

Chương 5

Sắp xếp (Sorting)

NỘI DUNG

5.1. Bài toán sắp xếp

5.2. Ba thuật toán sắp xếp cơ bản

5.3. Sắp xếp trộn (Merge Sort)

5.4. Sắp xếp nhanh (Quick Sort)

5.5. Sắp xếp vun đống (Heap Sort)

5.1. Bài toán sắp xếp

- 5.1.1. Bài toán sắp xếp
- 5.1.2. Giới thiệu sơ lược về các thuật toán sắp xếp



5.1.1. Bài toán sắp xếp

- Sắp xếp (Sorting)
 - Quá trình tổ chức lại dữ liệu theo thứ tự giảm dần hoặc tăng dần
- Dữ liệu cần sắp xếp có thể là
 - Số nguyên
 - Xâu ký tự
 - Đối tượng (Objects)
- Khoá sắp xếp (Sort key)
 - Bộ phận của bản ghi xác định thứ tự sắp xếp của bản ghi trong danh sách.
 - Bản ghi sắp xếp theo thứ tự của các khoá.

5.1.1. Bài toán sắp xếp

Chú ý:

- Nếu sắp xếp trực tiếp trên bản ghi -> đòi hỏi di chuyển vị trí bản ghi -> tốn kém.
- Xây dựng **bảng khoá** gồm các bản ghi chỉ có 2 trường:
 - "**khoá**" : chứa giá trị khoá,
 - "**con trỏ**": ghi địa chỉ của bản ghi tương ứng.
- Sắp xếp trên **bảng khoá** không làm thay đổi bảng chính
- Nhưng trình tự các bản ghi trong bảng khoá cho phép xác định trình tự các bản ghi trong bảng chính.

5.1.1. Bài toán sắp xếp

Dạng tổng quát:

Input: Dãy n số $A = (a_1, a_2, \dots, a_n)$

Output: Một hoán vị (sắp xếp lại) (a'_1, \dots, a'_n) của dãy số đã cho thoả mãn

$$a'_1 \leq \dots \leq a'_n$$

5.1.1. Bài toán sắp xếp

- **Ứng dụng của sắp xếp:**
 - Quản trị cơ sở dữ liệu
 - Khoa học và kỹ thuật
 - Các thuật toán lập lịch,
 - ví dụ thiết kế chương trình dịch, truyền thông,...
 - Máy tìm kiếm web
 - và nhiều ứng dụng khác...

5.1.1. Bài toán sắp xếp

- **Các loại thuật toán sắp xếp**
 - Sắp xếp trong (internal sort)
 - Đòi hỏi dữ liệu được đưa toàn bộ vào bộ nhớ trong của máy tính
 - Sắp xếp ngoài (external sort)
 - dữ liệu không thể cùng lúc đưa toàn bộ vào bộ nhớ trong, nhưng có thể đọc vào từng bộ phận từ bộ nhớ ngoài

5.1.1. Bài toán sắp xếp

Các đặc trưng :

- Tại chỗ (in place):
 - nếu không gian nhớ phụ mà thuật toán đòi hỏi là $O(1)$, nghĩa là bị chặn bởi hằng số không phụ thuộc vào độ dài của dãy cần sắp xếp.
- Ổn định (stable):
 - Nếu các phần tử có cùng giá trị vẫn giữ nguyên thứ tự tương đối của chúng như trước khi sắp xếp.

5.1.1. Bài toán sắp xếp

2 phép toán thường phải sử dụng:

- **Đổi chỗ** (Swap): Thời gian thực hiện là $O(1)$

```
void swap( datatype &a, datatype &b) {  
    datatype temp = a;  
    //datatype-kiểu dữ liệu của phần tử  
    a = b;  
    b = temp;  
}
```

- **So sánh**: Compare(a, b) trả lại
 - **true** nếu a đi trước b trong thứ tự cần sắp xếp
 - **false** nếu trái lại.

5.1.2. Giới thiệu sơ lược về các thuật toán sắp xếp

Simple
algorithms:
 $O(n^2)$

Insertion sort
Selection sort
Bubble sort
Shell sort
...

Fancier
algorithms:
 $O(n \log n)$

Heap sort
Merge sort
Quick sort
...

Comparison
lower bound:
 $\Omega(n \log n)$

Specialized
algorithms:
 $O(n)$

Bucket sort
Radix sort

Handling
huge data
sets

External
sorting

NỘI DUNG

5.1. Bài toán sắp xếp

5.2. Ba thuật toán sắp xếp cơ bản

5.3. Sắp xếp trộn (Merge Sort)

5.4. Sắp xếp nhanh (Quick Sort)

5.5. Sắp xếp vun đống (Heap Sort)

5.2. Ba thuật toán sắp xếp cơ bản

- 5.2.1. Sắp xếp chèn (Insertion Sort)
- 5.2.2. Sắp xếp lựa chọn (Selection Sort)
- 5.2.3. Sắp xếp nổi bọt (Bubble Sort)

5.2.1. Sắp xếp chèn (Insertion Sort)

- Thuật toán:

- Tại bước $k = 1, 2, \dots, n$, đưa phần tử thứ k trong mảng đã cho vào đúng vị trí trong dãy gồm k phần tử đầu tiên.
- Kết quả là sau bước k , k phần tử đầu tiên là được sắp thứ tự.

```
void insertionSort(int a[], int array_size) {
```

```
    int i, j, last;
```

```
    for (i=1; i < array_size; i++) {
```

```
        last = a[i];
```

```
        j = i;
```

```
        while ((j > 0) && (a[j-1] > last)) {
```

```
            a[j] = a[j-1];
```

```
            j = j - 1; } // end while
```

```
        a[j] = last;
```

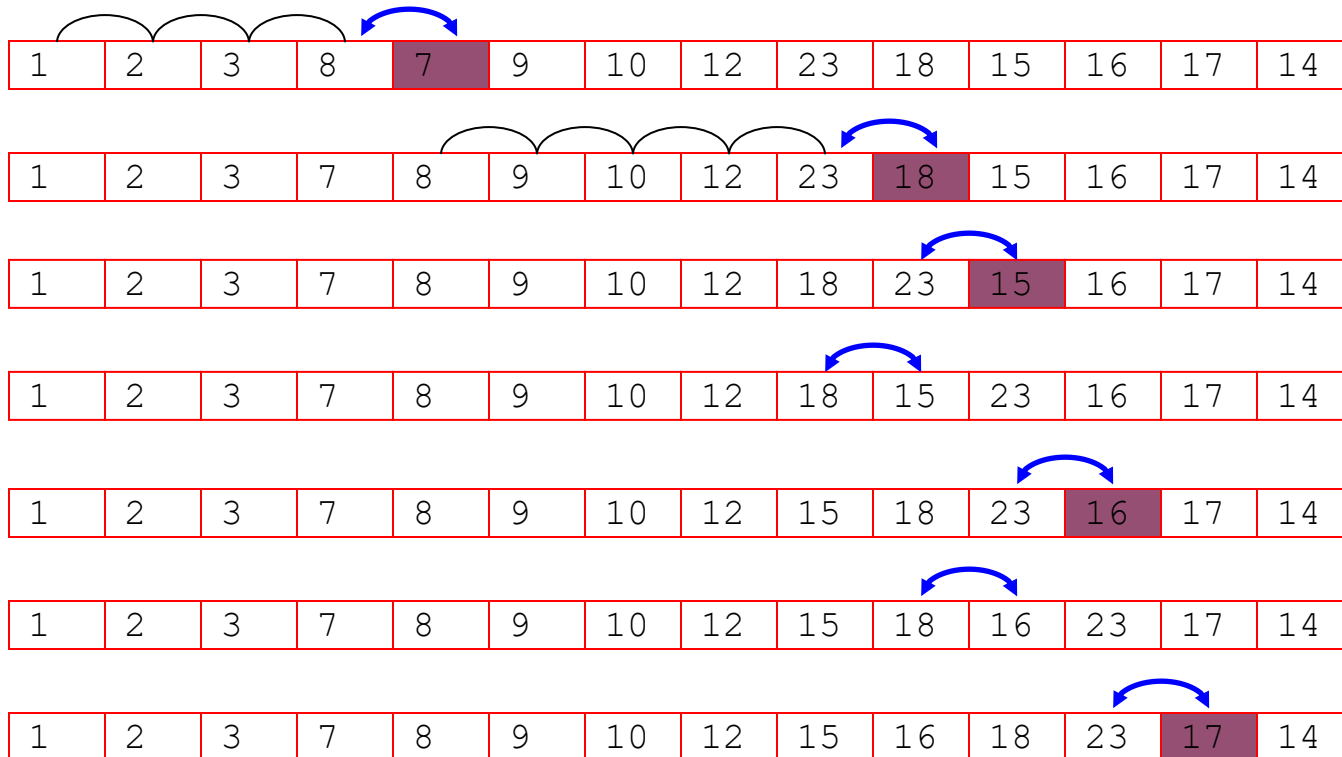
```
    } // end for
```

```
} // end of isort
```

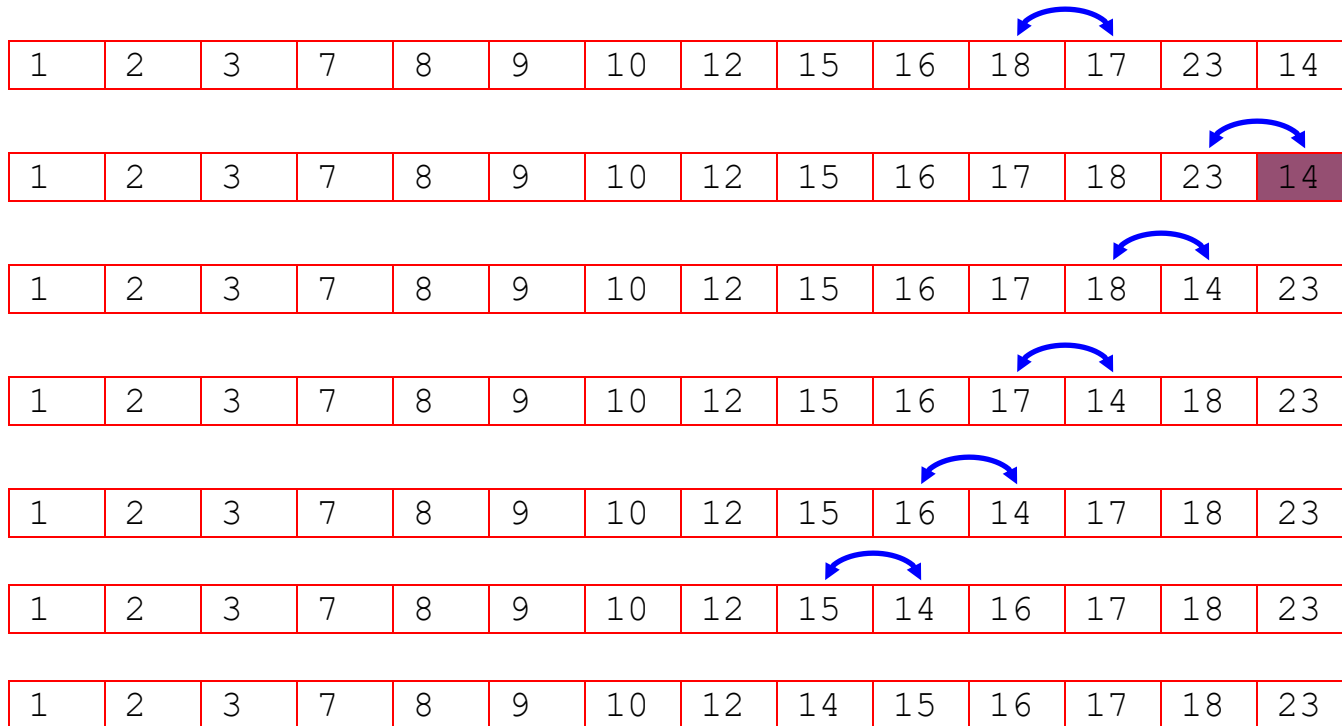
Giải thích:

- Ở đầu lần lặp i của vòng "for" ngoài, dữ liệu từ $a[0]$ đến $a[i-1]$ là được sắp xếp.
- Vòng lặp "while" tìm vị trí cho phần tử tiếp theo ($\text{last} = a[i]$) trong dãy gồm i phần tử đầu tiên.


Ví dụ Insertion sort (1)



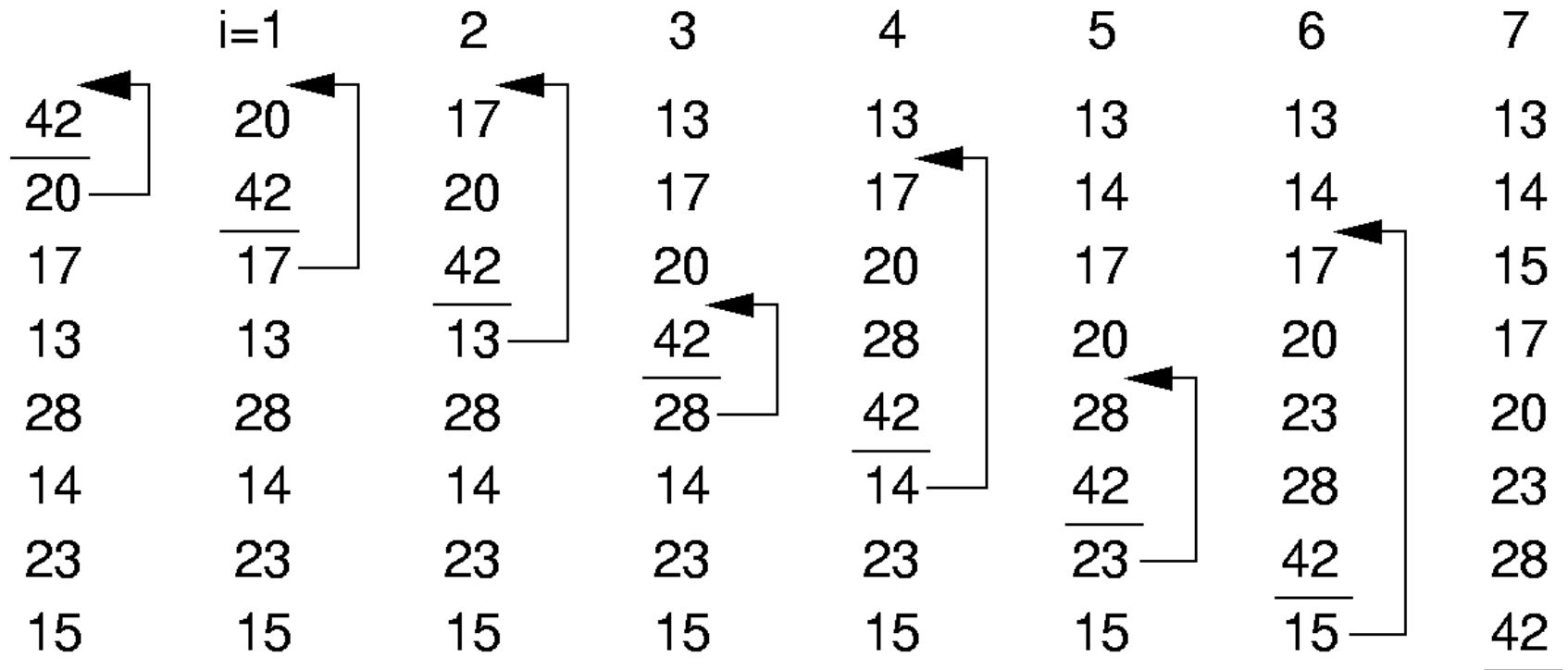
Ví dụ Insertion sort (1)



13 phép đổi chỗ: 

20 phép so sánh: 

Ví dụ: Insertion Sort (2)



Các đặc tính của Insertion Sort

- Tại chỗ và Ổn định (In place and Stable)
- Thời gian tính
 - **Best Case:** 0 hoán đổi, $n-1$ so sánh (*khi dãy đầu vào là đã được sắp*)
 - **Worst Case:** $n^2/2$ hoán đổi và so sánh (*khi dãy đầu vào có thứ tự ngược lại với thứ tự cần sắp xếp*)
 - **Average Case:** $n^2/4$ hoán đổi và so sánh
 - Trong tình huống tốt nhất là tốt nhất
- Thuật toán sắp xếp tốt đối với dãy đã *gần được sắp xếp*
 - mỗi phần tử đã đứng ở vị trí rất gần vị trí trong thứ tự cần sắp xếp

5.2.2. Sắp xếp chọn (Selection Sort)

- Thuật toán

- Tìm phần tử nhỏ nhất đưa vào vị trí 1
- Tìm phần tử nhỏ tiếp theo đưa vào vị trí 2
- Tìm phần tử nhỏ tiếp theo đưa vào vị trí 3
- ...

```
void swap(int &a,int &b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

```
void selectionSort(int a[], int n){
    int i, j, min, temp;
    for (i = 0; i < n-1; i++) {
        min = i;
        for (j = i+1; j < n; j++){
            if (a[j] < a[min]) min = j;
        }
        swap(a[i], a[min]);
    }
}
```

Selection Sort

Nhận xét:

- **Best case:** 0 đổi chỗ ($n-1$ như trong đoạn mã), $n^2/2$ so sánh.
- **Worst case:** $n - 1$ đổi chỗ và $n^2/2$ so sánh.
- **Average case:** $O(n)$ đổi chỗ và $n^2/2$ so sánh.
- **Ưu điểm nổi bật** :số phép đổi chỗ ít.
 - Có ý nghĩa nếu như thao tác đổi chỗ tốn kém.

Ví dụ: Selection Sort

	i=0	1	2	3	4	5	6
42	13	13	13	13	13	13	13
20	<u>20</u>	<u>14</u>	14	14	14	14	14
17	17	<u>17</u>	<u>15</u>	15	15	15	15
13	42	42	<u>42</u>	<u>17</u>	17	17	17
28	28	28	28	<u>28</u>	<u>20</u>	20	20
14	<u>14</u>	20	20	20	<u>28</u>	<u>23</u>	23
23	23	23	23	23	23	<u>28</u>	<u>28</u>
15	15	15	17	42	42	42	<u>42</u>

5.2.3. Sắp xếp nổi bọt - Bubble Sort

- Bắt đầu từ đầu dãy, thuật toán tiến hành so sánh mỗi phần tử với phần tử đi sau nó và thực hiện đổi chỗ, nếu chúng không theo đúng thứ tự.
- Quá trình này sẽ được lặp lại cho đến khi gặp lần duyệt từ đầu dãy đến cuối dãy mà không phải thực hiện đổi chỗ (tức là tất cả các phần tử đã đứng đúng vị trí).
- Cách làm này đã đẩy phần tử lớn nhất xuống cuối dãy, trong khi đó những phần tử có giá trị nhỏ hơn được dịch chuyển về đầu dãy.

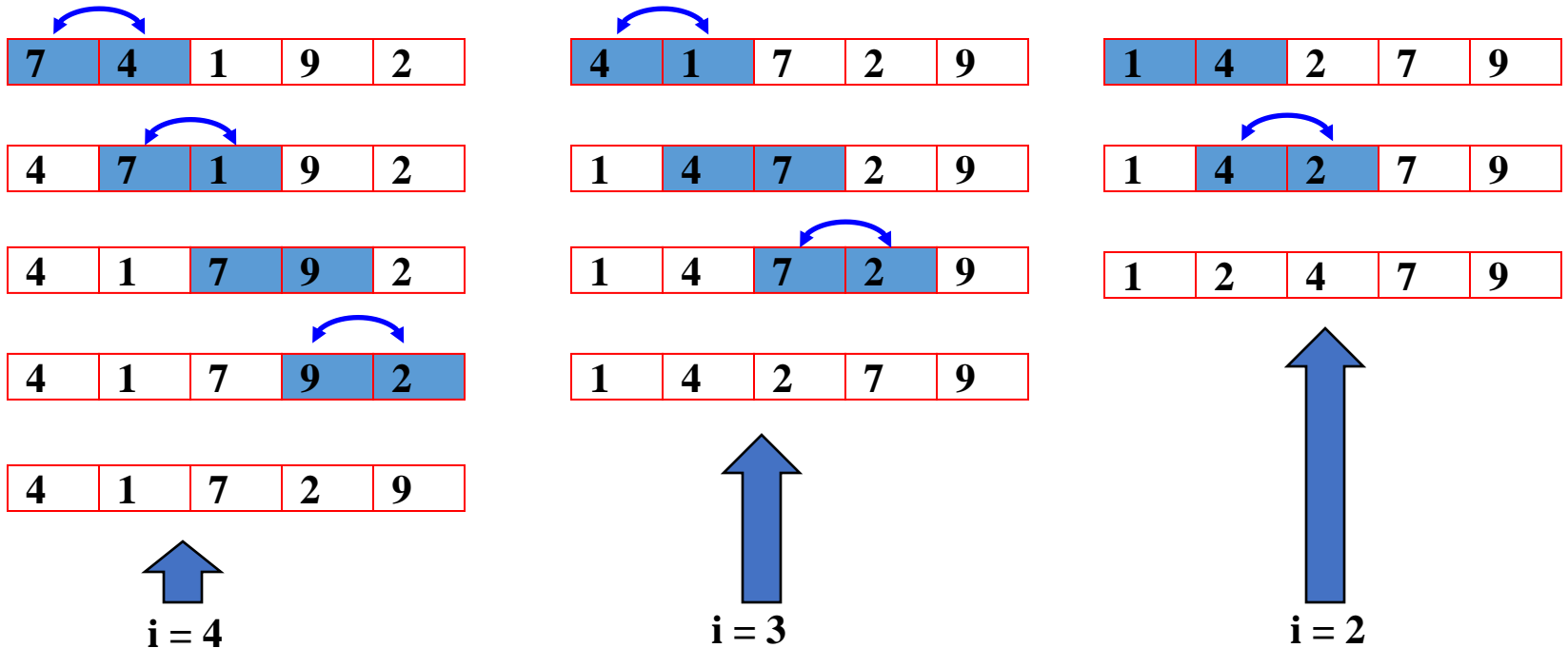
5.2.3. Sắp xếp nổi bọt - Bubble Sort

```
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-1] > a[j])
                swap(a[j-1], a[j]);
        }
    }
}
```

```
void swap(int &a, int &b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

- Best case: 0 đổi chỗ, $n^2/2$ so sánh.
- Worst case: $n^2/2$ đổi chỗ và so sánh.
- Average case: $n^2/4$ đổi chỗ và $n^2/2$ so sánh.

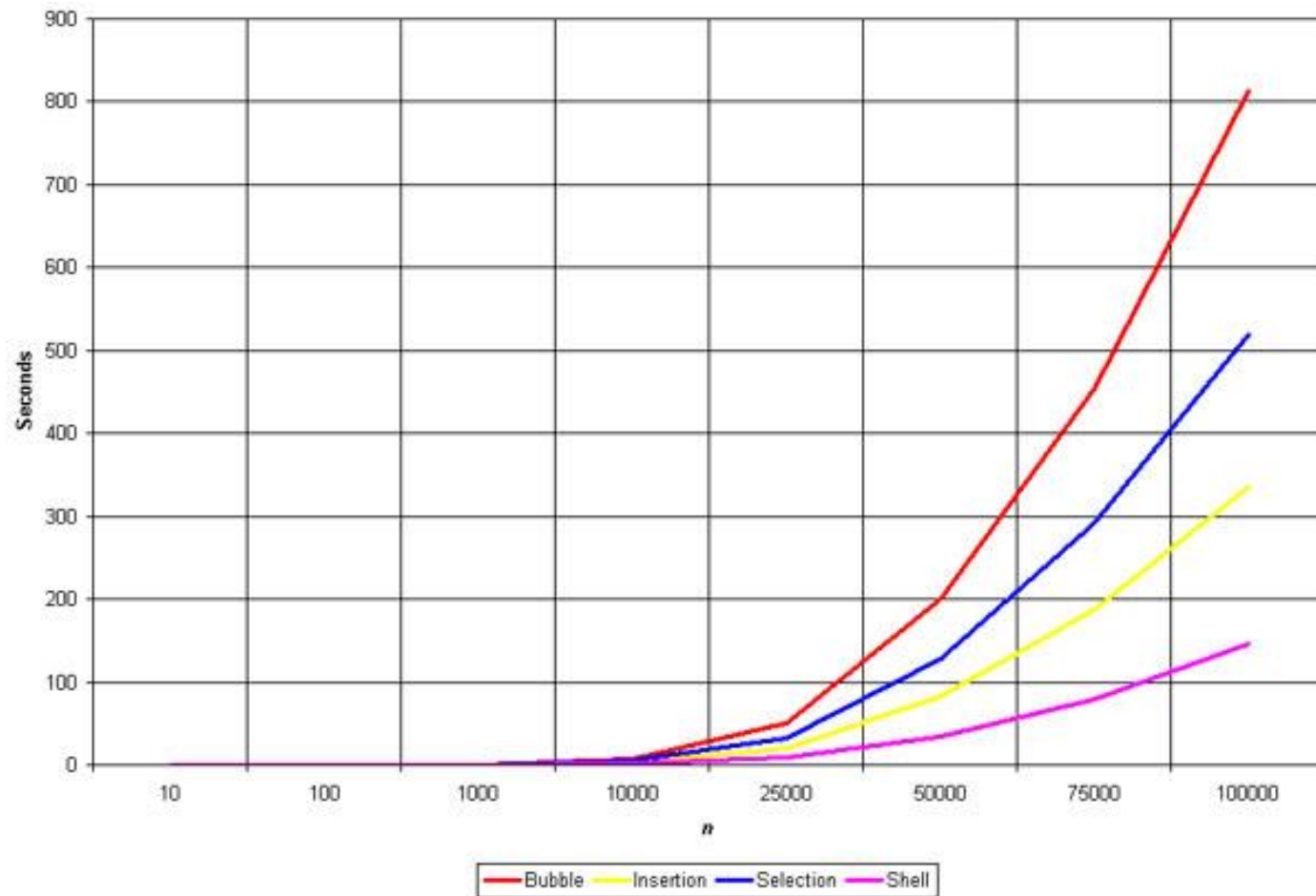
Ví dụ: Bubble Sort



Chú ý:

- Các phần tử được đánh chỉ số bắt đầu từ 0.
- $n=5$

So sánh ba thuật toán cơ bản



Tổng kết 3 thuật toán sắp xếp cơ bản

	Insertion	Bubble	Selection
Comparisons:			
Best Case	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Swaps			
Best Case	0	0	$\Theta(n)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

NỘI DUNG

5.1. Bài toán sắp xếp

5.2. Ba thuật toán sắp xếp cơ bản

5.3. Sắp xếp trộn (Merge Sort)

5.4. Sắp xếp nhanh (Quick Sort)

5.5. Sắp xếp vun đống (Heap Sort)

Sắp xếp trộn (Merge Sort)

- **Bài toán:** Cần sắp xếp mảng $A[1 .. n]$:
- **Chia (Divide)**
 - Chia dãy gồm n phần tử cần sắp xếp ra thành 2 dãy, mỗi dãy có $n/2$ phần tử
- **Trị (Conquer)**
 - Sắp xếp mỗi dãy con một cách đệ qui sử dụng **sắp xếp trộn**
 - Khi dãy chỉ còn một phần tử thì trả lại phần tử này
- **Tổ hợp (Combine)**
 - Trộn (Merge) hai dãy con được sắp xếp để thu được dãy được sắp xếp gồm tất cả các phần tử của cả hai dãy con

Merge Sort

MERGE-SORT(A, p, r)

if $p < r$

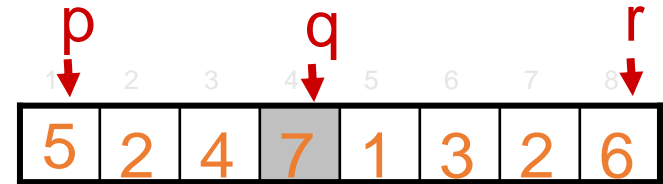
then $q \leftarrow \lfloor (p + r)/2 \rfloor$

MERGE-SORT(A, p, q)

MERGE-SORT($A, q + 1, r$)

MERGE(A, p, q, r)

endif



▷ Kiểm tra điều kiện neo

▷ Chia (Divide)

▷ Trị (Conquer)

Trị (Conquer)

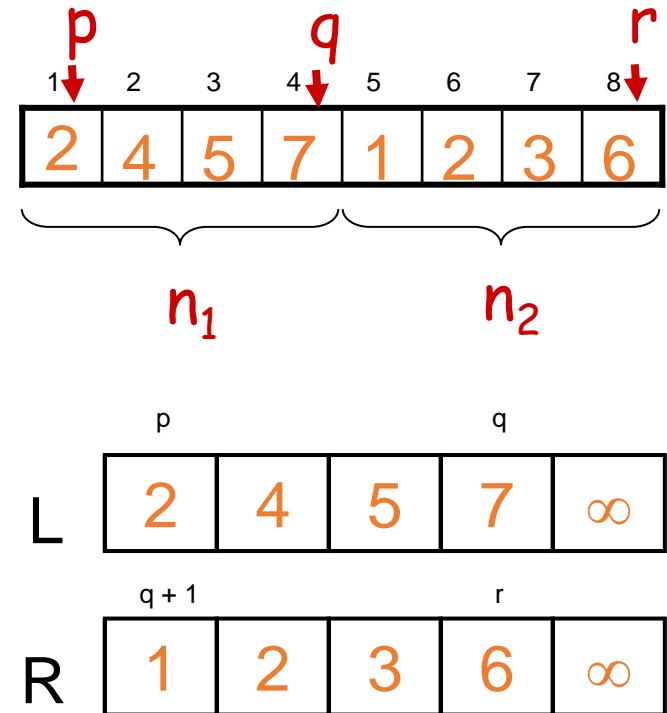
▷ Tổ hợp (Combine)

- Lệnh gọi thực hiện thuật toán: MERGE-SORT($A, 1, n$)

Trộn (Merge) - Pseudocode

MERGE(A, p, q, r)

1. Tính n_1 và n_2
2. Sao n_1 phần tử đầu tiên vào $L[1 \dots n_1]$ và n_2 phần tử tiếp theo vào $R[1 \dots n_2]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r **do**
6. **if** $L[i] \leq R[j]$
7. **then** $A[k] \leftarrow L[i]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[j]$
10. $j \leftarrow j + 1$



Thời gian tính của trộn

- Khởi tạo (tạo 2 mảng con tạm thời L và R):
 - $\Theta(n_1 + n_2) = \Theta(n)$
- Đưa các phần tử vào mảng kết quả (vòng lặp **for** cuối cùng):
 - n lần lặp, mỗi lần đòi hỏi thời gian hằng số $\Rightarrow \Theta(n)$
- Tổng cộng thời gian của trộn là:
 - $\Theta(n)$

Thời gian tính của sắp xếp trộn

MERGE-SORT Running Time

- **Chia:**

- tính q như là giá trị trung bình của p và r : $D(n) = \Theta(1)$

- **Trị:**

- giải đệ qui 2 bài toán con, mỗi bài toán kích thước $n/2 \Rightarrow 2T(n/2)$

- **Tổ hợp:**

- TRỘN (MERGE) trên các mảng con cỡ n phần tử đòi hỏi thời gian $\Theta(n) \Rightarrow C(n) = \Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{nếu } n = 1 \\ 2T(n/2) + \Theta(n) & \text{nếu } n > 1 \end{cases}$$

- **Suy ra theo định lý thợ: $T(n) = \Theta(n \log n)$**

Ví dụ: Sắp xếp trộn

Divide

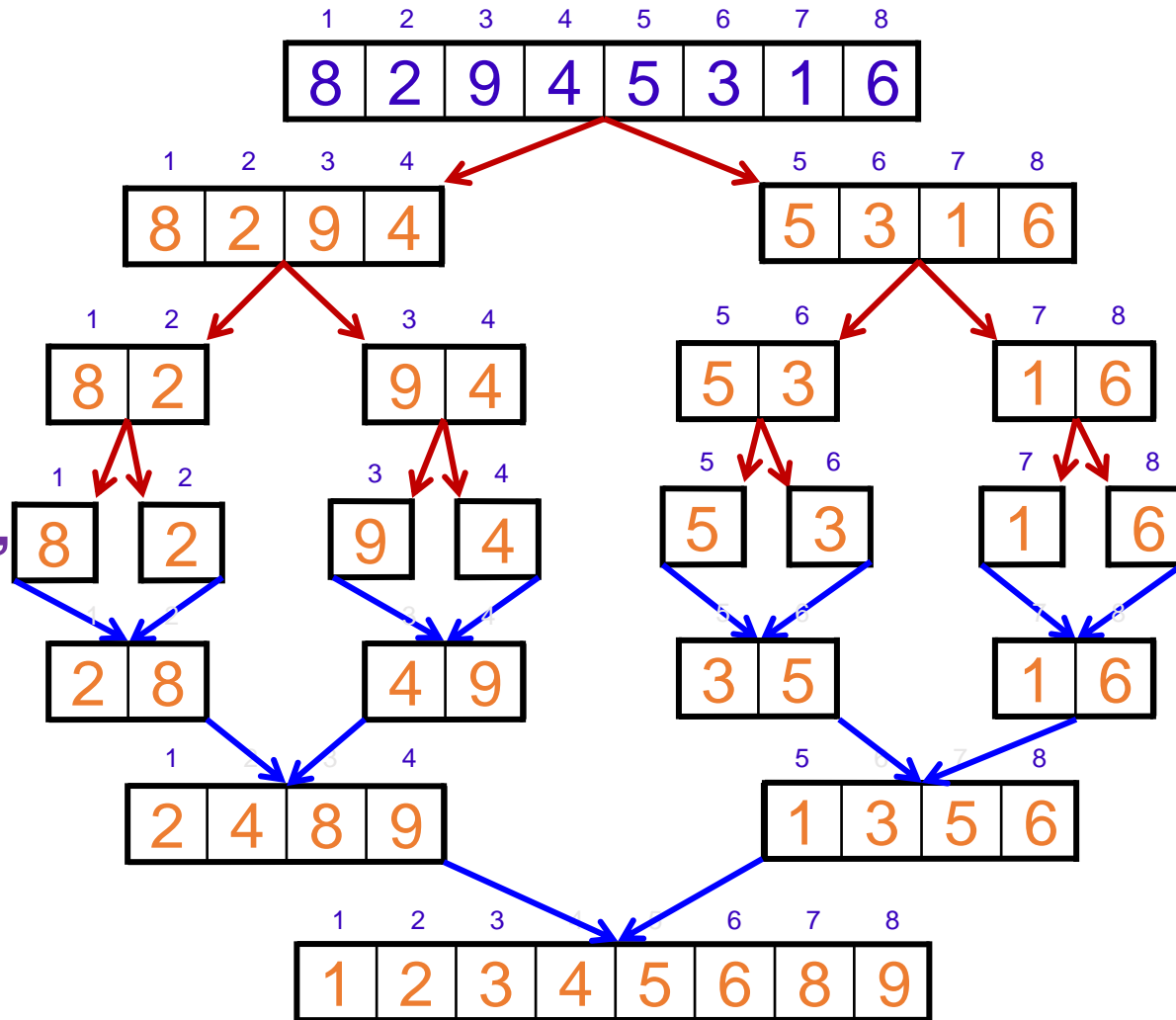
Divide

1 phần tử

Merge

Merge

Kết quả:



Cài đặt merge: Trộn A[first..mid] và A[mid+1.. last]

```
void merge(DataType A[], int first, int mid, int last) {
    DataType tempA[MAX_SIZE];    // mảng phụ
    int first1 = first; int last1 = mid;
    int first2 = mid + 1; int last2 = last; int index =
    first1;
    for (; (first1 <= last1) && (first2 <= last2); ++index) {
        if (A[first1] < A[first2]) {
            tempA[index] = A[first1]; ++first1;
        }
        else {
            tempA[index] = A[first2]; ++first2;
        }
    }
    for (; first1 <= last1; ++first1, ++index)
        tempA[index] = A[first1]; // sao nốt dãy con 1
    for (; first2 <= last2; ++first2, ++index)
        tempA[index] = A[first2]; // sao nốt dãy con 2
    for (index = first; index <= last; ++index)
        A[index] = tempA[index]; // sao trả mảng kết quả
} // end merge
```

Chú ý: DataType: kiểu dữ liệu phần tử mảng.

Cài đặt mergesort

```
void mergesort(DataType A[], int first, int last)
{
    if (first < last)
    {
        // chia thành hai dãy con
        int mid = (first + last)/2;    // chỉ số điểm giữa
        // sắp xếp dãy con trái A[first..mid]
        mergesort(A, first, mid);
        // sắp xếp dãy con phải A[mid+1..last]
        mergesort(A, mid+1, last);
        // Trộn hai dãy con
        merge(A, first, mid, last);
    } // end if
} // end mergesort
```

Lệnh gọi thực hiện mergesort (A, 0, n-1)

NỘI DUNG

5.1. Bài toán sắp xếp

5.2. Ba thuật toán sắp xếp cơ bản

5.3. Sắp xếp trộn (Merge Sort)

5.4. Sắp xếp nhanh (Quick Sort)

5.5. Sắp xếp vun đống (Heap Sort)

5.4. Sắp xếp nhanh (Quick Sort)

- 5.4.1. Sơ đồ tổng quát
- 5.4.2. Phép phân đoạn
- 5.4.3. Độ phức tạp của sắp xếp nhanh

5.4.1. Sơ đồ Quick Sort

- Phát triển bởi Hoare năm 1960 khi ông đang làm việc cho hãng máy tính nhỏ Elliott Brothers ở Anh.
- Theo thống kê tính toán, Quick sort (QS) là thuật toán sắp xếp nhanh nhất hiện nay.
- QS có thời gian tính trung bình là $O(n \log n)$, tuy nhiên thời gian tính tồi nhất của nó lại là $O(n^2)$.
 - sắp xếp tại chỗ, nhưng nó không có tính ổn định.
 - khá đơn giản về lý thuyết, nhưng lại không dễ cài đặt.

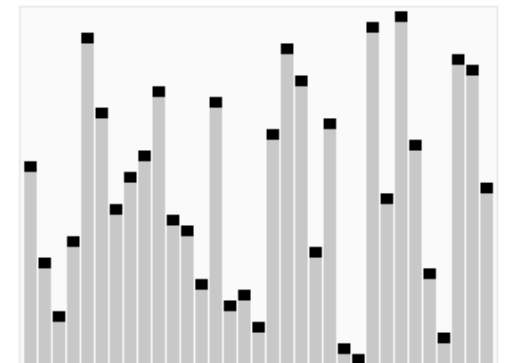


C.A.R. Hoare

January 11, 1934

ACM Turing Award, 1980

Photo: 2006



đường nằm ngang cho biết pivot

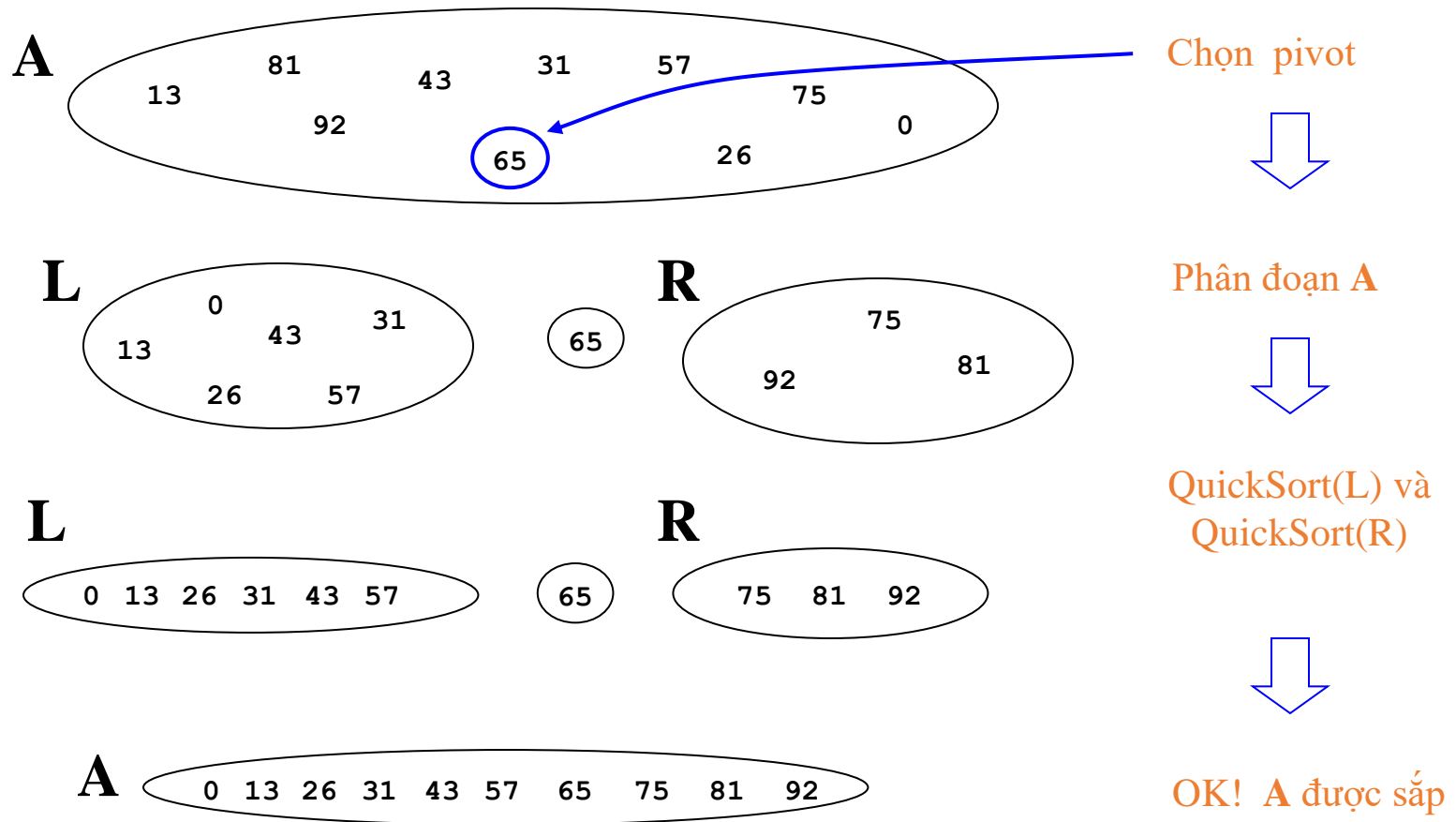
5.4.1. Sơ đồ Quick Sort

- Dựa trên kỹ thuật chia để trị.
- Có thể được mô tả đệ qui tương tự như Merge Sort
- Ngược lại với Merge Sort, trong QS
 - thao tác chia là phức tạp,
 - nhưng thao tác tổng hợp lại đơn giản.
- Điểm mấu chốt để thực hiện QS chính là thao tác chia.
- => Phụ thuộc vào thuật toán thực hiện thao tác này mà ta có các dạng QS cụ thể.

5.4.1. Sơ đồ Quick Sort

1. **Neo đệ qui**: Nếu dãy chỉ còn không quá 1 phần tử thì nó là dãy được sắp và trả lại ngay dãy này mà không phải làm gì cả.
2. **Chia** :
 - Chọn 1 phần tử trong dãy và gọi nó là **phần tử chốt p (pivot)**.
 - Chia dãy đã cho ra thành 2 dãy con:
 - Dãy con trái (L) gồm những phần tử \leq phần tử chốt,
 - Dãy con phải (R) gồm các phần tử $>$ phần tử chốt.
 - Thao tác này được gọi là **"Phân đoạn" (Partition)**.
3. **Trị** : Lặp lại một cách đệ qui thuật toán đối với 2 dãy con L và R .
4. **Tổng hợp** (Combine): Dãy được sắp xếp là $L \ p \ R$.

Các bước của QuickSort



Sơ đồ tổng quát của QS

- Sơ đồ tổng quát của QS có thể mô tả như sau:

Quick-Sort(*A, Left, Right*)

```
1.  if (Left < Right ) {  
2.      Pivot = Partition(A, Left, Right);  
3.      Quick-Sort(A, Left, Pivot - 1);  
4.      Quick-Sort(A, Pivot + 1, Right); }
```

- Hàm Partition(*A, Left, Right*) thực hiện chia $A[Left..Right]$ thành 2 đoạn $A[Left..Pivot - 1]$ và $A[Pivot + 1..Right]$ sao cho:
 - Các phần tử trong $A[Left..Pivot - 1] \leq A[Pivot]$
 - Các phần tử trong $A[Pivot + 1..Right] \geq A[Pivot]$.
- Lệnh gọi thực hiện thuật toán **Quick-Sort(*A, 1, n*)**

Sơ đồ tổng quát của QS

- Khi dãy con chỉ còn một số lượng không lớn phần tử (VD: 9 phần tử) -> sử dụng các thuật toán đơn giản để sắp xếp dãy này, chứ không nên tiếp tục chia nhỏ.
- Thuật toán trong tình huống như vậy có thể mô tả như sau:

Quick-Sort(A, Left, Right)

1. **if** (*Right - Left* < n_0)
2. *Insertion_sort(A, Left, Right);*
3. **else {**
4. *Pivot = Partition(A, Left, Right);*
5. *Quick-Sort(A, Left, Pivot - 1);*
6. *Quick-Sort(A, Pivot + 1, Right); }*

5.4.2. Thao tác chia

- Trong QS thao tác chia bao gồm 2 công việc:
 - Chọn phần tử chốt **p** .
 - Phân đoạn: Chia dãy đã cho ra thành 2 dãy con
- Thao tác phân đoạn có thể cài đặt (tại chỗ) với thời gian $\Theta(n)$.
- Hiệu quả của thuật toán phụ thuộc rất nhiều vào việc phần tử nào được chọn làm phần tử chốt:
 - Thời gian tính trong tình huống tồi nhất của QS là $O(n^2)$.
 - Xảy ra khi danh sách là đã được sắp xếp và phần tử chốt được chọn là phần tử trái nhất của dãy.
 - Nếu phần tử chốt được chọn ngẫu nhiên \rightarrow QS có độ phức tạp tính toán là $O(n \log n)$.

Chọn phần tử chốt

- Việc chọn phần tử chốt có vai trò quyết định đối với hiệu quả của thuật toán.
- Tốt nhất nếu chọn được phần tử chốt là phần tử đứng giữa trong danh sách được sắp xếp (ta gọi phần tử như vậy là **trung vị/median**).
- Khi đó, sau $\log_2 n$ lần phân đoạn ta sẽ đạt tới danh sách với kích thước bằng 1.
- Tuy nhiên, điều đó rất khó thực hiện.

Chọn phần tử chốt

- Người ta thường sử dụng các cách chọn phần tử chốt sau đây:
 - Chọn phần tử trái nhất (đứng đầu) làm phần tử chốt.
 - Chọn phần tử phải nhất (đứng cuối) làm phần tử chốt.
 - Chọn phần tử đứng giữa danh sách làm phần tử chốt.
 - Chọn phần tử trung vị trong 3 phần tử đứng đầu, đứng giữa và đứng cuối làm phần tử chốt ().
 - Chọn ngẫu nhiên một phần tử làm phần tử chốt.

Thuật toán phân đoạn

- xây dựng hàm **Partition(a, left, right)** :
- **Input:** Mảng $a[\textit{left} .. \textit{right}]$.
- **Output:** Phân bố lại các phần tử của mảng đầu vào và trả lại chỉ số \textit{jpivot} thoả mãn:
 - $a[\textit{jpivot}]$ chứa giá trị ban đầu của $a[\textit{left}]$,
 - $a[i] \leq a[\textit{jpivot}]$, với mọi $\textit{left} \leq i < \textit{pivot}$,
 - $a[j] \geq a[\textit{jpivot}]$, với mọi $\textit{pivot} < j \leq \textit{right}$.

Phần tử chốt là phần tử đứng đầu

Partition(a, left, right)

$i = \text{left}; j = \text{right} + 1; \text{pivot} = a[\text{left}];$

while $i < j$ **do**

$i = i + 1;$

while $i \leq \text{right}$ **and** $a[i] < \text{pivot}$ **do** $i = i + 1;$

$j = j - 1;$

while $j \geq \text{left}$ **and** $a[j] > \text{pivot}$ **do** $j = j - 1;$

$\text{swap}(a[i], a[j]);$

$\text{swap}(a[i], a[j]); \text{swap}(a[j], a[\text{left}]);$

return $j;$

pivot được chọn là
phần tử đứng đầu

j là chỉ số (jpivot) cần trả lại,
do đó cần đổi chỗ $a[\text{left}]$ và $a[j]$



i

Sau khi chọn *pivot*, dịch các con trỏ i và j từ đầu và cuối mảng và đổi chỗ cặp phần tử thoả mãn $a[i] > \text{pivot}$ và $a[j] < \text{pivot}$



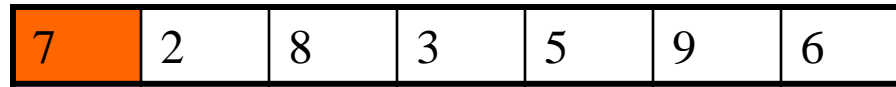
j

Ví dụ

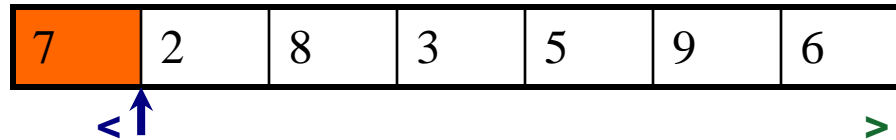
Vị trí:	0	1	2	3	4	5	6	7	8	9
Khoá (Key):	<u>9</u>	1	11	17	13	18	4	12	14	5
		>	>							<
lần 1×while:	<u>9</u>	1	5	17	13	18	4	12	14	11
				>			<	<	<	
lần 2×while:	<u>9</u>	1	5	4	13	18	17	12	14	11
				<	><	<				
lần 3×while:	<u>9</u>	1	5	13	4	18	17	12	14	11
2 lần đổi chỗ: 4		1	5	<u>9</u>	13	18	17	12	14	11

Ví dụ: Phân đoạn với pivot là phần tử đứng đầu

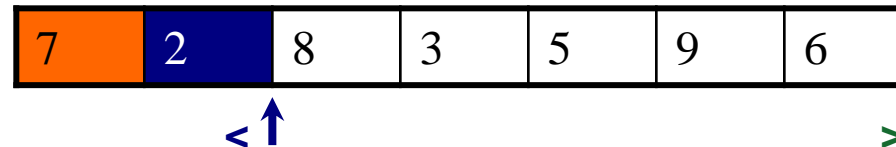
Chọn pivot:



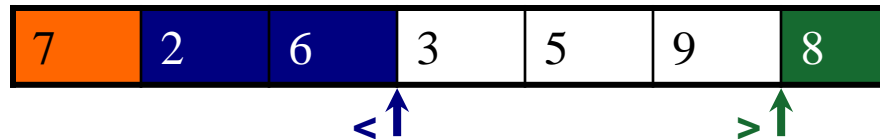
Phân đoạn: Con trỏ



2 nhỏ hơn pivot



đổi chỗ 6, 8



3,5 nhỏ hơn 9 lớn hơn



Kết thúc phân đoạn



Đưa pivot vào vị trí



Phần tử chốt là phần tử đứng giữa

PartitionMid(a, left, right);

i = left; j = right; pivot = a[(left + right)/2];

repeat

while *a[i] < pivot* **do** *i = i + 1;*

while *pivot < a[j]* **do** *j = j - 1;*

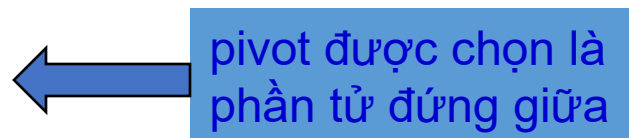
if *i <= j*

swap(a[i], a[j]);

i = i + 1; j = j - 1;

until *i > j;*

return *j;*



- Cài đặt thuật toán phân đoạn trong đó pivot được chọn là phần tử đứng giữa (Đây cũng là cách cài đặt mà TURBO C lựa chọn).*

Phần tử chốt là phần tử đứng cuối

```
int PartitionR(int a[], int p, int r) {  
    int x = a[r];  
    int j = p - 1;  
    for (int i = p; i < r; i++) {  
        if (x >= a[i])  
            { j = j + 1; swap(a[i], a[j]); }  
    }  
    a[r] = a[j + 1]; a[j + 1] = x;  
    return (j + 1);  
}
```

```
void quickSort(int a[], int p, int r) {  
    if (p < r) {  
        int q = PartitionR(a, p, r);  
        quickSort(a, p, q - 1);  
        quickSort(a, q + 1, r);  
    }  
}
```

Cài đặt QUICK SORT

```
void swap(int &a,int &b)
{ int t = a; a = b; b = t; }
```

```
int Partition(int a[], int left, int right) {
    int i, j, pivot;
    i = left; j = right + 1; pivot = a[left];
    while (i < j) {
        i = i + 1; while ((i <= right)&&(a[i] < pivot)) i++;
        j--; while ((j >= left)&& (a[j] > pivot)) j--;
        swap(a[i] , a[j]); }
    swap(a[i], a[j]); swap(a[j], a[left]);
    return j;
}

void quick_sort(int a[], int left, int right) {
    int pivot;
    if (left < right) {
        pivot = Partition(a, left, right);
        if (left < pivot) quick_sort(a, left, pivot-1);
        if (right > pivot) quick_sort(a, pivot+1, right);}
}
```

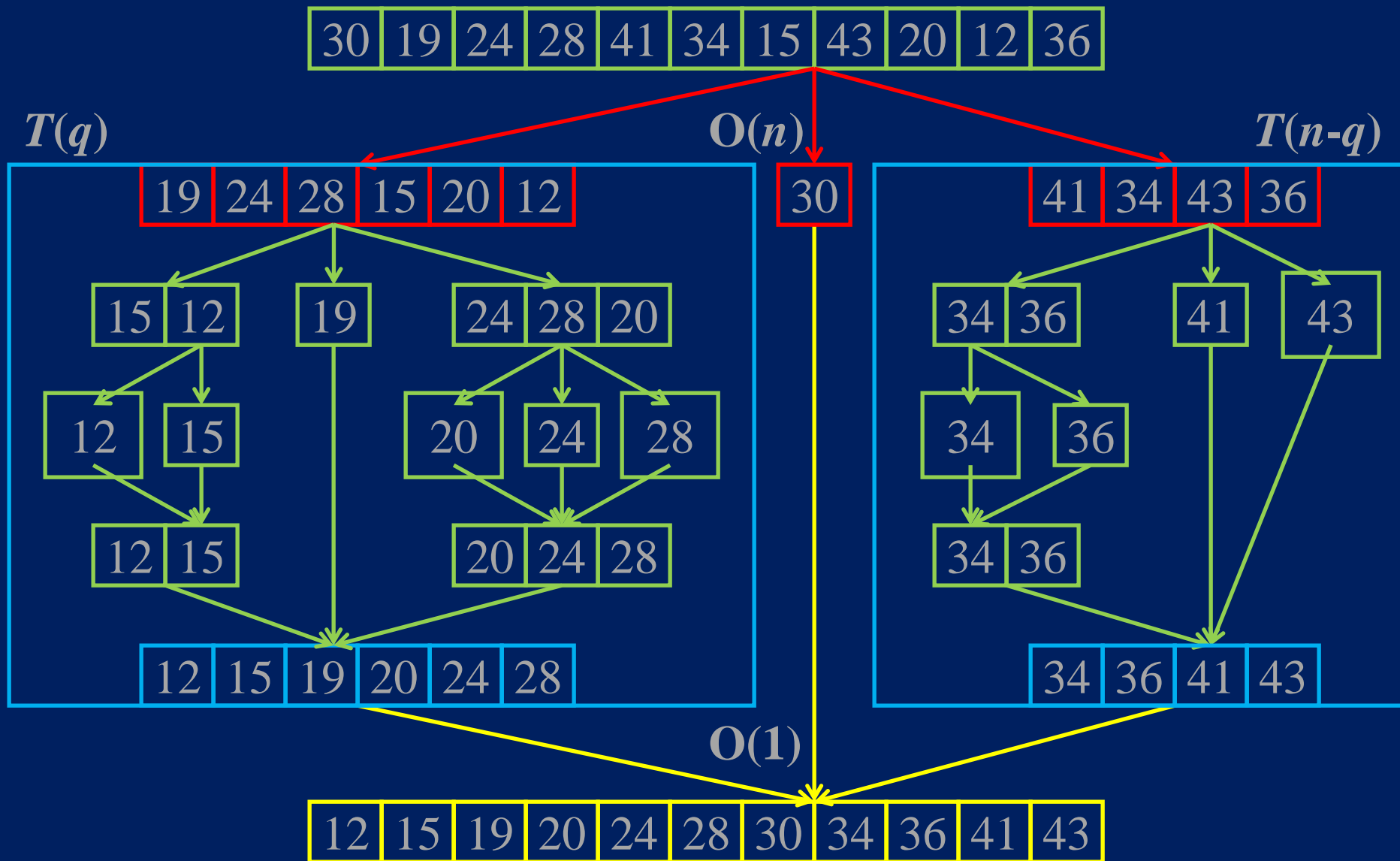
Cài đặt Quick Sort (dễ đọc hơn?)

```
int Partition(int a[], int L, int R)
{
    int i, j, p;
    i = L; j = R + 1; p = a[L];
    while (i < j) {
        i = i + 1;
        while ((i <= R)&&(a[i]<p)) i++;
        j--;
        while ((j >= L)&& (a[j]>p)) j--;
        swap(a[i] , a[j]);
    }
    swap(a[i], a[j]); swap(a[j], a[L]);
    return j;
}
```

```
void quick_sort(int a[], int left, int
    right)
{
    int p;
    if (left < right)
    {
        pivot = Partition(a, left, right);
        if (left < pivot)
            quick_sort(a, left, pivot-1);
        if (right > pivot)
            quick_sort(a, pivot+1, right);}
    }
```

Chương trình DEMO

```
/* CHUONG TRINH DEMO QUICK SORT */
/* include: stdlib.h; stdio.h; conio.h; process.h; include time.h */
void quick_sort(int a[], int left, int right);
void swap(int &a, int &b);
int Partition(int a[], int left, int right);
int a[1000]; int n; // n - so luong phan tu cua day can sap xep
void main() {
    int i; clrscr();
    printf("\n Give n = "); scanf("%i",&n); randomize();
    for (i = 0; i < n; i++) a[i] = rand(); // Tao day ngau nhien
    printf("\nDay ban dau la: \n");
    for (i = 0; i < n; i++) printf("%8i  ", a[i]);
    quick_sort(a, 0, n-1); // Thuc hien quick sort
    printf("\n Day da duoc sap xep.\n");
    for (i = 0; i < n; i++) printf("%8i  ", a[i]);
    a[n]=32767;
    for (i = 0; i < n; i++)
        if (a[i]>a[i+1]) {printf(" ??????? Loi o vi tri: %6i ", i); getch(); }
    printf("\n Correct!"); getch();
}
```



$$T(0) = T(1) = 1$$

$$T(n) = T(q) + T(n - q) + O(n) + O(1)$$

Thời gian tính của Quick-Sort

- Phụ thuộc vào việc phép phân chia là **cân bằng (balanced)** hay **không cân bằng (unbalanced)** -> phụ thuộc vào việc phần tử nào được chọn làm chốt.
- 1. **Phân đoạn không cân bằng:** 1 bài toán con có kích thước $n - 1$ còn bài toán kia có kích thước 0.
- 2. **Phân đoạn hoàn hảo (Perfect partition):** việc phân đoạn luôn được thực hiện dưới dạng phân đôi -> mỗi bài toán con có kích thước cỡ $n/2$.
- 3. **Phân đoạn cân bằng:** việc phân đoạn được thực hiện ở đâu đó quanh điểm giữa -> 1 bài toán con có kích thước $n - k$ còn bài toán kia có kích thước k .

Phân đoạn không cân bằng (Unbalanced partition)

Công thức đệ qui là:

$$T(n) = T(n-1) + T(0) + \Theta(n)$$

$$T(0) = T(1) = 1$$

$$T(n) = \cancel{T(n-1)} + n$$

$$\cancel{T(n-1)} = \cancel{T(n-2)} + (n-1)$$

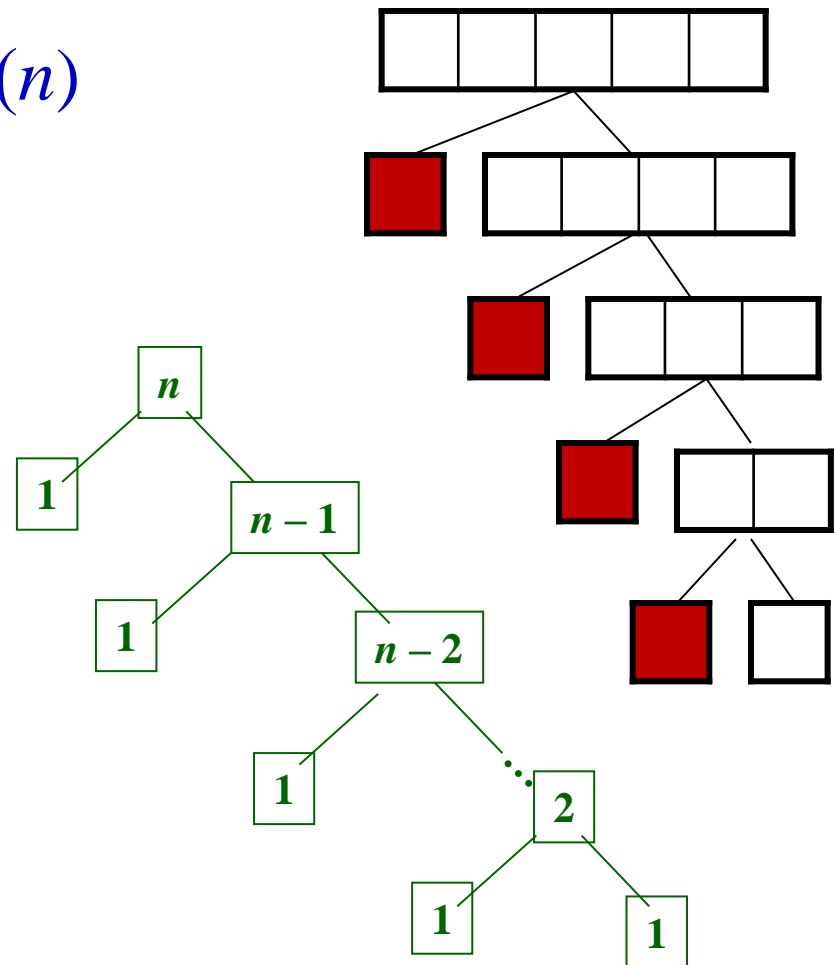
$$\cancel{T(n-2)} = \cancel{T(n-3)} + (n-2)$$

...

$$\cancel{T(2)} = T(1) + (2)$$

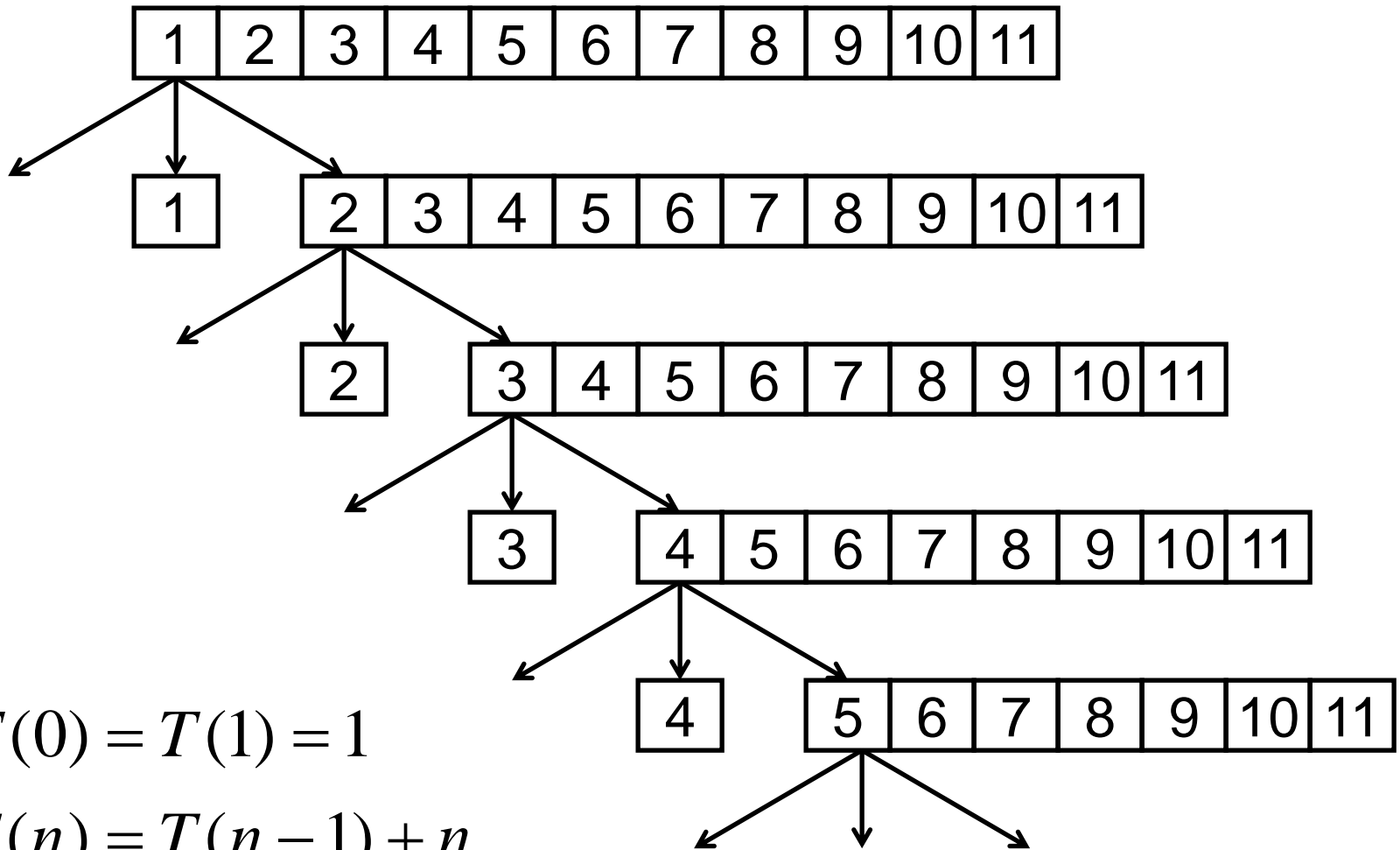
$$T(n) = T(1) + \sum_{i=2}^n i$$

$$= O(n^2)$$



Khi phần tử chốt được chọn là phần tử đứng đầu:

Tình huống tồi nhất xảy ra khi dãy đầu vào là đã được sắp xếp



$$T(0) = T(1) = 1$$

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

Phân đoạn hoàn hảo - Perfect partition

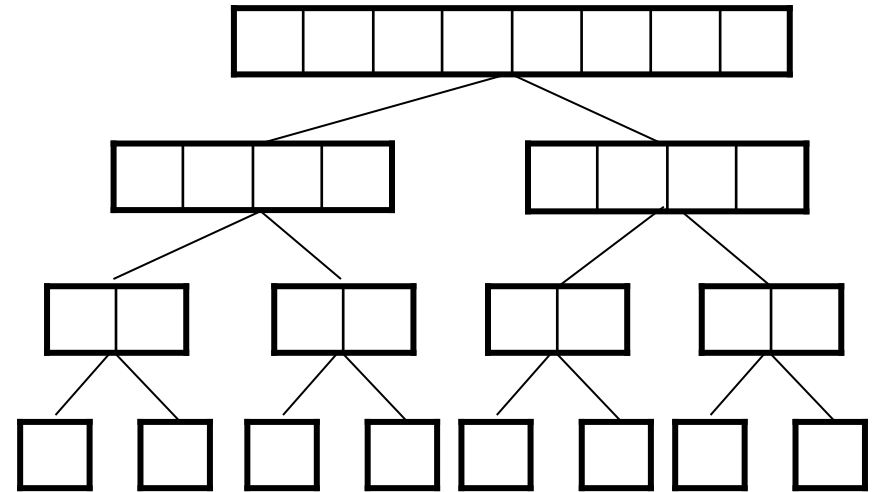
Công thức đệ quy:

$$T(n) = T(n/2) + T(n/2) + \Theta(n)$$

$$T(n) = 2T(n/2) + \Theta(n)$$

$$T(n) = \Theta(n \log n)$$

Theo định lý thợ !



It's Best Case!



Thời gian tính trung bình của QS

QuickSort Average Case

- Giả sử rằng pivot được chọn ngẫu nhiên trong số các phần tử của dãy đầu vào
- Tất cả các tình huống sau đây là đồng khả năng:
 - Pivot là phần tử nhỏ nhất trong dãy
 - Pivot là phần tử nhỏ nhì trong dãy
 - Pivot là phần tử nhỏ thứ ba trong dãy
 - ...
 - Pivot là phần tử lớn nhất trong dãy
- Điều đó cũng là đúng khi pivot luôn được chọn là phần tử đầu tiên, với giả thiết là dãy đầu vào là hoàn toàn ngẫu nhiên

Thời gian tính trung bình của QS

QuickSort Average Case

- Thời gian tính trung bình =

$\sum (\text{thời gian phân đoạn kích thước } i) \times (\text{xác suất phân đoạn có kích thước } i)$

- Trong tình huống ngẫu nhiên, tất cả các kích thước là đồng khả năng - xác suất chính là $1/N$

$$T(N) = T(i) + T(N - i - 1) + cN$$

$$E(T(N)) = \sum_{i=0}^{N-1} (1/N) [E(T(i)) + E(T(N - i - 1)) + cN]$$

$$E(T(N)) \leq (2/N) \sum_{i=0}^{N-1} [E(T(i)) + cN]$$

- Giải công thức đệ qui này ta thu được

$$E(T(N)) = O(N \log N).$$

NỘI DUNG

5.1. Bài toán sắp xếp

5.2. Ba thuật toán sắp xếp cơ bản

5.3. Sắp xếp trộn (Merge Sort)

5.4. Sắp xếp nhanh (Quick Sort)

5.5. Sắp xếp vun đống (Heap Sort)

5.5. Sắp xếp vun đống (Heap Sort)

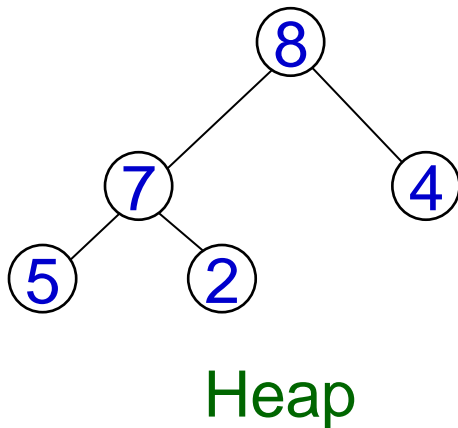
5.5.1. Cấu trúc dữ liệu đống (heap)

5.5.2. Sắp xếp vun đống

5.5.3. Hàng đợi có ưu tiên (priority queue)

5.5.1. The Heap Data Structure

- **Định nghĩa:** **Đống (heap)** là cây nhị phân gần hoàn chỉnh có 2 tính chất:
 - **Tính cấu trúc:** tất cả các mức đều là đầy, ngoại trừ mức cuối cùng, mức cuối được điền từ trái sang phải.
 - **Tính có thứ tự hay tính chất đống :** với mỗi nút x
 $\text{Parent}(x) \geq x$.
- Cây được cài đặt bởi mảng $A[i]$ có độ dài $\text{length}[A]$. Số lượng phần tử là $\text{heapsize}[A]$



Từ tính chất đống suy ra:

“Gốc chứa phần tử lớn nhất của đống!”

Như vậy có thể nói:

Đống là cây nhị phân được điền theo thứ tự

Biểu diễn đồng bởi mảng

- Đồng có thể cất giữ trong mảng A.

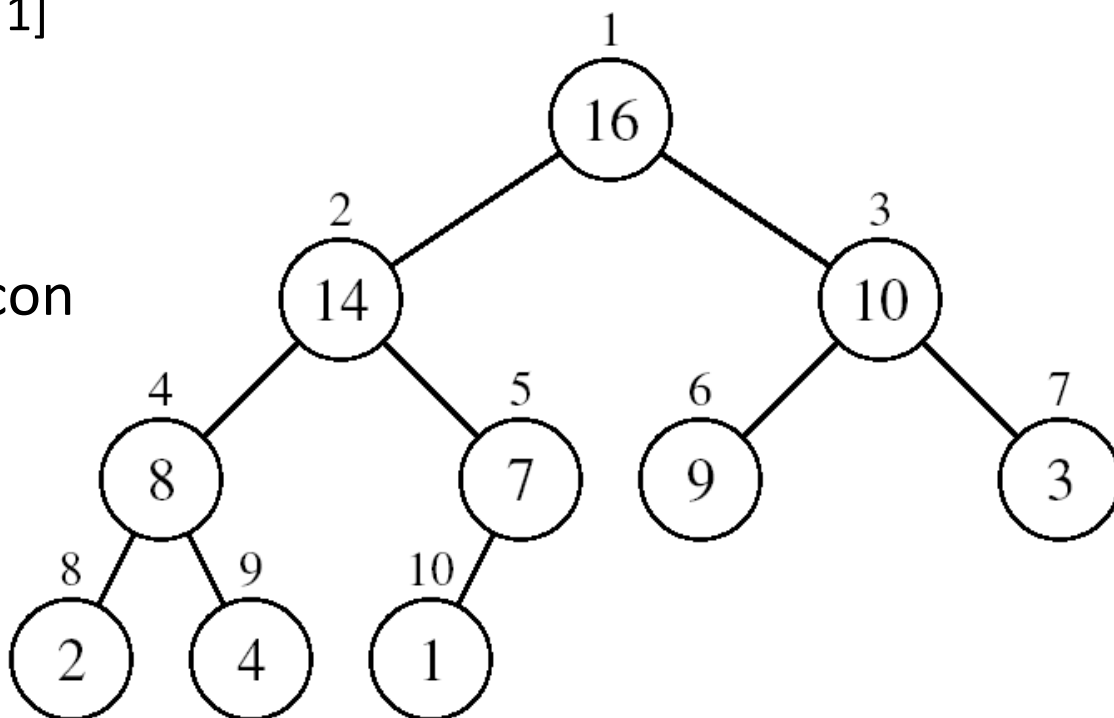
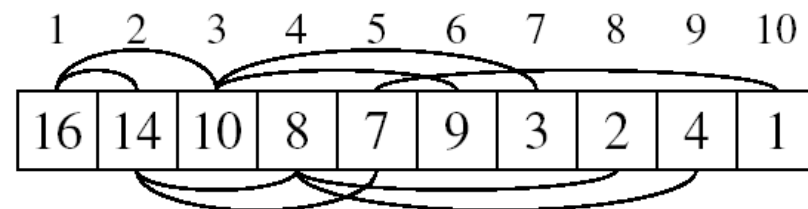
- Gốc của cây là $A[1]$
- Con trái của $A[i]$ là $A[2*i]$
- Con phải của $A[i]$ là $A[2*i + 1]$
- Cha của $A[i]$ là $A[\lfloor i/2 \rfloor]$
- $\text{Heapsize}[A] \leq \text{length}[A]$

- Các phần tử trong mảng con $A[(\lfloor n/2 \rfloor + 1) .. n]$ là các lá

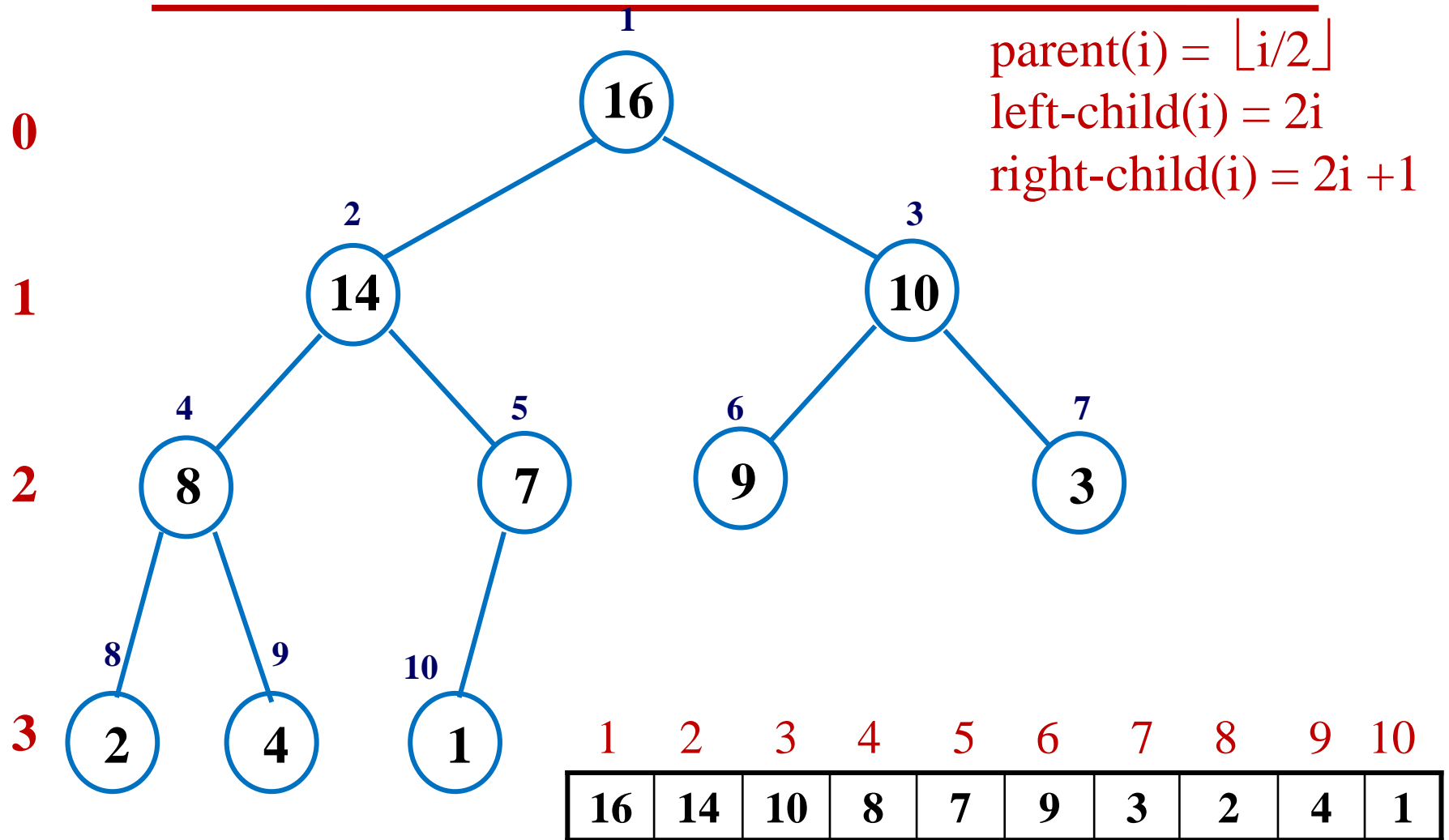
$$\text{parent}(i) = \lfloor i/2 \rfloor$$

$$\text{left-child}(i) = 2i$$

$$\text{right-child}(i) = 2i + 1$$



Ví dụ đồng



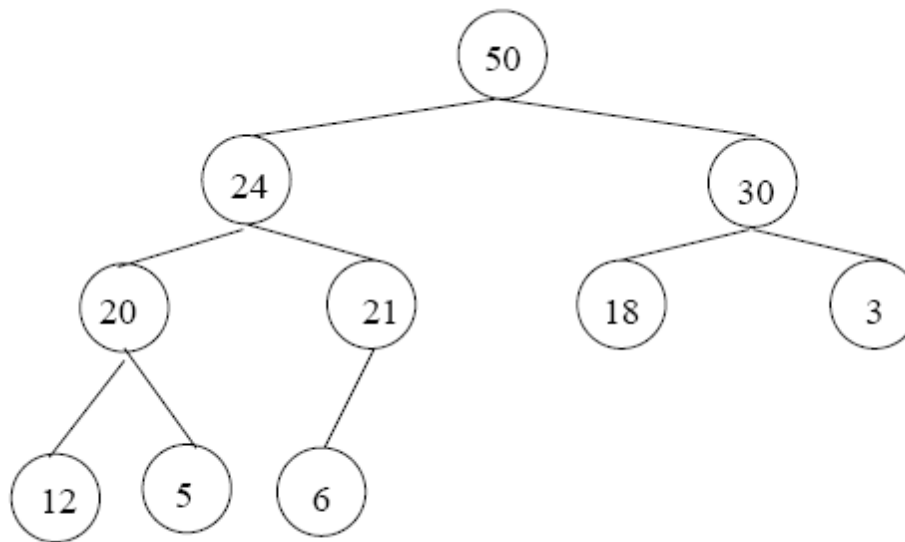
Hai dạng đống

- **Đống max - Max-heaps** (Phần tử lớn nhất ở gốc), có tính chất *max-heap*:
 - với mọi nút i , ngoại trừ gốc:
$$A[\text{parent}(i)] \geq A[i]$$
- **Đống min - Min-heaps** (phần tử nhỏ nhất ở gốc), có tính chất *min-heap*:
 - với mọi nút i , ngoại trừ gốc:
$$A[\text{parent}(i)] \leq A[i]$$
- Phần dưới đây ta sẽ chỉ xét đống max (max-heap). Đống min được xét hoàn toàn tương tự.

Bổ sung và loại bỏ nút

Adding/Deleting Nodes

- Nút mới được bổ sung vào mức đáy (từ trái sang phải)
- Các nút được loại bỏ khỏi mức đáy (từ phải sang trái)



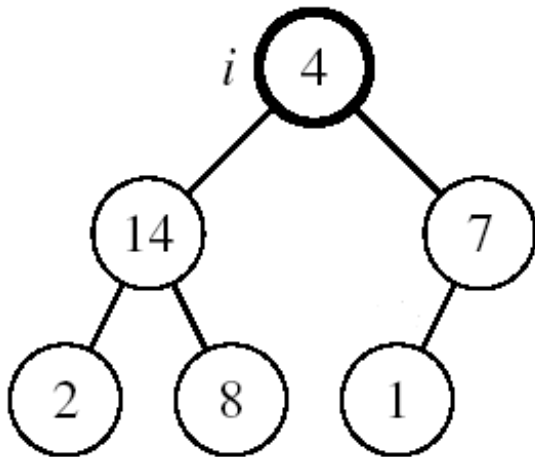
Các phép toán đối với đống

Operations on Heaps

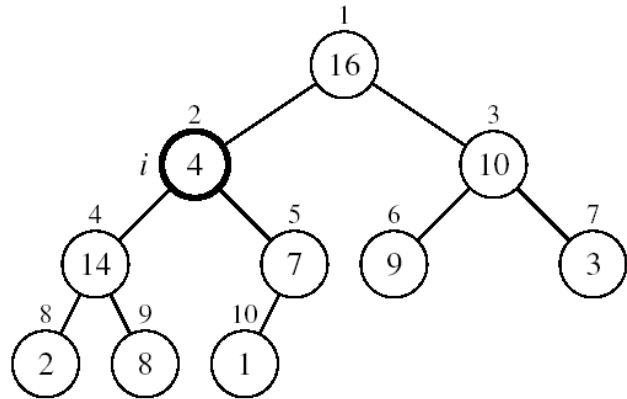
- Khôi phục tính chất max-heap (Vun lại đống)
 - Max-Heapify
- Tạo max-heap từ một mảng không được sắp xếp
 - Build-Max-Heap

Khôi phục tính chất đồng

- Giả sử có nút i với giá trị bé hơn con của nó
 - Giả thiết là: Cây con trái và Cây con phải của i đều là max-heaps
- Để loại bỏ sự vi phạm này ta tiến hành như sau:
 - Đổi chỗ với con lớn hơn
 - Di chuyển xuống theo cây
 - Tiếp tục quá trình cho đến khi nút không còn bé hơn con

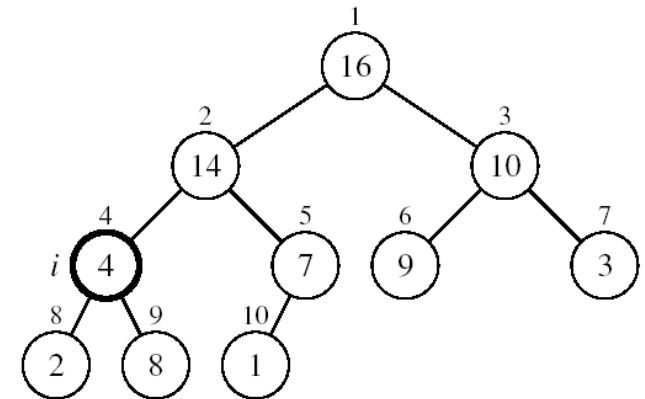


Ví dụ



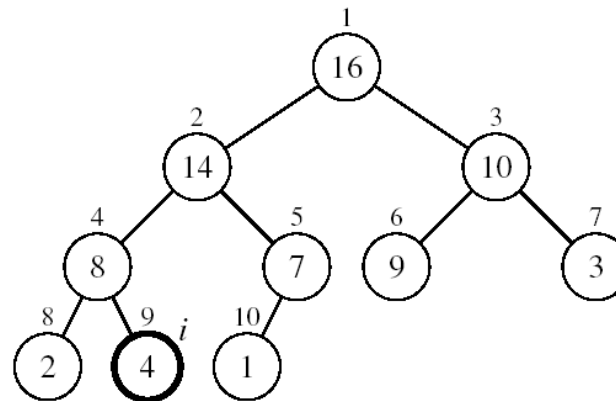
$A[2]$ vi phạm tính chất đồng

$A[2] \leftrightarrow A[4]$



$A[4]$ vi phạm

$A[4] \leftrightarrow A[9]$

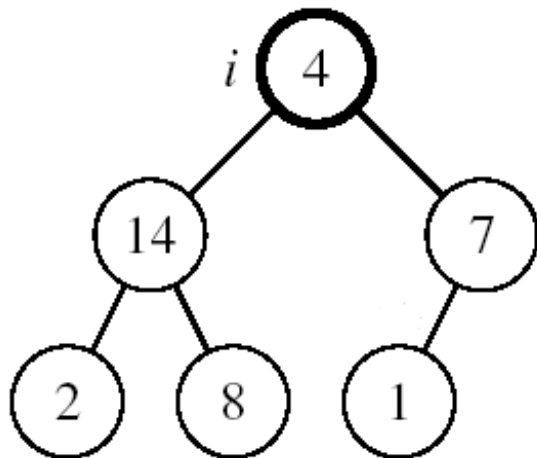


Tính chất đồng được khôi phục

Thuật toán khôi phục tính chất đồng

Giả thiết:

- Cả hai cây con trái và phải của i đều là max-heaps
- $A[i]$ có thể bé hơn các con của nó



Max-Heapify(A, i, n)

// $n = \text{heapsize}[A]$

- $l \leftarrow \text{left-child}(i)$
- $r \leftarrow \text{right-child}(i)$
- if** $(l \leq n)$ and $(A[l] > A[i])$
- then** $\text{largest} \leftarrow l$
- else** $\text{largest} \leftarrow i$
- if** $(r \leq n)$ and $(A[r] > A[\text{largest}])$
- then** $\text{largest} \leftarrow r$
- if** $\text{largest} \neq i$
- then** $\text{Exchange}(A[i], A[\text{largest}])$
- $\text{Max-Heapify}(A, \text{largest}, n)$

Thời gian tính của MAX-HEAPIFY

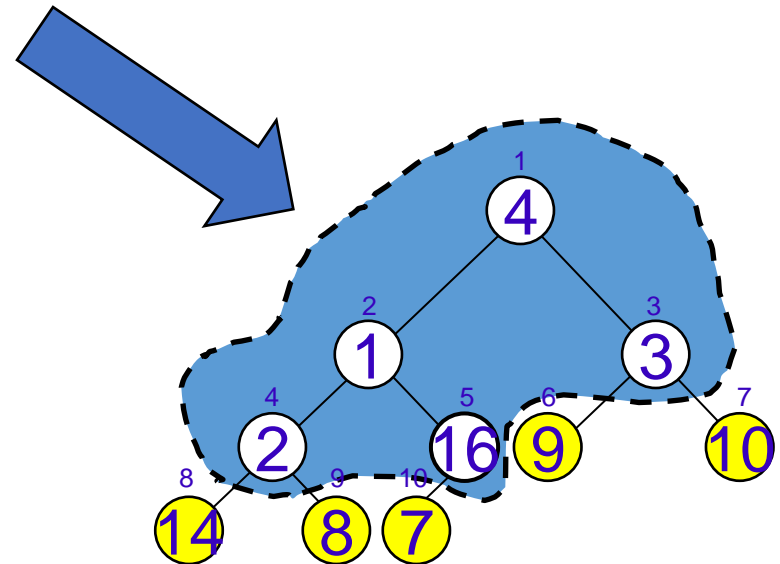
- Ta nhận thấy rằng:
 - Từ nút i phải di chuyển theo đường đi xuống phía dưới của cây. Độ dài của đường đi này không vượt quá độ dài đường đi từ gốc đến lá, nghĩa là không vượt quá h .
 - Ở mỗi mức phải thực hiện 2 phép so sánh.
 - Do đó tổng số phép so sánh không vượt quá $2h$.
 - Vậy, thời gian tính là $O(h)$ hay $O(\log n)$.
- **Kết luận: Thời gian tính của MAX-HEAPIFY là $O(\log n)$**
- Nếu viết trong ngôn ngữ chiều cao của đống, thì thời gian này là $O(h)$

Xây dựng đống (Building a Heap)

- Biến đổi mảng $A[1 \dots n]$ thành max-heap ($n = \text{length}[A]$)
- Vì các phần tử của mảng con $A[(\lfloor n/2 \rfloor + 1) \dots n]$ là các lá
- Do đó để tạo đống ta chỉ cần áp dụng MAX-HEAPIFY đối với các phần tử từ 1 đến $\lfloor n/2 \rfloor$

Alg: Build-Max-Heap(A)

1. $n = \text{length}[A]$
2. **for** $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1
3. **do** Max-Heappify(A, i, n)

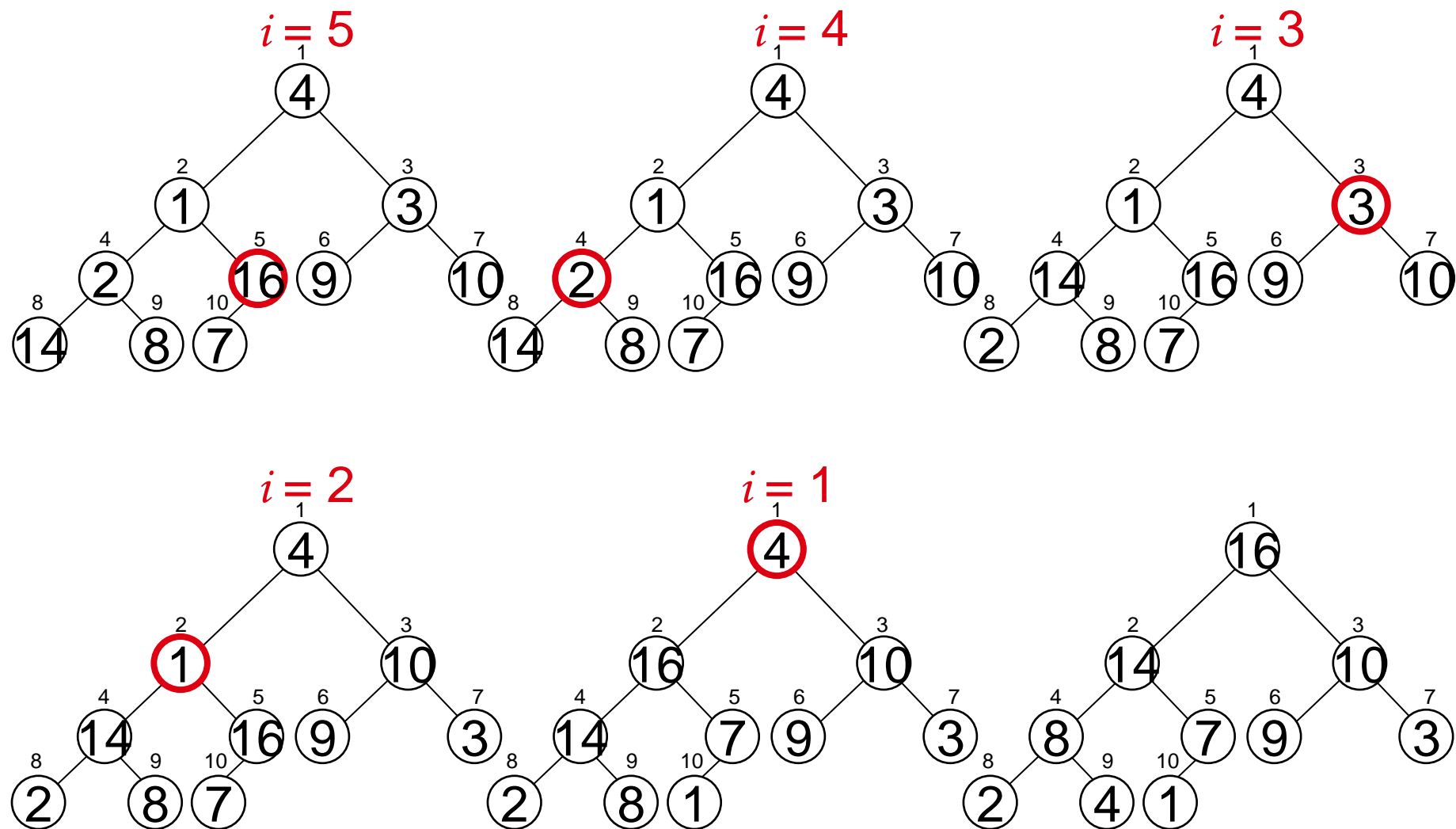


A:

4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---

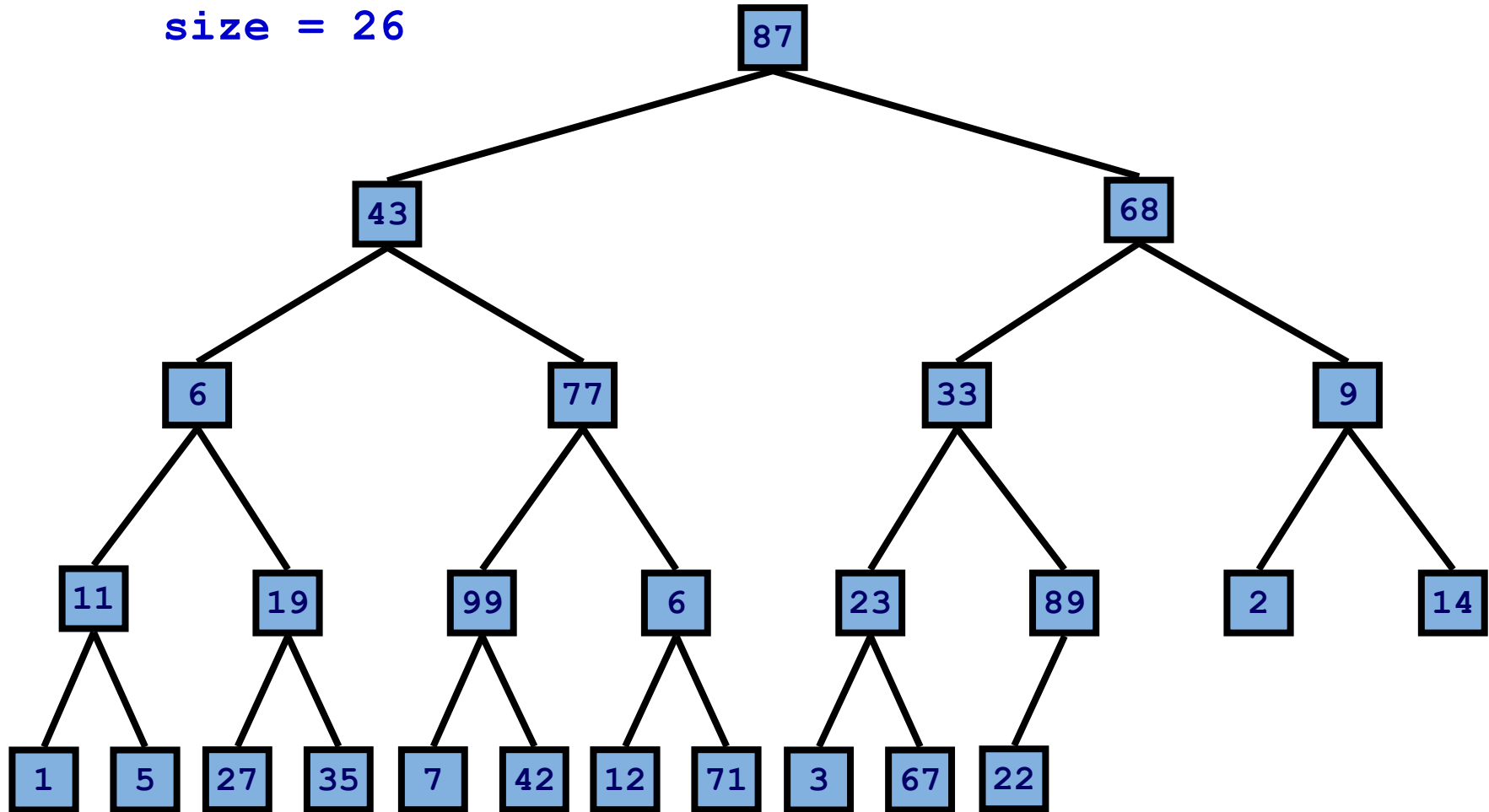
Ví dụ: A:

4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---



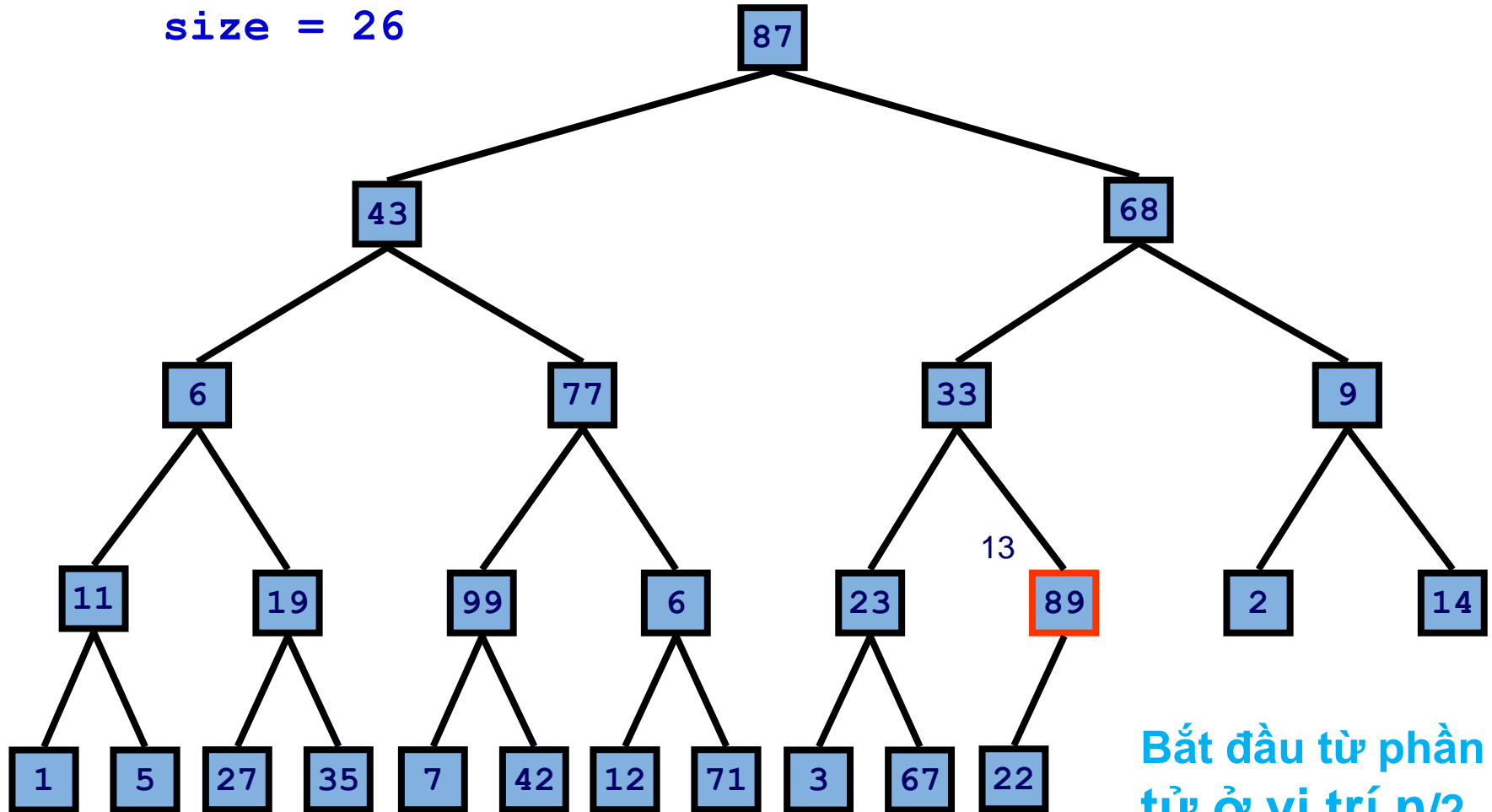
Ví dụ: Build-Min-Heap

size = 26



Ví dụ: Build-Min-Heap

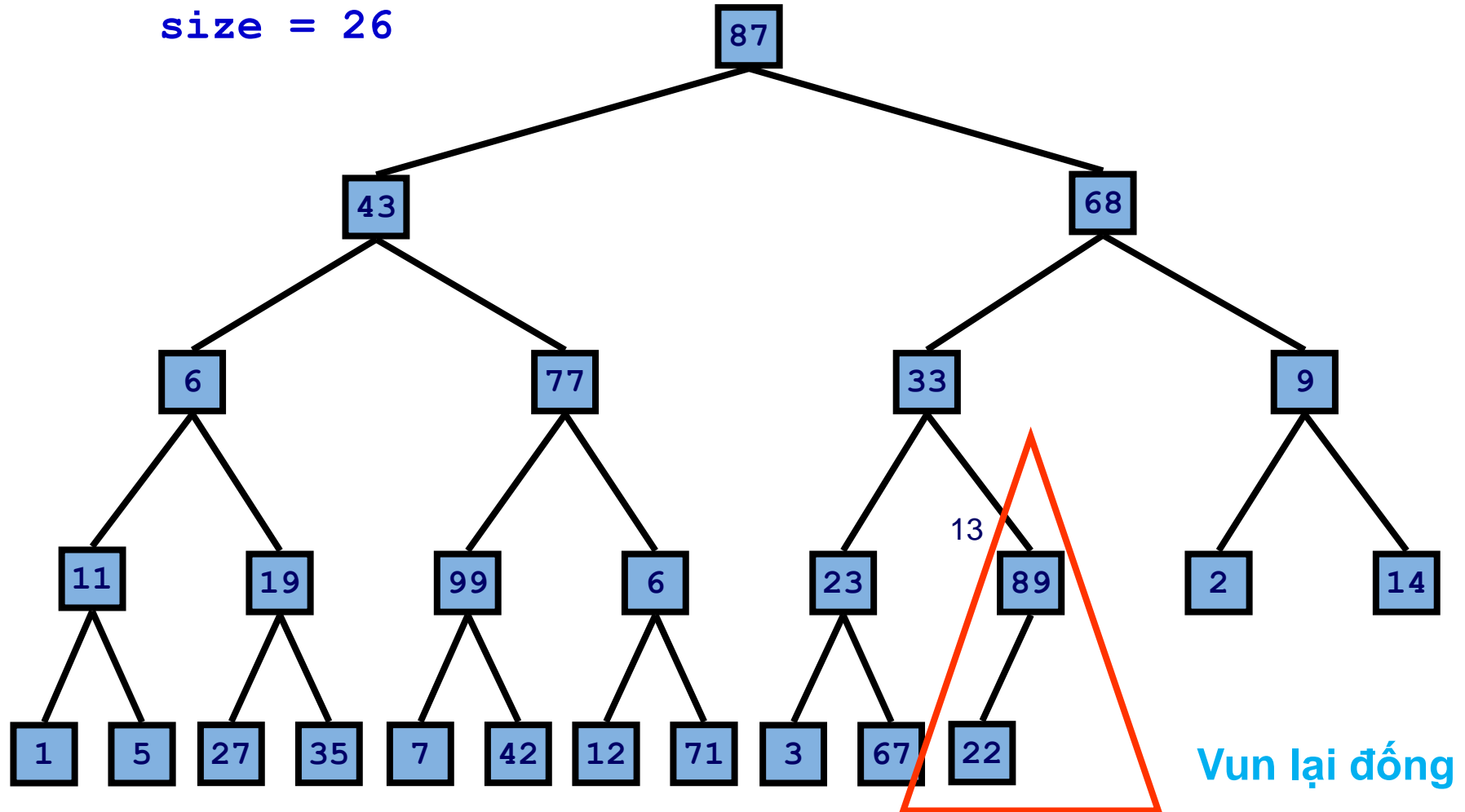
size = 26



Bắt đầu từ phần tử ở vị trí $n/2$

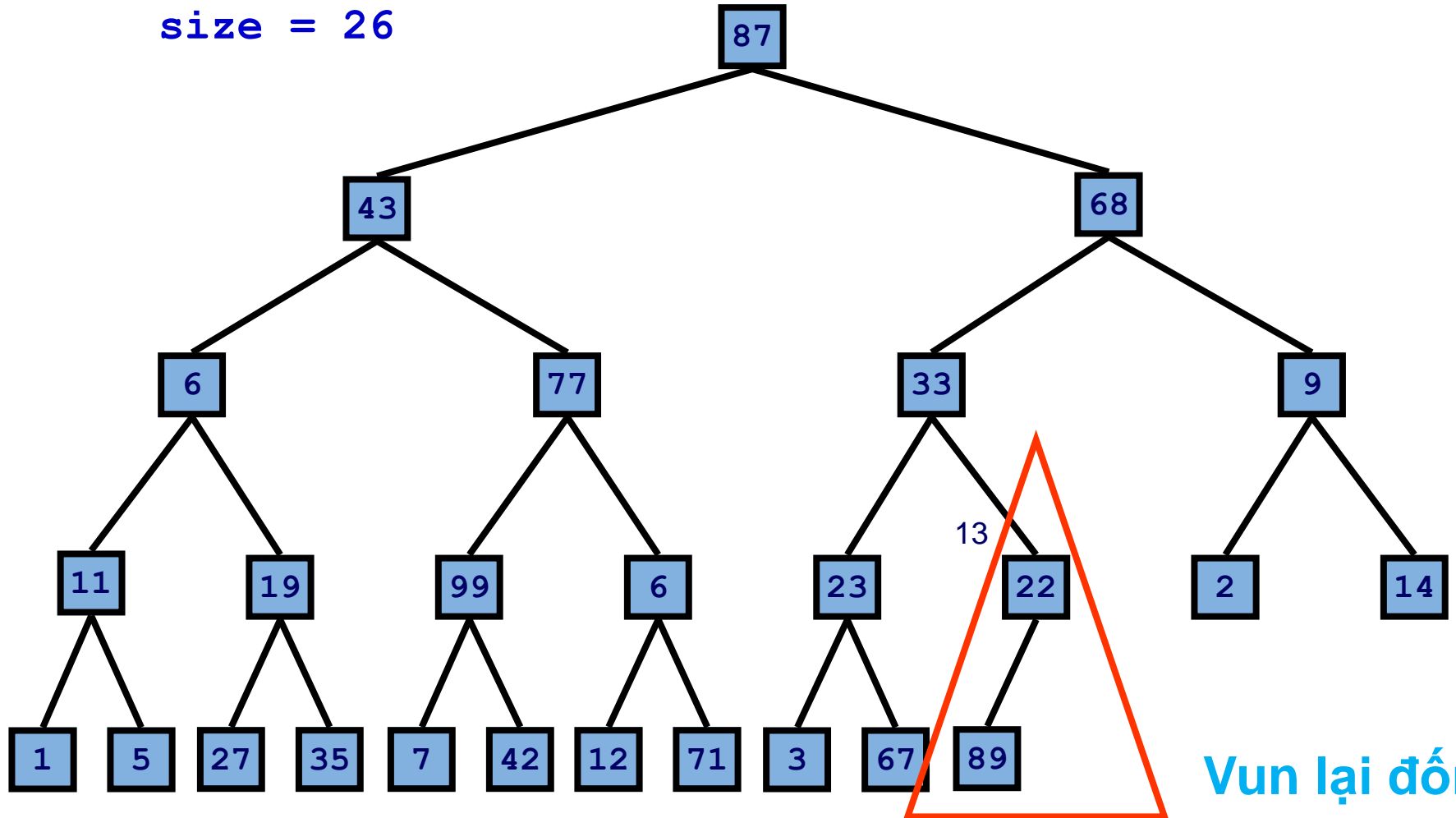
Build-Min-Heap

size = 26



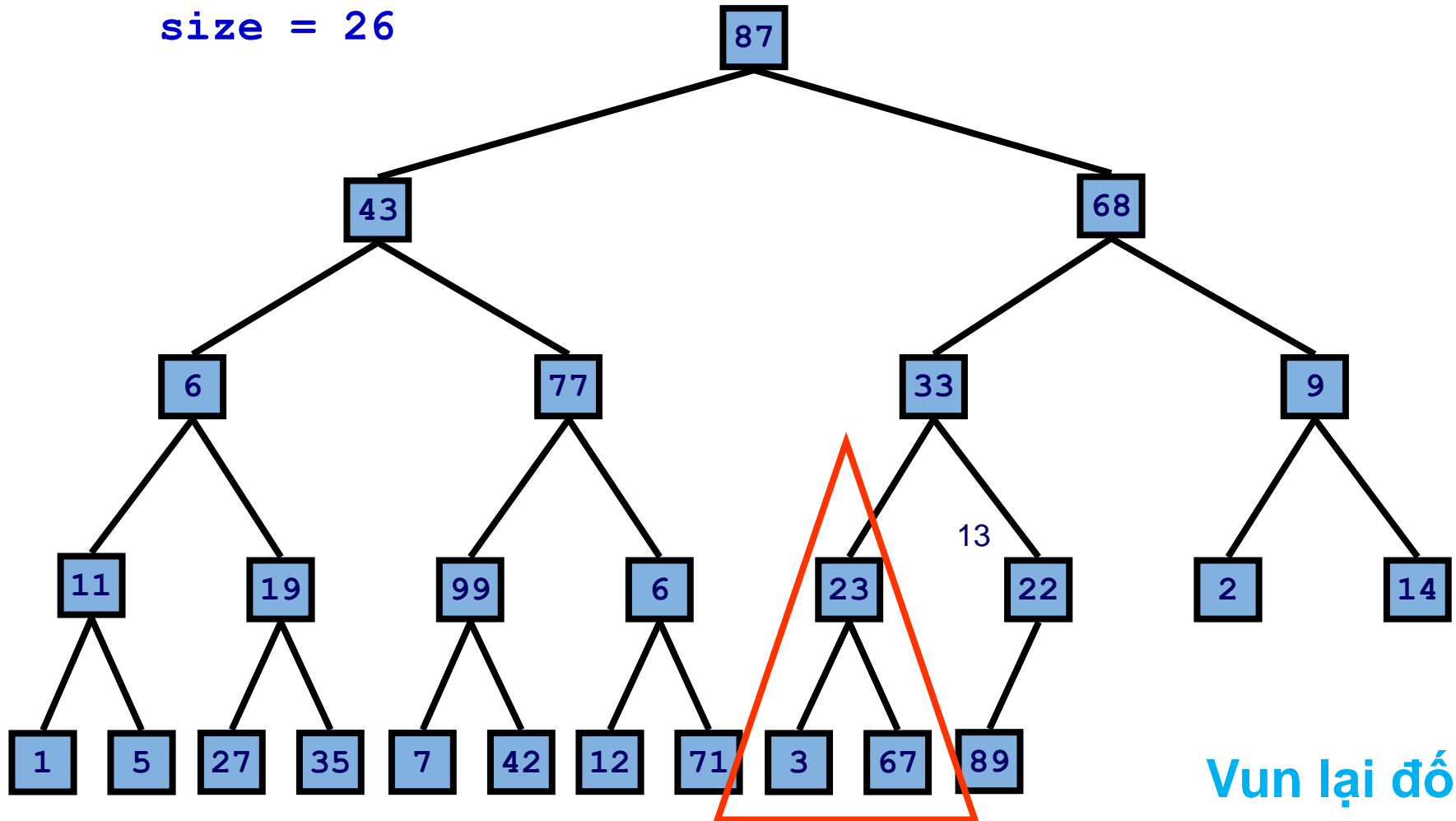
Build-Min-Heap

size = 26



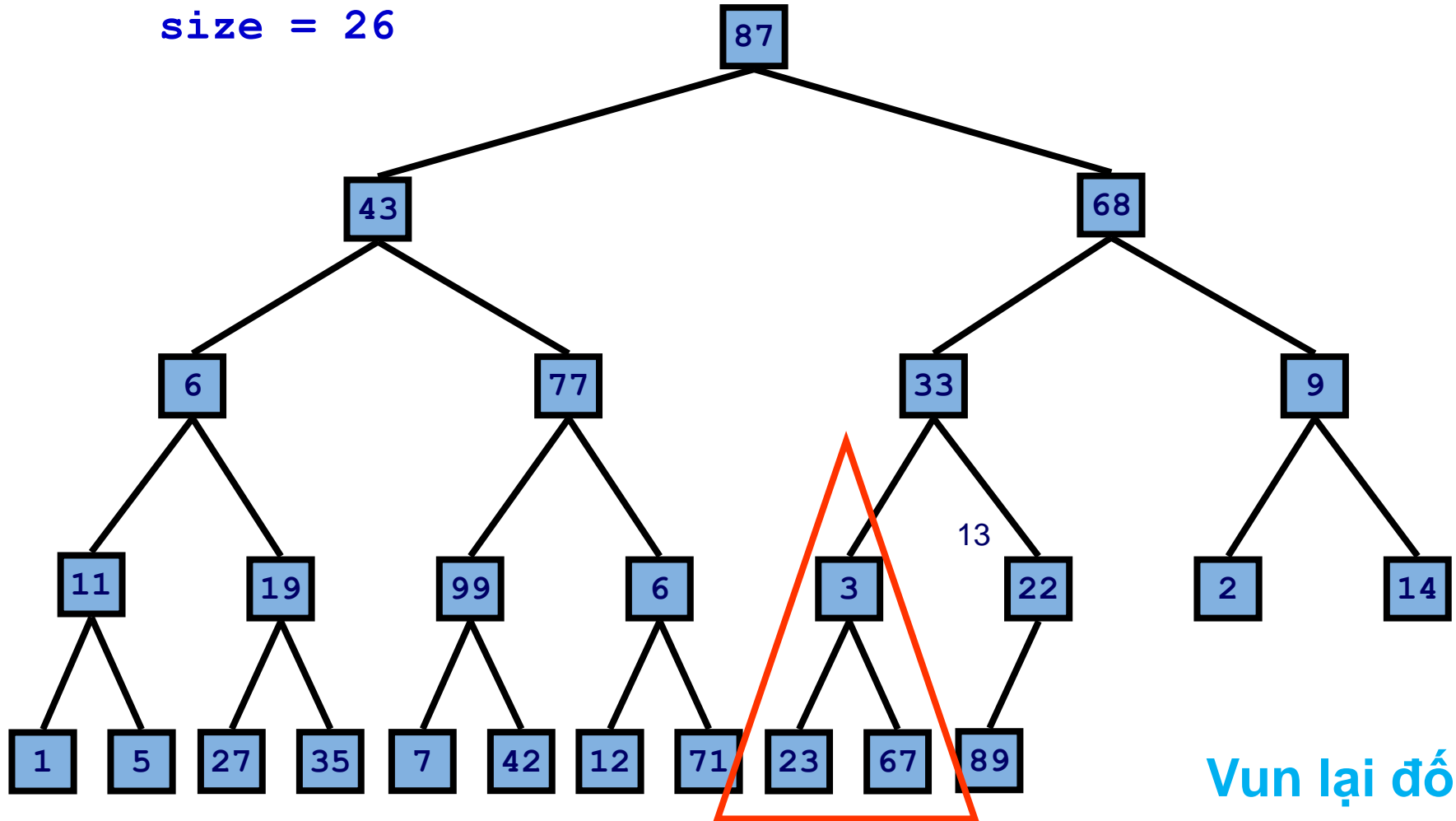
Build-Min-Heap

size = 26



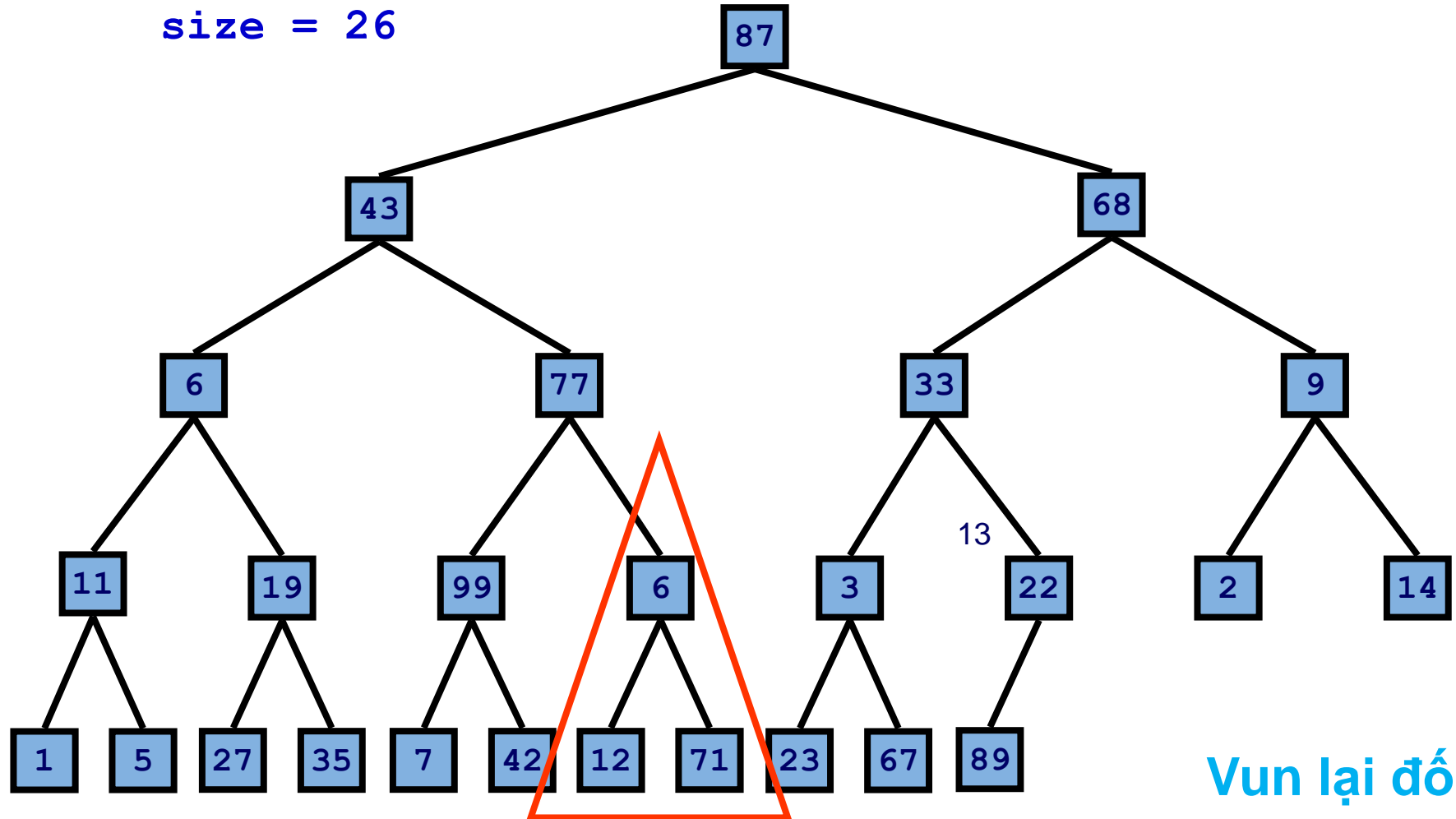
Build-Min-Heap

size = 26



Build-Min-Heap

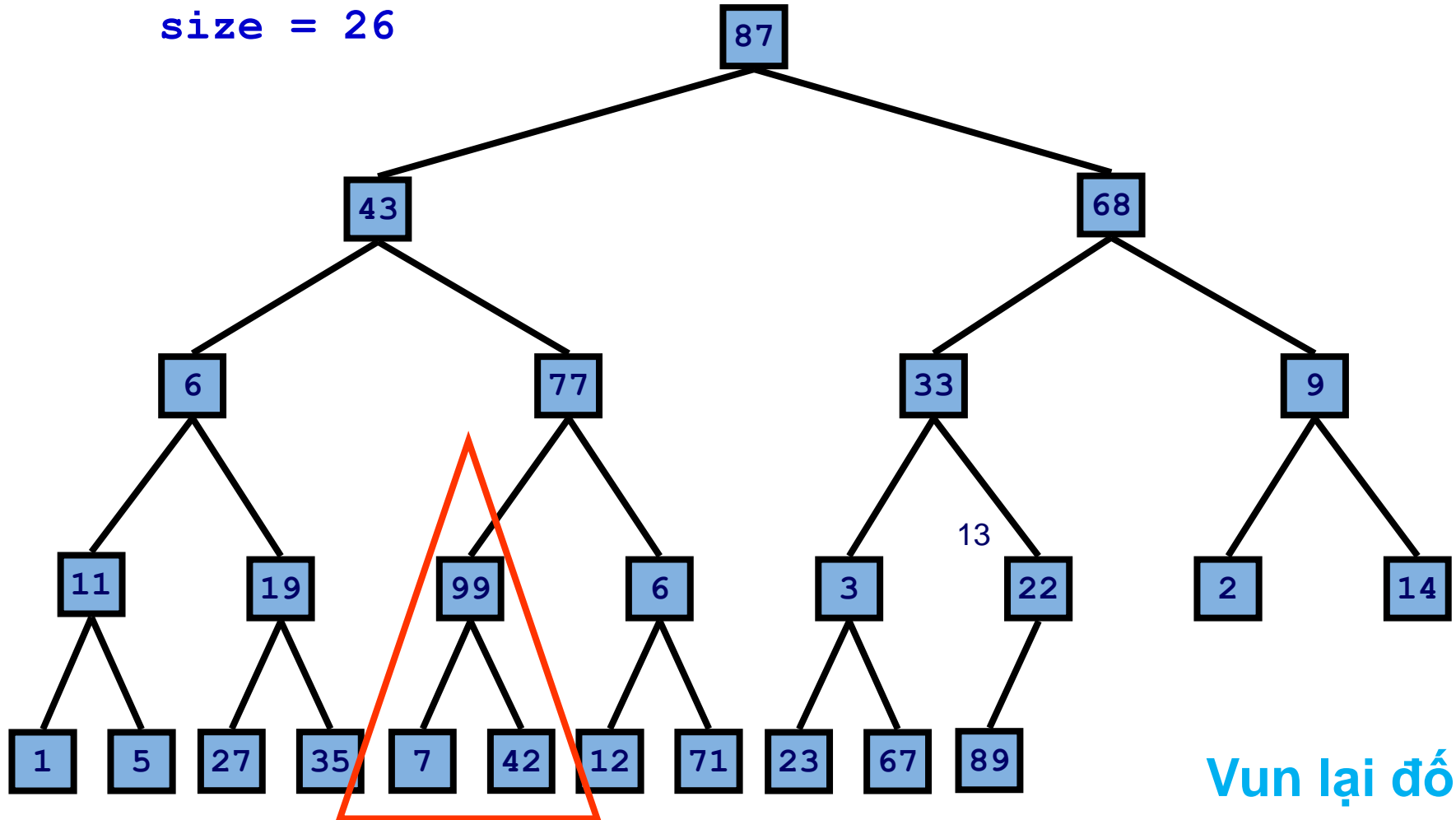
size = 26



Vun lại đồng

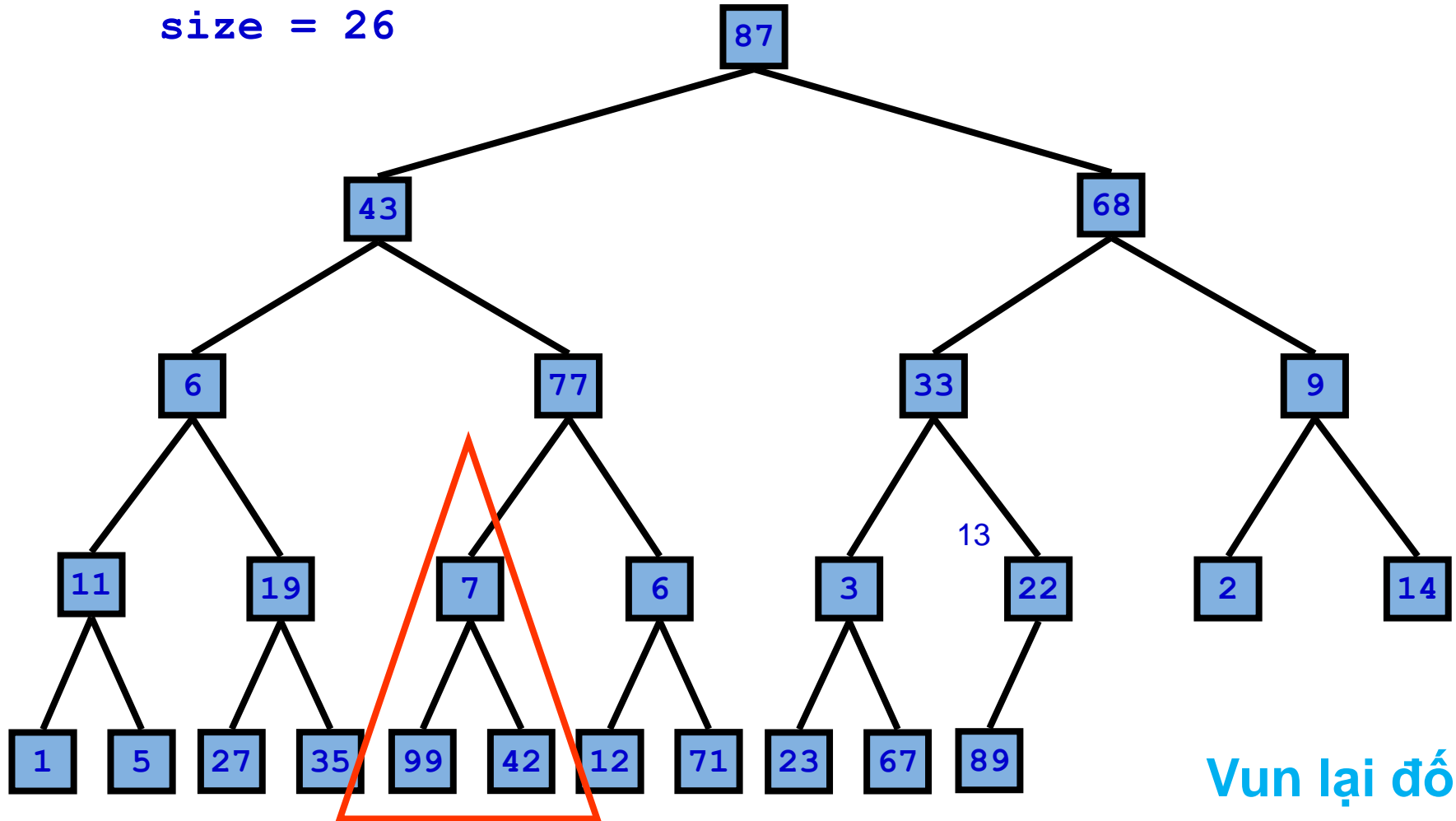
Build-Min-Heap

size = 26



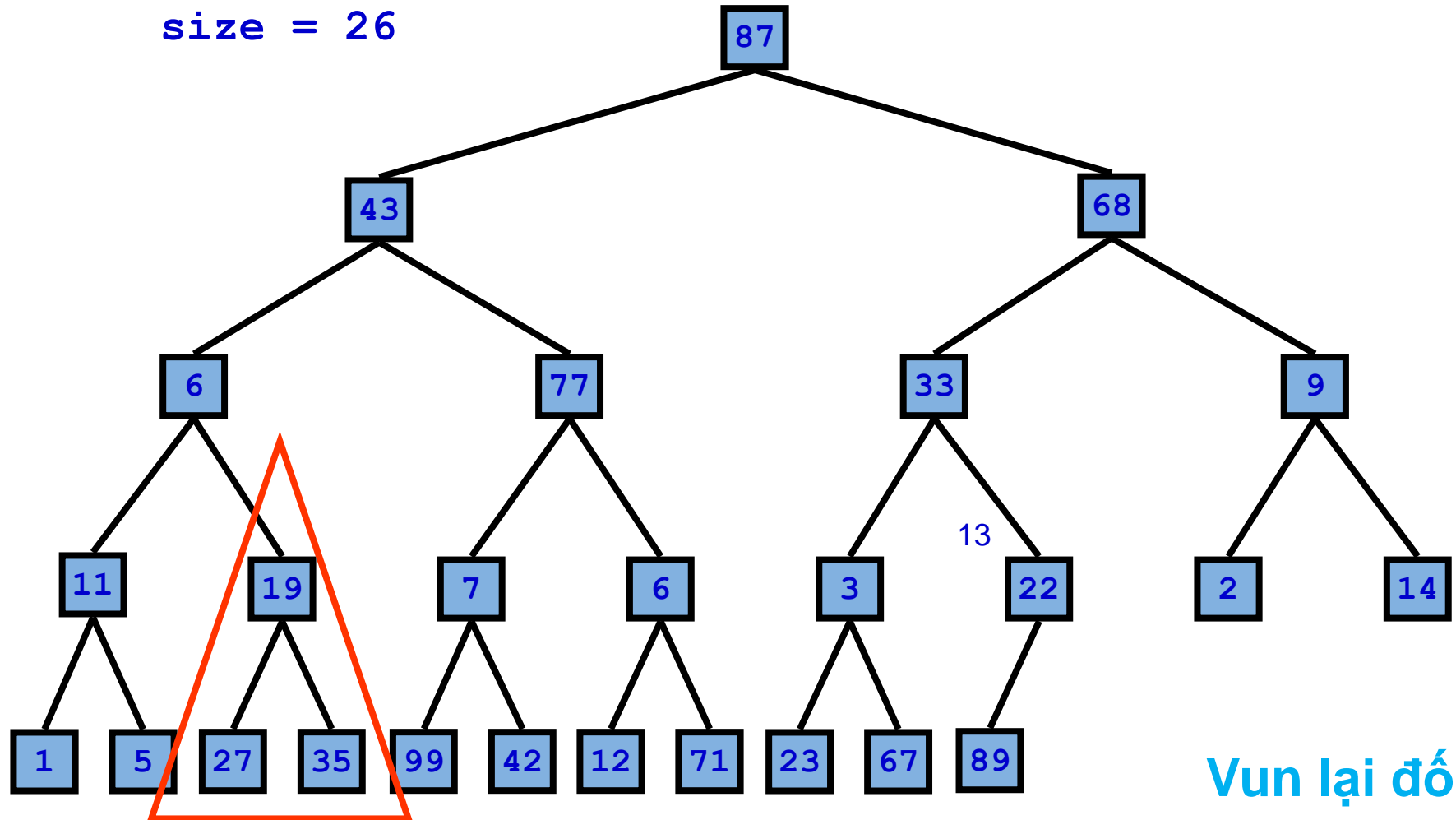
Build-Min-Heap

size = 26



Build-Min-Heap

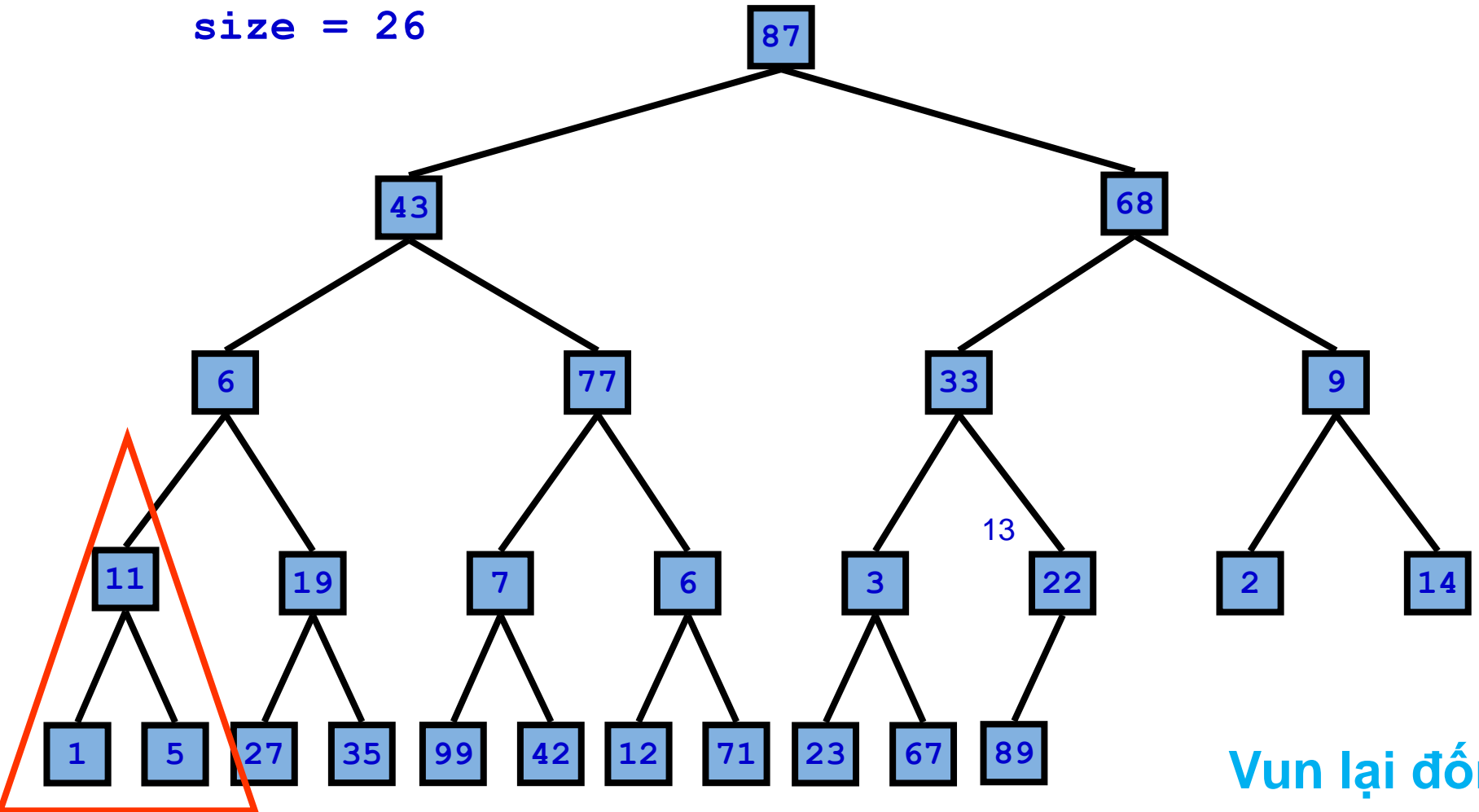
size = 26



Vun lại đồng

Build-Min-Heap

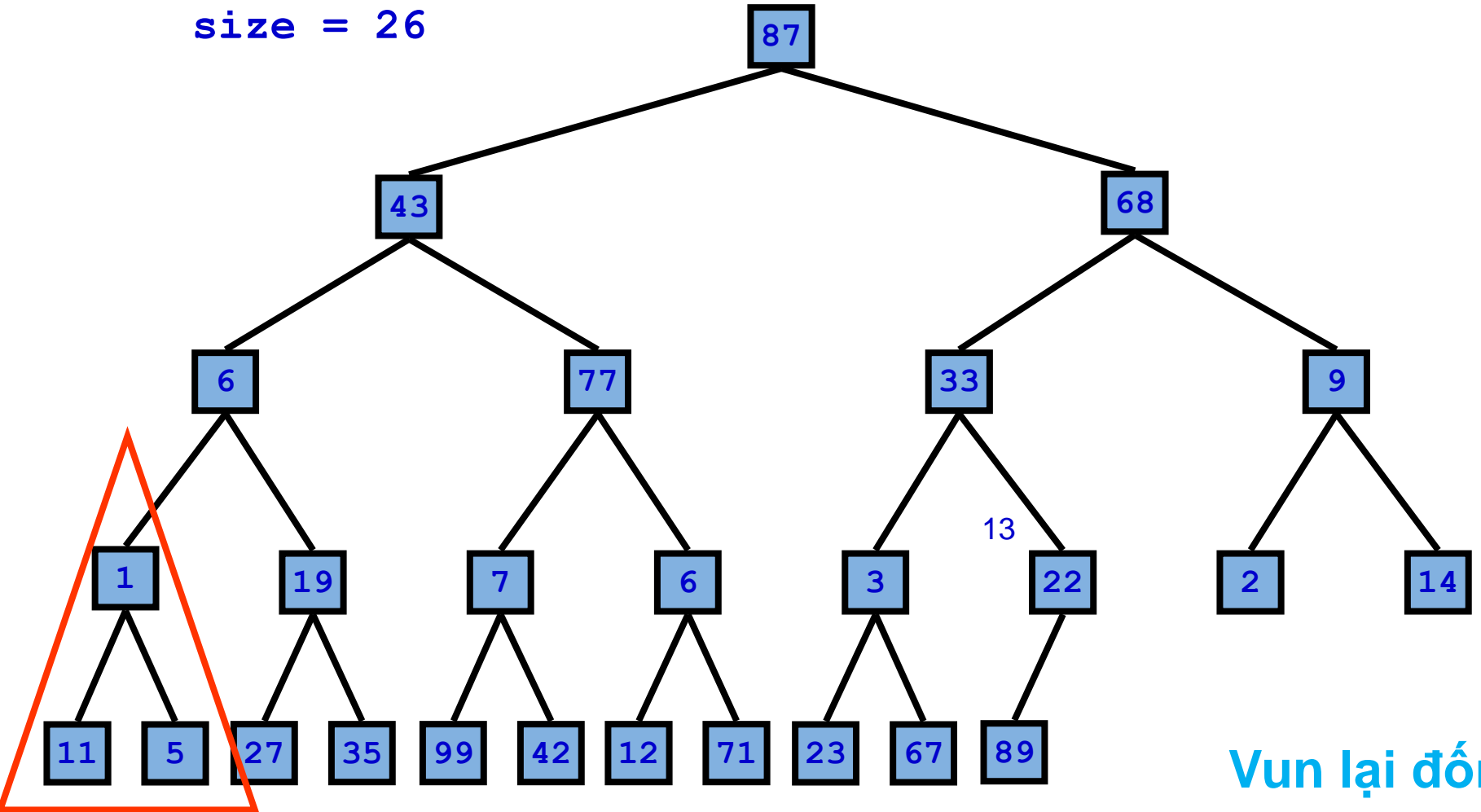
size = 26



Vun lại đồng

Build-Min-Heap

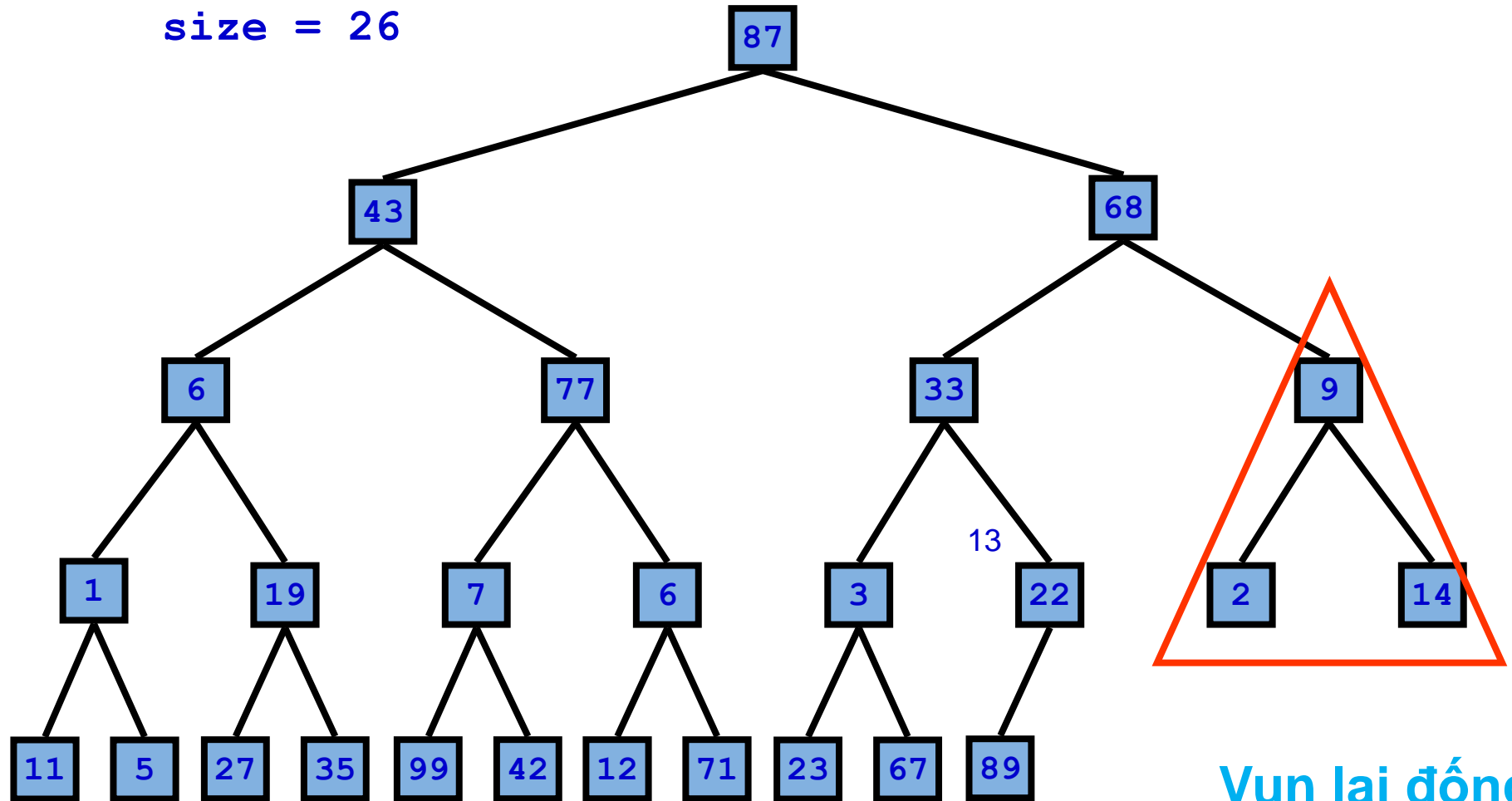
size = 26



Vun lại đồng

Build-Min-Heap

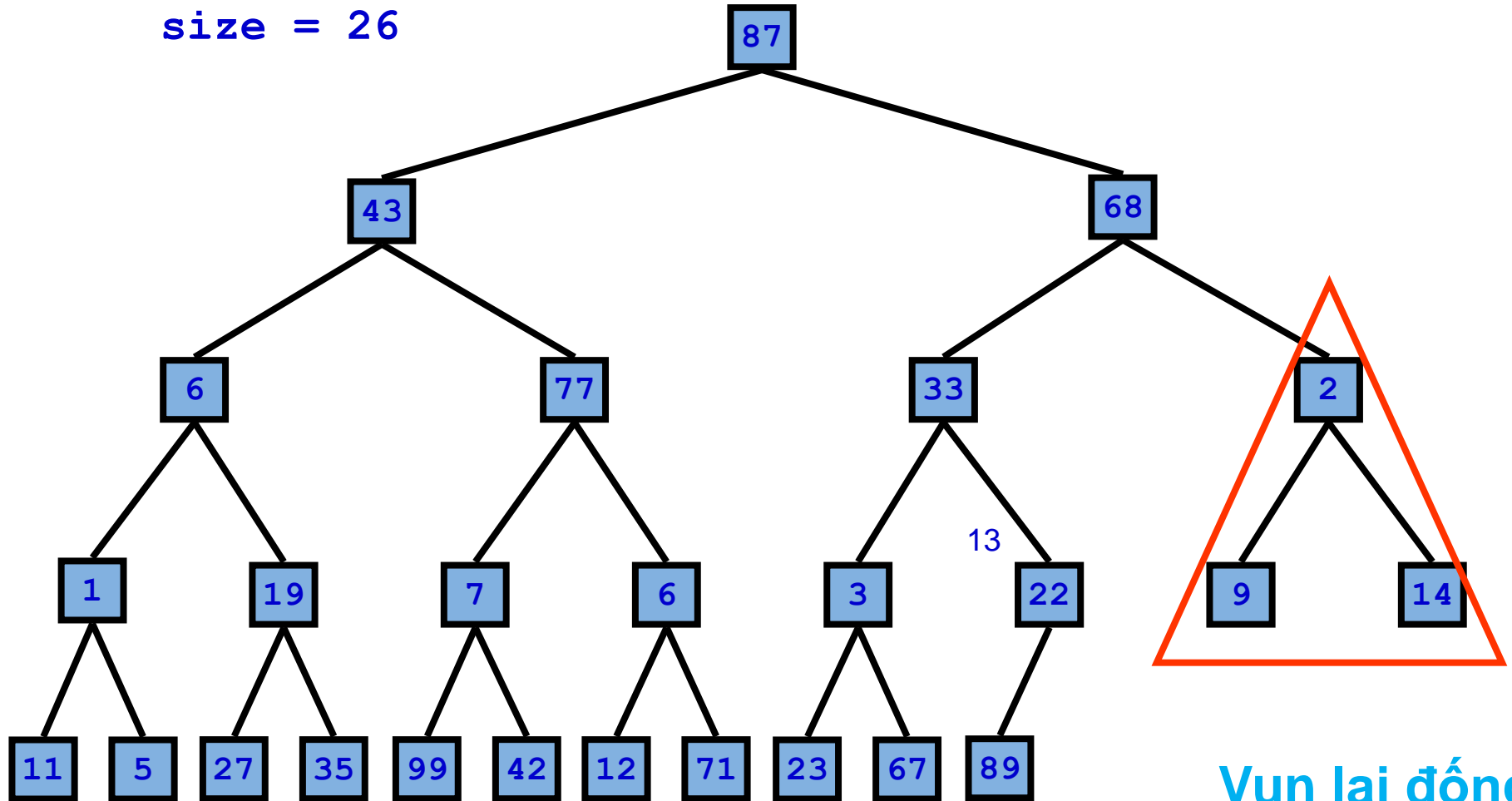
size = 26



Vun lại đồng

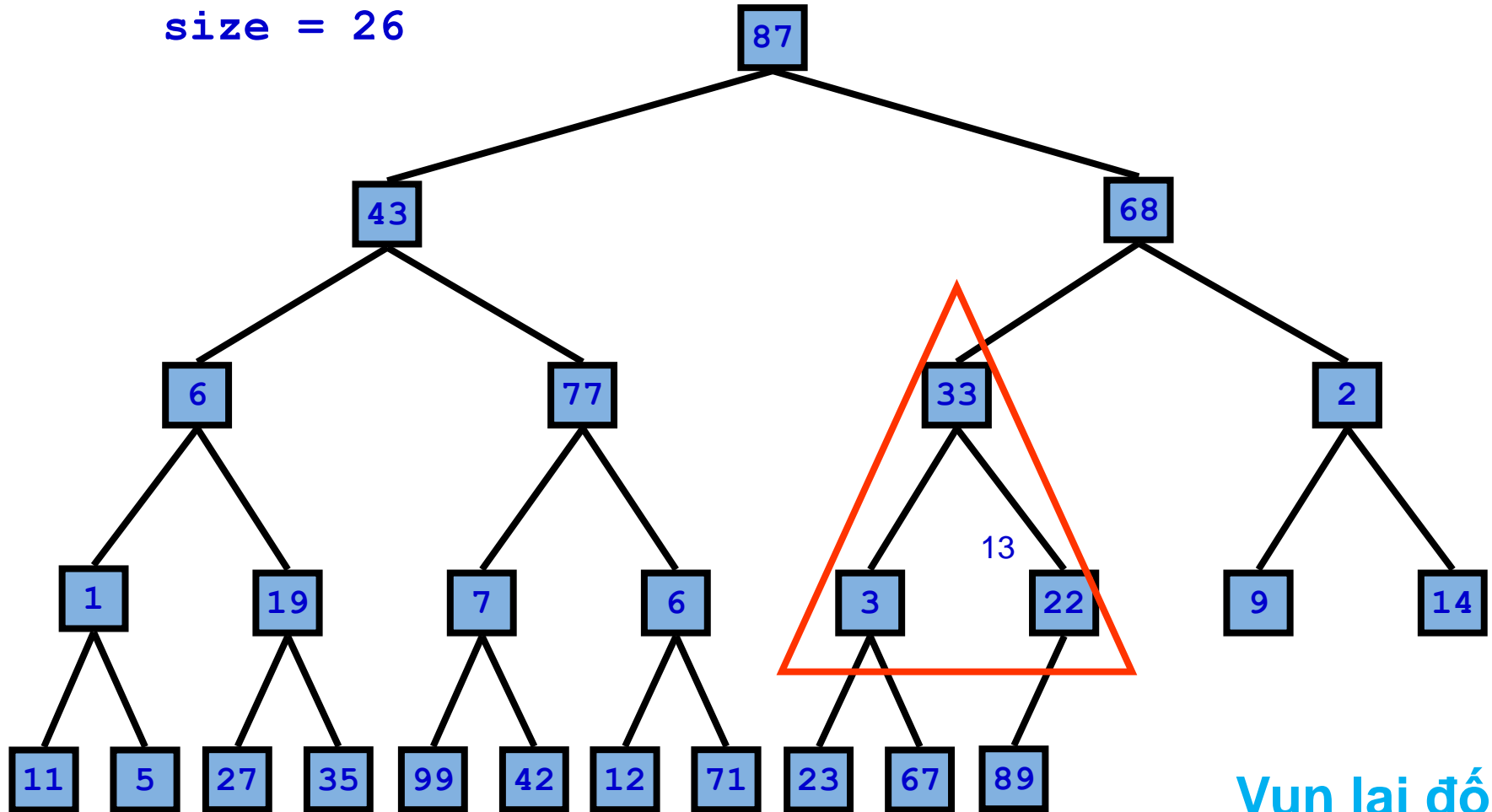
Build-Min-Heap

size = 26



Build-Min-Heap

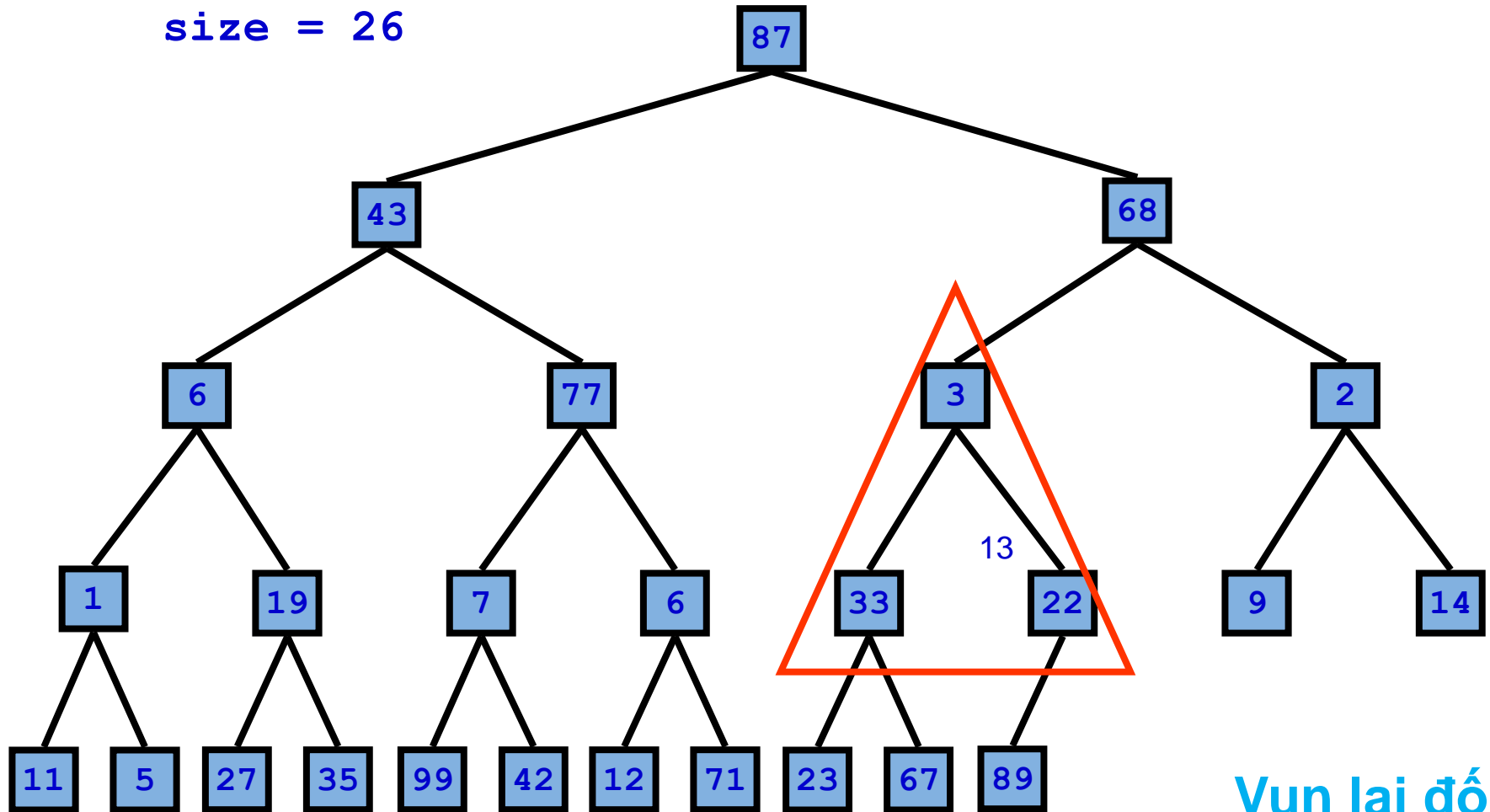
size = 26



Vun lại đồng

Build-Min-Heap

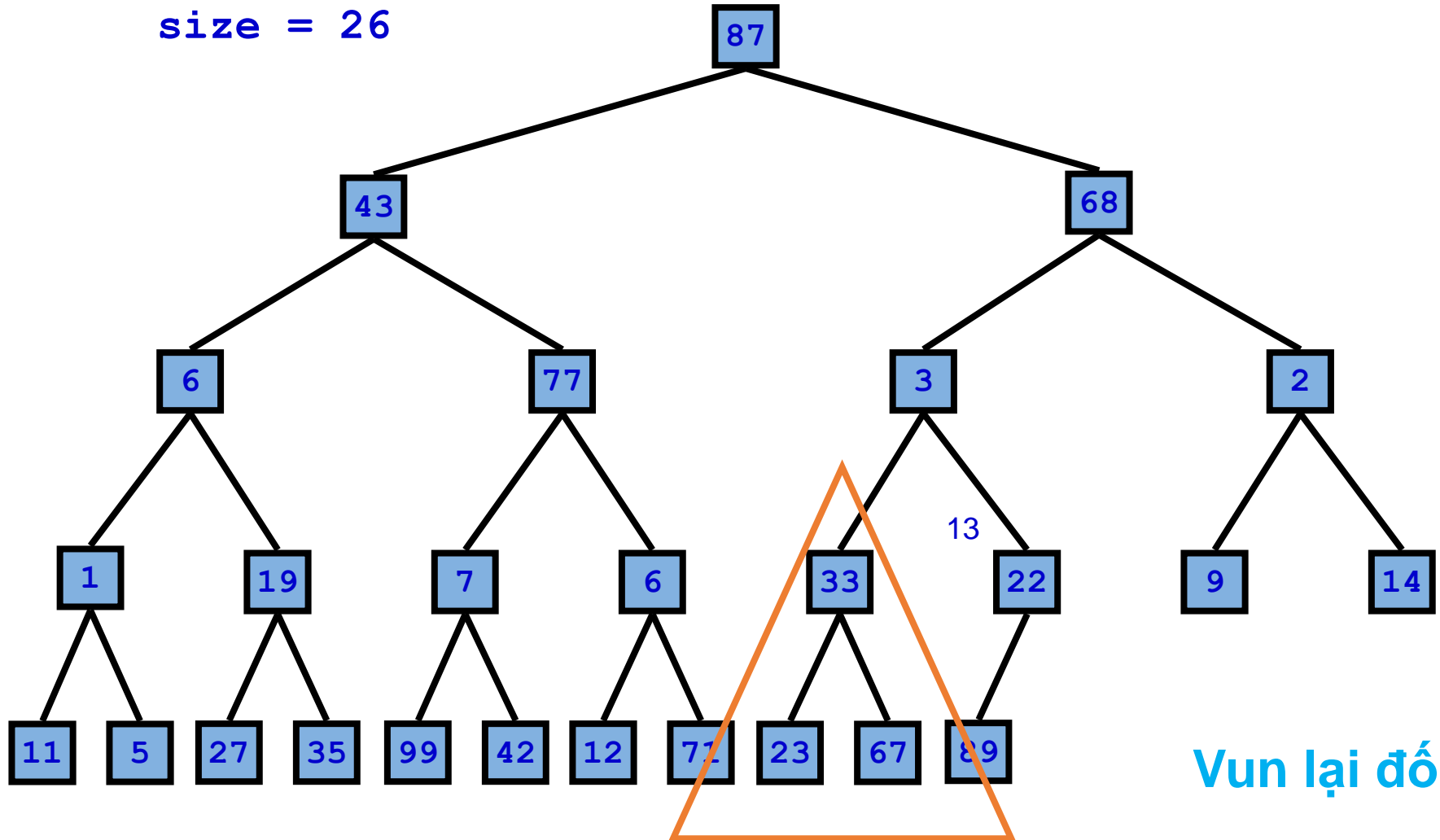
size = 26



Vun lại đồng

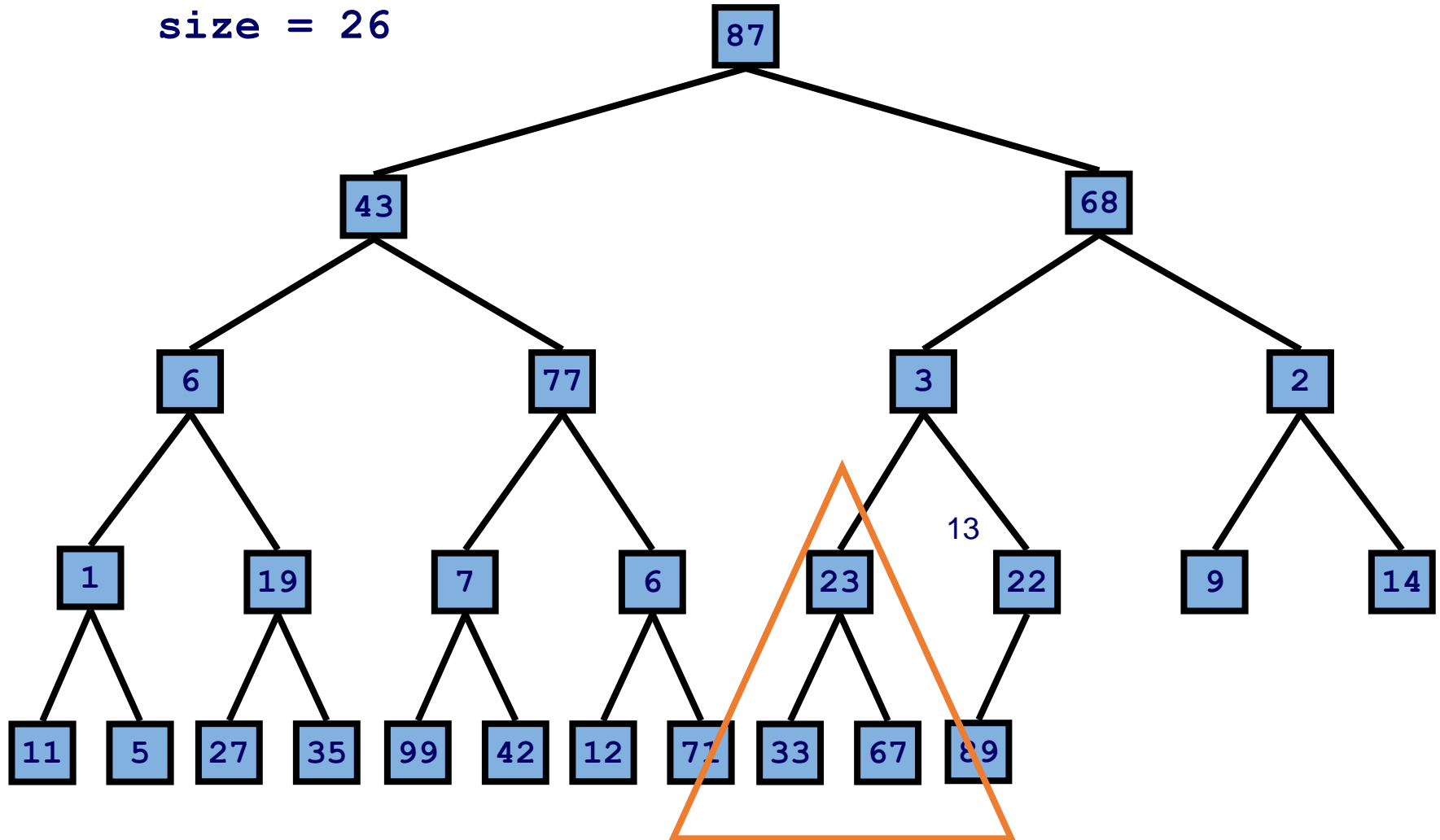
Build-Min-Heap

size = 26



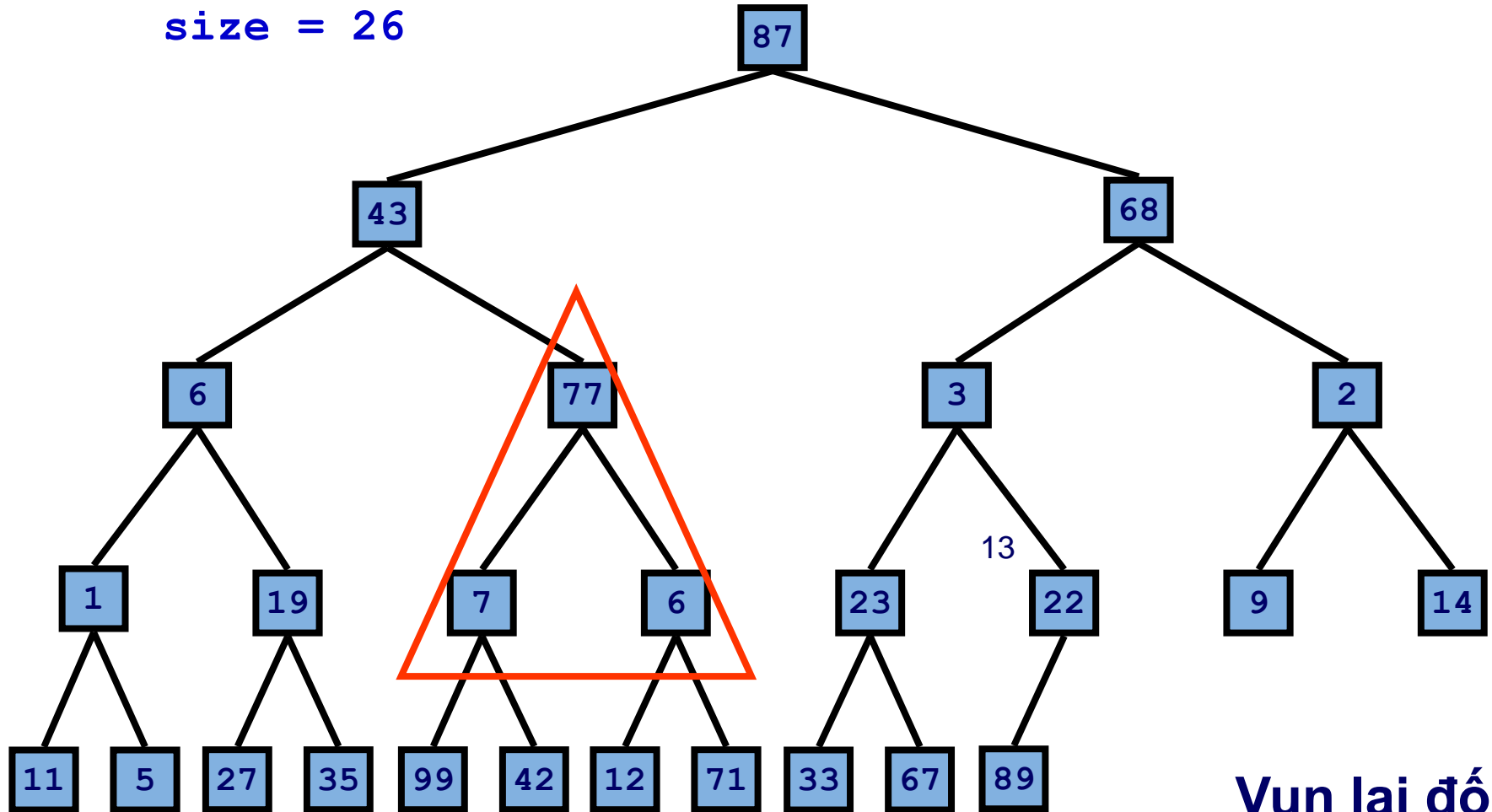
Build-Min-Heap

size = 26



Build-Min-Heap

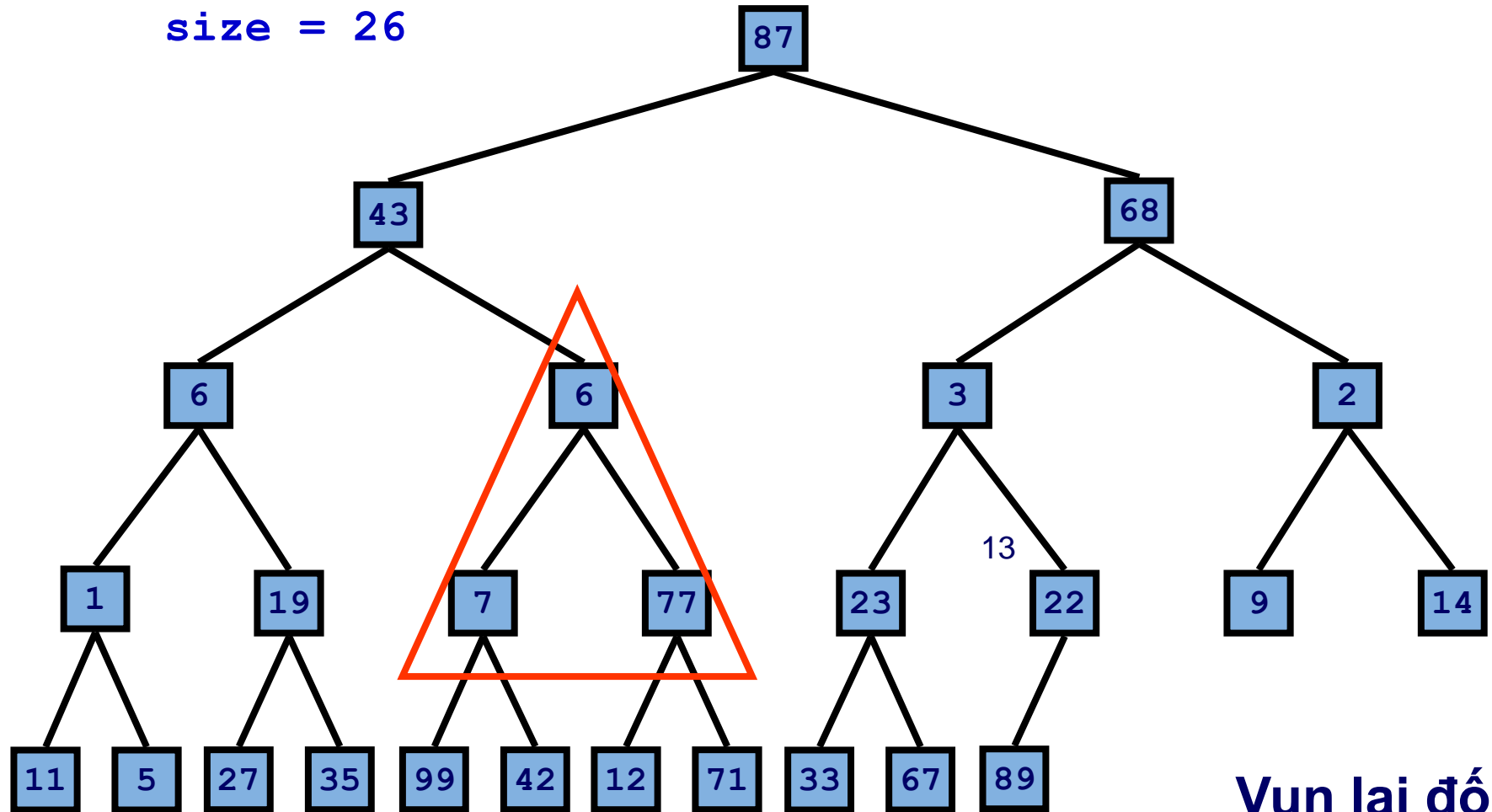
size = 26



Vun lại đồng

Build-Min-Heap

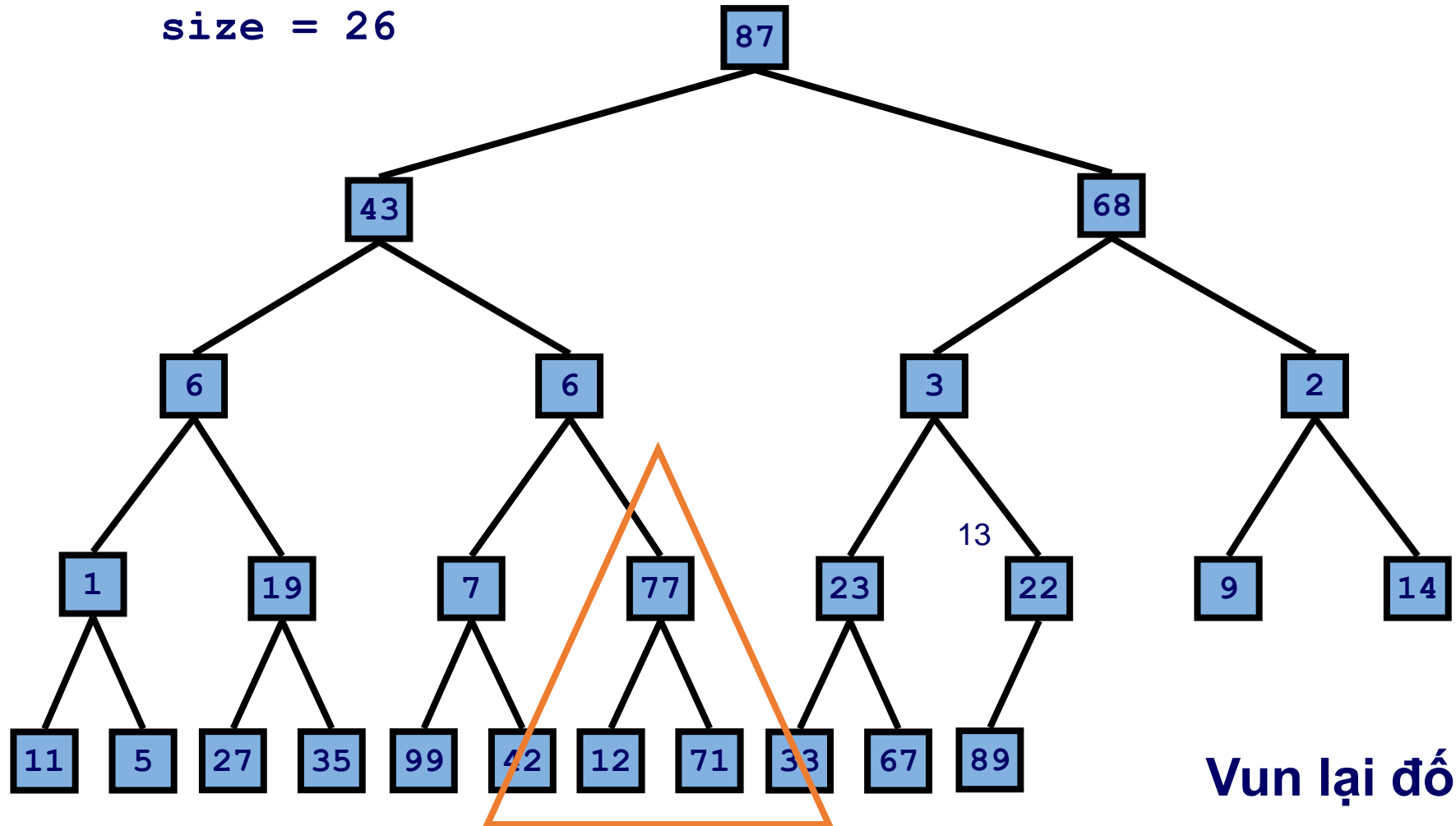
size = 26



Vun lại đồng

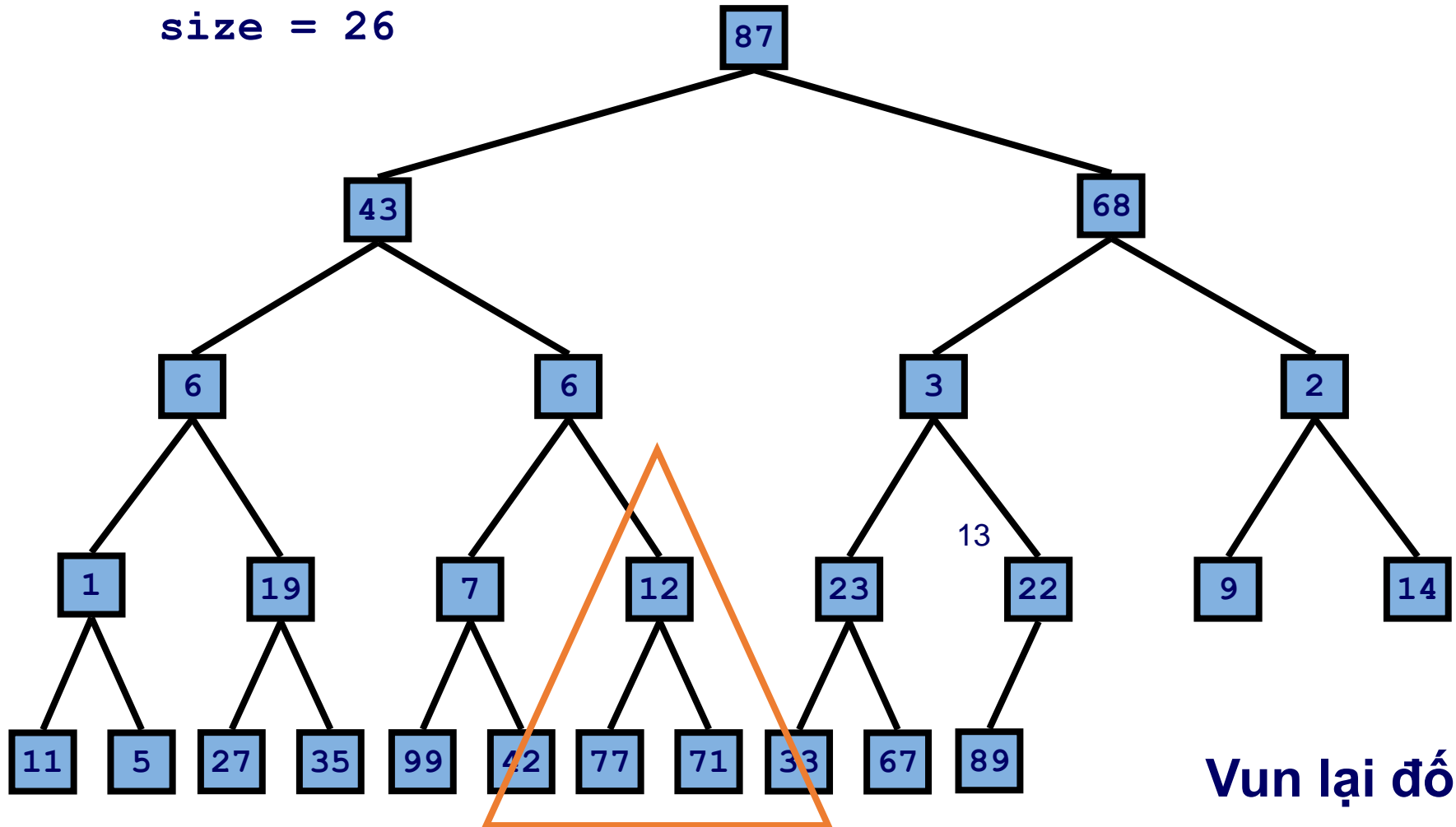
Build-Min-Heap

size = 26



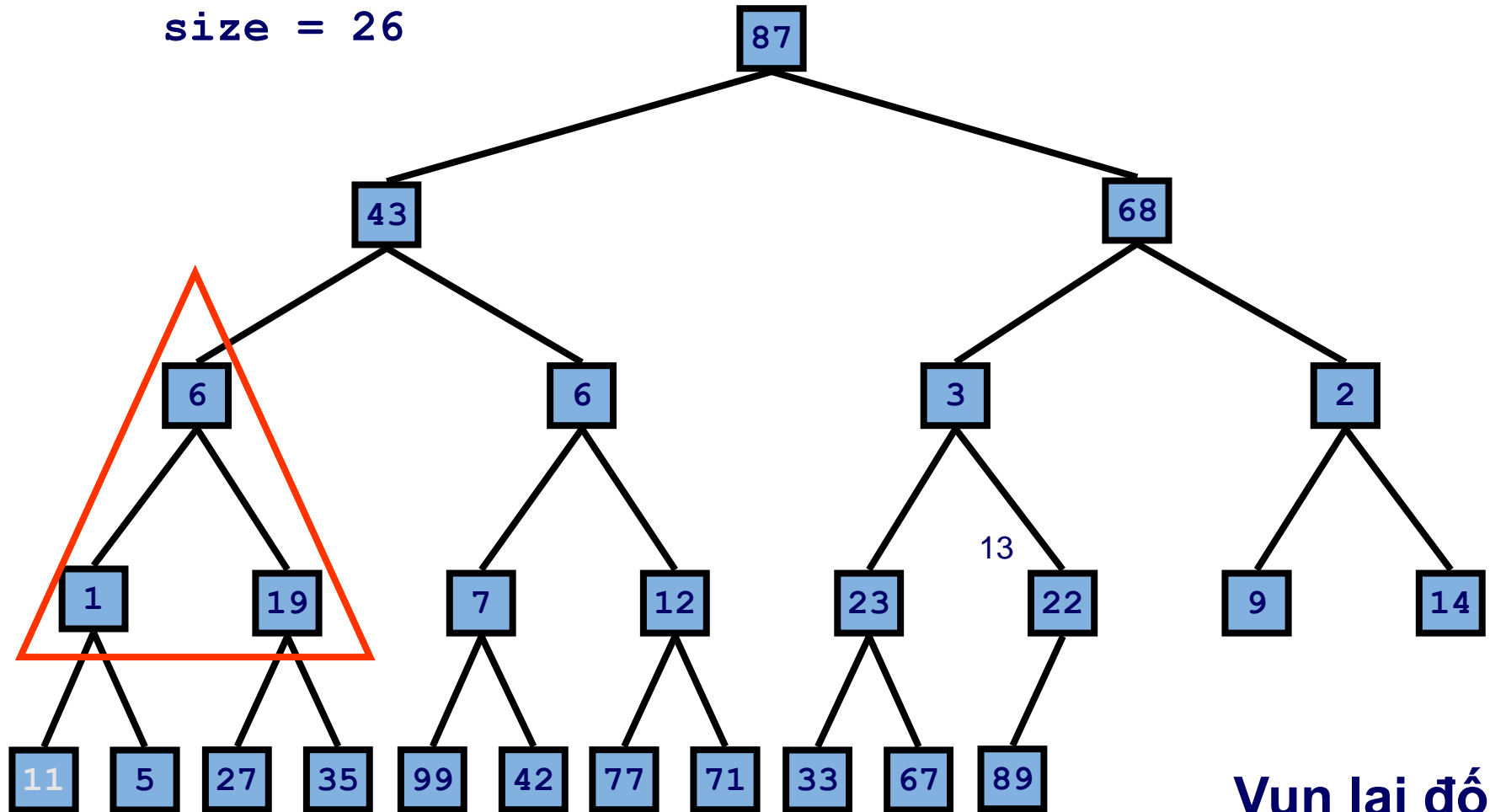
Build-Min-Heap

size = 26



Build-Min-Heap

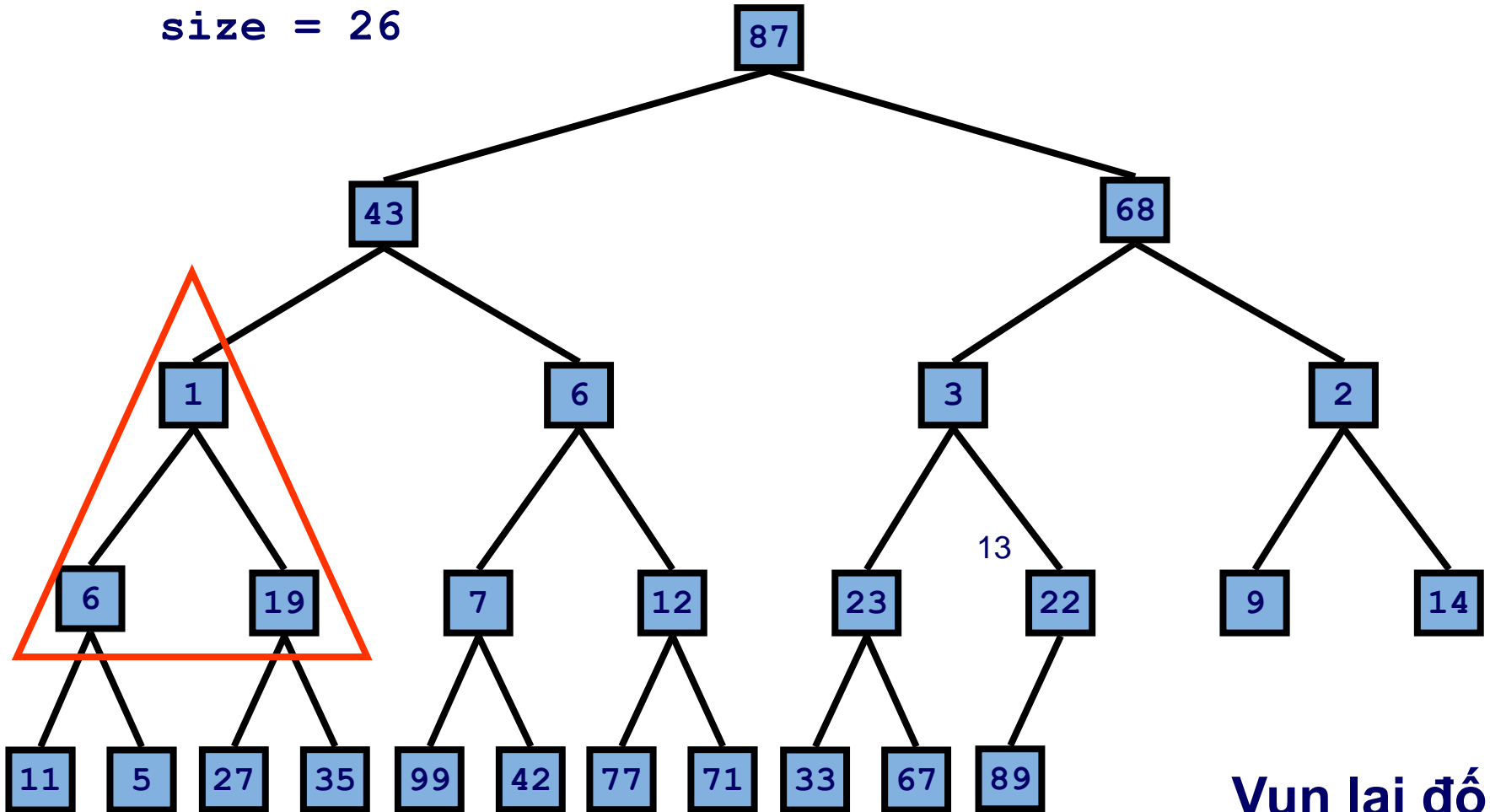
size = 26



Vun lại đồng

Build-Min-Heap

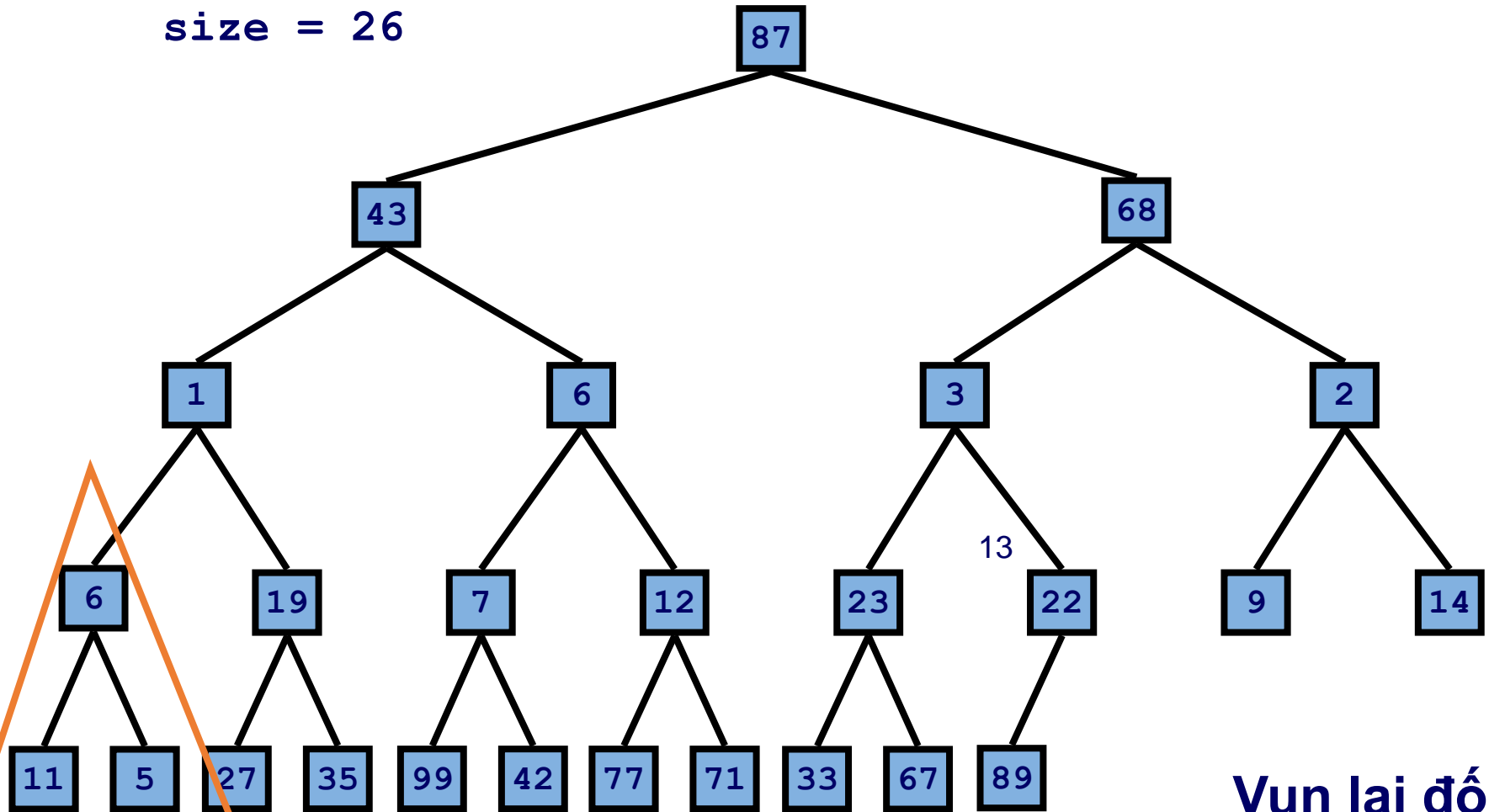
size = 26



Vun lại đồng

Build-Min-Heap

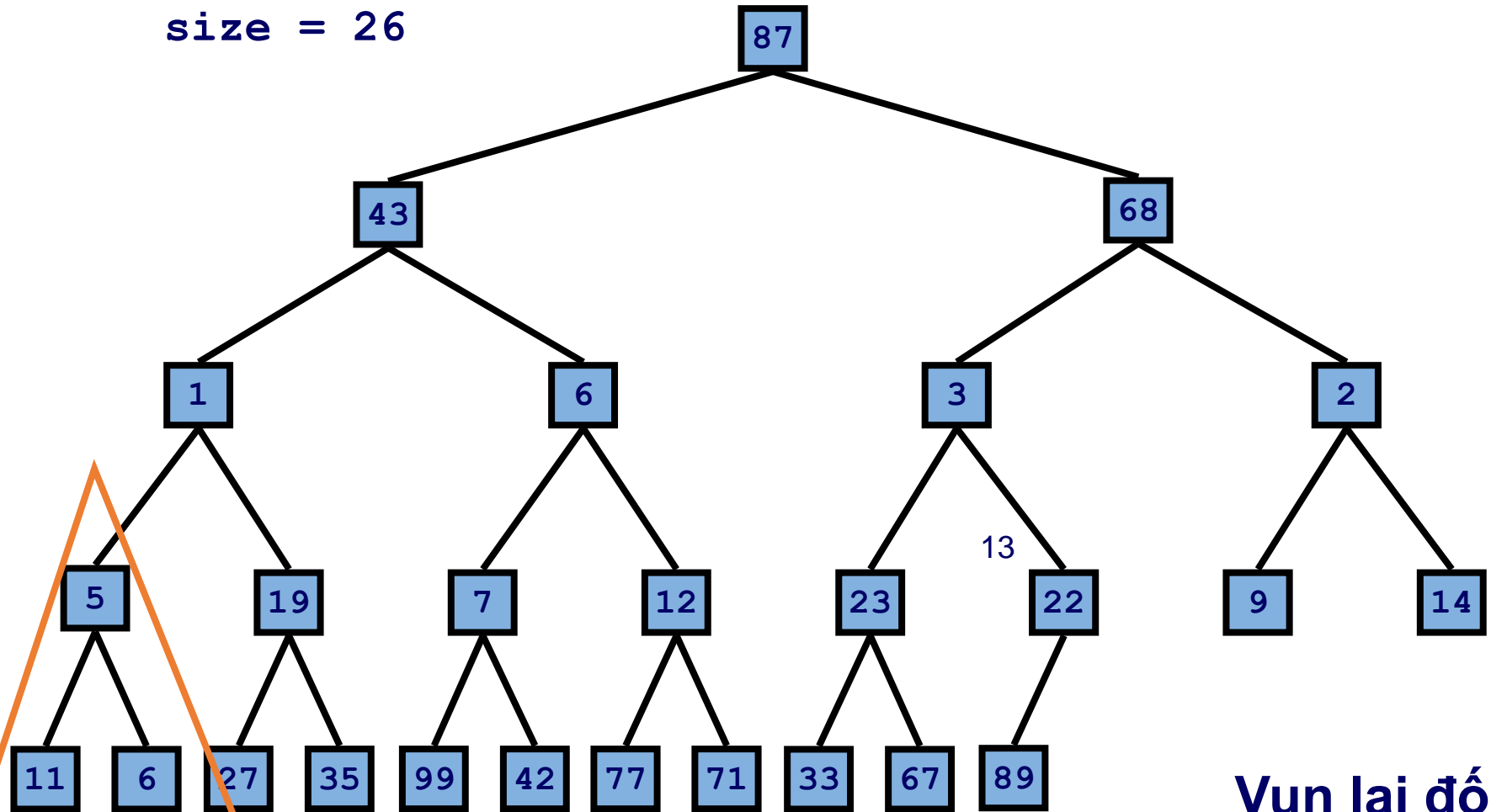
size = 26



Vun lại đồng

Build-Min-Heap

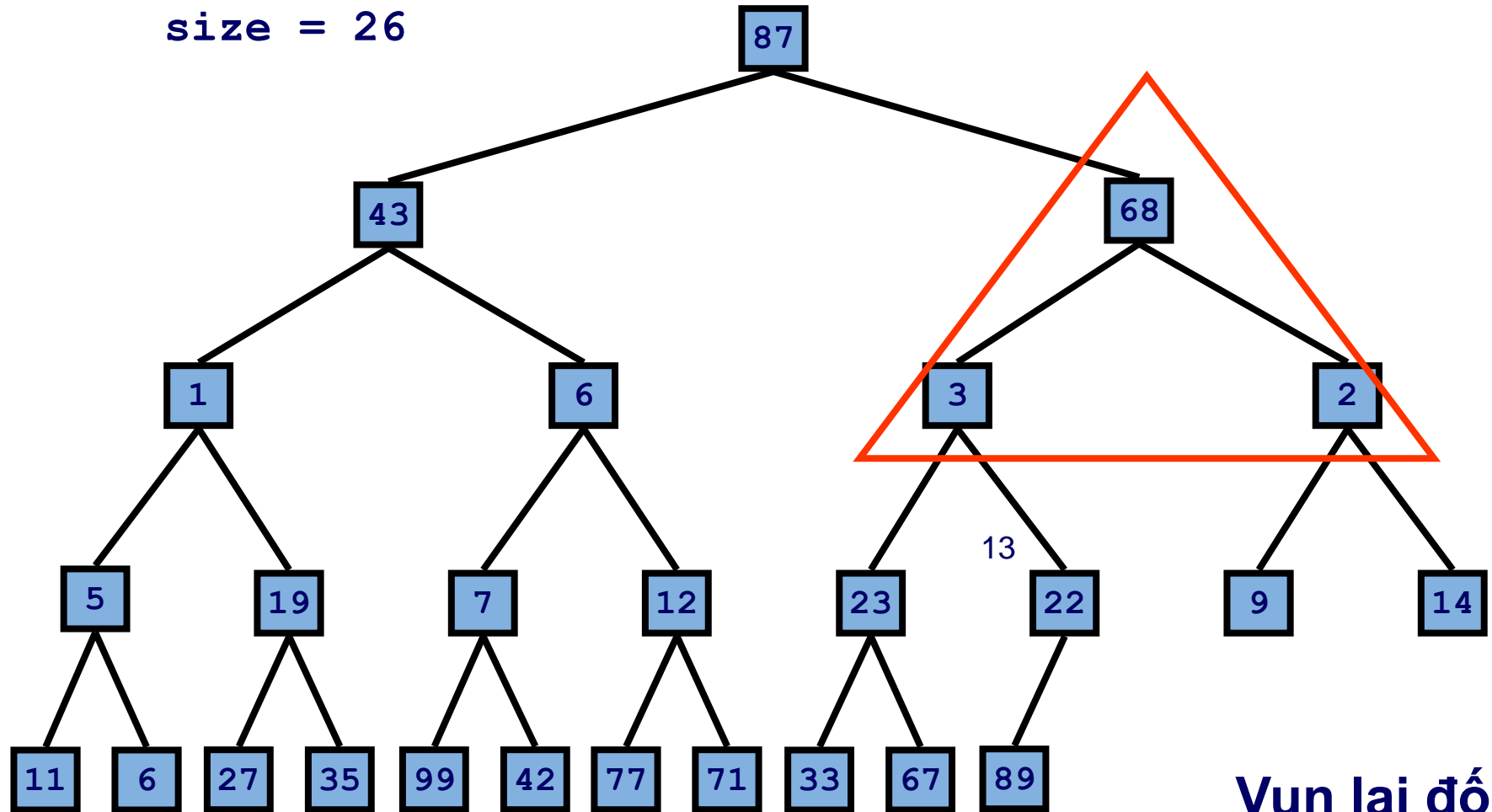
size = 26



Vun lại đồng

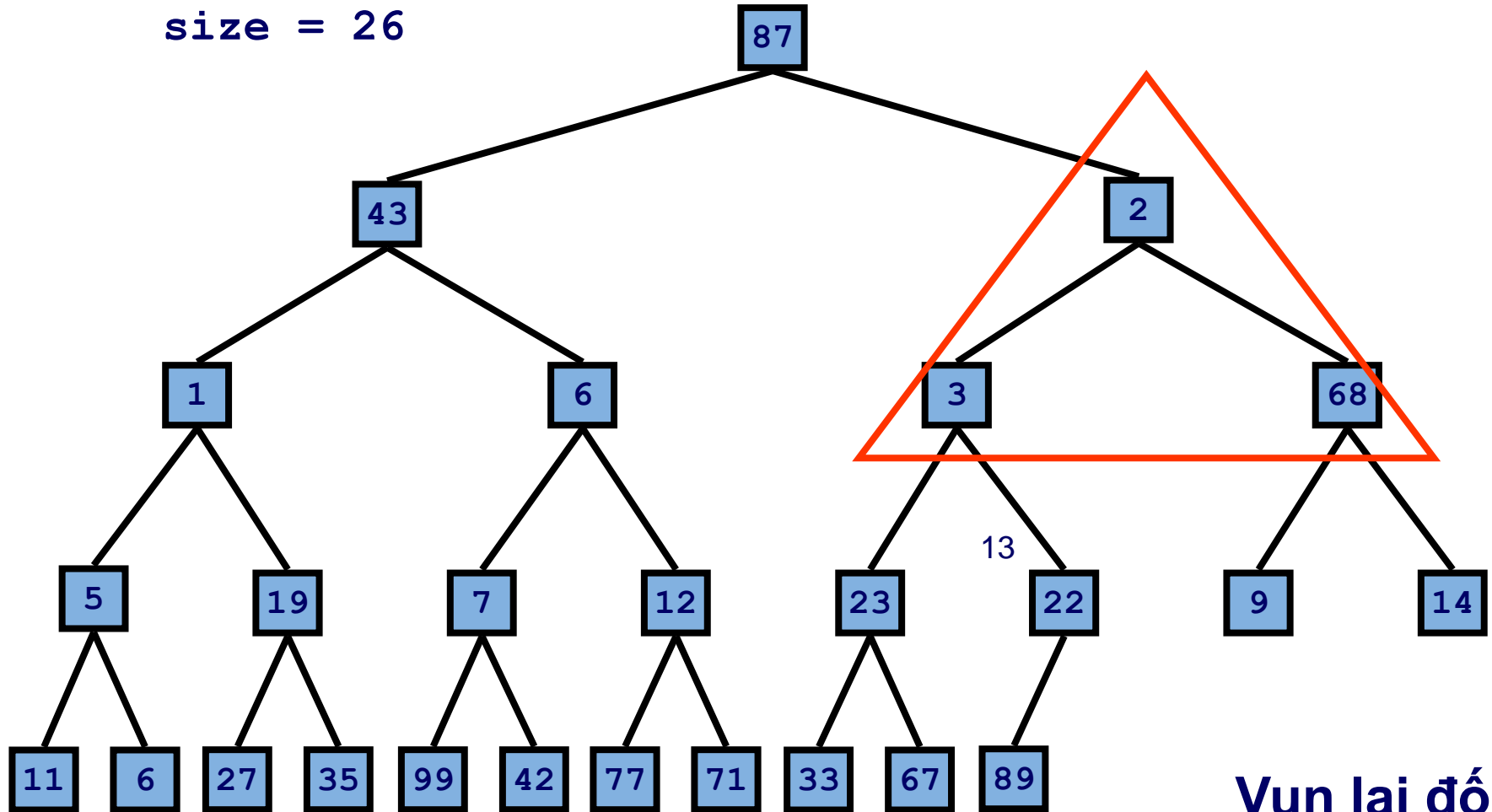
Build-Min-Heap

size = 26



Build-Min-Heap

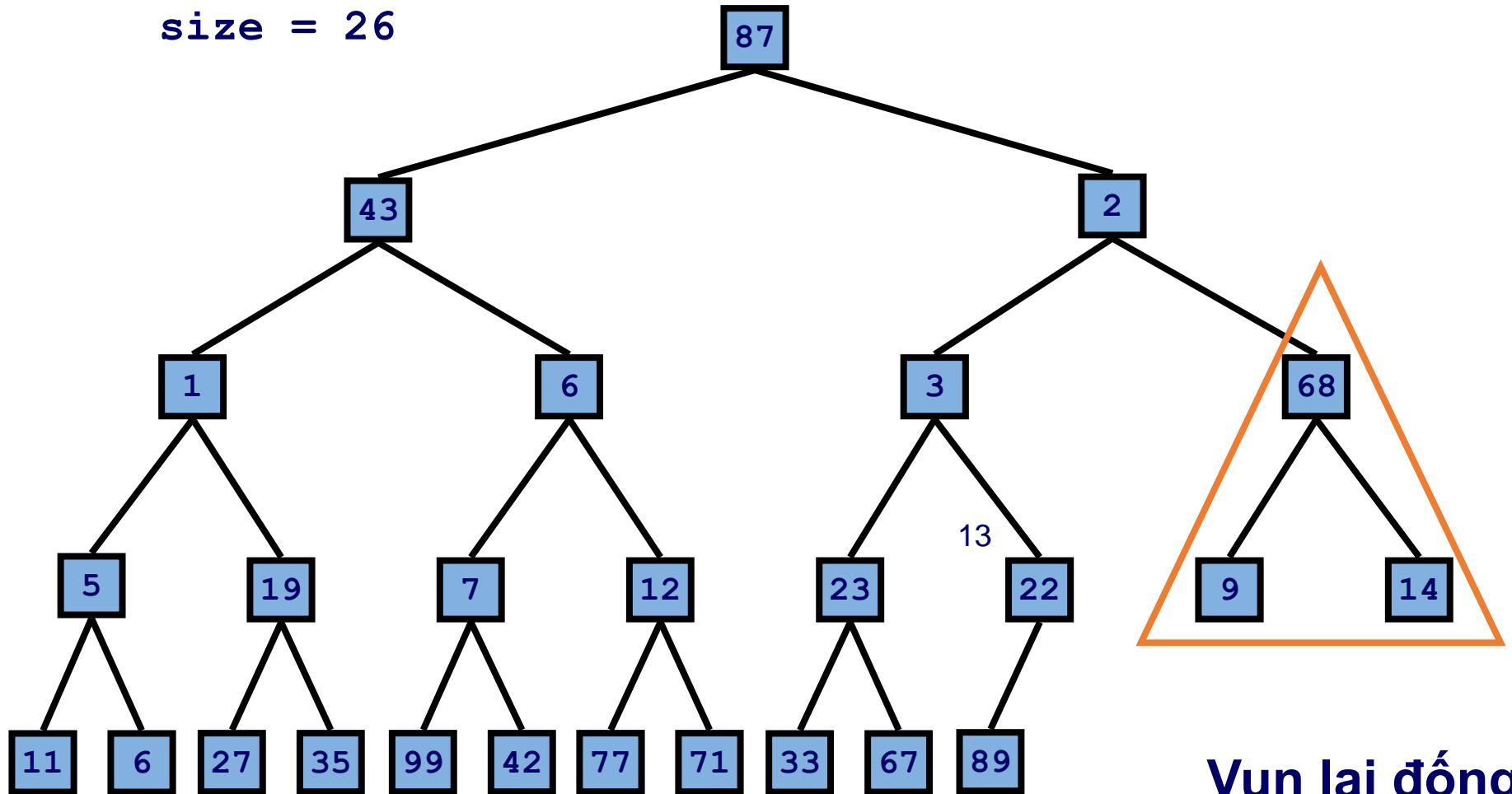
size = 26



Vun lại đồng

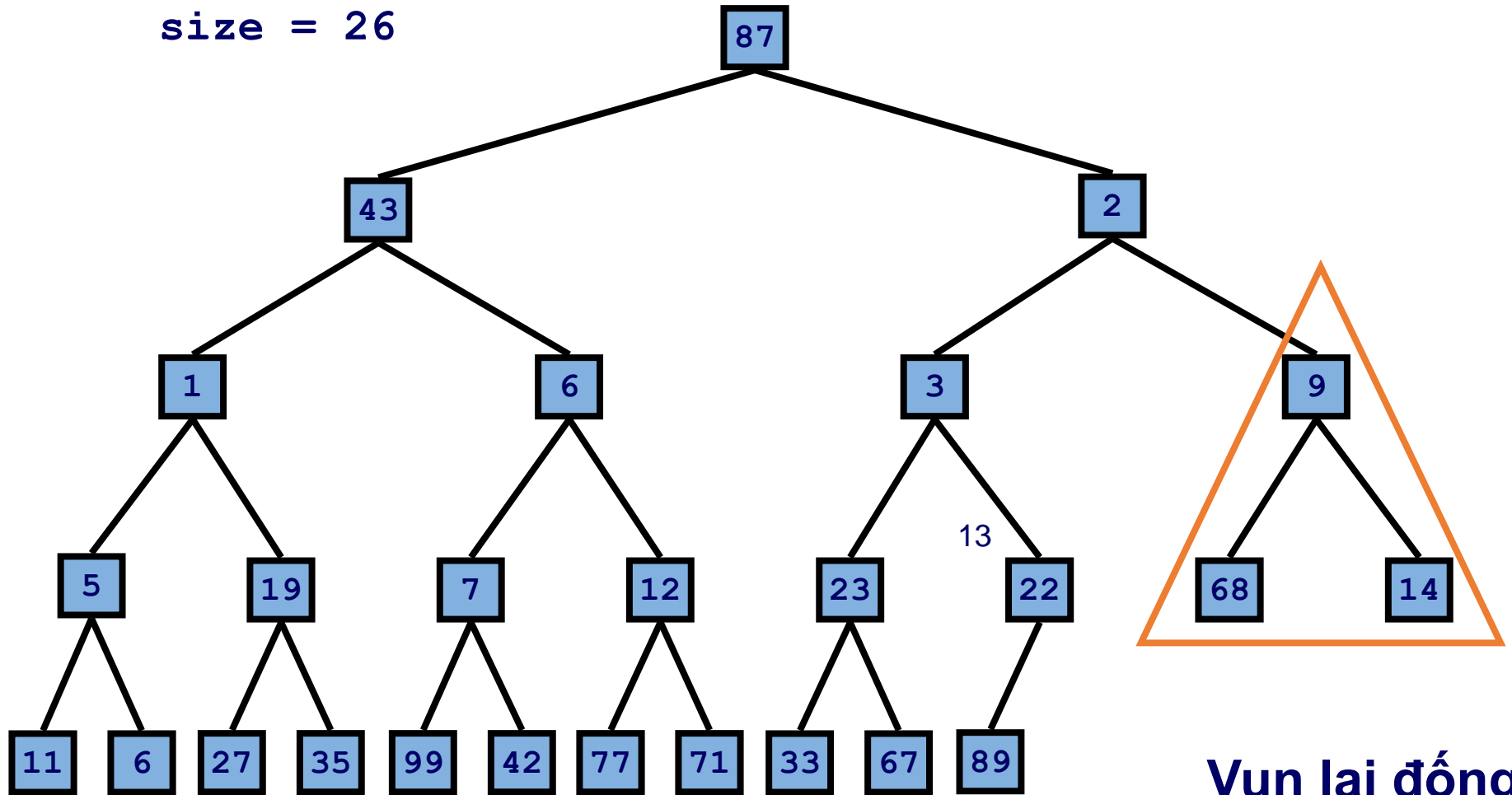
Build-Min-Heap

size = 26

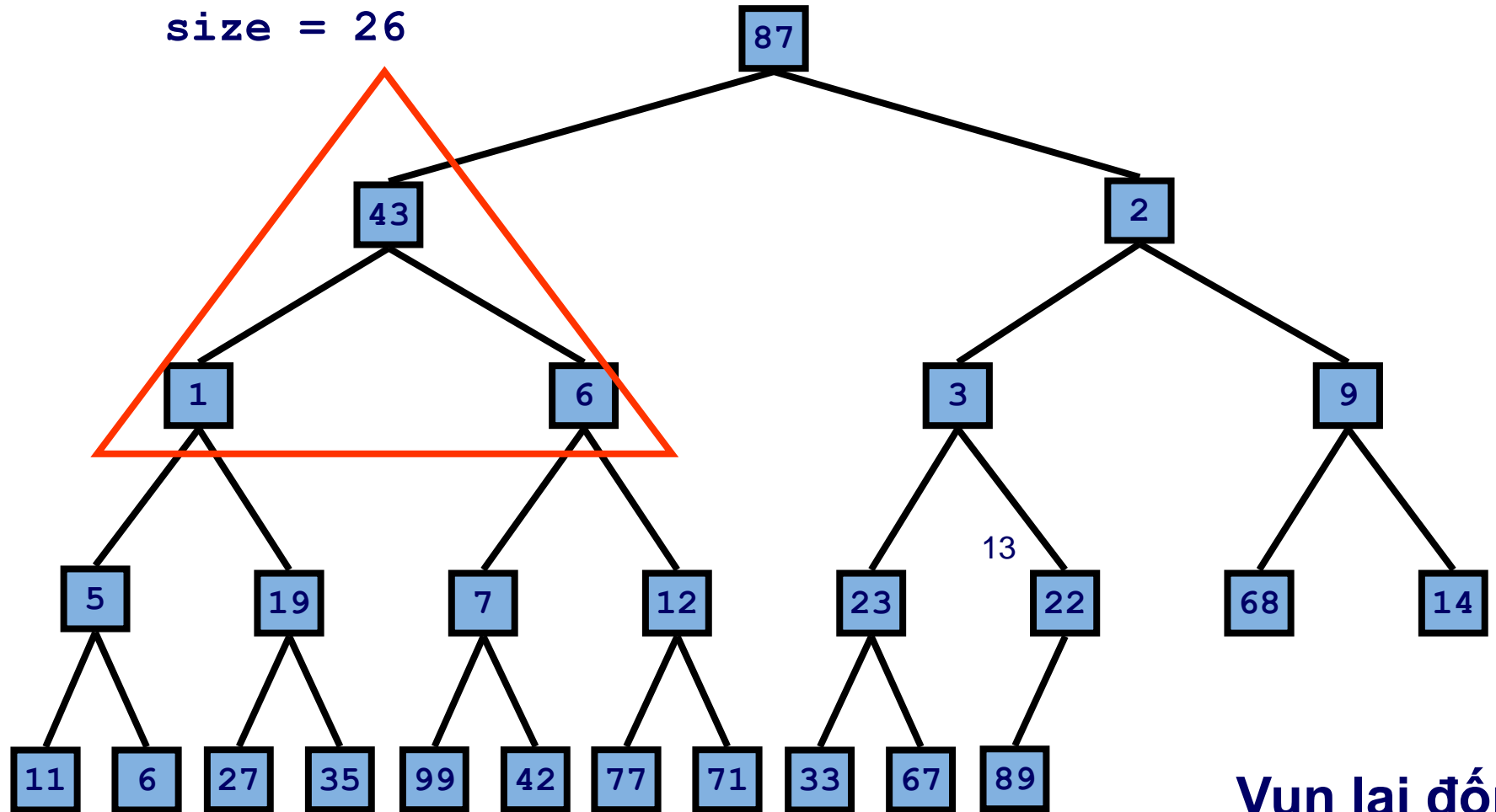


Build-Min-Heap

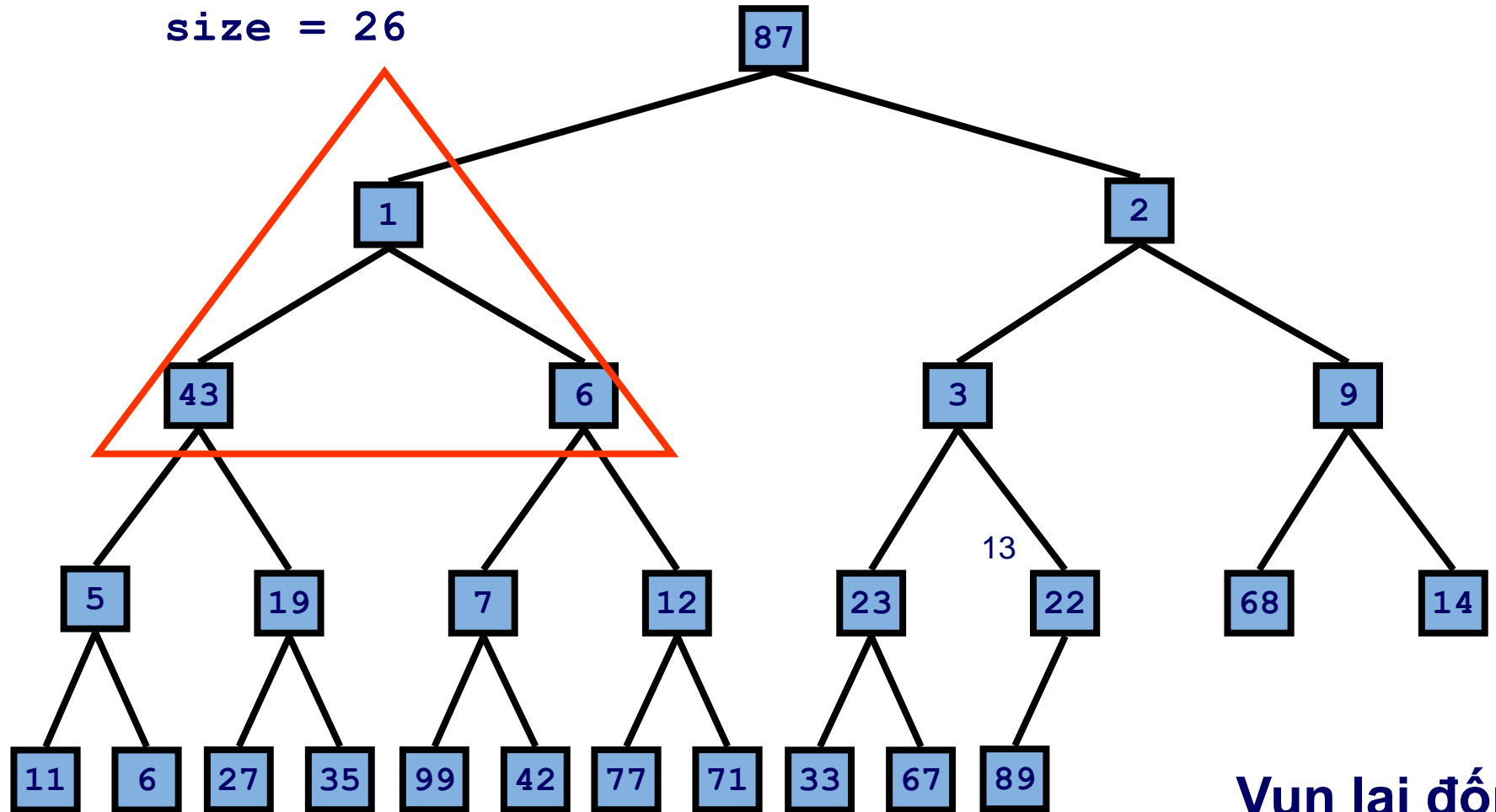
size = 26



Build-Min-Heap

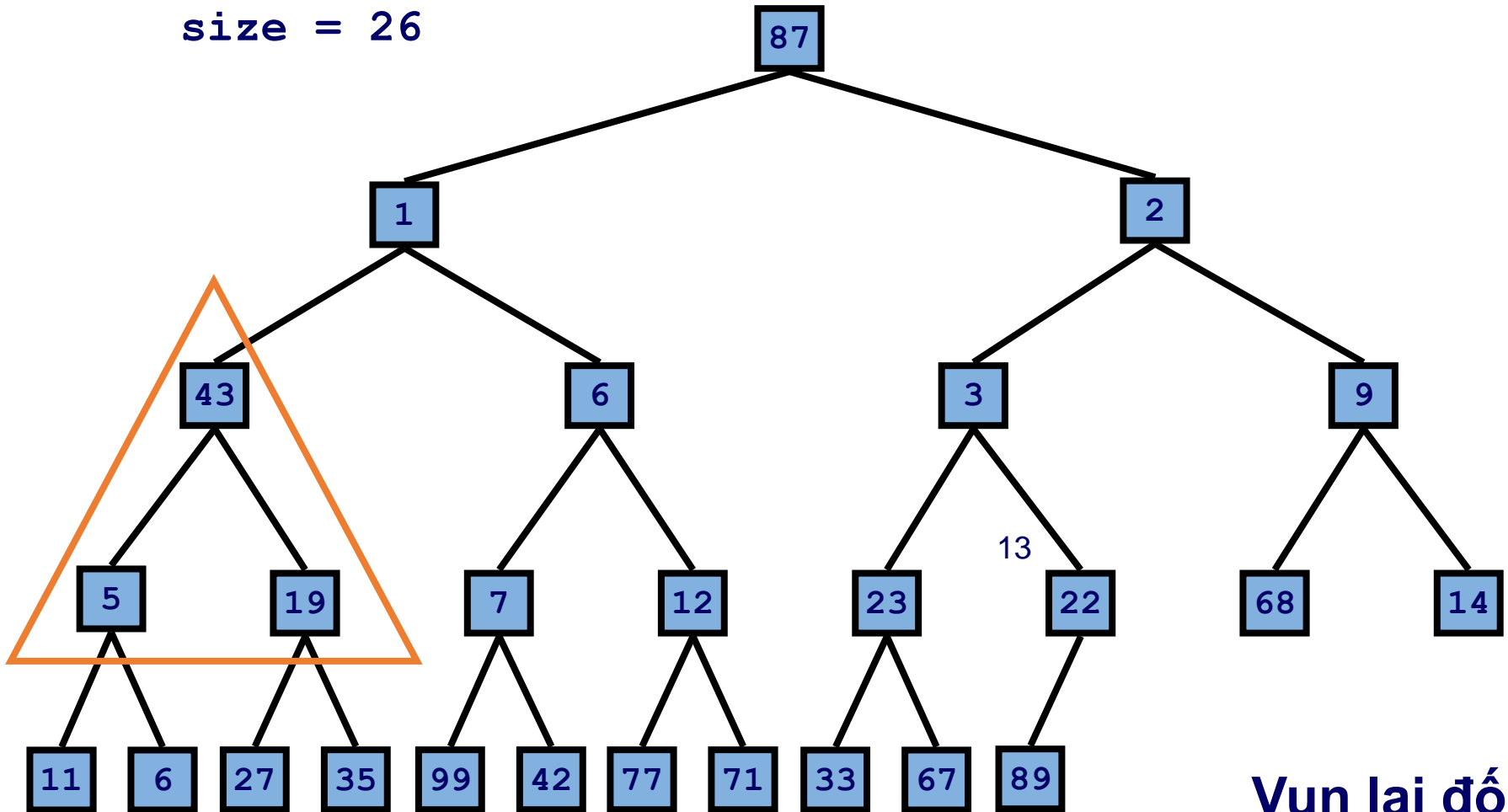


Build-Min-Heap



Build-Min-Heap

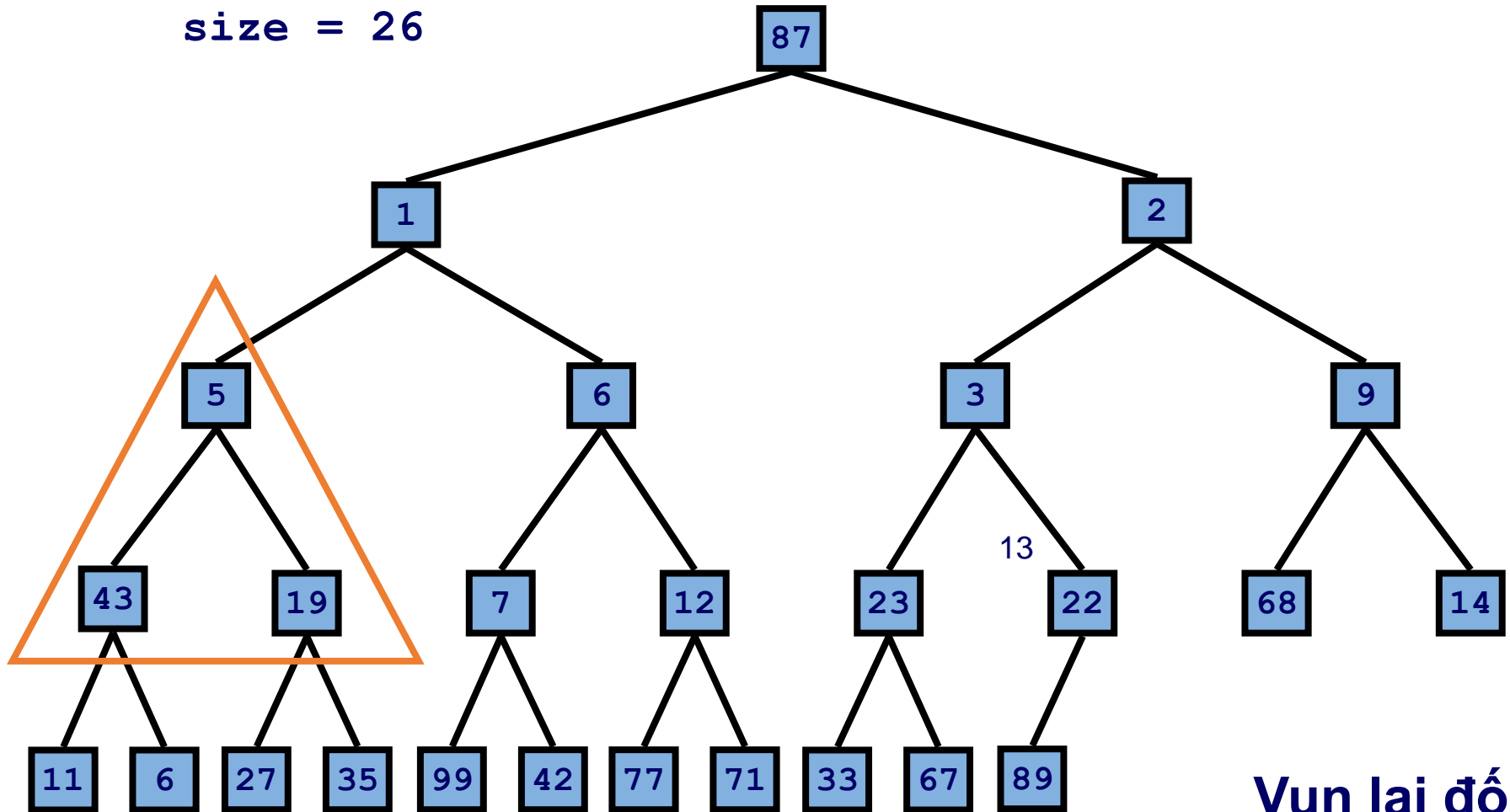
size = 26



Vun lại đồng

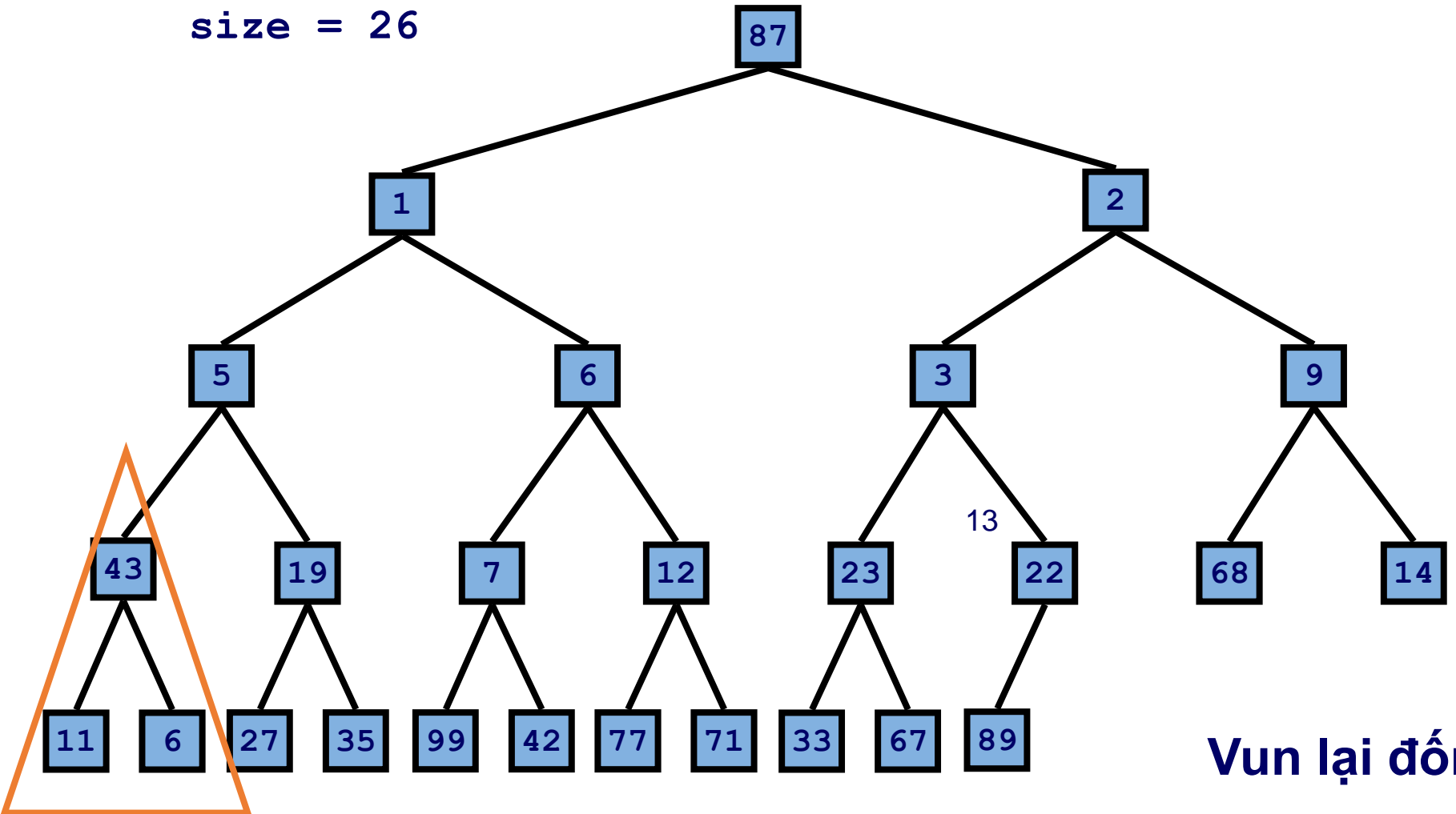
Build-Min-Heap

size = 26



Build-Min-Heap

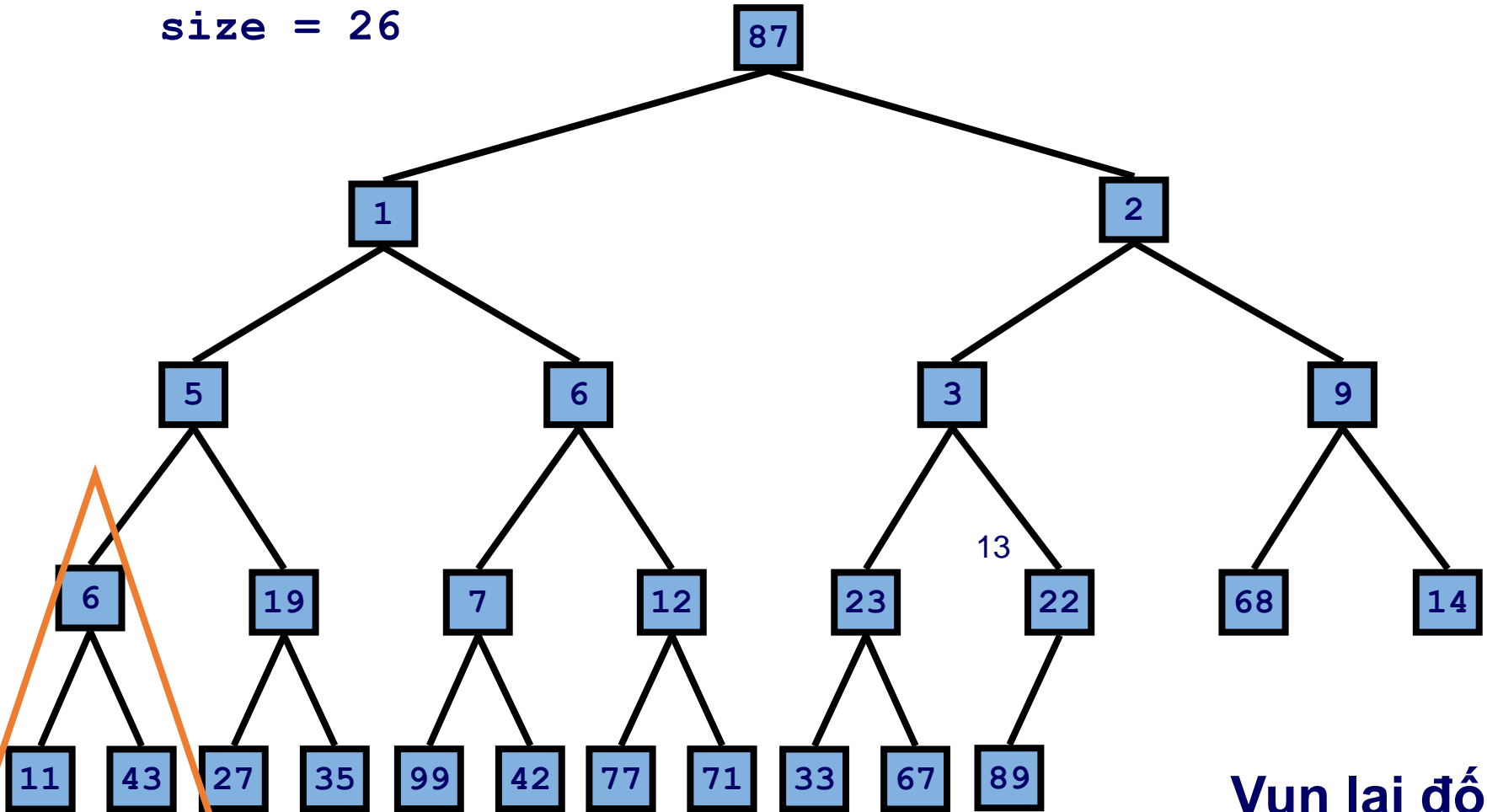
size = 26



Vun lại đồng

Build-Min-Heap

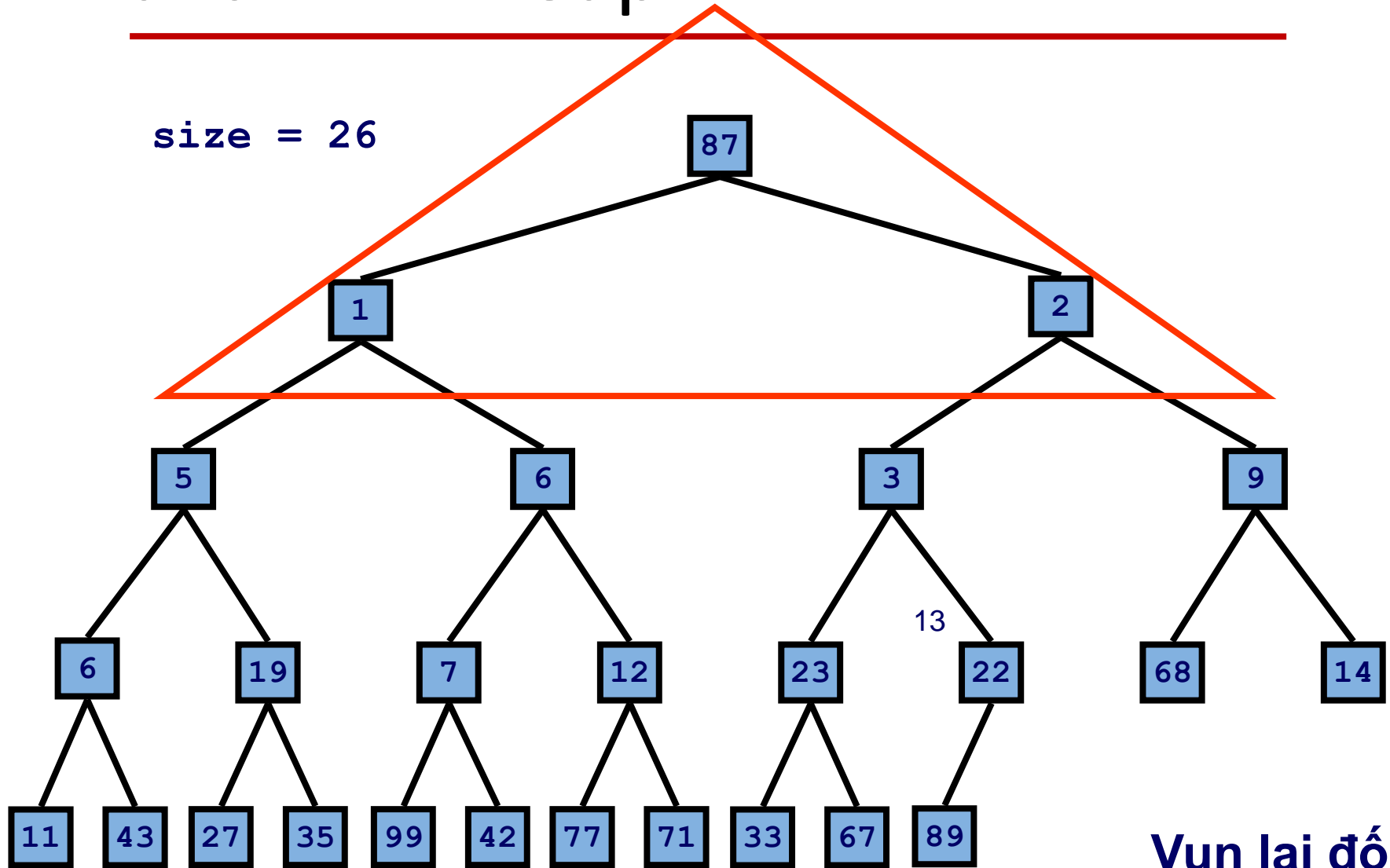
size = 26



Vun lại đồng

Build-Min-Heap

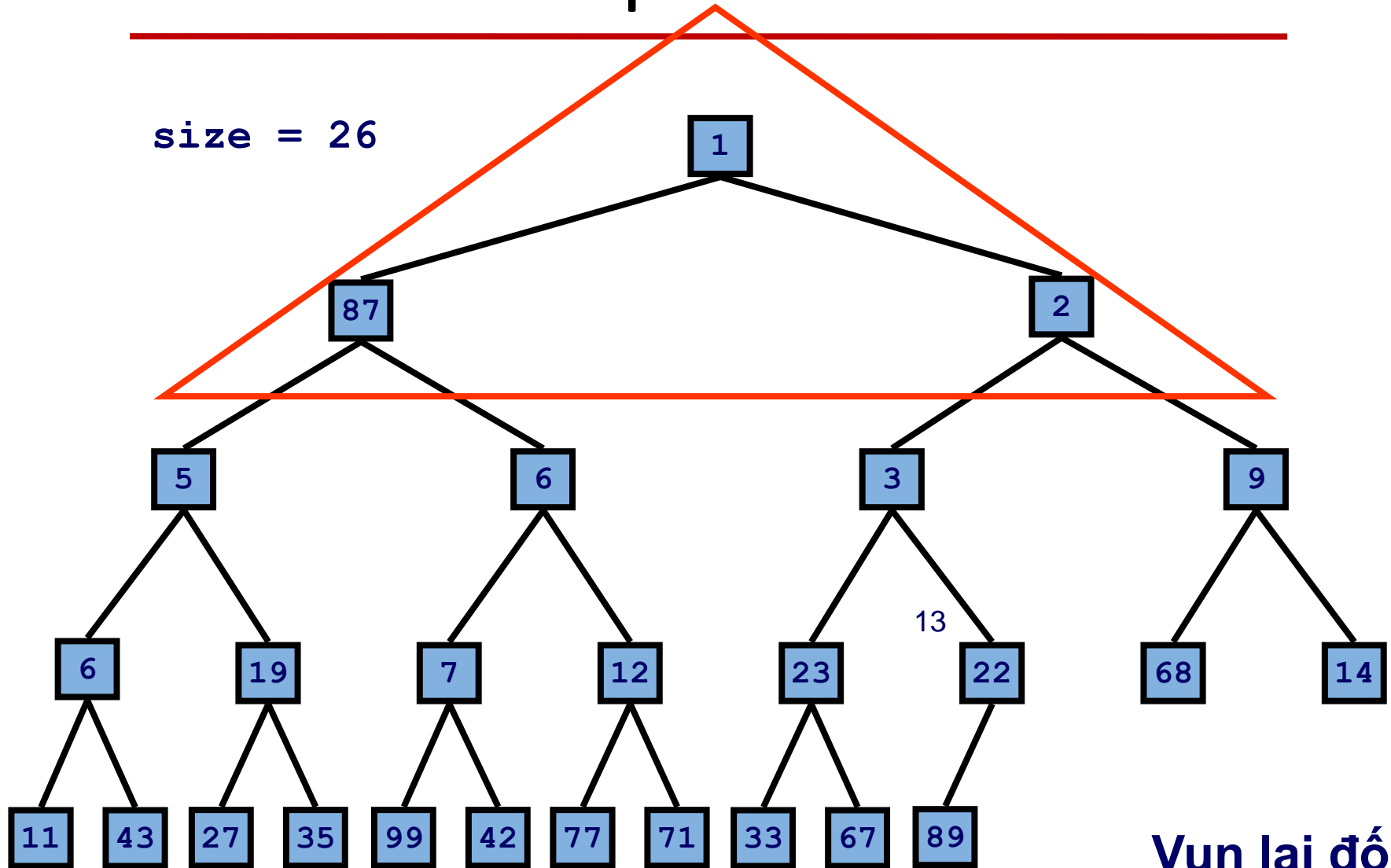
size = 26



Vun lại đồng

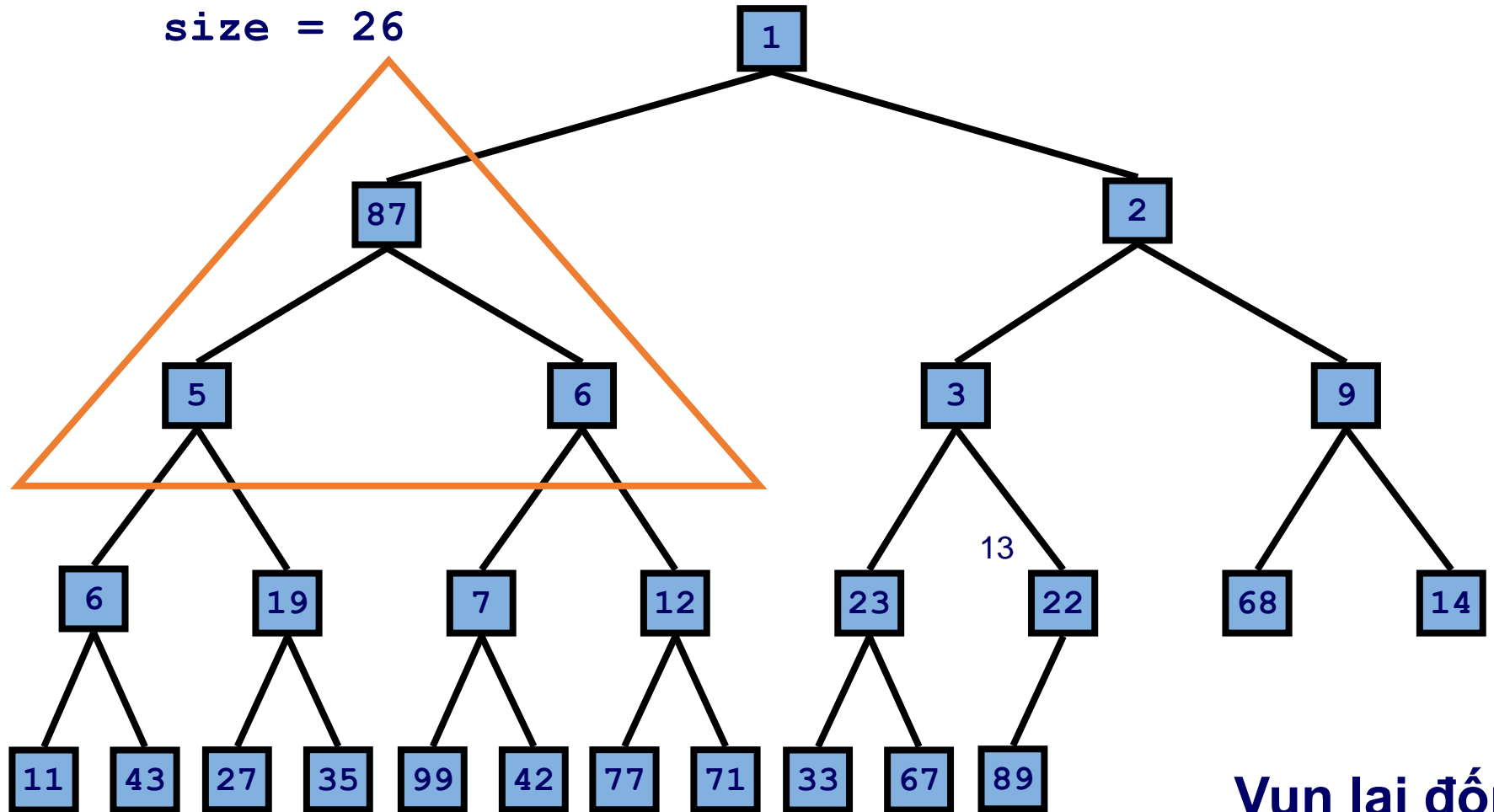
Build-Min-Heap

size = 26

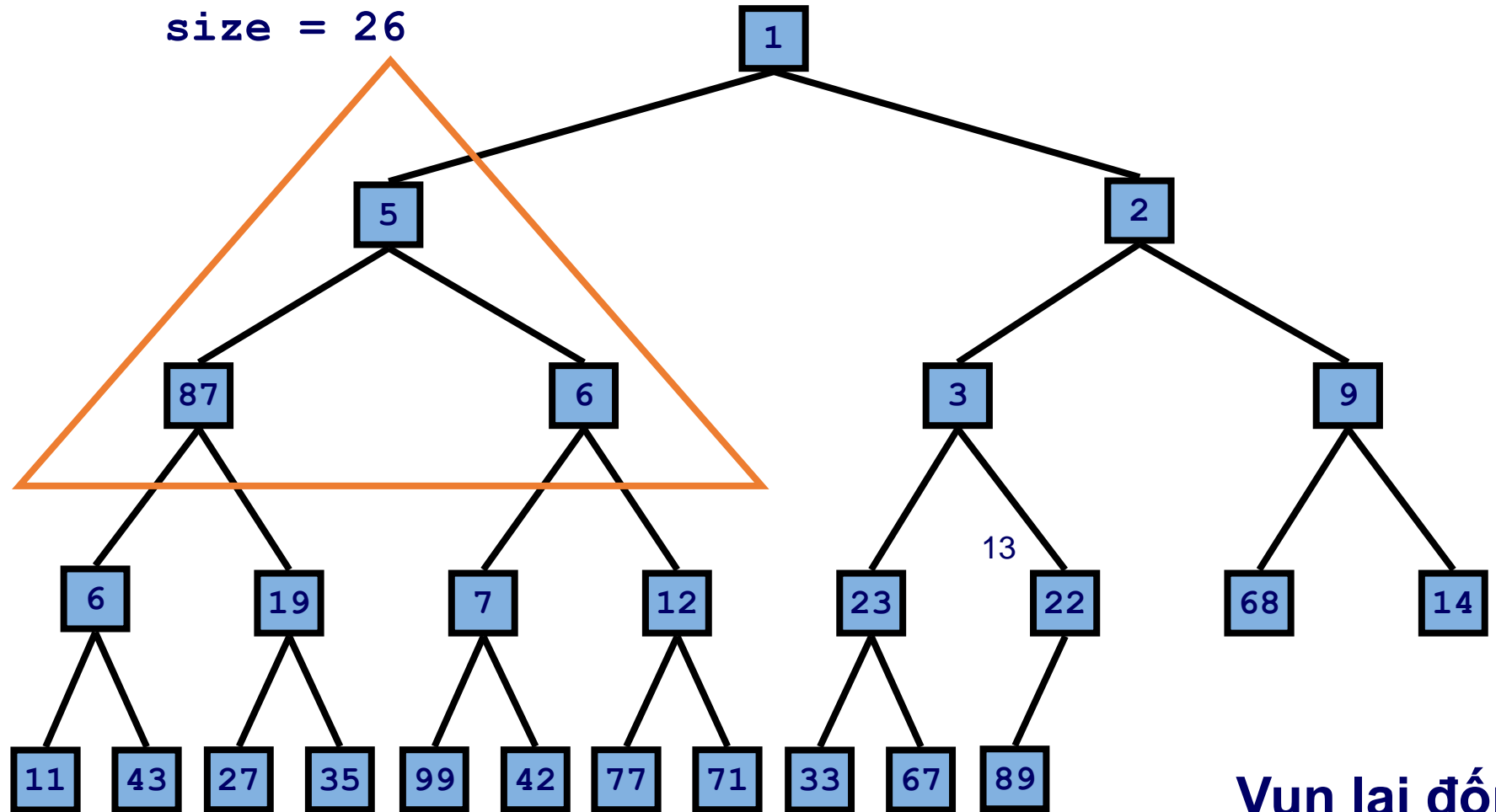


Vun lại đồng

Build-Min-Heap

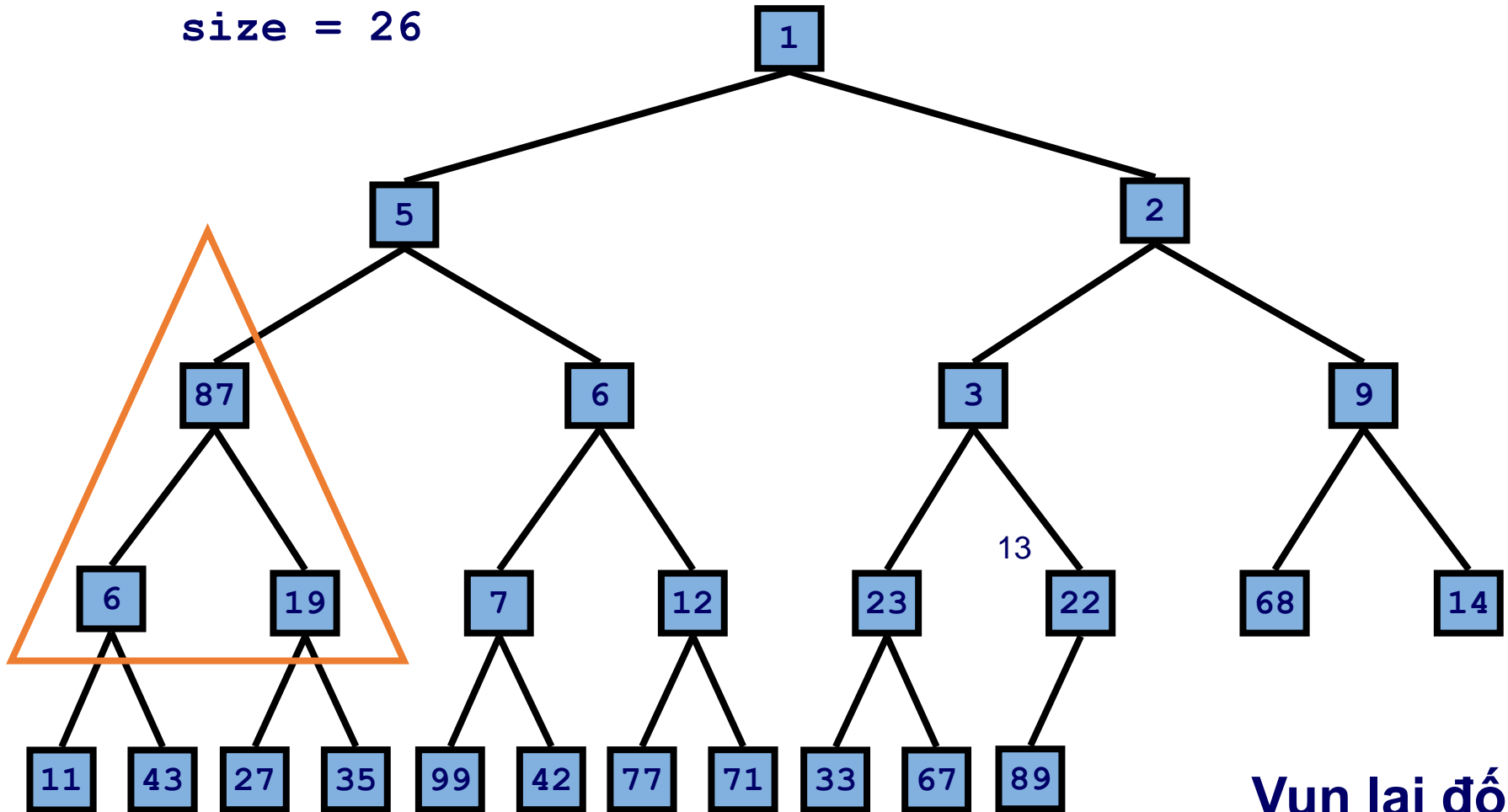


Build-Min-Heap



Build-Min-Heap

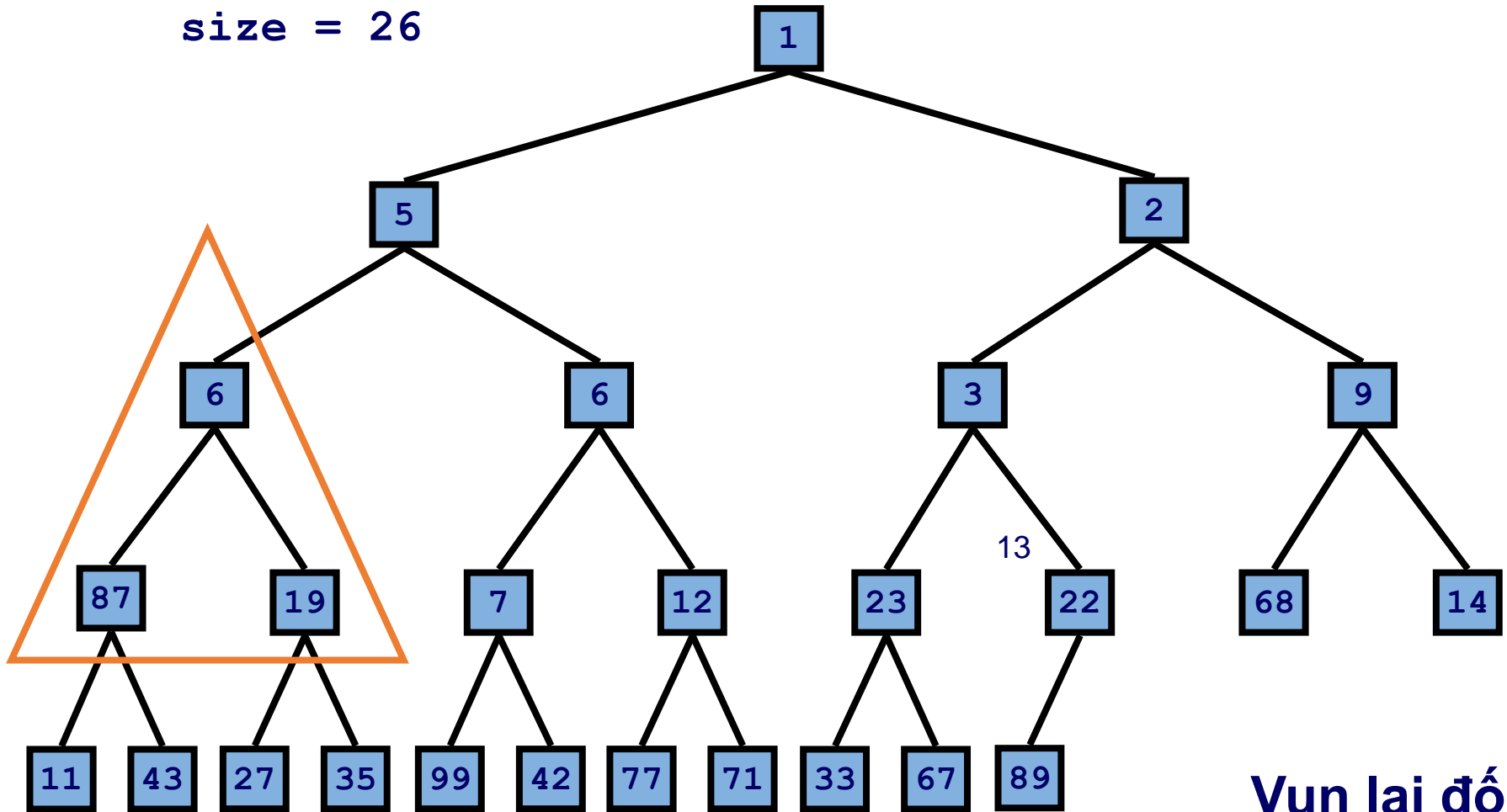
size = 26



Vun lại đồng

Build-Min-Heap

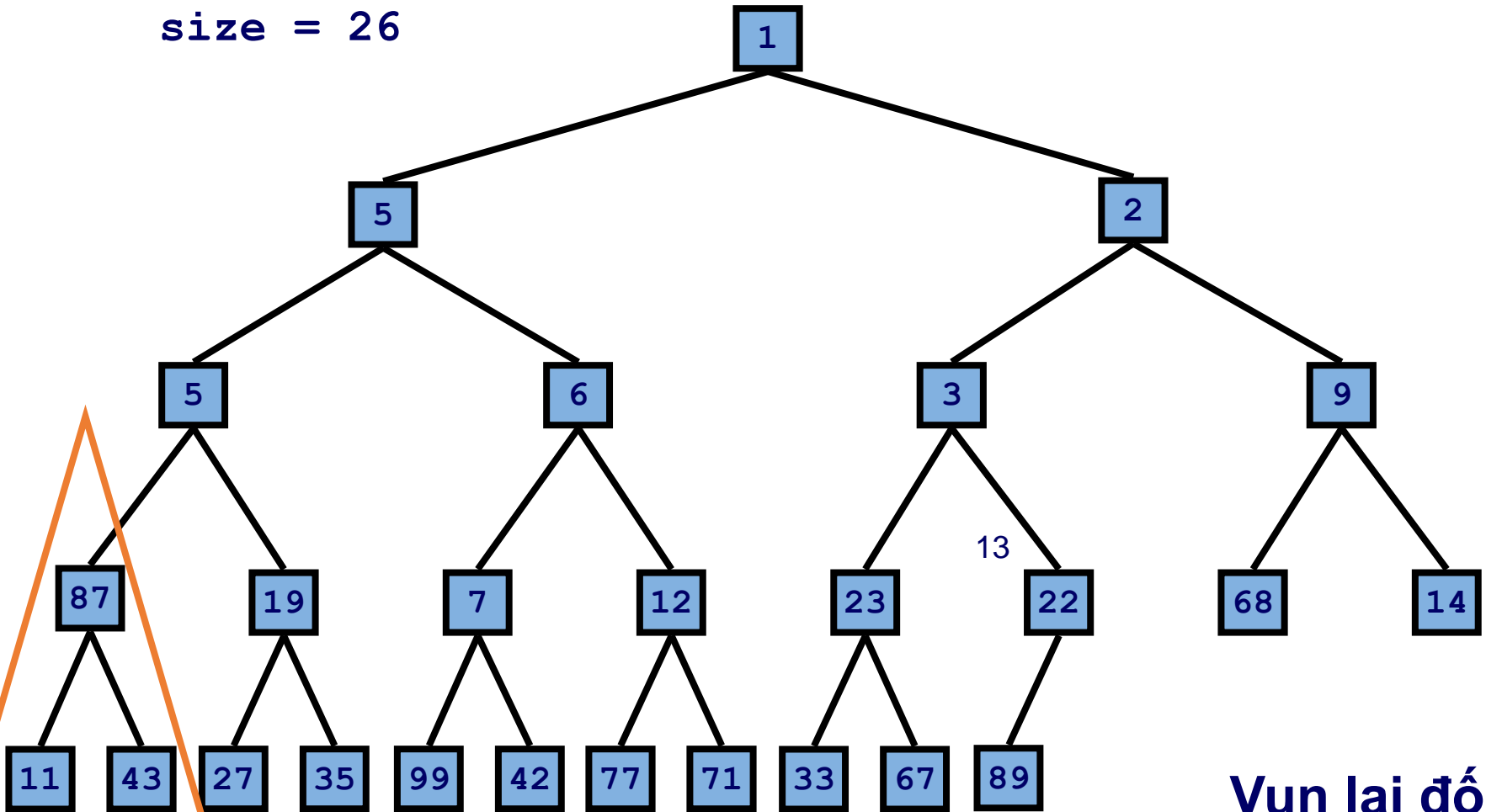
size = 26



Vun lại đồng

Build-Min-Heap

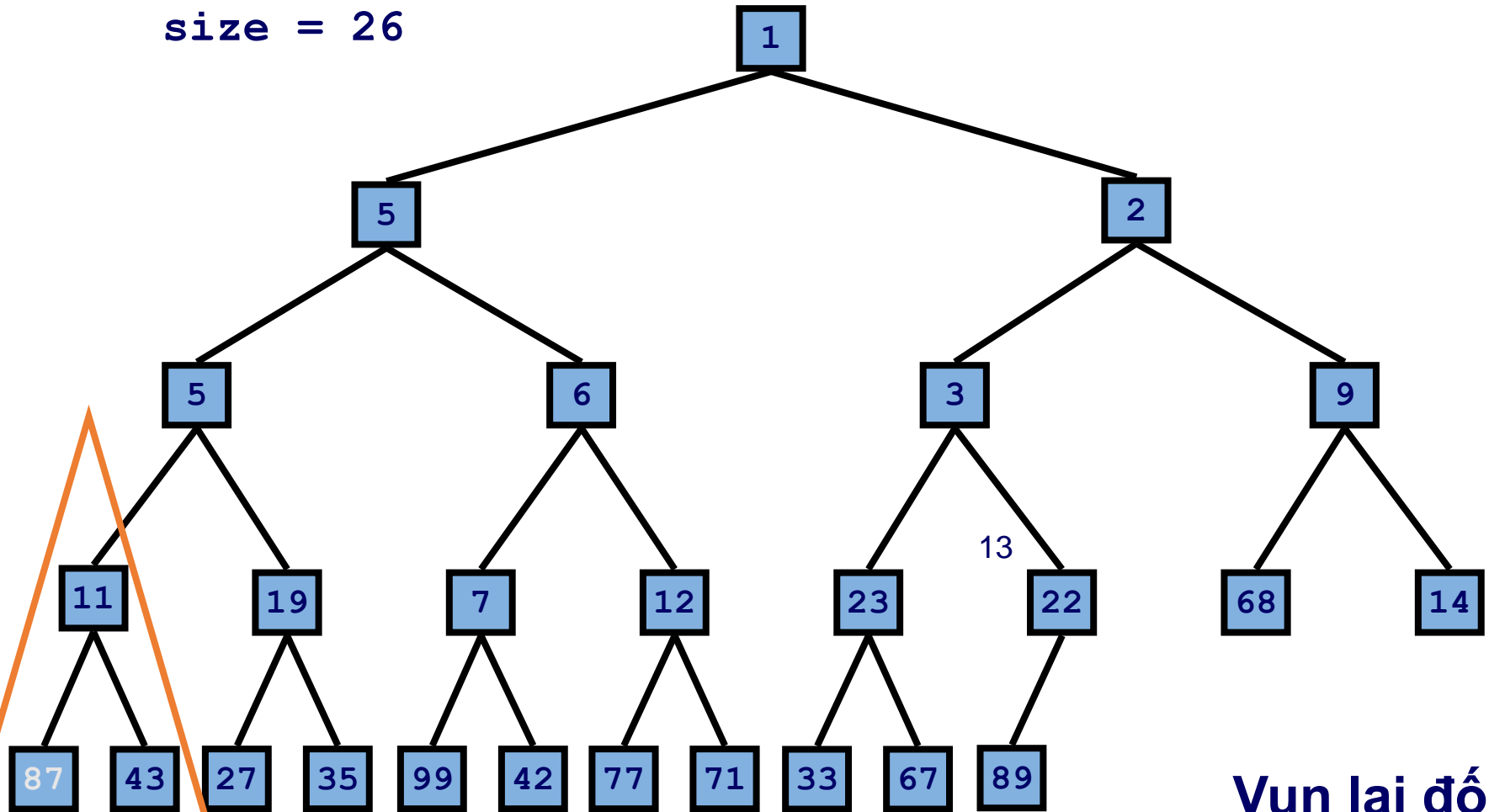
size = 26



Vun lại đồng

Build-Min-Heap

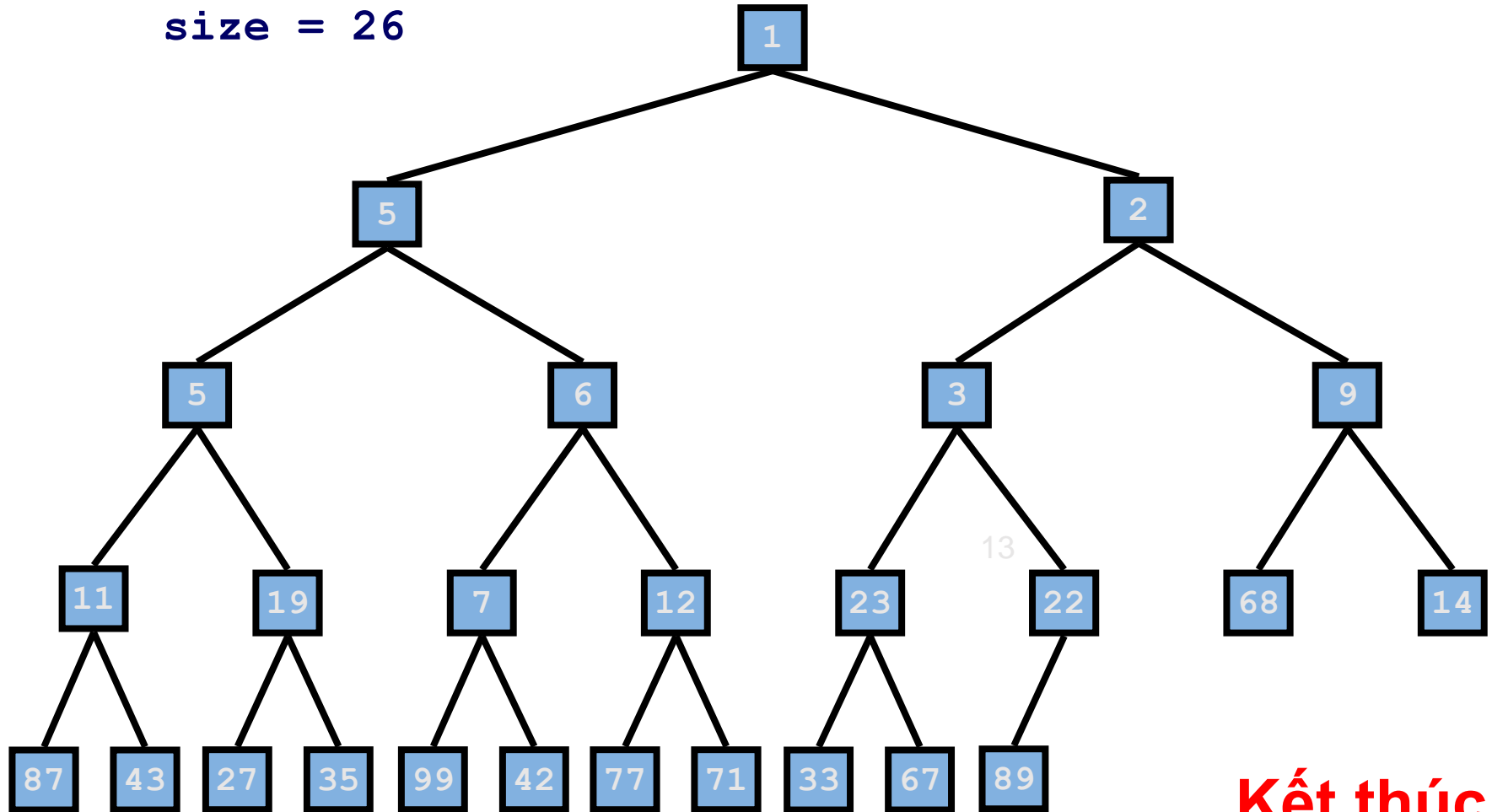
size = 26



Vun lại đồng

Build-Min-Heap

size = 26



Kết thúc

Thời gian tính của Build-Max-Heap

Alg: Build-Max-Heap(A)

1. $n = \text{length}[A]$
2. **for** $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1
3. **do** Max-Heapify(A, i, n)

$O(\log n)$ } $O(n)$

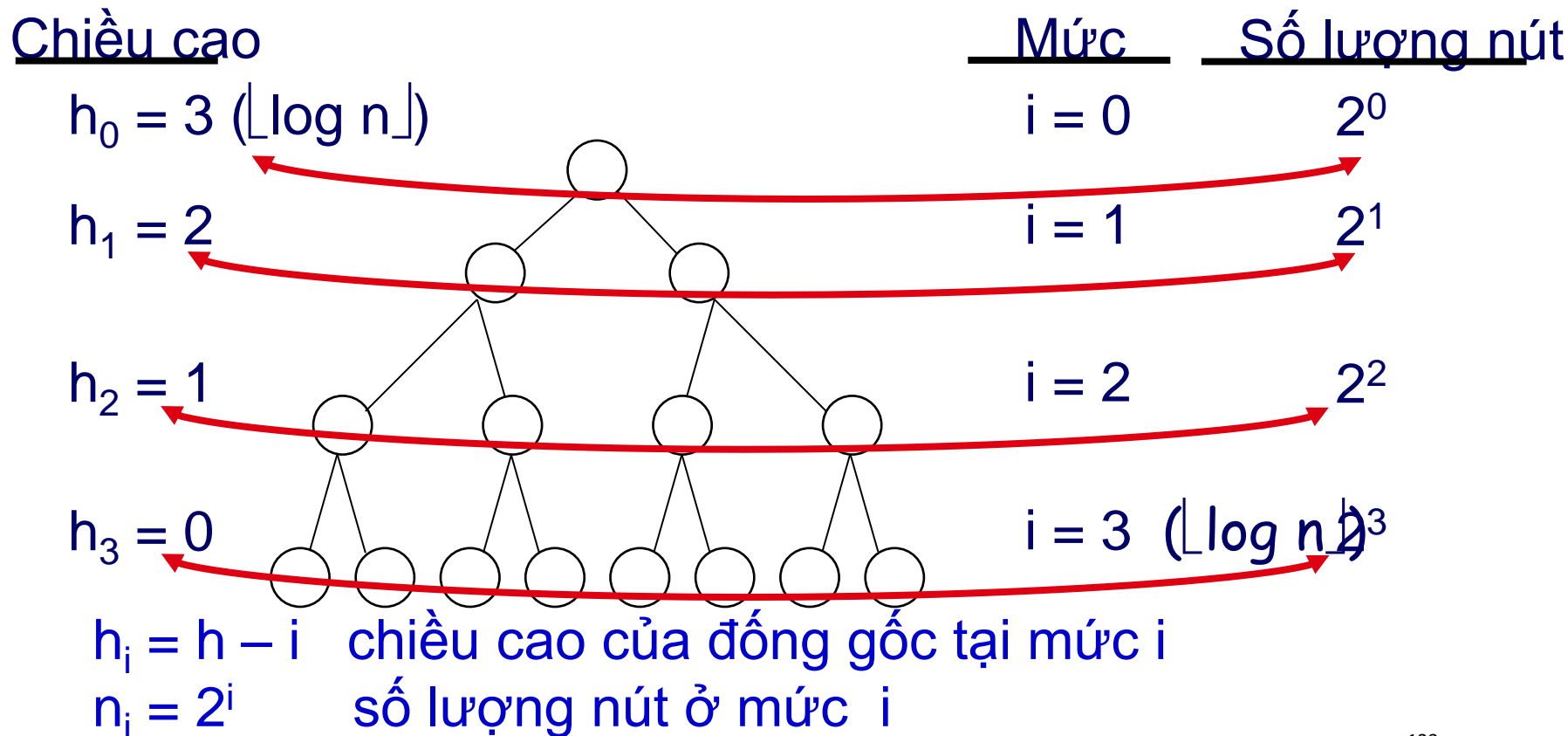
\Rightarrow Thời gian tính là: $O(n \log n)$

- Đánh giá này là không sát !

Thời gian tính của Buid-Max-Heap

- Heapify đòi hỏi thời gian $O(h) \Rightarrow$ chi phí của Heapify ở nút i là tỷ lệ với chiều cao của nút i trên cây

$$\Rightarrow T(n) = \sum_{i=0}^h n_i h_i = \sum_{i=0}^h 2^i (h - i) = O(n)$$



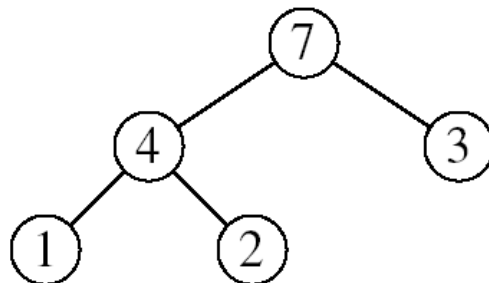
Thời gian tính của Build-Max-Heap

$$\begin{aligned} T(n) &= \sum_{i=0}^h n_i h_i && \text{(Chi phí Heapify tại mức } i) \times (\text{số lượng nút trên mức này)} \\ &= \sum_{i=0}^h 2^i (h-i) && \text{Thay giá trị của } n_i \text{ và } h_i \text{ tính được ở trên} \\ &= \sum_{i=0}^h \frac{h-i}{2^{h-i}} 2^h && \text{Nhân cả tử và mẫu với } 2^h \text{ và viết } \frac{1}{2^i} \text{ là} \\ &= 2^h \sum_{k=0}^h \frac{k}{2^k} && \text{Đổi biến: } k = h - i \\ &\leq n \sum_{k=0}^{\infty} \frac{k}{2^k} && \text{Thay tổng hữu hạn bởi tổng vô hạn} \\ & && \text{và } h = \log n \\ &= O(n) && \text{Tổng trên là nhỏ hơn 2} \end{aligned}$$

Thời gian tính của Build-Max-Heap: $T(n) = O(n)$

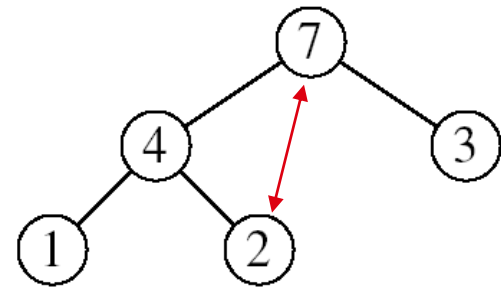
5.5.2. Sắp xếp vun đống - Heapsort

- Sử dụng đống ta có thể phát triển thuật toán sắp xếp mảng.
- Sơ đồ của thuật toán được trình bày như sau:
 - Tạo đống **max-heap** từ mảng đã cho
 - Đổi chỗ gốc (phần tử lớn nhất) với phần tử cuối cùng trong mảng
 - Loại bỏ nút cuối cùng bằng cách giảm kích thước của đống đi 1
 - Thực hiện Max-Heapify đối với gốc mới
 - Lặp lại quá trình cho đến khi đống chỉ còn 1 nút

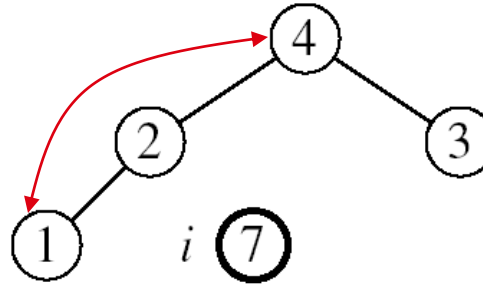


Ví dụ:

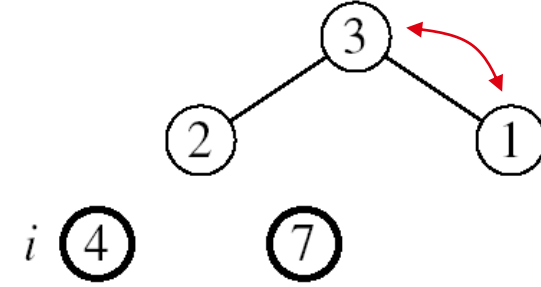
$A=[7, 4, 3, 1, 2]$



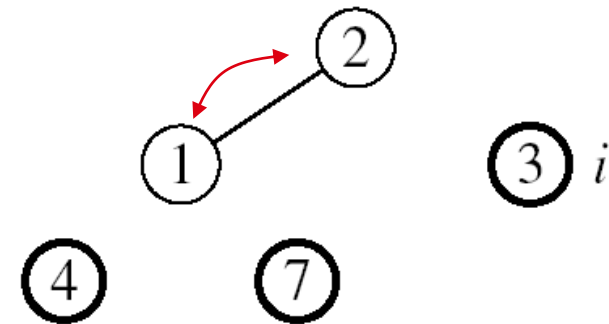
Max-Heapify($A, 1, 4$)



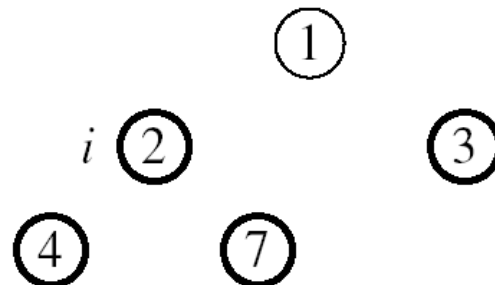
Max-Heapify($A, 1, 3$)



Max-Heapify($A, 1, 2$)



Max-Heapify($A, 1, 1$)



A

1	2	3	4	7
---	---	---	---	---

Algorithm: HeapSort(A)

1. Build-Max-Heap(A) $O(n)$
 2. **for** $i \leftarrow \text{length}[A]$ **downto** 2
 3. **do** exchange $A[1] \leftrightarrow A[i]$ $O(\log n)$
 4. Max-Heapify(A, 1, $i - 1$)
- } $n-1$ lần lặp

- Thời gian tính: $O(n \log n)$
- Có thể chứng minh thời gian tính là $\Theta(n \log n)$

5.5.3. Hàng đợi có ưu tiên - Priority Queues

- Cho tập S thường xuyên biến động, mỗi phần tử x được gán với một giá trị gọi là khoá (hay độ ưu tiên). Cần một cấu trúc dữ liệu hỗ trợ hiệu quả các thao tác chính sau:
 - $\text{Insert}(S, x)$ – Bổ sung phần tử x vào S
 - $\text{Max}(S)$ – trả lại phần tử lớn nhất
 - $\text{Extract-Max}(S)$ – loại bỏ và trả lại phần tử lớn nhất
 - $\text{Increase-Key}(S, x, k)$ – tăng khoá của x thành k
- Cấu trúc dữ liệu đáp ứng các yêu cầu đó là ***hàng đợi có ưu tiên***.
- Hàng đợi có ưu tiên có thể tổ chức nhờ sử dụng cấu trúc dữ liệu đồng để cất giữ các khoá.
- ***Chú ý: Có thể thay "max" bởi "min" .***

Các phép toán đối với hàng đợi có ưu tiên

- Hàng đợi có ưu tiên (max) có các phép toán cơ bản sau:
 - **Insert(S, x)**: bổ sung phần tử x vào tập S
 - **Extract-Max(S)**: loại bỏ và trả lại phần tử của S với khoá lớn nhất
 - **Maximum(S)**: trả lại phần tử của S với khoá lớn nhất
 - **Increase-Key(S, x, k)**: tăng giá trị của khoá của phần tử x lên thành k (Giả sử $k \geq$ khoá hiện tại của x)

Các phép toán đối với hàng đợi có ưu tiên (min)

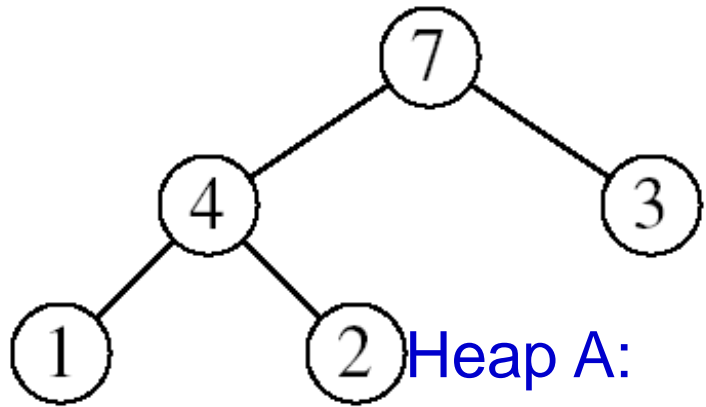
- Hàng đợi có ưu tiên (**min**) có các phép toán cơ bản sau:
 - **Insert(S, x)**: bổ sung phần tử x vào tập S
 - **Extract-Min(S)**: loại bỏ và trả lại phần tử của S với khoá nhỏ nhất
 - **Minimum(S)**: trả lại phần tử của S với khoá nhỏ nhất
 - **Decrease-Key(S, x, k)**: giảm giá trị của khoá của phần tử x xuống còn k (Giả sử $k \leq$ khoá hiện tại của x)

Phép toán HEAP-MAXIMUM

Chức năng: Trả lại phần tử lớn nhất của đống

Algo: Heap-Maximum(A)

Thời gian tính: $O(1)$



Heap A:

Heap-Maximum(A) trả lại 7

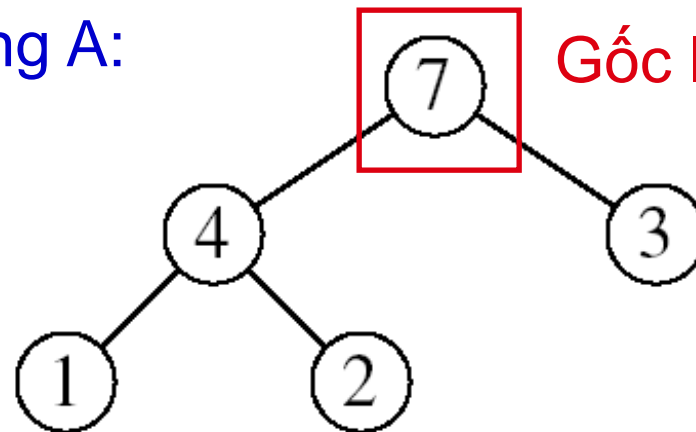
Heap-Extract-Max

Chức năng: Trả lại phần tử lớn nhất và loại bỏ nó khỏi đống

Thuật toán:

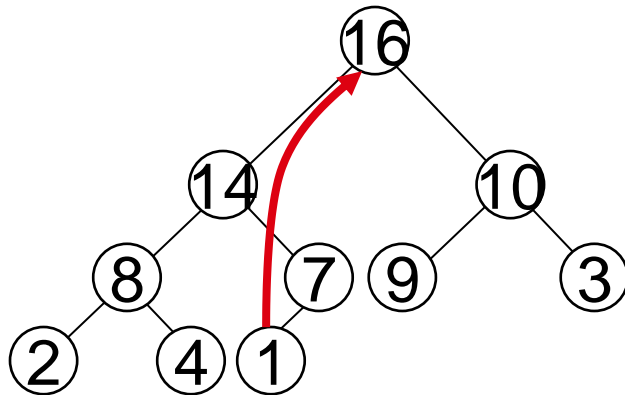
- Hoán đổi gốc với phần tử cuối cùng
- Giảm kích thước của đống đi 1
- Gọi Max-Heapify đối với gốc mới trên đống có kích thước $n-1$

Đống A:

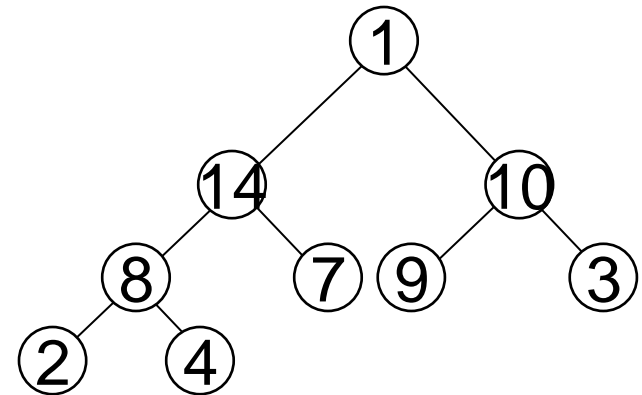


Gốc là phần tử lớn nhất

Ví dụ: Heap-Extract-Max

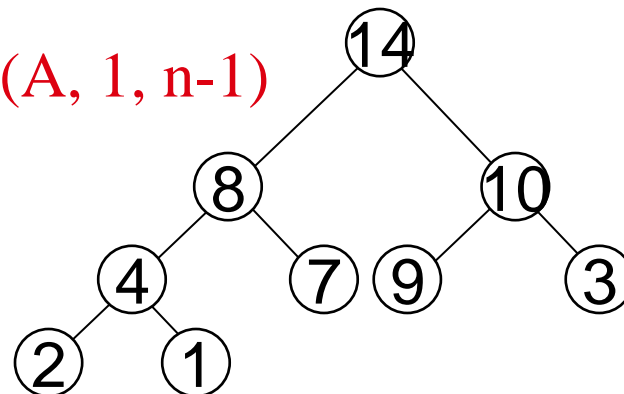


max = 16



Kích thước đống giảm đi 1

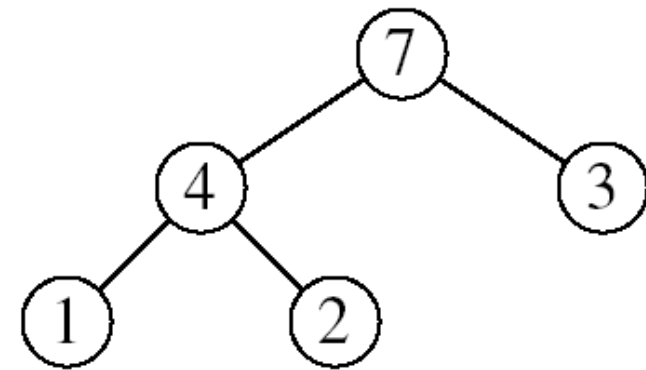
Thực hiện Max-Heapify(A, 1, n-1)



Heap-Extract-Max

Alg: Heap-Extract-Max(A, n)

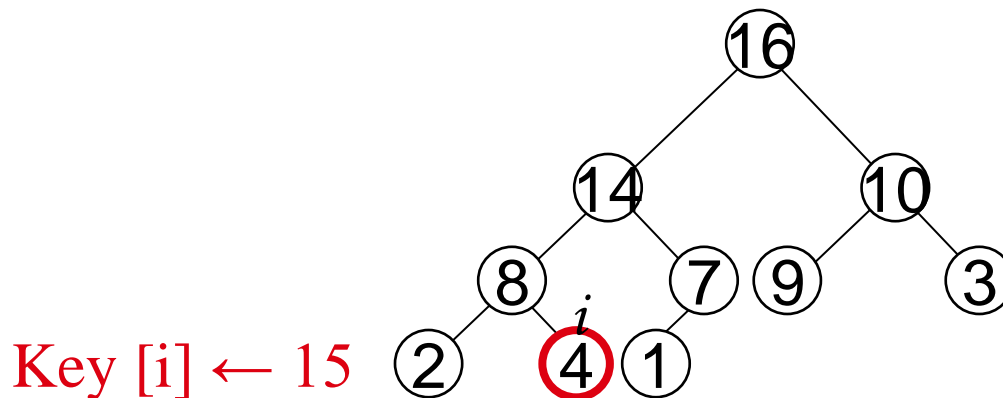
1. **if** $n < 1$
2. **then error** “heap underflow”
3. $\text{max} \leftarrow A[1]$
4. $A[1] \leftarrow A[n]$
5. Max-Heapify($A, 1, n-1$) // Vun lại đồng
6. **return** max



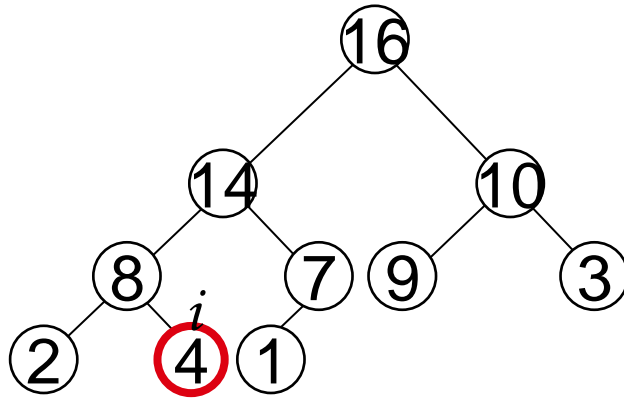
Thời gian tính: $O(\log n)$

Heap-Increase-Key

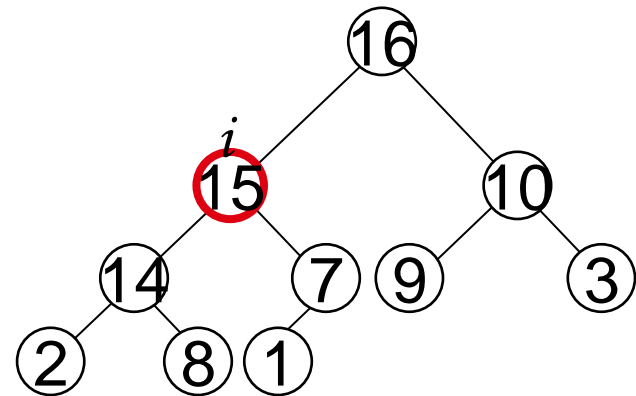
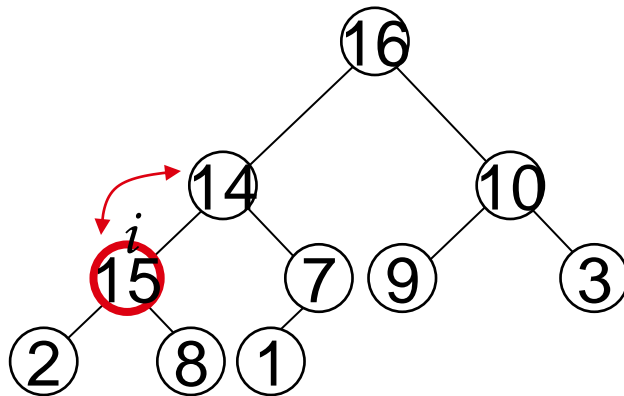
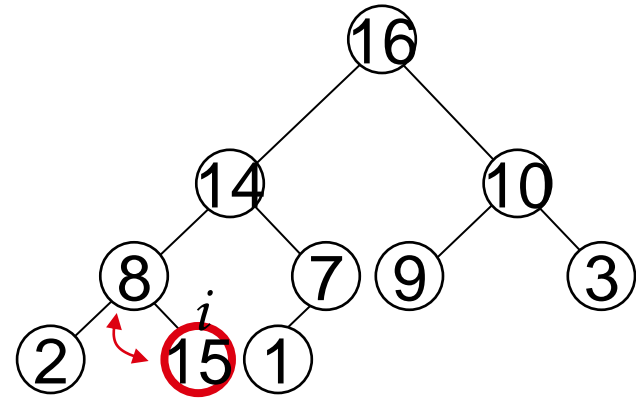
- **Chức năng:** Tăng giá trị khoá của phần tử i trong đống
- **Thuật toán:**
 - Tăng khoá của $A[i]$ thành giá trị mới
 - Nếu tính chất max-heap bị vi phạm: di chuyển theo đường đến gốc để tìm chỗ thích hợp cho khoá mới bị tăng này



Ví dụ: Heap-Increase-Key



$Key[i] \leftarrow 15$

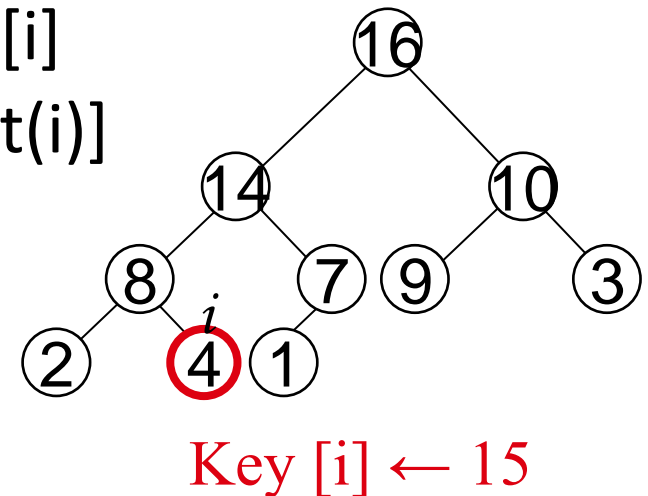


Heap-Increase-Key

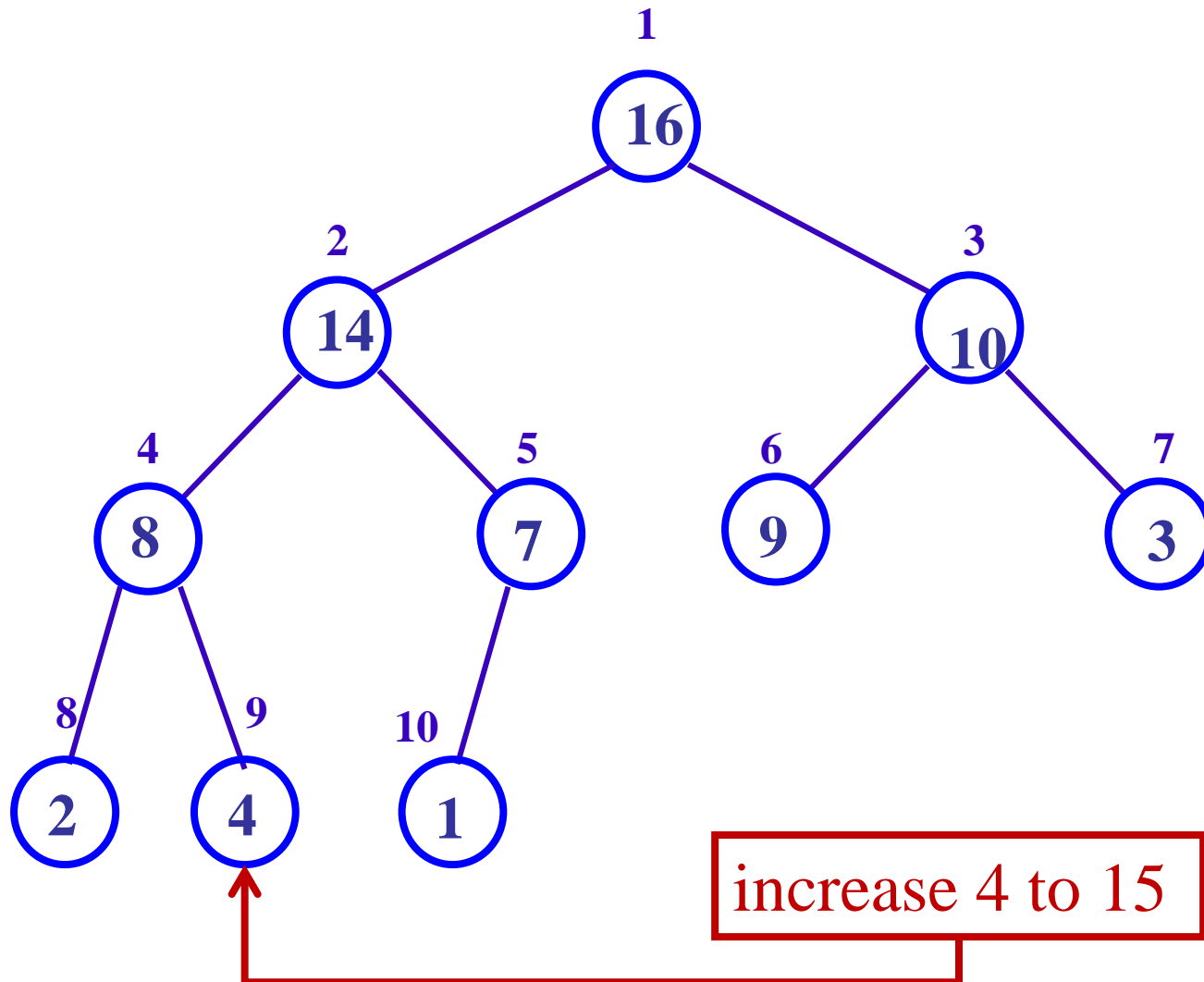
Alg: Heap-Increase-Key(A, i, key)

1. **if** $\text{key} < A[i]$
2. **then error** “khóa mới nhỏ hơn khóa hiện tại”
3. $A[i] \leftarrow \text{key}$
4. **while** $i > 1$ and $A[\text{parent}(i)] < A[i]$
5. **do** hoán đổi $A[i] \leftrightarrow A[\text{parent}(i)]$
6. $i \leftarrow \text{parent}(i)$

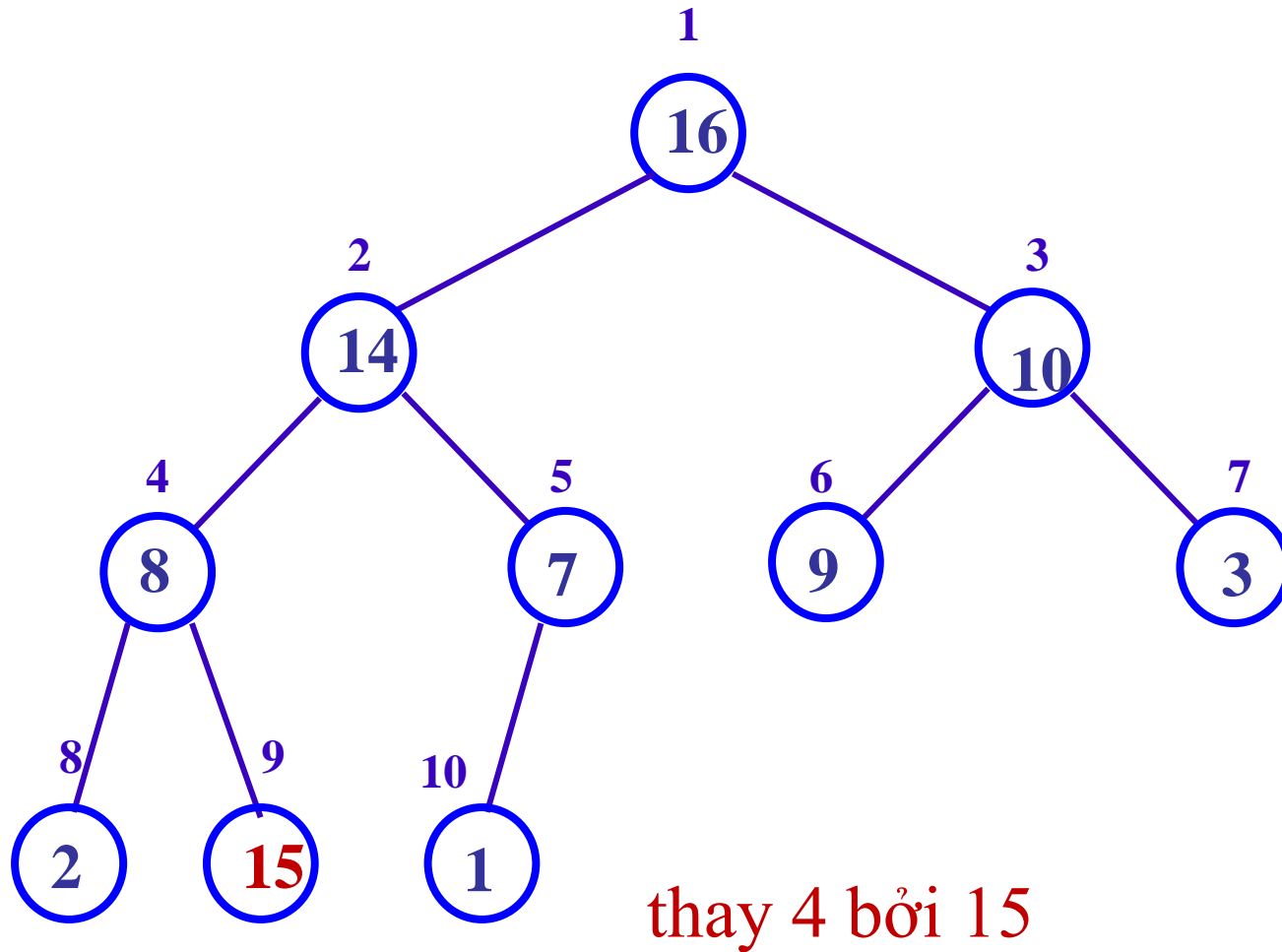
- Thời gian tính: $O(\log n)$



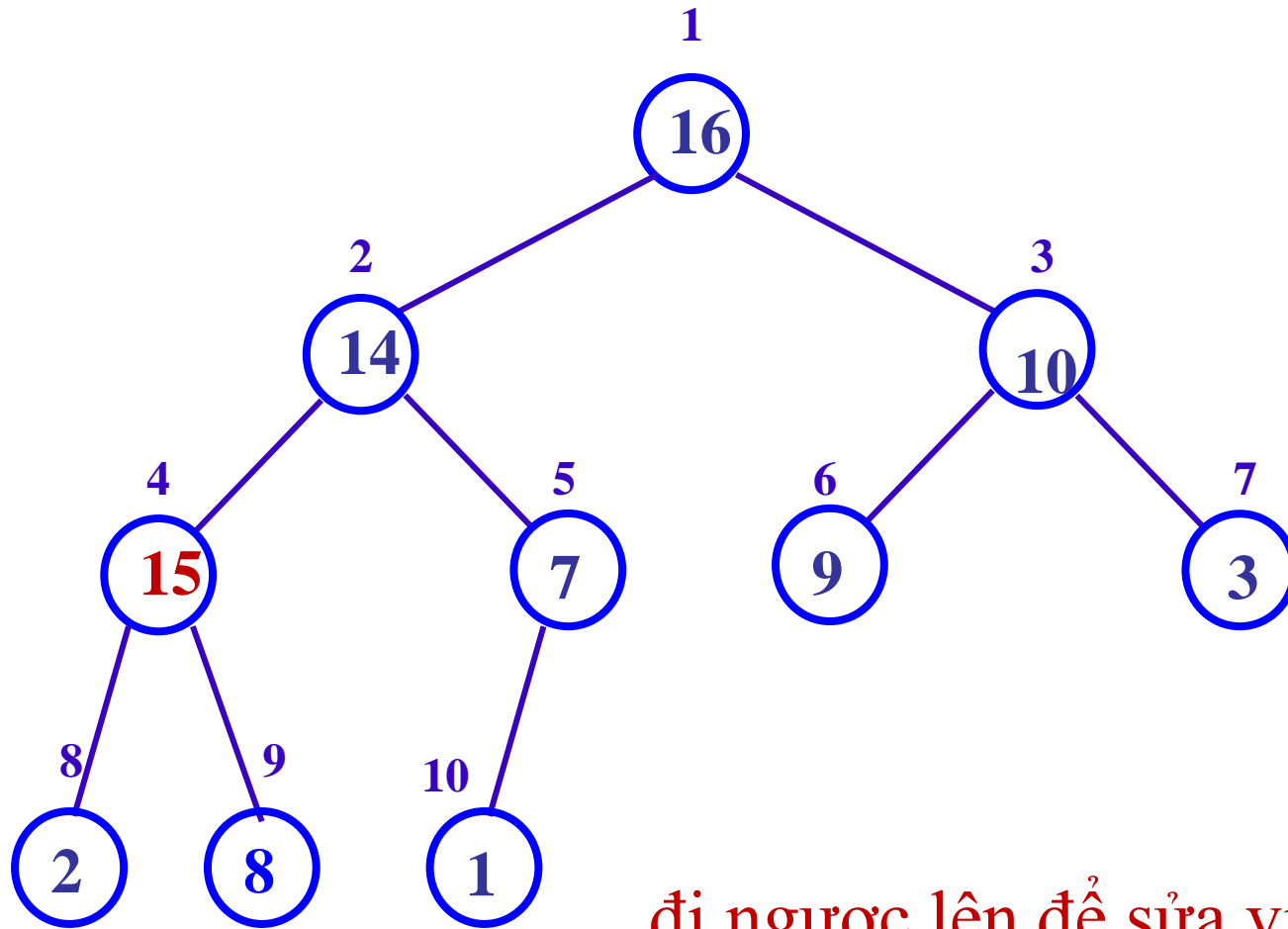
Ví dụ: Heap-Increase-Key (1)



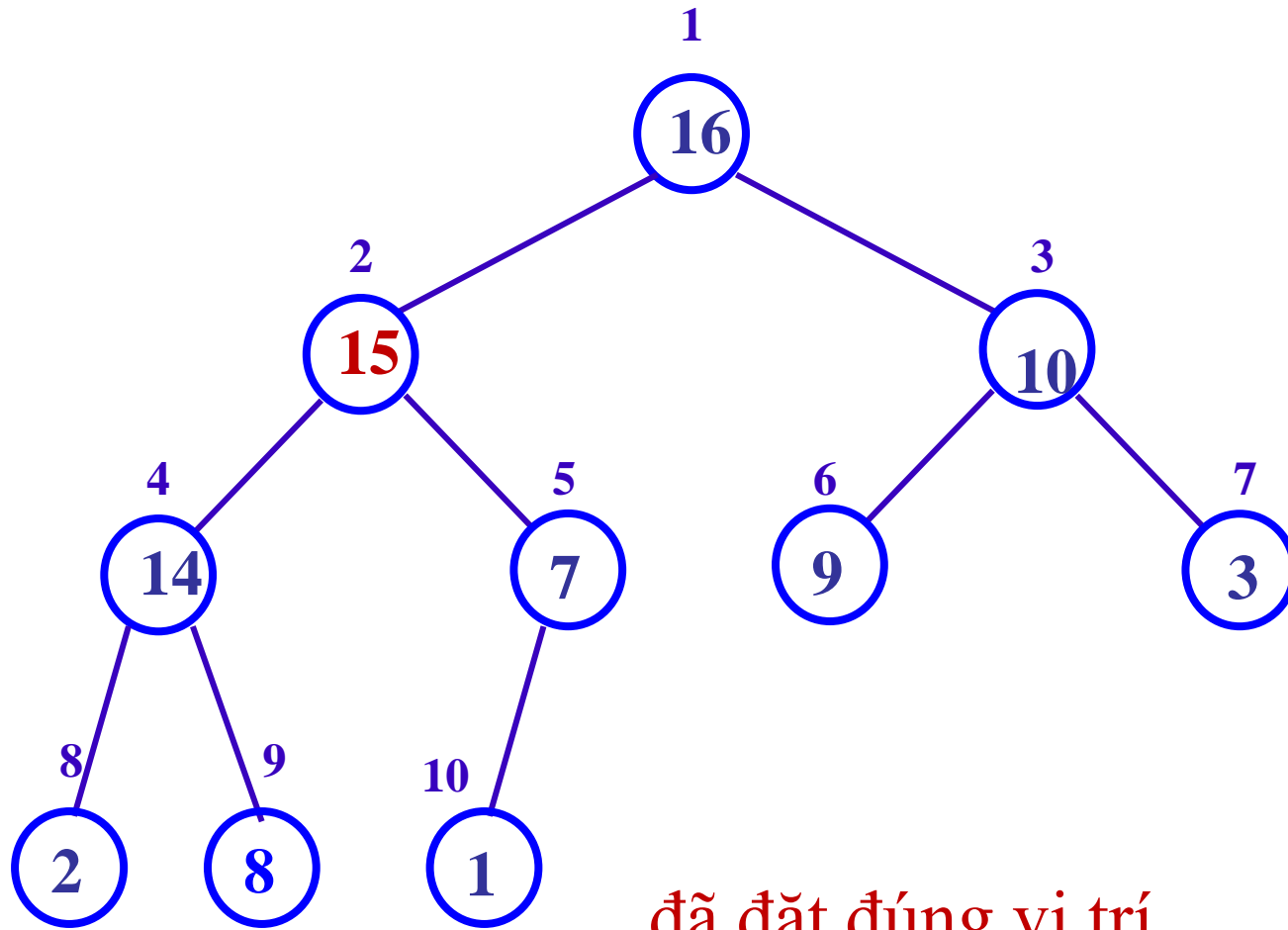
Ví dụ: Heap-Increase-Key (2)



Ví dụ: Heap-Increase-Key (3)

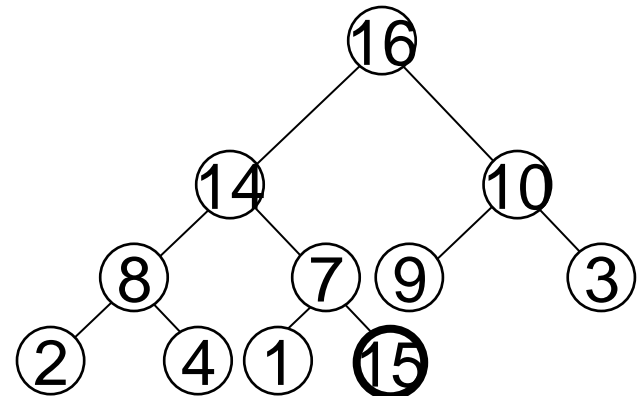
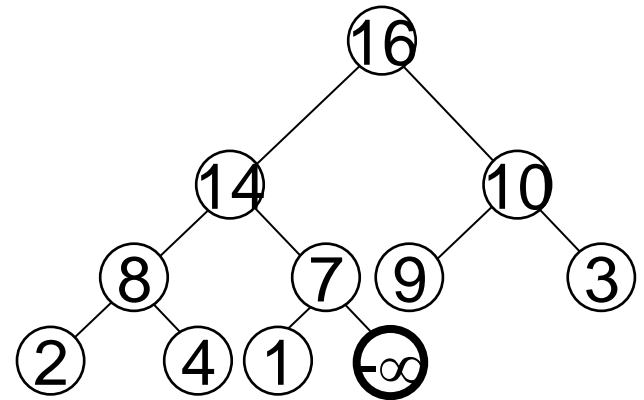


Ví dụ: Heap-Increase-Key (4)



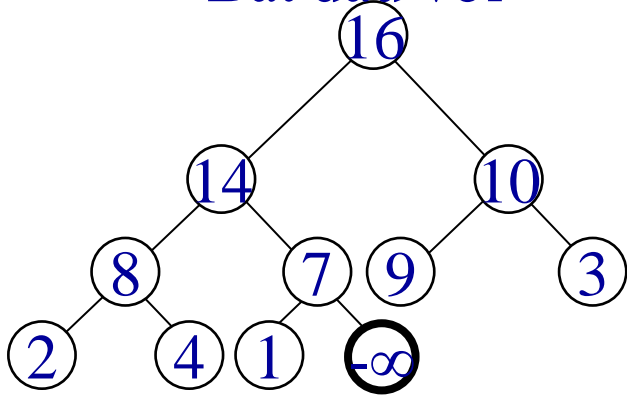
Max-Heap-Insert

- **Chức năng:** Chèn một phần tử mới vào max-heap
- **Thuật toán:**
 - Mở rộng max-heap với một nút mới có khoá là $-\infty$
 - Gọi Heap-Increase-Key để tăng khoá của nút mới này thành giá trị của phần tử mới và vun lại đống

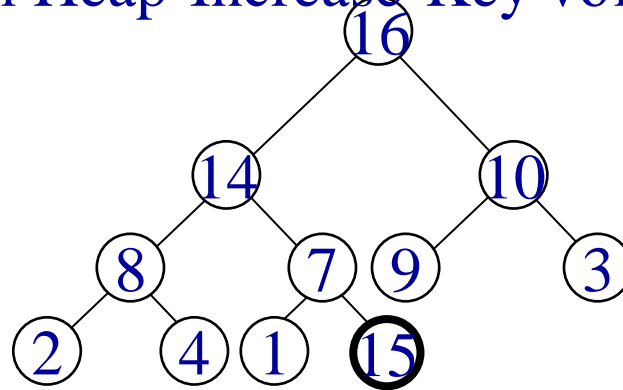


Ví dụ: Max-Heap-Insert

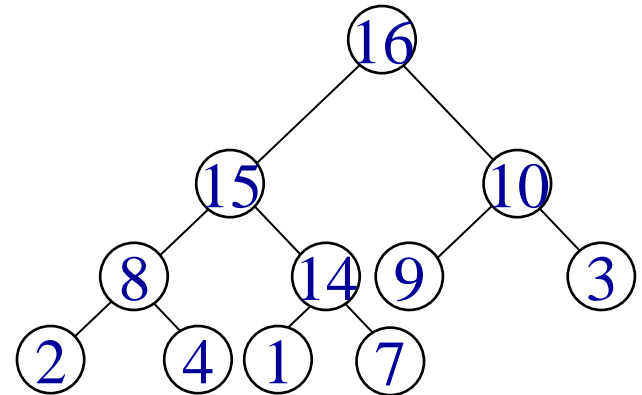
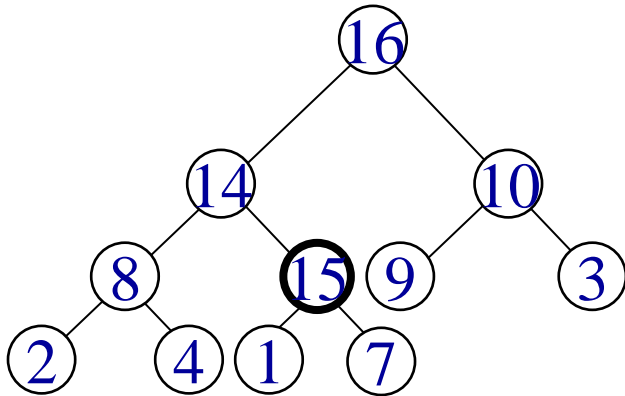
Chèn giá trị 15:
- Bắt đầu với $-\infty$



Tăng khoá thành 15
Gọi Heap-Increase-Key với $A[11] = 15$



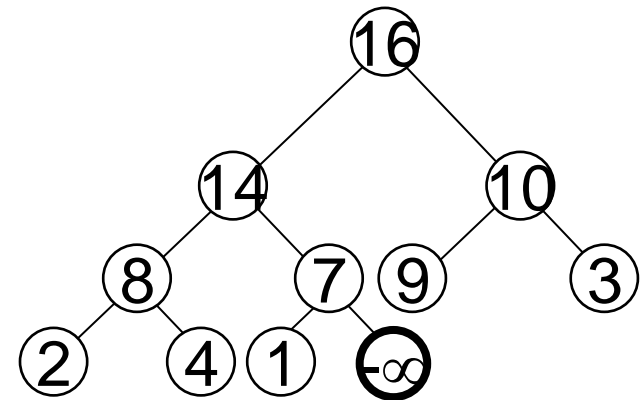
Vun lại đồng
với phần tử
mới bổ sung



Max-Heap-Insert

Alg: Max-Heap-Insert(A , key , n)

1. $heap-size[A] \leftarrow n + 1$
2. $A[n + 1] \leftarrow -\infty$
3. Heap-Increase-Key(A , $n + 1$, key)



Running time: $O(\log n)$

Tổng kết

- Chúng ta có thể thực hiện các phép toán sau đây với đống:

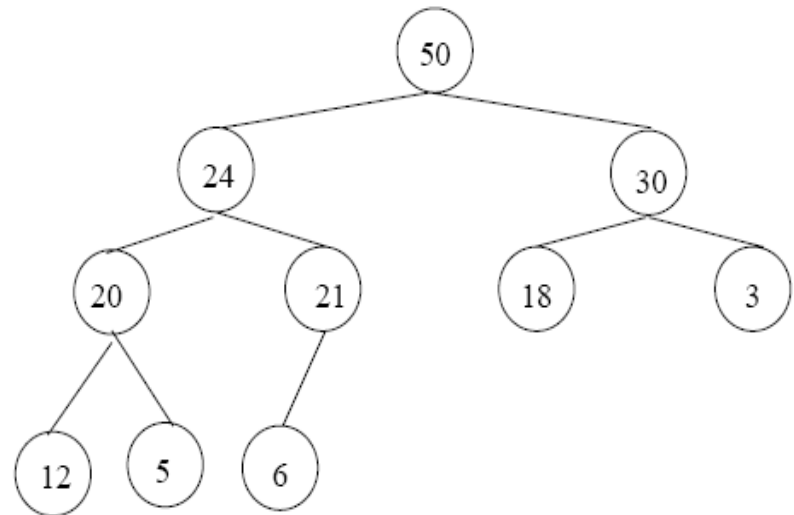
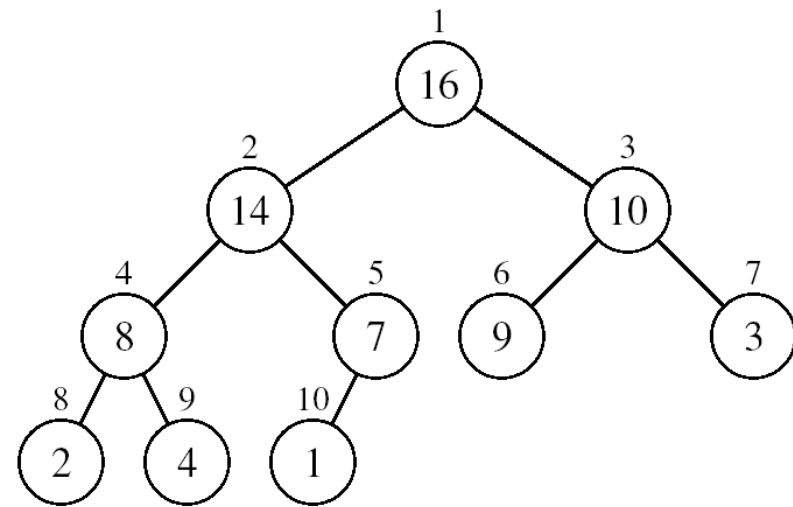
Phép toán

Thời gian tính

- | | |
|---------------------|---------------|
| • Max-Heapify | $O(\log n)$ |
| • Build-Max-Heap | $O(n)$ |
| • Heap-Sort | $O(n \log n)$ |
| • Max-Heap-Insert | $O(\log n)$ |
| • Heap-Extract-Max | $O(\log n)$ |
| • Heap-Increase-Key | $O(\log n)$ |
| • Heap-Maximum | $O(1)$ |

Câu hỏi

- Giả sử các số trong max-heap là phân biệt, phần tử lớn thứ hai nằm ở đâu?



- Ans:** Con lớn hơn trong hai con của gốc

Mã Huffman: Thuật toán

```
procedure Huffman(C, f);  
begin  
  n ← |C|;  
  Q ← C;  
  for i:=1 to n-1 do  
    begin  
      x, y ← 2 chữ cái có tần suất nhỏ nhất trong Q; (* Thao tác 1 *)  
      Tạo nút p với hai con x, y;  
      f(p) := f(x) + f(y);  
      Q ← Q \ {x, y} ∪ {p} (* Thao tác 2 *)  
    end;  
  end;
```

- Cài đặt trực tiếp:
 - Thao tác 1: $O(n)$
 - Thao tác 2: $O(1)$
- => Thời gian $O(n^2)$

Mã Huffman: Thuật toán

```
procedure Huffman(C, f);
begin
  n ← |C|;
  Q ← C;
  for i:=1 to n-1 do
    begin
      x, y ← 2 chữ cái có tần suất nhỏ nhất trong Q; (* Thao tác 1 *)
      Tạo nút p với hai con x, y;
      f(p) := f(x) + f(y);
      Q ← Q \ {x, y} ∪ {p} (* Thao tác 2 *)
    end;
  end;
```

- Cài đặt sử dụng **priority queue** Q:
 - Thao tác 1 và 2: $O(\log n)$
- => Thời gian $O(n \log n)$

Có thể sắp xếp nhanh đến mức độ nào?

- Heapsort, Mergesort, và Quicksort có thời gian $O(n \log n)$ là các thuật toán có thời gian tính tốt nhất trong các thuật toán trình bày ở trên
- Liệu có thể phát triển được thuật toán tốt hơn không?
- Câu trả lời là: "Không, nếu như phép toán cơ bản là phép so sánh".

Tổng kết:

Các thuật toán sắp xếp dựa trên phép so sánh

Name	Average	Worst	In place	Stable
Bubble sort	—	$O(n^2)$	Yes	Yes
Selection sort	$O(n^2)$	$O(n^2)$	Yes	No
Insertion sort	$O(n + d)$	$O(n^2)$	Yes	Yes
Merge sort	$O(n \log n)$	$O(n \log n)$	No	Yes
Heapsort	$O(n \log n)$	$O(n \log n)$	Yes	No
Quicksort	$O(n \log n)$	$O(n^2)$	No	No

QUESTIONS?

