TRƯỜNG ĐẠI HỌC PHENIKAA

Khoa khoa học cơ bản

BÀI TẬP LỚN KẾT THÚC HỌC PHẦN CTDL&TT

Sinh viên thực hiện: Nguyễn Hoàng Thạch

Lớp : CTDL&TT_1.1 (13.14FS) .1 _LT

Mã sv: 20010822

Hà Tĩnh, tháng 12, 2021

MỤC LỤC

BÀI 1:

1.1. Các thao tác danh sách liên kết.....2

1.2. Các thao tác hàng đợi.....5

1.3. Các thao tác ngăn xếp.....7

1.4. Duyệt cây nhị phân.....10

1.5. Tìm kiếm và chèn trên cây tìm kiếm nhị phân.....12

1.6. Giải thuật sắp xếp cơ bản.....14

1.7. Giải thuật sắp xếp Quicksort.....16

1.8. Giải thuật sắp xếp Heapsort.....17

1.9. Duyệt đồ thị theo chiều rộng và chiều sâu.....25

1.10. Giải thuật tìm đường đi ngắn nhất Dijkstra.....32

1.11. Giải thuật tìm đường đi ngắn nhất Bellman - Ford.....33

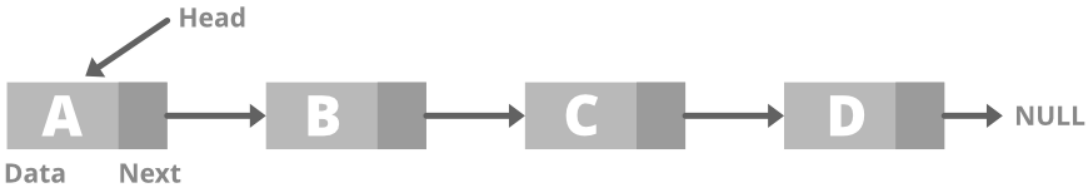
BÀI 2:

Biểu diễn những thuật toán sắp xếp căn bản……....39

BÀI 1:

Bài 1.1: ***Các thao tác trên danh sách liên kết***

Danh sách liên kết cũng giống như mảng là một cấu trúc dữ liệu tuyến tính tuy nhiên nó không giống mảng ở chỗ các nút được liên kết thông qua sử dụng con trỏ. Mỗi nút chứa một giá trị là Data và một con trỏ Next trỏ đến địa chỉ nút tiếp theo của danh sách liên kết đó. Nút được xem như là nút cuối cùng khi mà còn trỏ Next trỏ đến NULL.



Mảng có khả năng truy cập đến các nốt rất nhanh và dễ dàng tuy nhiên với danh sách liên kết việc truy cập vào các nút ở trong danh sách phải bắt đầu từ nút Head là nút đầu tiên của danh sách.

Tuy nhiên có những thao tác của danh sách liên kết nó sẽ tối ưu hơn so với mảng

Các thao tác trên danh sách liên kết bao gồm:

+) Liệt kê tất cả các phần tử trong danh sách liên kết (Print List)

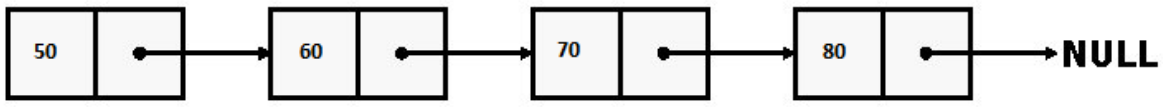
+) Tìm con trỏ chỉ tới phần tử có chỉ số là i (Find Node)

+) Chèn phần tử vào danh sách (InsertNode)

+) Xóa phần tử ra khỏi danh sách (Remove Node)

Trong đó, việc chèn phần tử và xóa phần tử đối với mảng sẽ rất khó khăn nhưng với danh sách liên kết thì ngược lại

Để mô tả các thao tác trên ta khởi tạo sẵn một ví dụ:

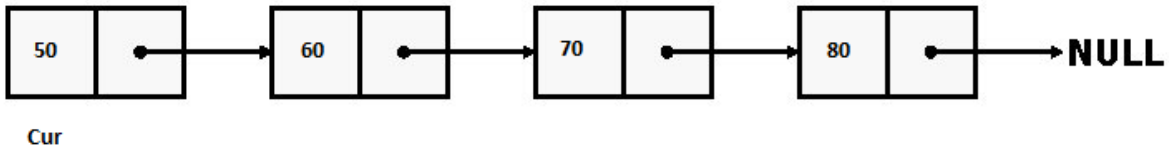


N=4, với head là có giá trị là 50 và 3 nút tiếp theo với giá trị là 60,70,80

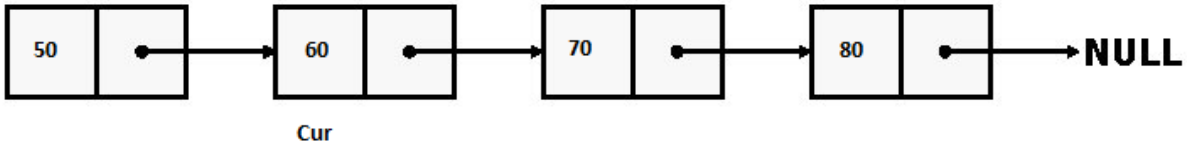
+) **Liệt kê tất cả các phần tử trong danh sách liên kết:**

Truyền con trỏ head tương ứng với danh sách liên kết trên vào hàm PrintList()

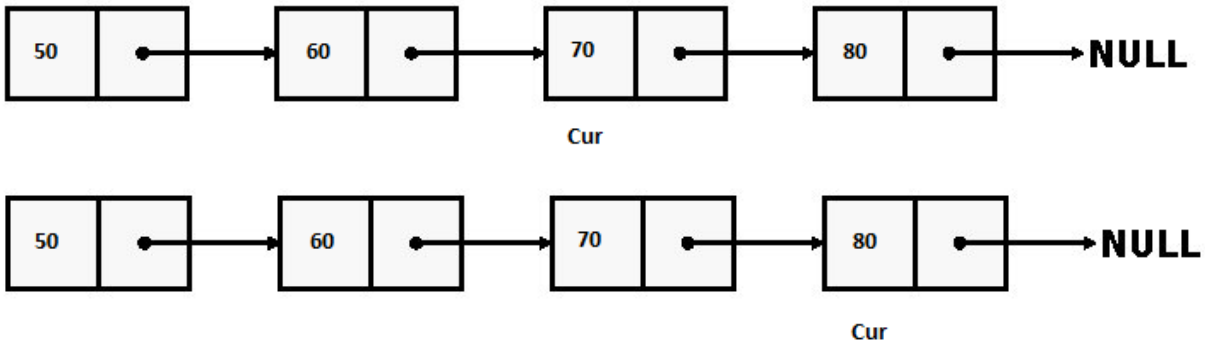
Ta sẽ tạo một con trỏ Cur để thực hiện truy cập danh sách và ban đầu nó được gán bởi head



In ra giá trị của nút con trỏ Cur ở đây là 50, kiểm tra xem cur->next có phải bằng NULL hay không rồi gán cur cho cur->next để di chuyển con trỏ đến nút tiếp theo



Cứ tiếp tục như trên:



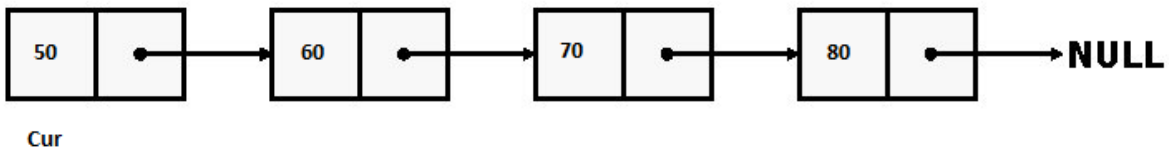
Cho đến khi ta kiểm tra cur->next của nút thứ 4 = NULL thì ta kết thúc vòng lặp và ta sẽ được kết quả như sau:

50 60 70 80

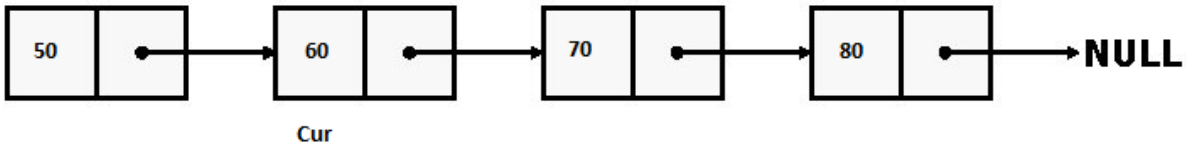
+) **Tìm con trỏ chỉ tới phần tử i:**

Để tìm con trỏ của nút thứ i,cũng tương tự như cách truy cập của PrintList() ta truy cập bắt đầu từ con trỏ head và gán cur cho cur-> next để truy cập nút tiếp theo, với mỗi lần truy cập ta sẽ giảm từ từ i đi 1 cho đến khi nào mà i=0 thì thực hiện kết thúc hàm FindNode() và trả con trỏ vị trí của nút có chỉ số bằng i

Ví dụ với i =2:



Giảm i đi 1 => i=1;



Giảm i đi 1 => i=0;

Đã tìm ra được con trỏ tại nút có giá trị là 60

Tuy nhiên trường hợp không thỏa mãn khi mà danh sách rỗng (head=NULL) hoặc là không tồn tại nút thứ i (i > n) hàm sẽ trả về giá trị NULL.

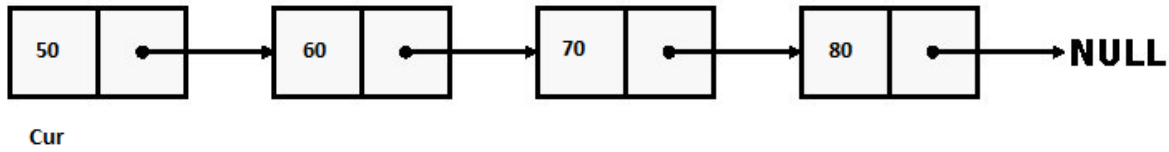
+) **Chèn phần tử vào danh sách:**

Để chèn một phần tử vào vị trí i của một danh sách liên kết ta áp dụng hàm FindNode() tìm kiếm con trỏ của nút có thứ tự trước i là i-1

Sau đó giá trị con trỏ next của nút thứ i-1 sẽ được gán cho địa chỉ nút mà ta cần chèn và giá trị con trỏ next của nút mới sẽ được gán cho địa chỉ nút thứ i trong quá khứ khi mà danh sách liên kết chưa chèn nút mới vào

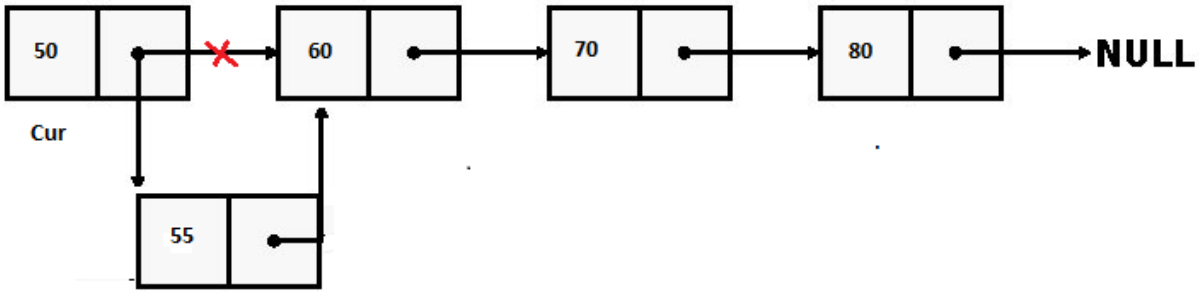
Ví dụ đối với i=1 và nút mới có giá trị là 55:

Sử dụng FindNode() đối với i=1 ta thấy con trỏ cur sẽ dừng lại ở vị trí như hình



Ở đây ta thấy Cur đang dừng ở con trỏ head ta bắt đầu gán cur->next = con trỏ nút mới;

Và con trỏ next của con trỏ nút mới sẽ bằng con trỏ của nút tiếp theo trước đó của phần tử thứ i=0



Đối với trường hợp chèn vào vị trí 0 hoặc danh sách rỗng (head=NULL) thì sẽ chỉ cần gán trực tiếp NewNode->next =head;

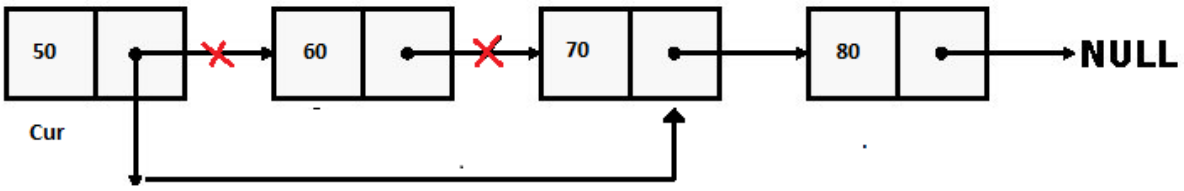
+) **Xóa một nút khỏi danh sách:**

Tương tự như thêm một nút ta áp dụng FindNode() để tìm nút có vị trí i-1

Sau đó cho nút có vị trí i-1 con trỏ next sẽ chỉ thẳng đến nút có vị trí i+1;

Và cuối cùng là giải phóng bộ nhớ cho node i bằng lệnh free();

Ví dụ đối với i=1;



Bài 1.2: Các thao tác trên hàng đợi

Hàng đợi có một cách hoạt động giống như so với hàng đợi ở ngoài đời thật. Với nguyên lí FIFO (First in First out) phần tử được đẩy vào trước sẽ là phần tử được xử lí và lấy ra trước.

Hàng đợi cũng dựa trên những cấu trúc dữ liệu khác như là mảng và danh sách liên kết.

Các thao tác trên hàng đợi bao gồm:

+) Thêm phần tử vào hàng đợi (enqueue)

+) Lấy phần tử ra khỏi hàng đợi (dequeue)

+) Lấy giá trị nút đầu hàng đợi và không làm thay đổi cấu trúc (peek)

+) Kiểm tra xem hàng đợi có rỗng hay không (isEmpty)

+) **Thêm phần tử vào hàng đợi:**

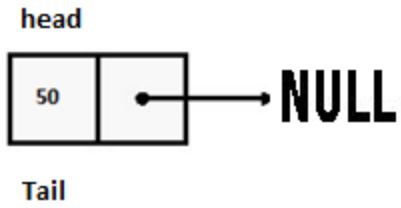
Thêm một phần tử vào hàng đợi nó tương đương với việc thêm một phần tử vào cuối danh sách liên kết cho nên nếu như hàng đợi quá dài thì sẽ không hiệu quả

Vậy nên chúng ta sẽ sử dụng con trỏ tail để trỏ cuối cùng của hàng đợi

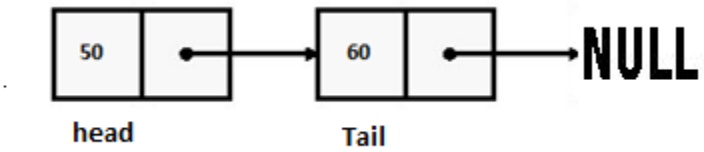
Nếu con trỏ tail = NULL khi đó hàng đợi đang trống và phần từ ta thêm vào là phần tử duy nhất nên sẽ được trỏ bởi cả head và tail. Nếu con trỏ tail khác NULL thì một node mới sẽ được khởi tạo sau tail với giá trị muốn thêm vào, cập nhật lại tail trỏ tới nút mới và kích thước hàng đợi tăng lên 1.

Ví dụ với một hàng đợi trống:

Thêm 50 vào hàng đợi:



Thêm 60 vào hàng đợi:

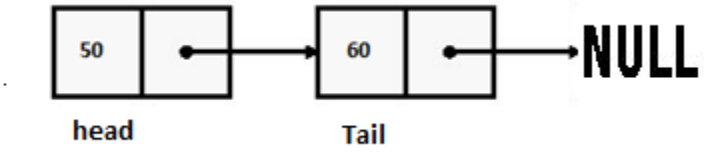


+) **Lấy 1 phần tử ra khỏi hàng đợi:**

Nếu mà hàng đợi không rỗng thì chúng ta có thể thực hiện lấy một phần tử ra khỏi hàng đợi bằng cách lấy giá trị của nút đầu tiên,sau đó sử dụng hàm removeNode của danh sách liên kết để xóa giá trị đã lấy ra đó. Và tất nhiên cuối cùng là giảm size của hàng đợi đi 1

Ví dụ:

N=2;



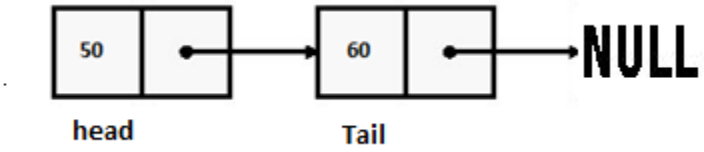
Sử dụng dequeue ta sẽ lấy ra được kết quả 50 và hàng đợi còn lại:

N=1;

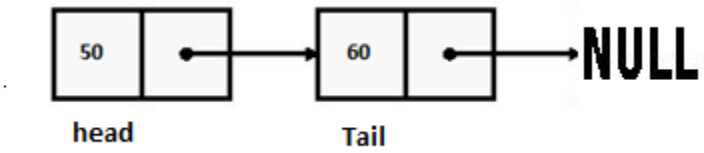
+) **Lấy giá trị nút đầu hàng đợi mà không hay đổi hàng đợi:**

Với một hàng đợi không rỗng ta thực hiện lấy giá trị của nút đầu tiên mà không cần xóa nút vừa lấy

Ví dụ:



Sử dụng peek ta lấy được kết quả là 50;

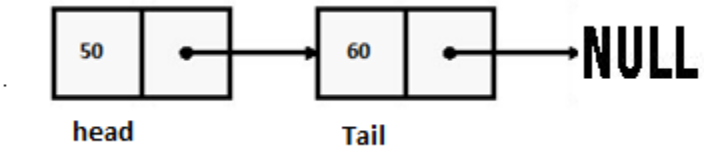


Và cấu trúc của hàng đợi vẫn không có gì thay đổi

+) **Kiểm tra hàng đợi có rỗng hay không:**

Hàm isEmpty cho ta thấy được nếu hàm trả về 1 tức là hàng đợi đang rỗng (size=0) nếu không sẽ trả về 0 (size!=0)

Ví dụ:

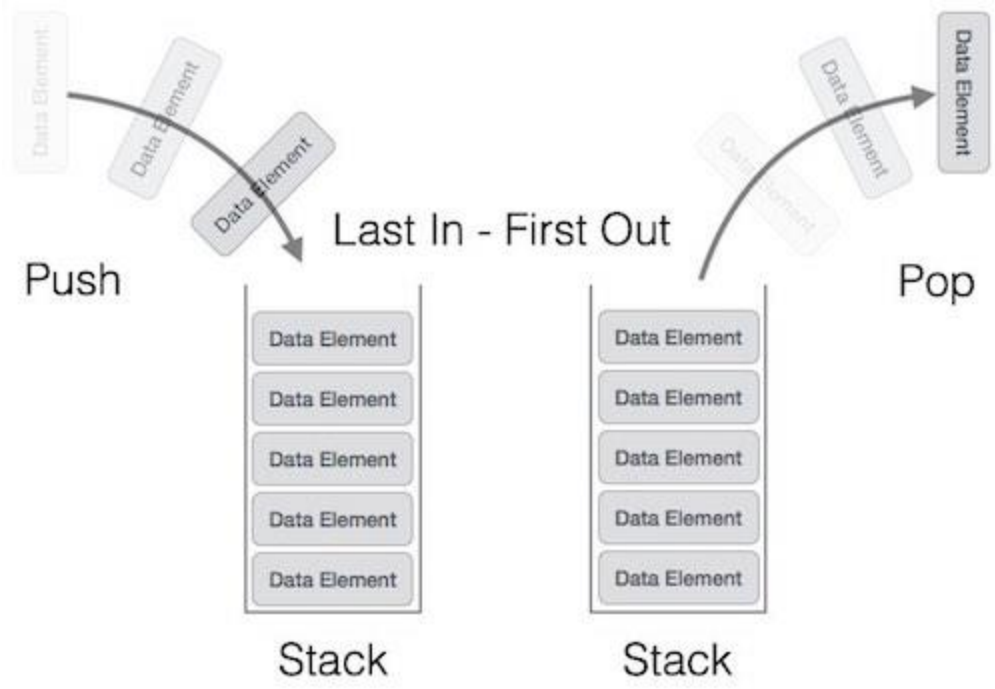


Với n=2 tức là size=2 thì hàm isEmpty ở đây sẽ trả về giá trị 0

Bài 1.3: Các thao tác trên ngăn xếp

Stack là một cấu trúc dữ liệu tuyến tính tuân theo một thứ tự cụ thể. Thứ tự ở đây có thể là LIFO (Last in first out) hoặc là FILO (First in last out).

Một ví dụ có thể biểu diễn được stack là xếp chồng đĩa ở trong căng tin. Chiếc đĩa nào nằm ở phía trên cùng sẽ được lấy ra trước.



Các thao tác trên stack bao gồm:

+) Thêm phần tử vào ngăn xếp (push)

+) Lấy một phần tử khỏi ngăn xếp (pop)

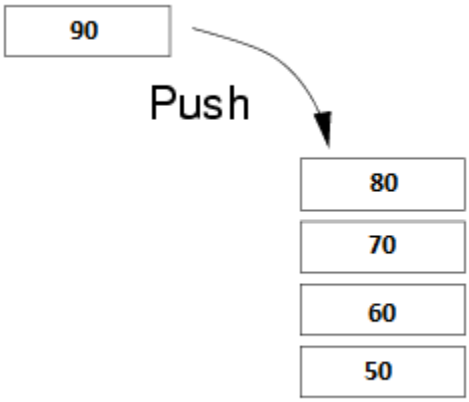
+) Lấy nút đầu ngăn xếp mà không làm thay đổi ngăn xếp (peek)

+) Kiểm tra ngăn xếp có rỗng không (isEmpty)

+) **Thêm phần tử vào ngăn xếp**

Ở cấu trúc dữ liệu ngăn xếp ta sẽ chỉ thực hiện thao tác ở đỉnh của nó kể cả việc thêm phần tử vào ngăn xếp. Có thể thêm vào đầu hoặc cuối ngăn xếp tùy vào cách cài đặt xem đầu hay cuối của danh sách là đỉnh.

Ví dụ đối với một ngăn xếp đã có 4 phần tử 50,60,70,80 ở đây ta sử dụng thứ tự LIFO:



Thêm 90 vào stack => nút mới sẽ nằm gọn ở đỉnh stack thay thế cho 80 trước đó là đỉnh

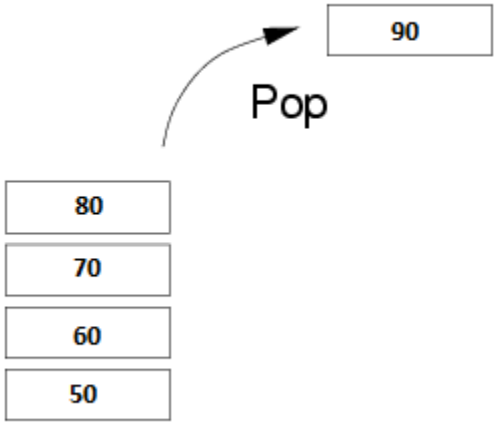
Size của danh sách +1=5

Chương trình dựa theo nguyên lý của danh sách liên kết và mảng

+) **Lấy một phần tử ra khỏi ngăn xếp**

Ngược lại đối với việc thêm phần tử chúng ta sẽ phải lấy một phần tử ra khỏi stack . Và tất nhiên lại tiếp tục thao tác trên đỉnh của nó. Nút bị lấy ra sẽ là nút ở đỉnh của stack và cuối cùng là giải phóng bộ nhớ được cấp phát cho nút đó.

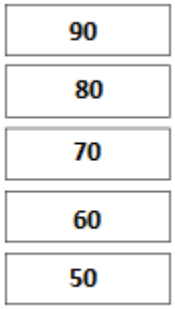
Ví dụ như ở trên:



Như vậy trước đó với size của stack là 5, đỉnh của stack là 90 thì ngay khi ta chạy hàm pop() phần tử 90 sẽ được lấy ra khỏi stack và size của stack bị giảm xuống còn 4. Và đỉnh stack lại quay trở lại được thay thế bởi 80.

+) **Lấy phần tử đầu ngăn xếp mà không làm thay đổi ngăn xếp**

Đơn giản là chỉ việc lấy giá trị của phần tử đỉnh mà không cần phải xóa nút từ đỉnh đi

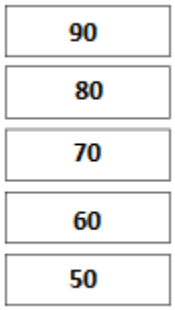


Khi chạy hàm peek kết quả sẽ trả về 90

+) **Kiểm tra ngăn xếp có rỗng không**

Cũng giống như đối với hàng đợi với mỗi danh sách tồn tại một giá trị size chứa đồ lớn của danh sách. Thì ngăn xếp cũng thế nếu size =0 tức là ngăn xếp rỗng, có thể kiểm tra theo cách khác khi mà đỉnh =NULL cũng cho thấy rằng ngăn xếp rỗng.Hàm kiểm tra sẽ trả về 1 nếu ngăn xếp không có phần tử nào và trả về 0 nếu ngược lại

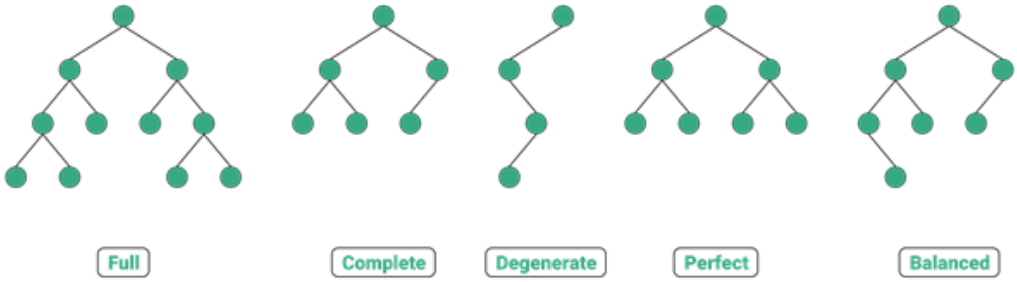
Ví dụ :



Danh sách trên có size=5 cho nên khi chạy hàm emptyStack sẽ trả về 0 vì ngăn xếp không rỗng

Bài 1.4: Duyệt cây nhị phân

Cây nhị phân là cây mà các nút có nhiều nhất là hai phần tử con. Vì với mỗi nút trong cây nhị phân chỉ có hai nút con nên chúng thường được gọi là nút con trái và nút con phải của nút cha.



Một cây nhị phân chứa những phần sau:

1.Data

2.Con trỏ đến nút con trái

3.Con trỏ đến nút con phải

Các thao tác trên cây nhị phân:

+) Duyệt theo thứ tự trước

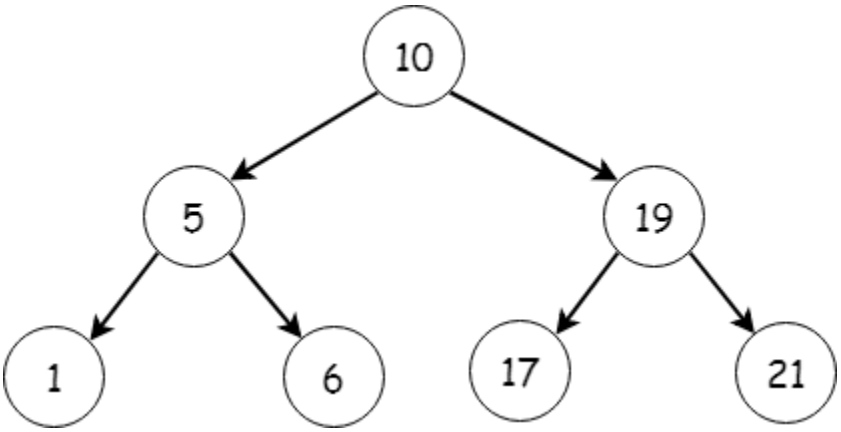
+) Duyệt theo thứ tự giữa

+) Duyệt theo thứ tự sau

+) **Duyệt theo thứ tự trước:**

Ở duyệt theo thứ tự trước ta xuất phát từ đỉnh gốc sau đó sang cây con trái rồi mới xét cây con bên phải

Ví dụ:



Ở ví dụ trên nút gốc là 10. Chúng ta di chuyển từ nút gốc sang nút con bên trái đó là 5. Từ đó lại tiếp tục duyệt theo cách làm ở nút gốc là sang nút con bên trái ở đây là 1. Khi nút 1 không còn

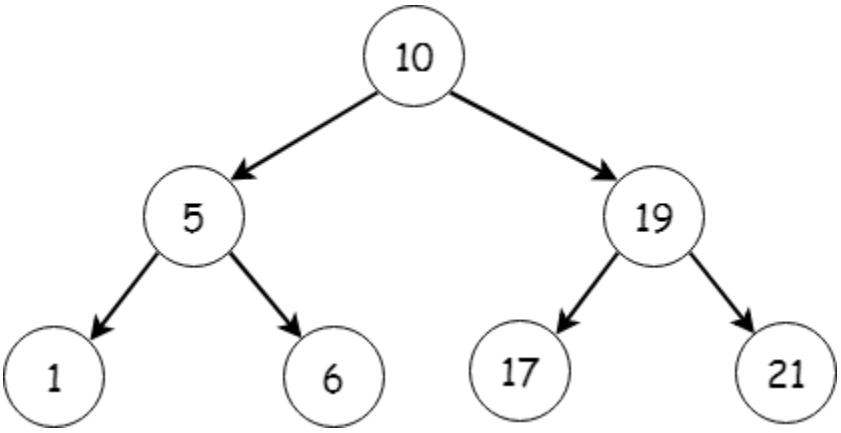
nút con nữa ta trở lại nút số 5 và bắt đầu duyệt sang phải ở đây là 6. Khi cây con bên trái nút gốc đã duyệt hết rồi ta quay trở lại nút gốc phải là chuyển tiếp để xét cây con bên phải.

Thứ tự duyệt của ví dụ trên sẽ là: 10 ->5->1->6->19->17->21

+) **Duyệt theo thứ tự giữa:**

Ở duyệt theo thứ tự giữa ta xuất phát từ nút con trái cùng sau đó sang đỉnh gốc rồi mới xét cây con bên phải

Ví dụ:

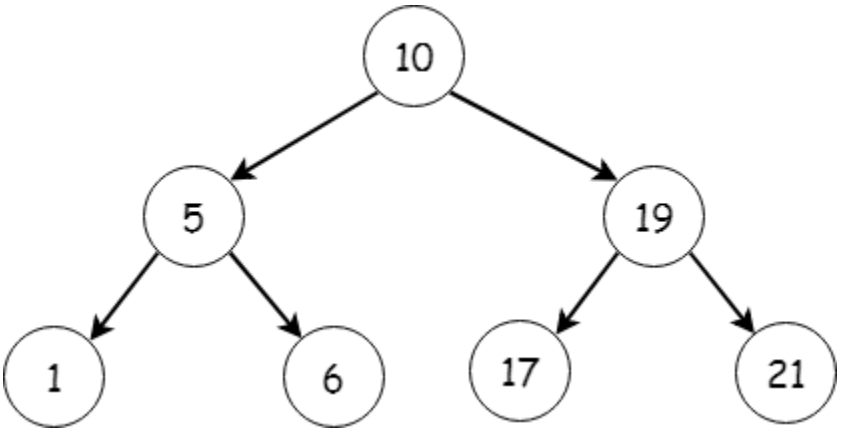


Kết quả: 1->5->6->10->17->19->21

+) Duyệt theo thứ tự sau:

Ở duyệt theo thứ tự sau nút gốc được duyệt sau cùng cho nên ta sẽ thăm cây con bên trái, rồi đến cây con bên phải cuối cùng là gốc

Ví dụ;



Kết quả:

1->6->5->17->21->19->10

Bài 1.5: Tìm kiếm và chèn trên cây tìm kiếm nhị phân

Cây tìm kiếm nhị phân là cấu trúc dữ liệu dạng cây nhị phân, mỗi nút chỉ có thể có tối đa 2 con và các nút con được sắp xếp, giá trị của node con bên trái nhỏ hơn node cha, giá trị của node con bên phải lớn hơn node cha. Giá trị nhỏ nhất chính là giá trị của node con trái nhỏ nhất, giá trị lớn nhất chính là giá trị của node con phải lớn nhất.

Các thao tác trên cây tìm kiếm nhị phân:

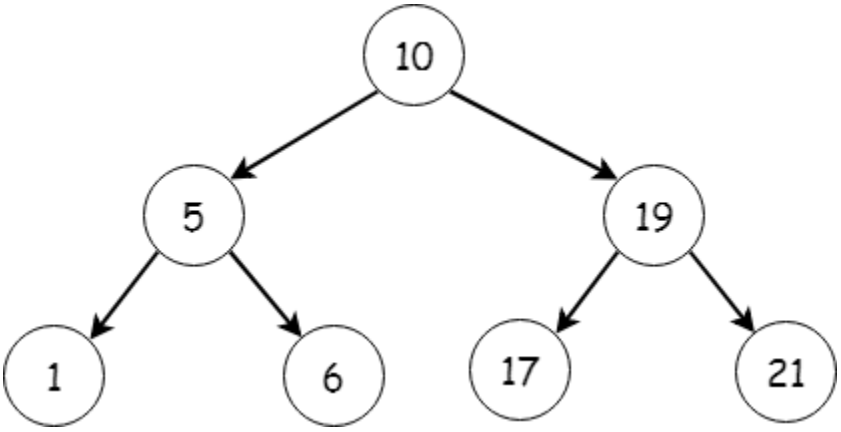
+) Tìm kiếm trên cây tìm kiếm nhị phân

+) Chèn một nút trên cây tìm kiếm nhị phân

+) **Tìm kiếm trên cây tìm kiếm nhị phân:**

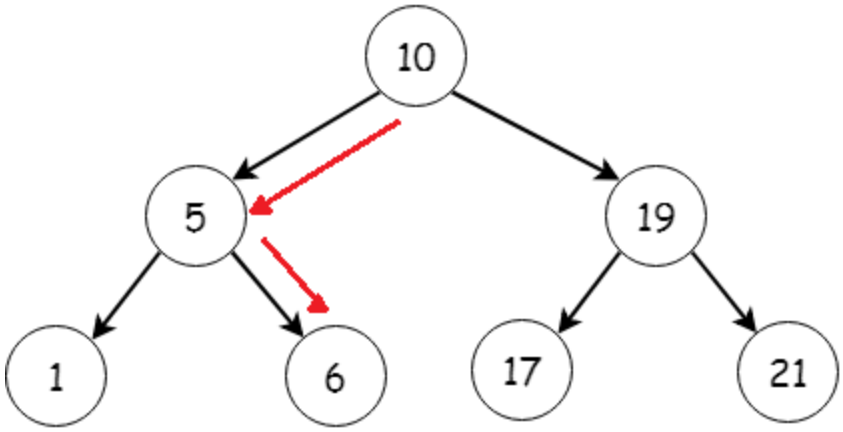
Để tìm kiếm một giá trị, ban đầu ta so sánh số cần tìm với nút gốc, nếu số đó nhỏ hơn giá trị nút gốc thì số đó sẽ chỉ nằm ở nửa bên trái nút gốc mà thôi , ngược lại đối với việc giá trị phần tử lớn hơn nút gốc thì số đó sẽ nằm ở nửa bên phải nút gốc. Nếu nó nằm ở bên trái ta chỉ cần cứ tiếp tục di chuyển đến nút bên trái gốc và lại so sánh thêm một lần nữa, xác định vùng tìm kiếm của nó bên trái hay phải cho đến khi tìm ra được số cần tìm.

Ví dụ:



Ta tìm nút số 6 trên cây ta thấy 6<10 => 6 nằm ở cây con bên trái của nút 10

Di chuyển đến 5 ta thấy 6 >5=> 6 nằm ở bên phải của nút 5 và tìm ra được vị trí của 6

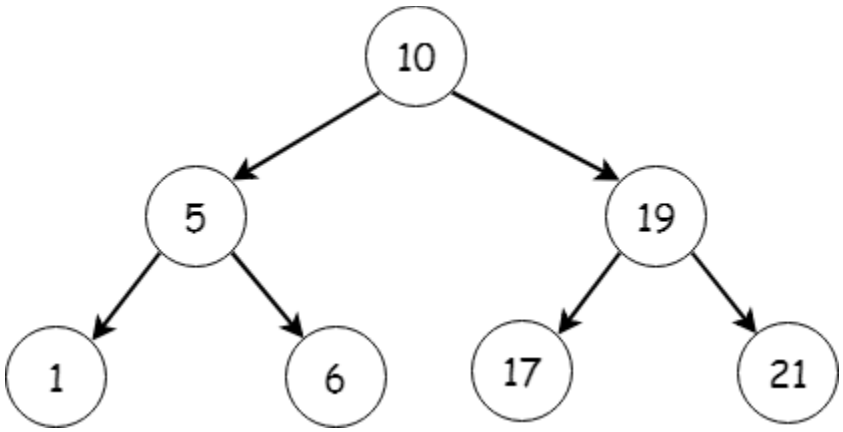


+) **Chèn một nút trên cây tìm kiếm nhị phân**

Để chèn một nút trên cây tìm kiếm nhị phân ta phải tìm vị trí mà số đó có thể thêm vào.

Ta sẽ áp dụng khả năng của hàm tìm kiếm trên cây nhị phân để tìm xem số đó đã tồn tại ở trên cây hay chưa, nếu không thì hàm sẽ trả về vị trí của nút cha sau khi tìm kiếm. Sau khi tìm thấy được nút cha việc còn lại là so sánh xem số đó lớn hơn hay nhỏ hơn nút cha, nếu lớn hơn thì nằm bên phải còn nếu nhỏ hơn thì nằm bên trái. Nút mới sẽ có con trỏ tham chiều = NULL

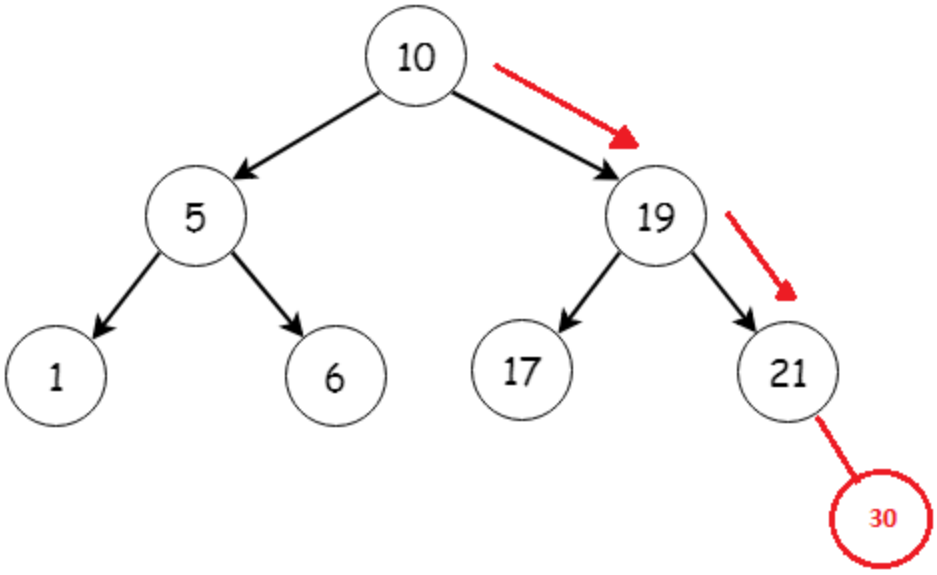
Ví dụ với việc ta thêm nút 30 vào cây nhị phân ở ví dụ trên:



Xét từ nút gốc ta thấy 30>10 cho nên 30 phải nằm bên phải của nút gốc

Tiếp đến ta thấy 30>19 cho nên 30 phải nằm bên phải nút 19

Di chuyển đến 21 ta thấy đây là nút cha cần tìm và 30>21 nên 30 phải là nút con bên phải của nút 21



Bài 1.6: Giải thuật sắp xếp cơ bản

Có 3 giải thuật sắp xếp cơ bản bao gồm:

+) Sắp xếp lựa chọn (Selection sort)

+) Sắp xếp thêm dần (Insertion sort)

+) Sắp xếp nổi bọt (Bubble sort)

+) **Sắp xếp lựa chọn:**

- Ở lượt thứ nhất, ta chọn trong dãy khóa k[1..n] ra khóa nhỏ nhất (khoá ≤ mọi khóa khác) và đổi giá trị của nó với k[1], khi đó giá trị khóa k[1] trở thành giá trị khóa nhỏ nhất.

- Ở lượt thứ hai, ta chọn trong dãy khóa k[2..n] ra khóa nhỏ nhất và đổi giá trị của nó với k[2].

- Ở lượt thứ i, ta chọn trong dãy khóa k[i..n] ra khóa nhỏ nhất và đổi giá trị của nó với k[i].

-Tới lượt thứ n - 1, chọn trong hai khóa k[n-1], k[n] ra khóa nhỏ nhất và đổi giá trị của nó với k[n-1].

Độ phức tạp của giải thuật : O(n^2)

Ví dụ với dãy ban đầu: 53 36 14 89 33 và có n=5

i	**K[i]**	**Lượt**			
		1	2	3	4

1	53	14	14	14	14
2	36	36	33	33	33
3	14	53	53	36	36
4	89	89	89	89	53
5	33	33	36	53	89

Dãy sau khi sắp xếp trở thành: 14 33 36 53 89

+) **Sắp xếp thêm dần:**

Với dãy khóa k[1..n]. Ta sẽ sắp xếp dãy k[1..i] trong điều kiện dãy k[1..i-1] đã sắp xếp rồi bằng cách chèn k[i] vào dãy đó tại vị trí đúng khi sắp xếp. Ta làm điều này qua việc tìm giá trị đầu tiên tìm được có giá trị lớn hơn giá trị của khóa thì bắt đầu di chuyển tất cả những phần tử lớn hơn khóa từ 1-> i-1 lên 1 vị trí để có khoảng trống cho khóa thực hiện hoán đổi, khi đó ta đươc k[1..i] đã sắp xếp.

Độ phức tạp của giải thuật : O(n^2)

Ví dụ đối với dãy ban đầu: 53 36 14 89 33 có n=5

Lượt	1	2	3	4	5
Khóa	53	36	14	89	33
1	53	36	14	14	14
2		53	36	36	33
3			53	53	36
4				89	53
5					89

Dãy sau khi sắp xếp**:** 14 33 36 53 89

+) **Sắp xếp nổi bọt:**

Khi sử dụng thuật toán sắp xếp nổi bọt, dãy các khóa sẽ được duyệt từ cuối dãy lên đầu dãy , nếu gặp hai khóa gần nhau bị ngược thứ tự (Sắp xếp tăng dần nhưng k[i]>k[i+1]) thì đổi chỗ của chúng cho nhau. Hết lần duyệt đầu tiên, khoá nhỏ nhất trong dãy khóa sẽ được chuyển về vị trí đầu tiên (k[1]) và ta tiếp tục thực hiện như trên với dãy k[2..n], rồi tiếp tục với dãy k[3..n]…..

Độ phức tạp của giải thuật : O(n^2)

Ví dụ đối với dãy ban đầu: 53 36 14 89 33 có n=5

i	**K[i]**	**Lượt**			
		1	2	3	4
1	53	14	14	14	14
2	36	53	33	33	33

3	14	36	53	36	36
4	89	33	36	53	53
5	33	89	89	89	89

Dãy sau khi sắp xếp: 14 33 36 53 89

Bài 1.7: Giải thuật Quicksort

Quicksort là một thuật toán chia để trị. Quicksort sẽ chia mảng ra thành nhiều mảng nhỏ mỗi mảng chọn một phần tử làm chốt (pivot) và phân vùng xử lí mảng xung quanh cái chốt đấy. Có nhiều cách chọn chốt ,có thể chọn một phần tử bất kì

Hàm quan trọng ở trong Quicksort đó là partitions. Mục tiêu của nó là giả sử một mảng với một phần tử làm chốt thì làm sao để cho chốt đứng đúng cái vị trí đối với mảng đã sắp xếp rồi, nói cách khác tất cả các nút bên trái nhỏ hơn hoặc bằng nút chốt và tất cả nút bên phải lớn hơn hoặc bằng chốt. Sau đó việc chúng ta cần làm là tiếp tục xử lí phần mảng bên trái nút chốt và phần mảng bên phải nút chốt bằng phương pháp như trên.

Như vậy tổng hợp thứ tự thực hiện Quicksort

+) Chọn một khóa ngẫu nhiên nào đó của dãy làm "chốt" (pivot). Mọi phần tử nhỏ hơn khoá "chốt" phải được xếp vào vị trí ở trước chốt, mọi phần tử lớn hơn chốt phải được xếp vào vị trí sau chốt.

+) Các phần tử trong dãy sẽ được so sánh với chốt và sẽ đổi vị trí cho nhau, hoặc cho chốt, nếu nó lớn hơn chốt mà lại nằm trước chốt hoặc nhỏ hơn chốt mà lại nằm sau chốt.

+) Khi việc đổi chỗ đã thực hiện xong thì dãy khoá lúc đó được phân làm hai đoạn: một đoạn gồm các khoá nhỏ hơn chốt, một đoạn gồm các khoá lớn hơn chốt còn khoá chốt thì ở giữa hai đoạn nói trên, đó cũng là vị trí thực của nó trong dãy khi đã được sắp xếp

+) Ở các lượt tiếp theo cũng áp dụng một kỹ thuật tương tự đối với các phân đoạn còn lại

Độ phức tạp: O(nlogn)

Áp dụng cho ví dụ bên dưới:

Dãy cần sắp xếp : 44 12 12 40 78 65 36

Lần	**K[i]**						
	44	12	12	40	78	65	36
1	->36	12	12	40	78	65	44<-
2	36	12	12	->40<-	78	65	44
3	->12	12	36<-	40	78	65	44
4	12	->12<-	36	40	78	65	44
5	12	12	36	40	->44	65	78<-
6	12	12	36	40	44	->65<-	78

Dãy số sau khi thực hiện quicksort: 12 12 36 40 44 65 78

Bài 1.8: Giải thuật sắp xếp heapsort

Heap sort là kỹ thuật sắp xếp dựa trên cấu trúc dữ liệu cây nhị phân hoàn chỉnh.

Heapsort gồm hai phần quan trọng:

+) Tạo đống

+) Sắp xếp

Đống tức là một cây nhị phân hoàn chỉnh trong đó các phần tử được lưu trữ theo một thứ tự đặc biệt sao cho giá trị của nút cha luôn lớn hơn (hoặc nhỏ hơn) giá trị của hai nút con. Đống có thể được biểu diễn bằng một cây nhị phân hoặc là một mảng .

Có thể biểu diễn dưới dạng mảng:

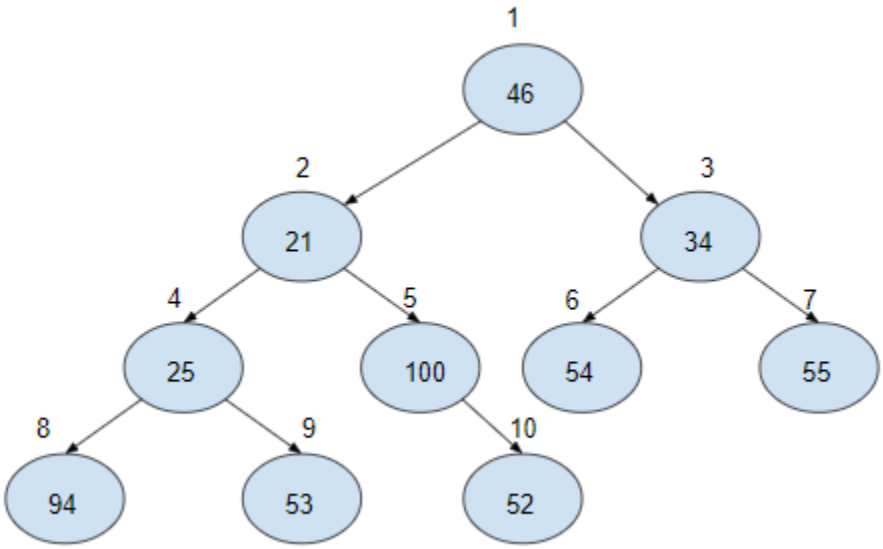
Đánh số các nút trên cây theo thứ tự lần lượt từ mức 1 trở đi, hết mức này đến mức khác và từ trái sang phải đối với các nút ở mỗi mức

Nếu như nút cha được lưu trữ ở vị trí I thì nút trái sẽ được lưu trữ ở 2*I+1 và nút bên phải được lưu trữ ở 2*I+2

+) **Tạo đống:**

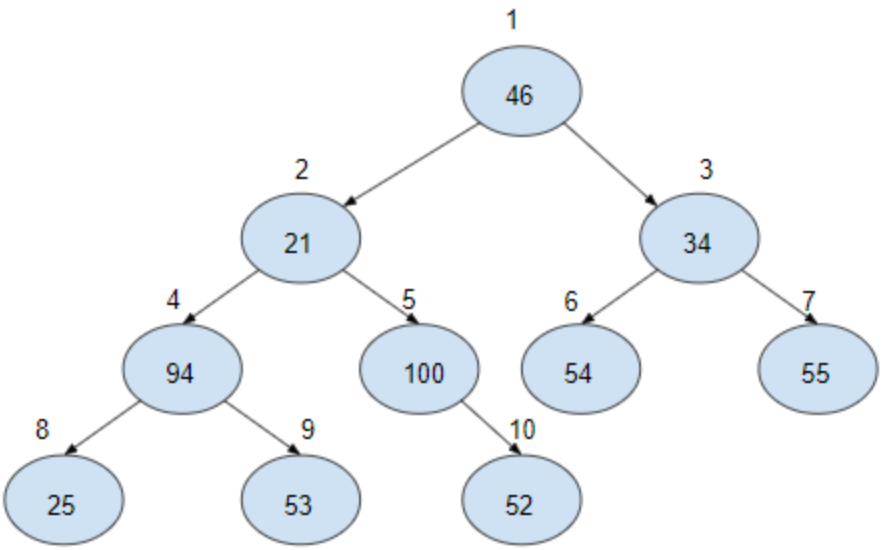
Ví dụ tạo đống:

Dãy 46 21 34 25 100 54 55 94 53 52



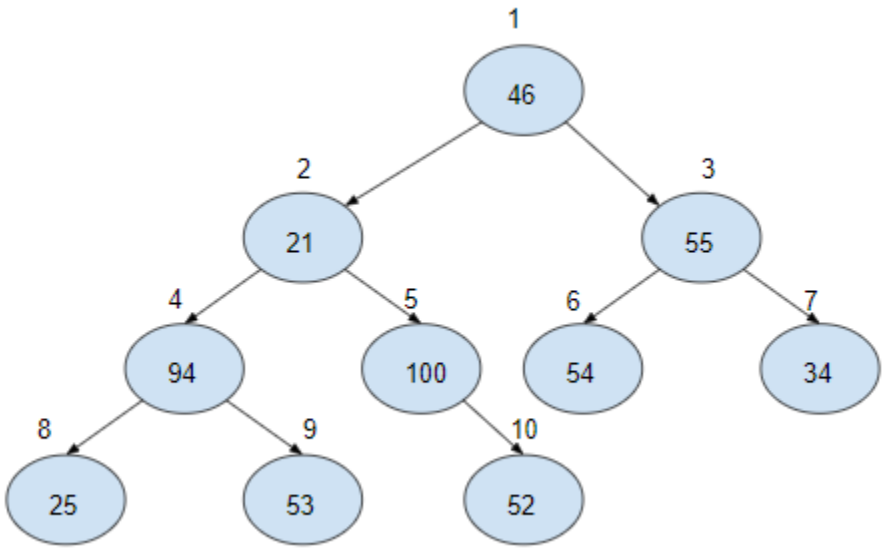
Ta bắt đầu vun đống với node floor(n/2)=5 (n=10) Với Adjust(5, 10)

Thấy cây con(5, 10) đang là cây nhị phân hoàn chỉnh có gốc(100) lớn hơn node con(52) nên không thay đổi



Di chuyển i đến 4 thực hiện adjust(4,10)

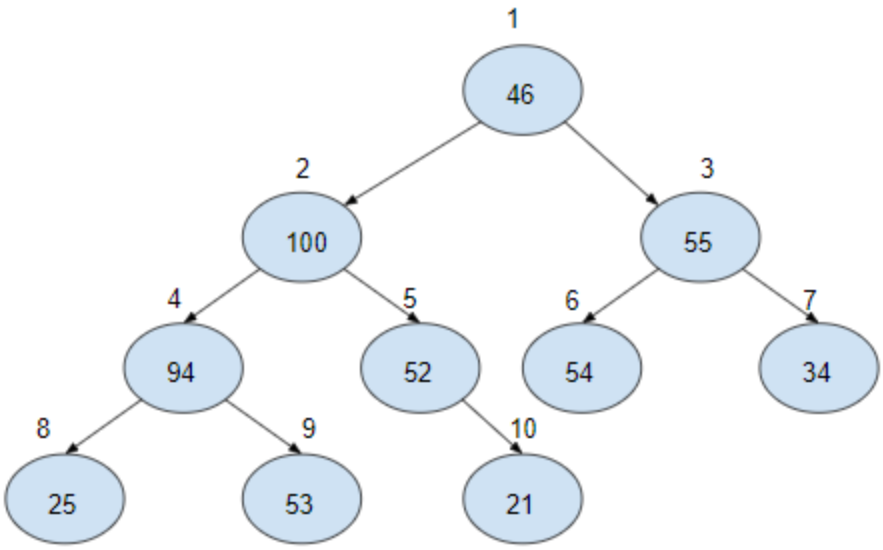
Thay đổi 94 và 25



i=3

Thực hiện adjust(3,10)

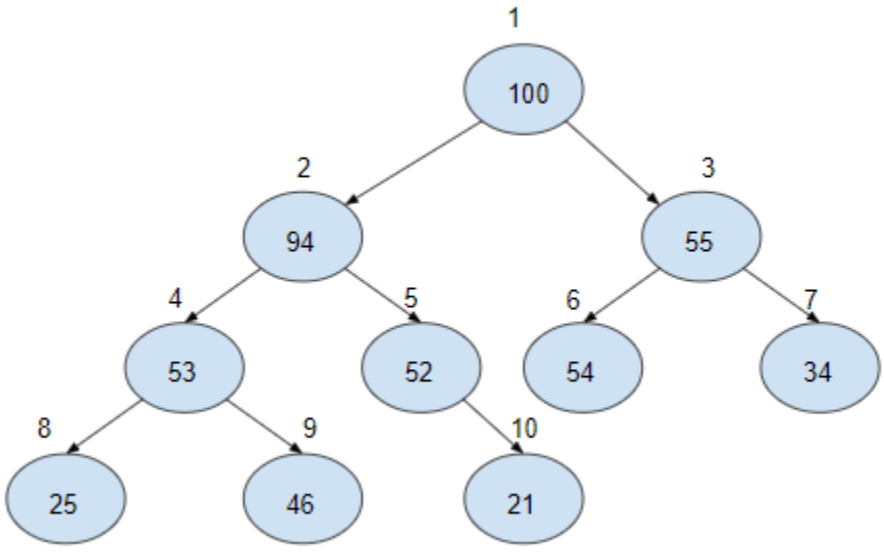
Thay đổi 55 và 34



i=2

Thực hiện adjust(2,10)

21 thay 100 và sau đó 21 thay đổi 52



i=1

Thực hiện adjust(1,10)

46 thay 100 và sau đó 46 thay đổi 94,46 lại thay đổi với 53

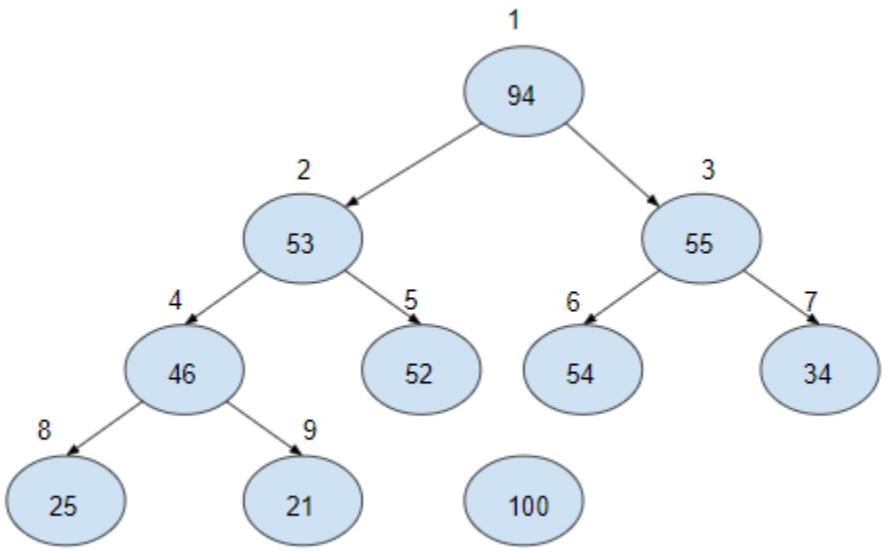
Ta được dãy sau khi vun đống : 100 94 55 53 52 54 34 25 46 21

+) **Sắp xếp**:

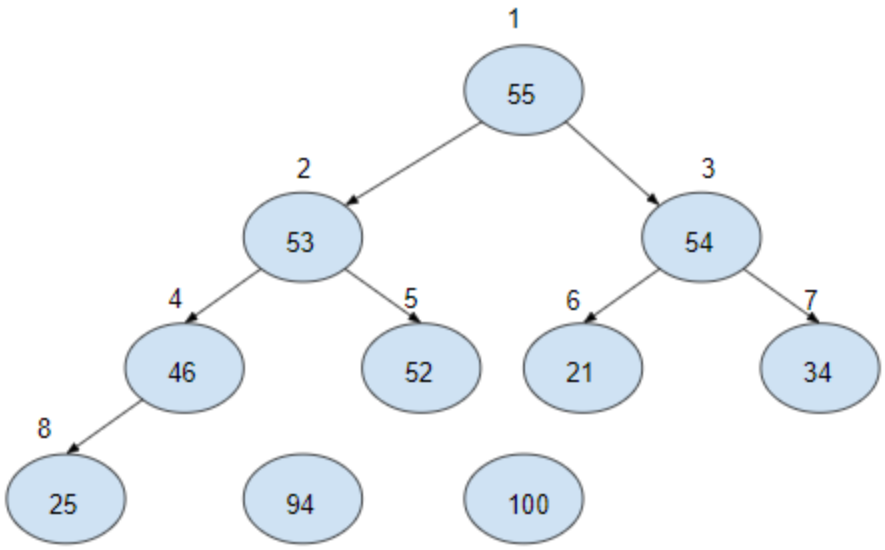
Đổi chỗ: Sau khi tạo đống ta tiến hành sắp xếp bằng cách đổi chỗ đỉnh đống với đáy đống và thực hiện vun đống.Việc sắp xếp được lặp lại cho đến khi đống còn lại 1 phần tử.

Vun đống: Sau khi đổi chỗ ta vun đống với số phần tử của đống giảm đi một và lại tiến hành đổi chỗ đỉnh đống và đáy đống. Sau mỗi lần sắp xếp số phần tử của đống giảm đi 1. Ta thu được các phần tử có giá trị giảm dần từ giá trị lớn nhất của đống.

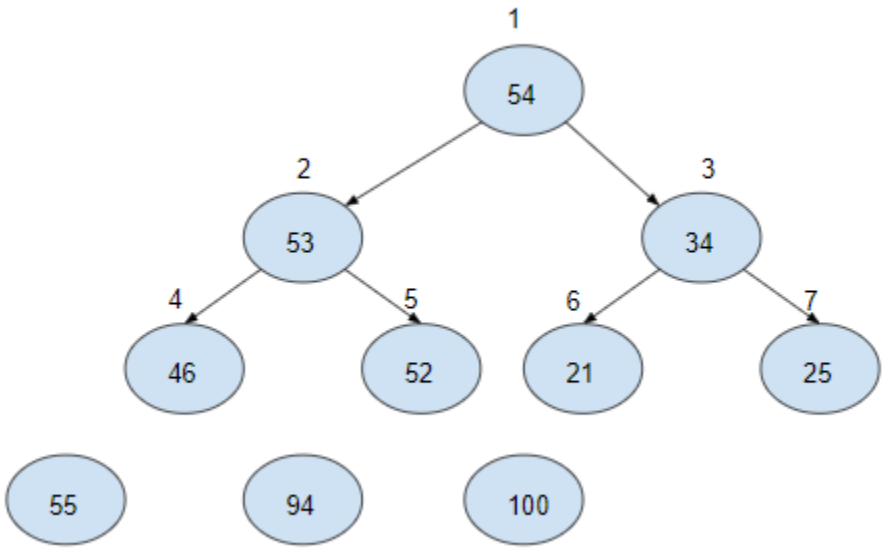
Ví dụ:



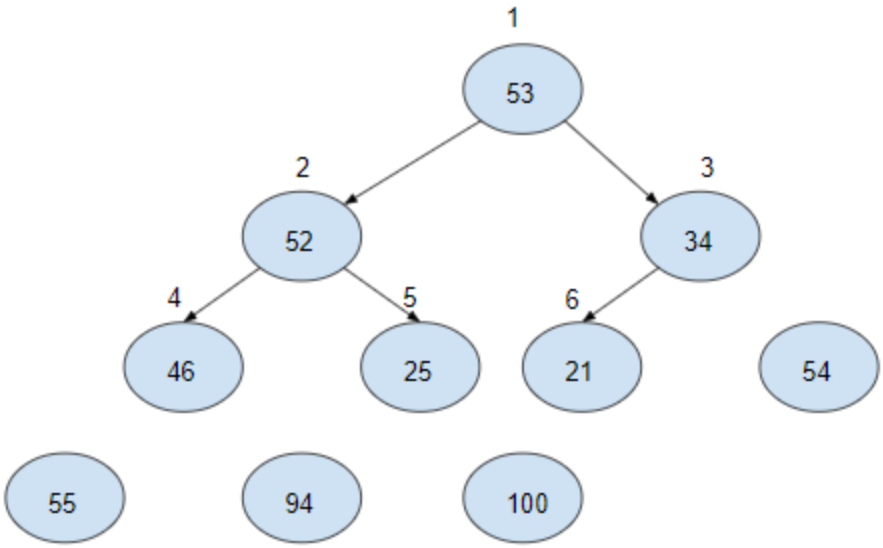
Đổi chỗ 100 cho 21 , size giảm đi 1 sau đó vun đống



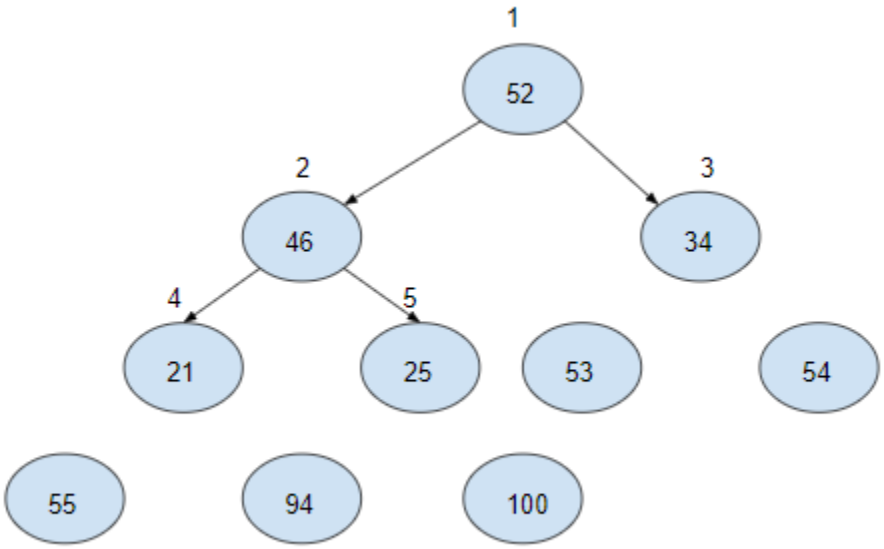
Đổi chỗ 94 cho 21 , size giảm đi 1 sau đó vun đống



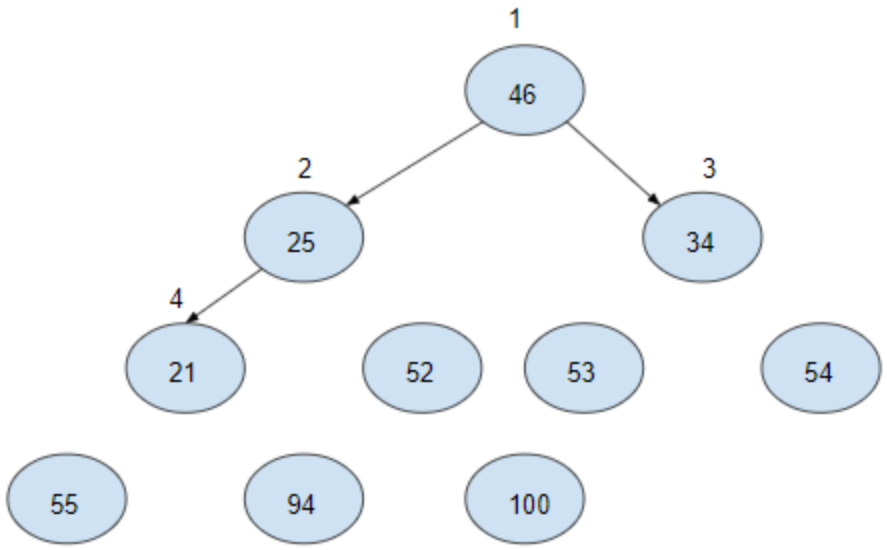
Đổi chỗ 55 cho 25 , size giảm đi 1 sau đó vun đống



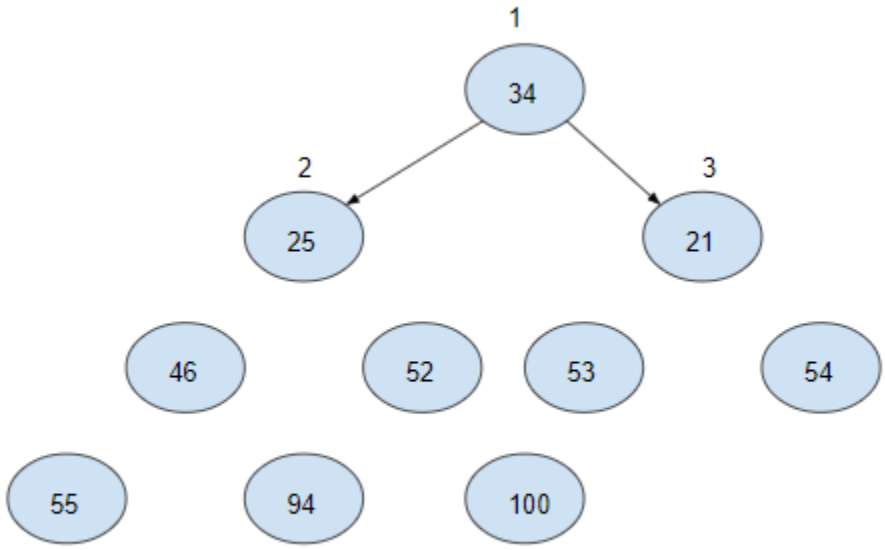
Đổi chỗ 54 cho 25 , size giảm đi 1 sau đó vun đống



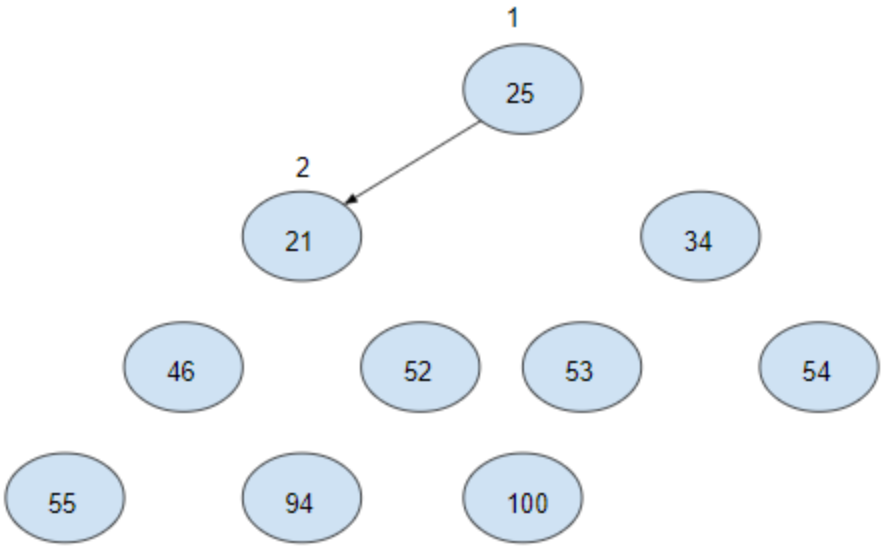
Đổi chỗ 53 cho 25 , size giảm đi 1 sau đó vun đống



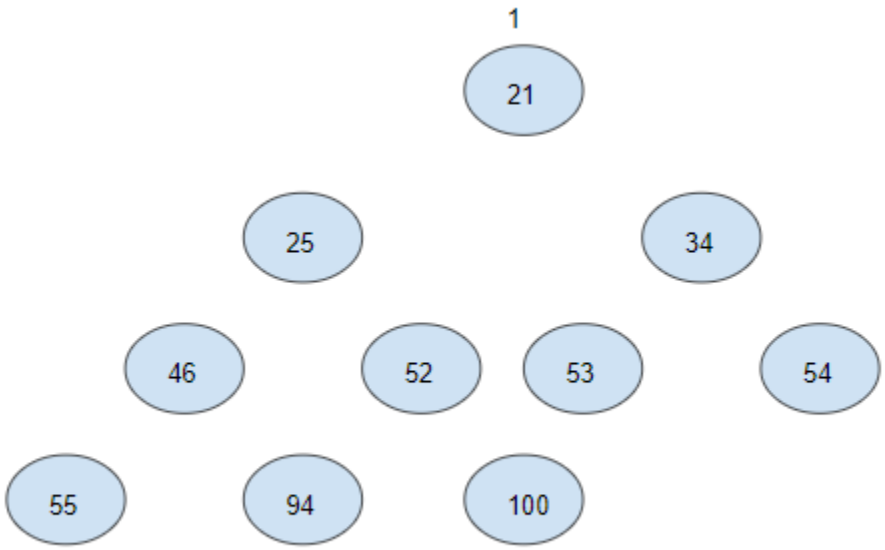
Đổi chỗ 52 cho 25 , size giảm đi 1 sau đó vun đống



Đổi chỗ 46 cho 21 , size giảm đi 1 sau đó vun đống



Đổi chỗ 34 cho 21 , size giảm đi 1 sau đó vun đống



Đổi chỗ 25 cho 21 , size giảm đi 1 và kết thúc khi size chỉ còn 1

Ta được dãy sắp xếp : 21 25 34 46 52 53 54 55 94 100

Bài 1.9: Duyệt đồ thị theo chiều rộng và chiều sâu

Đồ thị là một cấu trúc giữ liệu bao gồm các đỉnh và một tập cạnh mỗi một cạnh nối 2 đỉnh lại với nhau thể hiện mối quan hệ giữa các đỉnh trong đồ thị

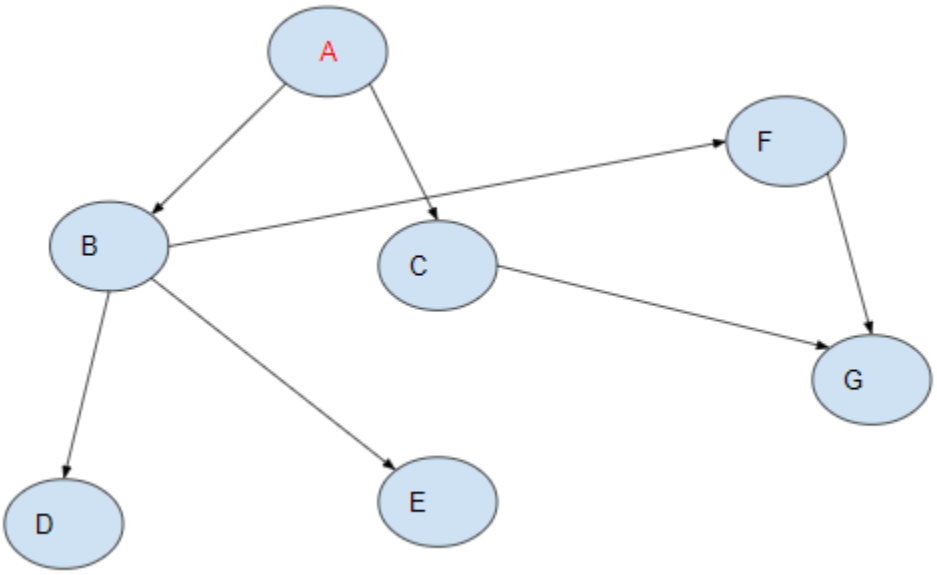
+) **Duyệt đồ thị theo chiều rộng (BFS):**

Với việc thực hiện duyệt theo chiều rộng ta duyệt tất cả các đường đi từ đỉnh hiện tại dừng chân rồi mới xét đỉnh tiếp theo và lại tiếp tục xét tất cả đường đi của đỉnh đó, cứ tiếp tục như vậy cho đến khi đã đi hết các đỉnh

Thứ tự thực hiện: Chọn đỉnh xuất phát -> tìm các đỉnh liền kề và lần lượt thăm từng đỉnh kề, với mỗi đỉnh ta thêm vào hàng đợi, đỉnh nào đã đi rồi ta đánh dấu lại -> thăm tất cả đỉnh liền kề của đỉnh tiếp theo -> Nếu tất cả các đỉnh đều được đánh dấu đã đi qua ta dừng lại

Ví dụ:

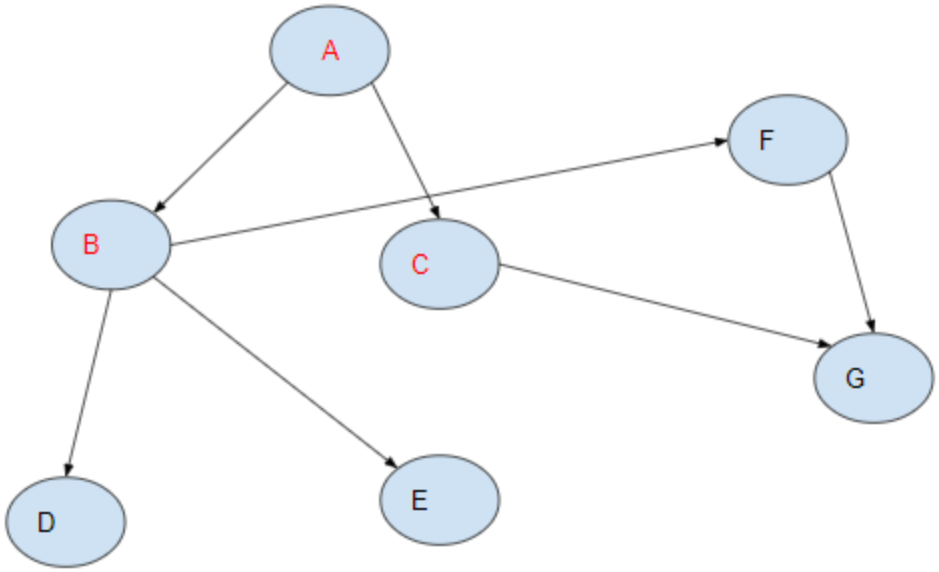
Bắt đầu từ đỉnh A



Cur : A

Hàng đợi A

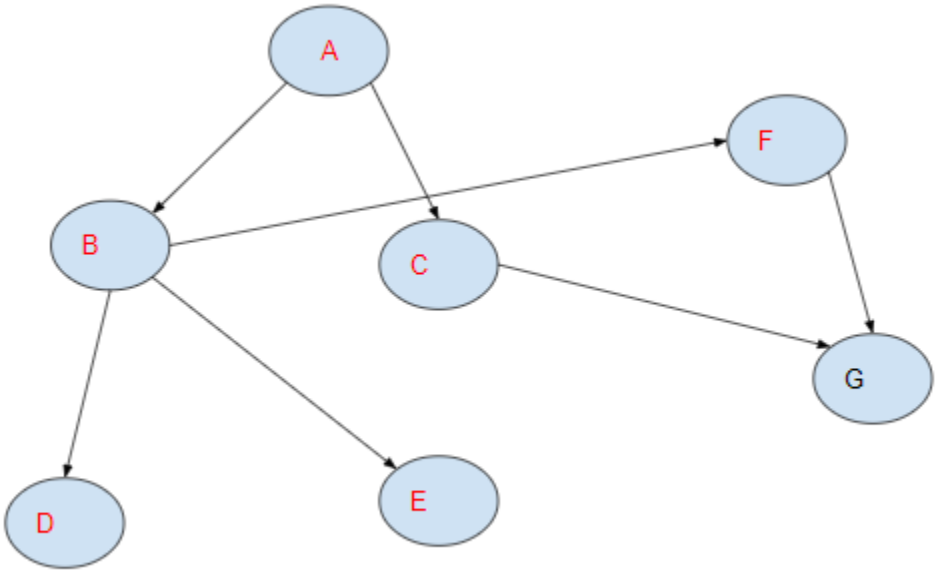
Thứ tự duyệt: A



Hàng đợi : B C

Cur: B

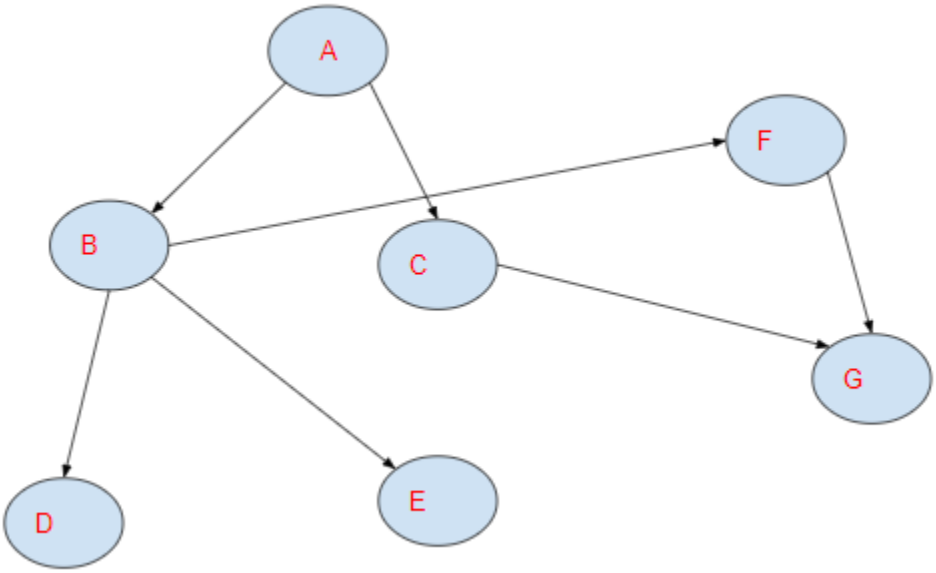
Thứ tự duyệt: A B C



Hàng đợi : C D E F

Cur: C

Thứ tự duyệt: A B C D E F



Hàng đợi : G

Cur: G

Thứ tự duyệt: A B C D E F G

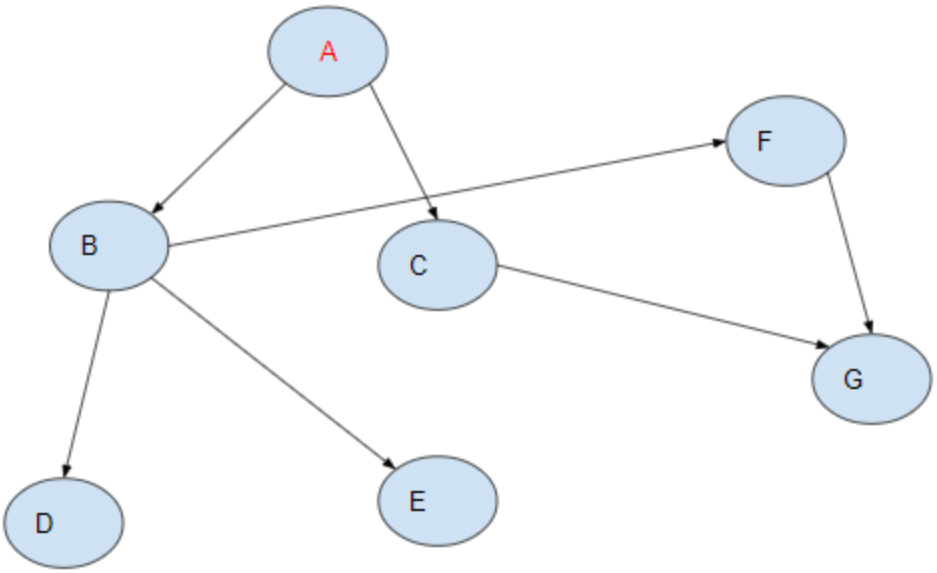
Đã duyệt hết các đỉnh nên không có đường đi nữa => kết quả thứ tự duyệt : A B C D E F G

+) Duyệt đồ thị theo chiều sâu (DFS):

Với việc thực hiện duyệt theo chiều sâu ta duyệt đường đi đầu tiên từ đỉnh hiện tại dừng chân rồi đến đỉnh tiếp theo ta lại tiếp tục xét đường đi đầu tiên của đỉnh đó đến một đỉnh khác, nếu từ một đỉnh không còn đỉnh nào để đi đến nữa ta qua trở lại đỉnh kề trước và tiếp tục xét một đường đi khác đường đã đi rồi. Cứ tiếp tục như vậy cho đến khi đã đi hết các đỉnh

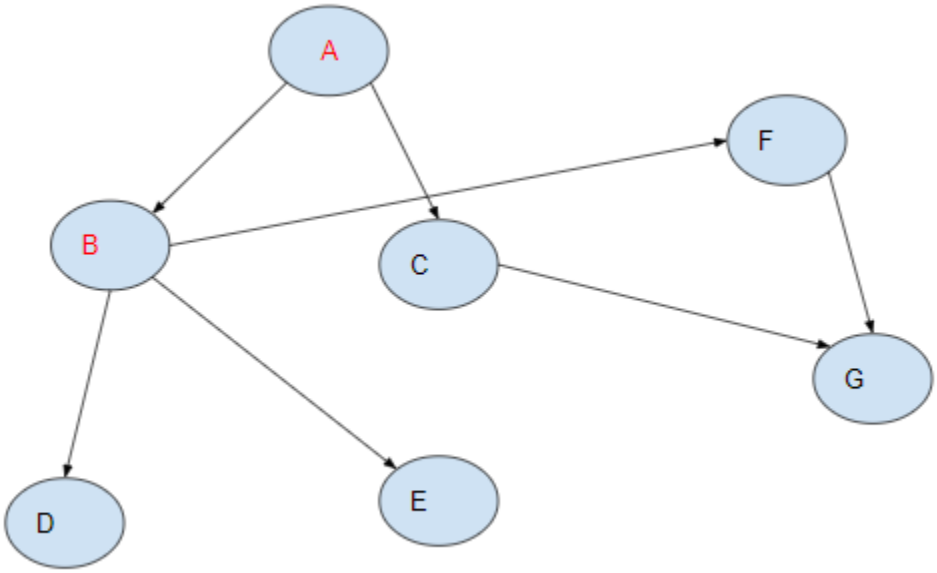
Thứ tự thực hiện: Chọn đỉnh xuất phát -> tìm đỉnh liền kề đầu tiên và thăm đỉnh đó, với mỗi đỉnh ta thêm vào hàng đợi, đỉnh nào đã đi rồi ta đánh dấu lại -> thăm đỉnh đầu tiên liền kề của đỉnh tiếp theo mà chưa từng thăm-> Sau khi thăm hết mọi cạnh ta quay lại đỉnh cũ và thăm một cạnh khác gần nhất chưa đi-> Nếu tất cả các đỉnh đều được đánh dấu đã đi qua ta dừng lại

Ví dụ:



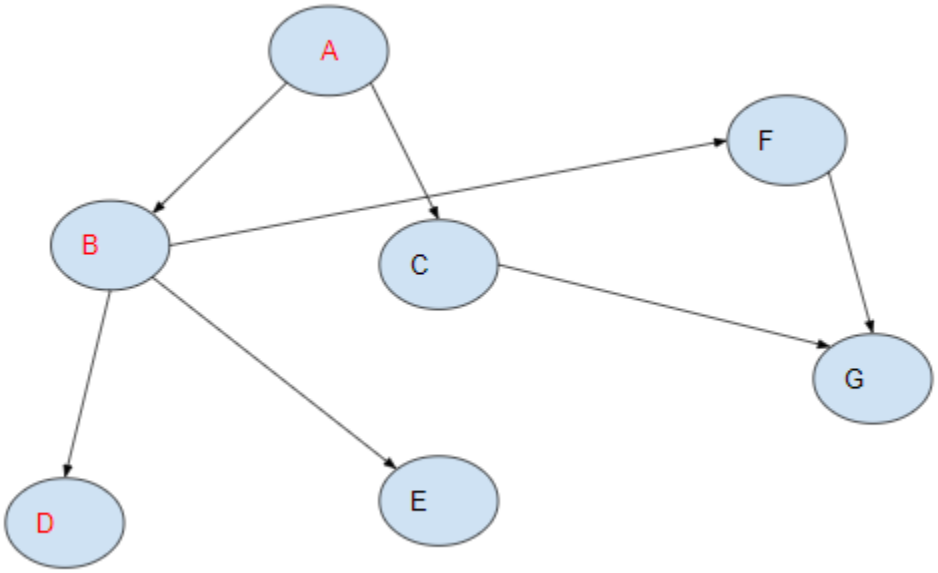
Cur: A

Thứ tự duyệt: A



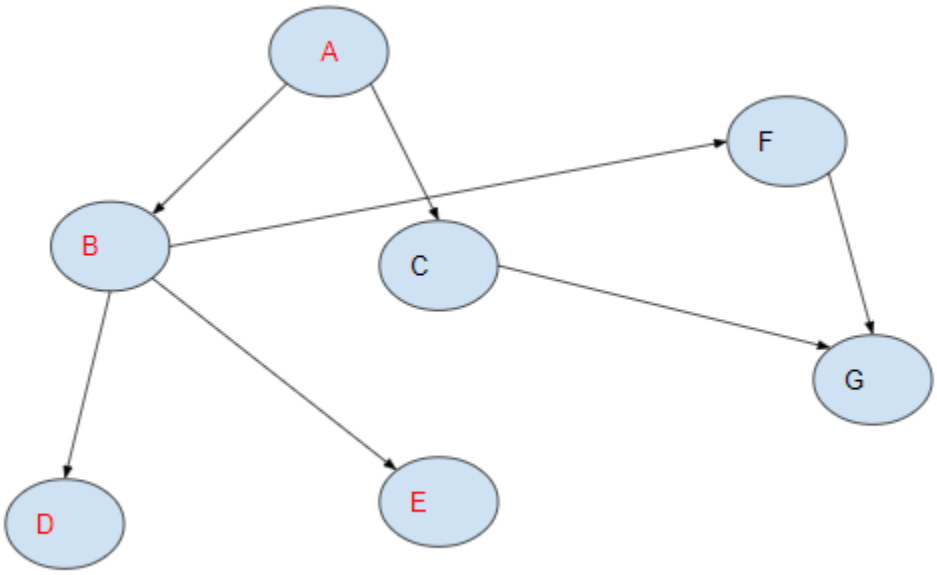
Cur: B

Thứ tự duyệt: A B



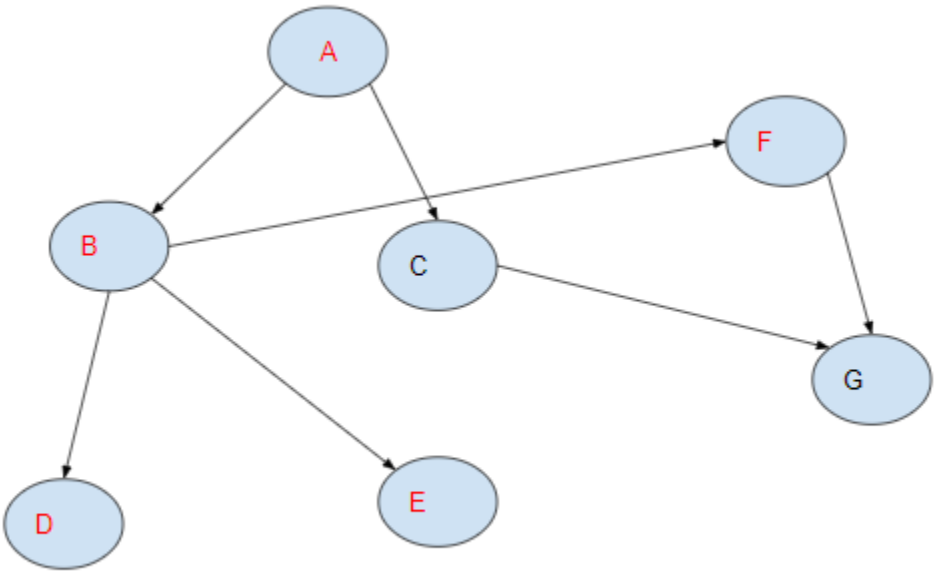
Cur: D

Thứ tự duyệt: A B D



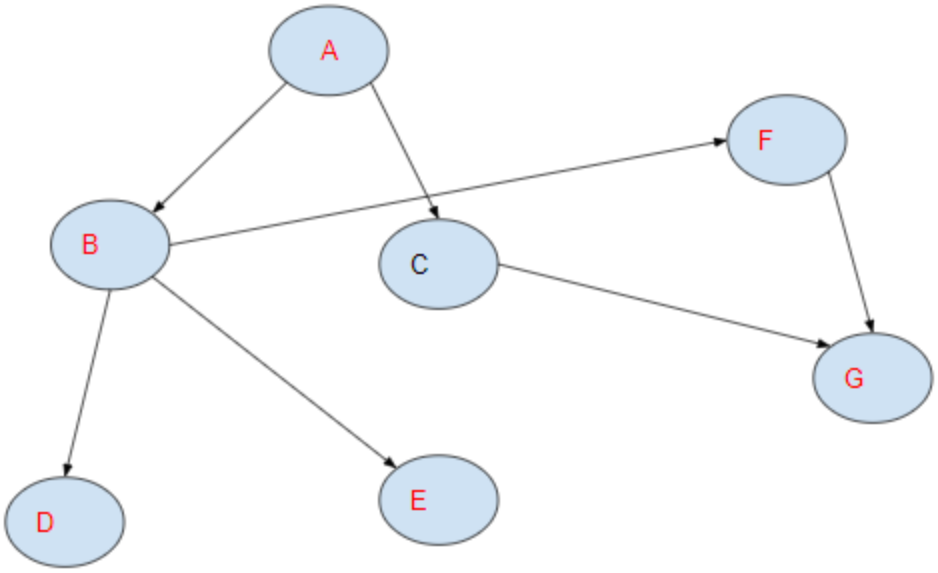
Cur: E

Thứ tự duyệt: A B D E



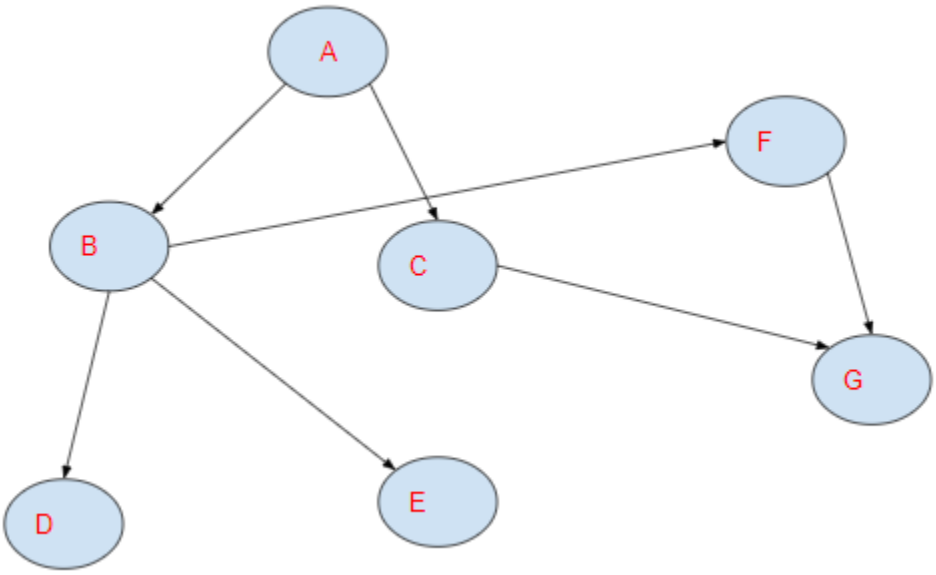
Cur: F

Thứ tự duyệt: A B D E F



Cur: G

Thứ tự duyệt: A B D E F G



Cur: C

Thứ tự duyệt: A B D E F G C

Đã duyệt hết các đỉnh nên không có đường đi nữa => kết quả thứ tự duyệt : A B D E F G C

Bài 1.10: Giải thuật tìm đường đi ngắn nhất dijkstra

Là thuật toán tham lam tìm đi ngắn nhất từ một đỉnh đến mọi đỉnh bất kì trong đồ thị

Thứ tự thực hiện:

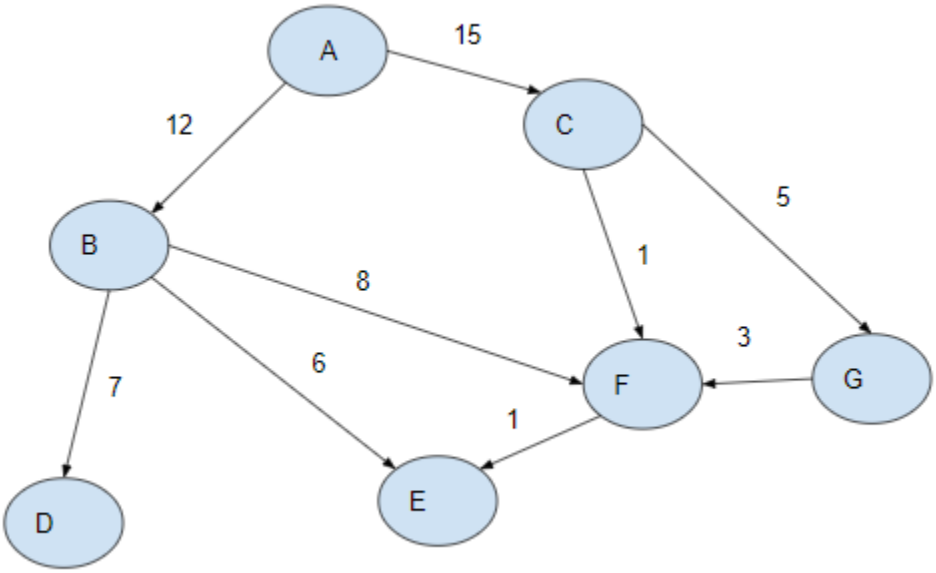
+) Khởi tạo tập N' chứa các đỉnh đã tìm được đường đi ngắn nhất đến đỉnh xuất phát

+) Tìm các đỉnh kề không nằm trong N' và có độ dài đường đi nhỏ nhất

+) Cập nhật đỉnh vừa tìm được vào N', cập nhật D[v] chứa độ dài nhỏ nhất từ đỉnh đầu tiên đến v, cập nhật p[v] là đỉnh trước đỉnh v có đường đi đến v và là đường đi ngắn nhất

Độ phức tạp: O((V^2+E)

Ví dụ:



Bước	N'	D(A)	D(B)	D(C)	D(D)	D(E)	D(F)	D(G)
1	A	0	12	15	∞	∞	∞	∞
2	AB	0	12	15	19	18	20	∞
3	ABC	0	12	15	19	18	16	20

4	ABCD	0	12	15	19	18	16	20
5	ABCDE	0	12	15	19	18	16	20
6	ABCDEF	0	12	15	19	17	16	20
7	ABCDEFG	0	12	15	19	17	16	20

Ta thêm A vào N' và cập nhật chi phí tới các đỉnh kề với A là B và C. Đỉnh B kề với A và không thuộc N', ta cập nhật B vào N' và cập nhật chi phí mới cho các đỉnh kề với B(chi phí mới tới các đỉnh kề sẽ hoặc là chi phí cũ tới các đỉnh kề đó hoặc là chi phí đường đi ngắn nhất tới B cộng với chi phí từ B tới các đỉnh kề; chi phí mới phải nhỏ hơn hoặc bằng chi phí mới). Ta lặp lại như với đỉnh B cho đến khi tất cả các đỉnh được thêm vào tập N'.

Bài 1.11: Giải thuật tìm đường đi ngắn nhất Bellman-Ford

Cho một đồ thị và từ một đỉnh nguồn ở trong đồ thị, thuật toán Bellman-ford sẽ tìm đường ngắn nhất từ nguồn đến tất cả các đỉnh trong đồ thị đã cho. Nghe có vẻ giống với dijkstra tuy nhiên dijkstra đòi hỏi trọng số với giá trị không âm

Độ phức tạp của Bellman-ford: O(V*E) nhanh hơn so với dijkstra đơn thuần.

Thứ tự thực hiện

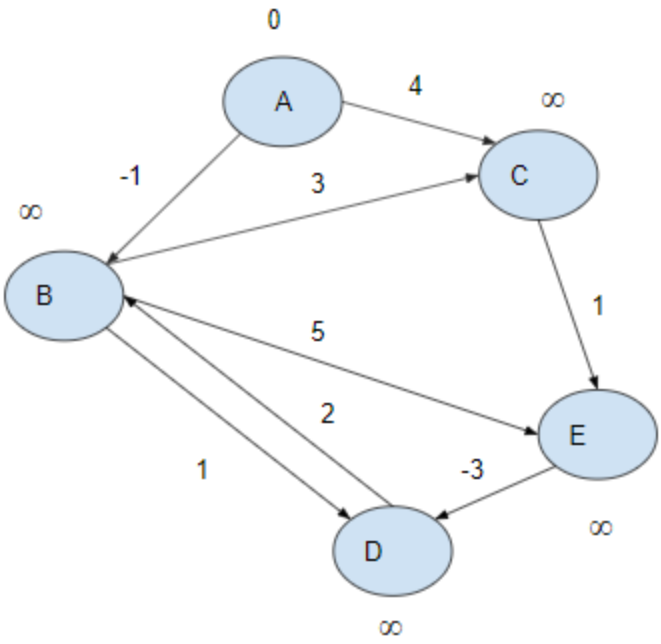
+) Ta thực hiện duyệt n lần, với n là số đỉnh của đồ thị.

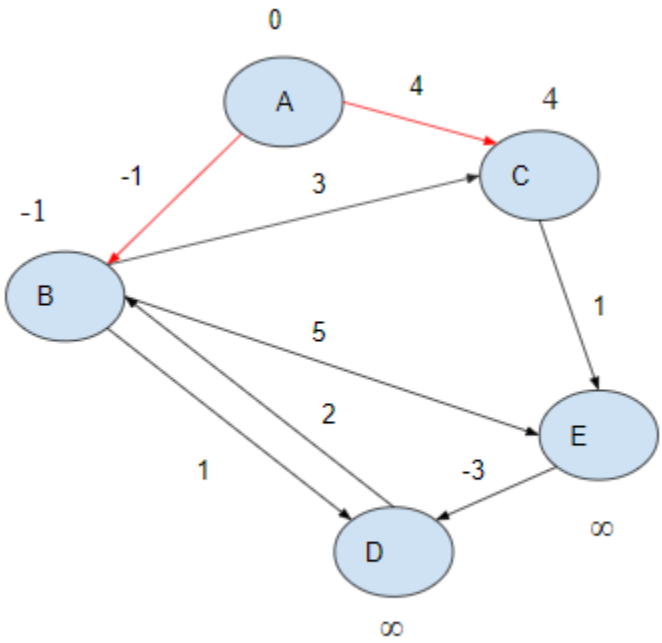
+) Với mỗi lần duyệt, ta tìm tất cả các cạnh mà đường đi qua cạnh đó sẽ rút ngắn đường đi ngắn nhất từ đỉnh gốc tới một đỉnh khác.

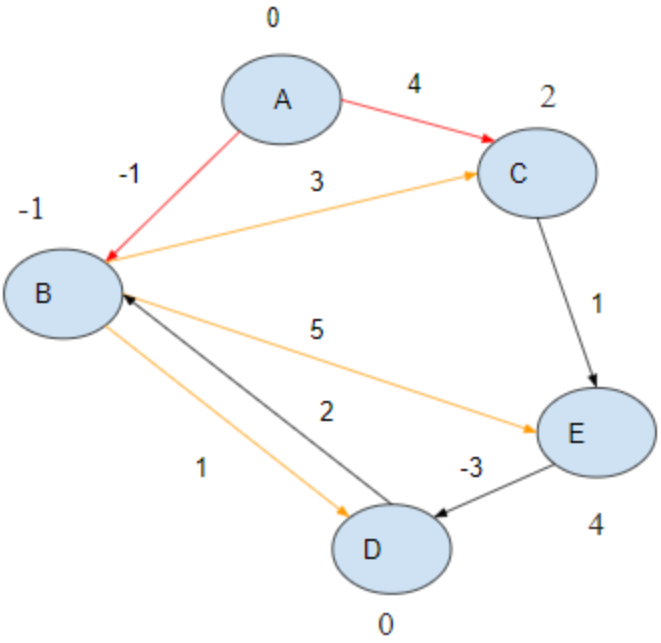
+) Ở lần duyệt thứ n, nếu còn bất kỳ cạnh nào có thể rút ngắn đường đi, điều đó chứng tỏ đồ thị có chu trình âm, và ta kết thúc thuật toán.

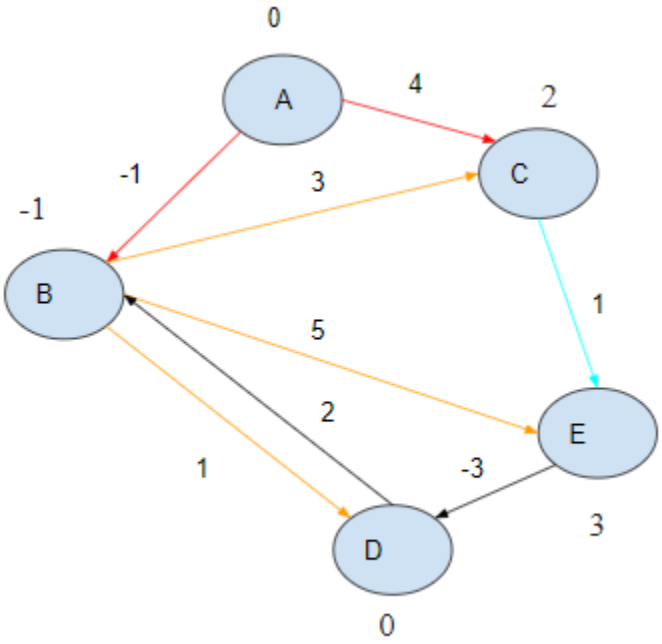
Ví dụ:

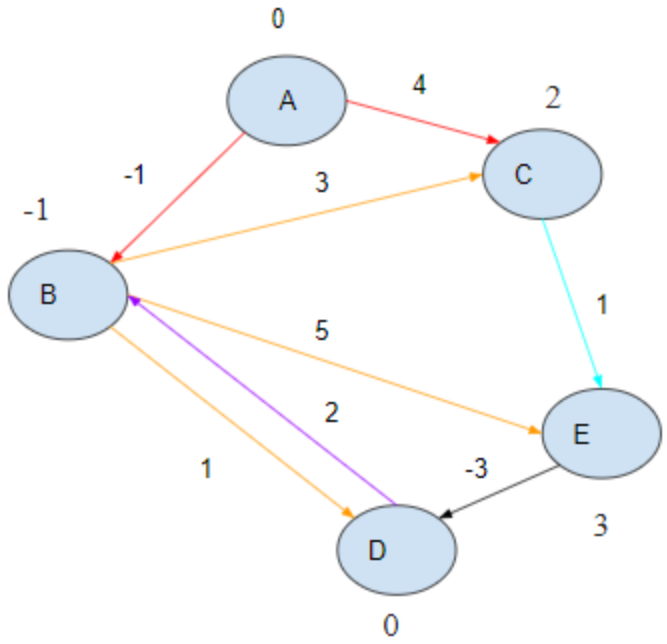
Thứ tự duyệt b các cạnh:(A, B); (A, C); (B, D);(B, E); (B, C); (C, E); (D, B); (E, D)

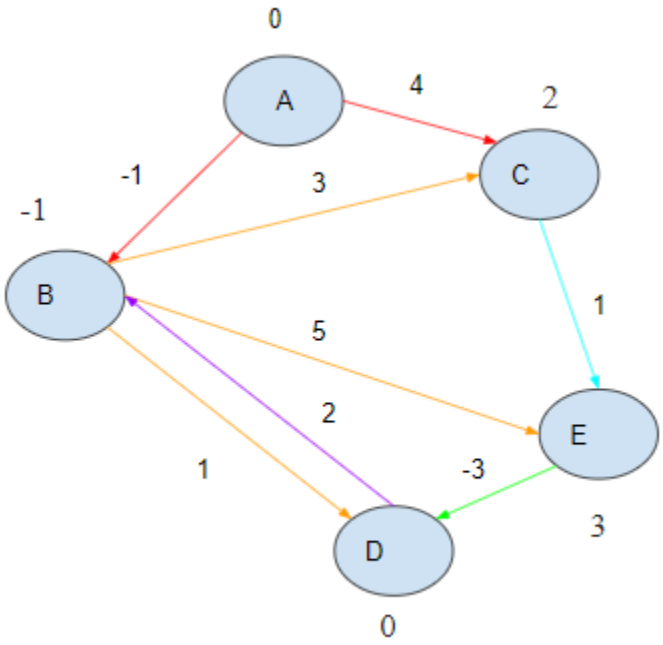

0
A
4
∞
-1
3
C
∞
B
1
5
2
E
1
-3
∞
D
∞


0
A
4
4
-1
3
C
-1
B
1
5
2
E
1
-3
∞
D
∞


0
A
4
2
C
-1
3
-1
B
1
5
2
E
1
-3
4
D
0


0
A
4
2
C
-1
3
-1
B
1
5
2
E
1
-3
3
D
0


0
A
4
2
C
-1
3
-1
B
1
5
2
E
1
-3
3
D
0


0
A
4
2
C
-1
3
-1
B
1
5
2
E
1
-3
3
D
0

	(A,B)	(A, C)	(B, D)	(B, E)	(B, C)	(C, E)	(D, B)	(E, D)
D[B]	-1	-1	-1	-1	-1	-1	-1	-1
D[C]	∞	4	2	2	2	2	2	2
D[D]	∞	∞	0	0	0	0	0	0
D[E]	∞	∞	4	4	4	3	3	3

BÀI 2:

Biểu diễn những thuật toán sắp xếp căn bản bằng phương pháp visualization

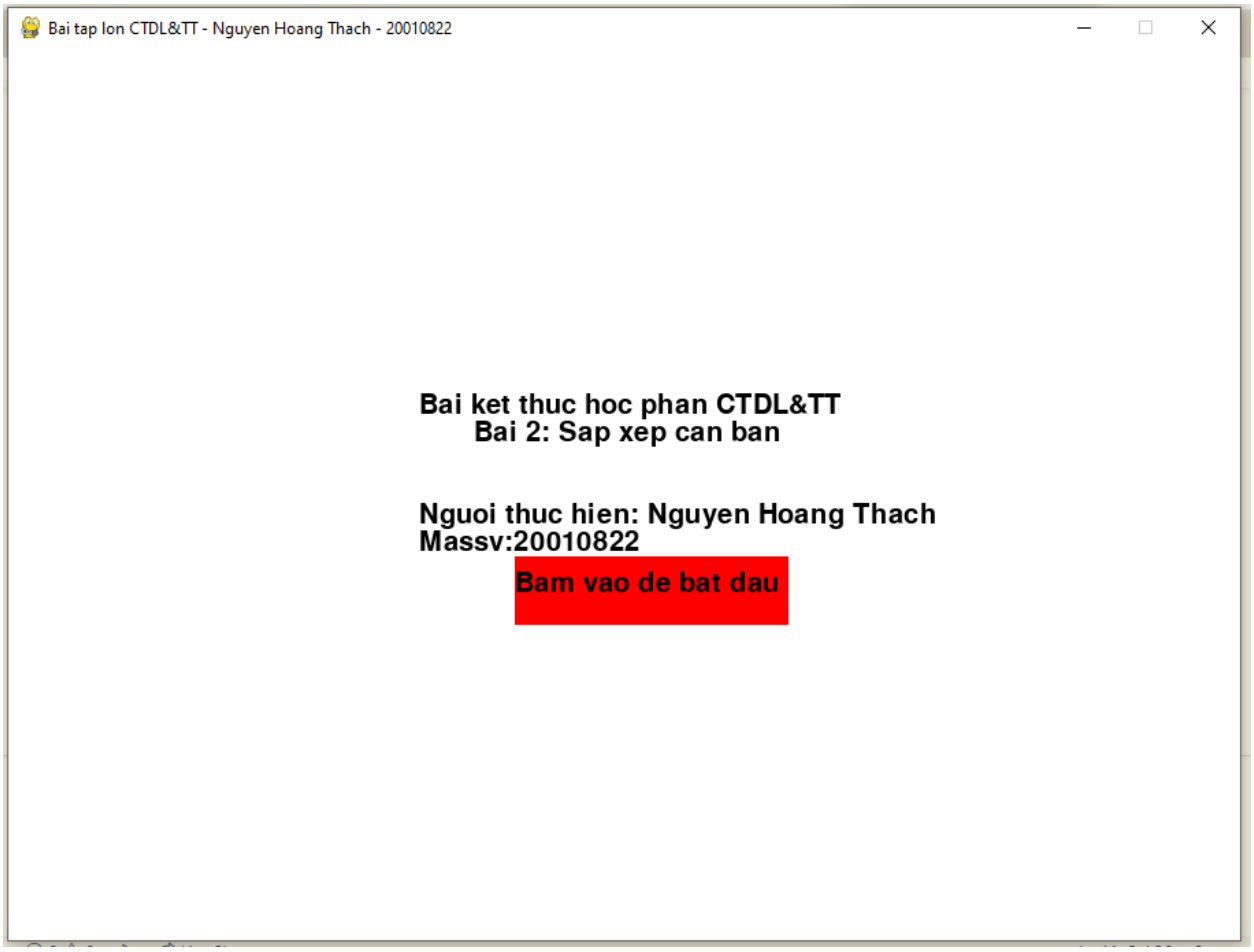
1. **Giới thiệu**

Visualization là phương pháp hình dung và mô phỏng lại những giải thuật sắp xếp một cách rõ ràng thông qua biểu đồ.

Phần mềm được code bằng Visual studio code bằng cách sử dụng thư viện pygame trong ngôn ngữ lập trình python.

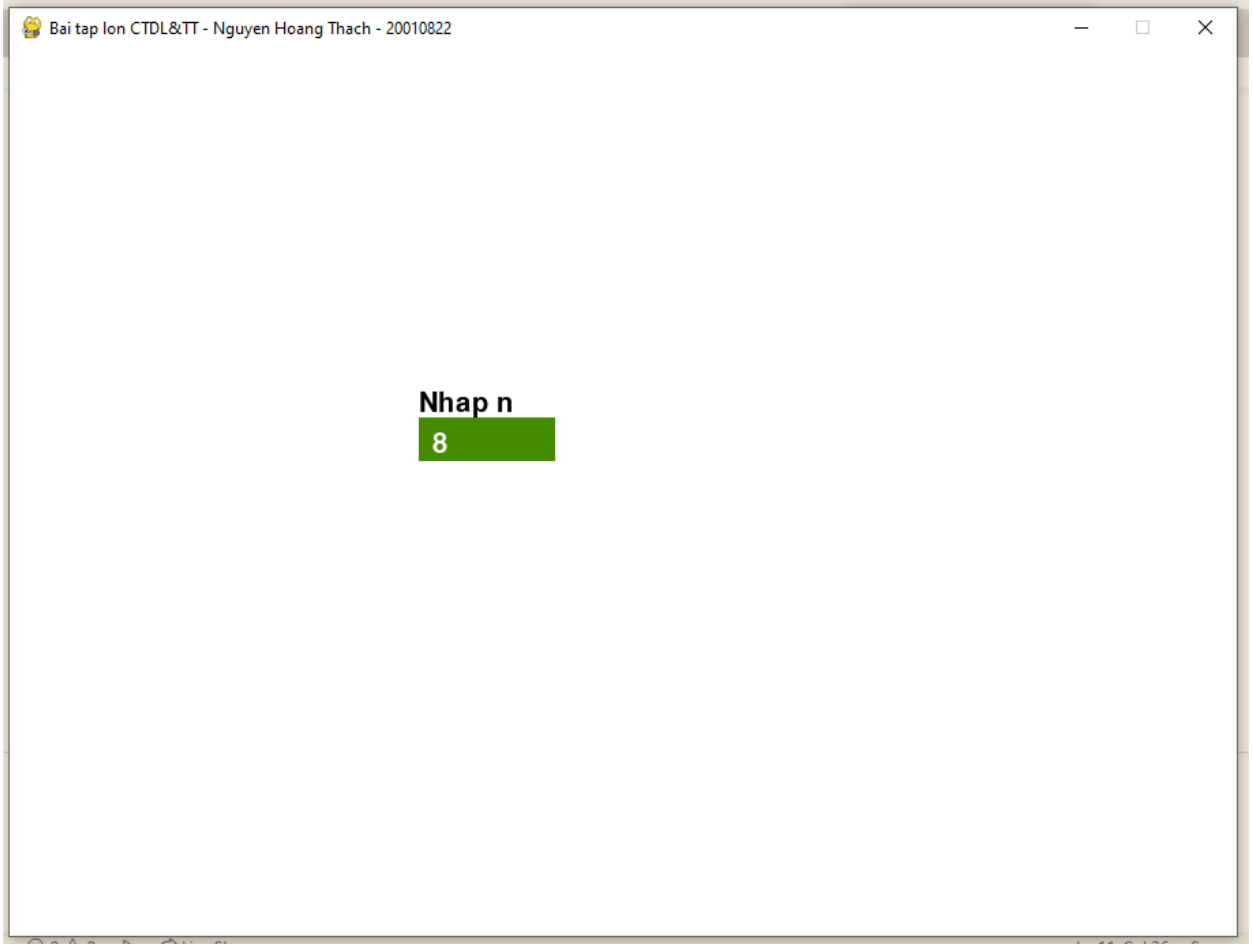
2. **Giao diện**

Phần mềm được cài đặt với resolution 900*650



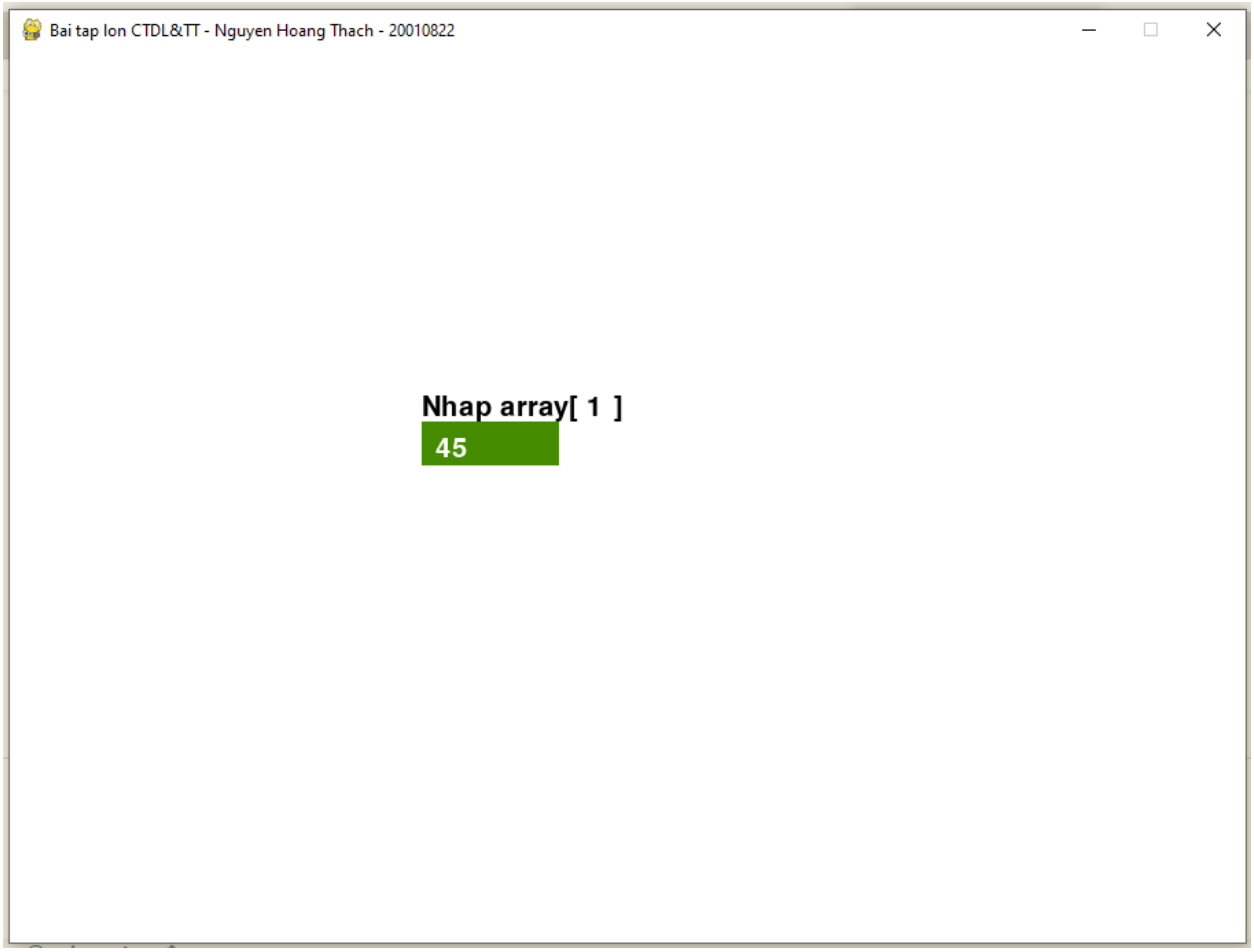
1) Menu chính

Khi bấm vào nút màu đỏ "Bam vao de bat dau" ta được đưa đến một menu khác, ở menu này ta sẽ tiến hành nhập n biểu thị số lượng phần tử mà dãy cần sắp xếp sẽ có



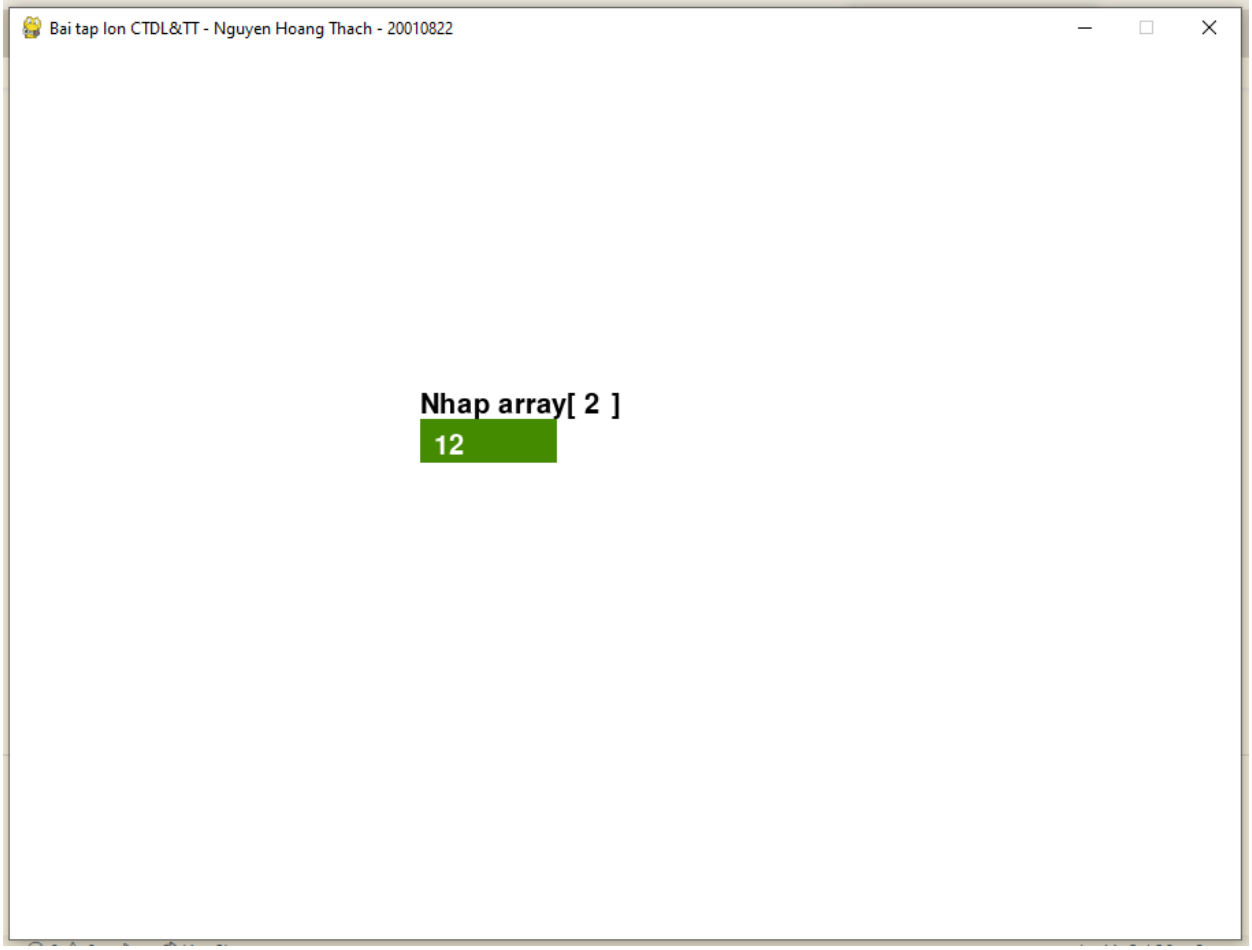
2) Nhập n

Bấm enter sau khi nhập n với n ở hình số 2 là 8 ta chuyển đến nhập phần tử đầu tiên



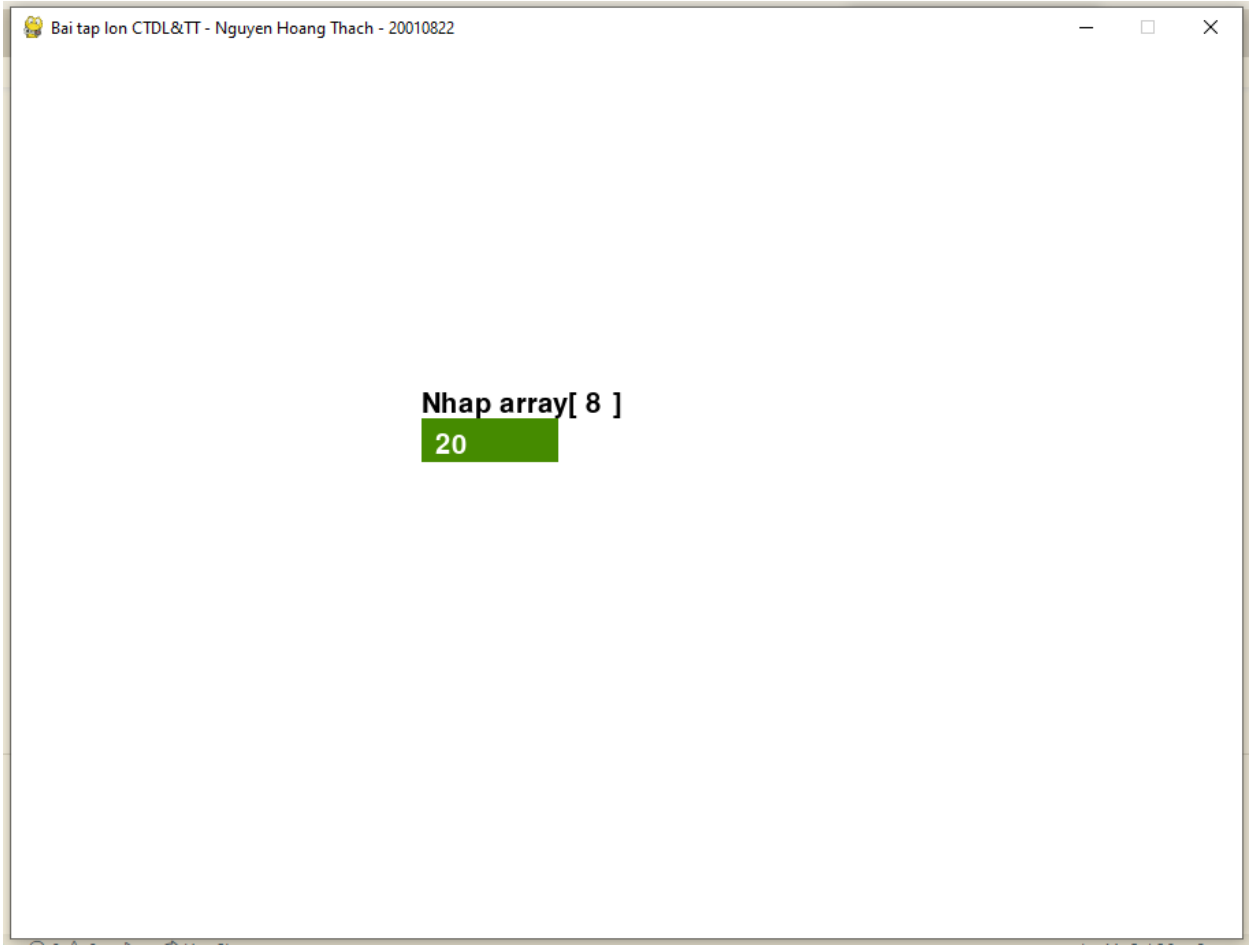
3) Nhập phần tử thứ nhất của dãy

Với ví dụ ở trên ta nhập array[1] =45 sau khi enter phần mềm sẽ chuyển sang nhập giá trị tiếp theo đó là array[2]



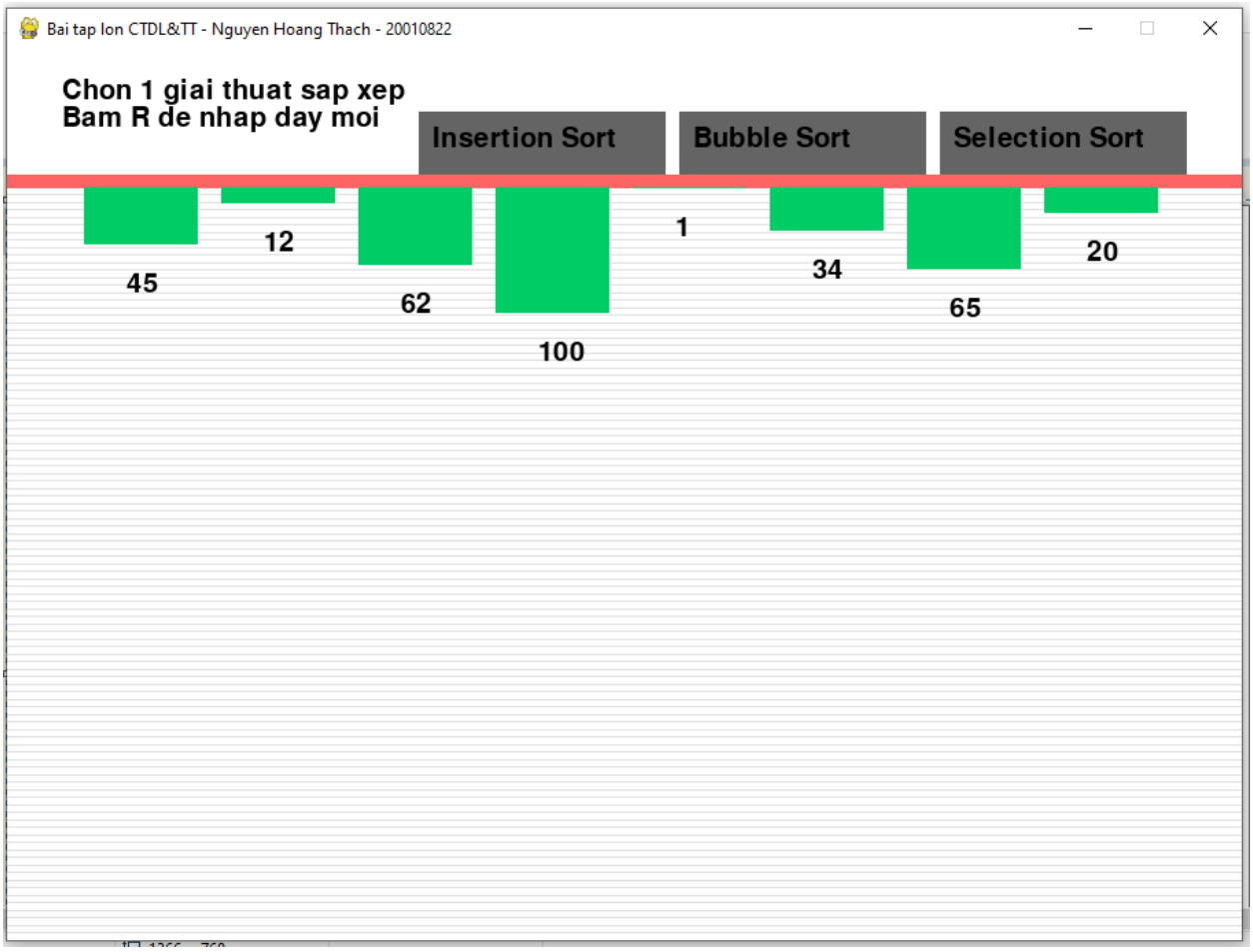
4) Nhập phần tử thứ 2 của dãy

Cứ tiếp tục nhập như vậy cho đến khi nhập phần tử cuối cùng của dãy vì n ta nhập với giá trị 8 nên phần tử cuối cùng có i=8



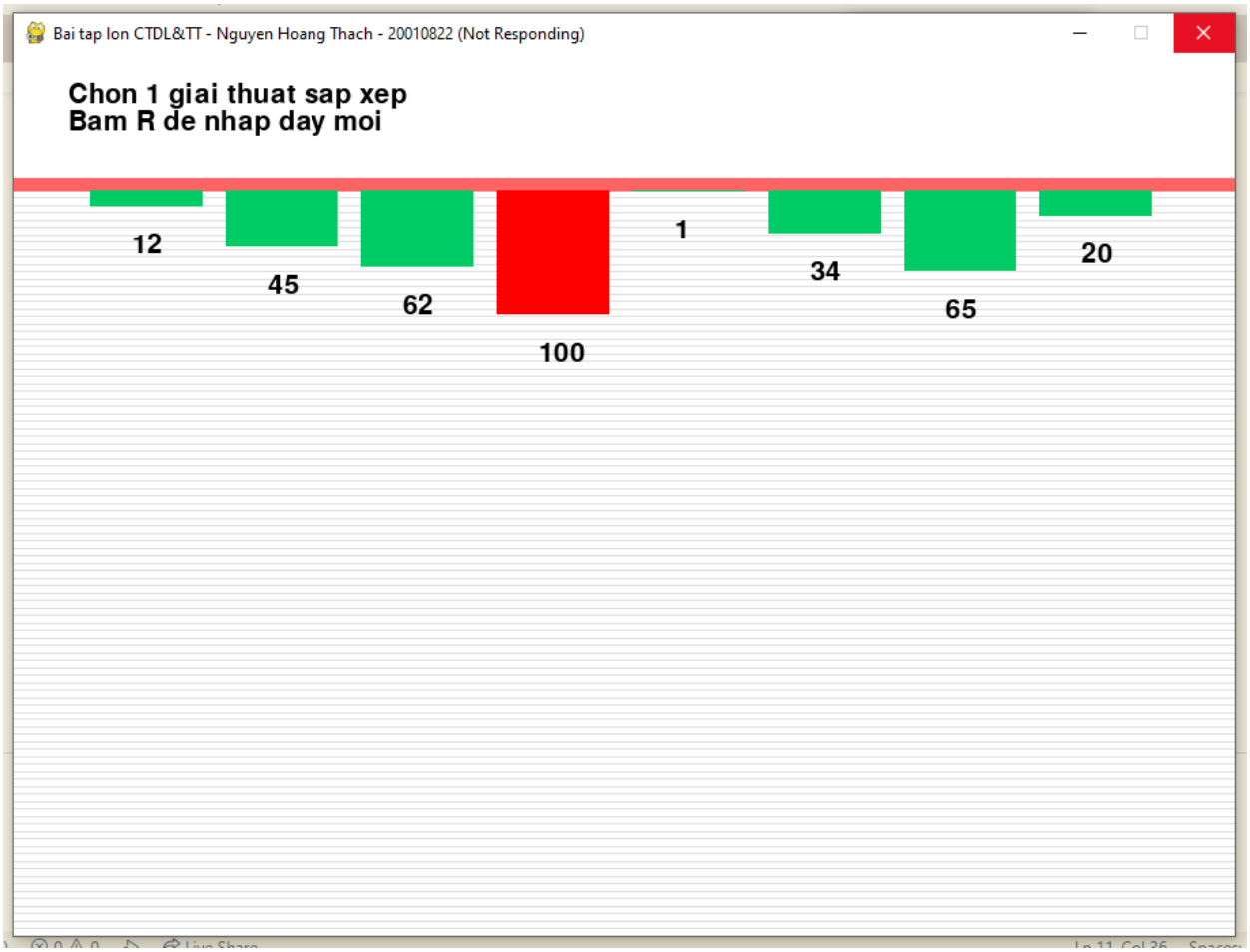
5) Nhập phần tử cuối cùng của dãy

Sau khi nhập xong phần tử cuối cùng ta sẽ được đưa đến một cửa số khác chứa một biểu đồ biểu thị dãy đã nhập và 3 giải thuật sắp xếp cơ bản

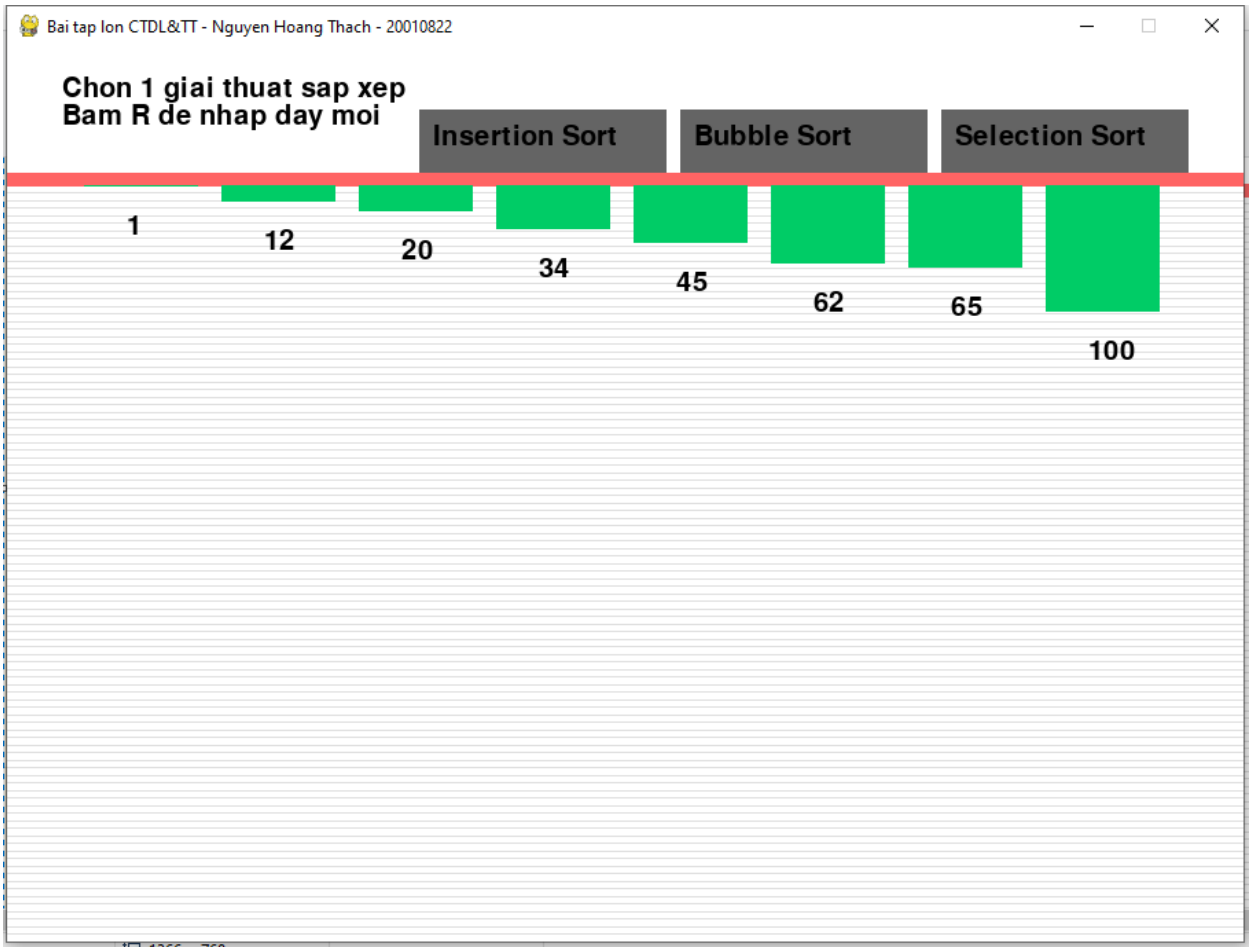


6) Cửa sổ thực hiện mô tả các thuật toán sx căn bản

Ở đây ta có 3 lựa chọn thực hiện visualize thuật toán Insertion sort, Bubble sort hoặc là Selection Sort ở đây lấy ví dụ khi ta click chuột vào Insertion sort phần mềm sẽ thực hiện mô tả từng bước giải thuật Insertion sort trên biểu đồ bên dưới



7) Khi đang thực hiện Insertion sort



8) Sau khi thực hiện xong Insertion Sort

Nó sẽ thực hiện từng bước từng bước một với tốc độ được cài đặt trong phần code cho đến khi dãy được sắp xếp hoàn chỉnh theo trình tự tăng dần

Có thể bấm nút R để nhập một mảng mới và thực hiện sắp xếp lại dãy mới đó

3. **Mô tả từng giải thuật**

- Insertion sort:

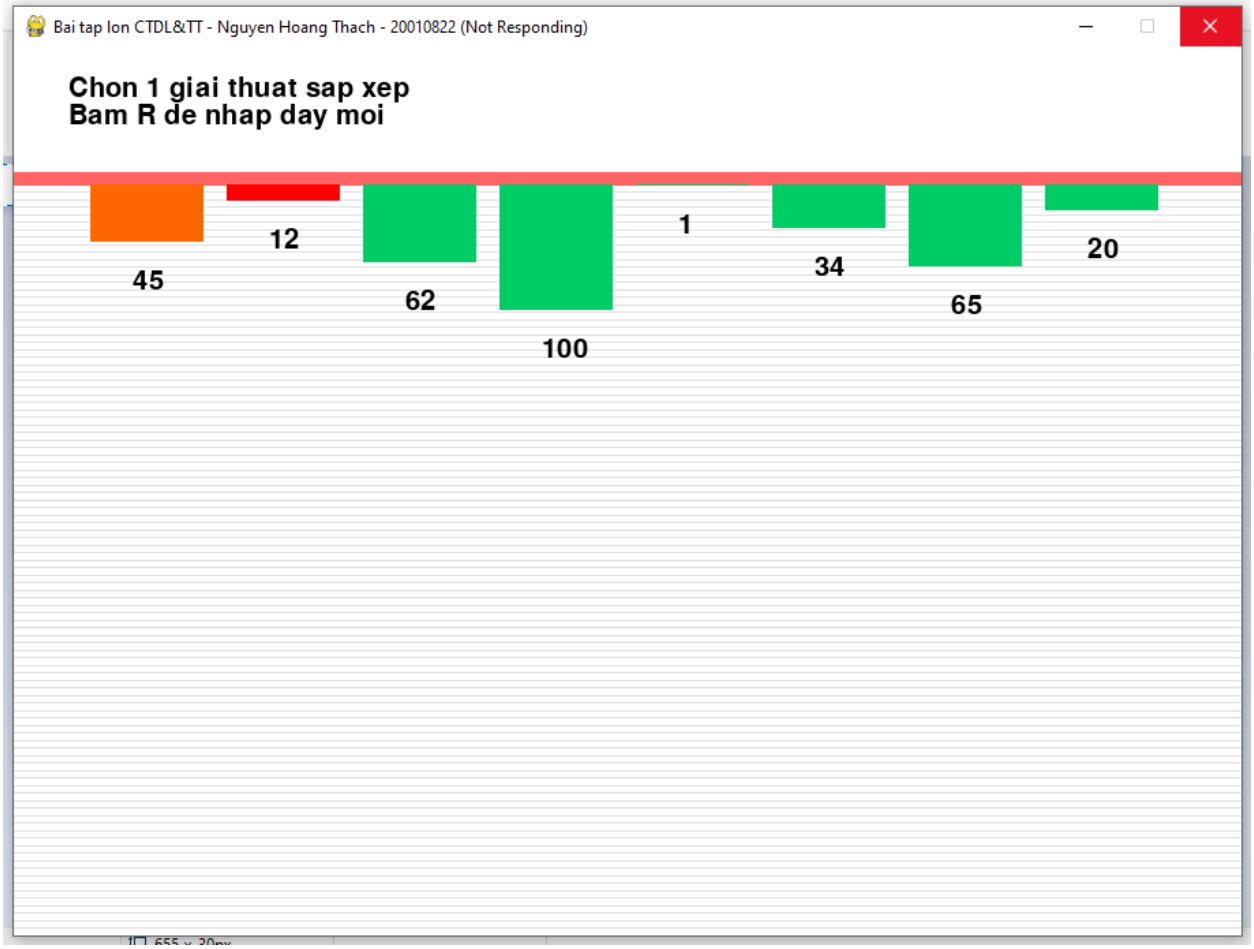
+) Lí thuyết:

Với dãy khóa array[1..n]. Ta sẽ sắp xếp dãy array[1..i] trong điều kiện dãy array[1..i-1] đã sắp xếp rồi bằng cách chèn array[i] vào dãy đó tại vị trí đúng khi sắp xếp. Ta làm điều này qua việc tìm giá trị đầu tiên tìm được có giá trị lớn hơn giá trị của khóa thì bắt đầu di chuyển tất cả những phần tử lớn hơn khóa từ 1-> i-1 lên 1 vị trí để có khoảng trống cho khóa thực hiện hoán đổi, khi đó ta đươc array[1..i] đã sắp xếp.

+) Mô tả lấy ví dụ với Visualize:

Lấy ví dụ như trên: n=8

array[1]= 45 array[2]=12 array[3]=62 array[4]=100 array[5]=1 array[6]=34 array[7]=65 array[8]=20

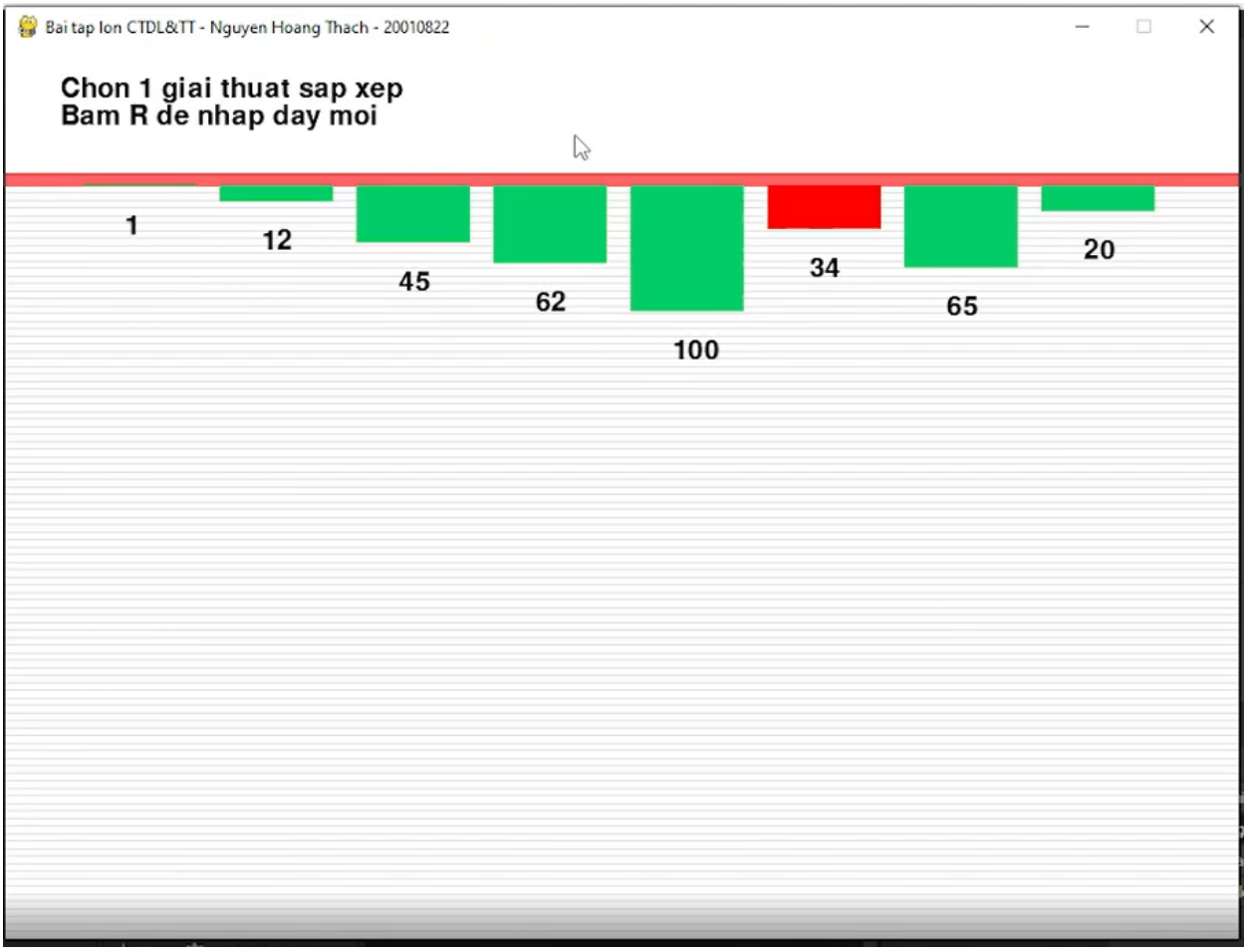


8) Mô tả Insertion sort (1)

Vì giải thuật thực hiện xét i từ trái qua phải nên dãy array[1..i-1] chắc chắn là một dãy tăng dần như vậy nhiệm vụ của j ở đây là truy ngược về dãy array[1..i-1] để tìm ra giá trị array[t] để làm sao khi chèn array[i] vào giữa dãy array[1..t-1] và dãy array[t..i-1] thì sẽ cho ra kết quả là một dãy tăng dần array[1..i]

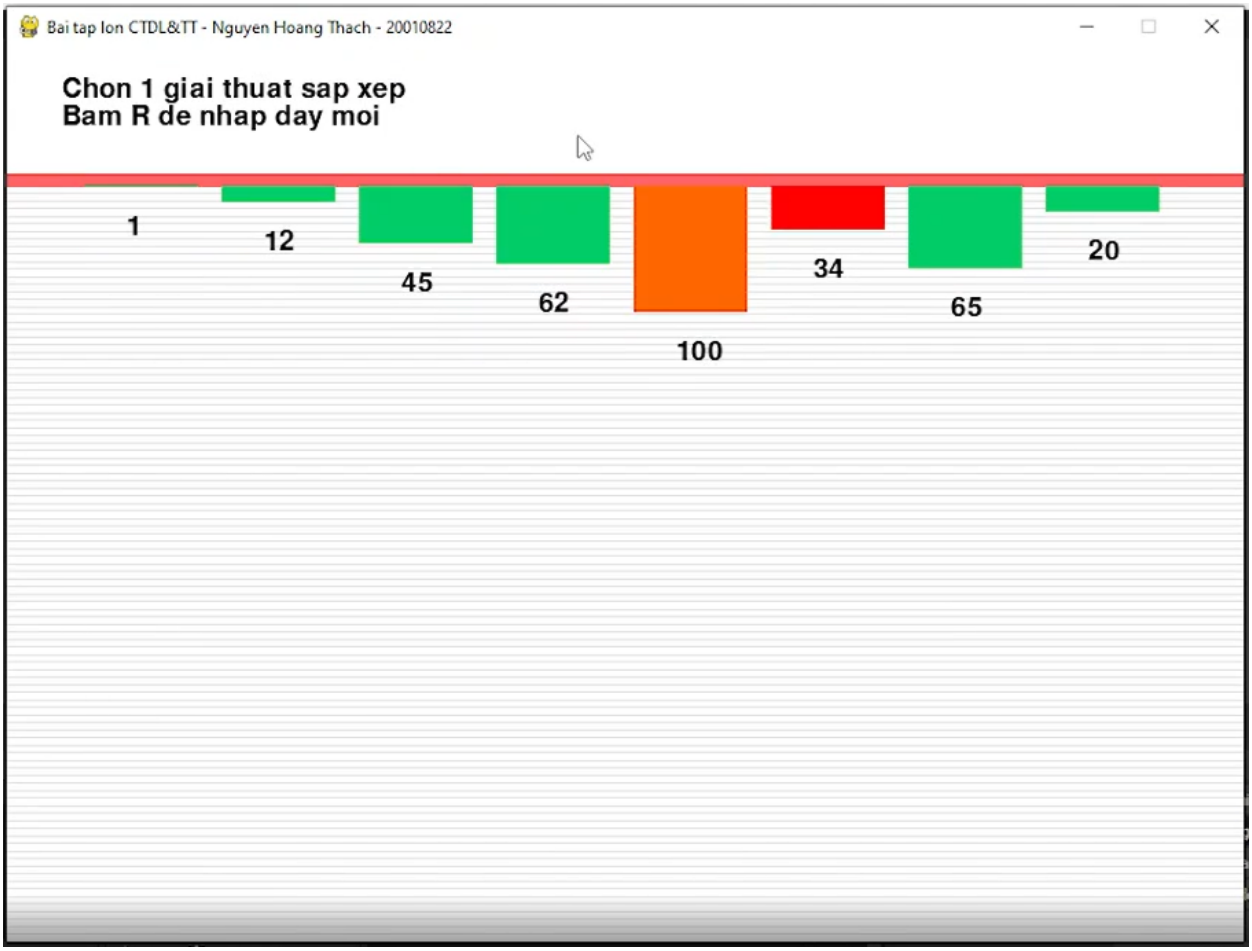
Ở đây phần tử thứ i trong vòng lặp được thể hiện qua màu đỏ, phần tử thứ j được thể hiện qua màu cam và nếu tìm được giá trị t thì array[t] sẽ được biểu thị bằng màu xanh đậm

Với ví dụ trên khi i=6

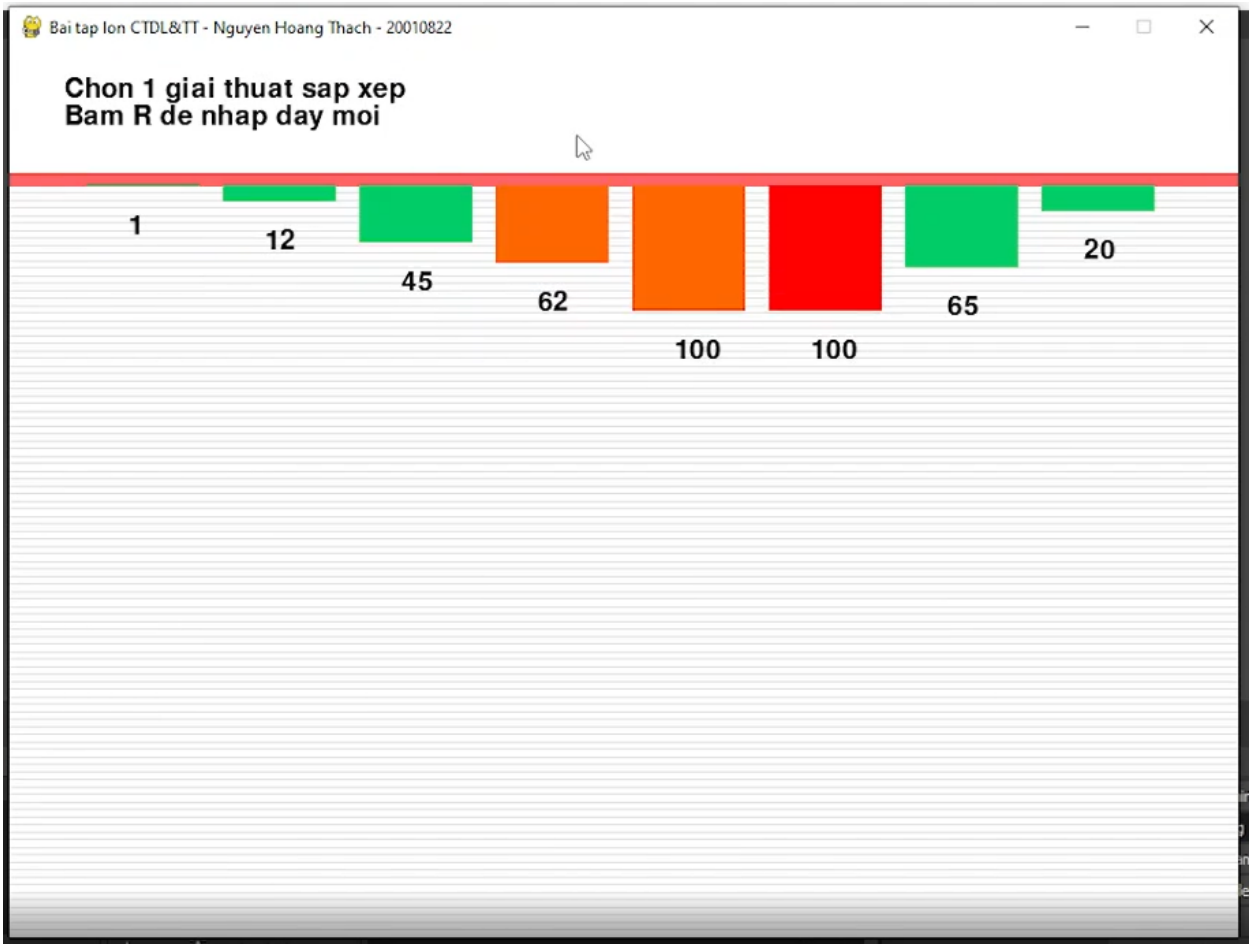


9) Mô tả Insertion sort (2)

Ta thấy dãy array[1..5] đã được sắp xếp, bắt đầu truy ngược j

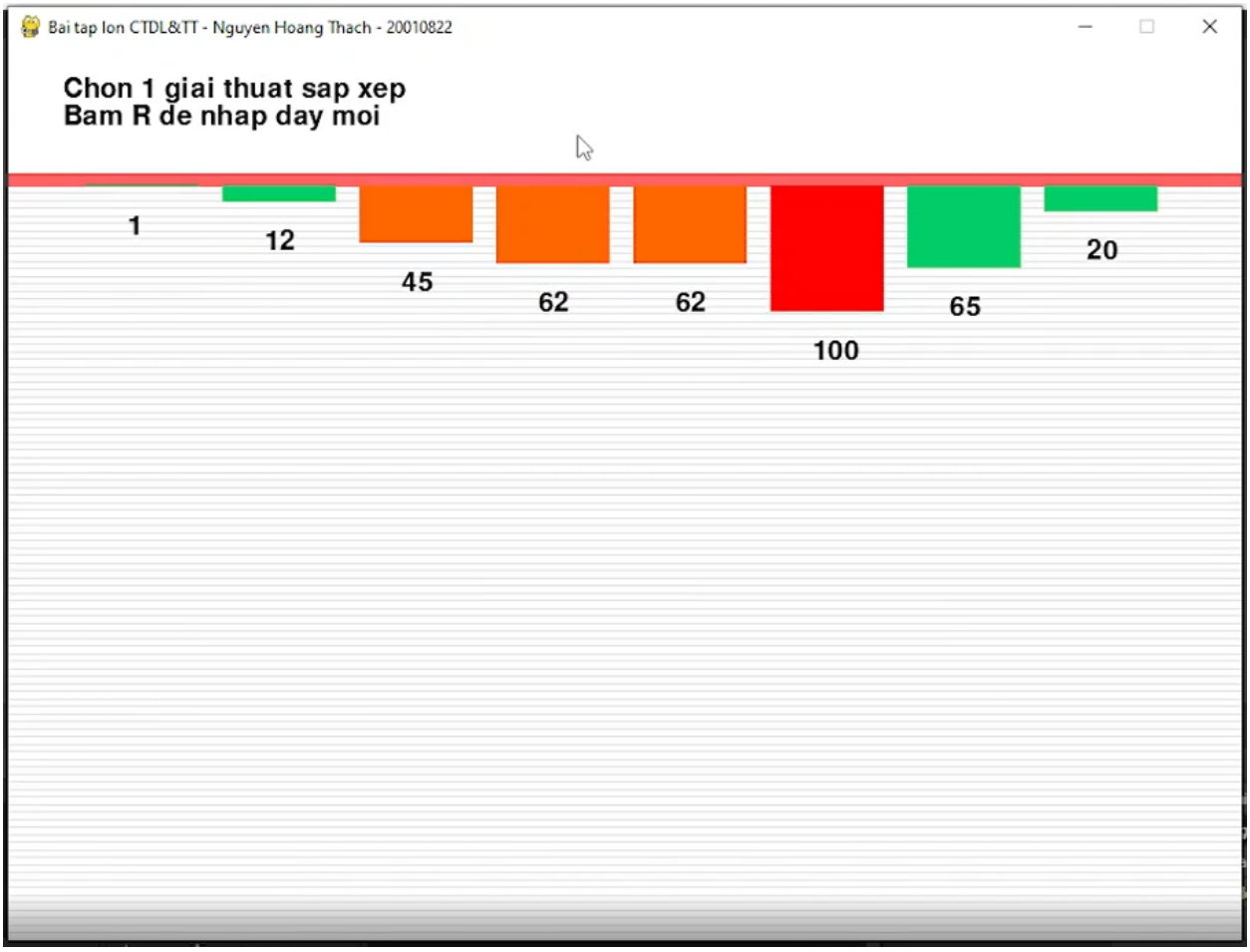


10) Mô tả Insertion sort (3)



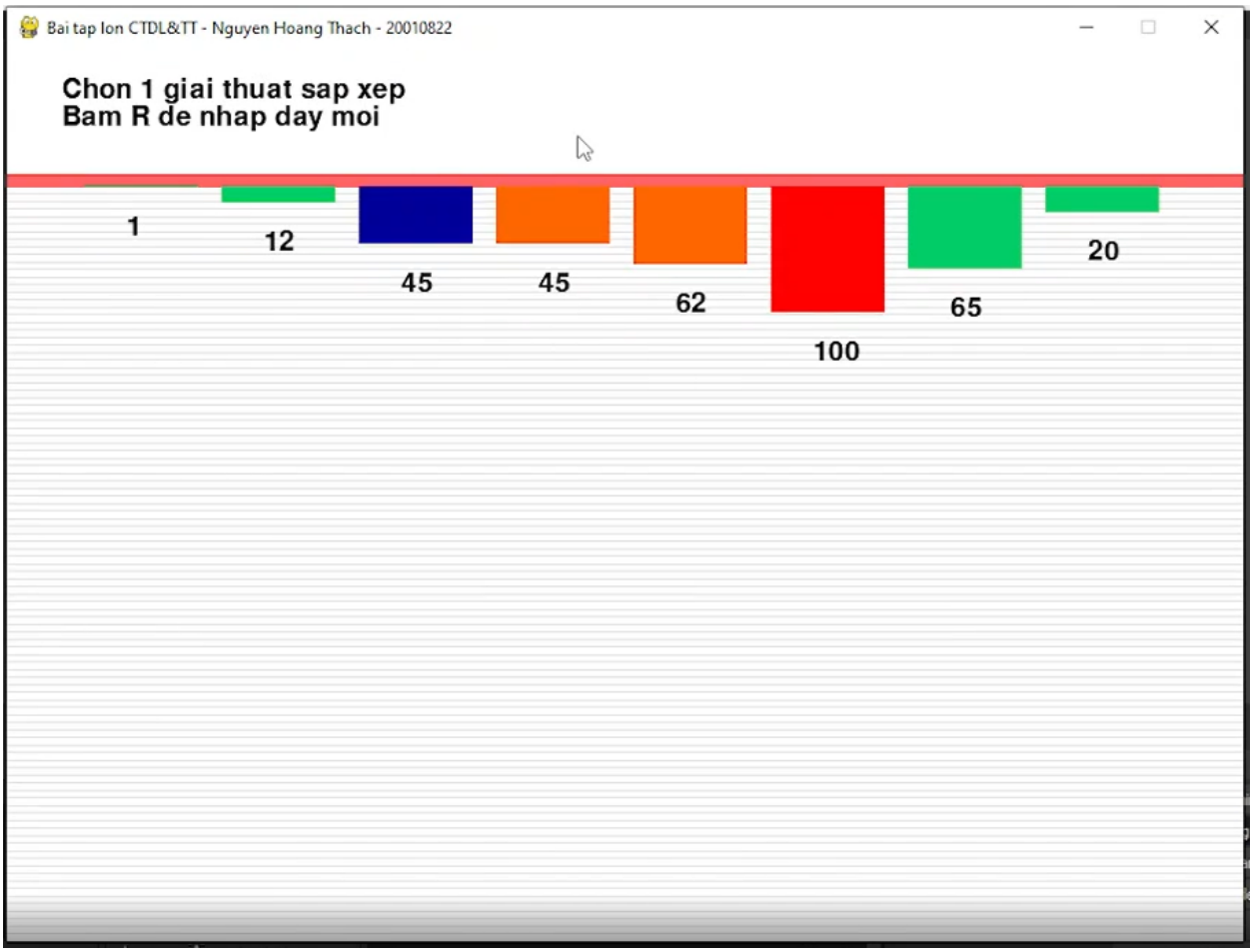
11) Mô tả Insertion sort (4)

Sau mỗi lượt truy ngược sẽ đồng thời lấy giá trị của array[i] ra và đẩy array[i-1] lên thay thế



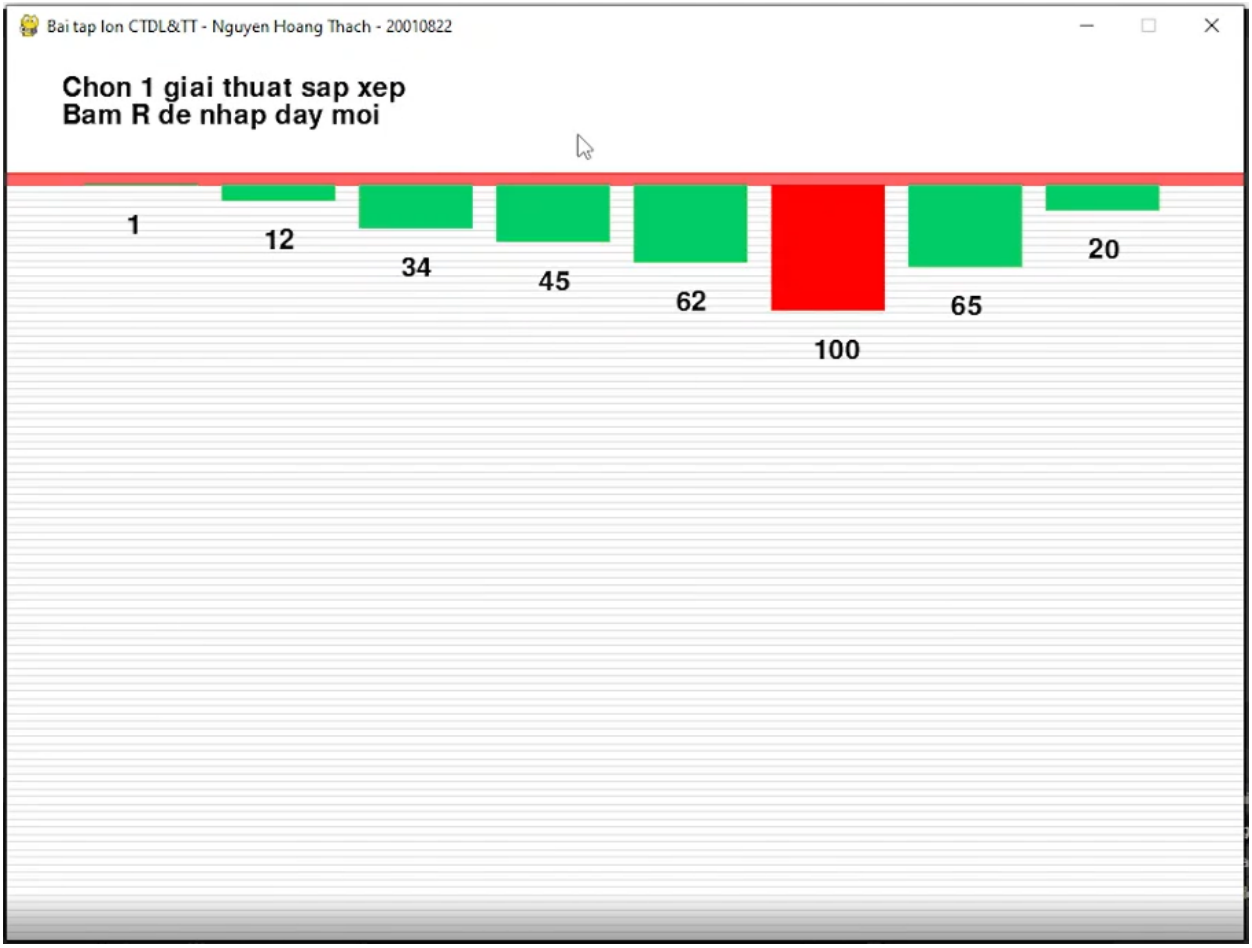
12) Mô tả Insertion sort (5)

Tương tự đẩy array[4] lên vị trí thứ 5



13) Mô tả Insertion sort (6)

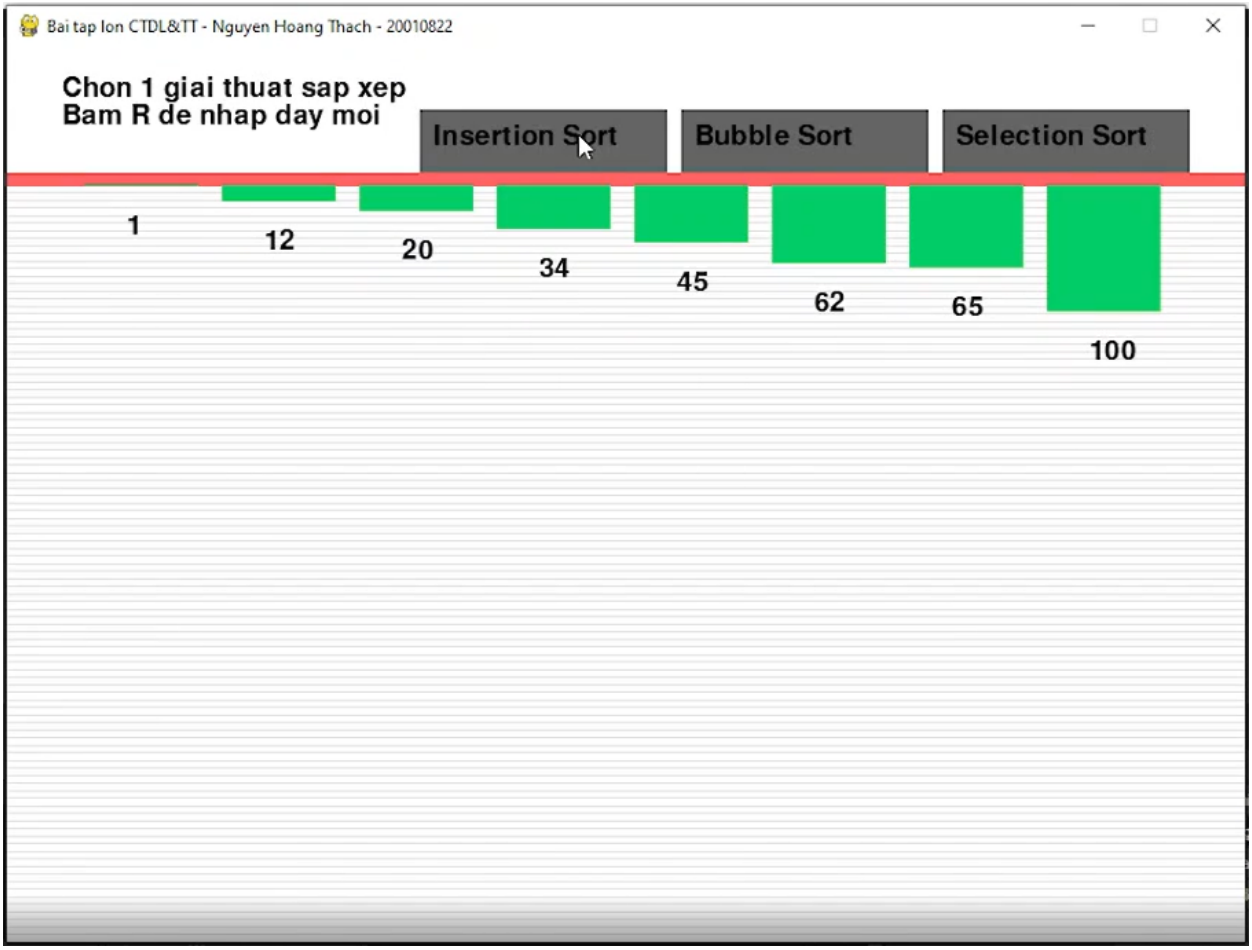
Đẩy array[3] lên 4 và tìm thấy giá trị array[t]



14) Mô tả Insertion sort (7)

Sắp xếp xong dãy array[1..6] và bắt đầu xét tiếp i=7

Xét tương tự như vậy cho đến khi ta được dãy cuối cùng :



15) Mô tả Insertion sort (8)

- Bubble sort:

+) Lí thuyết:

Khi sử dụng thuật toán sắp xếp nổi bọt, dãy các khóa sẽ được duyệt từ cuối dãy lên đầu dãy , nếu gặp hai khóa gần nhau bị ngược thứ tự (Sắp xếp tăng dần nhưng k[i]>k[i+1]) thì đổi chỗ của chúng cho nhau. Hết lần duyệt đầu tiên, khoá nhỏ nhất trong dãy khóa sẽ được chuyển về vị trí đầu tiên (k[1]) và ta tiếp tục thực hiện như trên với dãy k[2..n], rồi tiếp tục với dãy k[3..n]…..

+) Mô tả lấy ví dụ với Visualize:

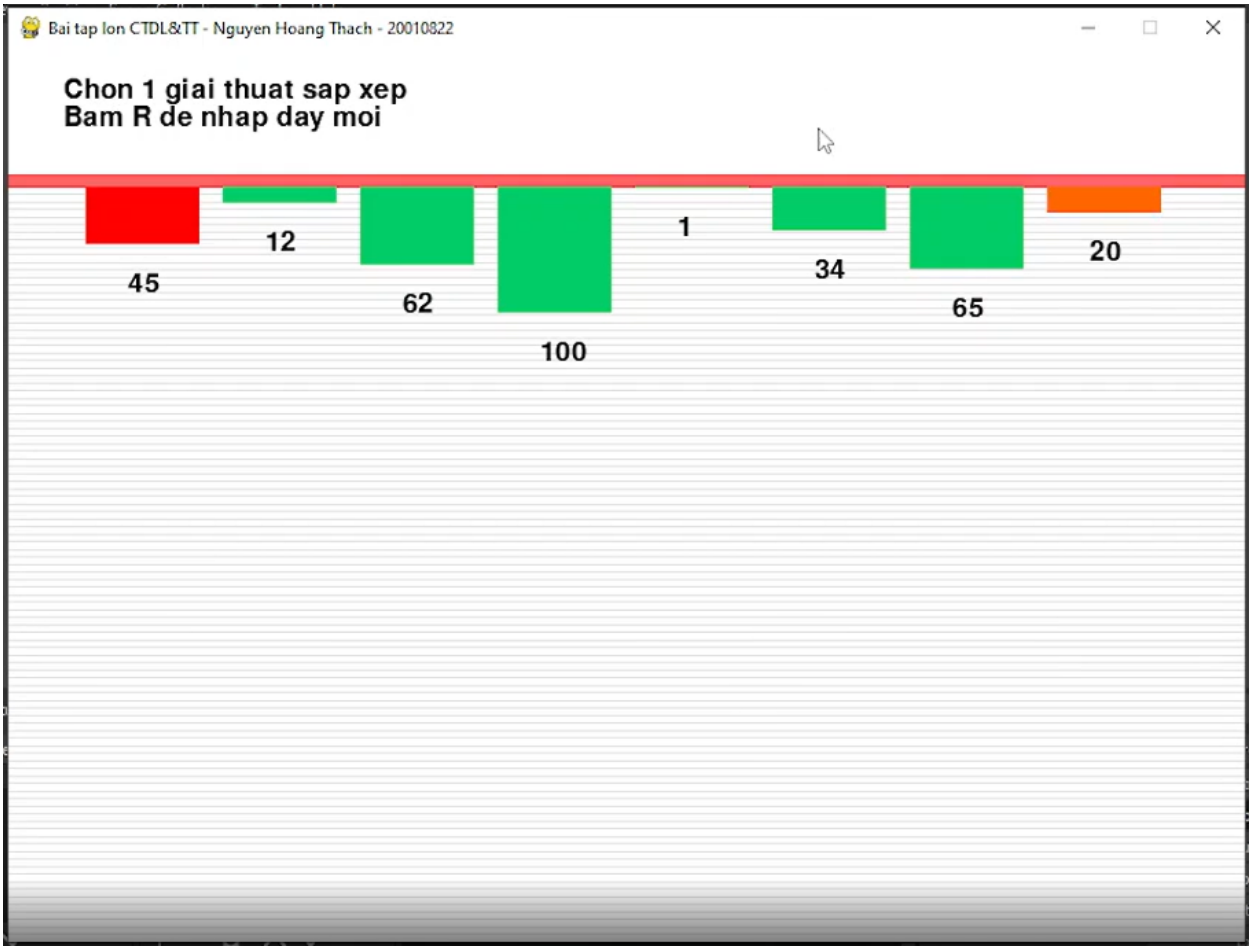
Lấy ví dụ như trên: n=8

array[1]= 45 array[2]=12 array[3]=62 array[4]=100 array[5]=1 array[6]=34 array[7]=65 array[8]=20

j ở đây sẽ truy ngược từ cuối về với dãy array[1..n] nếu phần tử array[j-1] có giá trị lớn hơn array[j] thì ta đổi chỗ

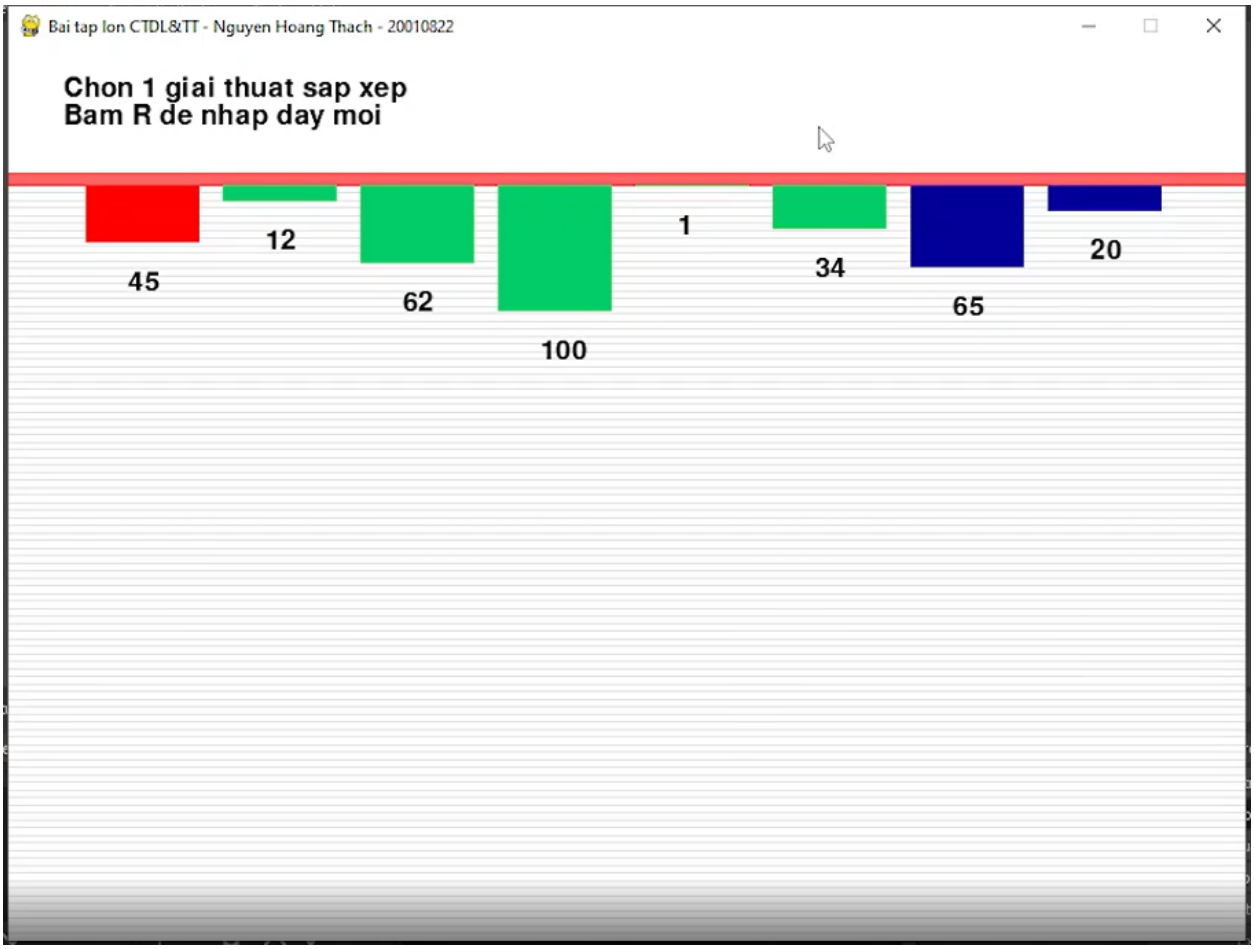
Khi đã xét hết dãy array[1..n] thì chắc chắn phần tử nhỏ nhất của dãy sẽ là array[1] cho nên việc ta cần làm là tiếp tục xét dãy con array[2..n]. Cứ như vậy với array[3..n],…., array[n-1..n]

Ở đây phần tử thứ i trong vòng lặp được thể hiện qua màu đỏ, phần tử thứ j được thể hiện qua màu cam và nếu như array[j]<array[j-1] thì việc đổi chỗ hai phần tử này này sẽ được thể hiện qua màu xanh đậm



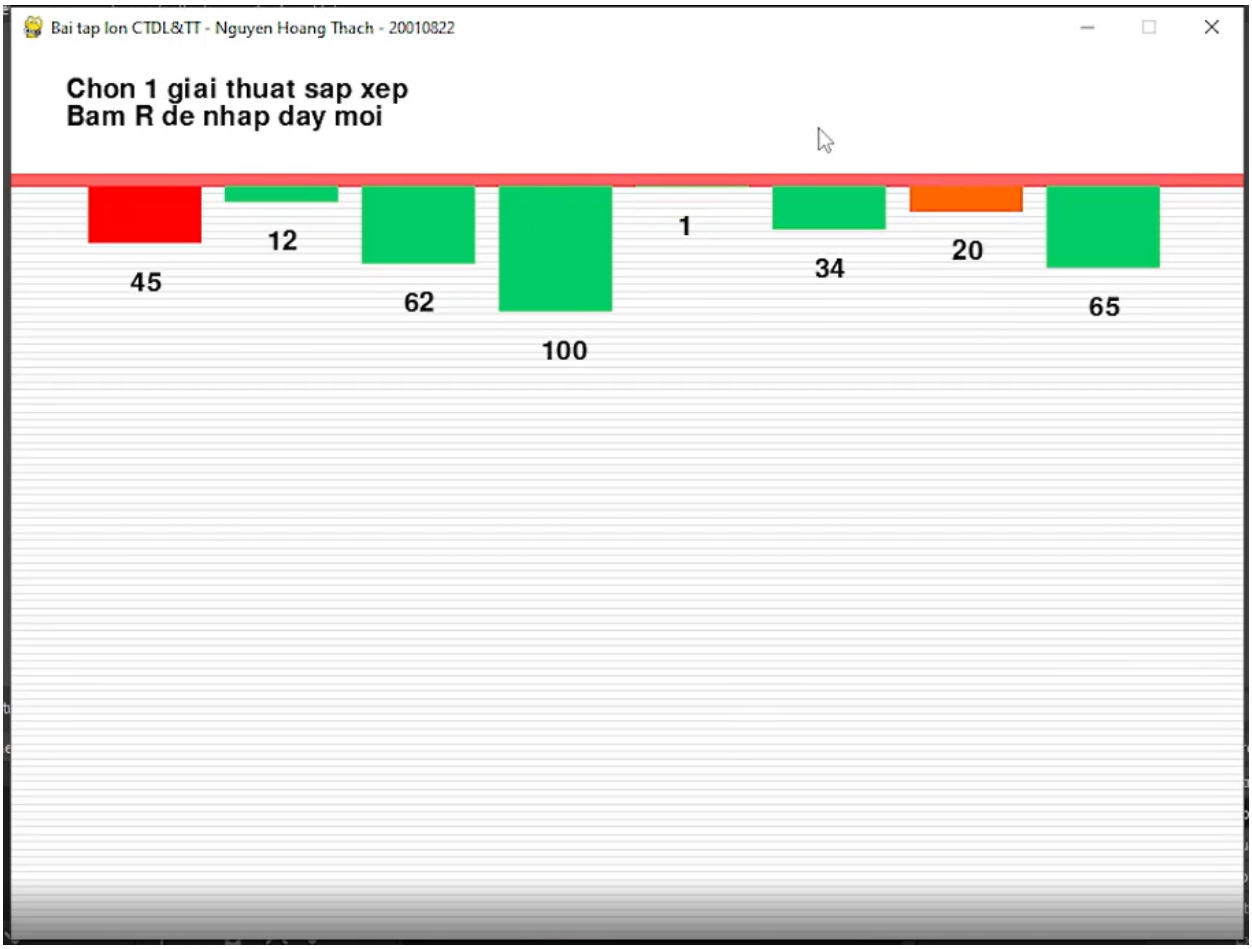
16) Mô tả Bubble sort (1)

Xét i=1, j=8



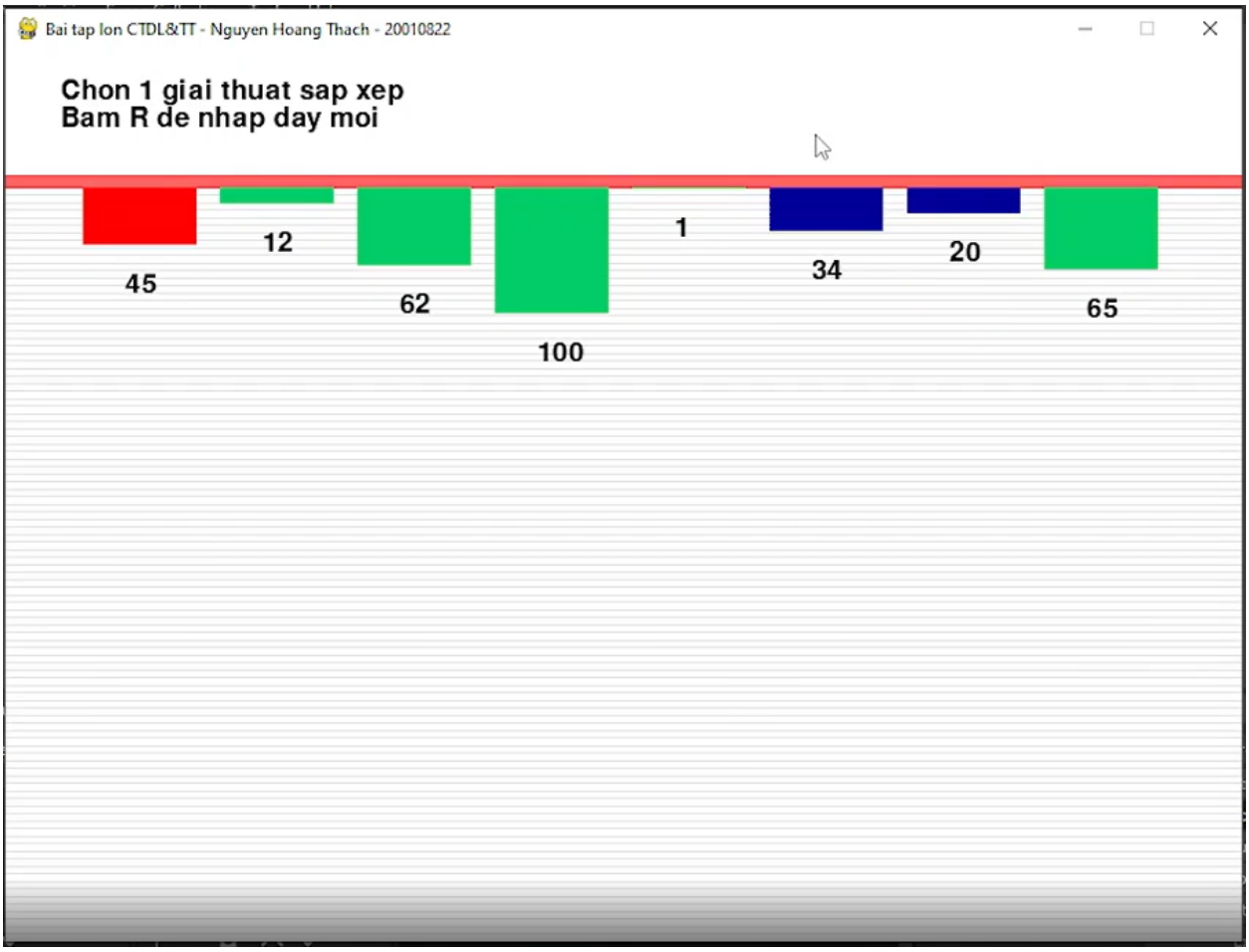
17) Mô tả Bubble sort (2)

Vì array[8]<array[7] nên ta thay đổi hai phần tử



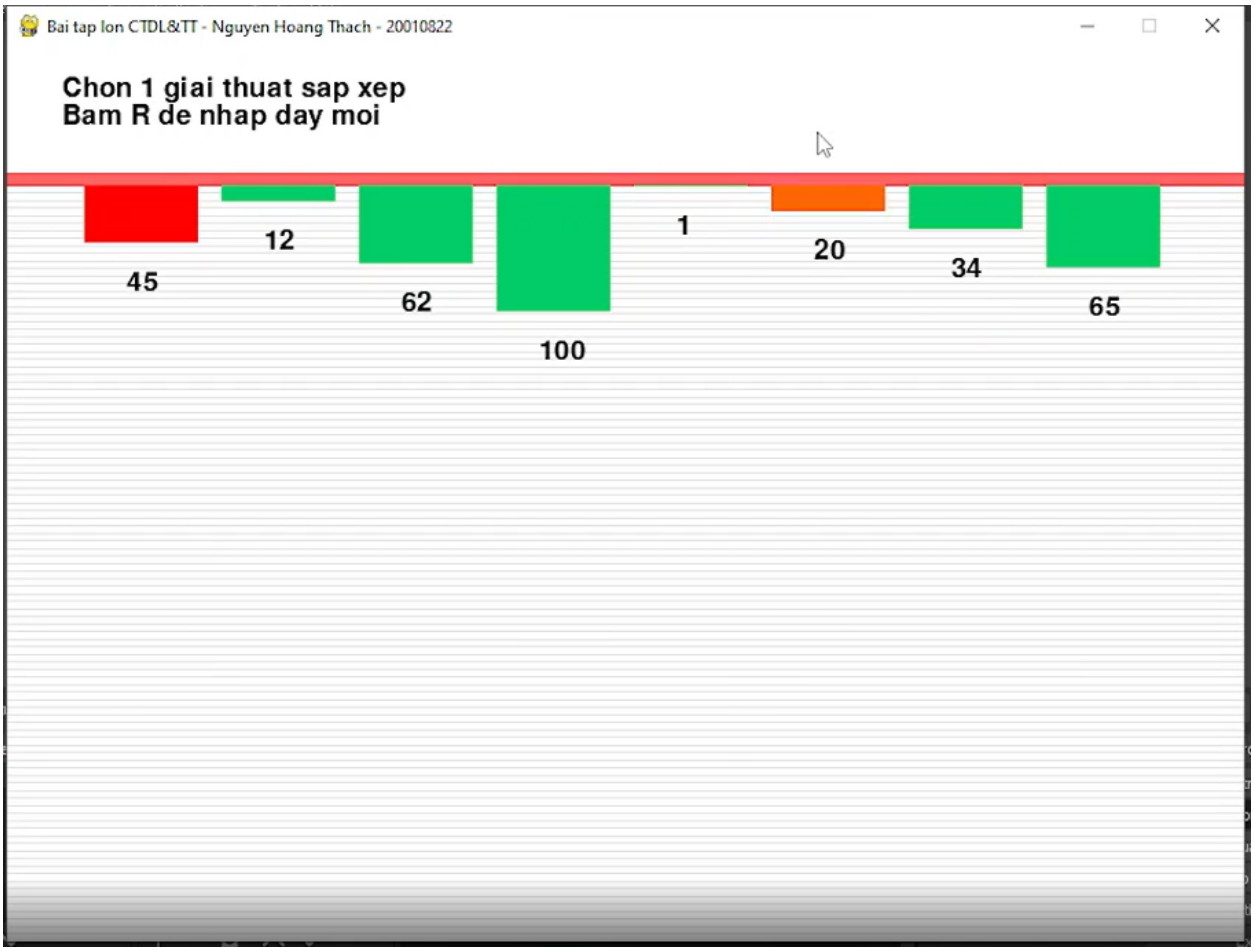
18) Mô tả Bubble sort (3)

Tiếp tục xét j=7



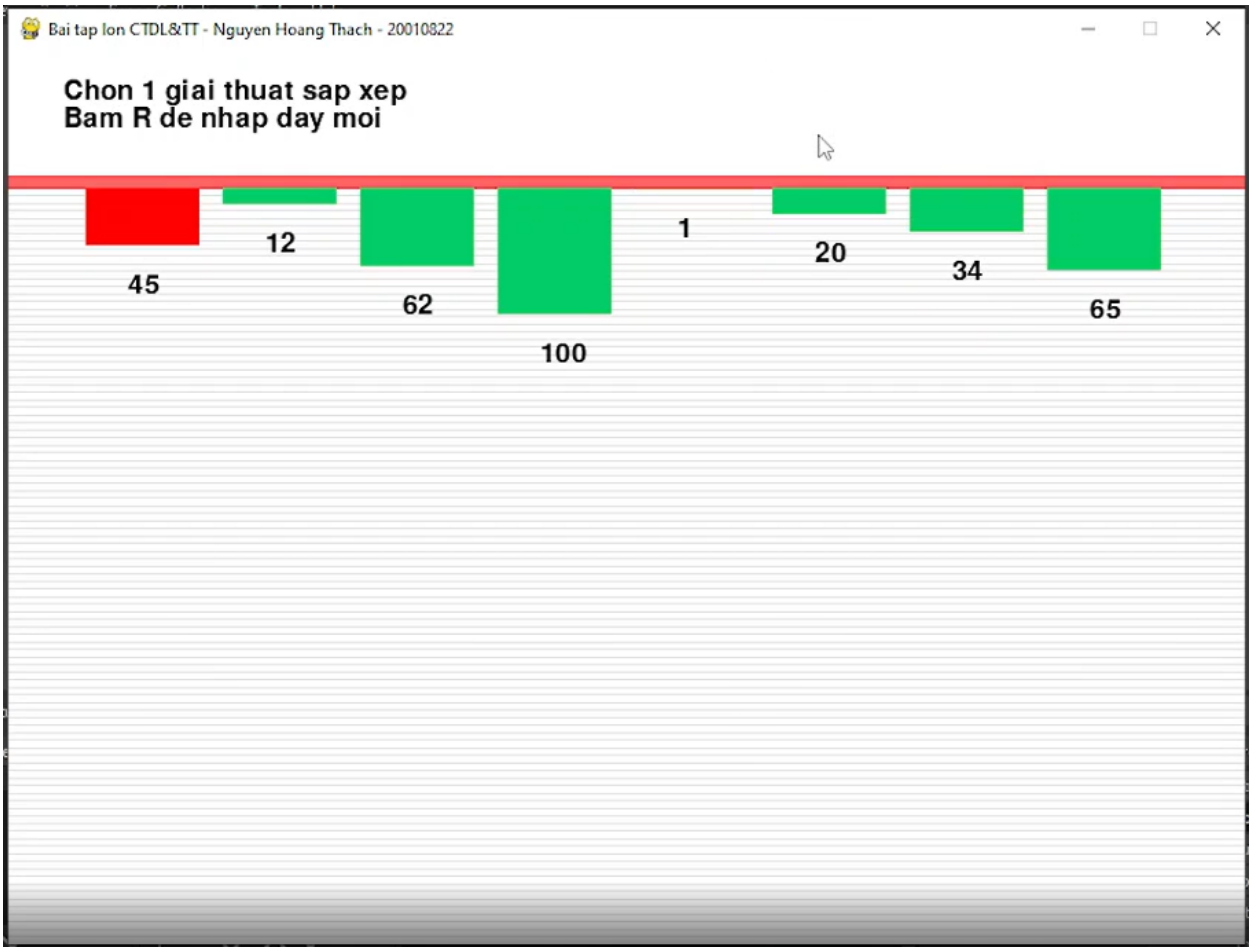
19) Mô tả Bubble sort (4)

Array[7]<array[6] => đổi chỗ



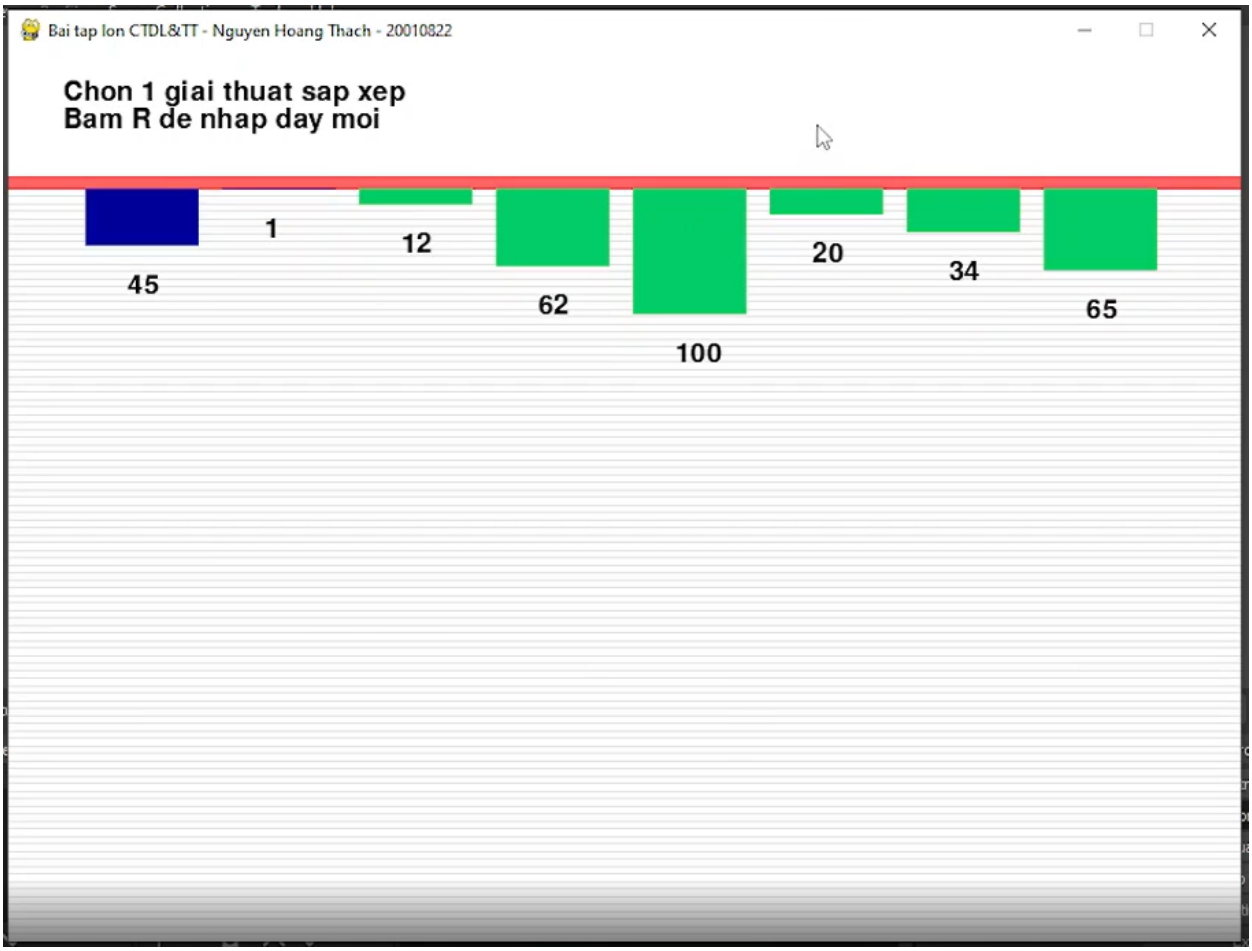
20) Mô tả Bubble sort (5)

Xét j=6



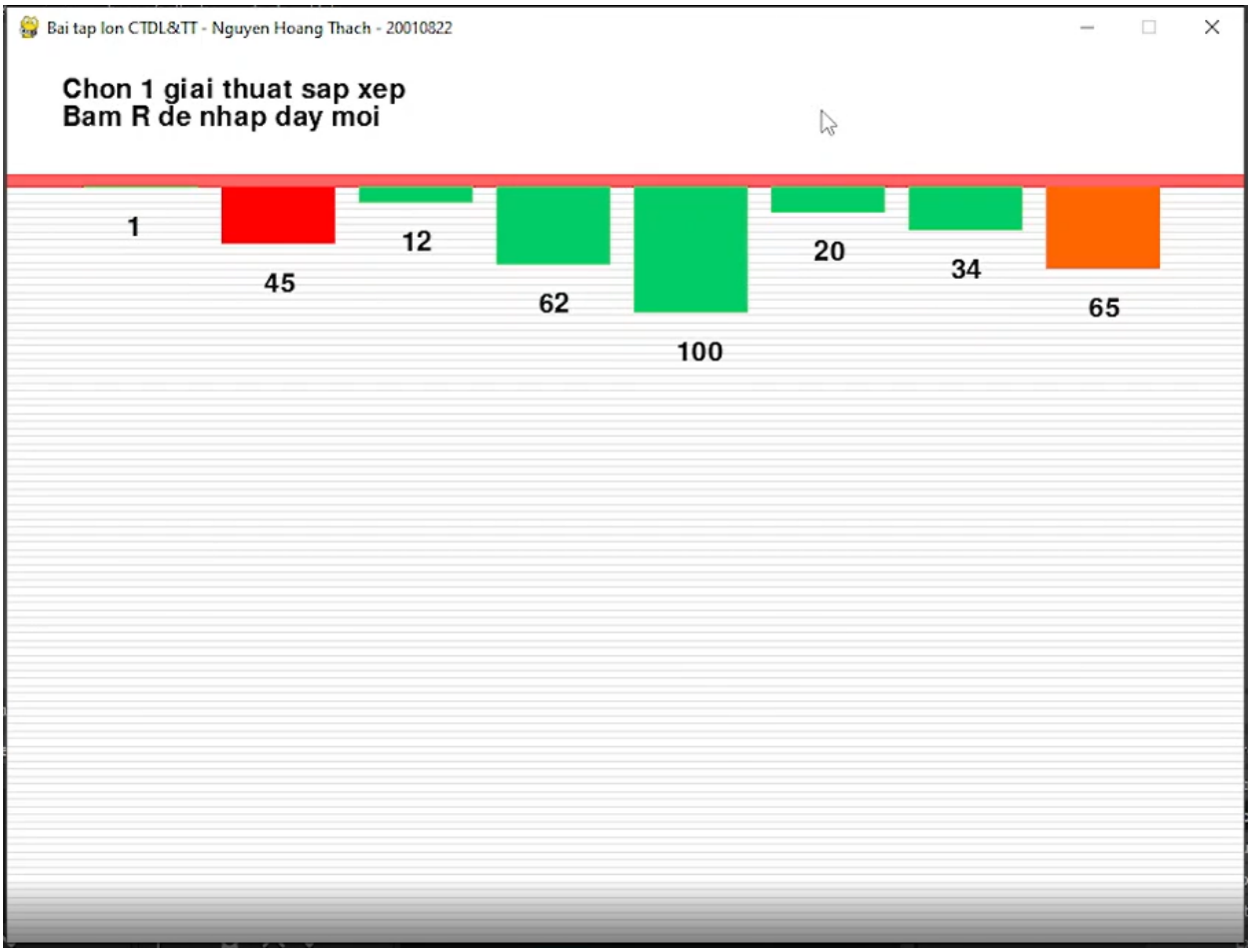
21) Mô tả Bubble sort (6)

Xét j=5



22) Mô tả Bubble sort (7)

Cứ như thế cho đến cuối cùng khi ta thấy array[2]<array[1] thì đổi chỗ



23) Mô tả Bubble sort (8)

Như vậy sau khi xét array[1..8] ta kéo được 1 là phần tử nhỏ nhất lên đầu dãy và tiếp tục xét tiếp dãy array[2..8]

- Selection sort:

+) Lí thuyết:

- Ở lượt thứ nhất, ta chọn trong dãy khóa k[1..n] ra khóa nhỏ nhất (khoá ≤ mọi khóa khác) và đổi giá trị của nó với k[1], khi đó giá trị khóa k[1] trở thành giá trị khóa nhỏ nhất.

- Ở lượt thứ hai, ta chọn trong dãy khóa k[2..n] ra khóa nhỏ nhất và đổi giá trị của nó với k[2].

- Ở lượt thứ i, ta chọn trong dãy khóa k[i..n] ra khóa nhỏ nhất và đổi giá trị của nó với k[i].

-Tới lượt thứ n - 1, chọn trong hai khóa k[n-1], k[n] ra khóa nhỏ nhất và đổi giá trị của nó với k[n-1].

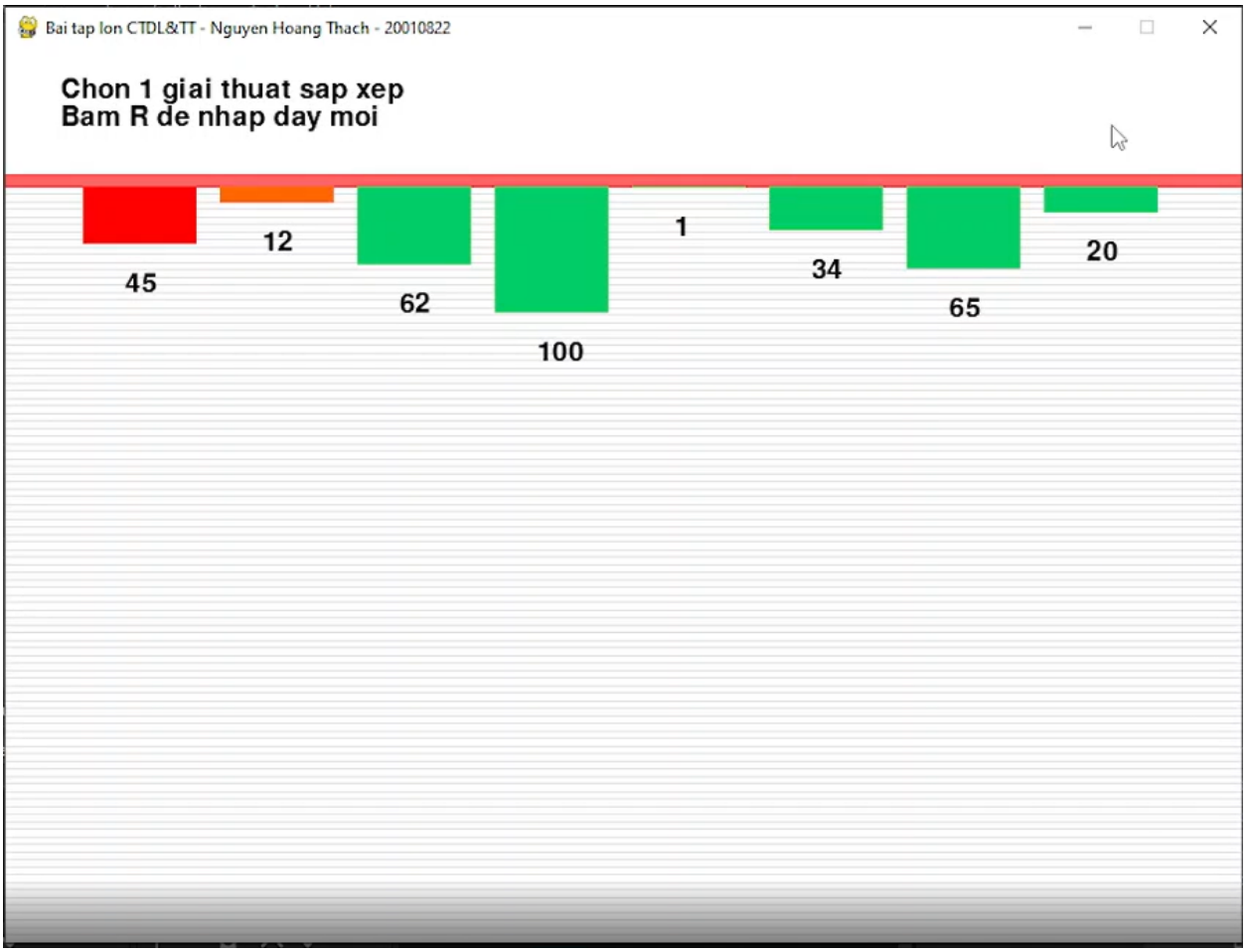
+) Mô tả lấy ví dụ với Visualize:

Lấy ví dụ như trên: n=8

array[1]= 45 array[2]=12 array[3]=62 array[4]=100 array[5]=1 array[6]=34 array[7]=65 array[8]=20

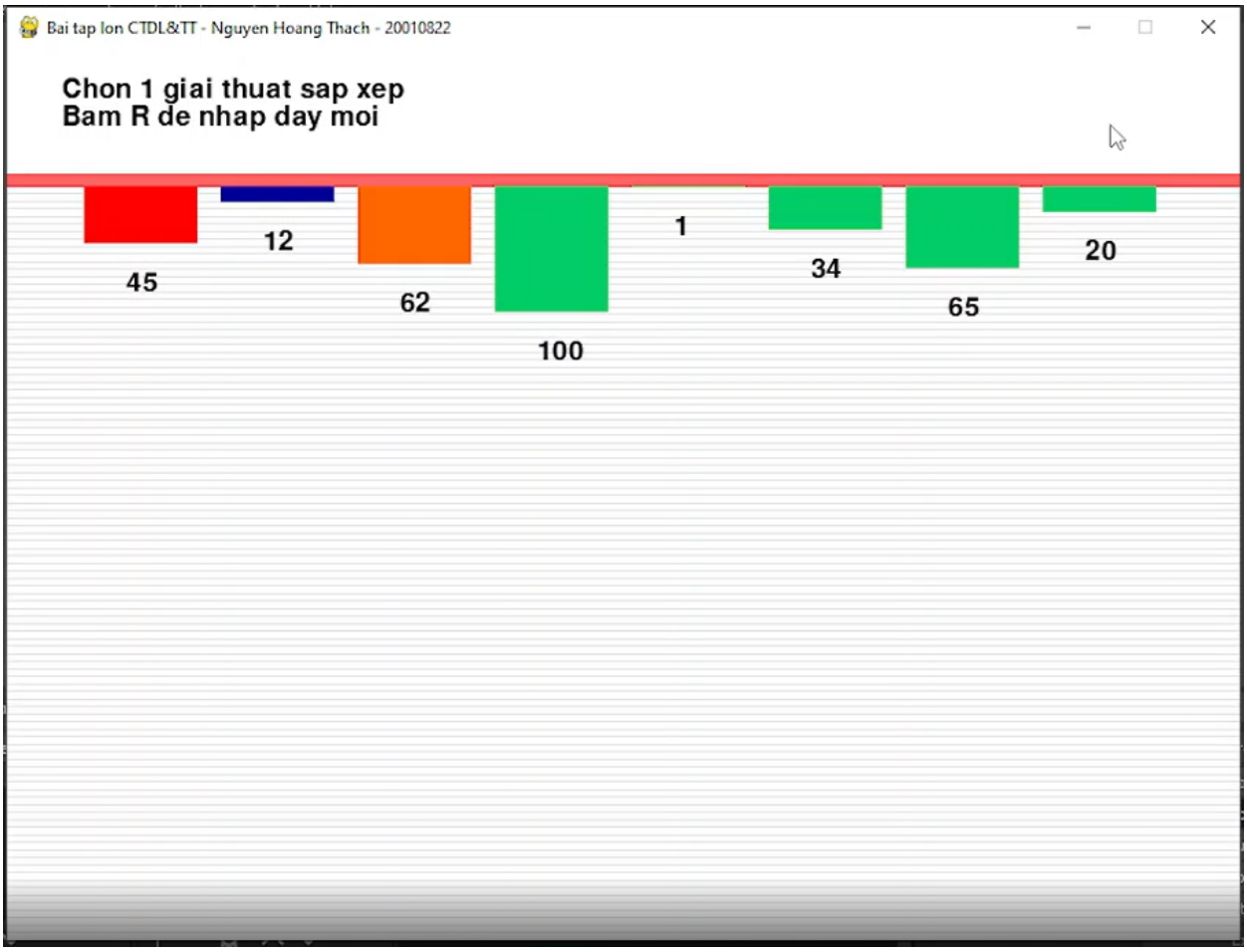
Ta xét phần tử thứ i sau đó xét tất cả các số ở sau nó trong dãy array[i+1..n] để tìm ra một số x được xem là nhỏ nhất mà nhỏ hơn số array[i]. Sau khi chạy hết dãy array[i+1..n] và tìm ra được số cần tìm ta sẽ thay đổi số đó với vị trí phần tử array[i] cho nhau. Cứ tiếp tục xét như vậy cho đến dãy cuối cùng

Ở đây phần tử thứ i trong vòng lặp được thể hiện qua màu đỏ, phần tử thứ j được thể hiện qua màu cam và nếu tìm được x thì x sẽ được biểu thị bằng màu xanh đậm



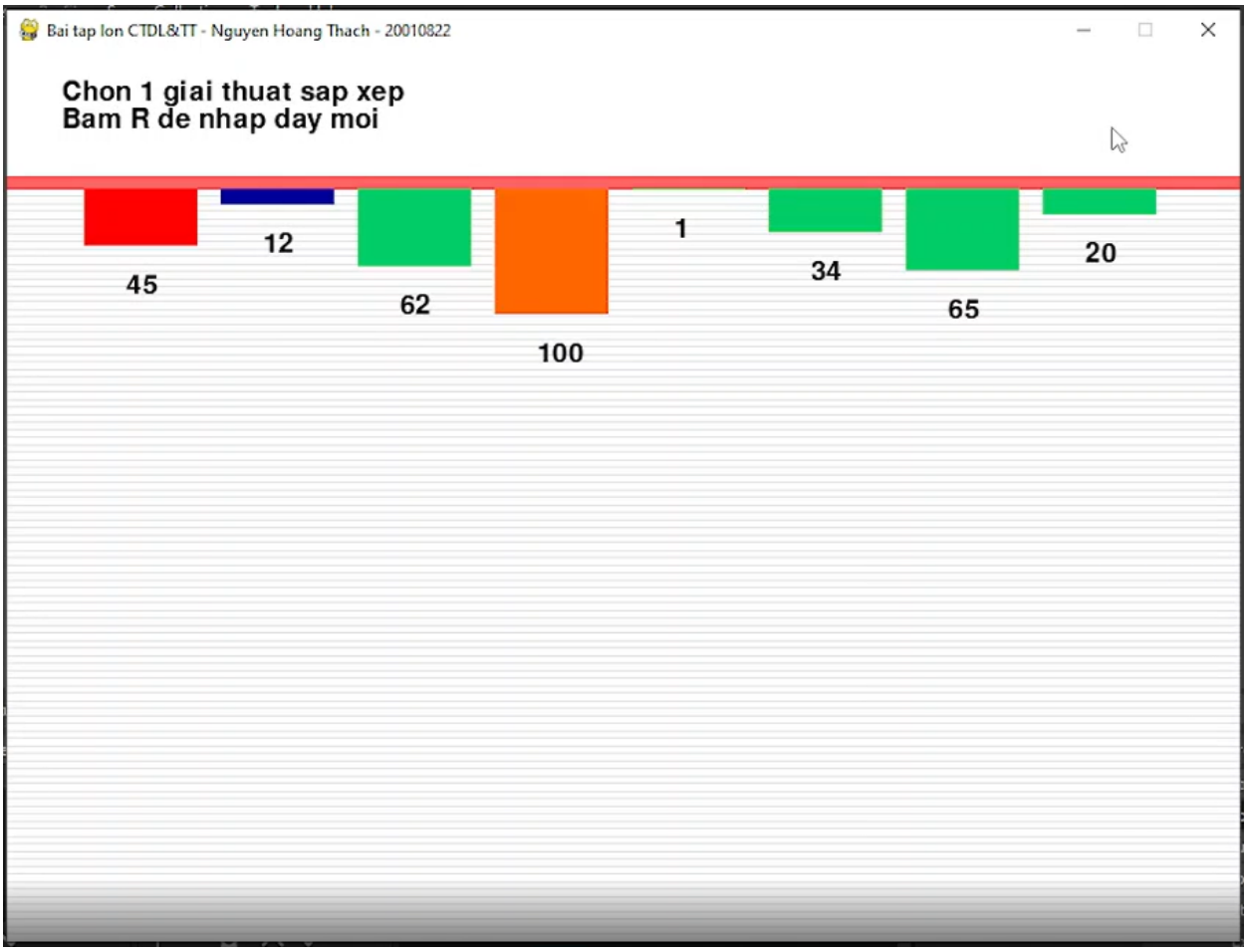
23) Mô tả Selection sort (1)

Ban đầu xét: i=1 xét tất cả các số sau phần tử đầu tiên j chạy từ 2



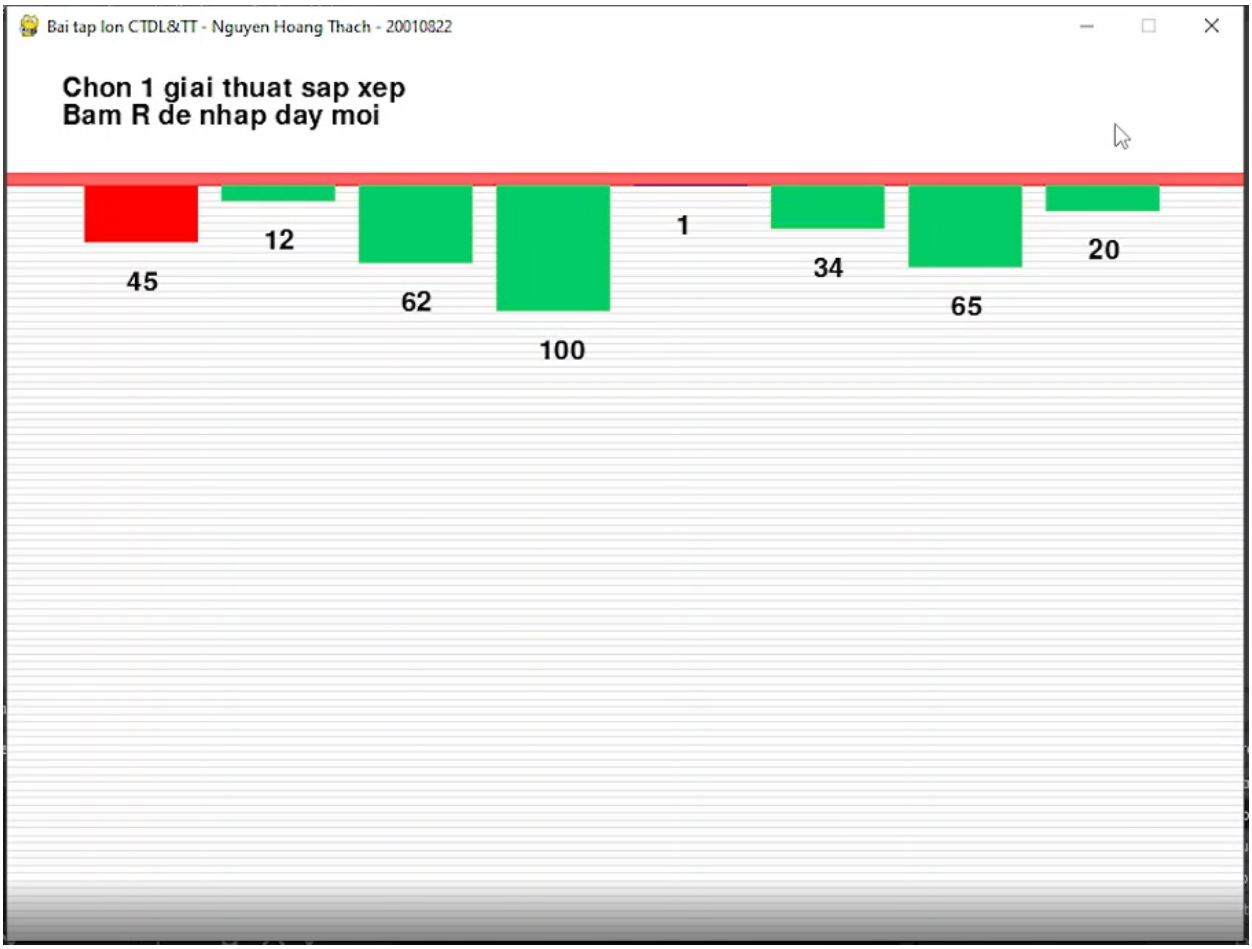
24) Mô tả Selection sort (2)

Ta thấy array[2]<45 => x= 2 và được biểu thị bằng xanh đậm và tiếp tục xét j=3



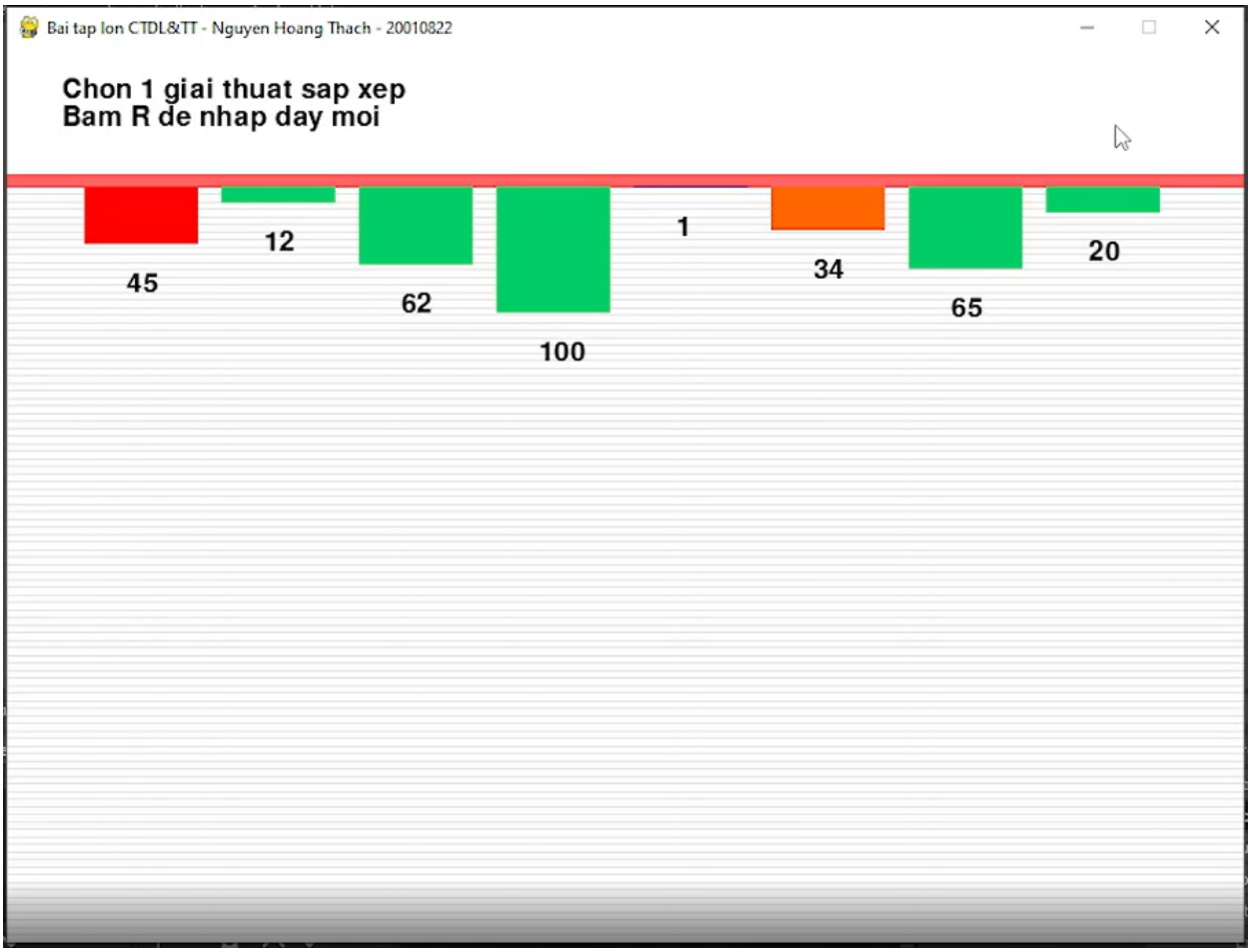
25) Mô tả Selection sort (3)

Xét j=4



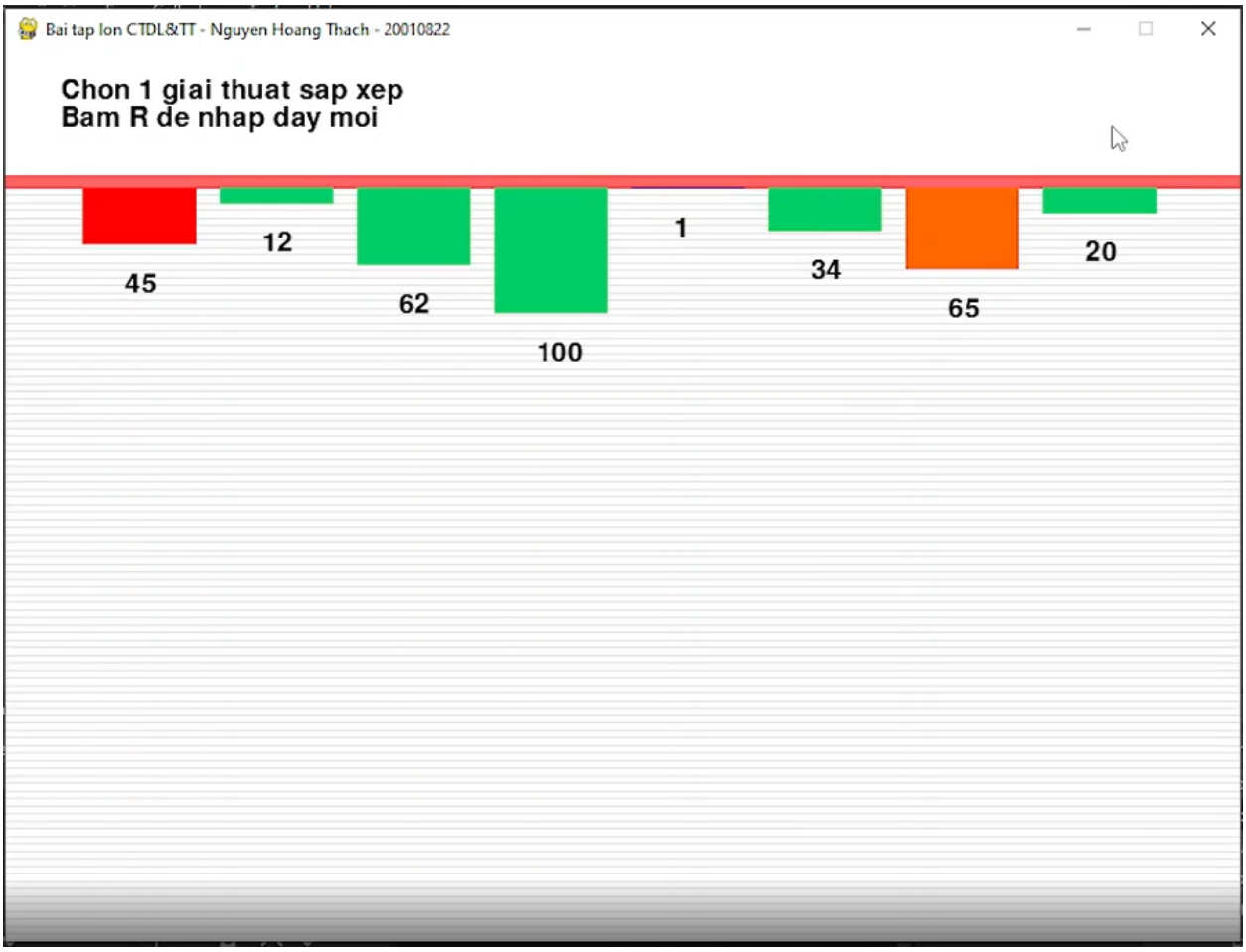
26) Mô tả Selection sort (4)

Xét j=5 và thấy array[5]< array[x] (array[x]=12) ta gán x=5



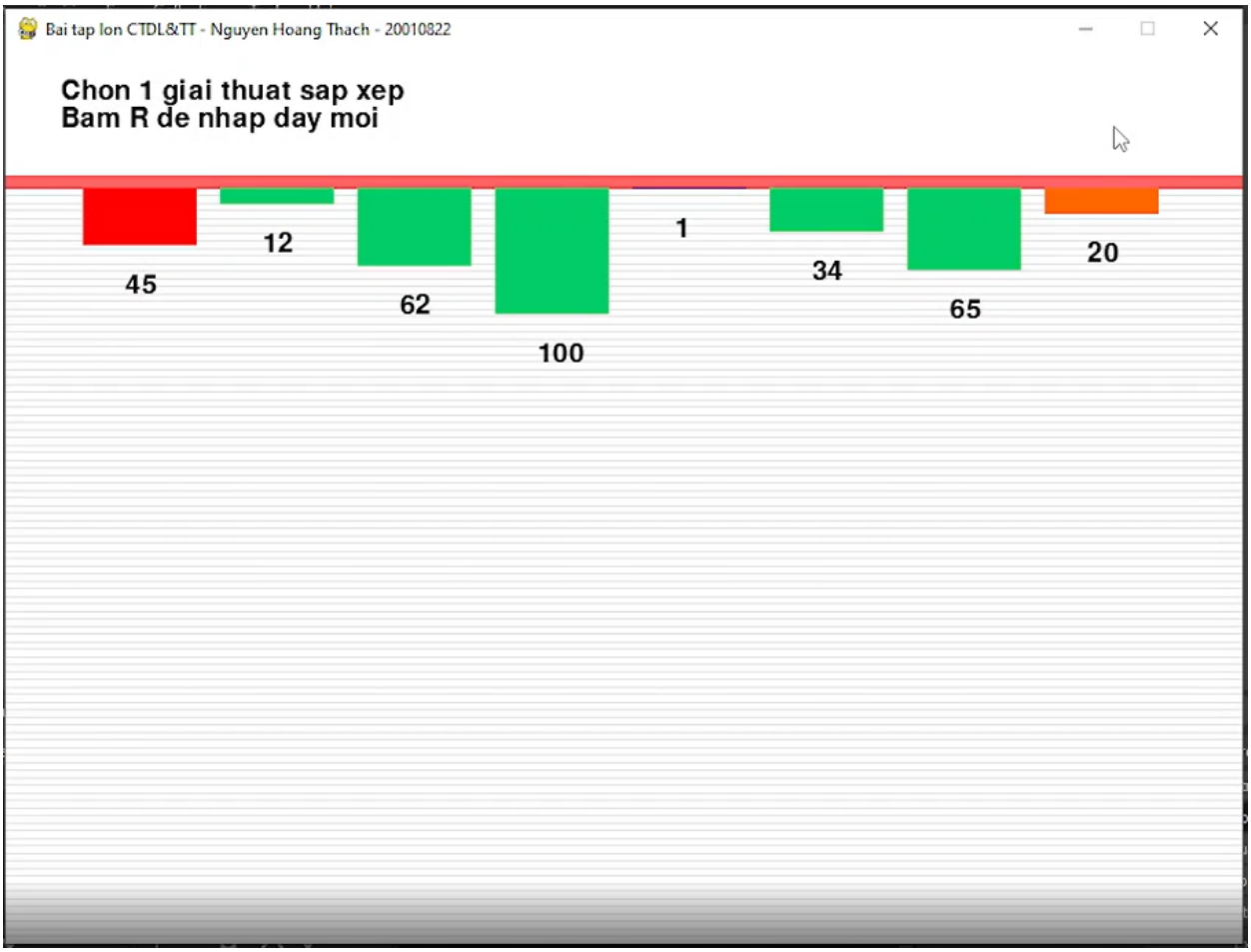
27) Mô tả Selection sort (5)

Xét j=6



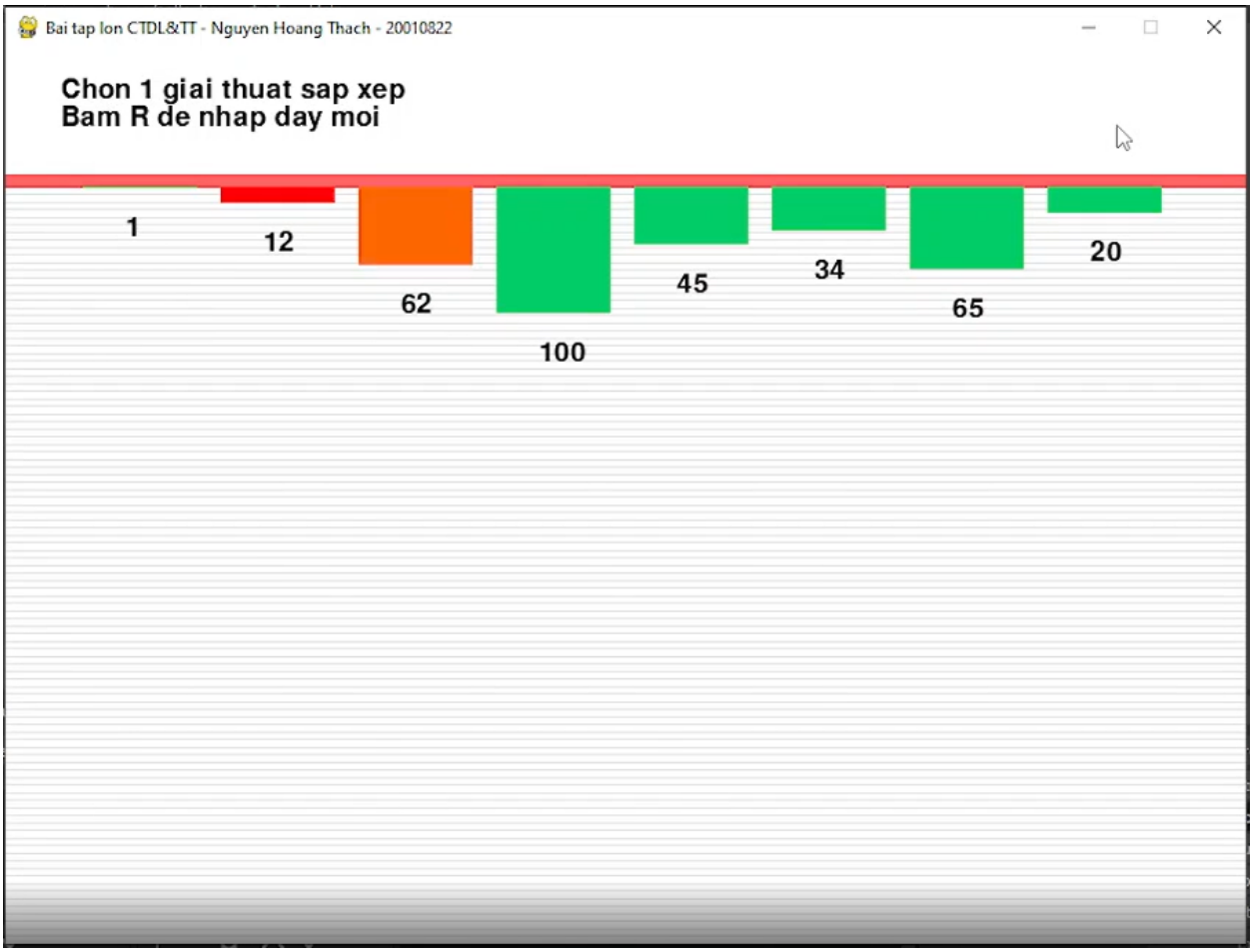
28) Mô tả Selection sort (6)

Xét j=7



29) Mô tả Selection sort (7)

Xét j=8 kết thúc vòng lặp đầu tiên



30) Mô tả Selection sort (8)

Sau khi kết thúc vòng lặp, phần tử array[1] sẽ đổi chỗ cho phần tử array[x] ở đây x=5 hay array[x]=1 như vậy số nhỏ nhất sẽ được đưa lên đầu và tiếp tục xét dãy array[2..8].

4. Cấu trúc hàm code

- **Hàm vẽ giao diện**

  Là hàm sẽ đóng vai trò vẽ biểu đồ biểu thị từng giá trị của dãy nhập vào thông qua những hình chữ nhật. độ cao của mỗi hình chữ nhật sẽ dựa vào độ lớn giá trị của mỗi phần tử trong dãy

  Màu sắc của mỗi cột sẽ dc biểu thị thông qua dãy arr_clr khi cột i sẽ có màu arr_clr[i]

```
def draw(array,arr_clr,n):
    txt = fnt.render("Chon 1 giai thuat sap xep", 1, (0, 0, 0))
    screen.blit(txt, (40, 20))
    txt1 = fnt.render("Bam R de nhap day moi",1, (0, 0, 0))
    screen.blit(txt1, (40, 40))
    #txt2 = fnt1.render("Selection sort", 1, (0, 0, 0))
    #screen.blit(txt2, (40, 60))
    element_width =(width-150)//n
    boundry_arr = 900 / n
    boundry_grp = 550 /100
```

```
    pygame.draw.line(screen,(255,100,100), (0,95), (900,95),10)
    for i in range(1, 100):
        pygame.draw.line(screen,(224, 224, 224),(0, 5.5 * i + 100),
(900, 5.5 * i + 100), 1)
    maxarr=-sys.maxsize-1
    for i in range (1,n):
        if array[i]>maxarr:
            maxarr=array[i]
    tile=maxarr/100
    if tile<1:
        tile=1
    else:
        tile=tile+5
    for i in range(1, n):
        number=str(array[i])
        txtnb=fnt.render(number,1,(0,0,0))
        pygame.draw.line(screen, arr_clr[i],(boundry_arr * i-3,
100),(boundry_arr * i-3,(array[i]*boundry_grp)/(tile) +
100),element_width)
        screen.blit(txtnb,(boundry_arr * i-13,
array[i]*boundry_grp/(tile)+ 120))
```

- **Hàm refill**

Là một hàm quan trong khi nó sẽ thực hiện hàm vẽ liên tục để thay đổi trên biểu đồ đảm bảo tính liên tục của visualize với một tốc độ (fps) được cài đặt trong đó

```
def refill(array,arr_clr,n):
    screen.fill((255, 255, 255))
    draw(array,arr_clr,n)
    pygame.display.update()
    pygame.time.delay(50)
```

- **Hàm Selection sort**

Hàm thực hiện Selection sort, sau mỗi lần i và j thay đổi sẽ thực hiện refill() để thay đổi biểu đồ

```
def Selectionsort(array, l, r,arr_clr):

    pygame.event.pump()
    for i in range (l,r+1):
        arr_clr[i]= clr[1]
        refill(array,arr_clr,len(array))
        m=i
        for j in range(i+1,r+1):
            arr_clr[j]=clr[3]
            refill(array,arr_clr,len(array))
            t=m
            if (array[m]>array[j]):
                m=j
```

```
            if (m!=i):
                if (t!=i):
                    arr_clr[t]=clr[0]
                arr_clr[m]=clr[2]
                refill(array,arr_clr,len(array))
            arr_clr[j]=clr[0]
            if m==j:
                arr_clr[m]=clr[2]
        if (m!=i):
            t=array[m]
            array[m]=array[i]
            array[i]=t
        arr_clr[m]=clr[0]
        arr_clr[i]=clr[0]
```

- **Hàm Bubble sort**

Hàm thực hiện Bubble sort, sau mỗi lần i và j thay đổi sẽ thực hiện refill() để thay đổi biểu đồ

```
def bubblesort(array,l,r,arr_clr):
    pygame.event.pump()
    for i in range(l,r+1):
        arr_clr[i]=clr[1];
        refill(array,arr_clr,len(array))
        for j in range(r,i,-1):
            arr_clr[j]=clr[3];
            refill(array,arr_clr,len(array))
            if (array[j]<array[j-1]):
                arr_clr[j]=clr[2];
                arr_clr[j-1]=clr[2];
                refill(array,arr_clr,len(array))
                t=array[j]
                array[j]=array[j-1]
                array[j-1]=t
                arr_clr[j]=clr[3];
                arr_clr[j-1]=clr[0];
                refill(array,arr_clr,len(array))
            arr_clr[j]=clr[0]
        arr_clr[i]=clr[0]
```

- **Hàm Insertion sort**

Hàm thực hiện Insertion sort, sau mỗi lần i và j thay đổi sẽ thực hiện refill() để thay đổi biểu đồ

```
def insertionsort(array,l,r,arr_clr):
    pygame.event.pump()
    for i in range(l+1,r+1):
```

```
        arr_clr[i]=clr[1];
        refill(array,arr_clr,len(array))
        x=array[i]
        j=i-1
        arr_clr[j]=clr[3];
        refill(array,arr_clr,len(array))
        check=0
        while x<array[j] and j>=1:
            check=1
            arr_clr[j]=clr[3];
            refill(array,arr_clr,len(array))
            array[j+1]=array[j];
            j=j-1
        #arr_clr[i]=clr[1];
        if check==1:
            arr_clr[j+1]=clr[2];
            refill(array,arr_clr,len(array))
            array[j+1]=x
        for z in range(j,i):
            arr_clr[z]=clr[0]
        refill(array,arr_clr,len(array))
        arr_clr[i]=clr[0];
```

-Và một số hàm liên quan đến biểu diễn giao diện của chương trình: main_menu(), input_n(),…..