CẤU TRÚC DỮ LIỆU

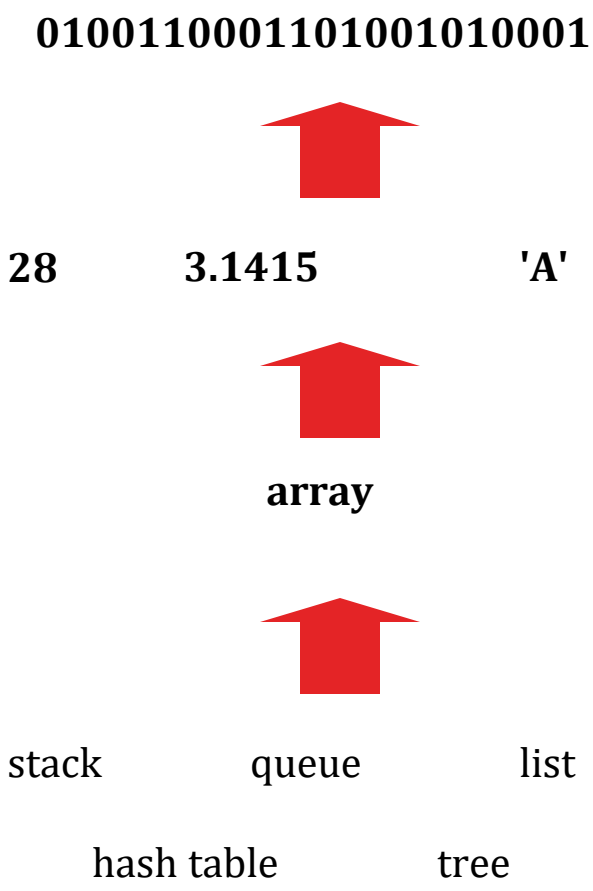
Dữ liệu, kiểu dữ liệu & cấu trúc dữ liệu

Machine Level Data Storage

Primitive Data Types

Basic Data Structures

High-Level Data Structures



Các kiểu dữ liệu

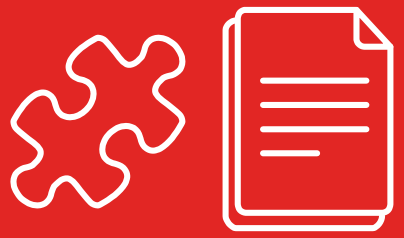
Kiểu dữ liệu cơ bản (primitive data type)

- 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, bool...*
  - *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

Nội dung



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

1. Mảng

Array

Mảng
Array

- Dãy hữu hạn các phần tử liên tiếp có cùng kiểu và tên
- Một hay nhiều chiều
  - *C không giới hạn số chiều của mảng*

Cú pháp

```
DataType ArrayName[size];
```

mảng nhiều chiều

```
DataType ArrayName[size 1][size 2]...[size n];
```

Khởi tạo giá trị mảng

- C1. Khi khai báo

```
float y[5] = { 3.2, 1.2, 4.5, 6.0, 3.6 }
int m[6][2] = { { 1, 1 }, { 1, 2 }, { 2, 1 }, { 2, 2
}, { 3, 1 }, { 3, 2 } };
char s1[6] = { 'H', 'a', 'n', 'o', 'i', '\0' }; //hoặc
char s1[6] = "Hanoi";
char s1[] = "Dai hoc Bach Khoa Hanoi";  //L = 24
int  m[][] = { { 1, 2, 3 }, { 4, 5, 6 } };
```

- C2. Khai báo rồi gán giá trị cho từng phần tử của mảng.

```
int m[4];
m[0] = 1; m[1] = 2; m[2] = 3; m[3] = 4;
```

2.
Danh sách

List

Danh sách
List

- 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 phi tuyến tính: các phần tử trong danh sách không được sắp thứ tự*

Danh sách
List

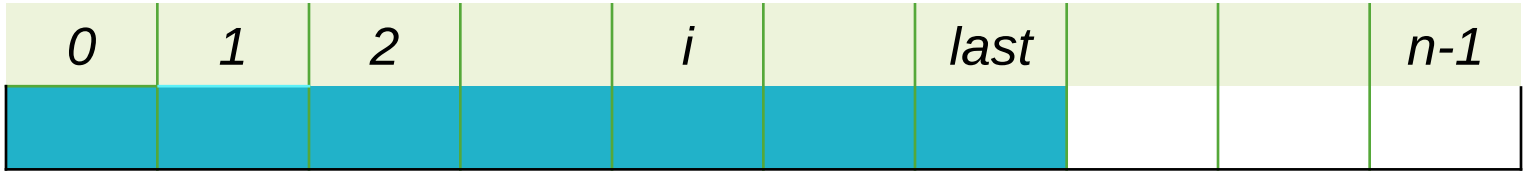
- Lưu trữ
  - *Sử dụng vùng các ô nhớ liên tiếp trong bộ nhớ → danh sách kế tiếp*
  - *Sử dụng vùng các ô nhớ không liên tiếp trong bộ nhớ → danh sách móc nối*
    - **Danh sách nối đơn**
    - **Danh sách nối kép**

Thao tác trên danh sách

- *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
  - *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
  - *Tốc độ truy cập vào các phần tử của danh sách nhanh*
- Nhược điểm
  - *Cần phải biết trước kích thước tối đa của danh sách*

?

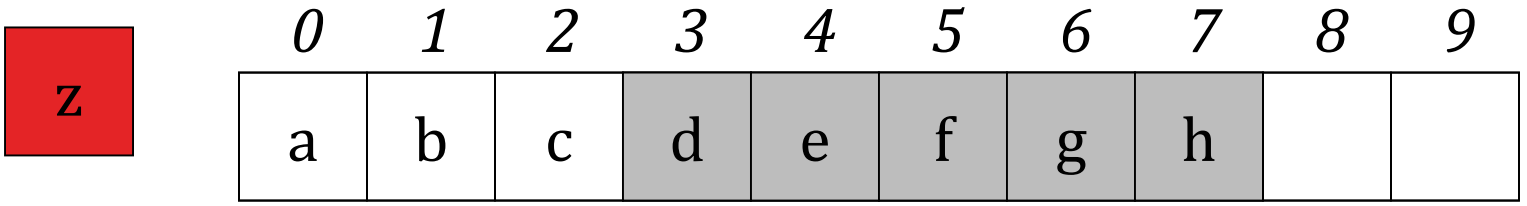
  - *Thực hiện các phép toán bổ sung các phần tử mới và loại bỏ các phần tử cũ khá tốn kém*

?

Thêm vào danh sách kế tiếp

- 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')

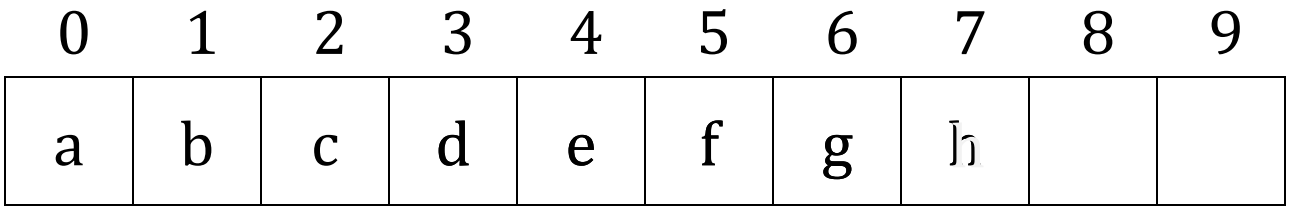


Thêm vào danh sách kế tiếp

```
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 khỏi danh sách kế tiếp

remove(3, ‘d’)



count=7

Xóa 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

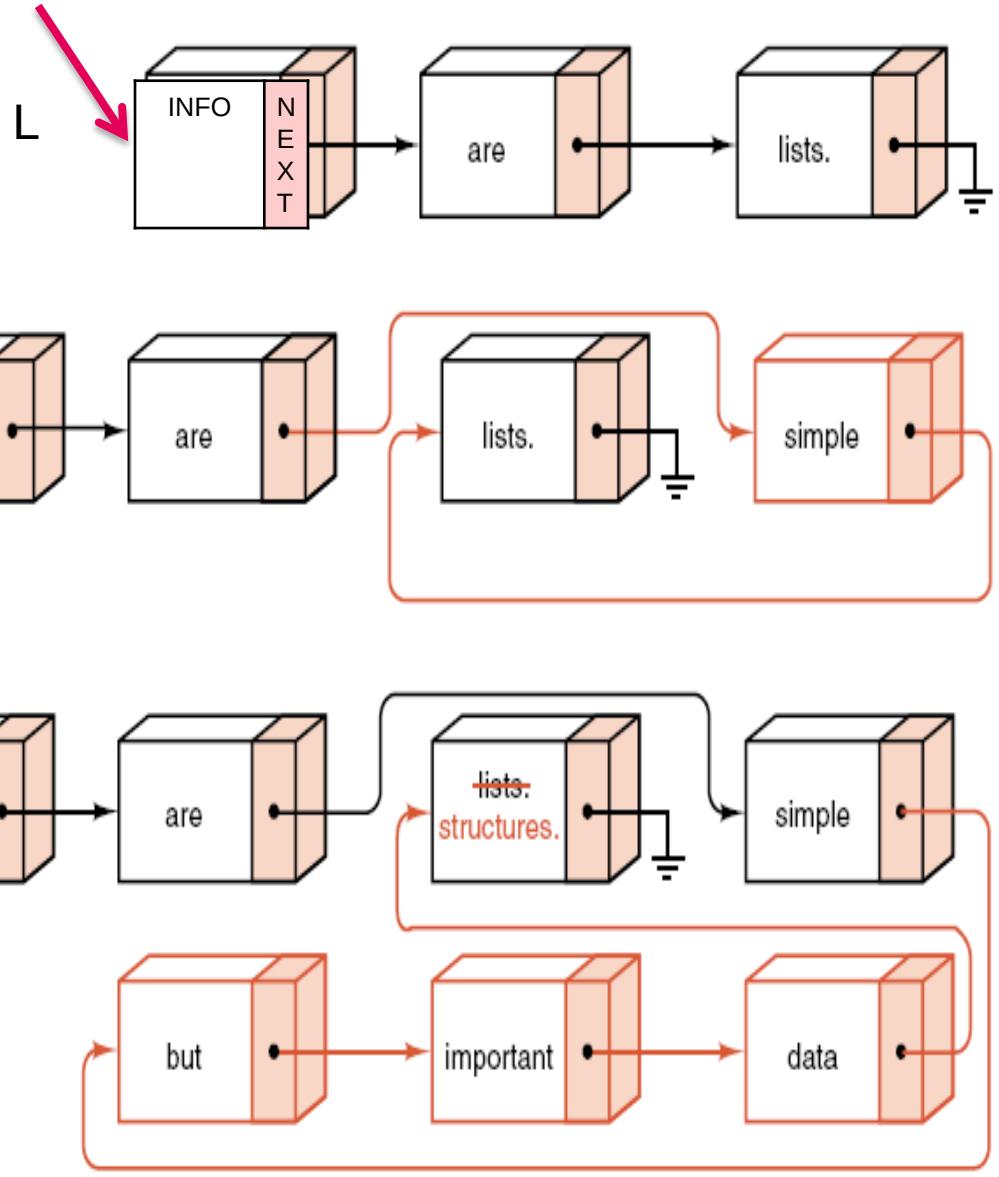
```
Algorithm Traverse
    Input: hàm visit dùng để tác động vào từng phần tử
    Output: danh sách được cập nhật bằng hàm visit

  //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
- Thành phần một nút:
  - *INFO: chứa thông tin (nội dung, giá trị) ứng với phần tử*
  - *NEXT: chứa địa chỉ của nút tiếp theo*

- Cần nắm được địa chỉ của nút đầu tiên trong danh sách

?



Danh sách
nối đơn

- 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

- 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.
- 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(int x)
```

- 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 nút

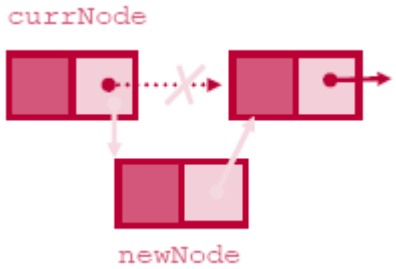
- Các trường hợp của thêm nút
  1. *Thêm vào danh sách rỗng*
  2. *Thêm vào đầu danh sách*
  3. *Thêm vào cuối danh sách*
  4. *Thêm vào giữa danh sách*
- Thực tế chỉ cần xét 2 trường hợp
  - *Thêm vào đầu danh sách*
  - *Thêm vào giữa hoặc cuối danh sách*

?

Thêm nút

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

- Thêm một nút mới với dữ liệu là x vào sau nút thứ index
- 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 nút

```
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(Node **head, 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

Xóa nút

```
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;
    }
}
```

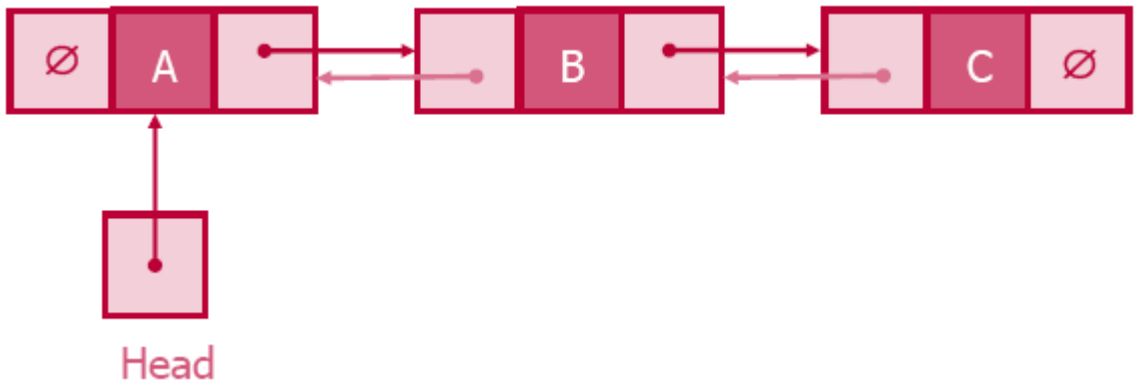
Mảng và
danh sách liên kết

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

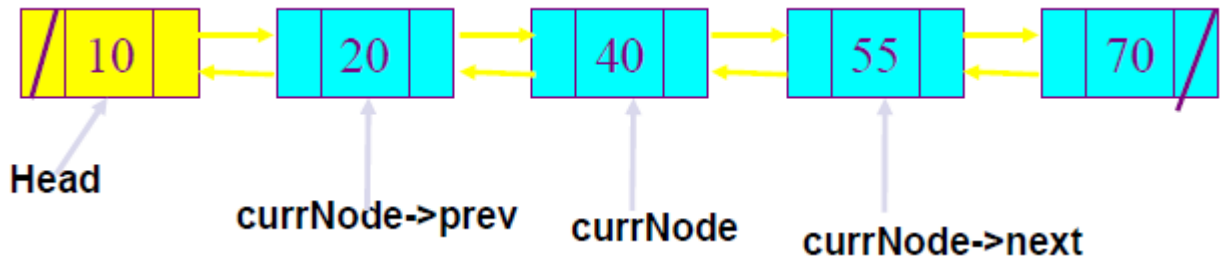
- Mỗi nút không chỉ nối đến nút tiếp theo mà còn nối đến nút trước nó
- Có 2 mối nối NULL: tại nút đầu và nút cuối của danh sách
- Ưu điểm: tại một nút có thể thăm nút trước nó một cách dễ dàng. Cho phép duyệt danh sách theo chiều ngược lại



Danh sách
nối kép

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

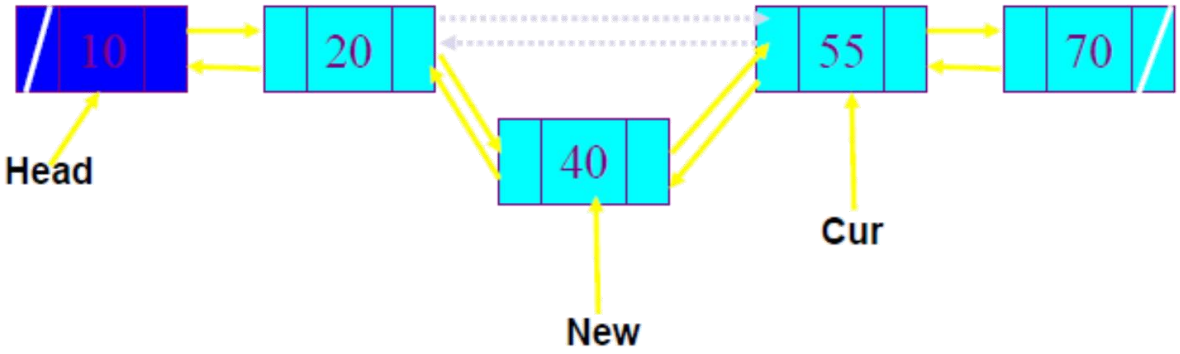
```
typedef struct Node {
   int data;
   struct Node *next;
   struct Node *prev;
} Node;
```



Danh sách
nối kép

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

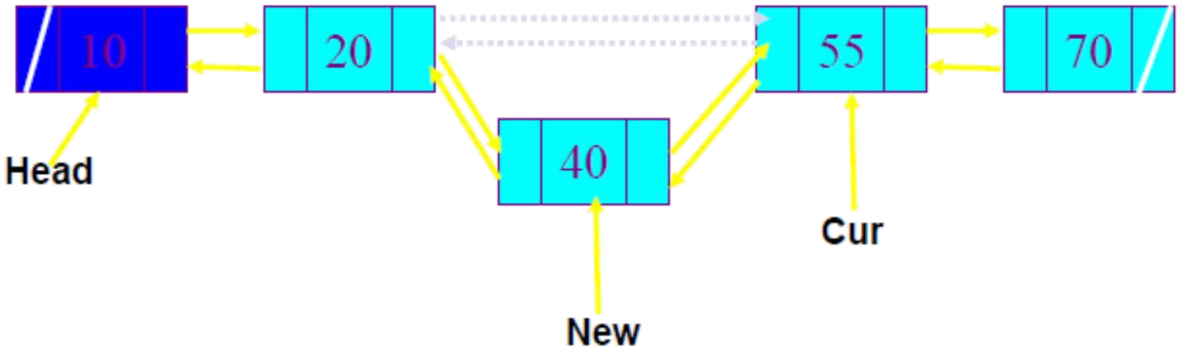
```
New->next = Cur;
New->prev= Cur->prev;
Cur->prev= New;
(New->prev)->next = New;
```



Danh sách
nối kép

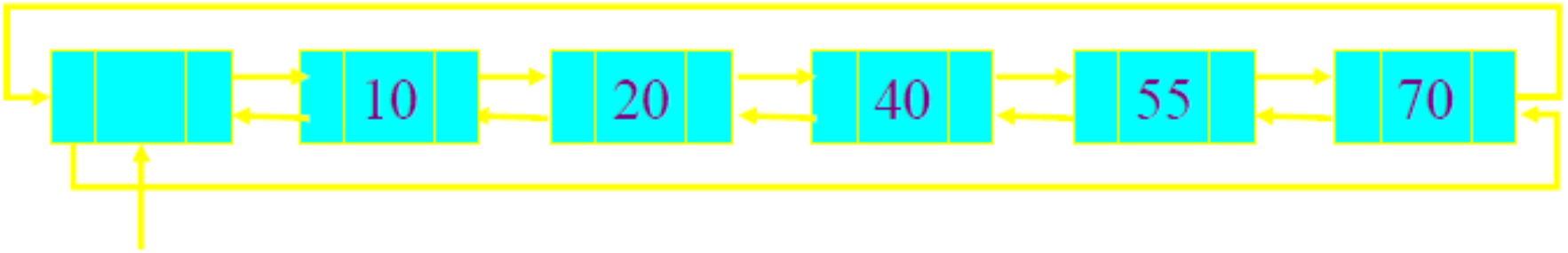
- Xóa nút Cur(không phải nút đầu hoặc cuối danh sách)

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

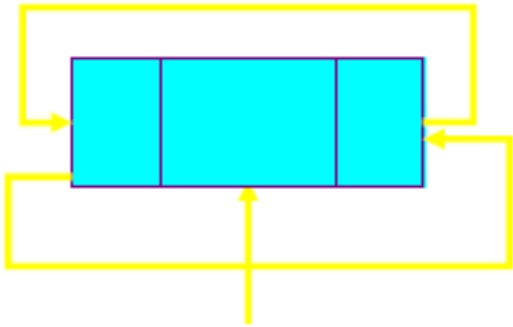


Danh sách nối kép với nút đầu giả

- Danh sách không rỗng



- Danh sách rỗng

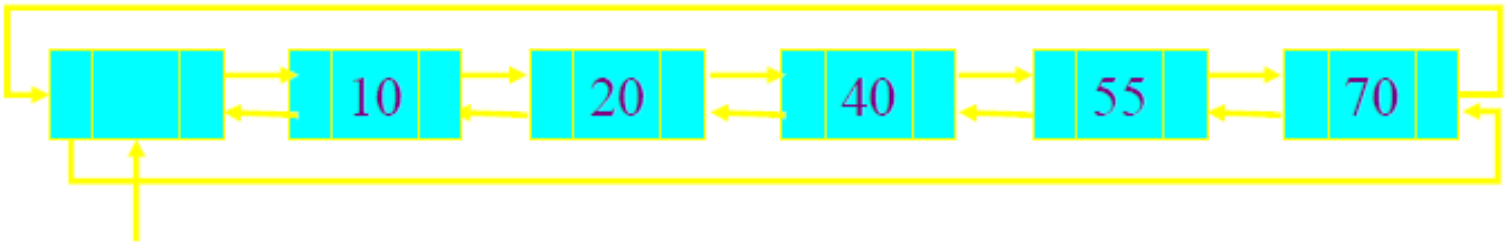


Danh sách nối kép với nút đầu giả

- 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?



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;
}
```

Xóa nút

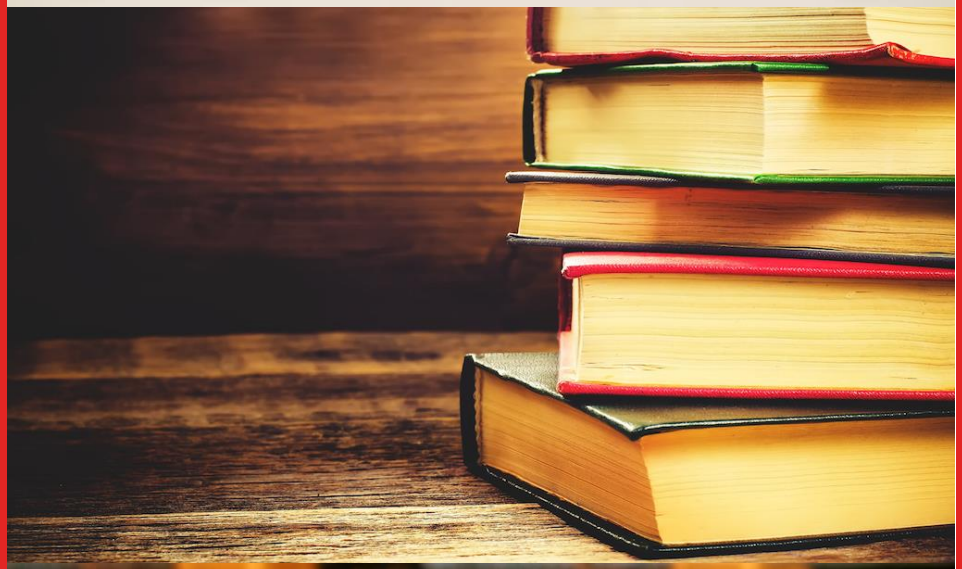
```
void deleteNode(Node *Head, int x){
      Node *Cur;
      Cur = FindNode(Head, x);
      if (Cur != NULL){
            Cur->prev->next = Cur->next;
            Cur->next->prev= Cur->prev;
            free(Cur);
      }
}
```

3.
Ngăn xếp

Stack

Ngăn xếp
Stack

danh sách mà xóa và thêm phần tử bắt buộc phải cùng được thực hiện tại một đầu duy nhất (đỉnh)



Thao tác trên ngăn xếp

Push

- Thêm một phần tử
- P: Tràn (overflow)

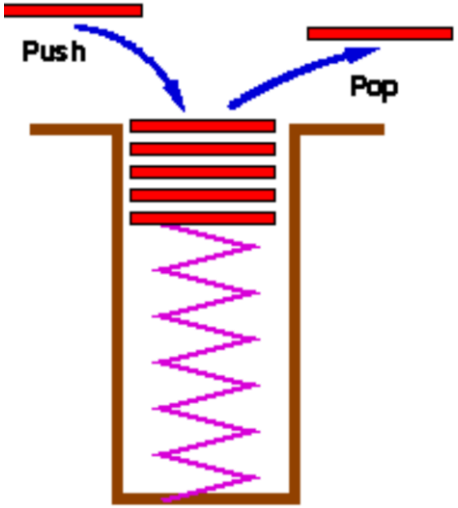
Pop

- Xóa một phần tử
- P: Underflow

Top

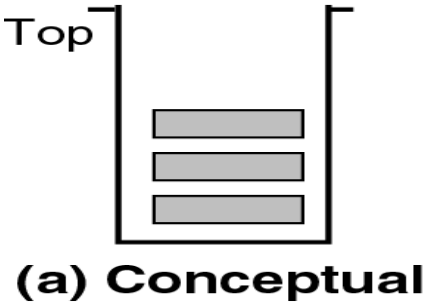
- Phần tử đỉnh
- P: Stack rỗng

Is Full / Is Empty

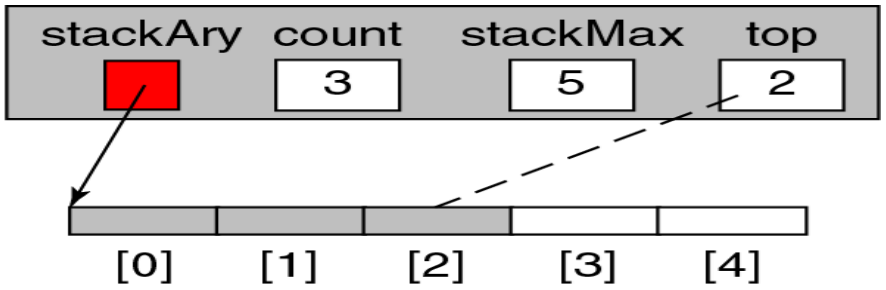


Lưu trữ
ngăn xếp

- **Lưu trữ kế tiếp**
  - *sử dụng mảng*
- **Lưu trữ móc nối**
  - *sử dụng danh sách móc nối (sau)*



(a) Conceptual



(b) Physical array

Cấu trúc dữ liệu
dùng mảng động

```
/* 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 ngăn xếp

```
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 ;
}
```

Thêm vào ngăn xếp
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;
}
```

Xóa khỏi ngăn xếp
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;
}
```

Kiểm tra rỗng đầy
Top, isEmpty, isFull

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
Bài toán đổi cơ số

Chuyển một số từ hệ thập phân sang hệ cơ số bất kỳ

- (cơ số 8) $28_{10} = 3 \times 8^1 + 4 \times 8^0 = 34_8$
- (cơ số 4) $72_{10} = 1 \times 4^3 + 0 \times 4^2 + 2 \times 4^1 + 0 \times 4^0 = 1020_4$
- (cơ số 2) $53_{10} =$

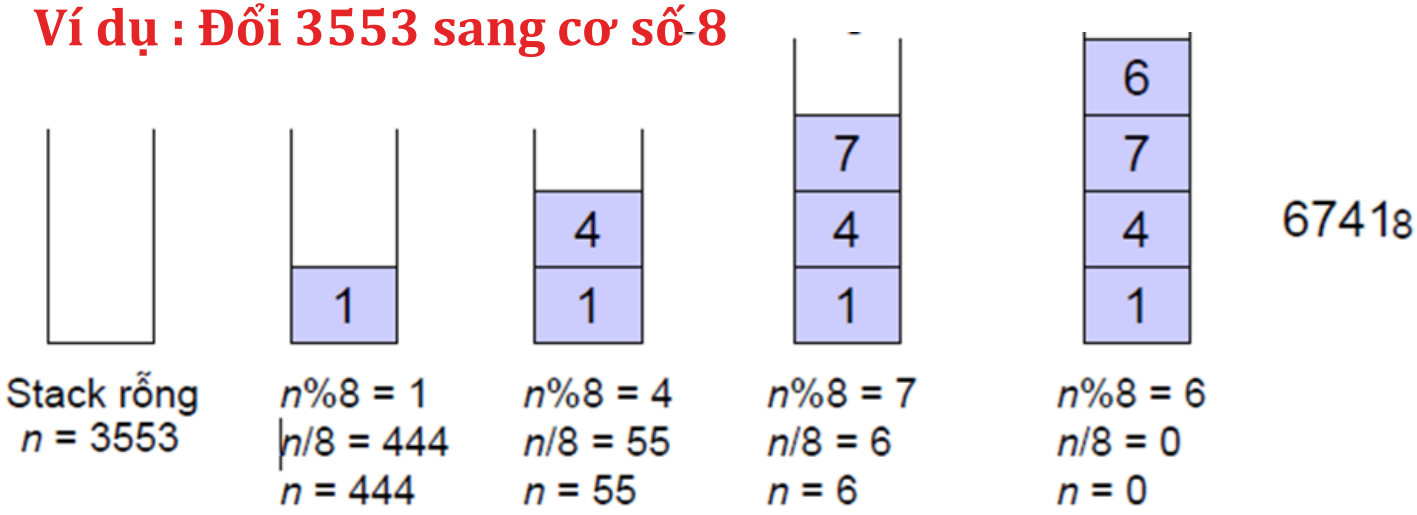
$1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 110101_2$

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



Ứng dụng
Bài toán đổi cơ số

- Đối với hệ cơ số 16
  - *Đổi sang ký tự tương ứng*

```
char *digitChar= “0123456789ABCDEF”;
char d = digitChar[13]; // 13₁₀= D₁₆
char f = digitChar[15]; // 13₁₀= F₁₆
```

```
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
Tính biểu thức dùng ký pháp hậu tố

- 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*

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

- Thứ tự ưu tiên của các toán tử

() > ^ > * = % = / > + = –

Ứng dụng
Tính biểu thức dùng ký pháp hậu tố

Ký pháp hậu tố

- 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 ^ –*

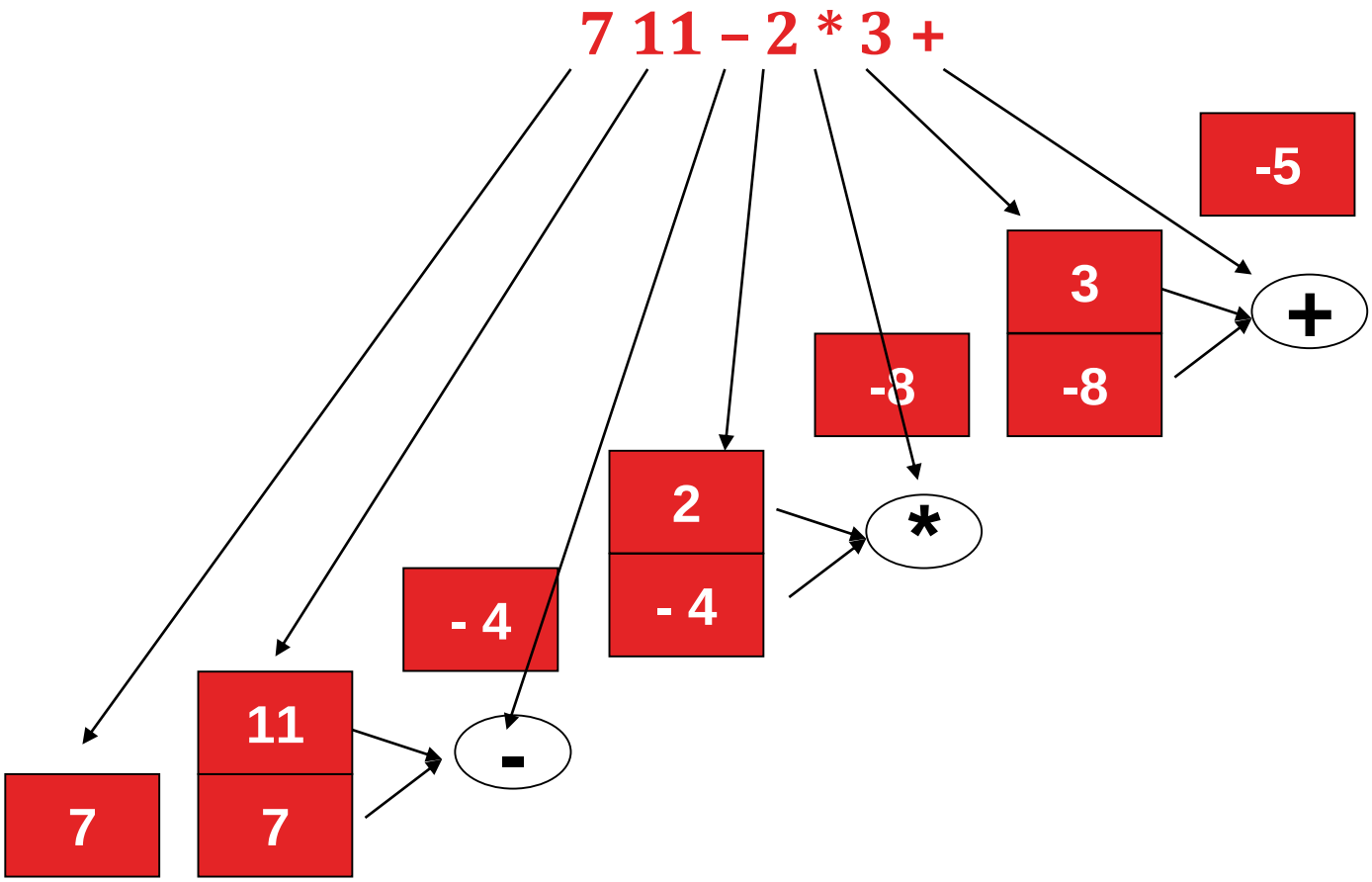
(x*y*z –x^2 / (y*2 –z^3) + 1/z) * (x –y) =>

xy*z*x2^y2*z3^ –/ –1z/+xy –*

Ví dụ

(7 –11) * 2 + 3

Biểu thức hậu tố:



Ứng dụng
Tính biểu thức dùng ký pháp 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
- Quy ước
  - ***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;
}
```

4.
Hàng đợi

Queue

Hàng đợi
Queue

danh sách mà việc thêm phải được thực hiện tại một đầu còn xóa phải thực hiện tại đầu kia.

Thao tác trên hàng đợi

Enqueue

- Thêm một phần tử
- P: Tràn (overflow)

Front

- Phần tử đầu
- P: Queue rỗng

Dequeue

- Xóa một phần tử
- P: Underflow

Rear

- Phần tử cuối
- P: Queue rỗng

Hàng đợi
Queue

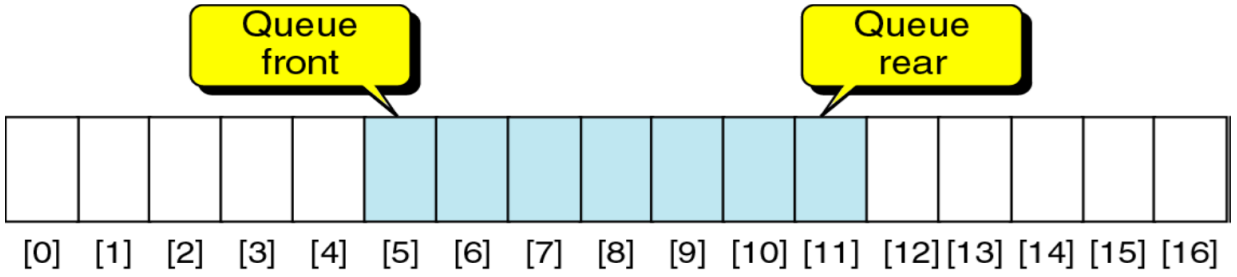
- Phần tử đầu hàng sẽ được phục trước, phần tử này được gọi là **front**, hay **head** của hàng. Tương tự, phần tử cuối hàng , cũng là phần tử vừa được thêm vào hàng, được gọi là **rear** hay **tail** của hàng.
- Biểu diễn hàng đợi
  - ***Mô hình vật lý***
  - ***Hiện thực tuyến tính***
  - ***Hiện thực dãy vòng***

Mô hình vật lý

- Dùng **mảng**. Phải nắm giữ cả **front** và **rear**.
- Giữ front luôn là vị trí đầu của dãy.
  - *Thêm phần tử vào hàng ⇒ thêm vào cuối dãy.*
  - *Lấy ra 1 phần tử ra ⇒ dịch chuyển tất cả các phần tử của dãy lên 1 vị trí.*
- Tương tự hình ảnh hàng đợi trong thực tế
  - *Không phù hợp với lưu trữ trong máy tính*

Mô hình
hiện thực tuyến tính

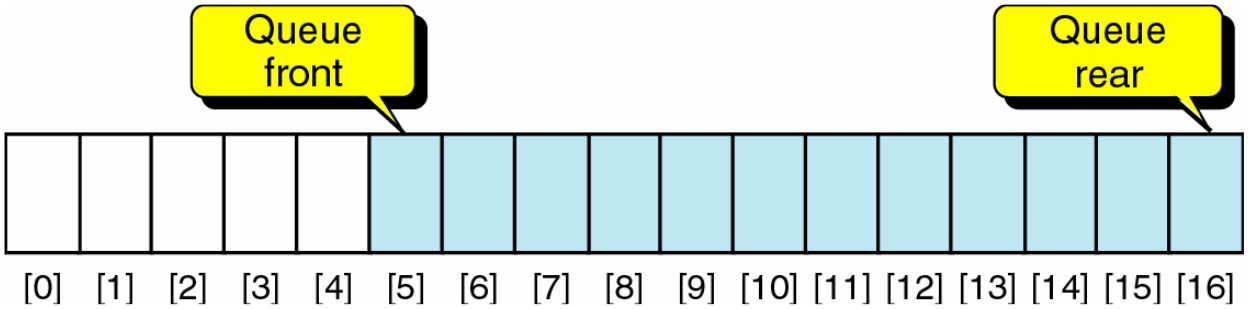
- Dùng 2 chỉ số **front** và **rear** để lưu trữ đầu và cuối hàng mà không di chuyển các phần tử.
  - *Thêm phần tử vào hàng ⇒ Tăng rear lên 1 và thêm phần tử vào vị trí đó*
  - *Lấy ra 1 phần tử ra ⇒ Lấy phần tử tại front và tăng front lên 1*
- Nhược điểm: front và rear chỉ tăng mà không giảm ⇒ lãng phí bộ nhớ
  - *Khắc phục: Khi hàng đợi rỗng thì ta gán lại front=rear=đầu dãy*

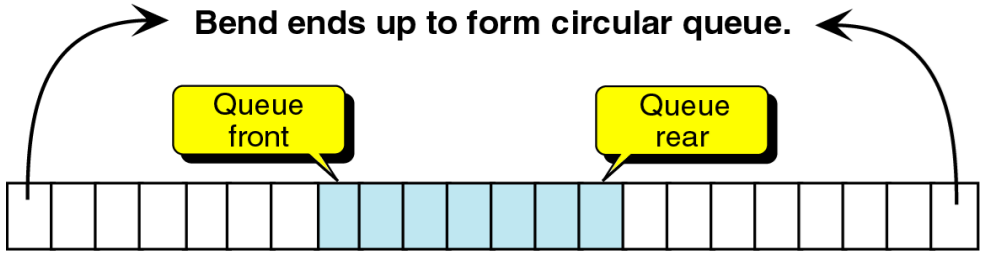


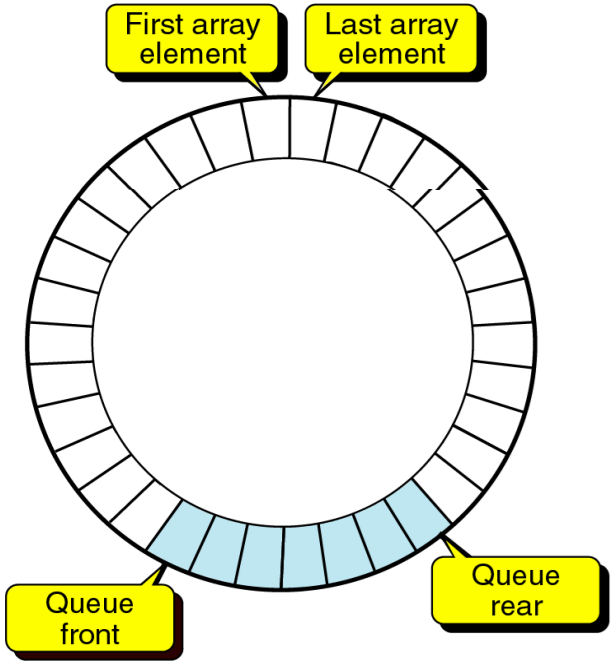
Mô hình
hiện thực dãy vòng

- 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.*
    - Tương tự với việc cộng thêm giờ trên đồng hồ mặt tròn

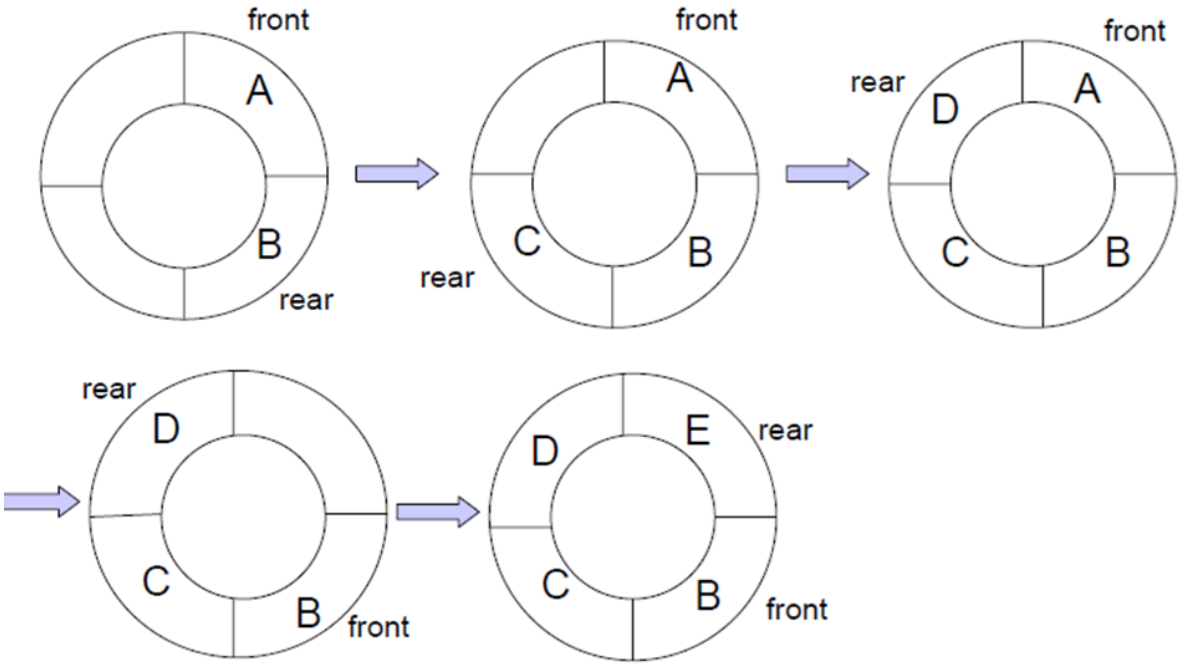
```
i = ((i+1) == max) ? 0: (i+1);
```
hoặc
```
if ((i+1) == max) i = 0; else i = i+1;
```
hoặc
```
i = (i+1) % max;
```

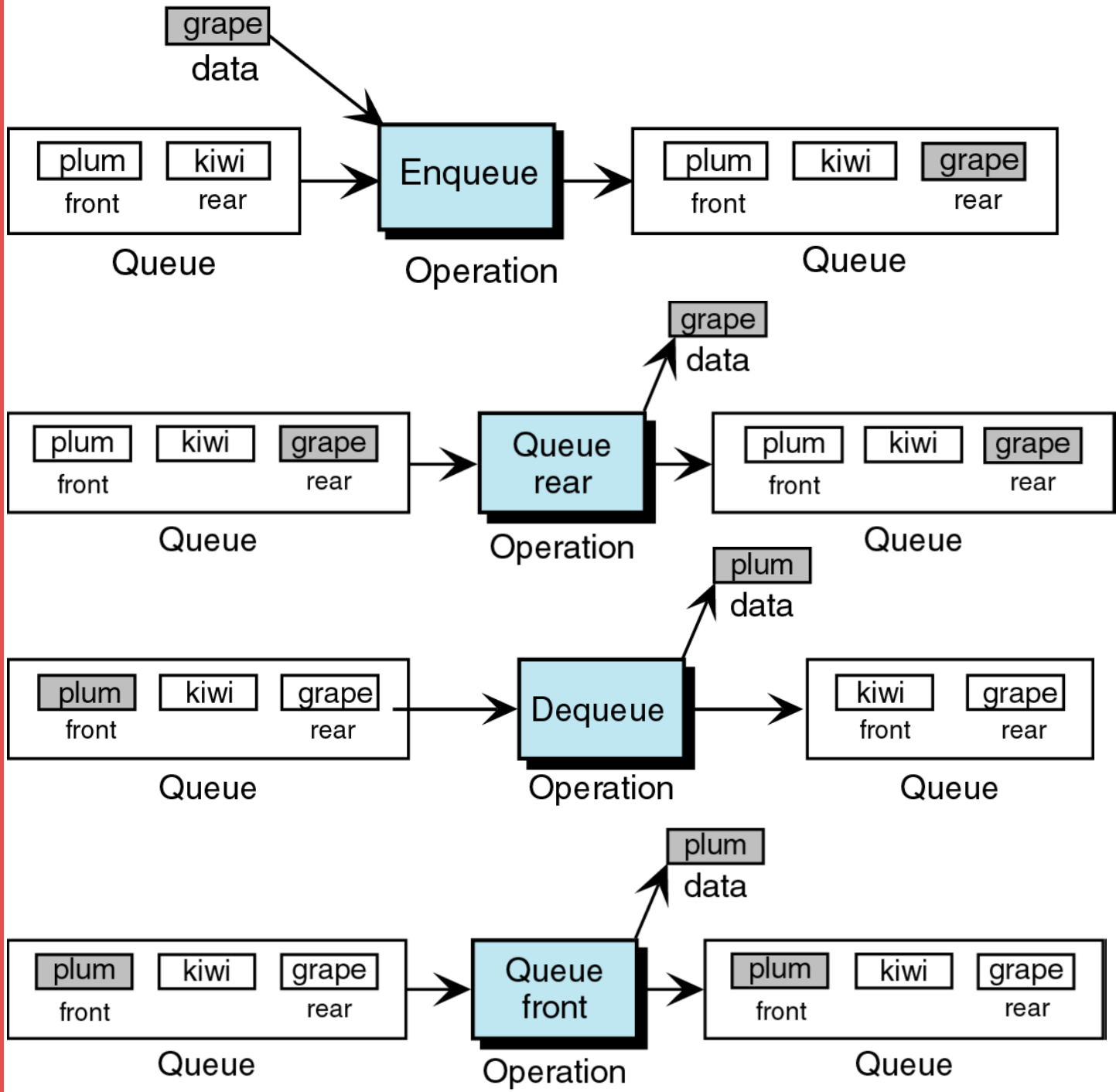

Queue front
Queue rear
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16]


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


First array element
Last array element
Queue front
Queue rear

Mô hình
hiện thực dãy vòng




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

Cấu trúc dữ liệu
dùng mảng động

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

Tạo
hàng đợi

```
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 hàng đợi

```
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 hàng đợi

```
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

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

Lấy phần tử cuối

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

Kiểm tra rỗng và đầy

- Empty

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

- Full

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

Xóa hàng đợi

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

5.
Cây

Tree

Cây
Tree

- Nhược điểm của danh sách: Tính tuần tự, chỉ thể hiện được các mối quan hệ tuyến tính.
- Cây được sử dụng để lưu trữ các thông tin phi tuyến như
  - *Các thư mục file*
  - *Các bước di chuyển của các quân cờ*
  - *Sơ đồ nhân sự của tổ chức*
  - *Cây phả hệ*

Cây
Tree

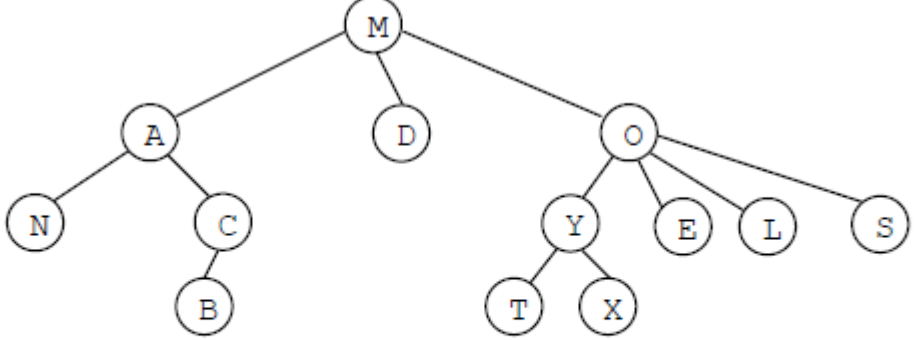
*Một **cây** (tree) gồm một tập hữu hạn các **nút** (node) và 1 tập hữu hạn các **cành** (branch) nối giữa các nút.*

Định nghĩa đệ quy

*Một **cây** là tập các **nút***

- *là tập rỗng, hoặc*
- *có 1 nút gọi là nút gốc có 0 hoặc nhiều cây con, các cây con cũng là cây*

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)

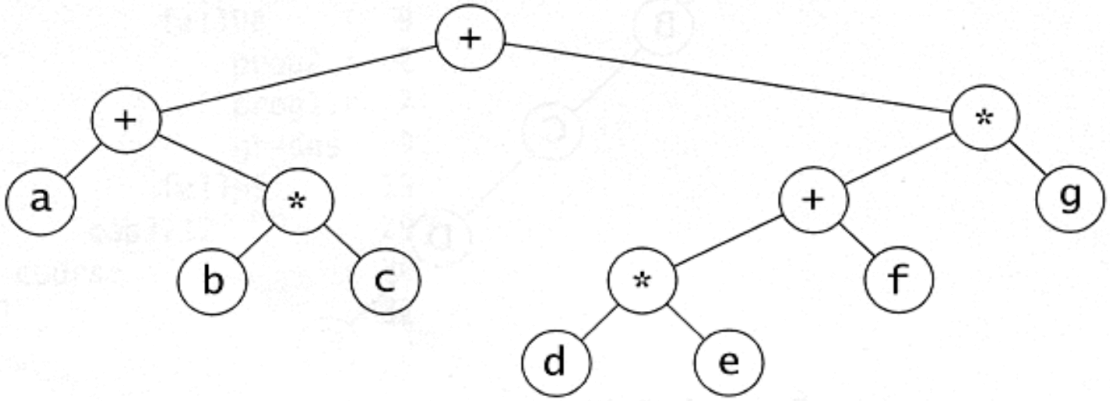
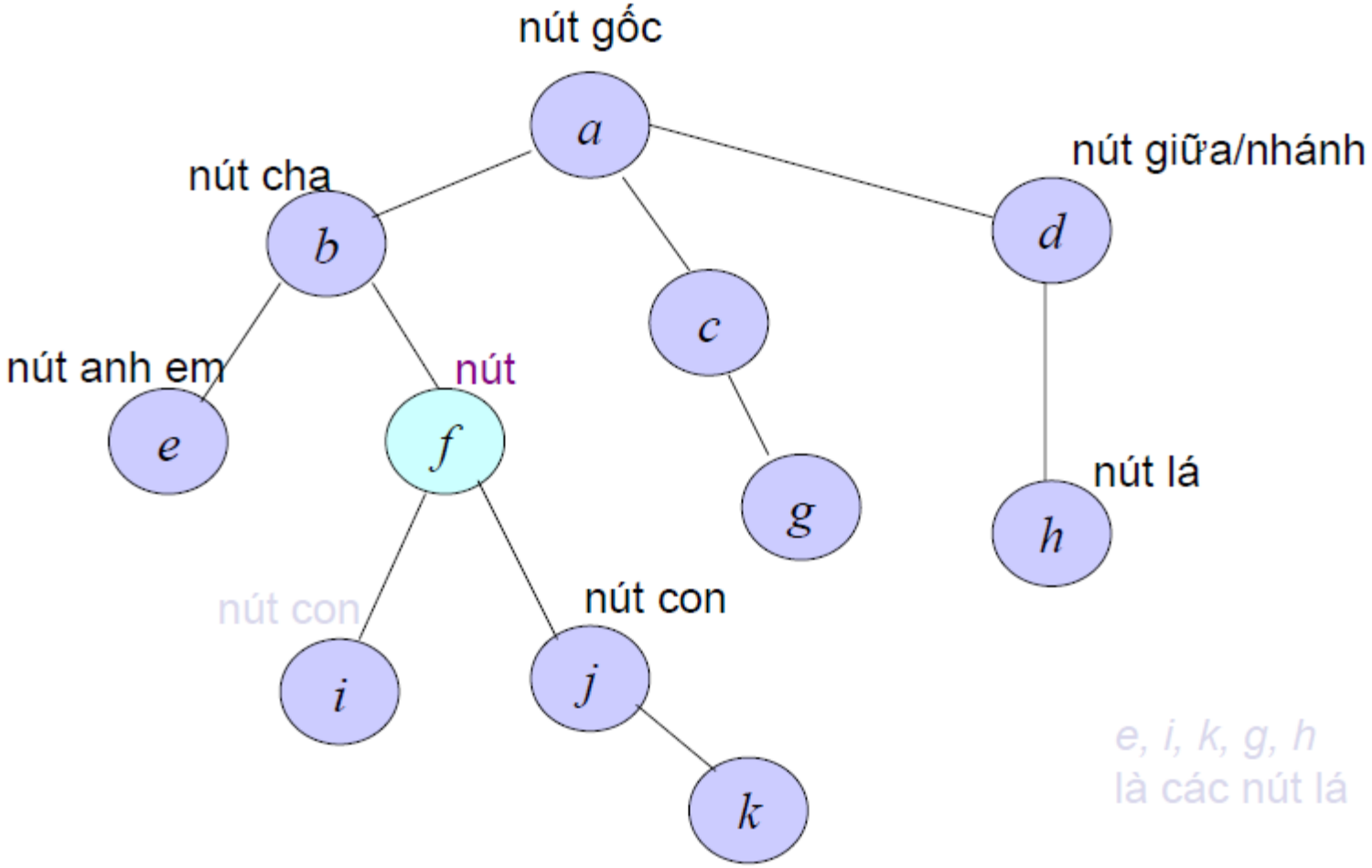


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

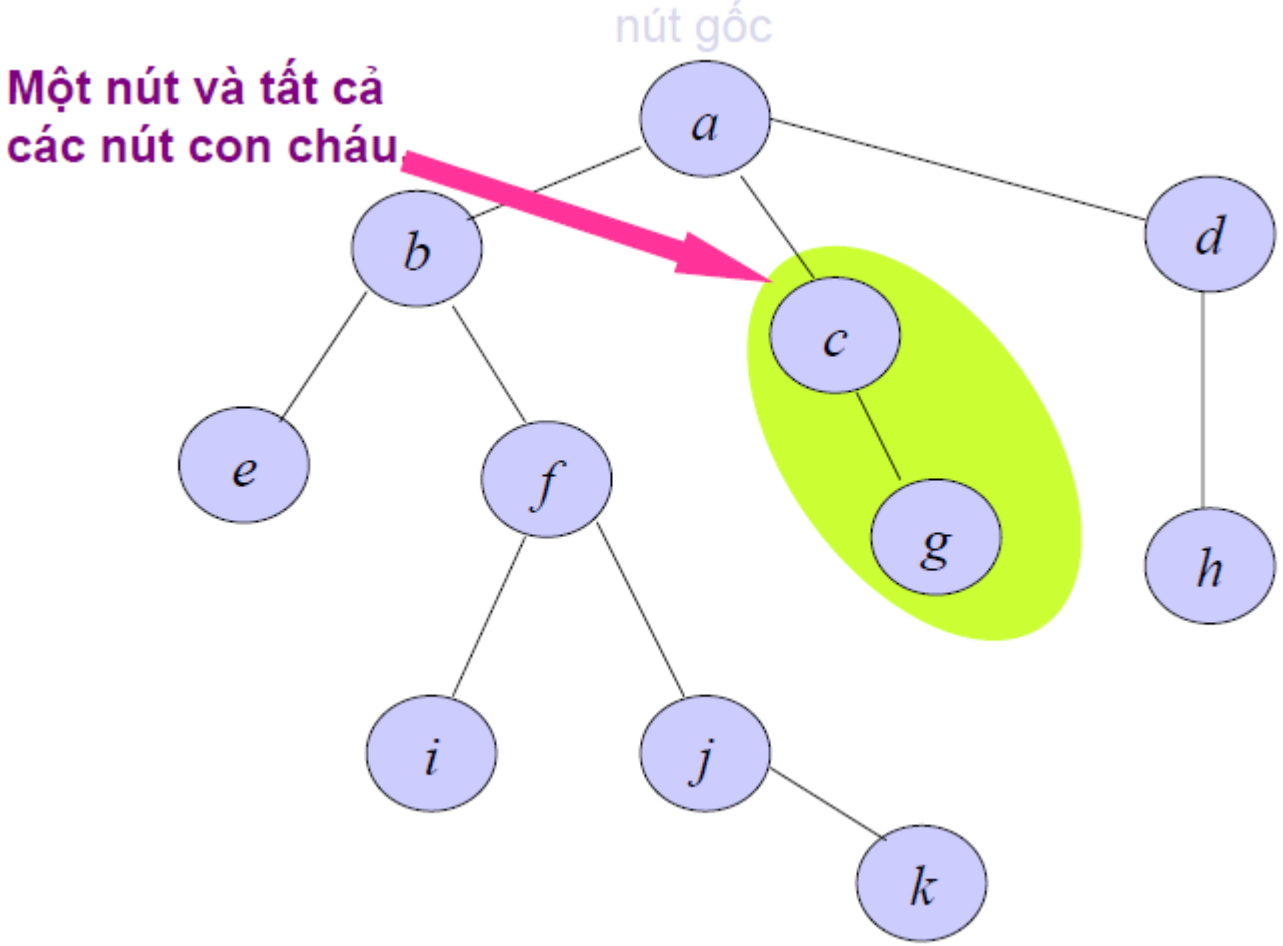
Các khái niệm cơ bản

- 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

Các khái niệm cơ bản



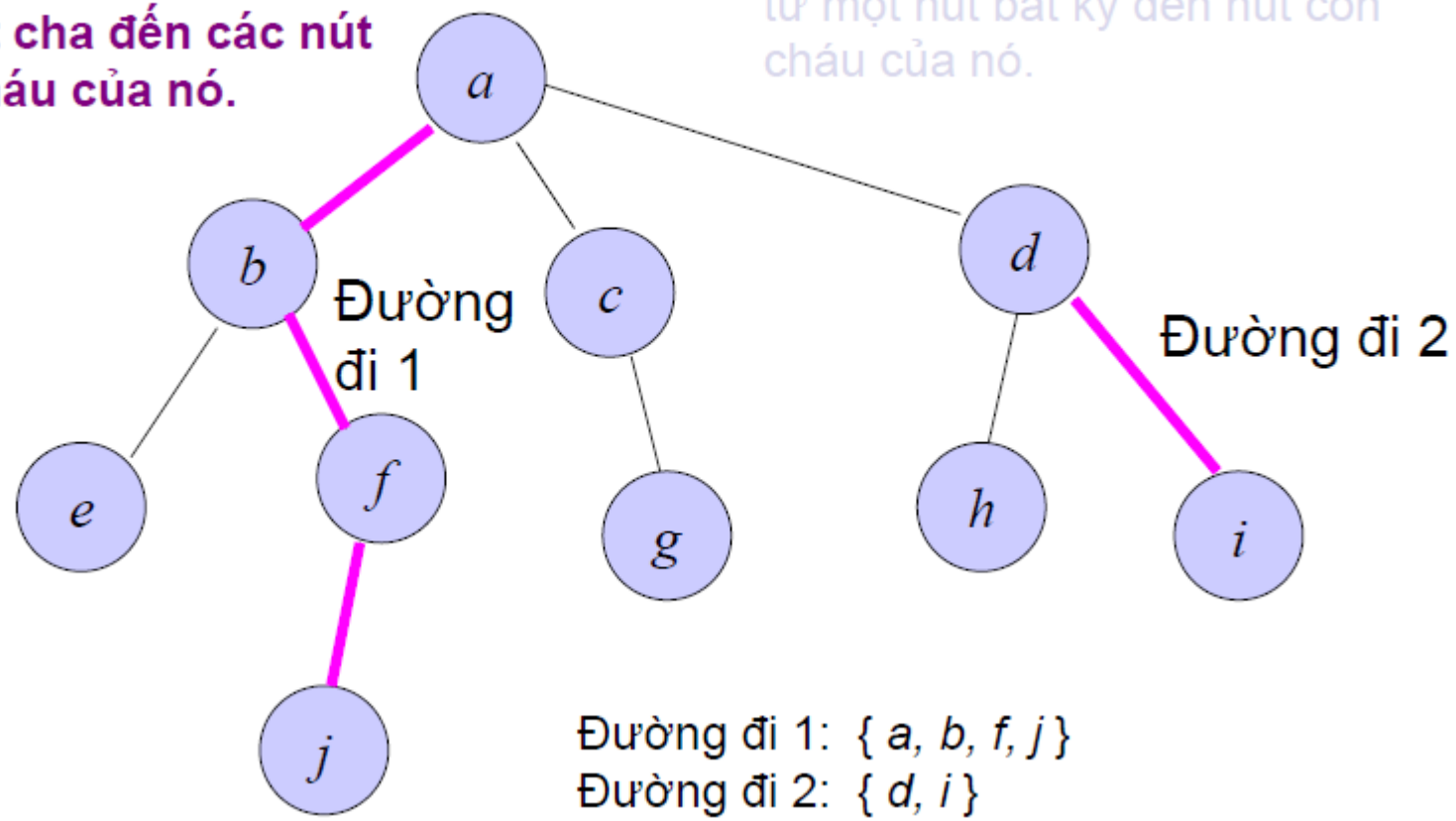
Cây con



Đường đi

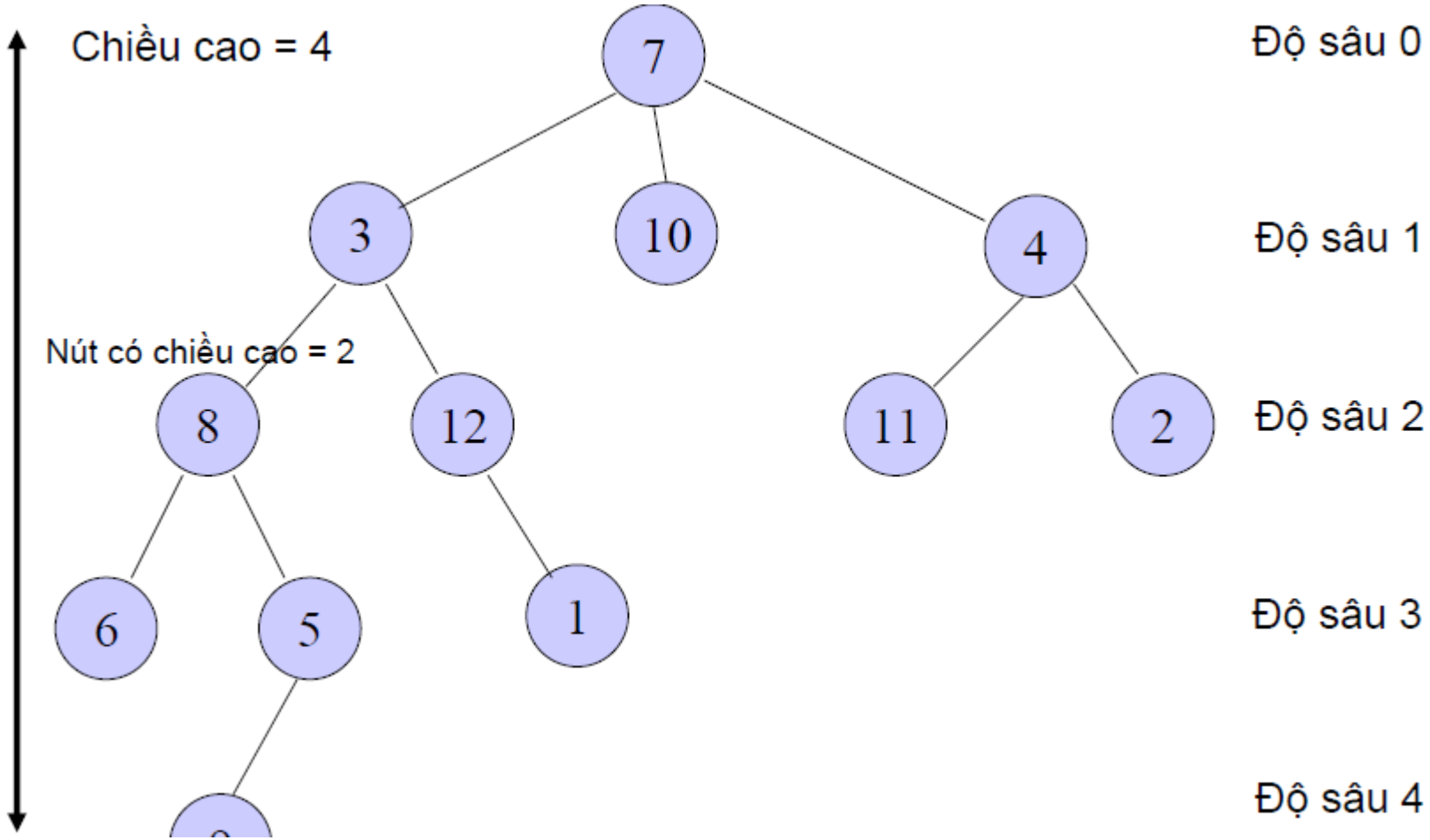
Tồn tại một đường đi duy nhất từ một nút bất kỳ đến nút con cháu của nó.

Từ nút cha đến các nút con cháu của nó.



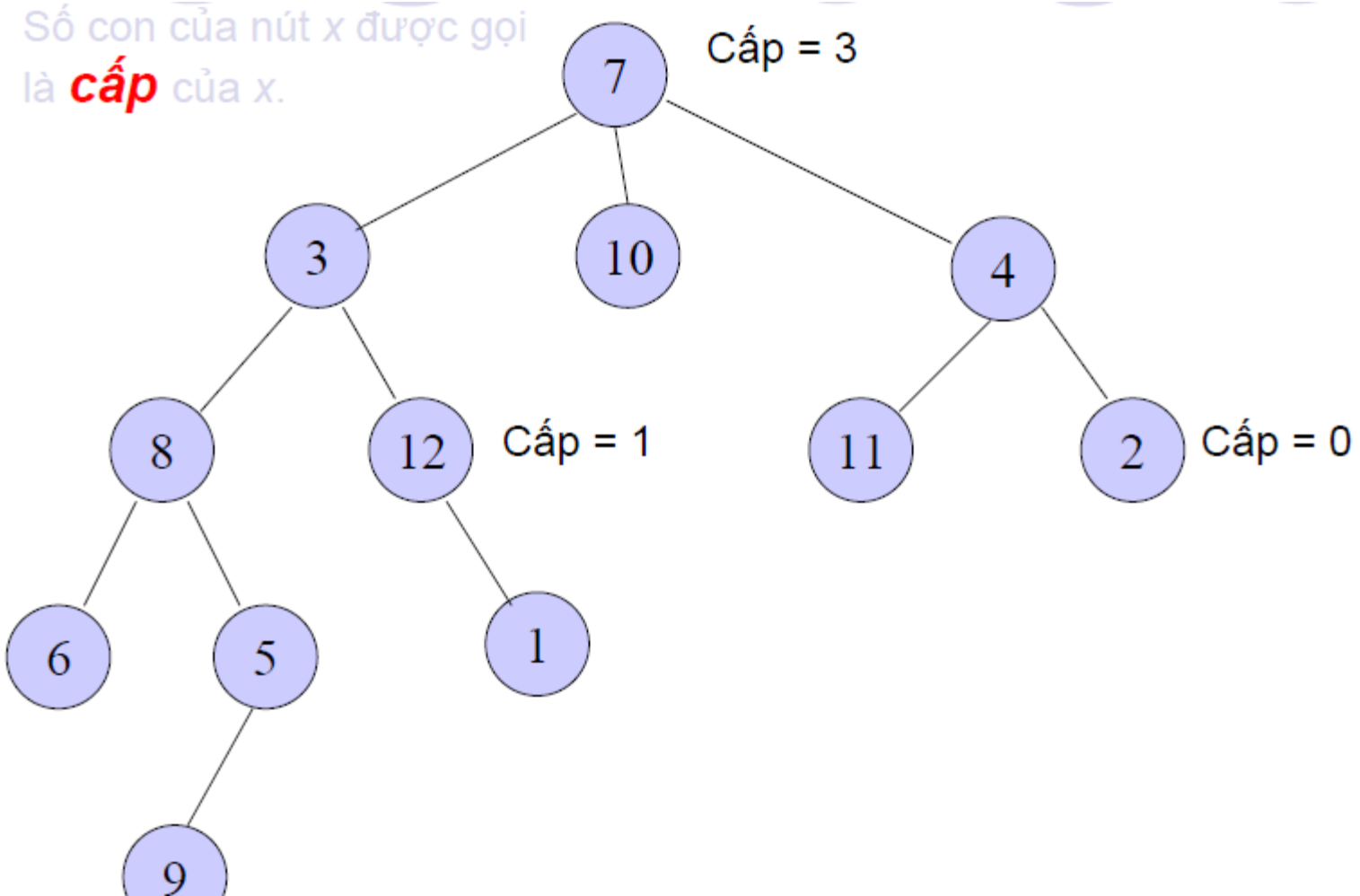
Đường đi 1: { *a*, *b*, *f*, *j* }
Đường đi 2: { *d*, *i* }

Độ sâu và chiều cao



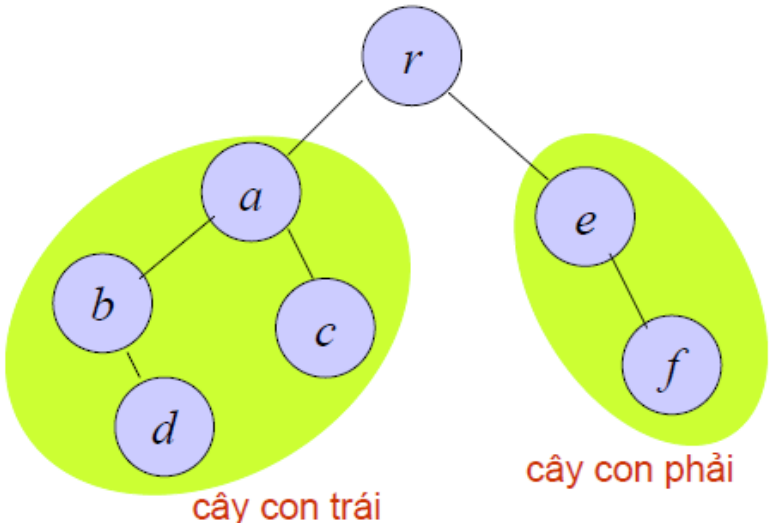
Cấp

Số con của nút *x* được gọi là ***cấp*** của *x*.



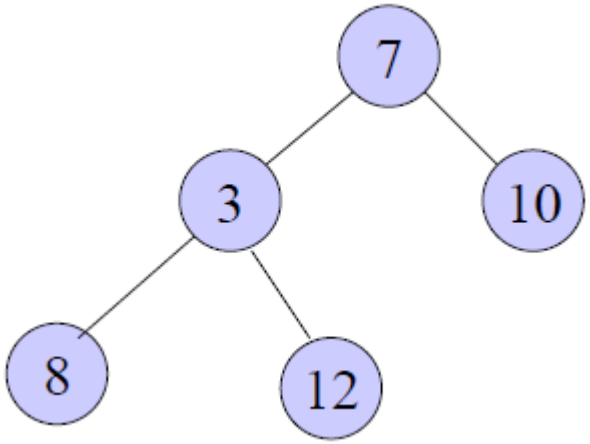
Cây
nhị phân

- 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

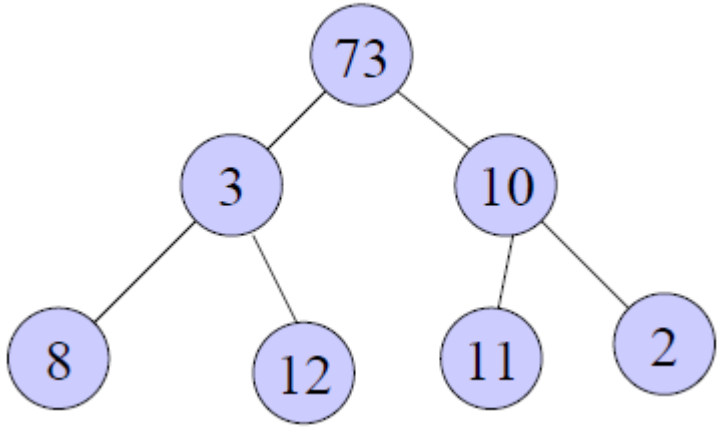


Cây nhị phân đầy đủ và cây nhị phân hoàn chỉnh

- Cây nhị phân đầy đủ
  - *Cây nút trừ 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*



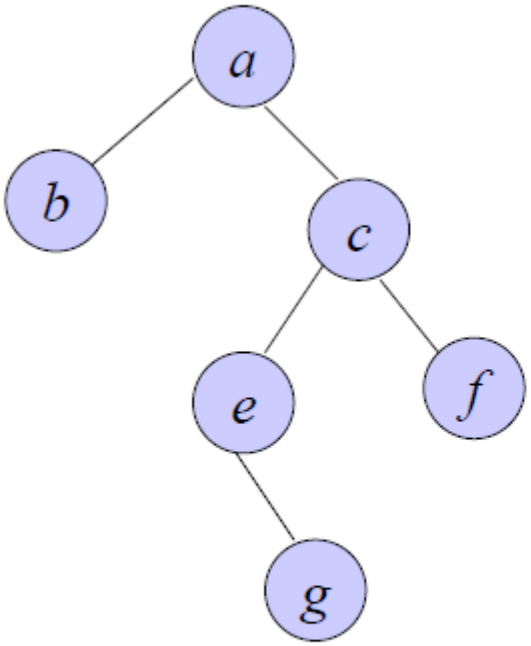
Cây cân bằng

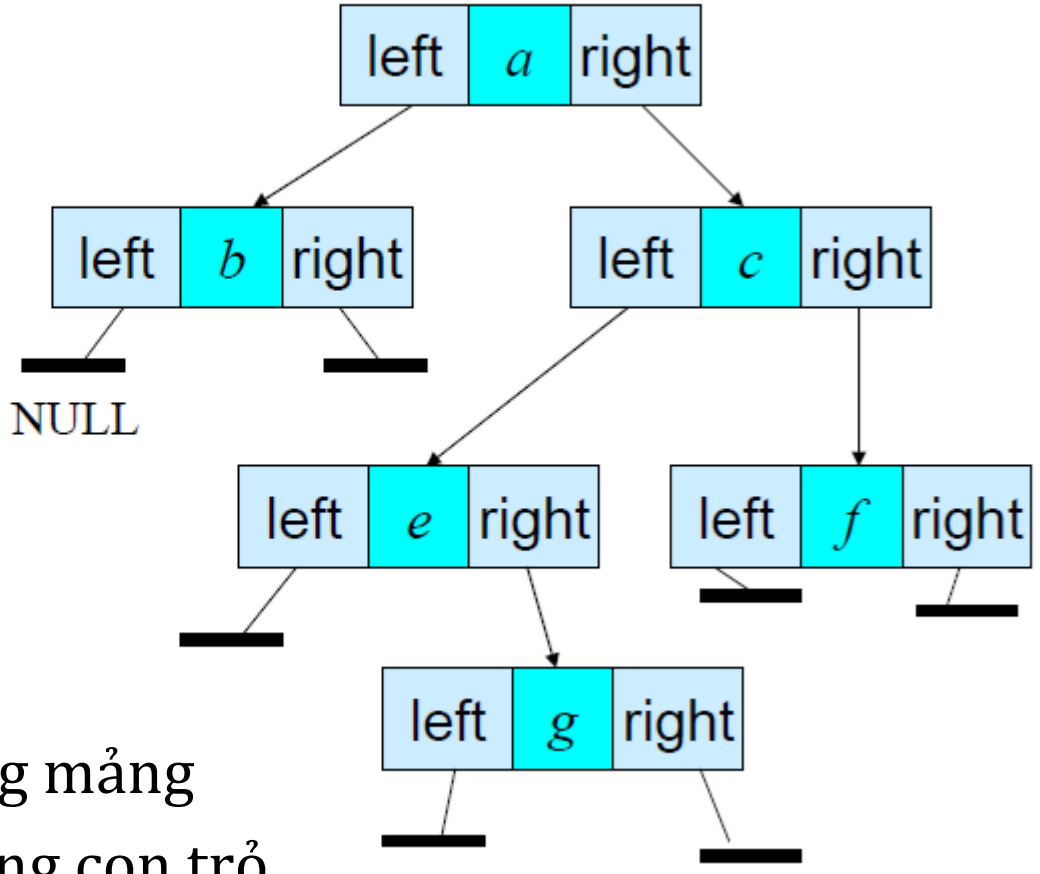
- Khoảng cách từ 1 nút đến nút gốc xác định chi phí cần để định vị nó:
  - *1 nút có độ sâu là 5 ⇒ phải đi từ nút gốc và qua 5 cành*
- Nếu cây càng thấp thì việc tìm đến các nút sẽ càng nhanh.
- Hệ số cân bằng của cây (*balance factor*) là số chênh lệch giữa chiều cao của 2 cây con trái và phải của nó:

$$B = HL - HR$$

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

Lưu trữ
cây nhị phân





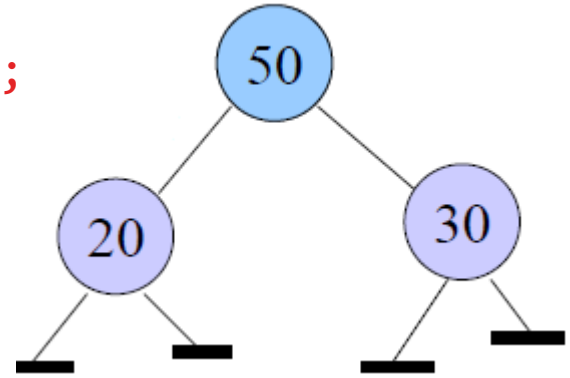
- Lưu trữ kế tiếp: Sử dụng mảng
- Lưu trữ móc nối: Sử dụng con trỏ

Cấu trúc dữ liệu
dùng mảng

```
typedef struct tree_node
{
    int data ;
    struct tree_node *left ;
    struct tree_node *right ;
}TREE_NODE;
```

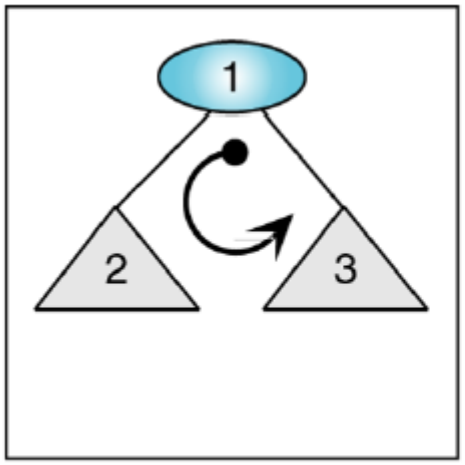
Tạo
cây nhị phân

```
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
```

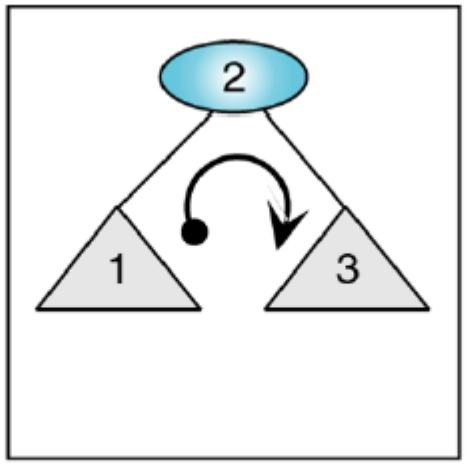


Duyệt
cây nhị phân

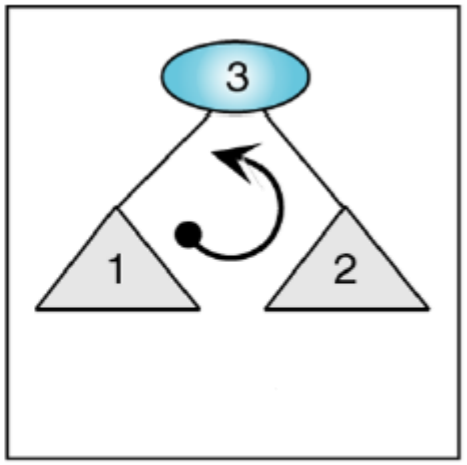
- Duyệt theo thứ tự trước
- Duyệt theo thứ tự giữa
- Duyệt theo thứ tự sau



(a) Thứ tự trước



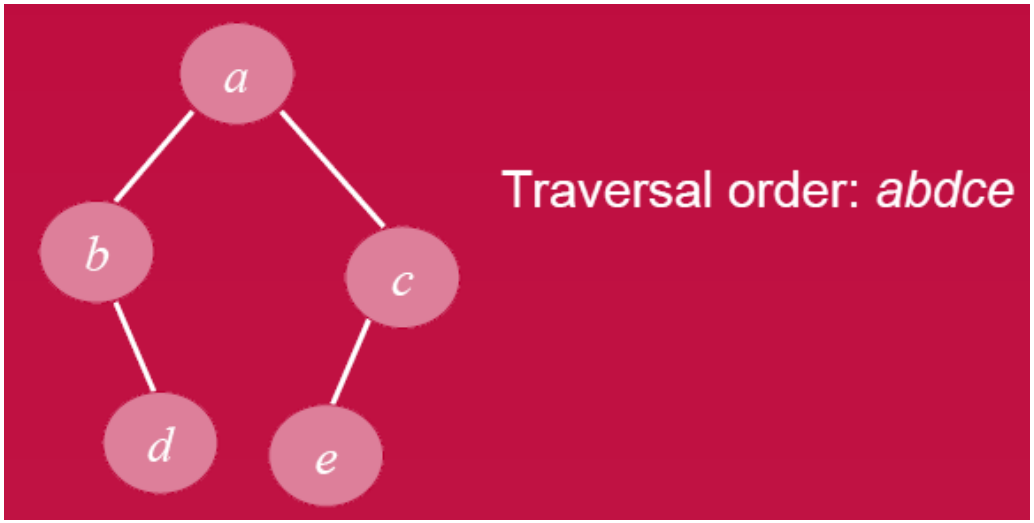
(b) Thứ tự giữa



(c) Thứ tự sau

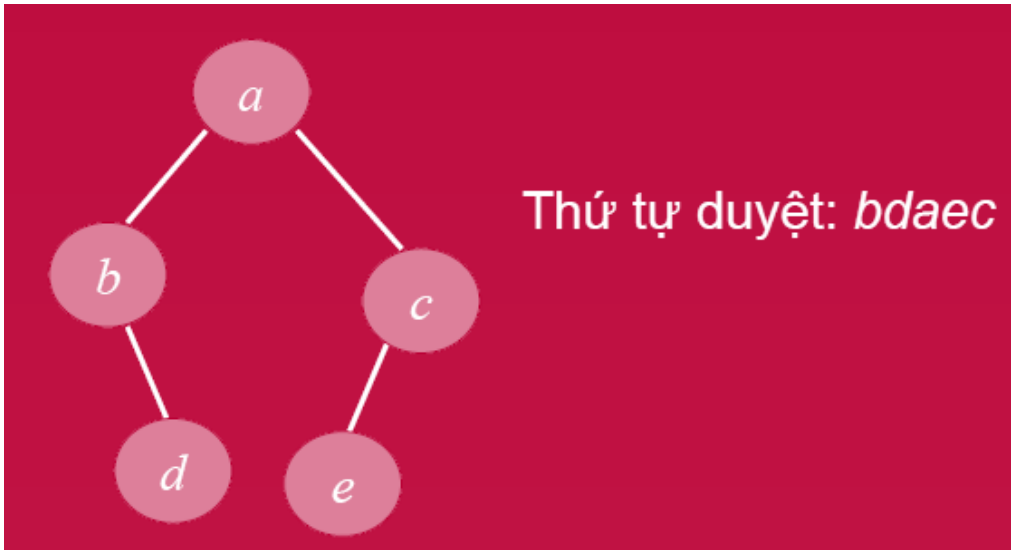
Duyệt theo thứ tự trước

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



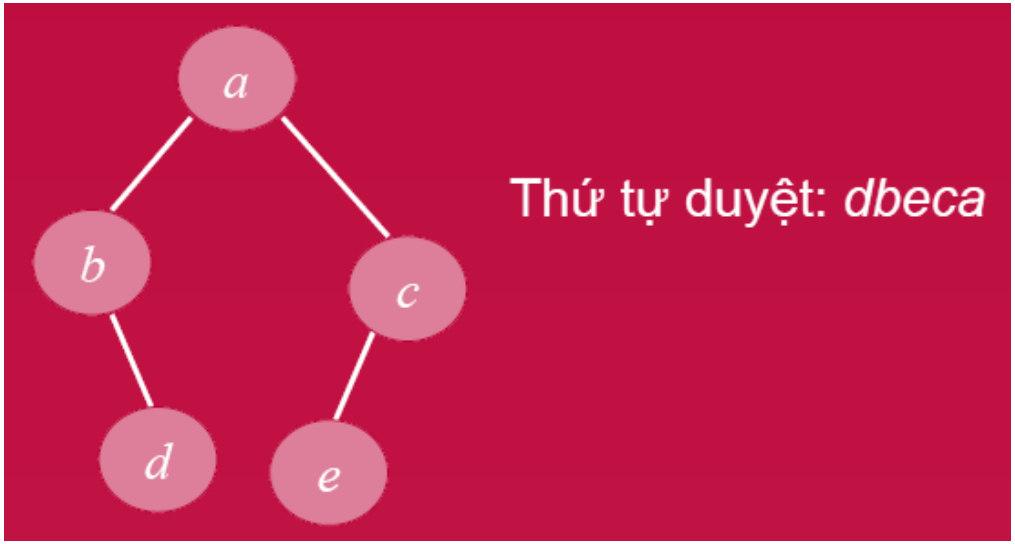
Duyệt theo thứ tự giữa

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



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 trước

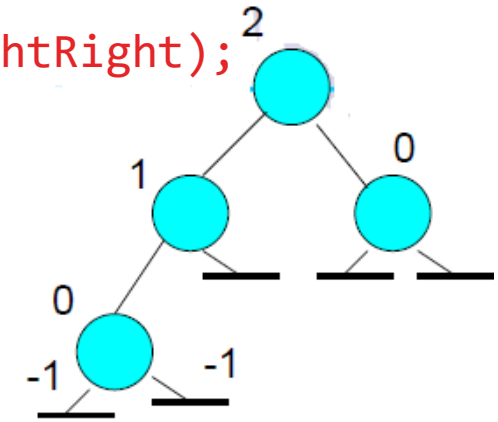
```
void Preorder(TREE_NODE *root)
{
	if (root != NULL)
	{
		// tham node
		printf("%d", root->data);
		// duyet cay con trai
		Preorder(root->left);
		// duyet cay con phai
		Preorder(root->right);
	}
}
```

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

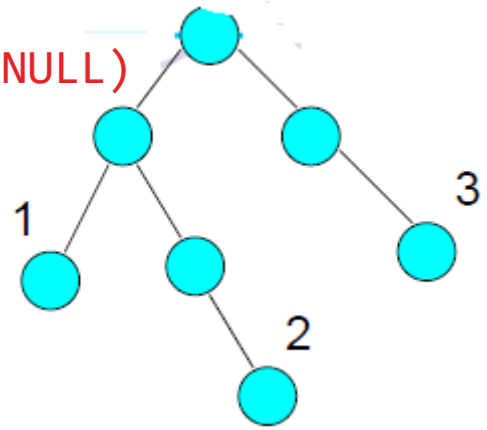
Tính độ cao của cây

```
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

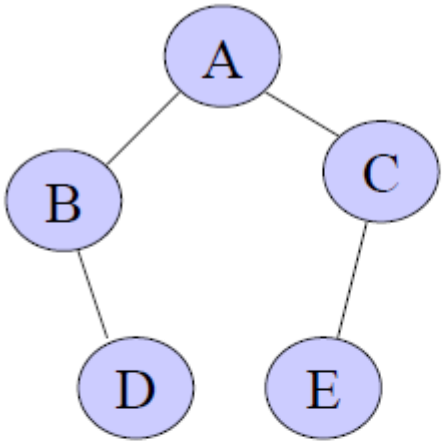
```
int CountLeaf(TREE_NODE *tree)
{
        if (tree == NULL)
                return 0;
        int count = 0;
        //Đếm theo thứ tự sau
        count += CountLeaf(tree->left);// Đếm trái
        count += CountLeaf(tree->right);//Đếm phải
        //nếu nút tree là nút lá, tăng count
        if(tree->left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

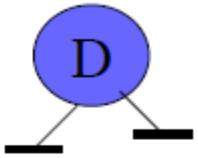


Tính kích thước

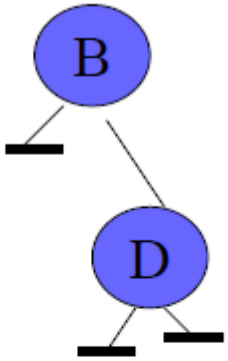
```
int TreeSize(TREE_NODE *tree)
{
        if(tree== NULL)
                return 0;
        else
                return( TreeSize(tree->left) +
                        TreeSize(tree->right) + 1 );
}
```

Sao chép
cây

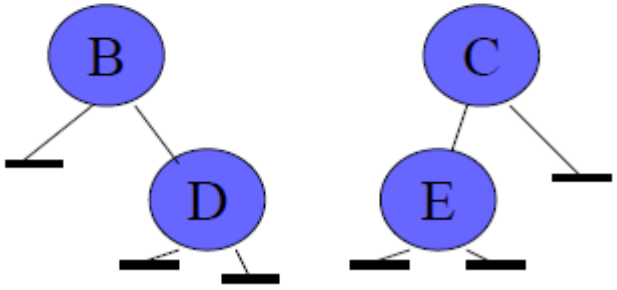




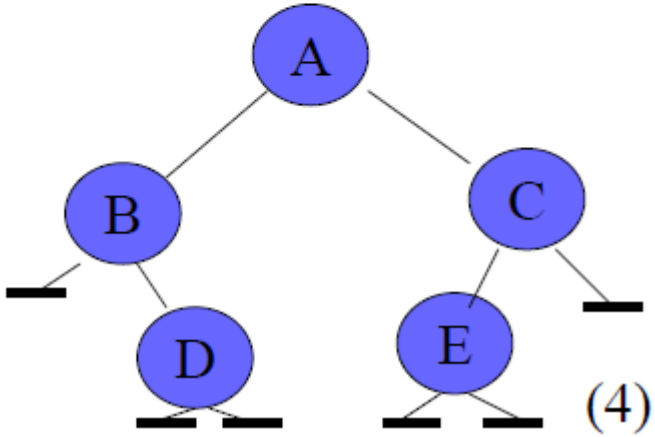
(1)



(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

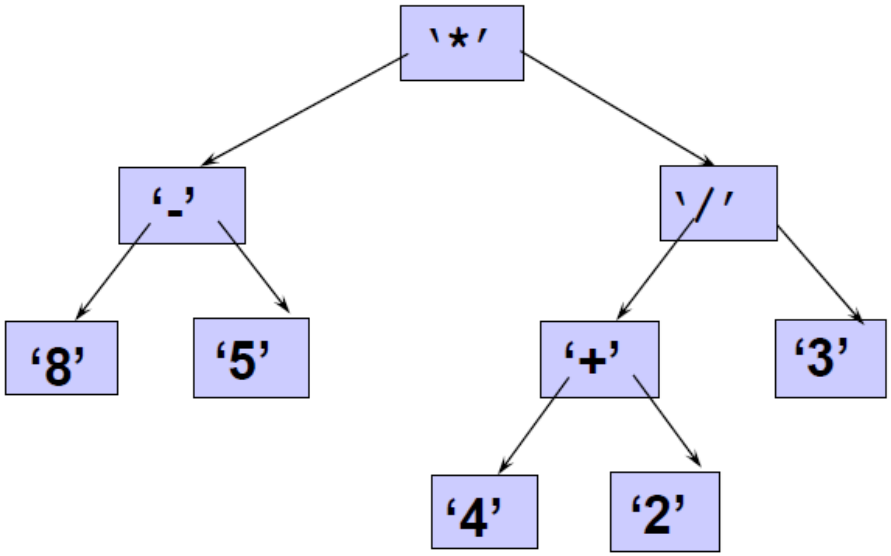
```
void DeleteTree(TREE_NODE *tree)
{
//xóa theo thứ tự sau
        if(tree != NULL)
        {
                DeleteTree(tree->left);
                DeleteTree(tree->right);
                free(tree);
        }
}
```

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*

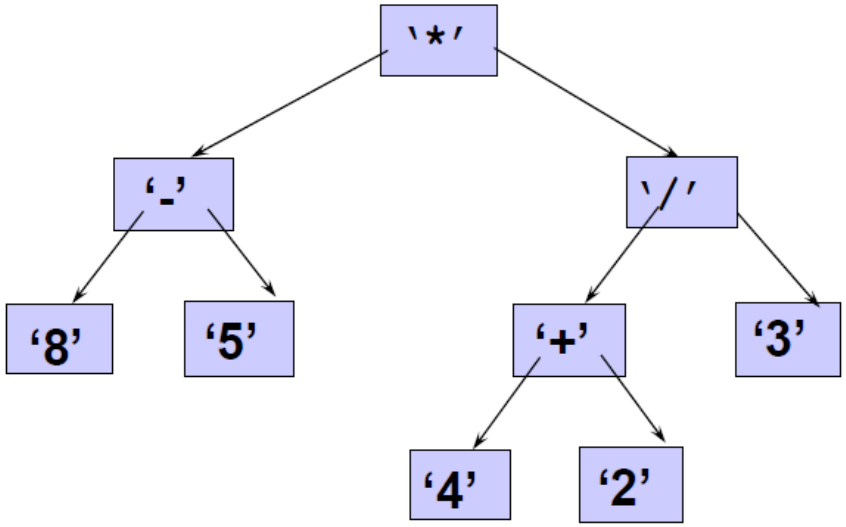


Cây biểu diễn biểu thức

- 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 thứ tự giữa, trước và sau)

Cài đặt
cây biểu thức

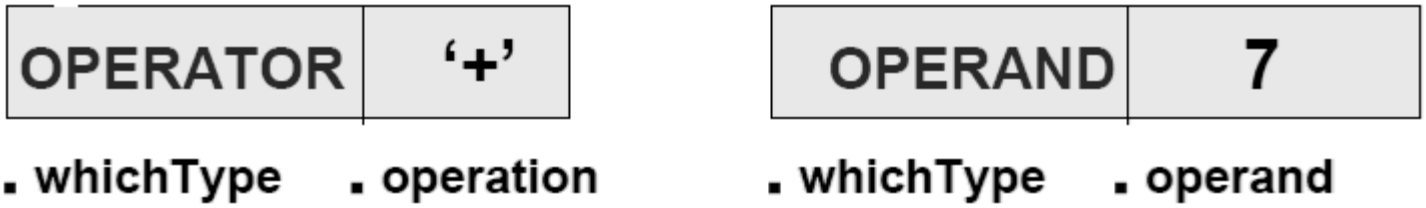
- Mỗi nút có 2 con trỏ

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

Cài đặt
cây biểu thức

- InfoNode có 2 dạng

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



Cài đặt
cây biểu thức

```
int Eval(TreeNode* ptr){
   switch(ptr->info.whichType) {
       case OPERAND :
          return ptr->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

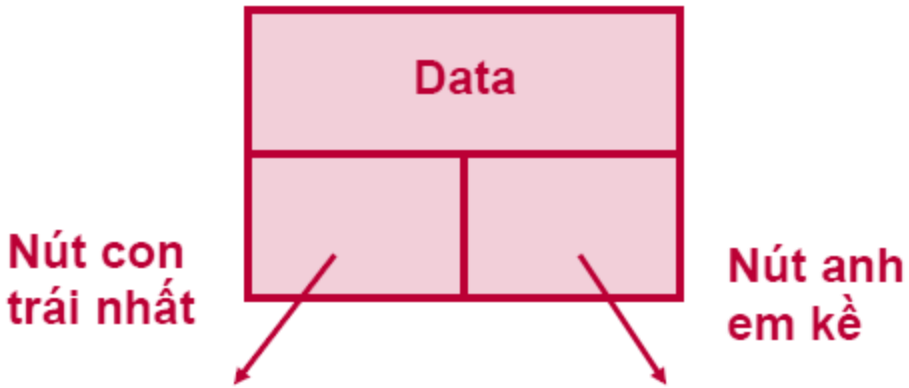
- Biểu diễn giống như cây nhị phân?
  - *Mỗi nút sẽ chứa giá trị và các con trỏ trỏ đến các nút con của nó?*
  - *Bao nhiêu con trỏ cho một nút?*

Không hợp lý

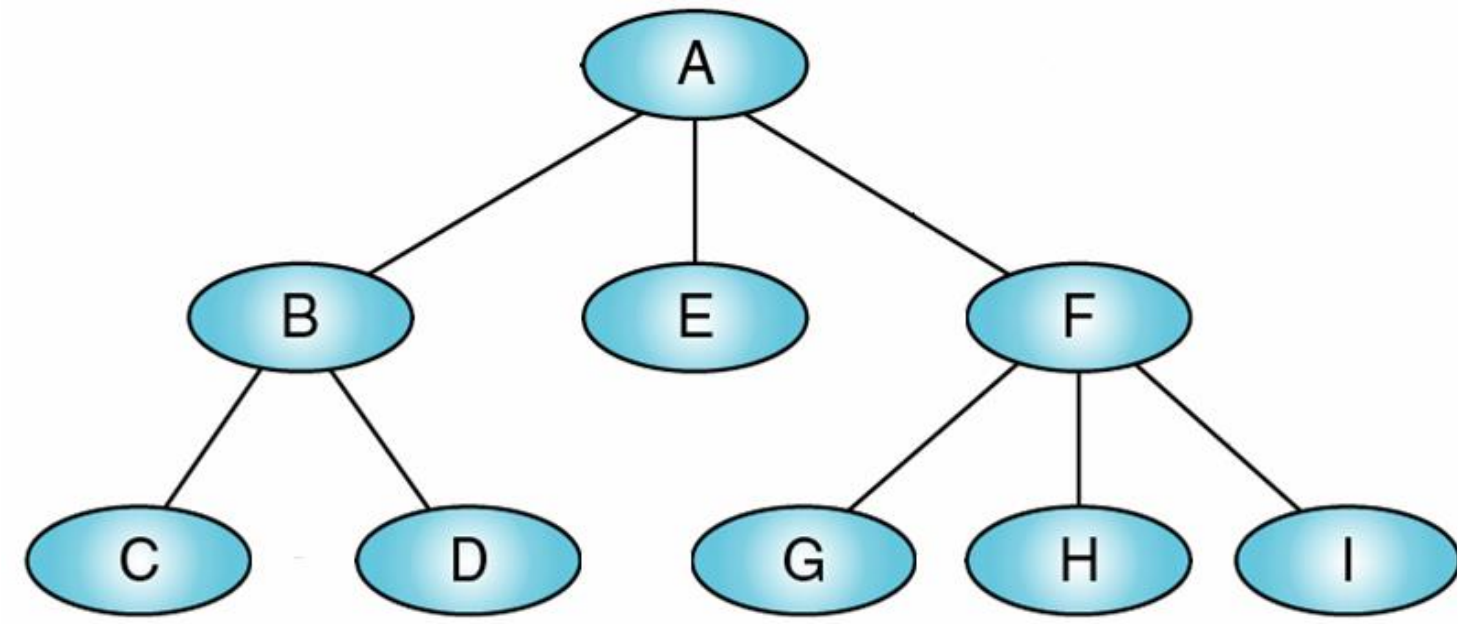
- Mỗi nút sẽ chứa giá trị và một con trỏ trỏ đến một "tập" các nút con
  - *Xây dựng "tập"?*

Cây
tổng quát

- Mỗi nút sẽ có 2 con trỏ:
  - *một con trỏ trỏ đến nút con đầu tiên của nó,*
  - *con trỏ trỏ đến nút anh em kề với nó*



Ví dụ
Cây tổng quát



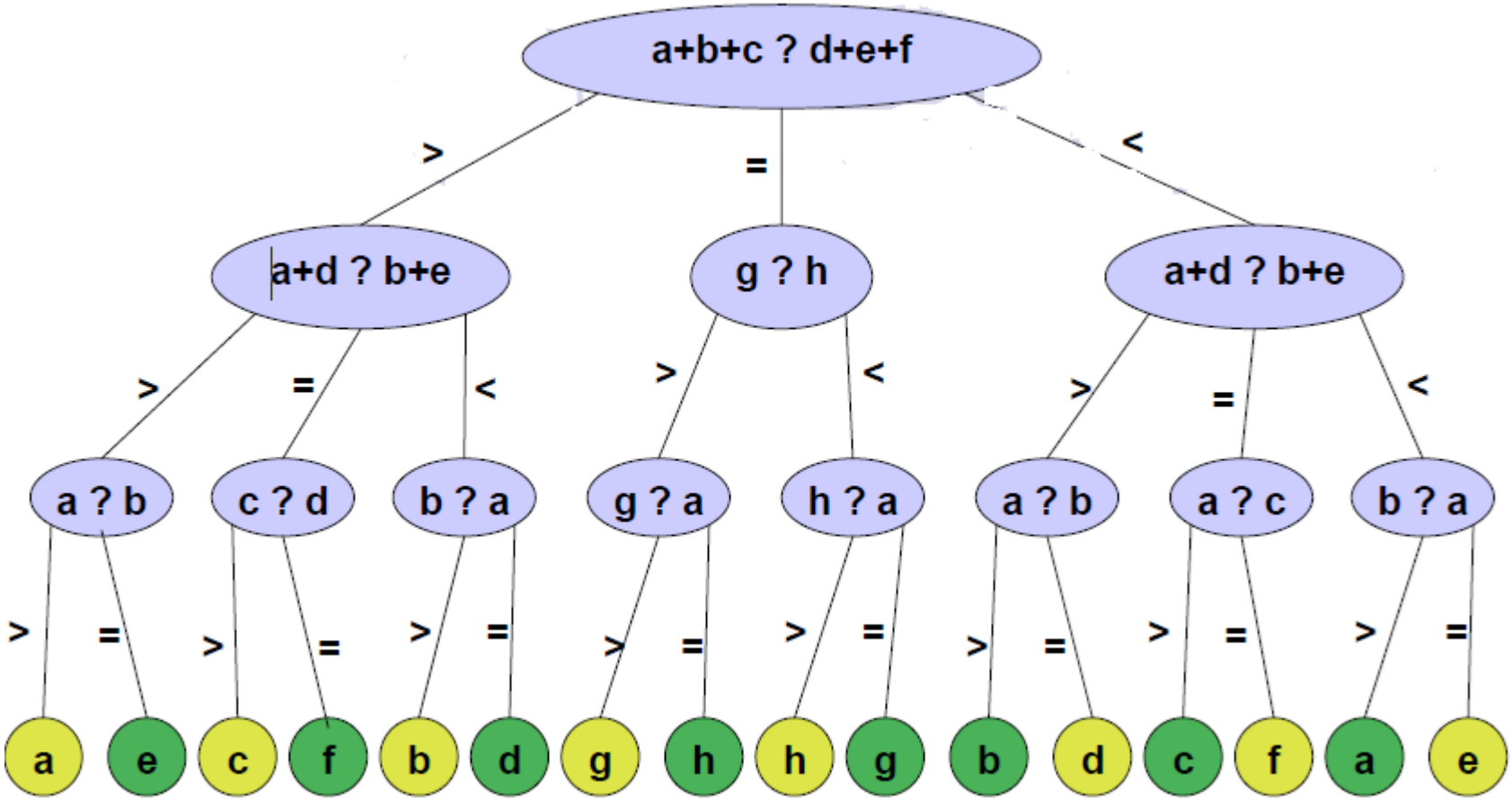
Duyệt
cây tổng quát

- 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*

Ứng dụng
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");
}
```