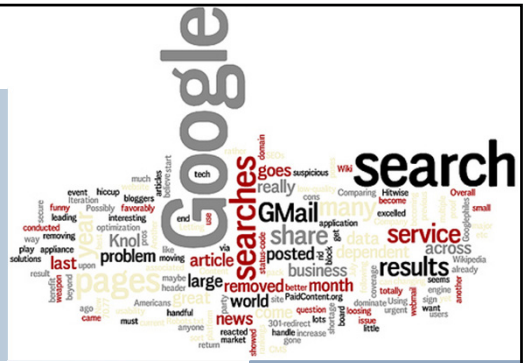
Chương 4 Tìm kiếm (phần 1)



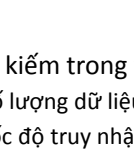
Hiepnd@soit.hut.edu.vn

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
- Khóa(key) là thông tin mà chúng ta dùng để tìm kiếm
- Tìm kiếm trong và tìm kiếm ngoài
  - Số lượng dữ liệu
  - Tốc độ truy nhập dữ liệu

Nội dung

- Bài toán tìm kiếm
- Tìm kiếm tuần tự
- Tìm kiếm nhị phân
- Cây quyết định

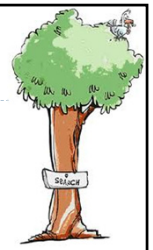


Tìm kiếm tuần tự

Tìm kiếm tuần tự

- **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

Tìm kiếm tuần tự



- Tìm kiếm trên danh sách móc nối

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

Tìm kiếm tuần tự

Tìm kiếm trên mảng

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

Tìm kiếm tuần tự

```
int sequentialSearch(NODE *pHead, char key[], NODE *&retVal)
{
    NODE *ptr=pHead;
    while(ptr!=NULL)
    {
        if(strcmp(ptr->hoten,key)==0)
        {
            retVal=ptr;
            return 0;
        }
        else ptr=ptr->pNext;
    }
    return -1;
}
```

Phân tích

- Giả sử tồn tại bản ghi chứa khóa cần tìm.
- Trường hợp tốt nhất: chỉ cần 1 phép so sánh
- Trường hợp tồi nhất: cần n phép so sánh
- Trung bình cần :

$$\frac{1+2+3+...+n}{n}$$

- Độ phức tạp : $O(n)$

Chỉ phù hợp cho danh sách ít phần tử

Tìm kiếm nhị phân

Tìm kiếm tuần tự

- **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 nhị phân

- Trường hợp các bản ghi đã được sắp xếp theo thứ tự, tìm kiếm tuần tự không hiệu quả
  - Tìm kiếm từ "study" trong từ điển
  - Tìm kiếm "Tran Van C" trong danh ba điện thoại
- Tìm kiếm nhị phân:
  - So sánh khóa với phần tử trung tâm danh sách, sau đó ta chỉ xét nửa trái hoặc nửa phải danh sách căn cứ vào khóa đứng trước hay sau phần tử trung tâm

Tìm kiếm nhị phân

- **Yêu cầu**: Danh sách phải được sắp xếp theo một thứ tự nào đó trước (danh sách có thứ tự).

  Ví dụ:
  - các từ trong từ điển,
  - tên người trong danh bạ điện thoại

- **Danh sách có thứ tự**: là danh sách trong đó mỗi mục chứa một khóa theo thứ tự.

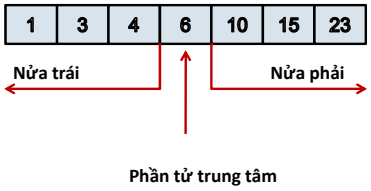
  Nếu mục ***i*** đứng trước ***j*** trong danh sách, thì khóa của mục ***i*** sẽ nhỏ hơn hoặc bằng khóa của mục ***j***

Tìm kiếm nhị phân

```
int binarySearch1_Re(int A[], int key, int begin,
                                    int end, int &position)
{
    if(begin>end)return -1;
    int mid=(begin+end)/2;
    if(A[mid]==key)
    {
        position=mid;
        return 0;
    }
    if(key<A[mid]) return binarySearch(A,key,begin,
                                          mid-1,position);
    else return binarySearch(A,key,mid+1,end,position);
}
```

Tìm kiếm nhị phân

khóa **15**



Tìm kiếm nhị phân

```
int binarySearch1(int A[], int key, int begin, int end,
                                                int &position)
{
    int mid;
    while(begin<=end)
    {
        mid=(begin+end)/2;
        if(A[mid]==key)
        {
            position=mid;
            return 0;
        }
        if(key<A[mid]) end=mid-1;
        else begin=mid+1;
    }
    return -1;
}
```

Ví dụ

VD1.Khi tìm kiếm bảng chứa 5000 dữ liệu bằng phép tìm kiếm nhị phân (binary search method), số lần so sánh bình quân sẽ là bao nhiêu?

$Log_{10}2 = 0.3010$

A. 8
B. 9
C. 10
D. 11
E. 12

Cây quyết định

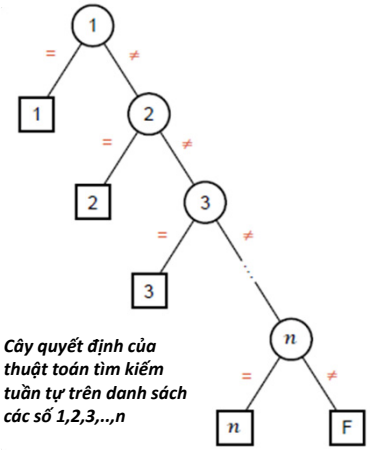
Ví dụ

- Trong tìm kiếm nhị phân, khi số lượng dữ liệu đã sắp xếp tăng gấp 4 lần thì số lượng phép so sánh tối đa tăng bao nhiêu?

Cây quyết định

- Tên khác: cây so sánh, cây tìm kiếm
- Cây quyết định của một thuật toán thu được bằng cách theo vết các hành động của thuật toán
  - Biểu diễn mỗi phép so sánh khóa bằng 1 nút của cây (biểu diễn bằng hình tròn)
  - Cạnh vẽ từ đỉnh biểu diễn các khả năng có thể xảy ra của phép so sánh
  - Kết thúc thuật toán ta đặt F nếu không tìm thấy, hoặc là vị trí tìm thấy khóa (đây gọi là các lá của cây)

Cây quyết định



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

Gốc: đỉnh của cây

Chiều cao của cây: số lượng nút trên đường đi dài nhất từ gốc đến lá

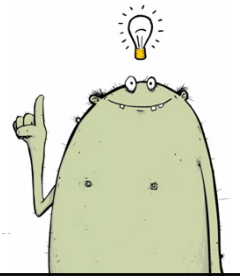
Mức của đỉnh: số lượng cạnh trên đường từ gốc đến đỉnh đó

Số lượng phép so sánh trong một trường hợp cụ thể là số lượng nút trong

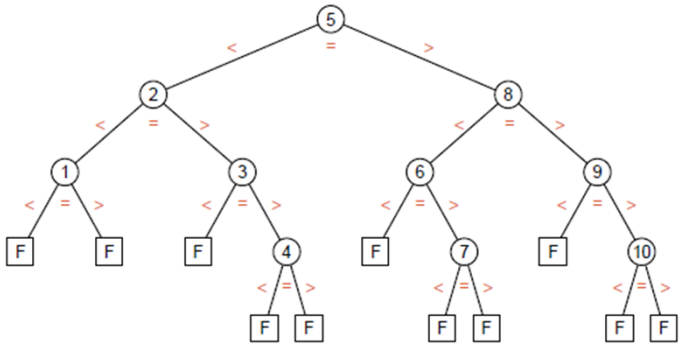
Cây quyết định

- Cây quyết định là 2-tree: các nút trong đều chỉ có 2 nút con.
- Số lượng phép so sánh trong thuật toán tìm kiếm nhị phân ở trên trong trường hợp tìm không thấy là $2\log(n+1)$
- Số phép so sánh trung bình trong trường hợp tìm thấy là

$$\frac{2(n+1)}{n}\log(n+1)-3$$



Cây quyết định



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

Extra section

Tìm kiếm nhị phân

Một cài đặt khác của tìm kiếm nhị phân

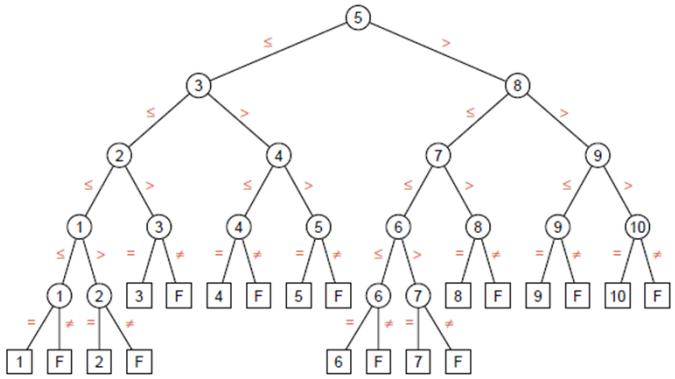
```
int binarySearch2(int A[], int key, int begin, int end, int &position)
{
    int mid;
    while(begin<end)
    {
        mid=(begin+end)/2;
        if(key<=A[mid]) end=mid-1;
        else begin=mid;
    }
    if(begin>end) return -1;
    else if(A[begin]=key) //begin=end
    {
        position=begin;
        return 0;
    }
}
```

Tìm kiếm nhị phân

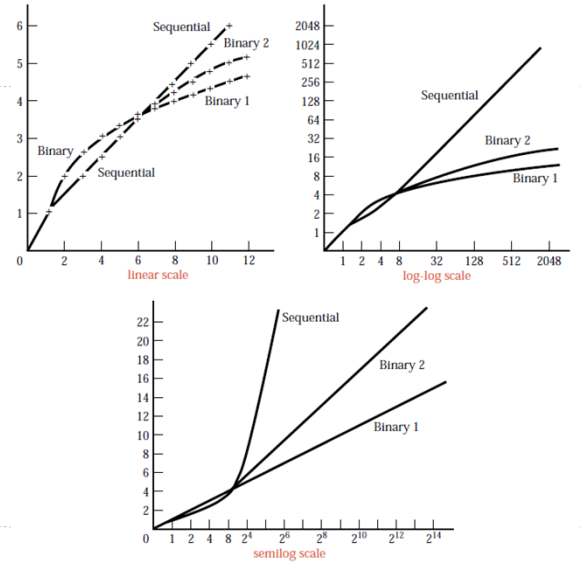
- So sánh hai cài đặt của thuật toán tìm kiếm nhị phân

	Tìm kiếm thành công	Tìm kiếm không thành công
binarySearch1	$\log n + 1$	$\log n + 1$
binarySearch2	$2\log n - 3$	$2\log n$

Tìm kiếm nhị phân



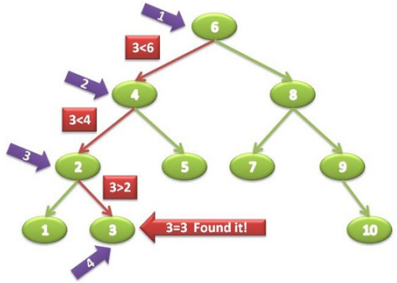
- Cây tìm kiếm tương ứng với thuật toán tìm kiếm nhị phân 2 trên dãy 1,2,3,4,5,6,7,8,9,10



Nhận xét

- Thuật toán tìm kiếm nhị phân là thuật toán tốt nhất trong các thuật toán tìm kiếm dựa trên việc so sánh giá trị các khóa trong dãy.
- Ý tưởng mở rộng thuật toán tìm kiếm: Nếu chúng ta biết khoảng tìm kiếm của mỗi khóa thì chúng ta có thể thu hẹp phạm vi của việc tìm kiếm

- Thuật toán ***interpolation search :***
  $O(\log\log n)$ trong trường hợp các khóa phân bố đều
  $O(n)$ trong trường hợp tồi nhất
  http://en.wikipedia.org/wiki/Interpolation_search



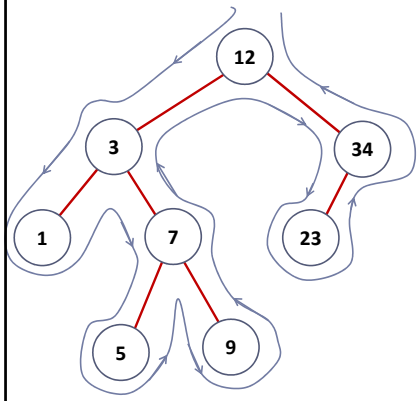
Cây tìm kiếm nhị phân
Binary search tree

- Áp dụng tìm kiếm nhị phân trên cấu trúc liên kết (danh sách liên kết) ?
- Thêm, xóa phần tử trên cấu trúc liên tiếp (mảng) ?
  - Chi phí về thời gian?
  - Chi phí về bộ nhớ?

Cây tìm kiếm nhị phân

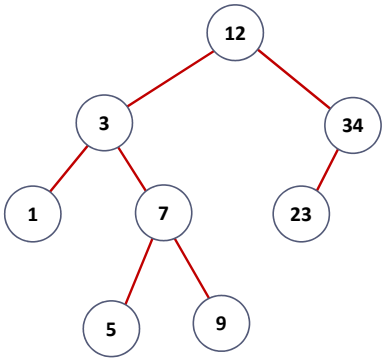
- Cây tìm kiếm nhị phân – binary search tree (BST):
  - Thời gian thực hiện tìm kiếm nhanh ($O(\log n)$)
  - Thêm, xóa các phần tử dễ dàng
- Cây tìm kiếm nhị phân là cây nhị phân rỗng hoặc mỗi nút có một giá trị khóa thỏa mãn các điều kiện sau:
  - Giá trị khóa của nút gốc (nếu tồn tại) lớn hơn giá trị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

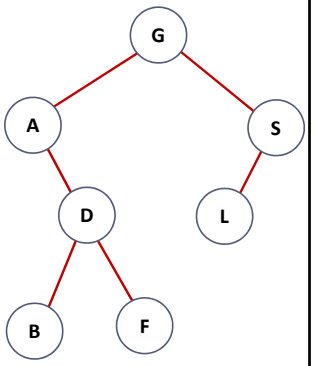
Cây tìm kiếm nhị phân



- 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

Cây tìm kiếm nhị phân





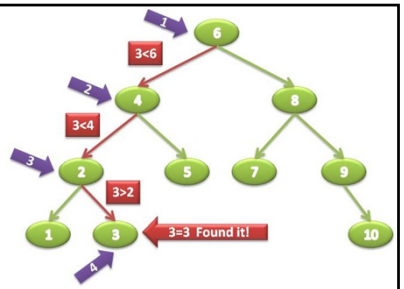
Các nút trên cây nhị phân phải có khóa phân biệt !

Cây tìm kiếm nhị phân

- Biểu diễn cây tìm kiếm nhị phân: giống biểu diễn cây nhị phân thông thường
- Biểu diễn bằng cấu trúc liên kết

```
struct TreeNode
{
   DATA_TYPE info;
   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 (lớn hơn) khóa của nút gốc

Thao tác tìm kiếm trên cây

- Cài đặt đệ quy

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

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

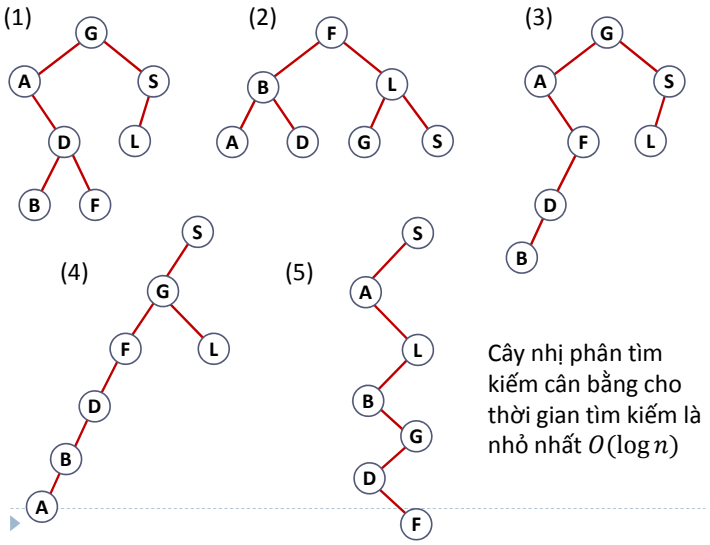
Cây tìm kiếm nhị phân

- Một số thao tác cơ bản của ADT cây tìm kiếm nhị phân
  - Search_for_node(): tìm kiếm xem một giá trị khóa có xuất hiện trên cây không
  - Insert(): Chèn một nút mới vào cây
  - Remove(): Gỡ bỏ một nút trên cây

Thao tác tìm kiếm trên cây

- Cài đặt không đệ quy

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



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

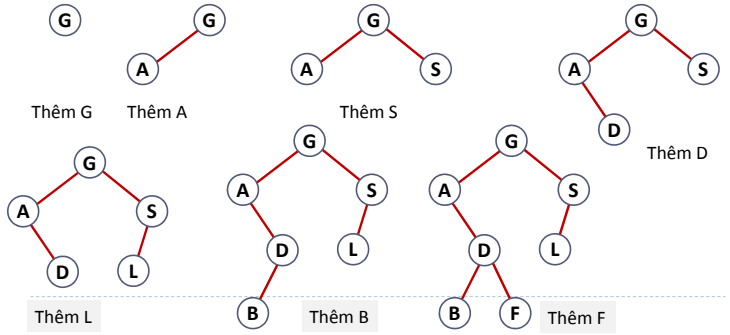
Thêm nút mới vào cây

- 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

- 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 nút mới vào cây

- Cài đặt đệ quy

```
int insert(NODE *&root, int value)
{
  if(root==NULL)
  {
    root = (NODE*)malloc(sizeof(NODE));
    root->info = value;
    root->leftChild = NULL;
    root->rightChild = NULL;
    return 0;// success
  }
  else if(value < root->info) insert(root->leftChild,value);
  else if(value > root->info) insert(root->rightChild,value);
  else return -1;//duplicate
}
```

Thêm nút mới vào cây

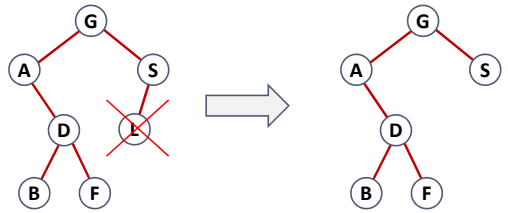
- Cài đặt không đệ quy

```
int insert(NODE *&root, int value)
{
  NODE *pRoot = root;
  while(pRoot!=NULL && pRoot->info!=key)
    if(key<pRoot->info) pRoot=pRoot->leftChild;
    else pRoot=pRoot->rightChild;

  if(pRoot!=NULL) return -1;//duplicate
  else
  {
    pRoot = (NODE*)malloc(sizeof(NODE));
    pRoot->info = value;
    pRoot->leftChild = NULL;
    pRoot->rightChild = NULL;
    return 0;// success
  }
}
```

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:



Thêm nút mới vào cây

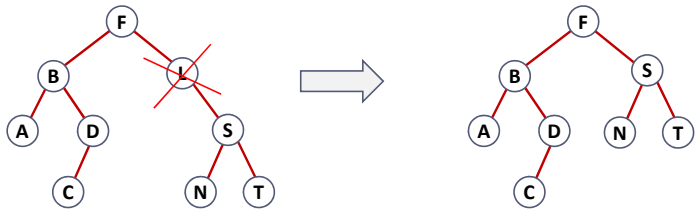
- **Nhận xét**: Thời gian thực hiện của thuật toán phụ thuộc vào dạng của cây
  - Cây cân bằng : thời gian cỡ $O(\log n)$
  - Cây bị suy biến (lệch trái, phải hoặc zig-zag): thời gian cỡ $O(n)$

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

  - **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: $1.39\, n \log n + O(n)$

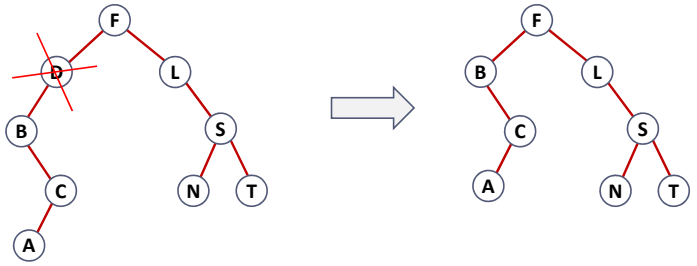
Loại bỏ nút khỏi cây

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

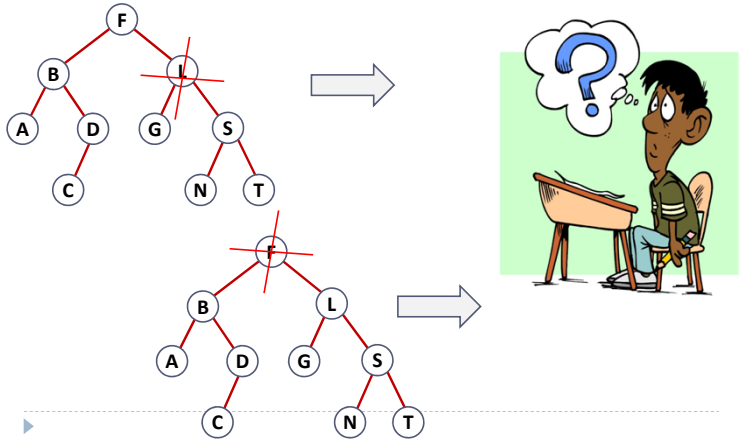


Loại bỏ nút khỏi cây

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

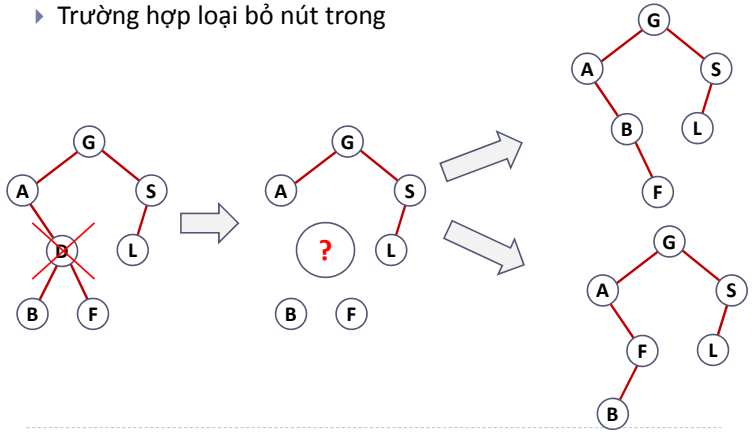


Loại bỏ nút khỏi cây



Loại bỏ nút khỏi cây

- Trường hợp loại bỏ nút trong



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->info = ptr->info;
    if(preP == root) root->leftChild = ptr->leftChild;
    else preP->rightChild = ptr->leftChild;
  }
  free(ptr);
  return 0;// remove success
}
```

Loại bỏ nút khỏi cây

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