Bài 5
CẤU TRÚC DỮ LIỆU

Trịnh Thành Trung
trungtt@soict.hust.edu.vn

MỞ ĐẦU

- Các bài toán thực tế thường phức tạp
- Hiểu bài toán đặt ra = để giải quyết bài toán, cần làm gì, không cần làm gì. Do đó, phải xác định được:
  - → Các dữ liệu liên quan đến bài toán
  - → Các thao tác cần thiết để giải quyết bài toán

Ví dụ: Bài toán quản lý nhân viên của một cơ quan

- Cần quản lý những thông tin nào ?
  - Thông tin về nhân viên: tên, ngày sinh, số bảo hiểm xã hội, phòng ban làm việc, … → nhân viên ảo
  - …
- Cần thực hiện những thao tác quản lý nào ?
  - Tạo ra hồ sơ cho nhân viên mới vào làm
  - Cập nhật một số thông tin trong hồ sơ
  - Tìm kiếm thông tin về 1 nhân viên
  - …
- Ai được phép thực hiện thao tác nào?

Cấu trúc dữ liệu

- là cách tổ chức và thao tác có hệ thống trên dữ liệu
- Cấu trúc dữ liệu:
  - Mô tả
    - Các dữ liệu cấu thành
    - Mối liên kết về mặt cấu trúc giữa các dữ liệu đó
  - Cung cấp các thao tác trên dữ liệu đó
  - Đặc trưng cho 1 kiểu dữ liệu

Kiểu dữ liệu

- Kiểu dữ liệu cơ bản (primitive data type)
  - Đại diện cho các dữ liệu giống nhau, không thể phân chia nhỏ hơn được nữa
  - Thường được các ngôn ngữ lập trình định nghĩa sẵn
  - Ví dụ:
    - C/C++: int, long, char, boolean, v.v.
    - Thao tác trên các số nguyên: + - * / ...
- Kiểu dữ liệu có cấu trúc (structured data type)
  - Được xây dựng từ các kiểu dữ liệu (cơ bản, có cấu trúc) khác
  - Có thể được các ngôn ngữ lập trình định nghĩa sẵn hoặc do lập trình viên tự định nghĩa

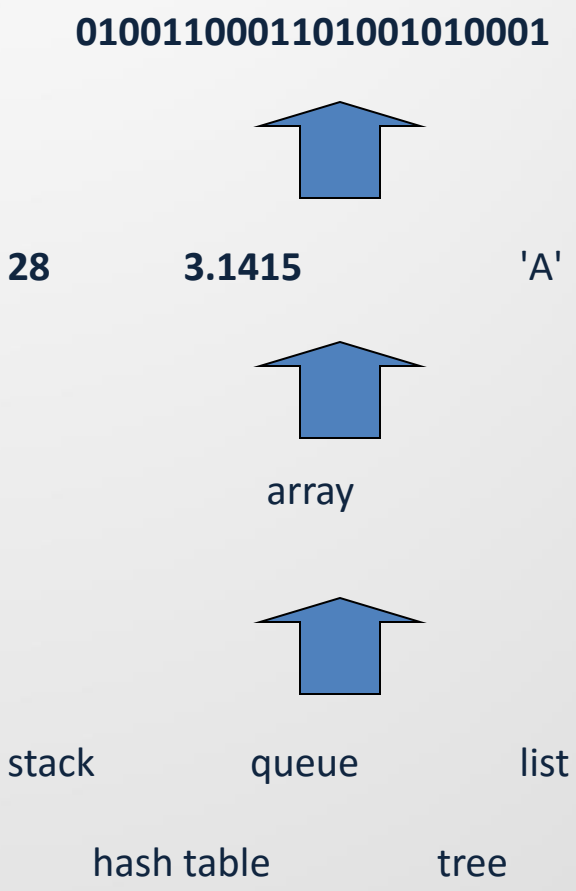
Dữ liệu, kiểu dữ liệu, cấu trúc dữ liệu

Machine Level Data Storage

Primitive Data Types

Basic Data Structures

High-Level Data Structures



CẤU TRÚC DỮ LIỆU

1. Mảng
2. Danh sách
3. Ngăn xếp
4. Hàng đợi
5. Cây

1

DANH SÁCH

Danh sách

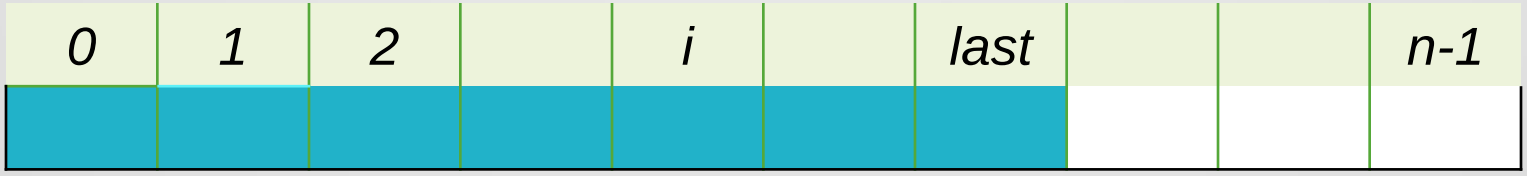
- Danh sách :
  - Tập hợp các phần tử cùng kiểu
  - Số lượng các phần tử của danh sách không cố định
- Phân loại:
  - Danh sách tuyến tính:
    - Có phần tử đầu tiên, phần tử cuối cùng
    - Thứ tự trước / sau của các phần tử được xác định rõ ràng, ví dụ sắp theo thứ tự tăng dần, giảm dần hay thứ tự trong bảng chữ cái
    - Các thao tác trên danh sách phải không làm ảnh hưởng đến trật tự này
  - Danh sách không tuyến tính: các phần tử trong danh sách không được sắp thứ tự
- Có nhiều hình thức lưu trữ danh sách
  - Sử dụng vùng các ô nhớ liên tiếp trong bộ nhớ → danh sách kế tiếp
  - Sử dụng vùng các ô nhớ không liên tiếp trong bộ nhớ → danh sách móc nối
    - Danh sách nối đơn
    - Danh sách nối kép

Danh sách

- Thao tác trên danh sách tuyến tính
  - Khởi tạo danh sách (create)
  - Kiểm tra danh sách rỗng (isEmpty)
  - Kiểm tra danh sách đầy (isFull)
  - Tính kích thước (sizeOf)
  - Xóa rỗng danh sách (clear)
  - Thêm một phần tử vào danh sách tại một ví trí cụ thể (insert)
  - Loại bỏ một phần tử tại một vị trí cụ thể khỏi danh sách (remove)
  - Lấy một phần tử tại một vị trí cụ thể (retrieve)
  - Thay thế giá trị của một phần tử tại một vị trí cụ thể (replace)
  - Duyệt danh sách và thực hiện một thao tác tại các vị trí trong danh sách (traverse)

Danh sách kế tiếp

- Sử dụng một vector lưu trữ gồm một số các ô nhớ liên tiếp để lưu trữ một danh sách tuyến tính
  - Các phần tử liền kề nhau được lưu trữ trong những ô nhớ liền kề nhau
  - Mỗi phần tử của danh sách cũng được gán một chỉ số chỉ thứ tự được lưu trữ trong vector
  - Tham chiếu đến các phần tử sử dụng địa chỉ được tính giống như lưu trữ mảng.



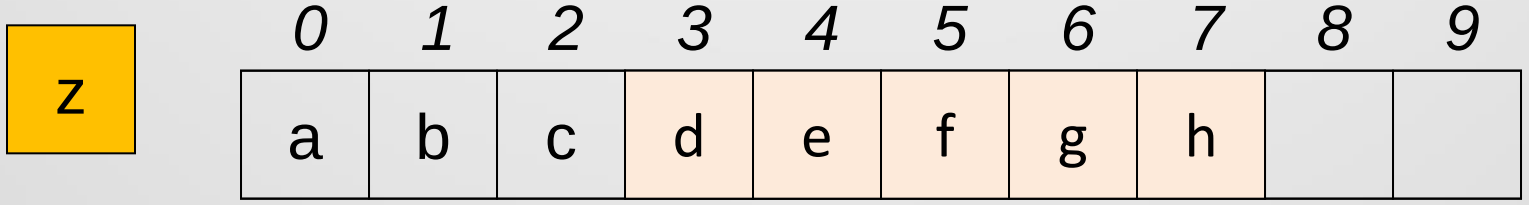
Danh sách kế tiếp

- **Ưu điểm của cách lưu trữ kế tiếp**
  - Tốc độ truy cập vào các phần tử của danh sách nhanh
- **Nhược điểm của cách lưu trữ kế tiếp**
  - Cần phải biết trước kích thước tối đa của danh sách
    - Tại sao?
  - Thực hiện các phép toán bổ sung các phần tử mới và loại bỏ các phần tử cũ khá tốn kém
    - Tại sao?

Thêm một phần tử vào danh sách kế tiếp

- 2 trường hợp
  - insert(index, element): thêm một phần tử element vào một vị trí cụ thể index
  - insert(list, element): thêm một phần tử element vào vị trí bất kỳ trong danh sách list
- Điều kiện tiên quyết:
  - Danh sách phải được khởi tạo rồi
  - Danh sách chưa đầy
  - Phần tử thêm vào chưa có trong danh sách
- Điều kiện hậu nghiệm:
  - Phần tử cần thêm vào có trong danh sách

insert(3, 'z')



Thêm một phần tử vào danh sách kế tiếp

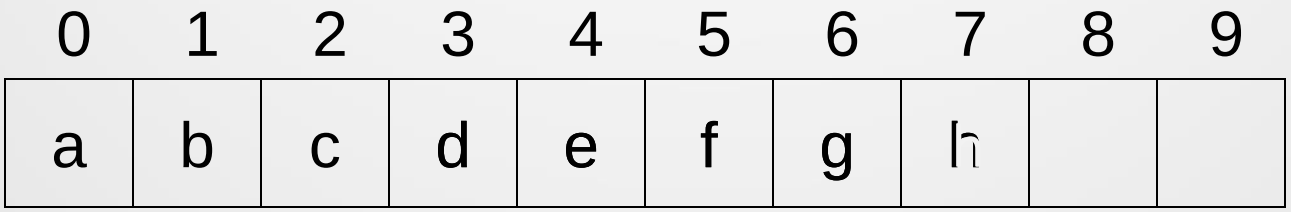
```
Algorithm Insert
    Input: index là vị trí cần thêm vào, element là giá trị cần thêm vào
    Output: tình trạng danh sách

    if list đầy
            return overflow
    if index nằm ngoài khoảng [0..count]
            return range_error
 //Dời tất cả các phần tử từ index về sau 1 vị trí
    for i = count-1 down to index
            entry[i+1] = entry[i]
    entry[index] = element          // Gán element vào vị trí index
    count++                         // Tăng số phần tử lên 1
    return success;

End Insert
```

Xóa một phần tử khỏi danh sách kế tiếp

remove(3, ‘d’)



count=7

Xóa một phần tử khỏi danh sách kế tiếp

```
Algorithm Remove
    Input: index là vị trí cần xóa bỏ, element là giá trị lấy ra được
    Output: danh sách đã xóa bỏ phần tử tại index

    if list rỗng
        return underflow
    if index nằm ngoài khoảng [0..count-1]
        return range_error
    element = entry[index]          //Lấy element tại vị trí index ra
    count--                          //Giảm số phần tử đi 1
    //Dời tất cả các phần tử từ index về trước 1 vị trí
    for i = index to count-1
        entry[i] = entry[i+1]
    return success;

End Remove
```

Duyệt danh sách kế tiếp

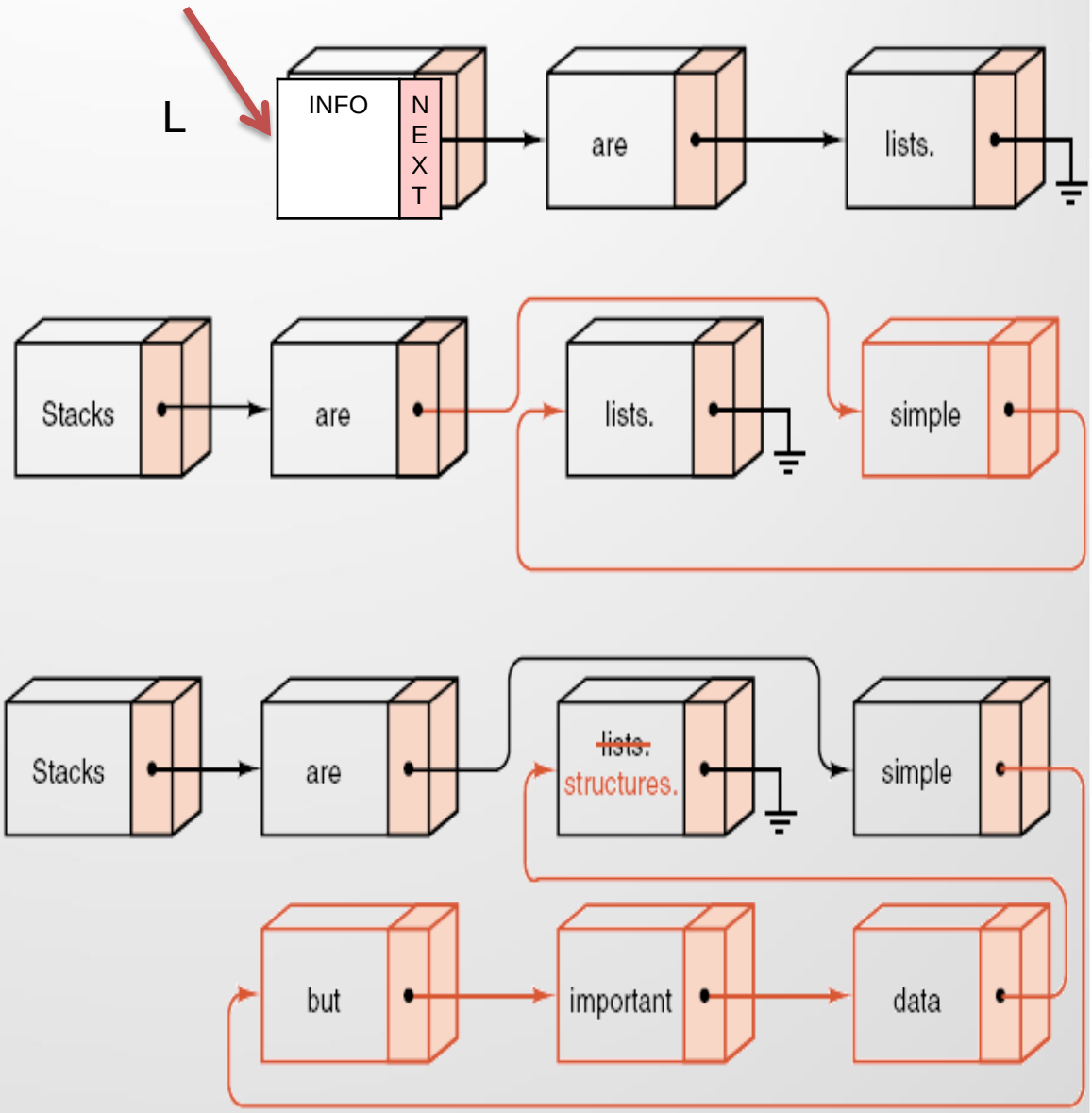
Algorithm Traverse
    Input: hàm visit dùng để tác động vào từng phần tử
    Output: danh sách được cập nhật bằng hàm visit

    //Quét qua tất cả các phần tử trong list
    for index = 0 **to** count-1
        Thi hành hàm visit để duyệt phần tử entry[index]

End Traverse

Danh sách nối đơn

- Một phần tử trong danh sách bằng một nút
- Quy cách của một nút
  - INFO: chứa thông tin (nội dung, giá trị) ứng với phần tử
  - NEXT: chứa địa chỉ của nút tiếp theo
- Để thao tác được trên danh sách, cần nắm được địa chỉ của nút đầu tiên trong danh sách, tức là biết được con trỏ L trỏ tới đầu danh sách



Tổ chức danh sách móc nối

- **Nút = dữ liệu + móc nối**
- **Định nghĩa:**

```
typedef struct hoso
{  ……
};
typedef struct node {
 struct hoso data;
 struct node  *next; } Node;
```

- **Tạo nút mới:**

```
Node  *p = malloc(sizeof(Node));
```

- **Giải phóng nút:**

```
free(p);
```

Khởi tạo và truy cập danh sách móc nối

- Khai báo một con trỏ

  Node *Head;
- Head là con trỏ trỏ đến nút đầu của danh sách.Khi danh sách rỗng thì Head =NULL.
- Tham chiếu đến các thành phần của một nút trỏ bởi p
  - INFO(p)
  - NEXT(p)
- Một số thao tác với danh sách nối đơn
  1. Thêm một nút mới tại vị trí cụ thể
  2. Tìm nút có giá trị cho trước
  3. Xóa một nút có giá trị cho trước
  4. Ghép 2 danh sách nối đơn
  5. Hủy danh sách nối đơn

Truyền danh sách móc nối vào hàm

- **Khi truyền danh sách móc nối vào hàm, chỉ cần truyền Head.**
- **Sử dụng Head để truy cập toàn bộ danh sách**
  - Note: nếu hàm thay đổi vị trí nút đầu của danh sách (thêm hoặc xóa nút đầu) thì Head sẽ không còn trỏ đến đầu danh sách
  - Do đó nên truyền Head theo tham biến (hoặc trả lại một con trỏ mới)

Tìm nút

```
int FindNode(intx)
```

- Tìm nút có giá trị x trong danh sách.
- Nếu tìm được trả lại vị trí của nút.Nếu không, trả lại 0.

```
int FindNode(Node  *head, int x) {
   Node  *currNode = head;
   int currIndex = 1;
   while (currNode && currNode->data != x) {
      currNode = currNode->next;
      currIndex++;
   }
   if (currNode) return currIndex;
   else return 0;
}
```

Thêm một nút mới

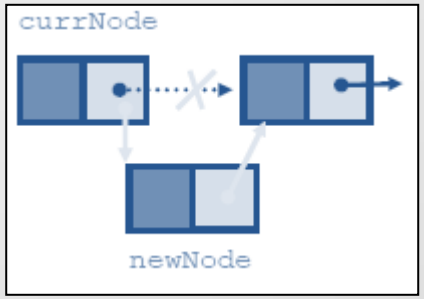
- **Các trường hợp của thêm nút**
  1. Thêm vào danh sách rỗng
  2. Thêm vào đầu danh sách
  3. Thêm vào cuối danh sách
  4. Thêm vào giữa danh sách
- **Thực tế chỉ cần xét 2 trường hợp**
  - Thêm vào đầu danh sách (TH1 và TH2)
  - Thêm vào giữa hoặc cuối danh sách (TH3 và TH4)

Thêm một nút mới

Node *InsertNode(Node *head, int index, int x)

- Thêm một nút mới với dữ liệu là x vào sau nút thứ index (Ví dụ,khi index = 0, nút được thêm là phần tử đầu danh sách; khi index = 1, chèn nút mới vào sau nút đầu tiên,v.v)
- Nếu thao tác thêm thành công,trả lại nút được thêm. Ngược lại,trả lại NULL.
  - (Nếu index < 0 hoặc > độ dài của danh sách, không thêm được.)
- Giải thuật
  1. Tìm nút thứ index – currNode
  2. Tạo nút mới
  3. Móc nối nút mới vào danh sách

```
newNode->next = currNode->next;
currNode->next = newNode;
```



Thêm một nút mới

```
Node  *InsertNode(Node *head,int index,int x)
{
   if (index < 0) return NULL;
   int currIndex = 1;
   Node *currNode = head;
   while(currNode &&  index > currIndex) {
        currNode = currNode->next;
        currIndex++;
   }
   if (index > 0 && currNode== NULL) return NULL;
   Node  *newNode = (Node *) malloc(sizeof(Node));
   newNode->data = x;
   if (index == 0) {
        newNode->next = head;
        head = newNode; }
   else {
        newNode->next = currNode->next;
        currNode->next = newNode; }
   return newNode;
}
```

Tìm nút thứ index, nếu Không tìm được trả về NULL

Tạo nút mới

Thêm vào đầu ds

Thêm vào sau currNode

Xóa nút

```
int DeleteNode(int x)
```

- Xóa nút có giá trị bằng x trong danh sách.
- Nếu tìm thấy nút, trả lại vị trí của nó.

  Nếu không, trả lại 0.
- Giải thuật
  - Tìm nút có giá trị x (tương tự như FindNode)
  - Thiết lập nút trước của nút cần xóa nối đến nút sau của nút cần xóa
  - Giải phóng bộ nhớ cấp phát cho nút cần xóa
  - Giống như InsertNode, có 2 trường hợp
    - Nút cần xóa là nút đầu tiên của danh sách
    - Nút cần xóa nằm ở giữa hoặc cuối danh sách

```
int DeleteNode(Node  *head, int x) {
   Node  *prevNode = NULL;
   Node  *currNode = head;
   int currIndex = 1;
   while (currNode && currNode->data != x) {
        prevNode = currNode;
        currNode = currNode->next;
        currIndex++;
   }
   if (currNode) {
        if (prevNode) {
              prevNode->next = currNode->next;
              free (currNode);
        } else {
              head = currNode->next;
              free (currNode);
        }
        return currIndex;
   }
   return 0;
}
```

Tìm nút có giá trị = x

Xóa nút ở giữa

Xóa nút head

Hủy danh sách

```
void DestroyList(Node  *head)
```

- Dùng để giải phóng bộ nhớ được cấp phát cho danh sách.
- Duyệt toàn bộ danh sách và xóa lần lượt từng nút.

```
void DestroyList(Node* head){
  Node  *currNode = head,   *nextNode= NULL;
  while(currNode !=  NULL){
   nextNode = currNode->next;
   free(currNode); // giải phóng nút vừa duyệt
   currNode = nextNode;
  }
}
```

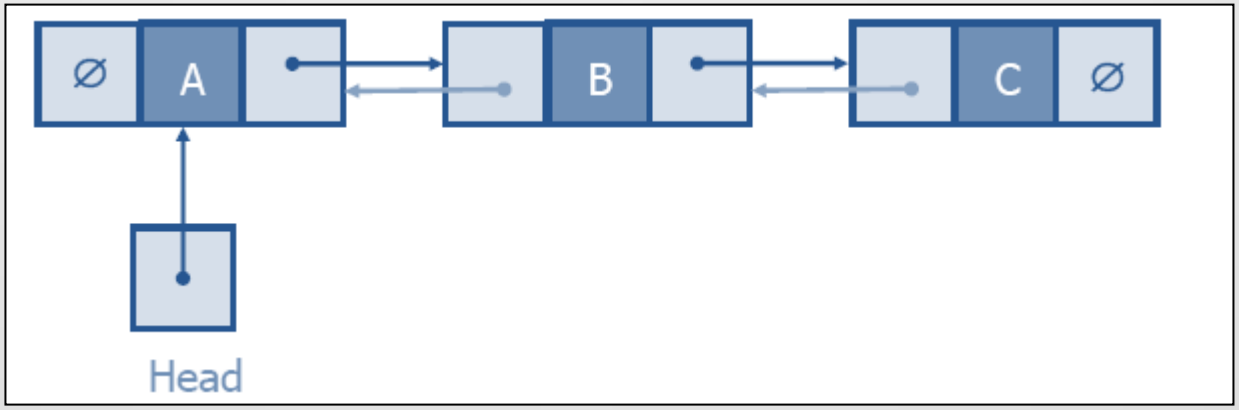
So sánh mảng và danh sách liên kết

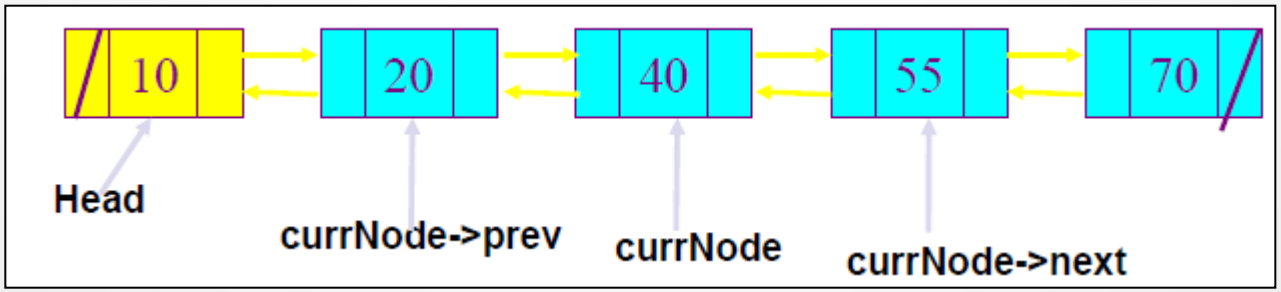
Việc lập trình và quản lý danh sách liên kết khó hơn mảng, nhưng nó có những ưu điểm:

- **Linh động**: danh sách liên kết có kích thước tăng hoặc giảm rất linh động.
  - Không cần biết trước có bao nhiêu nút trong danh sách.Tạo nút mới khi cần.
  - Ngược lại,kích thước của mảng là cố định tại thời gian biên dịch chương trình.
- **Thao tác thêm và xóa dễ dàng**
  - Để thêm và xóa một phần tử mảng, cần phải copy dịch chuyển phần tử.
  - Với danh sách móc nối, không cần dịch chuyển mà chỉ cần thay đổi các móc nối

Danh sách nối kép

- Mỗi nút không chỉ nối đến nút tiếp theo mà còn nối đến nút trước nó
- Có 2 mối nối NULL: tại nút đầu và nút cuối của danh sách
- Ưu điểm: tại một nút có thể thăm nút trước nó một cách dễ dàng. Cho phép duyệt danh sách theo chiều ngược lại



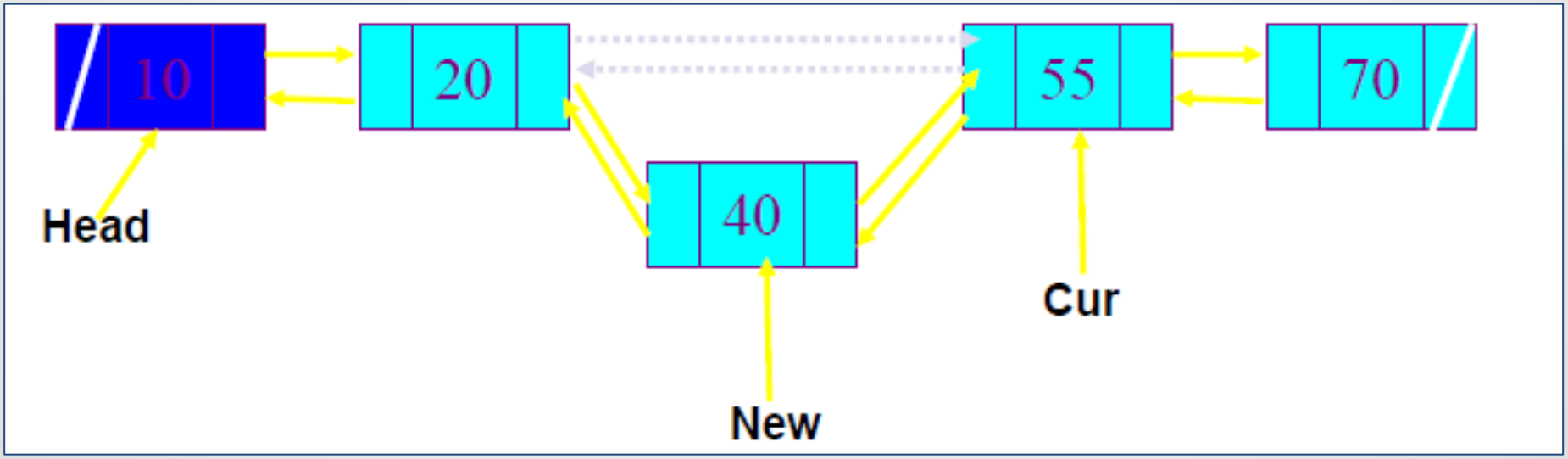


- Mỗi nút có 2 mối nối
  - prev nối đến phần tử trước
  - next nối đến phần tử sau

```
typedef struct Node{
  int data;
  struct Node *next;
  struct Node *prev;
} Node;
```

- Thêm nút New nằm ngay trước Cur (không phải nút đầu hoặc cuối danh sách)

```
New->next = Cur;
New->prev= Cur->prev;
Cur->prev= New;
(New->prev)->next = New;
```

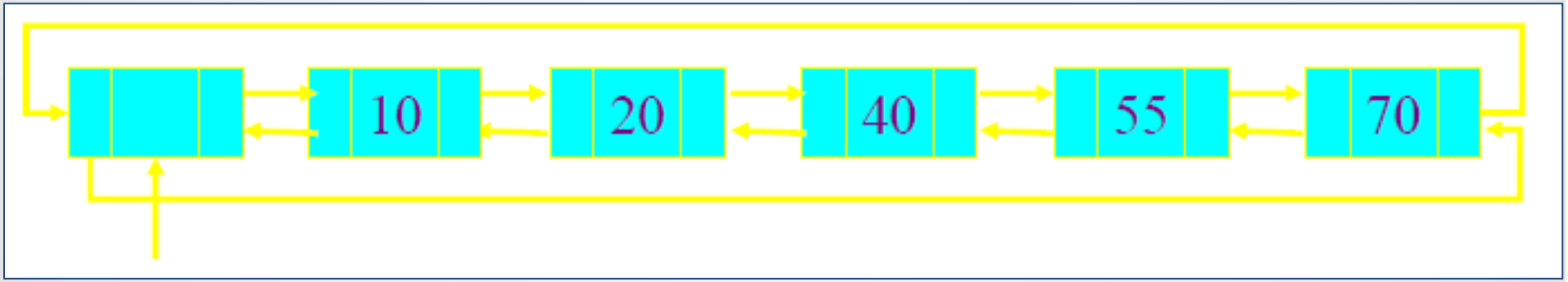


- Xóa nút Cur(không phải nút đầu hoặc cuối danh sách)

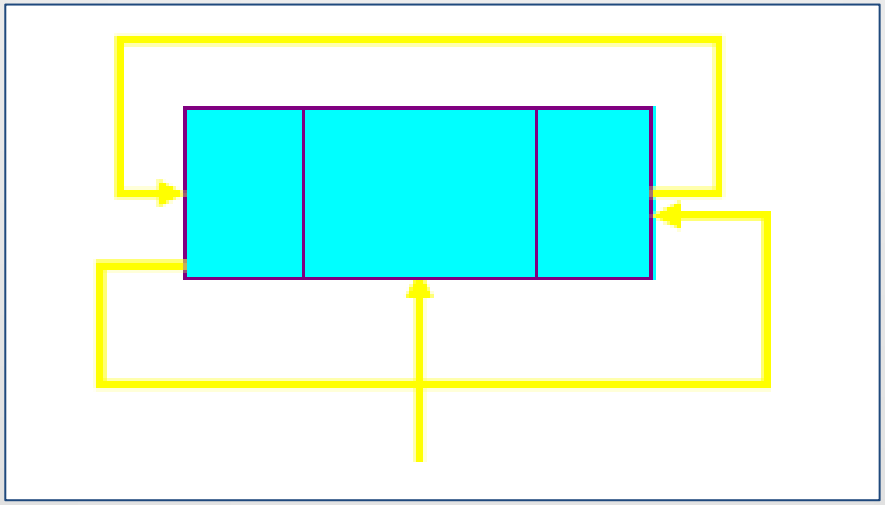
```
(Cur->prev)->next = Cur->next;
(Cur->next)->prev = Cur->prev;
free (Cur);
```

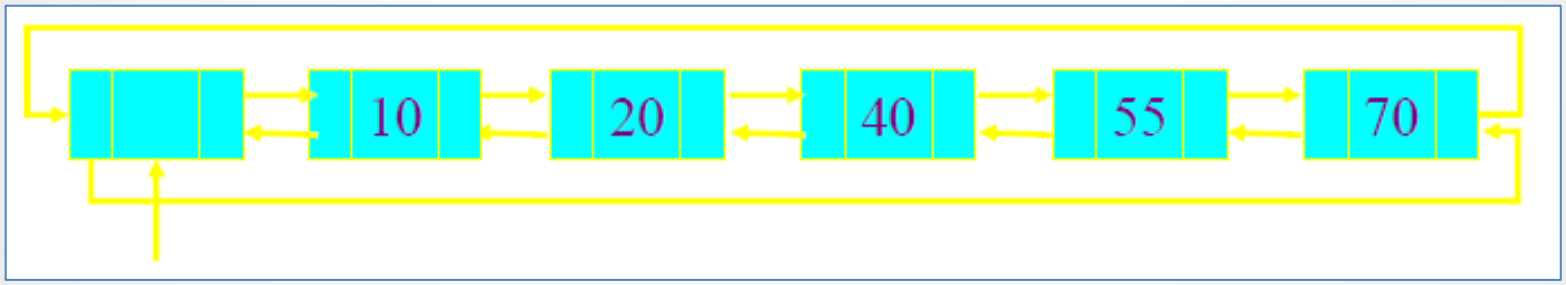
Danh sách nối kép với nút đầu giả

- **Danh sách không rỗng**



- **Danh sách rỗng**





- Tạo danh sách nối kép rỗng
  - Node *Head = malloc (sizeof(Node));
  - Head->next = Head;
  - Head->prev = Head;
- Khi thêm hoặc xóa các nút tại đầu, giữa hay cuối danh sách?

Xóa nút

```
void deleteNode(Node *Head, int x){
  Node *Cur;
  Cur = FindNode(Head, x);
  if (Cur != NULL){
   Cur->prev->next = Cur->next;
   Cur->next->prev= Cur->prev;
   free(Cur);
  }
}
```

Thêm nút

```
void insertNode(Node *Head, int item) {
  Node  *New, *Cur;
  New = malloc(sizeof(Node));
  New->data = item;
  Cur = Head->next;
  while (Cur != Head){
   if (Cur->data < item)
          Cur = Cur->next;
   else
          break;
  }
  New->next = Cur;
  New->prev= Cur->prev;
  Cur->prev= New;
  (New->prev)->next = New;
}
```

Bài tập

Sử dụng danh sách móc nối kép với nút đầu giả, xây dựng bài toán quản lý điểm SV đơn giản, với các chức năng sau :

1. Nhập dữ liệu vào danh sách
2. Hiển thị dữ liệu 1 lớp theo thứ tự tên
3. Sắp xếp dữ liệu
4. Tìm kiếm kết quả

Với thông tin về mỗi sinh viên được định nghĩa trong cấu trúc sau

```
typedef struct {
    int masv;  // mã hiệu sv
    char malop[12];
    char ho[30];
    char ten[30];
    float diemk1;
    float diemk2;
} sinhvien
```

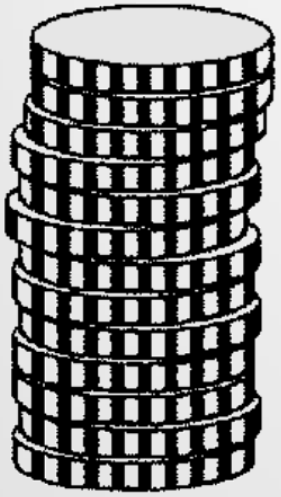
2

NGĂN XẾP

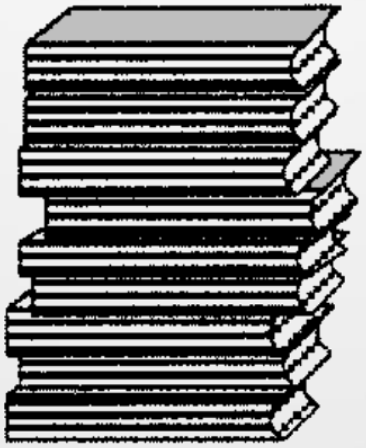
Ngăn xếp

- Hai danh sách tuyến tính đặcbiệt:
  - Ngăn xếp –Stack
  - Hàng đợi –Queue
- Stack: là danh sách mà xóa và thêm phần tử bắt buộc phải cùng được thực hiện tại một đầu duy nhất (đỉnh)

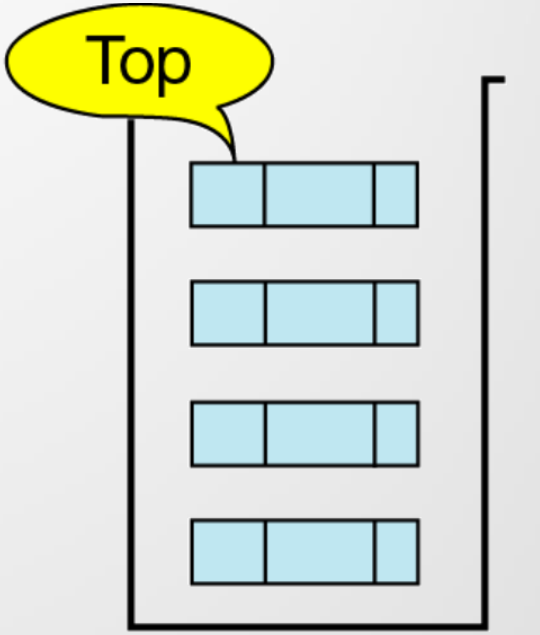
Ví dụ về Stack trong thực tế



Stack of coins



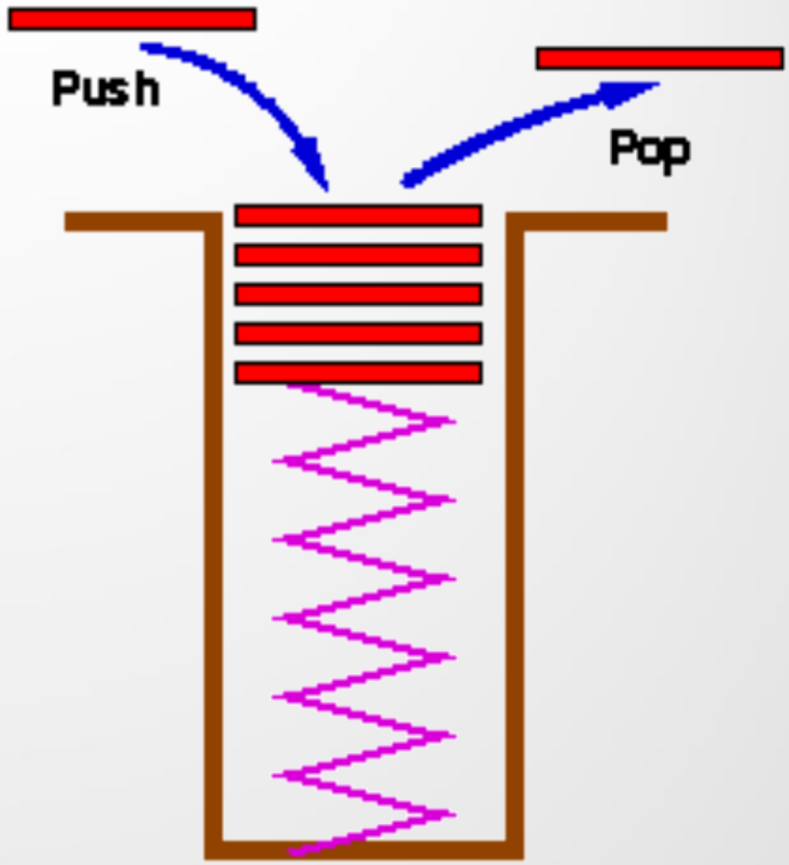
Stack of books



Computer stack

Stack là một cấu trúc LIFO: Last In First Out

- Push
  - Thêm một phần tử
  - Tràn (overflow)
- Pop
  - Xóa một phần tử
  - Underflow
- Top
  - Phần tử đỉnh
  - stack rỗng
- Kiểm tra rỗng/đầy

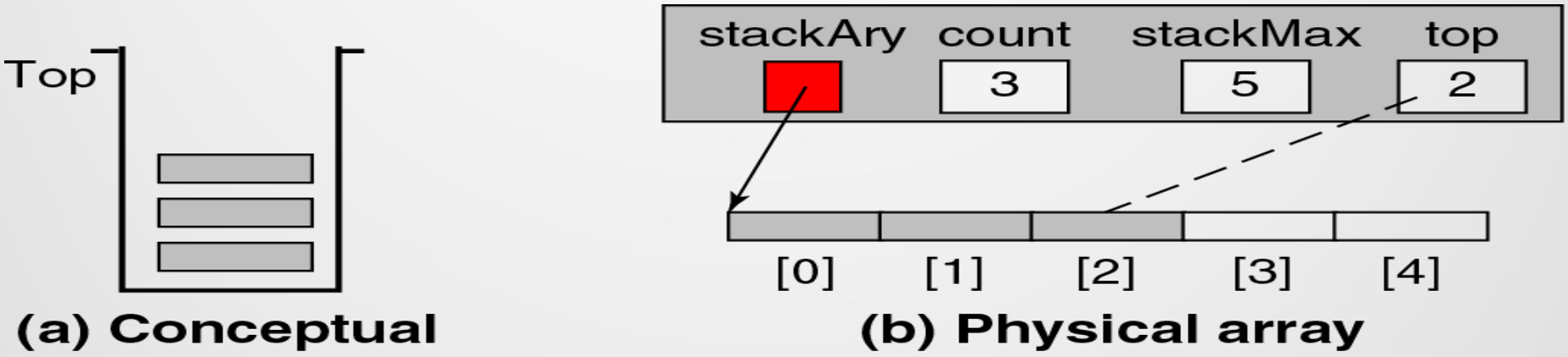


Các thao tác cơ bản trên Stack

Lưu trữ Stack

2 cách lưu trữ:

- Lưu trữ kế tiếp: sử dụng mảng



- Lưu trữ móc nối: sử dụng danh sách móc nối (sau)

Cấu trúc dữ liệu

```
/* Stack của các số nguyên: intstack*/
typedef struct intstack{
    int *stackArr; /* mảng lưu trữ các phần tử */
    int count;     /* số ptử hiện có của stack */
    int stackMax;  /* giới hạn Max của số phần tử */
    int top;       /* chỉ số của phần tử đỉnh */
} intStack;
```

Tạo Stack

```
IntStack *CreateStack(int max){
	IntStack *stack;
	stack =(IntStack *) malloc(sizeof(IntStack));
	if (stack == NULL)
		return NULL;
	/*Khởi tạo stack rỗng*/
	stack->top = -1;
	stack->count = 0;
	stack->stackMax= max;
	stack->stackArr=malloc(max * sizeof(int));
	return stack ;
}
```

Push

```
int PushStack(IntStack *stack, int dataIn) {
    /*Kiểm tra tràn*/
    if(stack->count == stack->stackMax)
        return 0;
    /*Thêm phần tử vào stack */
    (stack->count)++;
    (stack->top)++; /* Tăng đỉnh */
    stack->stackArr[stack->top] =dataIn;
    return 1;
}
```

Pop

```
int PopStack(IntStack *stack, int *dataOut){
    /* Kiểm tra stack rỗng */
    if(stack->count == 0)
        return 0;
    /* Lấy giá trị phần tử bị loại*/
    *dataOut=stack->stackArr[stack->top];
    (stack->count)--;
    (stack->top)--; /* Giảm đỉnh */
    return 1;
}
```

Top

```
int TopStack(IntStack *stack, int *dataOut){
   if(stack->count ==0) // Stack rỗng
       return 0;
   *dataOut = stack->stackArr[stack->top];
   return 1;
}
```

Kiểm tra rỗng?

```
int IsEmptyStack(IntStack *stack){
    return(stack->count == 0);
}
```

Kiểm tra đầy?

```
int IsFullStack(IntStack *stack) {
    return(stack->count==stack->stackMax);
}
```

Ứng dụng của Stack

Bài toán đổi cơ số: Chuyển một số từ hệ thập phân sang hệ cơ số bất kỳ

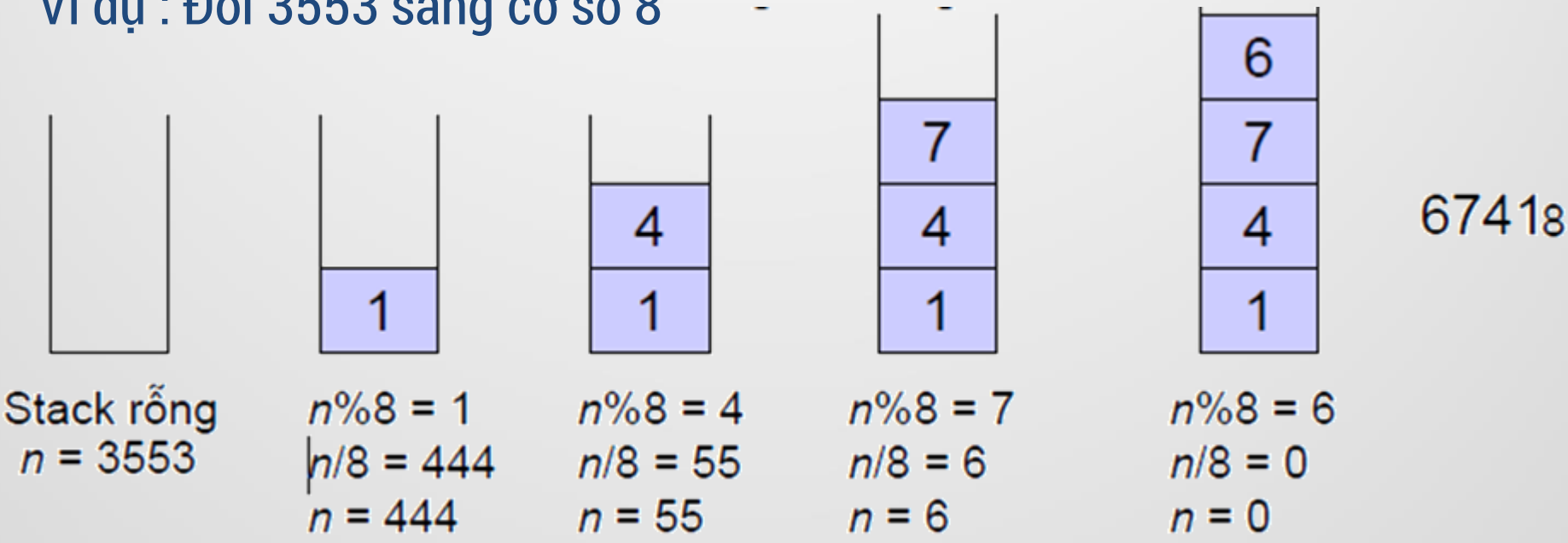
- (base 8) $28_{10} = 3 \cdot 8^1 + 4 \cdot 8^0 = 34_8$
- (base 4) $72_{10} = 1 \cdot 4^3 + 0 \cdot 4^2 + 2 \cdot 4^1 + 0 \cdot 4^0 = 1020_4$
- (base 2) $53_{10} = 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 110101_2$

Đầu vào số thập phân *n*, cơ số *b*

Đầu ra số hệ cơ số *b* tương đương

1. Chữ số bên phải nhất của kết quả=*n* % *b*. Đẩy vào Stack.
2. Thay *n*= *n* / *b* (để tìm các số tiếp theo).
3. Lặp lại bước 1-2 cho đến khi n = 0.
4. Rút lần lượt các chữ số lưu trong Stack, chuyển sang dạng ký tự tương ứng với hệ cơ số trước khi in ra kết quả

Ví dụ : Đổi 3553 sang cơ số 8



Chuyển sang dạng ký tự tương ứng:

```
char *digitChar= “0123456789ABCDEF”;
char d = digitChar[13]; // 13_10= D_16
char f = digitChar[15]; // 13_10= F_16
```

Đổi cơ số

```
void DoiCoSo(int n,int b) {
    char*digitChar= "0123456789ABCDEF“;
    //Tạo một stack lưu trữ kết quả
    IntStack *stack = CreateStack(MAX);
    do{
        //Tính chữ số bên phải nhất,đẩy vào stack
        PushStack(stack, n% b);
        n/= b; //Thay n = n/b để tính tiếp
    } while(n!= 0);  //Lặp đến khi n = 0
    while( !IsEmptyStack(stack) ){
       // Rút lần lượt từng phần tửcủa stack
       PopStack(stack, &n);
        // chuyển sang dạng ký tự và in kết quả
        printf(“%c”, digitChar[n]);
    }
}
```

Ứng dụng của Stack (tiếp)

Các biểu thức số học được biểu diễn bằng ký pháp trung tố. Với phép toán 2 ngôi: Mỗi toán tử được đặt giữa hai toán hạng. Với phép toán một ngôi: Toán tử được đặt trước toán hạng: vd

-2 + 3 * 5 <=> (-2) + (3 * 5)

- Thứ tự ưu tiên của các phép tử:

  () > ^ > * = % = / > + = –

- Việc đánh giá biểu thức trung tố khá phức tạp

Ký pháp hậu tố

Là giải pháp thay thế ký pháp trung tố, trong đó : Toán hạng đặt *trước* toán tử, Không cần dùng các dấu ().

Ví dụ :

a*b*c*d*e*f => ab*c*d*e*f*

1 + (-5) / (6 * (7+8)) => 1 5 -6 7 8 + * / +

(x/y−a*b) * ((b+x) −y) => x y / a b * −b x + y y ^ −*

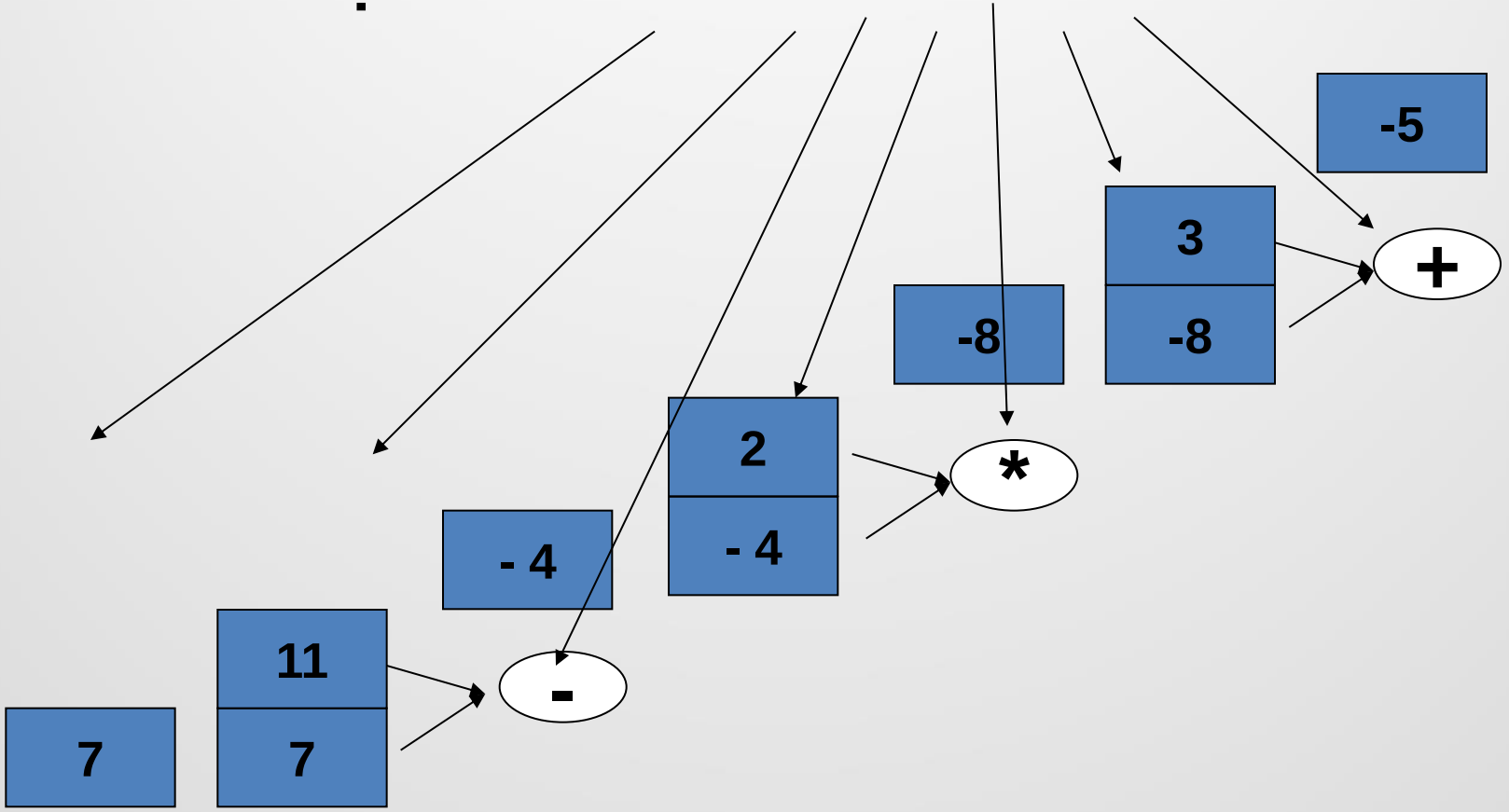
(x*y*z −x^2 / (y*2 −z^3) + 1/z) * (x −y) =>

xy*z*x2^y2*z3^ −/ −1z/+xy −*

Tính giá trị biểu thức hậu tố

Biểu thức trung tố: (7 –11) * 2 + 3

Biểu thức hậu tố: 7 11 – 2 * 3 +



Tính giá trị của biểu thức hậu tố

- **Tính giá trị của một một biểu thức hậu tố được lưu trong một xâu ký tự và trả về giá trị kết quả**
- **Với :**
  - Toán hạng: Là các số nguyên không âm một chữ số (cho đơn giản)
  - Toán tử: **+ , - , * , / , % , ^**

```
bool isOperator(char op) {
     return        op ==  '+‘  ||  op ==  '-'  ||
                   op ==  '*‘  ||  op ==  '%‘  ||
                   op ==  '/‘  ||  op ==  '^‘ ;
}

int compute(int left, int right, char op) {
    int value;
    switch(op){
          case '+‘ : value = left + right;     break;
          case '-‘  : value = left - right;    break;
          case '*‘  : value = left * right;    break;
          case '%': value = left % right;      break;
          case '/‘  : value = left / right;    break;
          case '^‘ : value = pow(left, right);
    }
    return value;
}
```

```
int TinhBtHauTo(string Bt) {
          Int left, right, kq;
          char ch;
          IntStack *stack = CreateStack(MAX);
          for(int i=0; i < Bt.length(); i++)
          {
                    ch = Bt[i];
                    if ( isdigit(ch) )
                              PushStack(stack, ch-'0'); // đẩy toán hạng vào stack
                    else if (isOperator(ch)) {
                              // rút stack 2 lần để lấy 2  toán hạng left và right
                              PopStack(stack, &right);
                              PopStack(stack, &left);
                              kq =compute(left, right, ch); // Tính "leftop right"
                              PushStack(stack, kq);          // Đẩy kq vào stack
                    } else //không phải toán hạng hoặc toán tử
                    printf(“Bieu thuc loi”);
          }
          // Kết thúc tính toán, giá trị biểu thức  nằm trên đỉnh stack, đưa vào kq
          PopStack(stack, kq);
          Return  kq;
}
```

Bài tập

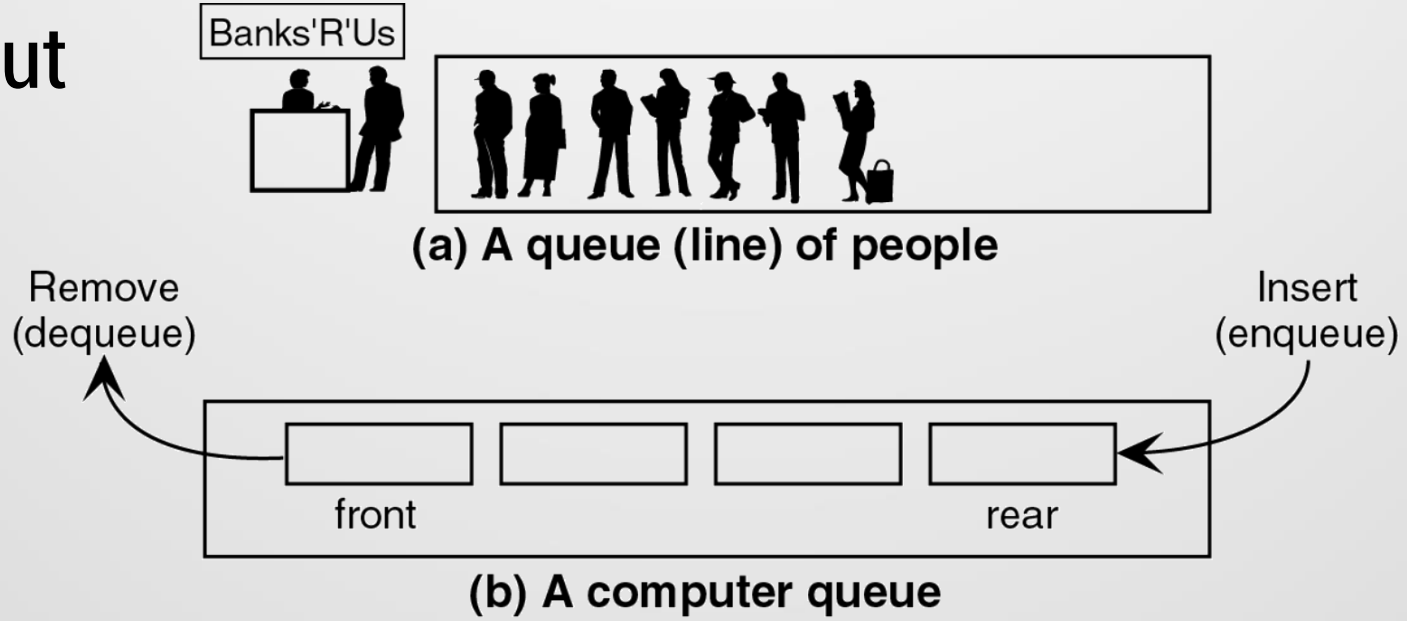
- Sửa chương trình trên để tính toán kết quả của 1 biểu thức hậu tố với các toán hạng tổng quát (có thể là số thực, có thể âm ...)
- Xây dựng chương trình chuyển đổi 1 biểu thức từ trung tố sang hậu tố, biểu thức trung tố là 1 xâu ký tự với các toán hạng tổng quát và các phép toán cùng độ ưu tiên như sau : () > ^ > * = % = / > + = –

3

HÀNG ĐỢI

Hàng đợi

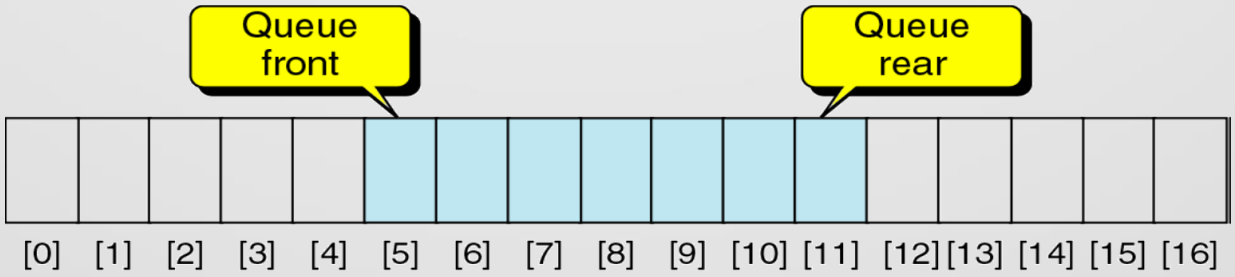
- Hàng đợi – Queue: là danh sách mà thêm phải được thực hiện tại một đầu còn xóa phải thực hiện tại đầu kia.
- Queue là một kiểu cấu trúc FIFO: First In First Out



- Phần tử đầu hàng sẽ được phục trước, phần tử này được gọi là **front**, hay **head** của hàng. Tương tự, phần tử cuối hàng , cũng là phần tử vừa được thêm vào hàng, được gọi là **rear** hay **tail** của hàng.

Các phương án thực hiện hàng

- **Mô hình vật lý**
  - Có thể dùng 1 mảng. Tuy nhiên, cần phải nắm giữ cả **front** và **rear**.
  - Một cách đơn giản là ta luôn giữ front luôn là vị trí đầu của dãy. Lúc đó nếu thêm phần tử vào hàng ta chỉ việc thêm vào cuối dãy. Nhưng nếu lấy ra 1 phần tử ta phải dịch chuyển tất cả các phần tử của dãy lên 1 vị trí.
  - Mặc dù cách làm này rất giống với hình ảnh hàng đợi trong thực tế, nhưng lại là 1 lựa chọn rất dở với máy tính
- **Hiện thực tuyến tính**
  - Ta dùng 2 chỉ số Front và Rear để lưu trữ đầu và cuối hàng mà không di chuyển các phần tử.
  - Khi thêm ta chỉ việc tăng rear lên 1 và thêm phần tử vào vị trí đó
  - Khi rút phần tử ra, ta lấy phần tử tại front và tăng front lên 1
  - Nhược điểm: front và rear chỉ tăng mà không giảm => lãng phí bộ nhớ
  - Có thể cải tiến bằng cách khi hàng đợi rỗng thì ta gán lại front=rear= đầu dãy

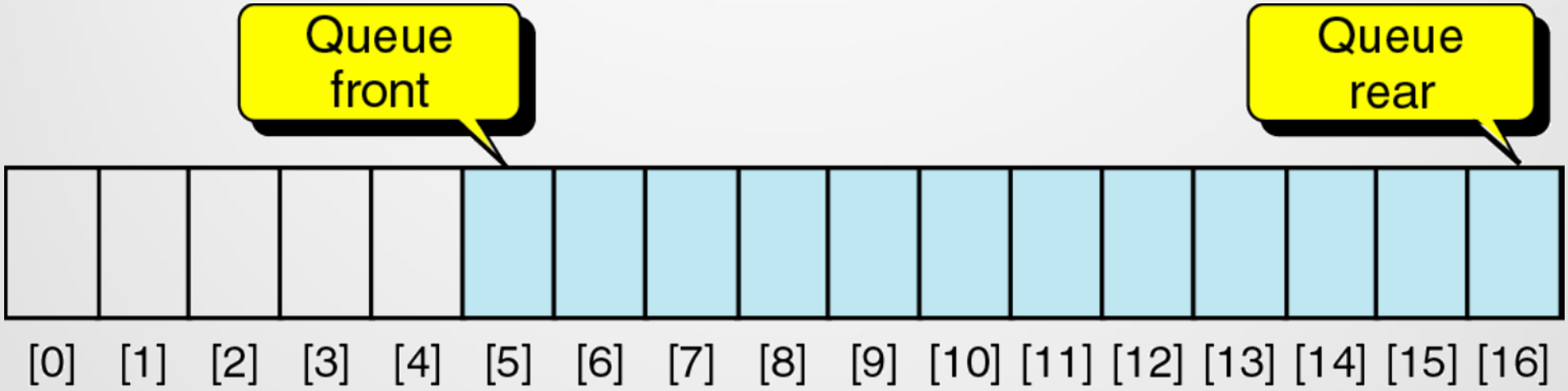


- Hiện thực của dãy vòng
  - Ta dùng 1 dãy tuyến tính để mô phỏng 1 dãy vòng.
  - Các vị trí trong vòng tròn được đánh số từ 0 đến max-1, trong đó max là tổng số phần tử
  - Để thực hiện dãy vòng, chúng ta cũng sử dụng các phân tử được đánh số tương tư dãy tuyến tính.
  - Sự thay đổi các chỉ số chỉ đơn giản là phép lấy phần dư số học: khi một chỉ số vợt quá max-1, nó đc bắt đầu trở lại vợi trị 0. Điều này tương tự với việc cộng thêm giờ trên đồng hồ mặt tròn

    i = ((i+1) == max) ? 0: (i+1);

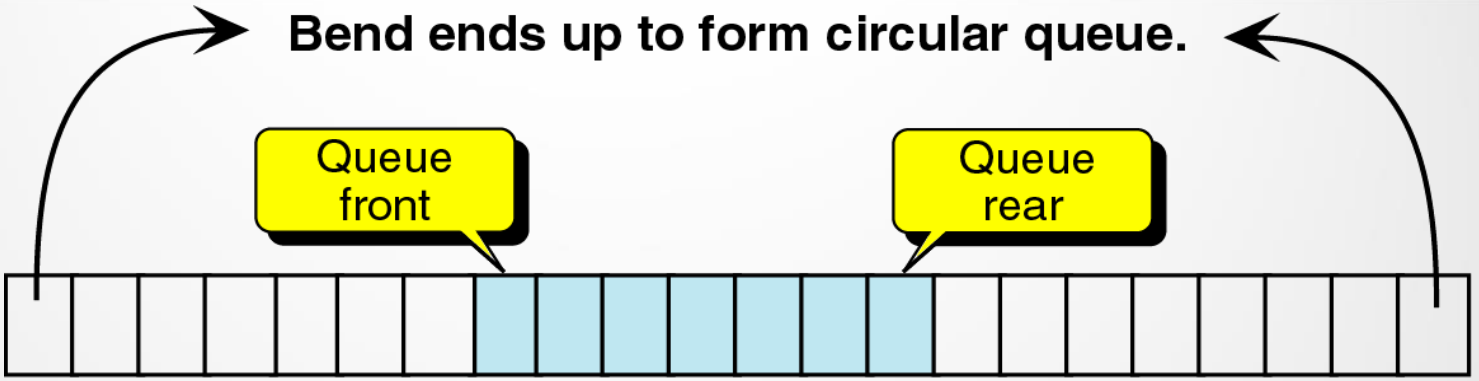
    Hoặc if ((i+1) == max) i = 0; else i = i+1;

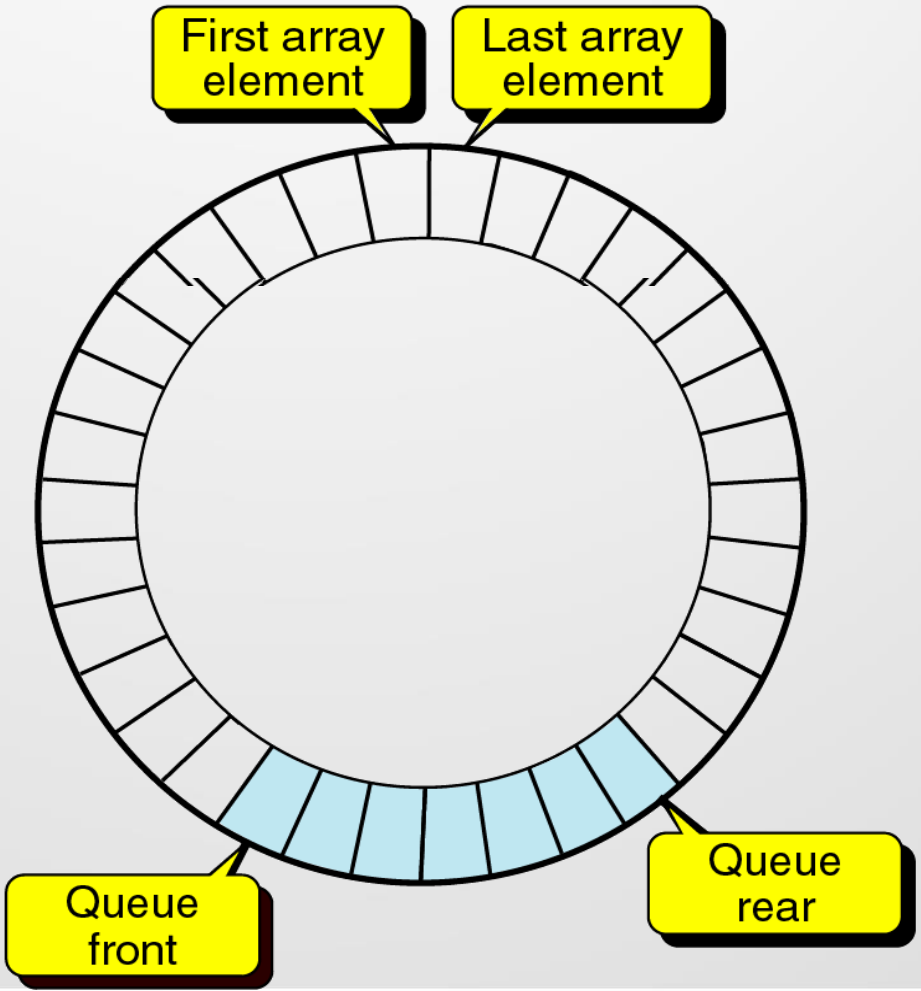
    Hoặc i = (i+1) % max;



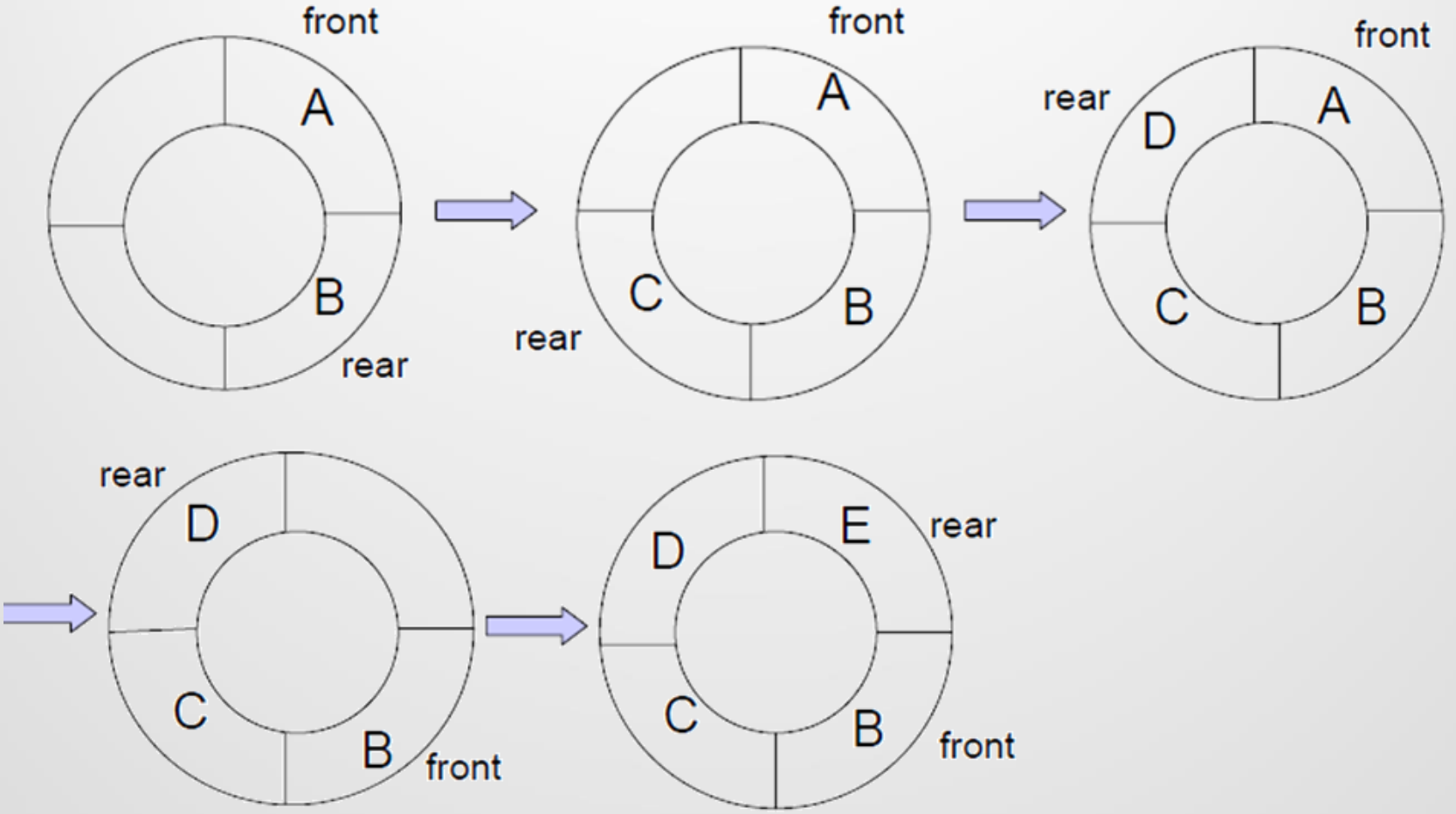
- Phaỉ xử dụng mảng với kích thước lớn

Queue tăng hết mảng

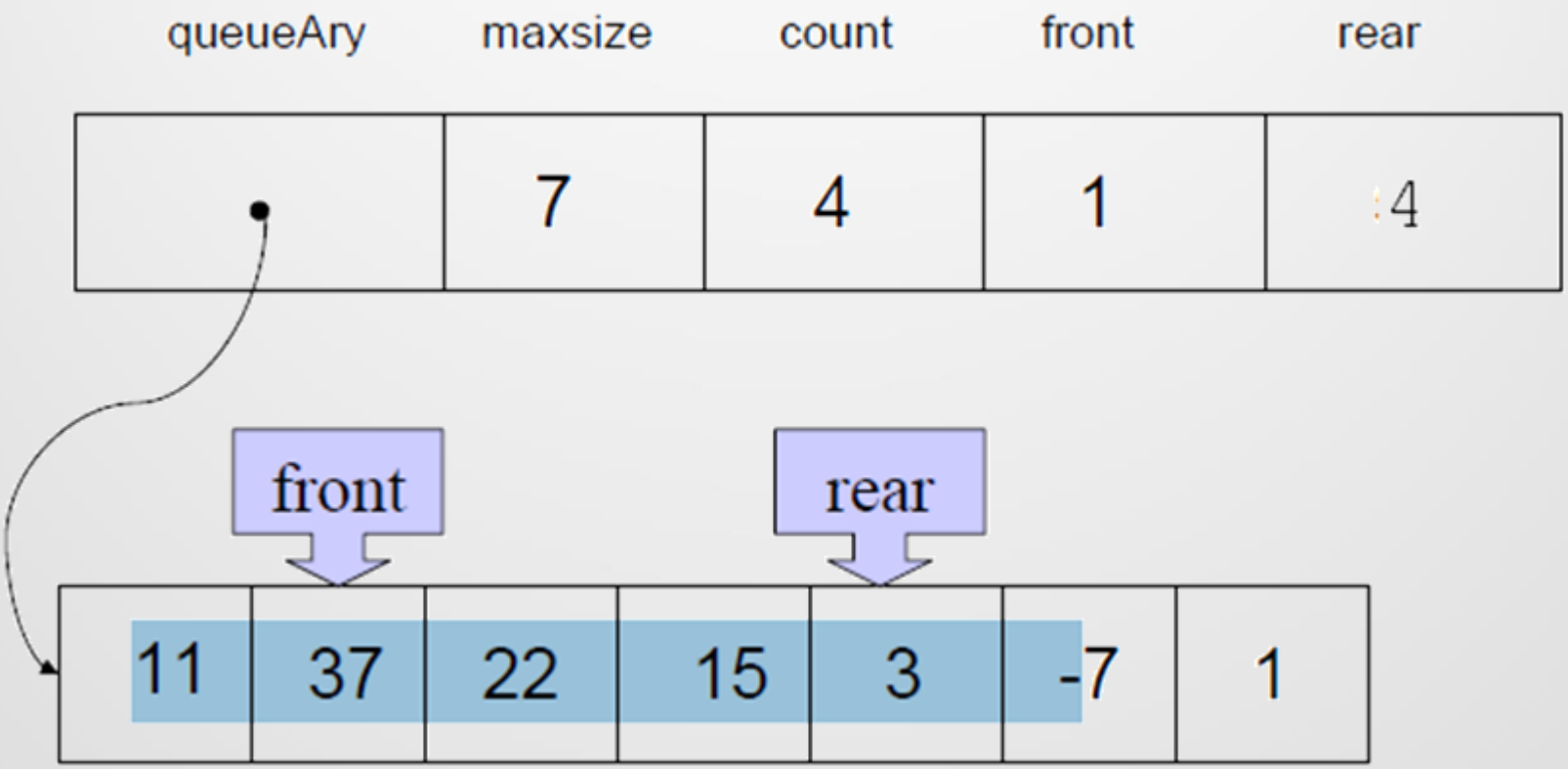

Bend ends up to form circular queue.
Queue front
Queue rear


First array element
Last array element
Queue front
Queue rear

Queue dạng vòng

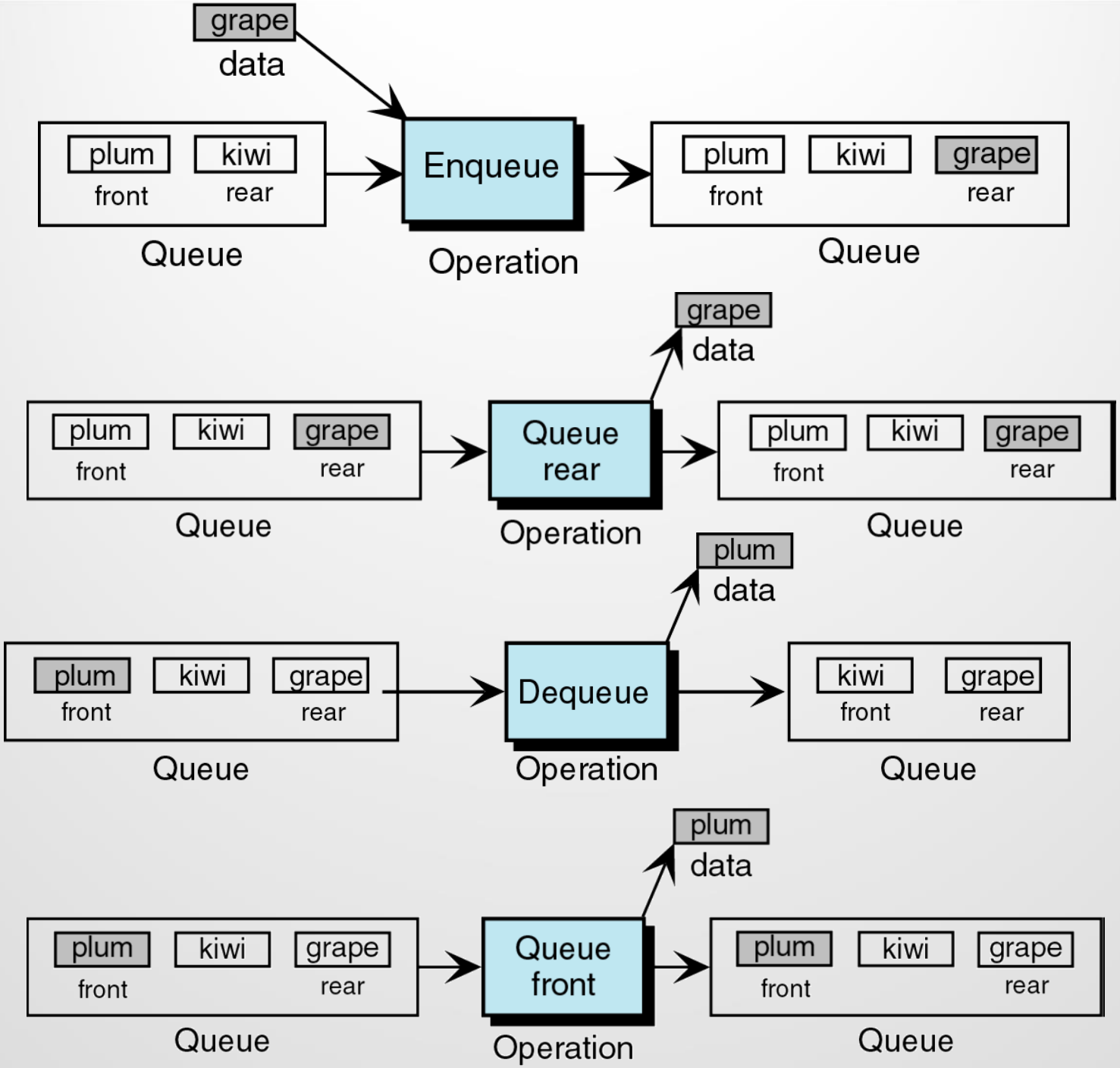


Queue thực hiện trên mảng



Các thao tác cơ bản với Queue

- Enqueue–Thêm một phần tử vào cuối queue
  - Tràn Overflow?
- Dequeue–Xóa một phần tử tại đầu queue
  - Queue rỗng?
- Front –Trả lại phần tử tại đầu queue
  - Queue rỗng?
- Rear –Trả lại phần tử tại cuối queue
  - Queue rỗng


grape
data
plum
kiwi
front
rear
Queue
Enqueue
Operation
plum
kiwi
grape
front
rear
Queue
grape
data
plum
kiwi
grape
front
rear
Queue
Queue
rear
Operation
plum
kiwi
grape
front
rear
Queue
plum
data
plum
kiwi
grape
front
rear
Queue
Dequeue
Operation
kiwi
grape
front
rear
Queue
plum
data
plum
kiwi
grape
front
rear
Queue
Queue
front
Operation
plum
kiwi
grape
front
rear
Queue

Định nghĩa cấu trúc Queue

```
typedef struct intqueue{
     int *queueArr;
     int maxSize;
     int count;
     int front;
     int rear;
} IntQueue;
```

Tạo Queue

```
IntQueue *CreateQueue(int max){
       IntQueue  *queue;
       queue = (IntQueue *)malloc(sizeof(IntQueue));
       /*Cấp phát cho mảng */
      queue->queueAry= malloc(max *sizeof(int));
       /* Khởi tạo queue rỗng */
       queue->front = -1;
       queue->rear = -1;
       queue->count = 0;
       queue->maxSize= maxSize;
       return  queue;
} /* createQueue*/
```

Thêm vào cuối Queue

- **Enqueue**

```
int enqueue(struct  intqueue *queue, int datain)
{
    if (queue->count >= queue->maxSize)
        return 0;
    (queue->count)++;
    queue->rear = (queue->rear + 1) % queue->maxSize;
    queue->queueAry[queue->rear] = datain;
    return 1;
}
```

Xóa ở đầu queue

- **Dequeue**

```
int dequeue(struct intqueue *queue, int
*dOutPtr)
   {if(!queue->count)
         return 0;
   *dOutPtr= queue->queueAry[queue->front];
    (queue->count)--;
    queue->front = (queue->front + 1) %
queue->maxSize;
   return 1;
}
```

Lấy phần tử đầu queue

- **Front**

```
int Front(struct intqueue *queue,int
*dOutPtr) {
    if(!queue->count)
          return 0;
    else{
    *dOutPtr= queue->queueAry[queue->front];
          return 1;
    }
}
```

Lấy phần tử cuối queue

- **Rear**

```
int Rear(struct intqueue *queue,int*dOutPtr)
{
   if(!queue->count)
     return 0;
   else{
     *daOutPtr= queue->queueAry[queue->rear];
     return 1;
   }
}
```

Kiểm tra empty và full

- **Empty**

```
int emptyQueue(struct intqueue *queue)
{
   return(queue->count == 0);
}/* emptyQueue*/
```

- **Full**

```
int fullQueue(struct intqueue *queue)
{
    return( queue->count == queue->maxSize);
}/* fullQueue*/
```

Xóa queue

- Destroy

```
struct intqueue *destroyQueue(struct intqueue
*queue)
{
   if(queue)
   {
        free(queue->queueAry);
        free(queue);
   }
   return NULL;
}/* destroyQueue*/
```

Bài tập

- Xây dựng Stack và Queue móc nối, cài đặt các thao tác tương ứng

4

CÂY

Cây

1. Định nghĩa và khái niệm
2. Cây nhị phân
   - Định nghĩa vàTính chất
   - Lưu trữ
   - Duyệt cây
3. Cây tổng quát
   - Biểu diễn cây tổng quát
   - Duyệt cây tổng quát (nói qua)
4. Ứng dụng của cấu trúc cây
   - Cây biểu diễn biểu thức (tính giá trị, tính đạo hàm)
   - Cây quyết định

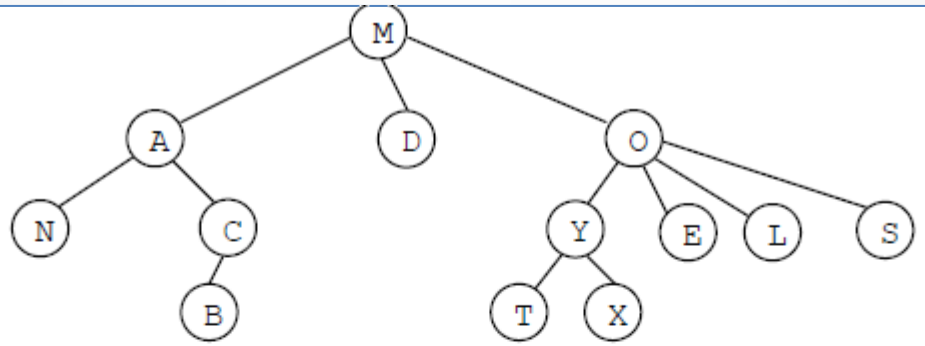
Định nghĩa cây

- So với cấu trúc liên tục như mảng, danh sách có ưu điểm vượt trội về tính mềm dẻo
- Nhưng nhược điểm lớn của danh sách là tính tuần tự và chỉ thể hiện được các mối quan hệ tuyến tính.
- Thông tin còn có thể có quan hệ dạng phi tuyến, vídụ:
  - Các thư mục file
  - Các bước di chuyển của các quân cờ
  - Sơ đồ nhân sự của tổ chức
  - Cây phả hệ
- Sử dụng cây cho phép tìm kiếm thông tin nhanh

Các khái niệm cơ bản về cây

- Một **cây** (*tree*) gồm một tập hữu hạn các **nút** (*node*) và 1 tập hữu hạn các **cành** (*branch*) nối giữa các nút. Cạnh đi vào nút gọi là **cành vào** (*indegree*), cành đi ra khỏi nút gọi là **cành ra** (*outdegree*).
- Số cạnh ra tại một nút gọi là **bậc** (*degree*) của nút đó. Nếu cây không rỗng thì phải có 1 nút gọi là **nút gốc** (*root*), **nút này không có cạnh vào**
- Các nút còn lại, mỗi nút phải có **chính xác 1 cành vào**. Tất cả các nút đều có thể có 0,1 hoặc nhiều cành ra
- *Định nghĩa*: Một cây là tập các nút mà :
  - là tập rỗng, hoặc
  - có 1 nút gọi là nút gốc có 0 hoặc nhiều cây con, các cây con cũng là cây

Các cách biểu diễn cây

- 

(a)

```
M
-   A
-   -   N
-   -   C
-   -   -   B
-   D

-   O
-   -   Y
-   -   -   T
-   -   -   X
-   -   E
-   -   L
-   -   S
```

(b)

M (A (N C (B)) D O (Y (T X) E L S))

(c)

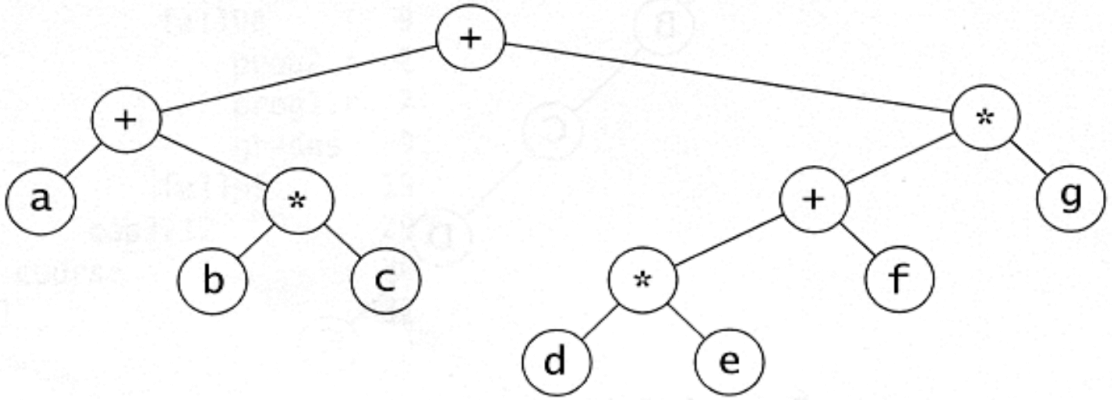
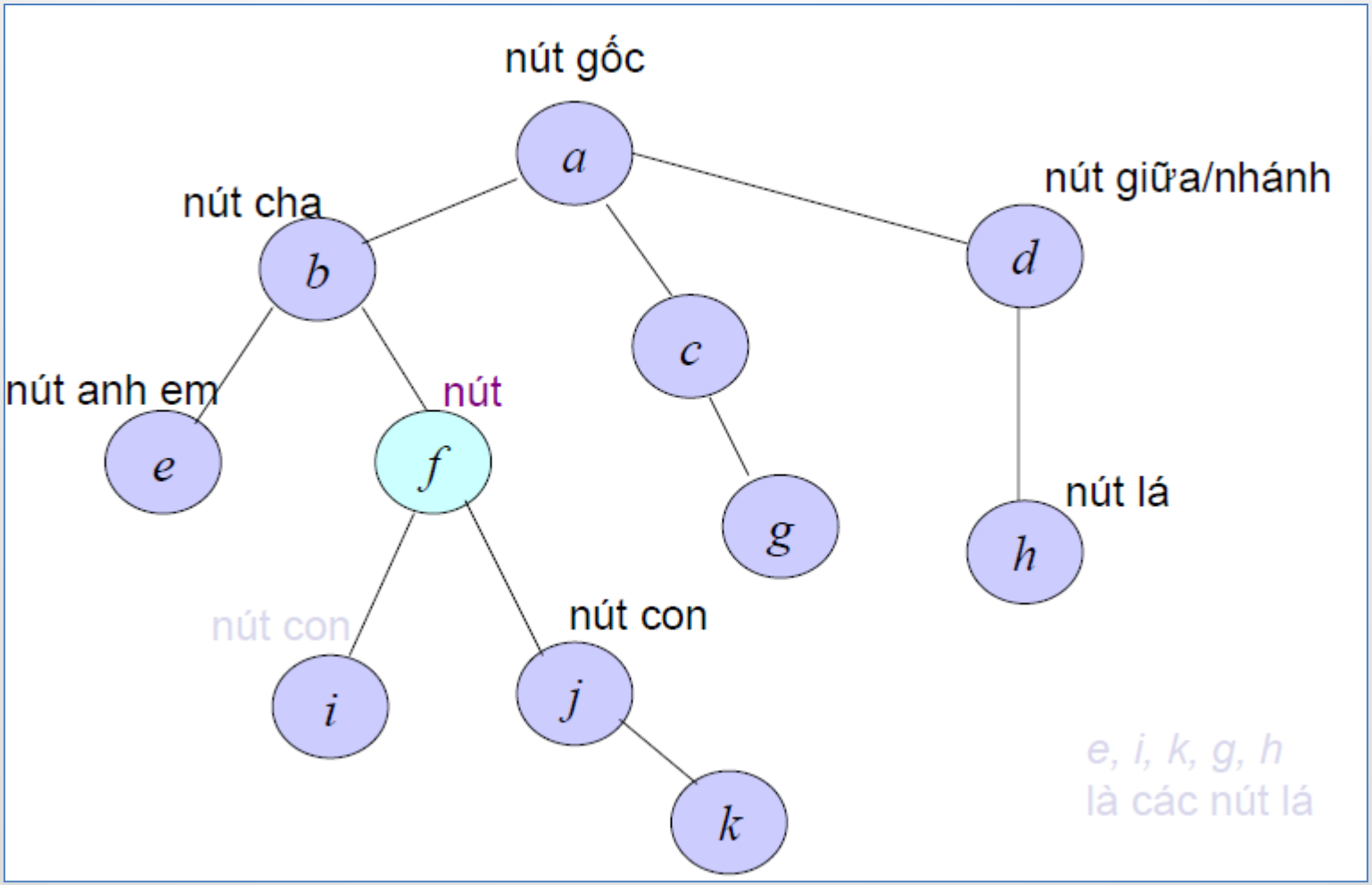
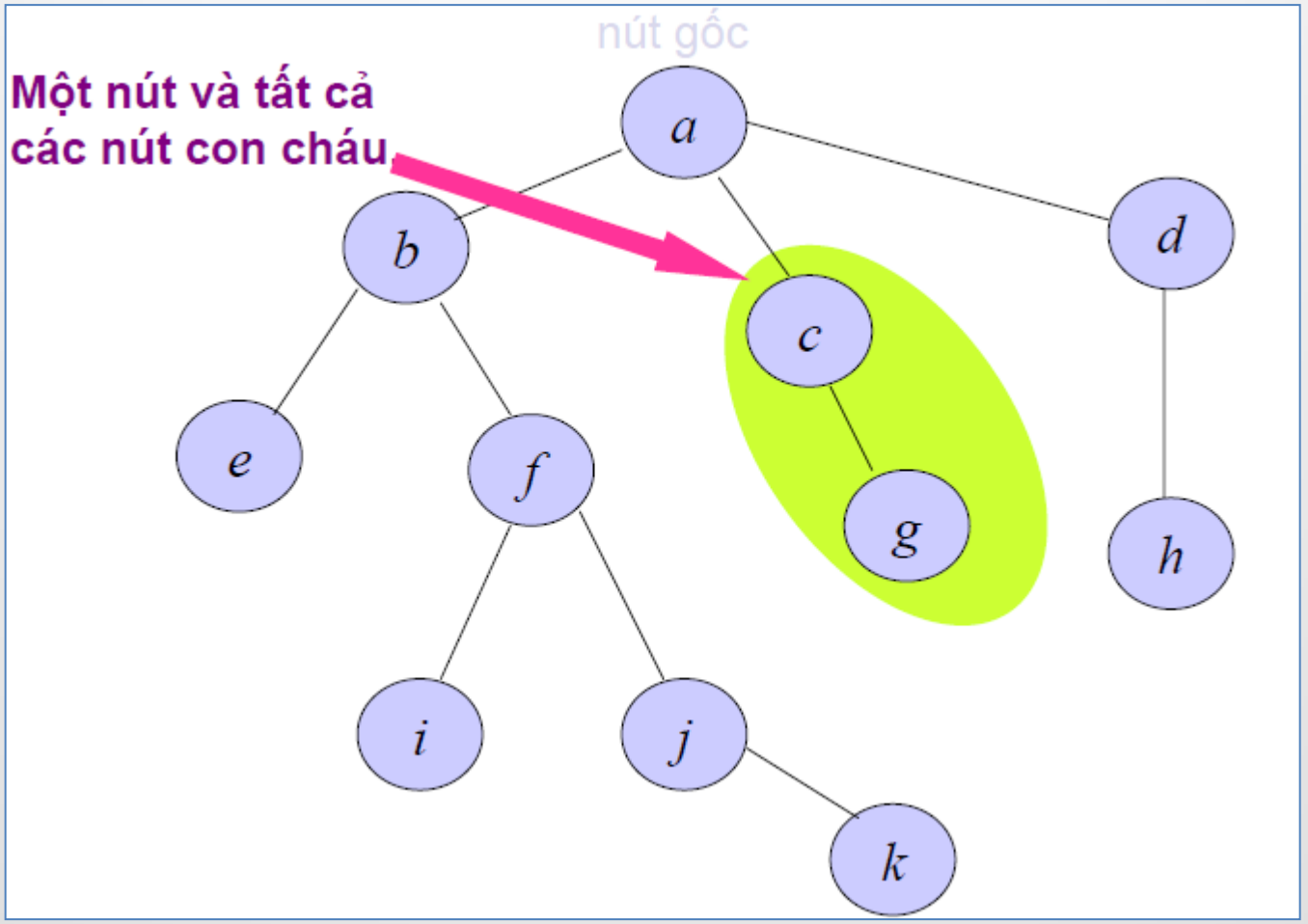


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

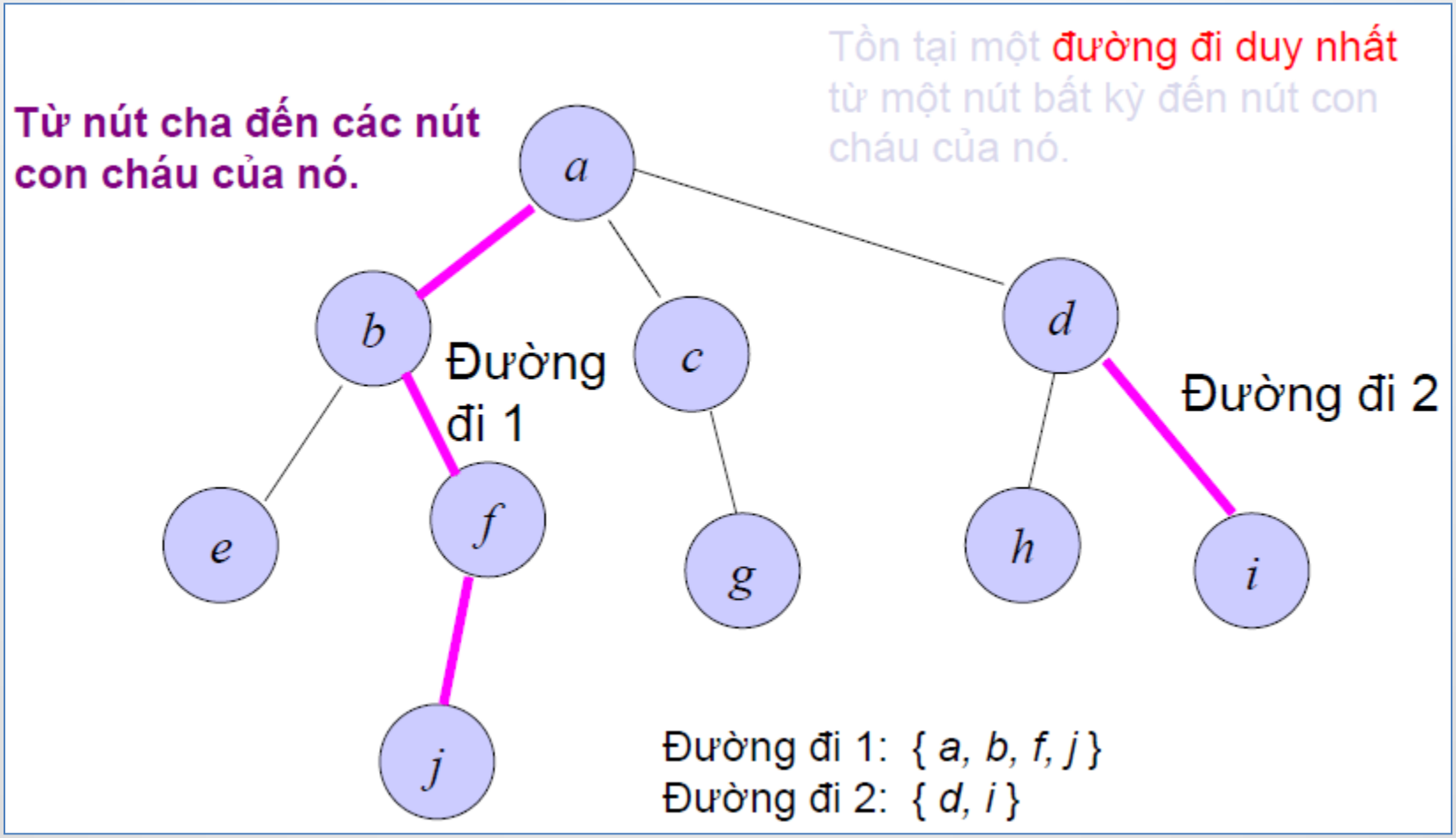
Các khái niệm



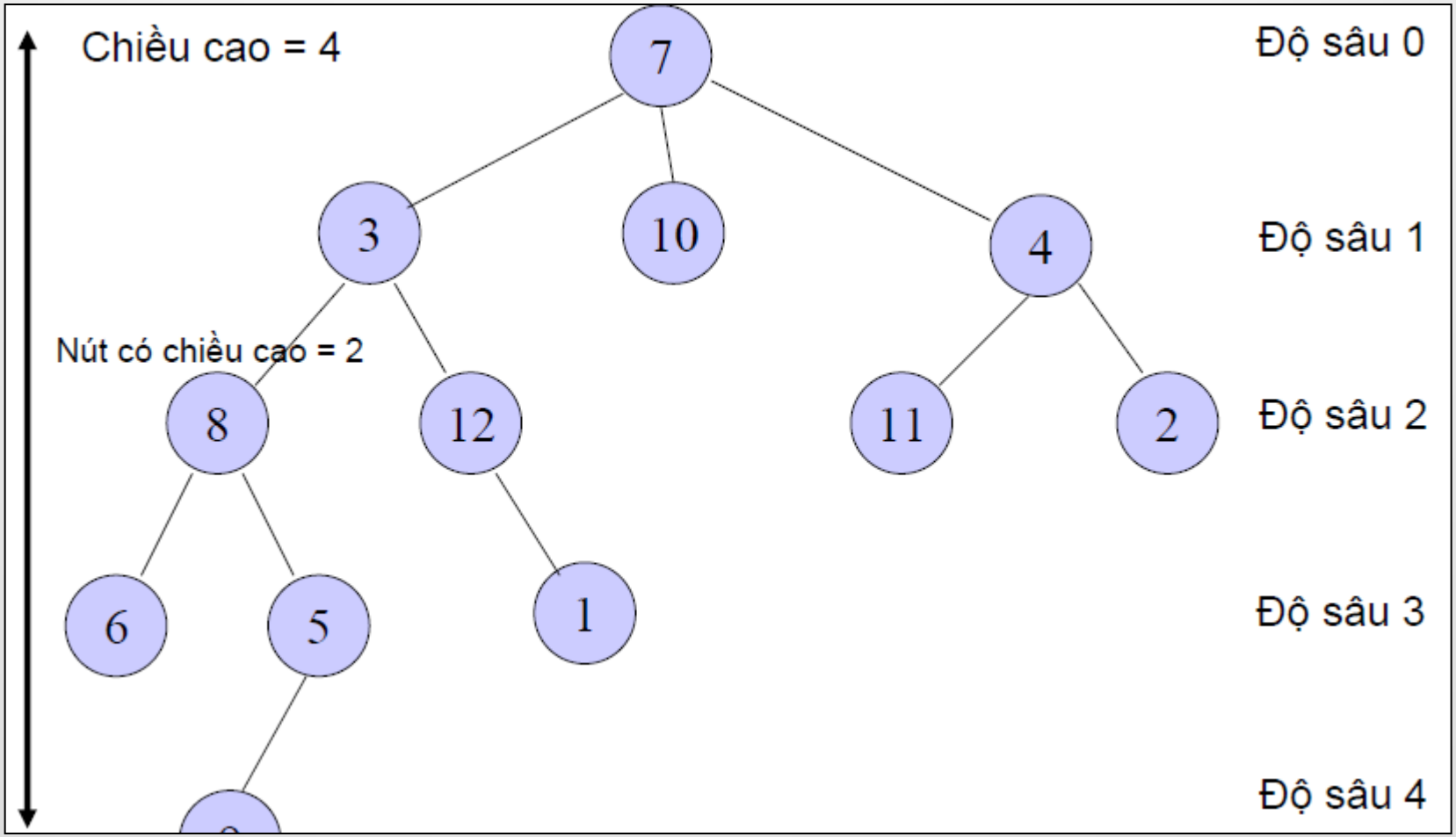
Cây con



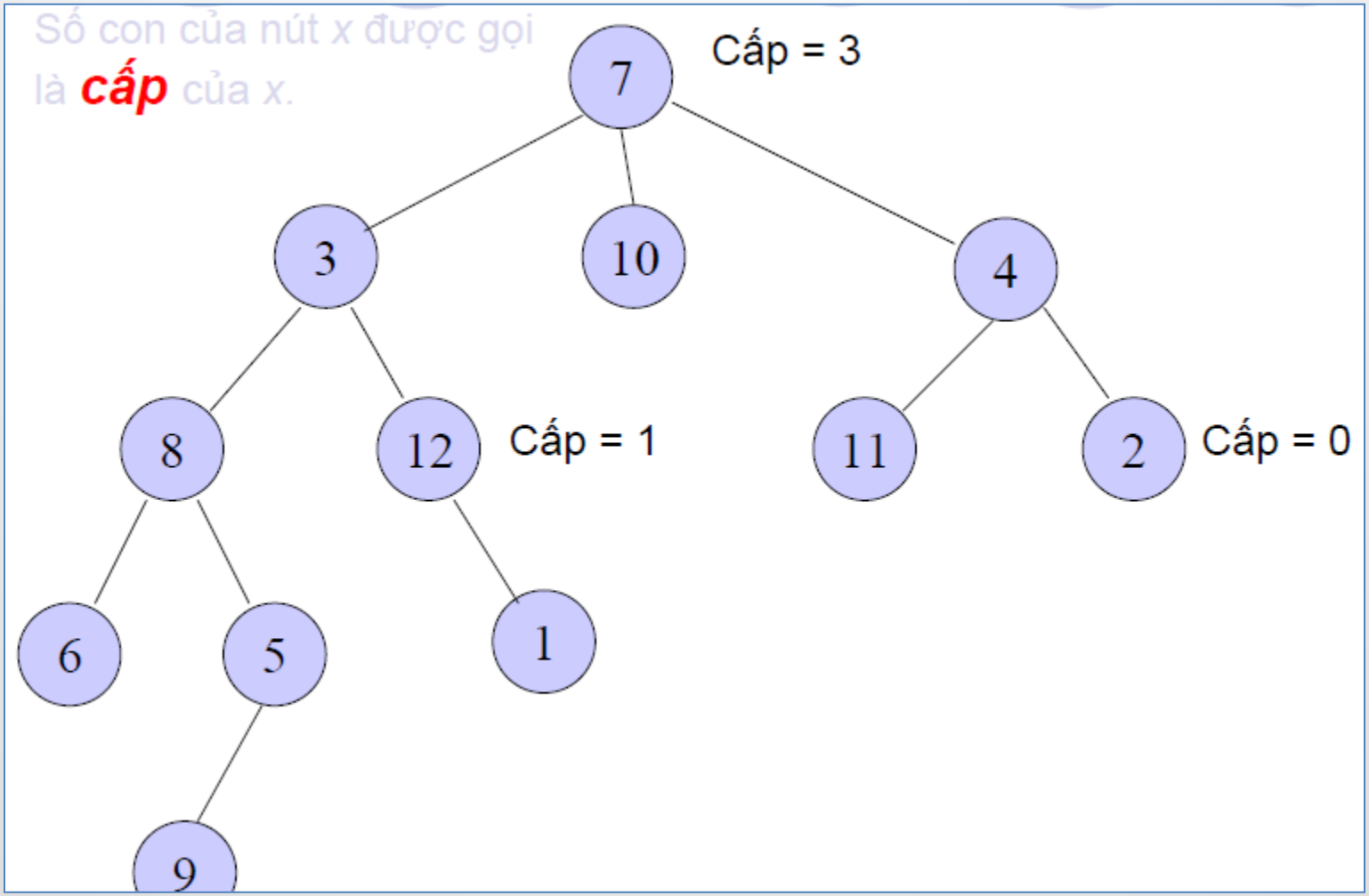
Đường đi


Tồn tại một đường đi duy nhất từ một nút bất kỳ đến nút con cháu của nó.
Từ nút cha đến các nút con cháu của nó.
a
b
c
d
e
f
g
h
i
j
Đường đi 1
Đường đi 2
Đường đi 1: { a, b, f, j }
Đường đi 2: { d, i }

Độ sâu và chiều cao

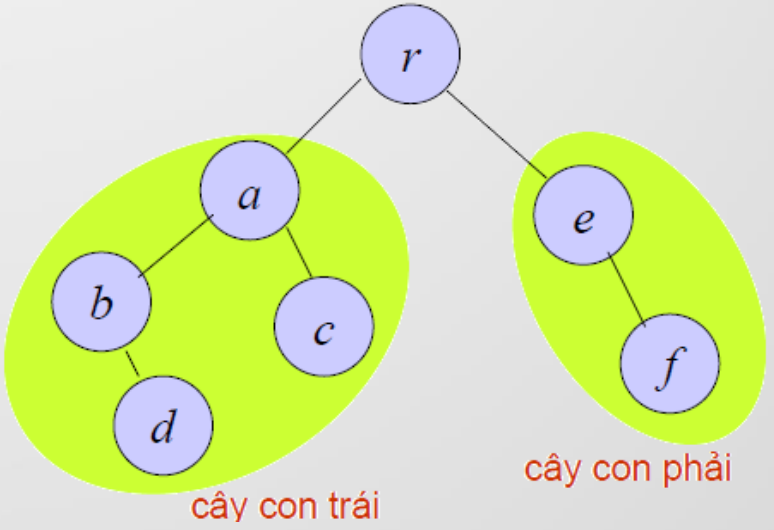


Cấp


Số con của nút x được gọi là cấp của x.
7
Cấp = 3
3
10
4
8
12
Cấp = 1
11
2
Cấp = 0
6
5
1
9

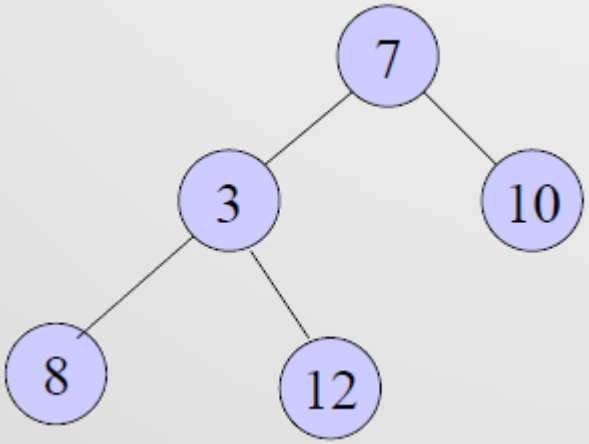
Cây nhị phân

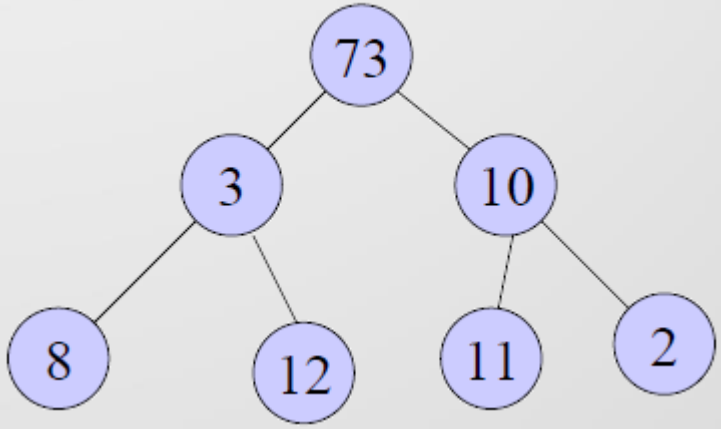
- Mỗi nút có nhiều nhất 2 nút con. Nút trái và nút phải
- Một tập các nút T được gọi là cây nhị phân, nếu :
  a) Nó là cây rỗng, hoặc
  b) Gồm 3 tập con không trùng nhau:
     1) Một nút gốc
     2) Cây nhị phân con trái
     3) Cây nhị phân con phải



Cây nhị phân đầy đủ và cây nhị phân hoàn chỉnh

- **Cây nhị phân đầy đủ**
  - Cây nút hoặc nút lá có cấp bằng 2
- **Cây nhị phân hoàn chỉnh**
  - Tất cả các nút lá có cùng độ sâu và tất cả các nút giữa có cấp bằng 2





Một số tính chất

- Số nút tối đa có độ sâu i : 2i
- Gọi N là số nút của cây nhị phân, H là chiều cao của cây thì,
  - Hmax = N, Hmin = $[\log_2 N] + 1$
  - Nmin = H, Nmax = $2^H - 1$

Cây cân bằng

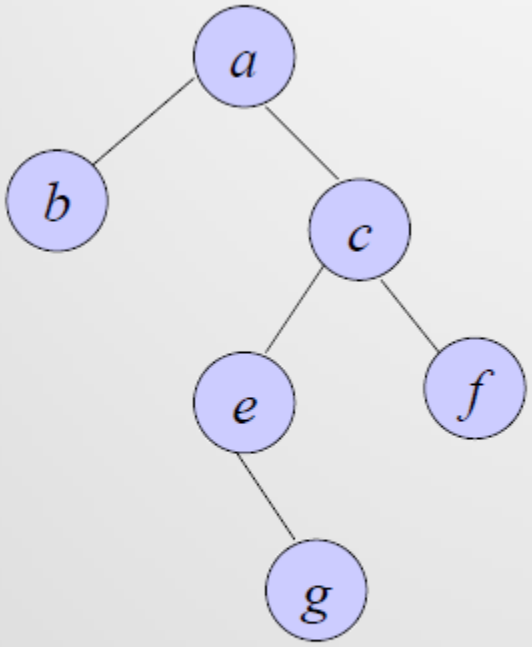
- Khoảng cách từ 1 nút đến nút gốc xác định chi phí cần để định vị nó : 1 nút có độ sâu là 5 => phải đi từ nút gốc và qua 5 cành
- Nếu cây càng thấp thì việc tìm đến các nút sẽ càng nhanh. Điều này dẫn đến tính chất cân bằng của cây nhị phân. Hệ số cân bằng của cây (*balance factor*) là sö chênh lệch giữa chiều cao của 2 cây con trái và phải của nó:

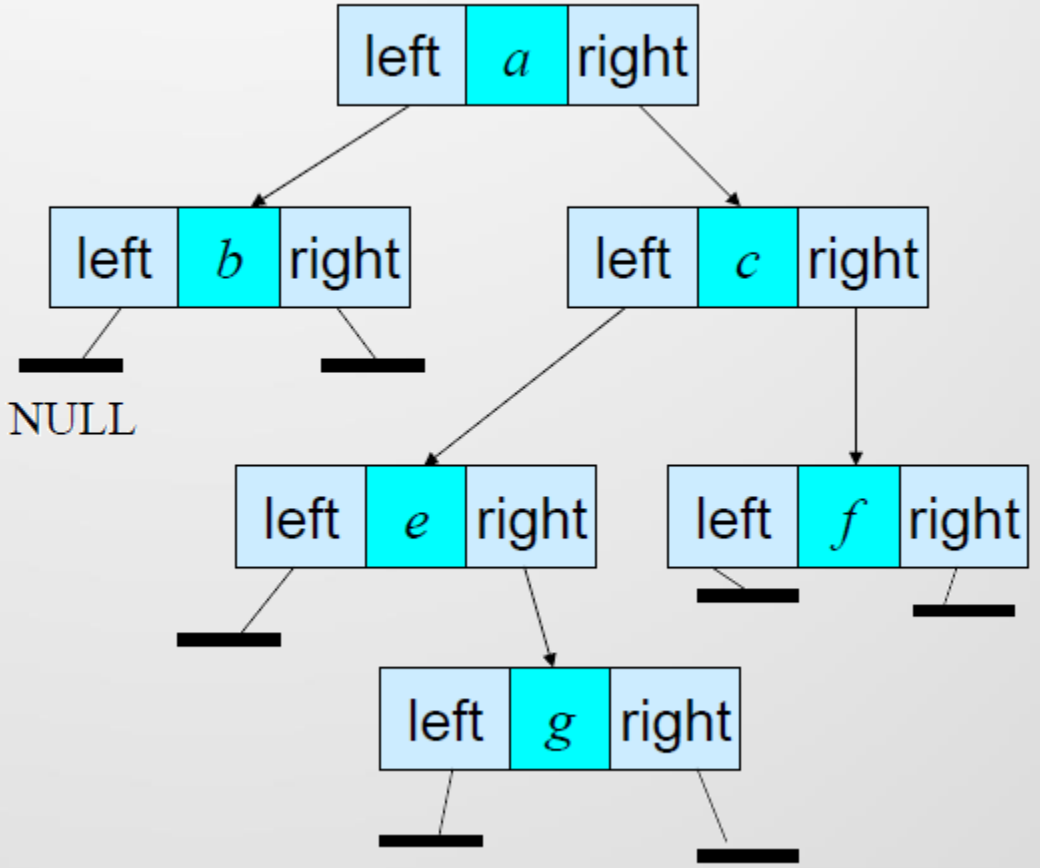
$$B = HL - HR$$

- Một **cây cân bằng** khi B = 0 và các cây con của nó cũng cân bằng

Lưu trữ cây nhị phân

- Lưu trữ kế tiếp: Sử dụng mảng
- Lưu trữ móc nối: Sử dụng con trỏ



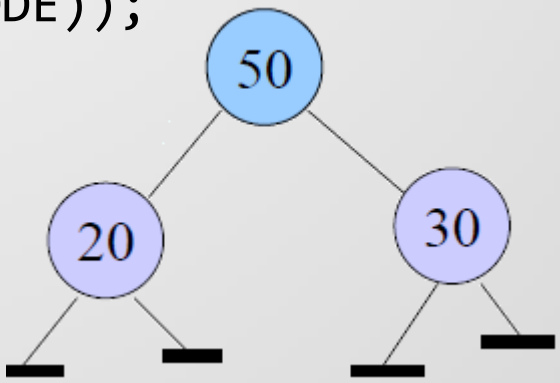


Cấu trúc cây nhị phân

```
typedef structtree_node
{
    int data ;
    structtree_node *left ;
    structtree_node *right ;
}TREE_NODE;
```

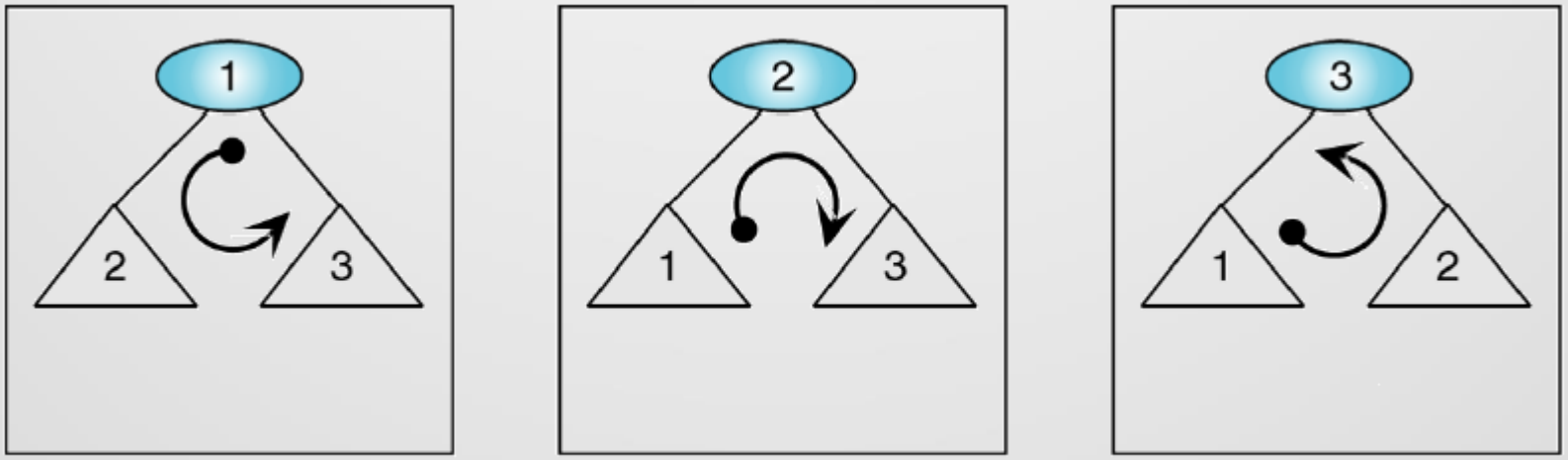
Tạo cây nhị phân

```
TREE_NODE *root, *leftChild, *rightChild;
// Tạo nút con trái
leftChild= (TREE_NODE *)malloc(sizeof(TREE_NODE));
leftChild->data = 20;
leftChild->left = leftChild->right = NULL;
// Tạo nút con phải
rightChild = (TREE_NODE *)malloc(sizeof(TREE_NODE));
rightChild->data = 30;
rightChild->left = rightChild->right = NULL;
// Tạo nút gốc
root = (TREE_NODE *)malloc(sizeof(TREE_NODE));
root->data = 10;
root->left = leftChild;
root->right = rightChild;
root -> data= 50;// gán 50 cho root
```



Duyệt cây nhị phân

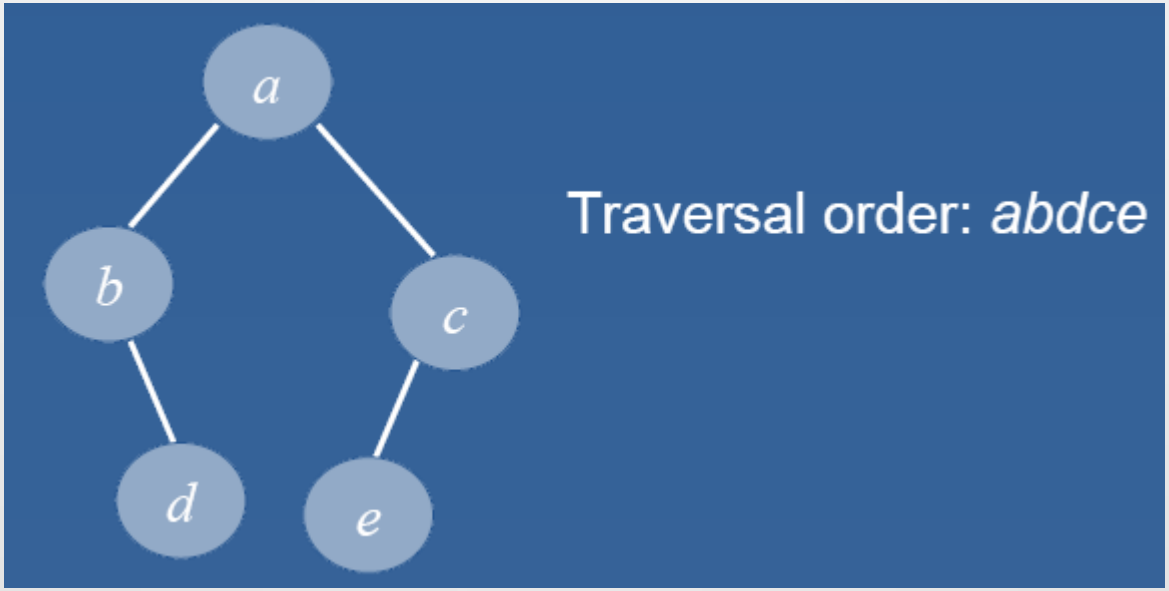
- Có 3 cách duyệt cây:
  - Duyệt theo thứ tự trước
  - Duyệt theo thứ tự giữa
  - Duyệt theo thứ tự sau
- Định nghĩa duyệt cây nhị phân là những định nghĩa đệ quy.



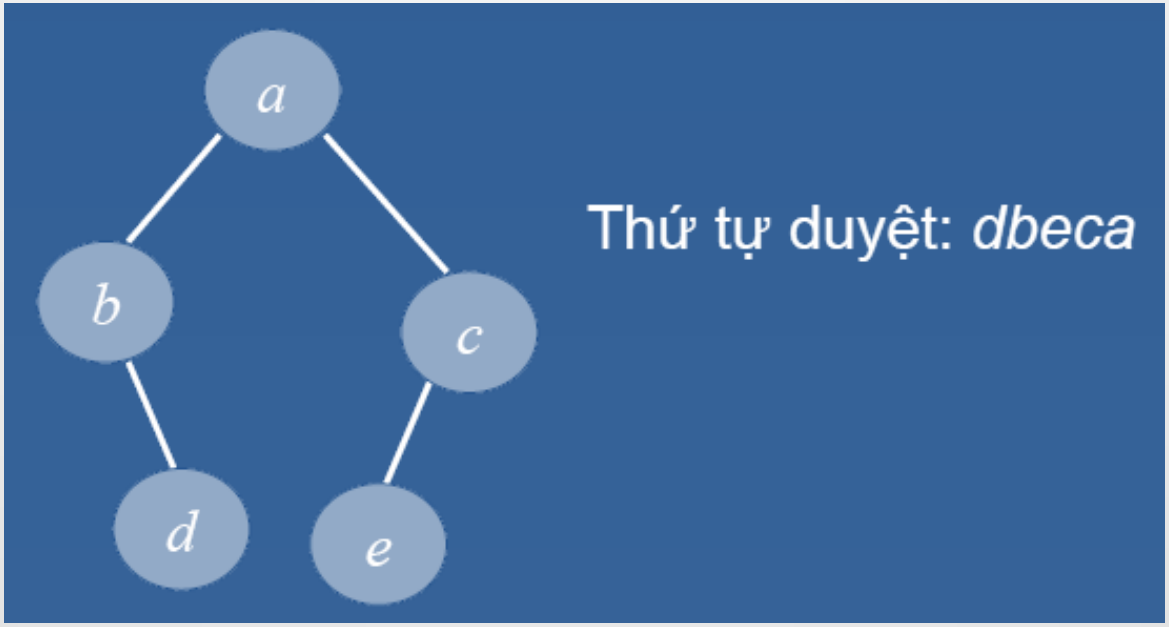
(a) Thứ tự trước (b) Thứ tự giữa (c) Thứ tự sau

Duyệt theo thứ tự trước

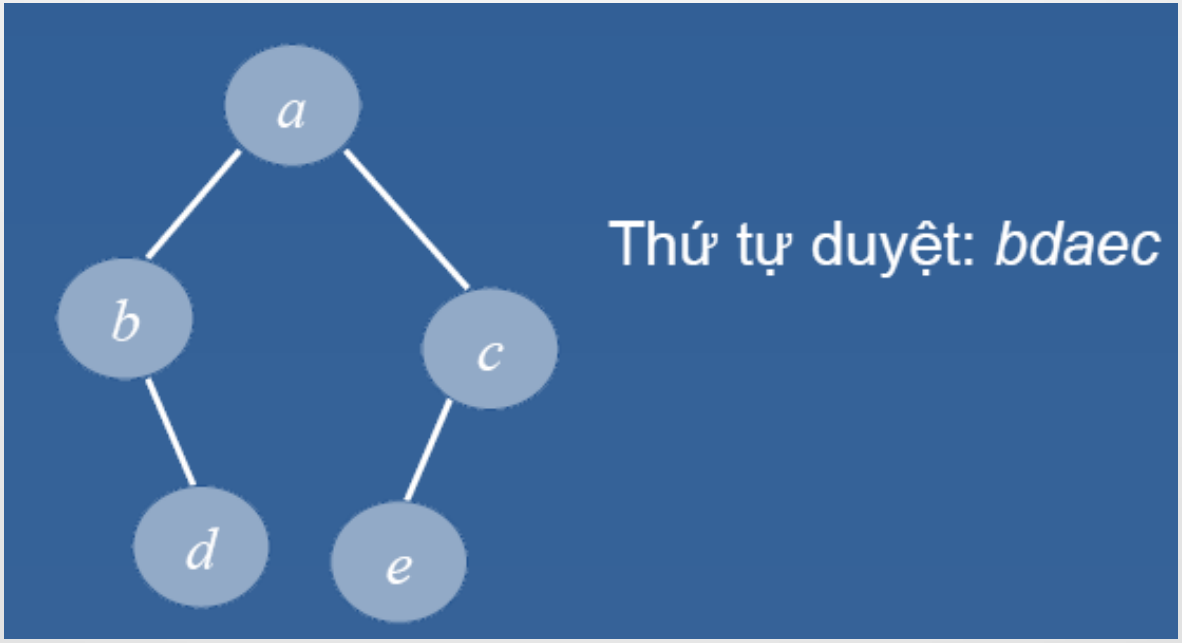
1. Thăm nút.
2. Duyệt cây con trái theo thứ tự trước.
3. Duyệt cây con phải theo thứ tự trước.

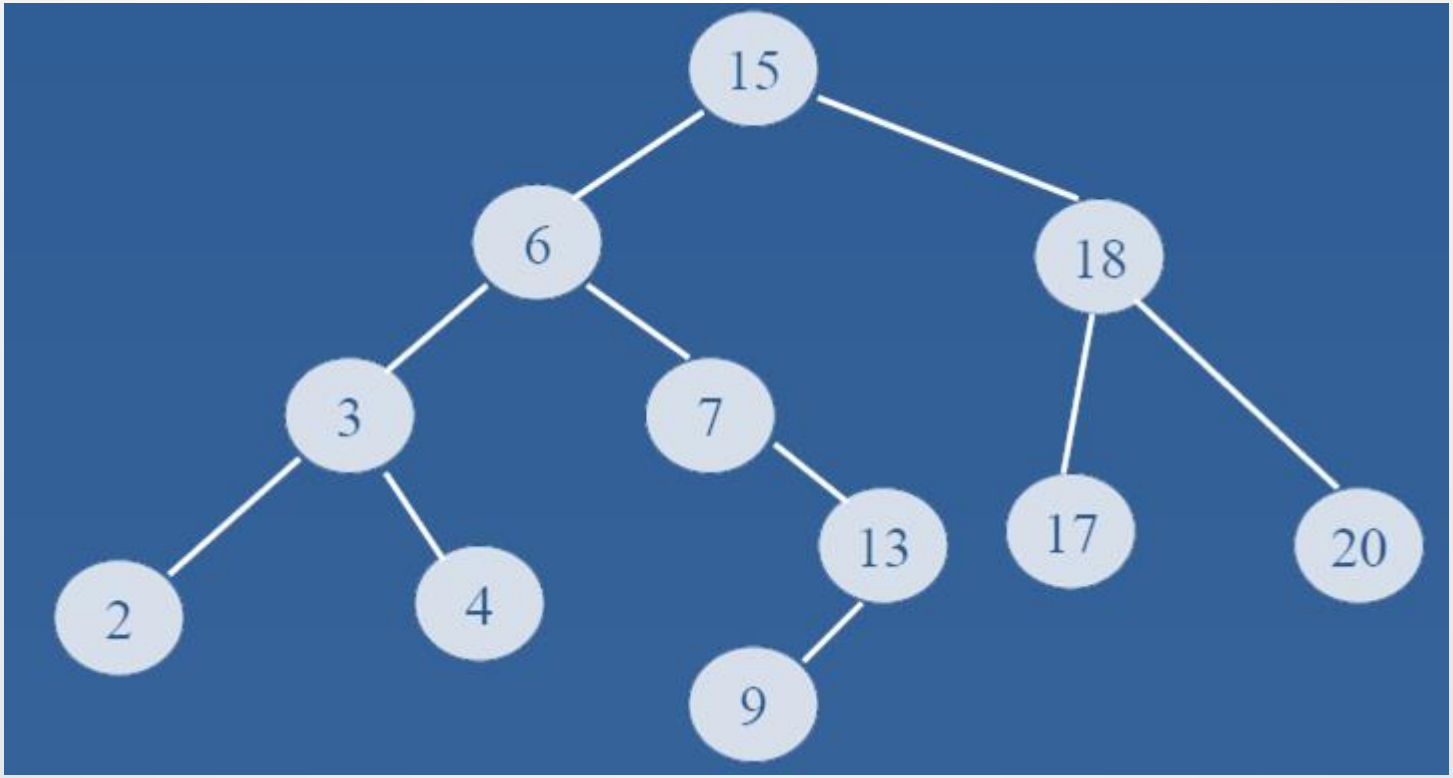


- Duyệt theo thứ tự sau
  1. Duyệt cây con trái theo thứ tự sau.
  2. Duyệt cây con phải theo thứ tự sau
  3. Thăm nút



- Duyệt theo thứ tự giữa
  1. Duyệt cây con trái theo thứ tự giữa
  2. Thăm nút.
  3. Duyệt cây con phải theo thứ tự giữa.





- Thứ tự trước: 15, 6, 3, 2, 4, 7, 13, 9, 18, 17, 20
- Thứ tự giữa: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20
- Thứ tự sau: 2, 4, 3, 9, 13, 7, 6, 17, 20, 18, 15

Duyệt theo thứ tự trước

```
void Preorder(TREE_NODE *root)
{
    if (root != NULL)
    {
        // tham node
        printf("%d", root->data);
        // duyet cay con trai
        Preorder(root->left);
        // duyet cay con phai
        Preorder(root->right);
    }
}
```

- Bài tập: Viết giải thuật đệ quy của
  - Duyệt theo thứ tự giữa
  - Duyệt theo thứ tự sau

Duyệt theo thứ tự trước

- Dùng vòng lặp

```
void Preorder_iter(TREE_NODE *treeRoot)
{
	TREE_NODE *curr= treeRoot;
	STACK *stack= createStack(MAX);// khởi tạostack
	while (curr != NULL || !IsEmpty(stack))
	{
		printf("%d", curr->data); // thămcurr
		// nếu có cây con phải, đẩy cây con phải vào stack
		if (curr->right != NULL)
			pushStack(stack, curr->right);
		if(curr->left!=NULL)
			curr= curr->left; // duyệt cây con trái
		else
			popStack(stack, &curr);// duyệt cây con phải
	}
	destroyStack(&stack);// giải phóng stack
}
```

Duyệt theo thứ tự giữa

```
void Inorder_iter(TREE_NODE *root) {
     TREE_NODE *curr= root;
     STACK *stack= createStack(MAX);//ktạo stack
     while(curr != NULL || !IsEmpty(stack))
     {
              if (curr==NULL) {
                       popStack(stack, &curr);
                       printf("%d", curr->data);
                       curr= curr->right;
              }
              else {
                       pushStack(stack, curr);
                       curr= curr->left; // duyệt cây con trái
              }
     }
     destroyStack(stack); //giải phóng stack
}
```

Duyệt theo thứ tự cuối

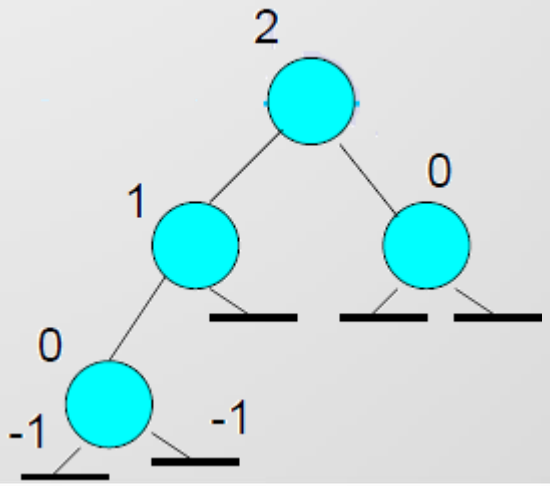
```
voidPostorder_iter(TREE_NODE *treeRoot)
{
     TREE_NODE *curr= treeRoot;
     STACK *stack= createStack(MAX);//ktạo một stack
     while( curr != NULL || !IsEmpty(stack)) {
              if (curr == NULL) {
                       while(!IsEmpty(stack) && curr==Top(stack)->right){
                                PopStack(stack, &curr);
                                printf(“%d”, curr->data);
                       }
                       curr= isEmpty(stack)? NULL: Top(stack)->right;
              }
              else{
                       PushStack(stack, curr);
                       curr= curr->left;
              }
     }
     destroyStack(&stack);// giải phóng stack
}
```

Một vài ứng dụng của duyệt cây

1. Tính độ cao của cây
2. Đếm số nút lá trong cây
3. Tính kích thước của cây (số nút)
4. Sao chép cây
5. Xóa cây

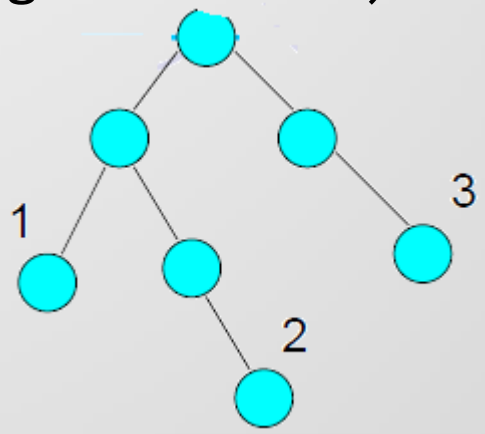
Tính độ cao của cây

```
int Height(TREE_NODE *tree)
{
    Int heightLeft, heightRight, heightval;
    if( tree== NULL )
          heightval= -1;
    else
    { // Sửdụng phương pháp duyệt theo thứ tự sau
          heightLeft= Height (tree->left);
          heightRight= Height (tree->right);
          heightval= 1 +
max(heightLeft,heightRight);
    }
    return heightval;
}
```



Đếm số nút lá

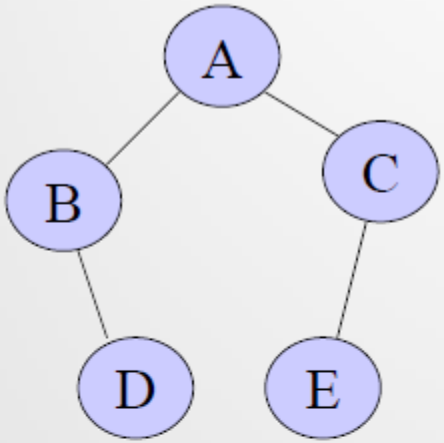
```
int CountLeaf(TREE_NODE *tree)
{
   if (tree == NULL)
         return 0;
   int count = 0;
   //Đếm theo thứ tự sau
   count += CountLeaf(tree->left);// Đếm trái
   count += CountLeaf(tree->right);//Đếm phải
   //nếu nút tree là nút lá, tăng count
   if(tree->left == NULL && tree->right == NULL)
         count++;
   return count;
}
```

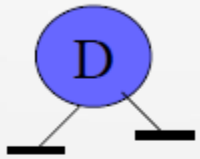


Kích thước của cây

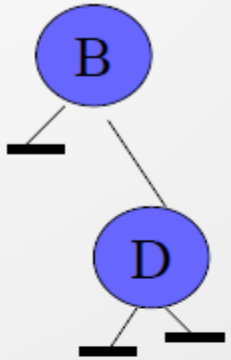
```
int TreeSize(TREE_NODE *tree)
{
   if(tree== NULL)
         return 0;
   else
         return( TreeSize(tree->left) +
               TreeSize(tree->right) + 1 );
}
```

Sao chép cây

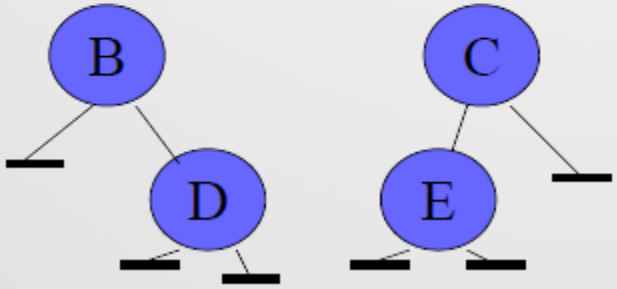




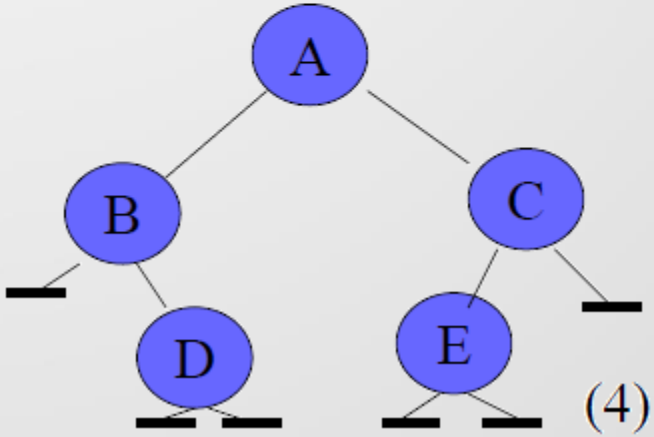
(1)



(2)



(3)



(4)

Sao chép cây

```
TREE_NODE *CopyTree(TREE_NODE *tree)
{
  // Dừng đệ quy khi cây rỗng
  if (tree== NULL) return NULL;
  TREE_NODE *leftsub, *rightsub, *newnode;
  leftsub=CopyTree(tree->left);
  rightsub= CopyTree(tree->right);
  // tạo cây mới
  newnode= malloc(sizeof(TREE_NODE));
  newnode->data = tree->data;
  newnode->left = leftsub;
  newnode->right = rightsub;
  return newnode;
}
```

Xóa cây

```
void DeleteTree(TREE_NODE *tree)
{
//xóa theo thứ tự sau
    if(tree != NULL)
    {
          DeleteTree(tree->left);
          DeleteTree(tree->right);
          free(tree);
    }
}
```

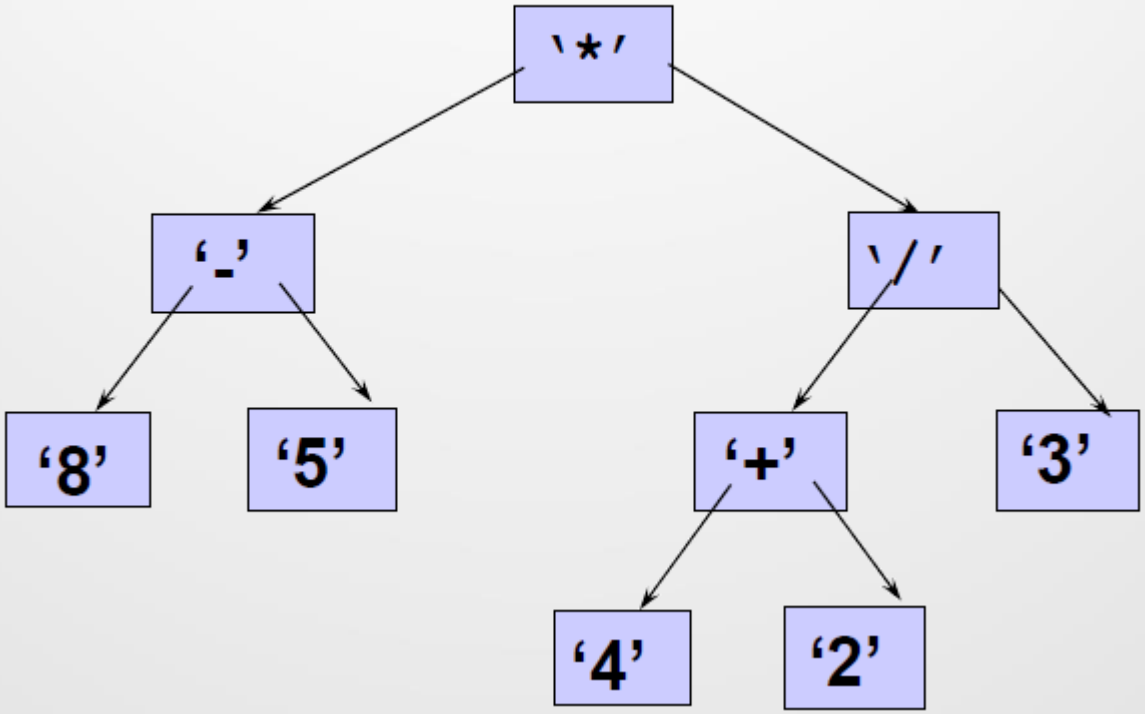
Ứng dụng của cây nhị phân

- **Cây biểu diễn biểu thức**
  - Tính giá trị biểu thức
  - Tính đạo hàm
- **Cây quyết định**

Cây biểu diễn biểu thức

- **là một loại cây nhị phân đặc biệt, trong đó:**
  1. Mỗi nút lá chứa một toán hạng
  2. Mỗi nút giữa chứa một toán tử
  3. Cây con trái và phải của một nút toán tử thể hiện các biểu thức con cần được đánh giá trước khi thực hiện toán tử tại nút gốc

Cây biểu diễn biểu thức

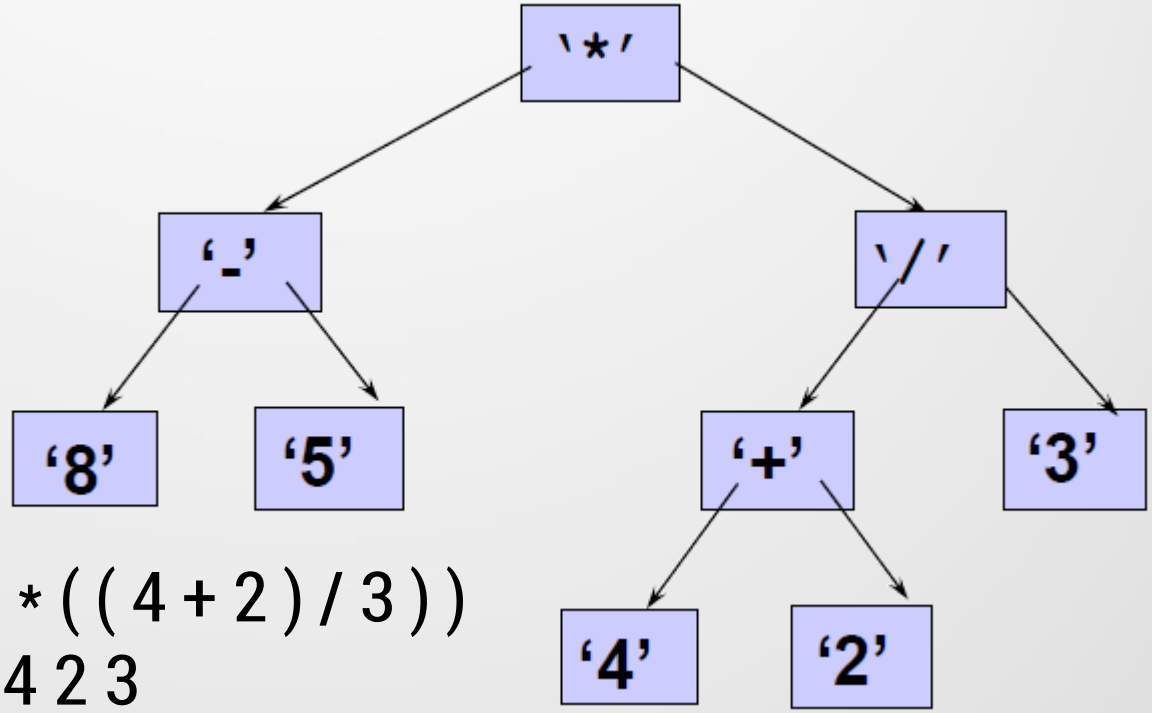


Cây biểu diễn biểu thức

- **Các mức chỉ ra thứ tự ưu tiên**
  - Các mức (độ sâu) của các nút chỉ ra thứ tự ưu tiên tương đối của chúng trong biểu thức (không cần dùng ngoặc để thể hiện thứ tự ưu tiên).
  - Các phép toán tại mức cao hơn sẽ được tính sau các các phép toán có mức thấp.
  - Phép toán tại gốc luôn được thực hiện cuối cùng.

Cây biểu diễn biểu thức

- Dễ dàng để tạo ra các biểu thức tiền tố, trung tố, hậu tố



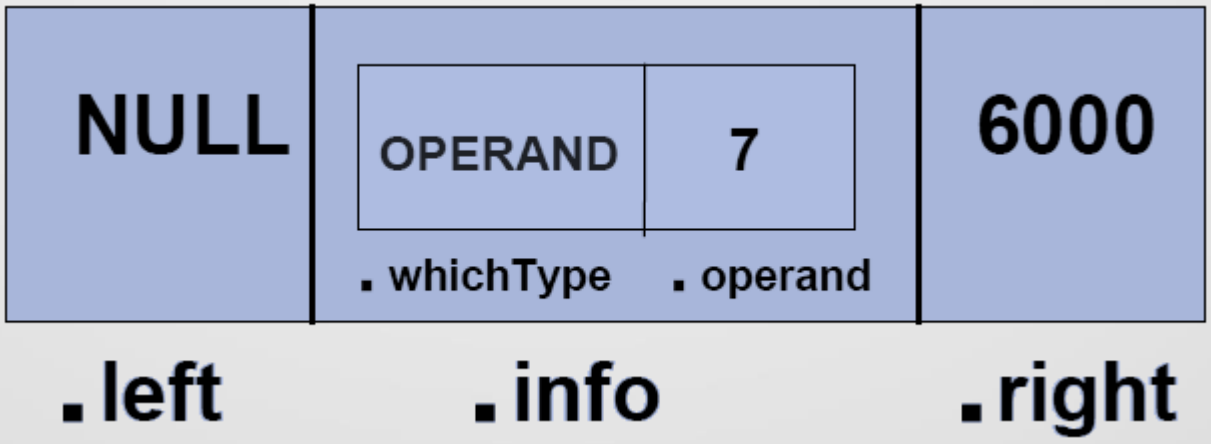
- Trung tố: ((8 -5) * ((4 + 2) / 3))
- Tiền tố: * -8 5 / + 4 2 3
- Hậu tố: 8 5 -4 2 + 3 / *

(thực chất là các phép duyệt theo tt giữa, trước và sau)

Cài đặt cây biểu thức

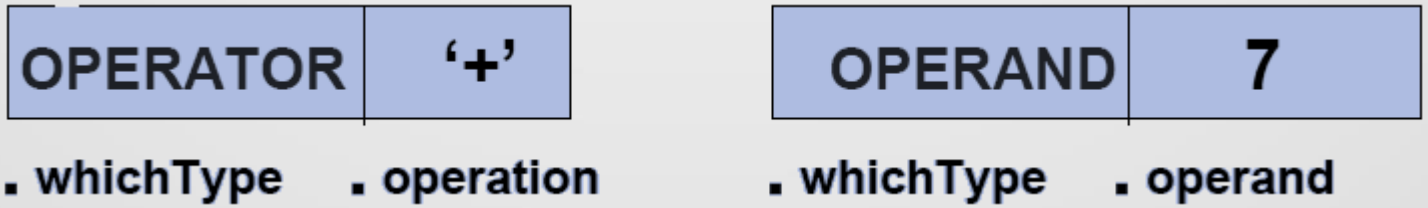
- **Mỗi nút có 2 con trỏ**

```
struct TreeNode {
    InfoNode info ;// Dữ liệu
    TreeNode *left ;// Trỏ tới nút con trái
    TreeNode *right ; // Trỏ tới nút con phải
};
```



- InfoNode có 2 dạng

```
enum OpType { OPERATOR, OPERAND } ;
struct InfoNode {
   OpType    whichType;
   union                        // ANONYMOUS  union
   {
        char operator;
        int operand ;
    }
};
```



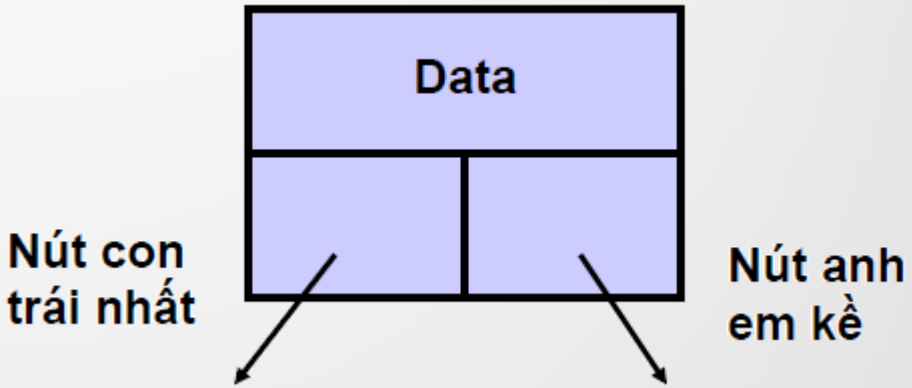
```
int Eval(TreeNode* ptr){
   switch(ptr->info.whichType) {
       case OPERAND :
          returnptr->info.operand ;
       case OPERATOR :
          switch ( tree->info.operation ){
                case ‘+’:
                return ( Eval( ptr->left ) + Eval( ptr->right ) ) ;
                case ‘-’:
                return ( Eval( ptr->left ) -Eval( ptr->right ) ) ;
                case ‘*’:
                return ( Eval( ptr->left ) * Eval( ptr->right ) ) ;
                case ‘/’:
                return ( Eval( ptr->left ) / Eval( ptr->right ) ) ;
           }
   }
}
```

Cây tổng quát

Biểu diễn cây tổng quát

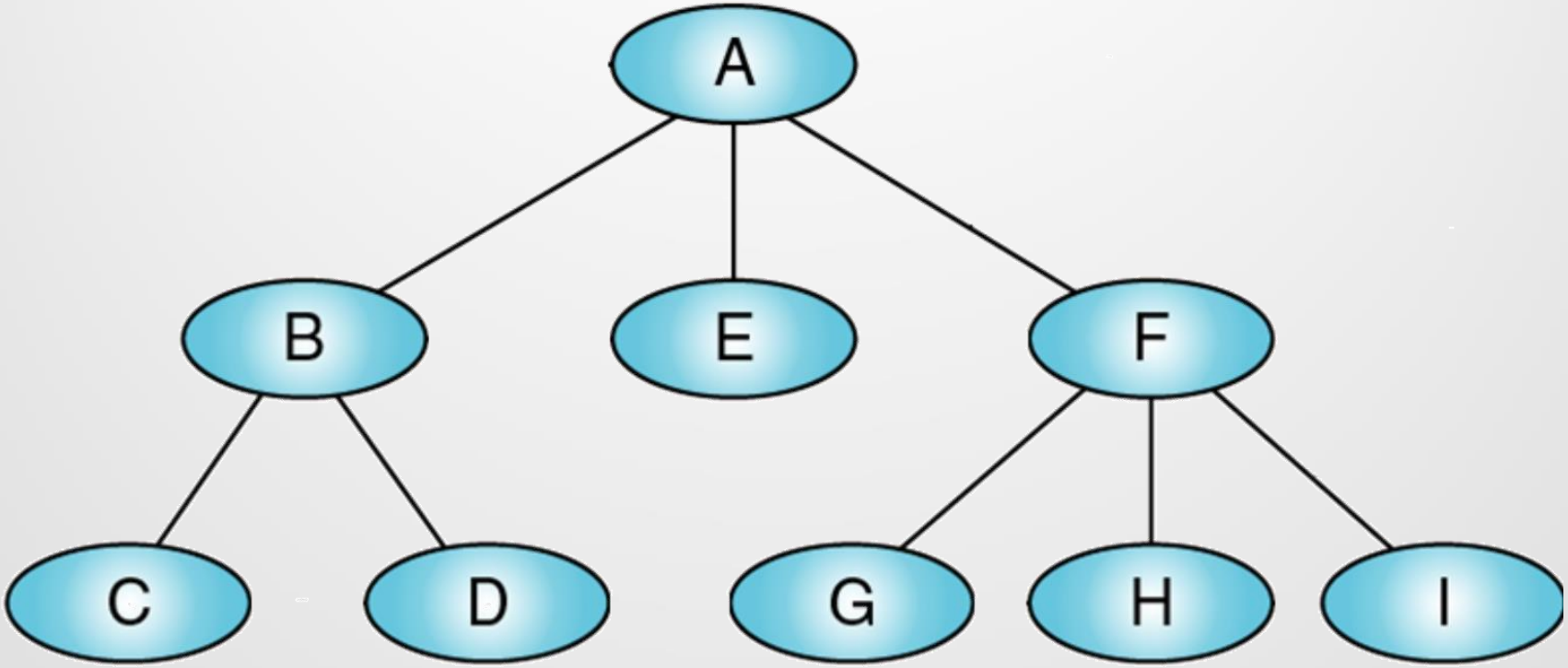
- Biểu diễn giống như cây nhị phân?
  - Mỗi nút sẽ chứa giá trị và các con trỏ trỏ đến các nút con của nó?
  - Bao nhiêu con trỏ cho một nút? -> Không hợp lý
- Mỗi nút sẽ chứa giá trị và một con trỏ trỏ đến một “tập" các nút con
  - Xây dựng “tập" như thế nào?

- Sử dụng con trỏ nhưng mở rộng hơn:
  - Mỗi nút sẽ có 2 con trỏ: một con trỏ trỏ đến nút con đầu tiên của nó, con trỏ kia trỏ đến nút anh em kề với nó
  - Cách này cho phép quản lý số lượng tùy ý của các nút con



Biểu diễn cây tổng quát

Ví dụ



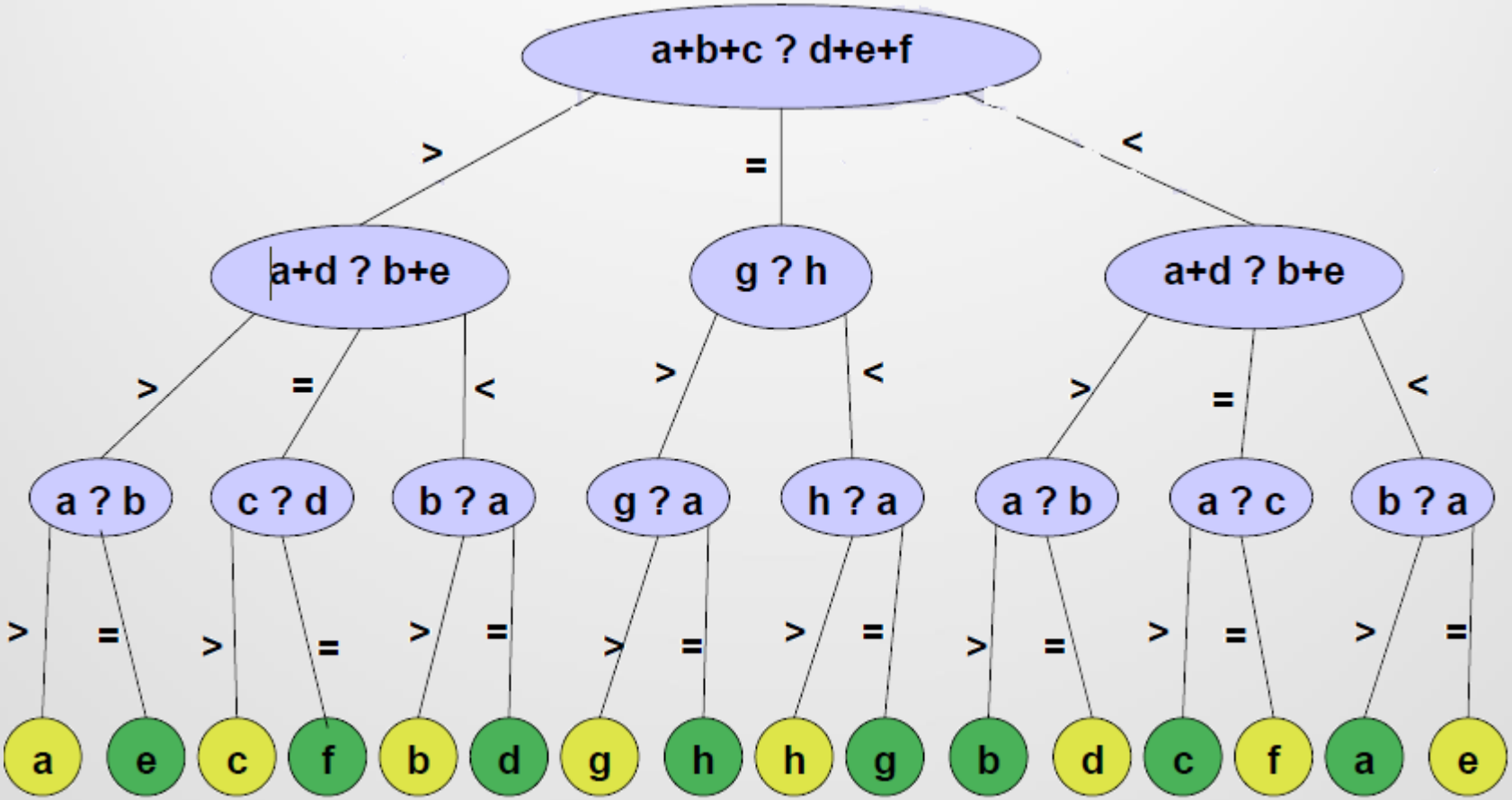
Duyệt cây tổng quát

- Thứ tự trước:
  1. Thăm gốc
  2. Duyệt cây con thứ nhất theo thứ tự trước
  3. Duyệt các cây con còn lại theo thứ tự trước
- Thứ tự giữa
  1. Duyệt cây con thứ nhất theo thứ tự giữa
  2. Thăm gốc
  3. Duyệt các cây con còn lại theo thứ tự giữa
- Thứ tự sau:
  1. Duyệt cây con thứ nhất theo thứ tự sau
  2. Duyệt các cây con còn lại theo thứ tự sau
  3. Thăm gốc

Cây quyết định

- Dùng để biểu diễn lời giải của bài toán cần quyết định lựa chọn
- Bài toán 8 đồng tiền vàng:
  - Có 8 đồng tiền vàng a, b, c, d, e, f, g, h
  - Có một đồng có trọng lượng không chuẩn
  - Sử dụng một cân Roberval (2 đĩa)
  - Output:
    - Đồng tiền k chuẩn là nặng hơn hay nhẹ hơn
    - Số phép cân là ít nhất

Cây quyết định



```
void EightCoins(a, b, c, d, e, f, g, h) {
    if (a+b+c == d+e+f) {
    if (g > h) Compare(g, h, a);
    else Compare(h, g, a);
  }
  else if (a+b+c > d+e+f){
    if (a+d == b+e) Compare(c, f, a);
    else if (a+d > b+e) Compare(a, e, b);
    else Compare(b, d, a);
  }
  else{
    if (a+d == b+e) Compare(f,c,a);
    else if (a+d > b+e) Compare(d, b, a);
    else Compare(e, a, b);
  }
}
// so sánh x với đồng tiền chuẩn z
void Compare(x,y,z){
  if(x>y) printf(“x nặng”);
  else printf(“y nhẹ”);
}
```