Big-O Blue

LECTURE 08

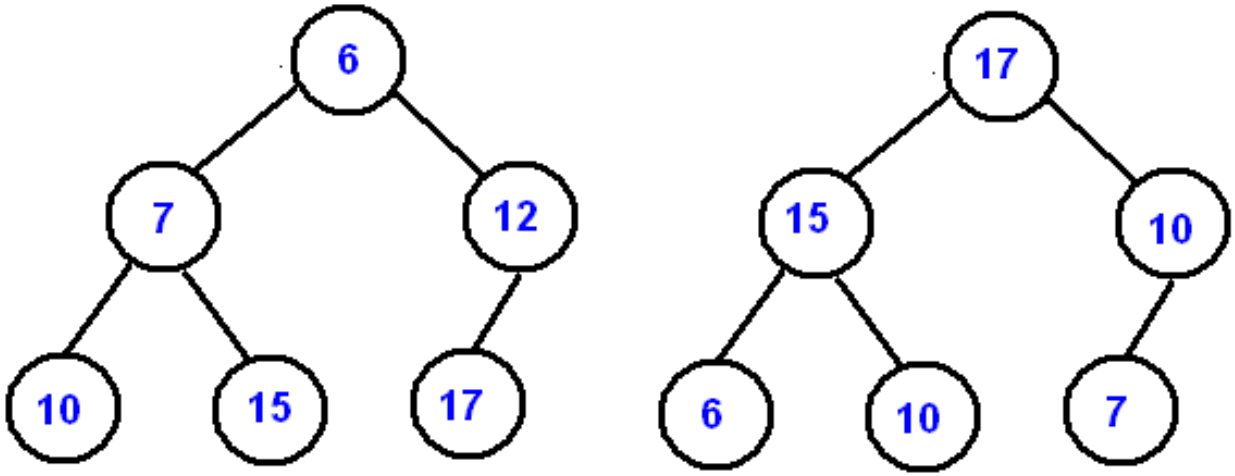
HEAP

Phạm Nguyễn Sơn Tùng

Email: sontungtn@gmail.com

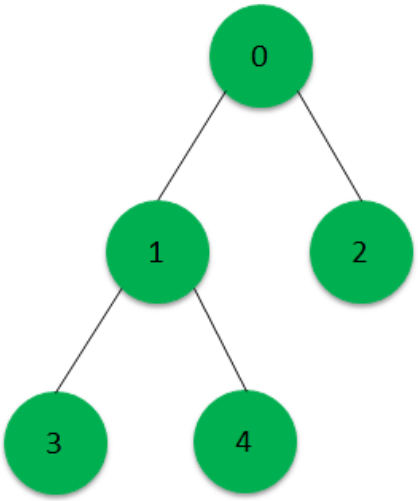
Định nghĩa Heap

Heap (đống) là cấu trúc cây nhị phân hoàn chỉnh (*complete binary tree)*. Nếu là Min-Heap thì mỗi node cha đều có giá trị nhỏ hơn hoặc bằng node con của nó. Ngược lại nếu là Max-Heap thì node cha lớn hơn hoặc bằng node con của nó.

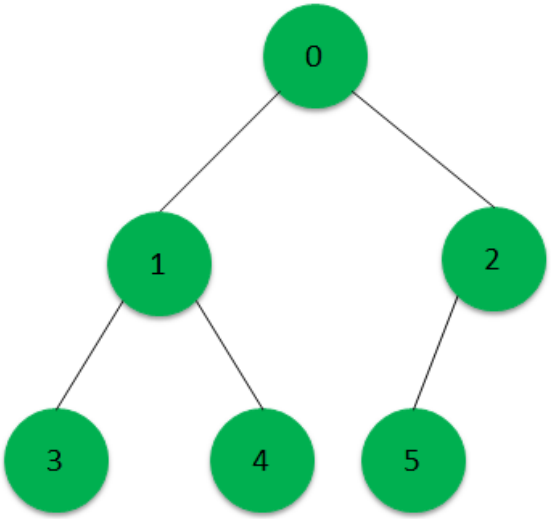


Các loại cây nhị phân

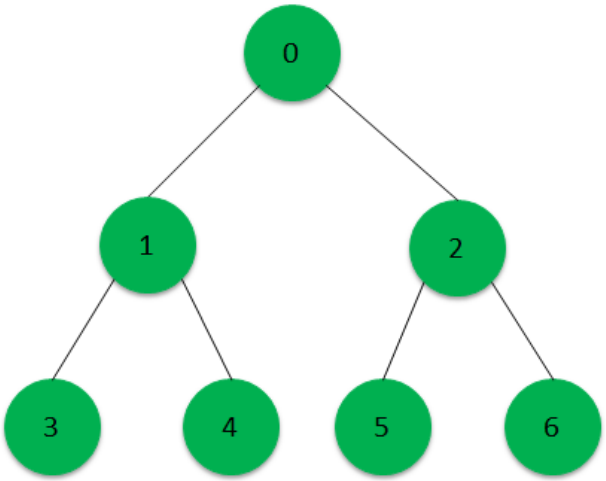
full binary tree



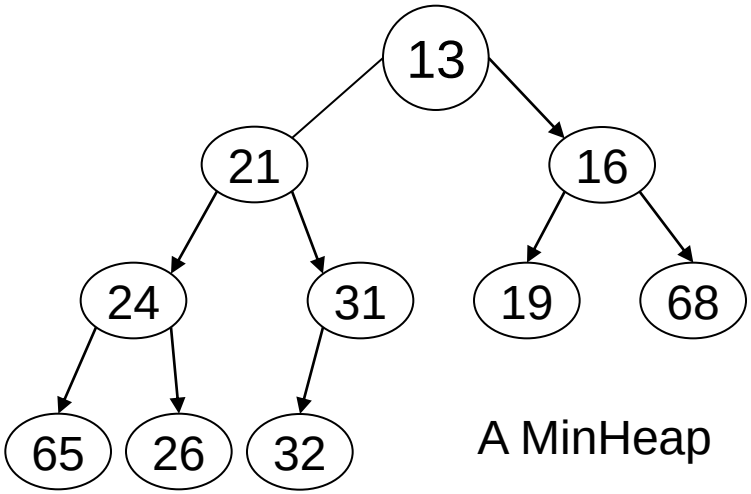
complete binary tree



perfect binary tree

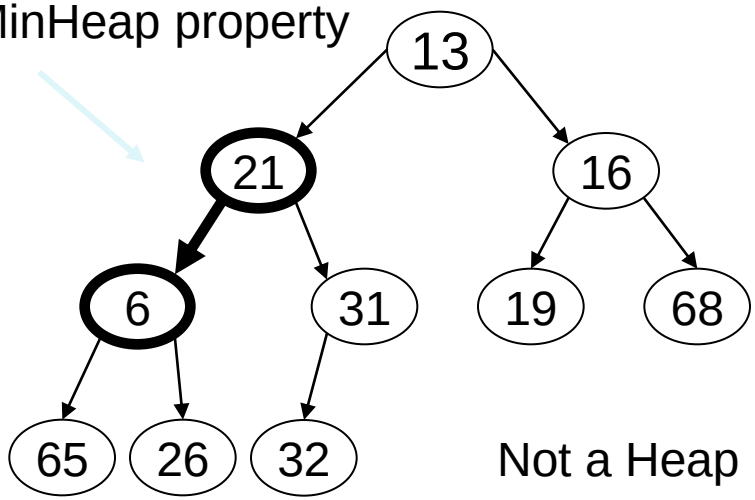


Phân biệt Heap

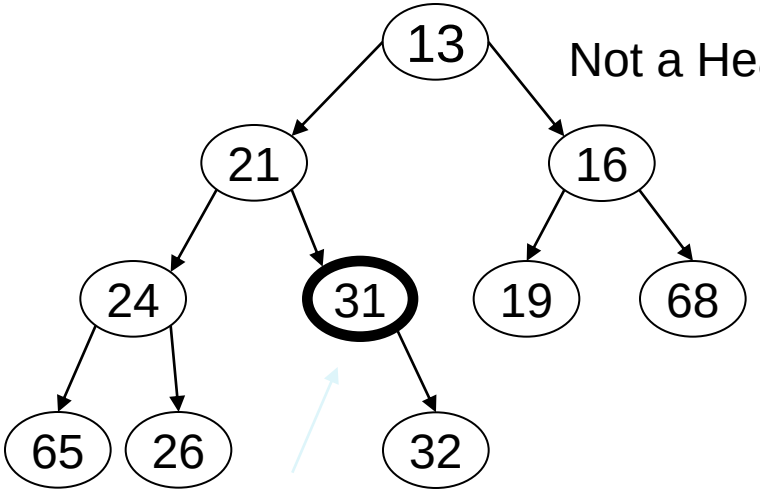


A MinHeap

Violates MinHeap property
21>6

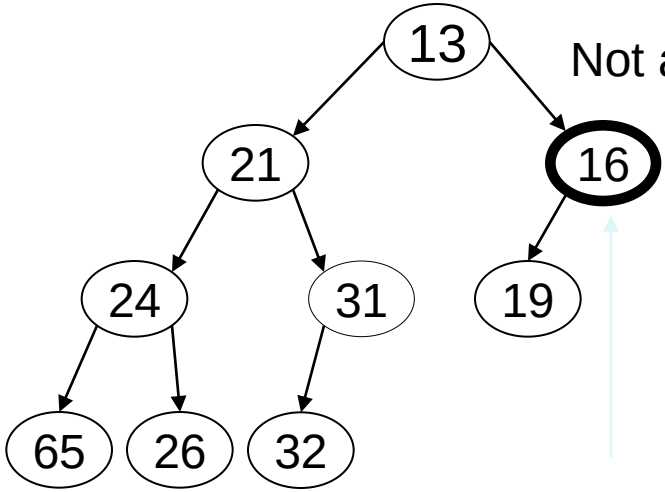


Not a Heap



Not a Heap

Violates heap structural property



Not a Heap

Violates heap structural property

Ứng dụng & độ phức tạp của Heap

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.

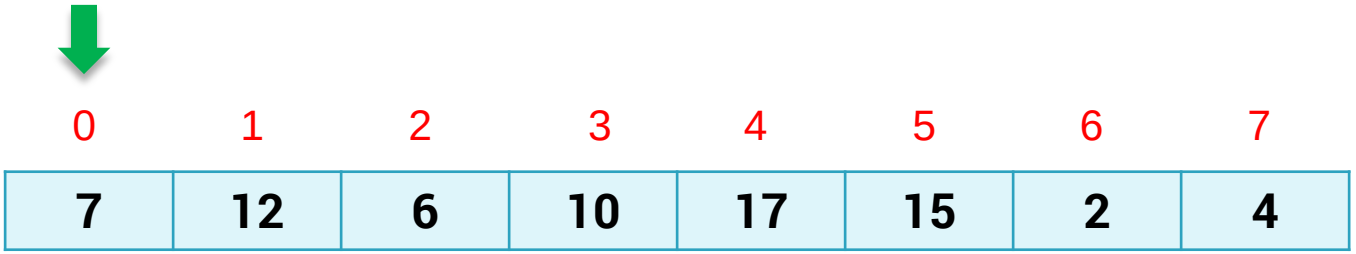
Các thao tác cơ bản của Heap

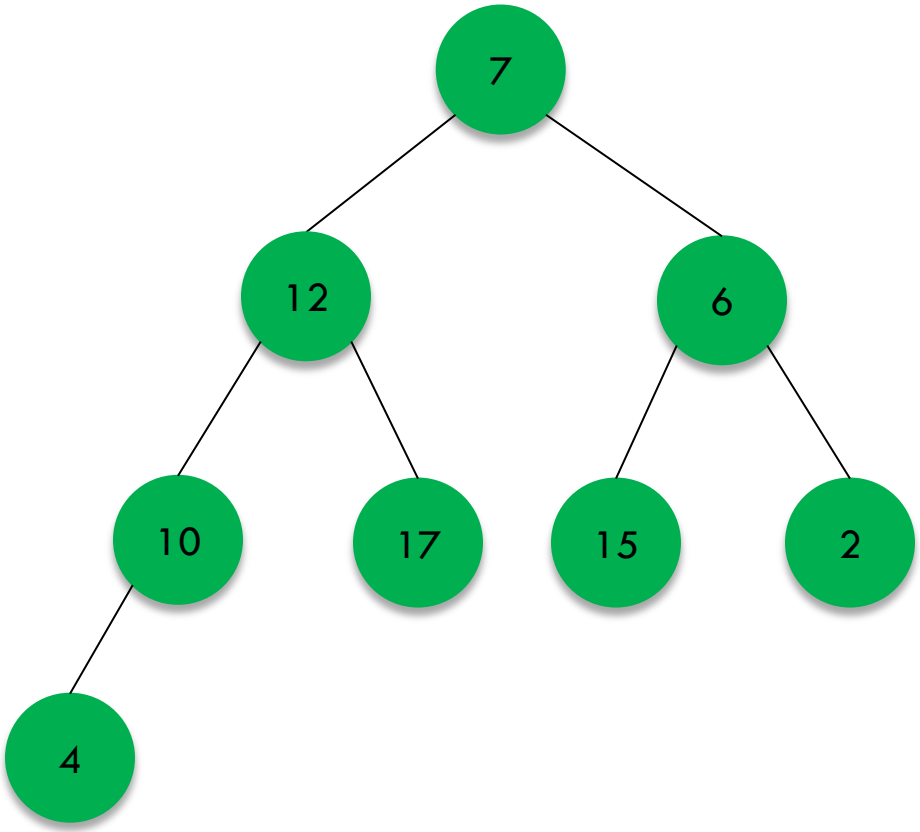
1. Tìm phần tử lớn nhất/nhỏ nhất trên Heap O(1).
2. Thêm một phần tử vào Heap O(logN).
3. Xóa một phần tử trong Heap O(logN).

MINH HỌA CÁC THAO TÁC CƠ BẢN CỦA HEAP

Xây dựng cây Heap

Bước 0: Chuẩn bị dữ liệu.

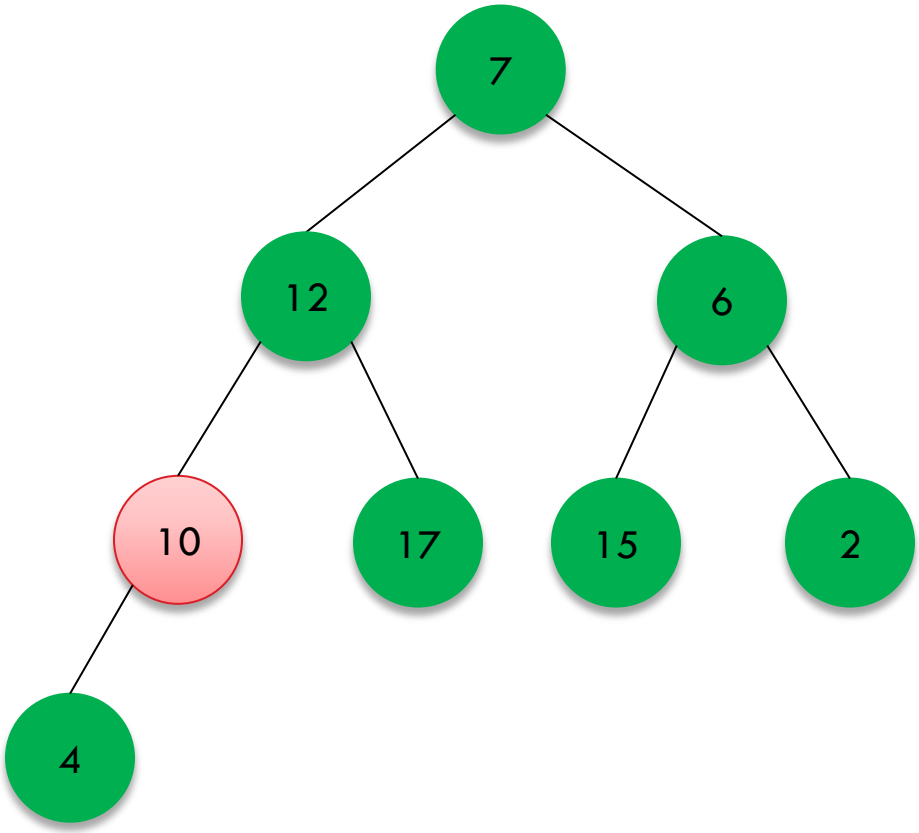




Xây dựng cây Heap

Bước 1: Tìm đến node có vị trí = n/2 – 1.

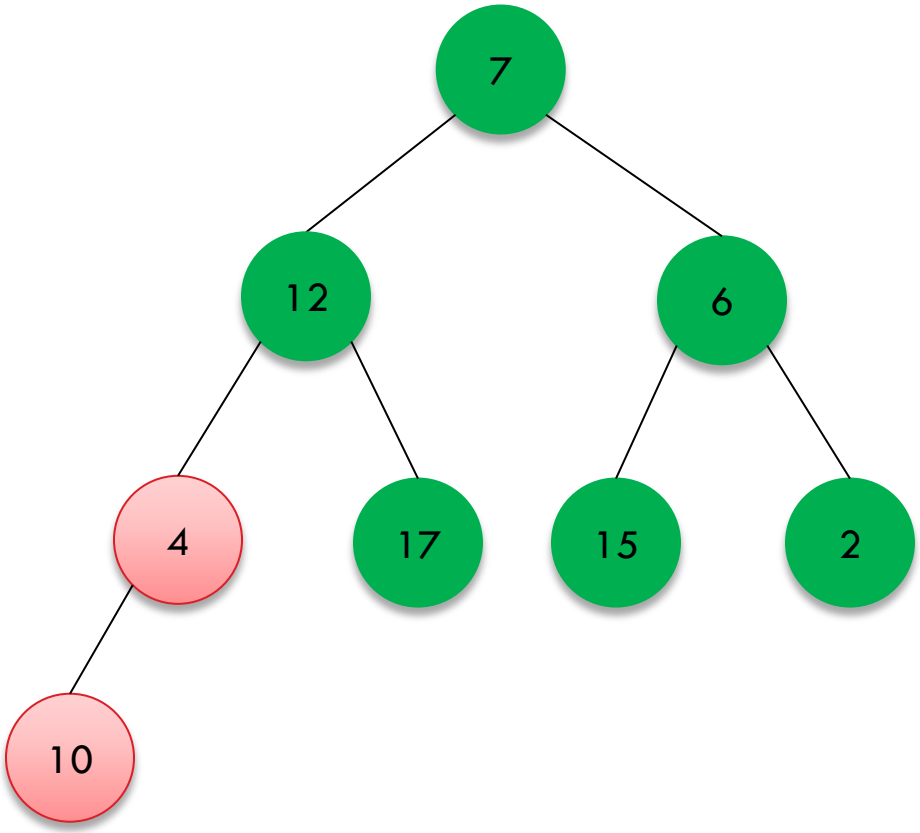
0	1	2	3	4	5	6	7
7	12	6	10	17	15	2	4



Xây dựng cây Heap

Bước 1: Tìm node con của node đó có giá trị nhỏ nhất và đổi chỗ.

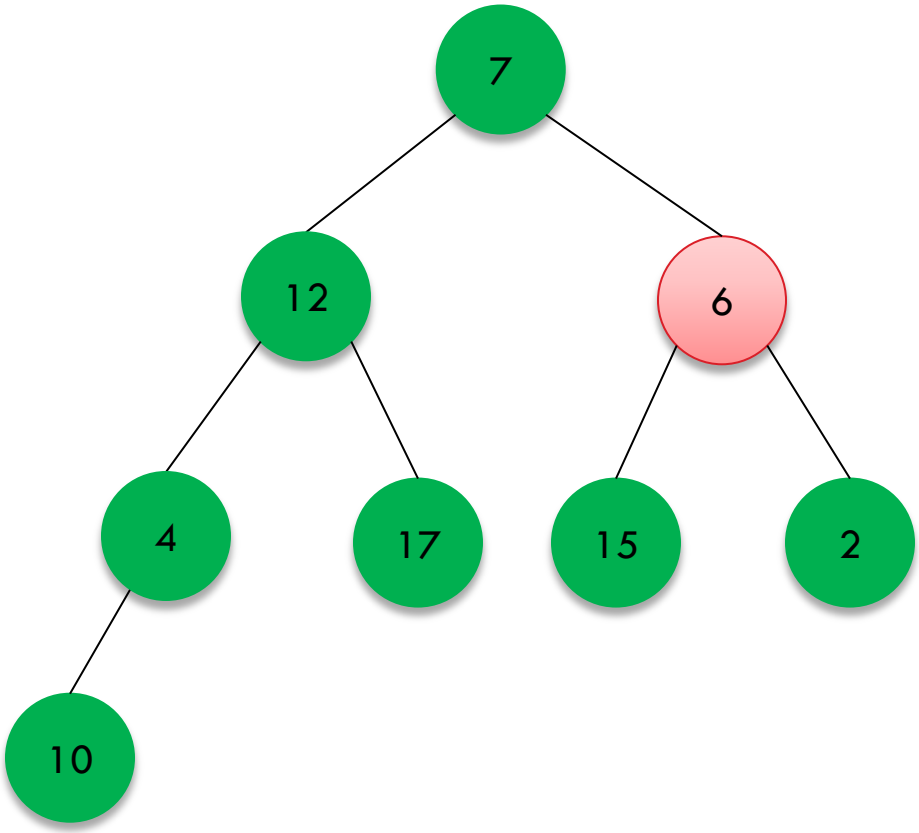
0	1	2	3	4	5	6	7
7	12	6	4	17	15	2	10



Xây dựng cây Heap

Bước 2: Di chuyển lên node phía trên, vị trí 2 node 6.

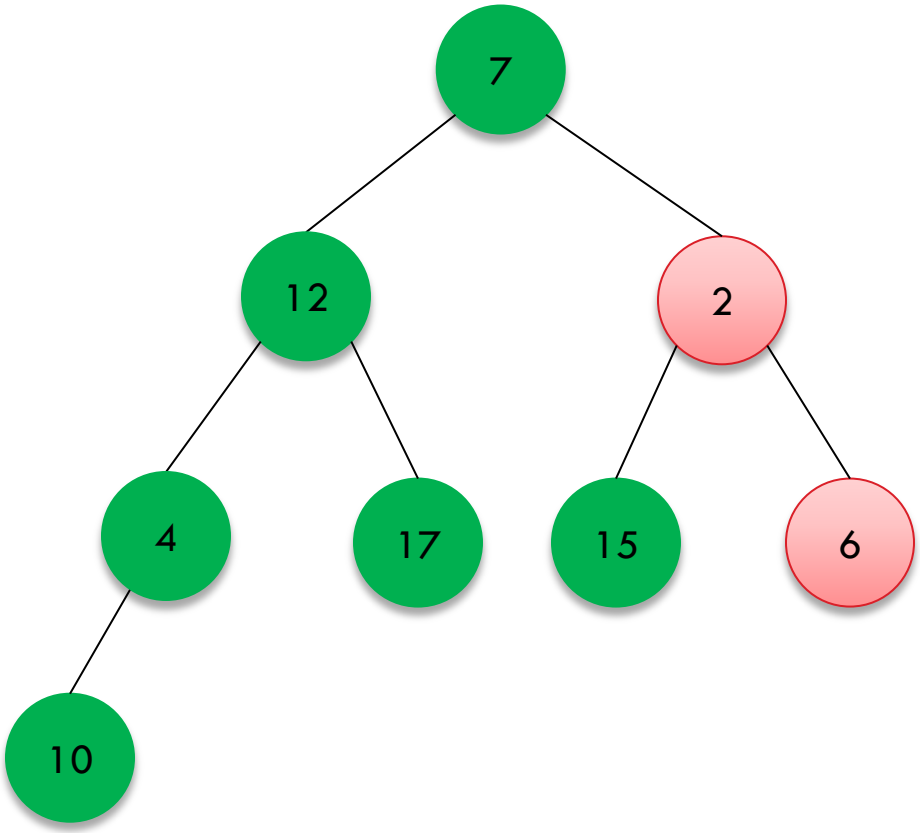
0	1	2	3	4	5	6	7
7	12	6	4	17	15	2	10



Xây dựng cây Heap

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

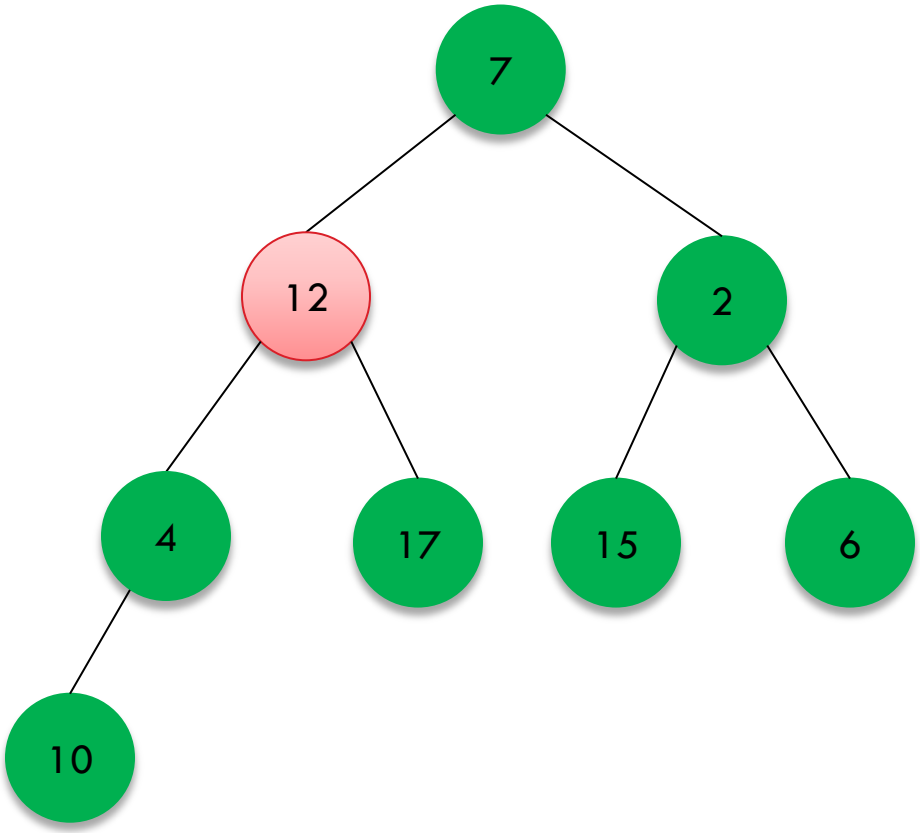
0	1	2	3	4	5	6	7
7	12	2	4	17	15	6	10



Xây dựng cây Heap

Bước 3: Di chuyển lên node phía trên, vị trí 1 node 12.

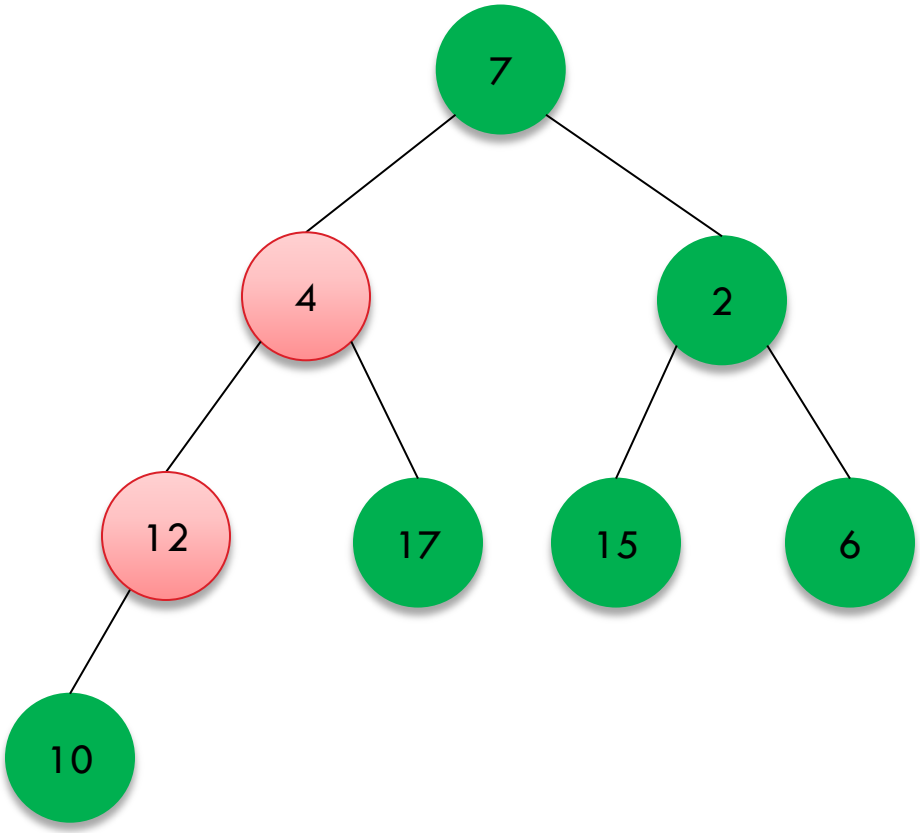
0	1	2	3	4	5	6	7
7	12	2	4	17	15	6	10



Xây dựng cây Heap

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

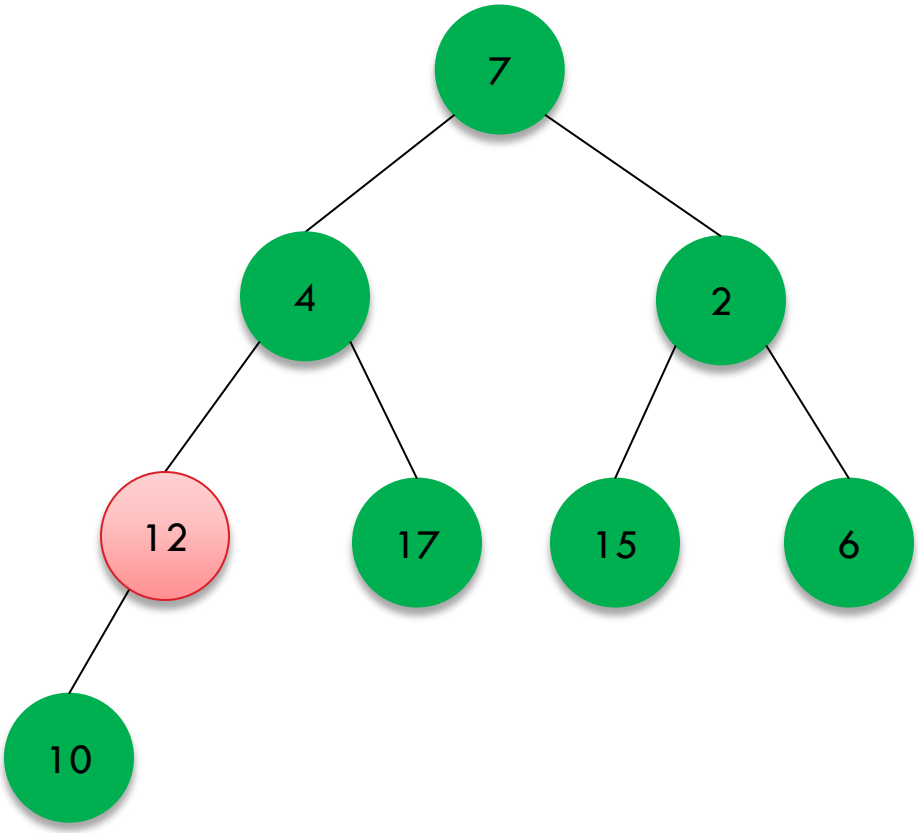
0	1	2	3	4	5	6	7
7	4	2	12	17	15	6	10



Xây dựng cây Heap

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

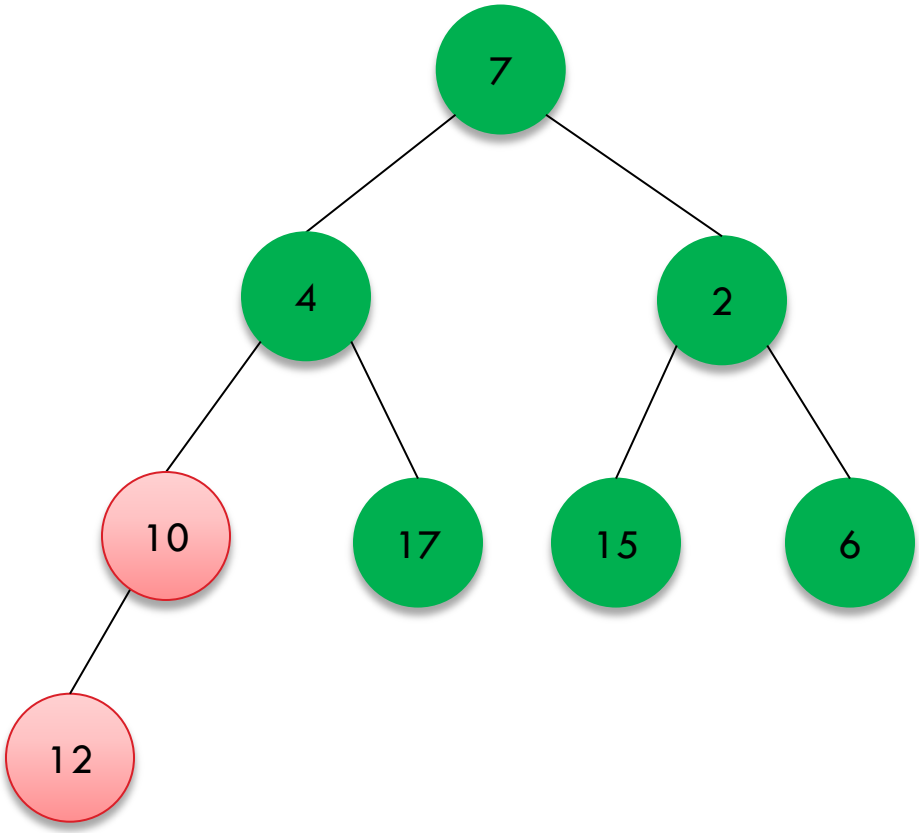
0	1	2	3	4	5	6	7
7	4	2	12	17	15	6	10



Xây dựng cây Heap

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

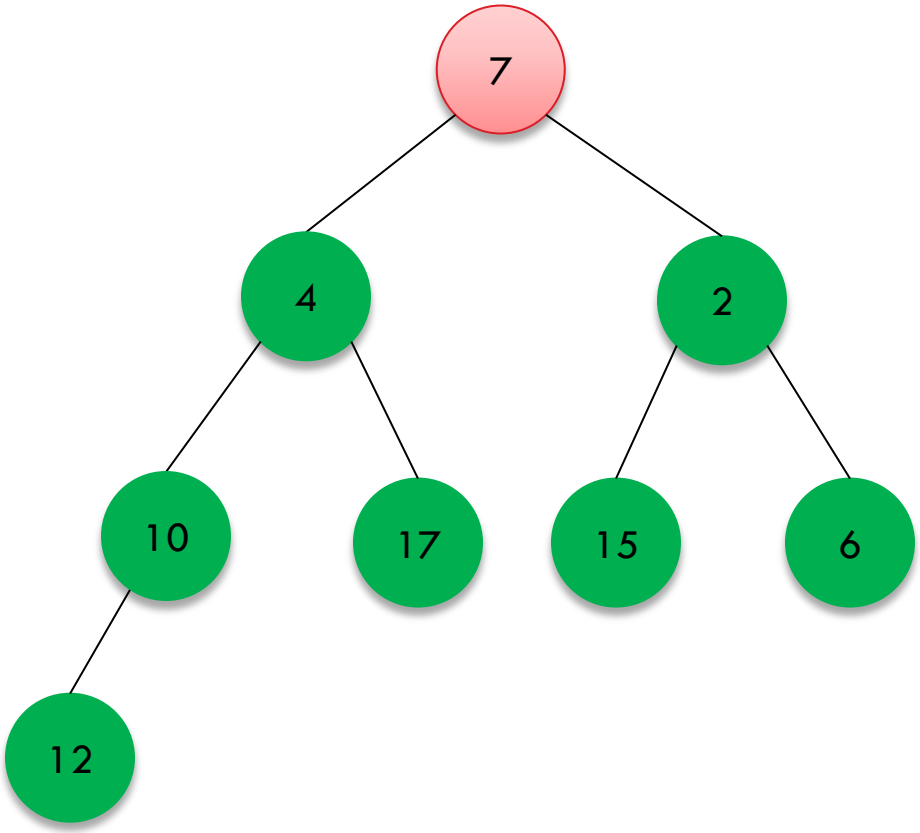
0	1	2	3	4	5	6	7
7	4	2	10	17	15	6	12



Xây dựng cây Heap

Bước 4: Di chuyển lên node phía trên, vị trí 0 node 12.

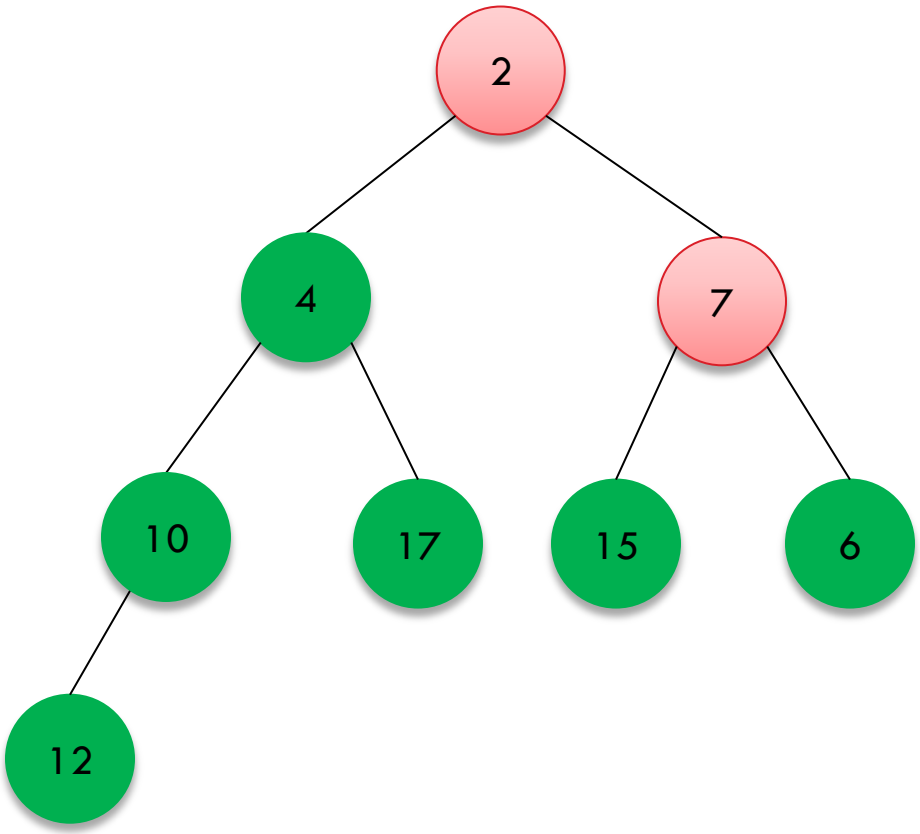
0	1	2	3	4	5	6	7
7	4	2	10	17	15	6	12



Xây dựng cây Heap

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

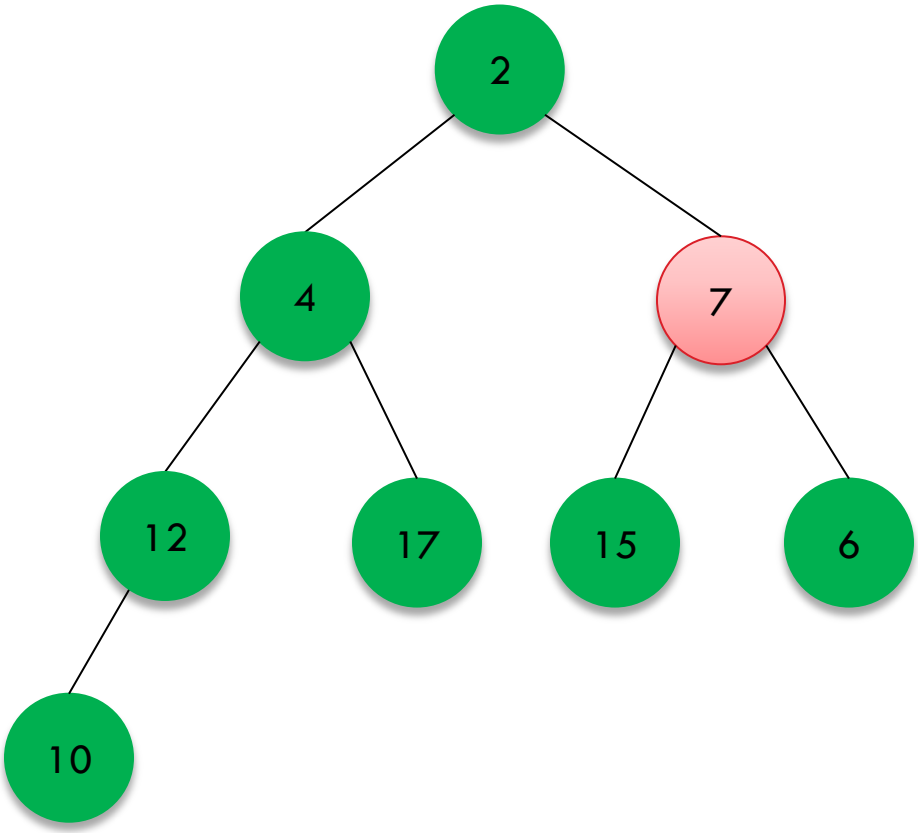
0	1	2	3	4	5	6	7
2	4	7	10	17	15	6	12



Xây dựng cây Heap

Bước 5: Tìm node không đúng vị trí và thay đổi, vị trí 2, node 7.

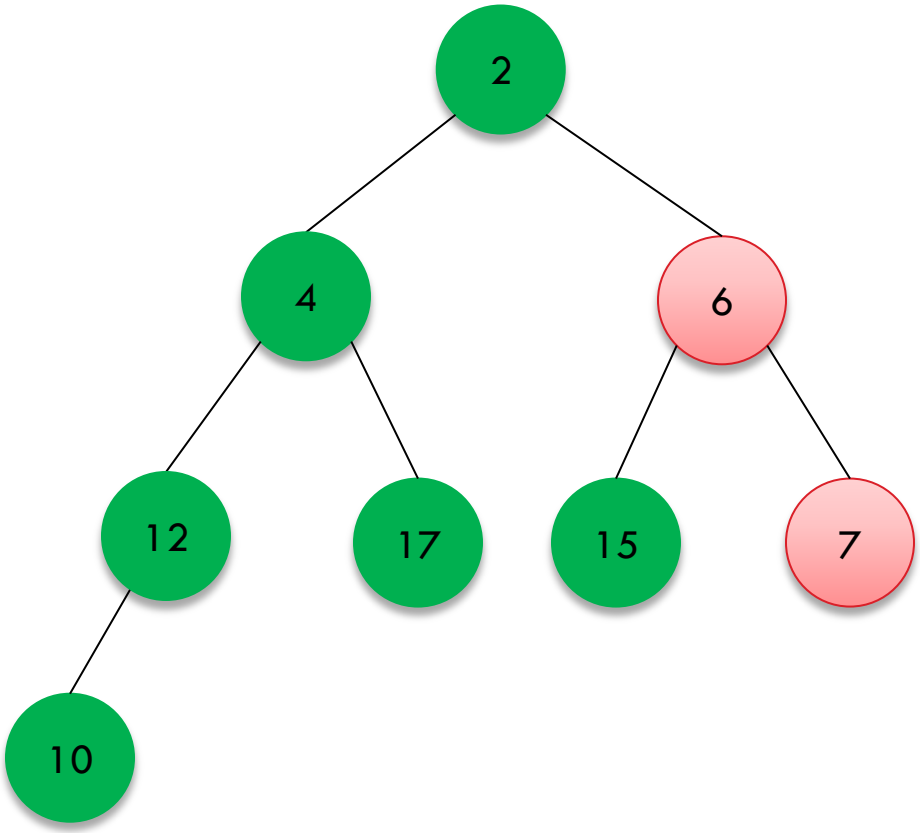
0	1	2	3	4	5	6	7
2	4	7	10	17	15	6	12



Xây dựng cây Heap

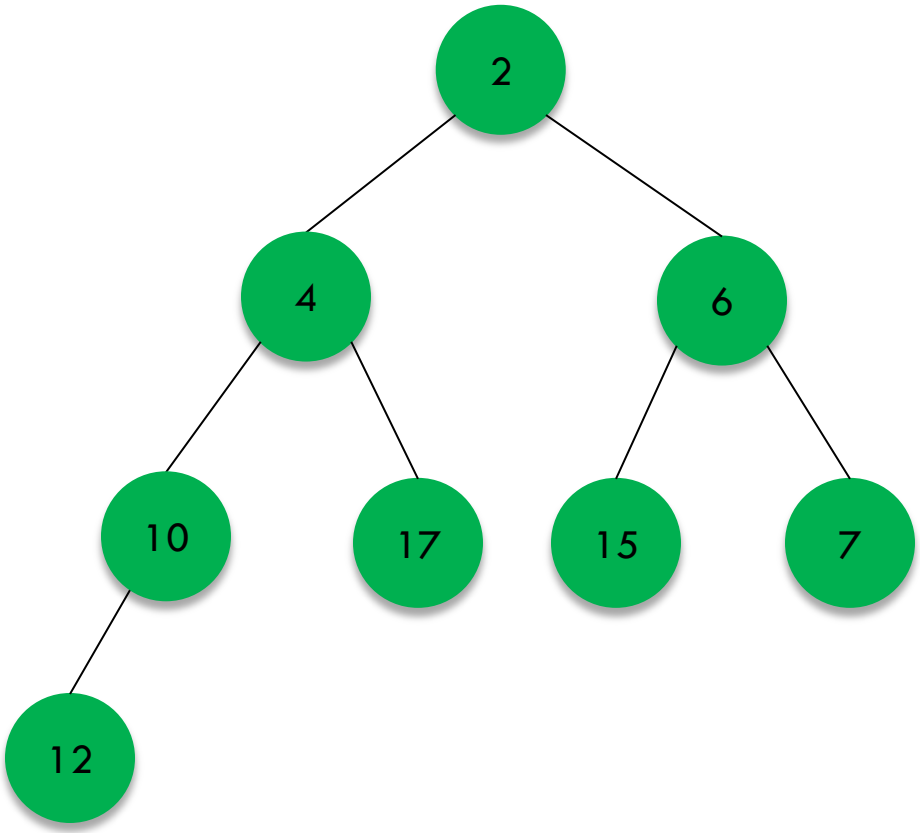
Bước 5: Tìm node con có giá trị nhỏ nhất và đổi chỗ.

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

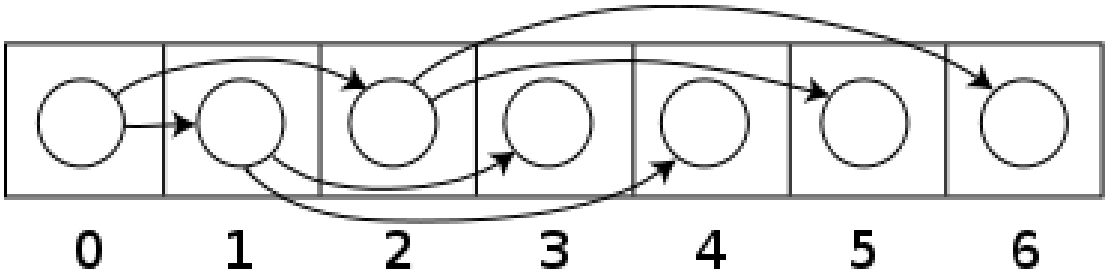


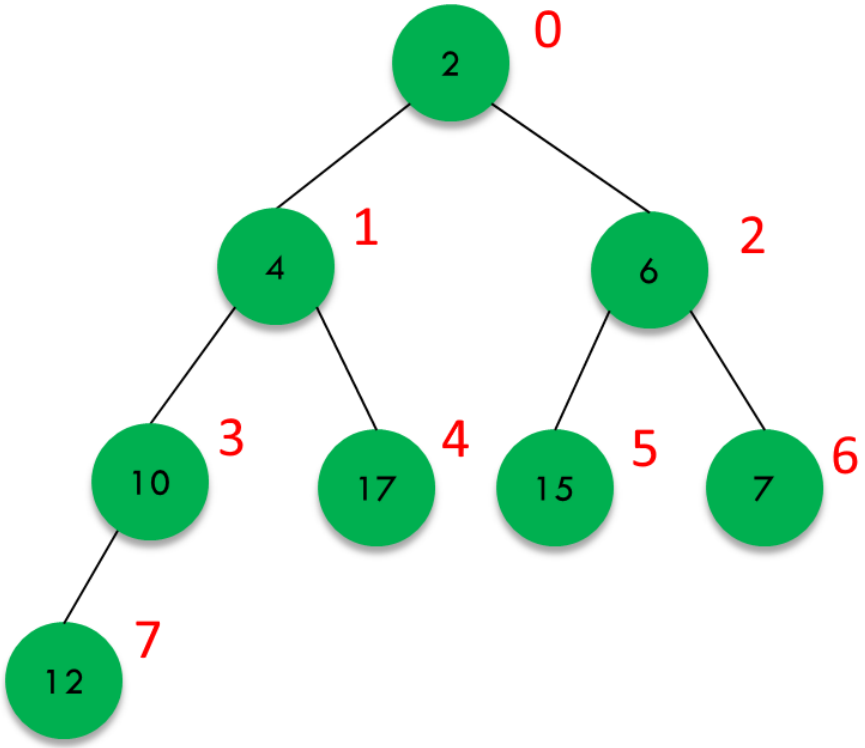
Xây dựng cây Heap

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



Lưu cây Heap trên mảng

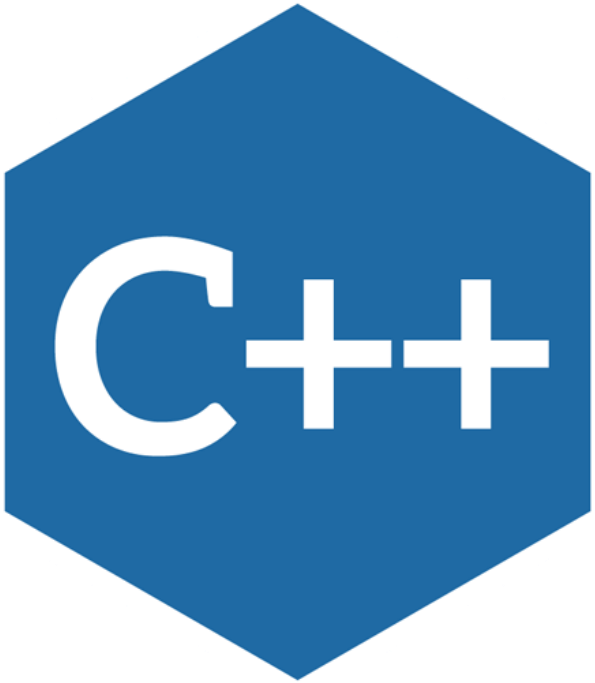




Left = index*2 + 1.

Right = index*2 + 2.

MÃ NGUỒN MINH HỌA BẰNG C++



Mã nguồn minh họa (1) C++

Hàm hóa đổi 2 node với nhau.

```
void swap(int& x, int& y)
{
    int z = x;
    x = y;
    y = z;
}
```

Mã nguồn minh họa (2) C++

Hàm xây dựng một minHeap

```
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);
    }
}
```

Mã nguồn minh họa (3) C++

Hàm xây dựng bắt đầu từ vị trí cuối cùng có node lá.

```
void buildHeap(int n)
{
	for (int i = n / 2 - 1; i >= 0; i--)
	{
		MinHeapify(i);
	}
}
```

Mã nguồn minh họa (4) C++

Hàm main chương trình.

```
int main()
{
     vector<int> a = { 7, 12, 6, 10, 17, 15, 2, 4 };
    h = a;
    buildHeap(h.size());
   return 0;
}
```

MÃ NGUỒN MINH HỌA BẰNG PYTHON

Mã nguồn minh họa (1) - python

Hàm xây dựng một minHeap

```
def MinHeapify(i):
    smallest = i
    left = 2*i + 1
    right = 2*i + 2
    if left < len(h) and h[left] < h[smallest]:
        smallest = left
    if right < len(h) and h[right] < h[smallest]:
        smallest = right
    if smallest != i:
        h[i], h[smallest] = h[smallest], h[i]
        MinHeapify(smallest)
```

Mã nguồn minh họa (2) - python

Hàm xây dựng bắt đầu từ vị trí cuối cùng có node lá.

```
def buildHeap(n):
    for i in range(n//2 - 1, -1, -1):
        MinHeapify(i)
```

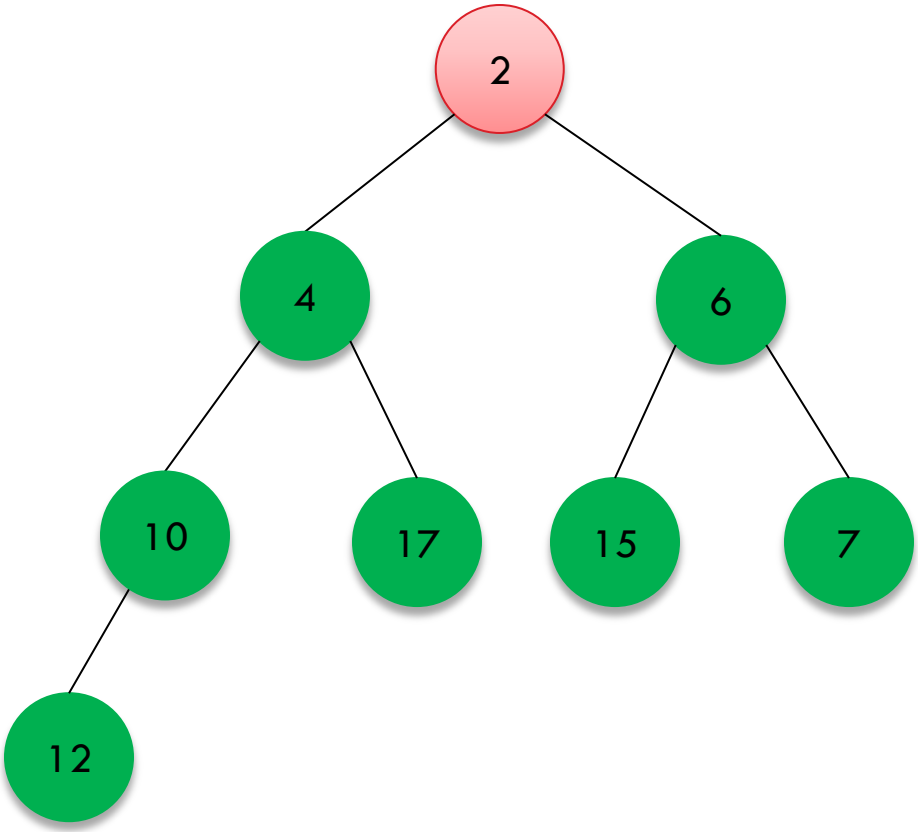
Hàm main chương trình.

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

Tìm phần tử nhỏ nhất của Heap

Trả về vị trí đầu tiên của mảng, vị trí chứa giá trị nhỏ nhất.

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



Mã nguồn minh họa C++

Trả về vị trí đầu tiên của Heap

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

Mã nguồn minh họa python

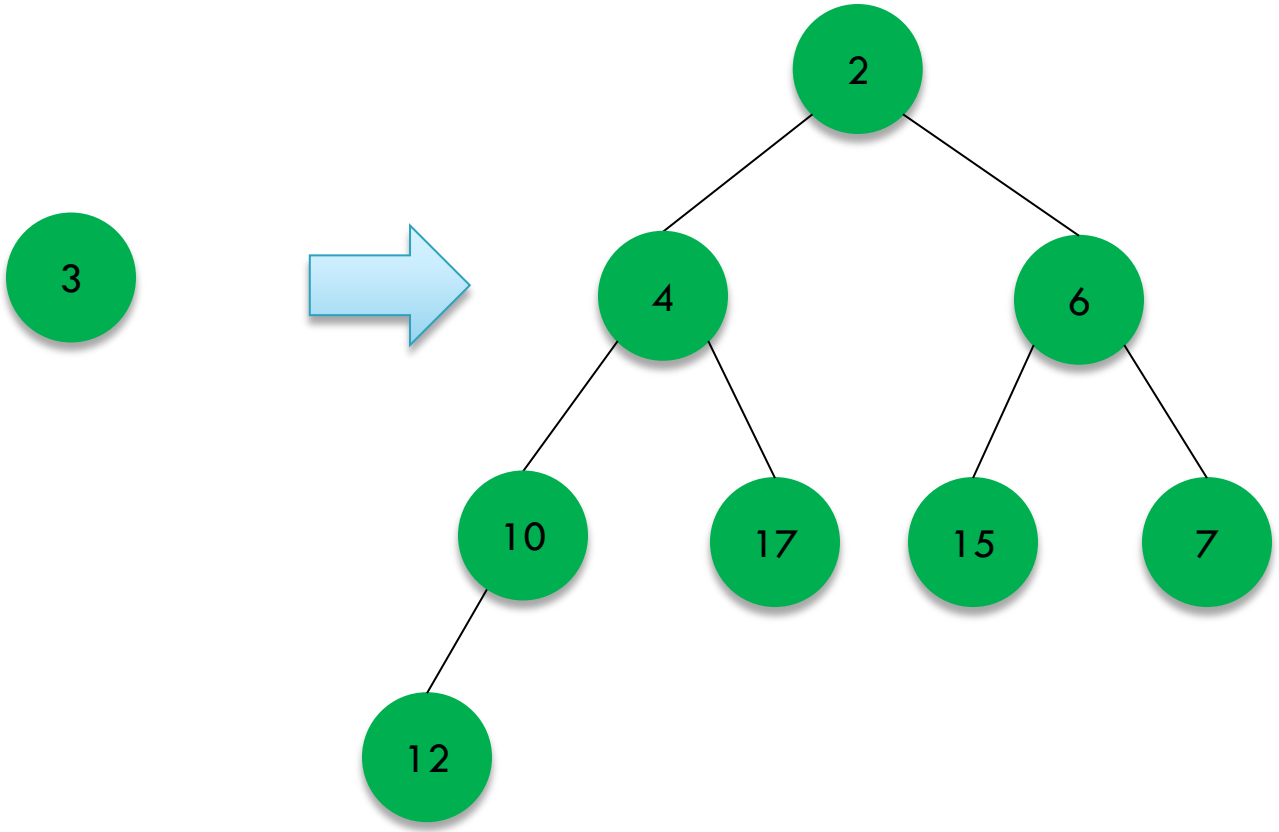
Trả về vị trí đầu tiên của Heap

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

Thêm phần tử vào trong Heap

Thêm phần tử value = 3 vào Heap.

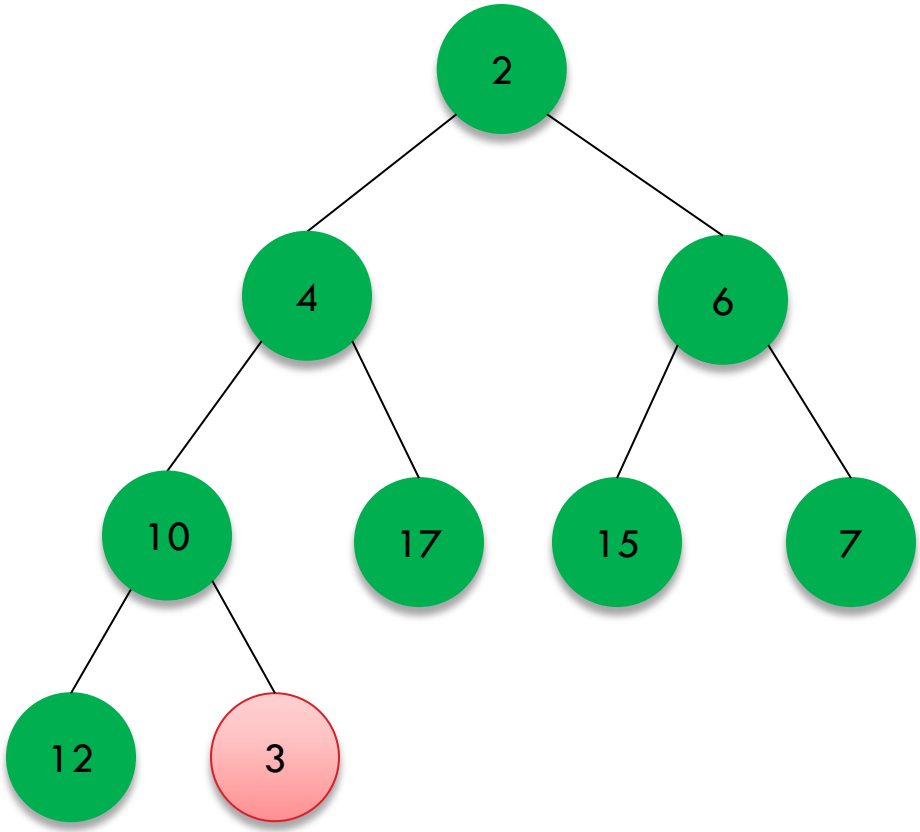
0	1	2	3	4	5	6	7
2	4	6	10	17	15	7	12



Thêm phần tử vào trong Heap

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

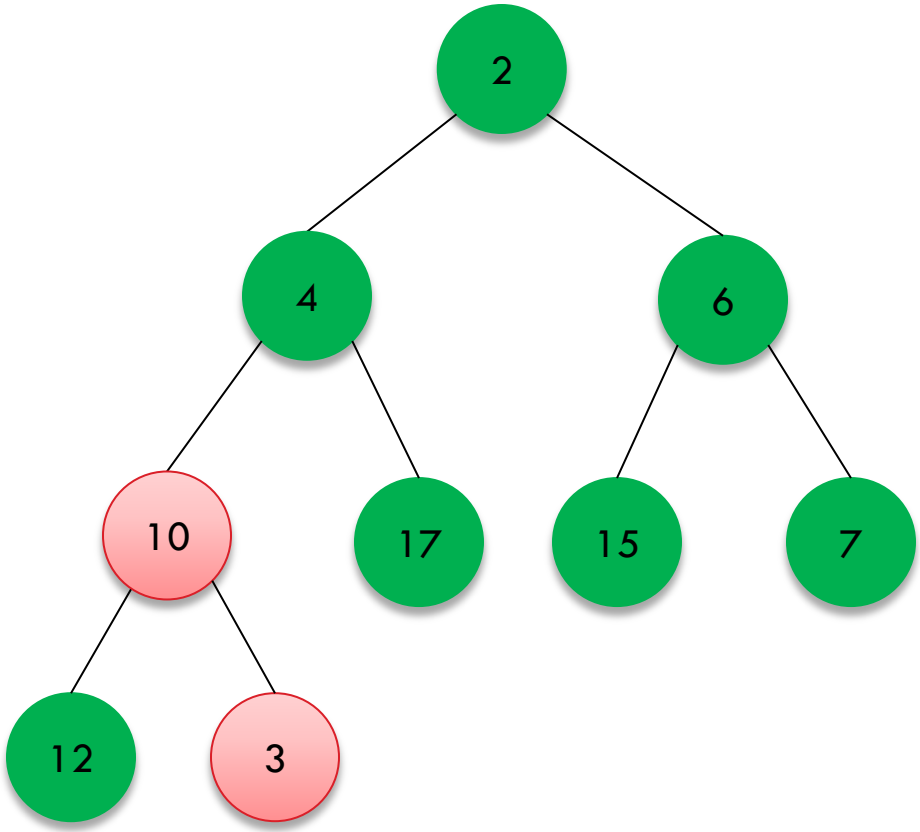
0	1	2	3	4	5	6	7	8
2	4	6	10	17	15	7	12	3



Thêm phần tử vào trong Heap

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

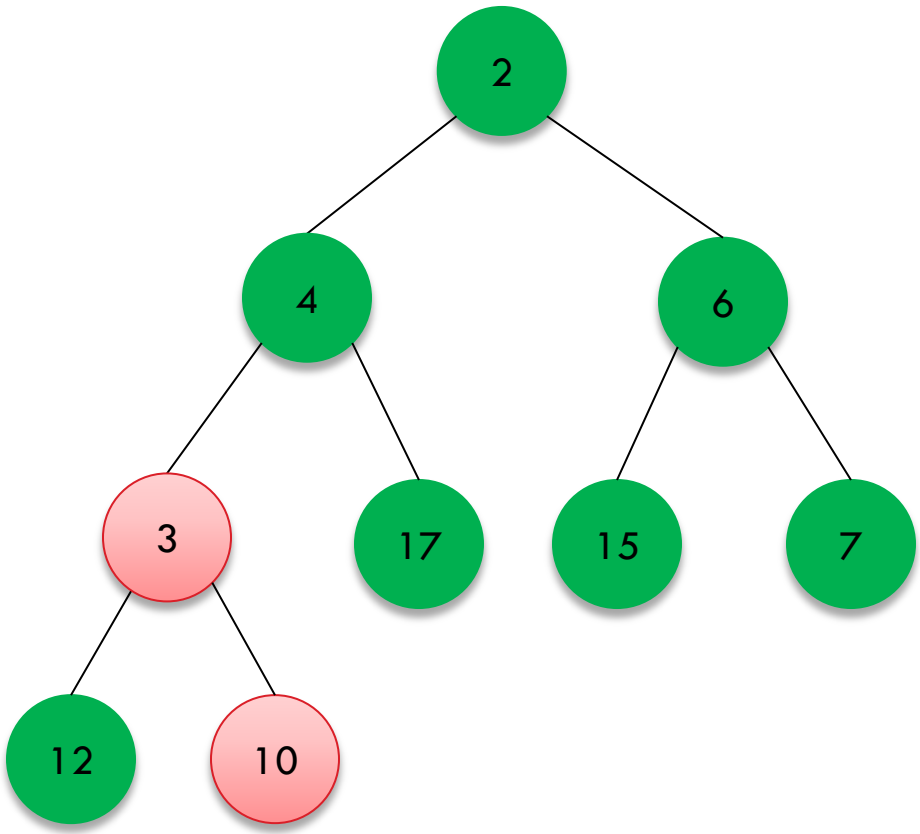
0	1	2	3	4	5	6	7	8
2	4	6	10	17	15	7	12	3



Thêm phần tử vào trong Heap

Bước 2: Hoán đổi node 3 và node 10 với nhau.

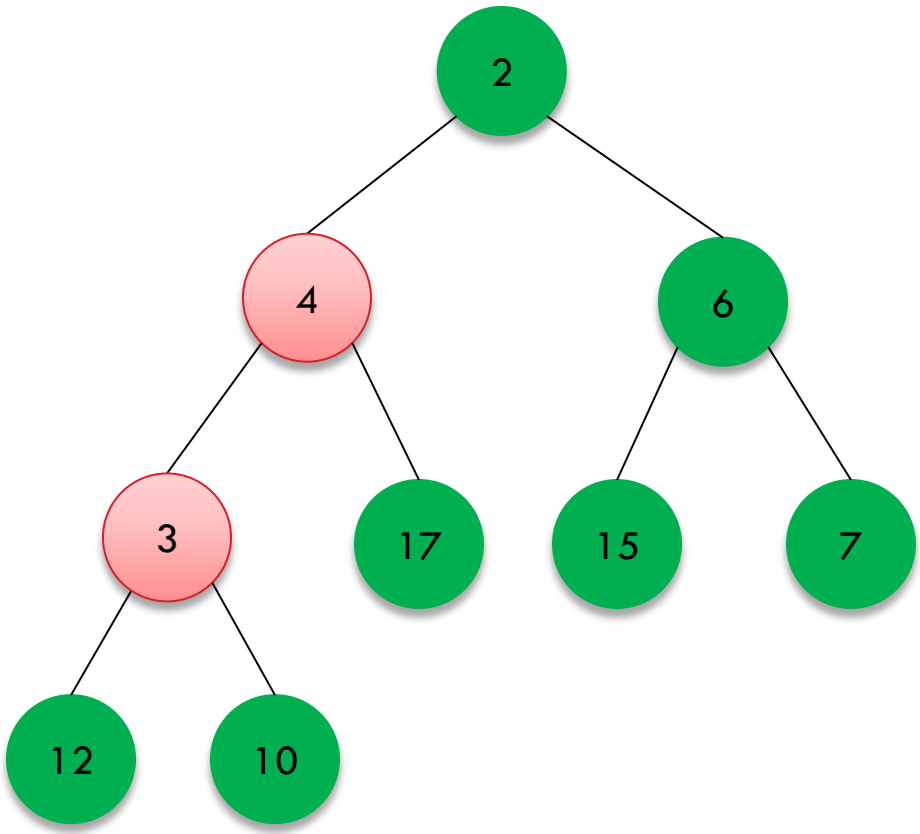
0	1	2	3	4	5	6	7	8
2	4	6	3	17	15	7	12	10



Thêm phần tử vào trong Heap

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

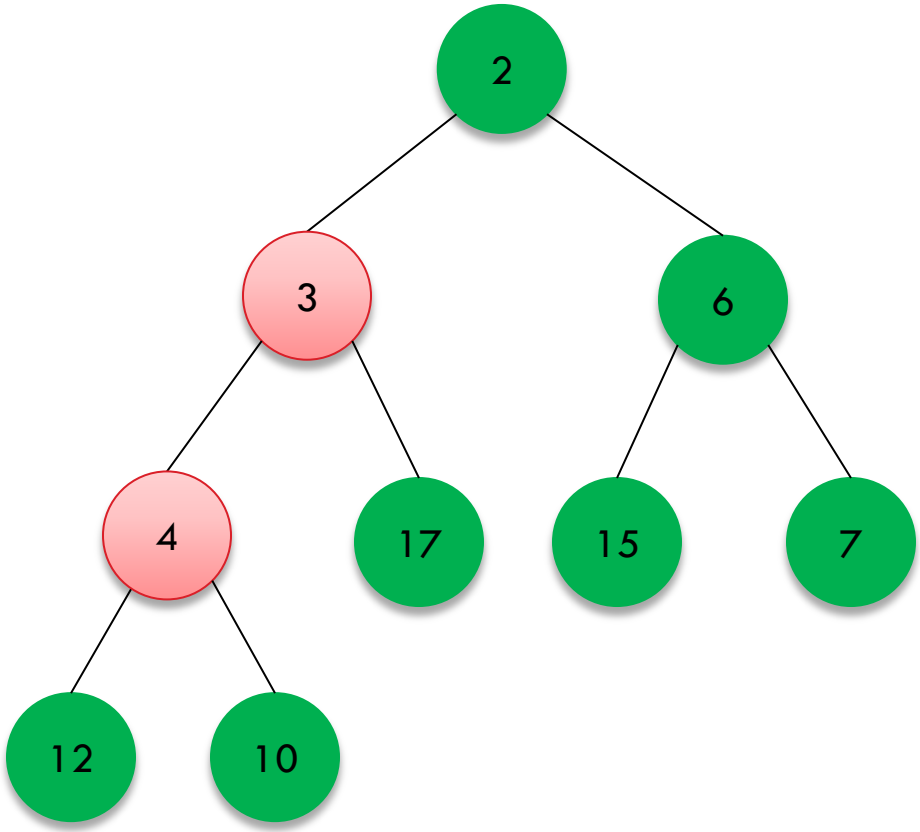
0	1	2	3	4	5	6	7	8
2	4	6	3	17	15	7	12	10



Thêm phần tử vào trong Heap

Bước 3: Hoán đổi node 3 và node 4 với nhau.

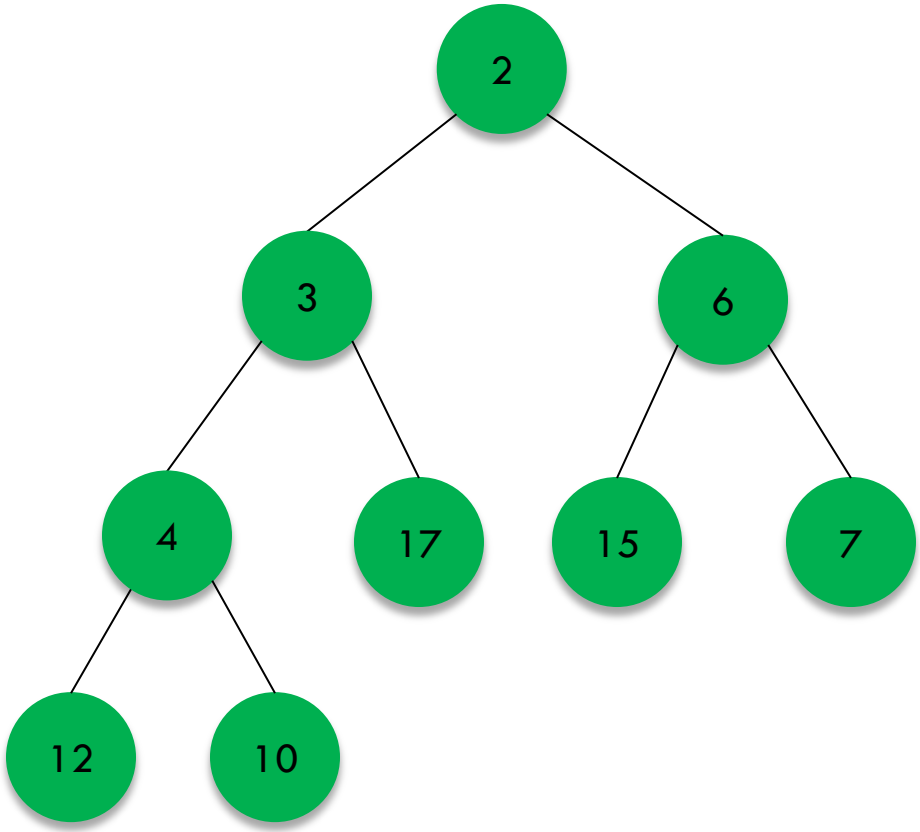
0	1	2	3	4	5	6	7	8
2	3	6	4	17	15	7	12	10



Thêm phần tử vào trong 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



Mã nguồn minh họa C++

Thêm phần tử vào trong Heap.

```
void push(int value)
{
    h.push_back(value);
    int i = h.size() - 1;
    while (i != 0 && h[(i - 1) / 2] > h[i])
    {
        swap(h[i], h[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
}
```

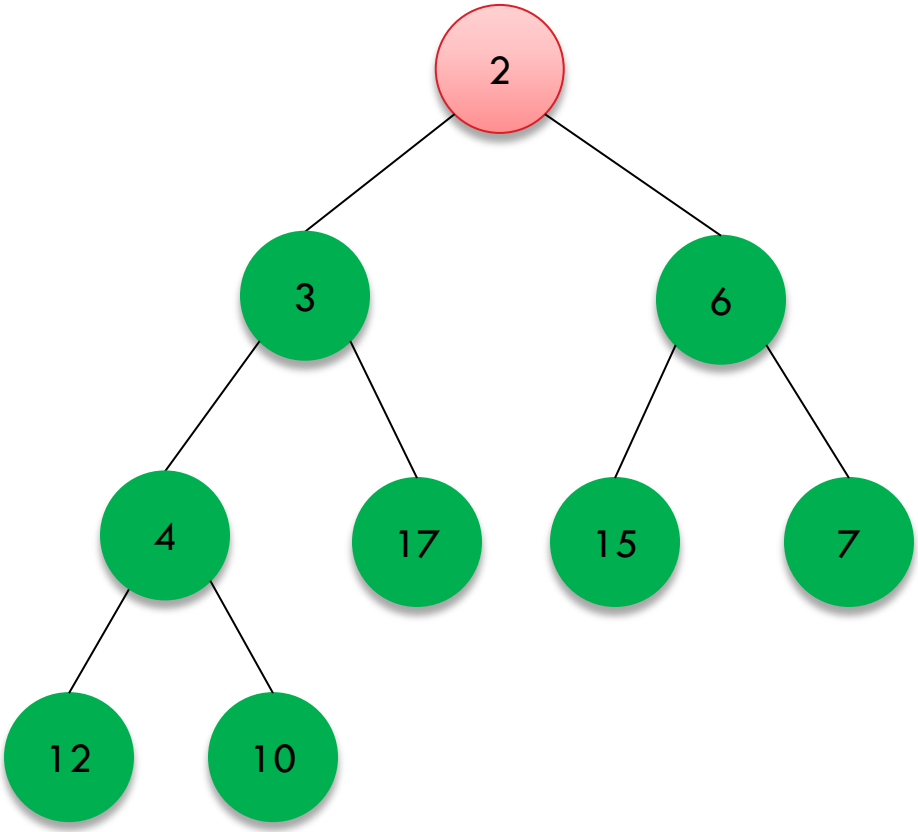
Mã nguồn minh họa python

Thêm phần tử vào trong Heap.

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

Xóa phần tử ra khỏi Heap

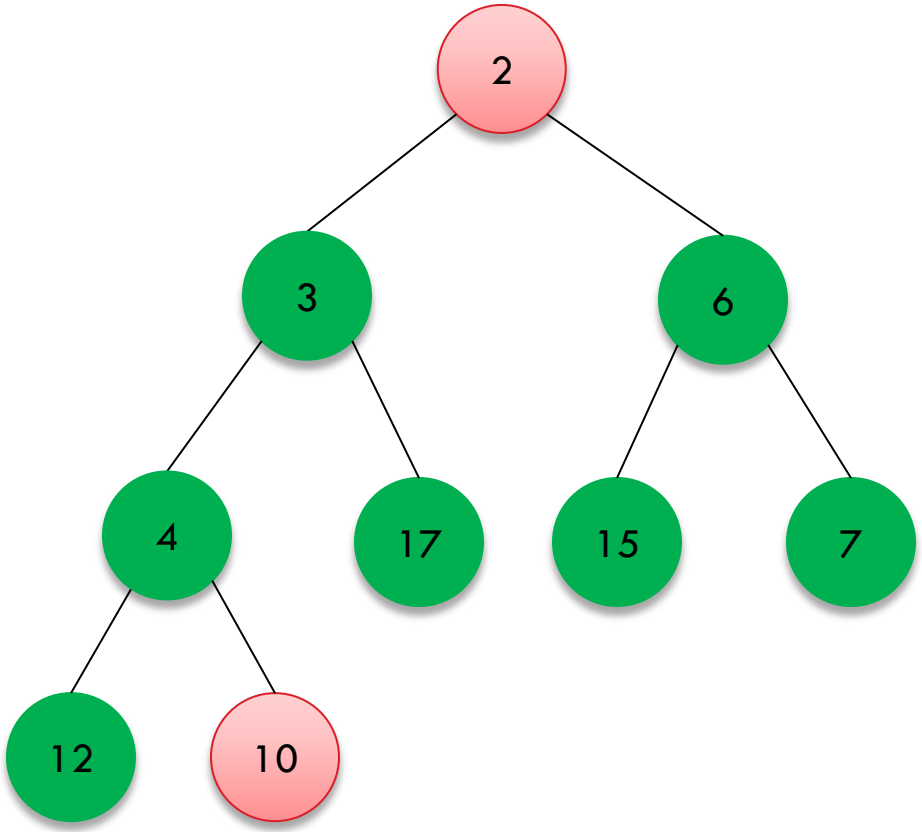
0	1	2	3	4	5	6	7	8
2	4	6	3	17	15	7	12	10



Xóa phần tử ra khỏi Heap

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

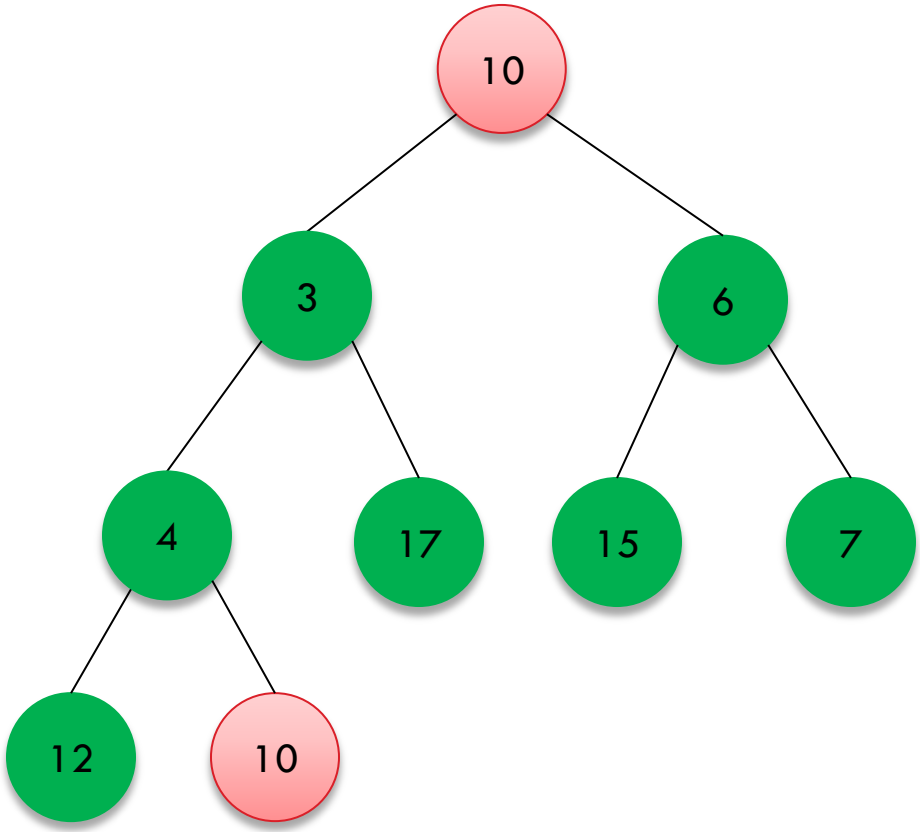
0	1	2	3	4	5	6	7	8
2	**4**	**6**	**3**	**17**	**15**	**7**	**12**	**10**



Xóa phần tử ra khỏi Heap

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

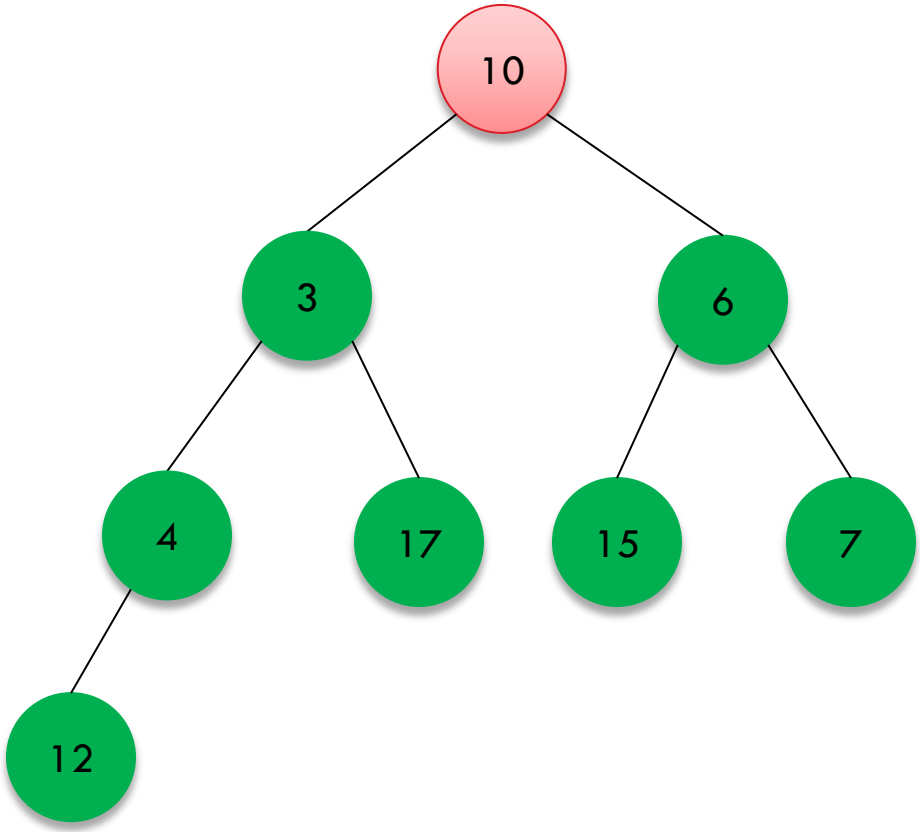
0	1	2	3	4	5	6	7	8
10	**4**	**6**	**3**	**17**	**15**	**7**	**12**	**10**



Xóa phần tử ra khỏi Heap

Bước 2: Xóa phần tử cuối Heap.

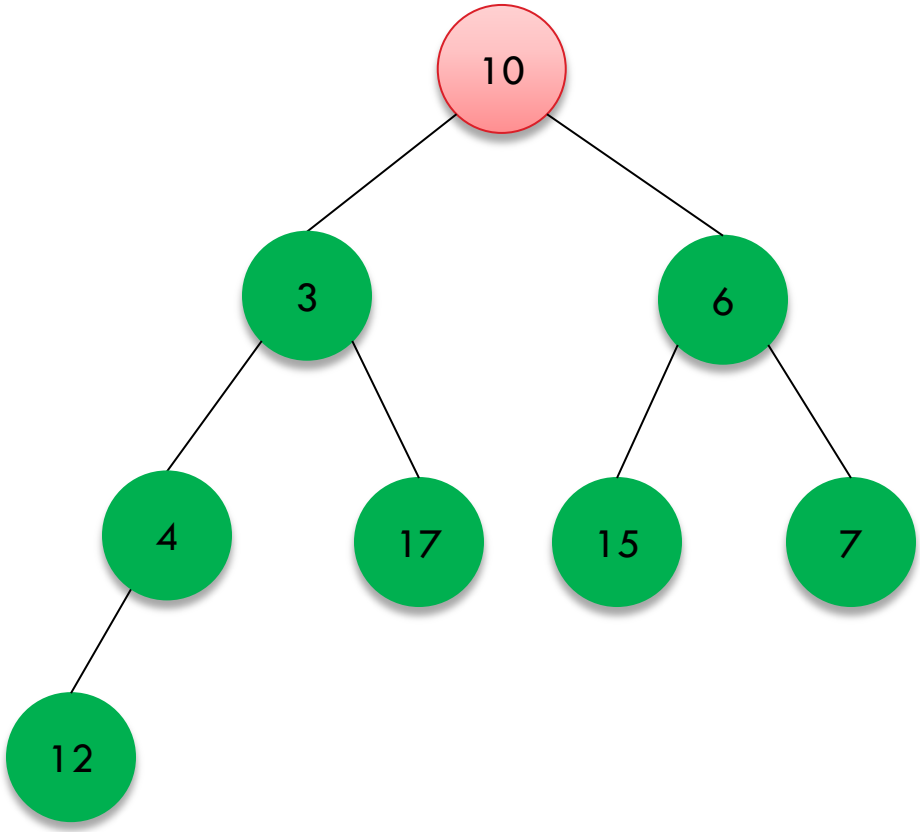
0	1	2	3	4	5	6	7
10	4	6	3	17	15	7	12



Xóa phần tử ra khỏi Heap

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

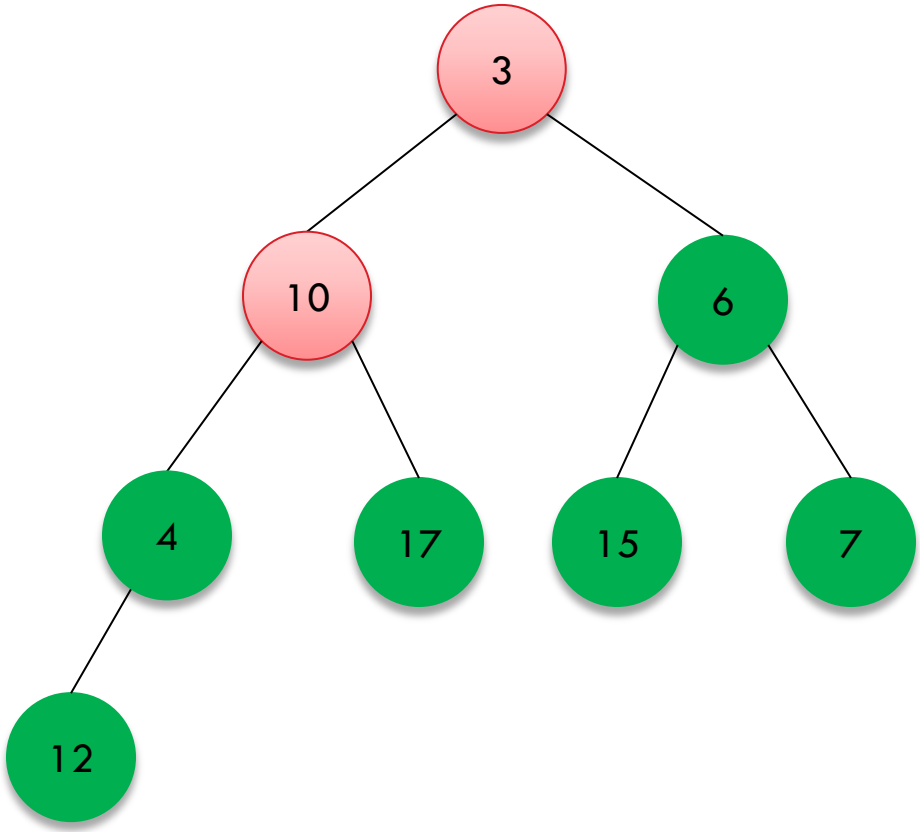
0	1	2	3	4	5	6	7
10	4	6	3	17	15	7	12



Xóa phần tử ra khỏi Heap

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

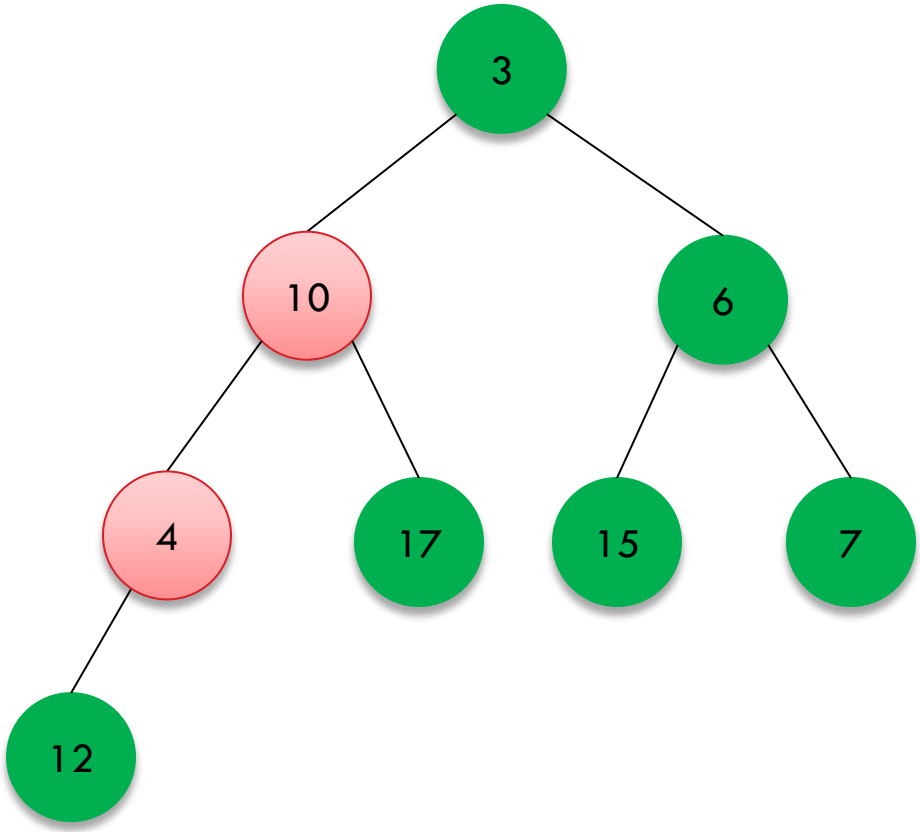
0	1	2	3	4	5	6	7
3	**4**	**6**	**10**	**17**	**15**	**7**	**12**



Xóa phần tử ra khỏi Heap

Bước 4: Tiếp tục cân bằng ở phần tử tiếp theo.

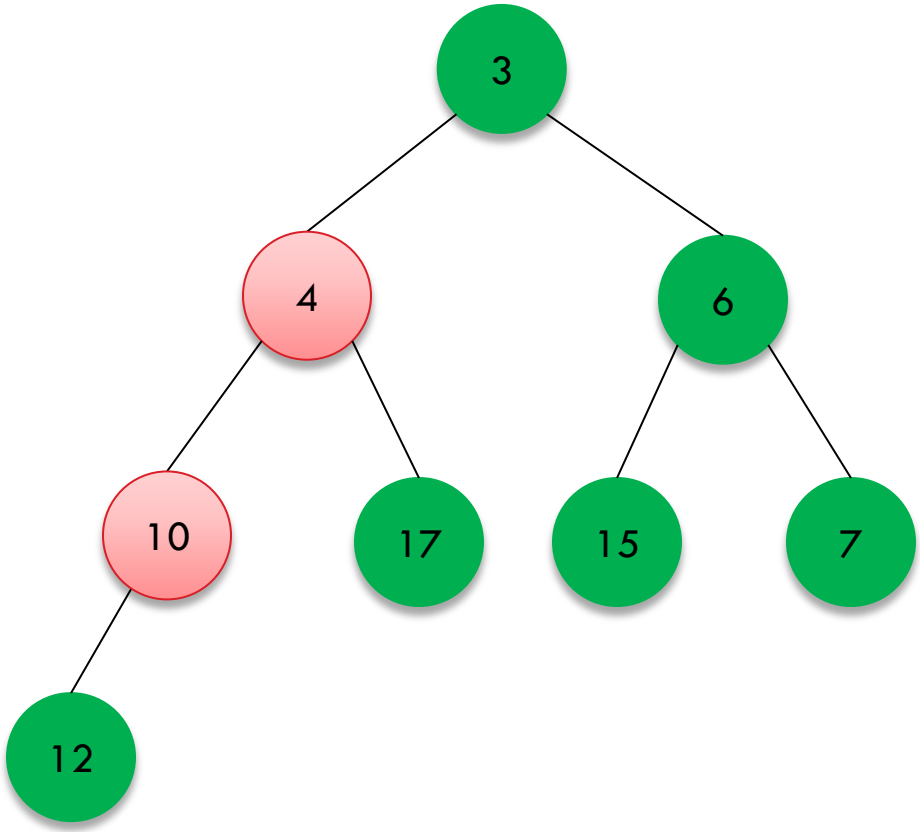
0	1	2	3	4	5	6	7
3	10	6	4	17	15	7	12



Xóa phần tử ra khỏi Heap

Bước 4: Hoán đổi 2 phần tử với nhau.

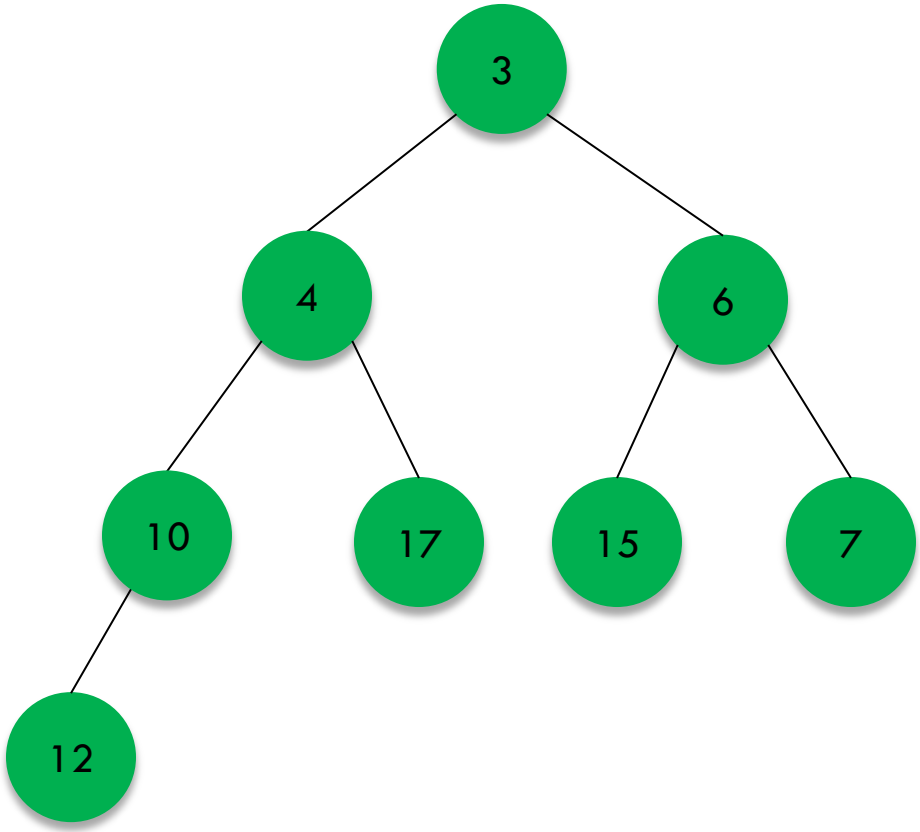
0	1	2	3	4	5	6	7
3	4	6	10	17	15	7	12



Xóa phần tử ra khỏi 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



Mã nguồn minh họa C++

Xóa phần tử ở đầu Heap.

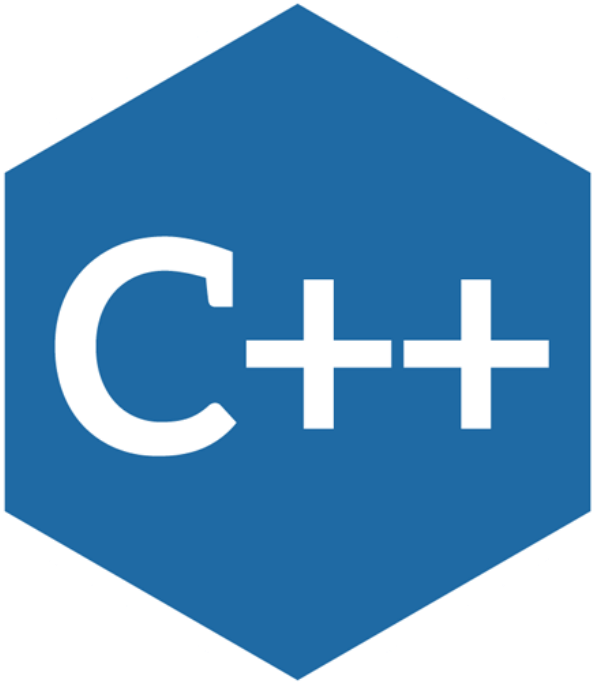
```
void pop()
{
    int length = h.size();
    if (length == 0)
    {
        return;
    }
    h[0] = h[length - 1];
    h.pop_back();
    MinHeapify(0);
}
```

Mã nguồn minh họa python

Xóa phần tử ở đầu Heap.

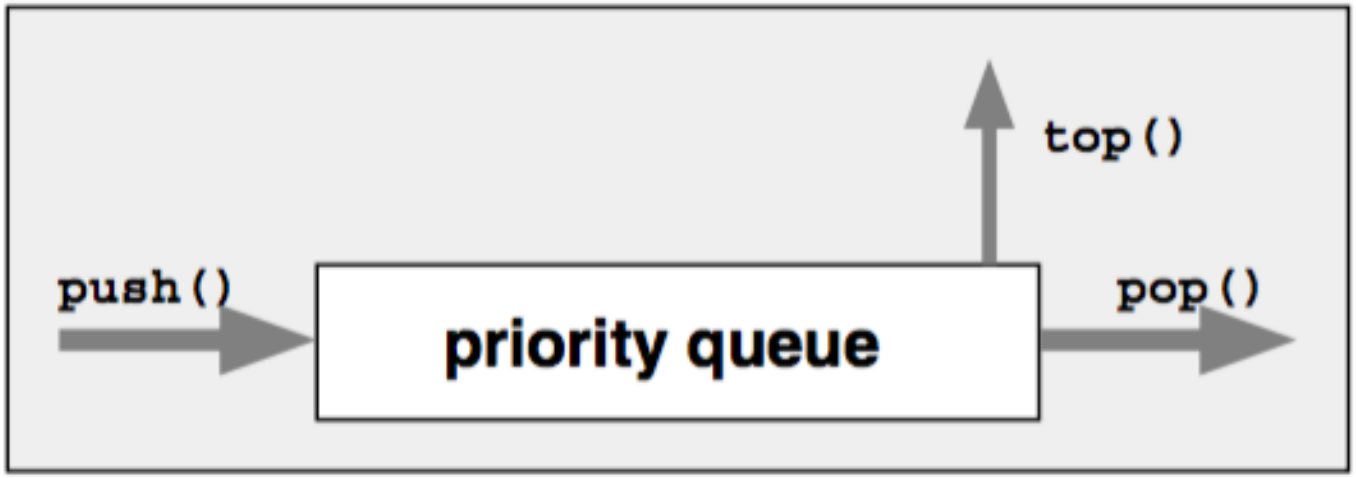
```
def pop():
    length = len(h)
    if length == 0:
        return
    h[0] = h[length - 1]
    h.pop()
    MinHeapify(0)
```

SỬ DỤNG HEAP BẰNG THƯ VIỆN STL



Định nghĩa priority_queue

priority_queue là hàng đợi ưu tiên, được thiết kế đặc biệt để 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)

Thư viện:

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

Khai báo: khai báo dạng mặc định.

```
priority_queue<data_type> variable_name;
```

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]);
}
```

Kết quả

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]);
}
```

Kết quả

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

Các hàm thành viên của priority_queue

push: Thêm một phần tử vào priority_queue.

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

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

Kết quả

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

Các hàm thành viên của priority_queue

pop: Xóa một phần tử đầu trong priority_queue.

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

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

Kết quả

0	1	2	3	4	5	6	7
3	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.

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

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

2

Một số hàm thành viên khác

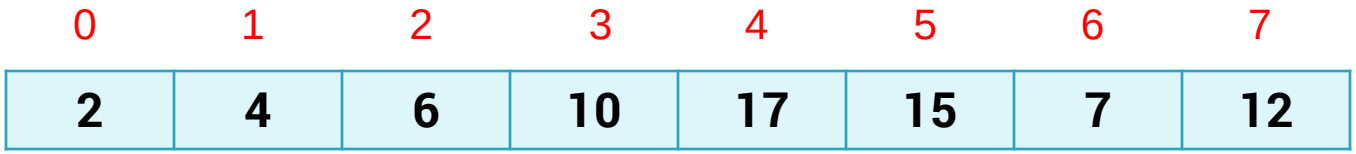
size: Trả về số lượng phần tử hiện tại có trong hàng đợi.

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

swap: Hóa đổi 2 hàng đợi ưu tiên với nhau.

Xóa toàn bộ phần tử trong priority_queue

Dùng phương pháp.

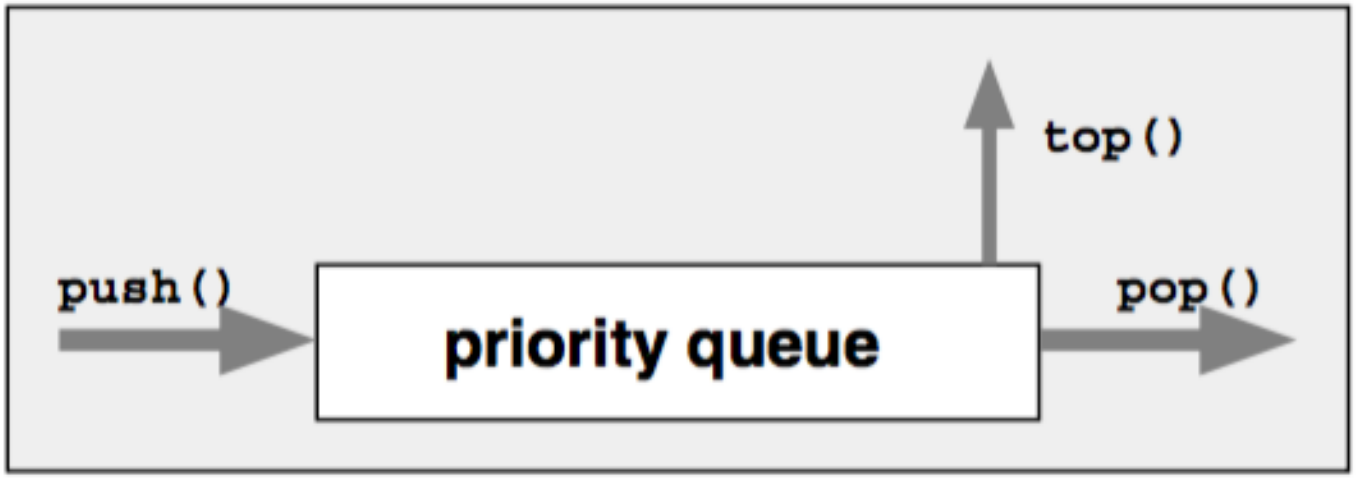



```
pq = priority_queue <int>();
```

SỬ DỤNG HEAP BẰNG THƯ VIỆN QUEUE TRONG PYTHON

Định nghĩa PriorityQueue

PriorityQueue là hàng đợi ưu tiên, được thiết kế đặc biệt để 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)

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

```
import queue
```

Khai báo: khai báo dạng mặc định.

```
variable_name = queue.PriorityQueue()
```

Khai báo và sử dụng (2)

Cách 1: sử dụng như **max-heap**: đảo dấu giá trị truyền vào heap để tạo max-heap, hoặc tạo một class mới và define operator __lt__ cho object đó.

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

Kết quả

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

Khai báo và sử dụng (2)

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)
```

Kết quả

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

Các hàm thành viên của PriorityQueue

put: Thêm một phần tử vào PriorityQueue.

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

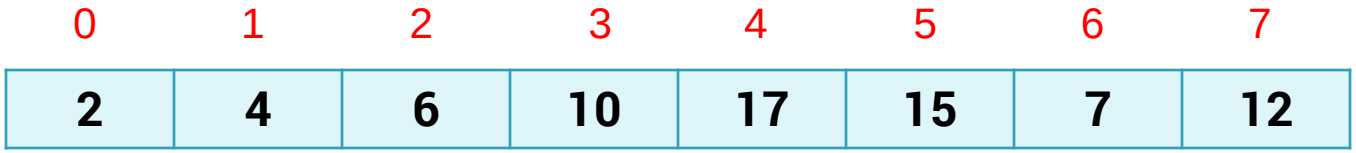
```
pq.put(3);
```

Kết quả

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

Các hàm thành viên của PriorityQueue

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




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

2

Một số hàm thành viên khác

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

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

Ví dụ: len(pq.queue)

Hỏi đáp