# CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

## CHƯƠNG 2: TÌM KIẾM VÀ SẮP XẾP

# Nội dung

- 2.1. Các giải thuật tìm kiếm nội
  - 2.1.1. Tìm kiếm tuyến tính
  - 2.1.2. Tìm kiếm nhị phân

# Khái quát về tìm kiếm

- Là công việc thường xuyên trong xử lý dữ liệu
- Tìm: Thấy hay không? Nếu thấy thì ở vị trí nào?
- Khóa tìm kiếm (key): Có tính duy nhất với mỗi phần tử dùng để nhận diện (ID).
- Cấu trúc chung cho một phần tử:

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

- Phép tìm kiếm chỉ quan tâm đến Key=> Coi dãy dữ liệu chỉ có Key

# Các giải thuật tìm kiếm nội(bộ nhớ trong)

- Bài toán
  - Cho một mảng A gồm N phần tử. Vấn đề đặt ra là có hay không phần tử có giá trị bằng X trong A? Nếu có thì phần tử có giá trị bằng X là phần tử thứ mấy trong A?
- Tìm tuyến tính (Linear Search)
  - Còn được gọi là Tìm kiếm tuần tự (Sequential Search).
  - Tư tưởng:
    - Lần lượt so sánh các phần tử của mảng A với giá trị X bắt đầu từ phần tử đầu tiên cho đến khi tìm đến được phần tử có giá trị X hoặc đã duyệt qua hết tất cả các phần tử của mảng A thì kết thúc.

# Tìm kiếm tuyến tính

- b. Thuật toán: Dùng giả mã
  - B1: k = 1 //Duyệt từ đầu mảng
  - B2: IF A[k] ≠ X AND k ≤N //Nếu chưa tìm thấy và cũng chưa duyệt hết mảng
    - k++
    - Lặp lại B2
  - B3: IF k ≤ N
    Tìm thấy tại vị trí k
  - B4: ELSE
    Không tìm thấy phần tử có giá trị X
  - B5: Kết thúc

# Tìm kiếm tuyến tính

- c. Cài đặt thuật toán:

Hàm LinearSearch có prototype:

```
int LinearSearch (T M[], int N, T X);
    { int k = 0;
    while (M[k] != X && k < N) k++;
    if (k < N)  return (k);
    return (-1);
    }
```

# Tìm kiếm tuyến tính

- d. 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: $S_{avg} = (3 + 2N + 1) : 2 = N + 2$

# Tìm kiếm tuyến tính

- e. Cải tiến thuật toán:
  - Trong thuật toán trên, ở mỗi bước lặp cần phải thực hiện 2 phép so sánh.
  - Có thể giảm bớt 1 phép so sánh nếu thêm vào cuối mảng một **phần tử cầm canh** (sentinel/stand by) có giá trị bằng X.
- B1: k = 1
- B2: M[N+1] = X //Phần tử cầm canh
- B3: IF M[k] ≠ X
  - k++
  - Lặp lại B3
- B4: IF k ≤ N Tìm thấy tại vị trí k
- B5: ELSE

  Không tìm thấy phần tử có giá trị X
- B6: Kết thúc

# Tìm kiếm tuyến tính

- Hàm LinearSearch được viết lại thành hàm LinearSearch1 như sau:

```
int LinearSearch1 (T M[], int N, T X)
{
    int k = 0;
    M[N] = X;
    while (M[k] != X)  k++;
    if (k < N) return (k);
    return (-1);
}
```

# Tìm kiếm tuyến tính

- f. Phân tích thuật toán cải tiến:

Trường hợp tốt nhất khi phần tử đầu tiên của có giá trị bằng X:

Số phép gán: $G_{min} = 2$

Số phép so sánh: $S_{min} = 1 + 1 = 2$

Trường hợp xấu nhất khi không tìm thấy phần tử nào có giá trị bằng X:

Số phép gán: $G_{max} = 2$

Số phép so sánh: $S_{max} = (N+1) + 1 = N + 2$

Trung bình:

Số phép gán: $G_{avg} = 2$

Số phép so sánh: $S_{avg} = (2 + N + 2) : 2 = N/2 + 2$

- - Như vậy, nếu thời gian thực hiện phép gán không đáng kể thì thuật toán cải tiến sẽ chạy nhanh hơn thuật toán nguyên thủy

# Tìm kiếm nhị phân (Binary Search)

- Thuật toán tìm tuyến tính đơn giản và thuận tiện trong trường hợp số phần tử của dãy không lớn.
- Khi số phần tử của dãy khá lớn thì mất rất nhiều thời gian.
- Trong thực tế, thông thường các phần tử của dãy đã có một thứ tự, do vậy cần có thuật toán khác để rút ngắn thời gian tìm kiếm trên dãy đã có thứ tự.
- Thuật toán này gọi là ***Tìm kiếm nhị phân*** với tư tưởng:
  - Giả sử các phần tử trong dãy đã có thứ tự tăng.
  - Khi đó, nếu X nhỏ hơn giá trị phần tử đứng ở giữa dãy (M[Mid]) thì X chỉ có thể tìm thấy ở nửa đầu của dãy và ngược lại, nếu X lớn hơn phần tử M[Mid] thì X chỉ có thể tìm thấy ở nửa sau của dãy.

# Tìm kiếm nhị phân (Binary Search)

- Tư tưởng:
  - Phạm vi tìm kiếm ban đầu của chúng ta là từ phần tử đầu tiên của dãy (First = 1) cho đến phần tử cuối cùng của dãy (Last = N).
  - So sánh giá trị X với giá trị phần tử đứng ở giữa của dãy M là M[Mid].
    - Nếu X = M[Mid]: Tìm thấy
    - Nếu X < M[Mid]: Rút ngắn phạm vi tìm kiếm về nửa đầu của dãy M (Last = Mid–1)
    - Nếu X > M[Mid]: Rút ngắn phạm vi tìm kiếm về nửa sau của dãy M (First = Mid+1)
- Lặp lại quá trình này cho đến khi tìm thấy phần tử có giá trị X hoặc phạm vi tìm kiếm của chúng ta không còn nữa (First > Last).

# Tìm kiếm nhị phân (Binary Search)

- **Thuật toán đệ quy** (Recursion Algorithm):
  - B1: First = 1
  - B2: Last = N
  - B3: IF (First > Last) //Hết phạm vi tìm kiếm
    - Không tìm thấy
    - Nhảy đến Kết thúc
  - B4: Mid = (First + Last)/ 2 // Lấy phần tử giữa
  - B5: IF (X = M[Mid])
    - Tìm thấy tại vị trí Mid
    - Nhảy đến Kết thúc
  - B6: IF (X < M[Mid]) // Tìm ở dãy bên trái
    - Tìm đệ quy từ First đến Last = Mid – 1
  - B7: IF (X > M[Mid]) // Tìm ở dãy bên phải
    - Tìm đệ quy từ First = Mid + 1 đến Last
  - Kết thúc

# Tìm kiếm nhị phân (Binary Search)

- Cài đặt thuật toán đệ quy:
- Hàm RecBinarySearch thực hiện việc tìm kiếm phần tử có giá trị X trên mảng M trong phạm vi từ phần tử thứ First đến phần tử thứ Last. Nếu tìm thấy, hàm trả về một số nguyên có giá trị từ First đến Last là **vị trí** tương ứng của phần tử tìm thấy.
- Trong trường hợp ngược lại, hàm trả về giá trị –1 (**không tìm thấy**). Nội dung của hàm như sau:

```
int RecBinarySearch (T M[], int First, int Last, T X)
{ if (First > Last) return (-1);
int Mid = (First + Last)/2;
if (X == M[Mid]) return (Mid);
if (X < M[Mid]) return(RecBinarySearch(M, First, Mid – 1, X));
else return(RecBinarySearch(M, Mid + 1, Last, X)); }
```

- Gọi ban đầu: RecBinarySearch(M[],0,n-1,X);

# Tìm kiếm nhị phân (Binary Search)

- . **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: Gmin = 1
    - Số phép so sánh: Smin = 2
  - Trường hợp xấu nhất khi không tìm thấy phần tử nào có giá trị bằng X:
    - Số phép gán: Gmax = $\log_2 N + 1$
    - Số phép so sánh: Smax = $3\log_2 N + 1$
  - Trung bình:
    - Số phép gán: Gavg = $\frac{1}{2} \log_2 N + 1$
    - Số phép so sánh: Savg = $\frac{1}{2}(3\log_2 N + 3)$

# Tìm kiếm nhị phân (Binary Search)

- . **Thuật toán không đệ quy** (Non-Recursion Algorithm):
  - B1: First = 1;  Last = N
  - B2: IF (First > Last)
    - Không tìm thấy
    - Nhảy đến Kết thúc
  - B3: Mid = (First + Last)/ 2
  - B4: IF (X = M[Mid])
    - Tìm thấy tại vị trí Mid
    - Thực hiện Bkt
  - B5: IF (X < M[Mid])
    - Last = Mid – 1
    - Lặp lại B3
  - B6: IF (X > M[Mid])
    - First = Mid + 1
    - Lặp lại B2
  - Kết thúc

# Tìm kiếm nhị phân (Binary Search)

- Cài đặt thuật toán không đệ quy:
  - Hàm NRecBinarySearch thực hiện việc tìm kiếm phần tử có giá trị X trong mảng M có N phần tử đã có thứ tự tăng. Nếu tìm thấy, hàm trả về một số nguyên có giá trị từ 0 đến N-1 là vị trí tương ứng của phần tử tìm thấy. Trong trường hợp ngược lại, hàm trả về giá trị –1 (không tìm thấy). Nội dung của hàm như sau:

```
int NRecBinarySearch (T M[], int N, T X)
{ int First = 0; int Last = N – 1;
while (First <= Last)
      { int Mid = (First + Last)/2;
      if (X == M[Mid]) return(Mid);
      if (X < M[Mid]) Last = Mid – 1;
      else First = Mid + 1;
      }
return(-1); }
```

# Tìm kiếm nhị phân (Binary Search)

- **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: Gmin = 3
    - Số phép so sánh: Smin = 2
  - - Trường hợp xấu nhất khi không tìm thấy phần tử nào có giá trị bằng X:
    - Số phép gán: Gmax = 2log2N + 4
    - Số phép so sánh: Smax = 3log2N + 1
  - - Trung bình:
    - Số phép gán: Gavg = log2N + 3.5
    - Số phép so sánh: Savg = ½(3log2N + 3)

# Tìm kiếm nhị phân (Binary Search)

- **Lưu ý:**
  - Thuật toán **tìm nhị phân** chỉ có thể vận dụng trong trường hợp **dãy/mảng đã có thứ tự**. Trong trường hợp **tổng quát** chúng ta chỉ có thể áp dụng thuật toán **tìm kiếm tuần tự**.
  - 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) 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**.

BÀI TẬP