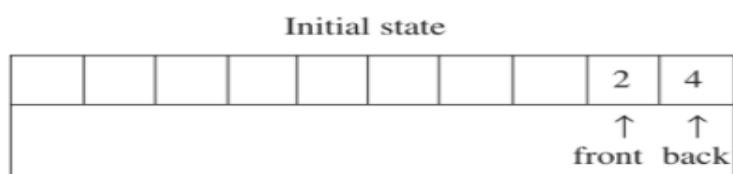
Các hoạt động phải rõ ràng. Để xếp hàng một phần tử x, chúng ta tăng currentSize và quay lại sau đó đặt Array[back]=x. Để dequeue một phần tử, chúng tôi đặt giá trị trả về cho currentSize Array[front] giảm và sau đó tăng lên phía trước. Các chiến lược khác là có thể (điều này sẽ được thảo luận sau). Chúng tôi sẽ bình luận về việc kiểm tra các lỗi hiện tại.

Có một vấn đề tiềm ẩn với việc triển khai này. Sau 10 hàng đợi, hàng đợi dường như đã đầy vì trở lại bây giờ là chỉ mục mảng cuối cùng và hàng đợi tiếp theo sẽ ở vị trí không tồn tại. Tuy nhiên, có thể chỉ có một vài phần tử trong hàng đợi vì một số phần tử có thể đã được xếp thứ tự. Hàng đợi như ngăn xếp thường nhỏ ngay cả khi có nhiều thao tác.

Giải pháp đơn giản là bất cứ khi nào phía trước hoặc phía sau đến cuối mảng, nó sẽ được bao quanh từ đầu. Các bảng sau đây hiển thị hàng đợi trong một số hoạt động. Đây được gọi là triển khai mảng tròn.



Hình 1.7.1 Mô hình hàng đợi



After dequeue, which returns 2

1	3							2	4
↑ back					↑ front				

After dequeue, which returns 4

1	3							2	4
↑ front					↑ back				

After dequeue, which returns 1

1	3							2	4
↑ back					↑ front				

Mã bổ sung cần thiết để thực hiện bao bọc là tối thiểu (mặc dù nó có thể tăng gấp đôi thời gian chạy). Nếu việc tăng lên phía sau hoặc phía trước khiến nó đi qua mảng, giá trị sẽ được đặt lại về vị trí đầu tiên trong mảng.

Một số lập trình viên sử dụng các cách khác nhau để biểu diễn mặt trước và mặt sau của hàng đợi. Ví dụ, một số không sử dụng một mục nhập để theo dõi kích thước vì chúng dựa vào trường hợp cơ sở mà khi hàng đợi trống trở lại front-1. Kích thước được tính ngầm bằng cách so sánh mặt sau và mặt trước. Đây là một cách rất phức tạp vì có một số trường hợp đặc biệt, vì vậy hãy cẩn thận nếu bạn cần sửa đổi mã được viết theo cách này. Nếu currentSize không được duy trì như một thành viên dữ liệu rõ ràng thì hàng đợi sẽ đầy khi có phần tử theArray.capacity () – 1, chỉ khi theArray.capacity() có thể phân biệt các kích thước khác nhau và một trong số này là 0. Chọn bất kỳ kiểu nào bạn thích và đảm bảo rằng tất cả các quy trình của bạn đều nhất quán. Vì có một số tùy chọn để triển khai, nên có thể đáng giá một hoặc hai nhận xét trong mã nếu bạn không sử dụng thành viên dữ liệu currentSize.

Trong các ứng dụng mà bạn chắc chắn rằng số lượng hàng đợi không lớn hơn dung lượng của hàng đợi, việc bao bọc là không cần thiết. Như với ngăn xếp, hàng đợi hiếm khi được thực hiện trừ khi các quy trình gọi chắc chắn rằng hàng đợi không trống. Do đó, kiểm tra lỗi thường bị bỏ qua cho hoạt động này ngoại trừ trong mã quan trọng. Điều này thường không chính đáng vì tiết kiệm thời gian mà bạn có thể đạt được là rất ít.

## **Ứng dụng của hàng đợi**

Có nhiều thuật toán sử dụng hàng đợi để cung cấp thời gian chạy hiệu quả. Bây giờ chúng ta sẽ đưa ra một số ví dụ đơn giản về cách sử dụng hàng đợi.

Khi công việc được nộp cho một máy in, chúng được sắp xếp theo thứ tự đến. Vì vậy, về cơ bản các công việc được gửi đến máy in được đặt trên một hàng đợi.

Hầu như mọi dòng trong cuộc sống thực đều (được cho là) là một hàng đợi. Ví dụ, các hàng tại quầy vé là hàng đợi vì dịch vụ đến trước được phục vụ trước.

Một ví dụ khác liên quan đến mạng máy tính. Có nhiều thiết lập mạng của máy tính cá nhân trong đó đã được gắn vào một máy được gọi là máy chủ tệp. Người dùng trên các máy khác được cấp quyền truy cập vào tệp trên cơ sở ai đến trước được phục vụ trước nên cấu trúc dữ liệu là một hàng đợi.

Các ví dụ khác bao gồm các:

- Cuộc gọi đến các công ty lớn sau đây thường được xếp vào hàng đợi khi tất cả các nhà khai thác đang bận.
- Trong các trường đại học lớn, nơi mà nguồn lực hạn chế, sinh viên phải ký vào danh sách chờ nếu tất cả các máy tính đều bị chiếm dụng. Học sinh sử dụng máy tính lâu nhất sẽ bị buộc tắt máy trước tiên và học sinh chờ đợi lâu nhất là người dùng tiếp theo được phép bật.

Toàn bộ một nhánh của toán học được gọi là lý thuyết xếp hàng đề cập đến việc tính toán theo xác suất thời gian mà người dùng mong đợi sẽ đợi trên một dòng bao lâu thì dòng đó và những câu hỏi khác như vậy. Câu trả lời phụ thuộc vào tần suất người dùng đến hàng và mất bao lâu để xử lý một người dùng sau khi người dùng được phục vụ. Cả hai tham số này được cho dưới dạng hàm phân phối xác suất. Trong những trường hợp đơn giản, một câu trả lời có thể được tính toán một cách phân tích. Ví dụ về một trường hợp đơn giản sẽ là một đường dây điện thoại có một nhà điều hành. Nếu nhà điều hành bận, người gọi sẽ được xếp vào hàng chờ (lên đến một số giới hạn tối đa). Vấn đề này quan trọng đối với các doanh nghiệp vì các nghiên cứu đã chỉ ra rằng mọi người rất nhanh chóng cúp máy.

Nếu có k toán tử thì vấn đề này khó giải hơn nhiều. Các vấn đề khó giải quyết về mặt phân tích thường được giải quyết bằng mô phỏng. Trong trường hợp của chúng tôi, chúng tôi sẽ cần sử dụng một hàng đợi để thực hiện mô phỏng. Nếu k lớn, chúng ta cũng cần các cấu trúc dữ liệu khác để thực hiện điều này một cách hiệu quả. Chúng ta sẽ xem cách thực hiện mô phỏng này trong Chương D. Sau đó, chúng ta có thể chạy mô phỏng cho một số giá trị của k và chọn k tối thiểu để có thời gian chờ hợp lý.

Các ứng dụng bổ sung cho hàng đợi rất nhiều và cũng như với các ngăn xếp, điều đáng kinh ngạc là một cấu trúc dữ liệu đơn giản lại có thể quan trọng đến vậy.

### **Tóm tắt**

Chương này mô tả khái niệm ADT và minh họa khái niệm này bằng ba trong số các kiểu dữ liệu trừu tượng phổ biến nhất. Mục tiêu chính là tách việc triển khai ADT ra khỏi chức năng của chúng. Chương trình phải biết các hoạt động làm gì nhưng thực sự tốt hơn là không biết nó được thực hiện như thế nào.

Ngăn xếp, danh sách và hàng đợi có lẽ là ba cấu trúc dữ liệu cơ bản trong tất cả khoa học máy tính và việc sử dụng chúng được ghi lại thông qua một loạt các ví dụ. Đặc biệt, chúng ta đã thấy cách các ngăn xếp được sử dụng để theo dõi các lệnh gọi hàm và cách đệ quy thực sự được triển khai. Điều quan trọng cần hiểu là không chỉ vì nó làm cho các lan theo thủ tục trở nên khả thi mà bởi vì biết cách thực hiện đệ quy sẽ loại bỏ rất nhiều bí ẩn xung quanh việc sử dụng nó.

## **2. Cây**

Ở phần này, chúng ta sẽ tìm hiểu về cấu trúc dữ liệu đơn giản mà thời gian chạy trung bình hầu hết là  $O(\log N)$ . Chúng tôi đã phác thảo một số trường hợp đơn giản về mặt khái niệm của cấu trúc dữ liệu này để đề phòng trường hợp xấu nhất và thảo luận về phương thức thứ 2 - về cơ bản cung cấp thời gian chạy  $O(\log N)$  cho mỗi hoạt động trong 1 chuỗi dài của hướng dẫn. Cấu trúc dữ liệu chúng ta đang nói tới là cây nhị phân tìm kiếm. Thuật toán cây nhị phân tìm kiếm là nền tảng cho việc triển khai 2 lớp thư viện, **set** (thiết lập) và **map** (bản đồ), cái mà được sử dụng nhiều trong các ứng dụng.

Trong phần này, chúng ta sẽ:

- Hiểu được cách mà cây được sử dụng để triển khai các hoạt động phổ biến của hệ thống
- Hiểu được cách cây đánh giá các biểu thức số học.
- Cách sử dụng cây để hỗ trợ các hoạt động tìm kiếm trong thời gian trung bình  $O(\log N)$  và làm thế nào để cẩn chỉnh các những ý tưởng này để có được giới hạn trong trường hợp xấu nhất  $O(\log N)$ . Chúng tôi cũng sẽ xem cách để thực hiện các hoạt động này khi dữ liệu được lưu trữ trên đĩa.
- Thảo luận và sử dụng các lớp **set** và **map**.

## 2.1 Khái quát cây

Một cây có thể được định nghĩa theo nhiều cách. Một cách tự nhiên để xác định cây là đệ quy. Một cây gồm có nhiều nút. Gốc của mỗi cây con được cho là con của r và r là cây cha của mỗi cây con. Từ định nghĩa đệ quy, chúng tôi thấy rằng một cây là tập hợp của N nút, một trong số đó có cái là gốc, và N-1 nút con.

### 2.1.1 Thực hiện cây

Một cách để triển khai cây là có trong mỗi nút, bên cạnh dữ liệu của nó, một liên kết đến từng con của nút. Tuy nhiên, vì số lượng nút con trên mỗi nút có thể khác nhau rất nhiều và không được biết trước, nên có thể không thực hiện được việc tạo liên kết trực tiếp các nút con trong dữ liệu.

### 2.1.2 Traversal cây với một số ứng dụng

Có rất nhiều ứng dụng cho cây. Một trong những cách sử dụng phổ biến là cấu trúc thư mục trong nhiều hệ điều hành phổ biến, bao gồm cả UNIX và DOS. Thư mục gốc của thư mục này là /usr. (Dấu hoa thị bên cạnh tên cho biết rằng /usr chính nó là một thư mục.) /usr có ba con là mark, alex và bill, đó là chúng- thư mục bản thân. Do đó, /usr chứa ba thư mục và không có tệp thông thường. Tên tệp /usr/mark/book/ch1.r có được bằng cách theo dõi con ngoài cùng bên trái ba lần. Mỗi / sau đầu tiên chỉ ra một cạnh; kết quả là tên đường dẫn đầy đủ. Hệ thống tệp phân cấp này rất phổ biến vì nó cho phép người dùng tổ chức dữ liệu của họ một cách hợp lý hơn nữa, hai tệp trong các thư mục khác nhau có thể chia sẻ cùng một tên, vì chúng phải có các đường dẫn khác nhau từ gốc và do đó có các tên đường dẫn khác

nhau. Thư mục trong hệ thống tệp UNIX chỉ là một tệp với danh sách tất cả các phần tử con của nó, do đó, các thư mục được cấu trúc gần như chính xác.

## 2.2 Cây nhị phân

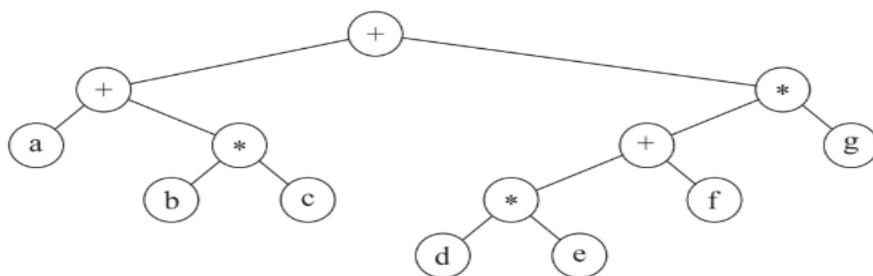
Cây nhị phân là cây trong đó không nút nào có thể có nhiều hơn hai nút con.

### 2.2.1 Thực hiện

Bởi vì một nút cây nhị phân có nhiều nhất là hai nút con, chúng ta có thể giữ các liên kết trực tiếp đến chúng. Các khai báo của các nút cây có cấu trúc tương tự như đối với danh sách được liên kết kép, trong đó một nút là một cấu trúc bao gồm thông tin phần tử cộng với hai con trỏ (trái và phải) đến các nút khác. Chúng tôi có thể vẽ các cây nhị phân bằng cách sử dụng các hộp hình chữ nhật thường dùng cho danh sách được liên kết, nhưng cây thường được vẽ dưới dạng vòng tròn được kết nối bằng các đường, vì chúng thực sự là đồ thị. Chúng tôi cũng không vẽ rõ ràng các liên kết nullptr khi đề cập đến cây, bởi vì mọi cây nhị phân với  $N$  nút sẽ yêu cầu  $N + 1$  liên kết nullptr. Cây nhị phân có nhiều công dụng quan trọng không gắn với việc tìm kiếm. Một trong những sử dụng chính của cây nhị phân là trong lĩnh vực thiết kế trình biên dịch, mà bây giờ chúng ta sẽ khám phá.

### 2.2.2 Cây biểu thức

Hình 2.1 cho thấy một ví dụ về cây biểu thức. Lá của cây biểu thức là toán hạng, chẳng hạn như hằng số hoặc tên biến và các nút khác chứa toán tử. Cây cụ thể này xảy ra là nhị phân, vì tất cả các toán tử đều là nhị phân, và mặc dù đây là trường hợp đơn giản nhất, các nút có thể có nhiều hơn hai con. Nó cũng là có thể đối với một nút chỉ có một nút con, như trường



Hình 2.1 Cây biểu hiện cho cho  $(a + (b * c)) + (((d * e) + f) * g)$ .

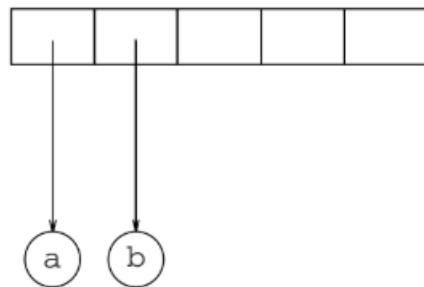
hợp của toán tử trừ một bậc. Chúng ta có thể đánh giá một cây biểu thức, T, bằng cách áp dụng toán tử ở gốc cho các giá trị thu được bằng cách đánh giá đệ quy các cây con trái và phải. Trong ví dụ của chúng tôi, bên trái cây con đánh giá là  $a + (b * c)$  và cây con bên phải đánh giá là  $((d * e) + f) * g$ . Các do đó toàn bộ cây đại diện cho  $(a + (b * c)) + (((d * e) + f) * g)$ . Chúng ta có thể tạo biểu thức infix (được đặt trong ngoặc đơn quá nhiều) bằng cách tạo đệ quy biểu thức trái dấu ngoặc đơn, sau đó in ra toán tử ở gốc, và cuối cùng là lặp lại- tạo ra một biểu thức bên phải trong ngoặc đơn. Chiến lược chung này (trái, nút, phải) được biết đến như là một người đi ngang hàng không; nó dễ nhớ vì kiểu diễn đạt nó sản xuất. Một chiến lược duyệt thay thế là in ra đệ quy cây con bên trái, cây con bên phải cây, và sau đó là toán tử. Nếu chúng ta áp dụng chiến lược này cho cây của chúng ta ở trên, đầu ra là  $abc * + de * f + g * +$ , có thể dễ dàng nhìn thấy là đại diện hậu tố của Phần 3.6.3. Chiến lược truyền tải này thường được biết đến như một chiến lược trình tự sau. Chúng tôi đã thấy điều này chiến lược đi ngang trước đó trong Phần 4.1. Chiến lược duyệt thứ ba là in toán tử trước rồi in đệ quy ra cây con bên trái và bên phải. Biểu thức kết quả,  $++ a * b c * + * defg$ , là ký hiệu tiền tố ít hữu ích hơn và chiến lược truyền tải là phương thức truyền tải có thứ tự trước, chúng ta cũng đã thấy trước đó trong phần 2.1

### **Cấu trúc cây biểu thức**

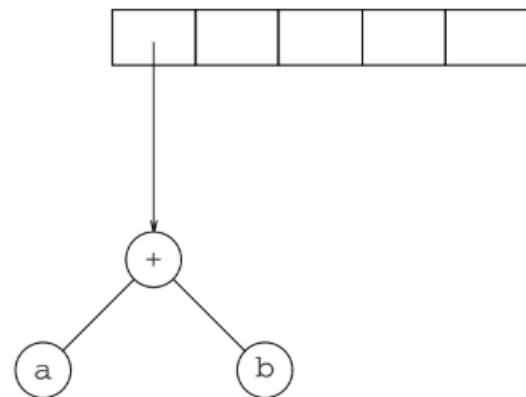
Bây giờ chúng ta đưa ra một thuật toán để chuyển đổi một biểu thức hậu tố thành một cây biểu thức. Vì chúng tôi đã có một thuật toán để chuyển đổi infix thành postfix, chúng tôi có thể tạo cây biểu thức từ hai loại đầu vào phổ biến. Phương pháp chúng tôi mô tả rất giống với postfix thuật toán đánh giá của Mục 3.6.3. Chúng tôi đọc biểu tượng của chúng tôi từng biểu tượng một. Nếu biểu tượng là một toán hạng, chúng tôi tạo một cây một nút và đẩy một con trỏ đến nó vào một ngăn xếp. Nếu biểu tượng là một toán tử, chúng tôi bật (con trỏ) đến hai cây T1 và T2 từ ngăn xếp (T1 được xuất hiện đầu tiên) và tạo thành một cây mới có gốc là toán tử và có bên trái và bên phải trẻ em lần lượt chỉ vào T2 và T1. Một con trỏ đến cây mới này sau đó được đẩy lên ngăn xếp. Ví dụ, giả sử đầu vào là:

$a b + c d e + * *$

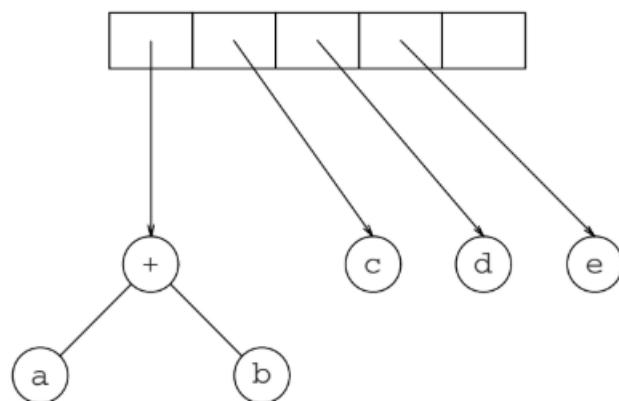
Hai biểu tượng đầu tiên là các toán hạng, vì vậy chúng tôi tạo cây một nút và đẩy con trỏ đến chúng thành một ngăn xếp.



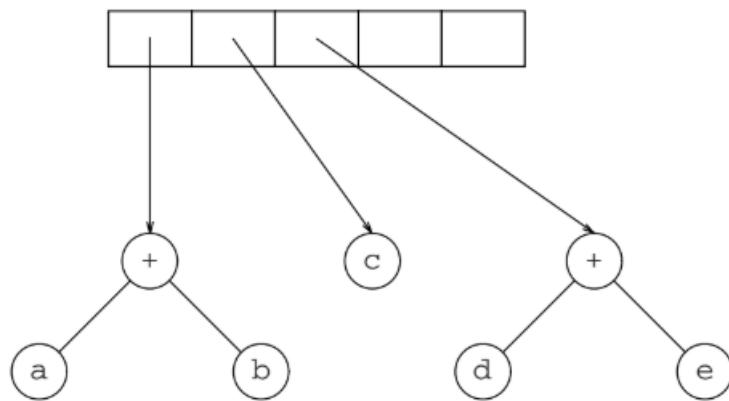
Tiếp theo, dấu + được đọc, vì vậy hai con trỏ đến cây được xuất hiện, một cây mới được hình thành và một con trỏ để nó được đẩy lên ngăn xếp.



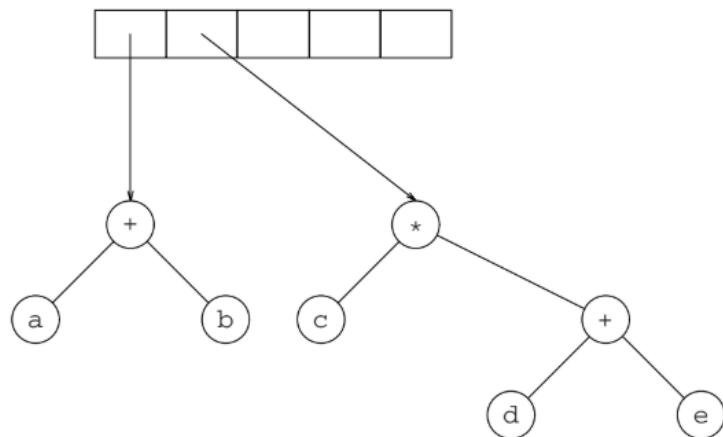
Tiếp theo, c, d và e được đọc và đối với mỗi cây một nút được tạo và một con trỏ đến cây tương ứng được đẩy lên ngăn xếp.



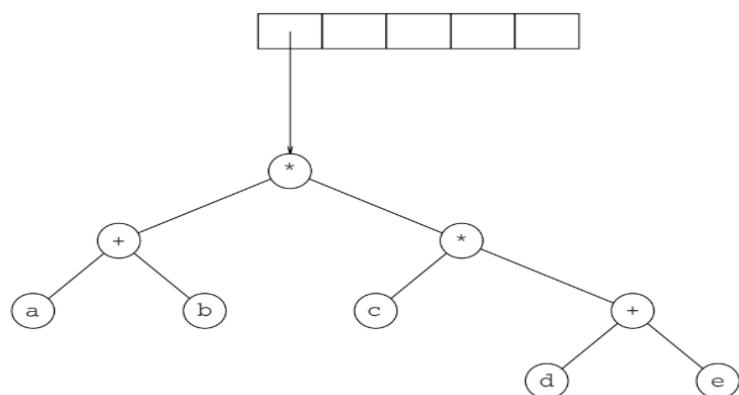
Bây giờ dấu + được đọc, vì vậy hai cây được hợp nhất.



Tiếp tục, dấu \* được đọc, vì vậy chúng ta bật hai con trỏ cây và tạo một cây mới với dấu \* làm gốc.



Cuối cùng, ký hiệu cuối cùng được đọc, hai cây được hợp nhất và một con trỏ đến cây cuối cùng còn lại trên ngăn xếp.



### 2.3 Cây tìm kiếm ADT - Cây tìm kiếm nhị phân.

Một ứng dụng quan trọng của cây nhị phân là việc sử dụng chúng trong việc tìm kiếm. Chúng ta hãy giả định rằng mỗi nút trong cây lưu trữ một mục. Trong các ví dụ của chúng tôi, chúng tôi sẽ giả định, vì đơn giản, những là các số nguyên, mặc dù các mục phức tạp tùy ý có thể dễ dàng xử lý trong C++. Chúng tôi cũng sẽ giả sử rằng tất cả các mục đều khác biệt và chúng tôi sẽ xử lý các mục trùng lặp sau.

Thuộc tính biến cây nhị phân thành cây tìm kiếm nhị phân là đối với mọi nút X, trong cây, giá trị của tất cả các mục trong cây con bên trái của nó nhỏ hơn mục trong X, và giá trị của tất cả các mục trong cây con bên phải của nó lớn hơn mục trong X. Lưu ý rằng điều này ngụ ý rằng tất cả các phần tử trong cây có thể được sắp xếp theo một số cách nhất quán. Trong Hình 2.2, cây bên trái là cây tìm kiếm nhị phân, nhưng cây bên phải thì không. Các cây bên phải có nút có mục 7 trong cây con bên trái của nút có mục 6 (mà xảy ra là gốc). Bây giờ chúng tôi đưa ra các mô tả ngắn gọn về các hoạt động thường được thực hiện trên hệ nhị phân cây tìm kiếm. Lưu ý rằng vì định nghĩa đệ quy của cây, người ta thường viết các quy trình này một cách đệ quy. Bởi vì độ sâu trung bình của cây tìm kiếm nhị phân hóa ra là  $O(\log N)$ , chúng ta thường không cần phải lo lắng về việc hết không gian ngăn xếp.



Hình 2.2 Minh họa về thành viên công cộng gọi hàm thành viên đệ quy riêng tư

Một số hàm thành viên riêng sử dụng kỹ thuật truyền một biến con trỏ sử dụng cuộc gọi theo tham chiếu. Điều này cho phép các hàm thành viên công cộng chuyển một con trỏ đến root tới các hàm thành viên đệ quy riêng. Sau đó, các hàm đệ quy có thể thay đổi giá trị của gốc để gốc trỏ đến nút khác. Chúng tôi sẽ mô tả kỹ thuật chi tiết hơn khi chúng tôi kiểm tra mã để chèn. Bây giờ chúng ta có thể mô tả một số phương pháp riêng tư.

### 2.3.1 Chứa đựng

Thao tác này yêu cầu trả về true nếu có một nút trong cây T có mục X hoặc false nếu không có nút như vậy. Cấu trúc của cây làm cho điều này trở nên đơn giản. Nếu T trống, thì chúng ta chỉ có thể trả về false. Ngược lại, nếu mục được lưu trữ tại T là X, chúng ta có thể trả về true. Nếu không thì, chúng ta thực hiện một cuộc gọi đệ quy trên cây con T, trái hoặc phải, tùy thuộc vào quan hệ- vận chuyển của X đến mặt hàng được lưu trữ trong T. Mã trong hình 2.3 là sự triển khai của điều này chiến lược.

```
1  /**
2   * Internal method to test if an item is in a subtree.
3   * x is item to search for.
4   * t is the node that roots the subtree.
5   */
6  bool contains( const Comparable & x, BinaryNode *t ) const
7  {
8      if( t == nullptr )
9          return false;
10     else if( x < t->element )
11         return contains( x, t->left );
12     else if( t->element < x )
13         return contains( x, t->right );
14     else
15         return true;    // Match
16 }
```

Hình 2.3 Chứa hoạt động cho cây tìm kiếm nhị phân

Lưu ý thứ tự của các bài kiểm tra. Điều quan trọng là việc kiểm tra cây trống phải được thực hiện trước tiên, vì nếu không, chúng tôi sẽ tạo ra lỗi thời gian chạy khi cố gắng truy cập một thành viên dữ liệu thông qua con trỏ nullptr. Các bài kiểm tra còn lại được sắp xếp với trường hợp ít có khả năng xảy ra

nhất sau cùng. Cũng lưu ý rằng cả hai cuộc gọi đệ quy thực sự là đệ quy đuôi và có thể dễ dàng loại bỏ bằng một vòng lặp while. Việc sử dụng đệ quy đuôi là hợp lý ở đây vì tính chất mô phỏng của biểu thức thuật toán bù cho sự giảm tốc độ và lượng không gian ngăn xếp được sử dụng dự kiến chỉ là  $O(\log N)$ .

### 2.3.2 Tìm Min và Max

Các quy trình riêng tư này trả về một con trỏ tới nút chứa nút nhỏ nhất và lớn nhất các phần tử trong cây, tương ứng. Để thực hiện findMin, hãy bắt đầu từ gốc và đi sang trái bao lâu như có một đứa trẻ bên trái. Điểm dừng là phần tử nhỏ nhất. Quy trình findMax là giống nhau, ngoại trừ việc phân nhánh là đến đúng con. Nhiều lập trình viên không bận tâm đến việc sử dụng đệ quy. Chúng tôi sẽ viết mã quy trình cho cả hai bằng cách thực hiện findMin đệ quy và findMax không đệ quy (xem Hình 2.6 và 2.7). Lưu ý cách chúng tôi xử lý cẩn thận trường hợp thoái hóa của cây trống. Mặc dù đây là luôn luôn quan trọng, nó đặc biệt quan trọng trong các chương trình đệ quy. Cũng lưu ý rằng nó là an toàn để thay đổi t trong findMax, vì chúng tôi chỉ làm việc với bản sao của một con trỏ. Luôn luôn cực kỳ cẩn thận, tuy nhiên, bởi vì một câu lệnh như t-> right = t-> right-> right sẽ thực hiện các thay đổi.

```
1 template <typename Object, typename Comparator=less<Object>>
2 class BinarySearchTree
3 {
4     public:
5
6         // Same methods, with Object replacing Comparable
7
8     private:
9
10    BinaryNode *root;
11    Comparator isLessThan;
12
13    // Same methods, with Object replacing Comparable
14
15    /**
16     * Internal method to test if an item is in a subtree.
```

Hình 2.4 Sử dụng một đối tượng hàm để triển khai cây tìm kiếm nhị phân (1/2)

```

17     * x is item to search for.
18     * t is the node that roots the subtree.
19     */
20     bool contains( const Object & x, BinaryNode *t ) const
21     {
22         if( t == nullptr )
23             return false;
24         else if( isLessThan( x, t->element ) )
25             return contains( x, t->left );
26         else if( isLessThan( t->element, x ) )
27             return contains( x, t->right );
28         else
29             return true;    // Match
30     }
31 };

```

Hình 2.5 Sử dụng một đối tượng hàm để triển khai cây tìm kiếm nhị phân (2/2)

### 2.3.3 Chèn

Quy trình chèn rất đơn giản về mặt khái niệm. Để chèn X vào cây T, hãy tiến hành xuống cây như bạn làm với một chứa. Nếu X được tìm thấy, không làm gì cả. Nếu không, hãy chèn X vào vị trí cuối cùng trên con đường đã đi qua. Hình 2.8 cho thấy điều gì xảy ra. Để chèn 5, chúng tôi đi ngang qua cây như mặc dù một chứa đã xảy ra. Tại nút có mục 4, chúng ta cần phải đi sang phải, nhưng ở đó không phải là cây con, vì vậy 5 không có trong cây, và đây là vị trí chính xác để đặt 5. Các bản sao có thể được xử lý bằng cách giữ một trường bổ sung trong bản ghi nút cho biết tần suất xuất hiện. Điều này tạo thêm một số không gian cho toàn bộ cây nhưng tốt hơn là đặt các bản sao trong cây (có xu hướng làm cho cây rất sâu). Tất nhiên, chiến lược này không hoạt

```

1  /**
2   * Internal method to find the smallest item in a subtree t.
3   * Return node containing the smallest item.
4   */
5   BinaryNode * findMin( BinaryNode *t ) const
6   {
7       if( t == nullptr )
8           return nullptr;
9       if( t->left == nullptr )
10          return t;
11       return findMin( t->left );
12   }

```

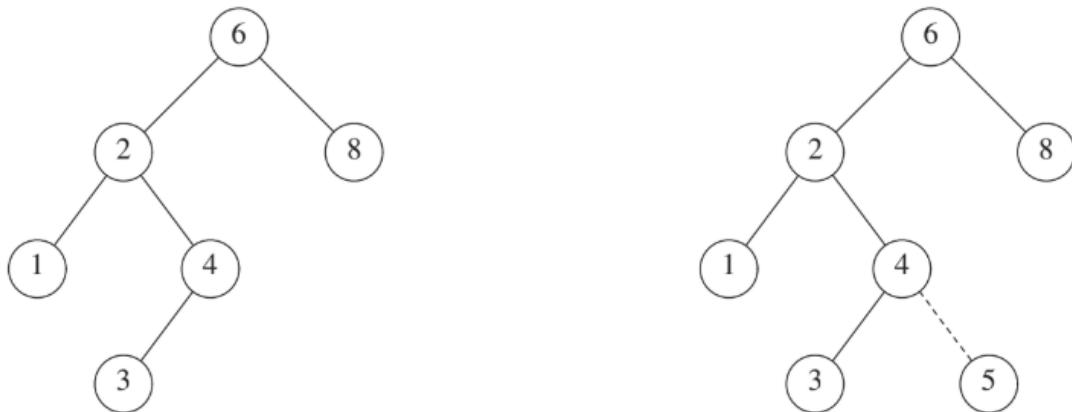
Hình 2.6 Triển khai đệ quy findMin cho cây tìm kiếm nhị phân

```

1  /**
2   * Internal method to find the largest item in a subtree t.
3   * Return node containing the largest item.
4   */
5  BinaryNode * findMax( BinaryNode *t ) const
6  {
7      if( t != nullptr )
8          while( t->right != nullptr )
9              t = t->right;
10     return t;
11 }

```

Hình 2.7 Triển khai không đệ quy của tìm Max cho cây tìm kiếm nhị phân



Hình 2.8 Cây tìm kiếm nhị phân trước và sau khi chèn 5

động nếu khóa hướng dẫn toán tử `<` chỉ là một phần của kết cấu. Nếu đúng như vậy, chúng ta có thể giữ lại tất cả các cấu trúc có cùng một khóa trong cấu trúc dữ liệu phụ trợ, chẳng hạn như danh sách hoặc cây tìm kiếm khác. Hình 2.9 cho thấy mã cho quy trình chèn. Dòng 12 và 14 chèn đệ quy và gắn x vào cây con thích hợp. Lưu ý rằng trong quy trình đệ quy, lần duy nhất `t` thay đổi là khi một lá mới được tạo ra. Khi điều này xảy ra, điều đó có nghĩa là đệ quy quy trình đã được gọi từ một số nút khác, `p`, là nút cha của lá. Cuộc gọi sẽ là `insert (x, p-> left)` hoặc `insert (x, p-> right)`. Dù bằng cách nào, `t` bây giờ là một tham chiếu đến `p-> left` hoặc `p-> right`, nghĩa là `p-> left`

hoặc p->right sẽ được thay đổi để trỏ đến cái mới nút. Nói chung, đây là một thao tác mượt mà

```
1  /**
2   * Internal method to insert into a subtree.
3   * x is the item to insert.
4   * t is the node that roots the subtree.
5   * Set the new root of the subtree.
6   */
7 void insert( const Comparable & x, BinaryNode * & t )
8 {
9     if( t == nullptr )
10        t = new BinaryNode{ x, nullptr, nullptr };
11    else if( x < t->element )
12        insert( x, t->left );
13    else if( t->element < x )
14        insert( x, t->right );
15    else
16        ; // Duplicate; do nothing
17 }
18
19 /**
20  * Internal method to insert into a subtree.
21  * x is the item to insert by moving.
22  * t is the node that roots the subtree.
23  * Set the new root of the subtree.
24  */
25 void insert( Comparable && x, BinaryNode * & t )
26 {
27     if( t == nullptr )
28        t = new BinaryNode{ std::move( x ), nullptr, nullptr };
29     else if( x < t->element )
30        insert( std::move( x ), t->left );
31     else if( t->element < x )
32        insert( std::move( x ), t->right );
33     else
34        ; // Duplicate; do nothing
35 }
```

Hình 2.9 Chèn vào cây tìm kiếm nhị phân

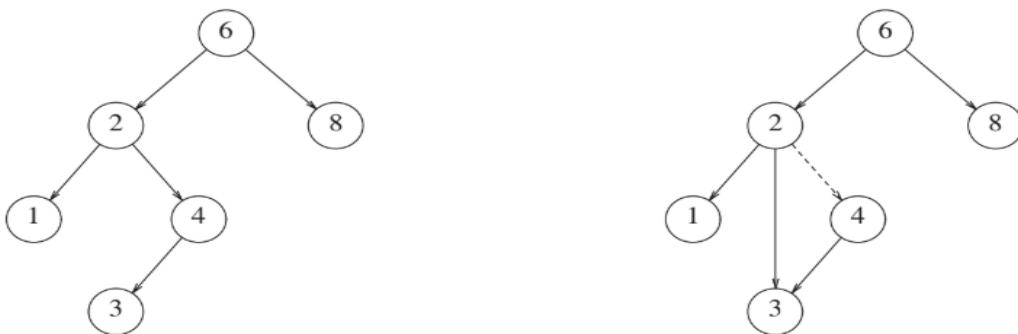
### 2.3.4 Xóa

Như thường thấy với nhiều cấu trúc dữ liệu, thao tác khó nhất là xóa. Nếu nút là một lá, nó có thể bị xóa ngay lập tức. Nếu nút có một nút con, nút có thể bị xóa sau khi cha của nó điều chỉnh một liên kết để bỏ qua nút.

Trường hợp phức tạp xử lý một nút có hai nút con. Chiến lược chung là thay thế dữ liệu của nút này bằng dữ liệu nhỏ nhất của cây con bên phải (dễ dàng tìm thấy) và xóa một cách đệ quy nút đó (hiện đang trống). Vì nút nhỏ nhất trong cây con bên phải không thể có con bên trái, loại bỏ thứ hai là một điều dễ dàng. Hình 2.11 hiển thị một cây ban đầu và kết quả của việc xóa. Nút bị xóa là nút con bên trái của gốc; giá trị khóa là 2. Nó được thay thế bằng dữ liệu nhỏ nhất trong cây con bên phải (3), và thì nút đó bị xóa như trước.

Đoạn mã trong Hình 2.12 thực hiện xóa. Nó không hiệu quả vì nó thực hiện hai lần xuống cây để tìm và xóa nút nhỏ nhất trong cây con bên phải khi điều này là sơn lót. Có thể dễ dàng loại bỏ sự kém hiệu quả này bằng cách viết một phương thức removeMin đặc biệt và chúng tôi đã để lại nó chỉ vì sự đơn giản.

Nếu số lần xóa dự kiến là nhỏ, thì một chiến lược phổ biến để sử dụng là xóa lười: Khi một phần tử bị xóa, nó sẽ được để lại trong cây và chỉ được đánh dấu khi bị xóa. Điều này đặc biệt phổ biến nếu có các mục trùng lặp, vì khi đó thành viên dữ liệu giữ số lượng tần suất xuất hiện có thể được giảm bớt. Nếu số nút thực trong cây bằng với số nút "đã xóa", sau đó độ sâu của cây chỉ được mong đợi tăng lên bởi một hằng số nhỏ (tại sao?), vì vậy có rất hình phạt thời gian nhỏ liên quan đến việc xóa lười biếng. Ngoài ra, nếu một mục đã xóa được lắp lại, chi phí phân bổ một ô mới được tránh.



Hình 2.10 Xóa một nút (4) với một nút con, trước và sau



Hình 2.11 Xóa một nút (2) có hai nút con, trước và sau

```

1  /**
2   * Internal method to remove from a subtree.
3   * x is the item to remove.
4   * t is the node that roots the subtree.
5   * Set the new root of the subtree.
6   */
7 void remove( const Comparable & x, BinaryNode * & t )
8 {
9     if( t == nullptr )
10        return; // Item not found; do nothing
11     if( x < t->element )
12         remove( x, t->left );
13     else if( t->element < x )
14         remove( x, t->right );
15     else if( t->left != nullptr && t->right != nullptr ) // Two children
16     {
17         t->element = findMin( t->right )->element;
18         remove( t->element, t->right );
19     }
20     else
21     {
22         BinaryNode *oldNode = t;
23         t = ( t->left != nullptr ) ? t->left : t->right;
24         delete oldNode;
25     }
26 }
```

Hình 2.12 Quy trình xóa cây tìm kiếm nhị phân

### **2.3.5 Hàm hủy và sao chép**

Như thường lệ, hàm hủy gọi `makeEmpty`. Công khai `makeEmpty` (không hiển thị) chỉ đơn giản gọi phiên bản đệ quy riêng. Như trong Hình 2.13, sau khi xử lý đệ quy các con của `t`, một cuộc gọi để xóa được thực hiện cho `t`. Do đó, tất cả các nút được thu hồi một cách đệ quy. Lưu ý rằng tại `end`, `t`, và do đó `root`, được thay đổi thành điểm `nullptr`. Hàm tạo bản sao, được hiển thị trong Hình 2.14, theo quy trình thông thường, đầu tiên khởi tạo `root` thành `nullptr` và sau đó thực hiện một bản sao của `rhs`. Chúng tôi sử dụng một hàm đệ quy rất mượt có tên là `clone` để thực hiện tất cả các công việc.

### **2.3.6 Phân tích trường hợp trung bình**

Theo trực giác, chúng tôi mong đợi rằng tất cả các hoạt động được mô tả trong phần này, ngoại trừ `makeEmpty` và sao chép, sẽ mất thời gian  $O(\log N)$ , bởi vì trong thời gian không đổi, chúng tôi giảm một cấp trong cây, do đó hoạt động trên một cây hiện đã lớn bằng một nửa. Thật vậy, thời gian chạy của tất cả các hoạt động (ngoại trừ `makeEmpty` và sao chép) là  $O(d)$ , trong đó  $d$  là độ sâu của nút có chứa mục đã truy cập (trong trường hợp xóa, đây có thể là nút thay thế trong vụ hai đứa trẻ). Trong phần này, chúng tôi chứng minh rằng độ sâu trung bình trên tất cả các nút trong cây là  $O(\log N)$  trên giả định rằng tất cả các chuỗi chèn đều có khả năng xảy ra như nhau. Tổng độ sâu của tất cả các nút trong cây được gọi là chiều dài đường dẫn nội bộ. Nay giờ chúng ta sẽ tính toán độ dài đường dẫn nội bộ trung bình của cây tìm kiếm nhị phân, trong đó trung bình được lấy trên tất cả các chuỗi chèn có thể có vào cây tìm kiếm nhị phân.

```

1  /**
2   * Destructor for the tree
3   */
4 -BinarySearchTree( )
5 {
6     makeEmpty( );
7 }
8 /**
9  * Internal method to make subtree empty.
10 */
11 void makeEmpty( BinaryNode * & t )
12 {
13     if( t != nullptr )
14     {
15         makeEmpty( t->left );
16         makeEmpty( t->right );
17         delete t;
18     }
19     t = nullptr;
20 }

```

Hình 2.13 Hàm hủy và hàm thành viên makeEmpty đệ quy

```

1 /**
2  * Copy constructor
3 */
4 BinarySearchTree( const BinarySearchTree & rhs ) : root{ nullptr }
5 {
6     root = clone( rhs.root );
7 }
8
9 /**
10 * Internal method to clone subtree.
11 */
12 BinaryNode * clone( BinaryNode *t ) const
13 {
14     if( t == nullptr )
15         return nullptr;
16     else
17         return new BinaryNode{ t->element, clone( t->left ), clone( t->right ) };
18 }

```

Hình 2.14 Sao chép hàm tạo và hàm thành viên sao chép đệ quy

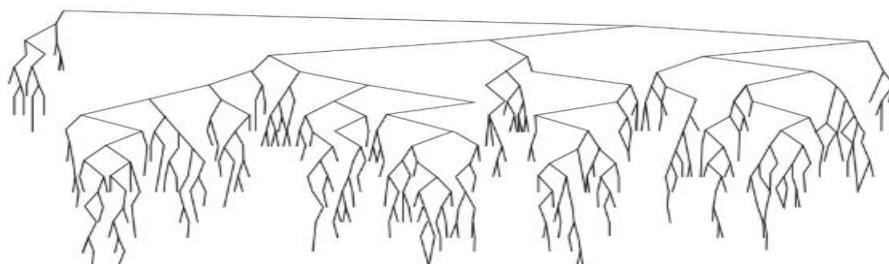
Gọi  $D(N)$  là độ dài đường đi trong của một số cây  $T$  gồm  $N$  nút.  $D(1) = 0$ . Một nút  $N$  cây bao gồm một cây con bên trái thứ  $i$  và một cây con bên phải ( $N - i - 1$ ), cộng với một gốc tại độ sâu bằng không đối với  $0 \leq i < N$ .  $D(i)$  là độ dài đường dẫn bên trong của cây con bên trái liên quan đến gốc của nó. Trong cây chính, tất cả các nút này sâu hơn một cấp. Điều tương tự đối với quyền cây con.

$$D(N) = D(i) + D(N - i - 1) + N - 1$$

Nếu tất cả các con cây kích thước đều có khả năng giống nhau, điều này đúng với nhị phân cây tìm kiếm, nhưng không với nhị phân cây thì giá trị trung bình của cả  $D(i)$  và  $D(N - i - 1)$  là  $(1/N) \sum_{j=0}^{N-1} D(j)$ . Điều này mang lại:

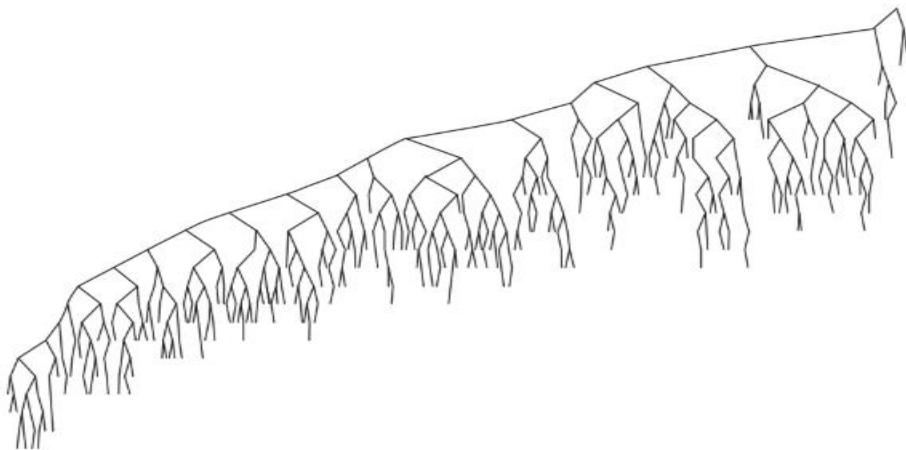
$$D(N) = \frac{2}{N} \left[ \sum_{j=0}^{N-1} D(j) \right] + N - 1$$

Sự lặp lại này sẽ được gấp và giải quyết trong Chương 7, thu được giá trị trung bình là  $D(N) = O(N \log N)$ . Do đó, độ sâu dự kiến của bất kỳ nút nào là  $O(\log N)$ . Ví dụ, Cây 500 nút được tạo ngẫu nhiên thể hiện trong Hình 2.15 có các nút ở độ sâu kỳ vọng 9,98. Thật hấp dẫn để nói ngay rằng kết quả này ngụ ý rằng thời gian chạy trung bình của tất cả các hoạt động được thảo luận trong phần trước là  $O(\log N)$ , nhưng điều này không hoàn toàn thật. Lý do cho điều này là do xóa, không rõ ràng rằng tất cả các tìm kiếm nhị phân cây có khả năng như nhau. Đặc biệt, thuật toán xóa được mô tả ở trên ủng hộ việc các cây con bên trái sâu hơn bên phải, vì chúng tôi luôn thay thế một nút đã xóa với một nút từ cây con bên phải. Hiệu quả chính xác của chiến lược này vẫn chưa được biết, nhưng nó dường như chỉ là một tính mới trên lý



Hình 2.15 Cây tìm kiếm nhị phân được tạo ngẫu nhiên

thuyết. Nó đã được chứng minh rằng nếu chúng ta thay thế các lần chèn và số lần xóa (N2), khi đó cây sẽ có độ sâu dự kiến là ( $\sqrt{N}$ ). Sau một phần tư triệu cặp chèn / loại bỏ ngẫu nhiên, cây hơi nặng Hình 2.15 có vẻ không cân đối (độ sâu trung bình = 12,51) trong Hình 2.16. Chúng tôi có thể cố gắng loại bỏ vấn đề bằng cách chọn ngẫu nhiên giữa các phần tử trong cây con bên phải và lớn nhất ở bên trái khi thay thế phần tử bị xóa. Điều này dường như loại bỏ sự thiên vị và sẽ giữ cho cây cân bằng, nhưng thực tế không ai có thể



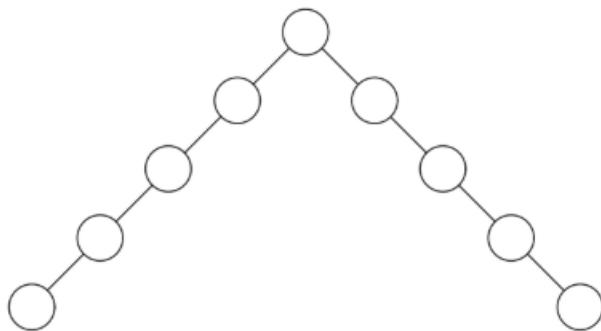
Hình 2.16 Cây tìm kiếm nhị phân sau (N2) chèn / xóa các cặp

chứng minh điều này. Trong mọi trường hợp, hiện tượng này dường như hầu hết là một lý thuyết tính mới, bởi vì hiệu ứng hoàn toàn không hiển thị đối với các cây nhỏ, và người lạ vẫn có, nếu o (N2) cặp chèn / loại bỏ được sử dụng, sau đó cây dường như đạt được sự cân bằng! Điểm chính của cuộc thảo luận này là việc quyết định “trung bình” nghĩa là gì- đồng minh cực kỳ khó khăn và có thể yêu cầu các giả định có thể có hoặc có thể không hợp lệ. bên trong không có xóa hoặc khi xóa lười được sử dụng, chúng tôi có thể kết luận rằng thời gian chạy của các hoạt động trên là O ( $\log N$ ). Trừ những trường hợp lạ như sau đã thảo luận ở trên, kết quả này rất phù hợp với hành vi quan sát được. Nếu đầu vào là một cây được sắp xếp trước, thì một loạt các phép chèn sẽ có dạng bậc hai thời gian và triển khai một danh sách liên kết rất tốn kém, vì cây sẽ bao gồm chỉ trong số các nút không có nút con bên trái. Một giải pháp cho vấn đề là nhấn mạnh thêm điều kiện cấu trúc được gọi là cân bằng: Không có nút nào được phép đi quá sâu. Có khá nhiều thuật toán

chung để thực hiện cây cân bằng. Hầu hết đều khá phức tạp hơn một chút so với cây tìm kiếm nhị phân tiêu chuẩn và trung bình tất cả đều mất nhiều thời gian hơn để cập nhật. Tuy nhiên, chúng cung cấp khả năng bảo vệ khỏi những trường hợp đơn giản đáng xấu hổ. Dưới đây, chúng tôi sẽ phác thảo một trong những dạng lâu đời nhất của cây tìm kiếm cân bằng, cây AVL. Phương pháp thứ hai là bỏ điều kiện cân bằng và cho phép cây tùy ý sâu, nhưng sau mỗi hoạt động, một quy tắc tái cấu trúc được áp dụng có xu hướng tạo ra tương lai hoạt động hiệu quả. Các loại cấu trúc dữ liệu này thường được phân loại là tự điều chỉnh. Trong trường hợp cây tìm kiếm nhị phân, chúng tôi không còn có thể đảm bảo ràng buộc  $O(\log N)$  trên bất kỳ hoạt động đơn lẻ nhưng có thể thấy rằng bất kỳ chuỗi  $M$  hoạt động nào cũng chiếm tổng thời gian là  $O(M \log N)$  trong trường hợp xấu nhất. Điều này nói chung là đủ bảo vệ chống lại trường hợp xấu nhất.

## 2.4 Cây AVL

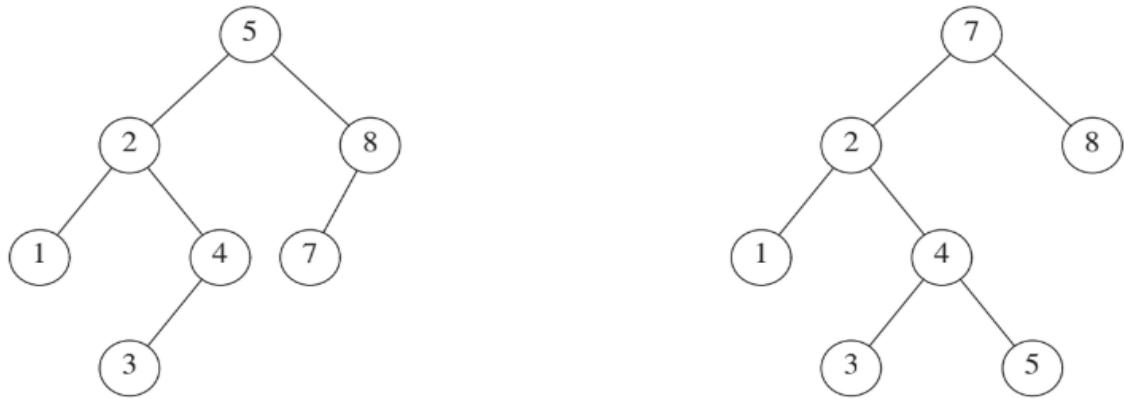
Cây AVL (Adelson-Velskii và Landis) là cây tìm kiếm nhị phân với điều kiện cân bằng. Điều kiện cân bằng phải dễ duy trì và đảm bảo độ sâu của cây là  $O(\log N)$ . Ý tưởng đơn giản nhất là yêu cầu các cây con bên trái và bên phải có cùng Chiều cao. Như Hình 2.17 cho thấy .



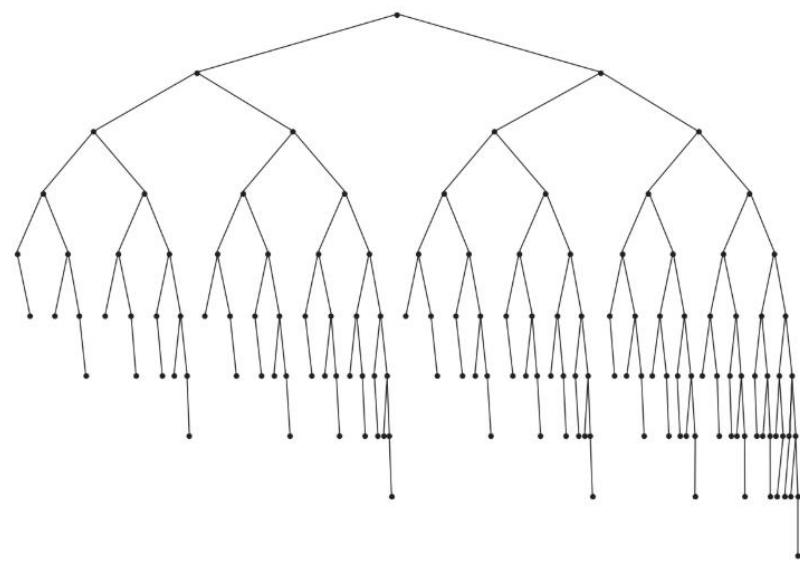
Hình 2.17 Một cây nhị phân xáu. Yêu cầu sự cân bằng ở gốc là không đủ

Một điều kiện cân bằng khác sẽ nhấn mạnh rằng mọi nút phải có con trái và con phải các cây cùng chiều cao. Nếu chiều cao của cây con trống được xác định là  $-1$  (như là thông thường), thì chỉ những cây cân bằng hoàn hảo gồm  $2k - 1$  nút mới thỏa mãn tiêu chí này. Do đó, mặc dù điều này đảm bảo cho cây có độ sâu nhỏ, nhưng điều kiện cân bằng quá khắt khe để hữu ích và cần

được thư giãn. Cây AVL giống hệt cây tìm kiếm nhị phân, ngoại trừ cây đối với mọi nút trong cây, chiều cao của cây con bên trái và bên phải có thể chênh lệch nhiều nhất là 1. (Chiều cao của cây trống cây được định nghĩa là -1.) Trong Hình 2.18, cây bên trái là cây AVL nhưng cây trên đúng là không. Thông tin độ cao được giữ cho mỗi nút (trong cấu trúc nút). Nó có thể cho thấy rằng chiều cao của cây AVL tối đa là khoảng  $1,44 \log(N + 2) - 1,328$ , nhưng trong thực hành nó chỉ nhiều hơn một chút so với  $\log N$ . Ví dụ, cây AVL có chiều cao 9 với ít nút nhất (143) được thể hiện trong Hình 2.19. Cây này có một cây con bên trái một cây AVL chiều cao 7 kích thước tối thiểu. Cây con bên phải là cây AVL có kích thước tối thiểu là 8 chiều cao. Điều này cho chúng ta biết rằng số lượng nút tối thiểu,  $S(h)$ , trong cây AVL có chiều cao  $h$  được đưa ra bởi  $S(h) = S(h - 1) + S(h - 2) + 1$ . Với  $h = 0$ ,  $S(0) = 1$ . Với  $h = 1$ ,  $S(1) = 2$ . Hàm  $S(h)$  có liên quan chặt chẽ với các số Fibonacci, từ đó giới hạn được yêu cầu ở trên chiều cao của cây AVL theo sau. Do đó, tất cả các hoạt động cây có thể được thực hiện trong thời gian  $O(\log N)$ , ngoại trừ có thể chèn và xóa. Khi chúng tôi thực hiện chèn, chúng tôi cần cập nhật tất cả số dư thông tin cho các nút trên đường dẫn trở lại gốc, nhưng lý do chèn có khả năng khó là việc chèn một nút có thể vi phạm thuộc tính cây AVL. (Đối với ví dụ, chèn 6 vào cây AVL trong Hình 2.18 sẽ phá hủy điều kiện cân bằng tại nút có khóa 8.) Nếu trường hợp này xảy ra, thì thuộc tính phải được khôi phục trước khi bước chèn coi như kết thúc. Nó chỉ ra rằng điều này luôn luôn có thể được thực hiện với một sửa đổi đối với cây, được gọi là một vòng quay. Sau khi chèn, chỉ các nút nằm trên đường dẫn từ điểm chèn đến gốc số dư của chúng có thể bị thay đổi vì chỉ những nút đó mới có các cây con bị thay đổi. Như chúng tôi đã theo đường dẫn đến gốc và cập nhật thông tin cân bằng, chúng tôi có thể tìm thấy nút có số dư mới vi phạm điều kiện AVL. Chúng tôi sẽ chỉ ra cách cân bằng lại cây ở nút đầu tiên (tức là sâu nhất) như vậy và chúng tôi sẽ chứng minh rằng sự cân bằng lại này đảm bảo rằng toàn bộ cây thỏa mãn thuộc tính AVL.



Hình 2.18 Hai cây tìm kiếm nhị phân. Chỉ có cây bên trái là AVL.



Hình 2.19 Cây AVL nhỏ nhất có chiều cao là 9

Chúng ta hãy gọi là nút phải được cân bằng lại  $\alpha$ . Vì bất kỳ nút nào cũng có nhiều nhất hai children và sự mất cân bằng chiều cao yêu cầu chiều cao của hai cây phụ của  $\alpha$  khác nhau hai, điều đó thật dễ dàng để thấy rằng vi phạm có thể xảy ra trong bốn trường hợp:

1. Một phần chèn vào cây con bên trái của con trái của  $\alpha$
2. Một phần chèn vào cây con bên phải của con trái của  $\alpha$
3. Một phần chèn vào cây con bên trái của cây con bên phải của  $\alpha$
4. Một phần chèn vào cây con bên phải của con bên phải của  $\alpha$

### 2.4.1 Xoay đơn

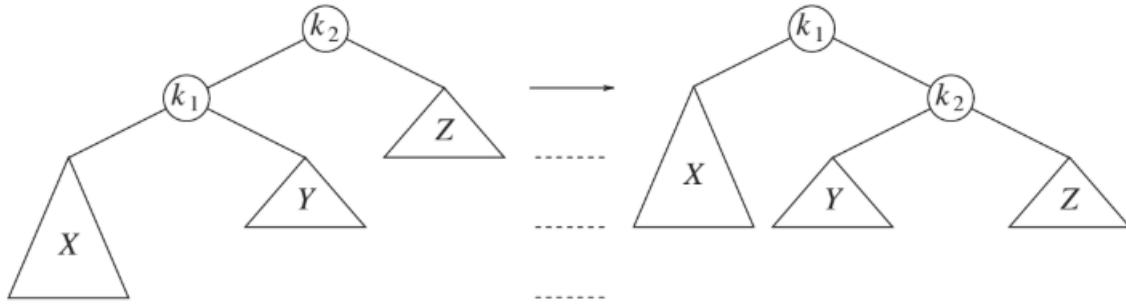
Hình 2.20 cho thấy một vòng quay duy nhất sửa chữa trường hợp 1. Hình trước ở bên trái và hình sau ở bên phải. Hãy cùng chúng tôi phân tích kỹ những gì đang diễn ra. Nút k2 vi phạm thuộc tính số dư AVL vì cây con bên trái của nó sâu hơn hai mức so với cây con bên phải (các đường đứt nét ở giữa biểu đồ đánh dấu các mức). Tình huống được mô tả là trường hợp duy nhất có thể xảy ra 1 kịch bản cho phép k2 thỏa mãn thuộc tính AVL trước khi chèn nhưng lại vi phạm sau đó. Cây con X đã phát triển đến một mức phụ, khiến nó sâu hơn chính xác hai mức so với Z. Y không thể ở cùng mức với X mới vì khi đó k2 sẽ mất cân bằng trước khi chèn và Y không thể ở cùng mức với Z vì khi đó k1 sẽ là nút đầu tiên trên đường dẫn tới gốc vi phạm điều kiện cân bằng AVL.

Để cân bằng lại cây một cách lý tưởng, chúng tôi muốn di chuyển X lên một cấp và Z xuống một cấp. Lưu ý rằng điều này thực sự nhiều hơn thuộc tính AVL sẽ yêu cầu. Để làm điều này, chúng tôi sắp xếp lại các nút thành một cây tương đương như thể hiện trong phần thứ hai của Hình 2.20. Đây là một tình huống trừu tượng: Hình dung cái cây như đang mềm dẻo, nắm lấy nút con k1, nhắm mắt và lắc nó, để cho trọng lực giữ. Kết quả là k1 sẽ là gốc mới. Thuộc tính cây tìm kiếm nhị phân cho chúng ta biết rằng trong cây ban đầu  $k2 > k1$ , do đó k2 trở thành con bên phải của k1 trong cây mới. X và Z tương ứng là con trái của k1 và con phải của k2. Cây con Y, chứa các mục nằm giữa k1 và k2 trong cây ban đầu, có thể được đặt làm con bên trái của k2 trong cây mới và đáp ứng tất cả các yêu cầu đặt hàng.

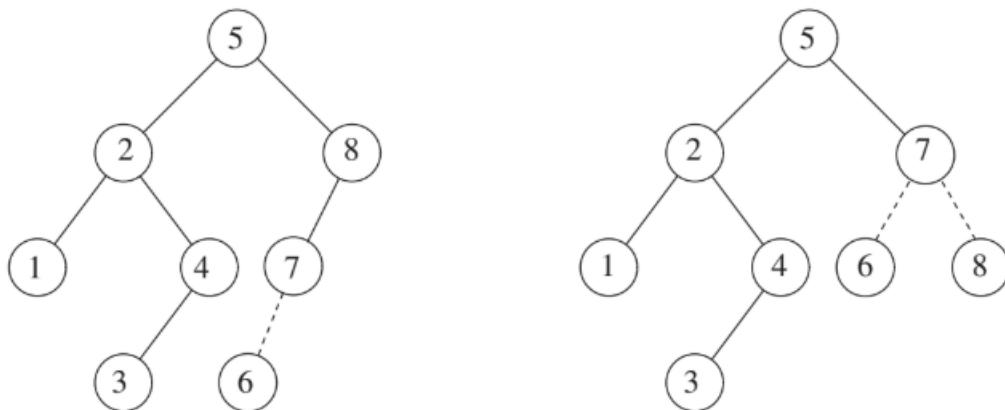
Kết quả của công việc này, chỉ yêu cầu một vài thay đổi con trỏ, chúng ta có một cây tìm kiếm nhị phân khác là cây AVL. Điều này xảy ra bởi vì X tăng một cấp, Y vẫn ở cùng một mức và Z giảm xuống một cấp. k2 và k1 không chỉ thỏa mãn các yêu cầu của AVL mà chúng còn có các cây con có cùng chiều cao. Hơn nữa, chiều cao mới của toàn bộ cây con hoàn toàn giống với chiều cao của cây con ban đầu trước khi chèn khiến X lớn lên. Do đó, không cần cập nhật thêm độ cao trên đường dẫn đến gốc, và do đó không cần quay thêm nữa. Hình 2.21 cho thấy sau khi chèn nút 6 vào cây AVL ban đầu ở bên trái, nút 8 trở nên không cân bằng. Do đó, chúng tôi thực hiện một vòng quay duy nhất giữa 7 và 8, thu được cây ở bên phải.

Như chúng ta đã đề cập trước đó, trường hợp 4 đại diện cho một trường hợp đối xứng. Hình 2.22 cho thấy cách áp dụng một phép quay. Hãy để chúng tôi

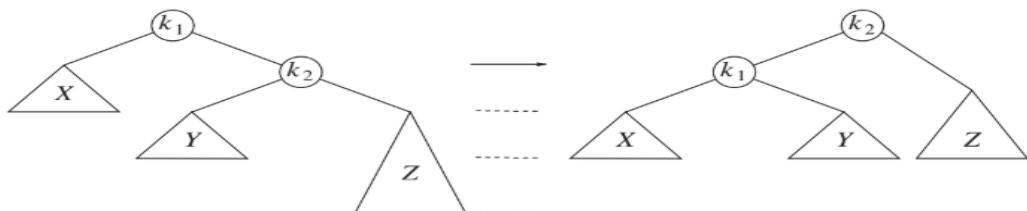
làm việc qua một ví dụ khá dài. Giả sử chúng ta bắt đầu với một cây AVL trống ban đầu và chèn các mục 3, 2, 1, và sau đó là 4 đến 7 theo thứ tự tuần tự. Sự cố đầu tiên xảy ra khi đến lúc chèn mục 1 vì thuộc tính AVL bị vi phạm tận gốc.



Hình 2.20 Xoay đơn để sửa trường hợp 1

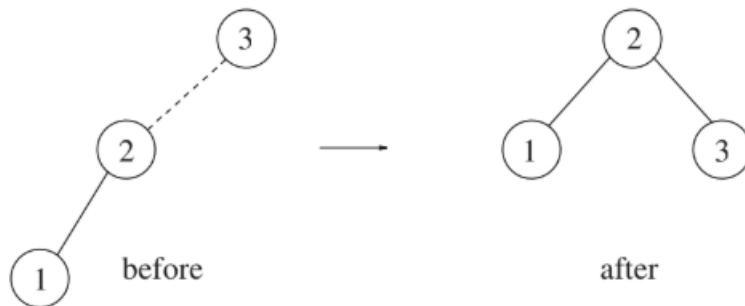


Hình 2.21 Thuộc tính AVL bị phá hủy bằng cách chèn 6, sau đó được cố định bằng một vòng quay

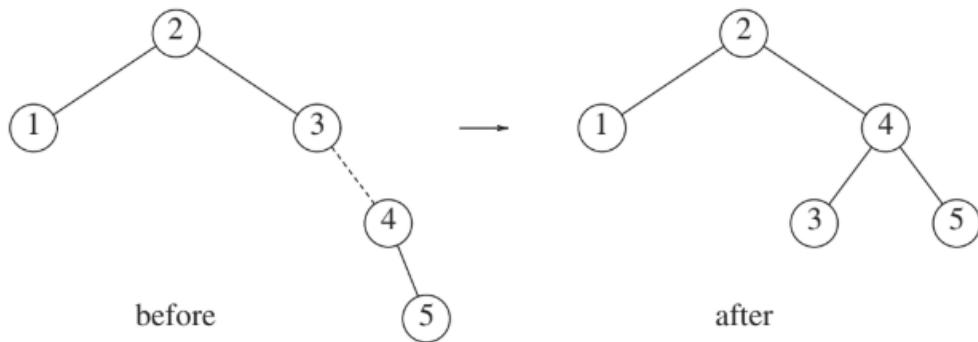


Hình 2.22 Xoay đơn sửa chữa trường hợp 4

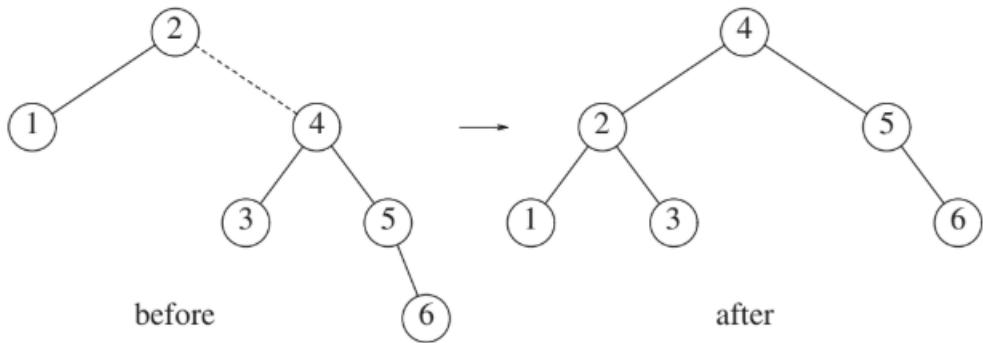
Chúng tôi thực hiện một vòng quay duy nhất giữa gốc và trái của nó con để khắc phục sự cố. Đây là những cây trước và sau:



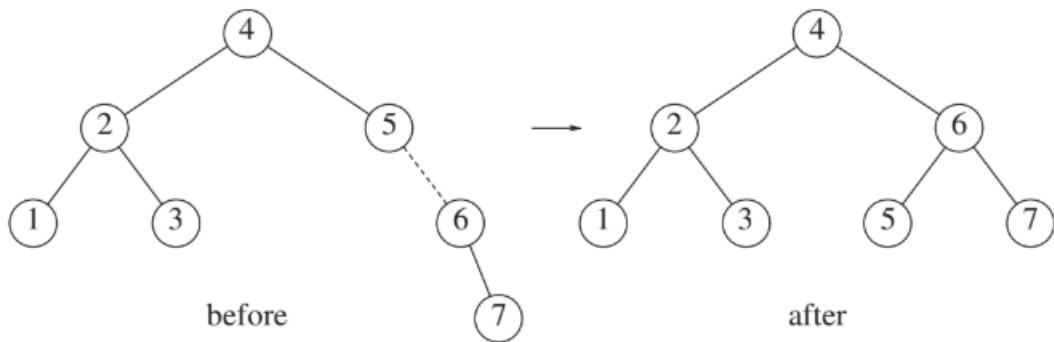
Một đường đứt nét nối hai nút là đối tượng của phép quay. Tiếp theo, chúng tôi chèn 4, không gây ra vấn đề gì, nhưng việc chèn 5 tạo ra vi phạm ở nút 3 đã được khắc phục bằng một vòng quay duy nhất. Bên cạnh sự thay đổi cục bộ do việc xoay vòng, người lập trình phải Hãy nhớ rằng phần còn lại của cây phải được thông báo về sự thay đổi này. Đây có nghĩa là Con bên phải của 2 phải được đặt lại thành liên kết thành 4 thay vì 3. Quên làm như vậy rất dễ và sẽ phá hủy cây (4 sẽ không thể truy cập được).



Tiếp theo, chúng tôi chèn 6. Điều này gây ra vấn đề số dư ở gốc, vì cây con bên trái của nó là chiều cao 0 và cây con bên phải của nó sẽ là chiều cao 2. Do đó, chúng tôi thực hiện một phép quay duy nhất tại gốc từ 2 đến 4.

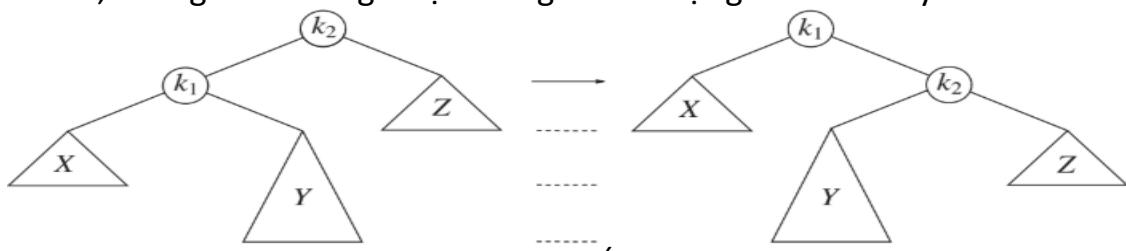


Phép quay được thực hiện bằng cách đặt 2 cây con của cây con bên trái ban đầu của 4 và 4 thành cây con bên phải mới cây con của 2. Mọi mục trong cây con này phải nằm giữa 2 và 4, vì vậy phép biến đổi này có ý nghĩa. Mục tiếp theo chúng tôi chèn là 7, gây ra một vòng quay khác:

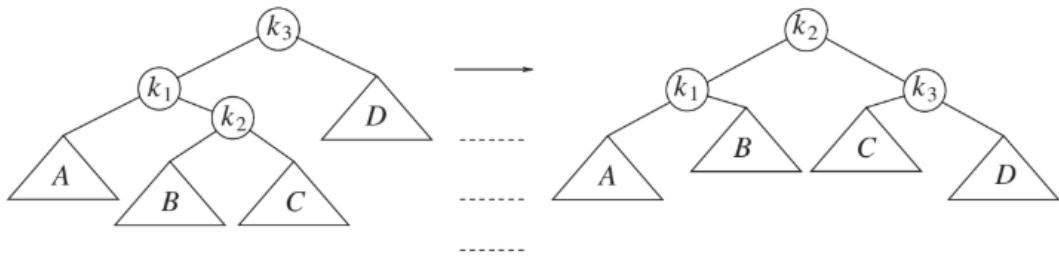


#### 2.4.2 Xoay kép

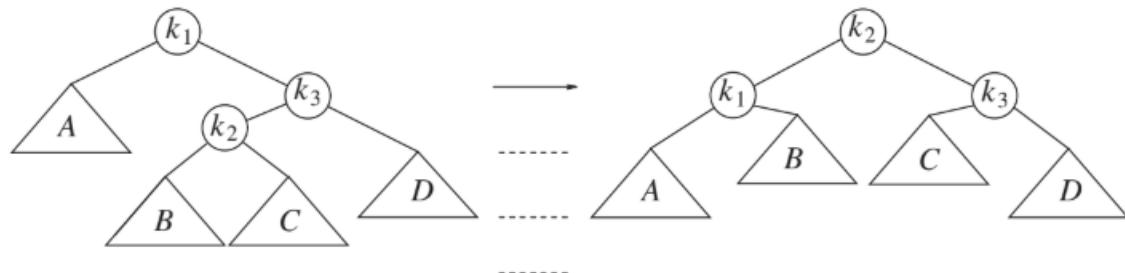
Thuật toán được mô tả ở trên có một vấn đề: Như Hình 2.23 cho thấy, nó không hoạt động đối với trường hợp 2 hoặc 3. Vấn đề là cây con Y quá sâu và một phép quay đơn không làm cho nó bớt sâu hơn. Phép quay kép giải quyết vấn đề được thể hiện trong Hình 2.24. Thực tế là cây con Y trong Hình 2.23 đã có một mục được chèn vào nó đảm bảo rằng nó là không ai cả. Do đó, chúng ta có thể giả định rằng nó có một gốc và hai cây con



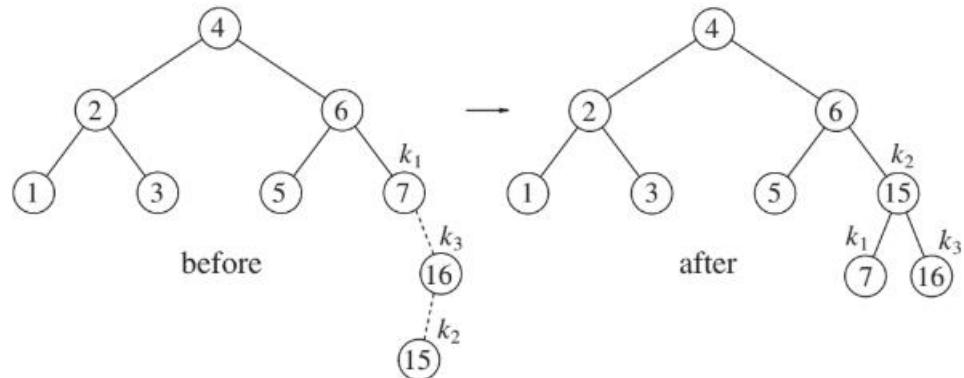
Hình 2.23 Xoay đơn không khắc phục được trường hợp 2



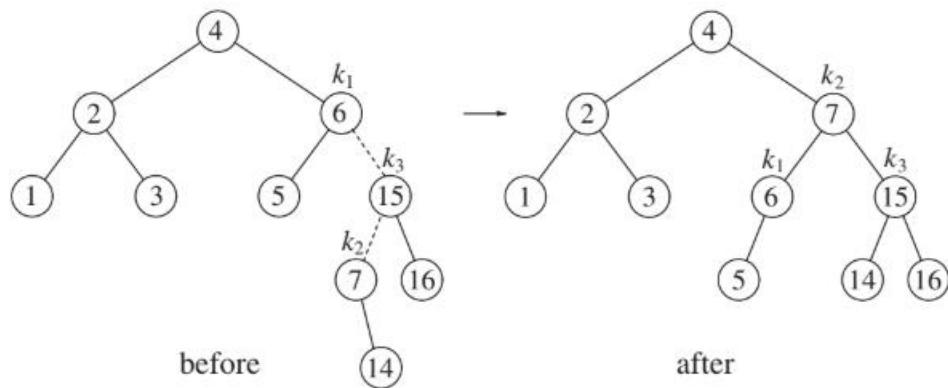
Hình 2.24 Xoay kép trái-phải để sửa trường hợp 2



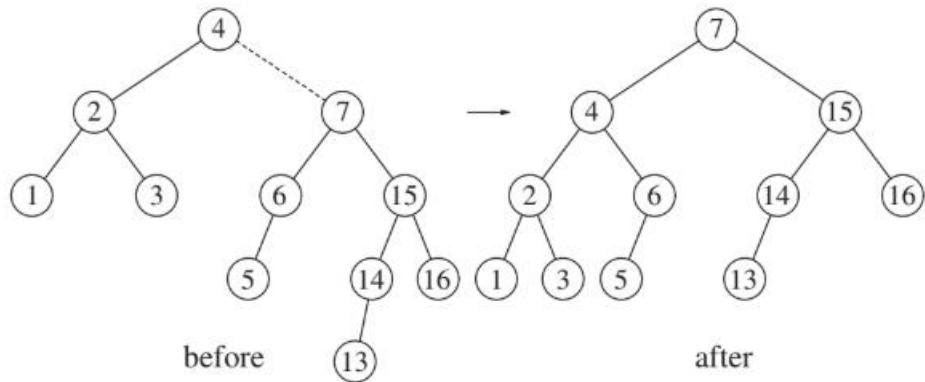
Hình 2.25 Xoay kép phải-trái để sửa trường hợp 3



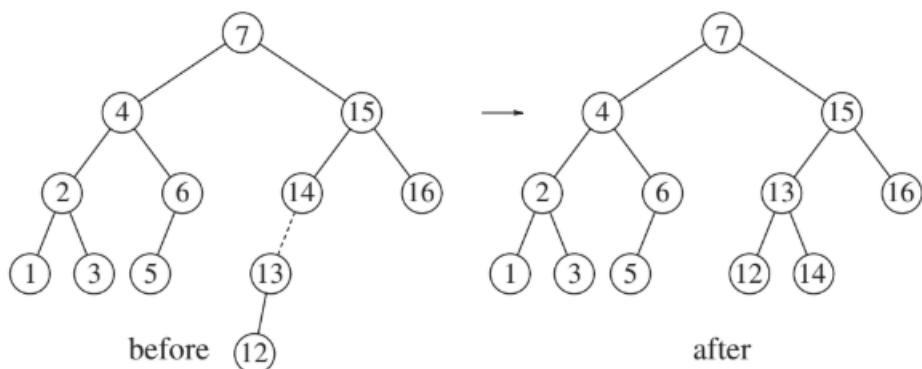
Tiếp theo, chúng tôi chèn 14, cũng yêu cầu quay kép. Đây là vòng quay kép điều đó sẽ khôi phục cây một lần nữa là một vòng quay kép phải-trái sẽ bao gồm 6, 15 và 7. Trong trường hợp này, k1 là nút có mục 6, k2 là nút có mục 7 và k3 là nút với mục 15. Cây con A là cây bắt nguồn từ nút có mục 5; cây con B là trống cây con ban đầu là con bên trái của nút có mục 7, cây con C là cây bắt nguồn từ nút có mục 14 và cuối cùng, cây con D là cây bắt nguồn từ nút có mục 16.



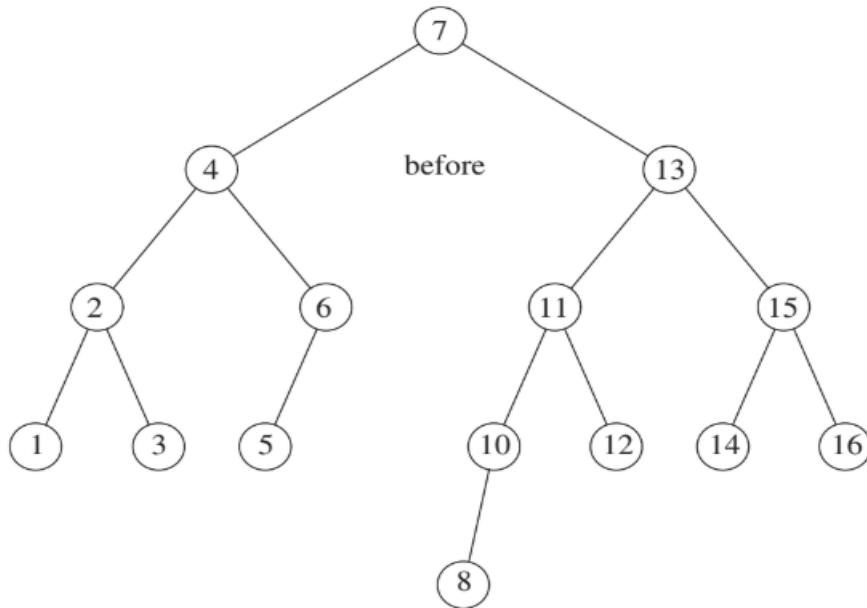
Nếu 13 bây giờ được chèn vào, sẽ có sự mất cân bằng ở gốc. Vì 13 không nằm giữa 4 và 7, chúng ta biết rằng vòng quay duy nhất sẽ hoạt động.



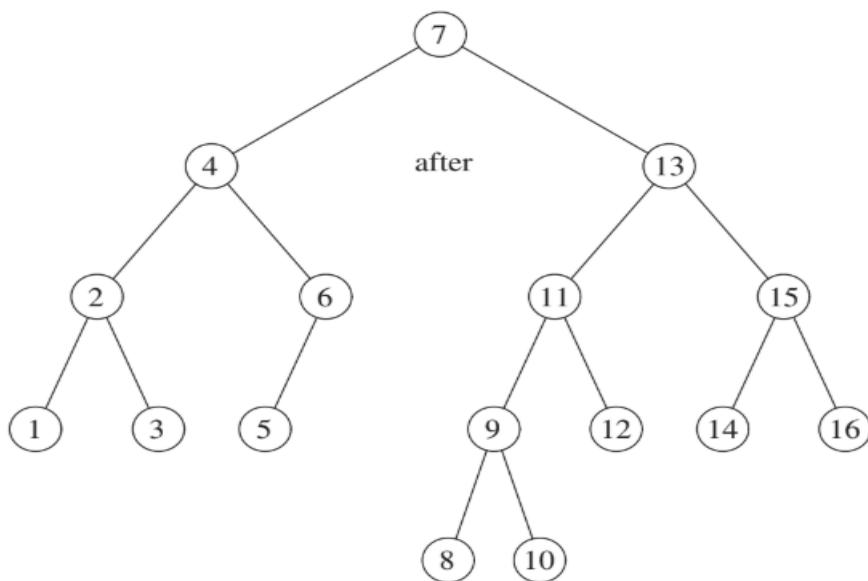
Việc chèn 12 cũng sẽ yêu cầu một lần quay



Để chèn 11, cần thực hiện một vòng quay đơn và điều này cũng đúng với tiếp theo là chèn 10. Chúng tôi chèn 8 mà không cần xoay, tạo ra một cây cân đối:



Cuối cùng, chúng tôi sẽ chèn 9 để hiển thị trường hợp đối xứng của phép quay kép. Để ý rằng 9 làm cho nút chứa 10 trở nên không cân bằng. Vì 9 là từ 10 đến 8 (là con của 10 trên đường dẫn đến 9), cần phải thực hiện xoay kép, mang lại cây sau:



Hãy để chúng tôi tóm tắt những gì xảy ra. Các chi tiết lập trình khá đơn giản ngoại trừ rằng có một số trường hợp. Để chèn một nút mới với mục X vào cây AVL T, chúng tôi đệ quy chèn X vào cây con thích hợp của T (chúng ta gọi đây là TLR). Nếu chiều cao của TLR không thay đổi, sau đó chúng tôi đã hoàn tất. Ngược lại, nếu sự mất cân bằng chiều cao xuất hiện ở T, chúng ta thực hiện xoay vòng đơn hoặc xoay kép thích hợp tùy thuộc vào X và các mục trong T và TLR, cập nhật độ cao (tạo kết nối từ phần còn lại của cây ở trên) và chúng tôi làm xong. Vì một lần xoay luôn đủ, nên thường phiên bản không đệ quy được mã hóa cẩn thận hóa ra nhanh hơn phiên bản đệ quy, nhưng trên các trình biên dịch hiện đại, sự khác biệt là không đáng kể như trong quá khứ. Tuy nhiên, các phiên bản không đệ quy khá khó viết mã một cách chính xác, trong khi một triển khai đệ quy thông thường có thể dễ dàng đọc được. Một vấn đề hiệu quả khác liên quan đến việc lưu trữ thông tin chiều cao. Vì tất cả những điều đó là thực sự cần thiết là sự khác biệt về chiều cao, được đảm bảo là nhỏ, chúng tôi có thể nhận được bằng hai bit (đại diện cho +1, 0, -1) nếu chúng ta thực sự cố gắng. Làm như vậy sẽ tránh lặp lại tính toán các yếu tố cân bằng nhưng dẫn đến một số mất rõ ràng. Mã kết quả là một số phức tạp hơn nếu chiều cao được lưu trữ tại mỗi nút. Nếu một quy trình đệ quy được viết, sau đó tốc độ có lẽ không phải là sự cân nhắc chính. Trong trường hợp này, tốc độ nhẹ lợi thế thu được bằng cách lưu trữ các yếu tố cân bằng dường như không đáng để mất đi sự rõ ràng và tính đơn giản tương đối. Hơn nữa, vì hầu hết các máy sẽ căn chỉnh điều này thành ít nhất 8-bit dù sao thì ranh giới, không có khả năng có bất kỳ sự khác biệt nào về lượng không gian được sử dụng. Một ký tự 8 bit (có dấu) sẽ cho phép chúng tôi lưu trữ chiều cao tuyệt đối lên đến 127. Vì cây là cân bằng, không thể nghĩ rằng điều này sẽ không đủ.

```

1 struct AvlNode
2 {
3     Comparable element;
4     AvlNode *left;
5     AvlNode *right;
6     int height;
7
8     AvlNode( const Comparable & ele, AvlNode *lt, AvlNode *rt, int h = 0 )
9         : element{ ele }, left{ lt }, right{ rt }, height{ h } { }
10
11    AvlNode( Comparable && ele, AvlNode *lt, AvlNode *rt, int h = 0 )
12        : element{ std::move( ele ) }, left{ lt }, right{ rt }, height{ h } { }
13 };

```

Hình 2.26 Khai báo nút cho cây AVL

```

1 /**
2  * Return the height of node t or -1 if nullptr.
3  */
4 int height( AvlNode *t ) const
5 {
6     return t == nullptr ? -1 : t->height;
7 }

```

Hình 2.27 Chức năng tính toán chiều cao của một nút AVL

Với tất cả những điều này, chúng tôi đã sẵn sàng để viết các quy trình AVL. Chúng tôi hiển thị một số mã ở đây; phần còn lại là trực tuyến. Đầu tiên, chúng ta cần lớp AvlNode. Điều này được đưa ra trong Hình 2.26. Chúng tôi cũng cần một hàm nhanh để trả về chiều cao của một nút. Chức năng này là cần thiết để xử lý trường hợp khó chịu của một con trỏ nullptr. Điều này được thể hiện trong Hình 2.27. Chèn cơ bản quy trình (xem Hình 2.28) chỉ thêm một dòng duy nhất ở cuối gọi phương thức cân bằng. Phương pháp cân bằng áp dụng xoay đơn hoặc xoay kép nếu cần, cập nhật chiều cao, và trả về cây kết quả.

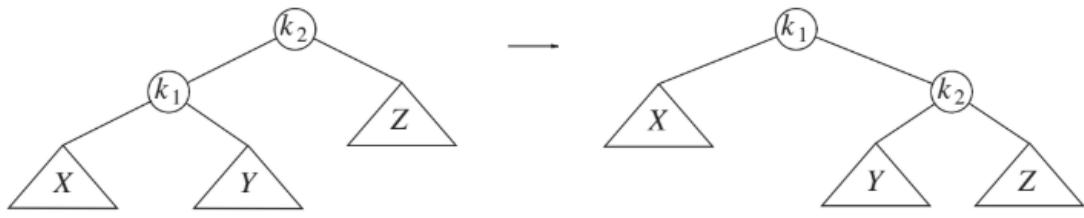
Đối với các cây trong Hình 2.29, xoay nút con bên trái chuyển đổi cây bên trái thành cây ở bên phải, trả về một con trỏ đến gốc mới. xoay nút con bên phải là đối xứng.

Tương tự như vậy, phép quay kép trong Hình 2.30 có thể được thực hiện bằng mã được thể hiện trong Hình 2.31. Vì việc xóa trong cây tìm kiếm nhị phân có phần phức tạp hơn việc chèn, người ta có thể cho rằng việc xóa trong cây AVL cũng phức tạp hơn. Trong một thế giới hoàn hảo, người ta hy vọng rằng quy trình xóa trong Hình 2.31 có thể dễ dàng được sửa đổi bằng cách thay đổi vào dòng cuối cùng để trả về sau khi gọi phương thức số dư, như đã được thực hiện đối với việc chèn. Điều này sẽ mang lại mã trong Hình 2.32. Thay đổi này hoạt động! Việc xóa có thể gây ra một bên của cây trở nên

```

1  /**
2   * Internal method to insert into a subtree.
3   * x is the item to insert.
4   * t is the node that roots the subtree.
5   * Set the new root of the subtree.
6  */
7 void insert( const Comparable & x, AvlNode * & t )
8 {
9     if( t == nullptr )
10        t = new AvlNode( x, nullptr, nullptr );
11    else if( x < t->element )
12        insert( x, t->left );
13    else if( t->element < x )
14        insert( x, t->right );
15
16    balance( t );
17 }
18
19 static const int ALLOWED_IMBALANCE = 1;
20
21 // Assume t is balanced or within one of being balanced
22 void balance( AvlNode * & t )
23 {
24     if( t == nullptr )
25         return;
26
27     if( height( t->left ) - height( t->right ) > ALLOWED_IMBALANCE )
28         if( height( t->left->left ) >= height( t->left->right ) )
29             rotateWithLeftChild( t );
30         else
31             doubleWithLeftChild( t );
32     else
33         if( height( t->right ) - height( t->left ) > ALLOWED_IMBALANCE )
34             if( height( t->right->right ) >= height( t->right->left ) )
35                 rotateWithRightChild( t );
36             else
37                 doubleWithRightChild( t );
38
39     t->height = max( height( t->left ), height( t->right ) ) + 1;
40 }
```

Hình 2.28 Chèn vào cây AVL

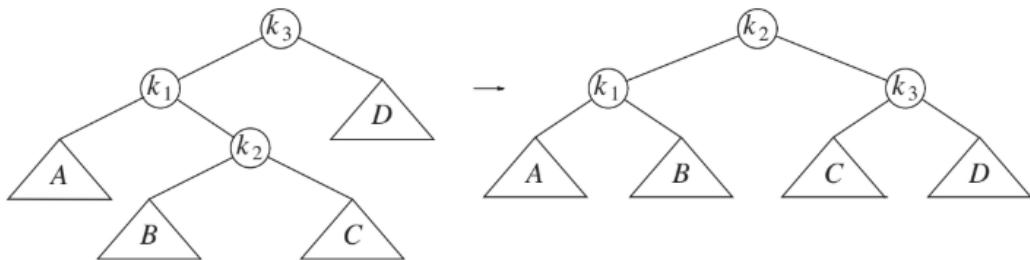


Hình 2.29 Xoay đơn

```

1  /**
2   * Rotate binary tree node with left child.
3   * For AVL trees, this is a single rotation for case 1.
4   * Update heights, then set new root.
5   */
6 void rotateWithLeftChild( AvlNode * & k2 )
7 {
8     AvlNode *k1 = k2->left;
9     k2->left = k1->right;
10    k1->right = k2;
11    k2->height = max( height( k2->left ), height( k2->right ) ) + 1;
12    k1->height = max( height( k1->left ), k2->height ) + 1;
13    k2 = k1;
14 }
```

Hình 2.30 Quy trình thực hiện 1 vòng xoay đơn



Hình 2.31 Xoay kép

thấp hơn hai cấp so với phía bên kia. Phân tích theo từng trường hợp- sis tương tự như sự mất cân bằng do chèn, nhưng không hoàn toàn giống nhau.

Đối với ví dụ, trường hợp 1 trong Hình 2.20, bây giờ sẽ phản ánh sự xóa khỏi cây Z (thay vì phần chèn vào X), phải được tăng cường với khả năng cây Y có thể sâu bằng như cây X. Ngay cả như vậy, có thể dễ dàng thấy rằng phép quay cân bằng lại trường hợp này và sự đối xứng trường hợp 4 trong Hình 2.22. Do đó, mã cho số dư trong Hình 2.28 dòng 28 và 34 sử dụng  $\geq$  thay vì  $>$

```

1  /**
2   * Double rotate binary tree node: first left child
3   * with its right child; then node k3 with new left child.
4   * For AVL trees, this is a double rotation for case 2.
5   * Update heights, then set new root.
6   */
7 void doubleWithLeftChild( AvlNode * & k3 )
8 {
9     rotateWithRightChild( k3->left );
10    rotateWithLeftChild( k3 );
11 }

```

Hình 2.32 Quy trình thực hiện xoay kép

```

1 /**
2  * Internal method to remove from a subtree.
3  * x is the item to remove.
4  * t is the node that roots the subtree.
5  * Set the new root of the subtree.
6 */
7 void remove( const Comparable & x, AvlNode * & t )
8 {
9     if( t == nullptr )
10        return; // Item not found; do nothing
11
12    if( x < t->element )
13        remove( x, t->left );
14    else if( t->element < x )
15        remove( x, t->right );
16    else if( t->left != nullptr && t->right != nullptr ) // Two children
17    {
18        t->element = findMin( t->right )->element;
19        remove( t->element, t->right );
20    }
21    else
22    {
23        AvlNode *oldNode = t;
24        t = ( t->left != nullptr ) ? t->left : t->right;
25        delete oldNode;
26    }
27
28    balance( t );
29 }

```

Hình 2.33 Xóa trong cây AVL

để đảm bảo rằng các phép quay đơn lẻ được thực hiện trong những trường hợp này hơn là phép quay kép. Chúng tôi để xác minh các trường hợp còn lại như một bài tập.

## 2.5 Phát họa cây

Bây giờ chúng ta mô tả một cấu trúc dữ liệu tương đối đơn giản được gọi là cây splay mà guaran- cho rằng bất kỳ phép toán nào liên tiếp trên cây M bắt đầu từ một cây rỗng sẽ mất nhiều nhất  $O(M \log N)$  thời gian. Mặc dù đảm bảo này không loại trừ khả năng rằng bất kỳ hoạt động có thể mất ( $N$ ) thời gian, và do đó ràng buộc không mạnh bằng  $O(\log N)$  tồi tệ nhất- trường hợp ràng buộc cho mỗi hoạt động, hiệu quả thực là như nhau: Không có chuỗi đầu vào xấu. Nói chung, khi một chuỗi M hoạt động có tổng thời gian chạy trong trường hợp xấu nhất là  $O(Mf(N))$ , chúng tôi nói rằng thời gian vận hành được khấu hao là  $O(f(N))$ . Do đó, một cây splay có  $O(\log N)$  chi phí khấu hao mỗi hoạt động. Qua một chuỗi dài các hoạt động, một số có thể mất nhiều hơn, một số ít hơn.

Cây ghép dựa trên thực tế là thời gian trong trường hợp xấu nhất  $O(N)$  trên mỗi hoạt động đối với hệ nhị phân cây tìm kiếm không phải là xấu, miễn là nó xảy ra tương đối không thường xuyên. Bất kỳ một quyền truy cập nào, ngay cả khi nó mất ( $N$ ), vẫn có khả năng là cực kỳ nhanh. Vấn đề với cây tìm kiếm nhị phân là có thể xảy ra cả một chuỗi các truy cập không hợp lệ. Các thời gian chạy tích lũy sau đó trở nên đáng chú ý. Cấu trúc dữ liệu cây tìm kiếm với  $O(N)$  thời gian trong trường hợp xấu nhất, nhưng đảm bảo tối đa  $O(M \log N)$  cho bất kỳ M hoạt động liên tiếp nào, chắc chắn là thỏa đáng, bởi vì không có trình tự xấu.

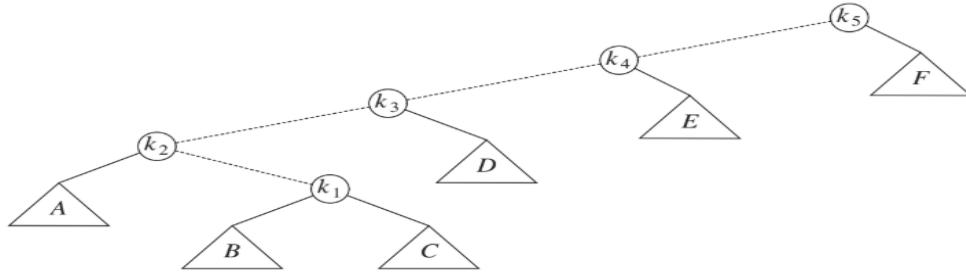
Nếu bất kỳ hoạt động cụ thể nào được phép có giới hạn thời gian trong trường hợp xấu nhất là  $O(N)$ , và chúng tôi vẫn muốn có giới hạn thời gian phân bổ theo  $O(\log N)$ , thì rõ ràng là bất cứ khi nào một nút đã truy cập, nó phải được di chuyển. Nếu không, một khi chúng tôi tìm thấy một nút sâu, chúng tôi có thể tiếp tục thực hiện- ing truy cập vào nó. Nếu nút không thay đổi vị trí và mỗi chi phí truy cập ( $N$ ), thì một chuỗi M truy cập sẽ có giá trị  $(M \cdot N)$ .

Ý tưởng cơ bản của cây splay là sau khi một nút được truy cập, nó sẽ được đẩy về gốc bằng một loạt các phép quay cây AVL. Lưu ý rằng nếu một nút ở

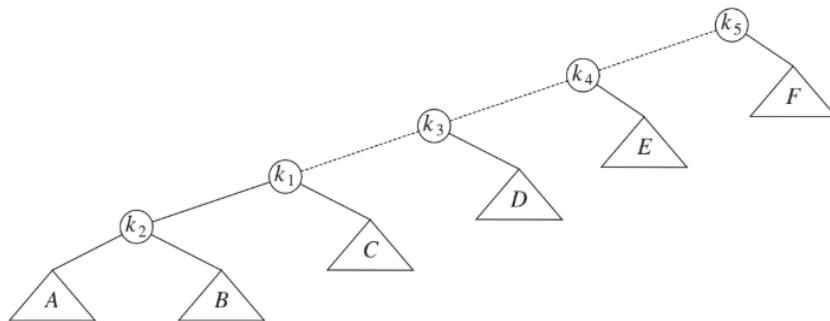
sâu, có nhiều nút trên con đường cũng tương đối sâu và bằng cách tái cấu trúc, chúng tôi có thể thực hiện các tiếp cận trong tương lai rẻ hơn trên tất cả các nút này. Do đó, nếu nút quá sâu, thì chúng ta muốn tái cấu trúc này lại để có tác dụng phụ là cân bằng cây (ở một mức độ nào đó). Bên cạnh việc cho một giới hạn thời gian về mặt lý thuyết, phương pháp này có thể có tiện ích thực tế, bởi vì trong nhiều các ứng dụng, khi một nút được truy cập, nó có khả năng được truy cập lại trong tương lai gần. Các nghiên cứu đã chỉ ra rằng điều này xảy ra thường xuyên hơn nhiều so với những gì người ta mong đợi. Phát cây cũng không yêu cầu duy trì thông tin về chiều cao hoặc cân bằng, do đó tiết kiệm không gian và đơn giản hóa mã ở một mức độ nào đó (đặc biệt là khi triển khai cẩn thận bằng văn bản).

### 2.5.1 Một ý tưởng đơn giản

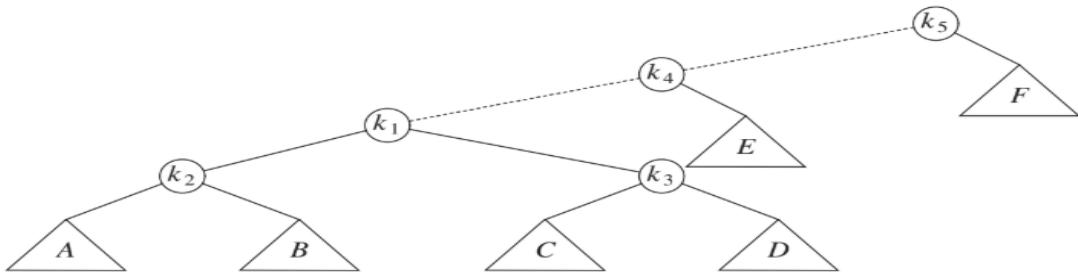
Một cách để thực hiện tái cấu trúc được mô tả ở trên là thực hiện các vòng quay đơn lẻ, từ dưới lên. Điều này có nghĩa là chúng tôi xoay mọi nút trên đường dẫn truy cập với cha của nó. Như một ví dụ, hãy xem xét điều gì sẽ xảy ra sau một lần truy cập (một tìm thấy) trên k1 trong cây sau:



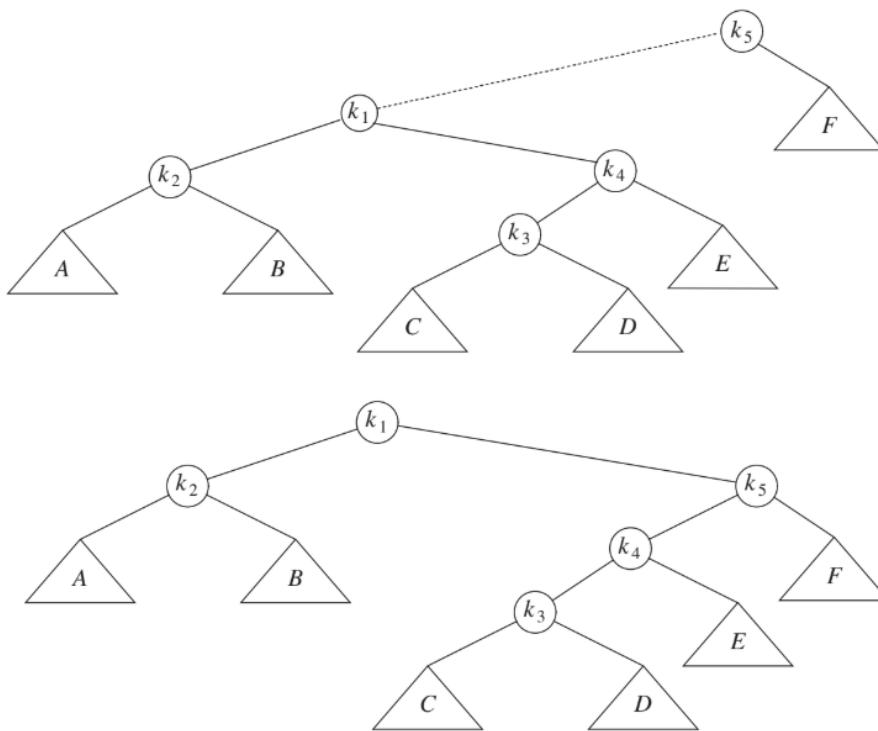
Đường dẫn truy cập bị gạch ngang. Đầu tiên, chúng tôi sẽ thực hiện một phép quay giữa k1 và cha mẹ, lấy cây sau:



Sau đó, chúng tôi xoay giữa k1 và k3, thu được cây tiếp theo:



Sau đó, hai lần quay khác được thực hiện cho đến khi chúng ta đến gốc:



Các phép quay này có tác dụng đẩy k1 về tận gốc, để các truy cập sau này trên k1 được dễ dàng (trong một thời gian). Thật không may, nó đã đẩy một nút khác (k3) xuống sâu gần như k1 trước đây. Một quyền truy cập trên nút đó sau đó sẽ đẩy một nút khác vào sâu, v.v. Mặc dù chiến lược này làm cho các truy cập trong tương lai của k1 rẻ hơn, nhưng nó không cải thiện đáng kể tình hình đối với các nút khác trên đường truy cập (ban đầu). Nó chỉ ra rằng có thể chứng minh rằng sử dụng chiến lược này, có một chuỗi M hoạt động yêu cầu ( $M \cdot N$ ) thời gian, vì vậy ý tưởng này không đủ tốt. Cách đơn giản

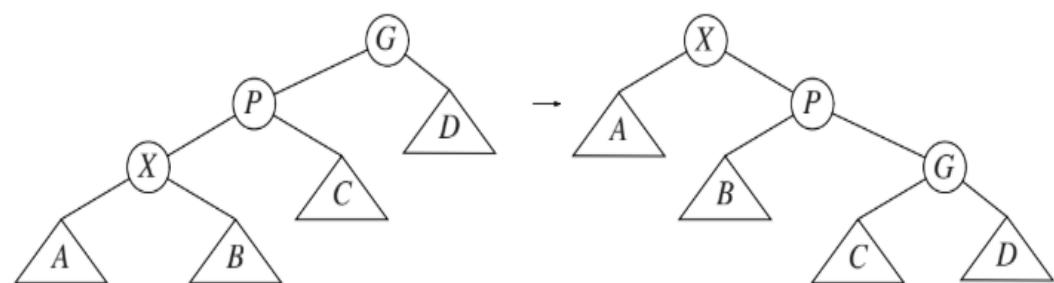
nhất để chỉ ra điều này là xem xét cây được tạo thành bằng cách chèn các phím 1, 2, 3, ..., N vào một cây trống ban đầu (làm ví dụ này). Điều này tạo ra một cây chỉ bao gồm các con còn lại. Tuy nhiên, điều này không hẳn là xấu, vì thời gian để xây dựng cây này là tổng số O (N). Phần tồi tệ là việc truy cập vào nút bằng khóa 1 mất N đơn vị thời gian, trong đó mỗi nút trên đường dẫn truy cập được tính là một đơn vị. Sau khi hoàn tất các phép quay, một lần truy cập vào nút có khóa 2 sẽ mất N đơn vị thời gian, phím 3 lấy N - 1 đơn vị, v.v. Tổng số để truy cập tất cả các khóa theo thứ tự là  $N + N i = 2 i = (N^2)$ . Sau khi chúng được truy cập, cây sẽ trở lại trạng thái ban đầu và chúng ta có thể lặp lại trình tự.

### 2.5.2 Phân bổ

Chiến lược phân bổ tương tự như ý tưởng xoay ở trên, ngoại trừ việc chúng tôi chọn lọc hơn một chút về cách thực hiện các phép quay. Chúng tôi sẽ vẫn xoay từ dưới lên dọc theo đường dẫn truy cập.

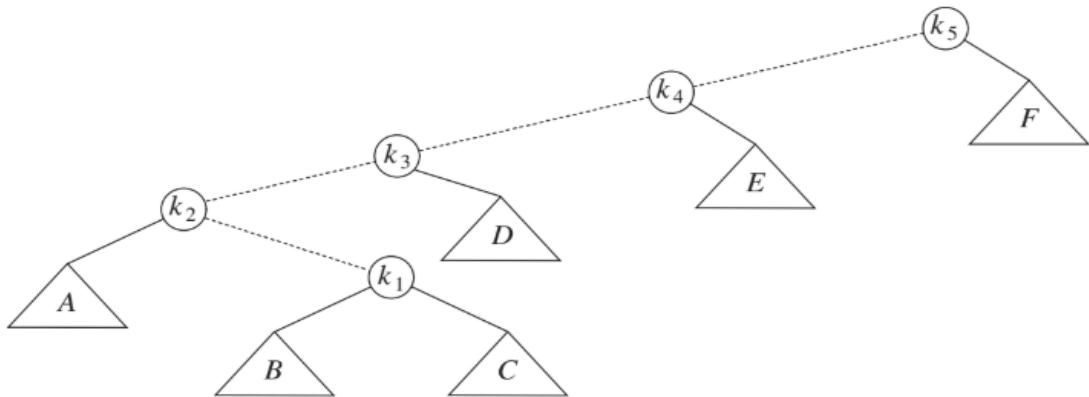


Hình 2.34 Zig-Zag

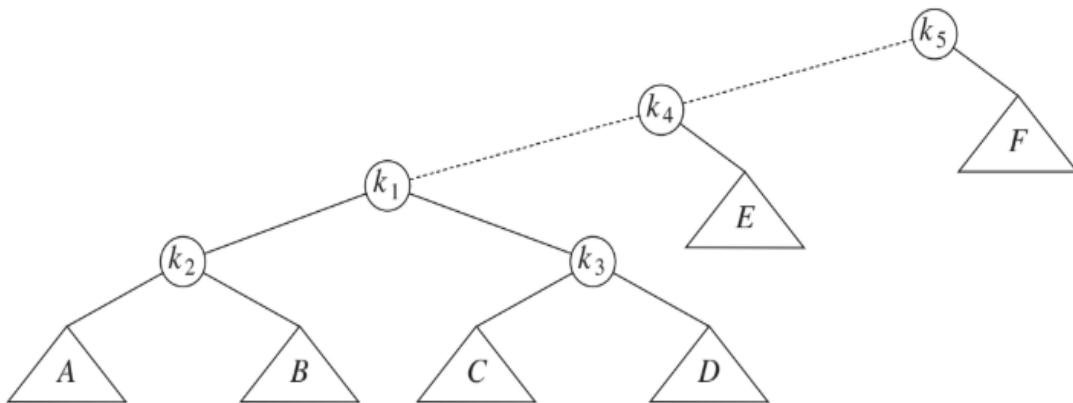


Hình 2.35 Zig-Zig

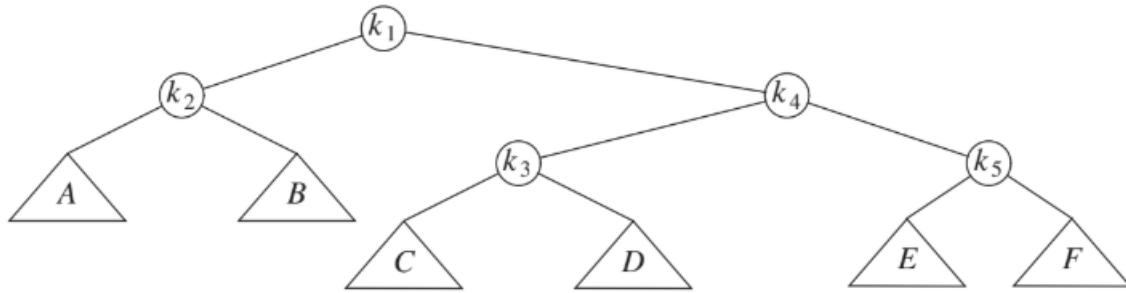
Gọi X là một nút (không phải nút gốc) trên đường dẫn truy cập mà chúng ta đang xoay. Nếu gốc của X là gốc của cây, chúng ta chỉ xoay X và gốc. Đây là vòng quay cuối cùng đọc theo đường dẫn vào. Nếu không, X có cả cha mẹ (P) và ông bà (G), và có hai trường hợp, cộng với các đối xứng, cần xem xét. Trường hợp đầu tiên là trường hợp zig-zag (xem Hình 2.34). Ở đây X là con phải và P là con trái (hoặc ngược lại). Nếu đúng như vậy, chúng tôi thực hiện quay kép, giống hệt như quay kép AVL. Nếu không, chúng ta có một trường hợp zig-zig: X và P đều là con trái (hoặc, trong trường hợp đối xứng, cả hai đều là con phải). Trong trường hợp đó, chúng ta biến đổi cây bên trái của Hình 2.35 thành cây bên phải. Ví dụ, hãy xem xét cây từ ví dụ cuối cùng, với một hàm chứa trên k1:



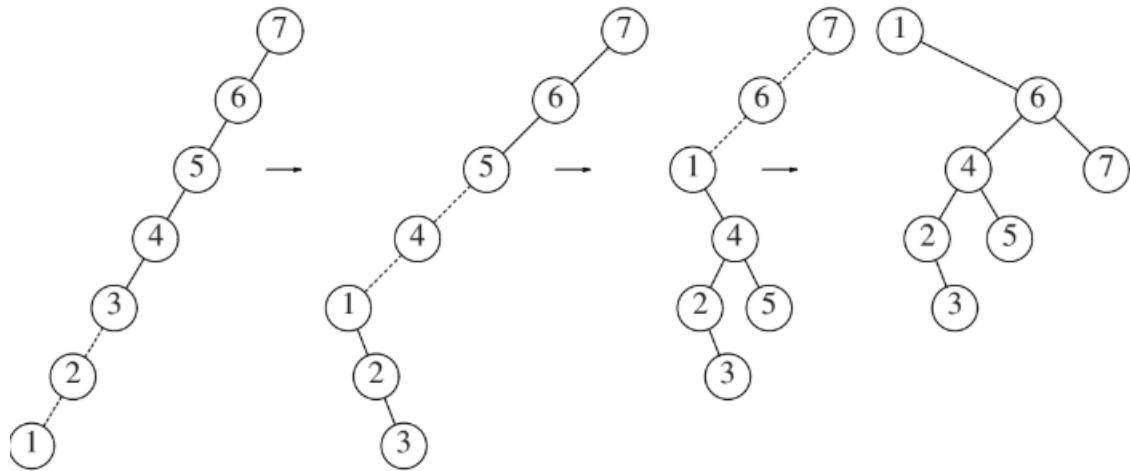
Bước splay đầu tiên là ở k1 và rõ ràng là zig-zag, vì vậy chúng tôi thực hiện kép AVL tiêu chuẩn quay sử dụng k1, k2 và k3. Cây kết quả như sau:



Bước splay tiếp theo tại  $k_1$  là zig-zig, vì vậy chúng tôi thực hiện xoay zig-zig với  $k_1$ ,  $k_4$  và  $k_5$ , lấy cây cuối cùng:



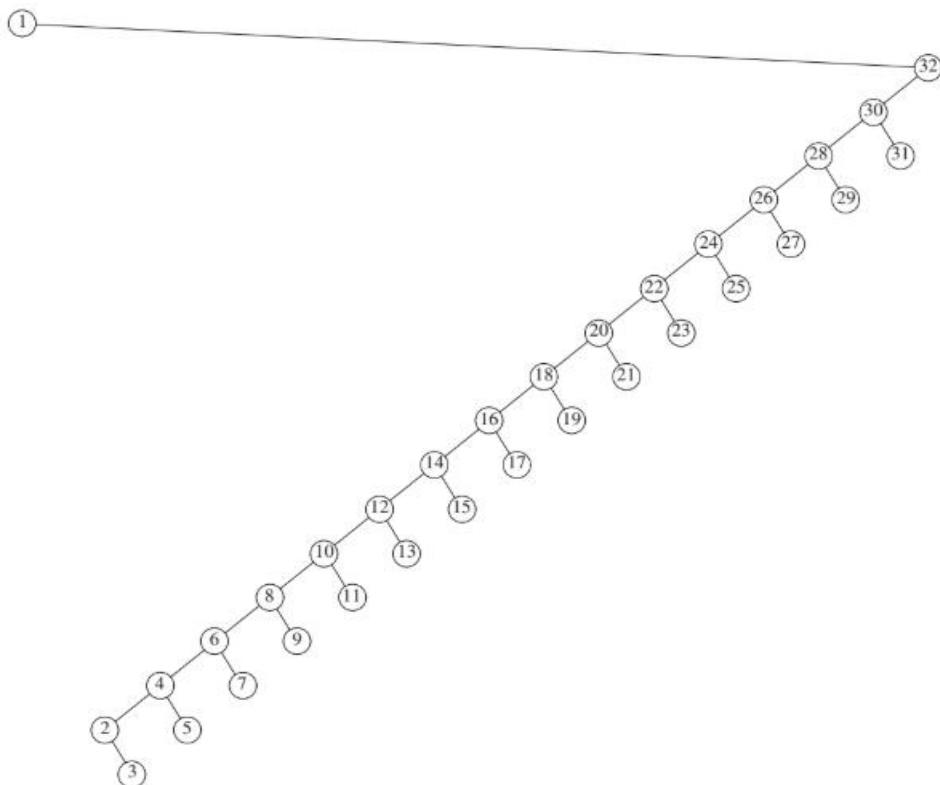
Mặc dù khó có thể nhìn thấy từ các ví dụ nhỏ, việc ghép nối không chỉ di chuyển nút được truy cập về gốc mà còn có tác dụng giảm khoảng một nửa độ sâu của hầu hết các nút trên đường dẫn truy cập (một số nút nông bị đẩy xuống ở nhiều nhất hai cấp). Để thấy sự khác biệt mà phép xếp chồng tạo ra so với phép quay đơn giản, hãy xem xét lại hiệu quả của việc chèn các mục 1, 2, 3, ..., N vào một cây trống ban đầu. Điều này chiếm tổng số  $O(N)$ , như trước đây và tạo ra cây giống như các phép quay đơn giản. Hình 2.36 cho thấy kết quả của sự phân chia tại nút có mục 1. Sự khác biệt là sau khi truy cập vào nút với mục 1, chiếm  $N$  đơn vị, quyền truy cập vào nút với mục 2 sẽ



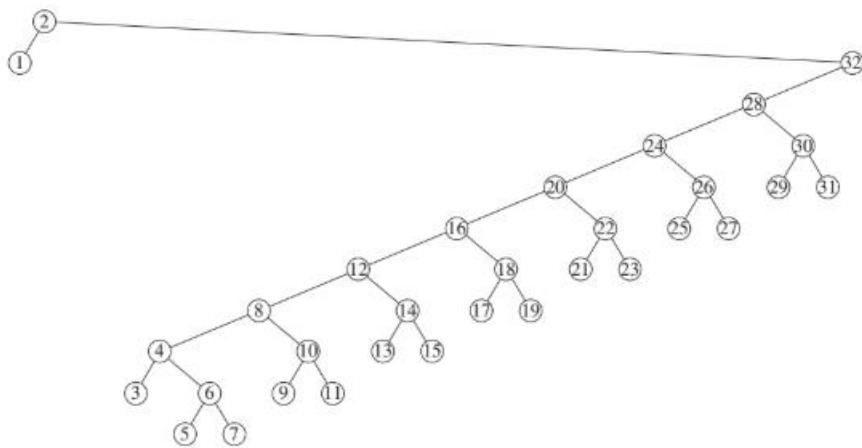
Hình 2.36 Kết quả xuất hiện tại node 1

chỉ mất khoảng  $N / 2$  đơn vị thay vì  $N$  đơn vị; không còn các nút khá sâu như trước. Một quyền truy cập vào nút với mục 2 sẽ đưa các nút đến bên trong  $N / 4$  của gốc và điều này được lặp lại cho đến khi độ sâu trở thành xấp xỉ

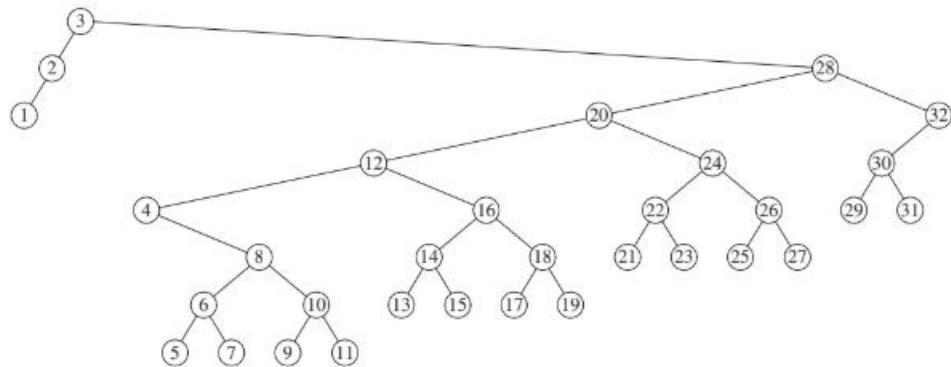
$\log N$  (ví dụ với  $N = 7$  quá nhỏ để thấy hiệu quả tốt). Hình 2.37 đến 2.45 cho thấy kết quả của việc truy cập các mục từ 1 đến 9 trong một cây 32 nút ban đầu chỉ chứa các nút con còn lại. Vì vậy, chúng tôi không nhận được cùng một xáu hành vi từ cây ghép phổ biến trong chiến lược xoay vòng đơn giản. (Thực ra, cái này hóa ra là một trường hợp rất tốt. Một bằng chứng khá phức tạp cho thấy ví dụ này,  $N$  lần truy cập mất tổng thời gian là  $O(N)$ .) Những số liệu này làm nổi bật tính chất cơ bản và quan trọng của cây gỗ. Khi nào đường dẫn truy cập dài, do đó dẫn đến thời gian tìm kiếm lâu hơn bình thường, các vòng quay có xu hướng để tốt cho các hoạt động sau này. Khi quyền truy cập rẻ, các vòng quay không tốt và có thể là xáu. Trường hợp cực đoan là cây ban đầu được hình thành bởi các phép chèn. Tất cả các phụ trang là các hoạt động thời gian liên tục dẫn đến một cây ban đầu xáu. Tại thời điểm đó, chúng tôi đã một cái cây rất xáu, nhưng chúng tôi đã chạy trước thời hạn và được đền bù ít hơn tổng thời gian chạy. Sau đó, một vài lần truy cập thực sự khủng khiếp để lại một cái cây gần như cân bằng, nhưng cái giá phải trả là chúng tôi phải trả lại một phần thời gian đã tiết kiệm được.



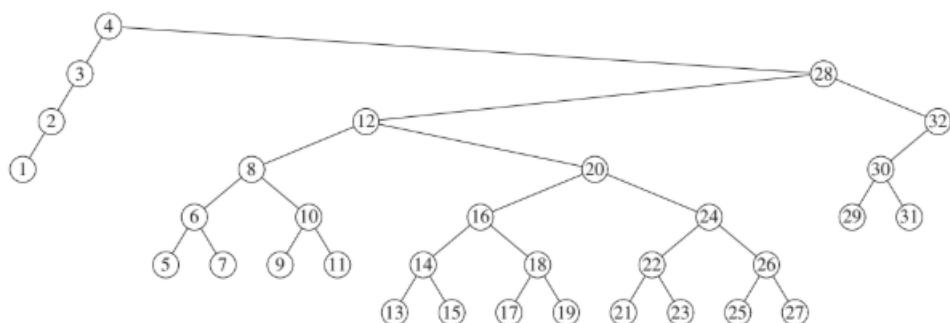
Hình 2.37 Kết quả xuất hiện tại nút 1 một cây của tất cả các con bên trái



Hình 2.38 Kết quả hiển thị cây trước đó tại nút 2



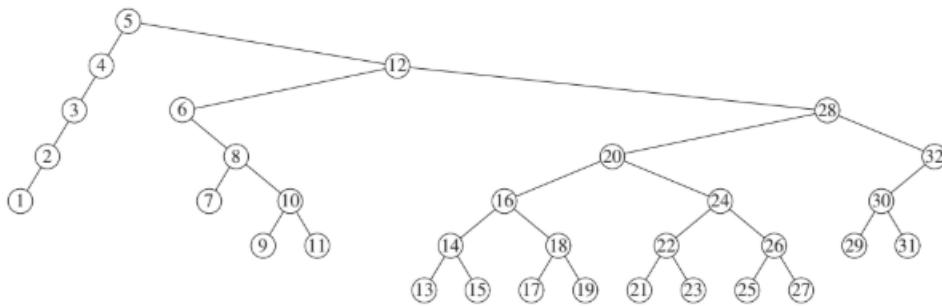
Hình 2.39 Kết quả hiển thị cây trước đó tại nút 3



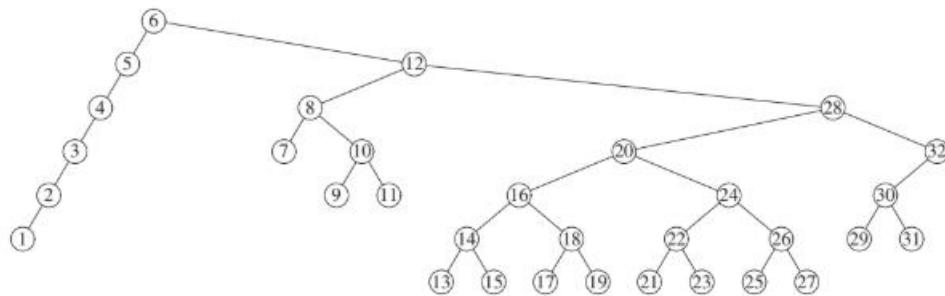
Hình 2.40 Kết quả hiển thị cây trước đó tại nút 4

Chúng ta có thể thực hiện xóa bằng cách truy cập vào nút cần xóa. Điều này đặt nút ở nguồn gốc. Nếu nó bị xóa, chúng ta nhận được hai cây con TL và TR (trái và phải). Nếu chúng tôi tìm thấy lớn nhất phần tử trong TL (dễ dàng),

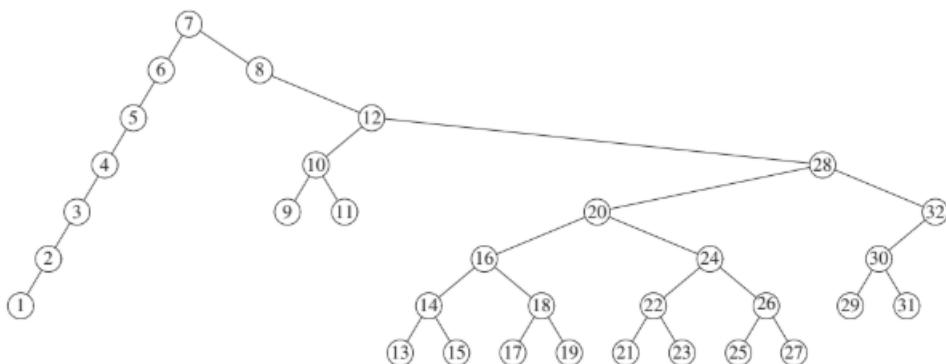
sau đó phần tử này được xoay đến gốc của TL và TL sẽ bây giờ có một gốc không có con bên phải. Chúng tôi có thể hoàn thành việc xóa bằng cách làm cho TR trở thành node con phù hợp.



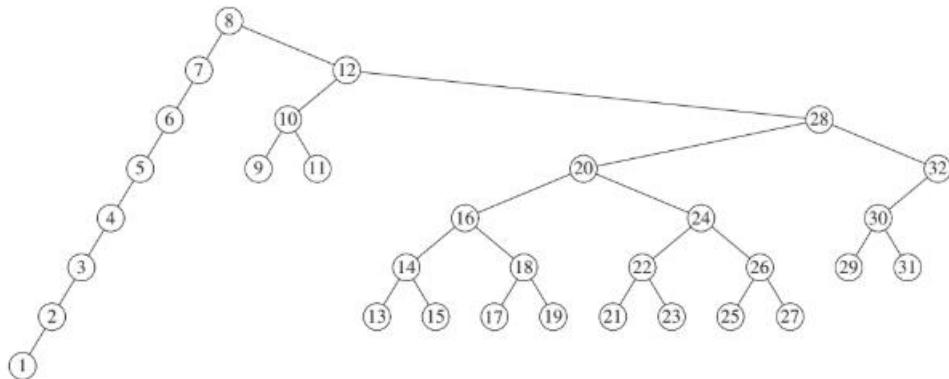
Hình 2.41 Kết quả hiển thị cây trước đó tại nút 5



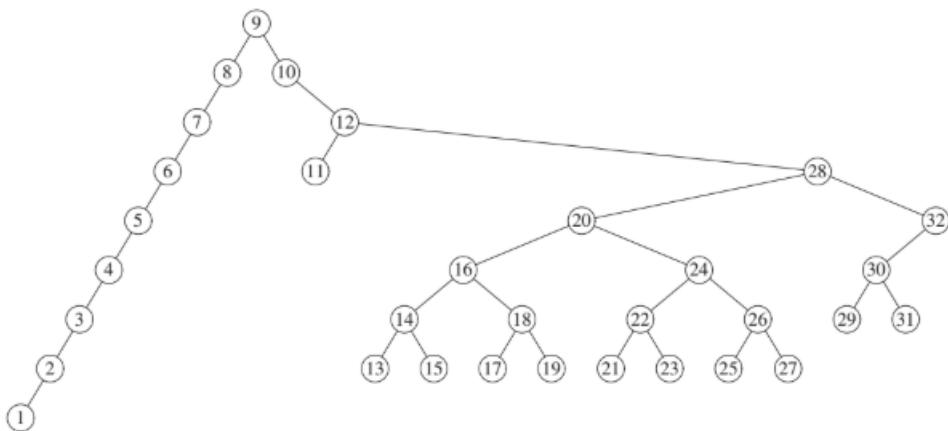
Hình 2.42 Kết quả hiển thị cây trước đó tại nút 6



Hình 2.43 Kết quả hiển thị cây trước đó tại nút 7



Hình 2.44 Kết quả hiển thị cây trước đó tại nút 8



Hình 2.45 Kết quả hiển thị cây trước đó tại nút 9

Việc phân tích cây lớp phủ rất khó, vì nó phải tính đến thay đổi cấu trúc của cây. Mặt khác, splay tree được lập trình đơn giản hơn nhiều so với hầu hết các cây tìm kiếm cân bằng, vì có ít trường hợp cần xem xét hơn và không có số dư thông tin cần duy trì. Một số bằng chứng thực nghiệm cho thấy rằng điều này chuyển thành nhanh hơn trong thực tế, mặc dù trường hợp này vẫn chưa hoàn thành. Cuối cùng, chúng tôi chỉ ra rằng có một số biến thể của cây splay có thể hoạt động tốt hơn trong thực tế.

## 2.6 Duyệt cây

Do thông tin đặt hàng trong cây tìm kiếm nhị phân, nên rất đơn giản để liệt kê tất cả các mục theo thứ tự đã sắp xếp. Hàm đệ quy trong Hình 2.46 hoạt động thực sự.

Thuyết phục bản thân rằng chức năng này hoạt động. Như chúng ta đã thấy trước đây, loại thói quen này khi áp dụng cho cây được biết đến như là một đường đi ngang nhỏ hơn (có ý nghĩa, vì nó liệt kê các mặt hàng theo thứ tự). Chiến lược chung của một trình duyệt nhỏ hơn là xử lý cây con bên trái đầu tiên, sau đó thực hiện xử lý tại nút hiện tại, và cuối cùng xử lý cây con bên phải. Các phần thú vị về thuật toán này, ngoài tính đơn giản của nó, là tổng thời gian chạy là  $O(N)$ . Điều này là do có công việc liên tục được thực hiện tại mọi nút trong cây. Mỗi nút được truy cập một lần và công việc được thực hiện ở mỗi nút đang thử nghiệm với `nullptr`, thiết lập hai lệnh gọi hàm và thực hiện một câu lệnh đầu ra. Vì phải làm việc liên tục mỗi nút và  $N$  nút, thời gian chạy là  $O(N)$ .

Đôi khi chúng ta cần xử lý cả hai cây con trước khi có thể xử lý một nút. Đối với Ví dụ, để tính toán chiều cao của một nút, chúng ta cần biết chiều cao của các cây con Đầu tiên. Đoạn mã trong Hình 2.47 tính toán điều này. Vì nó luôn luôn là một ý tưởng hay để kiểm tra các trường hợp đặc biệt — và rất quan trọng khi liên quan đến đệ quy — lưu ý rằng quy trình sẽ khai báo chiều cao của một chiếc lá bằng 0, điều này đúng. Thứ tự truyền tải chung này, mà chúng tôi cũng đã từng thấy trước đây, được biết đến như là một trình duyệt bưu điện. Một lần nữa, tổng thời gian chạy là  $O(N)$ , bởi vì công việc không đổi được thực hiện tại mỗi nút.

```

1  /**
2   * Print the tree contents in sorted order.
3   */
4 void printTree( ostream & out = cout ) const
5 {
6     if( isEmpty( ) )
7         out << "Empty tree" << endl;
8     else
9         printTree( root, out );
10 }
11
12 /**
13  * Internal method to print a subtree rooted at t in sorted order.
14  */
15 void printTree( BinaryNode *t, ostream & out ) const
16 {
17     if( t != nullptr )
18     {
19         printTree( t->left, out );
20         out << t->element << endl;
21         printTree( t->right, out );
22     }
23 }

```

Hình 2.46 Quy trình in cây tìm kiếm nhị phân theo thứ tự

```

1 /**
2  * Internal method to compute the height of a subtree rooted at t.
3  */
4 int height( BinaryNode *t )
5 {
6     if( t == nullptr )
7         return -1;
8     else
9         return 1 + max( height( t->left ), height( t->right ) );
10 }

```

Hình 2.47 Quy trình tính toán chiều cao của một cái cây bằng cách sử dụng phương thức truyền qua thứ tự bưu điện

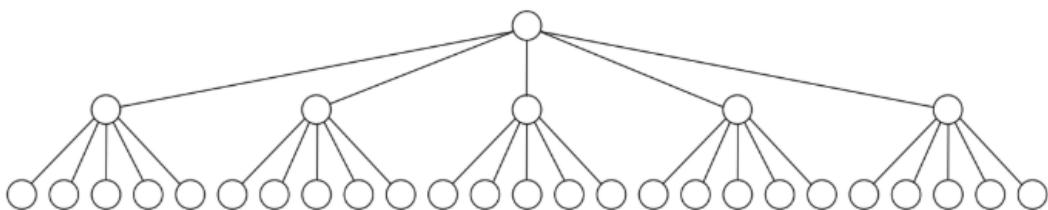
Sơ đồ truyền tải phổ biến thứ ba mà chúng tôi đã thấy là đặt hàng trước truyền tải. Đây, nút được xử lý trước nút con. Điều này có thể hữu ích, chẳng hạn, nếu bạn muốn gắn nhãn mỗi nút với độ sâu của nó. Ý tưởng chung trong

tất cả các quy trình này là bạn xử lý trường hợp nullptr trước và rồi phần còn lại. Chú ý thiếu các biến ngoại lai. Các quy trình này chỉ truyền con trỏ đến nút gốc của cây con và không khai báo hoặc chuyển bất kỳ biến phụ nào. Nhiều hơn làm gọn mã, càng ít có khả năng xuất hiện lỗi ngớ ngẩn. Thứ tư, ít được sử dụng hơn, traversal (mà chúng ta chưa thấy) là trình tự cấp. Trong một trình tự cấp, tất cả các nút ở độ sâu d đều được xử lý trước bất kỳ nút nào ở độ sâu  $d + 1$ . Duyệt theo thứ tự mức khác với các lần truyền khác ở chỗ nó không được thực hiện một cách đệ quy; một hàng đợi được sử dụng, thay vào đó của ngăn xếp đệ quy ngũ ý.

## 2.7 B - cây

Cho đến nay, chúng tôi đã giả định rằng chúng tôi có thể lưu trữ toàn bộ cấu trúc dữ liệu trong bộ nhớ chính của một máy tính. Tuy nhiên, giả sử rằng chúng ta có nhiều dữ liệu hơn mức có thể chứa trong bộ nhớ chính, và do đó, phải có cấu trúc dữ liệu nằm trên đĩa. Khi điều này xảy ra, các quy tắc của thay đổi trò chơi, vì mô hình Big-Oh không còn có ý nghĩa. Vấn đề là một phân tích Big-Oh giả định rằng tất cả các hoạt động đều bình đẳng. Tuy nhiên, điều này không đúng, đặc biệt khi có liên quan đến I / O đĩa. Máy tính hiện đại thực thi hàng tỷ hướng dẫn mỗi giây. Điều đó khá nhanh, chủ yếu là do tốc độ phụ thuộc phần lớn vào về tính chất điện. Mặt khác, đĩa là cơ học. Tốc độ của nó phụ thuộc phần lớn vào thời gian cần thiết để quay đĩa và di chuyển đầu đĩa. Nhiều đĩa quay ở tốc độ 7.200 RPM. Như vậy, trong 1 phút, nó tạo ra 7.200 vòng quay; do đó, một cuộc cách mạng xảy ra trong 1/120 của một giây hoặc 8,3 mili giây. Trung bình, chúng tôi có thể mong đợi rằng chúng tôi phải quay một nửa đĩa để tìm thấy những gì chúng tôi đang tìm kiếm, nhưng điều này được bù đắp bằng thời gian di chuyển đầu đĩa, vì vậy chúng tôi nhận được thời gian truy cập là 8,3 ms. (Đây là một ước tính rất từ thiện; thời gian truy cập 9–11 mili giây phổ biến hơn.) Do đó, chúng tôi có thể thực hiện khoảng 120 lần truy cập đĩa mỗi giây. Điều này nghe có vẻ khá tốt, cho đến khi chúng tôi so sánh nó với tốc độ bộ xử lý. Những gì chúng tôi có là hàng tỷ lệnh tương đương với 120 lần truy cập đĩa. Tất nhiên, mọi thứ ở đây là một tính toán, nhưng tốc độ tương đối rõ ràng: Truy cập đĩa cực kỳ đắt. Hơn nữa, tốc độ bộ xử lý đang tăng với tốc độ nhanh hơn nhiều so với tốc độ đĩa (nó là kích thước đĩa đang tăng lên khá nhanh). Vì vậy, chúng tôi sẵn sàng thực hiện nhiều phép tính chỉ để lưu một quyền truy cập đĩa. Trong hầu hết

các trường hợp, số lần truy cập đĩa sẽ chi phí thời gian chạy. Do đó, nếu chúng ta giảm một nửa số lần truy cập đĩa, việc chạy thời gian sẽ giảm đi một nửa. Đây là cách cây tìm kiếm điển hình hoạt động trên đĩa: Giả sử chúng ta muốn truy cập vào hồ sơ lái xe cho công dân ở bang Florida. Chúng tôi giả định rằng chúng tôi có 10.000.000 các mục, mỗi khóa là 32 byte (đại diện cho một tên) và một bản ghi là 256 byte. Chúng tôi giả sử điều này không phù hợp với bộ nhớ chính và chúng tôi là 1 trong 20 người dùng trên hệ thống (vì vậy chúng tôi có 1/20 tài nguyên). Vì vậy, trong 1 giây, chúng tôi có thể thực hiện hàng triệu lệnh hoặc thực hiện sáu lần truy cập đĩa. Cây tìm kiếm nhị phân không cân bằng là một thảm họa. Trong trường hợp xấu nhất, nó có độ sâu tuyến tính và do đó có thể yêu cầu 10.000.000 quyền truy cập đĩa. Trung bình, một tìm kiếm thành công sẽ yêu cầu truy cập đĩa  $1,38 \log N$  và kể từ bản ghi  $10000000 \approx 24$ , một tìm kiếm trung bình sẽ yêu cầu 32 lần truy cập đĩa, hoặc 5 giây. Trong một cây được xây dựng ngẫu nhiên điển hình, chúng tôi mong đợi rằng một vài nút sâu hơn ba lần; chúng sẽ yêu cầu khoảng 100 quyền truy cập đĩa, hoặc 16 giây. Một cây AVL có phần tốt hơn. Trường hợp xấu nhất là  $1,44 \log N$  khó có thể xảy ra, và trường hợp điển hình là rất gần với  $\log N$ .

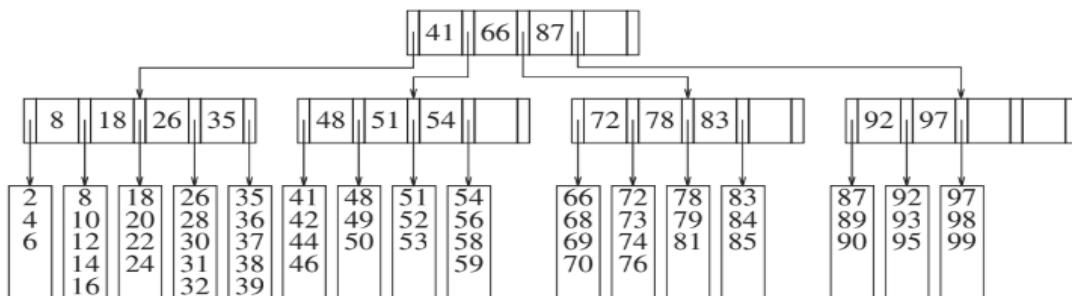


Hình 2.48 Cây 5 ary gồm 31 nút chỉ có ba cấp

Chúng tôi muốn giảm số lần truy cập đĩa xuống một hằng số rất nhỏ, chẳng hạn như ba hoặc bốn. Chúng tôi sẵn sàng viết mã phức tạp để thực hiện việc này, vì hướng dẫn máy về cơ bản là miễn phí, miễn là chúng ta không vô lý một cách lố bịch. Nó có lẽ nên rõ ràng rằng cây tìm kiếm nhị phân sẽ không hoạt động, vì cây AVL điển hình gần với Chiều cao. Chúng tôi không thể đi xuống dưới  $\log N$  bằng cách sử dụng cây tìm kiếm nhị phân. Giải pháp là trực quan đơn giản: Nếu chúng ta có nhiều nhánh hơn, chúng ta có ít chiều cao hơn. Vì vậy, trong khi một nhị phân hoàn hảo cây 31 nút có năm cấp, cây 5 ary gồm 31 nút chỉ có ba cấp, như được minh họa trong Hình 2.48. Cây tìm

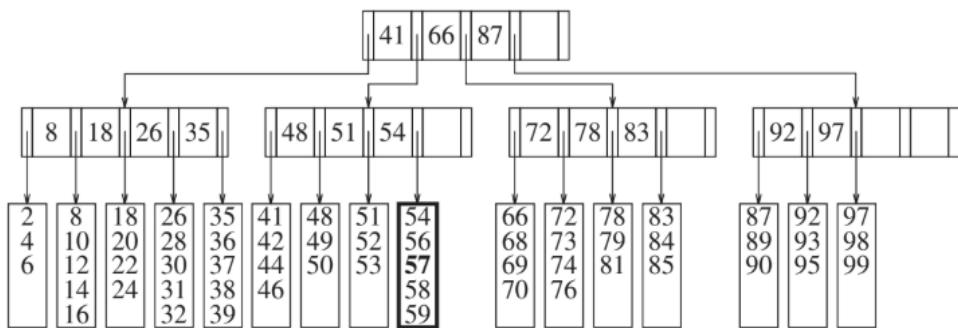
kiếm M-ary cho phép phân nhánh M-way. Khi phân nhánh tăng lên, độ sâu giảm dần. Trong khi một cây nhị phân hoàn chỉnh có chiều cao gần bằng  $\log_2 N$ , Cây hoàn chỉnh M có chiều cao xấp xỉ  $\log_M N$ . Chúng ta có thể tạo cây tìm kiếm M-ary theo cách giống như cây tìm kiếm nhị phân. Trong một cây tìm kiếm nhị phân, chúng ta cần một khóa để quyết định chọn nhánh nào trong hai nhánh. Trong một M-ary cây tìm kiếm, chúng ta cần M-1 phím để quyết định nhánh nào sẽ lấy. Để thực hiện kế hoạch này hiệu quả trong trường hợp xấu nhất, chúng tôi cần đảm bảo rằng cây tìm kiếm M-ary được cân bằng trong một số đường. Nếu không, giống như cây tìm kiếm nhị phân, nó có thể biến thành danh sách liên kết. Thực ra, chúng tôi muốn một điều kiện cân bằng hạn chế hơn. Đó là, chúng tôi không muốn một M-ary cây tìm kiếm biến đổi thành cây tìm kiếm nhị phân, bởi vì khi đó chúng ta sẽ bị mắc kẹt với quyền truy cập  $\log N$ . Một cách để thực hiện điều này là sử dụng B-tree. B-tree cơ bản được mô tả ở đây. Nhiều các biến thể và cải tiến đã được biết đến và việc triển khai hơi phức tạp vì có khá nhiều trường hợp. Tuy nhiên, có thể dễ dàng nhận thấy rằng, về nguyên tắc, cây B đảm bảo chỉ một số truy cập đĩa. Cây B bậc M là cây M có các tính chất sau:

1. Các mục dữ liệu được lưu trữ tại các lá.
2. Các nút không ở trang chính lưu trữ tối đa  $M - 1$  khóa để hướng dẫn tìm kiếm; chìa khóa tối đại diện cho khóa nhỏ nhất trong cây con  $i + 1$ .
3. Gốc là một chiếc lá hoặc có từ hai đến  $M$  con.
4. Tất cả các nút không phải lá (trừ nút gốc) có từ  $M / 2$  đến  $M$  con.
5. Tất cả các lá ở cùng độ sâu và có giữa các mục dữ liệu  $L / 2$  và  $L$ , đối với một số  $L$  (việc xác định  $L$  được mô tả ngay sau đây).



Hình 2.49 Cây B bậc 5

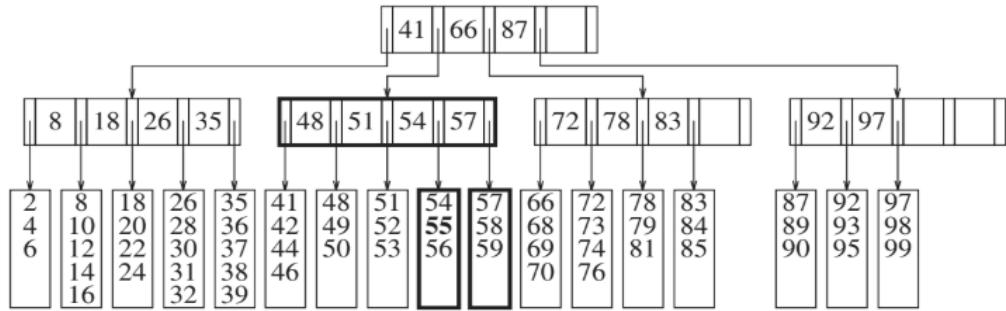
Ví dụ về cây B bậc 5 được thể hiện trong Hình 2.49. Lưu ý rằng tất cả các nút không phải trang chính có từ ba đến năm con (và do đó từ hai đến bốn khóa); rẽ có thể có thể chỉ có hai con. Ở đây, chúng ta có  $L = 5$ . Điều xảy ra rằng  $L$  và  $M$  là tương tự trong ví dụ này, nhưng điều này là không cần thiết. Vì  $L$  là 5 nên mỗi lá có từ ba và năm mục dữ liệu. Yêu cầu các nút phải đầy một nửa đảm bảo rằng cây B không suy biến thành cây nhị phân đơn giản. Mặc dù có nhiều định nghĩa khác nhau về cây B thay đổi cấu trúc này, chủ yếu là theo những cách nhỏ, định nghĩa này là một trong những hình thức phổ biến. Mỗi nút đại diện cho một khối đĩa, vì vậy chúng tôi chọn  $M$  và  $L$  trên cơ sở kích thước của các mục đang được lưu trữ. Ví dụ, giả sử một khối chứa 8.192 byte. Trong của chúng tôi Ví dụ ở Florida, mỗi khóa sử dụng 32 byte. Trong cây B thứ tự  $M$ , chúng ta sẽ có  $M-1$  khóa, với tổng số  $32M - 32$  byte, cộng với  $M$  nhánh. Vì mỗi nhánh về cơ bản là một số của một khối đĩa khác, chúng ta có thể giả định rằng một nhánh là 4 byte. Do đó, các chi nhánh sử dụng 4M byte. Do đó, tổng yêu cầu bộ nhớ cho một nút không phải là  $36M-32$ . Giá trị lớn nhất của  $M$  mà giá trị này không lớn hơn 8.192 là 228. Vì vậy, chúng ta sẽ chọn  $M = 228$ . Vì mỗi bản ghi dữ liệu là 256 byte, chúng tôi sẽ có thể phù hợp với 32 bản ghi trong một khối. Vì vậy, chúng tôi sẽ chọn  $L = 32$ . Chúng tôi đảm bảo rằng mỗi lá có từ 16 đến 32 bản ghi dữ liệu và rằng mỗi nút bên trong (trừ nút gốc) phân nhánh theo ít nhất 114 cách. Vì có 10.000.000 bản ghi, nhiều nhất là 625.000 lá. Do đó, trong trường hợp xấu nhất, lá sẽ ở cấp độ 4. Nói một cách cụ thể hơn, số lần truy cập trong trường hợp xấu nhất là được cho bởi khoảng  $\log M / 2 N$ , cho hoặc lấy 1. (Ví dụ: gốc và mức tiếp theo có thể được lưu vào bộ nhớ đệm trong bộ nhớ chính, do đó về lâu dài, sẽ cần truy cập đĩa chỉ dành cho cấp độ 3 trở lên.) Vấn đề còn lại là làm thế nào để thêm bớt các mục từ B-tree. Những ý tưởng liên quan được phác thảo tiếp theo. Lưu ý rằng nhiều chủ đề đã thấy trước đó tái diễn. Chúng tôi bắt đầu bằng cách kiểm tra sự chèn. Giả sử chúng ta muốn chèn 57 vào cây B trong Hình 2.50. Một cuộc tìm kiếm dưới gốc cây cho thấy nó chưa có trên cây. Chúng tôi có thể thêm nó đến lá như một mục thứ năm. Lưu ý rằng chúng tôi có thể phải sắp xếp lại tất cả dữ liệu trong lá để làm cái này. Tuy nhiên, chi phí của việc này là không đáng kể khi so sánh với đĩa truy cập, trong trường hợp này cũng bao gồm ghi đĩa. Tất nhiên, điều đó tương đối không đau, bởi vì chiếc lá chưa đầy. Giả sử bây giờ chúng ta muốn chèn 55. Hình 4.51 cho thấy một vấn đề: nơi mà lá 55 muốn tới đã đầy. Giải pháp rất đơn giản:



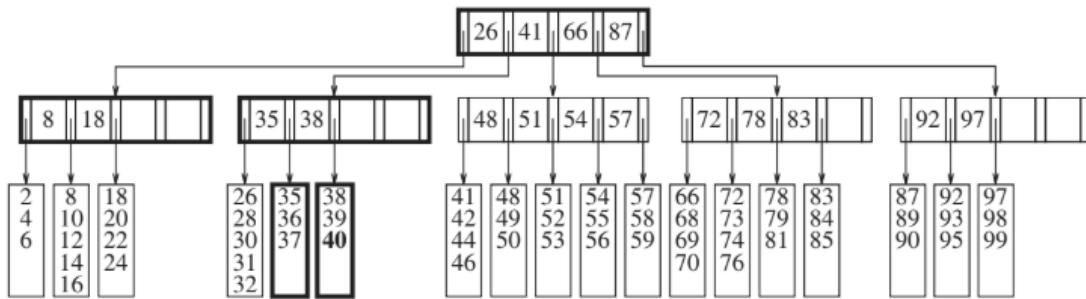
Hình 2.50 Cây B sau khi chèn 57 vào cây trong hình 2.49

Vì hiện tại chúng tôi có  $L + 1$  mục, chúng tôi chia chúng thành hai lá, cả hai đều được đảm bảo có số lượng bản ghi dữ liệu tối thiểu cần thiết. Chúng tôi tạo thành hai lá với ba mục mỗi loại. Cần có hai quyền truy cập đĩa để ghi các lá này, và cần có quyền truy cập đĩa thứ ba để cập nhật phụ huynh. Lưu ý rằng trong cha, cả khóa và nhánh đều thay đổi, nhưng chúng làm như vậy theo cách có kiểm soát và dễ dàng tính toán được. Cây B kết quả được thể hiện trong Hình 2.37. Mặc dù việc chia tách các nút tốn nhiều thời gian vì nó yêu cầu ít nhất hai lần ghi đĩa bổ sung, đây là một trường hợp tương đối hiếm. Ví dụ, nếu  $L$  là 32, thì khi một nút được tách ra, hai lá có 16 và 17 mục tương ứng sẽ được tạo ra. Đối với lá có 17 mục, chúng ta có thể thực hiện thêm 15 lần chèn nữa mà không cần chia thêm lần nữa. Nói một cách khác, đối với mỗi lần phân chia, có khoảng  $L / 2$  không có chỗ.

Việc tách nút trong ví dụ trước đã hoạt động vì nút cha không có phần bổ sung đầy đủ của nút con. Nhưng điều gì sẽ xảy ra nếu nó xảy ra? Ví dụ, giả sử rằng chúng ta chèn 40 vào cây B trong Hình 2.51. Chúng ta phải tách lá chứa các phím từ 35 đến 39, và bây giờ là 40, thành hai lá. Nhưng làm điều này sẽ cung cấp cho cha mẹ sáu đứa con, và nó chỉ được phép có năm. Giải pháp là tách cha mẹ. Kết quả của việc này được thể hiện trong Hình 2.52. Khi khóa chính được tách ra, chúng ta phải cập nhật giá trị của các khóa và cũng là khóa chính của khóa gốc, do đó phát sinh thêm hai lần ghi đĩa (do đó, việc chèn này tốn năm lần ghi đĩa). Tuy nhiên, một lần nữa, các khóa thay đổi theo cách rất được kiểm soát, mặc dù mã chắc chắn không đơn giản vì nhiều trường hợp.



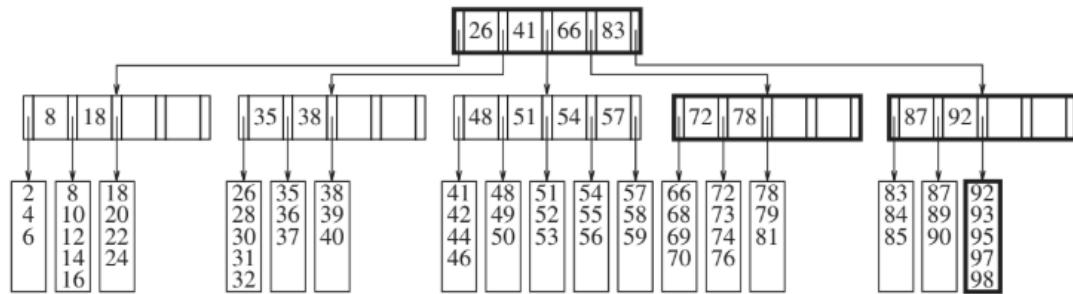
Hình 2.51 Chèn 55 vào cây B trong hình 2.50 gây ra hiện tượng tách thành hai lá



Hình 2.52 Chèn 40 vào cây B trong Hình 2.51 gây ra sự phân tách thành hai lá và sau đó là sự phân tách của nút cha

Khi một nút không phải được tách ra, như trường hợp ở đây, nút cha của nó sẽ nhận được một nút con. Chuyện gì xảy ra nếu cha mẹ đã đạt đến giới hạn của con cái? Sau đó, chúng tôi tiếp tục tách các nút lên cây cho đến khi chúng tôi tìm thấy cây bố mẹ không cần tách hoặc chúng tôi đạt được nguồn gốc. Nếu chúng ta tách gốc, thì chúng ta có hai gốc. Rõ ràng, điều này là không thể chấp nhận được, nhưng chúng tôi có thể tạo ra một rẽ mới có các rẽ tách làm hai con của nó. Đây là lý do tại sao gốc là được miễn trừ tối thiểu hai con đặc biệt. Đó cũng là cách duy nhất mà cây B tăng chiều cao. Không cần phải nói, tách toàn bộ đến tận gốc là một điều đặc biệt hiếm biến cố. Điều này là do một cây có bốn cấp chỉ ra rằng rẽ đã bị chia ba lần trong toàn bộ chuỗi chèn (giả sử không có lần xóa nào xảy ra). Trên thực tế, việc tách bất kỳ nút không phải lá nào cũng khá hiếm. Có nhiều cách khác để xử lý tình trạng trẻ bị tràn sữa. Một kỹ thuật là đặt một đứa trẻ được nhận làm con nuôi nên hàng xóm có chỗ. Để chèn 29 vào cây B trong Ví dụ, trong hình 2.52, chúng ta có thể tạo khoảng trống bằng cách chuyển 32 sang lá tiếp theo. Công nghệ

này- nique yêu cầu một sửa đổi của cha mẹ, vì các khóa bị ảnh hưởng. Tuy nhiên, nó có xu hướng để giữ cho các nút đầy đủ hơn và tiết kiệm dung lượng về lâu dài. Chúng tôi có thể thực hiện xóa bằng cách tìm mục cần xóa và sau đó xóa- ăn nó. Vấn đề là nếu chiếc lá nó nằm trong có số lượng mục dữ liệu tối thiểu, thì nó bây giờ dưới mức tối thiểu. Chúng tôi có thể khắc phục tình trạng này bằng cách sử dụng một mặt hàng lân cận, nếu người hàng xóm không phải là chính nó ở mức tối thiểu. Nếu đúng thì chúng ta có thể kết hợp với hàng xóm để tạo thành một lá đầy đủ. Thật không may, điều này có nghĩa là cha mẹ đã mất một đứa trẻ. Nếu điều này gây ra cha mẹ giảm xuống dưới mức tối thiểu của nó, sau đó nó tuân theo cùng một chiến lược. Quá trình này có thể thấm vào tận gốc rễ. Gốc không chỉ có một con (và ngay cả



Hình 2.53 Cây B sau khi xóa 99 khỏi cây B trong Hình 2.52

khi điều này được cho phép, nó sẽ là ngắn). Nếu một gốc còn lại một con là kết quả của quá trình nhận nuôi, thì chúng ta loại bỏ gốc và đặt con của nó làm gốc mới của cây. Đây là cách duy nhất để cây B giảm chiều cao. Ví dụ, giả sử chúng ta muốn loại bỏ 99 khỏi cây B trong Hình 2.52. Vì lá chỉ có hai mục và người hàng xóm của nó đã có tối thiểu ba mục, chúng tôi kết hợp các mục thành một lá mới gồm năm mục. Kết quả là cha mẹ chỉ có hai con. Tuy nhiên, nó có thể nhận con nuôi từ một người hàng xóm, vì người hàng xóm có bốn đứa con. Kết quả là cả hai có ba người con. Kết quả như hình 2.53.

## 2.8 Sets và Maps trong thư viện Chuẩn

Các vùng chứa STL được thảo luận trong phần 1 — cụ thể là vectơ và danh sách — không hiệu quả đối với đang tìm kiếm. Do đó, STL cung cấp hai vùng chứa bổ sung, set và map, đảm bảo chi phí logarit cho các hoạt động cơ bản như chèn, xóa và tìm kiếm.

### 2.8.1 Sets

**Set** là một vùng chứa có thứ tự không cho phép trùng lặp. Nhiều thành ngữ được sử dụng để truy cập các mục trong vectơ và danh sách cũng hoạt động cho một tập hợp. Cụ thể, lồng trong tập hợp là các loại trình lặp và `const_iterator` cho phép duyệt qua tập hợp và một số phương thức từ vectơ và danh sách được đặt tên giống hệt nhau trong `set`, bao gồm bắt đầu, kết thúc, kích thước và trống. Mẫu hàm in được mô tả trong Hình 3.6 sẽ hoạt động nếu được thông qua một bộ. Các hoạt động duy nhất được yêu cầu bởi `set` là khả năng chèn, loại bỏ và thực hiện tìm kiếm cơ bản (hiệu quả). Quy trình chèn được đặt tên phù hợp. Tuy nhiên, vì một tập hợp không cho phép trùng lặp cates, có thể xảy ra lỗi chèn. Do đó, chúng tôi muốn kiểu trả về có thể chỉ ra điều này bằng một biến Boolean. Tuy nhiên, `insert` có kiểu trả về phức tạp hơn một bool. Điều này là do `insert` cũng trả về một trình lặp đại diện cho `x` ở đâu khi chèn lợi nhuận. Trình lặp này đại diện cho mục mới được chèn hoặc mục hiện có điều đó khiến cho phần chèn bị lỗi, và nó rất hữu ích, vì biết vị trí của mục có thể làm cho việc xóa nó hiệu quả hơn bằng cách tránh tìm kiếm và truy cập trực tiếp vào nút chứa vật phẩm. STL xác định một khuôn mẫu lớp được gọi là cặp ít hơn một cấu trúc với thành viên thứ nhất và thứ hai để truy cập hai mục trong cặp. Có hai khác nhau chèn các thói quen: cặp `<iterator, bool>` `insert (const Object & x);` cặp `<biến lặp, bool>` chèn (gọi ý trình lặp, `const Object & x`); Chèn một tham số hoạt động như mô tả ở trên. Chèn hai tham số cho phép đặc điểm kỹ thuật của một gợi ý, đại diện cho vị trí mà `x` sẽ đến. Nếu gợi ý là chính xác, việc chèn nhanh, thường là  $O(1)$ . Nếu không, việc chèn được thực hiện bằng cách sử dụng bình thường thuật toán chèn và thực hiện tương đối với phép chèn một tham số. Ví dụ, mã sau có thể nhanh hơn bằng cách sử dụng `insert` hai tham số thay vì `insert` một tham số:

```

set<int> s;
for( int i = 0; i < 1000000; ++i )
    s.insert( s.end( ), i );

```

There are several versions of `erase`:

```

int erase( const Object & x );
iterator erase( iterator itr );
iterator erase( iterator start, iterator end );

```

Việc xóa một tham số đầu tiên sẽ xóa x (nếu được tìm thấy) và trả về số lượng mục đã thực sự bị xóa, rõ ràng là 0 hoặc 1. Lần xóa một tham số thứ hai hoạt động giống như trong vector và danh sách. Nó loại bỏ đối tượng tại vị trí được cung cấp bởi trình lặp, trả về một trình lặp đại diện cho phần tử sau nó ngay lập tức trước khi gọi để xóa và làm mất hiệu lực của nó, thứ trở nên cũ. Hai tham số xóa hoạt động giống như trong vector hoặc danh sách, xóa tất cả các mục bắt đầu từ đầu, lên đến nhưng không bao gồm mục ở cuối. Để tìm kiếm, thay vì một quy trình chứa trả về một biến Boolean, tập hợp cung cấp một quy trình tìm kiếm trả về một trình lặp đại diện cho vị trí của mục (hoặc điểm đánh dấu nếu tìm kiếm không thành công). Điều này cung cấp thêm thông tin đáng kể, miễn phí trong thời gian chạy. Dấu hiệu của tìm là:

```
iterator find( const Object & x ) const;
```

Theo mặc định, việc sắp xếp sử dụng đối tượng hàm `<Object>` ít hơn, đối tượng này tự nó được triển khai bằng cách gọi toán tử `<cho` Đối tượng. Một thứ tự thay thế có thể được chỉ định bởi `instan-` định vị mẫu thiết lập với một kiểu đối tượng hàm. Trong đoạn mã sau, tập hợp s có kích thước 1:

```

set<string,CaseInsensitiveCompare> s;
s.insert( "Hello" ); s.insert( "HeLLo" );
cout << "The size is: " << s.size( ) << endl;

```

## 2.8.2 Maps

Map được sử dụng để lưu trữ một tập hợp các mục nhập có thứ tự bao gồm các khóa và giá trị của chúng. Các khóa phải là duy nhất, nhưng một số khóa có thể ánh xạ đến các giá trị giống nhau. Vì vậy, các giá trị không cần là duy

nhất. Các phím trong Map được duy trì theo thứ tự được sắp xếp hợp lý. Map hoạt động giống như một tập hợp được khởi tạo với một cặp, có chức năng so sánh chỉ để cập đến khóa.<sup>5</sup> Vì vậy, nó hỗ trợ bắt đầu, kết thúc, kích thước và trống, nhưng dưới- trình lặp nằm là một cặp khóa-giá trị. Nói cách khác, đối với một trình lặp itr, \* itr thuộc loại cặp <KeyType, ValueType>. Map cũng hỗ trợ chèn, tìm và xóa. Để chèn, một phải cung cấp một cặp đổi tương <KeyType, ValueType>. Mặc dù tìm chỉ yêu cầu một khóa, trình lặp nó trả về một cặp tham chiếu. Chỉ sử dụng các thao tác này thường không đáng giá bởi vì hành lý cú pháp có thể đắt tiền. May mắn thay, Map có một thao tác bổ sung quan trọng mang lại cú pháp đơn giản. Các toán tử lập chỉ mục mảng được nạp chồng cho các Map như sau:

```
ValueType & operator[] ( const KeyType & key );
```

Ngữ nghĩa của toán tử [] như sau. Nếu khóa có trong Map, một tham chiếu đến giá trị tương ứng được trả về. Nếu khóa không có trong Map, nó sẽ được chèn với một khóa mặc định giá trị vào Map và sau đó một tham chiếu đến giá trị mặc định đã chèn được trả về. Mặc định giá trị thu được bằng cách áp dụng hàm tạo tham số không hoặc bằng 0 đối với giá trị nguyên thủy các loại. Các ngữ nghĩa này không cho phép phiên bản truy cập của toán tử [], vì vậy toán tử [] không thể được sử dụng trên một Map không đổi. Ví dụ: nếu một Map được truyền bằng tham chiếu không đổi, bên trong quy trình, toán tử [] không sử dụng được. Đoạn mã trong Hình 4.68 minh họa hai kỹ thuật để truy cập các mục trong Map. Đầu tiên, hãy quan sát rằng ở dòng 3, phía bên trái gọi toán tử [], do đó chèn "Pat" và nhân đôi của giá trị 0 vào Map và trả về một tham chiếu cho nhân đôi đó. Sau đó, bài tập thay đổi gấp đôi bên trong Map thành 75000. Dòng 4 xuất ra 75000. Thật không may, dòng 5 chèn "Jan" và mức lương 0,0 vào Map và sau đó in nó. Điều này có thể có hoặc không là điều thích hợp để làm, tùy thuộc vào ứng dụng. Nếu điều quan trọng là phải phân biệt giữa các mục có trong Map và các mục không có trong Map hoặc nếu điều quan trọng là không chèn vào Map (bởi vì nó là bất biến), sau đó một cách tiếp cận thay thế được hiển thị tại các dòng 7 đến 12 có thể được sử dụng. Ở đó chúng tôi thấy một cuộc gọi để tìm. Nếu khóa không được tìm thấy, trình lặp là điểm đánh dấu và có thể được kiểm tra. Nếu khóa được tìm thấy, chúng tôi có thể truy cập mục thứ hai trong cặp được tham

chiếu bởi trình lặp, là giá trị được liên kết với khóa. Chúng tôi cũng có thể gán cho `itr->second` nếu, thay vì một `const_iterator`, `itr` là một trình lặp.

### 2.8.3 Thực hiện của sets và maps

C++ yêu cầu bộ và Map đó hỗ trợ các thao tác chèn, xóa và tìm cơ bản trong logarit thời gian trong trường hợp xấu nhất. Do đó, việc triển khai cơ bản là một cây tìm kiếm nhị phân cân bằng.

```
1 map<string,double> salaries;
2
3     salaries[ "Pat" ] = 75000.00;
4     cout << salaries[ "Pat" ] << endl;
5     cout << salaries[ "Jan" ] << endl;
6
7     map<string,double>::const_iterator itr;
8     itr = salaries.find( "Chris" );
9     if( itr == salaries.end( ) )
10        cout << "Not an employee of this company!" << endl;
11    else
12        cout << itr->second << endl;
```

Hình 2.54 Truy cập các giá trị trong Map

Thông thường, một cây AVL không được sử dụng; thay vào đó là những cây đỏ-đen từ trên xuống, thường được sử dụng trong Phần 12.2. Một vấn đề quan trọng trong việc triển khai tập hợp và Map là cung cấp hỗ trợ cho trình lặp các lớp học. Tất nhiên, bên trong, trình vòng lặp duy trì một con trỏ đến nút "hiện tại" trong sự lặp lại. Phần cứng đang tiến đến nút tiếp theo một cách hiệu quả. Có thể có một số cách giải pháp được liệt kê ở đây:

1. Khi trình lặp được xây dựng, hãy để mỗi trình lặp lưu trữ dưới dạng dữ liệu của nó một mảng chứa các mục đã đặt. Điều này không hiệu quả: Nó khiến bạn không thể triển khai hiệu quả bất kỳ các quy trình trả về một trình lặp sau khi sửa đổi tập hợp, chẳng hạn như một số phiên bản xóa và chèn.
2. Yêu cầu trình lặp duy trì một ngăn xếp lưu trữ các nút trên đường dẫn đến nút hiện tại. Với thông tin này, người ta có thể suy ra nút tiếp theo trong lần lặp, là nút trong cây con bên phải của nút hiện tại có chứa mục tối thiểu hoặc mục gần nhất tổ tiên chứa nút hiện tại trong

cây con bên trái của nó. Điều này làm cho trình lặp hơi lớn và làm cho mã trình vòng lặp vụng về.

3. Có mỗi nút trong cây tìm kiếm lưu trữ cha của nó ngoài các nút con. Các trình vòng lặp không lớn bằng, nhưng hiện tại mỗi nút cần thêm bộ nhớ và mã để lặp vẫn còn vụng về.

4. Yêu cầu mỗi nút duy trì các liên kết bổ sung: một nút tiếp theo nhỏ hơn và một nút tiếp theo lớn hơn nút. Điều này tốn không gian, nhưng việc lặp lại rất đơn giản để thực hiện và dễ dàng duy trì các liên kết này.

5. Duy trì các liên kết bổ sung chỉ cho các nút có nullptr liên kết trái hoặc phải bằng cách sử dụng các biến Boolean bổ sung để cho phép các quy trình cho biết liệu một liên kết bên trái có đang được sử dụng làm liên kết trái cây tìm kiếm nhị phân tiêu chuẩn hoặc liên kết đến nút nhỏ hơn tiếp theo và tương tự cho liên kết bên phải. Ý tưởng này được gọi là cây luồng và được sử dụng trong nhiều triển khai STL.

#### 2.8.4 Ví dụ về sử dụng một số Maps

Nhiều từ tương tự với các từ khác. Ví dụ: bằng cách thay đổi chữ cái đầu tiên, từ rượu vang có thể trở thành bữa tối, tốt, dòng, tôi, chín, thông, hoặc nho. Bằng cách thay đổi chữ cái thứ ba, rượu có thể trở thành rộng, vợ, lau, hoặc dây, trong số những người khác. Bằng cách thay đổi chữ cái thứ tư, rượu vang có thể trở thành gió, cánh, nháy mắt, hoặc chiến thắng, trong số những người khác. Điều này cung cấp 15 từ khác nhau có thể có được bằng cách thay đổi chỉ một chữ cái trong rượu vang. Trên thực tế, có hơn 20 từ khác nhau, một số mờ mịt hơn. Chúng tôi muốn viết một chương trình để tìm tất cả các từ có thể thay đổi thành ít nhất 15 từ khác bằng cách thay thế một ký tự. Chúng tôi giả định rằng chúng tôi có một từ điển bao gồm khoảng 89.000 từ khác nhau với độ dài khác nhau. Phần lớn từ có từ 6 đến 11 ký tự. Bản phân phối bao gồm 8.205 từ sáu chữ cái, 11.989 từ bảy chữ cái, 13.672 từ tám chữ cái, 13.014 từ chín chữ cái, 11.297 từ mười chữ cái và 8.617 từ mười một chữ cái. (Trong thực tế, những từ dễ thay đổi nhất là từ ba, bốn và năm ký tự, nhưng những từ dài hơn là những từ tốn nhiều thời gian kiểm tra.) Chiến lược đơn giản nhất là sử dụng một Map trong đó các khóa là các từ và các giá trị là vector chứa các từ có thể được thay đổi từ khóa bằng một ký tự.

```

1 void printHighChangeables( const map<string,vector<string>> & adjacentWords,
2                               int minWords = 15 )
3 {
4     for( auto & entry : adjacentWords )
5     {
6         const vector<string> & words = entry.second;
7
8         if( words.size( ) >= minWords )
9         {
10            cout << entry.first << " (" << words.size( ) << ")";
11            for( auto & str : words )
12                cout << " " << str;
13            cout << endl;
14        }
15    }
16 }
```

Hình 2.55 Đưa ra một Map chứa các từ làm khóa và vectơ các từ khác nhau về chỉ một ký tự làm giá trị, xuất ra các từ có tối thiểu từ AdWords trở lên có thể đạt được bằng cách thay thế một ký tự.

Quy trình trong Hình 2.55 cho thấy cách Map đồng đều có thể được sử dụng để in các câu trả lời. Mã sử dụng một phạm vi cho vòng lặp để đi qua Map và xem các mục nhập các cặp bao gồm một từ và một vectơ từ. Các tham chiếu không đổi ở dòng 4 và 6 được sử dụng để thay thế các biểu thức phức tạp và tránh tạo các bản sao không cần thiết. Vấn đề chính là làm thế nào để xây dựng Map từ một mảng chứa 89.000 từ ngữ. Quy trình trong Hình 2.56 là một hàm đơn giản để kiểm tra xem hai từ có giống nhau ngoại trừ sự thay thế một ký tự. Chúng tôi có thể sử dụng quy trình để cung cấp thuật toán đơn giản nhất cho việc xây dựng Map, đó là một bài kiểm tra bạo lực của tất cả các cặp từ. Thuật toán này được thể hiện trong Hình 2.57. Nếu chúng tôi tìm thấy một cặp từ chỉ khác nhau ở một ký tự, chúng tôi có thể cập nhật Map tại dòng 12 và 13. Thành ngữ mà chúng tôi đang sử dụng ở dòng 12 là adjWords [str] đại diện cho vectơ của các từ giống với str, ngoại trừ một ký tự. Nếu trước đây chúng ta đã str đã thấy, sau đó nó có trong Map và chúng ta chỉ cần thêm từ mới vào vectơ trong Map và chúng tôi thực hiện điều này bằng cách gọi push\_back. Nếu chúng ta chưa bao giờ nhìn thấy str trước đây, thì hành động của sử dụng toán tử [] đặt nó vào Map, với một vectơ có kích thước 0 và trả về vectơ này, vì vậy push\_back cập nhật vectơ thành kích

thước 1. Nói chung, một thành ngữ siêu bóng bẩy để duy trì một Map trong đó giá trị là một tập hợp. Vấn đề với thuật toán này là nó chậm và mất 97 giây trên com- người đặt. Một cải tiến rõ ràng là tránh so sánh các từ có độ dài khác nhau. Chúng ta có thể thực hiện việc này bằng cách nhóm các từ theo độ dài của chúng và sau đó chạy thuật toán trước đó trên mỗi nhóm riêng biệt. Để làm điều này, chúng ta có thể sử dụng Map thứ hai! Đây là một số nguyên đại diện cho một từ độ dài và giá trị là tập hợp tất cả các từ có độ dài đó. Chúng ta có thể sử dụng một vectơ để lưu trữ từng bộ sưu tập và áp dụng cùng một thành ngữ.

```
1 // Returns true if word1 and word2 are the same length
2 // and differ in only one character.
3 bool oneCharOff( const string & word1, const string & word2 )
4 {
5     if( word1.length( ) != word2.length( ) )
6         return false;
7
8     int diffs = 0;
9
10    for( int i = 0; i < word1.length( ); ++i )
11        if( word1[ i ] != word2[ i ] )
12            if( ++diffs > 1 )
13                return false;
14
15    return diffs == 1;
16 }
```

Hình 2.56 Quy trình kiểm tra xem hai từ chỉ khác nhau một ký tự hay không

```

1 // Computes a map in which the keys are words and values are vectors of words
2 // that differ in only one character from the corresponding key.
3 // Uses a quadratic algorithm.
4 map<string,vector<string>> computeAdjacentWords( const vector<string> & words )
5 {
6     map<string,vector<string>> adjWords;
7
8     for( int i = 0; i < words.size( ); ++i )
9         for( int j = i + 1; j < words.size( ); ++j )
10            if( oneCharOff( words[ i ], words[ j ] ) )
11            {
12                adjWords[ words[ i ] ].push_back( words[ j ] );
13                adjWords[ words[ j ] ].push_back( words[ i ] );
14            }
15
16    return adjWords;
17 }
```

Hình 2.57 Chức năng tính toán một Map có chứa các từ làm khóa và một vectơ từ chỉ khác một ký tự làm giá trị. Phiên bản này chạy trong 1,5 phút trên 89.000 - từ điền từ.

Mã được hiển thị trong Hình 2.58. Hàng 8 hiển thị khai báo cho Map thứ hai, các dòng 11 và 12 điền vào Map, sau đó là một vòng lặp phụ được sử dụng để lặp lại từng nhóm từ. So với thuật toán đầu tiên, thuật toán thứ hai chỉ khó viết hơn một chút và chạy trong 18 giây, hoặc nhanh hơn khoảng sáu lần.

```

1 // Computes a map in which the keys are words and values are vectors of words
2 // that differ in only one character from the corresponding key.
3 // Uses a quadratic algorithm, but speeds things up a little by
4 // maintaining an additional map that groups words by their length.
5 map<string,vector<string>> computeAdjacentWords( const vector<string> & words )
6 {
7     map<string,vector<string>> adjWords;
8     map<int,vector<string>> wordsByLength;
9
10    // Group the words by their length
11    for( auto & thisWord : words )
12        wordsByLength[ thisWord.length( ) ].push_back( thisWord );
13
14    // Work on each group separately
15    for( auto & entry : wordsByLength )
16    {
17        const vector<string> & groupsWords = entry.second;
18
19        for( int i = 0; i < groupsWords.size( ); ++i )
20            for( int j = i + 1; j < groupsWords.size( ); ++j )
21                if( oneCharOff( groupsWords[ i ], groupsWords[ j ] ) )
22                {
23                    adjWords[ groupsWords[ i ] ].push_back( groupsWords[ j ] );
24                    adjWords[ groupsWords[ j ] ].push_back( groupsWords[ i ] );
25                }
26    }
27
28    return adjWords;
29 }

```

Hình 2.58 Chức năng tính toán một Map có chứa các từ làm khóa và một vectơ từ chỉ khái một ký tự làm giá trị. Nó chia các từ thành các nhóm theo độ dài từ. Điều này phiên bản chạy trong 18 giây trên từ điển 89.000 từ.

Thuật toán thứ ba của chúng tôi phức tạp hơn và sử dụng các Map bổ sung! Như trước đây, chúng tôi nhóm các từ theo độ dài từ, và sau đó làm việc trên từng nhóm riêng biệt. Để xem thuật toán này như thế nào hoạt động, giả sử chúng ta đang làm việc với các từ có độ dài 4. Trước tiên, chúng ta muốn tìm các cặp từ, chẳng hạn như rượu vang và rượu chín, giống hệt nhau ngoại trừ chữ cái đầu tiên. Một cách để làm điều này là như sau: Đối với mỗi từ có độ dài 4, bỏ ký tự đầu tiên, để lại ba ký tự đại diện từ. Tạo một Map trong đó khóa là đại diện và giá trị là một vectơ của tất cả các từ có đại diện đó. Ví dụ, trong việc xem xét đầu tiên ký tự của nhóm từ bốn chữ cái, đại diện "ine" tương ứng với "dine", "fine", "rượu", "chín", "của tôi", "cây nho", "cây

thông", "dòng". Đại diện "oot" tương ứng với "boot", "chân", "cuốc", "loot", "muội", "zô". Mỗi vectơ riêng lẻ là một giá trị mới nhất này Map tạo thành một nhóm các từ trong đó bất kỳ từ nào có thể được thay đổi thành bất kỳ từ nào khác bằng một thay thế một ký tự, vì vậy sau khi Map mới nhất này được xây dựng, có thể dễ dàng đi qua nó và thêm các mục vào Map gốc đang được tính toán. Sau đó chúng tôi sẽ tiếp tục ký tự thứ hai của nhóm từ bốn chữ cái, với một Map mới và sau đó là ký tự thứ ba ký tự, và cuối cùng là ký tự thứ tư.

```

for each group g, containing words of length len
    for each position p (ranging from 0 to len-1)
    {
        Make an empty map<string, vector<string>> repsToWords
        for each word w
        {
            Obtain w's representative by removing position p
            Update repsToWords
        }
        Use cliques in repsToWords to update adjWords map
    }
}

```

Hình 2.59 chứa cách triển khai thuật toán này. Thời gian chạy được cải thiện đến hai giây. Điều thú vị là mặc dù việc sử dụng các Map bổ sung làm cho thuật toán nhanh hơn và cú pháp tương đối rõ ràng, mà không sử dụng thực tế rằng các khóa của Map được duy trì theo thứ tự được sắp xếp.

---

```

1 // Computes a map in which the keys are words and values are vectors of words
2 // that differ in only one character from the corresponding key.
3 // Uses an efficient algorithm that is O(N log N) with a map
4 map<string, vector<string>> computeAdjacentWords( const vector<string> & words )
5 {
6     map<string, vector<string>> adjWords;
7     map<int, vector<string>> wordsByLength;
8
9     // Group the words by their length
10    for( auto & str : words )
11        wordsByLength[ str.length( ) ].push_back( str );
12
13    // Work on each group separately
14    for( auto & entry : wordsByLength )
15    {
16        const vector<string> & groupsWords = entry.second;
17        int groupNum = entry.first;
18
19        // Work on each position in each group
20        for( int i = 0; i < groupNum; ++i )

```

Hình 2.59 Chức năng tính toán một Map có chứa các từ làm khóa và một vectơ từ chỉ khái một ký tự làm giá trị. Phiên bản này chạy trong 2 giây trên 89.000- từ điển từ.

```

21      {
22          // Remove one character in specified position, computing representative.
23          // Words with same representatives are adjacent; so populate a map ...
24          map<string,vector<string>> repToWord;
25
26          for( auto & str : groupsWords )
27          {
28              string rep = str;
29              rep.erase( i, 1 );
30              repToWord[ rep ].push_back( str );
31          }
32
33          // and then look for map values with more than one string
34          for( auto & entry : repToWord )
35          {
36              const vector<string> & clique = entry.second;
37              if( clique.size( ) >= 2 )
38                  for( int p = 0; p < clique.size( ); ++p )
39                      for( int q = p + 1; q < clique.size( ); ++q )
40                      {
41                          adjWords[ clique[ p ] ].push_back( clique[ q ] );
42                          adjWords[ clique[ q ] ].push_back( clique[ p ] );
43                      }
44                  }
45          }
46      }
47      return adjWords;
48  }

```

Hình 2.59 (tiếp tục)

Như vậy, có thể cấu trúc dữ liệu hỗ trợ các hoạt động Map nhưng không đảm bảo thứ tự đã sắp xếp có thể hoạt động tốt hơn, vì nó đang được yêu cầu làm ít hơn. Chương 5 khám phá khả năng này và thảo luận về những ý tưởng đằng sau việc triển khai Map thay thế mà C++ 11 thêm vào Thư viện Chuẩn, được gọi là Map không có thứ tự. Một Map không có thứ tự giảm thời gian thực thi từ 2 giây xuống 1,5 giây.

### Tóm Lại

Chúng tôi đã thấy việc sử dụng cây trong hệ điều hành, thiết kế trình biên dịch và tìm kiếm. Cây biểu thức là một ví dụ nhỏ về cấu trúc tổng quát hơn được gọi là cây phân tích cú pháp, là một cấu trúc dữ liệu trung tâm trong thiết kế trình biên dịch. Cây phân tích cú pháp không phải là cây nhị phân,

nhưng là các phần mở rộng tương đối đơn giản của cây biểu thức (mặc dù các thuật toán để xây dựng chúng là không hoàn toàn đơn giản).

Cây tìm kiếm có tầm quan trọng lớn trong thiết kế thuật toán. Họ hỗ trợ gần như tất cả các phép toán hữu ích và chi phí trung bình lôgarit là rất nhỏ. Implemen không đệ quy- Số lượng cây tìm kiếm nhanh hơn một chút, nhưng các phiên bản đệ quy đẹp hơn, thanh lịch và dễ hiểu và dễ gỡ lỗi hơn. Vấn đề với cây tìm kiếm là hiệu suất phụ thuộc nhiều vào đầu vào là ngẫu nhiên. Nếu đây không phải là trường hợp, chạy- Thời gian kết thúc tăng đáng kể, đến mức các cây tìm kiếm trở nên đắt đỏ được liên kết danh sách. Chúng tôi đã thấy một số cách để giải quyết vấn đề này. Cây AVL hoạt động bằng cách nhấn mạnh rằng tất cả các cây con bên trái và bên phải của các nút khác nhau về độ cao nhiều nhất là một. Điều này đảm bảo rằng cây không thể vào quá sâu. Các hoạt động không thay đổi cây, như việc chèn, có thể tất cả đều sử dụng mã cây tìm kiếm nhị phân tiêu chuẩn. Các hoạt động thay đổi cây phải khôi phục cái cây. Điều này có thể hơi phức tạp, đặc biệt là trong trường hợp xóa. Chúng tôi đã cho thấy cách khôi phục cây sau khi chèn trong thời gian  $O(\log N)$ . Chúng tôi cũng đã kiểm tra cây splay. Các nút trong cây lan có thể sâu tùy ý, nhưng sau mỗi lần truy cập cây được điều chỉnh theo cách hơi bí ẩn. Hiệu ứng ròng là bất kỳ chuỗi hoạt động nào của  $M$  cần thời gian  $O(M \log N)$ , giống như một cây cân bằng sẽ mất. Cây B là cây M cách cân bằng (trái ngược với cây 2 chiều hoặc nhị phân), tốt thích hợp cho đĩa; một trường hợp đặc biệt là cây 2–3 ( $M = 3$ ), là một cách khác để triển khai cây tìm kiếm cân bằng. Trong thực tế, thời gian chạy của tất cả các lược đồ cây cân bằng, trong khi nhanh hơn một chút để tìm kiếm, việc chèn và xóa sẽ tệ hơn (bởi một yếu tố không đổi) so với cây tìm kiếm nhị phân, nhưng điều này nói chung có thể chấp nhận được về việc bảo vệ được đưa ra chống lại đầu vào trường hợp xấu nhất dễ dàng thu được. Chương 12 thảo luận về một số cây tìm kiếm bổ sung cấu trúc dữ liệu và cung cấp các triển khai chi tiết. Lưu ý cuối cùng: Bằng cách chèn các phần tử vào cây tìm kiếm và sau đó thực hiện inorder qua, chúng tôi thu được các phần tử theo thứ tự được sắp xếp. Điều này cho phép thuật toán  $O(N \log N)$  sắp xếp, là giới hạn trong trường hợp xấu nhất nếu sử dụng bất kỳ cây tìm kiếm phức tạp nào.

### 3. Hàng đợi ưu tiên

Mặc dù các công việc được gửi đến máy in thường được đặt trên một hàng đợi, nhưng đây có thể không phải lúc nào cũng là điều tốt nhất để làm. Ví dụ: một công việc có thể đặc biệt quan trọng, vì vậy có thể mong muốn cho phép công việc đó chạy ngay khi máy in khả dụng. Ngược lại, nếu khi máy in khả dụng, có một số lệnh in 1 trang và một lệnh in 100 trang, thì có thể hợp lý để thực hiện lệnh in dài này, ngay cả khi nó không phải là lệnh in cuối cùng được gửi. (Thật không may, hầu hết các hệ thống không làm điều này, điều này đôi khi có thể gây khó chịu đặc biệt.)

Tương tự, trong môi trường đa người dùng, bộ lập lịch hệ điều hành phải quyết định chạy quy trình nào trong số một số quy trình. Nói chung, quy trình chỉ được phép chạy trong một khoảng thời gian cố định. Một thuật toán sử dụng một hàng đợi. Công việc ban đầu được đặt ở cuối hàng đợi. Bộ lập lịch sẽ lặp lại công việc đầu tiên trong hàng đợi, chạy nó cho đến khi nó hoàn thành hoặc hết thời hạn và đặt nó ở cuối hàng đợi nếu nó không hoàn thành. Chiến lược này nói chung là không thích hợp, vì các công việc rất ngắn dường như sẽ mất nhiều thời gian do phải chờ đợi để chạy. Nói chung, điều quan trọng là các công việc ngắn hoàn thành càng nhanh càng tốt, vì vậy những công việc này nên được ưu tiên hơn những công việc đã và đang chạy. Hơn nữa, một số công việc không ngắn vẫn rất quan trọng và cần được ưu tiên.

Ứng dụng cụ thể này dường như yêu cầu một loại hàng đợi đặc biệt, được gọi là hàng đợi ưu tiên. Trong phần này, chúng ta sẽ thảo luận về ...

- Triển khai hiệu quả ADT hàng đợi ưu tiên.
- Công dụng của hàng đợi ưu tiên.
- Triển khai nâng cao của hàng đợi ưu tiên.

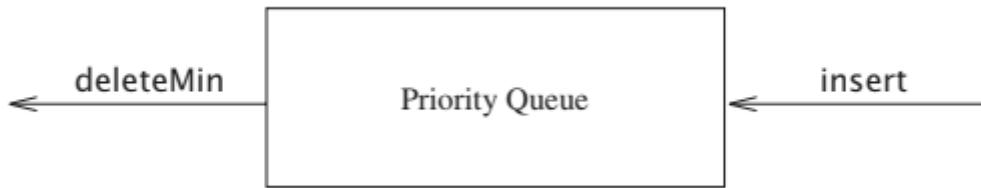
Cấu trúc dữ liệu mà chúng ta sẽ thấy là một trong những cấu trúc ưu tiên nhất trong khoa học máy tính.

#### 3.1 Mô hình

Hàng đợi ưu tiên là một cấu trúc dữ liệu cho phép ít nhất hai hoạt động sau: chèn, điều đó làm điều hiển nhiên; và deleteMin, tìm, trả lại và xóa phần tử tối thiểu trong hàng đợi ưu tiên. Thao tác chèn tương đương với enqueue,

và deleteMin là hàng đợi ưu tiên tương đương với hoạt động dequeue của hàng đợi.

Mã C ++ cung cấp hai phiên bản deleteMin. Một loại bỏ mức tối thiểu; cái kia loại bỏ tối thiểu và lưu trữ giá trị đã loại bỏ trong một đối tượng được truyền bởi tham chiếu.



Hình 3.1 Mô hình cơ bản của hàng đợi ưu tiên

Như với hầu hết các cấu trúc dữ liệu, đôi khi có thể thêm các thao tác khác, nhưng đây là những phần mở rộng và không phải là một phần của mô hình cơ bản được mô tả trong Hình 3.1.

### 3.2 Triển khai đơn giản

Có một số cách rõ ràng để triển khai hàng đợi ưu tiên. Chúng tôi có thể sử dụng một danh sách liên kết, thực hiện chèn ở phía trước trong  $O(1)$  và duyệt qua danh sách, yêu cầu  $O(N)$  thời gian, để xóa tối thiểu. Ngoài ra, chúng tôi có thể nhấn mạnh rằng danh sách luôn được giữ đã sắp xếp; điều này làm cho việc chèn đắt ( $O(N)$ ) và deleteMins rẻ ( $O(1)$ ). Trước đây là có lẽ là ý tưởng tốt hơn của cả hai, dựa trên thực tế là không bao giờ có nhiều lượt deleteMins hơn là chèn.

Một cách khác để thực hiện hàng đợi ưu tiên là sử dụng cây tìm kiếm nhị phân. Điều này mang lại thời gian chạy trung bình  $O(\log N)$  cho cả hai hoạt động. Điều này đúng mặc dù thực tế là mặc dù các lần chèn là ngẫu nhiên, nhưng việc xóa thì không. Nhớ lại rằng phần tử duy nhất chúng tôi xóa là phần tử tối thiểu. Liên tục xóa một nút nằm trong cây con bên trái dường như sẽ làm tổn hại đến sự cân bằng của cây bằng cách làm cho cây con bên

phải trở nên nặng nề. Tuy nhiên, cây con bên phải là ngẫu nhiên. Trong trường hợp xấu nhất, khi deleteMins đã làm cạn kiệt cây con bên trái, cây con bên phải sẽ có nhiều nhất gấp đôi số phần tử cần thiết. Điều này chỉ thêm một hằng số nhỏ vào độ sâu dự kiến của nó. Lưu ý rằng ràng buộc có thể được thực hiện vào trường hợp xấu nhất bị ràng buộc bằng cách sử dụng cây cân bằng; điều này bảo vệ một người chống lại sự chèn xấu trình tự.

Việc sử dụng cây tìm kiếm có thể là quá mức cần thiết vì nó hỗ trợ một loạt các hoạt động không yêu cầu. Cấu trúc dữ liệu cơ bản mà chúng tôi sẽ sử dụng sẽ không yêu cầu liên kết và sẽ hỗ trợ cả hai hoạt động trong trường hợp xấu nhất  $O(\log N)$ . Việc chèn sẽ thực sự mất thời gian liên tục vào trung bình và việc triển khai của chúng tôi sẽ cho phép xây dựng một hàng đợi ưu tiên gồm  $N$  mục trong tuyến tính thời gian, nếu không có xóa can thiệp. Sau đó, chúng ta sẽ thảo luận về cách triển khai các hàng đợi ưu tiên để hỗ trợ hợp nhất hiệu quả. Hoạt động bổ sung này dường như làm phức tạp vấn đề một chút và rõ ràng yêu cầu sử dụng cấu trúc liên kết.

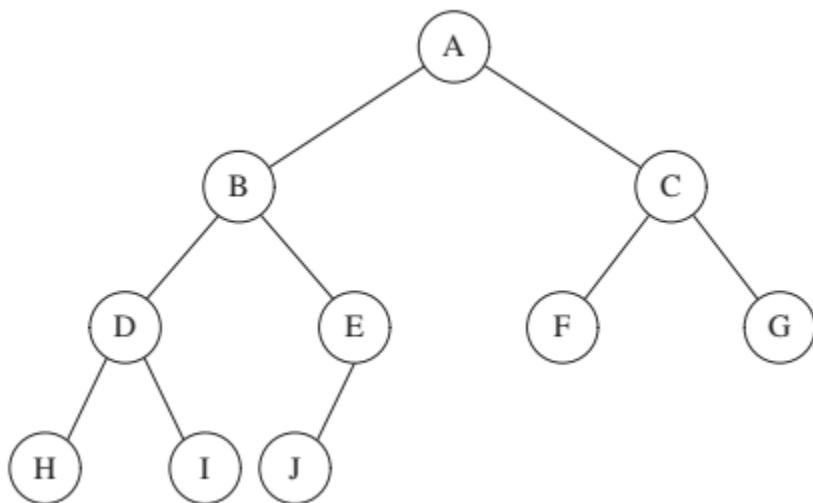
### 3.3 Đống nhị phân

Việc triển khai chúng tôi sẽ sử dụng được gọi là một đống nhị phân. Việc sử dụng nó rất phổ biến đối với triển khai hàng đợi ưu tiên, trong ngữ cảnh của hàng đợi ưu tiên, khi từ đống được sử dụng mà không có định tính, nó thường được giả định là đề cập đến việc triển khai này của cấu trúc dữ liệu. Trong phần này, chúng tôi sẽ đề cập đến các đống nhị phân chỉ đơn thuần là đống. Giống cây tìm kiếm nhị phân, heap có hai thuộc tính, đó là thuộc tính cấu trúc và một thuộc tính theo thứ tự đống. Như với cây AVL, một hoạt động trên một đống có thể phá hủy một trong các thuộc tính, vì vậy hoạt động heap không được kết thúc cho đến khi tất cả các thuộc tính heap theo thứ tự. Điều này hóa ra đơn giản để làm.

#### 3.3.1 Thuộc tính cấu trúc

Một đống là một cây nhị phân được lấp đầy hoàn toàn, với ngoại lệ có thể có ở dưới cùng cấp, được điền từ trái sang phải. Cây như vậy được gọi là cây nhị phân hoàn chỉnh. Hình 3.2 cho thấy một ví dụ.

Dễ dàng chỉ ra rằng một cây nhị phân hoàn chỉnh có chiều cao  $h$  có từ  $2^h$  đến  $2^{h+1}-1$  điểm giao. Điều này ngụ ý rằng chiều cao của một cây nhị phân hoàn chỉnh là  $\log N$ , rõ ràng là  $O(\log N)$ . Một quan sát quan trọng là bởi vì một cây nhị phân hoàn chỉnh rất đều đặn, nó có thể được biểu diễn trong một mảng và không cần liên kết. Mảng trong Hình 3.3 tương ứng với đống trong Hình 3.2.



Hình 3.2 Một cây nhị phân hoàn chỉnh

	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Hình 6.3 Triển khai mảng của cây nhị phân hoàn chỉnh

```

1 template <typename Comparable>
2 class BinaryHeap
3 {
4     public:
5         explicit BinaryHeap( int capacity = 100 );
6         explicit BinaryHeap( const vector<Comparable> & items );
7
8         bool isEmpty( ) const;
9         const Comparable & findMin( ) const;
10
11        void insert( const Comparable & x );
12        void insert( Comparable && x );
13        void deleteMin( );
14        void deleteMin( Comparable & minItem );
15        void makeEmpty( );
16
17    private:
18        int             currentSize; // Number of elements in heap
19        vector<Comparable> array;      // The heap array
20
21        void buildHeap( );
22        void percolateDown( int hole );
23    };

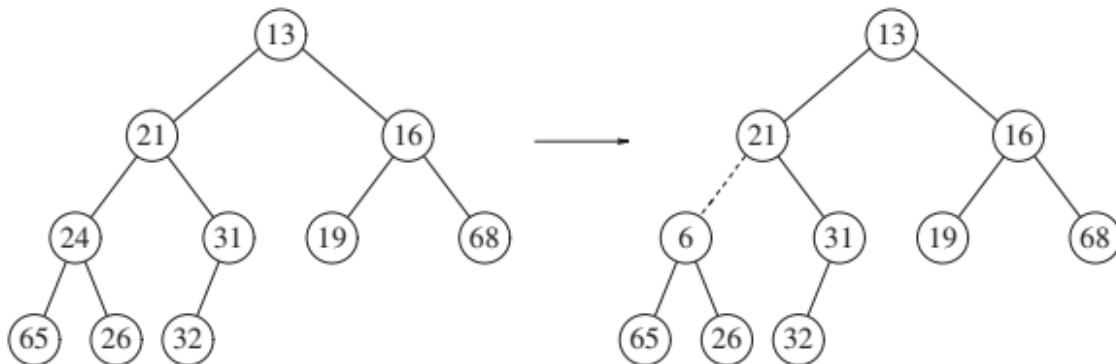
```

Hình 3.4 Giao diện lớp cho hàng đợi ưu tiên

Đối với bất kỳ phần tử nào ở vị trí mảng  $i$ , phần tử bên trái ở vị trí  $2i$ , phần tử bên phải ở ô sau ô con bên trái ( $2i + 1$ ) và ô cha ở vị trí  $i/2$ . Vì vậy, không chỉ liên kết không bắt buộc, nhưng các hoạt động cần thiết để đi qua cây cực kỳ đơn giản và có thể rất nhanh trên hầu hết các máy tính. Vấn đề duy nhất với việc triển khai này là cần phải có ước tính trước về kích thước heap tối đa, nhưng thường thì đây là không thành vấn đề (và chúng tôi có thể thay đổi kích thước nếu cần). Trong Hình 3.3, giới hạn về kích thước đống là 13 các yếu tố. Mảng có vị trí 0; thêm về điều này sau. Khi đó, cấu trúc dữ liệu heap sẽ bao gồm một mảng (các đối tượng có thể so sánh được) và số nguyên đại diện cho kích thước heap hiện tại. Hình 3.4 cho thấy một giao diện hàng đợi ưu tiên. Trong suốt chương này, chúng ta sẽ vẽ các đống dưới dạng cây, với ngụ ý rằng một thực tế sẽ sử dụng các mảng đơn giản.

### 3.3.2 Thuộc tính thứ tự đồng

Thuộc tính cho phép các hoạt động được thực hiện nhanh chóng là thuộc tính theo thứ tự đồng. Vì chúng ta muốn có thể nhanh chóng tìm được giá trị nhỏ nhất, nên phần tử nhỏ nhất phải ở gốc. Nếu chúng ta cho rằng bất kỳ cây con nào cũng phải là một heap, thì bất kỳ nút nào cũng phải nhỏ hơn tất cả các nút con của nó.



Hình 3.5 Hai cây hoàn chỉnh (chỉ cây bên trái là một đống)

Áp dụng logic này, chúng tôi đến thuộc tính thứ tự đồng. Trong một heap, đối với mọi nút X, khóa trong cha của X nhỏ hơn (hoặc bằng) khóa trong X, với ngoại lệ là gốc (không có cha). Trong hình 3.5, cây bên trái là một đống, nhưng quyền của cây thì không (đường đứt nét thể hiện sự vi phạm thứ tự đồng).

Bằng thuộc tính thứ tự đồng, phần tử tối thiểu luôn có thể được tìm thấy ở gốc. Do đó, chúng tôi nhận được phép toán phụ, `findMin`, trong thời gian không đổi.

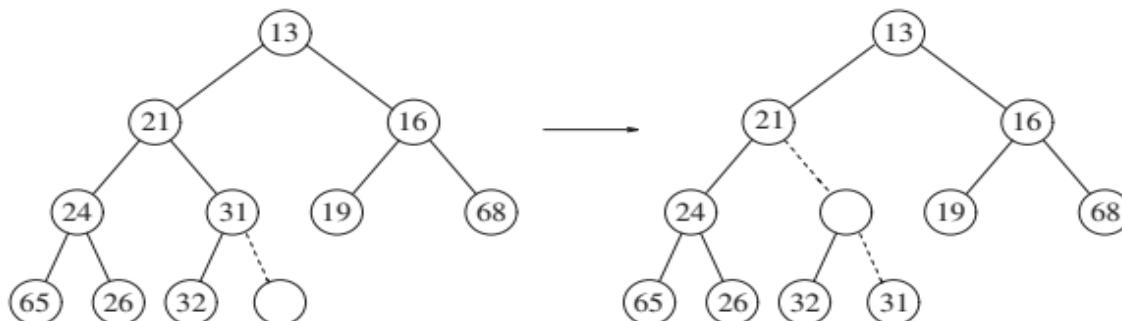
### 3.3.3 Hoạt động đống cơ bản

Thật dễ dàng (cả về mặt khái niệm và thực tế) để thực hiện hai hoạt động cần thiết. Tất cả công việc liên quan đến việc đảm bảo rằng thuộc tính đặt hàng đống được duy trì.

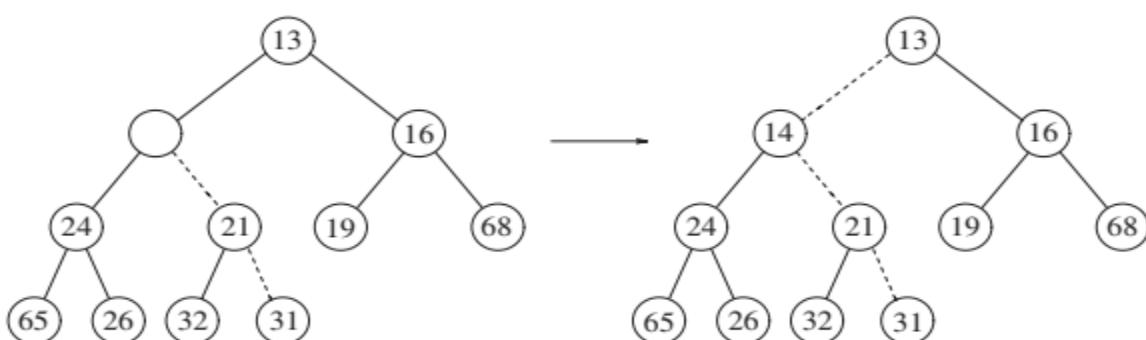
### **Chèn (insert)**

Để chèn một phần tử X vào heap, chúng ta tạo một lỗ ở vị trí có sẵn tiếp theo, vì nếu không, cây sẽ không hoàn chỉnh. Nếu X có thể được đặt vào lỗ mà không vi phạm thứ tự đồng, thì chúng ta làm như vậy và đã xong. Nếu không, chúng tôi trượt phần tử nằm trong nút cha của lỗ vào lỗ, do đó làm cho lỗ sủi bọt về phía gốc. Chúng tôi tiếp tục quá trình này cho đến khi có thể đặt X vào lỗ. Hình 3.6 cho thấy rằng để chèn 14, chúng ta tạo một lỗ ở vị trí đồng sẵn có tiếp theo. Chèn 14 vào lỗ sẽ vi phạm thuộc tính thứ tự đồng, do đó 31 sẽ bị trượt xuống lỗ. Chiến lược này được tiếp tục trong Hình 3.7 cho đến khi tìm được vị trí chính xác cho 14.

Chiến lược chung này được gọi là thấm dần (percolate up); phần tử mới được tõ màu lên đồng cho đến khi tìm thấy vị trí chính xác. Việc chèn được thực hiện dễ dàng với mã hiển thị trong Hình 3.8.



Hình 3.6 Cố gắng chèn 14: tạo lỗ, và làm sủi bọt lỗ lên



Hình 3.7 Hai bước còn lại để chèn 14 trong heap trước

```

1      /**
2       * Insert item x, allowing duplicates.
3       */
4      void insert( const Comparable & x )
5      {
6          if( currentSize == array.size( ) - 1 )
7              array.resize( array.size( ) * 2 );
8
9          // Percolate up
10         int hole = ++currentSize;
11         Comparable copy = x;
12
13         array[ 0 ] = std::move( copy );
14         for( ; x < array[ hole / 2 ]; hole /= 2 )
15             array[ hole ] = std::move( array[ hole / 2 ] );
16         array[ hole ] = std::move( array[ 0 ] );
17     }

```

Hình 3.8 Quy trình chèn vào một đống nhị phân

Chúng tôi có thể đã triển khai sự thấm đẫm trong quy trình chèn bằng cách thực hiện hoán đổi lặp lại cho đến khi thiết lập đúng thứ tự, nhưng hoán đổi yêu cầu ba câu lệnh gán. Nếu một phần tử được tô màu lên  $d$  cấp, số lượng nhiệm vụ được thực hiện bởi các hoán đổi sẽ là  $3d$ . Phương pháp của chúng tôi sử dụng phép gán  $d + 1$ .

Nếu phần tử được chèn là phần tử tối thiểu mới, phần tử đó sẽ được đẩy lên trên cùng. Tại một thời điểm nào đó, lỗ sẽ là 1 và chúng ta sẽ muốn thoát ra khỏi vòng lặp. Chúng tôi có thể làm điều này với một bài kiểm tra rõ ràng hoặc chúng tôi có thể đặt một bản sao của mục được chèn vào vị trí 0 để làm cho vòng lặp kết thúc. Chúng tôi chọn đặt X vào vị trí 0.

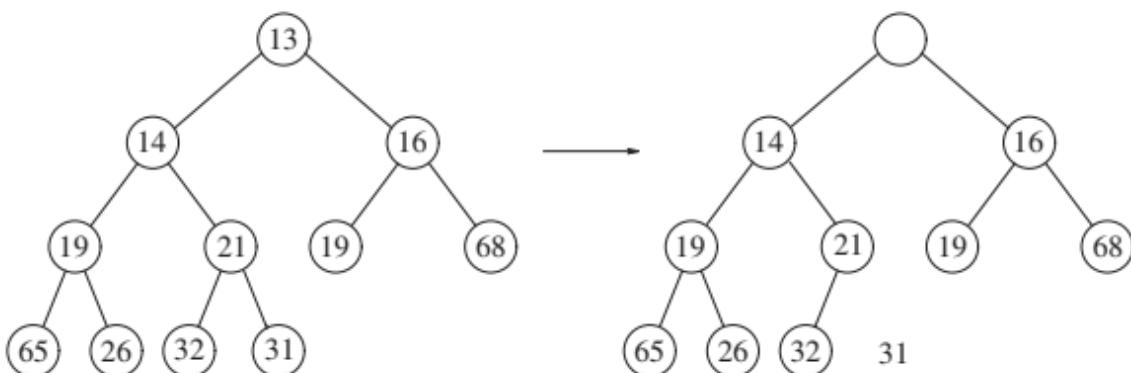
Thời gian để thực hiện việc chèn có thể bằng  $O(\log N)$ , nếu phần tử được chèn là phần tử tối thiểu mới và được tô màu đến tận gốc. Trung bình, sự thấm nước kết thúc sớm; người ta đã chỉ ra rằng trung bình cần có 2.607 phép so sánh để thực hiện chèn, do đó, chèn trung bình sẽ di chuyển một phần tử lên 1.607 cấp.

### ***deleteMin***

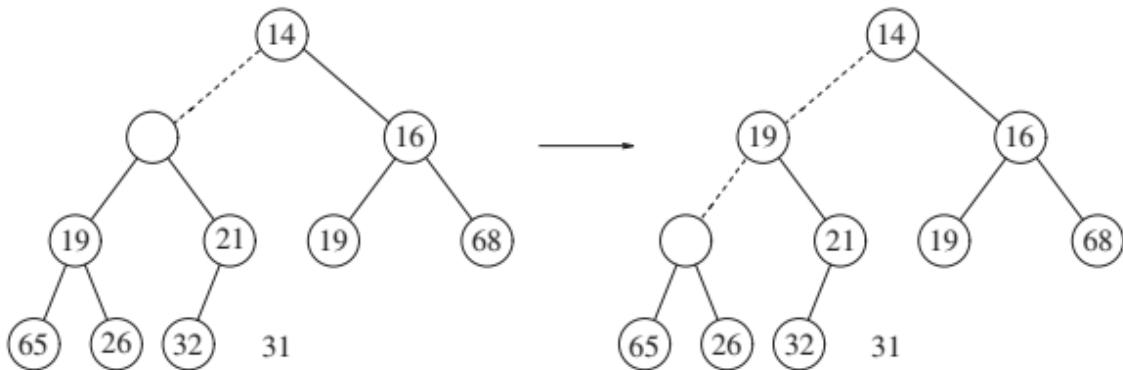
*deleteMins* được xử lý theo cách tương tự như chèn. Tìm mức tối thiểu rất dễ dàng; phần khó là loại bỏ nó. Khi loại bỏ mức tối thiểu, một lỗ được tạo ra ở gốc. Vì heap bây giờ trở nên nhỏ hơn, nên phần tử X cuối cùng trong heap phải di chuyển đến đâu đó trong heap. Nếu X có thể được đặt vào lỗ, thì chúng ta đã hoàn thành. Điều này khó xảy ra, vì vậy chúng tôi trượt con của lỗ nhỏ hơn vào lỗ, do đó đẩy lỗ xuống một mức. Chúng tôi lặp lại bước này cho đến khi có thể đặt X vào lỗ. Do đó, hành động của chúng ta là đặt X vào đúng vị trí của nó dọc theo một đường dẫn từ gốc chứa các con tối thiểu.

Trong hình 3.9, hình bên trái cho thấy một đống trước khi có *deleteMin*. Sau khi 13 bị loại bỏ, bây giờ chúng ta phải cố gắng đặt 31 vào đống. Giá trị 31 không thể được đặt vào lỗ, vì điều này sẽ vi phạm thứ tự đống. Do đó, chúng tôi đặt con nhỏ hơn (14) vào lỗ, trượt lỗ xuống một mức (xem Hình 3.10). Chúng tôi lặp lại điều này một lần nữa, và vì 31 lớn hơn 19, chúng tôi đặt 19 vào lỗ và tạo một lỗ mới sâu hơn một cấp. Sau đó, chúng tôi đặt 26 vào lỗ và tạo một lỗ mới ở mức dưới cùng vì một lần nữa, 31 là quá lớn. Cuối cùng, chúng ta có thể đặt 31 vào lỗ (Hình 3.11). Chiến lược chung này được biết đến như một chiến lược đi xuống. Chúng tôi sử dụng kỹ thuật tương tự như trong quy trình chèn để tránh việc sử dụng hoán đổi trong quy trình này.

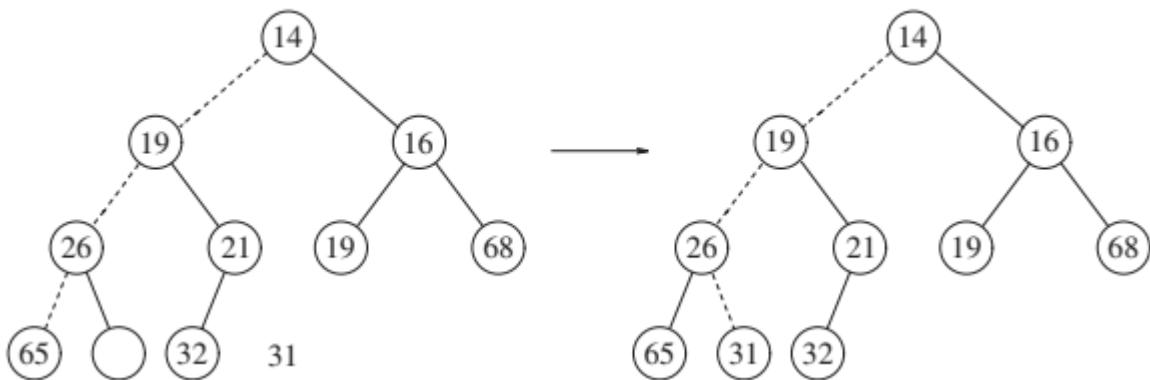
Lỗi triển khai thường xuyên trong heap xảy ra khi có một số phần tử chẵn trong heap và gặp phải một nút chỉ có một phần tử con. Bạn phải đảm bảo



Hình 3.9 Tạo lỗ ở gốc



Hình 3.10 Hai bước tiếp theo trong deleteMin



Hình 3.11 Hai bước cuối cùng trong deleteMin

không cho rằng luôn có hai trẻ em, vì vậy điều này thường bao gồm một bài kiểm tra bổ sung. Trong đoạn mã được mô tả trong Hình 3.12, chúng tôi đã thực hiện kiểm tra này ở dòng 40. Một giải pháp cực kỳ khó khăn là luôn đảm bảo rằng thuật toán của bạn nghĩ rằng mọi nút đều có hai nút con. Thực hiện điều này bằng cách đặt một sentinel, có giá trị cao hơn bất kỳ giá trị nào trong heap, ở vị trí sau khi heap kết thúc, ở đầu mỗi lần thăm xuống khi kích thước heap bằng nhau. Bạn nên suy nghĩ rất kỹ trước khi thực hiện điều này, và bạn phải đưa ra một nhận xét nổi bật nếu bạn sử dụng kỹ thuật này. Mặc dù điều này loại bỏ sự cần thiết phải kiểm tra sự hiện diện của một đứa trẻ phù hợp, bạn không thể loại bỏ yêu cầu bạn kiểm tra khi bạn chạm đến đáy, bởi vì điều này sẽ yêu cầu một lính canh cho mỗi lá. Thời gian chạy trường hợp xấu nhất cho thao tác này là  $O(\log N)$ . Trung bình, phần tử được đặt ở gốc được phủ

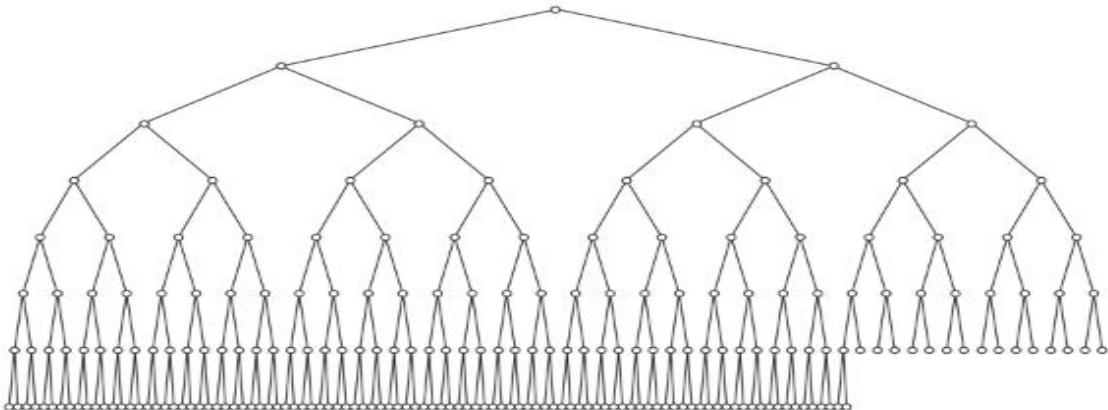
gần như xuống dưới cùng của heap (là mức mà nó xuất phát), vì vậy thời gian chạy trung bình là  $O(\log N)$ .

### 3.3.4 Hoạt động Heap khác

Lưu ý rằng mặc dù việc tìm kiếm giá trị nhỏ nhất có thể được thực hiện trong thời gian không đổi, một đống được thiết kế để tìm phần tử tối thiểu (còn được gọi là heap (min)) không giúp ích được gì- bao giờ trong việc tìm kiếm phần tử tối đa. Trên thực tế, một đống có rất ít thông tin đặt hàng, vì vậy

```
1  /**
2   * Remove the minimum item.
3   * Throws UnderflowException if empty.
4   */
5  void deleteMin( )
6  {
7      if( isEmpty( ) )
8          throw UnderflowException{ };
9
10     array[ 1 ] = std::move( array[ currentSize-- ] );
11     percolateDown( 1 );
12 }
13
14 /**
15  * Remove the minimum item and place it in minItem.
16  * Throws UnderflowException if empty.
17  */
18 void deleteMin( Comparable & minItem )
19 {
20     if( isEmpty( ) )
21         throw UnderflowException{ };
22
23     minItem = std::move( array[ 1 ] );
24     array[ 1 ] = std::move( array[ currentSize-- ] );
25     percolateDown( 1 );
26 }
27
28 /**
29  * Internal method to percolate down in the heap.
30  * hole is the index at which the percolate begins.
31  */
32 void percolateDown( int hole )
33 {
34     int child;
35     Comparable tmp = std::move( array[ hole ] );
36
37     for( ; hole * 2 <= currentSize; hole = child )
38     {
39         child = hole * 2;
40         if( child != currentSize && array[ child + 1 ] < array[ child ] )
41             ++child;
42         if( array[ child ] < tmp )
43             array[ hole ] = std::move( array[ child ] );
44         else
45             break;
46     }
47     array[ hole ] = std::move( tmp );
48 }
```

Hình 3.12 Phương pháp thực hiện deleteMin trong một đống nhị phân



Hình 3.13 Một cây nhị phân hoàn chỉnh rất lớn

không có cách nào để tìm bất kỳ phần tử cụ thể nào mà không cần quét tuyển tính qua toàn bộ đống. Để thấy điều này, hãy xem xét cấu trúc đống lớn (các phần tử không được hiển thị) trong Hình 3.13, nơi chúng ta thấy rằng thông tin duy nhất được biết về phần tử tối đa là nó nằm ở một trong các lá. Tuy nhiên, một nửa các phần tử được chứa trong lá, vì vậy đây thực tế là thông tin vô dụng. Vì lý do này, nếu điều quan trọng là biết vị trí của các phần tử, một số cấu trúc dữ liệu khác, chẳng hạn như bảng băm, phải được sử dụng ngoài heap. (Nhớ lại rằng mô hình không cho phép nhìn vào bên trong đống.) Nếu chúng ta giả định rằng vị trí của mọi phần tử được biết bằng một số phương pháp khác, thì một số phép toán khác sẽ trở nên rẻ tiền. Ba phép toán đầu tiên dưới đây đều chạy trong trường hợp xấu nhất theo thời gian logarit.

### ***Giảm dần***

Phép toán ReduceKey ( $p$ ,  $\Delta$ ) làm giảm giá trị của mục ở vị trí  $p$  một lượng dương  $\Delta$ . Vì điều này có thể vi phạm thứ tự đống, nên nó phải được khắc phục bằng cách tăng lên. Thao tác này có thể hữu ích cho quản trị viên hệ thống: Họ có thể làm cho chương trình của họ chạy với mức độ ưu tiên cao nhất.

### ***Tăng***

Phép toán gainKey ( $p$ ,  $\Delta$ ) làm tăng giá trị của mặt hàng ở vị trí  $p$  một lượng dương  $\Delta$ . Điều này được thực hiện với một chất thấm xuống. Nhiều bộ

lập lịch tự động bỏ mức ưu tiên của một quy trình đang tiêu tốn quá nhiều thời gian của CPU.

## Xóa

Thao tác remove (p) loại bỏ nút ở vị trí p khỏi heap. Điều này được thực hiện trước tiên bằng cách thực hiện ReduceKey (p,  $\infty$ ) và sau đó thực hiện deleteMin (). Khi một quá trình được kết thúc bởi người dùng (thay vì kết thúc bình thường), nó phải được xóa khỏi hàng đợi ưu tiên.

## Xây dựng Heap

Nhóm nhị phân đôi khi được xây dựng từ tập hợp các mục ban đầu. Cấu trúc này nhận N mục đầu vào và đặt chúng vào một đống. Rõ ràng, điều này có thể được thực hiện với N lần chèn liên tiếp. Vì mỗi lần chèn sẽ lấy trung bình O (1) và O ( $\log N$ ) thời gian trong trường hợp xấu nhất, nên tổng thời gian chạy của thuật toán này sẽ là trung bình O (N) nhưng trường hợp xấu nhất là O ( $N \log N$ ). Vì đây là một lệnh đặc biệt và không có thao tác nào khác can thiệp, và chúng ta đã biết rằng lệnh có thể được thực hiện trong thời gian biến đổi tuyến tính, nên có thể hy vọng rằng với sự cẩn thận hợp lý, giới hạn thời gian tuyến tính có thể được đảm bảo.

Thuật toán chung là đặt N mục vào cây theo thứ tự bất kỳ, duy trì thuộc tính cấu trúc. Sau đó, nếu percolateDown (i) thấm xuống từ nút i, thì quy trình buildHeap trong Hình 3.14 có thể được sử dụng bởi phương thức khởi tạo để tạo một cây có thứ tự đống.

Cây đầu tiên trong hình 3.15 là cây không có thứ tự. Bảy cây còn lại trong các Hình 3.15 đến 3.18 cho thấy kết quả của mỗi trong số bảy cây percolat. Mỗi đường đứt nét tương ứng với hai phép so sánh: một để tìm con nhỏ hơn và một để so sánh con nhỏ hơn với nút. Chú ý rằng chỉ có 10 đường đứt nét trong toàn bộ thuật toán (có thể 11th - ở đâu?) tương ứng với 20 phép so sánh.

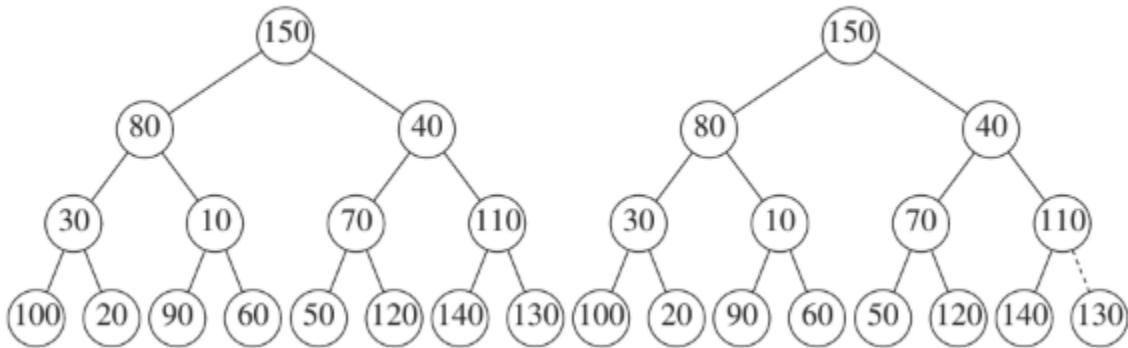
Để ràng buộc thời gian chạy của buildHeap, chúng ta phải ràng buộc số lượng đường đứt nét. Điều này có thể được thực hiện bằng cách tính tổng chiều cao của tất cả các nút trong heap, là số lượng đường đứt nét tối đa. Những gì chúng tôi muốn hiển thị là  $O(N)$ .

```

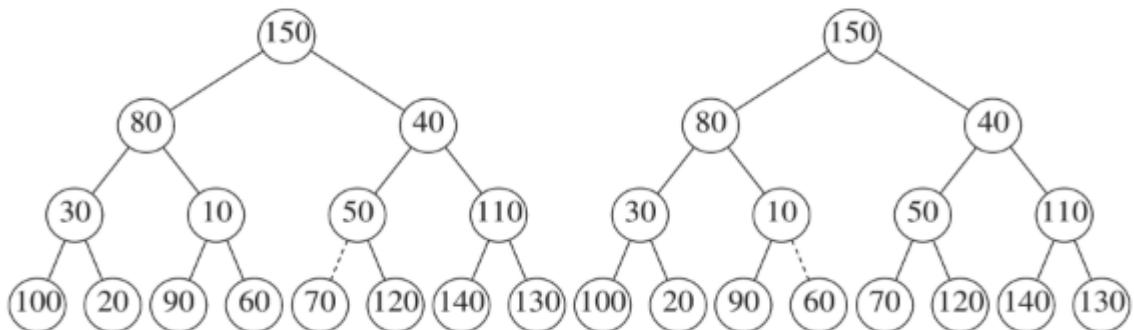
1     explicit BinaryHeap( const vector<Comparable> & items )
2         : array( items.size( ) + 10 ), currentSize{ items.size( ) }
3     {
4         for( int i = 0; i < items.size( ); ++i )
5             array[ i + 1 ] = items[ i ];
6         buildHeap( );
7     }
8
9     /**
10      * Establish heap order property from an arbitrary
11      * arrangement of items. Runs in linear time.
12      */
13     void buildHeap( )
14     {
15         for( int i = currentSize / 2; i > 0; --i )
16             percolateDown( i );
17     }

```

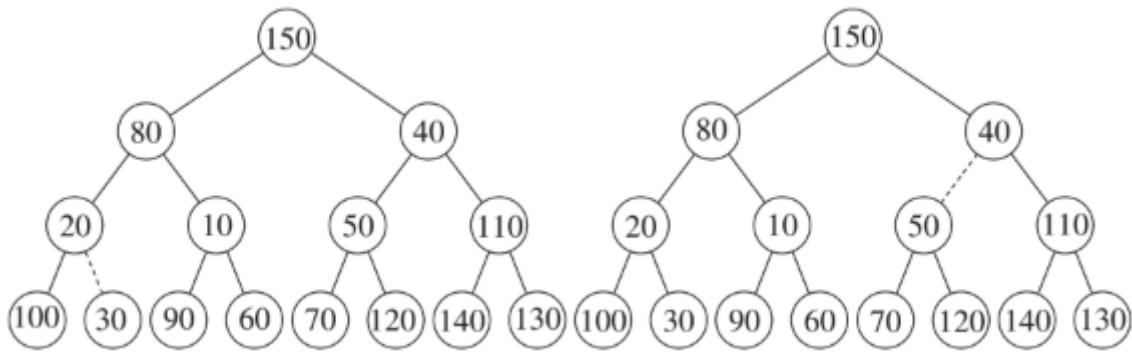
Hình 3.14 xây dựng Heap và hàm tạo



Hình 3.15 Bên trái: đống ban đầu; phải: sau khi thẩm xuống (7)



Hình 3.16 Bên trái: sau khi thẩm xuống (6); phải: sau khi thẩm xuống (5)



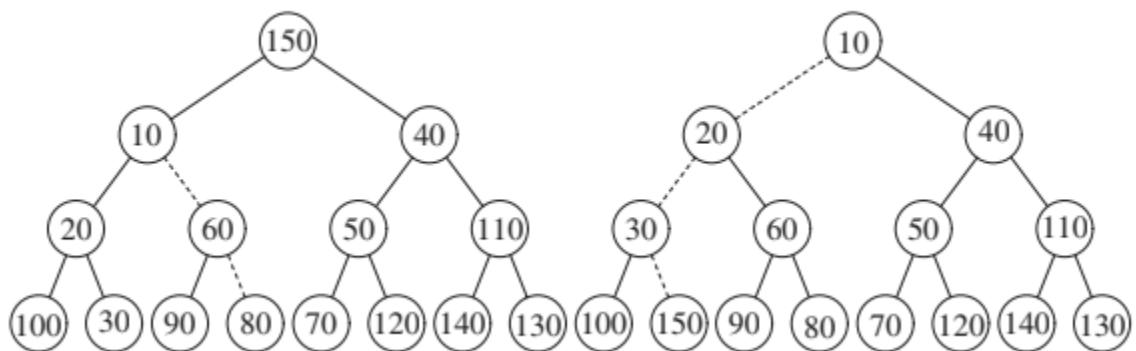
Hình 3.17 Bên trái: sau khi thăm xuống (4); phải: sau khi thăm xuống (3)

### Định lý 6.1

Đối với cây nhị phân hoàn hảo có chiều cao  $h$  chứa  $2^{h+1}-1$  nút, tổng các chiều cao của các nút là  $2^{h+1} - 1 - (h + 1)$ .

### Bằng chứng:

Dễ dàng nhận thấy cây này gồm 1 nút ở độ cao  $h$ , 2 nút ở độ cao  $h - 1$ ,  $2^2$  các nút ở độ cao  $h - 2$  và nói chung là các nút  $2^i$  ở độ cao  $h - i$ . Khi đó, tổng các chiều cao của tất cả các nút chứng minh định lý



Hình 3.18 Bên trái: sau khi thăm xuống (2); phải: sau khi thăm xuống (1)

$$S = \sum_{i=0}^h 2^i(h-i)$$

$$= h + 2(h - 1) + 4(h - 2) + 8(h - 3) + 16(h - 4) + \dots + 2^h - 1 \quad (1) \quad (3.1)$$

Nhân với 2 ta được phương trình

$$2S = 2h + 4(h - 1) + 8(h - 2) + 16(h - 3) + \dots + 2^h \quad (1) \quad (3.2)$$

Chúng tôi trừ hai phương trình này và thu được Công thức (3.3). Chúng tôi thấy rằng các điều khoản nhất định gần như hủy bỏ. Ví dụ, chúng ta có  $2h - 2(h - 1) = 2$ ,  $4(h - 1) - 4(h - 2) = 4$ , và như thế. Số hạng cuối cùng trong Phương trình (3.2),  $2^h$ , không xuất hiện trong Phương trình (3.1); do đó, nó xuất hiện trong Công thức (3.3). Số hạng đầu tiên trong Công thức (3.1),  $h$ , không xuất hiện trong phương trình (3.2); do đó,  $-h$  xuất hiện trong Công thức (3.3). Chúng tôi đạt được

$$S = -h + 2 + 4 + 8 + \dots + 2^{h-1} + 2^h = (2^{h+1} - 1) - (h + 1) \quad (3.3)$$

Một cây hoàn chỉnh không phải là một cây nhị phân hoàn hảo, nhưng kết quả chúng ta thu được là một ràng buộc trên tổng chiều cao của các nút trong một cây hoàn chỉnh. Vì một cây hoàn chỉnh có từ  $2^h$  đến  $2^{h+1}$  nút, định lý này ngụ ý rằng tổng này là  $O(N)$ , trong đó  $N$  là số lượng nút.

Mặc dù kết quả chúng tôi thu được là đủ để cho thấy rằng buildHeap là tuyến tính, bị ràng buộc về tổng chiều cao không mạnh nhất có thể. Để có một cây hoàn chỉnh với  $N = 2$  nút, giới hạn mà chúng ta thu được gần như là  $2N$ . Tổng các chiều cao có thể được hiển thị bằng quy nạp là  $N - b(N)$ , trong đó  $b(N)$  là số 1 trong hệ nhị phân đại diện của  $N$ .

### 3.4 Các ứng dụng của hàng đợi ưu tiên

Chúng tôi đã đề cập đến cách hàng đợi ưu tiên được sử dụng trong thiết kế hệ điều hành. Ở đây chúng tôi sẽ chỉ ra cách sử dụng hàng đợi ưu tiên để có được giải pháp cho hai vấn đề.

#### 3.4.1 Vấn đề lựa chọn

Thuật toán đầu tiên, mà chúng ta sẽ gọi là thuật toán 1A, là đọc các phần tử vào một mảng và sắp xếp chúng, trả về phần tử thích hợp. Giả sử một thuật toán sắp xếp đơn giản, thời gian chạy là  $O(N^2)$ . Thuật toán thay thế, 1B, là

đọc k phần tử vào một mảng và sắp xếp chúng. Nhỏ nhất trong số này là ở vị trí thứ k. Chúng tôi xử lý từng yếu tố còn lại một. Khi một phần tử đến, nó được so sánh với phần tử thứ k trong mảng. Nếu nó lớn hơn, thì phần tử thứ k bị loại bỏ và phần tử mới được đặt vào đúng vị trí trong số k - 1 phần tử còn lại. Khi thuật toán kết thúc, phần tử ở vị trí thứ k là câu trả lời. Thời gian chạy là  $O(N \cdot k)$  (tại sao?). Nếu  $k = N / 2$ , thì cả hai thuật toán đều là  $O(N^2)$ . Lưu ý rằng với k bất kỳ, chúng ta có thể giải bài toán đối xứng tìm phần tử nhỏ nhất thứ  $(N - k + 1)$ , vì vậy  $k = N / 2$  thực sự là trường hợp khó nhất cho các thuật toán này. Đây cũng là trường hợp thú vị nhất, vì giá trị này của k được gọi là giá trị trung bình. Chúng tôi đưa ra hai thuật toán ở đây, cả hai đều chạy trong  $O(N \log N)$  trong trường hợp cực đoan  $k = N / 2$ , đây là một cải tiến khác biệt.

### **Thuật toán 6A**

Để đơn giản, chúng ta giả sử rằng chúng ta quan tâm đến việc tìm phần tử nhỏ nhất thứ k. Thuật toán rất đơn giản. Chúng tôi đọc N phần tử thành một mảng. Sau đó, chúng tôi áp dụng thuật toán buildHeap cho mảng này. Cuối cùng, chúng ta thực hiện các thao tác k deleteMin. Cuối cùng phần tử được trích xuất từ đống là câu trả lời của chúng tôi. Rõ ràng rằng bằng cách thay đổi thuộc tính heap-order, chúng ta có thể giải quyết vấn đề ban đầu là tìm phần tử lớn thứ k. Tính đúng đắn của thuật toán phải rõ ràng. Thời gian trong trường hợp xấu nhất là  $O(N)$  đến xây dựng heap, nếu buildHeap được sử dụng và  $O(\log N)$  cho mỗi deleteMin. Vì có k deleteMins, chúng tôi thu được tổng thời gian chạy là  $O(N + k \log N)$ . Nếu  $k = O(N / \log N)$ , thì thời gian chạy bị chi phối bởi hoạt động buildHeap và là  $O(N)$ . Đối với các giá trị lớn hơn của k, thời gian chạy là  $O(k \log N)$ . Nếu  $k = N / 2$ , thì thời gian chạy là  $(N \log N)$ . Lưu ý rằng nếu chúng ta chạy chương trình này với  $k = N$  và ghi lại các giá trị khi chúng rời khỏi heap, về cơ bản chúng ta sẽ sắp xếp tệp đầu vào theo thời gian  $O(N \log N)$ .

### **Thuật toán 6B**

Đối với thuật toán thứ hai, chúng ta quay lại bài toán ban đầu và tìm phần tử lớn thứ k. Chúng tôi sử dụng ý tưởng từ thuật toán 1B. Tại bất kỳ thời điểm nào, chúng ta sẽ duy trì một tập S gồm k phần tử lớn nhất. Sau khi k phần tử đầu tiên được đọc, khi một phần tử mới được đọc nó được so sánh với phần

tử lớn thứ k, mà chúng tôi ký hiệu là  $S_k$ . Lưu ý rằng  $S_k$  là phần tử nhỏ nhất trong  $S$ . Nếu phần tử mới lớn hơn, thì nó thay thế  $S_k$  trong  $S$ . Khi đó  $S$  sẽ có một phần tử nhỏ nhất mới, có thể là phần tử mới được thêm vào hoặc không. Ở cuối đầu vào, chúng tôi tìm phần tử nhỏ nhất trong  $S$  và trả về nó dưới dạng câu trả lời. Tuy nhiên, ở đây, chúng ta sẽ sử dụng một heap để triển khai  $S$ . K phần tử đầu tiên được đặt vào heap trong tổng thời gian  $O(k)$  với lời gọi `buildHeap`. Thời gian để xử lý mỗi phần tử còn lại là  $O(1)$ , để kiểm tra xem phần tử đó có đi vào  $S$ , cộng với  $O(\log k)$  hay không, để xóa  $S_k$  và chèn phần tử mới nếu cần thiết. Như vậy, tổng thời gian là  $O(k + (N - k) \log k) = O(N \log k)$ . Thuật toán này cũng đưa ra giới hạn ( $N \log N$ ) để tìm giá trị trung bình.

### 3.4.2 Mô phỏng sự kiện

Nhớ lại rằng chúng ta có một hệ thống, chẳng hạn như ngân hàng, nơi khách hàng đến và xếp hàng chờ cho đến khi có một trong  $k$  giao dịch viên. Việc đến của khách hàng được điều chỉnh bởi một hàm phân phối xác suất, cũng như thời gian phục vụ (lượng thời gian được phục vụ khi có giao dịch viên). Chúng tôi quan tâm đến các số liệu thống kê chẳng hạn như thời gian trung bình một khách hàng phải đợi hoặc thời gian của hàng.

Với một số phân phối xác suất và giá trị của  $k$ , những câu trả lời này có thể được tính chính xác. Tuy nhiên, khi  $k$  lớn hơn, việc phân tích trở nên khó khăn hơn đáng kể, vì vậy việc sử dụng máy tính để mô phỏng hoạt động của ngân hàng là điều hấp dẫn. Bằng cách này, nhân viên ngân hàng có thể xác định cần bao nhiêu giao dịch viên để đảm bảo dịch vụ diễn ra suôn sẻ.

Một mô phỏng bao gồm các sự kiện xử lý. Hai sự kiện ở đây là (a) khách hàng đến và (b) khách hàng rời đi, do đó giải phóng giao dịch viên.

Chúng ta có thể sử dụng các hàm xác suất để tạo luồng đầu vào bao gồm các cặp thời gian đến và thời gian phục vụ được sắp xếp theo thứ tự cho mỗi khách hàng, được sắp xếp theo thời gian đến. Chúng ta không cần sử dụng thời gian chính xác trong ngày. Thay vào đó, chúng ta có thể sử dụng một đơn vị lượng tử, mà chúng ta sẽ gọi là dấu tích.

Một cách để thực hiện mô phỏng này là bắt đầu đồng hồ mô phỏng ở mức không tích tắc. Sau đó, chúng tôi tiến lên đồng hồ từng tích một, kiểm tra

xem có sự kiện nào không. Nếu có, thì chúng tôi xử lý (các) sự kiện và tổng hợp thống kê. Khi không còn khách hàng nào trong luồng đầu vào và tất cả các giao dịch viên đều rảnh, thì quá trình mô phỏng kết thúc.

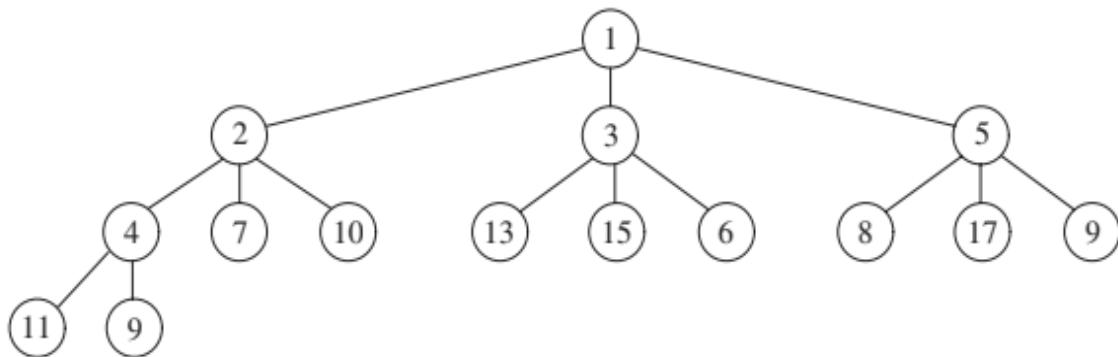
Vấn đề với chiến lược mô phỏng này là thời gian chạy của nó không phụ thuộc vào số lượng khách hàng hoặc sự kiện (có hai sự kiện cho mỗi khách hàng), mà thay vào đó phụ thuộc vào số lượng tick, không thực sự là một phần của đầu vào. Để xem tại sao điều này lại quan trọng, giả sử chúng ta đã thay đổi đơn vị đồng hồ thành mili giây và nhân tất cả các lần trong đầu vào với 1.000. Kết quả là mô phỏng sẽ lâu hơn 1.000 lần! Chìa khóa để tránh vấn đề này là chuyển đồng hồ đến thời gian sự kiện tiếp theo tại mỗi sân khấu. Đây là khái niệm dễ thực hiện. Tại bất kỳ thời điểm nào, sự kiện tiếp theo có thể xảy ra là (a) khách hàng tiếp theo trong tệp đầu vào đến hoặc (b) một trong các khách hàng tại quầy giao dịch viên rời đi. Vì tất cả các thời điểm mà các sự kiện sẽ xảy ra đều có sẵn, chúng ta chỉ cần tìm sự kiện xảy ra gần nhất trong tương lai và xử lý sự kiện đó. Nếu sự kiện là một chuyến khởi hành, quá trình xử lý bao gồm thu thập số liệu thống kê cho khách hàng khởi hành và kiểm tra hàng (hàng đợi) để xem liệu có khách hàng khác đang đợi hay không. Nếu vậy, chúng tôi thêm khách hàng đó, xử lý bất kỳ số liệu thống kê nào được yêu cầu, tính toán thời gian khách hàng đó sẽ rời đi và thêm lần khởi hành đó vào tập hợp các sự kiện đang chờ xảy ra.

Nếu sự kiện đến, chúng tôi sẽ kiểm tra một giao dịch viên có sẵn. Nếu không có, chúng tôi đặt hàng đến vào hàng (hàng đợi); nếu không, chúng tôi cung cấp cho khách hàng một giao dịch viên, tính toán thời gian khởi hành của khách hàng và thêm thời gian khởi hành vào tập hợp các sự kiện đang chờ xảy ra. Hàng đợi cho khách hàng có thể được thực hiện như một hàng đợi. Vì chúng ta cần tìm sự kiện gần nhất trong tương lai, nên tập hợp các chuyến khởi hành đang chờ xảy ra được sắp xếp theo hàng đợi ưu tiên. Sự kiện tiếp theo do đó là sự xuất hiện tiếp theo hoặc khởi hành tiếp theo (cái nào đến sớm hơn); cả hai đều có sẵn dễ dàng. Sau đó, rất dễ dàng, mặc dù có thể tốn thời gian, để viết các quy trình mô phỏng. Nếu có C khách hàng (và do đó là sự kiện 2C) và k giao dịch viên, thì thời gian chạy của mô phỏng sẽ là O(C log (k + 1)) vì tính toán và xử lý từng sự kiện lấy O(logH), trong đó H = k + 1 là kích thước của đống.

### 3.5 d-Heaps

Các đống nhị phân đơn giản đến mức chúng hầu như luôn được sử dụng khi cần hàng đợi ưu tiên. Một cách tổng quát đơn giản là d-heap, giống hệt như một đống nhị phân ngoại trừ việc tất cả các nút đều có d con (do đó, một đống nhị phân là một đống 2).

Hình 3.19 cho thấy một 3-heap. Lưu ý rằng d-heap nông hơn nhiều so với heap nhị phân, cải thiện thời gian chạy của các lần chèn thành  $O(\log d N)$ . Tuy nhiên, đối với d lớn, phép toán `deleteMin` đắt hơn, bởi vì mặc dù cây nông hơn, nhưng phải tìm được tối thiểu d con, điều này thực hiện phép so sánh  $d - 1$  bằng cách sử dụng thuật toán chuẩn. Điều này làm tăng thời gian cho hoạt động này lên  $O(d \log N)$ . Nếu d là hằng số, tất nhiên cả hai thời gian chạy đều là  $O(\log N)$ . Mặc dù vẫn có thể sử dụng một mảng, nhưng các phép nhân và phép chia để tìm con và bố mẹ hiện bằng d, trừ khi d là lũy thừa của 2, sẽ làm tăng nghiêm trọng thời gian chạy, bởi vì chúng ta không thể thực hiện phép chia theo dịch chuyển bit nữa. d-heaps rất thú vị về mặt lý thuyết, bởi vì có nhiều thuật toán mà số lần chèn là lớn hơn nhiều so với số lượng `deleteMins` (và do đó có thể tăng tốc độ theo lý thuyết). Chúng cũng được quan tâm khi hàng đợi ưu tiên quá lớn để vừa hoàn toàn trong bộ nhớ chính.



Hình 3.19 Một d-heap ( $d = 3$ )

Trong trường hợp này, một d-heap có thể có lợi giống như B-tree. Cuối cùng, ở đó là bằng chứng cho thấy 4-heaps có thể hoạt động tốt hơn heap nhị phân trong thực tế.

Điểm yếu rõ ràng nhất của việc triển khai heap, ngoài việc không thể tìm thấy trên mỗi biểu mẫu, là việc kết hợp hai heap thành một là một thao tác khó. Hoạt động bổ sung này được gọi là hợp nhất. Có khá nhiều cách để triển khai đống sao cho thời gian chạy của một hợp nhất là  $O(\log N)$ . Nay giờ chúng ta sẽ thảo luận về ba cấu trúc dữ liệu, phức tạp khác nhau, hỗ trợ hoạt động hợp nhất một cách hiệu quả.

### 3.6 Hàng đống cánh tả

Có vẻ như rất khó để thiết kế một cấu trúc dữ liệu hỗ trợ hiệu quả việc hợp nhất (nghĩa là ngừng hợp nhất trong  $O(N)$  time) và chỉ sử dụng một mảng, như trong một đống nhị phân. Lý giải cho việc đây là việc hợp nhất dường như sẽ yêu cầu sao chép một mảng này vào một mảng khác, điều này sẽ mất  $(N)$  thời gian cho các đống có kích thước bằng nhau. Vì lý do này, tất cả các cấu trúc dữ liệu nâng cao hỗ trợ hợp nhất hiệu quả yêu cầu sử dụng cấu trúc dữ liệu được liên kết. Trong thực tế, chúng ta có thể mong đợi rằng điều này sẽ làm cho tất cả các hoạt động khác chậm hơn. Giống như một đống nhị phân, một đống cánh tả có cả đặc tính cấu trúc và một đề xuất sắp xếp- sai lầm. Thật vậy, một đống cánh tả, giống như hầu như tất cả các đống được sử dụng, có cùng thuộc tính thứ tự đống chúng tôi đã thấy. Hơn nữa, một đống cánh tả cũng là một cây nhị phân. Sự khác biệt duy nhất giữa một đống cánh tả và một đống nhị phân là đống cánh tả không hoàn toàn cân bằng, nhưng thực sự cố gắng rất mất cân bằng.

#### 3.6.1 Thuộc tính đống cánh tả

Chúng tôi xác định độ dài đường dẫn rỗng, npl ( $X$ ), của bất kỳ nút  $X$  nào là độ dài của đường đi ngắn nhất từ  $X$  đến một nút không có hai nút con. Do đó, npl của một nút không có hoặc một nút con là 0, trong khi npl (nullptr) = -1. Trong cây ở Hình 3.20, độ dài đường dẫn rỗng được chỉ ra bên trong các nút cây. Lưu ý rằng độ dài đường dẫn null của bất kỳ nút nào đều hơn 1 so với độ dài tối thiểu của null độ dài đường đi của con cái của nó. Điều này áp dụng cho các nút có ít hơn hai nút con vì độ dài đường dẫn null của nullptr là -1. Thuộc tính heap bên cánh tả là đối với mọi nút  $X$  trong heap, độ dài đường

dẫn rỗng của đứa con bên trái ít nhất lớn bằng đứa trẻ bên phải. Tài sản này chỉ được hài lòng bởi một trong những cái cây trong Hình 3.20, cụ thể là cây bên trái. Tài sản này thực sự đi cách của nó để đảm bảo rằng cây không cân bằng, bởi vì nó rõ ràng sẽ thiên vị cho cây sâu về phía bên trái. Thật vậy, một cây bao gồm một đường dài của các nút bên trái là có thể (và thực sự thích hợp hơn để tạo điều kiện cho việc hợp nhất) —hence name heap. Bởi vì các nhóm cánh tả có xu hướng có những con đường sâu bên trái, nó theo sau rằng con đường bên phải phải trở nên ngắn. Thật vậy, con đường đúng xuống một đống cánh tả cũng ngắn như bất kỳ con đường nào trong đống đó. Nếu không, sẽ có một đường đi qua một số nút X và lấy nút con bên trái. Sau đó X sẽ xâm phạm tài sản cánh tả.

### Định lý 3.2

Một cây cánh tả có  $r$  nút trên đường đi phải có ít nhất  $2r - 1$  nút.



Hình 3.20 Chiều dài đường dẫn rỗng cho hai cây; chỉ có cây bên trái là cánh tả

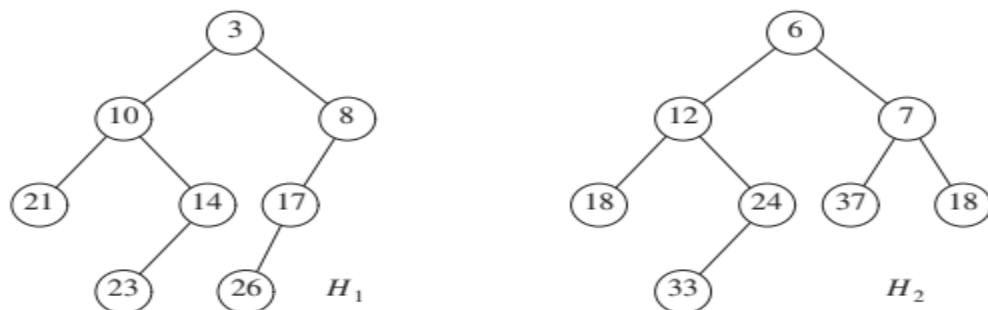
### Bằng chứng

Chứng minh là bằng quy nạp. Nếu  $r = 1$  thì phải có ít nhất một nút cây. Ngược lại, giả sử rằng định lý đúng với  $1, 2, \dots, r$ . Hãy xem xét một cây cánh tả với  $r + 1$  nút trên con đường bên phải. Khi đó gốc có một cây con bên phải với  $r$  nút trên con đường bên phải và một cây con bên trái với ít nhất  $r$  nút trên con đường bên phải (nếu không nó sẽ không thuận). Áp dụng giả thuyết quy nạp cho các cây con này sẽ thu được tối thiểu  $2r - 1$  nút trong mỗi cây con.

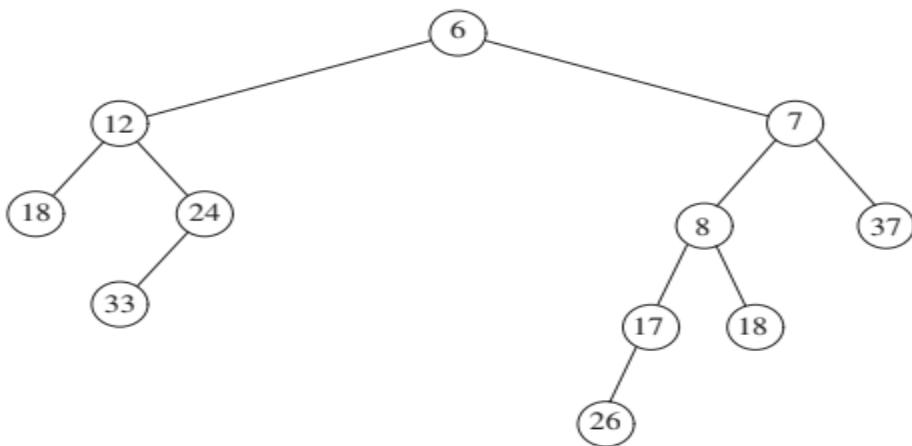
Điều này cộng với gốc tạo ra ít nhất  $2r + 1 - 1$  nút trong cây, chứng minh định lý. Từ định lý này, ngay lập tức có một cây tả N nút có một đường đi bên phải chứa nhiều nhất ( $N + 1$ ) nút. Ý tưởng chung cho các hoạt động heap của cánh tả là thực hiện tất cả công việc trên con đường đúng đắn, được đảm bảo là ngắn. Phần khó duy nhất là việc thực hiện chèn và hợp nhất trên đường có thể phá hủy tài sản đống của phe cánh tả. Nó chỉ ra là cực kỳ dễ dàng để khôi phục tài sản.

### 3.6.2 Hoạt động của nhóm cánh tả

Hoạt động cơ bản trên các nhóm cánh tả đang hợp nhất. Lưu ý rằng chèn chỉ là một trường hợp hợp nhất đặc biệt, vì chúng ta có thể xem việc chèn như là hợp nhất của một đống một nút với một đống lớn hơn. Đầu tiên chúng tôi sẽ đưa ra một giải pháp đệ quy đơn giản và sau đó chỉ ra cách điều này có thể được thực hiện không thường xuyên. Đầu vào của chúng tôi là hai nhóm cánh tả,  $H_1$  và  $H_2$ , trong Hình 3.21. Bạn nên kiểm tra xem những đống này thực sự là cánh tả. Chú ý rằng các phần tử nhỏ nhất nằm ở gốc. Ngoài không gian cho dữ liệu và các con trỏ trái và phải, mỗi nút sẽ có một mục nhập cho biết độ dài đường dẫn rỗng. Nếu một trong hai heap trống, thì chúng ta có thể trả về heap khác. Nếu không, để hợp nhất hai đống, chúng tôi so sánh gốc của chúng. Đầu tiên, chúng ta hợp nhất một cách đệ quy đống với gốc lớn hơn với phần con bên phải của đống với gốc nhỏ hơn. Trong ví dụ của chúng tôi, điều này có nghĩa là chúng tôi hợp nhất đệ quy  $H_2$  với subheap của  $H_1$  bắt nguồn từ 8, thu được heap trong Hình 3.22. Vì cây này được hình thành một cách đệ quy và chúng tôi vẫn chưa hoàn thành mô tả của thuật toán, nên tại thời điểm này, chúng tôi không thể chỉ ra cách thu được đống này. Tuy nhiên nó lại hợp lý khi giả định rằng cây kết quả là một đống thuận,



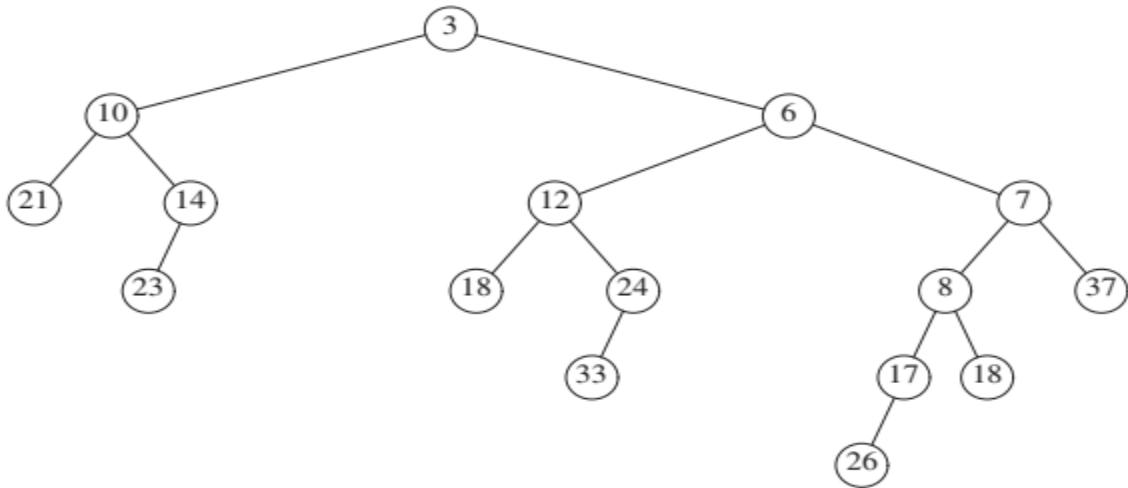
Hình 3.21 Hai nhóm cánh tả  $H_1$  và  $H_2$



Hình 3.22 Kết quả hợp nhất H2 với sơ đồ con bên phải của H1

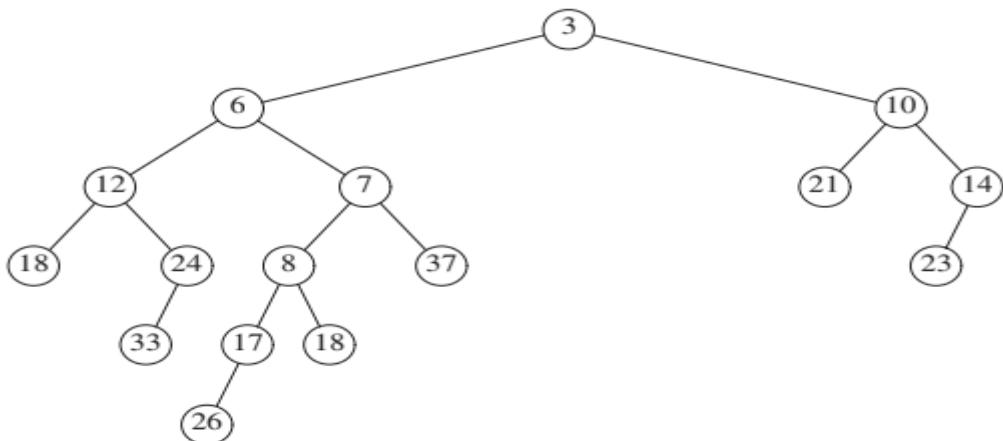
bởi vì nó được lấy thông qua một bước đệ quy. Điều này giống như giả thuyết quy nạp trong một chứng minh bằng quy nạp. Vì chúng ta có thể xử lý trường hợp cơ sở (xảy ra khi một cây trống), chúng ta có thể giả định rằng bước đệ quy hoạt động miễn là chúng ta có thể hoàn thành việc hợp nhất. Bây giờ chúng ta làm cho heap mới này trở thành con bên phải của gốc của H1 (xem Hình 3.23).

Mặc dù đúng kết quả thỏa mãn thuộc tính heap-order, nó không phải là cánh tả vì cây con bên trái của gốc có độ dài đường dẫn rỗng là 1 trong khi cây con bên phải có độ dài đường dẫn rỗng là 2. Do đó, thuộc tính cánh tả bị vi phạm tại nguồn gốc. Tuy nhiên, có thể dễ dàng nhận thấy rằng phần còn lại của cây phải là cánh tả. Cây con bên phải của gốc là bên trái vì bước đệ quy. Cây con bên trái của gốc không bị thay đổi, vì vậy nó cũng phải là cánh tả. Vì vậy, chúng ta chỉ cần sửa chữa gốc. Chúng ta có thể làm cho toàn bộ cây trở nên thuận lợi hơn chỉ bằng cách hoán đổi phần con bên trái và bên phải của gốc (Hình 3.24) và cập nhật độ dài đường dẫn rỗng - độ dài đường dẫn rỗng mới là 1 cộng với độ dài đường dẫn rỗng của cây con bên phải mới - hoàn thành



Hình3.23 Kết quả của việc gắn nhóm cánh tả của hình trước làm con phải của H1

Sự hợp nhất. Lưu ý rằng nếu độ dài đường dẫn null không được cập nhật, thì tất cả độ dài đường dẫn null sẽ là 0, và đống sẽ không phải là cánh tả mà chỉ là ngẫu nhiên. Trong trường hợp này, thuật toán sẽ làm việc, nhưng thời hạn mà chúng tôi yêu cầu sẽ không còn hiệu lực. Mô tả của thuật toán chuyển trực tiếp thành mã. Lớp nút (Hình 3.25) giống với cây nhị phân, ngoại trừ việc nó được tăng cường với npl (null độ dài đường dẫn) thành viên dữ liệu. Heap cánh tả lưu trữ một con trỏ tới gốc làm thành viên dữ liệu của nó. Chúng ta đã thấy trong Chương 2 rằng khi một phần tử được chèn vào cây nhị phân trống,



Hình 3.24 Kết quả hoán đổi con của gốc H1

```

1  template <typename Comparable>
2  class LeftistHeap
3  {
4      public:
5          LeftistHeap( );
6          LeftistHeap( const LeftistHeap & rhs );
7          LeftistHeap( LeftistHeap && rhs );
8
9          ~LeftistHeap( );
10
11         LeftistHeap & operator=( const LeftistHeap & rhs );
12         LeftistHeap & operator=( LeftistHeap && rhs );
13
14         bool isEmpty( ) const;
15         const Comparable & findMin( ) const;
16
17         void insert( const Comparable & x );
18         void insert( Comparable && x );
19         void deleteMin( );
20         void deleteMin( Comparable & minItem );
21         void makeEmpty( );
22         void merge( LeftistHeap & rhs );
23
24     private:
25         struct LeftistNode
26         {
27             Comparable   element;
28             LeftistNode *left;
29             LeftistNode *right;
30             int          npl;
31
32             LeftistNode( const Comparable & e, LeftistNode *lt = nullptr,
33                         LeftistNode *rt = nullptr, int np = 0 )
34                 : element{ e }, left{ lt }, right{ rt }, npl{ np } { }
35
36             LeftistNode( Comparable && e, LeftistNode *lt = nullptr,
37                         LeftistNode *rt = nullptr, int np = 0 )
38                 : element{ std::move( e ) }, left{ lt }, right{ rt }, npl{ np } { }
39         };
40
41         LeftistNode *root;
42
43         LeftistNode * merge( LeftistNode *h1, LeftistNode *h2 );
44         LeftistNode * mergel( LeftistNode *h1, LeftistNode *h2 );
45
46         void swapChildren( LeftistNode *t );
47         void reclaimMemory( LeftistNode *t );
48         LeftistNode * clone( LeftistNode *t ) const;
49     };

```

Hình 3.25 Khai báo kiểu đống cánh tả

Nút được tham chiếu bởi gốc sẽ cần thay đổi. Chúng tôi sử dụng kỹ thuật thông thường là đề cập đến các phương pháp đệ quy riêng để thực hiện hợp nhất. Khung lớp cũng được hiển thị trong Hình 3.25. Hai quy trình hợp nhất (Hình 3.26) là các trình điều khiển được thiết kế để loại bỏ các trường hợp đặc biệt và đảm bảo rằng H1 có gốc nhỏ hơn. Việc hợp nhất thực tế được thực hiện trong merge1 (Hình 3.27). Phương pháp hợp nhất công khai hợp nhất rhs vào đống kiểm soát. rhs trở nên trống rỗng. Các kiểm tra bí danh trong phương pháp công khai không cho phép h.merge (h). Thời gian thực hiện phép hợp nhất tỷ lệ với tổng chiều dài của quyền các đường dẫn, bởi vì công việc liên tục được thực hiện tại mỗi nút được truy cập trong các cuộc gọi đệ quy. Do đó, chúng tôi có được thời gian O (logN) ràng buộc để hợp nhất hai đống cánh tả. Chúng tôi cũng có thể thực hiện hoạt động này không theo quy luật bằng cách thực hiện hai lần về cơ bản. Trong lần vượt qua đầu tiên, chúng tôi tạo một cây mới bằng cách hợp nhất các đường dẫn bên phải của cả hai đống. Để làm điều này, chúng tôi sắp xếp các nút trên các con đường bên phải của H1 và H2 theo thứ tự được sắp xếp, giữ lại các con bên trái tương ứng của chúng. Trong ví dụ của chúng tôi, đường dẫn bên phải mới là 3, 6, 7, 8, 18 và cây kết quả được hiển thị trong Hình 3.28.

```

1      /**
2      * Merge rhs into the priority queue.
3      * rhs becomes empty. rhs must be different from this.
4      */
5      void merge( LeftistHeap & rhs )
6      {
7          if( this == &rhs )    // Avoid aliasing problems
8              return;
9
10         root = merge( root, rhs.root );
11         rhs.root = nullptr;
12     }
13
14 /**
15 * Internal method to merge two roots.
16 * Deals with deviant cases and calls recursive merge1.
17 */
18 LeftistNode * merge( LeftistNode *h1, LeftistNode *h2 )
19 {
20     if( h1 == nullptr )
21         return h2;
22     if( h2 == nullptr )
23         return h1;
24     if( h1->element < h2->element )
25         return merge1( h1, h2 );
26     else
27         return merge1( h2, h1 );
28 }

```

Hình 3.26 Các quy trình thúc đẩy để hợp nhất các nhóm cánh tả

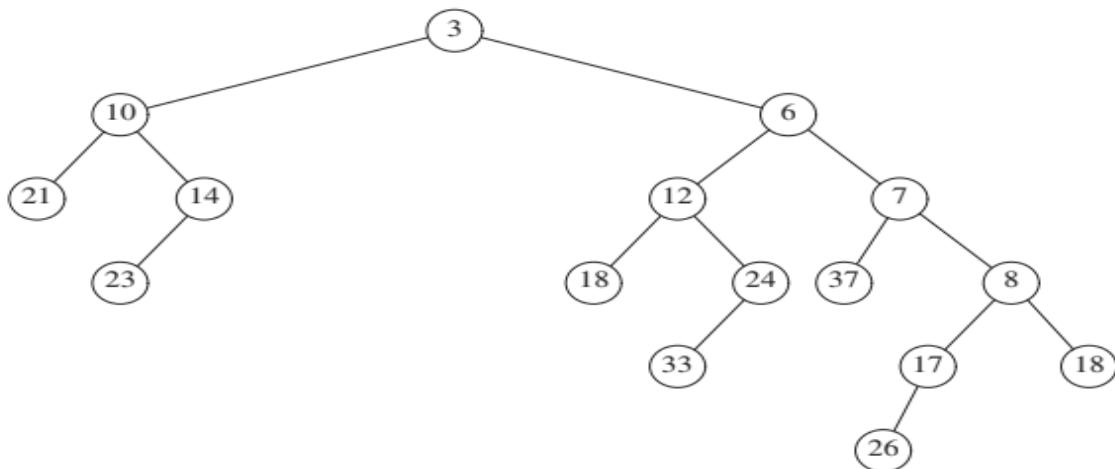
```

1      /**
2       * Internal method to merge two roots.
3       * Assumes trees are not empty, and h1's root contains smallest item.
4       */
5      LeftistNode * merge( LeftistNode *h1, LeftistNode *h2 )
6      {
7          if( h1->left == nullptr )    // Single node
8              h1->left = h2;           // Other fields in h1 already accurate
9          else
10         {
11             h1->right = merge( h1->right, h2 );
12             if( h1->left->npl < h1->right->npl )
13                 swapChildren( h1 );
14             h1->npl = h1->right->npl + 1;
15         }
16         return h1;
17     }

```

Hình 3.27 Quy trình thực tế để hợp nhất các nhóm cánh tả

Một lần vượt qua thứ hai được tạo thành heap và hoán đổi con được thực hiện tại các nút vi phạm tài sản heap của phe cánh tả. Trong hình 3.28, có một sự hoán đổi tại các nút 7 và 3, và cây giống như trước đó được thu được. Phiên bản phi đệ quy đơn giản hơn để hình dung nhưng khó viết mã hơn. Chúng tôi để nó cho người đọc để chỉ ra rằng các thủ tục đệ quy và không đệ quy làm điều tương tự.



Hình 3.28 Kết quả hợp nhất các đường đi phải của H1 và H2

```

1      /**
2       * Inserts x; duplicates allowed.
3       */
4      void insert( const Comparable & x )
5      {
6          root = merge( new LeftistNode{ x }, root );
7      }

```

Hình 3.29 Quy trình chèn ép cho các nhóm cánh tả

Như đã đề cập ở trên, chúng ta có thể thực hiện chèn bằng cách làm cho mục được chèn vào một đống một nút và thực hiện hợp nhất. Để thực hiện một deleteMin, chúng tôi chỉ cần phá hủy gốc, tạo ra hai đống, sau đó có thể được hợp nhất. Do đó, thời gian để thực hiện một deleteMin là O (logN). Hai thói quen này được mã hóa trong Hình 3.29 và Hình 3.30.

Cuối cùng, chúng ta có thể xây dựng một đống cánh tả trong thời gian O (N) bằng cách xây dựng một đống nhị phân (obviously sử dụng triển khai được liên kết). Mặc dù một đống nhị phân rõ ràng là cánh tả, nhưng đây không nhất thiết là giải pháp tốt nhất, bởi vì đống mà chúng ta có được là đống cánh tả tệ nhất có thể. Hơn nữa, việc duyệt cây theo thứ tự cấp độ ngược lại không dễ dàng với các liên kết. Hiệu ứng buildHeap có thể đạt được bằng cách xây dựng đệ quy các cây con bên trái và bên phải, sau đó tô màu gốc xuống. Các bài tập có chứa một giải pháp thay thế.

```

1      /**
2       * Remove the minimum item.
3       * Throws UnderflowException if empty.
4       */
5      void deleteMin( )
6      {
7          if( isEmpty( ) )
8              throw UnderflowException{ };
9
10         LeftistNode *oldRoot = root;
11         root = merge( root->left, root->right );
12         delete oldRoot;
13     }
14
15     /**
16      * Remove the minimum item and place it in minItem.
17      * Throws UnderflowException if empty.
18      */
19     void deleteMin( Comparable & minItem )
20     {
21         minItem = findMin( );
22         deleteMin( );
23     }

```

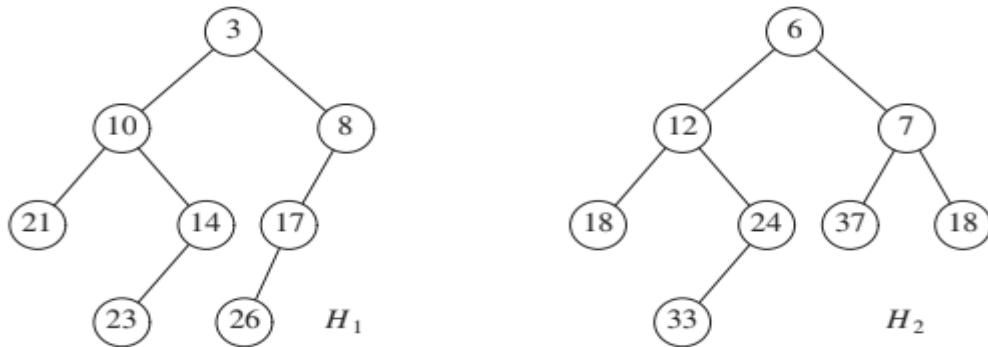
Hình 3.30 Quy trình xóaMin cho các nhóm cánh tả

### 3.7 Đống xiên

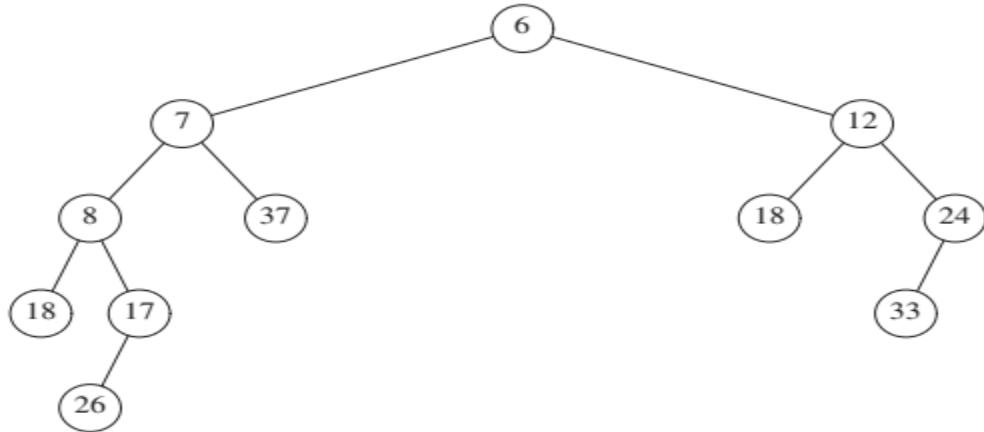
Một đống xiên là một phiên bản tự điều chỉnh của một đống cánh tả cực kỳ đơn giản để thực hiện- cỗ vấn. Mỗi quan hệ của đống xiên với đống cánh tả tương tự như mối quan hệ giữa cây splay và cây AVL. Skew heaps là cây nhị phân với thứ tự đống, nhưng không có hạn chế về cấu trúc đối với những cây này. Không giống như các nhóm cánh tả, không có thông tin nào được duy trì về độ dài đường dẫn rỗng của bất kỳ nút nào. Đường đi bên phải của một đống xiên có thể tùy ý- tôi dài bất cứ lúc nào, vì vậy thời gian chạy trong trường hợp xấu nhất của tất cả các hoạt động là  $O(N)$ . Giống như với các nhóm cánh tả, hoạt động cơ bản trên các nhóm xiên là hợp nhất. Hợp nhất quy trình một lần nữa là đệ quy và chúng tôi thực hiện các hoạt động chính xác giống như trước đây, với một ngoại lệ. Sự khác biệt là đối với các nhóm cánh tả, chúng tôi kiểm tra xem liệu bên trái và những đứa trẻ bên phải thỏa mãn tài sản cấu trúc đống cánh tả và hoán đổi chúng nếu chúng không làm như vậy. Đối với các đống xiên, việc hoán đổi là vô điều kiện; chúng tôi luôn làm điều đó, với một ngoại lệ nút lớn nhất trong tất cả các nút trên đường đi bên phải không có con của nó được hoán đổi. Điều này một ngoại lệ là những gì xảy ra trong quá trình triển khai đệ quy tự nhiên, vì vậy nó không thực sự một trường hợp đặc biệt. Hơn nữa, không cần thiết phải chứng minh các giới hạn, nhưng vì điều này nút được đảm bảo không có nút con phù hợp, sẽ thật ngớ ngẩn nếu thực hiện hoán đổi và cho nó một. (Trong ví dụ của chúng tôi, không có nút con nào của nút này, vì vậy chúng tôi không lo lắng về nó.) Một lần nữa, giả sử đầu vào của chúng ta là hai heap giống như trước đây, Hình 3.31. Nếu chúng ta hợp nhất đệ quy H2 với subheap của H1 bắt nguồn từ 8, chúng ta sẽ nhận được heap trong Hình 3.32. Một lần nữa, điều này được thực hiện một cách đệ quy, vì vậy theo quy tắc đệ quy thứ ba chúng ta cần không phải lo lắng về cách nó được lấy. Nhóm này tình cờ là cánh tả, nhưng không có đảm bảo rằng đây luôn là trường hợp. Chúng tôi làm cho đống này trở thành con bên trái mới của H1, và con trái cũ của H1 trở thành con bên phải mới (xem Hình 3.33).

Toàn bộ cây đều là cánh tả, nhưng có thể dễ dàng thấy rằng điều đó không phải lúc nào cũng đúng: Chèn 15 vào đống mới này sẽ phá hủy tài sản của cánh tả. Chúng ta có thể thực hiện tất cả các thao tác một cách không thường xuyên, như với các heaps bên cánh tả, bằng cách hợp nhất các đường dẫn

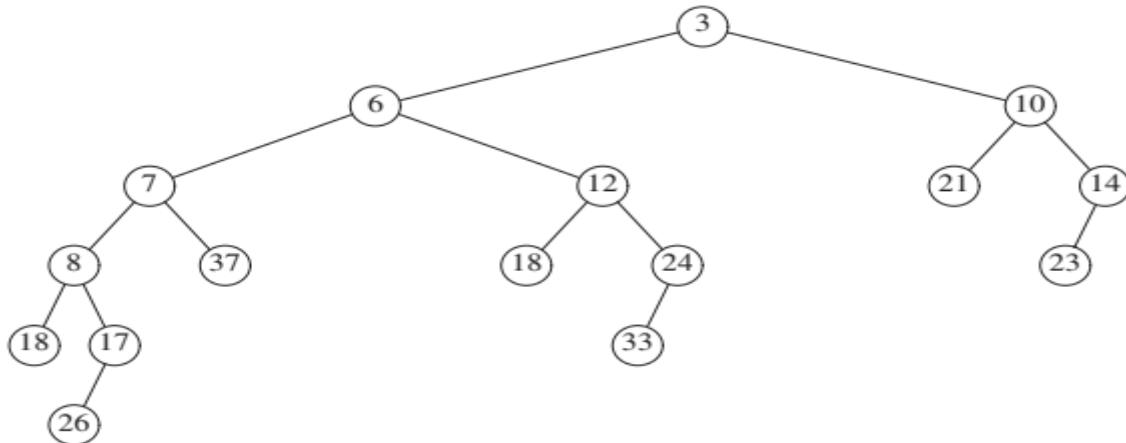
bên phải và hoán đổi các nút con trái và phải cho mọi nút trên đường dẫn bên phải, ngoại trừ con nút cuối cùng.



Hình 3.31 Hai đống xiên  $H_1$  và  $H_2$



Hình 3.32 Kết quả hợp nhất  $H_2$  với sơ đồ con bên phải của  $H_1$



Hình 3.33 Kết quả của việc hợp nhất các đống xiên  $H_1$  và  $H_2$

Sau một vài ví dụ, rõ ràng là vì tất cả trừ trường hợp cuối cùng nút trên con đường bên phải đã được hoán đổi vị trí con của chúng, kết quả thực là nút này trở thành con đường bên trái mới (xem ví dụ trước để thuyết phục bản thân). Điều này làm cho nó rất dễ dàng hợp nhất hai đống xiên một cách trực quan. Điều này không hoàn toàn giống với việc triển khai đệ quy (nhưng mang lại cùng giới hạn thời gian). Nếu chúng ta chỉ hoán đổi các nút con cho các nút trên đường đúng phía trên điểm hợp nhất các đường bên phải bị chấm dứt do hết đường dẫn bên phải của một đống, chúng tôi nhận được kết quả tương tự như phiên bản đệ quy.

Việc thực hiện các đống xiên được để lại như một bài tập (tầm thường). Lưu ý rằng vì a đường dẫn bên phải có thể dài, triển khai đệ quy có thể thất bại vì thiếu ngăn xếp không gian, mặc dù hiệu suất sẽ được chấp nhận. Skew heaps có lợi thế là không cần thêm không gian để duy trì độ dài đường dẫn và không cần kiểm tra để xác định thời điểm hoán đổi con cái. Đó là một vấn đề mở để xác định chính xác chiều dài con đường bên phải dự kiến của cả hai bên cánh tả và xiên (cái sau chắc chắn là nhiều hơn khó khăn). So sánh như vậy sẽ giúp dễ dàng xác định xem liệu sự mất mát nhỏ của thông tin cân bằng được bù đắp bởi việc thiếu thử nghiệm.

### 3.8 Hàng đợi nhị thức

Mặc dù cả hai phe cánh tả và xiên đều hỗ trợ hợp nhất, chèn và xóa tích cực theo thời gian O ( $\log N$ ) cho mỗi hoạt động, vẫn có khả năng cải thiện vì chúng tôi biết đống nhị phân đó hỗ trợ chèn trong thời gian trung bình không đổi cho mỗi hoạt động. Hàng đợi nhị thức hỗ trợ tất cả ba hoạt động trong trường hợp xấu nhất O ( $\log N$ ) thời gian cho mỗi hoạt động, nhưng việc chèn thời gian trung bình không đổi.

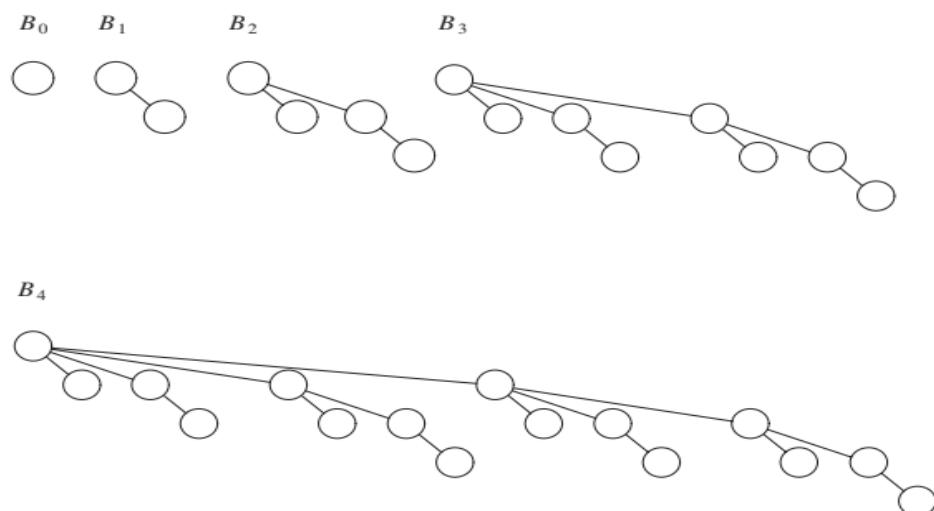
#### 3.8.1 Cấu trúc hàng đợi nhị thức

Hàng đợi nhị thức khác với tất cả các triển khai hàng đợi ưu tiên mà chúng ta đã thấy trong rằng một hàng đợi nhị thức không phải là một cây có thứ tự đống mà là một tập hợp các cây, được gọi là rừng. Mỗi cây được sắp xếp theo thứ tự đống đều có dạng hạn chế đã biết như một cây nhị thức (lý do cho cái tên sẽ rõ ràng ở phần sau). Có nhiều nhất môt cây nhị thức của mọi chiều cao. Cây nhị thức có chiều cao 0 là cây một nút; một nhị thức cây, Bk, có chiều cao k được hình thành bằng cách gắn một cây nhị thức, Bk – 1, vào

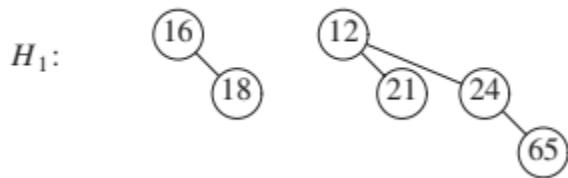
gốc của cây khác cây nhị thức,  $B_{k-1}$ . Hình 3.34 cho thấy cây nhị thức  $B_0$ ,  $B_1$ ,  $B_2$ ,  $B_3$  và  $B_4$ . Từ sơ đồ, chúng ta thấy rằng một cây nhị thức,  $B_k$ , bao gồm một gốc có con  $B_0, B_1, \dots, B_{k-1}$ . Cây nhị thức có chiều cao  $k$  có đúng  $2^k$  nút và số các nút ở độ sâu  $d$  là hệ số nhị thức  $\binom{k}{d}$ . Nếu chúng ta áp đặt thứ tự đồng cho nhị thức cây và cho phép nhiều nhất một cây nhị thức có chiều cao bất kỳ, chúng tôi có thể biểu diễn một hàng đợi ưu tiên có kích thước bất kỳ bởi một tập hợp các cây nhị thức. Ví dụ: hàng đợi ưu tiên có kích thước 13 có thể được đại diện bởi rừng  $B_3, B_2, B_0$ . Chúng tôi có thể viết đại diện này là 1101, không chỉ đại diện cho 13 trong hệ nhị phân mà còn đại diện cho thực tế là  $B_3, B_2$  và  $B_0$  có mặt trong biểu diễn và  $B_1$  thì không. Ví dụ, một hàng đợi ưu tiên gồm sáu phần tử có thể được biểu diễn như trong Hình 3.35.

### 3.8.2 Hoạt động hàng đợi nhị thức

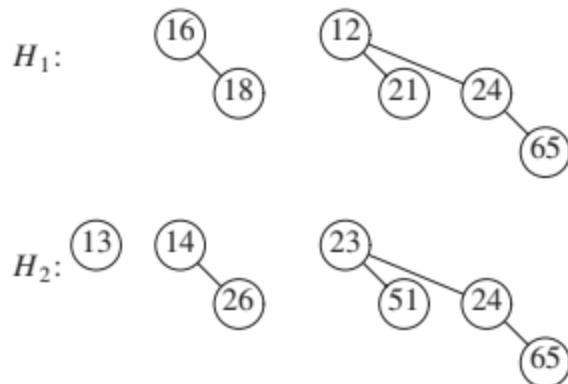
Sau đó, phần tử tối thiểu có thể được tìm thấy bằng cách quét rẽ của tất cả các cây. Từ đó là nhiều nhất  $\log N$  cây khác nhau, tối thiểu có thể được tìm thấy trong thời gian  $O(\log N)$ . Ngoài ra, chúng tôi có thể duy trì kiến thức về mức tối thiểu và thực hiện hoạt động trong  $O(1)$  thời gian nếu chúng tôi nhớ cập nhật mức tối thiểu khi nó thay đổi trong các hoạt động khác. Hợp nhất hai hàng đợi nhị thức là một hoạt động dễ dàng về mặt khái niệm, chúng tôi sẽ mô tả ví dụ như. Hãy xem xét hai hàng đợi nhị thức,  $H_1$  và  $H_2$ , với sáu và bảy phần tử, tương ứng như hình 3.36.



Hình 3.34 Cây nhị thức  $B_0, B_1, B_2, B_3$  và  $B_4$



Hình 3.35 Hàng đợi nhị thức  $H_1$  với sáu phần tử



Hình 3.36 Hai hàng đợi nhị thức  $H_1$  và  $H_2$

Việc hợp nhất được thực hiện bằng cách thêm hai hàng đợi với nhau. Hãy để  $H_3$  là hàng đợi nhị thức mới. Vì  $H_1$  không có cây nhị thức có chiều cao bằng 0 và  $H_2$  có, chúng ta có thể chỉ cần sử dụng cây nhị thức có chiều cao 0 trong  $H_2$  như một phần của  $H_3$ . Tiếp theo, chúng tôi thêm cây nhị thức chiều cao 1. Vì cả  $H_1$  và  $H_2$  đều có cây nhị thức có chiều cao 1, chúng tôi hợp nhất chúng bằng làm cho gốc lớn hơn trở thành cây con của cây nhỏ hơn, tạo cây nhị thức có chiều cao 2, được hiển thị trong Hình 3.37. Như vậy,  $H_3$  sẽ không có cây nhị thức có chiều cao 1. Nay giờ có ba cây nhị thức có chiều cao 2, cụ thể là cây ban đầu của  $H_1$  và  $H_2$  cộng với cây tạo thành bằng bước trước. Chúng tôi giữ một cây nhị thức có chiều cao 2 trong  $H_3$  và hợp nhất cây còn lại hai, tạo một cây nhị thức có chiều cao 3. Vì  $H_1$  và  $H_2$  không có cây nào có chiều cao 3, điều này cây trở thành một phần của  $H_3$  và chúng ta đã hoàn thành. Hàng đợi nhị thức kết quả được hiển thị trong Hình 3.38.

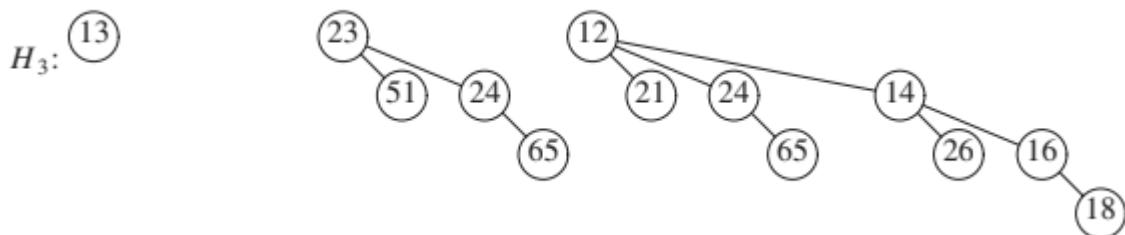
Vì việc hợp nhất hai cây nhị thức cần thời gian không đổi với hầu hết mọi triển khai và có cây nhị thức  $O(\log N)$ , quá trình hợp nhất mất  $O(\log N)$  thời gian trong trường hợp xấu nhất. Để làm cho hoạt động này hiệu quả, chúng ta cần giữ các cây trong hệ nhị thức hàng đợi được sắp xếp theo chiều cao, đó chắc chắn là một điều đơn giản để làm. Chèn chỉ là một trường hợp hợp nhất đặc biệt, vì chúng ta chỉ tạo một cây một nút và thực hiện hợp nhất. Thời gian trong trường hợp xấu nhất của thao tác này cũng là  $O(\log N)$ . Chính xác là, nếu hàng đợi ưu tiên mà phần tử đang được chèn vào có thuộc tính rằng cây nhị thức không tồn tại nhỏ nhất là  $B_i$ , thời gian chạy tỷ lệ với  $i + 1$ .

Ví dụ:  $H_3$  (Hình 3.38) thiếu cây nhị thức có chiều cao 1, vì vậy phần chèn sẽ chấm dứt trong hai bước. Vì mỗi cây trong hàng đợi nhị thức đều có xác suất  $1/2$ , theo đó chúng tôi mong đợi việc chèn kết thúc trong hai bước, vì vậy thời gian trung bình là hằng số. Hơn nữa, một phân tích sẽ chỉ ra rằng việc thực hiện  $N$  lần chèn trên một hàng đợi nhị thức trống sẽ chiếm  $O(N)$  thời gian trong trường hợp xấu nhất. Thật vậy, có thể làm được điều này hoạt động chỉ sử dụng  $N - 1$  so sánh; chúng tôi để điều này như một bài tập.

Ví dụ, chúng tôi chỉ ra trong Hình 3.39 đến 3.45 các hàng đợi nhị thức được hình thành bằng cách chèn từ 1 đến 7 theo thứ tự. Chèn 4 cho thấy một trường hợp xấu. Chúng tôi hợp nhất 4



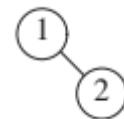
Hình 3.37 Hợp nhất hai cây  $B_1$  trong  $H_1$  và  $H_2$



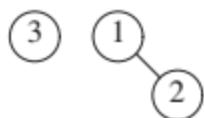
Hình 3.38 Hàng đợi nhị thức  $H_3$ : kết quả của việc hợp nhất  $H_1$  và  $H_2$

(1)

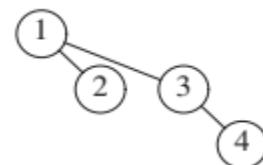
Hình 3.39 Sau khi chèn 1



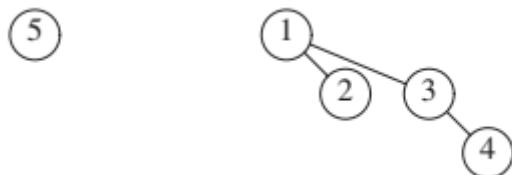
Hình 3.40 Sau khi 2 được lắp vào



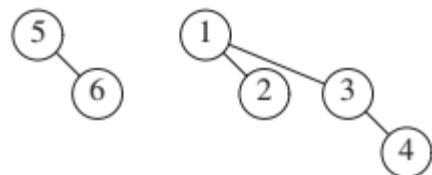
Hình 3.41 Sau khi 3 được lắp vào



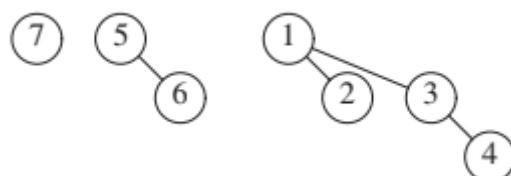
Hình 3.42 Sau khi 4 được lắp vào



Hình 3.43 Sau khi 5 được lắp vào



Hình 3.44 Sau khi chèn 6

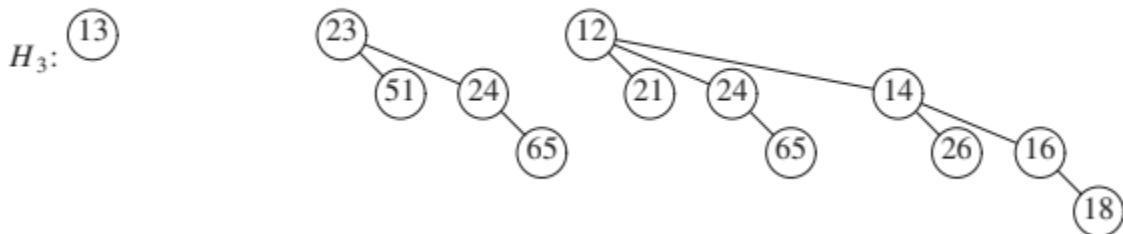


Hình 3.45 Sau khi 7 được chèn vào

với  $B_0$ , thu được một cây mới có chiều cao 1. Sau đó, chúng tôi hợp nhất cây này với  $B_1$ , thu được cây có chiều cao 2, là hàng đợi ưu tiên mới. Chúng tôi tính đây là ba bước (hai cây hợp nhất cộng với trường hợp dừng). Lần chèn tiếp theo sau khi 7 được chèn là một trường hợp xấu khác và sẽ yêu cầu hợp nhất ba cây. Một `deleteMin` có thể được thực hiện bằng cách đầu tiên tìm cây nhị thức có căn nhỏ nhất. Đặt cây này là  $B_k$  và đặt hàng đợi ưu tiên ban đầu là  $H$ . Chúng ta loại bỏ cây nhị thức  $B_k$  từ rừng cây trong  $H$ , tạo thành hàng đợi nhị thức mới  $H$ . Chúng tôi cũng xóa gốc của  $B_k$ , tạo cây nhị thức  $B_0$ ,

$B_1, \dots, B_{k-1}$ , chúng tạo thành hàng đợi ưu tiên chung  $H$ . Chúng tôi kết thúc hoạt động bằng cách hợp nhất  $H$  và  $H'$ .

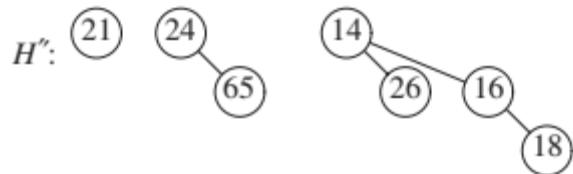
Ví dụ: giả sử chúng ta thực hiện một `deleteMin` trên  $H_3$ , được hiển thị lại trong Hình 3.46. Gốc tối thiểu là 12, vì vậy chúng tôi có được hai hàng đợi ưu tiên  $H$  và  $H'$  trong Hình 3.47 và Hình 3.48. Hàng đợi nhị thức là kết quả của việc hợp nhất  $H$  và  $H'$  là câu trả lời cuối cùng và được thể hiện trong Hình 3.49. Đối với phân tích, trước tiên hãy lưu ý rằng thao tác `deleteMin` phá vỡ nhị thức ban đầu xếp hàng thành hai. Phải mất  $O(\log N)$  thời gian để tìm cây chứa phần tử tối thiểu và để tạo hàng đợi  $H$  và  $H'$ . Việc hợp nhất hai hàng đợi này cần thời gian  $O(\log N)$ , vì vậy toàn bộ hoạt động `deleteMin` mất  $O(\log N)$  thời gian.



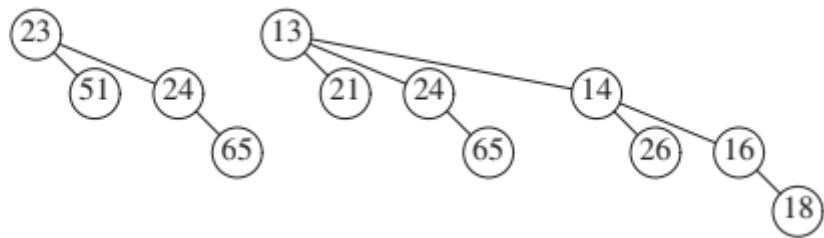
Hình 3.46 Hàng đợi nhị thức  $H_3$



Hình 3.47 Hàng đợi nhị thức  $H$ , chứa tất cả các cây nhị thức trong  $H_3$  ngoại trừ  $B_3$



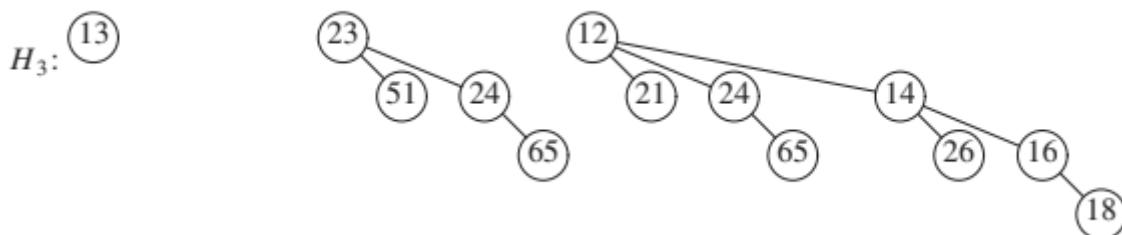
Hình 3.48 Hàng đợi nhị thức  $H$ :  $B_3$  với 12 bị loại bỏ



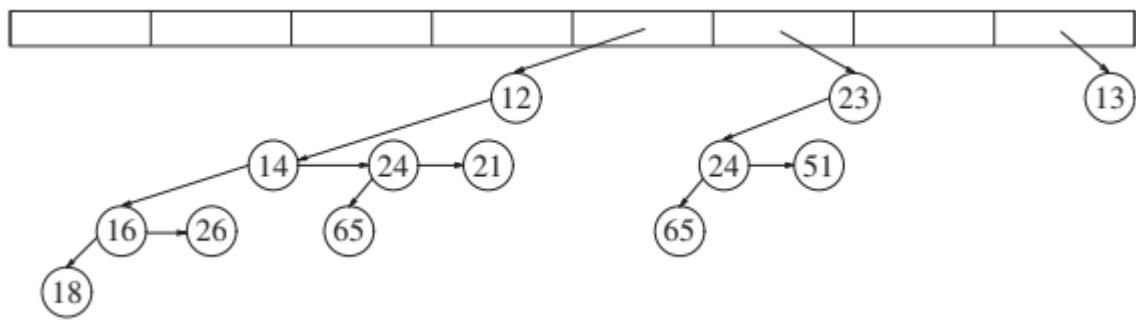
Hình 3.49 Kết quả áp dụng deleteMin cho  $H_3$

### 3.8.3 Triển khai hàng đợi nhị thức

Thao tác deleteMin yêu cầu khả năng tìm tất cả các cây con của gốc một cách nhanh chóng, vì vậy biểu diễn tiêu chuẩn của cây chung là bắt buộc: Các nút con của mỗi nút là được giữ trong danh sách liên kết và mỗi nút có một con trỏ đến con đầu tiên của nó (nếu có). Hoạt động này cũng yêu cầu các con phải được sắp xếp theo kích thước của các cây con của chúng. Chúng ta cũng cần đảm bảo rằng có thể dễ dàng hợp nhất hai cây. Khi hai cây được hợp nhất, một trong những cây được thêm vào như một đứa trẻ với người kia. Vì cây mới này sẽ là cây con lớn nhất, nó làm cho ý thức để duy trì các cây con với kích thước giảm dần. Chỉ khi đó chúng ta mới có thể hợp nhất hai cây nhị thức, và do đó hai hàng đợi nhị thức, một cách hiệu quả. Hàng đợi nhị thức sẽ là một mảng cây nhị thức. Để tóm tắt, sau đó, mỗi nút trong cây nhị thức sẽ chứa dữ liệu, con đầu tiên và đúng anh chị em ruột. Các con trong cây nhị thức được sắp xếp theo thứ tự giảm dần. Hình 3.51 cho thấy cách biểu diễn hàng đợi nhị thức trong Hình 3.50. Hình 3.52 hiển thị các khai báo kiểu cho một nút trong cây nhị thức và lớp hàng đợi nhị thức giao diện.



Hình 3.50 Hàng đợi nhị thức  $H_3$  được vẽ như một khu rừng



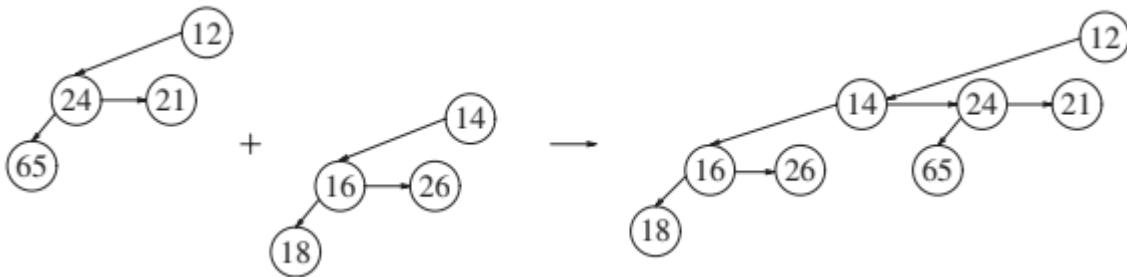
Hình 3.51 Biểu diễn hàng đợi nhị thức  $H_3$

```

1  template <typename Comparable>
2  class BinomialQueue
3  {
4      public:
5          BinomialQueue( );
6          BinomialQueue( const Comparable & item );
7          BinomialQueue( const BinomialQueue & rhs );
8          BinomialQueue( BinomialQueue && rhs );
9
10         ~BinomialQueue( );
11
12         BinomialQueue & operator=( const BinomialQueue & rhs );
13         BinomialQueue & operator=( BinomialQueue && rhs );
14
15         bool isEmpty( ) const;
16         const Comparable & findMin( ) const;
17
18         void insert( const Comparable & x );
19         void insert( Comparable && x );
20         void deleteMin( );
21         void deleteMin( Comparable & minItem );
22
23         void makeEmpty( );
24         void merge( BinomialQueue & rhs );
25
26     private:
27         struct BinomialNode
28         {
29             Comparable   element;
30             BinomialNode *leftChild;
31             BinomialNode *nextSibling;
32
33             BinomialNode( const Comparable & e, BinomialNode *lt, BinomialNode *rt )
34                 : element{ e }, leftChild{ lt }, nextSibling{ rt } { }
35
36             BinomialNode( Comparable && e, BinomialNode *lt, BinomialNode *rt )
37                 : element{ std::move( e ) }, leftChild{ lt }, nextSibling{ rt } { }
38         };
39
40         const static int DEFAULT_TREES = 1;
41
42         vector<BinomialNode *> theTrees; // An array of tree roots
43         int currentSize;                // Number of items in the priority queue
44
45         int findMinIndex( ) const;
46         int capacity( ) const;
47         BinomialNode * combineTrees( BinomialNode *t1, BinomialNode *t2 );
48         void makeEmpty( BinomialNode * & t );
49         BinomialNode * clone( BinomialNode * t ) const;
50     };

```

Hình 3.52 Giao diện lớp hàng đợi nhị thức và định nghĩa nút



Hình 3.53 Hợp nhặt hai cây nhị thức

Để hợp nhặt hai hàng đợi nhị thức, chúng ta cần một quy trình để hợp nhặt hai cây nhị thức có cùng kích thước. Hình 3.53 cho thấy các liên kết thay đổi như thế nào khi hai cây nhị thức đã hợp nhặt. Mã để thực hiện việc này rất đơn giản và được thể hiện trong Hình 3.54. Chúng tôi cung cấp một triển khai đơn giản của quy trình hợp nhặt.  $H_1$  được đại diện bởi đối tượng hiện tại và  $H_2$  được biểu diễn bằng rhs. Quy trình kết hợp  $H_1$  và  $H_2$ , đặt kết quả là  $H_1$  và làm cho  $H_2$  trống. Tại bất kỳ thời điểm nào chúng ta đang xử lý các cây cấp i.  $t_1$  và  $t_2$  lần lượt là các cây trong  $H_1$  và  $H_2$ , và carry là cây được mang từ trước bước (nó có thể là nullptr). Tùy thuộc vào từng trường hợp trong số tám trường hợp có thể xảy ra, cây kết quả cho hạng i và cây mang hạng  $i + 1$  được hình thành. Quá trình này tiến hành từ xếp hạng 0 đến hạng cuối cùng trong hàng đợi nhị thức kết quả. Mã được hiển thị trong Hình 3.55. Các cải tiến đối với mã được đề xuất trong Bài tập 3.35. Quy trình deleteMin cho hàng đợi nhị thức được đưa ra trong Hình 3.56. Chúng tôi có thể mở rộng hàng đợi nhị thức để hỗ trợ một số hoạt động không chuẩn mà đồng nhị phân cho phép, chẳng hạn như ReduceKey và remove, khi vị trí của yếu tố bị ảnh hưởng được biết đến. A ReduceKey là một percolateUp, có thể được thực hiện trong O ( $\log N$ ) thời gian nếu chúng ta thêm một thành viên dữ liệu vào mỗi nút lưu trữ liên kết mèo. An tùy ý xóa có thể được thực hiện bằng sự kết hợp của ReduceKey và deleteMin trong O ( $\log N$ ) thời gian.

```

1      /**
2       * Return the result of merging equal-sized t1 and t2.
3       */
4      BinomialNode * combineTrees( BinomialNode *t1, BinomialNode *t2 )
5      {
6          if( t2->element < t1->element )
7              return combineTrees( t2, t1 );
8          t2->nextSibling = t1->leftChild;
9          t1->leftChild = t2;
10         return t1;
11     }

```

Hình 3.54 Quy trình hợp nhất hai cây nhị thức có kích thước bằng nhau

```

1      /**
2       * Merge rhs into the priority queue.
3       * rhs becomes empty. rhs must be different from this.
4       * Exercise 6.35 needed to make this operation more efficient.
5       */
6      void merge( BinomialQueue & rhs )
7      {
8          if( this == &rhs )    // Avoid aliasing problems
9              return;
10
11         currentSize += rhs.currentSize;
12
13         if( currentSize > capacity( ) )
14         {
15             int oldNumTrees = theTrees.size( );
16             int newNumTrees = max( theTrees.size( ), rhs.theTrees.size( ) ) + 1;
17             theTrees.resize( newNumTrees );
18             for( int i = oldNumTrees; i < newNumTrees; ++i )
19                 theTrees[ i ] = nullptr;
20         }
21
22         BinomialNode *carry = nullptr;
23         for( int i = 0, j = 1; j <= currentSize; ++i, j *= 2 )
24         {
25             BinomialNode *t1 = theTrees[ i ];
26             BinomialNode *t2 = i < rhs.theTrees.size( ) ? rhs.theTrees[ i ]
27                                         : nullptr;
28             int whichCase = t1 == nullptr ? 0 : 1;
29             whichCase += t2 == nullptr ? 0 : 2;
30             whichCase += carry == nullptr ? 0 : 4;
31
32             switch( whichCase )
33             {
34                 case 0: /* No trees */
35                 case 1: /* Only this */
36                     break;
37                 case 2: /* Only rhs */
38                     theTrees[ i ] = t2;
39                     rhs.theTrees[ i ] = nullptr;
40                     break;
41                 case 4: /* Only carry */
42                     theTrees[ i ] = carry;
43                     carry = nullptr;
44                     break;

```

```

45         case 3: /* this and rhs */
46             carry = combineTrees( t1, t2 );
47             theTrees[ i ] = rhs.theTrees[ i ] = nullptr;
48             break;
49         case 5: /* this and carry */
50             carry = combineTrees( t1, carry );
51             theTrees[ i ] = nullptr;
52             break;
53         case 6: /* rhs and carry */
54             carry = combineTrees( t2, carry );
55             rhs.theTrees[ i ] = nullptr;
56             break;
57         case 7: /* All three */
58             theTrees[ i ] = carry;
59             carry = combineTrees( t1, t2 );
60             rhs.theTrees[ i ] = nullptr;
61             break;
62     }
63 }
64
65     for( auto & root : rhs.theTrees )
66     {
67         root = nullptr;
68     }

```

Hình 3.55 Quy trình hợp nhất hai hàng đợi ưu tiên

```

1      /**
2      * Remove the minimum item and place it in minItem.
3      * Throws UnderflowException if empty.
4      */
5      void deleteMin( Comparable & minItem )
6      {
7          if( isEmpty( ) )
8              throw UnderflowException{ };
9
10         int minIndex = findMinIndex( );
11         minItem = theTrees[ minIndex ]->element;
12

```

```

13     BinomialNode *oldRoot = theTrees[ minIndex ];
14     BinomialNode *deletedTree = oldRoot->leftChild;
15     delete oldRoot;
16
17     // Construct H'
18     BinomialQueue deletedQueue;
19     deletedQueue.theTrees.resize( minIndex + 1 );
20     deletedQueue.currentSize = ( 1 << minIndex ) - 1;
21     for( int j = minIndex - 1; j >= 0; --j )
22     {
23         deletedQueue.theTrees[ j ] = deletedTree;
24         deletedTree = deletedTree->nextSibling;
25         deletedQueue.theTrees[ j ]->nextSibling = nullptr;
26     }
27
28     // Construct H'
29     theTrees[ minIndex ] = nullptr;
30     currentSize -= deletedQueue.currentSize + 1;
31
32     merge( deletedQueue );
33 }
34
35 /**
36 * Find index of tree containing the smallest item in the priority queue.
37 * The priority queue must not be empty.
38 * Return the index of tree containing the smallest item.
39 */
40 int findMinIndex( ) const
41 {
42     int i;
43     int minIndex;
44
45     for( i = 0; theTrees[ i ] == nullptr; ++i )
46         ;
47
48     for( minIndex = i; i < theTrees.size( ); ++i )
49         if( theTrees[ i ] != nullptr &&
50             theTrees[ i ]->element < theTrees[ minIndex ]->element )
51             minIndex = i;
52
53     return minIndex;
54 }
```

Hình 3.56 deleteMin cho hàng đợi nhị thức

```

1 #include <iostream>
2 #include <vector>
3 #include <queue>
4 #include <functional>
5 #include <string>
6 using namespace std;
7
8 // Empty the priority queue and print its contents.
9 template <typename PriorityQueue>
10 void dumpContents( const string & msg, PriorityQueue & pq )
11 {
12     cout << msg << ":" << endl;
13     while( !pq.empty( ) )
14     {
15         cout << pq.top( ) << endl;
16         pq.pop( );
17     }
18 }
19
20 // Do some inserts and removes (done in dumpContents).
21 int main( )
22 {
23     priority_queue<int>                         maxPQ;
24     priority_queue<int,vector<int>,greater<int>> minPQ;
25
26     minPQ.push( 4 ); minPQ.push( 3 ); minPQ.push( 5 );
27     maxPQ.push( 4 ); maxPQ.push( 3 ); maxPQ.push( 5 );
28
29     dumpContents( "minPQ", minPQ );      // 3 4 5
30     dumpContents( "maxPQ", maxPQ );      // 5 4 3
31
32     return 0;
33 }

```

Hình 3.57 Quy trình thể hiện STL priority\_queue

### 3.9 Hàng đợi Ưu tiên trong Thư viện Chuẩn

Heap nhị phân được triển khai trong STL bởi mẫu lớp có tên là priority\_queue được tìm thấy trong hàng đợi tiêu đề chuẩn. STL thực hiện max-heap thay vì min- đống để mục lớn nhất thay vì nhỏ nhất là mục được truy cập. Thành viên chủ chốt chức năng là:

```
void push( const Object & x );
const Object & top( ) const;
void pop( );
bool empty( );
void clear( );
```

push thêm x vào hàng đợi ưu tiên, top trả về phần tử lớn nhất trong hàng đợi ưu tiên, và pop xóa phần tử lớn nhất khỏi hàng đợi ưu tiên. Các bản sao được cho phép; nếu có một số phần tử lớn nhất, chỉ một trong số chúng bị loại bỏ. Mẫu hàng đợi ưu tiên được khởi tạo với một loại mục, loại vùng chứa (hầu như lúc nào bạn cũng muốn sử dụng một vectơ lưu trữ các mục) và bộ so sánh; Giá trị mặc định được cho phép cho hai tham số cuối cùng và các giá trị mặc định mang lại một đống tối đa. Sử dụng một đối tượng chức năng lớn hơn khi bộ so sánh tạo ra một min-heap. Hình 3.57 cho thấy một chương trình thử nghiệm minh họa cách thức mẫu lớp priority\_queue có thể được sử dụng làm cả đống tối đa mặc định và một đống tối thiểu.

#### Tóm lại

Trong phần này, chúng ta đã thấy nhiều cách triển khai và sử dụng ADT của hàng đợi ưu tiên.

Việc triển khai heap nhị phân tiêu chuẩn rất dễ dàng vì tính đơn giản và nhanh chóng của nó.

Nó không yêu cầu liên kết và chỉ có một lượng không gian bổ sung không đổi, nhưng hỗ trợ mức độ ưu tiên hoạt động hàng đợi một cách hiệu quả.

## CHƯƠNG 2: NỘI DUNG ĐỀ TÀI

### GAME FLAPPY BIRD

#### 1. Lý do chọn đề tài

Hiện nay đa số mỗi chúng ta ai cũng đã từng tiếp xúc với thế giới ảo nhằm mục đích giải trí. Các Game hiện nay có xu hướng giúp người chơi tăng khả năng tính toán, khả năng xử lý vấn đề, tình huống... Đặc biệt hơn là game sẽ giúp chúng ta bớt căng thẳng sau giờ học, làm việc mệt mỏi. Vì vậy nhóm em cũng xin góp phần làm 1 game để đáp ứng nhu cầu giải trí của người dùng. Game có lối chơi đơn giản, dễ tiếp cận và cần sự khéo léo, tính toán chính xác từ người chơi để đạt được thành tích cao nhất.

#### 2. Ý tưởng làm đề tài

Đầu tiên tạo class cơ sở có áp dụng tính đa hình trong class gồm hàm ảo và hàm thuần ảo. Sau đó tạo thêm các class dẫn xuất kế thừa từ class cơ sở nhằm xử lý và lưu trữ. Trong các class sẽ có những cấu trúc như vector, stack,...nhằm lưu trữ hình ảnh, âm thanh, kiểu chữ. Trong các class dẫn xuất sẽ bao gồm các chức năng như xử lý chuyển động, xử lý đồ họa, thiết lập khung hình...

Bước tiếp theo ta sẽ sử dụng con trỏ thông minh (smart pointer) cấp quyền sở hữu tới các class để truy cập dữ liệu trong nó và dùng con trỏ thông minh để chuyển quyền sở hữu của nó từ class này sang class khác. Con trỏ thông minh ở đây nhằm mục đích để xử lý tình huống cho trò chơi thông qua các class không kế thừa từ class cơ sở.

Các class không kế thừa từ lớp cơ sở là các class dùng để xử lý trạng thái trò chơi. Dùng con trỏ thông minh để truy cập tới các class xử lý trạng thái. Sẽ có một class chứa vòng lặp để trò chơi sẽ không bao giờ kết thúc trừ khi người chơi tắt cửa sổ. Như đầu tiên là trạng thái Menu trò chơi đó dùng cấu trúc if, else để chuyển sang trạng thái bắt đầu chơi r sau đó là trạng thái khi thua sẽ chuyển về trạng thái xem thành tích trò chơi.

### 3. Hướng dẫn cài đặt và sử dụng

Bước 1: Cài đặt Visual Studio.

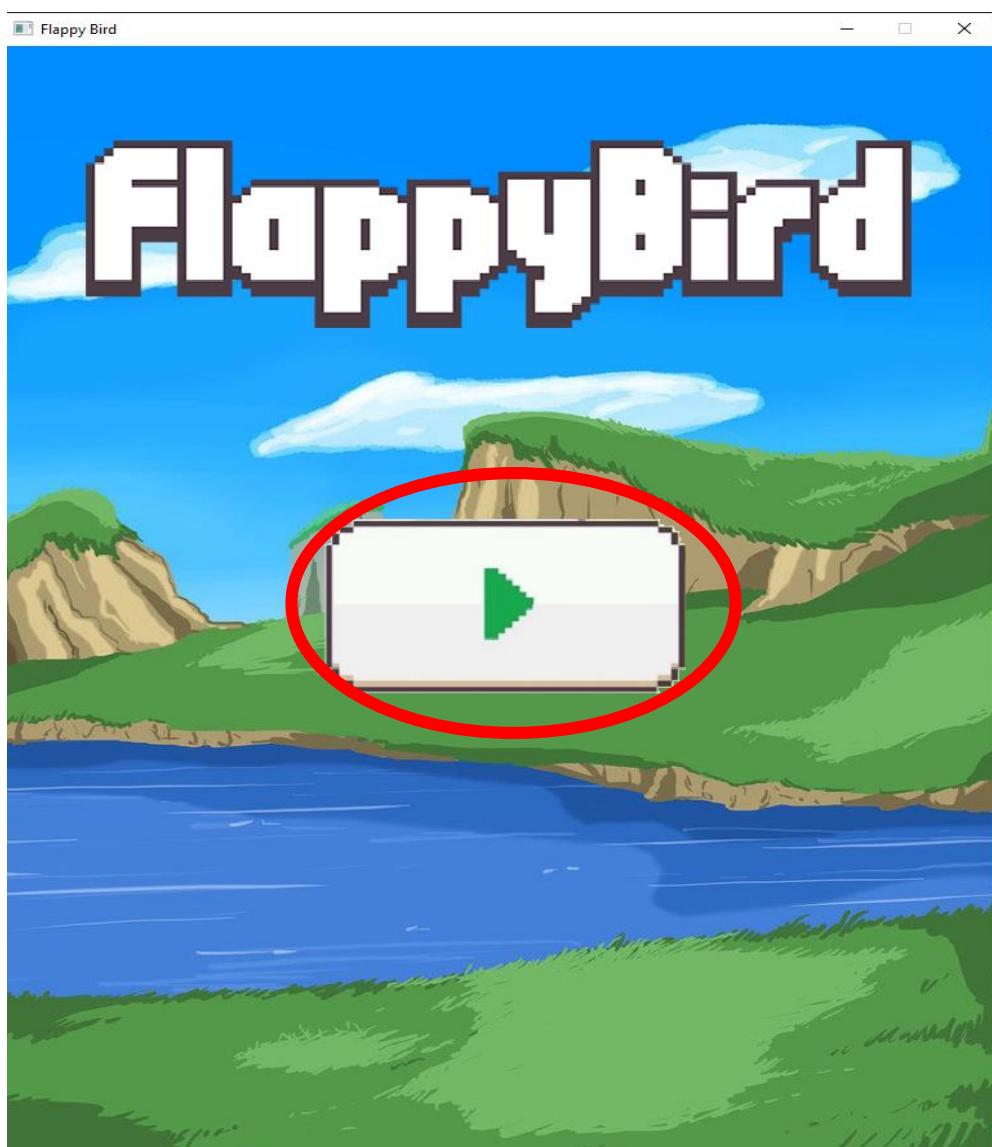
Bước 2: Tải source code từ link github:

<https://github.com/vodoanhhoanglong/BTL>

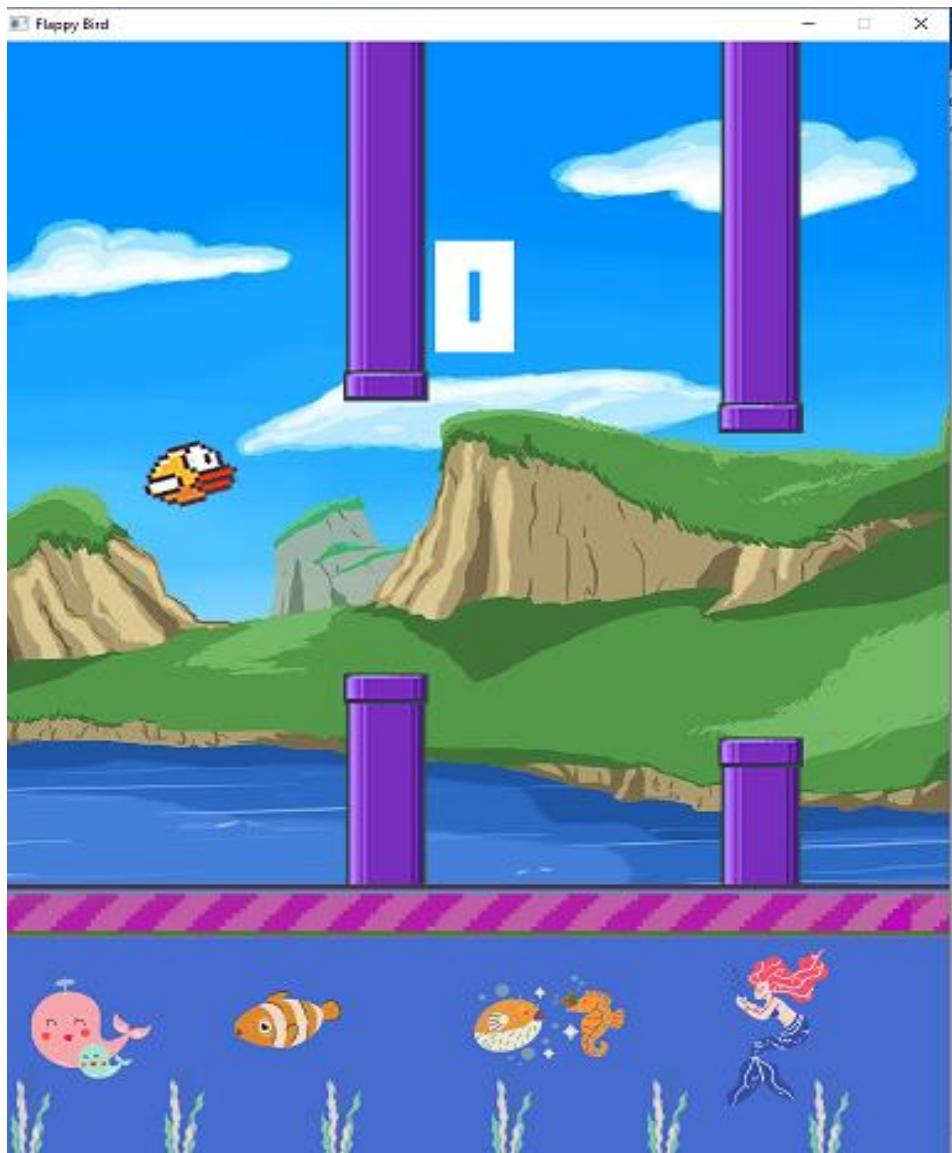
sau đó mở bằng Visual Studio.

Bước 3: Cài đặt thư viện SFML sau đó thêm vào trong Visual Studio theo hướng dẫn của: <https://www.youtube.com/watch?v=WrPn9vNFCsI&list=LL&index=3>

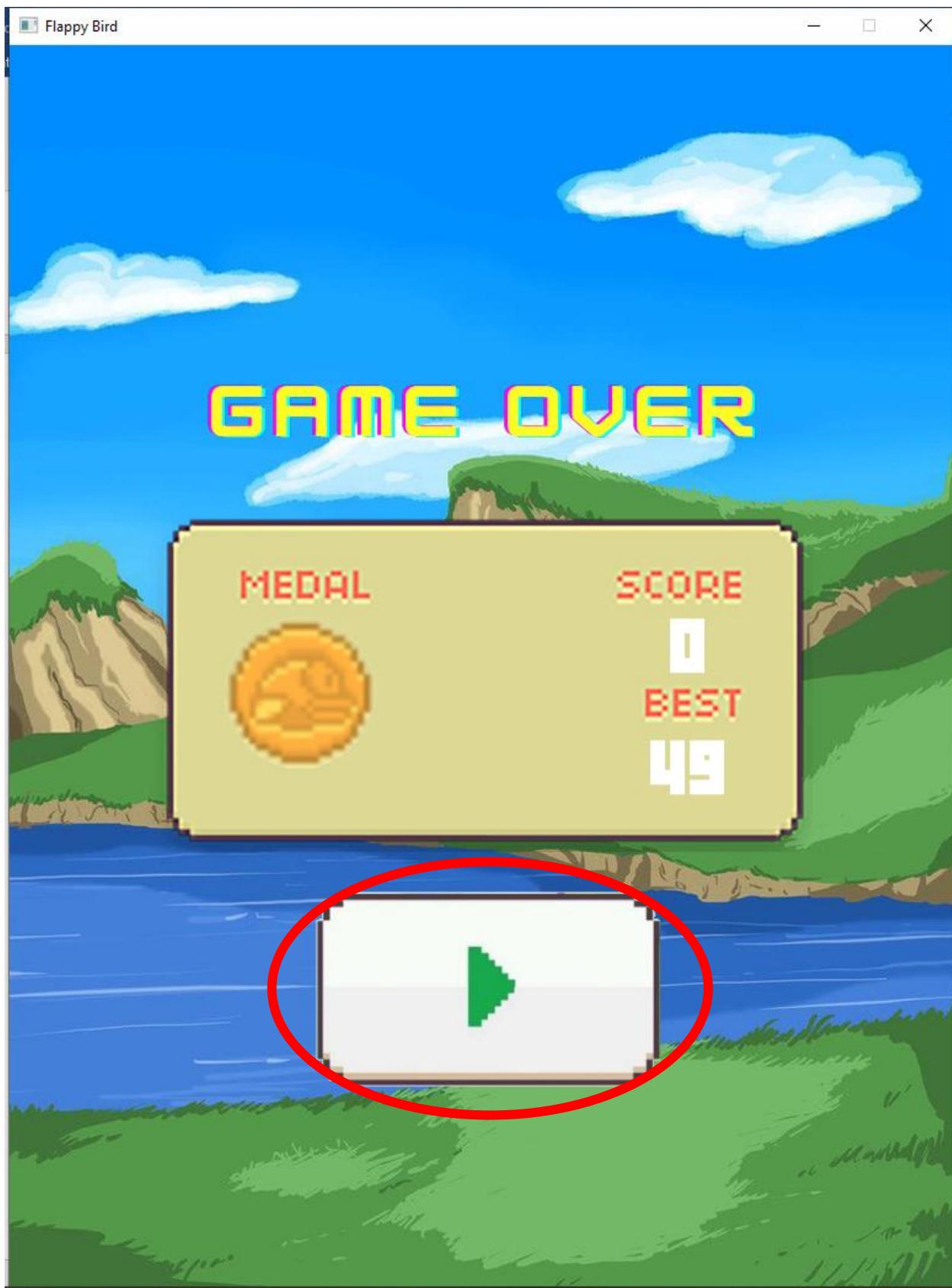
Bước 4: Nhấn vào nút được khoanh đỏ dưới đây để bắt đầu trò chơi.



Bước 5: Nhấn chuột trái vào cửa sổ để chim bay lên xuống, và nhiệm vụ của bạn là canh sao cho chim bay lọt qua khe hở giữa 2 ống để đạt điểm.



Bước 6: Khi bạn thua màn hình sẽ hiện điểm bạn đã đạt được trong lượt chơi đó và sẽ hiện điểm kỷ lục của bạn. Nếu bạn muốn chơi lại thì nhấn nút được khoanh đỏ dưới đây.



## **CHƯƠNG 3: KẾT LUẬN**

### **1. Kết quả đạt được**

Sau khoảng thời gian nghiên cứu và tự học hỏi nhóm chúng em đã hoàn thành xong trò chơi Flappy Bird. Từ đồ họa chỉ là những thanh màu, chim không có chuyển động, và nền đen...Chúng em đã phát triển và áp dụng thư viện đồ họa để làm cho trò chơi đẹp hơn bằng những hình ảnh sinh động hơn, thực tế hơn.

### **2. Nhược điểm**

- Còn hơi chậm hiểu trong quá trình làm đề tài.
- Một vài sai sót trong quá trình làm word.
- Chưa được tốt trong việc tự tìm hiểu, nghiên cứu

### **3. Hướng phát triển**

Nhóm chúng em sẽ cố gắng phát triển game với nhiều chức năng hơn như: tăng độ khó theo số điểm của người chơi đạt được, có hệ thống thi đấu online với người chơi khác để phân cấp thứ hạng, có thêm vài kỹ năng riêng cho từng loại chim (chim đại bàng, chim cánh cụt...) như lướt xuyên qua ống, bắt tử trong vài giây...theo điểm tích lũy hiện tại để có thể kích hoạt kỹ năng.

### **4. Công việc của mỗi thành viên và mức độ hoàn thành**

Võ Đoàn Hoàng Long: Code (100%).

Trần Huỳnh Lưu: Soạn Word (100%).

Lê Gia Minh: Dịch chương 6 (100%).

Hồ Ngọc Thống: Dịch chương 4 (100%).

Nguyễn Phi Thanh: Dịch chương 3 (100%).

### Tài liệu tham khảo

1. G. M. Adelson-Velskii and E. M. Landis, “An Algorithm for the Organization of Information,” Soviet. Mat. Doklady, 3 (1962), 1259–1263.
2. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass., 1974.
3. B. Allen and J. I. Munro, “Self Organizing Search Trees,” Journal of the ACM, 25 (1978), 526–535.
4. R. A. Baeza-Yates, “Expected Behaviour of B+-trees under Random Insertions,” Acta Informatica, 26 (1989), 439–471.
5. R. A. Baeza-Yates, “A Trivial Algorithm Whose Analysis Isn’t: A Continuation,” BIT, 29 (1989), 88–113.
6. R. Bayer and E. M. McCreight, “Organization and Maintenance of Large Ordered Indices,” Acta Informatica, 1 (1972), 173–189.
7. J. L. Bentley, “Multidimensional Binary Search Trees Used for Associative Searching,” Communications of the ACM, 18 (1975), 509–517.
8. J. R. Bitner, “Heuristics that Dynamically Organize Data Structures,” SIAM Journal on Computing, 8 (1979), 82–110.
9. D. Comer, “The Ubiquitous B-tree,” Computing Surveys, 11 (1979), 121–137.
10. J. Culberson and J. I. Munro, “Explaining the Behavior of Binary Search Trees under Prolonged Updates: A Model and Simulations,” Computer Journal, 32 (1989), 68–75.
11. J. Culberson and J. I. Munro, “Analysis of the Standard Deletion Algorithms in Exact Fit Domain Binary Search Trees,” Algorithmica, 5 (1990), 295–311.
12. K. Culik, T. Ottman, and D. Wood, “Dense Multiway Trees,” ACM Transactions on Database Systems, 6 (1981), 486–512.
13. B. Eisenbath, N. Ziviana, G. H. Gonnet, K. Melhorn, and D. Wood, “The Theory of Fringe Analysis and Its Application to 2–3 Trees and B-trees,” Information and Control, 55 (1982), 125–174.

14. J. L. Eppinger, “An Empirical Study of Insertion and Deletion in Binary Search Trees,” *Communications of the ACM*, 26 (1983), 663–669.
15. P. Flajolet and A. Odlyzko, “The Average Height of Binary Trees and Other Simple Trees,” *Journal of Computer and System Sciences*, 25 (1982), 171–213.
16. G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures*, 2d ed, Addison-Wesley, Reading, Mass., 1991.
17. E. Gudes and S. Tsur, “Experiments with B-tree Reorganization,” *Proceedings of ACM SIGMOD Symposium on Management of Data* (1980), 200–206.
18. L. J. Guibas and R. Sedgewick, “A Dichromatic Framework for Balanced Trees,” *Proceedings of the Nineteenth Annual IEEE Symposium on Foundations of Computer Science* (1978), 8–21.
19. T. H. Hibbard, “Some Combinatorial Properties of Certain Trees with Applications to Searching and Sorting,” *Journal of the ACM*, 9 (1962), 13–28.
20. A. T. Jonassen and D. E. Knuth, “A Trivial Algorithm Whose Analysis Isn’t,” *Journal of Computer and System Sciences*, 16 (1978), 301–322.
21. P. L. Karlton, S. H. Fuller, R. E. Scroggs, and E. B. Kaehler, “Performance of Height Balanced Trees,” *Communications of the ACM*, 19 (1976), 23–28.
22. D. E. Knuth, *The Art of Computer Programming: Vol. 1: Fundamental Algorithms*, 3d ed, Addison-Wesley, Reading, Mass., 1997.
23. D. E. Knuth, *The Art of Computer Programming: Vol. 3: Sorting and Searching*, 2d ed, AddisonWesley, Reading, Mass, 1998.
24. K. Melhorn, “A Partial Analysis of Height-Balanced Trees under Random Insertions and Deletions,” *SIAM Journal of Computing*, 11 (1982), 748–760.
25. K. Melhorn, *Data Structures and Algorithms 1: Sorting and Searching*, Springer-Verlag, Berlin, 1984.
26. J. Nievergelt and E. M. Reingold, “Binary Search Trees of Bounded Balance,” *SIAM Journal on Computing*, 2 (1973), 33–43.
27. A. J. Perlis and C. Thornton, “Symbol Manipulation in Threaded Lists,” *Communications of the ACM*, 3 (1960), 195–204.

28. S. Sen and R. E. Tarjan, “Deletion Without Rebalancing in Balanced Binary Trees,” Proceedings of the Twentieth Symposium on Discrete Algorithms (2010), 1490–1499.
29. D. D. Sleator and R. E. Tarjan, “Self-adjusting Binary Search Trees,” Journal of the ACM, 32 (1985), 652–686.
30. D. D. Sleator, R. E. Tarjan, and W. P. Thurston, “Rotation Distance, Triangulations, and Hyperbolic Geometry,” Journal of the AMS (1988), 647–682.
31. R. E. Tarjan, “Sequential Access in Splay Trees Takes Linear Time,” Combinatorica, 5 (1985), 367–378.
32. A. C. Yao, “On Random 2–3 Trees,” Acta Informatica, 9 (1978), 159–170.
33. M. D. Atkinson, J. R. Sack, N. Santoro, and T. Strothotte, “Min-Max Heaps and Generalized Priority Queues,” Communications of the ACM, 29 (1986), 996–1000.
34. J. D. Bright, “Range Restricted Mergeable Priority Queues,” Information Processing Letters, 47 (1993), 159–164.
35. G. S. Brodal, “Worst-Case Efficient Priority Queues,” Proceedings of the Seventh Annual ACMSIAM Symposium on Discrete Algorithms (1996), 52–58.
36. M. R. Brown, “Implementation and Analysis of Binomial Queue Algorithms,” SIAM Journal on Computing, 7 (1978), 298–319.
37. S. Carlsson, “The Deap—A Double-Ended Heap to Implement Double-Ended Priority Queues,” Information Processing Letters, 26 (1987), 33–36.
38. S. Carlsson and J. Chen, “The Complexity of Heaps,” Proceedings of the Third Symposium on Discrete Algorithms (1992), 393–402.
38. S. Carlsson, J. Chen, and T. Strothotte, “A Note on the Construction of the Data Structure ‘Deap’,” Information Processing Letters, 31 (1989), 315–317.
39. S. Carlsson, J. I. Munro, and P. V. Poblete, “An Implicit Binomial Queue with Constant Insertion Time,” Proceedings of First Scandinavian Workshop on Algorithm Theory (1988), 1–13.
40. S. C. Chang and M. W. Due, “Diamond Deque: A Simple Data Structure for Priority Deques,” Information Processing Letters, 46 (1993), 231–237.

41. D. Cheriton and R. E. Tarjan, “Finding Minimum Spanning Trees,” SIAM Journal on Computing, 5 (1976), 724–742.
42. C. A. Crane, “Linear Lists and Priority Queues as Balanced Binary Trees,” Technical Report STAN-CS-72-259, Computer Science Department, Stanford University, Stanford, Calif, 1972.
43. Y. Ding and M. A. Weiss, “The Relaxed Min-Max Heap: A Mergeable Double-Ended Priority Queue,” Acta Informatica, 30 (1993), 215–231.
44. J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan, “Relaxed Heaps: An Alternative to Fibonacci Heaps with Applications to Parallel Computation,” Communications of the ACM, 31 (1988), 1343–1354.
45. R. W. Floyd, “Algorithm 245: Treesort 3,” Communications of the ACM, 7 (1964), 701.
46. M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan, “The Pairing Heap: A New Form of Self-adjusting Heap,” Algorithmica, 1 (1986), 111–129.
47. M. L. Fredman and R. E. Tarjan, “Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms,” Journal of the ACM, 34 (1987), 596–615.
48. G. H. Gonnet and J. I. Munro, “Heaps on Heaps,” SIAM Journal on Computing, 15 (1986), 964–971.
49. A. Hasham and J. R. Sack, “Bounds for Min-max Heaps,” BIT, 27 (1987), 315–323.
50. D. B. Johnson, “Priority Queues with Update and Finding Minimum Spanning Trees,” Information Processing Letters, 4 (1975), 53–57.
51. C. M. Khoong and H. W. Leong, “Double-Ended Binomial Queues,” Proceedings of the Fourth Annual International Symposium on Algorithms and Computation (1993), 128–137.
52. D. E. Knuth, *The Art of Computer Programming*, Vol. 3: Sorting and Searching, 2d ed., AddisonWesley, Reading, Mass., 1998.
53. A. LaMarca and R. E. Ladner, “The Influence of Caches on the Performance of Sorting,” Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (1997), 370–379.

54. C. J. H. McDiarmid and B. A. Reed, “Building Heaps Fast,” *Journal of Algorithms*, 10 (1989), 352–365.
55. D. D. Sleator and R. E. Tarjan, “Self-adjusting Heaps,” *SIAM Journal on Computing*, 15 (1986), 52–69.
56. T. Strothotte, P. Eriksson, and S. Vallner, “A Note on Constructing Min-max Heaps,” *BIT*, 29 (1989), 251–256.
57. P. van Emde Boas, R. Kaas, and E. Zijlstra, “Design and Implementation of an Efficient Priority Queue,” *Mathematical Systems Theory*, 10 (1977), 99–127.
58. J. Vuillemin, “A Data Structure for Manipulating Priority Queues,” *Communications of the ACM*, 21 (1978), 309–314.
59. J. W. J. Williams, “Algorithm 232: Heapsort,” *Communications of the ACM*, 7 (1964), 347–348.
60. [sfml-dev.org](http://sfml-dev.org).
61. [cppdeveloper.com](http://cppdeveloper.com).
62. [programingz.com](http://programingz.com).