Hash Table

Searching Algorithms

Không gian tìm kiếm

Là đối tượng mà thuật toán tìm kiếm thực hiện lên, thường là mảng, đoạn giá trị nào đó, danh sách liên kết, cây,...

Điều kiện tìm kiếm

Là tiêu chuẩn tìm kiếm trình bày dưới dạng phát biểu không hình thức và lập trình viên cần hình thức hóa nó thành một biểu thức logic.

Linear Search

Idea

Duyệt toàn bộ danh sách A để xác định phần tử a_i cần tìm và trả về vị trí i của nó.

Input - Output

- **Input** : Danh sách A có n phần tử, giá trị khóa x cần tìm.
- **Output** : Chỉ số i của phần tử a_i trong A có giá trị khóa là x.

Complexity

- **Best case** : a_0 chứa khóa x - số lần lặp lại là 1 - độ phức tạp hằng số $O(1)$.
- **Worst case** : A không chứa phần tử có khóa x - số lần lặp là n - độ phức tạp tuyến tính $O(n)$.
- **Average case** : Độ phức tạp tuyến tính $O(n)$.

Note

Linear Search sẽ duyệt toàn bộ không gian tìm kiếm nên các đối tượng tìm kiếm không cần được sắp xếp, có thể nói là dữ liệu không cần được tổ chức.

Code

```
int linearSearch(int *a,int n,int x)
{
    for(int i = 0;i < n;i++){

        if(a[i] == x){

            return i;
        }
    }
    return -1;
}
```

Binary Search

Idea

Chọn a_M ở giữa A để so sánh với khóa x. A được chia thành hai phần trước và sau a_M. Chỉ số bắt đầu và kết thúc của A là L và R.

M = $\frac{L+R}{2}$, làm tròn phần nguyên.

So sánh x và a_M:

- Nếu x < a_M, thì tìm x trong đoạn bên trái a_M.
- Ngược lại thì tìm x trong đoạn bên phải a_M.
- Nếu x = a_M tức là đã tìm thấy và dừng thuật toán.

Mảng giảm dần thì làm ngược lại.

Input - Output

- **Input** : Danh sách A có n phần tử đã sắp xếp theo thứ tự *P* (tăng dần hoặc giảm dần), giá trị khóa x cần tìm.
- **Output** : Chỉ số i của phần tử a_i trong A có giá trị khóa là x. Không tìm thấy trả về i = -1.

Complexity

- **Best case** : Phần tử cần tìm nằm ở vị trí (L + R)/2 – Số lần lặp là 1 – Độ phức tạp hằng số $O(1)$.
- **Worst case** : Số lần tìm là số lần chia đôi đến khi dãy tìm kiếm còn 1 phần tử - Lặp khoảng $log_2(n) + 1$ lần – Độ phức tạp logarithm $O(log_2(n))$.
- **Average case** : Độ phức tạp là $O(log_2(n))$.

Note

Điều kiện tìm kiếm là không gian tìm kiếm phải được sắp xếp theo một thứ tự nào đó và ta sẽ dựa theo thứ tự này để tìm kiếm.

Code

Đệ quy

```
int binarySearch(int *a, int left, int right, int x)
{
    if (left > right)
    {
        return -1;
    }
```

```
    int mid = (left + right) / 2;
    if (x == a[mid])
        return mid;
    if (x < a[mid])
        return binarySearch(a, left, mid - 1, x);
    else
        return binarySearch(a, mid + 1, right, x);
}
```

Khử đệ quy

```
int binarySearch(int *a,int n,int x)
{
    int L = 0,R = n - 1;
    while(L <= R)
    {
        int M = (L + R) / 2;
        if(x == a[M])

            return M;
        if(x > a[M]){

            R = M - 1;
        }
        else{
            L = M + 1;
        }
    }
    return -1;
}
```

Interpolation Search

Idea

Thay vì xác định điểm M = (L+R)/2 thì xác định M như sau:

$$M = L + \frac{(R - L)(x - A[L])}{A[R] - A[L]}$$

Các bước còn lại như Binary Search. Đây là một phương pháp trong xác suất thống kê, bởi vì vậy nên nó mới áp dụng cho phân bố dữ liệu ngẫu nhiên

đồng đều.

Input - Output

Input : Danh sách A có n phần tử đã có thứ tự *P* (tăng dần hoặc giảm dần), giá trị x cần tìm.

Output : Chỉ số i của phần tử a_i trong A có giá trị khóa là x. Không tìm thấy trả về i = -1.

Complexity

- **Best case** : Tương tự Binary Search khi mà $a_M = x$ thì số lần lặp là 1, độ phức tạp sẽ là hằng số $O(1)$.
- **Worst case** : Trong trường hợp giá trị số của khóa tăng theo cấp số nhân thì độ phức tạp là $O(n)$.
- **Average case** : Nếu như dữ liệu được phân bố liên tục thì độ phức tạp sẽ là $log_2(log_2(n))$.

Note

Thuật toán Interpolation Search áp dụng trong trường hợp không gian tìm kiếm đã được tổ chức theo một thứ tự (tăng dần hoặc giảm dần) và khóa cần tìm được phân bố liên tục trong không gian tìm kiếm (rải đều).

Code

```
int interpolationSearch(int *a,int n,int x)
{
    int L = 0, R = n - 1;
    while(L <= R) {
        int M = L + (R - L)*((x - a[L])/(a[R] - a[L]));
        if(x == a[M]){
            return M;
        }
        if(x > a[M]){
            R = M - 1;
        }
        else{
```

```
                L = M + 1;
            }
        }
        return -1;
    }
```