Chương 6: Heapsort

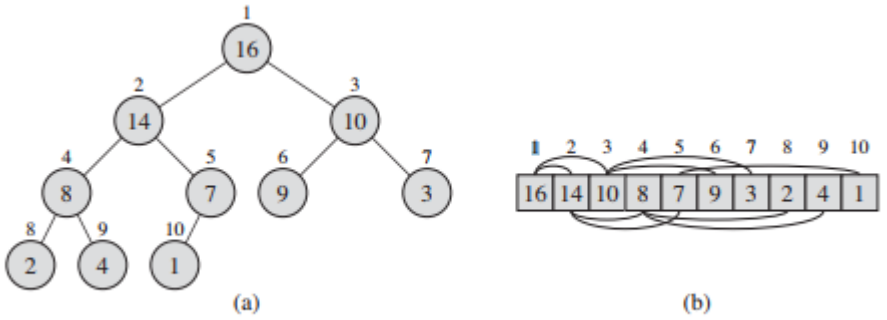
Ở chương này, ta trình bày về một loại thuật toán sắp sếp khác: heapsort (sắp sếp đống). Heapsort chạy với thời gian là $O(n \lg n)$ giống với megresort, khác với insertionsort. Và heapsort giống với insertionsort khác với megresort ở chỗ: chỉ có một hằng số từ thành phần của mảng được lưu bên ngoài mảng. Bên cạnh đó, heapsort bao gồm các thuộc tính tốt hơn của hai thuật toán sắp xếp mà chúng ta đã thảo luận.

Heapsort cũng giới thiệu một kỹ thuật thiết kế thuật toán khác: sử dụng cấu trúc dữ liệu, ở trường hợp này chúng ta gọi là "heap" (đống), để quản lý thông tin. Không những cấu trúc dữ liệu heap hữu dụng trong heapsort, mà còn tạo ra một hàng đợi hiệu quả hơn. Cấu trúc dữ liệu heap sẽ được gặp lại trong các chương tiếp theo.

Định nghĩa của heap ban đầu được đặt ra trong heapsort, nhưng nó gợi lên như là "hệ thống thu nhặt rác", như là các ngôn ngữ lập trình Java và Lisp. Cấu trúc dữ liệu heap của chúng ta không phải là hệ thống thu nhặt rác và mỗi lần chúng ta nhắc tới heap trong cuốn sách này, thì nó có nghĩa là một cấu trúc dữ liệu hơn là một phần của hệ thống nhặt rác.

6.1 Heaps

Cấu trúc dữ liệu ***heap (nhị phân)*** là một đối tượng của mảng mà chúng ta có thể xem như gần giống như là cây nhị phân (xem ở phần B.5.3), được trình bày ở hình 6.1. Mỗi node (nút) của cây tương ứng như là một thành phần của mảng. Một cây hoàn toàn được điền vào đầy đủ các tầng, ngoại trừ tầng thấp nhất có thể không được như thế, được lấp đầy từ trái lên một điểm. Một mảng A được thể hiện như là một heap với hai thành phần: $A.lenght$, là số các phần tử trong mảng và A.heap-size, biểu diễn cho có bao nhiêu phần tử trong heap mà được lưu giữ trong mảng A. Do đó, mặc dù $A[1..A.lenght]$ có thể chứa nhiều số, nhưng chỉ có các phần tử trong $A[1..A.heap\text{-}size]$ ($0 \leq A.heap\text{-}size \leq A.length$) là phần tử hợp pháp trong heap. Gốc của cây là phần tử $A[1]$ và i được biểu thị cho một node, chúng ta có thể dễ dàng tính được chỉ số của node cha, con trái và con phải:



Hình 6.1 Một heap được biểu diễn như là cây nhị phân **(a)** và mảng **(b)**. Số nằm trong vòng tròn của mỗi node trong cây là giá trị được lưu trong node. Số trên node biểu thị cho số thứ tự trong mảng. Trên và dưới mảng là dòng các số có quan hệ cha-con; số cha luôn ở bên trái của số con. Cây có độ cao là ba; node có thứ tự là 4 (với giá trị là 8) có độ cao là một.

CHA(i)

1 **return** [i/2]

TRÁI(i)

1 **return** 2i

PHẢi(i)

1 **return** 2i + 1

Hầu hết các máy tính, phương thức TRÁI có thể được tính 2i trong 1 dòng lệnh bằng cách đảo các biểu diễn nhị phân của i trái 1 bit. Tương tự, phương thức PHẢI có thể tính nhanh 2i + 1 bằng cách đảo biểu diễn nhị phân của i trái 1 bit sau đó thêm 1 vào như là biến bậc thấp. Phương thức CHA [i/2] có thể tính bằng cách đảo i phải 1 bit. Phương thức heapsort tốt thường vận hành các thủ tục như là "vi mô" hoặc "nội tuyến".

Có hai loại heap nhị phân: max-heap và min-heap. Ở cả hai loại, giá trị của các node thỏa mãn một ***thuộc tính của heap***, chi tiết cụ thể phụ thuộc vào loại của heap. Ở ***max-heap***, ***thuộc tính của max-heap*** là tất cả node i khác với node gốc.

A[CHA(i)] ≥ A[i].

vì thế, giá trị của một node là phần lớn giá trị của các node cha của nó. Do đó, phần tử lớn nhất ở trong max-heap được lưu ở gốc và các cây con bắt nguồn từ một node chứa giá trị không lớn hơn giá trị node mà cây con chứa. Min-heap được sắp xếp ngược lại; thuộc tính min-heap là với mọi node i khác node gốc,

A[CHA(i)] ≤ A[i].

Phần tử nhỏ nhất là gốc của min-heap

Với thuật toán heapsort, ta dùng max-heaps. Min-heaps thường được dùng trong hàng đợi ưu tiên, mà ta sẽ thảo luận trong Mục 6.5. Ta sẽ tóm lược rõ khi nào ta cần max- heap hoặc min-heap cho từng ứng dụng cụ thể và khi các thuộc tính được áp dụng cho max-heap và cà min-heap, ta sẽ chỉ gọi là "heap".

Xem heap như là một cây, ta định nghĩa độ cao của một node là một heap có số đường dài nhất từ gốc tới lá và ta định nghĩa độ cao của heap là độ cao của gốc. Vì một heap có n phần tử dựa trên một cây nhị phân hoàn chỉnh, độ cao là O(lg n) (xem bài tập 6.1-2). Ta thấy rằng các phép tính cơ bản trên heap chạy với thời gian tỷ lệ với độ cao cảu cây và do đó mất O(lg n) thời gian. Phần còn lại của chương này giới thiệu một vài thủ tục cơ bản và biểu diễn bằng cách nào chúng được dùng trong thuật toán sắp sếp và cấu trúc dữ liệu hàng đợi ưu tiên.

- Thủ tục MAX-HEAPIFY, chạy với thời gian O(lg n), là chìa khóa để duy trì thuộc tính max-heap.
- Thủ tục BUILD-MAX-HEAP, chạy với thời gian tuyến tính, thực hiện một max-heap từ một mảng input chưa sắp xếp.
- Thủ tục HEAPSORT, chạy với thời gian O(lg n), sắp sếp mảng tại chỗ.
- Thủ tục MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY và HEAP-MAXIMUM chạy với thời gian là O(lg n), cho phép cấu trúc dữ liệu heap thực hiện một hàng đợi ưu tiên.

Bài tập

(...)

6.2 duy trì thuộc tính của heap

Để duy trì tuộc tính max-heap, ta gọi thủ tục MAX-HEAPIFY. Với input là mảng A và chỉ số i vào mảng. Khi ta gọi, MAX-HEAPIFY giả sử rằng cây nhị phân có gốc tại TRÁI(i) và PHẢI(i) là max-heap, nhưng lúc đó $A[i]$ lại nhỏ hơn con của nó, do đó vi phạm đến thuộc tính của max-heap. MAX-HEAPIFY cho giá trị $A[i]$ "trôi nổi" trong max-heap để cây con i thỏa mãn thuộc tính max-heap.

MAX-HEAPIFY(*A*,*i*)

1 l = Left(i)

2 r = Right(i)

3 **if** $l \leq A.heap\text{-}size$ và $A[l] > A[i]$

4 *largest* = l

5 **else** *largest* = i

6 **if** $r \leq A.heap\text{-}size$ và $A[r] > A[largest]$

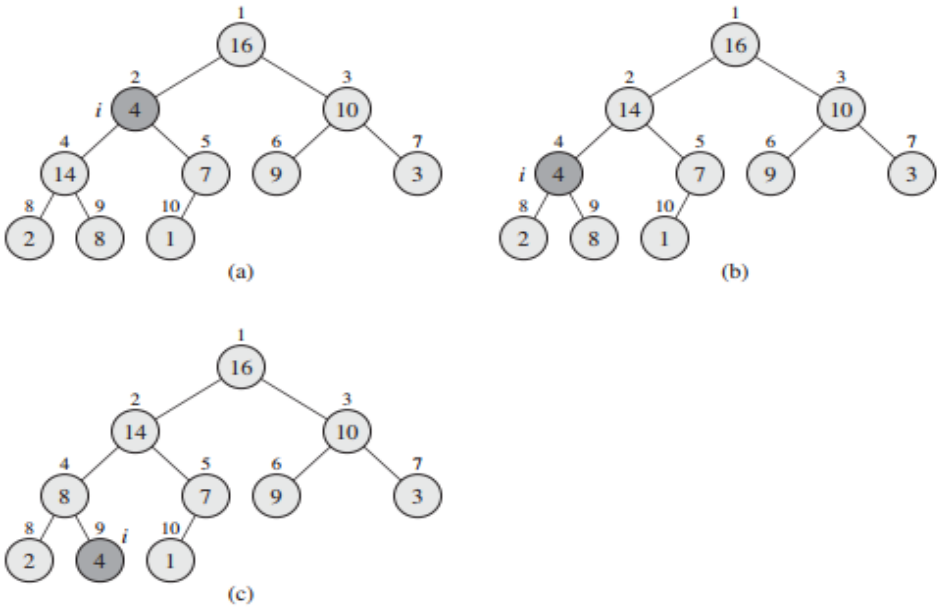
7 *largest* = r

8 **if** *largest* $\neq i$

9 *đảo* $A[i]$ với $A[largest]$

10 MAX-HEAPIFY(*A*,*largest*)

Hình 6.2 biểu diễn cách chạy của MAX-HEAPIFY. Bước đầu tiên, xác định phần tử lớn nhất từ A[i] , A[LEFT(i)] và A[PHẢI(i)] và được đánh dấu là *largest.* Nếu A[i] là lớn nhất, cây con bắt nguồn từ node i đã là một max-heap và thủ tục chấm dứt. Mặt khác, một trong hai con là phần tử lớn nhất và A[i] đảo với A[largest], trường hợp này i và con của



Hình 6.2 trình tự chạy chạy cảu MAX-HEAPIFY(*A*,2), khi *A.heap-size* = 10. **(a)** hình ban đầu, với *A*[2] với node i = 2 vi phạm thuộc tính max-heap khi mà nó không lớn hơn hai con của nó. Thuộc tính max-heap được phục hồi cho node 2 ở **(b)** bằng cách đảo *A*[2] và *A*[4], nhưng lại khiến cho thuộc tính của node 4 bị vi phạm. Gọi đệ quy

MAX-HEAPIFY(*A*,4) với i bây giờ là 4. Sau khi đảo *A*[4] với *A*[9], được biểu diễn trên **(c)**, node 4 được sửa và với gọi đệ quy MAX-HEAPIFY(*A*,9) không còn ảnh hưởng đến cấu trúc của dữ liệu.

nó thỏa mãn thuộc tính max-heap. Node được đánh dấu là largest, tuy nhiên, bây giờ có giá trị ban đầu là A[i] và do đó cây con bắt nguồn từ largest sẽ vi phạm thuộc tính ma-heap. Hậu quả là, ta gọi đệ quy MAX-HEAPIFY trên cây con đó.

Thời gian chạy của MAX-HEAPIFY trên một cây con bắt nguồn từ node i có kích thước là n là O(1) để sửa các mối liên hệ giữa các phần tử A[i], A[TRÁI(i)], A[PHẢI(i)], thêm vào đó thời gian để chạy MAX-HEAPIFY trên một cây con có gốc là con của node i (giả sử gọi đệ quy). Mỗi cây con con có kích thước lớn nhất là 2n/3–trường hợp tệ nhất xảy ra khi tầng đáy của cây đầy đúng phân nửa-do đó ta có thể miêu tả thời gian chạy của MAX-HEAPIFY bằng phép truy hồi

$T(n) \le T(2n/3) + O(1)$.

Giải pháp cho phép truy hồi, với trường hợp 2 của định lý tổng quát (định lý 4.1), là $T(n) = O(\lg n)$. Nói cách khác, ta có thể phân biệt thời gian chạy của MAX-HEAPIFY với node có độ cao *h* là $O(h)$.

Bài tập

(...)

6.3 Xây dựng một heap

Ta có thể sử dụng thủ tục MAX-HEEAPIFY từ dưới lên để chuyển mảng *A*[1..*n*], với *n* = *A.length*, thành một max-heap. Ở bài tập 6.1-7, các phần tử ở cây con $A[(\lfloor n/2 \rfloor + 1)..n]$ toàn bộ là lá của cây và mỗi phần tử là một phần tử heap để bắt đầu. Thủ tục của BUILD-MAX-HEAP sẽ đi qua các node còn lại của cây và chạy MAX-HEAPIFY trên từng node.

BUILD-MAX-HEAP(*A*)

1 *A.heap-size* = *A.length*

2 **for** $i = \lfloor A.length/2 \rfloor$ **downto** 1

3 MAX-HEAPIFY(*A*,*i*)

Hình 6.3 chỉ ra ví dụ cách chạy của BUILD-MAX-HEAP.

Để chỉ ra rằng làm thế nào BUILD-MAX-HEAP hoạt động đúng, ta có thể dùng lặp bất biến sau:

Khi bắt đầu mỗi vòng lặp for lặp ở dòng 2-3, mỗi node i + 1, i + 2,...,n là gốc của một max-heap.

Ta cần chỉ ra rằng lần lặp đầu có sự không đổi, mỗi vòng lặp duy trì sự không đổi và sự bất biến cung cấp một tính chất hữu ích để chỉ ra sự đúng đắn của vòng kết thúc.

Khởi tạo: Trước tiên là lần lặp đầu tiên của vòng lặp, $i = \lfloor n/2 \rfloor$. Mỗi node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2,...,n$ là một lá và do đó gốc là một max-heap không đáng kể.

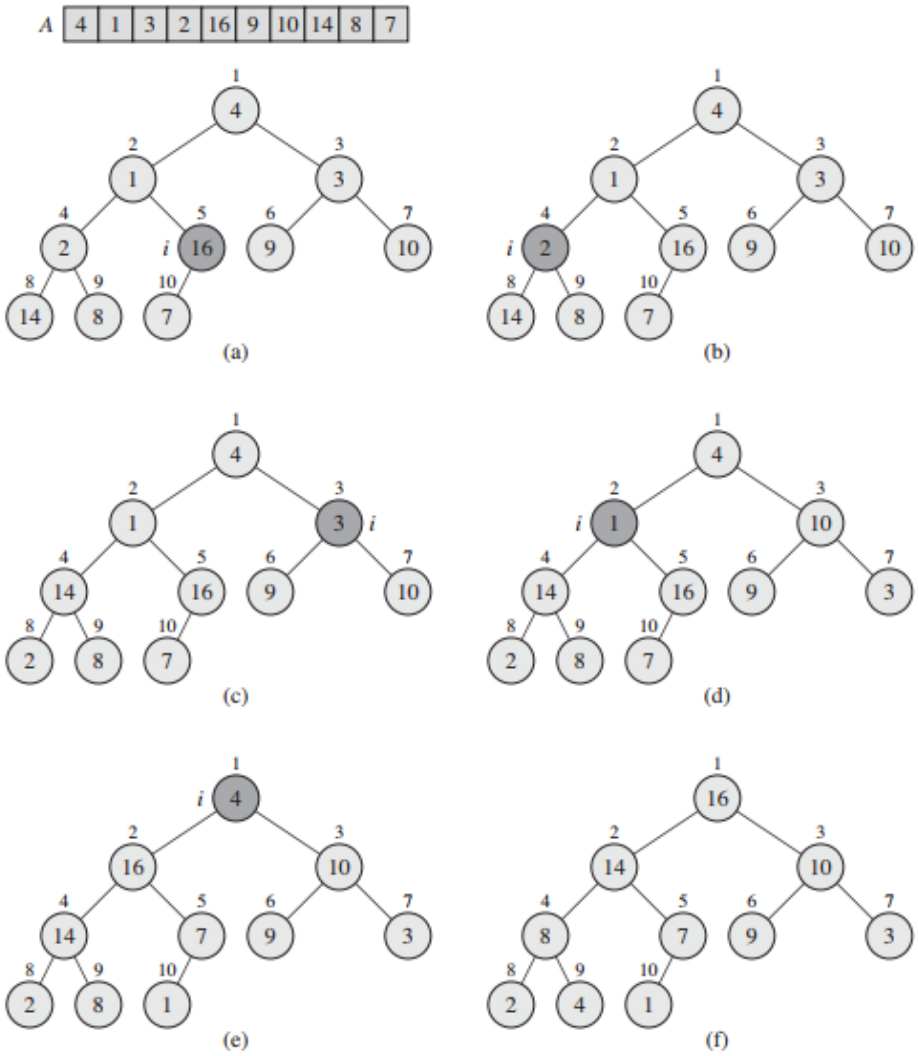
Duy trì: để thấy rằng mỗi vòng lặp duy trì một lặp bất biến, quan sát rằng mỗi node con của i được đánh số cao hơn i. Bởi lặp bất biến, do đó, chúng đều là gốc của max-heap. Điều này đúng với điều kiện gọi MAX-HEAPIFY(*A*,*i*) để tạo node *i* là một gốc của max-heap. Hơn thế nữa, MAX-HEAPIFY gọi giữ lại tính chất của của node *i* + 1, *i* + 2,...,*n* là các gốc của một max-heap. Giảm dần i trong vòng lặp for cập nhật tái thiết lặp bất biến cho lần lặp tiếp.

Chấm dứt: khi chấm dứt, *i* = 0. Với lặp bất biến, mỗi node 1,2,...,*n* là gốc của một max-heap. Đặc biệt với node 1.

Ta có thể tính toán đơn giản giới hạn trên bằng cách chạy BUILD-MAX-HEAP. Mỗi lần gọi MAX-HEAPIFY tốn O(lg n) và BUILD-MAX-HEAP O(n) cho các lần gọi. Do đó, thời gian chạy là O(n.lg n). Gới hạn trên này ,nếu như là chính xác, thì không có tiêm cận chặt chẽ.

Ta có thể miêu tả giới hạn chặt hơn bằng cách quan sát thời gian cho MAX-HEAPIFY để chạy tại một node bất kỳ với chiều cao của node trên cây và độ cao của phần lớn các node là nhỏ. Thuật phân tích chặt chẽ hơn của chúng ta dực trên thuộc tính của một heap có n phần tử có độ cao $\lfloor \lg n \rfloor$ (xem bài tập 6.1-2) và phần lớn $\lceil n/2^{h+1} \rceil$ node có độ cao h(xem bài tập 6.3-3).

Thời gian yêu cầu của MAX-HEAPIFY khi ta gọi một node có độ cao h là O(h) và khi đó ta có thể biểu diễn tổng cần có của BUILD-MAX-HEAP như là giới hạn trên bởi



Hình 6.3 Biểu diễn BUILD-MAX-HEAP chạy , cho cấu trúc dữ liệu trước khi gọi MAX-HEAPIFY ở dòng 3 của BUILD-MAX-HEAP. **(a)** Một mảng có 10 phần tử và một cây nhị phân biểu diễn mảng đó. Hình cho ta thấy lặp *i* với node 5 trước khi gọi MAX-HEAPIFY(A,i). **(b)** Cấu trúc dữ liệu kết quả. Vòng lặp *i* với lần lặp tiếp theo là node 4. **(c)-(e)** Vòng lặp tiếp theo với lặp for trong BUILD-MAX-HEAP. Ta thấy rằng khi nào MAX-HEAPIFY được gọi ở một node, hai cây con của node đó đều là max-heap.**(f)** Max-heap sau khi BUILD-MAX-HEAP hoàn thành.

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right).$$

Ta đánh giá tổng cuối cùng bằng cách thay thế $x = 1/2$ ở công thức (A.8)

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{\frac{1}{2}}{\left(1 - \frac{1}{2}\right)^2}$$

$$= 2 .$$

Do đó, ta có thể giới hạn thời gian chạy của BUILD-MAX-HEAP là

$$O\left(n\sum_{h=0}^{\lfloor \lg n \rfloor}\frac{h}{2^h}\right) = O\left(n\sum_{h=0}^{\infty}\frac{h}{2^h}\right)$$

$$= O(n)$$

Vì thế, ta có thể xây dựng một max-heap từ một mảng chưa sắp xếp với thời gian tuyến tính.

Ta có thể xây dựng một min-heap bằng thủ tục BUILD-MIN-HEAP, với cách giống như BUILD-MAX-HEAP nhưng ta gọi MAX-HEAPIFY ở dòng 3 thay bởi lệnh gọi MIN-HEAPIFY (xem bài tập 6.2-2). BUILD-MIN-HEAP tạo một min-heap từ mảng chưa sắp xếp với thời gian tuyến tính.

Bài tập

(...)

6.5 Hàng đợi ưu tiên

Heapsort là một thuật toán tuyệt vời, nhưng với một bộ cài đặt quicksort tốt, sẽ giới thiệu ở Chương 7, thường sẽ đánh bại heapsort trong thực tế. Tuy nhiên, cấu trúc dữ liệu heap có rất nhiều cách để sử dụng. Trong phần này, ta nói đến một trong những ứng dụng phổ biến nhất heap: hàng đợi ưu tiên hiệu quả. Như là heap, hàng đợi ưu tiên có hai dạng: hàng đợi ưu tiên lớn và hàng đợi ưu tiên nhỏ. Chúng ta sẽ tập trung vào làm thế nào để cài đặt một hàng đợi ưu tiên lớn, mà nó dựa trên max-heap; bài tập 6.5-3 yêu cầu bạn phải viết thủ tục cho hàng đợi ưu tiên nhỏ.

Hàng đợi ưu tiên là một cấu trúc dữ liệu để duy trì một bộ S các phần tử, mỗi phần tử liên kết với một giá trị gọi là ***key***. ***Hàng đợi ưu tiên lớn*** hỗ trợ các việc sau:

CHÈN(S,x) chèn vào phần tử x vào bộ S, tương đương với phép toán $S = S \cup \{x\}$.

TÌM-MAX(S) trả về phần tử trong S có giá trị key lớn nhất.

TRÍCH-MAX(S) thay thế và trả về phần tử trong S có giá trị key lớn nhất.

TĂNG-KEY(S,x,k) tăng giá trị key của phần tử x lên giá trị k, giả sử giá trị key bằng hoặc nhỏ hơn k.

Trong các ứng dụng khác, ta có thể dùng hàng đợi ưu tiên lớn để lên kế hoạch công việc trên một máy tính dùng chung. Hàng đợi ưu tiên lớn luôn theo dõi công việc được thực hiện và việc ưu tiên tương đối. Khi một công việc kết thúc hoặc bị gián đoạn, kế hoạch lựa chọn các việc ưu tiên cao hơn từ các việc đang chờ giải quyết bằng cách dùng TRÍCH-MAX. Có thể thêm việc mới vào hàng đợi bất cứ lúc nào bằng cách dùng CHÈN.

Mặt khác, hàng đợi ưu tiên nhỏ hỗ trợ CHÈN, TÌM-MIN, TRÍCH-MIN và GIẢM-KEY. Một hàng đợi nhỏ có thể dùng trong trình nô phỏng sự kiện. Các vật thể trong hàng đợi là sự kiện được mô phỏng, mỗi cái liên kết với thời gian xuất hiện được voi như là key của nó. Các sự kiện phải được mô phỏng với thời gian xuất hiện của nó, vì mô phỏng một sự kiện có thể dẫn đến các sự kiện khác được mô phỏng trong tương lai. Chương trình mô phỏng gọi TRÍCH-MIN ở mỗi bước để chọn sự kiện mô phỏng tiếp theo. Như là sự kiện mới được sản xuất, chương trình chèn sự kiện vào hàng đợi bằng cách dùng CHÈN. Ta sẽ thấy các cách dùng khác cho hàng đợi ưu tiên nhỏ, làm nổi bật GIẢM-KEY, ở Chương 23 và 24.

Không ngạc nhiên khi ta có thể dùng heap để cài đặt một hàng đợi ưu tiên. Ở ứng dụng trên, như lên kế hoạch công việc hoặc mô phỏng sự kiện, các phần tử của hàng đợi trao đổi với các đối tượng trong ứng dụng. Ta thường cần xác định đối tượng ứng dụng nào trao đổi với phần tử của hàng đợi được cho và ngược lại. Khi ta dùng heap để cài đặt một hàng đợi ưu tiên, do đó, ta thường cần lưu một ***tay cầm*** để trao đổi với phần tử heap với mỗi đối tượng của ứng dụng. Ở đây, tay cầm thường là một mảng. Vì phần tử của heap thay đổi vị trí trong mảng trong suốt quá trình của heap, như là một bộ cài đặt thực tế, khi di chuyển một phần tử heap, cũng sẽ phải cập nhật lại mảng với đối tượng ứng dụng tương ứng. Vì chi tiết việc truy cập các đối tượng phụ thuộc nhiều vào ứng dụng và bộ cài đặt, ta sẽ không nói đến nó ở đây, nên lưu ý nó ở thực tế, các tay cầm đó cần được duy trì chính xác.

Giờ ta sẽ thảo luận làm cách nào để cài đặt phương thức cho một hàng đợi lớn. Thủ tục của HEAP-MAXIMUM cài đặt TÌM-MAX trong O(1) thời gian.

HEAP-MAXIMUM(*A*)

1 **return** *A*[1]

Thủ tục HEAP-EXTRACT-MAX cài đặt EXTRACT-MAX. Nó tương tự như thân của vòng lặp for (dòng 3-5) của thủ tục HEAPSORT.

HEAP-EXTRACT-MAX(A)

1 **if** *A.heap-size* < 1

2 **error** "heap đang tràn"

3 *max* = *A*[1]

4 *A*[1] = *A*[*A.heap-size*]

5 *A.heap-size* = *A.heap-size* – 1

6 MAX-HEAPIFY(*A*,1)

7 **return** *max*

Thời gian chạy của HEAP-EXTRACT-MAX là *O*(lg *n*), vì nó biểu diễn chỉ một hằng trong các việc trên đầu với *O*(lg *n*) thời gian cho MAX-HEAPIFY.

Thủ tục HEAP-INCREASE-KEY cài đặt INCREASE-KEY. Ký tự *i* vào trong mảng xác định phần tử của hàng đợi có key mà chúng ta muốn tăng. Thủ tục cập nhật key của phần tử *A*[*i*] lần đầu cho nó một giá trị mới. Vì tăng key của *A*[*i*] có thể gây hại cho tính chất của max-heap, thủ tục khi đó, theo cách gợi lại vòng lặp chèn (dòng 5-7) của INSERTION-SORT từ phần 2.1, đi từ con đường đơn giản từ node thằng tới gốc để tìm một thích hợp cho key mới đã tăng. Như là HEAP-INCREASE-KEY đi cong đường này, nó lặp lại việc so sánh phần tử với cha của nó, thay key của chúng và tiếp tục nếu key của phần tử là lớn hơn và kết thúc nếu key của phần tử nhỏ hơn, vì tính chất của max-heap phải được giữ lại. (xem bài tập 6.5-5 cho một vòng lặp bất biến chuẩn xác.)

HEAP-INCREASE-KEY(A,i,key)

1 **if** *key* < *A*[*i*]

2 **error** "key mới nhỏ hơn key hiện tại"

3 *A*[*i*] = key

4 **while** *i* > 1 và *A*[CHA(*i*)] < *A*[*i*]

5 đổi $A[i]$ với $A[\text{CHA}(i)]$

6 $i = \text{CHA}(i)$

Hình 6.5 cho ta thấy ví dụ của một HEAP-INCREASE-KEY. Thời gian chạy của HEAP-INCREASE-KEY với heap có n phần tử là $O(\lg n)$, vì con đường dẫn từ node được cập nhật ở dòng 3 lên gốc có độ dài là $O(\lg n)$.

Thủ tục MAX-HEAP-INSERT cài đặt INSERT. Nó cần nhập key vủa phần tử mới để chèn vào max-heap A. Đầu tiên thủ tục mở rộng max-heap bằng cách thêm vào cây một lá mới có key là $-\infty$. Sau đó gọi HEAP-INCREASE-KEY để đặt key của node mới về giá trị đúng và duy trì tính chất của max-heap.

MAX-HEAP-INSERT(A,*key*)

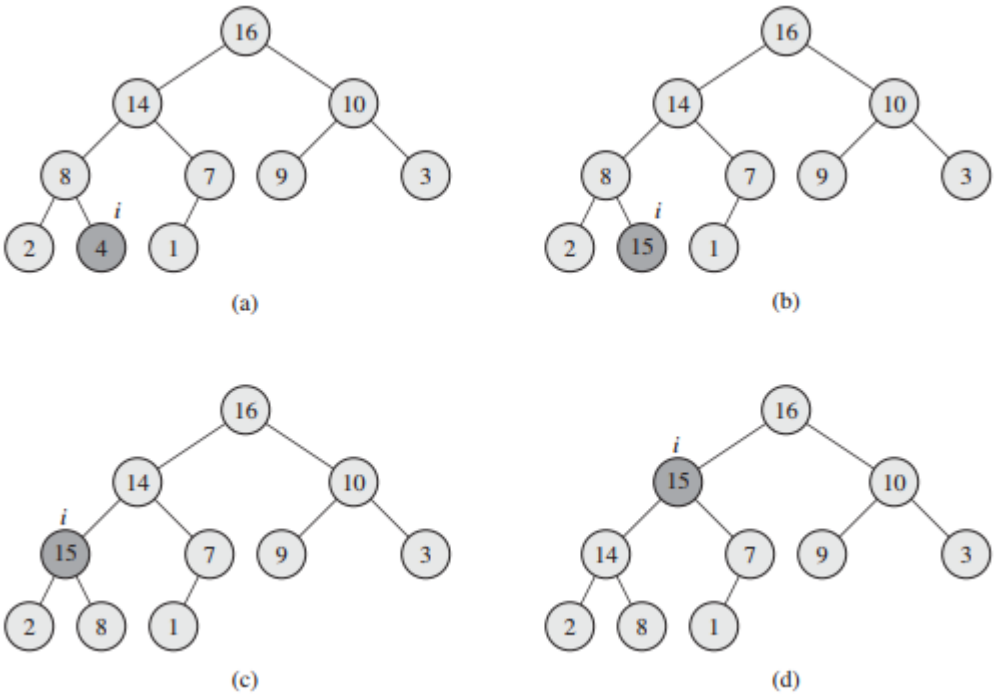
1 $A.heap\text{-}size = A.heap\text{-}size + 1$

2 $A[A.heap\text{-}size] = -\infty$

3 HEAP-INCREASE-KEY (A,$A.heap\text{-}size$,*key*)

Thời gian chạy của MAX-HEAP-INSERT của heap có n phần tử là $O(\lg n)$.

Tổng kết, một heap có thể hỗ trợ bất kỳ hàng đợi ưu tiên nào với bộ giá trị có kích thước n với thời gian $O(\lg n)$.



Hình 6.5 trình tự chạy của HEAP-INCREASE-KEY. **(a)** Max-heap của hình 6.4(a) với một node có ký tự là được tô đậm. **(b)** Node này có key tăng lên 15. **(c)** Sau khi một vòng lặp của vòng while ở dòng 4-6, node và cha của nó đổi key với nhau và ký tự *i* tiến lên cha. **(d)** Max-heap sau một vòng lặp nữa của vòng while. Ở thời điểm này, $A[\text{CHA}(i)] \geq A[i]$. Tính chất của max-heap giờ được bảo đảm và thủ tục kết thúc.

Bài tập

(...)

Chương 9: Trung vị và thống kê thứ tự

Thống kê thứ tự thứ i của một tập hợp n phần tử là phần tử thứ i nhỏ nhất. Ví dụ, giá trị cực tiểu của một tập hợp là thống kê thứ tự thứ nhất $(i = 1)$, và giá trị cực đại là thống kê thứ tự thứ n $(i = n)$. Một **trung vị**, hay nói cách khác, là "điểm chia đôi" của tập hợp. Với n lẻ, trung vị là duy nhất tại $i = (n + 1)/2$. Với n chẵn, ta có 2 trung vị tại $i = n/2$ và $i = \frac{n}{2} + 1$. Vì thế, với mọi n, trung vị luôn tại $i = \lfloor (n + 1)/2 \rfloor$ (**trung vị nhỏ**) và $i = \lceil (n + 1)/2 \rceil$ (**trung vị lớn hơn**). Đơn giản hơn trong tài liệu này, song, ta sử dụng cumhj từ "trung vị" để chỉ trung vị nhỏ.

Chương này đề cập đến vấn đề lựa chọn thống kê thứ tự thứ i từ một tập n phần tử khác nhau. Ta giả sử rằng vấn đề như sau:

Dữ liệu vào: một tập A gồm n số (khác nhau) và số nguyên i với $1 \leq i \leq n$.

Dữ liệu ra: phần tử $x \epsilon A$ mà lớn hơn chính xác (i - 1) phần tử còn lại thuộc A.

Ta có thể giải quyết bài toán chọn với thời gian O(n lgn), vì ta có thể sắp xếp các số bằng sắp xếp đống hay sắp xếp trộn và xuất ra phần tử thứ i của mảng đã sắp xếp. Chương này sẽ đưa ra thuật toán giải quyết vấn đề đó nhanh hơn.

Ở **mục 9.1**, ta giải quyết vấn đề chọn cực tiểu và cực đại của một tập hợp. Thú vị hơn là vấn đề lựa chọn chung, vấn đề mà ta sẽ tìm hiểu trong 2 phần tiếp theo. **Mục 9.2** phân tích thuật toán ngẫu nhiên thực tiễn đạt được thời gian mong muốn O(n). Còn **mục 9.3** gồm thuật toán về mặt lý thuyết đạt được thời gian thực thi O(n) trong trường hợp xấu nhất.

9.1 Cực tiểu và cực đại

Cần bao nhiêu phép so sánh để xác định giá trị nhỏ nhất trong tập n phần tử? Ta có thể dễ dàng biết được n – 1 phép so sánh: kiểm tra lần lượt mỗi phần tử trong tập và giữ lại giá trị nhỏ nhất. Theo thủ tục sau đây, ta giả định tập là mảng A, với A.length = n.

```
MINIMUM(A)
1  min = A[1]
2  for i = 2 to A.length
3       if min > A[i]
4            min = A[i]
5  return min
```

Tương tự, với n – 1 phép so sánh, ta có thể tìm được giá trị cực đại.

Liệu rằng đó có phải là cách tốt nhất ta làm được? Đúng vậy, bởi ta có thể tìm cận dưới của n – 1 phép so sánh trong vấn đề tìm cực tiểu. Ta hãy xem thuật toán tìm cực tiểu như là một "giải đấu" giữa các phần tử. Mỗi phép so sánh là một trận đấu nơi mà phần tử nhỏ hơn sẽ "chiến thắng". Nhận thấy rằng, mọi phần tử ngoại trừ phần tử cực tiểu – "người thắng cuộc" – sẽ phải "thua" ít nhất một trận đấu, ta kết luận rằng cần n-1 phép so sánh để xác định cực tiểu. Do đó, thuật toán tìm kiếm **CỰC TIỂU** là tối ưu đối với số lần thực hiện so sánh.

Cực tiểu và cực đại đồng thời

Trong nhiều ứng dụng, ta phải tìm cả cực tiểu lẫn cực đại trong tập hợp n phần tử. Cho ví dụ, một chương trình đồ họa cần căn tỉ lệ một tập dữ liệu (x, y) để vừa với hình chữ nhật hiển thị trên màn hình hay một thiết bị đồ họa kết xuất khác. Để làm được điều đó, trước hết chương trình phải xác định giá trị cực đại và cực tiểu của mỗi tạo độ.

Khi đó, hiển nhiên cần biết để xác định cực tiểu và cực đại trong n phần tử chỉ sử dụng $\Theta(n)$ phép so sánh, tiệm cận tối ưu: đơn giản hóa việc tìm kiếm cực đại và cực tiểu độc lập, sử dụng n – 1 phép so sánh với cả hai, tổng cộng là 2n – 2 phép so sánh.

Thực tế rằng, ta có thể tìm thấy cả cực tiểu lẫn cực đại chỉ với $3\lfloor n/2 \rfloor$ phép so sánh. Ta duy trì các phần tử cực tiểu và cực đại cho đến hết. Tuy nhiên, thay vì xử lý từng phần tử đầu vào bằng việc so sánh với cực đại và cực tiểu hiện hành, điều này sẽ tốn 2 phép so sánh cho mỗi phần tử, ta tiến hành so theo cặp. Ta so sánh cặp các phần tử theo dữ liệu đầu vào với mỗi cặp còn lại, và sau đó ta so sánh nhở hơn với cực tiểu hiện hành và lớn hơn với cực đại hiến hành, sẽ chỉ tốn 3 phép so sánh cho với mỗi 2 phần tử.

Ta cài đặt giá trị khởi tạo cho cực tiểu hiện hành và cực đại hiện hành phụ thuộc n chẵn hay lẻ. Nếu n lẻ, ta đặt cực tiểu hiện hành lẫn cực đại hiện hành là phần tử đầu tiên, và sau đó ta tiến hành xử lý các phần tử còn lại theo từng cặp. Nếu n chẵn, trước tiên ta thực hiện 1 phép so sánh 2 phần

tử đầu tiên nhằm xác định giá trị khởi tạo cho cực tiểu và cực đại, và sau đó ta cũng tiến hành xử lý các phần tử còn lại theo từng cặp tương tự trường hợp n lẻ.

Ta phân tích về tổng số phép so sánh. Nếu n lẻ, thì ta thực hiện 3[n/2] phép so sánh. Nếu n chẵn, ta tiền xử lý một phép so sánh và sau đó là 3(n – 2)/2 phép so sánh, tổng cộng 3n/2 – n. Vì thế, trong cả hai trường hợp, tổng số phép so sánh nhiều nhất là 3[n/2].

9.2 Lựa chọn trong thời gian kỳ vọng tuyến tính

Vấn đề lựa chọn chung thường khó hơn bài toán đơn giản tìm cực tiểu. Tuy nhiên, thời gian thực hiện cho cả hai là gần như nhau: $\Theta(n)$. Ở mục này, chúng tôi trình bày một thuật toán chia – để – trị để giải quyết bài toán lựa chọn. Đó là thuật toán **lựa chọn ngẫu nhiên (RANDOMIZED-SELECT)** mô hình hóa dựa trên thuật toán sắp xếp nhanh ở chương 7. Như trong sắp xếp nhanh, ta phân hoạch mảng ban đầu bằng đệ quy. Nhưng khác với sắp xếp nhanh, ta xử lý đệ quy cả hai phía của phân hoạch, lựa chọn ngẫu nhiên chỉ thực hiện trên một phía của phân hoạch. Điểm khác biệt này được thể hiện trong phân tích: Trong khi kĩ thuật sắp xếp nhanh có thời gian thực thi là $\Theta(nlgn)$, còn thời gian thực thi của chọn ngẫu nhiên là $\Theta(n)$, mặc định rằng các phần tử khác nhau.

Chọn ngẫu nhiên sử dụng hàm **phân hoạch ngẫu nhiên** được giới thiệu ở mục 7.3. Vì thế, giống như **sắp xếp nhanh ngẫu nhiên**, nó cũng là một thuật toán ngẫu nhiên hóa, bởi các hoạt động của nó được xác định bởi đầu ra của bộ phát sinh số ngẫu nhiên. Các dòng mã lệnh cho thuật toán chọn ngẫu nhiên sau đây trả về giá trị nhỏ nhất phần tử thứ i của mảng A[p..r].

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2       return A[p]
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k            // the pivot value is the answer
6       return A[q]
7  elseif i < k
8       return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

Hàm chọn ngẫu nhiên thực thi như sau. Dòng 1 kiểm tra về trường hợp cơ bản theo đệ quy, khi mà tập con A[p..r] có duy nhất 1 phần tử. Trong trường hợp này, i phải bằng 1, vì vậy ta đơn giản trả về A[p] tại dòng 2 là giá trị thứ i nhỏ nhất. Ngược lại, sau khi gọi hàm RANDOMIZED-PARTITION ở dòng 3 phân mảng A[p..r] thành hai (có thể rỗng) mảng con A[p..q – 1] và A[q + 1..r] sao cho mỗi phần tử thuộc A[p..q – 1] nhỏ hơn hoặc bằng A[q] và mỗi phần tử thuộc A[q + 1..r] lớn hơn A[q]. Giống như sắp xếp nhanh, ta xem A[q] như là một chốt (lính canh). Dòng 4 tính chỉ số k, đó là, chỉ số của phần tử thuộc phân hoạch thấp A[p..q], cộng thêm 1 cho phần tử chốt. Dòng 5 kiểm tra xem A[q] có phải là phần tử thứ i nhỏ nhất. Nếu đúng, dòng 6 trả về kết quả A[q]. Ngược lại, thuật toán xác định xem thành phần nhỏ nhất thứ ith nằm ở đâu trong 2 mảng con. Nếu i < k, thì phần tử mong muốn nằm trên tập phân hoạch thấp và được lựa chọn bằng đệ quy ở dòng 8. Nếu i > k, thì phần tử mong muốn nằm trên tập phan hoạch cao. Bởi ta đã biết giá trị k nhỏ hơn i, ta đệ quy tìm phần tử mong muốn thứ (i – k) nhỏ nhất thuộc mảng A[q+1..r]. Mã lệnh xuất hiện trường hợp chấp nhânj gọi đệ quy đối với tập con 0 phần tử.

Trường hợp tệ nhất thời gian thực thi cho việc tìm kiếm ngẫu nhiên là $\Theta(n^2)$, thậm chí là trong trường hợp tìm cực tiểu. Bởi vì ta có thể cực kì không may mắn và luôn phân hoạch xung quanh phần tử lớn nhất, và quá trình phân hoạch mất $\Theta(n)$. Ta thấy được thuật toán này có thời gian thực thi tuyến tính, dù vậy, bởi vì nó ngẫu nhiên, không có đầu vào đặc biết nào dẫn tới trường hợp xấu nhất cả.

Để phân tích thời gian dự trù của chọn ngẫu nhiên, ta cho chạy trên bộ dữ liệu đàu vào mảng A[p..r] của n phần tử là các biến ngẫu nhiên mà ta kí hiệu là T(n), và ta có điều kiện ràng buộc giới hạn trên E[T(n)] như sau. Hàm Phân hoạch – ngẫu nhiên trả về bất kì giá trị phân tử nào như là chốt. Do đó, với mỗi k sao cho $1 \leq k \leq n$, mảng con A[p..q] chứa k phần tử (nhỏ hơn hoặc bằng lính canh) có xác suất 1/n. cho k = 1, 2,…, n, ta định nghĩa biến ngẫu nhiên X_k là

X_k = I {tập con A[p..r] chứa chính xác k phần tử}, và vì vậy, giả sử rằng các phần tử phân biệt, ta có:

$$\mathrm{E}[X_k] = 1/n \ . \tag{9.1}$$

Khi gọi hàm Chọn ngẫu nhiên và chọn A[q] là phần tử lính canh, ta không biết, một tiên nghiệm, ta sẽ dừng ngay lặp tức nếu kết quả đúng, đệ quy trên mảng con A[p..q – 1] hoặc A[q + 1..r]. Lựa chọn này phụ thuộc

vị trí mà phần tử nhỏ nhất thứ i so với A[q]. Giả sử rằng T(n) là đơn điệu tăng, ta có thể ước lượng giới hạn trên thời gian cần cho quá trình gọi đệ quy bởi thời gian gọi đệ quy với đầu vào lớn nhất có thể. Nói cách khác, để xác định cận trên, ta giả sử rằng phần tử thứ i luôn thuộc phân hoạch với số phân tử lớn hơn. Với lời gọi hàm chọn ngẫu nhiên nhất định, biến ngẫu nhiên chỉ số X_k có giá trị là 1 với chính xác một giá trị k và 0 với các giá trị khác k. Khi $X_k = 1$, ta đệ quy với kích thước k – 1 và n – k trên hai mảng con tương ứng. Vì thế, ta có phép truy toán

$$\begin{aligned} T(n) &\le \sum_{k=1}^{n} X_k \cdot (T(\max(k-1, n-k)) + O(n)) \\ &= \sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n) \, . \end{aligned}$$

Lấy giá trị dự kiến, ta có:

$$\begin{aligned} &\mathrm{E}[T(n)] \\ &\le \mathrm{E}\left[\sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n)\right] \\ &= \sum_{k=1}^{n} \mathrm{E}[X_k \cdot T(\max(k-1, n-k))] + O(n) && \text{(by linearity of expectation)} \\ &= \sum_{k=1}^{n} \mathrm{E}[X_k] \cdot \mathrm{E}[T(\max(k-1, n-k))] + O(n) && \text{(by equation (C.24))} \\ &= \sum_{k=1}^{n} \frac{1}{n} \cdot \mathrm{E}[T(\max(k-1, n-k))] + O(n) && \text{(by equation (9.1))} \, . \end{aligned}$$

Để áp dụng đẳng thức (C.24), ta dựa vào X_k và T(max(k – 1, n – k)) là biến ngẫu nhiên độc lập. Xét biểu thức max(k – 1, n – k), ta có:

$$\max(k-1, n-k) = \begin{cases} k-1 & \text{if } k > \lceil n/2 \rceil \, , \\ n-k & \text{if } k \le \lceil n/2 \rceil \, . \end{cases}$$

Nếu n chẵn, mỗi giới hạn từ $T(\lceil n/2 \rceil)$ đến T(n – 1) xuất hiện chính xác 2 lần trong tổng, và nếu n lẻ, tất cả giới hạn xuất hiện 2 lần riêng T([n/2]) xuất hiện 1 lần. Vì thế, ta có:

$$\mathrm{E}[T(n)] \le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} \mathrm{E}[T(k)] + O(n) \, .$$

Ta chứng minh E[T(n)] = O(n) bằng phép thế. Giả sử $\mathrm{E}[\mathrm{T}(\mathrm{n})] \leq \mathrm{cn}$ với c là hằng số thỏa mãn điều kiện ban đầu của phép truy toán. Ta giả sử T(n) = O(1) với n nhỏ hơn hằng số nhiều. ta chọn hằng số sau. Dùng giả thuyết quy nạp này, ta có:

$$
\begin{aligned}
\mathrm{E}[T(n)] &\leq \frac{2}{n}\sum_{k=\lfloor n/2\rfloor}^{n-1} ck + an \\
&= \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2\rfloor-1} k\right) + an \\
&= \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2\rfloor - 1)\,\lfloor n/2\rfloor}{2}\right) + an \\
&\leq \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2}\right) + an \\
&= \frac{2c}{n}\left(\frac{n^2-n}{2} - \frac{n^2/4 - 3n/2 + 2}{2}\right) + an \\
&= \frac{c}{n}\left(\frac{3n^2}{4} + \frac{n}{2} - 2\right) + an \\
&= c\left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n}\right) + an \\
&\leq \frac{3cn}{4} + \frac{c}{2} + an \\
&= cn - \left(\frac{cn}{4} - \frac{c}{2} - an\right) .
\end{aligned}
$$

Để hoàn tắt chứng minh, ta cần chứng tỏ rằng đủ với n lớn, biểu thức cuối đạt tối đa cn hay tương đương là $cn/4 - c/2 - an \geq 0$. Cộng c/2 vào hai vế và rút n làm thừa số chung ta được $n(c/4 - a) \geq c/2$. Miễn là ta chọn hằng số csao cho $c/4 - a > 0$ hay $c/4 > a$, ta chia 2 vế cho $c/4 > a$, được:

$$n \geq \frac{c/2}{c/4 - a} = \frac{2c}{c - 4a} .$$

Vì thế, nếu giả sử T(n) = O(1) với $n < 2c/(c - 4a)$, thì E[T(n)] = O(n). Ta kết luận rằng cơ thể tìm kiếm bất kì thống kê thứ tự, và đặc biệt là trung vị, trong thời gian tuyến tính, giả định các phân tử khác biệt.

9.3 Lựa chọn trong thời gian tuyến tính với trường hợp xấu nhất

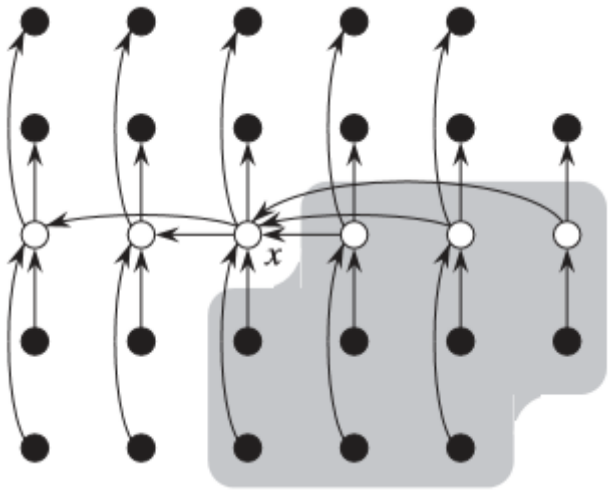
Bây giờ ta xét xem một thuật toán lựa chọn với thời gian thực thi $\Theta(n)$ trong trường hợp xấu nhất. Như chọn ngẫu nhiên, thuật toán SELECT (CHỌN) tìm phân tử mong muốn bằng phân hoạch đệ quy dữ liệu đầu vào. Tuy nhiên, ý tưởng sau thuật toán đó là đảm bảo sự phân chia tốt khi

mảng được phân hoạch. CHỌN dùng phân hoạch xác định của thuật toán phân hoạch từ sắp xếp nhanh (xem mục 7.1), nhưng thay đổi để lấy các yếu tố nhằm phân hoạch xung quanh một tham số đầu vào.

Thuật toán chọn xác định phân tử nhỏ nhất thứ i của một mảng số n phân tử (n > 1) phân biệt bằng cách thực hiện các bước như sau. (Nếu n = 1, thì chọn trả về giá trị đầu vào duy nhất của nó như phần tử thứ i nhỏ nhất)

1. Chia mảng n phần tử thành $\lfloor n/5 \rfloor$ nhóm gồm 5 phần tử và có tối đa một nhóm có số phần tử là n mod 5 thành phần còn lại.
2. Tìm trung vị của mỗi trong số $\lceil n/5 \rceil$ nhóm bằng sắp xếp chèn các thành phần của mỗi nhóm (có 5 là tối đa) và lấy phần tử trung vị từ danh sách đã sắp xếp của nhóm.
3. Dùng đệ quy chọn để tìm trung vị x của $\lceil n/5 \rceil$ các trung vị tìm được ở bước 2. (nếu số lượng trung vị là chẵn, theo quy ước của chúng tôi, x là trung vị nhỏ hơn).
4. Phân hoạch mảng đầu vào xung quanh trung vị của trung vị x bằng phiên bản thay đổi của PARTITION. Cho k là số lượng các phần tử thuộc phân hoạch dưới, do vậy x là phân tử thứ k nhỏ nhất và có n – k phân tử thuộc phân hoạch trên.
5. Nếu i = k, trả về x. Ngược lại, dùng CHỌN đệ quy tìm phân tử nhỏ nhất thứ i ở tập dưới nếu i < k, hoặc phần tử nhở nhất thứ (i - k) ở tập trên nếu i > k.

Phân tích vê thời gian thực thi của CHỌN, trước tiên ta xác định một cận dưới số lượng phần tử mà lớn hơn phần tử phân hoạch x. Hình 9.1 cho ta hình dung được tiến trình theo dõi thực hiện này.



Hình 9.1 phân tích thuật toán chọn.

Có ít nhất một nửa các trung vị được tìm trong bước 2 lớn hơn hoặc bằng trung vị của trung vị x. Vì thế, ít nhất một nửa $\lceil n/5 \rceil$ nhóm cho 3 phần

tử lớn hơn x, ngoại trừ nhóm 1 có ít hơn 5 phân tử nếu n không chia hết cho 5, và một nhóm sẽ chứa chính x. Trừ 2 nhóm này, sẽ dẫn đến số lượng phân tử lớn hơn x ít nhất là

$$3\left(\left\lceil\frac{1}{2}\left\lceil\frac{n}{5}\right\rceil\right\rceil-2\right) \geq \frac{3n}{10}-6\,.$$

Tương tự, có ít nhất 3n/10 – 16 phần tử nhỏ hơn x. Vì thế, trong trường hợp xấu nhất, ở bước 5 gọi đệ quy SELECT trên tối đa 7n/10 + 6 phần tử.

Bây giờ, ta có thể phát triển phép truy toán cho thời gian thực hiện trong trường hợp xấu nhất T(n) của thuật toán SELECT. Các bước 1, 2 và 4 mất thời gian thực hiện O(n) (bước 2 bao gồm O(n) lần gọi sắp xếp chèn trên tập hợp kích cỡ O(1)). Bước 3 mất T($\lceil n/5\rceil$) và bước 5 mất nhiều nhất T(7n/10 + 6), giả sử T đơn điệu tăng. Ta có giả thiết, rằng bất kì dữ liệu đầu vào nào ít hơn 140 phần tử cần thời gian O(1). nguồn gốc của hằng số ma thuật 140 sẽ rõ ràng ngay sau đây. Vì vậy ta có phép truy toán:

$$T(n) \leq \begin{cases} O(1) & \text{if } n < 140\,, \\ T(\lceil n/5\rceil) + T(7n/10+6) + O(n) & \text{if } n \geq 140\,. \end{cases}$$

Ta chứng minh thời gian thực thi là tuyến tính bằng phép thế. Đặc biệt hơn, ta sẽ chứng minh rằng $T(n) \leq cn$ với c là hằng số lớn thích hợp và n > 0. Ta giả định $T(n) \leq cn$ với c là hằng số lớn thích hợp và n < 140; giả thiết này với c đủ lớn. Ta cũng chọn một hằng số a sao cho hàm số dưới đây mô tả bởi giới hạn trên O(n) (mô tả khử đệ quy của quá trình thực thi thuật toán) bị chặn trên bởi an với mọi n > 0. Thay thế giả thuyết quy nạp này vào bên phải của kết quả truy toán, ta được:

$$\begin{aligned} T(n) &\leq c\lceil n/5\rceil + c(7n/10+6) + an \\ &\leq cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an)\,, \end{aligned}$$

đạt tối đa là cn nếu

$$-cn/10 + 7c + an \leq 0\,. \tag{9.2}$$

Bất đẳng thức (9.2) tương đương với bất đẳng thức $c \geq 10a(n/(n-70))$ với n > 70. Bởi ta giả sử $n \geq 140$, ta có $n(n-70) \leq 2$ và vì vậy

chọn $c \geq 20a$ sẽ thỏa mãn bất đẳng thức (9.2). Thời gian thực hiện thuật toán SELECT trong trường hợp xấu nhất là tuyến tính.

Giống như sắp xếp so sánh (xem mục 8.1), CHỌN và chọn – ngẫu nhiên xác định thông tin về thứ tự tương đối của các thành phần bằng cách so sánh các thành phần. Nhớ lại ở chương 8 là sắp xếp đòi hỏi thời gian thực hiện $\Omega(nlgn)$, kể cả trung bình (xem vấn đề 8-1). Thuật toán sắp xếp với thời gian tuyến tính trong chương 8 làm giả định đầu vào. Mặt khác, thuật toán chọn với thời gian tuyến tính trong chương này không cần thiết bất kì giả định nào về đầu vào. Chúng không phụ thuộc vào giới hạn dưới $\Omega(nlgn)$ bởi chúng tự giải quyết vấn đề chọn mà không cần sắp xếp. Do đó, việc giải bài toán chọn bằng cách sắp xếp và chỉ số, như đã trình bày ở trên phân giới thiệu của chương này gần như không hiệu quả.

Ghi chú của chương

Thuật toán tìm trung vị trong thời gian tuyến tính với trường hợp xấu nhất được đưa ra bởi Blum, Floyd, Pratt, Rivest, và Tarjan. Phiên bản ngẫu nhiên nhanh là của Hoare. Floyd và Rivest đã phát triển thêm phiên bản ngẫu nhiên cải tiến mà phân hoạch xung quanh phần tử đệ quy từ một phần tử ví dụ mẫu.

Vẫn chưa biết được cần bao nhiêu phép so sánh để xác định trung vị. Bet và John đưa ra kết luận cận dưới là 2n phép so sánh cho việc tìm trung vị, và Sch¨onhage, Paterson, và Pippenger đưa ra kết luận cận trên là 3n. Dor và Zwick đã chứng minh 2 cận đó. Cận trên của họ đưa ra sấp xỉ nhỏ hơn 2.95n, và cận dưới là (2 + ϵ)n, với ϵ là hằng số dương nhỏ. Paterson mô tả các kết quả cùng với các thực hiện liên quan.