NGUYỄN HỮU ĐIỂN

THUẬT TOÁN VÀ LẬP TRÌNH

QUYỂN 4
TÌM KIẾM VÀ THUẬT TOÁN

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI

NGUYỄN HỮU ĐIỂN

THUẬT TOÁN VÀ LẬP TRÌNH

QUYỂN 4
TÌM KIẾM VÀ THUẬT TOÁN

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI

LỜI NÓI ĐẦU

Những năm trước khi lập trình VieTeX tôi toàn dùng C/C++ thu thập tài liệu nhiều nhưng không có thời gian để viết lại. Nay muốn viết lại thì sức khỏe không ổn định. Tôi đã cố gắng gom lại thành các tập lập trình theo chủ đề. Nội dung mỗi thuật toán bắt đầu từ lý thuyết đến lập trình bằng C/C++.

Cuốn sách viết ra không dành riêng cho các bạn học tin học, mà các bạ học toán, thầy cô giáo, các bạn thích tìm hiểu về thuật toán. Cũng như tôi bắt đầu có biết gì về lập trình đâu, tự học và chăm chỉ là thành công thôi. Tôi dùng trình biên dịch Dev-C++: https://www.bloodshed.net/
Hiện nay Dev-C++ cải tiến rất nhiều và chạy tốt với môi trường unicode. Những ví dụ trong tài liệu các bạn chép thẳng vào soạn thảo và biên dịch không cần cấu hình trình biên dịch.

Tôi đã làm các quyển sách:

1. Thuật toán và số học.
2. Thuật toán và dữ liệu.
3. Thuật toán xắp xếp
4. Thuật toán tìm kiếm
5. Thuật toán đồ thị,
6. Thuật toán quay lui
7. Thuật toán chia để trị
8. Thuật toán động
9. Thuật toán tham
10. Thuật toán nén
11. Một số đề thi Olympic Tin học.

Cuốn sách dành cho học sinh phổ thông yêu toán, học sinh khá giỏi môn toán, các thầy cô giáo, sinh viên đại học ngành toán, ngành tin học và những người yêu thích Toán - Tin. Trong biên soạn không thể tránh khỏi sai sót và nhầm lẫn mong bạn đọc cho ý kiến.

Hà Nội, ngày 25 tháng 2 năm 2022

Nguyễn Hữu Điển

NHỮNG KÝ HIỆU

Trong cuốn sách này ta dùng những kí hiệu với các ý nghĩa xác định trong bảng dưới đây:

$\mathbb{N}$	tập hợp số tự nhiên
$\mathbb{N}^*$	tập hợp số tự nhiên khác 0
$\mathbb{Z}$	tập hợp số nguyên
$\mathbb{Q}$	tập hợp số hữu tỉ
$\mathbb{R}$	tập hợp số thực
$\mathbb{C}$	tập hợp số phức
$\equiv$	dấu đồng dư
∞	dương vô cùng (tương đương với $+\infty$)
$-\infty$	âm vô cùng
$\emptyset$	tập hợp rỗng
C_m^k	tổ hợp chập k của m phần tử
$\vdots$	phép chia hết
$\not\vdots$	không chia hết
$UCLN$	ước số chung lớn nhất
$BCNN$	bội số chung nhỏ nhất
deg	bậc của đa thức
IMO	International Mathematics Olympiad
APMO	Asian Pacific Mathematics Olympiad

NỘI DUNG

DANH SÁCH CÁC HÌNH

DANH SÁCH CÁC BẢNG

Danh sách chương trình

CHƯƠNG 4
TÌM KIẾM VÀ THUẬT TOÁN

4.1. Giới thiệu

Quá trình sắp xếp được thảo luận trong Chương 3 có liên quan chặt chẽ đến quá trình tìm kiếm. Thật vậy, việc sắp xếp một phần hoặc toàn bộ một tập hợp các mục có thể đẩy nhanh quá trình tìm kiếm. Tất nhiên, hai hoạt động cũng có thể được coi là hoàn toàn độc lập. Ví dụ, tên của nhân viên trong một công ty có thể được sắp xếp theo thứ tự bảng chữ cái, chỉ để có một danh sách được in ra giấy. Đồng thời, hầu hết các thuật toán tìm kiếm không giả định hoặc yêu cầu sắp xếp trước.

Tìm kiếm là một hoạt động cơ bản được thực hiện hàng ngày bởi mỗi chúng ta vào nhiều dịp khác nhau. Cho dù chúng ta có nhận ra hay không, thì cứ như thể chúng ta dành một khoảng thời gian đáng kể để tìm kiếm: tìm số điện thoại, tìm chìa khóa bị mất, tìm thứ gì đó để ăn trong tủ lạnh, tìm Zamunda trên bản đồ Châu Phi, tìm kiếm làm việc, tìm kiếm một chương trình truyền hình thú vị, chúng ta đang tìm kiếm vấn đề mới của "Lao động vàng", chúng ta đang tìm kiếm sự cố, v.v. Có thể nói rằng nhu cầu là một hoạt động nổi tiếng đối với tất cả chúng ta. Tuy nhiên, chỉ có một số người đã nghĩ về

các phương pháp được sử dụng. Đồng thời, sự đa dạng của các loại hình tìm kiếm gần như là vô hạn nên rất khó phân loại chúng. Trong quá trình tìm kiếm, hầu hết chúng ta tiến hành chủ yếu từ kinh nghiệm tích lũy được, và trong một tình huống không quen thuộc, họ xử lý theo phương pháp thử và sai. Có những phương pháp tìm kiếm đủ mạnh có thể khác biệt đáng kể với nhau tùy thuộc vào các điều kiện cụ thể. Ví dụ: tìm kiếm có thể được đặt không chính xác (tìm kiếm một con bê dưới một con bò), mục tiêu có thể được xác định mơ hồ (tìm kiếm một chương trình truyền hình thú vị), mục tiêu có thể di chuyển (tìm kiếm tầu ngầm của đối phương), tìm kiếm nó có thể bị giới hạn về thời gian (tìm bom bằng cơ chế đồng hồ), các mẫu có thể có giá khác (mò kim đáy bể), tập hợp có thể là vô tận (tìm kiếm dầu) và các loại khác. Đặc biệt, tìm kiếm có thể có tất cả các thuộc tính được liệt kê ở trên, cũng như có thể có một số thuộc tính khác (tìm kiếm ý nghĩa cuộc sống).

Rõ ràng là hầu như không thể bao gồm tất cả các trường hợp được liệt kê ở trên, đó là lý do tại sao một số trong số chúng cần được sửa chữa. Dưới đây, chúng ta sẽ làm việc với một mô hình đơn giản và được xác định rõ với các thuộc tính sau:

- mục tiêu cố định và được xác định rõ ràng;
- thời gian là không giới hạn;
- tập mà chúng ta tìm kiếm là hữu hạn và tĩnh, tức là nó không thay đổi trong quá trình tìm kiếm;
- tất cả các mẫu (so sánh) có cùng một mức giá;
- chúng ta không bao giờ mắc lỗi trong bài kiểm tra;
- tất cả thông tin từ các so sánh trước đó được giữ lại.

Tìm kiếm hiếm khi là một hoạt động cô lập. Mô hình trên rất hữu ích, nhưng không may là không phải lúc nào cũng có sẵn. Thường xảy ra rằng tập hợp mà bạn đang xem thay đổi động: các phần tử được bao gồm và loại trừ, sắp xếp lại, v.v. Do đó, sẽ rất hữu ích nếu coi các thuật toán tìm kiếm không phải là một cái gì đó riêng biệt và độc lập, mà là một phần của một gói hoàn chỉnh các phép toán cơ bản trên các phần tử của một tập hợp. Bằng cách này, khi xem xét tất cả các hoạt động cùng nhau, chúng ta sẽ có thể đánh giá tốt

hơn sự tương tác của chúng và tránh các ước tính không chính xác về hiệu suất tìm kiếm.

Ví dụ, tìm kiếm nhị phân, sẽ được thảo luận dưới đây, có độ phức tạp là $\Theta(\log_2 n)$ trong cả trường hợp trung bình và trường hợp xấu nhất. Tuy nhiên, đồng thời, nó yêu cầu các phần tử của tập hợp phải được sắp xếp. Nếu không đúng như vậy, việc sắp xếp trước chúng trước mỗi lần tìm kiếm sẽ yêu cầu thời gian theo thứ tự là $\Theta(n.\log_2 n)$. Do đó, độ phức tạp thực tế của nó sẽ là $\Theta(n.\log_2 n)$. Tất nhiên, chúng ta có thể giữ cho đám đông luôn được sắp xếp. Tuy nhiên, điều này sẽ làm phức tạp đáng kể các hoạt động bao gồm và loại trừ các phần tử. Thật tiện lợi khi coi tìm kiếm như một phần tử của tập hợp các thao tác sau:

- khởi tạo
- tìm kiếm
- chèn
- xóa
- thống nhất các bộ
- sắp xếp

Có một kết nối trực tiếp giữa một số hoạt động trên và chúng thường được thực hiện đồng thời. Ví dụ, trước khi chèn một mục, bạn thường tìm kiếm vị trí tương ứng. Như với việc sắp xếp (và các cấu trúc dữ liệu được thảo luận trong Chương 2), chúng ta sẽ giả định rằng các phần tử của tập hợp được đề cập là các bản ghi kiểu struct CElem, một trong các trường được chọn làm khóa. Đôi khi các trường khóa là duy nhất trong các phần tử của tập hợp, những trường hợp khác lặp lại chúng có thể chấp nhận được. Dưới đây chúng ta sẽ thấy rằng khả năng lặp lại hoặc không lặp lại các phần tử của tập hợp hóa ra lại là một yếu tố quan trọng của việc thực hiện tập hợp các phép toán trên. Giả sử rằng đối với các hoạt động trên, chúng ta đã triển khai các thuật toán thích hợp để ngăn chặn sự trùng lặp của các trường khóa một cách rõ ràng. Chúng ta có thể dễ dàng xử lý mã chương trình liên quan để cho phép sao chép không? Một cách tiếp cận khả thi để giải quyết vấn đề là thêm một con trỏ bổ sung vào bản ghi đại diện cho các phần tử của tập

hợp. Con trỏ này sẽ trỏ đến danh sách các mục có cùng khóa. Ưu điểm chính của phương pháp này là chỉ cần một lần tìm kiếm là đủ để tìm tất cả các khóa có giá trị cho trước. Một cách triển khai khả thi khác là cho phép sao chép các phần tử và khi tìm kiếm, bất kỳ phần tử nào cũng được trả về với khóa được chỉ định. Vấn đề chính ở đây sẽ là tìm kiếm tất cả các phần tử với khóa này, mà nếu cần, một cơ chế đặc biệt nên được cung cấp. Tất nhiên, chúng ta có thể giới thiệu một khóa kinh doanh thứ hai, mà chúng ta có thể đảm bảo là duy nhất cho toàn bộ. Sau đó, chúng ta chỉ có thể tìm kiếm khóa đầu tiên và cả hai khóa cùng một lúc. Có lẽ tùy chọn linh hoạt nhất là triển khai một hàm so sánh đặc biệt mà thủ tục tìm kiếm có thể tham khảo. Bằng cách này, trong quá trình tìm kiếm sẽ có thể kiểm tra xem mục hiện đang được xem xét với khóa được tìm kiếm có đáp ứng bất kỳ bộ điều kiện bổ sung nào hay không.

4.2. Tìm kiếm liên tiếp

Thuật toán tìm kiếm cơ bản nhất và rõ ràng nhất là tìm kiếm nhất quán. Giả sử rằng các phần tử của tập hợp được chứa trong mảng một chiều. Việc tìm kiếm được thực hiện bằng cách tìm kiếm tuần tự các phần tử của mảng cho đến khi tìm thấy phần tử cần tìm hoặc cho đến khi tìm thấy tất cả các phần tử. Điều sau có nghĩa là một phần tử có khóa như vậy không tồn tại. Khi một phần tử mới đến, chúng ta sẽ chèn nó vào cuối mảng seqSearch(). Sau đây là cách triển khai có thể có của các chức năng cơ bản dựa trên tìm kiếm nhất quán:

Chương trình 4.1. Tìm kiếm liên tiếp (401seq-arr.c)

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
#define DataType int

struct CElem {
  int key;
  DataType data;
  /* ... */
```

```
} m[MAX + 1]; /* Một mảng các bản ghi */
unsigned n; /*  Số phần tử trong mảng*/

void seqInit(void) {  n = 0;  }  /*  Khởi tạo*/
unsigned seqSearch(int key) /*Tìm kiếm liên tiếp */
{  unsigned x;
   m[0].key = key;  /*Giới hạn*/
   for (x  = n + 1;  key != m[--x].key;  )  ;
   return x;
}

void seqInsert(int key,  DataType data) /*Thêm một phần tử mới*/
{  m[++n].key = key;
   m[n].data = data;
}

void seqPrint(void) /*Hiển thị  danh sách trên màn hình*/
{  unsigned i;
   char buf[9];
   for (i  = 1;  i  <= n;  i++) {
      sprintf (buf,  "%d | %d", m[i].key, m[i].data);
      printf("%8s", buf);
   }
}

void performSearchTest(void)
{  unsigned ind, elem2Search;
   for (elem2Search = 0; elem2Search < 2*MAX; elem2Search++) {
   printf("Đang tìm kiếm một phần tử có khóa %u.\n", elem2Search);
   if (0  == (ind = seqSearch(elem2Search)))
      printf("%s","Một phần tử có khóa như vậy không tồn tại!\n");
   else
      printf("%Đã tìm thấy phần tử! phần thông tin: %d\n", m[ind].
      data);
   }
}

int main() {
   unsigned ind;
   seqInit();
```

```
    for (ind = 0; ind < MAX; ind++)
      seqInsert(rand() % (MAX*2), ind);
    printf("Danh sách bao gồm các mục sau: \n");
    seqPrint();
    printf("\n Thử nghiệm:\n");
    performSearchTest();
    return 0;
}
```

Như với việc sắp xếp, chúng ta đã giả định ở trên rằng phần thông tin thực tế của các phần tử mảng chỉ chứa trường dữ liệu. Tất nhiên, nếu cần, người đọc có thể dễ dàng sửa đổi cấu trúc của loại `struct` CElem theo nhu cầu và yêu cầu cụ thể của họ, điều này sẽ yêu cầu những thay đổi tối thiểu tương ứng trong đoạn mã trên.

Việc tìm kiếm được thực hiện tuần tự theo hướng giảm dần các chỉ số. Lưu ý rằng phần tử rỗng của mảng không chứa phần tử của tập hợp mà là thông tin dịch vụ. Nó được sử dụng như một phần tử giới hạn cho phép đơn giản hóa chu trình: chỉ một điều kiện được kiểm tra, vì chúng ta đã đảm bảo tìm được phần tử có khóa được chỉ định (trong trường hợp cực đoan là số 0). Trong trường hợp có khóa trùng lặp, thuật toán trên hiển nhiên trả về phần tử nhận được cuối cùng với khóa này.

Độ phức tạp của thuật toán ở giữa và trong trường hợp xấu nhất tương ứng là bao nhiêu? Rõ ràng rằng trường hợp xấu nhất là khi thiếu vật phẩm cần thiết. Sau đó, tất cả $n + 1$ phần tử của mảng được xem, bao gồm cả phần tử rỗng. Trong trường hợp chung, khi tìm kiếm thành công, số lần so sánh trung bình là $(n + 1)/2$ (Tại sao?). Đồng thời, cần quét toàn bộ mảng để tìm tất cả các phần tử có khóa cho trước.

Bài tập

▷ **4.1.** Chứng minh rằng số phép so sánh trung bình trong một tìm kiếm tuần tự là $(n + 1)/2$.

4.2.1. Tìm kiếm liên tiếp trong danh sách đã sắp xếp

Tìm kiếm tuần tự, đã thảo luận ở trên, là thuật toán tìm kiếm đơn giản nhất và không hiệu quả nhất. Chúng ta có thể cố gắng cải thiện hiệu quả của nó bằng cách sắp xếp các mục. Điều này sẽ tương đối dễ dàng nếu chúng ta sử dụng danh sách liên kết thay vì một mảng. Tất cả những gì chúng ta phải làm là chèn từng mục mới vào đúng vị trí của nó để danh sách vẫn được sắp xếp. Do đó, trước mỗi lần chèn sẽ phải thực hiện một tìm kiếm tương ứng để tìm ra vị trí tương ứng.

Thao tác chèn phức tạp đáng kể. Chúng ta đã làm điều đó ở trên trong một thời gian liên tục, không có bất kỳ so sánh nào và bây giờ nó sẽ yêu cầu tìm kiếm sơ bộ. Rốt cuộc chúng ta đã đạt được những gì? Thực tế là các mục được sắp xếp theo thứ tự tăng dần sẽ cho phép chúng ta ngừng tìm kiếm thêm khi đến một mục có khóa lớn hơn tìm kiếm. Số lượng phép so sánh cần thiết trong trường hợp tìm kiếm không thành công giảm xuống $(n+1)/2$, so với $n+1$ trong trường hợp không được sắp xếp. Trên thực tế, các tìm kiếm thành công và không thành công trở nên ngang nhau, vì việc sửa đổi này rõ ràng không ảnh hưởng đến tìm kiếm thành công. Trong trường hợp xấu nhất, tất cả các yếu tố phải được xem xét lại, tức là $n+1$ phép so sánh phải được thực hiện. Hóa ra là để đánh đồng độ phức tạp của việc chèn, mà trước đây là không đổi, với độ phức tạp của tìm kiếm, chúng ta chỉ có được sự bình đẳng của tìm kiếm thành công và không thành công. Kết quả thu được là vô cùng hướng dẫn và cho thấy rằng trước khi tiến hành bất kỳ tối ưu hóa "hiển nhiên" nào, người ta nên cân nhắc rất kỹ những lợi ích và tác hại có thể có mà nó sẽ mang lại.

Sau đây là một ví dụ về triển khai tìm kiếm tuần tự trong danh sách được sắp xếp, cùng với phiên bản tương ứng của các hàm chèn và khởi tạo:

Chương trình 4.2. Tìm kiếm liên tiếp (402seq-list.c)

```
struct CElem {
  int key;
  DataType data;
```

```
  struct CElem *next;
  /* ... */
} *head;

void listInit (void) /* Khởi tạo */
{ head = (struct CElem *) malloc(sizeof *head);
  head->next = NULL;
}

void listInsert (int key, DataType data) /* Thêm một phần tử mới */
{ struct CElem *p, *q, *r;
  q = (struct CElem *) malloc(sizeof *head),
  r = (p = head)->next;
  while (r != NULL && r->key < key) {
    p = r;
    r = r->next;
  }
  q->key = key;
  q->data = data;
  q->next = r;
  p->next = q;
}

struct CElem *listSearch(int key) /* Tìm kiếm liên tiếp*/
{ struct CElem *q;
  for (q = head->next; q != NULL && q->key < key; q = q->next) ;
  if (NULL == q || key != q->key)
    return NULL;
  else
    return q;
 }

void listPrint (void) /*Hiển thị danh sách trên màn hình*/
{ struct CElem *q;
  char buf[9];
  for (q = head->next; q != NULL; q = q->next) {
    sprintf (buf, "%d|%d", q->key, q->data);
    printf("%8s", buf);
  }
}
```

Bài tập

▷ **4.2.** Xác định số lần so sánh trung bình khi tìm kiếm trong danh sách đã sắp xếp. So sánh với tìm kiếm nhất quán cổ điển.

4.2.2. Tìm kiếm liên tục với sự sắp xếp lại

Trong trường hợp chúng ta có thông tin sơ bộ về xác suất truy cập vào từng mục, chúng ta có thể sắp xếp chúng sao cho mục được tìm kiếm thường xuyên nhất ở đầu danh sách, tiếp theo - ngay sau nó, v.v. Sử dụng chiến lược như vậy chứng tỏ cực kỳ hiệu quả, đặc biệt là với phân bố rất không đồng đều, trong đó một số lượng nhỏ các phần tử được tìm kiếm với xác suất rất cao.

Trong trường hợp chúng ta không có thông tin sơ bộ như vậy, chúng ta có thể tự lấy thông tin đó với sự trợ giúp của các quan sát thống kê đơn giản: chỉ cần gắn một máy đo truy cập cho mỗi phần tử là đủ. Mỗi lần chúng ta tìm kiếm, chúng ta cập nhật bộ đếm tìm kiếm. Rõ ràng là sau khi cập nhật, anh ấy cuối cùng có thể di chuyển lên danh sách. Do đó, nên so sánh với mục trước đó trong danh sách, hai mục này có thể được trao đổi. Do có thể có nhiều tần số truy cập trùng lặp trong danh sách nên trong trường hợp trao đổi cần phải so sánh với phần tử tiếp theo trước đó, v.v ... cho đến khi đạt được vị trí chính xác tương ứng với tần số của phần tử mới. Việc sắp xếp lại diễn ra đúng lúc, tương ứng với n, và trong trường hợp xấu nhất có thể xảy ra trường hợp phần tử cuối cùng của danh sách trở thành phần tử đầu tiên, trao đổi với mọi người trước nó [Wirth-1980].

Tuy nhiên, kế hoạch được đề xuất với việc duy trì các bộ đếm đặc biệt tỏ ra cồng kềnh, yêu cầu bộ nhớ bổ sung về thứ tự của n và có thể được tối ưu hóa. Một lần nữa, nó chỉ ra rằng đơn giản là tốt hơn. Thật vậy, trong trường hợp này, một chiến lược tổ chức lại đơn giản hơn đã tỏ ra cực kỳ hiệu quả. Không có quầy nào được duy trì và không có số liệu thống kê nào được lưu giữ. Thay vào đó, mỗi lần tìm kiếm thành công một mục sẽ đưa nó lên đầu danh sách.

Tất nhiên, chiến lược này không đảm bảo cho chúng ta sự sắp xếp tối ưu của các phần tử theo tần suất truy cập của chúng, nhưng nó dễ bảo trì và đủ hiệu quả. Thật vậy, không giống như tùy chọn

bộ đếm, ở đây việc tổ chức lại diễn ra theo thời gian. Mặt khác, một mục càng được truy cập thường xuyên thì càng có nhiều khả năng nằm trong số các mục đầu tiên trong danh sách. Đồng thời, các đặc thù của địa phương cũng được tính đến tốt hơn: nếu một mục hiếm khi được tìm kiếm trên toàn cầu thường xuyên được tìm kiếm tại một thời điểm nhất định, thì chiến lược này sẽ cho phép chúng ta tận dụng lợi thế của nó. (So sánh với các phương pháp nén thích ứng của **??**)

Chương trình 4.3. Tìm kiếm liên tiếp với sự sắp xếp lại (403reorder.c)

```
void listInsert (int key, DataType data) /* Thêm một mặt hàng mới*/
{ struct CElem *q = (struct CElem *) malloc(sizeof *head);
  q->key = key;
  q->data = data;
  q->next = head;
  head = q;
}

/*Tìm kiếm liên tục với sự sắp xếp lại*/
struct CElem *listSearch(int key)
{ struct CElem *q, *p = head;
  if (NULL == head)
  return NULL;
  if (head->key == key) return head;
    for (q = head->next; q != NULL; )
      if (q->key != key) {
        p = q;
        q = q->next;
    }
    else {
        p->next = q->next;
        q->next = head;
        return (head = q);
    }
  return NULL;
}
```

Bài tập

▷ **4.3.** Thực hiện tìm kiếm danh sách sắp xếp lại dựa trên bộ đếm truy cập. So sánh hiệu suất của nó với chuyển đổi không có bộ đếm ở trên.

▷ **4.4.** So sánh ý tưởng cơ bản của việc tìm kiếm liên tiếp có sắp xếp lại không có bộ đếm với ý tưởng của các phương pháp nén thích ứng (xem **??**).

▷ **4.5.** Hãy xác định số lượng so sánh trung bình trong một tìm kiếm nhất quán với sự sắp xếp lại. So sánh với tìm kiếm nhất quán cổ điển.

▷ **4.6.** Để so sánh trong trường hợp tốt nhất, trung bình và xấu nhất, ba phương pháp được xem xét để tìm kiếm liên tiếp:

a) trong một mảng không có thứ tự (xem 4.2)

(b) trong danh sách đã sắp xếp (xem 4.2.1)

(c) trong danh sách sắp xếp lại chưa được phân loại (xem 4.2.2)

4.2.

4.3. Tìm kiếm theo từng bước. Tìm kiếm bậc hai

Chúng ta hãy nhìn lại trường hợp của một tập hợp được đặt hàng. Phương pháp tìm kiếm theo danh sách được sắp xếp được thảo luận ở trên hơi khác so với tìm kiếm tuần tự tiêu chuẩn và không sử dụng đủ thứ tự mục. Chúng ta sẽ cố gắng sửa chữa sai lầm này bằng cách thực hiện một cách tiếp cận hoàn toàn khác. Hãy chọn một số bước k và liên tiếp kiểm tra xem khóa của phần tử được tìm kiếm có lớn hơn phần tử đầu tiên hay không, từ phần tử thứ $(k+1)$, từ phần tử thứ $(2k+1)$, từ phần tử $(3k+1)$ phần tử phần tử thứ, ... Tức là chúng ta so sánh nó với m [1] .key, m [k + 1] .key, m [2k + 1] .key, ... Quá trình kết thúc khi đến phần tử lớn hơn hoặc bằng x, hoặc ở cuối mảng.

Chúng ta hãy xem xét kỹ hơn sơ đồ trên (chúng ta sẽ gọi nó là *tìm kiếm theo bước*). Giả sử rằng bằng cách áp dụng nó, chúng ta đã đạt được một phần tử lớn hơn x. Bây giờ chúng ta có thể sử dụng tìm kiếm tuần tự trong khoảng thời gian được xác định bởi hai mẫu cuối cùng. Trong trường hợp chúng ta ở ngoài mảng, chúng ta có thể sử dụng tìm kiếm tuần tự từ mẫu trước đó đến cuối mảng. Rõ ràng, cách tiếp cận như vậy có thể dẫn đến việc giảm mạnh số lượng mục được xem bởi tìm kiếm nhất quán. Ngoài ra, có thể dễ dàng nhận thấy rằng phương pháp được trình bày là một bản tóm tắt của tìm kiếm tuyến tính. Thật vậy, giá trị thứ hai thu được tại $k = 1$.

Theo mô tả ở trên, phương pháp đề xuất luôn bắt đầu bằng m [1] .key. Khởi đầu như vậy như thế nào là hợp lý? Không khó để thấy rằng đây thực sự là một lựa chọn tồi, và vì những lý do tương tự mà lựa chọn m [n] .key là tệ: nó mang lại cho chúng ta một lượng thông tin tối thiểu. Nó chỉ ra rằng nó là rất thích hợp để bắt đầu trực tiếp với m [k] .key. (Tại sao?) Dễ dàng thấy rằng trong trường hợp này, tìm kiếm tuyến tính cũng thu được $k = 1$.

Hiệu quả của phương pháp được mô tả tại một k cố định là gì và trường hợp xấu nhất là gì? Rõ ràng là tìm kiếm tuyến tính có cùng giá cho tất cả các khoảng của kiểu [m [i * k + 1] .key; m [(i + 1) * k] .key], vì nó được thực hiện trên cùng một số phần tử. Ngoại lệ duy nhất có thể là khoảng cuối cùng, có thể chứa ít hơn k phần tử. Trong trường hợp xấu nhất, khóa tìm kiếm nằm trong khoảng cuối cùng, có nghĩa là chúng ta sẽ cần $[n/k]$ so sánh để xác định khoảng mà chúng ta cần áp dụng tìm kiếm tuyến tính. Với những điều này, chúng ta nên thêm độ dài của khoảng, với n, bội số của k, là $k - 1$. Chúng ta nhận được điều đó trong trường hợp xấu nhất khi tìm kiếm với bước k không quá $[n/k] + k - 1$ phép so sánh được thực hiện.

Sau đây là một ví dụ về triển khai thuật toán được mô tả:

Chương trình 4.4. Tìm kiếm theo bước(404jumpsear.c)

```
unsigned seqSearch(unsigned l, unsigned r, int key)
{ while (l <= r)
  if (m[l++].key == key)
```

```
    return l-1;
  return NOT_FOUND;
}

unsigned jmpSearch(int key, unsigned step)
{ unsigned ind;
  for (ind = 0; ind < n && m[ind].key < key; ind += step);
  return seqSearch(ind + 1 < step ? 0 : ind + 1 - step,
                   n < ind ? n : ind, key);
}
```

Câu hỏi quan trọng được đặt ra: Làm thế nào, cho trước n, để chọn k sao cho đảm bảo hiệu quả tối đa? Bảng 4.1 hiển thị số lượng phép so sánh tối đa được thực hiện bằng cách chèn từng bước tại các giá trị khác nhau của n và k:

$n\backslash k$	1	2	3	4	5	6	7	8
1	1							
2	2	2						
3	3	2	3					
4	4	3	3	4				
5	5	3	3	4	5			
6	6	4	4	4	5	6		
7	7	4	4	4	5	6	7	
8	8	5	4	5	5	6	7	8

Bảng 4.1. Số phép so sánh tối đa ở các giá trị khác nhau của n và k.

Có thể thấy rằng các giá trị tốt nhất của k gần với $n/2$, tức là chúng nằm ở giữa hàng tương ứng của bảng hoặc ngay bên trái của nó. Để xác định chính xác hơn giá trị nào của k là tốt nhất và chúng phụ thuộc như thế nào vào n, chúng ta nên xác định sự phụ thuộc nào giữa n và k thì hàm $f(k)$ nhận giá trị nhỏ nhất:

$$f(k) = [n/k] + k - 1.$$

Không khó để chỉ ra (ví dụ, bằng cách nghiên cứu dấu của đạo hàm cấp hai) rằng đây là giá trị của $k = \sqrt{n}$. Khi đó $f(k) \leq 2\sqrt{n} + 1$. Trong trường hợp này phép tìm được gọi là *bậc hai*.

Vì vậy, chúng ta đã thực hiện một cải tiến thực sự tốt: từ n lên $\sqrt{n}$. Chúng ta có thể đạt được nhiều hơn không? Đúng như dự đoán, câu trả lời cho câu hỏi này là có. Thật vậy, chúng ta đã tối thiểu hóa $f(k)$ ở trên, giả sử rằng sau khi xác định khoảng thời gian, chúng ta sẽ thực hiện tìm kiếm tuần tự. Và đây chính xác là nơi xuất phát ý tưởng: Và điều gì sẽ xảy ra nếu tại thời điểm này chúng ta áp dụng một tìm kiếm khác với một số bước mới l ($1 < l < k$) và chỉ sau đó là một tìm kiếm tuần tự? Thật vậy, độ dài của khoảng sau lần tìm kiếm bước đầu tiên k là $\sqrt{n}$, có thể là một số đủ lớn để một sự cải thiện như vậy là hợp lý. Bây giờ trong bước đầu tiên, chúng ta sẽ có không quá k phép so sánh để xác định khoảng đầu tiên, sau đó không quá l - để xác định khoảng thứ hai và cuối cùng là không quá $n/(k.l)$ - cho tìm kiếm tuần tự. Chúng ta có thể hình thành một hàm mới để tối thiểu hóa, trong trường hợp đó, dễ dàng hóa ra rằng giá trị nhỏ nhất đạt được đối với $k = l = n/(k.l)$, trong đó $n = k^3$. Trong trường hợp này, thuật toán sẽ thực hiện không quá $3\sqrt[3]{n}$ phép so sánh. Đối với $n \geq 12$, nó chỉ ra rằng $3\sqrt[3]{n} < 2\sqrt{n}$, tức là chúng ta đã đạt được sự cải thiện [Gregory, Rawlins-1997].

Và điều gì sẽ xảy ra nếu chúng ta áp dụng thuật toán lần thứ ba, thứ tư, v.v.? Lập luận tương tự như trên dẫn chúng ta đến giới hạn của độ phức tạp lôgarit. Mặc dù tầm quan trọng của bản tóm tắt được mô tả (chúng ta đã đạt được độ phức tạp lôgarit!) Chúng ta sẽ không tập trung vào nó nữa, vì có một cách đơn giản hơn để đạt được độ phức tạp như vậy.

Bài tập

▷ **4.7.** Hãy thực hiện tìm kiếm hai giai đoạn với các bước k và l. Thực nghiệm tìm các giá trị tối ưu của k và l dưới dạng các hàm của n.

▷ **4.8.** Chứng minh rằng bước tốt nhất trong tìm kiếm bậc hai trong một mảng có n phần tử là n.

4.4. Tìm kiếm nhị phân

Trong trường hợp một tập hợp có số lượng mục lớn và số lượng tìm kiếm lớn, sẽ thuận tiện khi sử dụng tìm kiếm nhị phân sau khi đã phân loại trước các mục. Tìm kiếm nhị phân là một ví dụ điển hình về việc áp dụng nguyên tắc chia và quy tắc La Mã, sẽ được thảo luận chi tiết trong Chương **??**. Ý tưởng chính là chia bài toán phức tạp thành nhiều bài toán đơn giản hơn, từ đó có thể được chia thành các bài toán đơn giản hơn. và Quá trình tiếp tục cho đến khi đạt được một bài toán đủ đơn giản với một giải pháp tầm thường.

Tình hình chính xác với tìm kiếm nhị phân là gì? Chúng ta giả sử chúng ta có một mảng được sắp xếp. Ý tưởng là chia mảng thành hai mảng con và xác định xem phần tử được yêu cầu chắc chắn không thể nằm trong đó. Mảng phụ được đề cập không được xem xét thêm và các nỗ lực tiếp theo sẽ được tập trung vào mảng có triển vọng hơn. Có thể lập luận rằng tìm kiếm nhị phân không phải là một ví dụ cổ điển về việc áp dụng chiến lược chia để trị, vì nó liên quan đến cả hai mảng con. Tuy nhiên, điều trên có thể thấy rằng nó không cấm việc cắt tỉa sớm các tập hợp con không tăng trưởng. Cách phân chia như thế nào? Hãy biểu thị phần tử tìm kiếm là x và giả sử rằng mảng đã được sắp xếp. Hãy so sánh x với phần tử trung bình của nó (Với số phần tử chẵn có hai phần tử trung bình và chúng ta chọn ai không quan trọng). Trong trường hợp hòa, cuộc tìm kiếm kết thúc thành công. Nếu x nhỏ hơn phần tử ở giữa, chúng ta có thể xác định ngay mảng con vô vọng. Thật vậy, đây là những phần tử ở bên phải của giữa: vì mảng được sắp xếp, chúng đều lớn hơn giá trị trung bình và đến lượt nó, anh ta đã lớn hơn x. Một cách tiếp cận tương tự được thực hiện khi x lớn hơn phần tử ở giữa, khi đó mảng con bên trái bị loại bỏ. Quá trình tương tự sau đó được áp dụng cho mảng con không bị từ chối. Ở mỗi bước, chúng ta chia mảng phối cảnh thành hai mảng con với số phần tử xấp xỉ bằng nhau. Quá trình kết thúc thành công khi tìm thấy phần tử được yêu cầu hoặc đến được mảng trống. Không khó để thấy rằng quá trình này luôn luôn kết thúc. Thật vậy, ở mỗi bước, mảng được đề cập giảm ít nhất một nửa. Ít nhất, vì phần tử giữa của mảng luôn bị từ

chối và điều này với số phần tử lẻ có nghĩa là chúng bị từ chối bởi một phần tử nhiều hơn phần tử còn lại. Nhân tiện, thực tế là ít nhất một phần tử luôn bị từ chối, cụ thể là phần tử ở giữa, là điều cần thiết, bởi vì một trong hai tập hợp (trái hoặc phải) có thể trống. Tuy nhiên, trong trường hợp này cũng vậy, số lượng phần tử trong mảng con được đề cập sẽ giảm mạnh (ít nhất là 1).

Thực hiện trực tiếp các hướng dẫn ở trên, chúng ta nhận được cách triển khai đệ quy rõ ràng sau đây của tìm kiếm nhị phân (được gọi với binSearch (key_who_search, 0, n-1)):

Chương trình 4.5. Tìm kiếm nhị phân(405binsear0.c)

```
unsigned binSearch(int key, int l, int r)
{ int mid;
  if (l > r)
    return NOT_FOUND;
  mid = (l + r) / 2;
if (key < m[mid].key)
  return binSearch(key,l,mid-1);
else if (key > m[mid].key)
  return binSearch(key,mid+1,r);
else { return mid; }
}
```

Quá trình được mô tả ở trên thường là đệ quy. Tuy nhiên, trong thực tế, chính xác một trong hai tập con được xem xét ở mỗi bước. Do đó, không khó để triển khai một giải pháp lặp lại tương ứng:

Chương trình 4.6. Tìm kiếm nhị phân(406binsear1.c)

```
unsigned binSearch(int key)
{ int l = 0, r = n-1, mid;
  while (l <= r) {
    mid = (l + r) / 2;
    if (key < m[mid].key)
      r = mid - 1;
    else if (key > m[mid].key)
      l = mid + 1;
    else
      return mid;
  }
```

```
    return NOT_FOUND;
}
```

Hàm trên sử dụng hai chỉ số l và r, lần lượt chỉ ra ranh giới bên trái và bên phải của khu vực được xem xét. Sự phân chia của từng bước diễn ra ở giữa khu vực giữa. Hàm trả về vị trí của phần tử có khóa value hoặc -1 trong trường hợp không thành công. Vì tại mỗi bước của thuật toán, mảng con được xem xét giảm ít nhất một nửa, thuật toán trên thực hiện không quá $\lceil \log_2 n \rceil + 1$ phép so sánh. Đây là giới hạn trên cho cả tìm kiếm thành công và không thành công [Wirth-1980].

Hãy cố gắng cải thiện phiên bản trên của chương trình. Rõ ràng, ý tưởng cơ bản của tìm kiếm nhị phân "đóng đinh" giới hạn trên của phép so sánh $\lceil \log_2 n \rceil + 1$. Tuy nhiên, vẫn có chỗ để cải thiện. Chúng ta sẽ bắt đầu với một sửa đổi đơn giản về việc thực hiện chương trình được đề xuất và đặc biệt là cách thiết lập ranh giới của khu vực được xem xét. Ở trên, điều này được thực hiện với sự trợ giúp của cặp chỉ số l và r, lần lượt cho biết các đầu bên trái và bên phải của nó. Bây giờ chúng ta sẽ chuyển sang một biểu diễn mới, sử dụng $offset = r - l$ thay vì r. Chúng ta sẽ muốn một sự bù đắp khác luôn là mức độ của hai vợ chồng. Trong trường hợp kích thước của khu vực ban đầu là một mức độ của cặp, thuộc tính này rõ ràng sẽ được duy trì bởi chính nó mà không cần nỗ lực bổ sung từ phía chúng ta. Nếu không, chúng ta sẽ phải bảo mật nó. Cách đơn giản nhất để xử lý vấn đề là trong bước đầu tiên chia mảng thành hai vùng, vùng này phải có kích thước, độ lớn của cặp. Điều này không khó thực hiện, mặc dù nó sẽ vượt qua các khu vực ngay từ bước đầu tiên. Ví dụ: nếu kích thước của mảng là 1000, giải pháp tốt là trước tiên hãy chia nó thành các vùng sau: $(1, 2, ..., 512)$ và $(489, 490, ..., 1000)$.

Cả hai vùng đều có cùng kích thước, cụ thể là 512, là lũy thừa lớn nhất của 2, không vượt quá kích thước của mảng $n = 1000$. Các phần tử $489, 490, \ldots, 512$ thuộc cả hai vùng. Tuy nhiên, ở mỗi bước tiếp theo, kích thước sẽ vẫn là một mức độ của cặp, với hiệu quả tổng thể của thuật toán dự kiến sẽ được cải thiện. Ở đây, phép toán phân chia nặng được thay thế bằng một chút dịch chuyển sang bên

phải, dẫn đến cải thiện hơn nữa. (Trên thực tế, điều đó gây ra một chút tranh cãi với các bộ vi xử lý ngày nay.)

Chương trình 4.7. Tìm kiếm nhị phân(407binsear2.c)

```
unsigned getMaxPower2(unsigned k)
{ unsigned pow2;
  for (pow2 = 1; pow2 <= k; pow2 <<= 1) ;
  return pow2 >> 1;
}

unsigned binSearch(int key)
{ unsigned i, l, ind;
  i = getMaxPower2(n);
  l = m[i].key >= key ? 0 : n - i + 1;
  while (i > 0) {
    i = i >> 1;
    ind = l + i;
    if (m[ind].key == key)
      return ind;
    else if (m[ind].key < key)
      l = ind;
  }
  return NOT_FOUND;
}
```

Chúng ta có thể loại bỏ một so sánh trong chu kỳ. Mặt khác, chúng ta sẽ không còn có thể kết thúc sớm thuật toán khi phần tử được tìm kiếm được tìm thấy trước bước cuối cùng. Do đó, sự thay đổi hóa ra có phần gây tranh cãi:

Chương trình 4.8. Tìm kiếm nhị phân(408binsear3.c)

```
unsigned binSearch(int key)
{ unsigned i, l;
  i = getMaxPower2(n);
  l = m[i].key >= key ? 0 : n - i + 1;
  while (i > 1) {
    i = i >> 1;
    if (m[l+i].key < key)
      l += i;
  }
```

```
  return (l < MAX && m[++l].key == key ? l : NOT_FOUND);
}
```

Chúng ta hãy xem xét kỹ hơn ý tưởng của thuật toán. Không khó để nhận thấy rằng chúng ta thực hiện tùy chọn nào trong số các tùy chọn trên, đối với mỗi giá trị của kích thước n của miền, thứ tự của các phép so sánh liên tiếp cho mỗi bước được xác định trước. Sau đó, chúng ta có thể viết chúng một cách rõ ràng, loại bỏ nhu cầu phân chia và một chu trình nói chung. Chương trình kết quả hóa ra cực kỳ hiệu quả. Bentley đã báo cáo sự cải tiến khoảng 4,5 lần so với phiên bản cổ điển ban đầu (xem [Bentley-1986]). Tất nhiên, cần lưu ý hạn chế rõ ràng là để vẽ vòng tuần hoàn trong các công trình có điều kiện liên tiếp, kích thước của khu vực ít nhất phải được biết gần đúng. Ngoài ra, sự xuất hiện của chương trình có thể giết chết bất kỳ sự nhiệt tình nào:

Chương trình 4.9. Tìm kiếm nhị phân(409binsear4.c)

```
unsigned binSearch(int key)
{ unsigned l = 0;
  if (m[512].key < key) l  = 1000-512+1;
  if (m[l+256].key < key) l  += 256;
  if (m[l+128].key < key) l  += 128;
  if (m[l+ 64].key < key) l  += 64;
  if (m[l+ 32].key < key) l  += 32;
  if (m[l+ 16].key < key) l  += 16;
  if (m[l+ 8].key < key) l  += 8;
  if (m[l+ 4].key < key) l  += 4;
  if (m[l+ 2].key < key) l  += 2;
  if (m[l+ 1].key < key) l  += 1;
  return (l < 1000 && m[++l].key == key ? l : NOT_FOUND);
}
```

Mặc dù tìm kiếm nhị phân cực kỳ hiệu quả, nó nên được sử dụng một cách thận trọng. Nhược điểm chính của nó là nó nhất thiết phải yêu cầu quyền truy cập trực tiếp vào các phần tử của tập hợp, điều này hạn chế chúng ta sử dụng một mảng và không cho phép bất kỳ cấu trúc động nào. Đến lượt mình, yêu cầu điều chỉnh các phần tử của tập hợp lại tạo ra các vấn đề lớn hơn cho việc chèn. Thật vậy,

nếu chúng ta chèn các phần tử mới vào cuối mảng, chúng ta sẽ phải sắp xếp nó trước mỗi lần tìm kiếm, lần chèn tiếp theo, điều này rõ ràng là vô cùng bất hợp lý. Cần có chèn để giữ cho mảng được sắp xếp. Thao tác này rất kém hiệu quả, đặc biệt là với giá trị nhỏ hơn của khóa của phần tử được chèn $(n + 1)$, vì điều này dẫn đến việc dịch chuyển tất cả các phần tử lớn hơn theo một vị trí sang bên phải. Điều này đòi hỏi sự dịch chuyển trung bình là $n/2$ và trong trường hợp xấu nhất là n phần tử (n là số phần tử trước khi chèn).

Ở trên, chúng ta đã xem xét rõ ràng rằng tập hợp không cho phép lặp lại khóa. Tuy nhiên, không khó để thấy rằng ngay cả khi chúng ta cho phép lặp lại, tìm kiếm nhị phân sẽ hoạt động trở lại. Thật vậy, nó sẽ lại tìm thấy chính xác một phần tử, nếu một phần tử đó tồn tại. Bây giờ nó vẫn còn để xem xét rằng mảng đã được sắp xếp. Sau đó, các phần tử khác có khóa này phải được đặt theo một hoặc cả hai hướng, bên cạnh phần tử tìm được. Ý tưởng tương tự có thể được sử dụng để giải quyết vấn đề tổng quát hơn là tìm tất cả các phần tử nằm trong một khoảng cho trước.

Các vấn đề về nhu cầu di chuyển một số lượng lớn các mục cũng xảy ra khi xóa. Trên thực tế, khi xóa một số lượng nhỏ các mục, bạn chỉ cần đánh dấu chúng là đã xóa mà không cần di chuyển. Khi tìm kiếm, chúng nên được bỏ qua một cách thích hợp. Tốt nhất là vẫn sử dụng các khóa của họ trong quá trình tìm kiếm và chỉ trong trường hợp hoàn thành thành công, để kiểm tra xem mục tìm thấy có bị xóa hay không. Sau đó, thực hiện tìm kiếm khả thi giữa các phần tử lân cận của nó theo cả hai hướng.

Bài tập

▷ **4.9.** Tại các giá trị nào của n, tìm kiếm nhị phân sẽ nhanh hơn 10, 100, 1000 lần so với tìm kiếm tuần tự cổ điển (xem 4.2)?

▷ **4.10.** Để so sánh về mặt lý thuyết và thực nghiệm các biến thể khác nhau của tìm kiếm nhị phân.

4.5. Tìm kiếm Fibonacci

Mặc dù thoạt nhìn có vẻ lạ, nhưng đôi khi chúng ta có thể đạt được tốc độ tìm kiếm cao hơn nếu từ bỏ việc chia thành hai phần *bằng nhau*. Dưới đây chúng ta sẽ xem xét hai sự thay đổi của nhu cầu nhị phân, mỗi sự thay đổi dựa trên một sự cân nhắc cụ thể. Thật vậy, nhược điểm chính của tìm kiếm nhị phân là phép toán chia nặng được thực hiện ở mỗi bước của thuật toán. Thật không may, tách làm đôi là một thao tác khó và có thể làm chậm thuật toán. Do đó, chúng ta khuyến khích khi phát triển các ứng dụng thực tế đặt hàng

```
mid = (l + r) / 2;
```

được thay thế bởi

```
mid = (l + r) >> 1;
```

Sửa đổi được đề xuất sử dụng đáng kể sự biểu diễn bên trong của các số trong hệ thống số nhị phân và thực tế là phép chia 2 có thể được thay thế bằng cách loại bỏ chữ số cuối cùng của số bị chia, tức là bằng cách dịch chuyển một vị trí sang bên phải. Hầu như tất cả các bộ vi xử lý hiện đại đều có hướng dẫn máy như vậy. Chúng ta sẽ nhắc nhở rằng đối với bất kỳ hệ thống số nào với cơ số p là hợp lệ: Thương trong phép chia số nguyên của p với tư và phần dư có thể được tìm thấy bằng cách loại bỏ chữ số cuối cùng của phép chia (phần dư của phép chia).

Mặc dù rất khó xảy ra, nhưng có thể chiếc máy bạn đang sử dụng không có hướng dẫn thích hợp để chuyển sang phải một chút (các tác giả không biết về chiếc máy như vậy). Trong trường hợp này, sử dụng cái gọi là tìm kiếm Fibonacci là thích hợp. Nhớ lại rằng số Fibonacci được cho bởi công thức truy hồi (xem **??**):

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}, n \geq 2.$$

Tìm kiếm Fibonacci rất giống với nhị phân - mảng được chia thành hai phần, một trong số đó bị loại trừ khỏi việc xem xét thêm. Sự khác biệt duy nhất là cách chúng ta chọn mặt hàng để so sánh. Giả sử rằng việc tìm kiếm diễn ra trong một mảng n phần tử, chẳng hạn như $n = F_{k-1}$. Trong bước đầu tiên, chúng ta so sánh khóa yêu

cầu x với $m[F_{k-1}].key$. Trong trường hợp hòa, cuộc tìm kiếm kết thúc thành công. Trong trường hợp $x < m[F_{k-1}]$. Khóa ở bước tiếp theo ta xét các phần tử $1, 2, ..., F_{k-1} - 1$. Nếu $x > m[F_{k-1}]$. Khóa ở bước tiếp theo, các phần tử $F_{k-1} + 1, F_{k-1} + 2, ..., n = F_k$ vẫn được xem xét. Người ta tính trực tiếp rằng trong trường hợp đầu tiên cho bước tiếp theo, chúng ta thu được một dãy có độ dài $F_{k-1} - 1$, và trong trường hợp thứ hai - $F_{k-2} - 1$. Quá trình được lặp lại trên phần không bị từ chối cho đến khi tìm thấy phần tử cần thiết hoặc đạt được dãy trống (xem [Horowitz-1977]).

Khả năng thực hiện tìm kiếm Fibonacci trông như thế này ($l \geq 0$ và sao cho $F_{k+l} = n + 1$ và $F_{k+1} > n + 1$):

Chương trình 4.10. Tìm kiếm fibonacci(410fibsear.c)

```
unsigned fib[MAX]; /* Số Fibonacci không vượt quá n*/
unsigned findFib(unsigned n)
{ unsigned k;
  fib[0] = 0;
  fib[1] = 1;
  for (k = 2; ; k++)
    if ((fib[k] = fib[k-1] + fib[k-2]) > n)
      return k-1;
  return 0;
}

unsigned fibSearch(int key)
{ int p,q,r,k;
  k = findFib(n);
  p = fib[k-1];
  q = fib[k-2];
  r = fib[k-3];
  if (key > m[p].key)
    p += n - fib[k] + 1;
  while (p > 0)
    if (key == m[p].key)
      return p;
    else
      if (key < m[p].key)
        if (0 == r)
          p = 0;
```

```
        else {
          int t;
          p -= r;
          t  = q;
          q = r;
          r  = t-r;
        }
      else
        if (1 == q)
          p = 0;
        else {
          p += r;
          q -= r;
          r  -= q;
        }
    return NOT_FOUND;
}
```

Hiệu quả của tìm kiếm Fibonacci là gì? Hóa ra lợi ích của việc xóa bỏ sự chia rẽ không lớn. Giống như tìm kiếm nhị phân, thứ tự thực hiện so sánh được xác định trước cho mỗi giá trị của n. Trong thực tế, nhu cầu Fibonacci xây dựng một cây. Dễ dàng nhận thấy sự chênh lệch chiều cao của các cây con đối với mỗi ngọn nhiều nhất là 1, tức là cây đã cân đối. Loại cây này được gọi trong tài liệu là cây Fibonacci và đã được thảo luận trong phần **??**, nơi chúng ta đã chỉ ra rằng chúng là trường hợp xấu nhất của cây cân bằng, cao hơn 45% so với cây cân bằng lý tưởng tương ứng. Do đó, trong trường hợp xấu nhất, nhu cầu Fibonacci khám phá gỗ, cao hơn 45% so với nhu cầu nhị phân (xem 4.4), Và do đó chứng tỏ kém hiệu quả hơn.

Bài tập

▷ **4.11.** Để so sánh về mặt lý thuyết và thực nghiệm tìm kiếm nhị phân và Fibonacci.

4.5.

4.6. Tìm kiếm nội suy

Khi một người tìm kiếm "Bonev" trong danh bạ điện thoại, anh ta sẽ nhìn vào đầu danh bạ và khi tìm kiếm "Bonev" - ở cuối. Chúng ta không thể tối ưu hóa nhu cầu nhị phân theo cách tương tự (xem 4.4)?

Chúng ta hãy xem xét kỹ hơn. Trong tìm kiếm nhị phân, chúng ta chọn phần tử giữa là mid theo công thức $mid = (l + r)/2$. Hãy viết nó ra theo một cách khác:

$$mid = 1 + (r - 1)/2 \qquad (*)$$

Công thức kết quả cho thấy rằng vị trí phân chia tiếp theo của khoảng nhận được bằng cách thêm vào nửa đầu của độ dài của nó. Trong trường hợp danh bạ điện thoại, một danh bạ không hoạt động trên công thức này. Thay vì ở giữa, anh ta lần lượt tìm kiếm ở đầu hoặc cuối thư mục. Điều này có thể được biểu thị bằng công thức kiểu (*), cho mục đích 1/2 được thay thế bằng một số thích hợp khác. Làm thế nào để chọn số này? Hãy quay lại danh bạ điện thoại. Có một người cố gắng xác định vị trí gần đúng của tên được tìm kiếm, bắt đầu từ chữ cái đầu tiên của nó, biết các chữ cái đầu tiên và cuối cùng của bảng chữ cái. Theo cách tương tự, chúng ta có thể cố gắng tìm vị trí gần đúng của bất kỳ phần tử x nào, biết giá trị của x, cũng như các giá trị của đầu và cuối của khoảng được xem xét. Rõ ràng là vị trí tương đối này có thể được đưa ra bằng công thức nội suy:

```
mid = 1 + k * (r-1),
```

ở đâu

```
k = (x - m[l].key) / (m[r].key - m[l].key)
```

Hệ số k, thay thế cho hằng số 1/2, thực sự cho chúng ta vị trí gần đúng của phần tử cần thiết trong khoảng được xem xét. Trong việc thực hiện nội suy tìm kiếm cần tính đến một số tính năng cụ thể liên quan đến việc tính giá trị của k. Trước hết, chúng ta nên tránh chia cho 0, trong trường hợp giá trị của tất cả các khóa từ khoảng $[l, r]$ được xem xét trùng nhau. Thứ hai, chúng ta nên đảm bảo rằng $k \in [0, 1]$. Thật vậy, nếu không thì giá trị của giữa sẽ vượt ra ngoài

giới hạn của khoảng $[l, r]$ được xem xét. Khi k vượt quá $[0, 1]$, chúng ta có thể giả sử một cách an toàn rằng phần tử bắt buộc bị thiếu.

Hiệu quả của cải tiến được coi là gì? Trong trường hợp phân phối đều, tìm kiếm nội suy cho kết quả thực sự đáng chú ý! Khi đó giá trị gần đúng trên của k hóa ra lại rất gần với vị trí thực của phần tử được tìm kiếm và nó được tìm thấy nhanh hơn nhiều so với tìm kiếm nhị phân. Nó được chứng minh rằng khi đó tìm kiếm nội suy thực hiện ít hơn $\log_2(\log_2 n) + 1$ so sánh trong cả tìm kiếm thành công và không thành công. Để người đọc có thể đánh giá hiệu quả của phương pháp đã trình bày, chúng ta sẽ chỉ ra rằng hàm $\log_2(\log_2 n)$ phát triển cực kỳ chậm. Ví dụ $\log_2(\log_2 1000000000) < 5$.

Chương trình 4.11. Tìm kiếm nội suy(411interpol.c)

```
unsigned interpolSearch(int key)
{ unsigned l, r, mid;
  float k;
  l = 0; r = n - 1;
  while (l <= r) {
    if (m[r].key == m[l].key)
      if (m[l].key == key)
        return l;
      else
        return NOT_FOUND;
    k = (float) (key - m[l].key) / (m[r].key - m[l].key);
    if (k < 0 || k > 1)
      return NOT_FOUND;
      mid = (unsigned)(l + k*(r-l) + 0.5);
      if (key < m[mid].key)
        r = mid - 1;
      else if (key > m[mid].key)
        l = mid + 1;
      else
        return mid;
      }
  return NOT_FOUND;
}
```

Mặc dù có những ưu điểm không thể phủ nhận của tìm kiếm nội suy, nó nên được sử dụng hết sức thận trọng. Trước hết, không

nên quên rằng nó là một cải tiến (rất tốt) của nhu cầu nhị phân, đó là lý do tại sao nó kế thừa những thiếu sót chính và khó chịu nhất liên quan đến yêu cầu truy cập trực tiếp và quy định các phần tử của tập hợp. Thứ hai, cần lưu ý những yêu cầu bổ sung và những thiếu sót cần thiết (cái gì cũng có giá cả!). Một mặt, các khóa bắt buộc phải là số hoặc để dễ hiểu như vậy, sao cho k luôn nằm trong phạm vi $[0,1]$. Nếu không, phương pháp này có thể không áp dụng được. Thứ ba, cần đặc biệt chú ý đến thực tế là các số dấu phẩy động được sử dụng ở đây và phép toán chia số học nặng được sử dụng. Thật không may, chúng ta không thể loại bỏ phép chia ở đây dễ dàng như trong tìm kiếm nhị phân: một mặt, phép chia không còn là 2, và mặt khác, nó không phải là số nguyên.

Yêu cầu rằng các khóa của các phần tử được phân bố đồng đều cũng trở nên cực kỳ quan trọng. Nếu không đúng như vậy, thì tốc độ của nó có thể giảm xuống đáng kể so với tốc độ của tìm kiếm nhị phân, tiến tới tuyến tính. Không khó để xây dựng một ví dụ như vậy. Với mục đích này, chúng ta hãy thử tìm số 2 trong tập hợp sau:

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 10000

Ngoài ra, đối với các giá trị nhỏ của n, các số $\log_2 n$ và $\log_2(\log_2 n)$ không khác nhau đáng kể, do đó, người ta thường coi việc sử dụng tìm kiếm nội suy là không đáng để mạo hiểm. Mặt khác, đối với các tệp đặc biệt lớn, khóa lớn hoặc bố trí dữ liệu bên ngoài, khi việc so sánh đặc biệt tốn kém, thì nên ưu tiên tìm kiếm nội suy.

Bài tập

▷ **4.12.** Hãy so sánh về mặt lý thuyết và thực nghiệm tìm kiếm nhị phân và nội suy.

4.7. Câu hỏi và bài tập

▷ **4.13.** *Tìm tất cả các mục bằng một chìa khóa.*
Hãy sửa đổi các thuật toán tìm kiếm được đề xuất để chúng không chỉ tìm thấy một, mà là tất cả các phần tử có khóa.

▷ **4.14.** *Tìm kiếm trong ma trận.*
Thực hiện một hàm tìm một số trong ma trận kích thước $n \times m$.

▷ **4.15.** *Tìm kiếm trong một chuỗi ký tự.*
Hãy triển khai một hàm tìm kiếm một chuỗi ký tự trong một mảng chuỗi ký tự nhất định.

▷ **4.16.** *Phạm vi tiếp cận.*
Cho trước một mảng các số tự nhiên, một phần tử có khóa k (k - số tự nhiên) và một số tự nhiên d. Để đề xuất một thuật toán và thực hiện một hàm để tìm tất cả các phần tử x trong mảng mà $|k - x.key| \leq d$.