

Chương 2:

TÌM KIẾM

SEARCHING TECHNIQUES

Nội dung

2

1. Khái quát về tìm kiếm
2. Tìm tuyến tính (Linear Search)
3. Tìm nhị phân (Binary Search)

Khái quát về tìm kiếm

3

- Tìm kiếm là quá trình tìm một phần tử dữ liệu có một thành phần khóa (Key), có kiểu dữ liệu là T nào đó, các thành phần còn lại là thông tin (Info) liên quan đến phần tử dữ liệu đó cần thỏa mãn điều kiện tìm kiếm.

- Mỗi phần tử dữ liệu có cấu trúc dữ liệu như sau:

```
typedef struct DataElement  
{   T   Key;  
    InfoType Info;  
} DataType;
```

Ví dụ:

Tìm kiếm một sinh viên trong lớp

Tìm kiếm một tập tin, thư mục trong n

- Việc tìm kiếm một phần tử có thể diễn ra trên một dãy/mảng (tìm kiếm nội) hoặc diễn ra trên một tập tin/ file (tìm kiếm ngoại).

Khái quát về tìm kiếm

4

- Các giải thuật tìm kiếm nội:
 - ▣ Tìm kiếm tuyến tính (Linear Search) hay còn gọi là tìm kiếm tuần tự (Sequential Search)
 - ▣ Tìm kiếm nhị phân (Binary Search)

Nội dung

5

1. Khái quát về tìm kiếm
2. Tìm tuyến tính (Linear Search)
3. Tìm nhị phân (Binary Search)

2. Tìm tuyến tính (**Linear Search**)

6

Ý tưởng:

- Bắt đầu từ phần tử đầu tiên của danh sách, so sánh lần lượt từng phần tử của danh sách với giá trị X cần tìm
 - Nếu có phần tử bằng X , thuật toán dừng lại (thành công)
 - Nếu đến cuối danh sách mà không có phần tử nào bằng X , thuật toán dừng lại (không thành công)
- ▣ If we find a match, the search terminates successfully by returning the index of the element
- ▣ If the end of the list is encountered without a match, the search terminates unsuccessfully

2. Tìm tuyến tính (Linear Search)

7

Thuật toán:

□ Thuật toán:

- ▣ B1: $i = 10$ // Duyệt từ đầu mảng
- ▣ B2: $A[i] \neq X \text{ AND } i \leq N$ // Nếu chưa tìm thấy và cũng chưa duyệt hết mảng
 - ▣ B2.1: $i++$
 - ▣ B2.2: Lặp lại B2
- ▣ B3: IF $i \leq N$
 - ▣ Tìm thấy tại vị trí i
- ▣ B4: ELSE
 - ▣ Không tìm thấy phần tử có giá trị X
- ▣ B5: Kết thúc

2. Tìm tuyến tính (Linear Search)

9

9

Khóa tìm

0	1	2	3	4	5	6	7
7	13	5	21	6	2	8	15



Không tìm thấy

Số lần so sánh: 8

2. Tìm tuyến tính (Linear Search)

10

5

Khóa tìm

Vị trí = 2

0	1	2	3	4	5	6	7
7	13	5	21	6	2	8	15



Tìm thành công

Số lần so sánh: 3

2. Tìm tuyến tính (Linear Search)

11

```
void LinearSearch (int list[], int n, int key) {  
    int flag = 0;      // giả sử lúc đầu chưa tìm thấy  
    for(int i=0; i<n; i++)  
        if (list[i] == key) {  
            cout<<"found at position"<<i;  
            flag =1;    // tìm thấy  
            break;  
        }  
    if (flag == 0)  
        cout<<"not found";  
}
```

2. Tìm tuyến tính (Linear Search)

12

```
int lsearch(int list[], int n, int key)
{
    int find= -1;
    for(int i=0; i<n; i++)
        if (list[i] == key)
        {
            find = i;
            break;
        }
    return find;
}
```

2. Tìm tuyến tính (Linear Search)

13

```
1 print("1.Tìm kiếm tuyến tính")
2 def linear_search(a,x):
3     #Duyệt qua từng phần tử của danh sách a
4     for i in range(len(a)):
5         #So sánh phần tử đang xét với giá trị cần tìm kiếm x
6         if a[i]==x:
7             #Trả về vị trí của phần tử tìm thấy
8             return i
9     #Nếu đã duyệt qua toàn bộ danh sách mà không tìm thấy giá trị cần tìm kiếm,
10    # trả về giá trị không tìm thấy
11    return -1
12 a=[10,30,20,45,67,32,15,90,86,120]
13 print(a)
14 print("Nhập giá trị cần tìm")
15 x =int(input())
16 kq =linear_search(a,x)
17 if kq==-1:
18     print("Không tìm thấy giá trị",x)
19 else:
20     print("Giá trị ",x,"được tìm thấy tại vị trí",kq)
```

Chú ý

Tìm kiếm tuyến tính

Phân tích thuật toán:

- Trường hợp tốt nhất khi phần tử đầu tiên của mảng có giá trị bằng X:
 - ▣ Số phép gán: $G_{min} = 1$
 - ▣ Số phép so sánh: $S_{min} = 2 + 1 = 3$
- Trường hợp xấu nhất khi không tìm thấy phần tử nào có giá trị bằng X:
 - ▣ Số phép gán: $G_{max} = 1$
 - ▣ Số phép so sánh: $S_{max} = 2N + 1$
- Trung bình:
 - ▣ Số phép gán: $G_{avg} = 1$
 - ▣ Số phép so sánh: $S_{avg} = (3 + 2N + 1) : 2 = N + 2$

□ Phân tích, đánh giá thuật toán

Trường hợp	Số lần so sánh	Giải thích
Tốt nhất	1	Phần tử đầu tiên có giá trị x
Xấu nhất	n	Phần tử cuối cùng có giá trị x
Trung bình	$n/2$	Giả sử xác suất các phần tử trong mảng nhận giá trị x là như nhau.

- Vậy giải thuật tìm tuyến tính có độ phức tạp tính toán cấp n: $T(n) = O(n)$

Tìm kiếm tuyến tính

Nhận xét:

- Giải thuật tìm tuyến tính không phụ thuộc vào thứ tự của các phần tử trong danh sách, do vậy đây là phương pháp tổng quát nhất để tìm kiếm trên một danh sách bất kỳ.
- Một thuật toán có thể được cài đặt theo nhiều cách khác nhau, kỹ thuật cài đặt ảnh hưởng đến tốc độ thực hiện của thuật toán.

Nội dung

17

1. Khái quát về tìm kiếm
2. Tìm tuyến tính (Linear Search)
3. Tìm nhị phân (Binary Search)

3. Tìm nhị phân (**Binary Search**)

18

□ Điều kiện:

- Danh sách phải được sắp xếp trước $a[i-1] \leq a[i] \leq a[i+1]$

□ Ý tưởng:

- So sánh giá trị muốn tìm X với phần tử nằm ở vị trí giữa của danh sách:
 - Nếu bằng, tìm kiếm dừng lại (thành công)
 - Nếu X lớn hơn thì tiếp tục tìm kiếm ở phần danh sách bên phải phần tử giữa
 - Nếu X nhỏ hơn thì tiếp tục tìm kiếm ở phần danh sách bên trái phần tử giữa
- We compare the element with the element placed approximately in the middle of the list
 - If a match is found, the search terminates successfully
 - Otherwise, we continue the search for the key in a similar manner either in the upper half or the lower half

3. Tìm nhị phân (**Binary Search**)

19

Thuật toán:

B1: $\text{Left} = 0, \text{Right} = n-1$

B2: $\text{Mid} = (\text{Left} + \text{Right})/2$ // lấy vị trí cận giữa

B3: So sánh X với $A[\text{Mid}]$, có 3 khả năng xảy ra:

- $A[\text{Mid}] = X$ // tìm thấy. Dừng thuật toán

- $A[\text{Mid}] > X$

$\text{Right} = \text{Mid}-1$ // Tiếp tục tìm trong dãy $A[0] \dots A[\text{Mid}-1]$

- $A[\text{Mid}] < X$

$\text{Left} = \text{Mid}+1$ // Tiếp tục tìm trong dãy $A[\text{Mid}+1] \dots A[\text{Right}]$

B4: Nếu $(\text{Left} \leq \text{Right})$ // Còn phần tử chưa xét

Lặp lại B2

Ngược lại: Kết thúc

3. Tìm nhị phân (**Binary Search**)

20

10

Khóa tìm

Vi trí = 3

Khóa cần tìm ~~lớn hơn~~ hoặc bằng

0	1	2	3	4	5	6	7	8	9
2	5	8	10	12	13	15	18	21	24
↑				↑					↑
left				mid					right

Tìm thấy

Số lần so sánh: 4

3. Tìm nhị phân (**Binary Search**)

22

```
void BSearch (int list[], int n, int key)
{
    int left, right, mid, flag = 0;
    left = 0; right = n-1;
    while (left <= right)
    {
        mid = (left + right)/2;
        if( list[mid] == key)
        {
            cout<<"found:"<< mid;
            flag =1;    // đánh dấu tìm thấy
            break;
        }
        else if (list[mid] < key) left = mid +1;
        else      right = mid -1;
    }
    if (flag == 0)
        cout<<"not found";
}
```

Không đệ quy

3. Tìm nhị phân (**Binary Search**)

23

```
int BSearch_Recursion (int list[], int key, int left, int right)    Đệ quy
{
    if (left <= right)
    {
        int mid = (left + right)/2;
        if (key == list[mid])
            return mid;      // trả về vị trí tìm thấy key
        else if (key < list[mid])
            return BSearch_Recursion (list, key, left, mid-1);
        else return BSearch_Recursion (list, key, mid+1, right);
    }
    else return -1;      // không tìm thấy
}
```

3. Tìm nhị phân (**Binary Search**)

24

```
22 print("2.Tìm kiếm nhị phân ")
23 def binary_search(a,x):
24     #Thiết lập các chỉ mục ban đầu
25     low =0
26     high = len(a)-1
27     mid=0
28     #Duyệt qua mảng để tìm kiếm phần tử x
29     while low<=high:
30         mid =(high +low)//2
31         #Kiểm tra xem phần tử có ở giữa không
32         if a[mid] <x:
33             low =mid+1
34         #Kiểm tra xem phần tử có ở giữa không
35         elif a[mid]>x:
36             high =mid-1
37         #Nếu phần tử được tìm thấy ở giữa thì trả về vị trí đó
38         else:
39             return mid
40     #Nếu không tìm thấy phần tử trong mảng, trả về -1
41     return -1
```

Chú ý

3. Tìm nhị phân (**Binary Search**)

25

```
42  #Kiểm tra
43  a=[10,30,20,45,67,32,15,90,86,120]
44  print(a)
45  print("Nhập giá trị cần tìm")
46  x =int(input())
47  kq =binary_search(a,x)
48  if kq==-1:
49      print("Không tìm thấy giá trị",x)
50  else:
51      print("Giá trị ",x,"được tìm thấy tại vị trí",kq)
```

Tìm kiếm nhị phân

- Phân tích thuật toán đệ quy:
- Trường hợp tốt nhất khi phần tử ở giữa của mảng có giá trị bằng X:
 - ▣ Số phép gán: $G_{\min} = 1$
 - ▣ Số phép so sánh: $S_{\min} = 2$
- Trường hợp xấu nhất khi không tìm thấy phần tử nào có giá trị bằng X:
 - ▣ Số phép gán: $G_{\max} = \log_2 N + 1$
 - ▣ Số phép so sánh: $S_{\max} = 3\log_2 N + 1$
- Trung bình:
 - ▣ Số phép gán: $G_{\text{avg}} = \frac{1}{2} \log_2 N + 1$
 - ▣ Số phép so sánh: $S_{\text{avg}} = \frac{1}{2}(3\log_2 N + 3)$

3. Tìm nhị phân (**Binary Search**)

27

- Phân tích, đánh giá thuật toán:

Trường hợp	Số lần so sánh	Giải thích
Tốt nhất	1	Phần tử giữa của mảng có giá trị x
Xấu nhất	$\log_2 n$	Không có x trong mảng
Trung bình	$\log_2 (n/2)$	Giả sử xác suất các phần tử trong mảng nhận giá trị x là như nhau

- Vậy giải thuật tìm nhị phân có độ phức tạp tính toán cấp n: $T(n) = O(\log_2 n)$

Tìm kiếm nhị phân

Thuật toán không đệ quy:

- B1: First = 1, Last = N
- B2: IF (First > Last)
 - ▣ B2.1: Không tìm thấy
 - ▣ B2.2: Thực hiện Bkt
- B3: Mid = (First + Last) / 2
- B4: IF (X = M[Mid])
 - ▣ B4.1: Tìm thấy tại vị trí Mid
 - ▣ B4.2: Thực hiện Bkt
- B5: IF (X < M[Mid])
 - ▣ B5.1: Last = Mid - 1
 - ▣ B5.2: Lặp lại B3
- B6: IF (X > M[Mid])
 - ▣ B6.1: First = Mid + 1
 - ▣ B6.2: Lặp lại B3
- Bkt: Kết thúc

Tìm kiếm nhị phân

```
int BinarySearch(int a[], int n, int x) //Không đệ qui
{
    int first = 0, last = n-1, mid;
    while (first <= last)
    {
        mid = (first + last) / 2;
        if (x == a[mid]) return mid; //Tìm thấy x tại mid
        if (x < a[mid]) last = mid - 1;
        else first = mid + 1;
    }
    return -1; // trong dãy không có x
}
```

Tìm kiếm nhị phân

Phân tích thuật toán không đệ quy:

- Trường hợp tốt nhất khi phần tử ở giữa của mảng có giá trị bằng X:
 - ▣ Số phép gán: $G_{\min} = 3$
 - ▣ Số phép so sánh: $S_{\min} = 2$
- Trường hợp xấu nhất khi không tìm thấy phần tử nào có giá trị bằng X:
 - ▣ Số phép gán: $G_{\max} = 2\log_2 N + 4$
 - ▣ Số phép so sánh: $S_{\max} = 3\log_2 N + 1$
- Trung bình:
 - ▣ Số phép gán: $G_{\text{avg}} = \log_2 N + 3.5$
 - ▣ Số phép so sánh: $S_{\text{avg}} = \frac{1}{2}(3\log_2 N + 3)$

Tìm kiếm nhị phân

31

□ *Đánh giá giải thuật:*

Trường hợp	Số lần so sánh	Giải thích
Tốt nhất	1	Phần tử giữa của mảng có giá trị x
Xấu nhất	$\log_2 n$	Không có x trong mảng
Trung bình	$\log_2 (n/2)$	Giả sử xác suất các phần tử trong mảng nhận giá trị x là như nhau

- Giải thuật tìm nhị phân có độ phức tạp tính toán cấp \log_n :

$$T(n) = O(\log_2 n)$$

Tìm kiếm nhị phân

Nhận xét:

- ▣ Giải thuật tìm nhị phân dựa vào quan hệ giá trị của các phần tử mảng để định hướng trong quá trình tìm kiếm, do vậy chỉ áp dụng được cho những dãy đã có thứ tự.
- ▣ Giải thuật tìm nhị phân tiết kiệm thời gian hơn rất nhiều so với giải thuật tìm tuần tự do

$$T_{\text{nhị phân}}(n) = O(\log_2 n) < T_{\text{tuần tự}}(n) = O(n).$$

- ▣ Các thuật toán đệ quy có thể ngắn gọn song tốn kém bộ nhớ để ghi nhận mã lệnh chương trình (mỗi lần gọi đệ quy) khi chạy chương trình, do vậy có thể làm cho chương trình chạy chậm lại. Trong thực tế, khi viết chương trình nếu có thể chúng ta nên sử dụng thuật toán không đệ quy.

Tìm kiếm nhị phân

Nhận xét:

- ▣ Khi muốn áp dụng giải thuật tìm nhị phân cần phải xét đến thời gian sắp xếp dãy số để thỏa điều kiện dãy số có thứ tự. Thời gian này không nhỏ, và khi dãy số biến động cần phải tiến hành sắp xếp lại => khuyết điểm chính cho giải thuật tìm nhị phân.
- ▣ Cần cân nhắc nhu cầu thực tế để chọn một trong hai giải thuật tìm kiếm trên sao cho có lợi nhất.

Tìm kiếm nhị phân

Hàm tìm kiếm bisect và insort trong Python

- Trong Python, hàm tìm kiếm nhị phân có sẵn được gọi là bisect. Hàm này cho phép tìm kiếm một giá trị trong một danh sách đã được sắp xếp theo thứ tự tăng dần và trả về vị trí của giá trị đó trong danh sách.

- Cú pháp:

Bisect(list, value, lo=0, hi=len(list))

- ▣ List: danh sách cần tìm kiếm giá trị đã được sắp xếp theo thứ tự tăng dần
- ▣ Value: giá trị cần tìm kiếm
- ▣ Lo: là chỉ số bắt đầu tìm kiếm trong danh sách mặc định là 0
- ▣ hi: là chỉ số kết thúc tìm kiếm trong danh sách mặc định là độ dài của danh sách.

Tìm kiếm nhị phân

- Hàm bisect sẽ trả về vị trí đầu tiên trong danh sách mà giá trị cần tìm kiếm có thể được chèn vào mà vẫn giữ được thứ tự tăng dần của danh sách.
- Ngoài ra, Python còn có hàm insert để chèn một giá trị vào danh sách đã được sắp xếp theo thứ tự tăng dần và vẫn giữ được thứ tự tăng dần của danh sách.
Hàm insert sử dụng bisect để xác định vị trí để chèn giá trị vào danh sách.
- Cú pháp của hàm insert là:
 - ▣ `insert(list, value, lo=0, hi=len(list))`
- Trong đó, các tham số có ý nghĩa tương tự như trong hàm bisect.

Tìm kiếm nhị phân

- Hàm bisect sẽ trả về vị trí đầu tiên trong danh sách mà
Dưới đây là một số ví dụ về cách sử dụng
hàm bisect và insort trong Python:
- ví dụ về cách sử dụng hàm bisect
- `print("3. Dùng Bisect và inSort")`
- `from bisect import inSort`
- `a=[10,30,20,45,67,32,15,90,86,120]`
- `inSort(a,3)`
- `print(a)`
- `inSort(a,0)`
- `print(a)`

Các giải thuật tìm kiếm ngoại (Tìm kiếm trên tập tin)

Đặt vấn đề

- Giả sử chúng ta có một tập tin F lưu trữ N phần tử. Vấn đề đặt ra là có hay không phần tử có giá trị bằng X được lưu trữ trong tập tin F ? Nếu có thì phần tử có giá trị bằng X là phần tử nằm ở vị trí nào trên tập tin F ?

1. Tìm tuyến tính

a. Ý tưởng:

- Lần lượt đọc các phần tử từ đầu tập tin F và so sánh với giá trị X cho đến khi đọc được phần tử có giá trị X hoặc đã đọc hết tập tin F thì kết thúc.

Các giải thuật tìm kiếm ngoại (Tìm kiếm trên tập tin)

b. Thuật toán:

B1: $k = 0$

B2: `rewind(F)` //Về đầu tập tin F

B3: `read(F, a)` //Đọc một phần tử từ tập tin F

B4: $k = k + \text{sizeof}(T)$ //Vị trí phần tử hiện hành (sau phần tử mới đọc)

B5: IF $a \neq X$ AND $!(\text{eof}(F))$

 Lặp lại B3

B6: IF $(a = X)$

 Tìm thấy tại vị trí k byte(s) tính từ đầu tập tin

B7: ELSE

 Không tìm thấy phần tử có giá trị X

B8: Kết thúc

Các giải thuật tìm kiếm ngoại (Tìm kiếm trên tập tin)

c. Cài đặt thuật toán:

```
long FLinearSearch (char * FileName, T X)
{
    FILE * Fp;
    Fp = fopen(FileName, "rb");
    if (Fp == NULL)
        return (-1);
    long k = 0;
    T a;
    int SOT = sizeof(T);

    while (!eof(Fp))
    {
        if (fread(&a, SOT, 1, Fp) == 0)
            break;
        k = k + SOT;
        if (a == X)
            break;
    }

    fclose(Fp);
    if (a == X)
        return (k - SOT);
    return (-1);
}
```

Các giải thuật tìm kiếm ngoại (Tìm kiếm trên tập tin)

d. Phân tích thuật toán:

- Trường hợp tốt nhất khi phần tử đầu tiên của tập tin có giá trị bằng X:
 - ▣ Số phép gán: $G_{\min} = 1 + 2 = 3$
 - ▣ Số phép so sánh: $S_{\min} = 2 + 1 = 3$
 - ▣ Số lần đọc tập tin: $D_{\min} = 1$
- Trường hợp xấu nhất khi không tìm thấy phần tử nào có giá trị bằng X:
 - ▣ Số phép gán: $G_{\max} = N + 2$
 - ▣ Số phép so sánh: $S_{\max} = 2N + 1$
 - ▣ Số lần đọc tập tin: $D_{\max} = N$
- Trung bình:
 - ▣ Số phép gán: $G_{\text{avg}} = \frac{1}{2}(N + 5)$
 - ▣ Số phép so sánh: $S_{\text{avg}} = (3 + 2N + 1) : 2 = N + 2$
 - ▣ Số lần đọc tập tin: $D_{\text{avg}} = \frac{1}{2}(N + 1)$

Các giải thuật tìm kiếm ngoại (Tìm kiếm trên tập tin)

2. Tìm kiếm theo chỉ mục (Index Search)

- Một tập tin dữ liệu thường có thêm các tập tin chỉ mục (Index File) để làm nhiệm vụ điều khiển thứ tự truy xuất dữ liệu trên tập tin theo một khóa chỉ mục (Index key) nào đó.
- Mỗi phần tử dữ liệu trong tập tin chỉ mục IDX gồm:

```
typedef struct IdxElement  
{  
    T IdxKey; // Khóa chỉ mục  
    long Pos; // Vị trí vật lý  
} IdxType;
```

- Tập tin chỉ mục luôn luôn được sắp xếp theo thứ tự tăng của khóa chỉ mục.

Các giải thuật tìm kiếm ngoại (Tìm kiếm trên tập tin)

2. Tìm kiếm theo chỉ mục (Index Search)

a. Ý tưởng:

- Lần lượt đọc các phần tử từ đầu tập tin IDX và so sánh thành phần khóa chỉ mục với giá trị X cho đến khi đọc được phần tử có giá trị khóa chỉ mục lớn hơn hoặc bằng X hoặc đã đọc hết tập tin IDX thì kết thúc.
- Nếu tìm thấy thì ta đã có vị trí vật lý của phần tử dữ liệu trên tập tin dữ liệu F, khi đó chúng ta có thể truy cập trực tiếp đến vị trí này để đọc dữ liệu của phần tử tìm thấy.

Các giải thuật tìm kiếm ngoại (Tìm kiếm trên tập tin)

2. Tìm kiếm theo chỉ mục (Index Search)

b. Thuật toán:

- ▣ B1: rewind(IDX)
- ▣ B2: read(IDX, ai)
- ▣ B3: IF ai.IdxKey < X AND !(eof(IDX))

Lặp lại B2

- ▣ B4: IF ai.IdxKey = X

Tìm thấy tại vị trí ai.Pos byte(s) tính từ đầu tập tin

- ▣ B5: ELSE

Không tìm thấy phần tử có giá trị X

- ▣ B6: Kết thúc

Các giải thuật tìm kiếm ngoại (Tìm kiếm trên tập tin)

c. Cài đặt thuật toán:

```
long IndexSearch (char * IdxFileName, T X)
{ FILE * IDXFP;
  IDXFP = fopen(IdxFileName, "rb");
  if (IDXFp == NULL)
    return (-1);
  IdxType ai;
  int SOIE = sizeof(IdxType);
```

Các giải thuật tìm kiếm ngoại (Tìm kiếm trên tập tin)

c. Cài đặt thuật toán:

```
while (!feof(IDXFp))
{
    if (fread(&ai, SOIE, 1, IDXFp) == 0)
        break;
    if (ai.IdxKey >= X)
        break;
}

fclose(IDXFp);
if (ai.IdxKey == X)
    return (ai.Pos);
return (-1);
}
```

Các giải thuật tìm kiếm ngoại (Tìm kiếm trên tập tin)

d. Phân tích thuật toán:

- Trường hợp tốt nhất khi phần tử đầu tiên của tập tin chỉ mục có giá trị khóa chỉ mục lớn hơn hoặc bằng X:
 - ▣ Số phép gán: $G_{min} = 1$
 - ▣ Số phép so sánh: $S_{min} = 2 + 1 = 3$
 - ▣ Số lần đọc tập tin: $D_{min} = 1$
- Trường hợp xấu nhất khi mọi phần tử trong tập tin chỉ mục đều có khóa chỉ mục nhỏ hơn giá trị X:
 - ▣ Số phép gán: $G_{max} = 1$
 - ▣ Số phép so sánh: $S_{max} = 2N + 1$
 - ▣ Số lần đọc tập tin: $D_{max} = N$
- Trung bình:
 - ▣ Số phép gán: $G_{avg} = 1$
 - ▣ Số phép so sánh: $S_{avg} = (3 + 2N + 1) : 2 = N + 2$
 - ▣ Số lần đọc tập tin: $D_{avg} = \frac{1}{2}(N + 1)$