Big-O Blue

LECTURE 07

HEAP

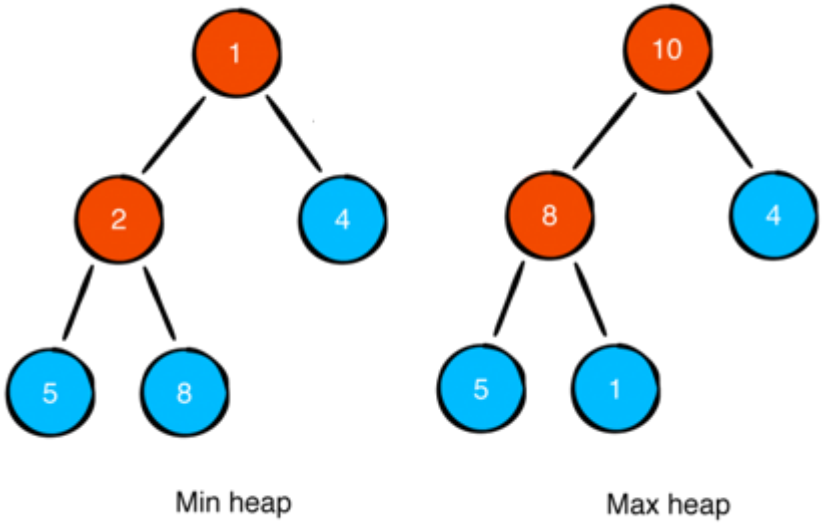
Phạm Nguyễn Sơn Tùng

Email: sontungtn@gmail.com

Heap

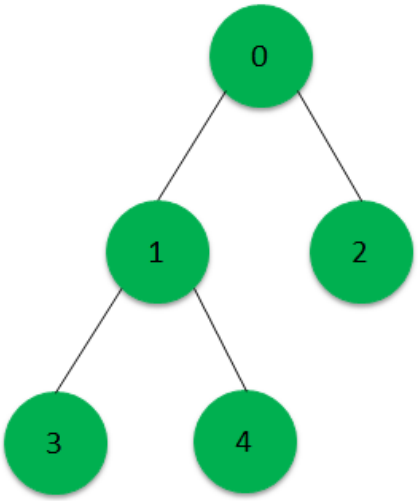
Heap (Đống) là cấu trúc cây nhị phân hoàn chỉnh (*complete binary tree)*. Có 2 loại Heap:

- Min-heap: Mỗi node cha đều có giá trị nhỏ hơn hoặc bằng node con của nó.
- Max-heap: Mỗi node cha đều có giá trị lớn hơn hoặc bằng node con của nó.

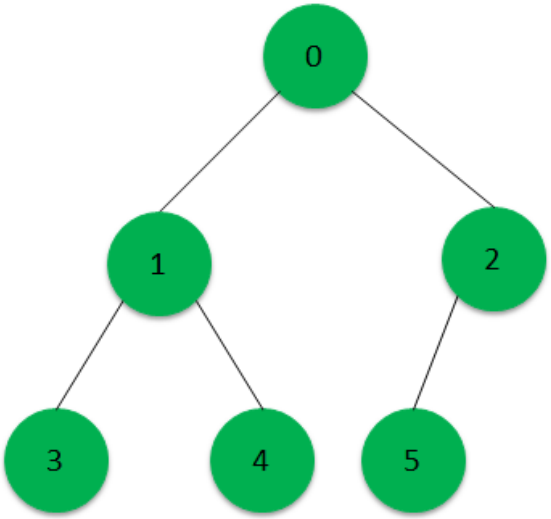


Các loại cây nhị phân

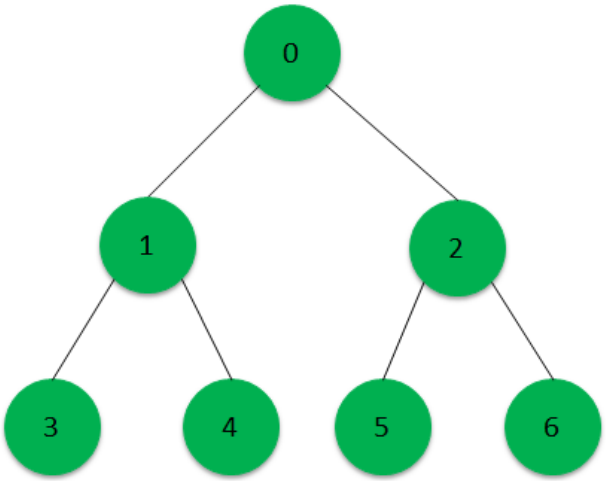
full binary tree



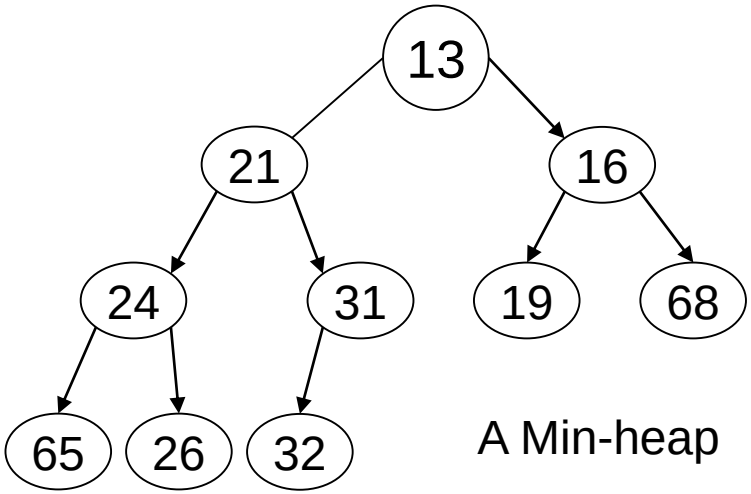
complete binary tree



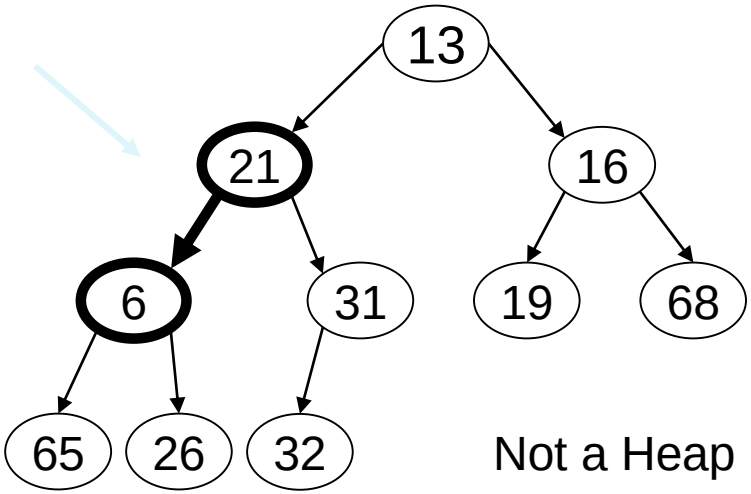
perfect binary tree



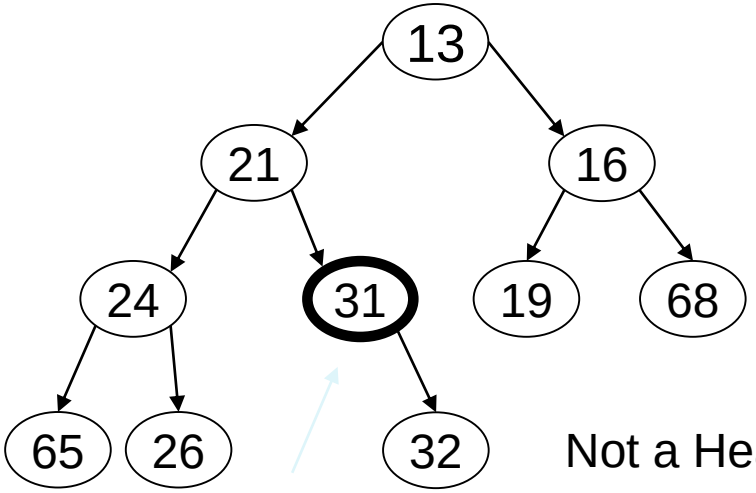
Phân biệt Heap



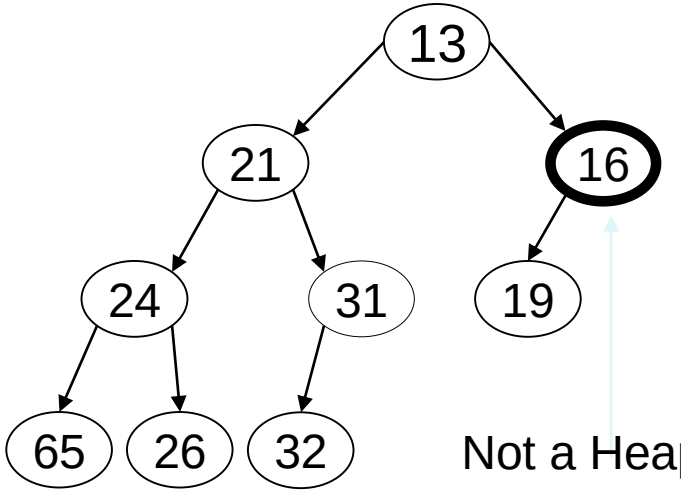
A Min-heap



Not a Heap



Not a Heap



Not a Heap

Ứng dụng & độ phức tạp của Heap

Big-O Blue

Ứng dụng:

1. Heap dùng để cài đặt và giải quyết bài toán liên quan đến hàng đợi ưu tiên **"priority queue"**.
2. Dùng để tối ưu hóa các thuật toán Dijkstra, Prim…
3. Thuật toán sắp xếp Heapsort.

Độ phức tạp của các thao tác trên Heap:

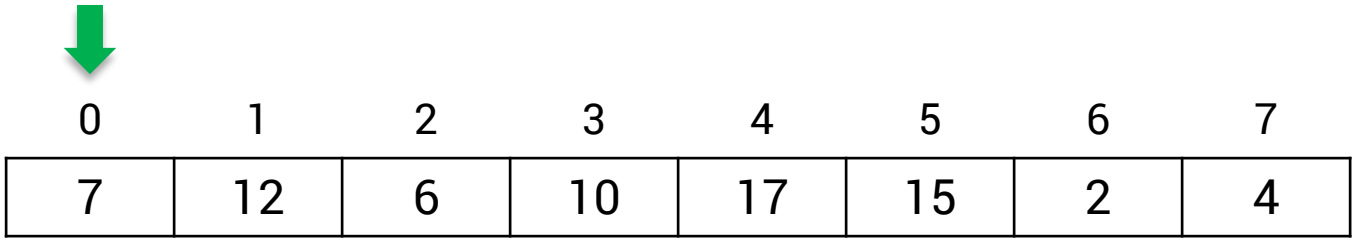
0. Xây dựng cây Heap từ mảng: **O(n).**
1. Tìm phần tử lớn nhất/nhỏ nhất trên Heap: **O(1).**
2. Thêm một phần tử vào Heap: **O(log(n)).**
3. Xóa một phần tử trong Heap: **O(log(n)).**

** Với n là số lượng phần tử của mảng.*

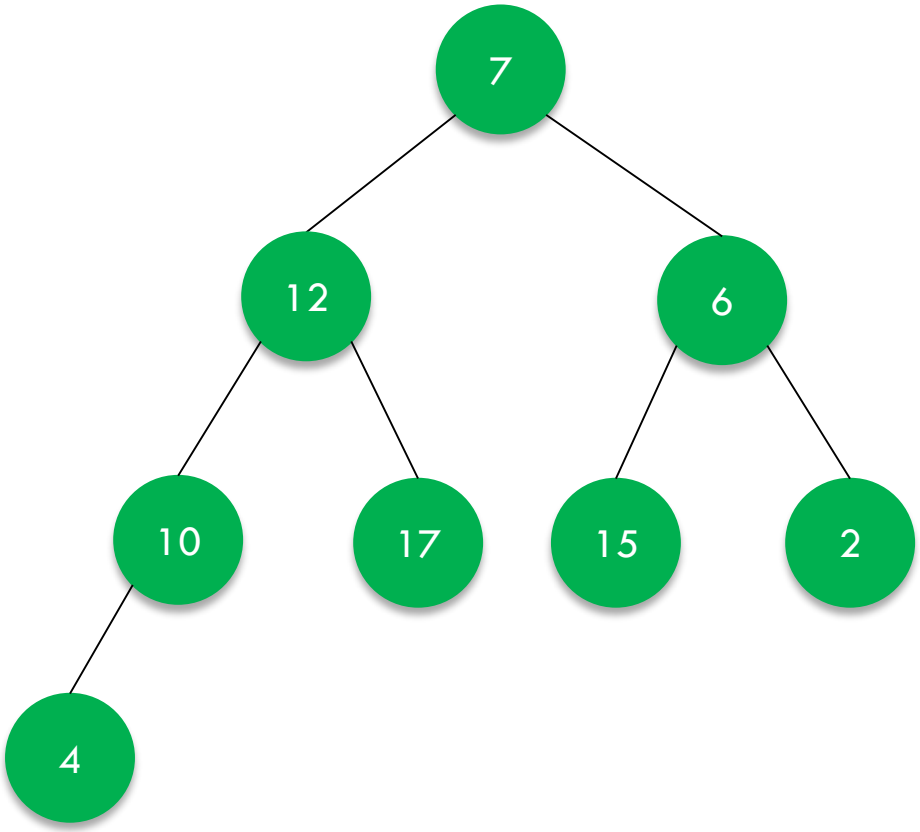
MINH HỌA CÁC THAO TÁC CƠ BẢN TRÊN HEAP

0. Xây dựng cây Min-heap

Bước 0: Chuẩn bị dữ liệu.



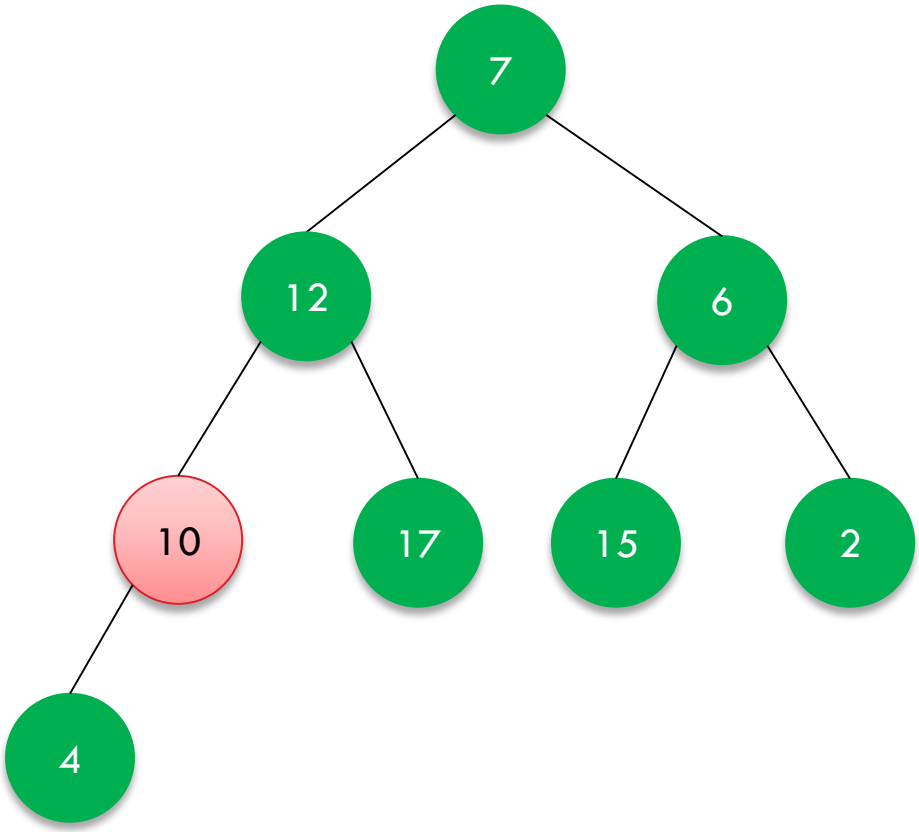
0	1	2	3	4	5	6	7
7	12	6	10	17	15	2	4



0. Xây dựng cây Min-heap

Bước 1: Tìm đến node có vị trí: n/2 – 1 = 8/2 – 1 = 3 (node có giá trị 10)

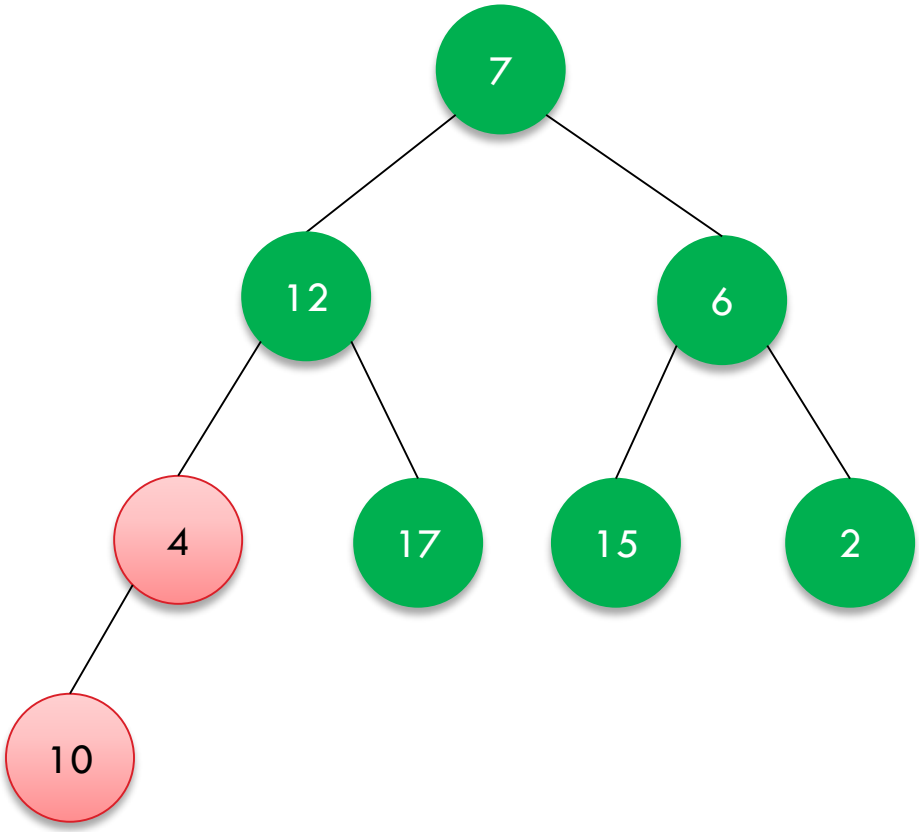
0	1	2	3	4	5	6	7
7	12	6	10	17	15	2	4



0. Xây dựng cây Min-heap

Bước 1: Đổi chỗ node vừa tìm với node con có giá trị nhỏ nhất của nó.

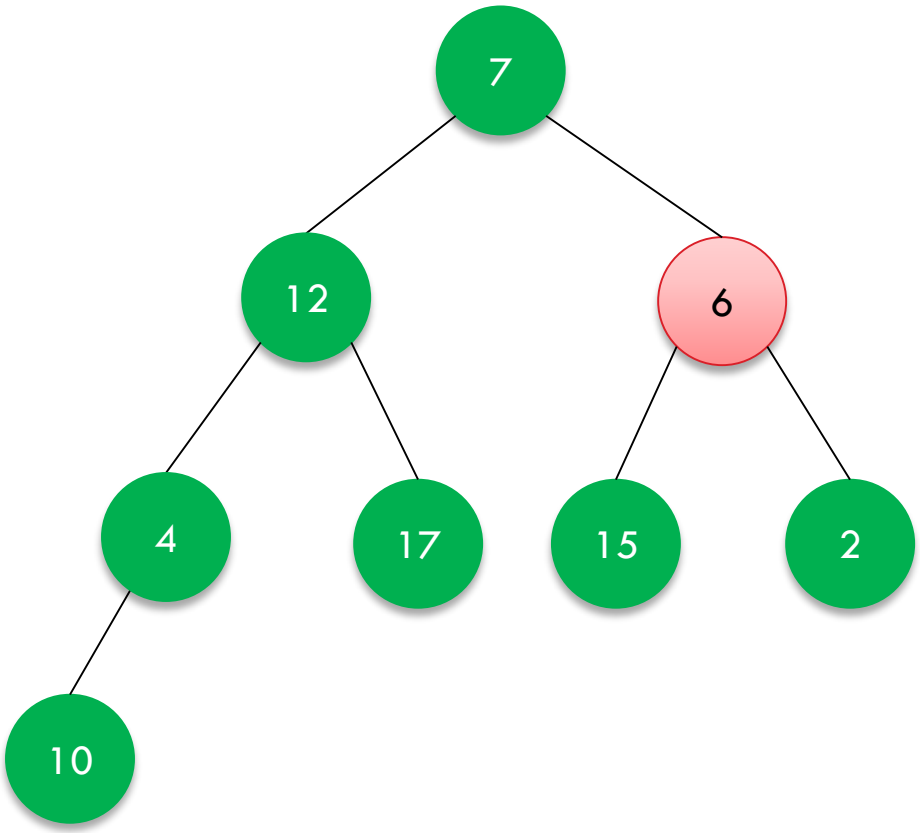
0	1	2	**3**	4	5	6	**7**
7	12	6	**4**	17	15	2	**10**



0. Xây dựng cây Min-heap

Bước 2: Di chuyển lên node phía trên, vị trí 2 (node có giá trị 6).

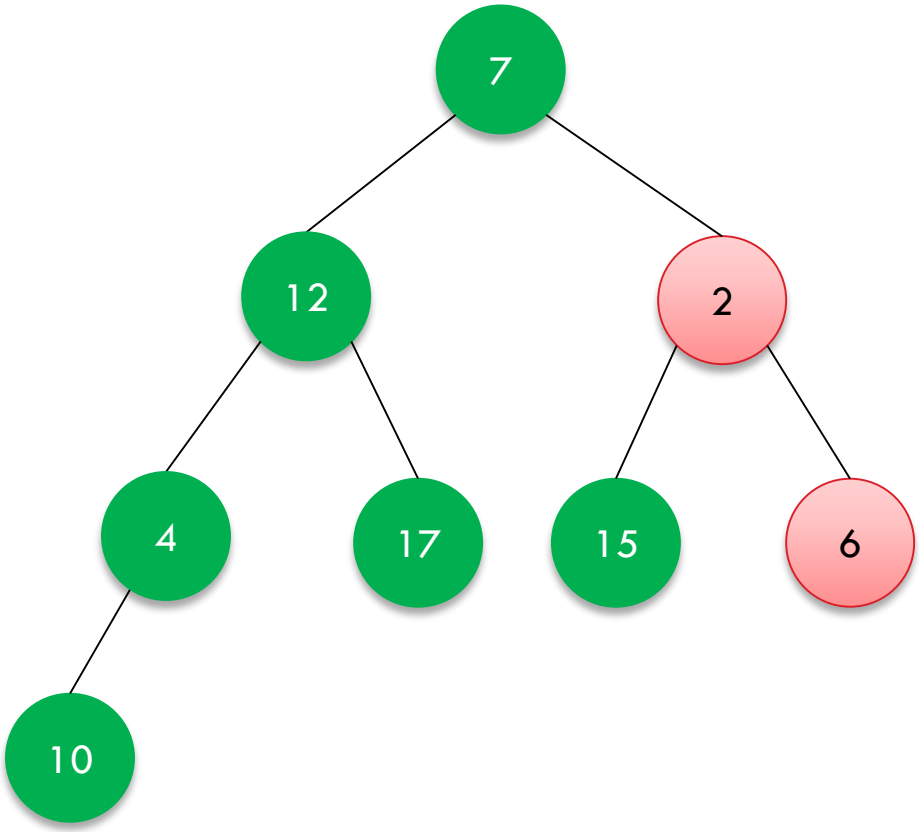
0	1	**2**	3	4	5	6	7
7	12	**6**	4	17	15	2	10



0. Xây dựng cây Min-heap

Bước 2: Tìm node con có giá trị nhỏ nhất và đổi chỗ.

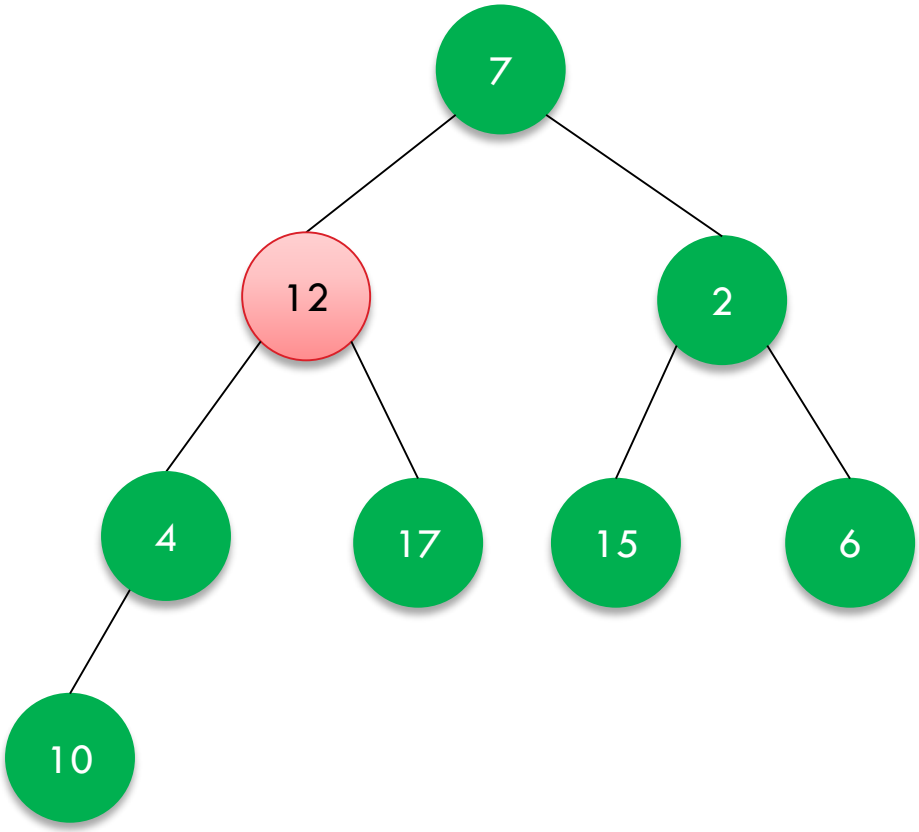
0	1	**2**	3	4	5	**6**	7
7	12	**2**	4	17	15	**6**	10



0. Xây dựng cây Min-heap

Bước 3: Di chuyển lên node phía trên, vị trí 1 (node có giá trị 12).

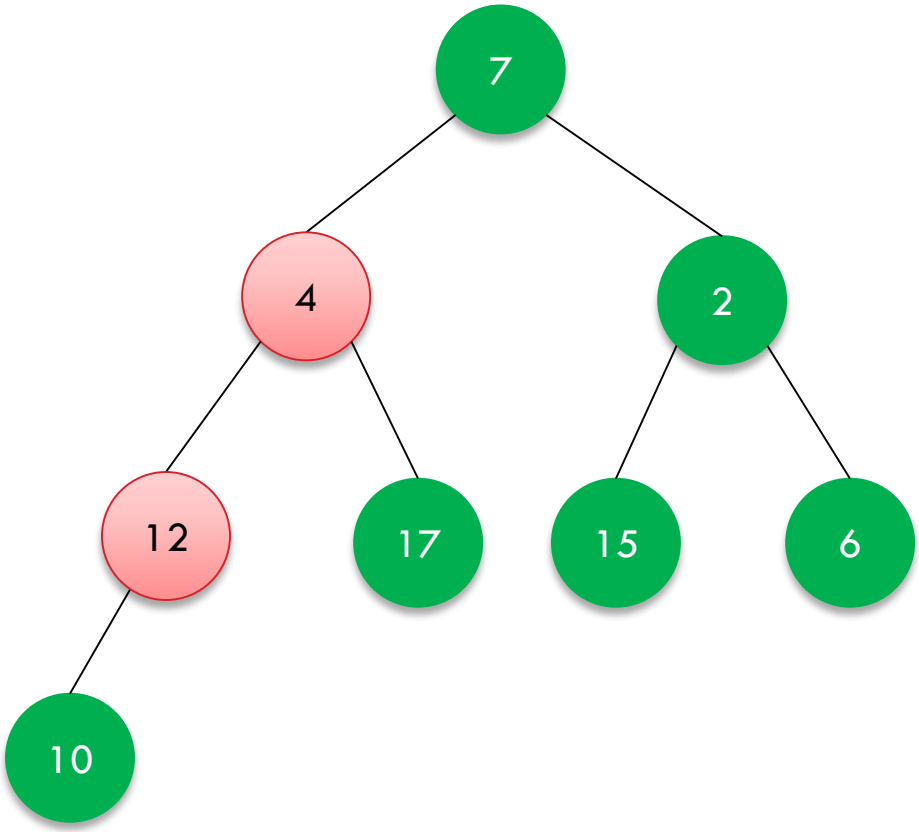
0	**1**	2	3	4	5	6	7
7	**12**	2	4	17	15	6	10



0. Xây dựng cây Min-heap

Bước 3: Tìm node con có giá trị nhỏ nhất và đổi chỗ.

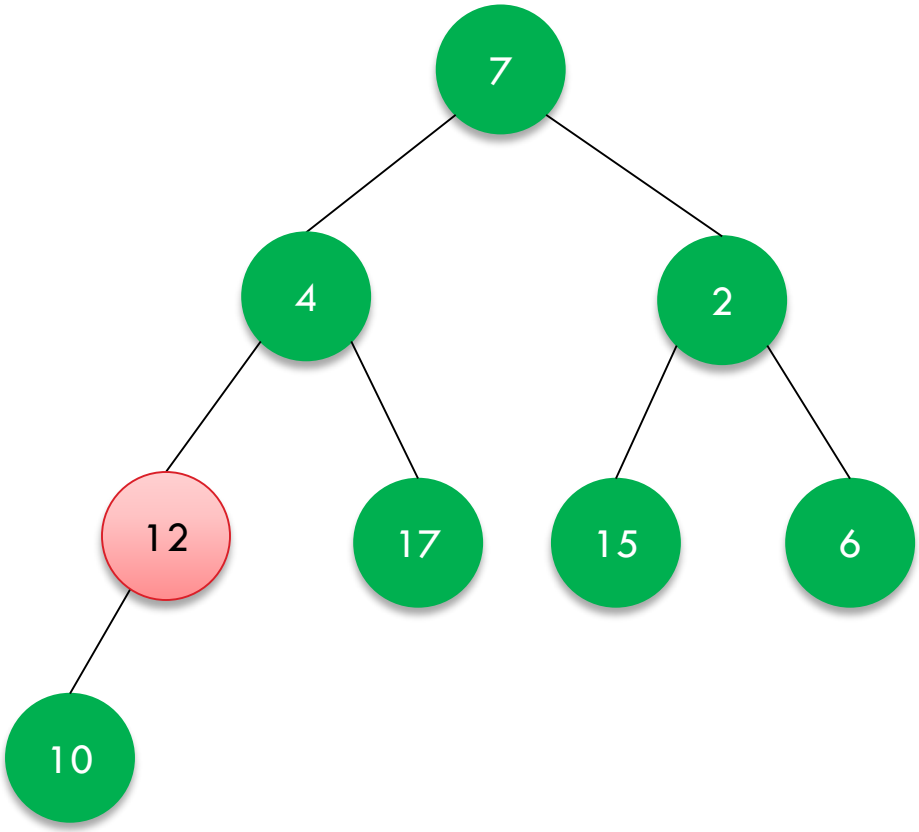
0	**1**	2	**3**	4	5	6	7
7	**4**	2	**12**	17	15	6	10



0. Xây dựng cây Min-heap

Bước 3: Node 12 chưa đúng vị trí, cần cân chỉnh cây lại.

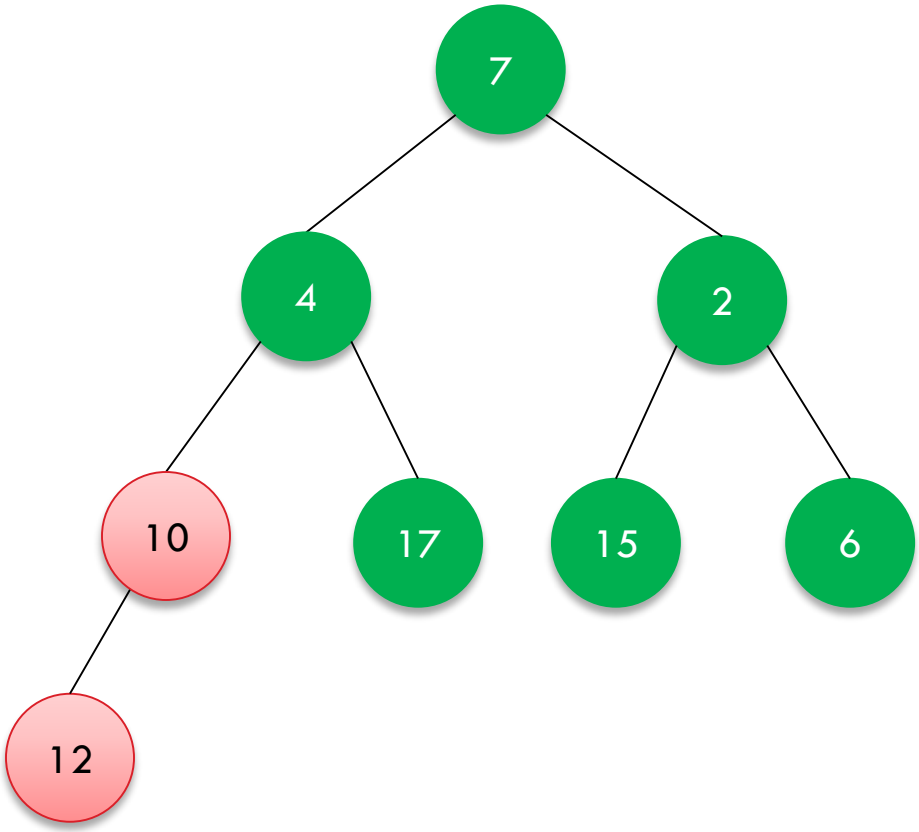
0	1	2	**3**	4	5	6	7
7	4	2	**12**	17	15	6	10



0. Xây dựng cây Min-heap

Bước 3: Đổi chỗ node 12 và node 10 với nhau.

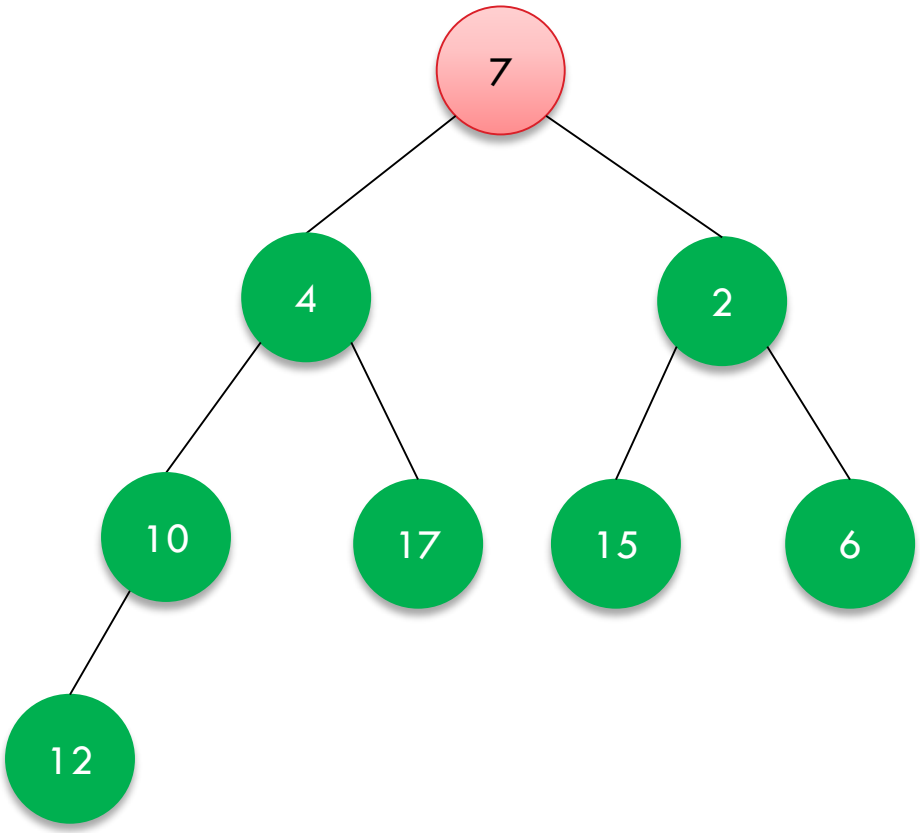
0	1	2	**3**	4	5	6	**7**
7	4	2	**10**	17	15	6	**12**



0. Xây dựng cây Min-heap

Bước 4: Di chuyển lên node phía trên, vị trí 0 (node có giá trị 7).

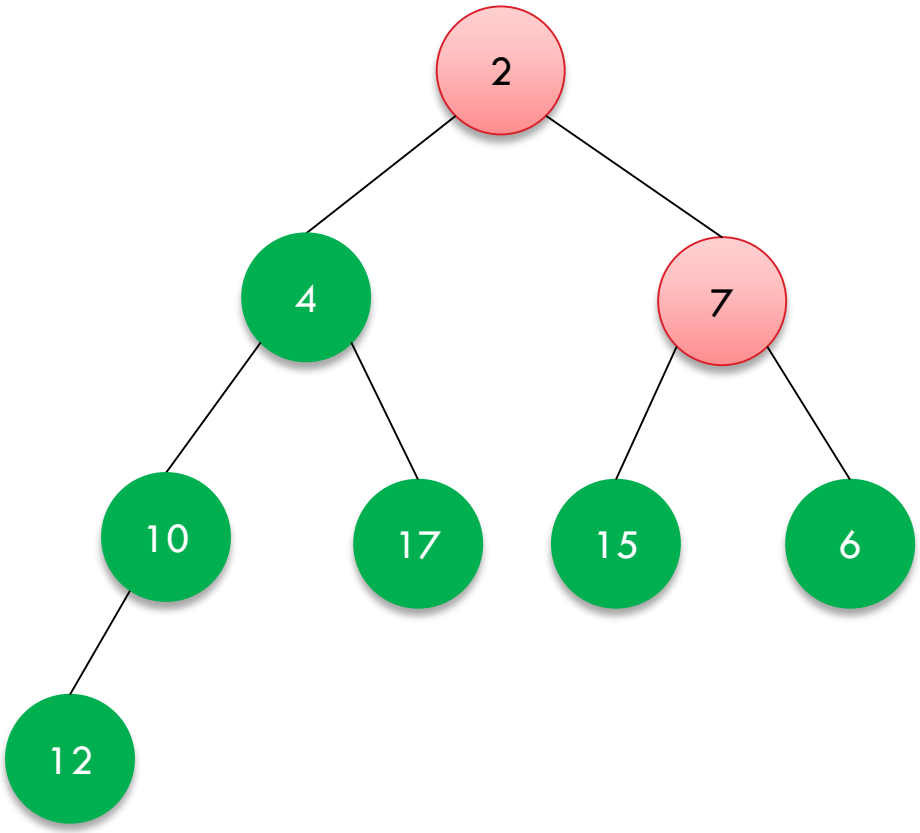
0	1	2	3	4	5	6	7
7	4	2	10	17	15	6	12



0. Xây dựng cây Min-heap

Bước 4: Tìm node con có giá trị nhỏ nhất và đổi chỗ.

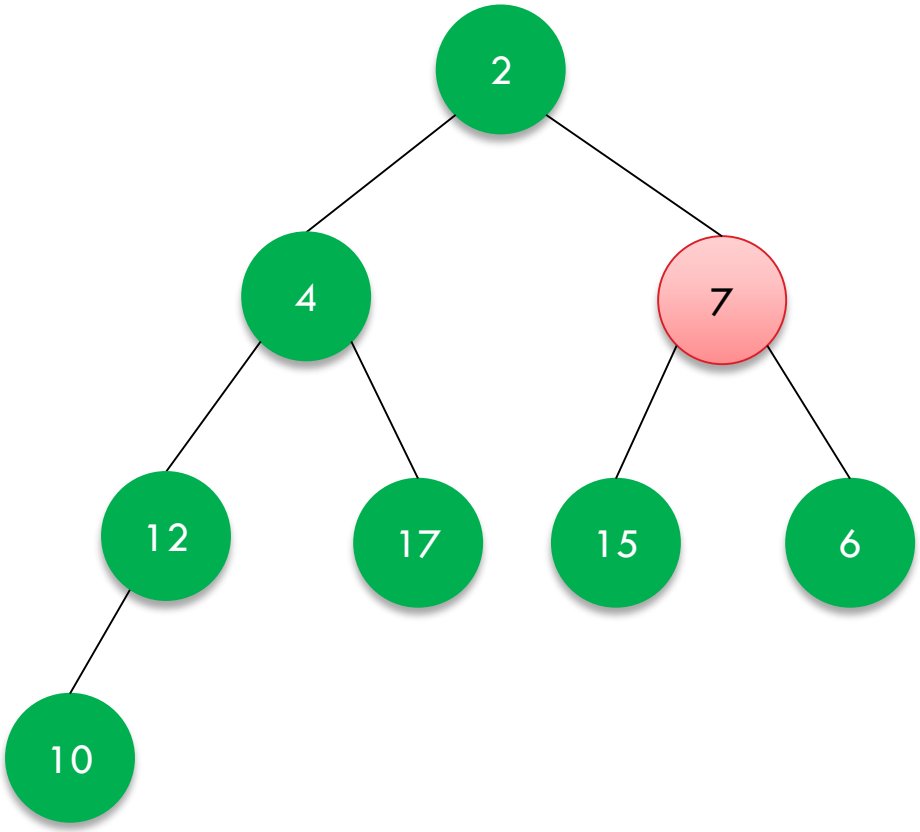
0	1	2	3	4	5	6	7
2	4	**7**	10	17	15	6	12



0. Xây dựng cây Min-heap

Bước 5: Tìm node không đúng vị trí và thay đổi, vị trí 2 (node có giá trị 7).

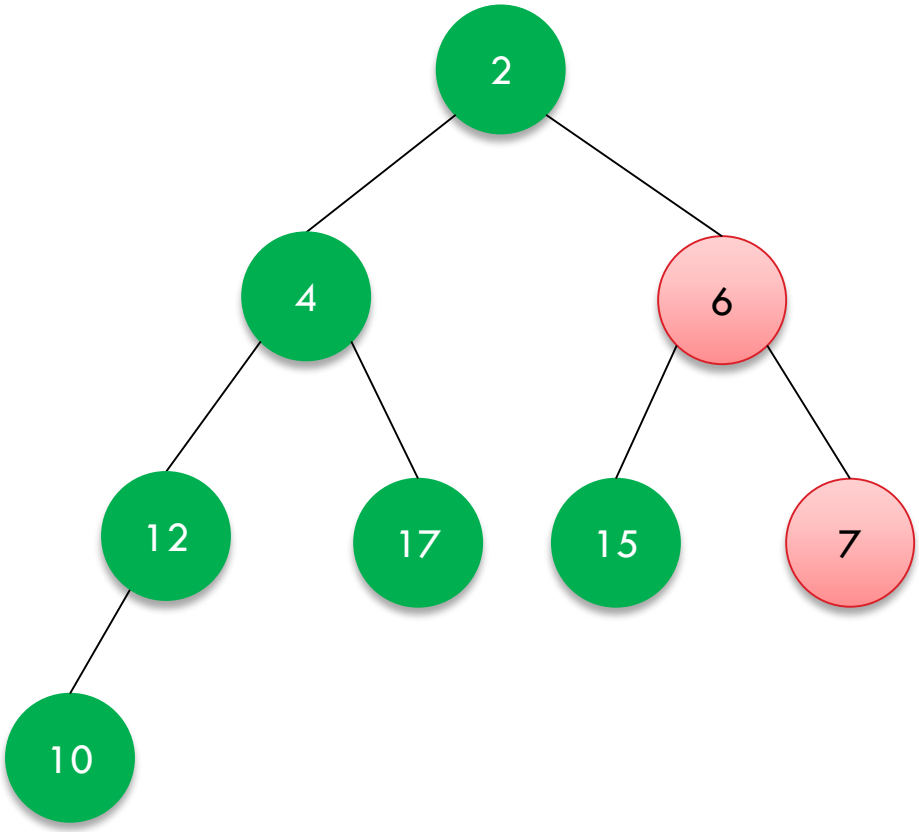
0	1	**2**	3	4	5	6	7
2	4	**7**	10	17	15	6	12



0. Xây dựng cây Min-heap

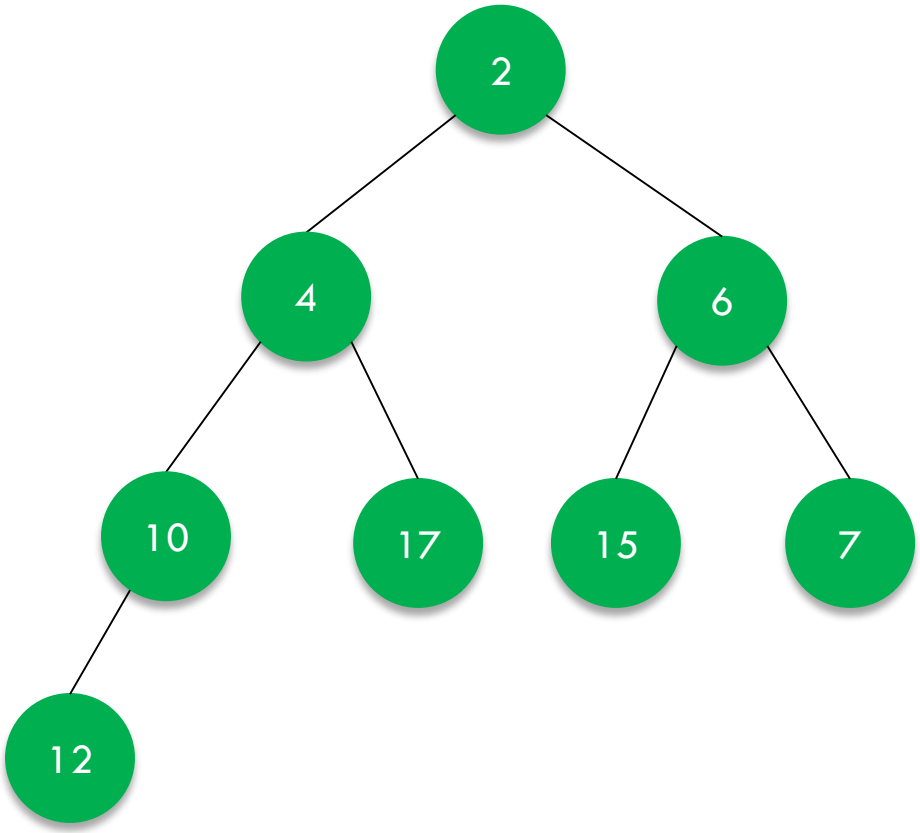
Bước 5: Tìm node con có giá trị nhỏ nhất và đổi chỗ.

0	1	**2**	3	4	5	**6**	7
2	4	**6**	10	17	15	**7**	12

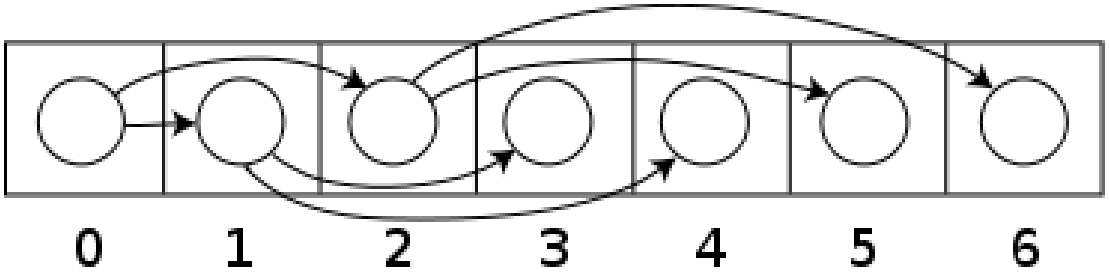


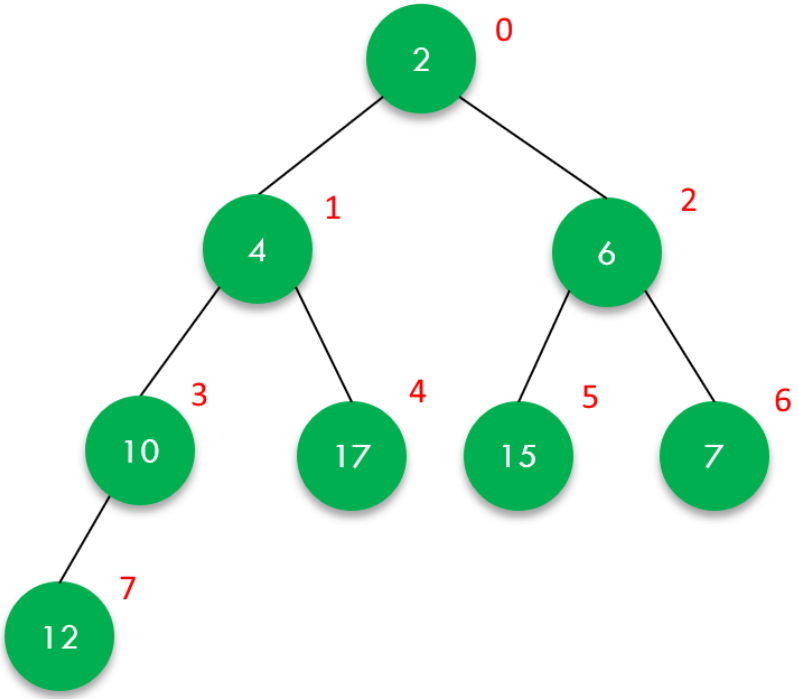
0. Xây dựng cây Min-heap

0	1	2	3	4	5	6	7
2	4	6	10	17	15	7	12



Lưu cây Heap trên mảng

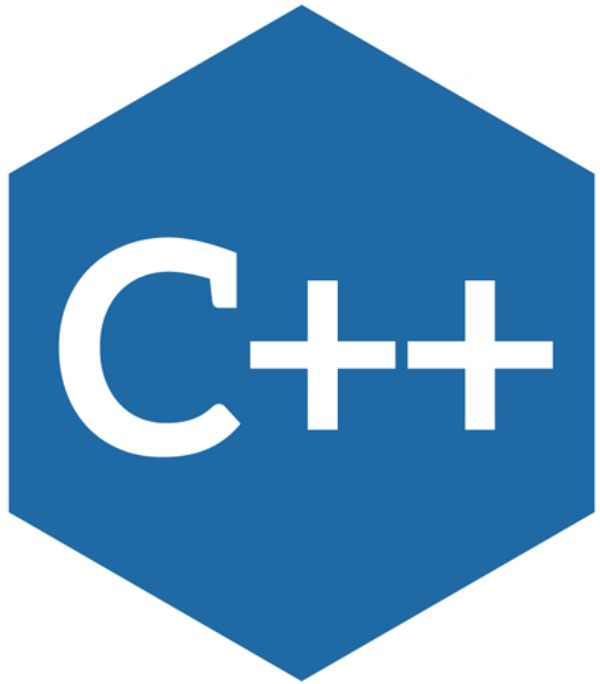




Công thức tính vị trí node con của node thứ index:

- Left = index*2 + 1.
- Right = index*2 + 2.

MÃ NGUỒN MINH HỌA BẰNG C++



Source code xây dựng cây Min-heap

Hàm chuẩn hóa cây thành Min-heap.

```cpp
vector<int> h;
void minHeapify(int i)
{
    int smallest = i;
    int left = 2*i + 1;
    int right = 2*i + 2;
    if (left < h.size() && h[left] < h[smallest])
        smallest = left;
    if (right < h.size() && h[right] < h[smallest])
        smallest = right;
    if (smallest != i)
    {
        swap(h[i], h[smallest]);
        minHeapify(smallest);
    }
}
```

Source code xây dựng cây Min-heap

Hàm xây dựng Min-Heap từ mảng h[] có n phần tử: thực hiện chuẩn hóa cây từ vị trí cuối cùng có node lá.

```
17. void buildHeap(int n)
18. {
19.       for (int i = n / 2 - 1; i >= 0; i--)
20.            minHeapify(i);
21. }
```

Hàm main chương trình.

```
22. int main()
23. {
24.      h = { 7, 12, 6, 10, 17, 15, 2, 4 };
25.      buildHeap(h.size());
26.      return 0;
27. }
```

MÃ NGUỒN MINH HỌA BẰNG PYTHON

Source code xây dựng cây Min-heap

Hàm chuẩn hóa cây thành Min-heap.

```
1.  def minHeapify(i):
2.      smallest = i
3.      left = 2*i + 1
4.      right = 2*i + 2
5.      if left < len(h) and h[left] < h[smallest]:
6.          smallest = left
7.      if right < len(h) and h[right] < h[smallest]:
8.          smallest = right
9.      if smallest != i:
10.         h[i], h[smallest] = h[smallest], h[i]
11.         minHeapify(smallest)
```

Source code xây dựng cây Min-heap

Hàm xây dựng Min-Heap từ mảng h[] có n phần tử: thực hiện chuẩn hóa cây từ vị trí cuối cùng có node lá.

```
12. def buildHeap(n):
13.     for i in range(n//2 - 1, -1, -1):
14.         minHeapify(i)
```

Hàm main chương trình.

```
15. if __name__ == '__main__':
16.     h = [7, 12, 6, 10, 17, 15, 2, 4]
17.     buildHeap(len(h))
18.     print(h)
```

MÃ NGUỒN MINH HỌA BẰNG JAVA

Source code xây dựng cây Min-heap

Hàm hoán đổi 2 node với nhau.

```
1. import java.lang.reflect.Array;
2. import java.util.*;
3. public class Main {
4.     private static ArrayList<Integer> h;
5.     private static void swap(int i, int j) {
6.         int x = h.get(i);
7.         h.set(i, h.get(j));
8.         h.set(j, x);
9.     }
```

Source code xây dựng cây Min-heap

Hàm chuẩn hóa cây thành Min-heap.

```
10.     private static void minHeapify(int i) {
11.         int smallest = i;
12.         int left = 2 * i + 1;
13.         int right = 2 * i + 2;
14.         if (left < h.size() && h.get(left) < h.get(smallest))
15.             smallest = left;
16.         if (right < h.size() && h.get(right) < h.get(smallest))
17.             smallest = right;
18.         if (smallest != i) {
19.             swap(i, smallest);
20.             minHeapify(smallest);
21.         }
22.     }
```

Source code xây dựng cây Min-heap

Hàm xây dựng Min-Heap từ mảng h[] có n phần tử: thực hiện chuẩn hóa cây từ vị trí cuối cùng có node lá.

```
23.     private static void buildHeap(int n) {
24.         for (int i = n / 2 - 1; i >= 0; i--) {
25.             minHeapify(i);
26.         }
27.     }
```

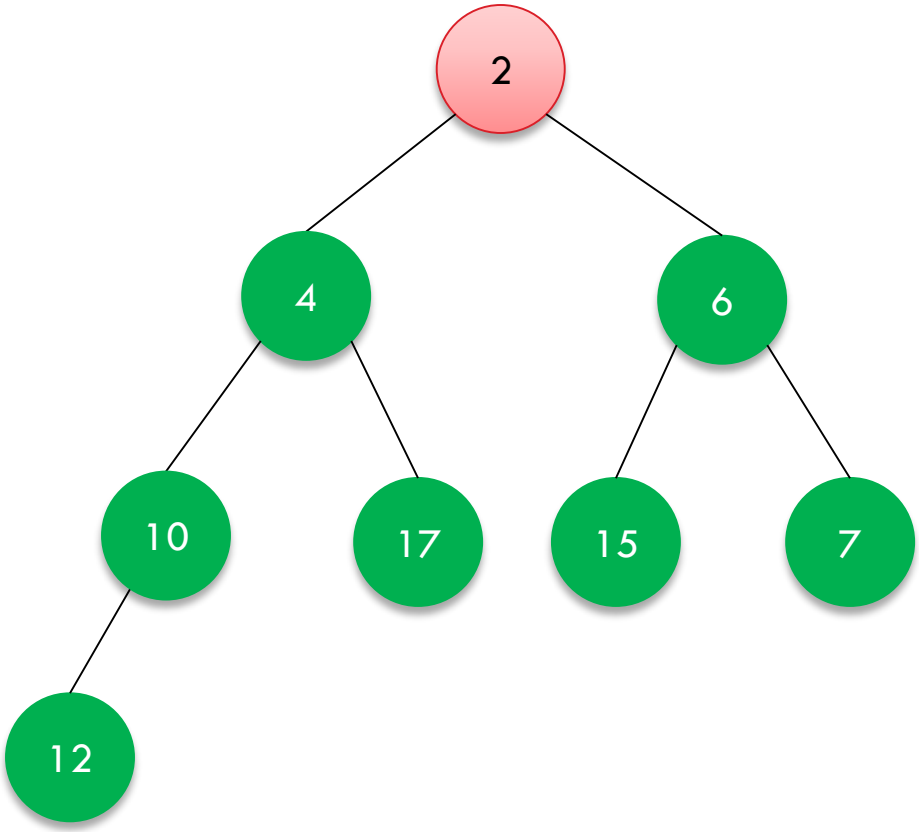
Hàm main chương trình.

```
28.     public static void main(String[] args) {
29.         h = new ArrayList<>(Arrays.asList(7, 12, 6, 10, 17, 15, 2, 4));
30.         buildHeap(h.size());
31.     }
32. }
```

1. Tìm phần tử nhỏ nhất của Min-Heap

Trả về phần tử tại vị trí đầu tiên là vị trí chứa giá trị nhỏ nhất.

0	1	2	3	4	5	6	7
2	4	6	10	17	15	7	12



Source Code tìm phần tử nhỏ nhất Min-Heap

Trả về vị trí đầu tiên của Heap.

```cpp
int top()
{
        return h[0];
}
```

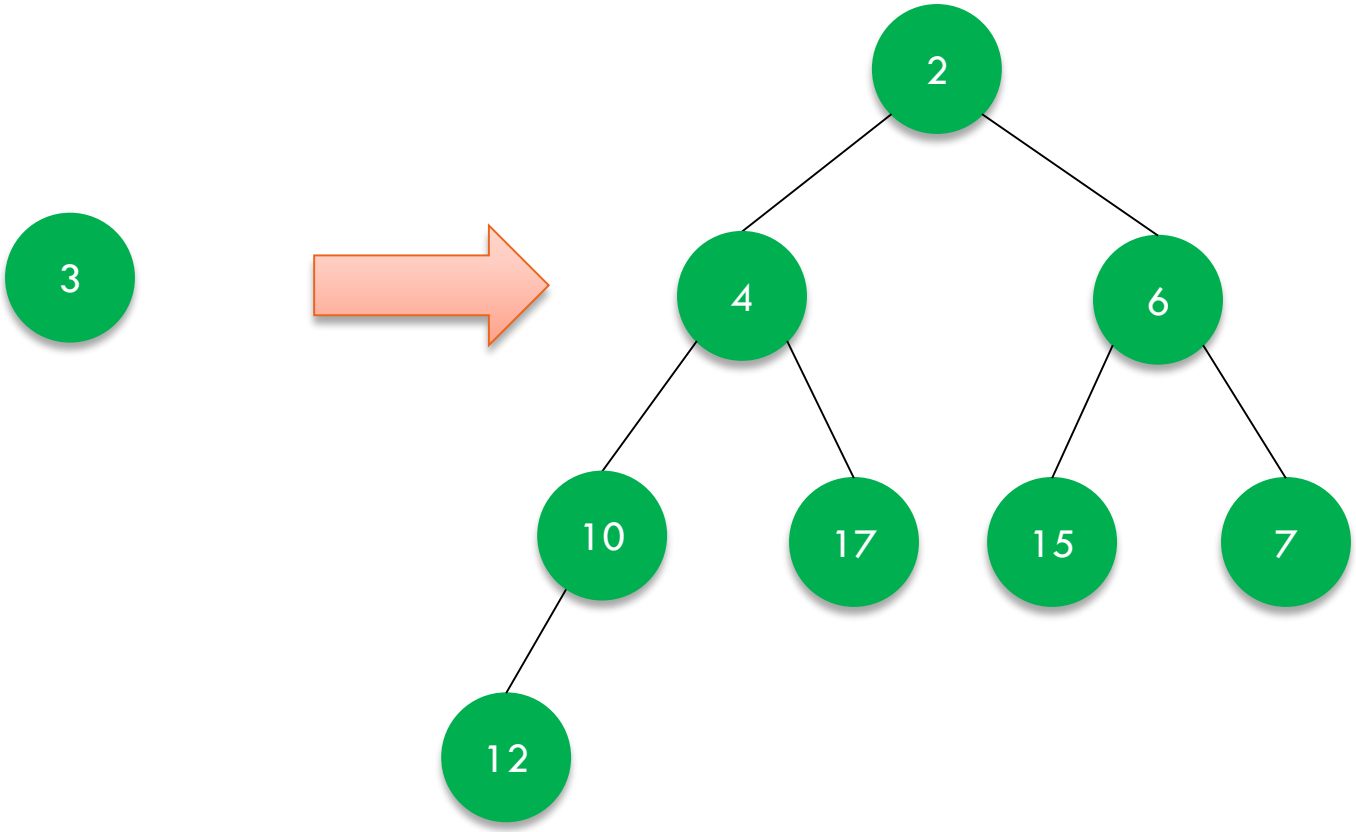

```python
def top():
    return h[0]
```


```java
    private static int top() {
        return h.get(0);
    }
```

2. Thêm phần tử vào trong Heap

Thêm phần tử có giá trị = 3 vào Heap.

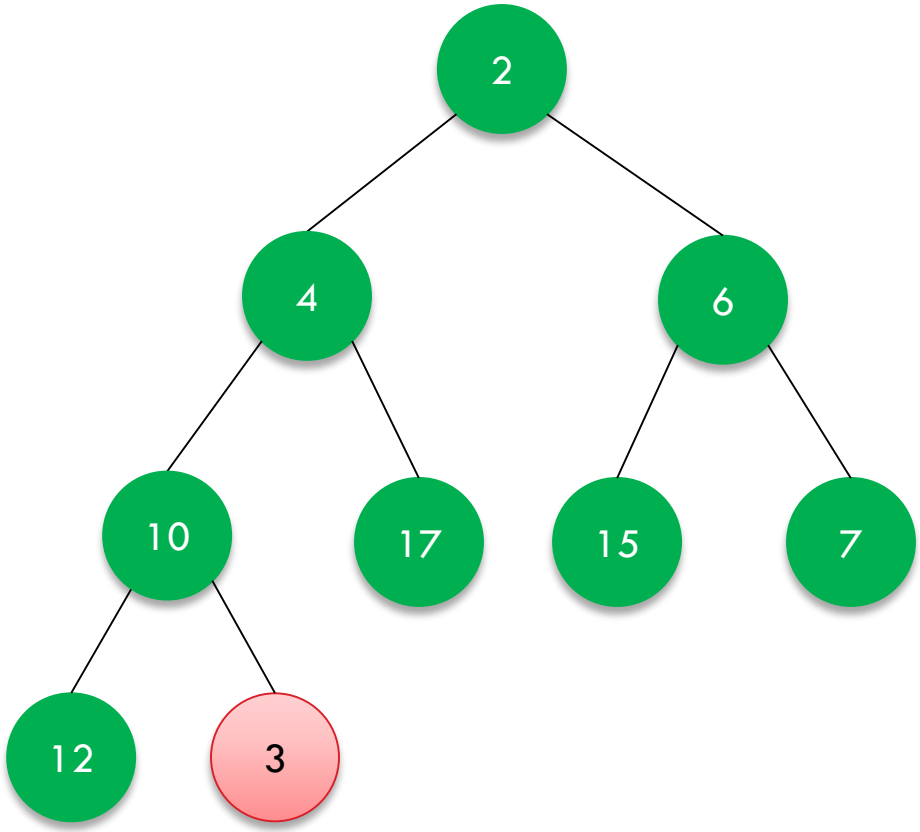
0	1	2	3	4	5	6	7
2	4	6	10	17	15	7	12



2. Thêm phần tử vào trong Min-Heap

Bước 1: Thêm 3 vào vị trí cuối cùng của Heap.

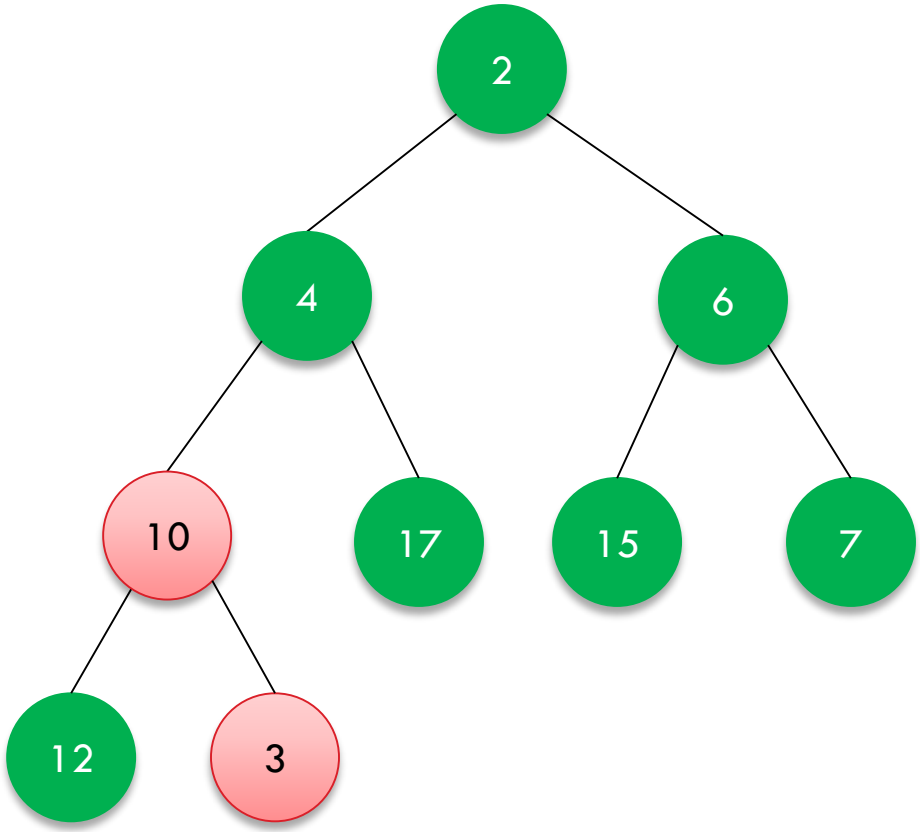
0	1	2	3	4	5	6	7	**8**
2	4	6	10	17	15	7	12	**3**



2. Thêm phần tử vào trong Min-Heap

Bước 2: Tìm node cha của node 3 ➔ node 10.

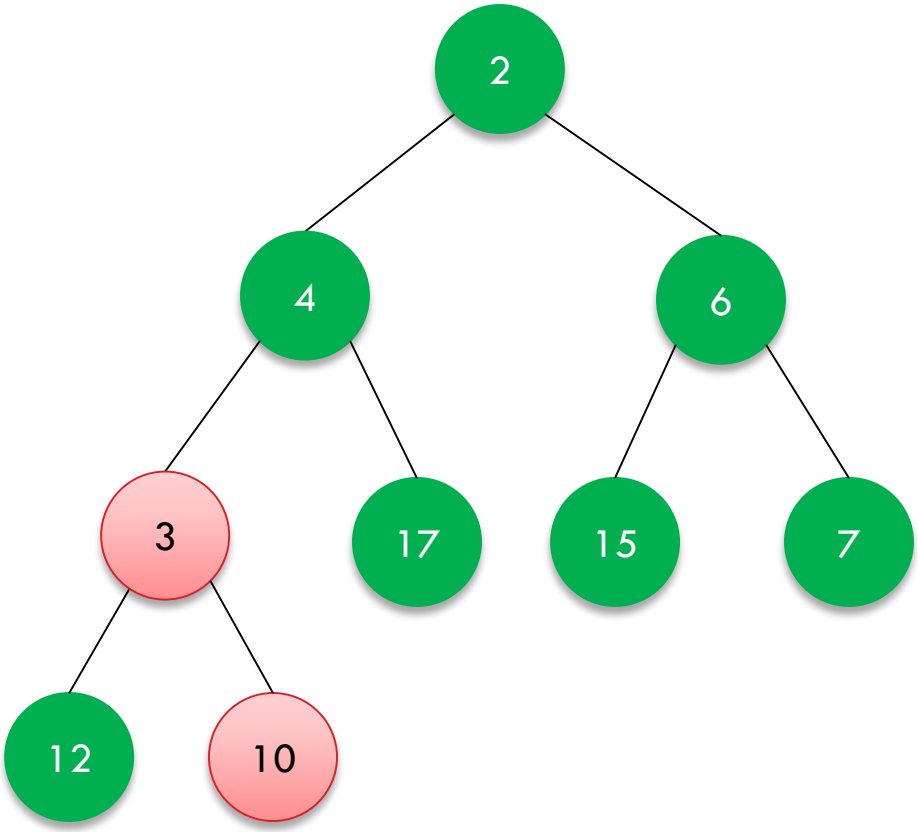
0	1	2	**3**	4	5	6	7	**8**
2	4	6	**10**	17	15	7	12	**3**



2. Thêm phần tử vào trong Min-Heap

Bước 2: Node 3 chưa đúng vị trí, hoán đổi node 3 và node 10 với nhau.

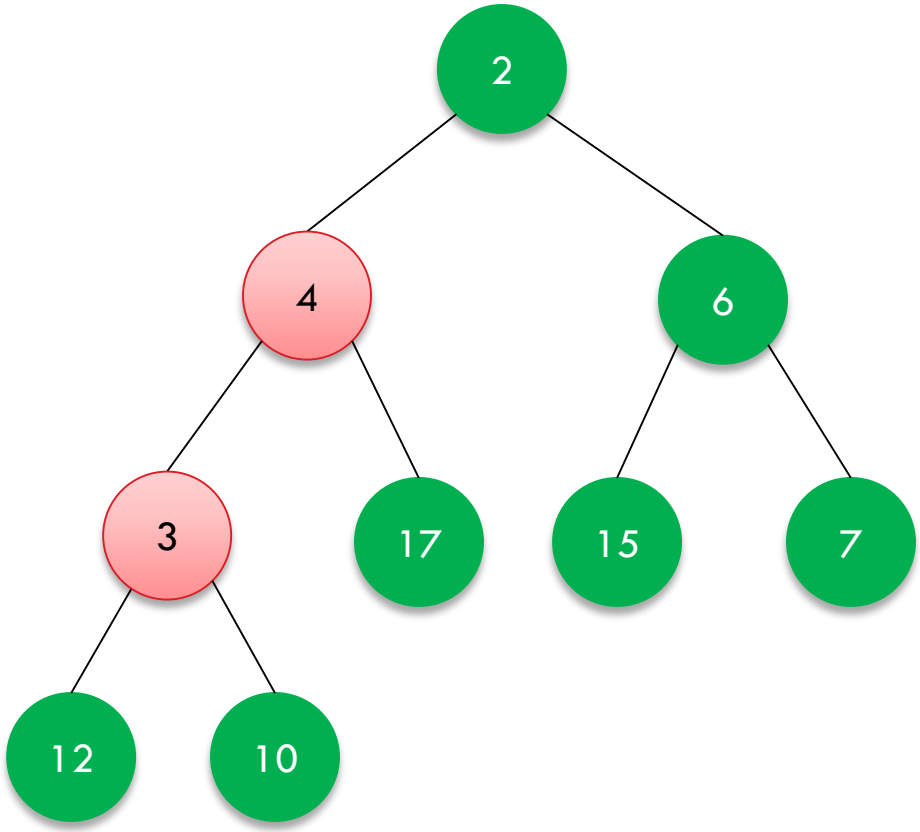
0	1	2	**3**	4	5	6	7	**8**
2	4	6	**3**	17	15	7	12	**10**



2. Thêm phần tử vào trong Min-Heap

Bước 3: Tìm nút cha của node 3 ➔ node 4.

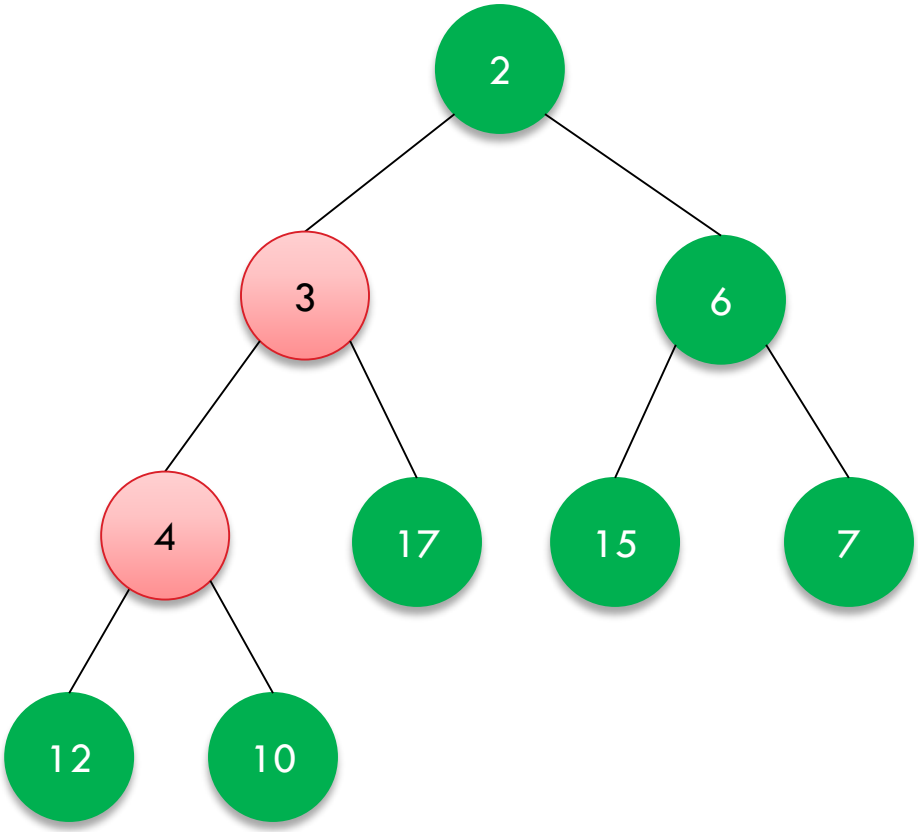
0	**1**	2	**3**	4	5	6	7	8
2	**4**	6	**3**	17	15	7	12	10



2. Thêm phần tử vào trong Min-Heap

Bước 3: Node 3 chưa đúng vị trí, hoán đổi node 3 và node 4 với nhau.

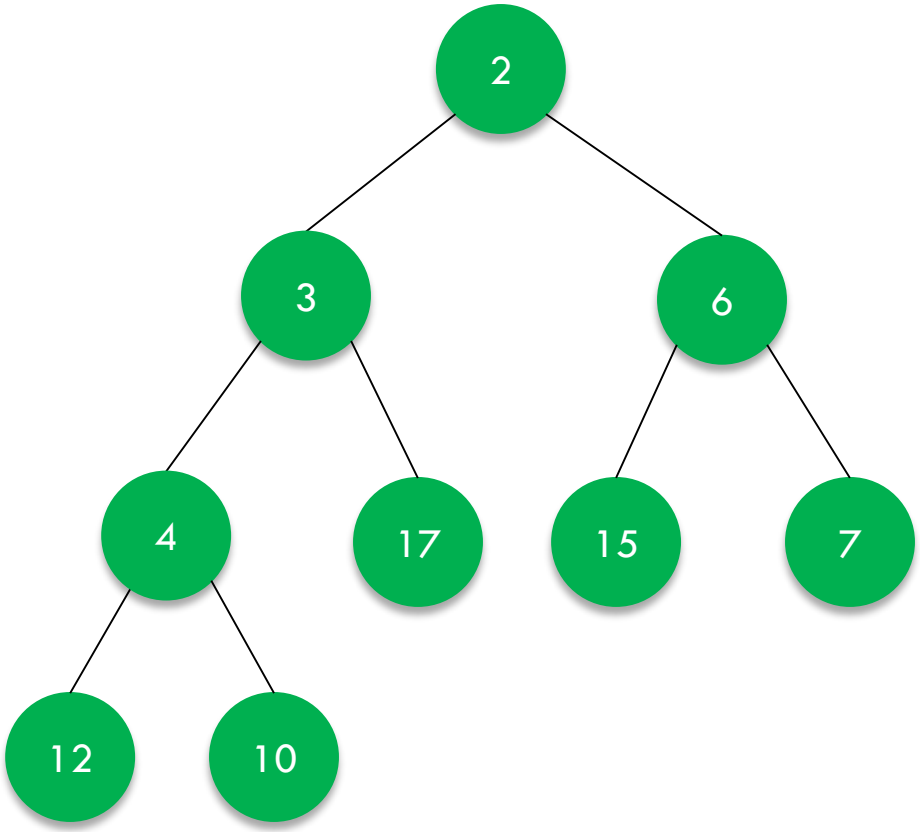
0	**1**	2	**3**	4	5	6	7	8
2	**3**	6	**4**	17	15	7	12	10



2. Thêm phần tử vào trong Min-Heap

Dừng thuật toán vì các node không còn mâu thuẫn nhau.

0	1	2	3	4	5	6	7	8
2	3	6	4	17	15	7	12	10



Source Code thêm phần tử vào Min-Heap

```
1.  void push(int value)
2.  {
3.      h.push_back(value);
4.      int i = h.size() - 1;
5.      while (i != 0 && h[(i - 1) / 2] > h[i])
6.      {
7.          swap(h[i], h[(i - 1) / 2]);
8.          i = (i - 1) / 2;
9.      }
10. }
```

Source Code thêm phần tử vào Min-Heap

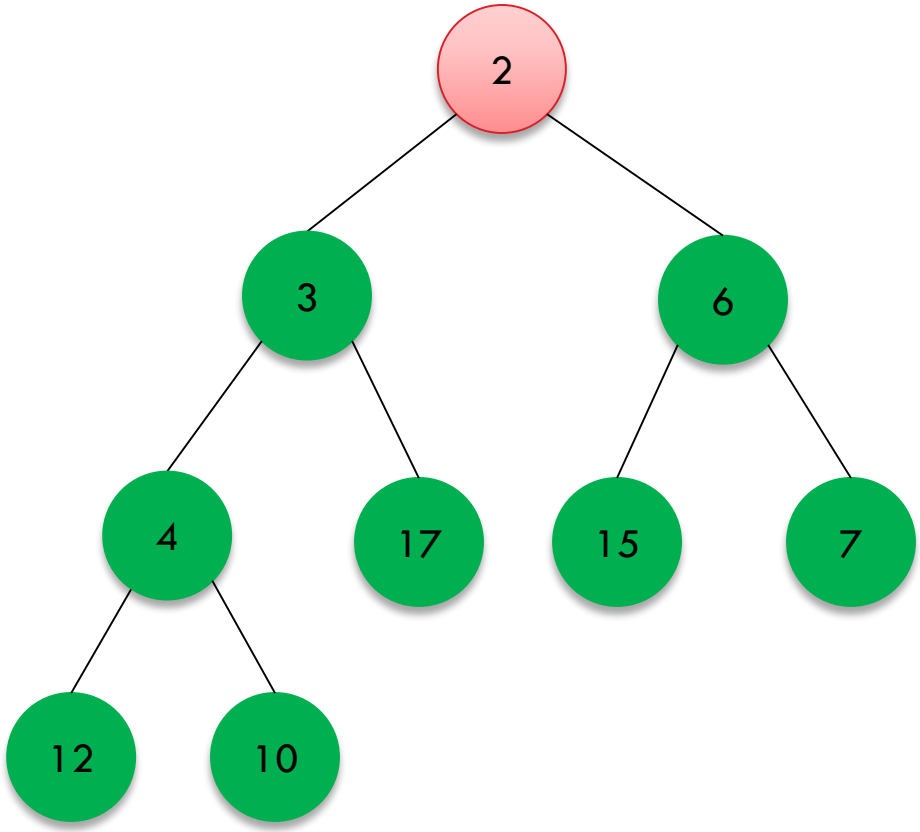
```
1.  def push(value):
2.      h.append(value)
3.      i = len(h) - 1
4.      while i != 0 and h[(i - 1) // 2] > h[i]:
5.          h[i], h[(i - 1) // 2] = h[(i - 1) // 2], h[i]
6.          i = (i - 1) // 2
```

Source Code thêm phần tử vào Min-Heap

```
1.      private static void push(int value) {
2.          h.add(value);
3.          int i = h.size() - 1;
4.          while (i != 0 && h.get((i - 1) / 2) > h.get(i)) {
5.              swap(i, (i - 1) / 2);
6.              i = (i - 1) / 2;
7.          }
8.      }
```

3. Xóa phần tử ra khỏi Min-Heap

0	1	2	3	4	5	6	7	8
2	3	6	4	17	15	7	12	10



3. Xóa phần tử ra khỏi Min-Heap

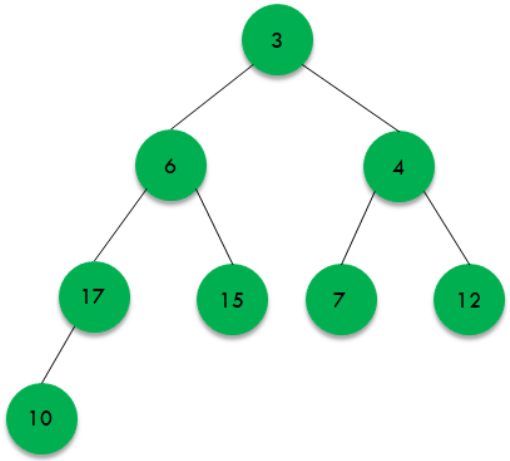
1. Xóa phần tử mang giá trị 2 khỏi mảng.

0	1	2	3	4	5	6	7	8
~~2~~	3	6	4	17	15	7	12	10

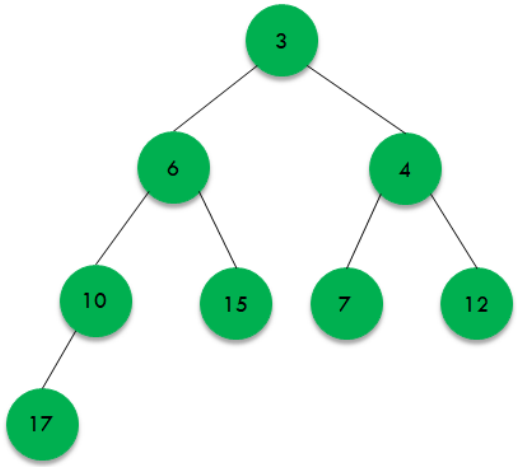
2. Di chuyển các giá trị khác trong mảng lên.

0	1	2	3	4	5	6	7
3	6	4	17	15	7	12	10

3. Kiểm tra lại cây Heap.



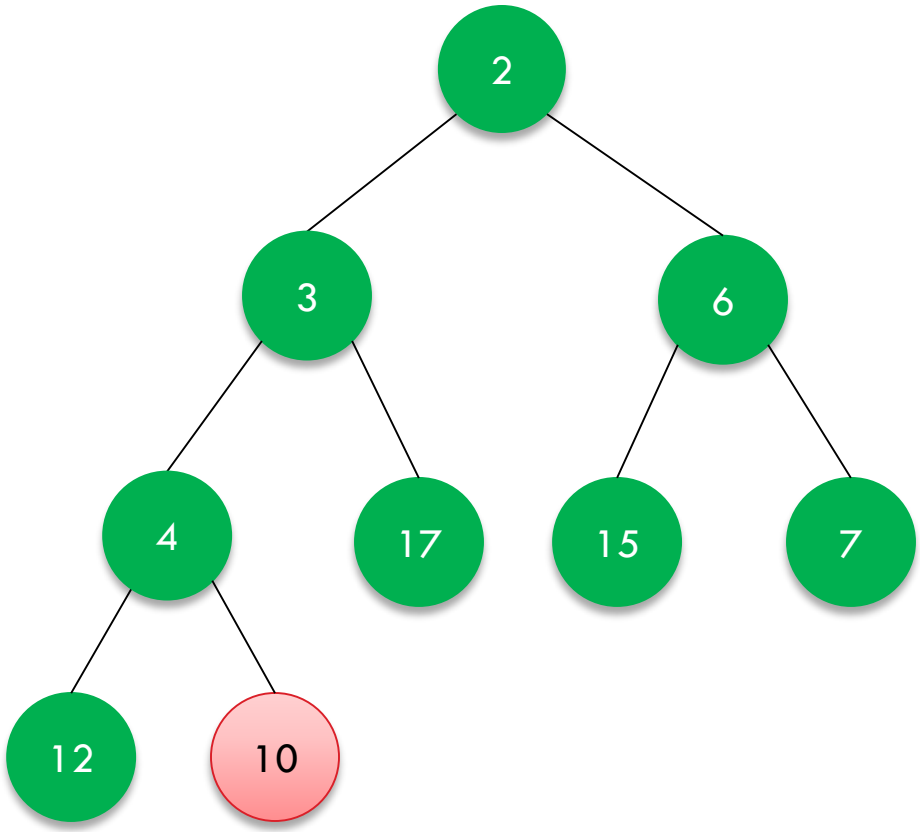
4. Chuẩn hóa lại cây Heap.



3. Xóa phần tử ra khỏi Min-Heap

Bước 1: Chọn phần tử cuối cùng của Heap.

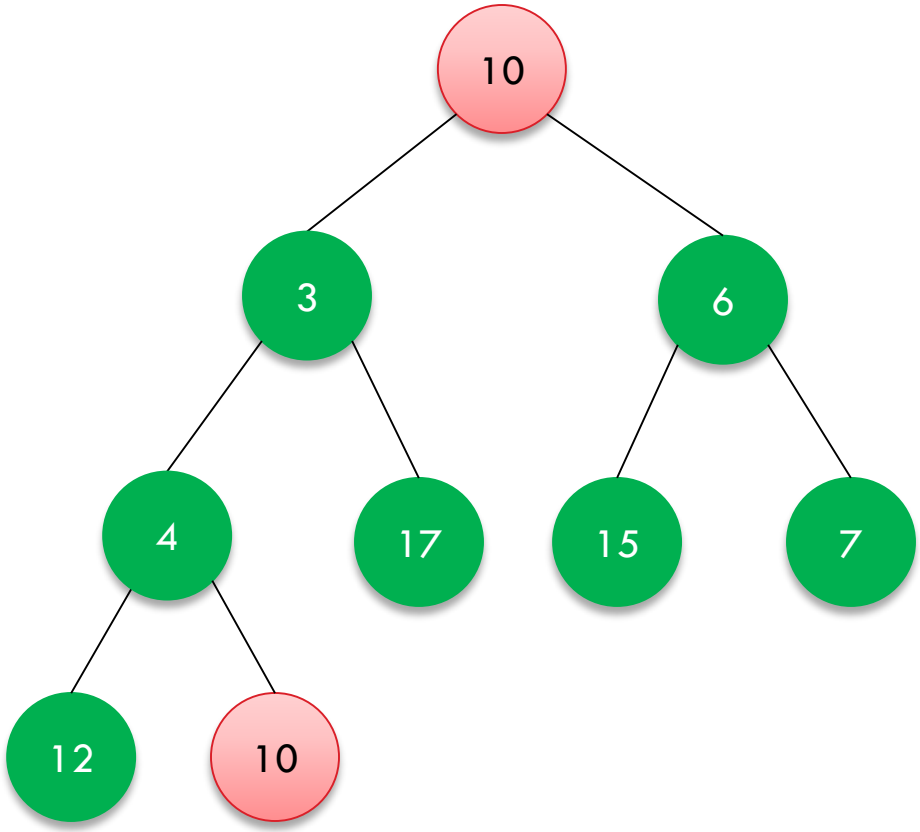
0	1	2	3	4	5	6	7	**8**
2	3	6	4	17	15	7	12	**10**



3. Xóa phần tử ra khỏi Min-Heap

Bước 1: Gán giá trị phần tử cuối Heap cho phần tử đầu Heap.

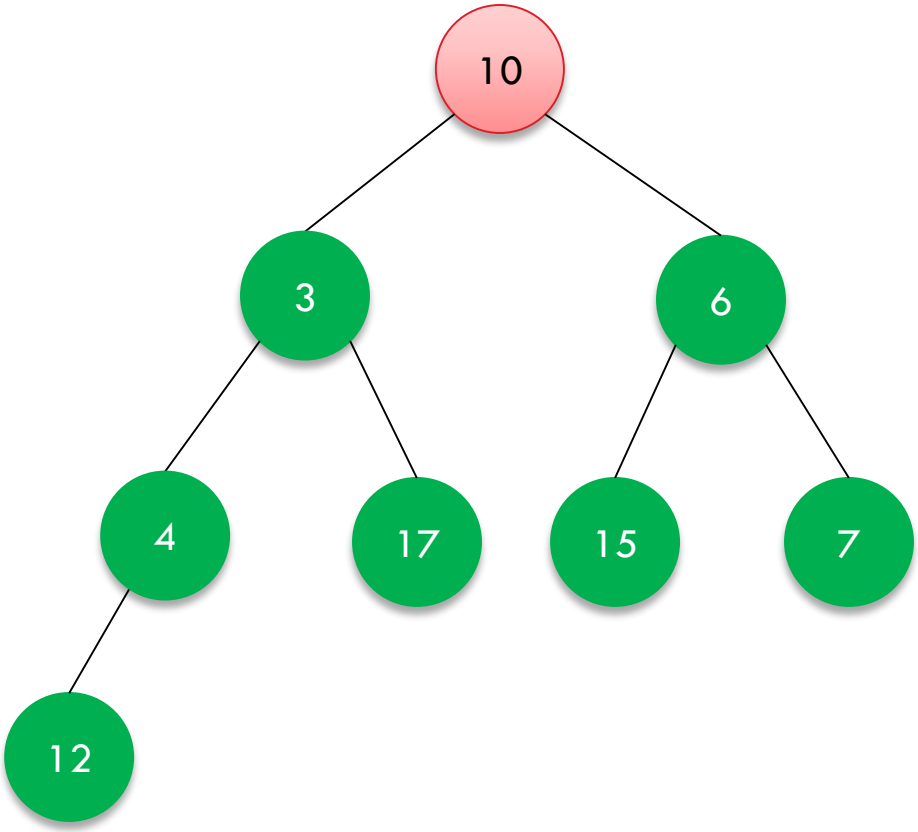
0	1	2	3	4	5	6	7	8
10	3	6	4	17	15	7	12	**10**



3. Xóa phần tử ra khỏi Min-Heap

Bước 2: Xóa phần tử cuối Heap.

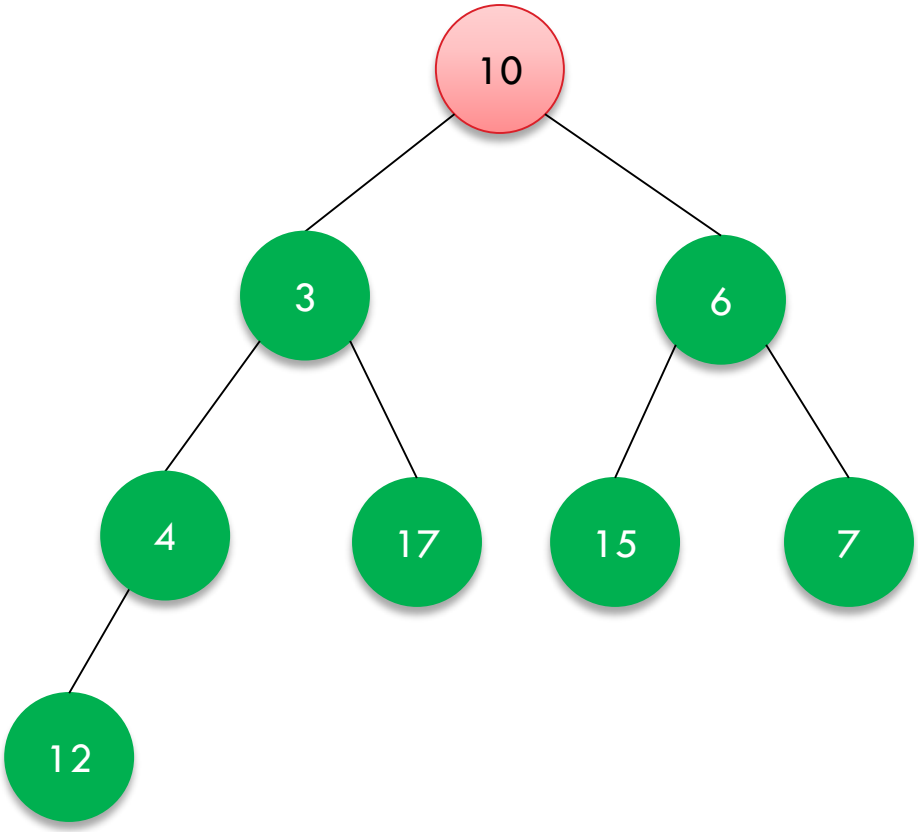
0	1	2	3	4	5	6	7
10	3	6	4	17	15	7	12



3. Xóa phần tử ra khỏi Min-Heap

Bước 3: Cân bằng lại Heap từ phần tử đầu tiên.

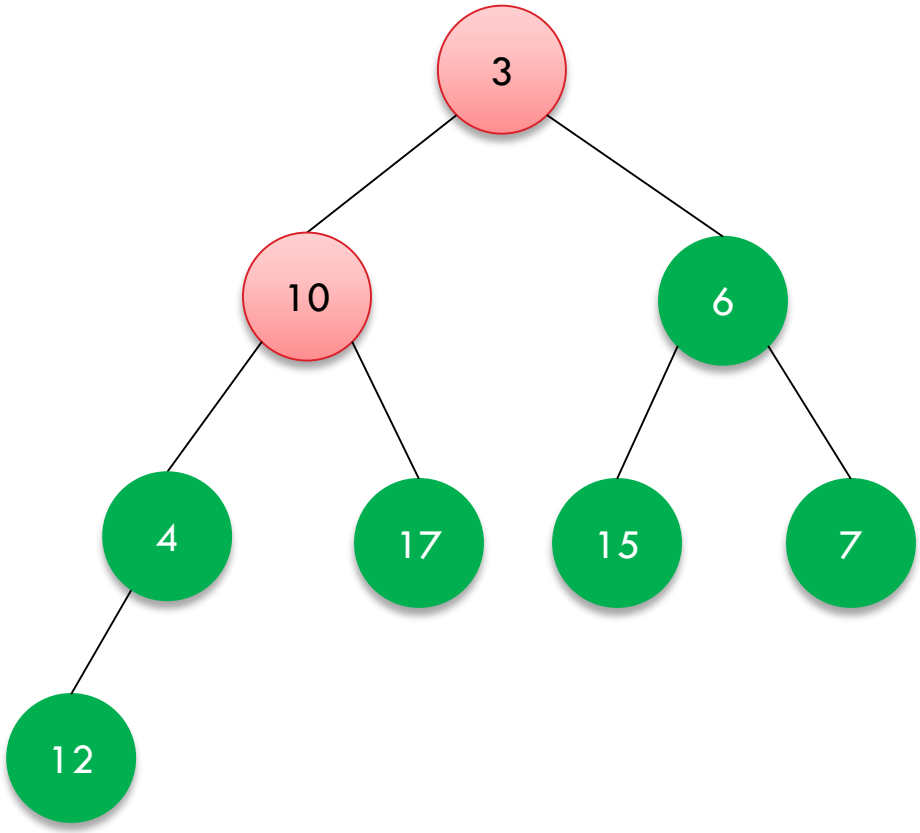
0	1	2	3	4	5	6	7
10	3	6	4	17	15	7	12



3. Xóa phần tử ra khỏi Min-Heap

Bước 3: Hoán đổi node 3 và node 10 với nhau.

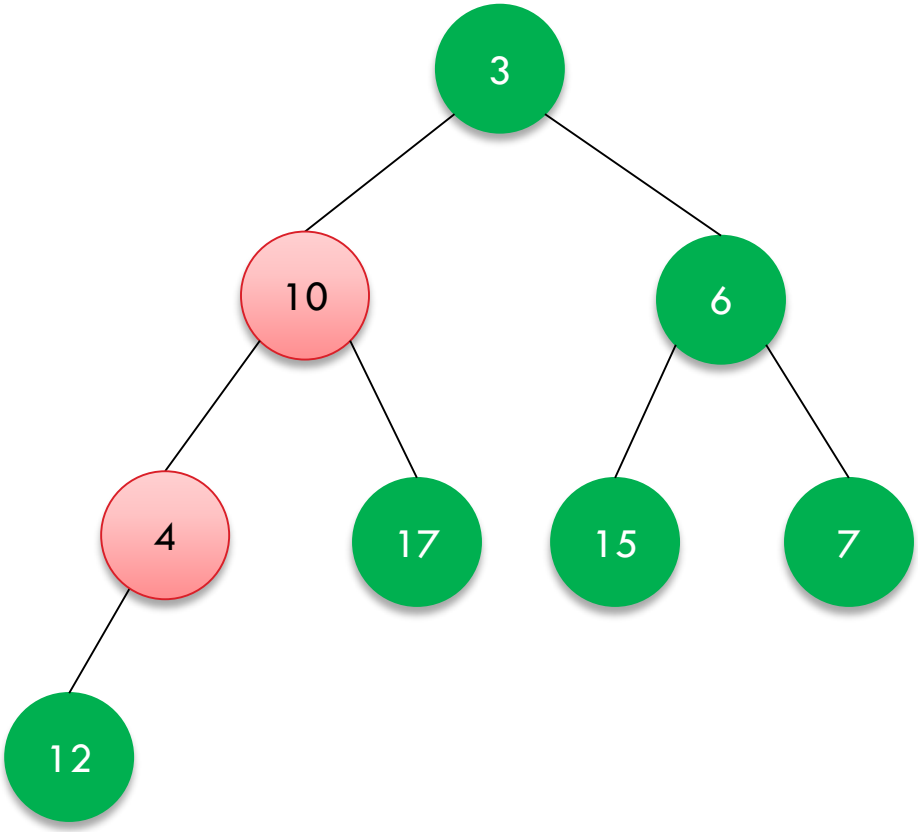
0	1	2	3	4	5	6	7
3	**10**	6	4	17	15	7	12



3. Xóa phần tử ra khỏi Min-Heap

Bước 4: Tiếp tục cân bằng ở node tiếp theo.

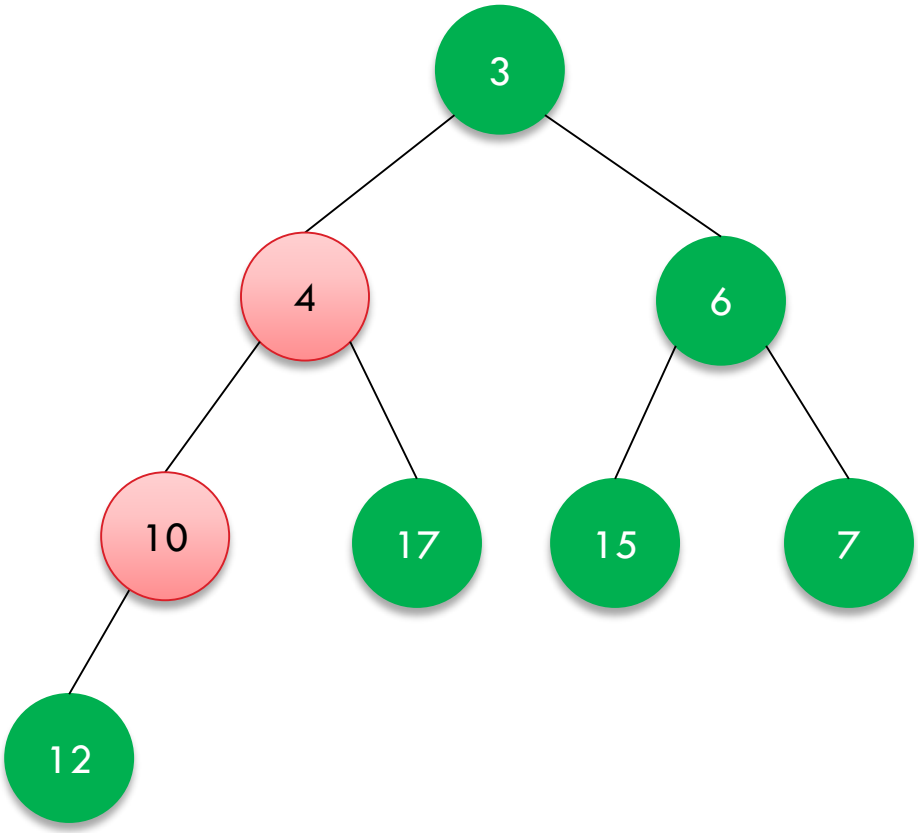
0	**1**	2	**3**	4	5	6	7
3	**10**	6	**4**	17	15	7	12



3. Xóa phần tử ra khỏi Min-Heap

Bước 4: Hoán đổi node 4 và node 10 với nhau.

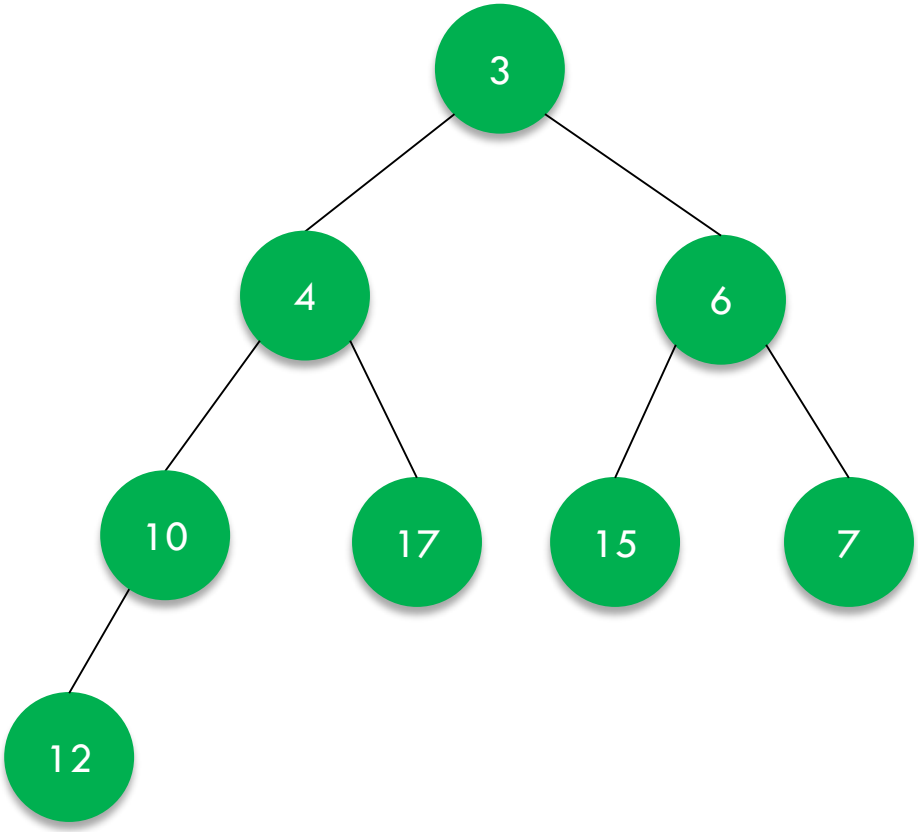
0	**1**	2	**3**	4	5	6	7
3	**4**	6	**10**	17	15	7	12



3. Xóa phần tử ra khỏi Min-Heap

Dừng thuật toán vì các node không còn mâu thuẫn nhau.

0	1	2	3	4	5	6	7
3	4	6	10	17	15	7	12



Souce code xóa phần tử đầu Min-Heap

```
1.  void pop()
2.  {
3.      int length = h.size();
4.      if (length == 0)
5.          return;
6.      h[0] = h[length - 1];
7.      h.pop_back();
8.      minHeapify(0);
9.  }
```

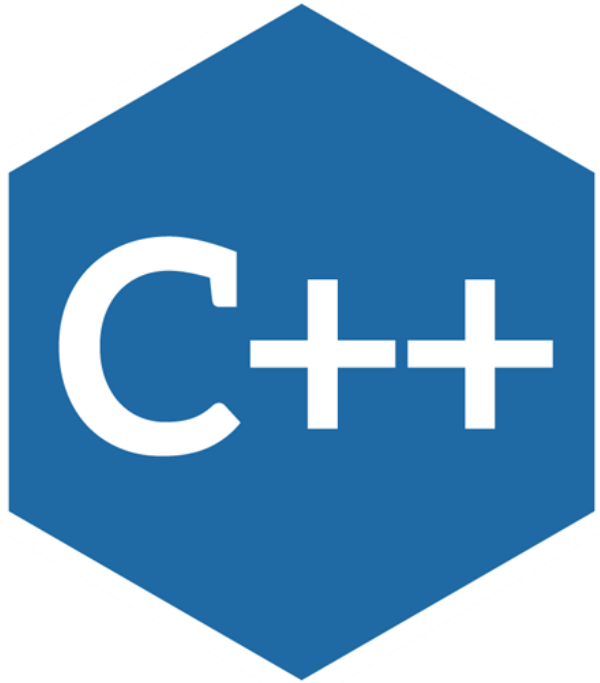
Souce code xóa phần tử đầu Min-Heap

```
1.  def pop():
2.      length = len(h)
3.      if length == 0:
4.          return
5.      h[0] = h[length - 1]
6.      h.pop()
7.      minHeapify(0)
```

Souce code xóa phần tử đầu Min-Heap

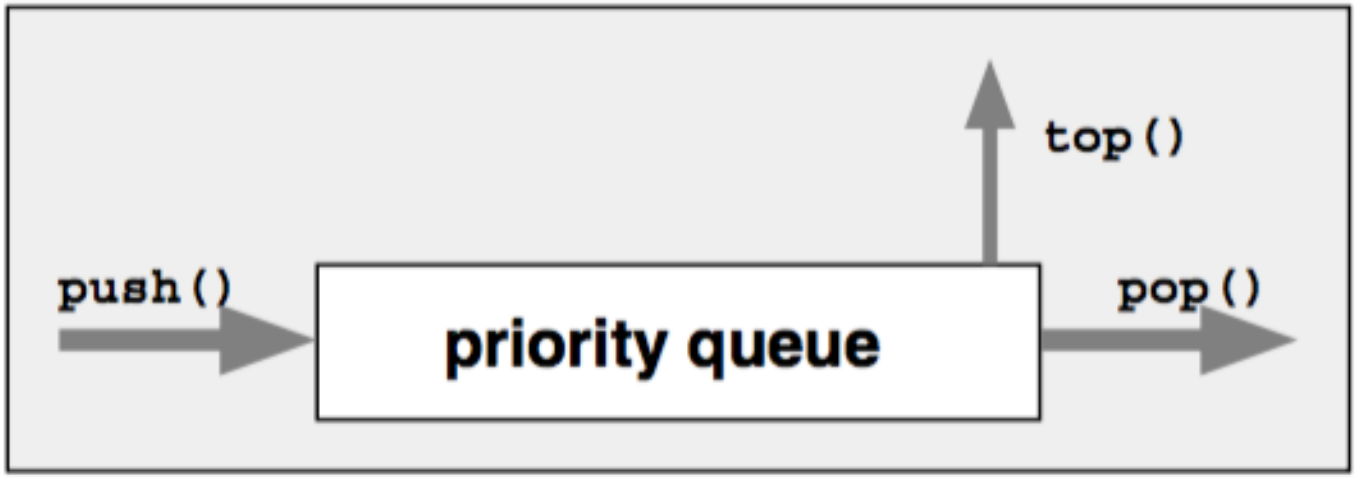
```
1.      private static void pop() {
2.          if (h.size() == 0) {
3.              return;
4.          }
5.          int n = h.size();
6.          h.set(0, h.get(n - 1));
7.          h.remove(n - 1);
8.          minHeapify(0);
9.      }
```

SỬ DỤNG HEAP BẰNG THƯ VIỆN STL



priority_queue

priority_queue (hàng đợi ưu tiên) trong C++ là một cấu trúc dữ liệu dùng để lưu trữ sao cho phần tử ở đỉnh luôn luôn là phần tử có độ ưu tiên lớn nhất so với các phần tử khác.



Khai báo và sử dụng (1)

Thao tác tương tự như bước **0. Xây dựng cây Heap.**

Thư viện:

```
#include <queue>
using namespace std;
```

Khai báo:

```
priority_queue<data_type> variable;
```

Khai báo và sử dụng (2)

Cách 1: Sử dụng như **max-heap**.

```
priority_queue<int> pq;
```

Ví dụ:

```
priority_queue<int> pq;
int h[] = {7, 12, 6, 10, 17, 15, 2, 4};
for (int i = 0; i < 8; i++) {
    pq.push(h[i]);
}
```

0	1	2	3	4	5	6	7
17	12	15	7	10	6	2	4

Khai báo và sử dụng (3)

Cách 2: Sử dụng như **min-heap**.

```
priority_queue<int, vector<int>, greater<int> > pq;
```

Ví dụ:

```
priority_queue<int, vector<int>, greater<int> > pq;
int h[] = { 7, 12, 6, 10, 17, 15, 2, 4};
for (int i = 0; i < 8; i++) {
    pq.push(h[i]);
}
```

0	1	2	3	4	5	6	7
2	4	6	10	17	15	7	12

Các hàm thành viên của priority_queue

top(): Trả về giá trị node gốc của hàng đợi ưu tiên chứa giá trị nhỏ nhất (đối với Min-heap) hoặc lớn nhất (đối với Max-heap).

0	1	2	3	4	5	6	7
2	4	6	10	17	15	7	12

```
int value = pq.top();
cout << value;
```

2

Các hàm thành viên của priority_queue

push(value): Thêm một phần tử vào priority_queue.

0	1	2	3	4	5	6	7
2	4	6	10	17	15	7	12

```
pq.push(3);
```

0	**1**	2	3	4	5	6	7	8
2	**3**	6	4	17	15	7	12	10

Các hàm thành viên của priority_queue

pop(): Xóa phần tử đầu trong priority_queue.

0	1	2	3	4	5	6	7	8
2	3	6	4	17	15	7	12	10

```
pq.pop();
```

0	1	2	3	4	5	6	7
3	4	6	10	17	15	7	12

Một số hàm thành viên khác

size(): Trả về số lượng phần tử hiện tại có trong hàng đợi ưu tiên.

empty(): Kiểm tra hàng đợi ưu tiên có rỗng hay không.

pq1.swap(pq2): Hoán đổi 2 hàng đợi ưu tiên với nhau.

Xóa toàn bộ phần tử trong priority_queue

clear: dùng phương pháp gán cho một priority queue mới.

0	1	2	3	4	5	6	7
2	4	6	10	17	15	7	12

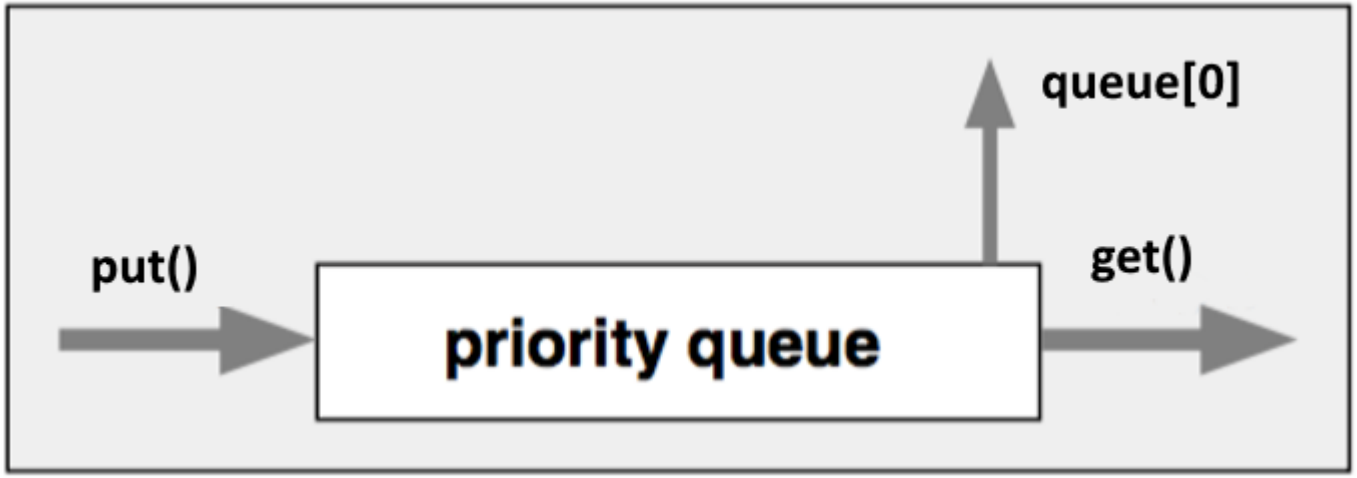
```
pq = priority_queue<int>();
```

0	1	2	3	4	5	…
…	…	…	…	…	…	…

SỬ DỤNG HEAP BẰNG THƯ VIỆN QUEUE

PriorityQueue

PriorityQueue (hàng đợi ưu tiên) trong python là một cấu trúc dữ liệu dùng để lưu trữ sao cho phần tử ở đỉnh luôn luôn là phần tử có độ ưu tiên lớn nhất so với các phần tử khác. Trong python có 2 cách sử dụng khác nhau: sử dụng PriorityQueue và sử dụng heapq.



Khai báo và sử dụng (1)

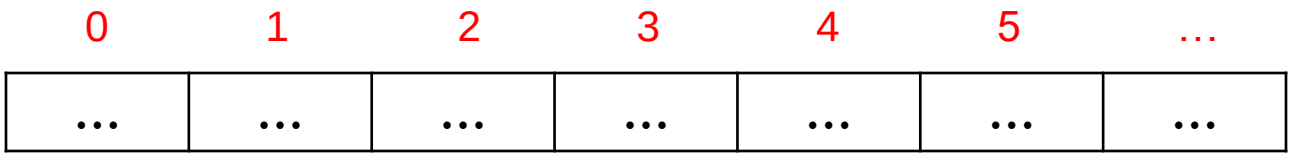
Thao tác tương tự như bước **0. Xây dựng cây Heap.**

Thư viện: Queue/queue tương ứng với Python 2.x/Python 3.x

```
import queue
```

Khai báo:

```
variable = queue.PriorityQueue()
```



Khai báo và sử dụng (2)

Cách 1: Sử dụng như **max-heap** bằng cách tạo một class mới và define operator __lt__ cho object đó.

```
class PQEntry:
    def __init__(self, value):
        self.value = value
    def __lt__(self, other):
        return self.value > other.value

a = [7, 12, 6, 10, 17, 15, 2, 4]
pq = queue.PriorityQueue()
for x in a:
    pq.put(PQEntry(x))
```

0	1	2	3	4	5	6	7
17	12	15	7	10	6	2	4

Khai báo và sử dụng (3)

Cách 2: Sử dụng như **min-heap**.

```
pq = queue.PriorityQueue()
```

Ví dụ:

```
pq = queue.PriorityQueue()
h = [7, 12, 6, 10, 17, 15, 2, 4]
for x in h:
    pq.put(x)
```

0	1	2	3	4	5	6	7
2	4	6	10	17	15	7	12

Các hàm thành viên của PriorityQueue

queue[0]: Trả về giá trị node gốc của hàng đợi ưu tiên.

0	1	2	3	4	5	6	7
2	4	6	10	17	15	7	12

```
value = pq.queue[0]
print(value)
```

2

Các hàm thành viên của PriorityQueue

put(obj): Thêm một phần tử vào PriorityQueue.

0	1	2	3	4	5	6	7
2	4	6	10	17	15	7	12

```
pq.put(3)
```

0	**1**	2	3	4	5	6	7	8
2	**3**	6	4	17	15	7	12	10

Các hàm thành viên của PriorityQueue

get(): Xóa phần tử đầu trong PriorityQueue và trả về giá trị của phần tử đó.

0	1	2	3	4	5	6	7	8
2	**3**	6	4	17	15	7	12	10

```
value = pq.get()
print(value)
```

0	1	2	3	4	5	6	7
3	4	6	10	17	15	7	12

2

Một số hàm thành viên khác

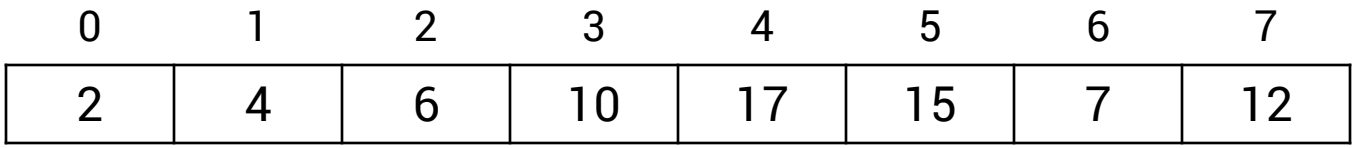
len(variable.queue): Lấy kích thước của PriorityQueue.

Ví dụ: len(pq.queue)

empty(): Kiểm tra hàng đợi ưu tiên có rỗng hay không.

Xóa toàn bộ phần tử trong PriorityQueue

Dùng phương pháp gán cho một priority queue mới.




```
pq = queue.PriorityQueue()
```

SỬ DỤNG HEAP BẰNG THƯ VIỆN HEAPQ TRONG PYTHON

heapq trong Python

Trong Python, ngoài PriorityQueue ra còn một thư viện khác hỗ trợ xây dựng một cây binary heap là heapq. heapq là thư viện chứa các hàm hỗ trợ các thao tác cơ bản trong heap.

- Ưu điểm: tốc độ nhanh hơn PriorityQueue và dễ sử dụng hơn PriorityQueue cũng như heap tự cài đặt.
- Nhược điểm: heapq không tự lưu lại cấu trúc heap mà chỉ sử dụng 1 mảng (list) trong tham số đầu vào để xử lý nên cần quản lý chặt vùng nhớ. Cần khai báo đầy đủ các toán tử so sánh cần thiết trong trường hợp sử dụng cấu trúc heap với cấu trúc dữ liệu tự định nghĩa.

Khai báo và sử dụng

Thao tác tương tự như **0. Xây dựng cây Heap.**

Thư viện: heapq

```
import heapq
```

Khai báo: heapq chỉ là tập hợp các hàm xử lý trong heap, nên để sử dụng được ta cần có một list sẵn để chứa heap.

```
variable = []
```

Tạo heap từ một list có sẵn

Big-O Blue

Để sử dụng như **max-heap**, có thể đảo dấu giá trị hoặc tạo một class mới và define operator __lt__ cho object như PriorityQueue:

```
class PQEntry:
    def __init__(self, value):
        self.value = value
    def __lt__(self, other):
        return self.value > other.value

a = [7, 12, 6, 10, 17, 15, 2, 4]
h = []
for x in a:
    heapq.heappush(h, PQEntry(x))
```

0	1	2	3	4	5	6	7
17	12	15	7	10	6	2	4

Tạo heap từ một list có sẵn

heapq.heapify(list): Chuẩn hóa list thành Heap.

Độ phức tạp **O(N),** mặc định sẽ là **Min-heap**.

Ví dụ:

```
h = [7, 12, 6, 10, 17, 15, 2, 4]
heapq.heapify(h)
print(h)
```

0	1	2	3	4	5	6	7
2	4	6	10	17	15	7	12

Các hàm thành viên khác của heapq

variable_name[0]: Trả về phần tử đầu tiên trong list chứa giá trị nhỏ nhất (đối với Min-heap) hoặc lớn nhất (đối với Max-heap)

0	1	2	3	4	5	6	7
2	4	6	10	17	15	7	12

```
value = h[0]
print(value)
```

2

Các hàm thành viên khác của heapq

heapq.heappush(list, obj): Thêm một phần tử vào heap.

0	1	2	3	4	5	6	7
2	4	6	10	17	15	7	12

```
heapq.heappush(h,3)
```

0	1	2	3	4	5	6	7	8
2	**3**	6	4	17	15	7	12	10

Các hàm thành viên của heapq

heapq.heappop(list): Lấy giá trị và xóa phần tử đầu trong heap.

0	1	2	3	4	5	6	7	8
2	3	6	4	17	15	7	12	10

```
heapq.heappop(h)
```

0	1	2	3	4	5	6	7
3	4	6	10	17	15	7	12

Một số hàm thành viên khác

len(variable): Lấy kích thước của list.

Ví dụ: len(h)

clear(): Xóa toàn bộ hàng đợi ưu tiên.

Ví dụ: h.clear()

*** *Lưu ý: Vì heapq chỉ là thư viện chứa hàm, nên để xóa toàn bộ phần tử của heap ta chỉ việc clear dữ liệu trong list chứa các phần tử của heap là được.*

Một số lưu ý:

- Cũng tương tự như PriorityQueue, để sử dụng heapq với lớp đối tượng, cần khai báo toán tử __lt__ cho lớp đó.
- Để sử dụng như max-heap, phải đảo dấu giá trị, hoặc tạo một lớp đối tượng với value là giá trị đầu vào, priority là số đối của value, khai báo __lt__ so sánh theo priority.

SỬ DỤNG HEAP BẰNG THƯ VIỆN

Khai báo và sử dụng (1)

Thao tác tương tự như bước **0. Xây dựng cây Heap.**

Thư viện:

```
import java.util.PriorityQueue;
```

Khai báo:

```
PriorityQueue <E> variable = new PriorityQueue <E>();
```

Khai báo và sử dụng (2)

Cách 1: Sử dụng như **max-heap**. Cần tạo thêm 1 comparator tương tự như sort (có thể gộp vào trong constructor lúc khai báo priority queue hoặc tạo class tường minh).

```
PriorityQueue<E> variable
      = new PriorityQueue<E>(new ComparatorClass());
```

```
class MaxHeapComparator implements Comparator<Integer> {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o2.compareTo(o1);
    }
}
```

Khai báo và sử dụng (3)

```
// inside main
PriorityQueue<Integer> pq = new PriorityQueue<Integer>(
                                        new MaxHeapComparator());
int[] h = new int[]{ 7, 12, 6, 10, 17, 15, 2, 4};
for (int i = 0; i < 8; i++) {
    pq.add(h[i]);
}
```

0	1	2	3	4	5	6	7
17	12	15	7	10	6	2	4

Khai báo và sử dụng (4)

Cách 2: Sử dụng như **min-heap**.

```
PriorityQueue<E> variable = new PriorityQueue<E>();
```

```
PriorityQueue<Integer> pq = new PriorityQueue<Integer>();
int[] h = new int[]{ 7, 12, 6, 10, 17, 15, 2, 4};
for (int i = 0; i < 8; i++) {
    pq.add(h[i]);
}
```

0	1	2	3	4	5	6	7
2	4	6	10	17	15	7	12

Các hàm thành viên của PriorityQueue

peek(): Trả về giá trị node gốc của hàng đợi ưu tiên.

0	1	2	3	4	5	6	7
2	4	6	10	17	15	7	12

```
Integer value = pq.peek();
System.out.print(value);
```

2

Các hàm thành viên của PriorityQueue

add(element): Thêm một phần tử vào PriorityQueue.

0	1	2	3	4	5	6	7
2	4	6	10	17	15	7	12

```
pq.add(3);
```

0	**1**	2	3	4	5	6	7	8
2	**3**	6	4	17	15	7	12	10

Các hàm thành viên của PriorityQueue

remove(): Xóa một phần tử đầu trong PriorityQueue.

0	1	2	3	4	5	6	7	8
2	3	6	4	17	15	7	12	10

```
pq.remove();
```

0	1	2	3	4	5	6	7
3	4	6	10	17	15	7	12

Một số hàm thành viên khác

size(): Trả về số lượng phần tử hiện tại có trong hàng đợi ưu tiên.

isEmpty(): Kiểm tra hàng đợi ưu tiên có rỗng hay không.

clear(): Xoá toàn bộ hang đợi ưu tiên.

Hỏi đáp