

TRƯỜNG ĐẠI HỌC CÔNG NGHIỆP THÀNH PHỐ HỒ CHÍ MINH



Cấu trúc dữ liệu và giải thuật

Tổng quan

TS. Ngô Hữu Dũng

Dẫn nhập

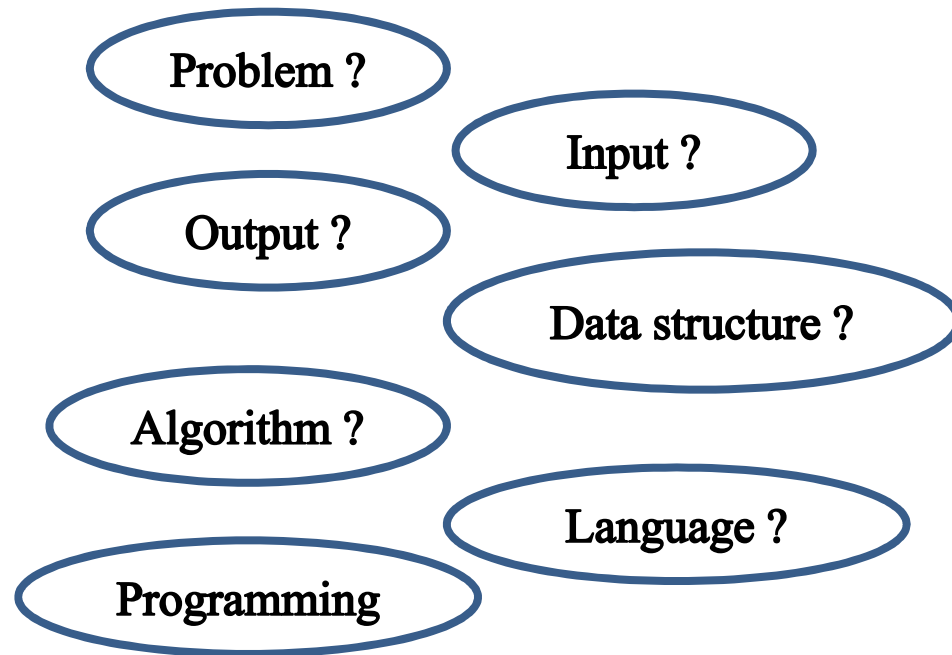
► Data structures and algorithms

► Data structures?

- Array
- Struct
- Linked list
- Stack
- Queue
- Tree
- Graph...

► Algorithm?

- Search
- Sort
- Recursion...

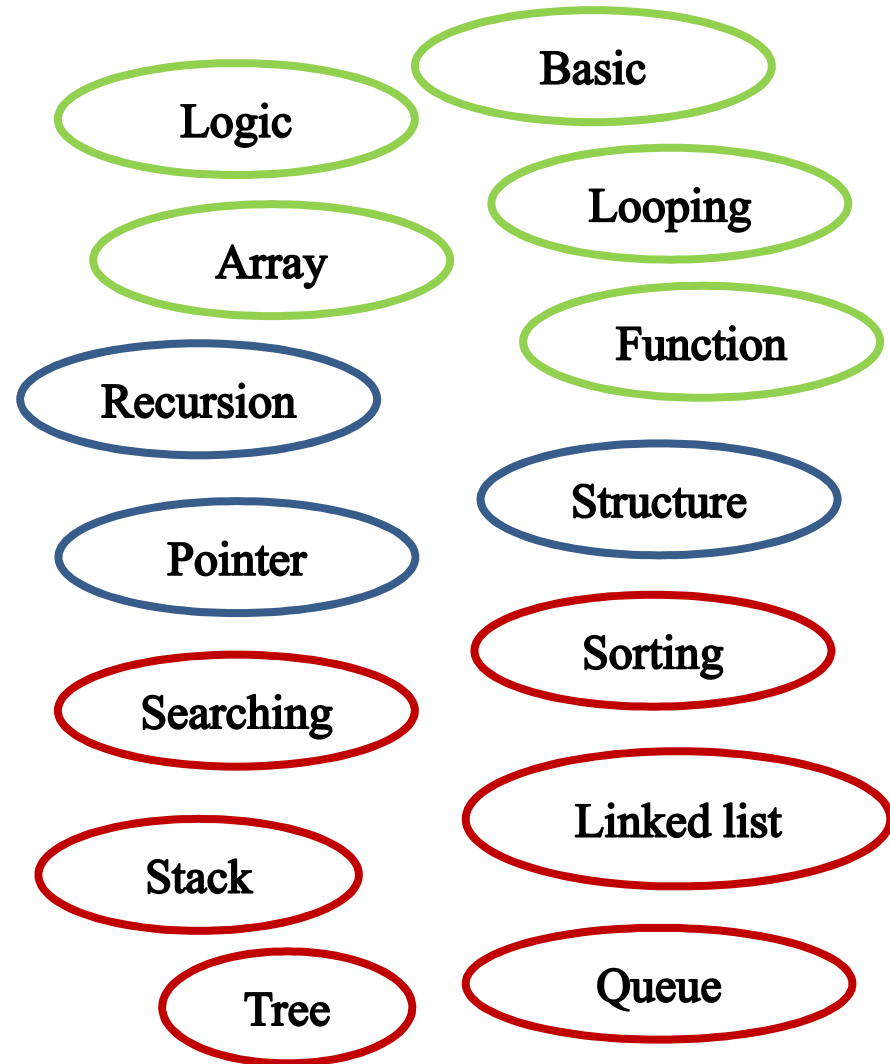


Blog ngohuudung.blogspot.com

Email ngohuudung@iuh.edu.vn

Nội dung

- ▶ Tổng quan
- ▶ Ôn tập căn bản
- ▶ Độ phức tạp thuật toán
- ▶ Giải thuật **tìm kiếm**
- ▶ Giải thuật **sắp xếp**
- ▶ **Danh sách liên kết**
- ▶ Ngăn xếp – stack
- ▶ Hàng đợi – Queue
- ▶ Cấu trúc cây – Tree
- ▶ Cấu trúc nâng cao



Tài liệu

- ▶ Trần Hạnh Nhi, Dương Anh Đức: Nhập môn cấu trúc dữ liệu và thuật toán. Khoa Công nghệ thông tin, ĐH KHTN TP HCM – 2000.
- ▶ Slide bài giảng
- ▶ Bài tập thực hành
- ▶ Đề tài bài tập lớn
- ▶ Tham khảo thêm
 - ▶ Robert L.Kruse, Alexander J.Ryba, Data Structures And Program Design In C++, PrenticeHall International Inc., 1999.
 - ▶ Nguyễn Ngô Bảo Trân, Giáo trình cấu trúc dữ liệu và giải thuật – Trường Đại học Bách Khoa TP.HCM, 2005.
 - ▶ Internet

Lịch trình

Tuần	Nội dung	Lý thuyết	Thực hành	Kiểm tra
1	Giới thiệu môn học- Tổng quan	3		
2	Giải thuật tìm kiếm	3		
3	Giải thuật sắp xếp	3	3	
4	Bài toán tìm kiếm, sắp xếp	3	3	
5-6	Danh sách liên kết	6	6	TK
7	Bài toán danh sách liên kết	3	3	GK
8-9	Ngăn xếp & Hàng đợi	6	6	
10	Bài toán ngăn xếp & Hàng đợi	3	3	
11	Cấu trúc cây	3	3	TK
12	Bài toán cấu trúc cây	3	3	Bài tập lớn
13-14	Bài toán tổng hợp	6		
15	Ôn tập	3		CK
		45	30	

Kiểm tra đánh giá

- ▶ Lý thuyết
 - ▶ Kiểm tra thường kỳ
 - ▶ Thi **cuối kỳ**
- ▶ Thực hành
 - ▶ Kiểm tra thường kỳ
 - ▶ Thi **giữa kỳ**
 - ▶ Bài tập lớn
- ▶ Điểm liệt: <3
- ▶ Số tín chỉ: 4
 - ▶ Lý thuyết: 45
 - ▶ Thực hành: 30
 - ▶ Tự học: 105



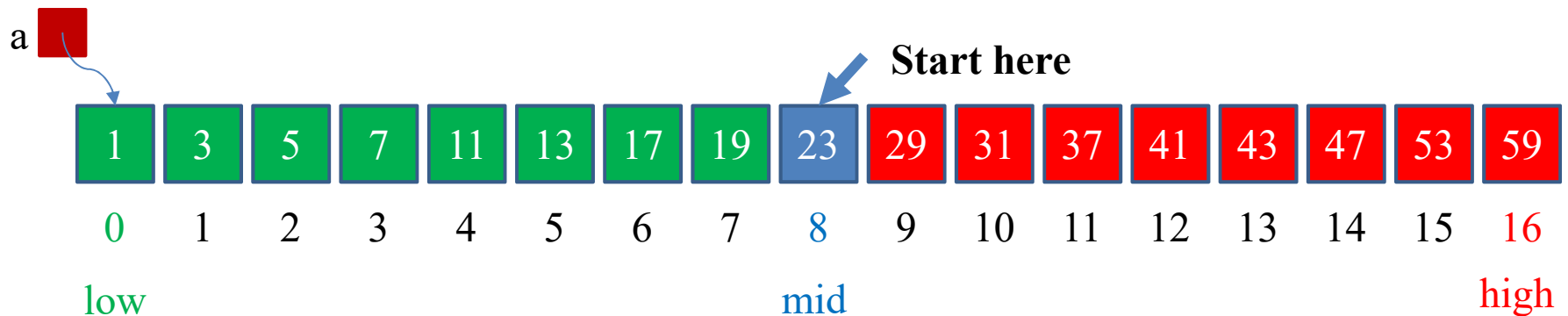
"Most good programmers do programming not because they expect to get paid or get adulation by the public, but because it is fun to program."

Linus Torvalds

Tìm kiếm – search

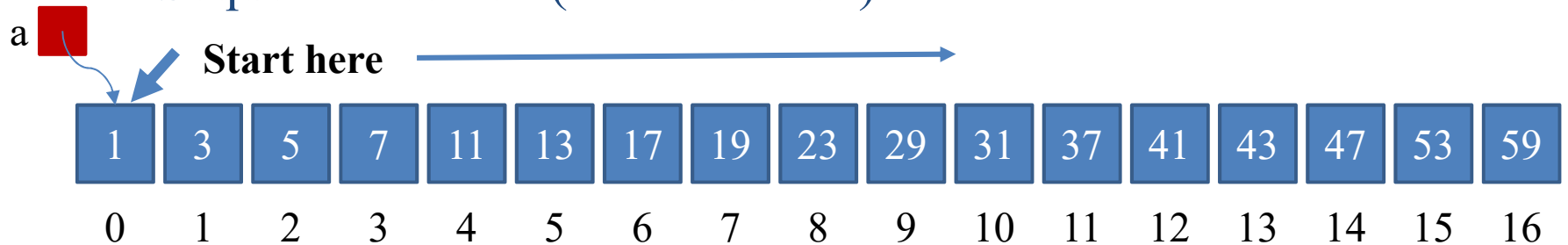
▶ Tìm kiếm nhị phân

▶ Binary search



▶ Tìm kiếm tuyến tính

▶ Sequential search (Linear search)



Binary vs Sequential search

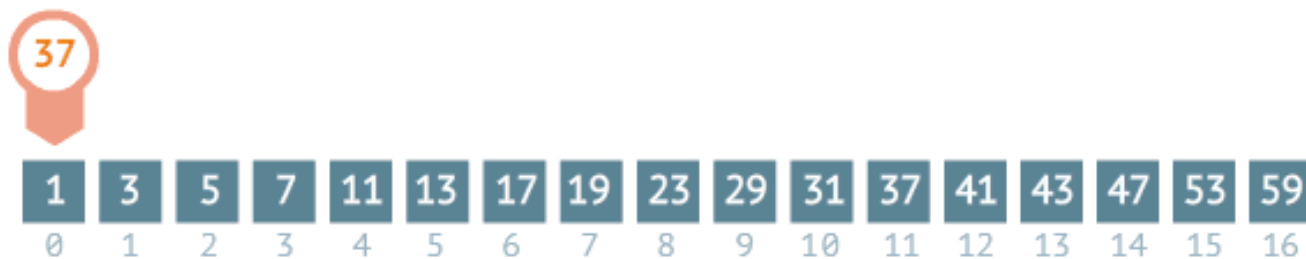
Binary search

steps: 0



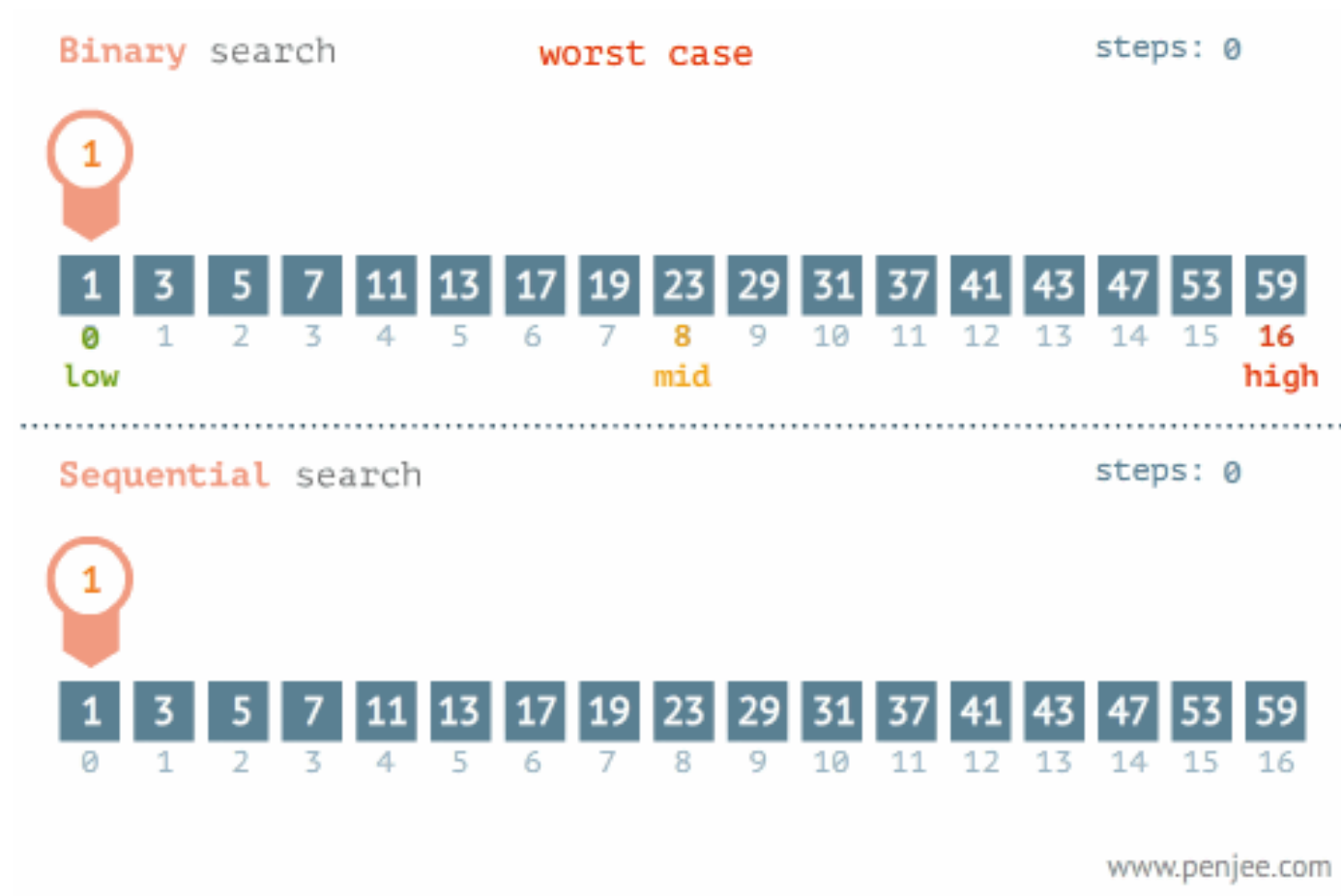
Sequential search

steps: 0



www.penjee.com

Binary search – worst case



Binary search – best case

Binary search

best case

steps: 0



Sequential search

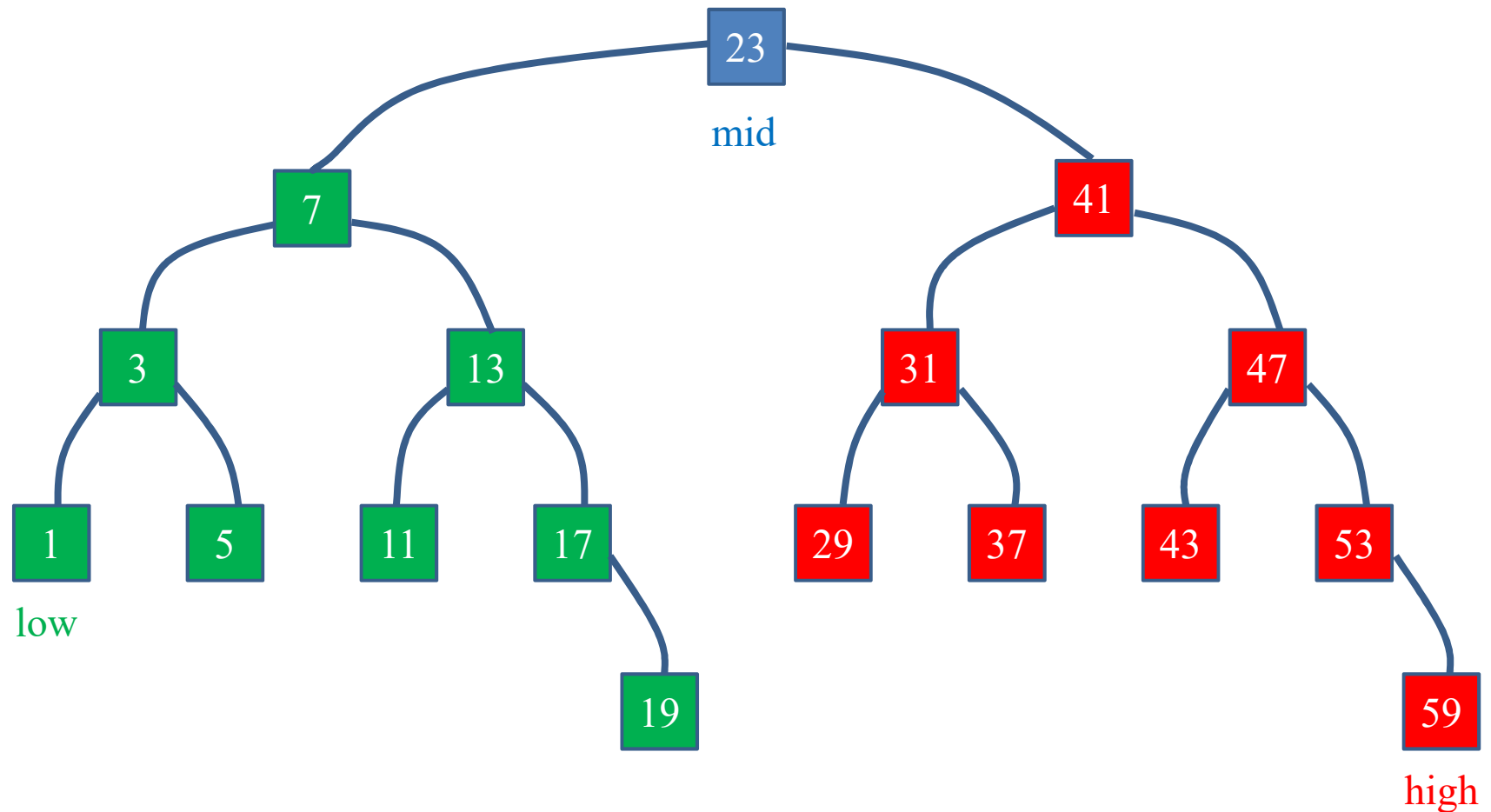
steps: 0



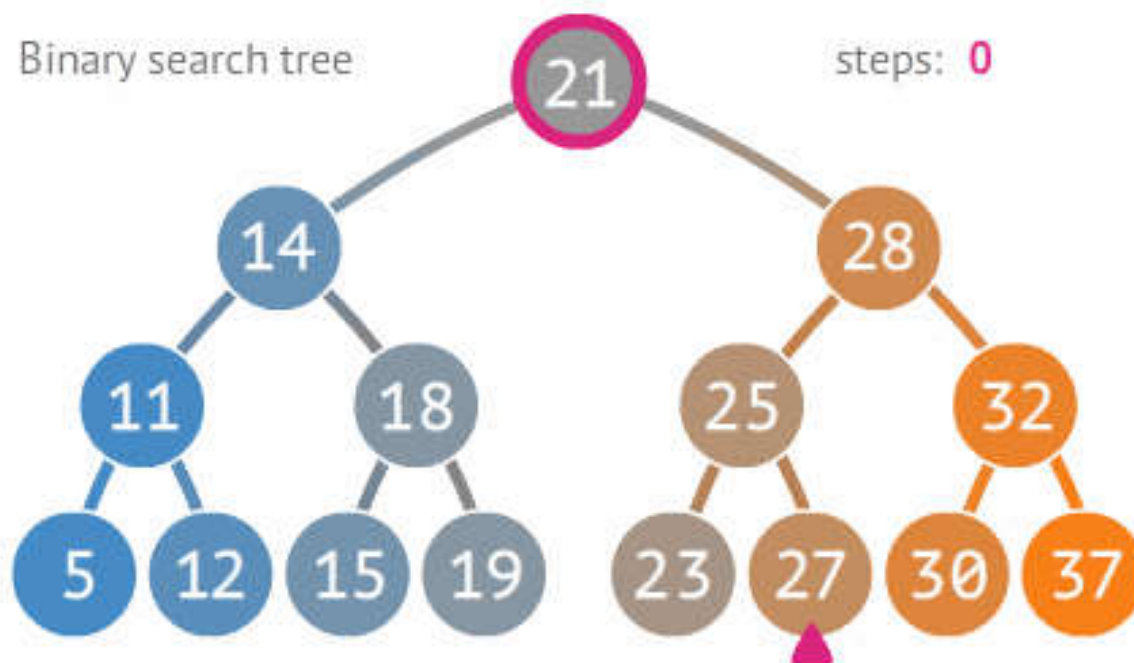
www.penjee.com

Binary search tree

- ▶ Viết mã giả cho thuật toán tìm kiếm nhị phân?



How's binary search tree work?



www.penjee.com

Sắp xếp – sort

- ▶ Selection Sort
- ▶ Insertion Sort và Shell Sort
- ▶ Interchange Sort
- ▶ Bubble sort và Shaker Sort
- ▶ Quick Sort
- ▶ Heap Sort
- ▶ Merge Sort

Các thuật toán sắp xếp

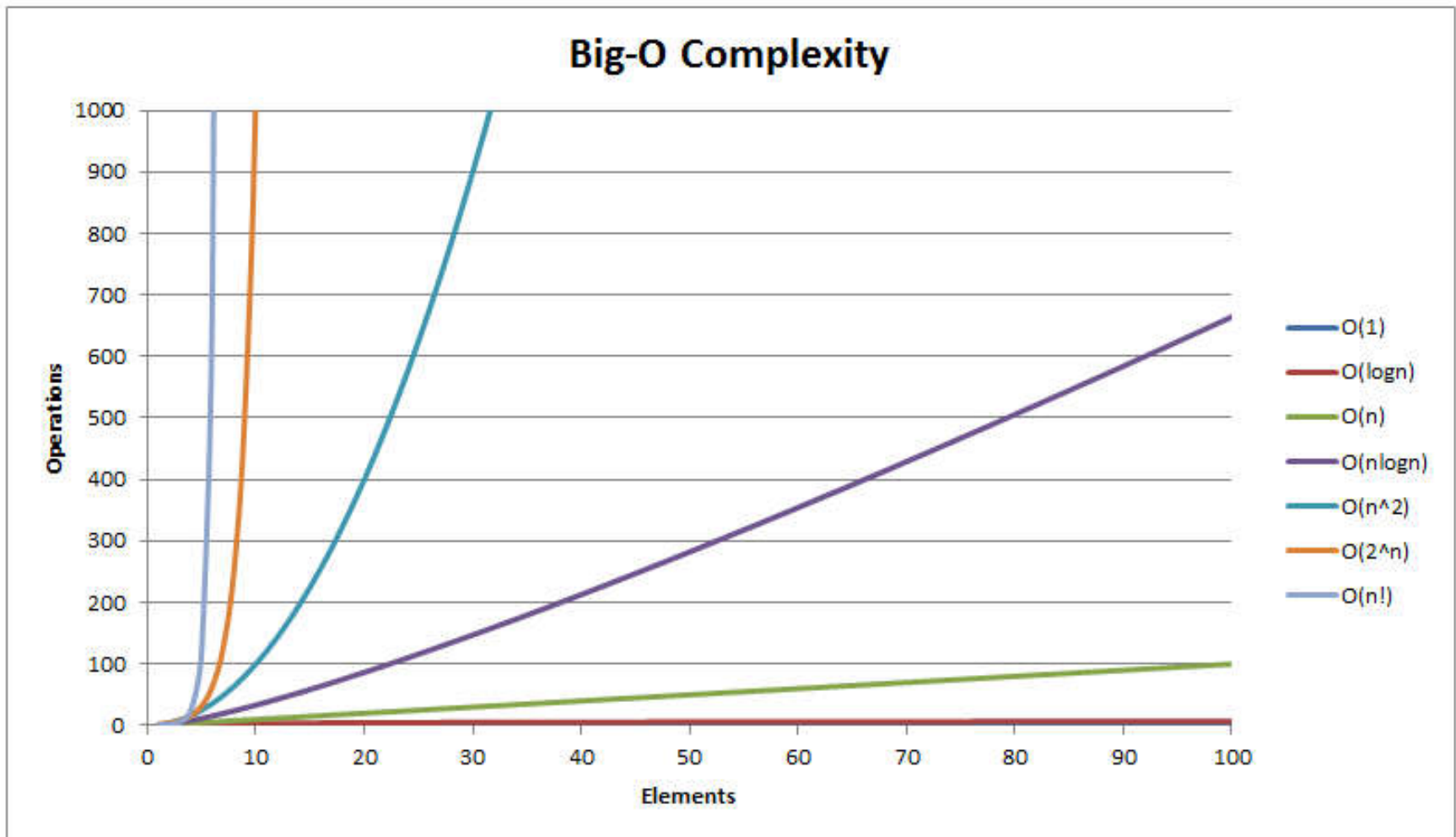
► <https://www.toptal.com/developers/sorting-algorithms/>

	 Insertion	 Selection	 Bubble	 Shell	 Merge	 Heap	 Quick	 Quick3
 Random								
 Nearly Sorted								
 Reversed								
 Few Unique								

Độ phức tạp của thuật toán?

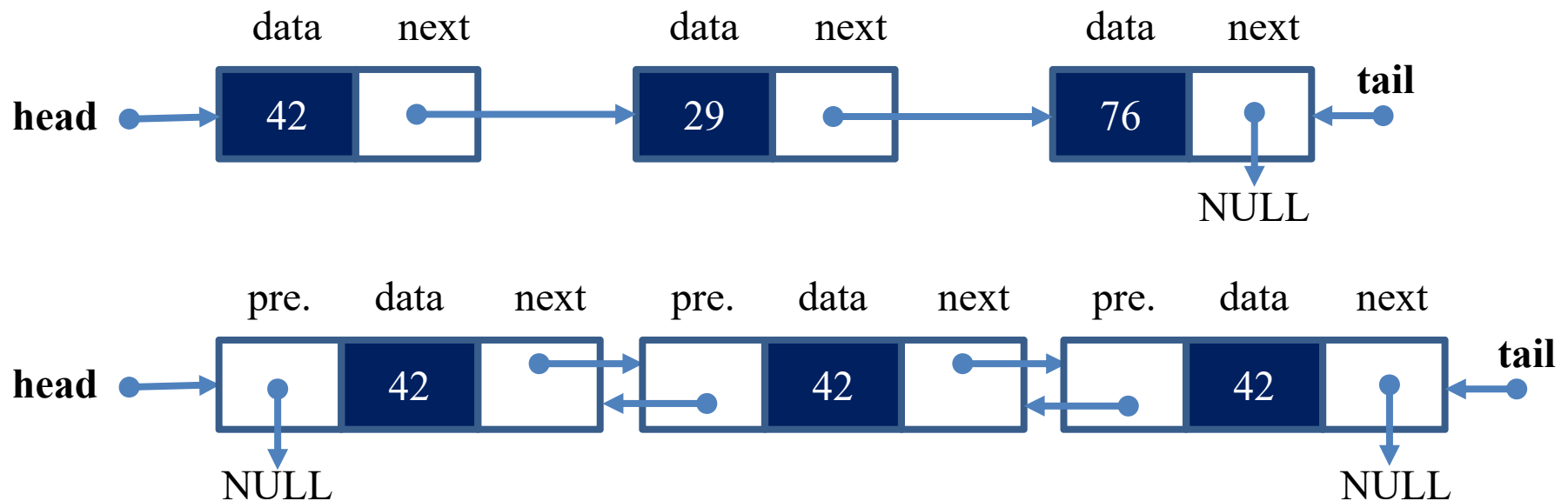
Algorithm	Best case	Average case	Worst case
Linear search	$O(1)$	$O(n)$	$O(n)$
Binary search	$O(1)$	$O(\log n)$	$O(\log n)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Shell sort	$O(n)$	$O((n \log(n))^2)$	$O((n \log(n))^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

Độ phức tạp big Oh



Danh sách liên kết – Linked list

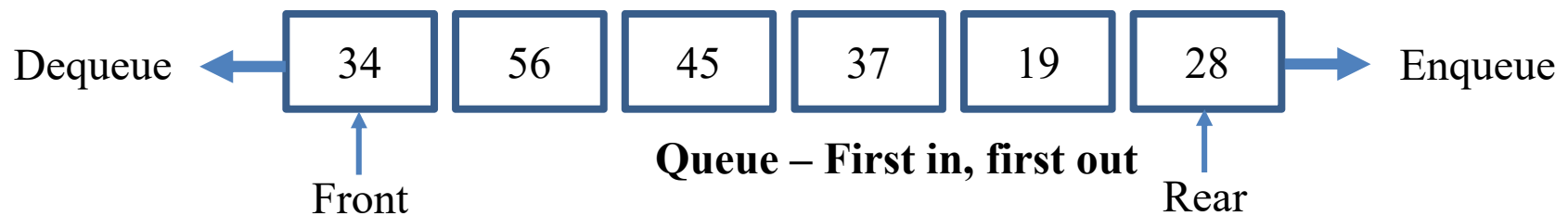
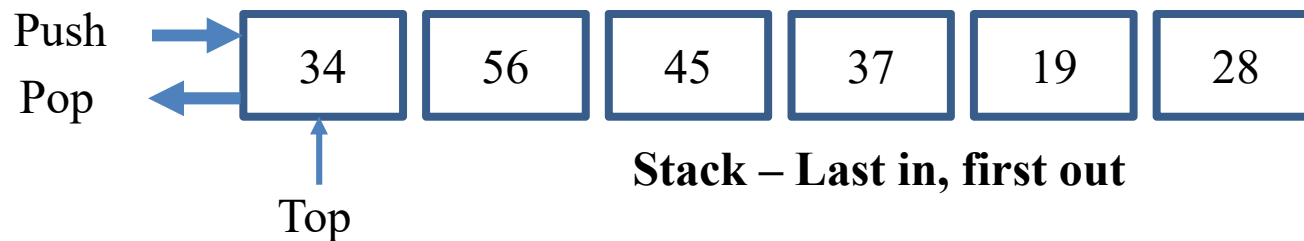
- ▶ Một dãy tuần tự các nút (Node)
- ▶ Giữa hai nút có con trỏ liên kết
- ▶ Các nút không cần phải lưu trữ liên tiếp nhau trong bộ nhớ
- ▶ Có thể mở rộng tùy ý
- ▶ <http://www.geeksforgeeks.org/data-structures/linked-list/>



Các thao tác cơ bản trên danh sách liên kết

- ▶ Thêm một nút
 - ▶ Vào đầu danh sách
 - ▶ Vào cuối danh sách
 - ▶ Vào sau một vị trí được chỉ ra (chèn)
- ▶ Xóa một nút
 - ▶ Ở đầu, cuối hoặc vị trí xác định trên danh sách
- ▶ Duyệt danh sách
 - ▶ Xuất, trích xuất, đếm, tính toán
- ▶ Tìm kiếm
 - ▶ Max, min, giá trị x
- ▶ Sắp xếp danh sách

Ngăn xếp và hàng đợi – Stack & Queue



Ví dụ về Stack và Queue

Stack (LIFO - last in, first out) - a collection of items in which only the most recently added item may be removed.



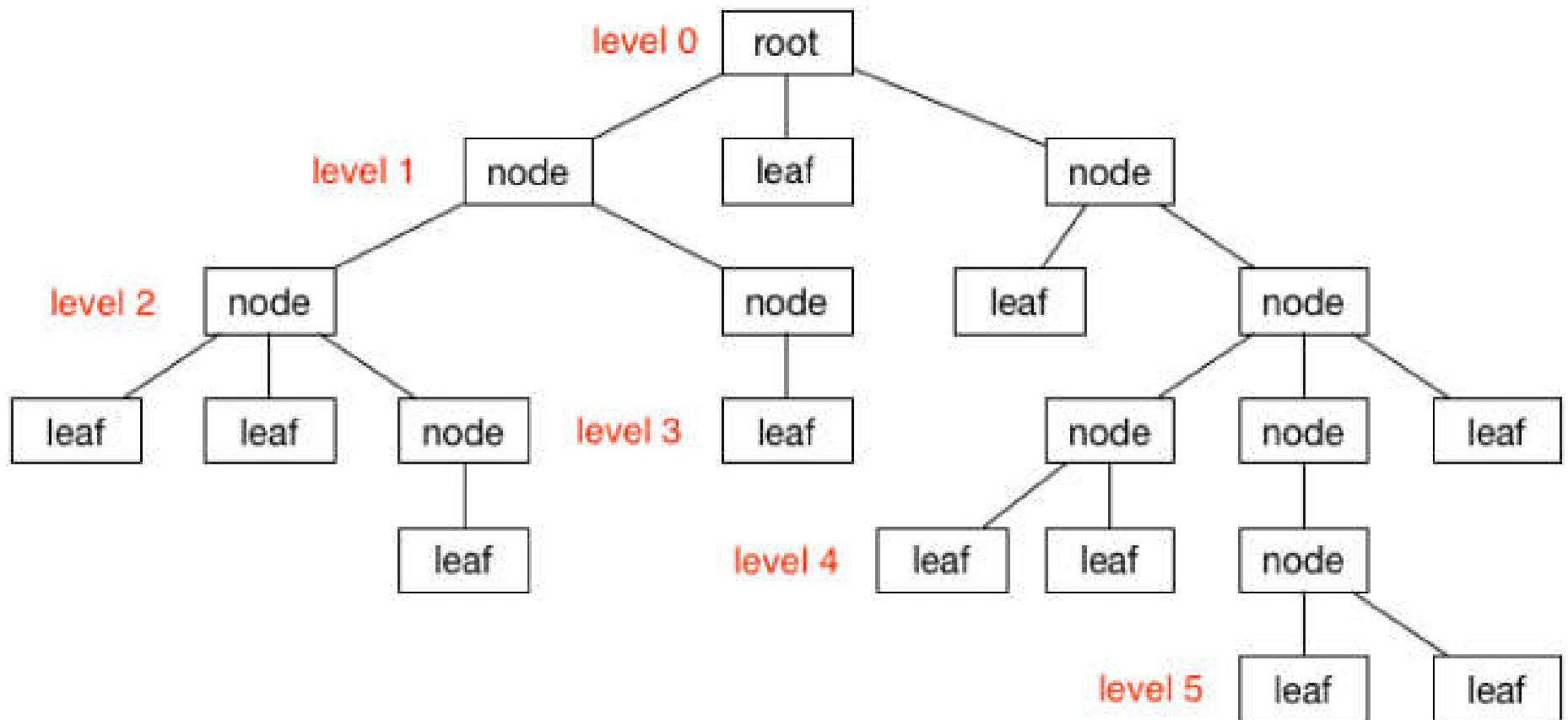
The only book you can take without ruining the pile is the one on the top

Queue (FIFO - first in, first out) - a collection of items in which first items entered are the first ones to be removed.



The only way to exit is to wait for your turn

Cấu trúc cây



Củng cố kiến thức qua bài tập ví dụ

- ▶ Đề bài: Để quản lý sinh viên gồm các thông tin: ID, Tên, Điểm thường kỳ, Điểm giữa kỳ, Điểm cuối kỳ, và Điểm tổng kết ($20\%ĐTK + 30\%ĐGK + 50\%ĐCK$) bạn hãy xây dựng các tác vụ theo mô tả sau:
 1. Nhập thông tin một sinh viên
 2. Xuất thông tin một sinh viên
 3. Thêm một sinh viên vào danh sách
 4. Xuất tất cả danh sách sinh viên
 5. Tìm kiếm sinh viên theo mã
 6. Xóa một sinh viên tại vị trí chỉ ra
 7. Cập nhật (sửa) thông tin một sinh viên
 8. Hiển thị sinh viên có số điểm cao nhất
 9. Sắp xếp sinh viên theo điểm tổng kết
 10. Menu thực hiện các tác vụ trên

TRƯỜNG ĐẠI HỌC CÔNG NGHIỆP THÀNH PHỐ HỒ CHÍ MINH



Cấu trúc dữ liệu và giải thuật

Giải thuật tìm kiếm

TS. Ngô Hữu Dũng

Tìm kiếm – Searching

- ▶ Tìm kiếm tuần tự - Sequential search
 - ▶ Còn gọi là tuyến tính – Linear search
 - ▶ Danh sách chưa sắp xếp hoặc đã sắp xếp
 - ▶ Thời gian tỉ lệ với n (số phần tử)
 - ▶ Độ phức tạp $O(n)$
- ▶ Tìm kiếm nhị phân – Binary search
 - ▶ Danh sách đã sắp xếp
 - ▶ Thời gian tỉ lệ với $\log_2 n$
 - ▶ Độ phức tạp $O(\log n)$

Sequential search

- ▶ Duyệt danh sách từ đầu đến cuối
 - ▶ Dừng khi tìm thấy hoặc kết thúc danh sách
 - ▶ Nếu tìm thấy: Trả về kết quả tìm thấy
 - ▶ True hoặc vị trí được tìm thấy hoặc thông báo
 - ▶ Nếu không tìm thấy: Trả về kết quả không tìm thấy
 - ▶ False hoặc một giá trị như -1 hoặc thông báo

Linear Search



Sequential search – Vòng lặp

- ▶ Trả về vị trí khi tìm thấy
- ▶ Trả về -1 khi không tìm thấy
- ▶ Lưu ý: *Các code chỉ mang tính minh họa cho giải thuật*
 - ▶ Có nhiều cách diễn đạt và cải tiến thuật toán

```
1. int linearSearch(int a[], int n, int x)
2. {
3.     int i;
4.     for(i=0; i<n; i++)
5.         if(a[i] == x)
6.             return i;
7.     return -1;
8. }
```

Sequential search – Vòng lặp

- ▶ Trả về kiểu bool
 - ▶ True: Tìm thấy
 - ▶ False: Không tìm thấy

```
1. bool linearSearch(int a[], int n, int x)
2. {
3.     int i;
4.     for(i=0; i<n; i++)
5.         if(a[i] == x)
6.             return true;
7.     return false;
8. }
```

Sequential search – Thông báo

- ▶ Xuất ra màn hình kết quả

```
1. void linearSearch(int a[], int n, int x)
2. {
3.     int i;
4.     for(i=0; i<n; i++)
5.         if(a[i] == x)
6.         {
7.             printf("Tim thay o vi tri %d", i);
8.             break;
9.         }
10.    if(i==n)
11.        printf("Khong tim thay");
12. }
```

Sequential search – Cờ hiệu

- ▶ Dùng cờ hiệu: Chương trình rõ ràng, dễ hiểu

```
1. void linearSearch(int a[], int n, int x)
2. {
3.     int i, flag = 0;        // Chưa tìm thấy
4.     for(i=0; i<n; i++)
5.         if(a[i] == x) {
6.             printf("Tim thay o vi tri %d", i);
7.             flag = 1;        // Đã tìm thấy
8.             break;
9.         }
10.    if(!flag)
11.        printf("Khong tim thay");
12. }
```

Sequential search – Độ quy

- ▶ Dùng đệ quy
 - ▶ Thực hiện gọi hàm nhiều lần

```
1. int linearSearch(int a[], int n, int x)
2. {
3.     if (n<0)
4.         return -1;
5.     else if (a[n-1] == x)
6.         return n-1;
7.     else
8.         return linearSearch(a, n-1, x);
9. }
```

Sequential search – Cầm canh

- ▶ Dùng phần tử cầm canh
 - ▶ Giảm bớt số lần so sánh

```
1. int linearSearch(int a[], int n, int x)
2. {
3.     int i = 0;
4.     a[n]=x;           // Phần tử cầm canh
5.     while (a[i] !=x)
6.         i++;
7.     if (i<n)
8.         return i;
9.     return -1;
10. }
```

Sequential search – Rút gọn

- ▶ Giảm thiểu số phép toán

```
1. int linearSearch(int a[], int n, int x)
2. {
3.     do{
4.         n--;
5.     }while (a[n] !=x && n>=0) ;
6.     return n;
7. }
```


Sequential search – Hai chiều

► Tìm cả hai chiều

```
1. int doubleSearch(int a[], int n, int x)
2. {
3.     int i=-1;
4.     do{
5.         if(a[--n]==x) return n;
6.         if(a[++i]==x) return i;
7.     }while(i<n) ;
8.     return -1;
9. }
```

So sánh thực nghiệm

- ▶ Thực hiện 1 triệu phép lặp cho mỗi hàm

- ▶ Cơ bản

- ▶ Độ quy

- ▶ Cầm canh

- ▶ Rút gọn

- ▶ Hai chiều

- 1. `for(int i=0; i<1000; i++)`

- 2. `for(int j=0; j<1000; j++)`

- 3. `k = linearSearch(a, n, x);`

- ▶ Phần tử cần tìm nằm ở vị trí trong trường hợp xấu nhất (worst case)

- ▶ Đo thời gian thực hiện của mỗi hàm để so sánh kết quả

Cách đo thời gian

- ▶ Một số IDE có sẵn chức năng đo thời gian

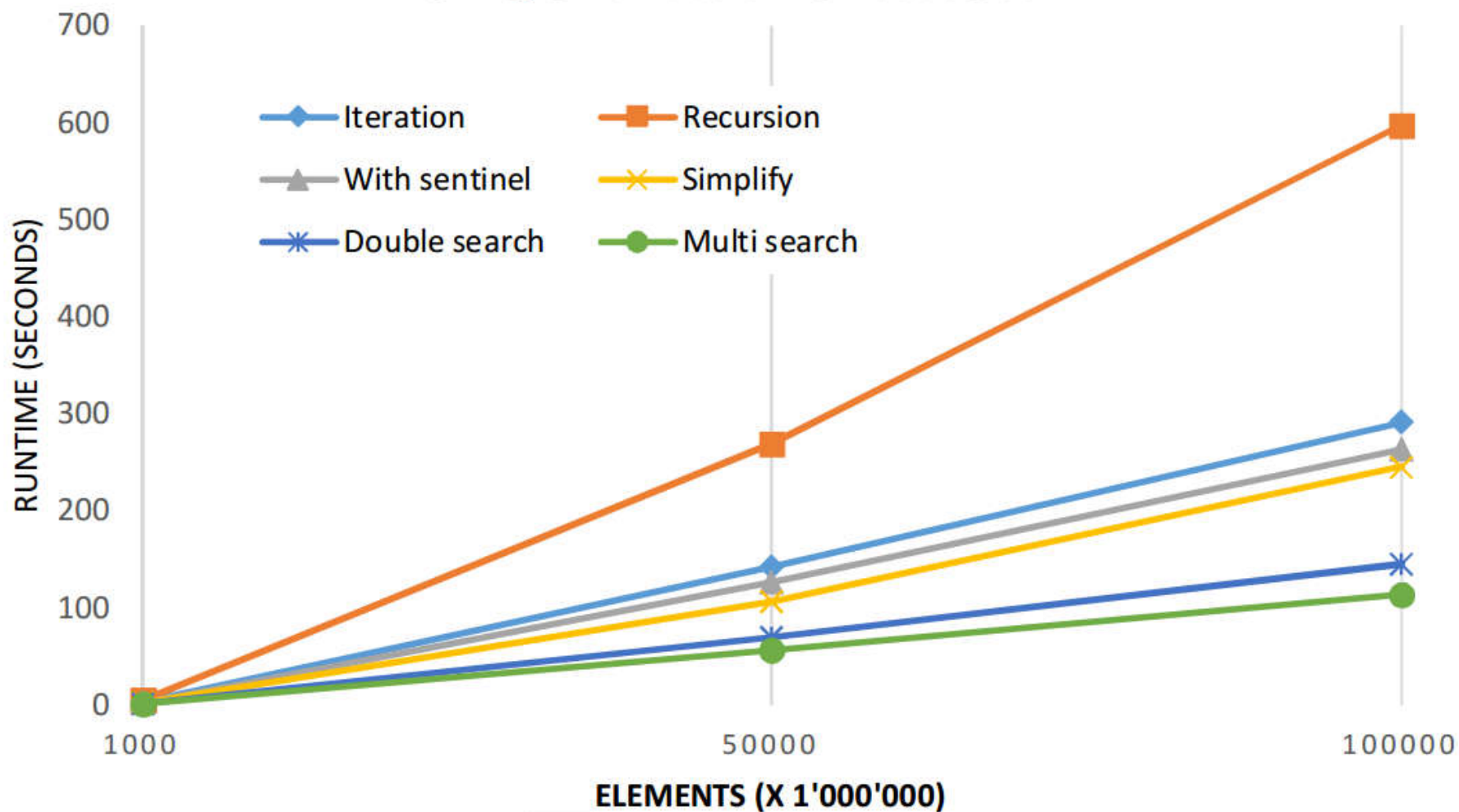
```
1. #include <stdio.h>
2. #include <time.h>
3. int main()
4. {
5.     clock_t t = clock();

6.     // Đoạn code cần đo thời gian

7.     t = clock() - t;
8.     printf("Time: %.2fs\n", (float) t / CLOCKS_PER_SEC);
9.     return 0;
10. }
```

Đánh giá

SEQUENTIAL SEARCH



Bài tập vận dụng

- ▶ Viết hàm tìm kiếm phần tử x trong khoảng từ *left* đến *right* trong mảng.
- ▶ Nguyên mẫu hàm?
- ▶ Sử dụng hàm?

Binary search

- ▶ Danh sách đã được sắp xếp, giả sử tăng dần
 - ▶ So sánh X với phần tử ở giữa danh sách
 - ▶ Nếu bằng nhau: Tìm kiếm thành công
 - ▶ Nếu X nhỏ hơn: Tiếp tục tìm bên trái danh sách
 - ▶ Nếu X lớn hơn: Tiếp tục tìm bên phải danh sách



Binary search – Iteration

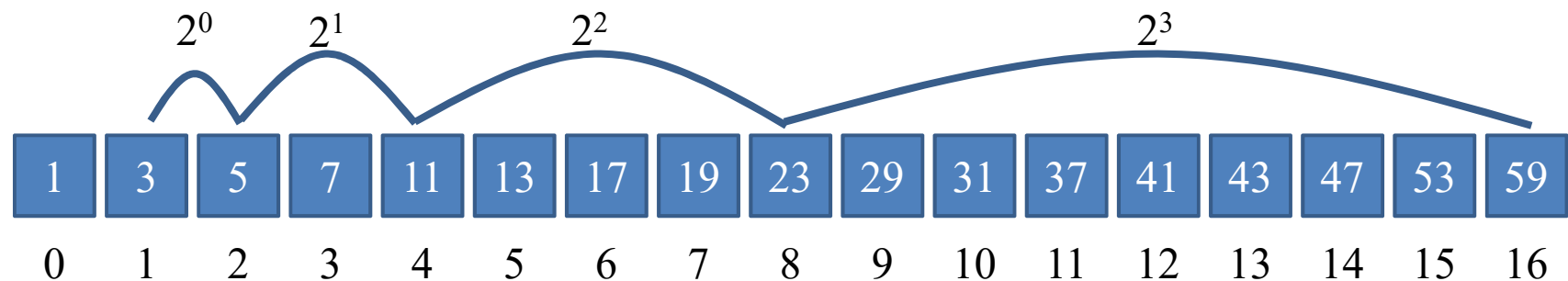
```
1. int binarySearch(int a[],int left,int right,int x)
2. {
3.     while(left<=right)
4.     {
5.         int mid = (left + right)/2;
6.         if(x==a[mid])
7.             return mid;
8.         if(x<a[mid])
9.             right = mid-1;
10.        else
11.            left = mid+1;
12.    }
13.    return -1;
14.}
```

Binary search – Recursion

```
1. int binarySearch(int a[], int left, int right, int x)
2. {
3.     if (left <= right)
4.     {
5.         int mid = left + (right - left) / 2;    //?
6.         if (x == a[mid])
7.             return mid;
8.         if (x < a[mid])
9.             return binarySearch(a, left, mid-1, x);
10.        else
11.            return binarySearch(a, mid+1, right, x);
12.    }
13.    return -1;
14.}
```


Exponential Search

- ▶ Bao gồm hai bước
 - ▶ Xác định vùng chứa X trong mảng
 - ▶ Lần lượt so sánh X với các phần tử i bắt đầu từ 1, 2, 4, 8, 16... tăng dần theo lũy thừa 2.
 - ▶ Khi tìm được vị trí của phần tử i có giá trị lớn hơn X , vùng cần tìm là từ $i/2$ đến $\min(i, n)$
 - ▶ Dùng binary search để tìm trong vùng đã xác định



Exponential Search

```
1. int exponentialSearch(int a[], int n, int x)
2. {
3.     if (a[0] == x)
4.         return 0;
5.
6.     int i=1;
7.     while (i < n && a[i] <= x)
8.         i = i*2;
9.
10.    return binarySearch(a, i/2, (i<n)?i:n, x);
10. }
```

Interpolation Search

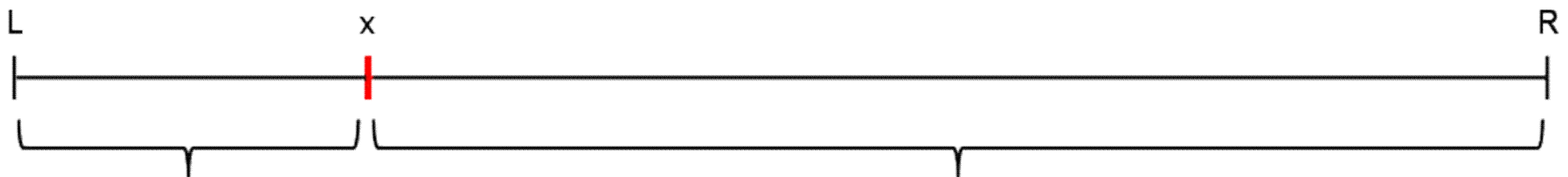
- ▶ Điểm mid không nhất thiết chính giữa

- ▶ Cách tính điểm ở giữa

`mid = low + (x - a[low]) * (high - low) / (a[high] - a[low]) ;`

- ▶ mid sẽ gần điểm low khi x gần a[low] hơn

- ▶ mid sẽ gần điểm high khi x gần a[high] hơn

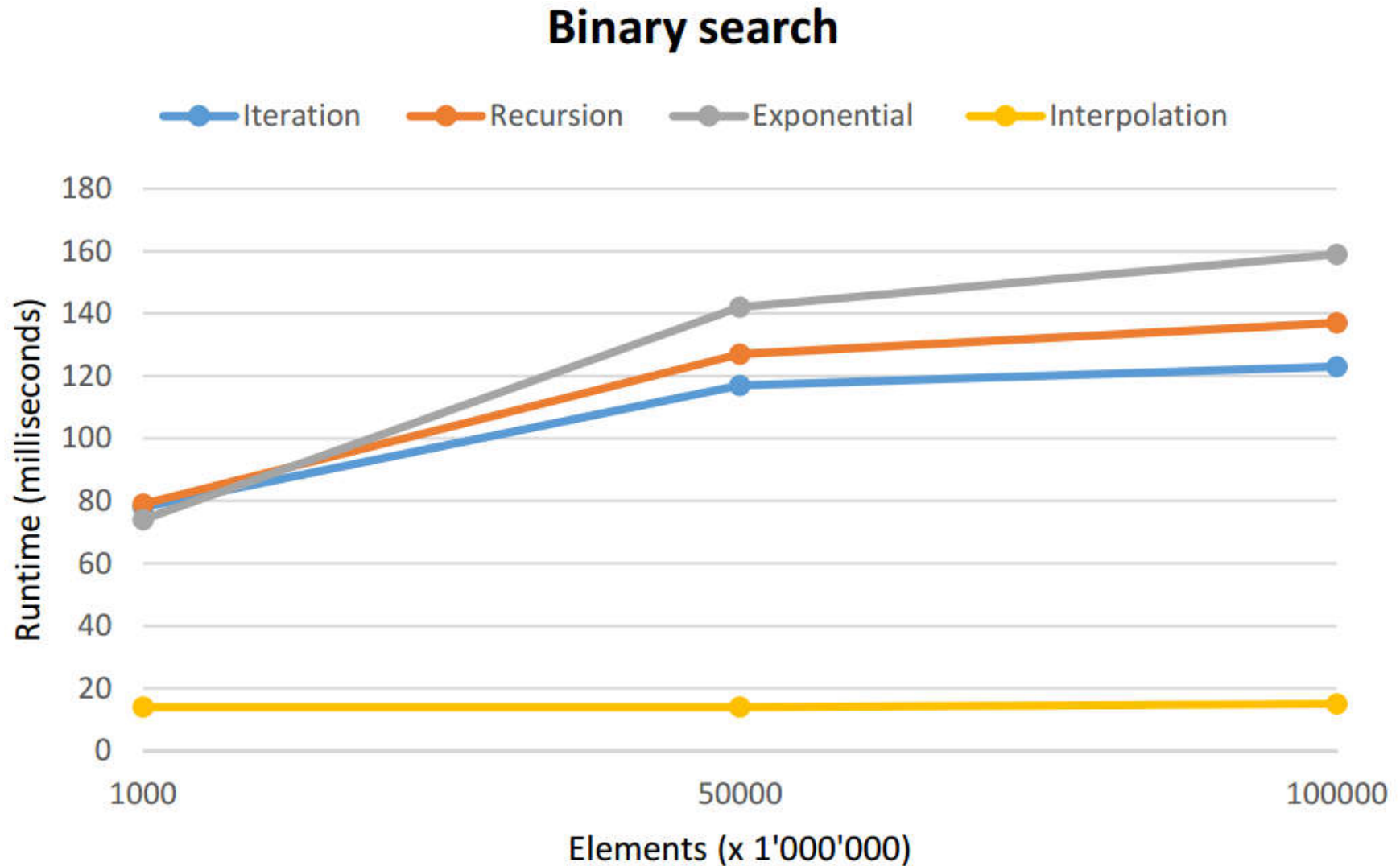


$$C = (x - L) / (R - L)$$

Interpolation Search

```
1. int interpolationSearch(int a[], int size, int x)
2. {
3.     int low = 0, high = size - 1, mid;
4.     while(high >= low && x >= a[low] && x <= a[high])
5.     {
6.         mid = low + (x - a[low]) * (high - low) / (a[high] - a[low]);
7.         if (a[mid] < x)
8.             low = mid + 1;
9.         else if (x < a[mid])
10.            high = mid - 1;
11.         else
12.            return mid;
13.     }
14.     return -1;
15. }
```

Một kết quả so sánh



Độ phức tạp?

- ▶ Một trường hợp so sánh không đánh giá đầy đủ về các thuật toán
 - ▶ Cần nhiều trường hợp hơn

Algorithm	Best case	Average case	Worst case
Linear search	$O(1)$	$O(n)$	$O(n)$
Binary search	$O(1)$	$O(\log n)$	$O(\log n)$
Exponential Search	$O(1)$	$O(\log i)$	$O(\log i)$ Với i là vị trí cần tìm
Interpolation Search	$O(1)$	$O(\log(\log n))$	$O(n)$

Bài tập vận dụng

- ▶ Viết chương trình
 - ▶ Phát sinh ngẫu nhiên một mảng tăng dần
 - ▶ Cài đặt các hàm tìm kiếm
 - ▶ Tìm giá trị x nhập từ bàn phím
 - ▶ Xuất ra số lần so sánh của mỗi phương pháp
 - ▶ Đánh giá các phương pháp
- ▶ Tìm hiểu hoặc đề xuất phương pháp mới

TRƯỜNG ĐẠI HỌC CÔNG NGHIỆP THÀNH PHỐ HỒ CHÍ MINH



Cấu trúc dữ liệu và giải thuật

Giải thuật sắp xếp

TS. Ngô Hữu Dũng

Sắp xếp – sort

- ▶ Selection Sort
 - ▶ Insertion Sort
 - ▶ Bubble sort
 - ▶ Shell Sort
 - ▶ Merge Sort
 - ▶ Heap Sort
 - ▶ Quick Sort
-
- ▶ Lưu ý: Các code ở đây chỉ **mang tính chất minh họa** cho giải thuật
 - ▶ Có nhiều cách diễn đạt và cải tiến thuật toán

Các thuật toán sắp xếp

► <https://www.toptal.com/developers/sorting-algorithms/>

	 Insertion	 Selection	 Bubble	 Shell	 Merge	 Heap	 Quick	 Quick3
 Random								
 Nearly Sorted								
 Reversed								
 Few Unique								

Interchange sort

- ▶ 51 90 26 23 63
- ▶ **26** 90 **51** 23 63 → Thừa vô ích
- ▶ **23** 90 51 **26** 63
- ▶ 23 **51** **90** 26 63
- ▶ 23 **26** 90 **51** 63
- ▶ 23 26 **51** **90** 63 Luôn lặp n^2 lần
- ▶ 23 26 51 **63** **90** Có nhiều hoán vị thừa

```
1. void interchangeSort(int a[], int n)
2. {
3.     for(int i=0; i<n-1; i++)
4.         for(int j=i+1; j<n; j++)
5.             if(a[i]>a[j])
6.                 swap(&a[i], &a[j]);
7. }
```

Insertion sort

- ▶ for $i = 2:n$,
 - ▶ for ($k = i; k > 1$ and $a[k] < a[k-1]; k--$)
 - ▶ swap $a[k, k-1]$
 - ▶ → invariant: $a[1..i]$ is sorted
 - ▶ end
-
- ▶ 51 90 26 23 63
 - ▶ **26 51 90** 23 63 Dịch chuyển nhiều phần tử
 - ▶ **23 26 51 90** 63 Dịch chuyển nhiều lần
 - ▶ 23 26 51 **63 90** Luôn lặp n^2 lần

Insertion sort – Minh hoạ

```
1. void insertionSort(int a[], int n)
2. {
3.     int i, key, j;
4.     for (int i = 1; i < n; i++)
5.     {
6.         int temp = a[i];
7.         int j = i-1;
8.
9.         while (j >= 0 && a[j] > temp)
10.        {
11.            a[j+1] = a[j];
12.            j = j-1;
13.        }
14.        a[j+1] = temp;
15.    }
16.}
```

Selection sort

- ▶ for $i = 1:n$,
- ▶ $k = i$
- ▶ for $j = i+1:n$, if $a[j] < a[k]$, $k = j$
- ▶ → invariant: $a[k]$ smallest of $a[i..n]$
- ▶ swap $a[i,k]$
- ▶ → invariant: $a[1..i]$ in final position
- ▶ end

Selection sort

- ▶ 51 90 26 23 63
 - ▶ **23** 90 26 **51** 63
 - ▶ 23 **26** **90** 51 63
 - ▶ 23 26 **51** **90** 63
 - ▶ 23 26 51 **63** **90**
-
- ▶ → Loại những hoán vị thừa ở thuật toán cơ bản

Selection sort – Minh hoạ

```
1. void selectionSort(int a[], int n)
2. {
3.     for(int i=0; i<n; i++)
4.     {
5.         int k = i;
6.         for(int j=i+1; j<n; j++)
7.             if (a[j]<a[k])
8.                 k=j;
9.         if (i!=k)
10.            swap(&a[i], &a[k]);
11.     }
12. }
```


Bubble Sort

- ▶ for $i = 1:n$,
- ▶ swapped = false
- ▶ for $j = n:i+1$,
- ▶ if $a[j] < a[j-1]$,
- ▶ swap $a[j,j-1]$
- ▶ swapped = true
- ▶ → invariant: $a[1..i]$ in final position
- ▶ break if not swapped
- ▶ end

Bubble Sort

- ▶ 51 90 26 23 63
- ▶ 51 90 **23 26** 63
- ▶ 51 **23 90** 26 63
- ▶ **23 51** 90 26 63
- ▶ 23 51 **26 90** 63
- ▶ 23 **26 51** 90 63
- ▶ 23 26 51 **63 90**
- ▶ → Nhiều lần hoán vị

Bubble Sort – Minh hoạ

```
1. void bubbleSort(int a[], int n)
2. {
3.     for(int i=0; i<n; i++)
4.     {
5.         bool swapped = false;
6.         for(int j=n-1; j>i; j--)
7.             if(a[j] < a[j-1])
8.             {
9.                 swap(&a[j], &a[j-1]);
10.                swapped = true;
11.            }
12.        if(!swapped) break;
13.    }
14.}
```

Shell Sort

- ▶ Tương tự như insertion sort
- ▶ Insertion sort
 - ▶ Khi hoán đổi di chuyển **từng phần tử liền kề**
- ▶ Shell sort
 - ▶ Khi hoán đổi di chuyển các phần tử cách nhau khoảng cách gap
 - ▶ Sắp xếp mảng con gap
 - ▶ Các phần tử cách nhau một khoảng gap
 - ▶ gap có thể bắt đầu từ $n/2$, giảm dần về 1

Shell Sort – Ví dụ

▶ 51 90 26 23 63

▶ **26** 90 **51** 23 63

→ gap = 2

▶ 26 **23** 51 **90** 63

→ gap = 2

▶ **26** 23 **51** 90 **63**

→ gap = 2

▶ **23** **26** 51 90 63

→ gap = 1

▶ 23 26 51 **63** **90**

→ gap = 1

Shell Sort – Ví dụ khác

- ▶ Sau một gap = 5: Các phần tử có khoảng cách là 5 được sắp xếp

8	6	0	3	2	7	5	1	9	4
---	---	---	---	---	---	---	---	---	---

After a gap = 5:

7	5	0	3	2	8	6	1	9	4
---	---	---	---	---	---	---	---	---	---

After a gap = 2:

0	1	2	3	6	4	7	5	9	8
---	---	---	---	---	---	---	---	---	---

After a gap = 1:

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

Shell Sort – Minh hoạ

```
1. void shellSort(int a[], int n)
2. {
3.     for (int gap = n/2; gap > 0; gap /= 2)
4.     {
5.         for (int i = gap; i < n; i += 1)
6.         {
7.             int temp = arr[i];
8.             int j;
9.             for(j=i; j>=gap && a[j-gap]>temp; j-=gap)
10.                 a[j] = a[j - gap];
11.             a[j] = temp;
12.         }
13.     }
14. }
```

Shell Sort – Biến thể khác

- ▶ $gap = 1$
- ▶ while $gap < n$, $gap = 3 * gap + 1$
- ▶ while $gap > 0$,
- ▶ $gap = gap / 3$
- ▶ for $k = 1:gap$, insertion sort $a[k:gap:n]$
- ▶ → invariant: each gap -sub-array is sorted
- ▶ end

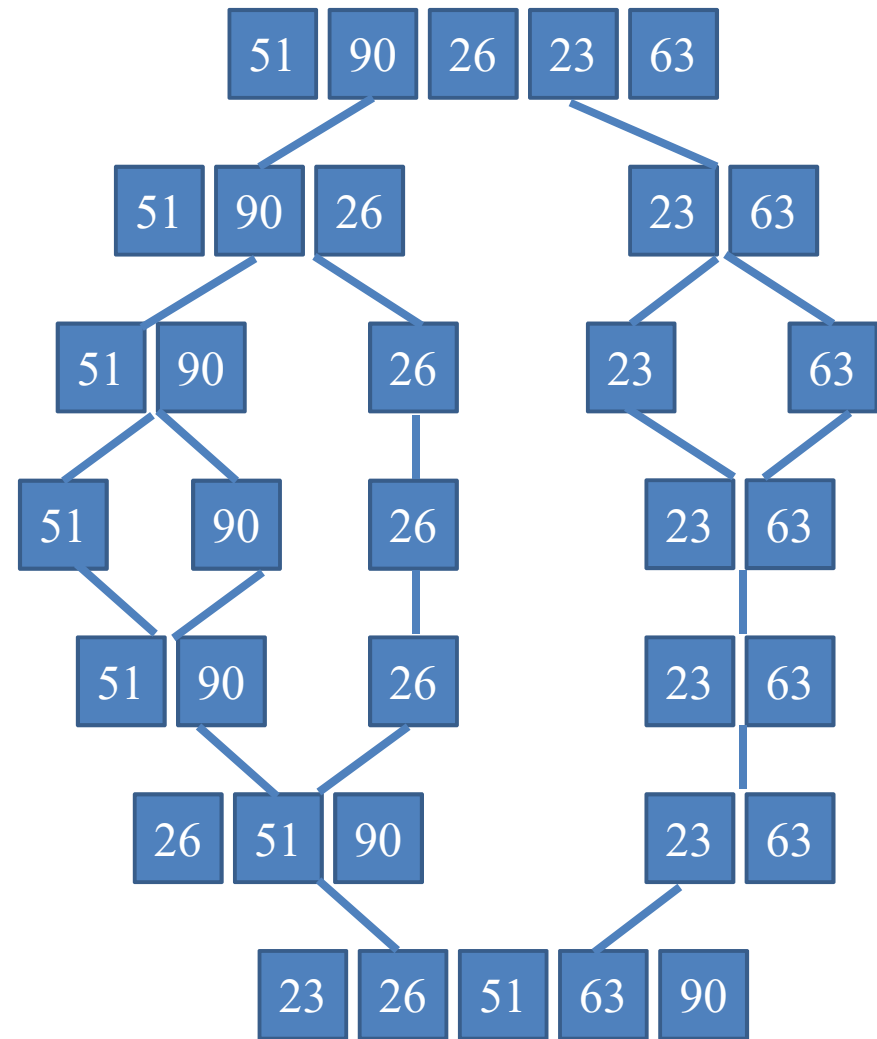
Shell Sort – Minh hoạ

```
1. void shellSort(int a[], int n)
2. {
3.     int gap=1;
4.     while(gap<n) gap=3*gap+1;
5.     while(gap>0)
6.     {
7.         gap=gap/3;
8.         for(int k=gap; k<n; k++)
9.         {
10.            int temp = a[k];
11.            int j;
12.            for(j=k; j>=gap && a[j-gap]>temp; j-=gap)
13.                a[j] = a[j-gap];
14.            a[j] = temp;
15.        }
16.    }
17.}
```

Merge Sort

► Chia để trị

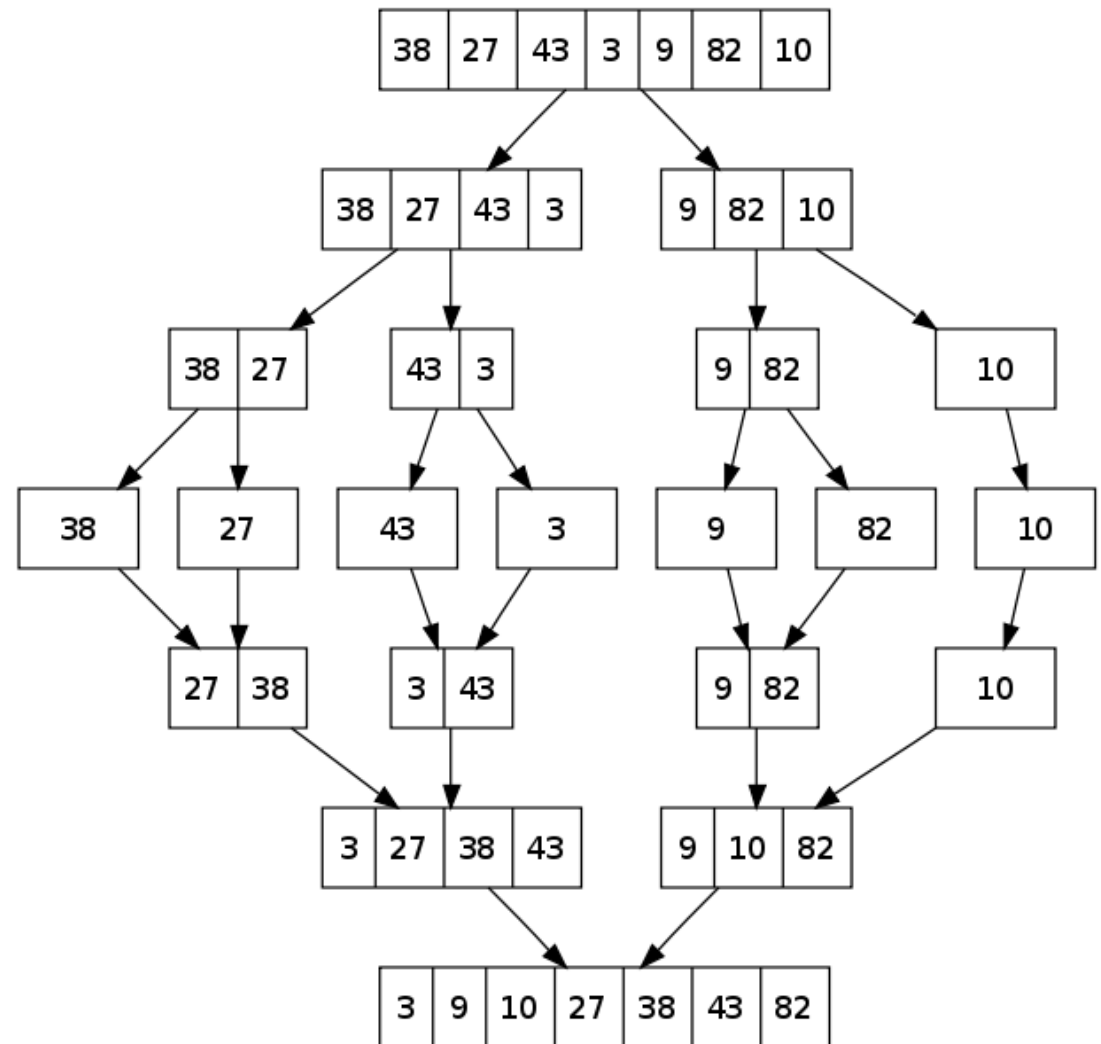
- Chia thành hai mảng con
- Tiếp tục chia đôi các mảng con như cây nhị phân
- Trộn các mảng con và sắp xếp tăng dần



Merge sort – Ví dụ khác

▶ Trộn

- ▶ Trộn hai mảng đồng thời sắp xếp tăng dần



Merge Sort – Algorithm

- ▶ # split in half
- ▶ $m = n / 2$
- ▶ # recursive sorts
- ▶ sort $a[1..m]$
- ▶ sort $a[m+1..n]$
- ▶ # merge sorted sub-arrays using temp array
- ▶ $b = \text{copy of } a[1..m]$
- ▶ $i = 1, j = m+1, k = 1$
- ▶ while $i \leq m$ and $j \leq n$,
 - ▶ $a[k++] = (a[j] < b[i]) ? a[j++] : b[i++]$
 - ▶ \rightarrow invariant: $a[1..k]$ in final position
- ▶ while $i \leq m$,
 - ▶ $a[k++] = b[i++]$
 - ▶ \rightarrow invariant: $a[1..k]$ in final position

Merge Sort – Minh hoạ

```
1. void mergeSort(int a[], int left, int right)
2. {
3.     if (left < right)
4.     {
5.         int mid = (left+right)/2;
6.         // Sắp xếp hai nửa trước và sau
7.         mergeSort(a, left, mid);
8.         mergeSort(a, mid+1, right);
9.         // Trộn lại
10.        merge(a, left, mid, right);
11.    }
12.}
```

Merge Sort – Minh họa

```
1. void merge(int a[], int left, int mid, int right)
2. {
3.     int i, j, k;
4.     int b[mid+1];
5.     for (i = left; i <= mid; i++)
6.         b[i] = a[i];

7.     i = left; // Initial index of first subarray
8.     j = mid+1; // Initial index of second subarray
9.     k = left; // Initial index of merged subarray
10.    while (i <= mid && j <= right)
11.        a[k++] = (b[i]<a[j])?b[i++]:a[j++];

12.    while (i <= mid)
13.        a[k++] = b[i++];
14.}
```

Heap Sort

- ▶ Cấu trúc binary Heap

- ▶ Cây nhị phân đầy đủ
- ▶ Giả sử một nút cha là i

- ▶ Nút con bên trái là $2*i + 1$
- ▶ Nút con bên phải là $2*i + 2$

- ▶ Nút cha (parent node)

- ▶ Lớn hơn hai nút con (max heap)
- ▶ Nhỏ hơn hai nút con (min heap)

- ▶ Heap có thể được biểu diễn

- ▶ Cây nhị phân
- ▶ Mảng

6 5 3 1 8 7 2 4

Heap Sort – Algorithm

- ▶ Giải thuật Heap sort
 - ▶ B1: Xây dựng max heap
 - ▶ B2: Phần tử lớn nhất ở gốc
 - ▶ B3: Thay thế gốc bằng phần tử cuối cùng
 - ▶ B4: Giảm kích thước heap
 - ▶ B5: Xây dựng lại max heap
 - ▶ B6: Lặp lại bước 2 cho đến khi hết mảng
- ▶ Vẽ cây nhị phân cho các dãy số bên

- ▶ 51 90 26 23 63
- ▶ **90** 51 26 23 63
- ▶ 90 **63** 26 23 **51**
- ▶ **51** 63 26 23 **90**
- ▶ **63** 51 26 23 90
- ▶ **23** 51 26 **63** 90
- ▶ **51** **23** 26 63 90
- ▶ **26** 23 **51** 63 90
- ▶ **23** **26** 51 63 90

Heap sort – Ví dụ khác

10	4	8	5	12	2	6	11	3	9	7	1
----	---	---	---	----	---	---	----	---	---	---	---

Heap Sort – Minh hoạ

```
1. void heapSort(int a[], int n)
2. {
3.     // Build heap (rearrange array)
4.     for (int i = n / 2 - 1; i >= 0; i--)
5.         heapify(a, n, i);
6.
7.     // One by one extract an element from heap
8.     for (int i=n-1; i>=0; i--)
9.     {
10.        // Move current root to end
11.        swap(&a[0], &a[i]);
12.        // call max heapify on the reduced heap
13.        heapify(a, i, 0);
14.    }
15.}
```

Heap Sort – Minh hoạ

```
1. void heapify(int a[], int n, int i)
2. {
3.     int largest = i; // Initialize largest as root
4.     int l = 2*i + 1; // left = 2*i + 1
5.     int r = 2*i + 2; // right = 2*i + 2
6.     // If left child is larger than root
7.     if (l < n && a[l] > arr[largest])
8.         largest = l;
9.     // If right child is larger than largest so far
10.    if (r < n && a[r] > arr[largest])
11.        largest = r;
12.    // If largest is not root
13.    if (largest != i)
14.    {
15.        swap(&a[i], &a[largest]);
16.        // Recursively heapify the affected sub-tree
17.        heapify(a, n, largest);
18.    }
19.}
```

Quick Sort

- ▶ Thuật toán chia để trị (Divide and Conquer)
- ▶ Chọn một phần tử trục (pivot)
 - ▶ Ngẫu nhiên, đầu, giữa hoặc cuối
- ▶ Phân vùng danh sách (partition)
 - ▶ Tìm vị trí chính xác của phần tử trục
 - ▶ Các phần tử nhỏ hơn pivot nằm phía trước
 - ▶ Các phần tử lớn hơn pivot nằm phía sau
- ▶ Tiếp tục với các danh sách con



<p >p <p <p pivot

Quick Sort – How's it work?

▶ Lấy pivot là điểm phải:

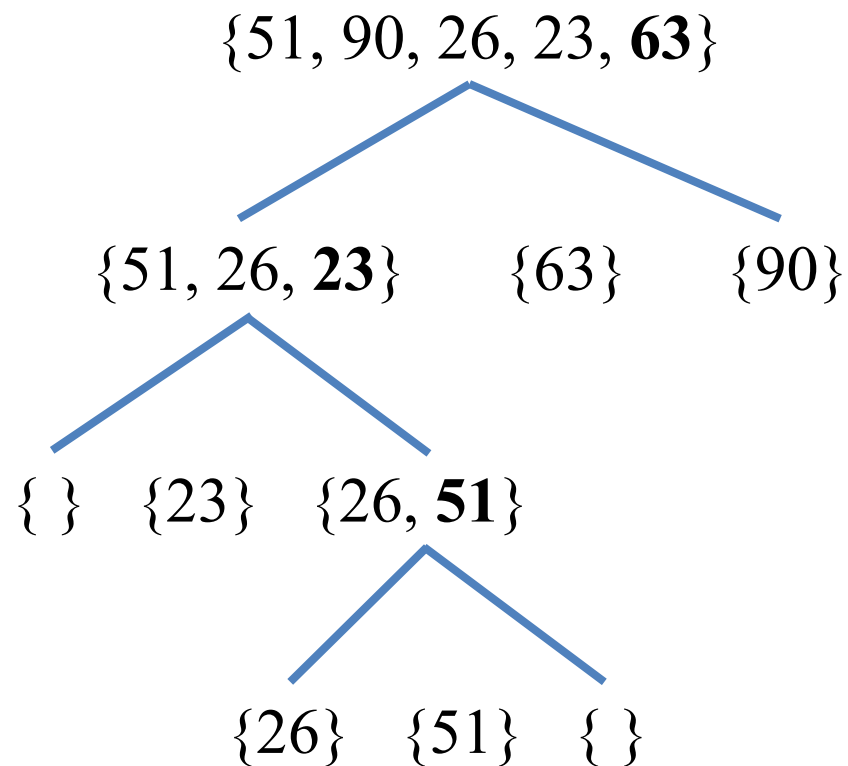
▶ 51 90 26 23 **63**

▶ 51 **26** **90** 23 63

▶ 51 26 **23** **90** 63

▶ 51 26 23 **63** **90**

▶ **23** 26 **51** 63 90



Quick Sort - Algorithm

- ▶ `_# choose pivot_ random`
- ▶ `swap a[1,rand(1,n)]`

- ▶ `_# 2-way partition_`
- ▶ `k = 1`
- ▶ `for i = 2:n, if a[i] < a[1], swap a[++k,i]`
- ▶ `swap a[1,k]`
- ▶ `_→ invariant: a[1..k-1] < a[k] <= a[k+1..n]_`

- ▶ `_# recursive sorts_`
- ▶ `sort a[1..k-1]`
- ▶ `sort a[k+1,n]`

Quick Sort – Minh hoạ

```
1. void quickSort(int arr[], int low, int high)
2. {
3.     if (low < high)
4.     {
5.         /* pi is partitioning index, arr[p] is now
6.            at right place */
7.         int pi = partition(arr, low, high);
8.
9.         // Separately sort elements before
10.        // partition and after partition
11.        quickSort(arr, low, pi - 1);
12.        quickSort(arr, pi + 1, high);
13.    }
14. }
```

Quick Sort – Minh hoạ

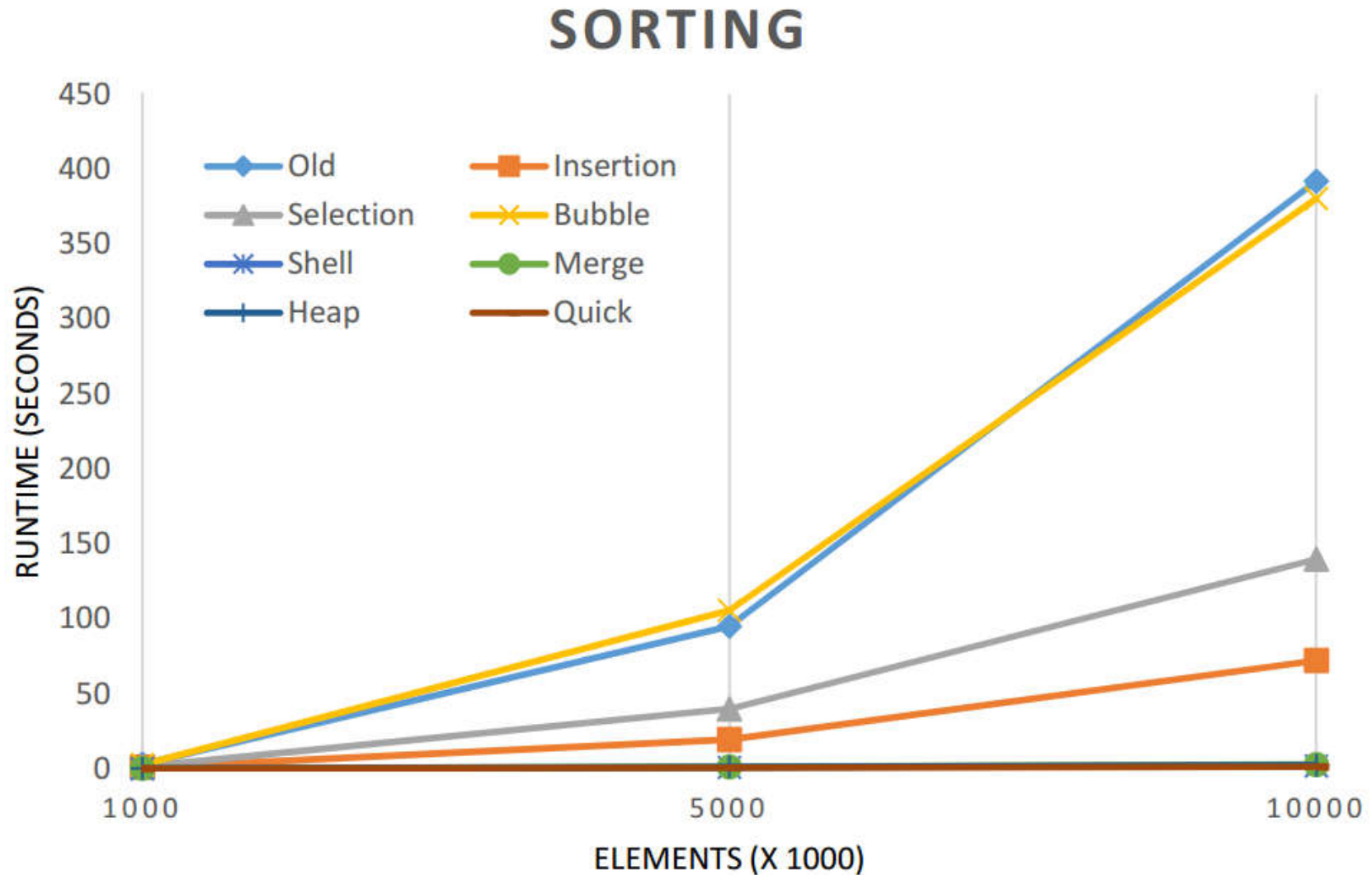
```
1.  int partition (int a[], int low, int high)
2.  {
3.      int pivot = a[high];
4.      int i = (low - 1);  // Index of smaller element
5.
6.      for (int j = low; j < high; j++)
7.      {
8.          // If current element is smaller than or
9.          // equal to pivot
10.         if (a[j] <= pivot)
11.         {
12.             i++; // increment index of smaller element
13.             swap(&a[i], &a[j]);
14.         }
15.     }
16.     swap(&a[i + 1], &a[high]);
17.     return (i + 1);
18. }
```


So sánh thực nghiệm

- ▶ Thực hiện 1000 phép lặp cho mỗi hàm
- ▶ Giá trị của mảng được phát sinh ngẫu nhiên
 - ▶ `b[i] = rand();`
- ▶ Đo thời gian thực hiện của mỗi hàm

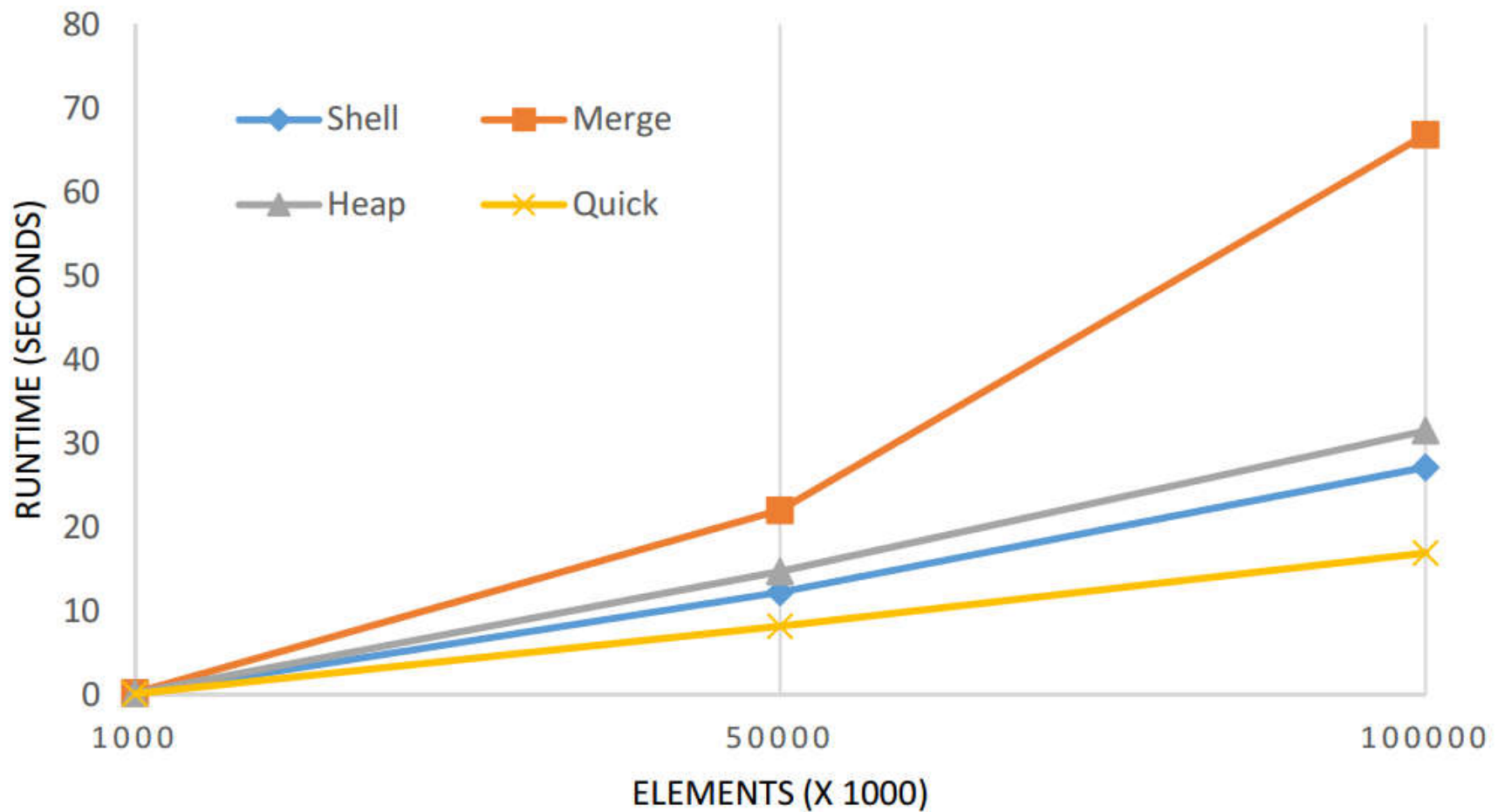
```
1. t = clock();
2. for(int i=0; i<LOOP; i++)
3. {
4.     copy(a, b, n); // b là mảng phát sinh ngẫu nhiên
5.     Sort(a, n);
6. }
7. t=clock()-t;      // loopTime là thời gian lặp và copy
8. printf("Sorting time: %.2fs\n", (t-loopTime)
    /(float)CLOCKS_PER_SEC);
```

Kết quả so sánh



Nhanh nhất?

SORTING



Độ phức tạp của thuật toán

Algorithm	Best case	Average case	Worst case
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Shell sort	$O(n)$	$O((n \log(n))^2)$	$O((n \log(n))^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

Bài tập vận dụng

- ▶ Viết chương trình
 - ▶ Phát sinh ngẫu nhiên một mảng
 - ▶ Cài đặt các hàm sắp xếp
 - ▶ Tính số thao tác của mỗi phương pháp
 - ▶ Đánh giá các phương pháp
- ▶ Tìm hiểu hoặc đề xuất phương pháp mới

TRƯỜNG ĐẠI HỌC CÔNG NGHIỆP THÀNH PHỐ HỒ CHÍ MINH



Cấu trúc dữ liệu và giải thuật

Danh sách liên kết

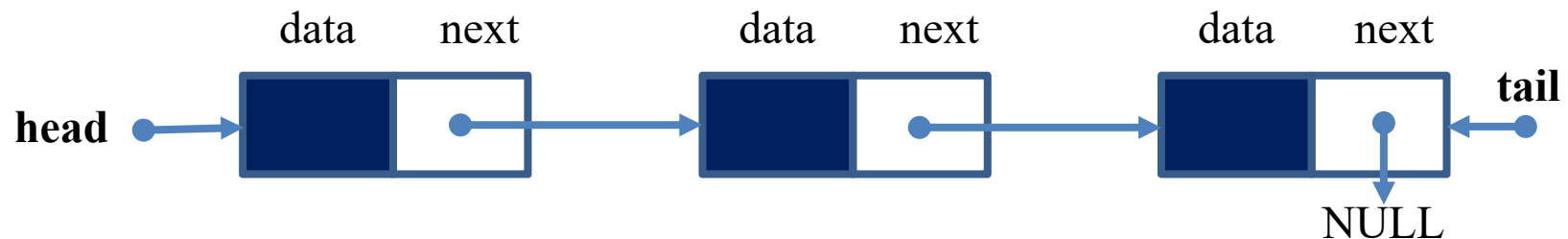
TS. Ngô Hữu Dũng

Dẫn nhập

- ▶ Mảng (array)
 - ▶ Kích thước khó thay đổi
 - ▶ Cần cấp phát trước một vùng nhớ liên tục
 - ▶ Mất nhiều thao tác để chèn/xoá phần tử
 - ▶ Phù hợp với dữ liệu nhỏ, truy xuất nhanh
- ▶ Danh sách liên kết (linked list)
 - ▶ Kích thước thay đổi linh động
 - ▶ Cấp phát bộ nhớ động, không cần vùng nhớ liên tục
 - ▶ Chèn/xoá dễ dàng
 - ▶ Cho phép dữ liệu lớn hơn, cấu trúc linh hoạt

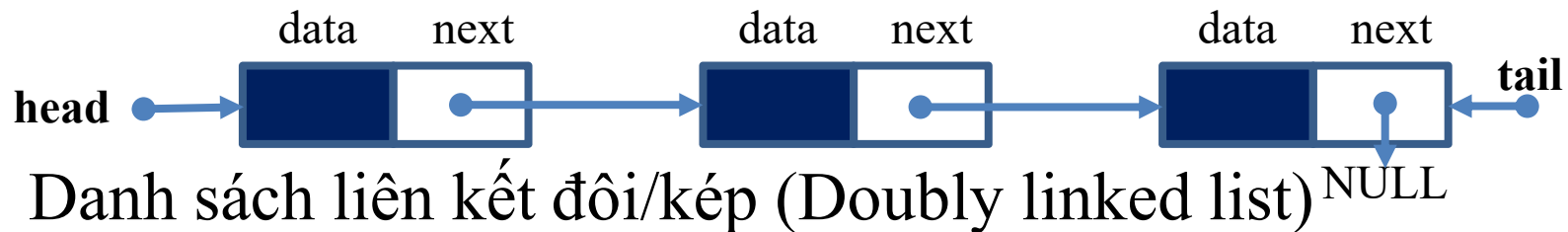
Linked list – Khái niệm

- ▶ Dãy phần tử nối với nhau bởi con trỏ (pointer)
- ▶ Mỗi phần tử là một nút (node)
 - ▶ Phần dữ liệu (int, float, char, struct...)
 - ▶ Phần liên kết (pointer)
- ▶ Con trỏ **head** trỏ vào nút đầu tiên
- ▶ Con trỏ **tail** trỏ vào nút cuối cùng
- ▶ Nút cuối cùng trỏ vào NULL

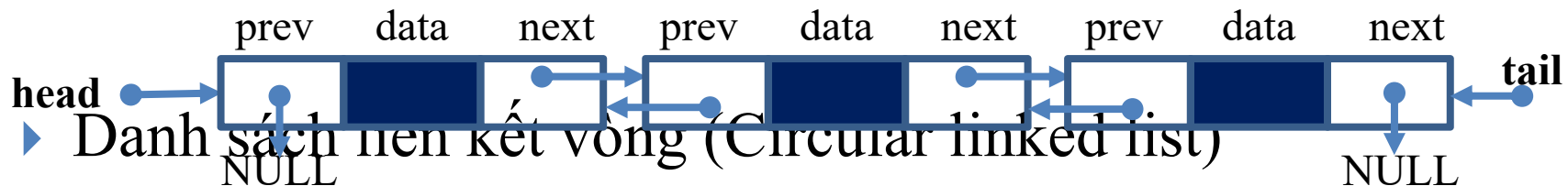


Các loại danh sách liên kết

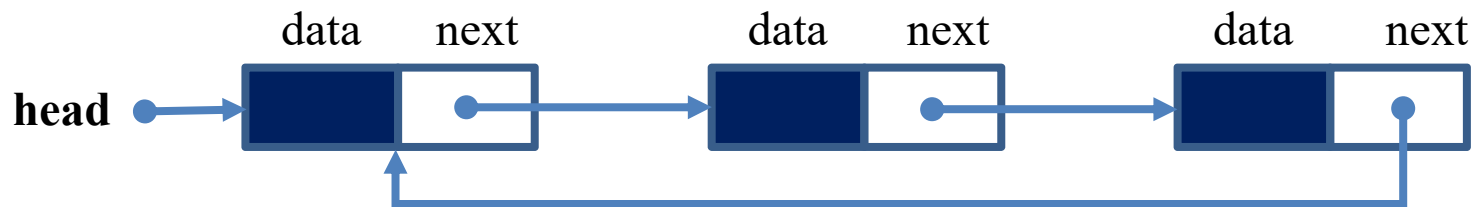
- ▶ Danh sách liên kết đơn (Singly linked list)



- ▶ Danh sách liên kết đôi/kép (Doubly linked list)

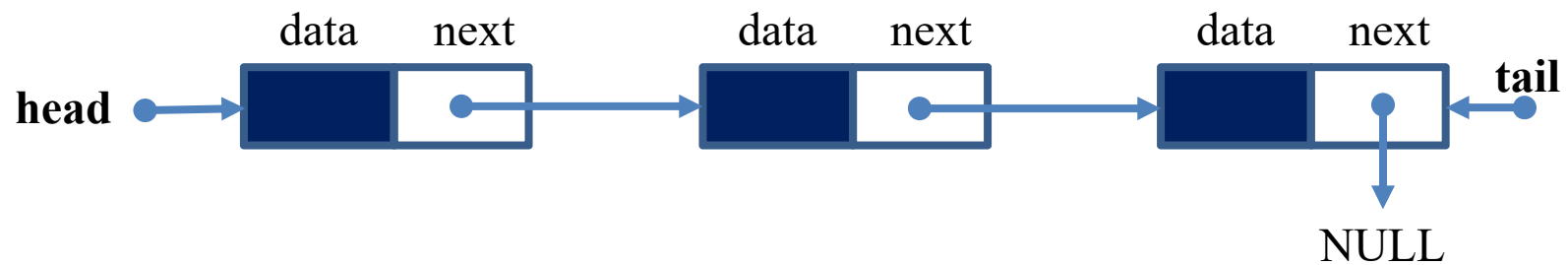


- ▶ Danh sách liên kết vòng (Circular linked list)



Một vài ứng dụng

- ▶ Tổ chức các cấu trúc dữ liệu khác nhau
 - ▶ Stack, queue, tree, graph, hash table...
- ▶ Lưu dấu
 - ▶ Lịch sử truy cập web (history)
 - ▶ Lưu các tác vụ (undo)
- ▶ Quản lý các thành phần trong máy tính
 - ▶ Bộ nhớ, tiến trình, tập tin...
- ▶ Phù hợp với các ứng dụng
 - ▶ Dữ liệu lớn, cấu trúc linh động



Danh sách liên kết đơn

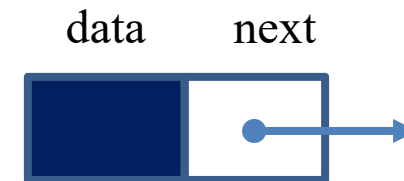
Singly linked list

Singly linked list – Khai báo

- ▶ Khai báo nút kiểu cấu trúc
 - ▶ Phần dữ liệu (int, float, char, struct...)
 - ▶ Phần liên kết (pointer)
- ▶ Khai báo con trỏ *head* và *tail*

```
1. struct Node
2. {
3.     int data;
4.     struct Node *next;
5. };

6. struct Node *head;
7. struct Node *tail;
```



head →
tail →

Định nghĩa kiểu nút

- ▶ Dùng typedef định nghĩa kiểu cấu trúc nút
 - ▶ Có nhiều cách khai báo biến kiểu nút

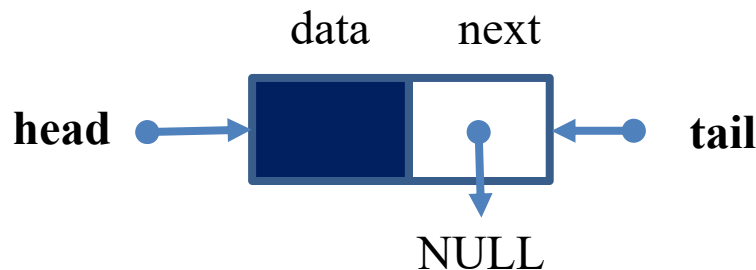
```
1. struct Node
2. {
3.     int data;
4.     struct Node *next;
5. };
6. // Định nghĩa kiểu nút
7. typedef struct Node tNode;

8. tNode *head;
9. struct Node *tail;

10. Node *temp; // C++
```

Kiểu danh sách

- ▶ Khai báo kiểu danh sách
 - ▶ Con trỏ head và tail
- ▶ Phù hợp với bài toán cần dùng nhiều danh sách
- ▶ Truy xuất?
 - ▶ `ds1.tail → data` (?)
 - ▶ `ds1.head → next` (?)



```
1. // Kiểu nút
2. struct Node
3. {
4.     int data;
5.     struct Node *next;
6. };

7. // Kiểu danh sách
8. struct List
9. {
10.    struct Node *head;
11.    struct Node *tail;
12. };

13. // Biến danh sách
14. struct List ds1, ds2;
```

Khai báo – Ví dụ

- ▶ Danh sách sinh viên
 - ▶ Cấu trúc sinh viên
 - ▶ ID, ten...
 - ▶ Cấu trúc nút
 - ▶ Dữ liệu: Kiểu cấu trúc sinh viên
 - ▶ Liên kết: Con trỏ kiểu nút
- ▶ Truy xuất?
 - ▶ `ds.tail` → `data.ID`
 - ▶ `ds.head` → `next` → `data.ID`

```
1. struct SV{
2.     int ID;
3.     char ten[50];
4.     bool gioiTinh;
5.     float diem;
6. };
7. struct Node{
8.     struct SV data;
9.     struct Node *next;
10. };
11. struct List{
12.     struct Node *head;
13.     struct Node *tail;
14. };
15. struct List ds;
```

Vận dụng

- ▶ Bài tập: Container tracking
- ▶ Một nhà vận chuyển sở hữu một số lượng container chưa xác định. Mỗi container chứa các thông số như ID, khối lượng hàng đang chứa, tình trạng đang dùng hay không, toạ độ GPS hiện tại (kinh độ, vĩ độ) ví dụ (10.823, 106.629).
- ▶ Hãy **thiết lập cấu trúc dữ liệu** để quản lý số container trên.

Thao tác cơ bản

- ▶ Khởi tạo danh sách, nút mới
- ▶ Thêm phần tử
 - ▶ Vào đầu, vào cuối, chèn vào sau một phần tử
- ▶ Duyệt danh sách
 - ▶ Xuất, trích xuất, đếm, tính toán
- ▶ Tìm kiếm
 - ▶ Min, max, giá trị X
- ▶ Xoá phần tử
 - ▶ Ở đầu, ở cuối, ở giữa
- ▶ Sắp xếp

Bài toán đặt ra

- ▶ Danh sách các số nguyên
 - ▶ Cấu trúc dữ liệu danh sách liên kết đơn
 - ▶ Các thao tác cơ bản trên danh sách liên kết đơn
 - ▶ Menu thực hiện

```
1. // Kiểu nút
2. struct Node
3. {
4.     int data;
5.     struct Node *next;
6. };
7. typedef struct Node tNode;
8. // Kiểu danh sách
9. struct List
10. {
11.     tNode *head;
12.     tNode *tail;
13. };
14. typedef struct List tList;
15. // Biến danh sách
16. tList ds;
```

Một số hàm tạo, thêm và chèn phần tử

1. `// Initiate a new list`
2. `void init(tList *);`

3. `// Make a new node`
4. `tNode* makeNode();`

5. `// Add a new node to the head of list`
6. `void addHead(tList *);`

7. `// Add a new node to the tail of list`
8. `void addTail(tList *);`

9. `// Insert a node after a given node`
10. `void insertAfter(tList *, tNode *, tNode *);`

Khởi tạo danh sách

- ▶ Danh sách ban đầu là danh sách rỗng
 - ▶ head và tail trỏ vào NULL

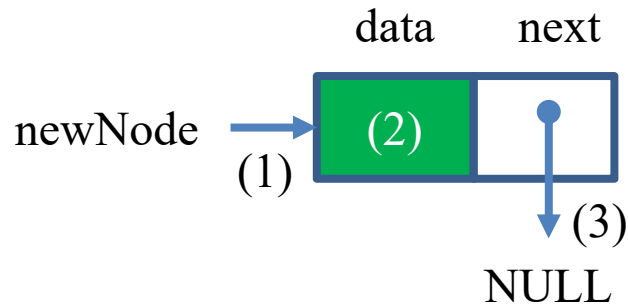
head  **tail**

- ▶ Chú ý cách dùng con trỏ (1) kiểu cấu trúc (2)

```
1. void init(tList *list)
2. {
3.     list->head = NULL; // (1)
4.     list->tail = NULL; // (2)
5. }
6. // Gọi hàm: init(&list);
```

```
7. //Hoặc dùng tham chiếu
8. void init(tList &list)
9. {
10.    list.head = NULL; // (1)
11.    list.tail = NULL; // (2)
12.}
13.// Gọi hàm: init(list);
```

Tạo một nút mới



```
1. tNode* makeNode ()
2. {
3.     tNode *newNode;
4.     newNode = (tNode*)malloc(sizeof(tNode)) ;
5.     //Or: newNode = new tNode;                (1)
6.     printf("Nhap du lieu: ");
7.     scanf("%d", &newNode->data) ;             // (2)
8.     newNode->next = NULL;                       // (3)
9.     return newNode;
10.}
11.// How to call? tNode *newNode = makeNode() ;
```

Tạo một nút mới – Dùng kiểu pointer

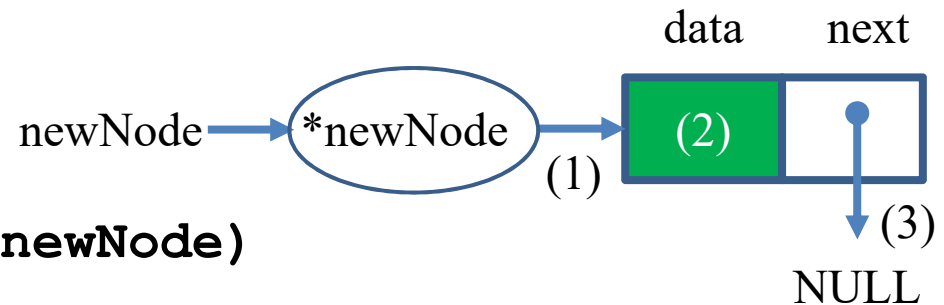
► Để thay đổi giá trị của một con trỏ

► Dùng con trỏ của con trỏ

```
1. // Make a new node
2. void makeNode (tNode **newNode)
3. {
4.     *newNode= (tNode*) malloc (sizeof (tNode)) ; // (1)

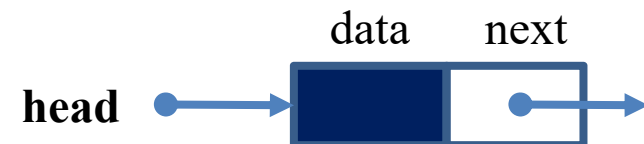
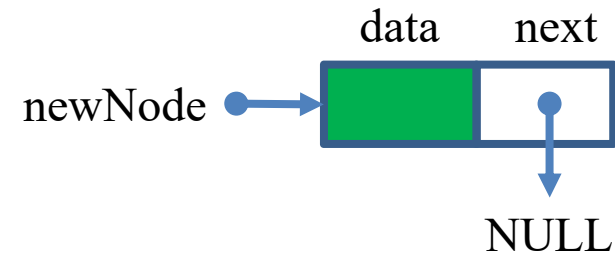
5.     printf("Input data: ");
6.     scanf ("%d", &(*newNode)->data) ;           // (2)

7.     (*newNode)->next = NULL;                     // (3)
8. }
9. // How to call it? tNode *node;
10. // makeNode (&node) ;
```



Thêm nút mới vào đầu danh sách

- ▶ Tạo nút mới
 - ▶ Tạo thế nào?
- ▶ Thêm vào đầu danh sách
 - ▶ Danh sách đang rỗng?
 - ▶ Kiểm tra điều kiện rỗng?
 - ▶ Thêm nút thế nào?
 - ▶ Danh sách không rỗng?
 - ▶ Kiểm tra điều kiện?
 - ▶ Thêm nút thế nào?



Thêm nút mới vào đầu danh sách

► Tạo nút mới

1. Gọi hàm makeNode()

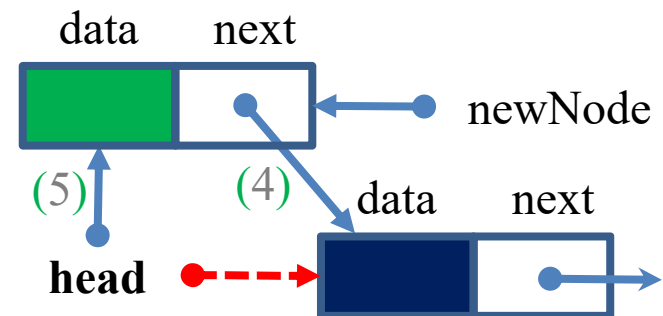
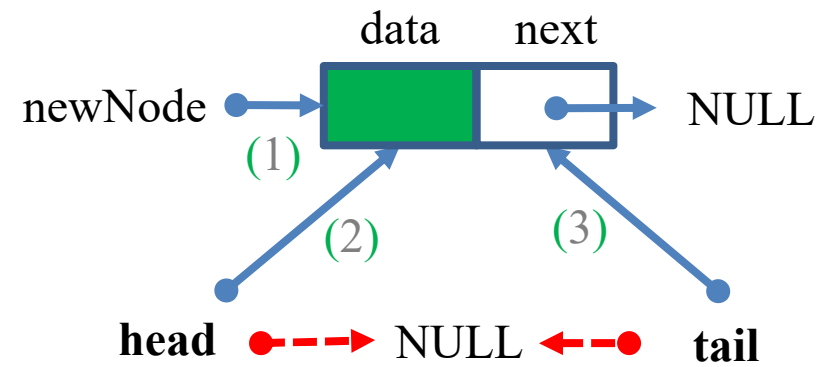
► Thêm vào đầu danh sách

► Danh sách rỗng?

2. head trỏ vào nút mới
3. tail trỏ vào nút mới

► Danh sách không rỗng?

4. next của nút mới trỏ đến nút đầu danh sách
5. head trỏ vào nút mới



Thêm nút mới vào đầu danh sách

```
1. // Add a new node into the head of list
2. void addHead(tList *list)
3. {
4.     // Make a new node
5.     tNode *newNode = makeNode(); // (1)

6.     if(list->head == NULL) {           // List is empty
7.         list->head = newNode;           // (2)
8.         list->tail = newNode;           // (3)
9.     }else{                             // not empty
10.        newNode->next = list->head;      // (4)
11.        list->head = newNode;            // (5)
12.    }
13.}
```

Thêm nút mới vào **cuối** danh sách

▶ Tạo nút mới

1. Gọi hàm makeNode()

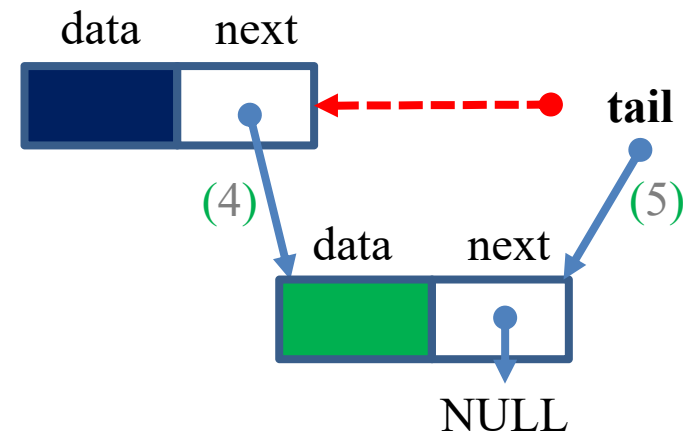
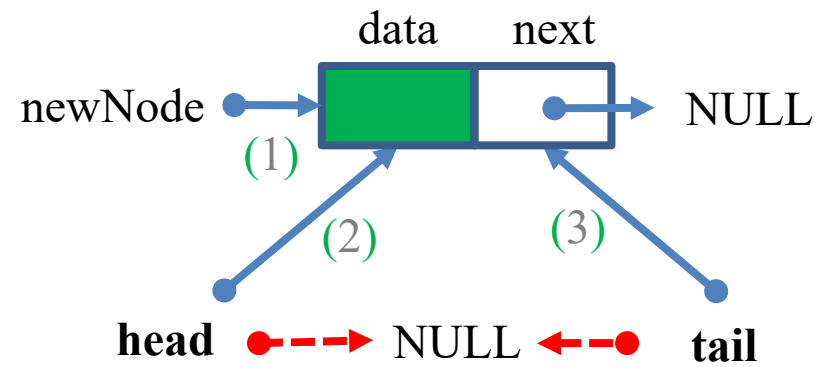
▶ Thêm vào cuối danh sách

▶ Danh sách rỗng

2. head trỏ vào nút mới
3. tail trỏ vào nút mới

▶ Danh sách không rỗng?

4. next của tail trỏ vào nút mới
5. tail trỏ vào nút mới

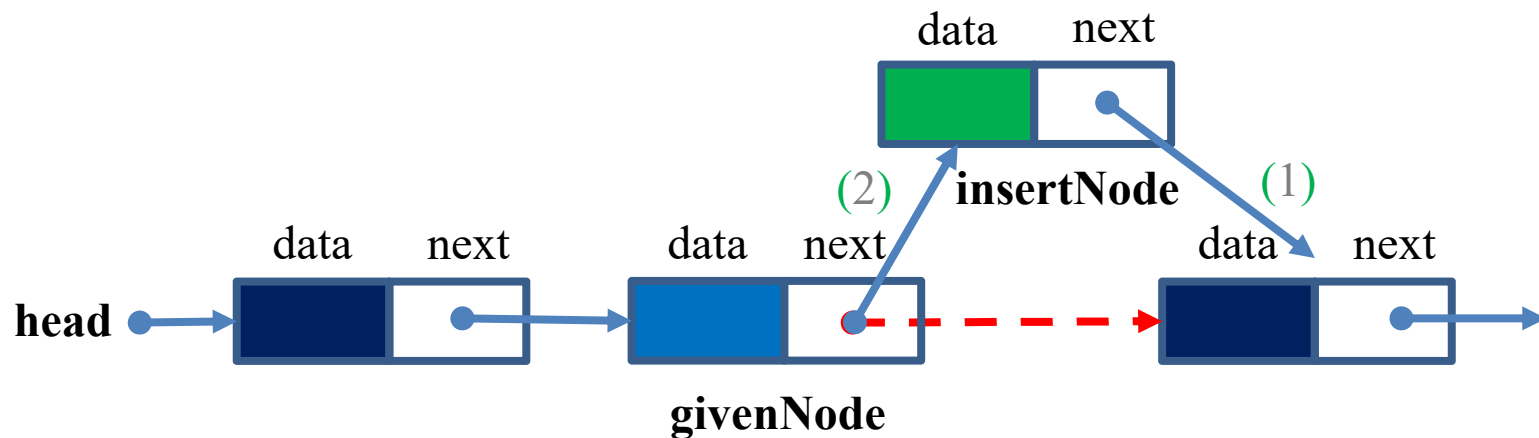


Thêm nút mới vào **cuối** danh sách

```
1. // Add a new node into the tail of list
2. void addTail(tList *list)
3. {
4.     // Make a new node
5.     tNode *newNode = makeNode();           // (1)
6.
7.     if(!list->head) {                       // List is empty
8.         list->head = newNode;               // (2)
9.         list->tail = newNode;               // (3)
10.    }else{                                   // Not empty
11.        list->tail->next = newNode;          // (4)
12.        list->tail = newNode;               // (5)
13.    }
14.}
```

Chèn một nút vào **sau** một nút

- ▶ Chèn nút insertNode vào sau givenNode?
 1. $\text{insertNode} \rightarrow \text{next} = \text{givenNode} \rightarrow \text{next}$
 2. $\text{givenNode} \rightarrow \text{next} = \text{insertNode}$
- ▶ Trường hợp givenNode rỗng?
- ▶ Trường hợp givenNode là nút cuối?



Chèn một nút vào sau một nút

```
1. // Insert insNode after givenNode
2. void insertAfter(tList *list, tNode *givenNode,
   tNode *insertNode)
3. {
4.     // Add after a NULL? return
5.     if(givenNode==NULL)
6.         return;

7.     insertNode->next = givenNode->next; // (1)
8.     givenNode->next = insertNode;      // (2)

9.     // Add after the tail? update the tail
10.    if(givenNode==list->tail)
11.        list->tail = insertNode;
12.}
```

Vận dụng

- ▶ Bài tập: Container tracking
- ▶ Một nhà vận chuyển sở hữu một số lượng container chưa xác định. Mỗi container chứa các thông số như ID, khối lượng hàng đang chứa, tình trạng đang dùng hay không, toạ độ GPS.
- ▶ Hãy thiết lập thao tác **nhập container mới**.

Một số hàm duyệt danh sách

1. `// Print all list`
2. `void output(tList);`

3. `// Count a list`
4. `int count(tList);`

5. `// Calculate a list`
6. `int total(tList);`

7. `// Search an item on a list`
8. `tNode *search(tList, int);`

9. `// Find the max node on a list`
10. `tNode* maxNode(tList);`

Duyệt danh sách

- ▶ Phần tử bắt đầu: `tNode *node = list.head;`
- ▶ Phần tử tiếp theo: `node = node→next`
- ▶ Phần tử kết thúc: `NULL`

```
1. tNode *node = list.head;
2. while (node!=NULL)
3. {
4.     // Thao tác xử lý
5.     node = node->next;           // Đến nút kế tiếp
6. }

7. // Hoặc:
8. for (node = list.head; node; node = node->next)
9.     // Thao tác xử lý
```


Xuất toàn bộ danh sách

```
1. // Print all list
2. void output(tList list)
3. {
4.     tNode *node = list.head;
5.     printf("List: ");
6.     if(!list.head)        // Empty list
7.     {
8.         printf("Empty.\n");
9.         return;
10.    }
11.    while (node)
12.    {
13.        printf("%d ", node->data);
14.        node = node->next;
15.    }
16.    printf("\n");
17.}
```

Đếm danh sách

```
1. // Count elements of list
2. int count(tList list)
3. {
4.     tNode *node;
5.     int dem = 0;
6.     for(node = list.head; node; node = node->next)
7.         dem++;
8.     return dem;
9. }
```

Tương tự, hãy viết các hàm sau:

```
int total(tList);
tNode *search(tList, int);
tNode* maxNode(tList);
```

Tính tổng

```
1. // Calculate the sum of list
2. int total(tList list)
3. {
4.     tNode *node = list.head;
5.     int tong = 0;
6.     while(node)
7.     {
8.         tong += node->data;
9.         node = node->next;
10.    }
11.    return tong;
12.}
```

Tìm kiếm

```
1. // Search a node
2. tNode *search(tList list, int x)
3. {
4.     tNode *node = list.head;
5.     while(node)
6.     {
7.         if(node->data == x)
8.             return node;
9.         node = node->next;
10.    }
11.    return NULL;
12.}
```

Tìm max

```
1. // Find the max node on list
2. tNode* maxNode(tList list)
3. {
4.     tNode *node = list.head;
5.     tNode *max = node;
6.     while(node)
7.     {
8.         if(node->data > max->data)
9.             max = node;
10.        node = node->next;
11.    }
12.    return max;
13.}
```

Vận dụng

- ▶ Bài tập: Container tracking

- ▶ Bổ sung các thao tác báo cáo
 - ▶ Xuất thông tin các container
 - ▶ Liệt kê các container đang dùng
 - ▶ Đếm số lượng container rồi
 - ▶ Tính tổng khối lượng hàng hoá của tất cả các container.
 - ▶ Tìm địa chỉ GPS của một container (nhập ID)
 - ▶ Cập nhật thông tin của một container

Một số hàm xoá node

1. `// Delete the head node`
2. `void delHead(tList *);`

3. `// Delete the node after a given node`
4. `void delAfter(tList *, tNode *);`

5. `// Delete the tail node`
6. `void delTail(tList *);`

7. `// Delete a given node`
8. `void delGivenNode(tList *, tNode *);`

9. `// Remove all list`
10. `void removeList(tList *);`

Xoá nút đầu danh sách

- ▶ Danh sách rỗng!?

- 1. Kết thúc

- ▶ Thay đổi liên kết

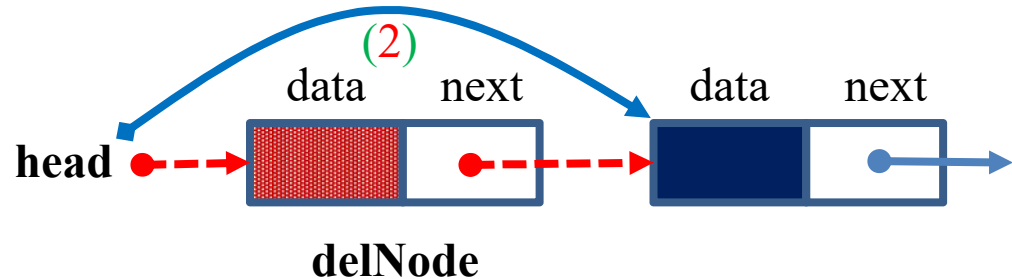
- 2. `head = delNode → next`

- ▶ Nút cần xoá là nút cuối cùng?

- 3. Trở thành danh sách rỗng, cập nhật tail

- ▶ Giải phóng bộ nhớ

- 4. `free()` hoặc `delete`



Xoá nút đầu danh sách

```
1. // Delete the head node
2. void delHead(tList *list)
3. {
4.     tNode *delNode = list->head;
5.     if (delNode==NULL)           // Empty list
6.         return;                 // (1)

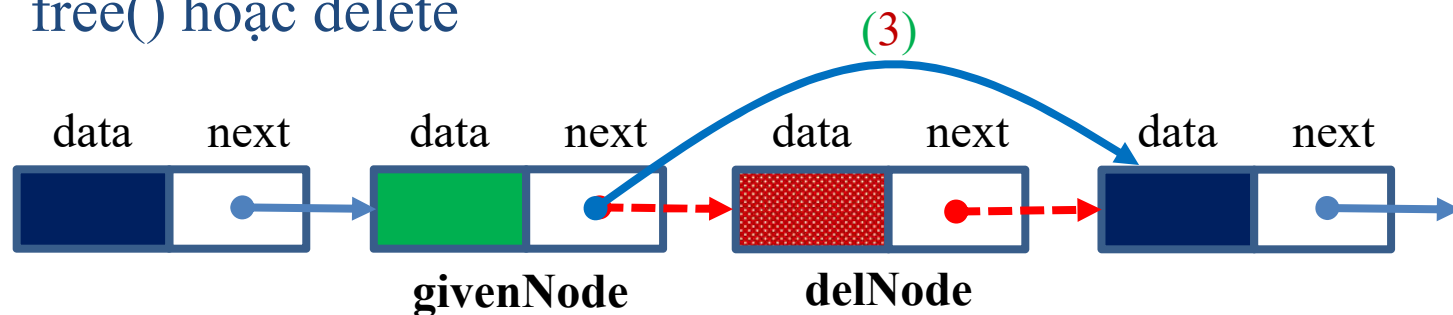
7.     list->head = delNode->next; // (2)

8.     if (list->head == NULL)      // Become empty
9.         list->tail = NULL;      // (3)

10.    free (delNode) ;            // (4)
11. }
```

Xoá nút **sau** nút cho trước

- ▶ Nút cho trước rỗng hoặc nút cần xoá rỗng!?
 1. Kết thúc
- ▶ Thay đổi liên kết
 2. $\text{delNode} = \text{givenNode} \rightarrow \text{next}$
 3. $\text{givenNode} \rightarrow \text{next} = \text{delNode} \rightarrow \text{next}$
- ▶ Nút cần xoá là nút cuối cùng?
 4. Cập nhật tail
- ▶ Giải phóng bộ nhớ
 5. $\text{free}()$ hoặc delete



Xoá nút sau nút cho trước

```
1. // Delete the node after a given node
2. void delAfter(tList *list, tNode *givenNode)
3. {
4.     tNode *delNode;
5.     if(givenNode==NULL || givenNode->next == NULL)
6.         return; // (1)

7.     delNode = givenNode->next; // (2)
8.     givenNode->next = delNode->next; // (3)

9.     if(delNode==list->tail)
10.         list->tail = givenNode; // (4)
11.     free(delNode); // (5)
12. }
```

Xoá nút **cuối** danh sách

- ▶ Danh sách rỗng?

1. Kết thúc

- ▶ Danh sách chỉ có một nút?

2. Danh sách trở thành rỗng

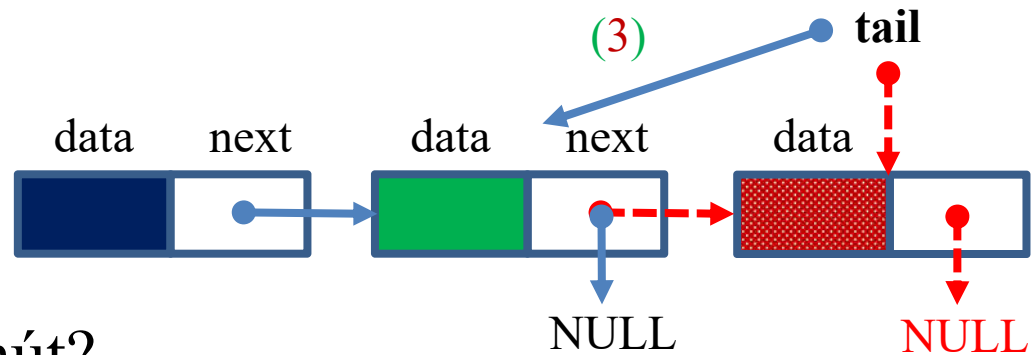
- ▶ Ngược lại, danh sách có nhiều nút?

3. Tìm nút áp cuối (trước nút cuối)

4. Nút áp cuối trở thành nút cuối cùng

- ▶ Giải phóng bộ nhớ

5. free() hoặc delete



Xoá nút cuối danh sách

```
1. // Delete the tail node
2. void delTail(tList *list){
3.     tNode *delNode = list->tail;
4.     if(delNode==NULL)                // (1)
5.         return;
6.     tNode *i = list->head;
7.     if(i==delNode) {                  // (2)
8.         list->head=NULL;
9.         list->tail=NULL;
10.    }else{
11.        while(i->next!=delNode) // (3)
12.            i = i->next;
13.        i->next = NULL;             // (4)
14.        list->tail = i;             // (4)
15.    }
16.    free(delNode);                  // (5)
17.}
```

Xoá một nút bất kỳ cho trước

- ▶ Xoá một nút bất kỳ cho trước
- ▶ Có thể vận dụng các hàm đã biết
- ▶ Ví dụ:
 - ▶ Nút cần xoá là head?
 - ▶ Gọi delHead
 - ▶ Nút cần xoá là tail?
 - ▶ Gọi delTail
 - ▶ Tìm nút trước nút cần xoá
 - ▶ Gọi delAfter

Xoá một nút bất kỳ cho trước

```
1. // Delete a given node
2. void delGivenNode(tList *list, tNode *delNode)
3. {
4.     if(delNode == list->head)
5.         delHead(list);
6.     else if(delNode == list->tail)
7.         delTail(list);
8.     else{
9.         tNode *tempNode = list->head;
10.        while(tempNode && tempNode->next!=delNode)
11.            tempNode = tempNode->next;
12.        if(tempNode)
13.            delAfter(list, tempNode);
14.    }
15.}
```

Xoá toàn bộ danh sách

```
1. // Remove all list
2. void removeList(tList *list)
3. {
4.     tNode *node;
5.     while(list->head)
6.     {
7.         node=list->head;
8.         list->head = node->next;
9.         free(node) ;
10.    }
11.    list->tail = NULL;
12.}
```


Xoá toàn bộ danh sách

► Đơn giản hơn

```
1. // Remove all list
2. void removeList(tList *list)
3. {
4.     while(list->head)
5.         delHead(list);
6. }
```

Vận dụng

- ▶ Bài tập: Container tracking
- ▶ Bổ sung thao tác xoá container
 - ▶ Xoá một container bất kỳ (Nhập ID)
 - ▶ Xoá toàn bộ danh sách

Sắp xếp danh sách – Interchange sort

```
1. // Sắp xếp tăng dần
2. void interchangeSort(tList *list)
3. {
4.     tNode *i, *j;
5.     for(i = list->head; i!=list->tail; i=i->next)
6.         for(j = i->next; j!=NULL; j=j->next)
7.             if(i->data > j->data)
8.                 swapData(i, j);
9. }
```

Sắp xếp danh sách – Selection sort

```
1. // Sắp xếp giảm dần
2. void selectionSort(tList *list)
3. {
4.     tNode *i, *j, *max;
5.     for(i = list->head; i!=list->tail; i=i->next)
6.     {
7.         max = i;
8.         for(j = i->next; j!=NULL; j=j->next)
9.             if(max->data < j->data)
10.                max = j;
11.         if(i!=max)
12.             swapData(i, max);
13.     }
14. }
```

Sắp xếp danh sách

- ▶ Một số thuật toán không được ưa thích trên danh sách liên kết đơn
 - ▶ Danh sách liên kết đơn khó duyệt lùi
- ▶ Vẫn có thể cài đặt được các thuật toán

Vận dụng

- ▶ Bài tập: Container tracking
- ▶ Bổ sung tác vụ sắp xếp container
 - ▶ Theo khối lượng đang chứa
- ▶ Tạo menu để thực hiện các tác vụ đã tạo.

Menu

1. Nhập container mới
2. Xuất thông tin các container
3. Liệt kê các container đang dùng
4. Đếm số lượng container rồi
5. Tính tổng khối lượng hàng hoá
6. Tìm địa chỉ GPS của một container
7. Cập nhật thông tin của một container
8. Sắp xếp danh sách theo khối lượng
9. Xoá một container
10. Xoá toàn bộ danh sách
11. Thoát chương trình

INDUSTRIAL UNIVERSITY OF HO CHI MINH CITY



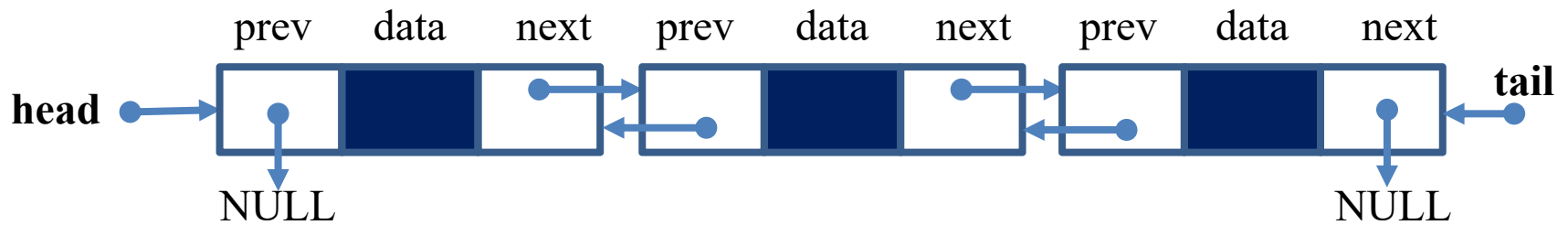
Data structures and algorithms

Doubly/Circular linked list

Dr. Ngo Huu Dung

Dẫn nhập

- ▶ Danh sách liên kết đôi
 - ▶ Hai chiều
 - ▶ Thêm con trỏ previous
 - ▶ Từ một nút có thể duyệt đến nút trước và sau nó
 - ▶ Các thao tác tương tự singly linked list
 - ▶ Xử lý thêm cho con trỏ previous
- ▶ Danh sách liên kết vòng
 - ▶ Nút cuối trỏ đến nút đầu
 - ▶ Có thể là danh sách đơn hoặc đôi
 - ▶ Các thao tác tương tự



Doubly linked list

Danh sách liên kết đôi

Doubly linked list – Khai báo

- ▶ Khai báo nút kiểu cấu trúc
 - ▶ Phần dữ liệu (int, float, char, struct...)
 - ▶ Phần liên kết (pointer)
- ▶ Khai báo con trỏ *head* và *tail*

```
1. struct Node
2. {
3.     int data;
4.     struct Node *next;
5.     struct Node *prev;
6. };
7. typedef struct Node tNode;
8. tNode *head;
9. tNode *tail;
```



head →

tail →

Thao tác cơ bản

- ▶ Khởi tạo danh sách, nút mới
- ▶ Thêm phần tử
 - ▶ Vào đầu, vào cuối, chèn vào sau một phần tử
- ▶ Duyệt danh sách
 - ▶ Xuất, trích xuất, đếm, tính toán
- ▶ Tìm kiếm
 - ▶ Min, max, giá trị X
- ▶ Xoá phần tử
 - ▶ Ở đầu, ở cuối, ở giữa
- ▶ Sắp xếp

Bài toán đặt ra

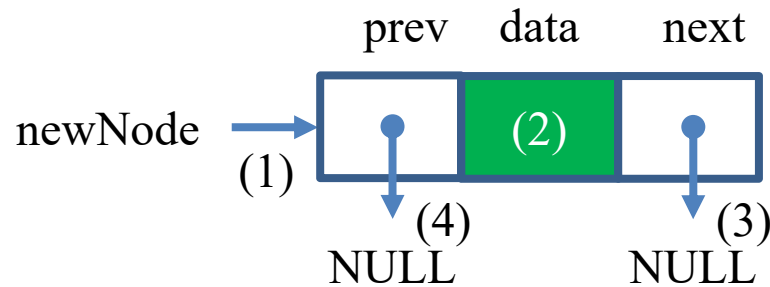
- ▶ Danh sách các số nguyên
 - ▶ Cấu trúc dữ liệu danh sách liên kết đôi
 - ▶ Các thao tác cơ bản trên danh sách liên kết đôi
 - ▶ Menu thực hiện

```
1. // Kiểu nút
2. struct Node
3. {
4.     int data;
5.     struct Node *next;
6.     struct Node *prev;
7. };
8. typedef struct Node tNode;
9. // Kiểu danh sách
10. struct List
11. {
12.     tNode *head;
13.     tNode *tail;
14. };
15. typedef struct List tList;
16. // Biến danh sách
17. tList ds;
```

Một số hàm tạo, thêm và chèn phần tử

1. `// Initiate a new list`
2. `void init(tList *);` // Giống singly linked list
3. `// Make a new node`
4. `tNode* makeNode();`
5. `// Add a new node to the head of list`
6. `void addHead(tList *);`
7. `// Add a new node to the tail of list`
8. `void addTail(tList *);`
9. `// Insert a node after a given node`
10. `void insertAfter(tList *, tNode *, tNode *);`

Tạo một nút mới

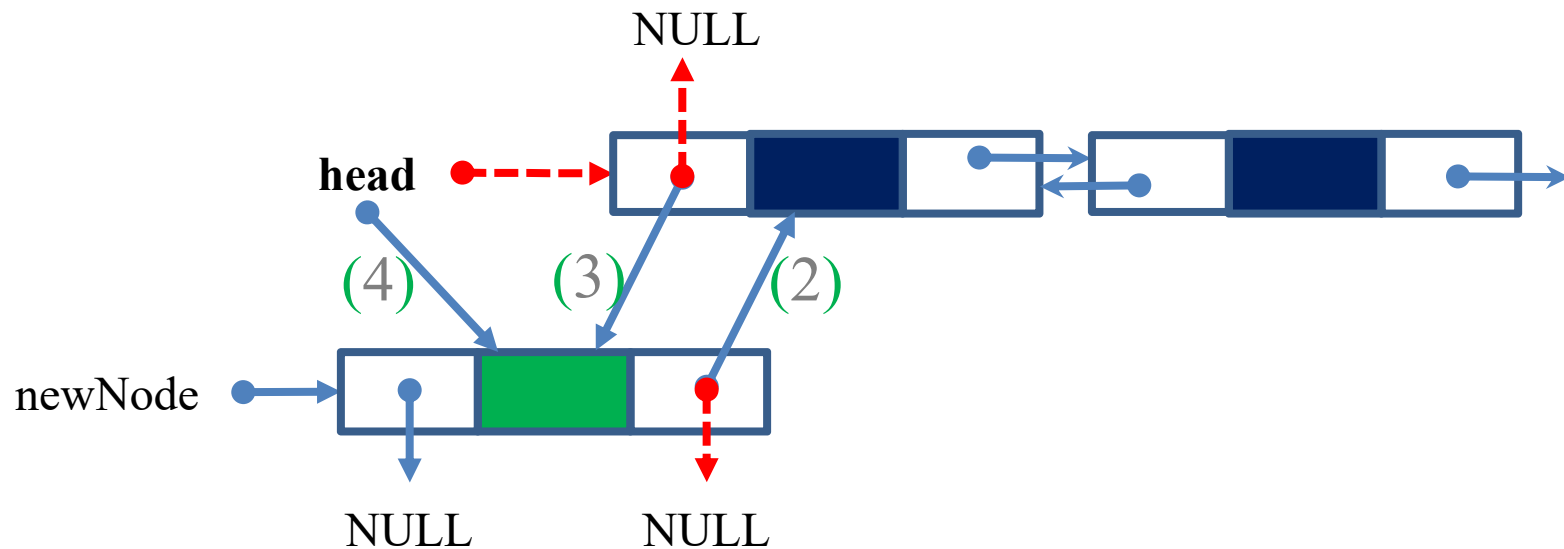


```
1. // Make a new node
2. tNode* makeNode()
3. {
4.     tNode *newNode;
5.     newNode = (tNode*)malloc(sizeof(tNode));
6.     // newNode = new tNode;
7.     printf("Nhap du lieu: ");
8.     scanf("%d", &newNode->data);
9.     newNode->next = NULL;
10.    newNode->prev = NULL;
11.    return newNode;
12.}
```

1. Cấp phát bộ nhớ
▶ Hàm malloc hoặc new
2. Nhập dữ liệu
3. Khởi tạo next rỗng
4. Khởi tạo prev rỗng

Thêm nút mới vào đầu danh sách

- ▶ Tạo nút mới
- ▶ Thêm vào đầu danh sách
 - ▶ Danh sách rỗng? (1)



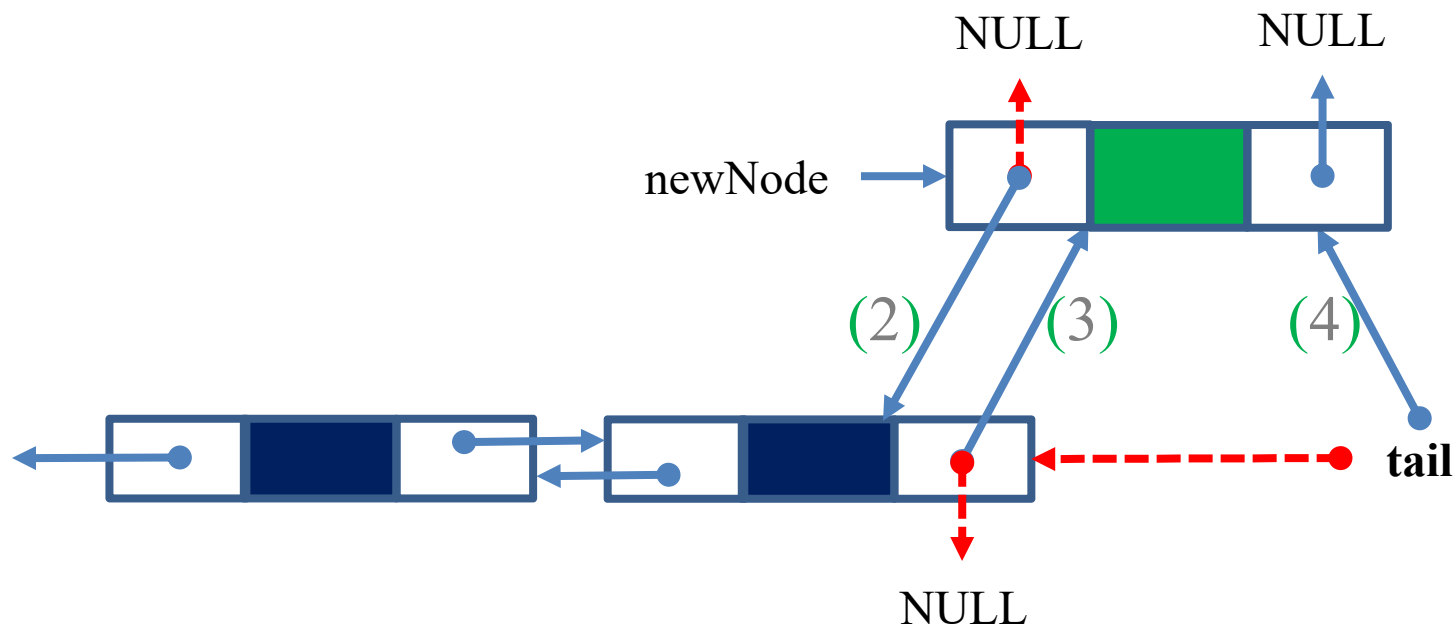
Thêm nút mới vào đầu danh sách

```
1. // Add a new node into the head of list
2. void addHead(tList *list)
3. {
4.     // Make a new node
5.     tNode *newNode = makeNode();

6.     if(list->head == NULL) {           // (1)
7.         list->head = newNode;
8.         list->tail = newNode;
9.     }else{                             // not empty
10.        newNode->next = list->head;      // (2)
11.        list->head->prev = newNode;      // (3)
12.        list->head = newNode;           // (4)
13.    }
14.}
```

Thêm nút mới vào **cuối** danh sách

- ▶ Tạo nút mới
- ▶ Thêm vào cuối danh sách
 - ▶ Danh sách rỗng? (1)
 - ▶ Danh sách không rỗng?

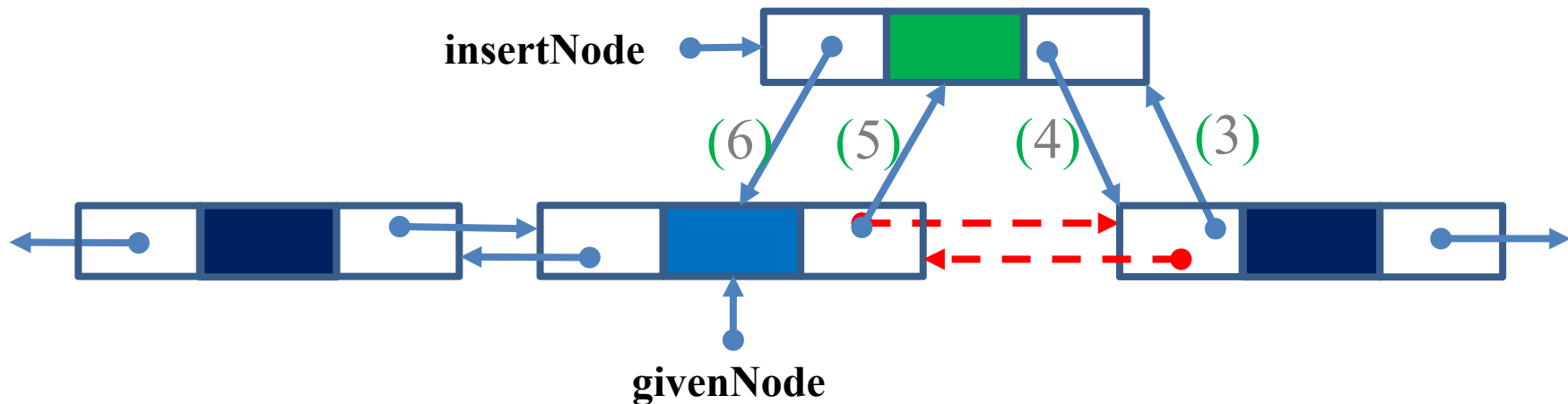


Thêm nút mới vào **cuối** danh sách

```
1. // Add a new node into the tail of list
2. void addTail(tList *list)
3. {
4.     // Make a new node
5.     tNode *newNode = makeNode();
6.
7.     if(!list->head) {                // (1)
8.         list->head = newNode;
9.         list->tail = newNode;
10.    }else{                            // Not empty
11.        newNode->prev = list->tail;    // (2)
12.        list->tail->next = newNode;    // (3)
13.        list->tail = newNode;         // (4)
14.    }
15.}
```

Chèn một nút vào **sau** một nút

- ▶ Chèn nút insertNode vào sau givenNode
 - ▶ givenNode hoặc insertNode rỗng? (1)
 - ▶ Trường hợp givenNode là nút cuối? (2)



Chèn một nút vào sau một nút

```
1. // Insert a node after a given node
2. void insertAfter(tList *list, tNode *givenNode,
   tNode *insertNode)
3. {
4.     if(givenNode==NULL||insertNode==NULL)           //(1)
5.         return;
6.     if(!givenNode->next)                             //(2)
7.         list->tail = insertNode;
8.     else
9.         givenNode->next->prev = insertNode;          //(3)
10.    insertNode->next = givenNode->next;               //(4)
11.    givenNode->next = insertNode;                     //(5)
12.    insertNode->prev = givenNode;                     //(6)
13.}
```

Một số hàm duyệt danh sách

- ✓ Tương tự danh sách liên kết đơn
- ✓ Có thể duyệt từ tail

```
1. // Print all list
2. void output(tList);

3. // Count a list
4. int count(tList);

5. // Calculate a list
6. int total(tList);

7. // Search an item on a list
8. tNode *search(tList, int);

9. // Find the max node on a list
10. tNode* maxNode(tList);
```

Duyệt danh sách

```
1. tNode *node = list.head;
2. while (node) {
3.     // Thao tác xử lý
4.     node = node->next;
5. }

6. for (node = list.head; node; node = node->next)
7.     // Thao tác xử lý

8. tNode *node = list.tail;
9. while (node) {
10.    // Thao tác xử lý
11.    node = node->prev;
12.}

13. for (node = list.tail; node; node = node->prev)
14.    // Thao tác xử lý
```

Một số hàm xoá node

1. `// Delete the head node`
2. `void delHead(tList *);`

3. `// Delete the node after a given node`
4. `void delAfter(tList *, tNode *);`

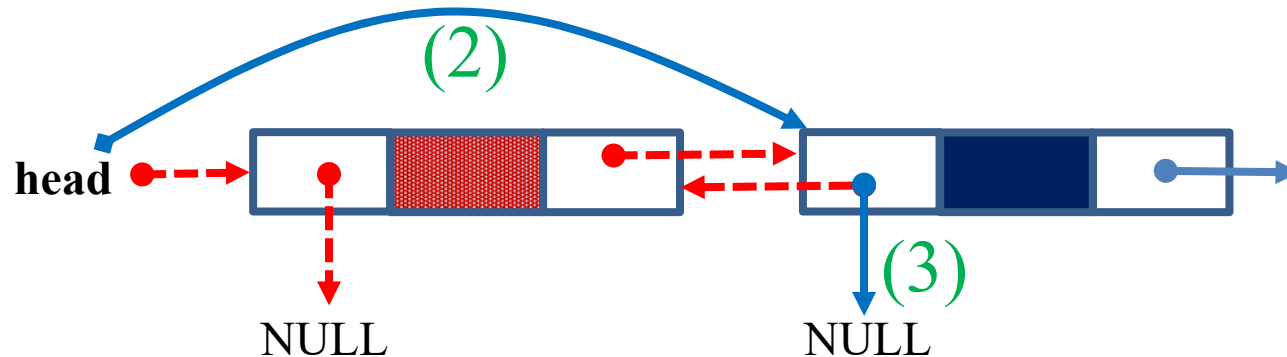
5. `// Delete the tail node`
6. `void delTail(tList *);`

7. `// Delete any given node`
8. `void delNode(tList *, tNode *);`

9. `// Remove all list - similar to singly linked list`
10. `void removeList(tList *);`

Xoá nút đầu danh sách

- ▶ Danh sách rỗng? (1)



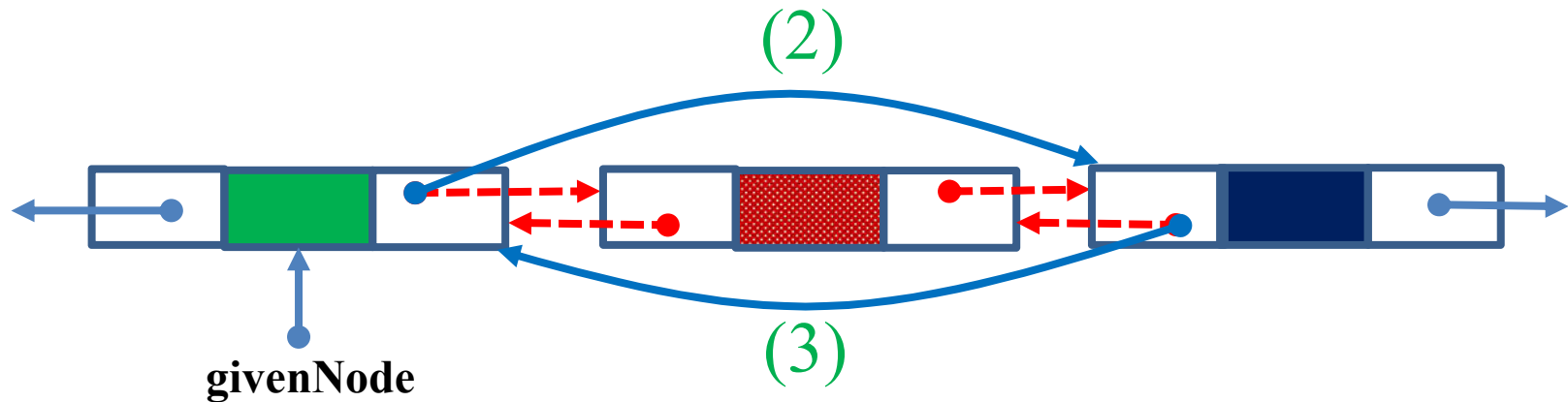
- ▶ Nút cần xoá là nút cuối cùng?
 - ▶ Trở thành danh sách rỗng, điều chỉnh tail (4)
- ▶ Giải phóng bộ nhớ (5)

Xoá nút đầu danh sách

```
1. // Delete the head node
2. void delHead(tList *list)
3. {
4.     tNode *delNode = list->head;
5.     if(!delNode)                // (1)
6.         return;
7.     // head point to next node
8.     list->head = delNode->next;  // (2)
9.     if(list->head)
10.        list->head->prev = NULL;  // (3)
11.     else
12.        list->tail = NULL;        // (4)
13.     free(delNode);              // (5)
14. }
```

Xoá nút **sau** nút cho trước

- ▶ givenNode hoặc givenNode→next rỗng? (1)



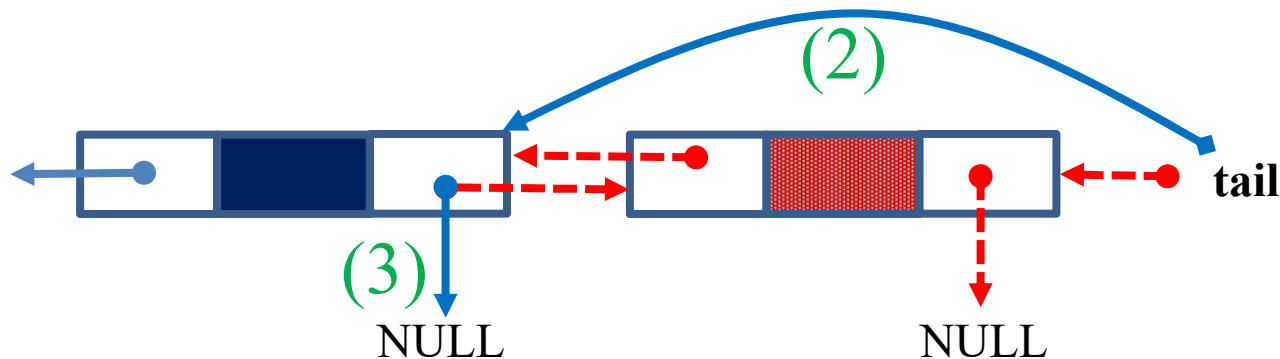
- ▶ givenNode là nút cuối? (4)
- ▶ Giải phóng bộ nhớ (5)

Xoá nút sau nút cho trước

```
1. // Delete the node after a given node
2. void delAfter(tList *list, tNode *givenNode)
3. {
4.     if(!givenNode || !givenNode->next)           //(1)
5.         return;
6.     tNode *delNode = givenNode->next;
7.     givenNode->next = delNode->next;               //(2)
8.     if(delNode->next)
9.         delNode->next->prev = givenNode;           //(3)
10.    else
11.        list->tail = givenNode;                     //(4)
12.    free(delNode);                                   //(5)
13.}
```

Xoá nút **cuối** danh sách

- ▶ Tương tự xoá đầu danh sách
 - ▶ Đảo ngược, tail thay vì head, prev thay vì next...
 - ▶ Danh sách rỗng? (1)



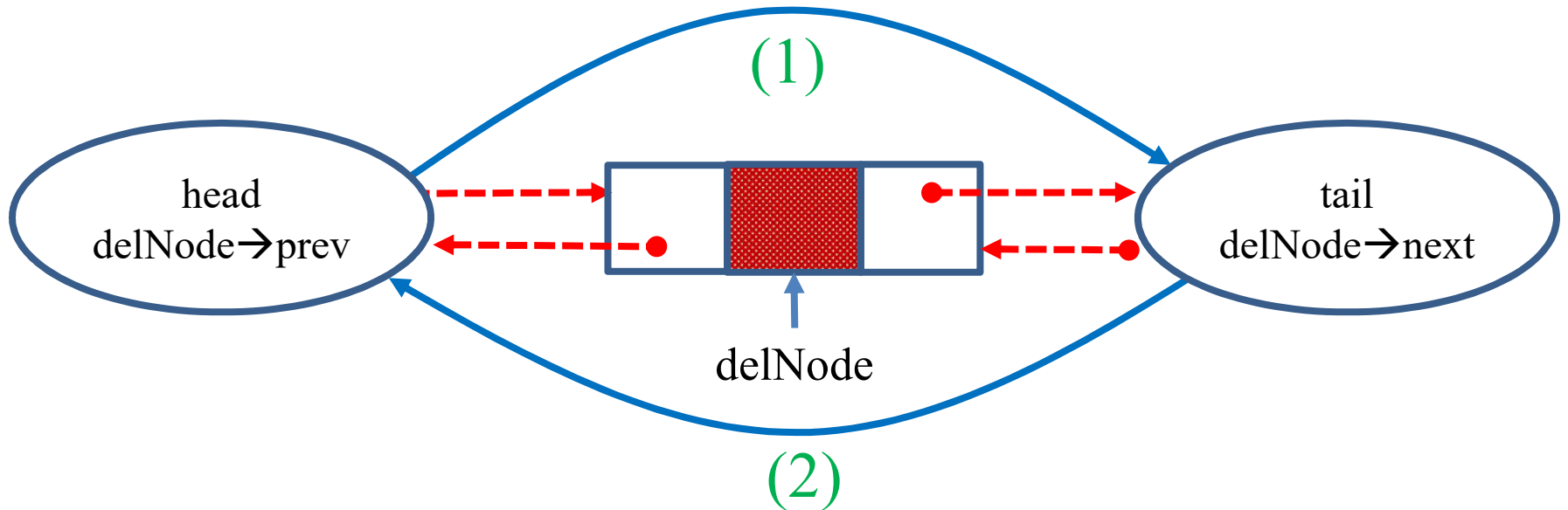
- ▶ Nút cần xoá là nút cuối cùng?
 - ▶ Trở thành danh sách rỗng, điều chỉnh head (4)
- ▶ Giải phóng bộ nhớ (5)

Xoá nút cuối danh sách

```
1. // Delete the tail node
2. void delTail(tList *list)
3. {
4.     tNode *delNode = list->tail;
5.     if(!delNode) // (1)
6.         return;
7.     // tail point to previous node
8.     list->tail = delNode->prev; // (2)
9.     if(list->tail) // Not empty
10.        list->tail->next = NULL; // (3)
11.     else // Become empty
12.        list->head = NULL; // (4)
13.     free(delNode); // (5)
14. }
```

Xoá một nút bất kỳ cho trước

- ▶ Hàm tổng quát, xoá bất kỳ nút nào



Xoá một nút bất kỳ cho trước

```
1. // Delete any given node
2. void delGivenNode(tList *list, tNode *delNode)
3. {
4.     if(!delNode || !list->head)
5.         return;
6.     if(delNode == list->head)
7.         list->head = delNode->next;           //(1)
8.     if(delNode->prev)
9.         delNode->prev->next = delNode->next; //(1)
10.    if(delNode == list->tail)
11.        list->tail = delNode->prev;           //(2)
12.    if(delNode->next)
13.        delNode->next->prev = delNode->prev; //(2)
14.    free(delNode);
15.}
```

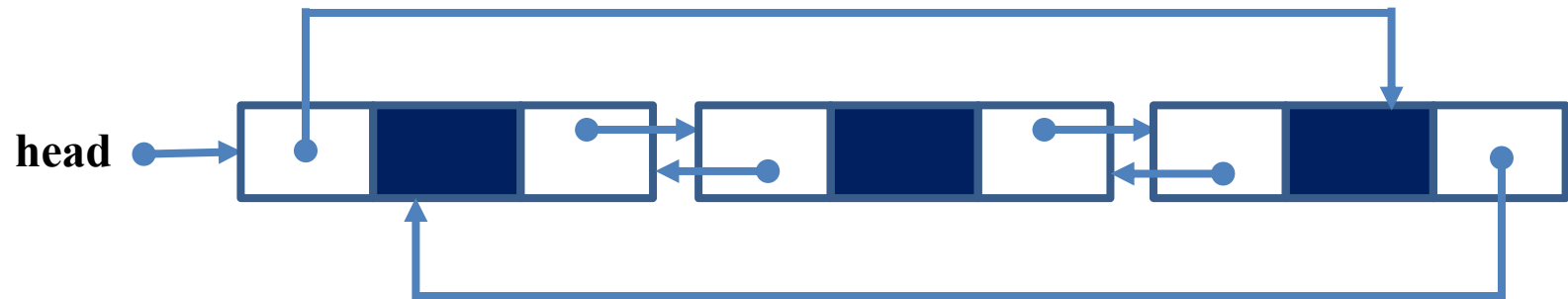
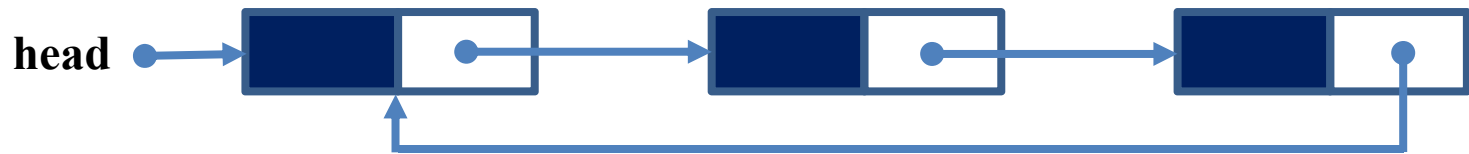

Sắp xếp danh sách

```
1. /* Tương tự danh sách đơn khi ta giữ nguyên liên
   kết và chỉ hoán vị dữ liệu*/

2. // Sắp xếp tăng dần
3. void selectionSort(tList *list)
4. {
5.     tNode *i, *j, *min;
6.     for(i = list->head; i!=list->tail; i=i->next)
7.     {
8.         min = i;
9.         for(j = i->next; j!=NULL; j=j->next)
10.            if(min->data > j->data)
11.                min = j;
12.         if(i!=min)
13.             swapData(i, min);
14.     }
15. }
```

Vận dụng

- ▶ Bài tập: Quiz
- ▶ Chương trình thi trắc nghiệm dễ dàng chuyển đến câu hỏi tiếp theo hoặc câu hỏi trước đó.
 - ▶ Thao tác cơ bản
 - ▶ Thêm câu hỏi,
 - ▶ Xoá câu hỏi,
 - ▶ Sửa câu hỏi,
 - ▶ Tiến hành thi, chấm điểm



Danh sách liên kết vòng

Circular linked list

Liên kết vòng

- ▶ Có thể là danh sách đơn hoặc đôi
- ▶ Không có nút cuối trỏ vào NULL
- ▶ Nút “cuối” nối vào nút đầu tiên
- ▶ Từ phần tử bất kỳ có thể duyệt toàn bộ danh sách
- ▶ Phù hợp với các ứng dụng lặp xoay vòng
 - ▶ Các ứng dụng chạy trong hệ điều hành đa nhiệm
 - ▶ Các cấu trúc dữ liệu nâng cao
 - ▶ Hàng đợi ưu tiên (priority queue)

Thao tác cơ bản – so sánh

- ▶ Khởi tạo danh sách, nút mới
 - ▶ Tương tự Singly/Doubly linked list
- ▶ Thêm phần tử
 - ▶ Chú ý nút cuối trở lại nút đầu
- ▶ Duyệt danh sách
 - ▶ Điều kiện dừng là trở lại nút bắt đầu
- ▶ Xoá phần tử
 - ▶ Chú ý nút cuối trở lại nút đầu
- ▶ Sắp xếp
 - ▶ Điều kiện dừng của vòng lặp

Thêm nút vào đầu danh sách

```
1. // Add a new node to the head of list
2. void addHead(tList *list)
3. {
4.     tNode *newNode = makeNode();

5.     if(list->head == NULL) {
6.         list->head = newNode;
7.         list->tail = newNode;
8.         newNode->next = newNode;    //Link to head
9.     }else{
10.        newNode->next = list->head;
11.        list->head = newNode;
12.        list->tail->next = newNode; //Link to head
13.    }
14.}
```

Duyệt danh sách

```
1. // Traversing a given Circular linked list
2. // Start from head and end at head
3. // Print all list
4. void printList(tList list)
5. {
6.     tNode *node = list.head;
7.     printf("The linked list: ");
8.     if(node)        // Not empty list
9.     {
10.         do{
11.             printf("%d ", node->data);
12.             node = node->next;
13.         }while(node != list.head);    // !?
14.     }else
15.         printf("Empty.");
16. }
```

Xoá nút đầu danh sách

```
1. // Delete the head node
2. void delHead(tList *list)
3. {
4.     tNode *delNode = list->head;
5.     if(delNode==NULL)
6.         return;
7.     if(list->head == list->tail)
8.     {
9.         list->head = NULL;
10.        list->tail = NULL;
11.    }else{
12.        list->head = delNode->next;
13.        list->tail->next = list->head;    //!?
14.    }
15.    free(delNode) ;
16.}
```


Sắp xếp danh sách

```
1. // Sort to ascending list
2. void interchangeSort(tList *list)
3. {
4.     tNode *i, *j;
5.     for(i = list->head; i!=list->tail; i=i->next)
6.         for(j = i->next; j!=list->head; j=j->next)
7.             if(i->data > j->data)
8.                 swapData(i, j);
9. }
```

Without the tail !?

```
1. // Add a new node at the beginning
2. void addNode(tList *list)
3. {
4.     tNode *newNode = makeNode();
5.     tNode *temp = list->head;
6.     newNode->next = list->head;
7.     if(list->head)
8.     {
9.         while(temp->next != list->head)
10.            temp = temp->next;
11.         temp->next = newNode;
12.     }
13.     else
14.         newNode->next = newNode; //For the first node
15.     list->head = newNode;
16. }
```

Vận dụng

- ▶ Bài tập: Game board
- ▶ Chương trình nhiều người chơi xoay vòng, mỗi người đến lượt sẽ được cung cấp một con số ngẫu nhiên $[0..9]$, người chơi phải nhập vào một số để tạo thành số nguyên tố. Ai nhập sai sẽ bị loại khỏi cuộc chơi.

INDUSTRIAL UNIVERSITY OF HO CHI MINH CITY



Data structures and algorithms

Stack and Queue

Dr. Ngô Hữu Dũng

Introduction

- ▶ **Stack** (LIFO – last in, first out): a collection of items in which only the most recently added item may be removed.



- ▶ **Queue** (FIFO – first in, first out): a collection of items in which first items entered are the first ones to be removed.



Stack vs. Queue

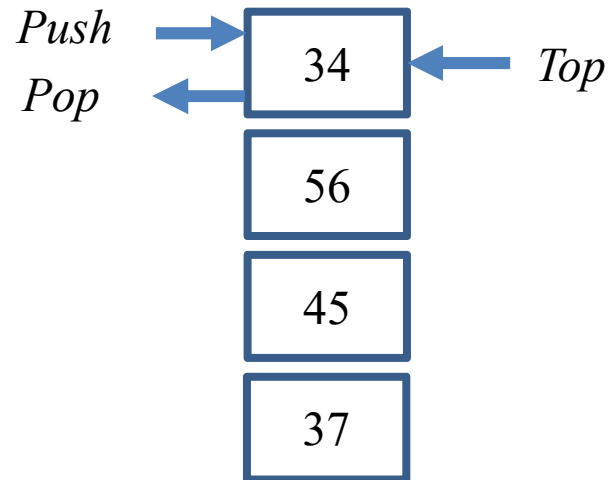
▶ Stack – Ngăn xếp

▶ Last In First Out (LIFO)

▶ Thao tác

▶ Push

▶ Pop



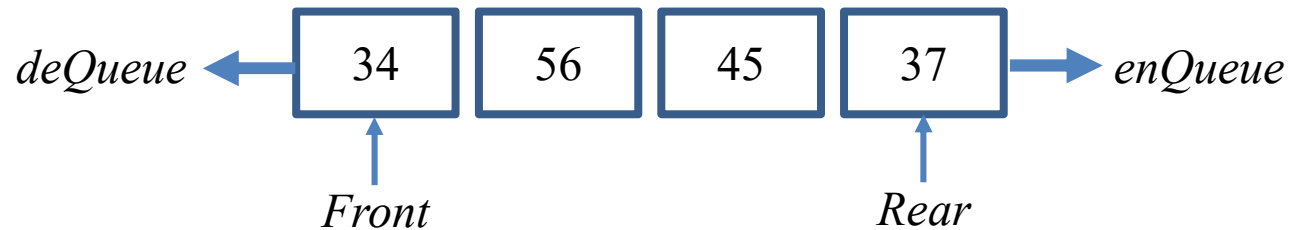
▶ Queue – Hàng đợi

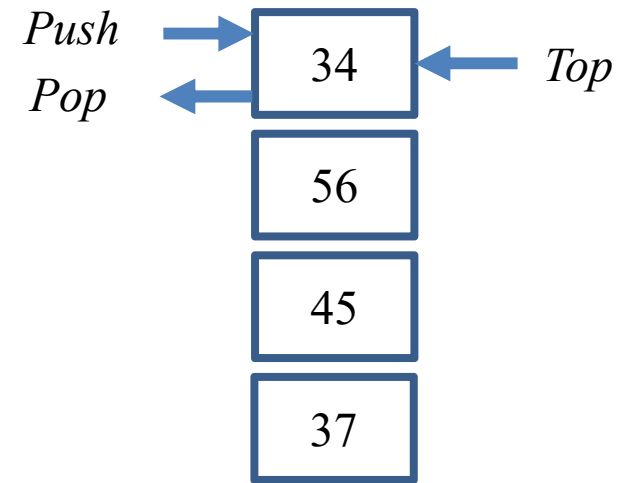
▶ First In First Out (FIFO)

▶ Thao tác

▶ enqueue

▶ dequeue





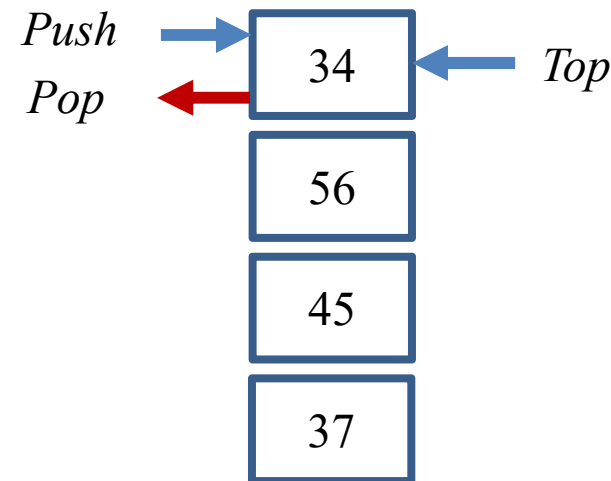
Stack – Last in, first out

Stack

Ngăn xếp

Khái niệm Stack

- ▶ Lưu trữ một tập các phần tử theo một trật tự nhất định
- ▶ Nguyên tắc: Last in, first out
 - ▶ Vào sau cùng, ra trước tiên
- ▶ Top: Phần tử trên cùng
- ▶ Chèn phần tử vào top
 - ▶ Thao tác push
 - ▶ Chèn vào đầu danh sách
- ▶ Xuất phần tử từ top
 - ▶ Thao tác pop
 - ▶ Xóa phần tử ở đầu danh sách



Applications

- ▶ Balancing of symbols
- ▶ Infix to Postfix /Prefix conversion
- ▶ Redo-undo features at many places like editors, Photoshop.
- ▶ Forward and backward feature in web browsers
- ▶ Used in many algorithms like Tower of Hanoi, tree traversals, stock span problem, histogram problem.
- ▶ Other applications can be Backtracking, Knight tour problem, rat in a maze, N queen problem and Sudoku solver

Thao tác trên Stack

- ▶ Push: Thêm một phần tử vào stack
 - ▶ Nếu stack chưa đầy thì thêm phần tử ở top
- ▶ Pop: Xoá một phần tử từ stack
 - ▶ Nếu stack không rỗng thì xoá phần tử ở top
- ▶ Top: Lấy phần tử ở top
- ▶ initStack: khởi tạo Stack
- ▶ isEmpty: Kiểm tra stack rỗng?
 - ▶ Trả về true nếu stack rỗng
- ▶ isFull: Kiểm tra stack đầy?
 - ▶ Trả về true nếu stack đầy

Tổ chức dữ liệu

► Array

```
1. struct Stack
2. {
3.     int top;
4.     int capacity;
5.     int* array;
6. };

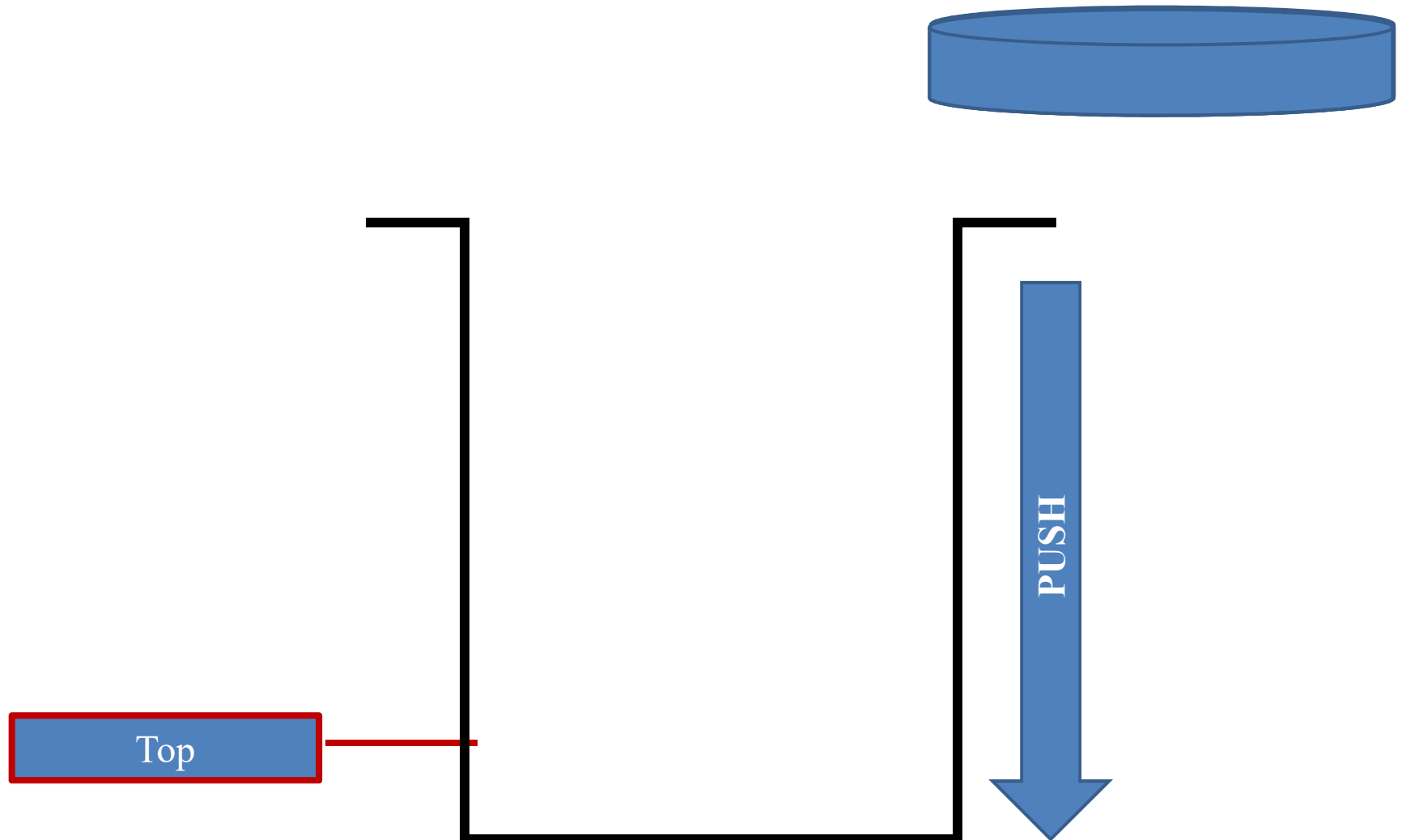
7. struct Stack* tStack;
```

► Linked list

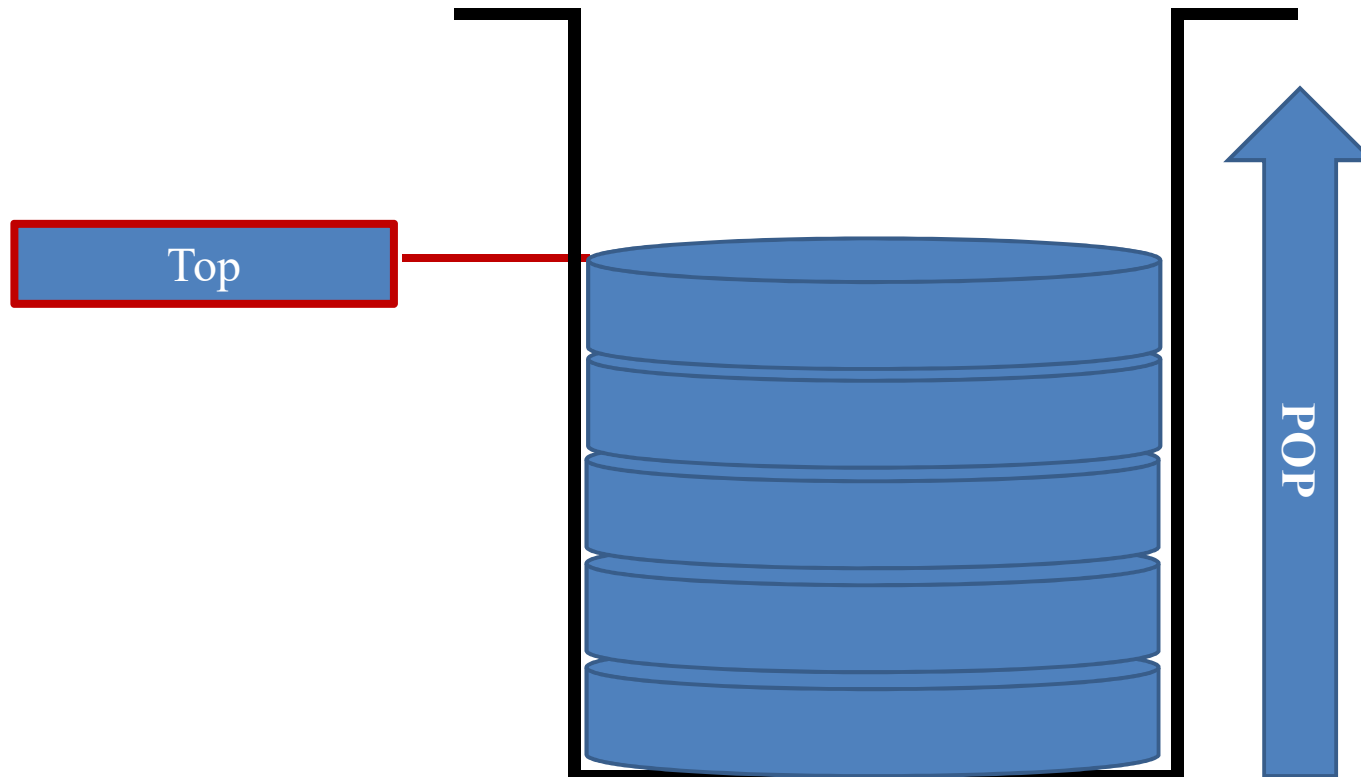
```
1. struct Node
2. {
3.     int data;
4.     struct Node* next;
5. };

6. struct Stack
7. {
8.     Node *top;
9. };
```

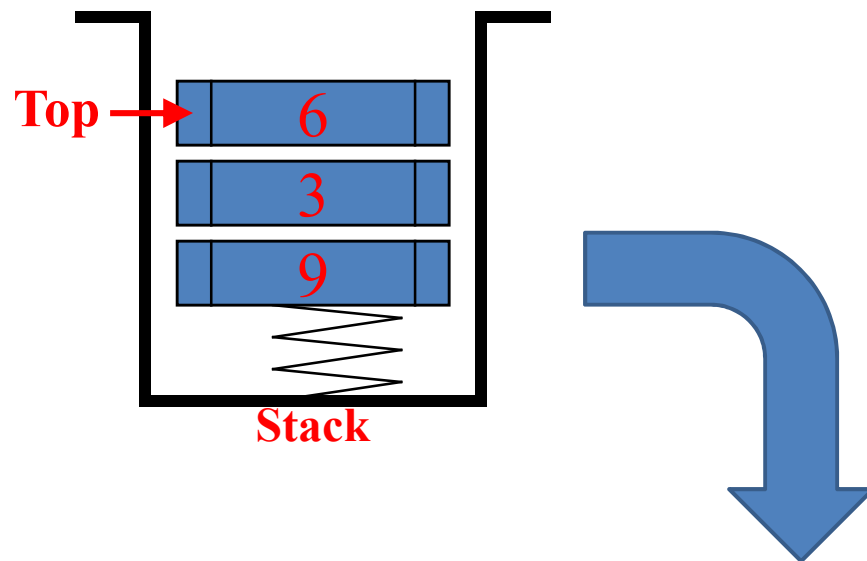
Thao tác Push vào Stack



Thao tác Pop khỏi stack



Stack – Sử dụng mảng



9	3	6							
0	1	2	3	4	5	6	7	8	9

Stack số nguyên – Sử dụng mảng

```
struct ttStack
{
    int* StkArray; // mảng chứa các phần tử
    int StkMax; // số phần tử tối đa
    int StkTop; // vị trí đỉnh Stack
};

typedef struct ttStack STACK;
```

Stack số nguyên – Sử dụng mảng

```
bool InitStack(STACK& s, int MaxItems)
{
    s.StkArray = new int[MaxItems];
    if (s.StkArray == NULL)
        return false;
    s.StkMax = MaxItems;
    s.StkTop = -1;
    return true;
}
```


Stack số nguyên – Sử dụng mảng

```
bool IsEmpty(const STACK &s)
{
    if (s.StkTop==-1)
        return true;
    return false;
}
```

Stack số nguyên – Sử dụng mảng

```
bool IsFull(const STACK &s)
{
    if (s.StkTop==s.StkMax-1)
        return true;
    return false;
}
```

Stack số nguyên – Sử dụng mảng

```
bool Push (STACK &s, int newitem)
{
    if (IsFull(s))
        return false;
    s.StkTop++;
    s.StkArray[s.StkTop] = newitem;
    return true;
}
```

Stack số nguyên – Sử dụng mảng

```
bool Pop(STACK &s, int &outitem)
{
    if (IsEmpty(s))
        return false;
    outitem = s.StkArray[s.StkTop];
    s.StkTop--;
    return true;
}
```

Bài tập

- ▶ Viết hàm nhập và xuất Stack số nguyên
- ▶ Khai báo cấu trúc và viết hàm tạo Stack từ chuỗi ký tự str (mỗi phần tử Stack là ký tự)
- ▶ Khai báo cấu trúc và viết hàm tạo Stack từ chuỗi ký tự str (mỗi phần tử Stack là một từ - từ cách nhau bởi khoảng trắng)

Stack – Ví dụ ứng dụng

- ▶ Kiểm tra sự tương ứng của các cặp ngoặc đơn trong một biểu thức

▶ $((A + B) / C$

$(A + B) / C)$

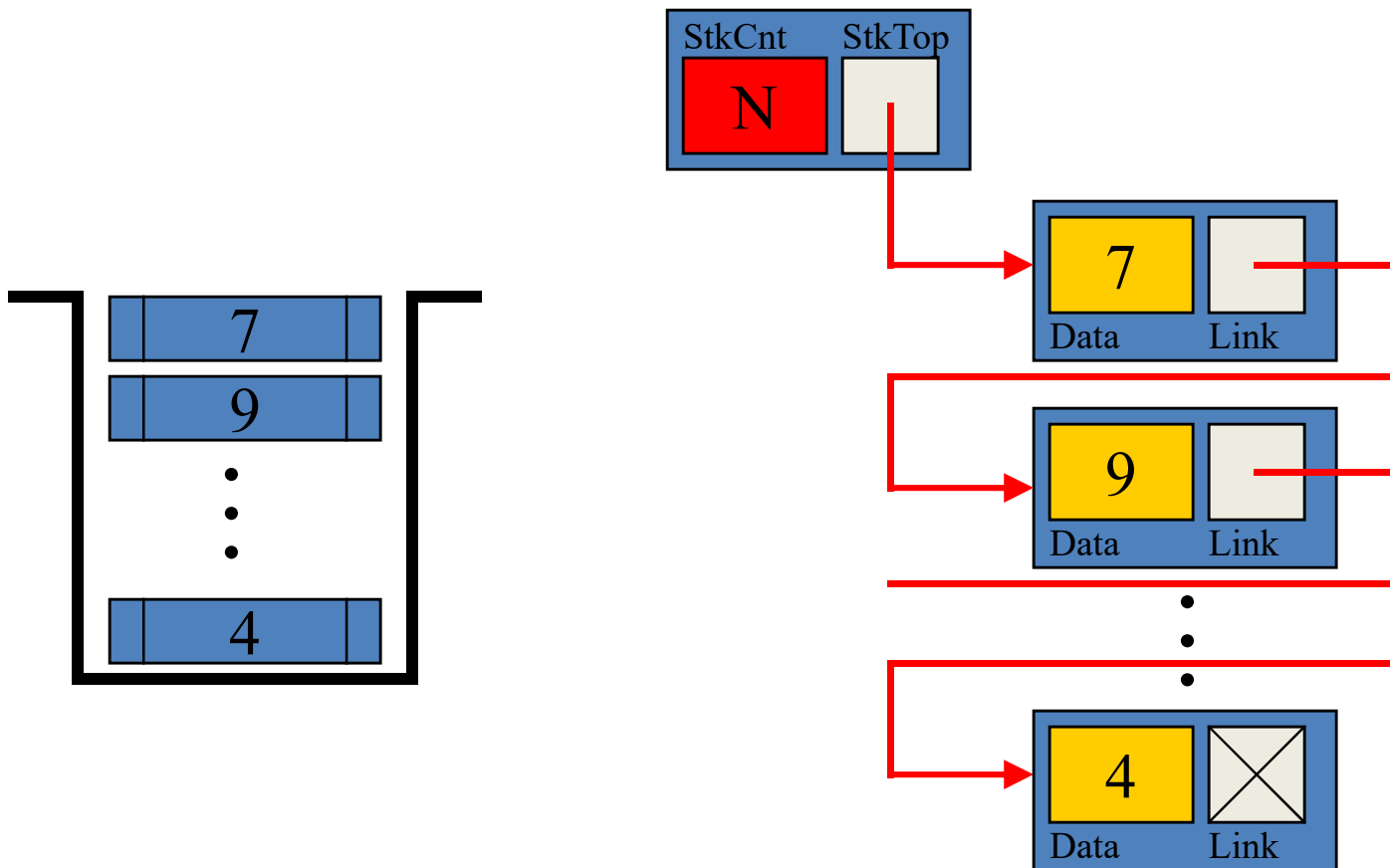
- ▶ Đảo ngược một chuỗi ký tự

▶ Cá Ăn Kiến



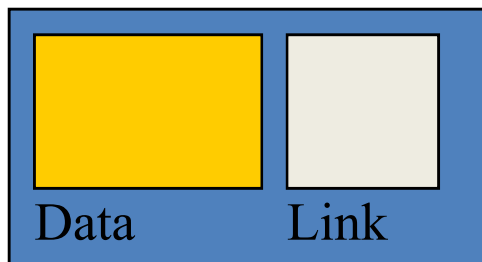
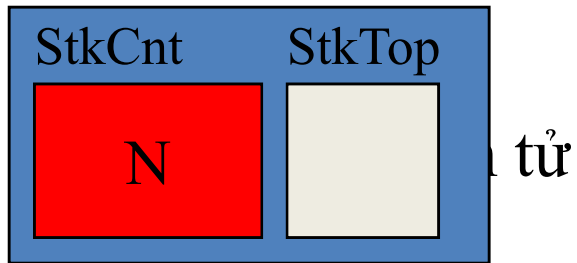
?
nếIK nĂ áC

Stack – Sử dụng DSLK



Stack – Sử dụng DSLK

► Cấu tạo đầu stack



stack

StkCnt

<integer>

StkTop

<node pointer>

end stack

node

Data

<datatype>

Link

<node pointer>

end node

Stack số nguyên – Sử dụng DSLK

```
typedef struct tagSTACK_NODE
{
    int Data;
    tagSTACK_NODE *pNext;
} STACK_NODE;
typedef struct STACK
{
    int StkCount;
    STACK_NODE *StkTop;
};
```

Stack – Sử dụng DSLK

- ▶ VD: Thực hiện một số thao tác trên stack

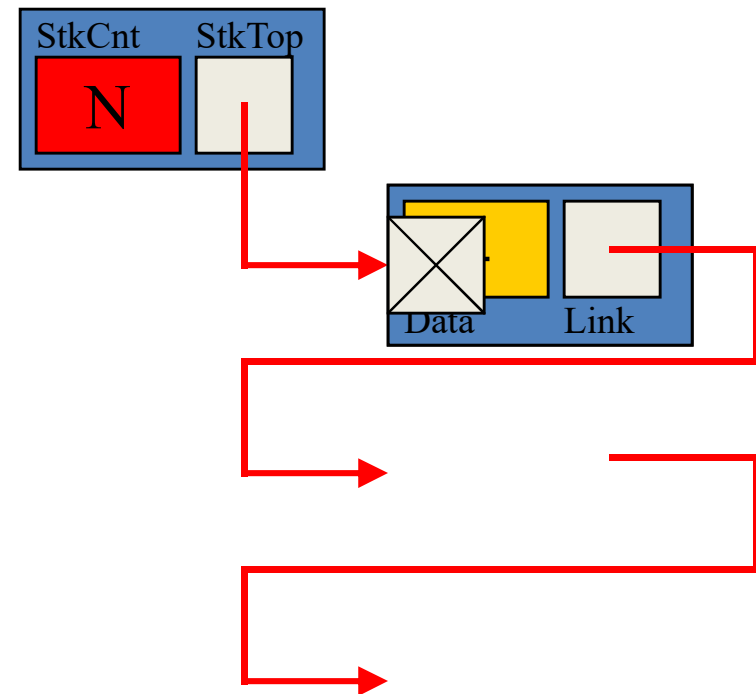
STACK s;

InitStack(s);

Push(s, 7);

Push(s, 4);

Pop(s, x); // x = ?



Stack số nguyên – Sử dụng DSLK

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

Stack số nguyên – Sử dụng DSLK

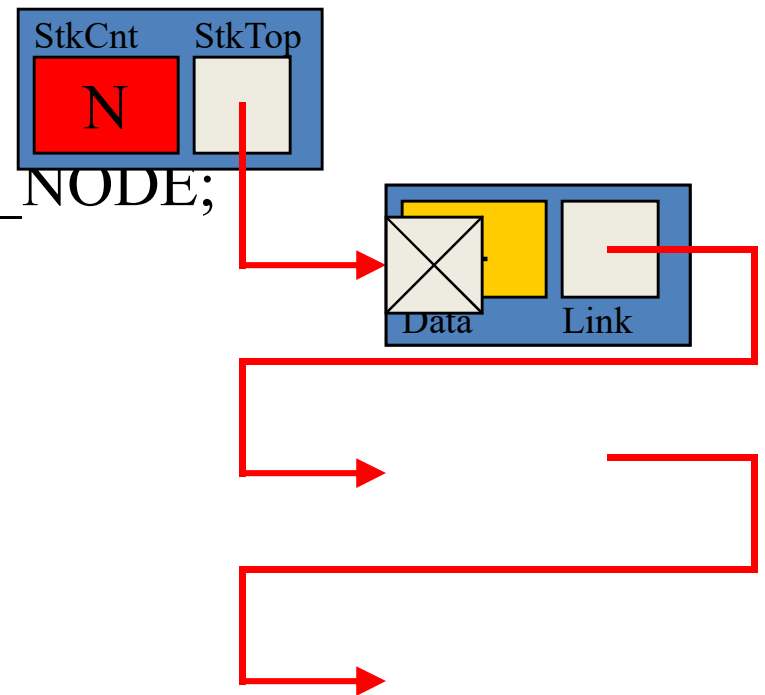
```
bool IsEmpty(const STACK &s)
{
    if (s.StkTop == NULL)
        return true;
    return false;
}
```

Stack số nguyên – Sử dụng DSLK

```
bool IsFull (const STACK s)
{
    STACK_NODE* temp = new STACK_NODE;
    if (temp == NULL)
        return true;
    delete temp;
    return false;
}
```

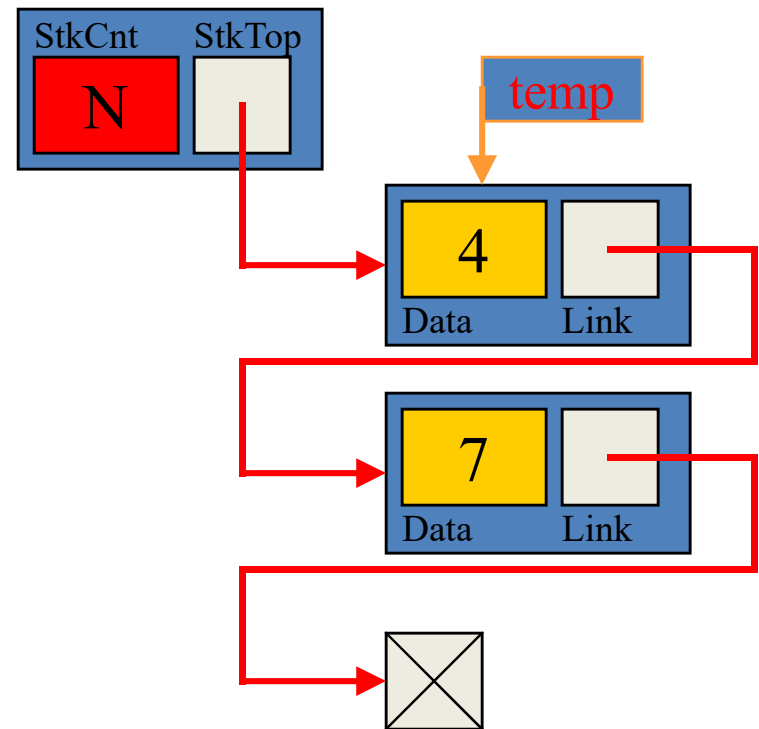
Stack số nguyên – Sử dụng DSLK

```
bool Push(STACK &s, int newitem)
{
    if (IsFull(s))
        return false;
    STACK_NODE *pNew = new STACK_NODE;
    pNew->Data = newitem;
    pNew->pNext = s.StkTop;
    s.StkTop = pNew;
    s.StkCount++;
    return true;
}
```



Stack số nguyên – Sử dụng DSLK

```
bool Pop(STACK &s, int &outitem)
{
    if (IsEmpty(s))
        return false;
    STACK_NODE *temp = s.StkTop;
    outitem = s.StkTop->Data;
    s.StkTop = s.StkTop->pNext;
    delete temp;
    s.StkCount--;
    return true;
}
```



outitem = 4

Stack – Ứng dụng

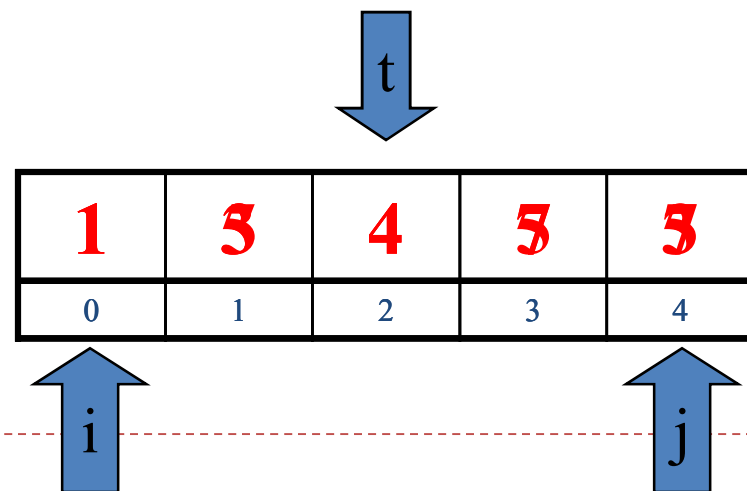
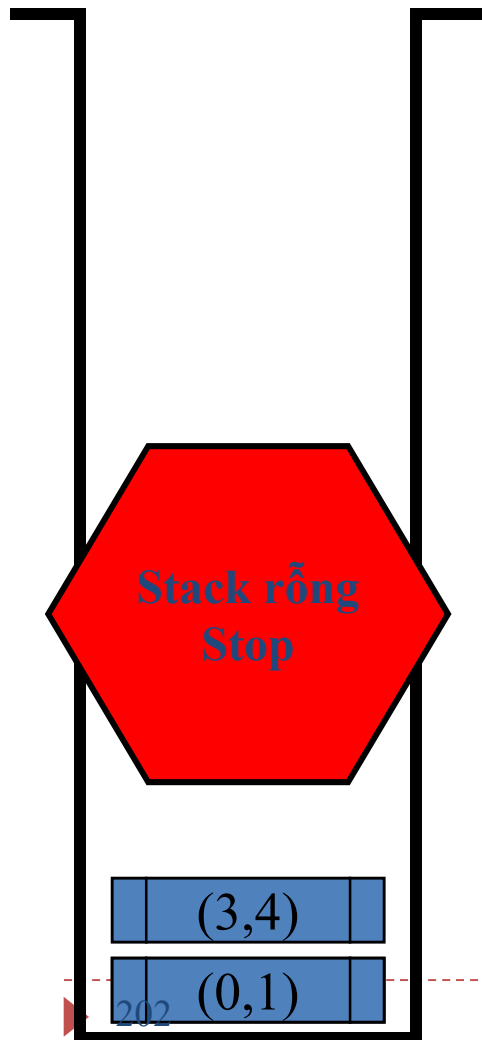
- ▶ Stack có nhiều ứng dụng:
- ▶ Lưu vết trong thuật toán “back-tracking” (theo dõi dấu vết)
- ▶ Tính giá trị biểu thức toán học (thuật toán Balan ngược)
- ▶ Khử đệ quy
- ▶ ...

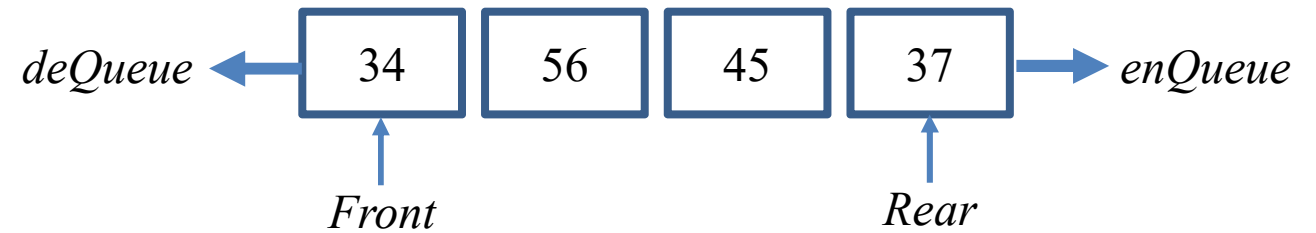
Stack – Quick Sort

- ▶ Để khử đệ quy cho Quick Sort, ta sử dụng một stack để lưu lại các partition (phân hoạch) cần tiến hành sắp xếp.
- ▶ **Ý tưởng:**
 - ▶ Push phân hoạch đầu tiên $(0, n-1)$ vào stack
 - ▶ Trong khi stack chưa rỗng
 - ▶ Pop một phân hoạch từ stack
 - ▶ Chọn phần tử trục trên phân hoạch này
 - ▶ Điều chỉnh phân hoạch tương ứng với trục
 - ▶ Push 2 phân hoạch bên trái và phải trục vào stack

Stack – Quick Sort

- ▶ Push phân hoạch đầu tiên $(0, n-1)$ vào stack
- ▶ Trong khi stack chưa rỗng
 - ▶ Pop một phân hoạch từ stack
 - ▶ Chọn phần tử trục trên phân hoạch này
 - ▶ Điều chỉnh phân hoạch tương ứng với trục
 - ▶ Push 2 phân hoạch bên trái và phải trục vào stack



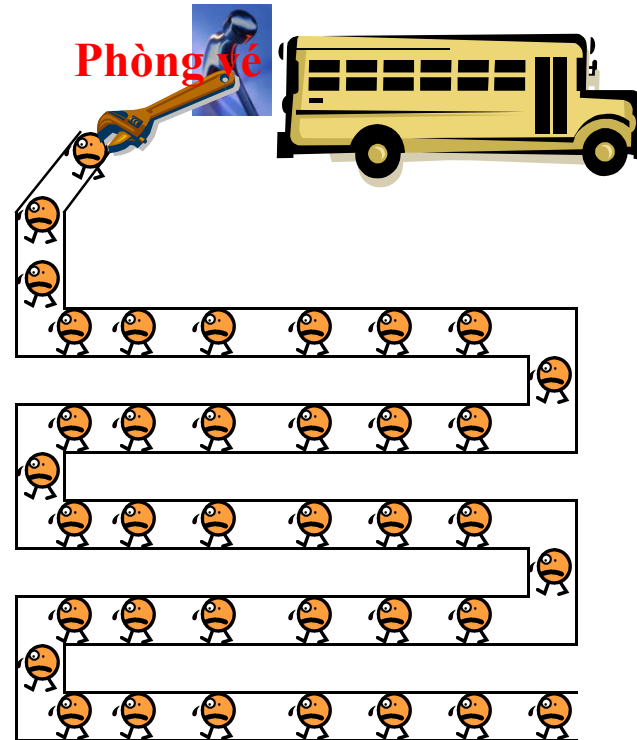
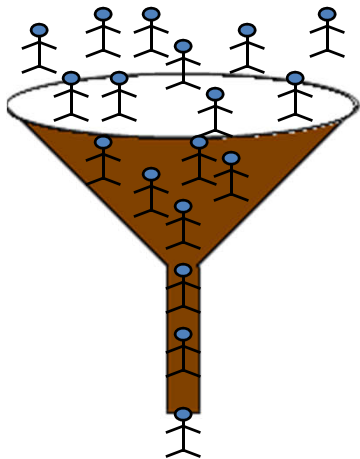


Queue – First in, first out

Queue

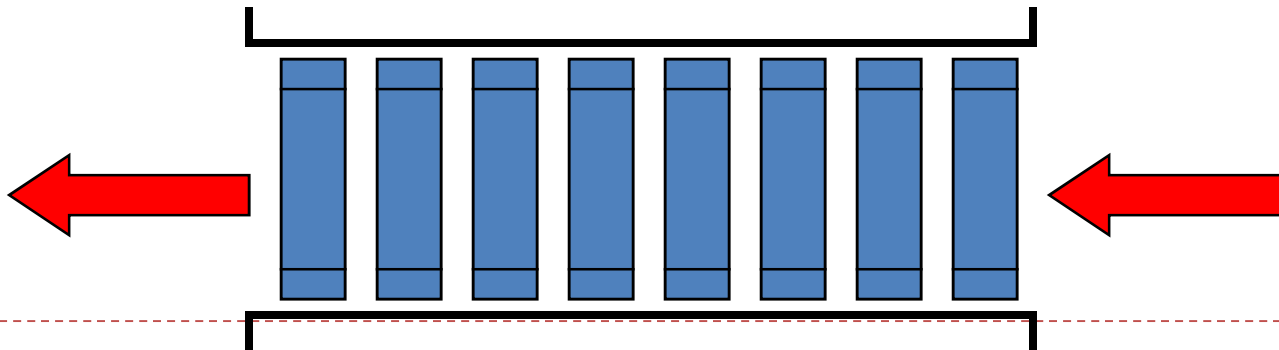
Hàng đợi

Queue



Queue – Định nghĩa

- ▶ Hàng đợi là một cấu trúc:
 - ▶ Gồm nhiều phần tử có thứ tự
 - ▶ Hoạt động theo cơ chế “Vào trước, ra trước” (FIFO - First In First Out)



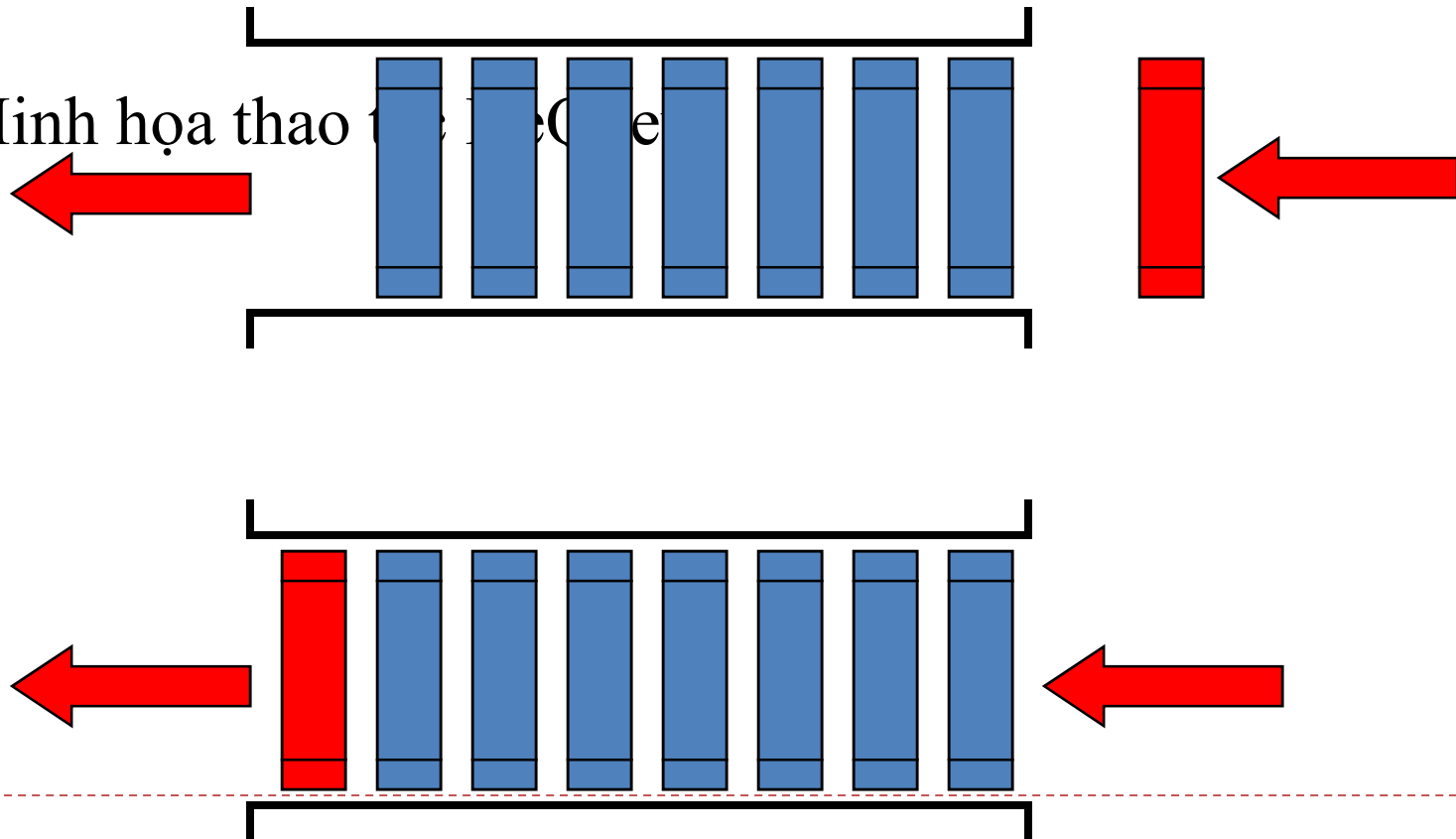
Queue – Định nghĩa

- ▶ Các thao tác cơ bản trên hàng đợi:
 - ▶ InitQueue: khởi tạo hàng đợi rỗng
 - ▶ IsEmpty: kiểm tra hàng đợi rỗng?
 - ▶ IsFull: kiểm tra hàng đợi đầy?
 - ▶ EnQueue: thêm 1 phần tử vào cuối hàng đợi, có thể làm hàng đợi đầy
 - ▶ DeQueue: lấy ra 1 phần tử từ đầu Queue, có thể làm Queue rỗng

Queue

- ▶ Minh họa thao tác EnQueue

- ▶ Minh họa thao tác DeQueue




Cách xây dựng Queue

- ▶ Sử dụng mảng một chiều
- ▶ Sử dụng danh sách liên kết đơn

Queue – Sử dụng mảng

- ▶ Dùng 1 mảng (**QArray**) để chứa các phần tử.
- ▶ Dùng 1 số nguyên (**QMax**) để lưu số phần tử tối đa trong hàng đợi
- ▶ Dùng 2 số nguyên (**QFront**, **QRear**) để xác định vị trí đầu, cuối hàng đợi
- ▶ Dùng 1 số nguyên (**QNumItems**) để lưu số phần tử hiện có trong hàng đợi

Queue – Sử dụng mảng

	0	1	2	3	4	5	6
Qarray 		37	22	15	3		
QMax = 7							
QNumItems = 4							
QFront = 1							
QRear = 4							

Queue số nguyên – Sử dụng mảng

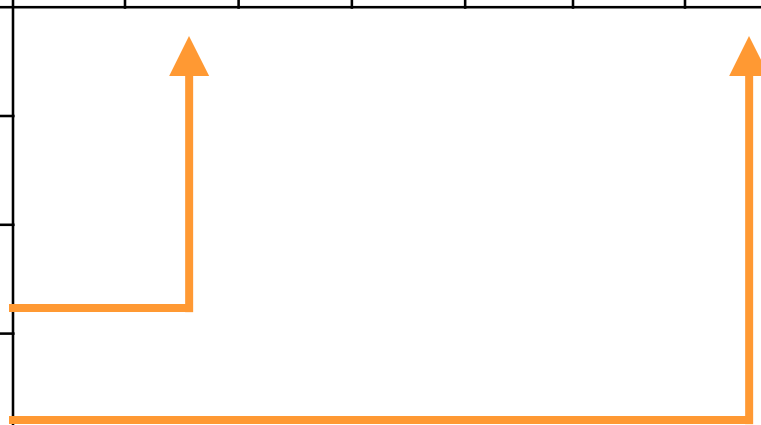
```
typedef struct QUEUE
{
    int*   QArray;
    int     QMax;
    int     QNumItems;
    int     QFront;
    int     QRear;
```

```
};
```



Queue số nguyên – Sử dụng mảng

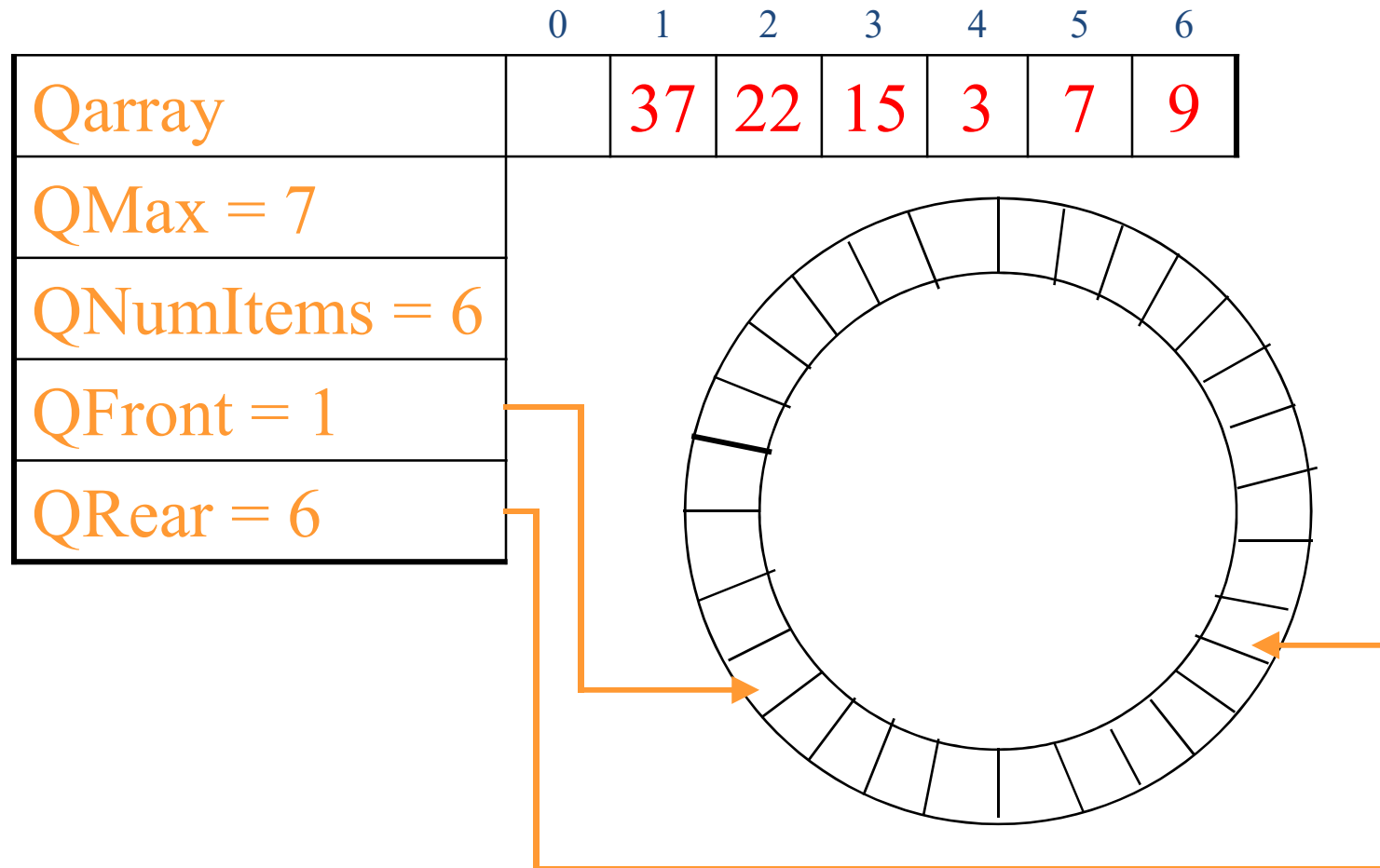
- ▶ Khi thêm nhiều phần tử sẽ xảy ra hiện tượng “tràn giả”

	0	1	2	3	4	5	6
Qarray		37	22	15	3	7	9
QMax = 7							
QNumItems = 6							
QFront = 1							
QRear = 6							

- ▶ Giải pháp? Nối dài mảng (mảng động) hay sử dụng một mảng vô cùng lớn?

Queue số nguyên – Sử dụng mảng

- Xử lý mảng như một danh sách liên kết vòng



Queue số nguyên – Sử dụng mảng

VD: Cho queue như sau

Chỉ số mảng	0	1	2	3	4	5	6
QArray		11	7	19	21	81	
QMax	7						
QNumItems	5						
QFront	1						
QRear	5						



Queue số nguyên – Sử dụng mảng

1. Thêm giá trị **123** vào hàng đợi

Chỉ số mảng	0	1	2	3	4	5	6
QArray		11	7	19	21	81	123
QMax	7						
QNumItems	6						
QFront	1						
QRear	6						



Queue số nguyên – Sử dụng mảng

2. Lấy một phần tử khỏi hàng đợi

Chỉ số mảng	0	1	2	3	4	5	6
QArray		11	7	19	21	81	123
QMax	7						
QNumItems	5						
QFront	2						
QRear	6						



Queue số nguyên – Sử dụng mảng

3. Thêm giá trị **456** vào hàng đợi

Chỉ số mảng	0	1	2	3	4	5	6
QArray	456	11	7	19	21	81	123
QMax	7						
QNumItems	6						
QFront	2						
QRear	0						



Queue số nguyên – Sử dụng mảng

```
bool InitQueue(Queue &q, int MaxItem)
{
    q.QArray = new int[MaxItem];
    if (q.QArray == NULL)
        return false;
    q.QMax = MaxItem;
    q.QNumItems = 0;
    q.QFront = q.QRear = -1;
    return true;
}
```

Queue số nguyên – Sử dụng mảng

```
bool IsEmpty(QUEUE q)
{
    if (q.QNumItems == 0)
        return true;
    return false;
}
```

Queue số nguyên – Sử dụng mảng

```
bool IsFull(QUEUE q)
{
    if (q.QMax == q.QNumItems)
        return true;
    return false;
}
```

Queue số nguyên – Sử dụng mảng

```
bool EnQueue(Queue &q, int newitem)
{
    if (IsFull(q))
        return false;
    q.QRear++;
    if (q.QRear==q.QMax)
        q.QRear = 0;
    q.QArray[q.QRear] = newitem;
    if (q.QNumItems==0)
        q.QFront = 0;
    q.QNumItems++;
    return true;
}
```

Queue số nguyên – Sử dụng mảng

```
bool DeQueue(Queue &q, int &itemout)
{
    if (IsEmpty(q))
        return false;
    itemout = q.QArray[q.QFront];
    q.QFront++;
    q.QNumItems--;
    if (q.QFront==q.QMax)
        q.QFront = 0;
    if (q.QNumItems==0)
        q.QFront = q.QRear = -1;
    return true;
}
```

Queue số nguyên – Sử dụng mảng

```
bool QueueFront(const QUEUE &q, int &itemout)
{
    if (IsEmpty(q))
        return false;
    itemout = q.QArray[q.QFront];
    return true;
}
```

Queue số nguyên – Sử dụng mảng

```
bool QueueRear(const QUEUE &q, int &itemout)
{
    if (IsEmpty(q))
        return false;
    itemout = q.QArray[q.QRear];
    return true;
}
```


Queue – Ví dụ ứng dụng

- ▶ Quản lý việc thực hiện các tác vụ (task) trong môi trường xử lý song song
- ▶ Hàng đợi in ấn các tài liệu
- ▶ Vùng nhớ đệm (buffer) dùng cho bàn phím
- ▶ Quản lý thang máy

Queue – Sử dụng DSLK

```
typedef struct tagNODE
{
    int data;
    tagNODE* pNext;
} NODE, *PNODE;
```

```
typedef struct tagQUEUE
{
    int NumItems;
    PNODE pFront, pRear;
} QUEUE;
```

Queue – Sử dụng DSLK

- ▶ Các thao tác cơ bản

bool InitQueue(QUEUE &q);

bool IsEmpty(**const** QUEUE &q);

bool IsFull(**const** QUEUE &q);

bool EnQueue(QUEUE &q, **int** newitem);

bool DeQueue(QUEUE &q, **int&** itemout);

Queue – Sử dụng DSLK

```
bool InitQueue(QUEUE &q)
{
    q.NumItems = 0;
    q.pFront = q.pRear = NULL;
    return true;
}
```

Queue – Sử dụng DSLK

```
bool IsEmpty(const QUEUE& q)
{
    return (q.NumItems==0);
}
```

Queue – Sử dụng DSLK

```
bool IsFull(const QUEUE &q)
{
    PNODE tmp = new NODE;
    if (tmp==NULL)
        return true;
    delete tmp;
    return false;
}
```

Queue – Sử dụng DSLK

```
bool EnQueue(Queue &q, int newitem)
{
    if (IsFull(q))
        return false;
    PNODE p = new NODE;
    p->data = newitem;
    p->pNext = NULL;
    if (q.pFront==NULL && q.pRear==NULL)
        q.pFront = q.pRear = p;
    else
    {
        q.pRear->pNext = p;
        q.pRear = p;
    }
    q.NumItems++;
    return true;
}
```

Queue – Sử dụng DSLK

```
bool DeQueue(Queue &q, int &itemout)
{
    if (IsEmpty(q))
        return false;
    PNODE p = q.pFront;
    q.pFront = p->pNext;
    itemout = p->data;
    q.NumItems--;
    delete p;
    if (q.NumItems==0)
        InitQueue(q);
    return true;
}
```


INDUSTRIAL UNIVERSITY OF HO CHI MINH CITY



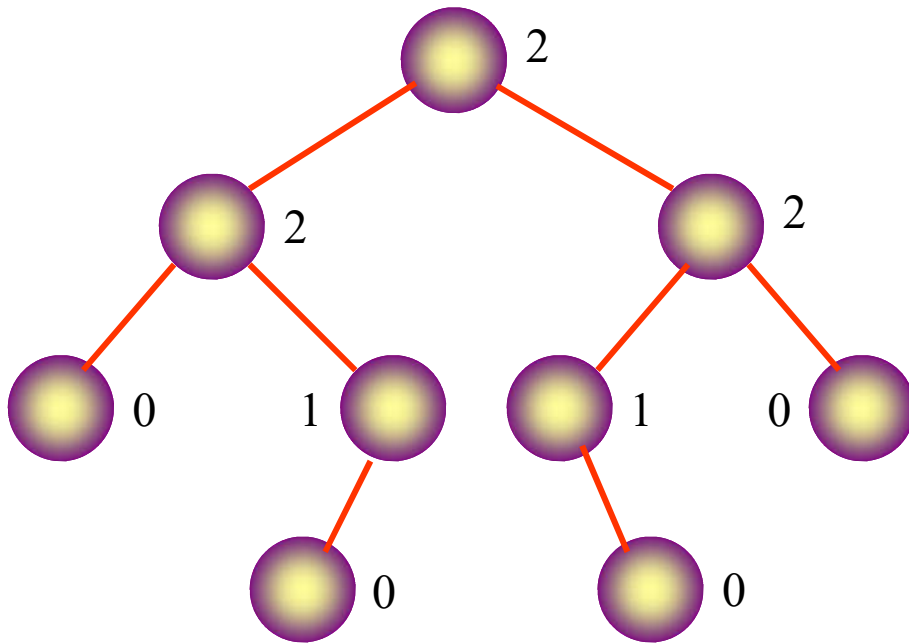
Data structures and algorithms

Tree structure

Nội dung

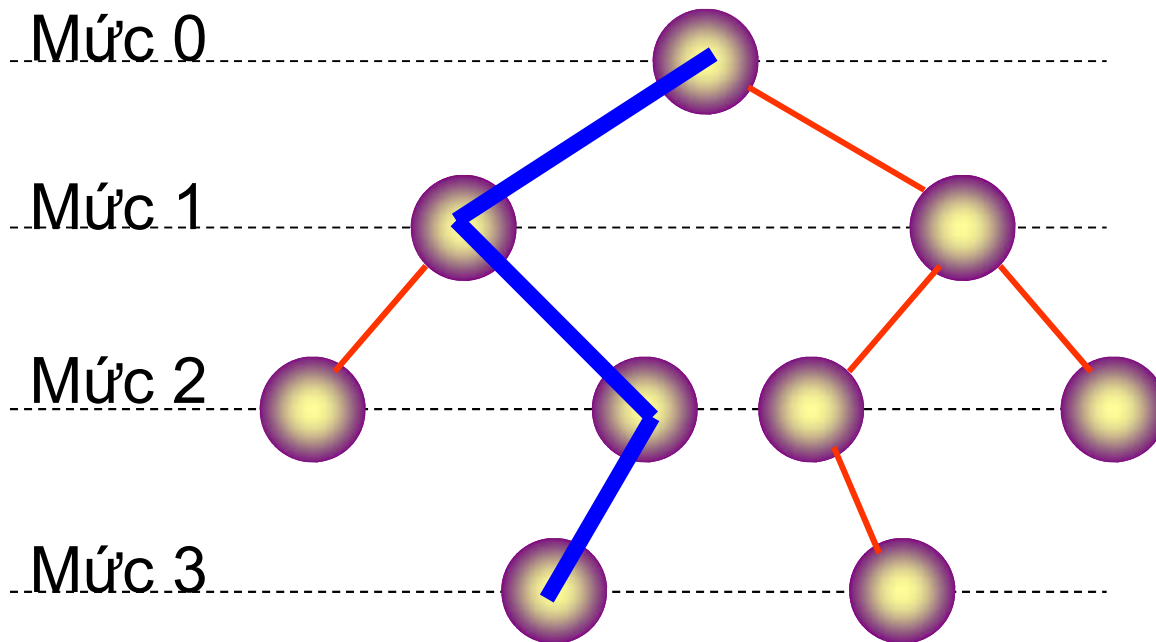
1. Khái niệm
2. Đặc điểm
3. Hình dạng
4. Định nghĩa kiểu dữ liệu
5. Các lưu ý khi cài đặt
6. Các thao tác

Khái niệm



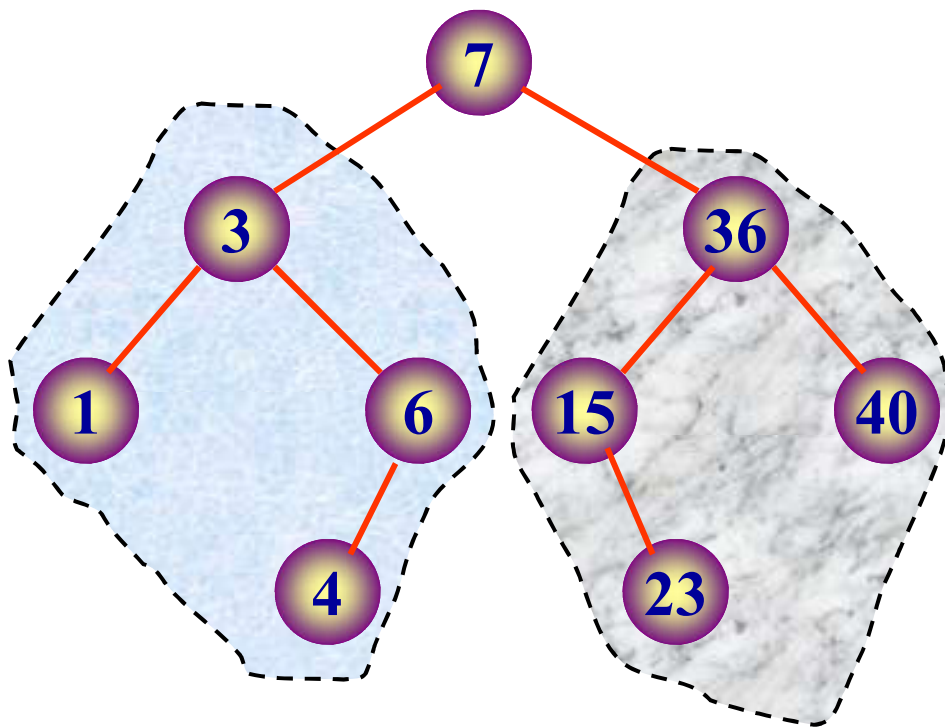
- ▶ **Bậc của một nút:** là số cây con của nút đó
- ▶ **Nút gốc:** là nút không có nút cha
- ▶ **Nút lá:** là nút có bậc bằng 0
- ▶ **Nút nhánh:** là nút có bậc khác 0 và không phải là gốc

Khái niệm



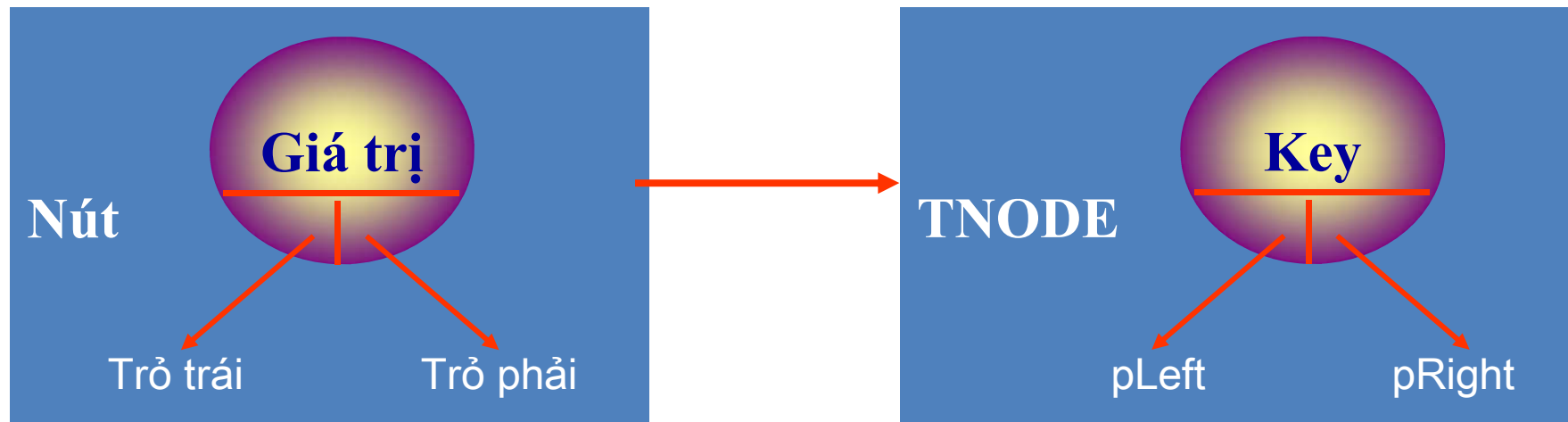
- ▶ **Độ dài đường đi từ gốc đến nút x:** là số nhánh cần đi qua kể từ gốc đến x
- ▶ **Độ cao của cây:** Độ dài đường đi từ gốc đến nút lá ở mức thấp nhất

Đặc điểm cây nhị phân tìm kiếm



- ▶ Là cây **nhị phân**
- ▶ Giá trị của một node bất kỳ luôn **lớn hơn giá trị của tất cả các node bên trái và nhỏ hơn giá trị tất cả các node bên phải**
- ➔ Nút có giá trị nhỏ nhất nằm ở trái nhất của cây
- ➔ Nút có giá trị lớn nhất nằm ở phải nhất của cây

Định nghĩa kiểu dữ liệu



```
typedef struct TNode
{
    <Data> Key;
    struct TNode *pLeft, *pRight;
} *TREE;
```

Ví dụ khai báo

```
typedef struct TNODE
{
    int Key;
    struct TNODE *pLeft, *pRight;
} *TREE;
```

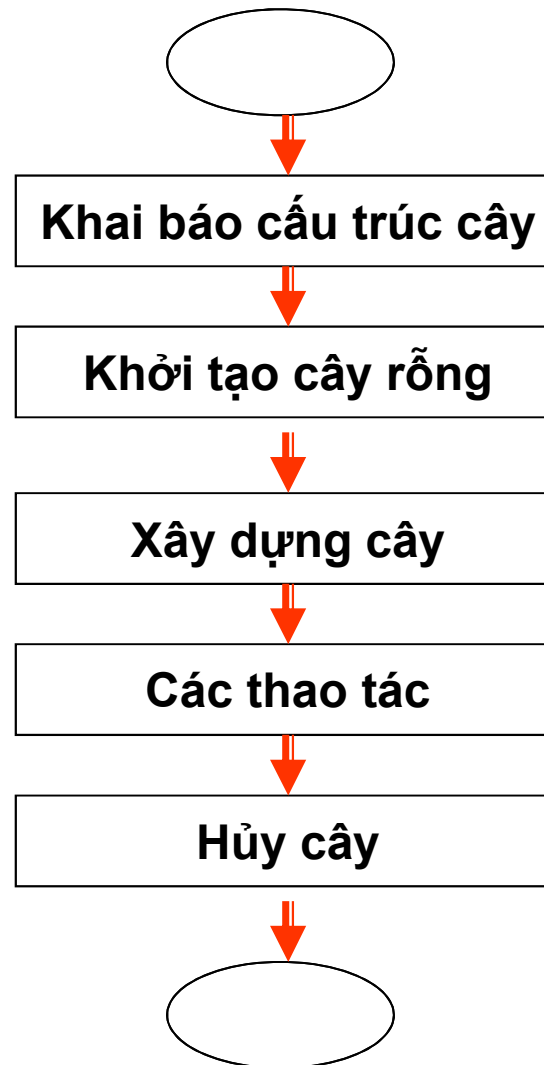
Các lưu ý khi cài đặt

Bước 1: Khai báo kiểu dữ liệu biểu diễn cây

Bước 2: Xây dựng hàm đưa dữ liệu (nhập) vào cây

Bước 3: Xây dựng các thao tác duyệt, tìm kiếm, huỷ, ...

Cấu trúc chương trình

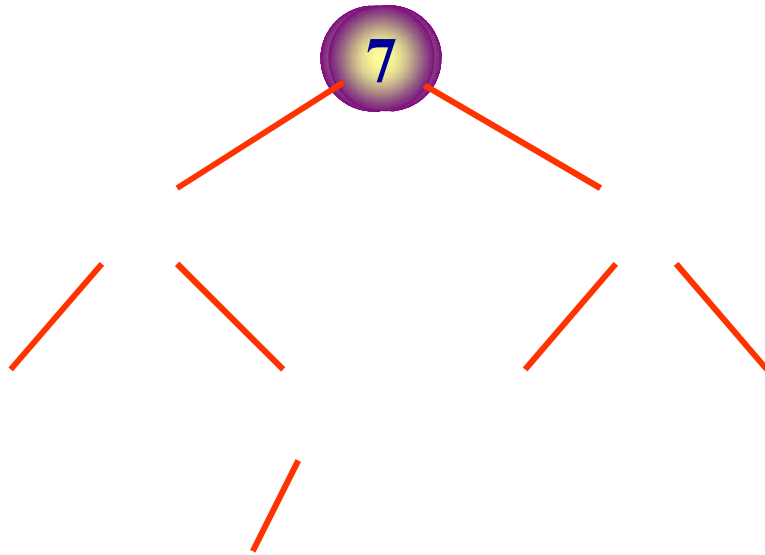


Các thao tác

1. Tạo cây
2. Duyệt cây
3. Cho biết các thông tin của cây
4. Tìm kiếm
5. Xoá node trên cây

Tạo cây

7	36	3	1	6	4	15	40
----------	-----------	----------	----------	----------	----------	-----------	-----------



- Nếu node cần thêm **nhỏ hơn** node đang xét thì thêm về **bên trái**
- Ngược lại thì thêm về **bên phải**

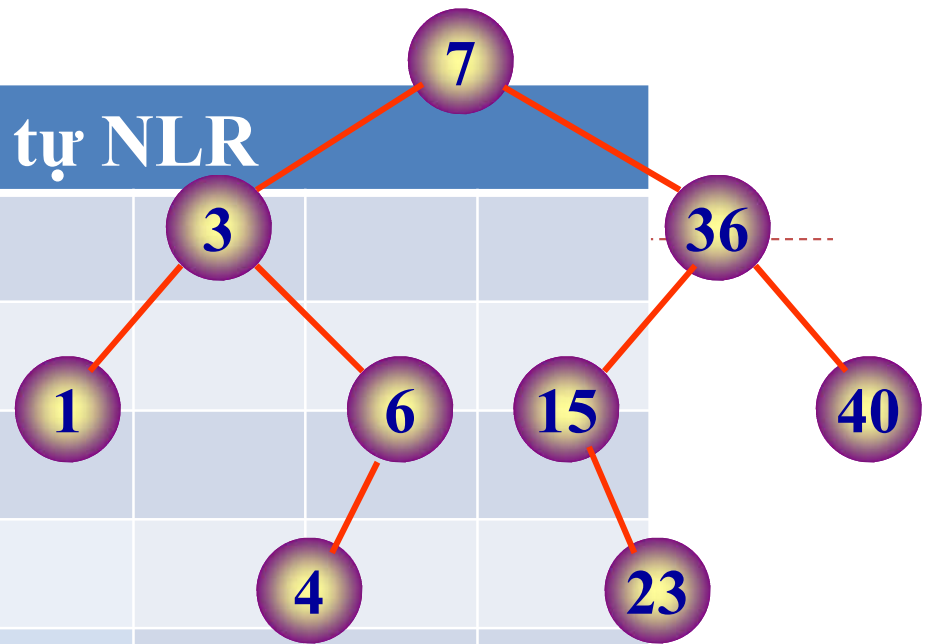
Hàm tạo cây

```
int ThemNut (TREE & t, int x)
{  if(t!=NULL)
    {  if(x==t->Key)    return 0; // x đã có trong cây
        else
        {  if(x<t->Key)    return ThemNut(t->pLeft, x);
            else          return ThemNut(t->pRight, x);
        }
    }
    else
    {  t=new TNODE;
        if(t==NULL)          return -1; //không đủ bộ nhớ
        t->Key=x;
        t->pLeft=t->pRight=NULL;
        return 1; //thêm x thành công
    }
}
```

Duyệt cây

Thứ tự trước	(N LR)
Thứ tự giữa	(L N R)
Thứ tự sau	(LR N)

Bước	Kết quả duyệt theo thứ tự NLR									
1	7	L7	R7					3		36
2		3	L3	R3	R7			1	6	15
3			1	R3	R7				4	23
4				6	L6	R7				
5					4	R7				
6						36	L36	R36		
7							15	R15	R36	
8								23	R36	
9									40	
KQ	7	3	1	6	4	36	15	23	40	



Hàm duyệt NLR

Tại node t đang xét, nếu khác rỗng thì

- ▶ In giá trị của t
- ▶ Duyệt cây con bên trái của t theo thứ tự NLR
- ▶ Duyệt cây con bên phải của t theo thứ tự NLR

```
void NLR (TREE t)
{
    if(t!=NULL)
    {
        cout<<t->Key<<"\t";
        NLR(t->pLeft);
        NLR(t->pRight);
    }
}
```

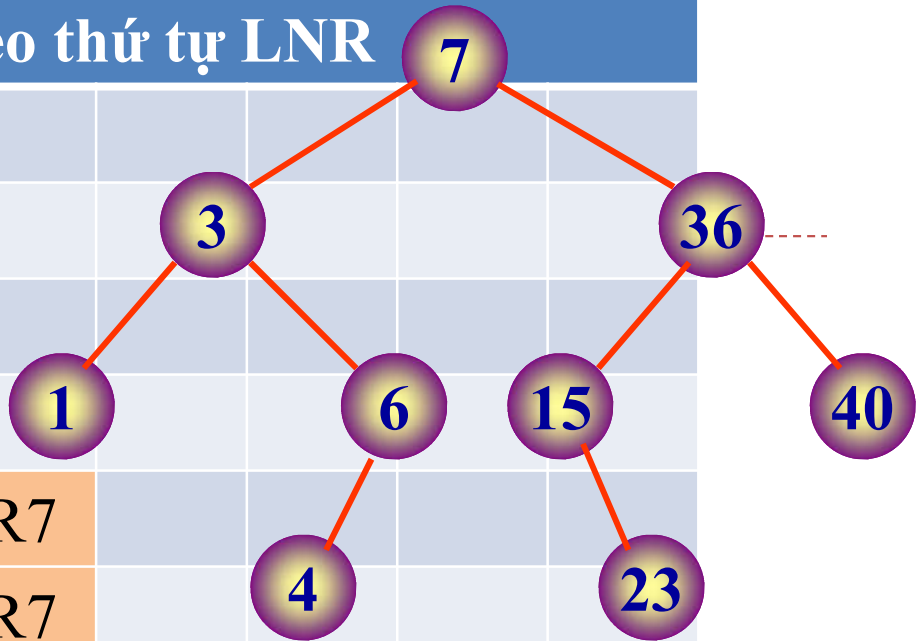
Bài tập

Vẽ cây nhị phân tìm kiếm theo thứ tự nhập từ trái sang phải và **duyet cây theo thứ tự trước**:

- a. 27; 19; 10; 21; 35; 25; 41; 12; 46; 7
- b. H; B; C; A; E; D; Z; M; P; T
- c. Huế; Đà Nẵng; Hà Nội; Vĩnh Long; Cần Thơ; Sóc Trăng; Nha Trang; Đồng Nai; Vũng Tàu; An Giang; Tiền Giang; Bình Dương; Hải Dương



Bước	Kết quả duyệt theo thứ tự LNR									
1	L7	7	R7							
2	L3	3	R3	7	R7					
3	1	3	R3	7	R7					
4		3	R3	7	R7					
5			L6	6	7	R7				
6			4	6	7	R7				
7				6	7	R7				
8					7	R7				
9						L36	36	R36		
10						15	R15	36	R36	
11							23	36	R36	
12								36	R36	
13									40	
KQ	1	3	4	6	7	15	23	36	40	



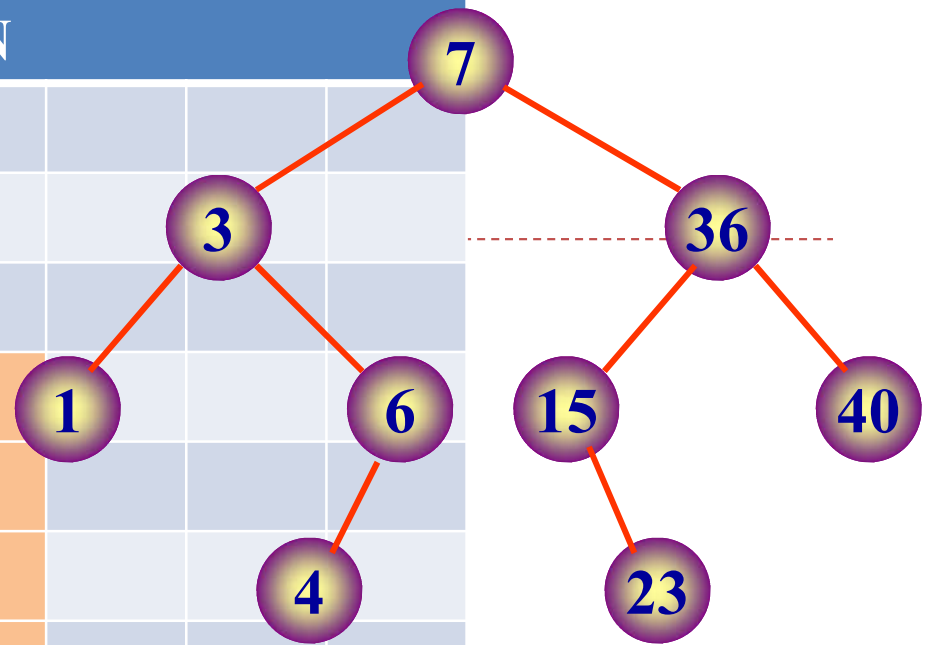
Hàm duyệt LNR

Tại node t đang xét, nếu khác rỗng thì

- ▶ Duyệt cây con bên trái của t theo thứ tự LNR
- ▶ In giá trị của t
- ▶ Duyệt cây con bên phải của t theo thứ tự LNR

```
void LNR (TREE t)
{
    if(t!=NULL)
    {
        LNR(t->pLeft);
        cout<<t->Key<<" ";
        LNR(t->pRight);
    }
}
```

Bước	Kết quả duyệt theo thứ tự LRN								
1	L7	R7	7						
2	L3	R3	3	R7	7				
3	1	R3	3	R7	7				
4		L6	6	3	R7	7			
5		4	6	3	R7	7			
6			6	3	R7	7			
7				3	R7	7			
8					L36	R36	36	7	
9					R15	15	R36	36	7
10					23	15	R36	36	7
11						15	R36	36	7
12							40	36	7
13								36	7
14									7
KQ	1	4	6	3	23	15	40	36	7



Hàm duyệt LRN

Tại node t đang xét, nếu khác rỗng thì

- ▶ Duyệt cây con bên trái của t theo thứ tự LRN
- ▶ Duyệt cây con bên phải của t theo thứ tự LRN
- ▶ In giá trị của t

```
void LRN (TREE t)
{
    if(t!=NULL)
    {
        LRN(t->pLeft);
        LRN(t->pRight);
        cout<<t->Key<<" ";
    }
}
```



Bài tập

- ▶ **Bài 4** Vẽ cây nhị phân tìm kiếm theo thứ tự nhập:

27, 19, 10, 21, 3, 15, 41, 50, 30, 27

Hãy duyệt cây trên theo thứ tự giữa

- ▶ **Bài 5** Vẽ cây nhị phân tìm kiếm theo thứ tự nhập:

H, B, C, A, E, D, T, M, X, O

Hãy duyệt cây trên theo thứ tự sau



Vấn đề cần quan tâm

Tạo cây từ kết quả duyệt NLR

- ▶ Chọn giá trị đầu tiên làm node gốc
- ▶ Lần lượt đưa các giá trị còn lại từ trái sang phải vào cây theo nguyên tắc tạo cây

Tạo cây từ kết quả duyệt LRN

- ▶ Chọn giá trị cuối cùng làm node gốc
- ▶ Lần lượt đưa các giá trị còn lại từ phải sang trái vào cây theo nguyên tắc tạo cây



Vấn đề cần quan tâm

Tạo cây từ kết quả duyệt LNR

- ▶ Gọi r : Số lượng giá trị cho trước
- ▶ Gọi $m = r \text{ div } 2$: Giá trị ở giữa
- ▶ Chọn giá trị thứ m làm node gốc
- ▶ Lần lượt đưa các giá trị bắt đầu từ vị trí $m-1$ lùi về trái vào cây theo nguyên tắc tạo cây
- ▶ Lần lượt đưa các giá trị bắt đầu từ vị trí $m+1$ đến cuối vào cây theo nguyên tắc tạo cây



Bài tập

Bài 6 Vẽ cây nhị phân tìm kiếm T biết rằng khi duyệt cây T theo thứ tự NLR thì được dãy sau: 9, 4, 1, 3, 8, 6, 5, 7, 10, 14, 12, 13, 16, 19

- ▶ Hãy duyệt cây T trên theo thứ tự LRN
- ▶ Liệt kê các nút lá của cây. Liệt kê các nút nhánh của cây



Bài tập

Bài 7 Vẽ cây nhị phân tìm kiếm T biết rằng khi duyệt cây T theo thứ tự LRN thì được dãy sau: 1, 4, 7, 5, 3, 16, 18, 15, 29, 25, 30, 20, 8

- ▶ Hãy duyệt cây T trên theo thứ tự NLR
- ▶ Cây T có chiều cao là bao nhiêu? Tìm các đường đi từ gốc có độ dài là 4 trên cây



Hàm nhập dữ liệu vào cây

```
void Nhap(TREE &t)
{
    int x;
    do{
        cout<<"Nhap gia tri: ";
        cin>>x;
        int kq=ThemNut(t, x);
        if(kq==0||kq==-1)
            break;
    }while (true);
}
```



Hàm main gọi thao tác duyệt LNR

```
void main()
{
    TREE t;
    t=NULL;
    Nhap(t);
    cout<<“Duyet cay theo thu tu giua: “;
    LNR(t);

    Huy(t);
}
```



Các thao tác

1. Tạo cây
2. Duyệt cây
3. Cho biết các thông tin của cây
4. Tìm kiếm
5. Xoá node trên cây



Các thao tác

1. Tạo cây
2. Duyệt cây
3. Cho biết các thông tin của cây
4. Tìm kiếm
5. Xoá node trên cây



Cho biết các thông tin của cây

1. Số node lá (node bậc 0)
2. Số node có 1 cây con (node bậc 1)
3. Số node chỉ có 1 cây con phải
4. Số node chỉ có 1 cây con trái
5. Số node có 2 cây con (node bậc 2)
6. Độ cao của cây
7. Số node của cây
8. Các node trên từng mức của cây
9. Độ dài đường đi từ gốc đến node x



Các thao tác

1. Tạo cây
2. Duyệt cây
3. Cho biết các thông tin của cây
4. Tìm kiếm
5. Xoá node trên cây



Các thao tác

1. Tạo cây
2. Duyệt cây
3. Cho biết các thông tin của cây
4. Tìm kiếm
5. Xoá node trên cây

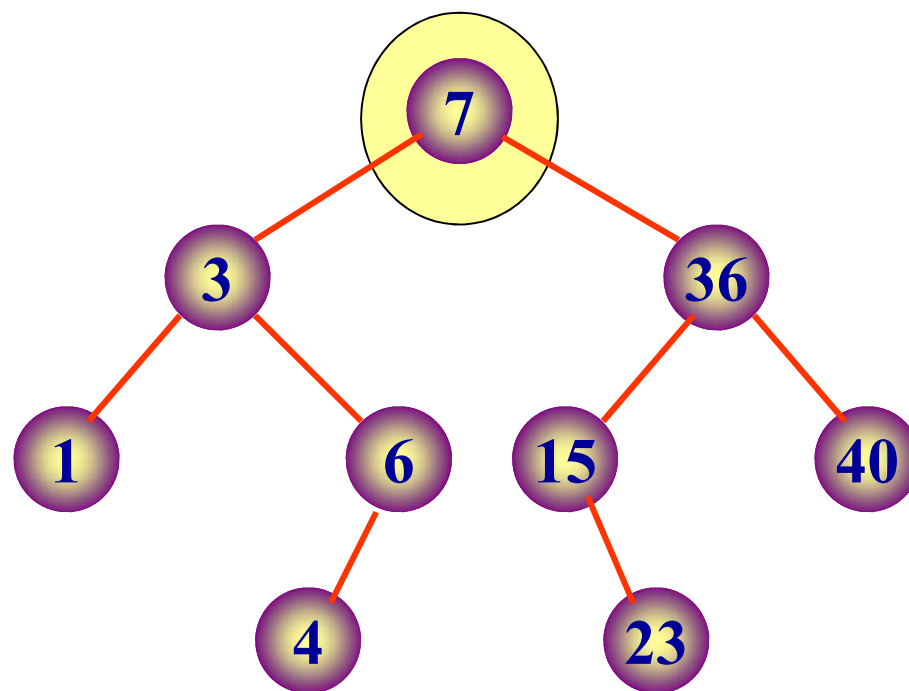


Tìm kiếm

1. Tìm x
2. Tìm min
3. Tìm min của cây con bên phải
4. Tìm max
5. Tìm max của cây con bên trái



Ví dụ tìm $x = 23$



Các thao tác

1. Tạo cây
2. Duyệt cây
3. Cho biết các thông tin của cây
4. Tìm kiếm
5. Xoá node trên cây

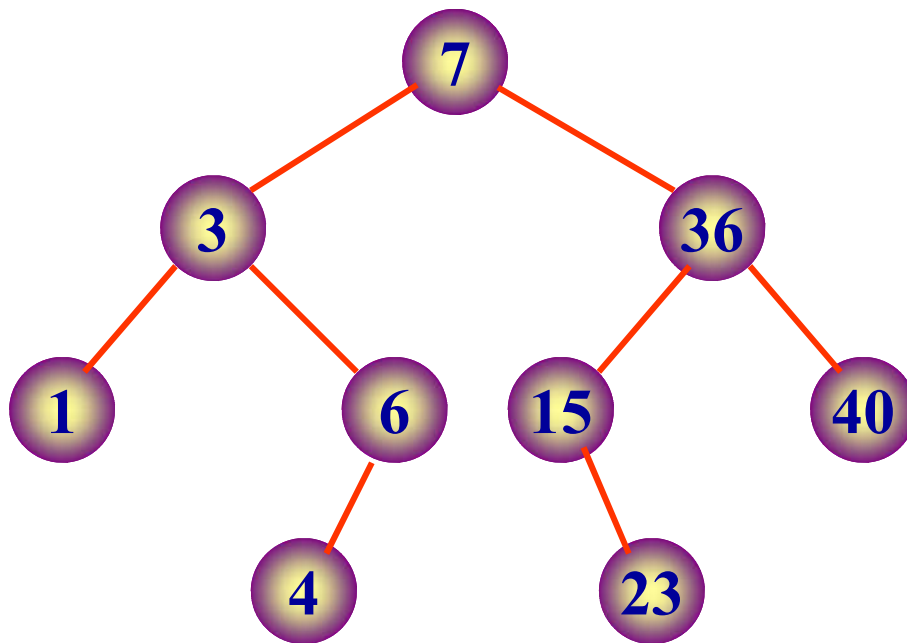


Các thao tác

1. Tạo cây
2. Duyệt cây
3. Cho biết các thông tin của cây
4. Tìm kiếm
5. Xoá node trên cây



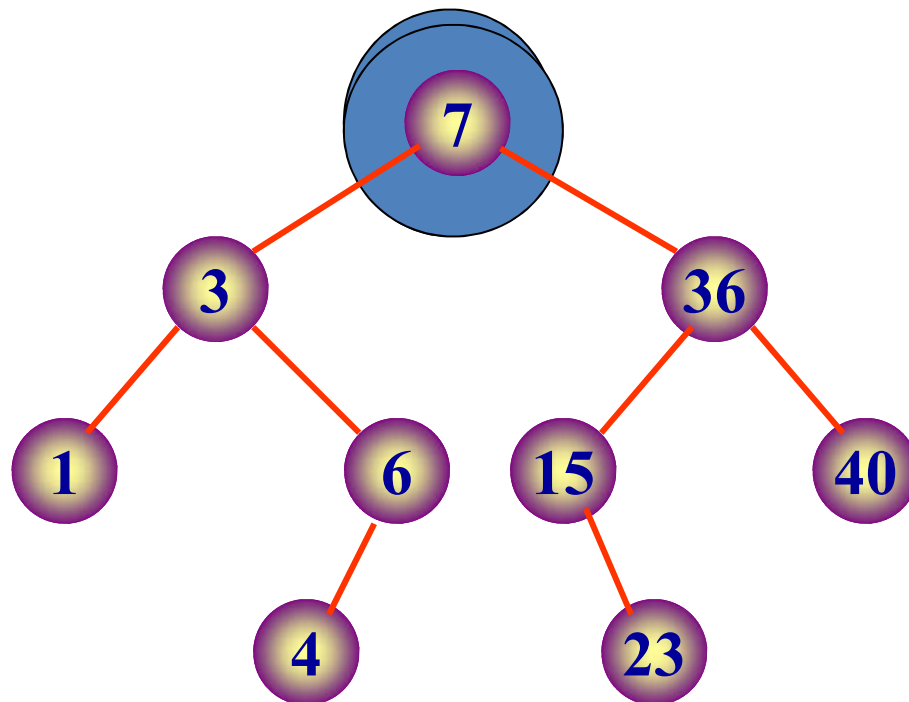
Xóa node trên cây



1. Node lá
2. Node có 1 cây con
3. Node có 2 cây con



Xóa node lá

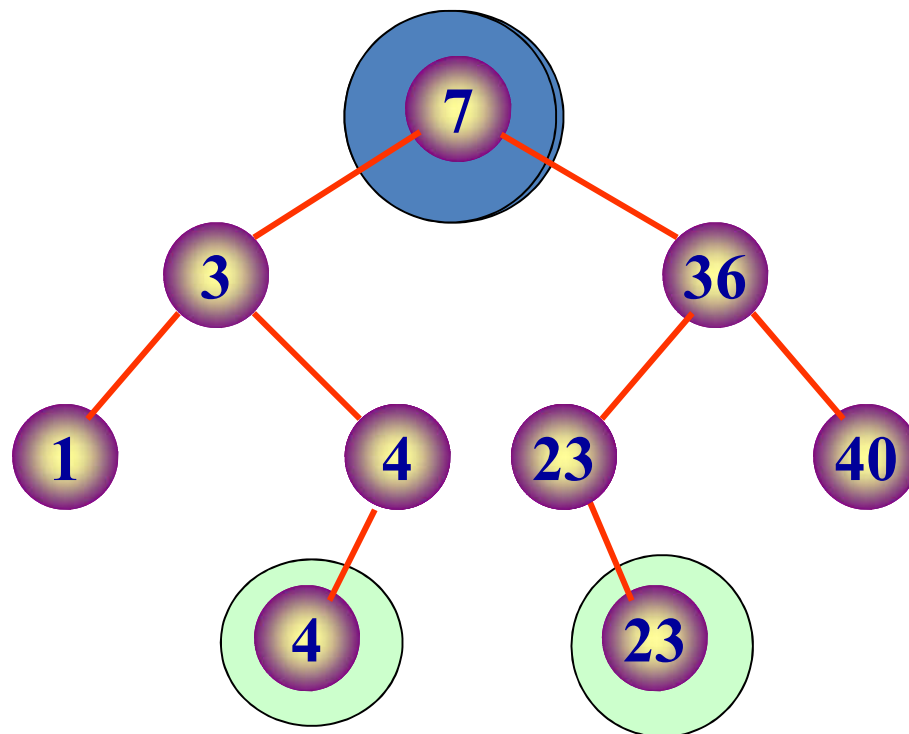


Xóa 1

Xóa 23



Xóa node 1 cây con

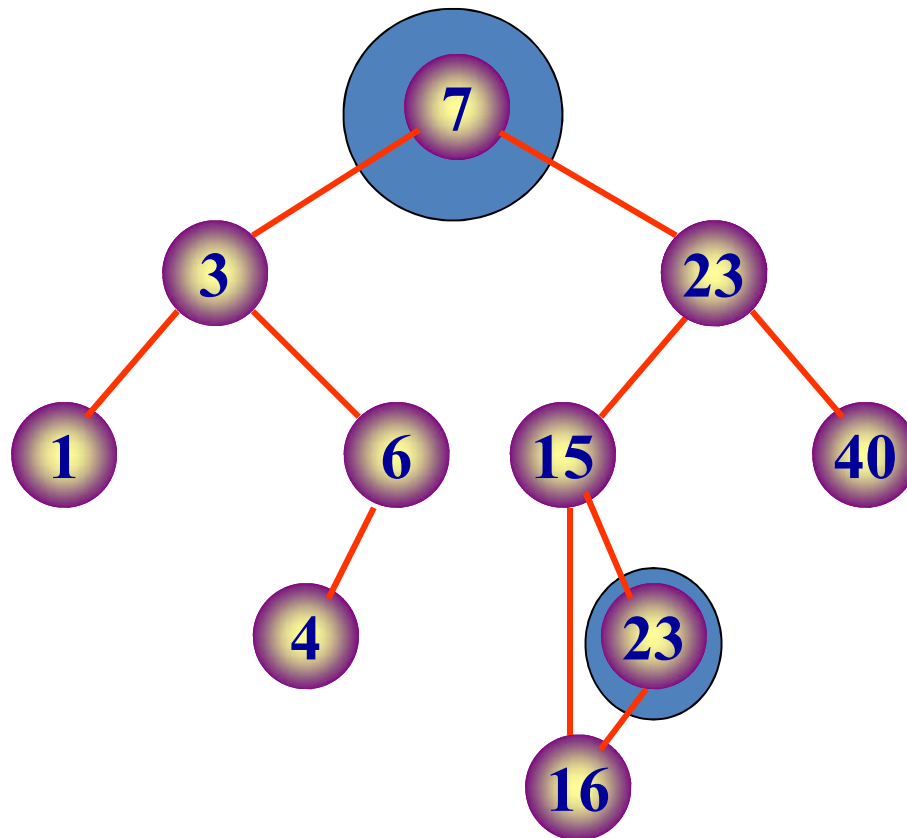


Xóa 6

Xóa 15



Xóa node 2 cây con



Tìm node thế mạng

► **Cách 1:** Tìm node trái nhất của cây con phải

(min của T->Right)

► **Cách 2:** Tìm node phải nhất của cây con trái

(max của T->Left)

Xóa 36 (Cách 2)



Cho dãy số theo thứ tự nhập từ trái sang phải: **20, 15, 35, 30, 11, 13, 17, 36, 47, 16, 38, 28, 14**

- ▶ Vẽ cây nhị phân tìm kiếm cho dãy số trên
- ▶ Cho biết kết quả duyệt cây trên theo thứ tự trước, giữa và sau
- ▶ Cho biết độ cao của cây, các nút lá, các nút có bậc 2
- ▶ Vẽ lại cây sau khi thêm nút: 25 và 91
- ▶ Trình bày từng bước và vẽ lại cây sau khi lần lượt xoá các nút: **11** và **35**

