CHƯƠNG V
CÂY (TREE)

Nội dung chính

1. Tổng quan về cấu trúc cây
2. Cây nhị phân
3. Cây nhị phân tìm kiếm
4. Cây nhị phân tìm kiếm cân bằng

1. Tổng quan về cấu trúc cây

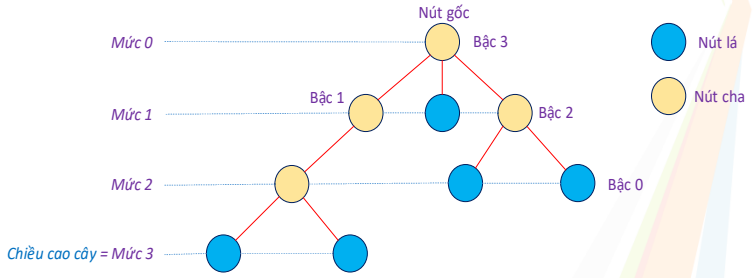
- **Định nghĩa:**

Cây là một tập hợp T các phần tử (gọi là nút của cây), trong đó có một nút đặc biệt gọi là nút gốc, các nút còn lại được chia thành những tập rời nhau T1, T2, …, Tn theo quan hệ phân cấp, trong đó Ti cũng là 1 cây. Mỗi nút ở cấp i sẽ quản lý một số nút ở cấp i+1. Quan hệ này người ta gọi là quan hệ cha – con.

Một số khái niệm cơ bản

- ***Bậc của một nút***: *Là số cây con của nút đó .*
- ***Bậc của một cây***: *Là bậc lớn nhất của các nút trong cây*
- ***Nút gốc (root)***: *Là nút không có nút cha.*
- ***Nút lá (leaf)***: *Là nút có bậc bằng 0 .*
- ***Nút trung gian (interior node):*** nút có bậc khác 0 và không phải là nút gốc
- ***Mức của một nút***:
  - *Mức của nút gốc bằng 0.*
  - *Mức của các nút con bằng mức của nút cha cộng thêm 1.*
- ***Độ dài đường đi từ gốc đến nút x***: *Là số nhánh cần đi qua kể từ gốc đến x.*
- ***Chiều cao của cây***: *Là mức lớn nhất trong cây.*

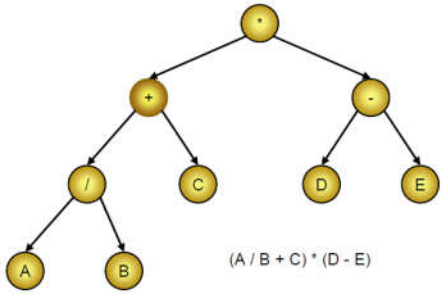
Một số khái niệm cơ bản



5

Một số ví dụ

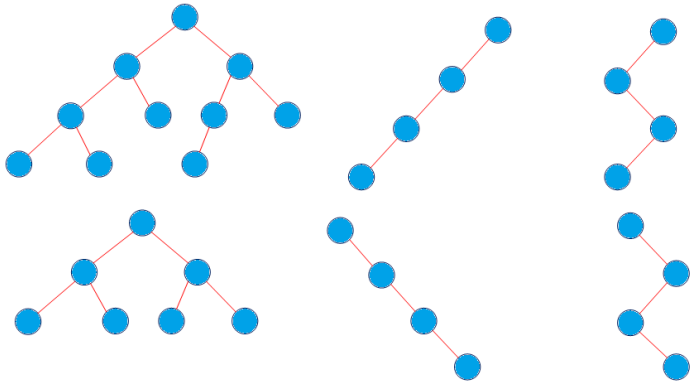
- Sơ đồ tổ chức của 1 công ty
- Mục lục của một cuốn sách
- Gia phả của một họ tộc
- Cấu trúc của thư mục trên ổ đĩa
- Biểu thức toán học cũng có thể biểu diễn bằng cây



6

2. Cây nhị phân (Binary tree)

Cây nhị phân là cây có đặc điểm là mọi nút trên cây chỉ có tối đa hai nhánh con.



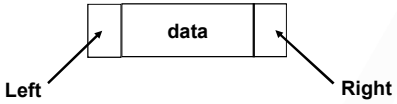
Tính chất của cây nhị phân

- Số lượng tối đa các nút ở mức i của cây nhị phân là 2^i, tối thiểu là 1 ($i \geq 1$).
- Số lượng tối đa các nút trên một cây nhị phân có chiều cao h là $2^{h+1} - 1$, tối thiểu là h+1 ($h \geq 1$).
- Cây nhị phân hoàn chỉnh có n nút thì chiều cao của nó là h = $log_2(n+1) - 1$.

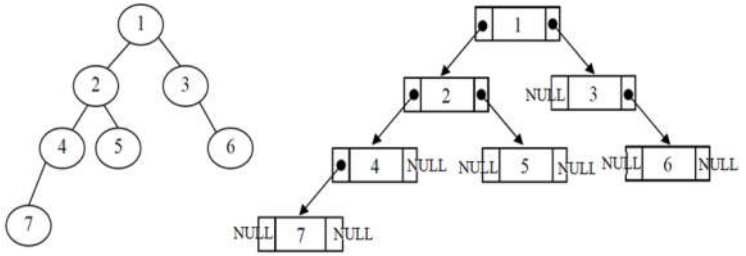
8

Biểu diễn cây nhị phân

- Mỗi nút của cây nhị phân có tối đa 2 nút con, do vậy sử dụng DSLK để biểu diễn cây nhị phân sẽ hữu hiệu nhất.
- Mỗi nút của cây nhị phân khi đó sẽ có 3 thành phần:
  - ***Thành phần data*** *chứa thông tin về nút*
  - ***Con trỏ Left*** *trỏ đến nút con bên trái*
  - ***Con trỏ Right*** *trỏ đến nút con bên phải*
- Nếu nút có ít hơn 2 nút con thì các thành phần Left hoặc Right sẽ trỏ tới NULL



Biểu diễn cây nhị phân



Cài đặt cây nhị phân bằng danh sách liên kết

Biểu diễn cây nhị phân

- **Cấu trúc dữ liệu của nút**

```
struct Node{
    int Data;
    Node *Left, *Right;
};
```

- **Định nghĩa kiểu dữ liệu cây**

```
typedef Node *Tree;
```

- **Khai báo và khởi tạo 1 cây rỗng**

```
Tree T=NULL;
```


Duyệt cây nhị phân

- **Phép duyệt cây nhị phân được chia thành 3 loại:**
  - *Duyệt thứ tự trước (Preorder): NLR hoặc NRL*
  - *Duyệt thứ tự giữa (Inorder): LNR hoặc RNL*
  - *Duyệt thứ tự sau (Posorder): LRN hoặc RLN*
- **Độ phức tạp: $O(\log_2(h))$**

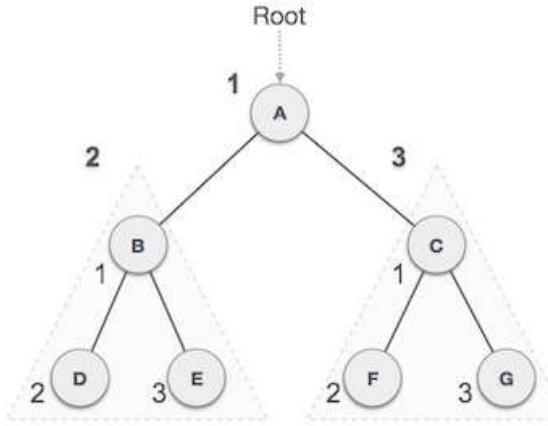
  Trong đó h là chiều cao cây

Duyệt cây nhị phân

- **Duyệt trước: NLR**

```
void NLR(Tree  T) {
   if(T != NULL)   {
         cout<<T->Data;
         NLR (T->Left);
         NLR(T->Right);
   }
}
```


Duyệt NLR



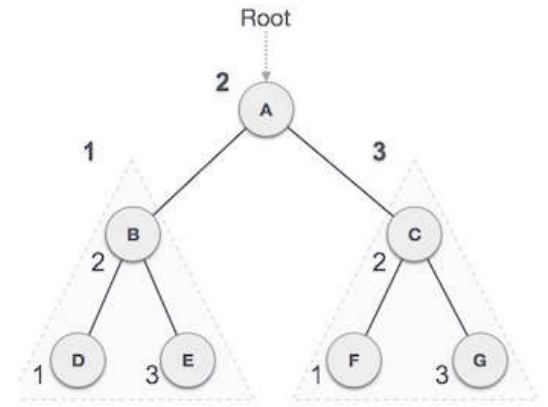
- ***A → B → D → E → C → F → G***

Duyệt cây nhị phân

- **Duyệt giữa: LNR**

```
void LNR(Tree  T) {
   if(T != NULL) {
         LNR(T->Left);
         cout<<T->Data;
         LNR(T->Right);
   }
}
```


Duyệt LNR



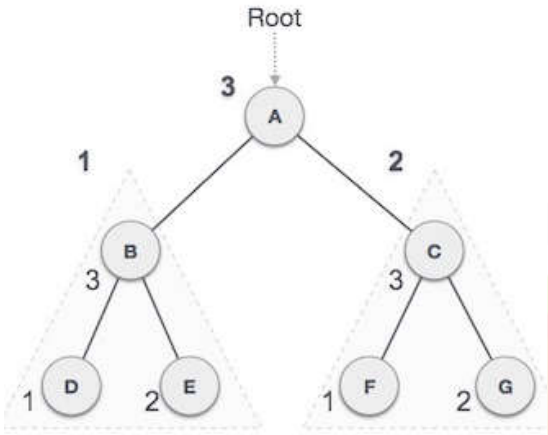
- ***D → B → E → A → F → C → G***

Duyệt cây nhị phân

- **Duyệt sau: LRN**

```
void LRN(Tree T) {
  if(T != NULL) {
        LRN(T->Left);
        LRN(T->Right);
        cout<<T->Data;
  }
}
```

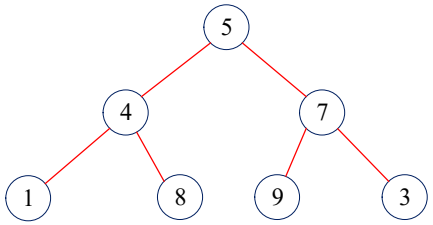

Duyệt LRN



- ***D → E → B → F → G → C → A***

Một số cách biểu diễn cây nhị phân khác

Biểu diễn cây nhị phân bằng mảng:

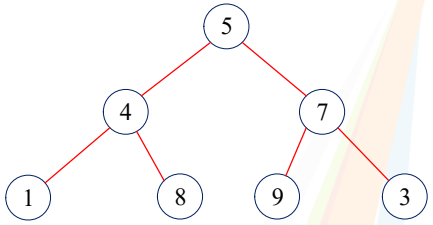


A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
5	4	7	1	8	9	3

Một số cách biểu diễn cây nhị phân khác

Thêm con trỏ trỏ về nút cha:

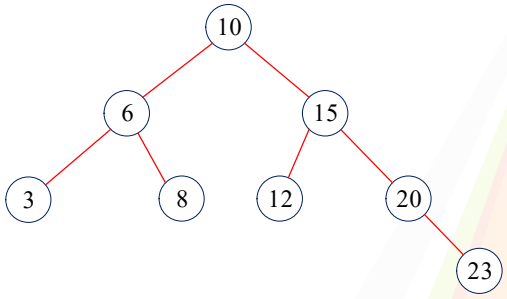
```
struct Node{
  int Data;
  Node *Left;
  Node *Right;
  Node *Parent;
};
```



A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
5	4	7	1	8	9	3
0	5	5	4	4	7	7

3. Cây nhị phân tìm kiếm (BST)

- **Định nghĩa**: Cây nhị phân tìm kiếm (BST - Binary Search Tree) Là cây nhị phân mà khoá tại mỗi nút cây lớn hơn khoá của tất cả các nút thuộc cây con bên trái và nhỏ hơn khoá của tất cả các nút thuộc cây con bên phải.



Khai báo và khởi tạo

- *Cấu trúc của nút*

```
struct Node {
   int Key;//Trường dữ liệu là số nguyên
   Node *Left, *Right;
};
```

- *Định nghĩa kiểu dữ liệu cây*

```
typedef  Node *Tree;
```

- *Khởi tạo cây rỗng*

```
void Create(Tree &T) {
      T=NULL;
}
```


Các thao tác cơ bản trên cây NPTK

- *Tạo 1 nút có trường Key bằng x.*
- *Thêm 1 nút vào cây nhị phân tìm kiếm.*
- *Xoá 1 nút có Key bằng x trên cây.*
- *Tìm 1 nút có khoá bằng x trên cây.*

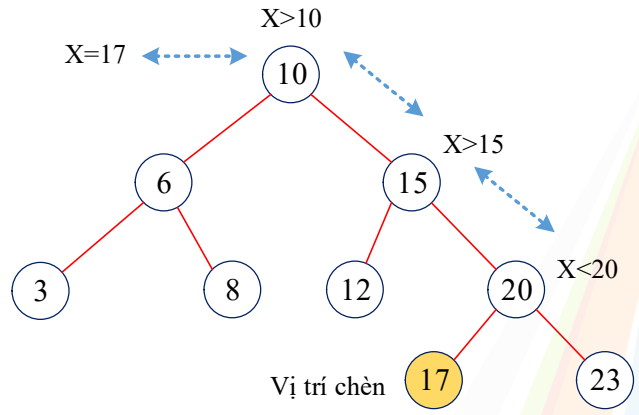
Tạo một nút có khóa bằng x

```
Node* CreateNode(int x) {
      Node *p=new Node;
      p->Key = x;
      p->Left = p->Right =NULL;
      return p;
}
```

Thêm một nút có khóa x vào cây

```
int InsNode(Tree &T, int x) {
    if(T!=NULL) {
       if(T->Key == x)
          return 0;//x trùng
       if(T->Key > x)
          return InsNode(T->Left, x);
       else
          return InsNode(T->Right, x);
    }
    T = CreateNode(x);
    return 1;
}
```


Minh họa thêm phần tử vào cây



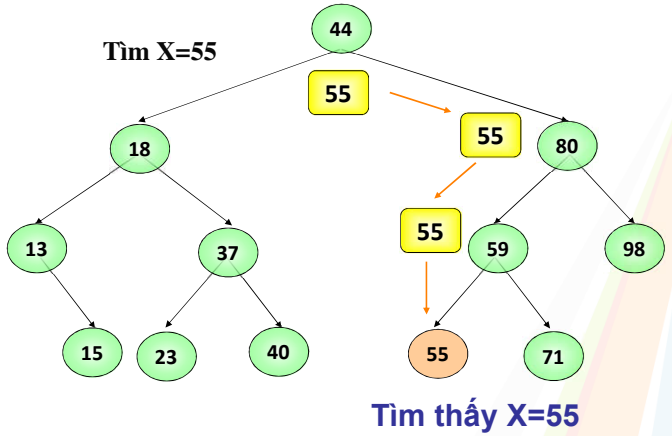
Tìm nút có khóa x, không dùng đệ quy

```
Node* Search(Tree T, int x) {
    Node *p = T;
    while(p!=NULL) {
       if(x==p->Key)
          return p; //Tim dc nut p co khoa x
       if(x<p->Key)
          p=p->Left; //Tim tiep o ben trai
       if(x>p->Key)
          p=p->Right; //Tim tiep o ben phai
    }
    return NULL; //Khong tim dc nut nao co khoa x
}
```


Tìm nút có khóa x, dùng đệ quy

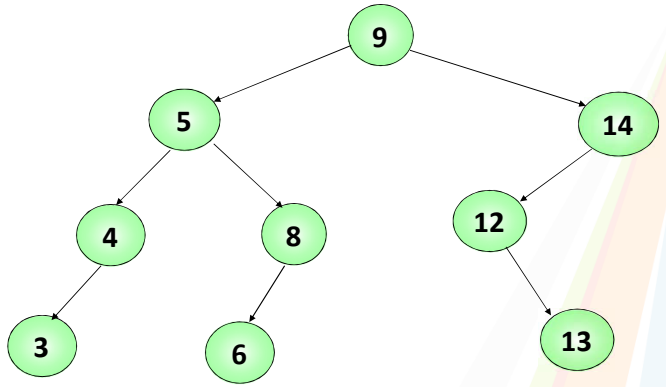
```
Node* Search(Tree T, int x) {
      if(T!=NULL) {
         if(x==T->Key)
            return T;
         if(x<T->Key)
            return Search(T->Left, x);
         if(x>T->Key)
            return Search(T->Right, x);
      }
      return NULL; //Khong tim dc nut nao co khoa x
}
```

Minh họa tìm nút



Minh họa tạo 1 cây NPTK

9, 5, 4, 8, 6, 3, 14,12,13



Hủy nút có khóa x trên cây

Có 3 trường hợp khi hủy 1 nút trên cây:

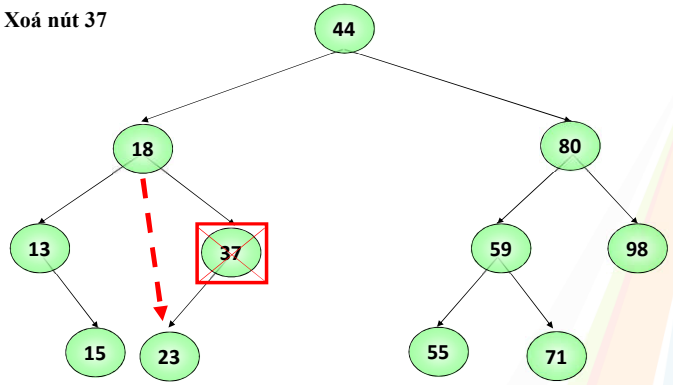
- X là nút lá – *Hủy bình thường mà không ảnh hưởng đến các nút khác trên cây.*
- X chỉ có 1 cây con - *Trước khi xoá x ta móc nối cha của X với con duy nhất cùa X*
- X có đầy đủ 2 cây con - *Ta dùng cách xoá gián tiếp*

Hủy nút có khóa x trên cây

- **Xóa gián tiếp:**
  - Thay vì hủy X ta tìm phần tử thế mạng Y. Nút Y có tối đa 1 cây con.
  - Thông tin lưu tại nút Y sẽ được chuyển lên lưu tại X.
  - Ta tiến hành xoá hủy nút Y (xoá Y giống 2 trường hợp đầu)
  - Cách tìm nút thế mạng Y cho X: Có 2 cách
    - C1: Nút Y là nút có **khoá nhỏ nhất** (trái nhất) bên **cây con phải** X
    - C2: Nút Y là nút có **khoá lớn nhất** (phải nhất) bên **cây con trái** của X

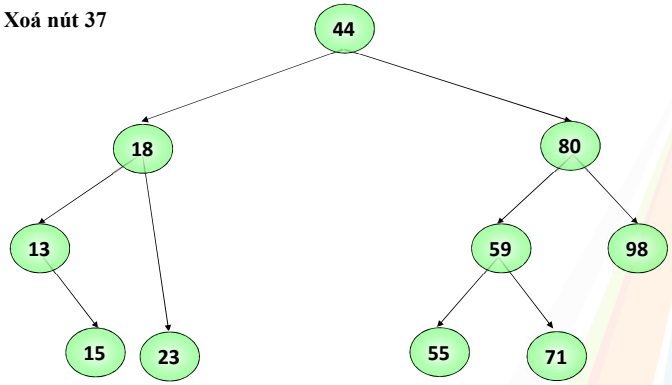
Minh họa hủy nút có 1 cây con

Xoá nút 37



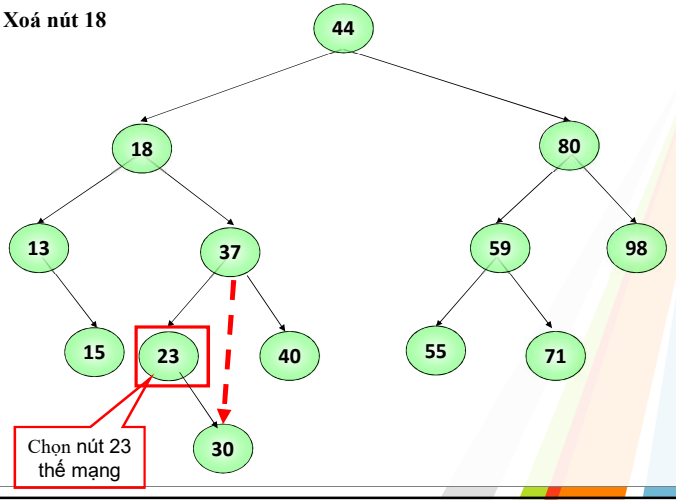
Minh họa hủy nút có 1 cây con

Xoá nút 37



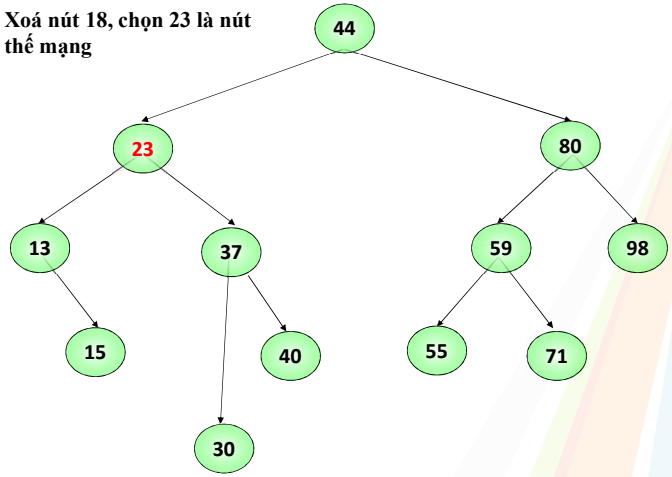
Minh họa hủy nút có 2 cây con

Xoá nút 18



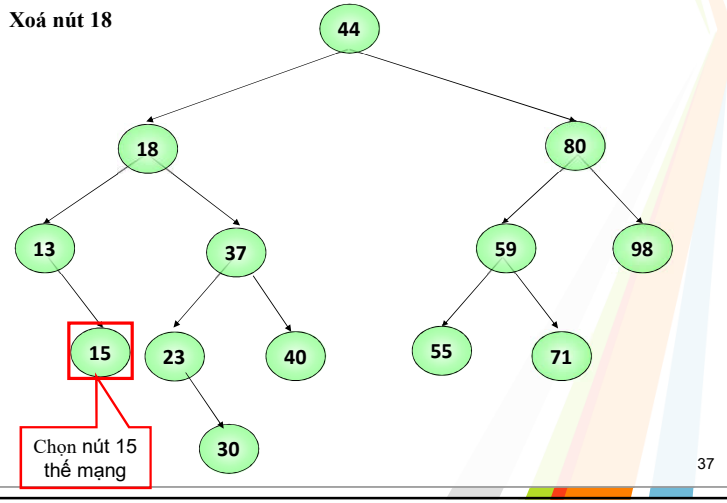
Minh họa hủy nút có 1 cây con

Xoá nút 18, chọn 23 là nút thế mạng



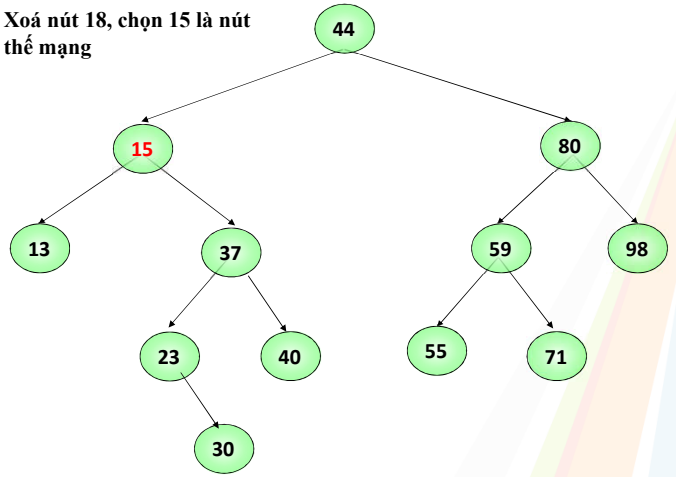
Minh họa hủy nút có 1 cây con

Xoá nút 18



Minh họa hủy nút có 1 cây con

Xoá nút 18, chọn 15 là nút thế mạng



Cài đặt thao tác xóa nút có Key x

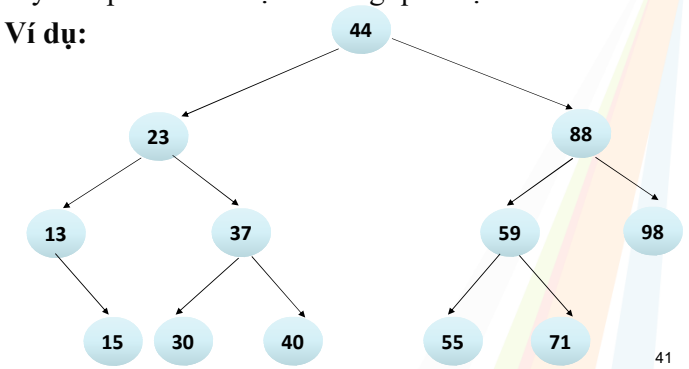
```
int DelNode(Tree &T, int x) {
   if (T==NULL) return 0;
   else if (T->Key>x) return DelNode(T->Left, x);
   else if (T->Key<x) return DelNode(T->Right, x);
   else {//T->Key == x
      Node *P=T;
      if (T->Left==NULL)
         T=T->Right;//Node chi co cay con phai
      else if (T->Right==NULL)
            T=T->Left; // Node chi co cay con trai
          else { // Node co ca 2 con
              Node *S = T,
              *Q=S->Left;
               //S la cha cua Q, Q la Node phai nhat cua
cay con trai cua P
```


Cài đặt thao tác xóa nút có Key x

```
              while (Q->Right!=NULL) {
                 S=Q;
                 Q=Q->Right;
              }
              P->Key=Q->Key;
              S->Right=Q->Left;
              delete Q;
      }
   }
   return 1;
}
```

4. Cây NPTK cân bằng (AVL)

- **Định nghĩa:** Cây nhị phân tìm kiếm cân bằng là cây mà tại mỗi nút của nó độ cao của cây con trái và của cây con phải chênh lệch không quá một
- **Ví dụ:**



CTDL của cây NPTK cân bằng

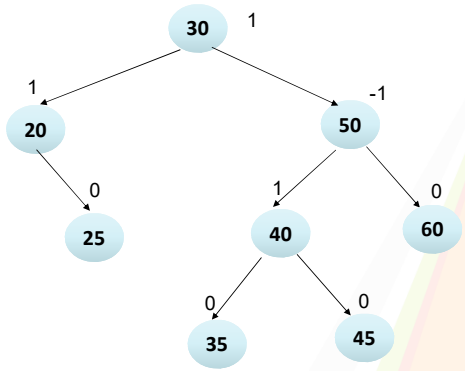
- Chỉ số cân bằng = độ lệch giữa cây trái và cây phải của một nút
- Các giá trị hợp lệ của CSCB:
  - CSCB(p) = 0 ⇔ Độ cao cây trái (p) = Độ cao cây phải (p)
  - CSCB(p) = 1 ⇔ Độ cao cây trái (p) < Độ cao cây phải (p)
  - CSCB(p) = -1⇔ Độ cao cây trái (p) > Độ cao cây phải (p)

CTDL của cây NPTK cân bằng

```
#define LH -1 //Cây con trái cao hơn
#define EH  0 //Cây con trái bằng cây con phải
#define RH  1 //Cây con phải cao hơn
struct Node {
      char  BalFactor; //chỉ số cân bằng
      int   Key;
      Node *Left;
      Node *Right;
};
typedef Node *Tree;
```


CTDL của cây NPTK cân bằng

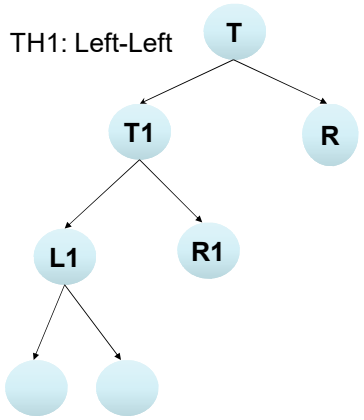
Hệ số cân bằng các nút trong cây AVL



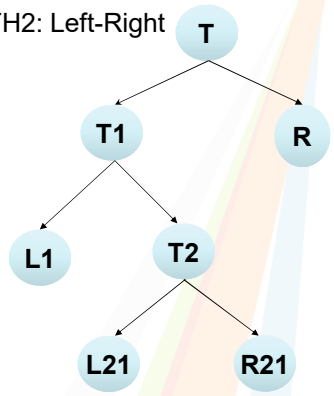
Các trường hợp mất cân bằng do lệch trái

- Cây mất cân bằng tại nút T

TH1: Left-Left



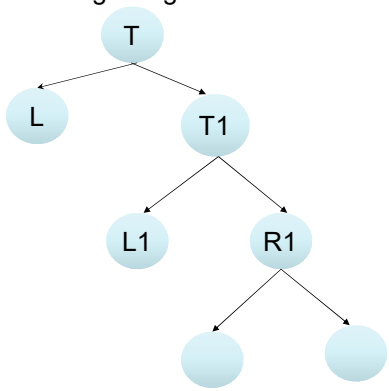
TH2: Left-Right



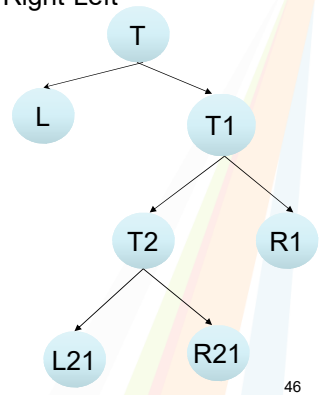
Các trường hợp mất cân bằng do lệch phải

- Cây mất cân bằng tại nút T

TH3: Right-Right



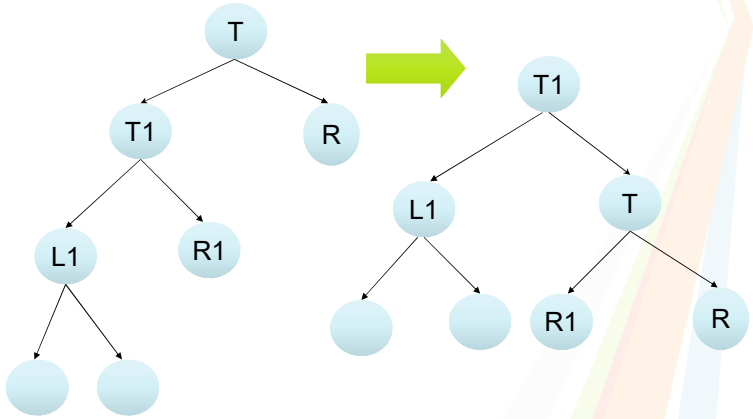
TH4: Right-Left



Các thao tác trên cây cân bằng

- Khi thêm hay xoá 1 nút trên cây, có thể làm cho cây mất tính cân bằng, khi ấy ta phải tiến hành cân bằng lại.
- Cây có khả năng mất cân bằng khi thay đổi chiều cao:
  - *Thêm bên trái -> lệch nhánh trái*
  - *Thêm bên phải -> lệch nhánh phải*
  - *Hủy bên phải -> lệch nhánh trái*
  - *Hủy bên trái -> lệch nhánh phải*
- Cân bằng lại cây : Tìm cách bố trí lại cây sao cho chiều cao 2 cây con cân đối:
  - *Kéo nhánh cao bù cho nhánh thấp*
  - *Phải bảo đảm cây vẫn là Nhị phân tìm kiếm*

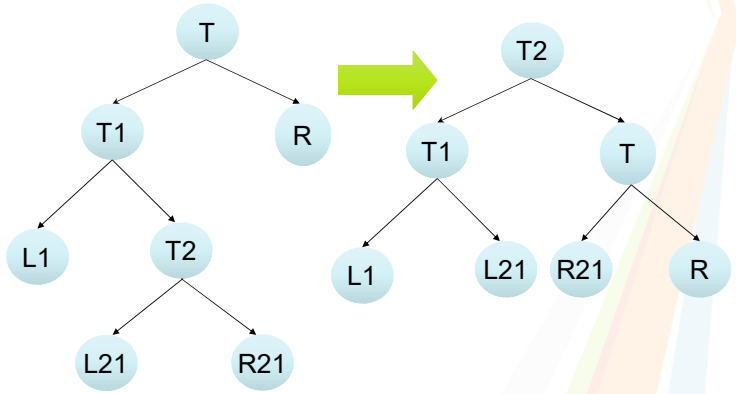
Cân bằng lại do lệch trái TH1 (LL)



Hàm cân bằng cho TH1 (LL)

```
void RotateLL (Tree &T) {
   Node *T1=T->Left;
   T->Left=T1->Right;
   T1->Right=T;
   switch(T1->BalFactor)   {
      case LH: T->BalFactor =EH;
               T1->BalFactor=EH; break;
      case EH: T->BalFactor =EH;
               T1->BalFactor=RH; break;
   }
   T=T1;
}
```

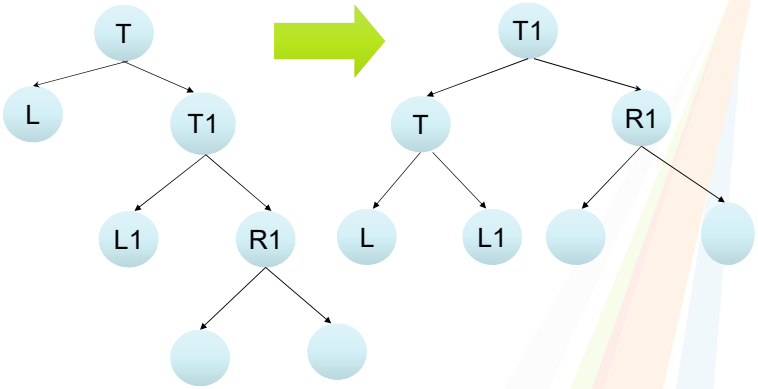

Cân bằng lại do lệch trái TH2 (LR)



Hàm cân bằng cho TH2 (LR)

```
void RotateLR (Tree &T) {
   Node *T1=T->Left, *T2=T1->Right;
   T->Left=T2->Right; T2->Right=T;
   T1->Right=T2->Left; T2->Left=T1;
   switch(T2->BalFactor) {
      case LH: T->BalFactor=EH;
               T1->BalFactor=RH; break;
      case EH: T->BalFactor=EH;
               T1->BalFactor=EH; break;
      case RH: T->BalFactor=LH;
               T1->BalFactor=EH; break;
   }
   T2->BalFactor=EH;
   T=T2;
}
```

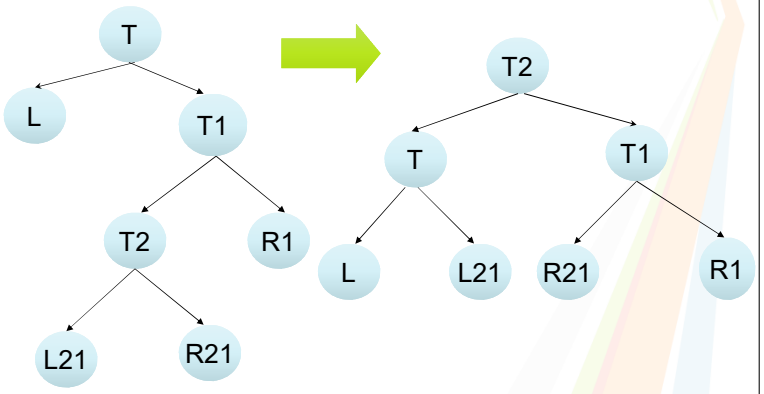

Cân bằng cho lệch phải TH3 (RR)



Hàm cân bằng TH3 (RR)

```
void RotateRR (Tree &T) {
   Node *T1=T->Right;
   T->Right=T1->Left;
   T1->Left=T;
   switch(T1->BalFactor) {
      case RH: T->BalFactor=EH;
               T->BalFactor=EH; break;
      case EH: T->BalFactor=EH;
               T1->BalFactor=LH; break;
   }
   T=T1;
}
```


Cân bằng cho lệch phải TH4 (RL)



Hàm cân bằng TH4 (RL)

```
void RotateRL (Tree &T) {
   Node *T1=T->Right, *T2=T1->Left;
   T->Right=T2->Left; T2->Left=T;
   T1->Left=T2->Right; T2->Right=T1;
   switch(T2->BalFactor)  {
      case RH: T->BalFactor=EH;
               T1->BalFactor=LH; break;
      case EH: T->BalFactor=EH;
               T1->BalFactor=EH; break;
      case LH: T->BalFactor=RH;
               T1->BalFactor=EH; break;
   }
   T2->BalFactor=EH; T=T2;
}
```


Hàm cân bằng khi cây lệch trái

Để thuận tiện, ta xây dựng 2 hàm cân bằng lại khi cây bị lệch trái hay lệch phải như sau:

```
 //Cân bằng khi cây bị lệch về bên trái
int BalanceLeft(Tree &T) {
    switch (T->Left->BalFactor) {
       case LH: RotateLL(T); return 2;
       case EH: RotateLL(T); return 1;
       case RH: RotateLR(T); return 2;
    }
    return 0;
}
```

Hàm cân bằng khi cây lệch phải

```
//Cân bằng khi cây bị lệch về bên phải
int BalanceRight(Tree &T) {
   switch (T-> Right -> BalFactor ) {
      case LH: RotateRL(T); return 2;
      case EH: RotateRR(T); return 1;
      case RH: RotateRR (T); return 2;
   }
   return 0;
}
```


Thêm nút vào cây NPTK CB

- Thêm bình thường như trường hợp cây NPTK
- Nếu cây tăng trưởng chiều cao
  - Lần ngược về gốc để phát hiện nút bị mất cân bằng
  - Tiến hành cân bằng lại nút đó bằng thao tác cân bằng thích hợp
- Việc cân bằng lại chỉ cần thực hiện 1 lần nơi mất cân bằng

Thêm nút vào cây NPTK CB

```
int InsNode(Tree &T, int x) {
    int res;
    if (T!=NULL) {
       if(T->Key==x) return 0;//đã có
       else if (T->Key>x) {
                res=InsNode(T->Left, x);
                if (res<2) return res;
                switch (T-> BalFactor) {
                   case RH:T->BalFactor=EH; return 1;
                   case EH:T->BalFactor=LH; return 2;
                   case LH: BalanceLeft(T); return 1;
                }
            }
```


Thêm nút vào cây NPTK CB

```
        else {//if (T->Key<x)
             res = InsNode(T->Right, x);
             if (res<2) return res;
             switch (T->BalFactor) {
                 case LH: T->BalFactor=EH; return 1;
                 case EH: T->BalFactor=RH; return 2;
                 case RH: BalanceRight(T); return 1;
             }
        }
    }
    T=new Node;
    if (T==NULL) return -1;//thiếu bộ nhớ
    T->Key=x;  T->BalFactor=EH;
    T->Left=T->Right=NULL;
    return 2;//thành công, chiều cao tăng
}
```

Hủy nút trên cây NPTK CB

- Hủy bình thường như trường hợp cây NPTK
- Nếu cây giảm chiều cao:
  - Lần ngược về gốc để phát hiện nút bị mất cân bằng
  - Tiến hành cân bằng lại nút đó bằng thao tác cân bằng thích hợp
  - Tiếp tục lần ngược lên nút cha…
- Việc cân bằng lại có thể lan truyền lên tận gốc

Hủy nút trên cây NPTK CB

```
int DelNode(Tree &T, int x) {
    int res ;
    if (T==NULL) return 0;
    if (T->Key>x) {
       res = DelNode (T-> Left , x);
       if (res<2) return res;
       switch (T->BalFactor) {
           case LH: T->BalFactor=EH; return 2;
           case EH: T->BalFactor=RH; return 1;
           case BalanceRight(T);
       }
    } else if (T->Key<x) {
              res = DelNode (T->Right, x);
```


Hủy nút vào cây NPTK CB

```
            if (res<2) return res;
            switch (T->BalFactor ) {
                case RH: T->BalFactor=EH; return 2;
                case EH: T->BalFactor=LH; return 1;
                case LH: return BalanceLeft(T);
            }
        } else { // if (T->Key == X)
               Node *p=T;
               if (T->Left==NULL) {
                  T=T->Right; res=2; }
               else if (T->Right==NULL) {
                       T=T->Left; res=2; }
                    else
              //if (T->Left!=NULL && T-> Right!=NULL)
```


Hủy nút vào cây NPTK CB

```
            {
             res=SearchStandFor(p, T->Right);
             if(res<2) return res;
             switch(T->BalFactor) {
                 case RH: T->BalFactor=EH; return 2;
                 case EH: T->BalFactor=LH; return 1;
                 case LH: return BalanceLeft(T);
             }
        }
        delete p;
        return res;
    }
}
```

Hủy nút vào cây NPTK CB

```
//Tìm phần tử thế mạng
int SearchStandFor(Tree &p, Tree &q) {
      int res;
      if (q->Left!=NULL) {
           res=SearchStandFor(p, q->Left);
           if (res<2) return res;
           switch (q->BalFactor) {
              case LH: q->BalFactor=EH; return 2;
              case EH: q->BalFactor=RH; return 1;
              case RH: return BalanceRight(T); }
      } else {
            p->Key=q->Key; p=q; q=q->Right; return 2;
        }
}
```


CHƯƠNG VI

BẢNG BĂM (Hash Tables)

Nội dung chính

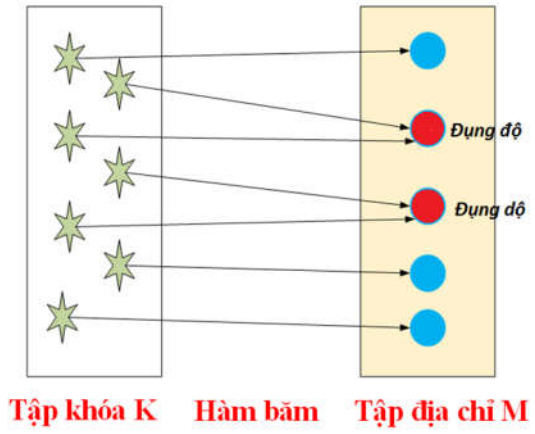
5.1 Tổng quan về bảng băm

- Tư tưởng của phép băm là dựa vào giá trị các khóa k[1..n], chia các khóa đó thành các nhóm.
- Những khóa cùng một nhóm sẽ có cùng một đặc điểm chung và đặc điểm này không có trong nhóm khác.
- Khi có một khóa tìm kiếm X, ta xác định xem nếu X có trong dãy khóa thì nó thuộc nhóm nào và ta tiến hành tìm trên nhóm đó.

Một số khái niệm

- **Bảng băm** là 1 CTDL tương tự như mảng nhưng kèm theo một hàm băm để ánh xạ nhiều giá trị vào cùng một phần tử trong mảng
- **Hàm băm** hay phép băm (hash function) là hàm được dùng để ánh xạ (hoặc biến đổi) giá trị của khóa vào một dãy các địa chỉ của bảng băm.
- **Khóa** là những giá trị không trùng nhau, có thể là dạng số hay số dạng chuỗi.
- Giả sử có 2 khóa phân biệt ki và kj nếu h(ki) = h(kj) thì ta nói **hàm băm bị đụng độ** (xung đột) *(với h(x) là hàm băm)*.

Minh họa



Ưu điểm của bảng băm

- Bảng băm là một cấu trúc dung hòa giữa thời gian truy xuất và dung lượng bộ nhớ:
  - *Nếu không có sự giới hạn về bộ nhớ thì chúng ta có thể xây dựng bảng băm với mỗi khóa ứng với một địa chỉ với mong muốn thời gian truy xuất tức thời.*
  - *Nếu dung lượng bộ nhớ có giới hạn thì tổ chức một số khóa có cùng địa chỉ, khi đó tốc độ truy xuất sẽ giảm.*
- Bảng băm được ứng dụng nhiều trong thực tế, rất thích hợp khi tổ chức dữ liệu có kích thước lớn và được lưu trữ ở bộ nhớ ngoài.

Các phép toán trên bảng băm

- Khởi tạo (Initialize)
- Kiểm tra rỗng (Empty)
- Lấy kích thước của bảng băm (Size)
- **Tìm kiếm (Search)**
- **Thêm mới phần tử (Insert)**
- **Loại bỏ (Remove)**
- Sao chép (Copy)
- Duyệt (Traverse)

5.2 Phương pháp xây dựng hàm băm

- **Một hàm băm tốt phải thỏa mãn các điều kiện sau:**
  - *Tính toán nhanh.*
  - *Các khoá được phân bố đều trong bảng.*
  - *Ít xảy ra đụng độ.*
- **Một số phương pháp xây dựng hàm băm:**
  - *Hàm băm dạng bảng tra*
  - *Phương pháp chia.*
  - *Phương pháp nhân.*

Hàm băm dạng bảng tra

- Hàm băm có thể tổ chức ở dạng bảng tra hoặc thông dụng nhất là ở dạng công thức.
- Ví dụ: Bảng tra với khóa là bộ chữ cái, bảng băm có 26 địa chỉ từ 0 đến 25. Khóa a ứng với địa chỉ 0, khoá b ứng với địa chỉ 1,… , z ứng với địa chỉ 25.

Khoá	Địa chỉ	Khóa	Địa chỉ	Khóa	Địa chỉ	Khóa	Địa chỉ
a	0	h	7	o	14	v	21
b	1	I	8	p	15	w	22
c	2	j	9	q	16	x	23
d	3	k	10	r	17	y	24
e	4	l	11	s	18	z	25
f	5	m	12	t	19	/	/
g	6	n	13	u	20	/	/

Phương pháp chia

$$h(k) = k \% m$$

- **m** (số nguyên ≤ M kích thước của bảng). Thông thường nên chọn m là nguyên tố.
- **k** là khóa.
- Giá trị của **h(k)** sẽ là: ***0, 1, 2, 3,…, m-1***.
- Ví dụ: Tập khoá là các giá trị số gồm 3 chữ số, và vùng nhớ cho bảng địa chỉ có khoảng 100 mục, như vậy ta sẽ lấy hai số cuối của khoá để làm địa chỉ theo phép chia lấy dư cho 100 : chẳng hạn 325 % 100 = 25

Phương pháp chia

- Tuy nhiên ta nhận thấy nếu hàm băm dùng công thức như trên thì địa chỉ của khoá tính được chỉ căn cứ và 2 ký số cuối. Vì thế, để hàm băm có thể tính địa chỉ khoá một cách "ngẫu nhiên" hơn ta nên chọn m=97 thay vì 100

m=100		m=97 (nguyên tố)	
Khoá	Địa chỉ	Khoá	Địa chỉ
325	25	325	34
125	25	125	28
147	47	147	50

Phương pháp nhân

h(k) = | m*(k*A % 1) |

- **k** là khóa, **m** ≤ M là kích thước của bảng – *thường chọn* $m = 2^n$, A là hằng số và 0<A<1.
- Theo Knuth chọn $A = (\sqrt{5}-1)/2 = 0.6180339887$ *là tốt.*

...

m=100, A=0.61803	
Khoá	Địa chỉ
325	86
125	25
147	85

5.3 Phương pháp giải quyết đụng độ

- ***Phương pháp kết nối***
- ***Phương pháp dò tuần tự***
- ***Phương pháp dò bậc hai***
- ***Phương pháp băm kép***

Phương pháp kết nối (Chaining Method)

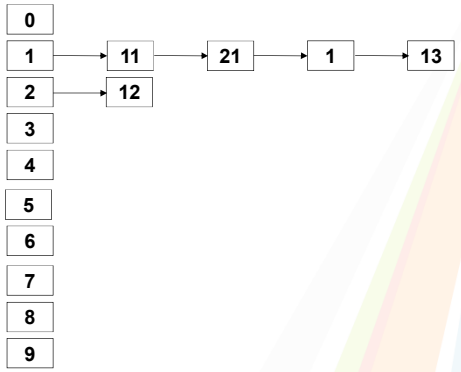
Phương pháp kết nối trực tiếp

- Bảng băm được cài đặt bằng các danh sách liên kết, các phần tử trên bảng băm được "băm" thành M danh sách liên kết (từ danh sách 0 đến danh sách M–1).
- Các phần tử bị xung đột tại địa chỉ i được kết nối trực tiếp với nhau qua danh sách liên kết i.

Ví dụ minh họa pp kết nối trực tiếp

- Tập khóa K = {11, 12, 21, 1, 13}, Tập địa chỉ M = {0, 1, …, 9}, Hàm băm h(key) = key % 10.

k	h(k)
11	1
12	2
21	1
1	1
13	1



Phương pháp kết nối

Phương pháp kết nối hợp nhất

- Bảng băm trong trường hợp này được cài đặt bằng danh sách liên kết dùng mảng, có M phần tử.
- Các phần tử bị xung đột tại một địa chỉ được kết nối với nhau qua một danh sách liên kết.
- Mỗi phần tử của bảng băm gồm hai trường:
  - Trường key: chứa khóa của mỗi phần tử
  - Trường next: con trỏ chỉ đến phần tử kế tiếp nếu có xung đột.
- Khởi động: Khi khởi động, tất cả trường key của các phần tử trong bảng băm được gán bởi giá trị NullKey, còn tất cả các trường next được gán –1.

Phương pháp kết nối

- Thêm mới một phần tử: Khi thêm mới một phần tử có khóa key vào bảng băm, hàm băm h(key) sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1.
  - Nếu chưa bị xung đột thì thêm phần tử mới vào địa chỉ này.
  - Nếu bị xung đột thì phần tử mới được cấp phát là phần tử trống phía cuối mảng. Cập nhật liên kết next sao cho các phần tử bị xung đột hình thành một danh sách liên kết

Ví dụ minh họa pp kết nối hợp nhất

- Ví dụ:
  - Tập khóa K = {11, 12, 21, 1, 13}
  - Tập địa chỉ M = {0, 1, …, 9}
  - Hàm băm h(key) = key % 10.

k	h(k)
11	1
12	2
21	1
1	1
13	1

	key	Next
0	Null	-1
1	Null	-1
2	Null	-1
…	Null	-1
7	Null	-1
8	Null	-1
9	Null	-1

	key	Next
0	Null	-1
1	11	-1
2	12	-1
…	Null	-1
7	Null	-1
8	Null	-1
9	Null	-1

	key	Next
0	Null	-1
1	11	9
2	12	-1
…	Null	-1
7	Null	-1
8	Null	-1
9	21	-1

	key	Next
0	Null	-1
1	11	9
2	12	-1
…	Null	-1
7	13	-1
8	1	7
9	21	8

Phương pháp dò tuyến tính (Linear Probe)

- Bảng băm trong trường hợp này được cài đặt bằng danh sách kề có M phần tử (kích thước bảng băm), mỗi phần tử của bảng băm là một mẫu tin có một trường key để chứa khoá của phần tử (ban đầu khởi tạo bằng -1);
- Khi thêm phần tử có khoá key vào bảng băm, hàm băm h(key) sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1:
  - *Nếu chưa bị xung đột thì thêm phần tử mới vào địa chỉ này.*
  - *Nếu bị xung đột thì hàm băm lại lần 1, hàm h1 sẽ xét địa chỉ kế tiếp, nếu lại bị xung đột thì hàm băm lại lần 2, hàm h2 sẽ xét địa chỉ kế tiếp nữa, …, và quá trình cứ thế cho đến khi nào tìm được địa chỉ trống và thêm phần tử mới vào địa chỉ này.*

Phương pháp dò tuyến tính

- Khi tìm một phần tử có khoá key trong bảng băm, hàm băm h(key) sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1, tìm phần tử khoá key trong bảng băm xuất phát từ địa chỉ i.
- Hàm băm lại lần i được biểu diễn bằng công thức sau:

$$h_i(key) = (h(key)+i) \% M$$

với h(key) là hàm băm chính của bảng băm.

- *Lưu ý:*
  - *Địa chỉ dò tìm kế tiếp là địa chỉ 0 nếu đã dò đến cuối bảng.*
  - *Hàm băm lần thứ i ta sẽ % cho M (kích thước bảng băm)*

Ví dụ

K={76,93,40,47,10,55}
M = {0,1, …, 6}
h(k)=k%7

k	h(k)
76	76%7=6
93	93%7=2
40	40%7=5
47	47%7=5
10	10%7=3
55	55%7=6

h = 47%7=5
h1=(5+1)%7=6
h2=(5+2)%7=0

h=55%7=6
h1=(6+1)%7=0
h2=(6+2)%7=1

76 ⇨ 6

0	1	2	3	4	5	6
						76

93 ⇨ 2

0	1	2	3	4	5	6
		93				76

40 ⇨ 5

0	1	2	3	4	5	6
		93			40	76

47 ⇨ 5
47 ⇨ 6
47 ⇨ 0

0	1	2	3	4	5	6
47		93			40	76

10 ⇨ 3

0	1	2	3	4	5	6
47		93	10		40	76

55 ⇨ 6
55 ⇨ 0
55 ⇨ 1

0	1	2	3	4	5	6
47	55	93	10		40	76

P/P dò bậc hai (Quadratic Probing Method)

- Tương tự dò tuần tự nhưng hàm băm lại lần thứ i được biểu diễn bằng công thức sau:

$$h_i(key) = (h(key)+ i^2) \% M$$

với h(key) là hàm băm chính của bảng băm.

- Nếu đã dò đến cuối bảng thì trở về dò lại từ đầu bảng.

Ví dụ

K={76,40,48,5,55}
M = {0,1, …, 6}
Hàm băm h(k)=k%7

k	h(k)
76	76%7=6
40	40%7=5
48	48%7=6
5	5%7=5
55	55%7=6

h=48%7=6
$h_1=(6+1^2)\%7=0$

h=5%7=5
$h_1=(5+12)\%7=6$
$h_2=(5+22)\%7=2$

h=55%7=6
$h_1=(6+1^2)\%7=0$
$h2=(6+2^2)\%7=3$

76 ⇨ 6

0	1	2	3	4	5	6
						76

40 ⇨ 5

0	1	2	3	4	5	6
					40	76

48 ⇨ 6
48 ⇨ 0

0	1	2	3	4	5	6
48					40	76

5 ⇨ 5
5 ⇨ 6
5 ⇨ 2

0	1	2	3	4	5	6
48	5				40	76

55 ⇨ 6
55 ⇨ 0
55 ⇨ 3

0	1	2	3	4	5	6
48			55		40	76

P/P băm kép (Double hashing Method)

- Phương pháp băm kép dùng thêm 1 hàm băm thứ hai, thông thường người ta chọn hàm băm thứ 2 như sau:

  Hàm băm 1: ***h(key) = key % m***

  Hàm băm 2: ***d(key) = p - key% p***

  (Trong đó ***m, p <= M)***

- Khi thêm phần tử có khoá key vào bảng băm:
  - *Nếu chưa bị xung đột thì thêm phần tử mới tại địa chỉ a này.*
  - *Nếu bị xung đột thì xét địa chỉ mới (h+d)%M.*
  - *Nếu bị xung đột lần 2 thì mới (h+2*d)%M.*
  - ...
  - *Nếu bị xung đột lần i thì xét địa chỉ:* ***(h+i*d)%M***

Ví dụ

K={76,93,40,47,10,55}
M = {0,1, …, 6}
h(k)=k%7
d(k)=5-(k%5)

k	h(k)	d(k)
76	6	4
93	2	2
40	5	5
47	5	3
10	3	5
55	6	5

76 ⇨ 6

0	1	2	3	4	5	6
						76

93 ⇨ 2

0	1	2	3	4	5	6
		93				76

40 ⇨ 5

0	1	2	3	4	5	6
		93			40	76

h=5, d=3
(h+d)%7=1
47 ⇨ 5
47 ⇨ 1

0	1	2	3	4	5	6
	47	93			40	76

10 ⇨ 3

0	1	2	3	4	5	6
	47	93	10		40	76

h=6, d=5
(h+d)%7=4
55 ⇨ 5
55 ⇨ 4

0	1	2	3	4	5	6
	47	93	10	55	40	76

Ví dụ

K={18,41,22,
44,59,32,31,73}
M={0, 1,..,12}
h(k)=k%13
d(k)=7-(k%7)

k	h(k)	d(k)
18	5	3
41	2	1
22	9	6
44	5	5
59	7	4
32	6	3
31	5	4
73	8	4

	0	1	2	3	4	5	6	7	8	9	10	11	12
18⇨5						18							
41⇨2			41			18							
22⇨9			41			18				22			
h=5,d=5 (5+5)%13=10													
44⇨5 44⇨10			41			18				22	44		
59⇨7			41			18		59		22	44		
32⇨6			41			18	32	59		22	44		
31⇨5 31⇨9 31⇨0	31		41			18	32	59		22	44		
h=5,d=4 (5+4)%13=9 (5+8)%13=0													
73⇨8	31		41			18	32	59	73	22	44		

Ví dụ

K={14,17,25,37,34,16,26}
M = {0, 1, …, 10}
h(k)=k%11
d(k)=k%7+1

k	h(k)	d(k)
14	3	1
17	6	4
25	3	5
37	4	3
34	1	7
16	5	3
26	4	6

14 ⇨ 3

0	1	2	3	4	5	6	7	8	9	10
			14							

17 ⇨ 6

0	1	2	3	4	5	6	7	8	9	10
			14			17				

h=3, d=5
(3+5)%11=8
25 ⇨ 8

0	1	2	3	4	5	6	7	8	9	10
			14			17		25		

37 ⇨ 4

0	1	2	3	4	5	6	7	8	9	10
			14	37		17		25		

34 ⇨ 1

0	1	2	3	4	5	6	7	8	9	10
	34		14	37		17		25		

16 ⇨ 5

0	1	2	3	4	5	6	7	8	9	10
	34		14	37	16	17		25		

h=4, d=6
(4+6)%11=10
26 ⇨ 4
26 ⇨ 10

0	1	2	3	4	5	6	7	8	9	10
	34		14	37	16	17		25		26