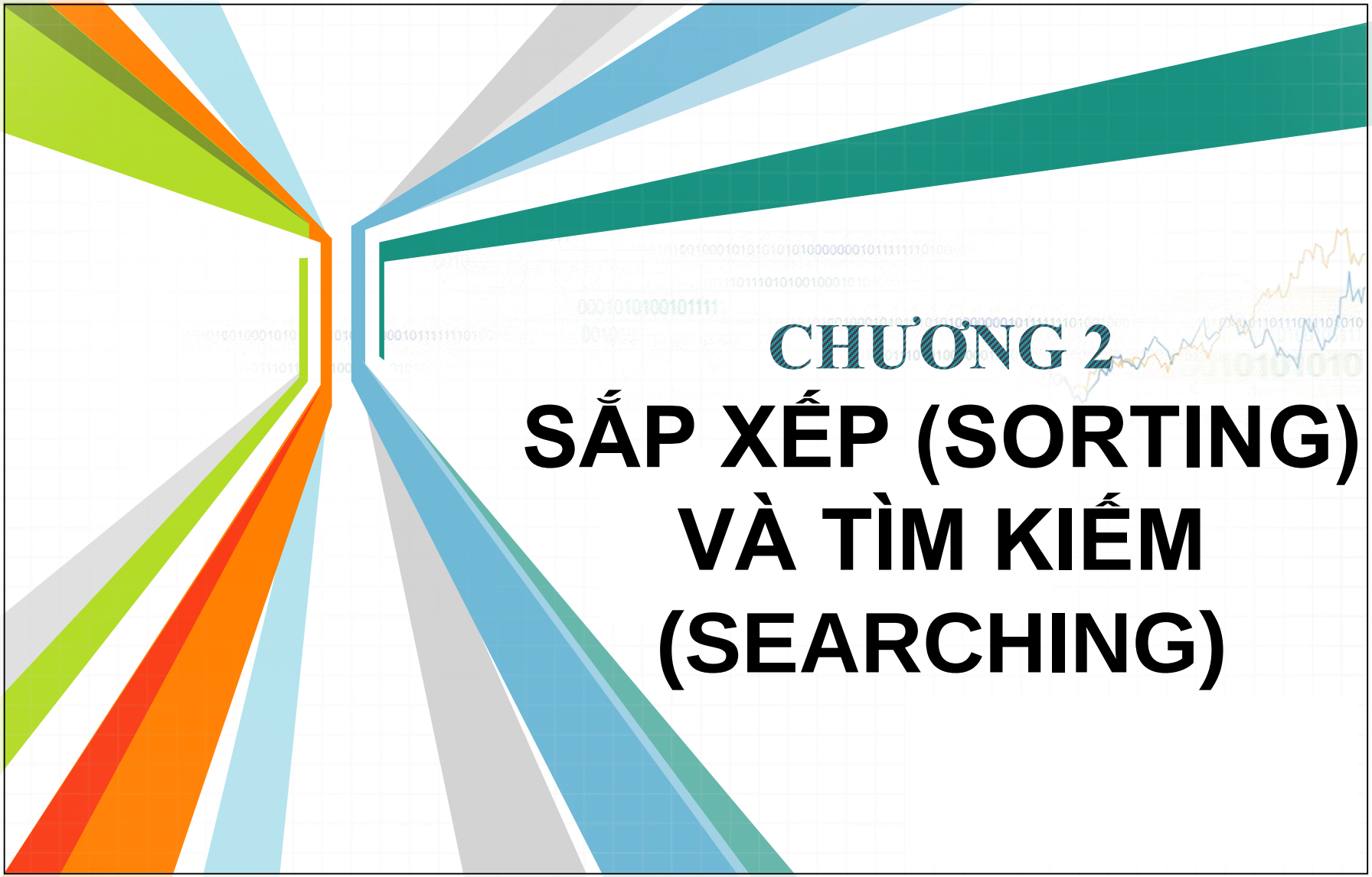


CHƯƠNG 2

SẮP XẾP (SORTING) VÀ TÌM KIẾM (SEARCHING)

Nội dung chính chương 2

2.1 Bài toán sắp xếp

2.2 Các phương pháp sắp xếp cơ bản

2.3 CTDL Heap, Sắp xếp vun đống

2.4 Tìm kiếm tuyến tính

2.5 Các vấn đề thực tế

2.1 Bài toán sắp xếp

❑ Tổng quan

- Trường (field): một đơn vị dữ liệu nào đó: tên, tuổi, số điện thoại...
- Bản ghi (record) : tập hợp các trường.
- File: tập hợp các bản ghi.

➢ Sắp xếp (sorting): xếp đặt lại các bản ghi theo một thứ tự, dựa trên một hay nhiều trường, các thông tin này gọi là khóa sắp xếp (key).

2.1 Bài toán sắp xếp

- Hầu hết các thuật toán sắp xếp: sắp xếp so sánh, sử dụng hai thao tác cơ bản là **so sánh** và **hoán vị** (swap).

❑ Sắp xếp trong (Internal sorting)

- Dữ liệu cần sắp xếp được lưu đầy đủ trong bộ nhớ trong để thực hiện thuật toán sắp xếp.

❑ Sắp xếp ngoài (External Sorting)

- Dữ liệu cần sắp xếp truy cập dữ liệu cũng mất nhiều thời gian, có kích thước quá lớn và không thể lưu vào bộ nhớ trong.

2.1 Bài toán sắp xếp

❑ Sắp xếp gián tiếp

- Các bản ghi có kích thước lớn việc hoán đổi các bản ghi rất tốn kém, để giảm chi phí người ta có thể sử dụng các phương pháp sắp xếp gián tiếp. Có thể:
  - Tạo ra một file mới chứa các trường khóa của file ban đầu.
  - Con trỏ tới
  - Chỉ số của các bản ghi ban đầu.

2.1 Bài toán sắp xếp

- Sắp xếp trên file mới này với các bản ghi có kích thước nhỏ và sau đó truy cập vào các bản ghi trong file ban đầu thông qua các con trỏ hoặc chỉ số.

Ví dụ:

Index	Dept	Last	First	Age	ID number
1	123	Smith	Jon	3	234-45-4586
2	23	Wilson	Pete	4	345-65-0697
3	2	Charles	Philip	9	508-45-6859
4	45	Shilst	Greg	8	234-45-5784

2.1 Bài toán sắp xếp

Index	Key
1	123
2	23
3	2
4	45

$\xrightarrow{Sort}$

Index	Key
3	2
2	23
4	45
1	123

Sau khi sắp xếp xong để truy cập vào các bản ghi theo thứ tự đã sắp xếp, sử dụng thứ tự được cung cấp bởi cột index (chỉ số). Trong trường hợp này là 3, 2, 4, 1.(không nhất thiết phải hoán vị các bản ghi ban đầu).

2.1 Bài toán sắp xếp

❑ **Các tiêu chuẩn đánh giá một thuật toán sắp xếp .**

- Thời gian thực hiện (run-time): số các thao tác thực hiện (so sánh và hoán đổi).
- Bộ nhớ sử dụng (Memory): dung lượng bộ nhớ cần thiết để thực hiện thuật toán ngoài dung lượng bộ nhớ sử dụng để chứa dữ liệu cần sắp xếp.
- Một vài thuật toán thuộc loại "in place" và không cần (hoặc cần một số cố định) thêm bộ nhớ cho việc thực hiện thuật toán.

2.1 Bài toán sắp xếp

- Sử dụng thêm bộ nhớ tỉ lệ thuận theo hàm tuyến tính hoặc hàm mũ với kích thước file sắp xếp.
- Bộ nhớ sử dụng càng nhỏ càng tốt mặc dù việc cân đối giữa thời gian và bộ nhớ cần thiết có thể là có lợi.
- Sự ổn định (Stability): nếu như nó có thể giữ được quan hệ thứ tự của các khóa bằng nhau (không làm thay đổi thứ tự của các khóa bằng nhau).
- Chúng ta thường lo lắng nhiều nhất là về thời gian thực hiện của thuật toán vì các thuật toán mà chúng ta bàn về thường sử dụng kích thước bộ nhớ tương đương nhau.

2.2 Các phương pháp sắp xếp cơ bản

2.2.1. Sắp xếp chọn (Selection sort)

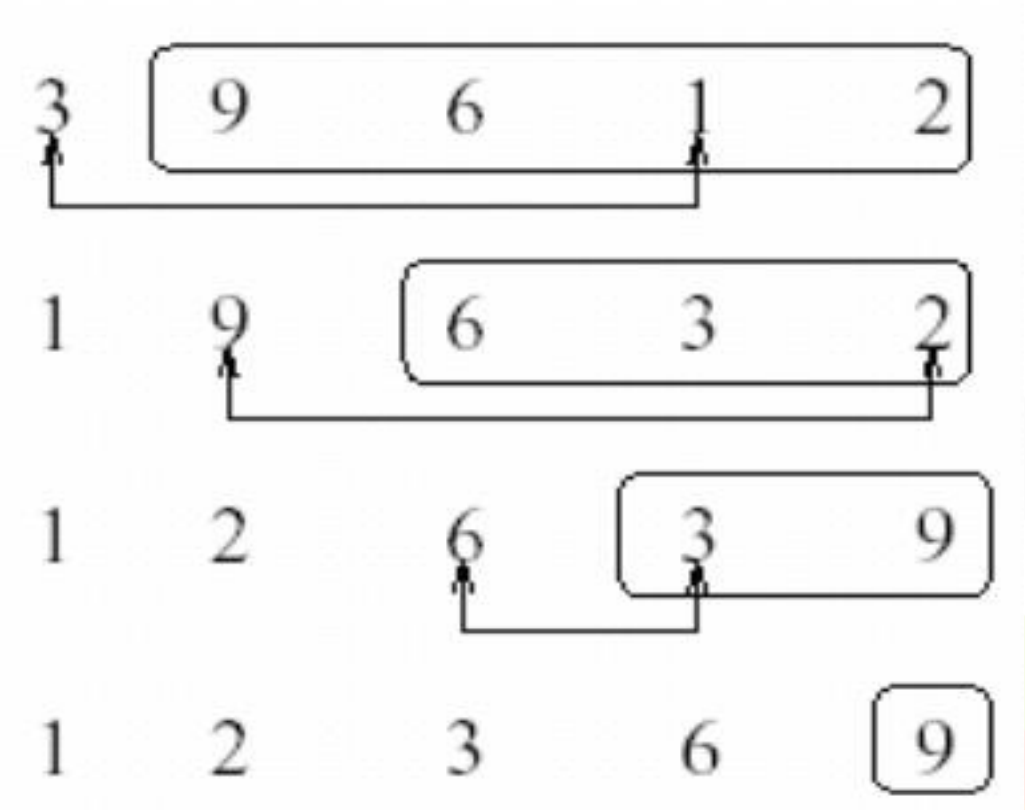
Ý tưởng

- Tại vị trí đầu tiên của dãy, chọn phần tử nhỏ nhất trong N phần tử trong dãy hiện hành ban đầu và đưa phần tử này về vị trí đầu dãy hiện hành.
- Xem dãy hiện hành chỉ còn N-1 phần tử của dãy hiện hành ban đầu. Bắt đầu từ vị trí thứ 2, chọn phần tử nhỏ nhất trong số N-1 phần tử kể trên và đưa nó về vị trí đầu của dãy gồm N-1 phần tử đang xét.
- Lặp lại quá trình trên cho dãy hiện hành... đến khi dãy hiện hành chỉ còn 1 phần tử.

2.2 Các phương pháp sắp xếp cơ bản

2.2.1. Sắp xếp chọn (Selection sort)

Minh họa



2.2 Các phương pháp sắp xếp cơ bản

2.2.1. Sắp xếp chọn (Selection sort)

```
void SelectionSort(int a[], int n )
{
        int vtmin;//dùng để lưu vị trí ptử nhỏ nhất
        for (int i=0; i<n-1 ; i++)
            {
                    vtmin = i;
                    for(int j= i+1; j<n ; j++)
                        if (a[j]< a[vtmin])
                                vtmin = j;
                    int t=a[i];
                    a[i]=a[vtmin];
                    a[vtmin]=t;
            }
}
```

2.2 Các phương pháp sắp xếp cơ bản

2.2.1. Sắp xếp chọn (Selection sort)

Phân tích thuật toán

- ***Trong mọi trường hợp***

Số phép so sánh: không thay đổi

$$\sum_{i=0}^{n-2}(n-i-1) = (n-1)+(n-2)+\cdots+1 = \frac{n(n-1)}{2}$$

$= O(n^2)$

Số phép hoán vị: n-1

Xét số phép gán: tốt nhất, xấu nhất, trung bình?

2.2.2. Sắp xếp đổi chỗ trực tiếp (Exchange sort)

Ý tưởng

- Xuất phát từ đầu dãy, tìm tất cả các nghịch thế của phần tử này, triệt tiêu chúng bằng cách hoán vị 2 phần tử trong cặp nghịch thế.
- Lặp lại xử lý trên với phần tử kế trong dãy.

2.2.2. Sắp xếp đổi chỗ trực tiếp (Exchange sort)

```
void ExchangeSort(int a[], int n)
{
    for(int  i=0; i<n-1; i++)
      for(int  j=i+1; j<n; j++)
          if(a[j]<a[i])
            {//Đổi chỗ a[i] và a[j] cho nhau
                int t=a[i];
                a[i]=a[j];
                a[j]=t;
            }
}
```

2.2.2. Sắp xếp đổi chỗ trực tiếp (Exchange sort)

Phân tích

Số phép so sánh: trong mọi trường hợp:

$$\sum_{i=0}^{n-2}(n-i-1) = (n-1)+(n-2)+\cdots+1 = \frac{n(n-1)}{2}$$

$= O(n^2)$

Số phép hoán vị:

Tốt nhất: 0 (Dãy đã sắp đúng thứ tự)

Xấu nhất: $\frac{n(n-1)}{2}$ (dãy có n-1 cặp nghịch thế)

Trung bình: $\frac{n(n-1)}{4}$

2.2.3. Sắp xếp chèn (Insertion sort)

Ý tưởng

- Ban đầu coi như phần tử đầu tiên đã sắp xếp.
- So sánh phần tử thứ 2 với phần tử đầu tiên để đổi chỗ nếu cần để sao cho 2 phần tử này được sắp xếp theo thứ tự chỉ định.
- Tìm vị trí để chèn phần tử thứ 3 của dãy hiện hành vào dãy 2 phần tử đầu đã được sắp xếp ở trên sao cho dãy vẫn được sắp xếp đúng thứ tự.
- Lặp lại liên tục các quá trình trên cho tới khi hết dãy.

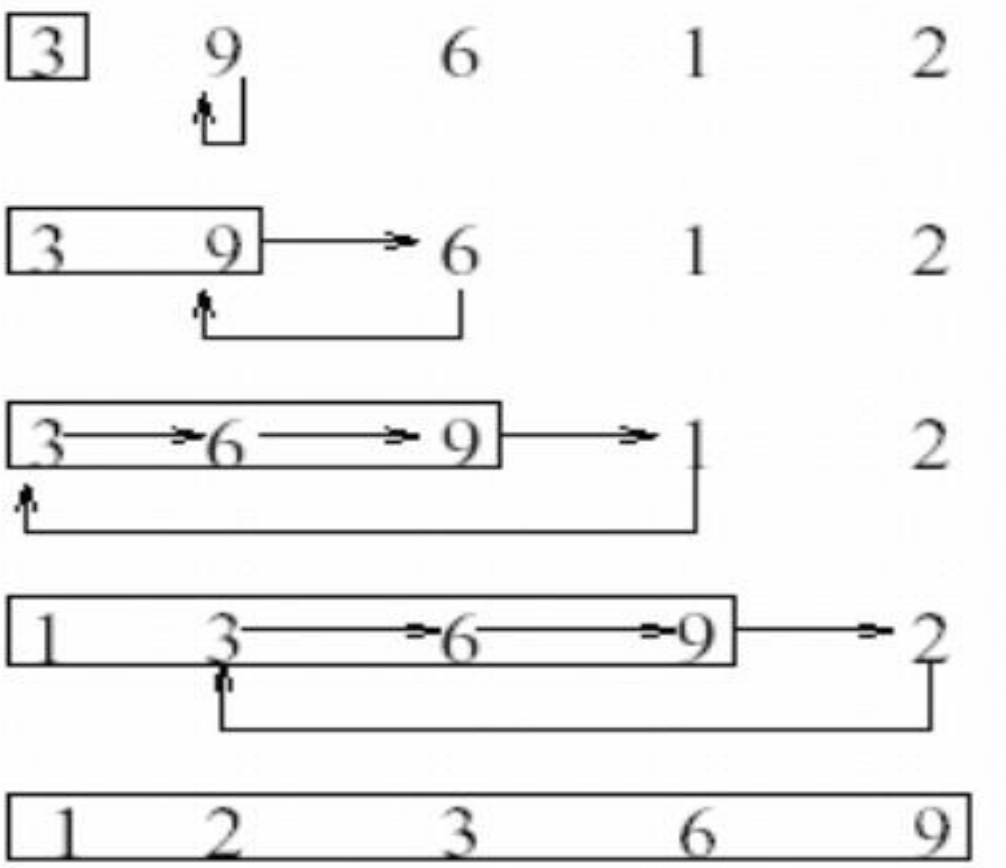
2.2.3. Sắp xếp chèn (Insertion sort)

```
void InsertionSort(int a[], int n){
    int pos, i, x;
      for(i=1; i<n; i++){
            x=a[i];
            pos=i-1;
            while(a[pos]>x &&pos>=0){
                  a[pos+1]=a[pos];
                  pos--;
            }
            a[pos+1]=x;
      }
}
```

2.2 Các phương pháp sắp xếp cơ bản

2.2.3. Sắp xếp chèn (Insertion sort)

Ví dụ:



2.2 Các phương pháp sắp xếp cơ bản

2.2.3. Sắp xếp chèn (Insertion sort)

Phân tích

- Tốt nhất khi mảng ban đầu đã có thứ tự tăng:
  Số phép so sánh: n-1=O(n)
  Số phép hoán vị : 0
- Xấu nhất khi mảng ban đầu đã có thứ tự giảm:
  Số phép so sánh: $1+2+\ldots+(n-1) = n(n-1)/2 = O(n^2)$
  Số phép hoán vị: $1+2+\ldots+(n-1) = n(n-1)/2 = O(n^2)$
- Trung bình :
  Số phép so sánh: $[n(n-1)/2 + (n-1)]/2 = O(n^2)$
  Số phép hoán vị: $n(n-1)/4$

Nếu có nhiều mảng con đã được sắp thì thuật toán chèn khá hiệu quả.

2.2 Các phương pháp sắp xếp cơ bản

2.2.4. Sắp xếp nổi bọt (Bubble sort)

Ý tưởng:

- Xuất phát từ cuối dãy, đổi chỗ các cặp phần tử kế cận để đưa phần tử nhỏ hơn trong cặp phần tử đó về vị trí đúng đầu dãy hiện hành, sau đó sẽ không xét đến nó ở bước tiếp theo, do vậy ở lần xử lý thứ i sẽ có vị trí đầu dãy là i.
- Lặp lại xử lý trên cho đến khi không còn cặp phần tử nào để xét.

2.2 Các phương pháp sắp xếp cơ bản

2.2.4. Sắp xếp nổi bọt (Bubble sort)

```
void BubbleSort(int a[], int n){
    for (int i=0; i<n-1; i++)
        for (int j=n-1; j>i; j--)
            if(a[j]<a[j-1]){
                int t=a[j];
                a[j]=a[j-1];
                a[j-1]=t;
            }
}
```

2.2 Các phương pháp sắp xếp cơ bản

2.2.4. Sắp xếp nổi bọt (Bubble sort)

Phân tích

- Tốt nhất khi mảng ban đầu đã có thứ tự tăng:
  Số phép so sánh:$1+2+\ldots+(n-1) = n(n-1)/2 = O(n^2)$
  Số phép hoán vị : 0
- Xấu nhất khi mảng ban đầu đã có thứ tự giảm:
  Số phép so sánh:$1+2+\ldots+(n-1) = n(n-1)/2 = O(n^2)$
  Số phép hoán vị: $1+2+\ldots+(n-1) = n(n-1)/2 = O(n^2)$
- Trung bình :
  Số phép so sánh:$[n(n-1)/2 + n(n-1)/2]/2 = O(n^2)$
  Số phép hoán vị: $n(n-1)/4$

Đổi chỗ hai phần tử kề nhau nên số lần đổi chỗ nhiều hơn so với đổi chỗ trực tiếp.

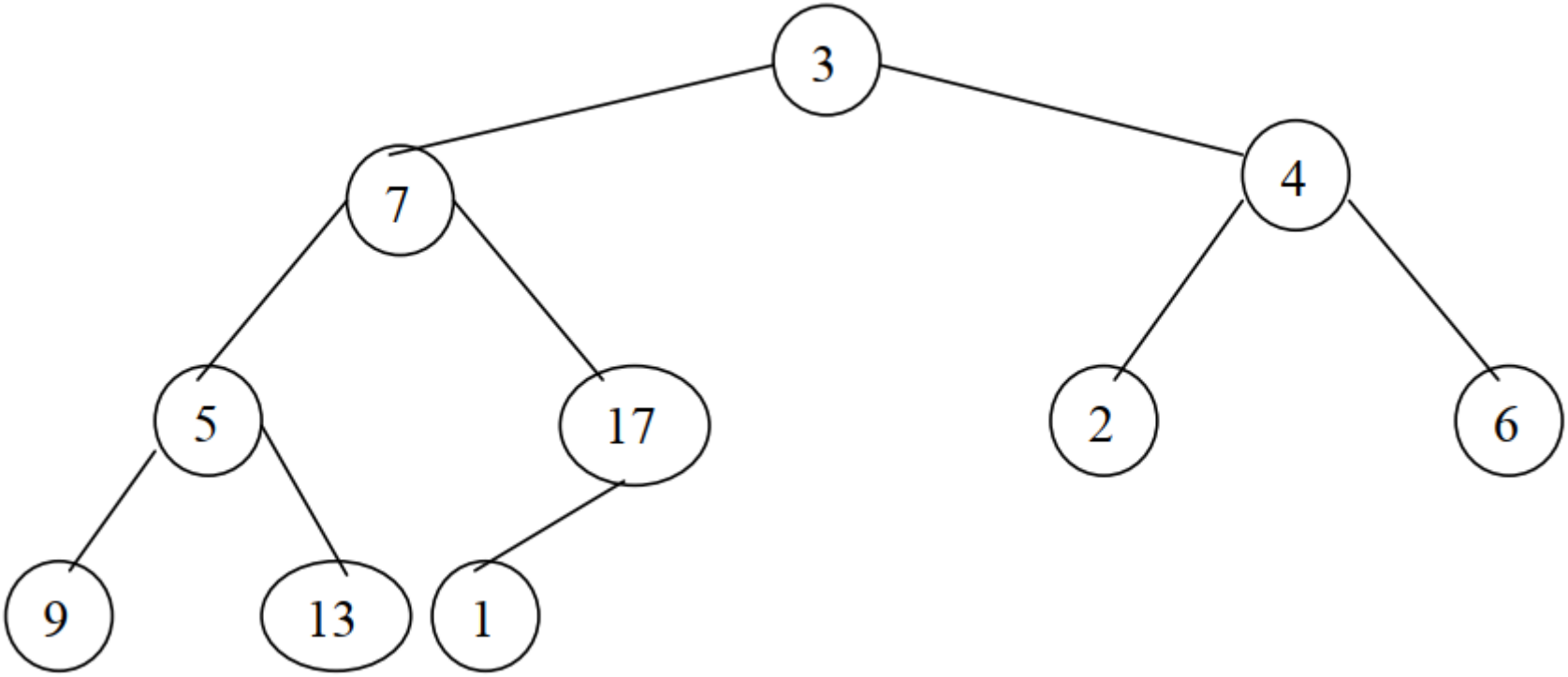
2.3 CTDL Heap, sắp xếp vun đống

2.3.1. Cấu trúc dữ liệu Heap

- Cây nhị phân đầy đủ là một cây nhị phân đầy ở tất cả các mức của cây trừ mức cuối cùng (có thể chỉ đầy về phía trái của cây).
- Heap là môt cây nhị phân đầy đủ và tại mỗi nút ta có:
  - Max heap: key (child) ≤ key(parent)
  - Min heap: key (child) >= key(parent)

2.3 CTDL Heap, sắp xếp vun đống

2.3.1. Cấu trúc dữ liệu Heap



Chiều cao của heap có n node: O(logn)

2.3 CTDL Heap, *sắp xếp vun đống*

2.3.1. Cấu trúc dữ liệu Heap

Hàm vun đống một đỉnh:

Là hàm sẽ giúp tạo cấu trúc heap từ mảng ban đầu.

- Nếu đỉnh cha có chỉ số là **index** thì:
  - Con trái có chỉ số là: **left = 2* index + 1**
  - Con phải có chỉ số là: **right = 2* index +2**

Trong đó **left** và **righrt** phải nhỏ hơn **n** (số phần tử của mảng)

- Tìm ra đỉnh lớn nhất trong 3 đỉnh: **index**, **left**, **right**
- Nếu đỉnh lớn nhất khác đỉnh ban đầu, ta tiến hành đổi chỗ. Đệ quy vun đống tại vị trí vừa đổi chỗ để vun các node tiếp theo.

2.3 CTDL Heap, sắp xếp vun đống

2.3.1. Cấu trúc dữ liệu Heap

```
void heapify(int a[], int n, int index)//n: size of heap, index: position of node current
{
    int left = 2 * index + 1;
    int right = 2 * index + 2;
    int pos = index;
    if (left<n && a[left]>a[pos])
        pos = left;
    if (right<n && a[right]>a[pos])
        pos = right;
    if (pos != index)
    {
        swap(a[index], a[pos]);
        heapify(a, n, pos);
    }
}
```

2.3 CTDL Heap, sắp xếp vun đống

2.3.1. Cấu trúc dữ liệu Heap

Phân tích:

- Gọi T(n) thời gian thực hiện heapify
- Số cây con của các con của node i nhiều nhất là 2n/3, do đó thời gian thực hiện trên một cây con có gốc tại một trong các node là con của node i là T(2n/3)
- Thời gian để chỉnh lại mối quan hệ giữa a[i], a[left], a[right]: O(1)

Suy ra: T(n)= T(2n/3)+1 => T(n) = O(logn)

Thủ tục heapify có độ phức tạp là **O(log n)**

2.3 CTDL Heap, sắp xếp vun đống

2.3.1. Cấu trúc dữ liệu Heap

Các thao tác khác trên heap

FindMin: Tìm phần tử có giá trị nhỏ nhất
Insert: Thêm một phần tử
ExtractMin: Tìm, trả lại và xóa phần tử nhỏ nhất
DecreaseKey: Giảm giá trị của một phần tử (cho trước) nào đó trong Heap
…
Có thể sử dụng heap để cài đặt hàng đợi ưu tiên

2.3 CTDL Heap, sắp xếp vun đống

2.3.1. Sắp xếp vun đống

Hàm sắp xếp vun đống (Heap sort)

Hàm heapify giúp vun một đỉnh thành một đống, bây giờ ta chỉ cần vun toàn bộ dãy ban đầu thành đống, sau đó lấy ra phần tử lớn nhất cho về cuối mảng. Lặp lại hành động này cho tới khi thu được dãy sắp xếp.

- Để vun đống, ta sẽ phải vun từ dưới lên, tức là vun từ đỉnh cha cuối cùng của heap, tức là từ vị trí n/2 -1 về 0.
- Đổi chỗ đỉnh gốc đống với phần tử cuối cùng của mảng. Như vậy phần tử lớn nhất sẽ nằm ở cuối mảng.
- Giảm số lượng phần tử (loại bỏ phần tử cuối cùng đúng vị trí).
- Lặp lại việc vun đống và lấy phần tử cho tới khi còn 1 phần tử thì dừng.
- Một mảng n phần tử sẽ phải thực hiện vun đống n lần.

2.3 CTDL Heap, sắp xếp vun đống

2.3.1. Sắp xếp vun đống

```
void heap_sort(int a[], int n)
{
    for (int i = (n - 2) / 2; i >= 0; i--)//build heap
        heapify(a, n, i);
    for (int i = n - 1; i > 0; i--)
    {
        swap(a[0], a[i]); //move root current to end
        heapify(a, i, 0);
    }
```

2.3 CTDL Heap, sắp xếp vun đống

2.3.1. Sắp xếp vun đống

Phân tích

- Thủ tục heapify có độ phức tạp là O(log n)
- Build heap có độ phức tạp là O(n)
- Heapsort: build heap thực hiện 1 lần và n-1 lần gọi tới heapify suy ra độ phức tạp của nó là: O(n + (n-1)logn) = O(n*log n).

2.4 Tìm kiếm tuyến tính

2.4.1. Bài toán tìm kiếm

- Vấn đề thuộc lĩnh vực nghiên cứu của ngành khoa học máy tính, ứng dụng rất rộng rãi trên thực tế.
- Trong môn học này, quan tâm tới bài toán tìm kiếm trên mảng, hoặc danh sách các phần tử cùng kiểu.
- Thông thường một bản ghi phân chia thành hai trường:
  - Trường lưu trữ các dữ liệu.
  - Trường để phân biệt gọi là trường khóa .
- Tập các phần tử này gọi là không gian tìm kiếm của bài toán tìm kiếm, được lưu hoàn toàn trên bộ nhớ của máy tính khi tiến hành tìm kiếm.

2.4 Tìm kiếm tuyến tính

2.4.1. Bài toán tìm kiếm

- Kết quả tìm kiếm **là vị trí của phần tử** thỏa mãn điều kiện tìm kiếm: có trường khóa bằng với một giá trị khóa cho trước (khóa tìm kiếm).
- Từ vị trí tìm thấy này chúng ta có thể truy cập tới các thông tin khác được chứa trong trường dữ liệu của phần tử tìm thấy.
- Nếu kết quả là không tìm thấy (trong trường hợp này thuật toán vẫn kết thúc thành công) thì giá trị trả về sẽ được gán cho một giá tri đặc biệt nào đó tương đương với việc không tồn tại phần tử nào có ví trí đó: chẳng hạn như -1 đối với mảng và NULL đối với danh sách liên kết.

2.4 Tìm kiếm tuyến tính

2.4.1. Bài toán tìm kiếm

Các thuật toán tìm kiếm cũng có rất nhiều :

- Các thuật toán tìm kiếm vét cạn, tìm kiếm tuần tự, tìm kiếm nhị phân.
- Những thuật toán tìm kiếm dựa trên các cấu trúc dữ liệu như các từ điển, các loại cây như cây tìm kiếm nhị phân, cây cân bằng, cây đỏ đen …

Tuy nhiên ở phần này chúng ta sẽ xem xét hai phương pháp tìm kiếm được áp dụng với cấu trúc dữ liệu mảng (dữ liệu tìm kiếm được chứa hoàn toàn trong bộ nhớ của máy tính).

2.4 Tìm kiếm tuyến tính

2.4.1. Bài toán tìm kiếm

- Điều đầu tiên mà chúng ta cần lưu ý là đối với cấu trúc mảng này, việc truy cập tới các phần tử ở các vị trí khác nhau là như nhau và dựa vào chỉ số.
- Tiếp đến chúng ta sẽ tập trung vào thuật toán nên có thể coi như mỗi phần tử chỉ có các trường khóa là các số nguyên.

2.4 Tìm kiếm tuyến tính

2.4.2. Tìm kiếm tuần tự (Sequential search)

- Duyệt qua tất cả các phần tử của mảng, trong quá trình duyệt nếu tìm thấy phần tử có khóa bằng với khóa tìm kiếm thì trả về vị trí của phần tử đó.
- Còn nếu duyệt tới hết mảng mà vẫn không có phần tử nào có khóa bằng với khóa tìm kiếm thì trả về -1 (không tìm thấy).

2.4 Tìm kiếm tuyến tính

2.4.2. Tìm kiếm tuần tự (Sequential search)

```
int sequential_search(int a[], int n, int k)
{
    int i;
        for(i=0;i<n;i++)
                if(a[i]==k)
                        return i;
     return -1;
}
```

Độ phức tạp trung bình và xấu nhất : O(n)

Tốt nhất O(1)

2.5. *Các vấn đề thực tế*

Ngoài các thuật toán đã được trình bày ở trên vẫn còn có một số thuật toán khác mà chúng ta có thể tham khảo: thuật toán sắp xếp Shell sort, sắp xếp bằng đếm Counting sort, sắp xếp cơ số Radix sort …

Bài tập

1. Chứng minh tính đúng đắn của giải thuật Selection Sort
2. Chứng minh tính đúng đắn của giải thuật Bubble Sort
3. Cài đặt các giải thuật sắp xếp để sắp xếp 1 mảng cấu trúc sinh viên theo thứ tự tăng dần:

```
struct Sinhvien{
        char fullName[33];
        int yearOld;
}
```

a) *Sắp xếp theo tuổi.*

b) *Sắp xếp theo tên (fullName bao gồm họ và tên, cần sắp xếp theo phần tên).*

Bài tập

4. Cho danh sách cấu trúc sinh viên gồm n phần tử:

```
struct Sinhvien{
        char fullName[33];
        struct birthday{
                int day; int month; int year;
        }
}
```

a) *Tìm sinh viên có tuổi lớn nhất.*

b) *Tìm sinh viên có tên bắt đầu bằng ký tự c nào đó nhập từ bàn phím.*