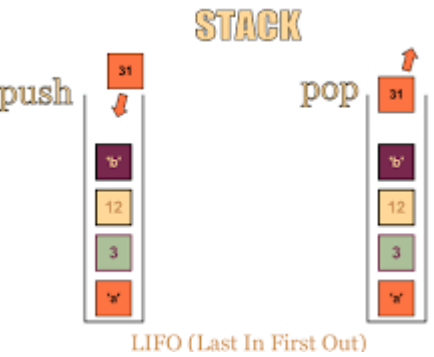
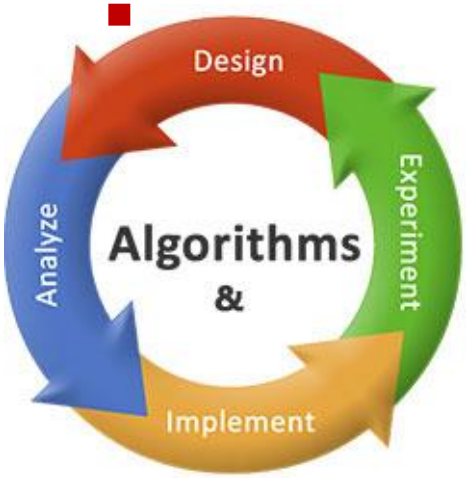
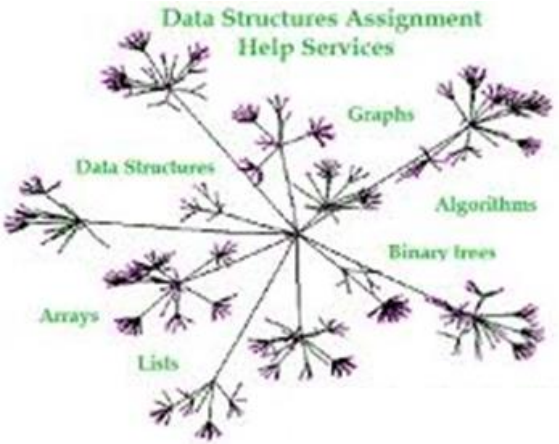


CẤU TRÚC DỮ LIỆU

&

GIẢI THUẬT





Lê Văn Hạnh

levanhanhvn@gmail.com

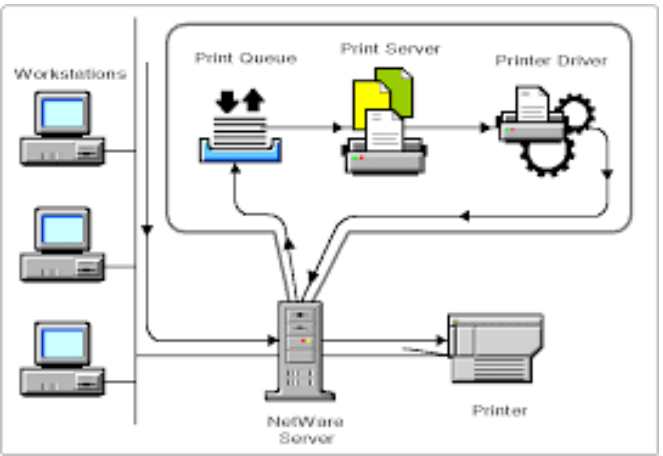
NỘI DUNG MÔN HỌC

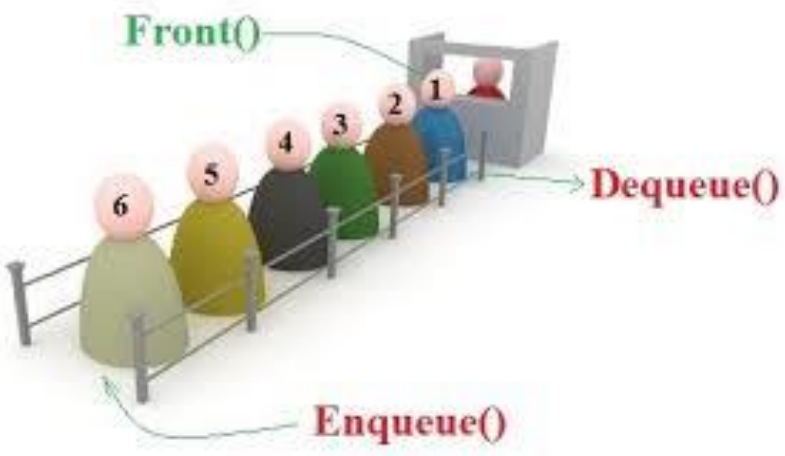
- Chương 1: Ôn tập ngôn ngữ lập trình C
- Chương 2: Kiểu dữ liệu con trỏ
- Chương 3: Tổng quan về cấu trúc dữ liệu và giải thuật
- Chương 4: Danh sách kề (Danh sách tuyến tính)
- Chương 5: Các giải thuật tìm kiếm trên danh sách kề
- Chương 6: Các giải thuật sắp xếp trên danh sách kề
- Chương 7: Danh sách liên kết động (*Linked List*)
- Chương 8: Ngăn xếp *(Stack)*
- **Chương 9: Hàng đợi *(Queue)***
- Chương 10: Cây nhị phân tìm kiếm *(Binary Search Tree)*
- Chương 11: Cây NPTK cân bằng *(Balanced binary search tree – AVL tree)*
- Chương 12: Bảng băm *(Hash Table)*

Chương 9

HÀNG ĐỢI

(Queue)

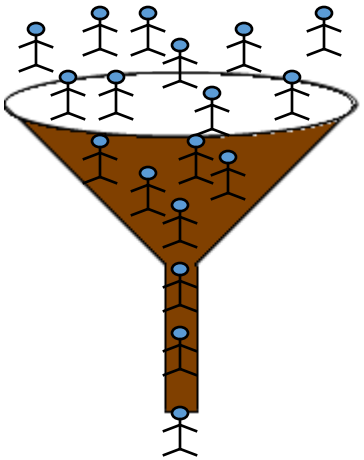


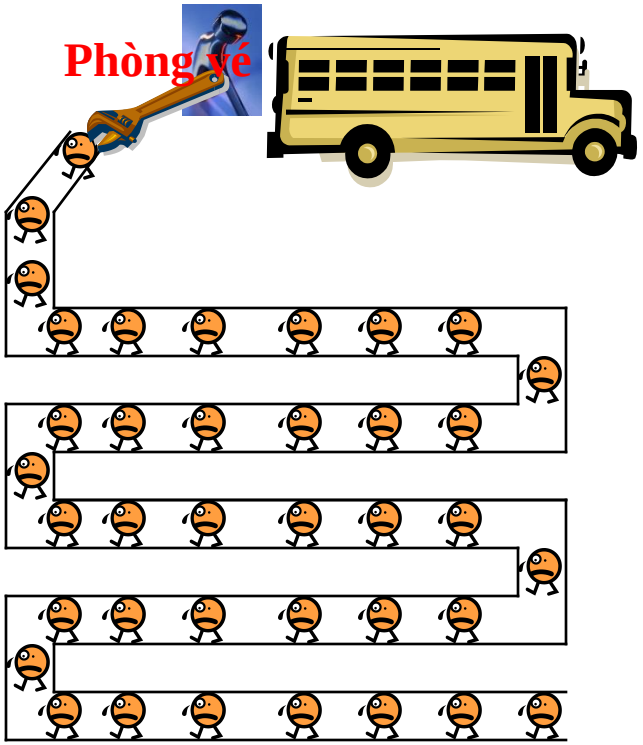


NỘI DUNG CHƯƠNG

1. Giới thiệu
2. Một số ứng dụng của Queue
3. Hàng đợi có ưu tiên (*Priority Queue*)
4. Cài đặt queue
5. Sử dụng danh sách kề làm Queue
6. Sử dụng danh sách liên kết Queue
7. Thực hành

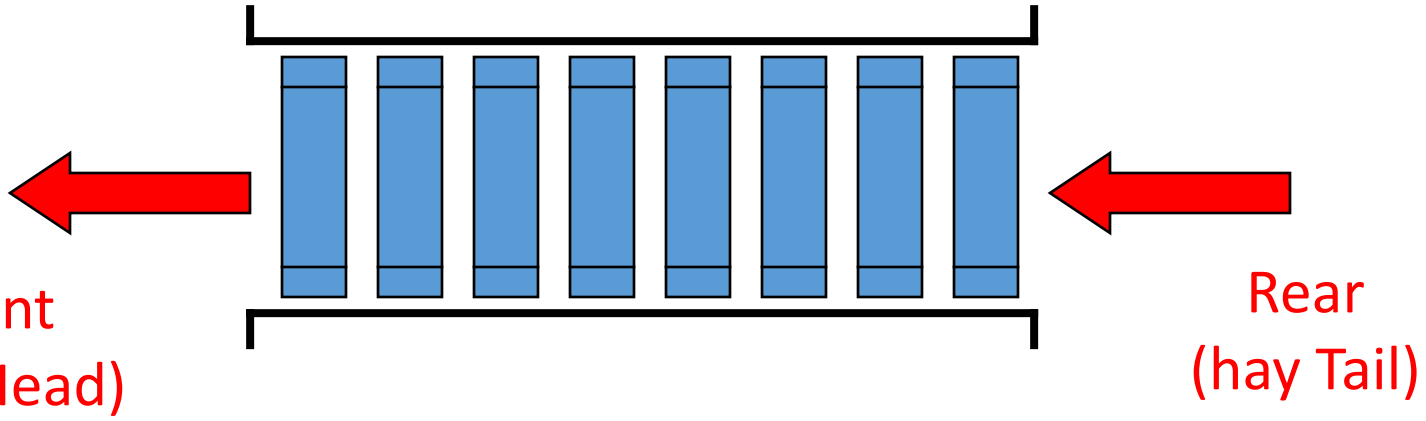
1. GIỚI THIỆU





1. GIỚI THIỆU

- Queue là kiểu danh sách tuyến tính, hoạt động theo cơ chế "***Vào trước, ra trước***" (FIFO - First In First Out).
- Queue là một cấu trúc:
  - Gồm nhiều phần tử có thứ tự.
  - Phép bổ sung một phần tử được thực hiện ở một đầu gọi là rear (hay tail)
  - Phép loại bỏ một phần tử được thực hiện ở đầu kia gọi là (front) của hàng đợi.

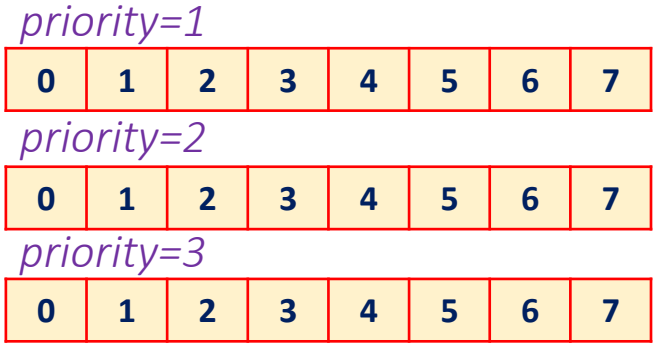


1. GIỚI THIỆU

- Một số dạng của Queues:
  - **Hàng đợi tuyến tính** (*Linear Queues*): Tổ chức *queue* theo nghĩa thông thường.

  - **Hàng đợi vòng** (*Circular Queues*): Giải quyết việc thiếu bộ nhớ khi sử dụng *queue*.

  - **Hàng đợi ưu tiên** (*Priority Queues*):
    - Mỗi phần tử có kết hợp thêm thông tin về độ ưu tiên.
    - Khi chương trình cần lấy một phần tử khỏi *queue*, nó sẽ xét những phần tử có độ ưu tiên cao trước.



2. Một số ứng dụng của Queue

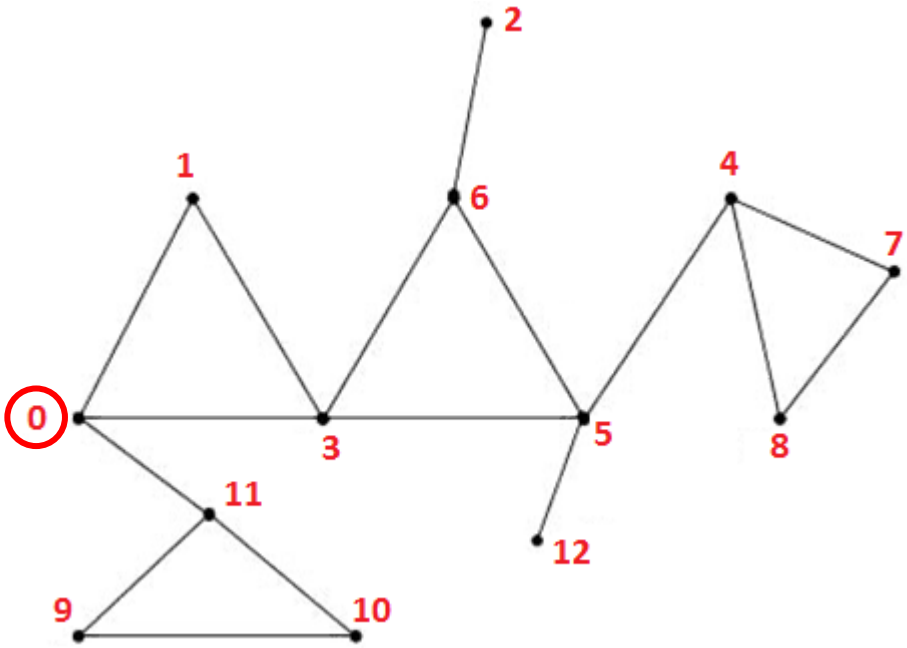
2.1. Tìm kiếm theo chiều rộng trên đồ thị *(BFS - Breadth First Search)*

Cho đồ thị như hình bên.

Xuất phát từ đỉnh 0,

yêu cầu duyệt qua

tất cả các đỉnh



Queue

0	-1	-1	-1	-1	-1	-1

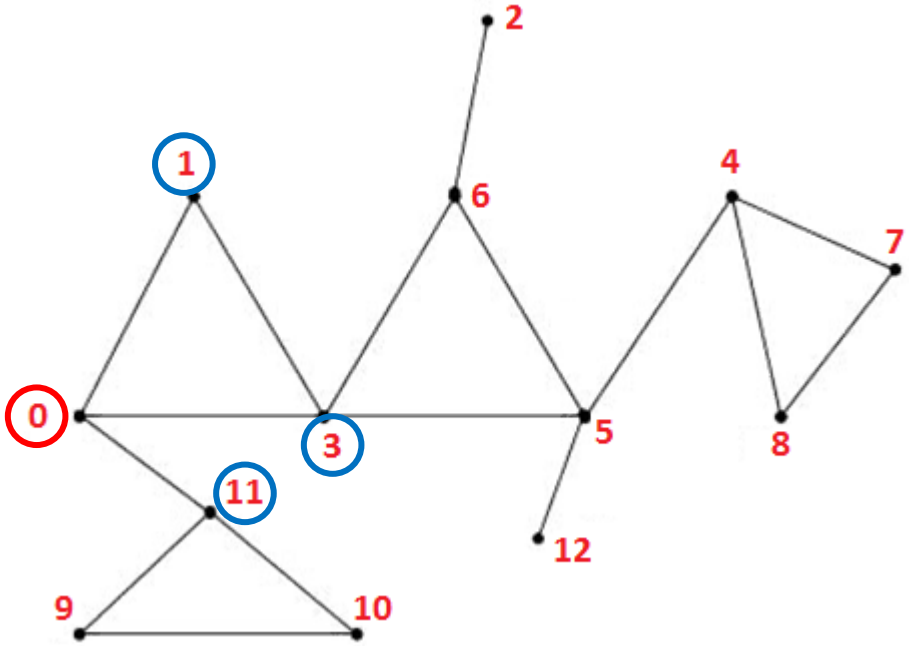
Mảng *ChuaXet*

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
F	T	T	T	T	T	T	T	T	T	T	T	T

Mảng *LuuVet*

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

2.1. Tìm kiếm theo chiều rộng trên đồ thị *(BFS - Breadth First Search)*



Queue

~~0~~	1	3	11	-1	-1

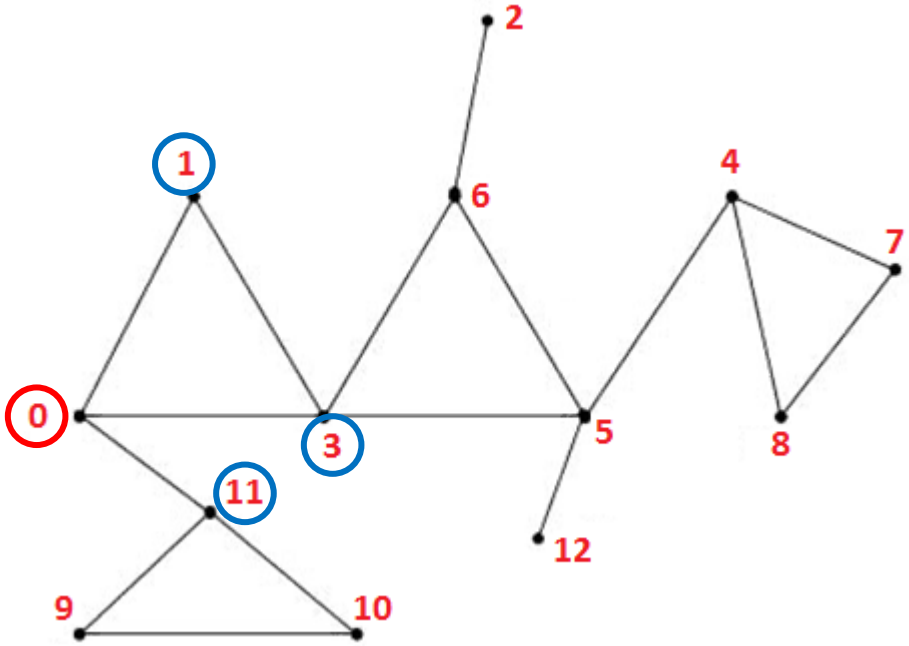
Mảng *ChuaXet*

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
F	F	T	F	T	T	T	T	T	T	T	F	T

Mảng *LuuVet*

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
0	0	-1	0	-1	-1	-1	-1	-1	-1	-1	0	-1

2.1. Tìm kiếm theo chiều rộng trên đồ thị *(BFS - Breadth First Search)*



Queue

~~0~~	1	3	11	-1	-1

Mảng *ChuaXet*

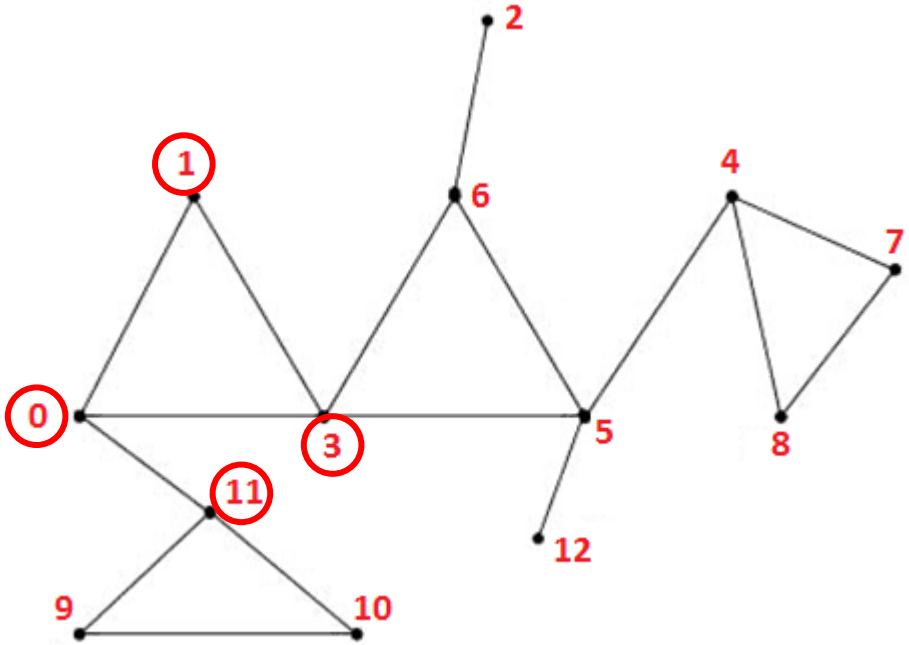
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
F	F	T	F	T	T	T	T	T	T	T	F	T

Mảng *LuuVet*

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
0	0	-1	0	-1	-1	-1	-1	-1	-1	-1	0	-1

2. Một số ứng dụng của Queue

2.1. Tìm kiếm theo chiều rộng trên đồ thị *(BFS - Breadth First Search)*



Lấy đỉnh 1 ra nhưng không tìm thấy đường đi tiếp

Queue

0	1	3	11	-1	-1

Mảng *ChuaXet*

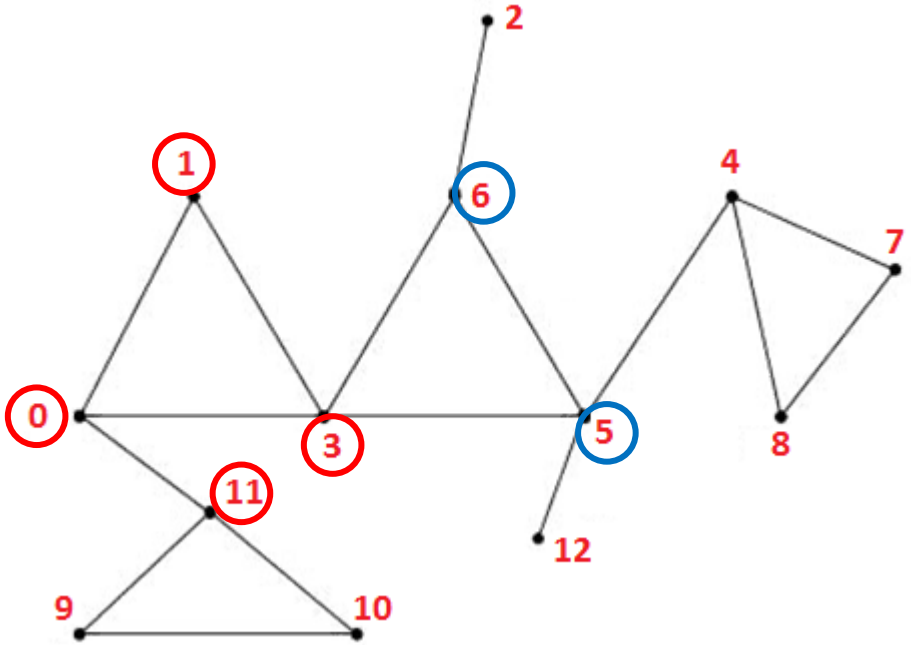
0	1	2	3	4	5	6	7	8	9	10	11	12
F	F	T	F	T	T	T	T	T	T	T	F	T

Mảng *LuuVet*

0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	-1	0	-1	-1	-1	-1	-1	-1	-1	0	-1

2. Một số ứng dụng của Queue

2.1. Tìm kiếm theo chiều rộng trên đồ thị *(BFS - Breadth First Search)*



*Lấy đỉnh **3** ra, tìm thấy hai đỉnh **5** và **6***

Queue

~~0~~	~~1~~	~~3~~	11	5	6

Mảng *ChuaXet*

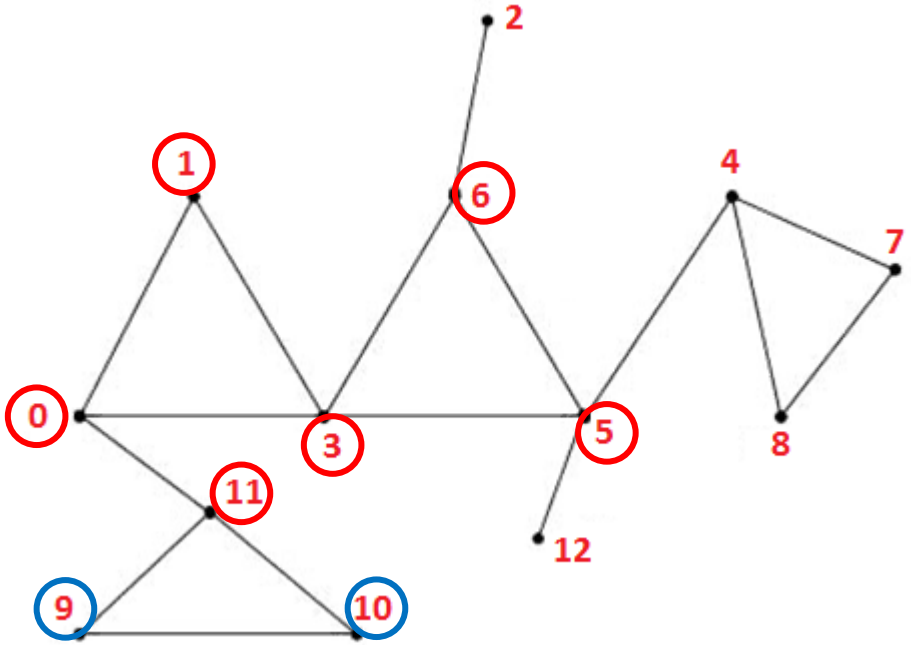
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
F	F	T	F	T	F	F	T	T	T	T	F	T

Mảng *LuuVet*

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
0	0	-1	0	-1	3	3	-1	-1	-1	-1	0	-1

2. Một số ứng dụng của Queue

2.1. Tìm kiếm theo chiều rộng trên đồ thị *(BFS - Breadth First Search)*



*Lấy đỉnh **11** ra, tìm thấy hai đỉnh **9** và **10***

Queue

~~0~~	~~1~~	~~3~~	~~11~~	5	6	9	10

Mảng *ChuaXet*

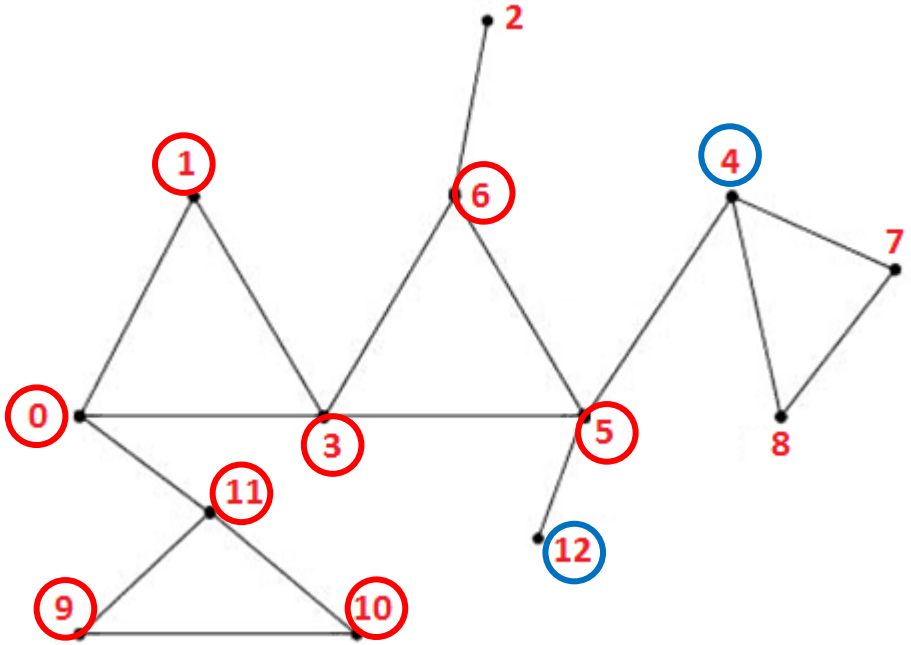
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
F	F	T	F	T	F	F	T	T	F	F	F	T

Mảng *LuuVet*

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
0	0	-1	0	-1	3	3	-1	-1	11	11	0	-1

2. Một số ứng dụng của Queue

2.1. Tìm kiếm theo chiều rộng trên đồ thị *(BFS - Breadth First Search)*



*Lấy đỉnh **5** ra, tìm thấy hai đỉnh **4** và **12***

Queue

0	1	3	11	5	6	9	10	4	12

Mảng *ChuaXet*

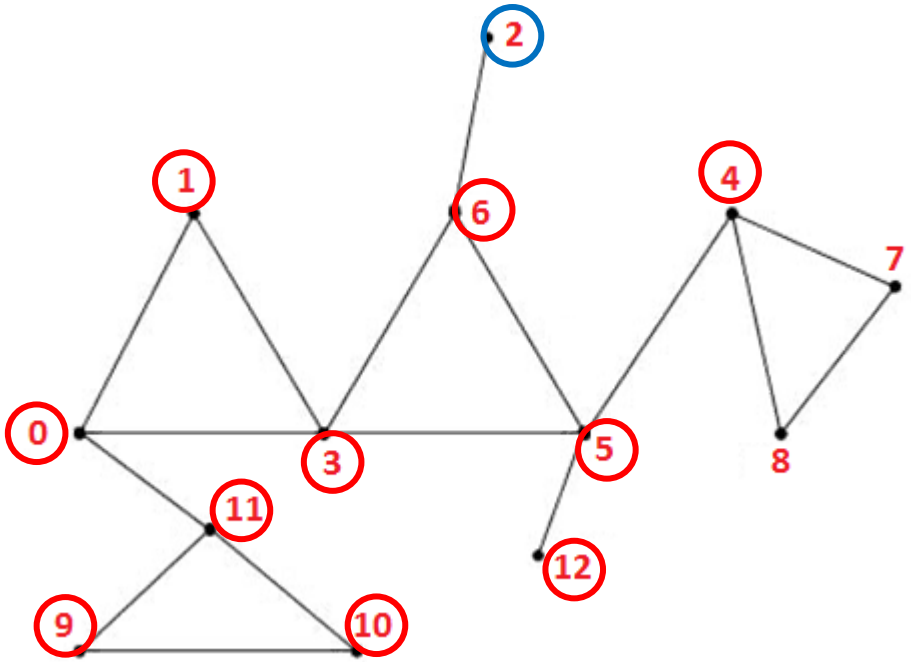
0	1	2	3	4	5	6	7	8	9	10	11	12
F	F	T	F	F	F	F	T	T	F	F	F	F

Mảng *LuuVet*

0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	-1	0	5	3	3	-1	-1	11	11	0	5

2. Một số ứng dụng của Queue

2.1. Tìm kiếm theo chiều rộng trên đồ thị *(BFS - Breadth First Search)*



*Lấy đỉnh **6** ra, tìm thấy đỉnh **2***

Queue

0	1	3	11	5	6	9	10	4	12	2

Mảng *ChuaXet*

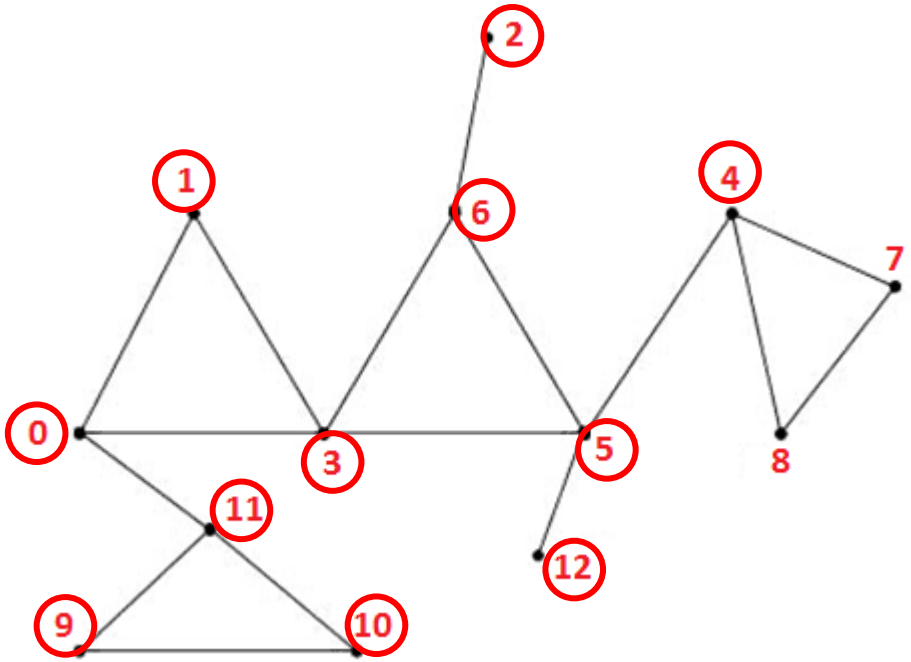
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
F	F	F	F	F	F	F	T	T	F	F	F	F

Mảng *LuuVet*

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
0	0	6	0	5	3	3	-1	-1	11	11	0	5

2. Một số ứng dụng của Queue

2.1. Tìm kiếm theo chiều rộng trên đồ thị *(BFS - Breadth First Search)*



*Lần lượt lấy đỉnh **9** và **10** ra nhưng đều không tìm thấy đường đi tiếp*

Queue

0	1	3	11	5	6	9	10	4	12	2

Mảng *ChuaXet*

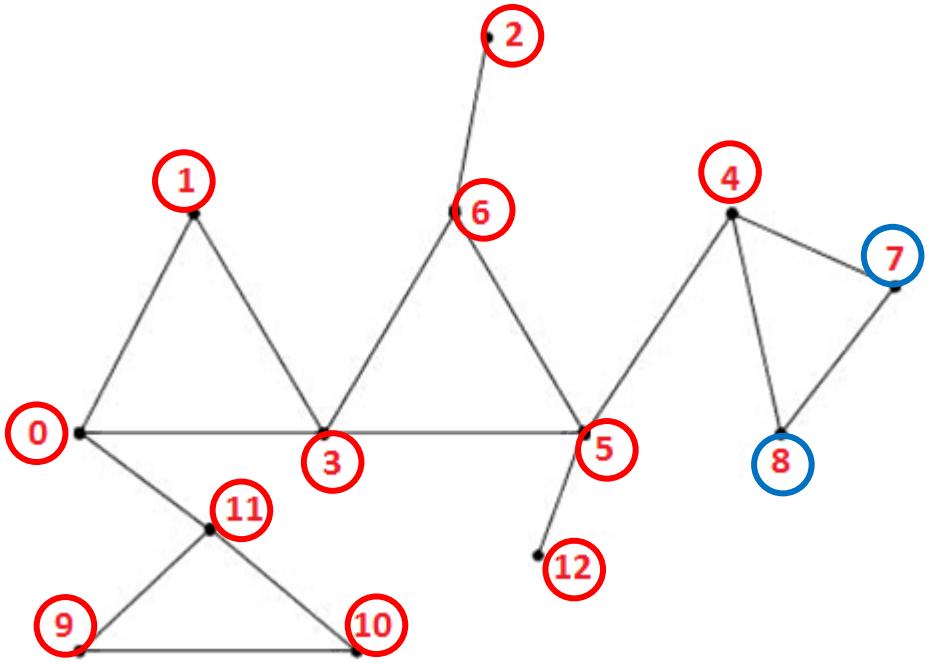
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
F	F	F	F	F	F	F	T	T	F	F	F	F

Mảng *LuuVet*

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
0	0	6	0	5	3	3	-1	-1	11	11	0	5

2. Một số ứng dụng của Queue

2.1. Tìm kiếm theo chiều rộng trên đồ thị *(BFS - Breadth First Search)*



*Lấy đỉnh **4** ra, thấy hai đỉnh **7** và **8***

Queue

0	1	3	11	5	6	9	10	4	12	2	7	8

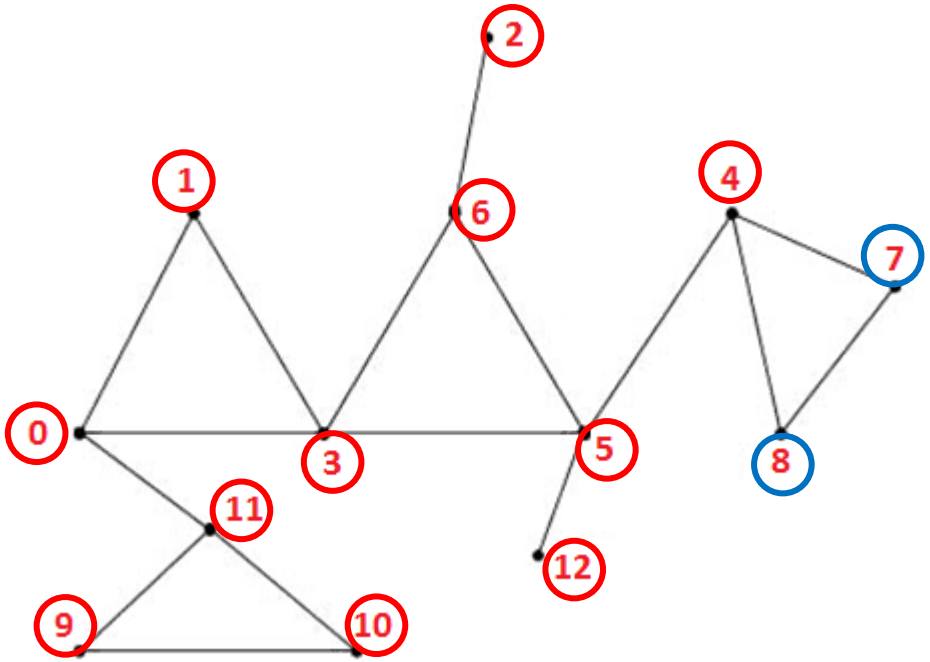
Mảng *ChuaXet*

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
F	F	F	F	F	F	F	F	F	F	F	F	F

Mảng *LuuVet*

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
0	0	6	0	5	3	3	4	4	11	11	0	5

2.1. Tìm kiếm theo chiều rộng trên đồ thị *(BFS - Breadth First Search)*



*Lần lượt lấy đỉnh **12, 2, 7** và **8** ra nhưng đều không tìm thấy đường đi tiếp và tất cả các đỉnh đã đều được duyệt*

Queue

0	1	3	11	5	6	9	10	4	12	2	7	8

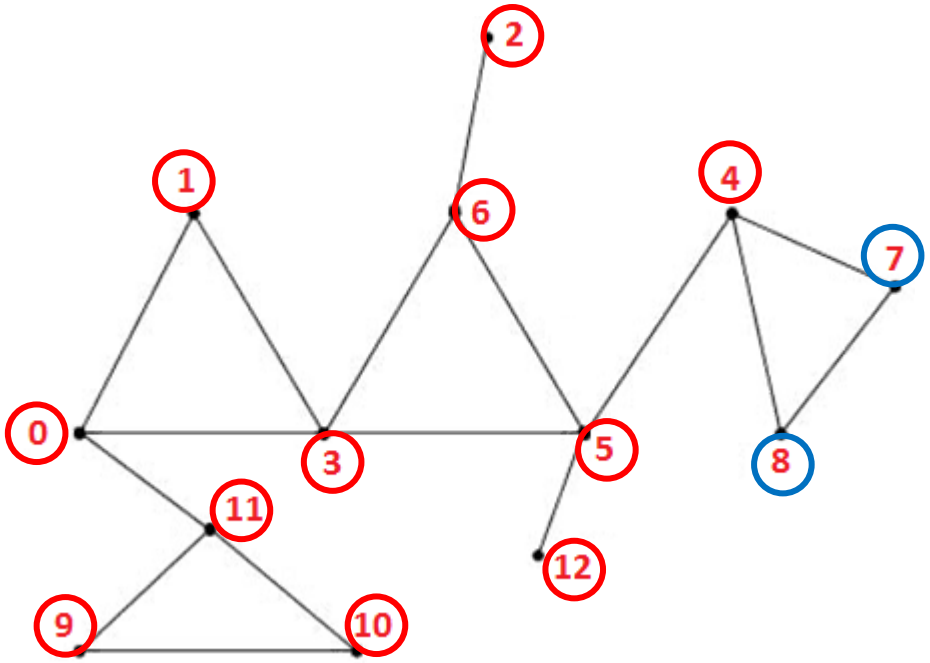
Mảng *ChuaXet*

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
F	F	F	F	F	F	F	F	F	F	F	F	F

Mảng *LuuVet*

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
0	0	6	0	5	3	3	4	4	11	11	0	5

2.1. Tìm kiếm theo chiều rộng trên đồ thị *(BFS - Breadth First Search)*



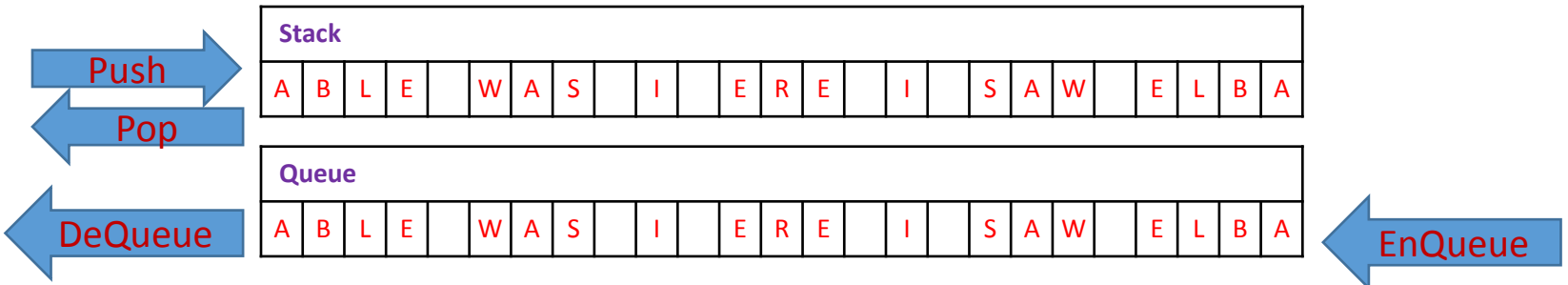
Giả sử cần tìm đường đi từ đỉnh 0 đến đỉnh 7:

Mảng *LuuVet*: 7 ← 4 ← 5 ← 3 ← 0

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
0	**0**	**6**	**0**	**5**	**3**	**3**	**4**	**4**	**11**	**11**	**0**	**5**

2.2. Chuỗi Palindromes

- **Khái niệm**: Một chuỗi được gọi là *Palindrome* nếu như đọc xuôi hay đọc ngược đều giống nhau (chuỗi đối xứng).
- **Bài toán**: Cho trước một chuỗi, kiểm tra xem chuỗi đó có phải là chuỗi *Palindrome* hay không?
- **Ví dụ:** **Able was I ere I saw Elba**
- **Giải pháp:**
  - Để tránh ảnh hưởng tới chuỗi ban đầu, đọc chuỗi nói trên vào *stack* và *queue*.
  - So sánh từng phần tử của stack và queue, nếu giống nhau từng cặp thì đó là chuỗi *Palindrome*



2.3. Giải thuật Demerging

- **Bài toán**: Xem xét bài toán về hệ thống quản lý nhân sự. Các bản ghi được lưu trên file.
  - Mỗi bản ghi gồm các trường: Họ tên, giới tính, ngày tháng năm sinh, ...
  - Dữ liệu trên đã được sắp theo ngày tháng năm sinh.
  - Cần tổ chức lại dữ liệu sao cho nữ được liệt kê trước nam nhưng vẫn giữ được tính đã sắp theo ngày tháng năm sinh.

2.3. Giải thuật Demerging

- **Đề xuất cách giải quyết**:
  - Ý tưởng không hiệu quả:
    - Sử dụng thuật toán sắp xếp.
    - Độ phức tạp của thuật toán O(nlogn) trong trường hợp tốt nhất.
  - Ý tưởng hiệu quả hơn:
    - Sử dụng giải thuật demerging.
    - Độ phức tạp của giải thuật này là O(n)

2.3. Giải thuật Demerging

- Giải thuật Demerging:
  - **B1**. Tạo 2 queue rỗng, có tên lần lượt là NU và NAM.
  - **B2**. Với mỗi bản ghi p, xem xét:
    - Nếu p có giới tính nữ, đưa vào queue NU.
    - Ngược lại, (p có giới tính nam) đưa vào queue NAM.
  - **B3**. Xét queue NU, khi queue chưa rỗng:
    - B3.1. Lấy từng phần tử trong queue này.
    - B3.2. Ghi vào file output.
  - **B4**. Xét queue NAM, khi queue chưa rỗng:
    - B4.1. Lấy từng phần tử trong queue này.
    - B4.2. Ghi tiếp vào file output trên.
  - **B5**. Kết thúc

2.4. Buffer

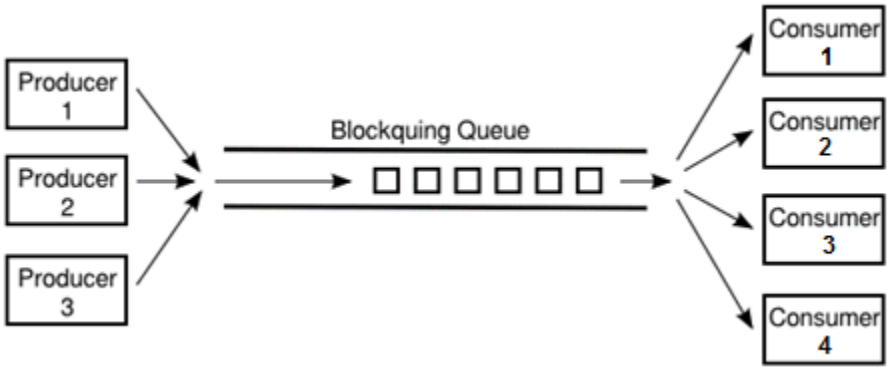
- **Giới thiệu**: Buffer là *dữ liệu tạm thời* và thường được lưu trữ trong bộ nhớ tạm (RAM).
- **Browser**: khi xem một đoạn video trực tuyến, có hai cách để trình duyệt tải dữ liệu từ đoạn video này:
  i. Tải toàn bộ dữ liệu của video rồi mới chạy
  ii. Tải từng phần của video và chạy từng phần nội dung mỗi khi dữ liệu được tải về máy. Từng phần dữ liệu video được tải về và lưu vào vùng nhớ tạm của máy vùng nhớ tạm này được gọi là *buffer*.

2.4. Buffer

- **Web Server**: khi có một *"slow" client request* (request đến từ một thiết bị có tốc độ kết nối/xử lý rất chậm) như một chiếc máy tính "yếu" hoặc điện thoại sử dụng 3G với đường truyền "chậm". Khi đó, Web Server sẽ phải làm 2 việc:
  - "*keep alive*" connection đến *"slow" client request* đó trong 1 queue (sử dụng buffer thông qua một reverse proxy), và đợi cho đến khi client đó nhận được kết quả trả về!
  - Dành resource cho những client request khác.
- **Keyboard Buffer**: là 1 vùng nhỏ trong bộ nhớ được dành riêng để giữ tạm các mã tín hiệu của các lần ấn phím sau cùng (của người dung) trên bàn phím. Nhờ vậy, máy tính có thể tiếp tục thu nhận các tín hiệu từ bàn phím ngay cả khi đang bận làm việc khác.

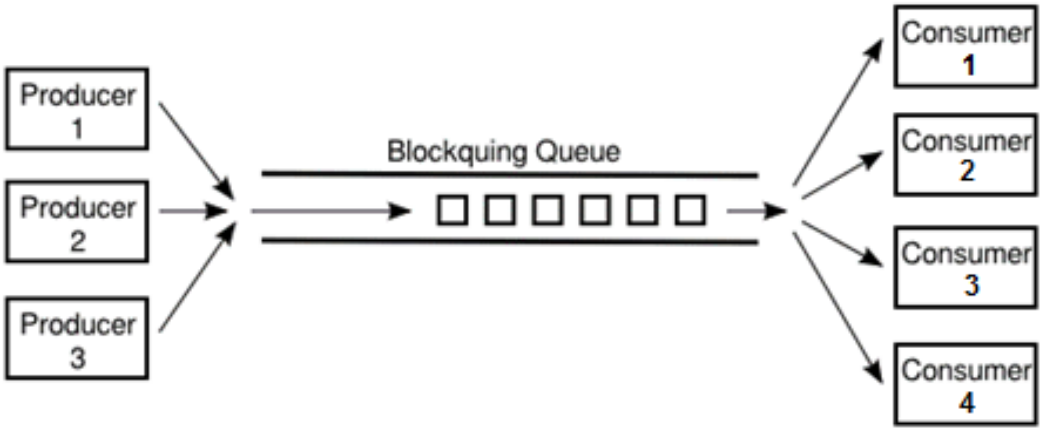
2.5. Mô hình Producer-Consumer

- Mô hình này hay được sử dụng trong các hệ thống phân tán, khi các công việc sẽ được tạo bởi một loạt các tiến trình gọi là *Producer*, và công việc đó sẽ được xử lý bởi một loạt các tiến trình khác gọi là *Consumer*.
- *Producer* và *Consumer* có thể nằm trên cùng 1 máy hoặc các máy khác nhau.
- Vấn đề là làm thế nào chúng ta có thể chuyển công việc từ *Producer* đến *Consumer* $\Rightarrow$ *BlockingQueue*



2.5. Mô hình Producer-Consumer

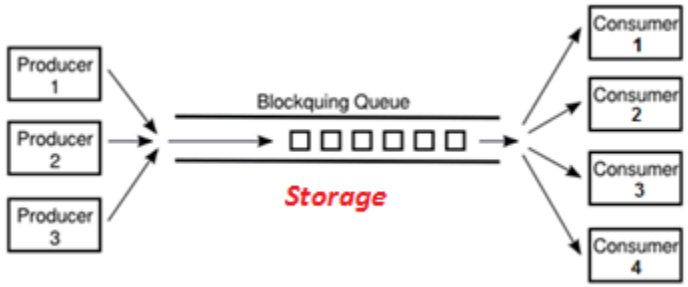
- *BlockingQueue* là cấu trúc dữ liệu nhằm để giải quyết bài toán *Producer-Consumer*. Cụ thể là trong môi trường *Multi Threaded* khi mà nhiều *Producers* nằm trên nhiều *Thread* đều muốn "*nhét*" dữ liệu vào cùng 1 queue. Tính năng *Thread safe* của *BlockingQueue* giúp quản lý quá trình đó.
- Tính năng *Thread safe* của *BlockingQueue*:
  - **Khi *queue* bị đầy**: hành vi đưa dữ liệu vào của *Producer* sẽ bị *blocked* cho đến khi *Consumer* giải phóng bớt dữ liệu ra
  - **Khi *queue* rỗng**: *Consumer* sẽ bị blocked cho đến khi *Producer* có đưa dữ liệu vào.

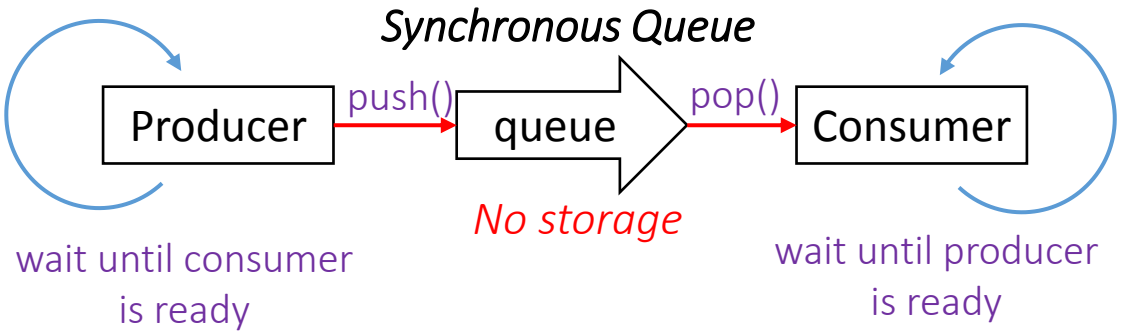


2.5. Mô hình Producer-Consumer

- *SynchronousQueue* và *BlockingQueue* cùng giải quyết bài toán queue cho mô hình *Producer-Consumer.*
- Điểm khác biệt: *SynchronousQueue* sẽ block hành vi đưa dữ liệu vào khi có hành vi lấy dữ liệu ra tương ứng.

Tức là *SynchronousQueue* chỉ cho phép thông lượng (*throughput*) là 1 item trong khi *BlockingQueue* cho phép giới hạn trên của buffer là ***n***.

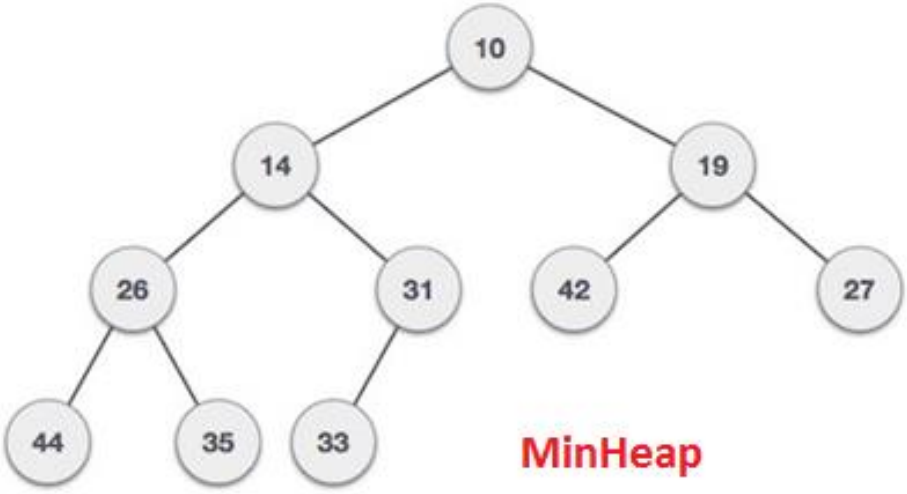


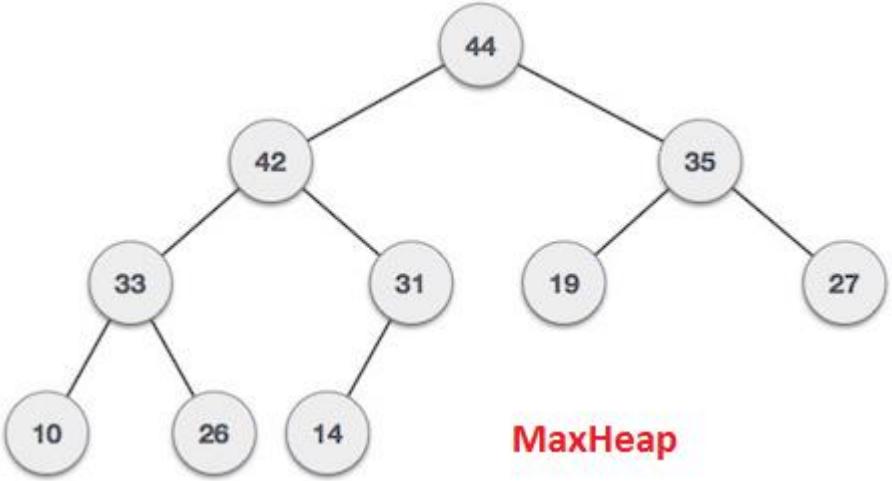


3. HÀNG ĐỢI CÓ ƯU TIÊN *(Priority Queue)*

3.1. Cấu trúc dữ liệu Heap

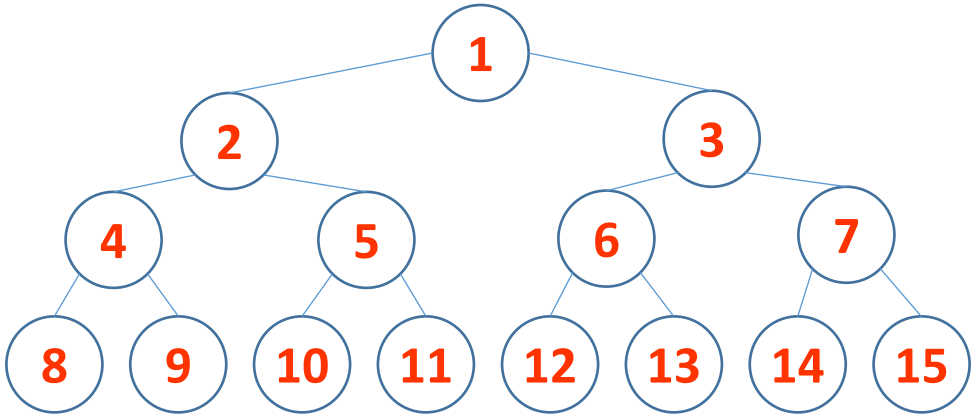
- Cấu trúc dữ liệu Heap là một trường hợp đặc biệt của cấu trúc dữ liệu cây nhị phân cân bằng, trong đó khóa của nút gốc được so sánh với các con của nó và được sắp xếp một cách phù hợp. Nếu node α có nút con β thì:
  - key(α) > key(β) ⇒ MaxHeap
  - key(α) < key(β) ⇒ MinHeap
- Ví dụ với thứ tự các giá trị đưa vào là: 35 33 42 10 14 19 27 44 26 31





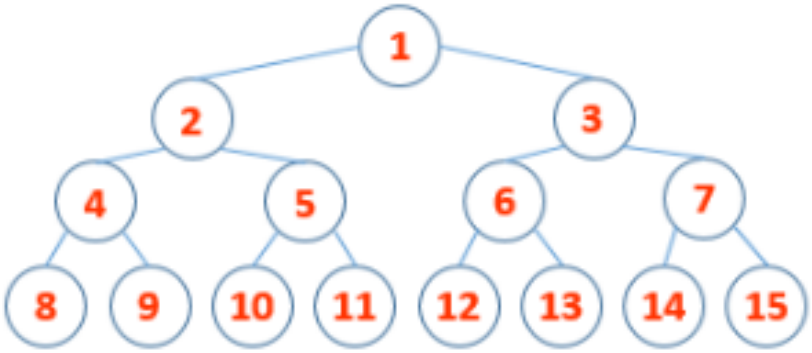
3.1. Cấu trúc dữ liệu Heap

- Thứ tự khi đưa giá trị vào heap:
  - Luôn theo thứ tự từ nhỏ đến lớn.
  - Sau khi được thêm vào, sẽ thực hiện hoán vị với node cha (nếu có) để đạt được MaxHeap (hoặc MinHeap)



- Thứ tự khi lấy giá trị ra khỏi heap:
  - Phần tử gốc luôn luôn được ưu tiên lấy ra trước
  - Phần tử thay thế gốc sẽ là phần tử có thứ tự lớn nhất đang có trên cây

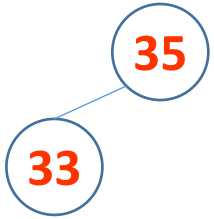
3.1. Cấu trúc dữ liệu Heap



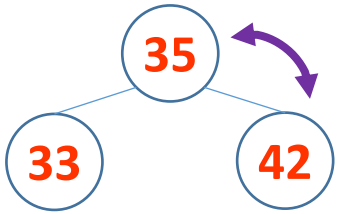
- Ví dụ với thứ tự các giá trị đưa vào MaxHeap là:

35 33 42 10 14 19 27 44 26 31

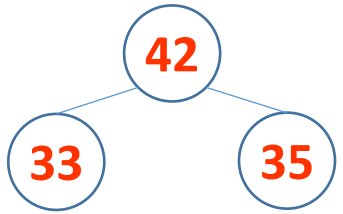
Add 35



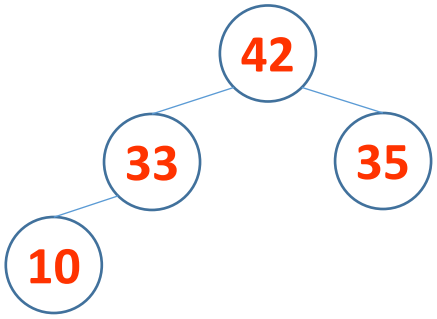
Add 33



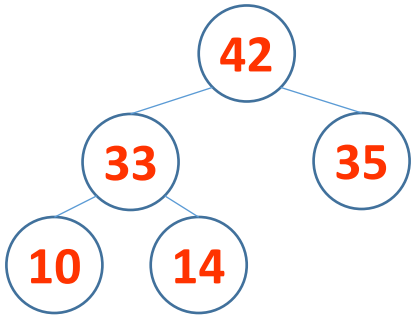
Add 42



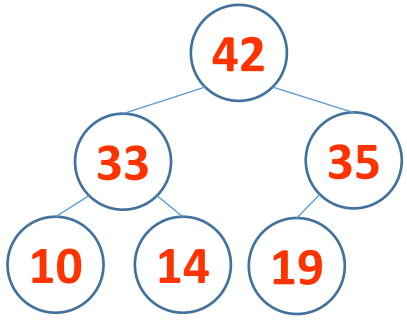
After swap 35 - 42



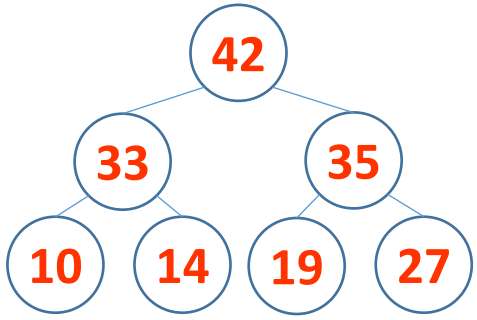
Add 10



Add 14

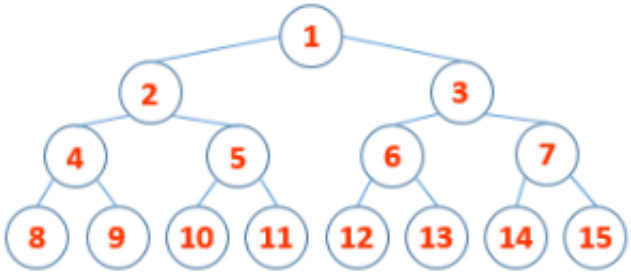


Add 19

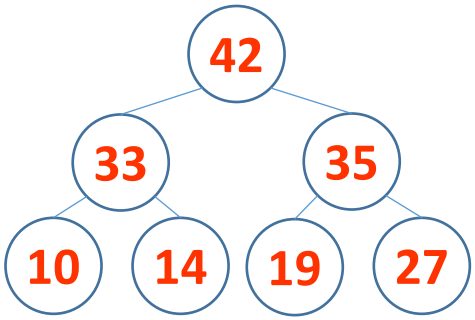


Add 27

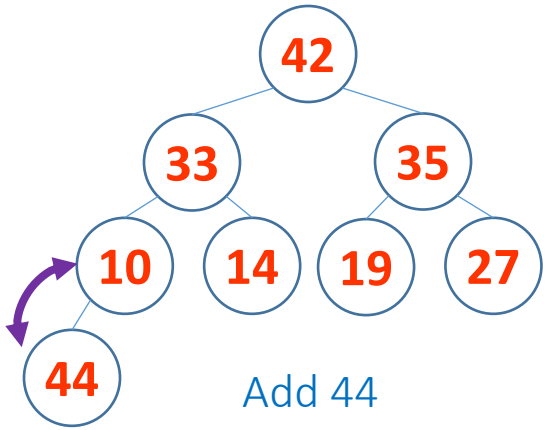
3.1. Cấu trúc dữ liệu Heap



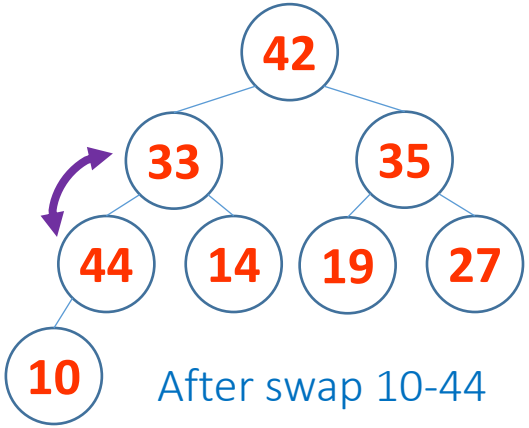
- Ví dụ với thứ tự các giá trị đưa vào MaxHeap là: 35 33 42 10 14 19 27 44 26 31



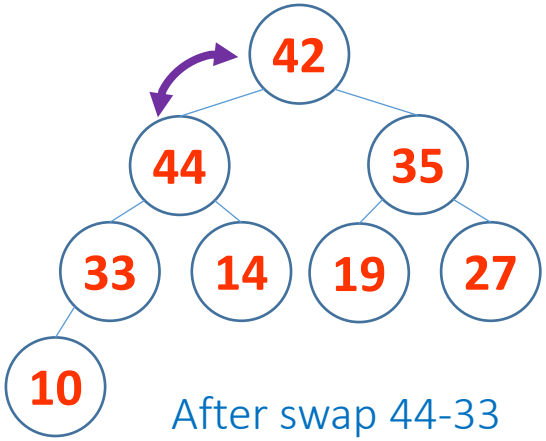
After add 27



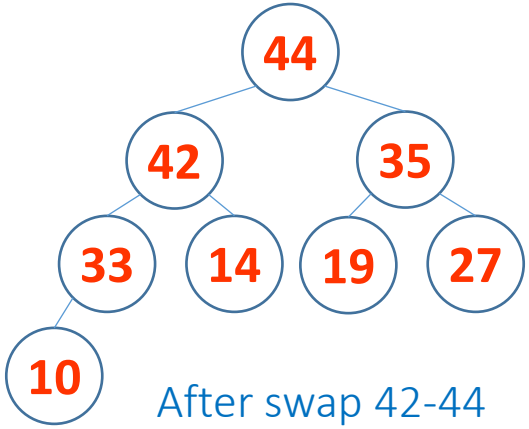
Add 44



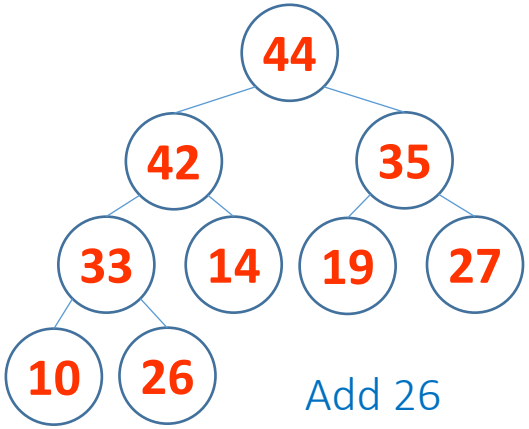
After swap 10-44



After swap 44-33

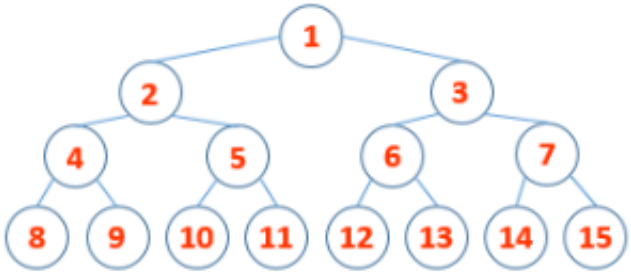


After swap 42-44

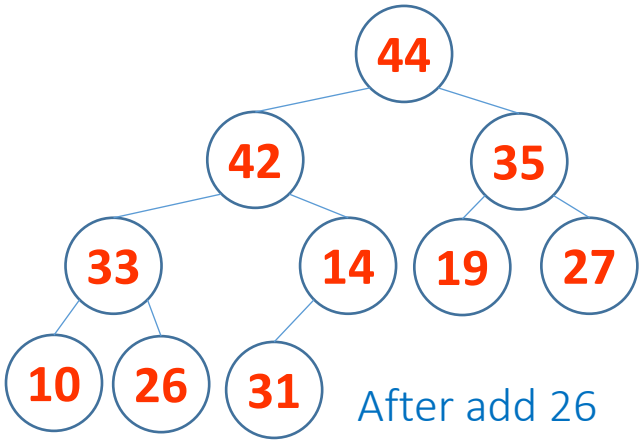


Add 26

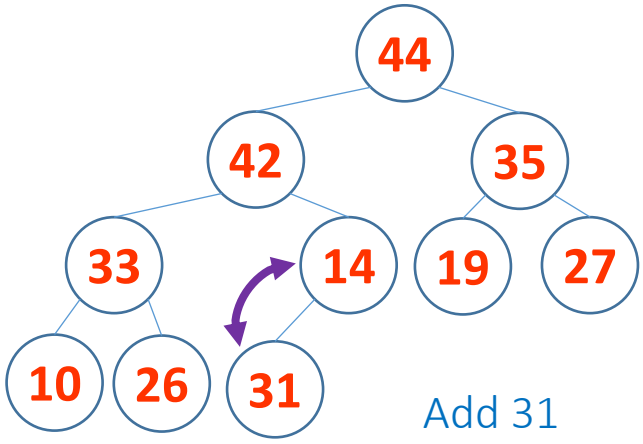
3.1. Cấu trúc dữ liệu Heap



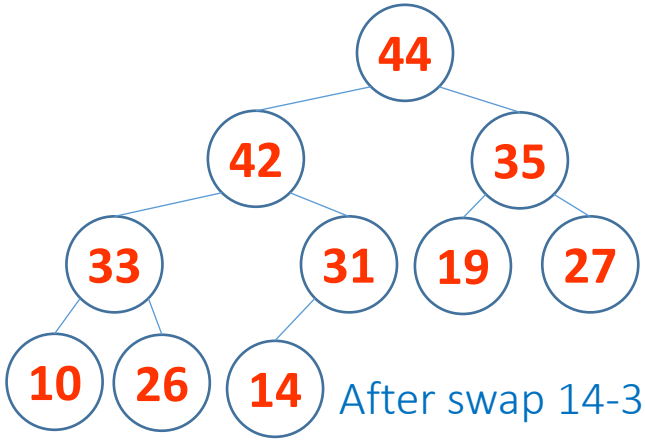
- Ví dụ với thứ tự các giá trị đưa vào MaxHeap là: 35 33 42 10 14 19 27 44 26 31



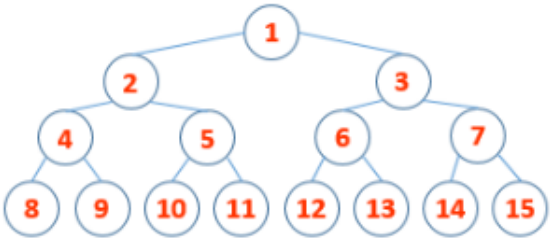
After add 26



Add 31

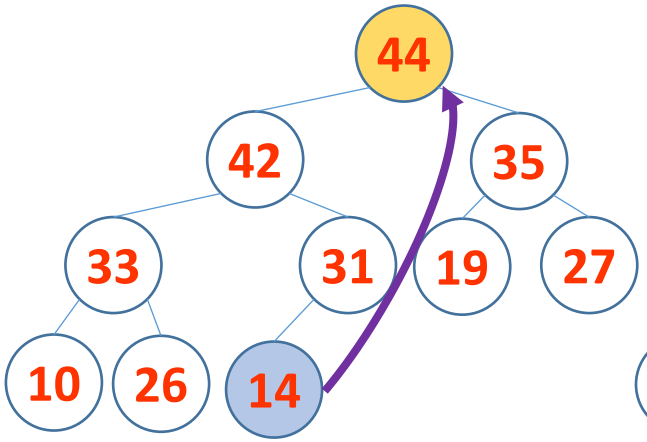


After swap 14-31

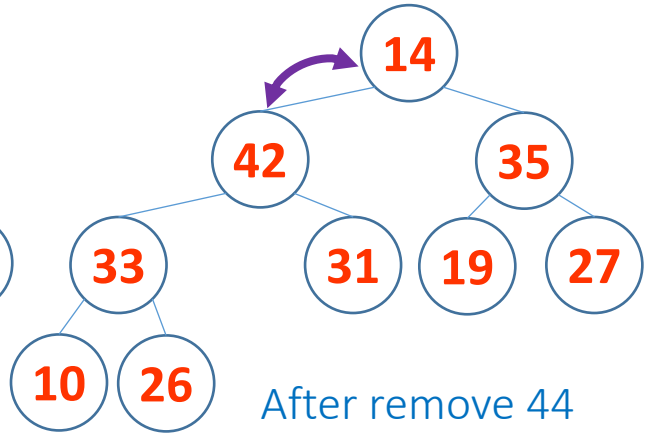


3.1. Cấu trúc dữ liệu Heap

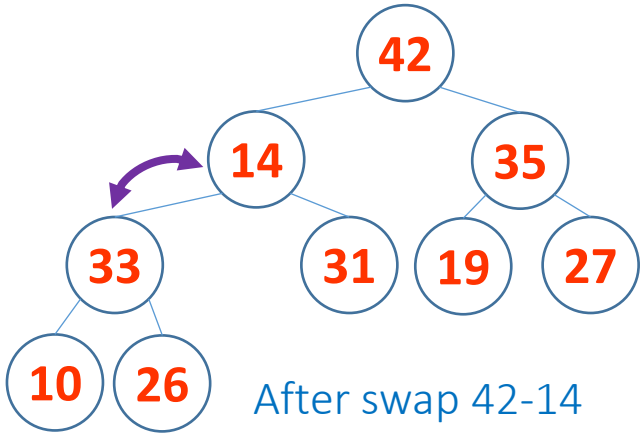
- Ví dụ: khi lấy ra, phần tử gốc (44) luôn được lấy ra, phần tử thay thế là phần tử có thứ tự cao nhất (với giá trị=14)



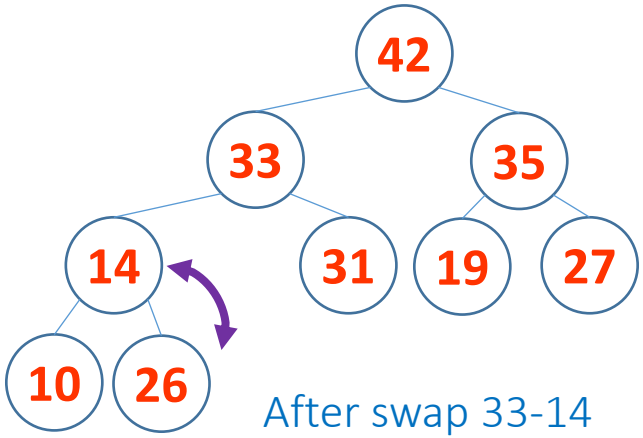
Remove 1 item



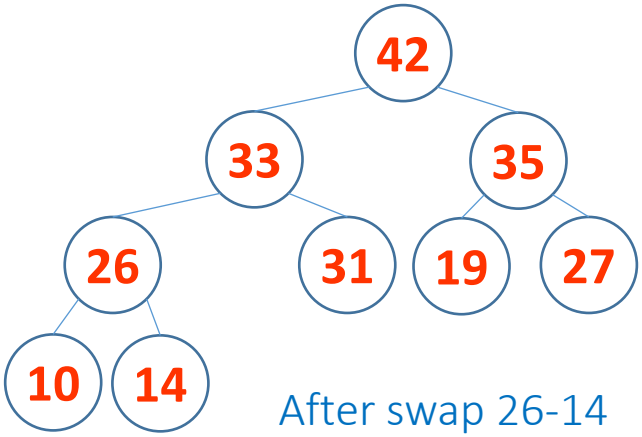
After remove 44



After swap 42-14



After swap 33-14



After swap 26-14

Remove next?

3.2. Hàng đợi có ưu tiên

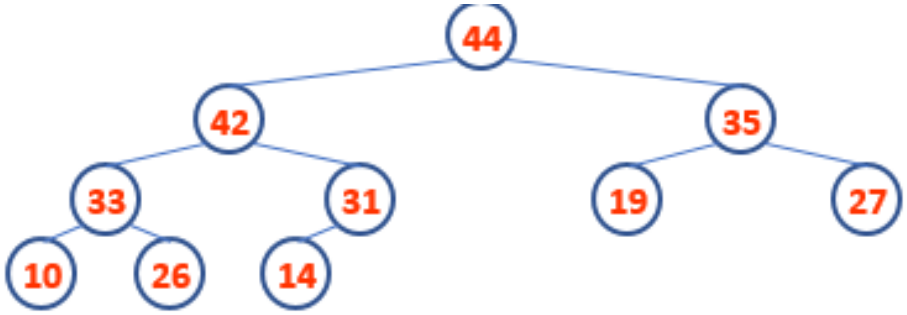
- Hàng đợi hoạt động trên nguyên tắc FIFO là hàng đợi không có ưu tiên.
- Trong thực tế có một dạng hàng đợi khác hoạt động theo cơ chế: node nào có độ ưu tiên cao hơn sẽ được lấy ra trước, node nào có độ ưu tiên thấp hơn sẽ được lấy ra sau. Hàng đợi loại này được gọi là hàng đợi có ưu tiên.
- Phân loại: có 2 loại:
  - ***Hàng đợi ưu tiên theo giá trị lớn nhất*** (*max -priority queue*): node có giá trị cao nhất sẽ được lấy ra trước.
  - ***Hàng đợi ưu tiên theo giá trị thấp nhất*** (*min -priority queue*): node có giá trị thấp nhất sẽ được lấy ra trước.

3.2. Hàng đợi có ưu tiên

- **Ví dụ**:

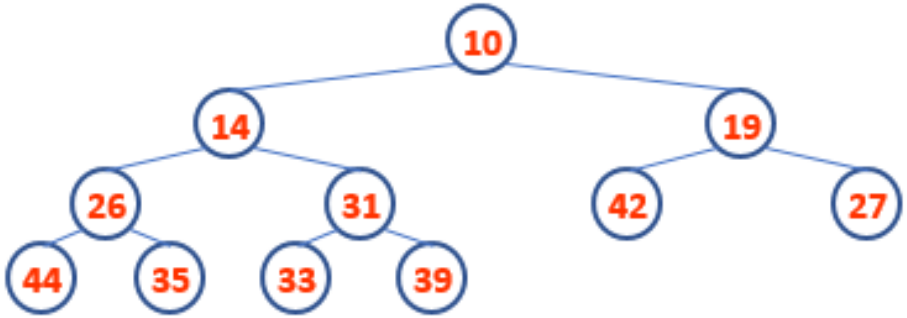
max -priority queue

44	42	35	33	31	19	27	10	26	14



min -priority queue

10	14	19	26	31	42	27	44	35	33	39



3.3. Cách hiện thực priority queue

- ***Cách tiếp cận đơn giản***: có thể sử dụng danh sách liên kết để thực hiện chèn các phần tử với độ phức tạp O(n) (khi chèn phần tử vào cuối danh sách liên kết) và có thể sắp xếp lại chúng để duy trì các đặc tính của PQ với độ phức tạp O(nlogn).
- ***Cách tiếp cận tối ưu***: sử dụng heap để xây dựng PQ với độ phức tạp là O(logn) cho việc chèn và xóa phần tử khỏi PQ.

3.4. Các tác vụ trên priority queue (PQ)

- Chèn phần tử vào priority queue.
- Xóa phần tử có giá trị lớn nhất (đối với hàng đợi ưu tiên theo giá trị lớn nhất - *max-priority queue*) ra khỏi priority queue và trả về giá trị của X.

  Ngược lại sẽ xóa phần tử X có giá trị nhỏ nhất (đối với hàng đợi ưu tiên theo giá trị nhỏ nhất - *min-priority queue*) ra khỏi priority queue và trả về giá trị của X.

3.4. Các tác vụ trên priority queue (PQ)

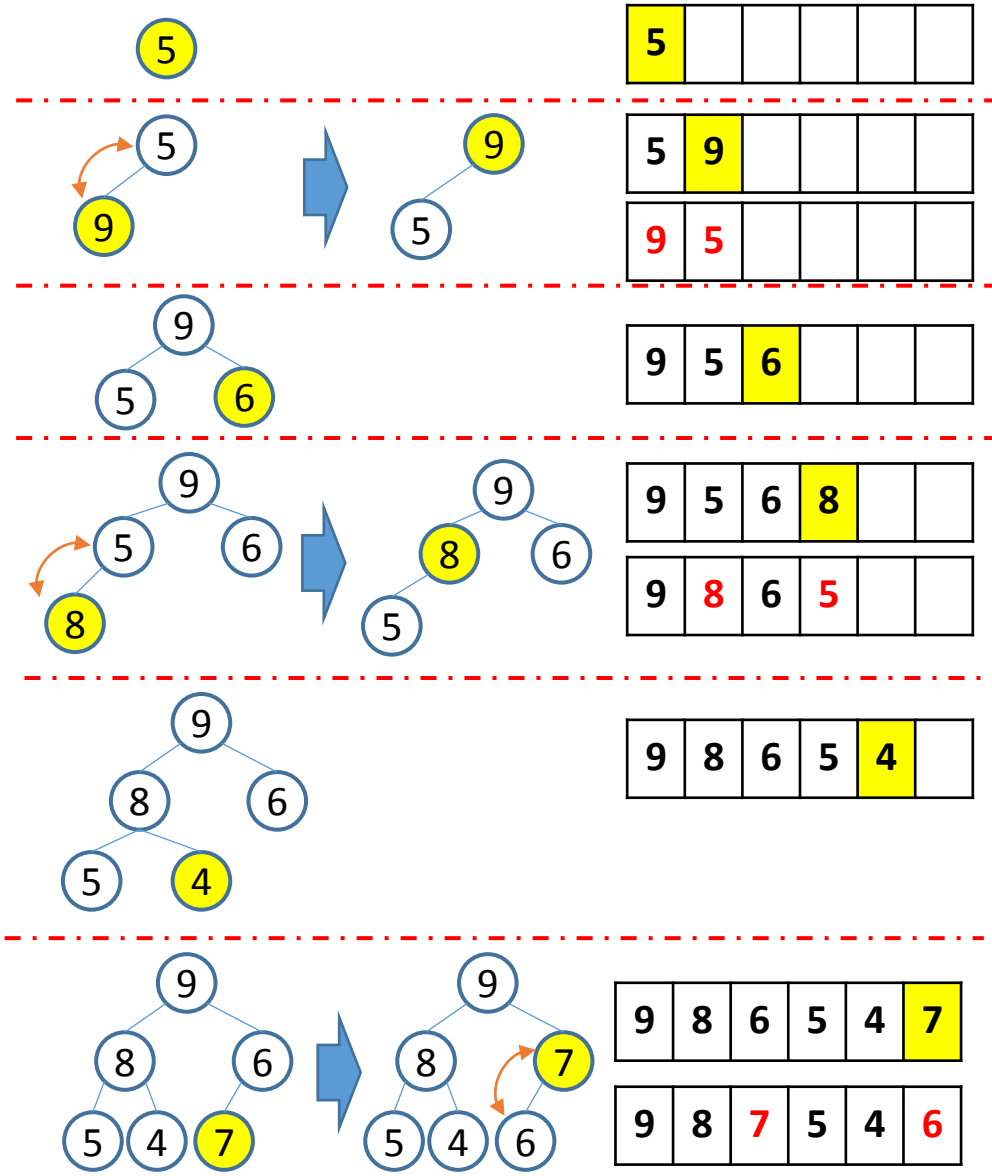
3.4.1. Tác vụ chèn

- Thực hiện
  - Lần lượt đưa từng phần tử vào heap theo thứ tự đã được quy định.
  - Ngay khi phần tử mới được thêm vào, nếu xảy ra vi phạm quy định về thứ tự của giá trị phần tử trên max/min-heap phải thực hiện hoán vị cho đến khi max/min-heap không còn bị vi phạm.
  - Sau đó mới được tiếp tục thêm các phần tử khác.

3.4. Các tác vụ trên priority queue

3.4.1. Tác vụ chèn

- Ví dụ: Minh họa cách hiện thực max priority queue bằng heap thông qua việc chèn 5 phần tử {5, 9, 6, 8, 4, 7}.



3.4. Các tác vụ trên priority queue (PQ)

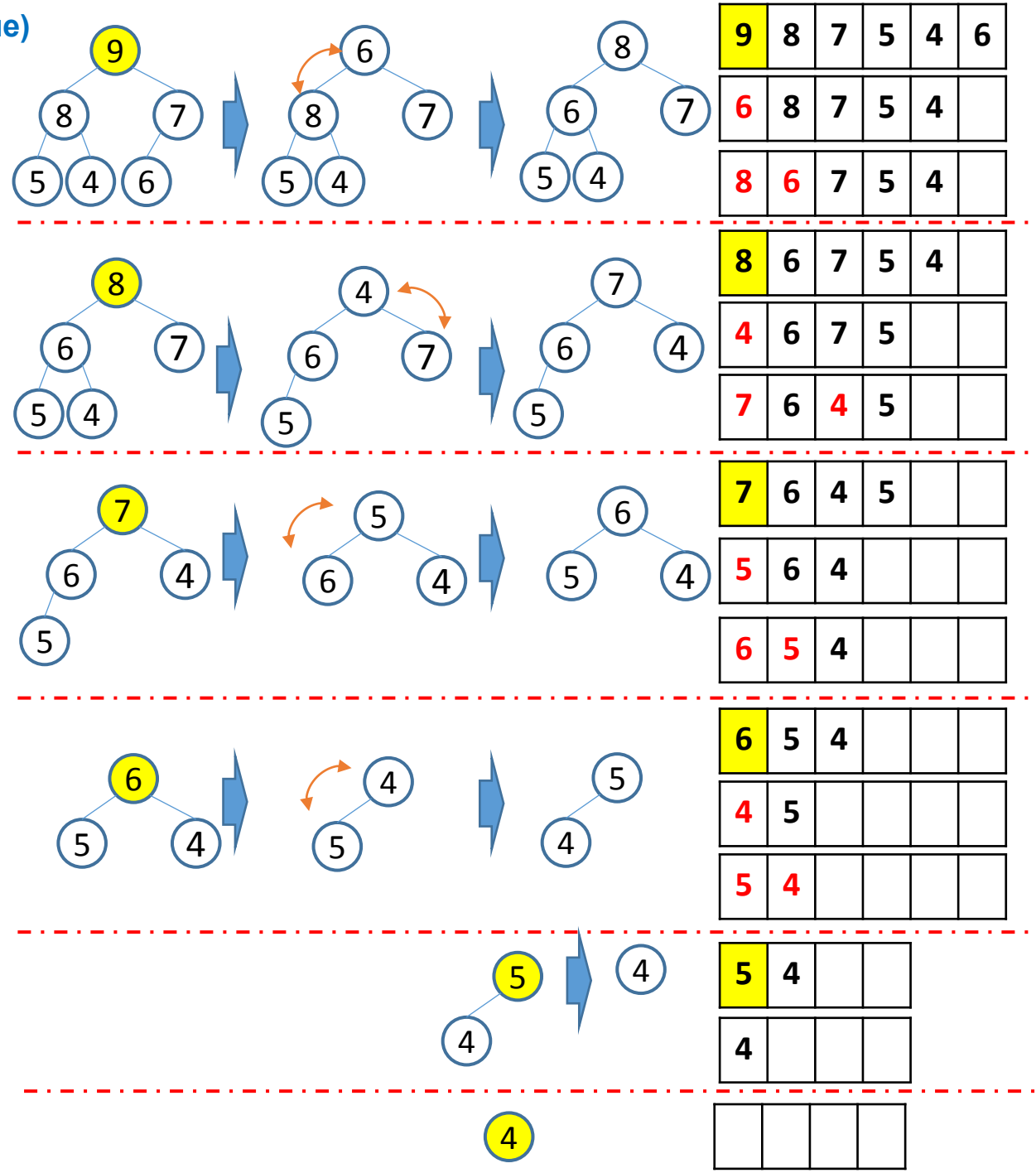
3.4.2. Tác vụ xóa

- Tác vụ xóa sẽ thực hiện
  - Lấy phần tử có giá trị lớn nhất X ra khỏi priority (đối với hàng đợi ưu tiên theo giá trị lớn nhất - max-priority queue), ngược lại sẽ lấy phần tử X có giá trị nhỏ nhất ra khỏi priority queue (đối với hàng đợi ưu tiên theo giá trị nhỏ nhất - min-priority queue).
  - Phần tử thay thế vị trí cho phần tử vừa bị xóa là phần tử đang có số thứ tự (chỉ số) lớn nhất trên queue.
  - Trả về giá trị X cho nơi gọi.

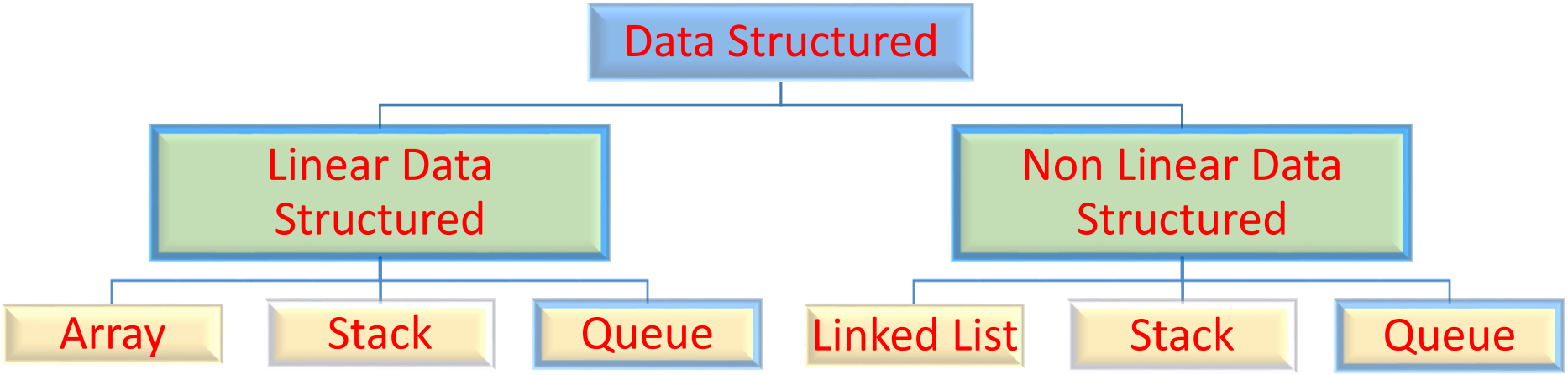
3.4. Các tác vụ trên priority queue

3.4.2. Tác vụ xóa

- Ví dụ lần lượt lấy ra từng phần tử của priority queue.



4. CÀI ĐẶT QUEUE



Có thể chọn 1 trong 2 cách cài đặt queue:

- Sử dụng cấu trúc dữ liệu dạng tuyến tính (cấu trúc dữ liệu tĩnh – danh sách kề mảng 1 chiều).
- Sử dụng cấu trúc dữ liệu dạng phi tuyến (cấu trúc dữ liệu động – danh sách liên kết).

Các thao tác cơ bản trên Queue

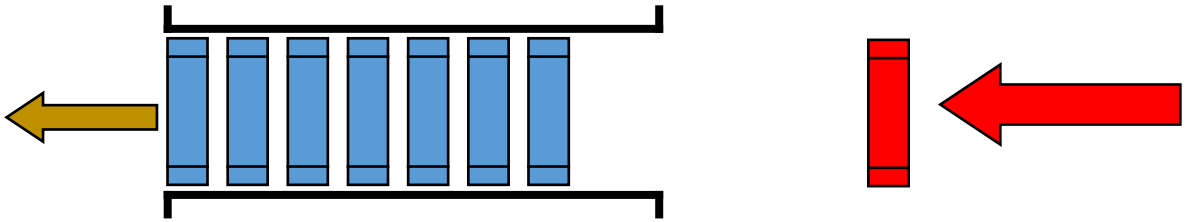
i. InitQueue: khởi tạo *Queue* rỗng

ii. IsEmpty: kiểm tra *Queue* rỗng?

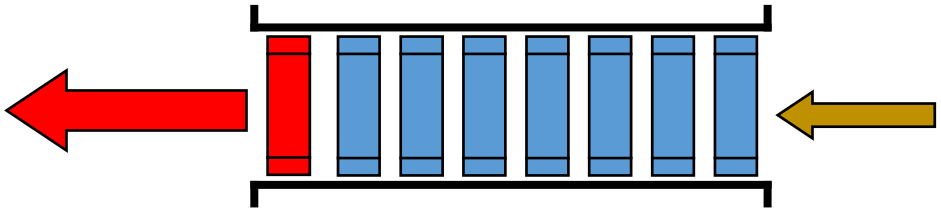
iii.IsFull: kiểm tra *Queue* đầy?

(có thể không cần cài đặt khi sử dụng DSLK)

iv. EnQueue: thêm 1 phần tử vào cuối Queue ⇨ có thể làm *Queue* đầy khi cài đặt queue bằng cách sử dụng danh sách kề.



v. DeQueue: lấy ra 1 phần tử từ đầu *Queue* ⇨ có thể làm *Queue* rỗng



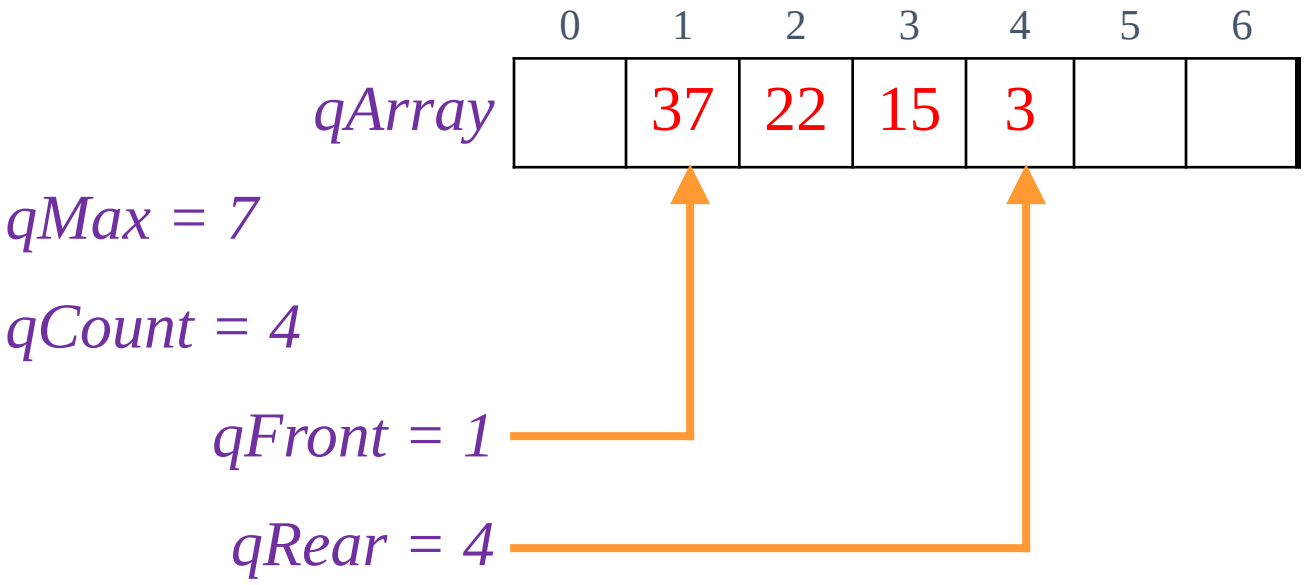
5. SỬ DỤNG DANH SÁCH KỀ LÀM QUEUE

5.1. Đặc điểm

- Viết chương trình dễ dàng, nhanh chóng
- Bị hạn chế do số lượng phần tử cố định
- Tốn chi phí tái cấp phát và sao chép vùng nhớ nếu sử dụng mảng động

5.2. Cấu trúc dữ liệu cần sử dụng

- 1 mảng (*qArray*) để chứa các phần tử.
- 1 số nguyên (*qMax*) để lưu sức chứa tối đa.
- 2 số nguyên (*qFront*, *qRear*) để xác định vị trí đầu, cuối.
- 1 số nguyên (*qCount*) để lưu số phần tử hiện có



5.3. Khai báo cấu trúc Queue

Giả sử Queue chứa các phần tử kiểu nguyên (int)

```
struct QUEUE
{  int     *qArray;//chứa các phần tử của Queue
   int      qMax;   //sức chứa tối đa
   int      qFront;// xác định vị trí đầu
   int      qRear; // xác định vị trí cuối
};
```

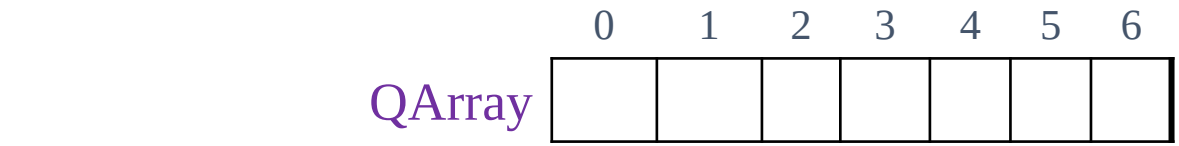
5.4. Thao tác "Khởi tạo Queue"

```
int InitQueue(QUEUE& q, int MaxItem)
{
   q.qArray = new int[MaxItem];
   if (q.qArray == NULL)
       return 0; // không đủ bộ nhớ
   q.qMax = MaxItem;
   q.qCount = 0;
   q.qFront = q.qRear = -1;
   return 1;      // thành công
}
```

QMax = 7; QCount = 0; QFront = -1; QRear = -1

5.5. Thao tác "Kiểm tra Queue rỗng"

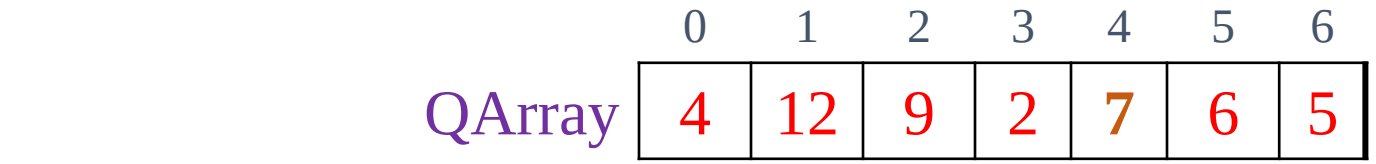
```
int IsEmpty(QUEUE q)
{
   return q.qCount == 0;
}
```



QMax = 7; QCount = **0**; QFront = -1; QRear = -1

5.6. Thao tác "Kiểm tra Queue đầy"

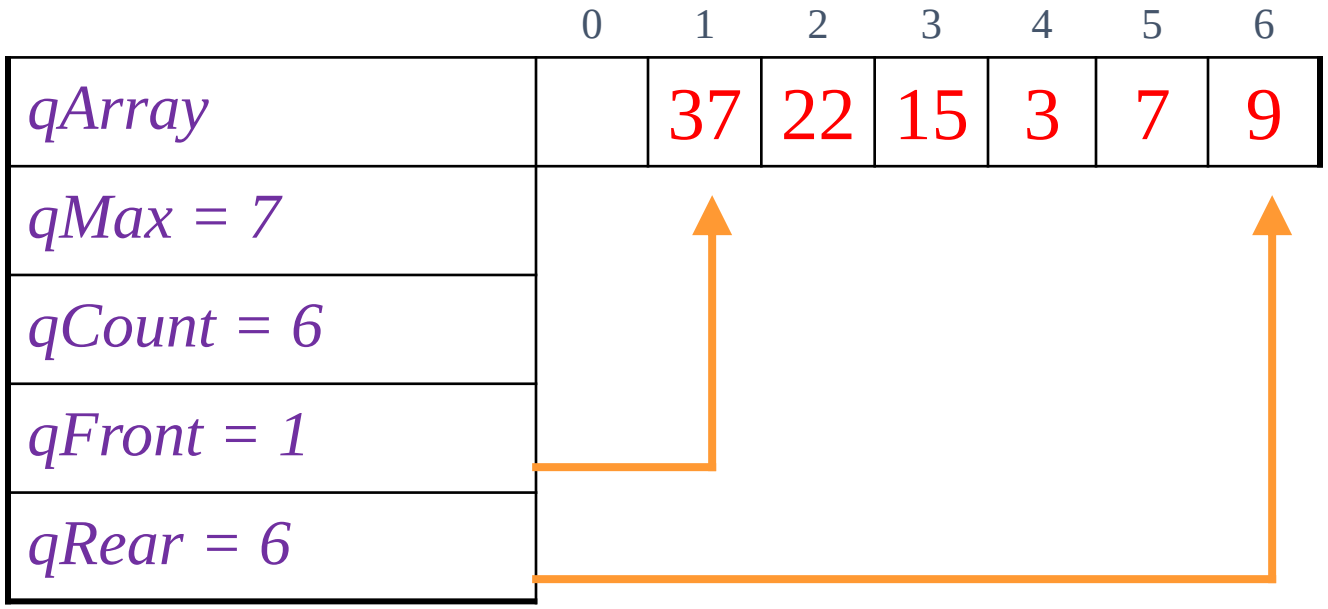
```
int IsFull(QUEUE q)
{
    return (q.qMax == q.qCount);
}
```



QMax = 7; QCount = 7; QFront = 0; QRear = 6

5.7. Thao tác thêm phần tử vào queue

- Khi thêm phần tử vào queue ⇒ xảy ra hiện tượng "tràn giả"

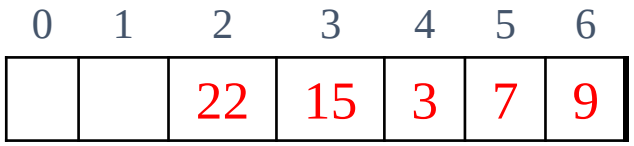


- Giải pháp? Nối dài mảng (mảng động) hay sử dụng một mảng vô cùng lớn?

5. Sử dụng danh sách kề làm Queue

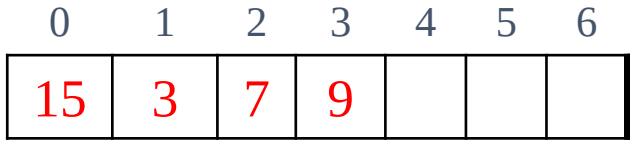
5.7. Thao tác thêm phần tử vào queue

- Cách 1: Dồn các phần tử về đầu mảng



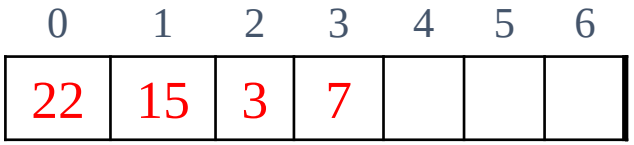
qMax = 7; qCount = 5; qFront = 2; qRear = 6

- Cách 1.1: ngay khi phần tử đầu trống.



qMax = 7; qCount = **4**; qFront = 0; qRear = **3**

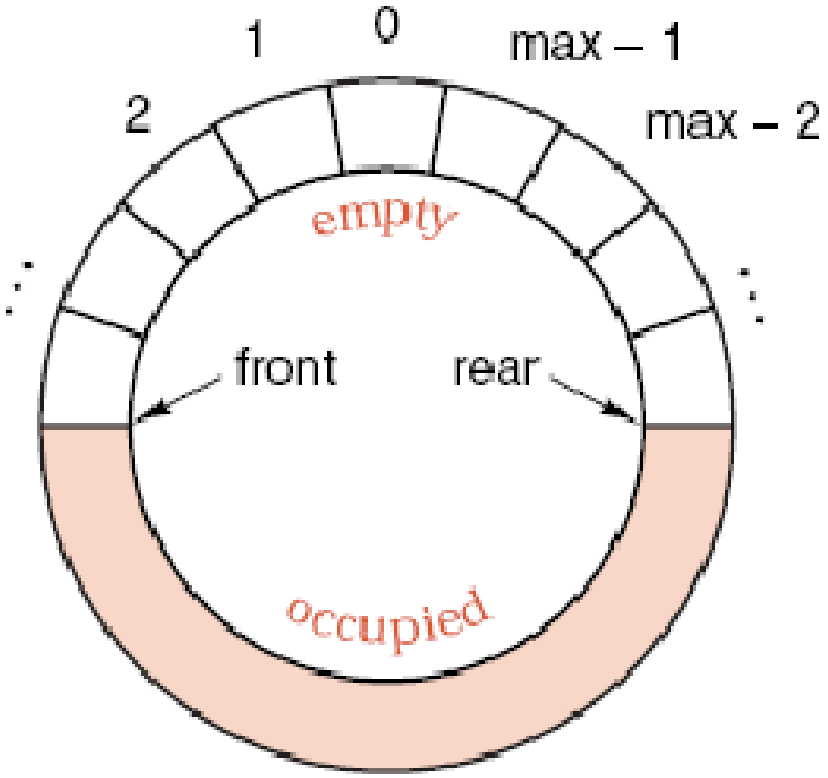
- Cách 1.2: khi phần tử cuối được dùng



qMax = 7; qCount = **5**; qFront = 0; qRear = **4**

5.7. Thao tác thêm phần tử vào queue

- Cách 2: tổ chức hàng đợi dạng vòng (Queue Circular), có thể coi mảng với các phân tử được xếp vòng tròn.

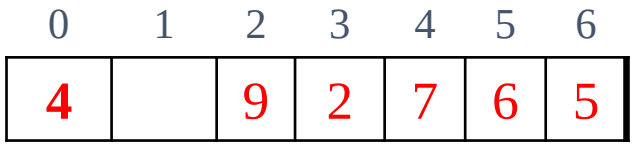


5. Sử dụng danh sách kề làm Queue

5.7. Thao tác EnQueue: thêm 1 phần tử vào dau Queue

```
int EnQueue(QUEUE &q, int NewItem)
{
    if (IsFull(q))
        return 0;                    // Queue đầy
    q.qRear++;
    if (q.qRear==q.qMax)             //"tràn giả"
        q.qRear = 0;                 // Quay trở về đầu mảng
    q.qArray[q.qRear] = NewItem;
    q.qCount++;
    return 1;                        // Thêm thành công
}
```

NewItem=4

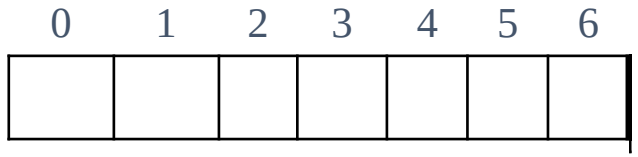


qMax = 7; qCount = **6**; qFront = 2; qRear = **0**

5.8. Thao tác DeQueue: lấy ra 1 phần tử ở đầu Queue

```
int DeQueue(QUEUE &q, int &ItemOut)
{   if (IsEmpty(q))
        return 0;                  //Queue rỗng, không lấy ra được
    ItemOut=q.qArray[q.qFront];    //lấy phần tử đầu ra
    q.qFront++;
     q.qCount--;
     if (q.qFront==q.qMax)         // nếu Qfront đã đi hết mảng …
        q.qFront = 0;              //cho Qfront quay trở về đầu mảng
     if (q.qCount==0)              //nếu lấy ra phần tử cuối cùng
        q.qFront= q.qRear= -1;     //khởi tạo lại Queue
     return 1;                     //Lấy ra thành công
}
```

ItemOut=5



QMax = 7; QCount = **0**; QFront = **-1**; QRear = **-1**

6. SỬ DỤNG DANH SÁCH LIÊN KẾT ĐƠN LÀM QUEUE

6.1. Khai báo cấu trúc

(Cấu trúc đơn giản:

data có kiểu dữ liệu là kiểu dữ liệu cơ sở của ngôn ngữ C)

```
struct NODE
{  DATATYPE data;
   NODE *next;
};
struct QUEUE
{  NODE *front;
   NODE *rear;
};
```

```
struct NODE
{   int data;
    NODE *next;
};
struct QUEUE
{   NODE *front;
    NODE *rear;
};
```

6. SỬ DỤNG DANH SÁCH LIÊN KẾT ĐƠN LÀM QUEUE

6.1. Khai báo cấu trúc

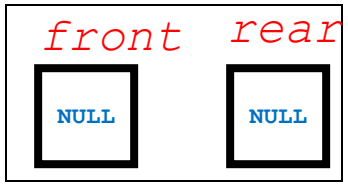
(Cấu trúc phức tạp: data có kiểu dữ liệu là kiểu dữ liệu do người lập trình tự định nghĩa)

```
struct DATA
{  DataType element1;
   …
   DataType elementn;
};
struct NODE
{  DATA data;
   NODE *next;
};
struct QUEUE
{  NODE *front;
   NODE *rear;
};
```

```
struct SINHVIEN
{   char Ma[3];
    char HoTen[30];
    float DiemTB;
};
struct NODE
{   SINHVIEN data;
    NODE *next;
};
struct QUEUE
{   NODE *front;
    NODE *rear;
};
```

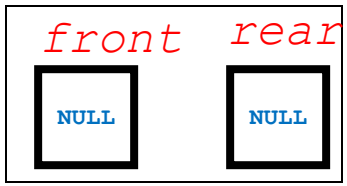
6.2. Khởi tạo hàng đợi rỗng

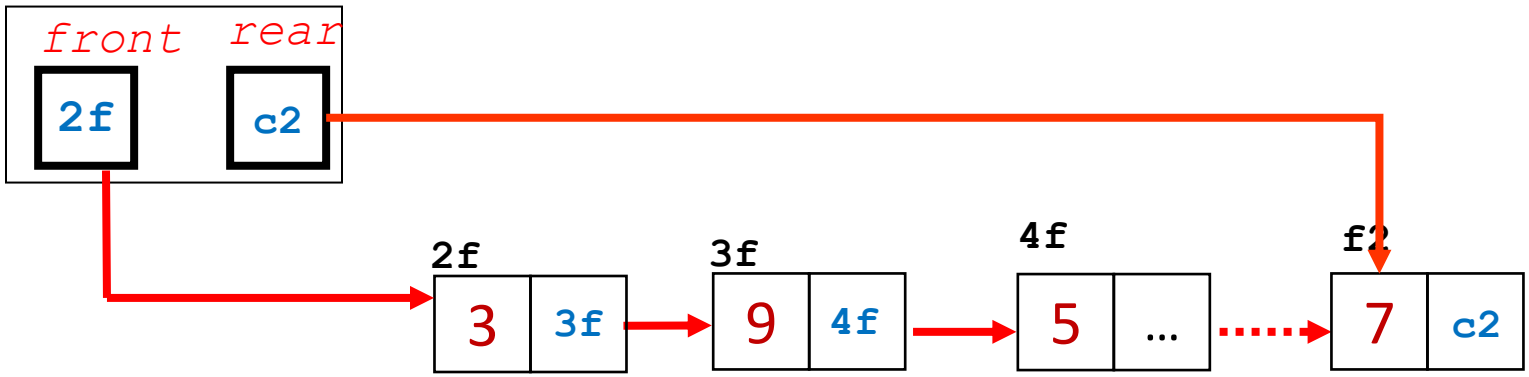
```
void Init(QUEUE &q)
{  q->front = NULL;
   q->rear= NULL;
}
```



6.3. Kiểm tra hàng đợi rỗng

```
bool IsEmpty(QUEUE &q)
{
   return (q.front == NULL);
}
```

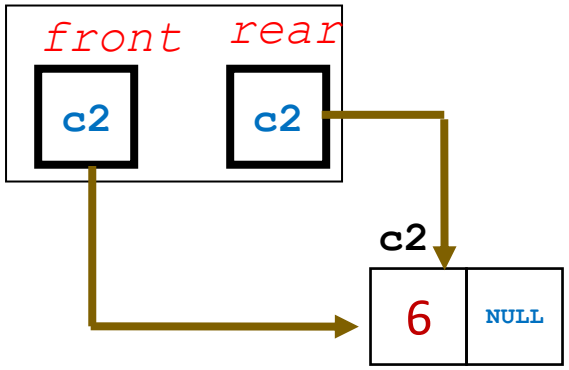


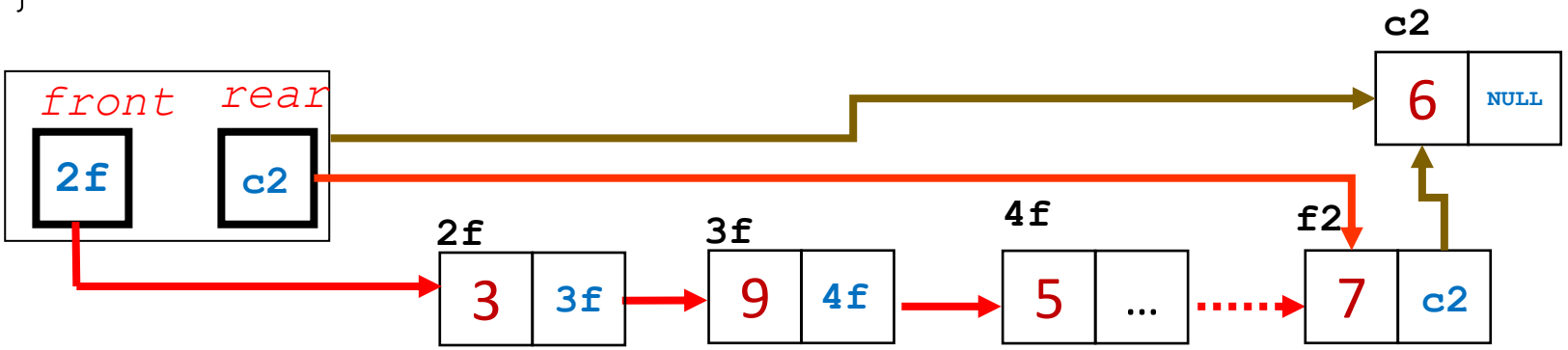


6.4. Thêm 1 NODE vào QUEUE(minh họa khi cấu trúc NODE đơn giản)

- Thêm vào cuối DSLK nên sẽ thay đổi giá trị của Rear.

```
void Insert_queue(QUEUE &q, int x)
{       QUEUE p;
        p = new node;
        p->info = x;
        p->next=NULL;
        if (q.Front==NULL)
             q.Front=p;
        else
             q.Rear->next=p;
        q.Rear=p;
}
```

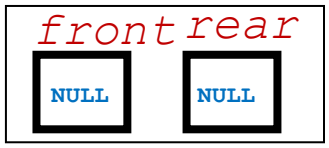


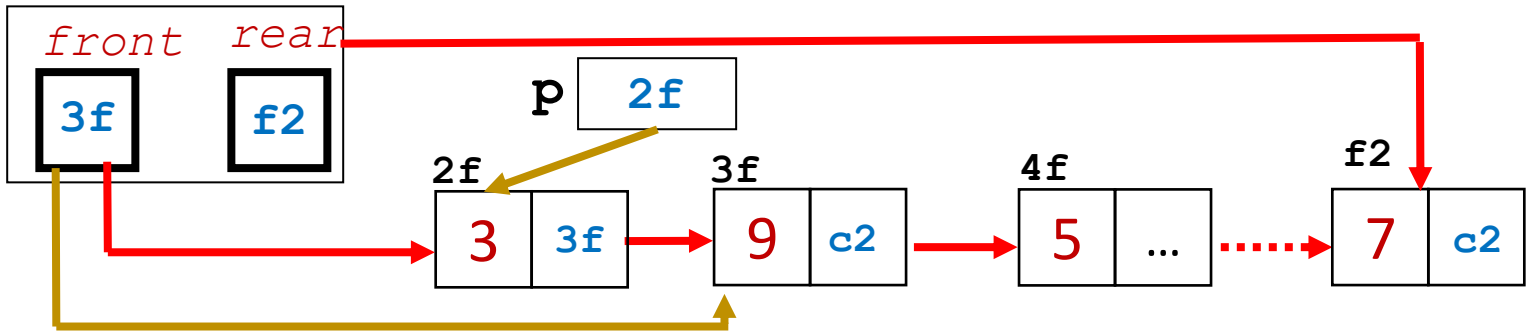


6.5. Lấy 1 NODE ra khỏi QUEUE (minh họa khi cấu trúc NODE đơn giản)

- Loại bỏ phần tử đầu DSLK ⇒ thay đổi giá trị của Front

```
int Delete_queue(QUEUE &q,int &x)
{       QUEUE p;
        if(q.Front==NULL)
             return 0;
        p = q.Front;     // nut can xoa la nut dau
        x = p->info;
        q.Front = p->next;
        delete p;
        return 1;
}
```





7. THỰC HÀNH

Lần lượt đưa vào và lấy ra trên hàng đợi ưu tiên theo giá trị lớn nhất (*max -priority queue*). Yêu cầu vẽ hình minh họa từng bước của *Priority Queue*:

i. 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

ii. 3, 6, 9, 1, 4, 7, 10, 2, 5, 8

7.2. Lần lượt viết chương trình quản lý 1 Queue bằng cách sử dụng danh sách kề và DSLK đơn với cấu trúc dữ liệu như sau

```
struct QUEUE
{     char  *qArray;
      int   qMax;
      int   qFront;
      int   qRear;
};
```

```
struct NODE
{  int data;
   NODE *pNext;
};
```

```
struct QUEUE
{  NODE * qFront;
   NODE * qRear;
};
```

- Tạo 1 queue gồm n node, với n do người dùng nhập vào
- Để tiện cho việc cài đặt các chức năng khác, viết hàm ShowQueue để liệt kê các giá trị đang có queue
- Thêm node vào danh sách (hàm EnQueue). Trong đó, nếu sử dụng cấu trúc dạng danh sách kề thì hàm sẽ thực hiện dồn mảng về đầu ngay sau khi phần tử cuối được dùng (`q.qRear=qMax`)
- Lấy 1 node ra khỏi danh sách (hàm DeQueue).
- Cho nhập số lượng node cần lấy ra (k), nếu còn đủ k node thì thực hiện lấy ra, ngược lại thông báo không đủ k node trong queue.

7.3. Game_01

- Tạo 1 DSLK gồm 6 số (1, 2, 3, 4, 5, 6)
- Với mỗi lượt xoay DSLK sẽ đưa số đang ở đầu DSLK thành số cuối DSLK. Ví dụ:
  - Ban đầu trạng thái của queue là 1, 2, 3 ,4 ,5, 6
  - Tại lượt chơi thứ 1 trạng thái của queue là 2, 3, 4, 5, 6, 1
  - Tại lượt chơi thứ 2 trạng thái của queue là 3, 4, 5, 6, 1, 2
- Cho người dùng nhập số lần cần xoay (n). In ra DSLK sau n lần xoay.

7.4. Game_02

- Tạo 1 DSLK gồm 52 node, với data của mỗi node là số nguyên, có giá trị từ 0-51, được phát sinh ngẫu nhiên và không trùng nhau.
- Mỗi lượt chơi:
  - Chương trình lần lượt chia xen kẽ cho 2 người chơi (A và B) mỗi người 1 số sao cho mỗi người đều có 3 số.
  - Cho biết tổng các số của mỗi người. Người thắng cuộc là người có tổng các số là lớn nhất

7.5. Viết chương trình thực hiện yêu cầu sau:

Có khai báo về queue cần dùng như sau:

```
struct Queue
{     int  *qArray;
      int  qMax;
      int  qFront;
      int  qRear;
}QUEUE;
```

Với khai báo đã có: `QUEUE q1, q2, q3, q4, q5;`

Viết chương trình thực hiện các yêu cầu sau:

- ***6.4.1.*** Tách q1 thành 2 queue, sao cho:
  - Queue thứ nhất (q4) chứa các phần tử là số nguyên tố.
  - Queue thứ hai (q5) chứa các phần tử còn lại.

  Với prototype sau:

  ```
  void Tach(QUEUE q, QUEUE &q4, QUEUE &q5)
  ```

- ***6.4.2.*** Giả sử q1 và q2 đã được sắp tăng dần. Nối q1 và q2 thành q3 sao cho q3 vẫn có thứ tự tăng dần, theo prototype sau:

  ```
  void Nhap(QUEUE q1, QUEUE q2, QUEUE &q3)
  ```

Bài tập 7.6. Viết chương trình thực hiện yêu cầu sau:

Viết chương trình mô phỏng quy trình xếp hàng đặt vé xem phim như sau:

- ***Danh sách liên kết A*** chứa số ghế của các ghế trống trong rạp (ban dầu khởi tạo các số ghế từ 1 đến n).
- ***Danh sách hàng đợi B*** chứa số thự tự và tên của khách xếp hàng.
- ***Danh sách liên kết C*** chứa thông tin khách đã mua vé (số ghế, tên).
- ***Chức năng lấy số xếp hàng***: Ban đầu (khi B rỗng) thì khách đầu tiên sẽ có số thứ tự xếp hàng là 1, ngược lại thì số thứ tự xếp hàng là k+1 với k là số thứ tự của node cuối của B.
- ***Chức năng mua vé:*** Nếu còn ghế trống và có khách đang chờ mua vé thì xóa node khỏi B, lấy tên khách và số ghế khách chọn để thêm node vào C đồng thời loại số ghế đó khỏi A.
- ***Chức năng hủy vé:*** Xóa node khỏi C đồng thời thêm số ghế mới hủy vào A.
- ***Chức năng hiển thị:*** Hiển thị thông tin những vé đã bán (DSLK C).


Q&A
How
Why
When
Where
What
Who