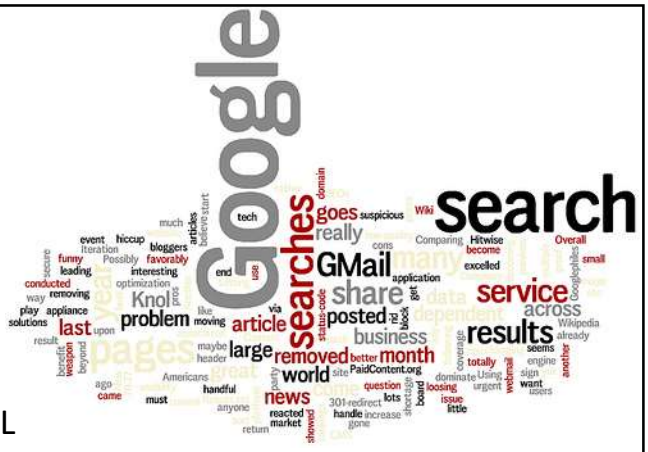
Tìm kiếm – searching

Tìm kiếm tuần tự, nhị phân, cây nhị phân tìm kiếm,
cây AVL, RB-tree, B-tree, Hash table

Nội dung



- Bài toán tìm kiếm
- Các thuật toán cơ bản
- Cây nhị phân tìm kiếm
- Cây nhị phân tìm kiếm cân bằng AVL
- Cây nhị phân tìm kiếm cân bằng RB-Tree
- Tìm kiếm trên bộ nhớ ngoài với B-tree
- Bảng băm và tìm kiếm với thời gian O(1)

Bài toán tìm kiếm

- Một số bài toán tìm kiếm:
  - Cho tên một người, tìm kiếm số điện thoại người đó trong danh bạ
  - Cho tên một tài khoản, tìm kiếm các giao dịch của tài khoản đó
  - Cho tên một sinh viên, tìm kiếm thông tin về kết quả học tập của sinh viên đó
  - Cho danh sách hồ sơ xét tuyển đại học, tìm và đưaa ra danh sách những hồ sơ có điểm xét tuyển nằm trong khoảng $k1 \leq x \leq k2$
  - Cho 1 từ khóa, tìm các tài liệu có từ khóa đó hoặc các tài liệu liên quan đến từ khóa đó
  - Cho 1 đoạn audio, hãy tìm cả bài hát đầy đủ
  - Cho 1 ảnh, hãy tìm các ảnh tương tự
- Đầu vào của bài toán tìm kiếm
  - Danh sách các phần tử với khóa tìm kiếm
  - Giá trị khóa cần tìm kiếm

Bài toán tìm kiếm

- Vấn đề cần xử lý với bài toán tìm kiếm
  - Kích thước danh sách tìm kiếm
    - Danh sách nhỏ có thể nạp hết vào RAM
    - Danh sách lớn phải lưu trữ ở bộ nhớ ngoài, hoặc
    - Danh sách lớn quá phải lưu trữ ở nhiều máy trong LAN hoặc internet
  - Kiểu giá trị khóa tìm kiếm (cần hỗ trợ toán tử ==, !=, >=, <=)
    - Kiểu đơn giản: kiểu số, xâu ký tự
    - Kiểu phức tạp: audio, ảnh, video,…
  - Kiểu bài toán
    - Tìm kiếm chính xác: giá trị trả về cần đúng bằng giá trị khóa cần tìm (hoặc nằm trong khoảng cho trước)
    - Tìm kiếm gần đúng: bài toán search text – full text search (google seach), tìm kiếm theo audio, tìm kiếm theo ảnh, tìm kiếm theo video,…

Bài toán tìm kiếm

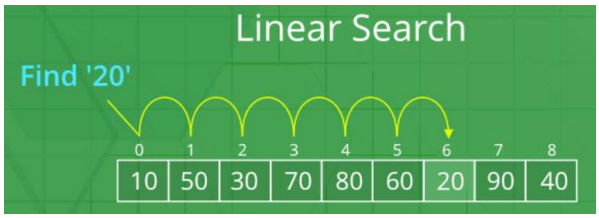
- Vấn đề với bài toán tìm kiếm
  - Nếu danh sách nhỏ đủ nạp vào RAM, thuật toán tìm kiếm cần tối ưu số lượng phép so sánh
  - Danh sách lớn, cần lưu trên bộ nhớ ngoài thì thuật toán cần tối ưu số lần phải truy cập ra bộ nhớ ngoài
  - Danh sách quá lớn, phải lưu trữ trên nhiều máy → cần cách tiếp cận song song hoặc phân tán (phân tán chương trình tới nơi có dữ liệu)
  - Với kiểu khóa phức tạp, thời gian so sánh khóa sẽ ảnh hưởng tới việc tìm kiếm nên cần phương pháp xử lý khóa trước (VD. chuyển khóa kiểu text sang số,…)
  - Danh sách lớn cần áp dụng các phương pháp tiền xử lý để tăng tốc tìm kiếm. VD. minHash
  - Tìm kiếm theo khoảng cần sắp xếp dữ liệu trước. VD. B-tree

Các thuật toán tìm kiếm cơ bản

- **Tìm kiếm tuần tự**:
  - Là phương pháp đơn giản nhất
  - Duyệt từ đầu đến cuối danh sách cho đến khi tìm được bản ghi mong muốn,
  - hoặc là đến cuối mà không tìm được

- Danh sách đầu vào KHÔNG cần có thứ tự,
- Khóa chỉ cần hỗ trợ toán tử == hoặc !=

```
int sequentialSearch(int records[], int key)
{
    int i;
    for(i=0;i<MAX; i++)
    {
       if(records[i]==key)
       {
           return i; //tim thay
        }
    }
    return -1;//khong tim thay
}
```




```c
typedef struct node
{
    char hoten[30];
    char diachi[50];
    float diemTB;
    struct node *pNext;
} NODE;
```

```c
NODE* sequentialSearch(NODE *head, char *key)
{
    NODE *ptr=head;
    while(head!=NULL)
    {
        if(strcmp(head->hoten,key)==0)
              return head;
        else head = head->pNext;
    }
    return NULL;
}
```

Tìm kiếm tuần tự

- Thời gian thực hiện
  - Trường hợp tốt nhất: chỉ cần 1 phép so sánh
  - Trường hợp tồi nhất: cần n phép so sánh
  - Trung bình cần : O(n)
- Độ phức tạp : O(n)

➔Chỉ phù hợp cho danh sách ít phần tử

➔hoặc tìm kiếm không thường xuyên, hoặc không yêu cầu cao về tốc độ

➔Dữ liệu không cần sắp xếp

➔Bộ nhớ hạn chế hoặc không cài đặt được các thuật toán phức tạp

Không cần danh sách có thứ tự, không cần truy cập các phần tử ngẫu nhiên → áp dụng được cả trên mảng và danh sách liên kết

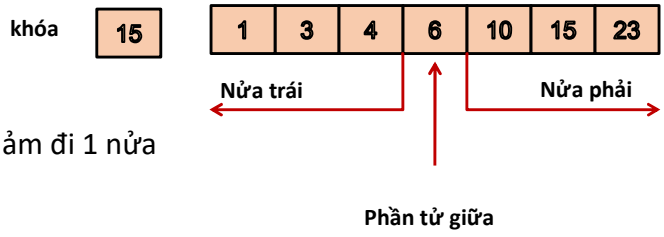
Bài tập: Viết lại hàm tìm kiếm tuần tự để có thể đếm được số lượng phép so sánh cần thiết trong quá trình tìm kiếm (đối với danh sách móc nối).

Tìm kiếm tuần tự

- Tìm kiếm trong danh sách nhỏ: danh bạ, giỏ hàng, bộ nhớ đệm,..
- Kiểm tra dữ liệu hợp lệ: danh sách dữ liệu của cảm biến, tài khoản ngân hàng, sdt,…
- Tìm kiếm trong dữ liệu chưa sắp xếp: đơn hàng, giỏ hàng,…
- Tìm kiếm trong văn bản ngắn
- Tìm kiếm trong tập nhỏ
  - Hệ thống quét mã POS, vé xe,…
- Tìm kiếm trong hệ thống nhúng: bộ nhớ và khả năng xử lý giới hạn

Các thuật toán tìm kiếm cơ bản

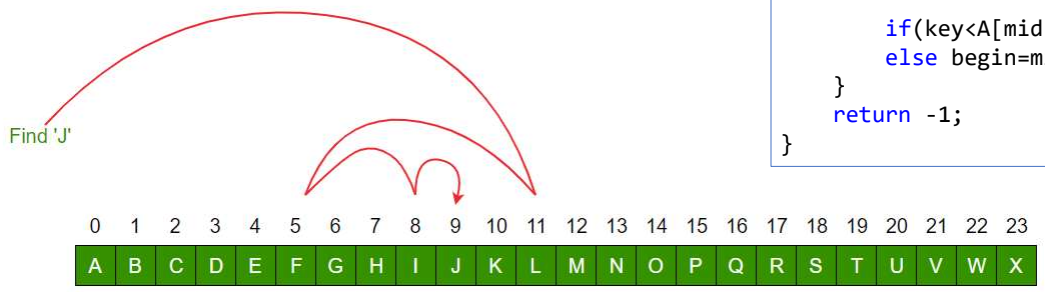
- **Tìm kiếm nhị phân**
  - Danh sách ban đầu cần được sắp thứ tự theo khóa
  - Khi tìm kiếm, so sánh trực tiếp giá trị khóa cần tìm với khóa của phần tử ở giữa danh sách hiện tại
  - Việc tìm kiếm tiếp theo dựa trên mỗi quan hệ với phần tử ở giữa danh sách
    - Nếu bằng → tìm thấy
    - Nếu nhỏ hơn → tìm tiếp ở nửa trái (dãy sắp theo thứ tự tăng dần)
    - Nếu lớn hơn → tìm tiếp tại nửa phải
  - Sau mỗi phép so sánh, kích thước danh sách tìm kiếm giảm đi 1 nửa



```
int binarySearch_Re(int A[], int key,int begin, int end)
{
    if(begin>end)return -1;
    int mid=(begin+end)/2;
    if(A[mid]==key) return mid;
    if(key<A[mid])
         return binarySearch_Re(A,key,begin,mid-1);
    else return binarySearch_Re(A,key,mid+1,end);
}
```

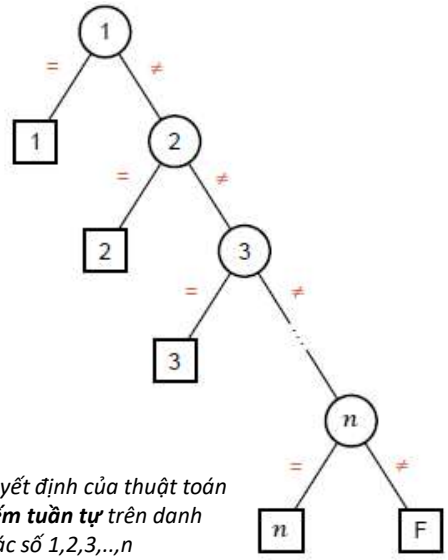
```
int binarySearch(int A[], int key,
            int begin, int end)
{
    int mid;
    while(begin<=end)
    {
        mid=(begin+end)/2;
        if(A[mid]==key)
                return mid;

        if(key<A[mid]) end=mid-1;
        else begin=mid+1;
    }
    return -1;
}
```



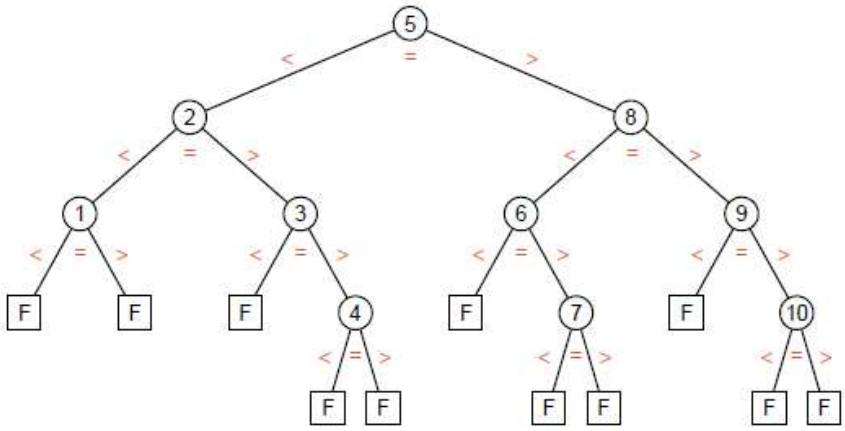
Tìm kiếm nhị phân

- Danh sách tìm kiếm ban đầu cố định
  - Quá trình tìm kiếm có thể mô tả lại bằng cây đệ quy
  - Mỗi nút trong là 1 phép so sánh với giá trị khóa cần thực hiện
  - Nút là là Fail tương ứng với việc tìm không thấy
  - Số lượng phép so sánh tối đa tỉ lệ với chiều cao của cây



*Cây quyết định của thuật toán **tìm kiếm tuần tự** trên danh sách các số 1,2,3,..,n*

Tìm kiếm nhị phân



*Cây quyết định cho thuật toán **tìm kiếm nhị phân** 1 trên dãy số 1, 2, 3, 4, 5, 6, 7, 8, 9, 10*

Tìm kiếm nhị phân

- Tìm kiếm nhị phân
  - Yêu cầu danh sách tìm kiếm phải có thứ tự (thời gian sắp xếp cỡ O(nlogn))
  - Phải hỗ trợ truy cập ngẫu nhiên phần tử (cấu trúc liên tiếp- mảng)
  - Là thuật toán tìm kiếm tổng quát tốt nhất dựa trên so sánh khóa
  - Thời gian tìm kiếm trong trường hợp tồi nhất O(logn)
  - Phù hợp khi tìm kiếm trên danh sách lớn và tìm kiếm thường xuyên

Tìm kiếm nhị phân

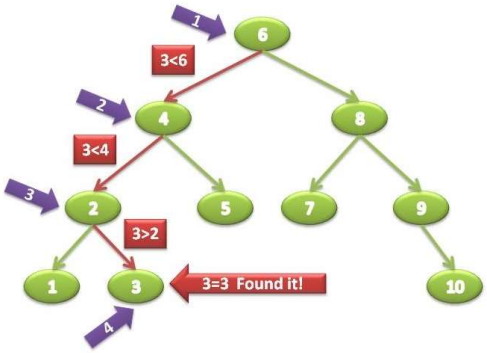
- Tìm kiếm thông tin khách hàng, sản phẩm, hoặc mã đặt hàng trong cơ sở dữ liệu đã sắp xếp.
  - VD. Đơn hàng, mã sách trong thư viện, danh sách thi,…
  - Tìm sản phẩm theo điểm số đánh giá, theo giá, năm sản xuất
- Xác định vị trí phân bổ trong danh sách đã sắp xếp
  - Tìm vị trí chỗ ngồi theo danh sách ưu tiên, vị trí chèn trong danh sách
  - Tìm kiếm thứ hạng theo điểm
- Tìm kiếm trong hệ thống tài chính
  - Tra cứu mã cổ phiếu, trái phiếu,..
- Tìm kiếm trong chuỗi thời gian (Time Series)

Các thuật toán tìm kiếm cơ bản

1	5	7	9	21	35
3	12	18	25	37	54
15	16	24	38	42	67
22	23	37	48	54	79

K=34

- Bài 1. Cho 1 dãy n số nguyên không trùng lặp và 1 giá trị k, tìm xem có tồn tại trong dãy 2 số a và b sao cho a+b=k
- Bài 2. Cho 1 ma trận trong đó các phần tử thuộc hàng và cột đều có thứ tự. Hãy xây dựng thuật toán tìm kiếm xem 1 giá trị k có xuất hiện trong ma trận đó không
- Bài 3. Cho 1 dãy các số dương đã có thứ tự nhưng bị xen lẫn bởi các số 0, hãy xây dựng thuật toán hiệu quả để tìm kiếm giá trị k>0 trong dãy {1,2,15,0,0,0,0,0,37,0,0,0,45,0,54,107,0,0,0}
- Bài 4. Cho dãy gồm n số nguyên có trùng lặp, hãy xây dựng thuật toán hiệu quả để loại bỏ các phần tử bị trùng lặp



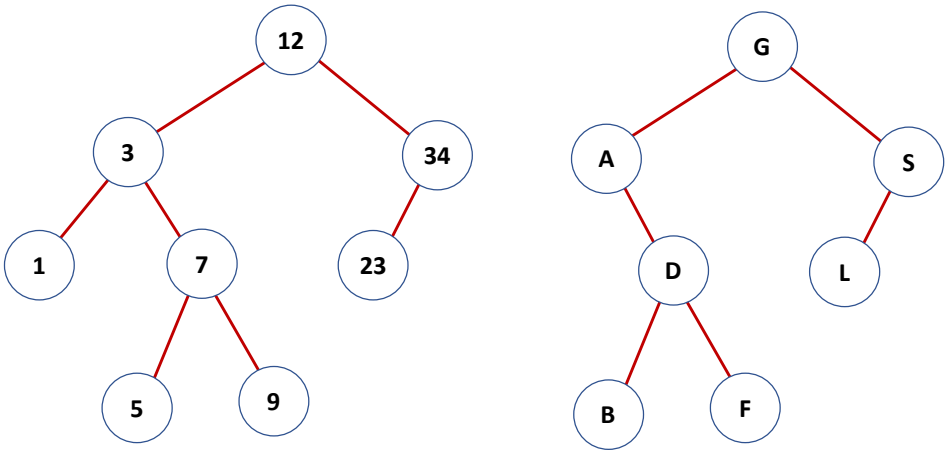
Cây tìm kiếm nhị phân
Binary search tree

- Áp dụng tìm kiếm nhị phân trên cấu trúc liên kết (danh sách liên kết) ?
- Bài toán tìm kiếm: Phải lưu trữ danh sách tìm kiếm (thao tác thêm, xóa phần tử)?
  - Cấu trúc dữ liệu mảng sẽ tồi nếu phải thêm/xóa
  - Danh sách liên kết không tìm kiếm phị phân được
  - Cần cấu trúc hỗ trợ tốt cho cả thêm/xóa và tìm kiếm

Cây tìm kiếm nhị phân

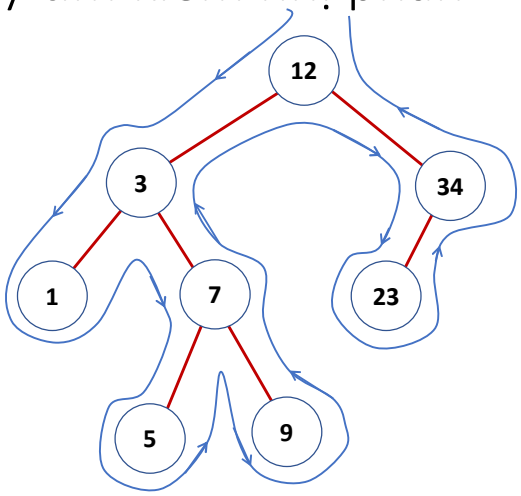
- Cây tìm kiếm nhị phân – binary search tree (BST):
  - Thời gian thực hiện tìm kiếm nhanh ($O(\log n)$)
  - Thêm, xóa các phần tử dễ dàng
- Cây tìm kiếm nhị phân là cây nhị phân rỗng hoặc mỗi nút có một giá trị khóa thỏa mãn các điều kiện sau:
  - Giá trị khóa của nút gốc (nếu tồn tại) lớn hơn giá trị khóa của bất kỳ nút nào thuộc cây con trái của gốc
  - Giá trị khóa của nút gốc (nếu tồn tại) nhỏ hơn giá trị khóa của bất kỳ nút nào thuộc cây con phải của gốc
  - Cây con trái và cây con phải của gốc cũng là các cây nhị phân tìm kiếm

Cây tìm kiếm nhị phân



Các nút trên cây nhị phân phải có khóa phân biệt !

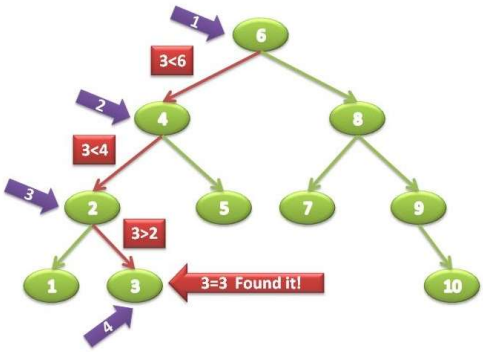
Cây tìm kiếm nhị phân



- Duyệt cây tìm kiếm nhị phân
- Thứ tự giữa
  1, 3, 5, 7, 9, 12, 23, 34
- Thứ tự trước
  12, 3, 1, 7, 5, 9, 34, 23
- Thứ tự sau
  1, 5, 9, 7, 3, 23, 34, 12

```
struct TreeNode
{
    DATA_TYPE key;
    struct TreeNode *leftChild;
    struct TreeNode *rightChild;
}
```

Cây tìm kiếm nhị phân



Tìm kiếm:

- Nếu cây rỗng → không tìm thấy
- So sánh khóa cần tìm với khóa của nút gốc
  - Nếu bằng → Tìm thấy
  - Ngược lại lặp lại quá trình tìm kiếm ở cây con trái (hoặc cây con phải)
    - Nếu Khóa cần tìm nhỏ hơn giá trị khóa tại gốc → xuống con trái
    - Ngược lại → xuống tiếp con phải

Cây tìm kiếm nhị phân

- Nếu khóa bị trùng?
  - Nếu có khóa bị trùng quá trình tìm kiếm sẽ không thực hiện được
  - Gộp các khóa bị trùng lại
  - Thêm trường của node đếm tần số xuất hiện khóa
- Một số thao tác cơ bản của ADT cây tìm kiếm nhị phân
  - Search(): tìm kiếm xem một giá trị khóa có xuất hiện trên cây không
  - Find(): tìm xem khóa key có xuất hiện trên cây hay không (trả về true/false)
  - Insert(): Chèn một nút mới vào cây
  - Remove(): Gỡ bỏ một nút trên cây
  - TreeTraversal(): tương tự cây nhị phân tổng quát

Thao tác tìm kiếm trên cây

```
typedef struct TreeNode
{
   int key;
   struct TreeNode *leftChild;
   struct TreeNode *rightChild;
}NODE;
```

```
NODE* search_recc(NODE *root, int searchkey)
{
  if(root==NULL || root->key== searchkey) return root;
  if(searchkey <root->key)
       return search_recc(root->leftChild, searchkey);
  else return search_recc(root->rightChild, searchkey);
}
```

```
NODE* search_loop(NODE *root, int searchkey)
{
  while(root!=NULL && root-> key!= searchkey)
    if(searchkey <root->key) root=root->leftChild;
    else root=root->rightChild;
  return root;
}
```

Cây tìm kiếm nhị phân

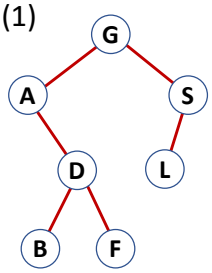
- **Ví dụ 1**. Kiểm tra 1 cây nhị phân có phải cây nhị phân tìm kiếm
- Gợi ý
  - Cách 1. Duyệt cây theo thứ tự giữa, dãy thu được phải là dãy tăng
    - Cần 2 lần duyệt
  - Cách 2. Chỉ cần 1 lần duyệt?
    - Duyệt và kiểm tra luôn

```
bool isBinarySearchTree(NODE* root, int LowerBound, int UpperBound)
{
   if (NULL == root) return true;
   if (root->key < LowerBound || root->key > UpperBound) return false;
   return isBinarySearchTree(root->leftChild, LowerBound, root->key)  &&
          isBinarySearchTree(root->rightChild, root->key, UpperBound);
}
```

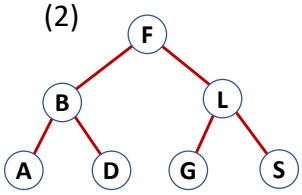
Cây tìm kiếm nhị phân

- Ví dụ 2. Cho 1 giá trị khóa k, hãy tìm ra nút gần với k nhất trên cây nhị phân tìm kiếm (có chênh lệch với giá trị k là nhỏ nhất)
- Ví dụ 3. Tìm nút lớn nhất/nhỏ nhất trên cây nhị phân tìm kiếm
- Ví dụ 4. chuyển cây nhị phân thông thường về thành cây nhị phân tìm kiếm mà vẫn giữ nguyên hình dạng của cây ban đầu
- Ví dụ 5. Xây dựng cây nhị phân tìm kiếm với chiều cao thấp nhất từ danh sách n phần tử ban đầu
- Ví dụ 6. In ra các nút trên cây nhị phân tìm kiếm có khóa nằm trong khoảng giá trị k1<=x<=k2
- Ví dụ 7. Tìm tổ tiên chung gần nhất của 2 nút trên cây nhị phân tìm kiếm
- Ví dụ 8. Tìm nút lá có giá trị lớn nhất trên cây nhị phân tìm kiếm

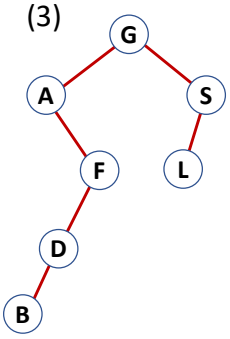
(1)



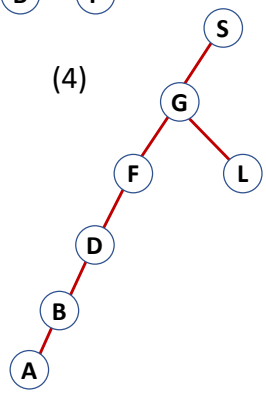
(2)



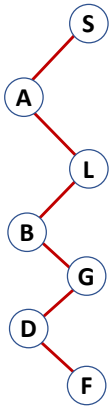
(3)



(4)



(5)

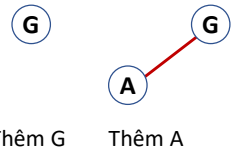


Cây nhị phân tìm kiếm cân bằng cho thời gian tìm kiếm là nhỏ nhất $O(\log n)$

Thêm nút mới vào cây

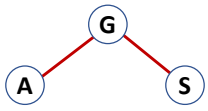
- Khi thêm nút mới phải đảm bảo những đặc điểm của cây nhị phân tìm kiếm không bị vi phạm

Thêm lần lượt các khóa : G, A, S, D, L, B, F vào cây nhị phân tìm kiếm ban đầu rỗng

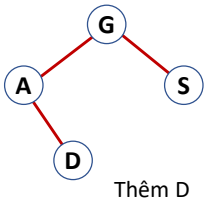


Thêm G

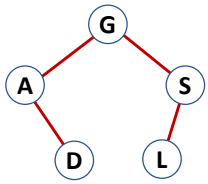
Thêm A



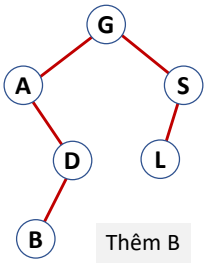
Thêm S



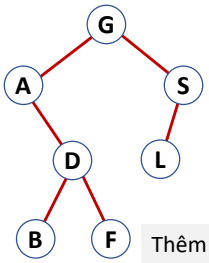
Thêm D



Thêm L



Thêm B



Thêm F

Thêm nút mới vào cây

- Nhận xét:
  - Thứ tự thêm các nút khác nhau ***có thể*** tạo ra các cây nhị phân tìm kiếm khác nhau
  - Khóa đầu tiên thêm vào cây rỗng sẽ trở thành nút gốc của cây
  - Để thêm các khóa tiếp theo ta phải thực hiện tìm kiếm để tìm ra vị trí chèn thích hợp
  - Các khóa được thêm tại nút lá của cây
  - Cây sẽ bị suy biến thành danh sách liên kết đơn nếu các nút chèn vào đã có thứ tự (tạo ra các cây lệch trái hoặc lệch phải)

Thêm nút mới vào cây

- Cài đặt đệ quy

```
NODE* makeNewNode(int key)
{
   NODE* newNode = (NODE*)malloc(sizeof(NODE));
   newNode->key = key;
   newNode->leftChild = NULL;
   newNode->rightChild = NULL;
   return newNode;
}
```

```
void insert(NODE*& root, int key)
{
   if (root == NULL) root = makeNewNode(key);
   else if (key < root->key)  insert(root->leftChild, key);
   else if (key > root->key)  insert(root->rightChild, key);
}
```

```
// Phien ban code = C
void insert(NODE** root, int key)
{
   if (*root == NULL) *root = makeNewNode(key);
   else if (key < (*root)->key)  insert(&((*root)->leftChild), key);
   else if (key > (*root)->key)  insert(&((*root)->rightChild), key);
}
```

Thêm nút mới vào cây

- Cài đặt không đệ quy

```
// root phai khac NULL
int insert(NODE* root, int key)
{
   NODE* pRoot = root;  // con tro toi truoc vi tri can chen
   while (true)  {
      if (pRoot->key == key) return -1;//duplicate
      if (pRoot->key > key)  {   // them vao con trai
         if (pRoot->leftChild == NULL) {
            pRoot->leftChild = makeNewNode(key);
            break;
         }
         else pRoot = pRoot->leftChild;
      }
      else {    // them vao con phai
         if (pRoot->rightChild == NULL) {
            pRoot->rightChild = makeNewNode(key);
            break;
         }
         else pRoot = pRoot->rightChild;
      }
   }
   return 0;//success
}
```

Thêm nút mới vào cây

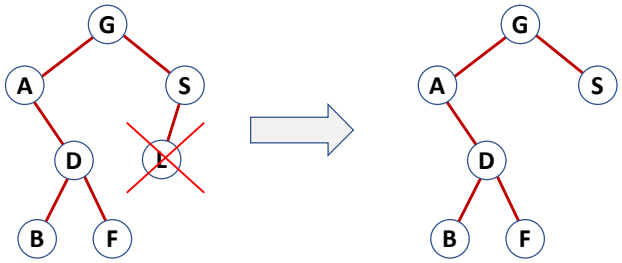
- **Nhận xét**: Thời gian thực hiện của thuật toán phụ thuộc vào dạng của cây
  - Cây cân bằng : thời gian cỡ $O(\log n)$
  - Cây bị suy biến (lệch trái, phải hoặc zig-zag): thời gian cỡ $O(n)$

$$O(\log n) \leq T(n) \leq O(n)$$

  - **Thuật toán sắp xếp (treeSort)**: thêm lần lượt các phần tử vào cây nhị phân tìm kiếm, sau đó duyệt theo thứ tự giữa.
    Số phép so sánh: $1.39\, n \log n + O(n)$

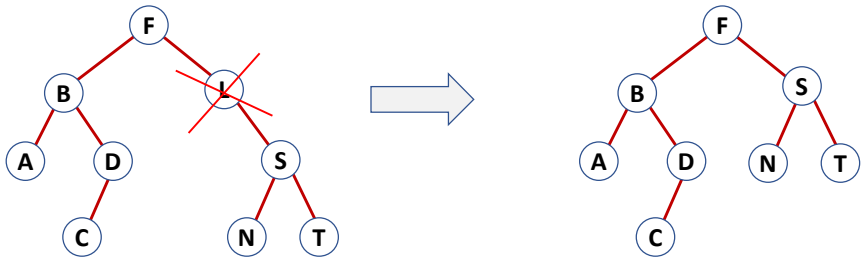
Loại bỏ nút khỏi cây

- **Loại bỏ nút trên cây**: cần đảm bảo những đặc điểm của cây không bị vi phạm
- Trường hợp loại bỏ nút lá:



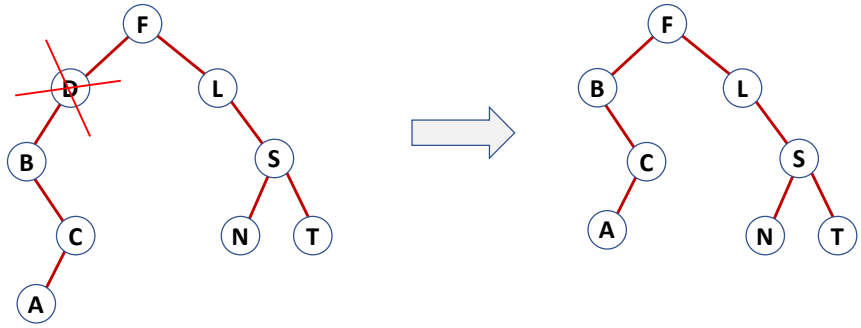
Loại bỏ nút khỏi cây

- Trường hợp loại bỏ nút trong: nút trong khuyết con trái hoặc con phải



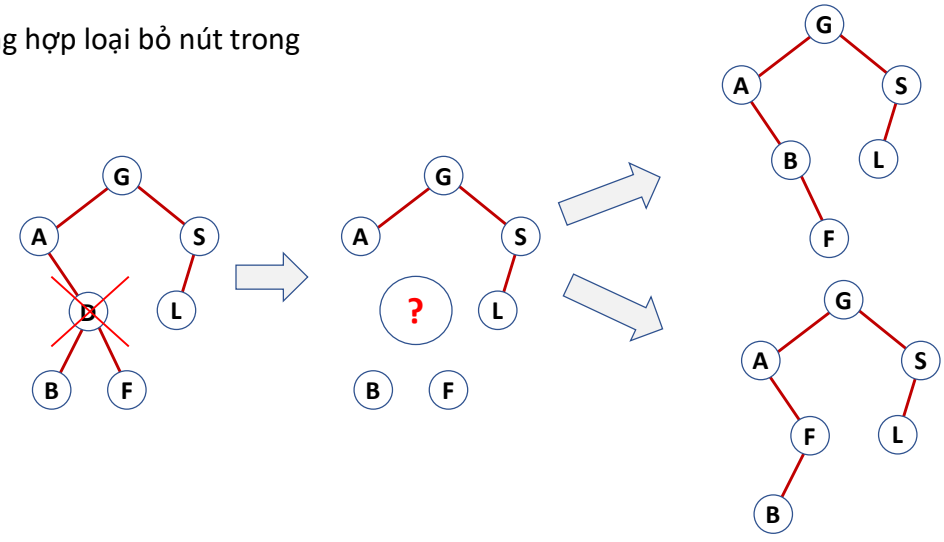
Loại bỏ nút khỏi cây

- Trường hợp loại bỏ nút trong: nút trong khuyết con trái hoặc con phải

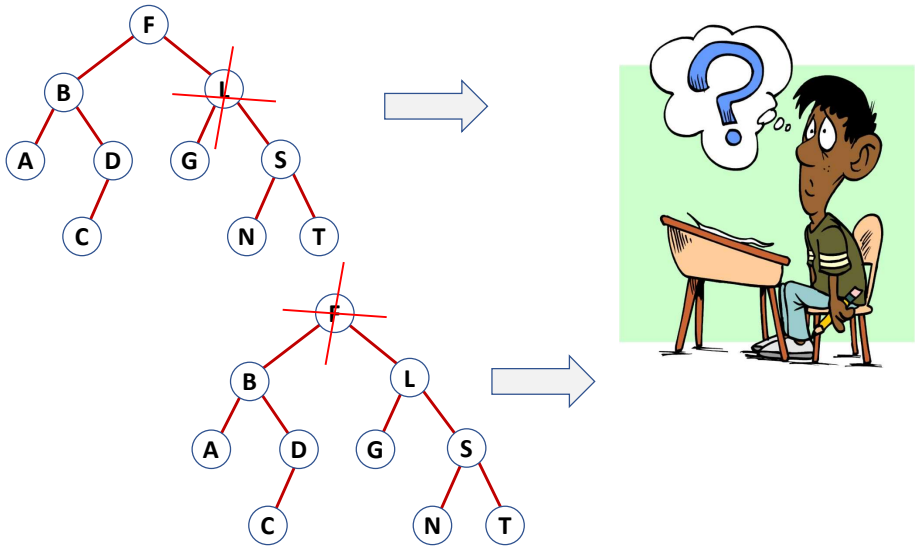


Loại bỏ nút khỏi cây

- Trường hợp loại bỏ nút trong



Loại bỏ nút khỏi cây



Loại bỏ nút khỏi cây

- Nhận xét:
  - Thực hiện tìm kiếm để xem khóa cần xóa có trên cây
  - Nếu nút có khóa cần xóa là nút lá: ngắt bỏ kết nối với nút cha của nó, giải phóng bộ nhớ cấp phát cho nút đó
  - Nếu nút cần xóa là nút trong không đầy đủ (khuyết con trái hoặc phải): Thay thế bằng cây con không khuyết
  - Nếu nút cần xóa là nút trong đầy đủ (có đủ các con): cần tìm một nút thuộc để thay thế cho nó, sau đó xóa nút được thay thế. Nút thay thế là nút ở liền trước (hoặc liền sau trong duyệt theo thứ tự giữa)
    - Tìm nút phải nhất của cây con trái (*)
    - Hoặc, nút trái nhất thuộc cây con phải

```
int remove_node(NODE *&root)
{
   if(root == NULL) return -1; //remove null
   NODE *ptr = root; //remember this node for delete later
   if(root->leftChild == NULL) root=root->rightChild;
   else if(root->rightChild == NULL) root=root->leftChild;
   else //find the rightmost node on the left sub tree
   {
      NODE *preP = root;
      ptr = root->leftChild;
      while(ptr->rightChild != NULL)
      {
         preP = ptr;
         ptr = ptr->rightChild;
      }
      root->key = ptr->key;
      if(preP == root) root->leftChild = ptr->leftChild;
      else preP->rightChild = ptr->leftChild;
   }
   free(ptr);
   return 0;// remove success
}
```

```
int remove(NODE*& root, int key)
{
   if (root == NULL || key == root->key)
         return remove_node(root);
   else if (key < root->key)
         return remove(root->leftChild, key);
   else return remove(root->rightChild, key);
}
```

```
int remove_node(NODE **root){
    if((*root) == NULL) return -1; //remove null
    NODE *ptr = *root; //remember this node for delete later
    if((*root)->leftChild == NULL) (*root)=(*root)->rightChild;
    else if((*root)->rightChild == NULL) *root=(*root)->leftChild;
    else{ //find the rightmost node on the left sub tree
        NODE *preP = *root;
        ptr = (*root)->leftChild;
        while(ptr->rightChild != NULL) {
            preP = ptr;
            ptr = ptr->rightChild;
        }
        (*root)->key = ptr->key;
        if(preP == *root) (*root)->leftChild = ptr->leftChild;
        else preP->rightChild = ptr->leftChild;
    }
    free(ptr);
    return 0;// remove success
}
```

```
int removeNode(NODE** root, int key)
{
   if (*root == NULL || key == (*root)->key)
      return remove_node(root);
   else if (key < (*root)->key)
      return removeNode(&((*root)->leftChild), key);
   else return removeNode(&((*root)->rightChild), key);
}
```

Ứng dụng

- Quản lý cơ sở dữ liệu (Database Management)
  - Lưu trữ, tìm kiếm thông tin khách hàng, bệnh nhân dựa trên khóa
- Hệ thống tệp tin (File System)
  - Sắp xếp và tìm kiếm file theo ngày tạo, kích thước, tên,…
- Tìm kiếm từ trong từ điển
- Tổ chức bảng mã trong hệ thống nén dữ liệu, vd Huffman
- Quản lý các sự kiện theo thời gian, vd log event,…
- Hệ thống tìm kiếm trên web
- Xử lý dữ liệu không gian: tọa độ
- Cấu trúc chỉ mục trong cơ sở dữ liệu: VD chỉ mục của sách, tra cứu biến hàm,.. Trong trình biên dịch
- Quản lý luồng truy cập mạng: ip bị cấm, blacklist,..

Loại bỏ nút khỏi cây

- Ví dụ 9. Cho 2 cây nhị phân tìm kiếm với các khóa phân biệt, hãy trộn 2 cây này lại để thu được 1 cây nhị phân tìm kiếm lớn hơn
  - Yêu cầu thời gian nhanh nhất
  - Yêu cầu dùng ít bộ nhớ phụ nhất
- Ví dụ 10. In ra các khoá trùng nhau trên 2 cây nhị phân tìm kiếm BST1 và BST2
  - Bộ nhớ phụ tốn ít nhất
  - Không giới hạn bộ nhớ phụ
- Ví dụ 11. tìm và in ra đường đi giữa 2 nút trên cây (giả sử có thể di chuyển theo chiều từ nút con ngược lại nút cha)
- Ví dụ 12. tìm k nút nhỏ nhất trên cây nhị phân tìm kiếm
- Ví dụ 13. Loại bỏ các nút trên cây nhị phân tìm kiếm có khóa nằm ngoài khoảng giá trị k1<=x<=k2
- Ví dụ 14. Tìm khóa lớn/nhỏ thứ k trên cây nhị phân tìm kiếm
- Ví dụ 15. Kiểm tra xem các nút trong của cây nhị phân chỉ có chính xác 1 con