Chương 11 – HÀNG ƯU TIÊN

Cấu trúc dữ liệu hàng đợi mà chúng ta đã xem xét trong chương 3 là theo đúng nguyên tắc FIFO. Tuy nhiên trong thực tế, có những trường hợp cần có sự linh động hơn. Chẳng hạn trong số các công việc cần xử lý, có một số ít công việc vô cùng quan trọng, chúng cần được xử lý càng sớm càng tốt ngay khi có thể. Hoặc trong trường hợp có nhiều tập tin cùng đang chờ được in, một số tập tin chỉ có 1 trang trong khi một vài tập tin khác thì rất dài. Nếu các tập tin 1 trang được in trước thì không ảnh hưởng đến thời gian chờ đợi của các tập tin khác bao nhiêu. Ngược lại, nếu cứ theo thứ tự FIFO, một số bản in chỉ có 1 trang lại phải chờ đợi quá lâu.

Hàng có xét thứ tự ưu tiên, hay gọi tắt là hàng ưu tiên (*priority queue*), là một cách giải quyết các trường hợp trên một cách thỏa đáng. Tùy vào ứng dụng, tiêu chí để xét độ ưu tiên do chúng ta quyết định. Trong chương này chúng ta sẽ đặc tả và hiện thực CTDL cho hàng ưu tiên này.

11.1. Định nghĩa hàng ưu tiên

Hàng ưu tiên có các phương thức gần giống như một hàng đợi thông dụng, chỉ khác về mặt chiến lược:

Định nghĩa: Một **hàng ưu tiên** các phần tử kiểu T gồm các phần tử của T, kèm các tác vụ sau:

1. Tạo mới một đối tượng hàng rỗng.
2. **`priority_append`**: Thêm một phần tử mới vào hàng, giả sử hàng chưa đầy (tùy vào độ ưu tiên của phần tử dữ liệu mới nó sẽ được đứng ở một vị trí thích hợp).
3. **`priority_serve`**: Loại một phần tử ra khỏi hàng, giả sử hàng chưa rỗng (phần tử bị loại là phần tử đến lượt được xem xét theo quy ước độ ưu tiên đã định).
4. **`priority_retrieve`**: Xem phần tử tại đầu hàng (phần tử sắp được xem xét).

11.2. Các phương án hiện thực hàng ưu tiên

Giả sử độ ưu tiên là sự kết hợp bởi độ ưu tiên theo tiêu chí mà chúng ta đã chọn cùng với thứ tự xuất hiện của công việc. Khi đưa vào hàng, mỗi công việc sẽ có một thông số để chứa độ ưu tiên này. Chúng ta quy ước rằng độ ưu tiên càng nhỏ thứ tự ưu tiên càng cao.

Chúng ta có thể dùng DSLK đơn để hiện thực hàng ưu tiên. Việc thêm vào luôn thực hiện ở đầu danh sách, việc lấy ra sẽ phải duyệt lần lượt hết danh sách để chọn phần tử có độ ưu tiên cao nhất. Ngược lại, nếu khi thêm vào luôn giữ

danh sách đúng thứ tự tăng dần của độ ưu tiên, khi lấy ra chỉ cần lấy phần tử đầu danh sách. Cả hai cách đều tốn O(1) cho một tác vụ và O(N) cho tác vụ còn lại.

Phương án thứ hai có thể hiện thực hàng ưu tiên bởi một cây nhị phân tìm kiếm (BST). Phương án này cần O(logN) cho mỗi tác vụ thêm hoặc loại phần tử. Tác vụ `priority_serve` sẽ luôn lấy ra phần tử cực trái của cây nhị phân, điều này sẽ làm cho cây mất cân bằng và có thể khắc phục bằng cách sử dụng cây AVL thay cho cây BST bình thường.

Tuy nhiên các phương án sử dụng cây trên đây hơi bị thừa. Việc quản lý cây quá phức tạp so với mục đích của chúng ta.

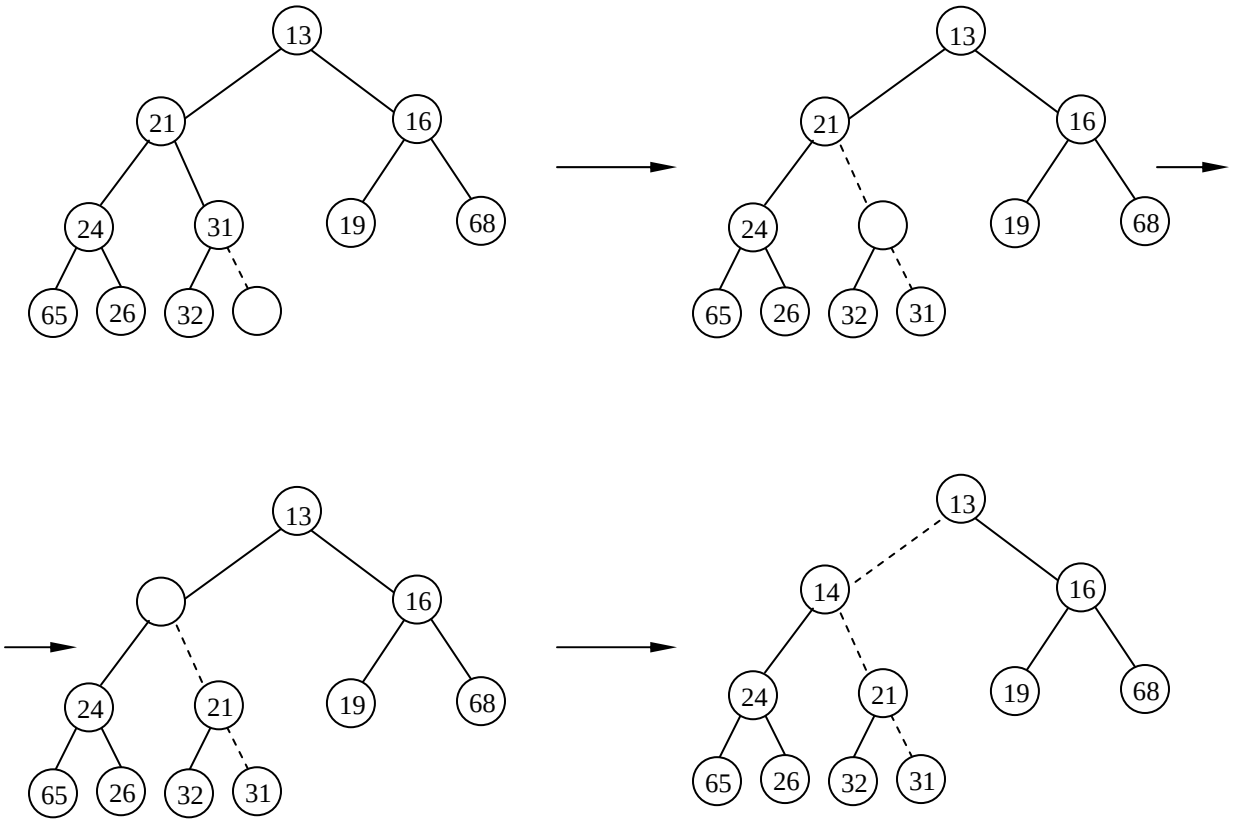
Cách hiện thực đơn giản và phổ biến cho hàng ưu tiên là `heap` nhị phân (*binary heap*), có khi còn được gọi tắt là `heap`. Và vì nó khá phổ biến nên nhiều lúc người ta chỉ gọi đơn giản là `heap`, chứ không còn gọi là hàng ưu tiên nữa. Định nghĩa `heap` nhị phân đã được trình bày trong chương 8. Trong chương này chúng ta sử dụng `heap` nhị phân là một `min-heap`.

Phần hiện thực một CTDL `heap` cụ thể được xem như bài tập. Phần tiếp sau đây trình bày các thao tác trên `heap` bằng mã giả, và để dễ dàng hình dung chúng ta cũng vẫn dùng hình ảnh của cây nhị phân để minh họa.

11.3. Hiện thực các tác vụ cơ bản trên heap nhị phân

11.3.1. Tác vụ thêm phần tử

Để thêm một phần tử mới vào heap, chúng ta tạo một chỗ trống ngay sau phần tử cuối của heap, điều này bảo đảm heap vẫn có cấu trúc cây nhị phân đầy đủ hoặc gần như đầy đủ. Nếu phần tử mới có thể đặt vào chỗ trống này mà không vi phạm điều kiện thứ hai của heap (bằng cách so sánh phần tử mới với nút cha của chỗ trống này) thì giải thuật kết thúc. Ngược lại, chúng ta lấy phần tử cha của chỗ trống này để lấp vào chỗ trống, lúc đó sẽ xuất hiện chỗ trống mới. Công việc lập lại tương tự cho đến khi tìm được vị trí thích hợp cho phần tử mới.
Hình 11.1 minh họa việc thêm phần tử 14 vào một `heap`.
Việc thêm phần tử mới tốn nhiều nhất là O(logN).

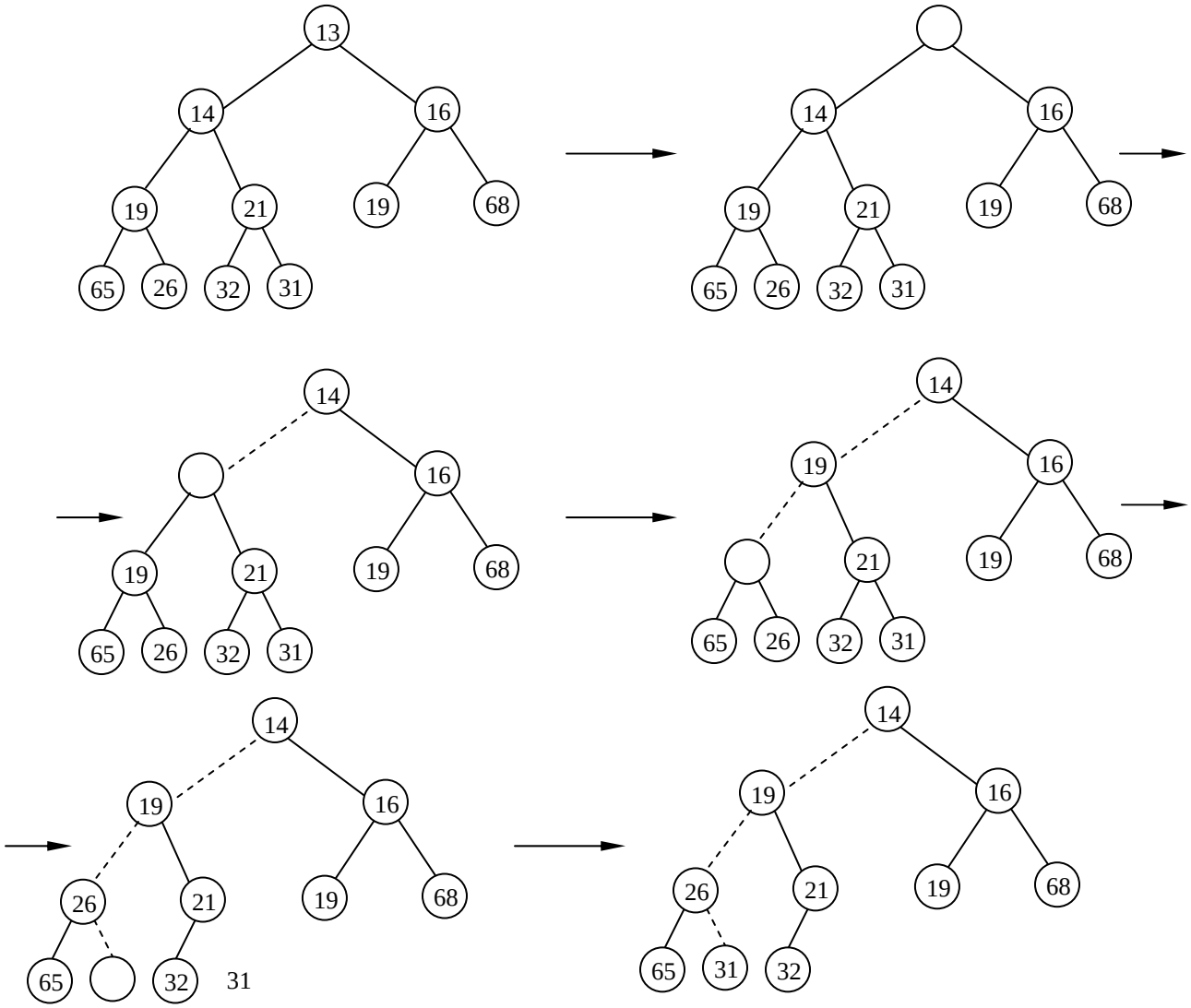


Hình 11.1. Thêm phần tử 14 vào heap

```
ErrorCode Priority_Queue::priority_append(Entry item)
/*
pre: đối tượng Priority_Queue có thuộc tính heap là mảng liên tục chứa các phần tử.
post: item được thêm vào hàng ưu tiên sao cho tính chất heap vẫn thoả.
*/
{
   1. if (full())
      1. return overflow;
   2. else
      1. current_position = size()
      2. loop ((tồn tại parent là cha của phần tử tại current_position) AND
                                                          (parent > item)
           // Lần lượt di chuyển các nút cha xuống nếu cha lớn hơn item để lấp chỗ trống
         1. heap[current_position] = parent
         2. Cho current_position là vị trí của parent
      3. endloop
      4. heap[current_position] = item
      5. // Cập nhật lại kích thước của heap.
      6. return success
   3. endif
}
```

11.3.2. Tác vụ loại phần tử

Việc loại phần tử cũng tương tự việc thêm vào. Phần tử lấy ra chính là phần tử tại gốc của cây vì đó là phần tử có độ ưu tiên cao nhất. Việc còn lại là giải quyết cho chỗ trống này. Trong hình 11.2 chúng ta sẽ thấy quá trình di chuyển của chỗ trống. Một trong hai phần tử con của nó sẽ di chuyển lấp đầy chỗ trống. Phần tử nhỏ nhất trong hai phần tử con được chọn để thỏa định nghĩa của heap. Cuối cùng, với chỗ trống không còn nút con thì được lấp đầy bởi phần tử cuối của heap vì chúng ta luôn biết rằng đây là cây nhị phân đầy đủ hoặc gần như đầy đủ, nó luôn chứa các phần tử có thể điền vào một mảng liên tục từ trái sang phải. Và thực sự chúng ta cũng hiện thực heap trong một mảng liên tục chứ không phải cấu trúc cây có con trỏ. Mọi thao tác với các chỉ số để định vị đến các phần tử cha, con, đều rất nhanh chóng. Chúng ta có thể chắc chắn rằng điều kiện thứ hai trong định nghĩa của heap cũng không bị vi phạm khi dời phần tử cuối bằng cách này. Chi phí trong việc loại phần tử là O(logN).



Hình 11.2. Loại một phần tử ra khỏi heap

```
ErrorCode Priority_Queue::priority_serve()
/*
pre: đối tượng Priority_Queue có thuộc tính heap là mảng liên tục chứa các phần tử.
post: phần tử nhỏ nhất trong hàng ưu tiên được lấy đi.
*/
{
   1. if (empty())
      1. return underflow;
   2. else
      1. current_position = 0
      2. loop (phần tử tại current_position có con)// Lần lượt di chuyển các nút
                                                    // con lên để lấp chỗ trống.
         1. child là phần tử nhỏ nhất trong hai con
         2. child được chép lên vị trí current_position
         3. Cho current_position là vị trí của child vừa được chép
      3. endloop
      4. heap[current_position] = heap[size()-1]
      5. // Cập nhật lại kích thước của heap.
      6. return success
   3. endif
}
```

11.4. Các tác vụ khác trên heap nhị phân

11.4.1. Tác vụ tìm phần tử lớn nhất

Tác vụ priority_retrieve truy xuất phần tử bé nhất trong min-heap có chi phí O(1). Đối với việc tìm ra phần tử lớn nhất trong min-heap khó khăn hơn. Tuy vậy, chúng ta cũng không phải dùng cách tìm tuyến tính trên toàn bộ heap, vì các phần tử trong heap luôn có một trật tự nhất định theo định nghĩa. Chúng ta thấy ngay rằng phần tử lớn nhất phải là một trong các nút lá, đó là một nửa bên phải của mảng liên tục chứa các phần tử của heap.

11.4.2. Tác vụ tăng giảm độ ưu tiên

Tại thời điểm khi mà các công việc được đưa vào hàng ưu tiên, mỗi công việc đều đã được xác định độ ưu tiên, và chỉ số này chính là khóa được xử lý bởi heap. Tuy nhiên, giả sử trong khi các công việc đang nằm trong hàng ưu tiên để chờ đến lượt được cung cấp dịch vụ, người điều hành muốn can thiệp vào thứ tự ưu tiên này vì một số lý do. Chẳng hạn có một công việc đang phải chờ quá lâu và có một yêu cầu đột xuất cần thúc đẩy nó được hoàn thành sớm hơn. Ngược lại người điều hành cũng có thể muốn điều chỉnh giảm độ ưu tiên một công việc nào khác. Những điều này rất thường xảy ra nếu chúng ta xét trong một tình huống rằng, trong chế độ phục vụ có phân chia thời gian, khi hết khoảng thời gian quy định, nếu công việc vẫn chưa thực hiện xong thì lại phải quay lại hàng đợi nằm chờ tiếp. Mỗi công việc thường phải đợi, được phục vụ, đợi, được phục vụ,... vài lần như thế mới kết thúc. Như vậy thì việc người điều hành được quyền can thiệp vào hàng đợi tùy vào những tình thế cụ thể là rất có lợi. Những công việc đôi khi do

tính thiếu hiệu quả, đã sử dụng tổng thời gian phục vụ quá lớn mà vẫn chưa kết thúc, thường bị giảm độ ưu tiên như một hình thức phạt.

Trước hết chúng ta cần bổ sung một vài CTDL và một vài hàm phụ trợ sao cho khi một công việc được đưa vào hàng ưu tiên chúng ta luôn nắm được vị trí của nó trong hàng ưu tiên, kể cả khi nó bị dịch chuyển do các thao tác của các tác vụ thêm, loại,...Tất nhiên những bổ sung này sẽ được đóng kín trong CTDL `Priority_Heap` của chúng ta. Sau đó, chúng ta có thể thiết kế hai tác vụ **`decrease_key`**`(position, delta)` và **`increase_key`**`(position, delta)` cho phép giảm hoặc tăng khóa của phần tử tại vị trí `position` trong `heap` một lượng `delta`. Khi giảm hoặc tăng như vậy, chúng ta chỉ cần xử lý dịch chuyển phần tử này lên hoặc xuống vị trí thích hợp trong `heap` so với giá trị mới của khóa, việc này rất dễ dàng và gần giống với những gì chúng ta đã làm trong hai tác vụ `priority_append` và `priority_serve`.

11.4.3. Tác vụ loại một phần tử không ở đầu hàng

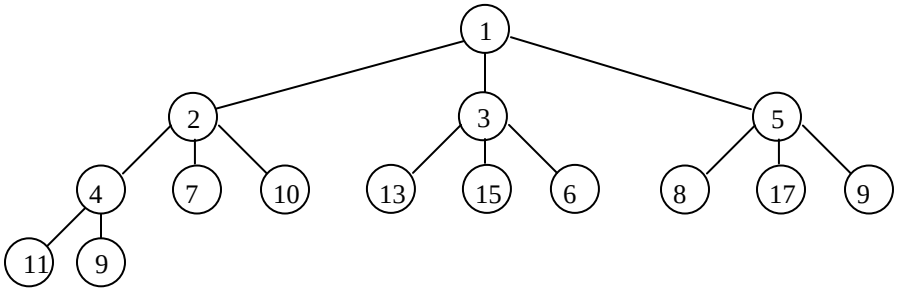
Chúng ta cũng có thể bổ sung thêm tác vụ loại hẳn một công việc đang đợi trong hàng (không phải là phần tử đang ở đầu hàng và có độ ưu tiên cao nhất) nhằm đáp ứng yêu cầu của một người sử dụng nào đó muốn ngưng không thực hiện công việc nữa. Tác vụ **`delete`**`(position)` đơn giản là gọi **`decrease_key`**`(position, delta)` với delta vô cùng lớn rồi gọi `priority_serve`.

11.5. Một số phương án khác của heap

Đặc tính chính yếu của heap là **trật tự giữa các phần tử cha – con**. Điều này đáp ứng mục đích của hàng ưu tiên là truy xuất nhanh chóng phần tử nhỏ nhất. `Heap` nhị phân khai thác tính năng của mảng liên tục tạo hiệu quả nhất định trong các thao tác trên hàng ưu tiên. Dưới đây là một vài phương án khác của `heap`, chúng khai thác các ưu điểm của các cách hiện thực khác nhau.

11.5.1. d-heaps

`Heap` nhị phân luôn phổ biến khi người ta cần dùng đến hàng ưu tiên. `d-heaps` hoàn toàn giống heap nhị phân ngoại trừ mỗi nút có d chứ không phải 2 con. d càng lớn càng lợi cho phép thêm vào, ngược lại, trong phép loại bỏ phần tử bé nhất, cần phải chi phí trong việc so sánh d phần tử con của một nút để lấy phần tử nhỏ nhất đẩy lên. Do đó `d-heap` thích hợp với các ứng dụng mà phép thêm vào được thực hiện thường xuyên. Ngoài ra còn phải tính đến chi phí trong việc định vị các nút cha, nút con trong mảng liên tục. Nếu d là lũy thừa của 2 thì các phép nhân, chia được thực hiện bởi phép dịch chuyển bit rất tiện lợi. Cuối cùng, tương tự `B-tree`, khi dữ liệu quá lớn không chứa đủ trong bộ nhớ thì `d-heap` cũng thích hợp với việc sử dụng thêm bộ nhớ ngoài.



Hình 11.3 . d-heap

Nhược điểm của các heap trên đây là việc tìm kiếm một phần tử bất kỳ hay việc trộn hai heap với nhau không thích hợp. Chúng ta sẽ xem xét một số cấu trúc phức tạp hơn nhưng rất thích hợp cho phép trộn.

11.5.2. Heap lệch trái (*Leftist heap*)

Việc sử dụng mảng liên tục như heap nhị phân thật không dễ để thực hiện phép trộn một cách hiệu quả, vì nó luôn đòi hỏi việc di chuyển các phần tử. Mọi CTDL thích hợp cho việc trộn đều dùng đến con trỏ. Nhược điểm của con trỏ là thời gian xác đinh vị trí các phần tử lâu hơn so với trong mảng liên tục. Heap lệch trái sẽ sử dụng cấu trúc liên kết với các con trỏ trái và phải tại mỗi nút để chứa địa chỉ của hai nút con, mục đích để khai thác điểm mạnh của phép trộn.

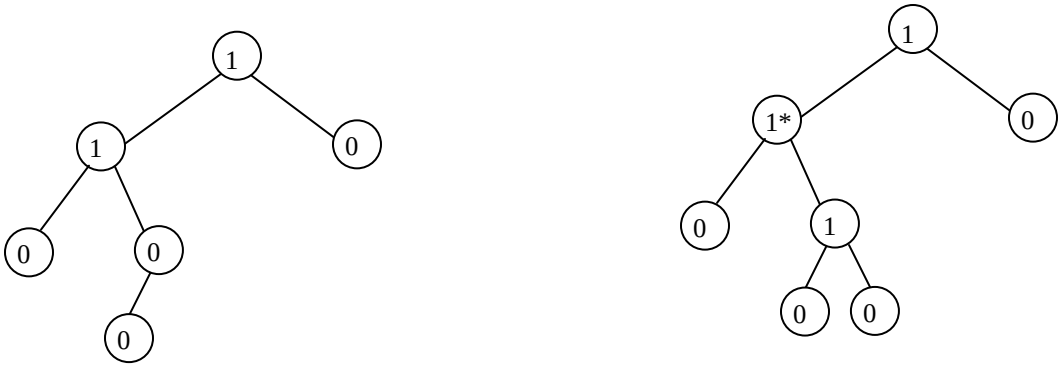
Heap lệch trái cũng giống với heap nhị phân ở cấu trúc nhị phân và trật tự giữa các phần tử cha – con. Chúng ta luôn nhớ rằng, trật tự giữa các phần tử cha – con là tính chất cơ bản nhất của mọi heap. Điểm khác ở đây là heap lệch trái không có sự cân bằng.

Chúng ta định nghĩa chiều dài đường đi đến NULL của một phần tử X (*null path length – Npl(X)*) là chiều dài của đường đi ngắn nhất từ X đến một nút lá. Npl của nút lá và nút bậc một bằng 0, Npl(NULL) = -1. Npl của bất kỳ nút nào bằng Npl của nút con có Npl nhỏ nhất cộng thêm 1.

Heap lệch trái có tính chất sau đây: tại mọi nút, Npl của nút con trái luôn lớn hơn hoặc bằng Npl của nút con phải. Tính chất này làm cho heap lệch trái mất cân bằng. Chúng ta gọi **đường đi trái** (**hoặc phải**) tại mỗi nút là đường đi đến nút dưới cực trái (cực phải) tương ứng của nút đó. Mọi nút trong heap lệch trái luôn có khuynh hướng có đường đi trái dài hơn đường đi phải, do đó đường đi phải tại nút gốc luôn là đường ngắn nhất trong các đường đi từ gốc đến các nút lá.

Có thể chứng minh bằng suy diễn rằng một cây heap lệch trái với r nút trên đường đi phải sẽ có ít nhất 2^r-1 nút. Từ đó cây heap lệch trái N nút sẽ có nhiều nhất $\lfloor \log(N+1) \rfloor$ nút trên đường đi phải. Ý tưởng chính của heap lệch trái là

mọi tác vụ đều được thực hiện trên đường đi phải để luôn được bảo đảm với chi phí `log(N)`.



(a) (b)

Hình 11.4. Npl tại mỗi nút.
(a)- Thoả điều kiện heap lệch trái.
(b)- Nút có dấu * vi phạm điều kiện heap lệch trái..

Các tác vụ trên heap lệch trái

Tác vụ cơ bản nhất của heap lệch trái là tác vụ trộn. Chúng ta sẽ thấy phép thêm vào và phép loại bỏ được thực hiện dễ dàng nhờ gọi phép trộn này.

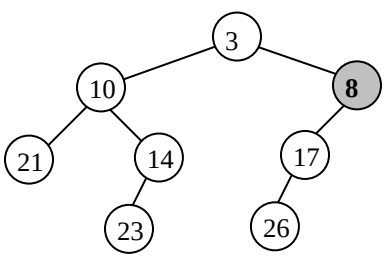
Ngoài hai con trỏ trái và phải, mỗi nút trong heap lệch trái còn có thêm thông tin là Npl của nó.

Để trộn hai heap lệch trái H_1 và H_2:

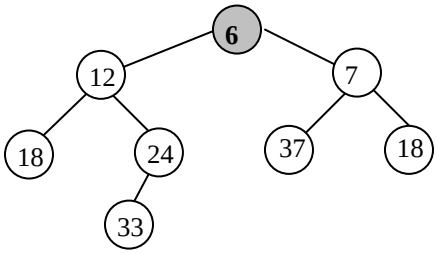
- Nếu một trong hai heap rỗng thì trả về heap còn lại.
- Ngược lại, so sánh dữ liệu tại hai nút gốc, thực hiện trộn heap có gốc lớn hơn với cây con phải của heap có gốc nhỏ hơn. Điều này bảo đảm gốc của heap kết quả luôn có trị nhỏ nhất trong tất cả các nút, vì heap kết quả có gốc chính là gốc của heap ban đầu có gốc nhỏ hơn.

Đây chính là quá trình đệ quy. Trước khi kết thúc một lần gọi đệ quy, chúng ta chỉ cần kiểm tra nút gốc này có thỏa điều kiện của heap lệch trái hay không, nếu cần chúng ta chỉ cần hoán vị hai con trái và phải của nó để có được Npl của nút con trái lớn hơn hoặc bằng Npl của nút con phải. Npl của nút gốc được cập nhật bằng Npl của nút con phải cộng thêm 1.

Hình 11.5 minh họa quá trình trộn hai heap lệch trái H_1 và H_2.

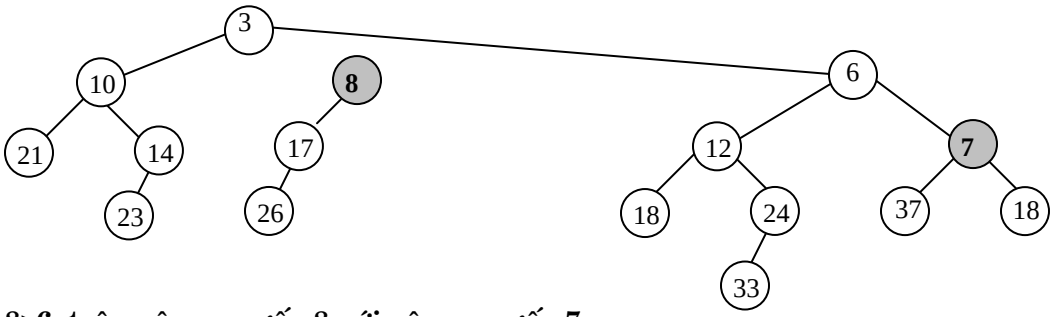


H_1

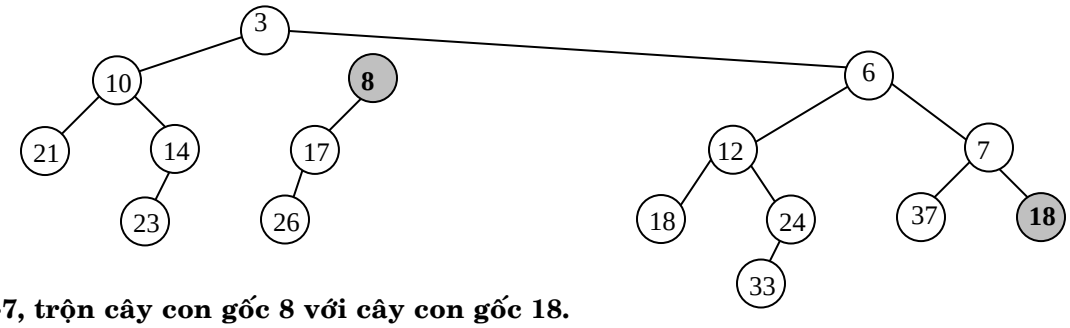


H_2

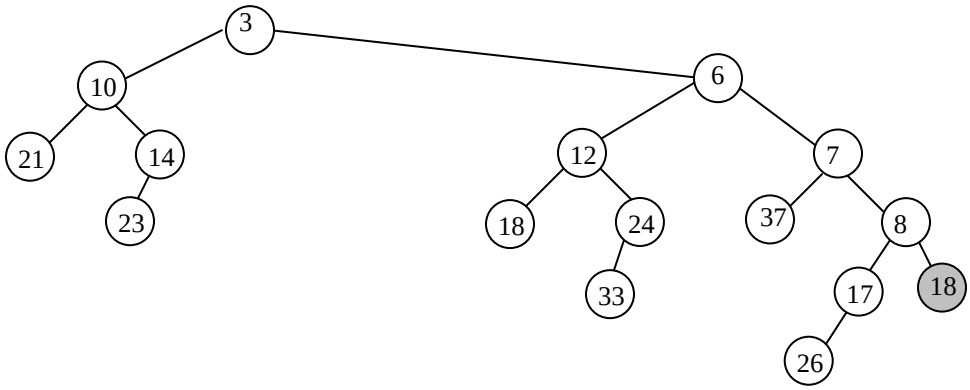
(a)- Do 6>3, trộn H_2 với cây con gốc 8.



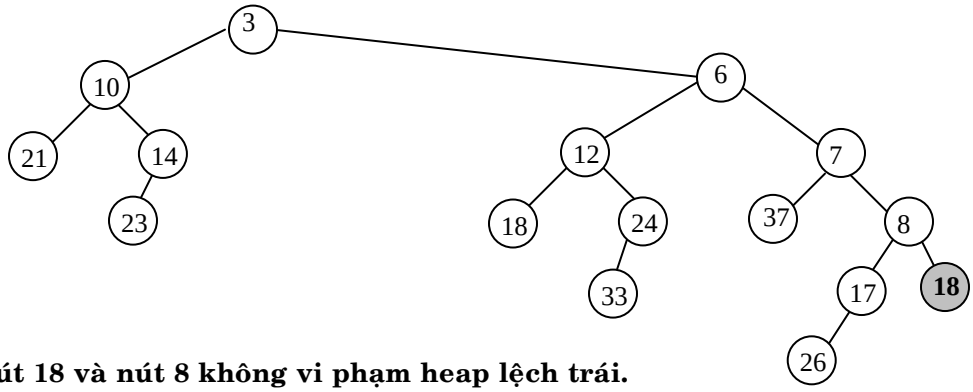
(b)- Do8>6, trộn cây con gốc 8 với cây con gốc 7.



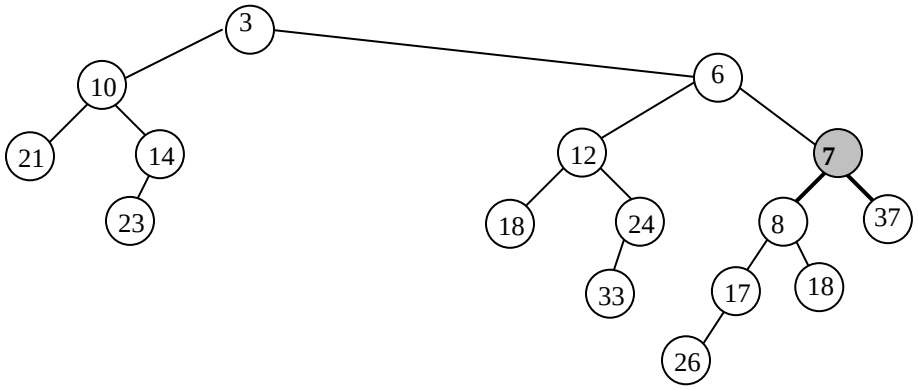
(c)-Do 8>7, trộn cây con gốc 8 với cây con gốc 18.



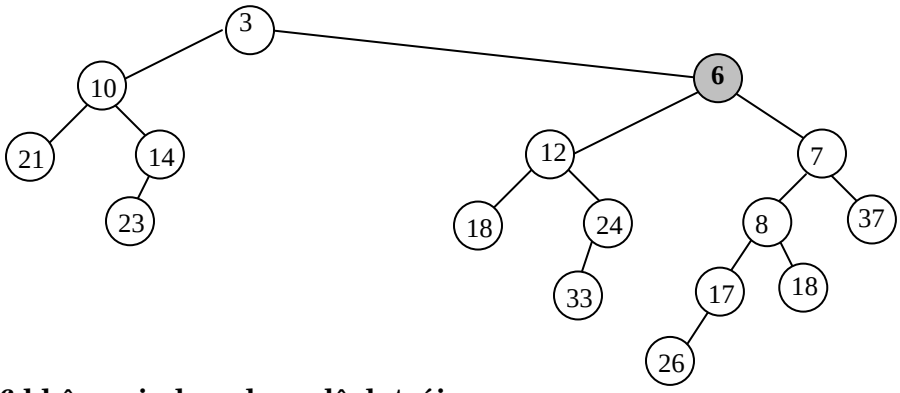
(d)- Do 18>8, trộn cây con gốc 18 với cây con phải của 8 (NULL)



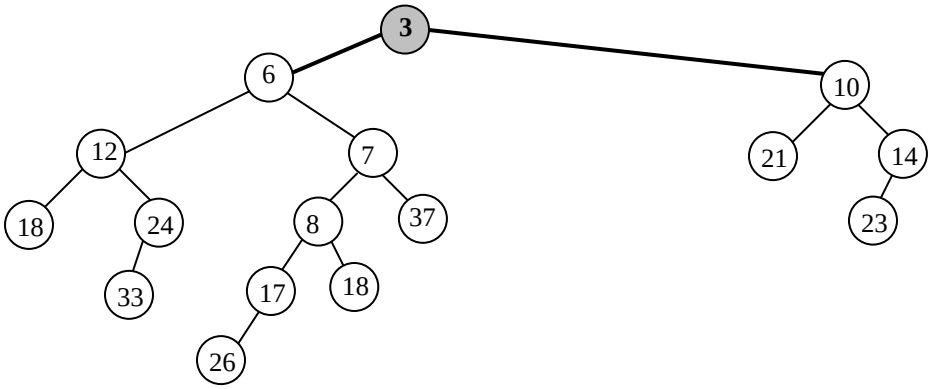
(e)- Tại nút 18 và nút 8 không vi phạm heap lệch trái.



(f)- Tại nút 7 vi phạm heap lệch trái, hoán vị hai cây con.



(g)- Tại nút 6 không vi phạm heap lệch trái.



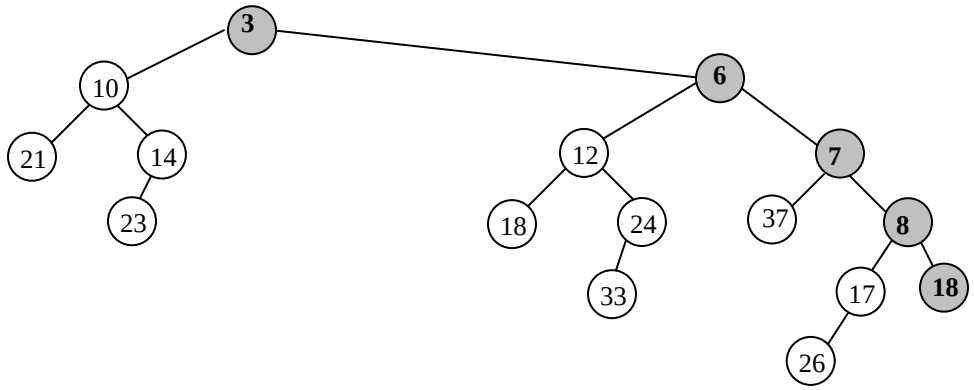
(h)- Tại nút 3 vi phạm heap lệch trái, hoán vị hai cây con.

Hình 11.5- Trộn hai heap lệch trái H_1 và H_2

```
//Phần mã giả cho khai báo và một số tác vụ cho LeftistHeap.

struct LeftistHeap_Node
   DataType          data
   LeftistHeap_Node* left
   LeftistHeap_Node* right
   int               Npl
end struct

class Leftist_Heap
   public:
      void merge(ref Leftist_Heap H1, ref Leftist_Heap H2)
   private:
      LeftistHeap_Node* recursive_merge(ref LeftistHeap_Node* p1,
                                        ref LeftistHeap_Node* p2)
      LeftistHeap_Node* aux_merge(ref LeftistHeap_Node* p1,
                                  ref LeftistHeap_Node* p2)
      LeftistHeap_Node* root
end class


void Leftist_Heap::merge(ref Leftist_Heap H1, ref Leftist_Heap H2)
/*
post: H1 là heap lệch trái là kết quả trộn hai heap H1 và H2, H2 rỗng.
*/
{
   1. recursive_merge(H1.root, H2.root);
}


LeftistHeap_Node* Leftist_Heap::recursive_merge(ref LeftistHeap_Node* p1,
                                                ref LeftistHeap_Node* p2)
/*
pre: p1 và p2 là đĩa chỉ nút gốc của hai heap lệch trái.
post: Trả về địa chỉ nút gốc của heap lệch trái là kết quả trộn hai heap ban đầu, p1 và p2 là
                                                                        NULL.
uses: Sử dụng hàm aux_merge.
*/
{
   LeftistHeap_Node* p;
   1. if (p1 == NULL)
      1. p = p2;
   2. if (p2 = NULL)
      1. p = p1;
   3. if (p1->data < p2->data)
      1. p = aux_merge(p1, p2);
   4. else
      1. p = aux_merge(p2, p1);
   5. p1 = NULL;
   6. p2 = NULL;
   7. return p;
}
```

```
LeftistHeap_Node* Leftist_Heap::aux_merge(ref LeftistHeap_Node* p1,
                                          ref LeftistHeap_Node* p2)
/*
pre: p1 và p2 là địa chỉ nút gốc của hai heap lệch trái.
post: Heap p2 được trộn với cây con phải của heap p1, p2 gán về NULL.
uses: Sử dụng hàm SwapChildren và recursive_merge.
*/
{
   1. if (p1->left == NULL) // Trường hợp này p1_>right đã là NULL do điều kiện
      1. p1->left = p2;     // của heap lệch trái.
   2. else
      1. p1->right = recursive_merge(p1->right, p2);
      2. if(p1->left->Npl < p1->right->Npl)
         1. SwapChildren(p1); // Tráo 2 cây con của p1.
      3. p1->Npl = p1->right->Npl + 1;
   3. return p1; // p2 đã được gán NULL trong recursive_merge.
}
```

Thời gian trộn hai heap lệch trái tỉ lệ với chiều dài của đường đi phải, và đó là O(logN). Giải thuật đệ quy trên có thể được khử đệ quy như sau. Bước thứ nhất thực hiện trộn đường đi phải của hai heap, đường đi phải của heap kết quả chính là thứ tự tăng dần của các nút trên đường đi phải của hai heap ban đầu (3, 6, 7, 8, 18). Kết quả cho thấy ở hình 11.6. Bước thứ hai đi lần ngược trên đường đi phải của cây kết quả để hoán vị các con trái và con phải khi cần thiết. Trường hợp chúng ta việc hoán vị cần thực hiện tại nút 7 và nút 3.



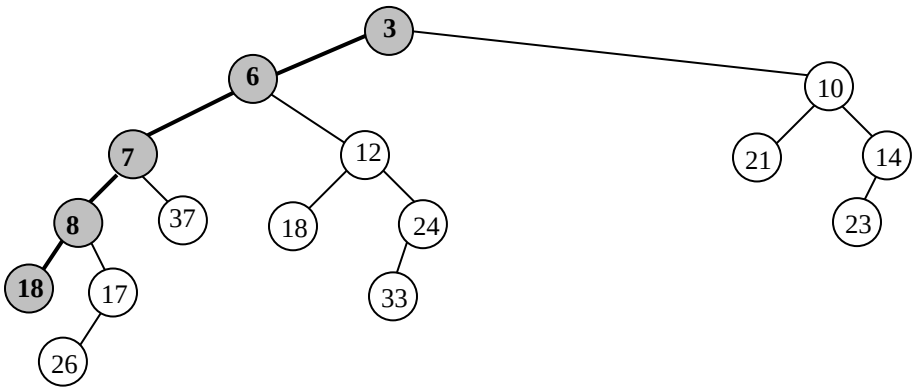
Hình 11.6. Kết quả sau bước thứ nhất của giải thuật trộn không đệ quy.

Phép thêm một phần tử mới chính là phép trộn heap lệch trái đã có với một heap lệch trái chỉ có duy nhất nút cần thêm vào. Phép loại phần tử nhỏ nhất của heap lệch trái là phép lấy đi phần tử tại gốc và trộn hai cây con của nó với nhau. Như vậy tất cả các tác vụ trên heap lệch trái đều có chi phí O(logN).

11.5.3. Skew heap

Skew heap giống như heap lệch trái nhưng không chứa thông tin về Npl và không có ràng buộc gì về chiều dài đường đi phải. Như vậy trong trường hợp xấu nhất các tác vụ chi phí đến O(N), khi cây nhị phân suy biến thành chuỗi mắc xích N nút về bên phải.

Tác vụ chính của skew heap cũng là phép trộn, trong đó việc hoán vị hai nhánh con luôn được làm. Tác vụ này có thể được hiện thực đệ quy hoặc không đệ quy. Kết quả việc hoán vị tại tất cả các nút sẽ là đường đi trái của heap cuối cùng sẽ chứa tất cả các nút trên hai đường đi phải của hai heap ban đầu theo đúng thứ tự tăng dần giữa chúng.



Hình 11.7. Kết quả trộn H1 và H2 theo cách của skew heap, hoán vị hai con trái và phải tại mọi nút

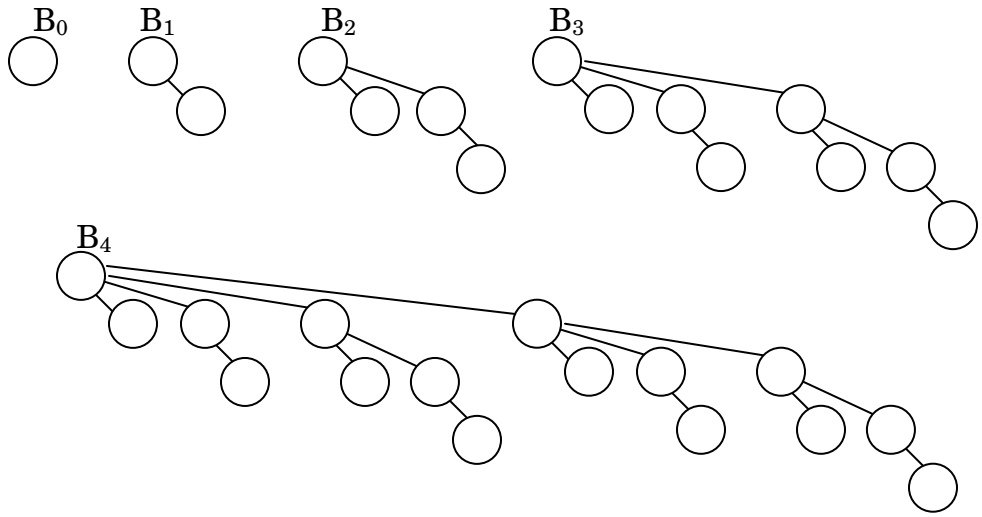
Như vậy tuy trường hợp xấu nhất của skew heap là O(log N), nhưng skew heap có ưu điểm là không phải chi phí để quản lý Npl tại mỗi nút và cũng không phải so sánh Npl để quyết định có hoán vị hai nút con tại mỗi nút hay không.

11.5.4. Hàng nhị thức (*Binomial Queue*)

Hàng nhị thức là một phương án hiện thực của hàng ưu tiên. Heap lệch trái và skew heap thực hiện O(log N) mỗi tác vụ đối với việc trộn, thêm và loại phần tử nhỏ nhất. Heap nhị phân thực hiện mỗi tác vụ thêm vào với thời gian trung bình là hằng số. Hàng nhị thức trong trường hợp xấu nhất thực hiện O(logN) cho mỗi tác vụ, nhưng việc thêm vào cũng có thời gian trung bình là hằng số.

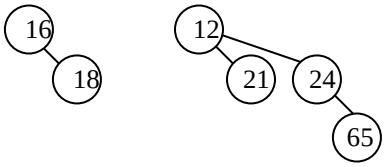
Khác với mọi hiện thực của hàng ưu tiên mà chúng ta đã xem xét, hàng nhị thức không phải là một cây có trật tự của heap, mà là một rừng các cây có trật tự của heap, trong đó không được phép có hai cây có cùng chiều cao. Theo quy ước, cây có chiều cao 0 là cây có 1 nút; cây có chiều cao k có được bằng cách nối một cây chiều cao k-1 vào nút gốc của một cây chiều cao k-1 khác. Hình 11.8 biểu diễn các cây có chiều cao lần lượt là 0, 1, 2, 3, 4. Từ hình vẽ chúng ta thấy, cây B_k bao gồm một nút gốc và các cây con B_0, B_1,..., B_{k-1}. Cây B_k có chính xác là 2^k nút, do đó

từ đây chúng ta sẽ gọi các cây này là **cây nhị thức**. Số nút ở mức d trong cây nhị thức là C^d_k. Nếu mọi cây nhị thức trong hàng nhị thức đều có trật tự của heap và không cho phép có nhiều hơn một cây nhị thức có cùng chiều cao thì hàng ưu tiên với kích thước bất kỳ đều có thể được biểu diễn bởi một hàng nhị thức như thế này. Chẳng hạn hàng ưu tiên có 13 nút sẽ gồm các cây nhị thức B_3, B_2, và B_0. Biểu diễn nhị phân của 13 chính là 1101, điều này hoàn toàn tương ứng với sự có mặt của B_3, B_2, và B_0, mà không có mặt của B_1.



Hình 11.8- Hình dạng các cây nhị thức với các chiều cao 0, 1, 2, 3, và 4 được quy định trong hàng nhị thức.

Hình 11.9 biểu diễn một hàng nhị thức có 6 nút.



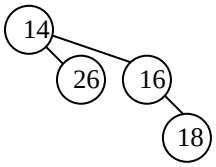
Hình 11.9- Hàng nhị thức có 6 nút.

Phần tử nhỏ nhất trong hàng nhị thức chính là một trong các nút gốc của các cây nhị thức có trong hàng. Do hàng nhị thức có nhiều nhất là log N cây nhị thức khác nhau, việc tìm kiếm chi phí `O(log N)`. Nếu chúng ta đánh dấu phần tử nhỏ nhất mỗi khi có sự thay đổi bởi một tác vụ nào thì chi phí này là `O(1)`.

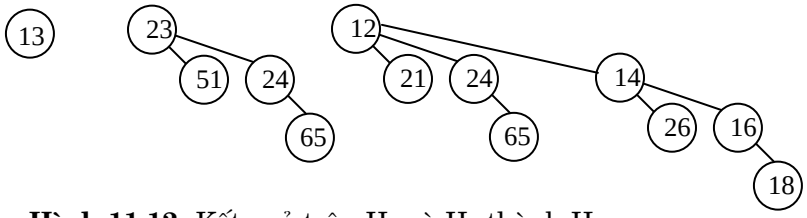
Phép trộn trong hàng nhị thức rất dễ dàng. Giả sử chúng ta cần trộn hai hàng nhị thức H_1 và H_2 trong hình 11.10. H_3 là hàng nhị thức kết quả. Trong H_1 và H_2 chỉ có một cây nhị thức B_0, vậy B_0 sẽ là một thành phần của H_3. Chúng ta kết hợp hai cây nhị thức B_1 trong H_1 và H_2 bằng cách cho cây nhị thức có gốc lớn hơn làm cây con của cây nhị thức còn lại. Với cây nhị thức B_2 vừa có được và hai cây nhị thức B_2 ban đầu trong H_1 và H_2, chúng ta để lại một cây trong H_3, và kết hợp

hai cây B_2 còn lại thành một cây B_3. Vì H_3 chưa có cây nhị thức B_3 nên B_3 cuối cùng này sẽ là thành phần của H_3.

Hình 11.10- Hai hàng nhị thức H_1 và H_2.



Hình 11.11 - Kết hợp hai cây nhị thức B_1 trong H_1 và H_2.

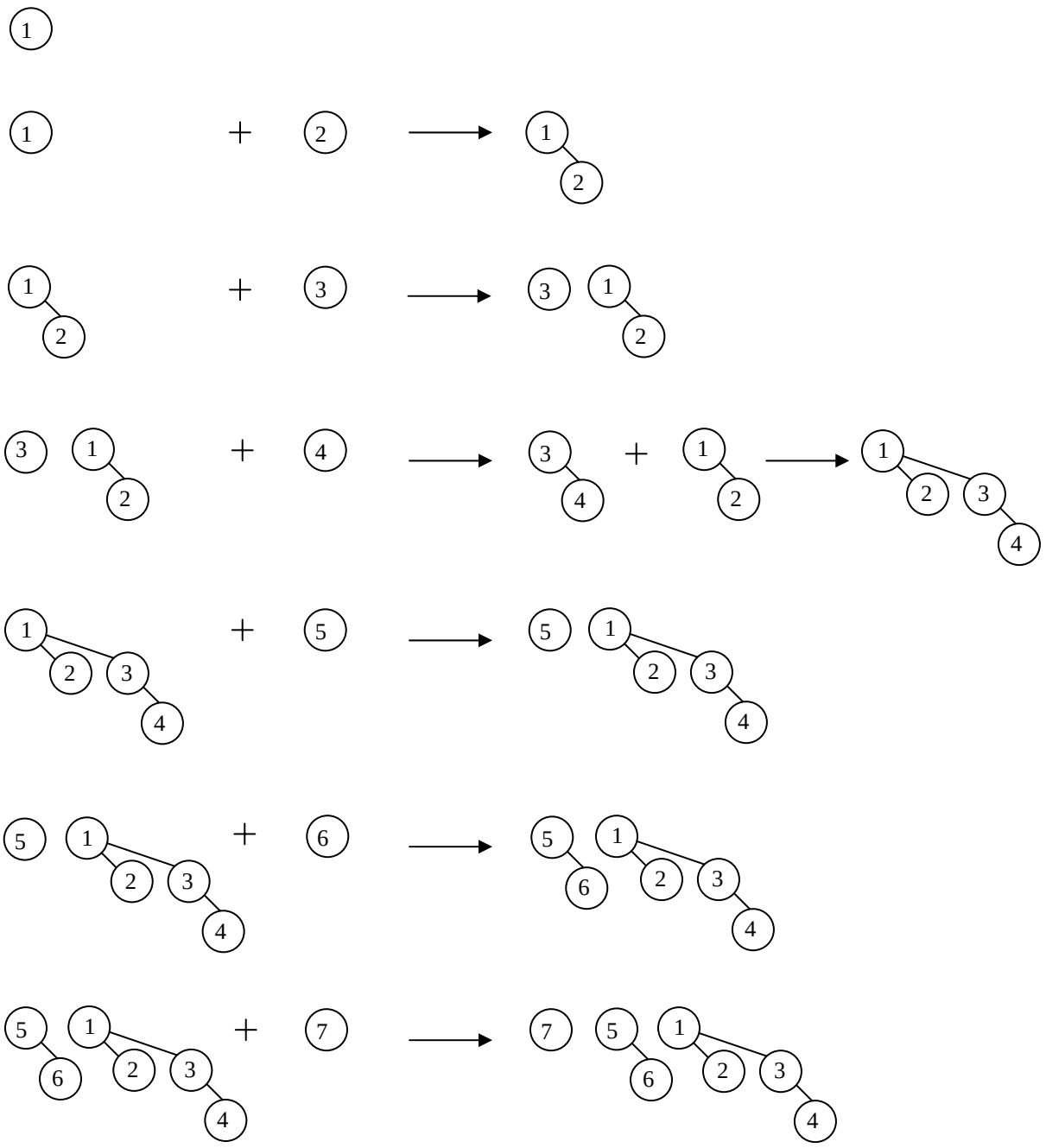


Hình 11.12- Kết quả trộn H_1 và H_2 thành H_3.

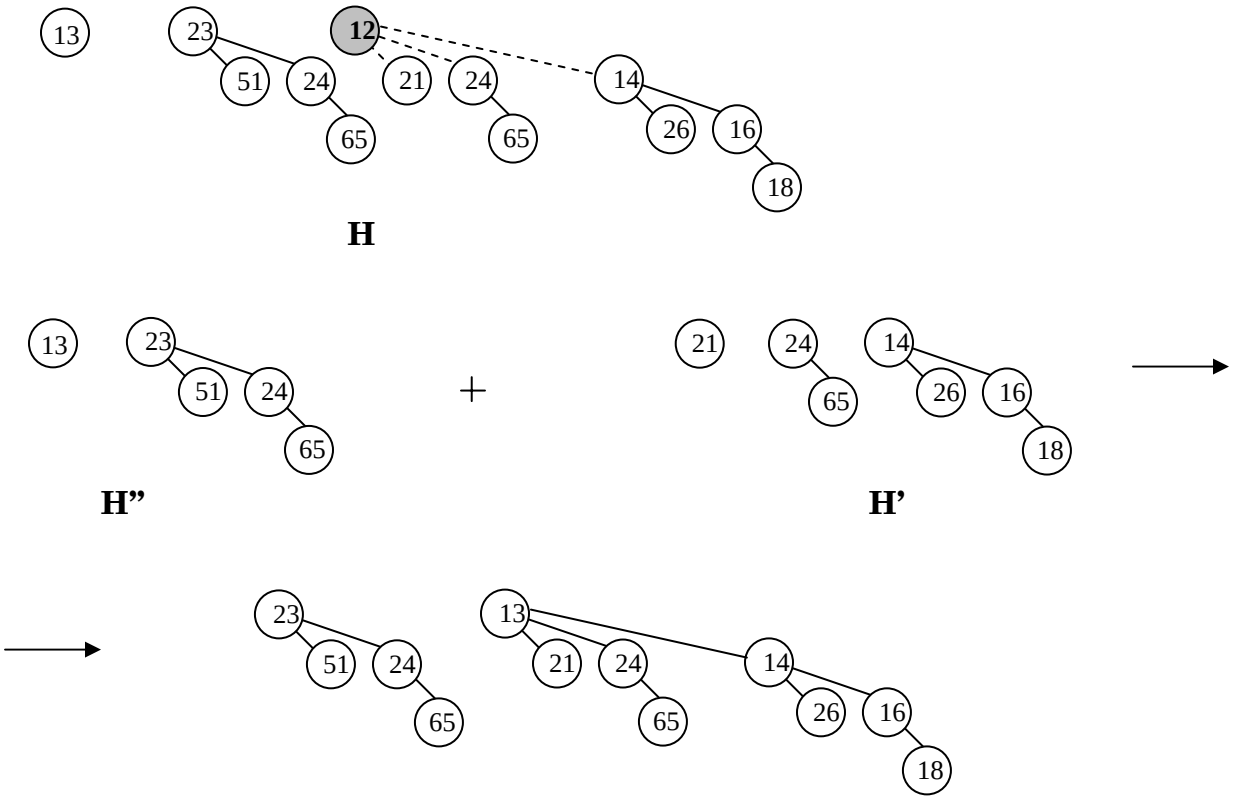
Việc kết hợp hai cây nhị thức tốn `O(1)`, với `O(log N)` cây nhị thức trong mỗi hàng nhị thức, việc trộn hai hàng nhị thức cũng tốn `O(log N)` trong trường hợp xấu nhất. Để tăng tính hiệu quả, chúng ta lưu các cây nhị thức trong mỗi hàng nhị thức theo thứ tự tăng dần của chiều cao các cây.

Việc thêm một phần tử mới vào hàng nhị thức tương tự như đối với heap lệch trái: trộn hàng nhị thức có duy nhất phần tử cần thêm với hàng nhị thức đã có.

Việc loại phần tử nhỏ nhất cũng đơn giản: tìm phần tử nhỏ nhất trong số các nút gốc của các cây nhị thức. Giả sử đó là cây B_k. Sau khi loại bỏ nút gốc của cây B_k, chúng ta còn lại các cây con của nó: B_0, B_1,... B_{k-1}. Gọi rừng gồm các cây con này là H', và H" là hàng nhị thức ban đầu không kể B_K. Việc cuối cùng cần làm là trộn H' và H". Chúng ta có thể tự kiểm tra rằng các phép thêm và loại này đều tốn `O(log N)` trong trường hợp xấu nhất.



Hình 11.13- Quá trình thêm các phần tử 1, 2,..., 7 vào hàng nhị thức.

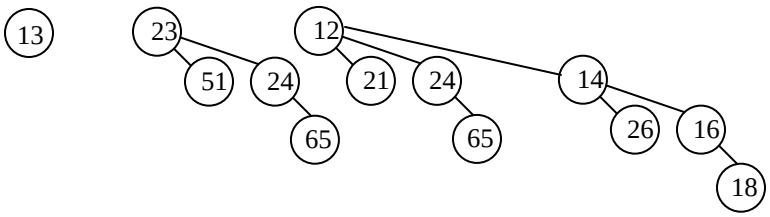


Hình 11.14- Quá trình loại phần tử nhỏ nhất trong hàng nhị thức H.

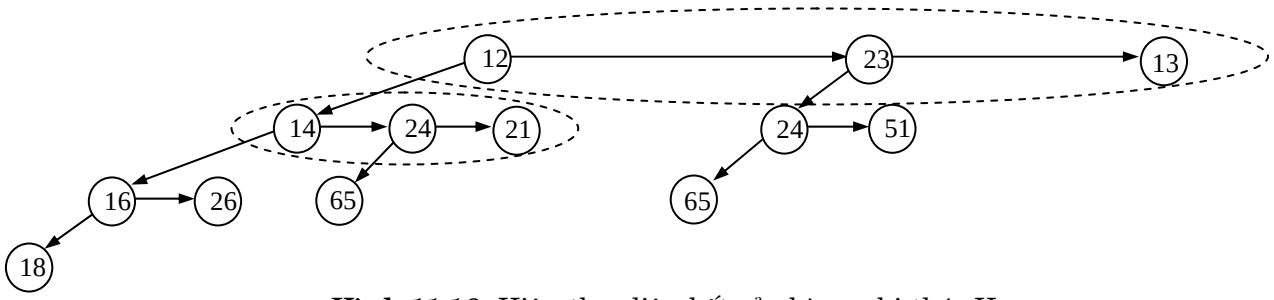
Hiện thực của hàng nhị thức

Việc tìm phần tử nhỏ nhất cần duyệt qua các gốc của các cây nhị thức trong hàng nhị thức (12, 23 và 13 trong hình 11.14). Chúng ta có thể dùng danh sách liên kết để chứa các nút gốc này. Danh sách sẽ có thứ tự theo chiều cao của các cây nhị thức để phục vụ cho phép trộn hai hàng nhị thức được dễ dàng.

Tương tự, các nút con của nút gốc của một cây nhị thức cũng được chứa trong một danh sách liên kết (14, 24 và 21 trong hình 11.14), để khi loại bỏ nút gốc (nút 12) thì phần còn lại cũng có cấu trúc tương tự như một hàng nhị thức mới, rất thuận lợi trong phép loại phần tử nhỏ nhất trong hàng nhị thức.



Hình 11.15- Hàng nhị thức H

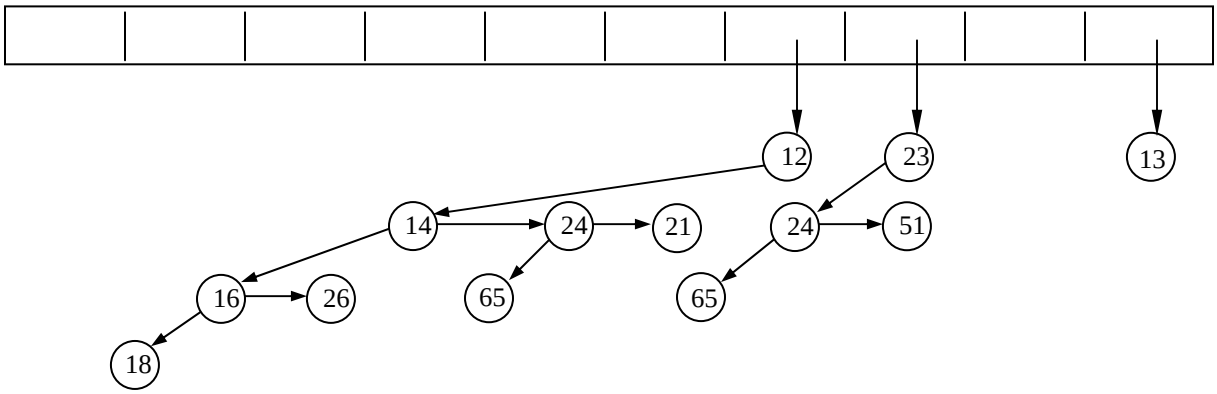


Hình 11.16- Hiện thực liên kết của hàng nhị thức H.

Trong hình vẽ trên chúng ta thấy hình *elipse* nét rời biểu diễn các danh sách liên kết các nút mà chúng ta thường phải duyệt qua. Hình *elipse* nét rời lớn chứa các gốc của các cây nhị thức trong hàng nhị thức, khi cần tìm phần tử nhỏ nhất trong hàng nhị thức chúng ta tìm trong danh sách này. Hình *elipse* nét rời nhỏ chứa các nút con của nút gốc trong một cây nhị thức.

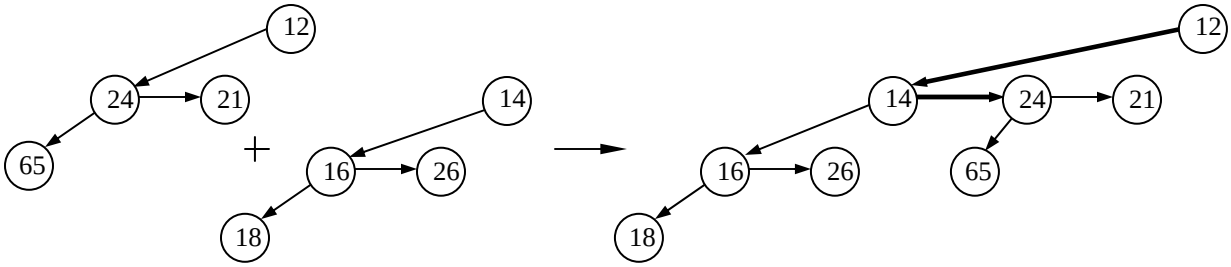
Trong quá trình hình thành hàng nhị thức trong một số thao tác dữ liệu, khi kết hợp hai cây nhị thức có cùng chiều cao (hình 11.18), chúng ta cần nối một trong hai cây thành cây con của cây còn lại, mà cây con mới này cũng chính là cây con có chiều cao lớn nhất so với các cây con đã có. Việc chèn cây con mới vào đầu của danh sách liên kết sẽ thuận tiện hơn, vì vậy chúng ta cho danh sách này có thứ tự giảm dần theo chiều cao của các cây con (hình 11.18).

Ngoài ra, mỗi nút trong cây nhị thức sẽ có một con trỏ cho phép truy xuất đến danh sách các con của nó. Tóm lại, hiện thực đơn giản và hiệu quả cho hàng nhị thức thật đơn giản như hình 11.18, đó là cấu trúc của một cây nhị phân, mỗi nút có con trỏ trái chỉ đến nút con đầu tiên của nó và con trỏ phải chỉ đến nút anh em của nó.



Hình 11.17- Gốc các cây nhị thức được chứa trong mảng liên tục.

Hình 11.17 là một phương án thay danh sách liên kết trên cùng bởi mảng liên tục. Chúng ta có thể dùng mảng liên tục cấp phát động để khắc phục nhược điểm do không biết trước chiều cao của cây nhị thức cao nhất trong hàng nhị thức. Việc dùng mảng liên tục cho phép tìm nhị phân phần tử nhỏ nhất, tuy nhiên sẽ là lãng phí lớn khi hàng nhị thức gồm quá ít cây nhị thức với nhiều chiều cao khác nhau.



Hình 11.18- Kết hợp hai cây nhị thức B_2 thành một cây nhị thức B_3.

Dưới đây là phần mã giả cho các khai báo cấu trúc của cây nhị thức và hàng nhị thức. Các tác vụ kết hợp hai cây nhị thức, trộn hai hàng nhị thức, và loại phần tử nhỏ nhất trong hàng nhị thức cũng được trình bày bằng mã giả.

```
//Phần mã giả cho khai báo và một số tác vụ cho hàng nhị thức.

struct Binomial_Node
   DataType          data
   Binomial_Node* leftChild
   Binomial_Node* nextSibling
end struct

struct Binomial_Tree
   Binomial_Tree CombineTrees(ref Binomial_Tree T1, ref Binomial_Tree T2)
   Binomial_Node* root
   int notEmpty() // Trả về 1 nếu cây không rỗng, ngược lại trả về 0.
end struct

class Binomial_Queue
   public:
      Binomial_Queue(Binomial_Node* p, int k)// k là số nút trong hàng nhị thức.
      int empty
      Binomial_Queue merge(ref Binomial_Queue H1, ref Binomial_Queue H2)
   protected:
      int count // Tổng số nút trong tất cả các cây nhị thức.
      Binomial_Tree Trees[MAX]
end class

Binomial_Tree Binomial_Tree::CombineTrees(ref Binomial_Tree T1,
                                          ref Binomial_Tree T2);
/*
pre: T1 và T2 khác rỗng.
post: T1 chứa kết quả kết hợp hai cây T1 và T2, T2 rỗng. Trả về cây T1.
*/
{
   1. if (T1.root->data > T2.root->data)
      1. Tráo T1 và T2 cho nhau
   2. T2.root->nextSibling = T1.root->leftChild // Chèn cây T2 vào đầu danh
                                                // sách các cây con của gốc của T1
   3. T1.root->leftChild = T2.root // Cập nhật nút con trái cho nút gốc của T1
   4. T2.root  = NULL
   5. return T1
}
```

```
Binomial_Queue Binomial_Queue::merge(ref Binomial_Queue H1,
                                     ref Binomial_Queue H2)
/*
post: H1 chứa kết quả trộn hai hàng nhị thức H1 và H2, H2 rỗng. Trả về hàng H1.
uses: hàm Binomial_Tree::notEmpty() trả về 1 nếu cây không rỗng, ngược lại trả về 0.
*/
{
   Binomial_Tree T1, T2, carry
   int i = 0
   1. H1.count = H1.count + H2.count
   2. loop (H2 chưa rỗng hoặc carry không rỗng)
      1. T1 = H1.Trees[i]
      2. T2 = H2.Trees[i]
      3. case (T1.notEmpty() + 2*T2.notEmpty() + 4*carry.notEmpty())
         1. case 0: // Không có cây nào
         2. case 1: // Chỉ có cây T1
            1. break
         3. case 2: // Chỉ có cây T2
            1. H1.Trees[i] = T2
            2. H2.Trees[i].root = NULL
            3. break
         4. case 4: // Chỉ có cây carry
            1. H1.Trees[i] = carry
            2. carry.root = NULL
            3. break
         5. case 3: // Có cây T1 và T2
            1. carry = combineTrees(T1, T2)
            2. H1.Trees[i].root = NULL
            3. H2.Trees[i].root = NULL
            4. break
         6. case 5: // Có cây T1 và carry
            1. carry = combineTrees(T1, carry)
            2. H1.Trees[i].root = NULL
            3. break
         7. case 6:// Có cây T2 và carry
            1. carry = combineTrees(T2, carry)
            2. H2.Trees[i].root = NULL
            3. break
         8. case 7:// Có cả 3 cây T1, T2, và carry
            1. H1.Trees[i] = carry
            2. carry = combineTrees(T1, T2)
            3. H2.Trees[i].root = NULL
            4. break
      4. endcase
      5. i = i + 1
   3. endloop
   4. return H1
}
```

```
Binomial_Queue::Binomial_Queue(Binomial_Node* p, int k)
/*
pre: p là địa chỉ nút đầu tiên của một cấu trúc liên kết các cây nhị thức B_k,B_k-1,...,B_0.
post: Hàng nhị thức mới được tạo thành từ các cây này.
*/
{
   1. count = (1 << (k+1)) –1
   2. loop (k >= 0)
      1. Trees[k] = p
      2. p = p->nextSibling
      3. Trees[k]->nextSibling = NULL// Cắt rời các cây con để đưa vào mảng liên
                                     // tục các cây nhị thức cho hàng nhị thức mới.
      4. k = k – 1
   3. endloop
}
```

Tóm lại, trong chương này chúng ta đã xem xét một số cách hiện thực của hàng ưu tiên. Heap nhị phân vừa đơn giản vừa hiệu quả vì không sử dụng con trỏ, nó chỉ hơi tốn nhiều vùng nhớ chưa sử dụng đến mà thôi. Chúng ta đã nghiên cứu thêm tác vụ trộn và bổ sung thêm ba phương án khác của hàng ưu tiên. Heap lệch trái là một ví dụ khá hay về giải thuật đệ quy. Skew heap giống heap lệch trái nhưng đơn giản hơn, với hy vọng rằng các ứng dụng có tính ngẫu nhiên thì các trường hợp xấu nhất sẽ không thường xuyên xảy ra. Cuối cùng hàng nhị thức cho thấy một ý tưởng đơn giản mà lại giới hạn được chi phí cho giải thuật khá tốt.