Bài 8. Các thuật toán tìm kiếm

8.1. Bài toán tìm kiếm

- Bài toán tìm kiếm có thể phát biểu như sau: Cho một dãy gồm n bản ghi r[0...n-1]. Mỗi bản ghi r[i] tương ứng với một khoá k[i]. Hãy tìm bản ghi có giá trị khoá bằng **X** cho trước.
- X được gọi là **khoá tìm kiếm**.
- Kết quả của quá trình tìm kiếm:
  - Thành cộng: Tìm được bản ghi có khoá bằng X
  - Thất bại: Không tìm được bản ghi nào có khoá bằng X cả

8.1. Bài toán tìm kiếm

- Ta coi **khoá** của một bản ghi là đại diện cho bản ghi đó.
- Kiểu dữ liệu:

```
#define n ...  // Số khoá trong dãy khoá
typedef ... TKey  // Kiểu dữ liệu một khoá
typedef TKey[n+1] TArray;
TArray k;  // Dãy khoá
```

- Bài toán tìm kiếm: tìm chỉ số i của phần tử có giá trị bằng X trong mảng k

8.2. Tìm kiếm tuần tự

- Thuật toán: bắt đầu từ bản ghi đầu tiên, lần lượt **so sánh** khoá tìm kiếm với khoá tương ứng của các bản ghi trong danh sách, cho tới khi **tìm thấy** bản ghi mong muốn hoặc đã **duyệt hết** danh sách mà chưa thấy

```
int SequentialSearch(TKey X) {
   int i = 0;
   while ((i < n)&&(k[i]!=X)) i++;
   if (i >= n) return -1;
   else return i;
}
```

8.2. Tìm kiếm tuần tự

- Trong cài đặt, ta có thể sử dụng kỹ thuật "lính cầm canh" để giảm số thao tác so sánh: thêm phần tử có giá trị = X vào cuối dãy, để đảm bảo tìm kiếm luôn thành công

```
int SequentialSearch(TKey X) {
   k[n] = X;
   int i = 0;
   while (k[i] != X) i++;
   if (i >= n) return -1;
   else return i;
}
```

- Dễ thấy rằng độ phức tạp của thuật toán tìm kiếm tuần tự trong trường hợp tốt nhất là **O(1)**, trong trường hợp xấu nhất là **O(n)** và trong trường hợp trung bình là **O(n)**.

8.3. Tìm kiếm nhị phân

- Trên thực tế, có nhiều trường hợp mà dãy khóa **đã được sắp xếp**, khi đó ta có phương án tìm kiếm tốt hơn.
- Giả sử ta cần tìm khóa X trong đoạn k[l...h], trước hết ta xét khoá nằm giữa dãy k[m] với m = (l + h) / 2;
  - Nếu k[m] = X → tìm kiếm thành công, kết thúc
  - Nếu k[m] < X thì X phải nằm ở đoạn sau (k[m+1..h]. Tiếp tục tìm kiếm trong đoạn con này
  - Nếu k[m] > X thì X phải nằm ở đoạn đầu (k[0..m-1]). Tiếp tục tìm kiếm trong đoạn con này
- Quá trình tìm kiếm sẽ thất bại nếu đến một bước nào đó, đoạn tìm kiếm là rỗng (l > h).

8.3. Tìm kiếm nhị phân – cài đặt

- Cài đặt:

```
int BinarySearch(TKey X) {
    int L, H, m;
    L = 0; H = n-1;
    while (L ≤ H) {
        m = (L + H) / 2;
        if (k[m] == X) return m;
        if (k[m] < X) L = m + 1;
        else H = m - 1;
    }
    return -1;
}
```

- Dễ thấy độ phức tạp tính toán của thuật toán tìm kiếm nhị phân trong trường hợp tốt nhất là **O(1)**, trong trường hợp xấu nhất là **O(logn)** và trong trường hợp trung bình là **O(logn)**.

8.3. Tìm kiếm nhị phân – cải tiến

- Một chút cải tiến: trong phép cài đặt trên, trong vòng lặp ta rẽ 3 nhánh, tức là cần 2 phép so sánh (== X và < X), ta có thể giảm bớt phép so sánh này như sau:

```
int BinarySearch(TKey X) {
    int L, H, m;
    L = 1; H = n-1;
    while (L < H) {
        m = (H - L) / 2 + L;
        if (k[m] < X) L = m + 1;
        else H = m;
    }
    if ((L==H) && (k[L] == X)) return L;
    else return -1;
}
```

8.3. Tìm kiếm nhị phân – ví dụ

- Trò chơi đoán số: có 2 người chơi
  - Người thứ nhất nghĩ trong đầu 1 số tự nhiên X từ 1 đến 100
  - Người thứ 2 cần đoán ra được con số đó bằng cách hỏi người thứ nhất các câu hỏi mà người thứ nhất chỉ trả lời "Đúng" hoặc "Sai"
- Bài toán: người thứ 2 hỏi như thế nào để số câu hỏi là ít nhất. Đáp án:
  - "Số đó có < 50 không?"
  - Nếu "đúng", hỏi tiếp "Số đó có < 25 không?", ngược lại, hỏi tiếp "Số đó có < 75 không?"
  - ... sau $\log_2 100 = 7$ câu hỏi ta sẽ có được kết quả

8.3. Tìm kiếm tuần tự vs tìm kiếm nhị phân

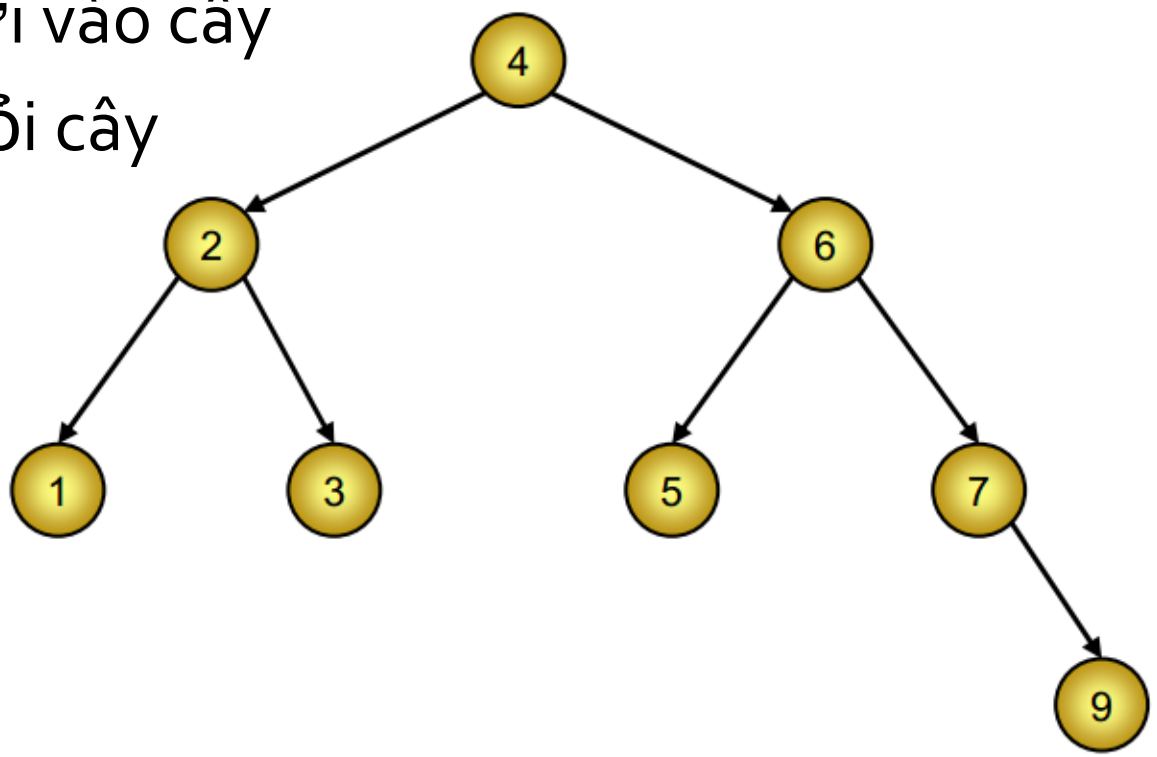
- Tìm kiếm tuần tự
  - Chậm hơn
  - Làm việc được trên dãy bất kỳ (chưa sắp xếp)
  - Làm việc được trên danh sách liên kết (không có thao tác truy cập ngẫu nhiên)
- Tìm kiếm nhị phân
  - Nhanh hơn rất nhiều
  - Danh sách phải được sắp xếp trước khi tìm kiếm
  - Không dùng được cho danh sách truy cập tuần tự (danh sách liên kết)

8.4. Cây nhị phân tìm kiếm (Binary Search Tree - BST)

- Cho n khoá k[0..n-1], cây nhị phân tìm kiếm ứng với dãy khoá đó là một cây nhị phân mà mỗi nút chứa giá trị một khoá trong n khoá đã cho và có tính chất:
  - Mọi khoá nằm trong **cây con trái** của nút đó đều **nhỏ hơn** khoá ứng với nút đó.
  - Mọi khoá nằm trong **cây con phải** của nút đó đều **lớn hơn khoá** ứng với nút đó

8.4. Cây nhị phân tìm kiếm (Binary Search Tree - BST)

- Các thao tác trên cây nhị phân tìm kiếm:
  - Tìm kiếm khóa X trong cây
  - Thêm khóa mới vào cây
  - Loại bỏ nút khỏi cây



8.4.1. Tìm kiếm trên cây nhị phân tìm kiếm

- Thuật toán: khoá tìm kiếm **X** được so sánh với khoá ở **gốc** cây, và 4 tình huống có thể xảy ra:
  - Không có gốc (cây rỗng) – phép tìm kiếm thất bại
  - X trùng với khoá ở gốc – tìm kiếm thành công
  - X **nhỏ hơn** khoá ở gốc, phép tìm kiếm được tiếp tục trong **cây con trái** của gốc với cách làm tương tự
  - X **lớn hơn** khoá ở gốc, phép tìm kiếm được tiếp tục trong **cây con phải** của gốc với cách làm tương tự

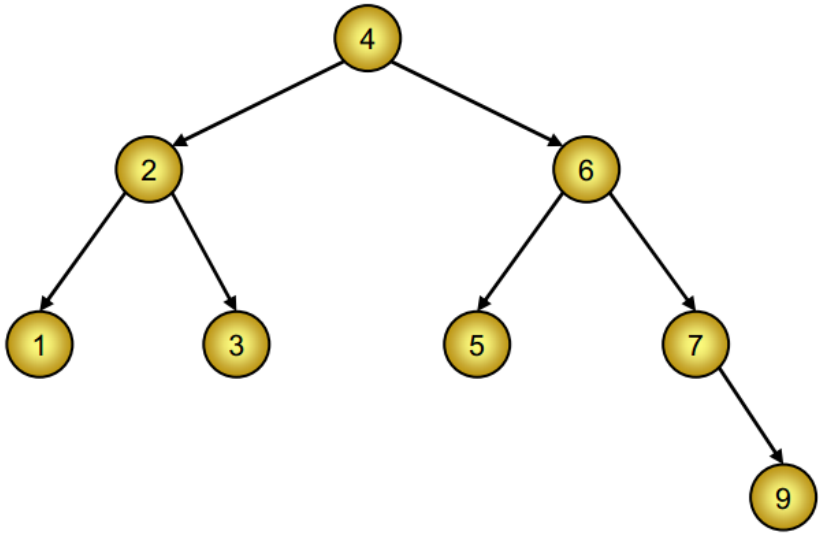
8.4.1. Tìm kiếm trên cây nhị phân tìm kiếm

- Cài đặt:

```
TNode * BSTSearch(TKey X) {
    TNode * p = Root;
    while (p != NULL) {
        if (X == p->Info) break;
        else if (X < p->Info) p=p->Left;
             else p = p->Right;
    }
    return p;
}
```

8.4.2. Dựng cây nhị phân tìm kiếm

- Thuật toán: ta **chèn lần lượt** các khoá vào cây:
  - Nếu khoá đó đã có trong cây → bỏ qua,
  - Nếu khóa chưa có trong cây:
    - Tạo nút mới chứa khoá cần chèn
    - Nối nút đó vào cây nhị phân tìm kiếm tại mối liên kết vừa rẽ sang khiến quá trình tìm kiếm thất bại



8.4.2. Dựng cây nhị phân tìm kiếm

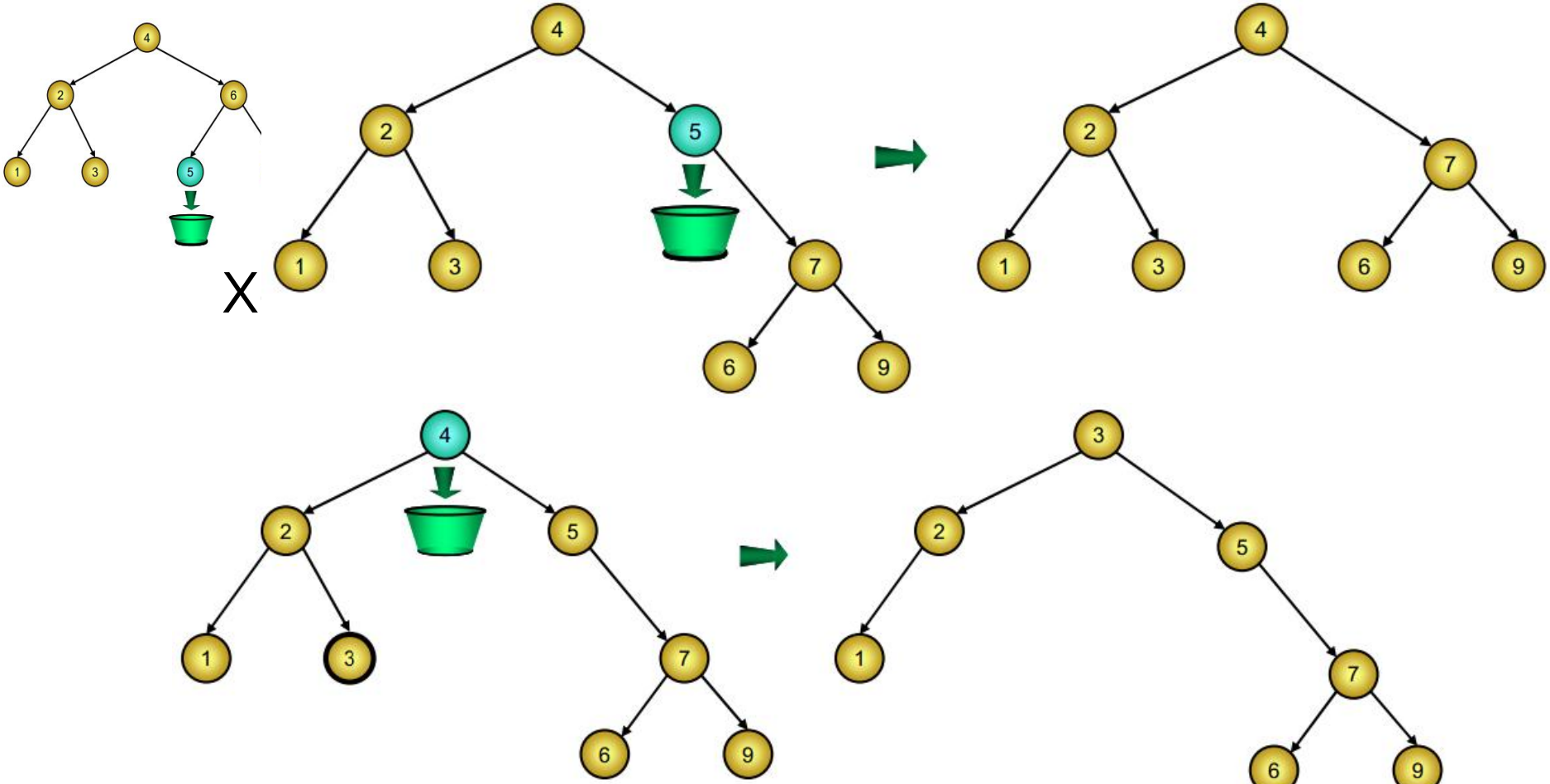
- Cài đặt:

```
void BSTInsert(TKey X) {
    TNode * p = Root;   // Bắt đầu với p = nút gốc
    TNnode * q = NULL;  // q chạy đuổi theo sau p
    while (p != NULL) {
        q := p;
        if (X == p->Info) break;
        else if (X < p->Info) p = p->Left;
             else p = p->Right;
    }
    if (p == NULL) {  // Khoá X chưa có trong BST
        p = <cấp phát nút mới, giá trị X>
        if (Root == NULL) Root = p;
        else  // Móc nút mới p vào nút cha q
            if (X < q->Info) q->Left = p;
            else q->Right = p;
    }
}
```

8.4.3. Xóa nút khỏi cây nhị phân tìm kiếm

- Thuật toán: trước hết tìm nút D chứa khóa X cần xóa
  - Nếu D là nút lá → loại bỏ D khỏi cây bằng cách gán NULL cho liên kết đến nút D
  - Nếu D chỉ có 1 nhánh con → đem nút gốc của nhánh con đó thế vào chỗ của D
  - Nếu D có 2 nhánh con, ta có 2 cách xử lý
    - Hoặc tìm nút chứa khoá **nhỏ nhất** trong **cây con phải**, đưa giá trị chứa trong đó sang nút D, rồi **xoá** nút này (có không quá 1 con)
    - Hoặc tìm nút chứa khoá **lớn nhất** trong **cây con trái**, đưa giá trị chứa trong đó sang nút D, rồi **xoá** nút này (có không quá 1 con)
  - Sau cùng là giải phóng bộ nhớ bị chiếm dụng bởi D

8.4.3. Xóa nút khỏi cây nhị phân tìm kiếm



Xóa nút bậc 2

8.4.3. Xóa nút khỏi cây nhị phân tìm kiếm

- Cài đặt:

```
void BSTDelete(TKey * p) {
    TNode * q, Node, Child;
    q = NULL;
    <duyệt để tìm nút p chứa khóa X, q chạy theo sau là cha của p>
    if (p==NULL) return; // nút cần xóa không tồn tại trong BST
    if ((p->Left != NULL) && (p->Right != NULL) { // p có 2 con
        Node = p; // Giữ lại nút chứa khoá X
        q = p; p = p->Left; // Chuyển sang nhánh con trái để tìm nút cực phải
        while (p->Right != NULL) { q = p; p = p->Right; }
        Node->Info = p->Info; // Chuyển giá trị từ nút tìm được lên Node
    }
    // Nút bị xoá giờ đây là nút p, nó chỉ có nhiều nhất một con trái là Child
    if (p->Left != NULL) Child = p->Left;
    if (p == Root) Root = Child; // Nút p bị xoá là gốc cây
    else // p không phải gốc, lấy mối nối từ cha của nó là q nối thẳng tới Child
        if (q->Left == p) q->Left = Child;
        else q->Right = Child;
    delete(p);
}
```

8.4.4. Cây nhị phân tìm kiếm – nhận xét

- Trường hợp trung bình, thì các thao tác **tìm kiếm**, **chèn**, **xoá** trên BST có độ phức tạp là **O(lgn)**.
- Trường hợp xấu nhất (cây suy biến) - **O(n)**.
- Khắc phục trường hợp suy biến: cây nhị phân cân bằng AVL, cây nhị phân tìm kiếm tối ưu, cây đỏ đen, cây xoay ra ngoài (splay tree), cây tango…
- Có thể mở rộng BST để có thể chứa các khóa bằng nhau
- Khi duyệt BST theo thứ tự giữa (inorder traversal), ta sẽ thu được dãy khóa tăng dần → **Tree sort** với **O(nlogn)**.
- Ưu điểm so với tìm kiếm nhị phân trên mảng đã sắp xếp: thao tác **chèn** và **xóa** trên BST có độ phức tạp trung bình là **O(logn)**, trong khi với mảng sắp thứ tự là **O(n)**.

8.5. Phép băm (Hash)

- Trong một số trường hợp, có thể dễ dàng **thu hẹp phạm vi tìm kiếm** một cách nhanh chóng bằng cách xác định khóa X cần tìm nằm trong một **nhóm nhỏ** các khóa có **đặc tính chung** nào đó.
- Việc chia các khoá thành các nhóm có một đặc điểm chung (không có trong các nhóm khác) như vậy gọi là phép băm.

8.5. Phép băm (Hash)

- Ví dụ:
  - Trong cuốn từ điển Anh-Việt, các bạn sinh viên thường dán vào 26 mảnh giấy nhỏ vào các trang để đánh dấu trang khởi đầu của một đoạn chứa các từ có **cùng chữ cái đầu**. Ta nói cuốn từ điển được "băm" thành 26 nhóm, từ trong mỗi nhóm có cùng chữ cái đầu tiên
  - Trên dãy các khoá số tự nhiên, ta có thể "băm" ra làm ***m*** nhóm, mỗi nhóm gồm các khoá đồng dư theo mô-đun ***m***

8.5. Phép băm (Hash)

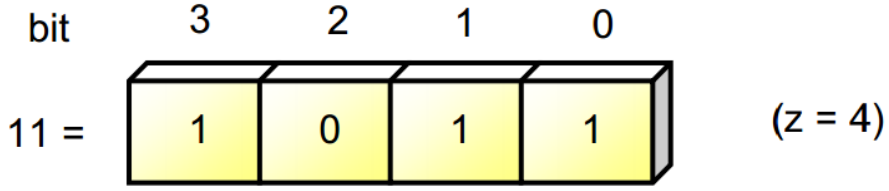
- Có nhiều cách cài đặt phép băm:
  - Chia dãy khoá làm **các đoạn**, mỗi đoạn chứa những khoá thuộc cùng một nhóm và ghi nhận lại vị trí các đoạn đó. Để khi có khoá tìm kiếm, có thể xác định được ngay cần phải tìm khoá đó trong đoạn nào
  - Chia dãy khoá làm **m nhóm**, mỗi nhóm là một **danh sách nối đơn** các giá trị khoá và ghi nhận lại chốt của mỗi danh sách nối đơn. Với một khoá tìm kiếm, ta xác định được phải tìm khoá đó trong danh sách nối đơn nào và tiến hành tìm kiếm tuần tự trên danh sách nối đơn đó.
  - Chia dãy khoá làm **m nhóm**, mỗi nhóm được lưu trữ dưới dạng **cây nhị phân tìm kiếm** và ghi nhận lại gốc của các cây nhị phân tìm kiếm đó

8.6. Khóa số với bài toán tìm kiếm

- Đối với các bài toán tìm kiếm mà khóa là các số, tồn tại những thuật toán rất tốt dựa trên tính chất "số" của khóa
- Trong bài toán tìm kiếm tổng quát, ta có thể mã hóa các khóa thành các số sao cho 2 khóa khác nhau luôn được mã hóa thành 2 số khác nhau, khi đó ta có thể áp dụng các thuật toán nói trên.
- Lưu ý: phương pháp mã hóa này không dùng được với bài toán sắp xếp

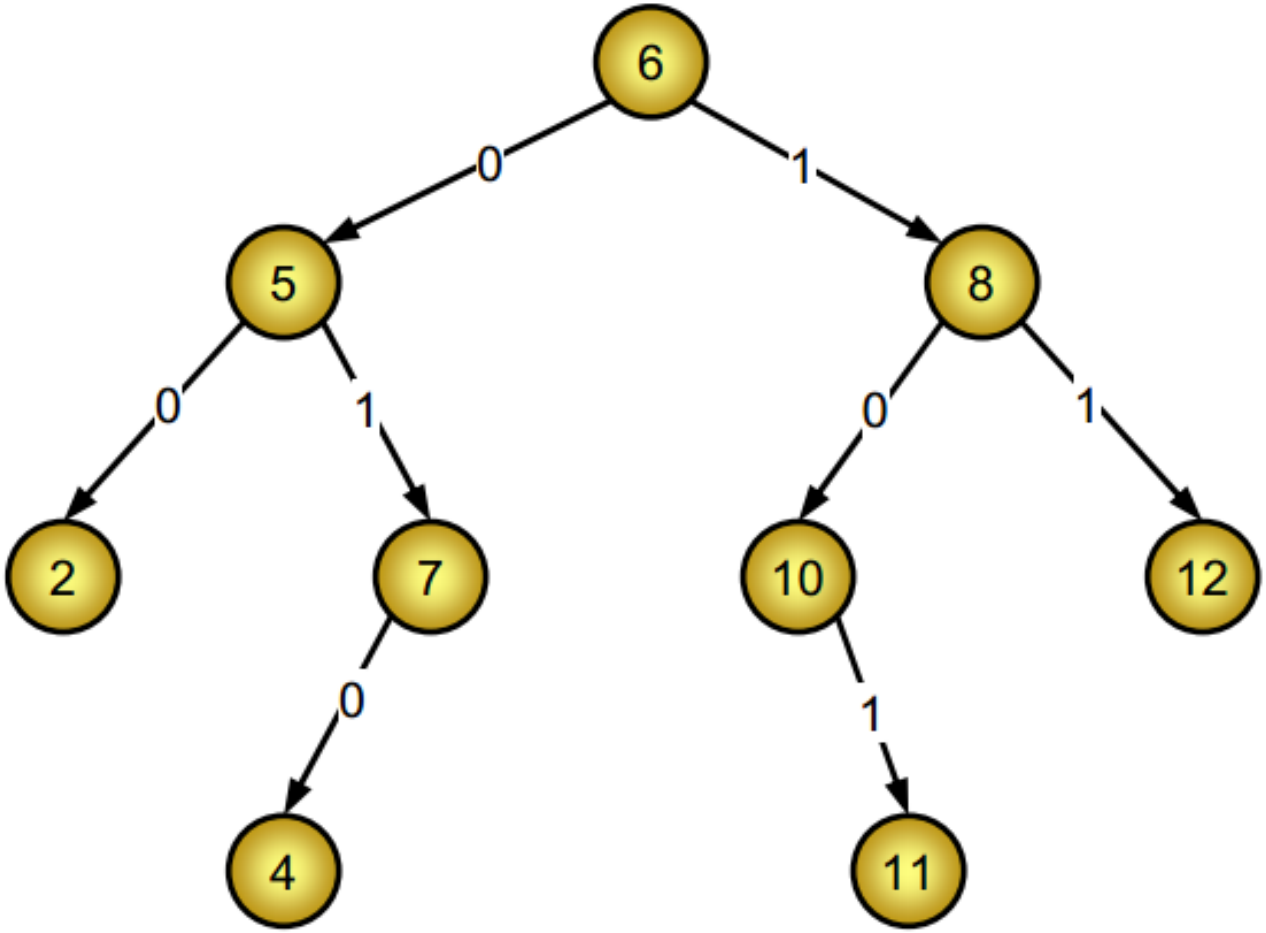
8.7. Cây tìm kiếm số học (Digital Search Tree - DST)

- Mỗi giá trị khoá số X ở hệ **nhị phân** có **z** chữ số (**z** bits), được đánh số từ **0** tới **z - 1** từ phải sang trái:



- Cây tìm kiếm số học:
  - Nút gốc có tối đa 2 cây con
    - Tất cả những khoá có bit cao nhất (**bit z-1**) là **0** nằm trong **cây con trái**,
    - Tất cả những khoá có bit cao nhất là **1** nằm ở cây **con phải**.
  - Đối với hai nút con của nút gốc, vấn đề tương tự đối với bit **z - 2** (bit đứng thứ nhì từ trái sang).

8.7. Cây tìm kiếm số học (Digital Search Tree - DST)




```
 6 = 0110
 5 = 0101
 2 = 0010
 7 = 0111
 8 = 1000
10 = 1010
12 = 1100
11 = 1011
 4 = 0100
```

8.7. Cây tìm kiếm số học (Digital Search Tree - DST)

- So với cây BST, DST chỉ khác về cách chia hai cây con trái/phải: tại nút gốc mức **d** thì chia cây con dựa trên bit thứ d tính từ trái sang (**bit z - d**) của mỗi khoá
- Giống BST: Với mỗi nút nhánh, mọi giá trị chứa trong cây con trái đều nhỏ hơn giá trị chứa trong cây con phải
- Các thao tác trên DST tương tự như trên BST. Cấu trúc nút:

```
typedef struct node {
    int Info;
    TNode * Left;
    TNode * Right;
} TNode;
```

8.7.1. Tìm kiếm trên cây tìm kiếm số học

- Thuật toán tìm khoá X trên DST:
  - Ban đầu đứng ở nút gốc, xét lần lượt các bit của X từ trái sang phải (từ bit **z - 1** tới bit **0**), hễ gặp bit bằng 0 thì rẽ sang nút con trái, nếu gặp bit bằng 1 thì rẽ sang nút con phải. Quá trình cứ tiếp tục như vậy cho tới khi gặp một trong hai tình huống sau:
  - Đi tới một nút rỗng- quá trình tìm kiếm thất bại
  - Đi tới một nút mang giá trị đúng bằng X - quá trình tìm kiếm thành công

8.7.1. Tìm kiếm trên cây tìm kiếm số học

- Cài đặt:

```
TNode * DSTSearch(TKey X) {
    int b = z;
    TNode * p = Root; //Bắt đầu với nút gốc
    while ((p!=NULL) && (p->Info!=X)) {
        b--; //Xét bit b của X
        if <Bit b của X là 0> then p=p->Left;
        else p = p->Right;
    }
    return p;
}
```

- Phép lấy bit b của X trong C: bit = X & (1<<b) >> b

8.7.2. Dựng cây tìm kiếm số học

- Ta chèn lần lượt các khoá vào cây, thao tác chèn tương tự thao tác tìm kiếm: tìm xem khoá đó đã có trong cây hay chưa,
  - nếu đã có rồi thì bỏ qua,
  - nếu chưa có thì ta móc nút mới chứa khoá cần chèn vào cây tìm kiếm số học tại mối nối rỗng vừa rẽ sang khiến quá trình tìm kiếm thất bại

8.7.2. Dựng cây tìm kiếm số học

- Cài đặt:

```
void DSTInsert(TKey X);
    int b = z;
    TNode * p = Root;
    TNode * q; // q chạy theo sau p, là cha của p
    <Duyệt cây để tìm nút p chứa khóa X>
    if (p == NULL) { // Giá trị X chưa có trong cây
        p = <tạo nút mới chứa X>;
        if (Root==NULL) Root = p; // Cây đang là
rỗng thì nút mới thêm trở thành gốc
        else // Không thì móc p vào mối nối vừa rẽ sang từ q
            if <Bit b của X là 0> q->Left = p;
            else q->Right = p;
    }
}
```

8.7.3. Xóa nút khỏi cây tìm kiếm số học

- Thuật toán xoá bỏ một giá trị khỏi cây tìm kiếm số học:
  - trước hết ta tìm nút D chứa giá trị cần xoá
  - trong nhánh cây gốc D ra một **nút lá bất kỳ**, chuyển giá trị chứa trong nút lá đó sang nút D rồi xoá nút lá.

8.7.3. Xóa nút khỏi cây tìm kiếm số học

- Cài đặt:

```
void DSTDelete(TKey X) {
    int b = z;
    TNode * p = Root;
    TNode * q; // q chạy theo sau p, là cha của p
    <Duyệt cây để tìm nút p chứa khóa X>
    if (p == NULL) return; // X không tìm được, kết thúc
    TNode * Node = p; // Giữ lại nút chứa khoá cần xoá
    while ((p->Left!=NULL) || (p->Right!=NULL) {
        q = p;
        if (p->Left != NULL) p = p->Left;
        else p = p->Right;
    }
    Node->Info = p->Info; // Chuyển giá trị từ nút lá p sang nút Node
    if (Root == p) Root = NULL; // Cây chỉ gồm 1 nút gốc, xoá cả gốc
    else // Cắt mối nối từ q tới p
        if (q->Left == p) q->Left = NULL;
        else q->Right = NULL;
    free(p); // giải phóng p
}
```

8.7.4. Cây tìm kiếm số học – nhận xét

- Về mặt trung bình, các thao tác **tìm kiếm**, **chèn**, **xoá** trên cây tìm kiếm số học đều có độ phức tạp là **O(min(z, logn))**
- Trường hợp xấu nhất, độ phức tạp của các thao tác đó là **O(z)**, bởi cây tìm kiếm số học có chiều cao không quá z + 1
- Cây tìm kiếm số học có thể mở rộng với cây K-phân, ở đó mỗi nút có thể có tới K nút con

8.8. Những nhận xét cuối cùng

- Tìm kiếm là công việc nhanh hơn sắp xếp nhưng lại được sử dụng nhiều hơn
- Bài toán tìm kiếm trong thực tế rất đa dạng: tìm bản ghi mang khoá **lớn hơn** hay **nhỏ hơn** khoá tìm kiếm, tìm bản ghi mang khoá **nhỏ nhất** mà lớn hơn khoá tìm kiếm…
- Tìm kiếm chuỗi ký tự cũng công việc khác được sử dụng rộng rãi và có những giải thuật đặc thù riêng: thuật toán BRUTE-FORCE, KNUTH- MORRIS-PRATT, BOYER-MOORE , RABIN-KARP…
- Không nên đánh giá giải thuật tìm kiếm này tốt hơn giải thuật khác và phải sử dụng thuật toán phù hợp với từng yêu cầu cụ thể