Chương 2

Chiến lược chia-để-trị (Divide-and-conquer)

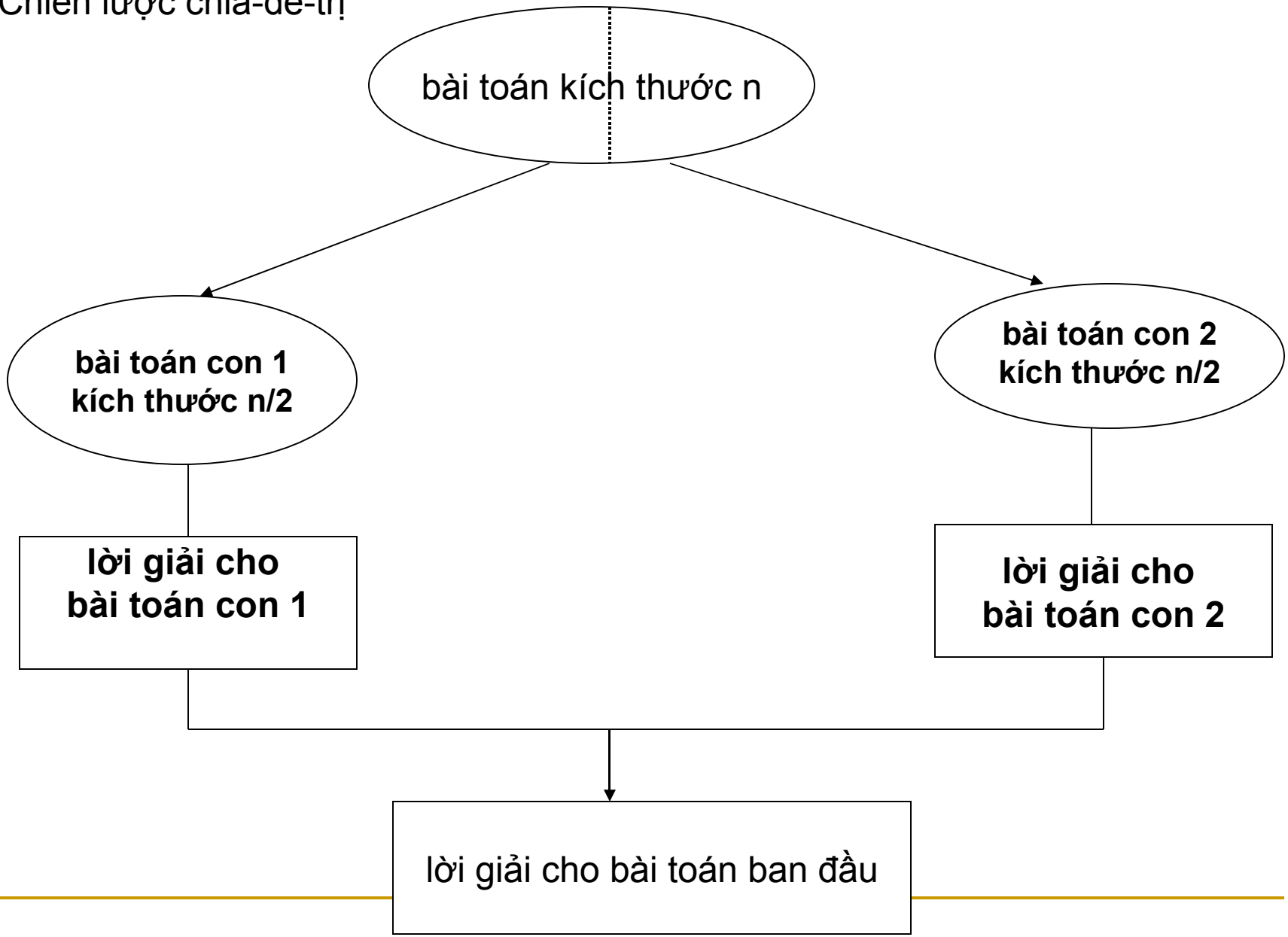
Nội dung

1. Chiến lược chia để trị
2. Quicksort
3. Xếp thứ tự bằng phương pháp trộn
4. Xếp thứ tự ngoại
5. Cây tìm kiếm nhị phân

Chiến lược chia-để-trị

- **Là chiến lược thiết kế giải thuật nổi tiếng nhất.**
- **Các giải thuật chia-để-trị thường tiến hành theo các bước sau:**
  - Thể hiện của bài toán được chia làm những thể hiện nhỏ hơn.
  - Những thể hiện nhỏ hơn này được giải quyết (thường là đệ quy, mặc dù đôi khi không cần đệ quy).
  - Những lời giải đạt được từ những thể hiện nhỏ hơn phối hợp lại làm thành lời giải của bài toán ban đầu.
- **Tìm kiếm bằng p.p. chia đôi (binary search) là một thí dụ của chiến lược chia-để-trị.**
- **Sơ đồ sau mô tả một chiến lược chia-để-trị mà trong đó chia bài toán thành hai bài toán nhỏ hơn. Đây là trường hợp phổ biến nhất của chiến lược này.**

Chiến lược chia-để-trị



2. Giải thuật Quick sort

Giải thuật căn bản của Quick sort được phát minh năm 1960 bởi C. A. R. Hoare.

Quicksort thể hiện tinh thần thiết kế giải thuật theo lối "Chia để trị" (divide-and-conquer).
Quicksort được ưa chuộng vì nó không quá khó để hiện thực hóa.
Quicksort chỉ đòi hỏi khoảng chừng NlgN thao tác căn bản để sắp thứ tự N phần tử.

Nhược điểm của Quick sort gồm:
- **Nó là một giải thuật đệ quy**
- **Nó cần khoảng N^2 thao tác căn bản trong trường hợp xấu nhất**
- **Nó dễ bị lỗi khi lập trình (fragile).**

Giải thuật căn bản của Quicksort

Quicksort là một phương pháp xếp thứ tự theo kiểu "chia để trị". Nó thực hiện bằng cách phân hoạch một tập tin thành hai phần và sắp thứ tự mỗi phần một cách độc lập với nhau.

Giải thuật có cấu trúc như sau:

```
procedure quicksort1(left,right:integer);
var i: integer;
begin
        if right > left then
        begin
                i:= partition(left,right);
                quicksort(left,i-1);
                quicksort(i+1,right);
        end;
end;
```

Phân hoạch

Phần then chốt của Quicksort là thủ tục phân hoạch (partition), mà sắp xếp lại mảng sao cho thỏa mãn 3 điều kiện sau:

i) phần tử *a[i]* được đưa về vị trí đúng đắn của nó, với một giá trị *i* nào đó,

ii) tất cả những phần tử trong nhóm *a[left]*, ..., *a[i-1]* thì nhỏ hơn hay bằng *a[i]*

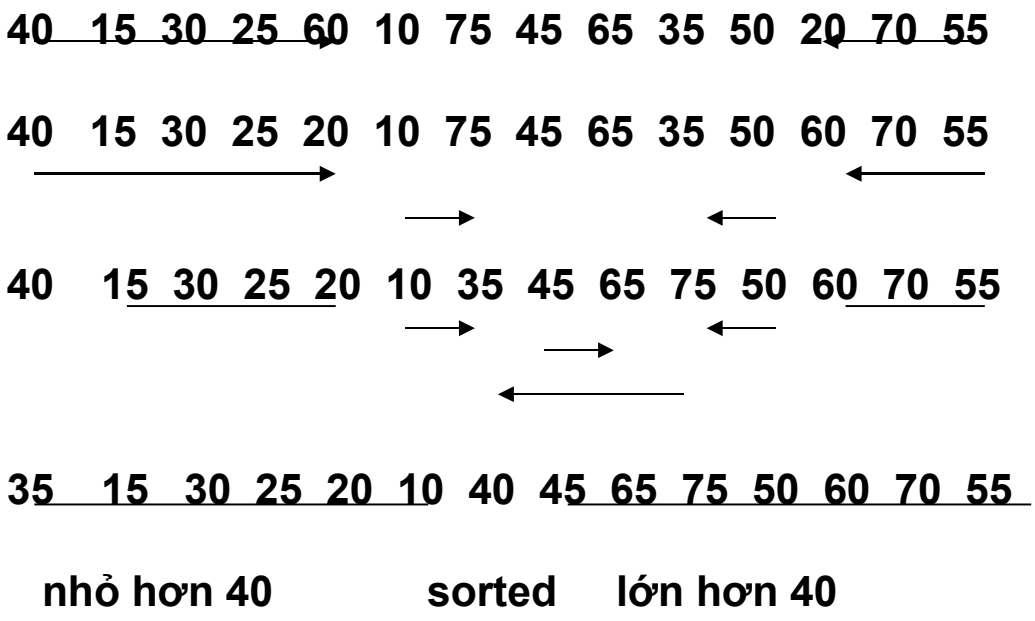
iii) tất cả những phần tử trong nhóm *a[i+1]*, ..., *a[right]* thì lớn hơn hay bằng *a[i]*

<u>Example</u>:

8	**59**	**56**	**52**	**55**	**58**	**51**	**57**	**54**
52	**51**	**<u>53</u>**	**56**	**55**	**58**	**59**	**57**	**54**

Thí dụ về phân hoạch

Giả sử chúng ta chọn phần tử thứ nhất hay phần tử tận cùng trái (*leftmost*) như là phần tử sẽ được đưa về vị trí đúng của nó (Phần tử này được gọi là phần tử chốt - *pivot*).



Giải thuật Quicksort

```
procedure quicksort2(left, right: integer);
var j, k: integer;
begin
        if right > left then
        begin
           j:=left; k:=right+1;
            //start partitioning
           repeat
                  repeat j:=j+1 until a[j] >= a[left];
                  repeat k:=k-1 until a[k]<= a[left];
                  if j< k then swap(a[j],a[k])
           until j>k;
           swap(a[left],a[k]);  //finish partitioning
           quicksort2(left,k-1);
           quicksort2(k+1,right)
        end;
end;
```

Phân tích độ phức tạp: trường hợp tốt nhất

Trường hợp tốt nhất xảy ra với Quicksort là khi mỗi lần phân hoạch chia tập tin ra làm hai phần bằng nhau.
điều này làm cho số lần so sánh của Quicksort thỏa mãn hệ thức truy hồi:

$$C_N = 2C_{N/2} + N.$$

Số hạnh $2C_{N/2}$ là chi phí của việc sắp thứ tự hai nửa tập tin và N là chi phí của việc xét từng phần tử khi phân hoạch lần đầu.
Từ chương 1, việc giải hệ thức truy hồi này đã đưa đến lời giải:

$$C_N \approx N \lg N.$$

Phân tích độ phức tạp: trường hợp xấu nhất

Một trường hợp xấu nhất của Quicksort là khi tập tin đã có thứ tự rồi.

Khi đó, phần tử thứ nhất sẽ đòi hỏi *n* so sánh để nhận ra rằng nó nên ở đúng vị trí thứ nhất. Hơn nữa, sau đó phân đoạn bên trái là rỗng và và phân đoạn bên phải gồm n – 1 phần tử. Do đó với lần phân hoạch kế, phần tử thứ hai sẽ đòi hỏi *n-1* so sánh để nhận ra rằng nó nên ở đúng vị trí thứ hai. Và cứ tiếp tục như thế.

Như vậy tổng số lần so sánh sẽ là:

$$n + (n-1) + \ldots + 2 + 1 = n(n+1)/2 = (n^2 + n)/2 = O(n^2).$$

Độ phức tạp trường hợp xấu nhất của Quicksort là $O(n^2)$.

Độ phức tạp trường hợp trung bình của Quicksort

Công thức truy hồi chính xác cho tổng số so sánh mà Quick sort cần để sắp thứ tự N phần tử được hình thành một cách ngẫu nhiên:

$$C_N = (N+1) + (1/N) \sum_{1}^{N} (C_{k-1} + C_{N-k})$$

$$\text{với } N \geq 2 \text{ và } C_1 = C_0 = 0$$

Số hạng (N+1) bao gồm số lần so sánh phần tử chốt với từng phần tử khác, thêm hai lần so sánh để hai pointer giao nhau. Phần còn lại là do sự kiện mỗi phần tử ở vị trí *k* có cùng xác xuất 1/N để được làm phần tử chốt mà sau đó chúng ta có hai phân đoạn với số phần tử lần lượt là *k-1* và *N-k*.

Chú ý rằng, $C_0 + C_1 + \ldots + C_{N-1}$ thì giống hệt

$$C_{N-1} + C_{N-2} + \ldots + C_0$$, **nên ta có**

$$C_N = (N+1) + (1/N) \sum_{1}^{N} 2C_{k-1}$$

Ta có thể loại trừ đại lượng tính tổng bằng cách nhân cả hai vế với N và rồi trừ cho cùng công thức nhân với N-1:

$$NC_N - (N-1)\, C_{N-1} = N(N+1) - (N-1)N + 2C_{N-1}$$

Từ đó ta được

$$NC_N = (N+1)C_{N-1} + 2N$$

Chia cả hai vế với N(N+1) ta được hệ thức truy hồi:

$$C_N/(N+1) = C_{N-1}/N + 2/(N+1)$$

$$= C_{N-2}/(N-1) + 2/N + 2/(N+1)$$

.

.

$$= C_2/3 + \sum_{3}^{N} 2/(k+1)$$

$$C_N/(N+1) \approx 2\sum_{1}^{N} 1/k \approx 2\int_{1}^{N} 1/x\,dx = 2\ln N$$

Suy ra:

$$C_N \approx 2N\ln N$$

Độ phức tạp trường hợp trung bình của Quicksort (tt.)

Vì ta có:

$\ln N = (\log_2 N).(\log_e 2) = 0.69 \lg N$

$2N \ln N \approx 1.38\ N \lg N$.

$\Rightarrow$Tổng số so sánh trung bình của Quicksort chỉ khoảng chừng 38% cao hơn trong trường hợp tốt nhất.

<u>Mệnh đề</u>. ***Quicksort cần khoảng 2NlnN so sánh trong trường hợp trung bình*.**

3. Sắp thứ tự bằng cách trộn (mergesort)

Trước tiên, chúng ta xét một quá trình được gọi là *trộn* (*merging*), thao tác phối hợp hai tập tin đã có thứ tự thành một tập tin có thứ tự lớn hơn.

<u>Trộn</u>

Trong nhiều ứng dụng xử lý dữ liệu, ta phải duy trì một tập dữ liệu có thứ tự khá lớn. Các phần tử mới thường xuyên được thêm vào tập tin lớn.

Nhóm các phần tử được đính vào đuôi của tập tin lớn và toàn bộ tập tin được sắp thứ tự trở lại.

Tình huống đó rất thích hợp cho thao tác *trộn*.

Trộn

Giả sử ta có hai mảng số nguyên có thứ tự a[1..M] và b[1..N]. Ta muốn trộn chúng thành một mảng thứ ba c[1..M+N].

```
i:= 1; j  :=1;
for k:= 1 to M+N do
  if a[i] < b[j] then
        begin c [k] := a[i]; i:= i+1 end
 else  begin c[k] := b[j]; j := j+1 end;
```

<u>Ghi chú:</u> Giải thuật dùng a[M+1] và b[N+1] để làm phần tử *cầm canh* chứa hai giá trị lớn hơn mọi trị khóa khác. Nhờ chúng, khi một trong hai mảng đã cạn thì vòng lặp sẽ đưa phần còn lại của mảng còn lại vào mảng *c*.

Sắp thứ tự bằng phương pháp trộn

Một khi ta đã có thủ tục trộn, ta dùng nó làm cơ sở để xây dựng một thủ tục sắp thứ tự đệ quy.

Để sắp thứ tự một tập tin nào đó, ta chia thành hai đoạn bằng nhau, sắp thứ tự hai đoạn này một cách đệ quy và rồi trộn hai đoạn lại với nhau.

Mergesort thể hiện chiến lược thiết kế giải thuật theo lối "Chia để trị" (divide-and-conquer).

Giải thuật sau sắp thứ tự mảng *a[1..r]*, dùng mảng *b[1..r]* làm trung gian,

```
procedure mergesort(1,r: integer);
var i, j, k, m : integer;
begin
    if r-1>0 then
        begin
          m:=(r+1)/2; mergesort(1,m); mergesort(m+1,r);
          for i := m downto 1 do b[i] := a[j];
          for j :=m+1 to r do b[r+m+1-j] := a[j];
          for k :=1 to r do
              if b[i] < b[j] then
              begin a[k] := b[i] ; i := i+1 end
              else begin a[k] := b[j]; j:= j-1 end;
        end;
end;
```

A S O R T I N G E X A M P L E

A S

O R

A O R S

I T

G N

G I N T

A G I N O R S T

E X

A M

A E M X

L P

E L P

A E E L M P X

A A E E G I L M N O P R S T X

Thí dụ: Sắp thứ tự một mảng gồm những ký tự chữ

Độ phức tạp của giải thuật Mergesort

Tính chất 4.1: ***Sắp thứ tự bằng phương pháp trộn cần khoảng NlgN so sánh để sắp bất kỳ tập tin N phần tử nào*****.**

Đối với giải thuật mergesort đệ quy, số lần so sánh được mô tả bằng hệ thức truy hồi: $C_N = 2C_{N/2} + N$, với $C_1 = 0$.

Suy ra:

$$C_N \approx N \lg N$$

Tính chất 4.2: ***Sắp thứ tự bằng phương pháp trộn cần dùng chỗ bộ nhớ thêm tỉ lệ với N*****.**

4. Sắp thứ tự ngoại

Sắp thứ tự các tập tin lớn lưu trữ trên bộ nhớ phụ được gọi là *sắp thứ tự ngoại* (*external sorting*).

Sắp thứ tự ngoại rất quan trọng trong các hệ quản trị cơ sở dữ liệu (DBMSs).

Khối (block) và truy đạt khối (Block Access)

Hệ điều hành phân chia bộ nhớ phụ thành những khối có kích thước bằng nhau. Kích thước của khối thay đổi tùy theo hệ điều hành, nhưng thường ở khoảng 512 đến 4096 byte.

Các tác vụ căn bản trên các tập tin là

- **mang một khối ra bộ đệm ở bộ nhớ chính (read)**
- **mang một khối từ bộ nhớ chính về bộ nhớ phụ (write).**

Sắp thứ tự ngoại

Khi ước lượng thời gian tính toán của các giải thuật mà làm việc trên các tập tin, chúng ta phải xét số lần mà chúng ta đọc một khối ra bộ nhớ chính hay viết một khối về bộ nhớ phụ.

Một tác vụ như vậy được gọi là một truy đạt khối (block access) hay một truy đạt đĩa (*disk access*).

khối = trang (page)

Xếp thứ tự ngoại bằng p.p. trộn (External Sort-merge)

Kỹ thuật thông dụng nhất để sắp thứ tự ngoại là giải thuật sắp thứ tự ngoại bằng phương pháp trộn (*external sort-merge* algorithm) .

Phương pháp sắp thứ tự ngoại này gồm 2 bước:

- **tạo các run**
- **trộn run**

Phương pháp sắp thứ tự ngoại bằng phương pháp trộn cũng áp dụng kỹ thuật thiết kế giải thuật chia-để-trị.

M: số *trang* (page) của bộ đệm trong bộ nhớ chính (memory-buffer) .

Xếp thứ tự ngoại bằng p.p. trộn (tt.)

1. Trong bước 1, một số run có thứ tự được tạo ra bằng cách sau:

i = 0;
repeat
read M blocks of the file, or the rest of the file, whichever is smaller;
sort the in-memory part of the file;
write the sorted data to the run file R_i;
i = i+1;
until the end of the file.

2. Trong bước 2, các run được trộn lại.

Trộn run (trường hợp tổng quát)

Tác vụ trộn là sự khái quát hóa của phép *trộn hai đường* (*two-way merge*) được dùng bởi giải thuật sắp thứ tự nội bằng phương pháp trộn. Nó trộn N run, do đó nó được gọi là *trộn nhiều đường* (*n-way merge*).

- **Trường hợp tổng quát:**

Về tổng quát, nếu tập tin lớn hơn sức chứa của bộ đệm

$$N > M$$

thì không thể dành một trang trong bộ đệm cho mỗi run trong bước trộn. Trong trường hợp này, sự trộn phải trải qua nhiều chuyến (*passes*).

Vì chỉ có M-1 trang của bộ đệm dành cho các đầu vào, sự trộn có thể tiếp nhận *M-1* runs như là các đầu vào.

Trộn run [trường hợp tổng quát] (tt.)

Chuyến trộn đầu tiên làm việc như sau:

M-1 run đầu tiên được trộn lại thành một run cho chuyến kế tiếp. Rồi thì M-1 runs sẽ được trộn theo cách tương tự và cứ thế cho đến khi tất cả các run đầu tiên đều được giải quyết. Tại điểm này, tổng số run được giảm đi một thừa số M-1.

Nếu số run đã được giảm đi này vẫn còn ≥ M, một chuyến nữa sẽ được thực thi với các run được tạo ra bởi chuyến đầu tiên làm đầu vào.

Mỗi chuyến làm giảm tổng số run một thừa số M – 1. Các chuyến cứ lặp lại nhiều như cần thiết cho đến khi tổng số run nhỏ hơn M; chuyến cuối cùng sẽ tạo ra kết quả là một tập tin có thứ tự.

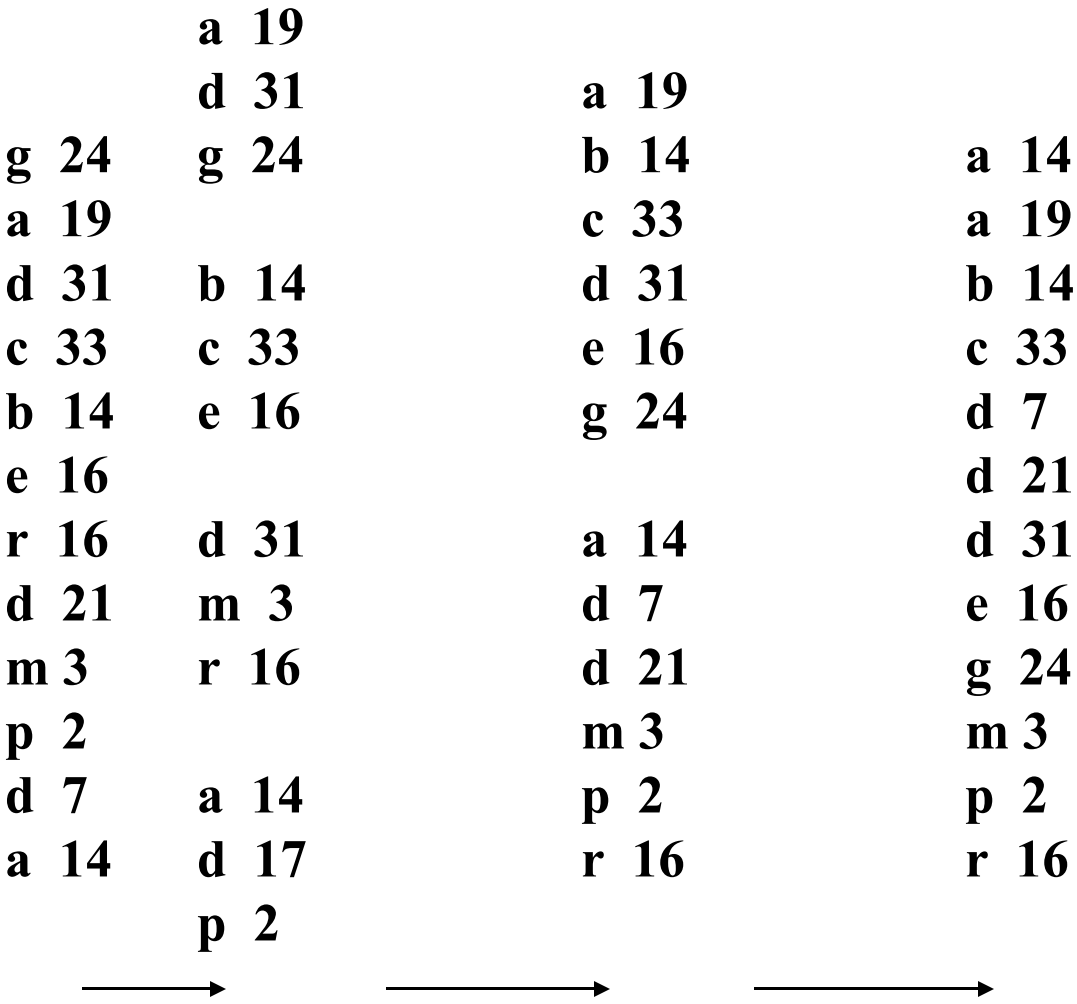
Một thí dụ của thứ tự ngoại bằng p.p. trộn

Giả sử: **i) một mẩu tin chiếm vừa một khối**

ii) bộ đệm chiếm 3 trang.

Trong giai đoạn trộn, hai trang được dùng làm đầu vào và một trang được dùng để chứa kết quả.

Giai đoạn trộn đòi hỏi hai chuyến.


g 24
a 19
d 31
c 33
b 14
e 16
r 16
d 21
m 3
p 2
d 7
a 14
a 19
d 31
g 24
b 14
c 33
e 16
d 31
m 3
r 16
a 14
d 17
p 2
a 19
b 14
c 33
d 31
e 16
g 24
a 14
d 7
d 21
m 3
p 2
r 16
a 14
a 19
b 14
c 33
d 7
d 21
d 31
e 16
g 24
m 3
p 2
r 16
Tạo run
trộn pass-1
trộn pass-2

Độ phức tạp của xếp thứ tự ngoại

Hãy tính số truy đạt khối (*block accesses*) của giải thuật sắp thứ tự ngoại bằng phương pháp trộn.

b_r : tổng số khối của tập tin.

Trong giai đoạn tạo run, một khối được đọc và ghi, đem lại một tổng số $2b_r$, truy đạt khối.

Tổng số run ban đầu là b_r/M.

Tổng số chuyến trộn: $\lceil \log_{M-1}(b_r/M) \rceil$

Trong mỗi chuyến trộn, từng khối của tập tin được đọc một lần và ghi một lần.

Độ phức tạp của xếp thứ tự ngoại(tt)

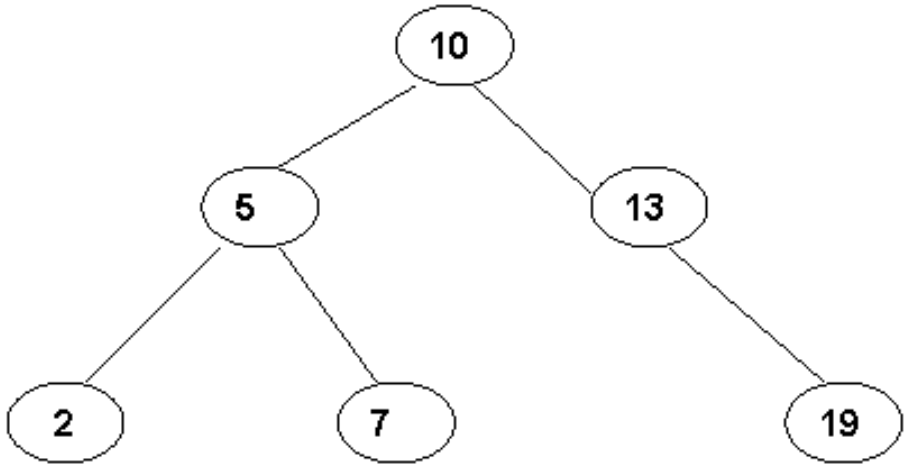
Tổng số truy đạt đĩa cho giải thuật sắp thứ tự ngoại bằng phương pháp trộn là:

$$2b_r + 2b_r \lceil \log_{M-1}(b_r/M) \rceil = 2b_r(\lceil \log_{M-1}(b_r/M) \rceil + 1)$$

tạo run — **các chuyến trộn**

5. Cây tìm kiếm nhị phân

Nhiều bài toán liên quan đến cây tìm kiếm nhị phân có thể được giải bằng cách áp dụng chiến lược chia-để-trị



Trong một *cây tìm kiếm nhị phân* (binary search tree), tất cả các mẩu tin với khóa nhỏ hơn khóa tại nút đang xét thì ở cây con bên trái của nút và các mẩu tin với khóa lớn hơn hay bằng khóa tại nút đang xét thì ở cây con bên phải của nút.

Khởi tạo cây nhị phân

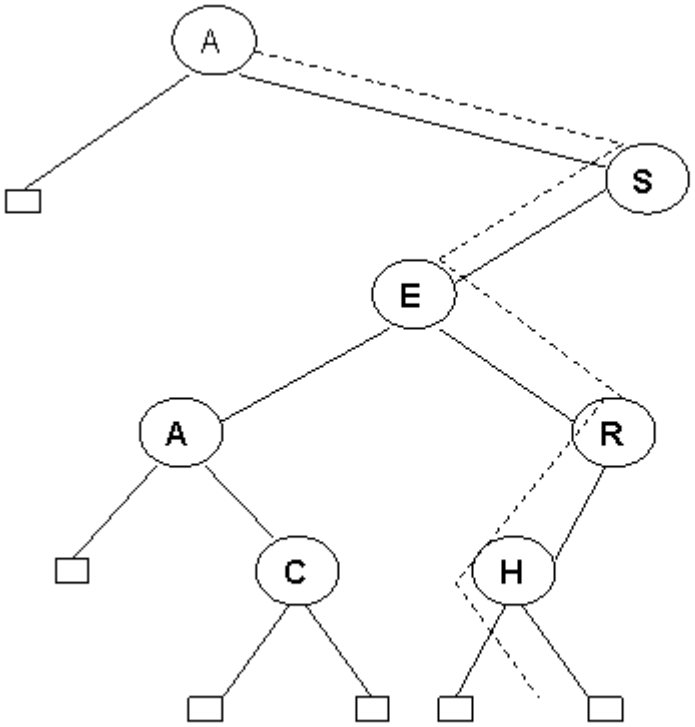
```
type link = ↑ node;
node = record key, info: integer;
                    l, r: link end;
var t, head, z: link;
```

Một cây rỗng được biểu diễn bằng cây có con trỏ bên phải chỉ đến nút giả *z*.

```
procedure tree_init;
begin
    new(z); z↑.1: = z; z↑.r: = z;
    new(head); head↑.key: = 0; head↑.r: = z;
end;
```

Tác vụ thêm vào

Thêm một nút vào trong cây, ta thực hiện một sự tìm kiếm (không thành công) nút ấy trên cây, rồi gắn nút ấy vào vị trí ứng với nút giả *z* tại điểm mà quá trình tìm kiếm kết thúc.



Hình vẽ minh họa việc thêm nút P vào cây nhị phân.

Tác vụ thêm vào (tt.)

```
procedure tree_insert (v: integer; x: link): link;
var p: link;
begin
  repeat
     p: = x;
     if v < x↑.key then x: = x↑.1 else x: = x↑.r
  until x = z;
  new(x); x↑.key: = v;
  x↑.1: = z; x↑.r: = z;                  /* create a new node  */
  if v < p↑. key then p↑.1: = x   /* p denotes the parent of
                                                the new node  */
   else p↑.r: = x;
   tree_insert: = x
end
```

Tác vụ tìm kiếm

```
type link = ↑ node;
        node = record key, info: integer;
        l, r: link end;
var t, head, z: link;

 function treesearch (v: integer, x: link): link; /* search the node with
                                  the key v in the binary search tree x  */
 begin
    while v <> x↑. key and x <> z do
     begin
        if v < x↑.key then x: = x↑.1
        else x: = x↑.r
    end;
    treesearch: = x
 end;
```

Tính chất của sự tìm kiếm trên cây nhị phân

Tính chất: ***Một tác vụ thêm vào hay tìm kiếm trên một cây nhị phân đòi hỏi chừng 2lnN so sánh trên một cây được tạo ra từ N trị khóa ngẫu nhiên*****.**

Chứng minh:

Chiều dài lối đi của 1 nút: là số cạnh cần duyệt qua để từ nút ấy về nút rễ **+1.**

Đối với mỗi nút trên cây nhị phân, số so sánh được dùng cho một sự tìm kiếm nút ấy thành công chính là chiều dài lối đi của nút ấy.

Tổng tất cả chiều dài lối đi của mọi nút trên cây nhị phân được gọi là *chiều dài lối đi* của cây nhị phân.

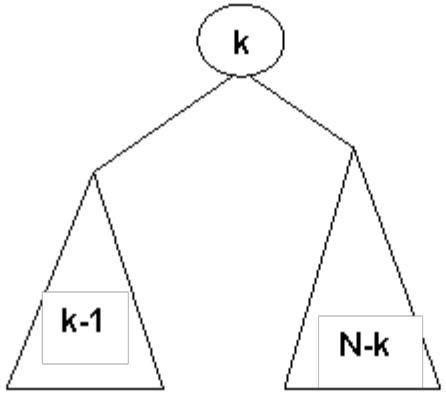
Chứng minh (tt.)

- Khi chia chiều dài lối đi toàn cây với N, ta sẽ được số so sánh trung bình đối với một sự tìm kiếm thành công trên cây.
- Nhưng nếu C_N biểu thị chiều dài lối đi trung bình của toàn cây, ta có một hệ thức truy hồi sau đây, với $C_1 = 1$

$$C_N = N + \sum_{1}^{N} (C_{k-1} + C_{N-k})$$

Số hạng N là do sự kiện nút rễ đóng góp 1 vào chiều dài lối đi của mỗi nút.
Số hạng thứ hai là do sự kiện khóa tại nút rễ có xác xuất bằng nhau để trở thành phần tử lớn thứ *k* trong cây, với hai cây con lần lượt chứa *k*-1 nút và *N-k.*

Chứng minh (tt.)



Hệ thức truy hồi này rất giống hệ thức truy hồi khi phân tích Quicksort, và nó đã được giải cùng một cách để đưa lại cùng một kết quả.

Do đó chiều dài trung bình của cây N nút là

$C_N \approx 2N \ln N$.

Suy ra chiều dài trung bình của một nút trong cây là $2\ln N$.

$\Rightarrow$ Một tác vụ tìm kiếm hay thêm vào đòi hỏi trung bình $2\ln N$ so sánh trên một cây gồm N nút.

Độ phức tạp trong trường hợp xấu nhất

- **<u>Tính chất:</u>** ***Trong trường hợp xấu nhất, một tác vụ tìm kiếm trên cây tìm kiếm nhị phân gồm N khóa có thể cần N so sánh*****.**
- **Trường hợp xấu nhất xảy ra khi cây nhị phân bị suy biến thành một danh sách liên kết.**