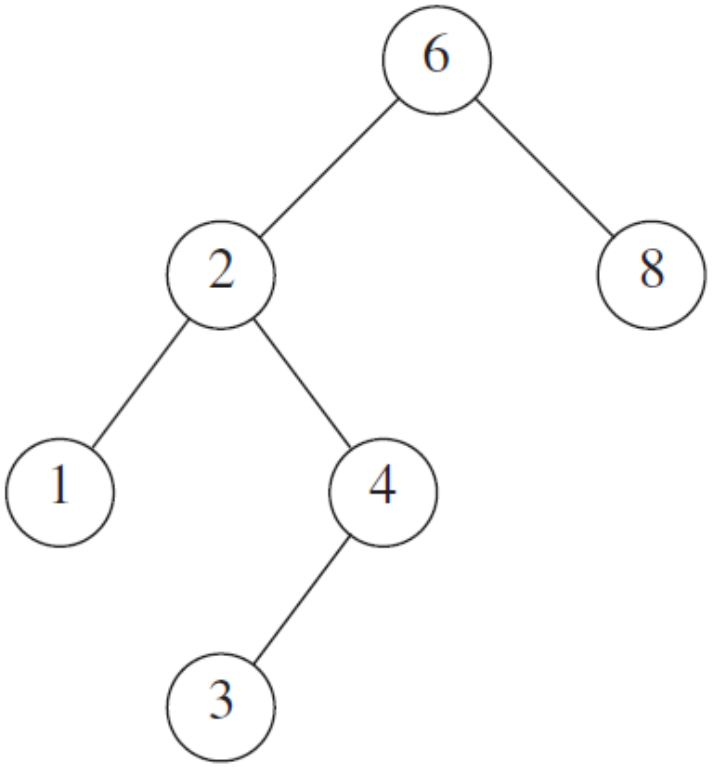
Cây nhị phân tìm kiếm (Binary Search Trees)

Bài giảng môn Cấu trúc dữ liệu và giải thuật

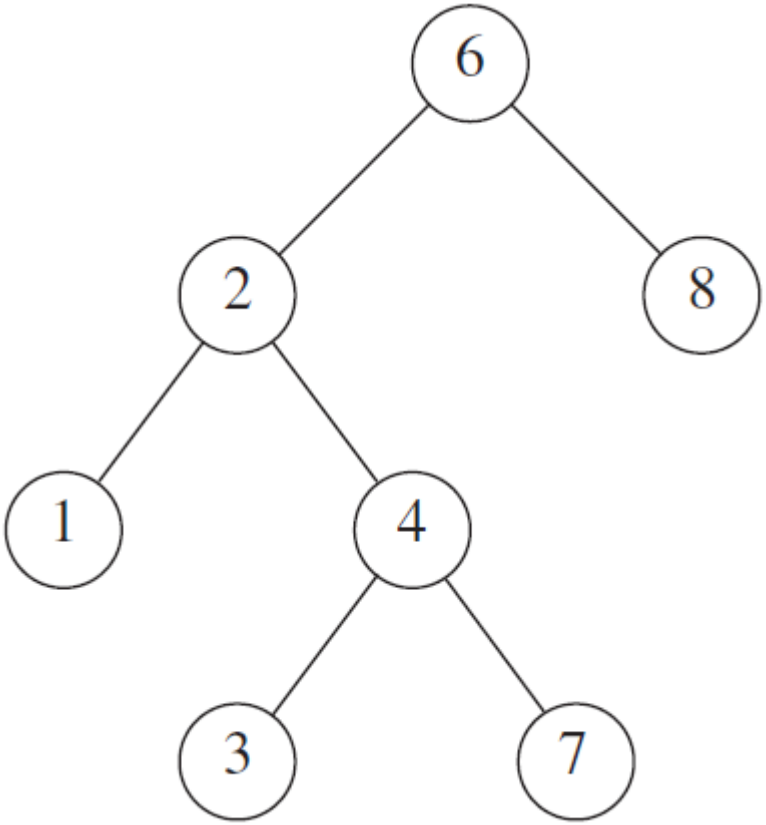
Khoa Công nghệ thông tin
Trường Đại học Thủy Lợi

Định nghĩa

- Giả thiết các giá trị trên cây khác nhau.
- Cây nhị phân tìm kiếm là cây nhị phân, trong đó với mọi nút X:
  - Tất cả các giá trị trên **cây con trái** của X nhỏ hơn X.
  - Tất cả các giá trị trên **cây con phải** của X lớn hơn X.



Đây có phải là cây nhị phân tìm kiếm?



Các thao tác chính

- Tìm phần tử nhỏ nhất
- Tìm phần tử lớn nhất
- Tìm phần tử x
- Chèn phần tử x
- Xóa phần tử x

Tất cả các thao tác trên có thời gian chạy trung bình là O(log N) $\rightarrow$ sẽ chứng minh sau.

Cài đặt

```
// Khai báo kiểu phần tử
typedef int T;

// Định nghĩa kiểu của các nút trên cây
struct BinaryNode {
    T elem;              // Phần tử
    BinaryNode * left;  // Con trỏ tới con trái
    BinaryNode * right; // Con trỏ tới con phải
};

// Định nghĩa cấu trúc cây nhị phân tìm kiếm
struct BinarySearchTree {
    BinaryNode * root; // Con trỏ tới nút gốc
};
```

Hàm khởi tạo, hàm hủy, xóa rỗng

```
void bstInit(BinarySearchTree & tree) {
    tree.root = NULL; // Ban đầu cây rỗng nên root chưa trỏ đi đâu
}

void bstDestroy(BinarySearchTree & tree) {
    bstMakeEmpty(tree); // Xóa hết các nút trên cây
}

// Hàm xóa rỗng cây (gọi hàm cùng tên nhận tham số là gốc của cây
// cần xóa rỗng thay vì bản thân cây đó).
void bstMakeEmpty(BinarySearchTree & tree) {
    bstMakeEmpty(tree.root); // Xem hàm này ở slide tiếp theo
}

// Hàm kiểm tra cây có rỗng hay không
bool bstIsEmpty(BinarySearchTree & tree) {
    return (tree.root == NULL);
}
```

Xóa rỗng cây có gốc t

```
// Hàm xóa rỗng cây, nhận tham số là gốc t của cây cần
// xóa rỗng (hàm được viết theo kiểu đệ quy).
void bstMakeEmpty(BinaryNode * & t)
{
    if (t == NULL)
        return;                 // Thoát ra nếu cây rỗng

    bstMakeEmpty(t->left);  // Xóa rỗng cây con trái
    bstMakeEmpty(t->right); // Xóa rỗng cây con phải
    delete t;               // Xóa nút gốc
    t = NULL; // Cây đã rỗng tức là không tồn tại gốc
}
```

Có dấu & trước t vì sẽ gán giá trị mới cho t trong thân hàm.

Tìm phần tử nhỏ nhất

```
// Hàm chính
T bstFindMin(BinarySearchTree & tree) {
    BinaryNode * v = bstFindMin(tree.root); // Gọi hàm phụ
    return v->elem; // Lấy ra phần tử nhỏ nhất rồi trả về
}

// Hàm phụ/trợ giúp (viết theo kiểu đệ quy).
// Chú ý: Phần tử nhỏ nhất nằm ở nút ngoài cùng bên trái của cây.
BinaryNode * bstFindMin(BinaryNode * t) {
    if (t == NULL)      // Cây rỗng?
        return NULL;

    if(t->left == NULL) // Nút ngoài cùng bên trái?
        return t;

    return bstFindMin(t->left); // Nếu chưa đứng ở nút ngoài
                                // cùng bên trái thì xuống
}                               // con trái để tìm tiếp.
```

Vì sao không cần dấu & trước t trong trường hợp này?

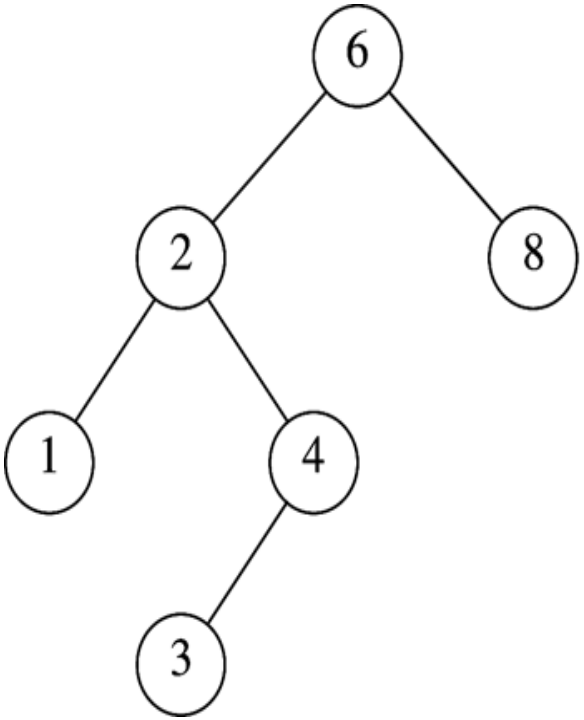
Tìm phần tử lớn nhất

```
// Hàm chính
T bstFindMax(BinarySearchTree & tree) {
    BinaryNode * v = bstFindMax(tree.root); // Gọi hàm phụ
    return v->elem; // Lấy ra phần tử lớn nhất rồi trả về
}

// Hàm phụ/trợ giúp (ở đây dùng vòng lặp thay cho đệ quy).
// Phần tử lớn nhất nằm ở nút ngoài cùng bên phải của cây.
BinaryNode * bstFindMax(BinaryNode * t) {
    if (t != NULL)
        while (t->right != NULL) // Chưa đến tận cùng?
            t = t->right;        // Đi tiếp sang bên phải
    return t;
}
```

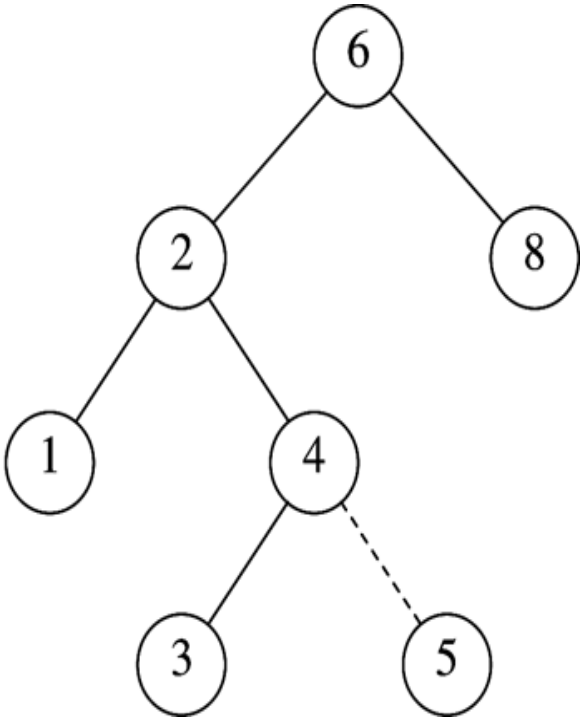
Tìm phần tử x

```
// Hàm chính
bool bstContains(BinarySearchTree & tree, T x) {
    return bstContains(x, tree.root); // Gọi hàm phụ
}

// Hàm phụ/trợ giúp (tìm x trên cây có gốc t)
bool bstContains(T x, BinaryNode * t) {
    if (t == NULL) // Cây rỗng tức là không tìm được
        return false;
    else if (x < t->elem) // Nếu x nhỏ hơn giá trị đang xét thì
        return bstContains(x, t->left);  // tìm x ở bên trái.
    else if (x > t->elem) // Nếu x lớn hơn giá trị đang xét thì
        return bstContains(x, t->right); // tìm x ở bên phải.
    else // Gặp x
        return true;
}
```

Chèn phần tử



Trước khi chèn 5



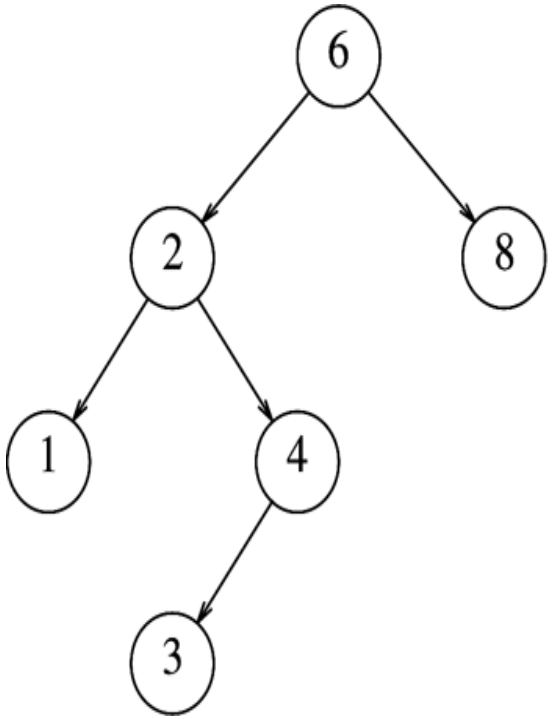
Sau khi chèn 5

Cài đặt thao tác chèn

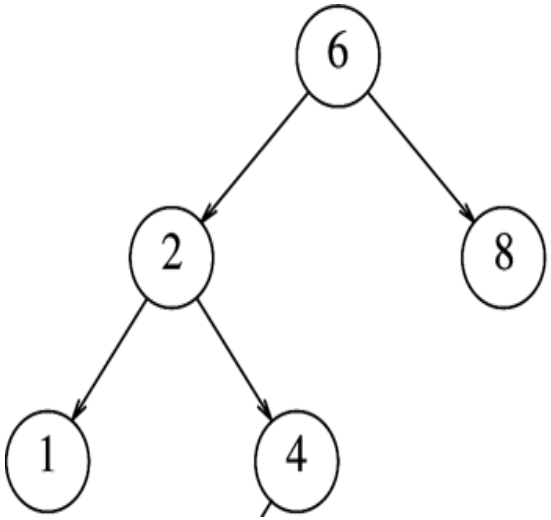
```
// Hàm chính (chèn x vào cây).
void bstInsert(BinarySearchTree & tree, T x) {
    bstInsert(x, tree.root); // Gọi hàm phụ
}

// Hàm phụ/trợ giúp (chèn x vào cây có gốc t).
void bstInsert(T x, BinaryNode * & t) {
    if (t == NULL) { // Cây rỗng thì tạo nút mới
        t = new BinaryNode; // Tạo nút mới
        t->elem = x;        // Nút mới chứa x
        t->left = NULL;     // Nút mới không có con trái
        t->right = NULL;    // Nút mới không có con phải
    }
    else if (x < t->elem)    // Nếu x nhỏ hơn giá trị đang xét
        bstInsert(x, t->left);  // thì chèn x vào cây con trái.
    else if (x > t->elem)    // Nếu x lớn hơn giá trị đang xét
        bstInsert(x, t->right); // thì chèn x vào cây con phải.
    else  // Gặp x, không làm gì cả (phần else này có thể bỏ qua)
        ; // Câu lệnh rỗng
}
```

Có dấu & trước t vì sẽ gán giá trị mới cho t trong thân hàm.

Xóa nút lá



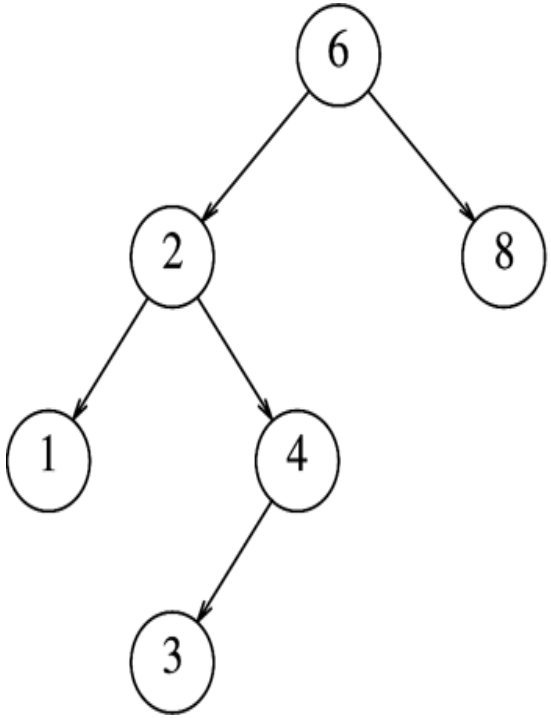
Trước khi xóa 3



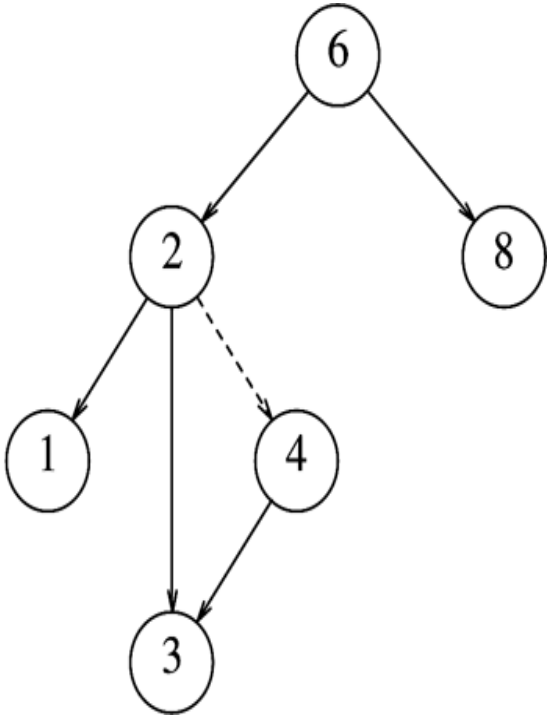
Sau khi xóa 3

Cách xóa: Chỉ đơn giản xóa nút đó.

Xóa nút chỉ có một con



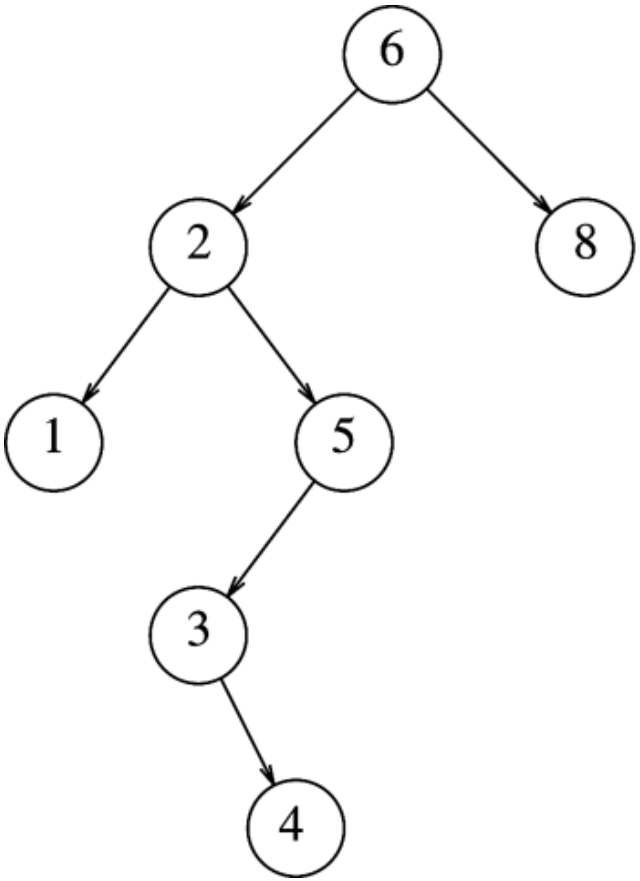
Trước khi xóa 4



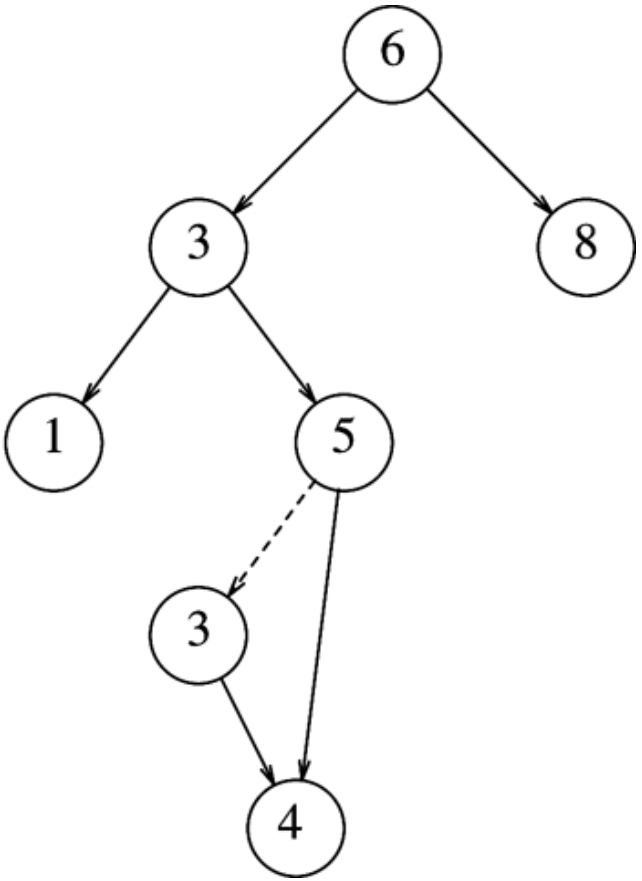
Sau khi xóa 4

Cách xóa: Trước khi xóa, cho nút cha trỏ tới nút con của nút bị xóa.

Xóa nút có hai con



Trước khi xóa 2



Sau khi xóa 2

Cách xóa: Thay nút bị xóa (2) bằng nút nhỏ nhất trên cây con phải (3). Sau đó, xóa nút nhỏ nhất trên cây con phải (là nút lá hoặc nút một con).

Cài đặt thao tác xóa

```
// Hàm chính (xóa x khỏi cây).
void bstRemove(BinarySearchTree & tree, T x) {
    bstRemove(x, tree.root); // Gọi hàm phụ
}

// Hàm phụ/trợ giúp (xóa x khỏi cây có gốc t).
void bstRemove(T x, BinaryNode * & t) {
    ... // Xem code ở slide sau
}
```

Có dấu & trước t vì sẽ gán giá trị mới cho t trong thân hàm.

Cài đặt thao tác xóa (tiếp)

```
void bstRemove(T x, BinaryNode * & t) {
    if (t == NULL)  return;      // Thoát ra nếu cây rỗng.
    if (x < t->elem)             // Nếu x nhỏ hơn giá trị đang xét
        bstRemove(x, t->left);   // thì xóa x ở cây con trái.
    else if (x > t->elem)        // Nếu x lớn hơn giá trị đang xét
        bstRemove(x, t->right); // thì xóa x ở cây con phải.
    else if (t->left != NULL && t->right != NULL) { // Nút 2 con
        t->elem = bstFindMin(t->right)->elem;
        bstRemove(t->elem, t->right);
    }
    else { // Nút 1 con hoặc lá
        BinaryNode * old = t; // Giữ lại địa chỉ của nút cần xóa
        t = (t->left != NULL) ? t->left : t->right;
        delete old; // Xóa nút
    }
}
```

Tìm min trên cây con phải rồi đặt vào nút cần xóa.

Xóa nút min trên cây con phải.

Xác định con duy nhất (có thể là NULL trong trường hợp nút lá) là con trái hay con phải.

Chú ý: t chính là liên kết từ nút cần xóa tới nút con của nó.

Phân tích thời gian chạy

- Gọi n là tổng số nút trên cây.
- Gọi d là độ sâu trung bình của các nút.
- Thao tác xóa rỗng cây có thời gian chạy là O(n), vì có bao nhiêu nút thì sẽ phải xóa một nút bấy nhiêu lần.
- Các thao tác tìm/chèn/xóa có thời gian chạy trung bình là O(d), vì sẽ diễn ra hai bước sau đây:
  1. Đi từ nút gốc tới nút v, nơi diễn ra thao tác, mất thời trung bình là O(d).
  2. Xử lý tại nút v chỉ mất vài thao tác cơ bản, tức là O(1).
- Tiếp theo, **ta sẽ chứng minh d = O(log n)**, và vì vậy thời gian tìm/chèn/xóa trung bình là O(log n).

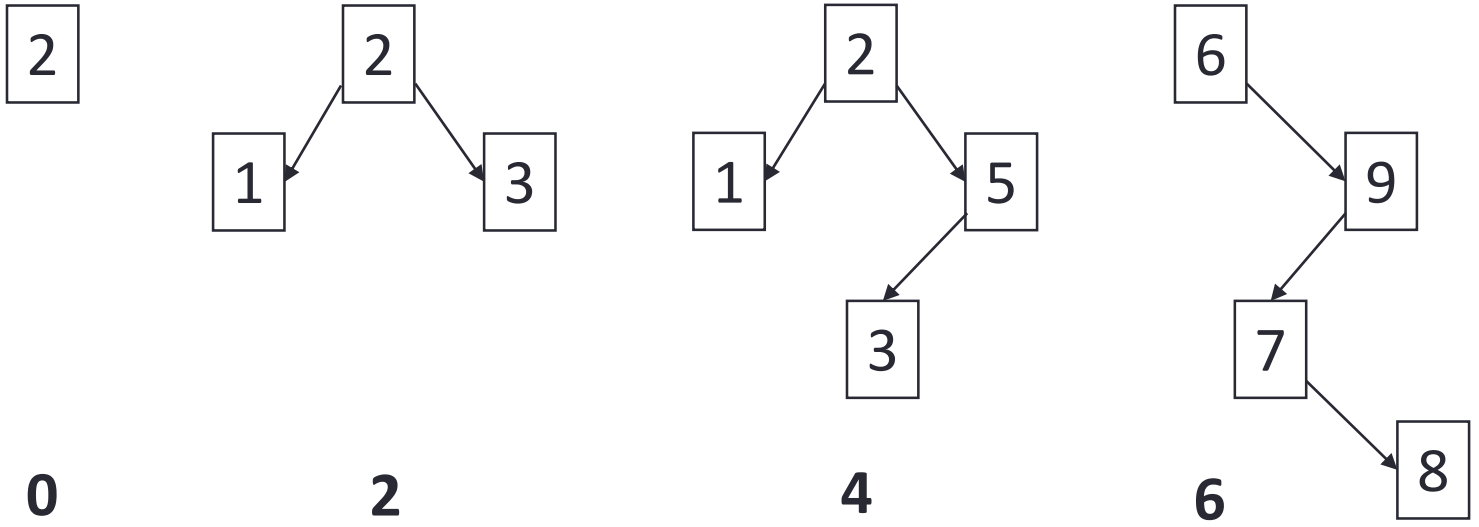
Chứng minh d = O(log n) (1)

- Độ sâu trung bình của các nút:

  d = Tổng-độ-sâu-của-các-nút / Số-nút = D/n

  Tổng độ sâu của các nút (D) được gọi là **độ dài đường đi bên trong**.

- Hãy tính độ dài đường đi bên trong của các cây sau:

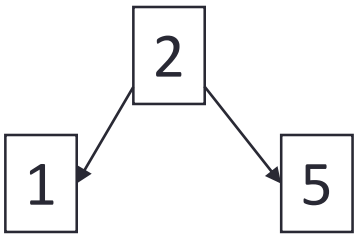


Chứng minh d = O(log n) (2)

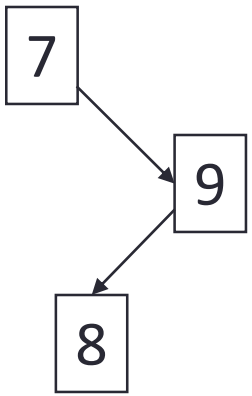
- Nếu cây con trái của nút gốc có i nút:

  $$D(n) = D(i) + D(n-i-1) + n-1$$

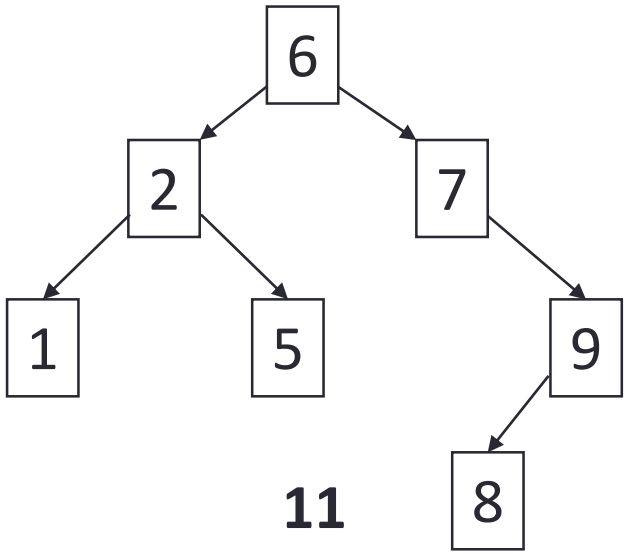
  - D(i) là độ dài đường đi bên trong của cây con trái.
  - D(n-i-1) là độ dài đường đi bên trong của cây con phải.
  - Độ dài đường đi của mỗi nút trong cả hai cây con được cộng thêm 1 khi tính từ nút gốc của toàn cây.



2



3



11

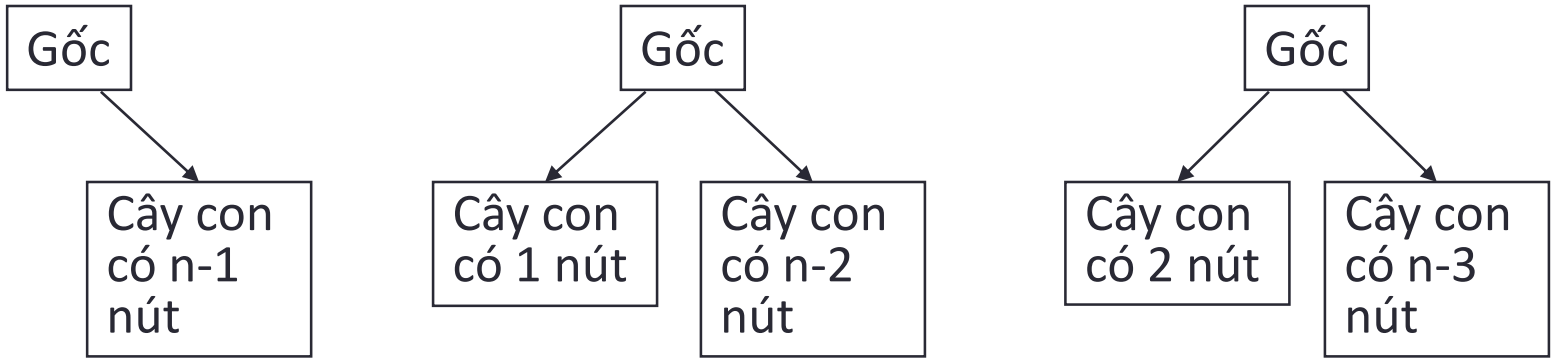
Chứng minh d = O(log n) (3)

Tính giá trị trung bình của D(n):

$D(1) = 0$

$D(n) = 1/n \sum_{i=0}^{n-1} [D(i) + D(n-i-1)] + n-1$

$= 2/n \sum_{i=0}^{n-1} D(i) + n-1$



Chứng minh d = O(log n) (4)

$D(n) = 2/n \sum_{i=0}^{n-1} D(i) + n-1$

$nD(n) = 2 \sum_{i=0}^{n-1} D(i) + n(n-1)$ (1)

$(n-1)D(n-1) = 2 \sum_{i=0}^{n-2} D(i) + (n-1)(n-2)$ (2)

Lấy (1) trừ (2) theo từng vế, ta có:

$nD(n) - (n-1)D(n-1) = 2D(n-1) + 2(n-1)$

$nD(n) = (n+1)D(n-1) + 2(n-1)$

$D(n)/(n+1) = D(n-1)/n + 2(n-1)/[n(n+1)]$

$< D(n-1)/n + 2/n$

$D(n)/(n+1) < D(n-1)/n + 2/n$

$D(n-1)/(n) < D(n-2)/(n-1) + 2/(n-1)$

$D(n-2)/(n-1) < D(n-3)/(n-2) + 2/(n-2)$

...

$D(2)/3 < D(1)/2 + 2/2$

Chứng minh d = O(log n) (5)

$D(n)/(n+1) < D(n-1)/n + 2/n$
$D(n-1)/(n) < D(n-2)/(n-1) + 2/(n-1)$
$D(n-2)/(n-1) < D(n-3)/(n-2) + 2/(n-2)$
...
$D(2)/3 < D(1)/2 + 2/2$

$$D(n)/(n+1) < D(n-1)/n + 2/n$$
$$< D(n-2)/(n-1) + 2/(n-1) + 2/n$$
$$< D(n-3)/(n-2) + 2/(n-2) + 2/(n-1) + 2/n$$
$$...$$
$$< D(1)/(2) + 2/2 + ... + 2/(n-2) + 2/(n-1) + 2/n$$
$$= 2 \sum_{i=2}^{n} 1/i$$

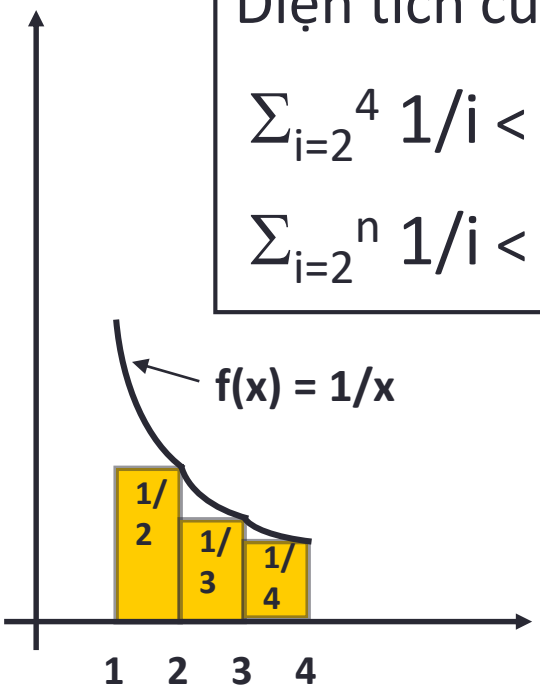
Nếu ta chứng minh được $\sum_{i=2}^{N} 1/i = O(\log n)$ thì sẽ suy ra độ sâu trung bình của một nút **$d = D(n)/n \approx D(n)/(n+1) = O(\log n)$**.

Chứng minh d = O(log n) (6)

Diện tích của các hình chữ nhật < diện tích dưới 1/x

$$\sum_{i=2}^{4} 1/i < \int_{1}^{4} 1/x \, dx$$

$$\sum_{i=2}^{n} 1/i < \int_{1}^{n} 1/x \, dx = \ln n - \ln 1 = O(\log n)$$



Bài tập

1. Chèn lần lượt các giá trị sau đây vào cây nhị phân tìm kiếm đang rỗng: 20, 15, 19, 26, 31, 21, 14, 23, 25. Sau đó, xóa nút gốc của cây.

2. Viết hàm nhận vào một cây nhị phân tìm kiếm và hai giá trị k_1 và k_2, trong đó $k_1 \leq k_2$. Hàm sẽ in ra tất cả các giá trị trên cây nằm trong khoảng $[k_1; k_2]$.