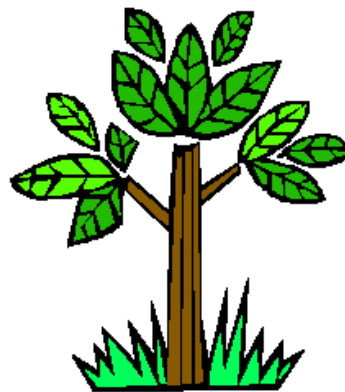




Cây nhị phân

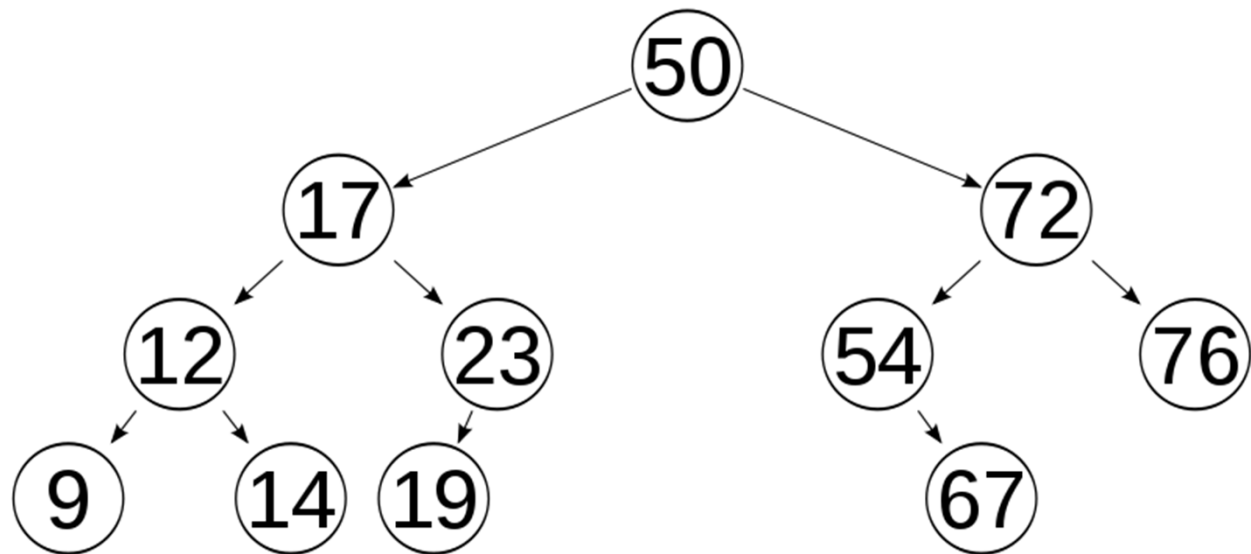
- Các khái niệm và thuật ngữ cơ bản
- Cài đặt cấu trúc dữ liệu
- Duyệt cây
- Cây nhị phân tìm kiếm – Binary Search Tree
- Hàng đợi ưu tiên – Priority Queue





Cây nhị phân tìm kiếm (BST)

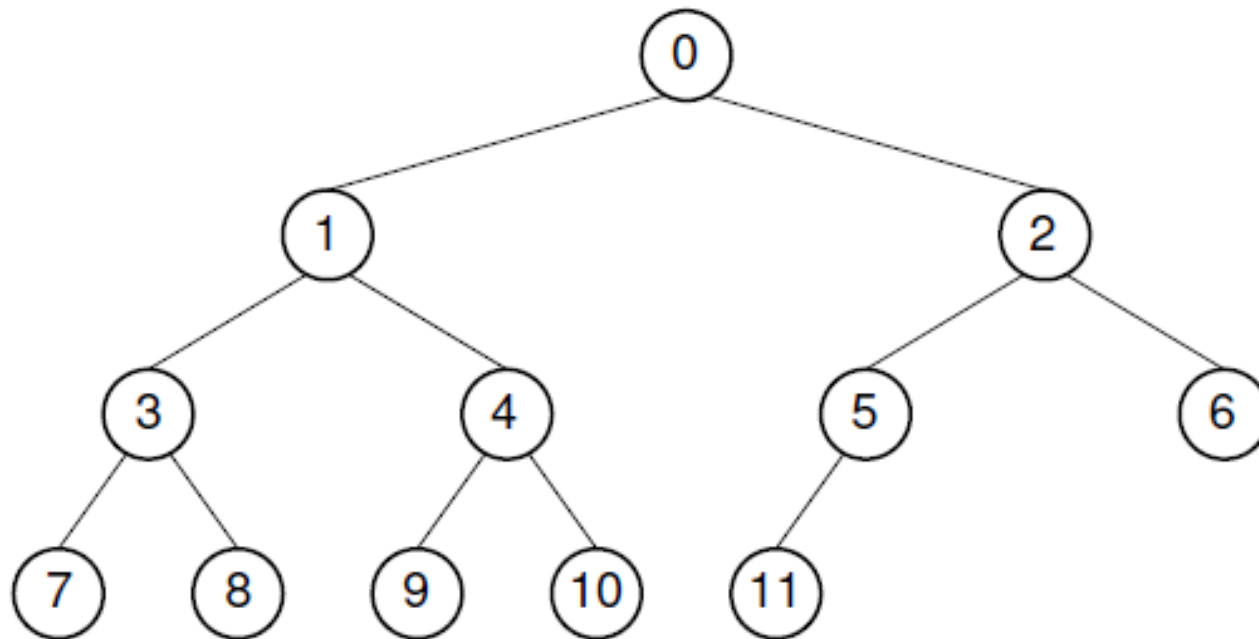
- Ý nghĩa của cây BST
- Binary Search Tree ADT
- Cài đặt cấu trúc dữ liệu BST
- Đánh giá/So sánh
- Bài tập





Ý nghĩa của cây BST (1)

- Tìm 1 phần tử trong cây nhị phân ?
 - Thuật toán ?
 - Chi phí ?





Ý nghĩa của cây BST (2)

- Điểm yếu và điểm mạnh của mảng ?
- Điểm yếu và điểm mạnh của danh sách liên kết ?
- Một cấu trúc dữ liệu có được cả điểm mạnh của mảng và danh sách liên kết ?

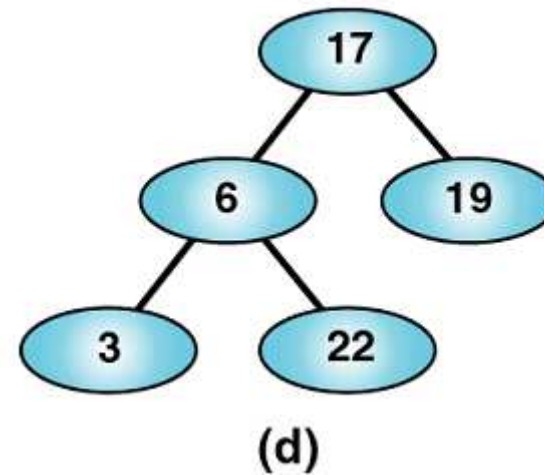
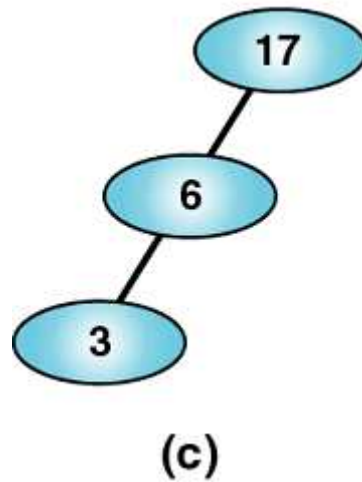
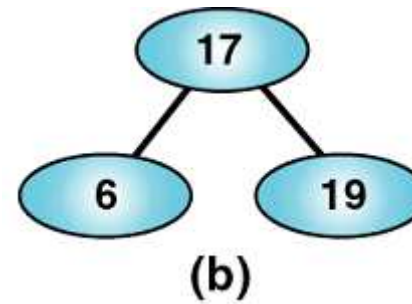
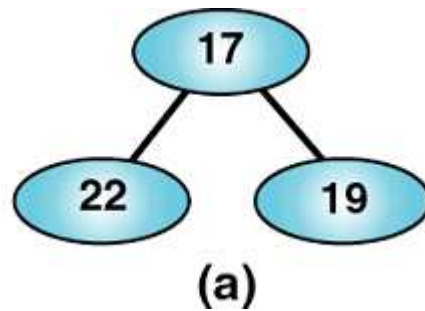


Binary Search Tree ADT (1)

- Cây nhị phân tìm kiếm là:
 - Một cây nhị phân
 - Mỗi node có một khóa (key)
 - Mỗi node p của cây đều thỏa:
 - Tất cả các node thuộc cây con trái đều có khóa nhỏ hơn khóa của p
 $\forall q \in p \rightarrow \text{left}: q \rightarrow \text{key} < p \rightarrow \text{key}$
 - Tất cả các node thuộc cây con phải đều có khóa lớn hơn khóa của p
 $\forall q \in p \rightarrow \text{right}: q \rightarrow \text{key} > p \rightarrow \text{key}$



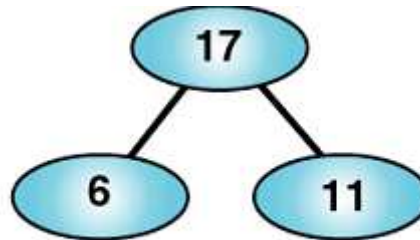
Binary Search Tree ADT (2)



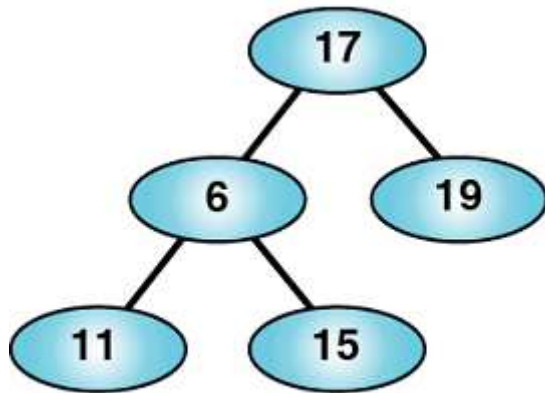
Binary Search Tree ADT (3)



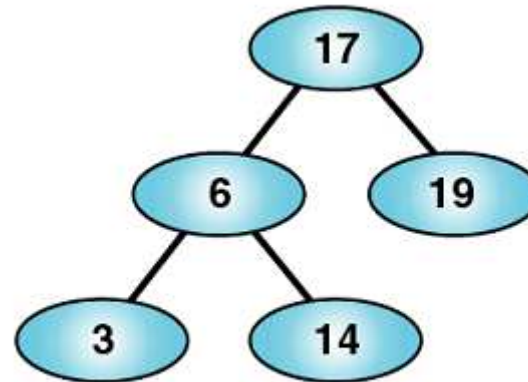
(a)



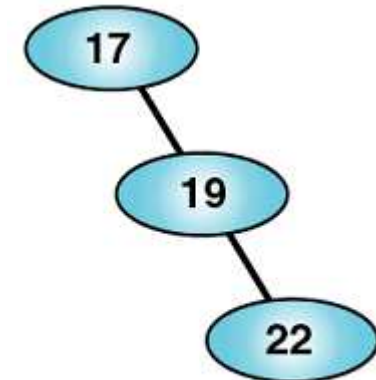
(b)



(c)



(d)



(e)



Binary Search Tree ADT (4)

- Các thao tác cơ bản:
 - Khởi tạo cây rỗng
 - Xóa cây
 - Thêm một node
 - Xóa một node
 - Tìm một node
 - Duyệt cây
 - Kiểm tra cây rỗng
 - Đếm số node trong cây
 - Tính chiều cao của cây



Cài đặt cấu trúc dữ liệu BST (1)

```
template <class T> class BSTNode {
public:
    T            key;           // key of node
    BSTNode      *left;         // pointer to left child
    BSTNode      *right;        // pointer to right child

    BSTNode() { }
    BSTNode(T aKey)
    {
        key = aKey;
        left = right = NULL;
    }
}; // end class
```



Cài đặt cấu trúc dữ liệu BST (2)

```
template <class T> class BINARY_SEARCH_TREE {
private:
    BSTNode<T>          *root;                // pointer to root of tree

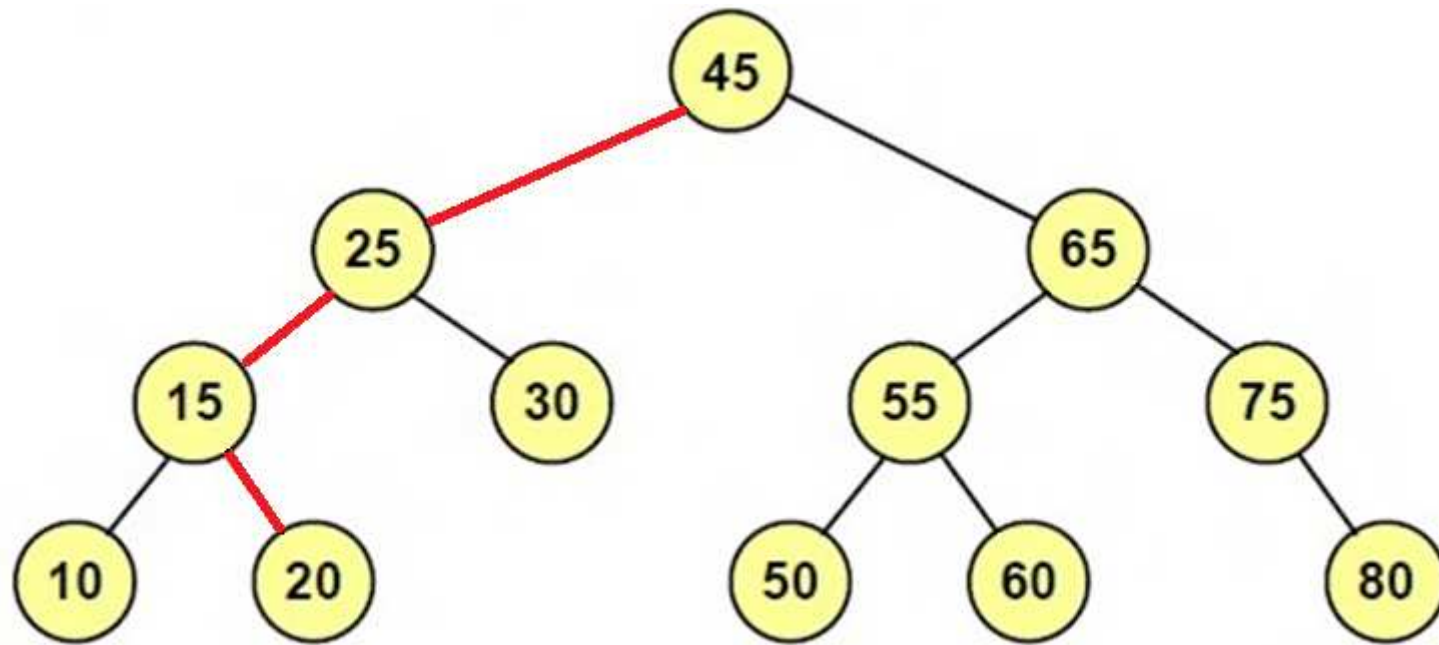
    bool                insertNode(BSTNode<T> *p, T newKey);
    bool                removeNode(BSTNode<T> *p, T aKey);
    int                 countNode(BSTNode<T> *p);
    int                 height(BSTNode<T> *p);
    void                LNR(BSTNode<T> *p);
    void                NLR(BSTNode<T> *p);
    void                LRN(BSTNode<T> *p);

public:
    BINARY_SEARCH_TREE();                    // default constructor
    BINARY_SEARCH_TREE(const BINARY_SEARCH_TREE &aTree); // copy constructor
    ~BINARY_SEARCH_TREE();                   // destructor

    // operations
    bool                insert(T newKey);    // add new node with 'newKey'
    bool                remove(T aKey);      // find and remove node with 'aKey'
    BSTNode<T>*findNode(T aKey);            // find node with 'aKey'
    bool                isEmpty();
    int                 countNode();         // call countNode(root)
    int                 height();           // call height(root)
    void                preorder();         // call NLR(root)
    void                inorder();          // call LNR(root)
    void                postorder();        // call LRN(root)
}; // end class
```



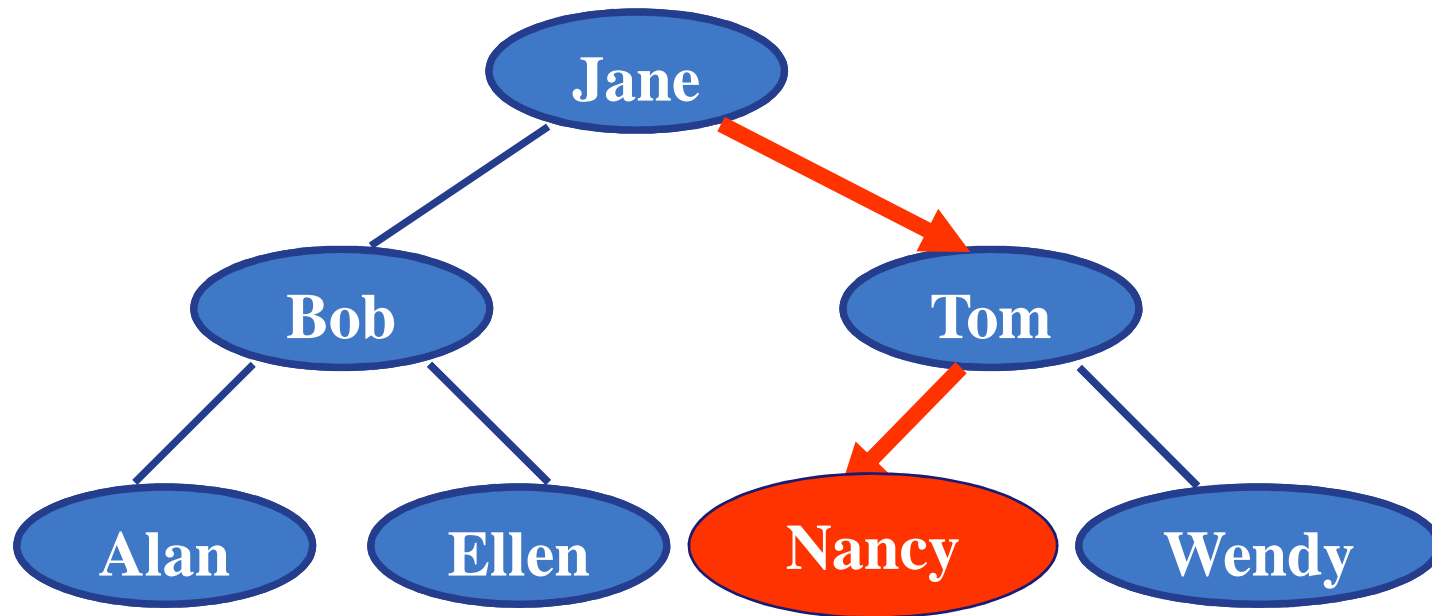
Tìm một node (1)



Tìm key = 20



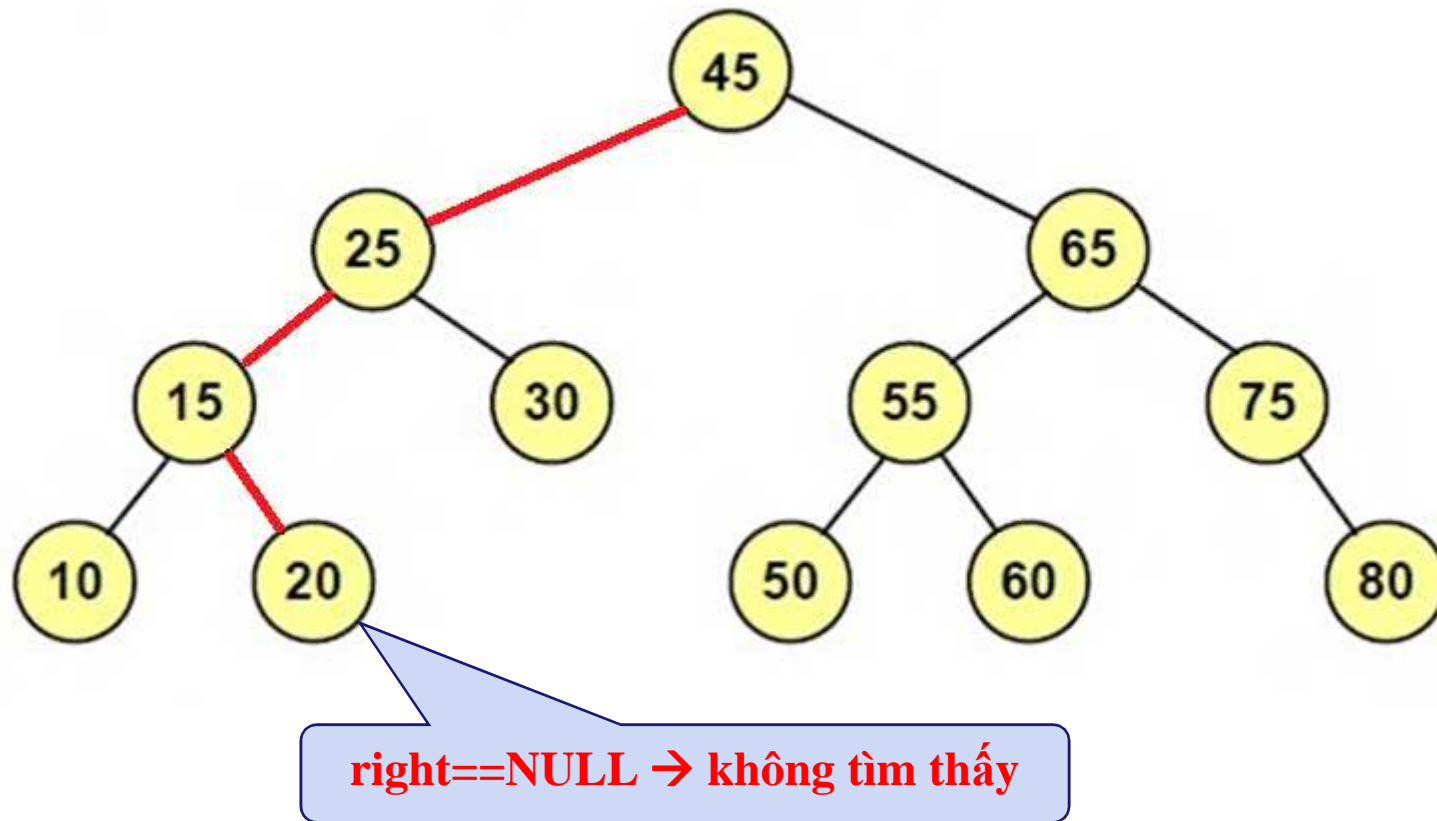
Tìm một node (2)



Tìm key = "Nancy"



Tìm một node (3)



Tìm key = 21 → not found !



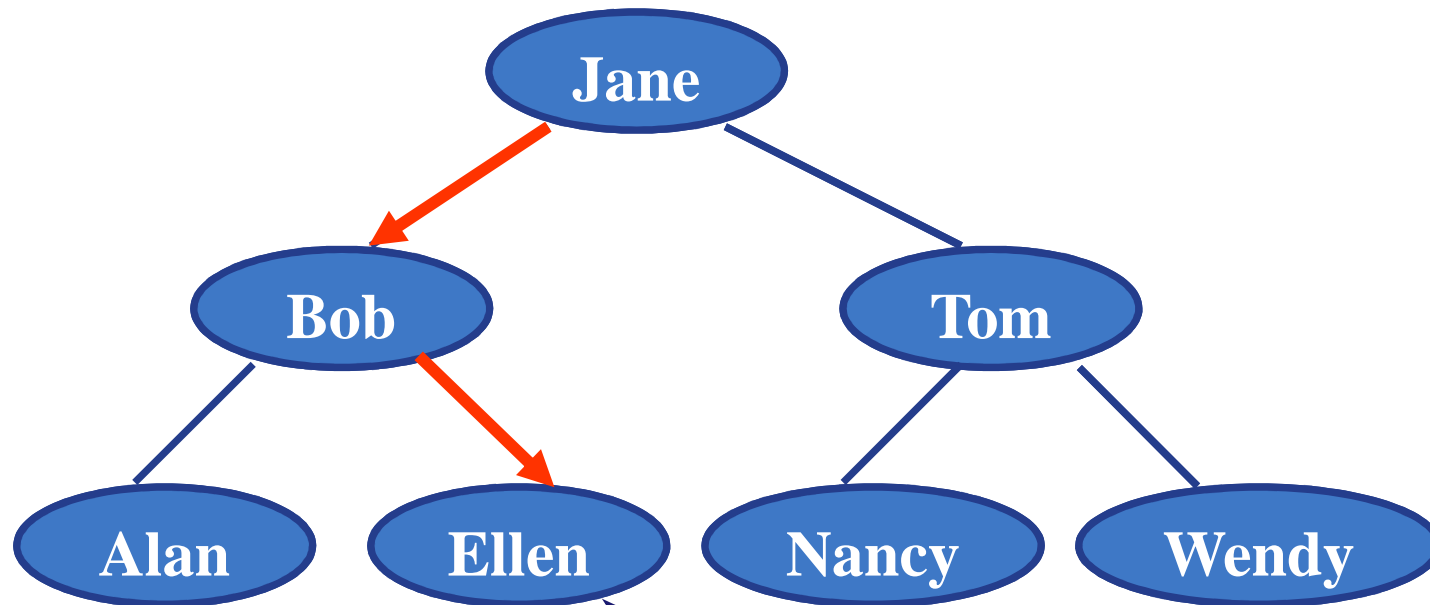
Tìm một node (4)

```
template <class T>
BSTNode<T>* BINARY_SEARCH_TREE<T>::findNode(T aKey)
{
    if (root==NULL) return NULL;
    BSTNode<T> *p = root;
    while (p) {
        if (p->key==aKey) return p; // Tìm thấy
        else if (p->key > aKey)
            p = p->left;           // Tìm nhánh trái
        else
            p = p->right;          // Tìm nhánh phải
    }
    return NULL; // Không tìm thấy
}
```



Thêm một node (1)

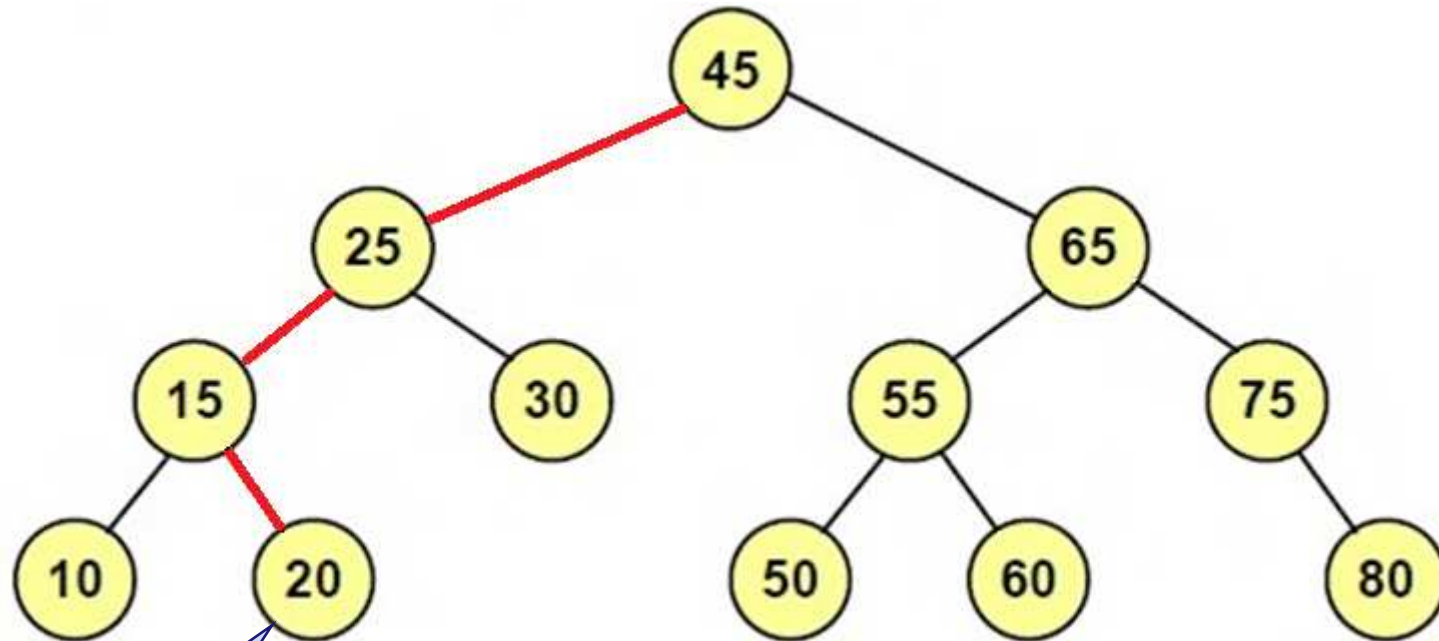
Thêm key = "Frank"



**right==NULL → ngừng tìm kiếm
→ thêm node mới ở đây**



Thêm một node (2)

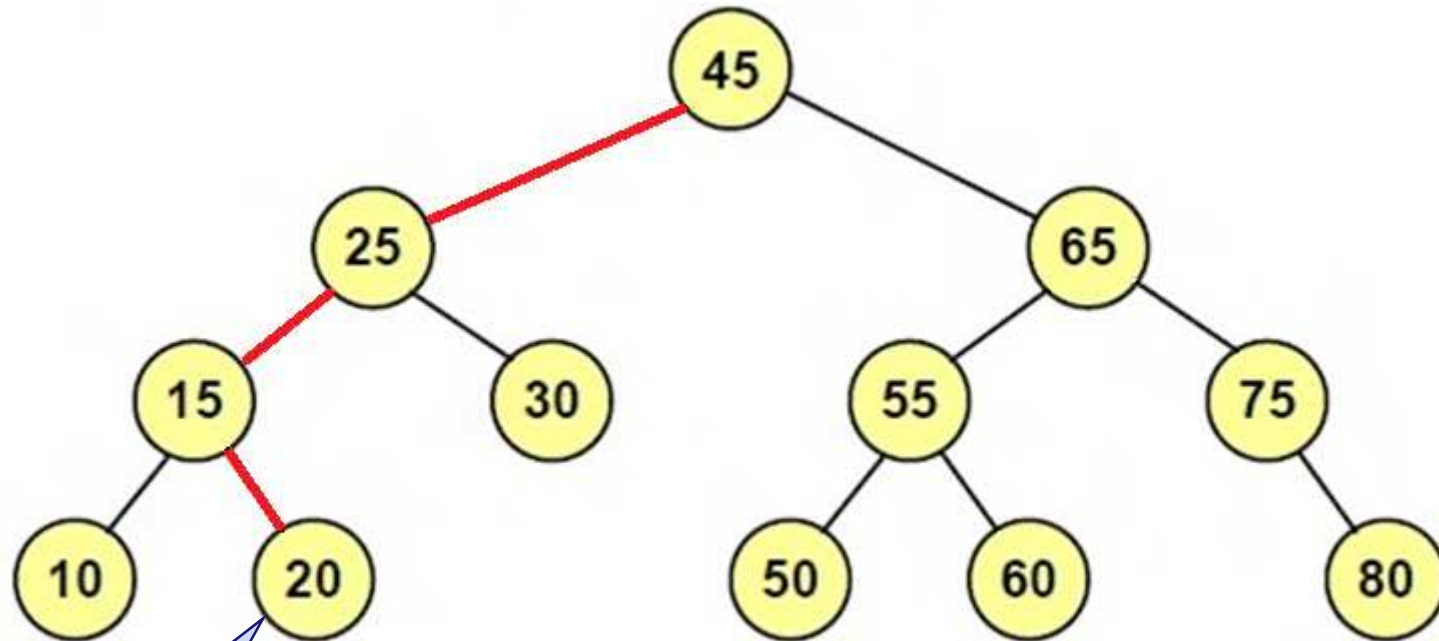


**left==NULL →
ngừng tìm kiếm
→ thêm node
mới ở đây**

Thêm key = 18



Thêm một node (3)



key đã tồn tại →
không thêm nữa

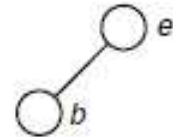
Thêm key = 20



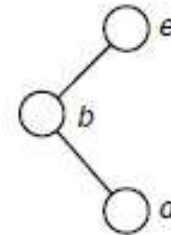
Thêm một node (4)



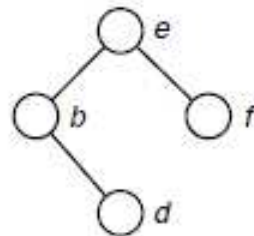
(a) Insert *e*



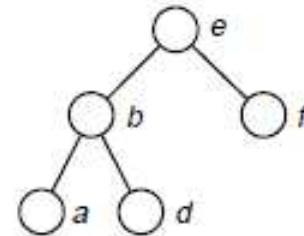
(b) Insert *b*



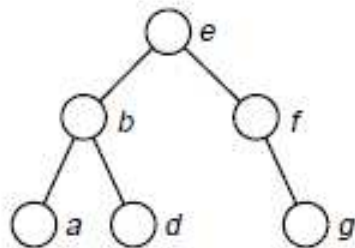
(c) Insert *d*



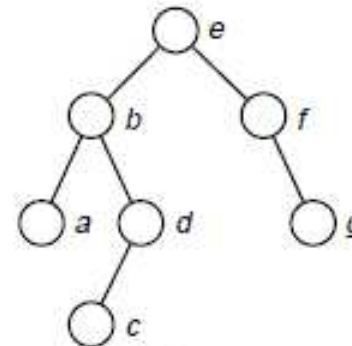
(d) Insert *f*



(e) Insert *a*



(f) Insert *g*



(g) Insert *c*

Thêm các key: e,b,d,f,a,g,c

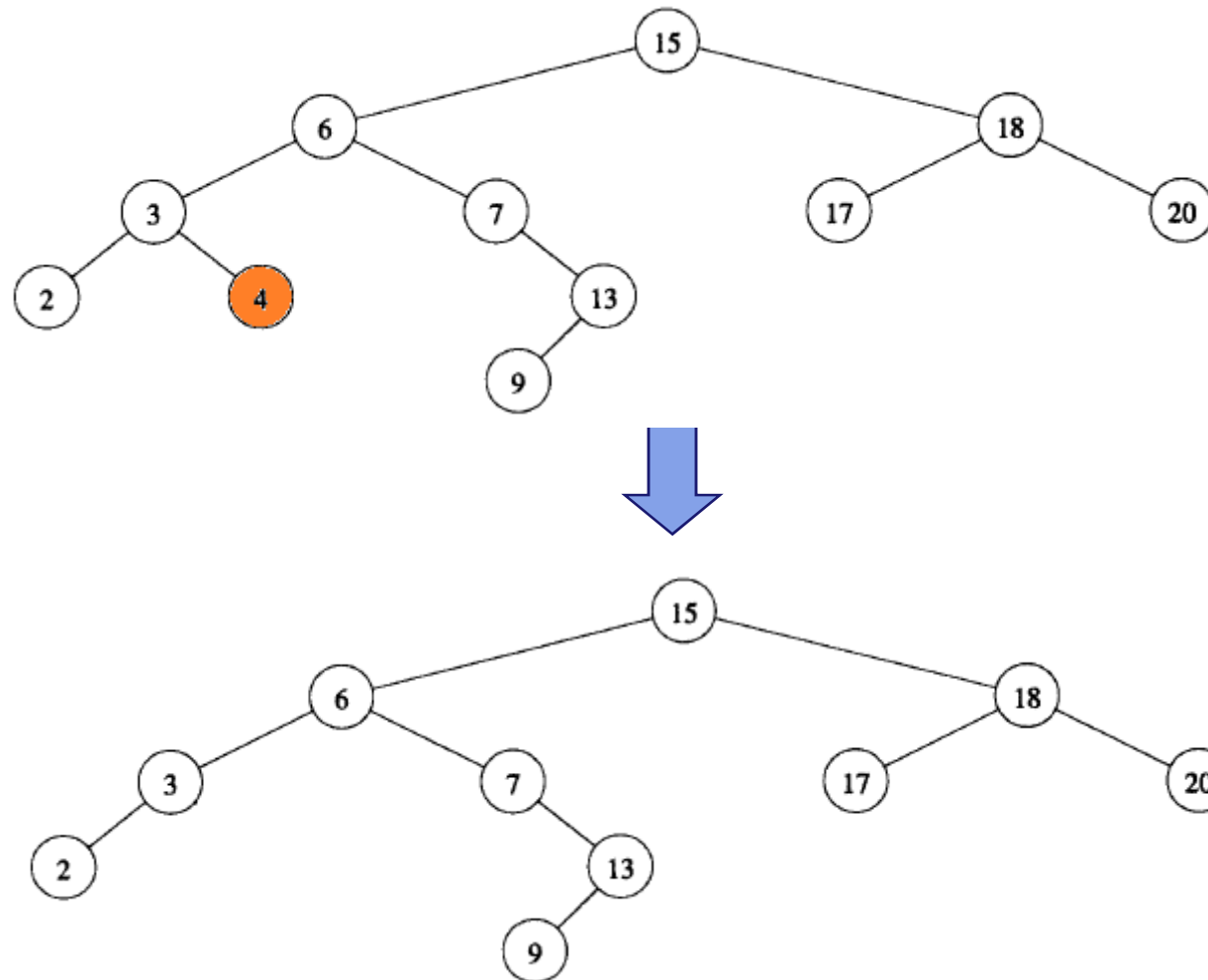


Xóa một node (1)

- Các trường hợp xảy ra:
 - Xóa node lá
 - Xóa node chỉ có 1 cây con
 - Xóa node có 2 cây con



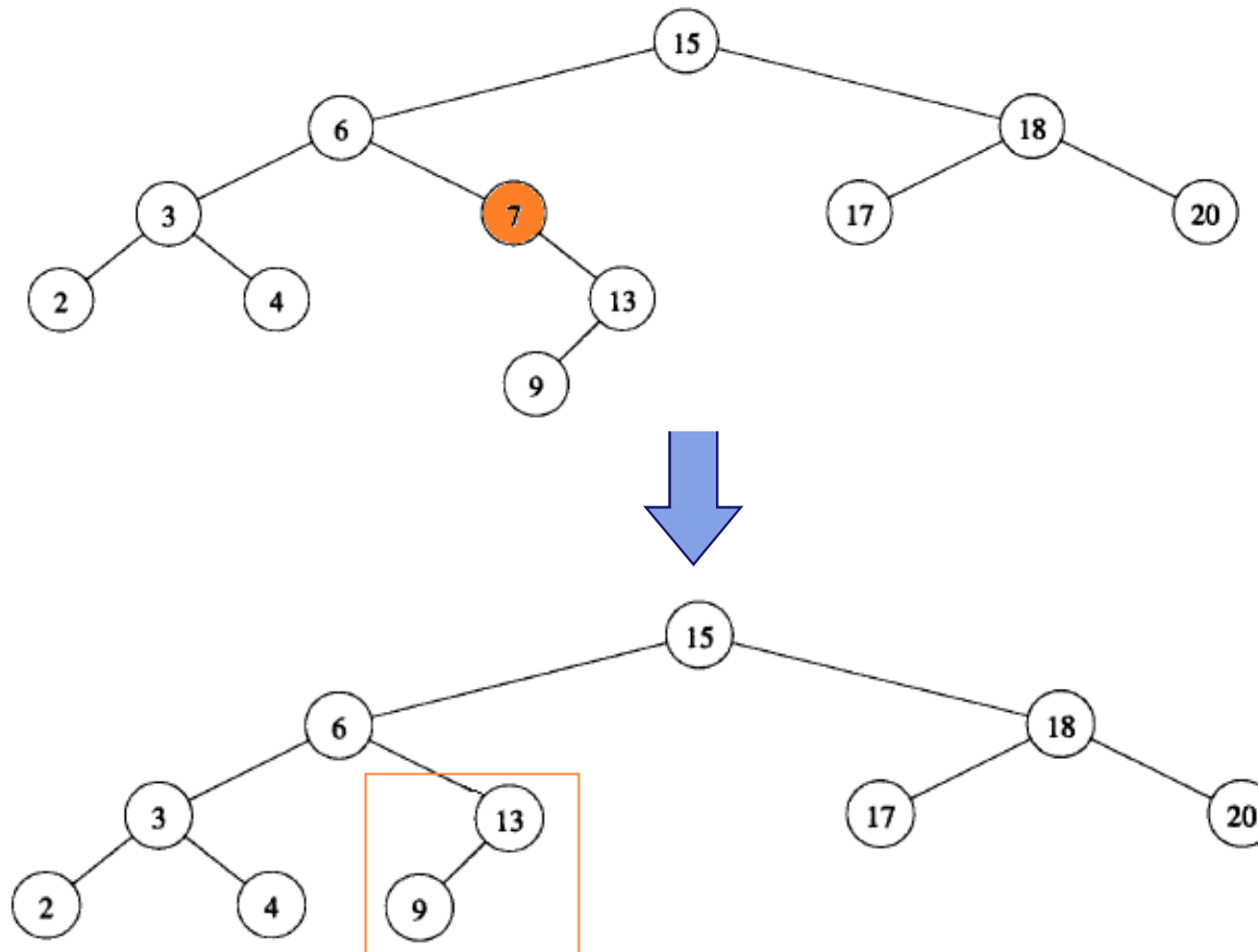
Xóa một node (2)



Xóa key = 4 (node lá)



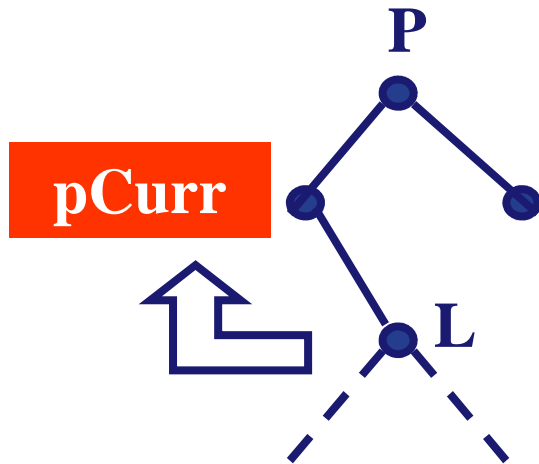
Xóa một node (3)



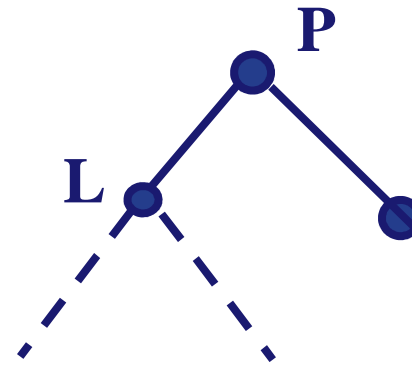
Xóa key = 7 (chỉ có 1 cây con phải)

Xóa một node (4)

- Xoá 1 node chỉ có cây con phải:



Trước khi xóa pCurr

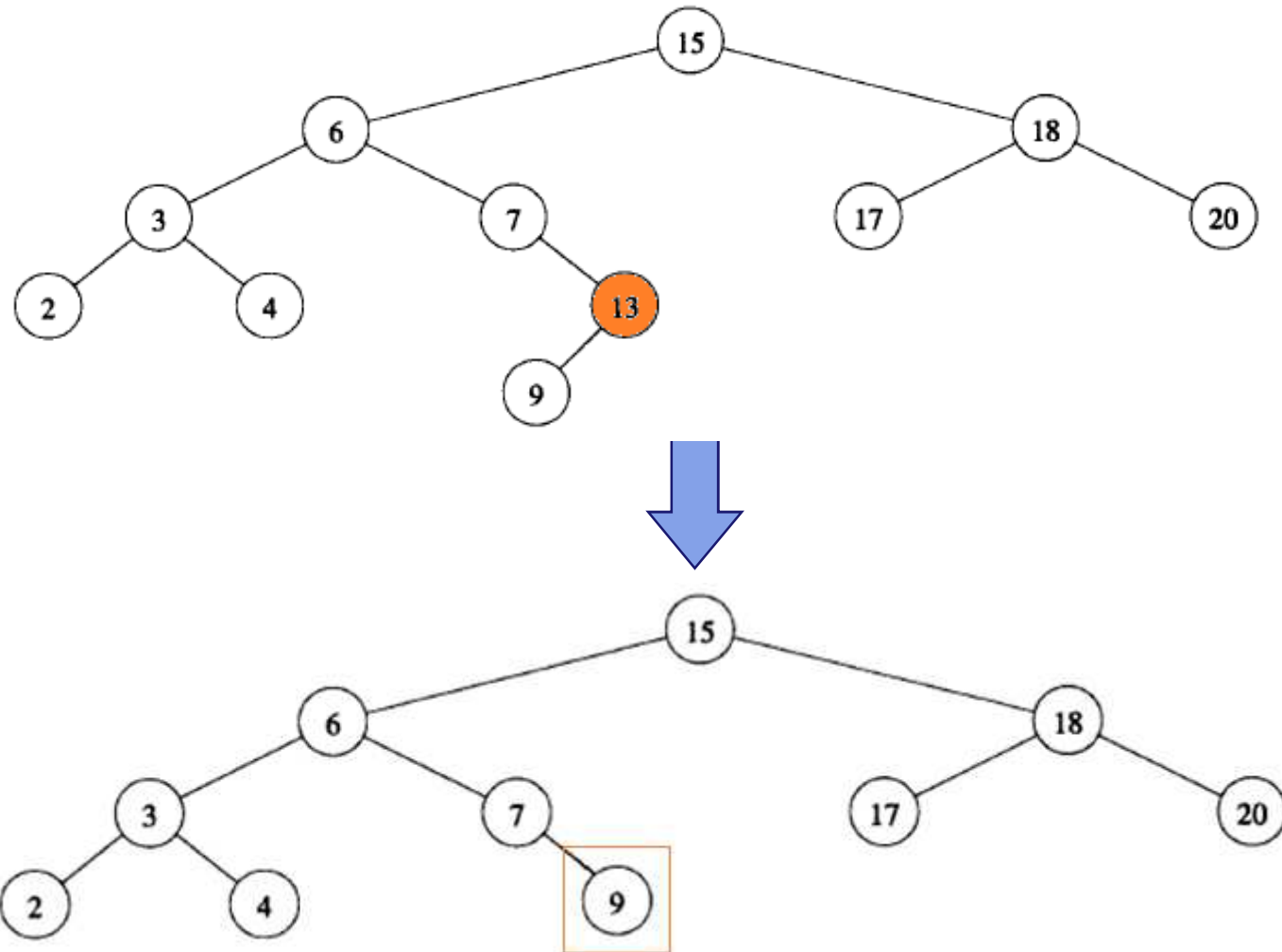


Sau khi xóa pCurr

```
P->left = pCurr->right;  
delete pCurr;
```



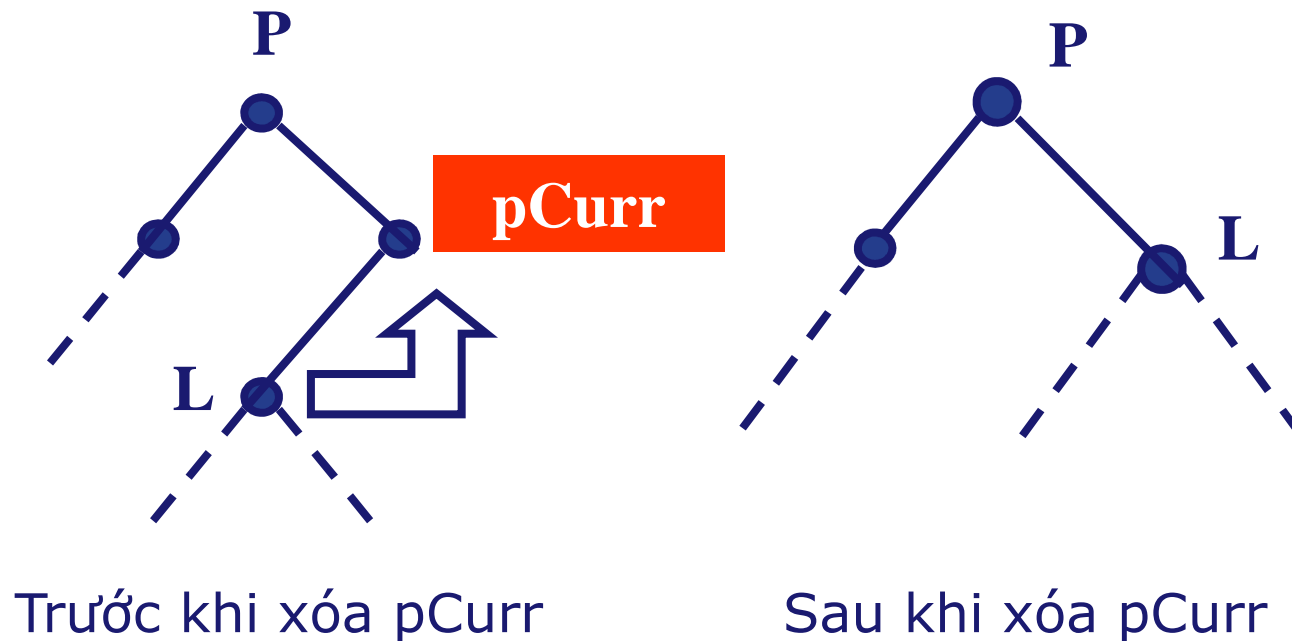
Xóa một node (5)



Xóa key = 13 (chỉ có 1 cây con trái)

Xóa một node (6)

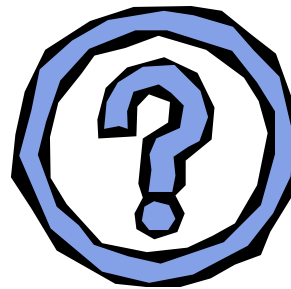
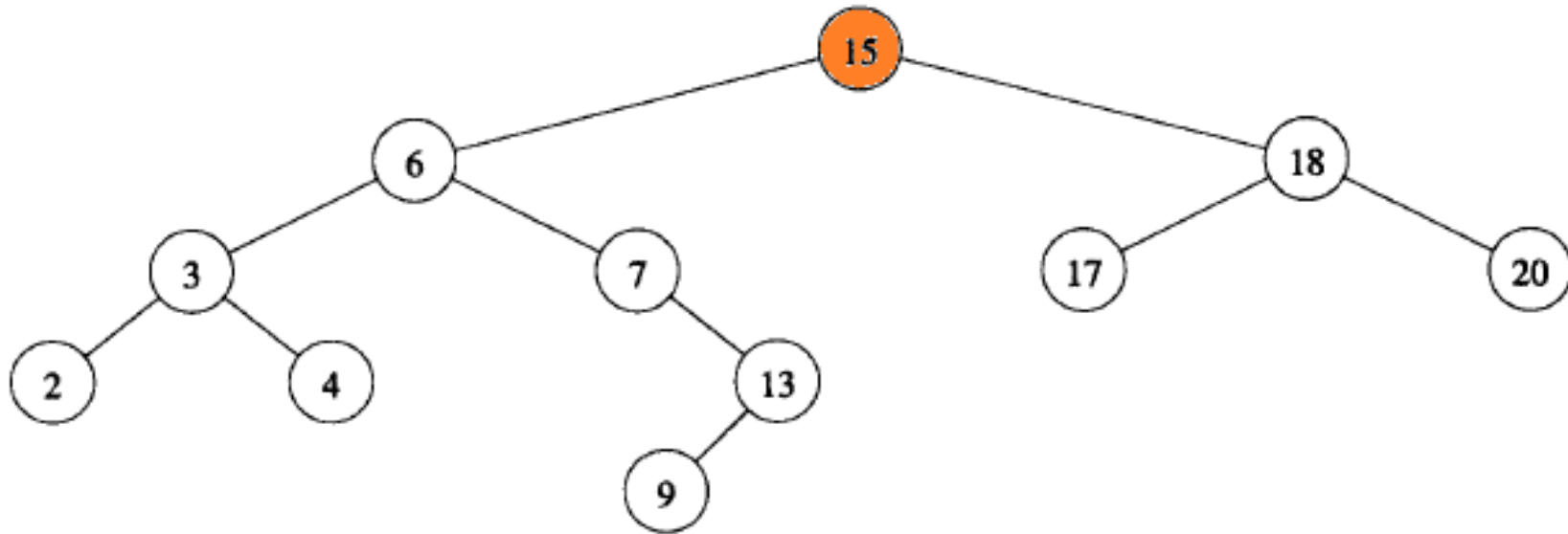
- Xóa 1 node chỉ có cây con trái:



```
P->right = pCurr->left;  
delete pCurr;
```




Xóa một node (7)



Xóa key = 15 (có 2 cây con)

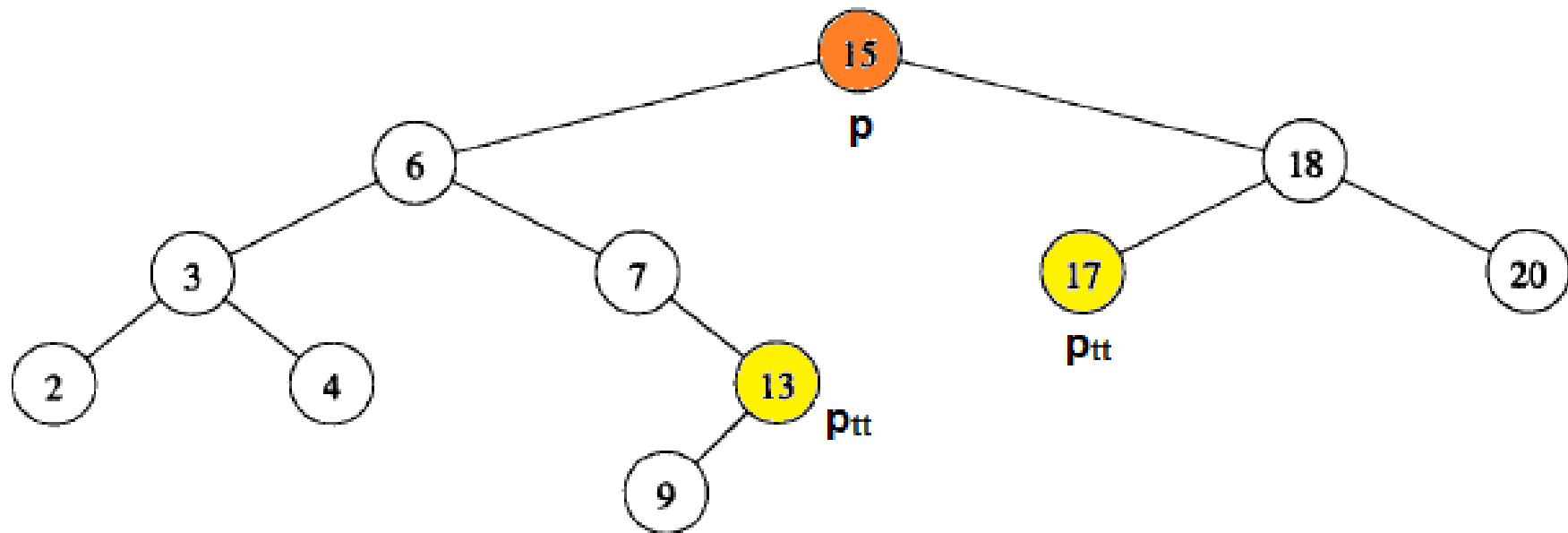


Xóa một node (8)

- Xóa node p có 2 cây con:
 - Thay vì xóa trực tiếp node p , ta (i) tìm 1 phần tử thay thế cho p (gọi là phần tử p_{tt}), (ii) copy nội dung của p_{tt} sang p , (iii) xóa node p_{tt}
 - Phần tử thay thế p_{tt} :
 - Cách 1: là phần tử lớn nhất trong cây con bên trái p
 - Cách 2: là phần tử nhỏ nhất trong cây con bên phải p
- phần tử p_{tt} sẽ có tối đa 1 cây con



Xóa một node (9)



Hai cách chọn phần tử thay thế cho p



Đánh giá/So sánh (1)

- So sánh cây BST với mảng được sắp thứ tự và Danh sách liên kết ?



Đánh giá/So sánh (2)

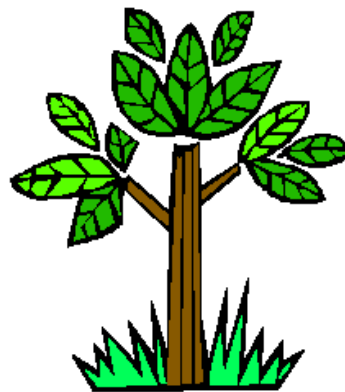
Tiêu chí	Cây BST (*)	Mảng sắp thứ tự	Danh sách liên kết
Chi phí tìm kiếm	$O(\log_2 n)$	$O(\log_2 n)$	$O(n)$
Chi phí thêm phần tử	$O(\log_2 n)$	$O(n)$	$O(1)$
Chi phí xóa phần tử	$O(\log_2 n)$	$O(n)$	$O(1)$
Bộ nhớ sử dụng cho 1 phần tử	$\text{Sizeof}(\text{key})+8$	$\text{Sizeof}(\text{key})$	$\text{Sizeof}(\text{key})+4$

(*) Xét khi cây cân bằng



Cây nhị phân

- Các khái niệm và thuật ngữ cơ bản
- Cài đặt cấu trúc dữ liệu
- Duyệt cây
- Cây nhị phân tìm kiếm – Binary Search Tree
- Hàng đợi ưu tiên – Priority Queue





Hàng đợi ưu tiên

- Priority Queue ADT
- Cài đặt cấu trúc dữ liệu



Priority Queue ADT (1)

- Trong một số ứng dụng thực tế, tính chất FIFO của queue nhiều khi không phù hợp
- Các ví dụ:
 - Sắp hàng mua vé: ưu tiên người già, phụ nữ có thai,...
 - Trạm thu phí: ưu tiên xe cứu thương, xe cảnh sát, xe cứu hỏa
 - Thang máy: yêu cầu xảy ra sau có thể được thực hiện trước (nếu cùng hướng trên đường thang di chuyển) → tối ưu hiệu suất
 - Process P_2 của HĐH có thể được thực hiện trước process P_1 vì có vai trò quan trọng hơn
 - ...

→ cần cấu trúc hàng đợi (có độ) ưu tiên



Priority Queue ADT (2)

- Hàng đợi ưu tiên
 - Một tập hợp nhiều phần tử
 - Mỗi phần tử có một “key”, là độ ưu tiên của phần tử đó
- Các thao tác cơ bản:
 - Khởi tạo hàng đợi rỗng
 - Xóa hàng đợi
 - Thêm 1 phần tử vào queue và hiệu chỉnh vị trí (insert)
 - Lấy phần tử nhỏ nhất (hay lớn nhất) và xóa nó (deleteMin)
 - Lấy phần tử nhỏ nhất (hay lớn nhất) nhưng không xóa nó
 - Kiểm tra queue rỗng

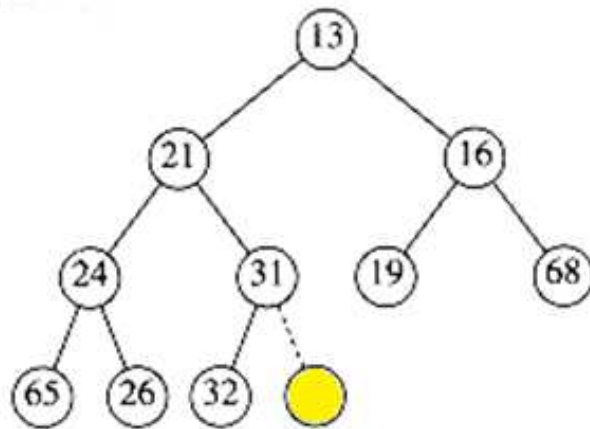


Cài đặt cấu trúc dữ liệu (1)

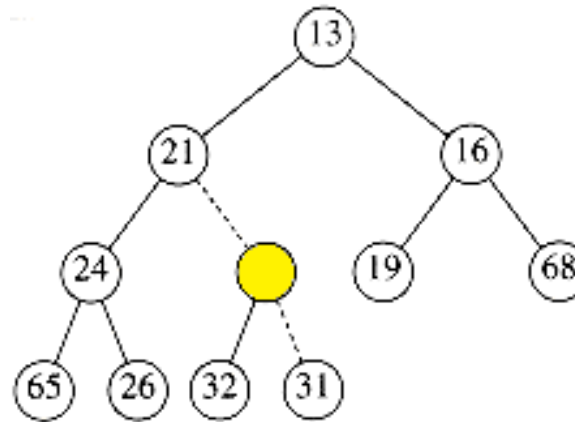
- Sử dụng mảng sắp thứ tự
 - deleteMin: $O(1)$
 - insert: $O(n)$
- Sử dụng BST (*)
 - deleteMin: $O(\log_2 n)$
 - insert: $O(\log_2 n)$
- Sử dụng Heap (min heap/max heap)
 - deleteMin: $O(\log_2 n)$
 - insert: $O(\log_2 n)$



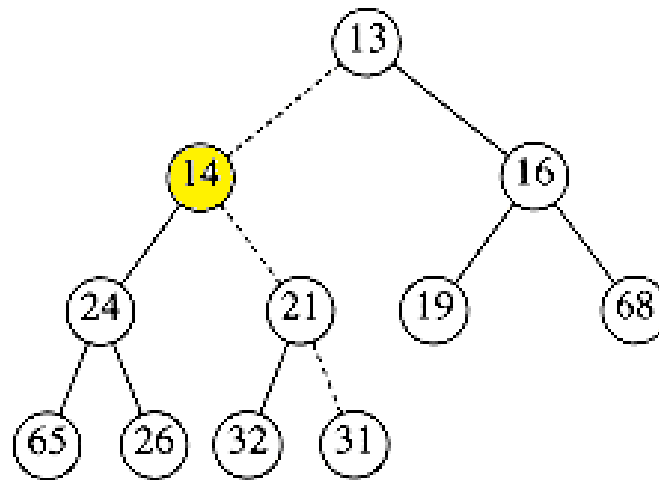
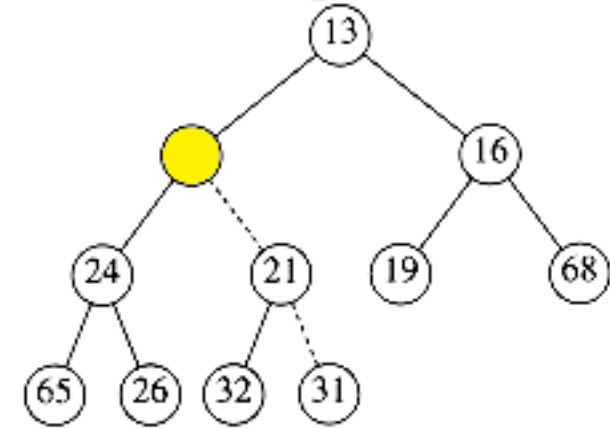
Cài đặt cấu trúc dữ liệu (2)



Bước 1: chèn vào cuối heap



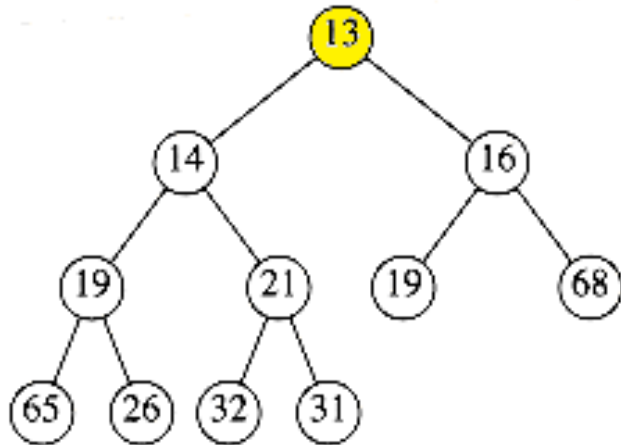
Bước 2: hiệu chỉnh ngược lên trên



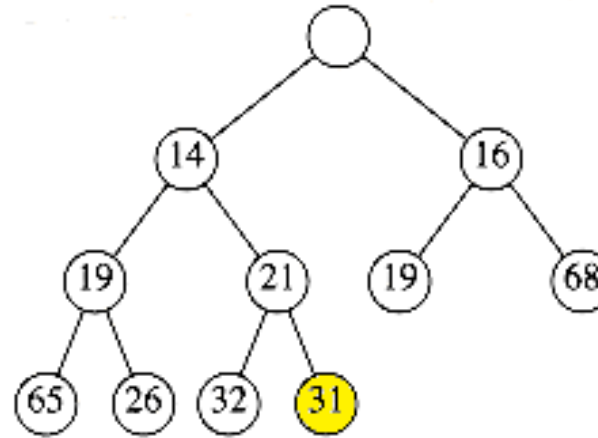
Insert: thêm và hiệu chỉnh vị trí
key=14



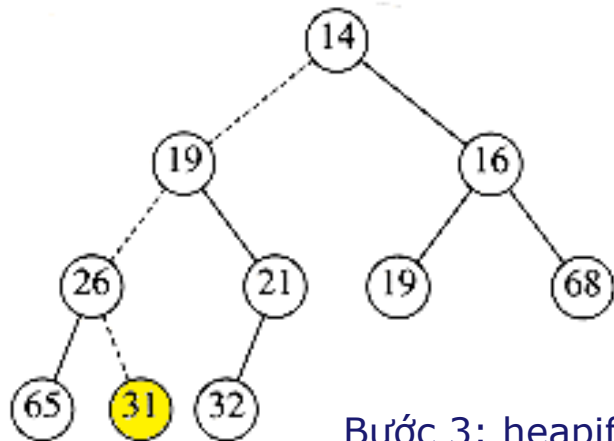
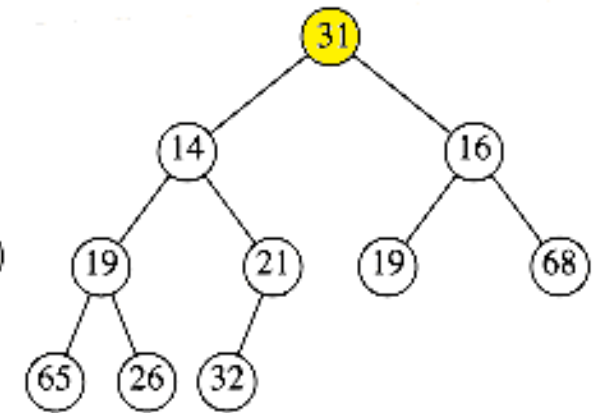
Cài đặt cấu trúc dữ liệu (3)



Bước 1: lấy phần tử ở đầu heap



Bước 2: thay phần tử đầu heap bằng phần tử cuối heap



Bước 3: heapify phần tử đầu

deleteMin: xóa phần tử ở đầu heap và Heapify



Cài đặt cấu trúc dữ liệu (4)

```
template <class T> class PRIORITY_QUEUE {
    private:
        T        *items;           // array of queue items
        int       rear;
        int       maxSize;         // maximum size of queue

        void     heapify(int i);    // adjust heap at position "i"

    public:
        PRIORITY_QUEUE(int size);   // create queue with
                                    // 'size' items
        PRIORITY_QUEUE(const PRIORITY_QUEUE &aQueue);
        ~PRIORITY_QUEUE();          // destructor

        // operations
        bool      isEmpty();
        bool      insert(T newItem);
        bool      deleteMin(T &item);
        bool      minValue(T &item);
}; // end class
```