BÀI 6

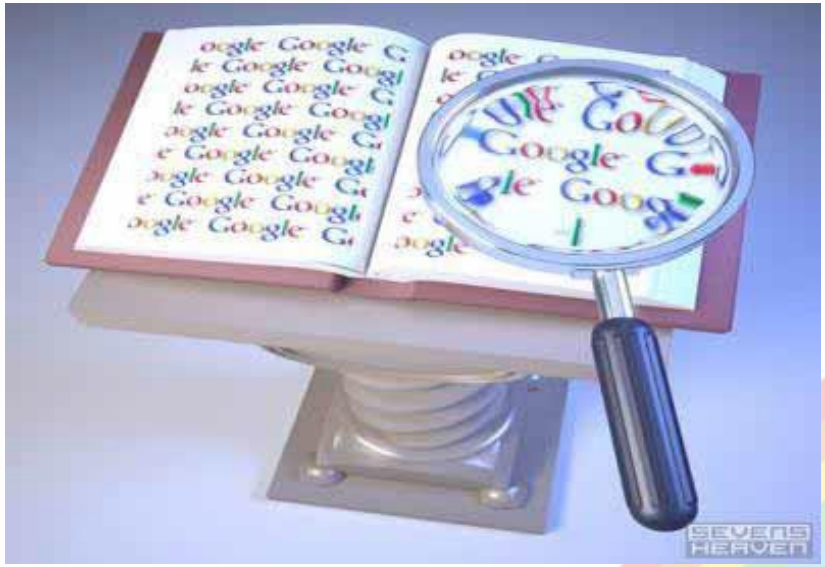
TÌM KIẾM

MỤC TIÊU

- Mô tả đúng nội dung bài toán tìm kiếm và hướng giải quyết bài toán tìm kiếm;
- Trình bày và thực hiện cài đặt các thuật toán tìm kiếm như tìm kiếm tuần tự, tìm kiếm nhị phân... một cách chính xác thông qua ngôn ngữ lập trình C;
- Mô tả đúng về cây nhị phân tìm kiếm. Trình bày và cài đặt các thao tác trên cây nhị phân tìm kiếm một cách chính xác;
- Sử dụng các giải thuật tìm kiếm thích hợp để giải quyết các bài toán trong thực tế.

NỘI DUNG

- Bài toán tìm kiếm;
- Tìm kiếm tuần tự;
- Tìm kiếm nhị phân;
- Cây nhị phân tìm kiếm.

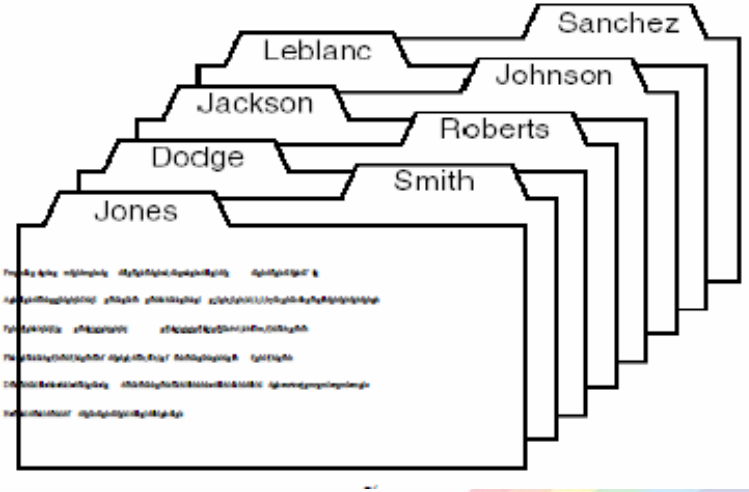


1. BÀI TOÁN TÌM KIẾM

1. BÀI TOÁN TÌM KIẾM

Ví dụ 6.1:

Một Ngân hàng nắm giữ tất cả thông tin của rất nhiều tài khoản khách hàng và cần tìm kiếm để kiểm tra các biến động. Một hãng Bảo hiểm hay một hệ thống trợ giúp bán vé xe, vé máy bay… Việc tìm kiếm thông tin để đáp ứng việc sắp đặt ghế và các yêu cầu tương tự như vậy là thực sự cần thiết.



1. BÀI TOÁN TÌM KIẾM

Một bài toán tìm kiếm tổng quát có thể được phát biểu như sau:

- "Cho một bảng gồm n mẩu tin R_1, R_2,…, R_n. Với mỗi mẩu tin R_i được tương ứng với một khóa k_i. Hãy tìm mẩu tin có giá trị khóa bằng X cho trước."
- Công việc tìm kiếm sẽ hoàn thành nếu có một trong hai tình huống sau xảy ra:
  - Tìm được mẩu tin có khóa tương ứng bằng X, lúc đó phép tìm kiếm thành công.
  - Không tìm được mẩu tin nào có khóa tìm kiếm bằng X, phép tìm kiếm thất bại.

1. BÀI TOÁN TÌM KIẾM

Giải thuật tìm kiếm là một thuật toán lấy đầu vào là một bài toán và trả về kết quả là một lời giải cho bài toán đó.

Các thao tác đó gồm:

- Khởi tạo cấu trúc dữ liệu (INITIALIZE);
- Tìm kiếm một hay nhiều mẩu tin có khoá đã cho (SEARCH);
- Chèn thêm một mẩu tin mới (INSERT);
- Nối lại từ điển để tạo thành một từ điển lớn hơn (JOIN);
- Sắp xếp từ điển; xuất ra tất các mẩu tin theo thứ tự được sắp xếp (SORT).

2. TÌM KIẾM TUẦN TỰ

2. TÌM KIẾM TUẦN TỰ

Tư tưởng của thuật toán (Sequential Search):

- Bắt đầu từ mẩu tin đầu tiên, lần lượt so sánh khóa tìm kiếm với khóa tương ứng của các mẩu tin trong danh sách, cho tới khi tìm thấy mẩu tin có khóa bằng khóa tìm kiếm hoặc đã duyệt hết danh sách mà chưa thấy;
- Thuật toán tìm kiếm tuần tự dễ thực hiện nhất đối với thông tin lưu trữ dạng mảng.

Nội dung và cách cài đặt của thuật toán:

- Giả sử các mẩu tin cần tìm kiếm được lưu trữ trong một danh sách R với n mẩu tin và X là khóa tìm kiếm;
- Thuật toán được mô tả như sau:

Bước 1: Gán i=0;//duyệt từ đầu mảng

Bước 2: while((R[i]!=X)&&(i<n))

{ i=i+1; }

Bước 3: if(i<n) tìm thấy mẩu tin có khóa bằng X tại nút i-1;

Bước 4: else không tìm thấy mẩu tin có khóa bằng X;

Bước 5: kết thúc.

2. TÌM KIẾM TUẦN TỰ

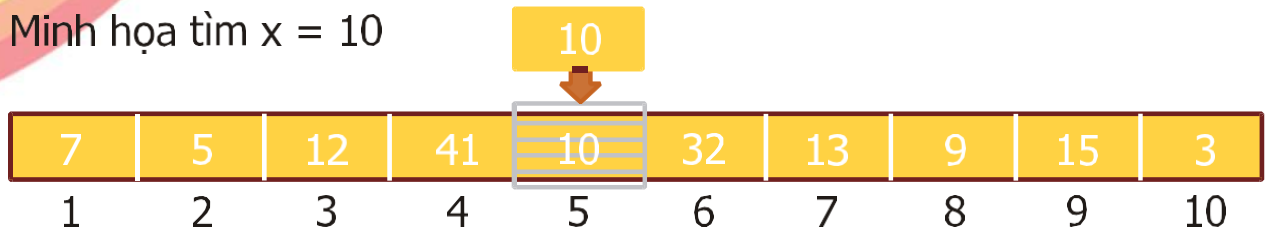
Cài đặt thuật toán:

```
typedef <kiểu_dữ_liệu> KeyType;
int Sequential_Search(dataArray R,KeyType X,int n);
    {
        int i;  i =
        0;
        while((R[i]!= X)&&(i < n))
        {
            i++;
        }
            if(i < n) return (1);
            else return(–1);
    }
```

2. TÌM KIẾM TUẦN TỰ

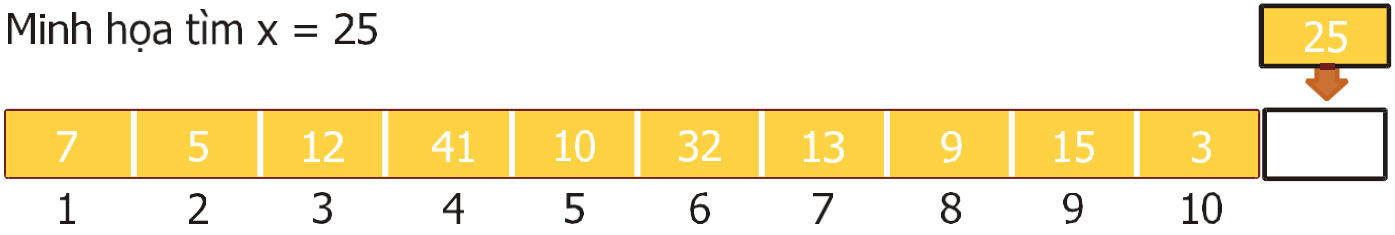
Ví dụ 6.2: Thuật toán tìm kiếm tuần tự được minh họa trong hình dưới đây:

Minh họa tìm x = 10



Đã tìm thấy tại vị trí 5

Minh họa tìm x = 25



Đã hết mảng

2. TÌM KIẾM TUẦN TỰ

Đánh giá thuật toán:

- Trong trường hợp tìm kiếm tốt nhất, ngay phần tử đầu tiên của mảng có khóa bằng X khi đó:
  - Số phép gán: Gmin = 1
  - Số phép so sánh: Smin = 2 + 1 = 3
- Trong trường hợp tồi nhất, khi đó ta phải duyệt hết mảng thì:
  - Số phép gán: Gmax = n
  - Số phép so sánh: Smax = 2n + 1
- Như vậy số phép toán trung bình được thực hiện trong tìm kiếm tuần tự đó là:
  - Số phép gán: Gavg = n / 2
  - Số phép so sánh: Savg = (3 + 2n + 1) / 2 = n + 2

2. TÌM KIẾM TUẦN TỰ

Chúng ta có thể giảm bớt đi một phép so sánh bằng cách ta thêm vào cuối mảng một phần tử "cầm canh" có giá trị bằng X để nhận diện ra sự duyệt hết mảng.

```
typedef <kiểu_dữ_liệu> KeyType;
int Sequential_Search(dataArray R,KeyType X,int n);
        {               int i;
                        i=0;        R[n]=X;
                        while(R[i]!=X)
                        {     i++;              }
                        if(i<n) return (1);
                        else return(-1);
        }
```

2. TÌM KIẾM TUẦN TỰ

- Với việc sử dụng phần tử cầm canh ở cuối mảng tìm kiếm, thì số phép toán thực hiện trung bình trong thuật toán này là:
  - Số phép gán: Gavg = n/2
  - Số phép so sánh: Savg = (n + 1) / 2
- Việc tìm kiếm sẽ đơn giản hơn nếu mảng đang tìm được sắp xếp thứ tự.

Ví dụ 6.3:

Khi tìm giá trị X = 35 trong mảng được sắp xếp theo thứ tự tăng dần như sau: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100.

Ta nhận thấy không cần thiết phải tìm vượt quá phần tử có giá trị 40 vì phần tử này và các phần tử sau nó đều có giá trị lớn hơn 35, và việc tìm kiếm có thể kết thúc tại đây.

2. TÌM KIẾM TUẦN TỰ

Từ đây ta có thuật toán tìm kiếm tuần tự cho các danh sách (ở trường hợp này là danh sách được lưu trữ dưới dạng mảng) có thứ tự như sau:

```
typedef <kiểu_dữ_liệu> KeyType;
int LinearSearch(KeyType X, dataArray R,int n)
{    int i;  for(i=0;i<n;i++)
{       if(R[i]==X) eturn(i);
        else if(X<R[i]) return(-1);
}
}
```

- Tuy nhiên trong trường hợp tồi nhất thì số lần thực hiện phép so sánh cùng là $O(n) \approx n$;
- Như vậy tìm kiếm tuần tự là đơn giản và thuận tiện khi danh sách tìm kiếm không lớn;
- Tuy nhiên trong trường hợp số phần tử của danh sách tìm kiếm là rất lớn mà theo tìm kiếm tuần tự thì sẽ mất rất nhiều thời gian.

3. TÌM KIẾM NHỊ PHÂN

3. TÌM KIẾM NHỊ PHÂN

Tư tưởng của thuật toán: Tìm kiếm dựa trên ứng dụng sơ đồ "chia – để – trị".

- Chia những mẩu tin trong danh sách tìm kiếm thành hai phần, xác định xem phần nào chứa khoá cần tìm;
- Việc chia đôi và tìm kiếm trên một nửa các mẩu tin được thực hiện được trong danh sách có thứ tự;
- Để tìm khóa X có trong một dãy hay không, trước tiên ta so sánh X với khóa của mẩu tin ở giữa danh sách, nếu X nhỏ hơn thì X chỉ có thể nằm trong nửa đầu tiên của danh sách. Ngược lại nếu X lớn hơn thì X nằm trong nửa còn lại. Và ta cứ lặp lại quá trình này cho đến khi tìm thấy X hoặc phạm vi tìm kiếm không còn nữa.

3. TÌM KIẾM NHỊ PHÂN

Nội dung và cách cài đặt thuật toán:

- Bước 1: đặt First=0 và Last=n-1;
- Bước 2: Found= -1;//Found là biến lưu vị trí tìm thấy X trong mảng
- Bước 3: while((First<=Last)&&(Found==-1))

```
{
    Mid=(First+Last)/2;
    if(X<R[i]) Last=Mid-1;
    else if(X>R[Mid])First=R[Mid]+1;
    else Found=Mid;
}
```

3. TÌM KIẾM NHỊ PHÂN

Cài đặt thuật toán:

```
int BinarySearch(ArrayType R,KeyType X,int n);
{
  int Mid,First,Last;  int
  Found;
        First = 0;
        Last = n – 1;
        Found = –1;
        while((First <= Last)&&(Found = –1))
  {
          Mid = (First + Last)/2;
          if(X < R[Mid])
                    Last = Mid – 1;
              else if (X > R[Mid]) First = Mid + 1;
              else Found = Mid;
        }
        return Found;
};
```

Đánh giá thuật toán: Trong trường hợp tồi nhất độ phức tạp của thuật toán này là O(log2n).

3. TÌM KIẾM NHỊ PHÂN

Ví dụ 6.4:

Giả sử ta có một mảng R gồm 10 phần tử như sau:

10 20 30 40 50 60 70 80 90 100

Xét trường hợp tìm phần tử có giá trị X = 40 (trường hợp tìm thấy) ta có:

Lần lặp	First	Last	First <= Last	Mid	R[Mid]	X = R[Mid']	X < R[Mid]	X > R[Mid]
Ban đầu	1	10	True	5	50	False	False	True
1	1	4	True	2	20	False	False	True
3	4	4	True	4	50	True		
4	5	4	False					

3. TÌM KIẾM NHỊ PHÂN

Cài đặt thuật toán qua xây dựng hàm tìm kiếm BinarySearch:

```
int BinarySearch(KeyType, ArrayType R,int First, int  Last)
{ int  Mid;
    if(First>Last) return (-1);  else
    {       Mid=(First+Last)/2;
            if(X<R[Mid]) BinarySearch(X,R,First,Mid-1);  else
            if(X>R[Mid]) BinarySearch(X,R,Mid+1,Last);  else
              return(Mid);
    }
}
```

Độ phức tạp của thuật toán tìm kiếm nhị phân đệ quy là O(log2n).

3. TÌM KIẾM NHỊ PHÂN

Chú ý:

- Thuật toán tìm kiếm nhị phân chỉ có thể vận dụng trong trường hợp dãy (mảng) đã có thứ tự. Trong trường hợp tổng quát chúng ta chỉ có thể áp dụng thuật toán tìm kiếm tuần tự;
- Các thuật toán đệ quy có thể ngắn gọn song tốn kém bộ nhớ để ghi nhận mã lệnh chương trình (mỗi lần gọi đệ quy) khi chạy chương trình, do vậy có thể làm chương trình chạy chậm lại. Vì vậy, khi viết chương trình nếu có thể chúng ta nên sử dụng thuật toán không đệ quy.

4. CÂY NHỊ PHÂN TÌM KIẾM

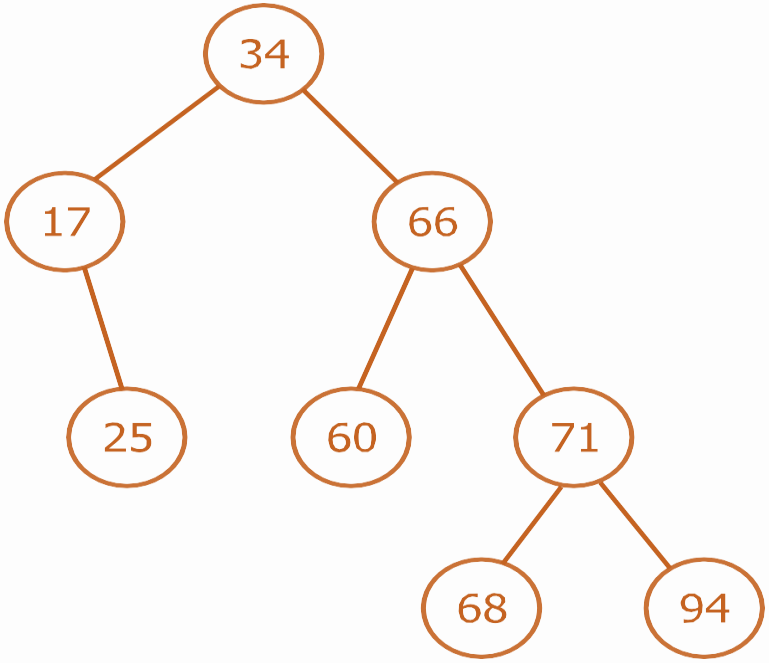
4. CÂY NHỊ PHÂN TÌM KIẾM

Khái niệm về cây nhị phân tìm kiếm:

Cây nhị phân tìm kiếm là cây nhị phân mà mỗi nút đều được gán một khóa sao cho với mỗi mỗi nút k:

- Mọi khóa trên cây con trái đều nhỏ hơn khóa trên nút k;
- Mọi khóa trên cây con phải đều lớn hơn khóa trên nút k.

Ví dụ 6.5: Hình dưới đây mô tả một cây nhị phân tìm kiếm.



4. CÂY NHỊ PHÂN TÌM KIẾM

Cấu trúc dữ liệu biểu diễn cây nhị phân tìm kiếm được cài đặt như sau:

```
typedef <kiểu_dữ_liệu> KeyType
typedef struct Node
{
                    KeyType info;
                    Node *left;
                    Node *right;
}Node;
//Dinh nghia cay nhi phan  typedef
Node * Tree;
Tree BST;
```

4. CÂY NHỊ PHÂN TÌM KIẾM

Các thao tác trên cây nhị phân tìm kiếm:

- Thao tác chèn thêm một nút
- Thao tác xóa một nút trên cây nhị phân tìm kiếm
- Tìm kiếm trên cây nhị phân tìm kiếm

4.1. THAO TÁC CHÈN MỘT NÚT

Thuật toán thực hiện thao tác chèn như sau:

- **Bước 1**: Tạo một nút mới NewNode có thành phần dữ liệu NewKey
- **Bước 2**: Gán con trỏ Root = BST, PrtTree = NULL
- **Bước 3**: Nếu((Root == NULL) || (Root->infor == Newkey))

  Chuyển đến bước 8

- **Bước 4**: Ngược lại: Gán PrtTree = Root;
- **Bước 5**: Nếu Root->infor > NewKey

  Root = Root->left;
- **Bước 6**: Nếu Root->infor < NewKey

  Root = Root->right;

4.1. THAO TÁC CHÈN MỘT NÚT

Thuật toán thực hiện thao tác chèn như sau:

- **Bước 7**: lặp lại bước 3.
- **Bước 8**: Nếu(Root != NULL)

  printf('Trùng khóa của một nút trên cây');
  - Ngược lại nếu (PrtTree == NULL)

    BST = NewNode; {* cây BST trống *}
  - Ngược lại:+ Nếu (PrtTree->infor > NewKey

    PrtTree^.left = NewNode;

    + Nếu (PrtTree->infor < NewKey)

    PrtTree->right = NewNode;
- **Bước 9**: Kết thúc

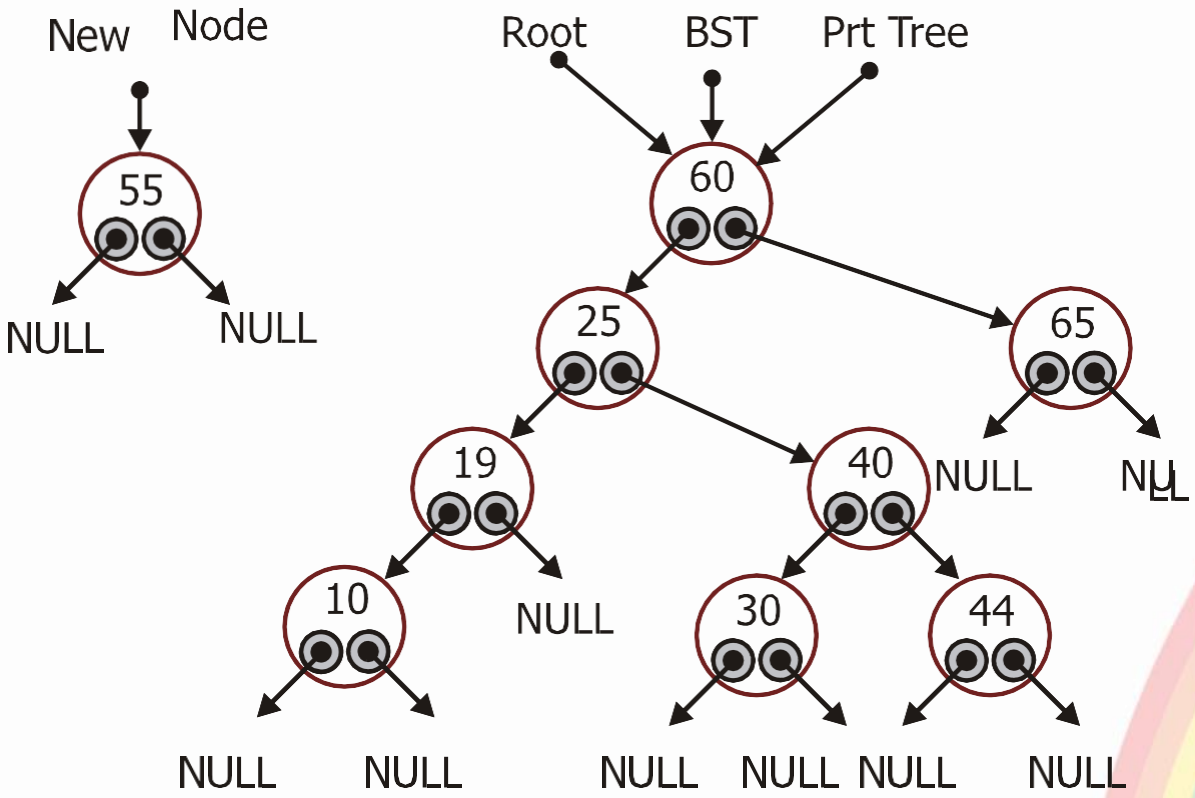
4.1. THAO TÁC CHÈN MỘT NÚT

Cài đặt thuật toán chèn một nút vào cây nhị phân tìm kiếm:

```
void BSTinseart (Tree BST, KeyType NewKey)
{   Tree Root, PrtTree;                Root=BST;        PrtTree=NULL;
    while((Root!=NULL)&&(Root->infor!=NewKey))
    {PrtTree=Root;
        if (Root->infor>NewKey) Root=Root->left;
        else if (Root->infor<NewKey) Root=Root->right; }
        if(Root!=NULL) printf("da co phan tu NewKey tren cay");
        else {Root=(Tree)malloc(sizeof(Tree));
            Root->infor=NewKey;
            Root->left=NULL;        Root->right=NULL;
            if(PrtTree==NULL) BST=Root;
            else if(PrtTree->infor>NewKey) PrtTree->left=Root;
                    else       PrtTree->right=Root;
        }
     }
```

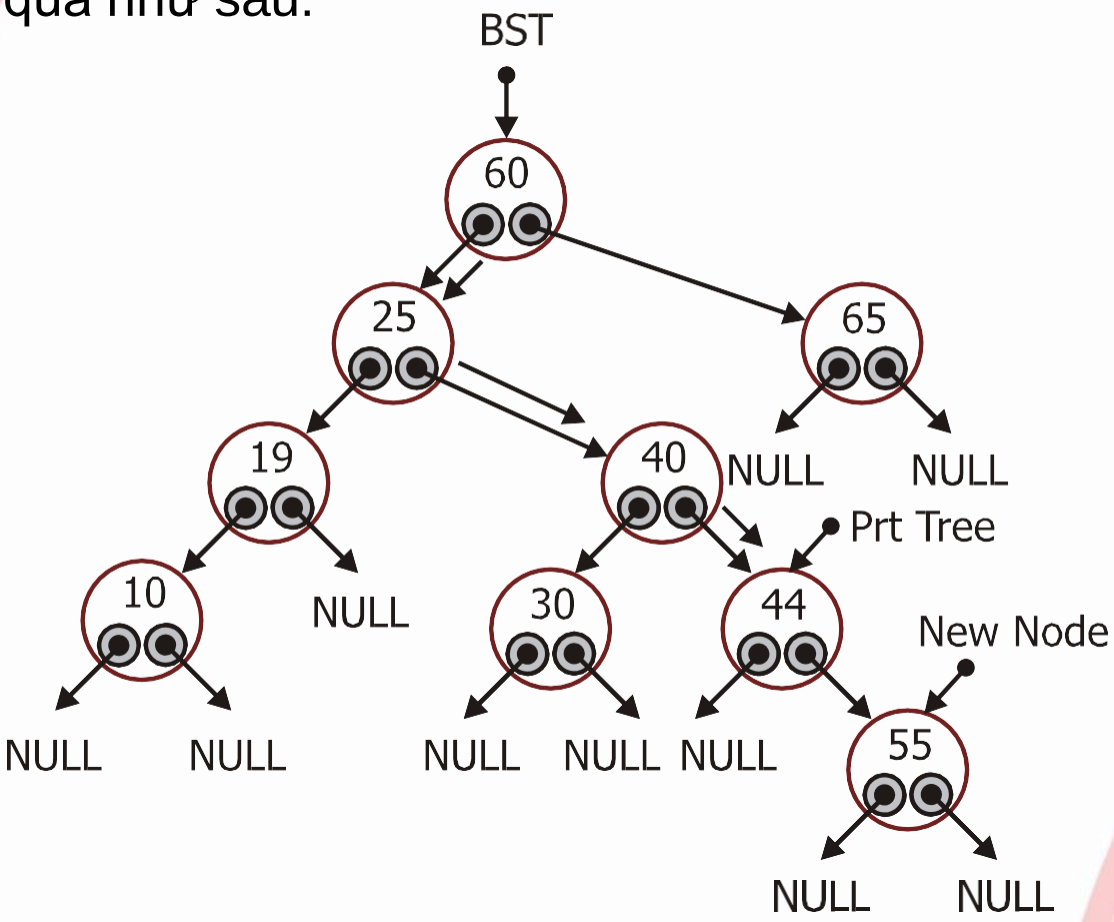
4.1. THAO TÁC CHÈN MỘT NÚT

Ví dụ 6.6: Chèn một nút mới NewNode chứa dữ liệu 55 vào cây nhị phân tìm kiếm sau:

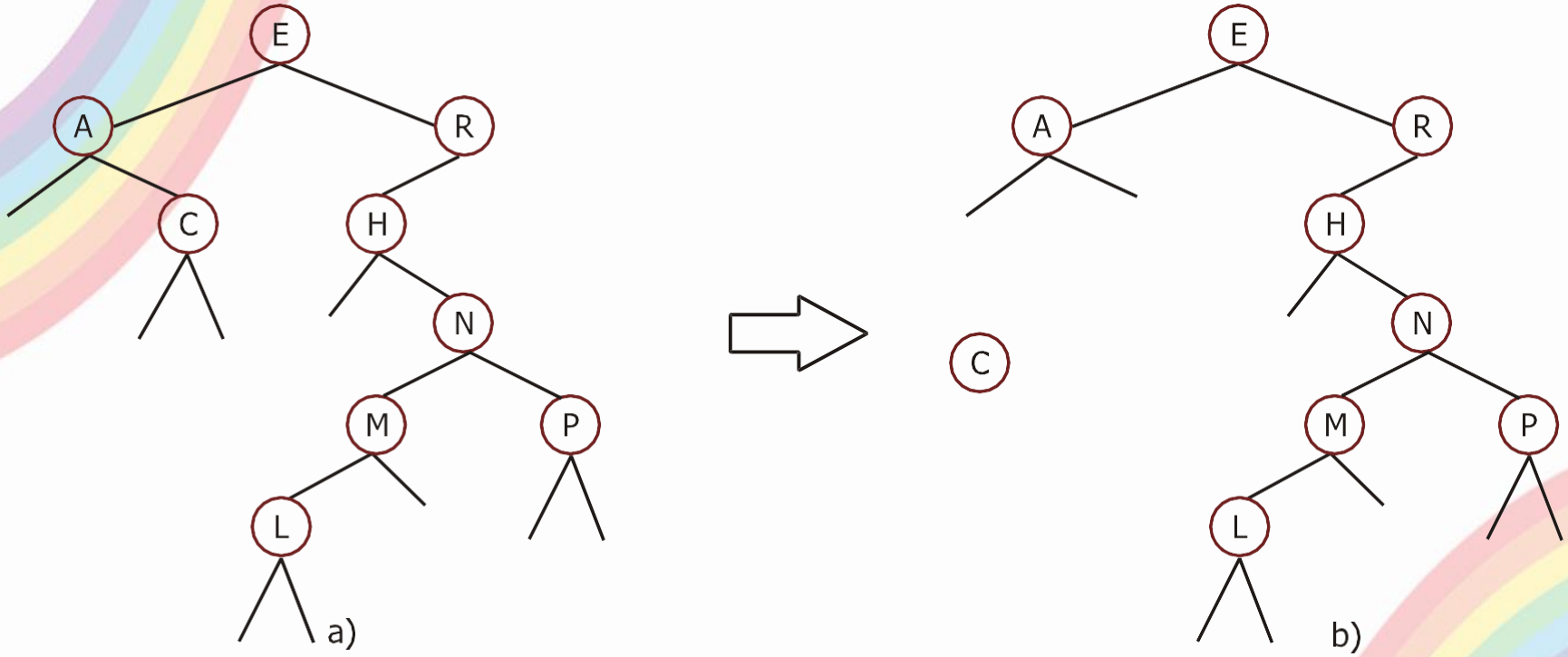


4.1. THAO TÁC CHÈN MỘT NÚT

Ta có kết quả như sau:



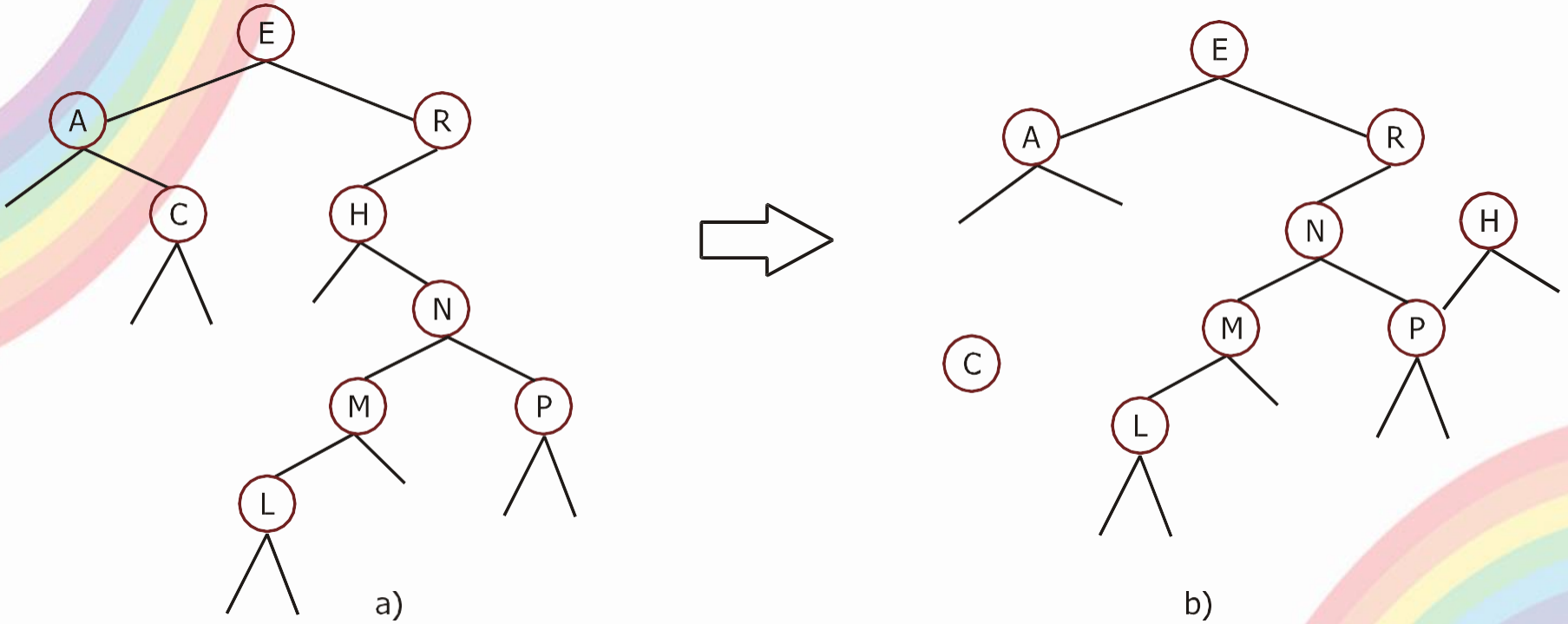
4.2. THAO TÁC XOÁ MỘT NÚT TRÊN CÂY NHỊ PHÂN TÌM KIẾM



Việc xóa một nút trên cây nhị phân tìm kiếm vẫn phải bảo đảm cây sau khi xóa là cây nhị phân tìm kiếm. Việc xóa một nút trên cây nhị phân tìm kiếm có thể xảy ra ở một trong ba trường hợp sau:

- Nút xóa là nút lá;

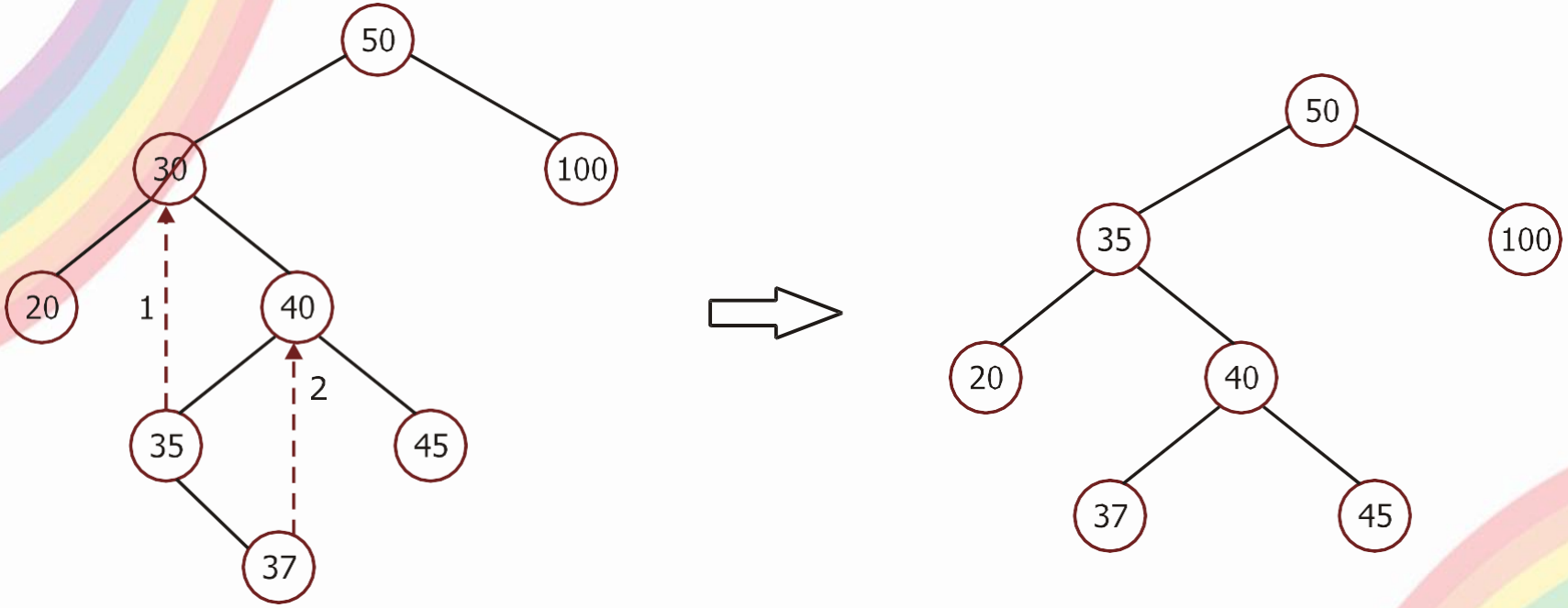
4.2. THAO TÁC XOÁ MỘT NÚT TRÊN CÂY NHỊ PHÂN TÌM KIẾM



Việc xóa một nút trên cây nhị phân tìm kiếm vẫn phải bảo đảm cây sau khi xóa là cây nhị phân tìm kiếm. Việc xóa một nút trên cây nhị phân tìm kiếm có thể xảy ra ở một trong ba trường hợp sau:

- Nút xóa là nút chỉ có một nút gốc cây con;

4.2. THAO TÁC XOÁ MỘT NÚT TRÊN CÂY NHỊ PHÂN TÌM KIẾM



Việc xóa một nút trên cây nhị phân tìm kiếm vẫn phải bảo đảm cây sau khi xóa là cây nhị phân tìm kiếm. Việc xóa một nút trên cây nhị phân tìm kiếm có thể xảy ra ở một trong ba trường hợp sau:

- Nút xóa có đủ hai nút gốc cây con.

4.2. THAO TÁC XOÁ MỘT NÚT TRÊN CÂY NHỊ PHÂN TÌM KIẾM

Cài đặt thao tác xóa một nút cho cây nhị phân tìm kiếm:

```
void BSTDelete(Tree BST, KeyType X)
{   Tree Root,PrtTree,Parent;
    Root=BST;     Parent=NULL;
    while(Root!=NULL)
    {       if(Root->infor==X) break;
            Parent=Root;
            if(Root->infor<X) Root=Root->left;
            else Root=Root->right;
    }
    if(Root==NULL)
            printf("khong có phan tu nay tren cay");
```

4.2. THAO TÁC XOÁ MỘT NÚT TRÊN CÂY NHỊ PHÂN TÌM KIẾM

Cài đặt thao tác xóa một nút cho cây nhị phân tìm kiếm:

```
else
    {if((Root->left!=X)&&(Root->right!=X)
    {
        • PrtTree=Root->right; Parent=Root;
        • while(PrtTree->left!=NULL)
            • { Parent=PrtTree;  PrtTree=PrtTree->left;
        • }
    • Root->infor=PrtTree->infor;
        }
        PrtTree=Root->left;                PrtTree=Root->right;
        if (PrtTree==NULL)                 Root=PrtTree;
        if (Parent==NULL)
    else if(Parent->left==Root)Parent->left=PrtTree;
        else       Parent->right=PrtTree;
}       }
```

4.2. THAO TÁC XOÁ MỘT NÚT TRÊN CÂY NHỊ PHÂN TÌM KIẾM

- Trong trường hợp trung bình, thao tác xóa trên cây nhị phân tìm kiếm có độ phức tạp là O(lgn);
- Còn trong trường hợp xấu nhất cây nhị phân tìm kiếm bị suy biến thì thao tác này có độ phức tạp là O(n) với n là số nút trên cây nhị phân tìm kiếm.

4.3. TÌM KIẾM TRÊN CÂY NHỊ PHÂN TÌM KIẾM

Việc tìm một khóa trên cây nhị phân tìm kiếm có thể thực hiện bằng thuật toán tìm kiếm nhị phân.

- Chúng ta bắt đầu từ gốc;
- Nếu khóa cần tìm bằng khóa của gốc thì khóa đó trên cây;
- Nếu khóa cần tìm nhỏ hơn khoa ở gốc, ta phải tìm nó trên cây con trái;
- Nếu khóa cần tìm lớn hơn khóa ở gốc, ta phải tìm nó trên cây con phải;
- Nếu cây con (trái hoặc phải) là rỗng thì khóa cần tìm không có trên cây.

4.3. TÌM KIẾM TRÊN CÂY NHỊ PHÂN TÌM KIẾM

Thuật toán thực hiện thao tác tìm kiếm trên cây nhị phân tìm kiếm:

- Bước 1: đặt con trỏ Root=BST;
- Bước 2: nếu (Root=NULL) hoặc (Root->infor=X) Kết thúc thuật toán;
- Bước 3: ngược lại:nếu (Root->infor>X) Root=Root->left;//tìm X ở cây con bên trái
- Bước 4: ngược lại nếu(Root->infor<X) Root=Root->right;//tìm X ở cây con bên phải
- Bước 5: lặp lại bước 2;

Như vậy thuật toán kết thúc việc tìm kiếm khi cây con trống hoặc nút gốc của cây con có giá trị khóa cần tìm.

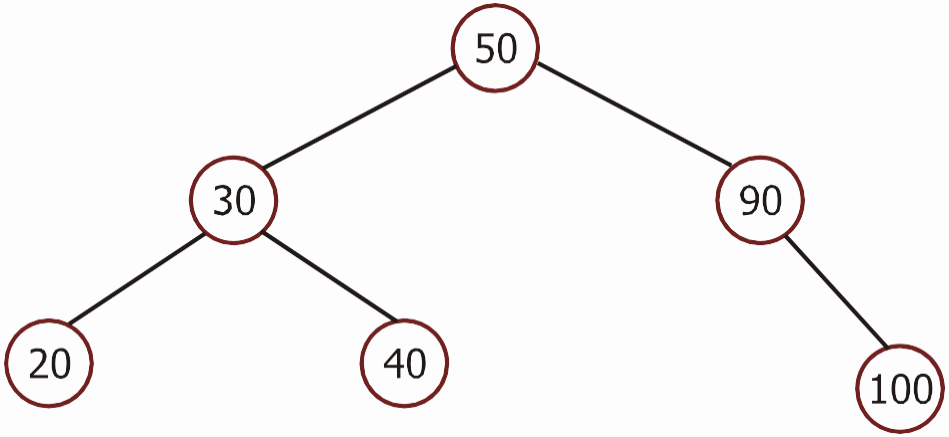
4.3. TÌM KIẾM TRÊN CÂY NHỊ PHÂN TÌM KIẾM

Cài đặt thuật toán:

```
Tree BSTSearch(Tree BST,KeyType X)
{    Tree Root;
     Root=BST;
     while((Root!=NULL)&&(Root->infor!=X))
     {       if(Root->infor>X) Root=Root->left;
                        else if(Root->infor<X) Root=Root->right;
     }
     return Root;
}
```

4.3. TÌM KIẾM TRÊN CÂY NHỊ PHÂN TÌM KIẾM

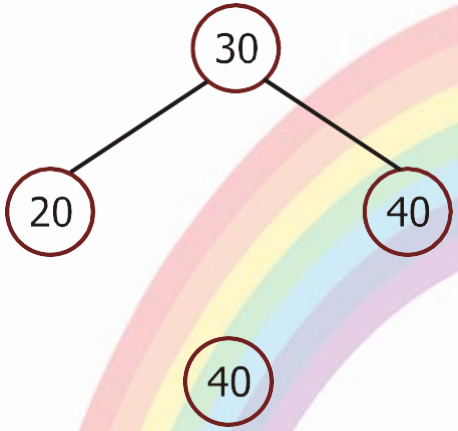
Ví dụ 6.10: Xét cây nhị phân tìm kiếm có dạng sau:



Ta tìm nút có thành phần dữ liệu là 40.

Bắt đầu từ nút gốc, vì 40 nhỏ hơn giá trị 50 của nút gốc nên cần tìm ở bên trái của nút gốc.

Tiếp tục tìm kiếm bằng cách so sánh 40 với gốc của cây con này. Vì 40 > 30 nên nút cần tìm sẽ nằm ở cây con bên phải đó là:



Ta tìm được nút chứa giá trị cần tìm.

TÓM LƯỢC CUỐI BÀI

- Tìm kiếm là công việc rất hay được sử dụng nhiều trong các ứng dụng. Trên đây, ta đã trình bày phép tìm kiếm trong một danh sách để tìm ra mẩu tin có khóa đúng bằng khóa tìm kiếm;
- Tuy nhiên, trong nhiều trường hợp ta có thể chuyển sang tìm mẩu tin mang khóa lớn hơn hay nhỏ hơn khóa tìm kiếm, tìm mẩu tin mang khóa nhỏ nhất mà lớn hơn khóa tìm kiếm v.v…;
- Để cài đặt những thuật toán cho các trường hợp này cần có sự mềm dẻo nhất định;
- Ngoài ra chúng ta cũng không nên đánh giá giải thuật tìm kiếm này tốt hơn giải thuật tìm kiếm khác mà căn cứ vào từng yêu cầu cụ thể để đưa ra giải thuật tìm kiếm cho phù hợp nhất.