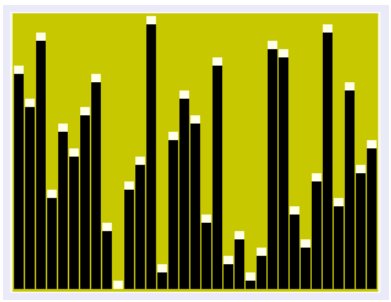
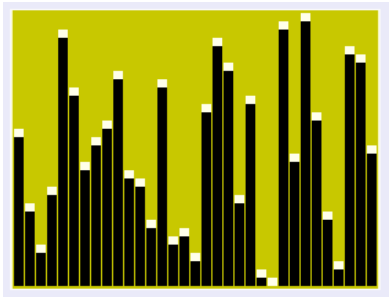
Chương 5
Sắp xếp (Sorting)



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!

NGUYỄN ĐỨC NGHĨA
Bộ môn KHMT - ĐHBKHN

NỘI DUNG

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 $A = (a_1, a_2, \ldots, a_n)$

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

$$a'_1 \leq \ldots \leq a'_n$$

- **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.

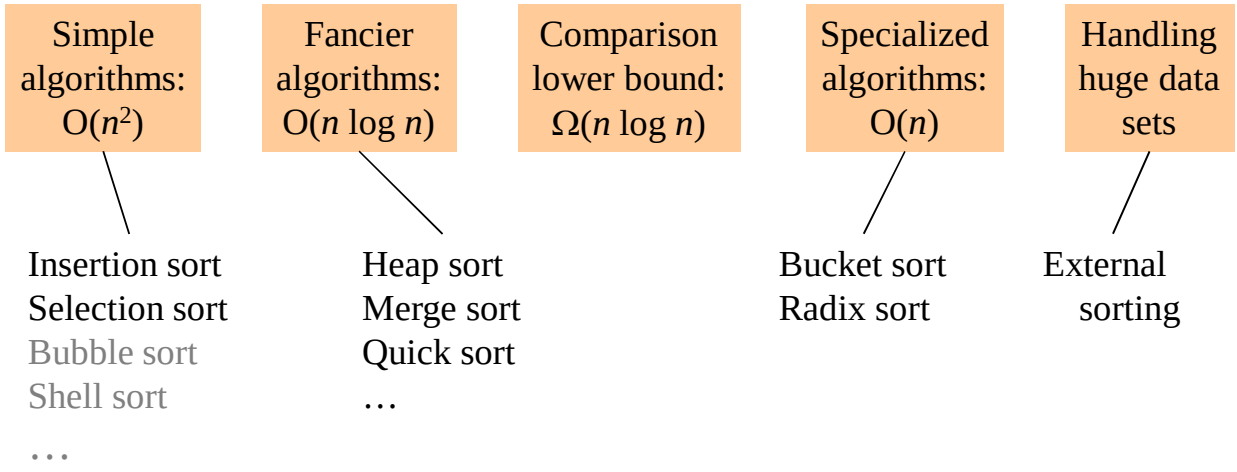
5.1.1. Bài toán sắp xếp

- 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*)

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



Các thuật toán sắp xếp dựa trên phép so sánh

Name	Average	Worst	Memory	Stable	Method
Bubble sort	—	O(n^2)	O(1)	Yes	Exchanging
Cocktail sort	—	O(n^2)	O(1)	Yes	Exchanging
Comb sort	O(n log n)	O(n log n)	O(1)	No	Exchanging
Gnome sort	—	O(n^2)	O(1)	Yes	Exchanging
Selection sort	O(n^2)	O(n^2)	O(1)	No	Selection
Insertion sort	O(n + d)	O(n^2)	O(1)	Yes	Insertion
Shell sort	—	O(n log^2 n)	O(1)	No	Insertion
Binary tree sort	O(n log n)	O(n log n)	O(n)	Yes	Insertion
Library sort	O(n log n)	O(n^2)	O(n)	Yes	Insertion
Merge sort	O(n log n)	O(n log n)	O(n)	Yes	Merging
In-place merge sort	O(n log n)	O(n log n)	O(1)	Yes	Merging
Heapsort	O(n log n)	O(n log n)	O(1)	No	Selection
Smoothsort	—	O(n log n)	O(1)	No	Selection
Quicksort	O(n log n)	O(n^2)	O(log n)	No	Partitioning
Introsort	O(n log n)	O(n log n)	O(log n)	No	Hybrid
Patience sorting	—	O(n^2)	O(n)	No	Insertion

Các thuật toán sắp xếp không dựa trên phép so sánh

Name	Average	Worst	Memory	Stable	$n << 2^k$?
Pigeonhole sort	$O(n+2^k)$	$O(n+2^k)$	$O(2^k)$	Yes	Yes
Bucket sort	$O(n \cdot k)$	$O(n^2 \cdot k)$	$O(n \cdot k)$	Yes	No
Counting sort	$O(n+2^k)$	$O(n+2^k)$	$O(n+2^k)$	Yes	Yes
LSD Radix sort	$O(n \cdot k/s)$	$O(n \cdot k/s)$	$O(n)$	Yes	No
MSD Radix sort	$O(n \cdot k/s)$	$O(n \cdot (k/s) \cdot 2^s)$	$O((k/s) \cdot 2^s)$	No	No
Spreadsort	$O(n \cdot k/\log(n))$	$O(n \cdot (k-\log(n))^{0.5})$	$O(n)$	No	No

Các thông số trong bảng

n - số phần tử cần sắp xếp

k - kích thước mỗi phần tử

s - kích thước bộ phận được sử dụng khi cài đặt.

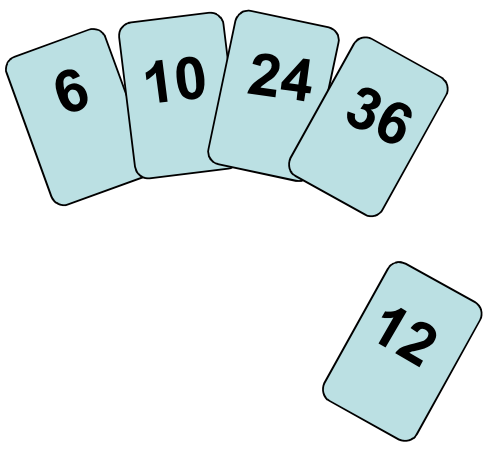
Nhiều thuật toán được xây dựng dựa trên giả thiết là $n << 2^k$.

NỘI DUNG

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)

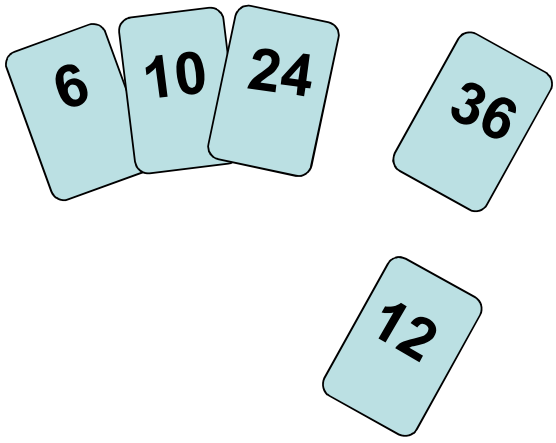
5.2.1. Sắp xếp chèn (Insertion Sort)



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.

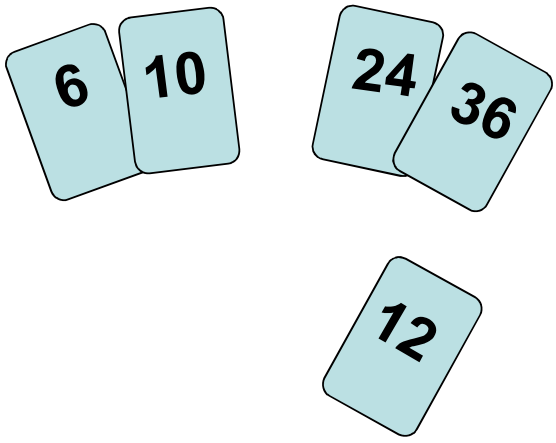
Sắp xếp chèn (Insertion Sort)



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.

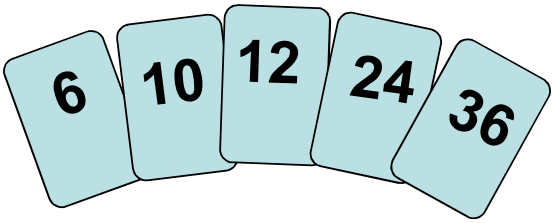
Sắp xếp chèn (Insertion Sort)



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.

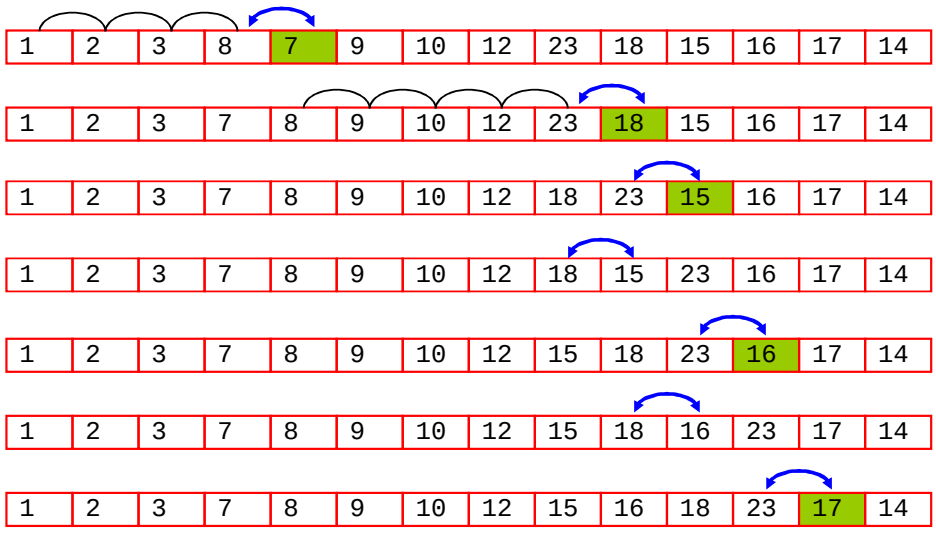
5.2.1. Sắp xếp chèn (Insertion Sort)

- **Thuật toán:**
  - Tại bước $k = 1, 2, ..., n$, 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.

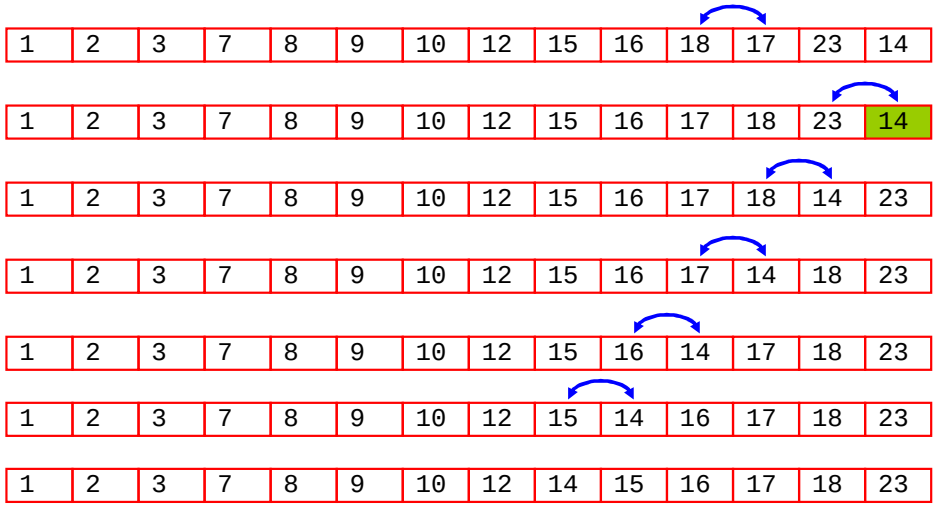
```
void insertionSort(int a[], int array_size) {
  int i, j, last;
  for (i=1; i < array_size; i++)  {
      last = a[i];
      j = i;
      while ((j > 0) && (a[j-1] > last))   {
            a[j] = a[j-1];
            j = j - 1;  }
      a[j] = last;
  } // end for
} // end of isort
```

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 (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;
}
```

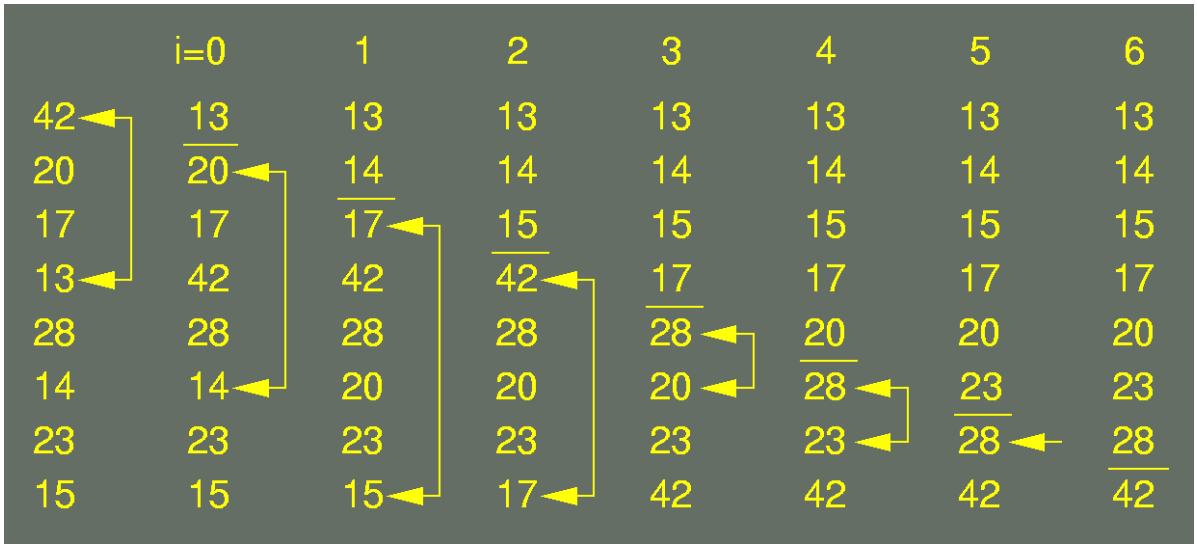
```
void selectionSort(int a[], int n){
  int i, j, min, temp;
  for (i = 0; i < n-1; i++) {
    min = i;
    for (j = i+1; j < n; j++){
       if (a[j] < a[min]) min = j;
    }
    swap(a[i], a[min]);
  }
}
```


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



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 hoạ 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.

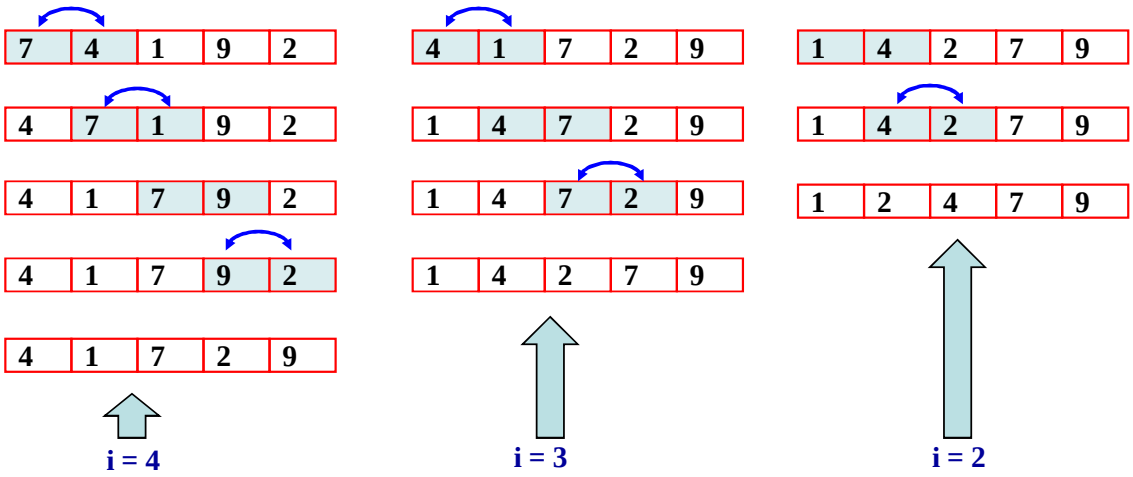
5.2.3. Sắp xếp nổi bọt - Bubble Sort

```
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.

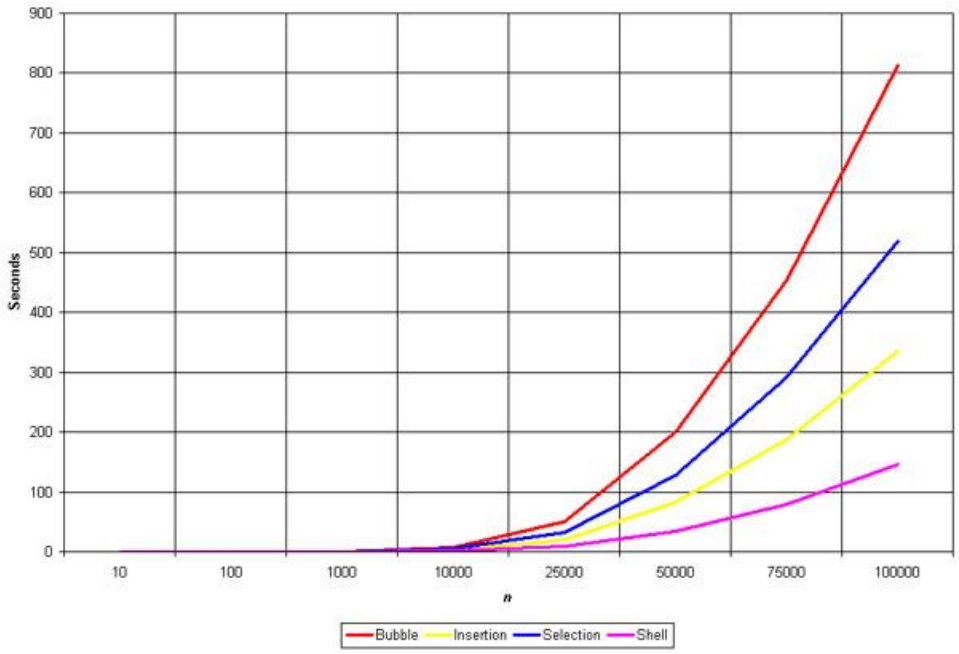
Ví dụ: Bubble Sort



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)

5.6. Cận dưới cho độ phức tạp tính toán của bài toán sắp xếp

5.7. Các phương pháp sắp xếp đặc biệt

Sắp xếp trộn (Merge Sort)

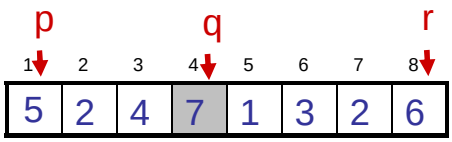
Hoà nhập hai dòng xe theo Khoá [độ liều lĩnh của lái xe].
Lái xe liều lĩnh hơn sẽ vào trước!

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)



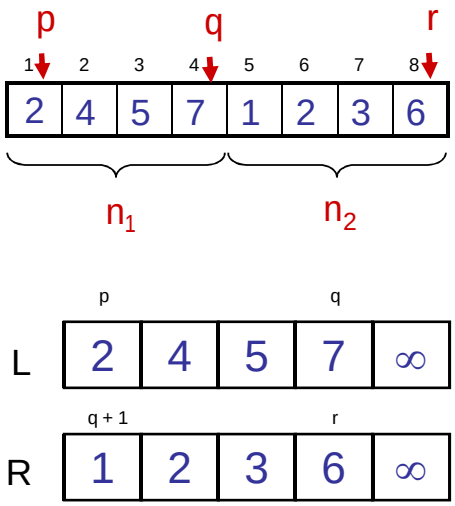

```
if p < r                          ▷ Kiểm tra điều kiện neo
  then q ← ⌊(p + r)/2⌋            ▷ Chia (Divide)
       MERGE-SORT(A, p, q)        ▷ Trị (Conquer)
       MERGE-SORT(A, q + 1, r)    ▷ Trị (Conquer)
       MERGE(A, p, q, r)          ▷ Tổ hợp (Combine)
endif
```

- 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 $L[1 . . n_1]$ và n_2 phần tử tiếp theo vào $R[1 . . n_2]$
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

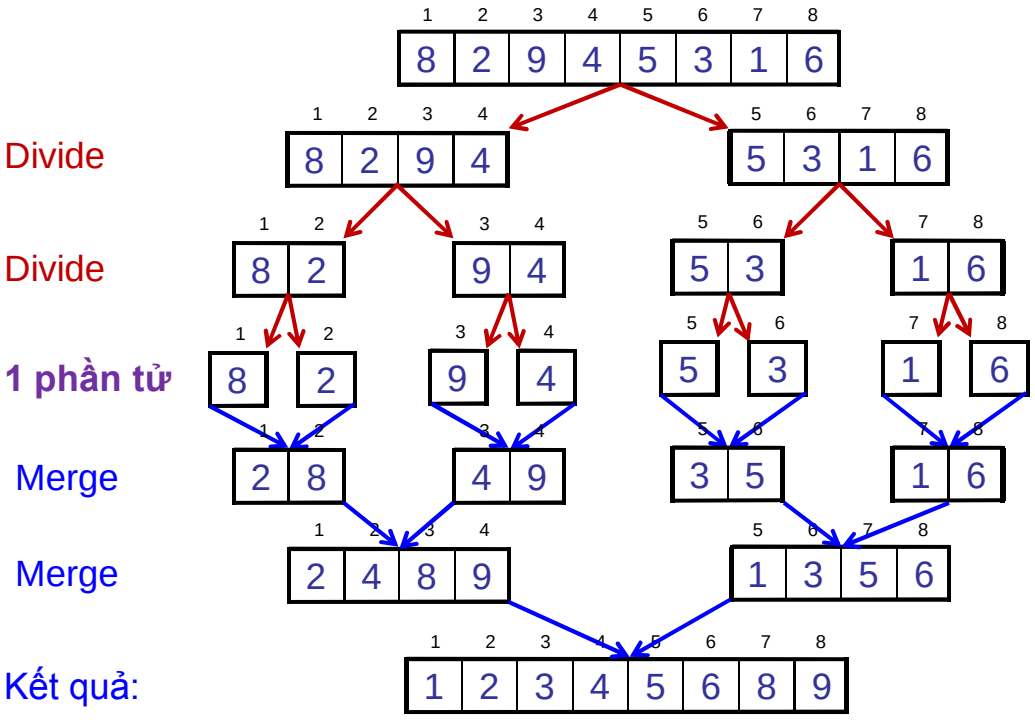
- **Chia:**
  - tính q như là giá trị trung bình của p và r: D(n) = Θ(1)
- **Trị:**
  - giải đệ qui 2 bài toán con, mỗi bài toán kích thước n/2 $\Longrightarrow$ 2T (n/2)
- **Tổ hợp:**
  - TRỘN (MERGE) trên các mảng con cỡ n phần tử đòi hỏi thời gian Θ(n)

    $\Longrightarrow$ C(n) = Θ(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

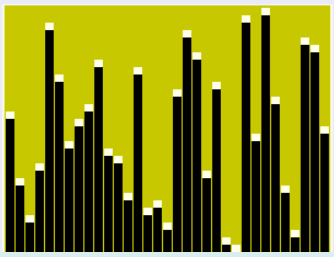
5.4. Sắp xếp nhanh (Quick Sort)

- 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

10 Algorithms in 20th Century

có ảnh hưởng sâu rộng đến việc phát triển và ứng dụng của khoa học kỹ thuật …

1946: The Metropolis Algorithm for Monte Carlo
1947: Simplex Method for Linear Programming // Tính toán khoa học
1950: Krylov Subspace Iteration Method
1951: The Decompositional Approach to Matrix Computations
1957: The Fortran Optimizing Compiler
1959: QR Algorithm for Computing Eigenvalues
1962: Quicksort Algorithms for Sorting
1965: Fast Fourier Transform // Xử lý ảnh
1977: Integer Relation Detection
1987: Fast Multipole Method

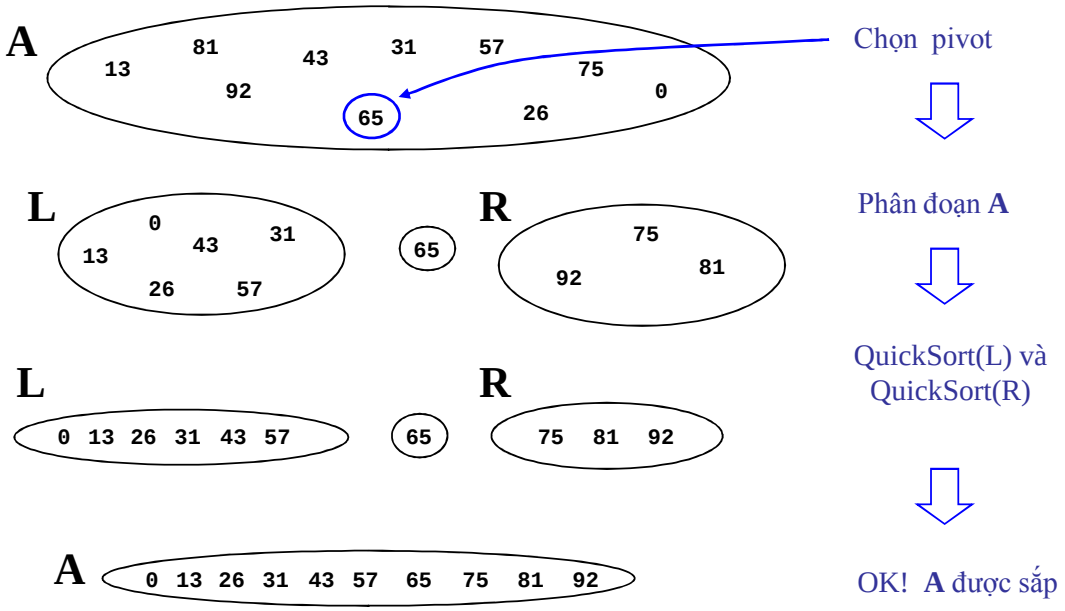
Computing in Science & Engineering, January/February 2000

Science, Vol. 287, No. 5454, p. 799, February 2000

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* = 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 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* = 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 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, 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*] ≤ *a*[*jpivot*], với mọi *left* ≤ *i* < *pivot*,
  - *a*[*j*] ≥ *a*[*jpivot*]), với mọi *pivot* < *j* ≤ *right*.

Phần tử chốt là phần tử đứng đầu

Partition(a, left, right)

i = *left*; *j* = *right* + 1; *pivot* = a[*left*];

pivot được chọn là phần tử đứng đầu

while *i* < *j* **do**

i = *i* + 1;

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

j = *j* – 1;

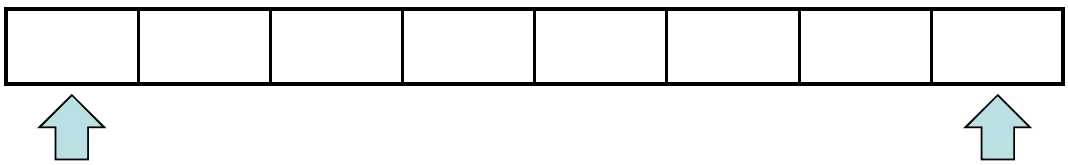
while *j* ≥ *left* **and** a[*j*] > *pivot* **do** *j* = *j* – 1;

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

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

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

return *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*] > *pivot* và a[*j*] < *pivot*

j

Ví dụ

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

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

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*;

pivot được chọn là phần tử đứng giữa

- 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);}
}
```

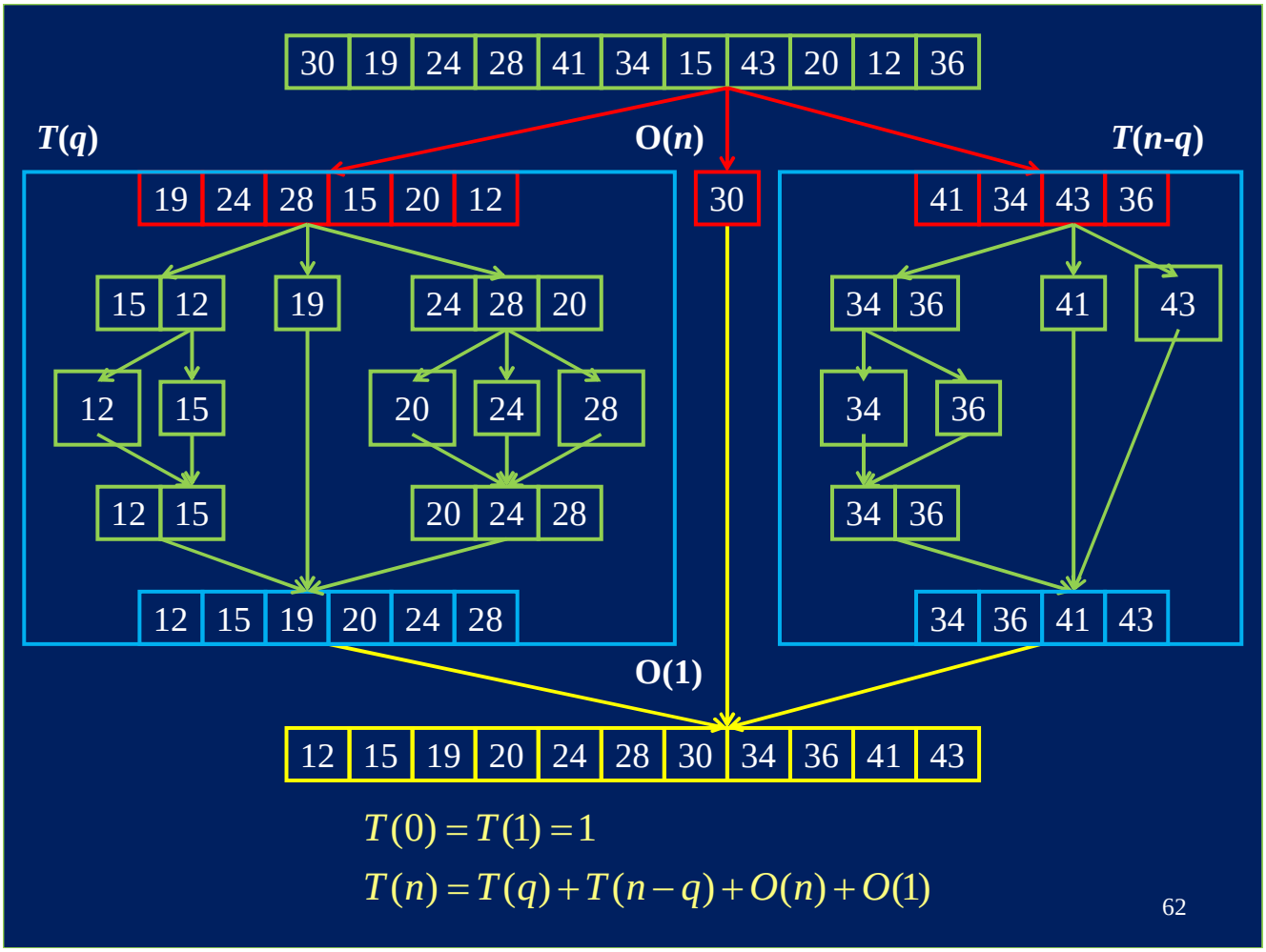

Cài đặt Quick Sort (dễ đọc hơn?)

```
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();
}
```

Thời gian tính của Quick-Sort

- Thời gian tính của Quick-Sort phụ thuộc vào việc phép phân chia là ***cân bằng* (*balanced*)** hay ***không cân bằng* (*unbalanced*),** và điều đó lại phụ thuộc vào việc phần tử nào được chọn làm chốt.
  1. **Phân đoạn không cân bằng:** thực sự không có phần nào cả, do đó một 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, như vậy 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, nghĩa là một bài toán con có kích thước $n - k$ còn bài toán kia có kích thước k.

Ta sẽ xét từng tình huống!

Phân đoạn không cân bằng
(Unbalanced partition)

Công thức đệ qui là:

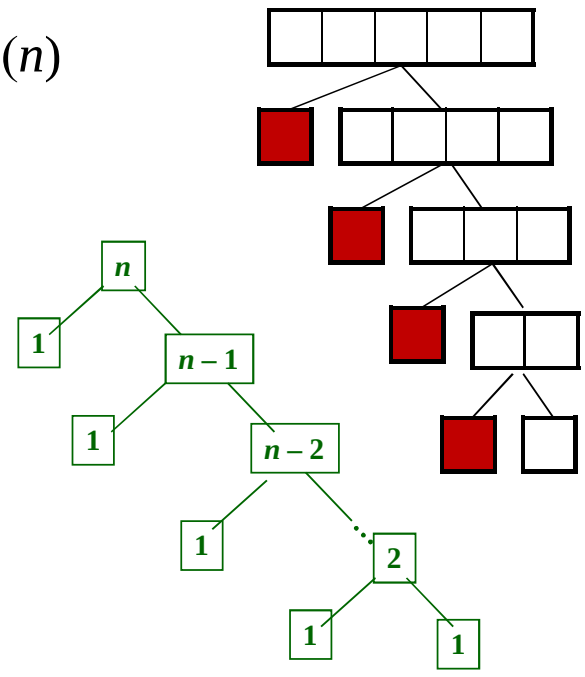
$$T(n) = T(n-1) + T(0) + \Theta(n)$$

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

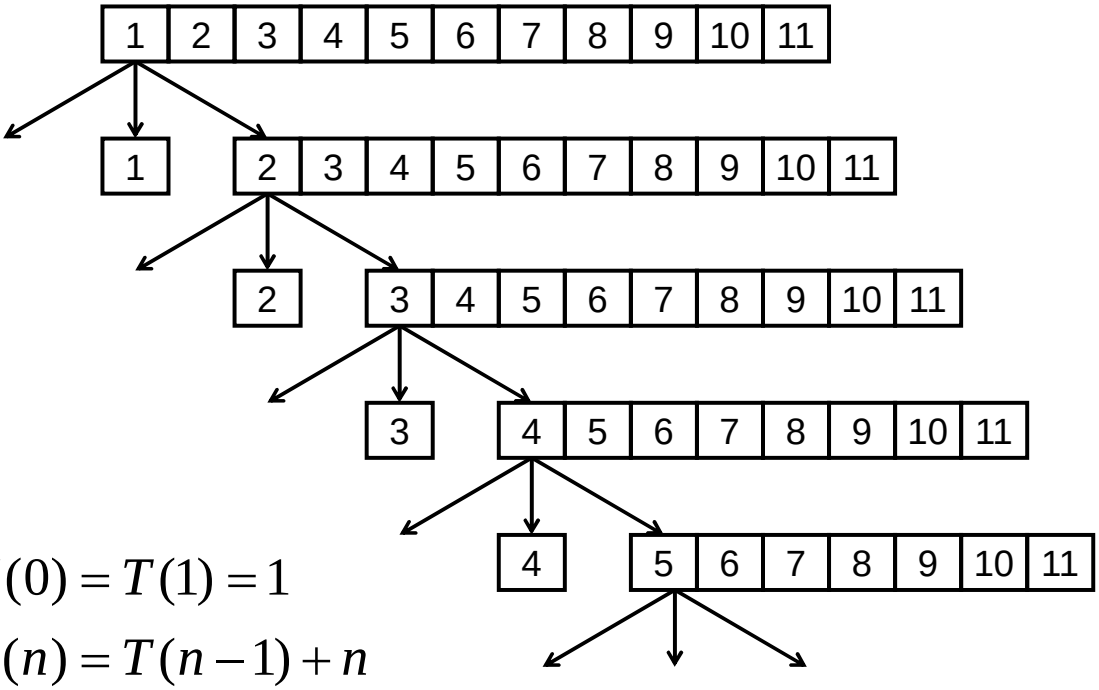
$$\left.\begin{aligned} T(n) &= T(n-1) + n \\ T(n-1) &= T(n-2) + (n-1) \\ T(n-2) &= T(n-3) + (n-2) \\ &\dots \\ T(2) &= T(1) + (2) \end{aligned}\right\}$$

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

$$= O(n^2)$$



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



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

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

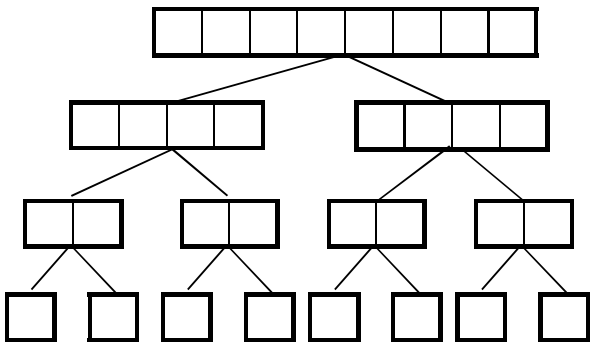
Phân đoạn hoàn hảo - Perfect partition

Công thức đệ qui:

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

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

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



It's Best Case!

Theo định lý thợ !

Phân tích tình huống tồi nhất

Ta có công thức đệ qui:

$$T(n) = \max_{1 \le q < n} \{ T(q) + T(n-q) \} + \Theta(n)$$

Mệnh đề: $T(n) \le cn^2 = O(n^2)$

Chứng minh: Qui nạp theo n.

- *Base case:* Rõ ràng bổ đề đúng cho n=1.
- *Induction step:* Giả sử bổ đề đúng với mọi $n < n'$, ta chứng minh nó đúng cho n'. Ta có:

$$\begin{aligned} T(n') &= \max_{1 \le q < n'} \{ T(q) + T(n'-q) \} + \Theta(n') \\ &\le cq^2 + c(n'-q)^2 + dn' \qquad \text{(giả thiết qui nạp)} \\ &= 2cq^2 + c(n')^2 - 2cqn' + dn' \end{aligned}$$

Phân tích tình huống tồi nhất

- Để hoàn thành chứng minh, ta cần chỉ ra rằng
  $2cq^2 + c(n')^2 - 2cqn' + dn' \le c(n')^2$,
- và bất đẳng thức này là tương đương với:
  $dn' \le 2cq(n' - q)$
- Do $q(n'-q)$ luôn lớn hơn $n'/2$ (để thấy điều này là đúng có thể xét hai tình huống: $q < n/2$ và $n \ge q \ge n/2$), nên ta có thể chọn c đủ lớn để bất đẳng thức cuối cùng là đúng.
- Mệnh đề được chứng minh.
- Do trong tình huống phân đoạn không cân bằng QS có thời gian tính $\Theta(n^2)$, nên từ mệnh đề suy ra thời gian tính trong tình huống tồi nhất của QS là $O(n^2)$.

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 =

  Σ (thời gian phân đoạn kích thước i)×(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

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

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

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

- Giải công thức đệ qui này ta thu được

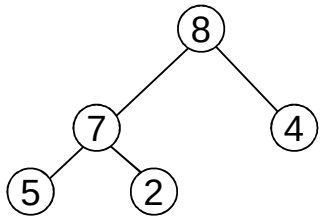
$$E(T(N)) = O(N \log N).$$

NỘI DUNG

5.5. Sắp xếp vun đống (Heap Sort)

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 $length[A]$. Số lượng phần tử là $heapsize[A]$



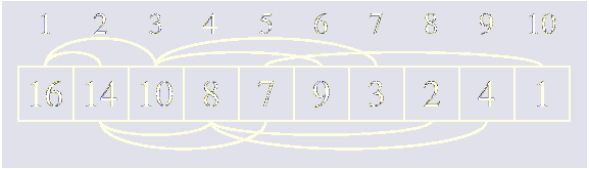
Heap

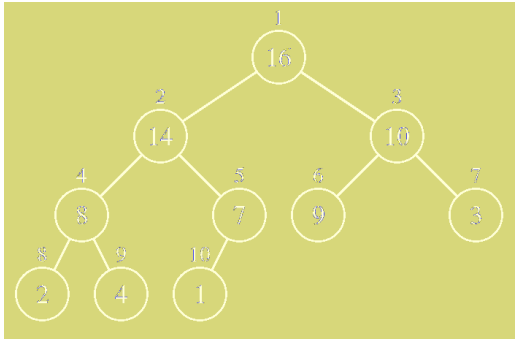
Từ tính chất đống suy ra:
"Gốc chứa phần tử lớn nhất của đống!"

Như vậy có thể nói:
Đống là cây nhị phân được điền theo thứ tự

Biểu diễn đống bởi mảng

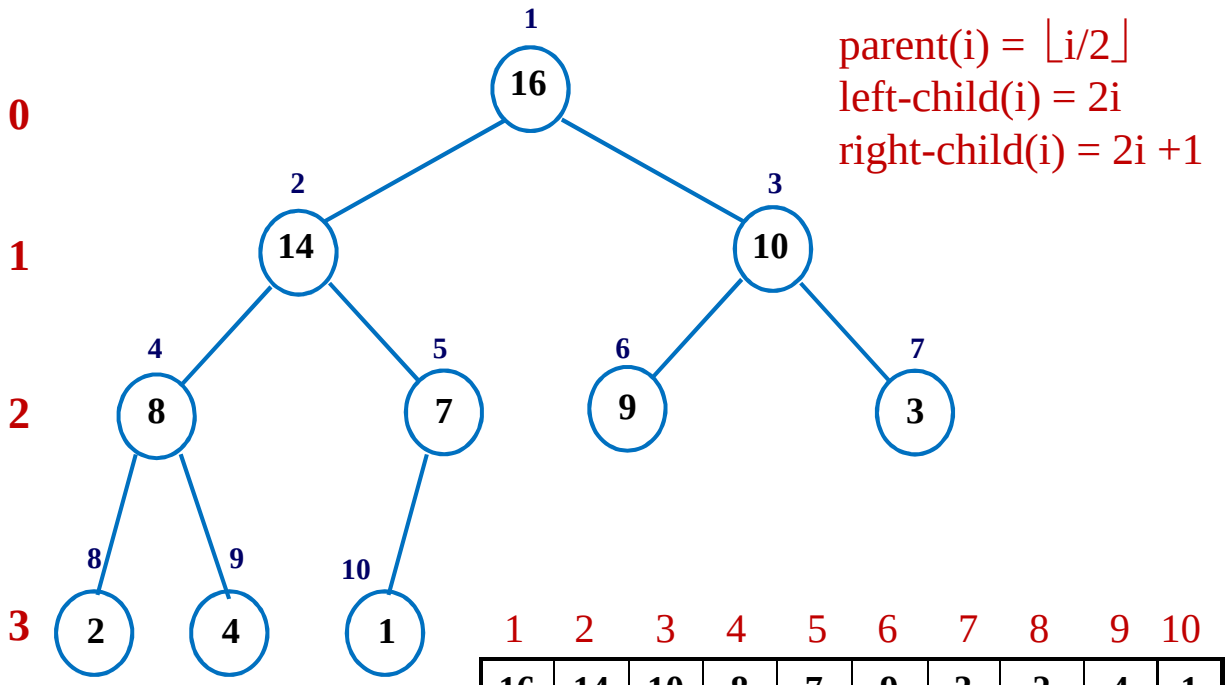
- Đống có thể cất giữ trong mảng A.
  - Gốc của cây là A[1]
  - Con trái của A[i] là A[2*i]
  - Con phải của A[i] là A[2*i + 1]
  - Cha của A[i] là A[$\lfloor i/2 \rfloor$]
  - Heapsize[A] $\leq$ length[A]
- Các phần tử trong mảng con A[($\lfloor n/2 \rfloor$+1) .. n] là các lá





parent(i) = $\lfloor i/2 \rfloor$
left-child(i) = 2i
right-child(i) = 2i +1

Ví dụ đống



parent(i) = ⌊i/2⌋
left-child(i) = 2i
right-child(i) = 2i +1

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

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:

    $A[parent(i)] \geq A[i]$

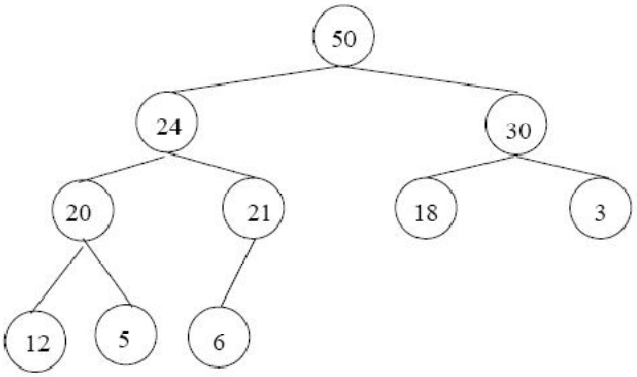
- **Đống min - Min-heaps** (phần tử nhỏ nhất ở gốc), có tính chất *min-heap*:
  - với mọi nút i, ngoại trừ gốc:

    $A[parent(i)] \leq A[i]$
- 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)



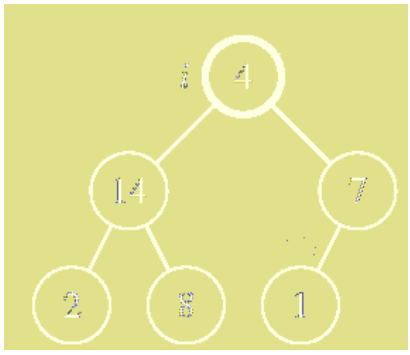
Các phép toán đối với đống

Operations on Heaps

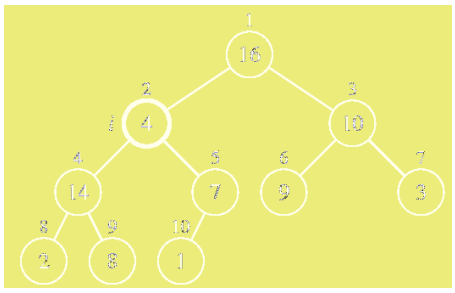
- Khôi phục tính chất max-heap (Vun lại đống)
  - Max-Heapify
- Tạo max-heap từ một mảng không được sắp xếp
  - Build-Max-Heap

Khôi phục tính chất đống

- 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

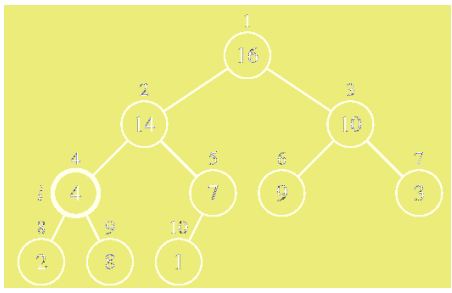


Ví dụ



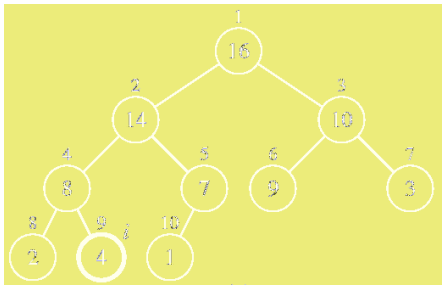
A[2] vi phạm tính chất đống

A[2] ↔ A[4]



A[4] vi phạm

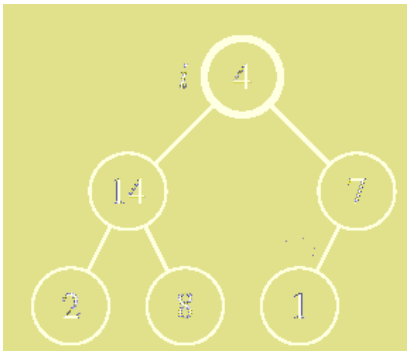
A[4] ↔ A[9]



Tính chất đống được khôi phục

Thuật toán khôi phục tính chất đống

- 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* = *heapsize*[*A*]
1. *l* ← *left-child*(*i*)
2. *r* ← *right-child*(*i*)
3. **if** (*l* ≤ *n*) and (*A*[*l*] > *A*[*i*])
4. **then** *largest* ← *l*
5. **else** *largest* ← *i*
6. **if** (*r* ≤ *n*) and (*A*[*r*] > *A*[*largest*])
7. **then** *largest* ← *r*
8. **if** *largest* != *i*
9. **then** Exchange(*A*[*i*],*A*[*largest*])
10. Max-Heapify(*A*,*largest*,*n*)

Thời gian tính của MAX-HEAPIFY

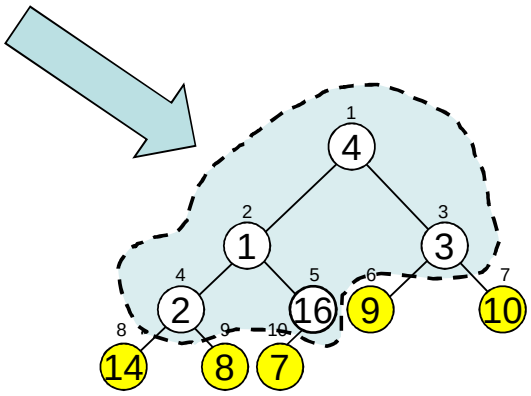
- Ta nhận thấy rằng:
  - Từ nút i phải di chuyển theo đường đi xuống phía dưới của cây. Độ dài của đường đi này không vượt quá độ dài đường đi từ gốc đến lá, nghĩa là khôngvượt quá *h*.
  - Ở mỗi mức phải thực hiện 2 phép so sánh.
  - Do đó tổng số phép so sánh không vượt quá 2*h*.
  - Vậy, thời gian 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 … n] thành max-heap (n = length[A])
- Vì các phần tử của mảng con A[(⌊n/2⌋+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 ⌊n/2⌋

Alg: Build-Max-Heap(A)

1. n = length[A]
2. **for** i ← ⌊n/2⌋ **downto** 1
3. **do** Max-Heappify(A, i, n)

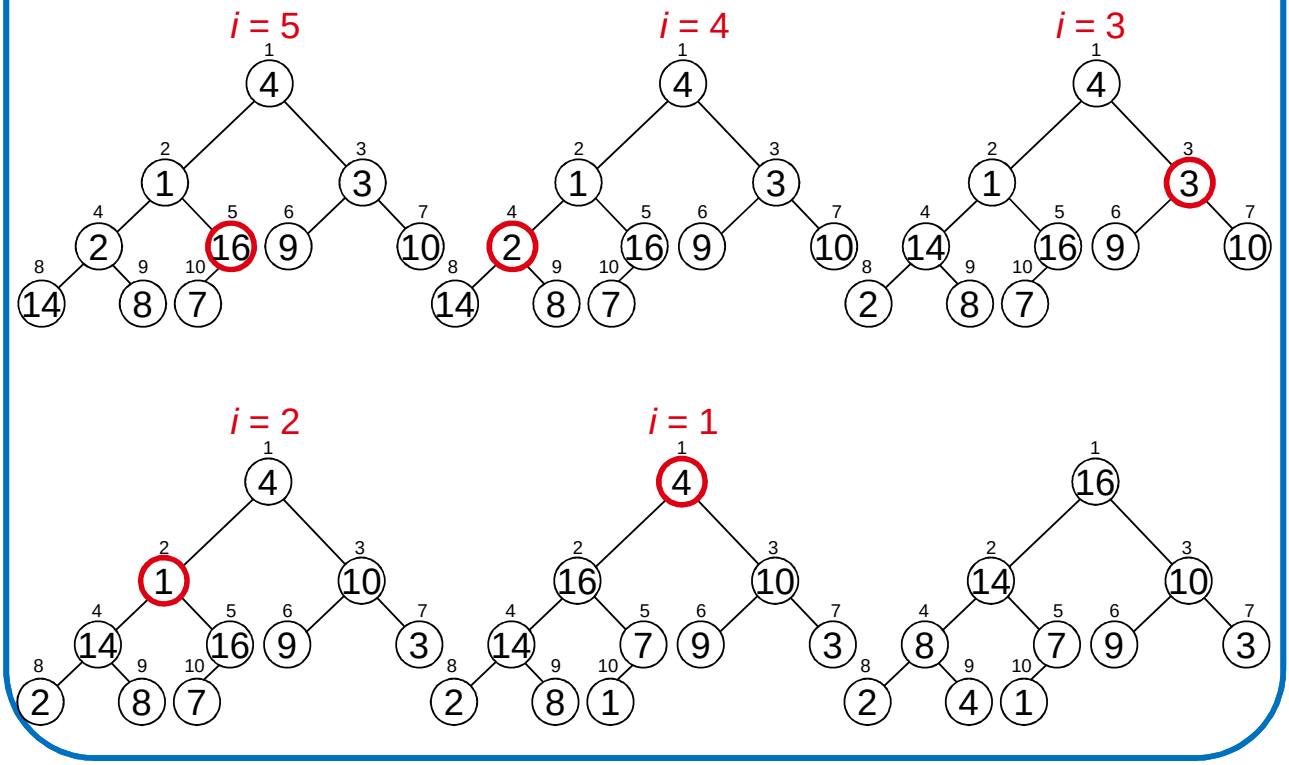


A:

4	1	3	2	16	9	10	14	8	7

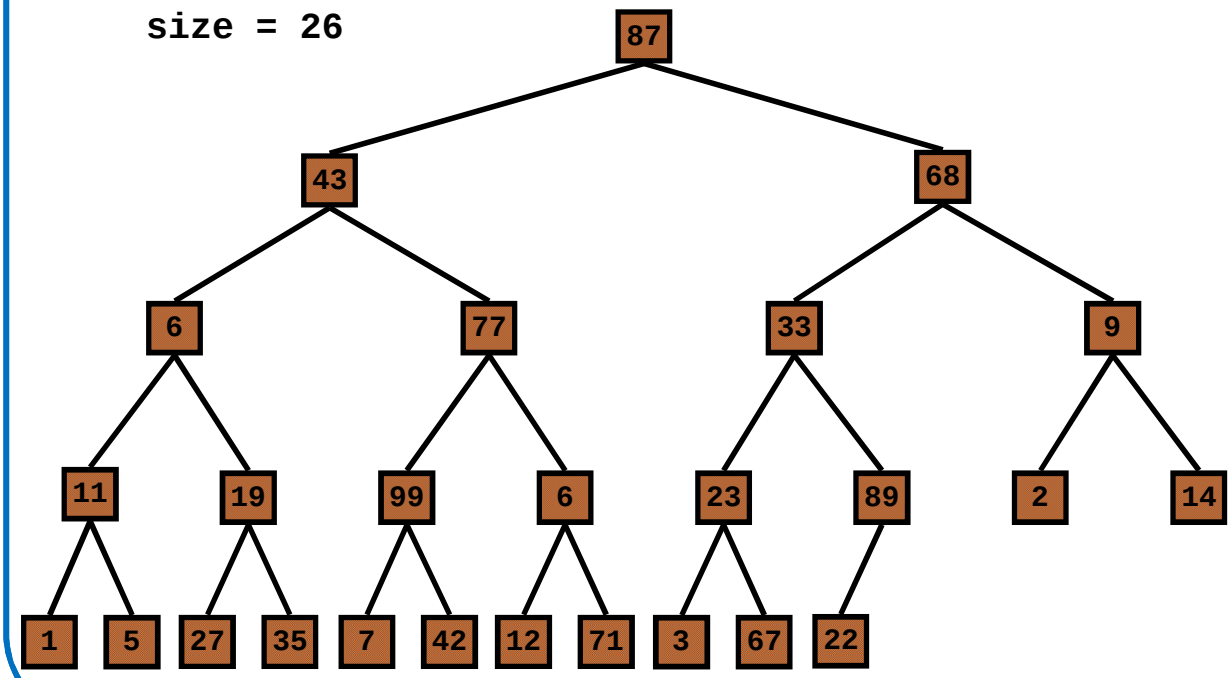
Ví dụ: **A:**

4	1	3	2	16	9	10	14	8	7



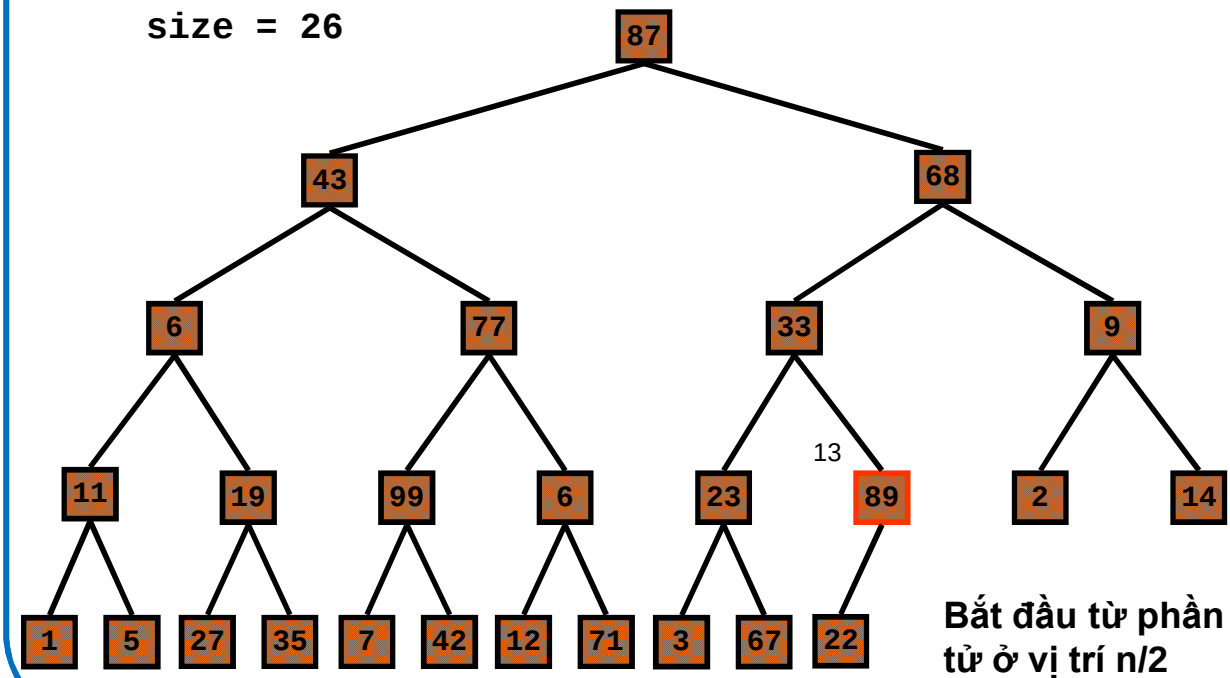
Ví dụ: Build-Min-Heap

size = 26



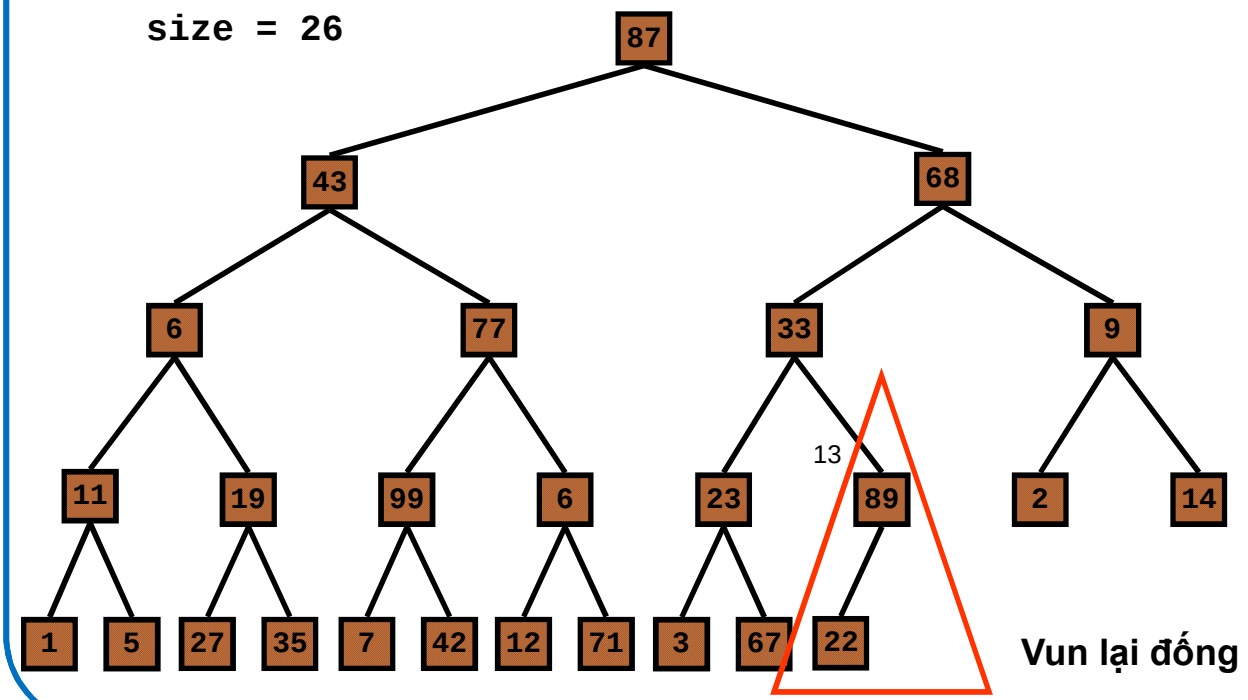
Ví dụ: Build-Min-Heap

size = 26

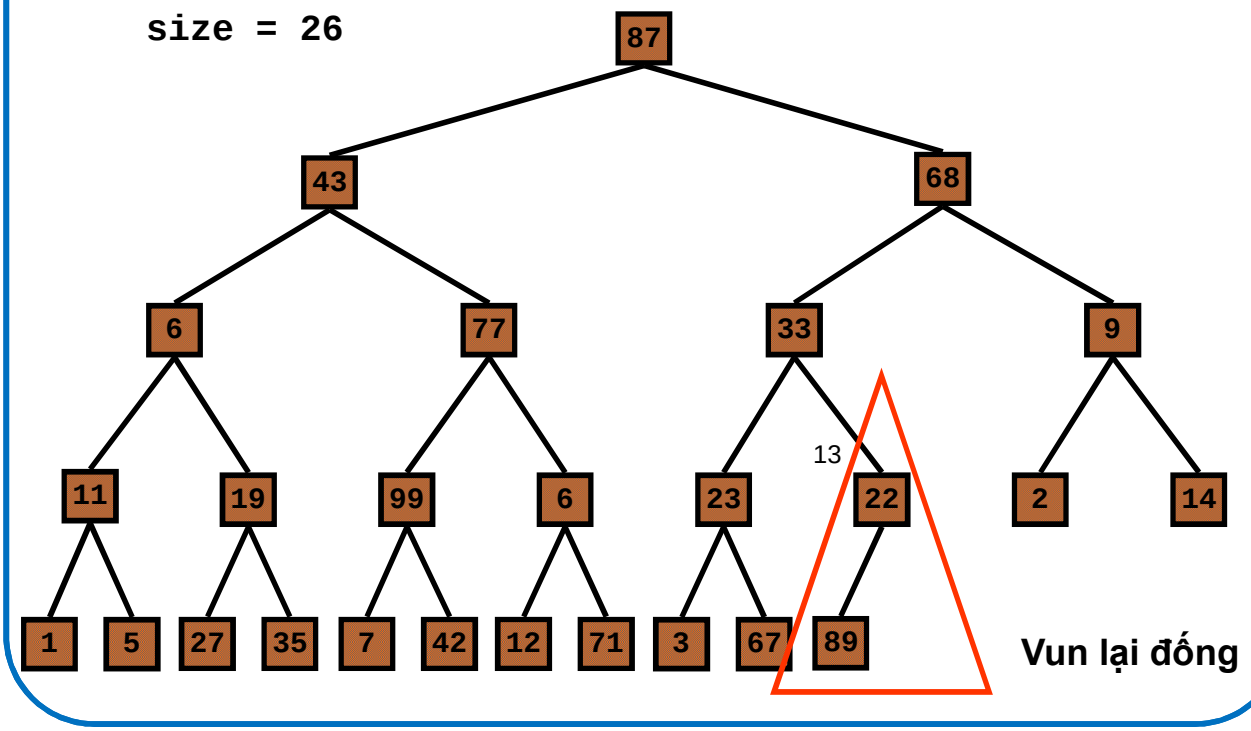


Bắt đầu từ phần tử ở vị trí n/2

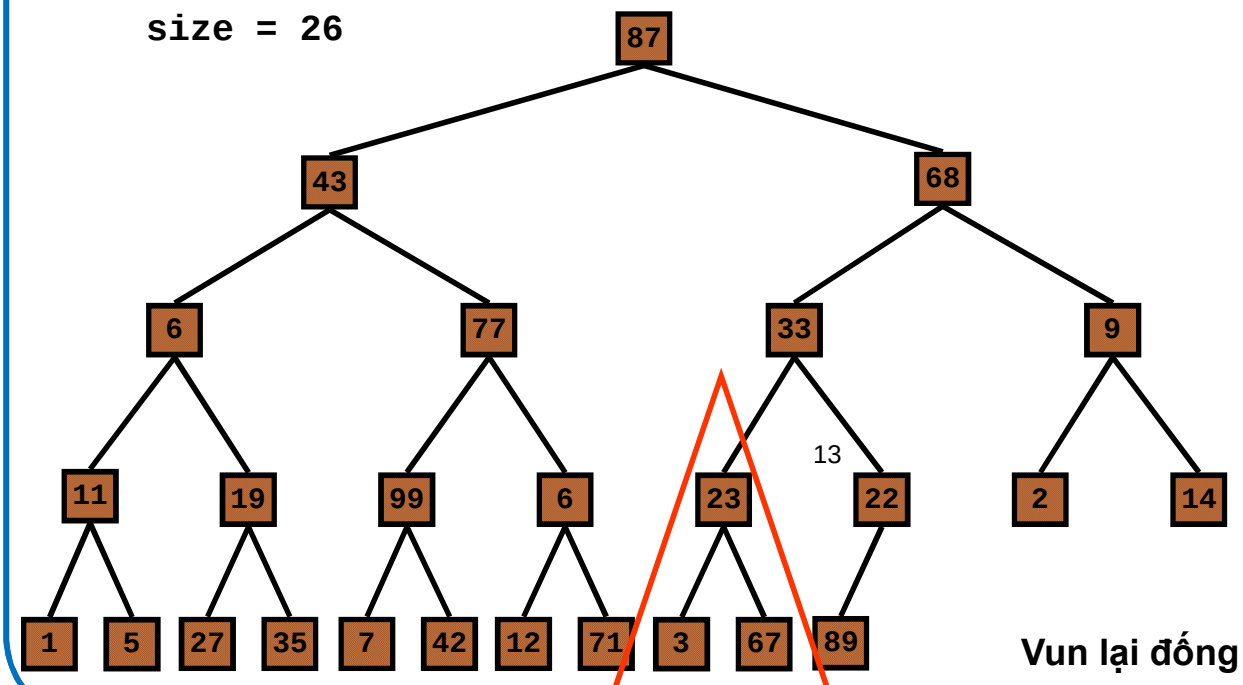
Build-Min-Heap



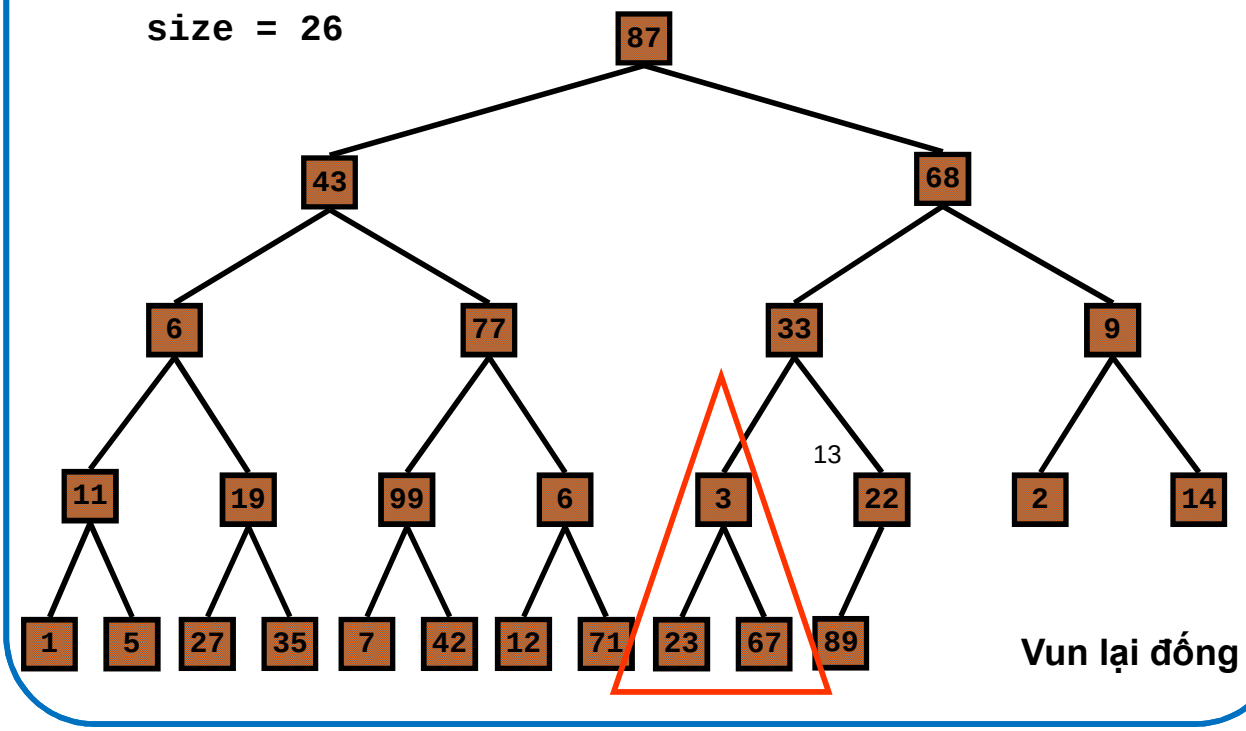
Build-Min-Heap



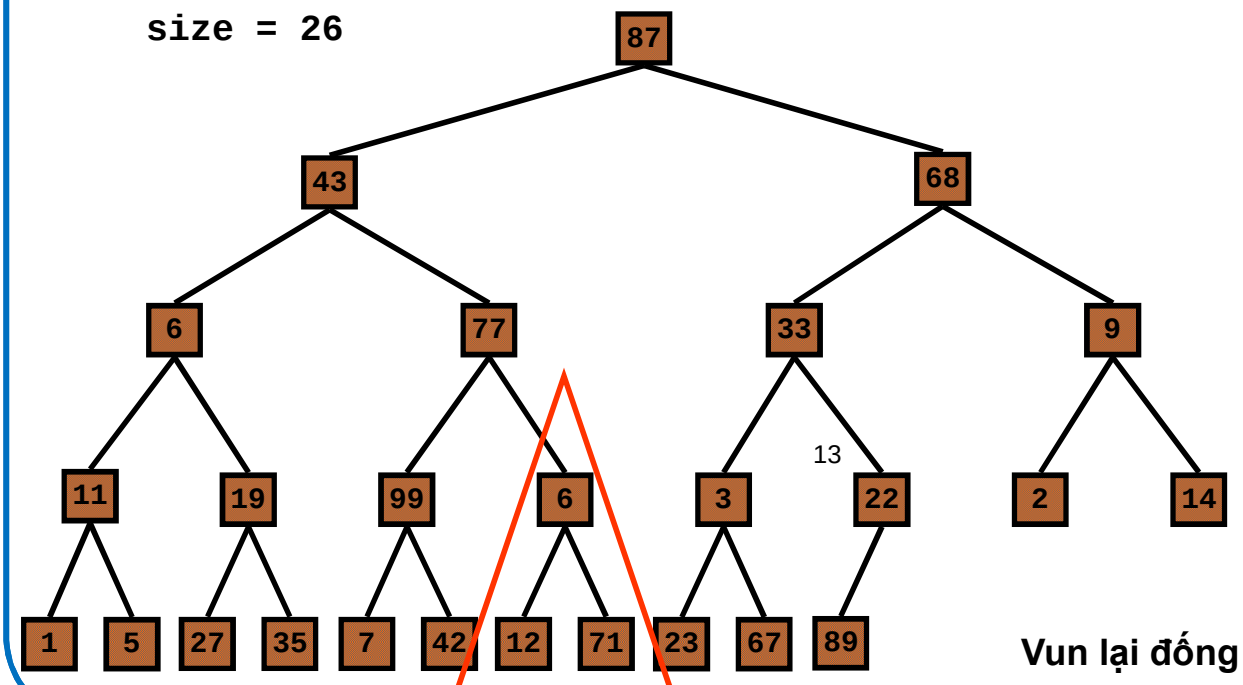
Build-Min-Heap



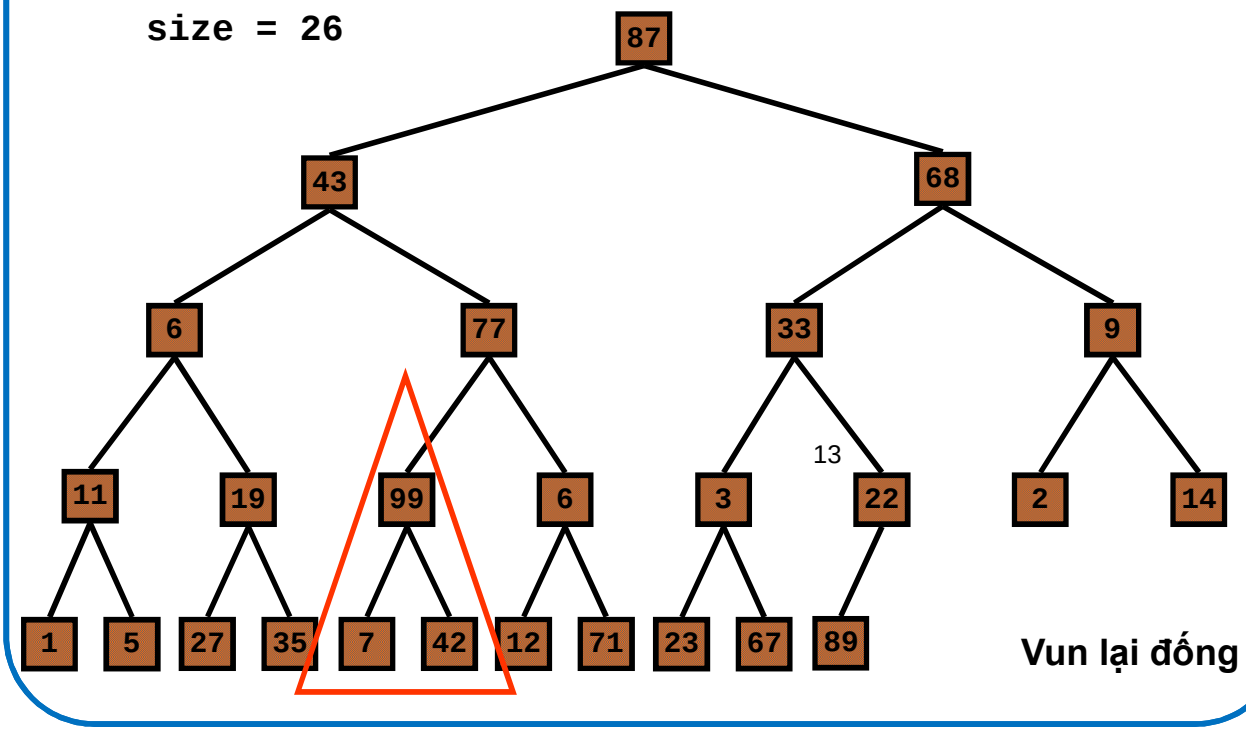
Build-Min-Heap



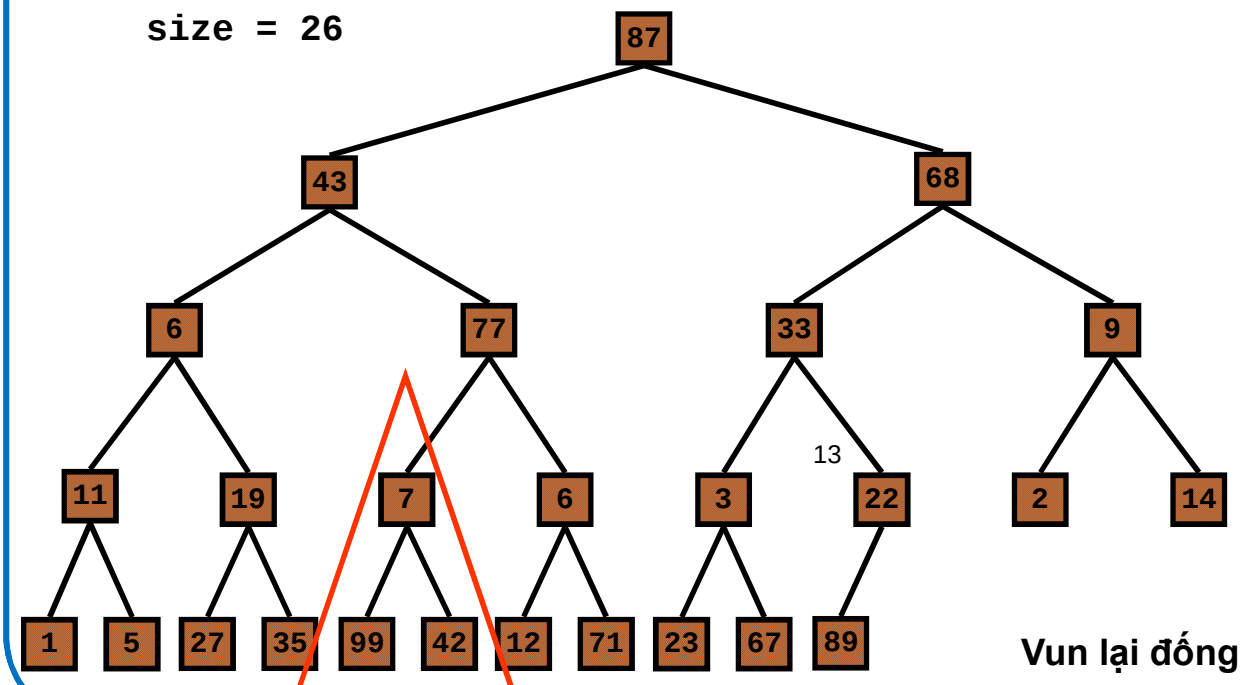
Build-Min-Heap



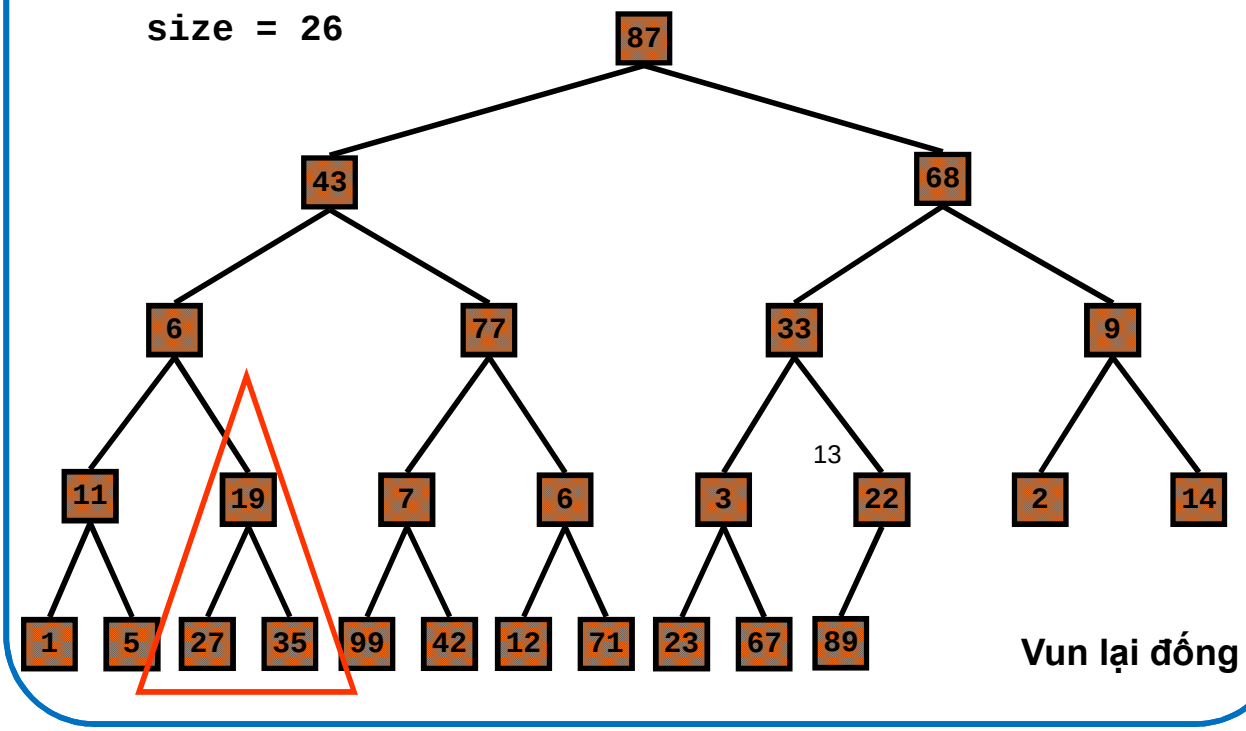
Build-Min-Heap



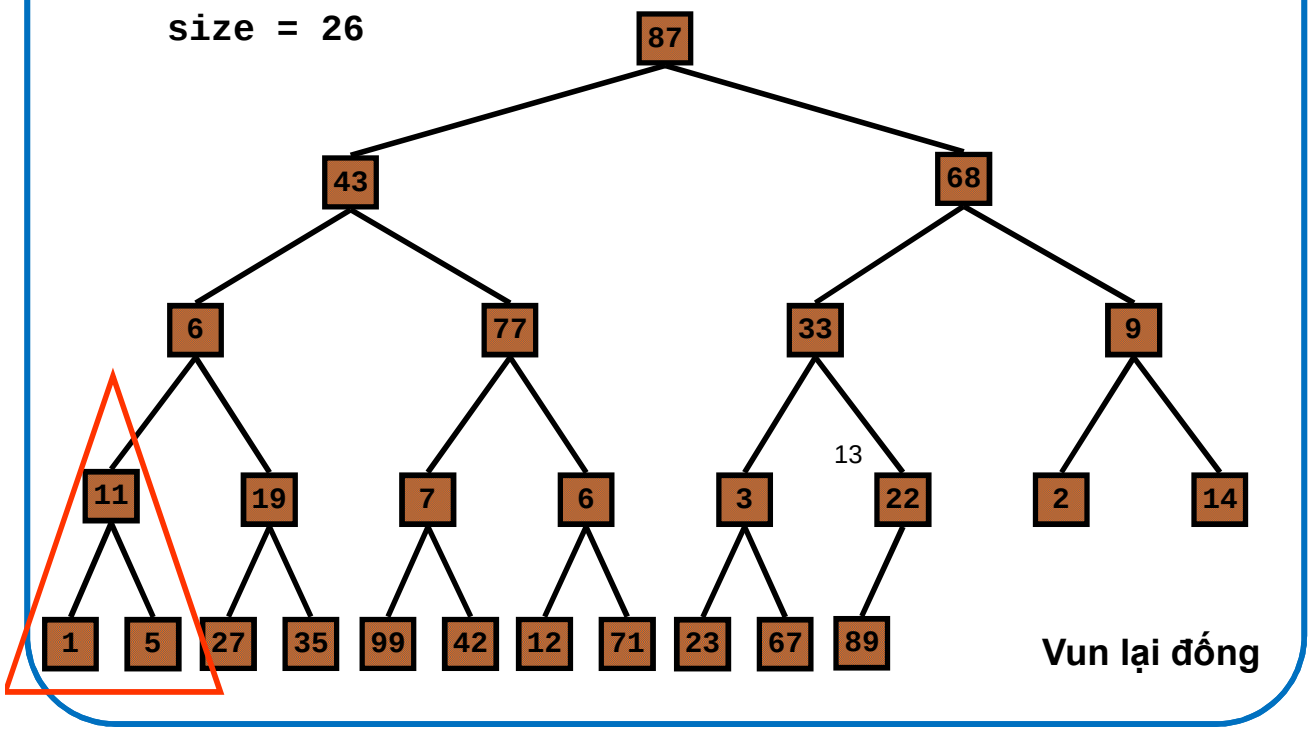
Build-Min-Heap



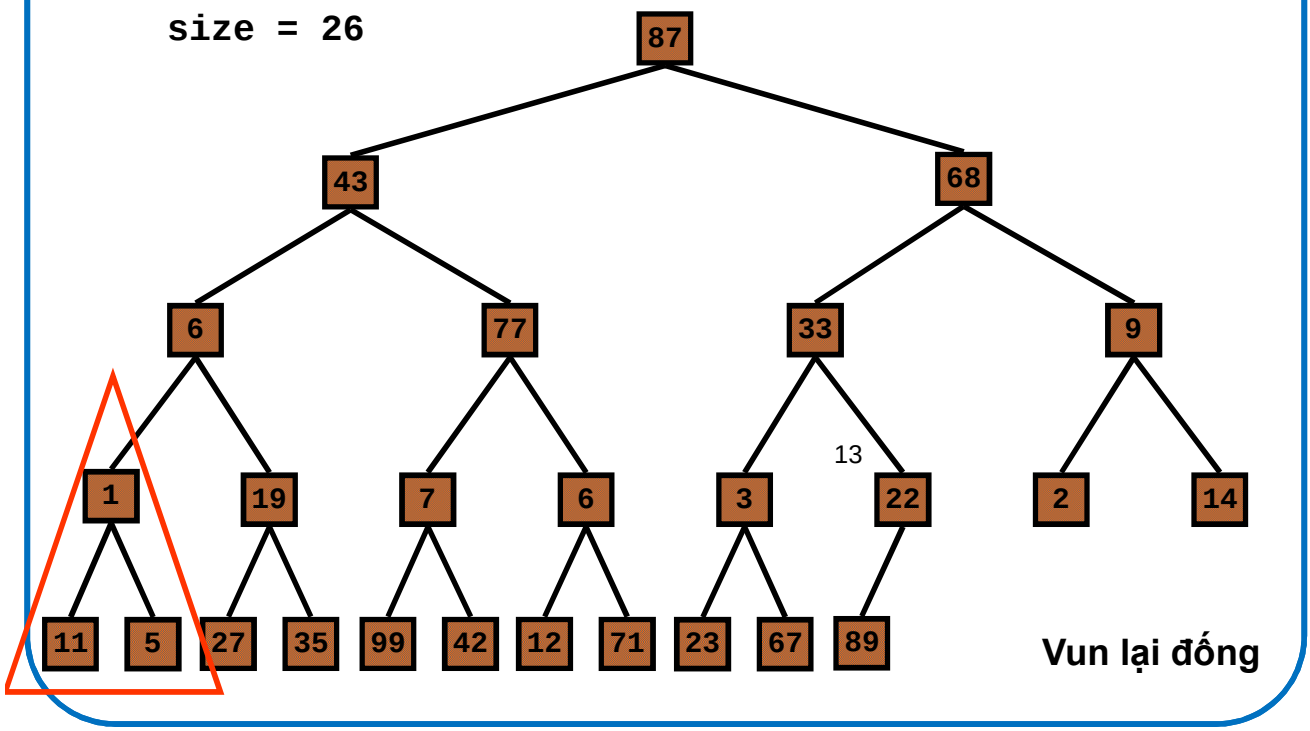
Build-Min-Heap



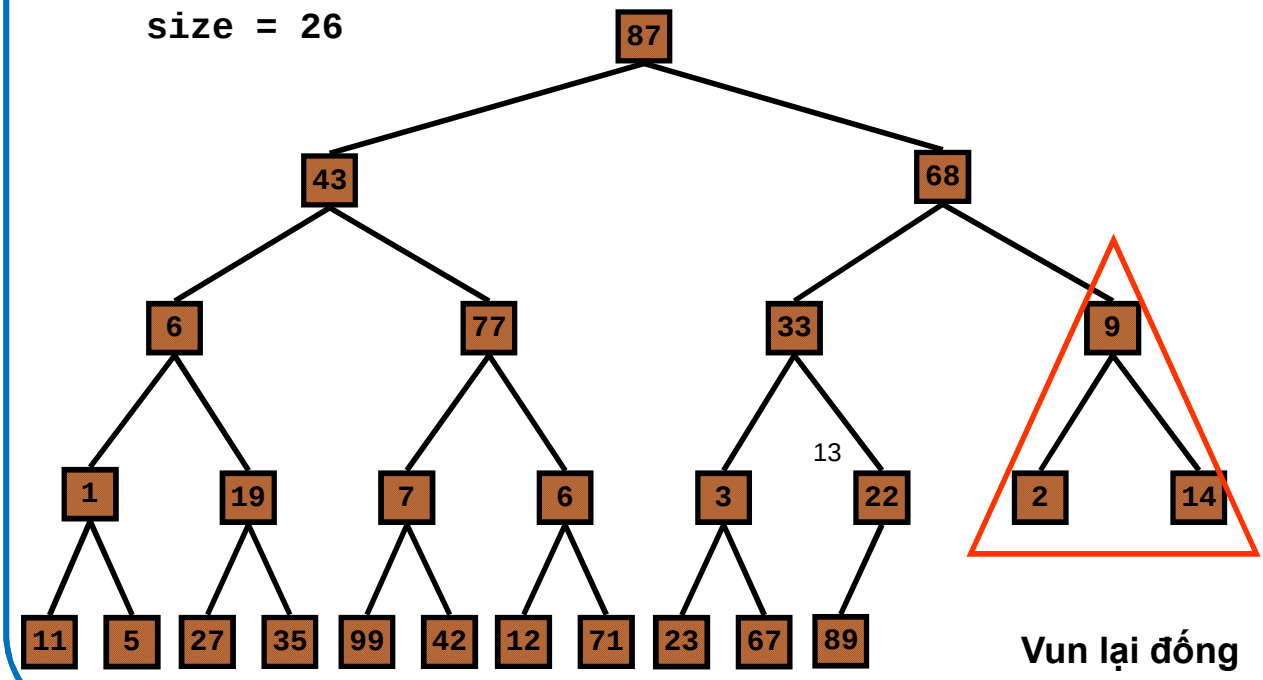
Build-Min-Heap



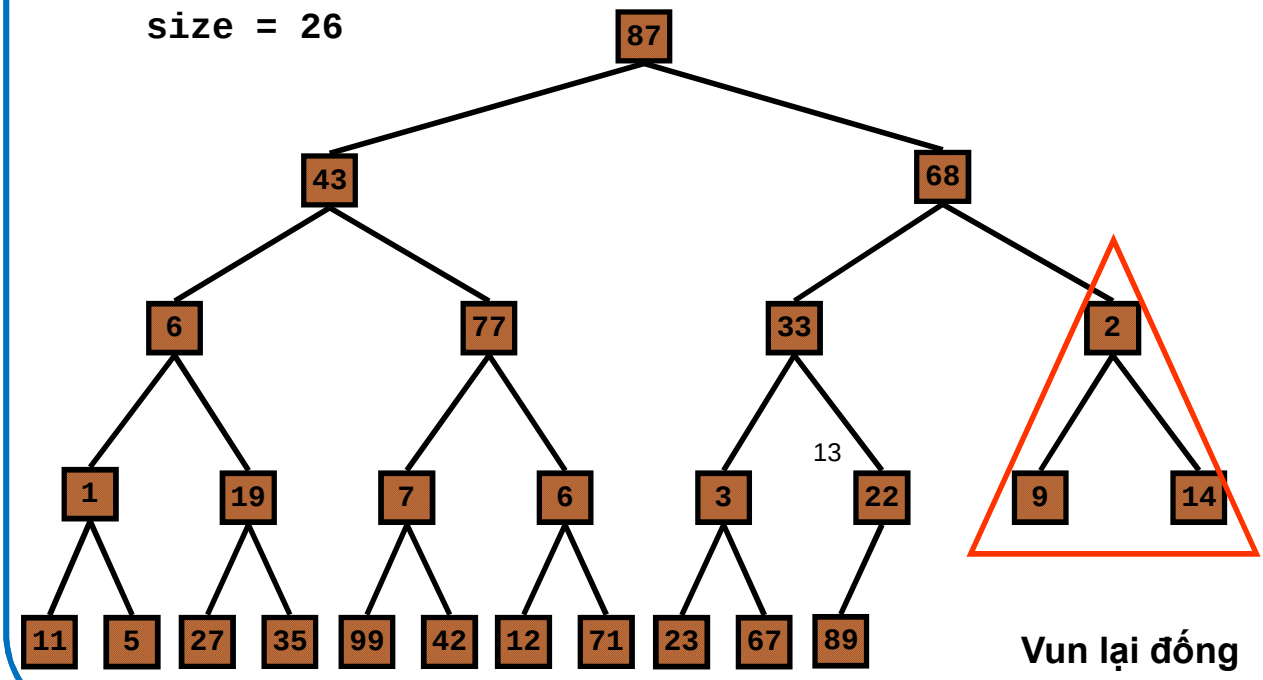
Build-Min-Heap



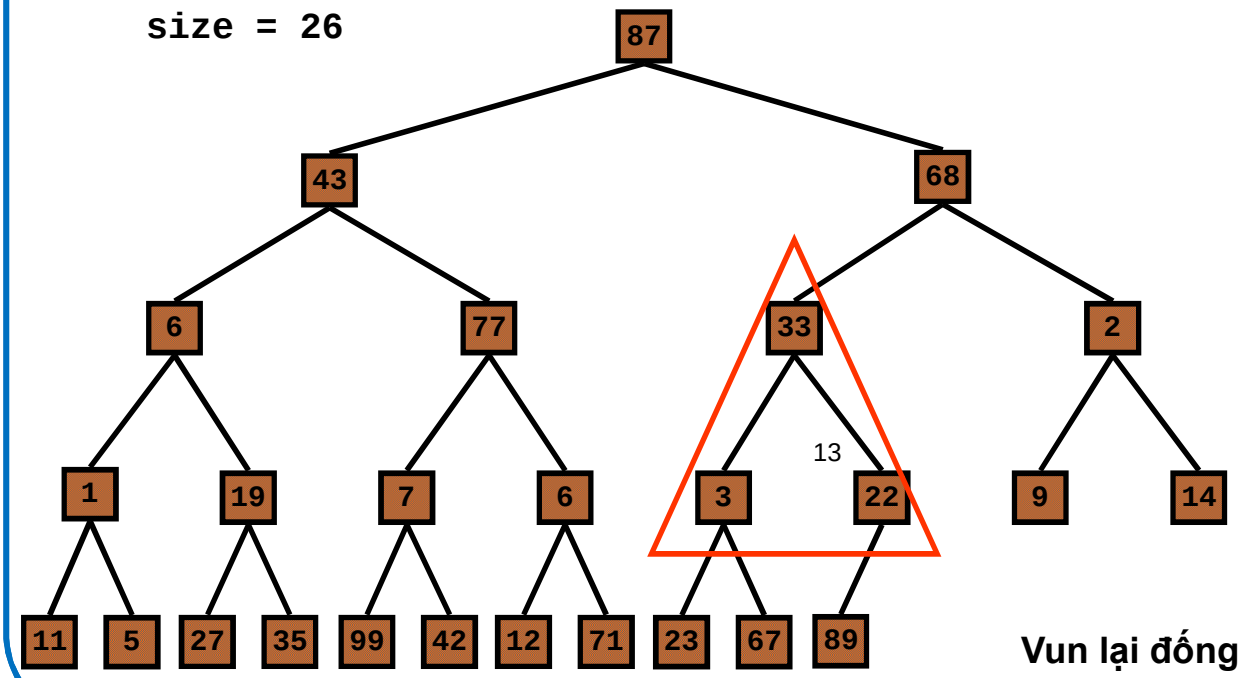
Build-Min-Heap



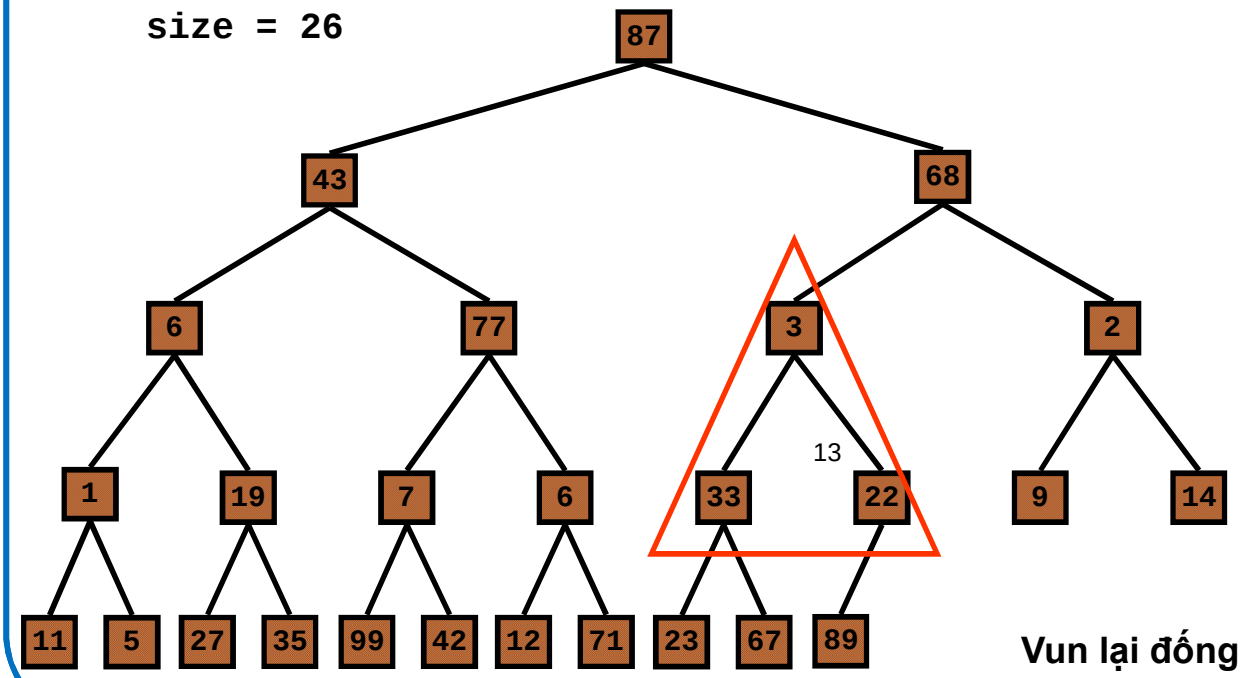
Build-Min-Heap



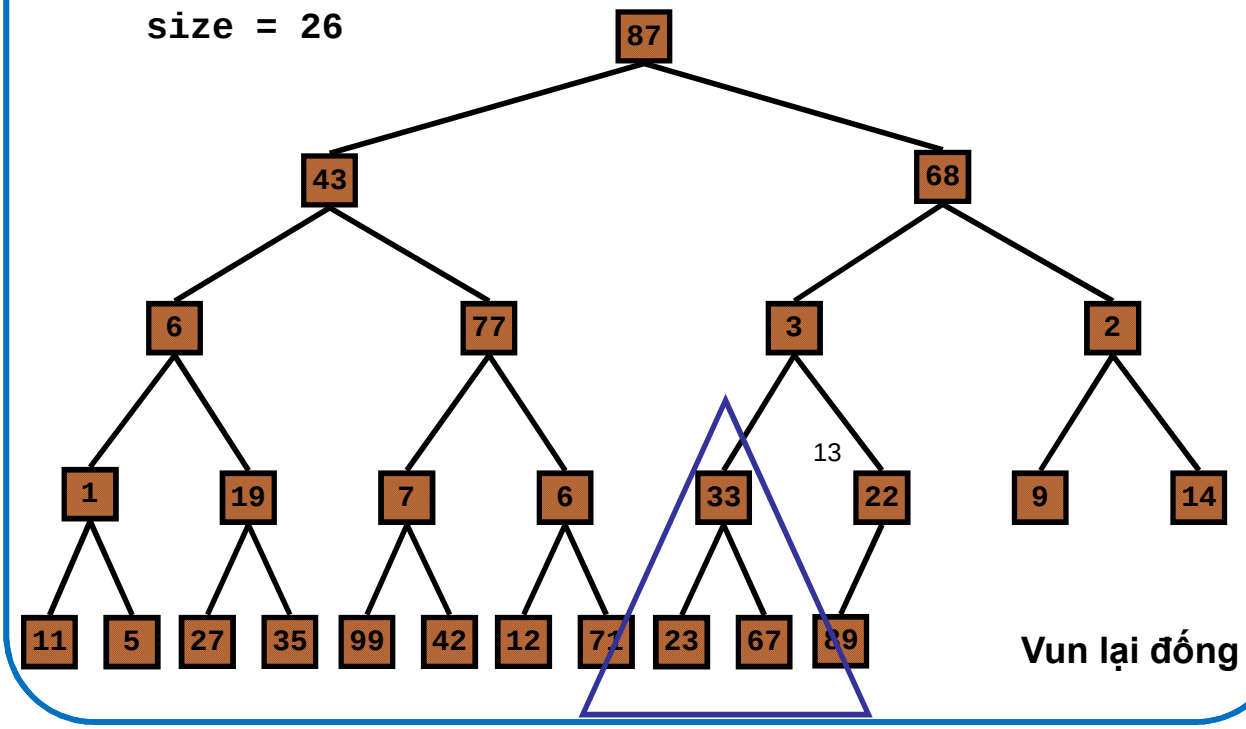
Build-Min-Heap



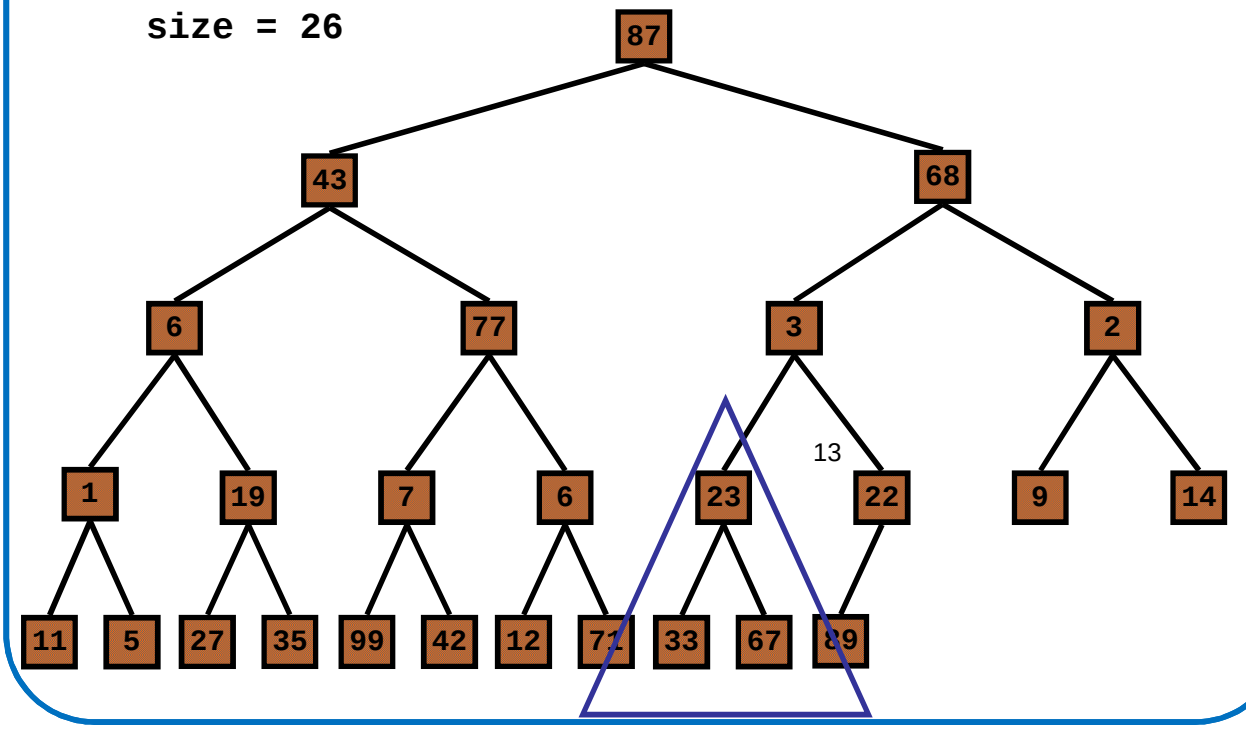
Build-Min-Heap



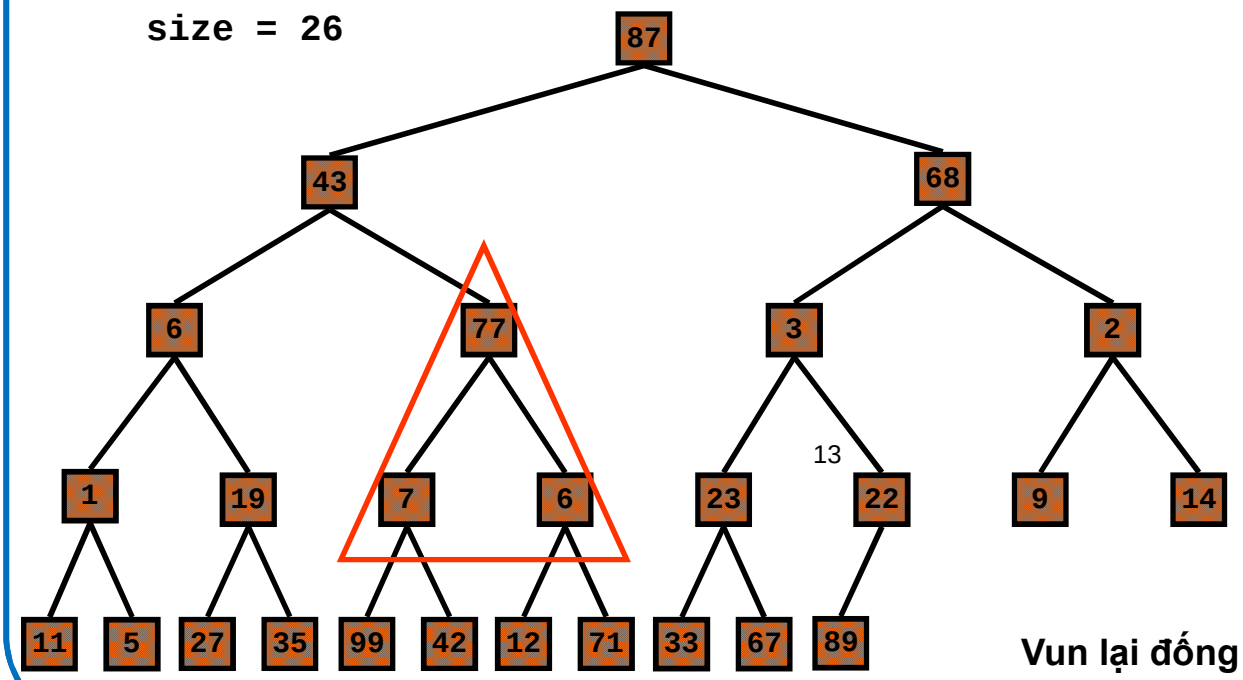
Build-Min-Heap



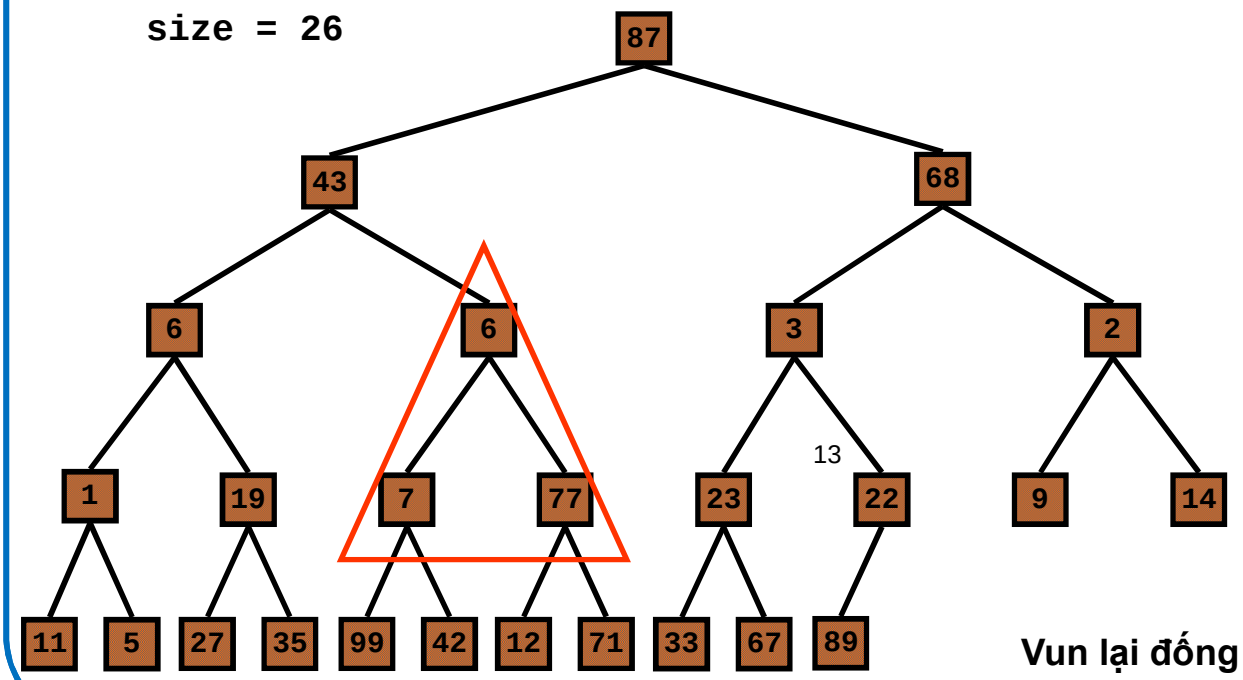
Build-Min-Heap



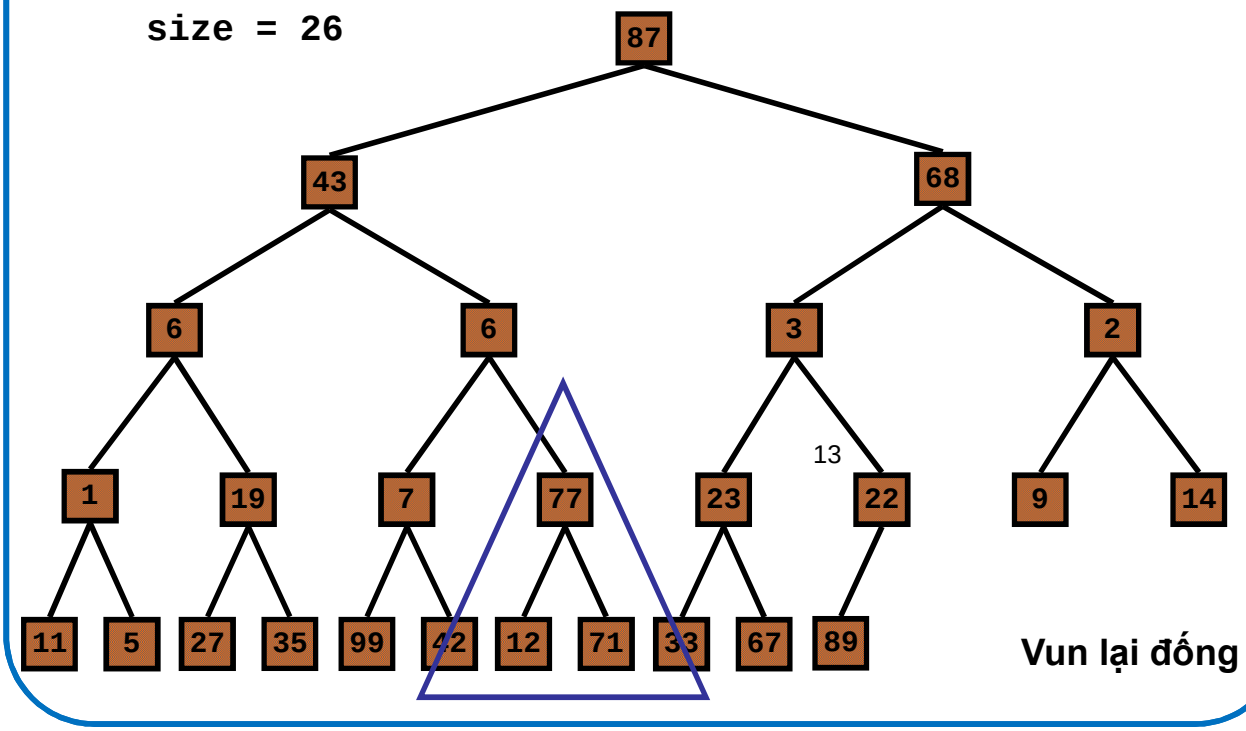
Build-Min-Heap



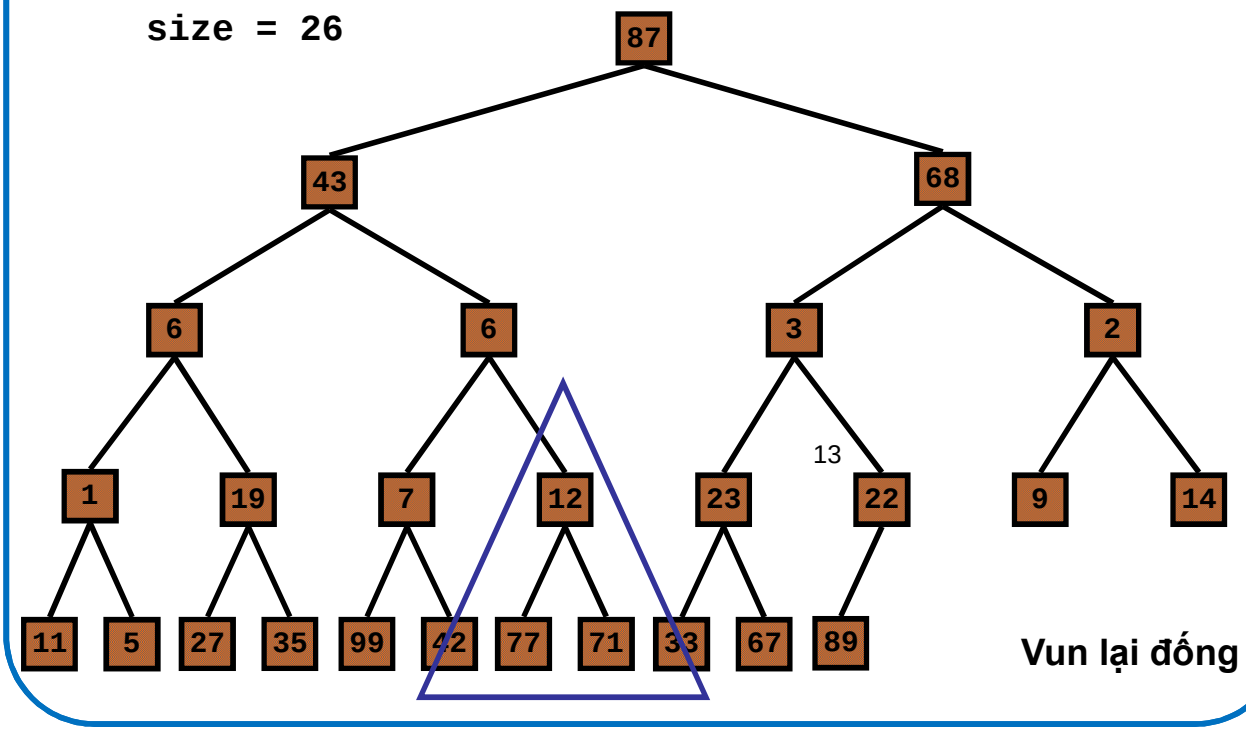
Build-Min-Heap



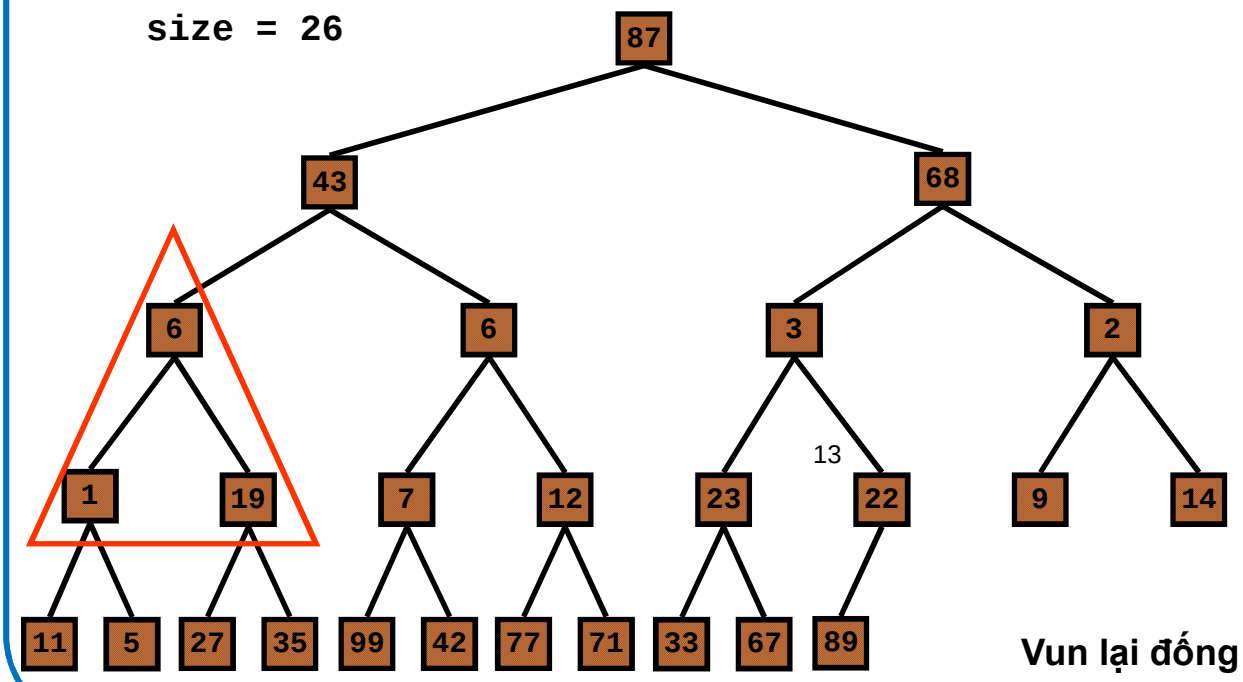
Build-Min-Heap



Build-Min-Heap

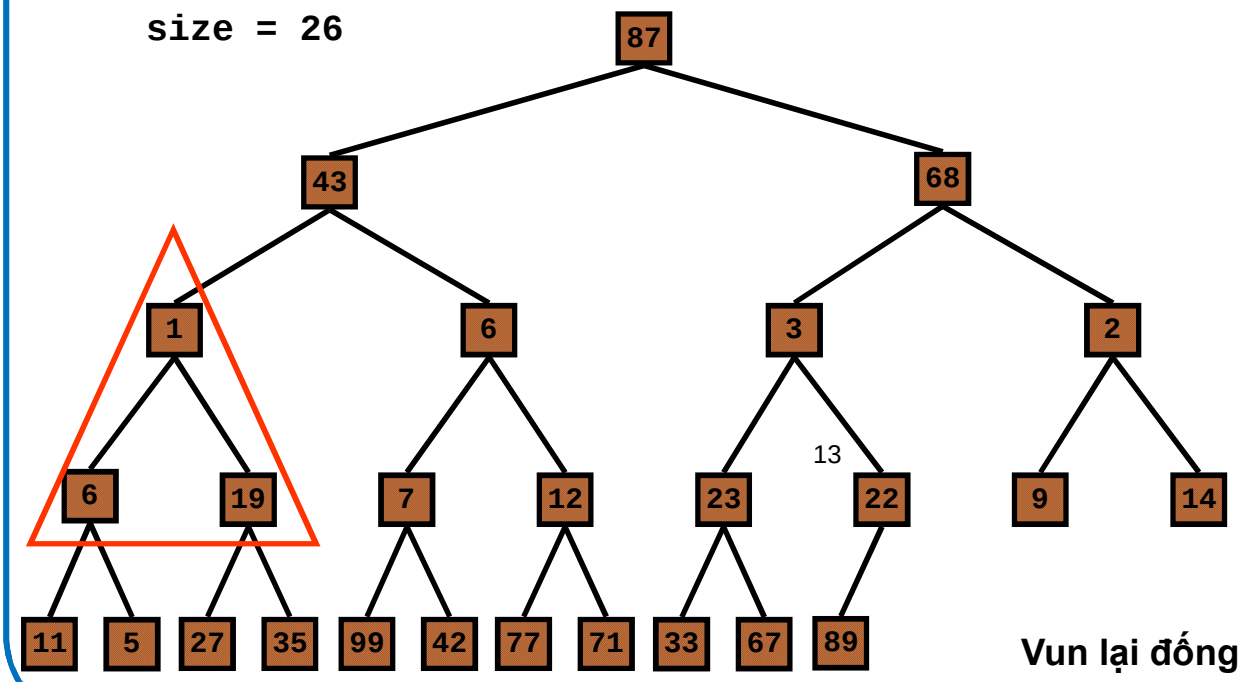


Build-Min-Heap



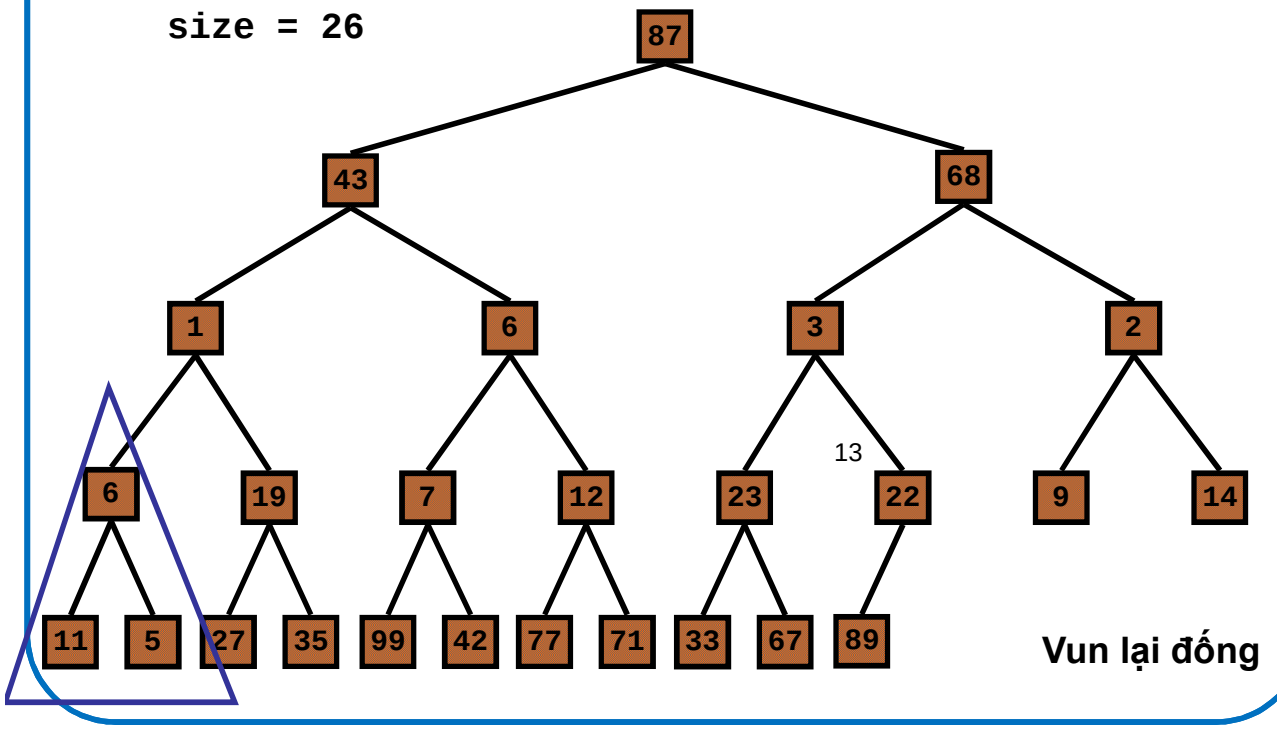
Vun lại đống

Build-Min-Heap

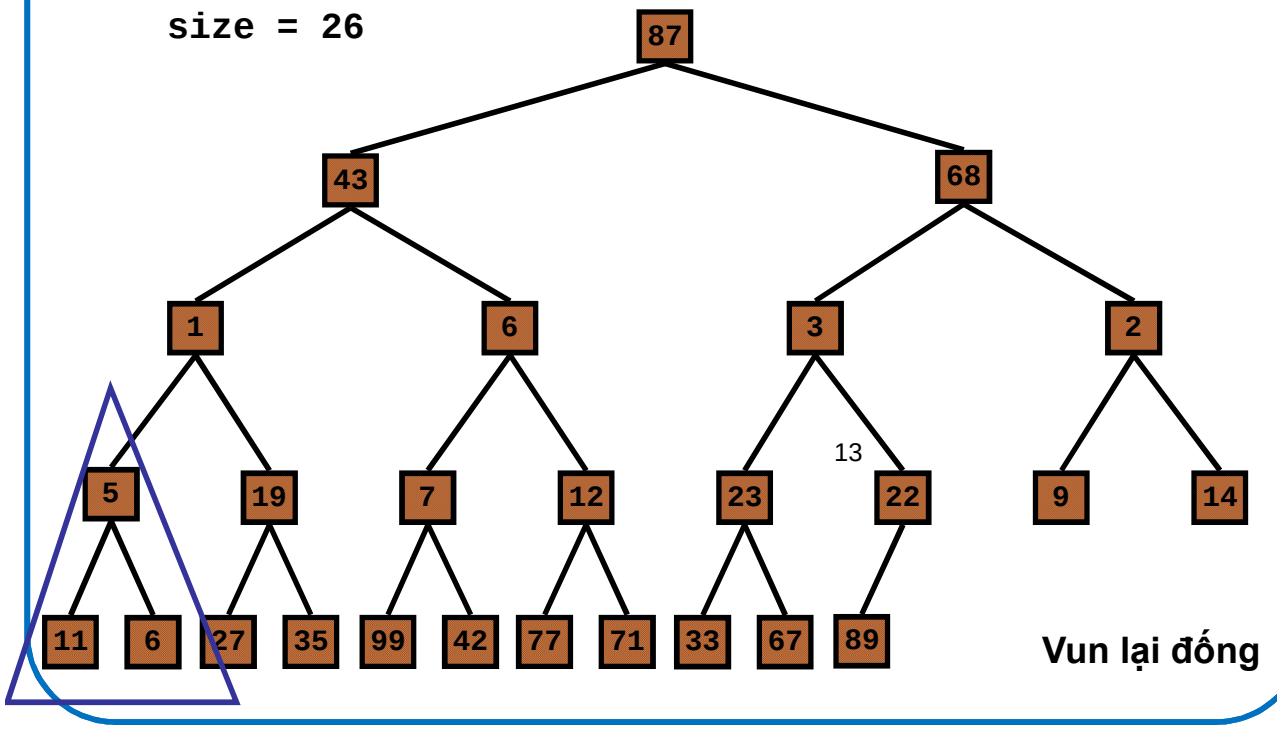


Vun lại đống

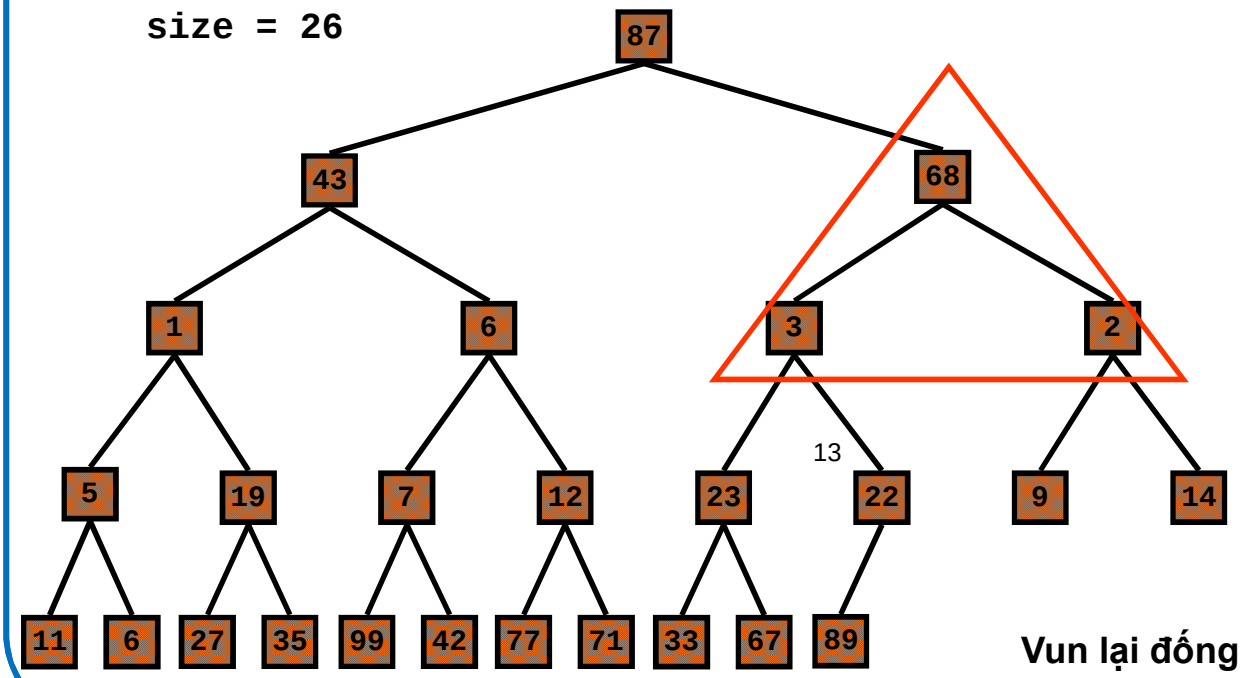
Build-Min-Heap



Build-Min-Heap

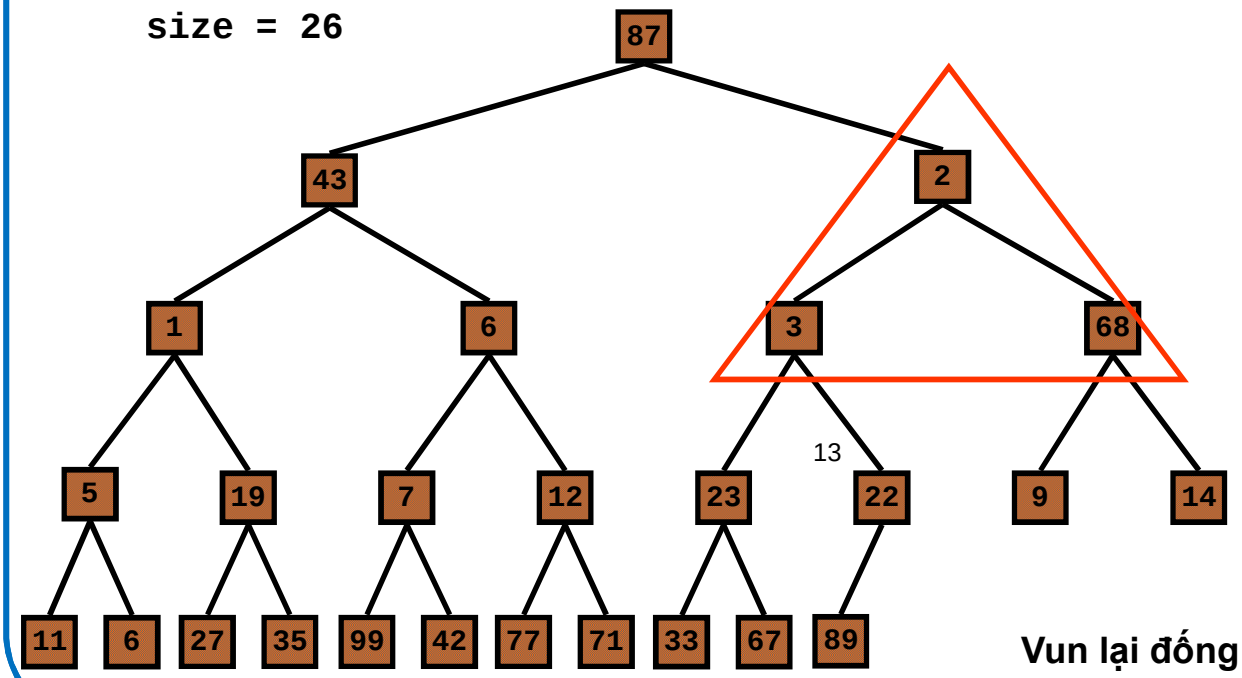


Build-Min-Heap



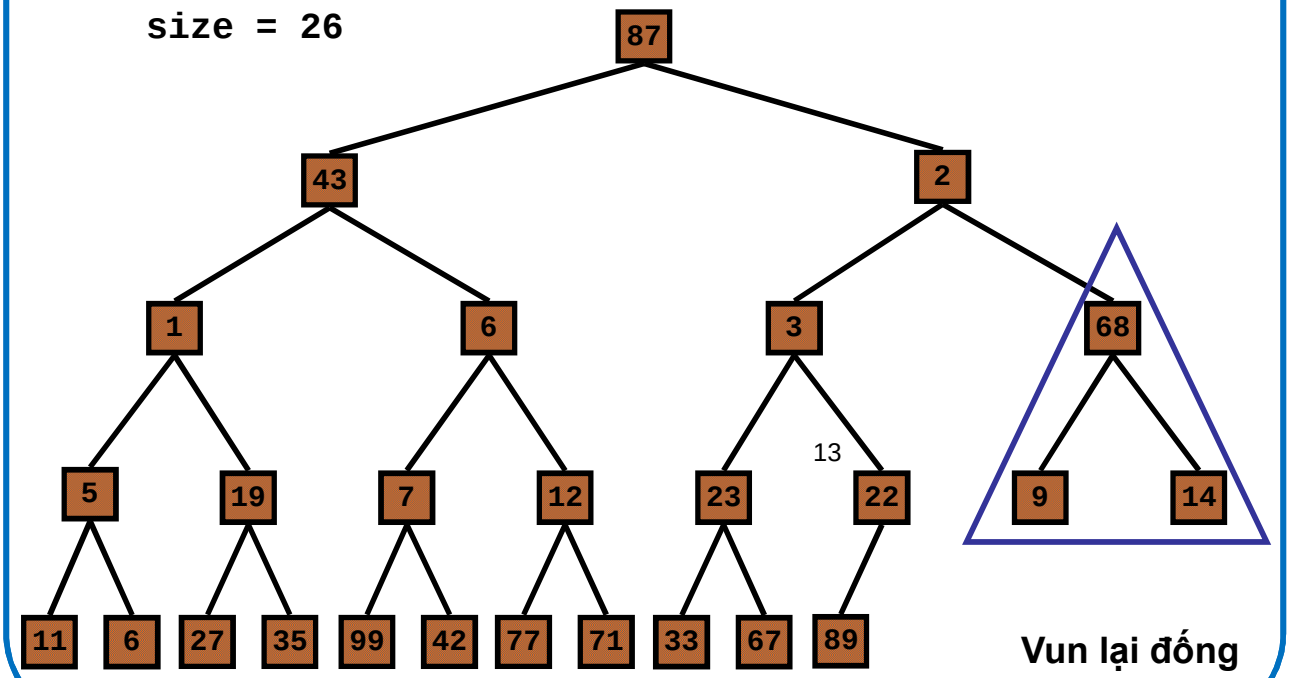
Vun lại đống

Build-Min-Heap

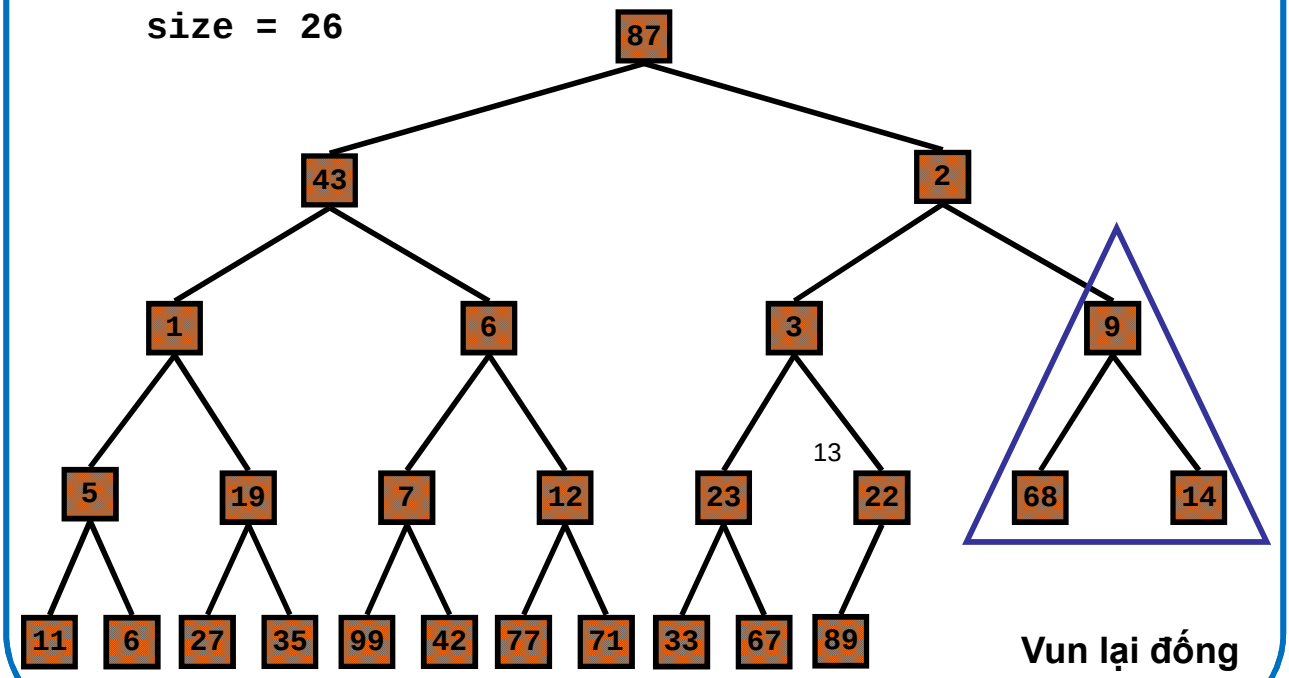


Vun lại đống

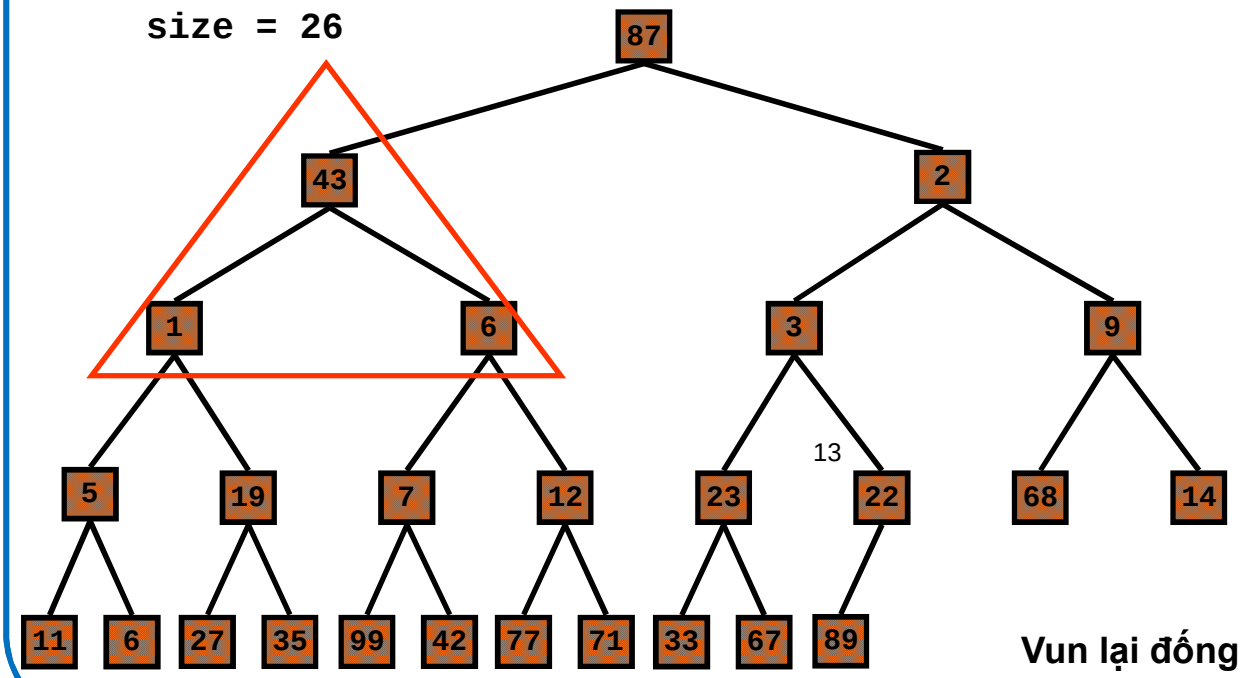
Build-Min-Heap



Build-Min-Heap

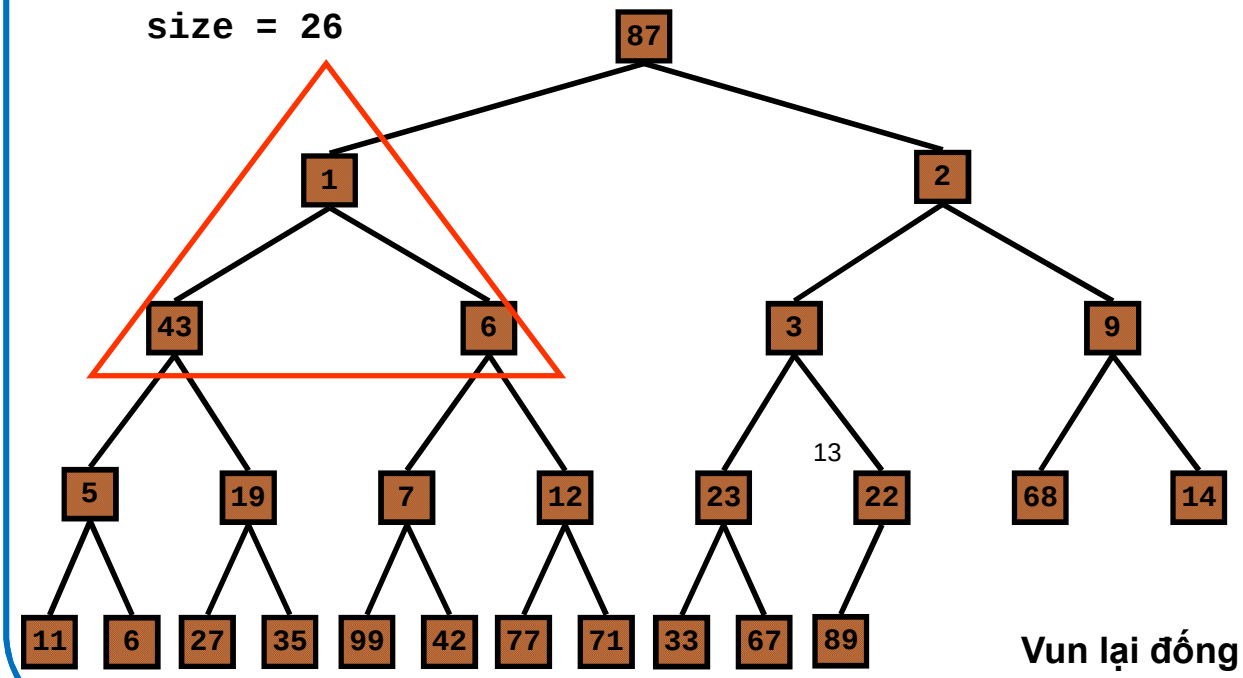


Build-Min-Heap



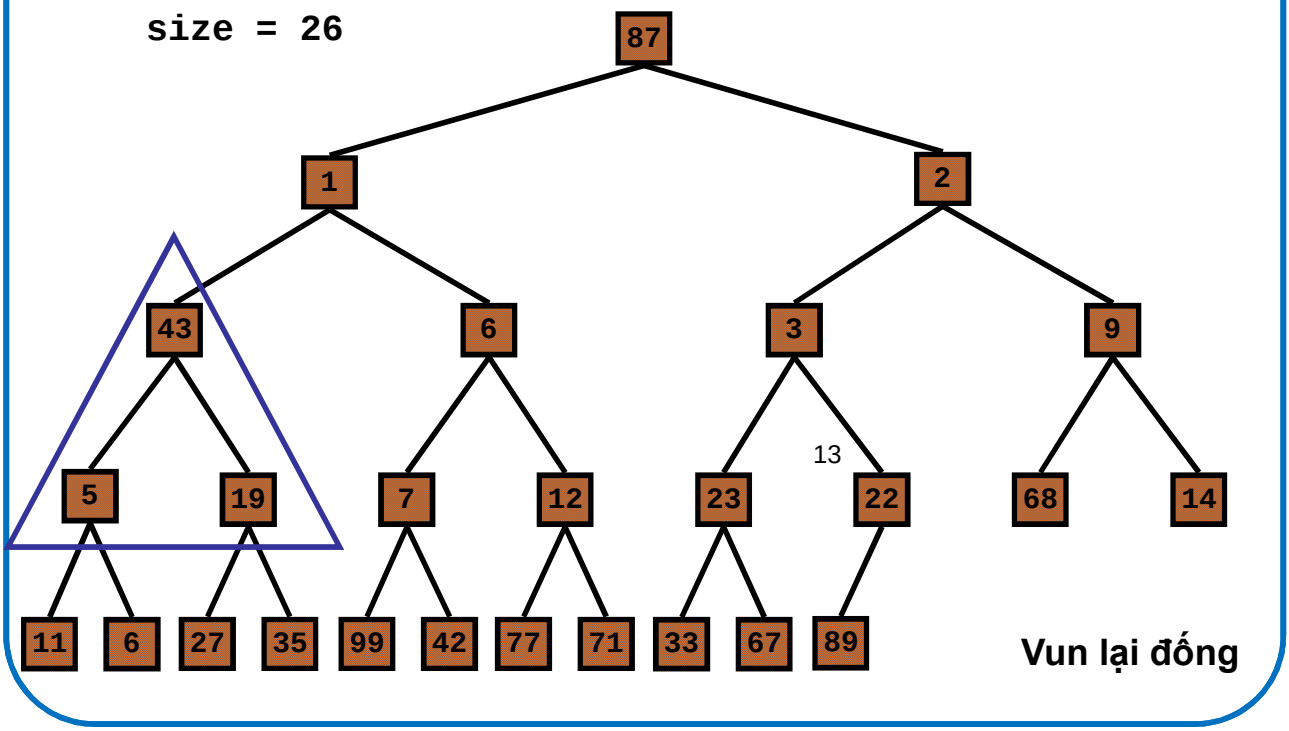
Vun lại đống

Build-Min-Heap

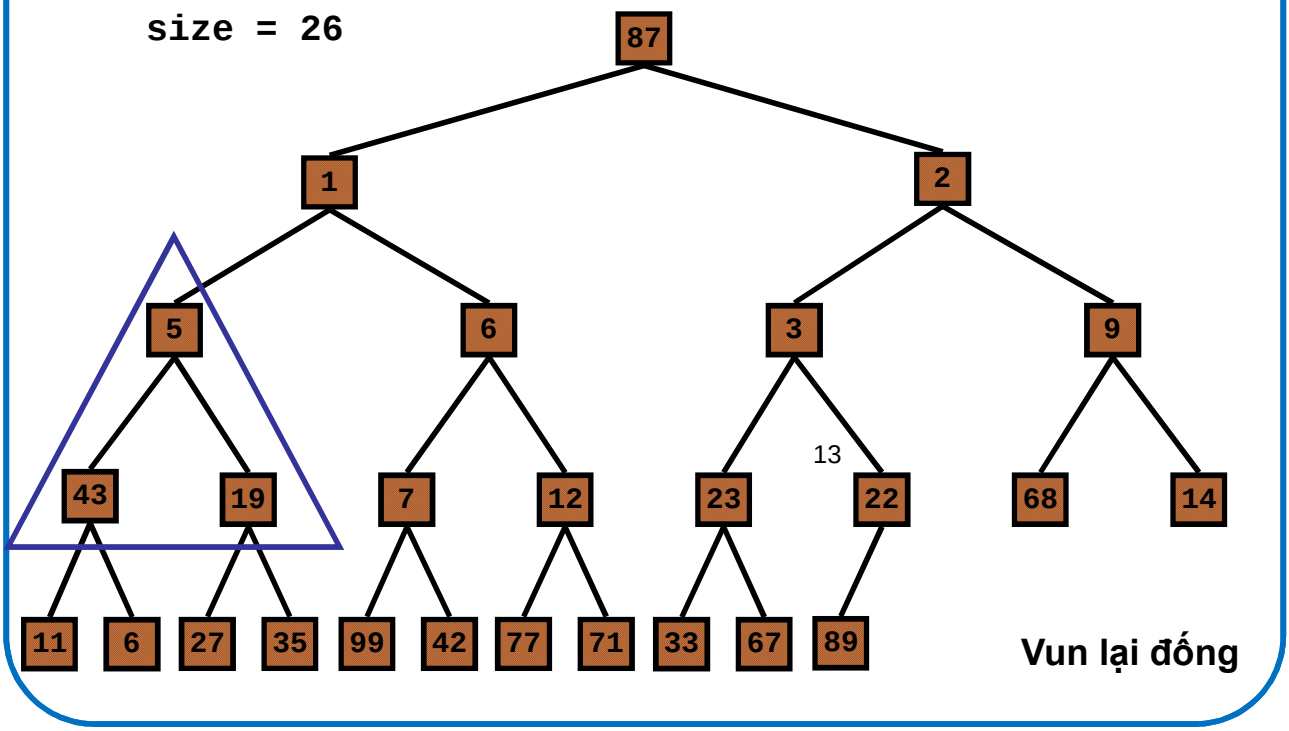


Vun lại đống

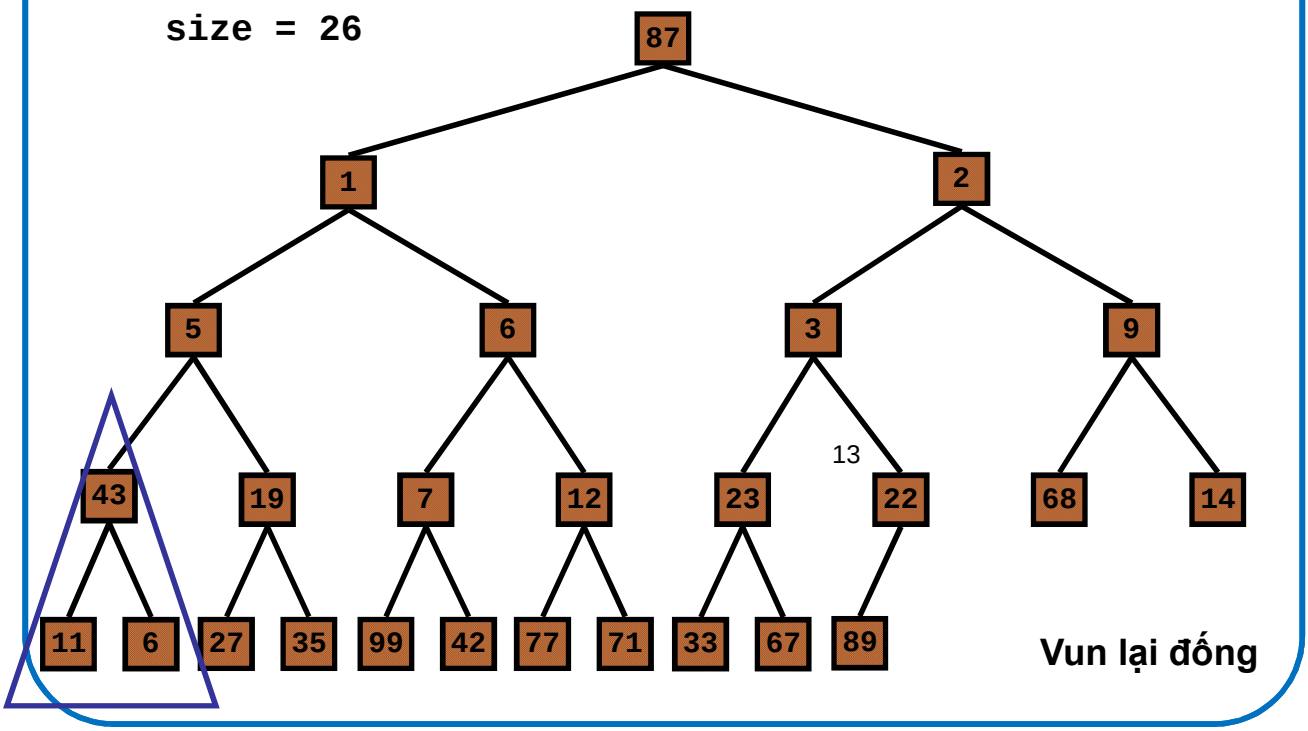
Build-Min-Heap



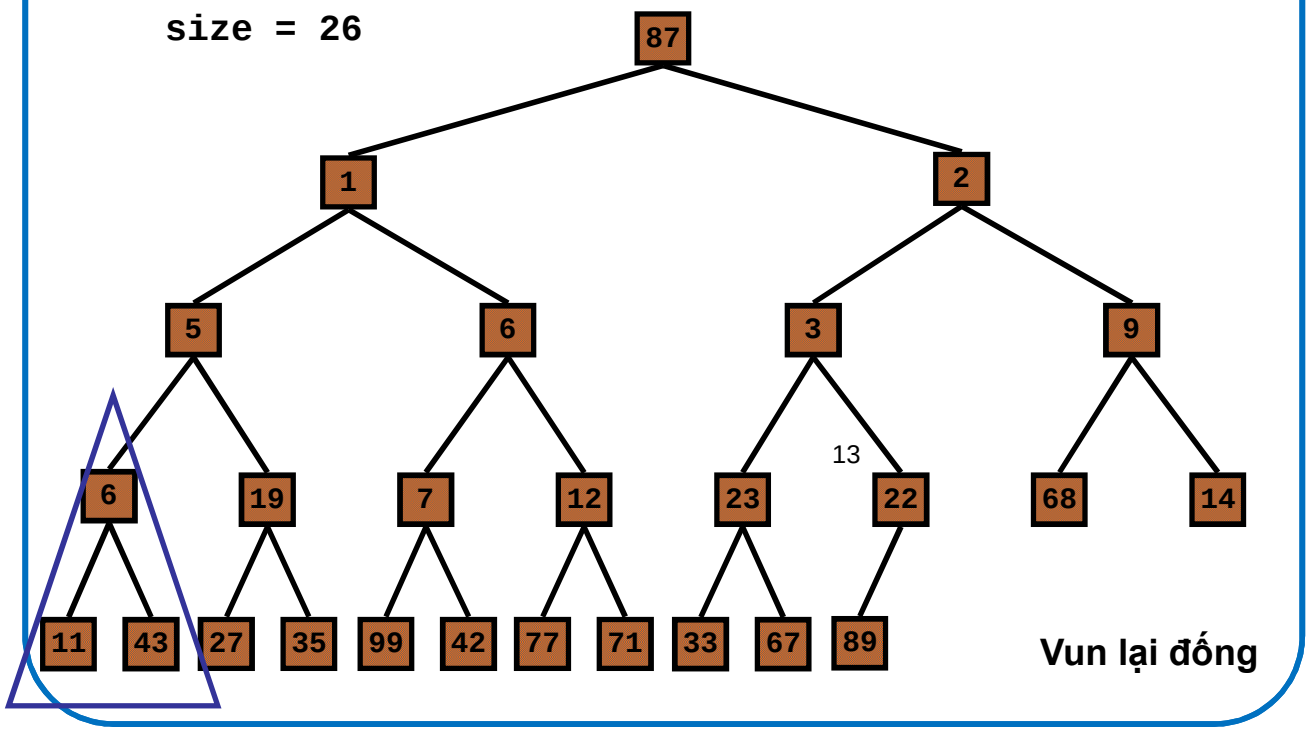
Build-Min-Heap



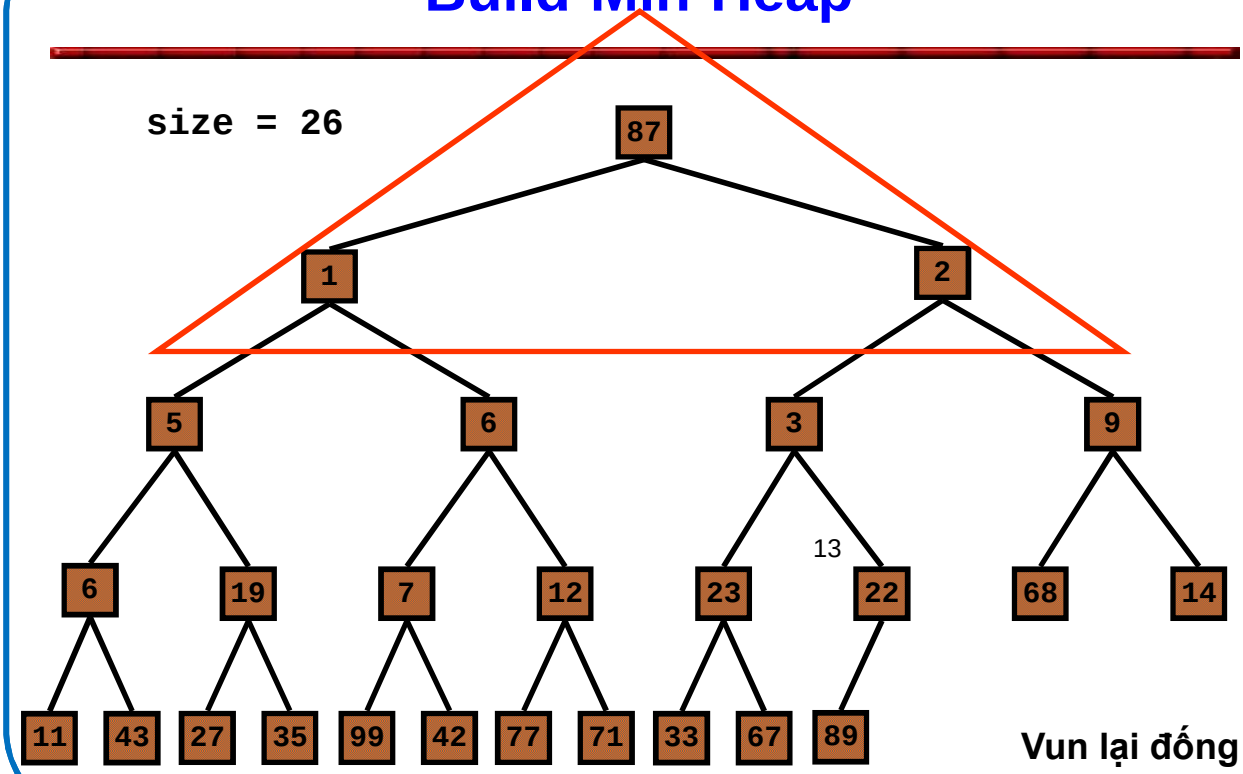
Build-Min-Heap



Build-Min-Heap

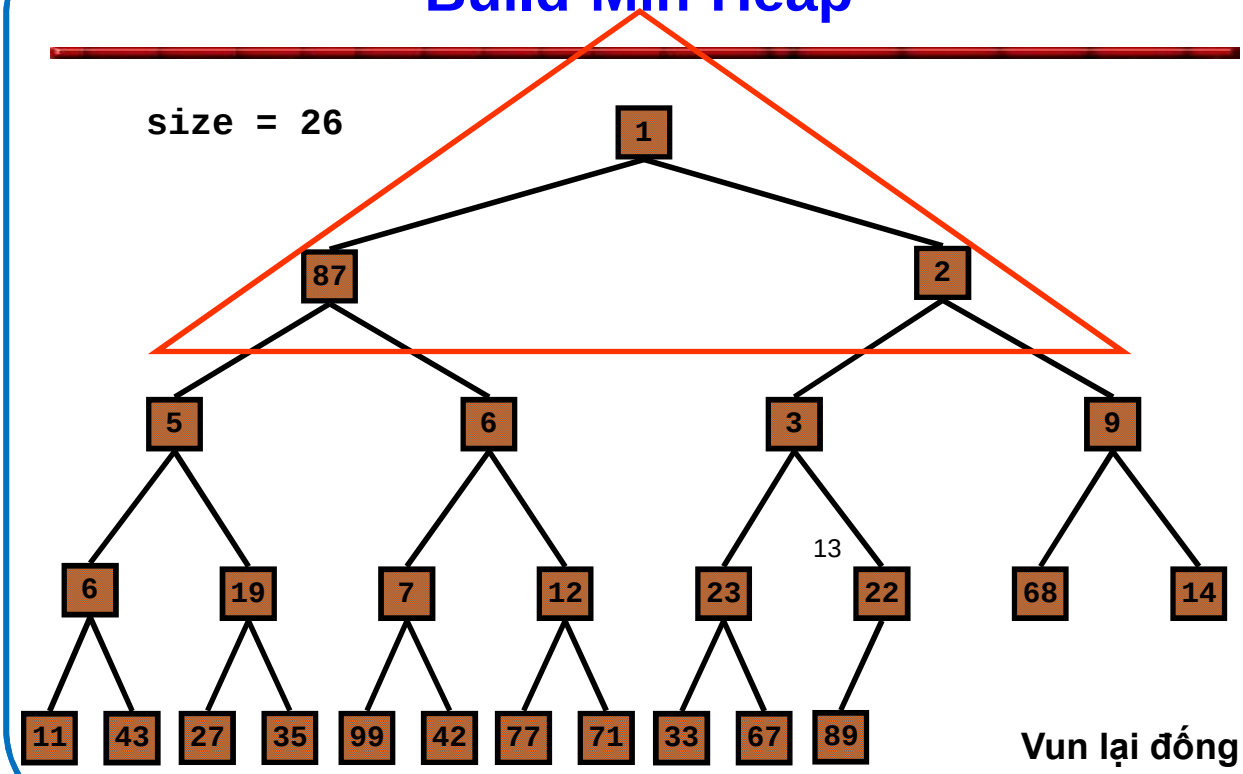


Build-Min-Heap



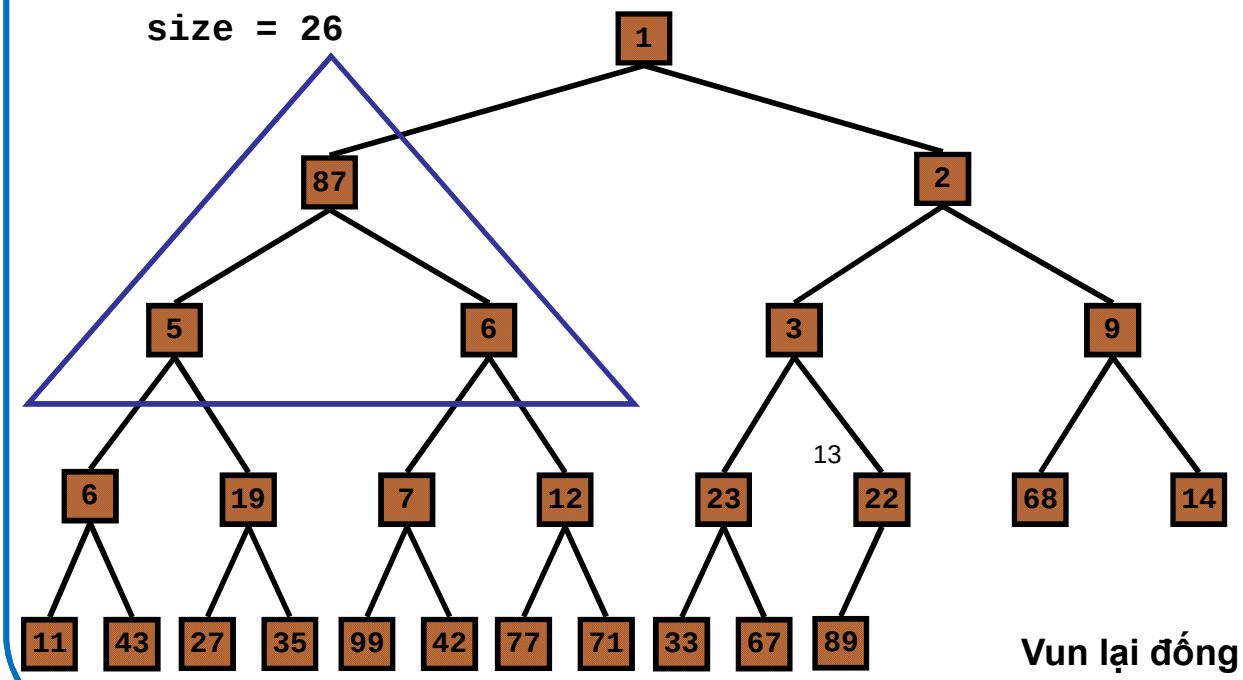
Vun lại đống

Build-Min-Heap



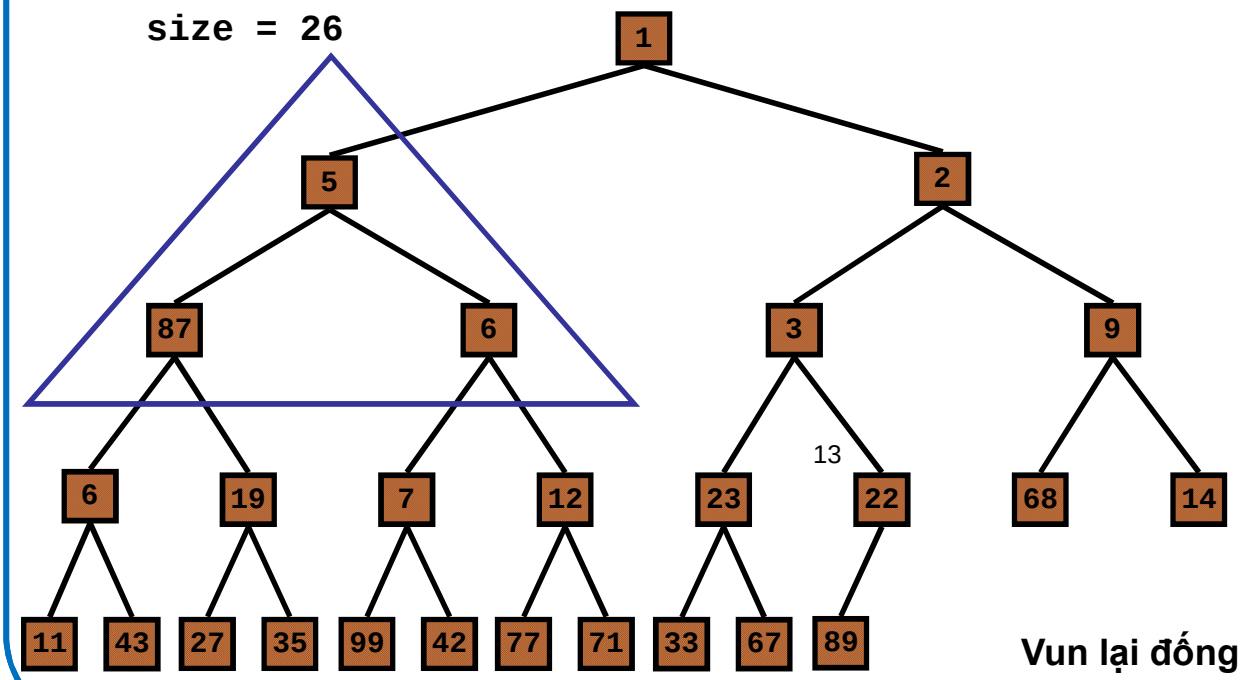
Vun lại đống

Build-Min-Heap



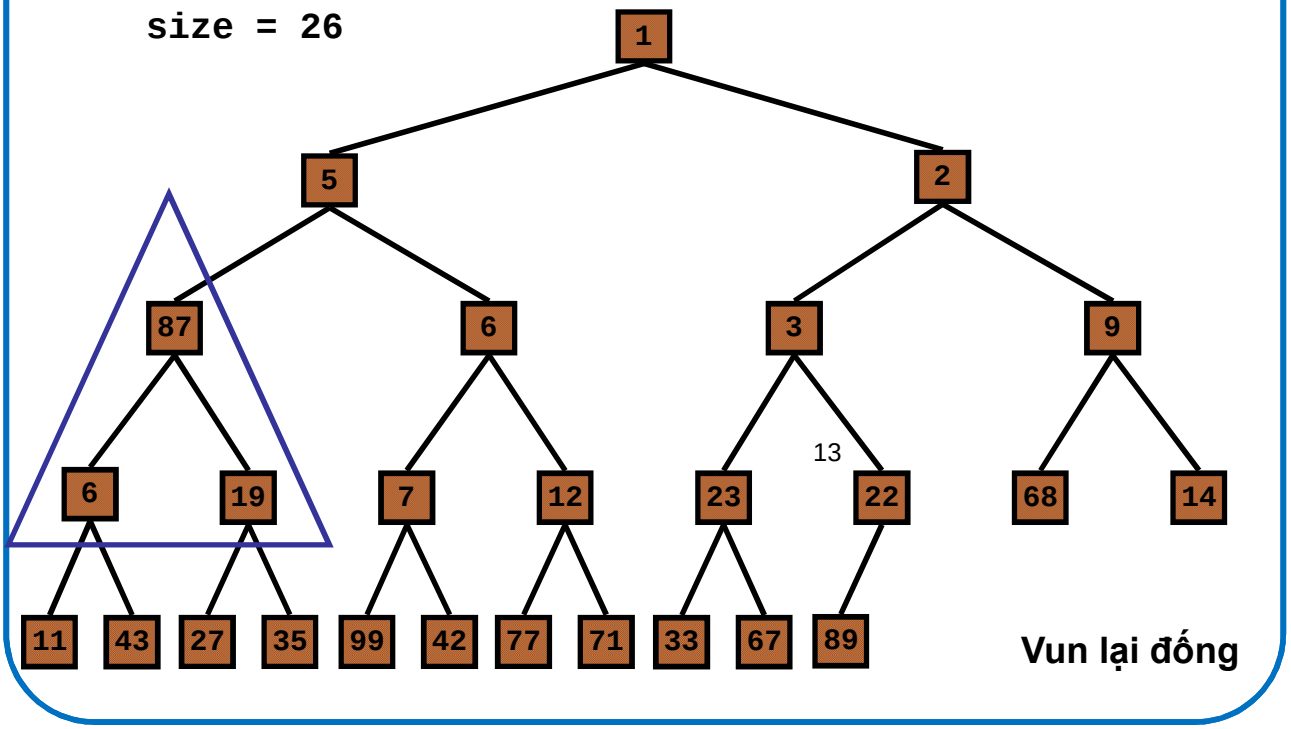
Vun lại đống

Build-Min-Heap

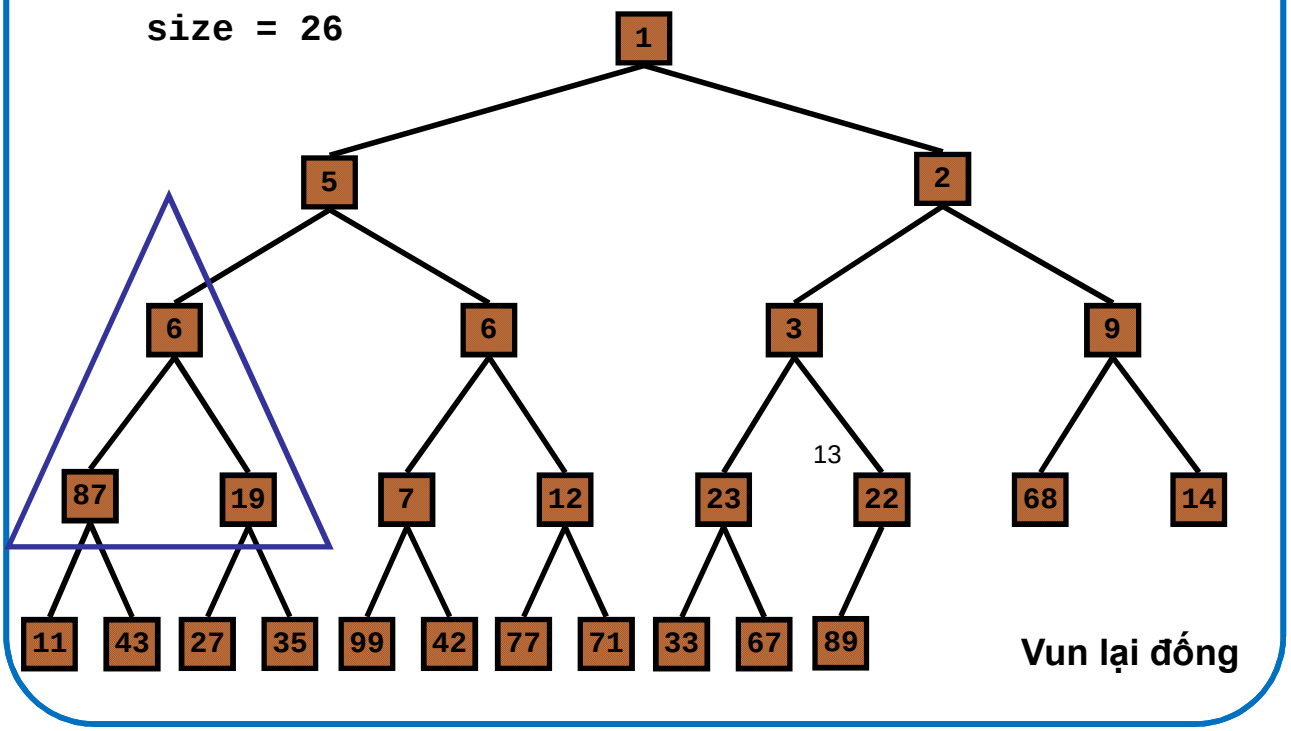


Vun lại đống

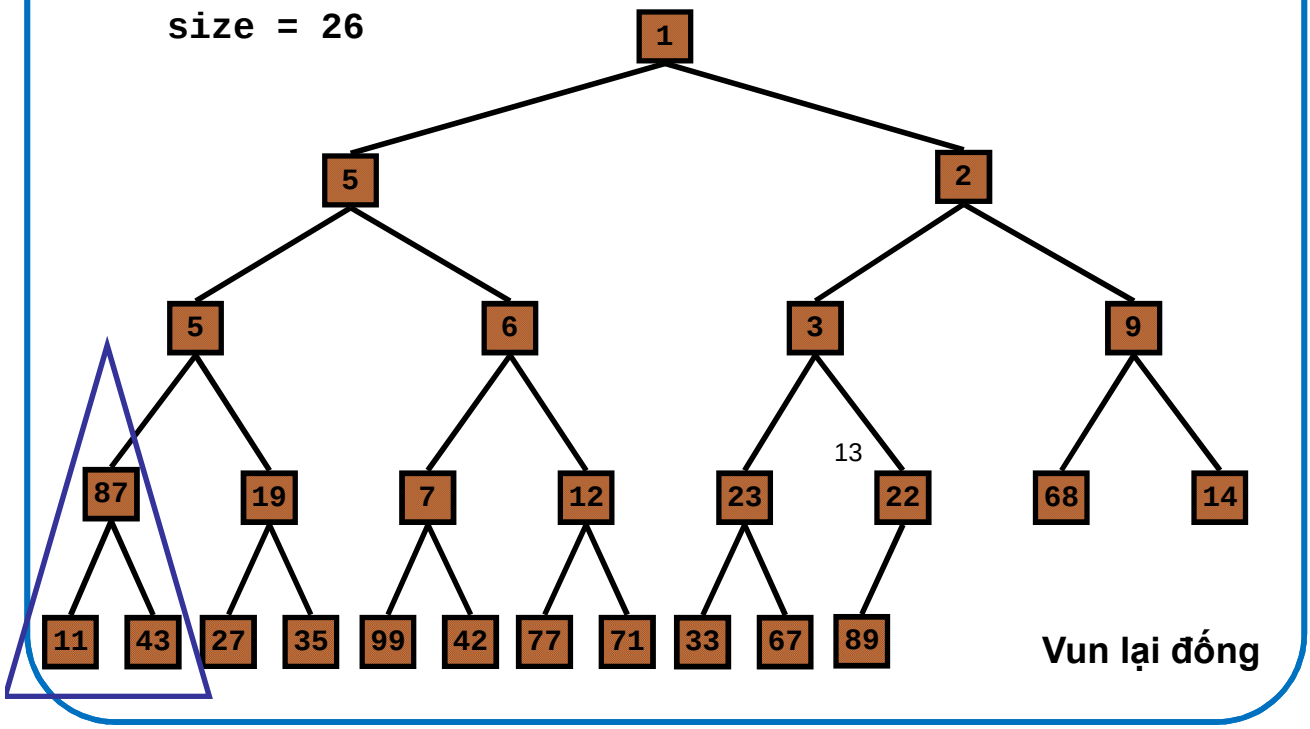
Build-Min-Heap



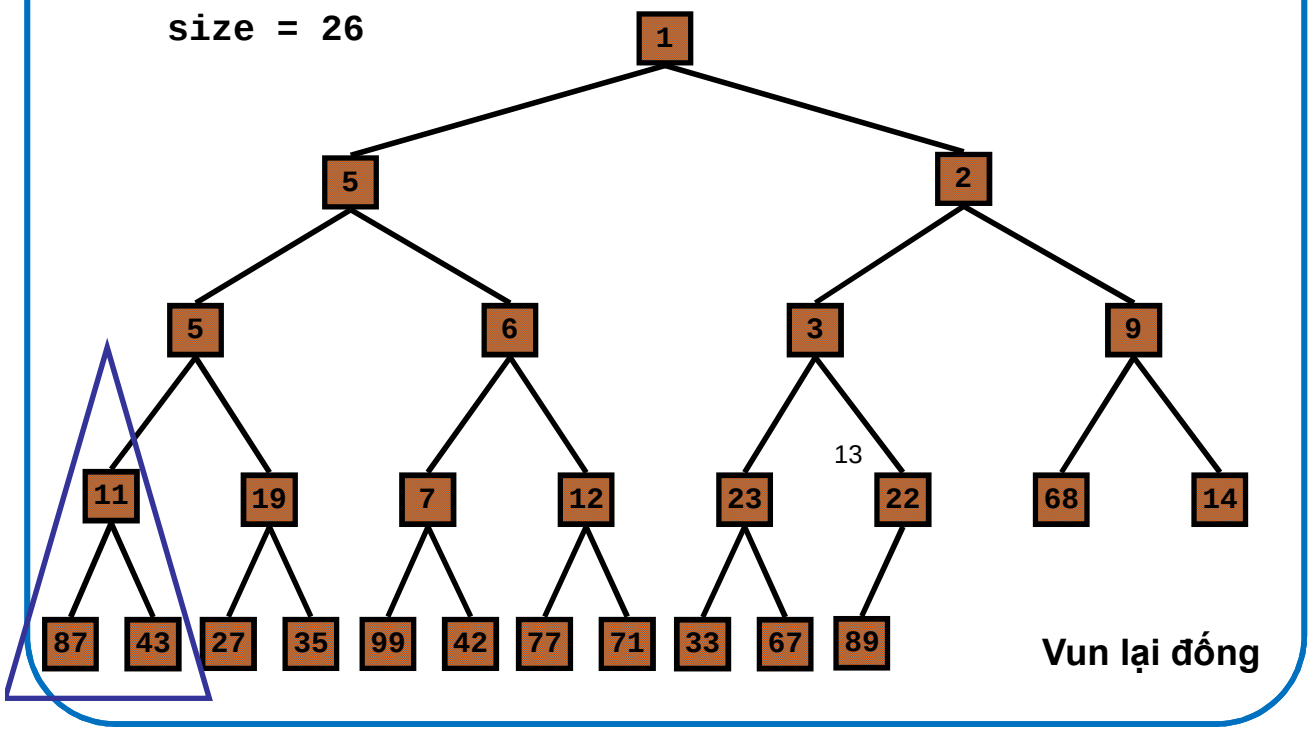
Build-Min-Heap



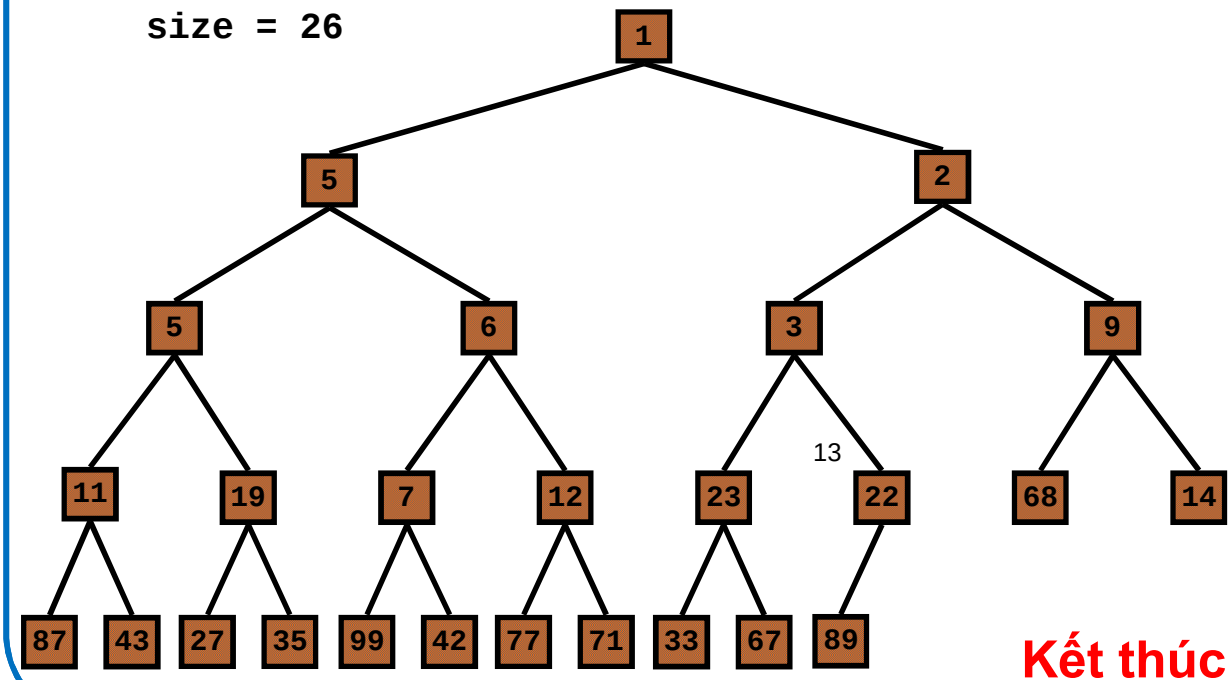
Build-Min-Heap



Build-Min-Heap



Build-Min-Heap



Kết thúc

Thời gian tính của Buil-Max-Heap

Alg: Build-Max-Heap(A)

1. n = length[A]
2. **for** i ← ⌊n/2⌋ **downto** 1
3. **do** Max-Heapify(A, i, n)

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

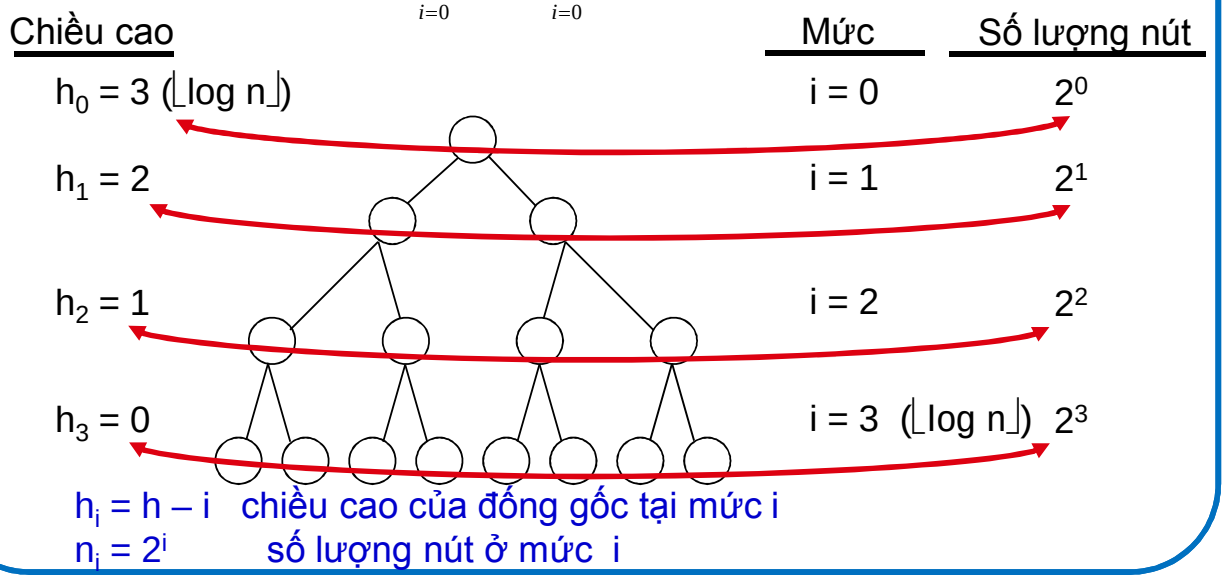
⇒ Thời gian tính là: $O(n \log n)$

- 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

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



$h_i = h - i$ chiều cao của đống gốc tại mức i

$n_i = 2^i$ số lượng nút ở mức i

Thời gian tính của Build-Max-Heap

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

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

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

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

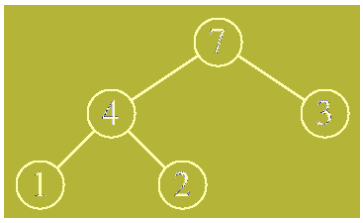
$$\leq n \sum_{k=0}^{\infty} \frac{k}{2^k}$$ Thay tổng hữu hạn bởi tổng vô hạn và $h = \log n$

$$= O(n)$$ Tổng trên là nhỏ hơn 2

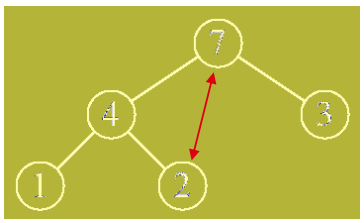
Thời gian tính của Build-Max-Heap: $T(n) = O(n)$

5.5.2. Sắp xếp vun đống - Heapsort

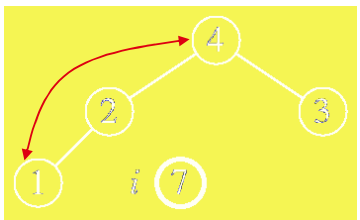
- Sử dụng đống ta có thể phát triển thuật toán sắp xếp mảng.
- Sơ đồ của thuật toán được trình bày như sau:
  - Tạo đống **max-heap** từ mảng đã cho
  - Đổi chỗ gốc (phần tử lớn nhất) với phần tử cuối cùng trong mảng
  - Loại bỏ nút cuối cùng bằng cách giảm kích thước của đống đi 1
  - Thực hiện Max-Heapify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



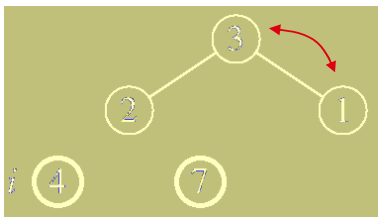
Ví dụ: A=[7, 4, 3, 1, 2]



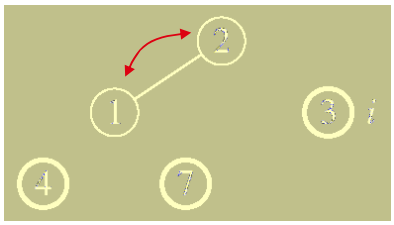
Max-Heapyfy(A, 1, 4)

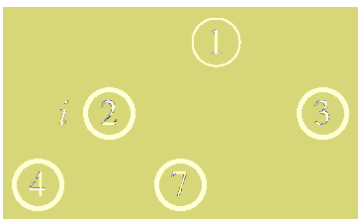


Max-Heapyfy(A, 1, 3)



Max-Heapyfy(A, 1, 2)





Max-Heapyfy(A, 1, 1)

Algorithm: HeapSort(*A*)

1. Build-Max-Heap(A) — $O(n)$
2. **for** i ← length[A] **downto** 2
3. **do** exchange A[1] ↔ 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 $\Theta(n \log n)$

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:
  - Insert(*S*, *x*) – Bổ sung phần tử *x* vào *S*
  - Max(*S*) – trả lại phần tử lớn nhất
  - Extract-Max(*S*) – loại bỏ và trả lại phần tử lớn nhất
  - 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 đống để cất giữ các khoá.
- **Chú ý:** *Có thể thay "max" bởi "min" .*

Các phép toán đối với hàng đợi có ưu tiên
Operations on Priority Queues

- 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 ≥ khoá hiện tại của x)

Các phép toán đối với hàng đợi có ưu tiên (min)
Operations on Priority Queues

- 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
  - **Deacrease-Key(S, x, k): giảm** giá trị của khoá của phần tử x xuống còn k (Giả sử k ≤ khoá hiện tại của x)

Phép toán HEAP-MAXIMUM

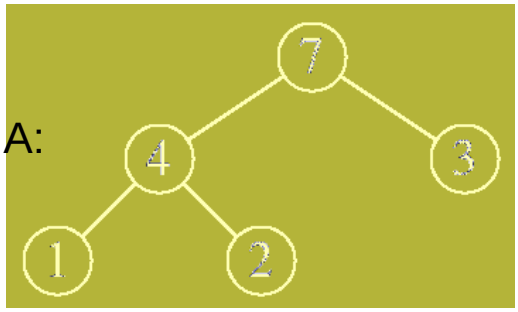
Chức năng: Trả lại phần tử lớn nhất của đống

Alg: Heap-Maximum(A)

1. **return** A[1]

Thời gian tính: $O(1)$

Heap A:



Heap-Maximum(A) trả lại 7

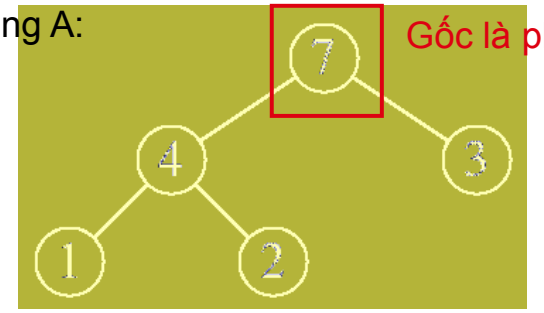
Heap-Extract-Max

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

Thuật toán:

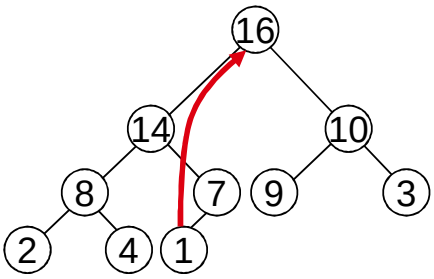
- Hoán đổi gốc với phần tử cuối cùng
- Giảm kích thước của đống đi 1
- Gọi Max-Heapify đối với gốc mới trên đống có kích thước n-1

Đống A:

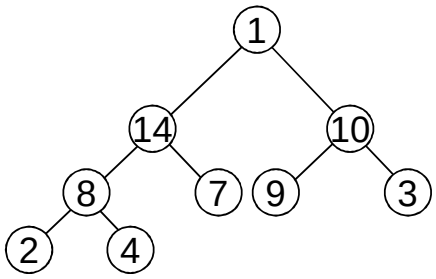


Gốc là phần tử lớn nhất

Ví dụ: Heap-Extract-Max

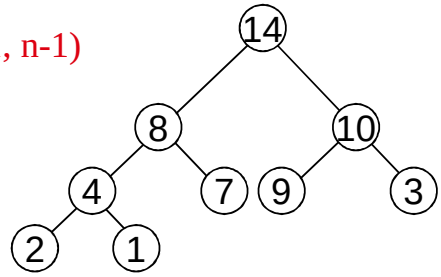


max = 16



Kích thước đống giảm đi 1

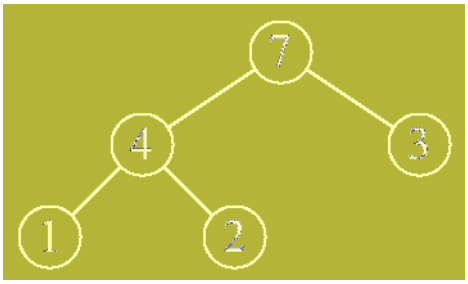
Thực hiện Max-Heapify(A, 1, n-1)



Heap-Extract-Max

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

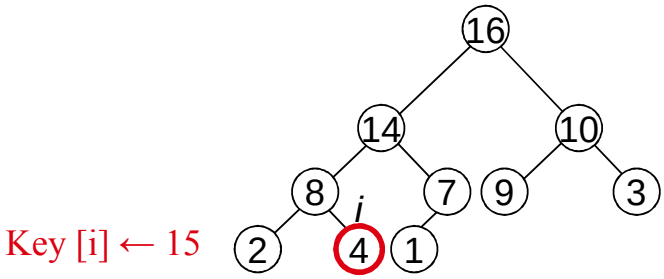
1. **if** n < 1
2. **then error** "heap underflow"
3. max ← A[1]
4. A[1] ← A[n]
5. Max-Heapify(A, 1, n-1) // Vun lại đống
6. **return** max



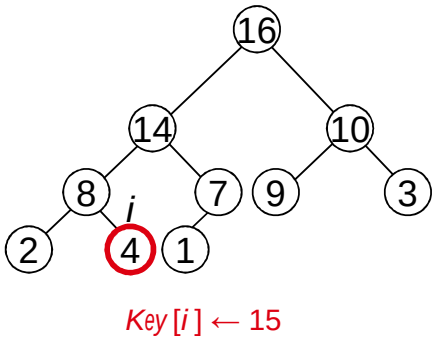
Thời gian tính: $O(\log n)$

Heap-Increase-Key

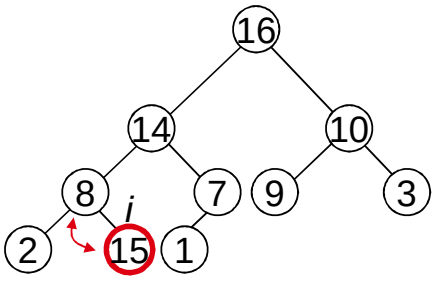
- **Chức năng:** Tăng giá trị khoá của phần tử i trong đống
- **Thuật toán:**
  - Tăng khoá của A[i] thành giá trị mới
  - Nếu tính chất max-heap bị vi phạm: di chuyển theo đường đến gốc để tìm chỗ thích hợp cho khoá mới bị tăng này

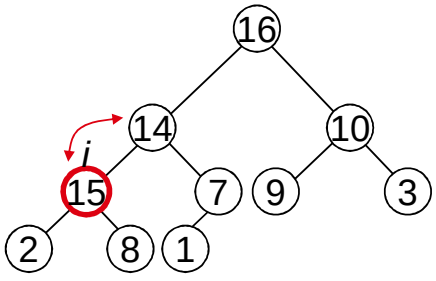


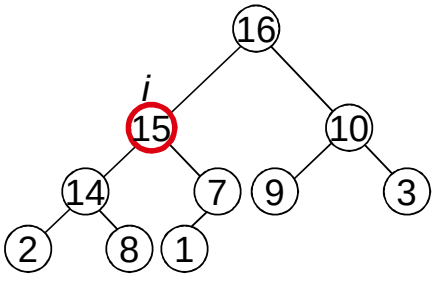
Ví dụ: Heap-Increase-Key



Key [*i*] ← 15





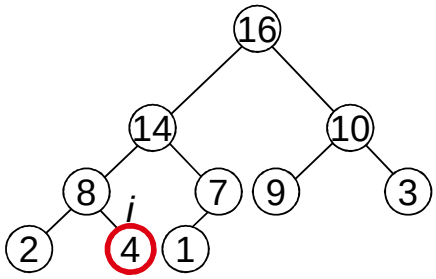


Heap-Increase-Key

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

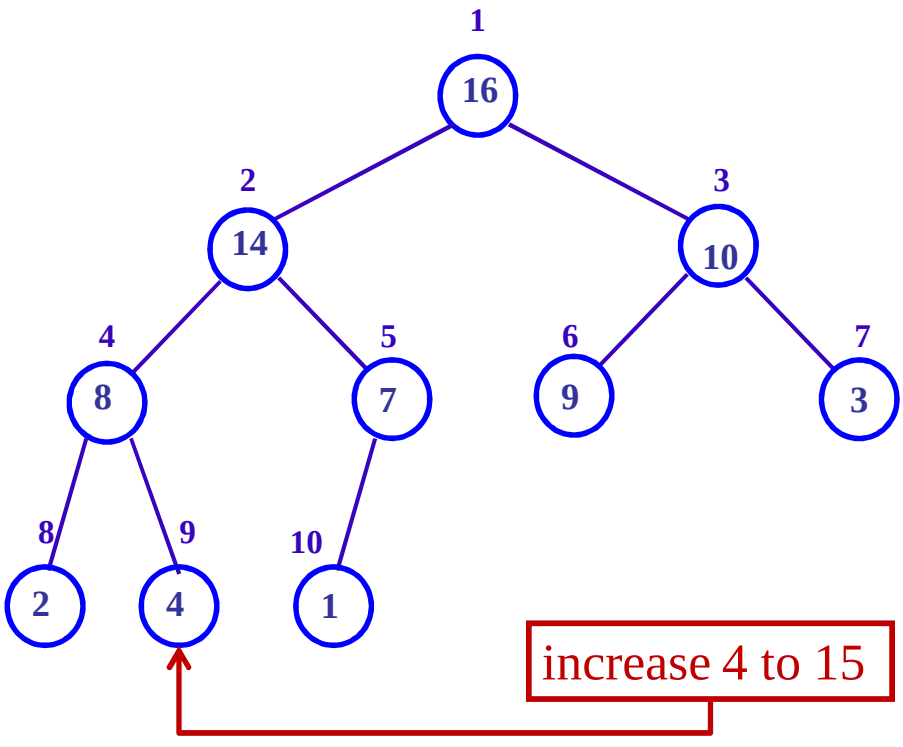
1. **if** key < A[i]
2. **then error** "khoá mới nhỏ hơn khoá hiện tại"
3. A[i] ← key
4. **while** i > 1 and A[parent(i)] < A[i]
5. **do** hoán đổi A[i] ↔ A[parent(i)]
6. i ← parent(i)

- Thời gian tính: $O(\log n)$

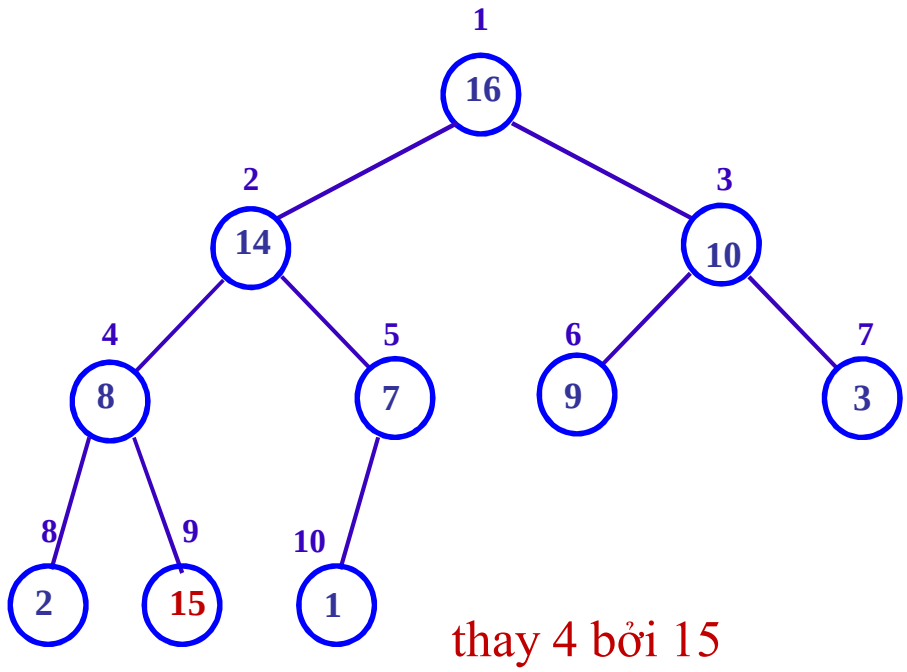


Key [i] ← 15

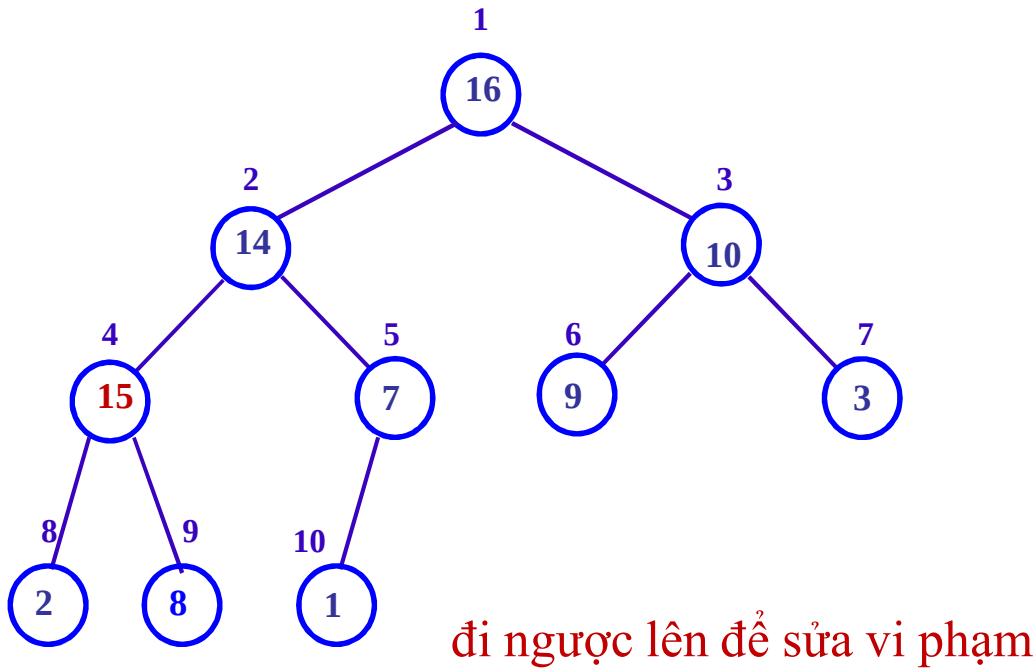
Ví dụ: Heap-Increase-Key (1)



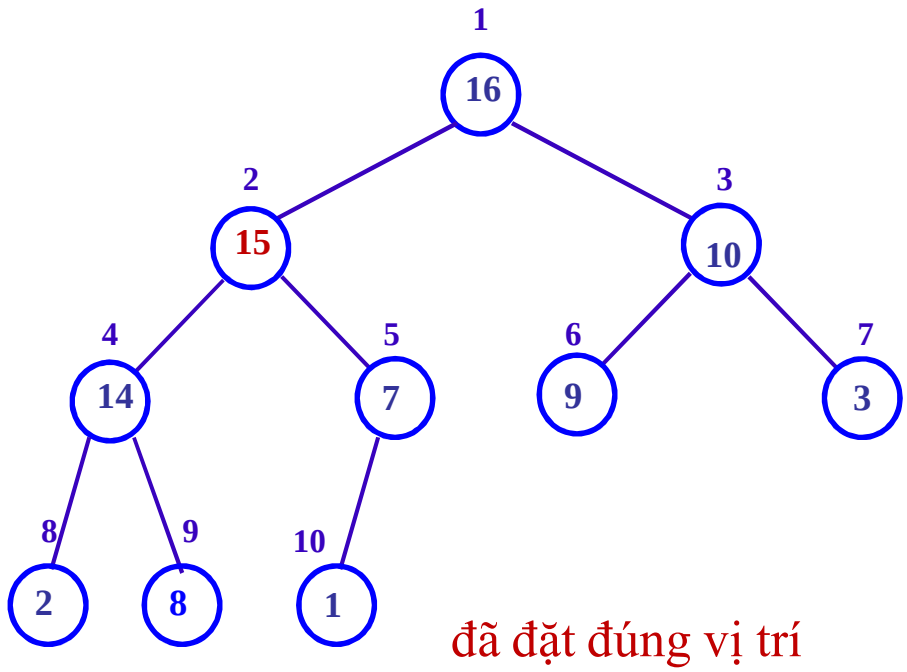
Ví dụ: Heap-Increase-Key (2)



Ví dụ: Heap-Increase-Key (3)

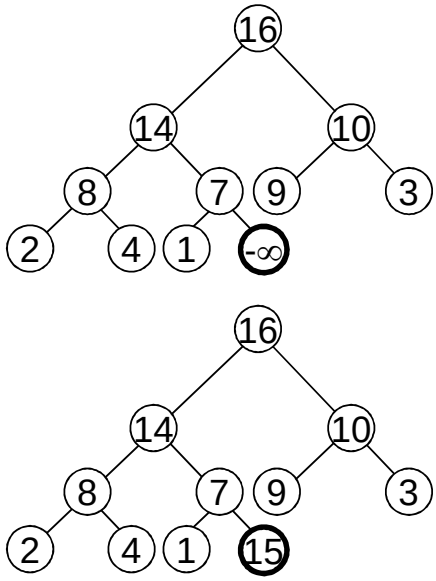


Ví dụ: Heap-Increase-Key (4)



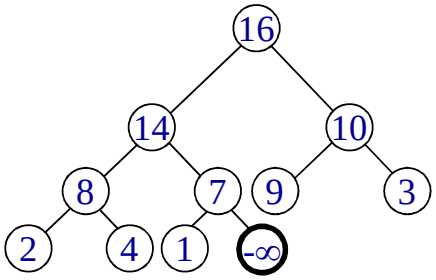
Max-Heap-Insert

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

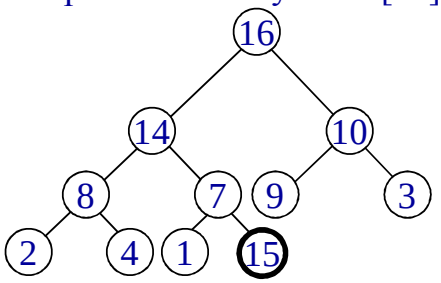


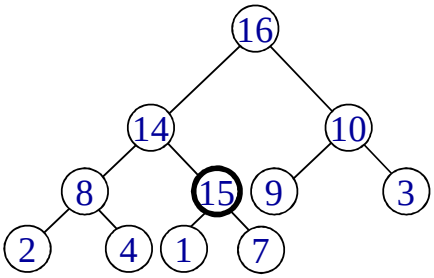
Ví dụ: Max-Heap-Insert

Chèn giá trị 15:
- Bắt đầu với -∞

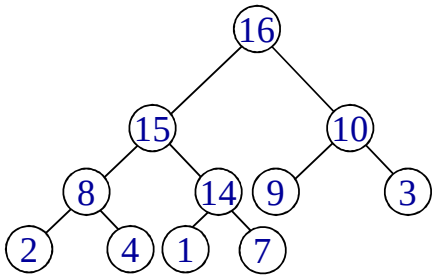


Tăng khoá thành 15
Gọi Heap-Increase-Key với A[11] = 15





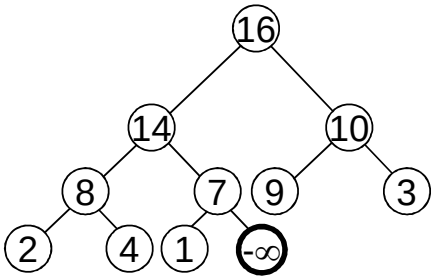
Vun lại đống với phần tử mới bổ sung



Max-Heap-Insert

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

1. heap-size[A] ← n + 1
2. A[n + 1] ← -∞
3. Heap-Increase-Key(A, n + 1, key)



Running time: $O(\log n)$

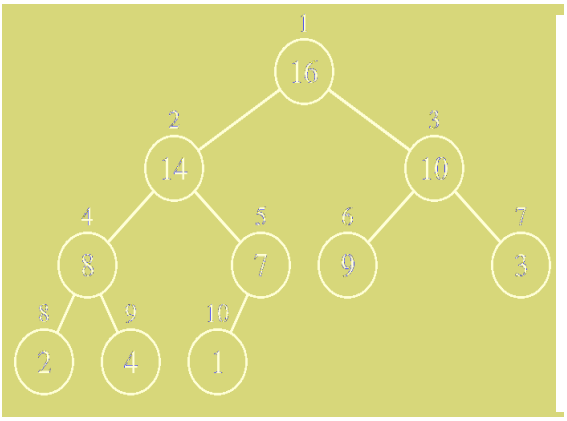
Tổng kết

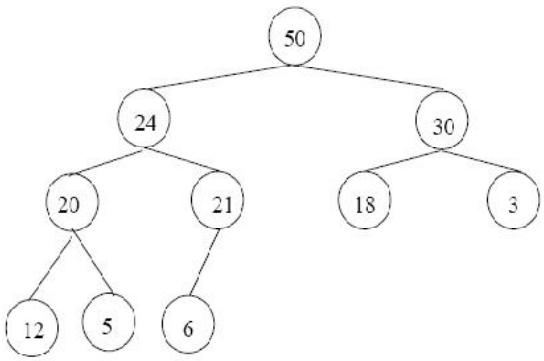
- Chúng ta có thể thực hiện các phép toán sau đây với đống:

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?

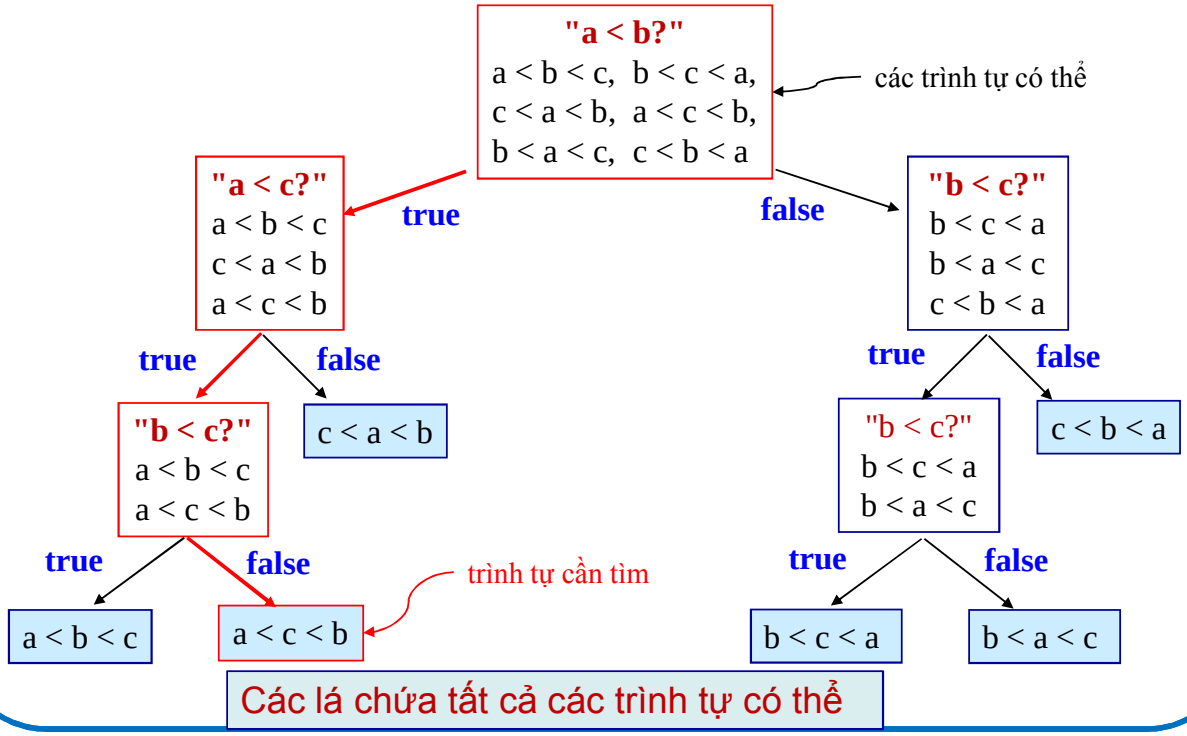
- Heapsort, Mergesort, và Quicksort có thời gian $O(n \log n)$ là các thuật toán có thời gian tính tốt nhất trong các thuật toán trình bày ở trên
- Liệu có thể phát triển được thuật toán tốt hơn không?
- Câu trả lời là: "Không, nếu như phép toán cơ bản là phép so sánh".

NỘI DUNG

Cây quyết định

- Cây quyết định là cây nhị phân thoả mãn:
  - Mỗi nút ≡ một phép so sánh "a < b"
    - cũng có thể coi tương ứng với một không gian con các trình tự
  - Mỗi cạnh ≡ rẽ nhánh theo câu trả lời (true hoặc false)
  - Mỗi lá ≡ 1 trình tự sắp xếp
  - Cây quyết định có bao nhiêu lá, nếu có N phần tử phân biệt cần sắp xếp?
    - N!, tức là tất cả các trình tự có thể
- Đối với mỗi bộ dữ liệu, chỉ có 1 lá chứa trình tự sắp xếp cần tìm

Ví dụ: Cây quyết định với N=3



Cây quyết định và sắp xếp

- Mỗi thuật toán đều có thể mô tả bởi cây quyết định
  - Tìm lá nhờ di chuyển theo cây (tức là thực hiện phép so sánh)
  - Mỗi quyết định sẽ giảm không gian tìm kiếm đi một nửa
- Thời gian tính trong tình huống tồi nhất ≥ số phép so sánh nhiều nhất phải thực hiện
  - số phép so sánh nhiều nhất cần thực hiện chính là độ dài đường đi dài nhất trên cây quyết định, tức là độ cao của cây

Cận dưới của chiều cao

- Cây nhị phân có chiều cao h có nhiều nhất bao nhiêu lá?

$$L \leq 2^h$$

- Cây quyết định có nhiều nhất bao nhiêu lá?

$$L = N!$$

- Cây quyết định với L lá có độ cao ít nhất là:

$$h \geq \log_2 L$$

- Vậy cây quyết định có độ cao:

$$h \geq \log_2(N!)$$

log(*N*!) là Ω(*N*log*N*)

$$\log(N!) = \log\big(N \cdot (N-1) \cdot (N-2) \cdots (2) \cdot (1)\big)$$

chỉ chọn $N/2$ số hạng đầu

$$= \log N + \log(N-1) + \log(N-2) + \cdots + \log 2 + \log 1$$

$$\geq \log N + \log(N-1) + \log(N-2) + \cdots + \log \frac{N}{2}$$

mỗi số hạng được chọn là $\geq \log N/2$

$$\geq \frac{N}{2} \log \frac{N}{2}$$

$$\geq \frac{N}{2}(\log N - \log 2) = \frac{N}{2} \log N - \frac{N}{2}$$

$$= \Omega(N \log N)$$

Cận dưới cho độ phức tạp của bài toán sắp xếp

- Thời gian tính của mọi thuật toán chỉ dựa trên phép so sánh là **Ω(*N* log *N*)**
- Độ phức tạp tính toán của bài toán sắp xếp chỉ dựa trên phép so sánh là **Θ(*N* log *N*)**
- Ta có thể phát triển thuật toán tốt hơn hay không nếu không hạn chế là chỉ được sử dụng phép so sánh?

NỘI DUNG

5.7. Các phương pháp sắp xếp đặc biệt

Mở đầu: Sắp xếp trong thời gian tuyến tính

Linear-time sorting

- Ta có thể làm tốt hơn sắp xếp chỉ sử dụng phép so sánh?
- Với những thông tin bổ sung (hay là giả thiết) về đầu vào, chúng ta có thể làm tốt hơn sắp xếp chỉ dựa vào phép so sánh.
- Thông tin bổ sung/Giả thiết:
  - Các số nguyên nằm trong khoảng [0..k] trong đó $k = O(n)$.
  - Các số thực phân bố đều trong khoảng [0,1)
- **Ta trình bày ba thuật toán có thời gian tuyến tính:**
  - Sắp xếp đếm (Counting-Sort)
  - Sắp xếp theo cơ số (Radix-Sort)
  - Sắp xếp đóng gói (Bucket-sort)
- Để đơn giản cho việc trình bày, ta xét việc sắp xếp các số nguyên không âm.

5.7.2. Sắp xếp đếm (Counting sort)

- **Input:** n số nguyên trong khoảng [0..k] trong đó k là số nguyên và $k = O(n)$.
- **Ý tưởng:** với mỗi phần tử x của dãy đầu vào ta xác định hạng (*rank*) của nó như là số lượng phần tử nhỏ hơn x.
- Một khi ta đã biết hạng r của x, ta có thể xếp nó vào vị trí r+1.
- **Ví dụ:** Nếu có 6 phần tử nhỏ hơn 17, ta có thể xếp 17 vào vị trí thứ 7.
- **Lặp:** khi có một loạt phần tử có cùng giá trị, ta sắp xếp chúng theo thứ tự xuất hiện trong dãy ban đầu (để có được tính ổn định của sắp xếp).

Cài đặt

```
void countSort(int a[],int b[],int c[],int k) {
// k - giá trị phần tử lớn nhất
// Đếm: b[i] - số phần tử có giá trị i
  for (int i=0; i <= k; i++) b[i] = 0;
  for (i=0; i<n; i++) b[a[i]]++;
// Tính hạng: b[i] hạng của phần tử có giá trị i
  for (i = 1; i <= k; i++)
       b[i] += b[i - 1];
// Sắp xếp
  for (i = n-1; i >= 0; i--) {
    c[b[a[i]] - 1] = a[i];
    b[a[i]] = b[a[i]] - 1;
  }
}
```


Chương trình demo

```
/*  include <iostream.h, stdlib.h,  stdio.h,
     conio.h,  process.h,  time.h  */
int a[10000], c[10000]; int n;
void print(int a[], int n) {
  for (int i = 0; i < n; i++)
        printf("%8i", a[i]);
  printf("\n");
}
void countSort(int a[], int b[], int c[], int amax);
void main() {
  int i, k; // gia tri lon nhat co the cua mang A
  clrscr();   randomize();
  printf("\ n  n = "); scanf("%i",&n);
  printf("\ n  max = "); scanf("%i",&k);
  for (i = 0; i < n; i++) a[i] = rand() % k;
  printf("\n  Day ban dau:\n"); print(a,n);
```

```
 int * b;    int amax = 0;
// Tinh phan tu lon nhat amax
  for (i = 0; i < n; i++)
       if (a[i] > amax)  amax = a[i];
  b = new int[amax];
  countSort(a, b, c, amax);
  printf("\n Day ket qua:\n");print(c,n);
// Verify result array
  c[n]=32767;
  for (i = 0; i < n; i++)
    if (c[i]>c[i+1]) {
      printf(" ??????? Loi o vi tri: %6i ", i);
      getch(); }
  printf("\n Correct!"); getch();
}
```

Phân tích độ phức tạp của sắp xếp đếm

```
void countSort(int a[],int b[],int c[],int k) {
// Đếm: b[i] - số phần tử có giá trị i
  for (int i=0; i <= k; i++) b[i] = 0;
  for (i=0; i<n; i++) b[a[i]]++;
// Tính hạng: b[i] hạng của phần tử có giá trị i
  for (i = 1; i <= k; i++) b[i] += b[i - 1];
// Sắp xếp
  for (i = n-1; i >= 0; i--) {
    c[b[a[i]] - 1] = a[i];
    b[a[i]] = b[a[i]] - 1; }
}
```

- Vòng lặp for đếm b[i] - số phần tử có giá trị i, đòi hỏi thời gian $\Theta(n+k)$.
- Vòng lặp for tính hạng đòi hỏi thời gian $\Theta(n)$.
- Vòng lặp for thực hiện sắp xếp đòi hỏi thời gian $\Theta(k)$.
- **Tổng cộng, thời gian tính của thuật toán là $\Theta(n+k)$**
- Do $k = O(n)$, nên thời gian tính của thuật toán là $\Theta(n)$, nghĩa là, trong trường hợp này nó là một trong những thuật toán tốt nhất.
- Điều đó là không xảy ra nếu giả thiết $k = O(n)$ là không được thực hiện. Thuật toán sẽ làm việc rất tồi khi $k >> n$.
- Sắp xếp đếm không có tính tại chỗ.

5.7.3. Sắp xếp theo cơ số (Radix sort)

- **Giả thiết:** Đầu vào gồm n số nguyên, mỗi số có d chữ số.
- **Ý tưởng của thuật toán:** Do thứ tự sắp xếp các số cần tìm cũng chính là thứ tự từ điển của các xâu tương ứng với chúng, nên để sắp xếp ta sẽ tiến hành d bước sau:

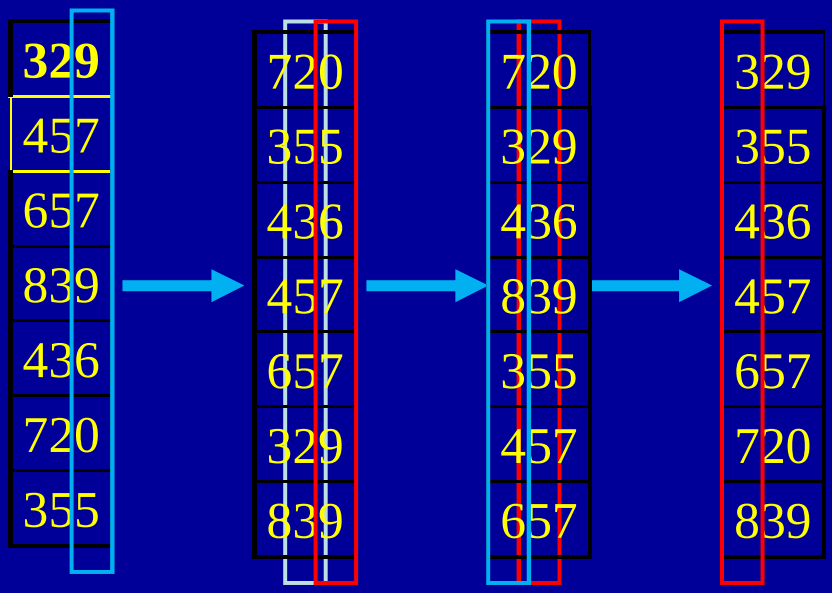
  Bước 1. Sắp xếp các số theo chữ số thứ 1,

  Bước 2. Sắp xếp các số trong dãy thu được theo chữ số thứ 2 (sử dụng sắp xếp ổn định)

  ...

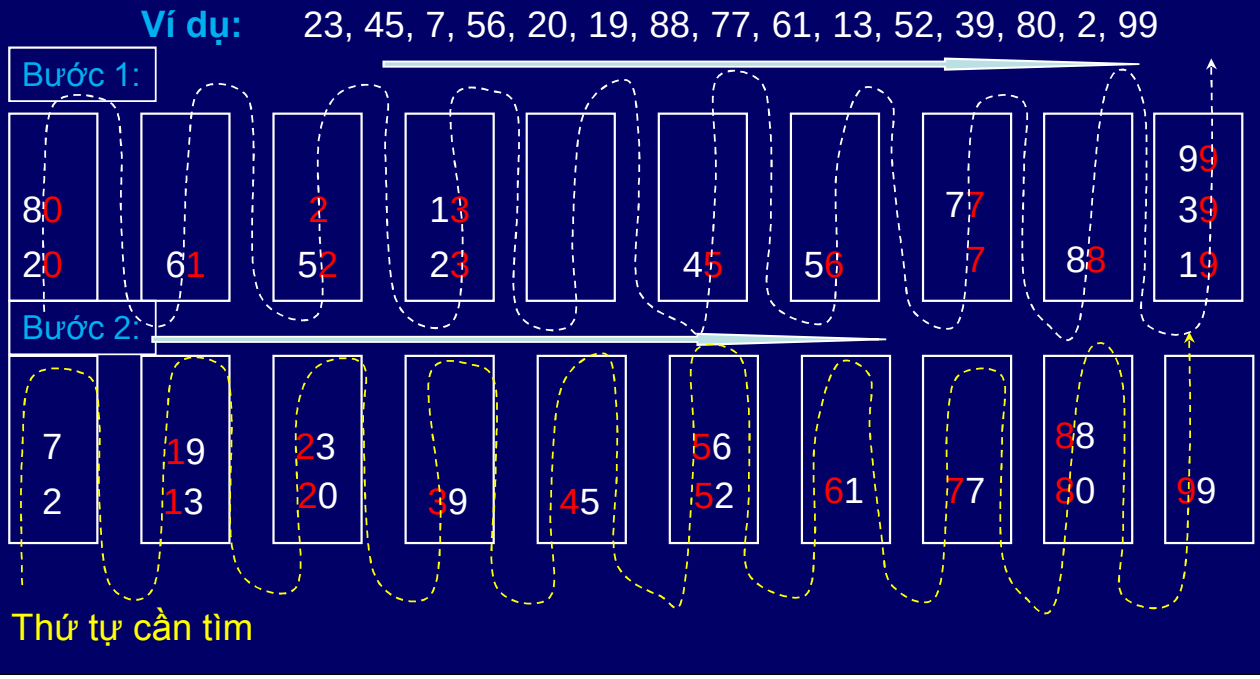
  Bước d. Sắp xếp các số trong dãy thu được theo chữ số thứ d (sử dụng sắp xếp ổn định)
- Giả sử các số được xét trong hệ đếm cơ số k. Khi đó mỗi chữ số chỉ có k giá trị , nên ở mỗi bước ta có thể sử dụng sắp xếp đếm với thời gian tính $O(n+k)$.

Ví dụ



Radix Sort

Việc sắp xếp bao gồm nhiều bước, bắt đầu từ chữ số hàng đơn vị, tiếp đến là chữ số hàng chục, hàng trăm, v.v...



Sơ đồ thuật toán Radix-Sort

Radix-Sort(A, d)

1. **for** i ← 1 to d **do**
2. Sử dụng sắp xếp ổn định để sắp xếp A theo chữ số thứ i

- **Phân tích độ phức tạp:**
- Thời gian tính: Nếu bước 2 sử dụng ***sắp xếp đếm*** thì thời gian tính của một lần lặp là $\Theta(n+k)$, do đó thời gian tính của thuật toán Radix Sort là $T(n) = \Theta(d(n+k))$. Thời gian này sẽ là $\Theta(n)$ nếu d là hằng số và $k = O(n)$
- ***Chú ý:*** Để thực hiện sắp xếp ở mỗi lần lặp của thuật toán, phải sử dụng thuật toán sắp xếp ổn định, nếu không sẽ không có kết quả đúng.

Thuật toán sắp xếp theo cơ số nhị phân

- Ta xét các số *nguyên không âm* 32-*bit* và ta sẽ chuyển chúng về các số có bốn chữ số trong hệ đếm cơ số 256.
- Giả sử

$$p = a_{31}\times 2^{31} + a_{30}\times 2^{30} + \ldots + a_2\times 2^2 + a_1\times 2^1 + a_0\times 2^0$$

trong đó a_i là bằng 0 hoặc 1.

- Khi đó trong hệ đếm cơ số 256 ta có:

$$p = b_3\times 256^3 + b_2\times 256^2 + b_1\times 256^1 + b_0\times 256^0$$

trong đó

$$b_3 = a_{31}2^7 + a_{30}2^6 + a_{29}2^5 + a_{28}2^4 + a_{27}2^3 + a_{26}2^2 + a_{25}2^1 + a_{24}2^0,$$
$$b_2 = a_{23}2^7 + a_{22}2^6 + a_{21}2^5 + a_{20}2^4 + a_{19}2^3 + a_{18}2^2 + a_{17}2^1 + a_{16}2^0,$$
$$b_1 = a_{15}2^7 + a_{14}2^6 + a_{13}2^5 + a_{12}2^4 + a_{11}2^3 + a_{10}2^2 + a_9 2^1 + a_8 2^0,$$
$$b_0 = a_7 2^7 + a_6 2^6 + a_5 2^5 + a_4 2^4 + a_3 2^3 + a_2 2^2 + a_1 2^1 + a_0 2^0.$$

Tính các chữ số

- Nếu số nguyên n nằm trong phạm vi $[0,2^{31})$, ta có thể tính các chữ số b_0 , b_1 , b_2 , b_3 của nó nhờ sử dụng các phép toán & (bitwise) và >> (dịch phải - right shift) được cung cấp sẵn trong C (các phép toán này thực hiện rất hiệu quả) như sau:
  - $b_0 = n \;\&\; 255$
  - $b_1 = (n >> 8) \;\&\; 255$
  - $b_2 = (n >> 16) \;\&\; 255$
  - $b_3 = (n >> 24) \;\&\; 255$

Cài đặt trên C

```
void radixsort(long a[], int n){
  int i, j, shift;
  long temp[1000];
  int bin_size[256], first_ind[256];

  for (shift=0; shift<32; shift+=8) {
   /* Tính kích thước mỗi cụm và sao chép các
      phần tử của mảng a vào mảng temp */
    for (i=0; i<256; i++) bin_size[i]=0;
    for (j=0; j<n; j++) {
        i=(a[j]>>shift)&255;
        bin_size[i]++;
        temp[j]=a[j];
    }
```

```
/* Tính chỉ số vị trí bắt đầu của mỗi cụm */
  first_ind[0]=0;
  for (i=1; i<256; i++)
   first_ind[i]=first_ind[i-1]+bin_size[i-1];

    /* Sao chép phần tử của mảng temp
    vào cụm của nó trong mảng a */
    for (j=0; j<n; j++) {
        i=(temp[j]>>shift)&255;
        a[first_ind[i]]=temp[j];
        first_ind[i]++; }
  } // end for shift
} // end radixsort
```

Chương trình chính

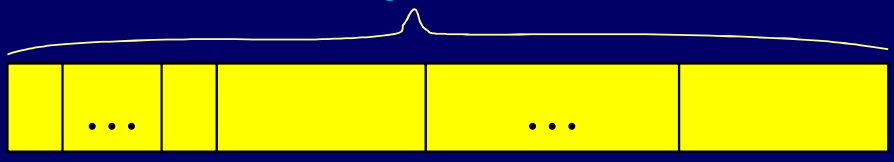
```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <conio.h>

void print(long a[], int n) {
  for (int i = 0; i < n; i++)
     printf("%8i", a[i]);
  printf("\n");
}
```

```
int main () {
   long data[1000];
   int i, N;
   printf("\ Nhap so luong phan tu n = ");
   scanf("%i",&N);
   // Randomdata
   for ( i=0; i<N; i++ ) data[i]=rand();
   printf("\n Day dau vao:\n"); print(data, N);
   radixsort (data, N);
   printf("\n Day duoc sap thu tu:\n"); print(data, N);
   data[N]=9999999;
   for ( i=0; i<N; i++ )
      if (data[i]>data[i+1])  printf("\n ??????
      ERROR:\n");
   getch();
}
```


Chọn cơ số như thế nào?

Khoá gồm b bit:



chữ số gồm r-bit

b/r chữ số

Mỗi chữ số trong khoảng $0 -- 2^{r-1}$

b/r bước, mỗi bước đòi hỏi thời gian $\Theta(n+2^r)$ (áp dụng Sắp xếp đếm).

Thời gian tính là $\Theta((b/r)(n + 2^r))$.

Nếu $b \le \log n$, chọn $r = b$ ⟹ $\Theta(n)$

Nếu $b > \log n$, chọn $r = \log n$ ⟹ $\Theta(bn/\log n)$

Radix-Sort tổng quát

- Cho n số, mỗi số có b bit và số $r \le b$. Khi đó Radix-Sort đòi hỏi thời gian $\Theta((b/r)(n+2^r))$:
  - Chọn $d=b/r$ chữ số, mỗi chữ số gồm r bit có giá trị trong khoảng $[0..2^r-1]$.
  - Vì thế ở mỗi bước có thể sử dụng Counting-Sort với $k=2^r-1$, đòi hỏi thời gian $\Theta(n+k)$,
  - Tất cả phải thực hiện d bước, do đó thời gian tổng cộng là $\Theta(d(n+2^r))$, hay $\Theta((b/r)(n+2^r))$.
- Với các giá trị của n và b cho trước, ta có thể chọn $r \le b$ để đạt cực tiểu của $\Theta((b/r)(n+2^r))$.
- Nếu $b \le \log n$, chọn $r = \log n$ ta thu được thuật toán với thời gian $\Theta(n)$.
- Nếu $b > \log n$, chọn $r = \log n$ ta thu được thuật toán với thời gian $\Theta(bn/\log n)$.

5.7.4. Sắp xếp phân cụm (Bucket sort)

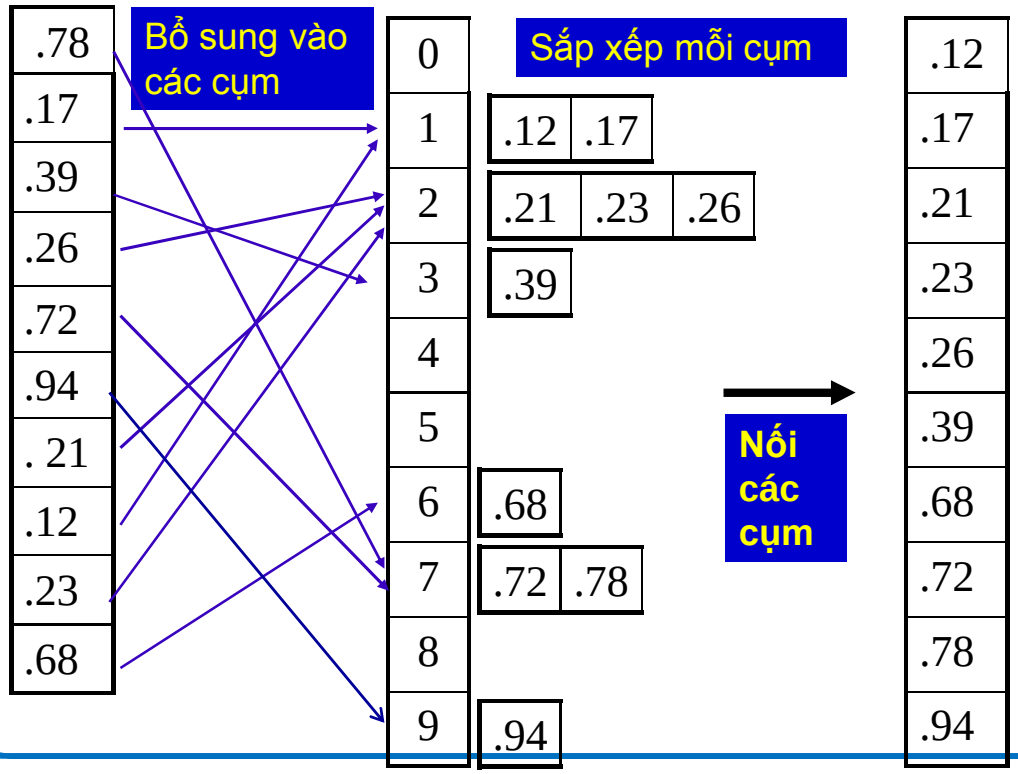
Giả thiết: Đầu vào gồm n số thực có phân bố đều trong khoảng $[0..1)$ (các số có xác suất xuất hiện như nhau)

Ý tưởng của thuật toán:

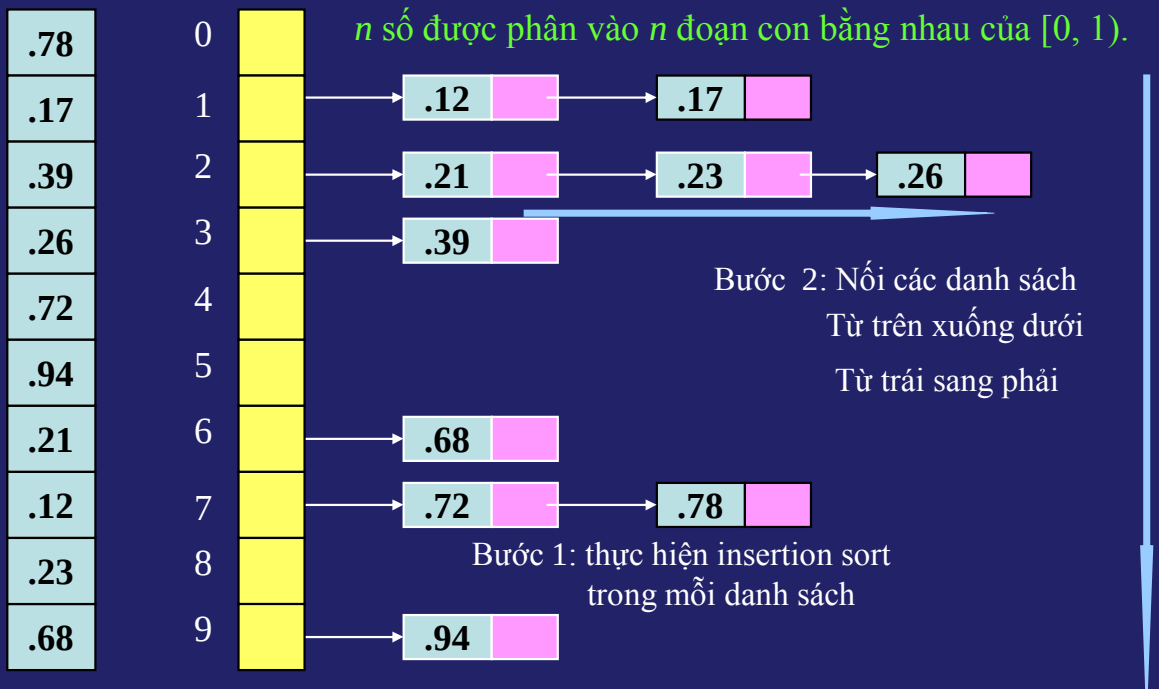
- Chia đoạn $[0..1)$ ra làm n cụm (buckets)

  $0, 1/n, 2/n. \ldots (n-1)/n.$
- Đưa mỗi phần tử a_j vào đúng cụm của nó $i/n \le a_j < (i+1)/n$.
- Do các số là phân bố đều nên không có quá nhiều số rơi vào cùng một cụm.
- Nếu ta chèn chúng vào các cụm (sử dụng sắp xếp chèn) thì các cụm và các phần tử trong chúng đều được sắp xếp theo đúng thứ tự.

Ví dụ: Sắp xếp phân cụm, n=10, các cụm là 0, 0.1, 0.2, ..., 0.9



Ví dụ



Sơ đồ thuật toán Bucket-Sort

$A[0..n\text{-}1]$ là mảng đầu vào
$B[0], B[1], \ldots, B[n-1]$ là các danh sách cụm

Bucket-Sort(A)

1. n = length(A)
2. **for** $i = 0$ **to** n-1
3. **do** Bổ sung $A[i]$ vào danh sách $B[\lfloor n*A[i] \rfloor]$
4. **for** $i = 0$ **to** $n-1$
5. **do** Insertion-Sort($B[i]$)
6. Nối các danh sách $B[0], B[1], \ldots B[n-1]$ theo thứ tự

Phân tích thời gian tính của Bucket-Sort

- Tất cả các dòng, ngoại trừ dòng 5 (Insertion-Sort), đòi hỏi thời gian $O(n)$ trong tình huống tồi nhất.
- Trong *tình huống tồi nhất*, $O(n)$ số được đưa vào cùng một cụm, do đó thuật toán có thời gian tính $O(n^2)$ trong tình huống tồi nhất.
- Tuy nhiên, trong *tình huống trung bình*, chỉ có một số lượng hằng số phần tử của dãy cần sắp xếp rơi vào trong mỗi cụm và vì thế thuật toán có thời gian tính trung bình là $O(n)$ (ta công nhận sự kiện này).
- ***Mở rộng:*** sử dụng các sơ đồ chỉ số hoá khác nhau để phân cụm các phần tử.

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?