Chương 5
Sắp xếp

Nội dung

- Bài toán sắp xếp
- Một số thuật toán sắp xếp
  - Sắp xếp chèn – insertion sort
  - Sắp xếp lựa chọn – selection sort
  - Sắp xếp nổi bọt – bubble sort
  - Sắp xếp shell-sort
  - Sắp xếp trộn – merge sort
  - Sắp xếp nhanh – quick sort
  - Sắp xếp vun đống – heap sort

Bài toán sắp xếp

- Để tìm kiếm thông tin hiệu quả ta phải lưu giữ chúng theo một thứ tự nào đó.
  - Cách sắp xếp sách trong thư viện
  - Lưu trữ từ trong từ điển
- Sắp xếp là một trong những bài toán quan trọng trong xử lý thông tin
- Nhiều thuật toán đã được đề xuất.
- Ta chỉ xét bài toán sắp xếp trong, không xét sắp xếp ngoài

Bài toán sắp xếp

- Mỗi bản ghi có một khóa (key), ta có thể áp dụng các phép so sánh < , > , <=, >=, ==, != trên khóa.
- Sắp xếp các bản ghi bằng cách sắp xếp đối với khóa tương ứng của chúng.

```
struct node
{
    long masoSV;
    char hoten[30];
    char diachi[50];
    float diemTB;
};
```

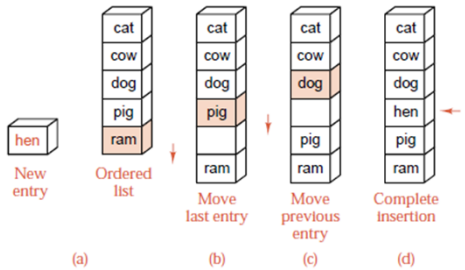
Các phương pháp sắp xếp cơ bản

- Sắp xếp chèn
- Sắp xếp lựa chọn
- Sắp xếp nổi bọt
- Sắp xếp shellsort

Sắp xếp chèn

- Sắp xếp chèn
- Cài đặt bằng mảng
- Cài đặt bằng danh sách moc nối
- Phân tích
- Bài tập

Sắp xếp chèn



- Chèn một phần tử mới vào một danh sách có thứ tự

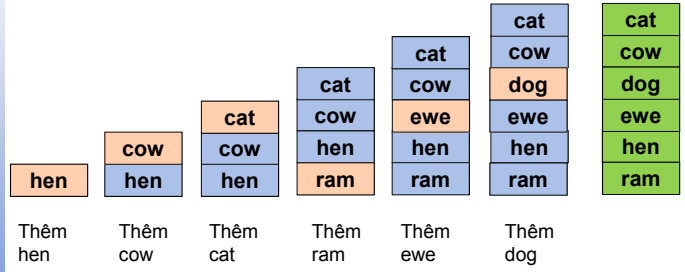
  VD. Danh sách các con vật sắp theo thứ tự bảng chữ cái

Sắp xếp chèn

- Sắp xếp bằng cách chèn:
  - Bắt đầu bằng một danh sách có thứ tự rỗng
  - Lần lượt chèn thêm các phần tử cần sắp xếp vào danh sách có thứ tự đó. (Trong quá trình chèn phải đảm bảo danh sách vẫn đúng thứ tự)
  - Kết thúc ta thu được danh sách các phần tử đã được sắp xếp theo thứ tự

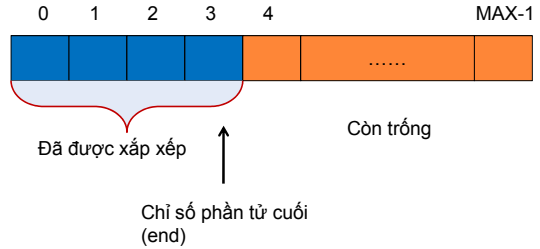
Sắp xếp chèn

- Các phần tử cần sắp xếp: hen, cow, cat, ram, ewe, dog



Sắp xếp chèn

- Danh sách lưu trữ bằng mảng



Sắp xếp chèn

- Lưu trữ bằng mảng:

```
void insert (int A[], int &end, int value)
{
    if(end==-1)
    {
        end++;
        A[end]=value;
    }
    else
    {
        int pos=0, i;
        while(A[pos]<value) pos++;
        for(i=end; i>=pos; i--) A[i+1]=A[i];
            A[pos]=value;
        end=end+1;
    }
}
```

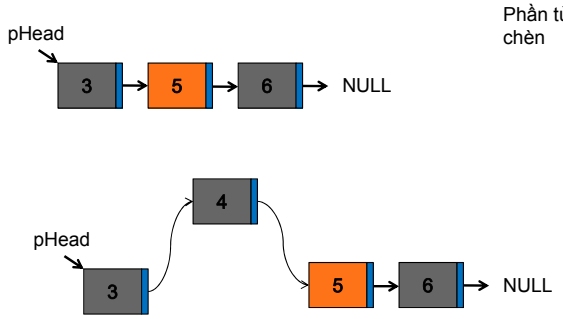
Sắp xếp chèn

- Hàm sắp xếp chèn, các phần tử cần sắp xếp được lưu ở mảng B, kết quả được lưu ở mảng A, n là số phần tử

```
void insertionSort(const int B[], int n, int A[])
{
    int end=-1;
    for(int i=0; i<n; i++)
        insert(A, end, B[i]);
}
```

Sắp xếp chèn

- Lưu trữ bằng danh sách móc nối



Sắp xếp chèn

- Định nghĩa một nút

```
typedef struct node
{
    long masoSV;
    char hoten[30];
    char diachi[50];
    float diemTB;  struct node *pNext;
} NODE;
```

- Danh sách

  NODE *list=NULL;

Sắp xếp chèn

```
int insert(NODE *&pHead, long msSV, char ht[], char dc[], float diem)
{
    NODE* ptr=(NODE*)malloc(sizeof(NODE));
    ptr->masoSV=msSV;
    strcpy(ptr->hoten,ht);
    strcpy(ptr->diachi,dc);
    ptr->diemTB = diem;

    if(pHead==NULL)
    {
        ptr->pNext=NULL;
        pHead=ptr;
    }
    else
    {
        if(pHead->masoSV >= msSV)
        {
            ptr->pNext=pHead;
            pHead=ptr;
        }
```

Sắp xếp chèn

```
        else
        {
            NODE *preQ=pHead;
            NODE *q=pHead->pNext;
            while(q!=NULL && q->masoSV < msSV)
            {
                  preQ=q;
                  q=q->pNext;
            }
            ptr->pNext=q;
            preQ->pNext=ptr;
        }
    }
}
```

Phân tích

- Thời gian thực hiện thuật toán chèn chính bằng tổng thời gian thực hiện các phép chèn vào danh sách có thứ tự.
- Trong trường hợp cài đặt bằng mảng:
  - Danh sách rỗng thì cần 1 so sánh + 2 phép gán
  - Tại thời điểm danh sách có end (0≤i<n) phần tử cần:
    - 1 phép so sánh
    - (i-k+1) phép so sánh và k phép dịch chuyển vị trí (k là vị trí cần chèn) → tổng là i
    - Hai phép gán

n là số lượng phần tử cần sắp xếp

Phân tích

Trong trường hợp cài đặt bằng mảng:

- Số lệnh và phép so sánh thực hiện :

Danh sách rỗng	O(1)
Danh sách khác rỗng	$O(n^2)$

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

Phân tích

- Trường hợp sử dụng danh sách móc nối
  - Nếu danh sách rỗng cần 5 lệnh gán + 1 so sánh + 2 gán
  - Danh sách khác rỗng (có k>0 phần tử) vòng lặp while thực hiện j lần (1≤j≤k) → trung bình là k/2
- Giá trị của k sẽ là từ 1 đến n-1 (số lượng phần tử).

Vậy

$$T(n) = O(1) + \sum_{j=1}^{n-1} \frac{j}{2} = O(1) + \frac{n(n-1)}{4} = O(n^2)$$

Phân tích

- Khi cài đặt trên mảng, số lượng các phần tử phải dịch là lớn, nhất là khi sắp xếp với số lượng phần tử lớn, làm ảnh hưởng rất nhiều tới hiệu quả của thuật toán
- Khi áp dụng trên danh sách móc nối thì không phải thực hiện dịch chuyển phần tử mà chỉ thay đổi giá trị một vài con trỏ
- Sắp xếp chèn hiệu quả khi thực hiện trên danh sách móc nối !

Bài tập

- Bài tập 1: Trường hợp tồi nhất của thuật toán sắp xếp chèn (số lượng phép so sánh phải thực hiện lớn nhất). Và trường hợp tốt nhất.
- Bài tập 2: Minh họa từng bước thuật toán sắp xếp chèn đối với các dãy sau theo thứ tự tăng dần:
  a) 26 33 35 29 19 12 22
  b) 12 19 33 26 29 35 22
  c) 12 14 36 41 60 81
  d) 81 60 41 36 14 12
  e) Tim Dot Eva Roy Tom Kim Guy Amy Jon Ann Jim Kay Ron Jan

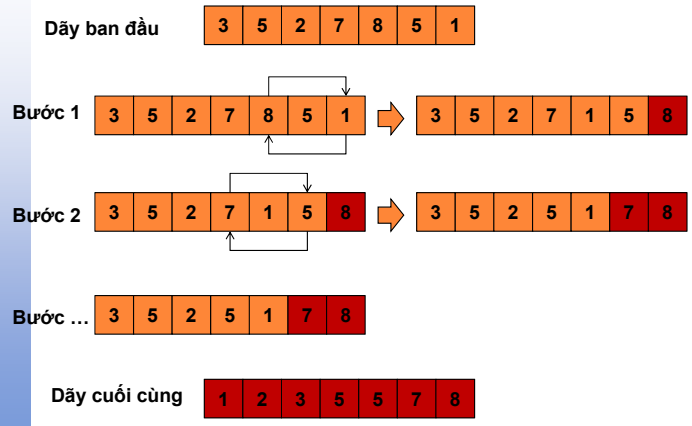
Sắp xếp lựa chọn

- Sắp xếp lựa chọn
- Cài đặt
- Phân tích
- Bài tập

Sắp xếp lựa chọn

- **Nhược điểm của thuật toán chèn**: cần rất nhiều thao tác di chuyển các bản ghi trong trường hợp lưu trữ bằng mảng.
- Kích thước bản ghi lớn, gồm nhiều trường và lưu trữ liên tục → tốc độ thực hiện sẽ bị ảnh hưởng rất lớn.
- Khắc phục bằng phương pháp sắp xếp lựa chọn

Sắp xếp lựa chọn



Sắp xếp lựa chọn

- Một phần tử:

```
typedef struct node
{
    char hoten [30];
    float diem;
} NODE;
```

Sắp xếp lựa chọn

```
void SelectionSort(NODE A[], int n)
{
     int i,j;
     int pos;
     for(i=n-1;i>=1;i--)
     {
          pos=0;
          for(j=1;j<=i;j++)
          {
             if(A[j].diem > A[pos].diem) pos=j;
          }
          swap(A,pos,i);
     }
}
```

Sắp xếp lựa chọn

- Hàm swap

```
void swap(NODE A[], int pos, int i)
{
     char tmp[30];
     float d;
     strcpy(tmp,A[pos].hoten);
     strcpy(A[pos].hoten,A[i].hoten);
     strcpy(A[i].hoten,tmp);

     d=A[pos].diem;
     A[pos].diem=A[i].diem;
     A[i].diem=d;
}
```

Phân tích

- Vòng lặp ngoài cùng với biến chạy i thực hiện n-1 lần.
- Vòng lặp bên trong
  - Lần thứ nhất thực hiện n-1 lần
  - Lần thứ hai thực hiện n-2 lần
  - ….
  - Lần thứ i thực hiện n-i lần

$$T(n) = (n-1) + (n-2) + .. + 1 = \frac{n(n-1)}{2} = O(n^2)$$

Phân tích

- Sắp xếp lựa chọn giảm số lần phải dịch chuyển dữ liệu tới mức tối thiểu.
- Cho hiệu quả tốt khi áp dụng sắp xếp với cấu trúc dữ liệu lưu trữ liên tiếp trong bộ nhớ (VD. Mảng)
- Không hiệu quả so với sắp xếp chèn khi thực hiện trên danh sách móc nối đơn!

Tại sao lại kém hiệu quả hơn sắp xếp chèn trên danh sách móc nối ?

Bài tập

- **Bài tập 1**. Minh họa sắp xếp lựa chọn trên các dãy

a) 26 33 35 29 19 12 22 theo thứ tự tăng

b) 12 19 33 26 29 35 22 theo thứ tự giảm

c) Tim Dot Eva Roy Tom Kim Guy Amy Jon Ann Jim Kay Ron Jan theo thứ tự tăng của bảng chữ cái

- **Bài tập 2**. viết lại hàm sắp xếp lựa chọn trên mảng để có thể đếm được số lần hoán đổi dữ liệu.
- **Bài tập 3**. Áp dụng hàm đếm số lần hoán đổi dữ liệu khi thực hiện sắp xếp chèn và lựa chọn trên mảng để so sánh hiệu quả của 2 phương pháp với 1 mảng số đầu vào có n phần tử (n=1000) sinh ngẫu nhiên

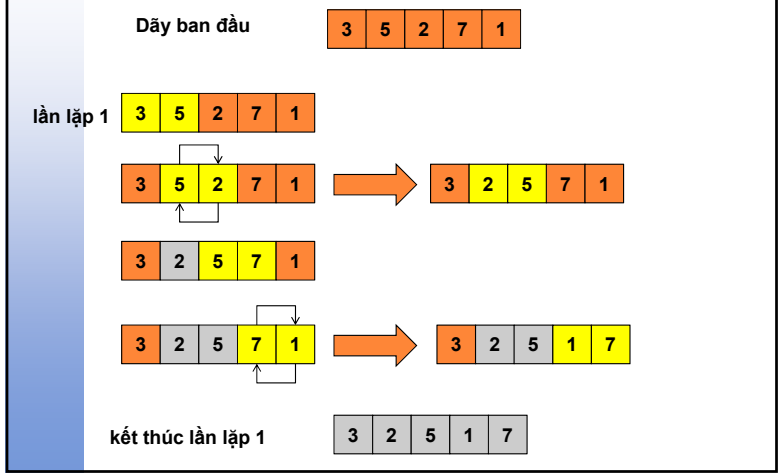
Sắp xếp nổi bọt

- Sắp xếp nổi bọt
- Cài đặt
- Phân tích
- Bài tập

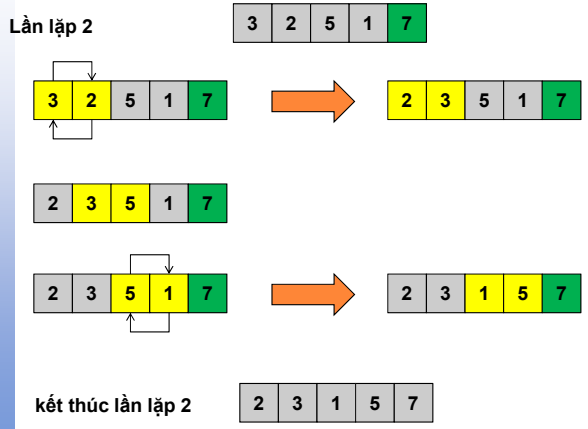
Sắp xếp nổi bọt

- Dựa trên ý tưởng trong tuyển quặng: "Quặng nặng thì chìm xuống dưới còn tạp chất nhẹ thì nổi lên trên"
- Thực hiện so sánh lần lượt các phần tử nằm kề nhau, nếu chúng không đúng thứ tự thì ta đổi chỗ chúng cho nhau.
- các phần tử có giá trị khóa lớn sẽ bị đẩy về cuối và khóa nhỏ sẽ bị đẩy lên trên (trong trường hợp sắp xếp tăng dần)

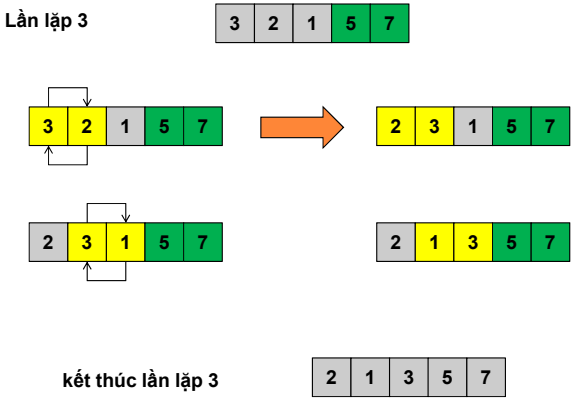
Sắp xếp nổi bọt



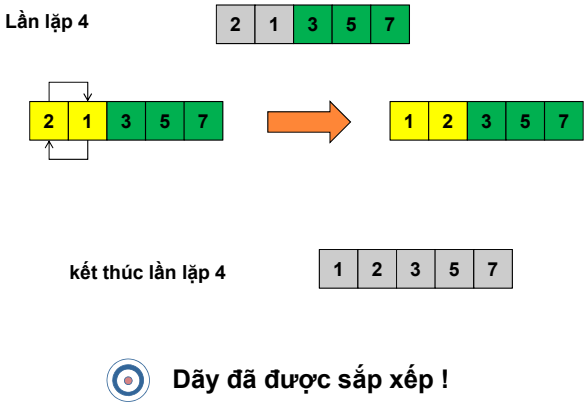
Sắp xếp nổi bọt



Sắp xếp nổi bọt



Sắp xếp nổi bọt



Sắp xếp nổi bọt

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

Phân tích

- Giống như sắp xếp lựa chọn, thời gian thực hiện cỡ $O(n^2)$
- Số lần thực hiện đổi chỗ các phần tử là nhiều hơn so với sắp xếp lựa chọn.
- không hiệu quả khi thực hiện trên danh sách với kích thước lớn

Bài tập

- **Bài tập 1**. mô phỏng hoạt động của thuật toán sắp xếp nổi bọt với các dãy sau

a) 26 33 35 29 19 12 22 theo thứ tự tăng

b) 12 19 33 26 29 35 22 theo thứ tự giảm

c) Tim Dot Eva Roy Tom Kim Guy Amy Jon Ann Jim Kay Ron Jan theo thứ tự tăng của bảng chữ cái

- **Bài tập 2**. Cải tiến hàm `BubbleSort` để có thể đếm được số lần thực hiện đổi chỗ các phần tử

Shellsort

- Shellsort
- Cài đặt
- Phân tích
- Bài tập

Shellsort

- Trong các phương pháp sắp xếp trên thì:
  - Sắp xếp lựa chọn thực hiện việc di chuyển phần tử ít nhất, tuy nhiên nó vẫn phải thực hiện nhiều phép so sánh không cần thiết
  - Sắp xếp chèn (trong trường hợp tốt nhất) phải thực hiện ít phép so sánh nhất

→ Cải tiến các phương pháp sắp xếp trên để tránh được các nhược điểm đó

Shellsort

- Sắp xếp chèn phải di chuyển nhiều phần tử vì nó chỉ di chuyển các phần tử nằm gần nhau, nếu ta di chuyển các phần tử ở xa thì hiệu quả thu được sẽ tốt hơn

➔ D. L. SHELL(1959) đề xuất phương pháp sắp xếp ***diminishing-increment sort*** (sắp xếp độ tăng giảm dần), thường gọi là shellsort

Shellsort

Original	32	95	16	82	24	66	35	19	75	54	40	43	93	68	
After 5-sort	32	35	16	68	24	40	43	19	75	54	66	95	93	82	6 swaps
After 3-sort	32	19	16	43	24	40	54	35	75	68	66	95	93	82	5 swaps
After 1-sort	16	19	24	32	35	40	43	54	66	68	75	82	93	95	15 swaps

Ban đầu các phần tử cách nhau 5 vị trí được sắp xếp, sau đó đến các phần tử cách nhau 3 vị trí, và cuối cùng là các phần tử nằm cạnh nhau (cách 1 vị trí)

Shellsort

- Ý tưởng của shellsort:
  - Ban đầu so sánh và sắp xếp các khóa ở cách nhau k vị trí
  - Sau mỗi lần sắp xếp hết các phần tử cách nhau k vị trí trong dãy ta cho k giảm đi
  - Giá trị cuối cùng của k=1, ta chỉ việc sắp xếp các phần tử kề nhau, và dãy thu được sau bước này chính là dãy đã được sắp xếp

- Để sắp xếp các phần tử ta sử dụng phương pháp sắp xếp chèn
- Dãy các số k được gọi là một dãy tăng(hoặc chuỗi khoảng cách)
  VD: 5, 3, 1 hoặc 1, 4, 10

Shellsort

```
void InsertIncSort(int A[], int n, int inc)
{
     int i,j,count,pos,tmp;
     for(i=0; i<inc; i++) //duyet het cac phan tu
     {
          count = (n-1-i)/inc; //so luong phan tu
          for(j=0; j<count; j++) //sap xep chen
          {
               tmp=A[i+(j+1)*inc];
               pos=i+(j+1)*inc;
               while(((pos-inc)>=0)&&(A[pos-inc]>tmp))
               {
                  A[pos]=A[pos-inc];
                  pos=pos-inc;
               }
               A[pos]=tmp;
          }
     }
}
```

Shellsort

```
void shellSort(int A[], int n)
{
     InsertIncSort(A,n,5);
     InsertIncSort(A,n,2);
     InsertIncSort(A,n,1);
}
```

Phân tích

- Cách chọn dãy tăng ảnh hưởng tới thời gian thực hiện của thuật toán ShellSort
  - Dãy tăng gồm các mũ của 2 là dãy tăng tồi nhất (VD. 8, 4, 2, 1): thời gian thực hiện bằng thời gian thực hiện insertionSort
  - Thời gian thực hiện tồi nhất: $O(n^2)$ với dãy tăng là ban đầu là n/2 sau đó giảm dần bằng cách chia đôi cho tới 1
  - Thời gian thực hiện tồi nhất: $O(n^{3/2})$ với dãy tăng $2^k - 1$

    $O(n^{4/3})$ với dãy tăng $9 \times 4^i - 9 \times 2^i + 1$ hoặc $4^i - 3 \times 2^i + 1$

    $O(n \log^2 n)$ với dãy tăng $2^i 3^j$
  - Dãy tốt nhất : 1, 4, 10, 23, 57, 132, 301, 701, 1750

Bài tập

- Bài tập 1: Giải thích tại sao ShellSort không dùng được với danh sách móc nối.
- Bài tập 2: Minh họa ShellSort với dãy tăng 1,4,10 khi thực hiện trên các dãy số sau:

a) 32, 95, 16, 82, 24, 66, 35, 19, 75, 54, 40, 43, 93, 68, 98, 23

b) 3, 7, 9, 0, 5, 1, 6, 8, 4, 2, 0, 6, 1, 5, 7, 3, 4, 9, 8, 2

c) 20, 18, 17, 15, 14, 13, 12, 9, 8, 5, 2, 1

Các phương pháp sắp xếp nâng cao

- Sắp xếp trộn – MergeSort
- Sắp xếp nhanh – QuickSort
- Sắp xếp đống – HeapSort

Sắp xếp trộn – mergeSort

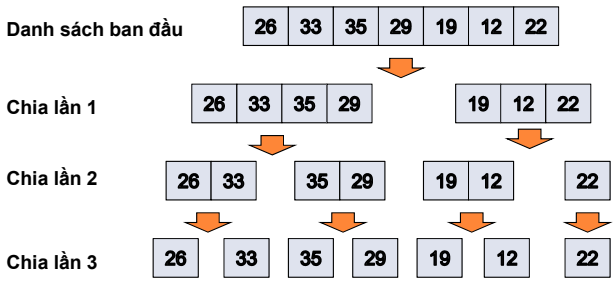
(John von Neumann 1945)

- Sắp xếp trộn
- Cài đặt
- Phân tích
- Bài tập

Sắp xếp trộn

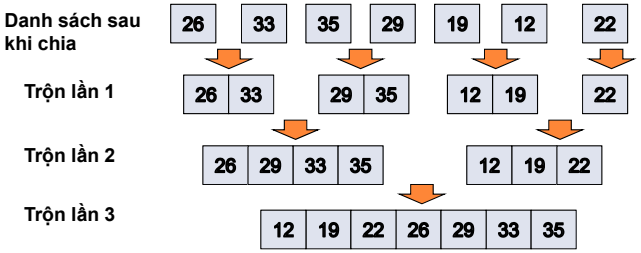
- Danh sách cần sắp xếp ban đầu là :

  26 33 35 29 19 12 22
- Ban đầu (**bước chia**) ta chia đôi danh sách thành các danh sách con, với mỗi danh sách con ta lại chia đôi cho đến khi các danh sách con chỉ có một phần tử

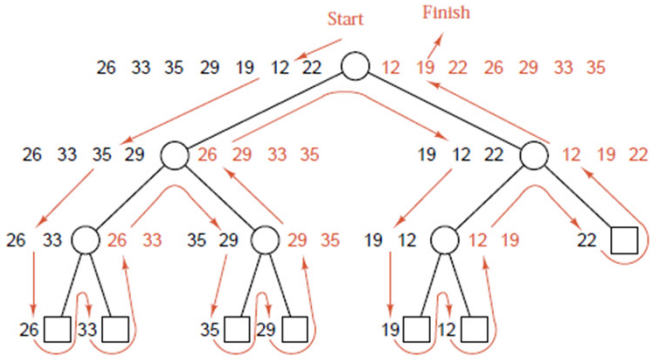


Sắp xếp trộn

- Mỗi danh sách chứa 1 phần tử là danh sách đã sắp xếp.
- Bước tiếp theo (**bước trộn**) ta lần lượt trộn các danh sách con lại với nhau để được danh sách có thứ tự.



Sắp xếp trộn



Cây đệ quy của sắp xếp trộn của danh sách
26 33 35 29 19 12 22

Sắp xếp trộn

- Cài đặt trên danh sách móc nối

```
typedef struct node
{
    int data;
    struct node *pNext;
} NODE;
```

Sắp xếp trộn

Hàm chia tách danh sách thành 2 nửa

```
void split(NODE *&first, NODE *&second)
{
     NODE *start=first;
     NODE *end=first->pNext;
     while(end->pNext!=NULL && end->pNext->pNext!=NULL)
     {
         start = start->pNext;
         end = end->pNext->pNext;
     }
     second = start->pNext;
     start->pNext=NULL;
}
```

Sắp xếp trộn

```
void merge(NODE *first, NODE *second, NODE *&third)
{
     NODE *ptr;
     third=(NODE*)malloc(sizeof(NODE));
     third->pNext=NULL;
     ptr=third;

     NODE *p=first;
     NODE *q=second;
     while(p!=NULL && q!=NULL)
     {
         if(p->data <= q->data)
         {
             ptr->pNext=p;
             p=p->pNext;
             ptr=ptr->pNext;
             ptr->pNext=NULL;
         }
```

Sắp xếp trộn

```
            else
            {
                ptr->pNext=q;
                q=q->pNext;
                ptr=ptr->pNext;
                ptr->pNext=NULL;
            }
        }
        if(p==NULL) ptr->pNext=q;
        else ptr->pNext=p;
        ptr=third;
        third=third->pNext;
        free(ptr);
}
```

Sắp xếp trộn

```
void MergeSort(NODE *&pHead)
{
    if(pHead!=NULL && pHead->pNext!=NULL)
    {
        NODE *second=NULL;
        split(pHead,second);
        MergeSort(pHead);
        MergeSort(second);
        NODE *third=NULL;
        merge(pHead,second,third);
        pHead=third;
    }
}
```

Sắp xếp trộn

- Phân tích thao tác chia:
  - Mỗi lần chia đôi ta phải duyệt hết danh sách
  - Danh sách ban đầu có n phần tử thì cần thực hiện n lần duyệt.
  - Lần chia thứ nhất danh sách chia thành 2 danh sách con n/2 phần tử. Mỗi danh sách con khi chia nhỏ ta cũng phải duyệt n/2 lần → tổng là n/2+ n/2 =n lần
  - ….
  - Tổng cộng mỗi lần chia phải thực hiện n thao tác duyệt, mà có log*n* lần chia vậy độ phức tạp của thao tác chia là O(*n*log*n*)

Sắp xếp trộn

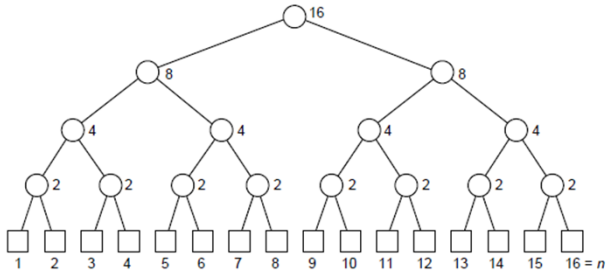


Figure 8.14. Lengths of sublist merges

Sắp xếp trộn

- Phân tích: trong thao tác kết hợp
  - Đánh giá thời gian thực hiện thuật toán thông qua số lượng phép so sánh.
  - Tại một mức số lượng phép so sánh tối đa không thể vượt số lượng phần tử của danh sách con tại mức đó
  - Mức nhỏ nhất là log*n* thì có n danh sách con, để kết hợp thành n/2 danh sách cần n nhiều nhất phép so sánh
  - ….
  - Tổng cộng mỗi mức phải thực hiện nhiều nhất n phép so sánh, mà có logn mức nên thời gian thực hiện thao tác kết hợp cỡ O(*n*log*n*)

Sắp xếp trộn

- Độ phức tạp tính toán trung bình của sắp xếp trộn là O(*n*log*n*)
- Trong trường hợp cài đặt với danh sách liên tục (VD. Mảng) thì ta gặp vấn đề khó khăn với thao tác kết hợp là:
  - Phải sử dụng thêm bộ nhớ phụ (dùng mảng phụ để tránh phải dịch chuyển các phần tử)
  - Nếu không sử dụng bộ nhớ phụ thì thời gian thực hiện là $O(n^2)$ – chi phí cho việc dịch các phần tử trong mảng
  - Chương trình viết phức tạp hơn so với dùng danh sách móc nối

Bài tập

- Bài tập 1. Minh họa MergeSort (vẽ cây đệ quy) cho các danh sách sau
  a) Tim Dot Eva Roy Tom Kim Guy Amy Jon Ann Jim Kay Ron Jan
  b) 32, 95, 16, 82, 24, 66, 35, 19, 75, 54, 40, 43, 93, 68
- Bài tập 2. Cài đặt MergerSort cho trường hợp sắp xếp trên mảng. Gợi ý: sử dụng thêm một danh sách phụ để lưu trữ.

Sắp xếp nhanh – QuickSort

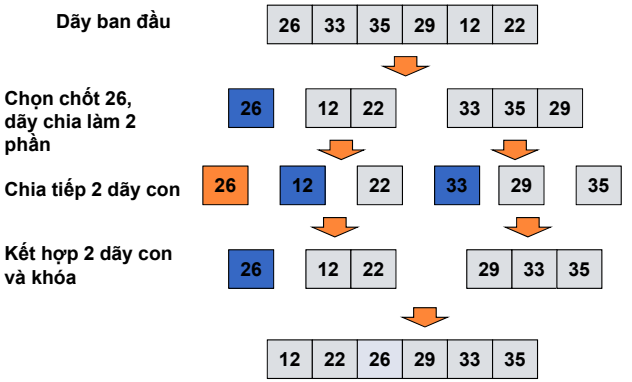
(C. A. R. Hoare 1962)

- Sắp xếp nhanh
- Cài đặt
- Phân tích
- Bài tập

Sắp xếp nhanh

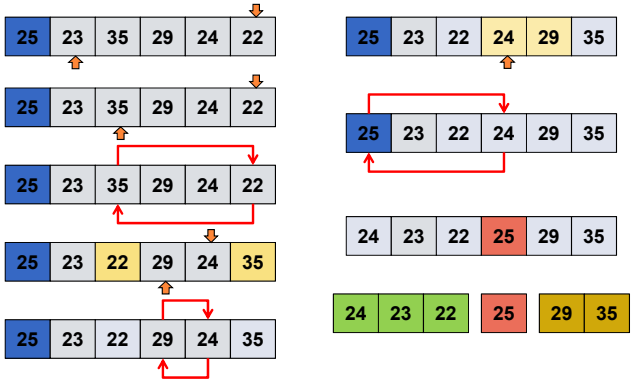
- **Ý tưởng**: giống như sắp xếp trộn là chia danh sách thành 2 phần, tuy nhiên trong sắp xếp nhanh ý tưởng chia khác một chút.
  - Một phần tử trong danh sách được chọn làm phần tử '**chốt**' (thường là phần tử đầu danh sách).
  - Danh sách sau đó được chia thành 2 phần, phần đầu gồm các phần tử nhỏ hơn hoặc bằng chốt, phần còn lại là các phần tử lớn hơn chốt.
  - Sau đó hai danh sách con lại được chọn chốt và chia tiếp cho đến khi danh sách con chỉ có 1 phần tử
  - Cuối cùng ta kết hợp 2 danh sách con và phần tử chốt từng mức lại ta được danh sách đã sắp xếp

Sắp xếp nhanh



Sắp xếp nhanh

- Sắp xếp nhanh trên mảng: Cách chia danh sách với phần tử chốt



Sắp xếp nhanh

```
void qSort(int A[], int start, int end)
{
     //chon phan tu dau lam chot
     if(start<end) //co nhieu hon 1 phan tu
     {
          int p,q,tmp;
          p=start; q=end;
          while(p<q)
          {
               while(A[p]<=A[start]) p++;
               while(A[q]>A[start]) q--;
```

Sắp xếp nhanh

```
            if(p<q)
            {
                tmp=A[p];
                A[p]=A[q];
                A[q]=tmp;
            }
        }

        tmp=A[start];
        A[start]=A[q];
        A[q]=tmp;

        //goi de quy
        qSort(A,start,q-1);
        qSort(A,q+1,end);
    }
}
```

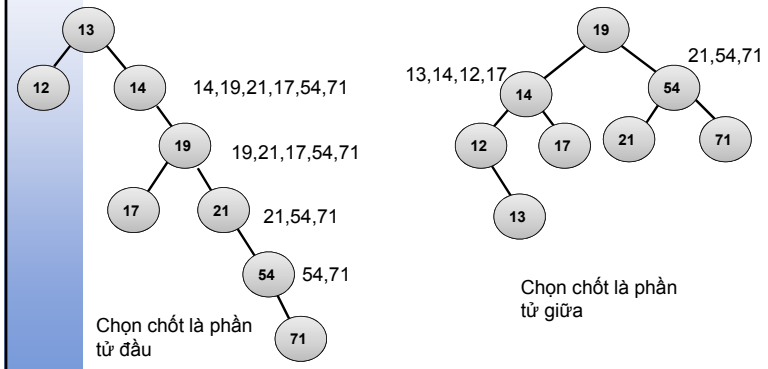
Sắp xếp nhanh

- Hàm thực hiện QuickSort

```
void quickSort(int A[], int n)
{
    qSort(A,0,n-1);
}
```

Phân tích

- Cách chọn chốt ảnh hưởng đến việc chia danh sách. Ví dụ xét danh sách : 13, 14, 12, 19, 21, 17, 54, 71



Phân tích

- Chọn chốt:
  - Cách chọn tồi nhất: chọn phần tử có khóa lớn nhất, hoặc nhỏ nhất là chốt
  - Chọn chốt tốt nhất : chọn khóa mà chia dãy thành 2 phần đều nhau
- Số lượng phép so sánh và đổi chỗ
  - C(n) số lượng phép so sánh

$$C(n) = n - 1 + C(r) + C(n - r - 1) \qquad C(1) = C(0) = 0$$

  - S(n) số lượng phép đổi chỗ

$$S(n) = p + 1 + S(p - 1) + S(n - p) \qquad n \geq 2$$

Phân tích

- Trong trường hợp tồi nhất

$$C(n) = n-1+C(0)+C(n-1) = n-1+C(n-1)$$

$$C(n) = n-1+n-2+..+1+0 = \frac{n(n+1)}{2}$$

$$S(n) = n+1+S(n-1)$$

$$S(n) = (n+1)+n+..+3 = \frac{(n+1)(n+2)}{2}-3 \qquad S(2)=3$$

Phân tích

- Trong trường hợp trung bình

$$C(n) \approx 1.39n\log n + O(n)$$

$$S(n) = \ 0.69n\log n + O(n)$$

- Độ phức tạp của thuật toán QuickSort
  - Trong trường hợp tồi nhất $O(n^2)$
  - Trường hợp trung bình $O(n\log n)$

Phân tích

- QuickSort tồi hơn sắp xếp chèn với mảng kích thước nhỏ, nên chỉ dùng khi mảng có kích thước lớn
- Hiệu quả phụ thuộc vào cách chọn chốt, có nhiều phương pháp chọn chốt : chốt đầu, giữa, ngẫu nhiên, trung vị của 3 khóa …
- So với sắp xếp trộn-mergerSort (cài đặt trên mảng) thì QuickSort cho hiệu quả tốt hơn
  - Không phải sử dụng bộ nhớ phụ
  - Không cần thực hiện thao tác trộn
- Nhưng kém hơn so với mergesort cài đặt trên danh sách móc nối tương ứng

Bài tập

- Bài tập 1. Minh họa QuickSort cho các danh sách sau (với 2 cách chọn chốt là đầu và chốt giữa)
  a) Tim Dot Eva Roy Tom Kim Guy Amy Jon Ann Jim Kay Ron Jan
  b) 32, 95, 16, 82, 24, 66, 35, 19, 75, 54, 40, 43, 93, 68
  c) 8, 8, 8, 8, 8 ,8 (các phần tử bằng nhau)
- Bài tập 2. Cải tiến hàm quickSort để có thể đếm được số phép so sánh và dịch chuyển phần tử.

Bài tập

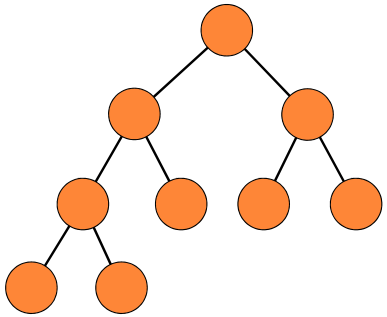
- Bài tập 3. Cải tiến thuật toán QuickSort để có thể tìm được phần tử lớn nhất thứ k trong danh sách gồm n phần tử.

Sắp xếp vun đống HeapSort

- Đống
- Sắp xếp vun đống
- Cài đặt
- Phân tích
- Bài tập

HeapSort

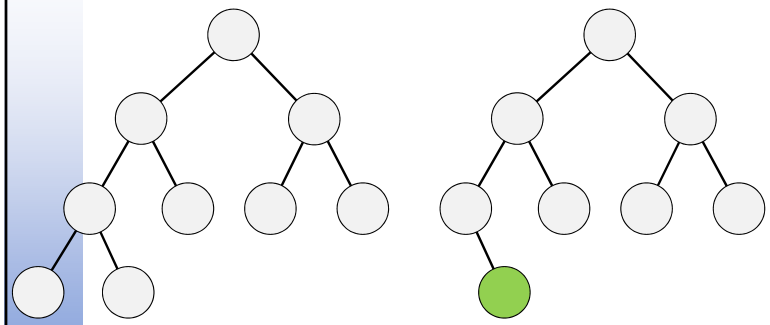
- **2-tree** : là cây nhị phân mà các nút trong (internal node) đều có đầy đủ 2 con



HeapSort

- Định nghĩa đống – Heap:
  - là 2-tree cân bằng,
  - lệch trái – left justified (nút lá xếp đầy phía bên trái trước)
  - Không nút con nào có giá trị lớn hơn giá trị nút cha của nó
- Khái niệm Heap này khác với khái niệm vùng nhớ rỗi HEAP trong tổ chức bộ nhớ của chương trình
- Heap không phải là một cây nhị phân tìm kiếm

HeapSort

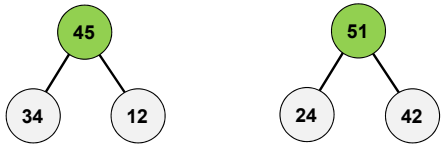


Cây lệch trái - left justified

Cây không lệch trái

HeapSort

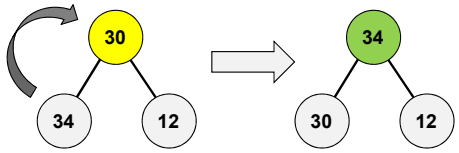
- Một nút được gọi là có ***thuộc tính Heap*** nếu giá trị tại nút đó lớn hơn hoặc bằng giá trị tại các nút con của nó



- ❑ Các nút lá trên cây đều có thuộc tính Heap
- ❑ Nếu ***mọi nút*** trên cây nhị phân đều có thuộc tính Heap thì ***cây nhị phân là Heap***

HeapSort

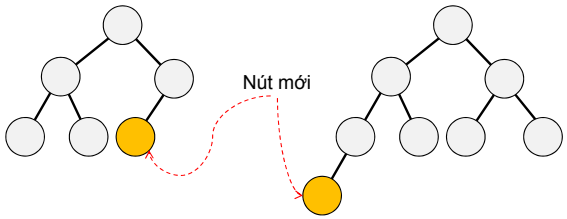
- Tại một nút không có thuộc tính Heap, nếu ta thay thế nó bằng nút con lớn hơn thì nút đó sẽ có thuộc tính Heap (tao tác ***sifting***)



- Nút con bị thay có thể mất thuộc tính Heap

HeapSort

- Xây dựng Heap – buildHeap
  - (i)Thêm nút vào cây rỗng → cây là Heap
  - (ii)Thêm nút vào bên phải nhất của mức sâu nhất trên cây. Nếu mức sâu nhất đã đầy → tạo một mức mới

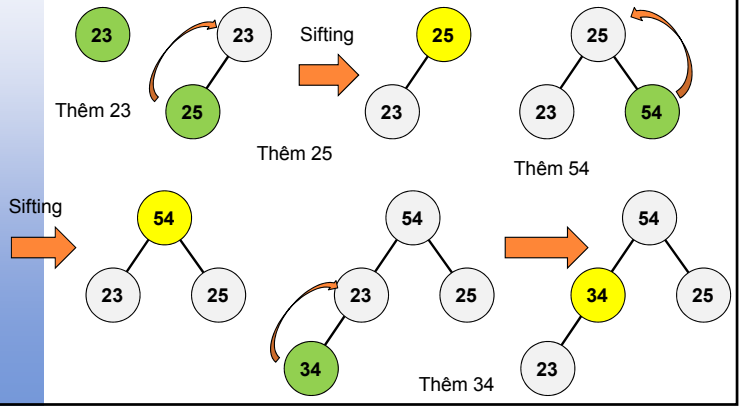


HeapSort

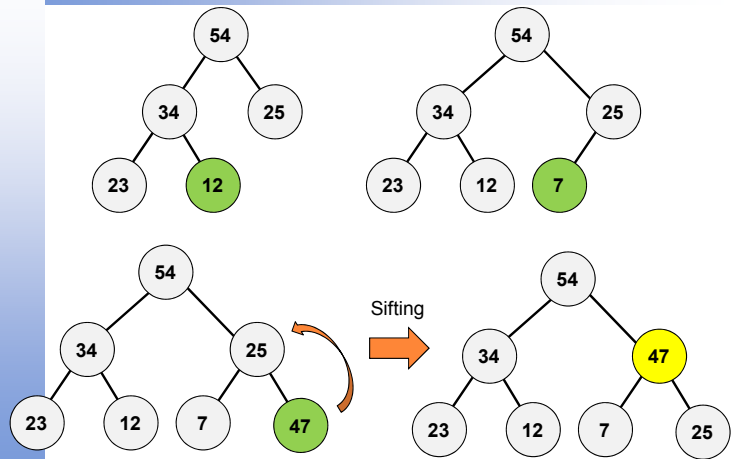
- Trong trường hợp thêm nút (ii) có thể nút mới thêm phá vỡ thuộc tính Heap của các nút thuộc nhánh mà nó thêm vào → cần thực hiện ***sift*** từ vị trí thêm vào trở về gốc để điều chỉnh lại.
- Nếu các nút nằm trên đường trở về gốc vẫn tiếp tục vi phạm thuộc tính Heap → điều chỉnh tiếp cho tới khi hết nút vi phạm

HeapSort

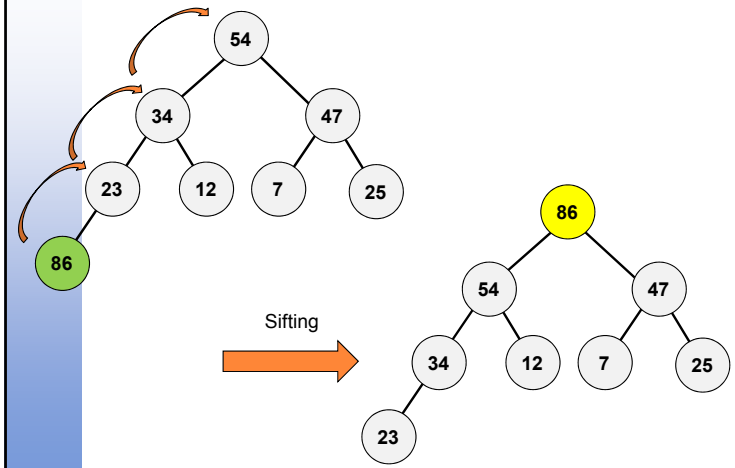
- Ví dụ. Vẽ Heap thu được khi thêm lần lượt các khóa sau vào Heap ban đầu rỗng: 23, 25, 54, 34, 12, 7, 47, 86, 56



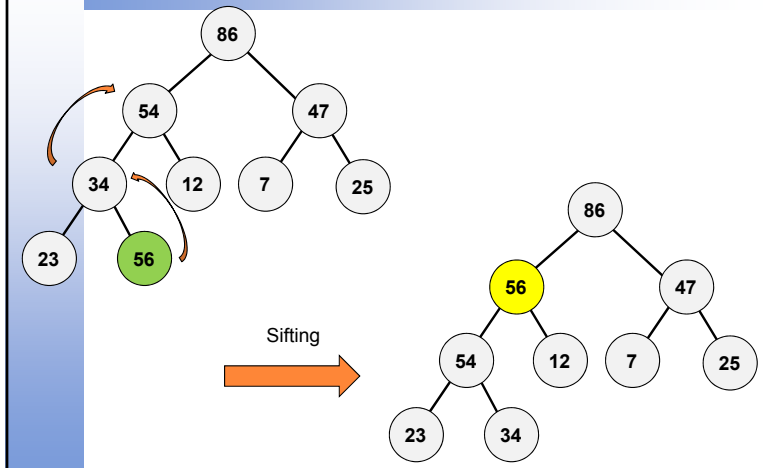
HeapSort



HeapSort

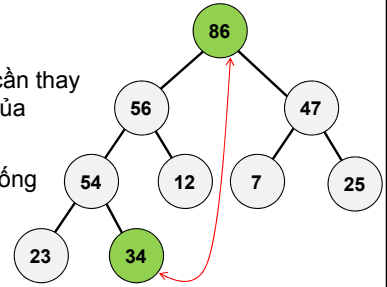


HeapSort

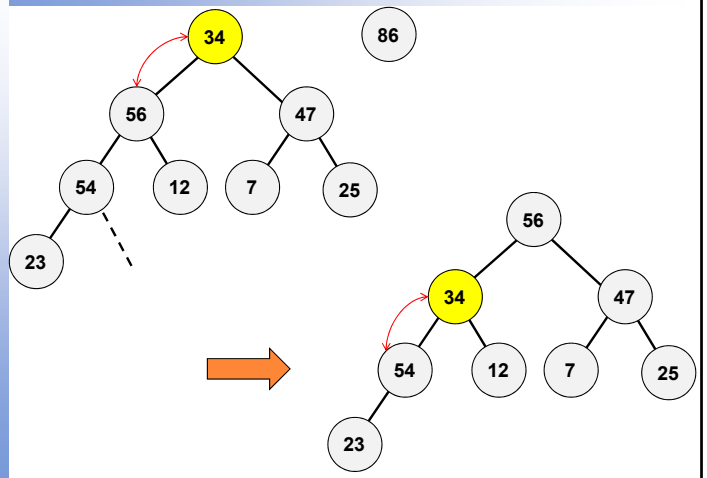


HeapSort

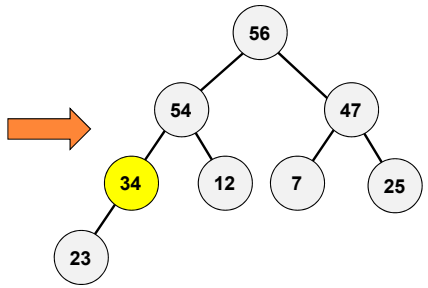
- Thao tác Sift không làm thay đổi hình dáng của cây
- Việc tạo Heap không có nghĩa là sắp xếp
- Sau khi thêm nút và Sift, nút ở gốc là nút có giá trị lớn nhất
- Nếu lấy nút ở gốc thì ta cần thay bằng nút bên phải nhất của mức sâu nhất trên cây
- Cần phải điều chỉnh lại đống sau khi thay thế nút gốc



HeapSort



HeapSort



Sau khi thực hiện sift, ta lại thu được Heap

- Giá trị lớn nhất tiếp theo lại ở gốc!

HeapSort

- Ý tưởng sắp xếp dùng Heap:
  - Thêm lần lượt các phần tử trong dãy vào Heap ban đầu rỗng
  - Lần lượt lấy các phần tử ở gốc sau đó thực hiện điều chỉnh lại để thu được Heap
- Áp dụng ý tưởng này trên mảng?

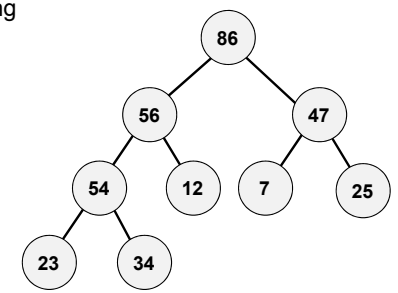
HeapSort

- Heap là 2-tree cân bằng, lệch trái (left justified) nên ta có thể biểu diễn bằng mảng

Nút gốc tương ứng với phần tử có chỉ số 0

Nút ở ô chỉ số k có con trái là $2 * k + 1$ và con phải là $2 * k + 2$

Phần tử cuối cùng của mảng là phần tử phải nhất nằm mức sâu nhất của Heap

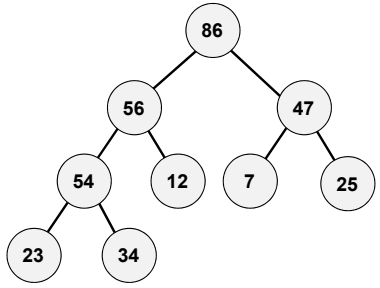


0	1	2	3	4	5	6	7	8
86	56	47	54	12	7	25	23	34

HeapSort

- HeapSort
  - Biểu diễn Heap bằng mảng
  - Thực hiện xây dựng Heap
  - Trong khi mảng còn khác rỗng
    - Lấy và thay thế phần tử gốc
    - Xây dựng lại Heap

HeapSort

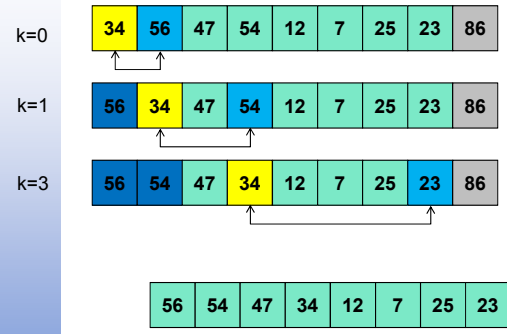


0	1	2	3	4	5	6	7	8
86	56	47	54	12	7	25	23	34

34	56	47	54	12	7	25	23	86

Hoán đổi phần tử đầu và phần tử cuối

HeapSort



Loại bỏ phần tử cuối và xây lại Heap

Heap thu được sau khi xây dựng lại

Tiếp tục quá trình lấy phần tử gốc, đổi chỗ và xây dựng lại cho tới khi mảng rỗng

HeapSort

```
void siftUp(int Heap[], int nodeIndex)
{
   if(nodeIndex==0) return;
   int k = (nodeIndex-1)/2;
   if(Heap[nodeIndex]>Heap[k])
   {
      int tmp = Heap[k];
      Heap[k] = Heap[nodeIndex];
      Heap[nodeIndex] = tmp;
      siftUp(Heap,k);
   }
}
```

HeapSort

```
void siftDown(int Heap[], int nodeIndex, int maxEle)
{
     int leftChildIndex, rightChildIndex, tmp;
     leftChildIndex = nodeIndex*2 + 1;
     rightChildIndex = nodeIndex*2 + 2;
     if(leftChildIndex > maxEle-1) return; //at leaf node
     else if(rightChildIndex > maxEle-1) //has only left child
     {
        if(Heap[nodeIndex]<Heap[leftChildIndex])
        {
            tmp = Heap[leftChildIndex];
            Heap[leftChildIndex] = Heap[nodeIndex];
            Heap[nodeIndex] = tmp;
        }
     }
```

```
     else //has two children
     {
        if(Heap[leftChildIndex]>=Heap[rightChildIndex])
           if(Heap[nodeIndex]<Heap[leftChildIndex])
           {
              tmp = Heap[leftChildIndex];
              Heap[leftChildIndex] = Heap[nodeIndex];
              Heap[nodeIndex] = tmp;
              siftDown(Heap,leftChildIndex,maxEle);
           }
        else
           if(Heap[nodeIndex]<Heap[rightChildIndex])
           {
              tmp = Heap[rightChildIndex];
              Heap[rightChildIndex] = Heap[nodeIndex];
              Heap[nodeIndex] = tmp;
              siftDown(Heap,rightChildIndex,maxEle);
           }
     }
}
```

HeapSort

- Phân tích HeapSort:
  - Tạo Heap từ n phần tử ban đầu có chi phí $n * O(\log n)$
  - Vòng lặp loại phần tử gốc và xây dựng lại Heap có thời gian cỡ $(n-1) * O(\log n)$
  - Do đó tổng thời gian thực hiện : $O(n \log n)$

HeapSort

- So sánh với Quicksort
  - Heapsort luôn có thời gian thực hiện trong tất cả các trường hợp là $O(n \log n)$
  - Quicksort thường là $O(n \log n)$ tuy nhiên trong một số trường hợp đặc biệt có thể tới $O(n^2)$
  - Quicksort thường nhanh hơn, tuy nhiên trong một số trường hợp đặc biệt thì Heapsort nhanh hơn
  - Heapsort cho đảm bảo thời gian thực hiện tốt hơn