

Tên bài giảng Cấu trúc dữ liệu nâng cao

Môn học: **Thuật toán và ứng dụng**
Chương: 2
Hệ: Đại học
Giảng viên: TS. Phạm Đình Phong
Email: phongpd@utc.edu.vn

Nội dung bài học

1. Cấu trúc dữ liệu tập hợp
2. Cấu trúc dữ liệu hàng đợi
3. Cấu trúc dữ liệu ánh xạ
4. Cấu trúc dữ liệu đống
5. Cấu trúc dữ liệu cây nâng cao

Cấu trúc dữ liệu tập hợp (set)

- Khái niệm
 - Các phần tử cùng kiểu dữ liệu
 - Không trùng nhau
- Ví dụ
 - Mảng a = {1, 2, 1, 1, 2}
 - Tập hợp s = {1, 2}

Cấu trúc dữ liệu tập hợp (set)

- Thứ tự các phần tử
 - Mặc định: thứ tự tăng dần
 - Thứ tự giảm dần: set<int, std::greater<int>> s;
 - Hoặc theo định nghĩa:

```
struct cmpStruct {  
    bool operator() (int const & lhs, int const & rhs) const  
    {  
        return lhs > rhs;  
    }  
};
```

```
set<int, cmpStruct > myInverseSortedSet;
```

Cấu trúc dữ liệu tập hợp (set)

- Cài đặt: kiểm tra trùng khi chèn
 - Lớp set
 - Các phần tử được sắp thứ tự
 - Sử dụng cây đỏ đen
 - Thời gian truy cập một phần tử: $O(\log(n))$
 - Lớp unordered_set
 - Các phần tử không được sắp
 - Được cài đặt bằng bảng băm
 - Thời gian truy cập $O(1)$

Cấu trúc dữ liệu tập hợp (set)

- Cài đặt:
 - Một số phương thức
 - insert, erase, swap, clear, size, find
 - Duyệt phần tử

```
set<int>::iterator it;  
  
for (it = s.begin(); it != s.end(); ++it)  
    cout << ' ' << *it;
```

Cấu trúc dữ liệu tập hợp (set)

- Ứng dụng
 - Bài toán liên quan tới tập hợp
 - Ví dụ 1: Cho dãy n phần tử. Hãy đếm số phần tử khác nhau trong mảng.

```
set<int> s;  
for (int i=0; i < n; i++)  
    s.insert(a[i]);  
return s.size();
```

Cấu trúc dữ liệu tập hợp (set)

- **Ứng dụng**

- Ví dụ 2: Một vector v được gọi là vector beautiful nếu một số trong v chỉ xuất hiện đúng một lần. Cần xóa ít nhất bao nhiêu phần tử trong v để v trở thành vector beautiful.
 - + Với $v = [2, 3, 6, 3]$ thì $\text{vectorBeautiful}(v) = 1$
→ chỉ cần xóa số 3.
 - + Với $v = [1, 2, 3]$ thì $\text{vectorBeautiful}(v) = 0$

Sử dụng set s , chèn tất cả phần tử trong v vào s .

Kết quả chính là $v.size() - s.size()$.

Cấu trúc dữ liệu tập hợp (set)

- ## Ứng dụng

Ví dụ 3: Viết chương trình liệt kê các số nguyên tố trong phạm vi từ 1 đến n theo phương pháp Eratosthene (không cần đến các phép nhân).

Thuật toán: xuất phát từ một tập các số nguyên từ 1 đến n. Mỗi lần gấp một số nguyên tố, ta sẽ loại trừ ra khỏi tập này tất cả các số là bội của số nguyên tố này.

(Tương truyền rằng Eratosthene là một nhà bác học thời chưa có giấy viết. Vì vậy bài toán tìm số nguyên tố được ông sáng tác và thực hiện trên mặt đất: ông kẻ một dãy ô, trong đó ghi các số tự nhiên từ 1 đến n. Sau đó, số nào không phải là số nguyên tố thì bị loại ra bằng cách lấy que chọc một lỗ vào đó để xóa nó đi. Kết quả trên mặt đất lỗ chỗ như một cái sàng, sàng lọc các số)

Cấu trúc dữ liệu hàng đợi (queue)

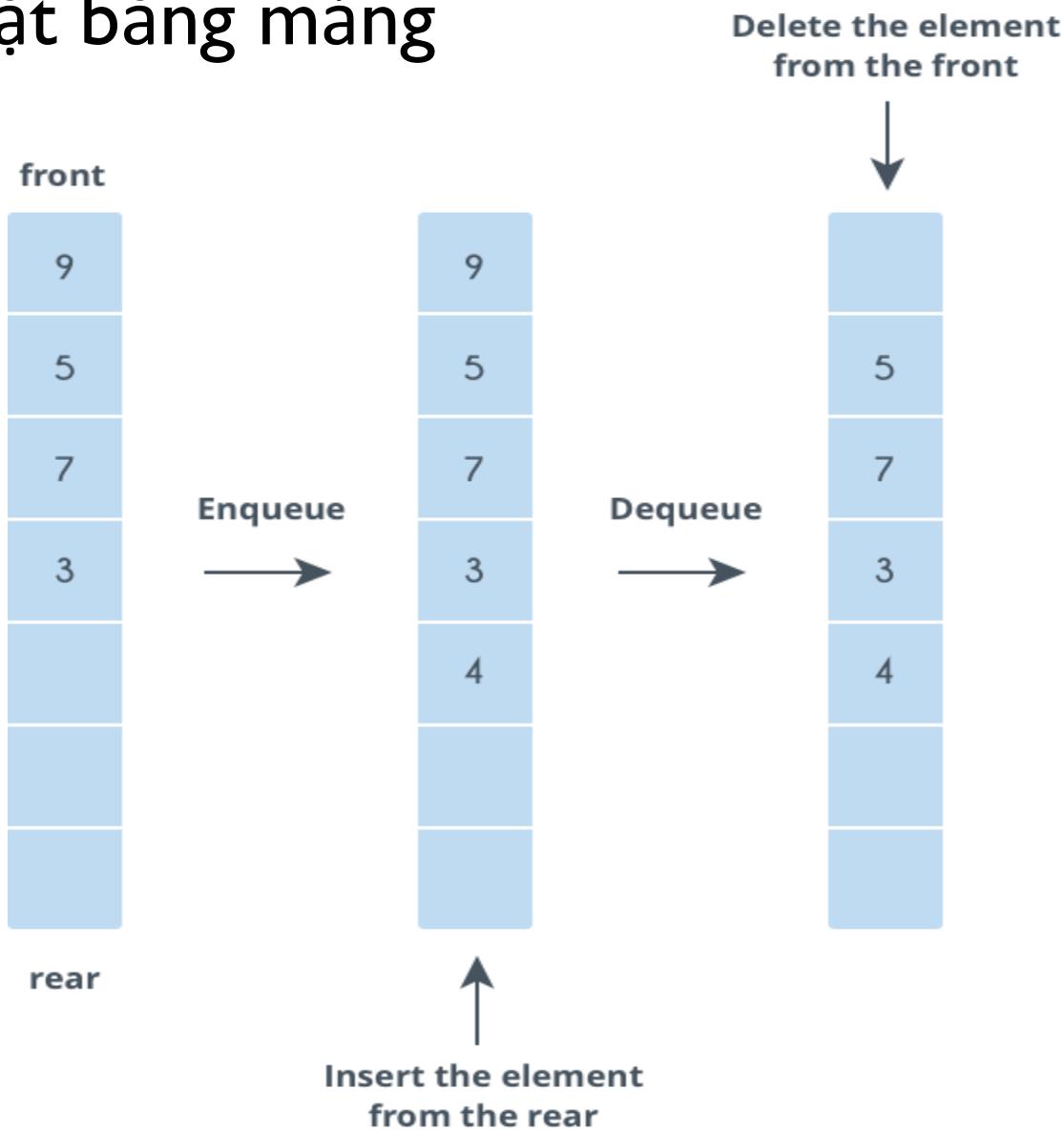
- Khái niệm

- Lưu trữ các đối tượng theo cơ chế và trước ra trước (FIFO)



Cấu trúc dữ liệu hàng đợi (queue)

- Cài đặt bằng mảng



Cấu trúc dữ liệu hàng đợi (queue)

- Thêm phần tử

```
if(rear == arraySize)      // Queue is full
    cout<<"OverFlow"<<endl;
else {
    queue[rear] = element; // Add the element to the back
    rear++;
}
```

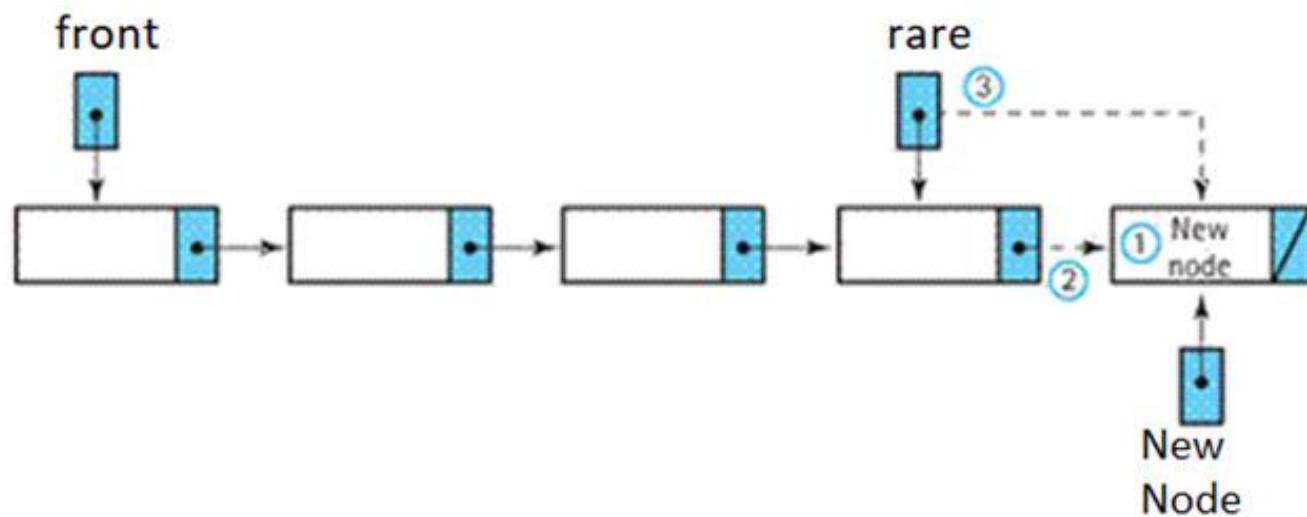
Cấu trúc dữ liệu hàng đợi (queue)

- Lấy ra một phần tử

```
if(front == rear)          // Queue is empty
    cout<<"UnderFlow"<<endl;
else {
    queue[front] = 0;      // Delete the front element
    front++;
}
```

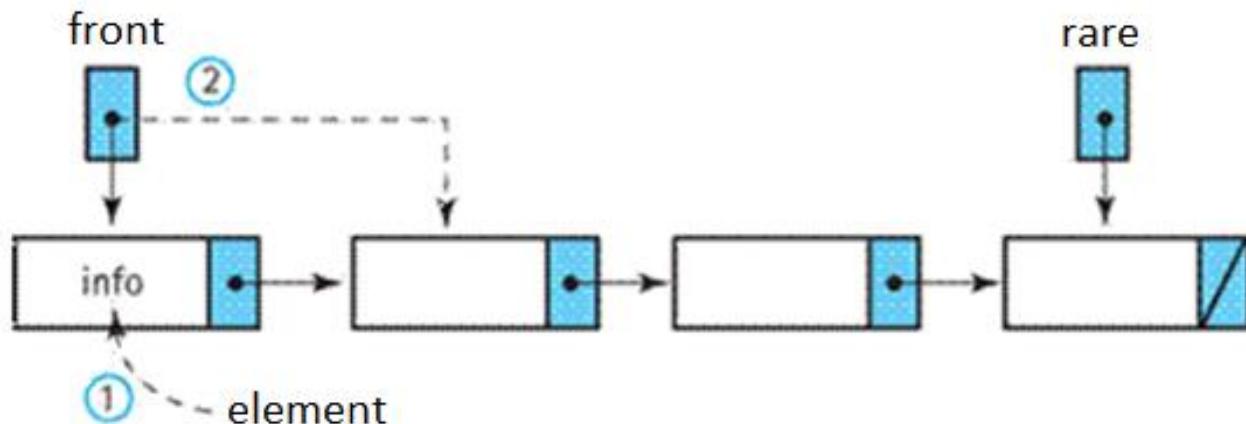
Cấu trúc dữ liệu hàng đợi (queue)

- Cài đặt bằng danh sách liên kết đơn
 - Thêm phần tử



Cấu trúc dữ liệu hàng đợi (queue)

- Cài đặt bằng danh sách liên kết đơn
 - Lấy phần tử khỏi danh sách



Cấu trúc dữ liệu hàng đợi (queue)

- Sử dụng lớp queue thư viện STL
 - Khai báo: queue <type> q_data
 - Hàm thêm và xóa phần tử push() & pop()
 - Hàm lấy kích thước size()
 - Hàm lấy giá trị phần tử front() & back()

Cấu trúc dữ liệu hàng đợi (queue)

- **Ứng dụng**

- Phục vụ các yêu cầu trên một tài nguyên được chia sẻ, như máy in, bộ nhớ đệm, ...
- Lập lịch tác vụ CPU
- Tìm kiếm theo chiều rộng trên đồ thị
- Thuật toán lập lịch công việc
- Quản lý tài nguyên được chia sẻ giữa các tiến trình (xử lý song song, đa tác vụ)

Cấu trúc dữ liệu hàng đợi (queue)

- Bài tập
 - Bài 1: Có một chuỗi ký tự. Hãy lấy ký tự ở đầu của chuỗi và thêm nó vào cuối chuỗi. Hãy cho thấy sự thay đổi của chuỗi sau khi thực hiện hành động trên N lần.

Cấu trúc dữ liệu hàng đợi (queue)

- Bài tập
 - Bài 2: Một chuỗi được gọi là có tính chất Palindrome nếu nó có tính chất đối xứng, tức là viết xuôi cũng giống viết ngược, ví dụ như "aaAaa". Nhập một chuỗi ký tự và kiểm tra tính chất này của chuỗi.

Cấu trúc dữ liệu hàng đợi hai đầu

- Khái niệm
 - Dữ liệu có thể thêm hoặc xóa ở cả đầu (front) và cuối (rear) của hàng đợi → sự kết hợp của cả stack và queue

Cấu trúc dữ liệu hàng đợi hai đầu

- Cài đặt
 - Cài đặt bằng mảng
 - Cài đặt bằng danh sách móc nối kép

Cấu trúc dữ liệu hàng đợi hai đầu

- Sử dụng lớp deque trong STL
 - Khai báo: `deque<type> dq_data;`
 - Các phương thức:
 - `push_front(const T& x)`
 - `push_back(const T& x)`
 - `pop_front()`
 - `pop_back()`
 - `T& front()`
 - `T& back()`
 - `size_type size()`

Cấu trúc dữ liệu hàng đợi hai đầu

- **Ứng dụng**
 - Giải các bài toán trong đó các phần tử cần phải được thêm và loại bỏ ở hai đầu
 - Lịch sử duyệt Web: Các trang Web được viếng thăm gần nhất được thêm vào một đầu và đầu kia loại bỏ các trang Web được viếng thăm cũ nhất (khi vượt quá kích thước của hàng đợi)
 - Lưu danh sách các thao tác Undo của phần mềm

Cấu trúc dữ liệu hàng đợi hai đầu

• Bài tập

- Bài 1: Cho một mảng số nguyên và một số nguyên k. Tìm số lớn nhất trong từng mảng con liên tiếp kích thước k.

Ví dụ, với mảng: $\text{arr}[] = \{1, 2, 3, 1, 4, 5, 2, 3, 6\}$, $k = 3$

Kết quả là: 3 3 4 5 5 5 6, vì

Số lớn nhất của {1, 2, 3} is 3

Số lớn nhất của {2, 3, 1} is 3

Số lớn nhất của {3, 1, 4} is 4

...

Cấu trúc dữ liệu hàng đợi hai đầu

• Bài tập

- Bài 2: Cho một mảng số nguyên và một số nguyên k. Tính tổng các số nhỏ nhất và lớn nhất trong từng mảng con liên tiếp kích thước k.

Ví dụ, với mảng: $\text{arr}[] = \{2, 5, -1, 7, -3, -1, -2\}$, $k = 4$

Kết quả là: 18, vì

$$\{2, 5, -1, 7\}, \min + \max = -1 + 7 = 6$$

$$\{5, -1, 7, -3\}, \min + \max = -3 + 7 = 4$$

$$\{-1, 7, -3, -1\}, \min + \max = -3 + 7 = 4$$

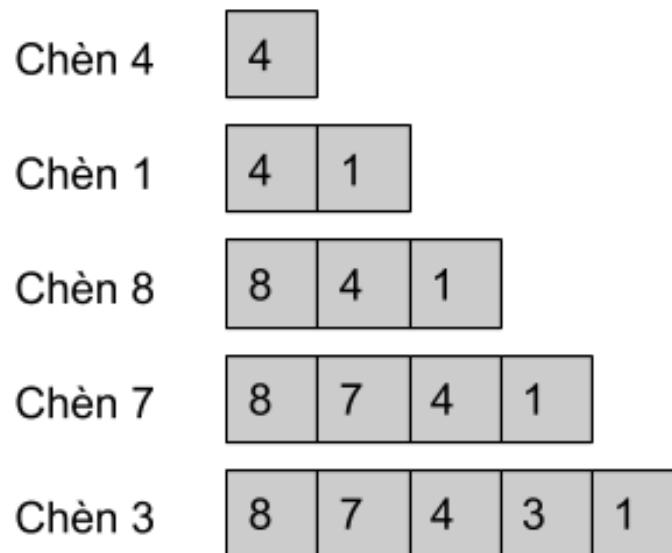
$$\{7, -3, -1, -2\}, \min + \max = -3 + 7 = 4$$

$$\text{Tổng tất cả min \& max} = 6 + 4 + 4 + 4 = 18$$

Cấu trúc dữ liệu hàng đợi ưu tiên

• Khái niệm

- Giống hàng đợi thông thường
- Thứ tự các phần tử phụ thuộc vào độ ưu tiên của phần tử đó
- Ưu tiên nhất → đầu dãy
- Ví dụ: {4, 1, 8, 7, 3} với số càng lớn độ ưu tiên càng cao



Cấu trúc dữ liệu hàng đợi ưu tiên

- Cài đặt
 - Sử dụng mảng thông thường
 - Thao tác chèn phải kiểm tra độ ưu tiên $\rightarrow O(n)$
 - Theo tac xóa phải dồn mảng $\rightarrow O(n)$
 - Sử dụng mảng với cấu trúc đống (heap)
 - Thao tác chèn, xóa $\rightarrow O(\log(n))$
 - Sử dụng danh sách liên kết
 - Thao tác chèn $\rightarrow O(n)$
 - Sử dụng thư viện C++ STL
 - Khai báo: priority_queue<type> pr_data;

Cấu trúc dữ liệu hàng đợi ưu tiên

- **Ứng dụng**

- Cài đặt cấu trúc đống (heap)
- Thuật toán cây bao trùm tối thiểu Prim
- Thuật toán tìm đường đi ngắn nhất Dijkstra
- Sinh Huffman Code
- Cân bằng tải, xử lý ngắt trong hệ điều hành
- Đảm bảo chất lượng dịch vụ (mạng)
- Dùng để cài đặt thuật toán khác

Cấu trúc dữ liệu hàng đợi ưu tiên

- **Bài tập**
 - **Bài 1:** Một hiệu trưởng muốn đi thăm các ký túc xá (có tất cả n ký túc). Ông ta rất bận nên chỉ thăm được k ($k \leq n$) nơi gần nhất. Các ký túc xá được tọa lạc trên một mặt phẳng hai chiều. Hãy giúp ông ấy, biết rằng bạn có tọa độ của n ký túc và vị trí hiện tại của ông ấy có tọa độ là $O(0,0)$.

Chú ý: khoảng cách giữa điểm $P(x,y)$ tới gốc $O(0,0)$ là khoảng cách Euclidean.

Ví dụ:

Input: $\{(1, 0), (2, 1), (3, 6), (-5, 2), (1, -4)\}$, $k = 3$

Output: $\{(1, 0), (2, 1), (1, -4)\}$

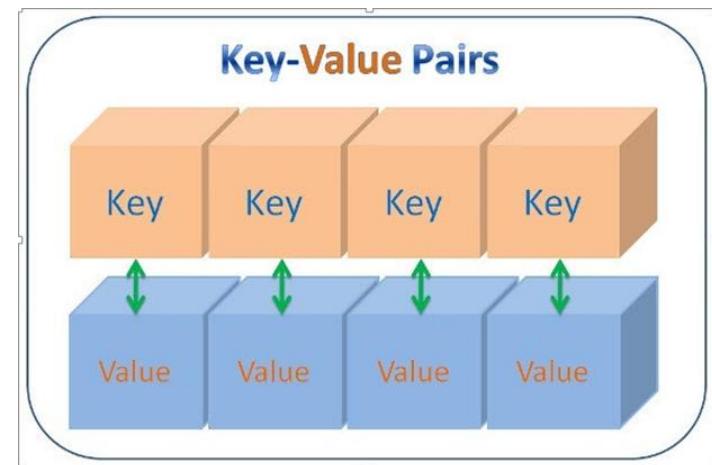
Cấu trúc dữ liệu hàng đợi ưu tiên

- Bài tập
 - Bài 2: Monk hiện có một mảng các số nguyên A. Với mỗi chỉ số i, anh ta muốn tìm tích của các số lớn nhất, số lớn thứ hai và số lớn thứ ba trong khoảng $[1, i]$.
Chú ý: các số có thể giống nhau nhưng chúng phải khác nhau về chỉ số.
 - **Gợi ý**: dùng hàng đợi ưu tiên để lưu các số từ 1 đến i rồi lấy ra để tính toán

Cấu trúc dữ liệu ánh xạ (map)

• Khái niệm

- Dữ liệu có kiểu cấu trúc liên kết
- Là sự kết hợp của khóa (key value) và ánh xạ của nó (mapped value)
- Giá trị các khóa
 - Duy nhất (không trùng)
 - Map
 - Được sắp thứ tự tăng dần
 - Unordered_map
 - Không được sắp
- Chú ý: trong Python, C# thì map là dictionary



Cấu trúc dữ liệu ánh xạ (map)

- Cài đặt
 - Trong C++
 - map
 - Dựa trên cây đỏ đen
 - Khai báo: map<type 1, type 2> tien_bien;
 - Lấy giá trị khóa (key): tien_bien.first
 - Lấy giá trị (value): ten_bien.second
 - unordered_map
 - Dựa trên bảng băm (hash table)
 - Khai báo: unordered_map<type 1, type 2> map_name;

Cấu trúc dữ liệu ánh xạ (map)

- Cài đặt
 - Thay đổi thứ tự sắp xếp các phần tử của map

```
struct cmp {  
    bool operator() (char a, char b) {  
        return a > b;  
    }  
};  
.....  
map <char, int, cmp> m;
```

Cấu trúc dữ liệu ánh xạ (map)

- Cài đặt
 - Các hàm cơ bản
 - `value = at(k)`: trả lại giá trị ứng với khóa k
 - `size()`: số phần tử hiện tại
 - `max_size()`: số phần tử mà map có thể chứa
 - `empty()`: kiểm tra map có rỗng không
 - `erase(const g)`: xóa khóa g khỏi map
 - `pair insert(keyvalue, mapvalue)`
 - `clear()`: xóa tất cả các phần tử

Cấu trúc dữ liệu ánh xạ (map)

- Cài đặt
 - Ví dụ
 - Chèn dữ liệu
 - map_name.insert(pair<int, int>(160, 42));
 - Xóa
 - map_name.erase(160);
 - Duyệt
 - map<int, int>::iterator itr;
 - for (itr = map_name.begin(); itr != map_name.end(); ++itr) {
 cout << itr->first << " \t" << itr->second << endl;
 - }

Cấu trúc dữ liệu ánh xạ (map)

- **Ứng dụng**
 - Tìm chủ thuê bao theo số điện thoại
 - Tìm địa chỉ IP theo tên miền
 - Tìm số lần vi phạm kỷ luật của một nhân viên

Cấu trúc dữ liệu ánh xạ (map)

- Bài tập:
 - Bài 1: Cho một chuỗi s, hãy đưa ra một dãy lần lượt là các ký tự và số lần xuất hiện của nó và các ký tự được sắp xếp theo thứ tự từ điển.

Cấu trúc dữ liệu ánh xạ (map)

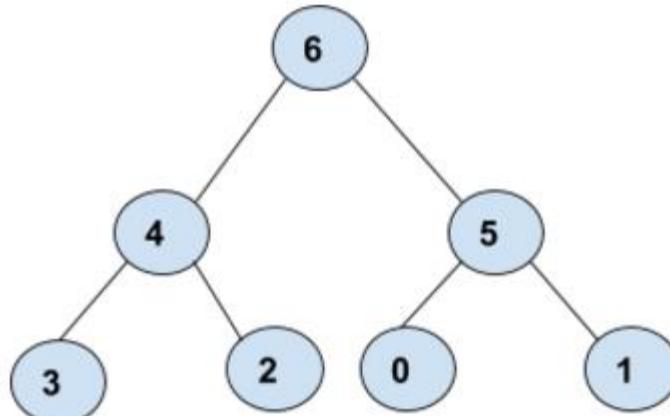
- Bài tập:
 - Bài 2: Cho danh sách các sản phẩm của 2 kho hàng A và B. Do chiến lược kinh doanh bạn được giao nhiệm vụ nhập các sản phẩm từ kho B vào kho A sao cho những sản phẩm nào đã có trong kho A thì không nhập.
 - Ví dụ: Với A = {"Banana", "Banana", "Apple"}, B = {"Orange", "Apple", "Banana", "Watermelon"}, thì mergeProducts(A, B) = {Apple → false, Banana → false, Orange → true, Watermelon → true}

Cấu trúc dữ liệu đống (heap)

- Khái niệm
 - Cấu trúc dữ liệu dạng cây
 - Tất cả các nút trong cây được sắp xếp theo một thứ tự nhất định, có thể là theo chiều tăng dần hoặc giảm dần

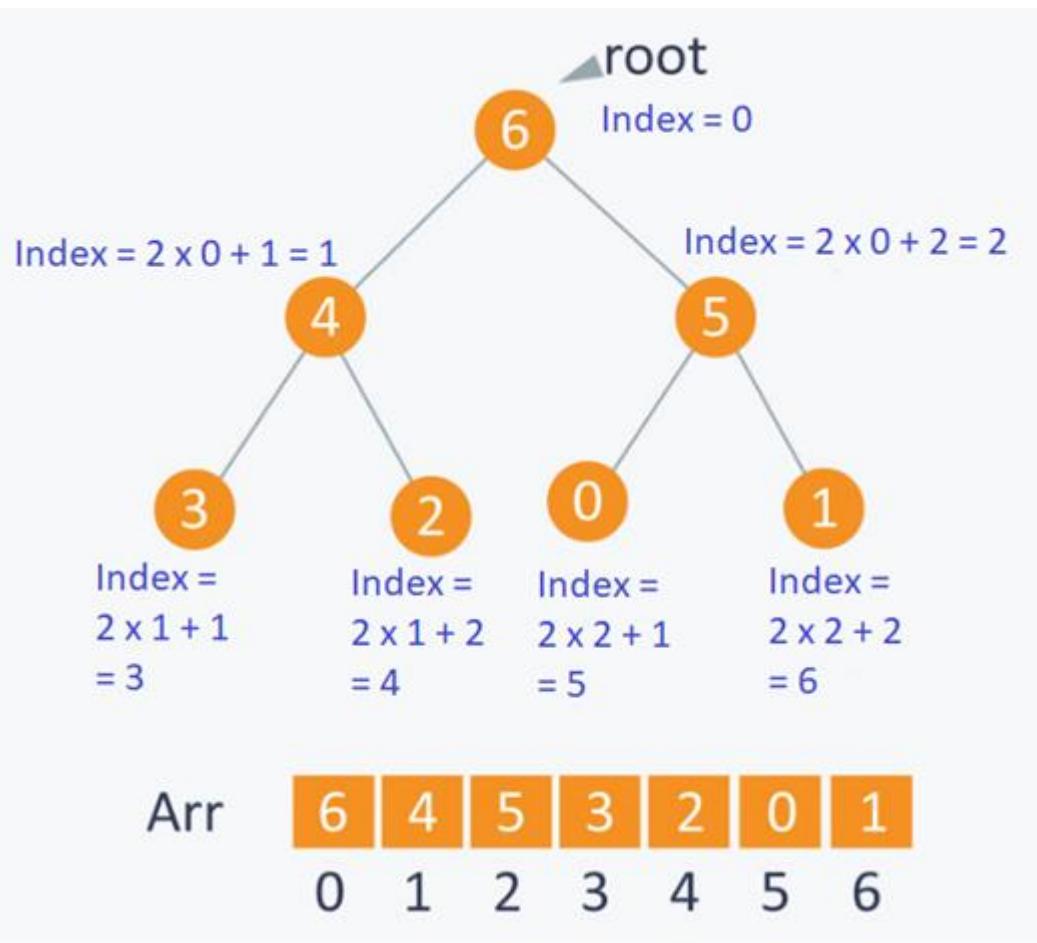
Cấu trúc dữ liệu đống (heap)

- Khái niệm
 - Max heap:
 - Nút A là cha của nút B và giá trị của A lớn hơn B
 - Áp dụng luật trên cho toàn bộ cây
 - Binary Heap được ứng dụng phổ biến
 - Một heap có 7 nút với giá trị: {6, 4, 5, 3, 2, 0, 1}



Cấu trúc dữ liệu đống (heap)

- Cài đặt: chuyển mảng thành max heap



Nút gốc có vị trí 0

Nút có vị trí thứ i thì:

- Nút cha ở $(i - 1) / 2$
- Nút con trái có vị trí: $2 * i + 1$
- Nút con phải có vị trí: $2 * i + 2$

Cấu trúc dữ liệu đống (heap)

- Cài đặt: chuyển mảng thành max heap
 - Bước 1: Lấy một phần tử từ mảng
 - Bước 2: Kiểm tra xem con bên trái của nó có phải max heap không
 - Bước 3: Kiểm tra xem con bên phải của nó có phải max heap không
 - Bước 4: Kiểm tra xem chính node đó có phải max heap không

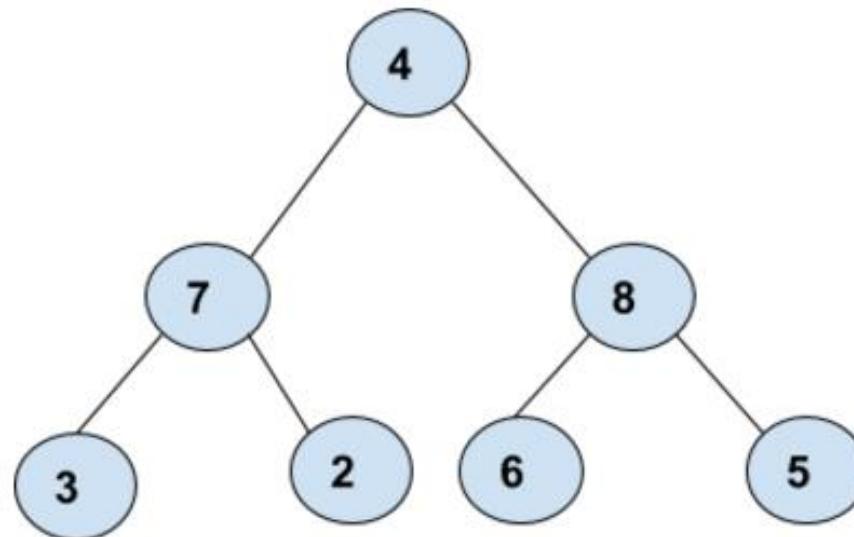
Cấu trúc dữ liệu đống (heap)

- Cài đặt: chuyển mảng thành max heap

```
void heapify (int A[], int i) {  
    int largest;                                //Lưu chỉ số của phần tử lớn nhất  
    int left = 2*i + 1;                          // Vị trí của con bên trái  
    int right = 2*i +2;                          // Vị trí của con bên phải  
    largest = i;  
    if (left< N and A[left] > A[i])           // N là số phần tử trong mảng  
        largest = left;  
    if (right < N and A[right] > A[largest])  
        largest = right;  
    if (largest != i ) {  
        swap (A[i] , A[largest]);  
        heapify (A, largest);  
    }  
}
```

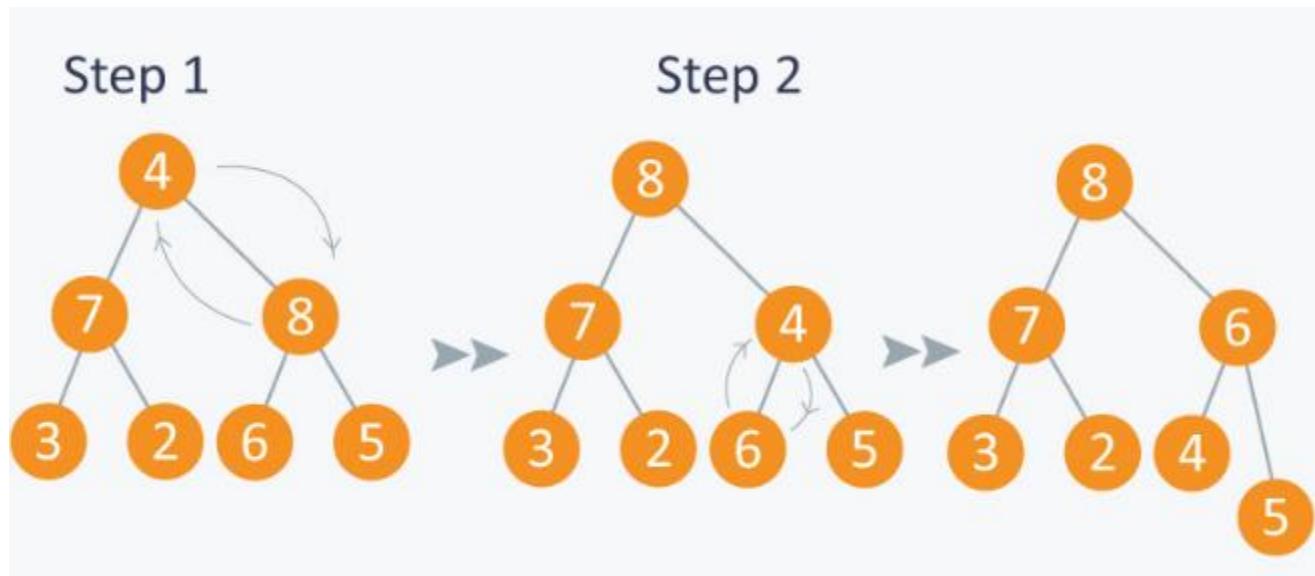
Cấu trúc dữ liệu đống (heap)

- Cài đặt: chuyển mảng thành max heap
 - Cây nhị phân chưa phải là max heap



Cấu trúc dữ liệu đống (heap)

- Cài đặt: chuyển mảng thành max heap
 - Mô phỏng hoạt động của hàm heapify



Bước 1: 8 lớn hơn 4, nên 8 được đổi chỗ với 4

Bước 2: vì 6 lớn hơn 4 nên 4 được đổi chỗ với 6

→ phải thực hiện lại hàm heapify tại nút có giá trị 4

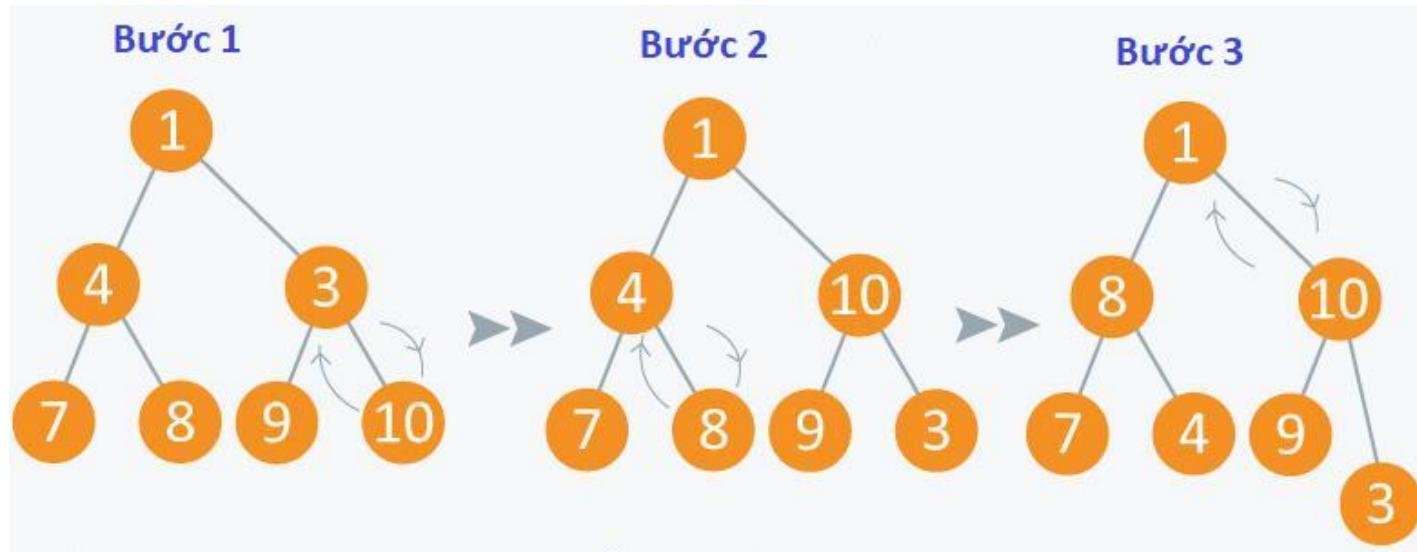
Cấu trúc dữ liệu đống (heap)

- Cài đặt: chuyển mảng thành max heap
 - Hàm heapify mới áp dụng cho một nút bất kỳ.
 - Cần thực hiện cho toàn bộ heap
 - Không cần áp dụng cho nút lá vì chúng không có con
 - Chỉ cần áp dụng đối với các nút trong
 - có chỉ số từ 0 đến $N / 2 - 1$

```
void run_heapify(int A[]) {  
    for(int i = N / 2 - 1 ; i >= 0 ; i-- ) {  
        heapify (A, i) ;  
    }  
}
```

Cấu trúc dữ liệu đống (heap)

- Cài đặt: chuyển mảng thành max heap



$N = 7 \rightarrow \text{heapify}(N/2-1=2)$

→ Bắt đầu từ $A[2]$ là nút có giá trị 3

→ Giá trị của con trái 9, con phải 10

| | | | | | | | |
|---|---|---|---|---|---|---|----|
| A | 1 | 4 | 3 | 7 | 8 | 9 | 10 |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

Bước 1: vì $10 > 3$ nên 3 và 10 đổi vị trí $\rightarrow A[2] = 10, A[6] = 3$

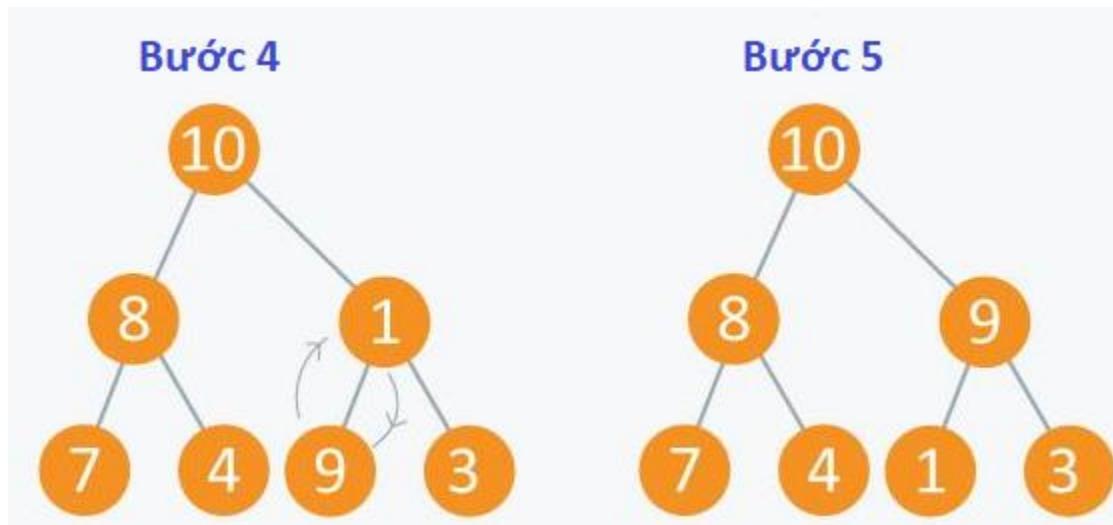
→ gọi $\text{heapify}(A, \text{largest} = 7)$, do 3 là lá nên không thay đổi gì

Bước 2: gọi $\text{heapify}(A, 1) \rightarrow 4$ đổi chỗ cho 8, $\text{heapify}(A, 4) \rightarrow$ không đổi

Bước 3: gọi $\text{heapify}(A, 0) \rightarrow 1$ và 10 đổi vị trí

Cấu trúc dữ liệu đống (heap)

- Cài đặt: chuyển mảng thành max heap



Bước 4: Lúc này chỉ số của 1 là 2 → gọi đệ quy heapify(A, 2) thì
1 đổi chỗ cho 9 → lúc này 1 là lá nên không đệ quy tiếp

Bước 5: Các phần tử đã được sắp → trả về một max heap

Cấu trúc dữ liệu đống (heap)

- Cài đặt: sử dụng thư viện STL trong C++
 - Hàm make_heap(): chuyển cấu trúc dạng range sang heap
 - Ví dụ:

```
vector<int> v1 = {20, 30, 40, 25, 15};  
make_heap(v1.begin(), v1.end());  
cout << "The maximum element of heap is : ";  
cout << v1.front() << endl;
```

- Hàm push_heap(), pop_heap(), is_heap()

Cấu trúc dữ liệu đống (heap)

- **Ứng dụng**

- Sắp xếp vung đống (heap sort)
- Hàng đợi ưu tiên (priority queue)
- Các thuật toán trên đồ thị
 - Tìm đường đi ngắn nhất Dijkstra
 - Cây bao trùm tối thiểu Prim

Cấu trúc dữ liệu đống (heap)

- Ứng dụng
 - Sắp xếp vun đống (heap sort)
 - Sắp xếp các phần tử trong mảng A[] theo thứ tự tăng dần
 - Sử dụng max heap với ý tưởng sau:
 - Tạo đống → Nút gốc có giá trị lớn nhất → lưu lại
 - Thay thế nó bằng nút có giá trị nhỏ hơn trong cây → gọi hàm heapify để tìm được giá trị lớn thứ hai
 - Cứ tiếp tục như vậy để tìm được giá trị lớn thứ ba, thứ tư, ...

Cấu trúc dữ liệu đống (heap)

- **Ứng dụng**
 - Sắp xếp vung đống (heap sort), gồm các bước
 - **Bước 1:** Tạo max heap các phần tử trong mảng A sử dụng hàm run_heapify
 - **Bước 2:** Có A[0] là phần tử lớn nhất → đổi chỗ A[0] cho phần tử cuối mảng (thường có giá trị nhỏ nhất)
 - **Bước 3:** Tạo max heap cho các phần tử trong mảng trừ phần tử cuối (là A[0] đã đúng vị trí)
→ Kích thước của heap giảm đi 1
 - **Bước 4:** Lặp lại bước 2 và bước 3 cho tới khi các phần tử trong mảng đã được sắp

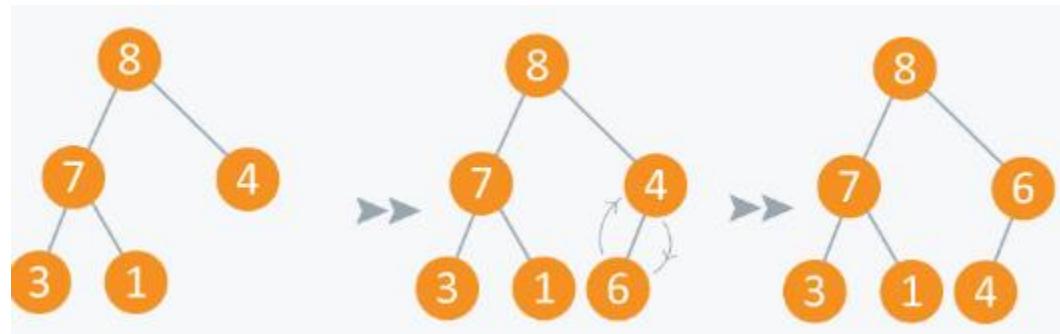
Độ phức tạp: $O(N \log N)$

Cấu trúc dữ liệu đống (heap)

- **Ứng dụng**
 - Cài đặt Hàng đợi ưu tiên (priority queue)
 - Giả sử dùng mảng A để lưu các phần tử
 - Số phần tử hiện có là N
 - Giá trị càng lớn thì độ ưu tiên càng cao
 - Sử dụng cấu trúc max heap

Cấu trúc dữ liệu đống (heap)

- Ứng dụng
 - Cài đặt Hàng đợi ưu tiên (priority queue)
 - Thao tác chèn phần tử
 - Có thể vi phạm quy tắc heap
 - Nếu vi phạm thì đổi chỗ nút đó với nút cha cho tới khi giá trị của nút cha lớn hơn



Chèn nút có giá trị 6 vào hàng đợi ưu tiên

Cấu trúc dữ liệu đống (heap)

- Ứng dụng
 - Cài đặt Hàng đợi ưu tiên (priority queue)
 - Thao tác chèn phần tử

```
void push (int A[], int x) {  
    int i = N;  
    N = N + 1; //Tăng số phần tử  
    A[i] = x; //Chèn vào vị trí cuối cùng của hàng đợi  
    while(i > 0 && A[i] > A[(i-1)/2]) { //A[i] lớn hơn giá trị của nút cha  
        swap(A[(i-1)/2 ], A[i]);  
        i = (i-1)/2; //Tiếp tục kiểm tra tại vị trí của nút cha  
    }  
}
```

Độ phức tạp thời gian: $O(\log_2 N)$

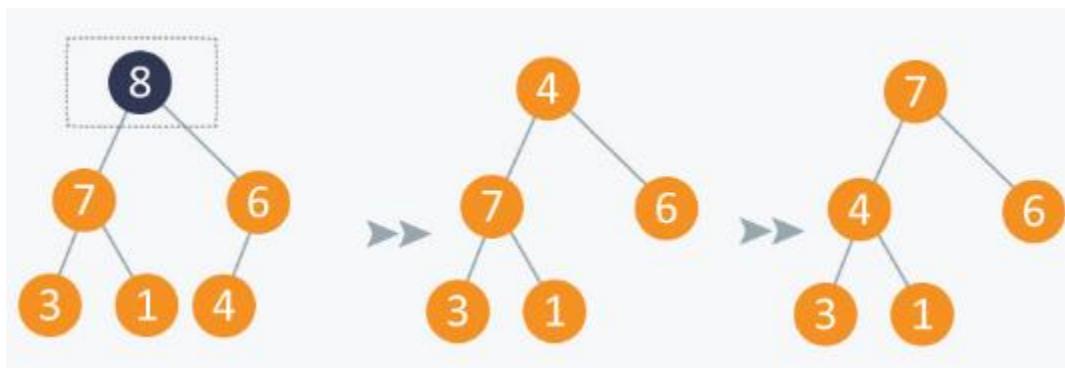
Cấu trúc dữ liệu đống (heap)

- **Ứng dụng**

- Cài đặt Hàng đợi ưu tiên (priority queue)

- Thao tác xóa một phần tử

Phần tử lớn nhất được lấy ra và thế chỗ bằng phần tử cuối mảng → quy tắc heap bị phá vỡ → gọi heapify(0)



Xóa phần tử lớn nhất khỏi hàng đợi ưu tiên

Cấu trúc dữ liệu đống (heap)

- Ứng dụng
 - Cài đặt Hàng đợi ưu tiên (priority queue)
 - Thao tác xóa một phần tử

```
void pop() {  
    if (n == 0) cout<< "Hang doi rong";  
    swap(A[n-1], A[0]);  
    n--;  
    heapify(A, 0);  
}
```

Độ phức tạp thời gian: $O(\log_2 N)$

Cấu trúc dữ liệu đống (heap)

- Ứng dụng
 - Cài đặt Hàng đợi ưu tiên (priority queue)
 - Lấy giá trị phần tử có giá trị lớn nhất

```
void top() {  
    return A[0];  
}
```

Độ phức tạp thời gian: O(1)

Cấu trúc dữ liệu đống (heap)

- **Bài tập**
 - **Bài 1:** Cài đặt cây tìm kiếm nhị phân ngẫu nhiên - Treap.
 - Treap là một cấu trúc kết hợp giữa cây tìm kiếm nhị phân và heap
 - Là một cây nhị phân có 2 khóa, trong đó một khóa thỏa mãn tính chất của heap, còn một khóa thỏa mãn tính chất của cây tìm kiếm
 - Khóa heap có vai trò giữ cho cây không quá cao

Cấu trúc dữ liệu đống (heap)

- Bài tập
 - **Bài 2:** Viết một thuật toán hiệu quả để in ra k phần tử lớn nhất trong mảng.
Ví dụ, với mảng {1, 23, 12, 9, 30, 2, 50}, in ra 3 phần tử lớn nhất ($k = 3$) là 50, 30 và 23.

Cấu trúc dữ liệu cây nâng cao

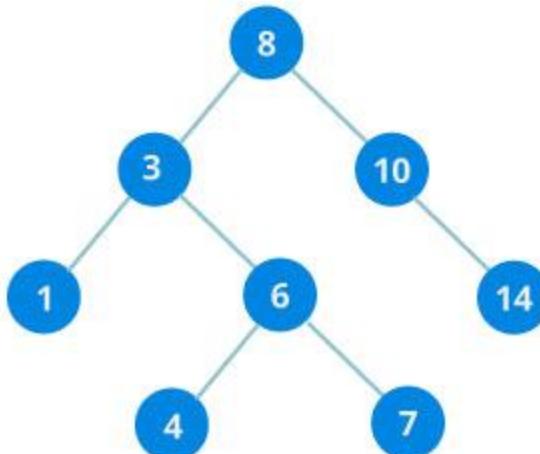
- Một số cấu trúc dữ liệu cây
 - Cây tìm kiếm nhị phân
 - Cây tự cân bằng (AVL tree)
 - Cây đỏ đen
 - Cây 2-3-4
 - Cây quyết định

Cấu trúc dữ liệu cây nâng cao

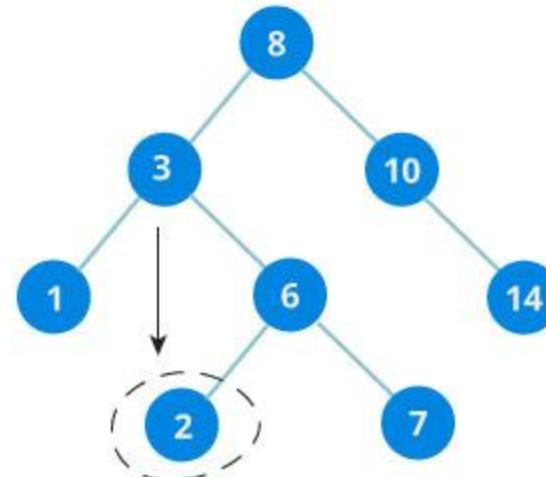
- Cây tìm kiếm nhị phân
 - Là cây có các ràng buộc sau
 - Giá trị của tất cả nút ở cây con bên trái phải nhỏ hơn hoặc bằng giá trị của nút gốc
 - Giá trị của tất cả nút ở cây con bên phải phải lớn hơn giá trị của nút gốc
 - Tất cả các cây con (trái và phải) đều phải đảm bảo 2 tính chất trên

Cấu trúc dữ liệu cây nâng cao

- Cây tìm kiếm nhị phân



Cây tìm kiếm nhị phân



Không là cây tìm kiếm nhị phân

Cấu trúc dữ liệu cây nâng cao

- Cây tìm kiếm nhị phân
 - Ý nghĩa
 - Được gọi là cây nhị phân vì mỗi nút của cây chỉ có tối đa hai con
 - Một cấu trúc dữ liệu hiệu quả để xây dựng một danh sách mà dữ liệu trên đó được sắp xếp
 - Được gọi là cây tìm kiếm nhị phân vì có thể được sử dụng để tìm kiếm một phần tử trong thời gian $O(\log_2(n))$

Cấu trúc dữ liệu cây nâng cao

- Cây tìm kiếm nhị phân
 - Thêm phần tử
 - Vị trí của các nút được thêm vào sẽ luôn nút lá
 - Nếu nút hiện tại là NULL thì đó là vị trí cần thêm
→ thêm và kết thúc
 - Nếu giá trị cần thêm nhỏ hơn hoặc bằng giá trị của gốc hiện tại, gọi đệ quy chèn vào cây con bên trái
 - Nếu giá trị cần thêm lớn hơn giá trị của gốc hiện tại, gọi đệ quy chèn vào cây con bên phải

Cấu trúc dữ liệu cây nâng cao

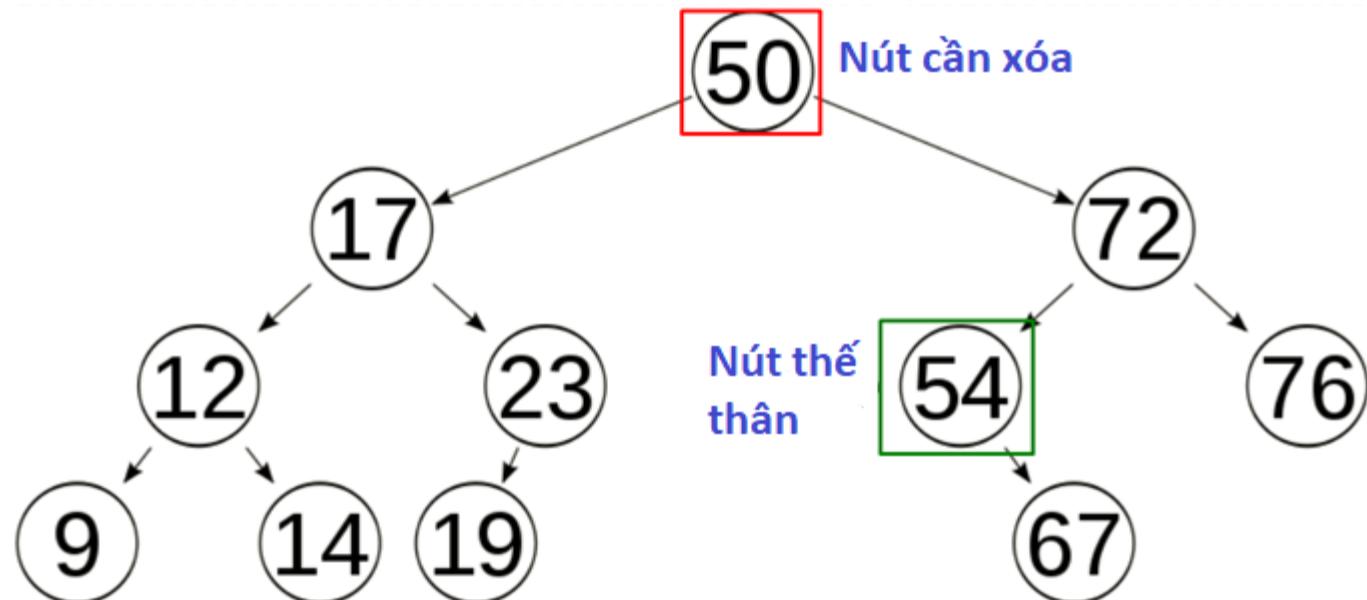
- Cây tìm kiếm nhị phân
 - Tìm phần tử
 - Nếu giá trị cần tìm bằng giá trị của nút hiện tại, trả về true và kết thúc
 - Nếu giá trị cần tìm nhỏ hơn giá trị của nút hiện tại, gọi đệ quy tìm ở cây con bên trái
 - Nếu giá trị cần tìm lớn hơn giá trị của nút hiện tại, gọi đệ quy tìm ở cây con bên phải
 - Nếu tìm đến hết cây mà không thấy, trả về false và kết thúc

Cấu trúc dữ liệu cây nâng cao

- Cây tìm kiếm nhị phân
 - Xóa phần tử
 - Trường hợp 1: nút cần xóa là lá → xóa khỏi cây
 - Trường hợp 2: nút cần xóa có một con
 - Giải phóng nút bị xóa
 - Liên kết trực tiếp cây con duy nhất của nó với cha của nút bị xóa.
 - Trường hợp 3: nút cần xóa có đủ hai con
 - Tìm nút con trái nhất của cây con bên phải của nút cần xóa (Left most)
 - Cập nhật giá trị của nút cần xóa bằng giá trị của nút trái nhất
 - Gọi đệ quy xóa nút Left most khỏi cây (rơi vào trường hợp 1 hoặc trường hợp 2)

Cấu trúc dữ liệu cây nâng cao

- Cây tìm kiếm nhị phân
 - Xóa phần tử
 - Trường hợp 3: nút cần xóa có đủ hai con



Cấu trúc dữ liệu cây nâng cao

- Cây tìm kiếm nhị phân
 - Duyệt cây → Thứ tự duyệt được đặt tên phụ thuộc vào vị trí của nút gốc trong quá trình duyệt
 - Theo thứ tự trước: Gốc → Trái → Phải
 - Theo thứ tự giữa: Trái → Gốc → Phải
 - Thứ tự sau: Trái → Phải → Gốc

Cấu trúc dữ liệu cây nâng cao

- Cây tìm kiếm nhị phân
 - Duyệt cây: Theo thứ tự trước
 - Ghé thăm nút gốc
 - Gọi đệ quy duyệt qua cây con bên trái
 - Gọi đệ quy duyệt qua cây con bên phải

```
void PreOrder(node_t* root) {  
    if(root != NULL) {  
        cout<<root->data;  
        PreOrder(root->left);  
        PreOrder(root->right);  
    }  
}
```

Cấu trúc dữ liệu cây nâng cao

- Cây tìm kiếm nhị phân
 - Duyệt cây: Theo thứ tự giữa
 - Gọi đệ quy duyệt qua cây con bên trái
 - Ghé thăm nút gốc
 - Gọi đệ quy duyệt qua cây con bên phải

```
void InOrder(node_t* root){  
    if (root != NULL) {  
        InOrder(root->left);  
        cout<<root->data;  
        InOrder(root->right);  
    }  
}
```

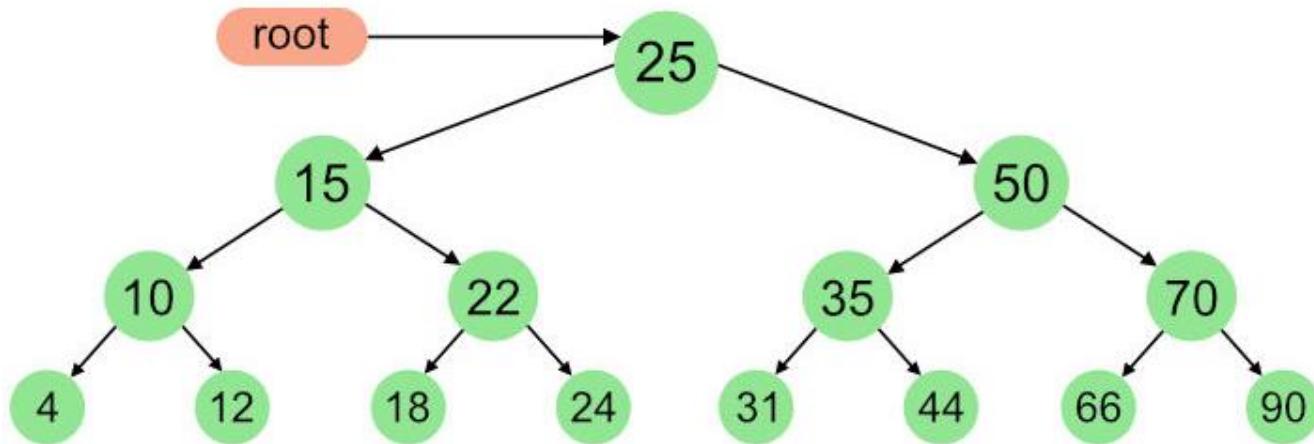
Cấu trúc dữ liệu cây nâng cao

- Cây tìm kiếm nhị phân
 - Duyệt cây: Theo thứ tự sau
 - Gọi đệ quy duyệt qua cây con bên trái
 - Gọi đệ quy duyệt qua cây con bên phải
 - Ghé thăm nút gốc

```
void PostOrder(node_t* root){  
    if(root != NULL) {  
        PostOrder(root->left);  
        PostOrder(root->right);  
        cout<<root->data;  
    }  
}
```

Cấu trúc dữ liệu cây nâng cao

- Cây tìm kiếm nhị phân



A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

InOrder(root) visits nodes in the following order:

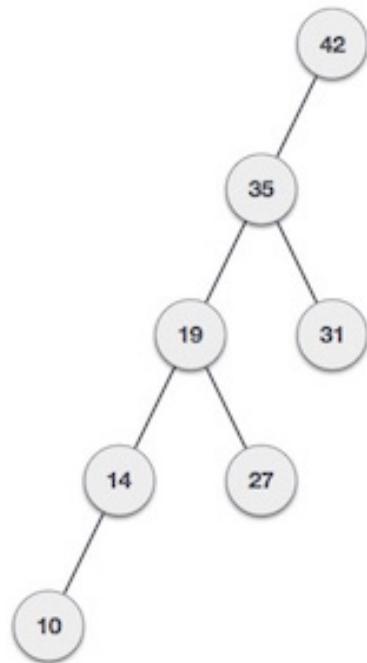
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Post-order traversal visits nodes in the following order:

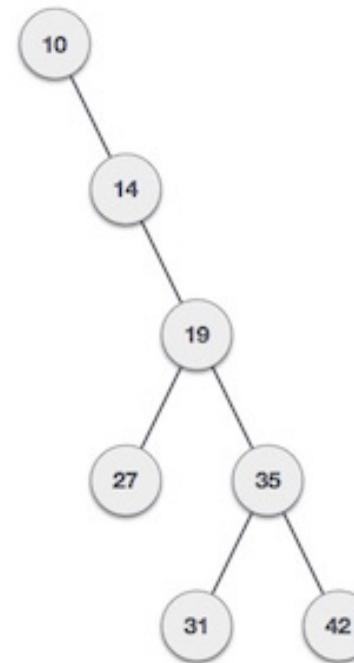
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25

Cấu trúc dữ liệu cây nâng cao

- Cây tìm kiếm nhị phân tự cân bằng AVL
 - Tại sao lại cần cây AVL?
 - Cây tìm kiếm nhị phân ở dạng đã được sắp thứ tự (tăng dần hoặc giảm dần) → độ phức tạp khi tìm kiếm $O(n)$



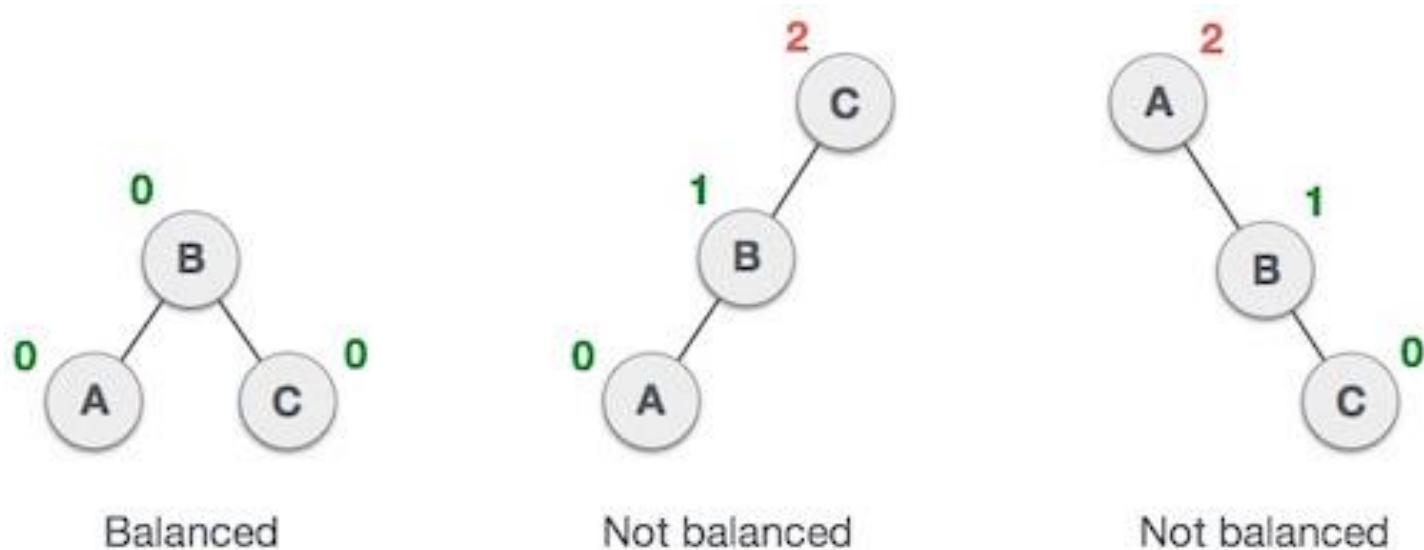
Giảm dần



Tăng dần

Cấu trúc dữ liệu cây nâng cao

- Cây tìm kiếm nhị phân tự cân bằng AVL
 - Đảm bảo hiệu số giữa độ cao của cây con bên trái và cây con bên phải không quá 1
→ Balance Factor (Nhân tố cân bằng)

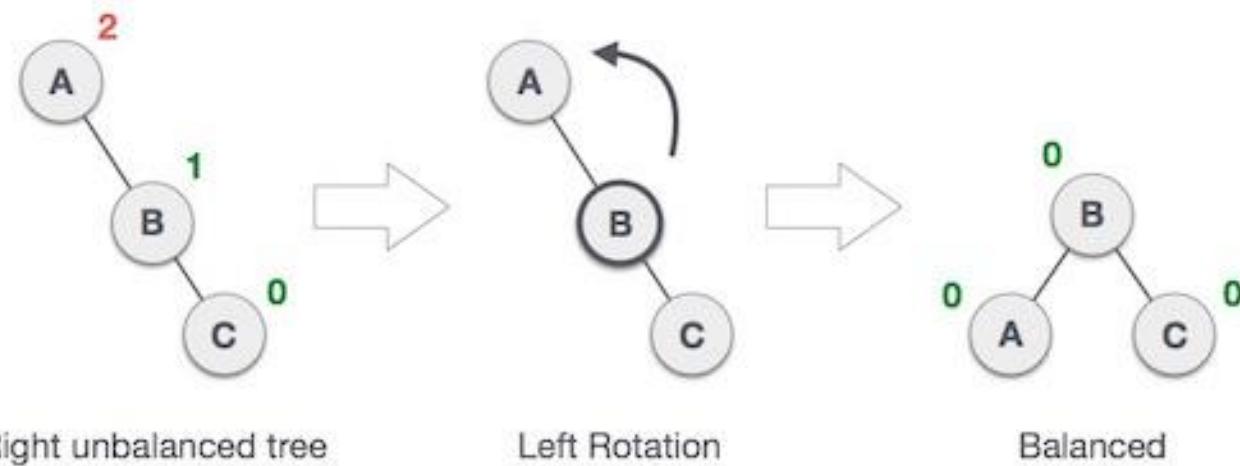


Cấu trúc dữ liệu cây nâng cao

- Cây tìm kiếm nhị phân tự cân bằng AVL
 - Kỹ thuật quay cây → 4 kỹ thuật
 - Kỹ thuật quay trái
 - Kỹ thuật quay phải
 - Kỹ thuật quay trái-phải
 - Kỹ thuật quay phải-trái
 - Hai kỹ thuật quay đầu → quay đơn
 - Hai kỹ thuật quay còn → quay ghép

Cấu trúc dữ liệu cây nâng cao

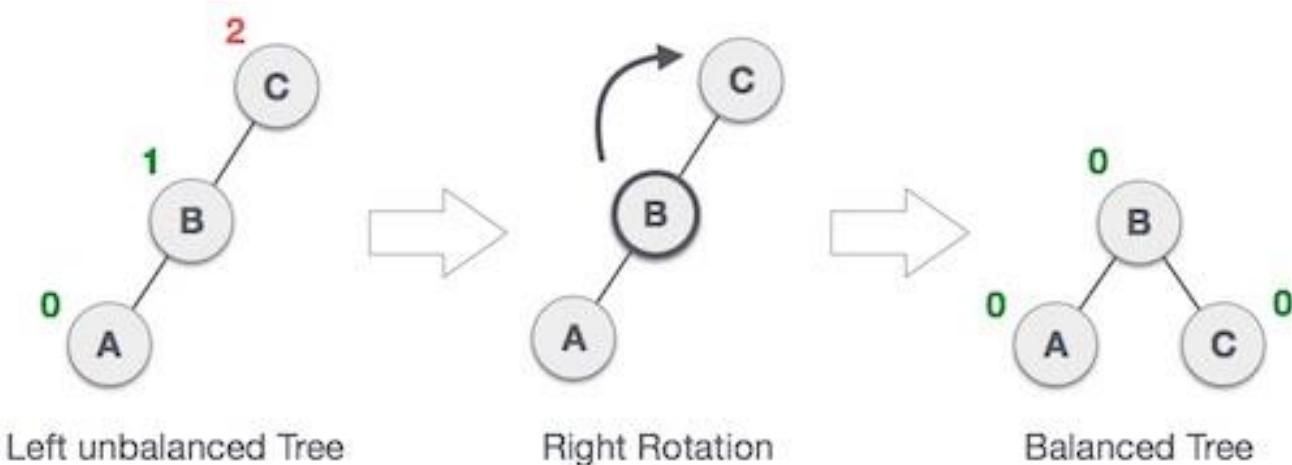
- Cây tìm kiếm nhị phân tự cân bằng AVL
 - Kỹ thuật quay trái
 - Áp dụng khi một nút được chèn vào trong cây con bên phải của cây con bên phải



- Nút **A** trở nên không cân bằng khi nút **C** được chèn vào

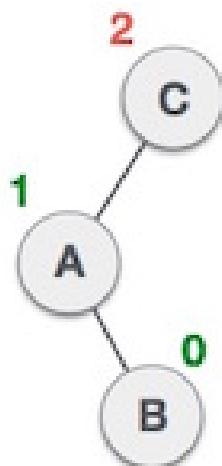
Cấu trúc dữ liệu cây nâng cao

- Cây tìm kiếm nhị phân tự cân bằng AVL
 - Kỹ thuật quay phải
 - Áp dụng khi một nút được chèn vào cây con bên trái của cây con bên trái



Cấu trúc dữ liệu cây nâng cao

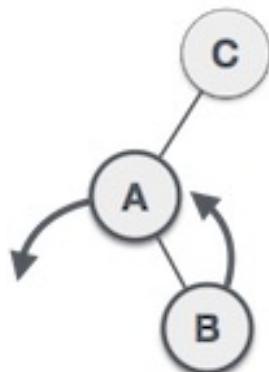
- Cây tìm kiếm nhị phân tự cân bằng AVL
 - Kỹ thuật quay trái-phải
 - Là sự kết hợp của kỹ thuật quay trái được theo sau bởi kỹ thuật quay phải
 - Áp dụng khi một nút đã được chèn vào trong cây con bên phải của cây con bên trái



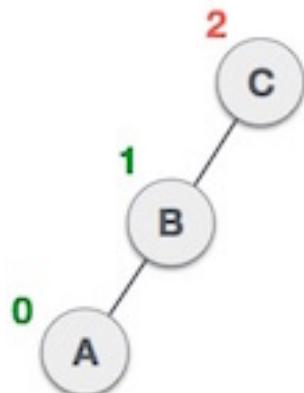
- Nút B được chèn vào bên phải nút A
→ nút C trở nên không cân bằng
- Cây AVL có thể thực hiện kỹ thuật quay trái-phải

Cấu trúc dữ liệu cây nâng cao

- Cây tìm kiếm nhị phân tự cân bằng AVL
 - Kỹ thuật quay trái-phải



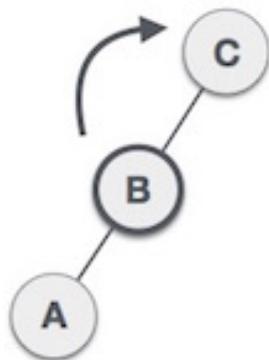
- Thực hiện phép quay trái trên cây con bên trái của C tại nút A
- A trở thành cây con bên trái của B
- B trở thành cây con trái của C



Nút C vẫn không cân bằng do xuất hiện cây con bên trái của cây con bên trái

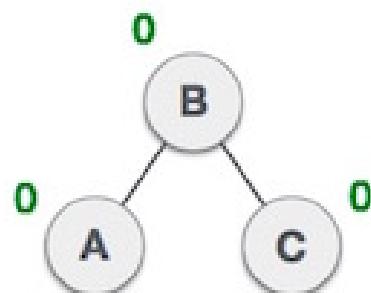
Cấu trúc dữ liệu cây nâng cao

- Cây tìm kiếm nhị phân tự cân bằng AVL
 - Kỹ thuật quay trái-phải



Thiện kỹ thuật quay phải để làm B trở thành nút gốc mới của cây này

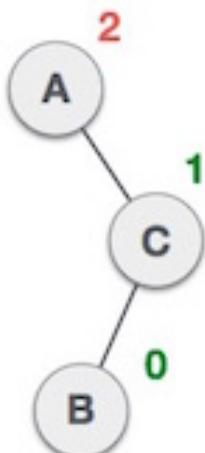
Nút C trở thành cây con bên phải của chính cây con bên trái của nó



Cây đã cân bằng

Cấu trúc dữ liệu cây nâng cao

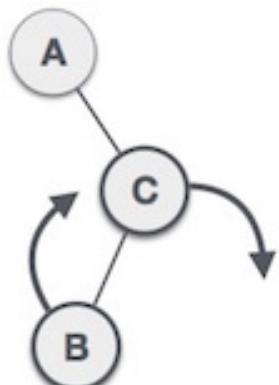
- Cây tìm kiếm nhị phân tự cân bằng AVL
 - Kỹ thuật quay phải-trái
 - Là sự kết hợp của kỹ thuật quay phải được thực hiện sau bởi kỹ thuật quay trái
 - Áp dụng khi một nút đã được chèn vào trong cây con bên trái của cây con bên phải



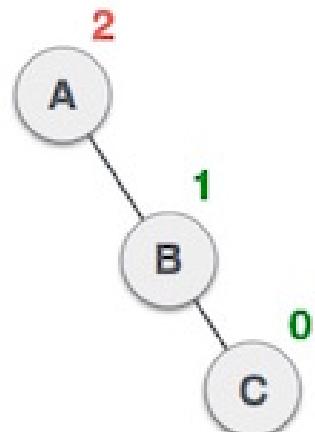
- Nút B được chèn vào bên trái nút C
→ nút A trở nên không cân bằng
- Cây AVL có thể thực hiện kỹ thuật quay phải-trái

Cấu trúc dữ liệu cây nâng cao

- Cây tìm kiếm nhị phân tự cân bằng AVL
 - Kỹ thuật quay phải-trái



- Thực hiện kỹ thuật quay phải với nút C
- C trở thành cây con bên phải của chính cây con bên trái B
- B trở thành cây con bên phải của C

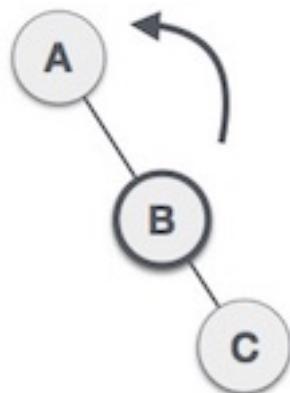


Nút A vẫn không cân bằng vì xuất hiện cây con bên phải của cây con bên phải của nó

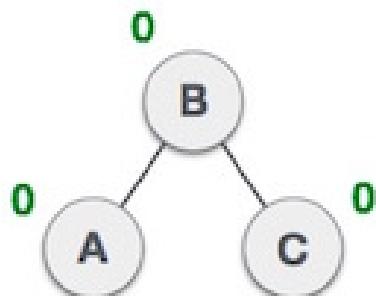
Cần phải thực hiện một kỹ thuật quay trái

Cấu trúc dữ liệu cây nâng cao

- Cây tìm kiếm nhị phân tự cân bằng AVL
 - Kỹ thuật quay phải-trái



- Thực hiện kỹ thuật quay trái làm cho B trở thành nút gốc mới của cây con
- Nút A trở thành cây con bên trái của cây con B bên phải của nó

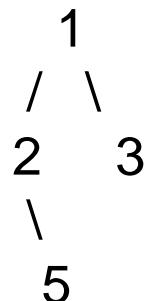


Cây đã cân bằng

Cấu trúc dữ liệu cây nâng cao

- Bài tập
 - **Bài 1:** Cho một cây nhị phân. Hãy liệt kê tất cả các đường đi từ gốc đến lá
 - Ví dụ:

Đầu vào:



Đầu ra: ["1->2->5", "1->3"]

Giải thích: Tất cả đường đi từ gốc đến lá: 1->2->5, 1->3

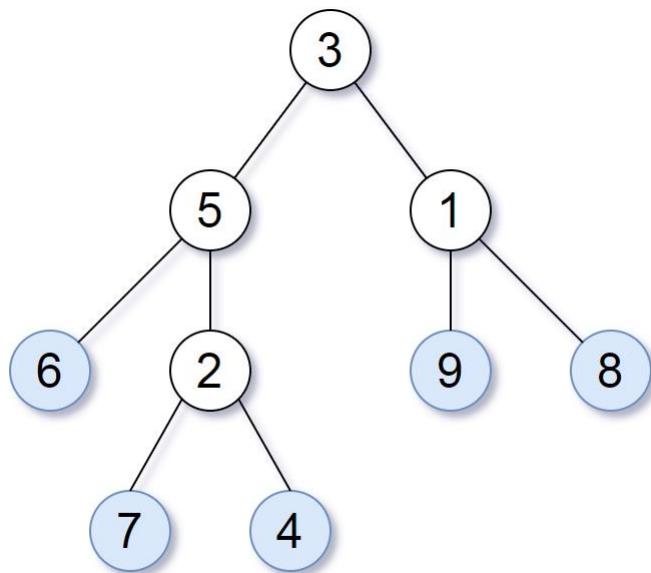
Cấu trúc dữ liệu cây nâng cao

- **Bài tập**
 - **Bài 2:** Xem xét tất cả các lá của một cây nhị phân. Theo thứ tự từ trái qua phải, giá trị của các lá lập thành một chuỗi giá trị lá

Ví dụ, với cây nhị phân ở hình bên, chuỗi giá trị lá là (6, 7, 4, 9, 8).

Hai cây nhị phân được xem là có lá tương đồng nếu chuỗi giá trị lá của chúng là như nhau.

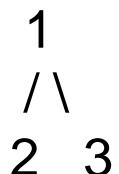
Trả lại giá trị đúng nếu và chỉ nếu hai cây với các nút gốc root1 và root2 có giá trị lá tương đồng.



Cấu trúc dữ liệu cây nâng cao

- Bài tập
 - **Bài 3:** Cho một cây nhị phân chỉ gồm các số từ 0 đến 9, mỗi đường đi từ gốc đến lá có thể biểu diễn một số. Ví dụ: đường đi từ gốc đến lá là 1->2->3 biểu diễn số 123
 - Tìm tổng cộng tất cả các số từ gốc đến lá

Đầu vào: [1,2,3]

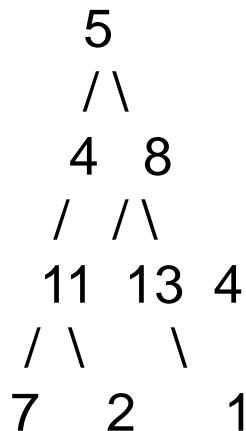


Đầu ra: $12 + 13 = 25$

Cấu trúc dữ liệu cây nâng cao

- Bài tập
 - **Bài 4:** Cho một cây nhị phân và một số a , xác định xem có đường đi từ gốc đến lá nào mà tổng giá trị của nó bằng a

Đầu vào: Với cây nhị phân bên dưới và $a = 22$



Đầu ra: True, tồn tại đường đi $5 \rightarrow 4 \rightarrow 11 \rightarrow 2$ mà tổng là 22

Cấu trúc dữ liệu cây nâng cao

- Cây đỏ đen (*red-black tree*)
 - Cây AVL duy trì sự cân bằng cứng nhắc hơn so với cây đỏ đen
 - Đường đi từ gốc đến lá sâu nhất
 - Trong cây AVL nhiều nhất là $\sim 1,44\log_2(n + 2)$
 - Trong cây đỏ đen nhiều nhất là $\sim 2\log_2(n + 1)$
 - ➔ Việc tra cứu trong cây AVL thường nhanh hơn
 - ➔ Nhưng phải trả giá bằng việc chèn và xóa chậm hơn do hoạt động xoay vòng nhiều hơn
 - ➔ Sử dụng cây AVL khi số lần tra cứu vượt trội so với số lượng cập nhật cây

Cấu trúc dữ liệu cây nâng cao

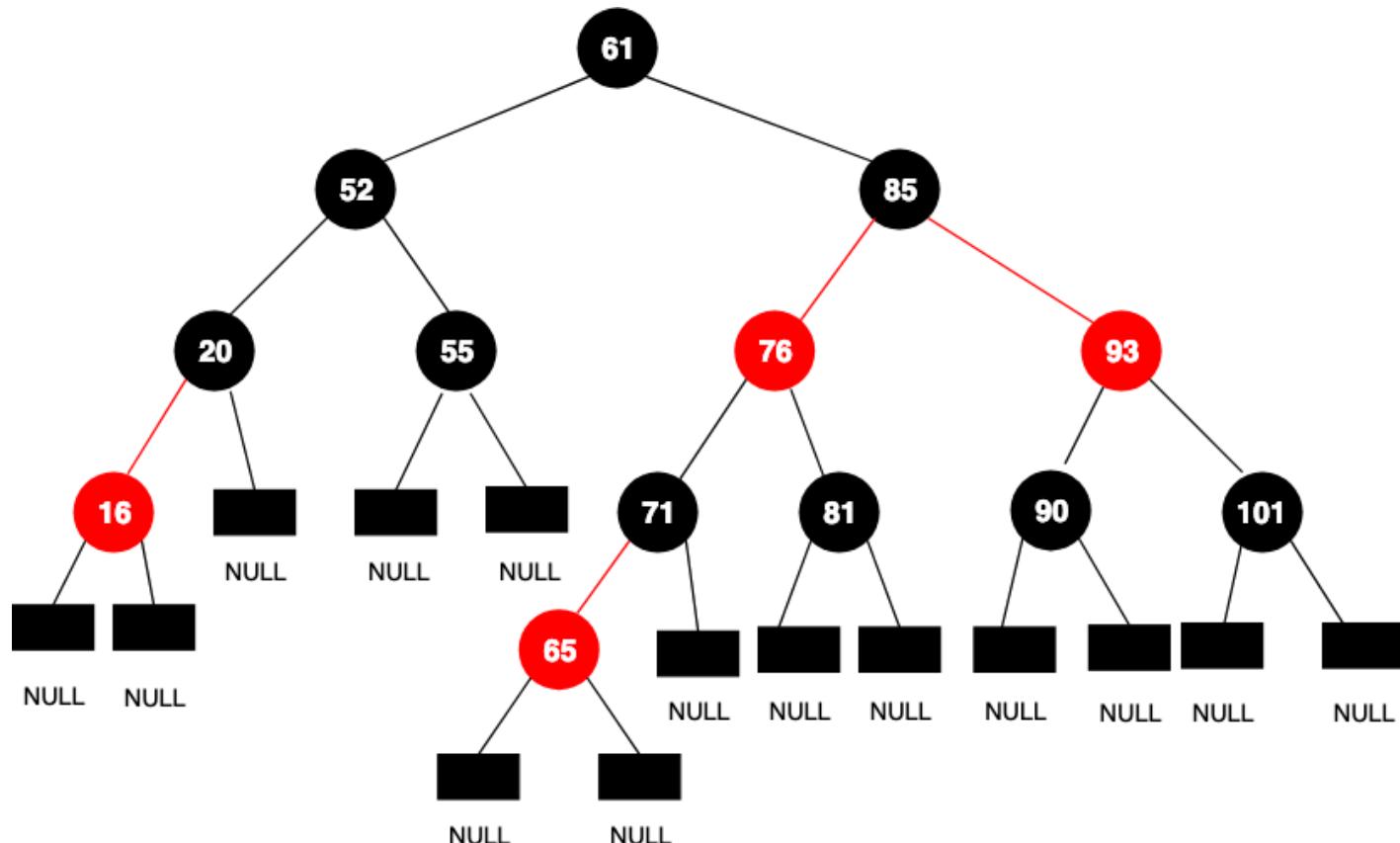
- Cây đỏ đen (*red-black tree*)
 - Là một dạng cây tìm kiếm nhị phân tự cân bằng
 - Cấu trúc ban đầu của nó được đưa ra vào năm 1972 bởi Rudolf Bayer với tên là “B-cây cân bằng”. Tên hiện nay được đưa ra từ 1978 bởi Leo J. Guibas và Robert Sedgewick.

Cấu trúc dữ liệu cây nâng cao

- Cây đỏ đen (*red-black tree*)
 - Mỗi nút của cây đỏ-đen có thuộc tính màu nhận một trong hai giá trị đỏ hoặc đen
 - Các tính chất:
 1. Một nút hoặc là đỏ hoặc là đen
 2. Gốc là đen
 3. Tất cả các lá (nút NULL) là đen
 4. Cả hai con của mọi nút đỏ là đen → mọi nút đỏ đều có nút cha là đen
 5. Tất cả các đường đi từ một nút bất kỳ tới các lá có số nút đen bằng nhau

Cấu trúc dữ liệu cây nâng cao

- Cây đỏ đen (*red-black tree*)
 - Một cây đỏ đen



Cấu trúc dữ liệu cây nâng cao

- Cây đỏ đen (*red-black tree*)
 - Một số đặc điểm
 - Các đường đi từ gốc tới các lá → đường đi dài nhất không vượt quá hai lần đường đi ngắn nhất
 - Không có đường đi nào từ gốc tới một lá chứa hai nút đỏ liền nhau
 - Trên mỗi đường số nút đỏ không nhiều hơn số nút đen
 - Đường đi ngắn nhất chỉ toàn nút đen
 - Đường đi dài nhất có thể xen kẽ giữa các nút đỏ và đen
 - Cây đỏ đen là cây gần cân bằng
 - Các thao tác chèn, xóa, tìm kiếm trên cây đỏ đen rất hiệu quả trong các trường hợp xấu nhất với độ phức tạp thời gian $O(\log_2 n)$

Cấu trúc dữ liệu cây nâng cao

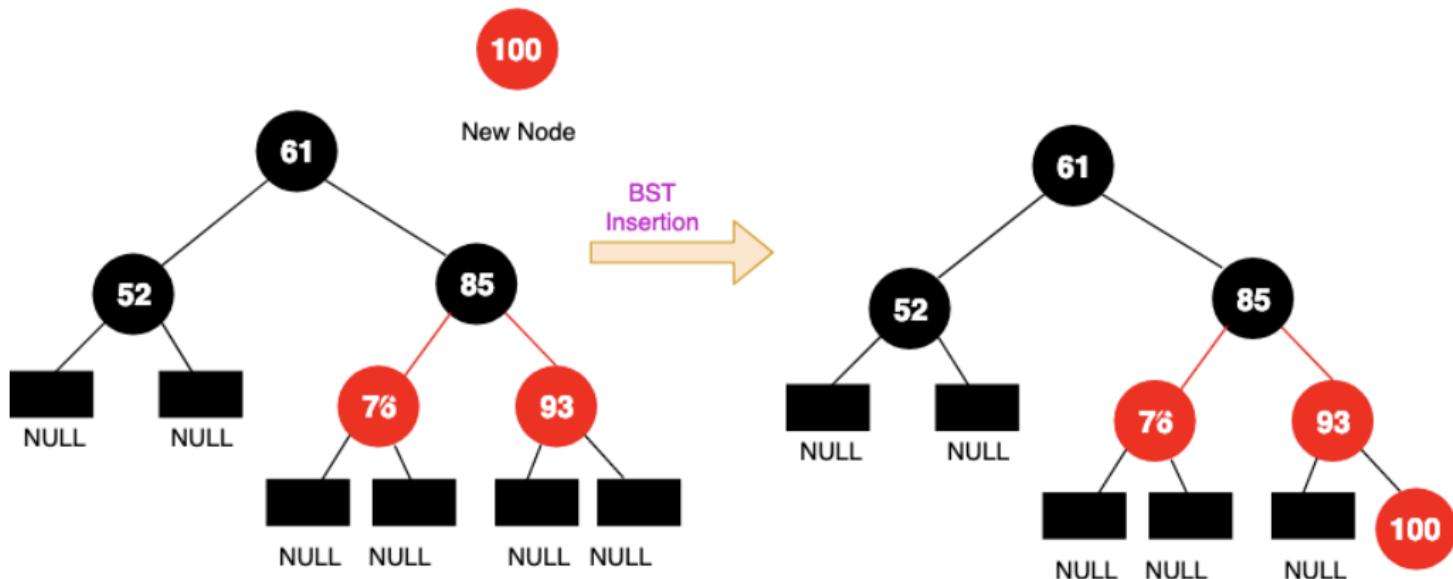
- Cây đỏ đen (*red-black tree*)
 - Thao tác chèn phần tử
 - Để chèn một nút K vào một cây đỏ đen T
 - Chèn K sử dụng thao tác chèn trên cây tìm kiếm nhị phân thông thường
 - Tô màu K là đỏ
 - Nếu thao tác chèn vi phạm các tính chất của cây đỏ đen thì điều chỉnh cây để khôi phục các tính chất đỏ đen
 - Xem xét một số trường hợp với giả sử rằng:
 - P là nút cha của K
 - U là nút chú bác của K
 - S là nút anh em của K
 - G là nút ông bà của K

Cấu trúc dữ liệu cây nâng cao

- Cây đỏ đen (*red-black tree*)
 - Thao tác chèn phần tử
 - **Trường hợp 1:** T là rỗng \rightarrow tạo K là nút gốc và tô màu đen
 - **Trường hợp 2:** P là nút đen \rightarrow không có tính chất nào bị vi phạm nên không cần làm gì
 - **Trường hợp 3:** P là nút đỏ \rightarrow vi phạm tính chất 4 vì P và K đều là nút đỏ. Nút ông bà G phải là nút đen \rightarrow kiểm tra nút chú bác U là đỏ hay đen

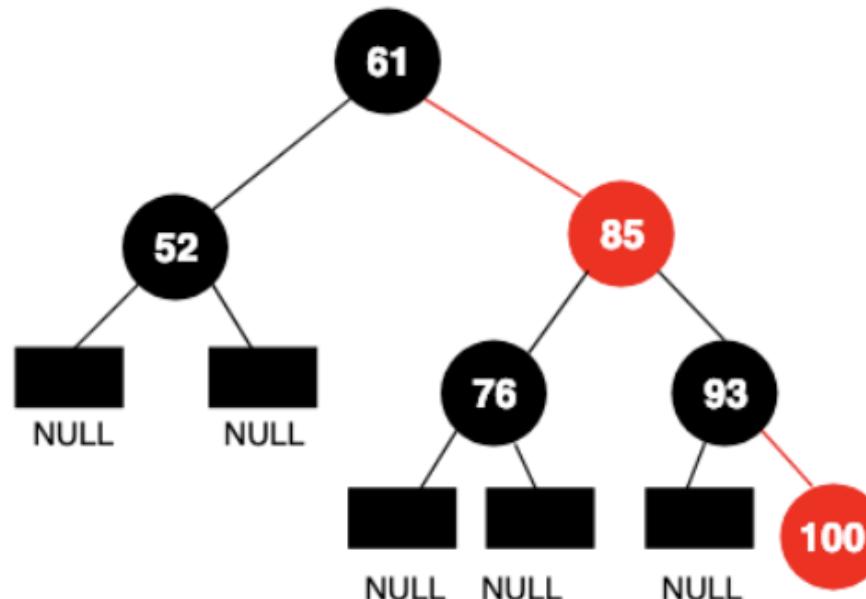
Cấu trúc dữ liệu cây nâng cao

- Cây đỏ đen (*red-black tree*)
 - Thao tác chèn phần tử
 - **Trường hợp 3.1:** P là nút đỏ và U cũng là nút đỏ



Cấu trúc dữ liệu cây nâng cao

- Cây đỏ đen (*red-black tree*)
 - Thao tác chèn phần tử
 - **Trường hợp 3.1:** P là nút đỏ và U cũng là nút đỏ
→ Đổi màu của cả P , U và G → P , U thành đen và G thành đỏ

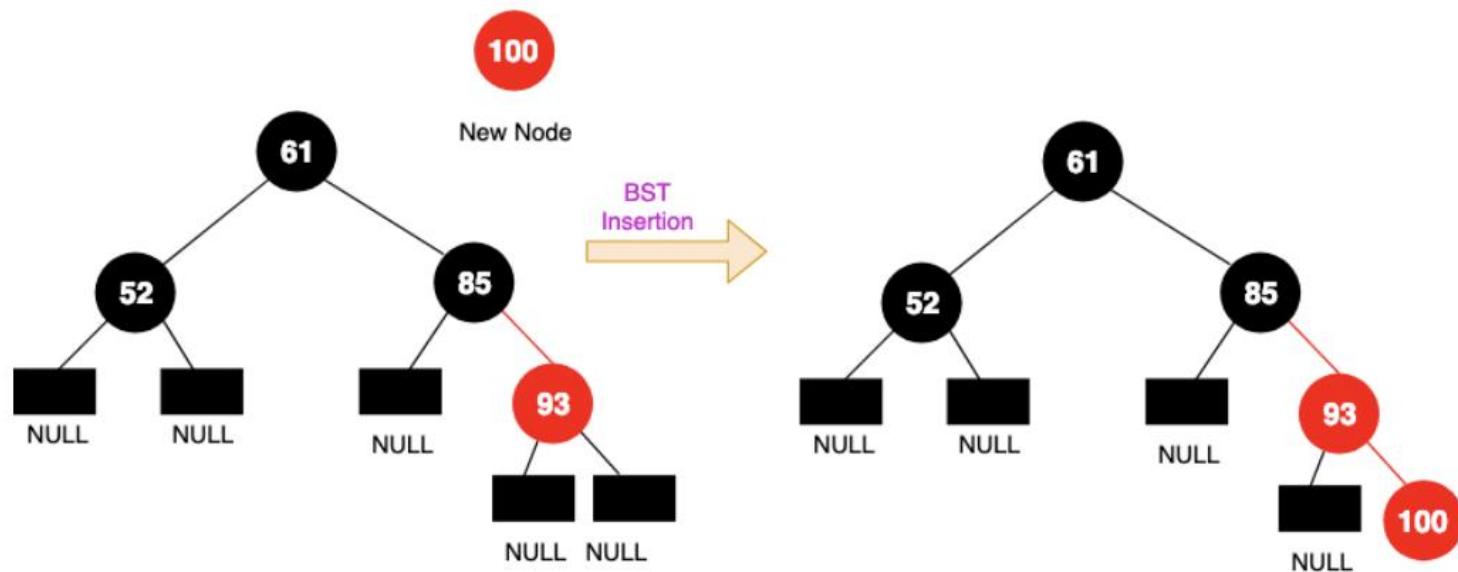


Cấu trúc dữ liệu cây nâng cao

- Cây đỏ đen (*red-black tree*)
 - Thao tác chèn phần tử
 - **Trường hợp 3.2:** P là nút đỏ và U là nút đen hoặc là NULL
→ cần thực hiện thao tác quay đơn hoặc quay kép tùy thuộc K là con trái hay phải của P

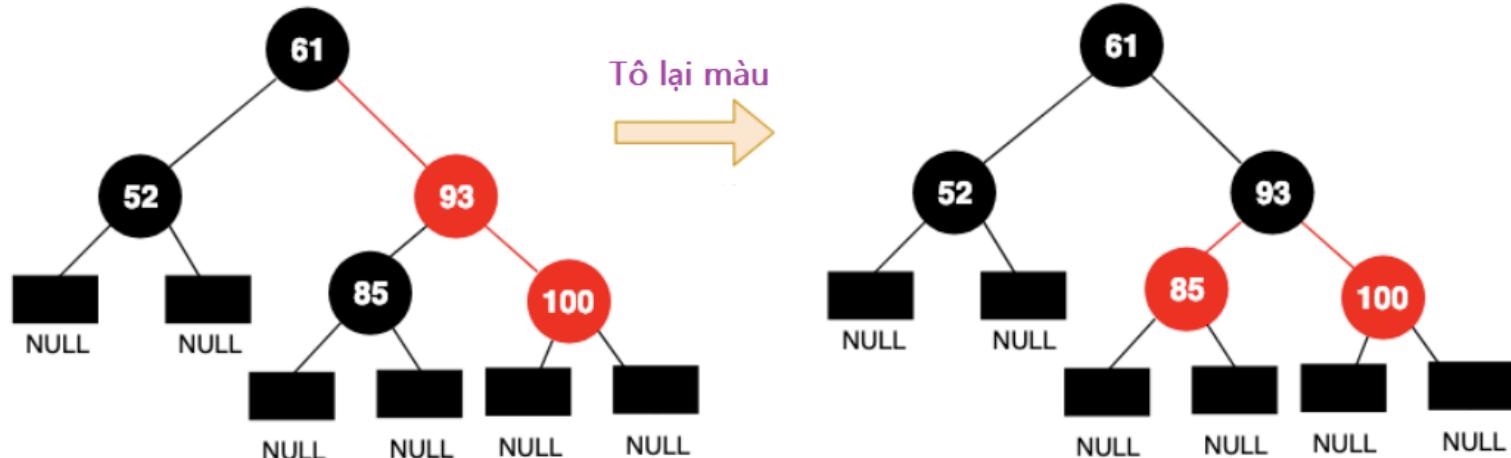
Cấu trúc dữ liệu cây nâng cao

- Cây đỏ đen (*red-black tree*)
 - Thao tác chèn phần tử
 - Trường hợp 3.2.1: P con **phải** của G và K là con **phải** của P



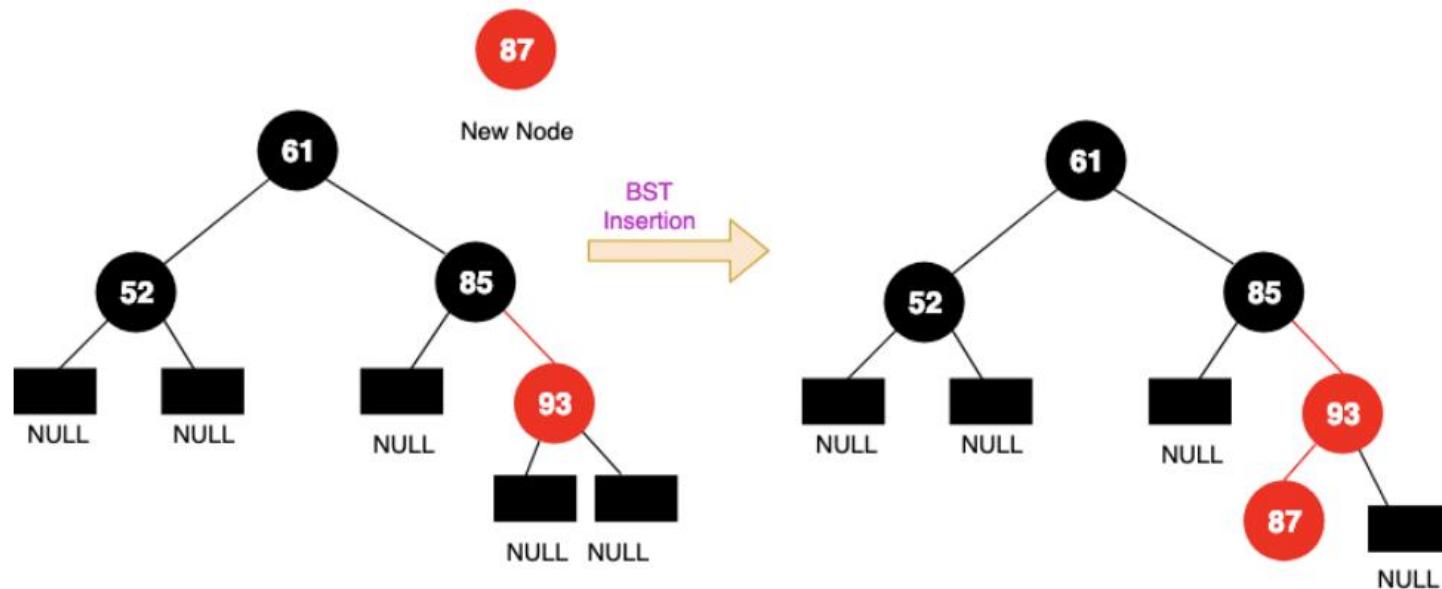
Cấu trúc dữ liệu cây nâng cao

- Cây đỏ đen (*red-black tree*)
 - Thao tác chèn phần tử
 - Trường hợp 3.2.1: P con **phải** của G và K là con **phải** của P
 - Quay trái tại $G \rightarrow G$ thành anh em S của K
 - Đổi màu của S thành đỏ và P thành đen



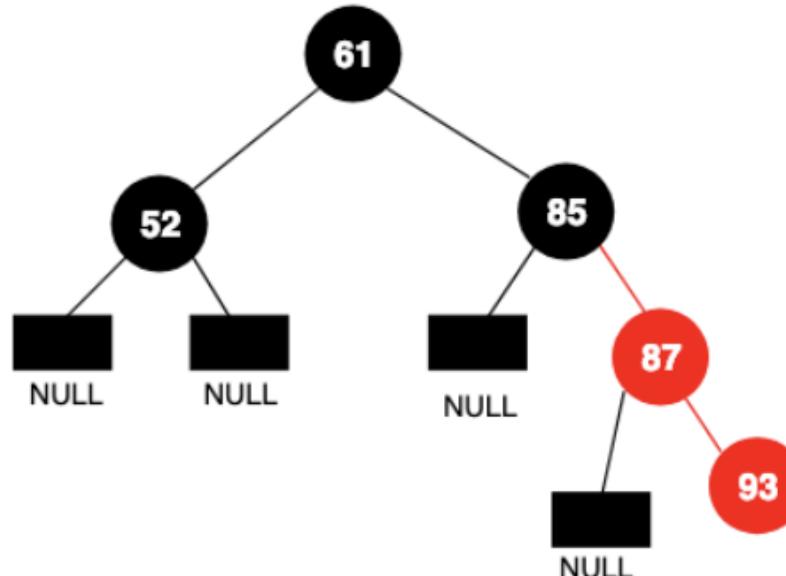
Cấu trúc dữ liệu cây nâng cao

- Cây đỏ đen (*red-black tree*)
 - Thao tác chèn phần tử
 - Trường hợp 3.2.2: P con phải của G và K là con trái của P



Cấu trúc dữ liệu cây nâng cao

- Cây đỏ đen (*red-black tree*)
 - Thao tác chèn phần tử
 - Trường hợp 3.2.2: P con **phải** của G và K là con **trái** của P
 - Quay phải tại $P \rightarrow$ trở về **trường hợp 3.2.1**



Cấu trúc dữ liệu cây nâng cao

- Cây đỏ đen (*red-black tree*)
 - Thao tác chèn phần tử
 - **Trường hợp 3.2.3:** P con **trái** của G và K là con **trái** của $P \rightarrow$ ngược với trường hợp 3.2.1
 - **Trường hợp 3.2.4:** P con **trái** của G và K là con **phải** của $P \rightarrow$ ngược với trường hợp 3.2.2

Cấu trúc dữ liệu cây nâng cao

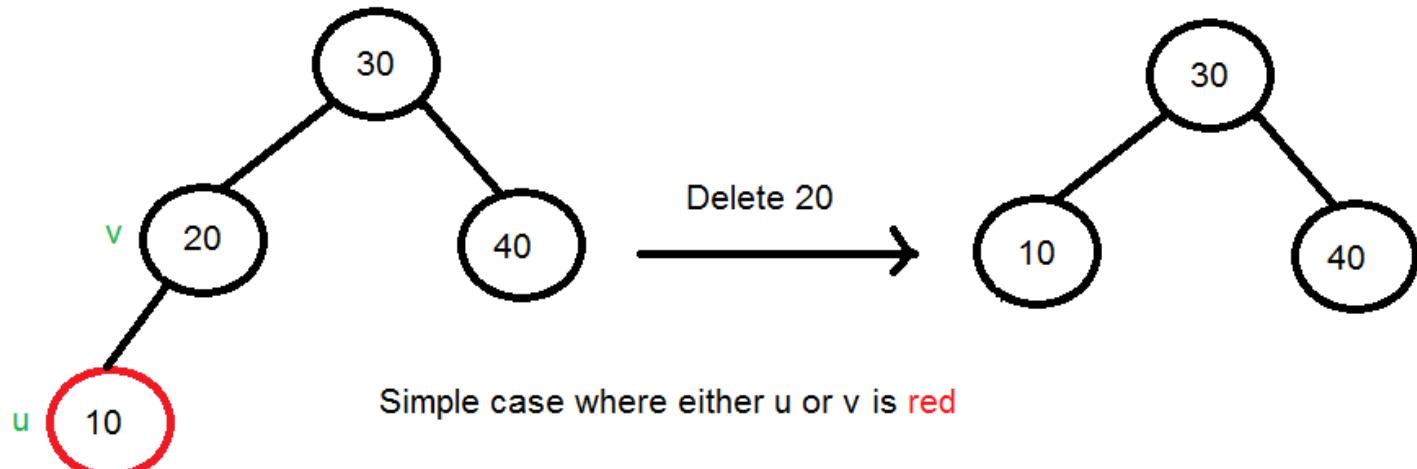
- Cây đỏ đen (*red-black tree*)
 - Thao tác xóa phần tử
 - Xóa nút v khỏi cây đỏ đen \rightarrow thực hiện giống như đối với cây tìm kiếm nhị phân \rightarrow đảm bảo rằng v là lá hoặc có duy nhất một con
 - Do đó, chỉ xử lý các trường hợp nút bị xóa là lá hoặc có một con
 - Giả sử v là nút bị xóa và u nút thay thế v (chú ý rằng u là NULL khi v là lá)

Cấu trúc dữ liệu cây nâng cao

- Cây đỏ đen (*red-black tree*)
 - Thao tác xóa phần tử
 - **Trường hợp 1:** v là gốc \rightarrow sau khi xóa v thì u là nút gốc NULL

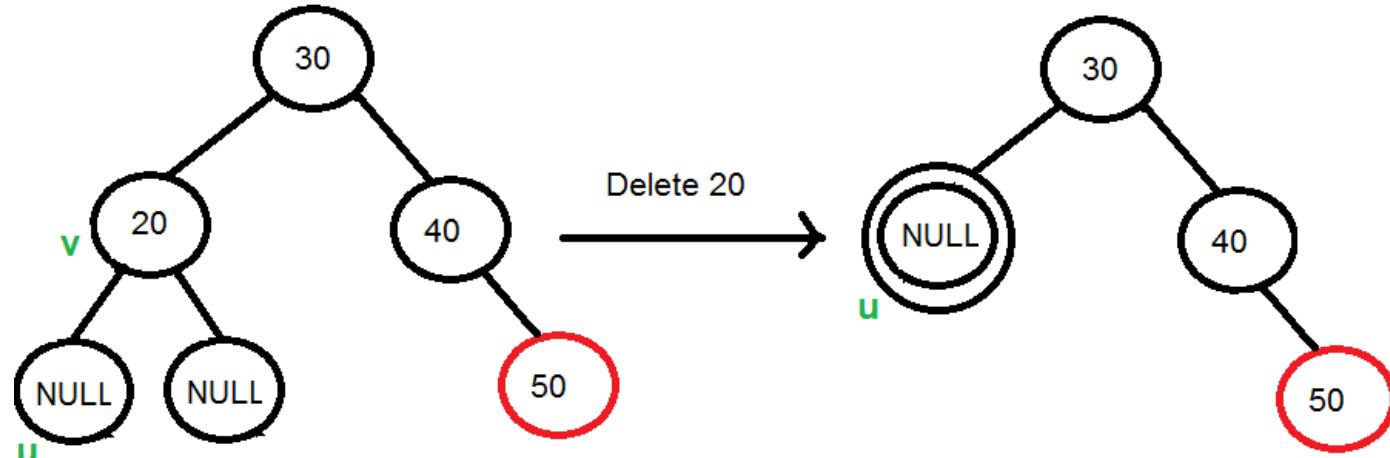
Cấu trúc dữ liệu cây nâng cao

- Cây đỏ đen (*red-black tree*)
 - Thao tác xóa phần tử
 - **Trường hợp 2:** Hoặc u hoặc v là đỏ \rightarrow xóa v và đổi màu u thành đen



Cấu trúc dữ liệu cây nâng cao

- Cây đỏ đen (*red-black tree*)
 - Thao tác xóa phần tử
 - **TH 3:** Cả u và v là đen
 - *TH 3.1:* u thành nút đen kép \rightarrow cần chuyển nút đen kép thành nút đen đơn
 - Lưu ý: nếu v là lá và u là NULL \rightarrow khi xóa v , để duy trì tính chất 5 (cả trước và sau biến đổi) \rightarrow coi u là nút đen kép

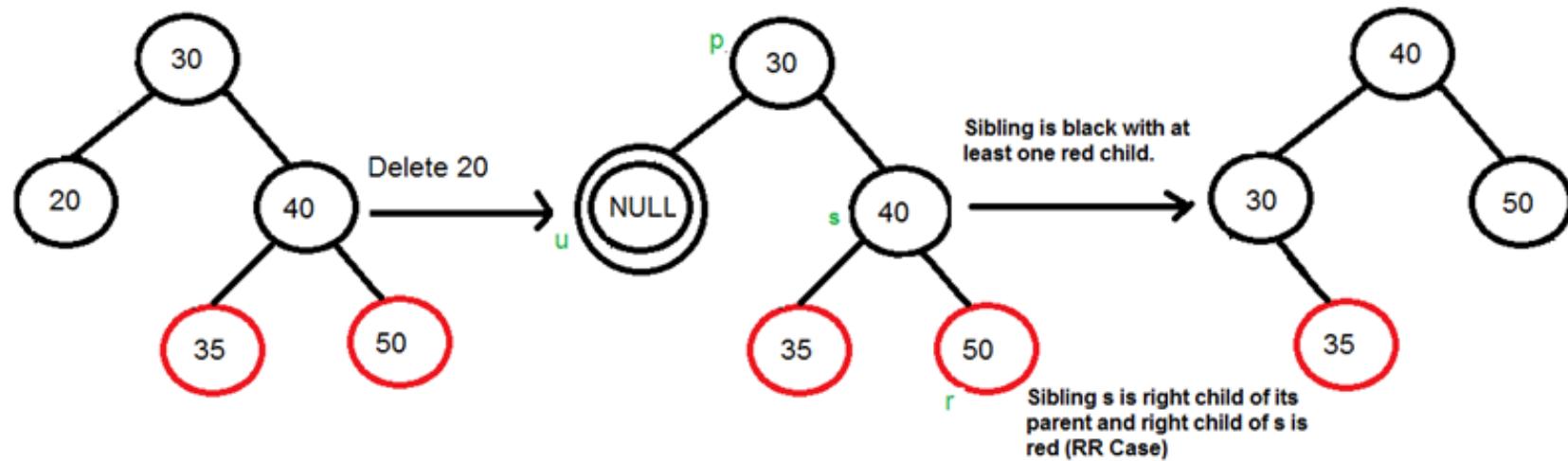


Cấu trúc dữ liệu cây nâng cao

- Cây đỏ đen (*red-black tree*)
 - Thao tác xóa phần tử
 - TH 3.2: với u là nút đen kép và không phải là gốc, s là nút anh em của u
 - TH 3.2.a: s là đen và **ít nhất một con của nó là đỏ** và ký hiệu là $r \rightarrow$ chia làm 4 trường hợp phụ thuộc vào vị trí của s và r .

Cấu trúc dữ liệu cây nâng cao

- Cây đỏ đen (*red-black tree*)
 - Thao tác xóa phần tử
 - (i) **Phải phải** (cả s và r đều là con phải)

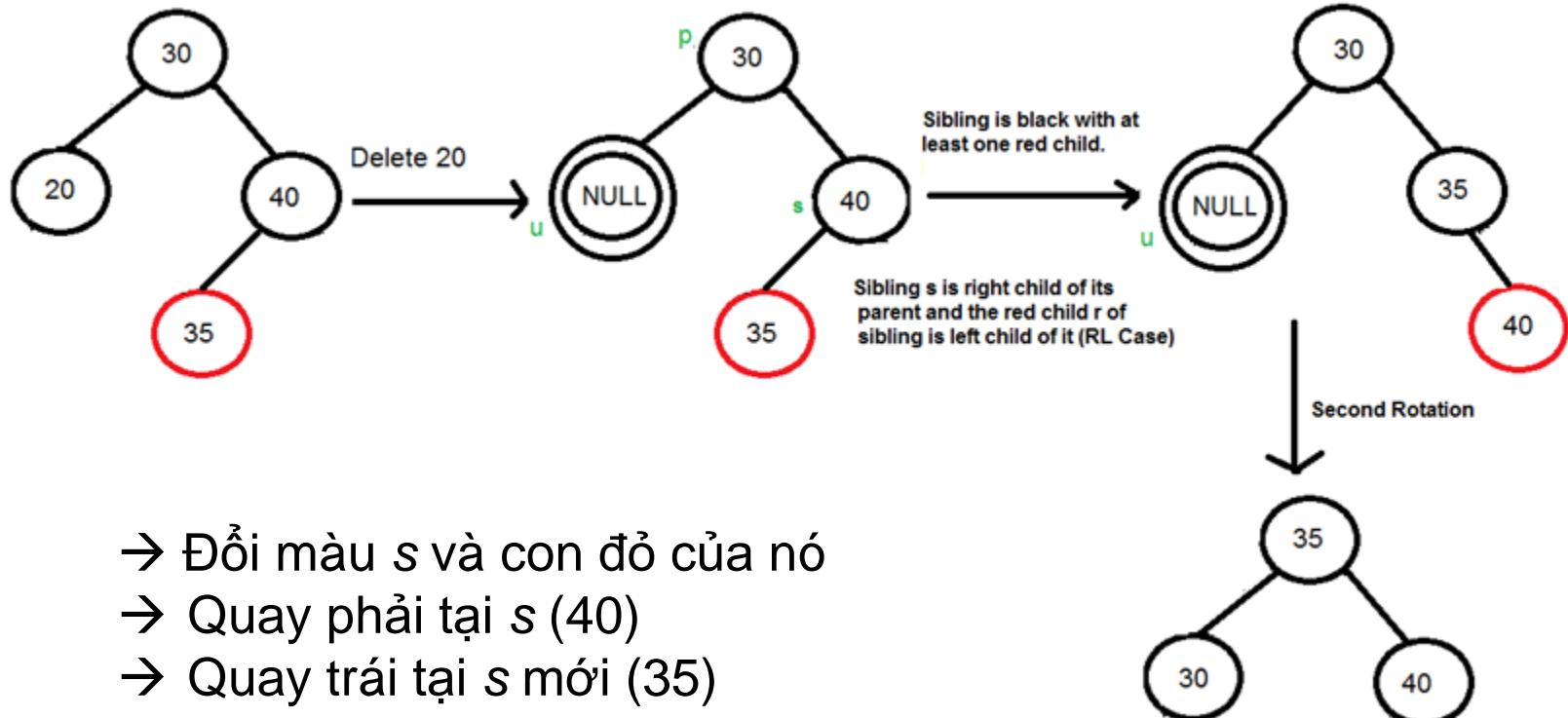


→ Đổi màu của r thành đen

→ Quay trái tại p là bố mẹ của u và s (loại bỏ nút đen kép)

Cấu trúc dữ liệu cây nâng cao

- Cây đỏ đen (*red-black tree*)
 - Thao tác xóa phần tử
 - (ii) **Phải trái** (s là con phải và r là con trái)

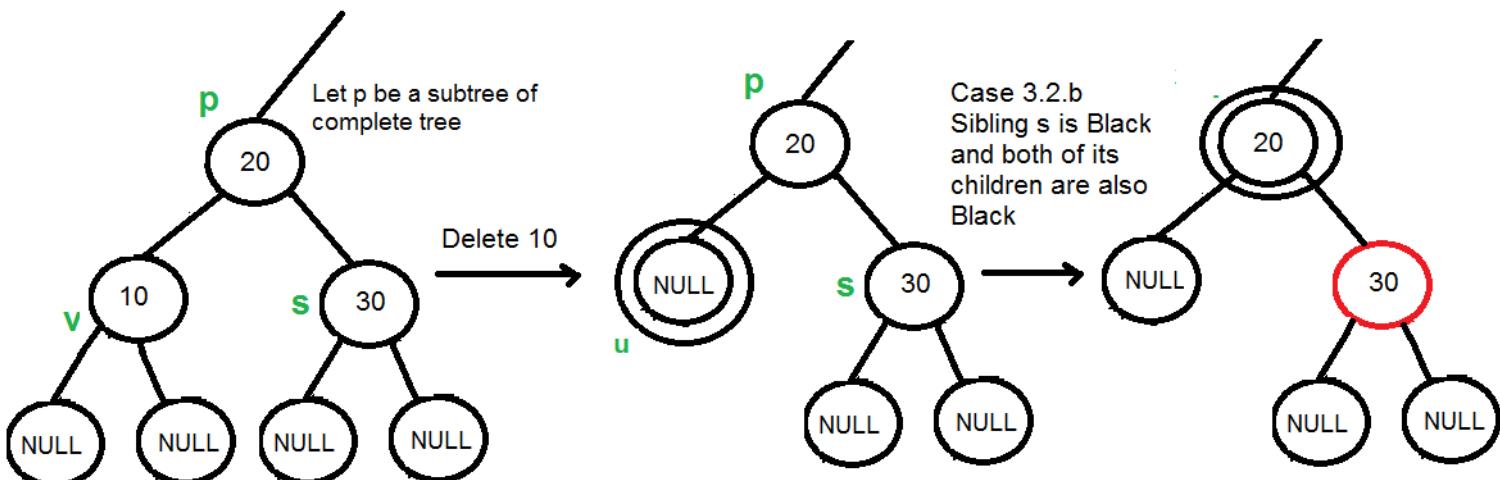


Cấu trúc dữ liệu cây nâng cao

- Cây đỏ đen (*red-black tree*)
 - Thao tác xóa phần tử
 - (iii) **Trái trái** (cả s và r đều là con trái) → ngược với trường hợp **Phải phải** ở trên
 - (iv) **Trái phải** (s là con trái và r là con phải) → ngược với trường hợp **Phải trái** ở trên

Cấu trúc dữ liệu cây nâng cao

- Cây đỏ đen (*red-black tree*)
 - Thao tác xóa phần tử
 - TH 3.2.b: s là đen và hai con của nó cũng là đen thì đổi s thành đỏ
 - Nếu cha (p) là đỏ thì đổi thành đen (vì trạng thái cũ là: đỏ (p) + đen kép (u) = đen), tạo u thành đen đơn
 - Nếu cha (p) là đen → để bù nút đen khi nút s thành đỏ thì chuyển một đen của u lên p và thực hiện TH 3.2.c

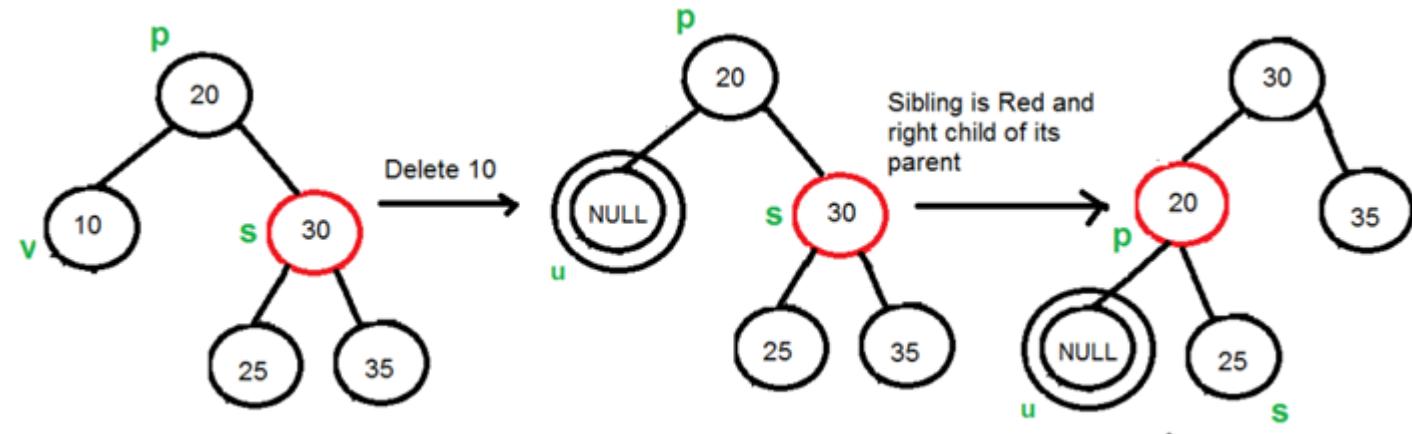


Cấu trúc dữ liệu cây nâng cao

- Cây đỏ đen (*red-black tree*)

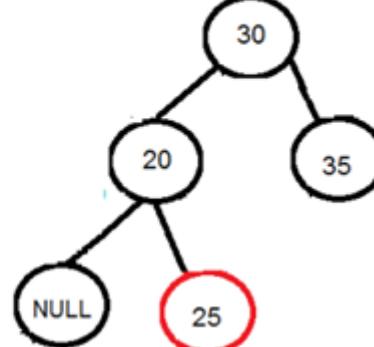
- TH 3.2.c: s là **đỏ**

- (i) **Phải** (s là con phải)



→ Quay trái tại *p*

→ Đổi màu của *p*, *u* và *s* → TH 3.2.b



Cấu trúc dữ liệu cây nâng cao

- Cây đỏ đen (*red-black tree*)
 - Thao tác xóa phần tử
 - *TH 3.2.c:* s là **đỏ**
 - (ii) **Trái** (s là con trái) → ngược với trường hợp **Phải** ở trên → quay phải tại p

Cấu trúc dữ liệu cây nâng cao

- Cây đỏ đen (*red-black tree*)
 - Ứng dụng
 1. Completely Fair Scheduler in Linux Kernel
 2. Computational Geometry Data structures
 3. Cấu trúc tập hợp set trong C++ STL
 4. Cấu trúc map trong C++ STL

Cấu trúc dữ liệu cây nâng cao

- Cây đỏ đen (*red-black tree*)
 - Bài tập
 1. Cài đặt lại cấu trúc tập hợp set sử dụng cấu trúc cây đỏ đen
 2. Cài đặt lại cấu trúc map sử dụng cấu trúc cây đỏ đen

Cấu trúc dữ liệu cây nâng cao

- Cây 2-3-4

- Mỗi nút có thể lưu trữ 1, 2 hoặc 3 mục dữ liệu



- Các số 2, 3 và 4 trong cụm từ cây 2-3-4 → khả năng có bao nhiêu liên kết đến các nút con có thể có được trong một nút cho trước

Cấu trúc dữ liệu cây nâng cao

- Cây 2-3-4

- Đối với các nút không phải là lá, có thể có 3 cách sắp xếp sau:
 - Một nút với một mục dữ liệu thì luôn luôn có 2 con
 - Một nút với hai mục dữ liệu thì luôn luôn có 3 con
 - Một nút với ba mục dữ liệu thì luôn luôn có 4 con
- Như vậy, một nút không phải là lá phải luôn luôn có số nút con nhiều hơn 1 so với số mục dữ liệu của nó

Có nghĩa là mọi nút với số con là k và số mục dữ liệu là d thì $k = d + 1$

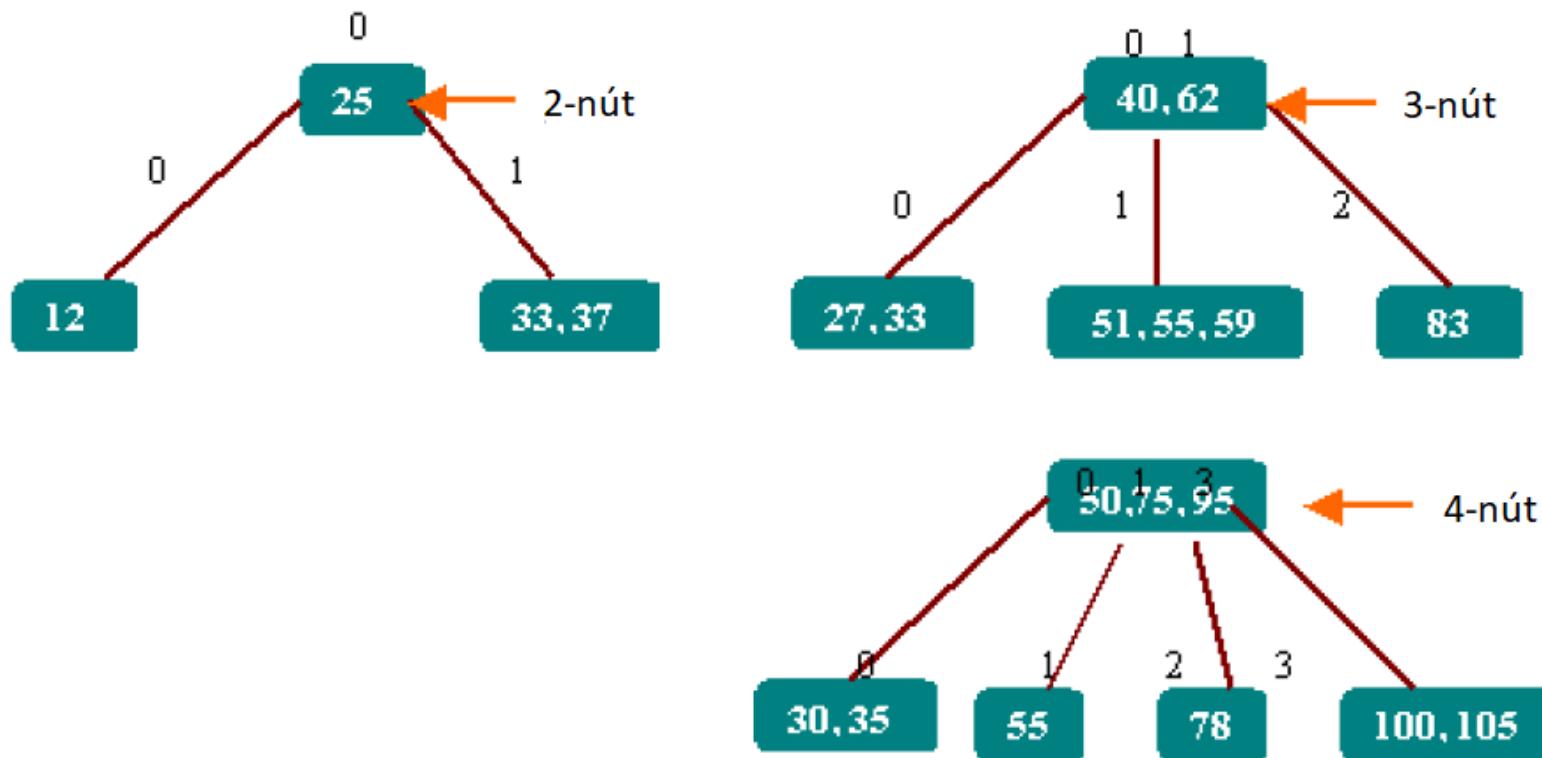
Cấu trúc dữ liệu cây nâng cao

- Cây 2-3-4

- Với mọi nút lá thì không có nút con nhưng có thể chứa 1, 2 hoặc 3 mục dữ liệu
- Một cây 2-3-4 có thể có đến 4 cây con nên được gọi là *cây nhiều nhánh bậc 4*
- Trong cây 2-3-4 mỗi nút có ít nhất là 2 liên kết, trừ nút lá (nút không có liên kết nào)

Cấu trúc dữ liệu cây nâng cao

- Cây 2-3-4
 - Các trường hợp của cây 2-3-4
 - Một nút với 2 liên kết gọi là một 2-nút, một nút với 3 liên kết gọi là một 3-nút và một nút với 4 liên kết gọi là một 4-nút



Cấu trúc dữ liệu cây nâng cao

- Tổ chức cây 2-3-4
 - Các mục dữ liệu trong mỗi nút được sắp xếp theo thứ tự tăng dần từ trái sang phải (từ thấp đến cao)
 - Tất cả nút của cây con bên trái có khóa nhỏ hơn khóa của nút đang xét và tất cả nút của cây con bên phải có khóa lớn hơn hoặc bằng khóa của nút đang xét. Ngoài ra,
 - Tất cả các node con của cây con có gốc tại nút con thứ 0 thì có các giá trị khóa nhỏ hơn khóa 0
 - Tất cả các node con của cây con có gốc tại nút con thứ 1 thì có các giá trị khóa lớn hơn khóa 0 và nhỏ hơn khóa 1
 - Tất cả các node con của cây con có gốc tại nút con thứ 2 thì có các giá trị khóa lớn hơn khóa 1 và nhỏ hơn khóa 2
 - Tất cả các node con của cây con có gốc tại nút con thứ 3 thì có các giá trị khóa lớn hơn khóa 2

Cấu trúc dữ liệu cây nâng cao

- Tổ chức cây 2-3-4
 - Các nút lá đều nằm trên cùng một mức
 - Các nút ở mức trên thường không đầy đủ, nghĩa là chúng có thể chỉ chứa 1 hoặc 2 mục dữ liệu thay vì 3 mục
 - Cây 2-3-4 là cây cân bằng



Cấu trúc dữ liệu cây nâng cao

- Tìm kiếm
 - Tương tự như thủ tục tìm kiếm trong cây nhị phân
 - Việc tìm kiếm bắt đầu từ nút gốc và chọn liên kết dẫn đến cây con với phạm vi giá trị phù hợp
 - Ví dụ: để tìm kiếm mục dữ liệu với khoá là 64. Tại nút gốc không tìm thấy mục khoá này. Vì 64 lớn 50, chúng ta đi đến nút con 1 (60/70/80) vì đánh số các nút con và các liên kết bắt đầu tại 0 từ bên trái. Tại đây bởi vì 64 lớn hơn 60 nhưng nhỏ hơn 70 nên đi tiếp đến nút con 1 → tìm thấy tại đây

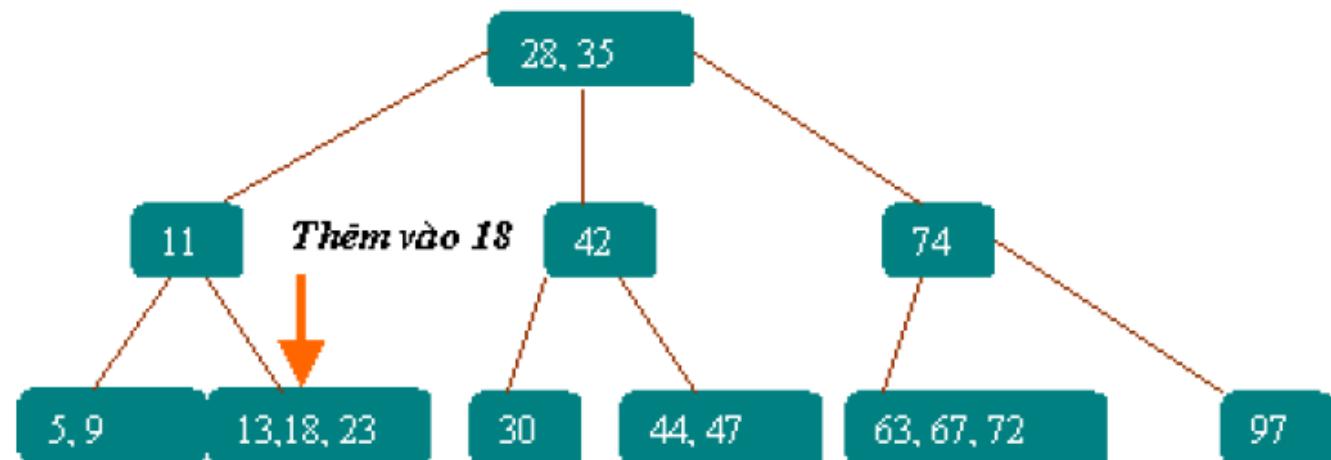
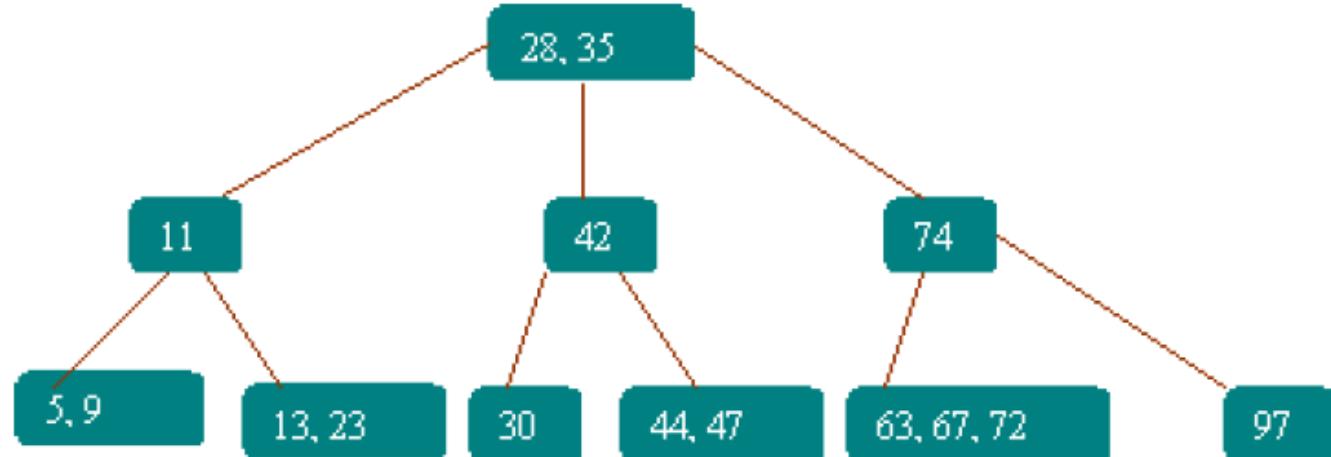


Cấu trúc dữ liệu cây nâng cao

- Thêm vào
 - Các mục dữ liệu mới luôn luôn được chèn vào tại các nút lá
 - Nếu mục dữ liệu được thêm vào nút mà có nút con thì số lượng của các nút con cần thiết phải được biến đổi để duy trì cấu trúc cho cây
 - Trong bất cứ trường hợp nào thì quá trình thêm cũng bắt đầu bằng cách tìm kiếm nút lá phù hợp
 - Khi nút lá phù hợp (*chưa đủ 3 mục dữ liệu*) được tìm thấy, đơn giản là thêm mục dữ liệu mới vào
 - Việc chèn vào có thể dẫn đến phải thay đổi vị trí của một hoặc hai mục dữ liệu

Cấu trúc dữ liệu cây nâng cao

- Thêm vào

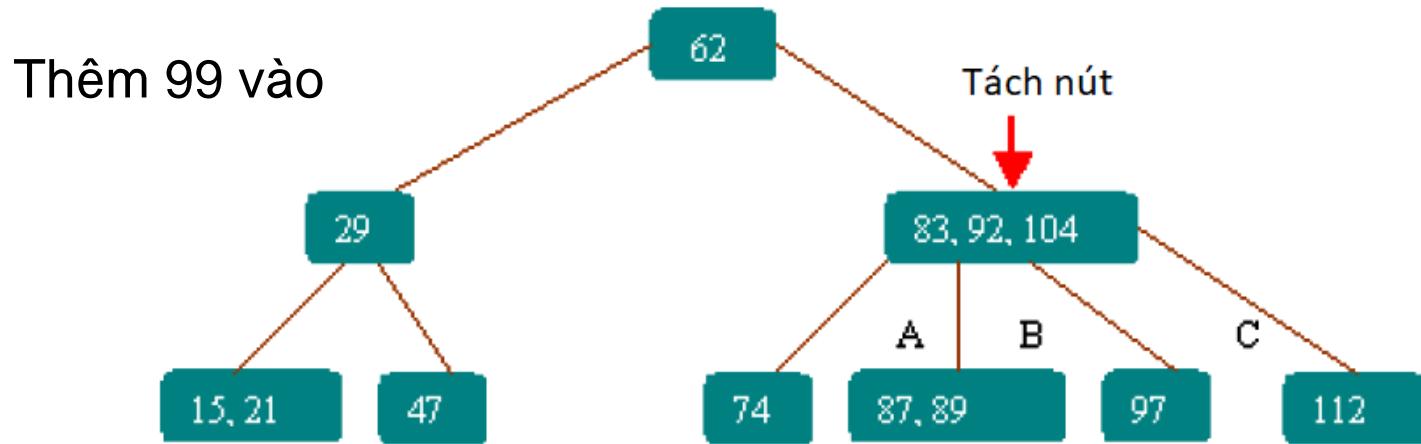


Cấu trúc dữ liệu cây nâng cao

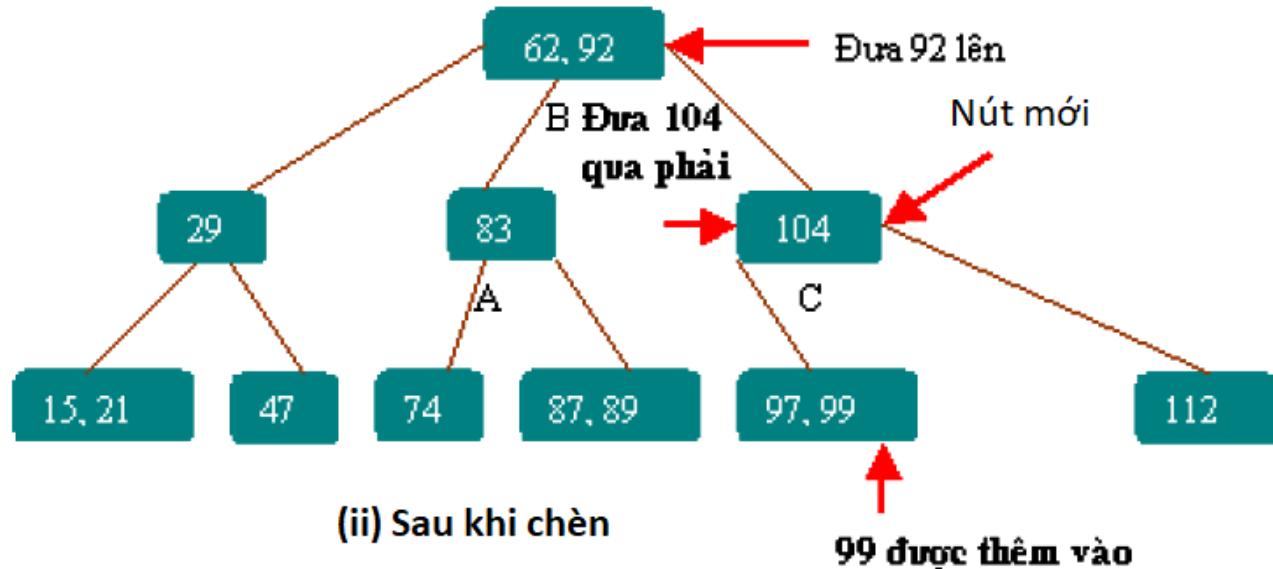
- Tách nút
 - Khi thêm vào có thể gặp nút đầy (đủ 3 mục dữ liệu)
→ nút này cần phải được tách ra để giữ cho cây cân bằng
 - Giả sử nút bị tách không phải là nút gốc và đặt tên các mục dữ liệu trên nút bị phân chia là A, B và C
 - Một nút mới (rỗng) được tạo. Nó là anh em với nút sẽ được tách và được đưa vào bên phải của nó
 - Mục dữ liệu C (phải nhất) được đưa vào nút mới
 - Mục dữ liệu B được đưa vào nút cha của nút được tách
 - Mục dữ liệu A (trái nhất) không thay đổi
 - Hai nút con bên phải nhất bị hủy kết nối từ nút được tách và kết nối đến nút mới

Một 4-nút được tách thành hai 2-nút

Cấu trúc dữ liệu cây nâng cao



(i) Trước khi chèn



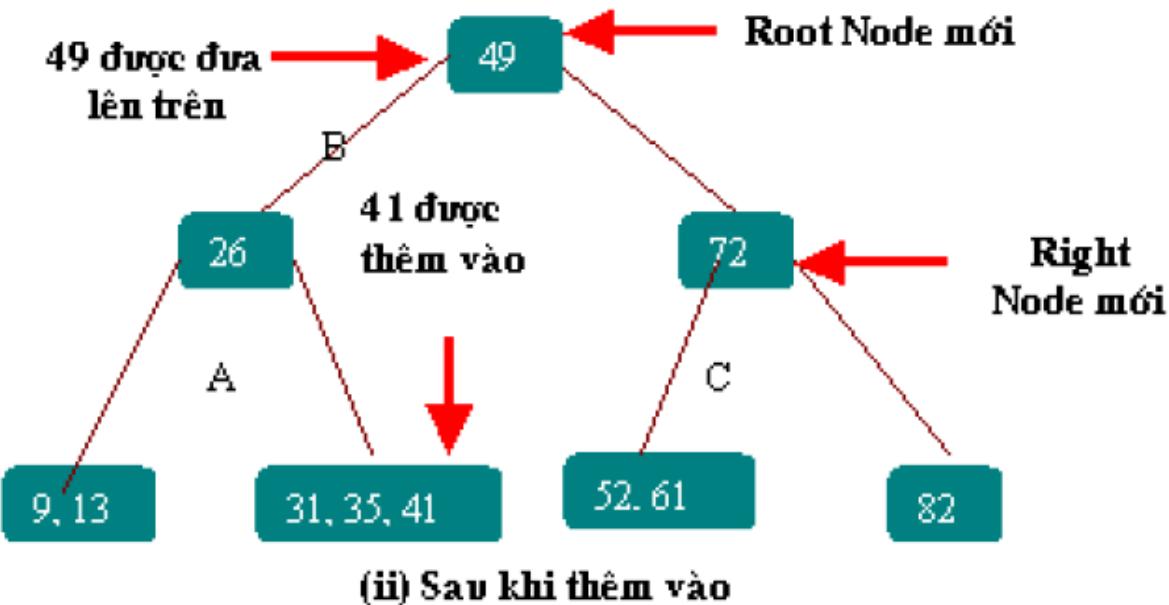
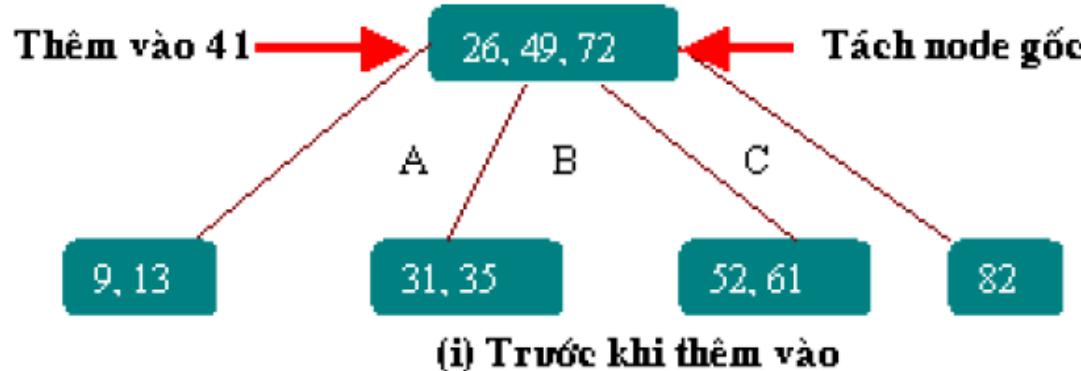
(ii) Sau khi chèn

Cấu trúc dữ liệu cây nâng cao

- Tách nút gốc
 - Khi gặp phải nút gốc đầy tại thời điểm bắt đầu tìm kiếm điểm chèn, kết quả của việc tách được thực hiện như sau
 - Nút mới được tạo ra để trở thành gốc mới và là cha của nút được tách
 - Mục dữ liệu C được dịch đưa sang nút anh em mới
 - Mục dữ liệu B được dịch đưa sang nút gốc mới
 - Mục dữ liệu A vẫn không đổi
 - Hai nút con bên phải nhất của nút được phân chia bị hủy kết nối khỏi nó và kết nối đến nút mới bên phải

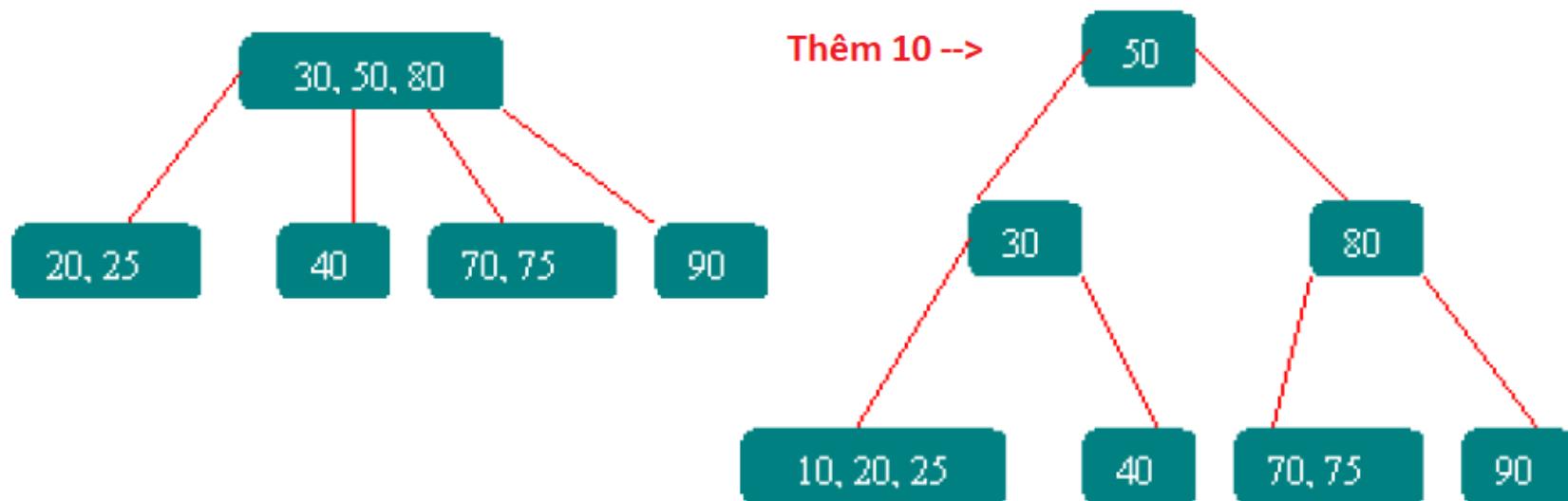
Cấu trúc dữ liệu cây nâng cao

- Tách nút gốc



Cấu trúc dữ liệu cây nâng cao

- Tách theo hướng đi xuống
 - Vì tất cả các nút đầy được tách trên đường đi xuống nên việc tách nút không gây ảnh hưởng gì khi phải đi ngược lên trên của cây
 - Nút cha của bất cứ nút nào bị tách phải đảm bảo rằng không phải là nút đầy → để đảm bảo nút cha này có thể chấp nhận mục dữ liệu B mà không cần thiết nó phải tách ra



TỔNG KẾT