

# Cấu trúc dữ liệu và giải thuật

**TS. Phạm Tuấn Minh**

Khoa Công nghệ Thông tin, Đại học Phenikaa

[minh.phamtuan@phenikaa-uni.edu.vn](mailto:minh.phamtuan@phenikaa-uni.edu.vn)

<https://sites.google.com/site/phamtuanminh/>

## Bài học này

- ❑ Mảng là khái niệm cơ bản trong hầu hết các ngôn ngữ lập trình (ví dụ C, C++, Java, ...)
- ❑ Ý tưởng: Có biến dùng để tập hợp một nhóm các phần tử có cùng kiểu dữ liệu
- ❑ Ví dụ: Để mô tả sinh viên 50 sinh viên, nếu không sử dụng mảng, có thể cần định nghĩa 50 biến

## Chương 2: Các cấu trúc dữ liệu cơ bản

- ❑ Cấu trúc lưu trữ mảng
  - Mảng một chiều
  - Mảng nhiều chiều
- ❑ Danh sách liên kết
- ❑ Ngăn xếp
- ❑ Hàng đợi

1-3

### Mảng một chiều

- ❑ **Khai báo mảng, khởi tạo giá trị, thao tác trên mảng**
- ❑ Con trỏ và mảng
- ❑ Mảng là tham số của hàm

1-4

## ❑ Tại sao học và sử dụng mảng?

1-5

## Tại sao học và sử dụng mảng

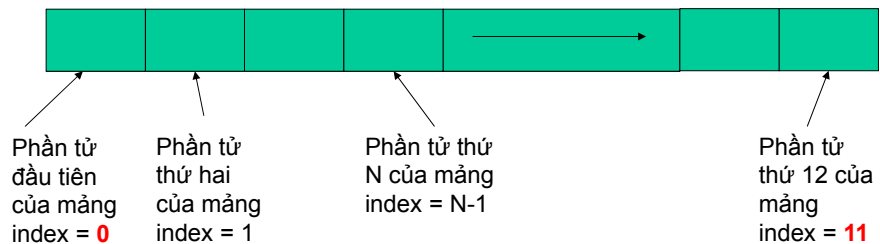
- ❑ Mảng là khái niệm cơ bản trong hầu hết các ngôn ngữ lập trình (ví dụ C, C++, Java, ...)
- ❑ Ý tưởng: Có biến dùng để tập hợp một nhóm các phần tử có cùng kiểu dữ liệu
- ❑ Ví dụ: Để mô tả sinh viên 50 sinh viên, nếu không sử dụng mảng, có thể cần định nghĩa 50 biến

1-6

## Khái niệm mảng

- ❑ **Mảng** là một **danh sách** các phần tử có **cùng kiểu dữ liệu**. Giá trị của mỗi phần tử được lưu trữ tại vị trí được đánh số cụ thể trong mảng
- ❑ Mảng dùng một số nguyên làm chỉ số (**index**) để tham chiếu tới một phần tử trong mảng
- ❑ **Kích thước** của mảng là cố định mỗi khi tạo mảng
- ❑ Index bắt đầu với giá trị **0**

Mảng có kích thước bằng **12**

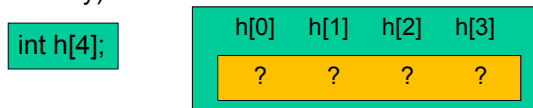


1-7

## Khai báo mảng

- ❑ Khai báo mảng **không khởi tạo** giá trị của các phần tử  

```
float sales[365]; /*mảng 365 phần tử float */  
char name[12]; /*mảng 12 phần tử character*/  
int states[50]; /*mảng 50 phần tử integer*/  
int *pointers[5]; /* mảng 5 con trỏ tới phần tử integer */
```
- ❑ Khi khai báo mảng, chương trình dịch sẽ cấp phát một số vị trí **vùng nhớ liên tục** cho toàn bộ mảng (2 hoặc 4 byte cho kiểu integer tùy thuộc vào máy)



Phần tử:	1	2	3	4
Địa chỉ vùng nhớ:	2021	2023	2025	2027

- ❑ Kích thước mảng phải là hằng số nguyên hoặc biểu thức hằng số nguyên (ANSI/C99 cho phép)  

```
char name[i]; // biến i ==> không hợp lệ  
int states[i*6]; // biến i ==> không hợp lệ
```

1-8

## Khởi tạo mảng

- Khởi tạo mảng khi khai báo mảng

```
#define MTHS 12    /* khai báo hằng số */
```

```
int days[MTHS]={31,28,31,30,31,30,31,31,30,31,30,31};
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
days	31	28	31	30	31	30	31	31	30	31	30	31

- Khởi tạo mảng một phần: Ví dụ khởi tạo 7 phần tử đầu tiên

```
#define MTHS 12    /* khai báo hằng số */
```

```
int days[MTHS]={31,28,31,30,31,30,31}; /* các phần tử còn lại sẽ được khởi tạo  
bằng 0 */
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
days	31	28	31	30	31	30	31	0	0	0	0	0

1-9

## Khởi tạo mảng

- Bỏ qua kích thước mảng khi Khởi tạo mảng

```
int days[]={31,28,31,30,31,30,31}; /* mảng 7 phần tử */
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
days	31	28	31	30	31	30	31

1-10

## Thao tác trên mảng

- ❑ **Truy cập** phần tử mảng

```
sales[0] = 122.5;
```

```
if (sales[0] == 50.0) ... // sử dụng chỉ số mảng
```

- ❑ Chỉ số trong khoảng từ **0** tới **n-1** trong đó n là kích thước của mảng khi khai báo

```
char name[12];
```

```
name[12] = 'c'; // index out of range
```

- ❑ Làm việc với các giá trị

```
(1) days[1] = 29; OK ?
```

```
(2) days[2] = days[2] + 4; OK ?
```

```
(3) days[3] = days[2] + days[3]; OK ?
```


```
(4) days[MTHS] = {2,3,4,5,6}; OK ?
```

1-11

## Duyệt mảng - Dùng chỉ số mảng

- ❑ Ví dụ: Duyệt mảng days[] để hiện giá trị của mỗi phần tử

days	31	28	31	30	31	30	31	31	30	31	30	31
chỉ số	0	1	2	3	4	5	6	7	8	9	10	11



1-12


## Duyệt mảng: In giá trị các phần tử

```
#include <stdio.h>
#define MTHS 12 /* define a constant */
int main( )
{
    int i;
    int days[MTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
    /* print the number of days in each month */
    for (i = 0 ; i < MTHS ; i++)
        printf("Month %d has %d days\n", i+1, days[i]);
    return 0;
}
```

### Output

Month 1 has 31 days  
Month 2 has 28 days  
...  
Month 12 has 31 days

days	31	28	31	30	31	30	31	31	30	31	30	31
chỉ số	0	1	2	3	4	5	6	7	8	9	10	11



1-13

## Duyệt mảng: Tìm kiếm giá trị

```
#include <stdio.h>
#define SIZE 5 /* define a constant */
int main ( )
{
    char myChar[SIZE] = {'b', 'a', 'c', 'k', 's'};
    int i;
    char searchChar;
    // Reading in user's input to search
    printf("Enter a char to search: ");
    scanf("%c", &searchChar);
    // Traverse myChar array and output character if found
    for (i = 0; i < SIZE; i++) {
        if (myChar[i] == searchChar){
            printf("Found %c at index %d", myChar[i], i);
            break; //break out of the loop
        }
    }
    return 0;
}
```

### Output

Enter a char to  
search: **a**  
Found a at index 1

1-14

## Duyệt mảng: Tìm giá trị lớn nhất

```
#include <stdio.h>
int main( )
{
    int index, max, numArray[10];
    max = -1; printf("Enter 10 numbers: \n");
    for (index = 0; index < 10; index++)
        scanf("%d", &numArray[index]);
    // Find maximum from array data
    for (index = 0; index < 10; index++) {
        if (numArray[index] > max)
            max = numArray[index];
    }
    printf("The max value is %d.\n", max);
    return 0;
}
```

### Output

```
Enter 10 numbers:
4 3 8 9 15 25 3 6 7 9
The max value is 25
```

1-15

## Mảng một chiều

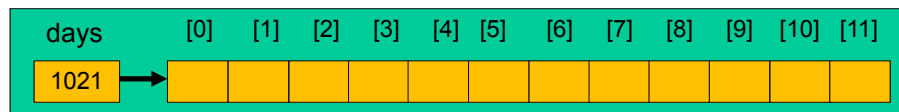
- ☐ Khai báo mảng, khởi tạo giá trị, thao tác trên mảng
- ☐ **Con trỏ và mảng**
- ☐ Mảng là tham số của hàm

1-16



## Hằng số con trỏ

- Tên mảng là **hằng số con trỏ**
- Giả sử một số integer biểu diễn bởi 4 byte (hoặc 2 byte tùy máy tính) và mảng **days** bắt đầu tại vị trí nhớ 1021
- `int days[12];` // days - hằng số con trỏ



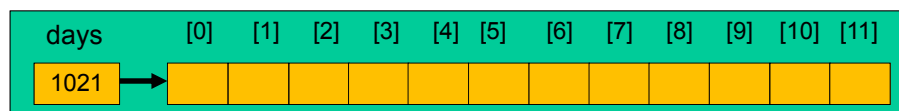
Địa chỉ nhớ: 1021 1023 1025 1027 1029 102B 102D 102F 1031 1033 1035 1037

- Địa chỉ của một phần tử mảng: ví dụ `int h[5];`
- **&h[0]** là **địa chỉ** của phần tử **đầu tiên**
- **&h[i]** là **địa chỉ** của phần tử thứ **(i+1)**

1-17

## Hằng số con trỏ

- Tên mảng, **days**, là **địa chỉ (hoặc con trỏ)** của **phần tử đầu tiên** của mảng



Địa chỉ nhớ: 1021 1023 1025 1027 1029 102B 102D 102F 1031 1033 1035 1037

days

days+1

days+11

`int days[12];`

`days == &days[0]`

`*days == days[0]`

`days + 1 == &days[1]`

`*(days+1) == days[1]`

- Không thể thay đổi con trỏ cơ sở (base pointer) của mảng:

`days += 5;` // i.e., `days = days+5;` **không hợp lệ**

1-18

## Biến con trỏ

- **Biến con trỏ** có thể có **địa chỉ khác nhau**

int **days**[MTHS]; // days - **hằng số con trỏ không thể thay đổi**

```
/* pointer arithmetic */
#define MTHS 12
#include <stdio.h>
int main()
{
    int days[MTHS]= {31,28,31,30,31,30,31,31,30,31,30,31};
    int *day_ptr; // biến con trỏ
    day_ptr = days;
    printf("First element = %d\n", *day_ptr);
    day_ptr = &days[3]; /* points to the fourth element */
    printf("Fourth element = %d\n", *day_ptr);
    day_ptr += 3; /* points to the seventh element */
    printf("Seventh element = %d\n", *day_ptr);
    day_ptr--; /* points to the sixth element */
    printf("Sixth element = %d\n", *day_ptr);
    return 0;
}
```

### Output

First element = 31  
Fourth element = 30  
Seventh element = 31  
Sixth element = 30

1-19

## Thao tác với con trỏ

Thao tác	day_ptr	days	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	...	[11]
			1021	1023	1025	1027	1029	102B	102D	102F		1037
int days[MTH] = {...};	?	1021	31	28	31	30	31	30	31	31	...	31
day_ptr=days;	1021	1021	31	28	31	30	31	30	31	31	...	31
day_ptr=&days[3];	1027	1021	31	28	31	30	31	30	31	31	...	31
day_ptr+=3;	102D	1021	31	28	31	30	31	30	31	31	...	31
day_ptr--;	102B	1021	31	28	31	30	31	30	31	31	...	31

1-20

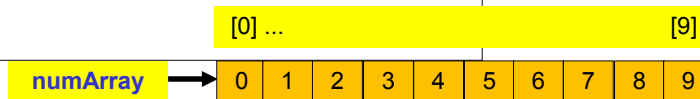
## Dùng hằng số con trỏ - Tìm giá trị lớn nhất

```
#include <stdio.h>
int main( )
{
    int index, max, numArray[10];
    printf("Enter 10 numbers: \n");
    for (index = 0; index < 10; index++)
        scanf("%d", numArray + index);
    // Find maximum from array data
    max = *numArray;
    for (index = 1; index < 10; index++) {
        if (*(numArray + index) > max)
            max = *(numArray + index);
    }
    printf("The max value is %d.\n", max);
    return 0;
}
```

### Output

Enter 10 numbers:  
0 1 2 3 4 5 6 7 8 9

The max value is 9

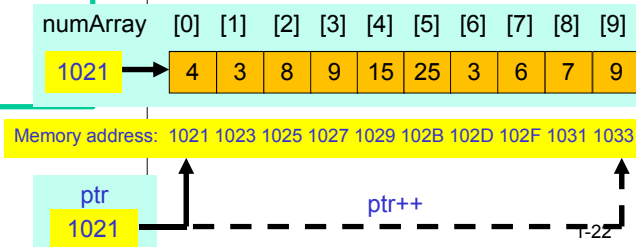


## Dùng biến con trỏ - Tìm giá trị lớn nhất

```
#include <stdio.h>
int main(){
    int index, max, numArray[10];
    int *ptr;
    ptr = numArray;
    printf("Enter 10 numbers: \n");
    for (index = 0; index < 10; index++)
        scanf("%d", ptr++);
    // Find maximum from array data
    ptr = numArray;
    max = *ptr;
    for (index = 0; index < 10; index++) {
        if (*ptr > max)
            max = *ptr;
        ptr++;
    }
    printf("max is %d.\n", max);
    return 0;
}
```

### Output

Enter 10 numbers:  
4 3 8 9 15 25 3 6 7 9  
max is 25.



## Mảng một chiều

- ❑ Khai báo mảng, khởi tạo giá trị, thao tác trên mảng
- ❑ Con trỏ và mảng
- ❑ **Mảng là tham số của hàm**

1-23

## Mảng là tham số của hàm

- ❑ Mảng với số chiều bất kì có thể truyền như tham số của hàm  
**fn(table);** /\* gọi hàm \*/  
**fn** là hàm và **table** là mảng 1 chiều
- ❑ Mảng **table** được truyền vào hàm bằng **tham chiếu** (reference): **Địa chỉ** của **phần tử đầu tiên** của mảng được truyền vào hàm

1-24

## Mảng là tham số của hàm

```
void fn(int table[], int n)
{
    ...
} // n: kích thước của mảng
```

```
void fn(int table[ TABLESIZE])
{
    ...
}
```

```
void fn(int *table, int n)
{
    ...
}
```

- Prototype của hàm:  
void fn(int **table**[], int **n**)  
void fn(int **table**[ TABLESIZE])  
void fn(int **\*table**, int **n**)

1-25

## Truyền mảng như là tham số của hàm

```
#include <stdio.h>
int maximum(int table[], int n);
int main( )
{
    int max, index, n;
    int numArray[10];
    printf("Enter the number of values: ");
    scanf("%d", &n);
    printf("Enter %d values: ", n);
    for (index = 0; index < n; index++)
        scanf("%d", &numArray[index]);

    // find maximum
    max = maximum(numArray, n);
    printf("The maximum value is %d\n", max);

    return 0 ;
}
```

### Output

```
Enter the number of values: 5
Enter 5 values: 1 2 3 4 5
The maximum value is 5
```

1-26

## Truyền mảng như là tham số của hàm

```
int maximum(int table[ ], int n)
{
    int i, max;
    max = table[0];
    for (i = 1; i < n; i++)
        if (table[i] > max)
            max = table[i];
    return max;
}
```

(1) Dùng chỉ số

```
int maximum(int table[ ], int n)
{
    int i, max;
    max = *table;
    for (i = 1; i < n; i++)
        if (*(table+i) > max)
            max = *(table+i);
    return max;
}
```

(2) Dùng con trỏ

1-27

## Chương 2: Các cấu trúc dữ liệu cơ bản

- ❑ Cấu trúc lưu trữ mảng
  - Mảng một chiều
  - Mảng nhiều chiều
- ❑ Danh sách liên kết
- ❑ Ngăn xếp
- ❑ Hàng đợi

1-28

## Cấu trúc dữ liệu và giải thuật

- Nội dung bài giảng được biên soạn bởi TS. Phạm Tuấn Minh.

1-29

# Cấu trúc dữ liệu và giải thuật

**TS. Phạm Tuấn Minh**

Khoa Công nghệ Thông tin, Đại học Phenikaa

[minh.phamtuan@phenikaa-uni.edu.vn](mailto:minh.phamtuan@phenikaa-uni.edu.vn)

<https://sites.google.com/site/phamtuanminh/>

## Chương 2: Mảng và danh sách liên kết

### ❑ Cấu trúc lưu trữ mảng

- Mảng một chiều
- Mảng nhiều chiều
  - Khai báo mảng, khởi tạo thao tác trên mảng nhiều chiều
  - Mảng nhiều chiều là tham số của hàm
  - Sử dụng mảng một chiều trong mảng hai chiều
  - Toán tử sizeof

### ❑ Danh sách liên kết

- ❑ Ngăn xếp
- ❑ Hàng đợi



## Khai báo mảng nhiều chiều

- Ví dụ mảng 2 chiều:

`int x[3][5];` // mảng 3 phần tử của các mảng 5 phần tử

- Ví dụ mảng 3 chiều

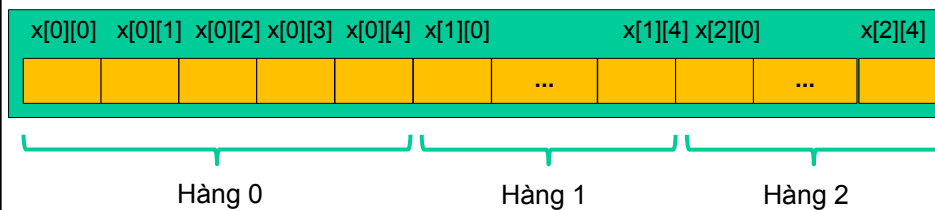
`char x[3][4][5];` // mảng 3 phần tử của các mảng 4 phần tử của các mảng 5 phần tử

1-3

## Mảng nhiều chiều

	Cột 0	Cột 1	Cột 2	Cột 3	Cột 4
Hàng 0	<code>x[0][0]</code>	<code>x[0][1]</code>	<code>x[0][2]</code>	<code>x[0][3]</code>	<code>x[0][4]</code>
Hàng 1	<code>x[1][0]</code>	<code>x[1][1]</code>	<code>x[1][2]</code>	<code>x[1][3]</code>	<code>x[1][4]</code>
Hàng 2	<code>x[2][0]</code>	<code>x[2][1]</code>	<code>x[2][2]</code>	<code>x[2][3]</code>	<code>x[2][4]</code>

Tổ chức lưu trữ trong Bộ nhớ



1-4

## Khởi tạo mảng nhiều chiều

- Khởi tạo mảng nhiều chiều:

```
int x[2][2] = { {1, 2}, // hàng thứ nhất  
               {6, 7} }; // hàng thứ hai
```

hoặc

```
int x[2][2] = {1, 2, 6, 7};
```

- Khởi tạo một phần

```
int exam[3][3] = {{1, 2, 4}, {5,7}};
```

```
int exam[3][3] = {1, 2, 4, 5,7};
```

```
//tương đương
```

```
int exam[3][3] = {{1, 2, 4}, {5,7}};
```

1-5

## Khởi tạo mảng nhiều chiều

- Có thể bỏ qua **chiều ngoài cùng** vì trình biên dịch có thể nhận biết, ví dụ

```
int arr[ ][3][2] = {  
    { {1,1}, {0,0}, {1,1} },  
    { {0,0}, {1,2}, {0,1} }  
};
```

tạo ra mảng có chiều là [2][3][2]

- Khai báo sau là không hợp lệ

```
int wrong_arr[ ][ ] = {1,2,3,4};
```

1-6

## Thao tác trên mảng nhiều chiều - Dùng chỉ số mảng

```
#include <stdio.h>
int main()
{ // declare an array with initialization
    int array[3][3]={
        {5, 10, 15},
        {10, 20, 30},
        {20, 40, 60}
    };
    int row, column, sum;
    /* compute sum of row - traverse each row first */
    for (row = 0; row < 3; row++) { // nested loop
        /* for each row - compute the sum */
        sum = 0;
        for (column = 0; column < 3; column++)
            sum += array[row][column]; // using array indexes
        printf("The sum of elements in row %d is %d\n", row+1, sum);
    }
}
```

## Thao tác trên mảng nhiều chiều - Dùng chỉ số mảng

```
/* compute sum of each column */
for (column = 0; column < 3; column++) {
    sum = 0;
    for (row = 0; row < 3; row++)
        sum += array[row][column];
    printf("The sum of elements in column %d is %d\n", column+1, sum);
}
return 0;
}
```

### Output

The sum of elements in row 1 is 30  
The sum of elements in row 2 is 60  
The sum of elements in row 3 is 120  
The sum of elements in column 1 is 35  
The sum of elements in column 2 is 70  
The sum of elements in column 3 is 105

## Chương 2: Mảng và danh sách liên kết

- ❑ Cấu trúc lưu trữ mảng
  - Mảng một chiều
  - Mảng nhiều chiều
    - Khai báo mảng, khởi tạo thao tác trên mảng nhiều chiều
    - **Mảng nhiều chiều là tham số của hàm**
    - Sử dụng mảng một chiều trong mảng hai chiều
    - Toán tử sizeof
- ❑ Danh sách liên kết
- ❑ Ngăn xếp
- ❑ Hàng đợi

1-9

## Mảng nhiều chiều là tham số của hàm

- ❑ Định nghĩa của hàm có mảng 2 chiều là tham số như sau:

```
void fn(int ar2[2][4])  
{  
    ...  
}
```

```
void fn(int ar2[][4])  
{  
    ...  
}
```

- ❑ Trong định nghĩa trên, chiều thứ nhất có thể bỏ qua vì trình biên dịch cần thông tin của mọi chiều trừ chiều thứ nhất của mảng

1-10

## Tại sao chiều thứ nhất có thể bỏ qua

- Ví dụ, lệnh gán

`ar2[1][3] = 100;`

yêu cầu trình biên dịch địa chỉ của `ar2[1][3]` và ghi giá trị 100 vào địa chỉ này.

Để tính địa chỉ, thông tin về chiều phải chuyển cho trình biên dịch.

- Giả sử định nghĩa ar2 như sau

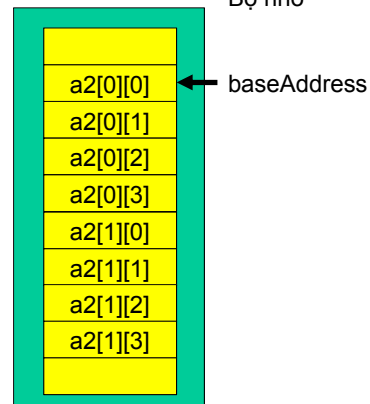
`int ar2[D1][D2];`

Địa chỉ của `ar2[1][3]` được tính như sau

$\text{baseAddress} + \text{row} * D2 + \text{column}$

$\Rightarrow \text{baseAddress} + 1 * 4 + 3$

$\Rightarrow \text{baseAddress} + 7$



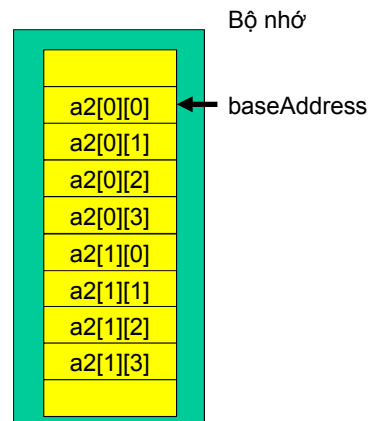
## Tại sao chiều thứ nhất có thể bỏ qua

- `baseAddress` là địa chỉ trỏ tới phần tử đầu tiên của `ar2`
- Vì không cần `D1` khi tính địa chỉ, nên có thể bỏ qua giá trị của chiều đầu tiên khi định nghĩa hàm có mảng là tham số
- Prototype của hàm như sau

`void fn(int ar2[2][4]);`

hoặc

`void fn(int ar2[ ][4]);`



## Truyền mảng 2 chiều là tham số hàm

```
#include <stdio.h>
int sum_rows(int ar[ ][3]);
int sum_columns(int ar[ ][3]);
int main()
{
    int array[3][3]= {
        {5, 10, 15},
        {10, 20, 30},
        {20, 40, 60}
    };
    int total_row, total_column;
    total_row = sum_rows(array); // sum of all rows
    total_column = sum_columns(array); // all columns
    printf("The sum of all elements in rows is %d\n", total_row);
    printf("The sum of all elements in columns is %d\n", total_column);
    return 0;
}
```

### Output

The sum of all elements in rows is 210  
The sum of all elements in columns is 210

## Truyền mảng 2 chiều là tham số hàm

```
int sum_rows(int ar[ ][3])
{
    int row, column;
    int sum=0;
    for (row = 0; row < 3; row++){
        for (column = 0; column < 3; column++){
            sum += ar[row][column];
        }
    }
    return sum;
}
int sum_columns(int ar[ ][3])
{
    int row, column;
    int sum=0;
    for (column = 0; column < 3; column++){
        for (row = 0; row < 3; row++){
            sum += ar[row][column];
        }
    }
    return sum;
}
```

Bỏ qua khai báo  
chiều thứ nhất

## Chương 2: Mảng và danh sách liên kết

### ❑ Cấu trúc lưu trữ mảng

- Mảng một chiều
- Mảng nhiều chiều
  - Khai báo mảng, khởi tạo thao tác trên mảng nhiều chiều
  - Mảng nhiều chiều là tham số của hàm
  - **Sử dụng mảng một chiều trong mảng hai chiều**
  - Toán tử Sizeof

### ❑ Danh sách liên kết

### ❑ Ngăn xếp

### ❑ Hàng đợi

1-15

## Sử dụng mảng 1 chiều trong mảng 2 chiều

```
#include <stdio.h>
void display1(int *ptr, int size);
void display2(int ar[ ], int size);

int main()
{
    int array[2][4] = { 0, 1, 2, 3, 4, 5, 6, 7 };
    int i;

    for (i=0; i<2; i++) { /* as 2-D Array */
        display1(array[i], 4);
        display2(array[i], 4);
    }

    display1(array, 8); /* as 1-D array */
    display2(array, 8); /* as 1-D array */
    return 0;
}
```

#### Output:

Display1 result: 0 1 2 3

**Display2 result: 0 5 10 15**

Display1 result: 4 5 6 7

**Display2 result: 20 25 30 35**

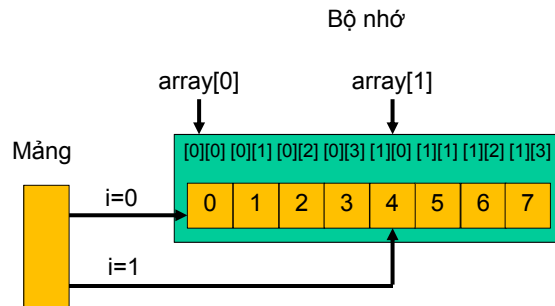
Display1 result: 0 1 2 3 4 5 6 7

Display2 result: 0 5 10 15 20 25 30 35

## Sử dụng mảng 1 chiều trong mảng 2 chiều

```
void display1(int *ptr, int size)
{
    int j;
    printf("Display1 result: ");
    for (j=0; j<size; j++)
        printf("%d ", *ptr++);
    putchar('\n');
}
```

```
void display2(int ar[ ], int size)
{
    int k;
    printf("Display2 result: ");
    for (k=0; k<size; k++)
        printf("%d ", 5*ar[k]);
    putchar('\n');
}
```



## Chương 2: Mảng và danh sách liên kết

### □ Cấu trúc lưu trữ mảng

- Mảng một chiều
- Mảng nhiều chiều
  - Khai báo mảng, khởi tạo thao tác trên mảng nhiều chiều
  - Mảng nhiều chiều là tham số của hàm
  - Sử dụng mảng một chiều trong mảng hai chiều
  - **Toán tử Sizeof**

### □ Danh sách liên kết

- Ngăn xếp
- Hàng đợi



## Toán tử Sizeof

- ❑ sizeof() là toán tử trả về **kích thước** (theo byte) của toán hạng. Cú pháp  
**sizeof(operand)**  
hoặc  
**sizeof operand**
- ❑ **operand** có thể là
  - int, float,... tên kiểu dữ liệu phức tạp, tên biến, tên mảng

1-19

## Toán tử sizeof

```
#include <stdio.h>
int sum(int a[], int);
int main(){
    int ar[6] = {1,2,3,4,5,6};
    int total;
    printf("Array size is %d\n", sizeof(ar)/sizeof(ar[0]));
    total = sum(ar, 6);
    return 0;
}
int sum ( int a[], int n ) {
    int i, total=0;
    printf("Size of a = %d\n", sizeof(a));
    for ( i=0; i<n ; i++)
        total += a[i];
    return total;
}
```

### Output:

Array size is 6 (i.e. 24/4=6)  
Size of a = 4

**sizeof** cho **biến con trỏ** (i.e., a) cho kết quả là kích thước của con trỏ

## Tóm tắt: Mảng 2 chiều

	Cột 0	Cột 1	Cột 2	Cột 3	Cột 4	
Hàng 0	x[0][0]	x[0][1]	x[0][2]	x[0][3]	x[0][4]	Vị trí bộ nhớ liên tục
Hàng 1	x[1][0]	x[1][1]	x[1][2]	x[1][3]	x[1][4]	
Hàng 2	x[2][0]	x[2][1]	x[2][2]	x[2][3]	x[2][4]	

```
// Printing array elements
#include <stdio.h>
int main ( ) {
    int ar[3][3]= {
        {5, 10, 15},
        {10, 20, 30},
        {20, 40, 60}
    };
    int i, j;
    /* using index – nested loop*/
    printf("\n");
    for (i=0; i<3; i++)
        for (j=0; j<3; j++)
            printf("%d ", ar[i][j]);
    printf("\n");
    return 0;
}
```

[Dùng chỉ số](#)

```
// Print array elements
#include <stdio.h>
#define SIZE 9
int main ( ) {
    int ar[3][3]= {
        {5, 10, 15},
        {10, 20, 30},
        {20, 40, 60}
    };
    int i, *ptr;
    ptr = ar;
    /* using pointer – looping */
    for (i=0; i<SIZE; i++)
        printf("%d ", *ptr++);
    printf("\n");
    return 0;
}
```

[Dùng con trỏ như mảng 1 chiều](#)

## Chương 2: Mảng và danh sách liên kết

- ❑ Cấu trúc lưu trữ mảng
  - Mảng một chiều
  - Mảng nhiều chiều
    - Khai báo mảng, khởi tạo thao tác trên mảng nhiều chiều
    - Mảng nhiều chiều là tham số của hàm
    - Sử dụng mảng một chiều trong mảng hai chiều
    - Toán tử sizeof
- ❑ Danh sách liên kết
- ❑ Ngăn xếp
- ❑ Hàng đợi

## Cấu trúc dữ liệu và giải thuật

- Nội dung bài giảng được biên soạn bởi TS. Phạm Tuấn Minh.

1-23

# Cấu trúc dữ liệu và giải thuật

**TS. Phạm Tuấn Minh**

Khoa Công nghệ Thông tin, Đại học Phenikaa

[minh.phamtuan@phenikaa-uni.edu.vn](mailto:minh.phamtuan@phenikaa-uni.edu.vn)

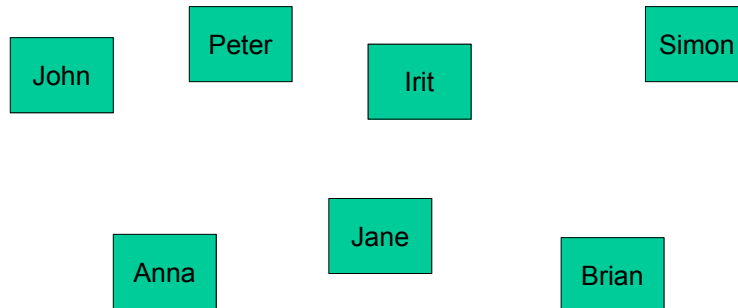
<https://sites.google.com/site/phamtuanminh/>

## Chương 2: Mảng và danh sách liên kết

- ❑ Cấu trúc lưu trữ mảng
- ❑ **Danh sách liên kết**
  - **Giới thiệu danh sách liên kết**
  - Cài đặt danh sách liên kết
  - Các thao tác trên danh sách liên kết
- ❑ Ngăn xếp
- ❑ Hàng đợi

## Cấu trúc dữ liệu tuyến tính

- Giả sử có một tập các tên



1-3

## Cấu trúc dữ liệu tuyến tính

- Nếu các tên sắp thành một danh sách xếp hàng



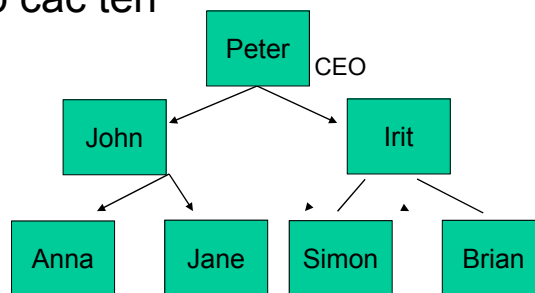
Danh sách

- Quản lý danh sách dữ liệu trên như thế nào ?

1-4

## Cấu trúc dữ liệu không tuyến tính

- Tập các tên



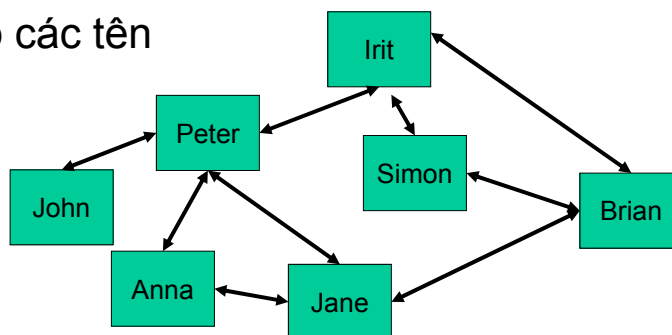
Cây

- Tổ chức công ty

1-5

## Cấu trúc dữ liệu không tuyến tính

- Tập các tên



Đồ thị

- Mạng bạn bè

1-6

## Cấu trúc dữ liệu đơn giản nhất

- Danh sách



- Dữ liệu tuần tự

- Thứ tự giữa các phần tử (No.1, No.2., No.3, ...)
- Mỗi phần tử có một vị trí trong chuỗi
- Mỗi phần tử đến sau phần tử khác

- Lưu trữ danh sách các phần tử

- Danh sách tên, danh sách số, ...
- Dùng mảng để lưu trữ danh sách: Hạn chế?
- Dùng danh sách liên kết để lưu trữ danh sách

1-7

## Danh sách liên kết

- Chúng ta muốn

- Dễ dàng thêm một phần tử mới vào bất kì vị trí nào trong danh sách
- Dễ dàng xóa một phần tử trong danh sách
- Dễ dàng di chuyển vị trí của một phần tử trong danh sách

- Mảng không hỗ trợ các yêu cầu này

- Hỗ trợ truy cập ngẫu nhiên nhanh: Vị trí trong danh sách = vị trí trong mảng

arr[0] arr[1] arr[2] arr[3] arr[4]

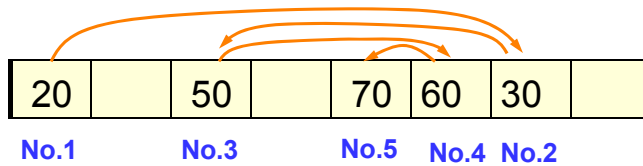
20	30	50	60	70			
----	----	----	----	----	--	--	--

No.1 No.2 No.3 No.4 No.5

1-8

## Danh sách liên kết

- Danh sách liên kết
  - Vị trí trong danh sách khác vị trí trong bộ nhớ
  - Phần tử có thể lưu trữ ở bất kì vị trí nào
  - Cần thêm dữ liệu để chỉ ra vị trí trong danh sách
  - Liên kết (Link): Con trỏ tới phần tử tiếp theo
  - Danh sách liên kết (Linked list): các nút với các liên kết



1-9

## Nút trong danh sách liên kết

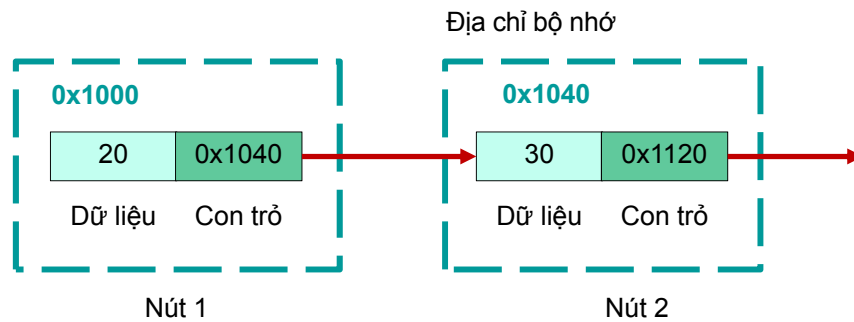
- Mỗi nút có cấu trúc ListNode
- Một nút cơ bản có hai thành phần
  - Dữ liệu lưu trữ bởi nút: integer, char, ...
  - Liên kết: con trỏ tham chiếu tới nút tiếp theo trong danh sách

```
typedef struct _listnode{  
    int num;  
    struct _listnode *next;  
}ListNode;
```

1-10



## Nút trong danh sách liên kết



```
typedef struct _listnode{  
    int num;  
    struct _listnode *next;  
} ListNode;
```

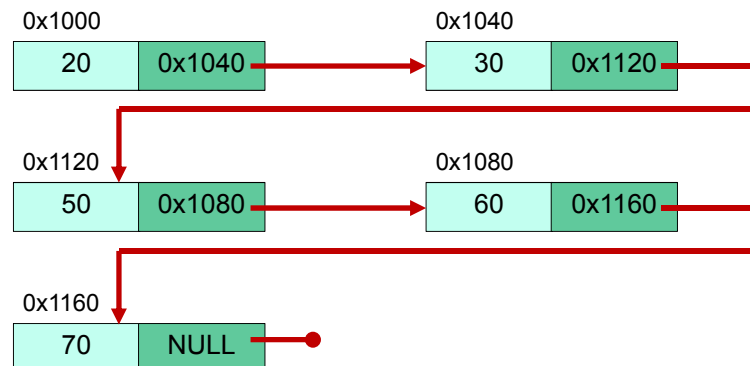
Mỗi structure có hai phần

- Dữ liệu
- Liên kết

1-11

## Nút trong danh sách liên kết

- Mỗi nút chứa một phần tử
- Con trỏ của mỗi nút chỉ tới nút tiếp theo
- Mỗi nút có thể được lưu trữ ở bất kì vị trí nào (không cần liên tục) trong bộ nhớ
- Các nút có thể tạo động bằng malloc()



1-12

## Truy cập tới phần tử trong danh sách

- Đối với danh sách lưu trữ trong mảng:
  - Dễ dàng truy cập phần tử thứ  $i$  trong danh sách:  $arr[i-1]$
  - Phần tử tiếp theo của  $arr[i]$  trong danh sách được lưu trữ trong  $arr[i+1]$
  - Phần tử trước phần tử chứa trong  $arr[i]$  trong danh sách được lưu trữ trong  $arr[i-1]$

$arr[0]$   $arr[1]$   $arr[2]$   $arr[3]$   $arr[4]$

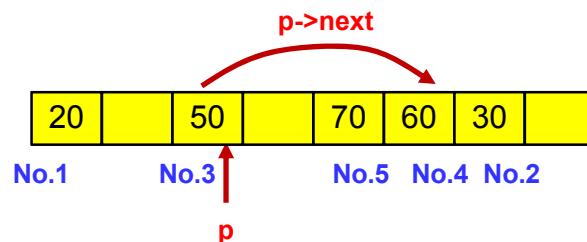
20	30	50	60	70			
----	----	----	----	----	--	--	--

No.1 No.2 No.3 No.4 No.5

1-13

## Truy cập tới phần tử trong danh sách

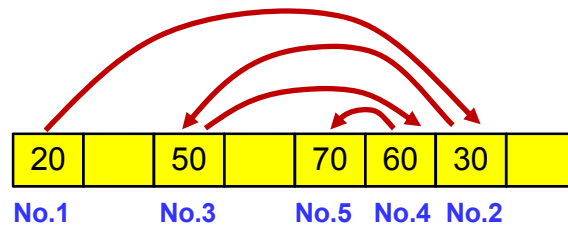
- Đối với danh sách lưu trữ trong mảng: dễ thực hiện
- Đối với danh sách lưu trữ trong danh sách liên kết:
  - Mỗi nút theo dấu của nút tiếp theo sau nó
  - Nếu  $p$  chỉ tới phần tử thứ  $i$  trong danh sách,  $p \rightarrow next$  chỉ tới phần tử thứ  $(i+1)$



1-14

## Theo vết các phần tử trong danh sách

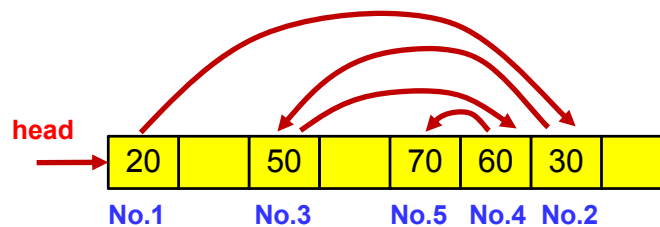
- Đối với danh sách lưu trữ trong mảng: dễ thực hiện
- Đối với danh sách lưu trữ trong danh sách liên kết:
  - Mỗi nút theo dấu của nút tiếp theo sau nó
  - Nếu **p** chỉ tới phần tử thứ *i* trong danh sách, **p->next** chỉ tới phần tử thứ (*i*+1)
  - Mọi nút trong danh sách đều truy cập bắt đầu từ nút đầu tiên trong danh sách



1-15

## Theo vết các phần tử trong danh sách

- Không thể truy cập tới các nút trong danh sách liên kết nếu không có địa chỉ của nút đầu tiên
- Cần một biến con trỏ chỉ tới nút đầu tiên: **head**



1-16

## So sánh lưu trữ danh sách bằng mảng và danh sách liên kết

- Truy cập ngẫu nhiên tới các phần tử trong danh sách
  - **Mảng: dễ**
  - **Danh sách liên kết: không dễ**
- Thay đổi danh sách: thêm, xóa các phần tử
  - **Mảng: không dễ**
  - **Linked List: dễ**

1-17

## Chương 2: Mảng và danh sách liên kết

- Cấu trúc lưu trữ mảng
- Danh sách liên kết
  - Giới thiệu danh sách liên kết
  - **Cài đặt danh sách liên kết**
  - Các thao tác trên danh sách liên kết
- Ngăn xếp
- Hàng đợi

1-18

## Nút trong danh sách liên kết

- Mỗi nút có cấu trúc ListNode
- Một nút cơ bản có hai thành phần
  - Dữ liệu lưu trữ bởi nút: integer, char, ...
  - Liên kết: con trỏ tham chiếu tới nút tiếp theo trong danh sách

```
typedef struct _listnode{  
    int num;  
    struct _listnode *next;  
} ListNode;
```

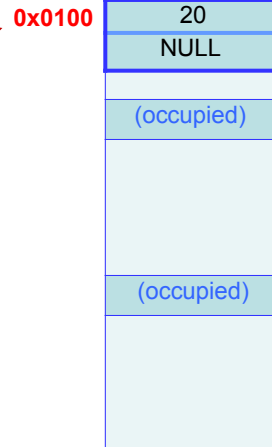
```
typedef struct _listnode{  
    int num;  
    char name[20];  
    ...  
    struct _listnode *next;  
} ListNode;
```

1-19

## Tạo nút

```
ListNode *newNode;  
newNode=malloc(sizeof(ListNode));  
newNode->num=20;  
newNode->next=NULL;
```

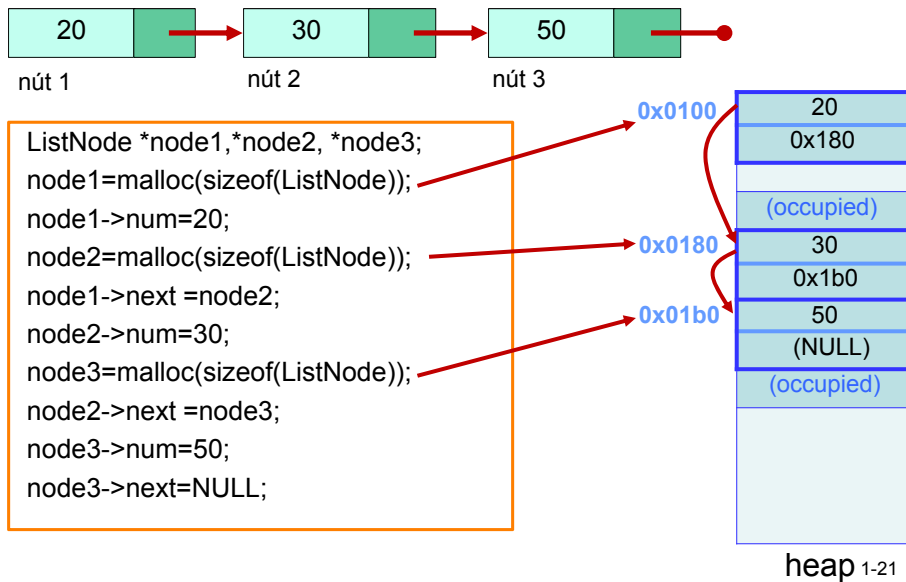
newNode = 0x0100



- Tạo một nút động
  - Trong thời gian chạy
  - Sử dụng malloc()

heap 1-20

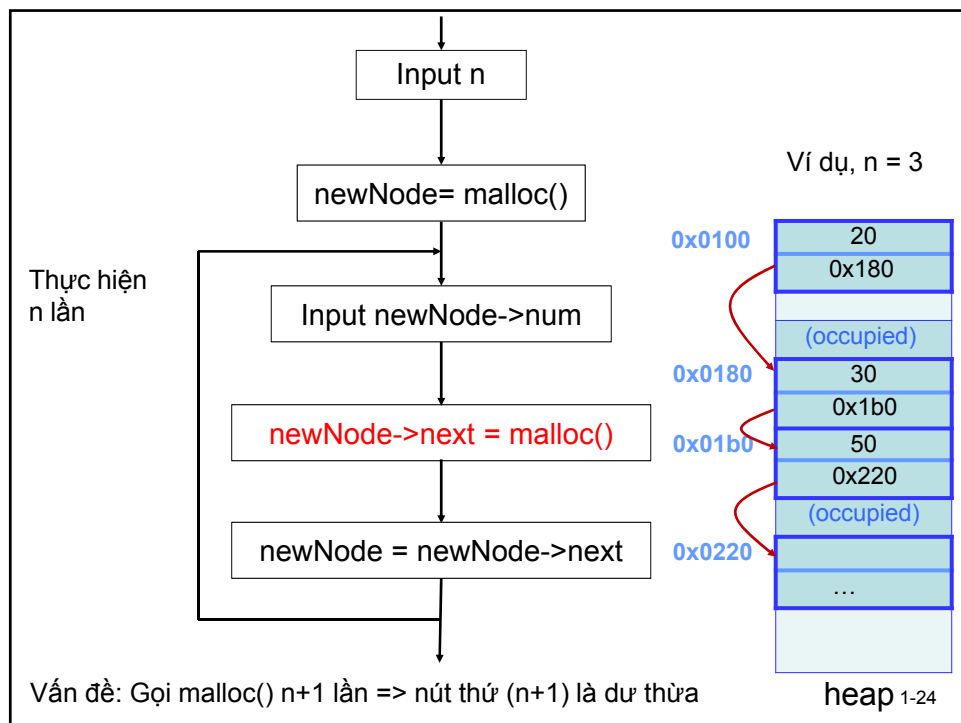
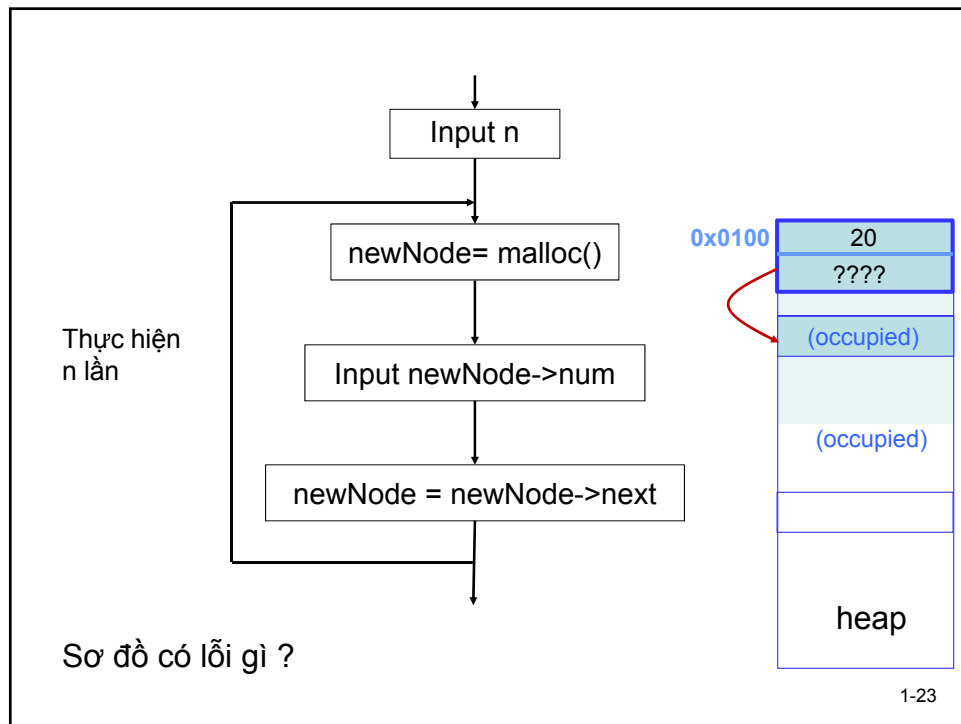
## Tạo một danh sách liên kết có 3 nút

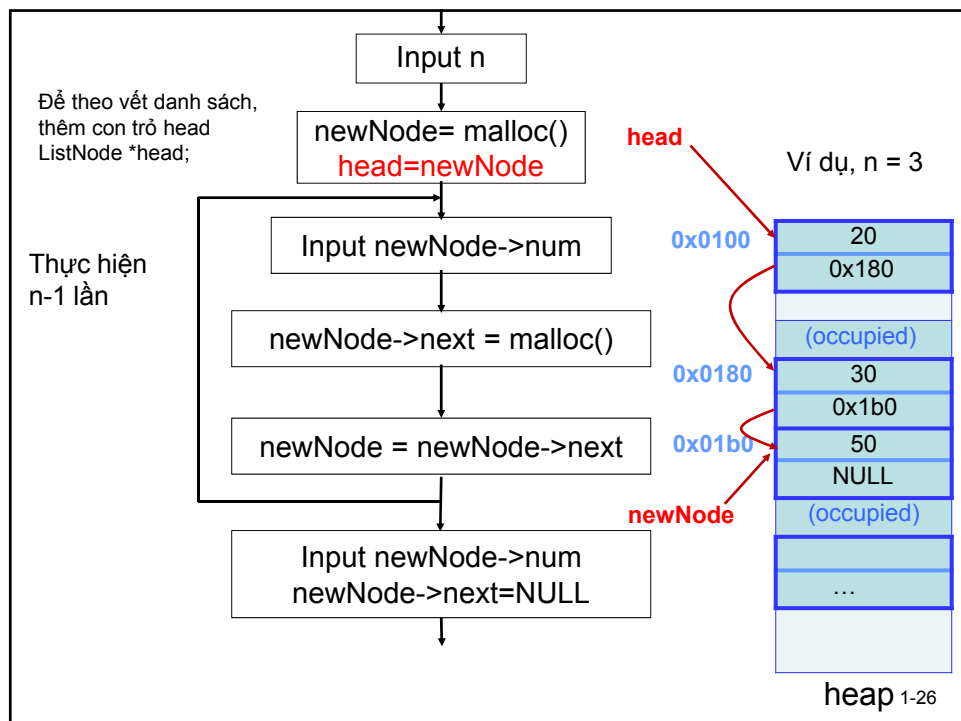
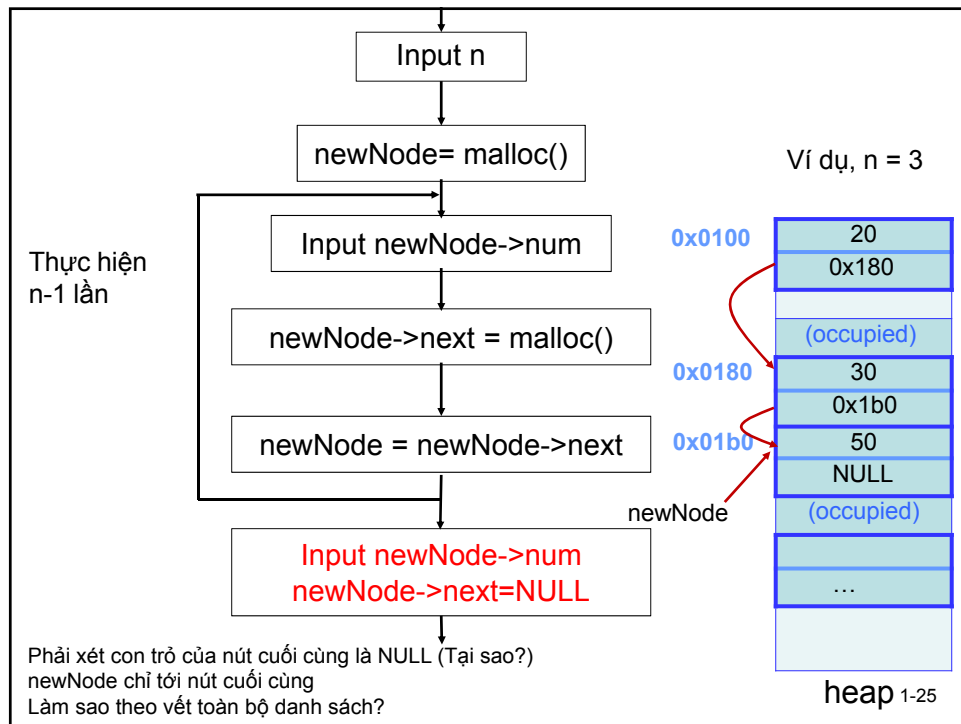


## Tạo danh sách liên kết có n nút

Viết một chương trình yêu cầu người dùng nhập vào số lượng số nguyên sẽ nhập n (giả sử  $n > 0$ ) và sau đó hỏi giá trị từng số nguyên.

- Lặp n lần: dùng malloc() để tạo nút mới, sau đó thêm nút này vào danh sách liên kết







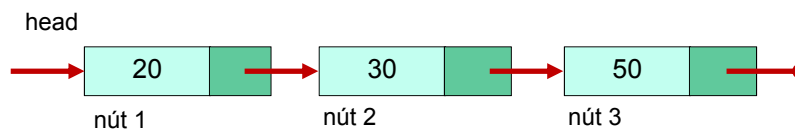
## Tạo danh sách liên kết có $n > 0$ nút

```
int n;
ListNode *newNode, *head;
scanf("%d", &n);
newNode=malloc(sizeof(ListNode));
head=newNode;
for (int i=1; i<n; i++){
    scanf("%d", &newNode->num);
    newNode->next=malloc(sizeof(ListNode));
    newNode=newNode->next;
}
scanf("%d", &newNode->num); //the last num
newNode->next=NULL;
```

1-27

## Ví dụ

- $n=3$ , dữ liệu 20, 30, 50, kết quả chạy chương trình cho danh sách liên kết như sau



- Theo vết các phần tử trong danh sách:
  - head chỉ tới node1, head->num là 20
  - head->next chỉ tới node2, head->next->num là 30
  - head->next->next chỉ tới node3, head->next->next->num là 50
  - head->next->next->next là NULL

1-28

## Tạo danh sách liên kết có $n \geq 0$ nút

```
int n;
ListNode *newNode, *head=NULL;
scanf("%d", &n);
if (n>0) {
    newNode=malloc(sizeof(ListNode));
    head=newNode;
    for (int i=1; i<n; i++){
        scanf("%d", &newNode->num);
        newNode->next=malloc(sizeof(ListNode));
        newNode=newNode->next;
    }
    scanf("%d", &newNode->num); //the last num
    newNode->next=NULL;
}
```

1-29

## Chương 2: Mảng và danh sách liên kết

- ❑ Cấu trúc lưu trữ mảng
- ❑ Danh sách liên kết
  - Giới thiệu danh sách liên kết
  - Cài đặt danh sách liên kết
  - **Các thao tác trên danh sách liên kết**
- ❑ Ngăn xếp
- ❑ Hàng đợi

1-30

## Tạo danh sách liên kết có n nút

Viết một chương trình yêu cầu người dùng nhập vào số lượng số nguyên sẽ nhập n (giả sử  $n > 0$ ) và sau đó hỏi giá trị từng số nguyên.

Sau đó người dùng có thể liên tục tạo các thay đổi: thêm mới, xóa phần tử trong danh sách

- Để tránh lặp lại mã chương trình, cần viết các hàm cho một số thao tác cơ bản

1-31

## Các hàm cơ bản cho danh sách liên kết

### □ Các thao tác cơ bản

#### ○ Chèn nút mới

- Vào đầu
- Vào cuối
- Vào giữa

**InsertNode()**

#### ○ Xóa một nút

- Ở đầu
- Ở cuối
- Ở giữa

**RemoveNode()**

#### ○ Liệt kê toàn bộ các phần tử trong danh sách

**PrintList()**

#### ○ Tìm kiếm nút có chỉ số i trong danh sách

**FindNode()**

1-32

## PrintList()

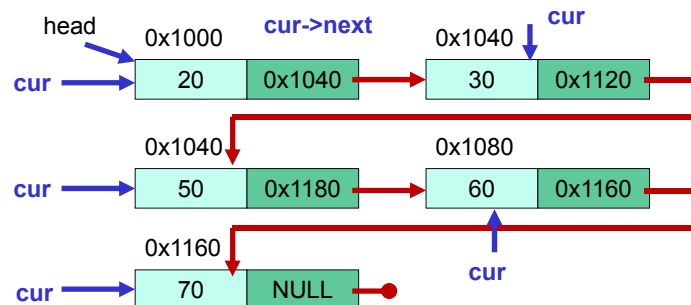
- Liệt kê tất cả các phần tử trong danh sách liên kết bắt đầu từ phần tử đầu tiên và duyệt danh sách tới phần tử cuối cùng
- Chuyển con trỏ head vào hàm  
`void printList(ListNode *head)`
- Tại mỗi nút, sử dụng con trỏ next để di chuyển tới phần tử tiếp theo

1-33

## PrintList()

```
void printList(ListNode *head){  
    ListNode *cur=head;  
    if (cur== NULL) return;  
    while (cur!= NULL){  
        printf("%d\n", cur ->num);  
        cur = cur ->next;  
    }  
}
```

20
30
50
60
70



1-34

## findNode()

- ❑ Tìm con trỏ chỉ tới nút có chỉ số i
  - ❑ Truyền con trỏ head vào hàm
- `ListNode * findNode(ListNode *head, int i)`

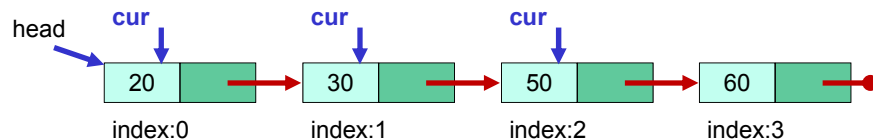
1-35

## findNode()

```
ListNode *findNode(ListNode*head, int i) {  
    ListNode *cur=head;  
    if (head==NULL || i<0) return NULL;  
    while(i>0){  
        cur=cur->next;  
        if (cur==NULL) return NULL;  
        i--;  
    }  
    return cur;  
}
```

Khi danh sách trống hoặc chỉ số không hợp lệ

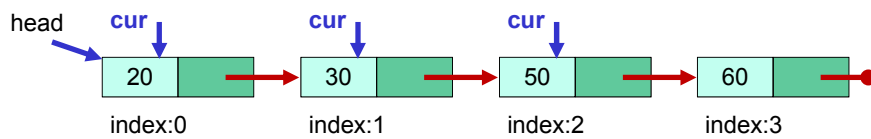
Khi danh sách ngắn hơn chỉ số



1-36

## findNode()

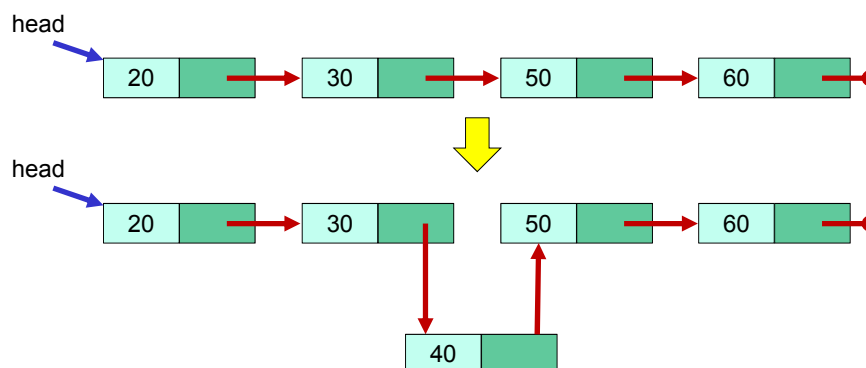
```
ListNode *findNode(ListNode*head, int i) {  
    ListNode *cur=head;  
    if (head==NULL || i<0) return NULL;  
    if (i == 0) return head;  
    while(i>0){  
        cur=cur->next;  
        if (cur==NULL) return NULL;  
        i--;  
    }  
    return cur;  
}
```



1-37

## insertNode()

- Thêm nút (40) vào giữa của danh sách liên kết không rỗng
- Thêm nút này vào vị trí có chỉ số 2, ngay sau nút có chỉ số 1



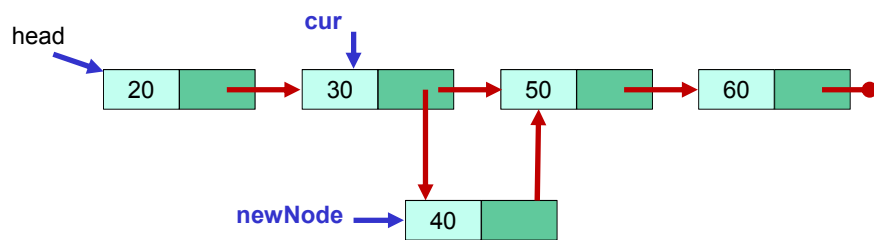
1-38

## insertNode()

- ❑ Thêm nút (40) vào giữa của danh sách liên kết không rỗng
- ❑ Thêm nút này vào vị trí có chỉ số 2, ngay sau nút có chỉ số 1

```
newNode->next = cur->next;  
cur->next = newNode;
```

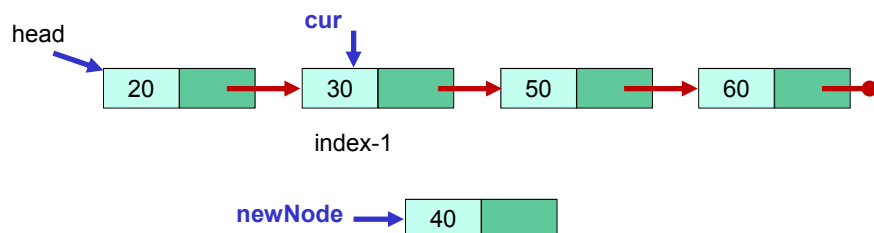
Chú ý thứ tự của lệnh  
Điều gì xảy ra nếu thay đổi thứ  
tự câu lệnh?



1-39

## insertNode(): vị trí giữa, danh sách không trống

- ❑ Sử dụng findNode() để tìm địa chỉ của con trỏ cur
- ❑ Nếu chèn nút mới vào vị trí chỉ số 2, con trỏ cur cần chỉ tới nút có chỉ số 1  
findNode(..., **index-1**)

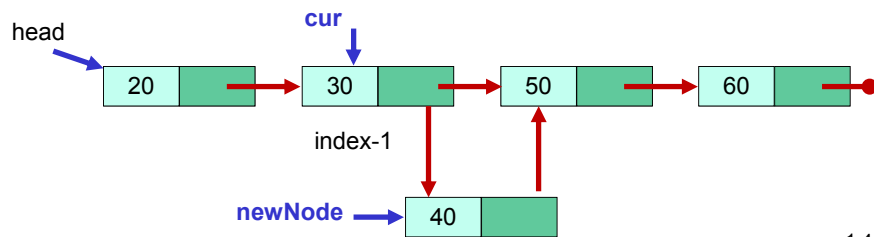


1-40

## insertNode(): vị trí giữa, danh sách không trống

```
// Tìm nút trước vị trí cần chèn
// Tạo nút mới và liên kết lại
➡ if ((cur = findNode(*ptrHead, index-1)) != NULL) {
➡     newNode = malloc(sizeof(ListNode));
➡     newNode->num=40;
➡     newNode->next = cur->next;
➡     cur->next = newNode;
➡ }
```

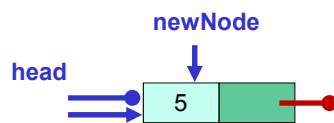
Đúng cả với trường  
hợp thêm vào cuối  
của danh sách



1-41

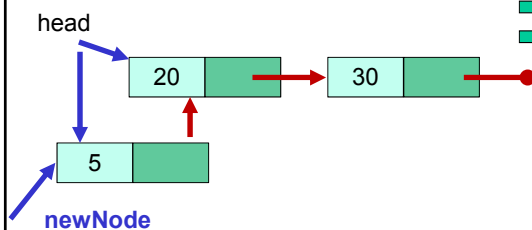
## insertNode(): vị trí bất kì, danh sách trống/không trống

### □ Trường hợp danh sách trống



```
➡ newNode=malloc(sizeof(ListNode));
➡ newNode->num = 5;
➡ newNode->next =head;
➡ head = newNode;
```

### □ Trường hợp chèn vào vị trí chỉ số 0



```
➡ newNode=malloc(sizeof(ListNode));
➡ newNode->num = 5;
➡ newNode->next =head;
➡ head = newNode;
```

1-42



## InsertNode()

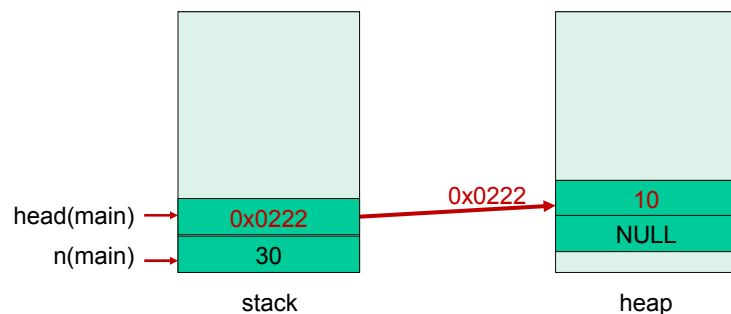
- ❑ Danh sách tham số của hàm insertNode()?  
`int insertNode(ListNode *head, ... )`
- ❑ Gợi ý: Có thể thay đổi địa chỉ lưu trong con trỏ head từ trong hàm insertNode() không?

1-43

## Con trỏ và truyền tham số

```
main()
{
    int n;
    ListNode *head=NULL;
    head=malloc(sizeof(ListNode));
    head->num=10;
    ...
    insertNode(head,n,0);
}
```

```
insertNode(ListNode *head, int n, int i)
{
    if (head==NULL || i == 0) {
        newNode=malloc(sizeof(ListNode));
        newNode->num=n;
        newNode->next =head;
        head = newNode;
    }
    ...
}
```



1-44

## Con trỏ và truyền tham số

```

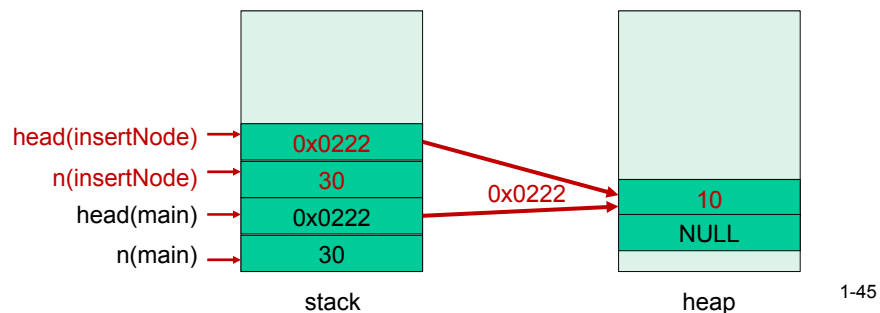
main()
{
    int n;
    ListNode *head=NULL;
    head=malloc(sizeof(ListNode));
    head->num=10;
    ...
    insertNode(head,n,0);
}

```

```

insertNode(ListNode *head, int n, int i)
{
    if (head==NULL || i==0) {
        newNode=malloc(sizeof(ListNode));
        newNode->num=n;
        newNode->next=head;
        head = newNode;
    }
    ...
}

```



1-45

## Con trỏ và truyền tham số

```

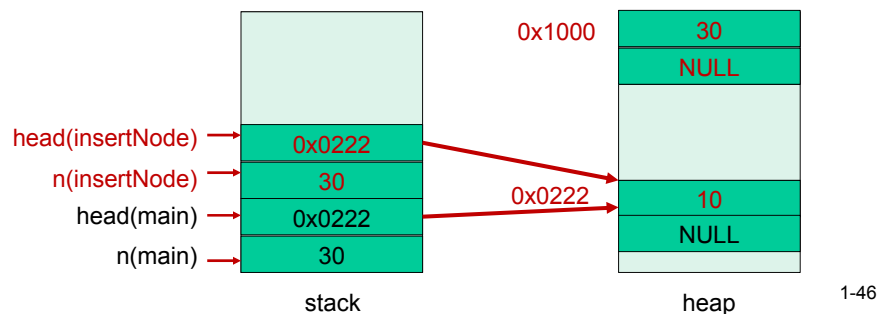
main()
{
    int n;
    ListNode *head=NULL;
    head=malloc(sizeof(ListNode));
    head->num=10;
    ...
    insertNode(head,n,0);
}

```

```

insertNode(ListNode *head, int n, int i)
{
    if (head==NULL || i==0) {
        newNode=malloc(sizeof(ListNode));
        newNode->num=n;
        newNode->next=head;
        head = newNode;
    }
    ...
}

```

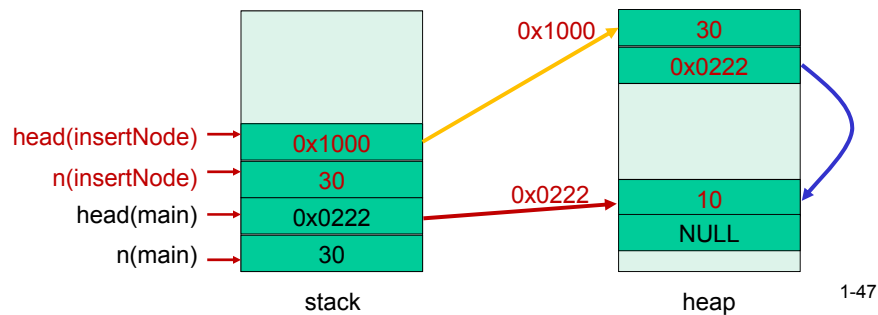


1-46

## Con trỏ và truyền tham số

```
main()
{
    int n;
    ListNode *head=NULL;
    head=malloc(sizeof(ListNode));
    head->num=10;
    ...
    insertNode(head,n,0);
}
```

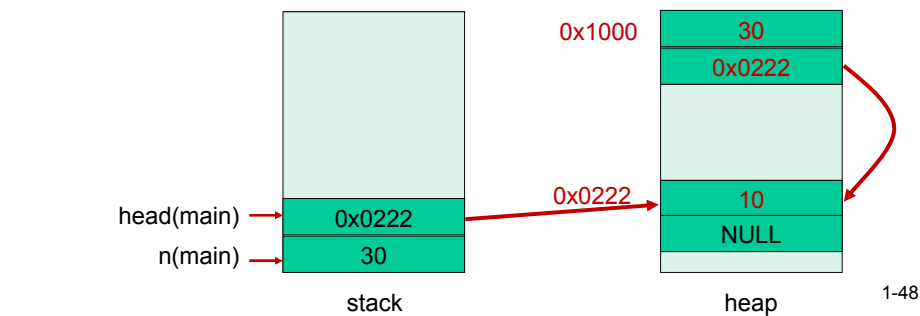
```
insertNode(ListNode *head, int n, int i)
{
    if (head==NULL || i==0) {
        newNode=malloc(sizeof(ListNode));
        newNode->num=n;
        newNode->next=head;
        head = newNode;
    }
    ...
}
```



## Con trỏ và truyền tham số

```
main()
{
    int n;
    ListNode *head=NULL;
    head=malloc(sizeof(ListNode));
    head->num=10;
    ...
    insertNode(head,n,0);
    ...
}
```

```
insertNode(ListNode *head, int n, int i)
{
    if (head==NULL || i==0) {
        newNode=malloc(sizeof(ListNode));
        newNode->num=n;
        newNode->next=head;
        head = newNode;
    }
    ...
}
```



## InsertNode()

head(insertNode)

n(insertNode)

head(main)

n(main)

head(insertNode)	0x1000
n(insertNode)	30
head(main)	0x0222
n(main)	30

stack

- ❑ Không thực hiện được  
int insertNode(ListNode \*head, ... )
- ❑ Nếu muốn chèn một nút vào một danh sách trống hoặc chèn một nút vào vị trí chỉ số 0 thì cần thay đổi địa chỉ chứa trong con trỏ pointer
- ❑ Tuy nhiên, chỉ giá trị của bản sao của head trong hàm insertNode() thay đổi, giá trị của con trỏ head ngoài insertNode() không thay đổi
- ❑ Cách thay đổi một biến trong hàm?

1-49

## Con trỏ và truyền tham số

- ❑ Truyền vào một con trỏ, chỉ tới biến mà chúng ta muốn thay đổi
- ❑ Biến muốn thay đổi là con trỏ head

ListNode \*head



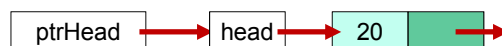
- ❑ Cần truyền vào con trỏ chỉ tới con trỏ head

ListNode \*\*head



- ❑ Có thể viết lại như sau

ListNode \*\*ptrHead



1-50

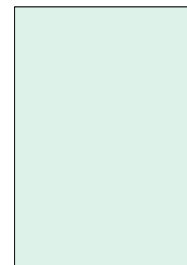
## Con trỏ và truyền tham số

```
main()
{
    int n;
    ListNode **ptrHead=NULL;
    head=malloc(sizeof(ListNode));
    head->num=10;
    head->next= NULL;
    ...
    ptrHead=&head;
    insertNode(ptrHead,30,0);
}
```

```
insertNode(ListNode *ptrHead, int n, int i)
{
    if (*ptrHead==NULL || i=0) {
        newNode=malloc(sizeof(ListNode));
        newNode->num = n;
        newNode->next =*ptrHead;
        *ptrHead= newNode;
    }
    ...
}
```



stack



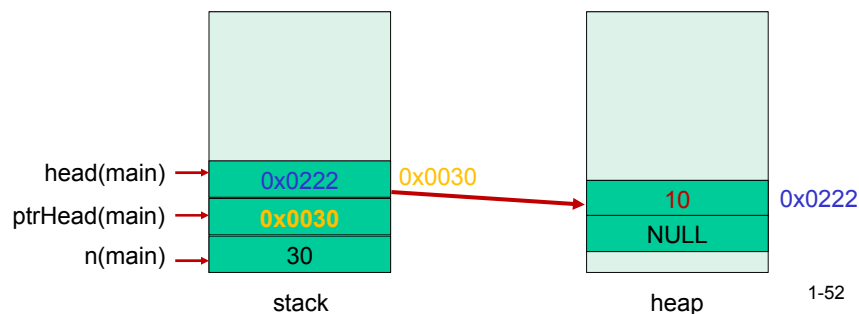
heap

1-51

## Con trỏ và truyền tham số

```
main()
{
    int n;
    ListNode **ptrHead=NULL;
    head=malloc(sizeof(ListNode));
    head->num=10;
    head->next= NULL;
    ...
    ptrHead=&head;
    insertNode(ptrHead,30,0);
}
```

```
insertNode(ListNode *ptrHead, int n, int i)
{
    if (*ptrHead==NULL || i=0) {
        newNode=malloc(sizeof(ListNode));
        newNode->num=n;
        newNode-> next =*ptrHead;
        *ptrHead= newNode;
    }
    ...
}
```



1-52

## Con trỏ và truyền tham số

```

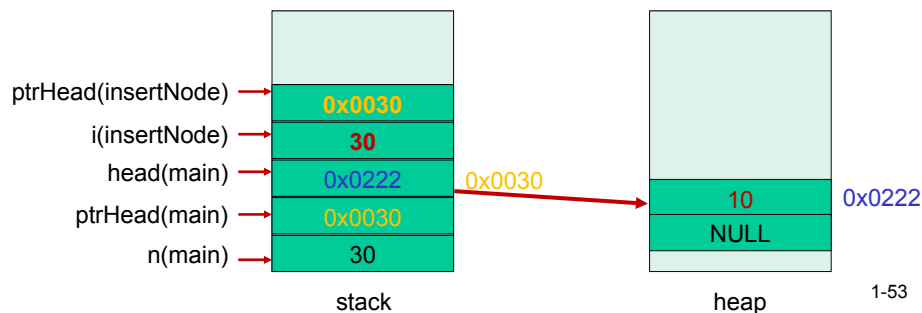
main()
{
    int n;
    ListNode **ptrHead=NULL;
    head=malloc(sizeof(ListNode));
    head->num=10;
    head->next= NULL;
    ...
    ptrHead=&head;
    insertNode(ptrHead,30,0);
}

```

```

insertNode(ListNode *ptrHead, int n, int i)
{
    if (*ptrHead == NULL || i=0) {
        newNode = malloc(sizeof(ListNode));
        newNode->num = n;
        newNode->next = *ptrHead;
        *ptrHead= newNode;
    }
    ...
}

```



1-53

## Con trỏ và truyền tham số

```

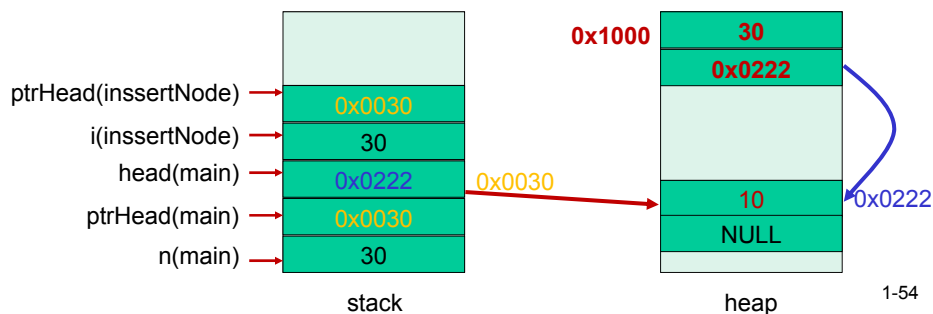
main()
{
    int n;
    ListNode **ptrHead=NULL;
    head=malloc(sizeof(ListNode));
    head->num=10;
    head->next= NULL;
    ...
    ptrHead=&head;
    insertNode(ptrHead,30,0);
}

```

```

insertNode(ListNode *ptrHead, int n, int i)
{
    if (*ptrHead==NULL || i=0) {
        newNode=malloc(sizeof(ListNode));
        newNode->num = n;
        newNode->next = *ptrHead;
        *ptrHead= newNode;
    }
    ...
}

```

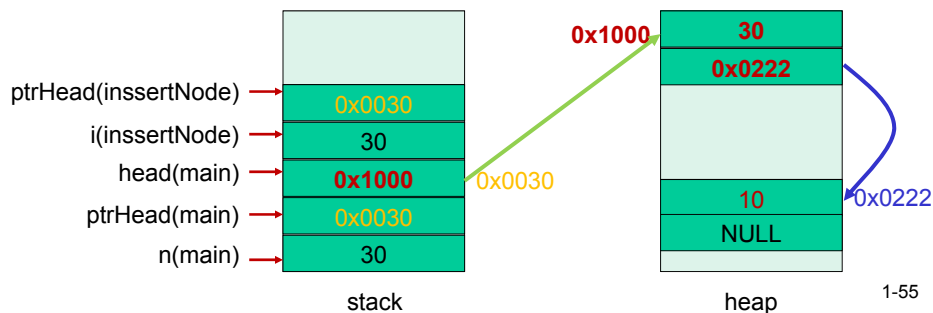


1-54

## Con trỏ và truyền tham số

```
main()
{
    int n;
    ListNode **ptrHead=NULL;
    head=malloc(sizeof(ListNode));
    head->num=10;
    head->next= NULL;
    ...
    ptrHead=&head;
    insertNode(ptrHead,30,0);
}
```

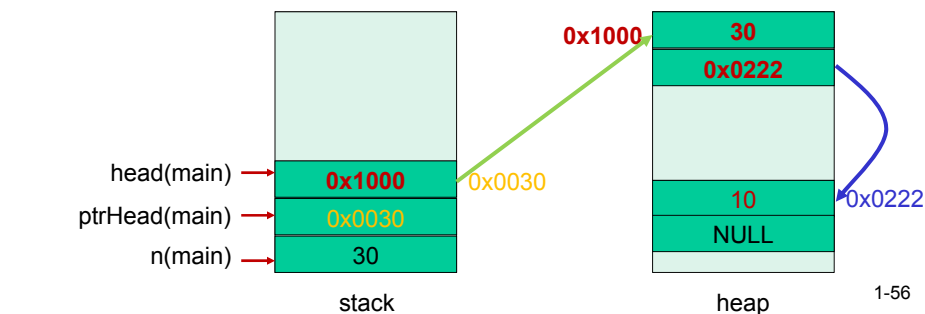
```
insertNode(ListNode *ptrHead, int n, int i)
{
    if (*ptrHead==NULL || i=0) {
        newNode=malloc(sizeof(ListNode));
        newNode->num = n;
        newNode -> next = *ptrHead;
        *ptrHead= newNode;
    }
    ...
}
```



## Con trỏ và truyền tham số

```
main()
{
    int n;
    ListNode **ptrHead=NULL;
    head=malloc(sizeof(ListNode));
    head->num=10;
    head->next= NULL;
    ...
    ptrHead=&head;
    insertNode(ptrHead,30,0);
}
```

```
insertNode(ListNode *ptrHead, int n, int i)
{
    if (*ptrHead==NULL || i=0) {
        newNode=malloc(sizeof(ListNode));
        newNode->num = n;
        newNode->next = *ptrHead;
        *ptrHead= newNode;
    }
    ...
}
```



## InsertNode()

### □ Chúng ta đã hoàn thành code cho hàm insertNode()

- Đã xem xét việc chèn một nút mới tại mọi vị trí
  - Trước
  - Sau
  - Giữa
- Đã xem xét tất cả các trạng thái của danh sách
  - Trống
  - Một nút
  - Nhiều nút

1-57

## InsertNode()

```
void insertNode(ListNode **ptrHead, int index, int value){
    ListNode *cur, *newNode;
    // If empty list or inserting first node, need to update head pointer
    if (*ptrHead == NULL || index == 0){
        newNode = malloc(sizeof(ListNode));
        newNode->num = value;
        newNode->next = *ptrHead;
        *ptrHead = newNode;
    }
    // Find the nodes before and at the target position
    // Create a new node and reconnect the links
    else if ((cur = findNode(*ptrHead, index-1)) != NULL){
        newNode = malloc(sizeof(ListNode));
        newNode->num = value;
        newNode->next = cur->next;
        cur->next = newNode; }
    else printf(" can not insert the new item at index %d!\n", index);
}
```

1-58



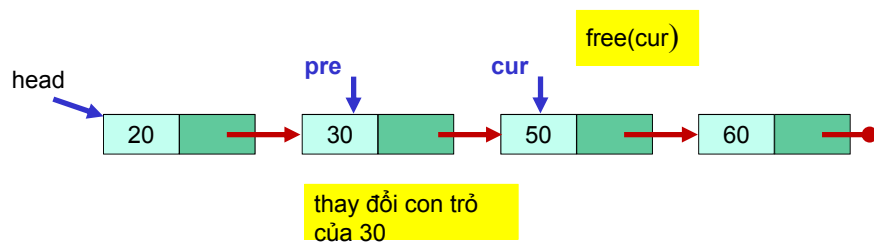
## Phiên bản khác của InsertNode()

```
int insertNode(ListNode **ptrHead, int index, int value){
    ListNode *pre, *cur;
    // If empty list or inserting first node, need to update head pointer
    if (*ptrHead == NULL || index == 0){
        cur = *ptrHead;
        *ptrHead = malloc(sizeof(ListNode));
        (*ptrHead)->num = value;
        (*ptrHead)->next = cur;
        return 0;
    }
    // Find the nodes before and at the target position
    // Create a new node and reconnect the links
    if ((pre = findNode(*ptrHead, index-1)) != NULL){
        cur = pre->next;
        pre->next = malloc(sizeof(ListNode));
        pre->next->num = value;
        pre->next->next = cur;
        return 0;
    }
    return -1;
}
```

1-59

## removeNode(..., 2)

- ❑ Tìm nút, loại bỏ
- ❑ Giải phóng bộ nhớ không sử dụng



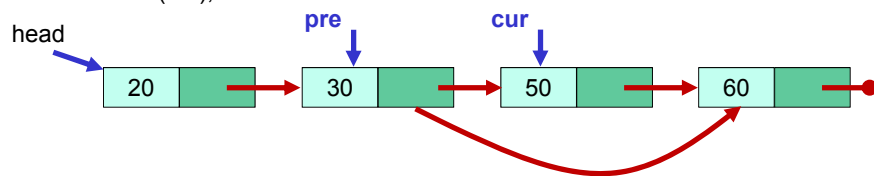
1-60

## removeNode(..., 2)

- Tìm nút, loại bỏ
- Giải phóng bộ nhớ không sử dụng

Nếu `pre->next` là nút cần xóa

```
cur = pre->next;  
pre->next = cur->next;  
free(cur);
```

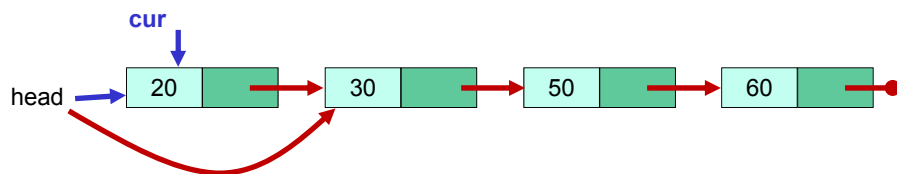


1-61

## removeNode(..., 0)

- Nếu nút muốn xóa là nút đầu tiên

```
cur = head;  
head = cur->next;  
free(cur);
```



1-62

## removeNode(..., i)

❑ void removeNode(ListNode \*\*ptrHead, int index);

1-63

## Chương 2: Mảng và danh sách liên kết

- ❑ Cấu trúc lưu trữ mảng
- ❑ Danh sách liên kết
  - Giới thiệu danh sách liên kết
  - Cài đặt danh sách liên kết
  - Các thao tác trên danh sách liên kết
- ❑ Ngăn xếp
- ❑ Hàng đợi

1-64

## Cấu trúc dữ liệu và giải thuật

- Nội dung bài giảng được biên soạn bởi TS. Phạm Tuấn Minh.

1-65

# Cấu trúc dữ liệu và giải thuật

**TS. Phạm Tuấn Minh**

Khoa Công nghệ Thông tin, Đại học Phenikaa

[minh.phamtuan@phenikaa-uni.edu.vn](mailto:minh.phamtuan@phenikaa-uni.edu.vn)

<https://sites.google.com/site/phamtuanminh/>

## Chương 2: Mảng và danh sách liên kết

- ❑ Cấu trúc lưu trữ mảng
- ❑ Danh sách liên kết
- ❑ **Hàng đợi**
- ❑ Ngăn xếp

## Bài trước... bài này



## Hàng đợi

- ❑ **Ví dụ về hàng đợi**
- ❑ Cấu trúc dữ liệu hàng đợi
- ❑ Cài đặt hàng đợi dùng danh sách liên kết
- ❑ Các thao tác hàng đợi
  - enqueue()
  - dequeue()
  - peek()
  - isEmptyQueue()
- ❑ Ví dụ ứng dụng

## Ví dụ 1

### ❑ Xếp hàng thanh toán tại siêu thị

- Khách hàng xếp trước được thanh toán trước
- Khách hàng muốn thanh toán thì xếp vào cuối của hàng đợi và đợi đến lượt thanh toán



## Ví dụ 2

### ❑ Trình điều khiển máy in:

- Công việc in được gửi tới trình điều khiển máy in tại bất kì thời điểm nào
- Công việc in phải được lưu tới khi chuyển tới máy in
- Công việc in được chuyển tới máy in theo thứ tự yêu cầu gửi tới trước được phục vụ trước
- Công việc in cần một khoảng thời gian để hoàn thành
- Khi một công việc in hoàn thành, công việc in tiếp theo đang chờ sẽ được chuyển tới máy in



## Hàng đợi

- ❑ Ví dụ về hàng đợi
- ❑ **Cấu trúc dữ liệu hàng đợi**
- ❑ Cài đặt hàng đợi dùng danh sách liên kết
- ❑ Các thao tác hàng đợi
  - enqueue()
  - dequeue()
  - peek()
  - isEmptyQueue()
- ❑ Ví dụ ứng dụng

## Mảng, danh sách liên kết, hàng đợi

- ❑ Mảng
  - Cấu trúc dữ liệu truy cập ngẫu nhiên
  - Truy cập trực tiếp bất kì phần tử nào của mảng
    - array[index]
- ❑ Danh sách liên kết
  - Cấu trúc dữ liệu truy cập tuần tự
  - Để truy cập một phần tử phải đi qua các phần tử trước nó
    - cur->next
- ❑ Hàng đợi
  - Cấu trúc dữ liệu tuần tự truy cập có giới hạn



## Cấu trúc dữ liệu hàng đợi

- ❑ Hàng đợi là một cấu trúc dữ liệu hoạt động như hàng đợi gặp trong cuộc sống
  - Ví dụ: Hàng đợi để sử dụng máy ATM, thanh toán
  - Các phần tử chỉ có thể thêm vào cuối hàng đợi
  - Các phần tử chỉ có thể được lấy ra từ đầu hàng đợi
- ❑ Nguyên tắc: First-In, First-Out (FIFO)
  - hoặc Last-In, Last-Out (LIFO)
- ❑ Hàng đợi thường được xây dựng dựa trên các cấu trúc dữ liệu khác
  - Mảng, danh sách liên kết
  - Bài giảng tập trung vào cài đặt hàng đợi dựa trên danh sách liên kết



## Cấu trúc dữ liệu hàng đợi

- ❑ Thao tác chính
  - Enqueue: Thêm vào một phần tử ở cuối hàng đợi
  - Dequeue: Lấy ra một phần tử ở đầu hàng đợi
- ❑ Thao tác hỗ trợ
  - Peek: Xem một phần tử ở đầu hàng đợi nhưng không lấy ra khỏi hàng đợi
  - IsEmptyQueue: Kiểm tra xem hàng đợi có phần tử nào không
- ❑ Các hàm tương ứng
  - enqueue()
  - dequeue()
  - peek()
  - isEmptyQueue()
- ❑ Trong các ví dụ cài đặt hàng đợi sẽ giả sử xử lý với số nguyên
  - Nhưng với danh sách liên kết, có thể xử lý bất kì dữ liệu nào

## Hàng đợi

- ❑ Ví dụ về hàng đợi
- ❑ Cấu trúc dữ liệu hàng đợi
- ❑ **Cài đặt hàng đợi dùng danh sách liên kết**
- ❑ Các thao tác hàng đợi
  - enqueue()
  - dequeue()
  - peek()
  - isEmptyQueue()
- ❑ Ví dụ ứng dụng

## Cài đặt hàng đợi sử dụng danh sách liên kết

- ❑ Chúng ta đã định nghĩa cấu trúc LinkedList bài trước

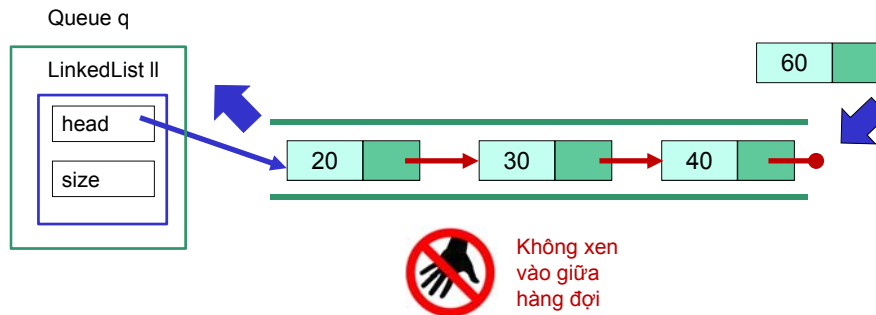
```
typedef struct _linkedlist {
    ListNode *head;
    int size;
} LinkedList;
```
- ❑ Cấu trúc Queue xây dựng dựa trên danh sách liên kết

```
typedef struct _queue {
    LinkedList ll;
} Queue;
```

## Cài đặt hàng đợi sử dụng danh sách liên kết

- Cấu trúc Queue

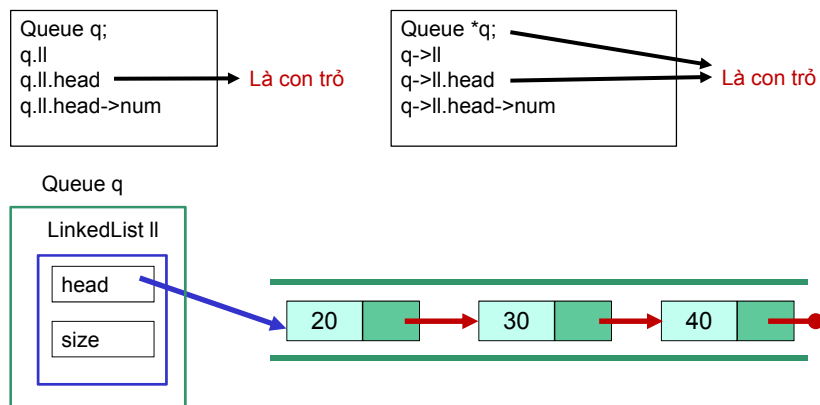
```
typedef struct _queue {
    LinkedList ll;
} Queue;
```
- Sử dụng danh sách liên kết để chứa dữ liệu
- So với danh sách liên kết, thao tác trong hàng đợi có thay đổi



## Cài đặt hàng đợi sử dụng danh sách liên kết

- Cấu trúc Queue

```
typedef struct _queue {
    LinkedList ll;
} Queue;
```

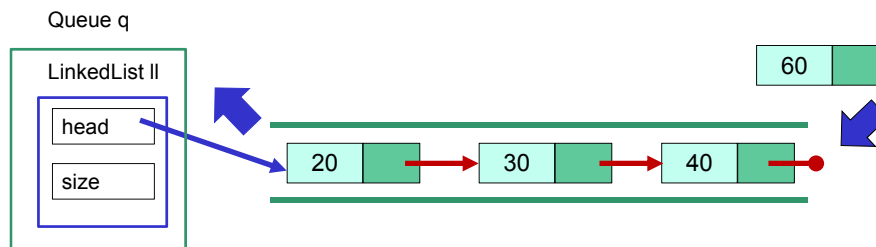


## Hàng đợi

- ❑ Ví dụ về hàng đợi
- ❑ Cấu trúc dữ liệu hàng đợi
- ❑ Cài đặt hàng đợi dùng danh sách liên kết
- ❑ **Các thao tác hàng đợi**
  - enqueue()
  - dequeue()
  - peek()
  - isEmptyQueue()
- ❑ Ví dụ ứng dụng

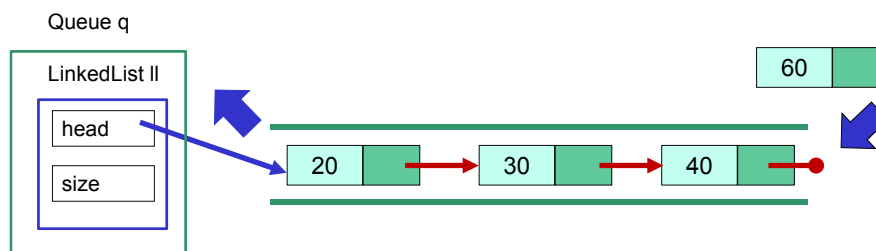
## enqueue()

- ❑ enqueue() là cách duy nhất để thêm một phần tử vào cấu trúc dữ liệu hàng đợi
- ❑ enqueue() chỉ cho phép thêm một phần tử vào cuối hàng đợi
- ❑ Nút đầu tiên của danh sách liên kết là đầu của hàng đợi (hoặc cuối của hàng đợi thì thay đổi code)



## enqueue()

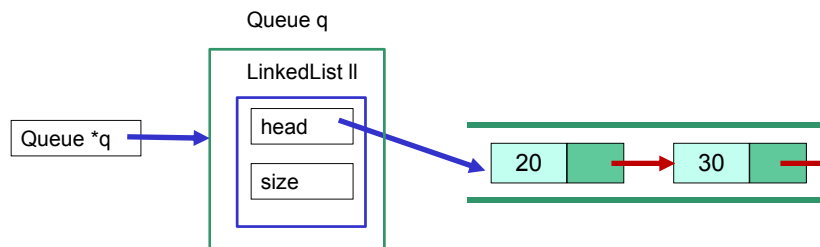
- ❑ Viết hàm enqueue()
  - Khai báo prototype
  - Cài đặt hàm
- ❑ Yêu cầu
  - Sử dụng các hàm cho LinkedList đã cài đặt
  - Chỉ thêm vào cuối hàng đợi



## enqueue()

```
void insertNode(ListNode **ptrHead, int index, int value);  
void enqueue(Queue *q, int item) {  
    insertNode(&(q->ll.head), q->ll.size, item);  
}
```

Thêm một phần tử  
vào hàng đợi ->  
thêm một nút vào  
cuối của danh sách  
liên kết

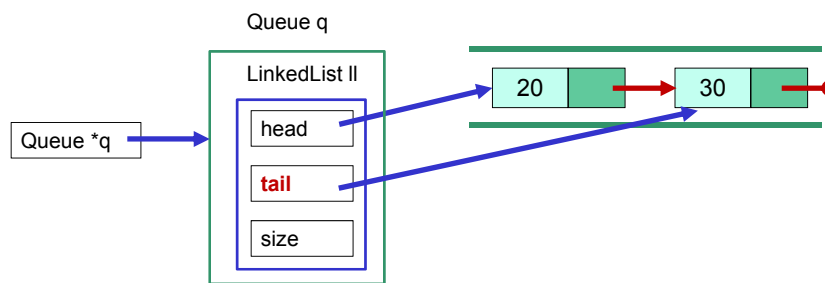


## enqueue()

```
void enqueue(Queue *q, int item) {  
    insertNode(&(q->ll.head), q->ll.size, item);  
}
```

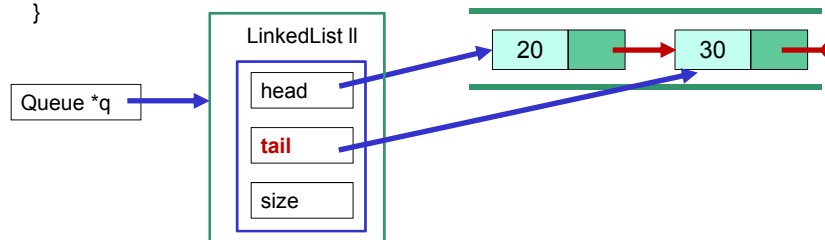
Không hiệu quả nếu  
hàng đợi dài

- Cần sử dụng thêm một **con trỏ tail** để thao tác hiệu quả
  - Con trỏ tail chỉ tới phần tử cuối cùng của danh sách liên kết



## enqueue()

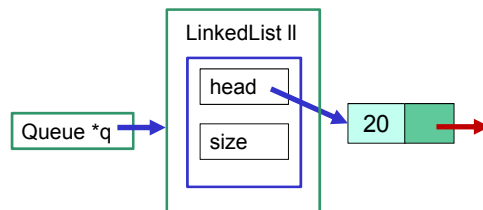
```
void enqueue(Queue *q, int item) {  
    if (q->ll.tail==NULL) {  
        insertNode(&(q->ll.head), 0, item);  
        q->ll.tail=q->ll.head;  
    } else {  
        q->ll.tail->next=malloc(...);  
        q->ll.tail=q->ll.tail->next;  
        q->ll.tail->num = item;  
        q->ll.tail->next=NULL;  
        q->ll.size ++;  
    }  
}
```



## dequeue()

- ❑ Thao tác lấy một phần tử ra khỏi hàng đợi gồm 2 bước
  - Lấy giá trị của nút ở đầu hàng đợi
  - Xóa nút đó trong danh sách liên kết
- ❑ Yêu cầu
  - Sử dụng các hàm cho LinkedList đã cài đặt
  - Chỉ thêm vào cuối hàng đợi

```
int dequeue(Queue *q) {
    int item;
    if (q->ll.head!=NULL) {
        item = ((q->ll).head)->num;
        removeNode(&(q->ll).head, 0);
        (q->ll).size--;
        return item;
    }
    else return NULL_VALUE;
}
```



- ❑ Cần biến item để chứa giá trị vì không thể lấy giá trị sau khi loại bỏ phần tử

## peek()

- ❑ Không tạo sự thay đổi gì trong hàng đợi
- ❑ Lấy giá trị của nút ở đầu hàng đợi
  - Lấy giá trị của nút đầu tiên của danh sách liên kết
  - Không xóa nút này

```
int peek(Queue *q){
    if (q->ll.head!=NULL)
        return (q->ll).head->num;
    else return NULL_VALUE;
}
```

## isEmptyQueue()

- ❑ Kiểm tra xem số phần tử trong hàng đợi có phải bằng 0 không
- ❑ Sử dụng biến size trong cấu trúc LinkedList

```
int isEmptyQueue(Queue *q){  
    if ((q->ll).size == 0) return 1;  
    return 0;  
}
```

## Hàng đợi

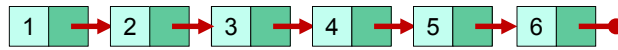
- ❑ Ví dụ về hàng đợi
- ❑ Cấu trúc dữ liệu hàng đợi
- ❑ Cài đặt hàng đợi dùng danh sách liên kết
- ❑ Các thao tác hàng đợi
  - enqueue()
  - dequeue()
  - peek()
  - isEmptyQueue()
- ❑ **Ví dụ ứng dụng**



## Ứng dụng đơn giản kiểm tra hoạt động của Queue

### ❑ Ứng dụng đơn giản

- Thêm vào hàng đợi một vài số nguyên
- Lấy các phần tử trong hàng đợi và đưa giá trị của các phần tử đó ra màn hình



```
int main() {  
    Queue q;  
    q.ll.head = NULL;  
    q.ll.tail = NULL;  
    q.ll.size = 0;  
    enqueue(&q, 1);  
    enqueue(&q, 2);  
    enqueue(&q, 3);  
    enqueue(&q, 4);  
    enqueue(&q, 5);  
    enqueue(&q, 6);  
    while (!isEmptyQueue(&q))  
        printf("%d ", dequeue(&q));  
}
```

## Chương 2: Mảng và danh sách liên kết

- ❑ Cấu trúc lưu trữ mảng
- ❑ Danh sách liên kết
- ❑ Hàng đợi
- ❑ Ngăn xếp

## Cấu trúc dữ liệu và giải thuật

- Nội dung bài giảng được biên soạn bởi TS. Phạm Tuấn Minh.

1-27

# Cấu trúc dữ liệu và giải thuật

**TS. Phạm Tuấn Minh**

Khoa Công nghệ Thông tin, Đại học Phenikaa

[minh.phamtuan@phenikaa-uni.edu.vn](mailto:minh.phamtuan@phenikaa-uni.edu.vn)

<https://sites.google.com/site/phamtuanminh/>

## Chương 2: Mảng và danh sách liên kết

- ❑ Cấu trúc lưu trữ mảng
- ❑ Danh sách liên kết
- ❑ Hàng đợi
- ❑ **Ngăn xếp**

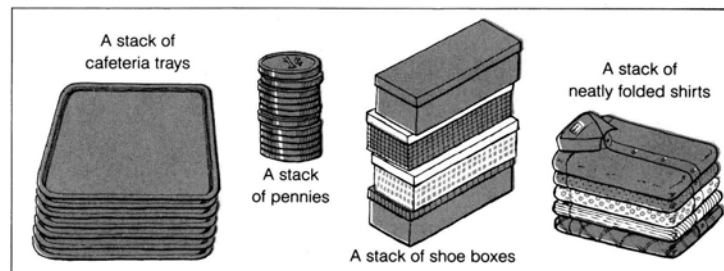
## Ngăn xếp

- ❑ **Ví dụ ngăn xếp**
- ❑ Cấu trúc dữ liệu ngăn xếp
- ❑ Cài đặt ngăn xếp dùng danh sách liên kết
- ❑ Các thao tác trên ngăn xếp
  - push()
  - pop()
  - peek()
  - isEmptyStack()
- ❑ Ví dụ ứng dụng

1-3

## Ví dụ

- ❑ Ngăn xếp là một nhóm có thứ tự các phần tử
  - Các phần tử được thêm vào và lấy ra từ đầu của danh sách

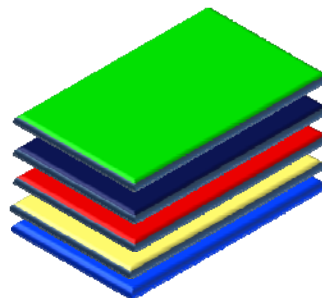


## Ngăn xếp

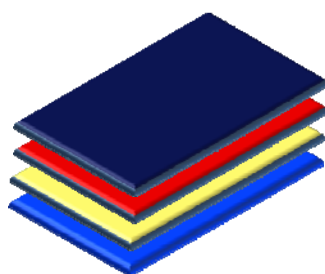
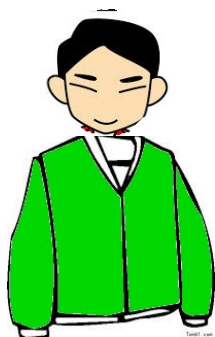
- ❑ Bạn chỉ lấy một cái áo bên trên cùng
- ❑ Bạn chỉ thêm áo vào trên cùng



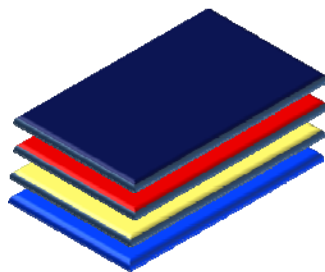
## Hôm nay mặc áo nào



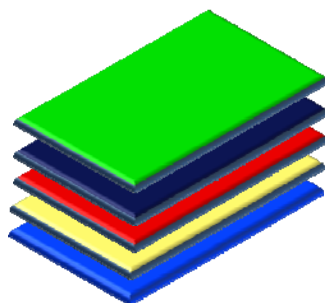
Hôm nay mặc áo nào



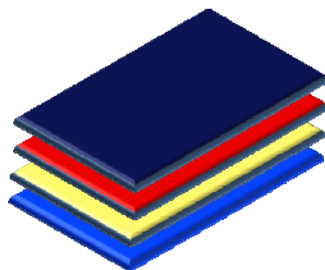
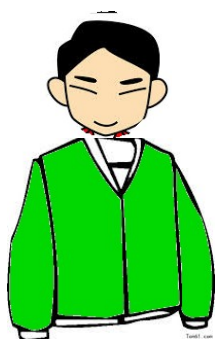
Hôm nay mặc áo nào



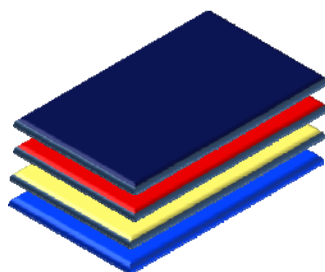
Hôm nay mặc áo nào



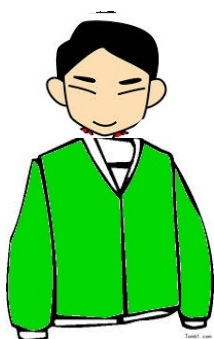
Hôm nay mặc áo nào



Hôm nay mặc áo nào

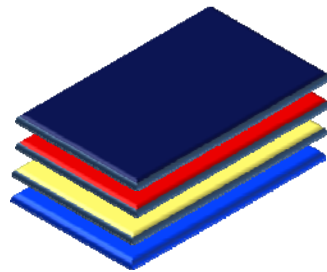
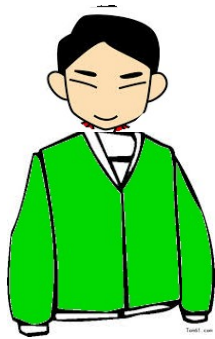


Bạn mặc cùng một cái áo mọi ngày

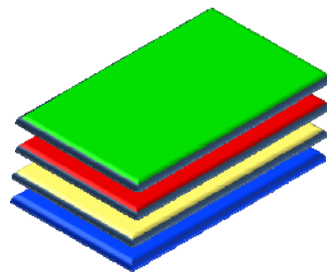
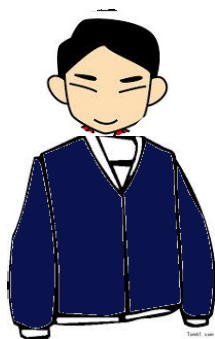




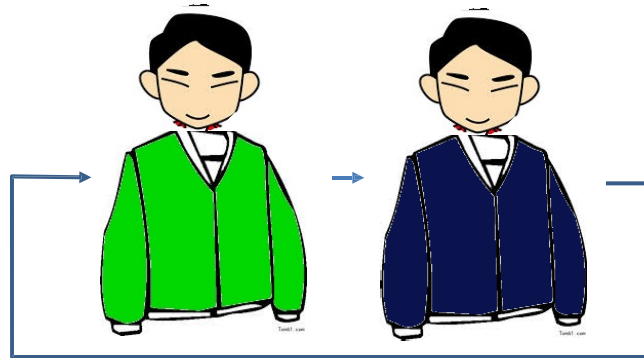
Nếu cần 1 ngày để áo khô



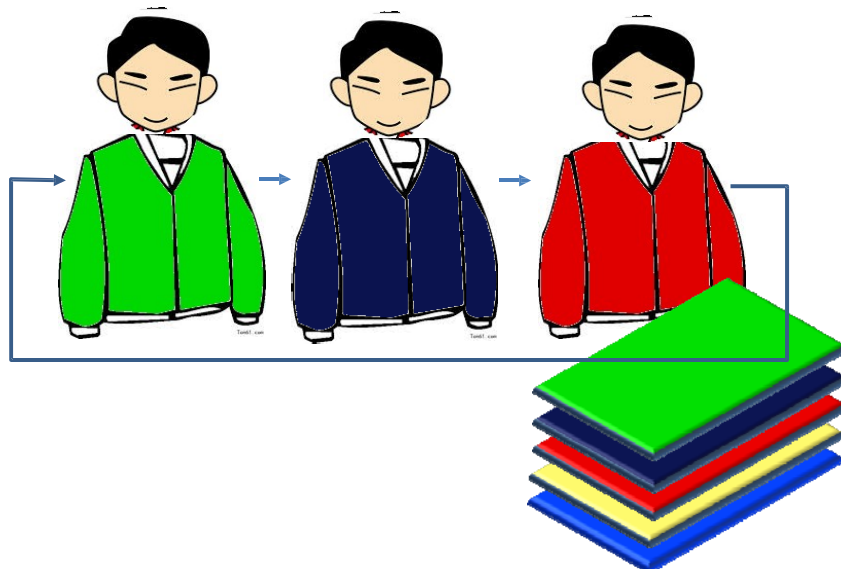
Nếu cần 1 ngày để áo khô



Bạn mặc luân phiên 2 áo



Nếu cần 2 ngày để áo khô



## Ngăn xếp

- ❑ Ví dụ ngăn xếp
- ❑ **Cấu trúc dữ liệu ngăn xếp**
- ❑ Cài đặt ngăn xếp dùng danh sách liên kết
- ❑ Các thao tác trên ngăn xếp
  - push()
  - pop()
  - peek()
  - isEmptyStack()
- ❑ Ví dụ ứng dụng

1-17

## Mảng, danh sách liên kết, hàng đợi, ngăn xếp

- ❑ Mảng
  - Cấu trúc dữ liệu truy cập ngẫu nhiên
  - Truy cập trực tiếp bất kì phần tử nào của mảng
    - array[index]
- ❑ Danh sách liên kết
  - Cấu trúc dữ liệu truy cập tuần tự
  - Để truy cập một phần tử phải đi qua các phần tử trước nó
    - cur->next
- ❑ Hàng đợi
  - Cấu trúc dữ liệu tuần tự truy cập có giới hạn:
    - Đến trước phục vụ trước (FIFO - First In First Out)
- ❑ Ngăn xếp
  - Cấu trúc dữ liệu tuần tự truy cập có giới hạn:
    - Đến cuối phục vụ trước (LIFO - Last In First Out)



1-18

## Danh sách liên kết, hàng đợi, ngăn xếp



## Cấu trúc dữ liệu ngăn xếp

- ❑ Ngăn xếp là một cấu trúc dữ liệu hoạt động như một ngăn xếp vật lý
  - Ví dụ ngăn xếp sách
  - Các phần tử chỉ có thể thêm vào hoặc lấy ra từ đỉnh ngăn xếp
- ❑ Nguyên tắc: Last-In, First-Out (LIFO)
  - Hoặc First-In, Last-Out (FILO)
- ❑ Ngăn xếp được xây dựng dựa trên các cấu trúc dữ liệu khác
  - Ví dụ như mảng, danh sách liên kết
  - Bài giảng tập trung phân tích cài đặt ngăn xếp dựa trên danh sách liên kết

## Ngăn xếp

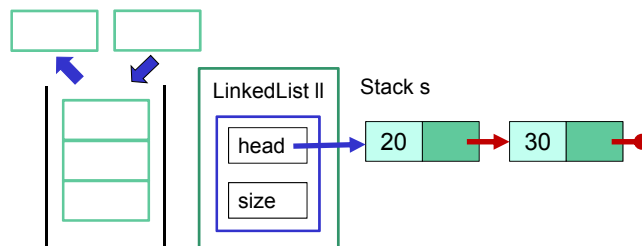
- ❑ Ví dụ ngăn xếp
- ❑ Cấu trúc dữ liệu ngăn xếp
- ❑ **Cài đặt ngăn xếp dùng danh sách liên kết**
- ❑ Các thao tác trên ngăn xếp
  - push()
  - pop()
  - peek()
  - isEmptyStack()
- ❑ Ví dụ ứng dụng

1-21

## Cài đặt ngăn xếp dùng danh sách liên kết

- ❑ Cài đặt cấu trúc Stack dựa trên danh sách liên kết

```
typedef struct _stack{
    LinkedList ll;
    int size;
} Stack;
```
- ❑ Sử dụng danh sách liên kết để chứa dữ liệu
- ❑ Cần điều chỉnh các thao tác thêm, xóa phần tử



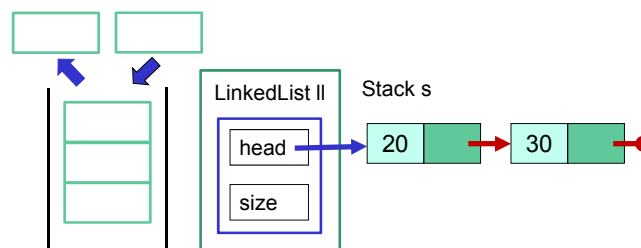
## Ngăn xếp

- ❑ Ví dụ ngăn xếp
- ❑ Cấu trúc dữ liệu ngăn xếp
- ❑ Cài đặt ngăn xếp dùng danh sách liên kết
- ❑ **Các thao tác trên ngăn xếp**
  - push()
  - pop()
  - peek()
  - isEmptyStack()
- ❑ Ví dụ ứng dụng

1-23

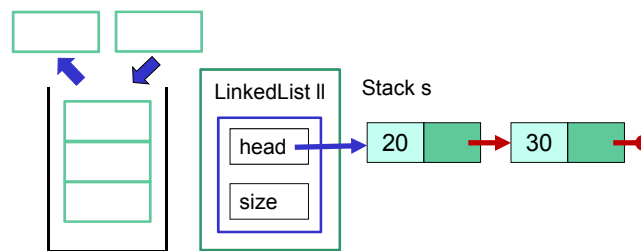
## push()

- ❑ push() là cách duy nhất để thêm một phần tử vào cấu trúc dữ liệu ngăn xếp
- ❑ push() chỉ thao tác trên đỉnh của ngăn xếp
- ❑ Sử dụng danh sách liên kết để chứa dữ liệu của ngăn xếp, phần tử đầu tiên của danh sách liên kết là đỉnh hay đáy của ngăn xếp?



## Cài đặt hàm push()

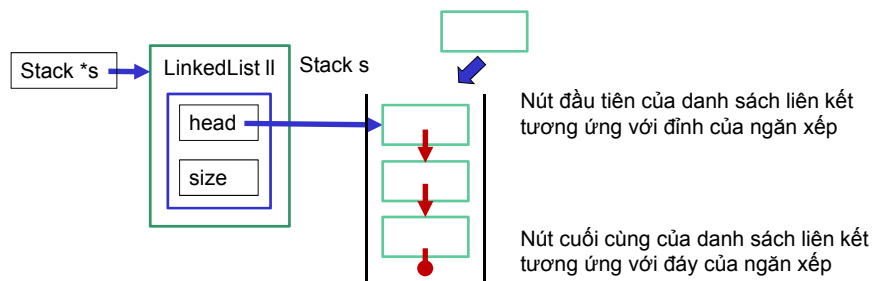
- ❑ Cài đặt hàm push()
  - Khai báo prototype
  - Cài đặt hàm
- ❑ Yêu cầu
  - Sử dụng các hàm của LinkedList đã cài đặt
  - Chỉ thao tác trên đỉnh ngăn xếp



## Cài đặt hàm push()

```
void push(Stack *s, int item ) {  
    insertNode(&(s->ll.head), 0, item);  
}
```

Đẩy một nút mới vào ngăn xếp -> thêm một nút mới vào đầu của danh sách liên kết



## Cài đặt hàm push()

```
void push(Stack *s, int item ) {  
    insertNode(&(s->ll.head), 0, item);  
    s->ll.size++;  
}
```

**Hiệu quả**

- ❑ Có thể thêm nút mới vào cuối của danh sách liên kết
  - Nếu nút cuối biểu diễn nút đỉnh của ngăn xếp
  - Khi đó cần sử dụng con trỏ tail để thực hiện thao tác hiệu quả

## isEmptyStack()

- ❑ Kiểm tra xem số phần tử trong ngăn xếp có phải bằng 0 không
- ❑ Sử dụng biến size trong cấu trúc LinkedList

```
int isEmptyStack(Stack *s) {  
    if ((s->ll).size == 0) return 1;  
    return 0;  
}
```

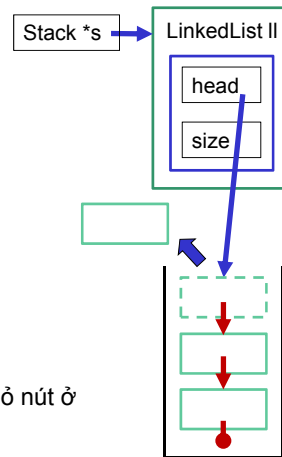


## pop()

- ❑ Thao tác lấy một giá trị khỏi ngăn xếp gồm 2 bước
  - Lấy giá trị của nút ở đỉnh của danh sách liên kết
  - Xóa nút này khỏi danh sách liên kết

```
int pop(Stack *s) {  
    int item;  
    if (!isEmptyStack(s)) {  
        item = ((s->ll).head)->num;  
        removeNode(&(s->ll), 0);  
        (s->ll).size--;  
        return item;  
    }  
    else return NULL;  
}
```

- ❑ Cần biến item để chứa giá trị trước khi loại bỏ nút ở đỉnh ngăn xếp



## peek()

- ❑ peek()
  - Lấy giá trị của nút ở đỉnh của danh sách liên kết
  - Không xóa nút này khỏi danh sách liên kết

```
int peek(Stack *s) {  
    if ((s->ll).head)!=NULL)  
        return ((s->ll).head)->num;  
    else return NULL_VALUE;  
}
```

## Ngăn xếp

- ❑ Ví dụ ngăn xếp
- ❑ Cấu trúc dữ liệu ngăn xếp
- ❑ Cài đặt ngăn xếp dùng danh sách liên kết
- ❑ Các thao tác trên ngăn xếp
  - push()
  - pop()
  - peek()
  - isEmptyStack()
- ❑ **Ví dụ ứng dụng**

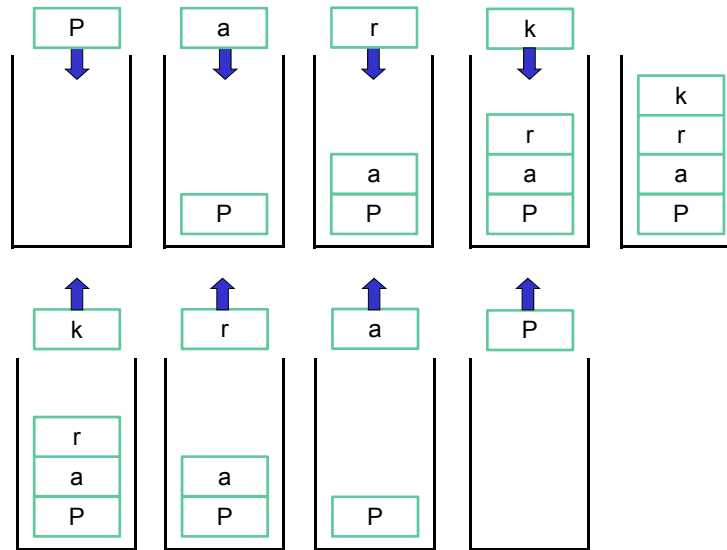
1-31

## Ứng dụng 1

- ❑ Đảo ngược một chuỗi: Park -> kraP
- ❑ Dùng ngăn xếp:
  - Đẩy lần lượt kí tự vào ngăn xếp
  - Khi không còn kí tự nào trong chuỗi ban đầu, lấy lần lượt từng kí tự ra khỏi ngăn xếp

1-32

## Ứng dụng 1



1-33

## Ứng dụng 2

□ Tương tự ứng dụng 1, nhưng với số

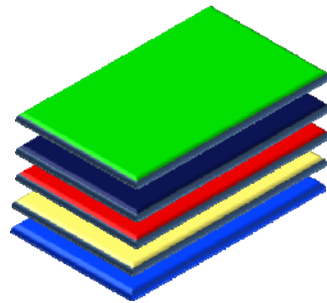
```
int main() {
    int i = 0;
    Stack s;
    s.ll.head = NULL;
    printf("Enter a number: ");
    scanf("%d", &i);
    while (i != -1) {
        push(&s, i);
        printf("Enter a number: ");
        scanf("%d", &i);
    }
    printf("Popping stack: ");
    while (!isEmptyStack(&s))
        printf("%d ", pop(&s));
    return 0;
}
```

1-34

## Ứng dụng 3: Trang phục hôm nay



- Bạn có 5 áo với màu khác nhau: 1-blue, 2-yellow, 3-red, 4-deep blue, 5-green
- Giả sử cần  $M=2$  ngày để khô áo
- Mô phỏng trang phục của bạn trong 10 ngày



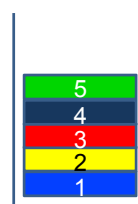
### whichClother()



dryLeftDay[]

-1   -1   -1   -1   -1

day =1



Stack s

## whichClother()

```
void main(){
    int day = 1;
    int j, clothes;
    Stack s;
    int dryLeftDay[6]; //với mỗi áo (số 1 tới số 5), cần bao ngày nữa sẽ khô
    s.ll.head = NULL; s.ll.size=0; //khởi tạo stack s.
    for (j=1; j<=5; j++) push(&s, j);
    for (j=1; j<=5; j++) dryLeftDay[j] = -1; //<0 nghĩa là áo khô
    while (day <=10) {
        for (j=1; j<=5; j++)
            dryLeftDay[j]--; //giảm 1 ngày cần để khô
        clothes=pop(&s);
        printf("Day %d is clothes No. %d. \n", day, clothes);
        dryLeftDay[clothes]=M; //ví dụ, M=2, cần 2 ngày để khô
        for (j=1; j<=5; j++)
            if (dryLeftDay[j]==0) //nghĩa là áo vừa khô
                push(&s, j);
        day++;
    }
}
```

## Chương 2: Mảng và danh sách liên kết

- ❑ Cấu trúc lưu trữ mảng
- ❑ Danh sách liên kết
- ❑ Hàng đợi
- ❑ Ngăn xếp

## Cấu trúc dữ liệu và giải thuật

- Nội dung bài giảng được biên soạn bởi TS. Phạm Tuấn Minh.

1-39

# Cấu trúc dữ liệu và giải thuật

**TS. Phạm Tuấn Minh**

Khoa Công nghệ Thông tin, Đại học Phenikaa

[minh.phamtuan@phenikaa-uni.edu.vn](mailto:minh.phamtuan@phenikaa-uni.edu.vn)

<https://sites.google.com/site/phamtuanminh/>

## Chương 3: Cây và bảng băm

- ❑ Các khái niệm cây
- ❑ Cây nhị phân tìm kiếm
- ❑ Cây AVL
- ❑ Bảng băm

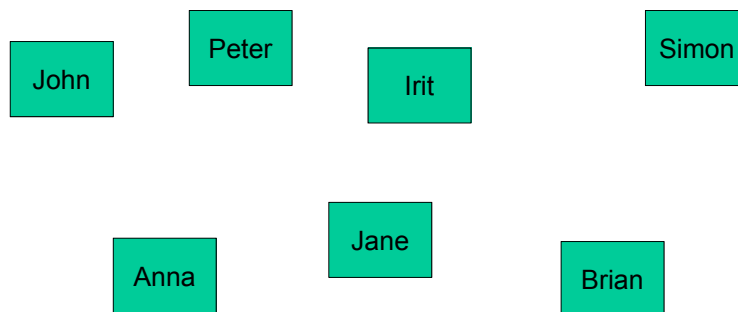
## Cây

- ❑ Các cấu trúc dữ liệu không tuyến tính
- ❑ Cấu trúc dữ liệu cây
  - Cây nhị phân
- ❑ Cài đặt cây nhị phân
- ❑ Duyệt cây nhị phân
- ❑ Ứng dụng ví dụ

1-3

## Ví dụ về sử dụng cấu trúc dữ liệu

- ❑ Giả sử có một tập các tên



- ❑ Quản lý tập tên như thế nào?

1-4



## Cấu trúc dữ liệu tuyến tính

- Nếu các tên xếp danh sách xếp hàng



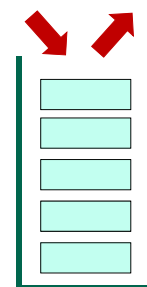
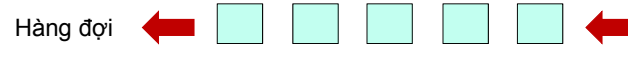
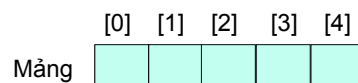
### Danh sách

- Quản lý danh sách dữ liệu trên như thế nào ?

1-5

## Cấu trúc dữ liệu tuyến tính

- Mảng, danh sách liên kết, hàng đợi, ngăn xếp

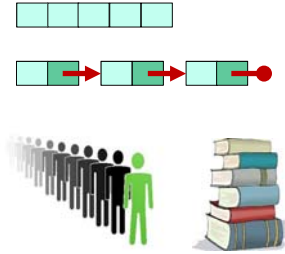


Ngăn xếp 1-6

## Cấu trúc dữ liệu đã học

### □ Tuyến tính:

- Tất cả các phần tử được sắp xếp có thứ tự
- Truy cập ngẫu nhiên
  - Mảng
- Truy cập tuần tự
  - Danh sách liên kết
- Truy cập tuần tự có hạn chế
  - Hàng đợi
  - Ngăn xếp

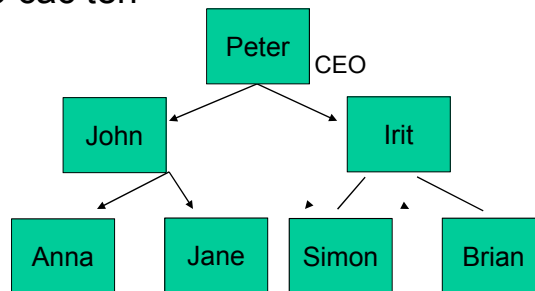


- Sử dụng để chứa danh sách các số, danh sách tên người
  - Cấu trúc tuyến tính

1-7

## Cấu trúc dữ liệu không tuyến tính

### □ Tập các tên



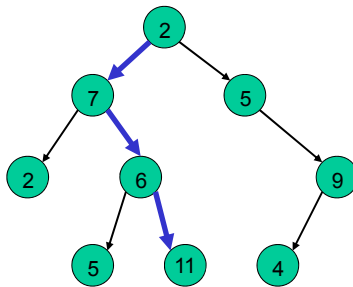
Cây

- Tổ chức công ty: Không dùng cấu trúc tuyến tính để lưu trữ do quan hệ phân cấp

1-8

## Cấu trúc dữ liệu cây

- Vẫn sử dụng các biểu diễn nút và liên kết
- Đặc điểm mới:
  - Mỗi nút có liên kết tới nhiều hơn một nút khác
  - Không có lặp



Nếu đi theo một đường trên cây, sẽ nhận được một danh sách liên kết

1-9

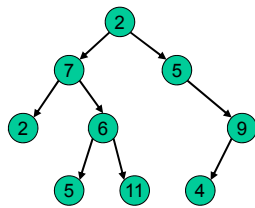
## Cây

- Các cấu trúc dữ liệu không tuyến tính
- **Cấu trúc dữ liệu cây**
  - Cây nhị phân
- Cài đặt cây nhị phân
- Duyệt cây nhị phân
- Ứng dụng ví dụ

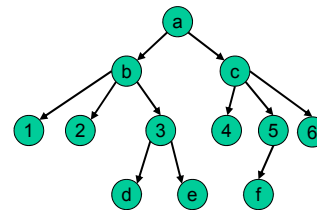
1-10

## Cấu trúc dữ liệu cây

- Cấu trúc dữ liệu cây dạng như cây: gốc, nhánh, lá
  - Chỉ có một nút gốc
  - Mỗi nút nhánh trở tới một số nút khác
  - Mỗi nút chỉ có một nút cha (nút trở tới nó), trừ nút gốc



Cây nhị phân



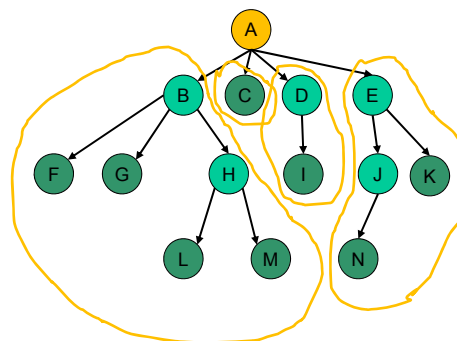
Cây tổng quát

- Cây tổng quát: Mỗi nút có liên kết tới không giới hạn các nút khác
- Cây nhị phân: Mỗi nút có liên kết tới tối đa hai nút

1-11

## Cấu trúc dữ liệu cây

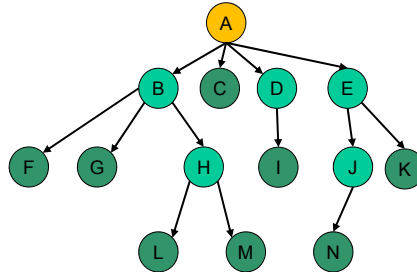
- Tương tự khái niệm cây gia đình
  - Một nút đặc biệt: nút gốc (root)
  - Mỗi nút có thể có nhiều con (children node)
  - Mỗi nút (trừ nút gốc) có một nút cha (parent node)
  - Các con của cùng cha là anh chị em (sibling node)
  - Cây con (subtree): Nút con và các nút cháu (descendant node) tạo thành cây con



1-12

## Cấu trúc dữ liệu cây

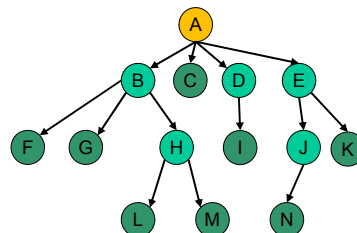
- ❑ Một cây bao gồm các nút
- ❑ Mỗi nút chứa một giá trị
- ❑ Các kiểu nút
  - Nút gốc (root): cây chỉ có một nút gốc, nút gốc không có nút cha
  - Nút trong (internal node): Là nút có nút con
  - Nút lá (leaf node): Là nút không có nút con



1-13

## Sử dụng cây

- ❑ Các mô hình có quan hệ phân cấp giữa các phần tử
  - Chuỗi mệnh lệnh trong quân đội
  - Cấu trúc nhân sự của một công ty
- ❑ Cấu trúc cây cho phép
  - Mô tả một số bài toán tối ưu, mô tả trò chơi,...
  - Cho phép thực hiện nhanh
    - Tìm kiếm một nút với giá trị của nó
    - Thêm một giá trị vào danh sách
    - Xóa một giá trị từ danh sách



1-14

## Ví dụ ứng dụng của cây

### ❑ Có thông tin sau

- F có con là G
- J có các con là I và K
- B có các con là A và C
- L có các con J và M
- H có các con là E và L
- C có con là D
- E có các con là B và F

### ❑ Trả lời các câu hỏi

- Ai không có con?
- Ai là con cháu của L?
- Ai là tổ tiên của J?
- Ai có chính các 3 con cháu?

1-15

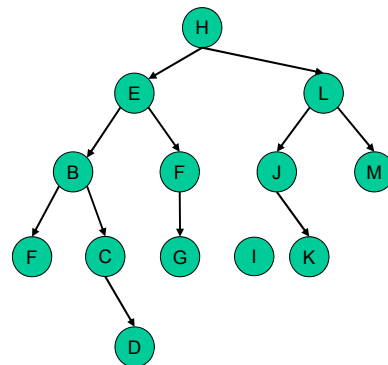
## Ví dụ ứng dụng của cây

### ❑ Biểu diễn cây

- F có con là G
- J có các con là I và K
- B có các con là A và C
- L có các con J và M
- H có các con là E và L
- C có con là D
- E có các con là B và F

### ❑ Trả lời các câu hỏi

- Ai không có con?
- Ai là con cháu của L?
- Ai là tổ tiên của J?
- Ai có chính xác 3 con cháu?



1-16

## Cây

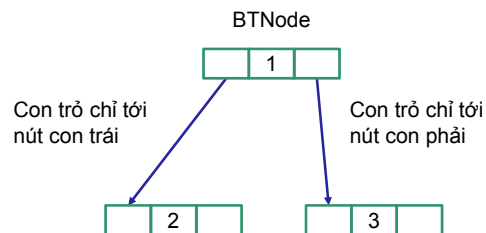
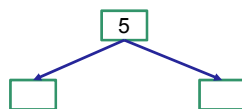
- ❑ Các cấu trúc dữ liệu không tuyến tính
- ❑ Cấu trúc dữ liệu cây
  - Cây nhị phân
- ❑ **Cài đặt cây nhị phân**
- ❑ Duyệt cây nhị phân
- ❑ Ứng dụng ví dụ

1-17

## Cài đặt

- ❑ Nhắc lại cài đặt LinkedList
  - Nút có liên kết tới nhiều nhất một nút
  - Định nghĩa một ListNode với một con trỏ next và dữ liệu item
- ❑ BinaryTree
  - Một nút có liên kết tới nhiều nhất hai nút khác
  - Định nghĩa BTreeNode với
    - Hai con trỏ
    - Một đơn vị dữ liệu

```
typedef struct _listnode{  
    int item;  
    struct _listnode *next;  
} ListNode;
```



1-18

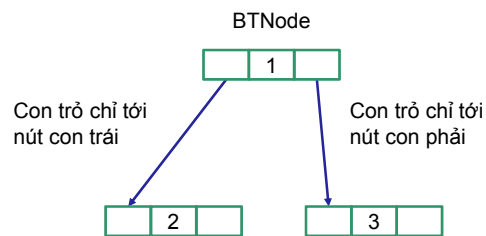
## BTNode

```
typedef struct _listnode {  
    int item;  
    struct _listnode *next;  
} ListNode;
```

- Ví dụ một BTNode đơn giản chứa một số nguyên

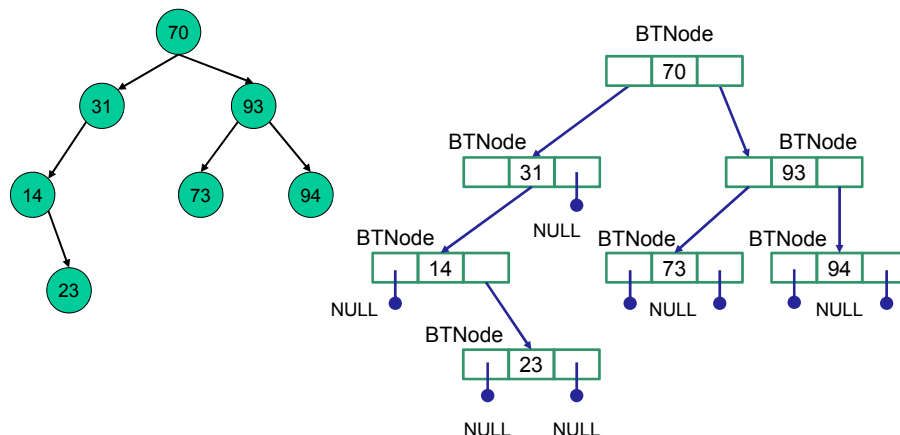
- Kiểu của item có thể là kí tự, chuỗi, cấu trúc,...

```
typedef struct _bnode {  
    int item;  
    struct _bnode *left;  
    struct _bnode *right;  
} BTreeNode;
```



1-19

## Ví dụ cây nhị phân



1-20



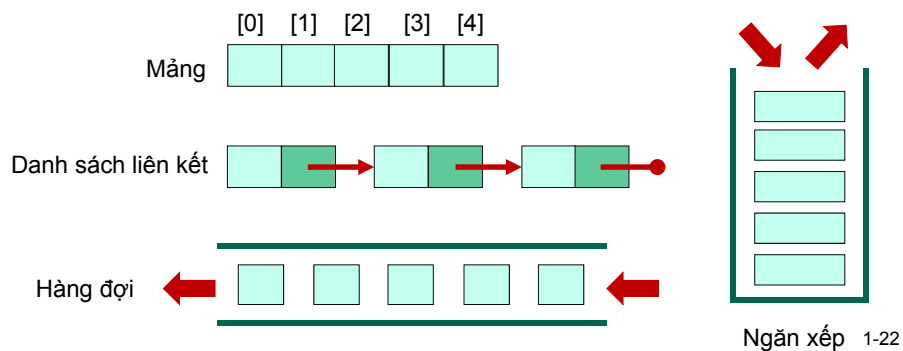
## Cây

- ❑ Các cấu trúc dữ liệu không tuyến tính
- ❑ Cấu trúc dữ liệu cây
  - Cây nhị phân
- ❑ Cài đặt cây nhị phân
- ❑ **Duyệt cây nhị phân**
- ❑ Ứng dụng ví dụ

1-21

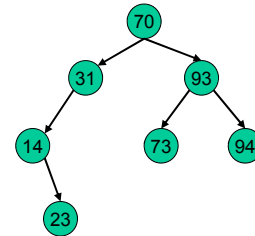
## Duyệt cây

- ❑ Cho một cấu trúc dữ liệu tuyến tính và một phần tử cụ thể:
  - Rõ ràng xác định mỗi nút có một nút trước và một nút sau



## Duyệt cây

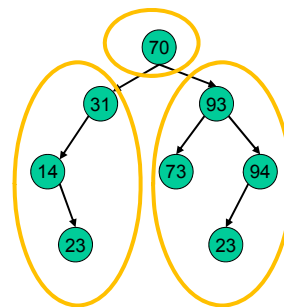
- ❑ Cho một cấu trúc dữ liệu tuyến tính và một phần tử cụ thể:
  - Rõ ràng xác định mỗi nút có một nút trước và một nút sau
- ❑ Cây là cấu trúc dữ liệu không tuyến tính
  - Lấy dữ liệu từ cây nhị phân như thế nào
  - Thứ tự duyệt qua các phần tử như thế nào? trái/trái/trái, rồi trái/trái/phải, rồi?
- ❑ Cần có cách để thăm mọi nút trên cây, không đi lặp lại



1-23

## Duyệt cây

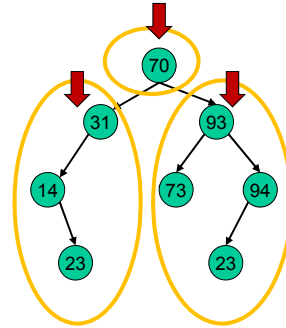
- ❑ Tại sao cần thực hiện duyệt cây
  - Duyệt cây là cơ sở cho các thao tác khác
- ❑ Thao tác rất phổ biến: Duyệt cây, tại mỗi nút, thực hiện một số công việc
- ❑ Ví dụ: Đếm số nút của cây
  - Tại mỗi nút N, kích thước của cây con đó = kích thước của cây con trái của N + kích thước của cây con phải của N + 1



1-24

## Duyệt cây

- Duyệt cây là quá trình đệ quy
  - Đệ quy: Quá trình lặp tại các phần tử các thao tác giống nhau, chia bài toán thành các bài toán con
  - Tại mỗi nút: Thăm nút đó và hai nút con
- Đảm bảo thăm mọi nút và mỗi nút chỉ thăm một lần



1-25

Trong main(), gọi TreeTraversal(root)

TreeTraversal(Node N):

Visit N;  
if (N has left child)

TreeTraversal(Node N):

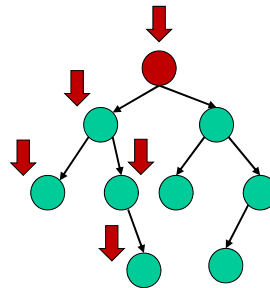
Visit N;  
if (N has left child)

TreeTraversal(Node N):

Visit N;  
if (N has left child)  
TreeTraversal(LeftChild);  
if (N has right child)

TreeTraversal(Node N):

Visit N;  
if (N has left child)  
TreeTraversal(Node N):  
Visit N;  
if (N has left child)  
TreeTraversal(LeftChild);  
if (N has right child)  
TreeTraversal(RightChild);  
Return; // return to parent



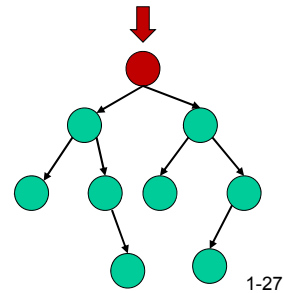
1-26

## Duyệt cây

```
TreeTraversal(Node N):  
  Visit N;  
  if (N has left child)  
    TreeTraversal(LeftChild);  
  if (N has right child)  
    TreeTraversal(RightChild);  
  Return; // return to parent
```

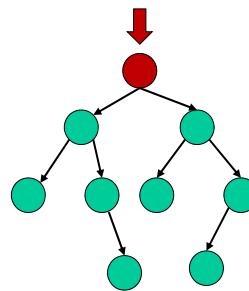
```
TreeTraversal2(Node N):  
  if N==NULL return;  
  Visit N;  
  TreeTraversal2(LeftChild);  
  TreeTraversal2(RightChild);  
  Return; // return to parent
```

- ❑ Mẫu 1: Cần kiểm tra sự tồn tại của nút con trái và phải trước khi duyệt
- ❑ Mẫu 2: Luôn thăm nút con, sau đó kiểm tra liên kết có NULL không
- ❑ Độ quy: Gọi lại trong chính hàm của nó
- ❑ Duyệt theo chiều sâu: Thăm các nút đi theo chiều sâu nhất có thể, trước khi quay lại và thăm nút bên



## Cài đặt TreeTraversal2()

```
void TreeTraversal2(BTNode *cur) {  
  If (cur == NULL) return;  
  PrintNode(cur); //visit cur;  
  TreeTraversal2(cur->left);  
  TreeTraversal2(cur->right);  
}
```



## Cây

- ❑ Các cấu trúc dữ liệu không tuyến tính
- ❑ Cấu trúc dữ liệu cây
  - Cây nhị phân
- ❑ Cài đặt cây nhị phân
- ❑ Duyệt cây nhị phân
- ❑ **Ứng dụng ví dụ**

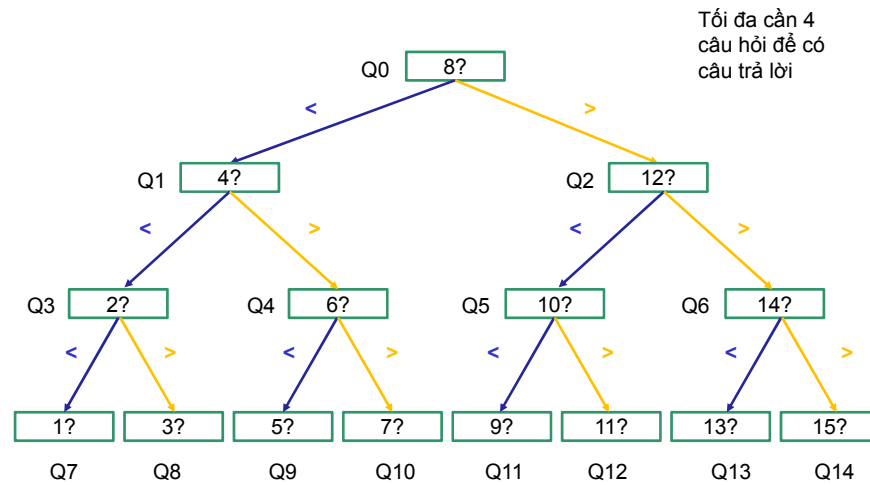
1-29

## Trò chơi đoán số

- ❑ Người chơi nghĩ một số giữa 1 và 63
- ❑ Máy tính đưa ra câu hỏi, người chơi sẽ trả lời Yes/Too big/Too small
- ❑ Trò chơi kết thúc khi câu trả lời là Yes
- ❑ Ví dụ: khi người chơi nghĩ là 23, 6 câu hỏi để tìm được câu trả lời Yes là
  - 32? => too big
  - 6? => too small
  - 24? => too big
  - 20? => too small
  - 22? => too small
  - 23? => Yes!

1-30

## Trò chơi đoán số [1, 15]



1-31

## Trò chơi đoán số [1, 15]

```

void main() {
    int i; // 0-Yes; -1 -too big; 1-too small
    BTNode root, *cur;

    buildTree(&root, 8, 4);
    cur=&root;
    do {
        printf("is it %d?\n", cur->item);
        scanf("%d", &i);
        if (i== -1)
            cur=cur->left;
        else if (i==1)
            cur=cur->right;
    } while (i!=0 && cur!=NULL);
    if (i==0) printf("I guess it! %d\n", cur->item);
}
  
```

```

void buildTree(BTNode *r, int v, int h) {
    r->item=v;
    if (h>0) {
        r->left=malloc(sizeof(BTNode));
        r->right=malloc(sizeof(BTNode));
        buildTree(r->left, v-h, h/2);
        buildTree(r->right, v+h, h/2);
    }
    else {
        r->left=NULL;
        r->right=NULL;
    }
    return;
}
  
```

1-32

## Cấu trúc dữ liệu và giải thuật

- Nội dung bài giảng được biên soạn bởi TS. Phạm Tuấn Minh.

1-33

# Cấu trúc dữ liệu và giải thuật

**TS. Phạm Tuấn Minh**

Khoa Công nghệ Thông tin, Đại học Phenikaa

minh.phamtuan@phenikaa-uni.edu.vn

<https://sites.google.com/site/phamtuanminh/>

## Chương 3: Cây và bảng băm

- ❑ Các khái niệm cây
- ❑ Cây nhị phân tìm kiếm
- ❑ Cây AVL
- ❑ Bảng băm



## Duyệt cây nhị phân

### □ Thứ tự duyệt cây

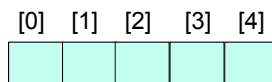
- Thứ tự trước
- Thứ tự giữa
- Thứ tự sau

### □ Ví dụ ứng dụng

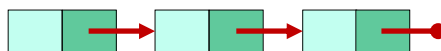
- Đếm số nút trên cây nhị phân
- Tìm nút cháu
- Tính chiều cao của nút

1-3

## Duyệt mảng và danh sách liên kết



$i = i + 1;$



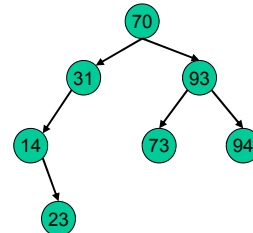
$cur = cur \rightarrow next;$

1-4

## Duyệt cây

### □ Duyệt cây:

- Thăm mọi nút theo một thủ tục xác định các bước rõ ràng và thủ tục có thể thực hiện lặp lại trên cây
- Không thăm lặp lại các nút



1-5

## Giải thích lặp đệ quy

```
Void tellStory(){  
    printf("Once upon a time there was a mountain. ");  
    printf("On the mountain, there was a temple. ");  
    printf("In the temple was an monk, telling a story. ");  
    printf("What is the story? It is: ");  
    tellStory();  
}
```

Không kết thúc!



1-6

## Giải thích lặp đệ quy

```
Void tellStory(){  
    printf("Once upon a time there was a mountain. ");  
    printf("On the mountain, there was a temple. ");  
    printf("In the temple was an monk, telling a story. ");  
    printf("What is the story? It is: ");  
    tellStory();  
    printf("The monk asked: do you like it?\n");  
}
```

Không bao giờ hiện "The monk asked..."!



1-7

## Giải thích lặp đệ quy

```
Void tellStory(int i){  
    if (i==0) { printf("nothing!\n"); return;}  
    printf("Once upon a time there was a mountain. ");  
    printf("On the mountain, there was a temple. ");  
    printf("In the temple was an monk, telling a story. ");  
    printf("What is the story? It is: ");  
    tellStory(i-1);  
    printf("The monk asked: do you like it?\n");  
}
```

Nếu gọi tellStory(5),  
thì câu chuyện kể bao nhiêu lần?



1-8

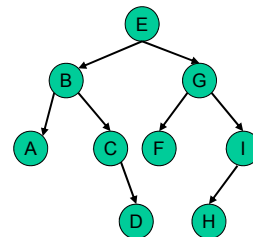
## Giải thích lặp đệ quy

```
Void tellStory(int i){  
    if (i==0) { printf("nothing!\n"); return;}  
    printf("Once upon a time there was a mountain. ");  
    printf("On the mountain, there was a temple. ");  
    printf("In the temple was an monk %d, telling a story. ", i);  
    printf("What is the story? It is: ");  
    tellStory(i-1);  
    printf("The monk %d asked: do you like it?\n", i);  
}  
  
tellStory(5);
```



## Ba cách cơ bản để duyệt cây

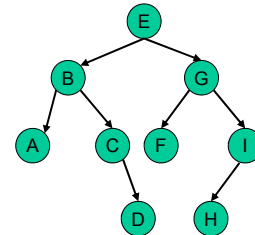
- ❑ Thứ tự trước (pre-order)
  - Xử lý dữ liệu của nút hiện tại
  - Thăm nút con trái trên cây con
  - Thăm nút con phải trên cây con
- ❑ Thứ tự giữa (in-order)
  - Thăm nút con trái trên cây con
  - Xử lý dữ liệu của nút hiện tại
  - Thăm nút con phải trên cây con
- ❑ Thứ tự sau (post-order)
  - Thăm nút con trái trên cây con
  - Thăm nút con phải trên cây con
  - Xử lý dữ liệu của nút hiện tại



1-10

## Duyệt cây theo thứ tự trước

```
void TreeTraversal(BTNode *cur) {  
    if (cur == NULL)  
        return;  
  
    printf("%c", cur->item);  
  
    TreeTraversal(cur->left); //Thăm nút con trái  
    TreeTraversal(cur->right); //Thăm nút con phải  
}
```

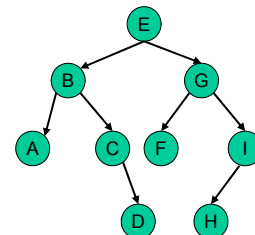


Kết quả:  
E B A C D G F I H

1-11

## Duyệt cây theo thứ tự giữa

```
void TreeTraversal(BTNode *cur) {  
    if (cur == NULL)  
        return;  
  
    TreeTraversal(cur->left); //Thăm nút con trái  
    printf("%c", cur->item);  
    TreeTraversal(cur->right); //Thăm nút con phải  
}
```

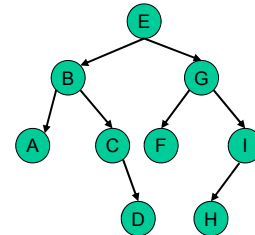


Kết quả:  
A B C D F G H I

1-12

## Duyệt cây theo thứ tự sau

```
void TreeTraversal(BTNode *cur) {  
    if (cur == NULL)  
        return;  
  
    TreeTraversal(cur->left); //Thăm nút con trái  
    TreeTraversal(cur->right); //Thăm nút con phải  
    printf("%c", cur->item);  
}
```



Kết quả:  
A D C B F H I E

1-13

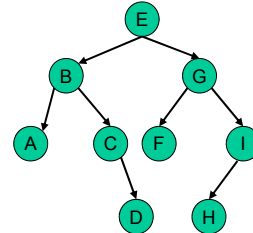
## Duyệt cây nhị phân

- Thứ tự duyệt cây
  - Thứ tự trước
  - Thứ tự giữa
  - Thứ tự sau
- Ví dụ ứng dụng
  - Đếm số nút trên cây nhị phân
  - Tìm nút cháu
  - Tính chiều cao của nút

1-14

## Đếm số nút trên cây nhị phân

- Định nghĩa đệ quy:
  - Số nút trên cây = 1 + số nút trên cây con trái + số nút trên cây con phải
- Mỗi nút trả về số nút trên cây con của nó
- Nút lá trả về 1



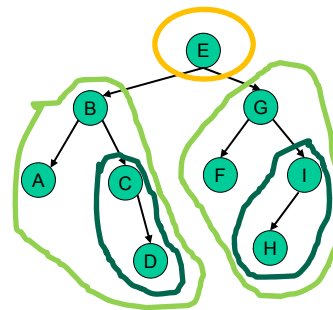
1-15

## countNode()

- Trả về kích thước của cây con của nó cho nút cha
- Nút lá trả về 1 cho nút cha
- Nút gốc trả về kích thước của toàn bộ cây

```
void TreeTraversal(BTNode *cur) {  
    if (cur == NULL)  
        return;  
    // Có thể thực hiện một số thao tác  
    TreeTraversal(cur->left); //Thăm nút con trái  
    // Có thể thực hiện một số thao tác  
    TreeTraversal(cur->right); //Thăm nút con phải  
    // Có thể thực hiện một số thao tác  
}
```

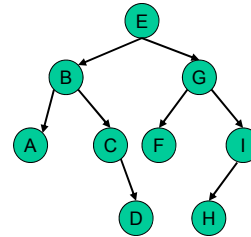
```
int countNode(BTNode*cur) {  
    if (cur == NULL)  
        return ???;  
    countNode(cur->left);  
    countNode(cur->right);  
    ??? //tính tổng  
}
```



1-16

## countNode()

- ❑ Nút lá trả về 1 cho nút cha
- ❑ Nút Null trả về 0

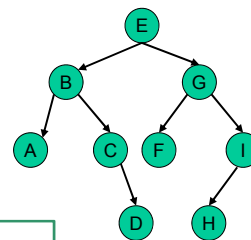


```
int countNode(BTNode*cur) {  
    if (cur == NULL)  
        return 0;  
    int l = countNode(cur->left);  
    int r = countNode(cur->right);  
    return l+r+1; //tính tổng  
}
```

1-17

## countNode()

- ❑ Nút lá trả về 1 cho nút cha
- ❑ Nút Null trả về 0



```
int countNode(BTNode*cur) {  
    if (cur == NULL)  
        return 0;  
    return (countNode(cur->left) + countNode(cur->right) + 1);  
}
```

1-18



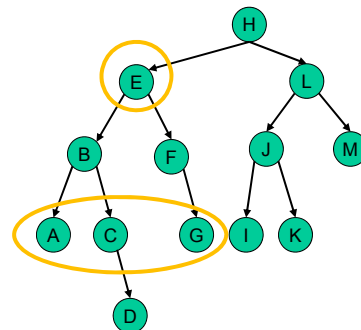
## Duyệt cây nhị phân

- Thứ tự duyệt cây
  - Thứ tự trước
  - Thứ tự giữa
  - Thứ tự sau
- **Ví dụ ứng dụng**
  - Đếm số nút trên cây nhị phân
  - **Tìm nút cháu**
  - Tính chiều cao của nút

1-19

## Tìm nút cháu

- Cho nút X, tìm tất các nút cháu của X
- Ví dụ: Cho nút E thì sẽ trả về các nút cháu là A, C và G
- Tìm các nút cháu trong **mức k**
  - Cần lưu vết về số mức đã đi qua



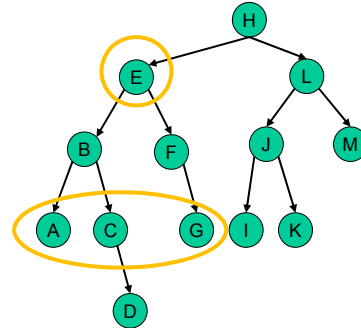
- Các nút cháu ở mức 2:
  - X->left->left
  - X->left->right
  - X->right->left
  - X->right->right

1-20

## Tìm nút cháu

- Muốn đi xuống mức k: Sử dụng biến đếm counter để xác định độ sâu

```
void TreeTraversal(BTNode *cur) {
    if (cur == NULL)
        return;
    // Kiểm tra biến đếm counter
    TreeTraversal (cur->left);
    TreeTraversal(cur->right);
}
```



1-21

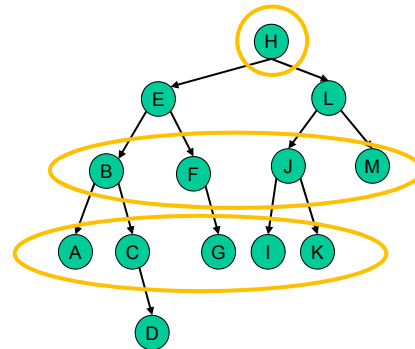
## Tìm nút cháu

```
void main() { ...
    if ( X == null) return;
    findgrandchildren(X, 0);
}
```

```
void findgrandchildren(BTNode *cur, int c) {
    if (cur == NULL) return;

    if (c == k) {
        printf("%c ", cur->item);
        return;
    }

    findgrandchildren(cur->left, c+1);
    findgrandchildren(cur->right, c+1);
}
```



- Nếu k = 2, gọi findgrandchildren(H,0), kết quả là gì?
- Nếu k = 3, gọi findgrandchildren(H,0), kết quả là gì?
- Nếu k = 2, gọi findgrandchildren(H,1), kết quả là gì?

1-22

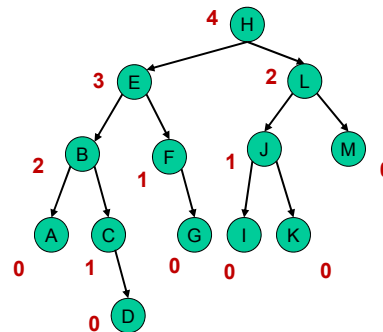
## Duyệt cây nhị phân

- ❑ Thứ tự duyệt cây
  - Thứ tự trước
  - Thứ tự giữa
  - Thứ tự sau
- ❑ **Ví dụ ứng dụng**
  - Đếm số nút trên cây nhị phân
  - Tìm nút cháu
  - **Tính chiều cao của nút**

1-23

## Tính chiều cao của một nút

- ❑ Chiều cao của một nút = số liên kết từ nút đó tới nút lá sâu nhất
- ❑ Chiều cao của nút D, C, H?
- ❑ Cách tính chiều cao của một nút?



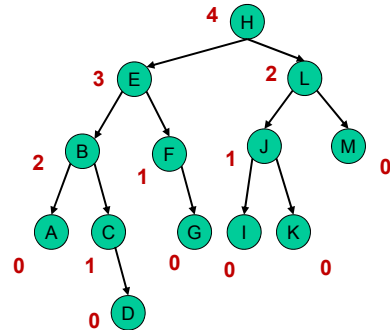
- ❑ leaf.height = 0
- ❑ Non-leaf node X:  $X.height = \max(X.left.height, X.right.height) + 1$

1-24

## Tính chiều cao của một nút

- ❑ Mỗi nút tính chiều cao của nó:
- ❑ Nút lá trả về chiều cao bằng 0
- ❑ Các nút NULL trả về -1

```
int TreeTraversal(BTNode *cur) {  
    if (cur == NULL)  
        return -1;  
  
    int l = TreeTraversal(cur->left);  
    int r = TreeTraversal(cur->right);  
  
    int c = max(l, r) + 1;  
  
    return c;  
}
```



1-25

## Cấu trúc dữ liệu và giải thuật

- ❑ Nội dung bài giảng được biên soạn bởi TS. Phạm Tuấn Minh.

1-26

# Cấu trúc dữ liệu và giải thuật

**TS. Phạm Tuấn Minh**

Khoa Công nghệ Thông tin, Đại học Phenikaa

[minh.phamtuan@phenikaa-uni.edu.vn](mailto:minh.phamtuan@phenikaa-uni.edu.vn)

<https://sites.google.com/site/phamtuanminh/>

## Chương 3: Cây và bảng băm

- ❑ Các khái niệm cây
- ❑ Cây tìm kiếm nhị phân
- ❑ Cây AVL
- ❑ Bảng băm

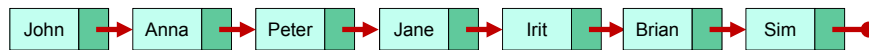
## Cây tìm kiếm nhị phân

- ❑ **Tìm kiếm một phần tử**
- ❑ Cây tìm kiếm nhị phân
- ❑ Thao tác trên cây tìm kiếm nhị phân
  - Duyệt
  - Chèn nút
  - Xóa nút

1-3

## Tìm kiếm phần tử trên danh sách liên kết

- ❑ Cho một danh sách liên kết các tên, kiểm tra xem một tên có trong danh sách không?



cur = cur->next;

```
while (cur!=NULL) {  
    if cur->item == "Irit"  
        found and stop searching;  
    else  
        cur = cur->next;  
}
```

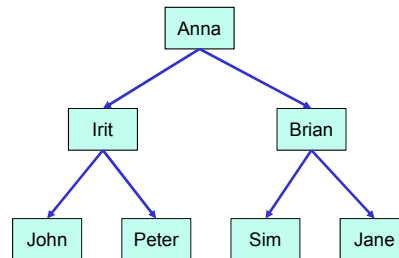
- ❑ Số nút phải duyệt qua khi thực hiện tìm kiếm:
  - Tốt nhất: 1 nút (John)
  - Tồi nhất: 7 nút (Sim)
  - Trung bình:  $(1+2+...+7)/7 = 4$  nút
- ❑ **Không hiệu quả!**

## Tìm kiếm phần tử trên cây nhị phân tìm kiếm

- ❑ Cho một cây nhị phân các tên, kiểm tra xem một tên có trong cây không?

Sử dụng `TreeTraversal(Pre-order)` để kiểm tra mọi nút

```
TreeTraversal(Node N)
  If N==NULL return;
  if N.item==given_name return;
  TreeTraversal(LeftChild);
  TreeTraversal(RightChild);
  Return;
```



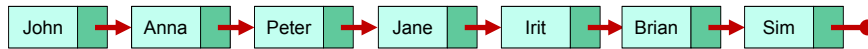
- ❑ Số nút phải duyệt qua khi thực hiện tìm kiếm:
  - Tốt nhất: 1 nút (Anna)
  - Tồi nhất: 7 nút (Jane)
  - Trung bình:  $(1+2+...+7)/7 = 4$  nút
- ❑ **Không hiệu quả!**

## Có cách tổ chức dữ liệu để tìm kiếm nhanh hơn?

- ❑ Có cách nào để sắp xếp dữ liệu trên cây để có thể lưu trữ và xác định phần tử hiệu quả?
- ❑ Làm thế nào để tìm nhanh một phần tử trên cây nhị phân?



## Có cách tổ chức dữ liệu để tìm kiếm nhanh hơn?

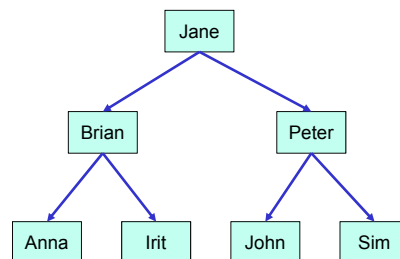


- ❑ Đối với danh sách: Sắp xếp theo thứ tự alphabet
- ❑ Chia thành nhóm
- ❑ Chọn một phần tử ở giữa, "Jane", nếu  $X \neq \text{"Jane"}$ 
  - Với tên trước "Jane", tìm bên nhóm trái, bỏ qua nhóm phải
    - Chọn một phần tử ở giữa trong nhóm con, "Brian", nếu  $X \neq \text{"Brian"}$ :
      - Với tên trước "Brian", tìm bên nhóm trái
      - Với tên sau "Brian", tìm bên nhóm phải
  - Với tên sau "Jane", tìm bên nhóm phải, bỏ qua nhóm trái
    - Chọn một phần tử ở giữa trong nhóm con, "Peter", nếu  $X \neq \text{"Peter"}$ :
      - Với tên trước "Peter", tìm bên nhóm trái
      - Với tên sau "Peter", tìm bên nhóm phải



## Có cách tổ chức dữ liệu để tìm kiếm nhanh hơn?

- ❑ Đối với danh sách: Sắp xếp theo thứ tự alphabet
- ❑ Chia thành nhóm
- ❑ Chọn một phần tử ở giữa, "Jane", nếu  $X \neq \text{"Jane"}$ 
  - Với tên trước "Jane", tìm bên nhóm trái, bỏ qua nhóm phải
    - Chọn một phần tử ở giữa trong nhóm con, "Brian", nếu  $X \neq \text{"Brian"}$ :
      - Với tên trước "Brian", tìm bên nhóm trái
      - Với tên sau "Brian", tìm bên nhóm phải
  - Với tên sau "Jane", tìm bên nhóm phải, bỏ qua nhóm trái
    - Chọn một phần tử ở giữa trong nhóm con, "Peter", nếu  $X \neq \text{"Peter"}$ :
      - Với tên trước "Peter", tìm bên nhóm trái
      - Với tên sau "Peter", tìm bên nhóm phải



- ❑ **Tạo thành một cây nhị phân tìm kiếm (BST - Binary Search Tree)**



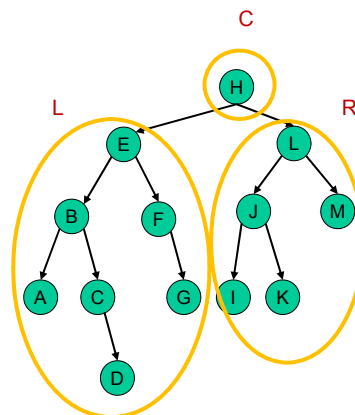
## Cây tìm kiếm nhị phân

- ❑ Tìm kiếm một phần tử
- ❑ **Cây tìm kiếm nhị phân**
- ❑ Thao tác trên cây tìm kiếm nhị phân
  - Duyệt
  - Chèn nút
  - Xóa nút

1-9

## Cây tìm kiếm nhị phân

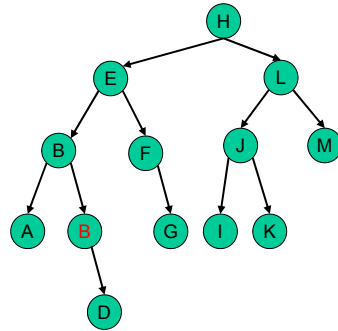
- ❑ Cây tìm kiếm nhị phân (BST - Binary Search Tree) là một dạng đặc biệt của cây nhị phân (BT)
- ❑ **Luật trên BST:** Tại mỗi nút **C**,  $L < C < R$ , trong đó
  - **C** là dữ liệu tại nút hiện tại
  - **L** là dữ liệu của bất kì nút nào từ cây con trái của C
  - **R** là dữ liệu của bất kì nút nào từ cây con phải của C



1-10

## Cây tìm kiếm nhị phân

- ❑ Cây tìm kiếm nhị phân (BST - Binary Search Tree) là một dạng đặc biệt của cây nhị phân (BT)
- ❑ **Luật trên BST:** Tại mỗi nút  $C$ ,  $L \leq C \leq R$ , trong đó
  - $C$  là dữ liệu tại nút hiện tại
  - $L$  là dữ liệu của bất kì nút nào từ cây con trái của  $C$
  - $R$  là dữ liệu của bất kì nút nào từ cây con phải của  $C$
- ❑ Không được phép có dấu bằng " $=$ " trên BST! Không được phép có nút lặp trên BST!

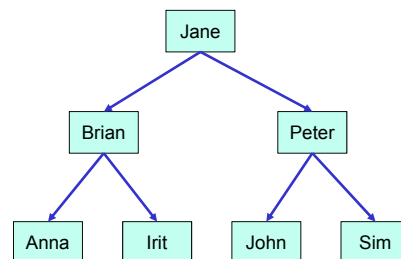


Đây không phải là BST

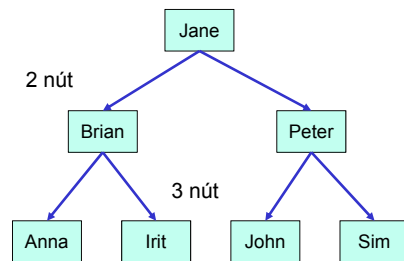
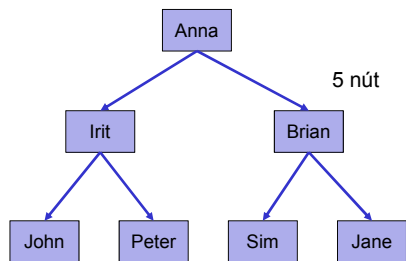
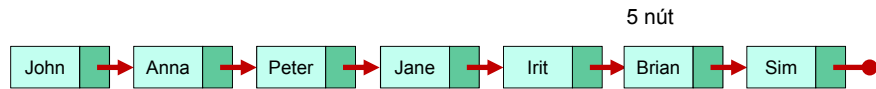
1-11

## Việc tìm kiếm phần tử sẽ hiệu quả với BST

- ❑ Đây là BST, thỏa mãn  $L < C < R$

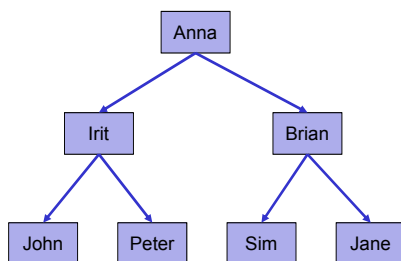


## Việc tìm kiếm phần tử sẽ hiệu quả với BST

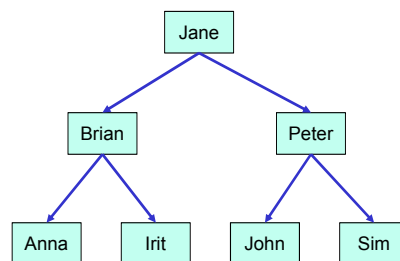


- ❑ Cần thăm bao nhiêu nút khi tìm kiếm?
  - Tốt nhất: 1 node (Anna)
  - Tồi nhất: **7** nút (Jane)
  - Trung bình:  $(1+2+\dots+7)/7 = 4$  nút
- ❑ Cần thăm bao nhiêu nút khi tìm kiếm?
  - Tốt nhất: 1 node (Jane)
  - Tồi nhất: **3** nút (Anna)
  - Trung bình:  $(1+2*2+3*4)/7 = 2,43$  nút

## Việc tìm kiếm phần tử sẽ hiệu quả với BST



Không hiệu quả



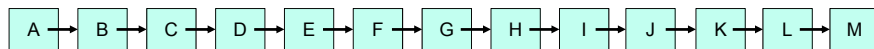
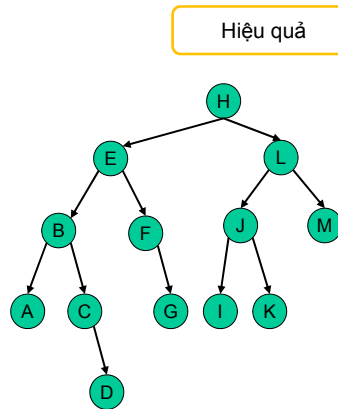
Hiệu quả

- ❑ Cần thăm bao nhiêu nút khi tìm kiếm?  
Tổng quát, với cây nhị phân n nút
  - Tốt nhất: Nút đầu tiên khi duyệt cây
  - Tồi nhất: **Nút cuối cùng** khi duyệt cây, **n**
- ❑ Cần thăm bao nhiêu nút khi tìm kiếm?  
Tổng quát với BST n nút
  - Tốt nhất: Nút đầu tiên khi duyệt cây
  - Tồi nhất: **nút lá, h = chiều cao của nút gốc. Giá trị nhỏ nhất h =  $\log_2(n+1) - 1$**

Khi n lớn, sự khác biệt về hiệu quả càng lớn

## Ví dụ tìm kiếm trên BST

- ❑ Để tìm D, cần thăm H-E-B-C-D: 5 nút
- ❑ Để tìm bất kì nút nào trên cây, cần thăm nhiều nhất 5 nút

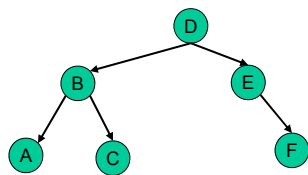


- ❑ Với danh sách liên kết, trường hợp tồi nhất là cần thăm tất cả các nút 13 nút

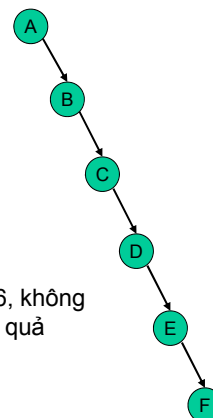
Không hiệu quả

1-15

## Chú ý: Việc tìm kiếm là hiệu quả không phải với tất cả BST



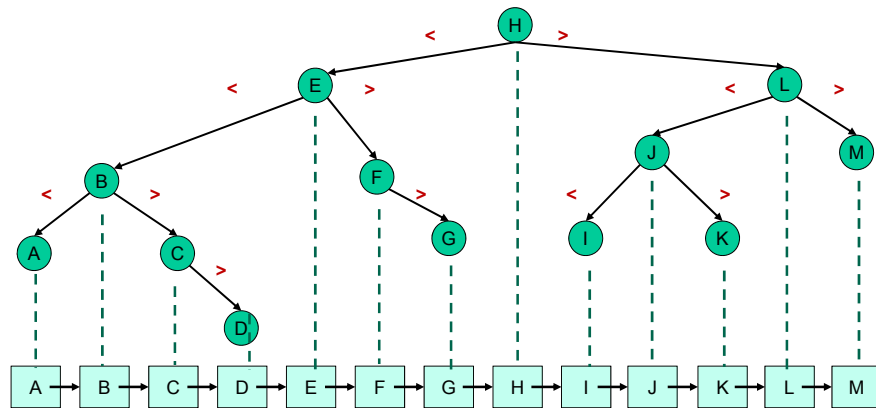
- ❑ Số nút tối đa cần kiểm tra:  $h = 3$



- ❑  $h = 6$ , không hiệu quả

1-16

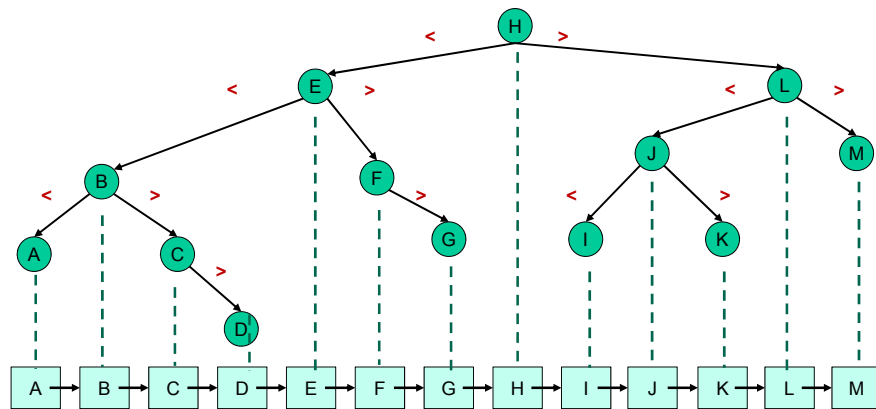
## Ảnh xạ: BST (thứ tự giữa) thành danh sách



- Nếu vẽ BST:
  - Cây con trái ở bên trái của nút hiện tại
  - Cây con phải ở bên phải của nút hiện tại
- Ảnh xạ xuống trực x sẽ thu được danh sách sắp thứ tự

1-17

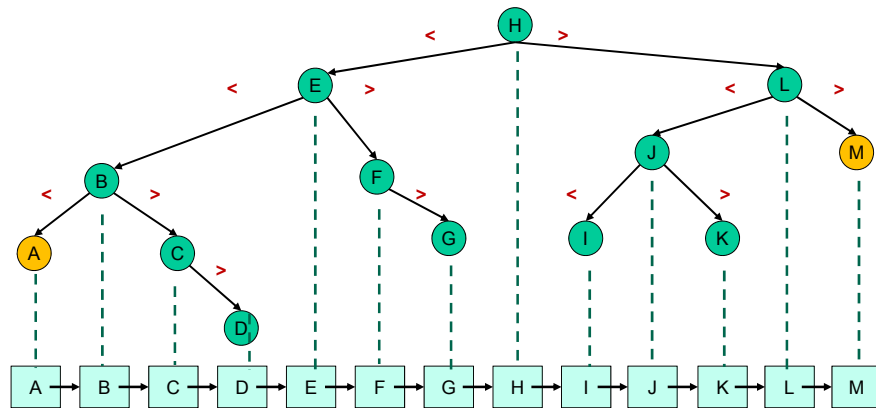
## BST và duyệt thứ tự giữa



- Luật  $L < C < R$  đảm bảo thứ tự sắp xếp
- Duyệt thứ tự giữa trên BST cho kết quả một danh sách được sắp thứ tự

1-18

## Đặc điểm của BST



- Cây tìm kiếm nhị phân đảm bảo rằng
  - Giá trị nhỏ nhất nằm ở nút trái nhất
  - Giá trị lớn nhất nằm ở nút phải nhất

1-19

## Cây tìm kiếm nhị phân

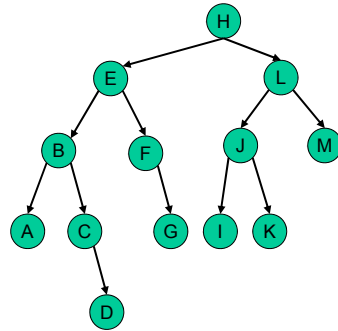
- Tìm kiếm một phần tử
- Cây tìm kiếm nhị phân
- **Thao tác trên cây tìm kiếm nhị phân**
  - **Duyệt**
  - Chèn nút
  - Xóa nút

1-20

## Duyệt BST

- ❑ Duyệt BST (BSTT - BST Traversal) thăm BST để tìm kiếm một nút có giá trị nào đó
- ❑ Bắt đầu với mẫu TreeTraversal

```
void BSTT(btnode *cur) {  
    if (cur == NULL)  
        return;  
    // printf(cur->item);  
    BSTT(cur->left);  
    BSTT(cur->right);  
}
```

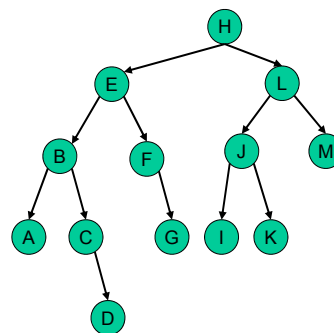


1-21

## Duyệt BST

- ❑ Duyệt BST (BSTT - BST Traversal) thăm BST để tìm kiếm một nút có giá trị nào đó
- ❑ Bắt đầu với mẫu TreeTraversal

```
void BSTT(btnode *cur, char c) {  
    if (cur == NULL)  
        return;  
    if (c==cur->item) { printf("found!\n"); return;}  
    if (c < cur->item)  
        BSTT(cur->left,c);  
    else  
        BSTT(cur->right,c);  
}
```

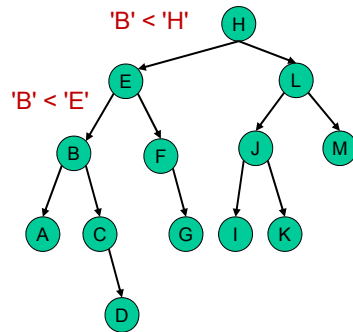


1-22

## Duyệt BST

□ BSTT(root, 'B')

```
void BSTT(btnode *cur, char c) {  
    if (cur == NULL)  
        return;  
    if (c==cur->item) { printf("found!\n"); return;}  
    if (c < cur->item)  
        BSTT(cur->left,c);  
    else  
        BSTT(cur->right,c);  
}
```

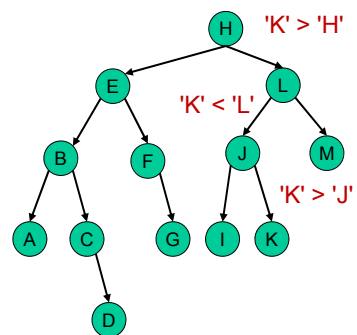


1-23

## Duyệt BST

□ BSTT(root, 'K')

```
void BSTT(btnode *cur, char c) {  
    if (cur == NULL)  
        return;  
    if (c==cur->item) { printf("found!\n"); return;}  
    if (c < cur->item)  
        BSTT(cur->left,c);  
    else  
        BSTT(cur->right,c);  
}
```



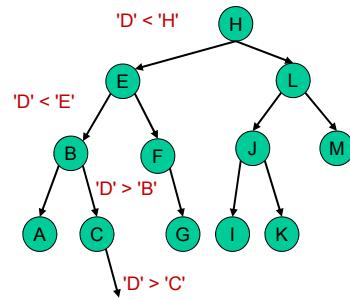
1-24



## Duyệt BST

- ❑ Nếu phần tử cần tìm không có trên cây thì sao?
- ❑ Ví dụ bỏ nút 'D', và gọi BSTT(root, 'D')

```
void BSTT(btnode *cur, char c) {  
    if (cur == NULL)  
        { printf("can't find!"); return; }  
    if (c==cur->item) { printf("found!\n"); return;}  
    if (c < cur->item)  
        BSTT(cur->left,c);  
    else  
        BSTT(cur->right,c);  
}
```



1-25

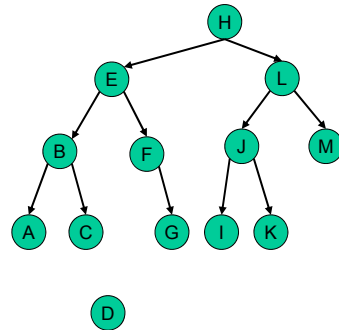
## Cây nhị phân tìm kiếm

- ❑ Tìm kiếm một phần tử
- ❑ Cây nhị phân tìm kiếm
- ❑ **Thao tác trên cây nhị phân tìm kiếm**
  - Duyệt
  - **Chèn nút**
  - Xóa nút

1-26

## Chèn một nút vào BST

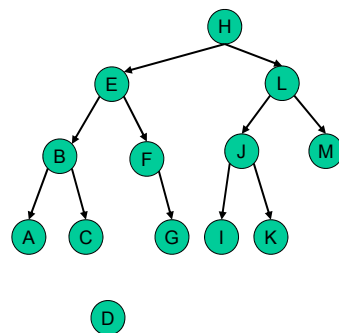
- ❑ Cho BST, thao tác chèn phải có kết quả là BST
- ❑ Làm sao biết vị trí đặt nút 'D'?



1-27

## Chèn một nút vào BST

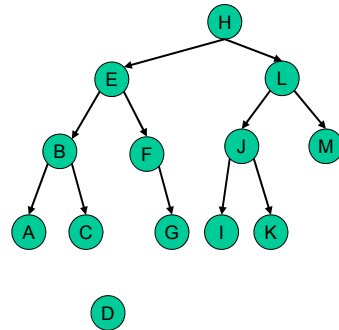
- ❑ Điểm then chốt: Cho BST và một giá trị cần chèn, có một vị trí duy nhất cho giá trị mới này trên BST



1-28

## Chèn một nút vào BST

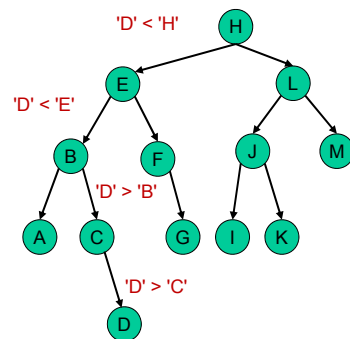
- ❑ Sử dụng BSTT() để tìm vị trí trống
- ❑ Chèn nút mới vào vị trí trống này



1-29

## Chèn một nút vào BST

- ❑ Sử dụng BSTT(root, 'D') để **xác định vị trí trống sẽ chèn 'D'**
- ❑ **Chèn nút mới 'D'**

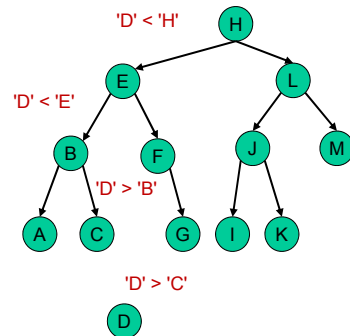


1-30

## Xác định vị trí sẽ chèn

- Sử dụng BSTT(root, 'D') để xác định vị trí trống sẽ chèn 'D'

```
BTNode* BSTT2(btNode *cur, char c) {  
    if (cur == NULL)  
        { printf("can't find!"); return; }  
    if (c==cur->item) { printf("found!\n"); return NULL;}  
    if (c < cur->item) {  
        if (cur->left == NULL)  
            return cur;  
        else BSTT2(cur->left,c);  
    } else {  
        if (cur->right == NULL)  
            return cur;  
        BSTT2(cur->right,c);  
    }  
}
```

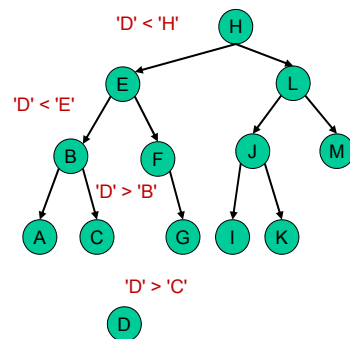


1-31

## Chèn nút mới

- Sử dụng BSTT(root, 'D') để xác định vị trí trống sẽ chèn 'D'
- **Chèn nút mới 'D'**

```
BTNode* posNode = BSTT2(&btNodeH,c);  
  
BTNode *btNewNode = malloc(sizeof(BTNode));  
btNewNode->item = c;  
btNewNode->left = NULL;  
btNewNode->right = NULL;  
  
if (posNode == NULL) {  
    printf("Phan tu da ton tai");  
    return 0;  
}  
if (c < posNode->item)  
    posNode->left = btNewNode;  
else posNode->right = btNewNode;
```



1-32

## Cây nhị phân tìm kiếm

- ❑ Tìm kiếm một phần tử
- ❑ Cây nhị phân tìm kiếm
- ❑ **Thao tác trên cây nhị phân tìm kiếm**
  - Duyệt
  - Chèn nút
  - **Xóa nút**

1-33

## Xóa nút

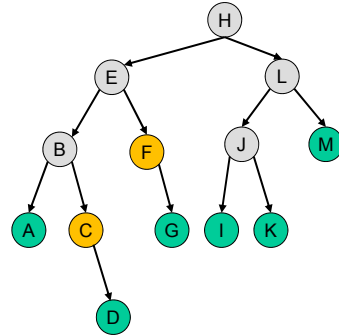
- ❑ Xóa nút phức tạp hơn chèn nút
- ❑ Kết quả BST sau khi xóa nút vẫn phải là BST
- ❑ Tuân theo luật:  $L < C < R$

1-34

## Xóa nút

❑ Xóa một nút x, có ba trường hợp

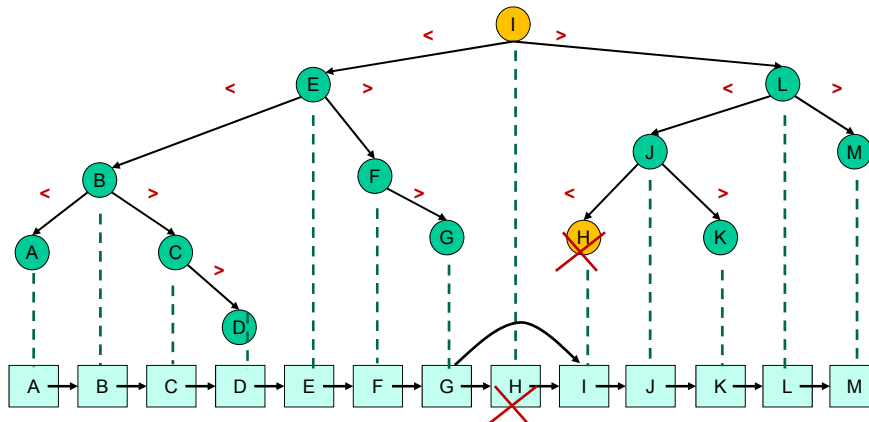
- 1) x không có nút con:
  - Xóa x
- 2) x chỉ có một con y:
  - Thay x bởi y
- 3) x có hai con:
  - Hoán đổi x với **nút sau** nó, rồi thực hiện 1) hoặc 2)



1-35

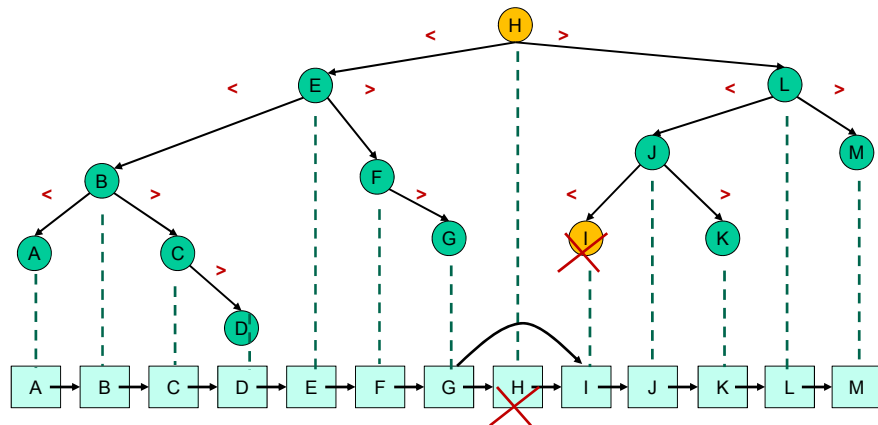
## Xóa nút

- ❑ Thay thế một nút bởi nút sau nó (duyệt theo thứ tự giữa) đảm bảo duy trì luật BST ( $L < C < R$ )
- ❑ Duyệt theo thứ tự giữa cho kết quả là một danh sách sắp xếp theo thứ tự tăng dần.
- ❑ **Nút sau** là nút ngay sau trong danh sách đã sắp xếp, hay là nút tiếp theo sẽ được thăm khi duyệt theo thứ tự giữa
- ❑ X có hai con, vì vậy nút sau của X là nút có giá trị nhỏ nhất trên cây con phải của X
- ❑ Ví dụ: Nút sau của H là I, nút sau của E là F, nút sau của J là K



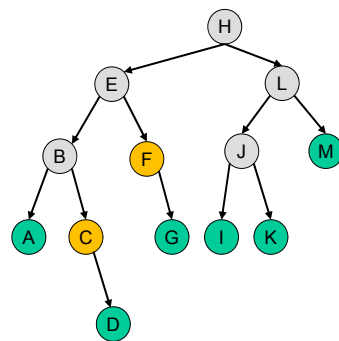
## Xóa nút

- Xóa nút H



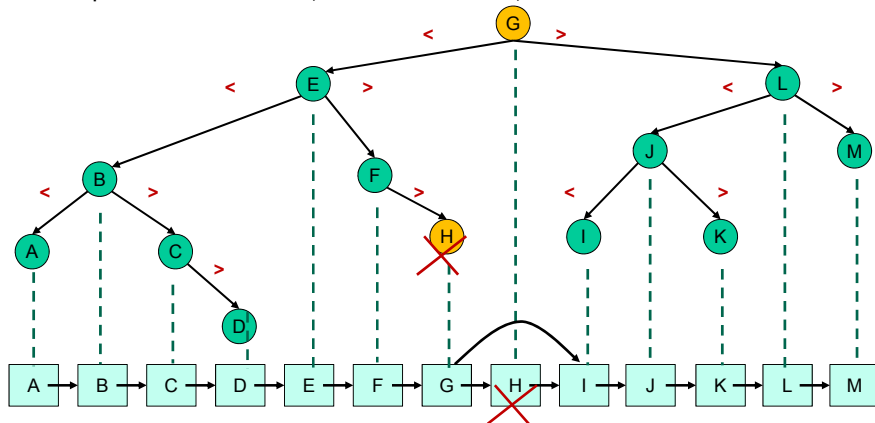
## Câu hỏi

- Tại sao trường hợp 3 luôn dẫn tới trường hợp 1 hoặc trường hợp 2
  - Khi x có 2 con, nút sau của nó là nút có giá trị nhỏ nhất trên cây con phải. Vì vậy, nút sau không có nút con trái => Nút sau có thể không có nút con (trường hợp 1) hoặc có một nút con phải (trường hợp 2)
- Có thể hoán đổi x với nút trước thay vì nút sau
  - Được

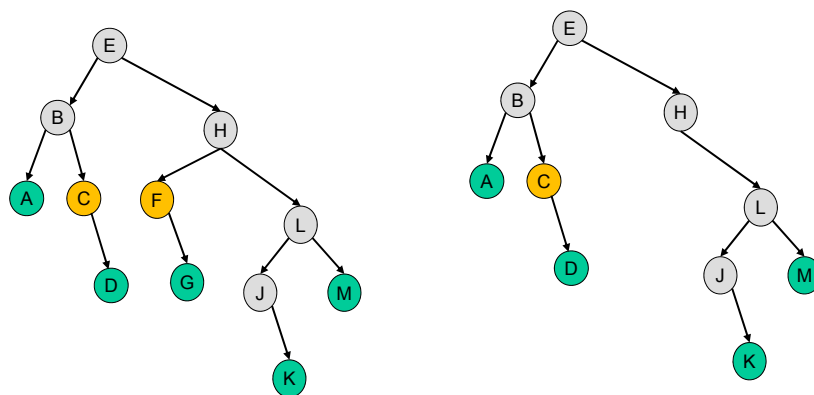


## Xóa nút

- Thay thế một nút bởi **nút trước** nó (duyệt theo thứ tự giữa) đảm bảo duy trì luật BST ( $L < C < R$ )
- Duyệt theo thứ tự giữa cho kết quả là một danh sách sắp xếp theo thứ tự tăng dần.
- Nút sau/nút trước** là nút ngay **sau/trước** trong danh sách đã sắp xếp, hay là **nút tiếp theo sẽ được thăm/nút ngay trước đã thăm** khi duyệt theo thứ tự giữa
- X có hai con, vì vậy nút trước của X là nút có giá trị lớn nhất trên cây con trái của X
- Ví dụ: Nút trước của H là G, nút trước của E là D, nút trước của J là I



## Xóa nút





## Tóm tắt

- ❑ Tìm kiếm một phần tử
- ❑ Cây nhị phân tìm kiếm
- ❑ **Thao tác trên cây nhị phân tìm kiếm**
  - Duyệt
  - Chèn nút
  - **Xóa nút**

1-41

## Cấu trúc dữ liệu và giải thuật

- ❑ Nội dung bài giảng được biên soạn bởi TS. Phạm Tuấn Minh.

1-42

# Cấu trúc dữ liệu và giải thuật

**TS. Phạm Tuấn Minh**

Khoa Công nghệ Thông tin, Đại học Phenikaa

[minh.phamtuan@phenikaa-uni.edu.vn](mailto:minh.phamtuan@phenikaa-uni.edu.vn)

<https://sites.google.com/site/phamtuanminh/>

## Chương 3: Cây và bảng băm

- ❑ Các khái niệm cây
- ❑ Cây nhị phân tìm kiếm
- ❑ Cây AVL
- ❑ Bảng băm

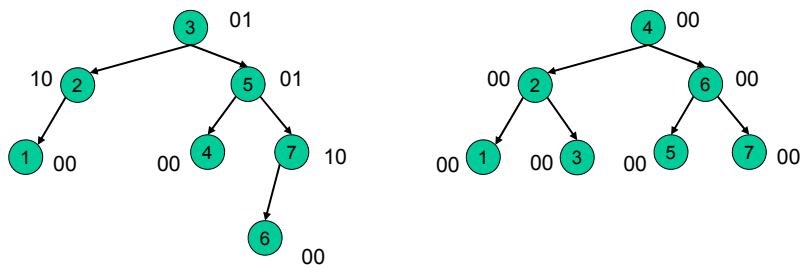
## Cây AVL

- ❑ **Khái niệm và đặc điểm cây AVL**
- ❑ Bổ sung trên cây AVL

1-3

## Khái niệm cây AVL

- ❑ Cây nhị phân tìm kiếm được gọi là cân đối AVL nếu như đối với mọi nút của nó chiều cao của hai cây con tương ứng chỉ chênh nhau một đơn vị



- ❑ Hệ số cân đối (balance factor):
  - 00 ứng với cây con trái và cây con phải có chiều cao bằng (cân bằng)
  - 01 ứng với cây con phải có chiều cao lớn hơn 1 (lệch phải)
  - 10 ứng với cây con trái có chiều cao lớn hơn 1 (lệch trái)

1-4

## Đặc điểm của cây AVL

- ❑ Tính cân đối của cây AVL có giảm nhẹ hơn so với tính cân đối của cây nhị phân hoàn chỉnh hoặc gần đầy: Cây nhị phân hoàn chỉnh hoặc gần đầy bao giờ cũng là cây cân đối AVL nhưng cây AVL thì chưa chắc đã là cây nhị phân hoàn chỉnh hoặc gần đầy
- ❑ Adelson - Velski và Landis đã chứng minh được rằng chiều cao của cây cân đối AVL chỉ vượt hơn khoảng 45% so với chiều cao của cây nhị phân hoàn chỉnh. Cụ thể là nếu gọi chiều cao của cây cân đối AVL có  $n$  nút là  $h_{AVL}(n)$  thì  $\log_2(n+1) < h_{AVL}(n) < 1,4404 \log_2(n+2) - 0,328$
- ❑ Nếu cây nhị phân tìm kiếm mà luôn luôn có dạng cân đối AVL, thì chi phí cho tìm kiếm đối với nó, ngay trong trường hợp xấu nhất vẫn là  $O(\log_2 n)$

1-5

## Cây AVL

- ❑ Khái niệm và đặc điểm cây AVL
- ❑ **Bổ sung trên cây AVL**

1-6

## Bổ sung trên cây AVL

- ❑ Thêm một trường BIT để ghi nhận hệ số cân đối của nút (chỉ cần 2 bit)
- ❑ Việc đi theo đường tìm kiếm trên cây để thấy được khoá mới chưa có sẵn trên cây và biết được "chỗ" để bổ sung nó vào, có một điều khác so với tìm kiếm trên cây nhị phân thông thường là đường đi này phải được ghi nhận lại để phục vụ cho việc xem xét và chỉnh lý hệ số cân đối của nút trên đường đi đó, bị tác động bởi phép bổ sung.

1-7

## Bổ sung trên cây AVL

- ❑ Giả sử phép bổ sung được thực hiện vào phía trái (phía phải thì cũng tương tự), sau khi nút mới được bổ sung, có ba tình huống có thể xảy ra với các nút tiền bối của nó:
  - Tình huống 1: Cây con phải đã cao hơn 1 (lệch phải) sau phép bổ sung chiều cao hai cây con bằng nhau
  - Tình huống 2: Chiều cao hai cây con vốn đã bằng nhau, sau phép bổ sung cây con trái cao hơn 1 (lệch trái)
  - Tình huống 3: Cây con trái đã cao hơn 1 (lệch trái), sau phép bổ sung nó cao hơn 2: tính "cân đối AVL" bị phá vỡ

1-8

## Bổ sung trên cây AVL: Tình huống 1

- Tình huống 1: Cây con phải đã cao hơn 1 (lệch phải) sau phép bổ sung chiều cao hai cây con bằng nhau
  - Chỉ cần chỉnh lý các hệ số cân đối ở nút đang xét

1-9

## Bổ sung trên cây AVL: Tình huống 2

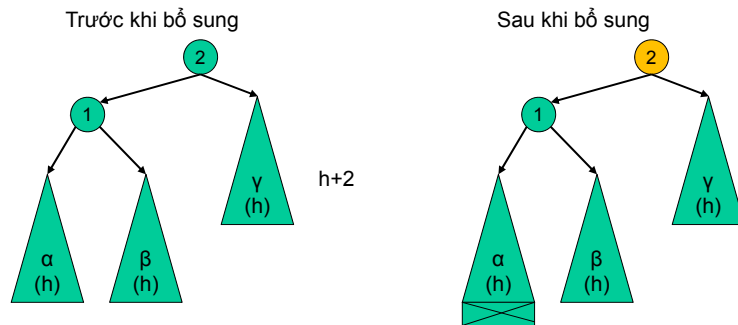
- Tình huống 2: Chiều cao hai cây con vốn đã bằng nhau, sau phép bổ sung cây con trái cao hơn 1 (lệch trái)
  - Chiều cao của cây có gốc là nút đang xét bị thay đổi, nên không những phải chỉnh lý hệ số cân đối ở nút đang xét mà còn phải chỉnh lý hệ số cân đối ở các nút tiền bối của nó.
  - Dọc trên đường đi đã ghi nhận khi tìm kiếm, cũng phải xem xét để biết có xảy ra tình huống nào trong ba tình huống đã nêu đối với nút đó không và có biện pháp xử lý thích hợp. Như vậy có khi cần phải lần ngược lại tới tận gốc cây.

1-10

## Bổ sung trên cây AVL: Tình huống 3

- Tình huống 3: Cây con trái đã cao hơn 1 (lệch trái), sau phép bổ sung nó cao hơn 2: tính "cân đối AVL" bị phá vỡ
  - Đòi hỏi phải sửa lại cây con mà nút đang xét là nút gốc (ta sẽ gọi là "nút bất thường" critical node), để nó cân đối lại.

Trường hợp 1 (LL): Nút mới bổ sung làm tăng chiều cao **cây con trái của nút con trái** nút bất thường.



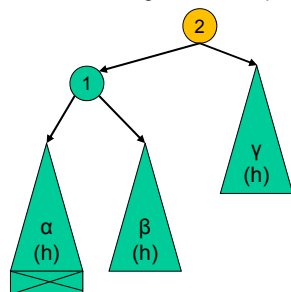
Nút mới bổ sung làm tăng chiều cao của cây  $\alpha$ , cây con trái của nút 1

1-11

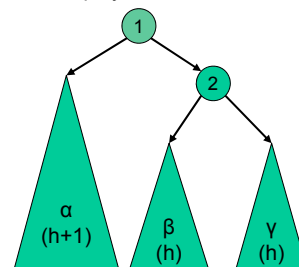
## Bổ sung trên cây AVL: Tình huống 3

- Trường hợp 1 (LL): Nút mới bổ sung làm tăng chiều cao **cây con trái của nút con trái** nút bất thường.
  - Để tái cân đối ta phải thực hiện một phép quay từ trái sang phải để đưa nút (1) lên vị trí gốc cây con, nút (2) sẽ trở thành con phải của nó, và p được gắn vào thành con trái của (2) (Gọi là phép quay đơn)

Sau khi bổ sung, trước khi quay



Sau khi quay

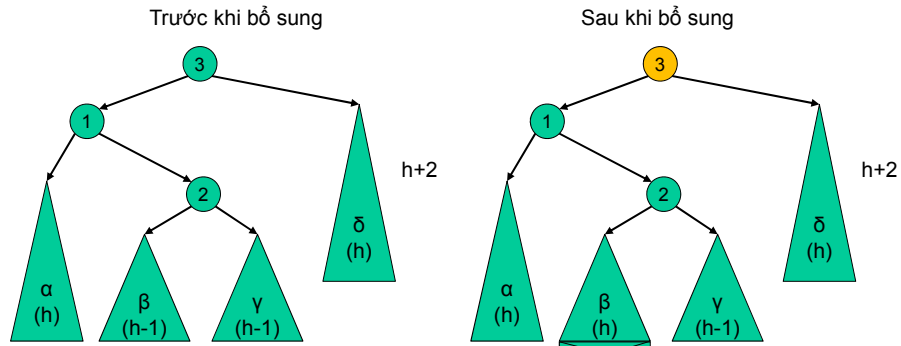


1-12

## Bổ sung trên cây AVL: Tình huống 3

- Tình huống 3: Cây con trái đã cao hơn 1 (lệch trái), sau phép bổ sung nó cao hơn 2: tính "cân đối AVL" bị phá vỡ
  - Đòi hỏi phải sửa lại cây con mà nút đang xét là nút gốc (ta sẽ gọi là "nút bất thường" critical node), để nó cân đối lại.

Trường hợp 2 (LR): Nút mới bổ sung làm tăng chiều cao **cây con phải của nút con trái** nút bất thường.



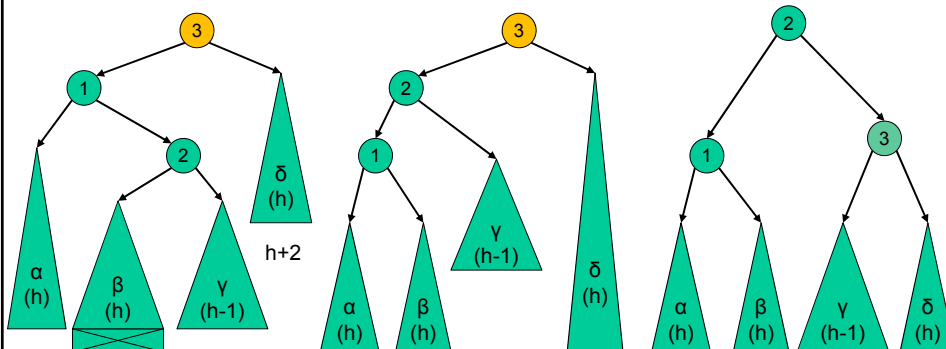
Nút mới bổ sung làm tăng chiều cao của cây  $\beta$ , cây con phải của nút 1

1-13

## Bổ sung trên cây AVL: Tình huống 3

- Trường hợp 2 (LR): Nút mới bổ sung làm tăng chiều cao cây con phải của nút con trái nút bất thường.
  - Để tái cân đối ta phải thực hiện một phép quay kép (double rotation), đó là việc phối hợp hai phép quay đơn: **quay trái đối với cây con trái** ((1), (2)) và **quay phải đối với cây** ((3), (2))

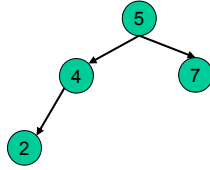
Sau khi bổ sung, trước khi quay      Sau phép quay đơn thứ nhất      Sau phép quay đơn thứ hai



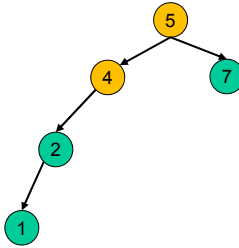
1-14



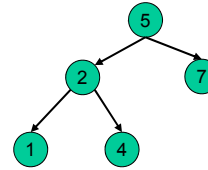
## Ví dụ



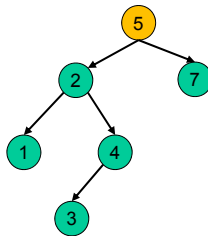
Cây ban đầu



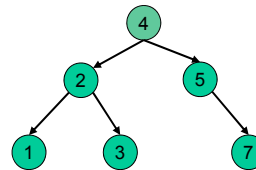
Bổ sung thêm nút (1): mất cân đối (trường hợp 1)



Tái cân đối bằng mất cân đối bằng phép quay đơn



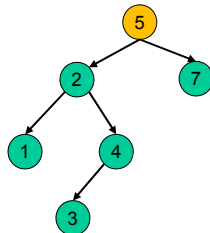
Bổ sung thêm nút (3): mất cân đối (trường hợp 2)



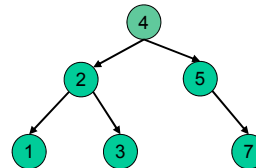
Tái cân đối bằng phép quay kép

1-15

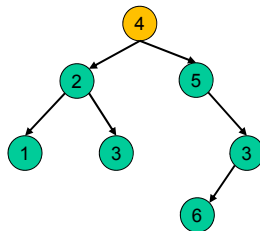
## Ví dụ



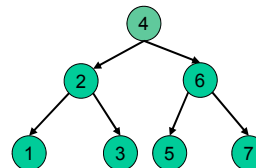
Bổ sung thêm nút (3): mất cân đối (trường hợp 2)



Tái cân đối bằng phép quay kép



Bổ sung thêm nút (6): mất cân đối (trường hợp 2)



Tái cân đối bằng phép quay kép

1-16

## Bổ sung trên cây AVL

### □ Nhận xét:

- Sau khi thực hiện phép quay để tái cân đối cây con mà nút gốc là nút bất thường thì chiều cao của cây con đó vẫn giữ nguyên như trước lúc bổ sung, nghĩa là phép quay không làm ảnh hưởng gì tới chiều cao các cây có liên quan tới cây con này.
- Ngoài ra tính chất của cây nhị phân tìm kiếm cũng luôn luôn được đảm bảo
- Phép loại bỏ cũng được thực hiện tương tự với chi phí không khác gì lắm so với phép bổ sung. Phép loại bỏ có thể cũng gây ra mất cân đối và phải tái cân đối bằng các phép quay như đã làm khi bổ sung

1-17

## Cấu trúc dữ liệu và giải thuật

- ### □ Nội dung bài giảng được biên soạn bởi TS. Phạm Tuấn Minh.

1-18

# Cấu trúc dữ liệu và giải thuật

**TS. Phạm Tuấn Minh**

Khoa Công nghệ Thông tin, Đại học Phenikaa

[minh.phamtuan@phenikaa-uni.edu.vn](mailto:minh.phamtuan@phenikaa-uni.edu.vn)

<https://sites.google.com/site/phamtuanminh/>

## Chương 3: Cây và bảng băm

- ❑ Các khái niệm cây
- ❑ Cây nhị phân tìm kiếm
- ❑ Cây AVL
- ❑ **Bảng băm**

## Ý tưởng

- ❑ Cấu trúc dữ liệu bảng băm là một mảng có kích thước cố định chứa các phần tử. Kích thước bảng là TableSize, bảng chạy từ 0 tới TableSize-1
- ❑ Tìm kiếm thực hiện trên một thành phần dữ liệu của phần tử, gọi là khóa (key/pivot)
- ❑ Hàm băm (Hash Function): Mỗi key được ánh xạ tới một số trong khoảng 0 tới TableSize-1 và đặt trong ô tương ứng

0	
1	
2	
3	john 25000
4	phil 31250
5	
6	dave 27500
7	
8	mary 28200
9	

1-3

## Hàm băm

- ❑ Nếu các khóa đầu vào là số nguyên, thì chỉ cần trả về  $\text{Key mod TableSize}$  thường là một cách hợp lý, trừ khi Key có đặc điểm đặc biệt.
- ❑ Ví dụ, nếu kích thước bảng là 10 và các khóa đều kết thúc bằng 0, thì đó là một lựa chọn tồi => Thường chọn kích thước bảng là số nguyên tố
- ❑ Khi key là số nguyên ngẫu nhiên, thì hàm không chỉ tính toán mà cần phân bố khóa đều
- ❑ Thông thường key là chuỗi, khi đó cần chọn hàm băm hợp lý
  - Ví dụ: Cộng các giá ASCII của các kí tự trong chuỗi

```
int hash( const string & key, int tableSize ) {  
    int hashVal = 0;  
    for( char ch : key )  
        hashVal += ch;  
    return hashVal % tableSize;  
}
```

1-4

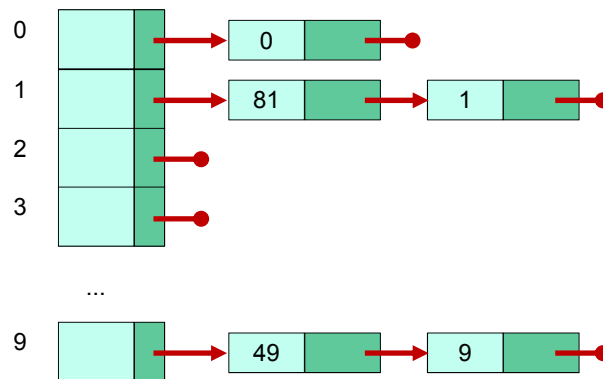
## Đụng độ

- ❑ Khi thêm một phần tử, nếu giá trị hàm băm của key trùng với giá trị ô đã có phần tử thì xảy ra đụng độ
- ❑ Một số phương pháp giải quyết đụng độ: separate chaining và open addressing

1-5

## Separate chaining

- ❑ Để chứa danh sách các phần tử có cùng giá trị hàm băm
- ❑ Tìm kiếm phần tử: Sử dụng hàm băm để xác định danh sách, sau đó tìm trên danh sách đó
- ❑ Chèn: Kiểm tra danh sách xem có phần tử trong đó không, nếu không có thể chèn vào đầu danh sách



1-6

## Open Addressing

- ❑ Separate chaining hashing có hạn chế là sử dụng danh sách liên kết -> chậm
- ❑ Open addressing:
  - Cố gắng tìm ô chưa sử dụng
  - Nếu tất cả các ô đều sử dụng, thì cần bảng lớn hơn
- ❑ Ví dụ:
  - $f$  là hàm giải quyết xung đột,  $h_i(x) = \text{hash}(x) + f(i) \bmod \text{TableSize}$
  - $f(i) = i$
  - Chèn key {89, 18, 49, 58, 69} vào bảng, đụng độ đầu tiên xảy ra khi chèn 49: Nó sẽ được chèn vào ô trống đầu tiên tiếp theo là ô 0

1-7

## Cấu trúc dữ liệu và giải thuật

- ❑ Nội dung bài giảng được biên soạn bởi TS. Phạm Tuấn Minh.

1-8

# Cấu trúc dữ liệu và giải thuật

**TS. Phạm Tuấn Minh**

Khoa Công nghệ Thông tin, Đại học Phenikaa

[minh.phamtuan@phenikaa-uni.edu.vn](mailto:minh.phamtuan@phenikaa-uni.edu.vn)

<https://sites.google.com/site/phamtuanminh/>

## Chương 4: Sắp xếp

- ❑ **Bài toán sắp xếp**
- ❑ Một số phương pháp sắp xếp cơ bản
- ❑ Sắp xếp QuickSort
- ❑ Sắp xếp HeapSort

## Bài toán sắp xếp

- ❑ Sắp xếp (Sorting) là quá trình bố trí lại các phần tử của một tập đối tượng nào đó, theo một thứ tự ấn định. Chẳng hạn thứ tự tăng dần (hay giảm dần) đối với một dãy số, thứ tự từ điển đối với một dãy chữ...
- ❑ Yêu cầu về sắp xếp thường xuyên xuất hiện trong các ứng dụng tin học, với những mục đích khác nhau: sắp xếp dữ liệu lưu trữ trong máy tính để tìm kiếm cho thuận lợi, sắp xếp các kết quả xử lý để in ra trên bảng biểu
- ❑ Trong chương này ta chỉ xét tới các phương pháp sắp xếp trong (internal sorting), nghĩa là các phương pháp tác động trên một tập các bản ghi lưu trữ đồng thời ở bộ nhớ trong

1-3

## Chương 4: Sắp xếp

- ❑ Bài toán sắp xếp
- ❑ **Một số phương pháp sắp xếp cơ bản**
- ❑ Sắp xếp QuickSort
- ❑ Sắp xếp HeapSort

1-4



## Sắp xếp kiểu lựa chọn

- Sắp xếp kiểu lựa chọn (selection sort)
  - Nguyên tắc cơ bản: Ở lượt thứ  $i$  ( $i = 1, 2, n$ ), chọn trong dãy khoá  $K(i), K(i+1), \dots, K(n)$  khoá nhỏ nhất và đổi chỗ nó với  $K(i)$
  - Sau  $j$  lượt,  $j$  khoá nhỏ hơn lần lượt ở vị trí thứ nhất, thứ hai, ... thứ  $j$  theo đúng thứ tự sắp xếp

1-5

## Ví dụ

i	K(i)	Lượt 1	2	3	4	...	9
1	42	11	11	11	11		11
2	23	23	23	23	23		23
3	74	74	74	36	36		36
4	11	42	42	42	42		42
5	65	65	65	65	65		58
6	58	58	58	58	58		65
7	94	94	94	94	94		74
8	36	36	36	74	74		87
9	99	99	99	99	99		94
10	87	87	87	87	87		99

1-6

## Select-sort(K, n)

```
for (i = 1; i < n; i++) {  
    m = i;  
    for (j = i+1; j < n; j++)  
        if (K[j] < K[m]) m = j;  
    if (m != i) {  
        X = K [i];  
        K [i] = K [m];  
        K[m] = x;  
    }  
}
```

1-7

## Sắp xếp kiểu thêm dần

### □ Sắp xếp kiểu thêm dần (insertion sort)

#### ○ Nguyên tắc cơ bản:

- Khi có  $i-1$  phần tử đã được sắp xếp, nay thêm phần tử thứ  $i$  nữa thì sắp xếp lại như thế nào?
- Có thể so sánh phần tử mới lần lượt với phần tử thứ  $(i-1)$ , thứ  $(i-2)$ ... để tìm ra "chỗ" thích hợp và "chèn" nó vào chỗ đó

1-8

## Ví dụ

Lượt	1	2	3	4	.	8	9	10
Khoá đưa vào	42	23	74	11	.	36	99	87
1	42	<b>23</b>	23	<b>11</b>	.	11	11	11
2	-	42	42	23	.	23	23	23
3	-	-	<b>74</b>	42	.	<b>36</b>	36	36
4	-	-	-	74	.	42	42	42
5	-	-	-	-	.	58	58	58
6	-	-	-	-	.	65	65	65
7	-	-	-	-	.	74	74	74
8	-	-	-	-	.	94	94	<b>87</b>
9	-	-	-	-	.	-	<b>99</b>	94
10	-	-	-	-	.	-	-	99

1-9

## Insert-sort(K, n)

```

K[0] = vô cùng bé
for (i = 2; i < n; i++) {
    X = K[i] ; j = i - 1;
    // Xác định chỗ cho khoá mới được xét và dịch chuyển các khoá cần thiết
    while (X < K[j]) {
        K[j+1] = K[j];
        j = j - 1;
    }

    //Đưa X vào đúng chỗ
    K[j + 1] = X;
}

```

1-10

## Sắp xếp kiểu đổi chỗ

### □ Sắp xếp kiểu đổi chỗ (exchange sort)

#### ○ Nguyên tắc cơ bản:

- Bảng các khoá sẽ được duyệt từ đáy lên đỉnh. Dọc đường, nếu gặp hai khoá kề cận ngược thứ tự thì đổi chỗ chúng cho nhau.
- Như vậy trong lượt đầu khoá có giá trị nhỏ nhất sẽ chuyển dần lên đỉnh. Đến lượt thứ hai khoá có giá trị nhỏ thứ hai sẽ được chuyển lên vị trí thứ hai...

#### ○ Phương pháp này còn gọi là sắp xếp kiểu nổi bọt (bubble sort)

1-11

## Ví dụ

i	K(i)	Lượt 1	2	3	4	...	9
1	42	11	11	11	11		11
2	23	42	23	23	23		23
3	74	23	42	36	36		36
4	11	74	36	42	42		42
5	65	36	74	58	58		58
6	58	65	58	74	65		65
7	94	58	65	65	74		74
8	36	94	87	87	87		87
9	99	87	94	94	94		94
10	87	99	99	99	99		99

1-12

## Bubble-sort(K, n)

```
for (i = 1; i < n; i++)  
  for (j = n; j > i; j--)  
    if (K[j] < K[j-1]) {  
      x = K[j];  
      K[j] = K[j-1];  
      K[j-1] = x;  
    }
```

1-13

## So sánh ba phương pháp

- ❑ Thời gian chủ yếu phụ thuộc vào việc thực hiện các phép so sánh giá trị khoá và các phép chuyển chỗ bản ghi, khi sắp xếp
- ❑ Với  $n$  khá lớn, chi phí về thời gian thực hiện được đánh giá qua cấp độ lớn, thì cả ba phương pháp đều có cấp  $O(n^2)$

1-14

## Chương 4: Sắp xếp

- ❑ Bài toán sắp xếp
- ❑ Một số phương pháp sắp xếp cơ bản
- ❑ **Sắp xếp QuickSort**
- ❑ Sắp xếp HeapSort

1-15

## Sắp xếp QuickSort

- ❑ Sắp xếp QuickSort
  - Chọn một khoá ngẫu nhiên nào đó của dãy làm "chốt" (pivot). Mọi phần tử nhỏ hơn khoá "chốt" phải được xếp vào vị trí ở trước "chốt" (đầu dãy), mọi phần tử lớn hơn khoá "chốt" phải được xếp vào vị trí sau "chốt" (cuối dãy).
    - Các phần tử trong dãy sẽ được so sánh với khoá chốt và sẽ đổi vị trí cho nhau, hoặc cho chốt, nếu nó lớn hơn chốt mà lại nằm trước chốt hoặc nhỏ hơn chốt mà lại nằm sau chốt.
    - Khi việc đổi chỗ đã thực hiện xong thì dãy khoá lúc đó được phân làm hai đoạn: một đoạn gồm các khoá nhỏ hơn chốt, một đoạn gồm các khoá lớn hơn chốt còn khoá chốt thì ở giữa hai đoạn nói trên, đó cũng là vị trí thực của nó trong dãy khi đã được sắp xếp, tới đây coi như kết thúc một lượt sắp xếp.
  - Ở các lượt tiếp theo cũng áp dụng một kỹ thuật tương tự đối với các phân đoạn còn lại

1-16

## QUICK-SORT(K, LB, UB)

- ❑ Quy ước chọn khóa "chốt" là khóa đầu tiên của dãy
- ❑ LB là chỉ số của phần tử đầu của dãy khoá đang xét (biên dưới)
- ❑ UB là chỉ số của phần tử cuối của dãy khoá đó (biên trên)
- ❑ Còn K là vector biểu diễn dãy khoá cho
- ❑ j là chỉ số ứng với khoá chốt sau khi đã tách dãy khoá đang xét thành 2 phân đoạn.
- ❑ PART(K, LB, UB, j): Thủ tục phân đoạn dãy khoá K thành một đoạn gồm các khoá nhỏ hơn chốt, một đoạn gồm các khoá lớn hơn chốt còn khoá chốt thì ở giữa hai đoạn nói trên

```
QUICK-SORT(K, LB, UB)
if (LB < UB) {
    call PART(K, LB, UB, j);
    call QUICK_SORT(K, LB, j - 1);
    call QUICK_SORT(K, j + 1, UB);
}
```

1-17

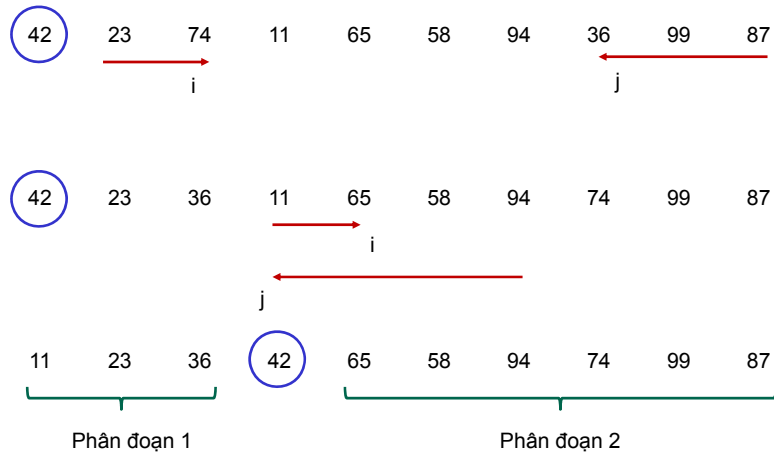
## Thủ tục phân đoạn

- ❑ Đưa thêm vào một giá trị khoá giả K[n+1], lớn hơn mọi giá trị khoá trong dãy khoá cho. Nó sẽ đóng vai trò như một lính gác (khoá gác biên) để khống chế biên trên, giúp cho việc xử lý được thuận lợi.
- ❑ Quá trình tìm số nhỏ hơn chốt để chuyển về phía trước chốt và khoá lớn hơn chốt để chuyển về phía sau chốt sẽ dựa vào hai biến chỉ số i và j để duyệt qua dãy khoá theo chiều ngược nhau.

```
PART(K, LB, UB, j)
i = LB + 1; j = UB;
while (i <= j) {
    while (K[i] < K[j]) i = i + 1;
    while (K[j] > K[j]) j = j - 1;
    if (i < j) {
        < Đổi chỗ K[i] <-> K[j] >
        i = i + 1;
        j = j - 1;
    }
}
if (K[LB] > K[j]) < Đổi chỗ K[LB] <-> K[j] >
```

1-18

## Ví dụ: Lướt phân đoạn đầu tiên



1-19

## Ví dụ: Kết quả sau từng lượt

	K(i)	42	23	74	11	65	58	94	36	99	87
i = 1		(11	23	36)	42	(65	58	94	74	99	87)
2		11	(23	36)	42	(65	58	94	74	99	87)
3		11	23	(36)	42	(65	58	94	74	99	87)
4		11	23	36	42	(58)	65	(94	74	99	87)
5		11	23	36	42	58	65	(94	74	99	87)
6		11	23	36	42	58	65	(87	74)	94	(99)
7		11	23	36	42	58	65	(74)	87	94	(99)
8		11	23	36	42	58	65	74	87	94	(99)
9		11	23	36	42	58	65	74	87	94	99

1-20



## Thảo luận

- ❑ Vấn đề chọn chốt
- ❑ Vấn đề phối hợp với cách sắp xếp khác
- ❑ Đánh giá giải thuật:  $O(n \log_2 n)$

1-21

## Thảo luận

- ❑ Gọi  $T(n)$  là thời gian thực hiện giải thuật ứng với một bảng  $n$  khoá,  $P(n)$  là thời gian để phân đoạn một bảng  $n$  khoá thành hai bảng con. Ta có thể viết:  $T(n) = P(n) + T(j - LB) + T(UB - j)$
- ❑ Chú ý rằng  $P(n) = C \cdot n$  với  $c$  là một hằng số.
- ❑ Trường hợp xấu nhất xảy ra khi bảng khoá vốn đã có thứ tự sắp xếp: sau khi phân đoạn một trong hai bảng con là rỗng ( $j = LB$  hoặc  $j = UB$ ).
- ❑ Giả sử  $j = LB$ , ta có:  
$$\begin{aligned} T_x(n) &= P(n) + T_x(0) + T_x(n-1) \quad (T_x(0) = 0) \\ &= C \cdot n + T_x(n-1) \\ &= C \cdot n + C \cdot (n-1) + T_x(n-2) \\ &\dots \\ &= \sum_{k=1..n} (C \cdot k + T_x(0)) \\ &= C \cdot n(n+1)/2 = O(n^2) \end{aligned}$$

1-22

## Thảo luận

- Trường hợp tốt nhất xảy ra khi mảng luôn luôn được chia đôi, nghĩa là  $j = (LB + UB) / 2$   
 $Tt = P(n) + 2 \cdot Tt(n/2)$   
 $= C \cdot n + 2 \cdot Tt(n/2)$   
 $= C \cdot n + 2 \cdot C \cdot (n/2) + 4 \cdot Tt(n/4) = 2 \cdot C \cdot n + 2^2 \cdot Tt(n/4)$   
 $= C \cdot n + 2 \cdot C \cdot (n/2) + 4 \cdot C \cdot (n/4) + 8 \cdot Tt(n/8) = 3 \cdot C \cdot n + 2^3 \cdot Tt(n/8)$   
...  
 $= (\log_2 n) \cdot C \cdot n + 2^{\log_2 n} \cdot Tt(1)$   
 $= O(n \log_2 n).$
- Giá trị thời gian tính trung bình, chứng minh được là  $O(n \log_2 n)$

1-23

## Chương 4: Sắp xếp

- Bài toán sắp xếp
- Một số phương pháp sắp xếp cơ bản
- Sắp xếp QuickSort
- **Sắp xếp HeapSort**

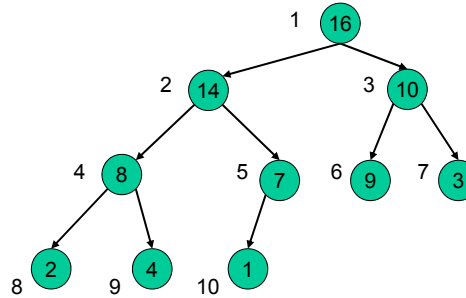
1-24

## Định nghĩa "đồng"

- Đồng là một cây nhị phân hoàn chỉnh mà mỗi nút được gán một giá trị khoá sao cho khoá ở nút cha bao giờ cũng lớn hơn khoá ở nút con nó.
- Đồng được lưu trữ trong máy bởi một vector  $K$  mà  $K[i]$  thì lưu trữ giá trị khoá ở nút thứ  $i$  trên cây nhị phân hoàn chỉnh, theo cách đánh số thứ tự khi lưu trữ kế tiếp.

- Lưu trữ kế tiếp:

- Đánh số các nút trên cây theo thứ tự lần lượt từ mức 1 trở đi, hết mức này đến mức khác và từ trái sang phải đối với các nút ở mỗi mức
- Quy luật: Con của nút thứ  $i$  là các nút thứ  $2i$  và  $2i+1$ , cha của nút  $j$  là nút  $\text{floor}(j/2)$



K

16	14	10	8	7	9	3	2	4	1
----	----	----	---	---	---	---	---	---	---

1-25

## Phép tạo đồng

- Xét một cây nhị phân hoàn chỉnh có  $n$  nút, mà mỗi nút đã được gán một giá trị khoá. Cây này chưa phải là đồng.
- Nhận xét:
  - Nếu một cây nhị phân hoàn chỉnh là đồng: các cây con của các nút (nếu có) cũng là cây nhị phân hoàn chỉnh và cũng là đồng.
  - Trên cây nhị phân hoàn chỉnh có  $n$  nút thì chỉ có  $\lfloor n/2 \rfloor$  nút được là "cha"
  - Một nút lá bao giờ cũng có thể coi là đồng
- Tạo đồng từ đáy lên: Hãy tạo thành đồng cho một cây nhị phân hoàn chỉnh có gốc được đánh số thứ tự là  $i$ , và gốc có 2 cây con đã là đồng rồi

1-26

## Phép tạo đồng

- Vector K với n phần tử được coi như vector lưu trữ của một cây nhị phân hoàn chỉnh có n nút
- Thủ tục tạo đồng ADJUST (i,n): Xét cây nút gốc là i

```
KEY = K[i]; //KEY nhận giá trị khoá ở nút gốc i
j = 2 * i; //j ghi nhận số thứ tự nút con trái của nút i
while (j <= n) {
    if ( (j < n) && (K[j] < K[j + 1]) ) j = j + 1;
    //Nếu khoá con phải lớn hơn thì j ghi nhận số thứ tự của nó

    if (KEY > K[j]) { // TH KEY lớn hơn khoá con
        K[ floor(j/2) ] = KEY; // thì xác định được vị trí của KEY -> dừng lại
        // KEY sẽ ở vị trí nút cha của khóa con
        return;
    }
    // TH KEY nhỏ hơn khóa con -> Đưa khoá con lên
    K[ floor(j/2) ] = K[j];

    j = 2 * j; // Tiếp tục đi xuống nhánh này để tìm vị trí cho KEY
    // vì KEY có thể nhỏ hơn nút ở nhánh này
```

1-27

## Bước tạo đồng

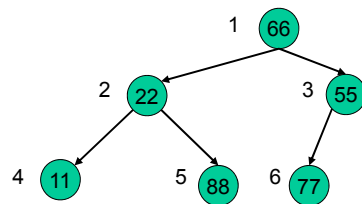
- Tạo thành đồng cho một cây nhị phân hoàn chỉnh có n nút: Thực hiện từ đáy lên

```
for (i = floor(n/2); i >= 1; i--) ADJUST(i,n);
```

1-28

## Ví dụ tạo đồng

- Dữ liệu và cấu trúc lưu trữ ban đầu



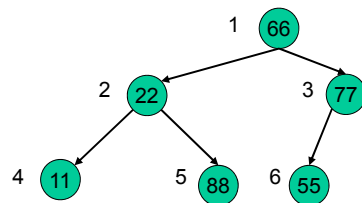
K

66	22	55	11	88	77
----	----	----	----	----	----

1-29

## Ví dụ tạo đồng

- Sau khi gọi ADJUST(3,6)



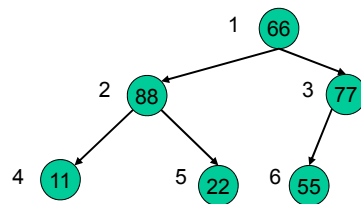
K

66	22	77	11	88	55
----	----	----	----	----	----

1-30

## Ví dụ tạo đồng

- Sau khi gọi ADJUST(2,6)



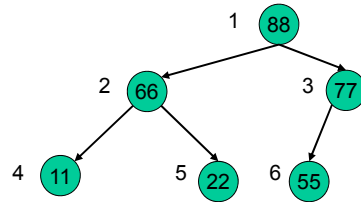
K

66	88	77	11	22	55
----	----	----	----	----	----

1-31

## Ví dụ tạo đồng

- Sau khi gọi ADJUST(1,6), cây đã là đồng, khóa lớn nhất ở đỉnh đồng



K

88	66	77	11	22	55
----	----	----	----	----	----

1-32

## Sắp xếp HeapSort

- Sắp xếp kiểu vun đống bao gồm 2 giai đoạn:
  - Giai đoạn tạo đống ban đầu
  - Giai đoạn sắp xếp được thực hiện (n-1) lần, bao gồm 2 bước:
    - Vun đống
    - Đổi chỗ
- Khoá lớn nhất sẽ được xếp vào cuối dãy, nghĩa là nó được đổi chỗ với khoá đang ở "đáy đống", và sau phép đổi chỗ này một khoá trong dãy đã vào đúng vị trí của nó trong sắp xếp.
- Nếu không kể tới khoá này thì phần còn lại của dãy khoá ứng với một cây nhị phân hoàn chỉnh, với số lượng khoá nhỏ hơn 1, sẽ không còn là đống nữa, ta lại gặp bài toán tạo đống mới cho cây này (gọi là "vun đống") và lại thực hiện tiếp phép đổi chỗ giữa khoá ở đỉnh đống và khoá ở đáy đống tương tự như đã làm.v.v...
- Cho tới khi cây chỉ còn là 1 nút thì các khoá đã được xếp vào đúng vị trí của nó trong sắp xếp.

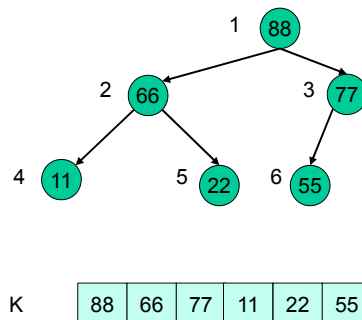
```
// Tạo đống
for (i = floor(n/2); i >= 1; i--)
    ADJUST(i,n);

// Sắp xếp
for (i = n - 1; i >= 1; i--) {
    < Đổi chỗ K[1] <-> K[i + 1] >
    ADJUST(1,i);
}
```

1-33

## Ví dụ sắp xếp HeapSort

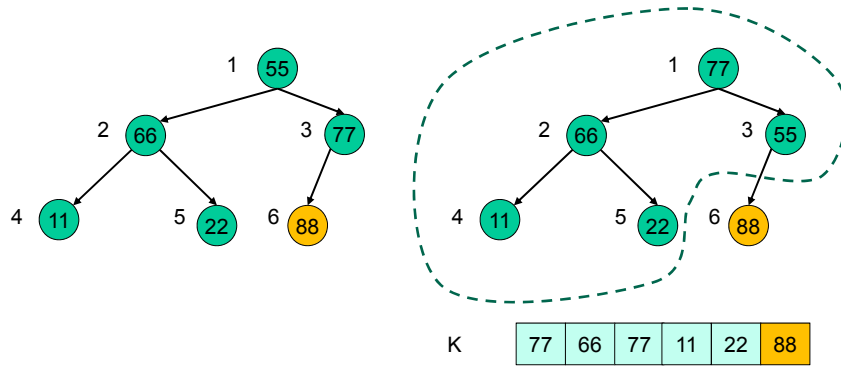
- Đống đã được tạo ra sau khi thực hiện giai đoạn 1



1-34

## Ví dụ sắp xếp HeapSort

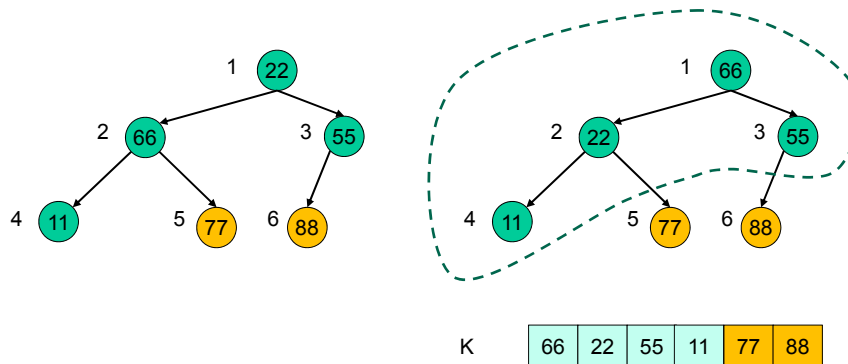
- Sau khi đổi chỗ và thực hiện ADJUST(1, 5)



1-35

## Ví dụ sắp xếp HeapSort

- Sau khi đổi chỗ và thực hiện ADJUST(1, 4)

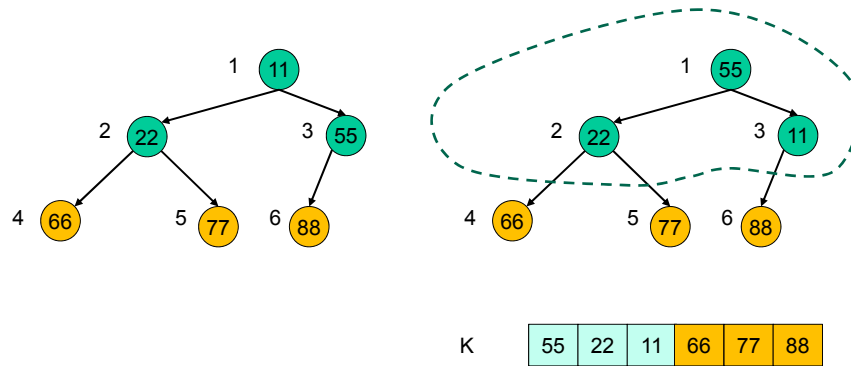


1-36



## Ví dụ sắp xếp HeapSort

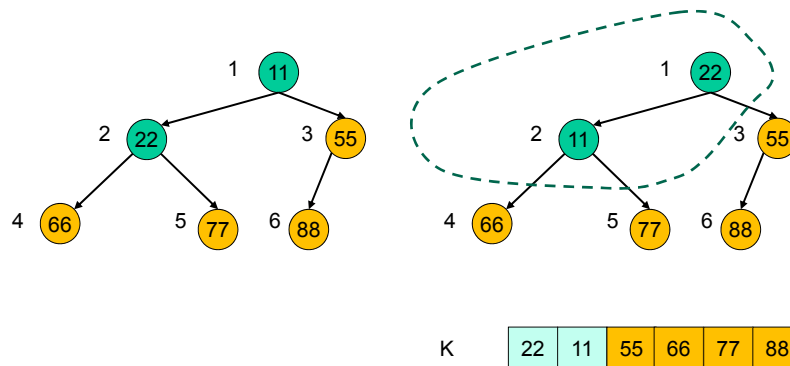
- Sau khi đổi chỗ và thực hiện ADJUST(1, 3)



1-37

## Ví dụ sắp xếp HeapSort

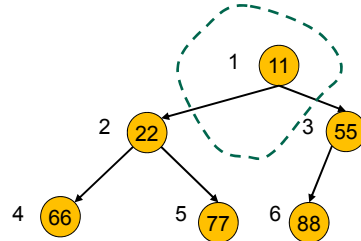
- Sau khi đổi chỗ và thực hiện ADJUST(1, 2)



1-38

## Ví dụ sắp xếp HeapSort

- Sau khi đổi chỗ và thực hiện ADJUST(1, 1), lúc này các khóa đã được sắp xếp



K

11	22	55	66	77	88
----	----	----	----	----	----

1-39

## Thảo luận

- Ở giai đoạn 1 (tạo đồng ban đầu) có  $\text{floor}(n/2)$  lần gọi thực hiện ADJUST(i,n).
- Ở giai đoạn 2 (sắp xếp) thì phải gọi thực hiện (n-1) lần ADJUST(1,i).
- Như vậy có thể coi như phải gọi khoảng  $3n/2$  lần thực hiện giải thuật ADJUST mà cây được xét ứng với ADJUST thì nhiều nhất là n nút, nghĩa là chiều cao của cây lớn nhất cũng chỉ xấp xỉ  $\log_2 n$ .
- Số lượng phép so sánh giá trị khoá, khi thực hiện giải thuật ADJUST, cùng lắm cũng chỉ bằng chiều cao của cây tương ứng.
- Như vậy, trường hợp tồi nhất số lượng phép so sánh cũng chỉ xấp xỉ  $(3/2)n \log_2 n$ .

1-40

## Thảo luận

- Thời gian thực hiện trung bình của HeapSort là  $O(n \log_2 n)$
- Thời gian thực hiện trường hợp xấu nhất  $O(n \log_2 n)$ , là ưu điểm so với QuickSort

1-41

## Cấu trúc dữ liệu và giải thuật

- Nội dung bài giảng được biên soạn bởi TS. Phạm Tuấn Minh.

1-42

# Cấu trúc dữ liệu và giải thuật

**TS. Phạm Tuấn Minh**

Khoa Công nghệ Thông tin, Đại học Phenikaa

[minh.phamtuan@phenikaa-uni.edu.vn](mailto:minh.phamtuan@phenikaa-uni.edu.vn)

<https://sites.google.com/site/phamtuanminh/>

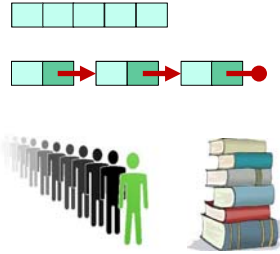
## Chương 5: Đồ thị

- ❑ **Khái niệm và biểu diễn đồ thị**
- ❑ Duyệt đồ thị
- ❑ Tìm đường đi
- ❑ Ứng dụng

## Cấu trúc dữ liệu đã học

### □ Tuyến tính:

- Tất cả các phần tử được sắp xếp có thứ tự
- Truy cập ngẫu nhiên
  - Mảng
- Truy cập tuần tự
  - Danh sách liên kết
- Truy cập tuần tự có hạn chế
  - Hàng đợi
  - Ngăn xếp

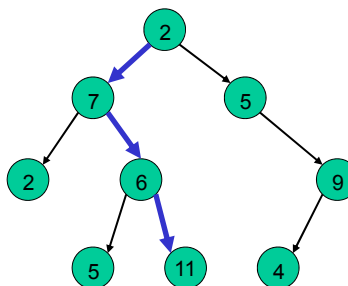


### □ Không tuyến tính

- Cây
- Đồ thị

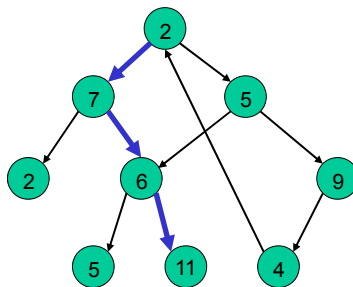
## Nhắc lại cấu trúc dữ liệu cây

- Vẫn sử dụng các biểu diễn nút và liên kết
- Đặc điểm mới:
  - Mỗi nút có liên kết tới nhiều hơn một nút khác
  - Không có lặp



## Cấu trúc dữ liệu đồ thị

- Vẫn sử dụng các biểu diễn nút và liên kết
- Đặc điểm mới:
  - Mỗi nút có liên kết tới nhiều hơn một nút khác
  - Cho phép có chu trình, các liên kết có thể kết nối giữa bất kì hai nút nào
  - Liên kết có thể có hướng (một chiều) hoặc vô hướng (hai chiều)



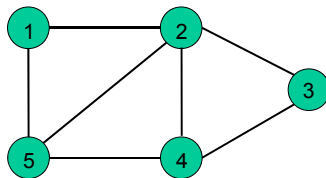
Cây là một trường hợp đặc biệt của đồ thị

## Khái niệm đồ thị

- Đồ thị là một cấu trúc dữ liệu bao gồm một tập nút (đỉnh - vertex) và tập cạnh (liên kết - link) mô tả quan hệ giữa các nút
- $G = (V, E)$ , trong đó  $V$  là tập nút,  $E$  là tập các cặp có thứ tự các phần tử của  $V$

## Thuật ngữ trong đồ thị

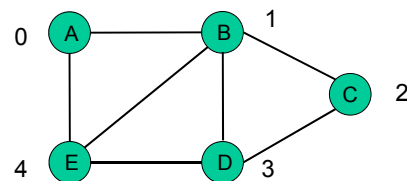
- ❑ Nút kề (Adjacent node): Hai nút là kề nhau nếu chúng nối với nhau bởi một cạnh
- ❑ Bậc của nút  $i$  (Degree): Số nút kề với nút  $i$
- ❑ Đường đi (Path): Chuỗi tiếp nối các cạnh nối hai nút trên đồ thị
- ❑ Đồ thị liên thông (Connected graph): Đồ thị trong đó có đường đi giữa hai nút bất kì trên đồ thị
- ❑ Đồ thị đầy đủ: Đồ thị mà mọi đỉnh có cạnh nối tới mọi đỉnh khác



- ❑ Nút 1 kề với nút 2 và 5
- ❑ Bậc của nút 1 là 2
- ❑ Một đường đi từ nút 1 tới nút 4 là: 1->2->3->4
- ❑ Đây là đồ thị liên thông
- ❑ Đây không phải là đồ thị đầy đủ

## Biểu diễn đồ thị

- ❑ Biểu diễn bằng mảng:
  - Một mảng 1 chiều để biểu diễn đỉnh
  - Một mảng 2 chiều (ma trận kề - adjacency matrix) để biểu diễn cạnh



[0]	A
[1]	B
[2]	C
[3]	D
[4]	E

Chứa dữ liệu của đỉnh

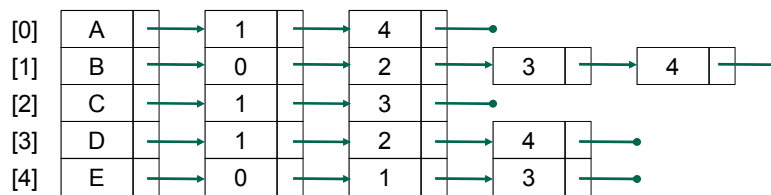
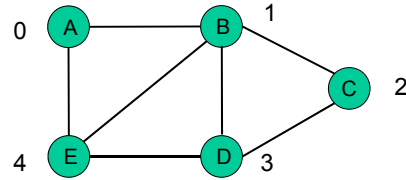
	[0]	[1]	[2]	[3]	[4]
[0]	0	1	0	0	1
[1]	1	0	1	1	1
[2]	0	1	0	1	0
[3]	0	1	1	0	1
[4]	1	1	0	1	0

Ma trận kề

## Biểu diễn đồ thị

### □ Biểu diễn bằng danh sách liên kết:

- Một mảng 1 chiều để biểu diễn đỉnh
- Một danh sách cho mỗi đỉnh  $v$ , danh sách chứa các đỉnh kề với đỉnh  $v$  (danh sách kề)



Chứa dữ liệu của đỉnh

Danh sách kề

## Ma trận kề và danh sách kề

### □ Ma trận kề

- Tốt với **đồ thị dày**:  $|E| \sim O(|V|^2)$
- Yêu cầu bộ nhớ:  $O(|V| + |E|) = O(|V|^2)$
- Có thể kiểm tra nhanh chóng **kết nối giữa hai đỉnh**

### □ Danh sách kề

- Tốt với **đồ thị thưa**:  $|E| \sim O(V)$
- Yêu cầu bộ nhớ:  $O(|V| + |E|) = O(|V|)$
- Có thể tìm nhanh chóng **các đỉnh kề với một đỉnh**



## Chương 5: Đồ thị

- Khái niệm và biểu diễn đồ thị
- **Duyệt đồ thị**
- Tìm đường đi
- Ứng dụng

1-11

## Duyệt đồ thị

- Duyệt danh sách: dễ
- Duyệt cây nhị phân: Thứ tự giữa, thứ tự trước, thứ tự sau
- Duyệt đồ thị?

1-12

## Duyệt theo chiều rộng (BFS)

- ❑ Duyệt theo chiều rộng (BFS - Breadth-first Search):
  - Xem tất cả các đường đi có thể với cùng một độ sâu trước khi đi tiếp với độ sâu lớn hơn
- ❑ Nếu đồ thị liên thông
  - Chọn một đỉnh xuất phát
  - Tìm mọi đỉnh kề
  - Lần lượt thăm từng đỉnh kề, sau đó quay lại thăm tất cả các đỉnh kề của nó

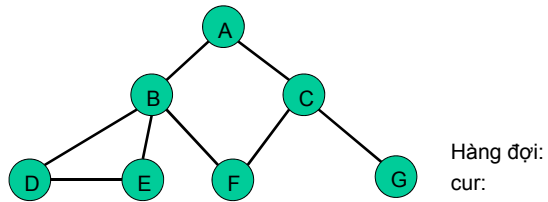
1-13

## breadthFirstSearch()

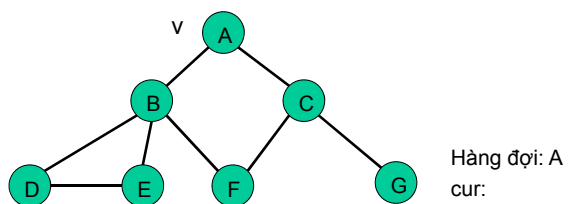
```
breadthFirstSearch(gNode cur, graph g) {  
    queue q;  
    initialize q;  
    mark cur as visited;  
    enqueue(&q, cur);  
    while (!emptyQueue(&q)) {  
        cur= dequeue(&q);  
        for all <đỉnh j chưa thăm và kề với cur> {  
            mark j as visited;  
            enqueue(&q, j);  
        }  
    }  
}
```

1-14

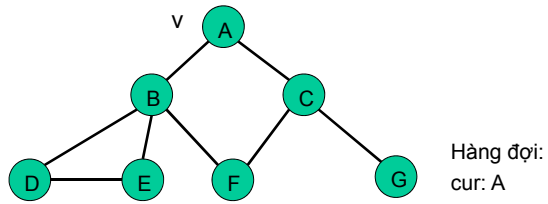
## Duyệt theo chiều rộng



## Duyệt theo chiều rộng

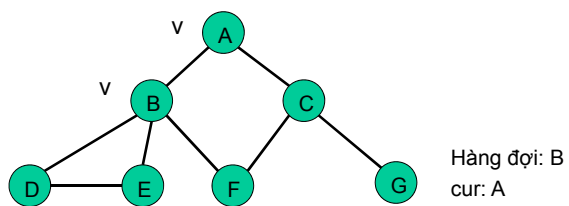


## Duyệt theo chiều rộng



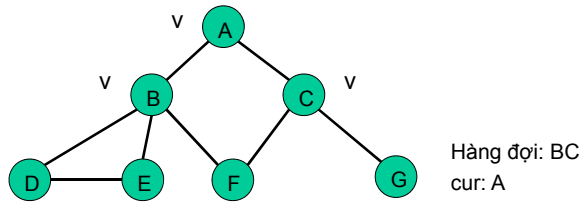
Kết quả duyệt: A

## Duyệt theo chiều rộng



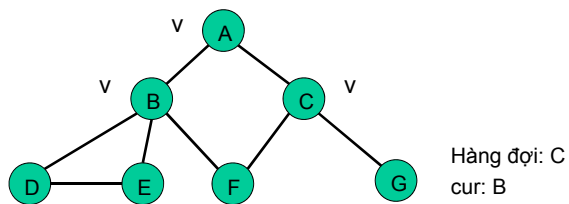
Kết quả duyệt: A

## Duyệt theo chiều rộng



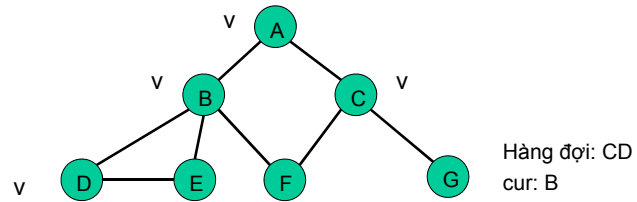
Kết quả duyệt: A

## Duyệt theo chiều rộng



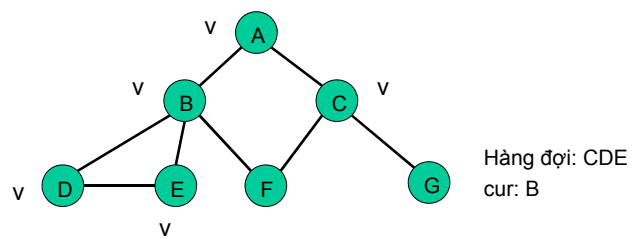
Kết quả duyệt: A B

## Duyệt theo chiều rộng



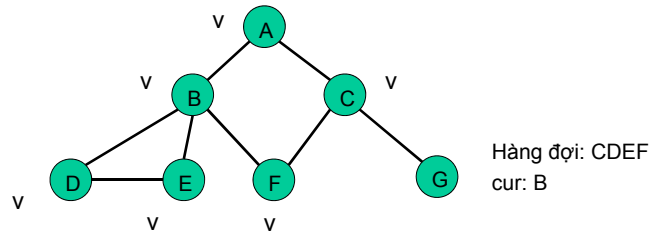
Kết quả duyệt: A B

## Duyệt theo chiều rộng



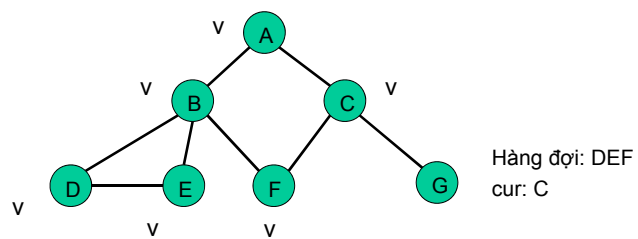
Kết quả duyệt: A B

## Duyệt theo chiều rộng



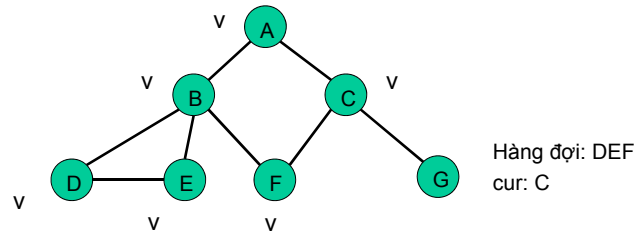
Kết quả duyệt: A B

## Duyệt theo chiều rộng



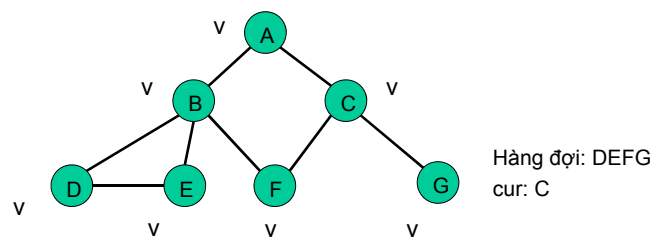
Kết quả duyệt: A B

## Duyệt theo chiều rộng



Kết quả duyệt: A B C

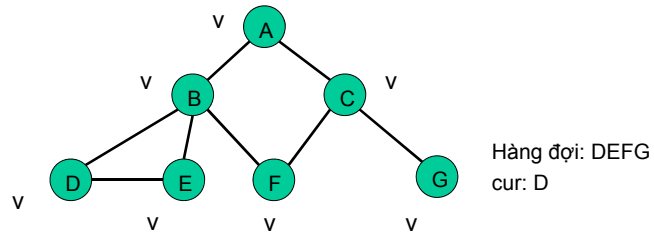
## Duyệt theo chiều rộng



Kết quả duyệt: A B C

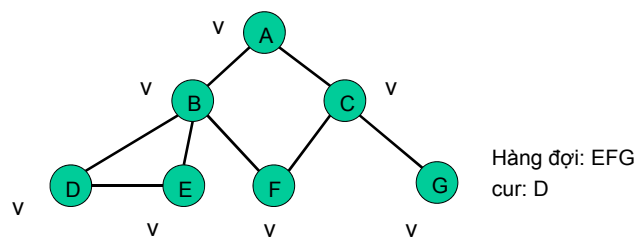


## Duyệt theo chiều rộng



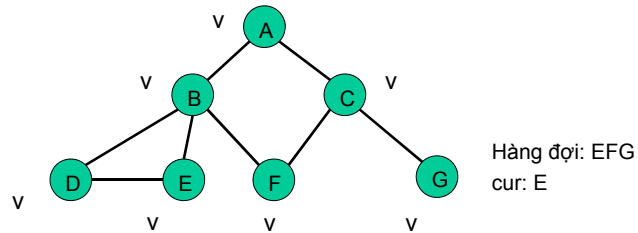
Kết quả duyệt: A B C

## Duyệt theo chiều rộng



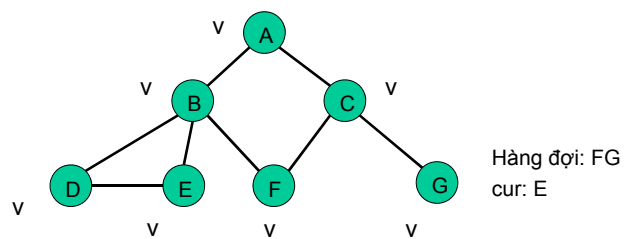
Kết quả duyệt: A B C D

## Duyệt theo chiều rộng



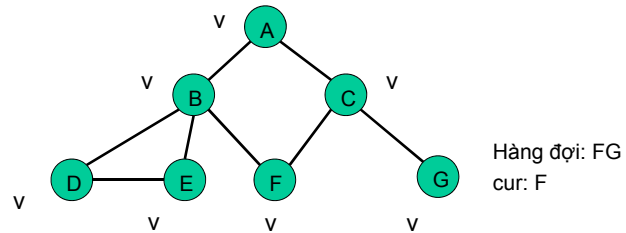
Kết quả duyệt: A B C D

## Duyệt theo chiều rộng



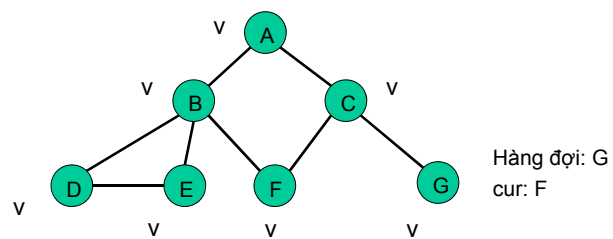
Kết quả duyệt: A B C D E

## Duyệt theo chiều rộng



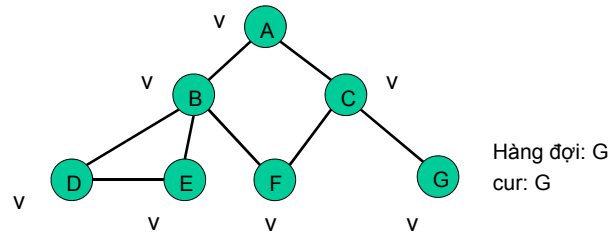
Kết quả duyệt: A B C D E

## Duyệt theo chiều rộng



Kết quả duyệt: A B C D E F

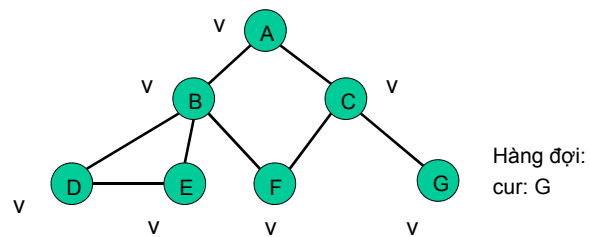
## Duyệt theo chiều rộng



Hàng đợi: G  
cur: G

Kết quả duyệt: A B C D E F G

## Duyệt theo chiều rộng



Hàng đợi:  
cur: G

Kết quả duyệt: A B C D E F G

## Duyệt theo chiều sâu (DFS)

- ❑ Duyệt theo chiều sâu (DFS - Depth-first Search):
  - Thăm nút sâu hơn trên đồ thị khi còn có thể
- ❑ Nếu đồ thị liên thông
  - Chọn một đỉnh xuất phát
  - Theo cạnh của đỉnh mới thăm gần nhất v mà vẫn còn cạnh chưa thăm
  - Sau khi thăm hết mọi cạnh của v, quay ngược lại theo cạnh mà từ đó v được thăm

1-35

## depthFirstSearch()

```
depthFirstSearch(v) {  
    Đánh dấu v là đã thăm;  
    for w chưa thăm và kề với v {  
        depthFirstSearch(w)  
    }  
}
```

1-36

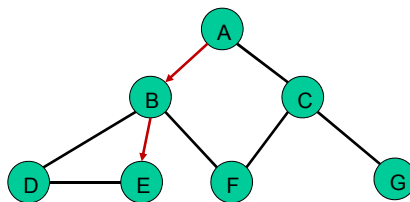
## Chương 5: Đồ thị

- Khái niệm và biểu diễn đồ thị
- Duyệt đồ thị
- **Tìm đường đi**
- Ứng dụng

1-37

## Tìm đường

- Bài toán: Tìm một đường giữa hai nút của đồ thị (ví dụ từ A tới E), các nút trên đường đi là không trùng nhau
- Cách giải: Dùng BFS

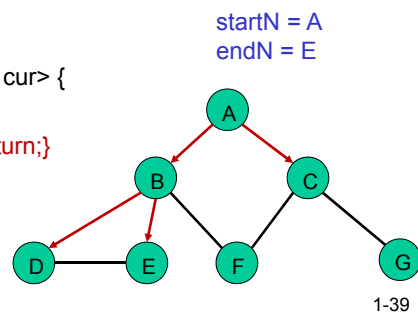


Sử dụng BFS với  
đỉnh xuất phát là A,  
tới khi thăm E

1-38

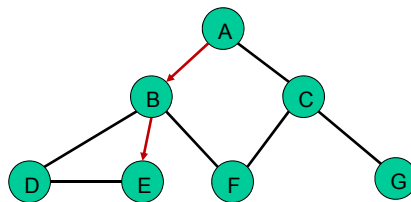
## BFS\_Path()

```
BFS_Path(gNode *startN, gNode *endN, graph g) {
    queue q;
    gNode *cur = startN;
    initialize q;
    mark cur as visited;
    enqueue(&q, cur);
    while (!emptyQueue(&q)) {
        cur = dequeue(&q);
        for all <đỉnh j chưa thăm và kề với cur> {
            if (j == endN)
                { printf "found the path!"; return;}
            mark j as visited;
            enqueue(&q, j);
        }
    }
}
```



## Đường đi ngắn nhất trên đồ thị không có trọng số

- Có nhiều đường đi từ một đỉnh nguồn tới một đỉnh đích
- Đường đi ngắn nhất đồ thị không có trọng số: Đường đi có số cạnh ít nhất
- Đường đi do BFS tìm là đường đi ngắn nhất



Sử dụng BFS với  
đỉnh xuất phát là A,  
tới khi thăm E

1-40

## Đường đi ngắn nhất trên đồ thị có trọng số Giải thuật Dijkstra

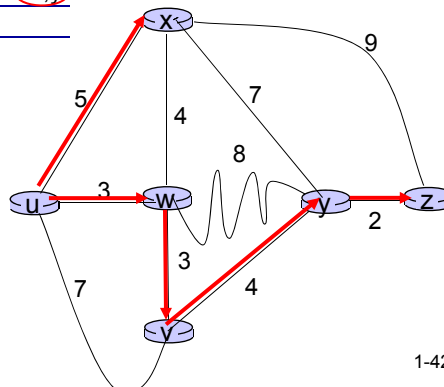
```

1  Khởi tạo:
2   $N' = \{u\}$ 
3  for all nodes  $v$ 
4    if  $v$  kề  $u$ 
5      then  $D(v) = c(u,v)$ 
6    else  $D(v) = \infty$ 
7
8  Loop
9    tìm  $w$  không trong  $N'$  mà  $D(w)$  nhỏ nhất
10   thêm  $w$  vào  $N'$ 
11   cập nhật  $D(v)$  cho mọi  $v$  kề với  $w$  và không trong  $N'$  :
12    $D(v) = \min(D(v), D(w) + c(w,v))$ 
13   /* chi phí mới tới  $v$  sẽ hoặc là chi phí cũ tới  $v$  hoặc là chi phí
    đường đi ngắn nhất tới  $w$  cộng với chi phí từ  $w$  tới  $v$  */
15  until mọi nút nằm trong  $N'$ 
    
```

1-41

## Giải thuật Dijkstra: Ví dụ 1

Bước	$N'$	$D(v)$ $p(v)$	$D(w)$ $p(w)$	$D(x)$ $p(x)$	$D(y)$ $p(y)$	$D(z)$ $p(z)$
0	$u$	7, $u$	3, $u$	5, $u$	$\infty$	$\infty$
1	$uw$	6, $w$		5, $u$	11, $w$	$\infty$
2	$uwx$	6, $w$			11, $w$	14, $x$
3	$uwxv$				10, $v$	14, $x$
4	$uwxvy$					12, $y$
5	$uwxvyz$					

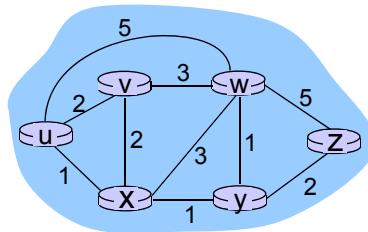


1-42



## Giải thuật Dijkstra: Ví dụ 2

Bước	N'	D(v),p(v)	D(w),p(w)	D(x),p(x)	D(y),p(y)	D(z),p(z)
0	u	2,u	5,u	1,u	$\infty$	$\infty$
1	ux	2,u	4,x		2,x	$\infty$
2	uxy	2,u	3,y			4,y
3	uxyv		3,y			4,y
4	uxyvw					4,y
5	uxyvwz					



1-43

## Giải thuật Bellman-Ford

- Tìm đường đi ngắn nhất từ một nút nguồn, cạnh đồ thị có thể có trọng số âm
- Thông báo nếu tồn tại chu trình âm

1-44

## Giải thuật Bellman-Ford: Khởi tạo

Initialize-Single-Source( $G, s$ )

- 1     for each vertex  $v$  in  $G.V$
- 2          $d[v] = \text{INFINITY}$
- 3          $p[v] = \text{NIL}$
- 4      $d[s] = 0$

1-45

## Giải thuật Bellman-Ford: Điều chỉnh

RELAX( $u, v, c$ )

- 1     if  $d[v] > d[u] + c(u, v)$
- 2          $d[v] = d[u] + c(u, v)$
- 3          $p[v] = u$

1-46

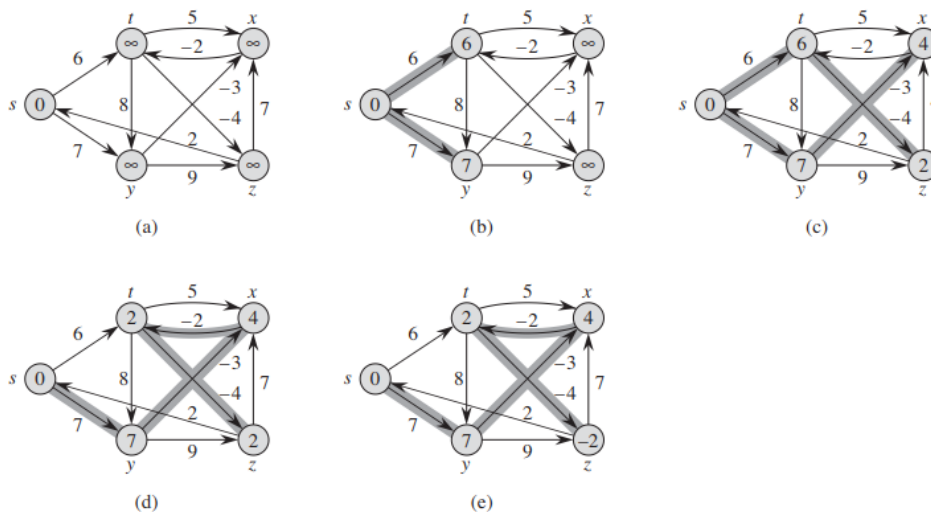
## Giải thuật Bellman-Ford

BELLMAN-FORD( $G, c, s$ )

- 1 Initialize-Single-Source( $G, s$ )
- 2 for  $i = 1$  to  $|G.V| - 1$
- 3     for each edge  $(u,v)$  in  $G.E$
- 4         RELAX( $u,v,c$ )
- 5     for each edge  $(u,v)$  in  $G.E$
- 6         if  $d[v] > d[u] + c(u,v)$
- 7             return FALSE
- 8     return TRUE

1-47

## Ví dụ Bellman-Ford



Thứ tự duyệt cạnh  $(t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)$

1-48

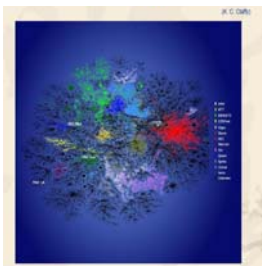
## Chương 5: Đồ thị

- Khái niệm và biểu diễn đồ thị
- Duyệt đồ thị
- Tìm đường đi
- **Ứng dụng**

1-49

## Đồ thị (mạng) xuất hiện trong mọi vấn đề của đời sống

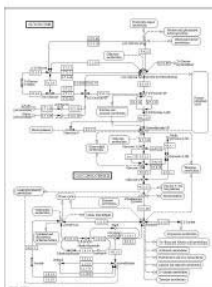
Internet



US Airline network

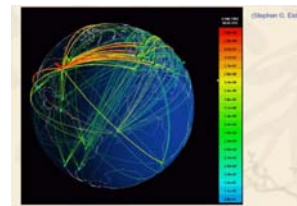


Neural network



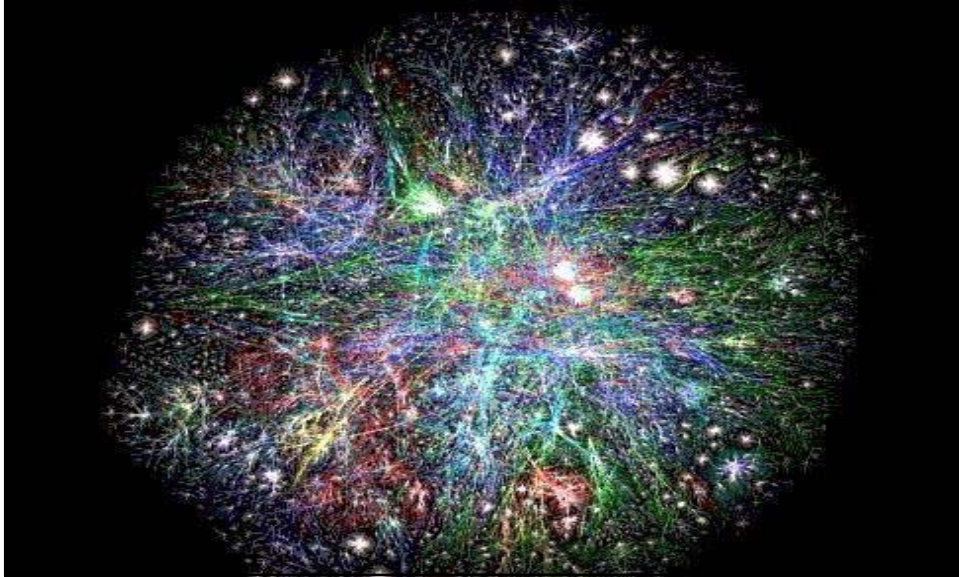
Metabolic network

Telecommunication network



Social network

## Đồ thị của WWW



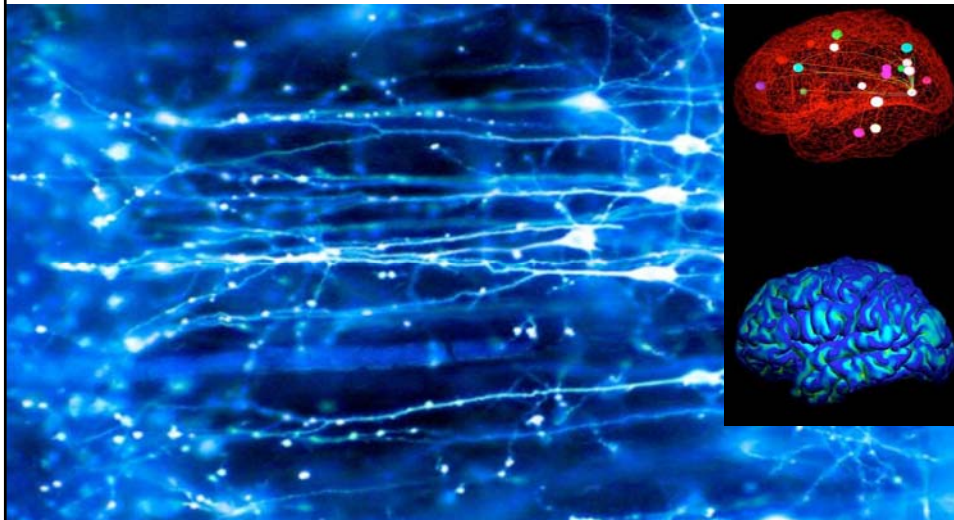
## Mạng xã hội Facebook



## Mạng xã hội Facebook



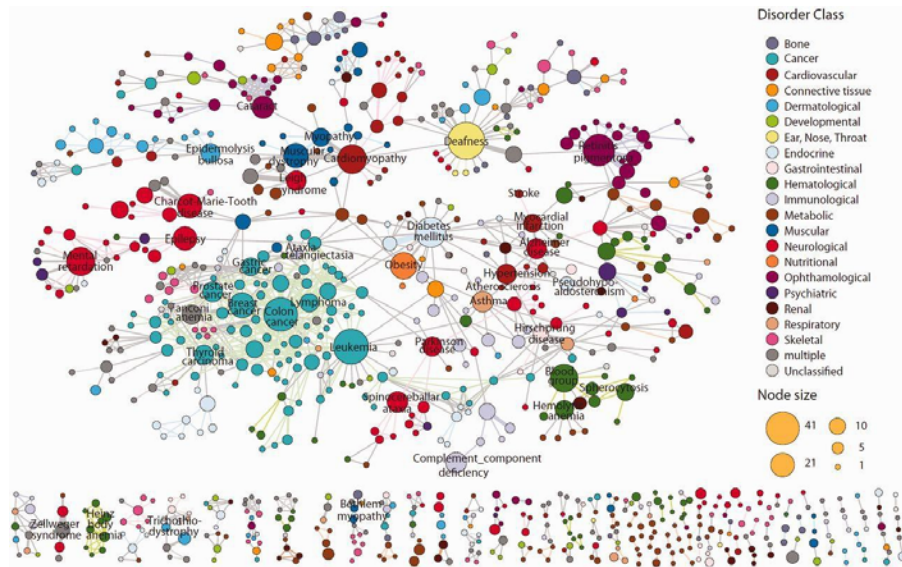
## Mạng nơ-ron bộ não con người: 10-100 tỉ nơ-ron



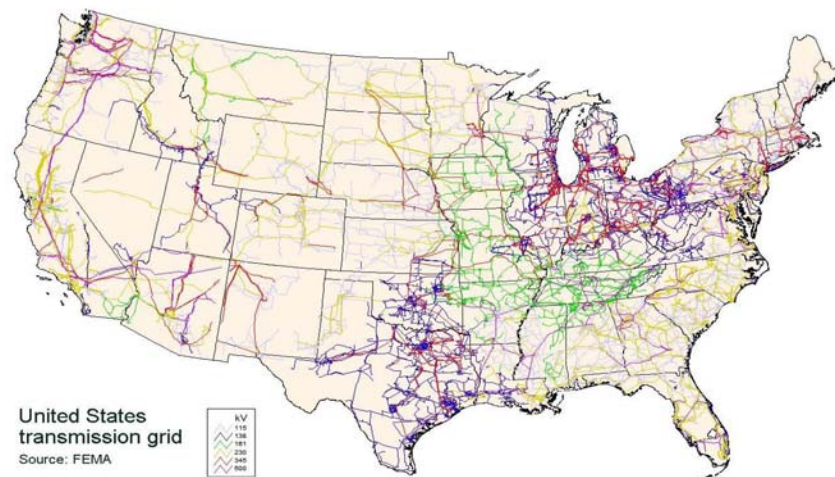
From Barabasi's ppt



## Mạng gen và bệnh

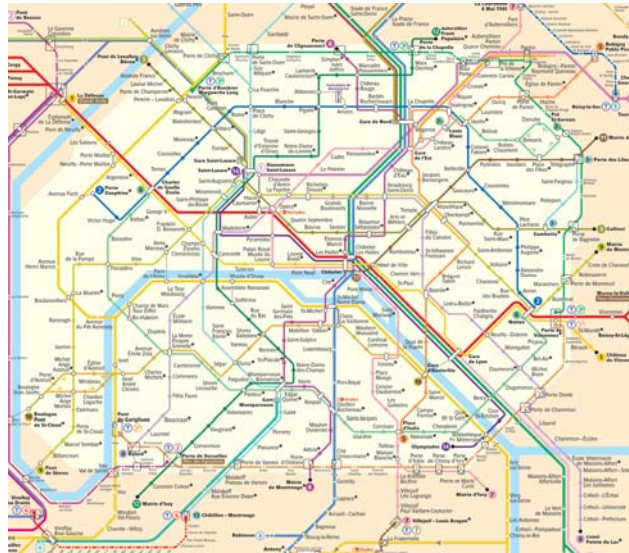


## Mạng năng lượng



Lưới năng lượng của nước Mỹ có 300 000 km đường dây với 500 công ty

## Mạng giao thông



## Cấu trúc dữ liệu và giải thuật

- Nội dung bài giảng được biên soạn bởi TS. Phạm Tuấn Minh.



# Cấu trúc dữ liệu và giải thuật

**TS. Phạm Tuấn Minh**

Khoa Công nghệ Thông tin, Đại học Phenikaa

[minh.phamtuan@phenikaa-uni.edu.vn](mailto:minh.phamtuan@phenikaa-uni.edu.vn)

<https://sites.google.com/site/phamtuanminh/>

## Chương 6: Một số vấn đề nâng cao

# Loop Invariant

1-3

## InsertionSort

**pre:** b 

0	b.length
?	

**post:** b 

0	b.length
đã sắp xếp	

**inv:** b 

0	i	b.length
đã sx?		

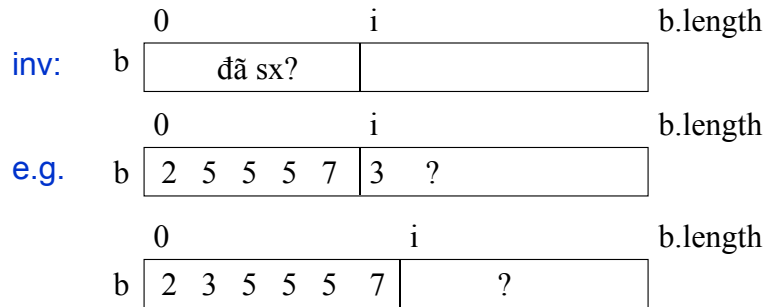
  
b[0..i-1] đã sắp xếp

**inv:** b 

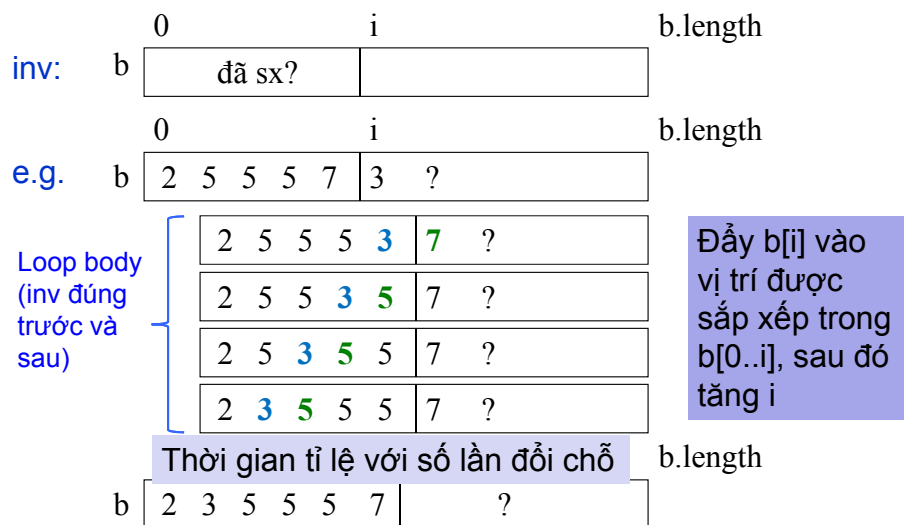
0	i	b.length
đã xử lý?		

  
b[0..i-1] đã xử lý

Mỗi vòng lặp,  $i = i+1$ : Giữ inv đúng?



Xử lý trong mỗi vòng lặp?



## Insertion Sort

```
// sắp xếp mảng số nguyên, b[]  
// inv: b[0..i-1] đã sắp xếp  
for (int i= 0; i < b.length; i= i+1) {  
    // Đẩy b[i] vào vị trí được sắp xếp  
    // trong b[0..i]  
  
}
```

## Insertion Sort

```
// sắp xếp mảng số nguyên, b[]  
// inv: b[0..i-1] đã sắp xếp  
for (int i= 0; i < b.length; i= i+1) {  
    // Đẩy b[i] vào vị trí được sắp xếp  
    // trong b[0..i]  
  
    int k= i;  
    while (k > 0 && b[k] < b[k-1]) {  
        <Đổi chỗ b[k] và b[k-1]>  
        k= k-1;  
    }  
}
```

invariant P: b[0..i] được sắp xếp, **riêng** b[k] có thể < b[k-1]

k			i	
2	5	3	5	5
7	?			

ví dụ

bắt đầu?

dừng?

tiến triển?

duy trì bất biến?

## Insertion Sort

```
// sắp xếp mảng số nguyên, b[]  
// inv: b[0..i-1] đã sắp xếp  
for (int i= 0; i < b.length; i= i+1) {  
    // Đẩy b[i] vào vị trí được sắp xếp  
    // trong b[0..i]  
}
```

`n = b.length`

- Worst-case:  $O(n^2)$  (reverse-sorted input)
- Best-case:  $O(n)$  (sorted input)
- Expected case:  $O(n^2)$

## Insertion Sort: Không xáo trộn

```
// sắp xếp mảng số nguyên, b[]  
// inv: b[0..i-1] đã sắp xếp  
for (int i= 0; i < b.length; i= i+1) {  
    // Đẩy b[i] vào vị trí được sắp xếp  
    // trong b[0..i]  
}
```

Một thuật toán sắp xếp gọi là không xáo trộn (stable) nếu hai giá trị bằng nhau giữ nguyên vị trí tương đối.

Ban đầu: (3 7, 2 8, 7, 6)

Sắp xếp không xáo trộn (2, 3, 6, 7, 7, 8)

Sắp xếp xáo trộn (2, 3, 6, 7, 7, 8)

## Hiệu năng

Algorithm	Độ phức tạp trường hợp trung bình và tập trường hợp tồi nhất	Bộ nhớ	Không xáo trộn?
Insertion Sort	$O(n^2)$ . $O(n^2)$	$O(1)$	Không xáo trộn

11

## SelectionSort

pre: b 0      ?      b.length      post: b 0      đã sx      b.length

inv: b 0      đã sx <= b[i..]      i      >= b[0..i-1]      b.length

Giữ invariant đúng?

e.g.: b 0      1 2 3 4 5 6      i      9 9 9 7 8 6 9      b.length

Tăng i lên 1 và inv đúng chỉ khi b[i] là min của b[i..]

## SelectionSort

```
// sắp xếp b[]  
// inv: b[0..i-1] đã sắp xếp và  
//      b[0..i-1] <= b[i..]  
for (int i = 0; i < b.length; i = i+1) {  
    int m = index of min of b[i..];  
    <Đổi chỗ b[i] và b[m]>  
}
```

$n = b.length$

- Worst-case  $O(n^2)$
- Best-case  $O(n^2)$
- Expected-case  $O(n^2)$

0                      i                      length  
b   đã sx, giá trị nhỏ hơn   |   giá trị lớn hơn

Mỗi lần lặp, đổi chỗ giá trị min của đoạn này và b[i]

## SelectionSort: Xáo trộn

```
// sắp xếp b[]  
// inv: b[0..i-1] đã sắp xếp và  
//      b[0..i-1] <= b[i..]  
for (int i = 0; i < b.length; i = i+1) {  
    int m = index of min of b[i..];  
    <Đổi chỗ b[i] và b[m]>  
}
```

Đổi chỗ b[i] với giá trị nhỏ nhất của b[i..], 3 đổi chỗ với 8 => thay đổi vị trí tương đối của hai giá trị 8

0                      i                      length  
b   đã sx, giá trị nhỏ hơn   |   8 7 8 3 5 6

## Hiệu năng

Algorithm	Độ phức tạp trường hợp trung bình và tập trường hợp tồi nhất		Bộ nhớ	Không xáo trộn?
Insertion Sort	$O(n^2)$ .	$O(n^2)$	$O(1)$	Không xáo trộn
Selection Sort	$O(n^2)$ .	$O(n^2)$	$O(1)$	Xáo trộn

15

## Quicksort



## Thuật toán phân đoạn

pre: 

h	h+1	k
x	?	

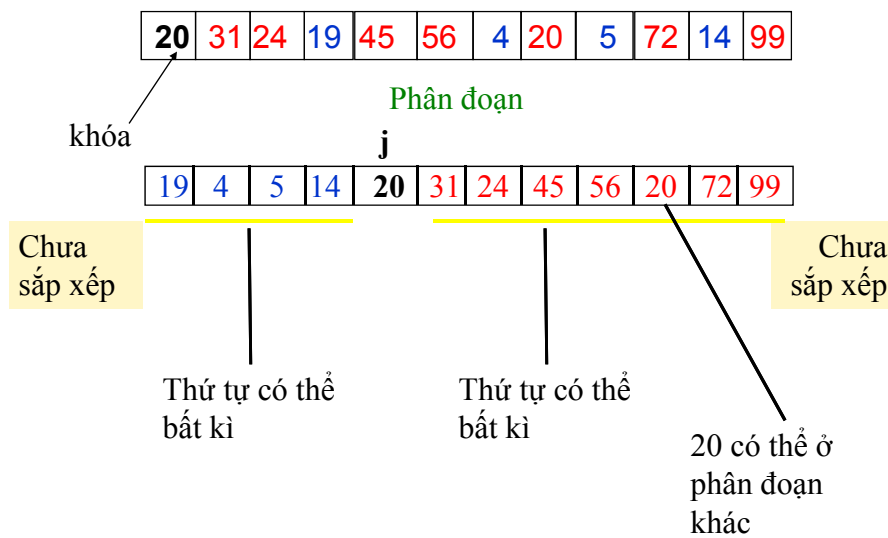
x là khóa

Đổi chỗ tới khi :

post: 

h	j	k
$\leq x$	x	$\geq x$

## Thuật toán phân đoạn



## Thuật toán phân đoạn

pre:  $b$   $\begin{array}{|c|c|} \hline x & ? \\ \hline \end{array}$   $\begin{array}{c} h \quad h+1 \quad k \end{array}$

post:  $b$   $\begin{array}{|c|c|c|} \hline \leq x & x & \geq x \\ \hline \end{array}$   $\begin{array}{c} h \quad j \quad k \end{array}$

Kết hợp pre và post để xác định invariant

$b$   $\begin{array}{|c|c|c|c|} \hline \leq x & x & ? & \geq x \\ \hline \end{array}$   $\begin{array}{c} h \quad j \quad t \quad k \end{array}$

invariant  
cần ít  
nhất 4  
phần

## Thuật toán phân đoạn

$b$   $\begin{array}{|c|c|c|c|} \hline \leq x & x & ? & \geq x \\ \hline \end{array}$   $\begin{array}{c} h \quad j \quad t \quad k \end{array}$

```
j = h; t = k;
while (j < t) {
    if (b[j+1] <= b[j]) {
        Đổi chỗ b[j+1], b[j]; j = j+1;
    } else {
        Đổi chỗ b[j+1], b[t]; t = t-1;
    }
}
```

Khởi tạo,  $j = h$  và  $t = k$ , sơ đồ giống sơ đồ bắt đầu

Kết thúc, khi  $j = t$ , phần “?” rỗng, sơ đồ thành sơ đồ kết quả

Thời gian tuyến tính:  $O(k+1-h)$

## QuickSort

**/\*\* Sắp xếp b[h..k]. \*/**

**public static void QS(int[] b, int h, int k) {**

**if (b[h..k] có 1 phần tử) return;**

Base case

**int j= partition(b, h, k);**

**// Có b[h..j-1] <= b[j] <= b[j+1..k]**

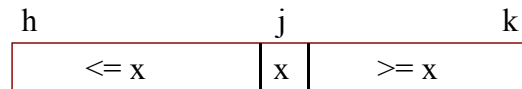
**// Sắp xếp b[h..j-1] and b[j+1..k]**

**QS(b, h, j-1);**

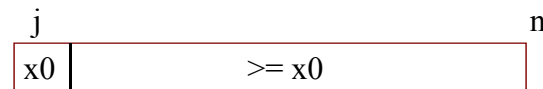
**QS(b, j+1, k);**

**}**

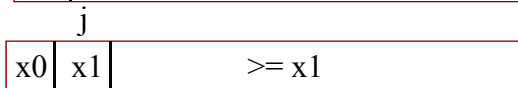
Hàm thực hiện thuật toán phân đoạn và trả về vị trí j của khóa



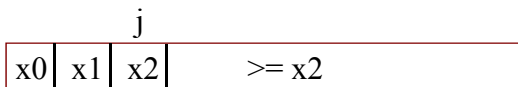
## Trường hợp tồi nhất: Khóa luôn là giá trị nhỏ nhất



Phân đoạn tại 0



Phân đoạn tại 1



Phân đoạn tại 2

**/\*\* Sắp xếp b[h..k]. \*/**

**public static void QS(int[] b, int h, int**

**k) {**

**if (b[h..k] has < 2 elements) return;**

**int j= partition(b, h, k);**

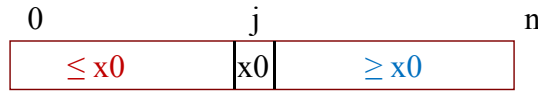
**QS(b, h, j-1); QS(b, j+1, k);**

Độ sâu đệ quy:  
 $O(n)$

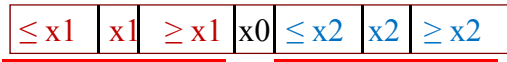
Xử lý tại độ sâu  
i:  $O(n-i)$

$O(n*n)$

## Trường hợp tốt nhất: Khóa luôn là giá trị ở giữa



Độ sâu 0. Phân đoạn n



Độ sâu 1. Phân 2 đoạn độ dài  $\leq n/2$



Độ sâu 2. Phân 4 đoạn độ dài  $\leq n/4$

Độ sâu tối đa:  $O(\log n)$ . Thời gian phân đoạn tại mỗi độ sâu  $O(n)$

Tổng thời gian:  $O(n \log n)$ .

## Cấu trúc dữ liệu và giải thuật

- Nội dung bài giảng được biên soạn bởi TS. Phạm Tuấn Minh.