Cấu trúc dữ liệu & Giải thuật
(Data Structures and Algorithms)

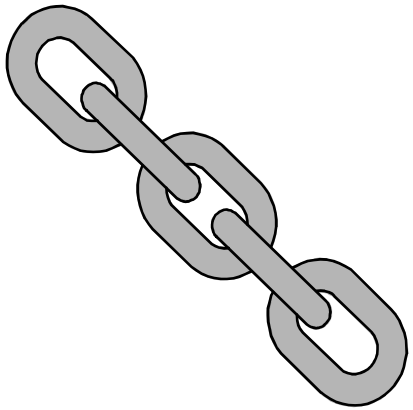
Các cấu trúc dữ liệu

Nguyễn Tri Tuấn
Khoa CNTT – ĐH.KHTN.Tp.HCM
Email: nttuan@fit.hcmus.edu.vn

Nội dung

Các cấu trúc dữ liệu cơ bản

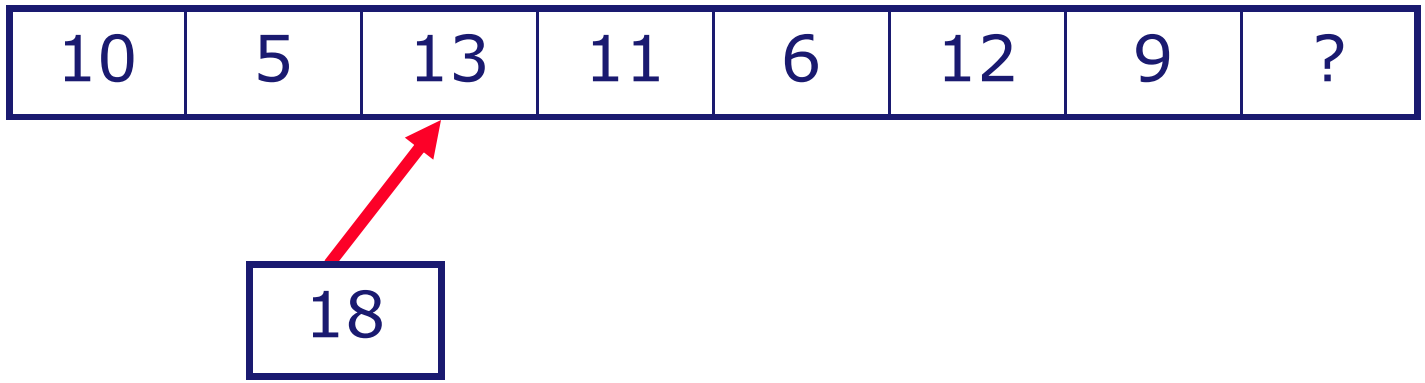
Danh sách liên kết – Linked Lists



- Đặt vấn đề
- Danh sách liên kết là gì ?
- So sánh Mảng và Danh sách liên kết
- Danh sách liên kết đơn
- Giới thiệu các loại danh sách liên kết khác

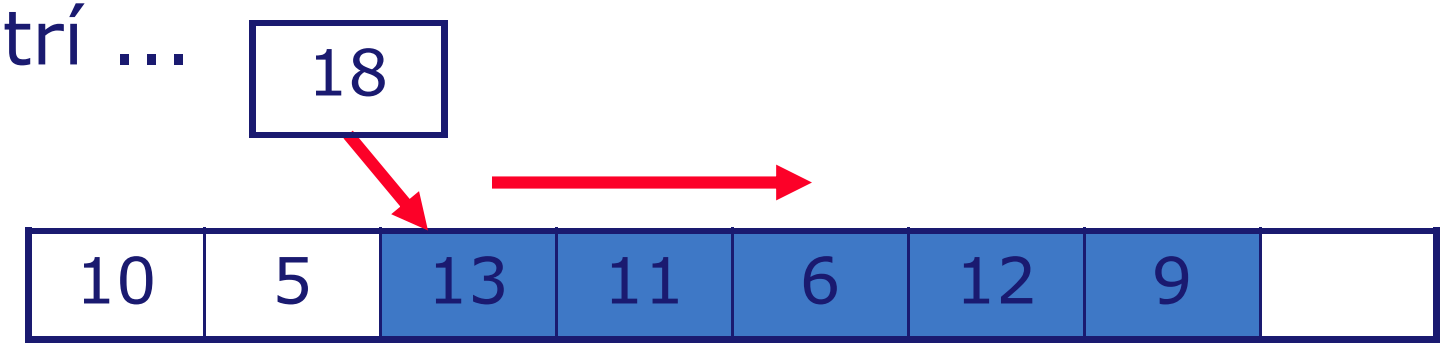
Đặt vấn đề [1/5]

- Nếu muốn thêm (Insert) 1 phần tử vào mảng, phải làm sao ?

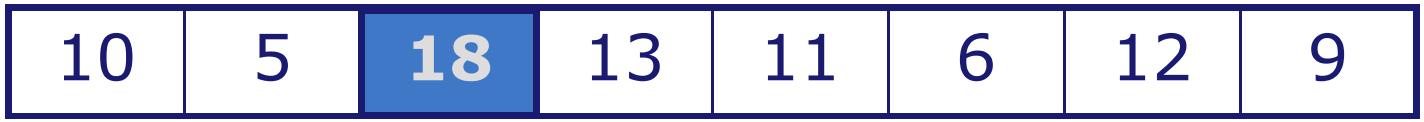


Đặt vấn đề [2/5]

- Phải di chuyển các phần tử về phía sau 1 vị trí ...



- …rồi chèn phần tử mới vào



- Vậy chi phí là **O(n)**

Đặt vấn đề [3/5]

- Tương tự, chi phí xóa 1 phần tử trong mảng cũng là **O(n)**

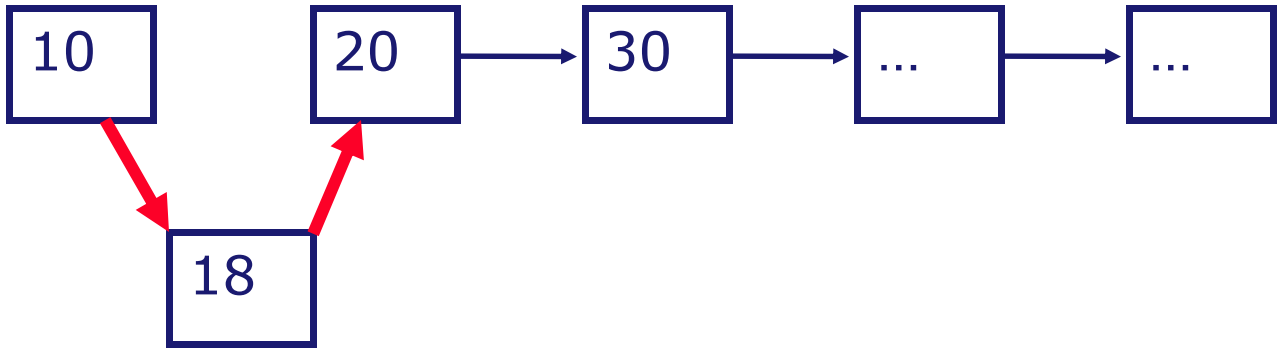
- Làm sao có thể thêm (hay xoá) 1 phần tử mà không phải di chuyển các phần tử khác ?

Đặt vấn đề [4/5]

- Ta tách rời các phần tử của mảng, và kết nối chúng lại với nhau bằng một "móc xích"

Đặt vấn đề [5/5]

- Thao tác thêm phần tử chỉ cần thay đổi các mối liên kết tại chỗ



- Chi phí **O(1)**

Danh sách liên kết là gì ?

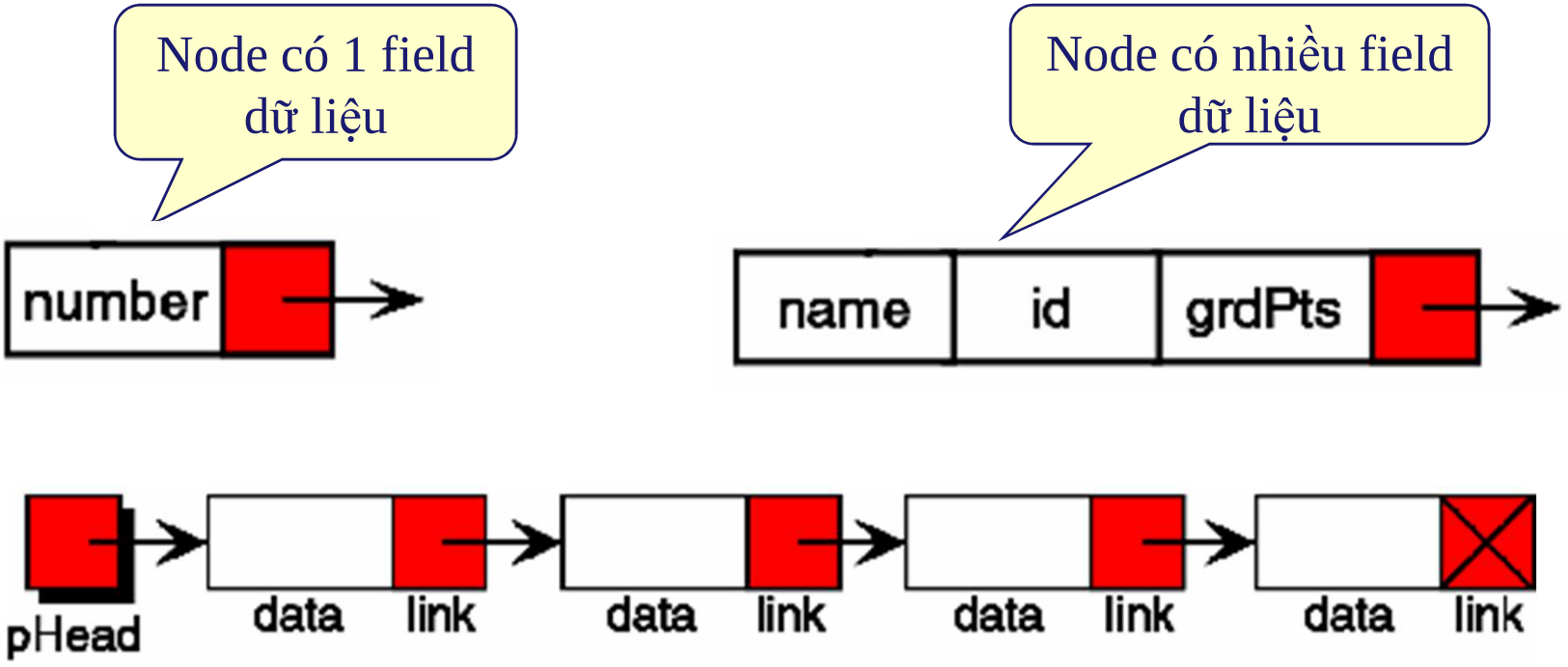
- Một dãy tuần tự các nút (Node)
- Giữa hai node có con trỏ liên kết
- Các node không cần phải lưu trữ liên tiếp nhau trong bộ nhớ
- Có thể mở rộng tuỳ ý (chỉ giới hạn bởi dung lượng bộ nhớ)
- Thao tác Chèn/Xóa không cần phải dịch chuyển phần tử

So sánh Mảng và Danh sách liên kết

Mảng	Danh sách liên kết
Kích thước cố định	Số phần tử thay đổi tùy ý
Các phần tử lưu trữ tuần tự (địa chỉ tăng dần) trong bộ nhớ	Các phần tử lưu rời rạc, liên kết với nhau bằng con trỏ
Phải tịnh tiến các phần tử khi muốn Chèn/Xóa 1 phần tử - O(n)	Chỉ cần thay đổi con trỏ liên kết khi muốn Chèn/Xóa 1 phần tử - O(1)
Truy xuất ngẫu nhiên (nhanh)	Truy xuất tuần tự (chậm)

Danh sách liên kết đơn [1/10]

- Đặc điểm:
  - Mỗi node chỉ có 1 con trỏ liên kết (đến node kế tiếp trong danh sách)



Danh sách liên kết đơn [2/10]

- Các thao tác cơ bản trên DSLK đơn
  - Khởi tạo danh sách rỗng
  - Thêm 1 node vào đầu danh sách
  - Xóa node ở đầu danh sách
  - Duyệt danh sách
  - Tìm 1 phần tử
  - Kiểm tra danh sách rỗng
  - Kiểm tra số phần tử trong danh sách

Danh sách liên kết đơn [3/10]

- Khai báo cấu trúc DSLK đơn bằng struct

```
// cấu trúc lưu trữ 1 node
typedef  struct tagSTUDENT {
    char        name[30];        // họ tên
    int         id;              // mã số
    float       grdPts;          // điểm TB
    tagSTUDENT *pNext;
} STUDENT;
// cấu trúc quản lý DSLK đơn
typedef struct STUDENT_LIST {
    STUDENT         *pHead;
    unsigned int    nCount;   // số node trong danh sách
};
```

Danh sách liên kết đơn [4/10]

- Khai báo cấu trúc DSLK đơn bằng struct (tt)

```
// Các hàm xử lý thao tác
void  CreateEmptyList(STUDENT_LIST &list);
int   AddNode(STUDENT_LIST  &list, char *new_Name,
                        int new_Id, float new_Pts);
int   DeleteNode(STUDENT_LIST  &list);
void  TraverseList(const STUDENT_LIST &list);
STUDENT* FindNode(const STUDENT_LIST &list, int key);
int   IsEmptyList(const STUDENT_LIST &list);
int   CountNode(const STUDENT_LIST &list);
```

Danh sách liên kết đơn [5/10]

- Khai báo cấu trúc DSLK đơn bằng class

```
class STUDENT {                          // cấu trúc 1 node trong danh sách
        public:
          char          name[30];        // họ tên
          int           id;              // mã số
          float         grdPts;          // điểm TB
          STUDENT  *pNext;               // trỏ đến node kế tiếp
};
class STUDENT_LIST {
        private:
           STUDENT *pHead;               // trỏ đến phần tử đầu danh sách
           unsigned int nCount;          // Số node trong danh sách
        public:
           // constructors và destructor
           STUDENT_LIST();
           ~STUDENT_LIST();
           // các thao tác
           int       IsEmpty();
           int       Count();
           int       AddNode(char *aName, int aID, float aPts);
           int       DeleteNode();
           void      Traverse();
           STUDENT *FindNode(int key);
}; // end class
```

Danh sách liên kết đơn [6/10]

```
// Tạo danh sách rỗng
void  CreateEmptyList(STUDENT_LIST &list)
{
    list.nCount = 0;
    list.pHead = NULL;
}

// Kiểm tra danh sách rỗng
int  IsEmptyList(const STUDENT_LIST &list)
{
    return (list.pHead == NULL);
}

// Kiểm tra số phần tử trong danh sách
int  CountNode(const STUDENT_LIST &list)
{
    return list.nCount;
}
```

Danh sách liên kết đơn [7/10]

```
// Thêm node mới vào đầu danh sách
int  AddNode(STUDENT_LIST  &list, char *new_Name, int new_Id,
                    float new_Pts)
{
   // Cấp phát node mới
   STUDENT *pNew = new STUDENT;
   if (pNew==NULL)  return 0; // Lỗi không thể cấp phát node mới

   strcpy(pNew->name, new_Name);
   pNew->id = new_Id;
   pNew->grdPts = new_Pts;

   pNew->pNext = list.pHead;
   list.pHead = pNew;
   list.nCount++;
   return 1;              // thành công
}
```

Danh sách liên kết đơn [8/10]

```
// Xoá node ở đầu danh sách
int DeleteNode(STUDENT_LIST  &list)
{
   if (IsEmptyList(list)) return 0;     // danh sách rỗng

   STUDENT *pTemp;
   pTemp = list.pHead;                  // lưu lại địa chỉ node đầu
   list.pHead = list.pHead->pNext;
   delete pTemp;                        // xoá node
   list.nCount--;
   return 1;    // xoá thành công
}
```

Danh sách liên kết đơn [9/10]

```
// Duyệt danh sách
void     TraverseList(const STUDENT_LIST &list)
{
   // bắt đầu từ đầu danh sách
   STUDENT  *pCurr = list.pHead;

   while (pCurr!=NULL) {
        // Xử lý node pCurr, vd. In giá trị của node
        printf("Name:%s, Id:%d, Point:%.2f\n",
                  pCurr->name, pCurr->id, pCurr->grdPts);
        pCurr = pCurr->pNext;  // chuyển sang node kế
   }
}
```

Danh sách liên kết đơn [10/10]

```
// tìm node có khoá = key trong danh sách
STUDENT* FindNode(const STUDENT_LIST &list, int key)
{
   // bắt đầu từ đầu danh sách
   STUDENT  *pCurr = list.pHead;

   while (pCurr!=NULL) {
        if (pCurr->id==key)  return  pCurr;   // Tìm thấy
        pCurr = pCurr->pNext;  // chuyển sang nút kế
   }
   return NULL;         // Không tìm thấy
}
```

Các cấu trúc dữ liệu cơ bản

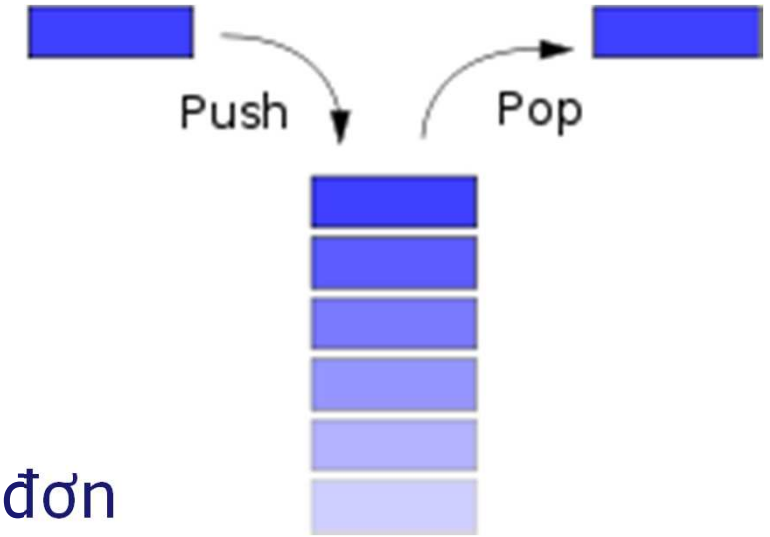
1.1 Các danh sách liên kết – Linked Lists

1.2 Ngăn xếp – Stack

1.3 Hàng đợi - Queue

Ngăn xếp - Stack

- Định nghĩa
- Các thao tác cơ bản
- Xây dựng Stack bằng mảng
- Xây dựng Stack bằng DSLK đơn
- Ứng dụng Stack



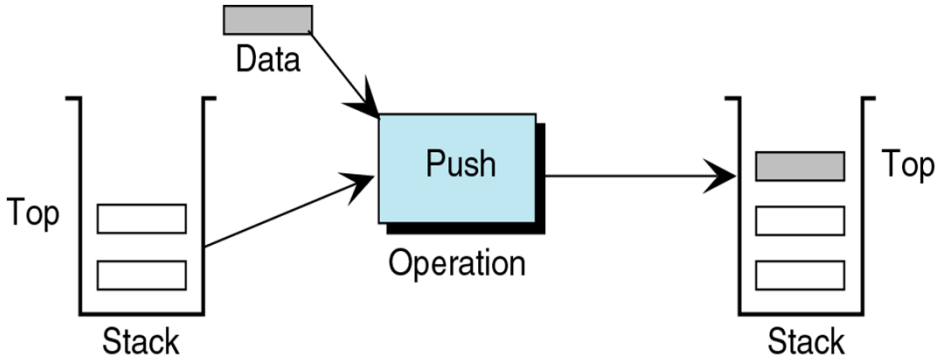
Định nghĩa

- Stack là một cấu trúc dữ liệu:
  - Dùng để lưu trữ nhiều phần tử dữ liệu
  - Hoạt động theo cơ chế "Vào sau – Ra trước" (Last In/First Out – LIFO)

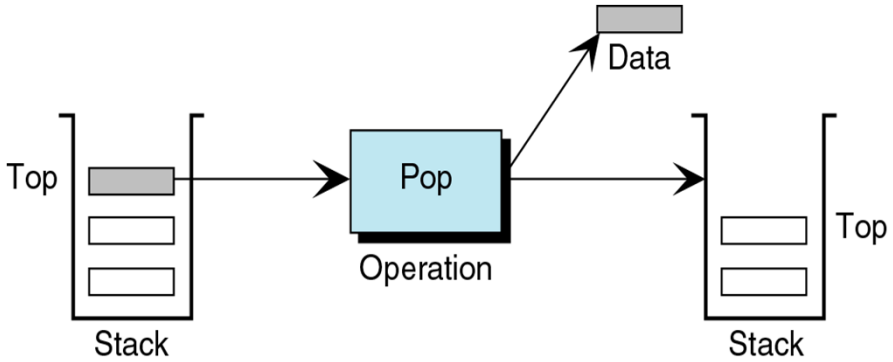
*** Cấu trúc Stack được phát minh năm 1955, được đăng ký bản quyền năm 1957, bởi tác giả Friedrich L. Bauer (người Đức)*

Các thao tác cơ bản [1/2]

- Thao tác cơ bản trên Stack chỉ bao gồm:
  - Push: thêm 1 phần tử vào đỉnh Stack



  - Pop: lấy ra 1 phần tử ở đỉnh Stack



Các thao tác cơ bản [2/2]

- Tuy nhiên, trong thực tế người ta cũng thường cài đặt thêm các thao tác phụ trợ khác:
  - InitStack: khởi tạo Stack rỗng
  - IsEmpty: kiểm tra Stack rỗng ?
  - IsFull: kiểm tra Stack đầy ?
  - Peek: trả về phần tử đầu Stack (mà không loại bỏ nó khỏi Stack)

Xây dựng Stack bằng mảng [1/5]

- Khai báo cấu trúc Stack bằng struct

```
// Giả sử Stack chứa các phần tử kiểu nguyên (int)
// Khai báo cấu trúc Stack
typedef  struct STACK {
   int       *StkArray;    // mảng chứa các phần tử
   int        StkMax;      // số phần tử tối đa
   int        StkTop;      // vị trí đỉnh Stack
}
// Các hàm xử lý thao tác
int  InitStack(STACK &s, int MaxItems);
int  IsEmpty(const STACK &s);
int  IsFull(const STACK &s);
int Push(STACK &s, int newitem);
int  Pop(STACK &s, int &outitem);
int  Peek(const STACK &s, int &outitem);
```

Xây dựng Stack bằng mảng [2/5]

- Khai báo cấu trúc Stack bằng class

```
class STACK {
    private:
      int *StkArray;
      int   StkMax;
      int   StkTop;
      int   IsEmpty();
      int   IsFull();
    public:
      // constructors và destructor
      STACK (int MaxItems);
      ~STACK();
      // các thao tác
      int   Push(int newitem);
      int   Pop(int &outitem);
      int   Peek(int &outitem);
};
```

Xây dựng Stack bằng mảng [3/5]

```
// khởi tạo stack rỗng, có tối đa MaxItems phần tử
int  InitStack(STACK &s, int MaxItems)
{
    s.StkArray = new int[MaxItems];
    if (s.StkArray==NULL)
          return 0;          // Fail. Không cấp phát được bộ nhớ
    s.StkMax = MaxItems;
    s.StkTop = -1;           // chưa có phần tử nào trong Stack
    return 1;                // khởi tạo thành công
}

// kiểm tra stack rỗng ?
int  IsEmpty(const STACK &s)
{
    if (s.StkTop==-1) return 1;  // Stack rỗng
    return 0;                    // Stack không rỗng
}
```

Xây dựng Stack bằng mảng [4/5]

```
// kiểm tra stack đầy ?
int  IsFull(const STACK &s)
{
    if (s.StkTop==s.StkMax-1)
       return 1;          // Stack đầy
    return 0;             // Stack chưa đầy
}

// Thêm 1 phần tử vào đỉnh stack
int Push(STACK &s, int newitem)
{
    if (IsFull(s))
       return 0;          // Stack đầy, không thêm vào được

    s.StkTop++;
    s.StkArray[s.StkTop] = newitem;
    return 1;             // Thêm thành công
}
```

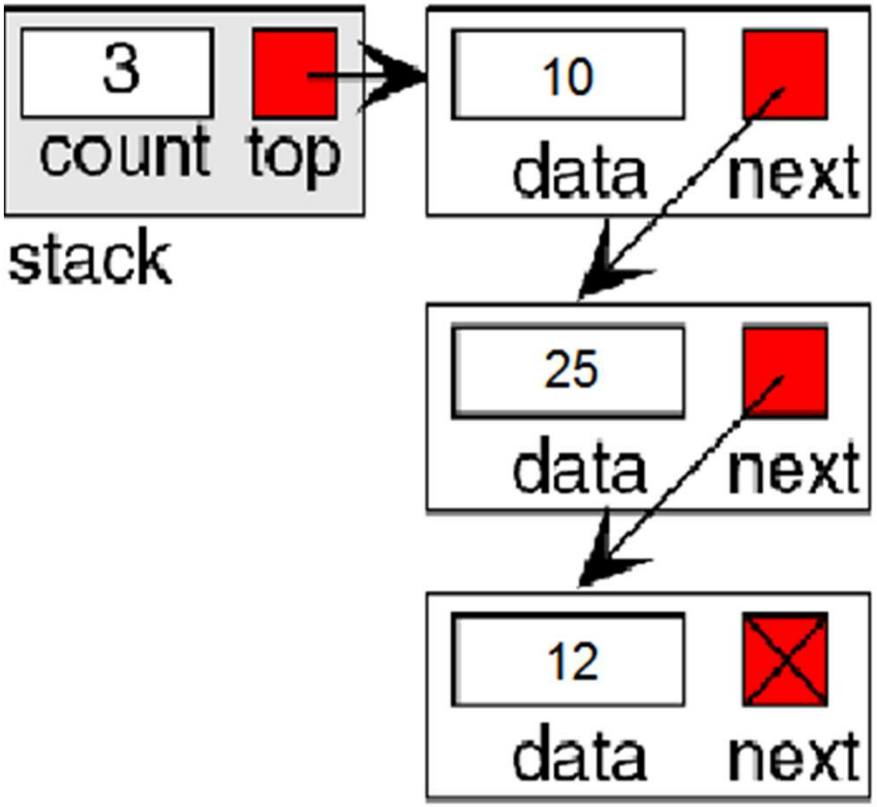
Xây dựng Stack bằng mảng [5/5]

```
// lấy ra 1 phần tử ở đỉnh stack
int Pop(STACK &s, int &outitem)
{
    if (IsEmpty(s))
        return 0;          // Stack rỗng, không lấy ra được
    outitem = s.StkArray[s.StkTop];
    s.StkTop--;
    return 1;              // Lấy ra thành công
}

// trả về phần tử ở đỉnh stack (không xóa)
int Peek(const STACK &s, int &outitem)
{
    if (IsEmpty(s))
        return 0;          // Stack rỗng, không lấy ra được
    outitem = s.StkArray[s.StkTop];
    return 1;              // Lấy ra thành công
}
```

Áp dụng

- Viết lệnh để thực hiện các yêu cầu sau đây:
  - Khai báo biến ngăn xếp S1
  - Khởi tạo S1 có 500 phần tử
  - Đưa các giá trị sau vào S1: 15, 8, 6, 21
  - Lấy 21 ra khỏi S1
  - Lấy 8 ra khỏi S1
  - Gán các giá trị 1-99 vào S1
  - Lần lượt lấy các phần tử trong S1 và in lên màn hình
  - Cho mảng a chứa dãy số nguyên từ 1-N, hãy đảo ngược các phần tử của mảng a

Xây dựng Stack bằng DSLK đơn [1/7]



Hình minh họa cấu trúc Stack sử dụng DSLK đơn

Xây dựng Stack bằng DSLK đơn [2/7]

- Khai báo cấu trúc Stack bằng struct

```
// cấu trúc 1 phần tử trong Stack
typedef struct tagSTACK_ITEM {
    int                 Data;
    tagSTACK_ITEM       *pNext;
} STACK_ITEM;

// cấu trúc quản lý Stack
typedef struct STACK {
    unsigned int    StkCount;    // số phần tử trong stack
    STACK_ITEM      *StkTop;     // con trỏ đến đỉnh stack
};
```

Xây dựng Stack bằng DSLK đơn [3/7]

- Khai báo cấu trúc Stack bằng struct (tt)

```
// Các hàm xử lý thao tác
int  InitStack(STACK &s);
int  IsEmpty(const STACK &s);
int  Push(STACK &s, int newitem);
int  Pop(STACK &s, int &outitem);
int  Peek(const STACK &s, int &outitem);
```

Xây dựng Stack bằng DSLK đơn [4/7]

```
void  InitStack(STACK  &s)
{
   s.StkTop = NULL;
   s.StkCount = 0;
}

int  IsEmpty(const STACK &s)
{
   if (s.StkTop==NULL)
         return 1;        // Stack rỗng
   return 0;              // Stack không rỗng
}
```

Xây dựng Stack bằng DSLK đơn [5/7]

```
int Push(STACK &s, int newitem)
{
   STACK_ITEM  *pNew = new STACK_ITEM;
   if (pNew==NULL)
         return 0;       // Fail. Không còn bộ nhớ trống
   pNew->Data = newitem;
   pNew->pNext = s.StkTop;
   s.StkTop = pNew;
   s.StkCount++;
   return 1;             // thêm thành công
}
```

Xây dựng Stack bằng DSLK đơn [6/7]

```
int Pop(STACK &s, int &outitem)
{
    if (IsEmpty(s))
        return 0;          // Stack rỗng, không lấy ra được
    STACK_ITEM  *temp = s.StkTop;
    outitem = temp->Data;
    s.StkTop = temp->pNext;
    delete temp;
    s.StkCount--;
    return 1;              // Lấy ra thành công
}

int Peek(const STACK &s, int &outitem)
{
    if (IsEmpty(s))
        return 0;          // Stack rỗng, không lấy ra được
    outitem = s.StkTop->Data;
    return 1;              // Lấy ra thành công
}
```

Xây dựng Stack bằng DSLK đơn [7/7]

- Khai báo cấu trúc Stack bằng class

```
class STACK_ITEM {                  // cấu trúc 1 phần tử của stack
        public:
          int               Data;
          STACK_ITEM        *pNext;  // trỏ đến item kế tiếp
};
class STACK {
        private:
          STACK_ITEM        *StkTop;          // trỏ đến đỉnh stack
          unsigned int      StkCount;         // số phần tử trong stack
          int               IsEmpty();

        public:
          // constructors và destructor
          STACK ();
          ~STACK();
          // các thao tác
          int   Push(int newitem);
          int   Pop(int &outitem);
          int   Peek(int &outitem);
};
```

Ứng dụng của Stack

- Tính giá trị biểu thức toán học (thuật toán Balan ngược – Reverse Polish notation)
- Bài toán tìm đường đi trong mê cung, bài toán mã đi tuần, bài toán 8 quân hậu,…
- Khử đệ qui
- …

Thuật toán Balan ngược

- Cho 1 biểu thức ở dạng chuỗi:
  - S = “5 + ((1 + 2) * 4) – 3”
  - Biểu thức gồm các toán tử +,-,*,/ và dấu ngoặc ()
- Tính giá trị biểu thức trên

Các cấu trúc dữ liệu cơ bản

1.1 Các danh sách liên kết – Linked Lists

1.2 Ngăn xếp – Stack

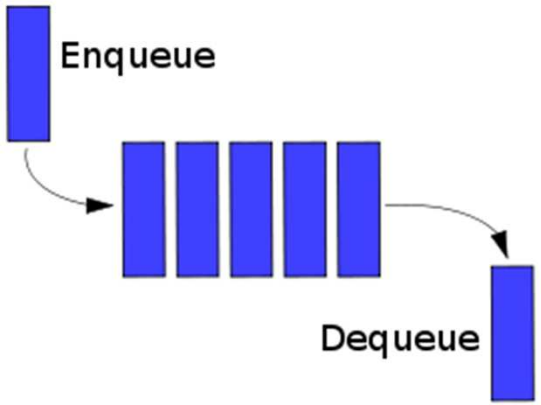
1.3 Hàng đợi - Queue

Hàng đợi - Queue

- Định nghĩa
- Các thao tác cơ bản
- Xây dựng Queue bằng mảng
- Xây dựng Queue bằng DSLK đơn
- Ứng dụng Queue

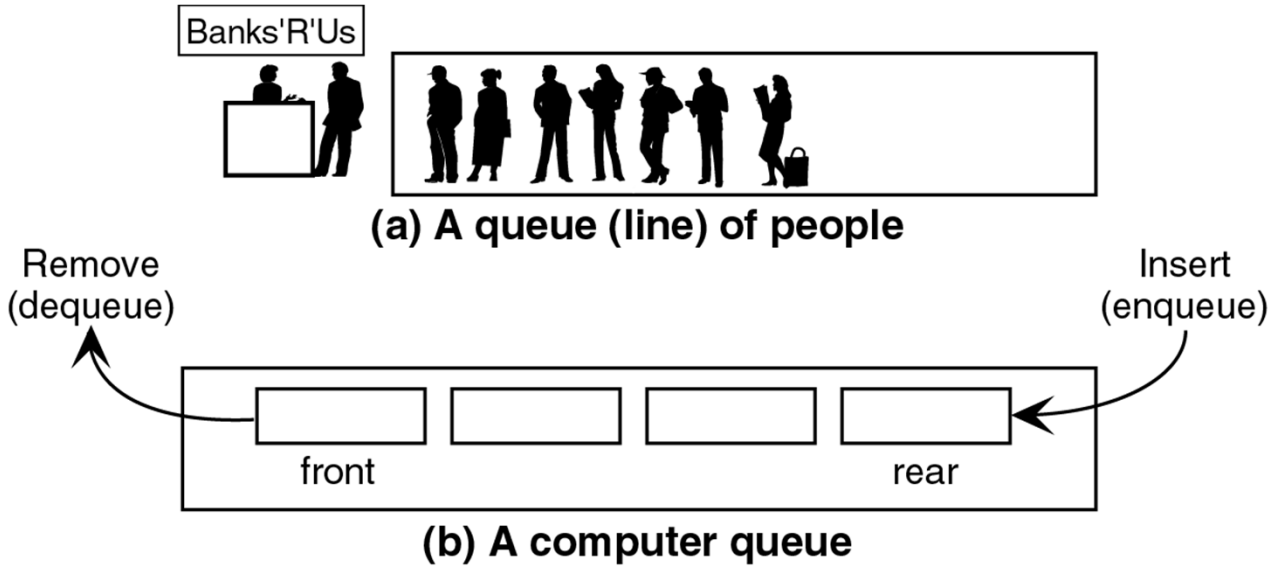
Định nghĩa

- Queue là một cấu trúc dữ liệu:
  - Dùng để lưu trữ nhiều phần tử dữ liệu
  - Hoạt động theo cơ chế "Vào trước – Ra trước" (First In/First Out – FIFO)

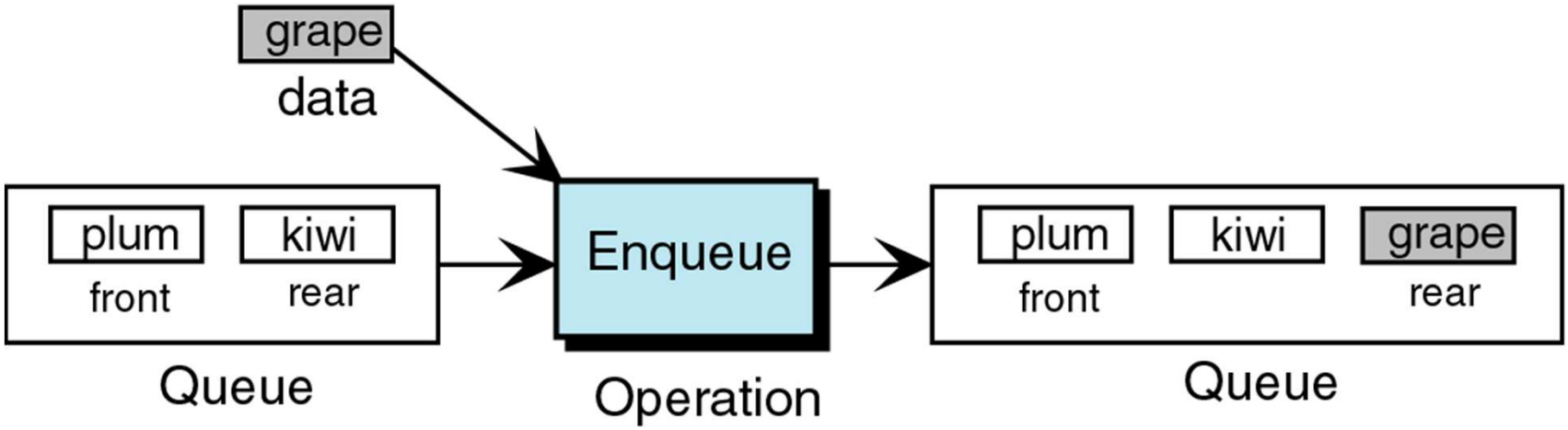


Các thao tác cơ bản [1/4]

- Thao tác cơ bản trên Queue bao gồm:
  - EnQueue: thêm 1 phần tử vào cuối Queue, có thể làm Queue đầy
  - DeQueue: lấy ra 1 phần tử ở đầu Queue, có thể làm Queue rỗng

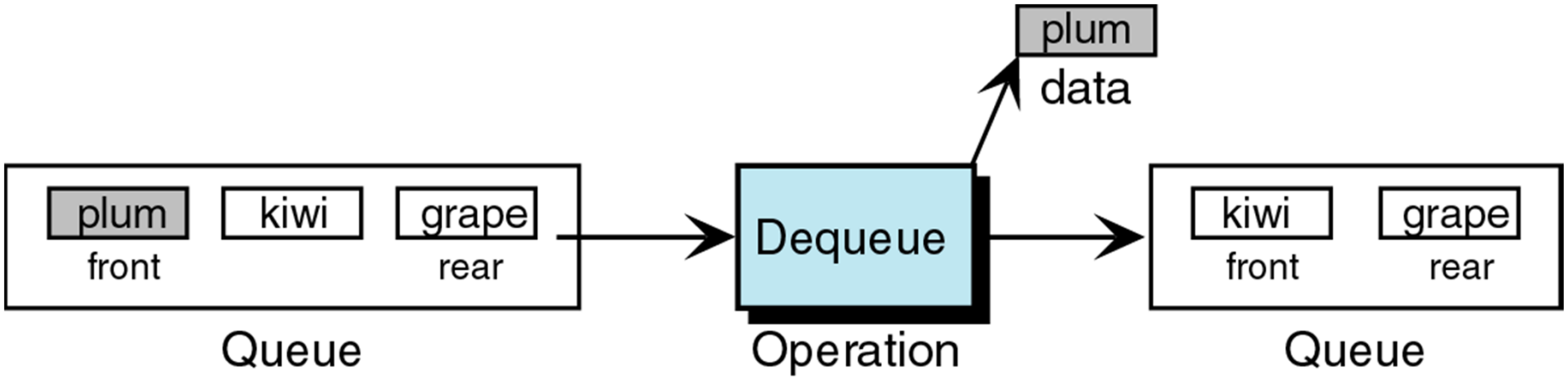


Các thao tác cơ bản [2/4]



Minh họa thao tác EnQueue

Các thao tác cơ bản [3/4]



Minh họa thao tác DeQueue

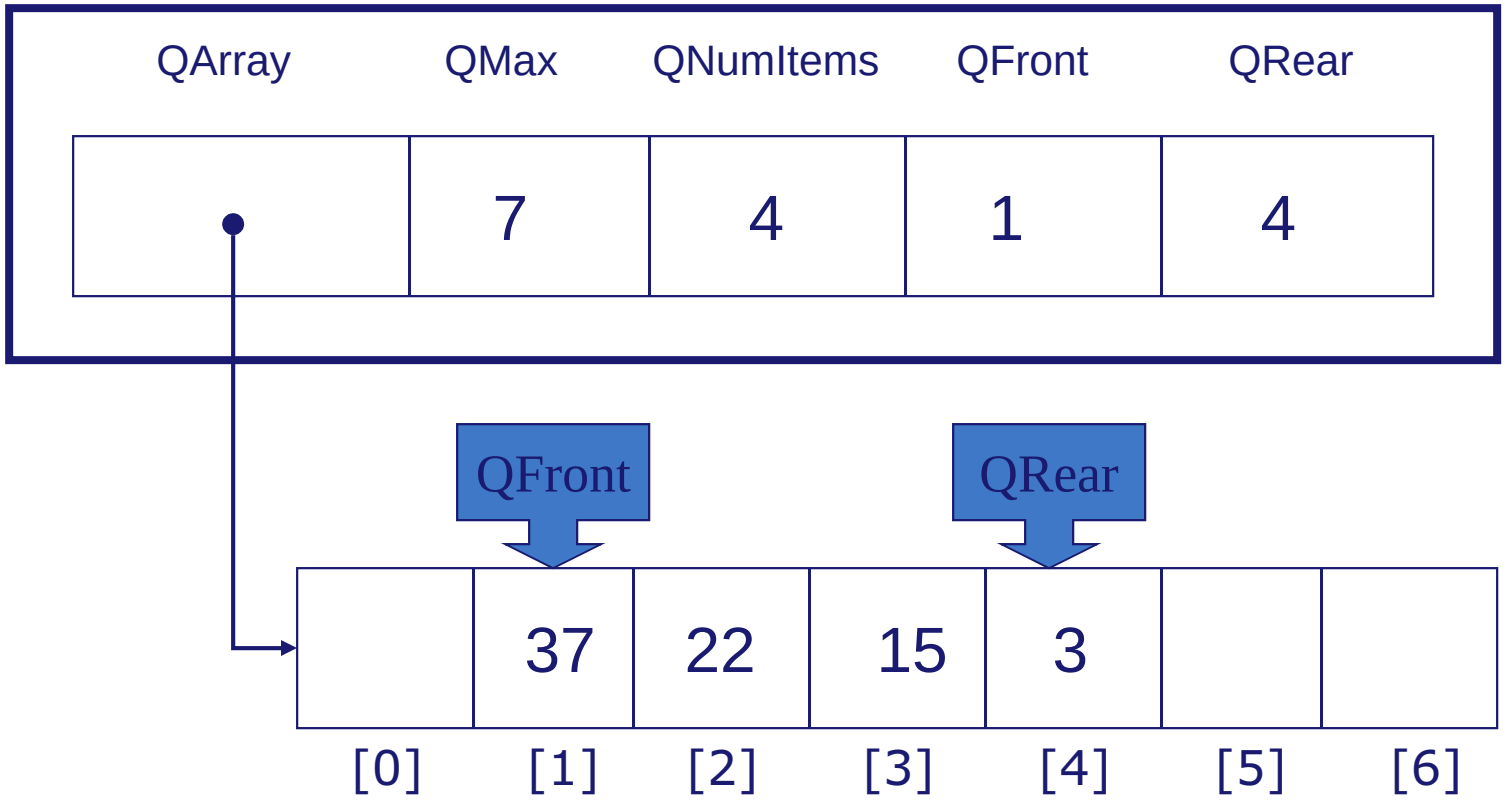
Các thao tác cơ bản [4/4]

- Các thao tác hỗ trợ khác:
  - InitQueue: khởi tạo Queue rỗng
  - IsEmpty: kiểm tra Queue rỗng ?
  - IsFull: kiểm tra Queue đầy ?
  - QueueFront: kiểm tra phần tử ở đầu Queue (không xóa phần tử)
  - QueueSize: cho biết số phần tử hiện đang có trong Queue

Xây dựng Queue [1/12]

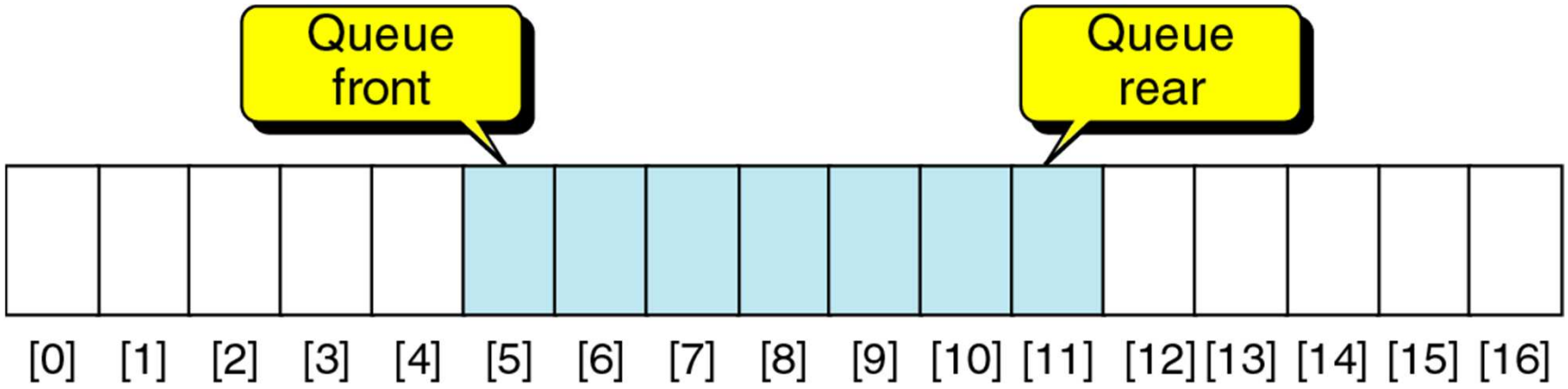
- Có 2 cách xây dựng Queue:
  - Sử dụng mảng 1 chiều
  - Sử dụng danh sách liên kết đơn

Xây dựng Queue - dùng mảng [2/12]



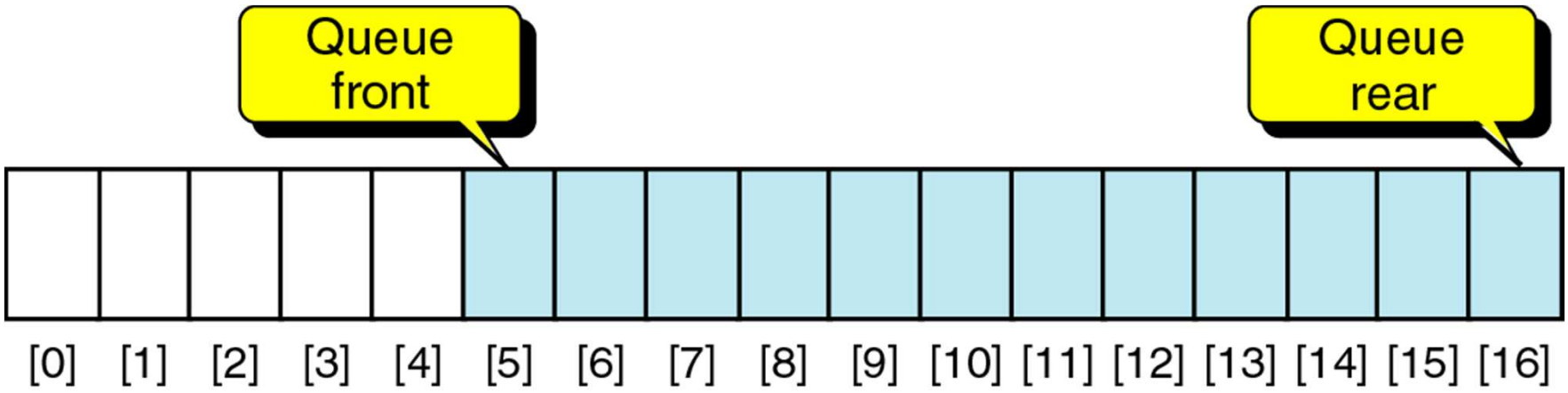
Cấu tạo của Queue

Xây dựng Queue - dùng mảng [3/12]



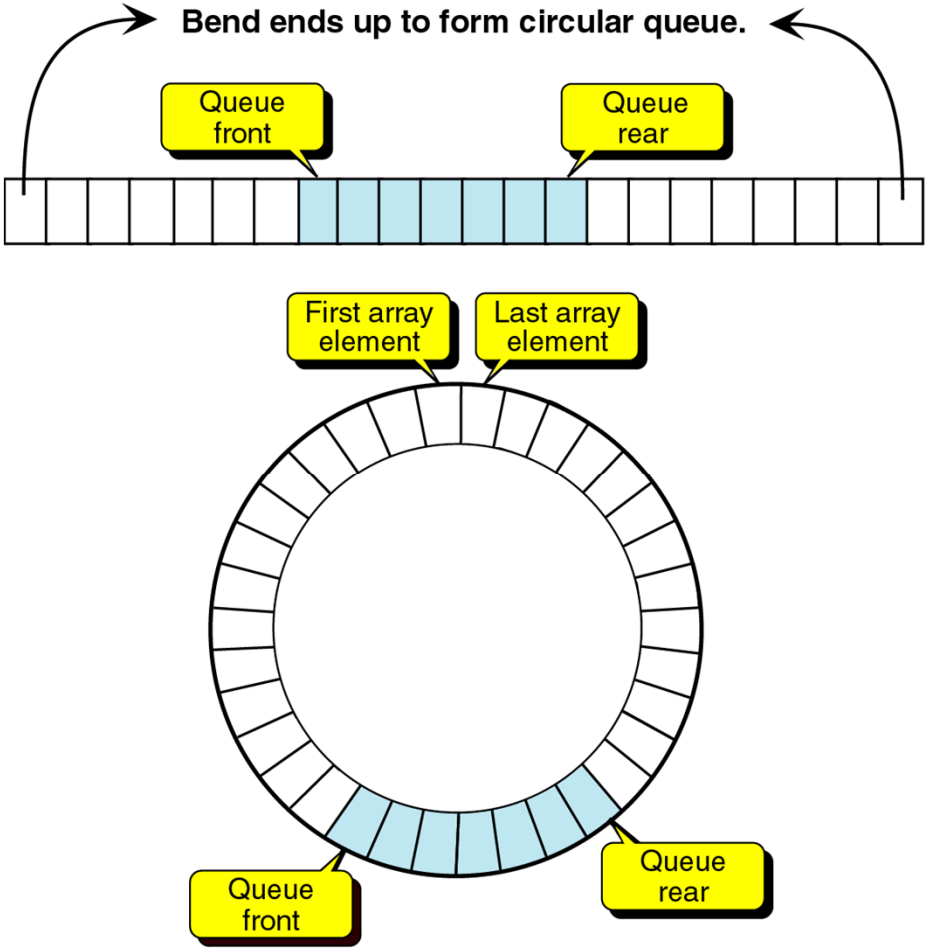
Minh họa hình ảnh các phần tử đang chứa trong Queue

Xây dựng Queue - dùng mảng [4/12]



Khi thêm nhiều phần tử, sẽ làm "tràn" mảng → "Tràn giả"

Xây dựng Queue - dùng mảng [5/12]



Giải pháp cho tình huống "tràn giả": xử lý mảng như là 1 danh sách vòng

Xây dựng Queue - dùng mảng [6/12]

- Khai báo cấu trúc Queue bằng struct

```
// Giả sử Queue chứa các phần tử kiểu nguyên (int)
// Khai báo cấu trúc Queue
typedef struct QUEUE {
    int *QArray;          // mảng chứa các phần tử
    int  QMax;            // số phần tử tối đa
    int  QNumItems;       // số phần tử hiện có
    int  QFront;          // vị trí đầu queue
    int  QRear;           // vị trí cuối queue
};
// Các hàm xử lý thao tác
int  InitQueue(QUEUE &q, int MaxItems);
int  IsEmpty(const QUEUE &q);
int  IsFull(const QUEUE &q);
int  EnQueue(QUEUE &q, int newitem);
int  DeQueue(QUEUE &q, int &itemout);
int  QueueFront(const QUEUE &q, int &itemout);
```

Xây dựng Queue - dùng mảng [7/12]

```
// Khởi tạo queue rỗng, chứa tối đa MaxItems phần tử
int  InitQueue(QUERE &q, int MaxItems)
{
   q.QArray = new int[MaxItems];
   if (q.QArray==NULL)
        return 0;              // Không cấp phát được bộ nhớ
   q.QMax = MaxItems;
   q.QNumItems = 0;       // chưa có phần tử nào trong Queue
   q.QFront = q.QRear = -1;
   return 1;                   // khởi tạo thành công
}
```

Xây dựng Queue - dùng mảng [8/12]

```
// Hàm kiểm tra queue = rỗng ?
int  IsEmpty(const QUEUE &q)
{
   if (q.QNumItems==0)
        return 1;                  // Queue rỗng
   return 0;                       // Queue không rỗng
}

// Hàm kiểm tra queue đầy ?
int  IsFull(const QUEUE &q)
{
   if (q.QNumItems==q.QMax)
        return 1;                  // Queue đầy
   return 0;                       // Queue không đầy
}
```

Xây dựng Queue - dùng mảng [9/12]

```
// Thao tác "EnQueue": thêm 1 phần tử vào cuối Queue
int EnQueue(QUEUE &q, int newitem)
{
   if (IsFull(q))    return 0;   // Queue đầy, không thêm vào được

   q.QRear++;
   if (q.QRear==q.QMax)     // "tràn giả"
        q.QRear = 0;            // Quay trở về đầu mảng

   q.QArray[q.QRear] = newitem;   // thêm phần tử vào Queue
   if (q.QNumItems==0)  q.QFront = 0;
   q.QNumItems++;

   return 1;                              // Thêm thành công
}
```

Xây dựng Queue - dùng mảng [10/12]

```
// Thao tác "DeQueue": lấy ra 1 phần tử ở đầu Queue
int DeQueue(QUEUE &q, int &itemout)
{
   if (IsEmpty(q))     return 0;        // Queue rỗng, kết quả=FALSE

   itemout = q.QArray[q.QFront];   // lấy phần tử đầu ra
   q.QFront++;
   q.QNumItems--;
   if (q.QFront==q.QMax)              // nếu đi hết mảng …
        q.QFront = 0;                 // … quay trở về đầu mảng

   if (q.QNumItems==0)  q.QFront = q.QRear = -1;

   return 1;                          // Thêm thành công
}
```

Xây dựng Queue - dùng mảng [11/12]

```
// Thao tác "QueueFront": kiểm tra phần tử ở đầu Queue
// *** không xoá phần tử
int QueueFront(const QUEUE &q, int &itemout)
{
   if (IsEmpty(q))
        return 0;       // Queue rỗng, kết quả=FALSE

   // lấy phần tử đầu ra
   itemout = q.QArray[q.QFront];

   return 1;
}
```

Xây dựng Queue [12/12]

- Các nội dung tự nghiên cứu:
  - Xây dựng Queue dùng mảng – cài đặt bằng class
  - Xây dựng Queue dùng DSLK đơn – cài đặt bằng struct
  - Xây dựng Queue dùng DSLK đơn – cài đặt bằng class

Ứng dụng của Queue

- Quản lý xếp hàng (theo số thứ tự).
  VD. Tại các ngân hàng, bệnh viện,…

- Quản lý phục vụ in ấn (máy in)

Nội dung

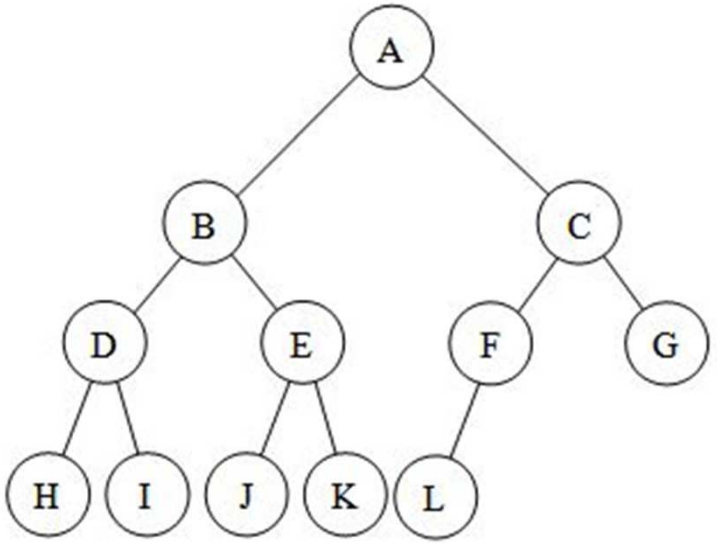
1. Các cấu trúc dữ liệu cơ bản
2. Cấu trúc cây – Tree Structure
3. Cây nhị phân tìm kiếm – Binary Search Tree
4. Các dạng cây nhị phân tìm kiếm cân bằng
5. Bảng băm – Hash Table

Cấu trúc cây – Tree Structure

- Các khái niệm và thuật ngữ cơ bản
- Tổng quan về cây nhị phân (Binary Tree)

Các khái niệm và thuật ngữ cơ bản

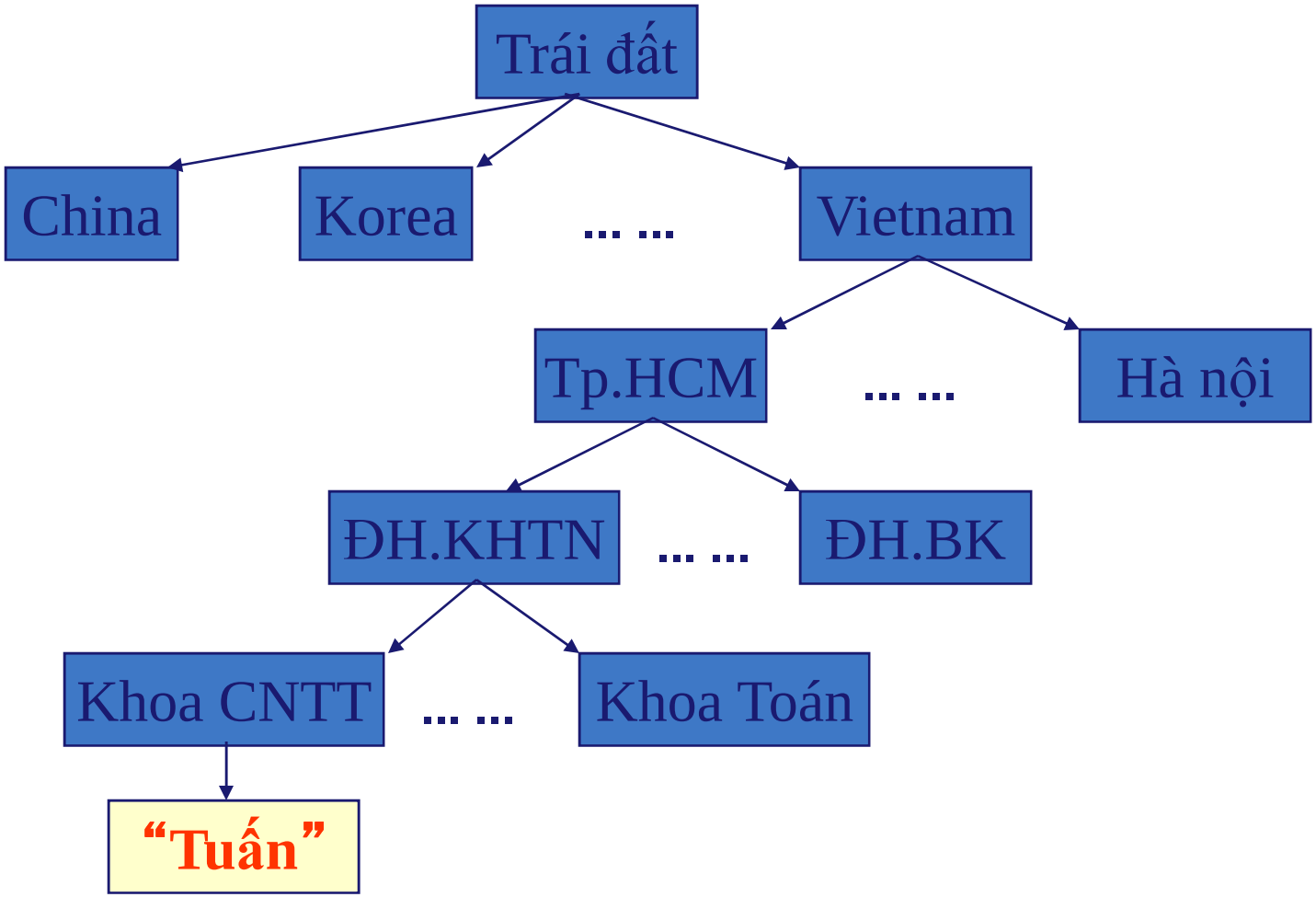
- Các ví dụ
- Định nghĩa cấu trúc cây
- Các thuật ngữ liên quan



Các ví dụ [1/5]

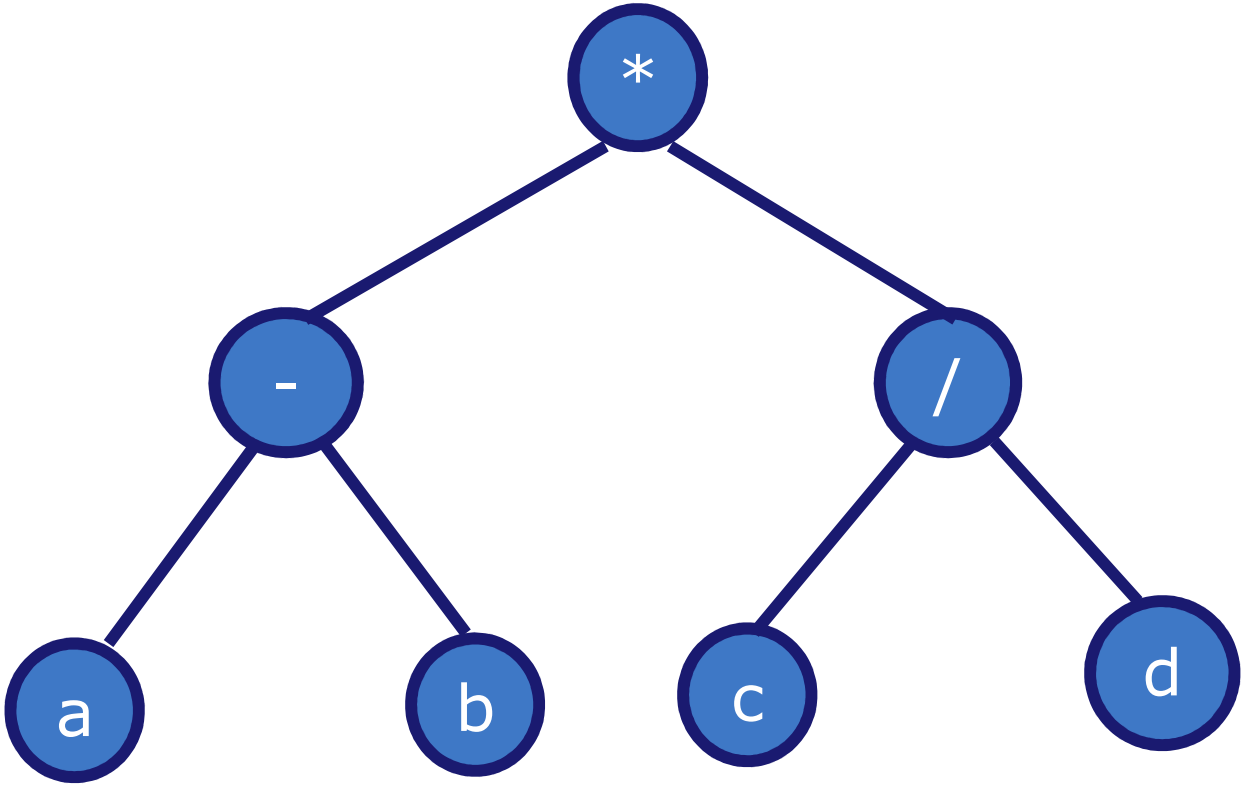
- Ví dụ 1: cách lưu trữ phân cấp → bài toán đưa thư
  - Trên thế giới hiện có > 6 tỉ người
  - Tuấn, khoa CNTT, ĐH KHTN, Tp.HCM, Việt nam
  - Cách tìm ra "Tuấn" nhanh nhất ?
  - Sử dụng mảng (array) ?
  - Sử dụng danh sách liên kết (linked list) ?

Các ví dụ [2/5]


Trái đất
China
Korea
... ...
Vietnam
Tp.HCM
... ...
Hà nội
ĐH.KHTN
... ...
ĐH.BK
Khoa CNTT
... ...
Khoa Toán
"Tuấn"

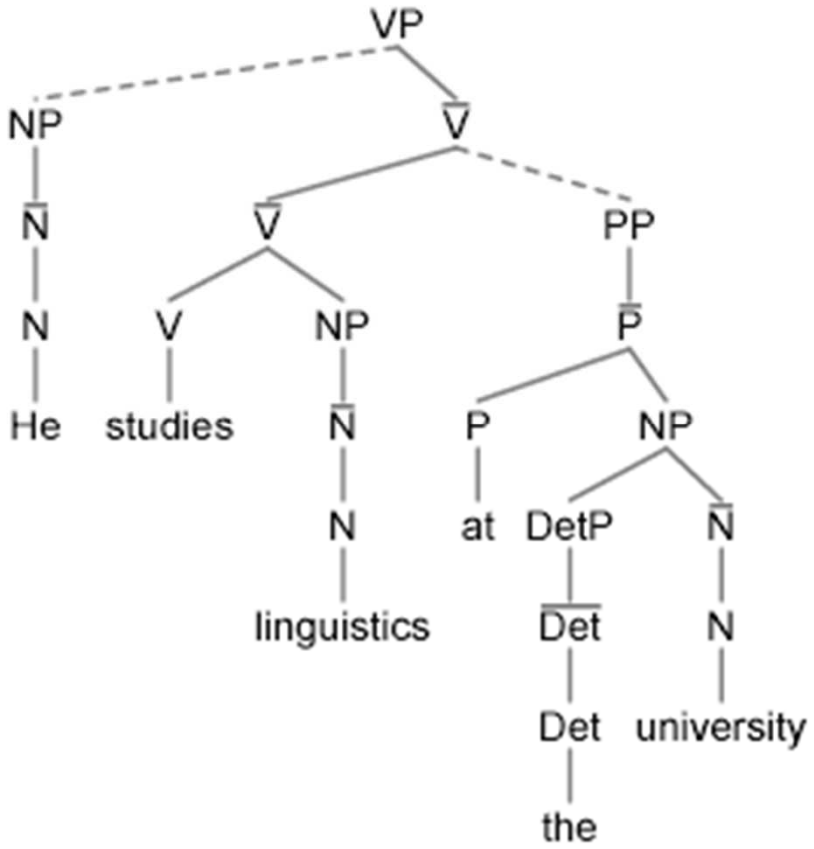
Các ví dụ [3/5]

- Ví dụ 2: cây biểu thức (a-b)*(c/d)



Các ví dụ [4/5]

- Ví dụ 3: cây ngữ pháp

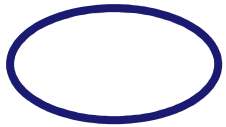


Các ví dụ [5/5]

- Cây là 1 cấu trúc dữ liệu quan trọng để biểu diễn tính "kế thừa", "phân cấp"
- Các cây mô tả tính kế thừa:
  - Cây gia phả (trong các dòng họ)
  - Cây phân cấp các loài (trong sinh vật)
  - …

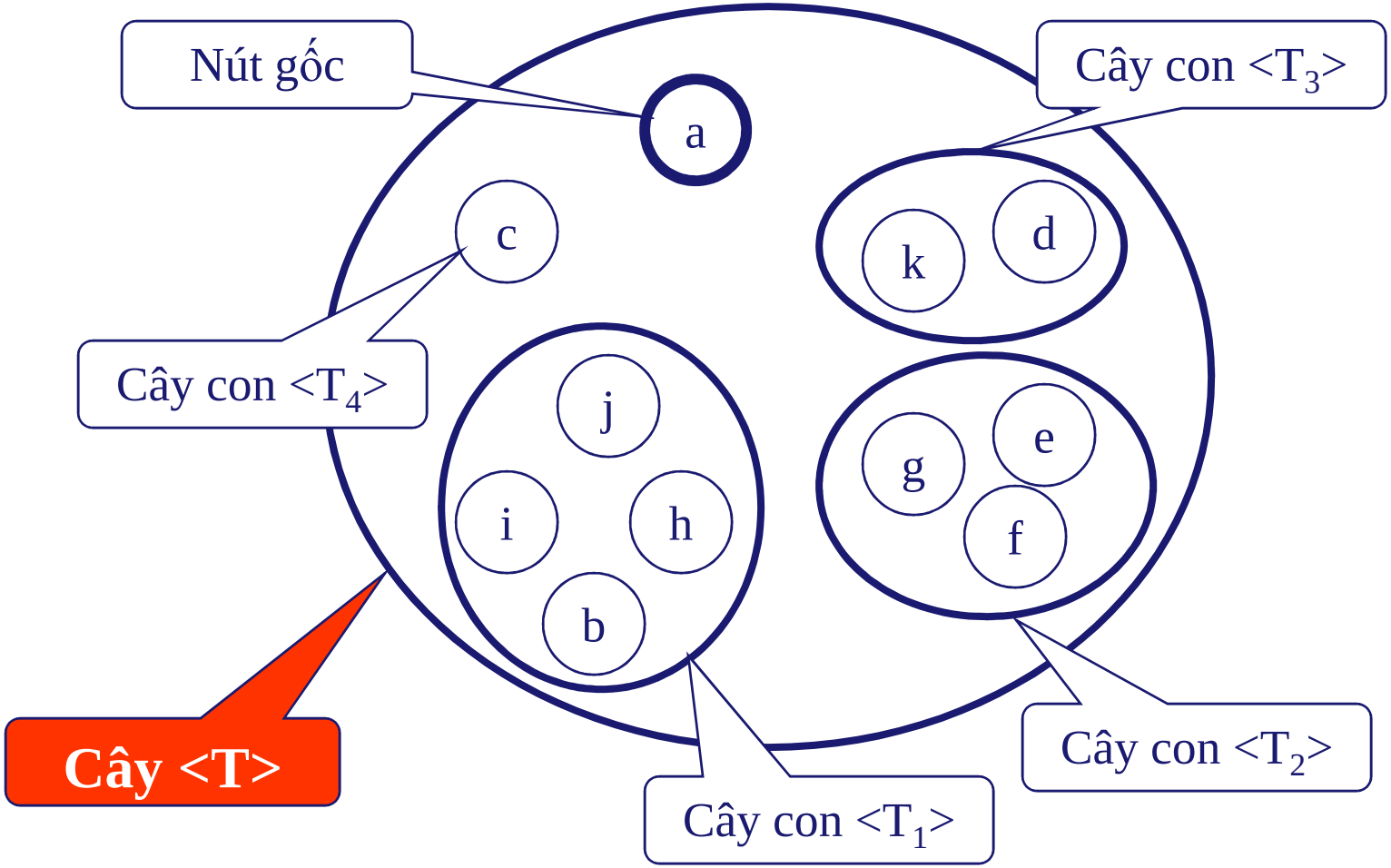
Định nghĩa cấu trúc cây [1/5]

- Một cây <T> (Tree) là:
  - Một tập các phần tử, gọi là các nút (Node) $p_1,p_2,\ldots,p_N$
  - Nếu N=0, cây <T> gọi là cây rỗng (NULL)
  - Nếu N>0:
    - Tồn tại duy nhất 1 nút p_r gọi là gốc của cây
    - Các nút còn lại được chia thành m tập không giao nhau: T_1, T_2, …, T_m
    - Mỗi <T_i> là 1 cây con của cây <T>

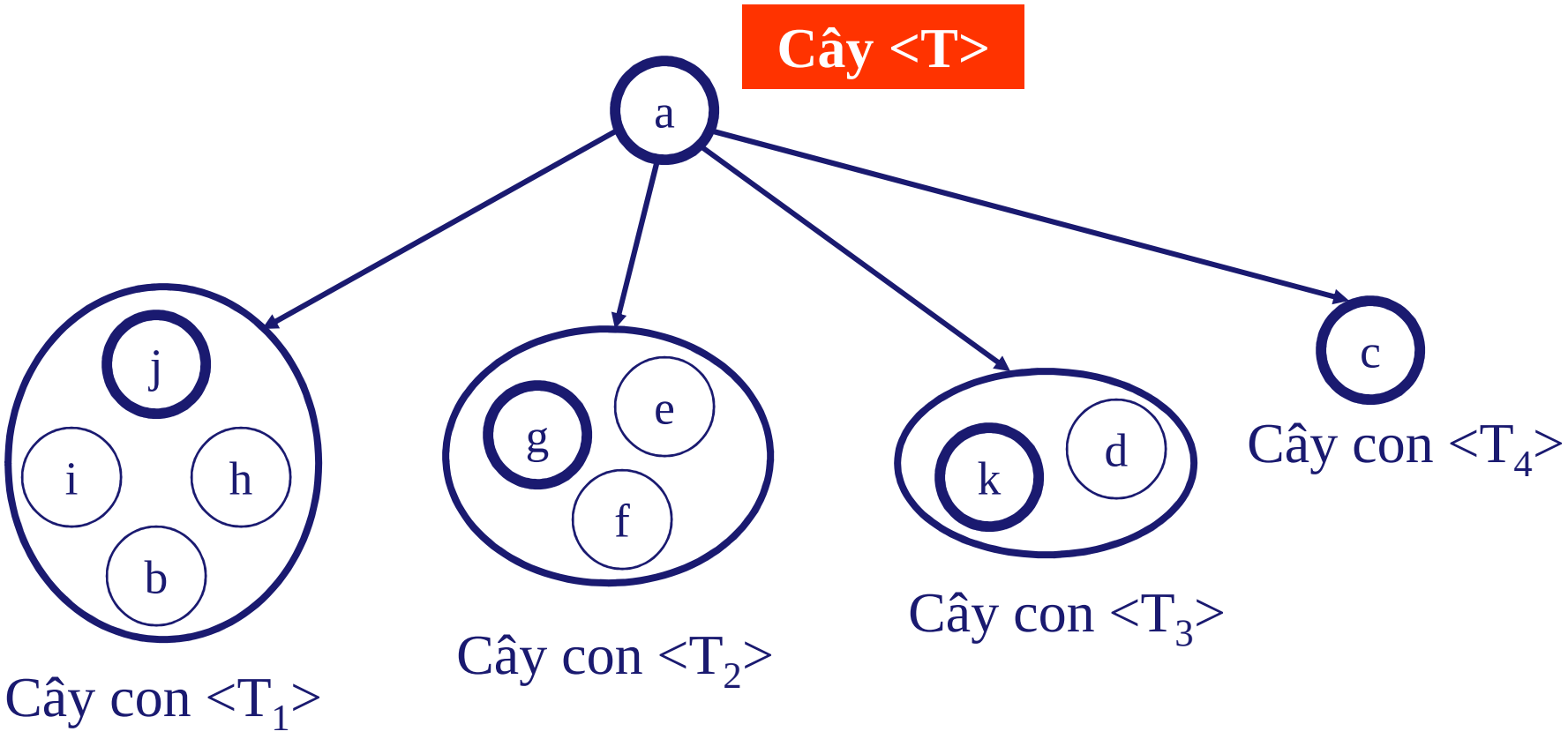


Tập rỗng → Cây <T> rỗng (NULL)

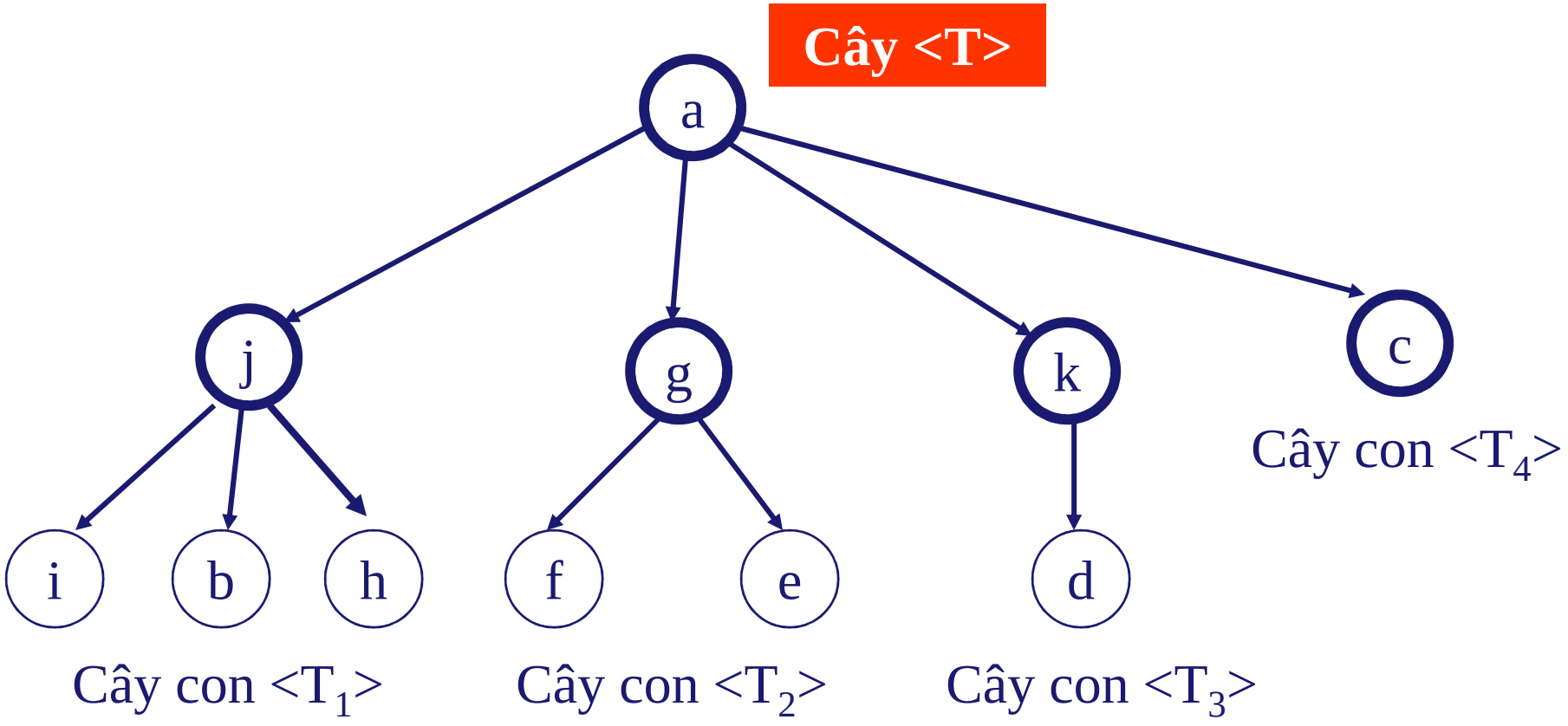
Định nghĩa cấu trúc cây [2/5]



Định nghĩa cấu trúc cây [3/5]



Định nghĩa cấu trúc cây [4/5]



Định nghĩa cấu trúc cây [5/5]

- Các tính chất của cây:
  - Nút gốc không có nút cha
  - Mỗi nút khác chỉ có 1 nút cha
  - Mỗi nút có thể có nhiều nút con
  - Không có chu trình

Các thuật ngữ liên quan [1/13]

- Nút (Node): là 1 phần tử trong cây. Mỗi nút có thể chứa 1 dữ liệu bất kỳ
- Nhánh (Branch): là đoạn nối giữa 2 nút
- Nút cha (Parent node)
- Nút con (Child node)
- Nút anh em (sibling nodes): là những nút có cùng nút cha
- Bậc của 1 nút p_i: là số nút con của p_i
  - Bậc (a) = 4; Bậc (j) = 3; Bậc (g) = 2;
  - Bậc (k) = 1; Bậc (c) = 0

Các thuật ngữ liên quan [2/13]

- Nút gốc (Root node): nút không có nút cha
- Nút lá (Leaf node): nút có bậc = 0 (không có nút con)
- Nút nội (Internal node): là nút có nút cha và có nút con
- Cây con (Subtree)
  - *Trắc nghiệm*: có bao nhiêu cây con trong cây <T> ?

Các thuật ngữ liên quan [3/13]

- Bậc của cây: là bậc lớn nhất của các nút trong cây
  - Bậc (<T>) = max {bậc (p_i) / $p_i \in$ <T>}
  - Bậc của cây <T> ?

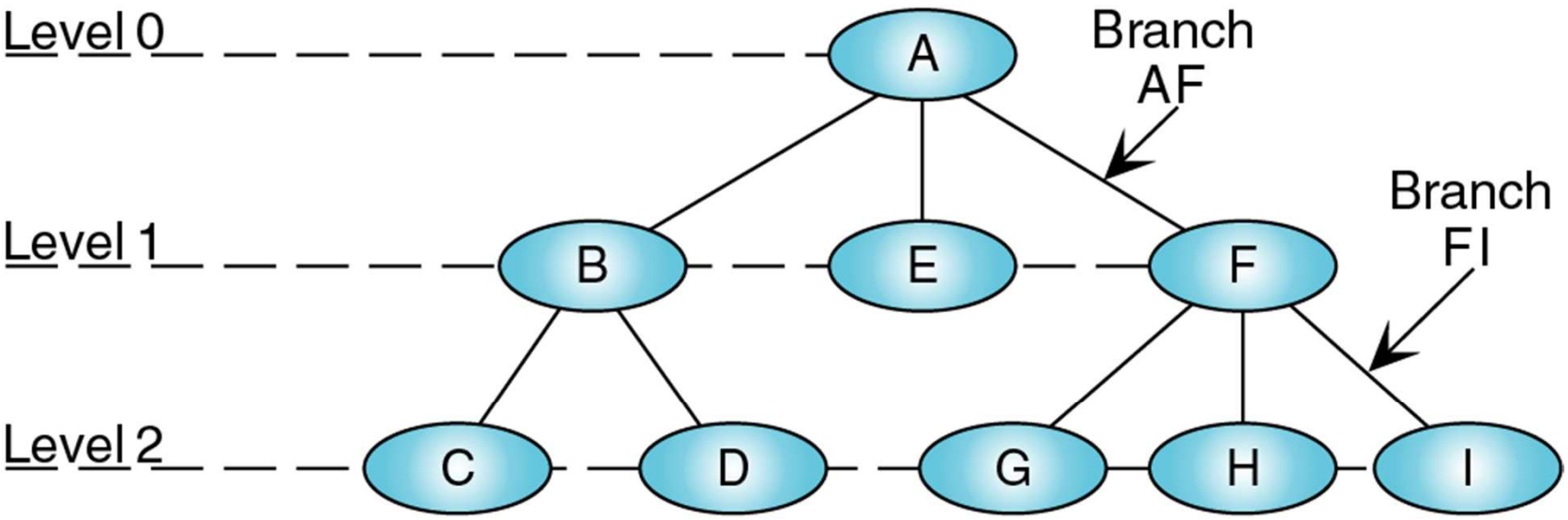
- Đường đi (Path) giữa nút p_i đến nút p_j: là dãy các nút liên tiếp từ p_i đến p_j sao cho giữa hai nút kề nhau đều có nhánh
  - Path(a, d) ?

Các thuật ngữ liên quan [4/13]

- Mức (Level):
  - Mức (p) = 0 nếu p = root
  - Mức (p) = 1 + Mức (Cha (p)) nếu p!=root

- Chiều cao của cây (Height - h_T): đường đi dài nhất từ nút gốc đến nút lá
  - h_T = max {Path(root, p_i) / p_i là nút lá $\in$ <T>}
  - h_T ?

Các thuật ngữ liên quan [5/13]

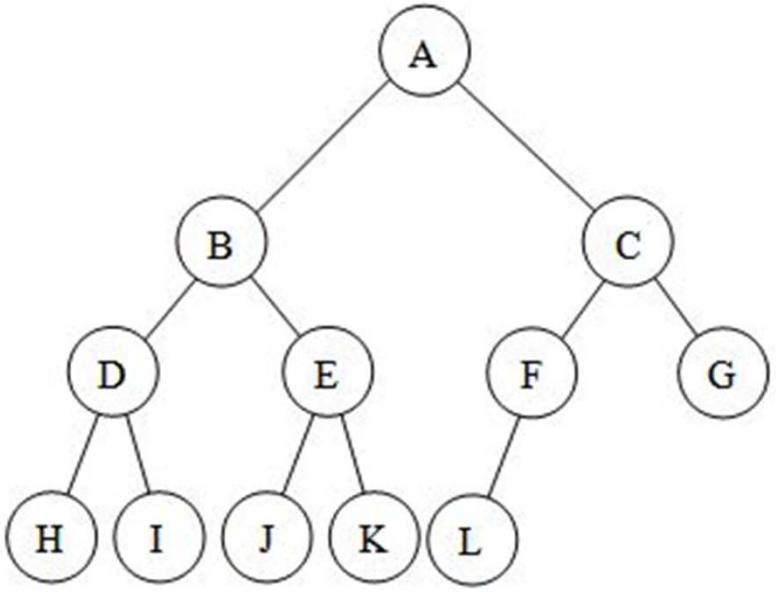


Parents: A, B, F
Children: B, E, F, C, D, G, H, I
Siblings: {B,E,F}, {C,D}, {G,H,I}

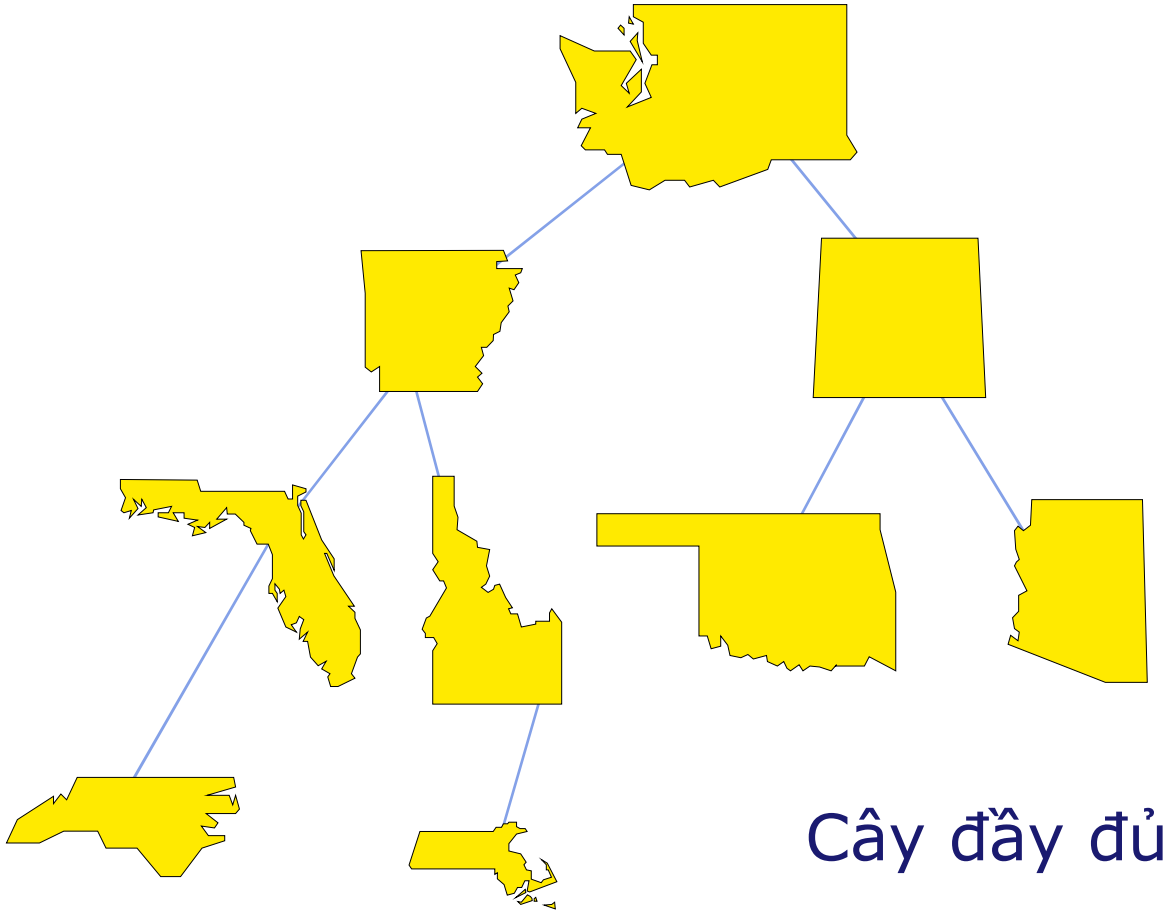
Leaves C,D,E,G,H,I
Internal nodes B,F

Các thuật ngữ liên quan [6/13]

- Cây đầy đủ (Complete tree) với *h* mức: là 1 cây thoả các điều kiện
  - Những mức 0 đến mức *h-2* đều có đủ số node
  - Ở mức *h-1*, các node được thêm vào cây từ trái sang phải

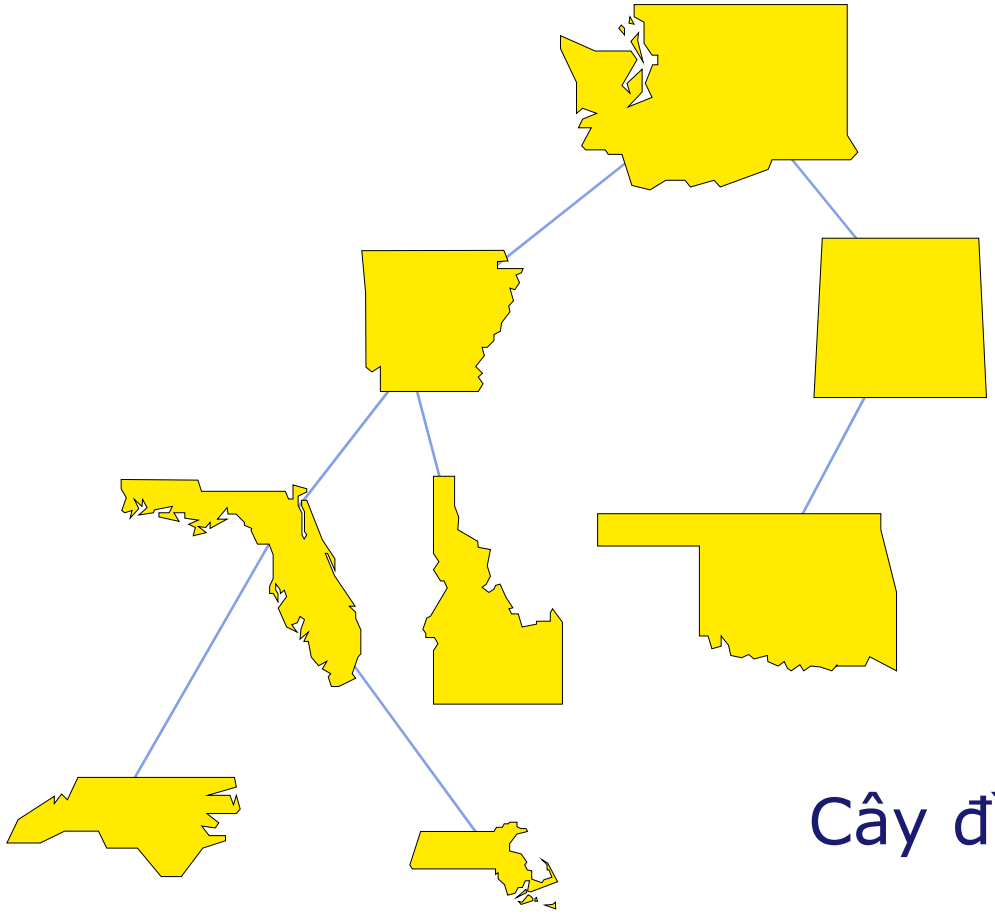


Các thuật ngữ liên quan [7/13]



Cây đầy đủ ?

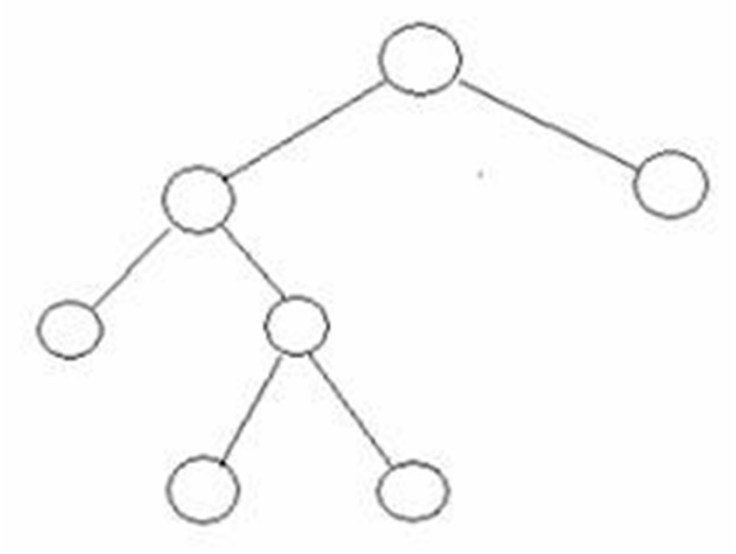
Các thuật ngữ liên quan [8/13]



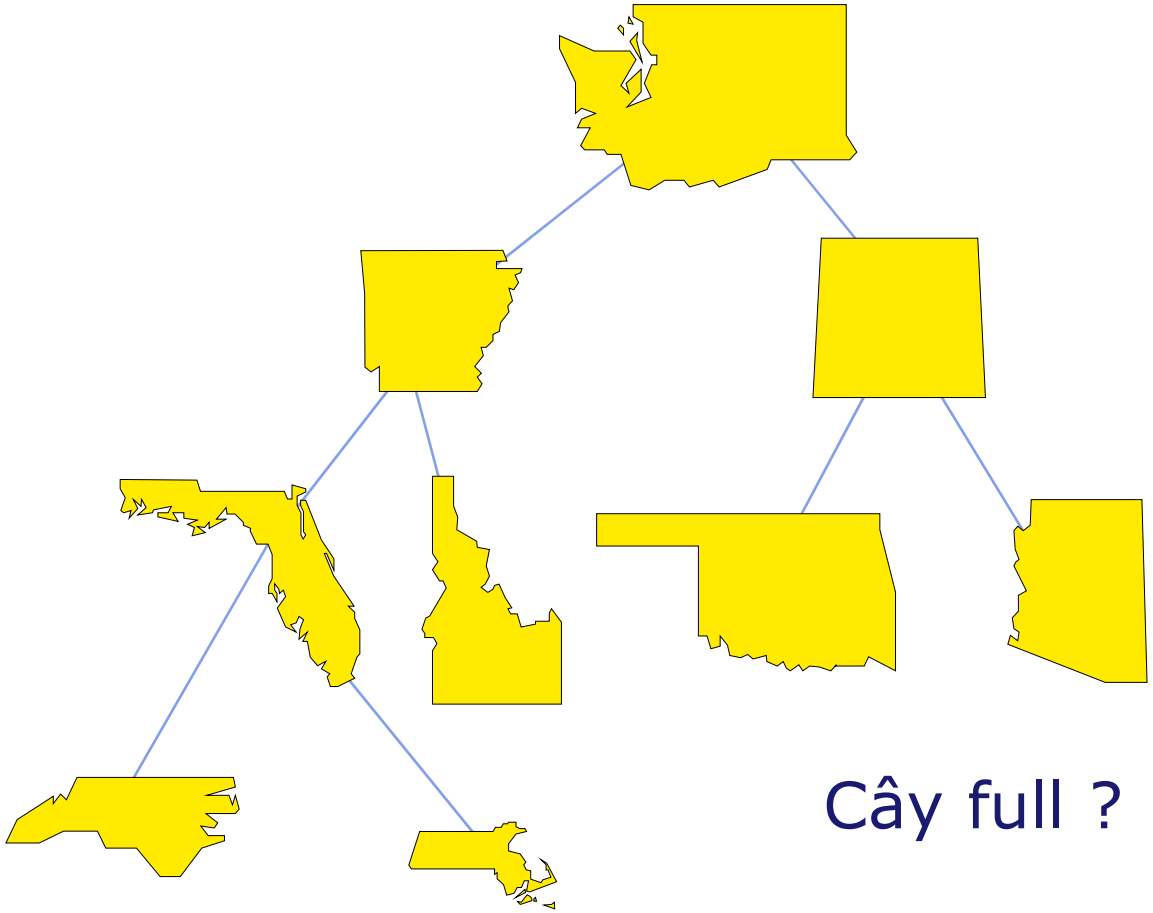
Cây đầy đủ ?

Các thuật ngữ liên quan [9/13]

- Cây full (Full tree): cây full nhị phân là 1 cây thoả
  - Mỗi node có 0 hoặc 2 node con

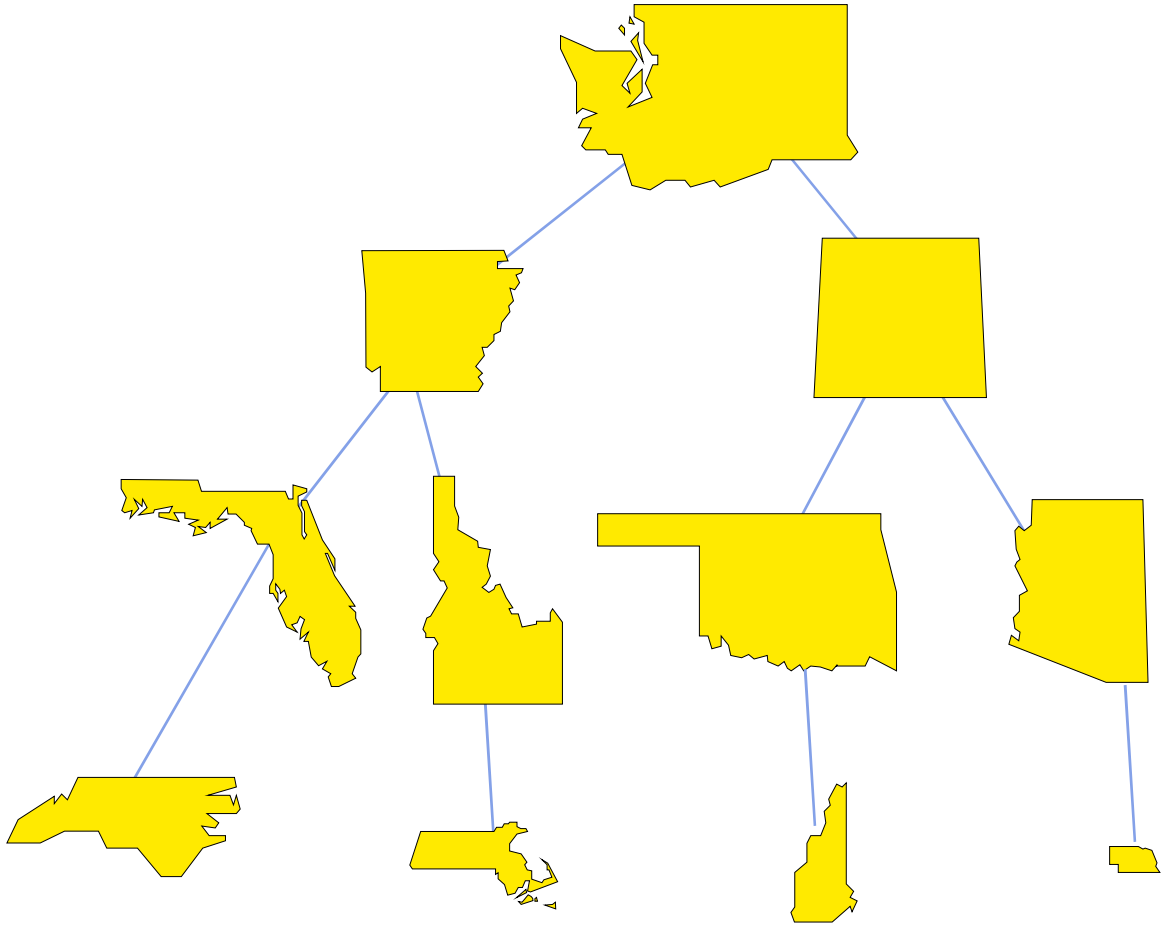


Các thuật ngữ liên quan [10/13]



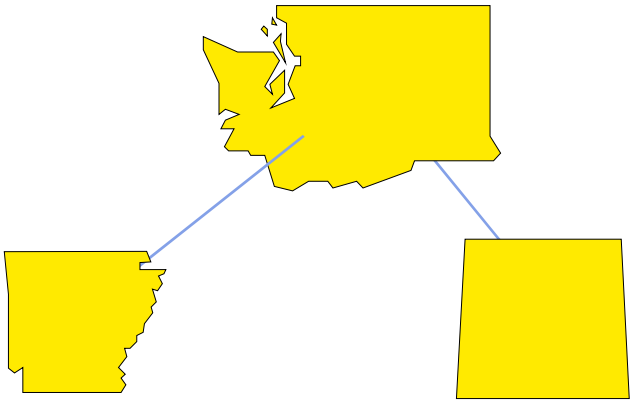
Cây full ?

Các thuật ngữ liên quan [11/13]



Cây full ?

Các thuật ngữ liên quan [12/13]



Cây full ?

Các thuật ngữ liên quan [13/13]

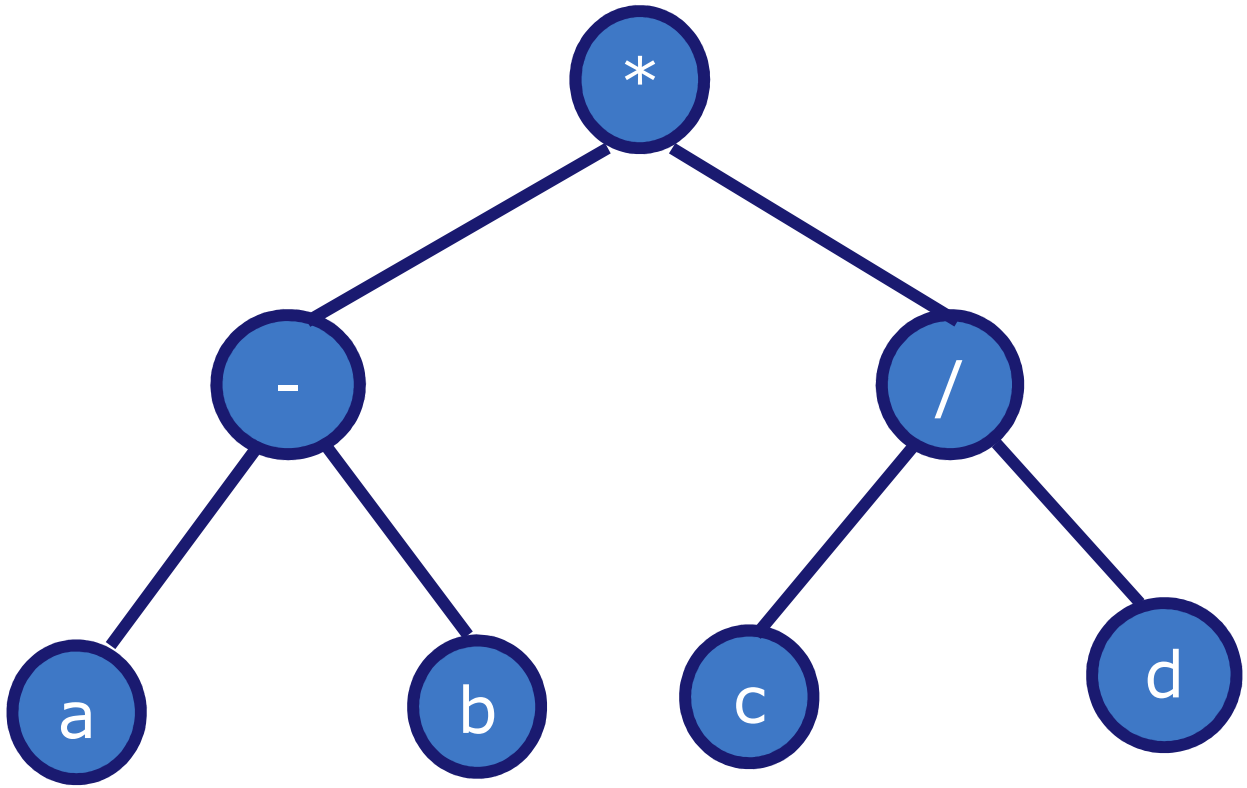
- Mức *h* của cây bậc *d* có tối đa d^h nút
  - VD. mức h=2 của cây bậc 3 có bao nhiêu nút ?
- *h* mức đầu tiên của cây bậc *d* có số nút là:
  - $1 + d + d^2 + d^3 + \ldots + d^{h-1} = (d^h - 1)/(d - 1)$
  - 3 mức đầu tiên của cây bậc 3 có bao nhiêu nút ?

Tổng quan về cây nhị phân (Binary Tree)

- Định nghĩa
- Cách thức lưu trữ cây
- Các phương pháp duyệt cây

Định nghĩa [1/3]

- Cây nhị phân là cây có bậc = 2



Định nghĩa [2/3]

- Độ cao của cây nhị phân có N nút:
  - $h_{T(max)} = N$
  - $h_{T(min)} = [\log_2 N] + 1$

Định nghĩa [3/3]

- Trắc nghiệm:
  - Hãy vẽ tất cả các dạng cây nhị phân có 3 nút ?
  - Hãy vẽ tất cả các dạng cây nhị phân có 4 nút ?
  - Có thể xác định số dạng cây nhị phân có N nút ?

Cách thức lưu trữ cây

- Có 2 cách tổ chức cây nhị phân:
  - Lưu trữ bằng mảng
  - Lưu trữ bằng con trỏ cấu trúc cấp phát động

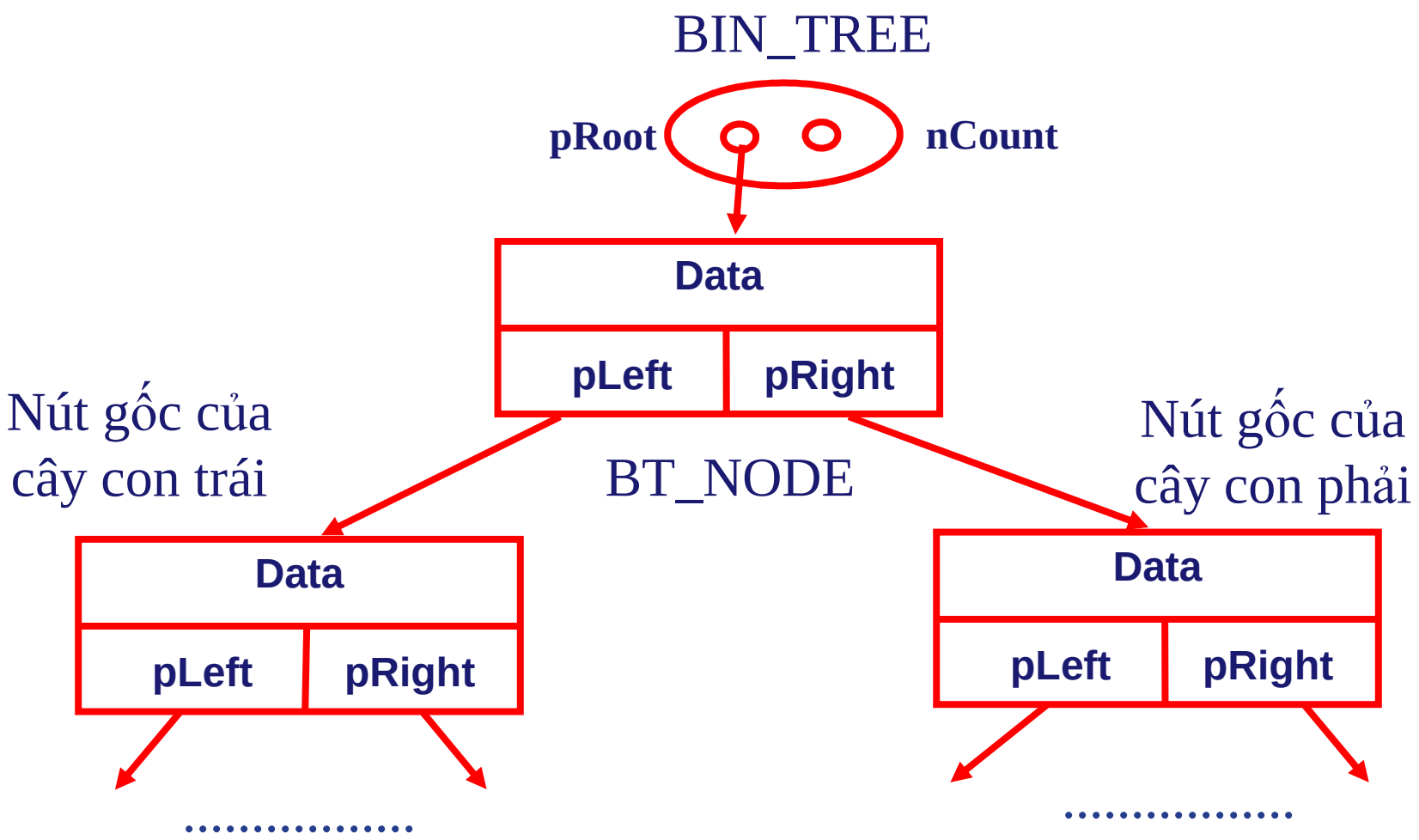
Cách lưu trữ cây – dùng mảng [1/2]

# index	Nút	Con trái	Con phải
0	*	1	2
1	-	3	4
2	/	5	6
3	a	-1	-1
4	b	-1	-1
5	c	-1	-1
6	d	-1	-1

Cách lưu trữ cây – dùng mảng [2/2]

```
// Định nghĩa các cấu trúc dữ liệu
typedef struct tagBT_NODE {
    int         Data;
    int         Left;       // chỉ số (index) nút con trái
    int         Right;      // chỉ số (index) nút con phải
} BT_NODE;                  // binary tree node

BT_NODE   tree[N];          // cây nhị phân có N nút
```

Cách lưu trữ cây – dùng con trỏ [1/2]

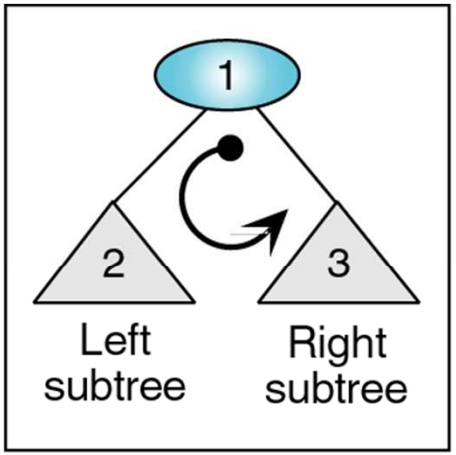


Cách lưu trữ cây – dùng con trỏ [1/2]

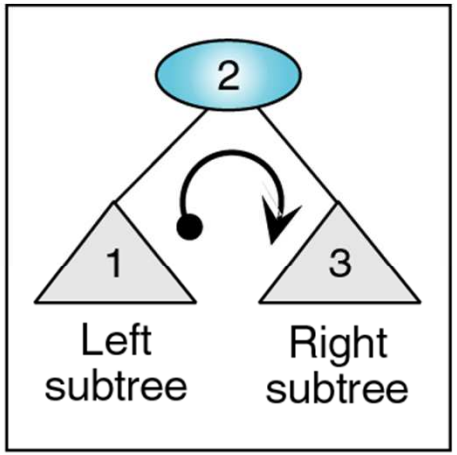
```
// Định nghĩa các cấu trúc dữ liệu
// … cấu trúc node
typedef struct tagBT_NODE {
    int                 Data;
    tagBT_NODE          *pLeft;     // con trỏ đến nút con trái
    tagBT_NODE          *pRight;    // con trỏ đến nút con phải
} BT_NODE;                          // binary tree node

// … cấu trúc tree
typedef struct BIN_TREE {
    int                 nCount;     // số nút trong cây
    BT_NODE             *pRoot;     // con trỏ đến nút gốc
};                                  // binary tree
```

Các phương pháp duyệt cây [1/7]

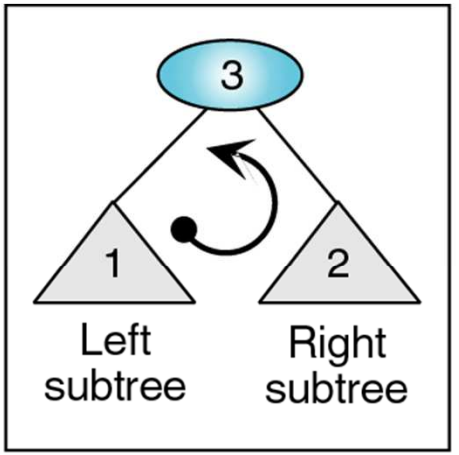
- Có 3 cách duyệt cây:
  - Duyệt gốc trước (Pre-Order) NLR
  - Duyệt gốc giữa (In-Order) LNR
  - Duyệt gốc sau (Post-Order) LRN



(a) Preorder traversal



(b) Inorder traversal

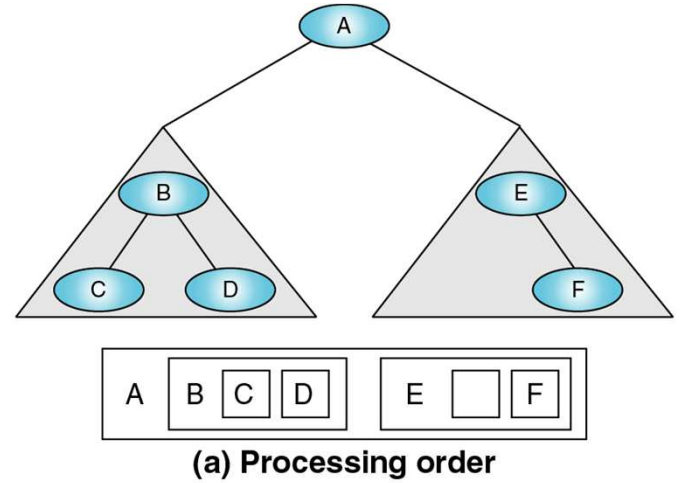


(c) Postorder traversal

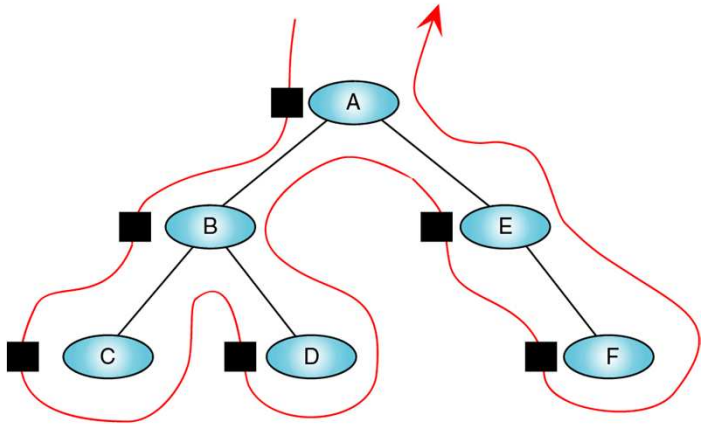
Các phương pháp duyệt cây [2/7]

- Duyệt gốc trước (Pre-Order) NLR

```
void NLR(const BT_NODE
                *pRoot)
{
   if (pRoot==NULL) return;
   "Xử lý nút gốc pRoot"
   NLR(pRoot->pLeft);
   NLR(pRoot->pRight);
}
```

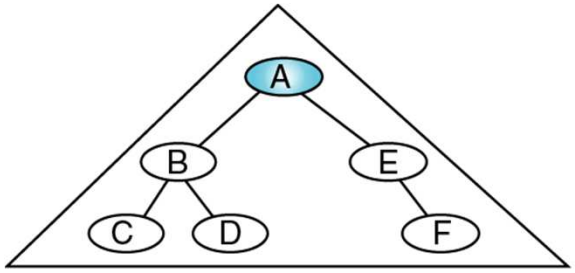


(a) Processing order

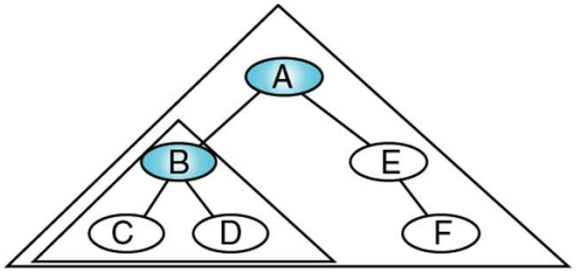


(b) "Walking" order

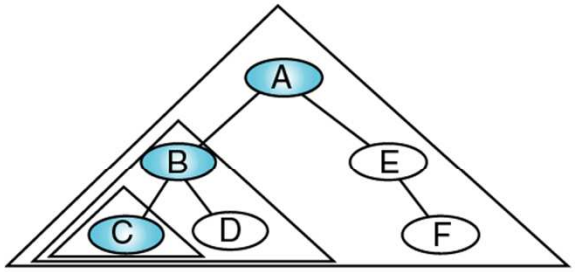
Các phương pháp duyệt cây [3/7]



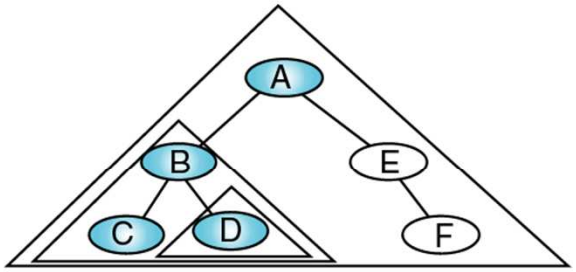
(a) Process tree A



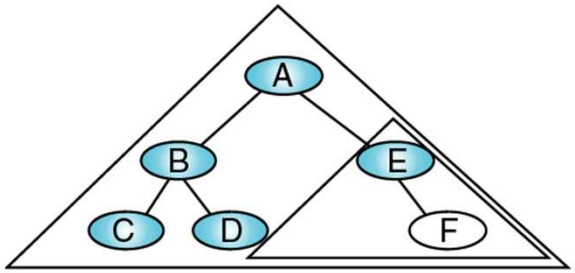
(b) Process tree B



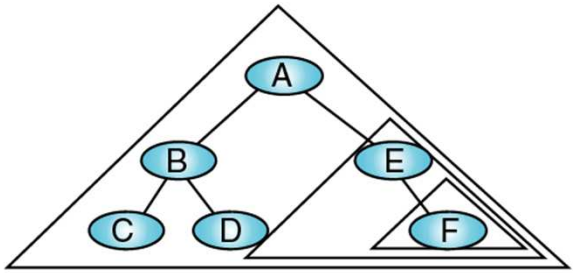
(c) Process tree C



(d) Process tree D



(e) Process tree E



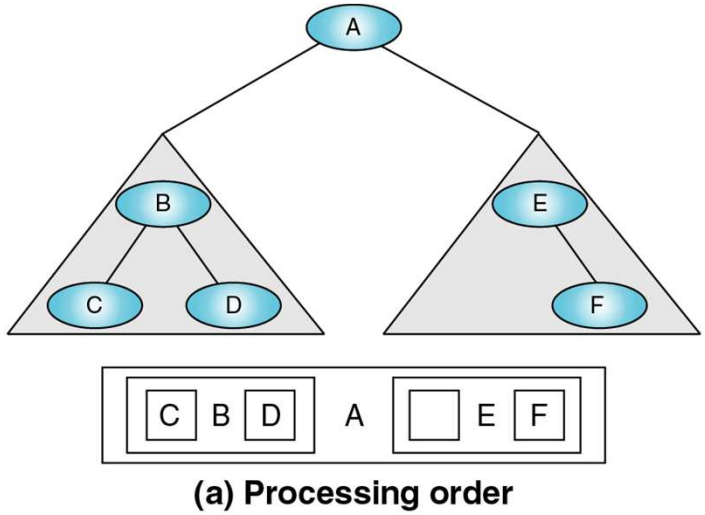
(f) Process tree F

Minh họa cách duyệt "gốc trước"

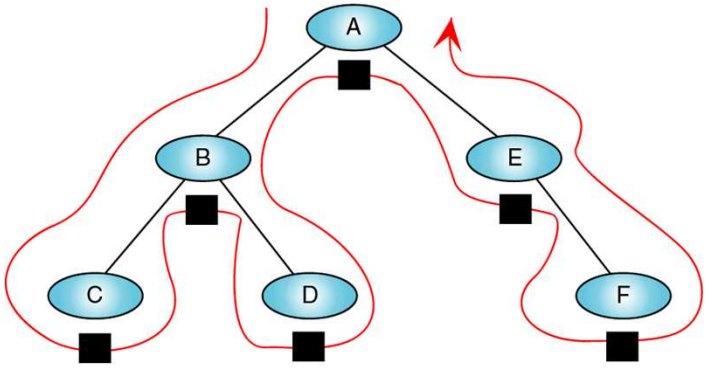
Các phương pháp duyệt cây [4/7]

- Duyệt gốc giữa (In-Order) LNR

```
void LNR(const BT_NODE
                *pRoot)
{
   if (pRoot==NULL) return;
   LNR(pRoot->pLeft);
   "Xử lý nút gốc pRoot"
   LNR(pRoot->pRight);
}
```



(a) Processing order

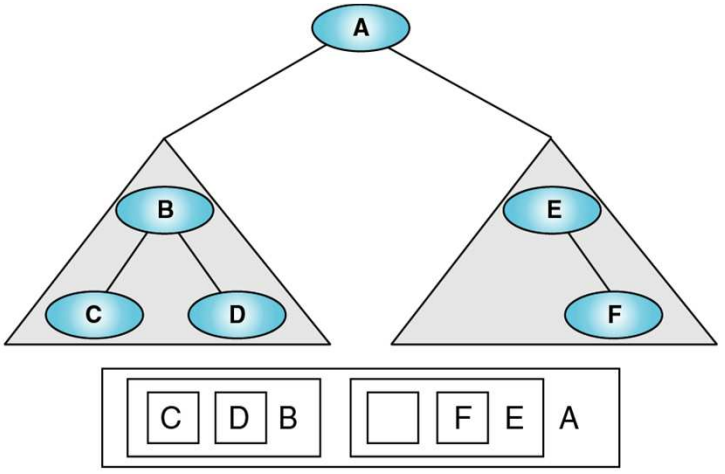


(b) "Walking" order

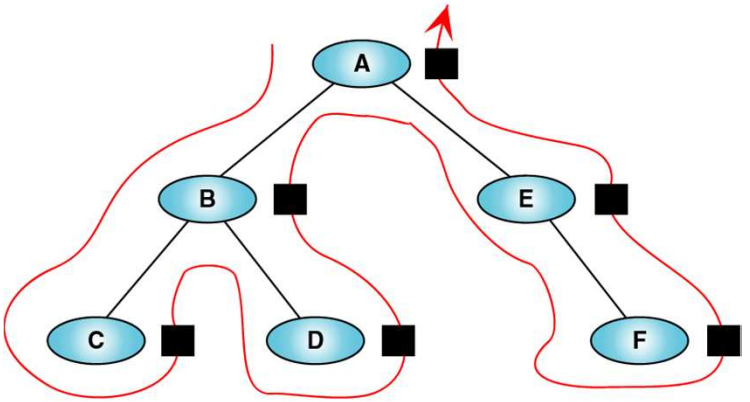
Các phương pháp duyệt cây [5/7]

- Duyệt gốc sau (Post-Order) LRN

```
void LRN(const BT_NODE
                *pRoot)
{
   if (pRoot==NULL) return;
   LRN(pRoot->pLeft);
   LRN(pRoot->pRight);
   "Xử lý nút gốc pRoot"
}
```



(a) Processing order



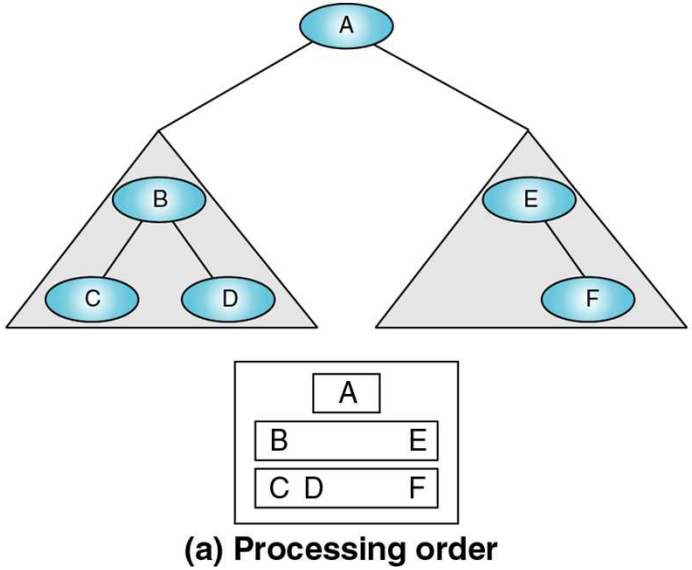
(b) "Walking" order

Các phương pháp duyệt cây [6/7]

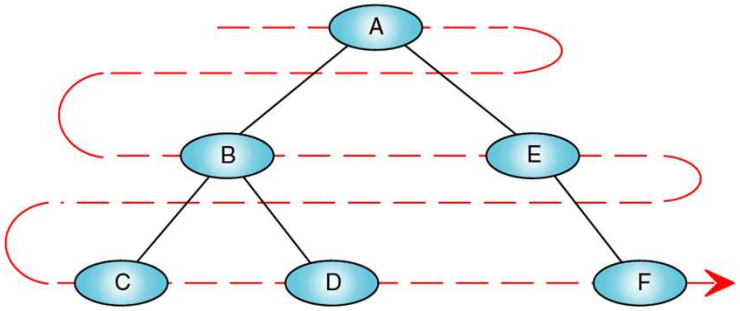
- Trắc nghiệm:
  - Cho biết kết quả duyệt cây biểu thức ở slide #88 theo mỗi cách NLR, LNR, LRN ?
  - Viết thủ tục/hàm đếm số nút trong cây ?
  - Viết thủ tục/hàm đếm số nút lá trong cây ?
  - Viết thủ tục/hàm tính chiều cao của cây ?
  - Viết thủ tục/hàm duyệt cây theo cách NLR không dùng đệ qui

Các phương pháp duyệt cây [7/7]

- Trắc nghiệm:
  - Viết giải thuật duyệt cây theo mức ?



(a) Processing order



(b) "Walking" order

Nội dung

Binary Search Tree (BST)

- Ý nghĩa của cây BST
- Định nghĩa
- Ví dụ
- Mô tả cấu trúc dữ liệu
- Xây dựng các thao tác cơ bản trên cây
- Các đánh giá
- Trắc nghiệm

Ý nghĩa của cây BST

- Điểm yếu và điểm mạnh của việc sử dụng mảng ?
- Điểm yếu và điểm mạnh của việc sử dụng danh sách liên kết ?
- Có 1 cấu trúc (mà) tổng hợp được các điểm mạnh của mảng và danh sách liên kết ?
- Trong cây nhị phân, chi phí để tìm kiếm 1 phần tử là ?

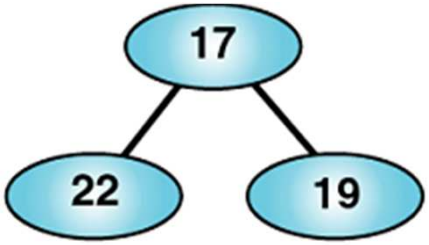
Định nghĩa BST

- Cây nhị phân tìm kiếm là:
  - Một cây nhị phân
  - Mỗi nút p của cây đều thỏa:
    - Tất cả các nút thuộc cây con trái (p->pLeft) đều có giá trị nhỏ hơn giá trị của p

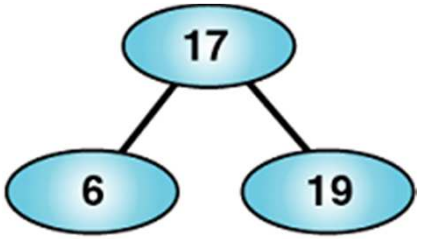
      $\forall q \in$ `p->pLeft: q->Data < p->Data`
    - Tất cả các nút thuộc cây con phải (p->pRight) đều có giá trị lớn hơn giá trị của p

      $\forall q \in$ `p->pRight: q->Data > p->Data`

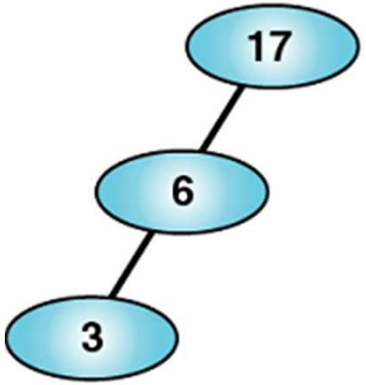
Ví dụ [1/2]



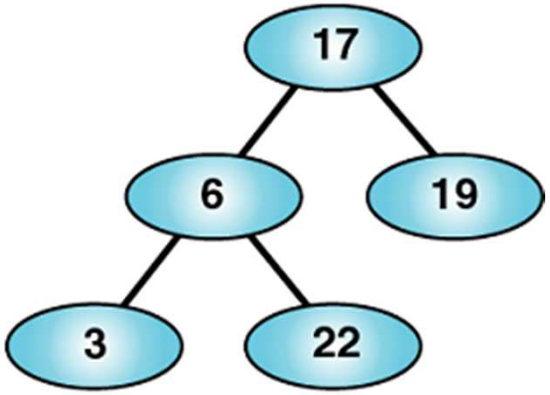
(a)



(b)

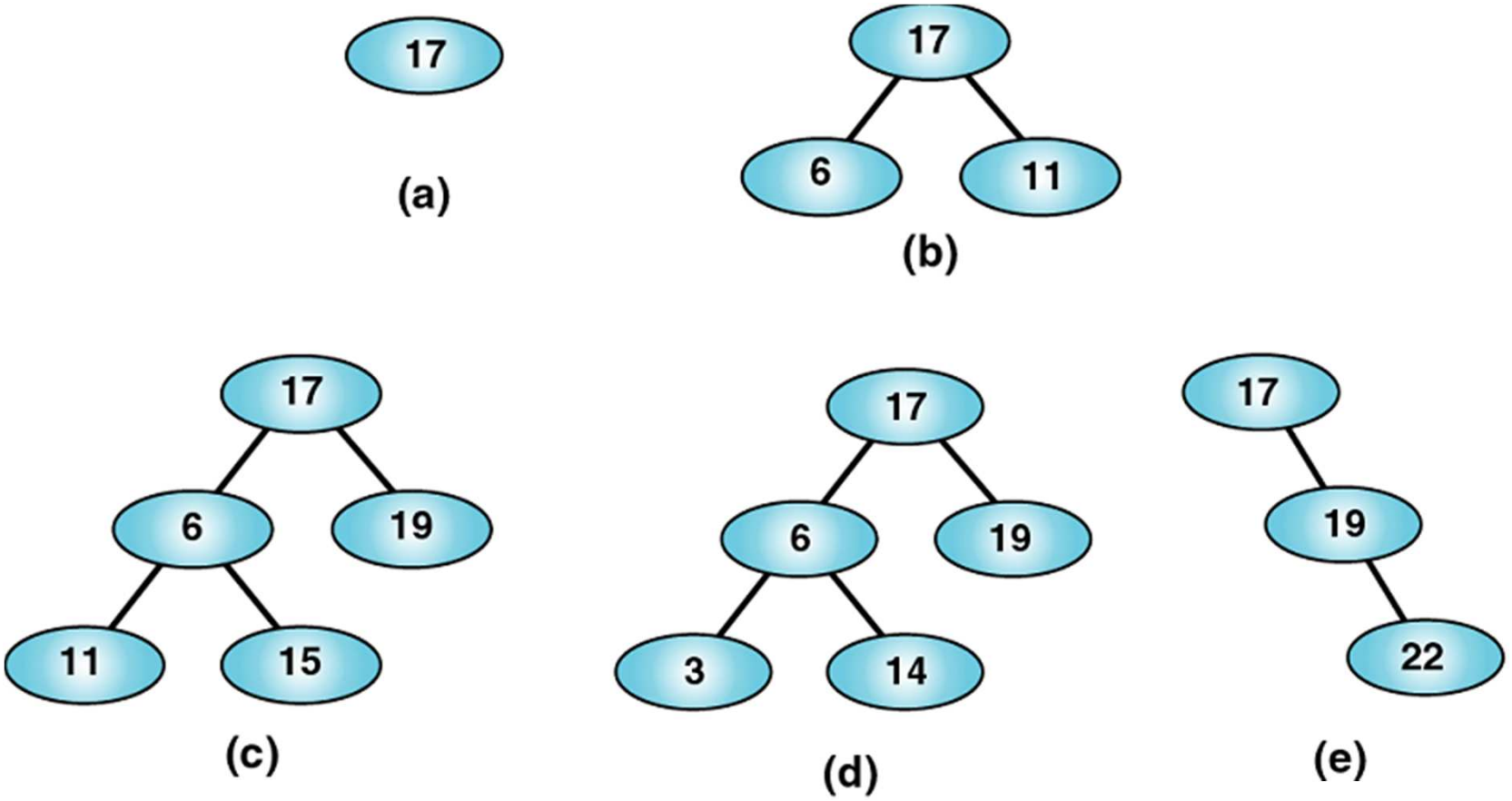


(c)



(d)

Ví dụ [2/2]



Mô tả cấu trúc dữ liệu

- Cách lưu trữ cây BST: giống như cây nhị phân
- Xem lại phần *"Tổng quan về cây nhị phân - Cách thức lưu trữ cây" (slide 91)*

Xây dựng các thao tác cơ bản trên cây [1/24]

- Các thao tác trên cây BST:
  - Tạo lập cây rỗng
  - Kiểm tra cây rỗng
  - Tìm kiếm 1 phần tử
  - Thêm 1 phần tử
  - Xóa 1 phần tử

Xây dựng các thao tác cơ bản trên cây [2/24]

- Tạo lập cây rỗng:

```
void  BSTCreate(BIN_TREE &t)
{
    t.nCount = 0;     // Số nút trong cây
    t.pRoot = NULL;  // Con trỏ đến nút gốc
}
```

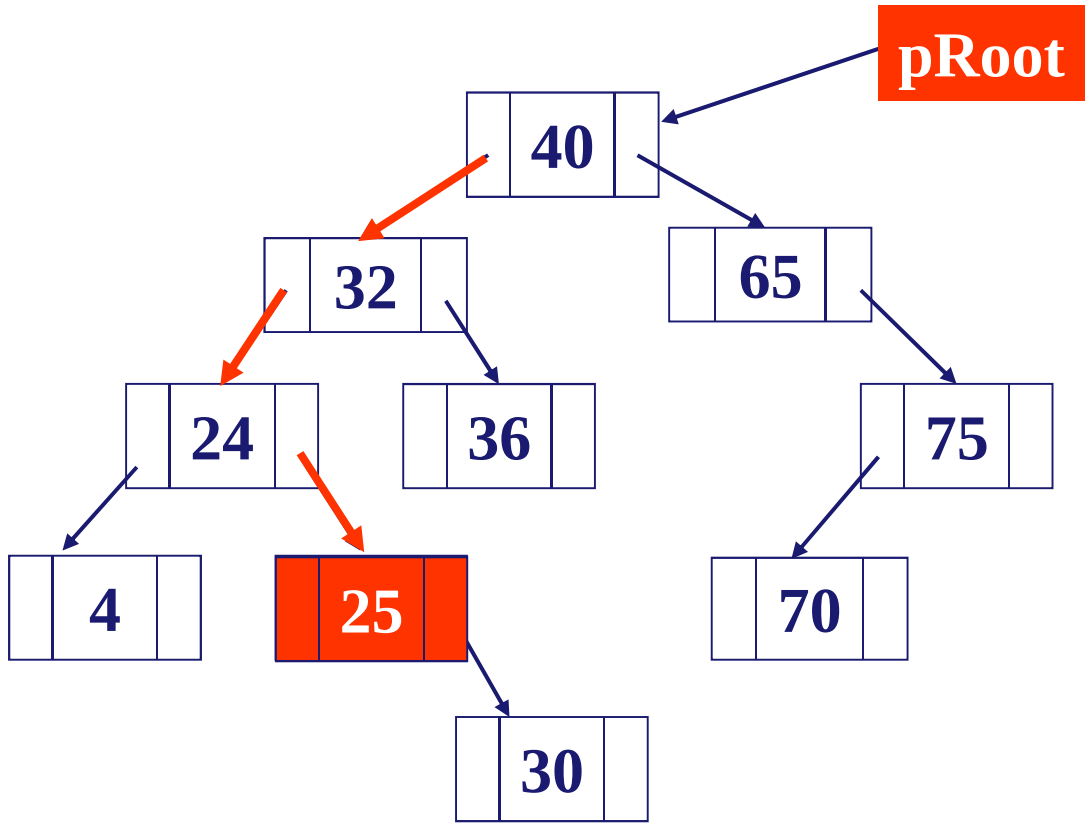
Xây dựng các thao tác cơ bản trên cây [3/24]

- Kiểm tra cây rỗng:

```
int  BSTIsEmpty(const BIN_TREE &t)
{
    if (t.pRoot==NULL)  return 1;
    return 0;
}
```

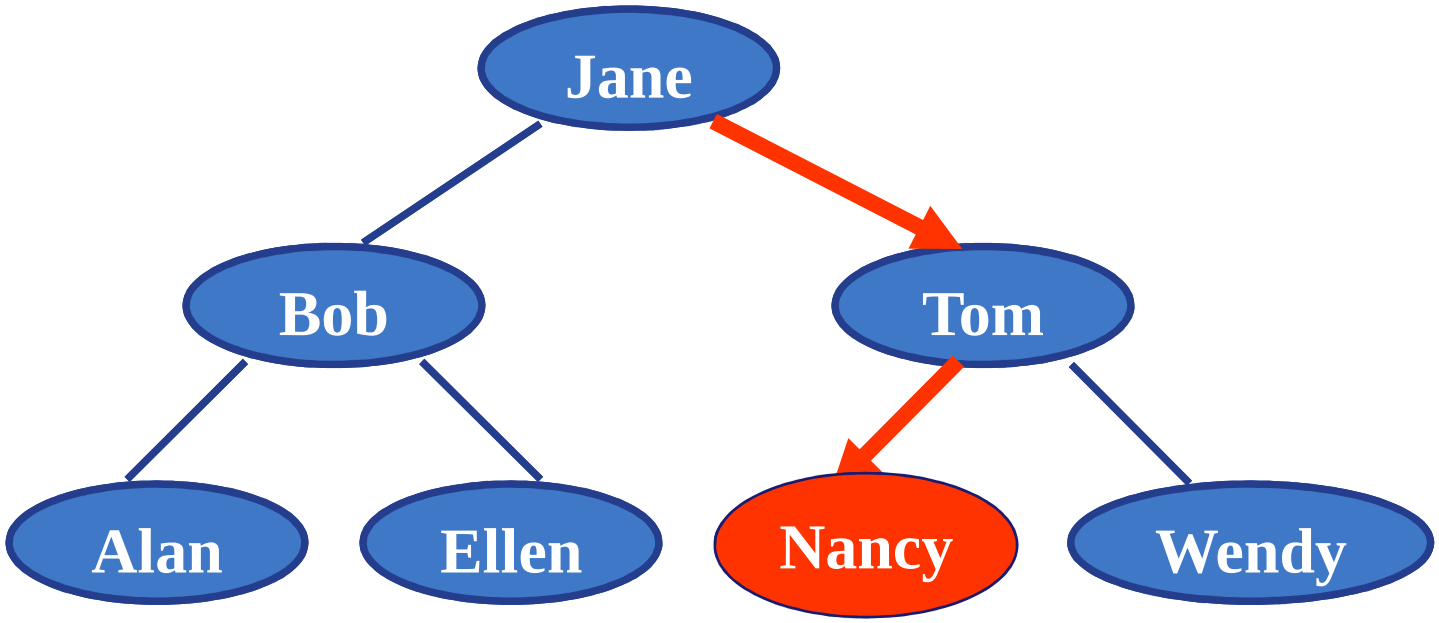
Xây dựng các thao tác cơ bản trên cây [4/24]

- Tìm kiếm 1 phần tử
  - Ví dụ tìm kiếm phần tử 25:



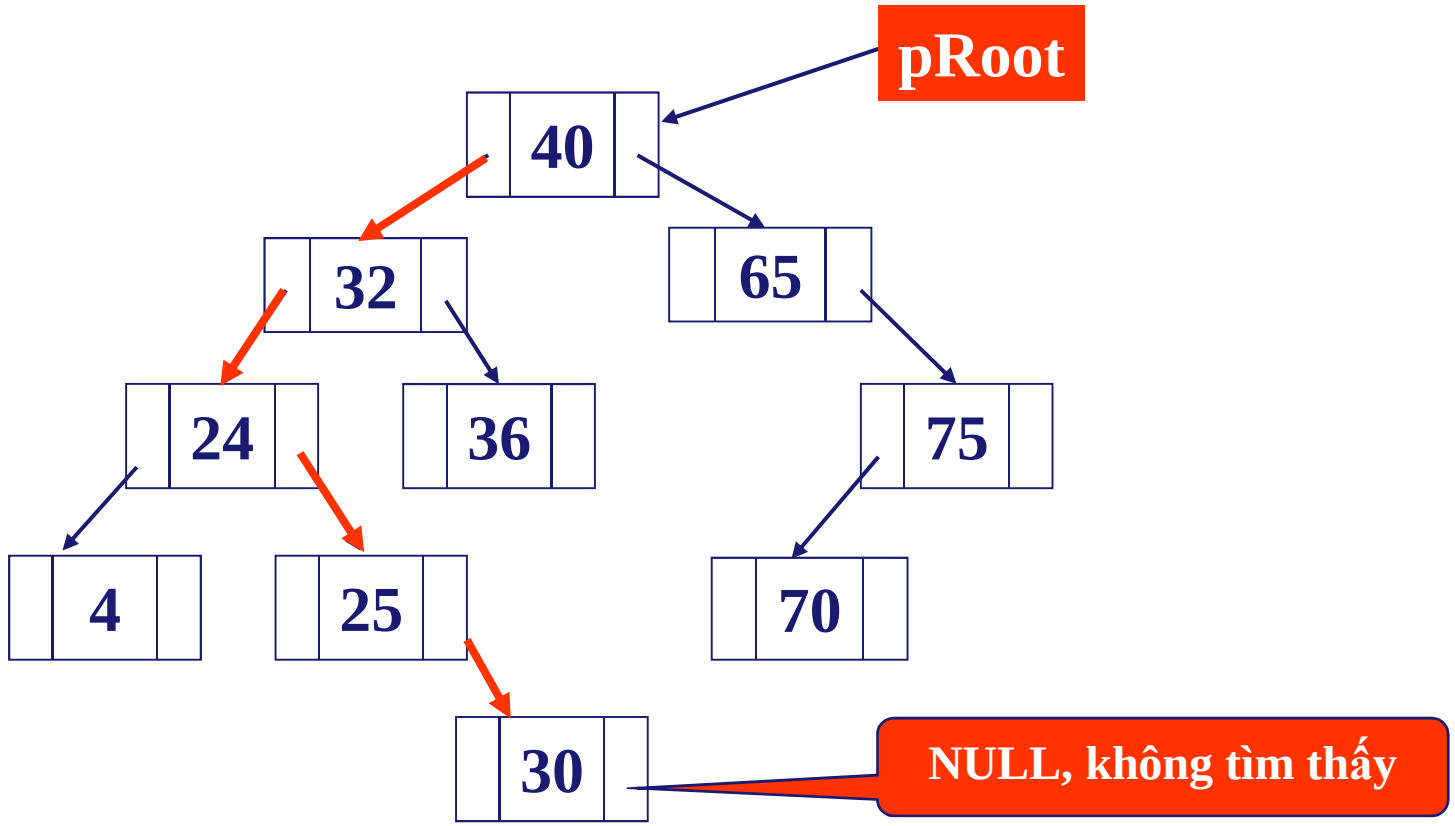
Xây dựng các thao tác cơ bản trên cây [5/24]

- Tìm kiếm 1 phần tử
  - Ví dụ tìm kiếm phần tử "Nancy":



Xây dựng các thao tác cơ bản trên cây [6/24]

- Tìm kiếm 1 phần tử
  - Ví dụ tìm kiếm phần tử 31:



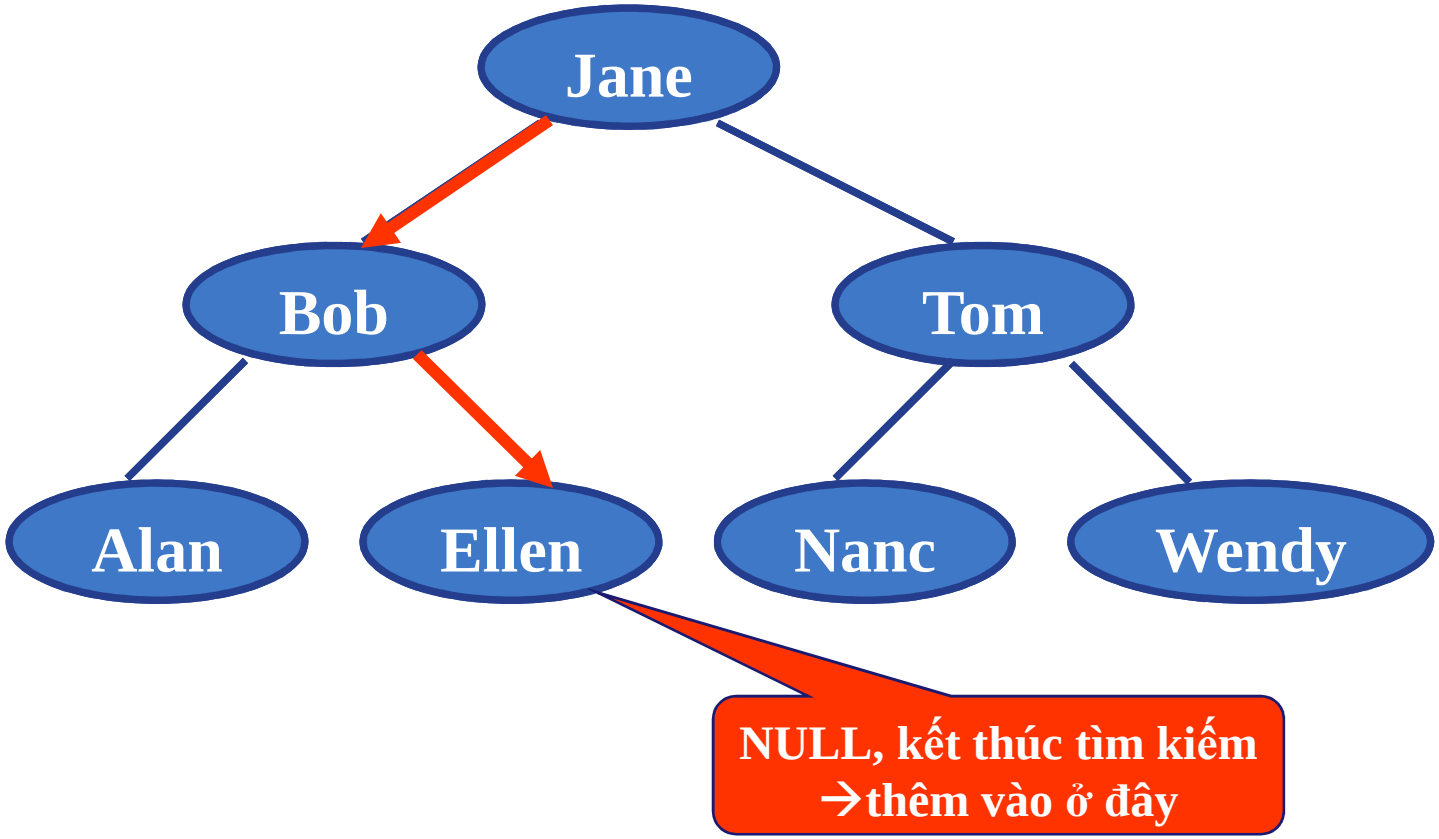
Xây dựng các thao tác cơ bản trên cây [7/24]

- Tìm kiếm 1 phần tử:

```
BT_NODE *BSTSearch(const BT_NODE *pCurr, int Key)
{
        // Không tìm thấy
      if (pCurr==NULL)  return NULL;
      if (pCurr->Data==Key) return pCurr;       // Tìm thấy
      else if (pCurr->Data > Key)
              // Tìm trong cây con trái
             return BSTSearch(pCurr->pLeft, Key);
           else
              // Tìm trong cây con phải
             return BSTSearch(pCurr->pRight, Key);
}
```

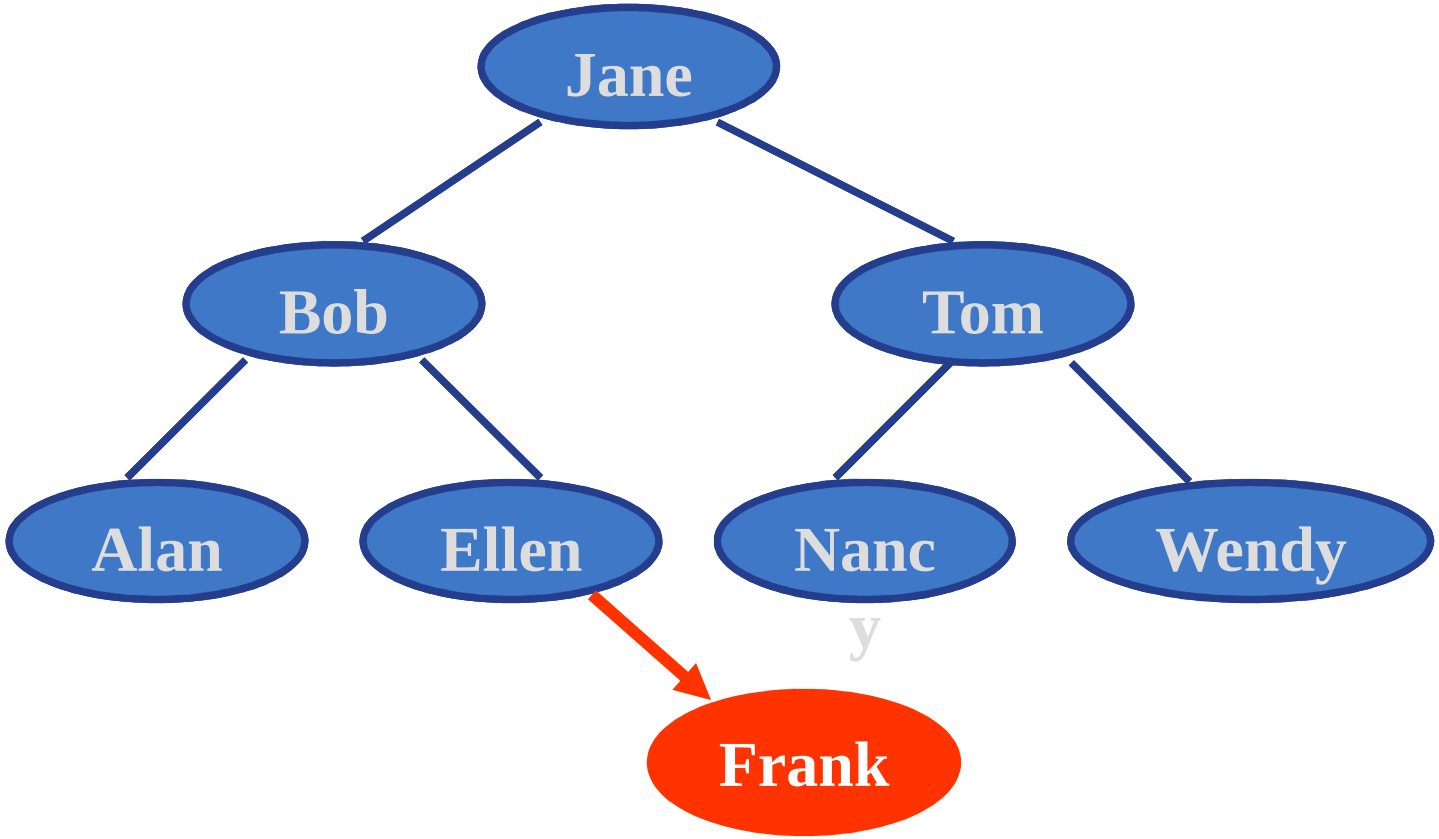
Xây dựng các thao tác cơ bản trên cây [8/24]

- Thêm 1 phần tử
  - Ví dụ thêm phần tử "Frank":



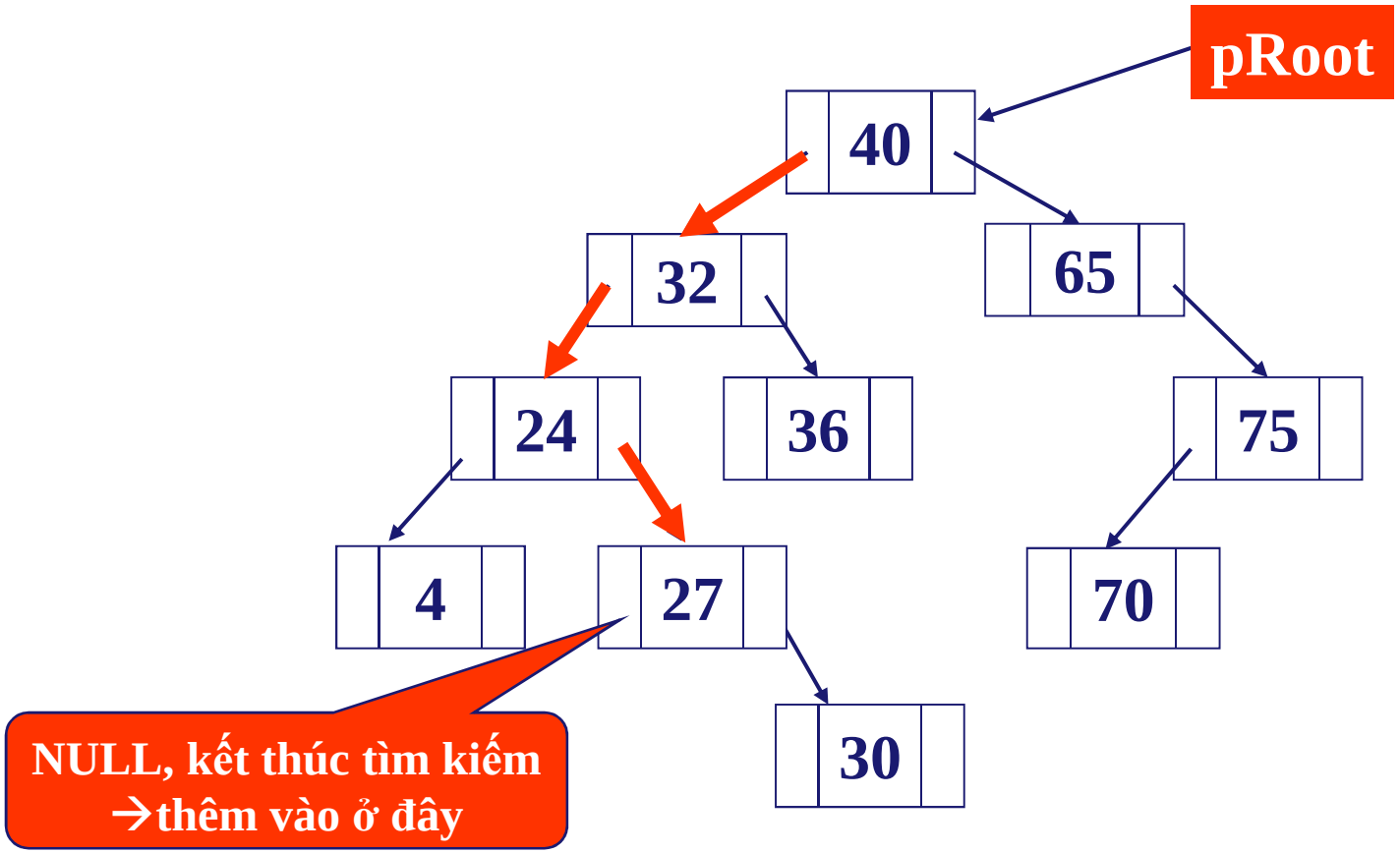
Xây dựng các thao tác cơ bản trên cây [9/24]

- Thêm 1 phần tử
  - Ví dụ thêm phần tử "Frank":



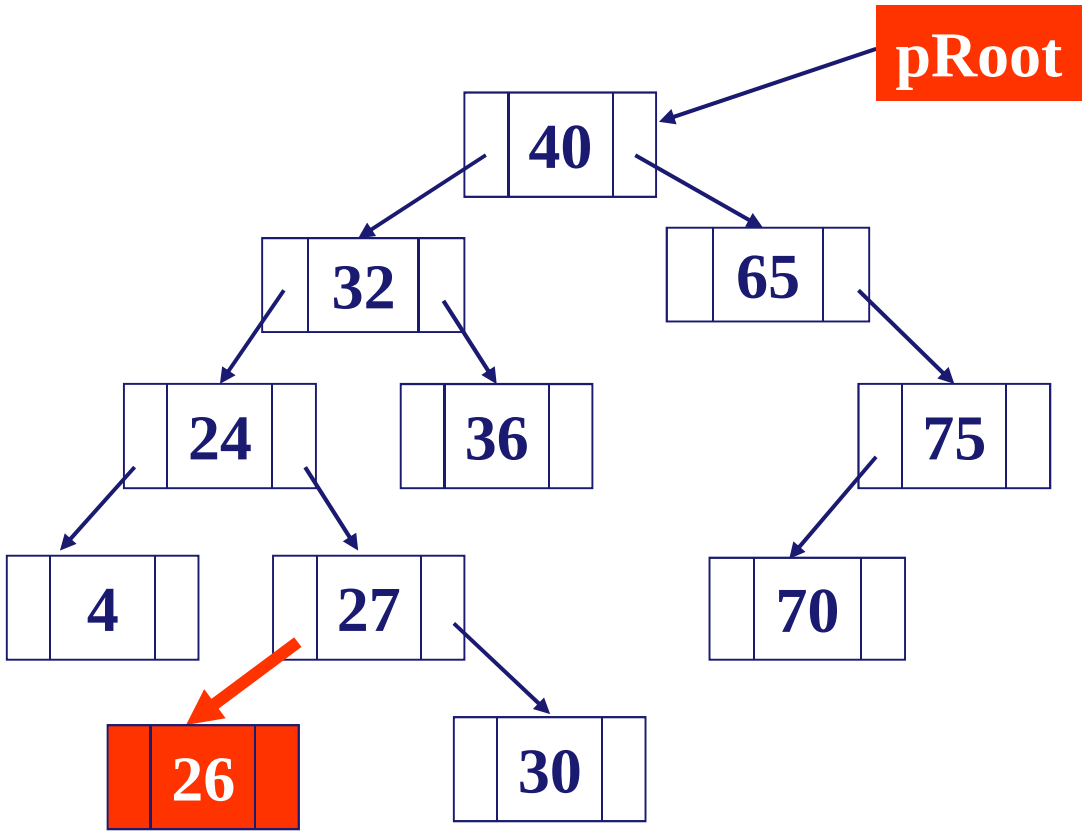
Xây dựng các thao tác cơ bản trên cây [10/24]

- Thêm 1 phần tử
  - Ví dụ thêm phần tử "26":



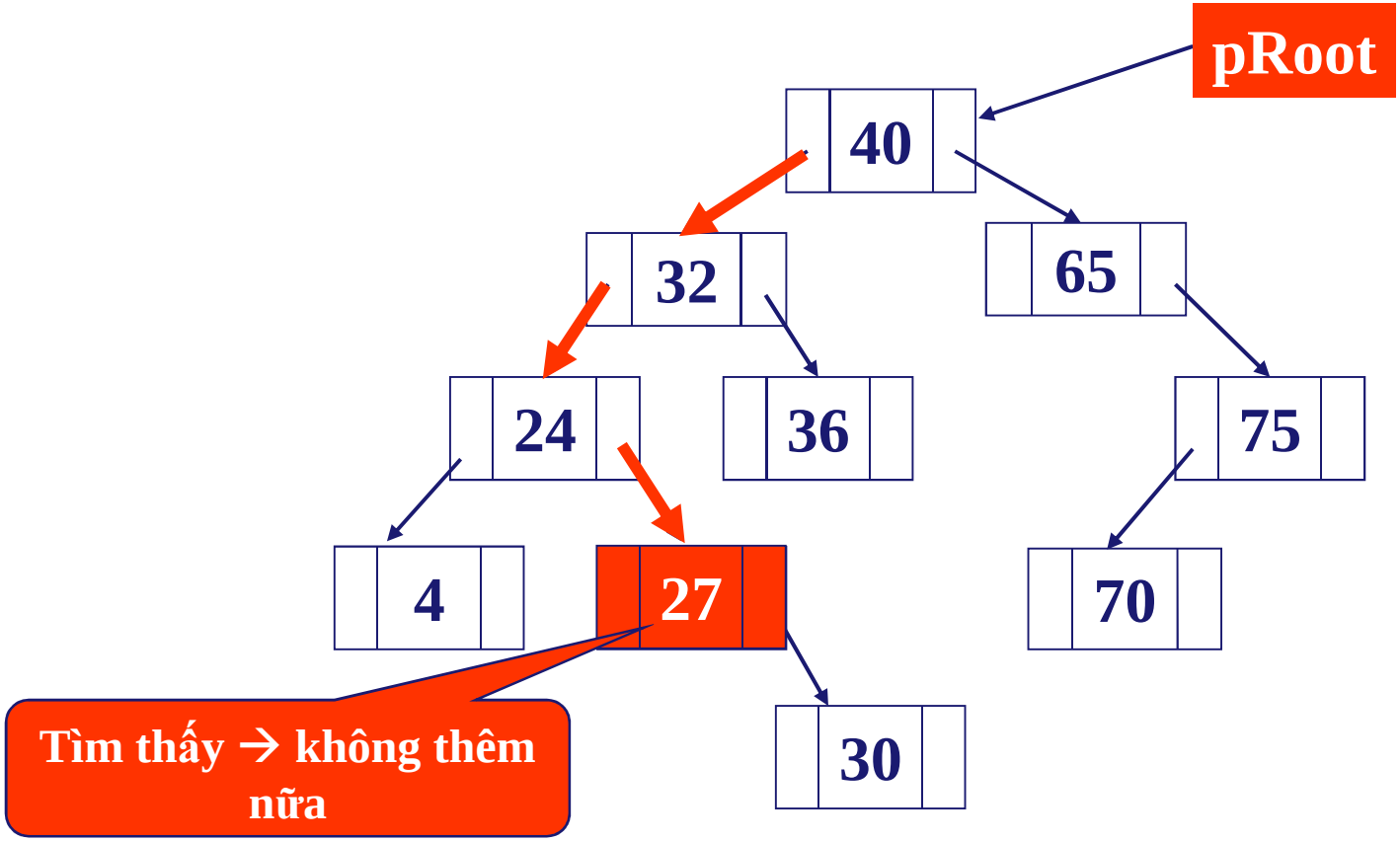
Xây dựng các thao tác cơ bản trên cây [11/24]

- Thêm 1 phần tử
  - Ví dụ thêm phần tử "26":



Xây dựng các thao tác cơ bản trên cây [12/24]

- Thêm 1 phần tử
  - Ví dụ thêm phần tử "27":



Xây dựng các thao tác cơ bản trên cây [13/24]

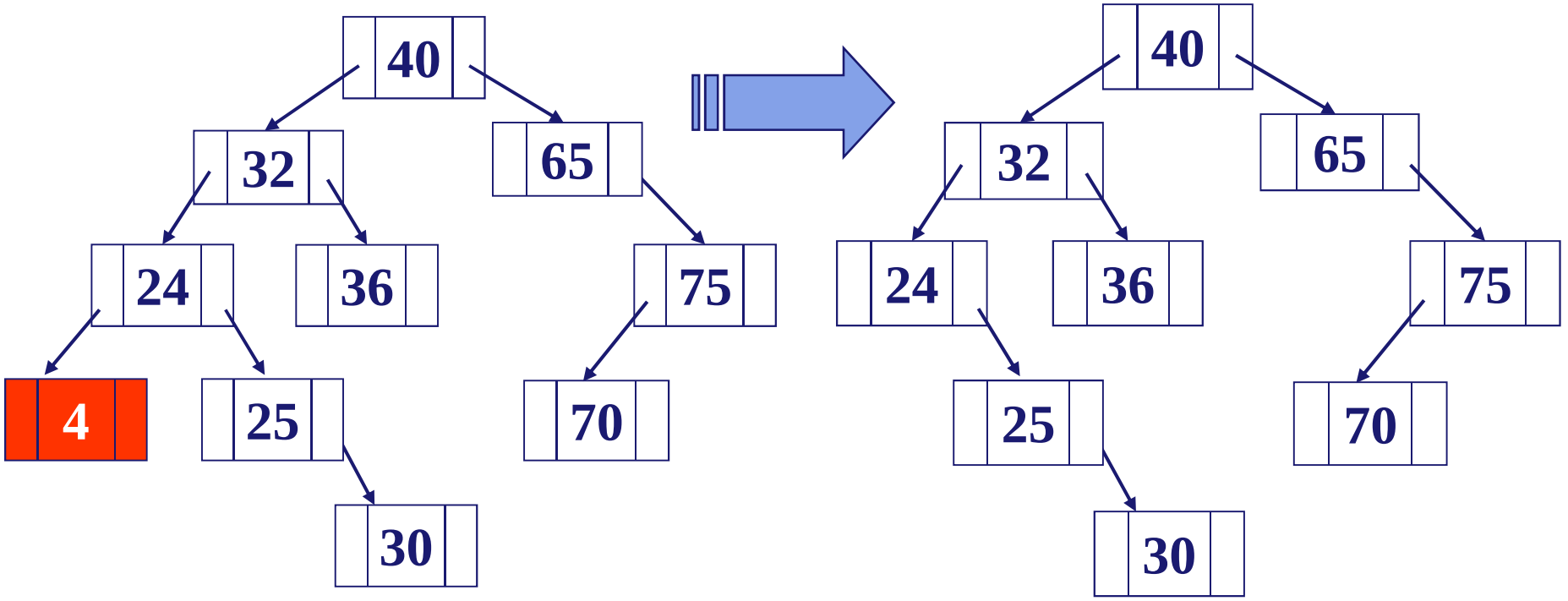
- Thêm 1 phần tử:

```
int  BSTInsert(BT_NODE *&pCurr, int newKey)
{
      if (pCurr==NULL) {
            pCurr = new BT_NODE;            // Tạo 1 nút mới
            pCurr->Data = newKey;
            pCurr->pLeft = pCurr->pRight = NULL;
            return 1;                       // Thêm thành công
      }
      if (pCurr->Data > newKey)
               // Thêm vào cây con trái
               return BSTInsert(pCurr->pLeft, newKey);
      else if (pCurr->Data < newKey)
               // Thêm vào cây con phải
               return BSTInsert(pCurr->pRight, newKey);
           else return 0;       // Trùng khóa, không thêm nữa
}
```

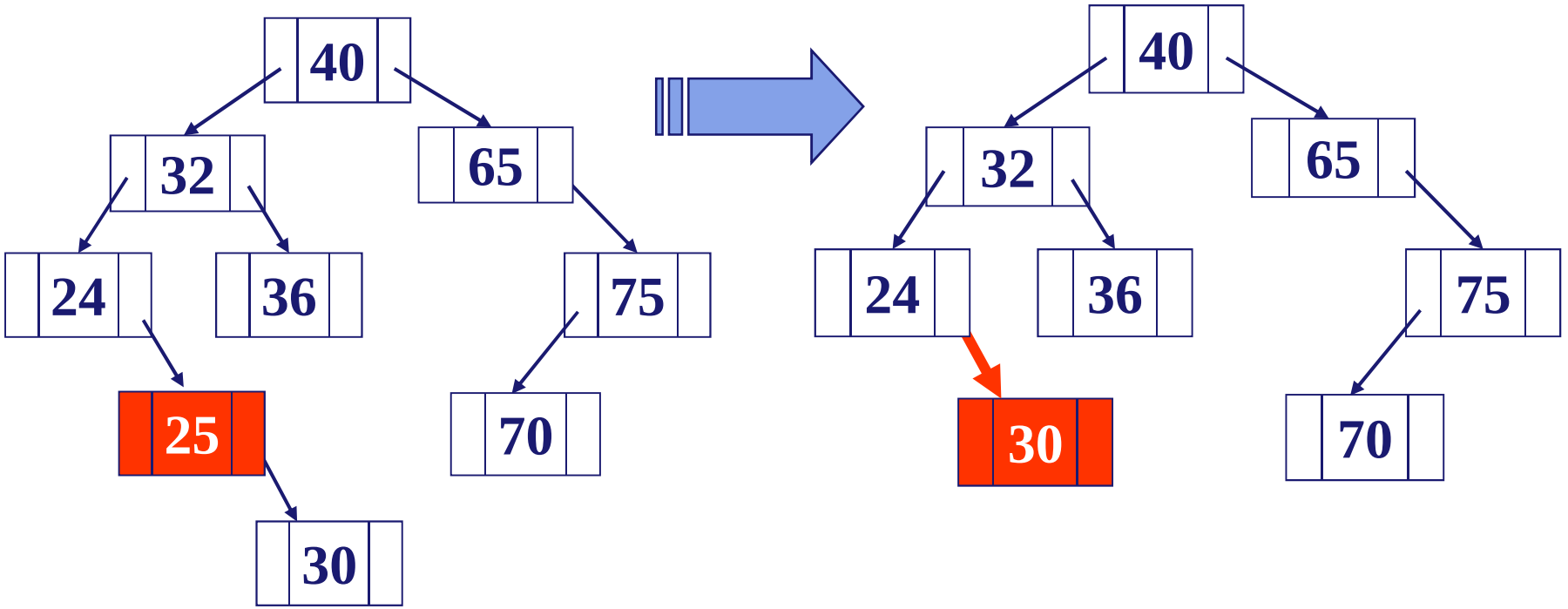
Xây dựng các thao tác cơ bản trên cây [14/24]

- Xóa 1 phần tử:
  - Áp dụng giải thuật tìm kiếm để xác định node chứa phần tử cần xóa
  - Nếu tìm thấy, xóa phần tử đó khỏi cây
- Các trường hợp xảy ra:
  - Xóa 1 node không có node con
  - Xóa 1 node chỉ có 1 node con
  - Xóa 1 node có 2 node con

- Xoá 1 phần tử:
  - Ví dụ xóa phần tử "4" (không có node con)

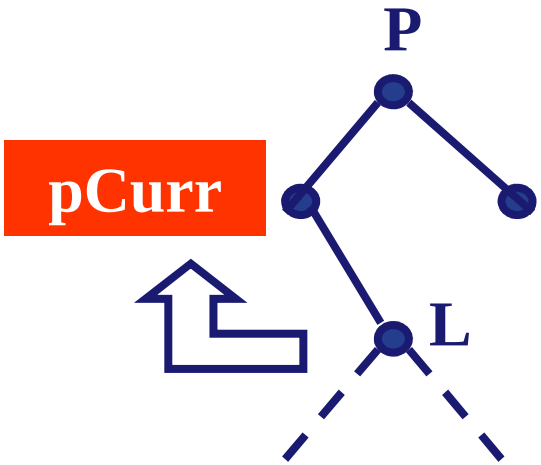


- Xoá 1 phần tử:
  - Ví dụ xóa phần tử "25" (chỉ có node con phải)

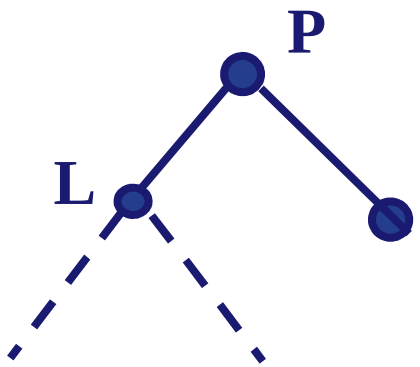


Xây dựng các thao tác cơ bản trên cây [17/24]

- Xoá 1 node chỉ có cây con phải:



Trước khi xóa pCurr

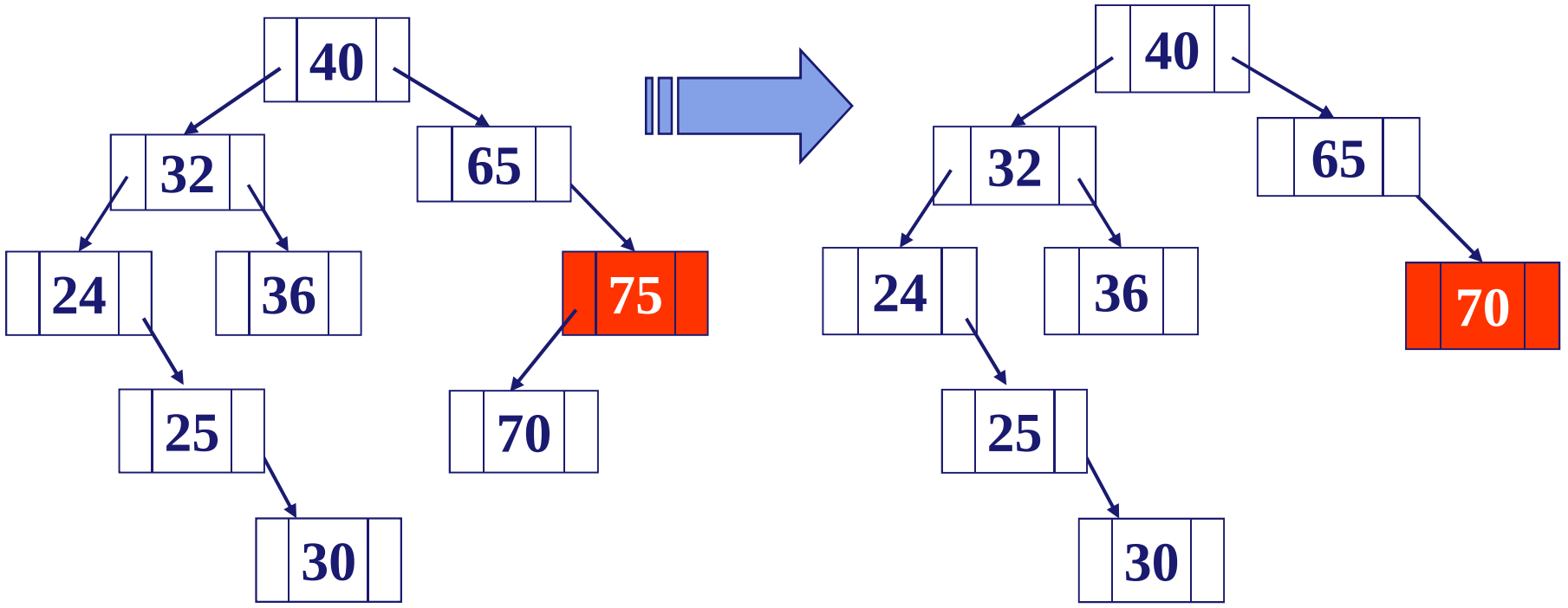


Sau khi xóa pCurr

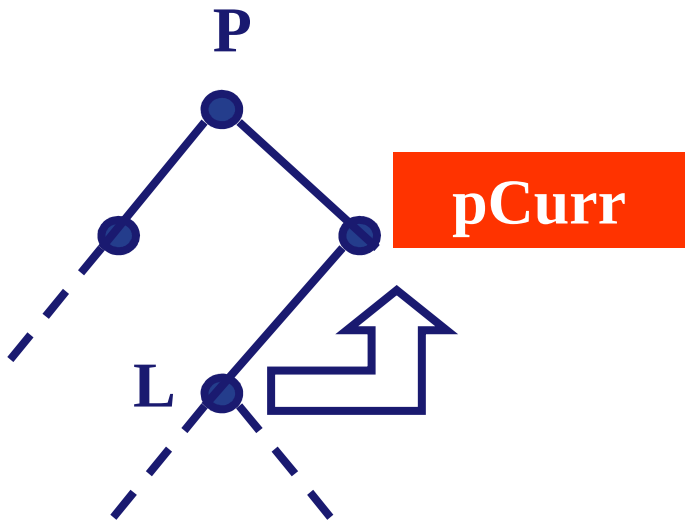
```
P->pLeft = pCurr->pRight;
delete pCurr;
```

Xây dựng các thao tác cơ bản trên cây [18/24]

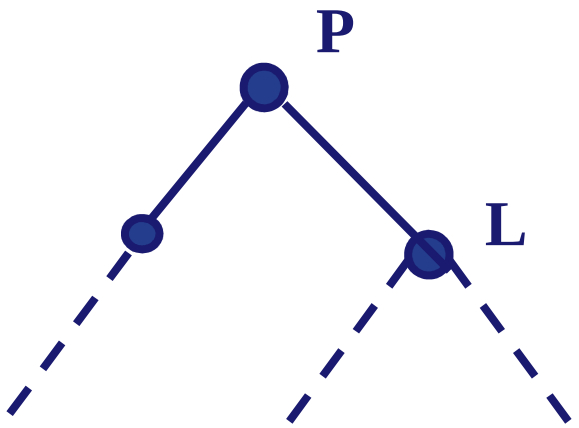
- Xoá 1 phần tử:
  - Ví dụ xóa phần tử "75" (chỉ có node con trái)



- Xoá 1 node chỉ có cây con phải:



Trước khi xóa pCurr

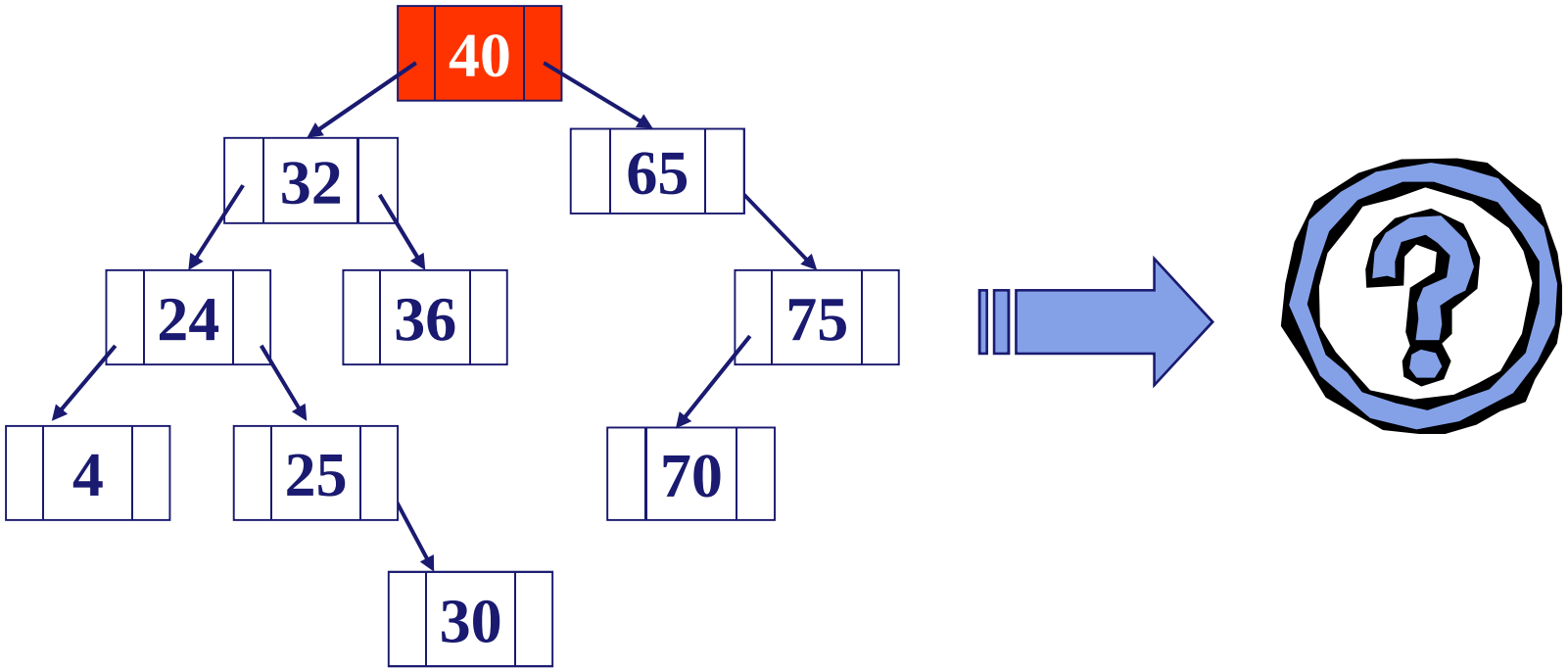


Sau khi xóa pCurr

```
P->pRight = pCurr->pLeft;
delete pCurr;
```

Xây dựng các thao tác cơ bản trên cây [20/24]

- Xoá 1 phần tử:
  - Ví dụ xóa phần tử "40" (có 2 node con)



Xây dựng các thao tác cơ bản trên cây [21/24]

- Xóa phần tử pCurr có 2 node con:
  - Thay vì xóa trực tiếp node pCurr, ta tìm 1 phần tử thay thế cho pCurr (gọi là phần tử p) → copy nội dung của p sang pCurr → xóa node p

- Phần tử thay thế p:
  - *Cách 1:* là phần tử lớn nhất trong cây con bên trái
  - *Cách 2:* là phần tử nhỏ nhất trong cây con bên phải

  → phần tử p sẽ có nhiều nhất là 1 cây con

Xây dựng các thao tác cơ bản trên cây [22/24]

- Xóa 1 phần tử:

```
int  BSTDelete(BT_NODE *&pCurr, int Key)
{
        // Đã duyệt hết cây, không tìm thấy phần tử
       if (pCurr==NULL)   return 0;
       if (pCurr->Data > Key)          // Xóa trên cây con trái
                return BSTDelete(pCurr->pLeft, Key);
       else if (pCurr->Data < Key)     // Xóa trên cây con phải
                return BSTDelete(pCurr->pRight, Key);

       // Tìm thấy nút cần xóa pCurr → Xóa !
       _Delete(pCurr);
       return 1;
}
```

Xây dựng các thao tác cơ bản trên cây [23/24]

- Xóa 1 phần tử (tt):

```
void  _Delete(BT_NODE *&pCurr)
{
      BT_NODE  *pTemp = pCurr;
      if (pCurr->pRight==NULL)      // Chỉ có 1 cây con trái
            pCurr = pCurr->pLeft;  // Giữ lại cây con trái
      else if (pCurr->pLeft==NULL)      // Chỉ có 1 cây con phải
            pCurr = pCurr->pRight;    // Giữ lại cây con phải
          else                          // Có 2 nhánh con
            pTemp = _SearchStandFor(pCurr->pLeft, pCurr);

      delete pTemp;
}
```

Xây dựng các thao tác cơ bản trên cây [24/24]

- Xóa 1 phần tử (tt):

```
// Tìm phần tử thay thế: "Phần tử lớn nhất trong cây con bên trái"
BT_NODE *  _SearchStandFor(BT_NODE *&p, BT_NODE *pCurr)
{
        if (p->pRight != NULL)
                return _SearchStandFor(p->pRight, pCurr);

        // Tìm thấy phần tử thay thế…
        pCurr->Data = p->Data;    // Copy dữ liệu của p vào pCurr
        BT_NODE *pTemp = p;
        p = p->pLeft;             // Lưu lại nhánh con trái
        return  pTemp;            // return & Xóa phần tử thay thế
}
```

Các đánh giá

- Cây có N node sẽ có độ cao trong khoảng từ $[\log_2(N+1)]$ đến N
  - *Trắc nghiệm*: khi nào độ cao của cây nhỏ nhất ? Lớn nhất ?

- So sánh giữa cây BST với mảng được sắp thứ tự:
  - Có cùng chi phí tìm kiếm $O(\log_2 N)$
  - Cây BST có chi phí thêm 1 phần tử $O(\log_2 N)$; mảng có chi phí thêm 1 phần tử O(N)
  - Tương tự đối với thao tác xóa 1 phần tử
  - Cây BST tốn nhiều bộ nhớ lưu trữ hơn (vì phải lưu các con trỏ left, right)

Các bài tập trắc nghiệm

- Viết hàm "Tìm phần tử thay thế: *Phần tử nhỏ nhất trong cây con bên phải*"
- Bài tập #5, #6, #7, #8: phần Ôn tập - 02

Nội dung

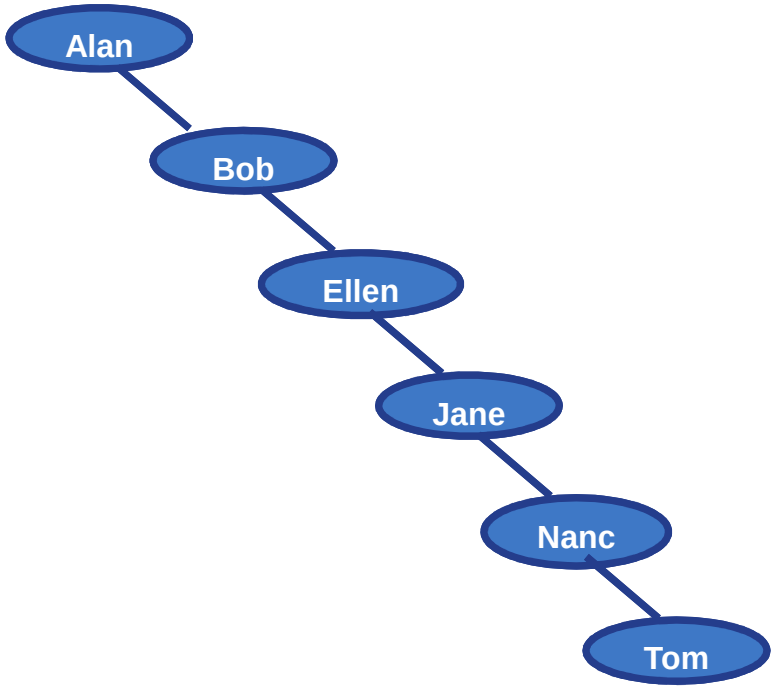
Các dạng cây nhị phân tìm kiếm cân bằng

- Vì sao phải cân bằng ?
- Cây AVL
- Cây Đỏ - Đen (Red – Black tree)
- Cây AA

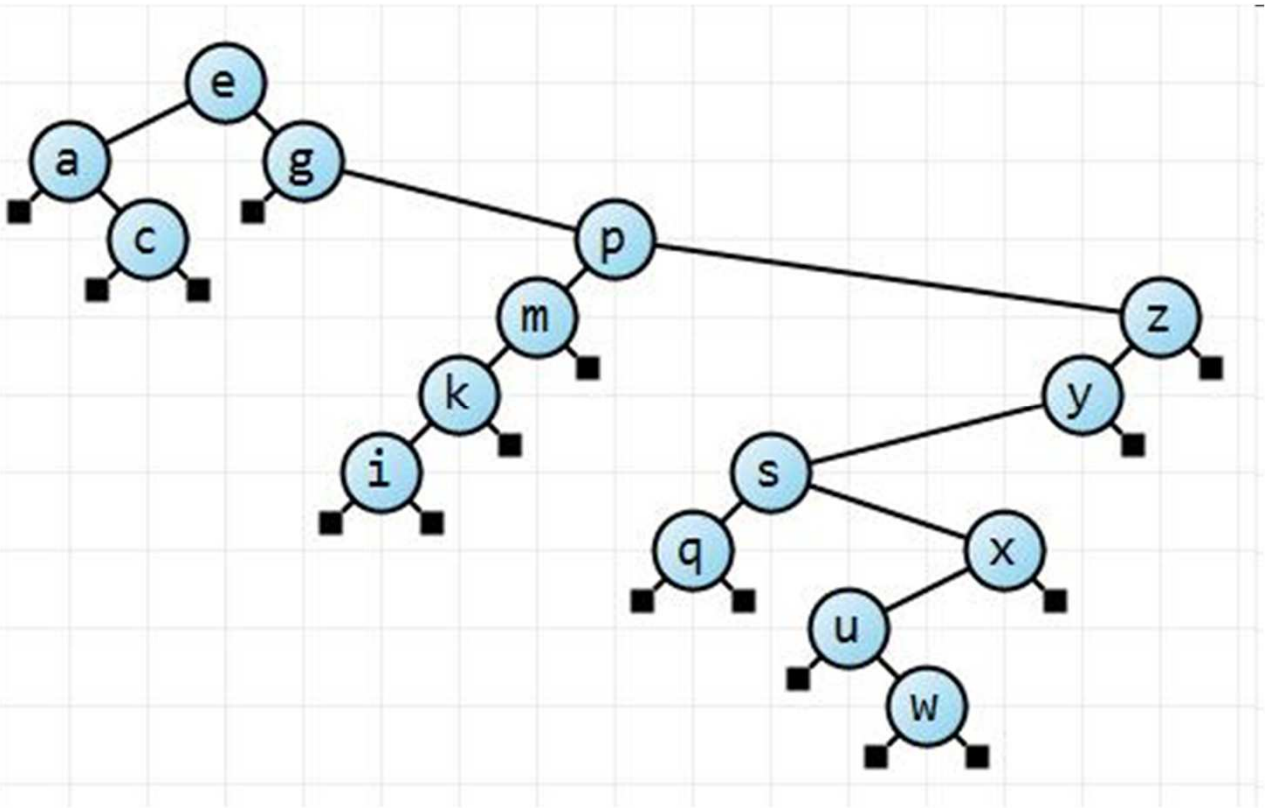
Vì sao phải cân bằng ? [1/3]

- Cây BST có thể bị lệch
  - Vì sao nào cây BST trở nên bị lệch ?
  - Chi phí tìm kiếm trên cây bị lệch ?

- Cần có phương pháp để duy trì tính cân bằng cho cây BST

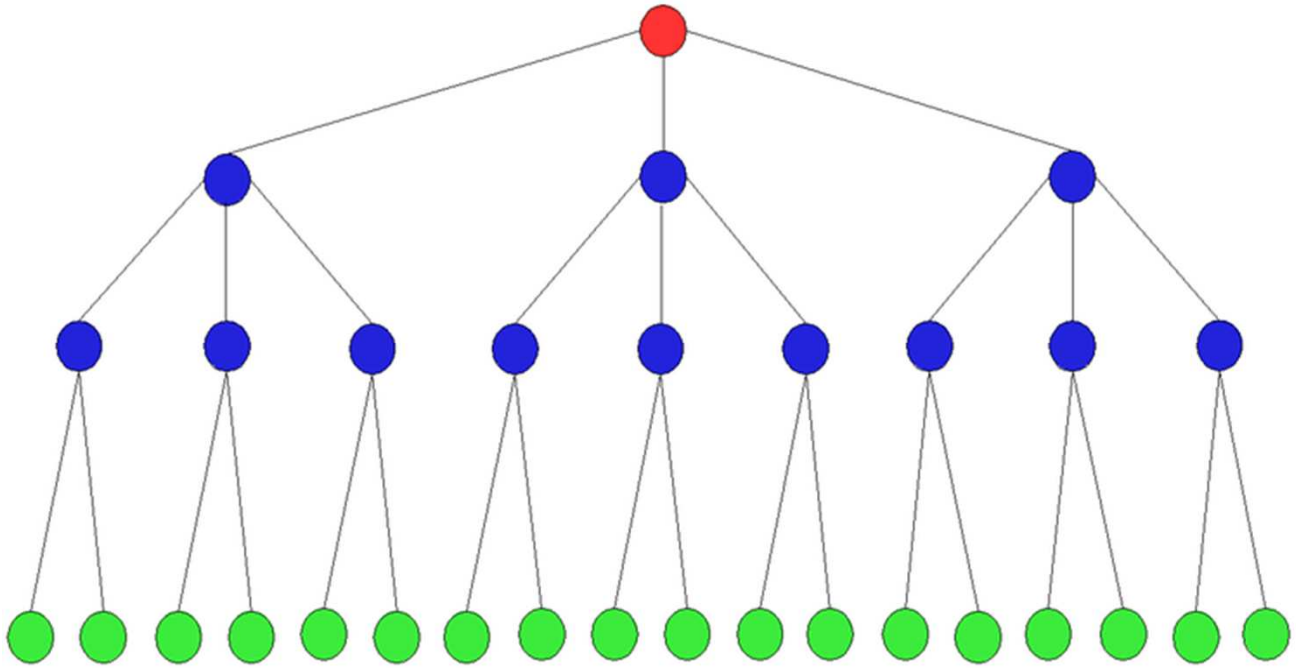


Vì sao phải cân bằng ? [2/3]



Một dạng cây không cân bằng

Vì sao phải cân bằng ? [3/3]



Cây cân bằng → chiều cao và chi phí tìm kiếm tối ưu $O(\log_2 N)$

Cây AVL

- Định nghĩa
- Ví dụ
- Mô tả cấu trúc dữ liệu
- Thao tác điều chỉnh cây
- Ví dụ tạo cây
- Các đánh giá

Định nghĩa cây AVL [1/2]

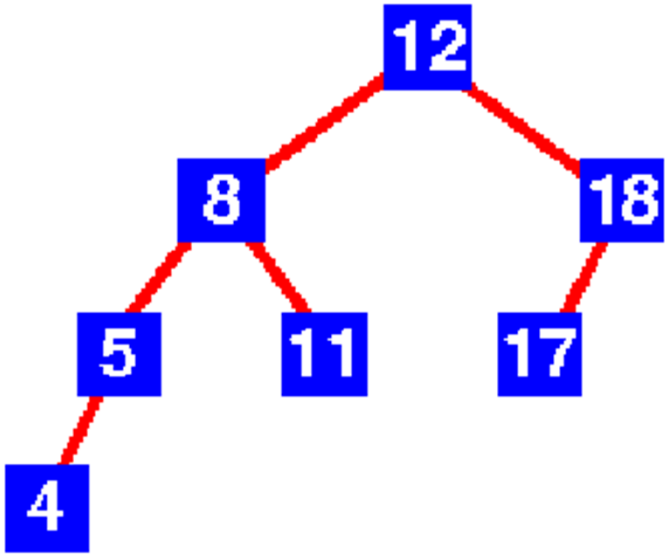
- Cây AVL là:
  - Một cây nhị phân tìm kiếm (BST)
  - Mỗi nút p của cây đều thỏa: chiều cao của cây con bên trái (p->pLeft) và chiều cao của cây con bên phải (p->pRight) chênh lệch nhau không quá 1

$$\forall p \in T_{AVL}: abs(h_{p->pLeft} - h_{p->pRight}) \leq 1$$

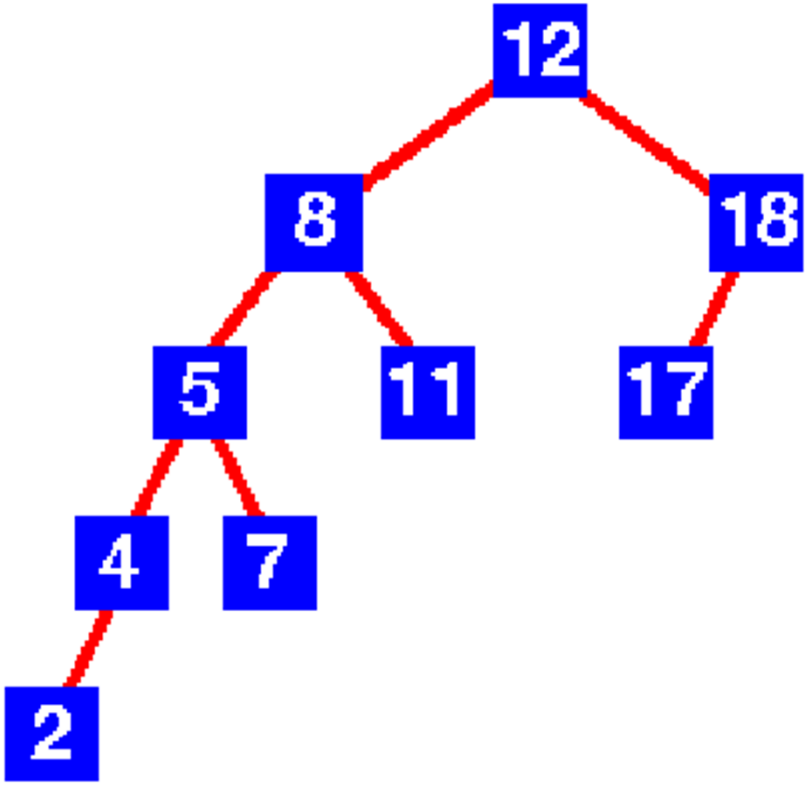
Định nghĩa cây AVL [2/2]

- Cây AVL là 1 dạng cây BST cân bằng
- Cấu trúc cây AVL do 3 tác giả: Adelson, Velskii, Landis đề xuất năm 1962
- Đây là mô hình cây cân bằng động đầu tiên được đề xuất
- Cây AVL không có độ cân bằng "tuyệt đối", nhưng 2 cây con không bao giờ có độ cao chênh lệch quá 1 đơn vị.

Ví dụ



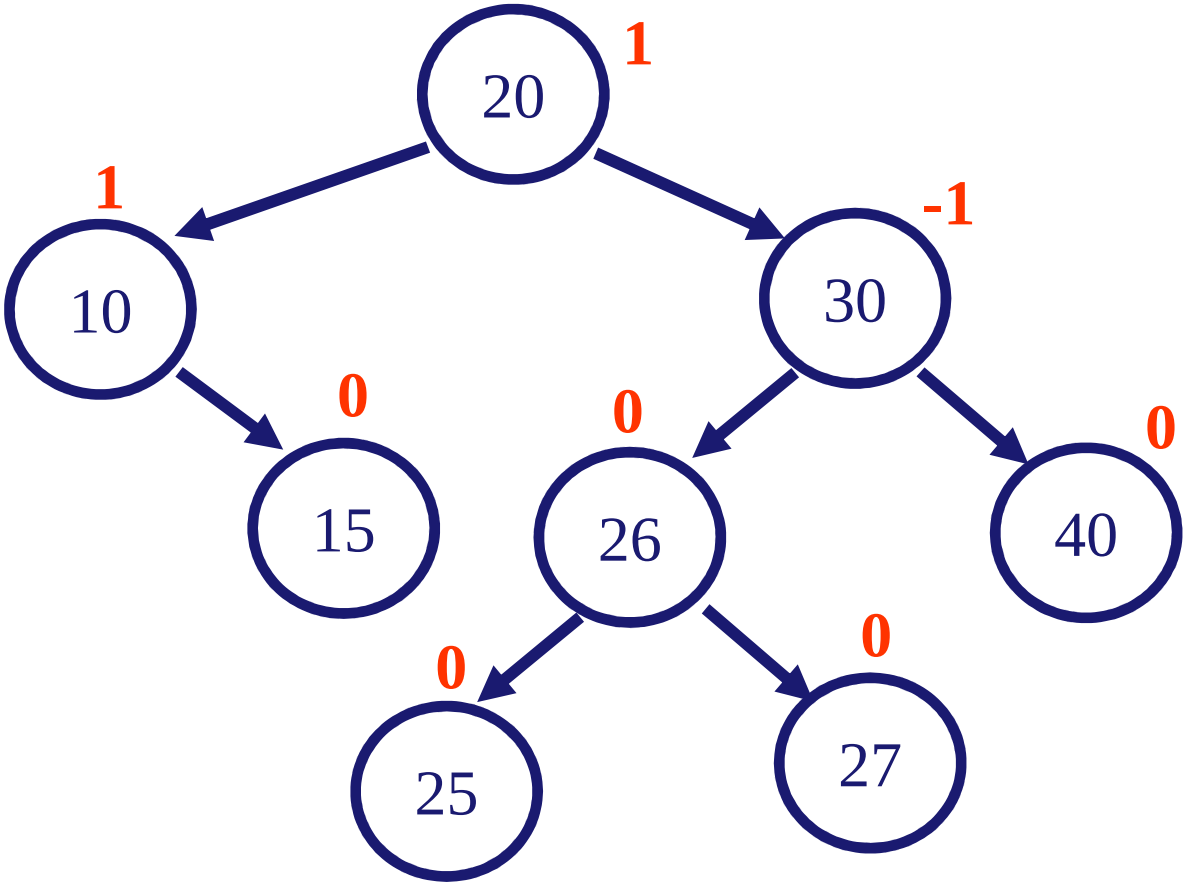
Cây AVL ?



Mô tả cấu trúc dữ liệu [1/4]

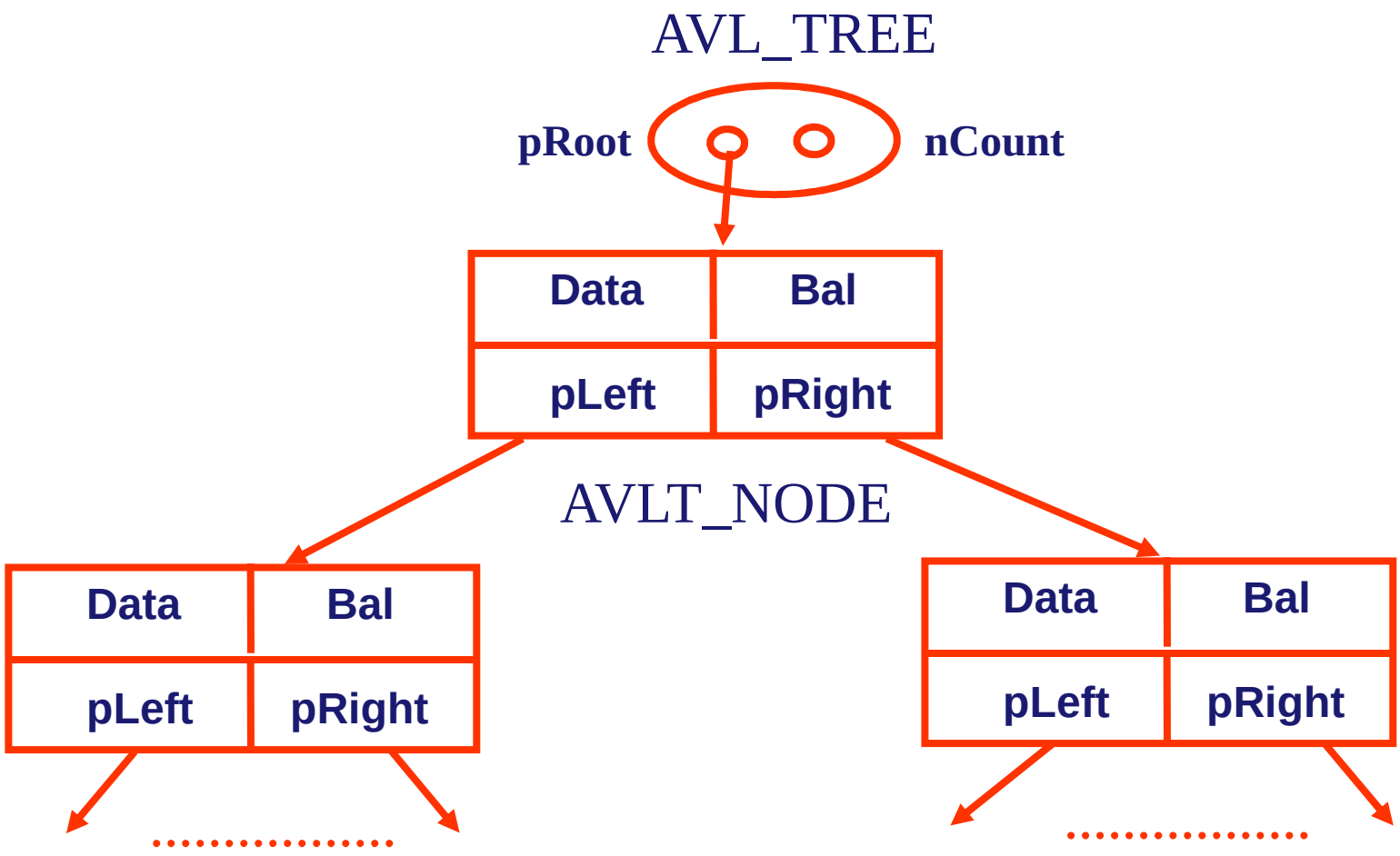
- Cấu trúc node, tree tương tự như BST
- Thêm vào mỗi node một field *Bal*, diễn tả trạng thái của node đó:
  - ***Bal* = *-1***: node lệch trái (cây con trái cao hơn cây con phải)
  - ***Bal* = *0***: node cân bằng (cây con trái cao bằng cây con phải)
  - ***Bal* = *+1***: node lệch phải (cây con phải cao hơn cây con trái)

Mô tả cấu trúc dữ liệu [2/4]



Hệ số cân bằng của các node trong cây AVL

Mô tả cấu trúc dữ liệu [3/4]



Mô tả cấu trúc dữ liệu [4/4]

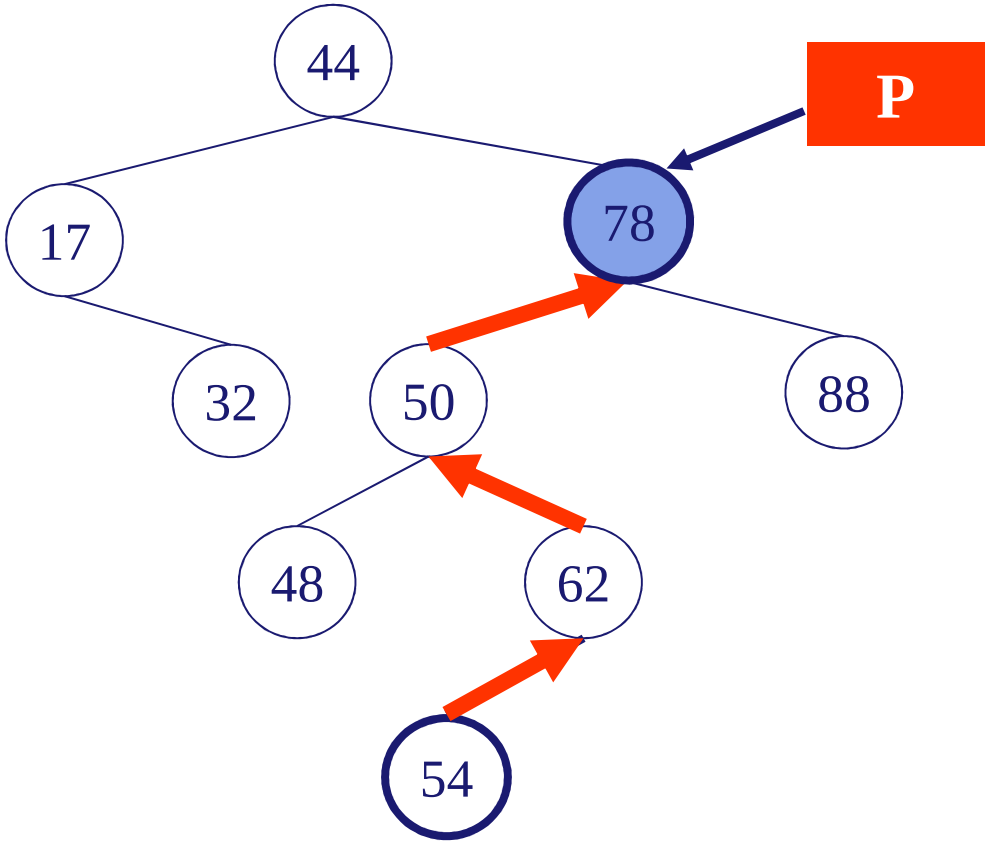
```
typedef struct tagAVLT_NODE {
    int                 Data;
    int                 Bal;        // Hệ số cân bằng (-1,0,1)
    tagAVLT_NODE    *pLeft;     // con trỏ đến cây con trái
    tagAVLT_NODE    *pRight;    // con trỏ đến cây con phải
} AVLT_NODE;                    // Cấu trúc node của cây AVL

typedef struct AVL_TREE {
    int                 nCount;     // Số node trong cây
    AVLT_NODE       *pRoot;     // Con trỏ đến node gốc
};                              // Cấu trúc cây AVL
```

Thao tác điều chỉnh cây [1/3]

- [Insert – Thêm 1 phần tử vào cây]: có thể làm cây mất cân bằng.
  - Ta duyệt từ nút vừa thêm ngược về nút gốc, …
  - …nếu tìm ra 1 nút P bị mất cân bằng, …
  - …thì tiến hành điều chỉnh lại cây tại nút P (điều chỉnh 1 lần duy nhất)

Thao tác điều chỉnh cây [2/3]

- *Ví dụ:* thêm phần tử làm cây mất cân bằng tại nút P



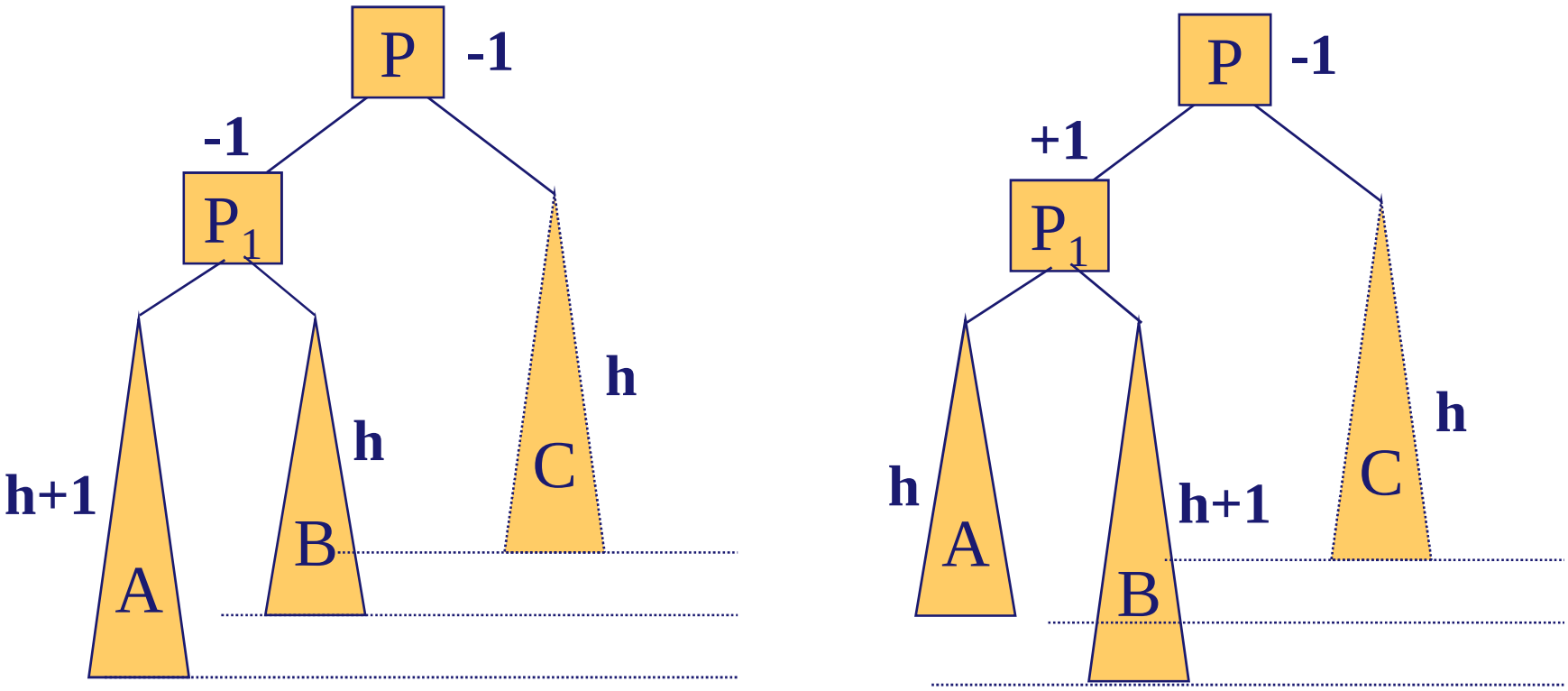
Thao tác điều chỉnh cây [3/3]

- [Delete – Xóa 1 phần tử]: có thể làm cây mất cân bằng.
  - Ta duyệt từ nút vừa xóa ngược về nút gốc, …
  - …nếu tìm ra 1 nút P bị mất cân bằng, …
  - …thì tiến hành điều chỉnh lại cây tại nút P
  - *Lưu ý:* Thao tác điều chỉnh có thể làm cho những nút phía trên nút P bị mất cân bằng → cần điều chỉnh cho đến khi không còn nút nào bị mất cân bằng nữa (lùi dần về node gốc)

Cây AVL

Những trường hợp cây bị mất cân bằng và
Các cách điều chỉnh cây

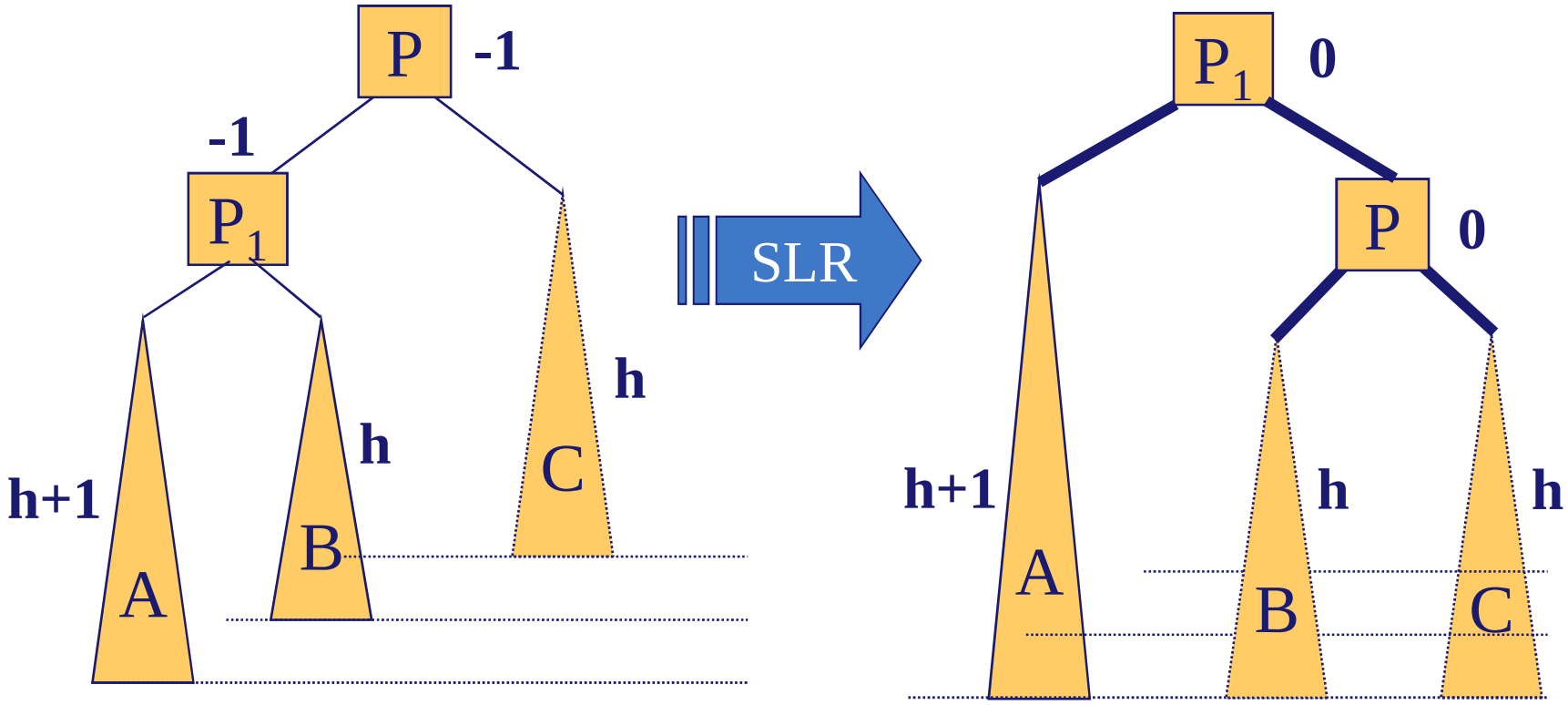
Các cách điều chỉnh cây [1/9]



(a) (b)

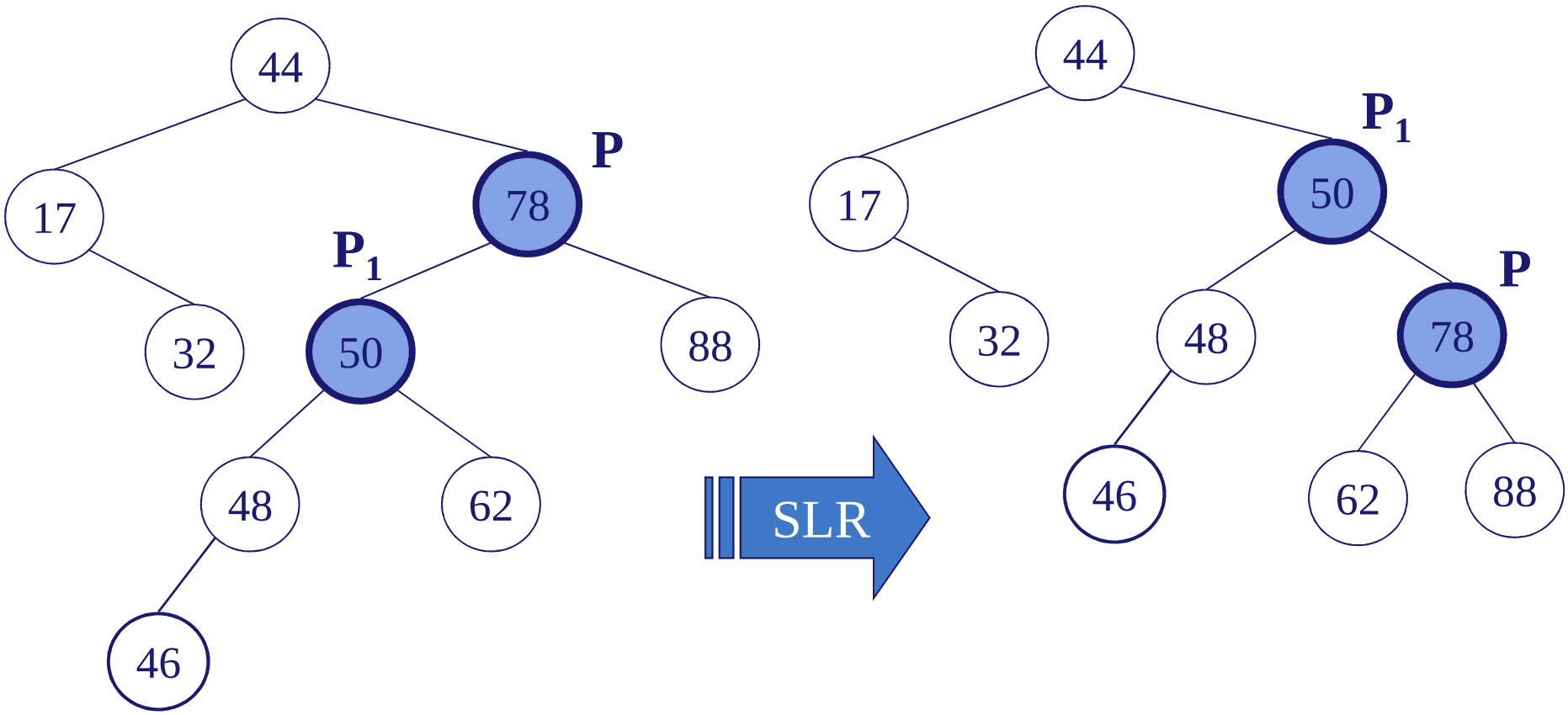
Hai trường hợp cây bị mất cân bằng ở nhánh trái

Các cách điều chỉnh cây [2/9]



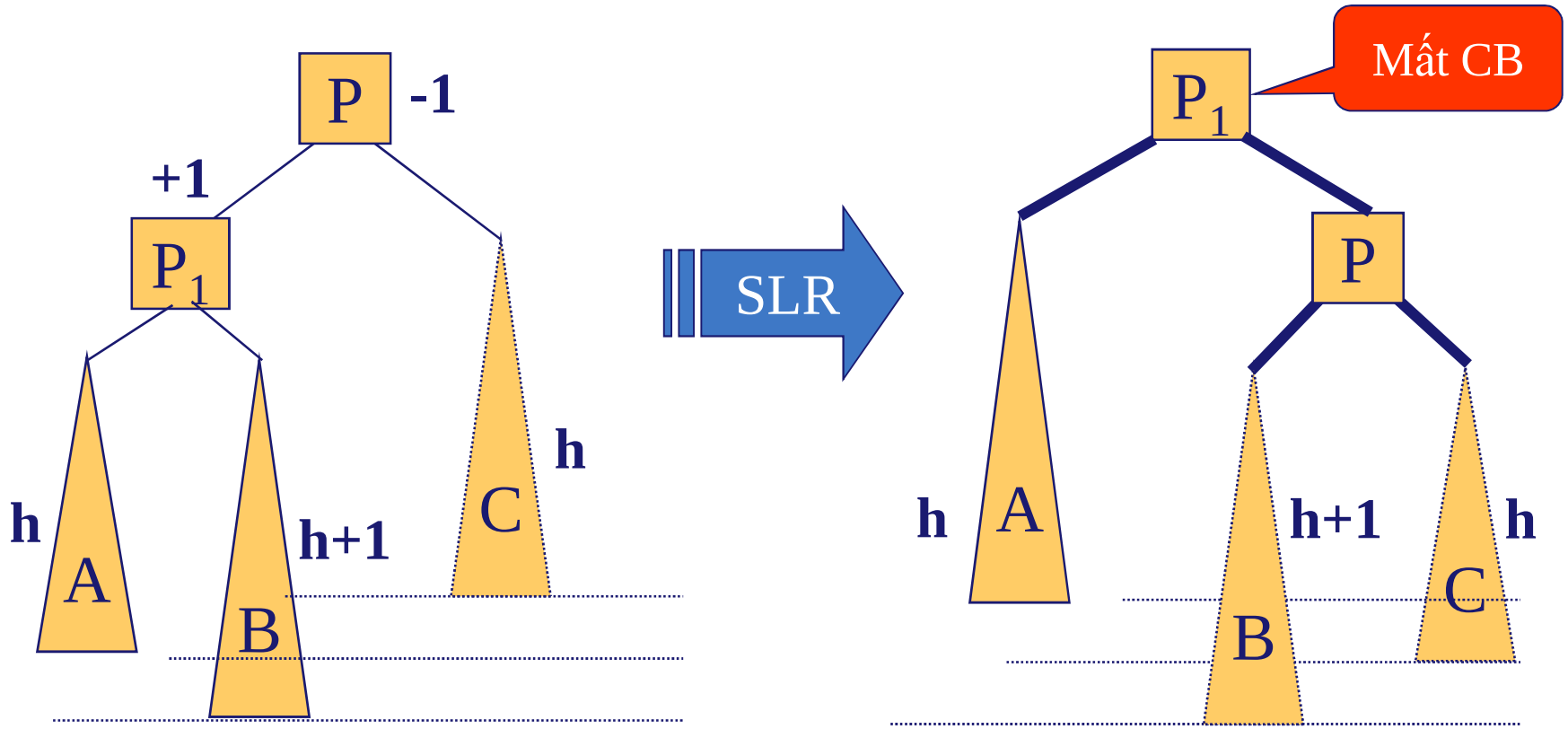
Trường hợp (a): áp dụng phép xoay đơn Trái - Phải
(SLR – Single Left-to-Right)

Các cách điều chỉnh cây [3/9]



Ví dụ: điều chỉnh cây bằng thao tác xoay đơn SLR

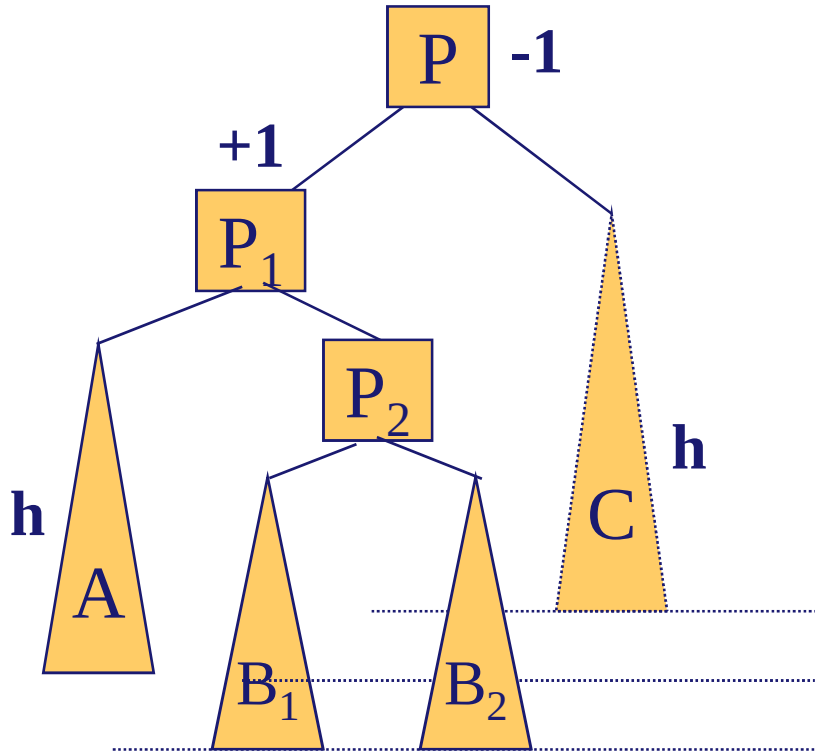
Các cách điều chỉnh cây [4/9]



Trường hợp (b): thử áp dụng phép xoay đơn SLR ?

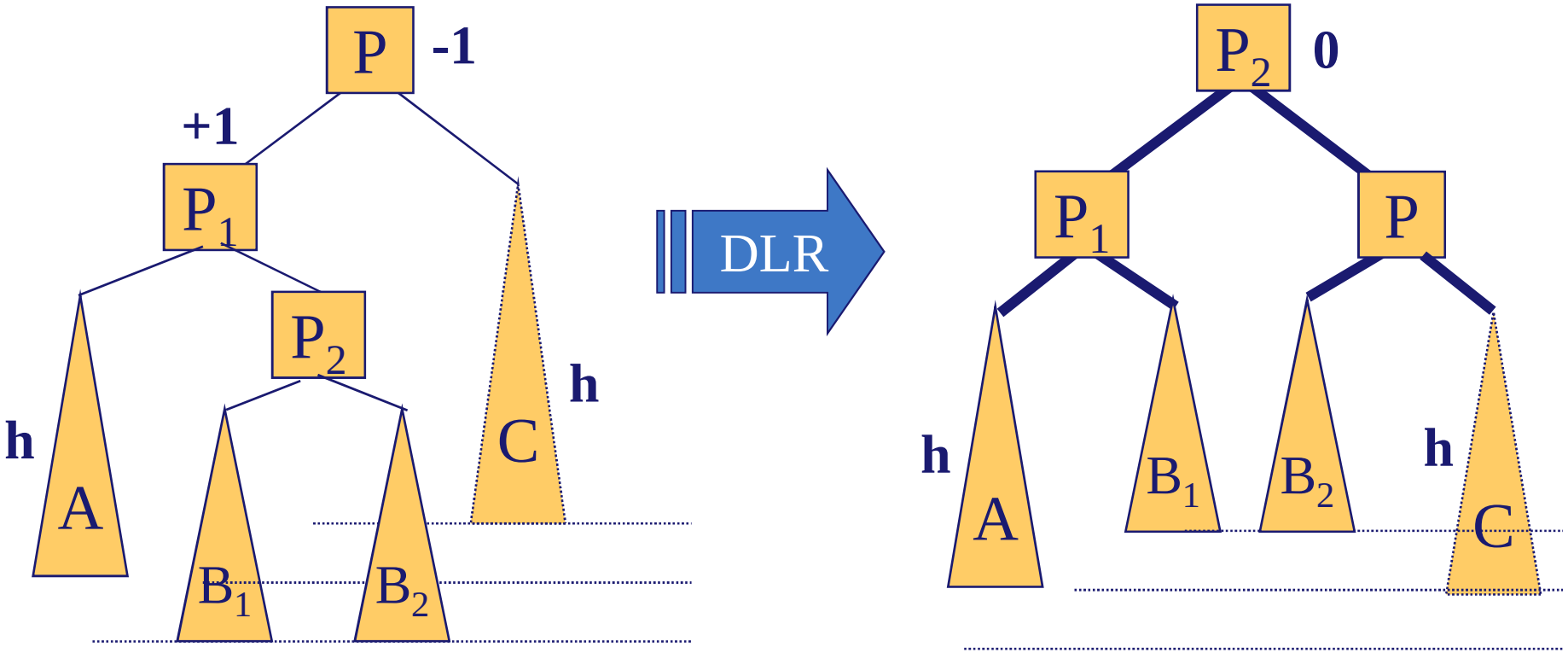
Các cách điều chỉnh cây [5/9]

Phép xoay kép Trái - Phải
(DLR – Double Left–to-Right)



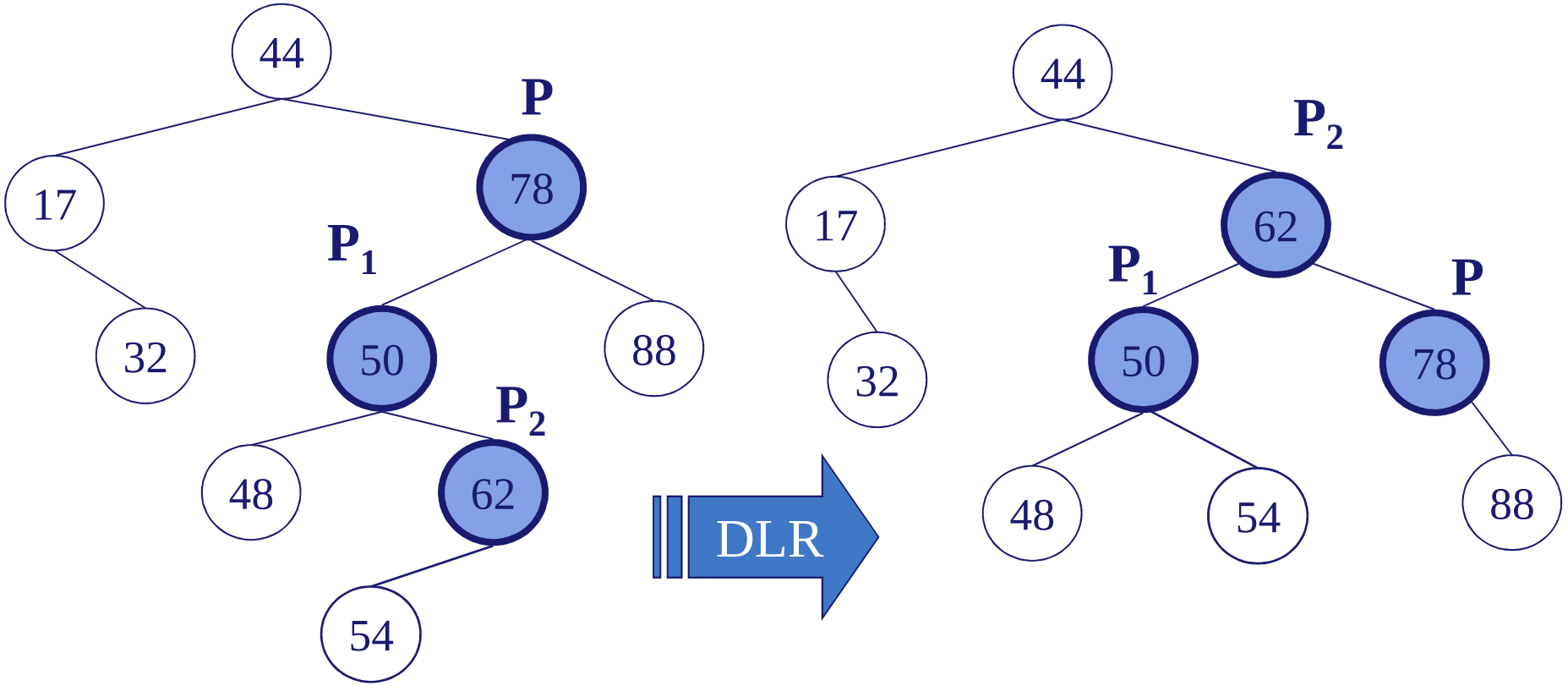
Trường hợp (b)

Các cách điều chỉnh cây [6/9]



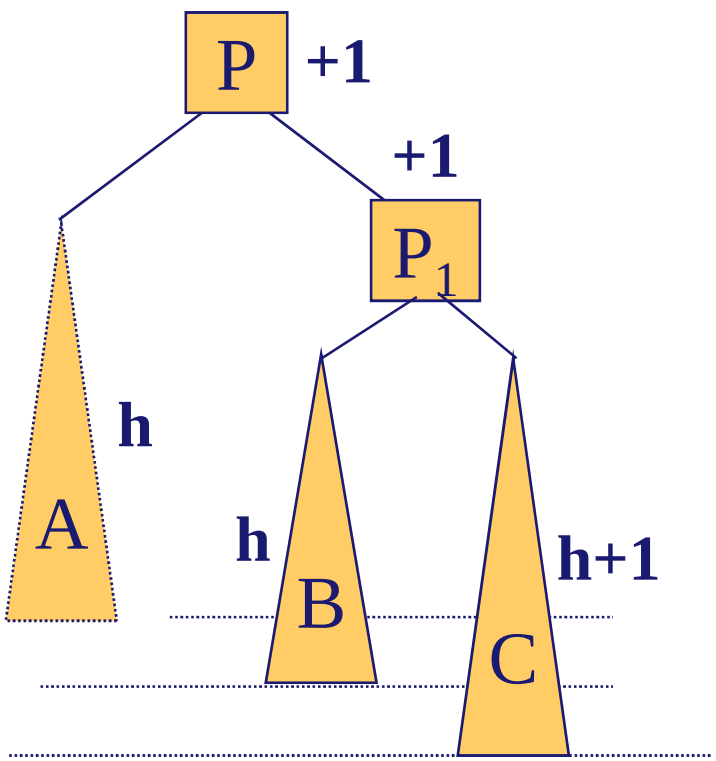
Trường hợp (b): áp dụng phép xoay kép Trái - Phải (DLR)

Các cách điều chỉnh cây [7/9]

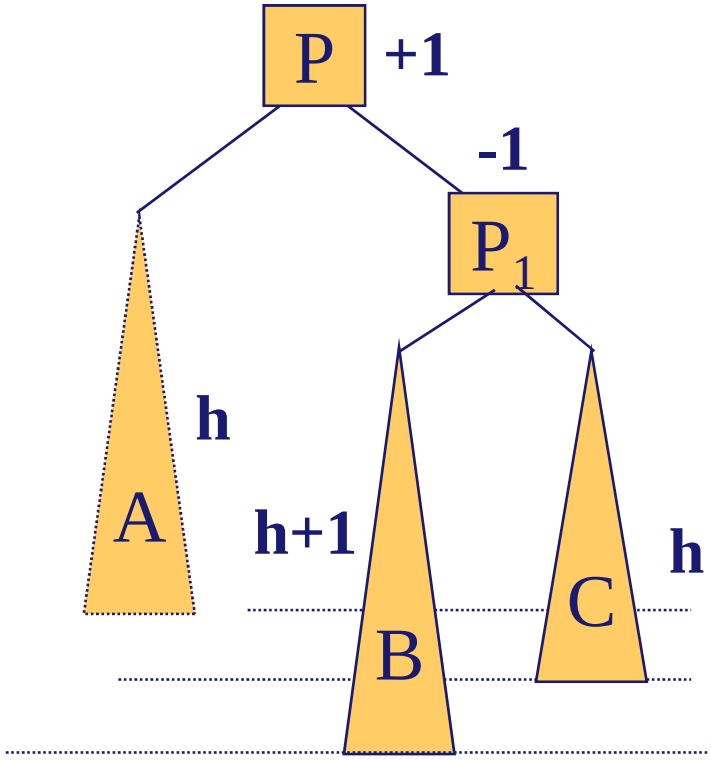


Ví dụ: thao tác xoay kép DLR

Các cách điều chỉnh cây [8/9]



(a')



(b')

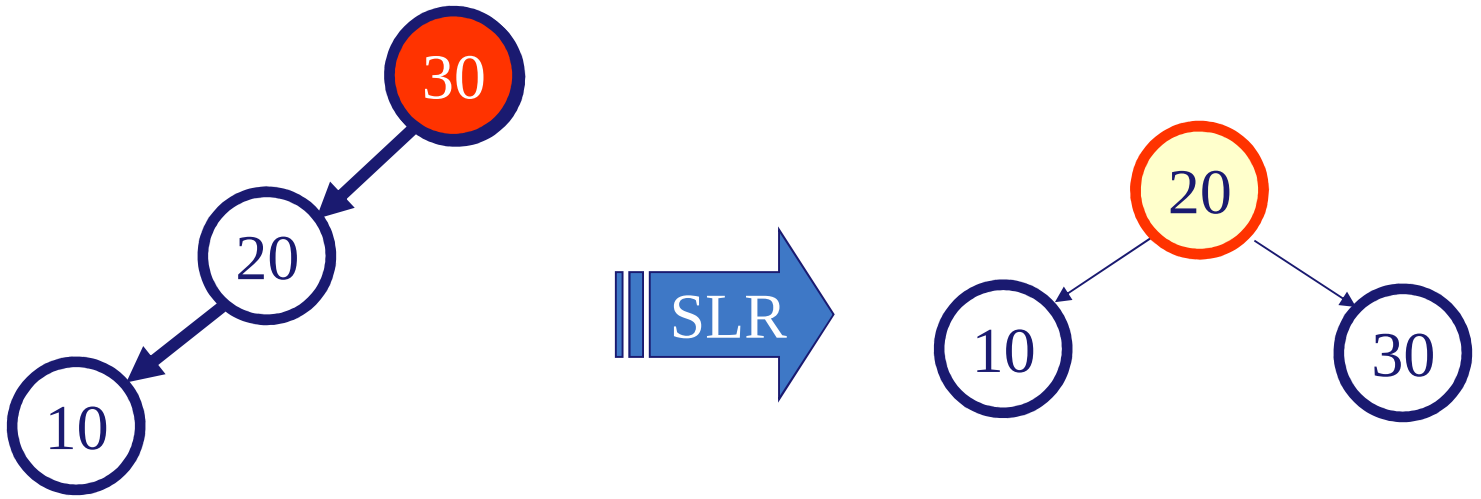
Hai trường hợp cây bị mất cân bằng ở nhánh phải

Các cách điều chỉnh cây [9/9]

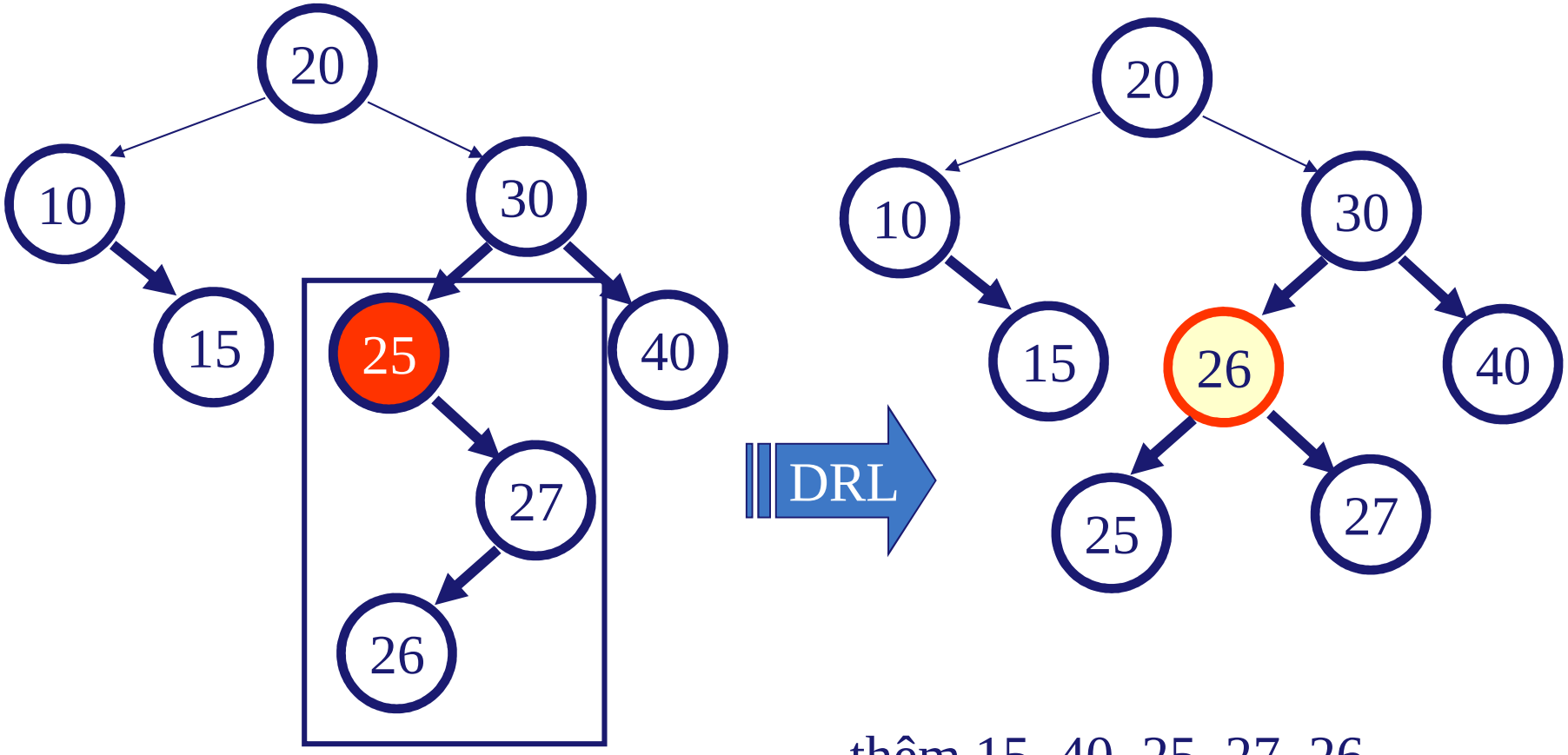
- Phương pháp xử lý cho trường hợp mất cân bằng cân bằng ở nhánh phải:
  - Tương tự như các xử lý mất cân bằng ở nhánh trái
  - Áp dụng 1 trong 2 phép xoay:
    - SRL – Single Right-to-Left
    - DRL – Double Right-to-Left

Ví dụ tạo cây AVL [1/3]

- Tạo cây AVL với các khóa lần lượt là: 30, 20, 10,…

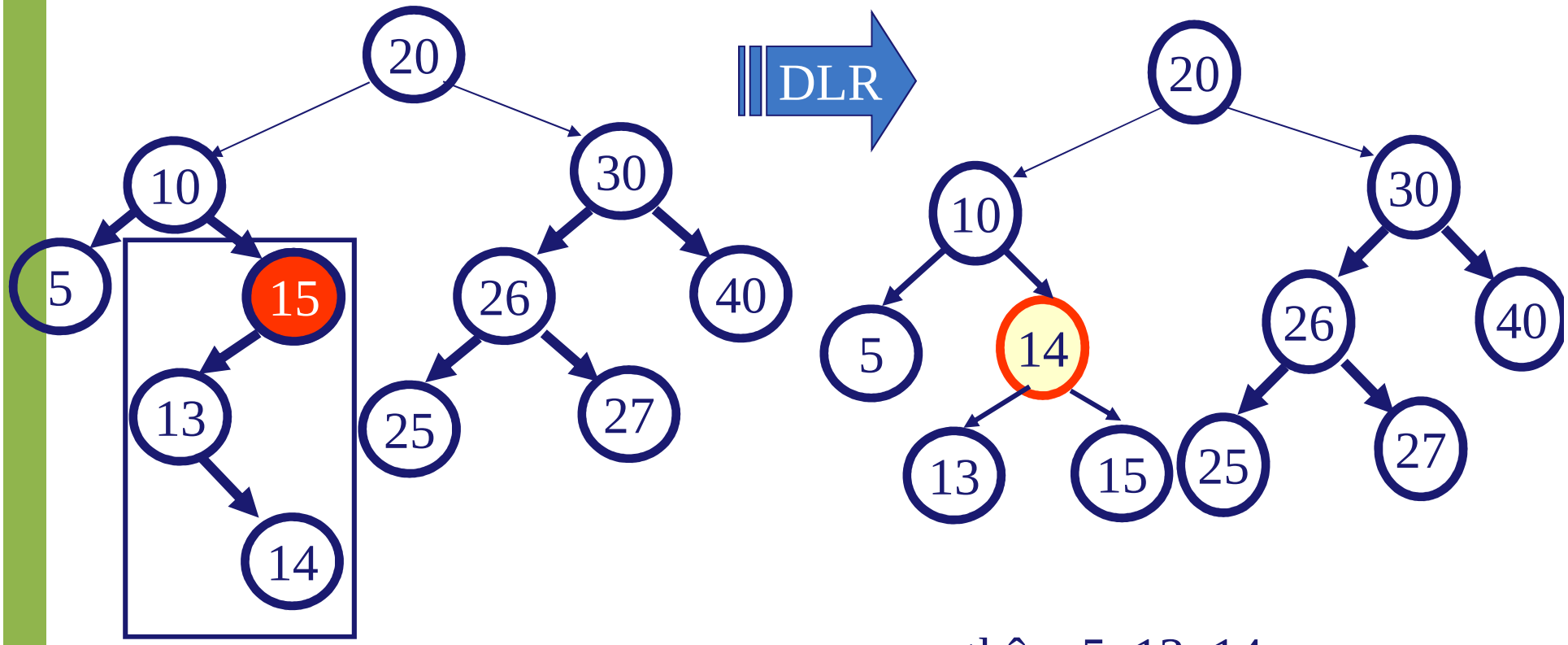


Ví dụ tạo cây AVL [2/3]



…thêm 15, 40, 25, 27, 26

Ví dụ tạo cây AVL [3/3]



…thêm 5, 13, 14

Các đánh giá cây AVL

- Độ cao của cây: $h_{AVL} < 1.44\log_2(N+1)$

  Cây AVL có độ cao nhiều hơn không quá 44% so với độ cao của 1 cây nhị phân tối ưu.

- Chi phí tìm kiếm $O(\log_2 N)$

- Chi phí thêm phần tử $O(\log_2 N)$
  - Tìm kiếm: $O(\log_2 N)$
  - Điều chỉnh cây: $O(\log_2 N)$

- Chi phí xóa phần tử $O(\log_2 N)$
  - Tìm kiếm: $O(\log_2 N)$
  - Điều chỉnh cây: $O(\log_2 N)$