Chương 6 : Tìm kiếm

Trịnh Anh Phúc [1]

[1] Bộ môn Khoa Học Máy Tính, Viện CNTT & TT,
Trường Đại Học Bách Khoa Hà Nội.

Ngày 28 tháng 4 năm 2014

Giới thiệu

1. Tìm kiếm tuần tự và tìm kiếm nhị phân
   - Tìm kiếm tuần tự
   - Tìm kiếm nhị phân
2. Cây nhị phân tìm kiếm
   - Định nghĩa
   - Biểu diễn cây nhị phân tìm kiếm
   - Sắp xếp nhờ sử dụng BST
   - Cây nhị phân tìm kiếm cân bằng
3. Tìm kiếm xâu mẫu
   - Thuật toán trực tiếp
   - Thuận toán Boyer-Moore
   - Thuận toán Rabin-Karp
   - Thuận toán Knuth-Morris-Pratt
4. Bảng băm (Mappping and Hashing)
   - Đặt vấn đề
   - Địa chỉ trực tiếp
   - Hàm băm
5. Tổng kết

Tìm kiếm tuần tự và tìm kiếm nhị phân

Định nghĩa bài toán tìm kiếm

Bài toán đặt ra Cho danh sách list[0...n-1] và phần tử target, ta cần tìm vị trí i sao cho list[i] = target hoặc trả lại giá trị -1 nếu không có phần tử như vậy trong danh sách

Tìm kiếm tuần tự và tìm kiếm nhị phân

Tìm kiếm tuần tự (linear search or sequential search)

Thuật toán tìm kiếm tuần tự được thực hiện theo ý tưởng sau đây : Bắt đầu từ phần tử đầu tiên, duyệt qua từng phần tử cho đến khi tìm được phần tử đích hoặc kết luận không tìm được.

-7	9	-5	2	8	3	-4

Độ phức tạp : $O(n)$

```
int linearSearch(dataArray list, int size, dataElem target){
    int i;
    for(i = 0;i<size;i++){
        if(list[i]==target) return i;
    }
    return -1;
}
```

Tìm kiếm nhị phân (binary search)

Điều kiện để thực hiện tìm kiếm nhị phân là :

- Danh sách phải được sắp xếp
- Phải cho phép truy vấn trực tiếp

Mã nguồn ngôn ngữ C

```
int binarySearch(dataArray list, int size, dataElem target){
    int lower = 0, upper = size-1, mid;
    while(lower<=upper){
        mid = (upper+lower)/2;
        if(list[mid]>target) upper = mid - 1;
            else if(list[mid]<target) lower = mid+1;
                else return mid;
    }
    return -1;
}
```

1. Tìm kiếm tuần tự và tìm kiếm nhị phân
   - Tìm kiếm tuần tự
   - Tìm kiếm nhị phân
2. Cây nhị phân tìm kiếm
   - Định nghĩa
   - Biểu diễn cây nhị phân tìm kiếm
   - Sắp xếp nhờ sử dụng BST
   - Cây nhị phân tìm kiếm cân bằng
3. Tìm kiếm xâu mẫu
   - Thuật toán trực tiếp
   - Thuận toán Boyer-Moore
   - Thuận toán Rabin-Karp
   - Thuận toán Knuth-Morris-Pratt
4. Bảng băm (Mappping and Hashing)
   - Đặt vấn đề
   - Địa chỉ trực tiếp
   - Hàm băm
5. Tổng kết

Cây nhị phân tìm kiếm

Đặt vấn đề

Ta cần xây dựng cấu trúc dữ liệu biểu diễn các tập động

- Các phần tử có khóa (key) và thông tin (satellite data)
- Tập động cần hỗ trợ các truy vấn (queries) như :
  - Search(S,k) : Tình phần tử có khóa k
  - Minimum(S), Maximum(S) : Tìm phần tử có khóa nhỏ nhất, lớn nhất
  - Predecessor(S,x), Successor(S,x) : Tìm phần tử kế cận trước, kế cận sau

  đồng thời cũng hỗ trợ các thao tác biến đổi (modifying operations) như :
  - Insert(S,x) : Bổ sung (chèn)
  - Delete(S,x) : Loại bỏ (xóa)

Cây nhị phân tìm kiếm là cấu trúc dữ liệu quan trọng để biểu diễn tập động, trang đó tất cả các thao tác đều thực hiện với thời gian $O(h)$ trong đó h là chiều cao của cây.

Cây nhị phân tìm kiếm

Định nghĩa

Cây nhị phân tìm kiếm (Binary Search Tree - BST) là cây nhị phân có các tính chất sau :

left	key	right	parent

- mỗi nút ngoài thông tin đi kèm có thêm các trường :
  - *left* : con trỏ đến con trái
  - *right* : con trỏ đến con phải
  - *parent* : con trỏ đến cha (tùy chọn)
  - *key* : khóa
- giả sử x là gốc của một cây con, khi đó
  - với mọi nút y thuộc cây con trái của x thì : $key(y) < key(x)$
  - với mọi nút y thuộc cây con phải của x thì : $key(y) > key(x)$

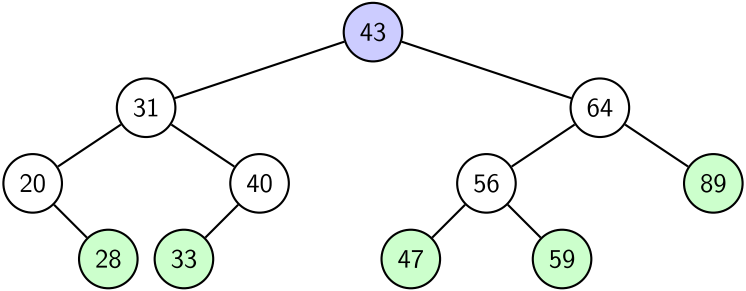
Cây nhị phân tìm kiếm

Các phép toán với cây nhị phân tìm kiếm

- Tìm kiếm (search) : Tìm kiếm một phần tử khóa trước
- Tìm cực tiểu, cực đại (maximum, minimum) : Tìm phần tử với khóa nhỏ nhất (lớn nhất) trên cây
- Kế cận sau, kế cận trước (predecessor, successor) : Tìm phân tử kế cận sau (kế cận trước) của một phần tử trên cây
- Chèn (insert) : Bổ sung vào cây một phần tử với khóa cho trước
- Xóa (delete) : Loại bỏ khỏi cây một phần tử khóa cho trước

Cây nhị phân tìm kiếm

Ví dụ minh họa về cây nhị phân tìm kiếm



Duyệt BST theo thứ tự giữa thì ra dãy khóa được sắp xếp

$$20, 28, 31, 33, 40, 43, 47, 56, 59, 64, 89$$

Cây nhị phân tìm kiếm

Biểu diễn cây nhị phân tìm kiếm

Với khóa số nguyên

```
struct TreeNodeRec{
    int key;
    struct TreeNodeRec *leftPtr;
    struct TreeNodeRec *rightPtr;
};
typedef struct TreeNodeRec TreeNode;
```

Với khóa là chuỗi ký tự

```
# define MAXLEN 15
struct TreeNodeRec{
    char key[MAXLEN];
    struct TreeNodeRec *leftPtr;
    struct TreeNodeRec *rightPtr;
};
typedef struct TreeNodeRec TreeNode;
```

Cây nhị phân tìm kiếm

Các phép toán cơ bản

- **makeTreeNode(value)** - Tạo một nút với khóa cho bởi value
- **search(nodePtr,k)** - Tìm kiếm nút có giá trị khóa bằng k trên BST trỏ bởi nodePtr;
- **find-min(nodePtr)** - Trả lại nút có khóa có giá trị nhỏ nhất trên BST
- **find-max(nodePtr)** - Trả lại nút có khóa có giá trị lớn nhất trên BST
- **successor(nodePtr, x)** - Trả lại nút kế cận sau nút x
- **predecessor(nodePtr, x)** - Trả lại nút kế cận trước nút x
- **insert(nodePtr, item)** - Chèn một nút với khóa cho bởi item vào BST
- **delete(nodePtr, item)** - Xóa nút có giá trị bằng khóa trên BST

Cây nhị phân tìm kiếm

Các mô tả trên C đối với các phép toán

```
struct TreeNodeRec {
    float key;
    struct TreeNodeRec *leftPtr;
    struct TreeNodeRec *rightPtr;
};
typedef struct TreeNodeRec TreeNode;
TreeNode* makeTreeNode(float value);
TreeNode* delete(TreeNode* T, float x);
TreeNode* findmin(TreeNode* T);
TreeNode* findmax(TreeNode* T);
TreeNode* insert(TreeNode* nodePtr, float item);
TreeNode* search(TreeNode* nodePtr, float item);
void PrintInorder(const TreeNode* nodePtr);
```

Cây nhị phân tìm kiếm

Các mô tả trên C đối với các phép toán

```
struct TreeNodeRec {
    float key;
    struct TreeNodeRec *leftPtr;
    struct TreeNodeRec *rightPtr;
};
typedef struct TreeNodeRec TreeNode;
TreeNode* makeTreeNode(float value);
TreeNode* delete(TreeNode* T, float x);
TreeNode* findmin(TreeNode* T);
TreeNode* findmax(TreeNode* T);
TreeNode* insert(TreeNode* nodePtr, float item);
TreeNode* search(TreeNode* nodePtr, float item);
void PrintInorder(const TreeNode* nodePtr);
```

Trong các thao tác trên, thao tác loại bỏ (delete) một nút trong cây là phức tạp nhất. Trong khi thao tác tìm kiếm (search) lại đặc trưng nhất cùng với thao tác chèn (insert) một phần tử vào cây BST.

Cây nhị phân tìm kiếm

Thuật toán bổ sung trên BST

Thuật toán bổ sung

- Tạo nút mới chứa phần tử cần chèn
- Di chuyển trên cây từ gốc để tìm cha của nút mới : So sánh khóa của nút mới với nút đang xét (bắt đầu là gốc của cây), nếu khóa của phần tử cần chèn lớn hơn (nhỏ hơn) khóa của nút đang xét thì rẽ theo con phải (con trái) của nút đang xét. Nếu gặp NULL thì dừng, nút đang xét là cha cần tìm.
- Gắn nút con là nút con của nút cha tìm được. Chú ý là nút mới luôn là nút lá.

Cây nhị phân tìm kiếm

Thuật toán bổ sung trên BST (tiếp)

Mã giả của giải thuật bổ sung

Function Insert(T, item)

1. **if** (T=NULL) **then** T $\leftarrow$ makeTreeNode(item)
2.   **else if** (item $<$ T.key) **then**
3.   T $\leftarrow$ Insert(T.left,item)
4.   **else if** (item $>$ T.key) **then**
5.   T $\leftarrow$ Insert(T.right,item) **endif**
6.   **endif**
7. **endif**
8. **return** T

End

Cây nhị phân tìm kiếm

Thuật toán bổ sung trên BST (tiếp)

Cài đặt ngôn ngữ lập trình C

```
TreeNode *insert(TreeNode * nodePtr, float item){
    if(nodePtr==NULL) nodePtr = makeTreeNode(item);
    else if(item < nodePtr->key)
        nodePtr->leftPtr = insert(nodePtr->leftPtr, item);
        else if(item > nodePtr->key)
            nodePtr->rightPtr = insert(nodePtr->rightPtr, item);
    return nodePtr;
}
```

Cây nhị phân tìm kiếm

Thuật toán tìm kiếm trên BST

Để tìm kiếm một khóa trên cây BST ta tiến hành như sau :

- Nếu khóa cần tìm nhỏ hơn nút hiện tại thì tìm tiếp cây con trái
- ngược lại, tìm cây con phải
- ngược lại, nếu bằng giá trị tại nút hiện tại thì đưa ra
- ngược lại, trả về giá trị NULL không tìm thấy

Cây nhị phân tìm kiếm

Thuật toán tìm kiếm trên BST (tiếp)

Mã giả của thuật toán

```
Function search(T, target)
    if (T not NULL) then
        if (target < T.key) then
            T ← search(T.left, target)
        else
            if (target > T.key) then
                T ← search(T.right, target) endif
        endif
    endif
    return T
End
```

Cây nhị phân tìm kiếm

Thuật toán tìm kiếm trên BST (tiếp)

```
TreeNode *search(TreeNode *nodePtr, float target){
    if(nodePtr!=NULL){
    if(target < nodePtr->key){
        nodePtr = search(nodePtr->leftPtr, target);
    }else{
        if(target > nodePtr->key)
            nodePtr = search(nodePtr->rightPtr, target);
    }
    return nodePtr;
}
```

Cây nhị phân tìm kiếm

Thuật toán tìm phân tử lớn nhất, nhỏ nhất trên BST

Việc tìm phần tử nhỏ nhất (lớn nhất) trên cây nhị phân tìm kiếm có thể thực hiện nhờ việc di chuyển trên cây

- Để tìm phần tử nhỏ nhất, ta đi theo con trái đến khi gặp NULL
- Để tìm phần tử lớn nhất, ta đi theo con phải đến khi gặp NULL

Cây nhị phân tìm kiếm

Thuật toán tìm phần tử lớn nhất, nhỏ nhất trên BST (tiếp)

Mã giả của hai giải thuật

Function find-min(T)

1. **while** (T.left $\neq$ NULL) **do**
2. T $\leftarrow$ T.left
3. **endwhile**
4. **return** T

End

Function find-max(T)

1. **while** (T.right $\neq$ NULL) **do**
2. T $\leftarrow$ T.right
3. **endwhile**
4. **return** T

End

Cây nhị phân tìm kiếm

Thuật toán loại bỏ trên BST

Khi loại bỏ một nút, cần phải đảm bảo cây thu được vẫn là cây nhị phân tìm kiếm. Vì thế khi xóa cần phải xét cẩn thận các con của nó. Có bốn tình huống xảy ra :

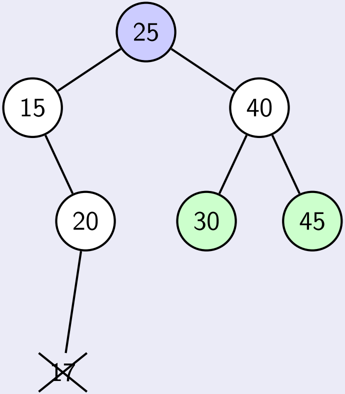
- Tình huống 1 : Nút cần xóa là lá
- Tình huống 2 : Nút cần xóa chỉ có con trái
- Tình huống 3 : Nút cần xóa chỉ có con phải
- Tình huống 4 : Nút cần xóa có hai con

Cây nhị phân tìm kiếm

Thuật toán loại bỏ trên BST (tiếp)

Tình huống 1 : Nút cần xóa x là nút lá

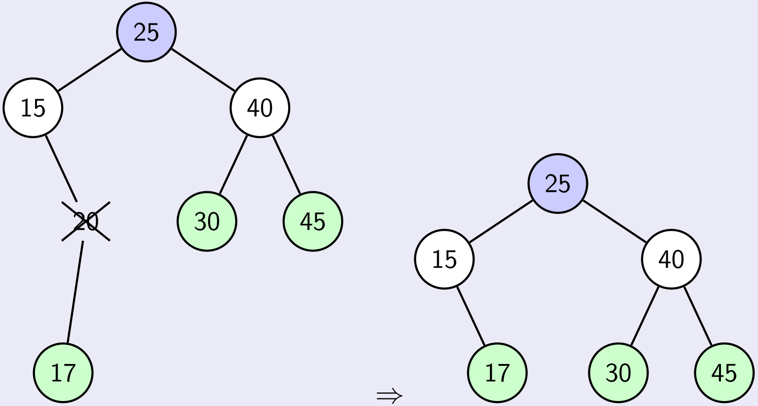
Thao tác : Chữa lại nút cha của x có con rỗng



Thuật toán loại bỏ trên BST (tiếp)

Tình huống 2 : Nút cần xóa x có con trái mà không có con phải

Thao tác : Gắn cây con trái của x vào cha

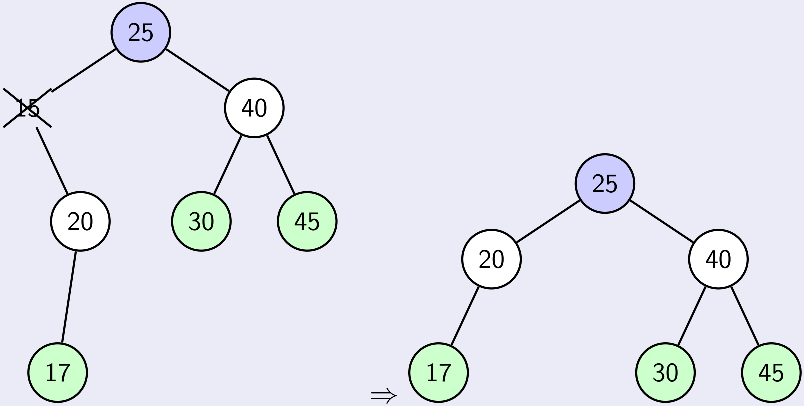


Cây nhị phân tìm kiếm

Thuật toán loại bỏ trên BST (tiếp)

Tình huống 3 : Nút cần xóa x có con phải mà không có con trái

Thao tác : Gắn cây con phải của x vào cha



Cây nhị phân tìm kiếm

Thuật toán loại bỏ trên BST (tiếp)

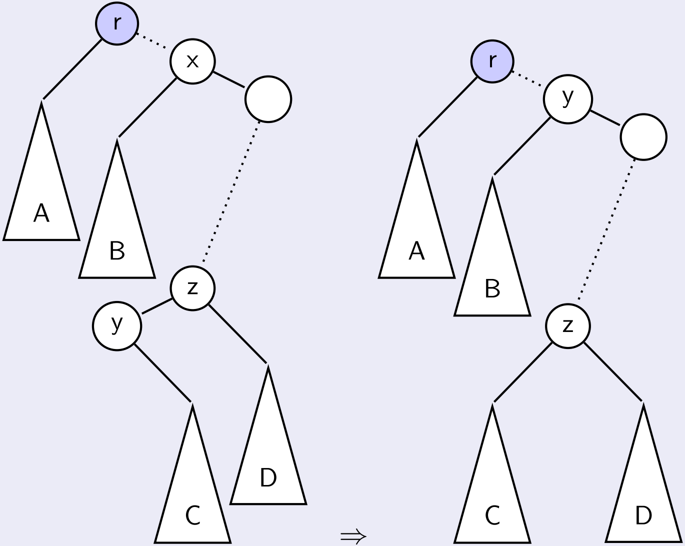
Tình huống 4 : Nút cần xóa x có cả con phải lẫn con trái

Thao tác :

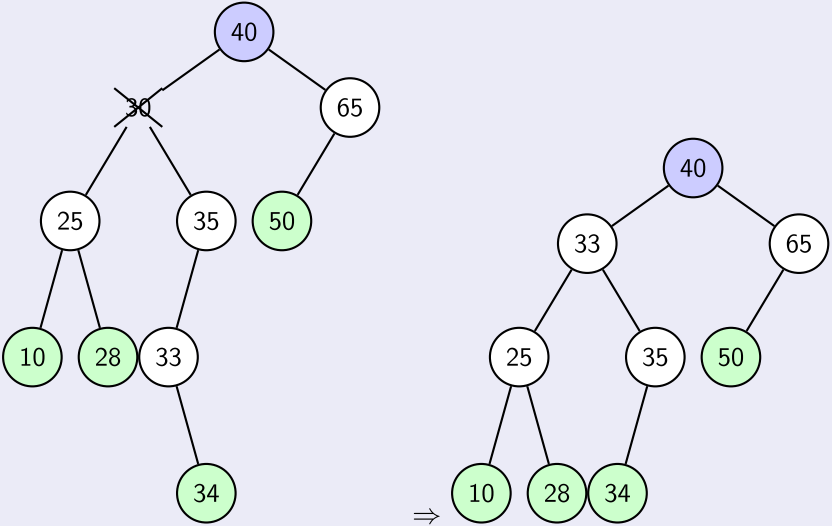
1. Chọn nút y để thế vào chỗ của nút x, nút y sẽ là nút kế tiếp (successor) của x. Như vậy, y là giá trị nhỏ nhất còn lớn hơn x, nói cách khác y là giá trị nhỏ nhất của cây con phải của x.
2. Gỡ nút y khỏi cây
3. Nối con phải của y vào cha của y
4. Thay thế y vào nút cần xóa

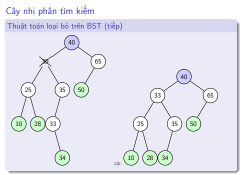
Cây nhị phân tìm kiếm

Thuật toán loại bỏ trên BST (tiếp)



Thuật toán loại bỏ trên BST (tiếp)





Vậy nút thế chỗ của nút 30 cần xóa là nút 33. Nút 33 là nút kế cận của nút 30 khi ta duyệt theo thứ tự giữa để đảm bảo thứ tự giá trị các nút trên cây BST.

Cây nhị phân tìm kiếm

Mã giả của thao tác loại bỏ

Funtion delete(T,x)

if (T=NULL) **then** "Không tìm thấy"
else if (x<T.key) **then** /* Đi bên trái */
    T.left ← delete(T.left,x)
    else if (x>T.key) **then** /* Đi bên phải */
      T.right ← delete(T.right,x)
      else /* Tìm được phần tử cần xóa */
      if (T.left ≠ NULL **and** T.right ≠ NULL) **then**
        /* Tình huống 4 : có cả cây con phải lẫn con trái */
        tmp ← find-min(T.right) /* Thế chỗ ptử min cây con phải */
        T.key ← tmp.key

Cây nhị phân tìm kiếm

Mã giả của thao tác loại bỏ (tiếp)

```
            T.right ← delete(T.right, T.key)
            else /* Có một con hoặc không có con*/
            tmp ← T
            if (T.left = NULL) then T ← T.right /* Chỉ con phải */
            else if (T.right = NULL) then T ← T.left /* Chỉ con trái */
            endif endif
            free(tmp)
          endif endif
      endif
  endif
  return T
End
```

Cây nhị phân tìm kiếm

Thuật toán loại bỏ trên BST (tiếp)

```
TreeNode *delete(TreeNode *T, float x){
    TreeNode tmp;
    if(T==NULL) printf("not found");
    else if(x< T->key) T->leftPtr = delete(T->leftPtr,x);
      else if(x> T->key) T->rightPtr = delete(T->rightPtr,x);
        else /*Tìm được phần tử cần xóa */
        if(T->leftPtr && T->rightPtr){/* Tình huống 4 */
          tmp = findmin(T->right);
          T->key = tmp->key;
          T->rightPtr = delete(T->key,T->rightPtr);
        }else{/* có một con hoặc không có con*/
          tmp = T;
...
```

Cây nhị phân tìm kiếm

Thuật toán loại bỏ trên BST (tiếp)

```
...
        if(T->leftPtr==NULL)/* chỉ có con phải */
            T = T->rightPtr;
        else /* chỉ có con trái */
            T = T->leftPtr;
        free(tmp);
    }
    return(T);
}
```

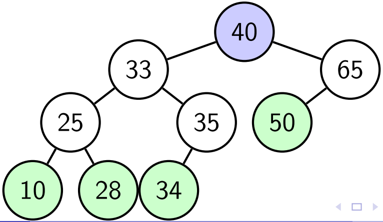
Cây nhị phân tìm kiếm

Sắp xếp nhờ sử dụng BST

Do duyệt cây BST theo thứ tự giữa ra dãy các từ khóa được sắp xếp nên ta có thể sử dụng cây BST để giải quyết bài toán sắp xếp như sau

- Xây dựng cây BST tương ứng với dãy số đã cho bằng cách chèn (insert) từng khóa trong dãy vào cây BST.
- Duyệt cây BST thu được theo thứ tự giữa để đưa ra dãy được sắp xếp.

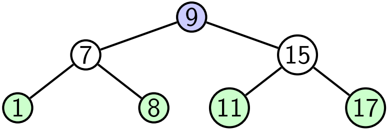
Minh họa cây BST với dãy khóa chưa sắp xếp : 40, 65, 33, 35, 34, 25, 50, 28, 10



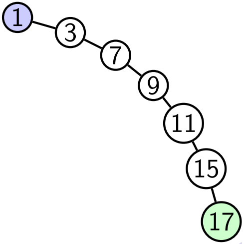
Cây nhị phân tìm kiếm

Phân tích hiệu quả của sắp xếp nhờ sử dụng cây BST

- Tình huống trung bình : $O(n \log n)$ vì chèn phần tử thứ $(i+1)$ tốn quãng thời gian $\log_2(i)$ phép so sánh. Ví dụ như dãy : 9, 15, 7, 8, 1, 11, 17



- Tình huống tồi nhất : $O(n^2)$ bởi vì bổ sung phần tử thứ $(i+1)$ tốn quãng i phép so sánh. Ví dụ dãy đã đc sắp xếp : 1, 3, 7, 9, 11, 15, 17



Sắp xếp nhờ sử dụng BST (tiếp)

Độ phức tạp trung bình của các thao tác Ta biết được rằng độ cao trung bình của cây BST là : $h = O(\log n)$ từ đó suy ra độ phức tạp trung bình của các thao tác với BST

- Chèn $O(\log n)$
- Xóa $O(\log n)$
- Tìm giá trị lớn nhất $O(\log n)$
- Tìm giá trị nhỏ nhất $O(\log n)$
- Sắp xếp $O(n \log n)$

Tất nhiên trường hợp tồi nhất là khi cây nhị phân BST bị mất cân đối do dãy đã được sắp xếp làm tối đa hóa chiều cao của cây $h = n$, như ví dụ ở slice trước

Cây nhị phân tìm kiếm

Sắp xếp nhờ sử dụng BST (tiếp)

Vấn đề đặt ra : Có cách nào để tạo ra một cây nhị phân BST sao cho chiều cao của cây là nhỏ nhất có thể, hay nói cách khác chiều cao $h = \log n$. Có hai cách tiếp cận để nhằm đảm bảo độ cao của cây là nhỏ nhất $O(\log n)$

- Luôn giữ cho cây cân bằng tại mọi thời điểm (AVL tree)
- Thỉnh thoảng lại kiểm tra lại xem cây có "quá mất cân bằng" không (Splay tree).

Trong giáo trình này ta chỉ đề cập đến cây nhị phân tìm kiếm cân bằng AVL

Cây nhị phân tìm kiếm cân bằng

Cây nhị phân tìm kiếm cân bằng

Định nghĩa Cây AVL (Adelson-Velskii-Landis) là cây nhị phân tìm kiếm thỏa mãn tính chất

- Chiều cao của cây con trái và cây con phải của nuts gốc chỉ sai khác nhau không quá một đơn vị.
- Cả cây con phải và cây con trái cũng đều là AVL

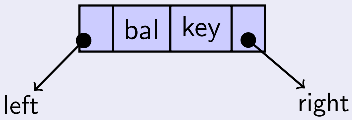
Tính chất : Chênh lệch độ cao của cây con trái và cây con phải của một nút bất kỳ trong cây là không quá một.
Hệ số cân bằng (balance factor) : của nút x, ký hiệu $bal(x)$, là bằng hiệu giữa chiều cao của cây con phải và cây con trái của x.

Cây nhị phân tìm kiếm cân bằng

Biểu diễn cây nhị phân tìm kiếm cân bằng

Biểu diễn cấu trúc cây AVL trong ngôn ngữ C

```
struct treeNode{
    int bal;/* Hệ số cân bằng */
    float key;
    struct TreeNode* left;
    struct TreeNode* right;
}
typedef struct TreeNode AvlTree;
```



Cây nhị phân tìm kiếm cân bằng

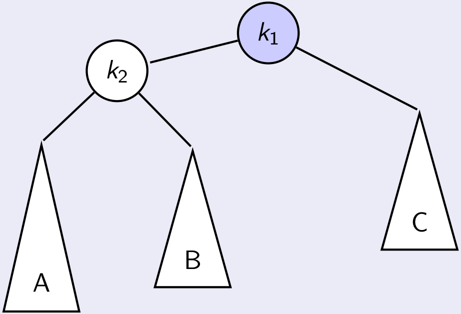
Khôi phục tính cân bằng của cây AVL

- Trước khi khôi phục tính chất này cây đã cân bằng
- Sau khi thực hiện thao tác bổ sung hay loại bỏ cây có thể trở thành mất cân bằng
- Chiều cao của cây con chỉ có thể hoặc giảm nhiều nhất là 1, vì thế nếu xảy ra mất cân bằng thì chênh lệch chiều cao giữa hai cây con chỉ có thể tối đa là 2.

Có tất cả 4 tình huống với chênh lệch chiều cao là 2

Khôi phục tính cân bằng của cây AVL

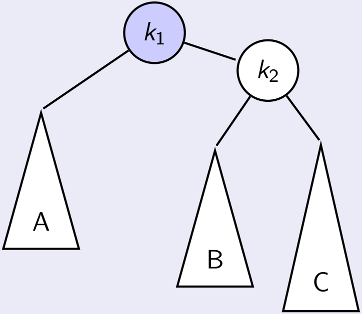
Tình huống 1 : Cây con trái cao hơn cây con phải bởi cây con trái của con trái



Cây nhị phân tìm kiếm cân bằng

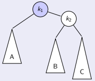
Khôi phục tính cân bằng của cây AVL (tiếp)

Tình huống 2 : Cây con phải cao hơn cây con trái bởi cây con phải của con phải



Khôi phục tính cân bằng của cây AVL (tiếp)

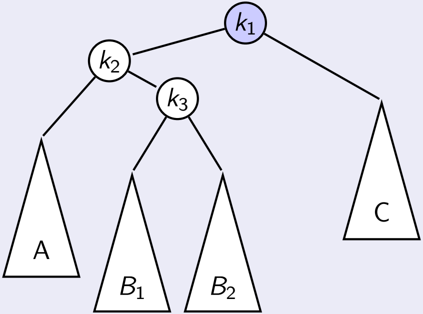
Tình huống 2 : Cây con phải cao hơn cây con trái bởi cây con phải của con phải



Trong tình huống 1 và 2, ta khôi phục tính cân bằng nhờ sử dụng phép quay đơn : Quay phải hoặc quay trái

Khôi phục tính cân bằng của cây AVL (tiếp)

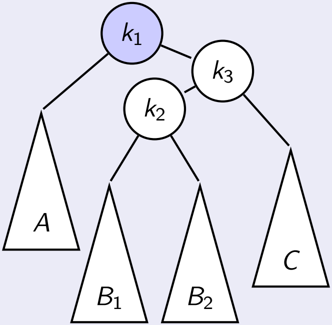
Tình huống 3 : Cây con trái cao hơn cây con phải nguyên do bởi cây con phải của con trái

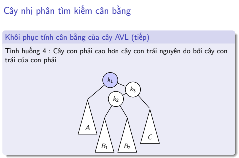


Cây nhị phân tìm kiếm cân bằng

Khôi phục tính cân bằng của cây AVL (tiếp)

Tình huống 4 : Cây con phải cao hơn cây con trái nguyên do bởi cây con trái của con phải

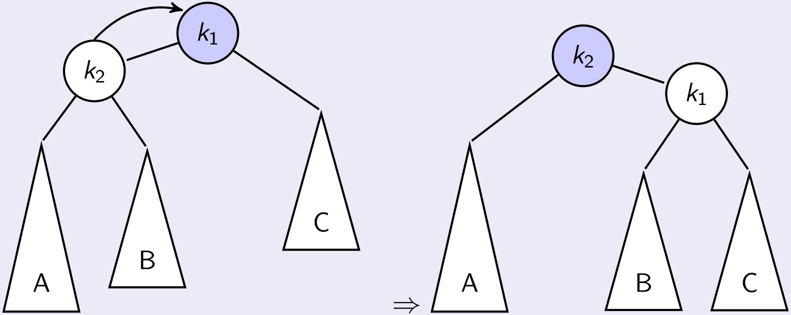




Trong tình huống 3 và 4, ta khôi phục tính cân bằng nhờ sử dụng phép quay kép : Quay phải rồi quay trái hoặc quay trái rồi quay phải

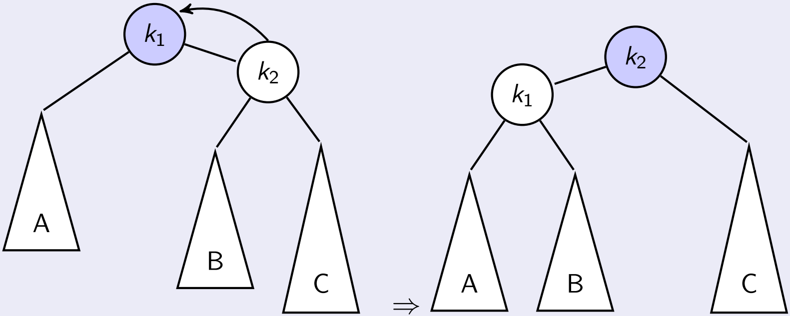
Cây nhị phân tìm kiếm cân bằng

Khôi phục tính cân bằng của cây AVL : Tình huống 1



Cây nhị phân tìm kiếm cân bằng

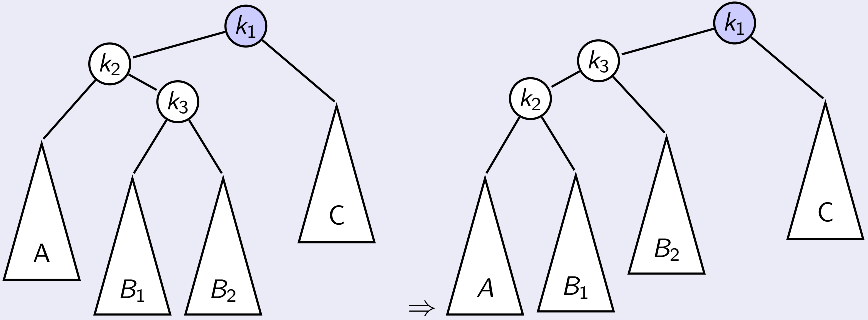
Khôi phục tính cân bằng của cây AVL : Tình huống 2



Cây nhị phân tìm kiếm cân bằng

Khôi phục tính cân bằng của cây AVL : Tình huống 3

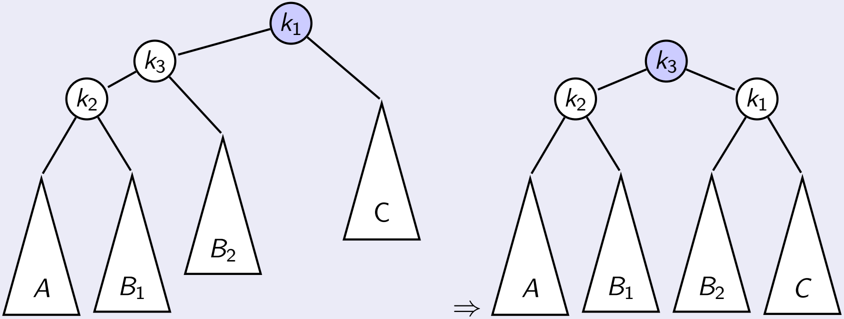
Trước tiên là quay trái quanh k_2



Cây nhị phân tìm kiếm cân bằng

Khôi phục tính cân bằng của cây AVL : Tình huống 3 (tiếp)

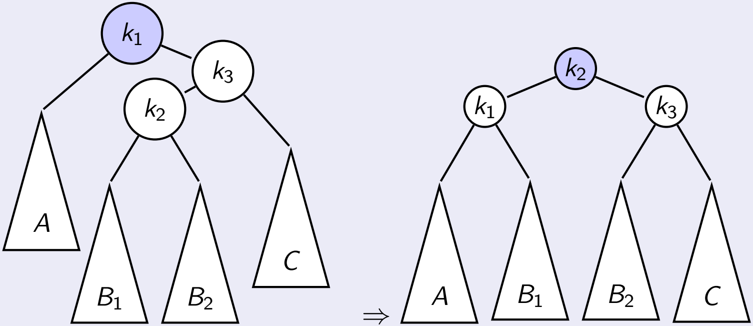
Quay phải quanh k_1



Cây nhị phân tìm kiếm cân bằng

Khôi phục tính cân bằng của cây AVL : Tình huống 4

Quay phải quanh k_3, sau đó quay trái quanh k_1



1

[1]Xem thêm về cây đỏ-đen (Red-Black tree)

Tìm kiếm xâu mẫu - string searching

Phát biểu bài toán

- **Xâu (String)** T là một dãy ký hiệu lấy từ bảng chữ cái (alphabet) $\sum$. Ký hiệu $T[i \cdots j]$ là xâu con của T bắt đầu từ vị trí i kết thúc ở vị trí j.

$$\overbrace{a_1 a_2 \cdots a_{i-1}\, \underbrace{a_i a_{i+1} \cdots a_{j-1} a_j}_{T[i \cdots j]}\, a_{j+1} \cdots a_{n-1} a_n}^{T[1 \cdots n]}$$

- **Trượt** Cho T_1 và T_2 là hai xâu, trong đó chiều dài hai xâu $|T_1| = m$ và $|T_2| = n$ với $m < n$. Ta nói T_1 xuất hiện nhờ trượt đến s trong T_2 nếu $T_1[1 \cdots m] = T_2[s+1 \cdots s+m]$

$$\underbrace{a_1 \cdots \overbrace{a_{s+1} \cdots a_{s+m}}^{T_1} \cdots a_n}_{T_2}$$

Tìm kiếm xâu mẫu - string searching

Phát biểu bài toán (tiếp)

- **Ví trị khớp và không khớp** Giả sử T_1 và T_2 là hai xâu. Nếu T_1 xuất hiện nhờ trượt đến s được gọi là vị trí khớp của T_1 trong T_2. Trong trường hợp ngược lại, vị trí s được gọi là ví trí không khớp.

Bài toán tìm kiếm xâu mẫu - the string matching problem
Cho xâu T độ dài $|T| = n$ và xâu mẫu P trong đó $|P| = m$ có độ dài $m << n$. Tìm tất cả vị trí khớp s của P trong T.

Thuật toán trực tiếp - Naive algorithm

Ý tưởng

Trượt đến từng vị trí $s = 0, 1, \cdots, n - m$ với mỗi vị trí kiểm tra xem xâu mẫu có xuất hiện ở vị trí đó hay không.

Mã nguồn ngôn ngữ C

```
void NaiveSM(char *P, int m, char *T, int n) {
    int i,j;
     for(j=0;j<=n-m;j++){
        for(i=0;i<m && P[i]==T[i+j];i++);
        if(i >=m) OUTPUT(j);
     }
}
```

Thuận toán Boyer-Moore

Ý tưởng

Ta hãy xác định yếu tố ký tự tồi trong một xâu mẫu

- trong giải thuật trực tiếp, do việc duyệt từ trái sang phải nên vị trí càng bên phải thì càng tồi.
- nếu ký tự không có trong xâu mẫu thì ta có thể trượt sang vị trí bên phải không cần kiểm tra nữa.

bởi vậy, ta tạo ra hàm last gồm chỉ số vị trí cực phải của các ký tự trong $\sum$ nằm trong xâu mẫu P.

Ví dụ

Cho xâu mẫu của các ký tự $\sum = \{a, b, c, d\}$ gồm 6 phần tử như sau

a	c	a	b	a	c

Như vậy hàm last : last(a) = 5, last(b) = 4, last(c) = 6 và last(d) = 0

Thuận toán Boyer-Moore

Mã giả của giải thuật Boyer-Moore

$s \leftarrow 0$
while $(s \leq n - m)$ **do**
    $j \leftarrow m$
    while $(j > 0$ **and** $T[j + s] = P[j])$ **do** $j \leftarrow j - 1$ **endwhile**
    if (j=0) **then**
        In s là vị trí khớp
        $s \leftarrow s + 1$
    else
        $k \leftarrow last(T[j + s])$
        $s \leftarrow s + max(j - k, 1)$
    endif
endwhile

Thuận toán Rabin-Karp

Ý tưởng

Coi mẫu P là một khóa khi chuyển nó thành số nguyên p, tương tự như vậy ta cũng chuyển đổi các xâu con của $T[1 \cdots n]$ thành các số nguyên tương ứng với $n - m$ vị trí, như vậy ta chuyển đổi các xâu con liên tiếp của T[] thành các số nguyên

- Với $s = 0, 1, \cdots, n - m$ chuyển thành các số nguyên tương đương t_s

Như vậy, mẫu xuất hiện ở vị trí s khi và chỉ khi $p = t_s$

Thuận toán Rabin-Karp

Tính số mẫu p

Giả sử $\sum = \{0, 1\}$ ta có số nguyên p được tính như sau

$$p = 2^{m-1}P[0] + 2^{m-2}P[1] + \cdots + 2P[m-2] + P[m-1]$$

hoặc có thể viết lặp theo sơ đồ Horner

$$p = P[m-1] + 2 * (P[m-2] + 2 * (P[m-2] + \cdots + 2 * P[0]) \cdots)$$

ta có cài đặt trên C đoạn chương trình tính p như sau :

```
p = 0;
for(i=0;i<m;i++)
    p = 2*p + P[i];
```

thủ tục đòi hỏi thời gian tính toán là $O(m)$ nếu phép toán gán tính là $O(1)$

Thuận toán Rabin-Karp

Tính các số nguyên trong xâu T[]

Một cách tương tự, tính (n-m+1) số nguyên t_s từ xâu văn bản thực hiện trong ngôn ngữ C là :

```
for(s=0;s<=n-m;s++){
    t[s] = 0;
    for(i=0;i<m;i++) t[s] = 2*t[s] + T[s+i];
}
```

Việc này đòi hỏi thời gian $O((n-m+1)m)$ với giả thiết các phép toán số học được thực hiện với thời gian O(1) trên đây rõ ràng là công đoạn tốn kém.

Thuận toán Rabin-Karp

Tính các số nguyên trong xâu T[] cải tiến

Ta có thể cải tiến công đoạn trên như sau

```
t[0] = 0;
offset = 1;
for(i=1;i<m;i++) offset = 2*offset;
for(i=0;i<m;i++) t[0] = 2*t[0]+T[i];
for(s=1;s<=n-m;s++) t[s] = 2*(t[s-1]-offset*T[s-1])+T[s+m-1];
```

Việc này đòi hỏi thời gian $O(n + m)$ với giả thiết các phép toán số học được thực hiện với thời gian O(1)

Thuận toán Knuth-Morris-Pratt

Sử dụng hàm bổ trợ (prefix)

Định nghĩa của **prefix** : Xâu W được gọi là prefix của xâu X nếu X=WY với một xâu Y nào đó, ký hiệu là $W \subset X$.
Định nghĩa của **suffix** : Xâu W được gọi là suffix của xâu X nếu X= YW với một xâu Y nào đó, ký hiệu là $W \supset X$.
Ví dụ : W = ab là prefix của X=abefac, trong đó Y = efac. Ngược lại, W = cdaa là suffix của X = acbecdaa, trong đó Y = acbe
Chú ý : Xâu rỗng, ký hiệu ϵ, là prefix và suffix của mọi xâu.

Thuận toán Knuth-Morris-Pratt

Bổ đề

Giả sử $X \supset Z$ và $Y \supset Z$ khi đó

1. nếu $|X| \leq |Y|$ thì $X \supset Y$
2. nếu $|X| \geq |Y|$ thì $Y \supset X$
3. nếu $|X| = |Y|$ thì $Y = X$

Dịch chuyển tối thiểu

Vấn đề đặt ra : Biết rằng prefix P[1..q] của xâu mẫu là khớp với đoạn T[(s+1)..(s+q)] tìm giá trị nhỏ nhất s'>s sao cho :

$$P[1..k] = T[(s'+1)..(s'+k)], \text{ trong đó } s'+k = s+q$$

Khi đó, tại vị trí s', không cần thiết so sánh k ký tự đầu của P với các ký tự tương ứng của T, bởi vì ta biết chắc chúng khớp nhau.

Thuận toán Knuth-Morris-Pratt

Hàm tiền tố

prefix function : $\pi[q]$ là độ dài của prefix dài nhất của P[1..m] đồng thời là suffix thực sự của P[1..q] nghĩa là

$$\pi[q] = \max\{k : k < q \text{ và } P[1..k] \text{ là suffix của } P[1..q]\}$$

Ví dụ

Xét xâu mẫu P = ababababca còn bảng dưới đây cho giá trị của hàm tiền tố

i	1	2	3	4	5	6	7	8	9	10
P[i]	a	b	a	b	a	b	a	b	c	a
π[i]	0	0	1	2	3	4	5	6	0	1

Thuận toán Knuth-Morris-Pratt

Giải thuật tính giá trị hàm prefix

Compute-Prefix-Function(P)

m $\leftarrow$ length(P)
$\pi[1] \leftarrow 0$
k $\leftarrow$ 0
    for q $\leftarrow$ 2 **to** m **do**
        while (k>0) **and** (P[k+1] $\neq$ P[q]) **do** k $\leftarrow \pi$[k] **endwhile**
        if (P[k+1] = P[q]) **then** k $\leftarrow$ k+1 **endif**
        π[q] $\leftarrow$ k
    endfor
return π

End

Thời gian tính giải thuật $\Theta(m)$

Thuận toán Knuth-Morris-Pratt

KMP-Matcher(T,P) // n = |T| và m = |P|

$\pi \leftarrow$ Compute-Prefix-Function(P)
q $\leftarrow$ 0
for i $\leftarrow$ 1 **to** n **do**
    while (q > 0) **and** (P[q+1] $\neq$ T[i]) **do** q $\leftarrow \pi$[q] **endwhile**
    if (P[q+1] = T[i]) **then** q $\leftarrow$ q+1 **endif**
    if (q=m) **then**
        In ra pattern ở vị trí i-m
        q $\leftarrow \pi$[q]
    endif
endfor

End

Thời gian tính giải thuật $\Theta(m + n)$

Bảng băm

Đặt vấn đề

Cho bảng T và các bản ghi x với từ khóa và dữ liệu đi kèm, ta cần hỗ trợ các thao tác sau :

- Chèn : Insert(T,x)
- Xóa : Delete(T,x)
- Search(T,x)

Ta muốn thực hiện thao tác này một cách nhanh chóng mà không phải thực hiện việc sắp xếp các bản ghi. Bảng băm là các tiếp cận giải quyết vấn đề đặt ra.

Chú ý

Ta sẽ chỉ xét các khóa là số nguyên dương

Bảng băm

Ứng dụng

- Xây dựng chương trình của ngôn ngữ lập trình (Compiler) : Ta cần thiết lập bảng ký hiệu trong đó khóa của các phần tử là dãy ký tự
- Bảng băm là cấu trúc dữ liệu hiệu quả để cài đặt từ điển
- Mặc dù trong tình huống xấu nhất việc tìm kiếm đòi hỏi thời gian O(n) giống như tìm kiếm tuyến tính, nhưng trên thực tế bảng băm làm việc hiệu quả hơn nhiều. Với một số giả thiết hợp lý, việc tìm kiếm phần tử trong bảng băm đòi hỏi thời gian O(1)
- Bảng băm có thể xem như sự mở rộng mảng thông thường. Việc **địa chỉ hóa trực tiếp** trong mảng cho phép truy cập đến phần tử bất kỳ trong thời gian O(1)

Bảng băm

Địa chỉ trực tiếp - Direct addressing

Giả thiết rằng :

- Các khóa là các số trong khoảng từ 0 đến m-1
- Các khóa là khác nhau từng đôi một

Ý tưởng : Thiết lập mảng T[0..m-1] trong đó

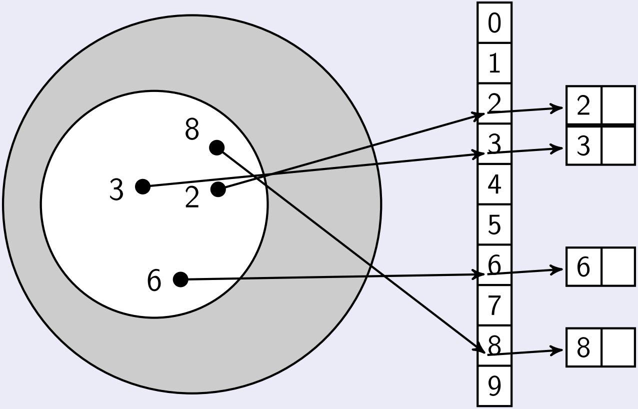
- $T[i] = x$ nếu $x \in T$ và $key[x] = i$
- $T[i] =$ NULL nếu trái lại

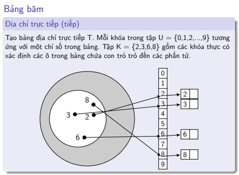
T được gọi là bảng địa chỉ trực tiếp (direct-address table) các phần tử trong bảng T sẽ được gọi là các ô.

Bảng băm

Địa chỉ trực tiếp (tiếp)

Tạo bảng địa chỉ trực tiếp T. Mỗi khóa trong tập U = {0,1,2,...,9} tương ứng với một chỉ số trong bảng. Tập K = {2,3,6,8} gồm các khóa thực có xác định các ô trong bảng chứa con trỏ trỏ đến các phần tử.





Tập U, hay tập khóa toàn bộ, được minh họa bởi hình tròn mầu đen bao ngoài. Tập K, hay tập khóa thực, được minh họa bởi hình tròn mầu trắng nằm trong. Bảng địa chỉ trực tiếp T được minh họa bởi cột giá trị tương ứng của tập khóa toàn bộ U.

Bảng băm

Địa chỉ trực tiếp (tiếp)

Các phép toán được cài đặt một cách trực tiếp

- DIRECT-ADDRESS-SEARCH(T,k)
  return T[k]
- DIRECT-ADDRESS-INSERT(T,k)
  T[key[x]] $\leftarrow$ x
- DIRECT-ADDRESS-DELETE(T,k)
  T[key[x]] $\leftarrow$ NULL

Thời gian thực hiện các phép toán đều là O(1)

Địa chỉ trực tiếp (tiếp)

Hạn chế của địa chỉ trực tiếp là việc chỉ thích hợp nếu biên độ m của các khóa là nhỏ. Giả sử các khóa là số nguyên dương có chiều dài 32 bit thi sao ?

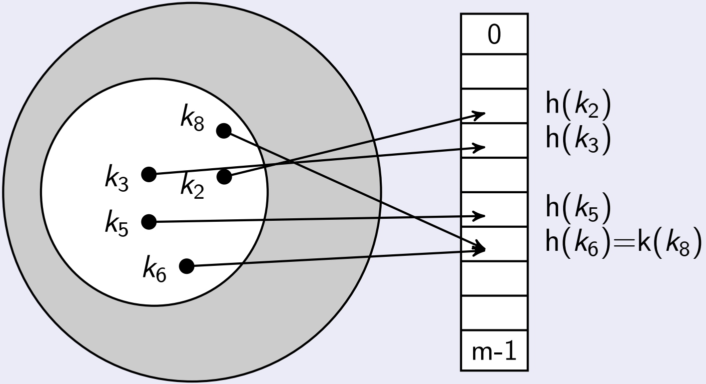
- Vấn đề 1 : bảng địa chỉ trực tiếp sẽ phải có 2^{32} (hơn 4 tỷ) phần tử
- Vấn đề 2 : ngay cả khi bộ nhớ không là vấn đề thì thời gian khởi tạo các phần tử NULL cũng rất tốn kém

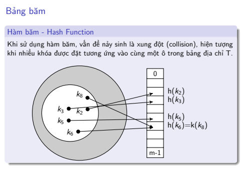
Cách giải quyết là ánh xạ khóa vào khoảng biến đổi nhỏ hơn 0..m-1. Ánh xạ này được gọi là hàm băm (hash function)

Bảng băm

Hàm băm - Hash Function

Khi sử dụng hàm băm, vấn đề nảy sinh là xung đột (collision), hiện tượng khi nhiều khóa được đặt tương ứng vào cùng một ô trong bảng địa chỉ T.





Trong hình minh họa, vị trí xung đột khi hai khóa k_6 và k_8, cùng được trỏ vào cùng ô địa chỉ trong bảng T

Bảng băm

Hàm băm (tiếp)

Để giải quyết xung đột khi dùng hàm băm, ta có hai cách tiếp cận chính để giải quyết xung đột

- Cách 1 : Dùng địa chỉ mở (open addressing)
- Cách 2 : Tạo chuỗi (chaining)

Địa chỉ mở

Trong phương pháp địa chỉ mở, tất cả các phần tử đều được cất giữ vào bảng. Do đó mỗi ô của bảng hoặc là chứa khóa hoặc là NULL. Ý tưởng chính của phương pháp này là

- Để thực hiện việc bổ sung, nếu ô tìm được là bận, ta sẽ tiến hành khảo sát lần lượt (hay còn gọi là dò thử) các ô của bảng cho đến khi tìm được ô rỗng để nạp khóa vào.
- Khi khảo sát, hay dò thử ô còn trống, ta sẽ tìm dọc theo dãy các phép thử khi thực hiện chèn phần tử vào bảng.
  - Nếu tìm được phần tử với khóa đã cho thì trả lại nó.
  - Nếu tìm được con trỏ NULL, thì phần cần tìm không có trong bảng

Để xác định được ô dò thử, ta cần mở rộng định nghĩa hàm băm như sau

$$h : U \times \{0, 1, \cdots, m-1\} \mapsto \{0, 1, \cdots, m-1\}$$

Bảng băm

Địa chỉ mở (tiếp)

Trong phương pháp địa chỉ mở ta đòi hỏi, với mỗi khóa k, dãy dò thử

$$< h(k,0), h(k,1), \cdots, h(k,m-1) >$$

phải là hoán vị của $< h(k,0), h(k,1), \cdots, h(k,m-1) >$ do đó mỗi vị trí trong bảng sẽ được xét như là một ô để chứa khóa mới khi ta tiến hành bổ sung vào bảng

Bảng băm

Địa chỉ mở (tiếp)

Việc bổ sung khóa k sẽ được mô tả trong đoạn mã giả sau

HASH-INSERT(T,k)

1. i ← 0
2. **repeat**
3. j ← h(k,i)
4. **if** (T[j] = NULL) **then**
   T[j] ← k
   return j
5. **else**
   i ← i+1
6. **endif**
7. **until** (i=m)
8. **error** "lỗi tràn bảng băm"

End

Địa chỉ mở (tiếp)

Việc tìm kiếm khóa k sẽ được mô tả trong đoạn mã giả sau

HASH-SEARCH(T,k)

1. i $\leftarrow$ 0
2. **repeat**
3. j $\leftarrow$ h(k,i)
4. **if** (T[j]=j) **then return** j **endif**
5. i $\leftarrow$ i+1
6. **until** ((T[j]=NULL) **or** (i=m))
7. **return** NULL

End

Bảng băm

Địa chỉ mở (tiếp)

Việc tìm kiếm khóa k sẽ được mô tả trong đoạn mã giả sau

```
HASH-SEARCH(T,k)
  i ← 0
  repeat
     j ← h(k,i)
     if (T[j]=j) then return j endif
     i ← i+1
  until ((T[j]=NULL) or (i=m))
  return NULL
End
```

Việc loại bỏ gặp khó khăn hơn. Thông thường ta sẽ đánh dấu loại bỏ chứ không bỏ thực sự.

Địa chỉ mở (tiếp)

Việc dò thường dùng 3 kỹ thuật sau

- Dò tuyến tính (linear probing)

$$h(k,i) = (h'(k) + i) \textbf{ mod } m$$

- Dò toàn phương (quadratic probing)

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \textbf{ mod } m$$

- Băm kép (double hashing)

$$h(k,i) = (h_1(k) + i h_2(k)) \textbf{ mod } m$$

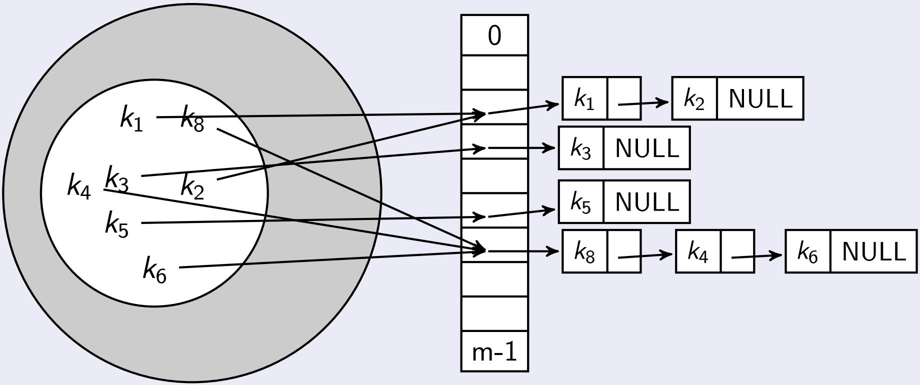
  trong đó $h_1(k)$ và $h_2(k)$ là hàm băm bổ trợ

Với i=0,1,$\cdots$ m-1, h'(k) là hàm băm ban đầu còn c_1 và $c_2 \neq 0$ là hằng số cho trước.

Bảng băm

Tạo chuỗi (chaining)

Theo phương pháp này, ta sẽ tạo ra danh sách móc nối để chứa các phần tử được gắn vào cùng vị trí



Bảng băm

Tạo chuỗi (tiếp)

Vấn đề khi thực hiện việc tạo chuỗi

- Ta nên thực hiện bổ sung phần tử như thế nào ? Trả lời : được bổ sung như danh sách móc nối với hình minh họa trên.
- Ta cần loại bỏ phần tử như thế nào ? Trả lời : Nên dùng danh sách móc nối đơn cho việc loại bỏ dữ liệu được dễ dàng.
- Thực hiện tìm kiếm phần tử khóa cho trước như thế nào ? Trả lời : Chúng ta dùng hàm băm xác định ô trên T, sau đó duyệt tuần tự theo danh sách móc nối để xác định vị trí phần tử.

Bảng băm

Tạo chuỗi (tiếp)

Chọn hàm băm - choising hash funtion :

- Thời gian tính của hàm băm là bao nhiêu ?
- Thời gian tìm kiếm phân tử sẽ như thế nào ?

Do đó này sinh ra hai yêu cầu

- Phải phân bố đều các khóa vào các ô
- Không phụ thuộc vào khuôn mẫu của dữ liệu

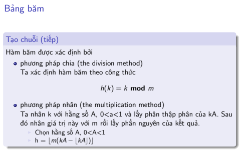
Tạo chuỗi (tiếp)

Hàm băm được xác định bởi

- phương pháp chia (the division method)
  Ta xác định hàm băm theo công thức

$$h(k) = k \textbf{ mod } m$$

- phương pháp nhân (the multiplication method)
  Ta nhân k với hằng số A, 0<a<1 và lấy phân thập phân của kA. Sau đó nhân giá trị này với m rồi lấy phần nguyên của kết quả.
  - Chọn hằng số A, 0<A<1
  - h $= \lfloor m(kA - \lfloor kA \rfloor) \rfloor$



Đối với phương pháp chia, nếu m là luy thừa của 2 chẳng hạn 2^p thì h(k) chính là số p bit cuối của k. Vì thế người ta thường chọn kích thước bảng m là số nguyên tố không quá gần với lũy thừa của 2. Đối với phương pháp nhân, thông thường m $= 2^p$ còn A thì không quá gần 0 hoặc 1 Knuth khuyên A $= (\sqrt{5} - 1)/2$

Tổng kết

- Định nghĩa bài toán tìm kiếm
- Thuật toán tìm kiếm tuyến tính và nhị phân
- Định nghĩa và cài đặt cây nhị phân tìm kiếm
- Định nghĩa cây AVL
- Bài toán tìm kiếm xâu mẫu
- Bảng băm, lưu trữ và tìm kiếm