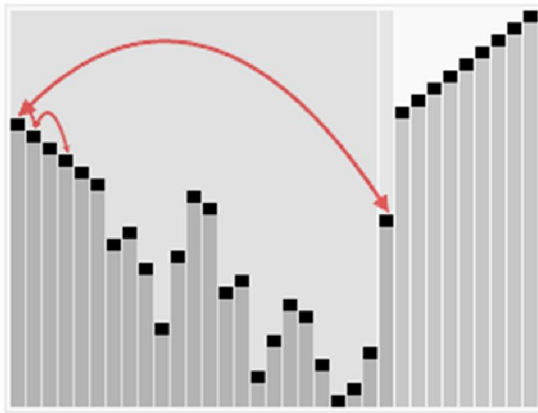
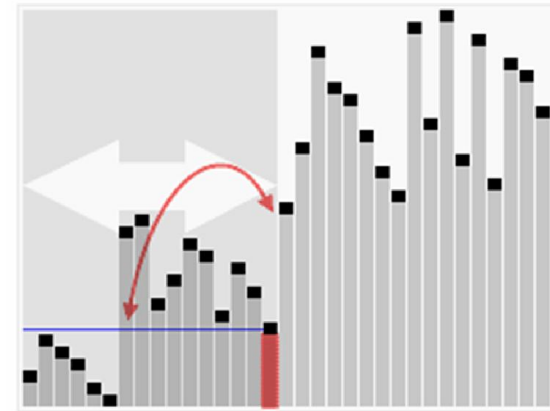


Chương 5

Sắp xếp (Sorting)



Heap Sort



Quick Sort

William A. Martin, *Sorting*. ACM Computing Surveys, Vol. 3, Nr 4, Dec 1971, pp. 147-174.
" ...The bibliography appearing at the end of this article lists 37 sorting algorithms and 100 books and papers on sorting published in the last 20 years...
Suggestions are made for choosing the algorithm best suited to a given situation."

D. Knuth: 40% thời gian hoạt động của các máy tính là dành cho sắp xếp!

Bài giảng của
PGS.TS. NGUYỄN ĐỨC NGHĨA

Khoa học Máy tính
Đại học Bách khoa Hà nội

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)
 - Là quá trình tổ chức lại họ các dữ liệu theo thứ tự giảm dần hoặc tăng dần (ascending or descending order)
- Dữ liệu cần sắp xếp có thể là
 - Số nguyên (Integers)
 - Xâu ký tự (Character strings)
 - Đối tượng (Objects)
- Khoá sắp xếp (Sort key)
 - Là bộ phận của bản ghi xác định thứ tự sắp xếp của bản ghi trong họ các bản ghi.
 - Ta cần sắp xếp các bản ghi theo thứ tự của các khoá.

5.1.1. Bài toán sắp xếp

- **Chú ý:**
- Việc sắp xếp tiến hành trực tiếp trên bản ghi đòi hỏi di chuyển vị trí bản ghi, có thể là thao tác rất tốn kém.
- Vì vậy, người ta thường xây dựng bảng khoá gồm các bản ghi chỉ có hai trường là (khoá, con trỏ)
 - trường "khoá" chứa giá trị khoá,
 - trường "con trỏ" để ghi địa chỉ của bản ghi tương ứng.
- Việc sắp xếp theo khoá 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

Ta có thể hạn chế xét bài toán sắp xếp dưới dạng sau đây:

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

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

- **Ứng dụng của sắp xếp:**

- Quản trị cơ sở dữ liệu (Database management);
- Khoa học và kỹ thuật (Science and engineering);
- Các thuật toán lập lịch (Scheduling algorithms),
 - ví dụ thiết kế chương trình dịch, truyền thông,... (compiler design, telecommunication);
- Máy tìm kiếm web (Web search engine);
- 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 họ dữ liệu được đưa toàn bộ vào bộ nhớ trong của máy tính
 - Sắp xếp ngoài (external sort)
 - Họ 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
- **Các đặc trưng của một thuật toán sắp xếp:**
 - 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

- Có hai phép toán cơ bản mà thuật toán sắp xếp 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 và false nếu trái lại.
- Các thuật toán chỉ sử dụng phép toán so sánh để xác định thứ tự giữa hai phần tử được gọi là thuật toán sử dụng phép so sánh (*Comparison-based sorting algorithm*)

NỘI DUNG

5.1. Bài toán sắp xếp

5.2. Ba thuật toán sắp xếp cơ bản

5.3. Sắp xếp trộn (Merge Sort)

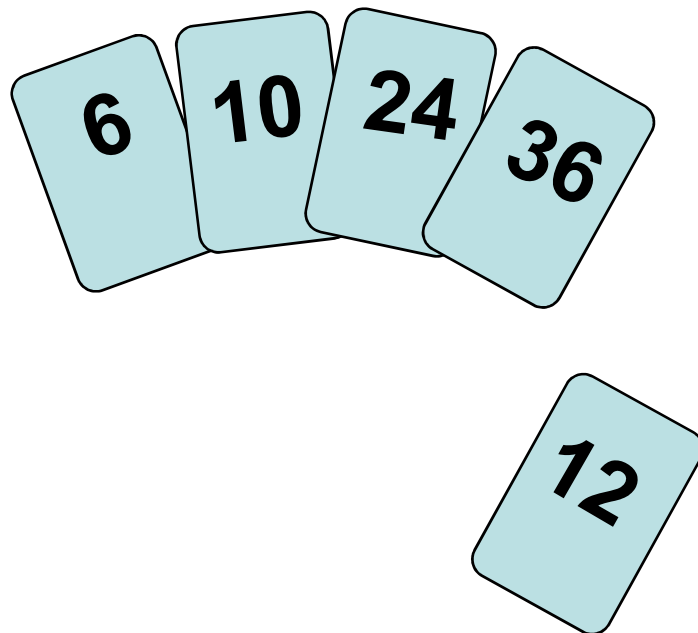
5.4. Sắp xếp nhanh (Quick Sort)

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)

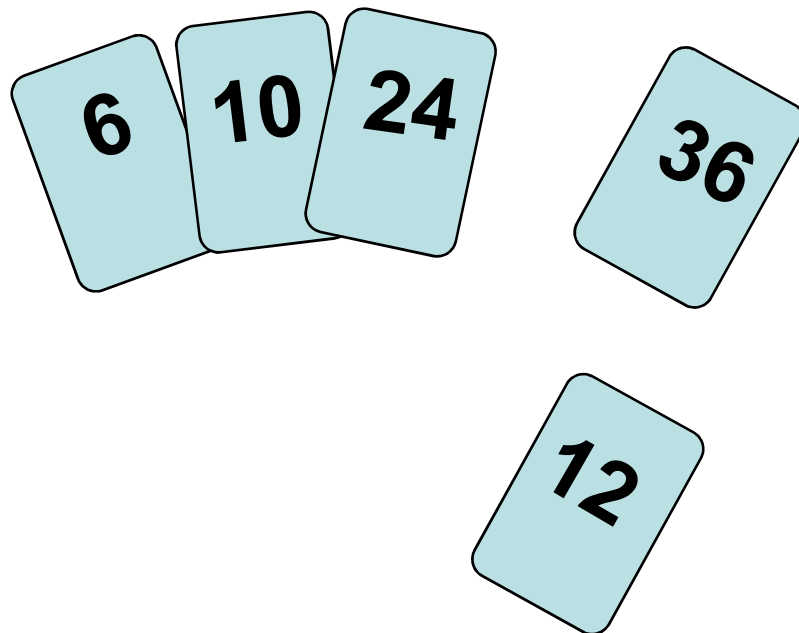
5.2.1. Sắp xếp chèn (Insertion Sort)



Phỏng theo cách làm của người chơi bài khi cần "chèn" thêm một con bài vào bộ bài đã được sắp xếp trên tay.

Để chèn 12, ta cần tạo chỗ cho nó bởi việc dịch chuyển đầu tiên là 36 và sau đó là 24.

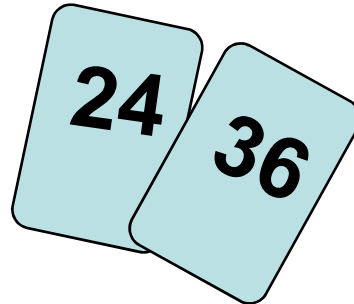
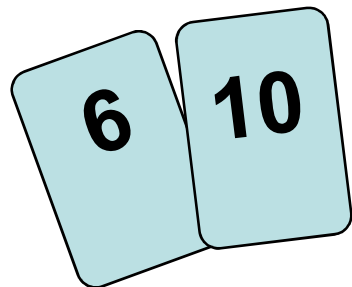
Sắp xếp chèn (Insertion Sort)



Phỏng theo cách làm của người chơi bài khi cần "chèn" thêm một con bài vào bộ bài đã được sắp xếp trên tay.

Để chèn 12, ta cần tạo chỗ cho nó bởi việc dịch chuyển đầu tiên là 36 và sau đó là 24.

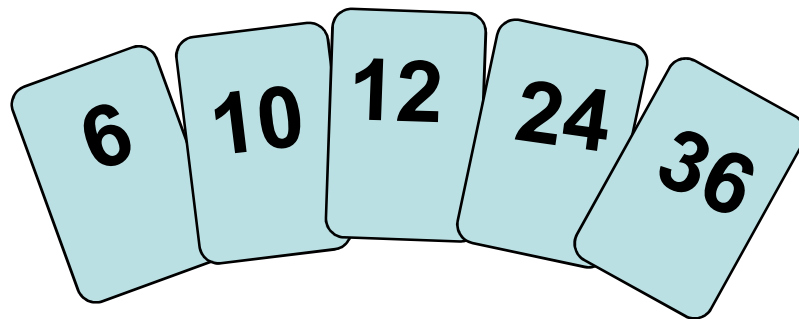
Sắp xếp chèn (Insertion Sort)



Phỏng theo cách làm của người chơi bài khi cần "chèn" thêm một con bài vào bộ bài đã được sắp xếp trên tay.

Để chèn 12, ta cần tạo chỗ cho nó bởi việc dịch chuyển đầu tiên là 36 và sau đó là 24.

Sắp xếp chèn (Insertion Sort)



Phỏng theo cách làm của người chơi bài khi cần "chèn" thêm một con bài vào bộ bài đã được sắp xếp trên tay.

Để chèn 12, ta cần tạo chỗ cho nó bởi việc dịch chuyển đầu tiên là 36 và sau đó là 24.

Don't start coding

You must design a working algorithm first.



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; }
```

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

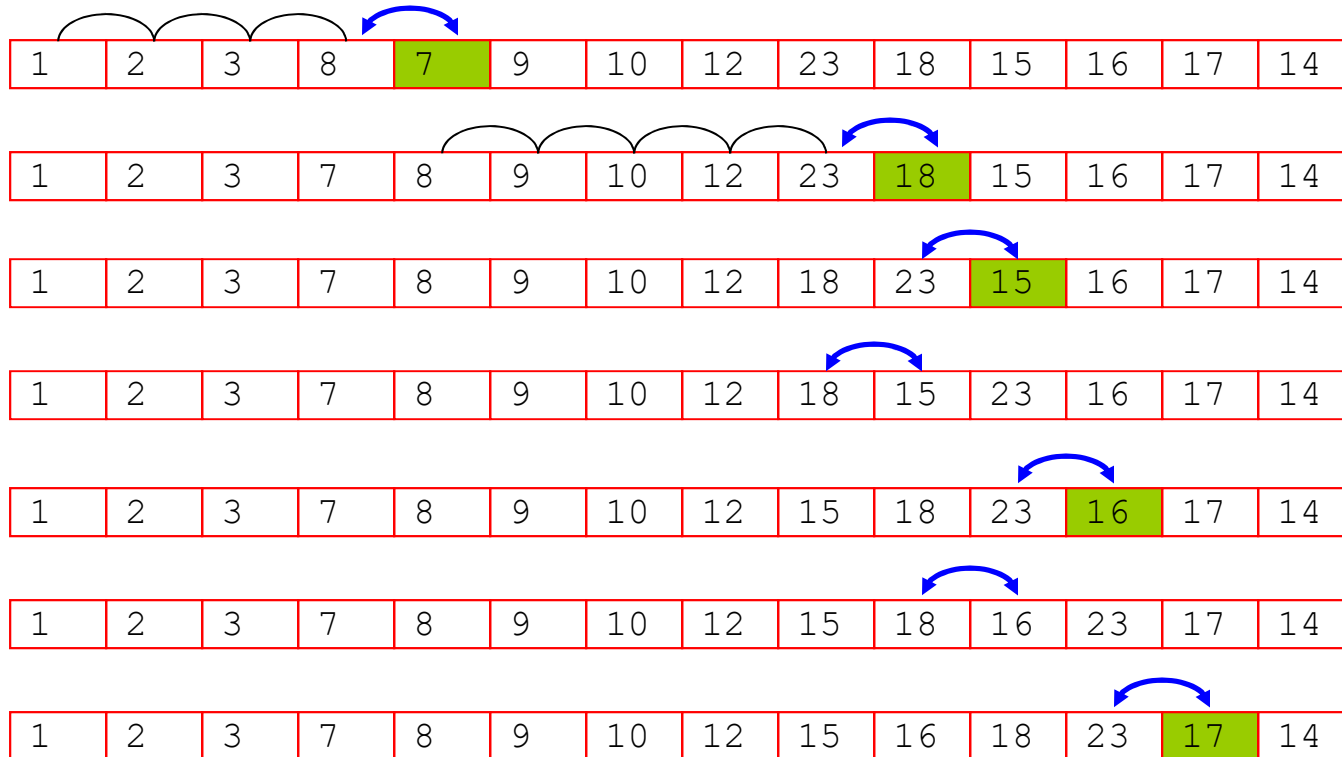
```
    } // end for
```

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

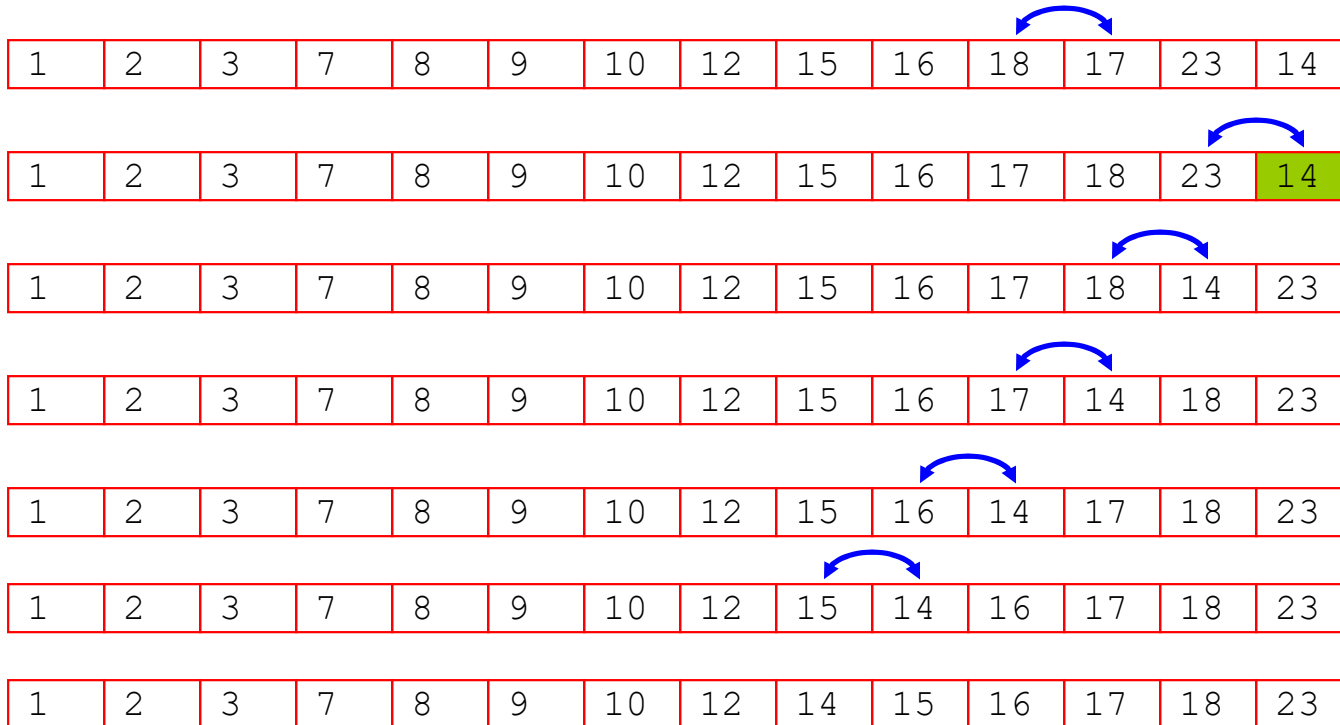
Giải thích:



- 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)

	i=1	2	3	4	5	6	7
42	20	17	13	13	13	13	13
20	42	20	17	17	14	14	14
17	17	42	20	20	17	17	15
13	13	13	42	28	20	20	17
28	28	28	28	42	28	23	20
14	14	14	14	14	42	28	23
23	23	23	23	23	23	42	28
15	15	15	15	15	15	15	42

Các đặc tính của Insertion Sort

- Sắp xếp chèn là tại chỗ và ổn định (In place and Stable)
- Phân tích thời gian tính của thuật toán
 - **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
- Thuật toán này có thời gian tính trong tình huống tốt nhất là tốt nhất
- Là thuật toán sắp xếp tốt đối với dãy đã gần được sắp xếp
 - Nghĩa là mỗi phần tử đã đứng ở vị trí rất gần vị trí trong thứ tự cần sắp xếp

Thử nghiệm insertion sort

```
#include <stdlib.h>
#include <stdio.h>
void insertionSort(int a[], int array_size);
int a[1000];
int main() {
    int i, N;
    printf("\nGive n = "); scanf("%i", &N);
    //seed random number generator
    srand(getpid());
    //fill array with random integers
    for (i = 0; i < N; i++)
        a[i] = rand();
    //perform insertion sort on array
    insertionSort(a, N);
    printf("\nOrdered sequence:\n");
    for (i = 0; i < N; i++)
        printf("%8i", a[i]);
    getch();
}
```

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

```
template <class Elem, class Comp>
void selection_sort(Elem A[], int n) {
    for (int i=0; i<n-1; i++) {
        int lowindex = i; // Remember its index
        for (int j=n-1; j>i; j--) // Find least
            if (Comp::lt(A[j], A[lowindex]))
                lowindex = j; // Put it in place
        swap(A, i, lowindex);
    }
}
```

- **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 của sắp xếp chọn là số phép đổi chỗ là ít. Điều này là có ý nghĩa nếu như việc đổi chỗ là 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>	<u>14</u>	<u>14</u>	<u>14</u>	<u>14</u>	<u>14</u>
17	17	<u>17</u>	<u>15</u>	<u>15</u>	<u>15</u>	<u>15</u>	<u>15</u>
13	42	42	<u>42</u>	<u>17</u>	<u>17</u>	<u>17</u>	<u>17</u>
28	28	28	28	<u>28</u>	<u>20</u>	<u>20</u>	<u>20</u>
14	<u>14</u>	<u>20</u>	<u>20</u>	<u>20</u>	<u>28</u>	<u>23</u>	<u>23</u>
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

- *Bubble sort* là phương pháp sắp xếp đơn giản thường được sử dụng như ví dụ minh họa cho các giáo trình nhập môn lập trình.
- 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.
- Mặc dù thuật toán này là đơn giản, nhưng nó là thuật toán kém hiệu quả nhất trong ba thuật toán cơ bản trình bày trong mục này. Vì thế ngoài mục đích giảng dạy, Sắp xếp nổi bọt rất ít khi được sử dụng.
- Giáo sư **Owen Astrachan** (Computer Science Department, Duke University) còn đề nghị là không nên giảng dạy về thuật toán này.

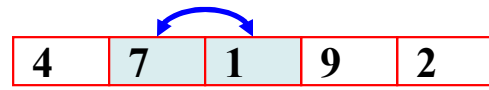
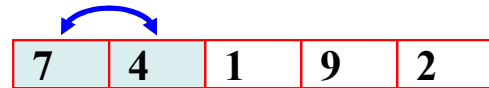
5.2.3. Sắp xếp nổi bọt - Bubble Sort


```
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]);
        }
    }
}
```

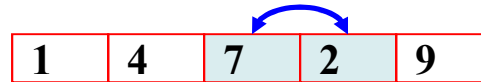
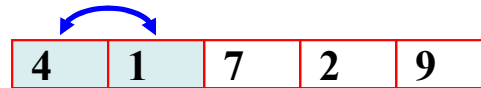
```
void swap(int &a, int &b)
{
    int temp = a;
    a = b;
    b = temp;
}
```


- 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.

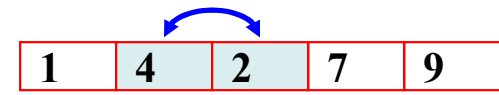
Ví dụ: Bubble Sort

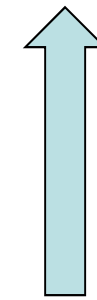



 $i = 4$




 $i = 3$

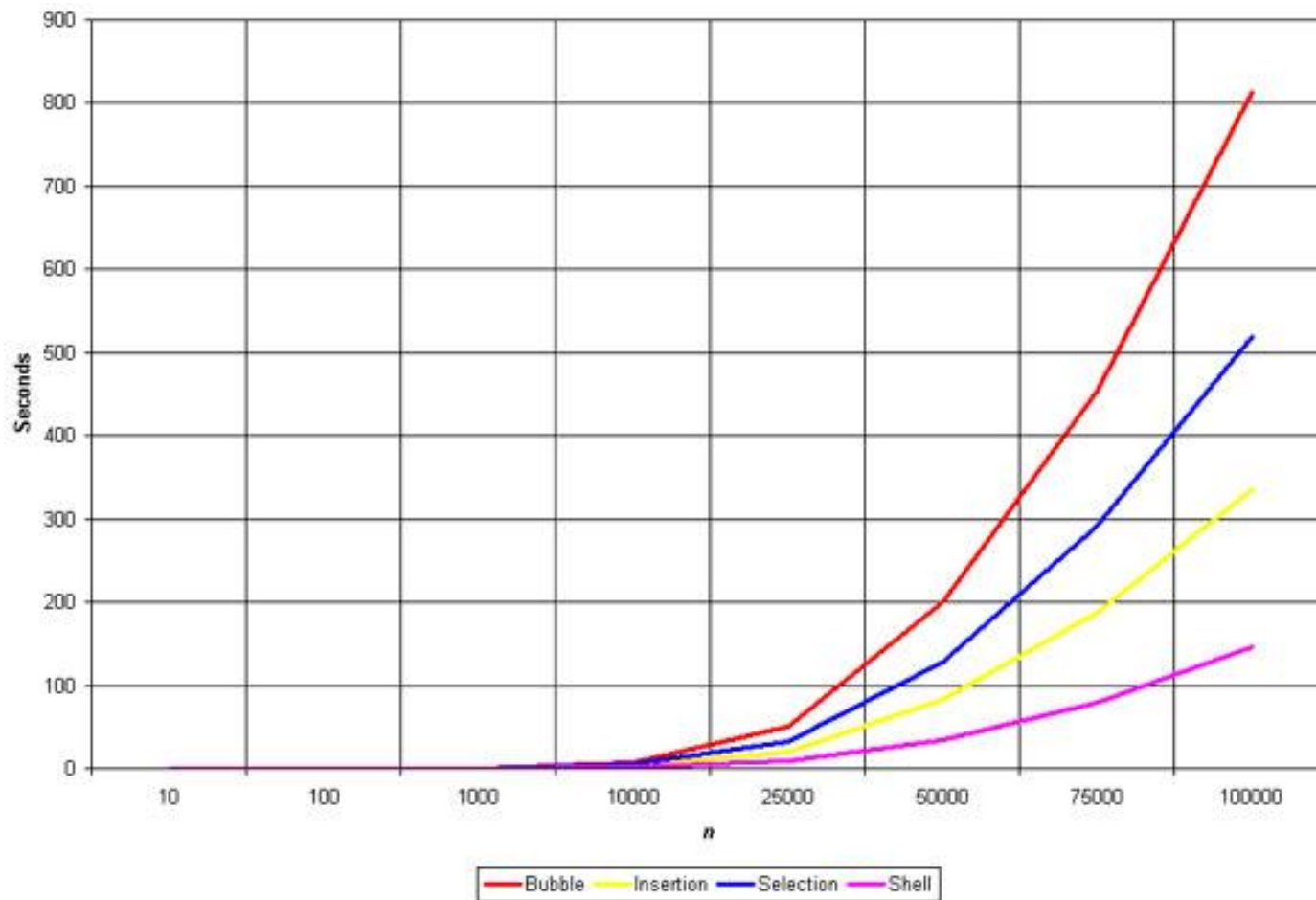



 $i = 2$

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)



Hoà nhập hai dòng xe theo Khoá
[độ liều lĩnh của lái xe].
Lái xe liều lĩnh hơn sẽ vào trước!

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

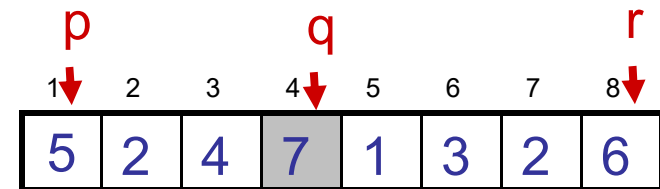
MERGE-SORT(A, p, q)

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

MERGE(A, p, q, r)

endif

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



▷ Kiểm tra điều kiện neo

▷ Chia (Divide)

▷ Trị (Conquer)

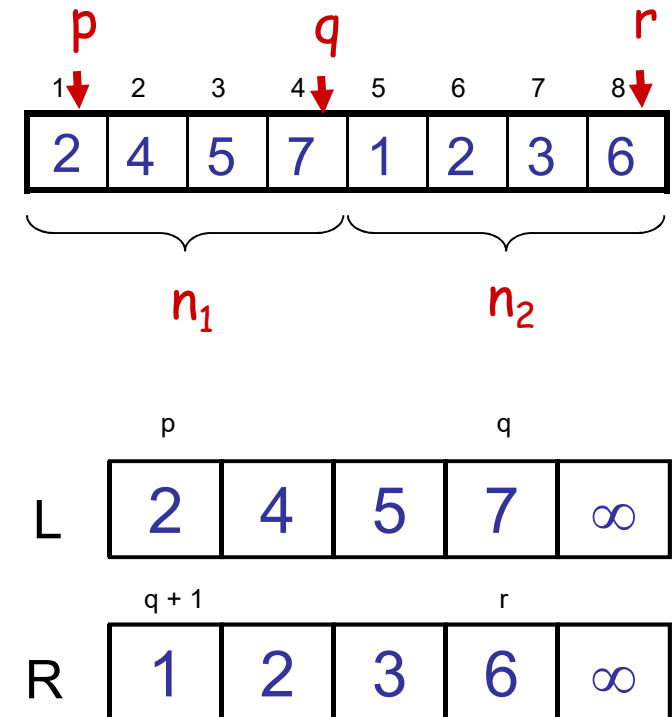
▷ Trị (Conquer)

▷ Tổ hợp (Combine)

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 hai 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:**

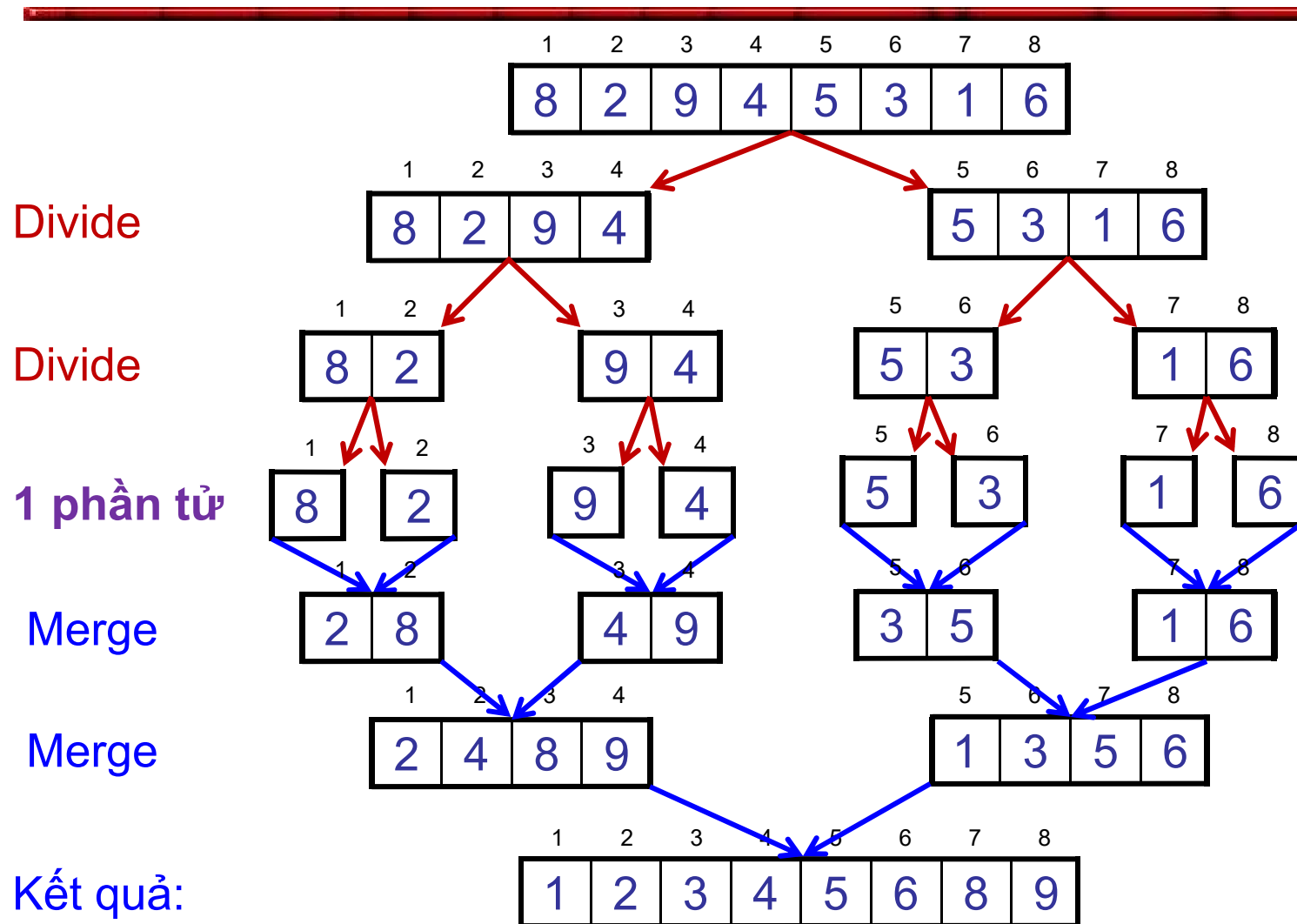
- TRỘN (MERGE) trên các mảng con cỡ n phần tử đòi hỏi thời gian $\Theta(n)$

$$\Rightarrow C(n) = \Theta(n)$$

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

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

Ví dụ: Sắp xếp trộn



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

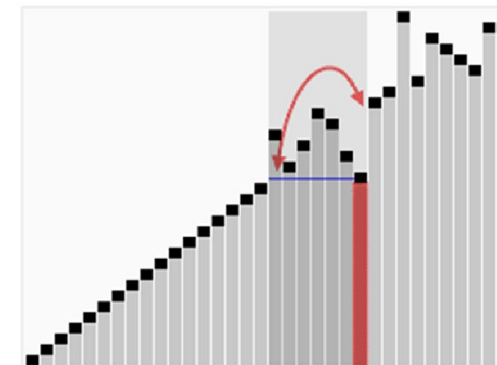
5.4.1. Sơ đồ Quick Sort

- Thuật toán sắp xếp nhanh được 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 (sẽ viết tắt là 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 $O(n^2)$.
- QS là thuật toán sắp xếp tại chỗ, nhưng nó không có tính ổn định.
- QS 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

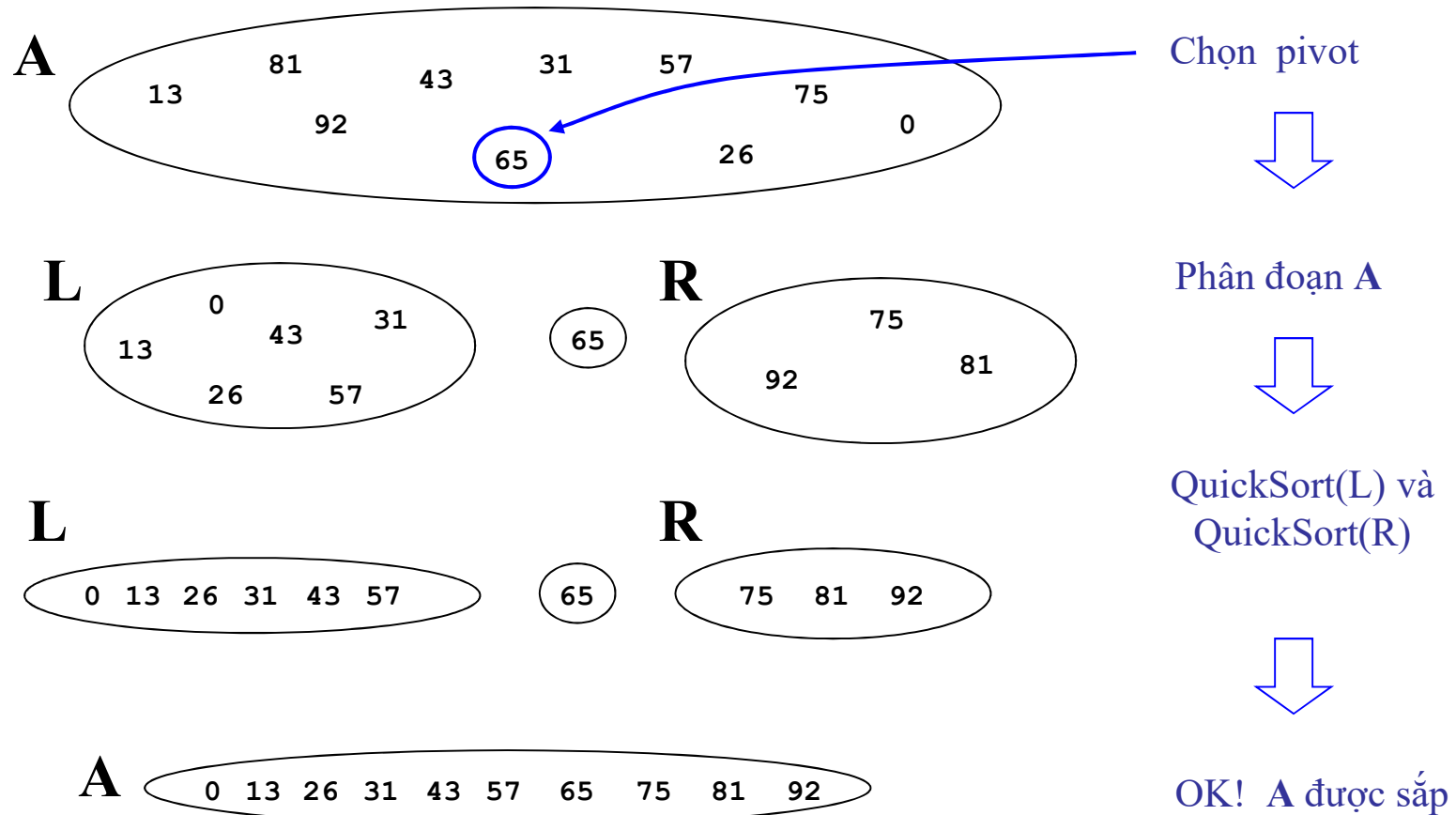


đường nằm ngang cho biết pivot

5.4.1. Sơ đồ Quick Sort

- Quick sort là thuật toán sắp xếp được phát triển dựa trên kỹ thuật chia để trị.
- Thuật toán có thể mô tả đệ qui như sau (có dạng tương tự như merge sort):
 1. **Neo đệ qui** (Base case). Nếu dãy chỉ còn không quá một phần tử thì nó là dãy được sắp và trả lại ngay dãy này mà không phải làm gì cả.
 2. **Chia** (Divide):
 - Chọn một phần tử trong dãy và gọi nó là **phần tử chốt p** (pivot).
 - Chia dãy đã cho ra thành hai dãy con: Dãy con trái (L) gồm những phần tử không lớn hơn phần tử chốt, còn dãy con phải (R) gồm các phần tử không nhỏ hơn phần tử chốt. Thao tác này được gọi là "**Phân đoạn**" (Partition).
 3. **Trị** (Conquer): Lặp lại một cách đệ qui thuật toán đối với hai dãy con L và R .
 4. **Tổng hợp** (Combine): Dãy được sắp xếp là $L p R$.
- 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.

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. $Pivot = \text{Partition}(A, Left, Right);$
3. $\text{Quick-Sort}(A, Left, Pivot - 1);$
4. $\text{Quick-Sort}(A, Pivot + 1, Right);$ }

- Hàm $\text{Partition}(A, Left, Right)$ thực hiện chia $A[Left..Right]$ thành hai đoạn $A[Left..Pivot - 1]$ và $A[Pivot + 1..Right]$ sao cho:
 - Các phần tử trong $A[Left..Pivot - 1]$ là nhỏ hơn hoặc bằng $A[Pivot]$
 - Các phần tử trong $A[Pivot + 1..Right]$ là lớn hơn hoặc bằng $A[Pivot]$.
- Lệnh gọi thực hiện thuật toán **Quick-Sort($A, 1, n$)**

Sơ đồ tổng quát của QS

- Knuth cho rằng khi dãy con chỉ còn một số lượng không lớn phần tử (theo ông là không quá 9 phần tử) thì ta nên 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. $Pivot = \text{Partition}(A, Left, Right)$;
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 hai dãy con: Dãy con trái (L) gồm những phần tử không lớn hơn phần tử chốt, còn dãy con phải (R) gồm các phần tử không nhỏ hơn phần tử chốt.
- 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 $O(n^2)$. Trường hợp xấu nhất xảy ra khi danh sách 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, thì 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. 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 (Knuth).
 - Chọn ngẫu nhiên một phần tử làm phần tử chốt.

Thuật toán phân đoạn

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

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** $j = j - 1;$

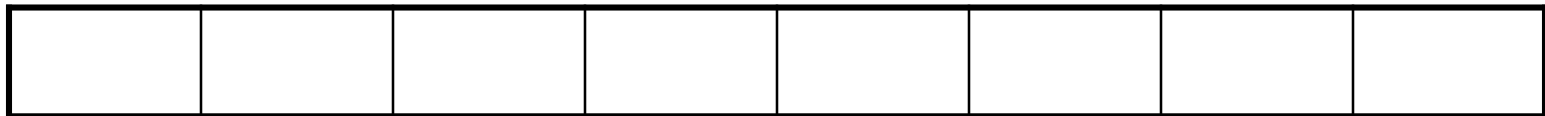
$\text{swap}(a[i], a[j]);$

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

return $j;$

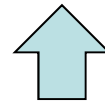
← pivot được chọn là
phần tử đứng đầu

← 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:

7	2	8	3	5	9	6
---	---	---	---	---	---	---

Phân đoạn: Con trỏ

7	2	8	3	5	9	6
---	---	---	---	---	---	---

<↑ >↑

2 nhỏ hơn pivot

7	2	8	3	5	9	6
---	---	---	---	---	---	---

<↑ >↑

đổi chỗ 6, 8

7	2	6	3	5	9	8
---	---	---	---	---	---	---

<↑ >↑

3,5 nhỏ hơn 9 lớn hơn

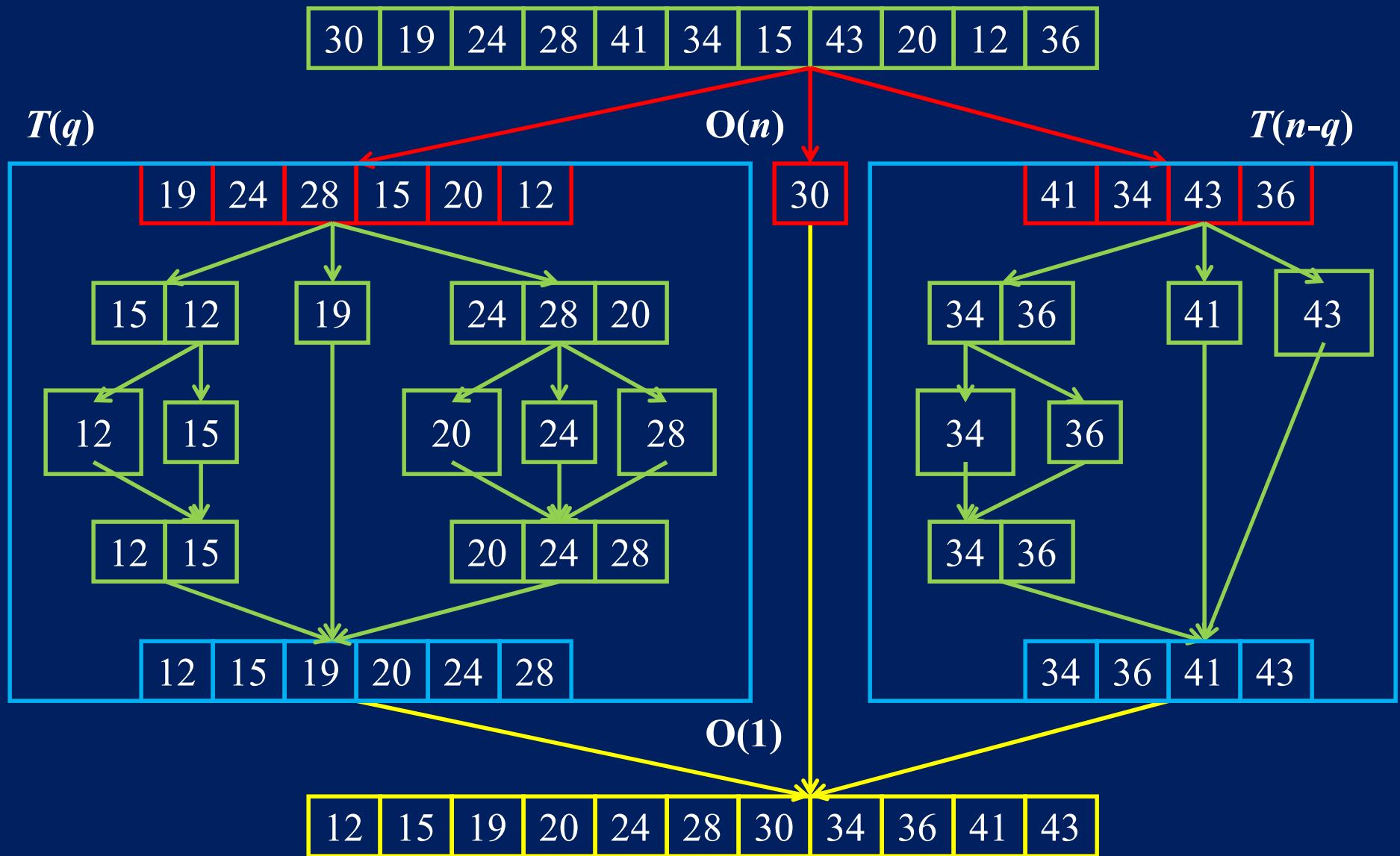
7	2	6	3	5	9	8
---	---	---	---	---	---	---

Kết thúc phân đoạn

7	2	6	3	5	9	8
---	---	---	---	---	---	---

Đưa pivot vào vị trí

5	2	6	3	7	9	8
---	---	---	---	---	---	---



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

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

Thời gian tính trung bình của QS

QuickSort Average Case

- Thời gian tính trung bình =

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

- Tính được

Thời gian tính trung bình $T(N) = O(N \log N)$.

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)

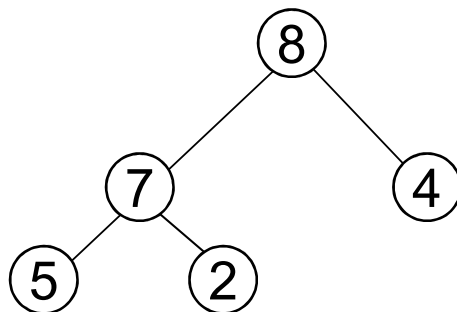
5.5. Sắp xếp vun đống (Heap Sort)

5.5.1. Cấu trúc dữ liệu đống (heap)

5.5.2. Sắp xếp vun đống

5.5.1. The Heap Data Structure

- **Định nghĩa:** **Đống (heap)** là cây nhị phân gần hoàn chỉnh có hai tính chất sau:
 - **Tính cấu trúc (Structural property):** 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 (heap property):** 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 $\text{heapsize}[A]$



Heap

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 đống bởi mảng

- Đống có thể cất giữ trong mảng A .

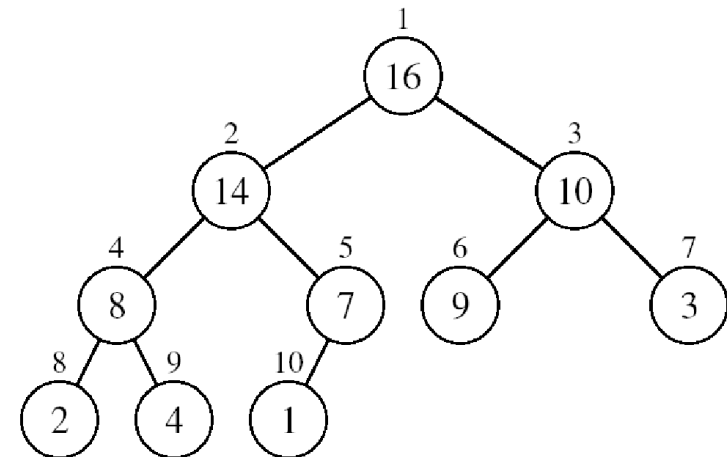
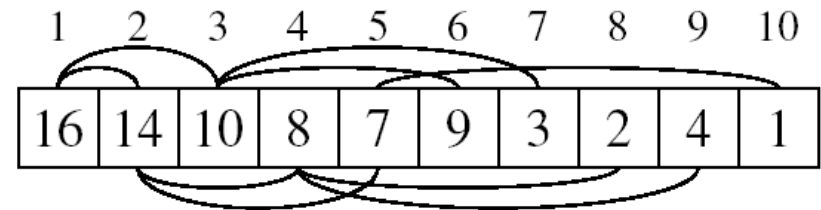
- Gốc của cây là $A[1]$
- Con trái của $A[i]$ là $A[2*i]$
- Con phải của $A[i]$ là $A[2*i + 1]$
- Cha của $A[i]$ là $A[\lfloor i/2 \rfloor]$
- $\text{Heapsize}[A] \leq \text{length}[A]$

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

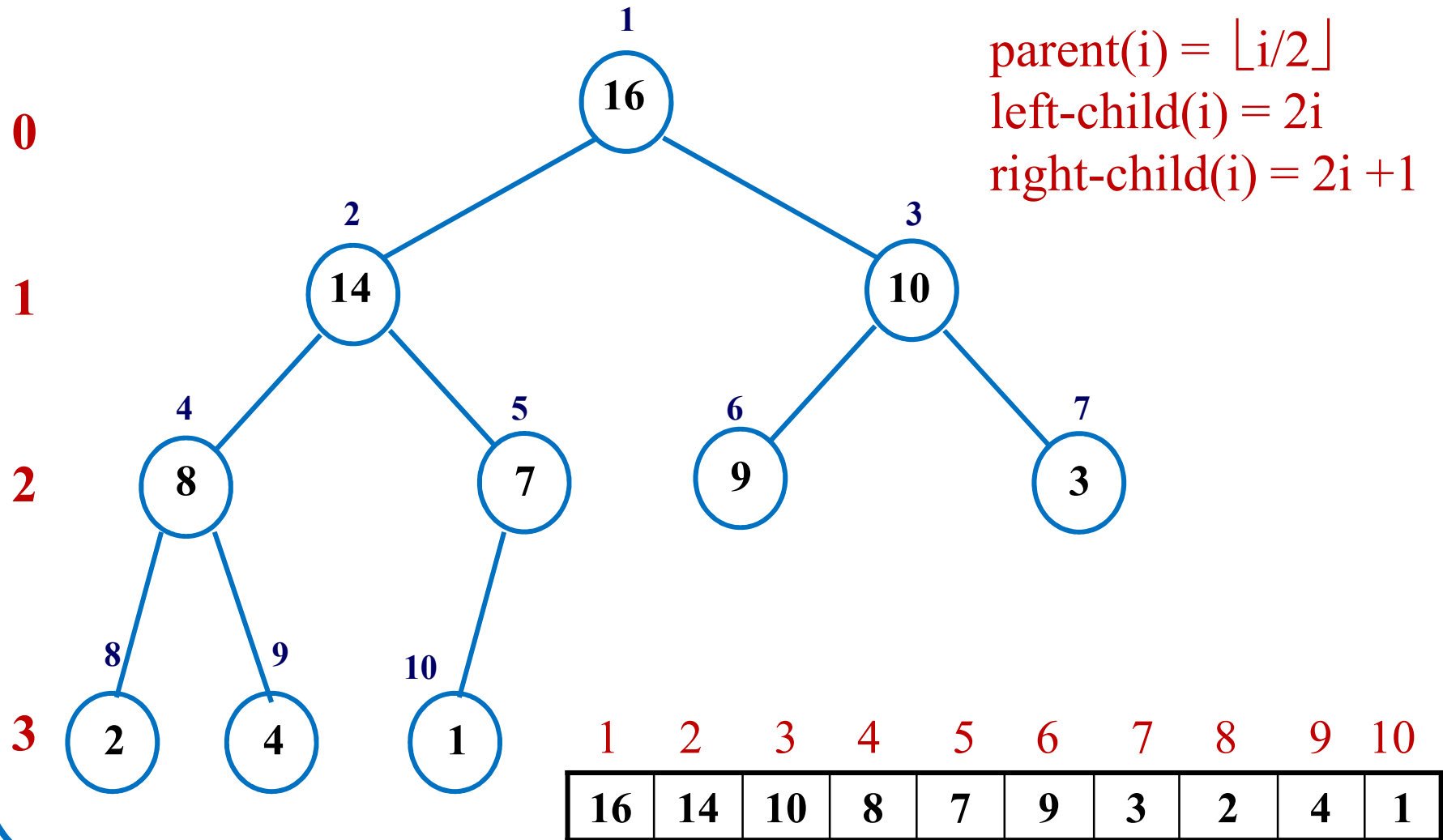
$$\text{parent}(i) = \lfloor i/2 \rfloor$$

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

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



Ví dụ đồng



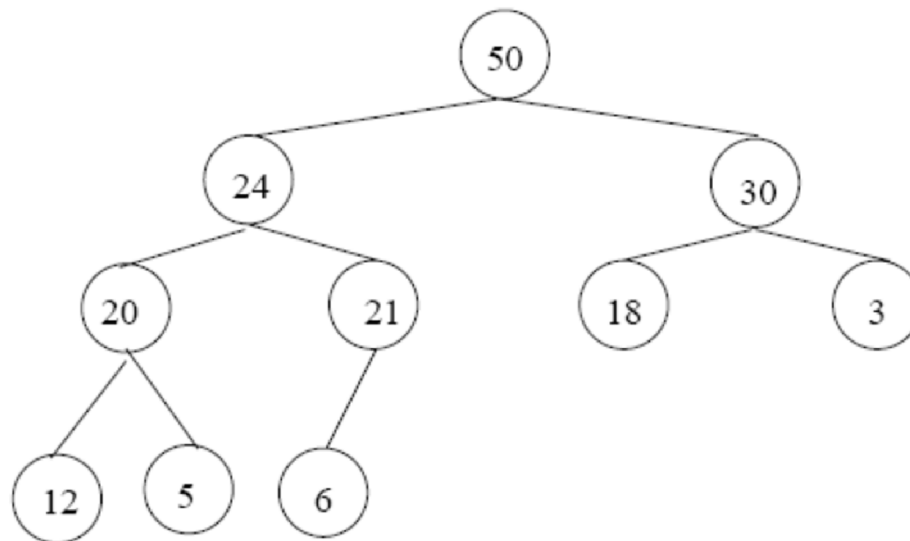
Hai dạng đống

- **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)



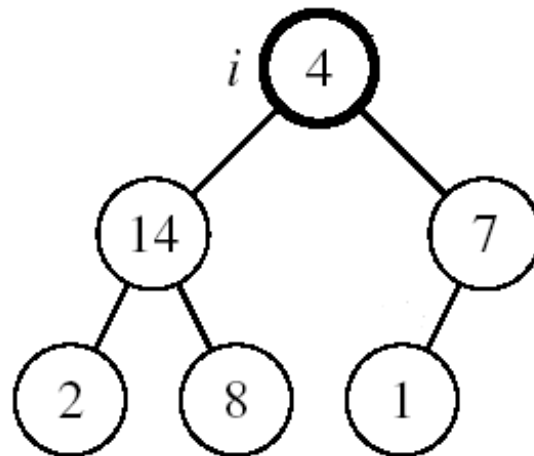
Các phép toán đối với đống

Operations on Heaps

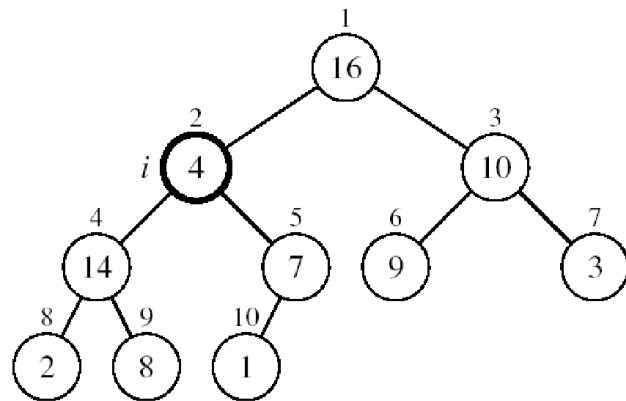
- Khôi phục tính chất max-heap (Vun lại đống)
 - Max-Heapify
- Tạo max-heap từ một mảng không được sắp xếp
 - Build-Max-Heap

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

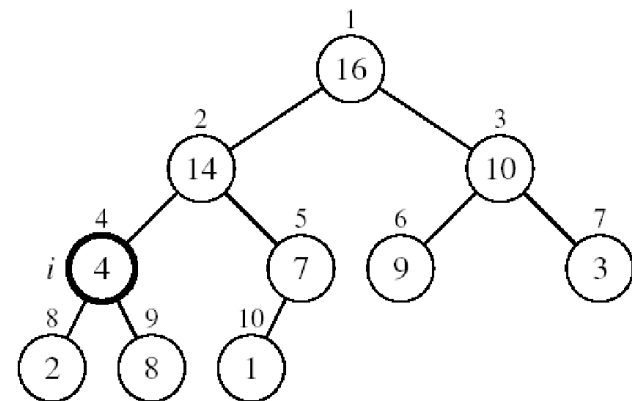


Ví dụ



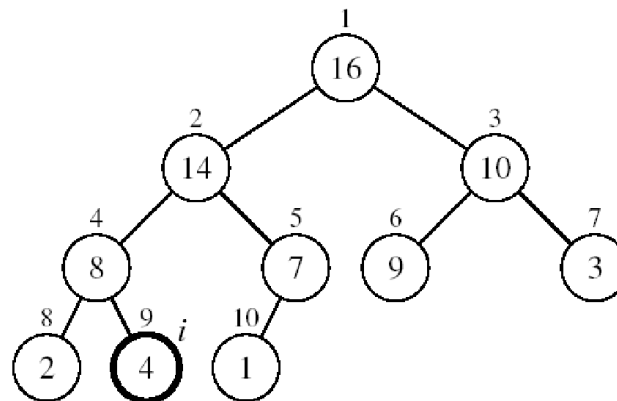
A[2] vi phạm tính chất đồng

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



A[4] vi phạm

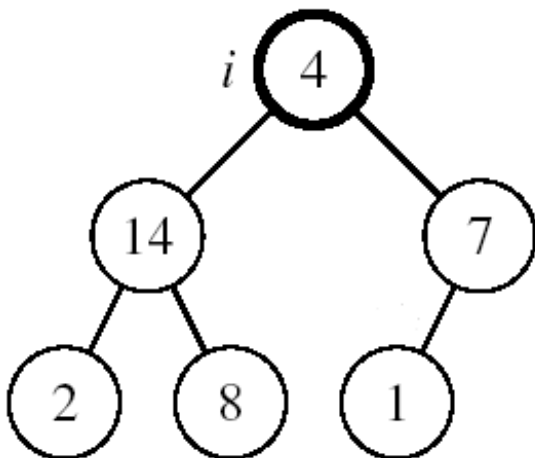
$A[4] \leftrightarrow A[9]$



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)

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

1. $l \leftarrow \text{left-child}(i)$
2. $r \leftarrow \text{right-child}(i)$
3. **if** $(l \leq n)$ and $(A[l] > A[i])$
4. **then** $\text{largest} \leftarrow l$
5. **else** $\text{largest} \leftarrow i$
6. **if** $(r \leq n)$ and $(A[r] > A[\text{largest}])$
7. **then** $\text{largest} \leftarrow r$
8. **if** $\text{largest} \neq i$
9. **then** $\text{Exchange}(A[i], A[\text{largest}])$
10. $\text{Max-Heapify}(A, \text{largest}, n)$

Thời gian tính của MAX-HEAPIFY

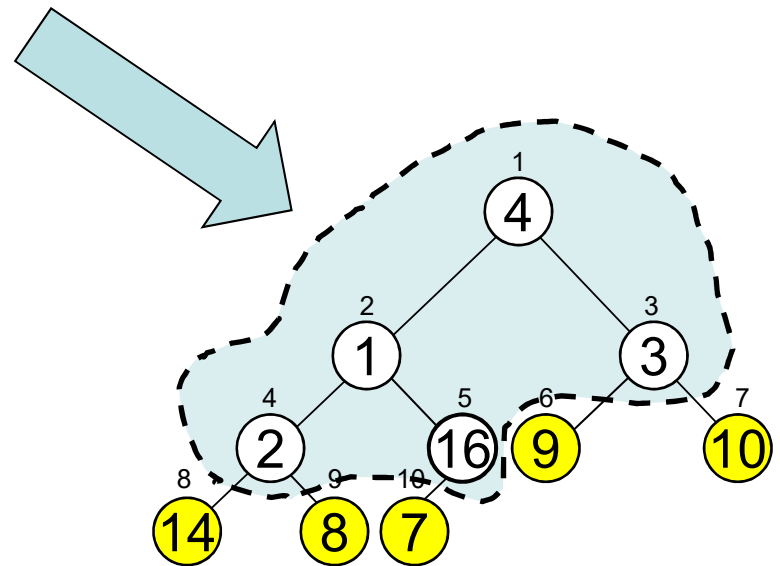
- Ta nhận thấy rằng:
 - Từ nút i phải di chuyển theo đường đi xuống phía dưới của cây. Độ dài của đường đi này không vượt quá độ dài đường đi từ gốc đến lá, nghĩa là không vượt quá h .
 - Ở mỗi mức phải thực hiện 2 phép so sánh.
 - Do đó tổng số phép so sánh không vượt quá $2h$.
 - Vậy, thời gian tính 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 $A[(\lfloor n/2 \rfloor + 1) .. n]$ 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-Heapify(A, i, n)



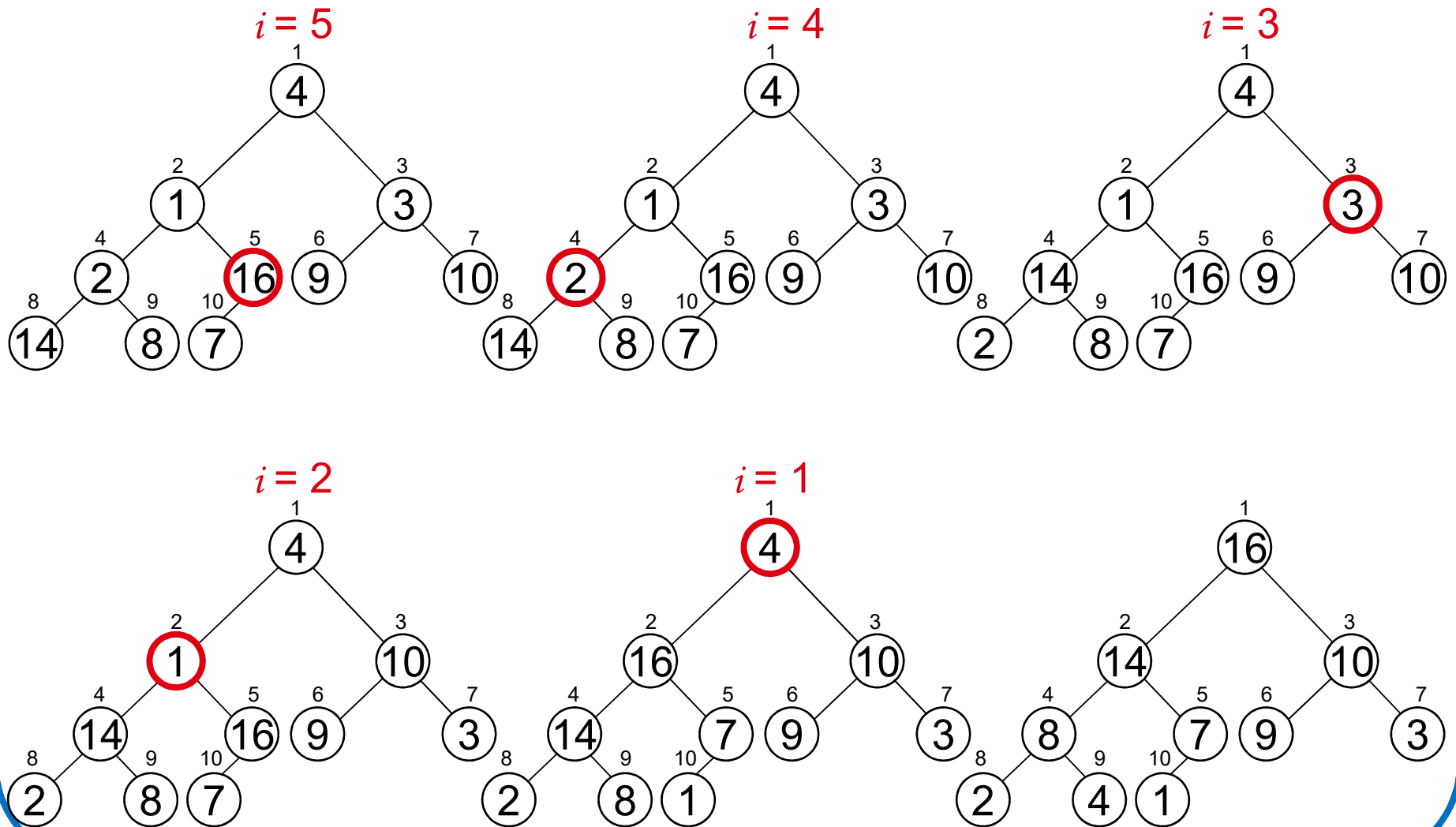
A:

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

Ví dụ:

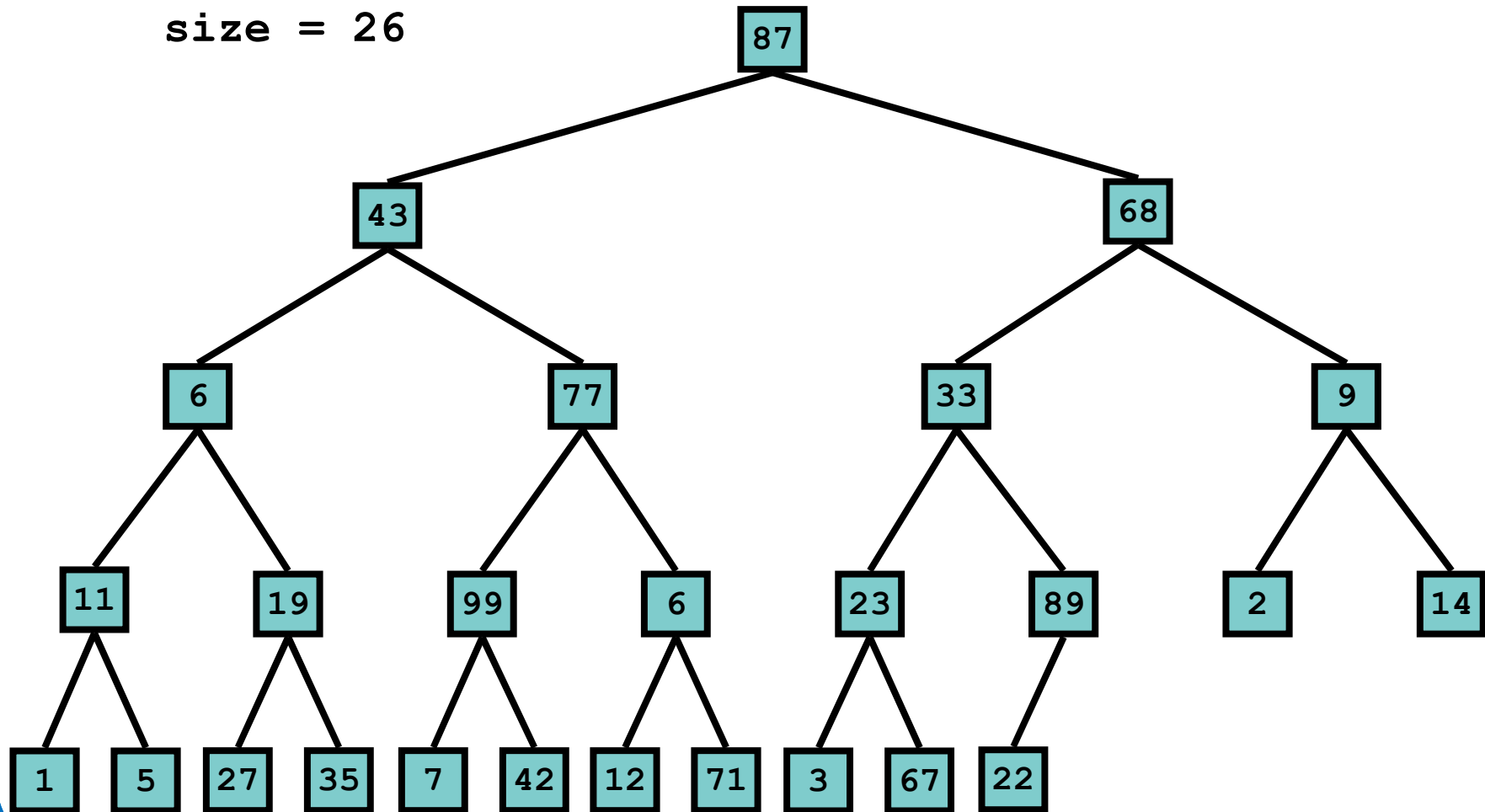
A:

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



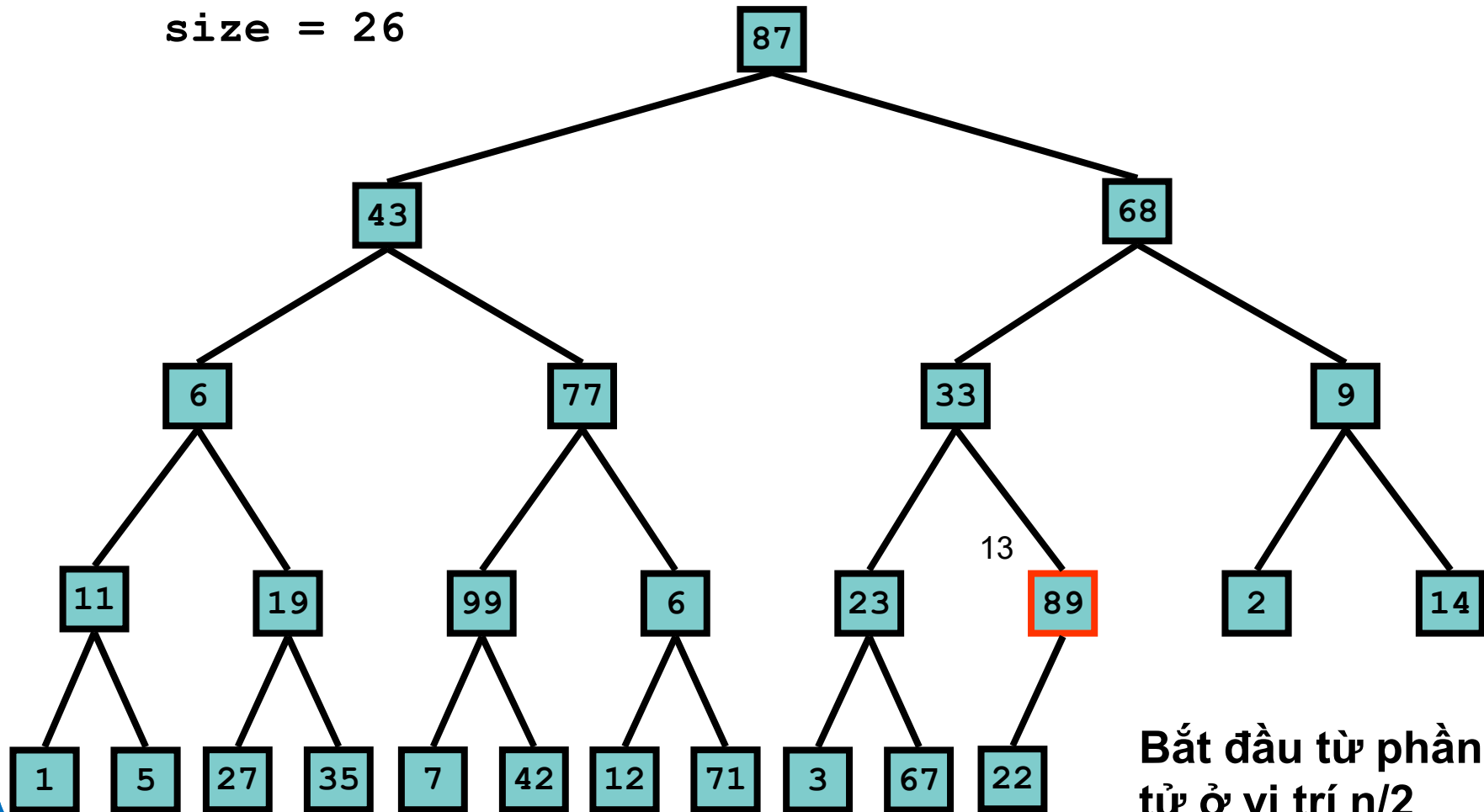
Ví dụ: Build-Min-Heap

size = 26



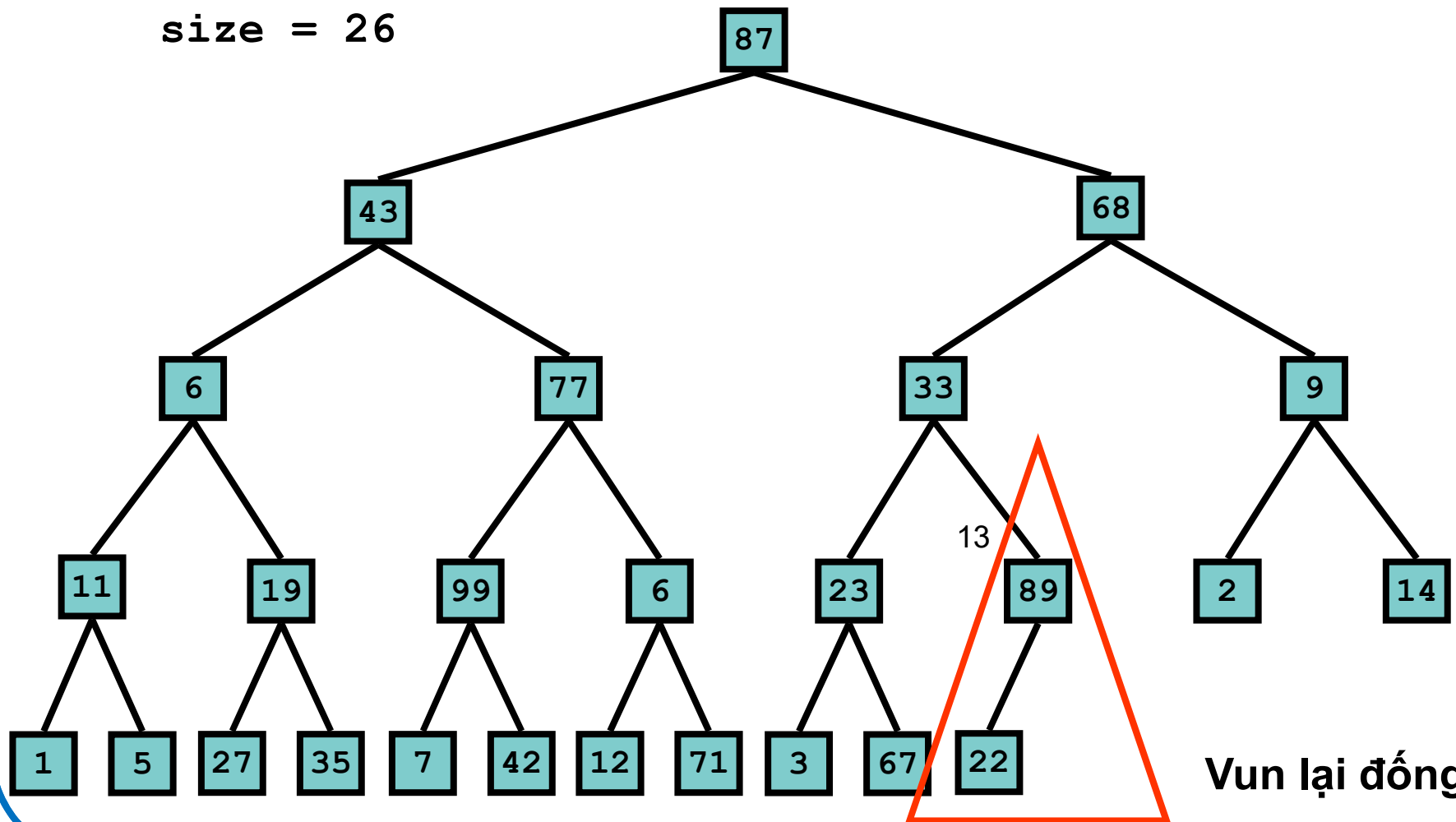
Ví dụ: Build-Min-Heap

size = 26



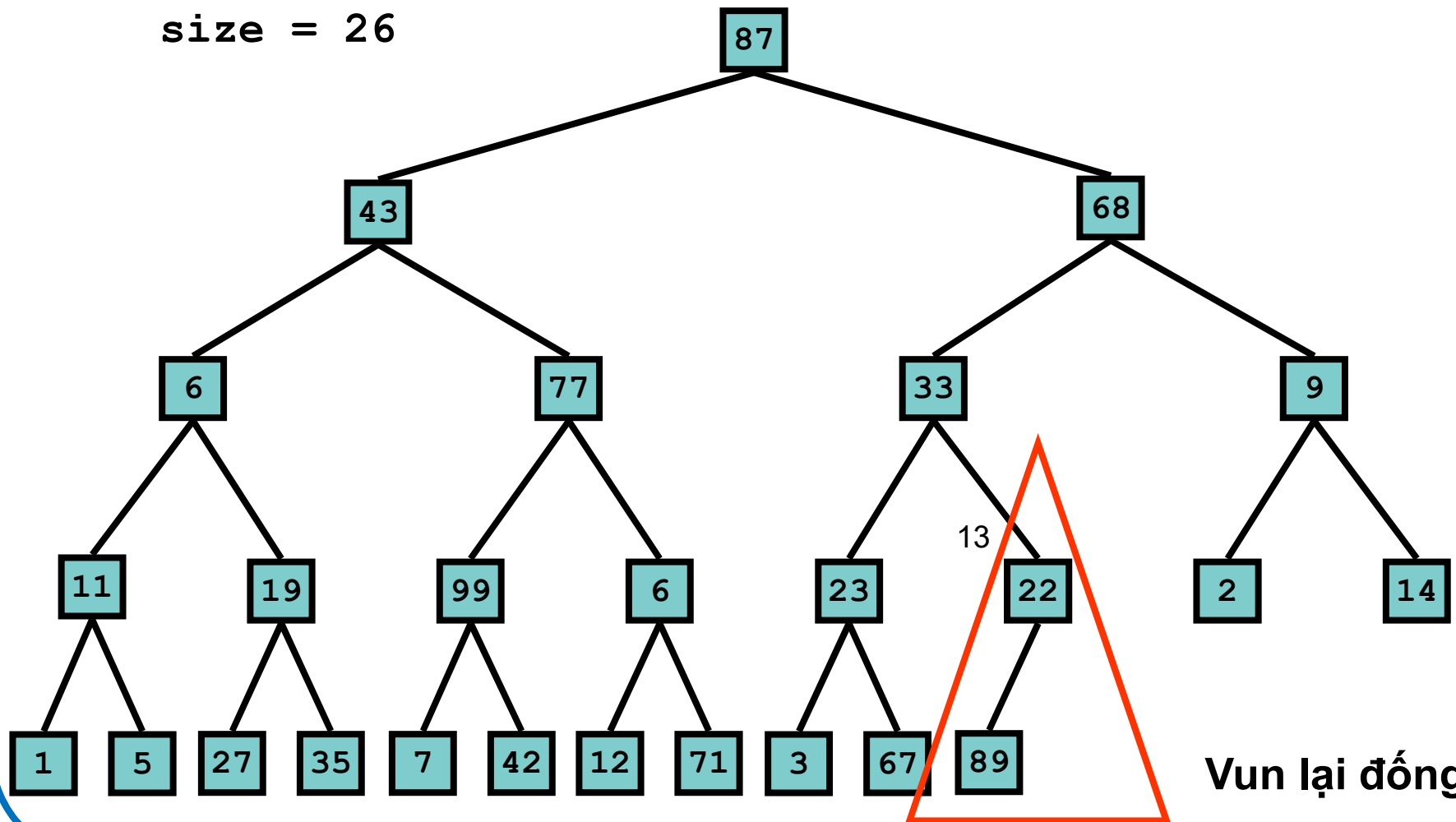
Build-Min-Heap

size = 26



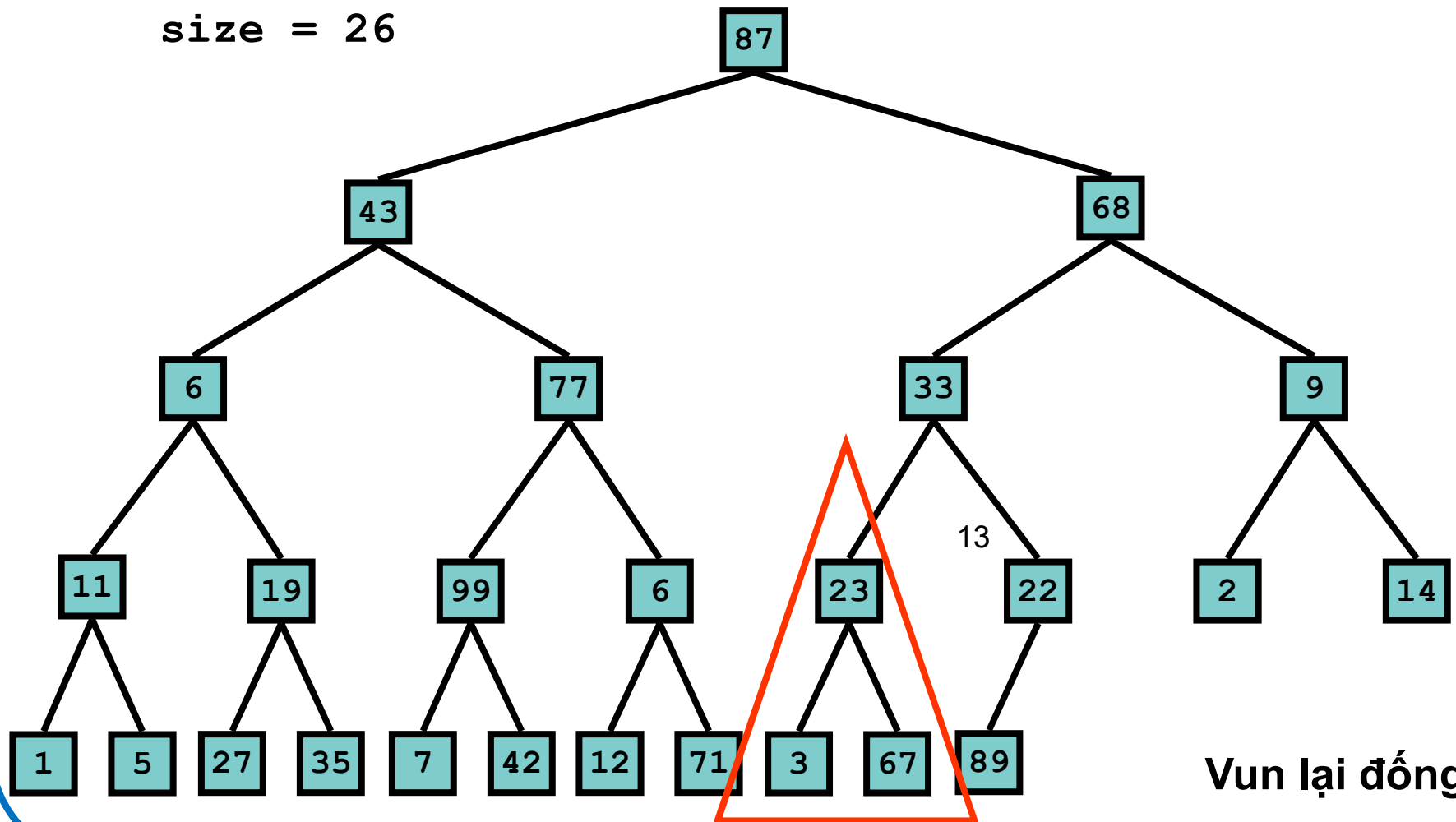
Build-Min-Heap

size = 26



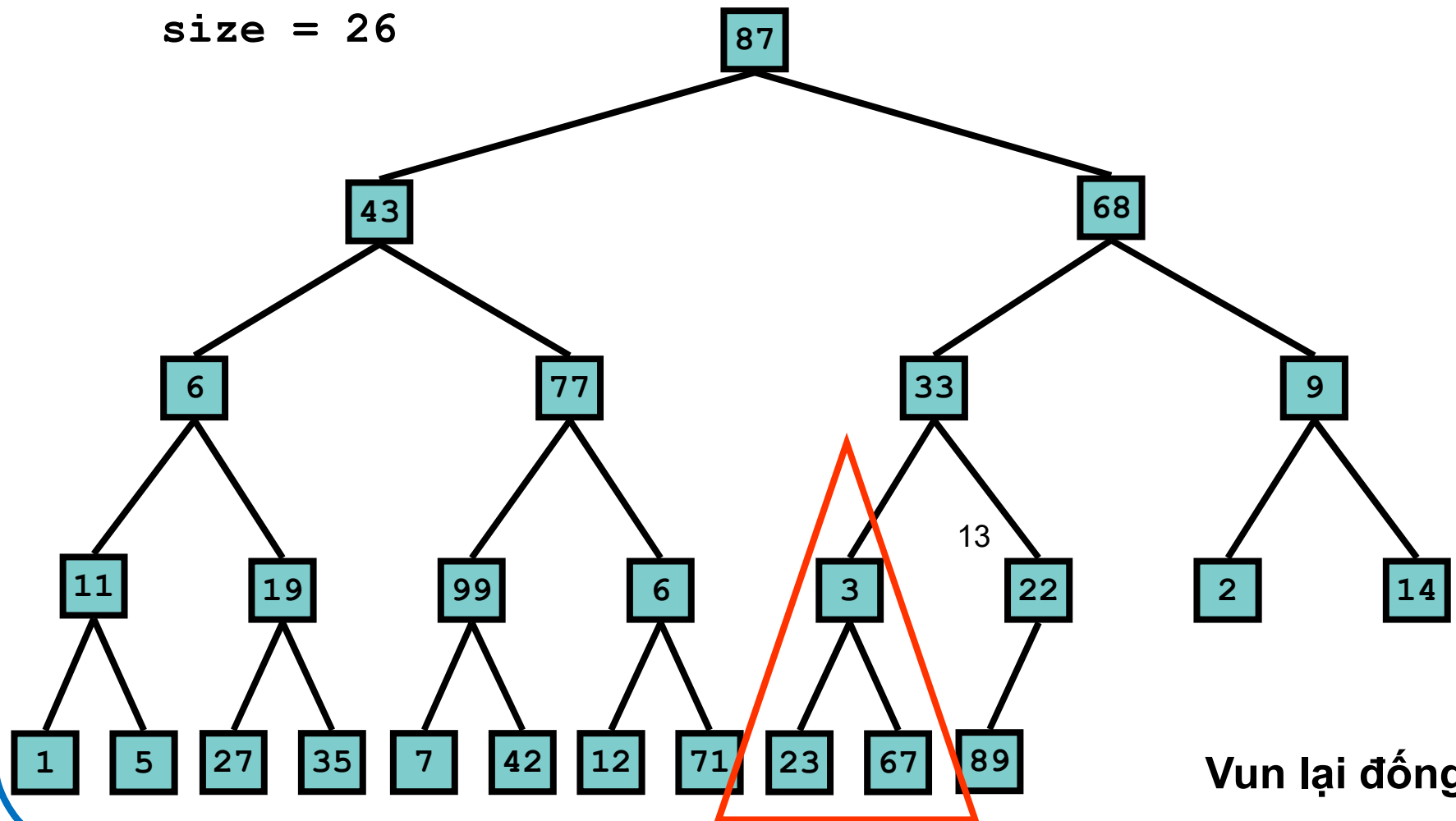
Build-Min-Heap

size = 26



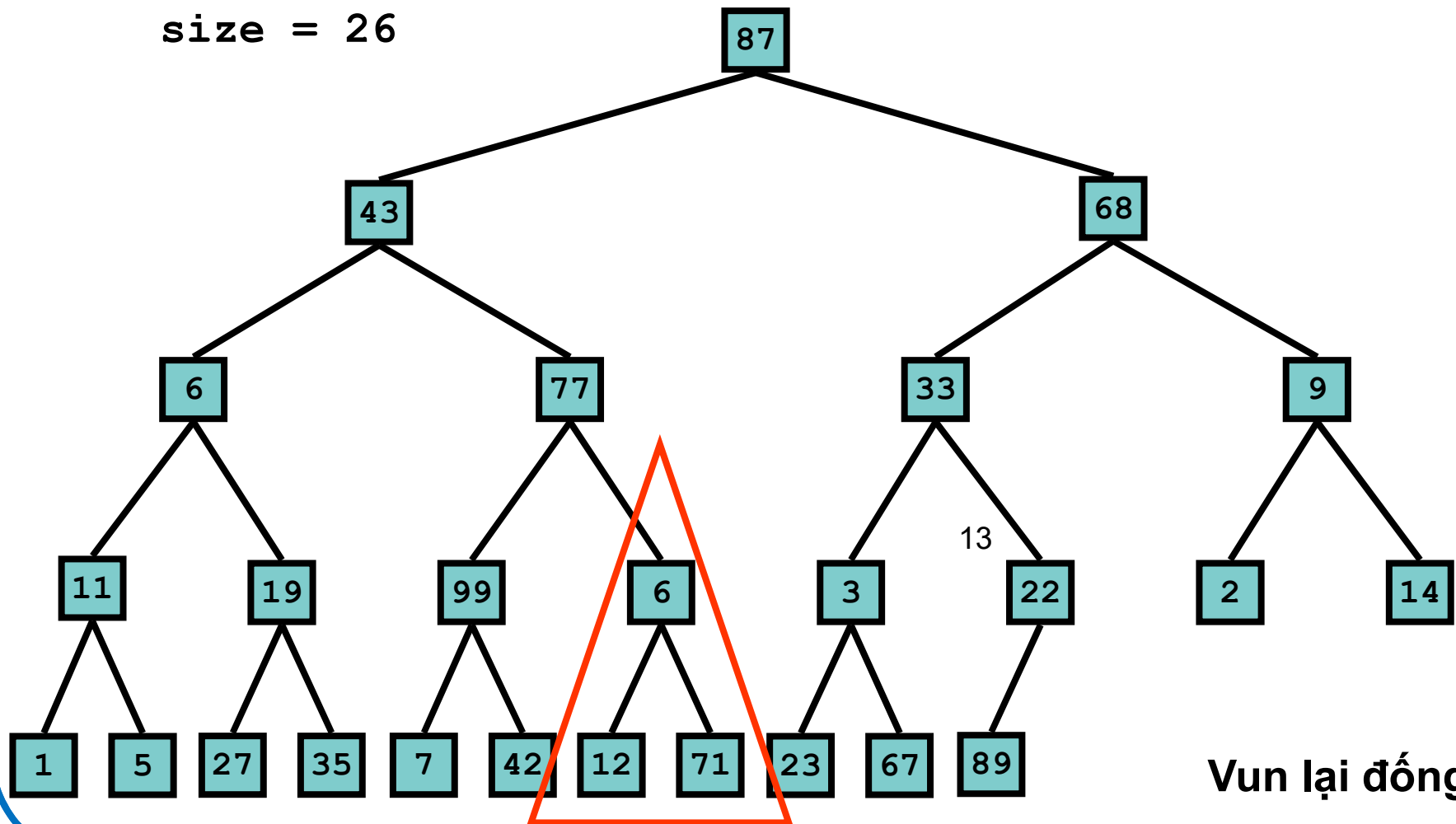
Build-Min-Heap

size = 26



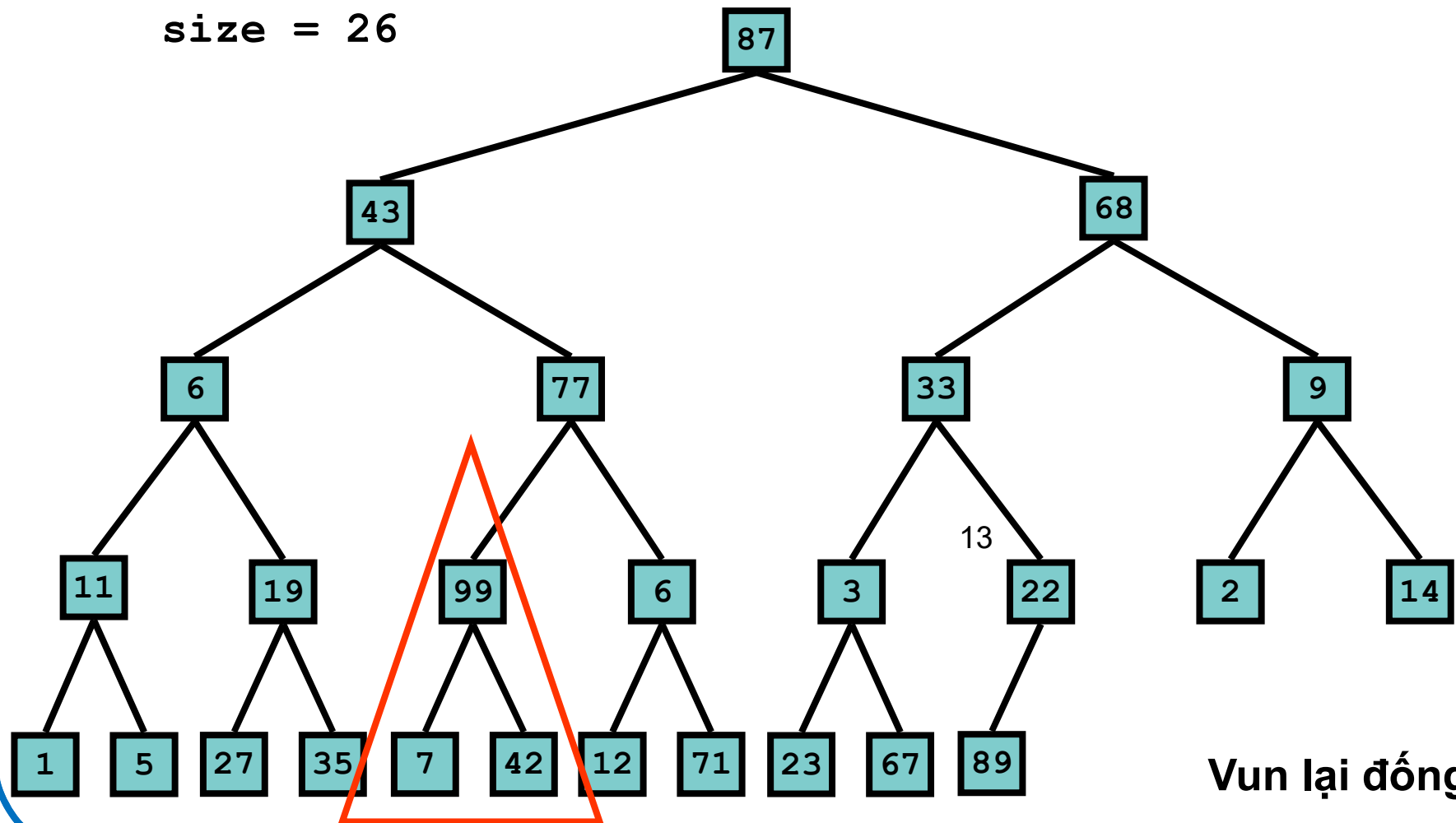
Build-Min-Heap

size = 26



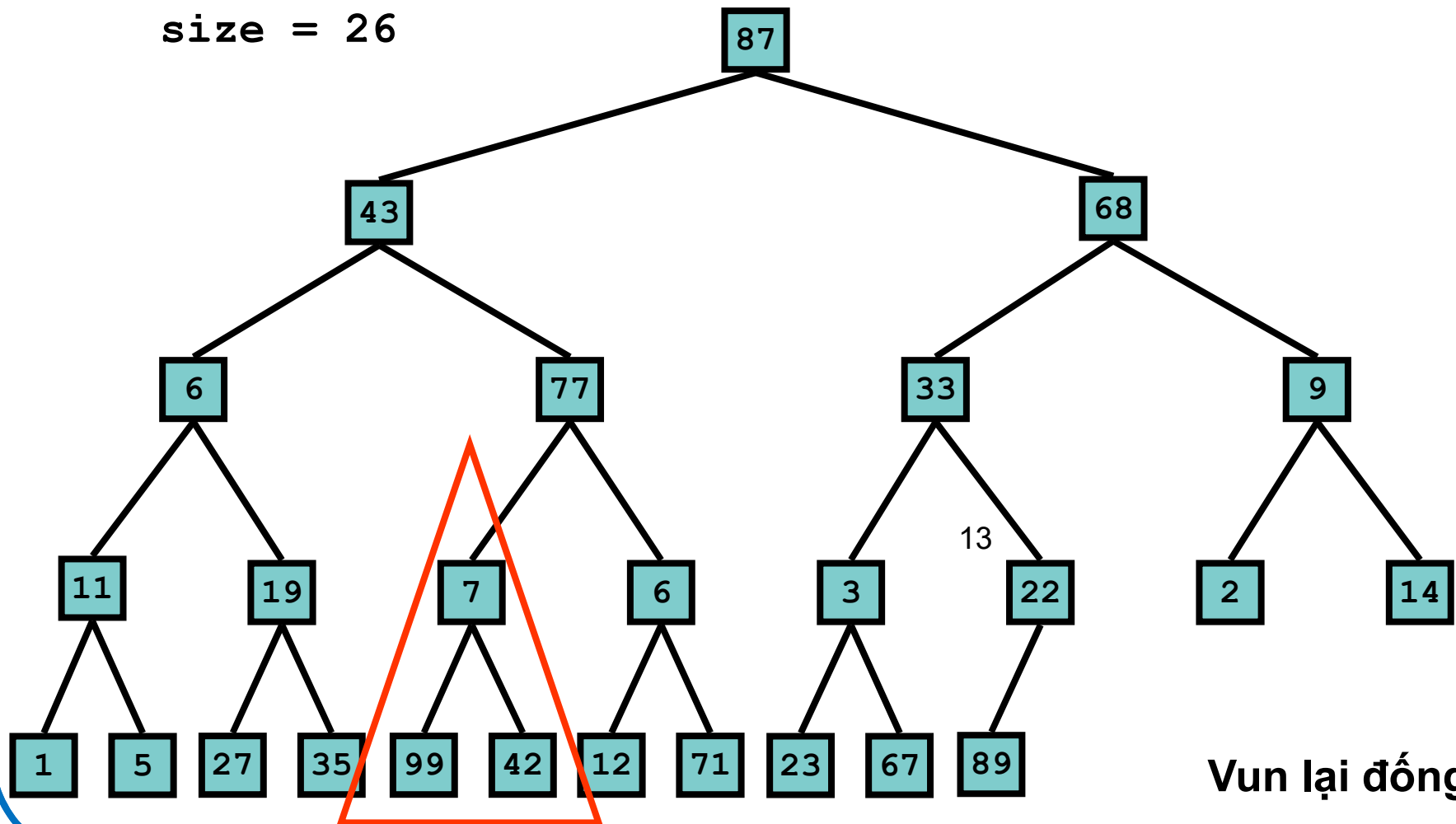
Build-Min-Heap

size = 26



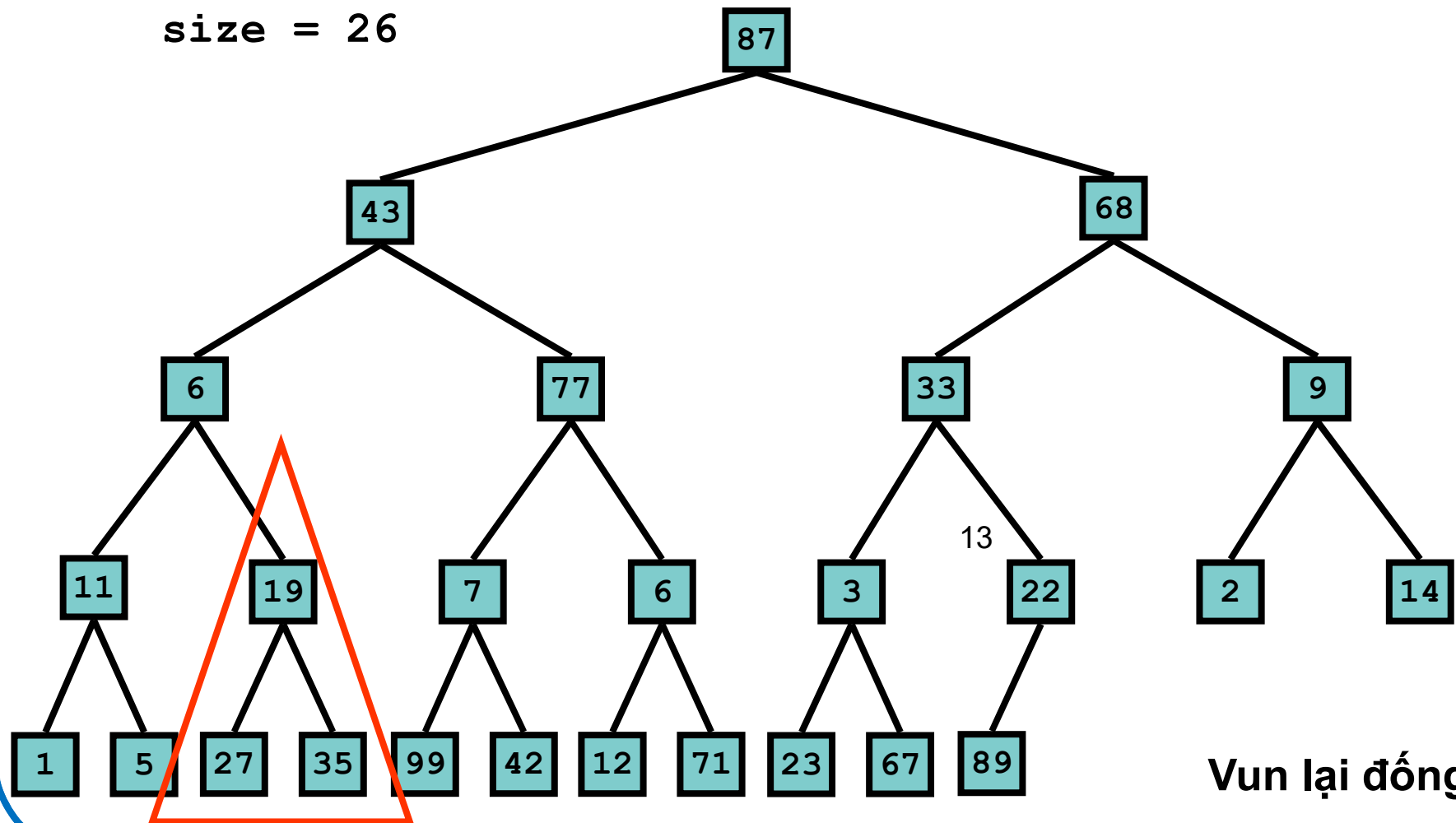
Build-Min-Heap

size = 26



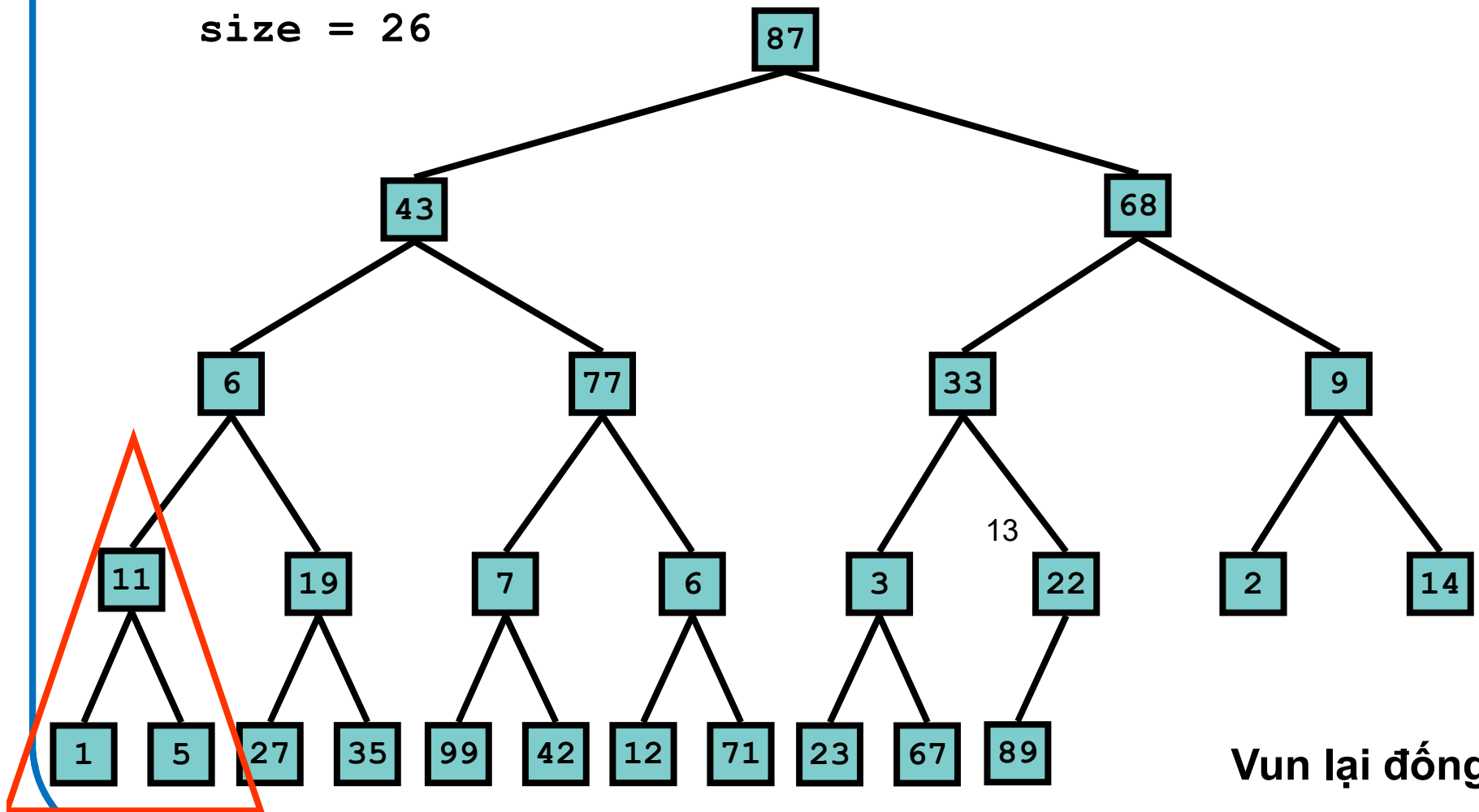
Build-Min-Heap

size = 26



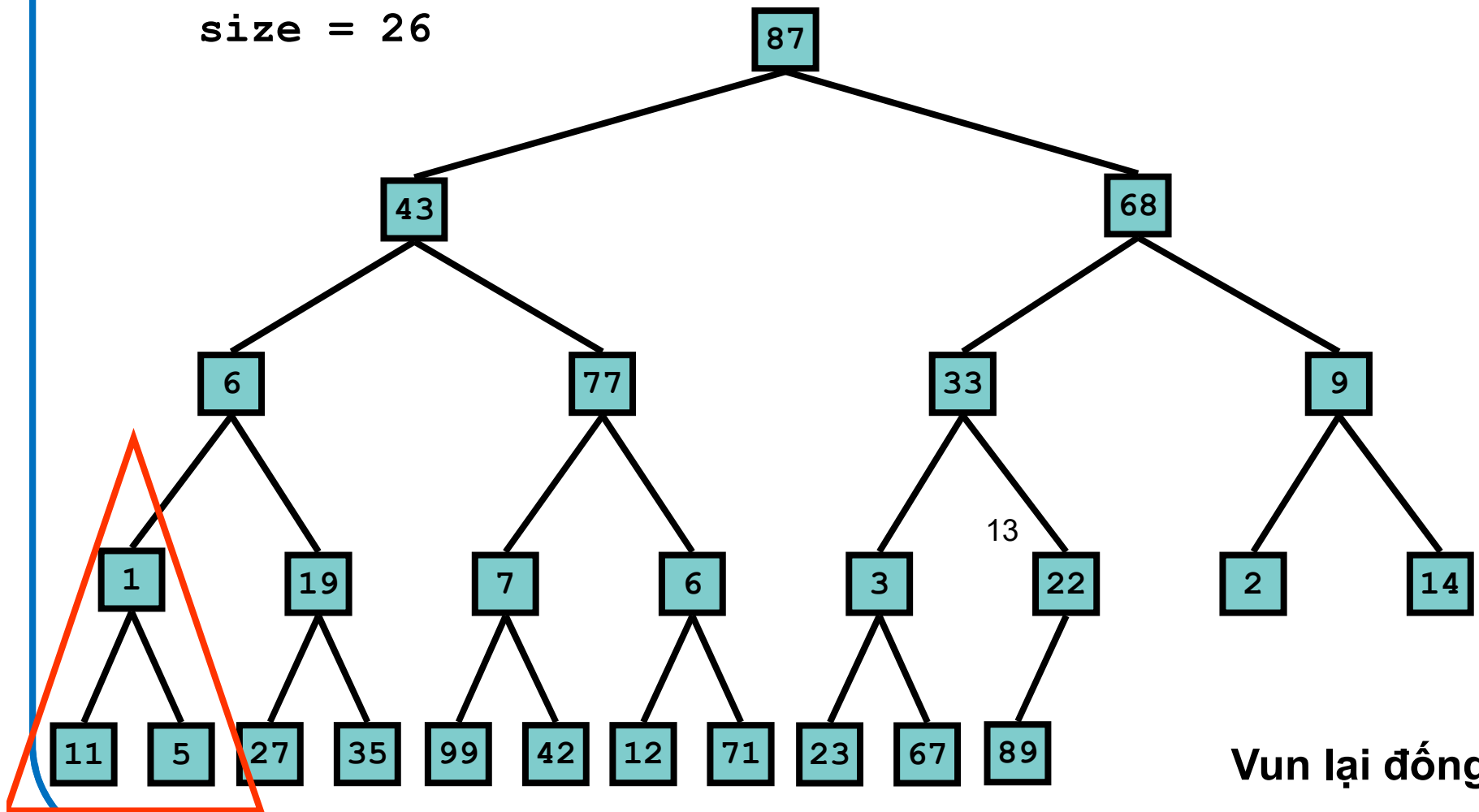
Build-Min-Heap

size = 26



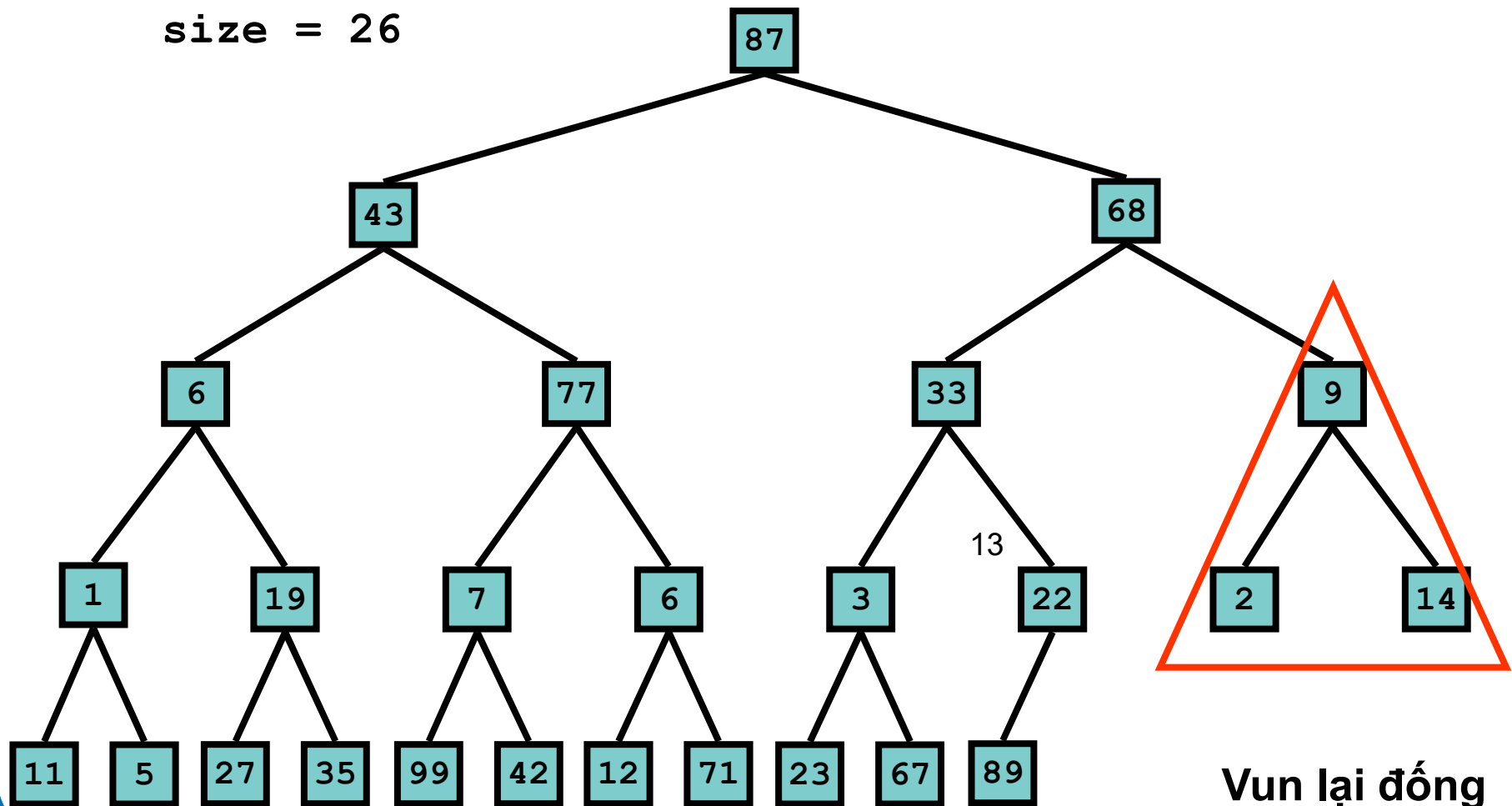
Build-Min-Heap

size = 26



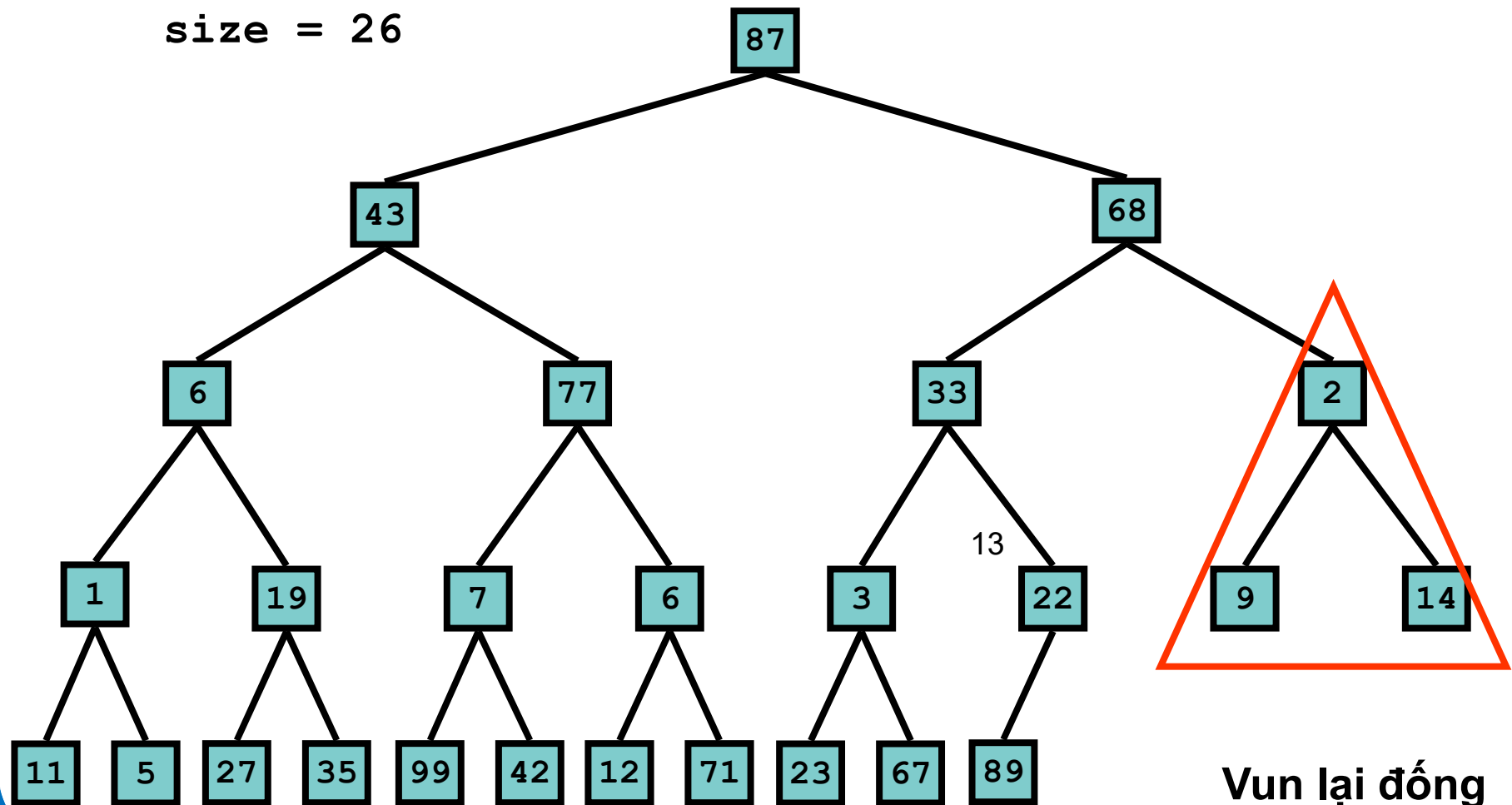
Build-Min-Heap

size = 26



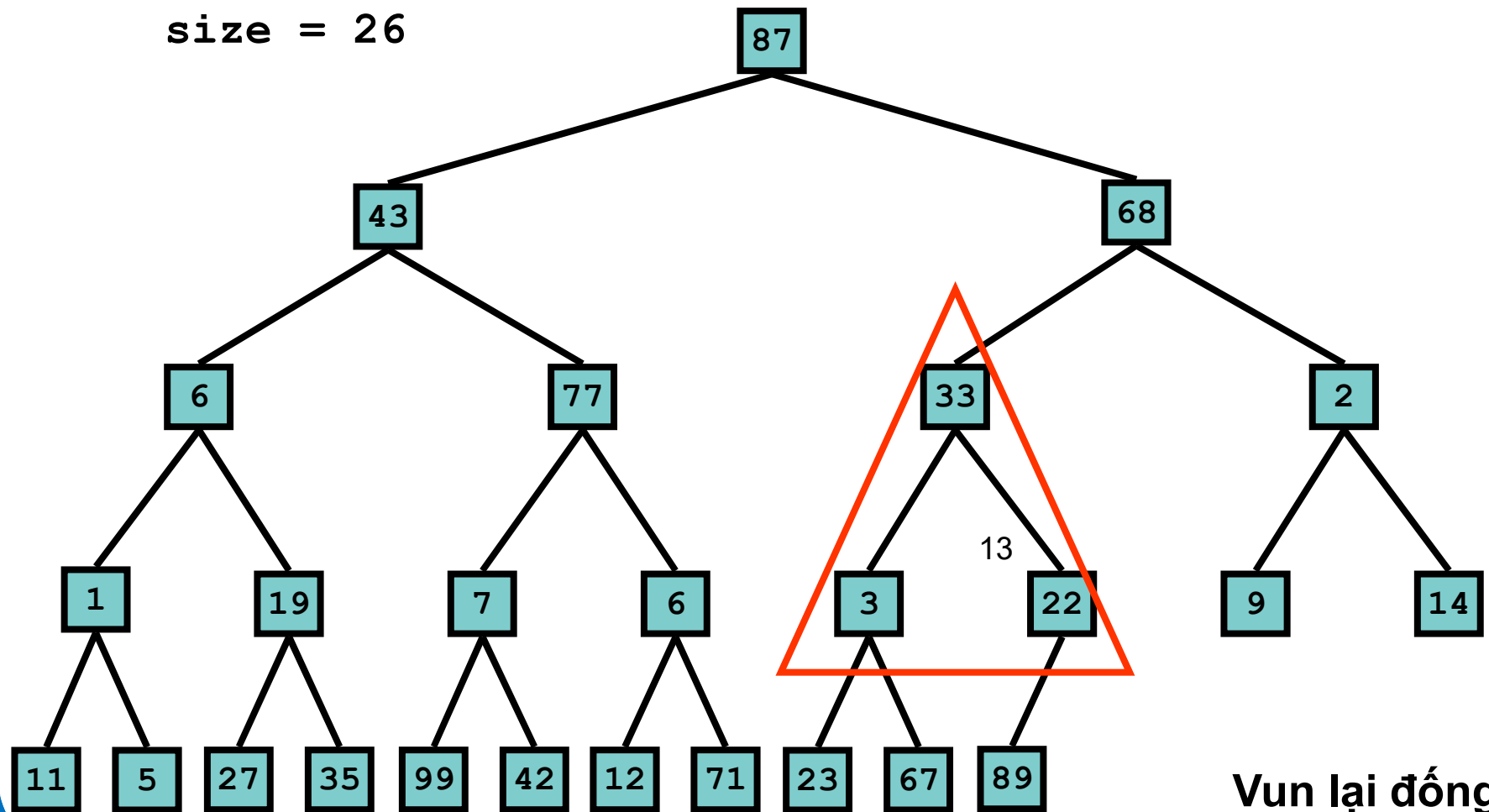
Build-Min-Heap

size = 26



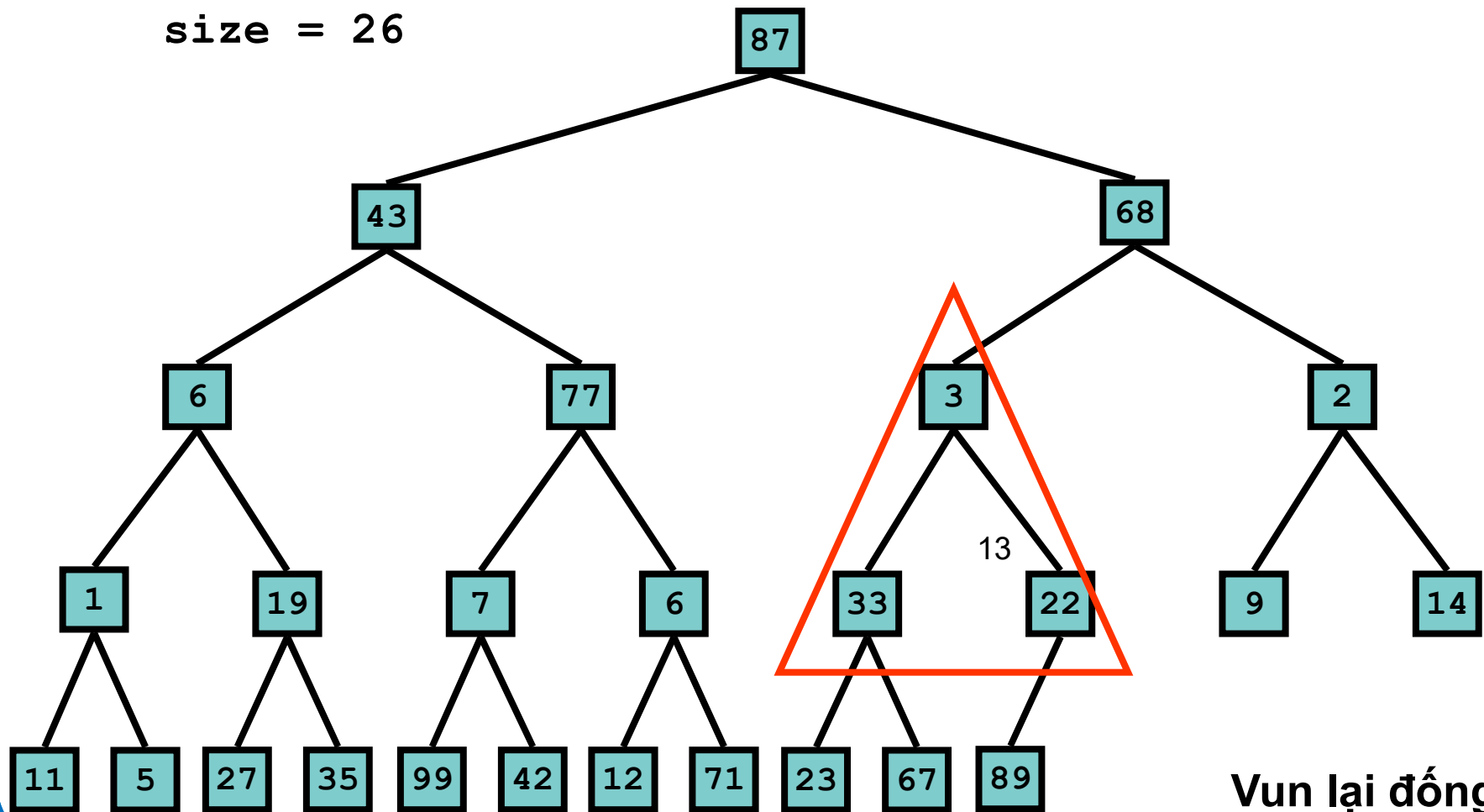
Build-Min-Heap

size = 26



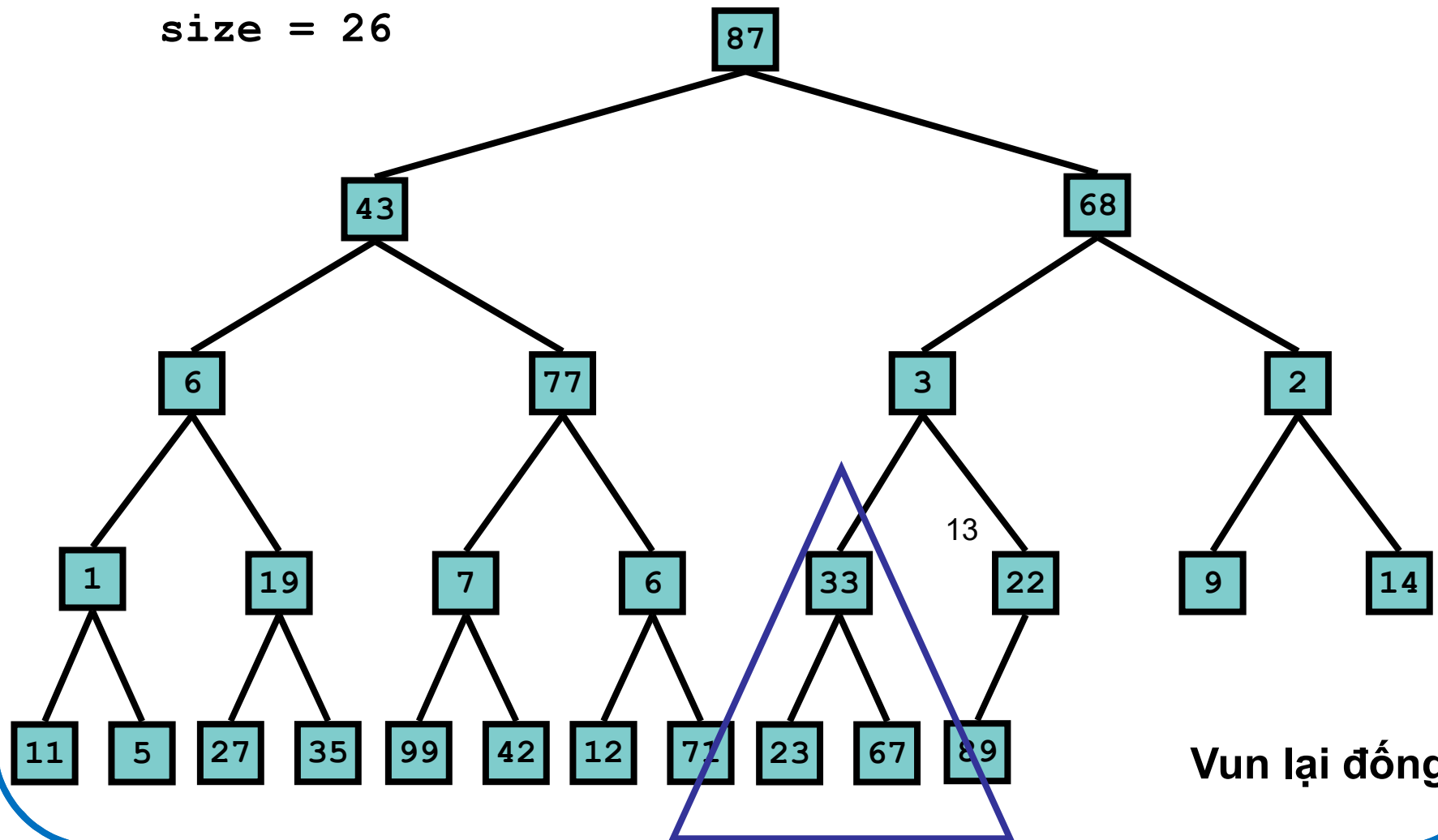
Build-Min-Heap

size = 26



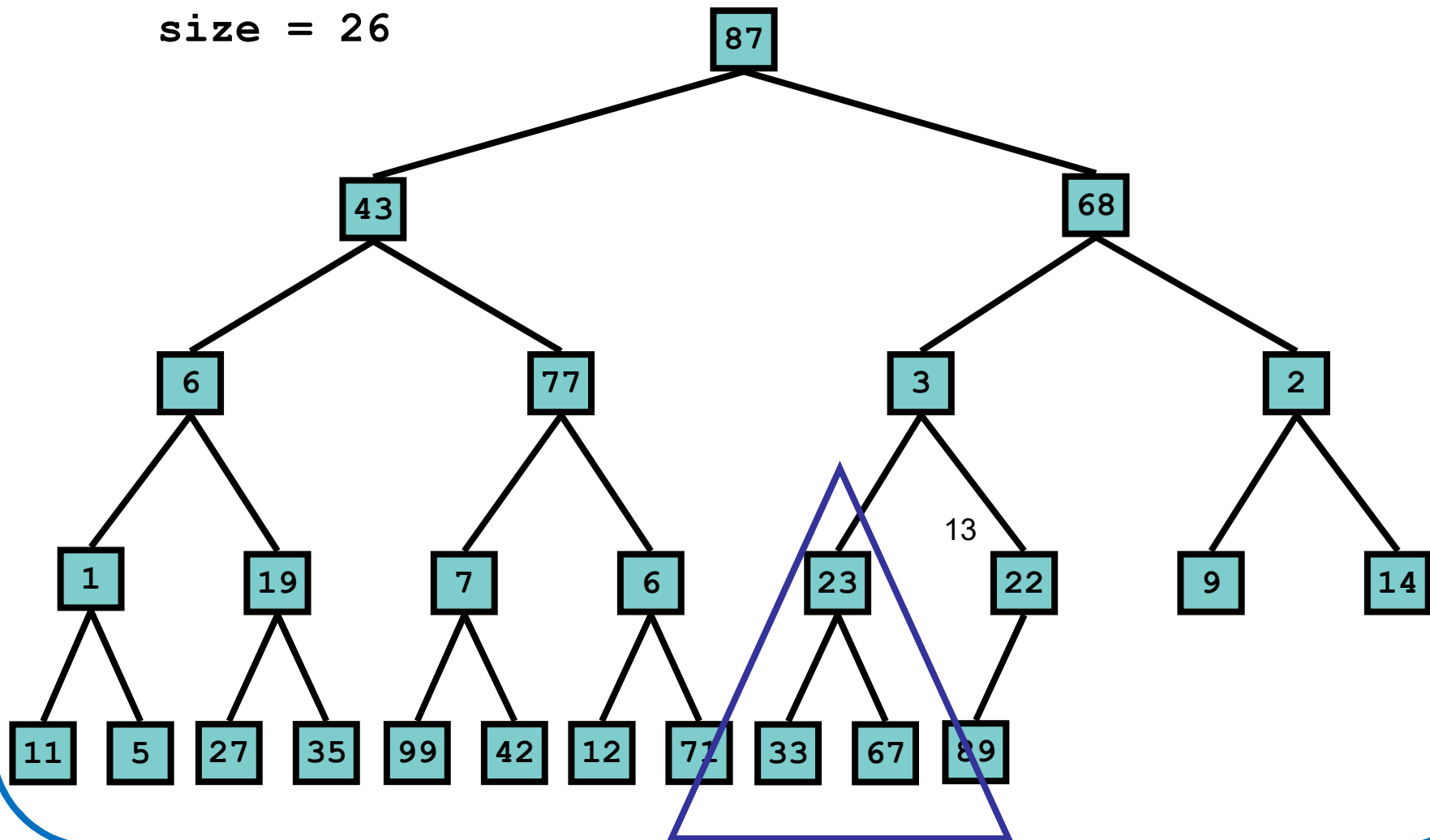
Build-Min-Heap

size = 26



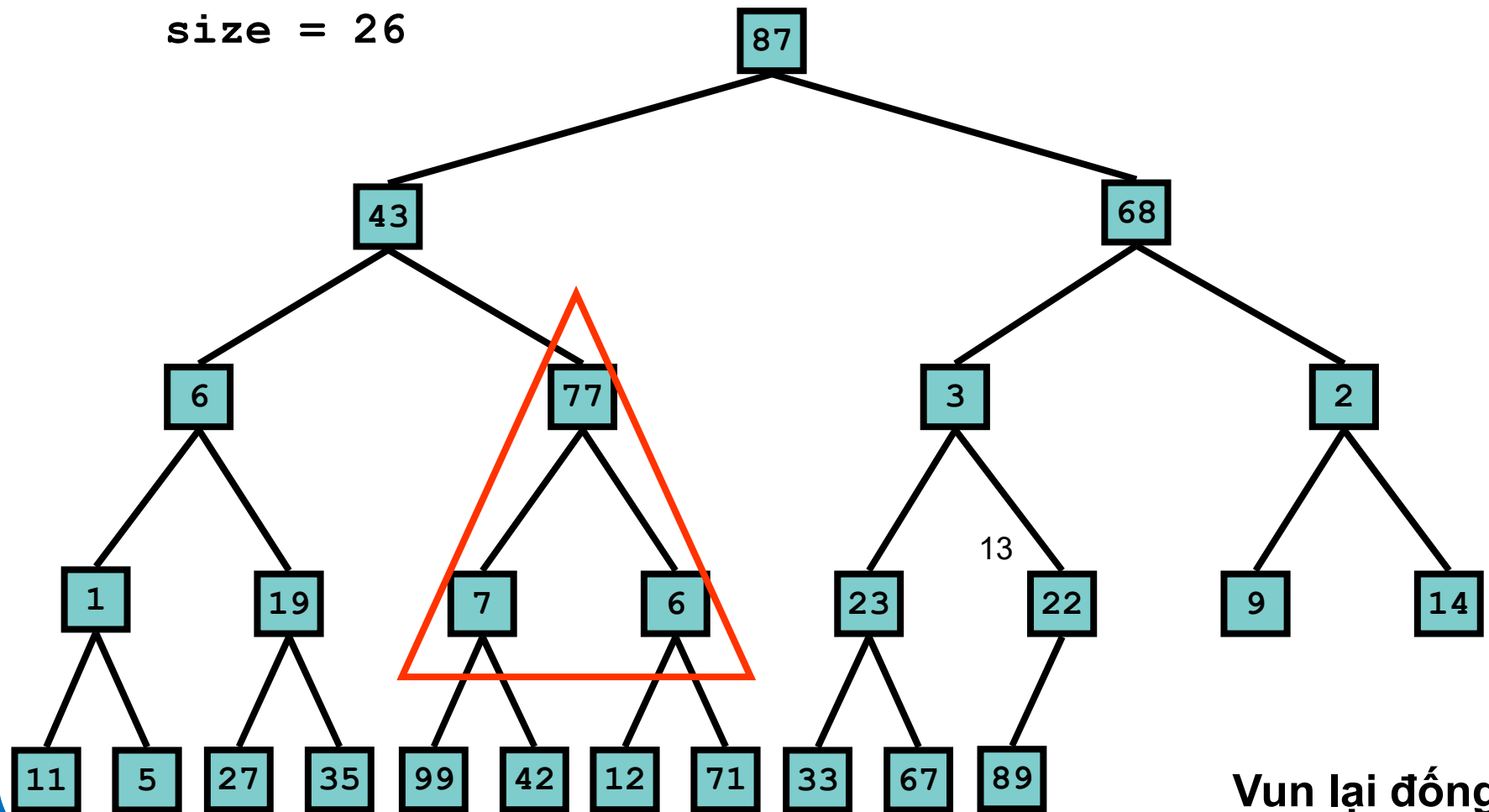
Build-Min-Heap

size = 26



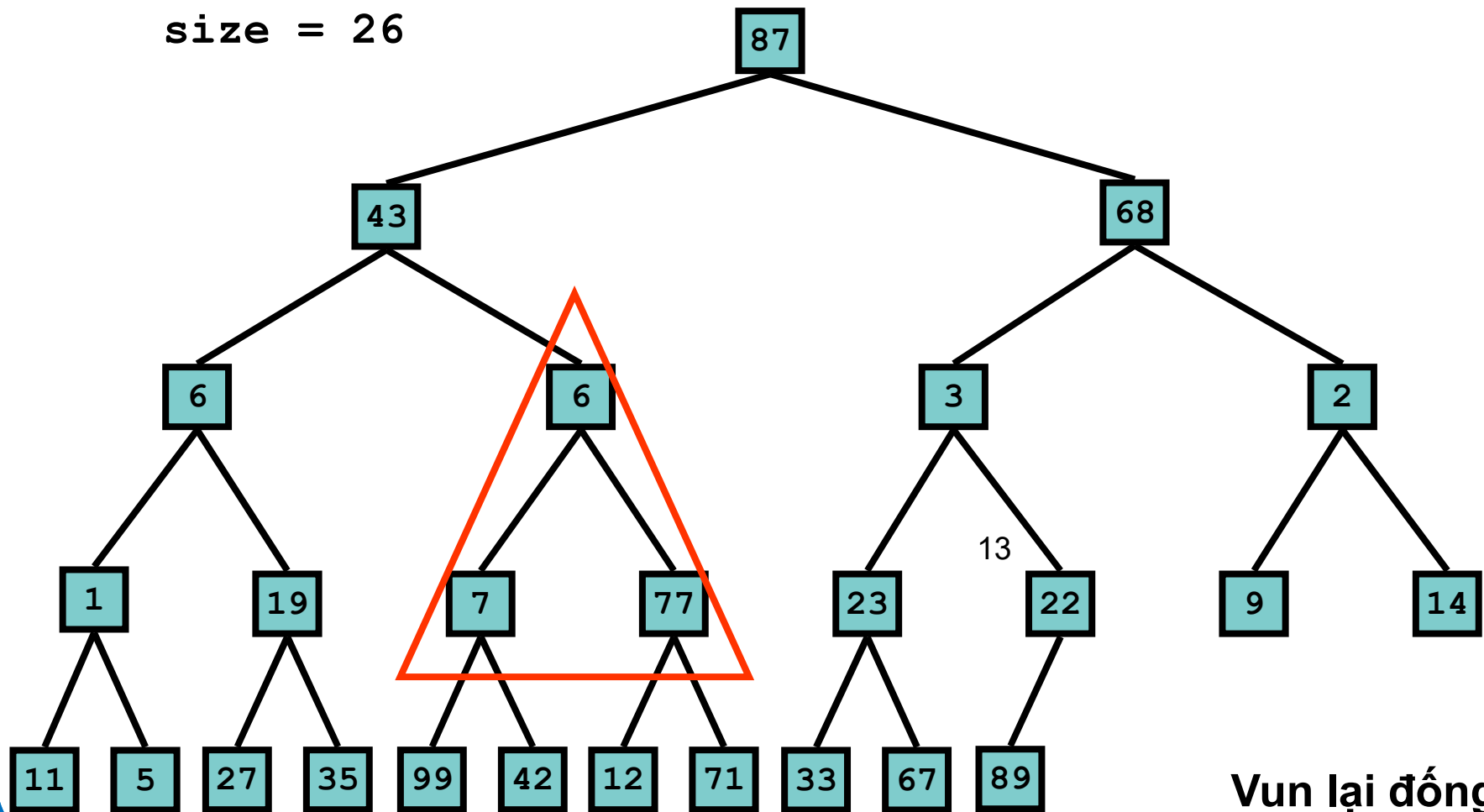
Build-Min-Heap

size = 26



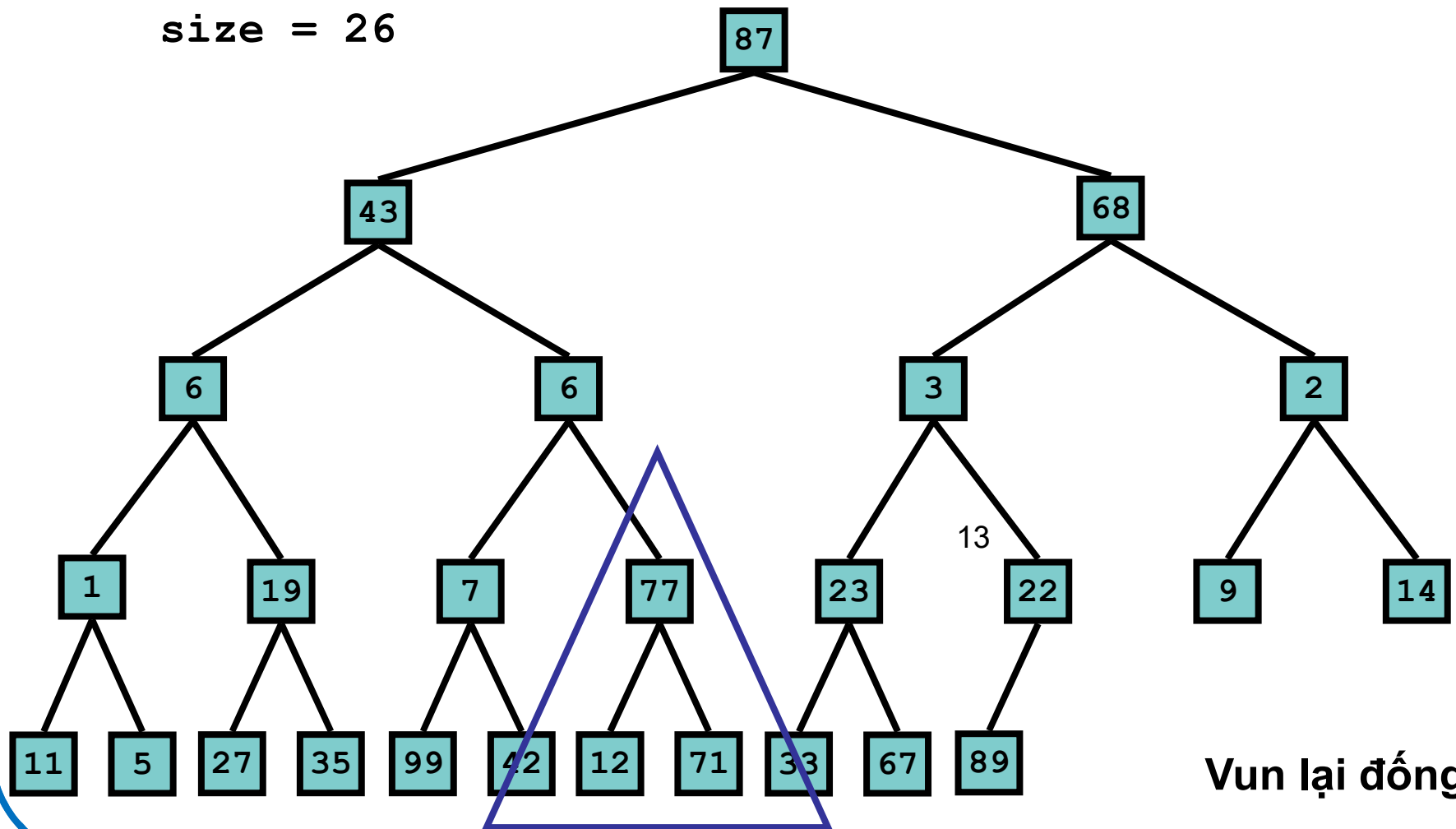
Build-Min-Heap

size = 26



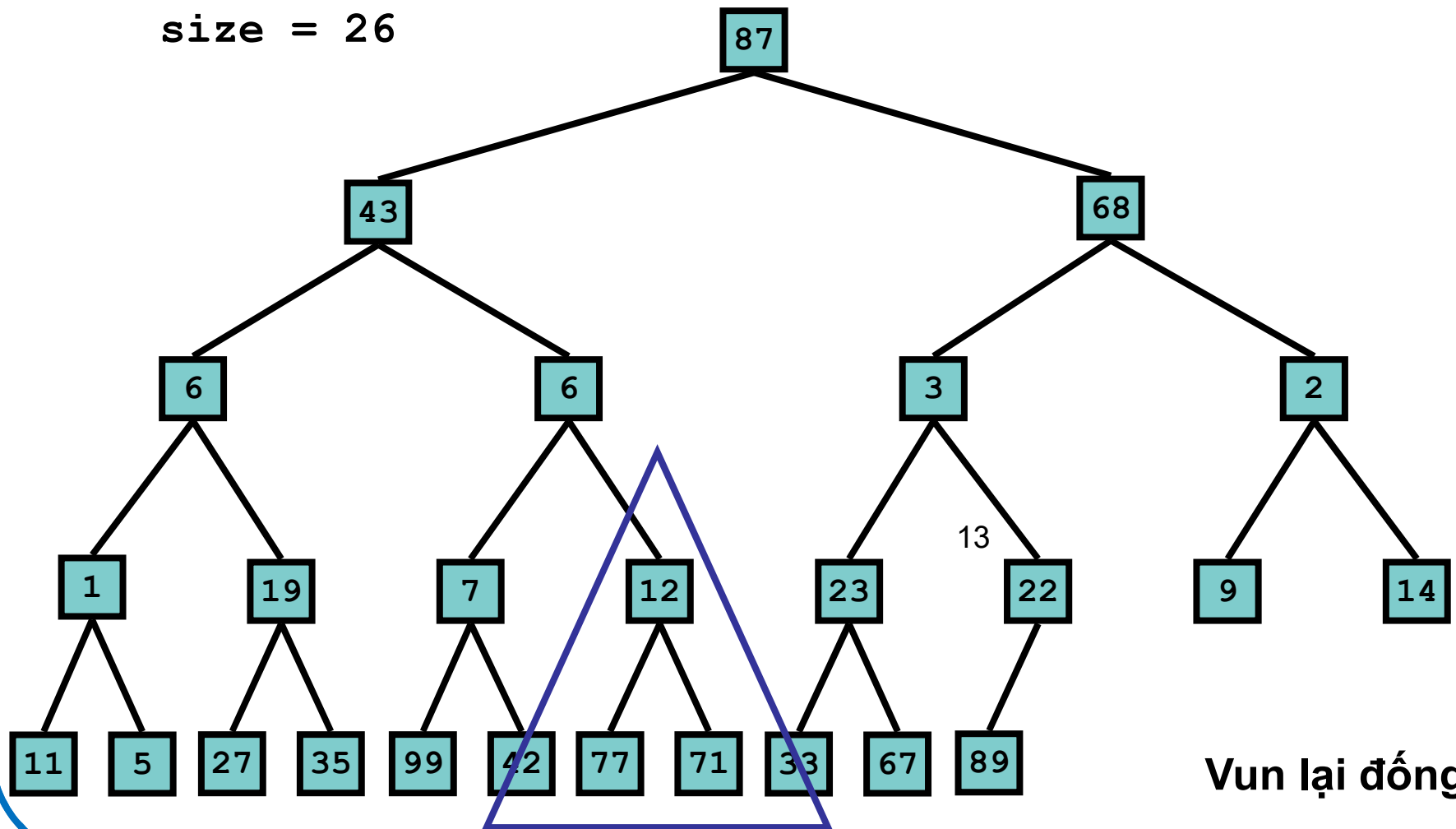
Build-Min-Heap

size = 26



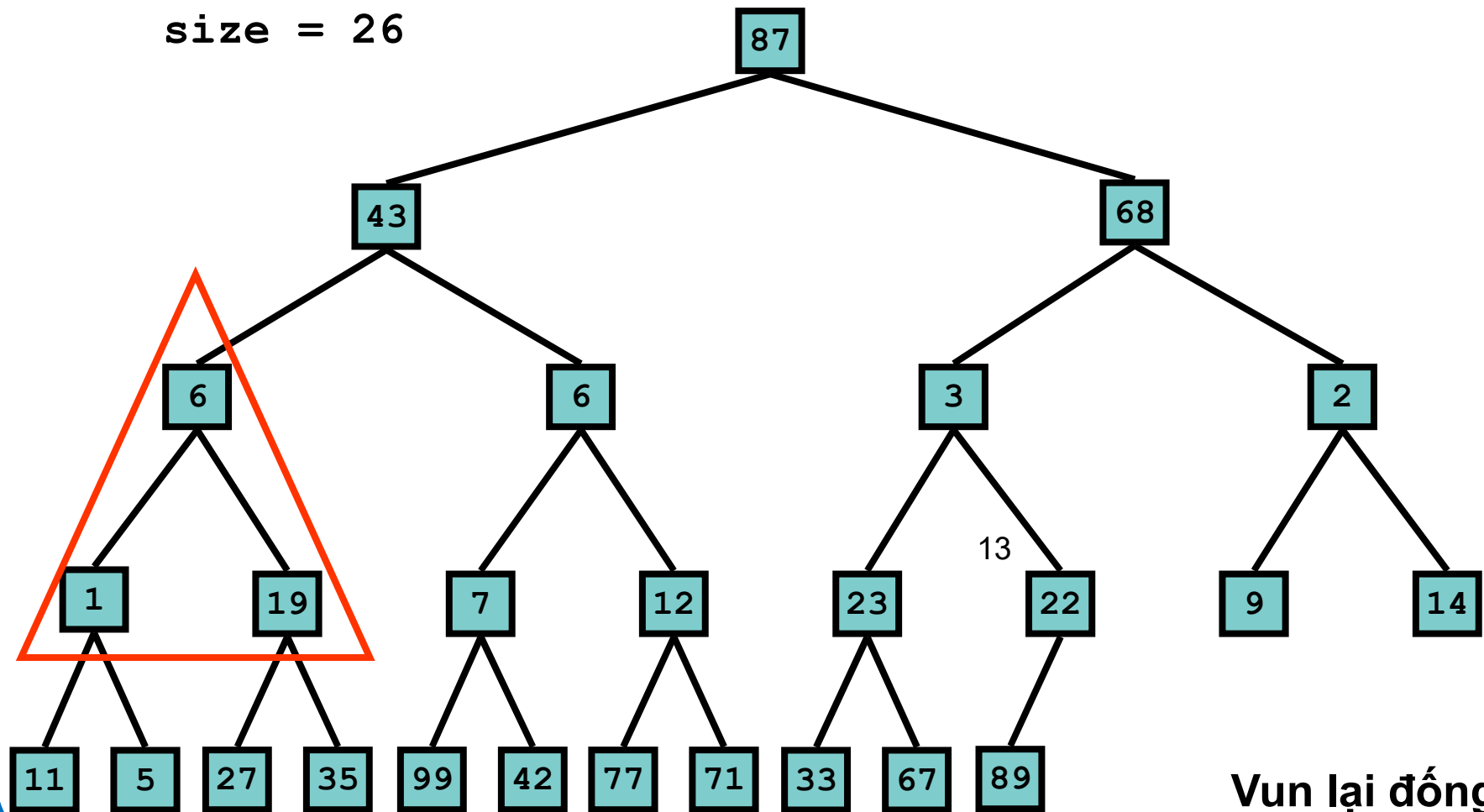
Build-Min-Heap

size = 26



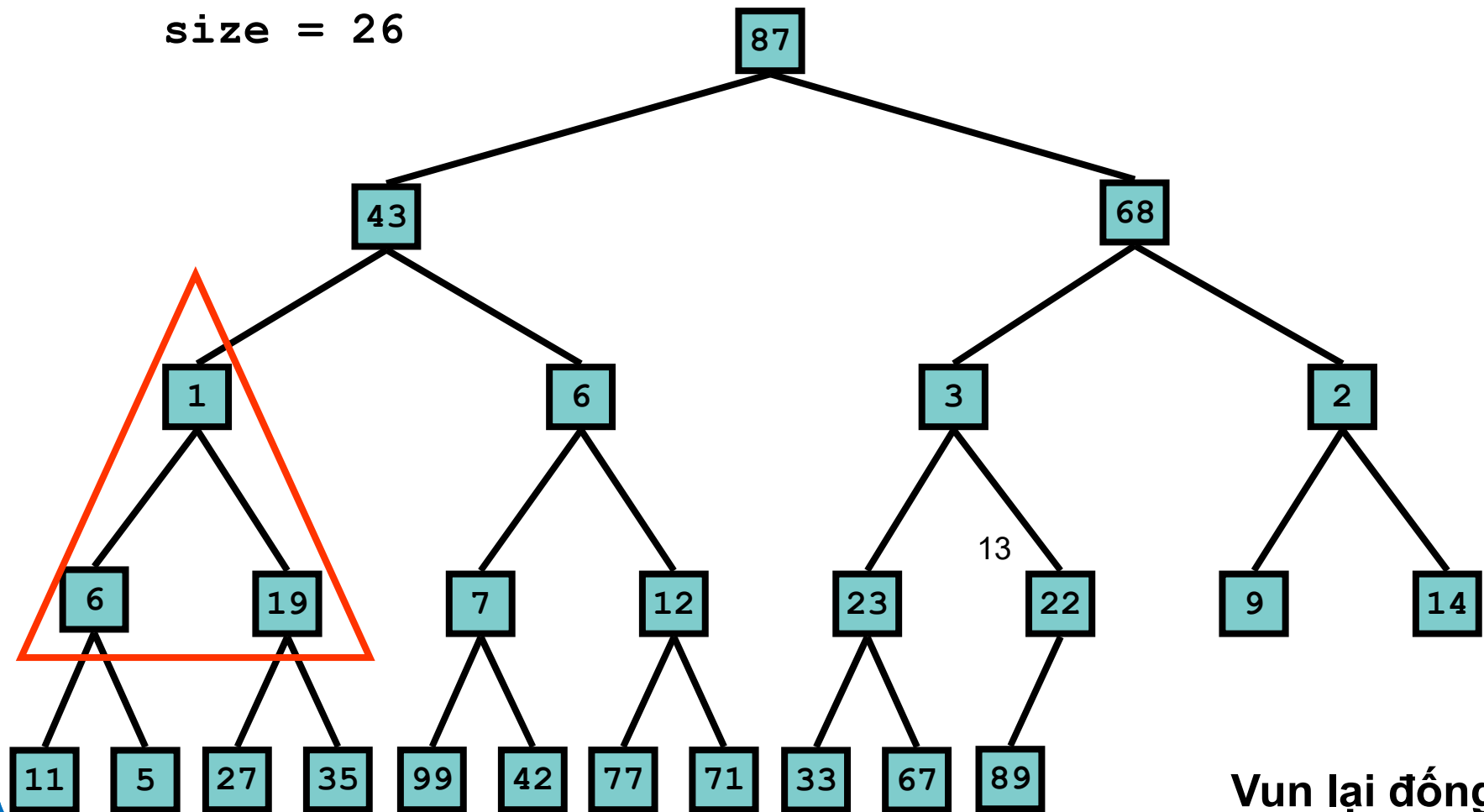
Build-Min-Heap

size = 26



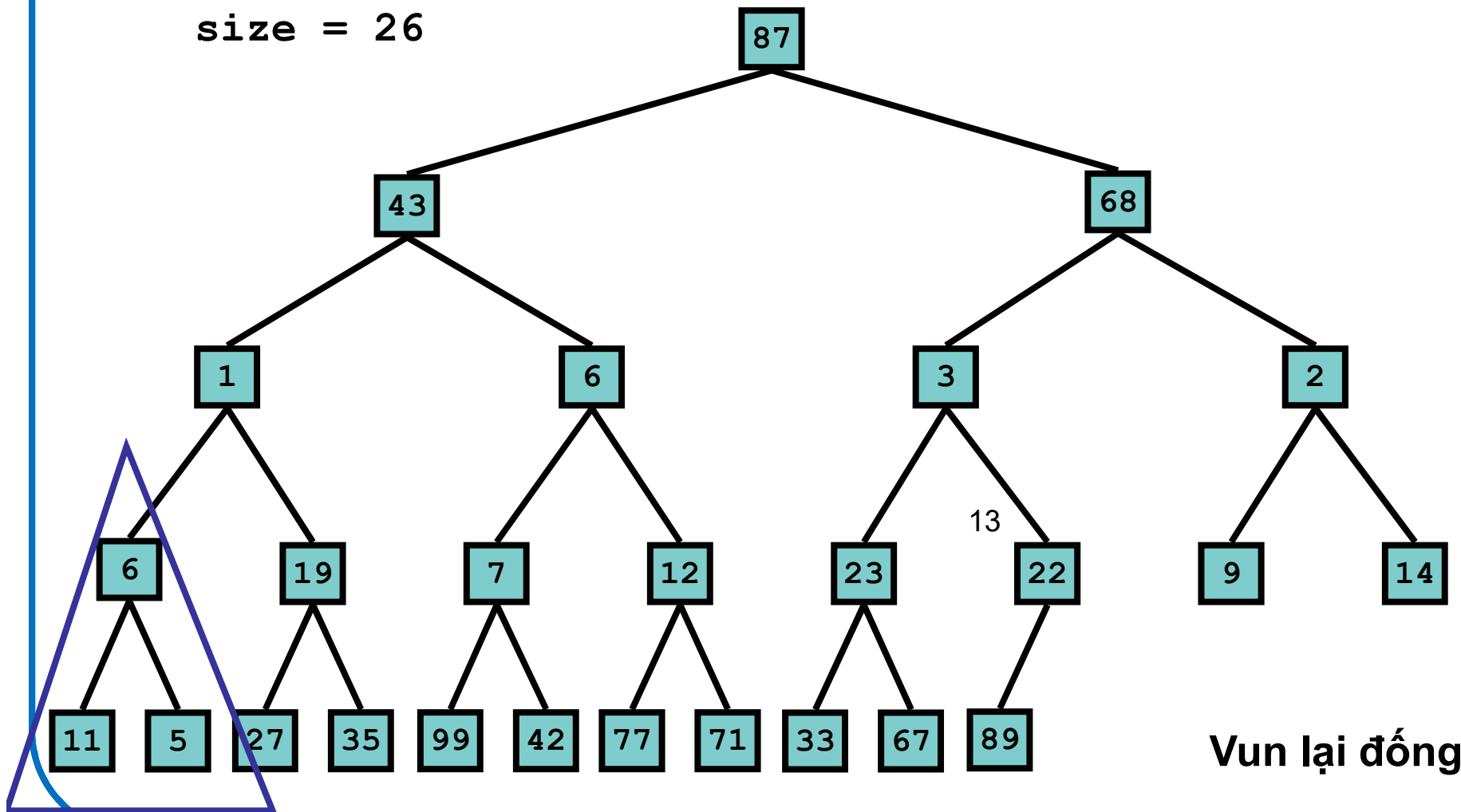
Build-Min-Heap

size = 26



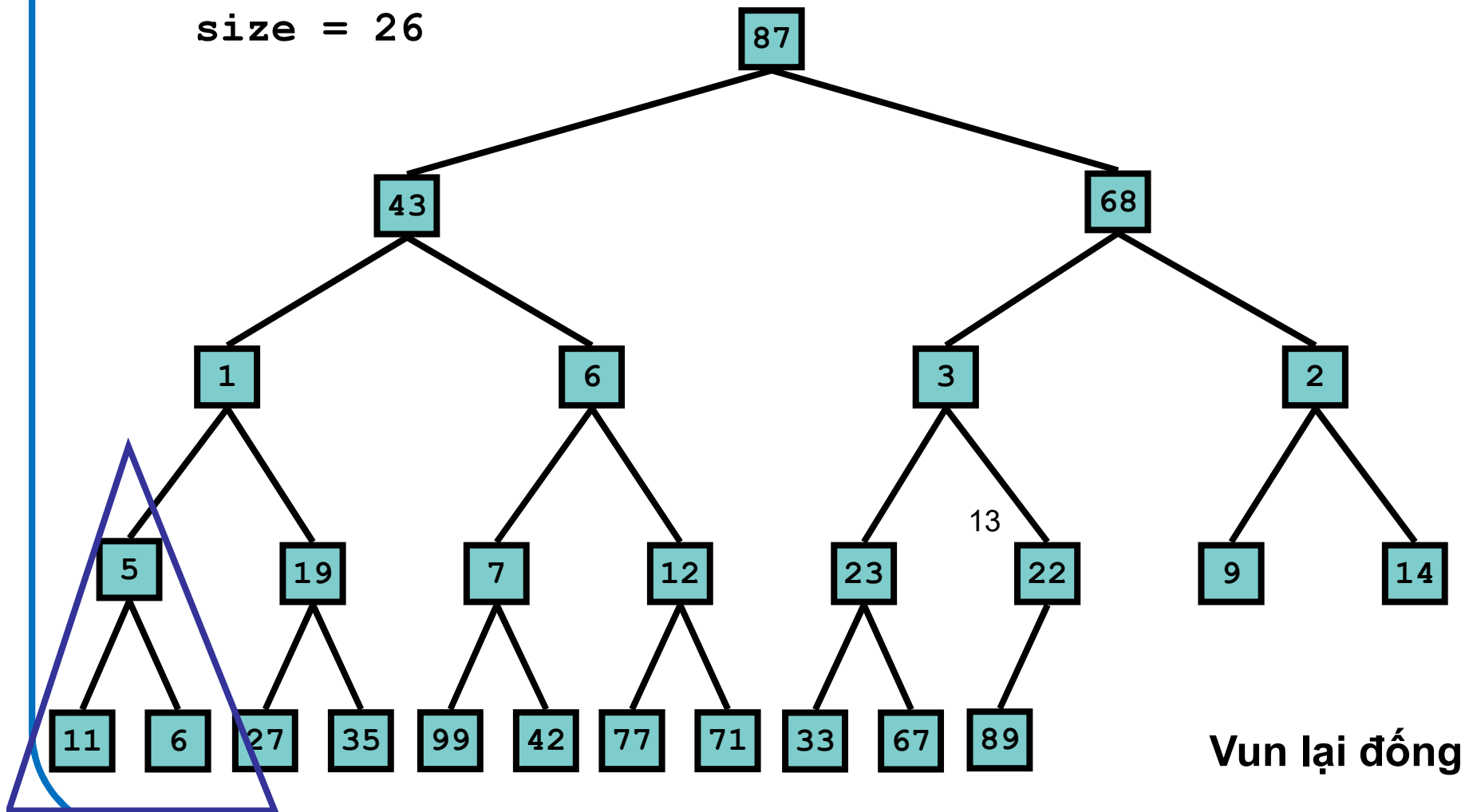
Build-Min-Heap

size = 26



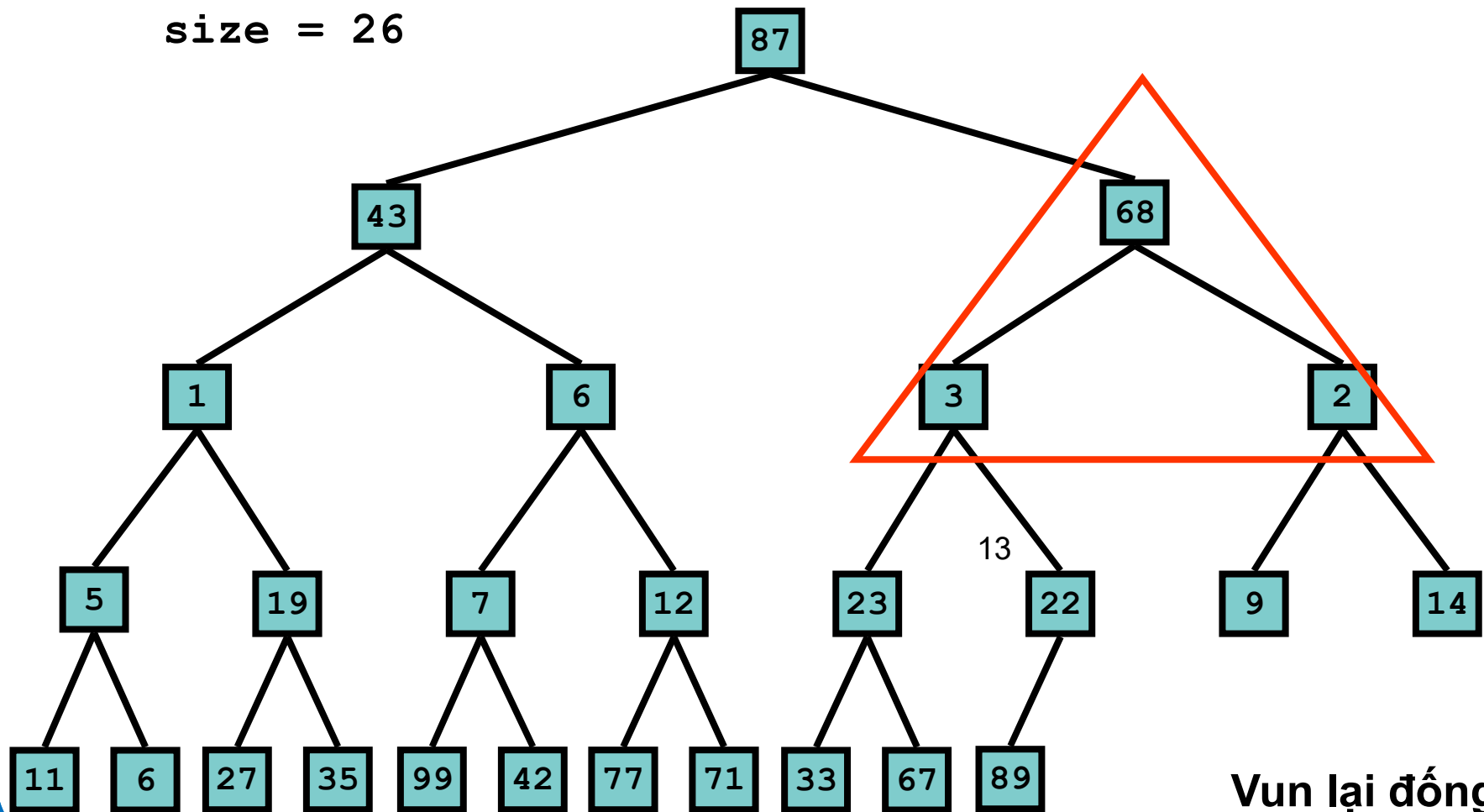
Build-Min-Heap

size = 26



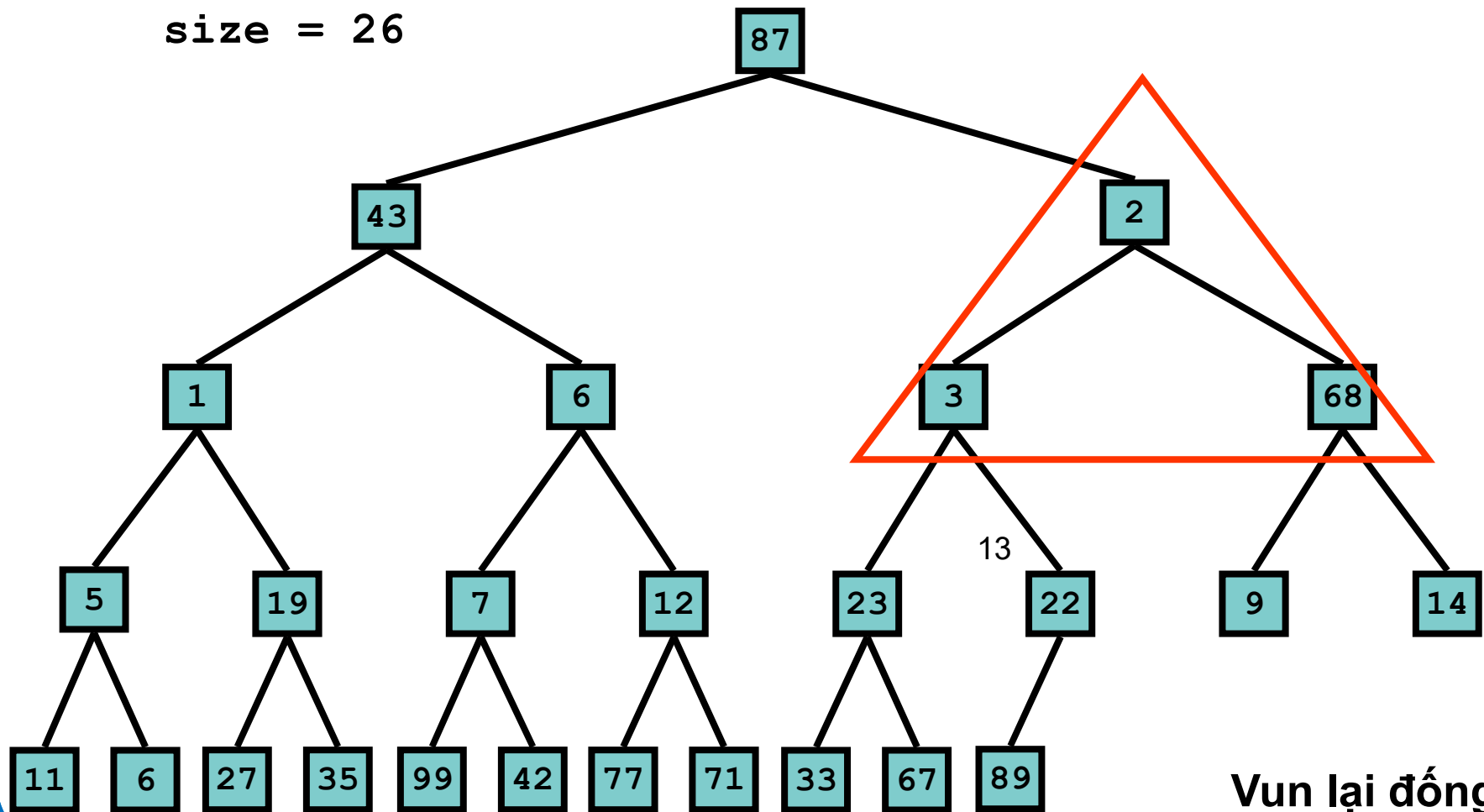
Build-Min-Heap

size = 26



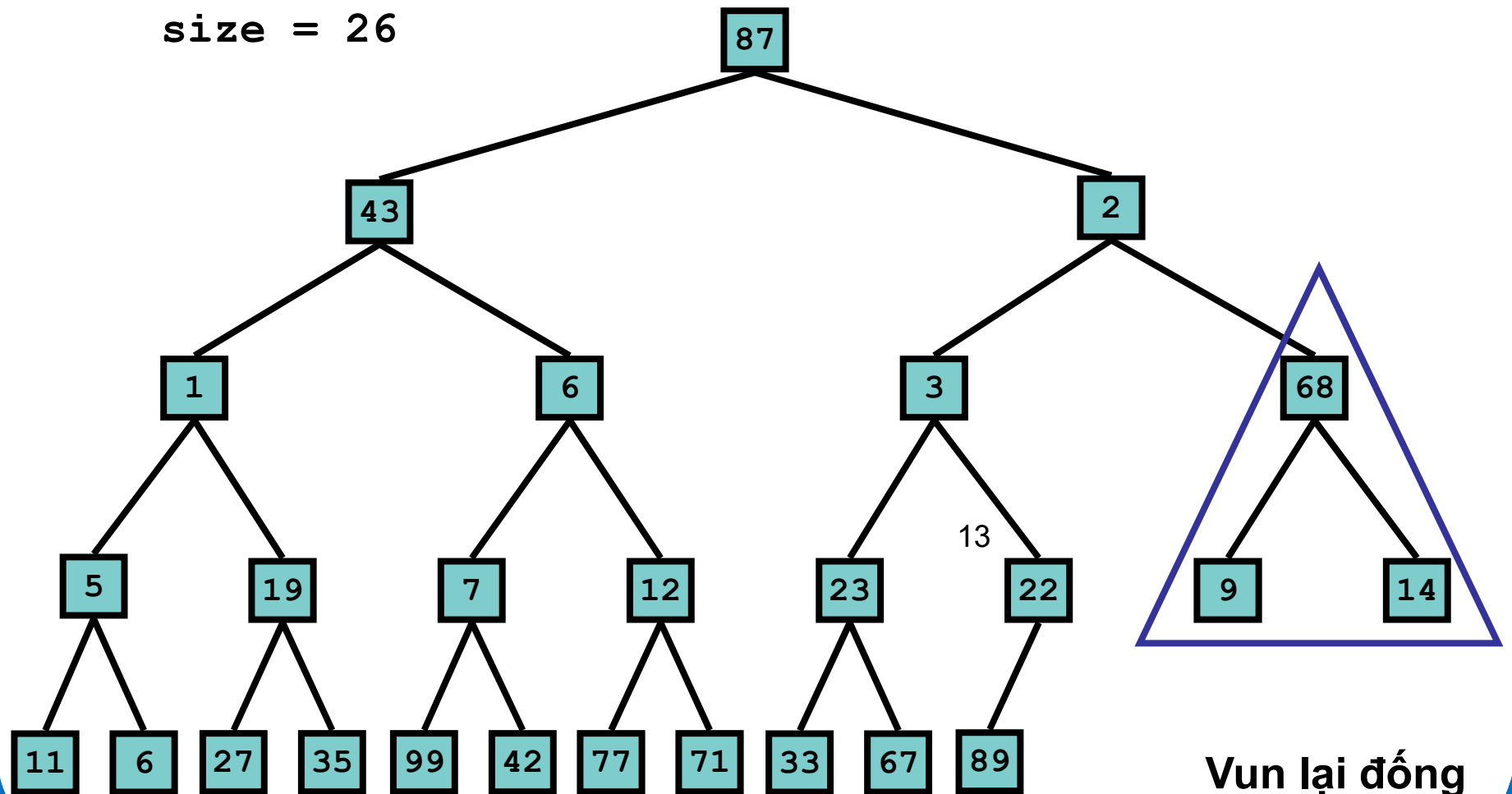
Build-Min-Heap

size = 26



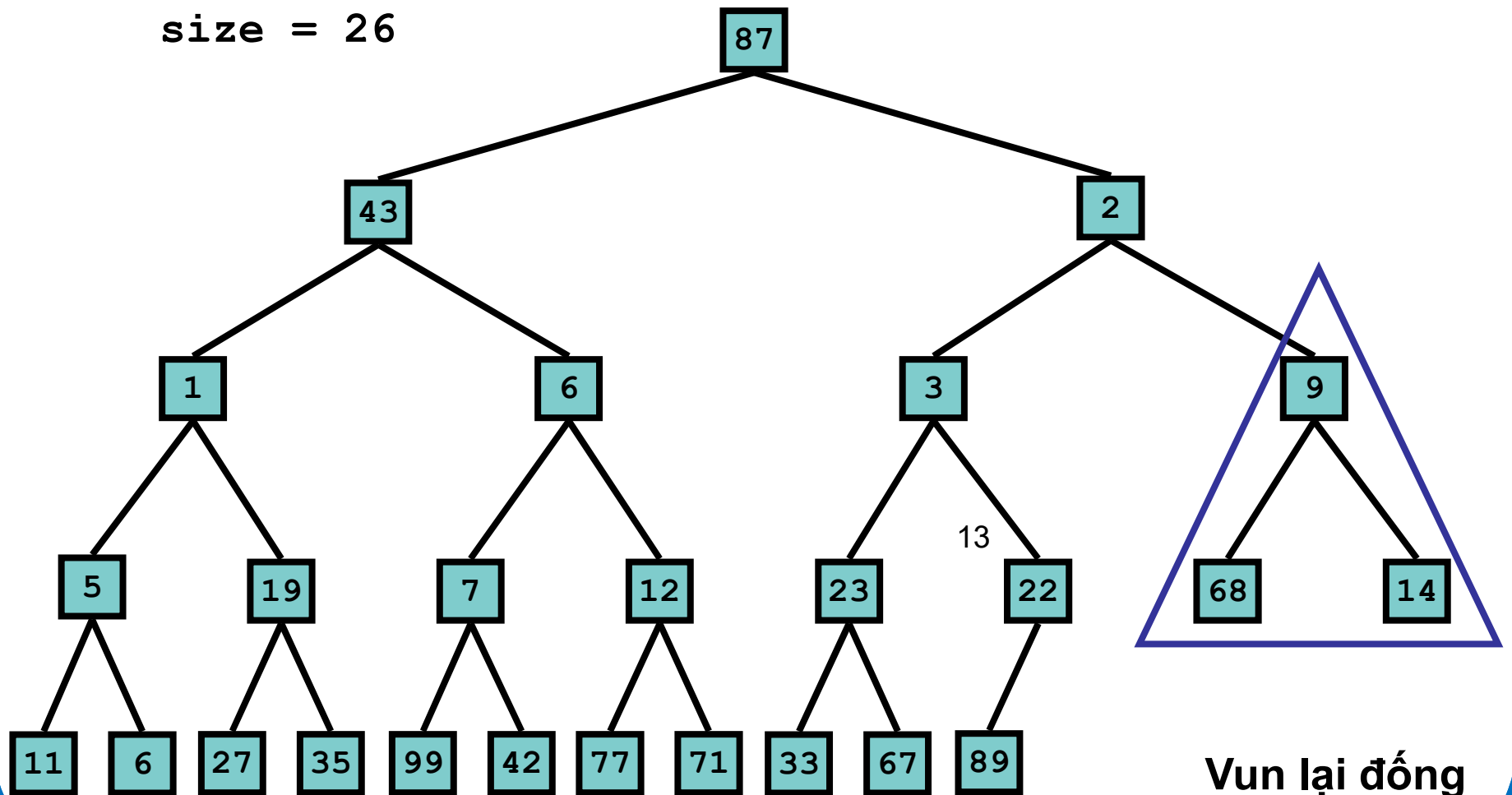
Build-Min-Heap

size = 26

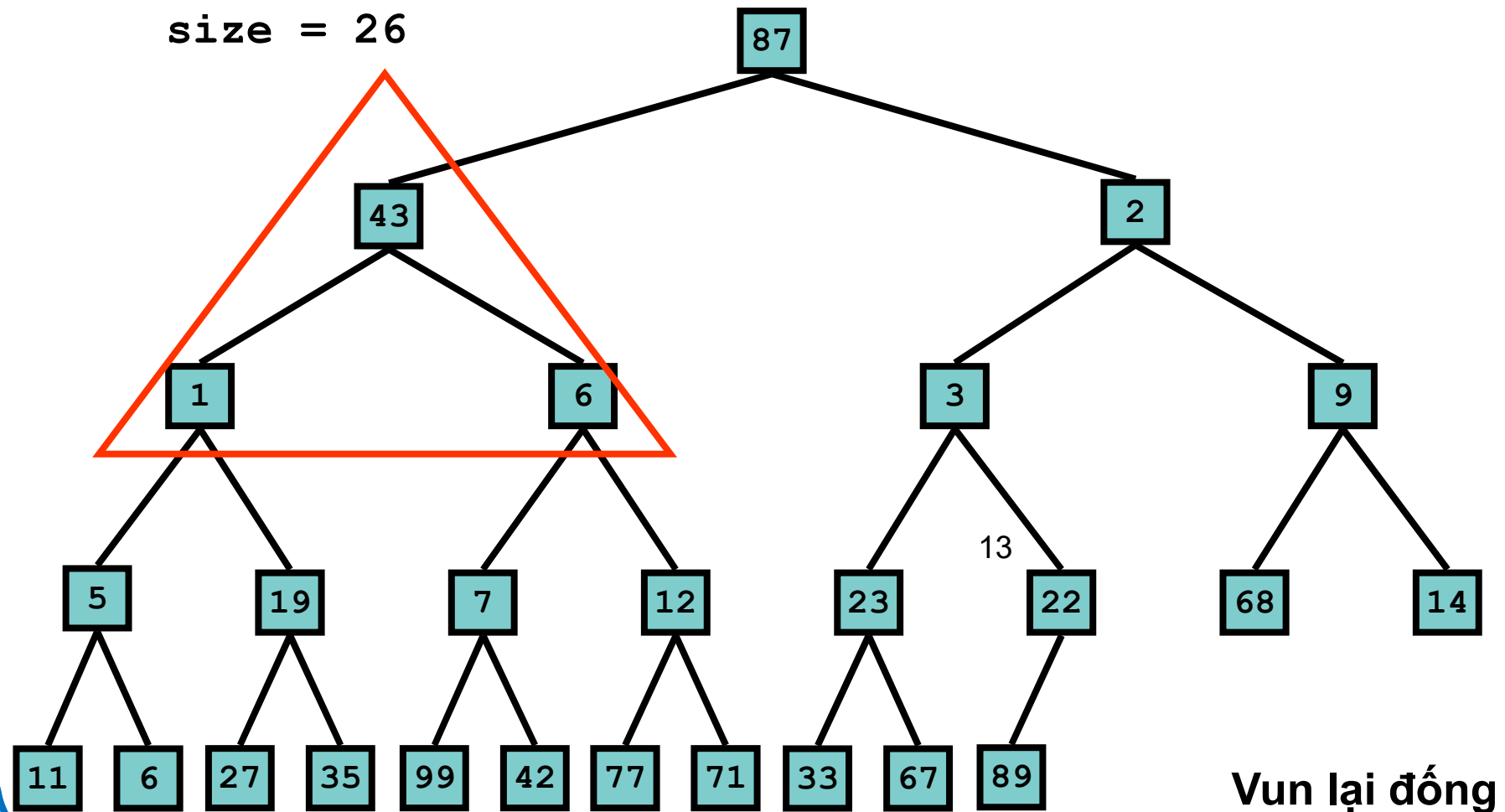


Build-Min-Heap

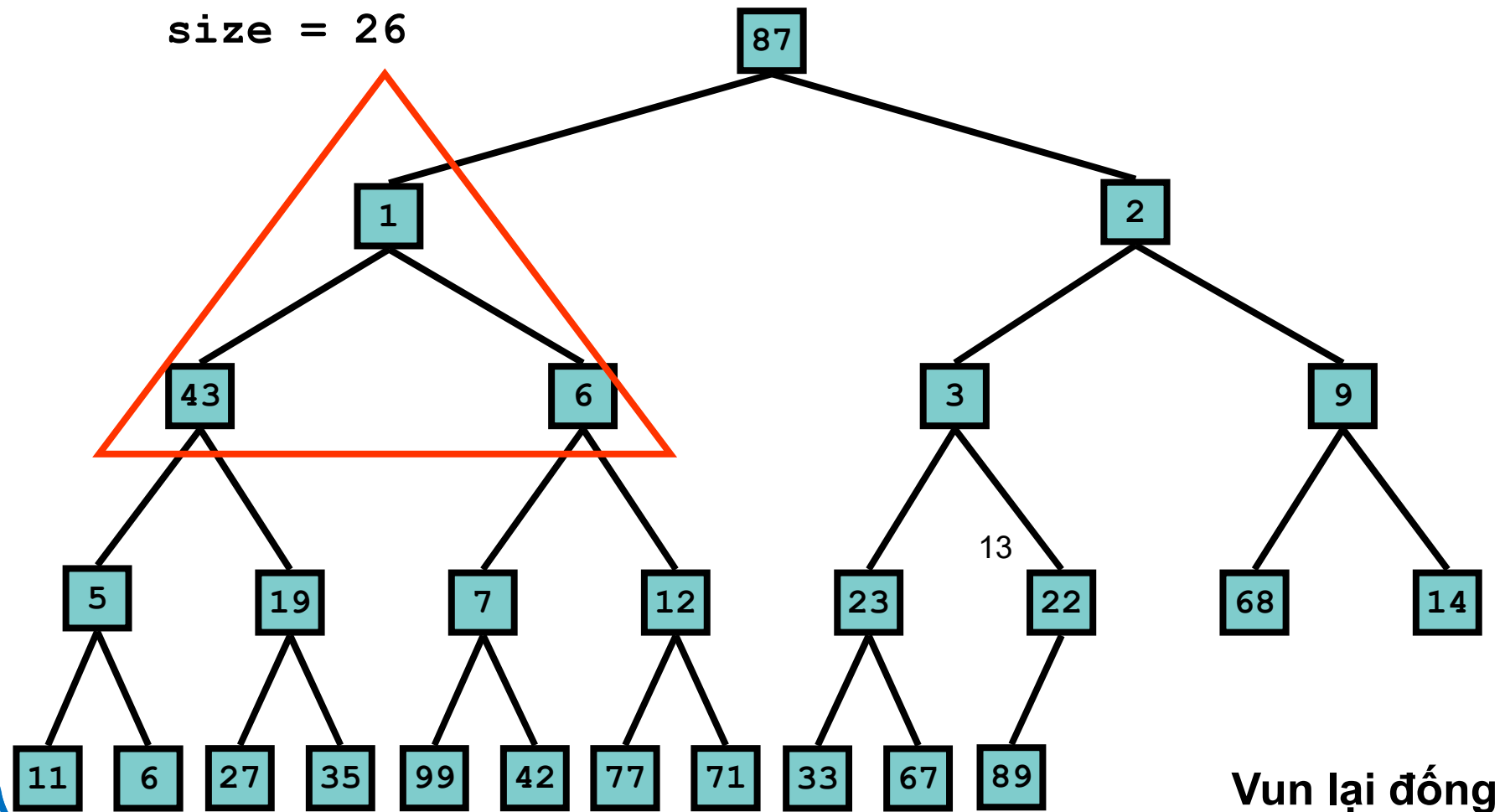
size = 26



Build-Min-Heap

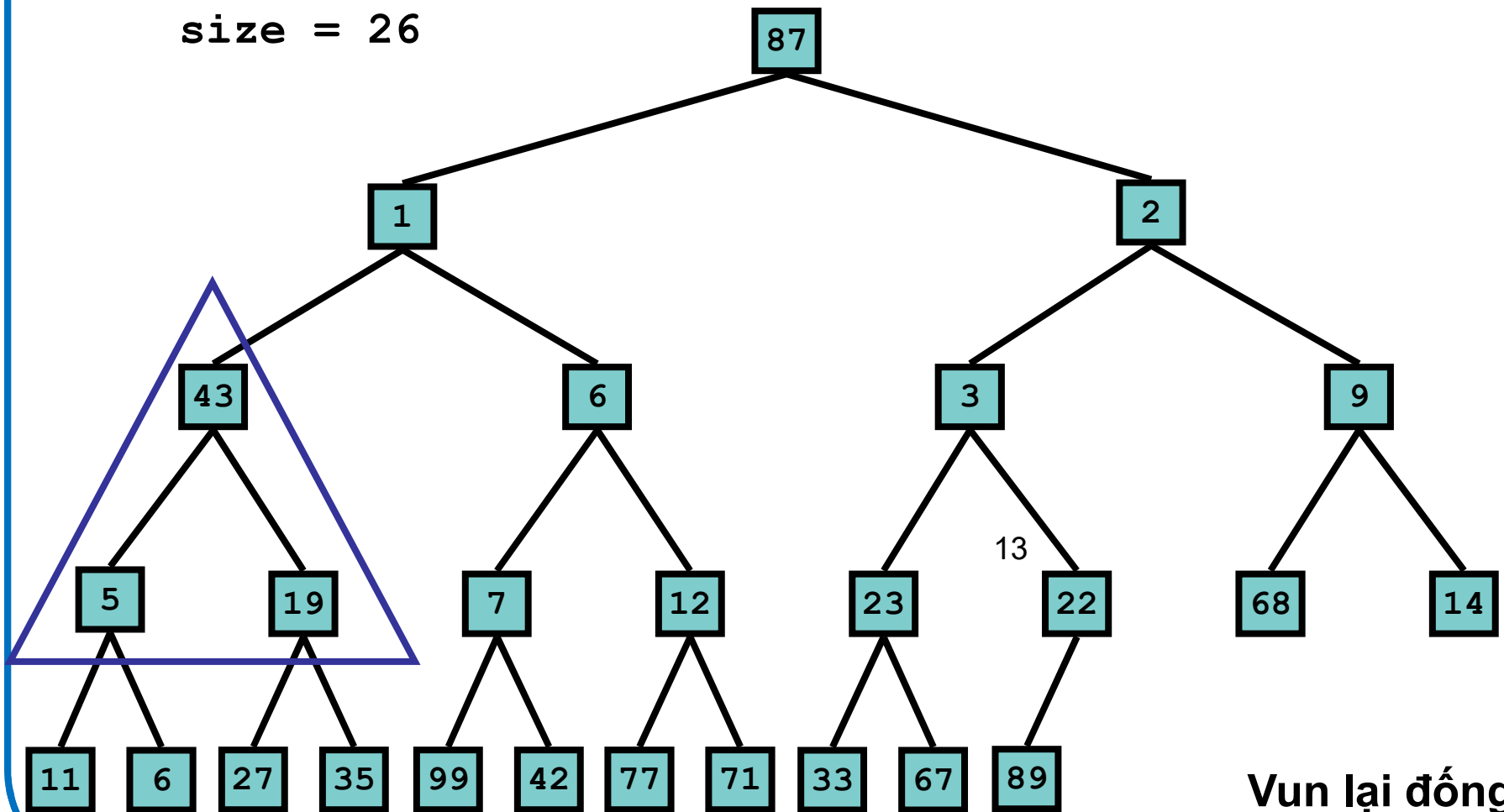


Build-Min-Heap



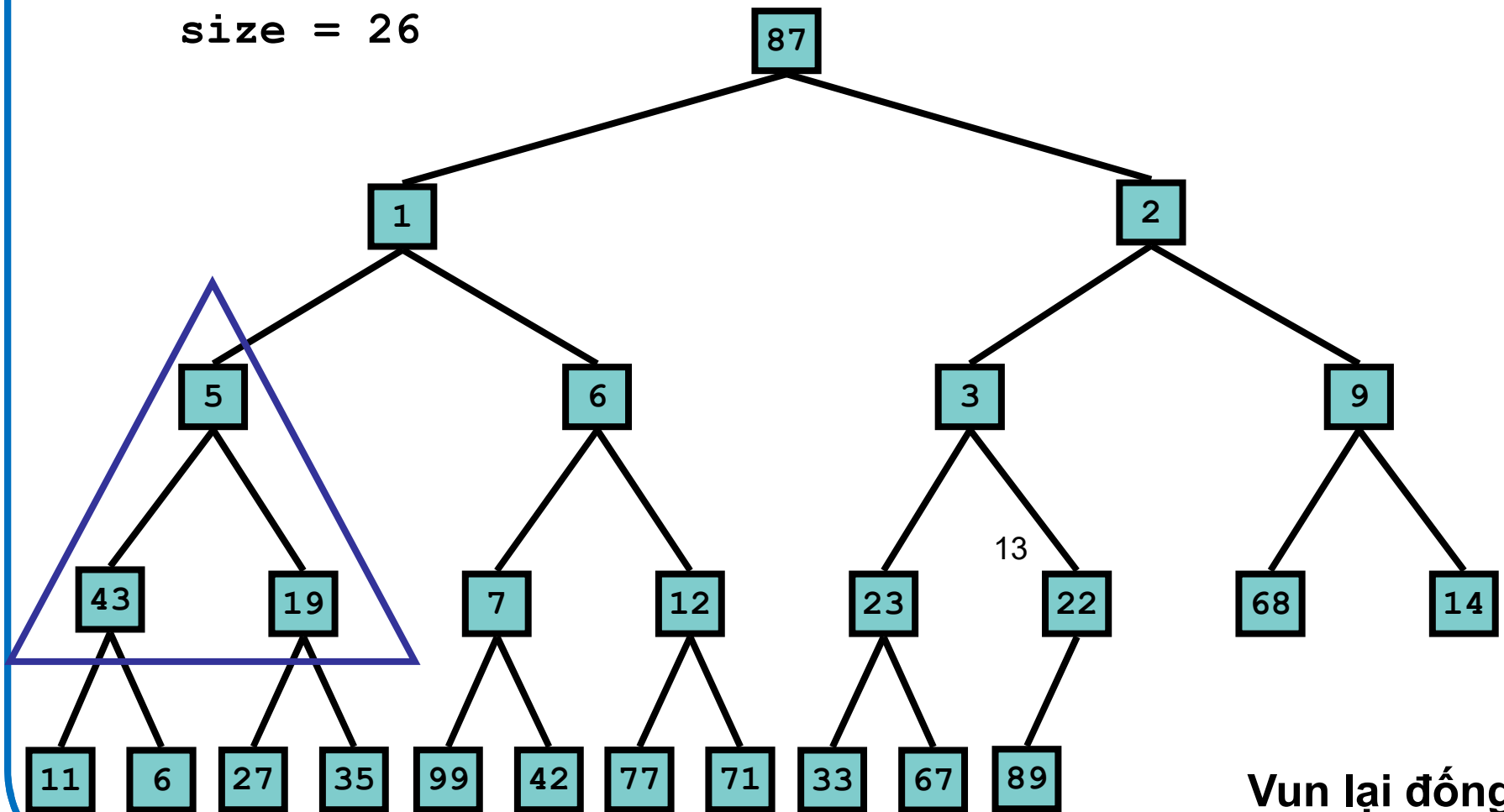
Build-Min-Heap

size = 26



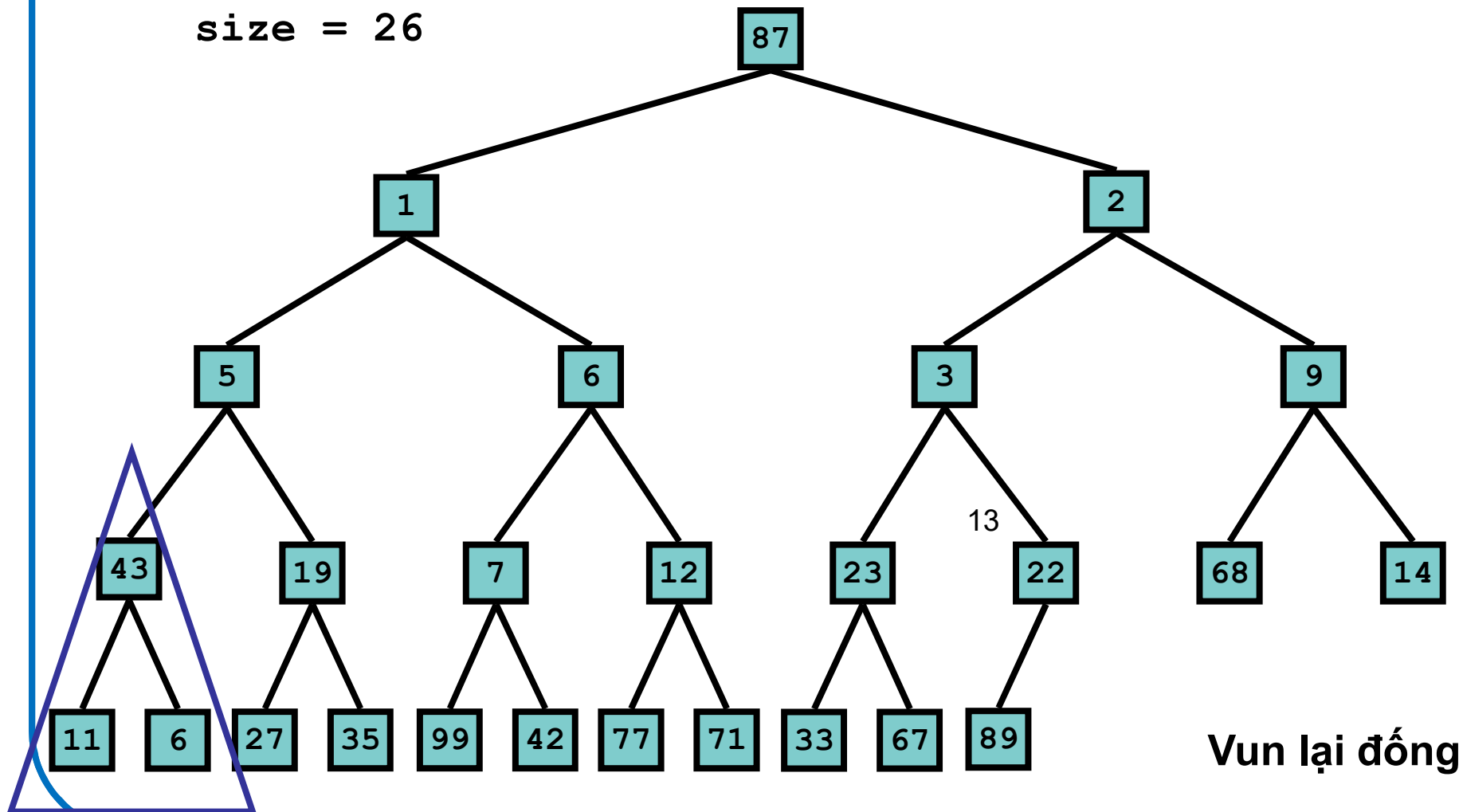
Build-Min-Heap

size = 26



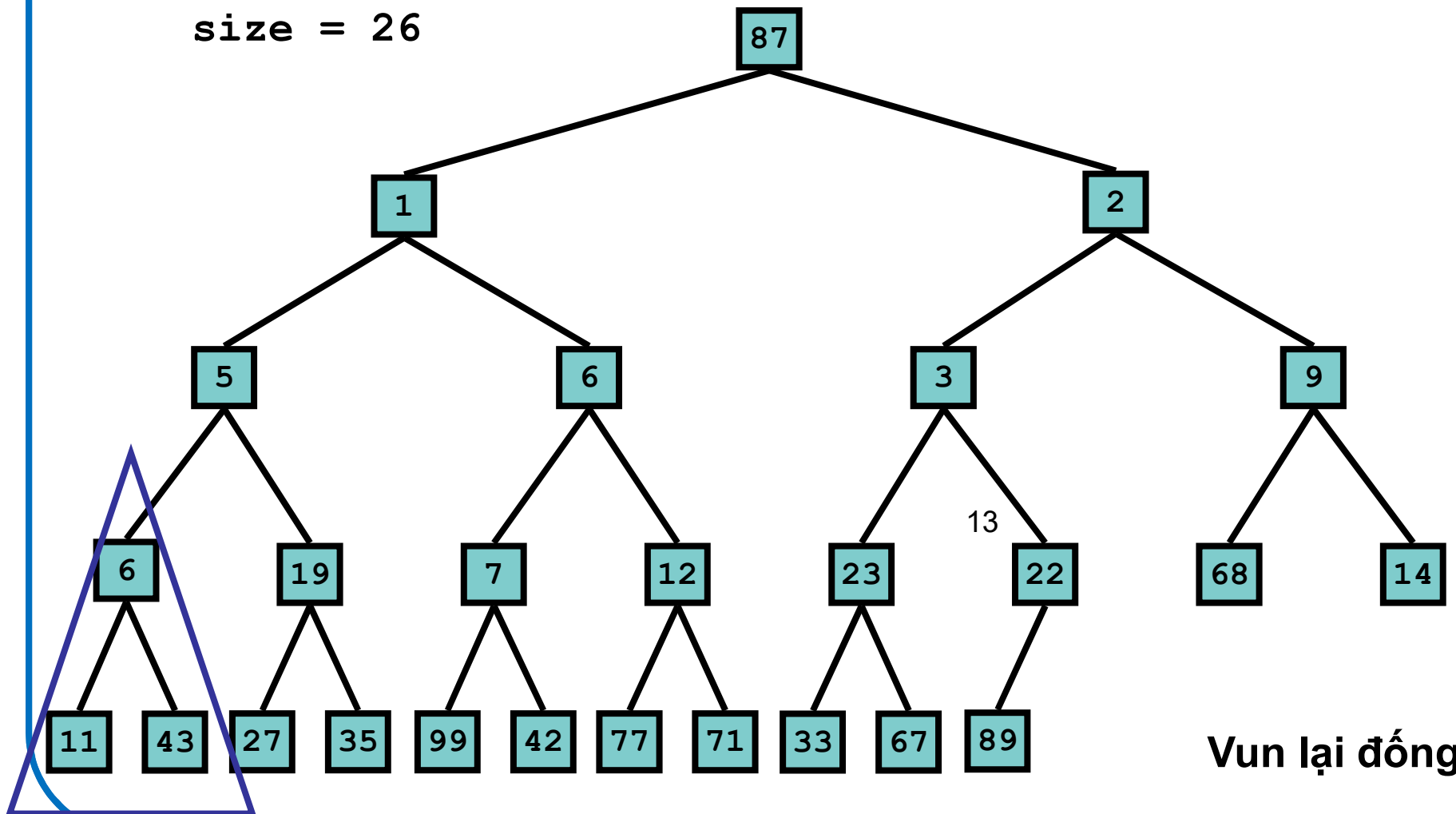
Build-Min-Heap

size = 26



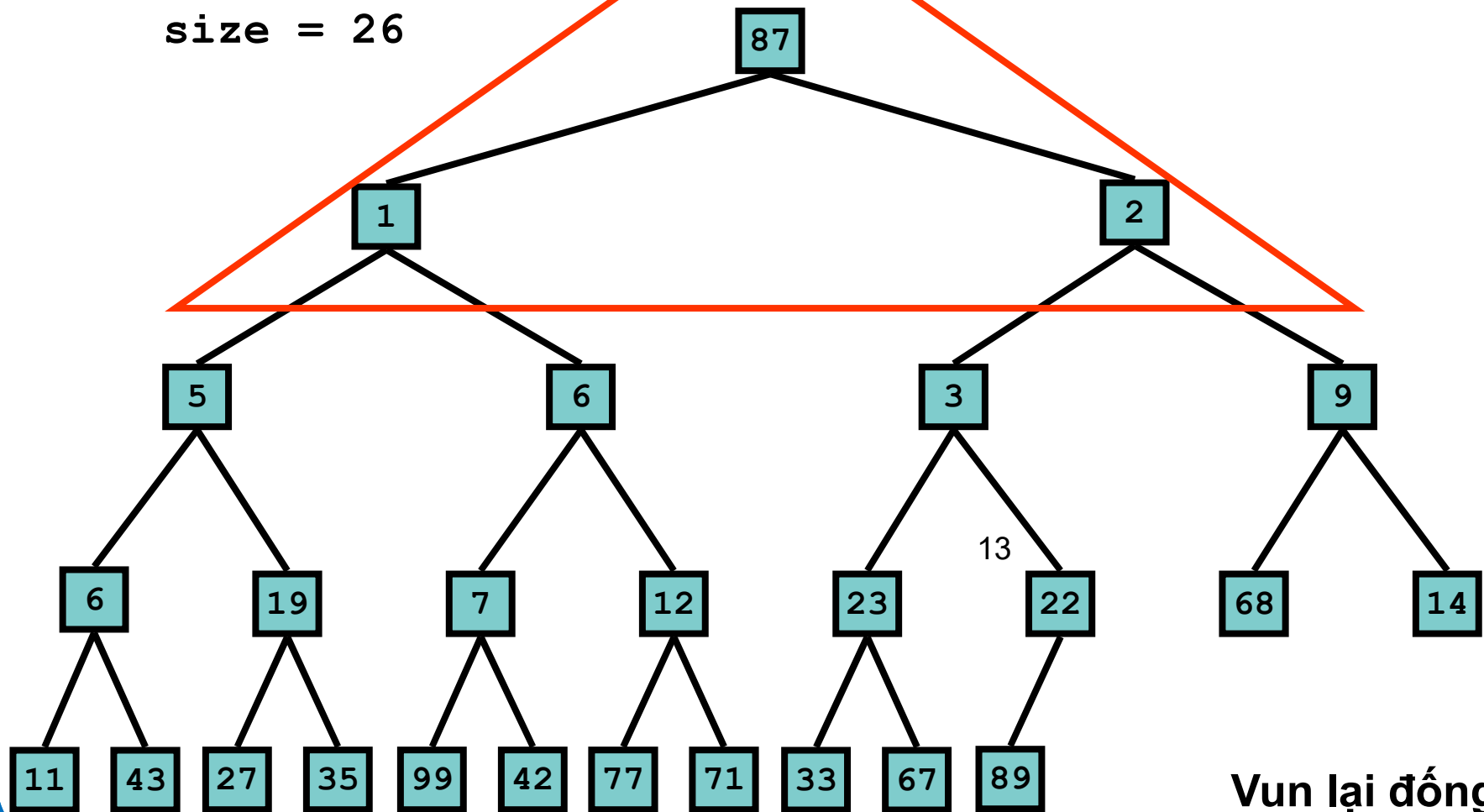
Build-Min-Heap

size = 26



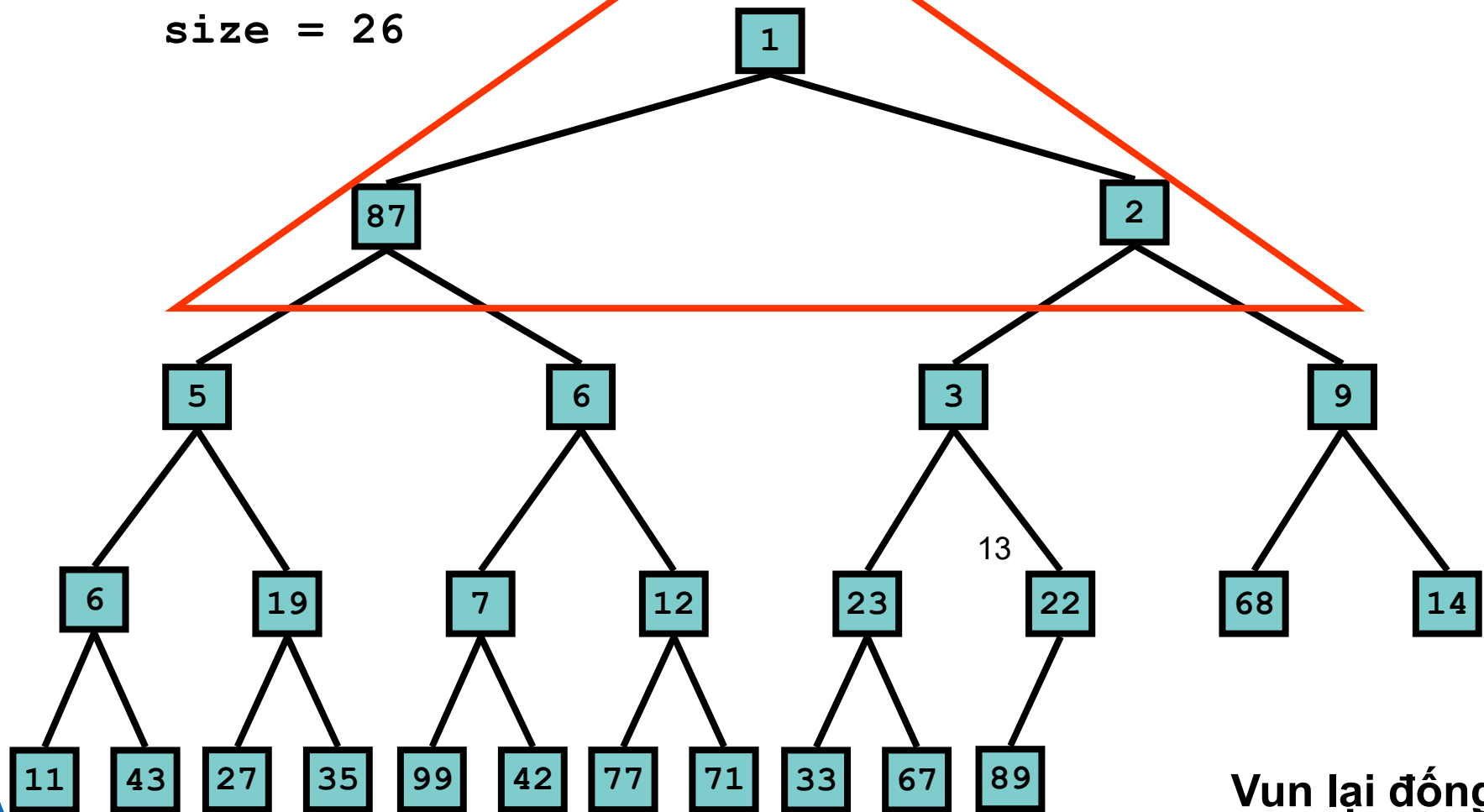
Build-Min-Heap

size = 26

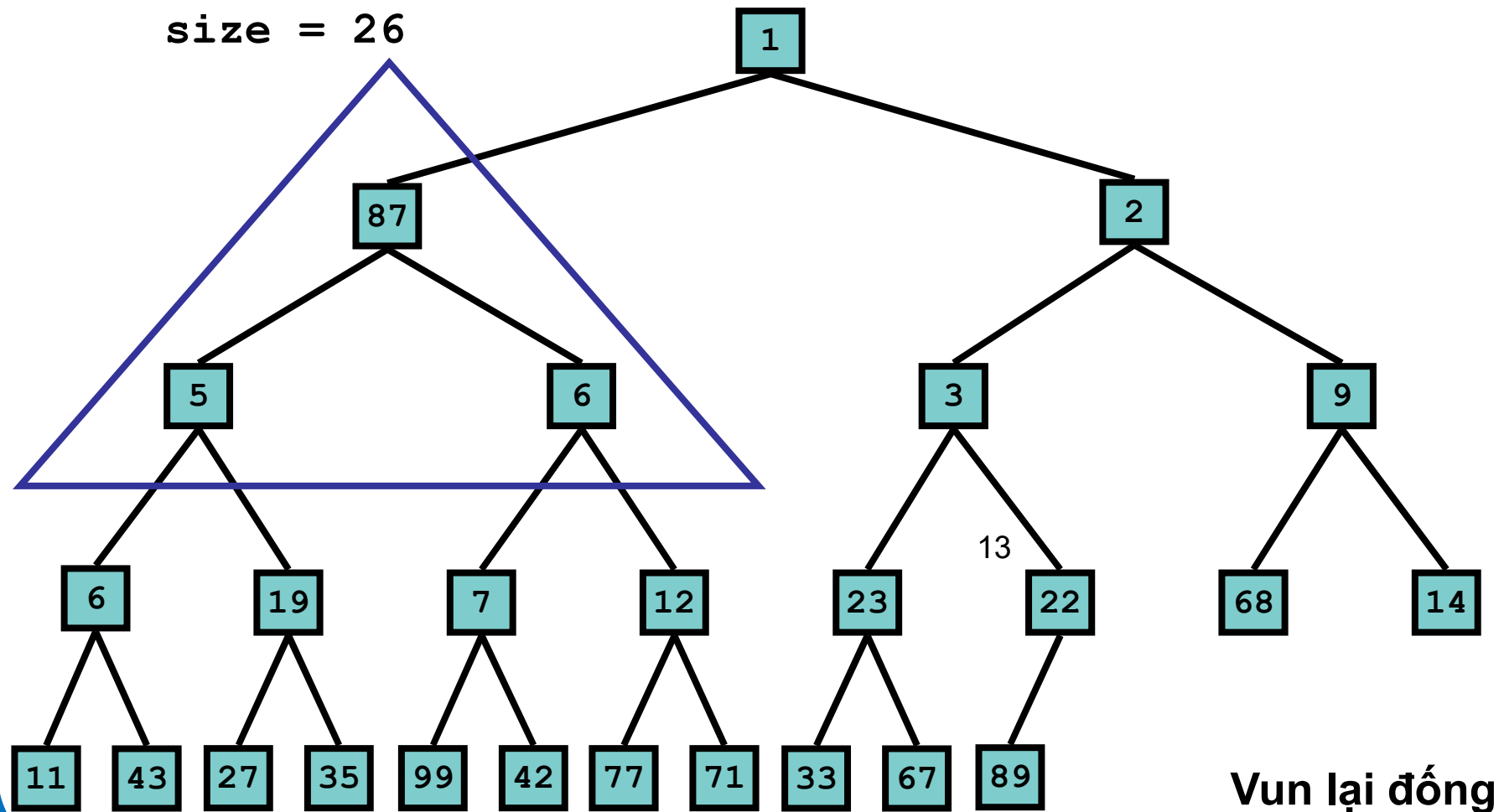


Build-Min-Heap

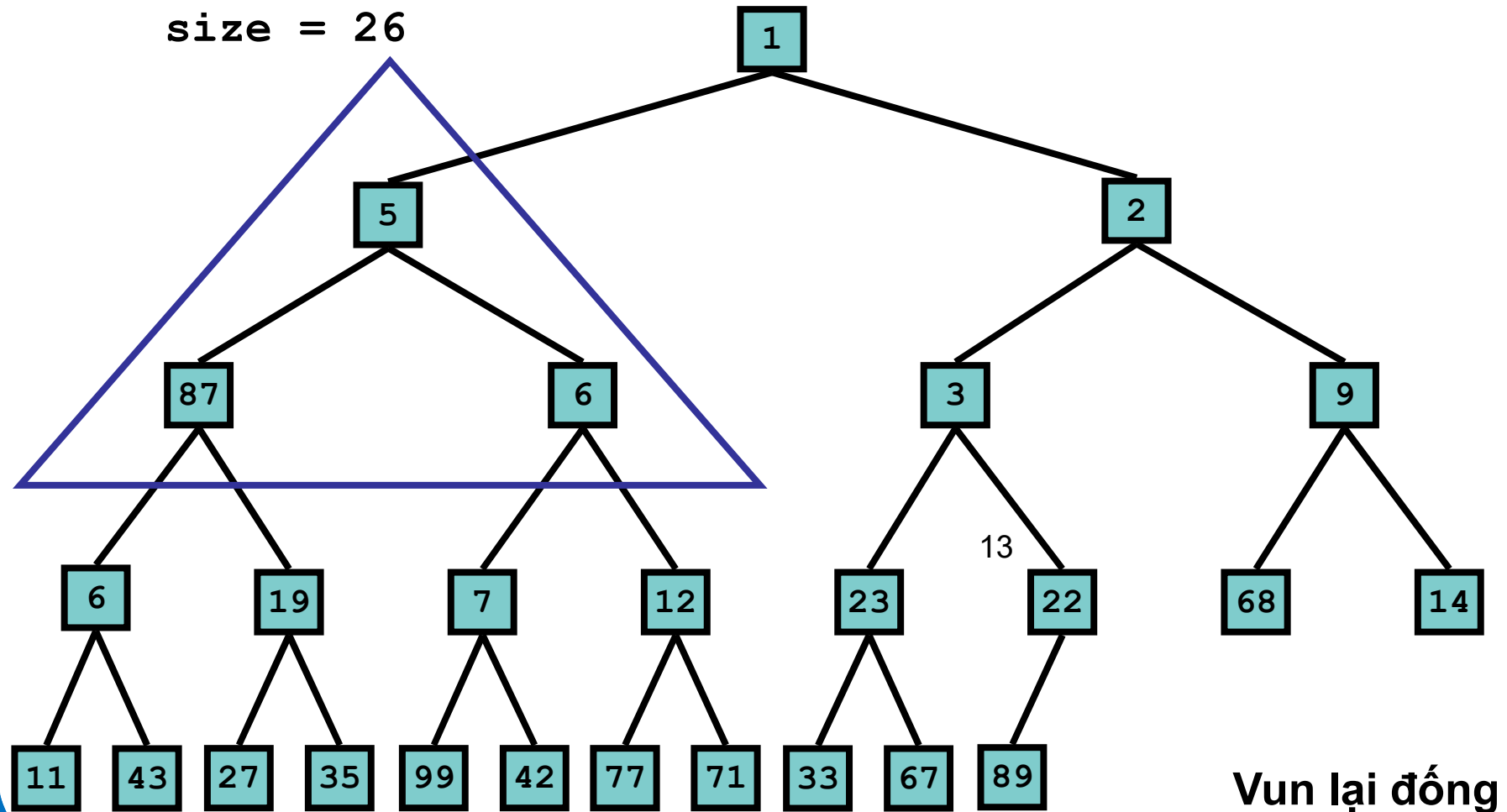
size = 26



Build-Min-Heap

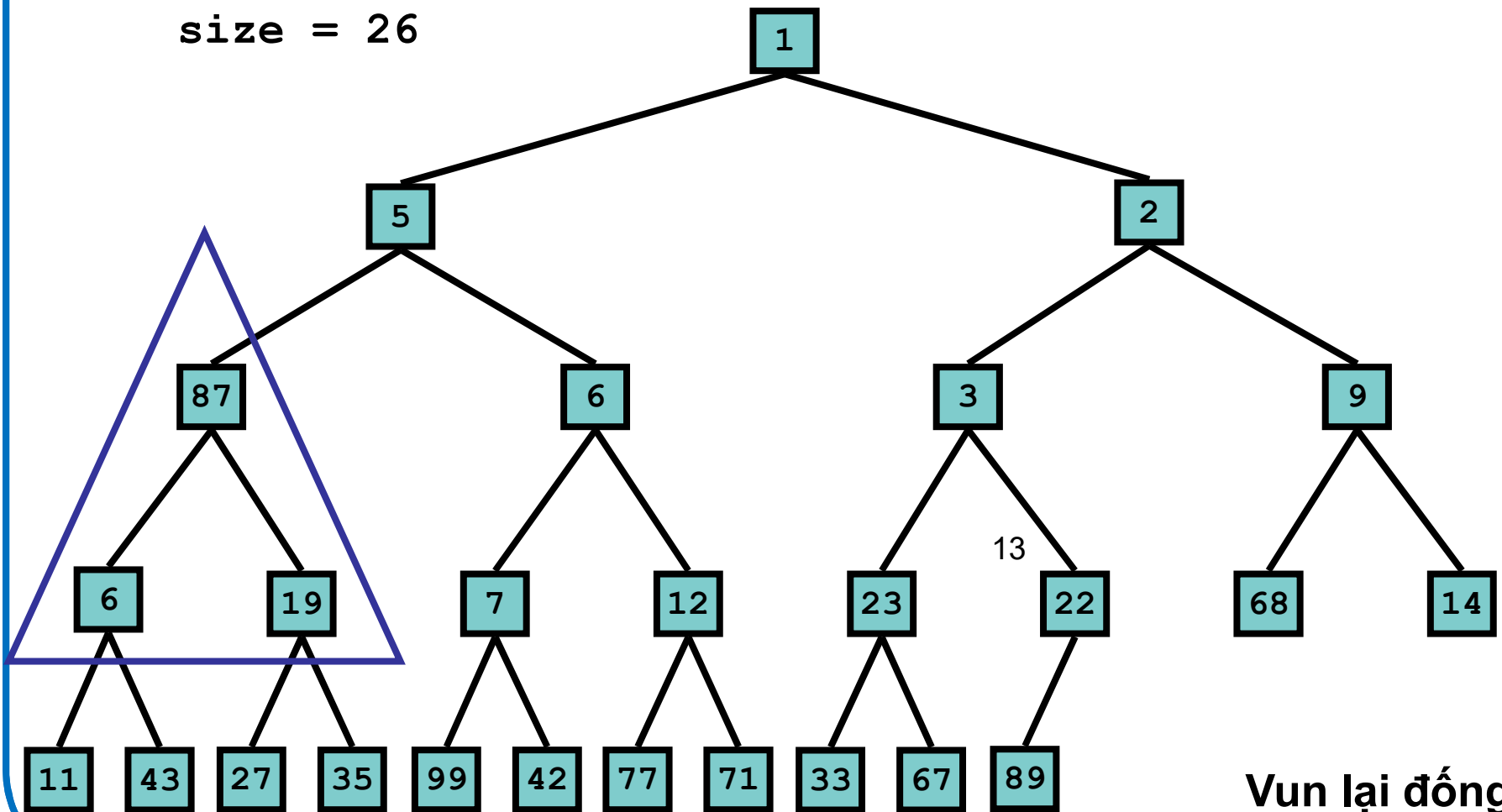


Build-Min-Heap



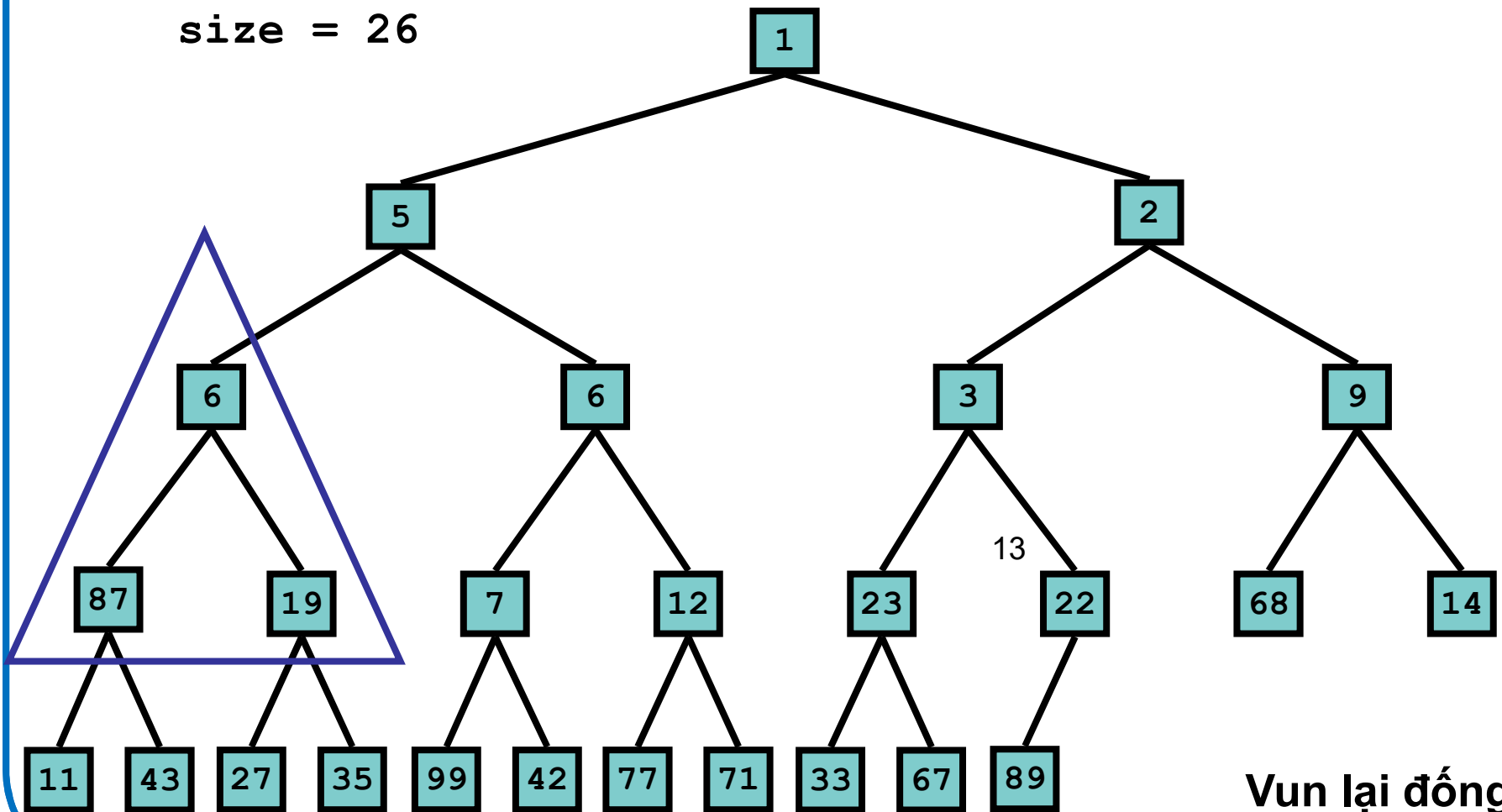
Build-Min-Heap

size = 26



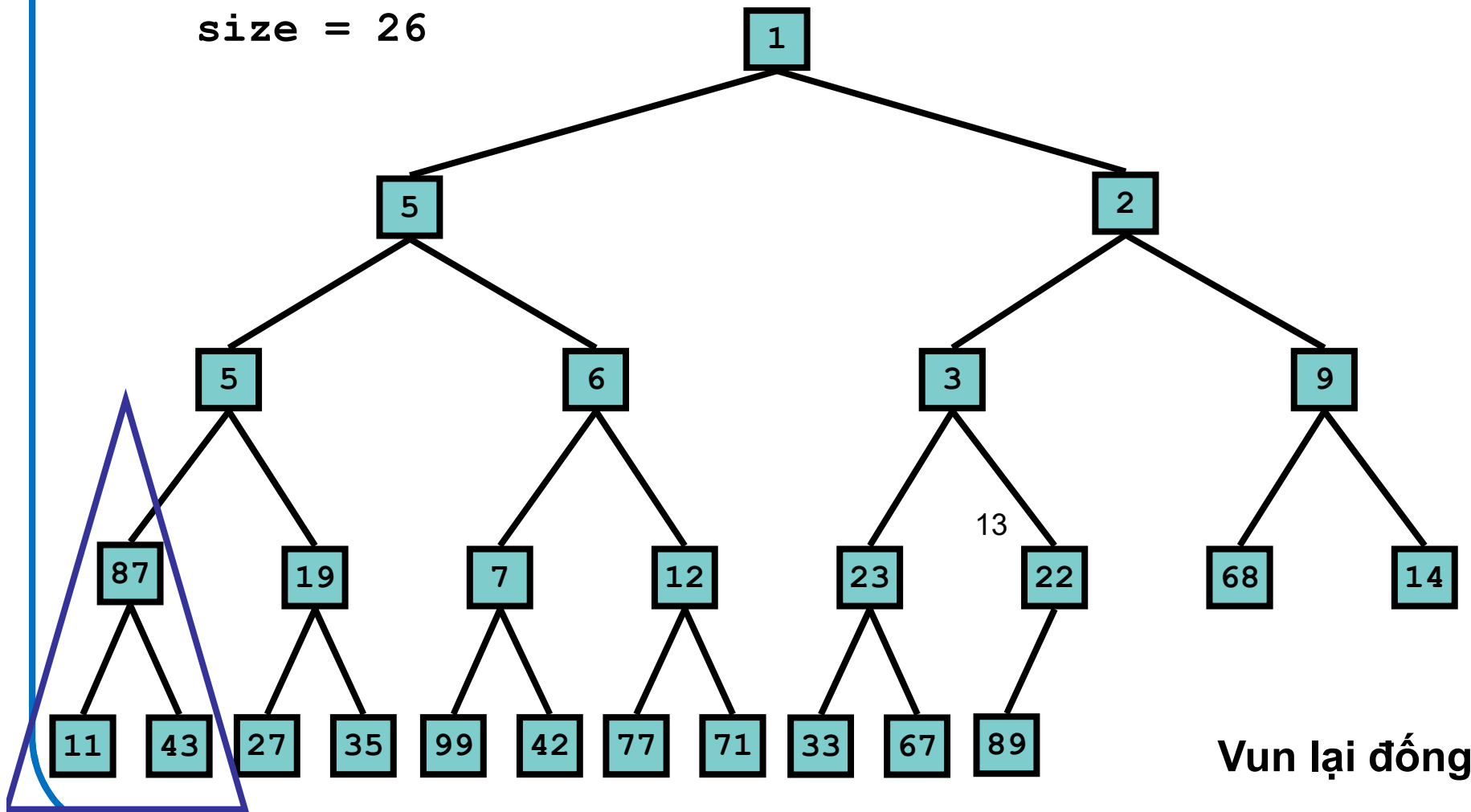
Build-Min-Heap

size = 26



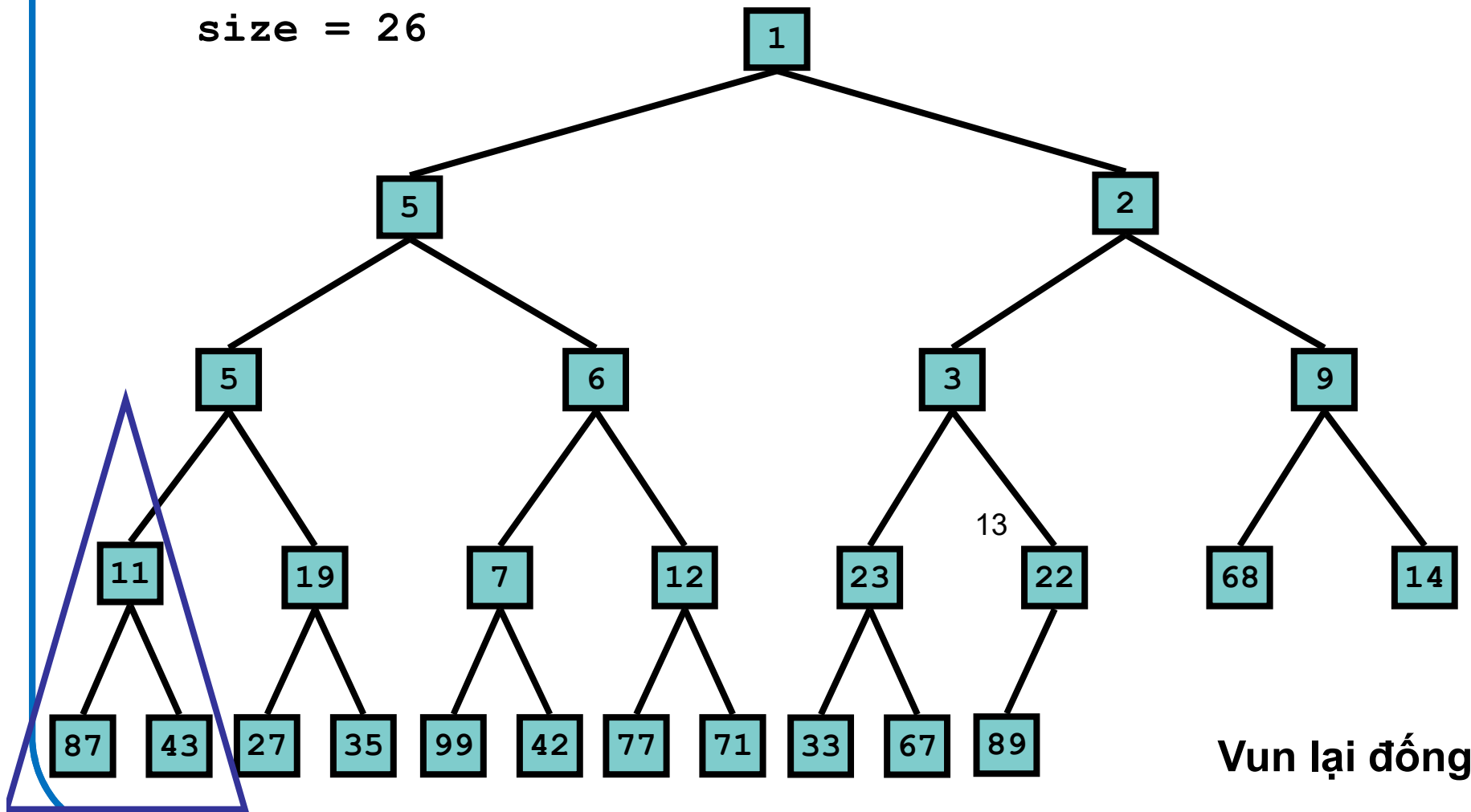
Build-Min-Heap

size = 26



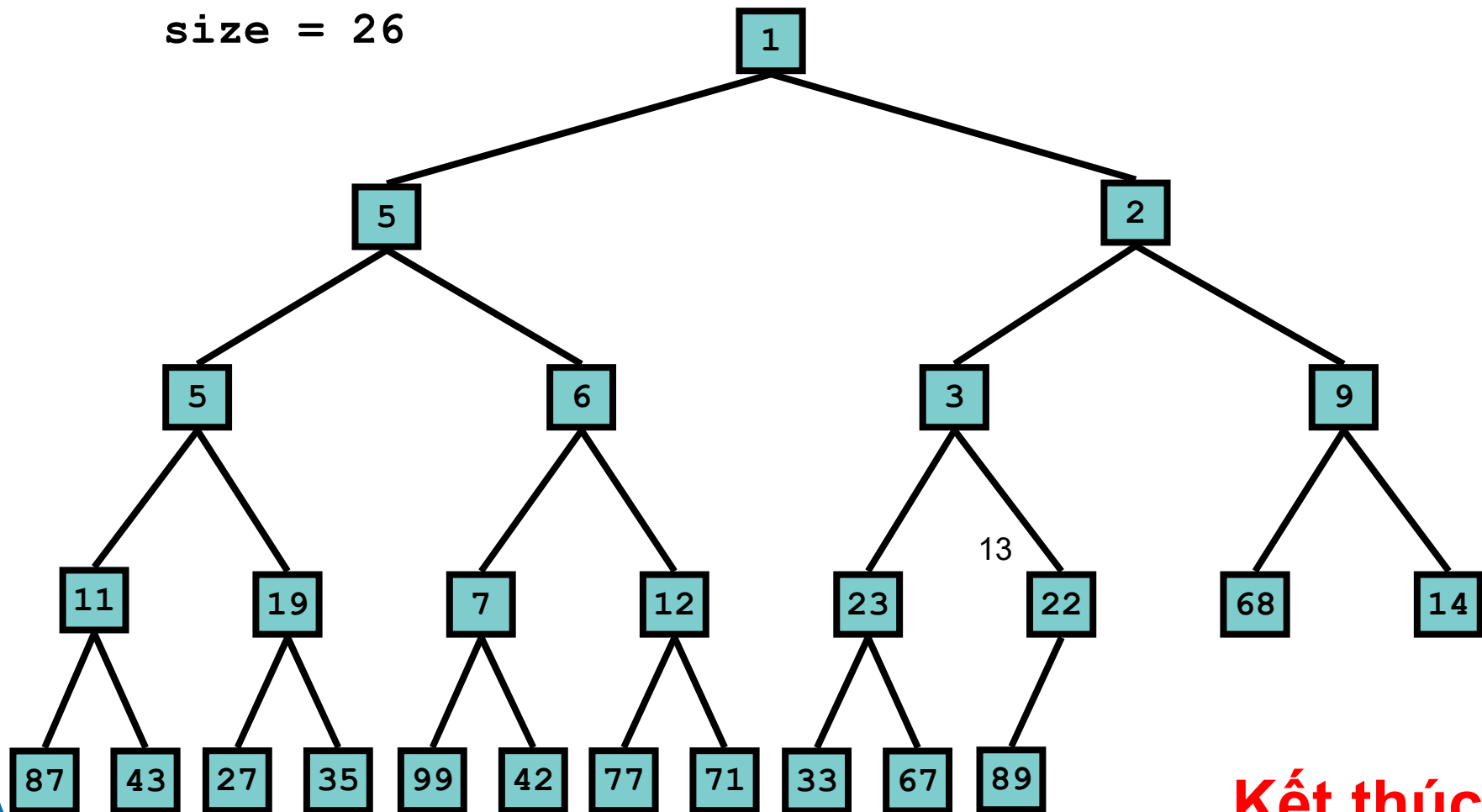
Build-Min-Heap

size = 26



Build-Min-Heap

size = 26



Thời gian tính của Buil-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)

$O(\log n)$ $\left. \vphantom{\begin{matrix} 1 \\ 2 \\ 3 \end{matrix}} \right\} O(n)$

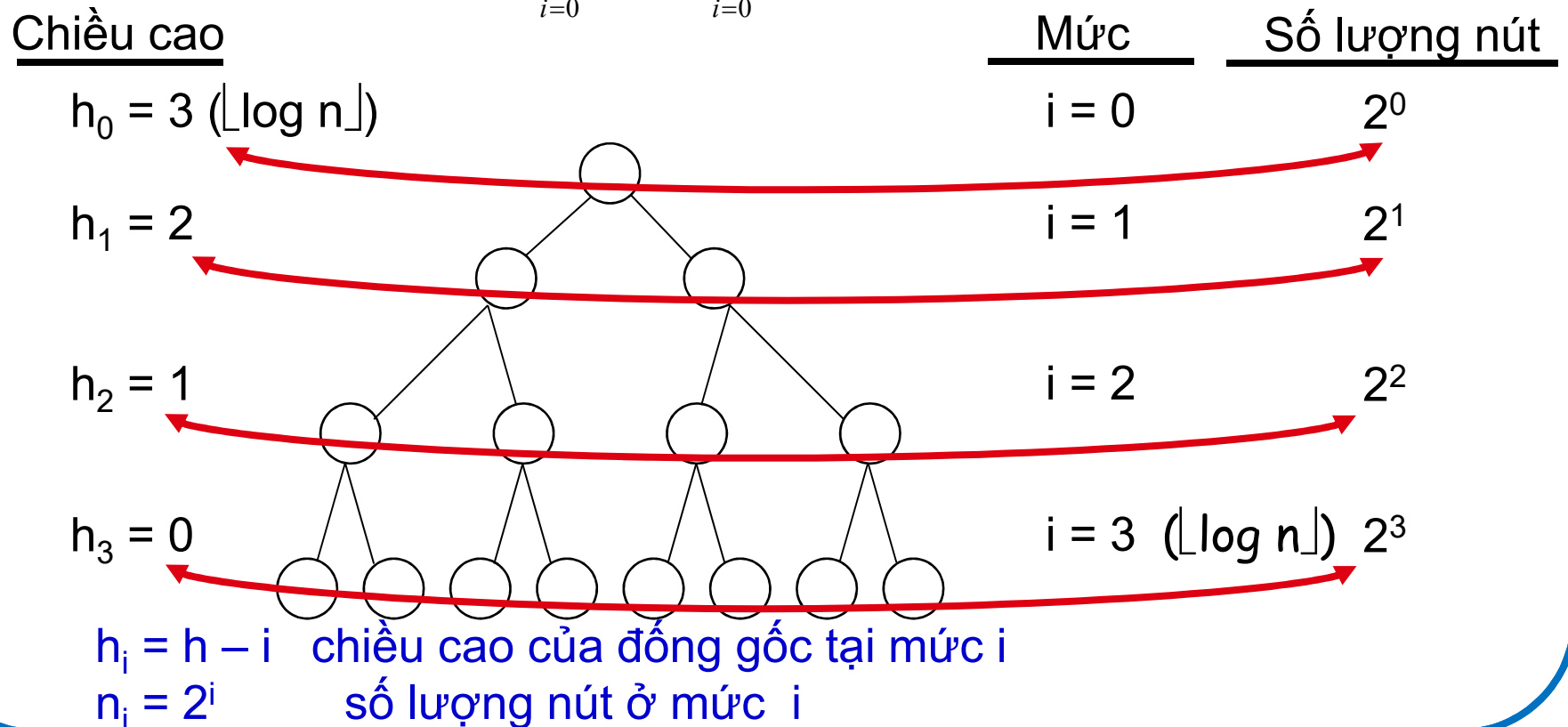
\Rightarrow Thời gian tính là: $O(n \log n)$

- Đánh giá này là không sát !

Thời gian tính của Build-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

$$T(n) = \sum_{i=0}^h n_i h_i \quad (\text{Chi phí Heapify tại mức } i) \times (\text{số lượng nút trên mức này})$$

$$= \sum_{i=0}^h 2^i (h - i) \quad \text{Thay giá trị của } n_i \text{ và } h_i \text{ tính được ở trên}$$

$$= \sum_{i=0}^h \frac{h-i}{2^{h-i}} 2^h \quad \text{Nhân cả tử và mẫu với } 2^h \text{ và viết } 2^i \text{ là } \frac{1}{2^{-i}}$$

$$= 2^h \sum_{k=0}^h \frac{k}{2^k} \quad \text{Đổi biến: } k = h - i$$

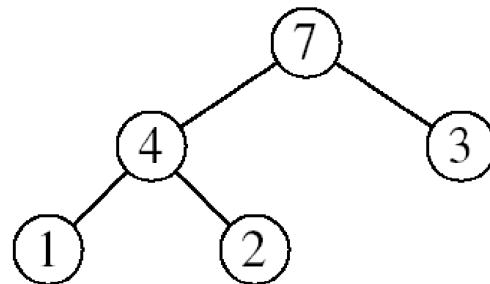
$$\leq n \sum_{k=0}^{\infty} \frac{k}{2^k} \quad \text{Thay tổng hữu hạn bởi tổng vô hạn và } h = \log n$$

$$= O(n) \quad \text{Tổng trên là nhỏ hơn 2}$$

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

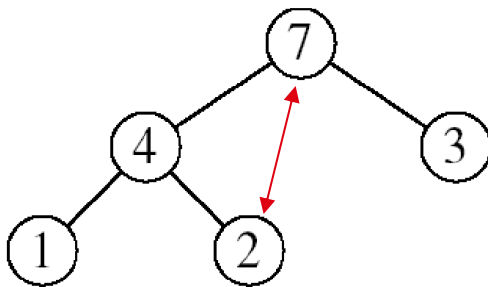
5.5.2. Sắp xếp vun đống - Heapsort

- 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

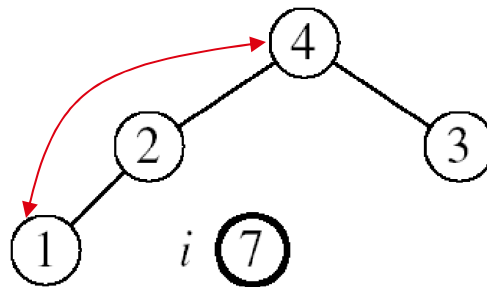


Ví dụ:

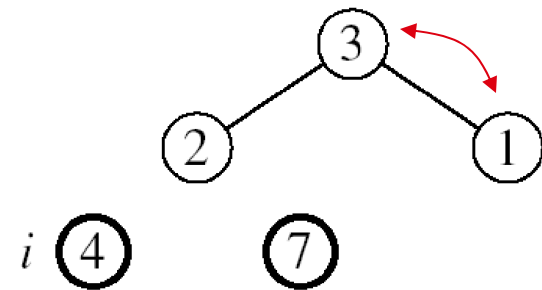
$A=[7, 4, 3, 1, 2]$



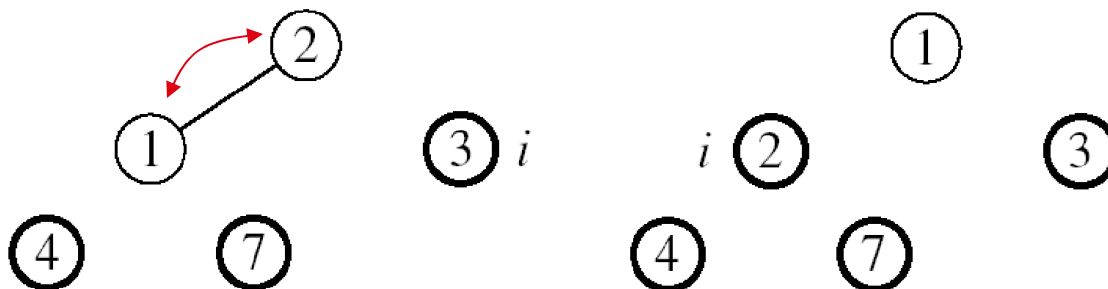
Max-Heapify($A, 1, 4$)



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 $A[1] \leftrightarrow A[i]$
 4. Max-Heapify(A, 1, $i - 1$) $O(\log n)$
- } $n-1$ lần lặp
- Thời gian tính: $O(n \log n)$
 - Có thể chứng minh thời gian tính là $\Theta(n \log n)$

Tổng kết

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

Phép toán

- Max-Heapify
- Build-Max-Heap
- Heap-Sort

Thời gian tính

$O(\log n)$

$O(n)$

$O(n \log n)$

QUESTIONS?

