**Phần 1: Cơ bản Về Giải Thuật**

**Bài 1: giới thiệu về giải thuật**

**Thuật toán Euclid**

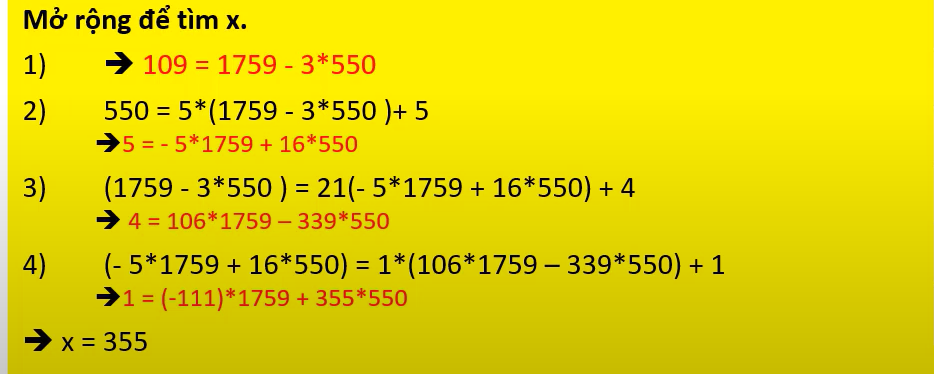
Giả sử cần tìm x sao cho

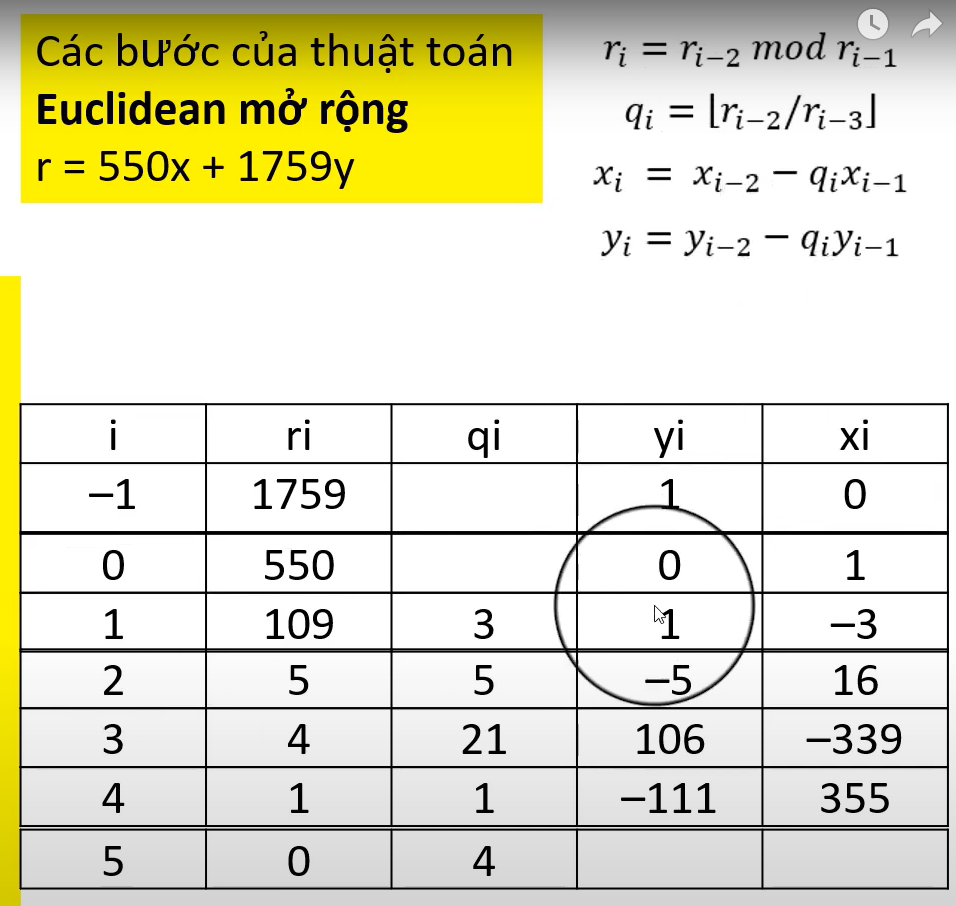
550x = 1 (mod 1759), hay nói cách khác là cần tìm biểu diễn

550x + 1759y = 1

Ta sẽ tìm USCLN:

* Biểu diễn số lớn qua số nhỏ: **1759** = 3 \* **550** + 109
* Biểu diễn số nhỏ qua số dư: **550** = 5 + 109 + 5
* Biểu diễn **109** qua số dư của biểu thức ở trên:: **109** = 21 \* 5 + 4
* Tiếp tục: 5 = 4 \* 1 + 1
* Tiếp tục: 4 = 4 \* 1 + 0





Thuật toán Euclid dùng để xác định các hệ số trong đẳng thức Bezout

Bổ đề Bezout: nếu **d** là ước số chung lớn nhất của hai số nguyên **a** và **b**, thì sẽ tồn tại hai số nguyên **x** và **y** sao cho:

d = a\*x + b \*y

**--------------- --------------- --------------- ---------------**

**Thuật toán nổi bọt**

package com.example.sort**;**import java.util.Scanner**;**public class BubbleSort {  
 static void swap(int[] arr**,** int i**,** int j) {  
 var temp = arr[i]**;** arr[i] = arr[j]**;** arr[j] = temp**;** }  
  
 public static void main(String[] args) {  
 Scanner sc = new Scanner(System.*in*)**;** System.*out*.print("Enter the number of elements you want to store: ")**;** int n = sc.nextInt()**;** int[] array = new int[n]**;** System.*out*.println("Input your array from keyboard:")**;** for (int i = **0;** i < n**;** i++) {  
 array[i] = sc.nextInt()**;** }  
  
 for (int i = **0;** i < array.length**;** i++) {  
 for (int j = array.length - **1;** j > i**;** j--) {  
 if (array[j] < array[j - **1**]) {  
 *swap*(array**,** j**,** j- **1**)**;** }  
 }  
 }  
  
 System.*out*.println("New array after sorted")**;** for (int j : array) {  
 System.*out*.println(j)**;** }  
 }  
}

**Sắp xếp lựa chọn**

Đây là thuật toán sắp xếp dựa trên quá trình tìm số nhỏ nhất trong chuỗi dữ liệu (không bao gồm các phần tử đã được sắp xếp). Trong đó phần tử có giá trị nhỏ nhất sẽ thay thế cho phần tử có số thứ tự nhỏ nhất

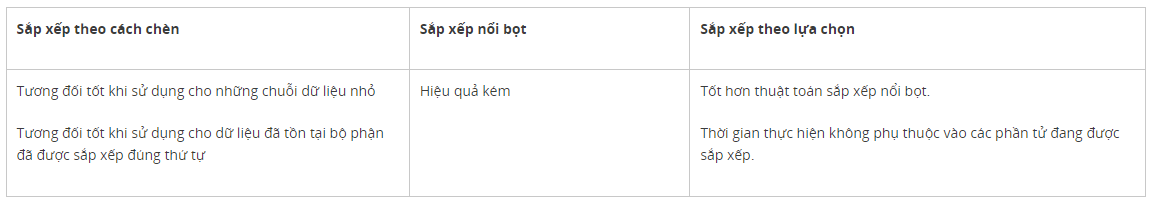
static void selectionSort(int[] array) {  
 for (int i = ***0***; i < array.length - ***1***; i++) {  
 int minIndex = i;  
 for (int j = i + ***1***; j < array.length; j++) {  
 if (array[j] < array[minIndex]) {  
 minIndex = j;  
 }  
 }  
 swap(array, i, minIndex);  
 }  
}

**Sắp xếp chèn**

Thuật toán sắp xếp chèn sử dụng thứ tự đã tuân theo quy tắc sắp xếp có sẵn của dữ liệu. Trong quá trình thực hiện, mỗi phần tử sẽ được so sánh với các phần tử theo thứ tự đó và chèn (vào giữa, phía trước hoặc phía sau) nhằm thỏa mãn quy tắc sắp xếp

static void insertionSort(int[] arr) {  
 for (int i = ***1***; i < arr.length; i++) {  
 int ele = arr[i];  
 int j = i;  
 while (j - ***1*** >= ***0*** && arr[j - ***1***] > ele) {  
 arr[j] = arr[j - ***1***];  
 arr[j - ***1***] = ele;  
 j--;  
 }  
 }  
}

Độ hiệu quả của các thuật toán sắp xếp



**Giải thích kết quả thực nghiệm.**

* Các thuật toán sắp xếp đều có hai vòng lặp được chạy lồng nhau, do đó độ phức tạp trung bình của các thuật toán sẽ là O(n2). Tuy nhiên các thuật toán này vẫn có những điểm khác biệt như sau, tương ứng với từng trường hợp cụ thể:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Sắp xếp nổi bọt (Bubble sort)** | **Sắp xếp chọn (Selection sort)** | **Sắp xếp chèn (Insertion Sort)** |
| **Nhận xét:** | Là một thuật toán sắp xếp dễ hiểu, nhưng kém hiệu quả và ít khi được sử dụng trong thực tế | Thuật toán này tốt hơn thuật toán sắp xếp nổi bọt, và thuật toán này có thời gian chạy **không** phụ thuộc vào thứ tự của mảng ban đầu, do chúng ta luôn luôn chạy với 1 con trỏ từ phần tử đầu tiên đến phần tử cận cuối cùng, và một con trỏ chạy từ con trỏ đầu tiên đến phần tử sau cùng để tìm vị trí của phần tử nhỏ nhất trong những phần tử còn lại của mảng và thay đổi vị trí với phần tử ở vị trí bên phải | Đây là một thuật toán sắp xếp linh hoạt, hiệu quả ở những mảng ít phần tử, và thuật toán này chạy khá nhanh trên những mảng đã gần như được sắp xếp vì các bước để “đẩy” các phần tử để tạo ra slot cho item mới sẽ được giảm đi rất nhiều. |
| **Tình huống tốt nhất (best case)** | Độ phức tạp: O(n), đi qua đúng n phần tử, đối với mảng đã được sắp xếp hoặc gần như đã được sắp xếp, khi đó thuật toán sẽ dừng ở lần lặp đầu tiên, và không có sự hoán đổi vị trí nào được thực hiện | Như đã nói, thuật toán này có thời gian chạy không thực sự phụ thuộc vào thứ tự của mảng như thế nào, tuy nhiên với mảng đã được sắp xếp, thuật toán này sẽ chạy nhanh hơn vì không mất thời gian cho việc “swap” các phần tử | Best case: O(n), đối với mảng đã được sắp xếp, khi đó sắp xếp chèn sẽ chỉ duyệt mảng đúng một lần mà thôi. |
| **Tình huống xấu nhất:** | Độ phức tạp O(n2), mảng bị sắp xếp ngược, khi đó với mỗi lần duyệt của con trỏ đầu tiên, con trỏ thử hai sẽ duyệt để đưa phần tử nhỏ nhất về vị trí của nó (về bên trái) | Gần như không có worst case, thời gian chạy sẽ gần như nhau trong mọi trường hợp | O(n2), trường hợp mảng bị sắp xếp ngược, khi đó với mỗi lần duyệt từ con trỏ thứ hai về phần từ đầu tiên, nó luôn luôn phải duyệt tất cả phần tử để “đẩy” mảng lên, thay thế phần tử ở vị trí con trỏ đầu tiên cho phần tử đầu tiên này. |

**--------------- --------------- --------------- ---------------**

**Độ chính xác của thuật toán**

**--------------- --------------- --------------- ---------------**

**Giới thiệu về cấu trúc dữ liệu**

Trong khoa hoc máy tính, cấu trúc dữ liệu là một cách tổ chức dữ liệu trong máy tính sao cho nó có thể được sử dụng một cách hiệu quả cho các thuật toán tác động đến nó

Trong việc thiết kế chương trình, việc chọn cấu trúc dữ liệu là vấn đề quan trọng, kinh nghiệm cho thấy khó khăn của việc triển khai chương trình, chất lượng và hiệu năng của kết quả cuối cùng phụ thuộc rất nhiều vào việc chọn CTDL phù hợp

**Quiz**

**1 A**

**2 A**

**3 A + B**

**4 B + D**

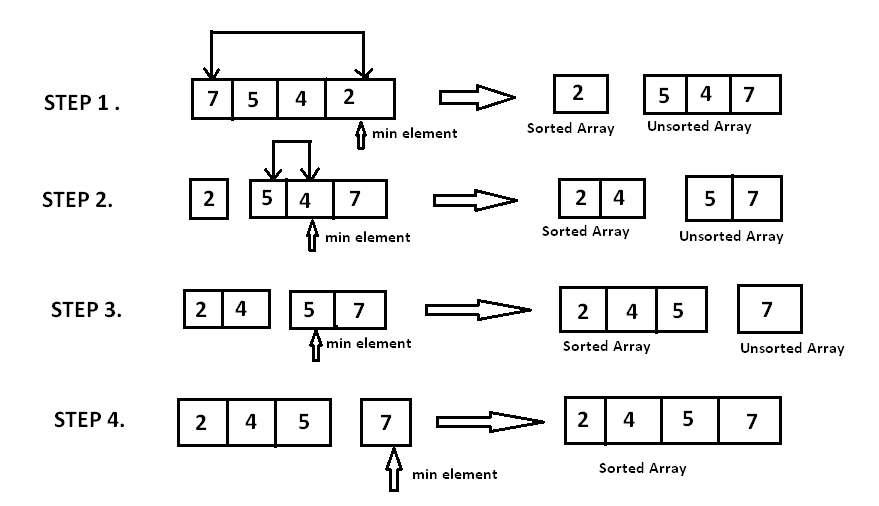
**5 A + B + C**

**--------------- --------------- --------------- --------------- ---------------**

**Thuật toán tìm kiếm và sắp xếp cơ bản**

**Sắp sếp chọn (selection sort)**

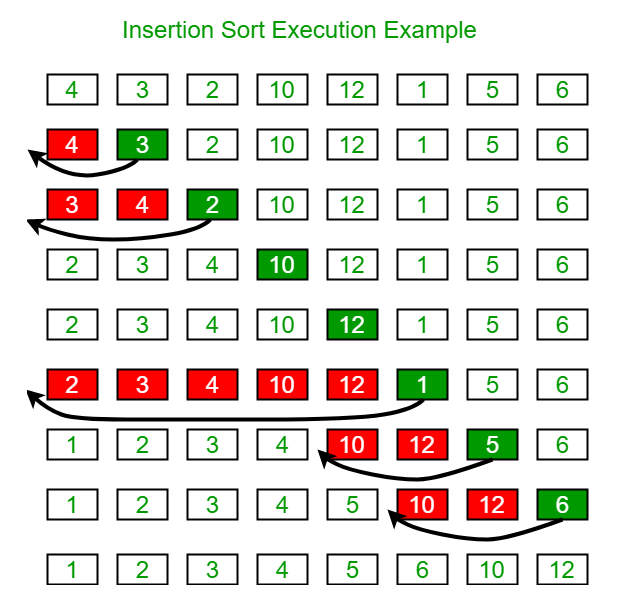
Danh sách chứa các phần tử sẽ được chia làm hai phần, phần bên trái là phần được sắp xếp (sort list) và phần bên phải là phần chưa được sắp xếp (unsorted list). Phần tử nhỏ nhất trong list sẽ được chọn và tráo đổi với vị trí đầu tiên, tiếp đến là phần tử nhỏ thứ 2 sẽ được đặt ở vi jtris thử 2 …



**--------------- --------------- --------------- ---------------**

**Thuật toán chèn (Insertion sort)**

Sắp xếp chèn là thuật toán sắp xếp khi đã có một dãy đã có thử tự, chèn thêm 1 phần tử vào vị trí thích hợp của dãy số đã sắp xếp sao cho dẫy số đó vẫn là dãy có thứ tự.



**--------------- --------------- --------------- ---------------**

**Sắp xếp tại chỗ**

**In place (tại chỗ) sorting algorithms** là những thuật toán mà khi chúng ta sắp xếp dữ liệu trong mảng mà không cần sử dụng thêm bộ nhớ bổ sung, điều này có nghĩa là chúng ta sẽ không cần sử dụng các biến tạm để đổi vị trí các element, như vậy tức là có một vùng bộ nhớ đã được sử dụng ngoài dữ liệu lưu trữ của mảng. Các implementations của chúng ta là IN PLACE, có điều nếu chúng ta sử dụng một lượng không đổi bộ nhớ thêm, như trong ví dụ của chúng ta, chúng ta chỉ sử dụng tạm thời một biến – đó có thể đã được định nghĩa một lần bên ngoài vòng lặp và tái sử dụng, việc sắp xếp này là in place

**--------------- --------------- --------------- ---------------**

**Sắp xếp với Stable và Unstable**

Khi sắp xếp các phần tử trong 1 mảng có rất nhiều phần tử có giá trị sắp xếp giống nhau, vậy sau khi sắp xếp mà trình tự của các phần tử **có giá trị giống nhau không thay đổi** thì gọi là stable, ngược lại là unstable

**--------------- --------------- --------------- --------------- --------------- --------------- --------------- --------------- --------------- --------------- --------------- ---------------**

**Phần 2: cấu trúc dữ liệu tuyến tính**

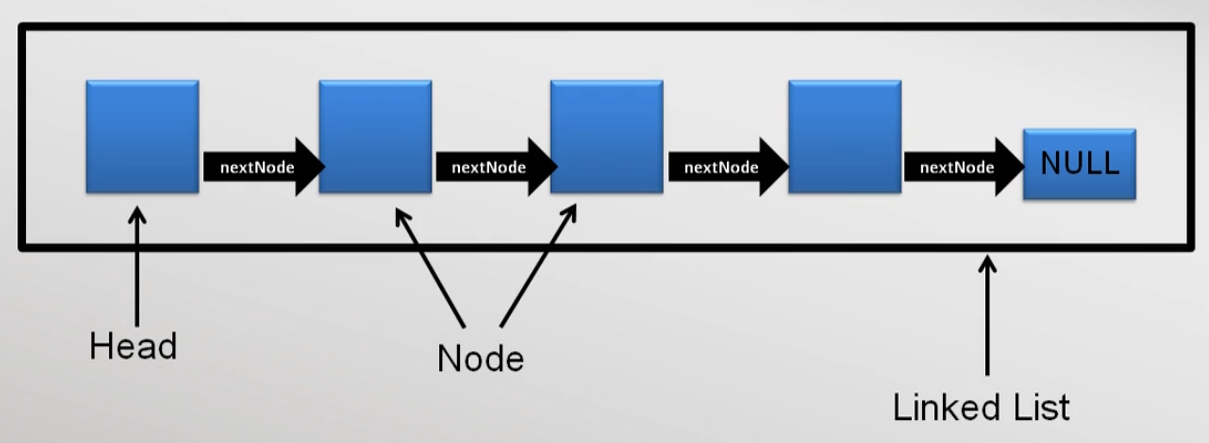
**Bài 3: danh sách dữ liệu Link – List**

**Giới thiệu về danh sách móc nối đơn**

**Danh sách móc nối (link list):** là một kiểu cấu trúc dữ liệu bao gồm một nhóm các node tạo thành một chuỗi. Thông thường mỗi nút gồm dữ liệu (data) ở mút đó và tham chiếu (reference) đến nút kế tiếp trong chuỗi (next). Kiểu cấu trúc dữ liệu này gọi là referential data structure, khác với kiểu array vốn là sequential data structure.

Danh sách móc nối có 3 kiểu chính:

* Danh sách móc nối đơn
* Danh sách móc nối kép
* Danh sách móc nối vòng.



**--------------- --------------- --------------- --------------- --------------- ---------------**

**Bài 4: ngăn xếp (stack) và hàng đợi (queue)**

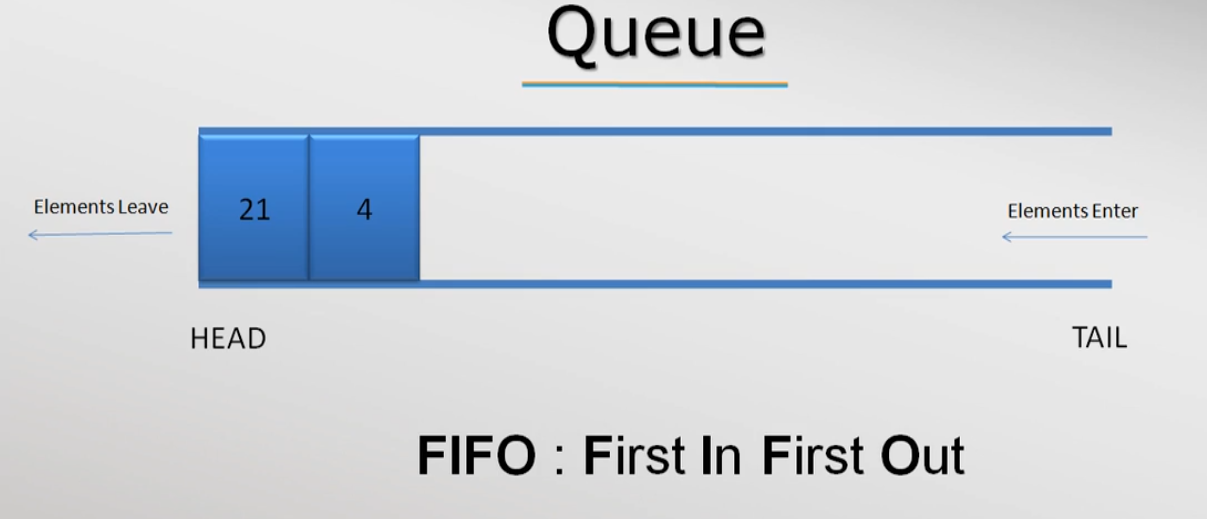
**Stack:** là một cấu trúc dữ liệu trừu tượng hoạt động theo nguyên lý “vào sau ra trước” (last in first out).

Trong **Stack,** các đối tượng có thể đc bổ sung vào stack bất kỳ lúc nào nhưng chỉ được phép lấy ra phần tử ở đỉnh của stack.

**Queue**: Một ngăn xếp là một cấu trúc dữ liệu dạng thùng chưa (container) của các phần tử (thường được gọi là các node) và có hai phép toán cơ bản: push và pop. Push là bổ sung một phần tử vào đỉnh (top) của ngăn xếp, nghĩa là bổ sung vào sau các phần tử đãcó trông ngăn xếp, pop là giải phóng, trả về phần tử đứng ở đỉnh của ngăn xếp. Đối với Queue sẽ là First In First Out,

Đối với **Queue** ta có các method như sau:

* Enqueue: thêm một element vào từ tail
* Dequeue: xóa phần tử head ra khỏi queue và return ra phần tử đó.
* Peek: trả về phần tử ở head mà không remove phần tử đó ra khỏi queue.



**--------------- --------------- --------------- --------------- --------------- ---------------**

**Bài 5: Đệ quy**

**Giới thiệu về đệ quy**

Đệ quy là phương pháp dùng trong các chương trình máy tính, trong đó một hàm tự gọi đến chính nó. Hay nói cách khác, một khái niệm X được định nghĩa theo đệ quy nếu như trong định nghĩa X có sử dụng ngay chính khải niệm X.

Một hàm đệ quy căn bản gồm hai phần:

1. Phần cơ sở (hay còn gọi là tính dừng của đệ quy): chứa các tác động của hàm hoặc thủ tục với một số giá trị cụ thể ban đầu của tham số
2. Phần đệ quy: định nghĩa tác động cần thực hiện cho giá trị hiện thời của các tham số bằng các tác động đã được định nghĩa trước đây với kích thước tham số nhỏ hơn

**Đệ quy đuôi (tail recursion)**

**Tail Recurion** là hàm đệ quy nhưng việc gọi đệ quy được thực hiện ở câu lệnh cuối cùng của hàm, do đó không có bất kỳ câu lệnh nào bị block (câu lệnh chưa được thực hiện)

Ngược lại với **Tail Recurion** tức là hàm mà đệ quy được thực hiện không phải ở vị trí cuối cùng 🡺 có câu lệnh sẽ bị block. Tuy nhiên Java không support việc sử dụng phương pháp này.

**Bài toán tháp Hà Nội**

Có một bộ đĩa gồm các kích thước khác nhau, có lỗ ở giữa, nằm xuyên trên 3 cái cọc. Bài toán rằng phải sắp xếp các đĩa theo trật tự kích thước vào một cọc sao cho các đĩa tạo thành dạng hình nón với đĩa nhỏ nhất nằm trên cùng. Quy tắc chuyển đĩa:

1. Mỗi lần chỉ được chuyển một đĩa
2. Một đĩa chỉ được đăt lên trên đĩa lớn hơn.

**--------------- --------------- --------------- --------------- --------------- --------------- --------------- --------------- --------------- --------------- --------------- ---------------**

**Phần 3: cấu trúc dữ liệu phi tuyến tính**

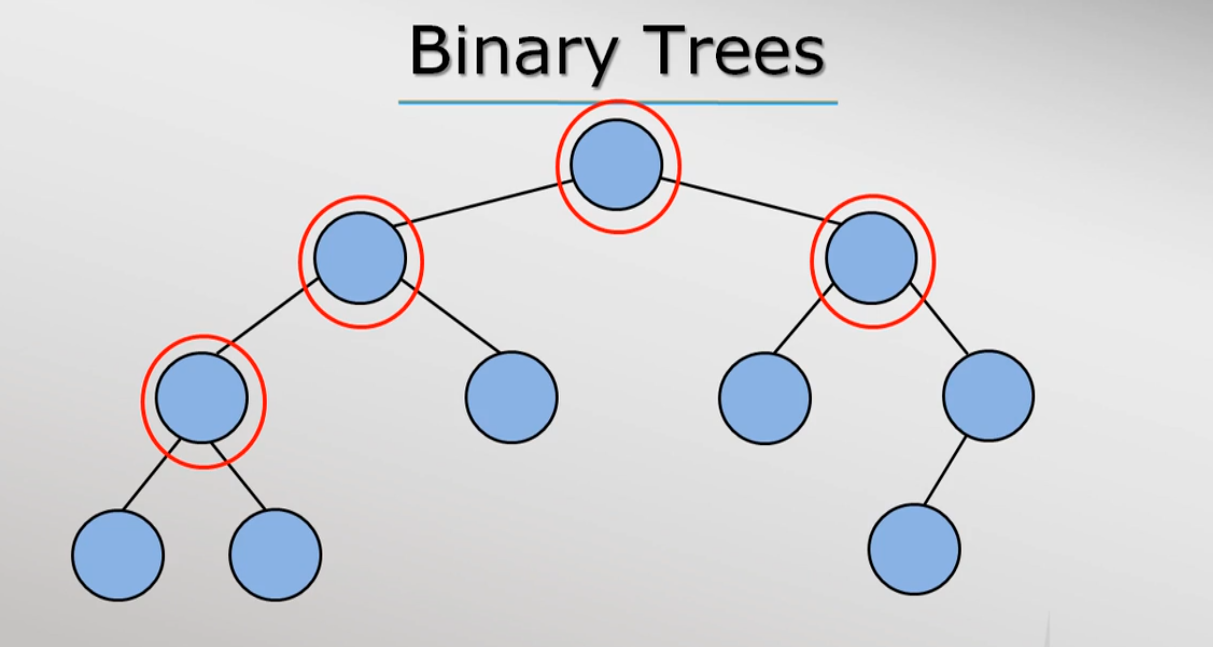
**Bài 6: Cấu trúc dữ liệu cây**

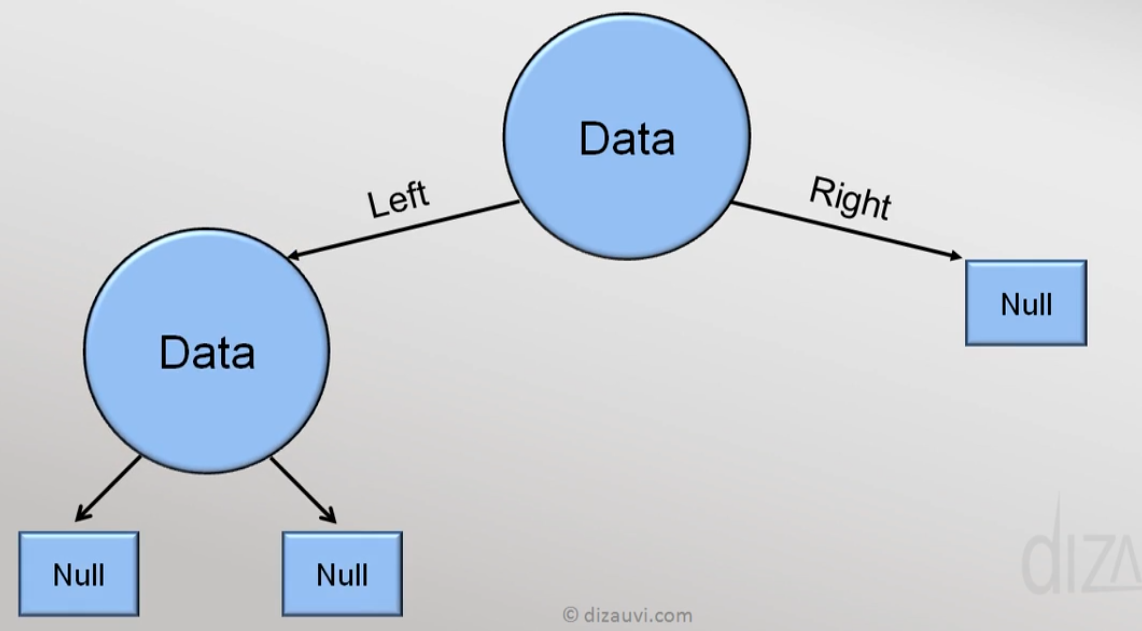
**Giới thiệu cấu trúc dữ liệu cây**

Cấu trúc dữ liệu cây (tree) là một abstract data type được dùng để lưu trữ cấu trúc phân cấp. do đó Tree không phải là cấu trúc dữ liệu tuyến tính. Nó cũng lưu trữ data trong các **node.** Lấy nguồn cảm hứng từ các cây gia phả vậy.

**Cây nhị phân**

Là cấu trúc **Tree** nhưng mỗi node chỉ có 2 node con, mỗi node con sẽ được biểu diễn bằng một con trỏ **left** và **right.** Cây nhị phân hội tụ đủ các lợi ích của mảng sắp xếp (sorted array) và danh sách liên kết (linked list). Chẳng hạn tìm kiếm sẽ nhanh hơn như rtrong sorted array, và việc chèn hoặc xóa phần tử trong cây cũng nhanh như trong linked list



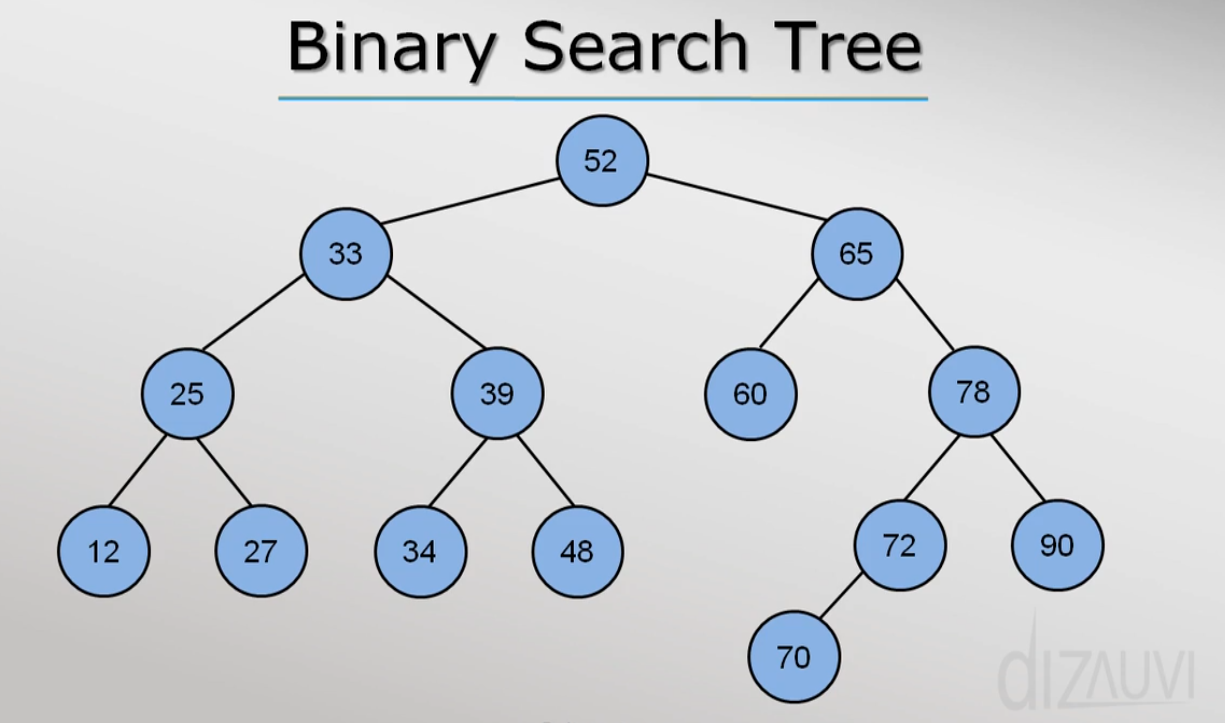


**--------------- --------------- --------------- ---------------**

**Binary Search Tree**

Là một Binary Tree thỏa mãn những điều kiện sau:

* Tất cả các Left Node có value nhỏ hơn Parent Node, Grandparent Node … của nó.
* Tất cả các Right Node có value lớn hơn Parent Node, Grandparent Node … của nó

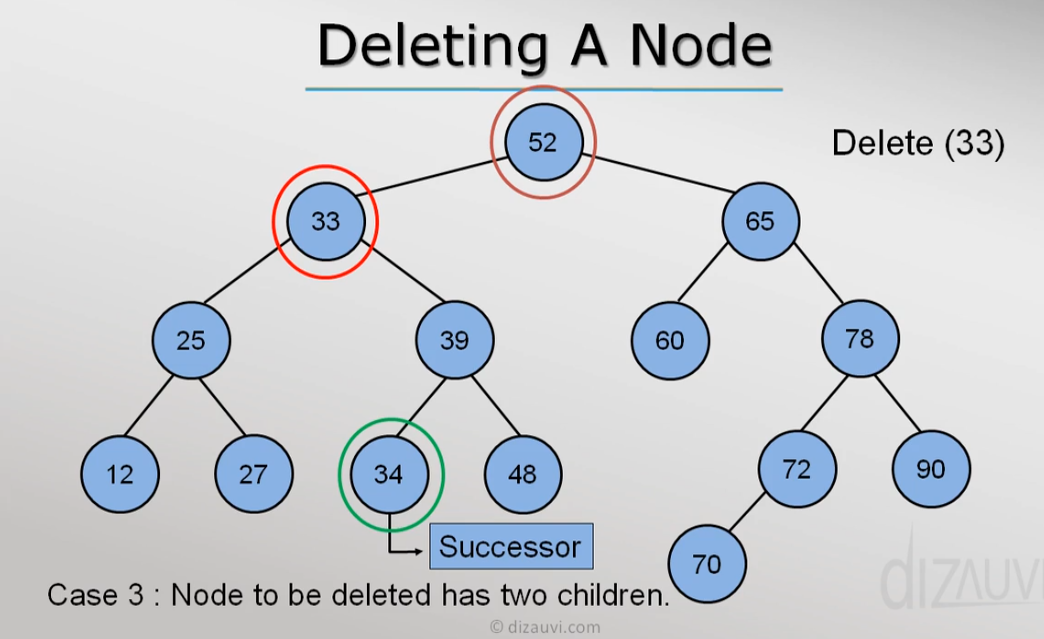


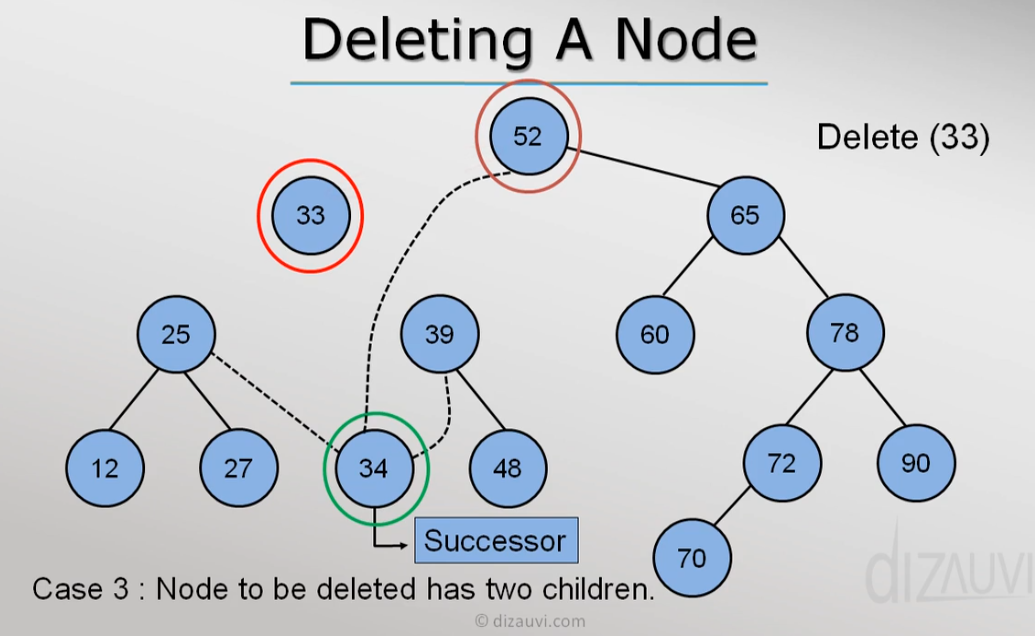
**--------------- --------------- --------------- ---------------**

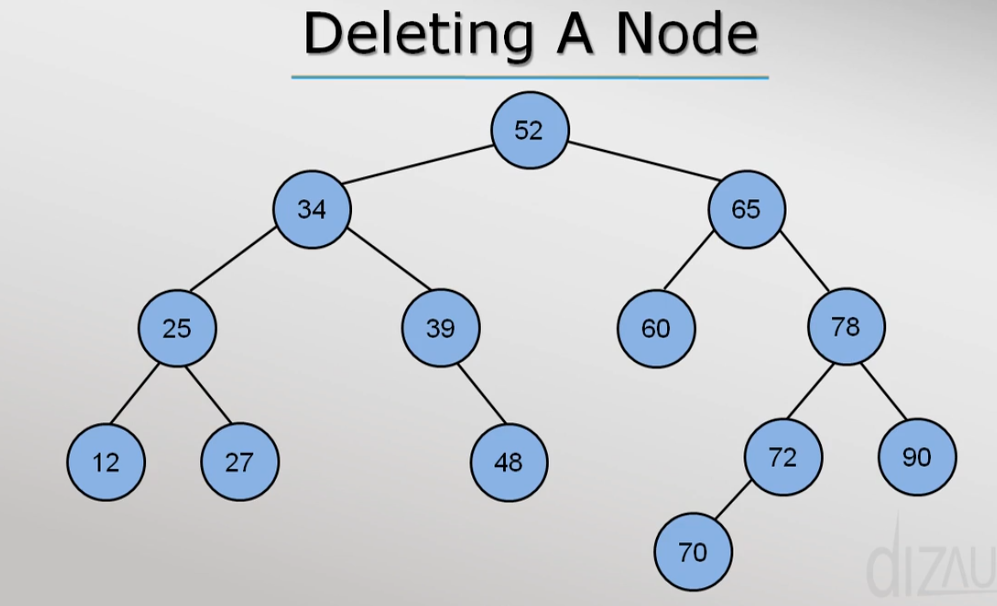
**Xóa một phần tử**

Với case 1 và case 2, khá đơn giản.

Với case 3, là node cần xóa có cả left và right child Node:

****

****

****

Cách xóa trên khá là phức tạp, do đó chúng ta thường sử dụng một phương pháp khác, đó là **Soft Delete**

**Soft Delete** là không hẳn xóa node đi, mà chúng ta sẽ đánh dấu là xóa, tuy nhiên mỗi khi chúng ta duyệt qua Tree, hoặc tìm kiếm một item, chúng ta sẽ không quan tâm đến phần tử bị đánh dấu là đã xóa này.

**--------------- --------------- --------------- ---------------**

**Duyệt cây nhị phân**

**In-Order traversal:**

* Duyệt left sub tree
* Visit the root
* Duyệt the right sub tree

**Pre-Order Traversal:**

* Visit the Root
* Duyệt sang left sub tree
* Duyệt sang right sub tree

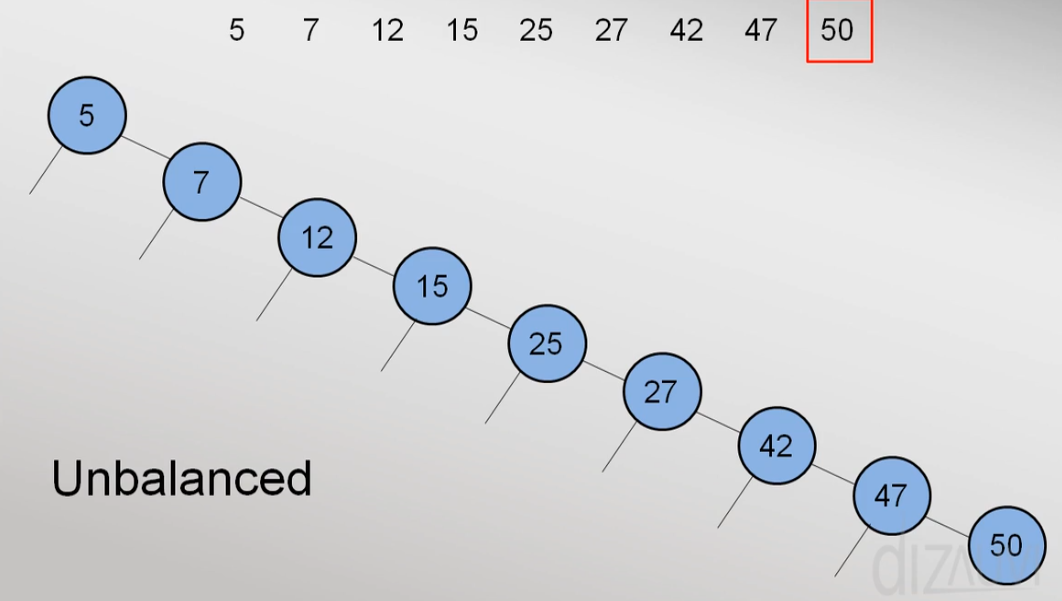
**Post-Order Traversal**

* Duyệt qua left sub tree
* Duyệt tiếp đến right sub tree
* Cuối cùng visit the Root

**--------------- --------------- --------------- ---------------**

**Unbalanced Trees & Balanced Tree**

Gỉa sử chúng ta liên tục thêm các phần tử có giá trị một cách tuần tự (tăng dần), thì khi đó các phần tử được thêm vào sẽ luôn luôn là phần tử right child node đối với parent. => Unbalanced, đây gần giống như là LinkedList, khi đó chúng ta sẽ không dùng được các đặc tính của một TreeNode nữa, khi đó nếu duyệt để tìm kiếm phần thử, ta sẽ có O(n) là độ phức tạp.

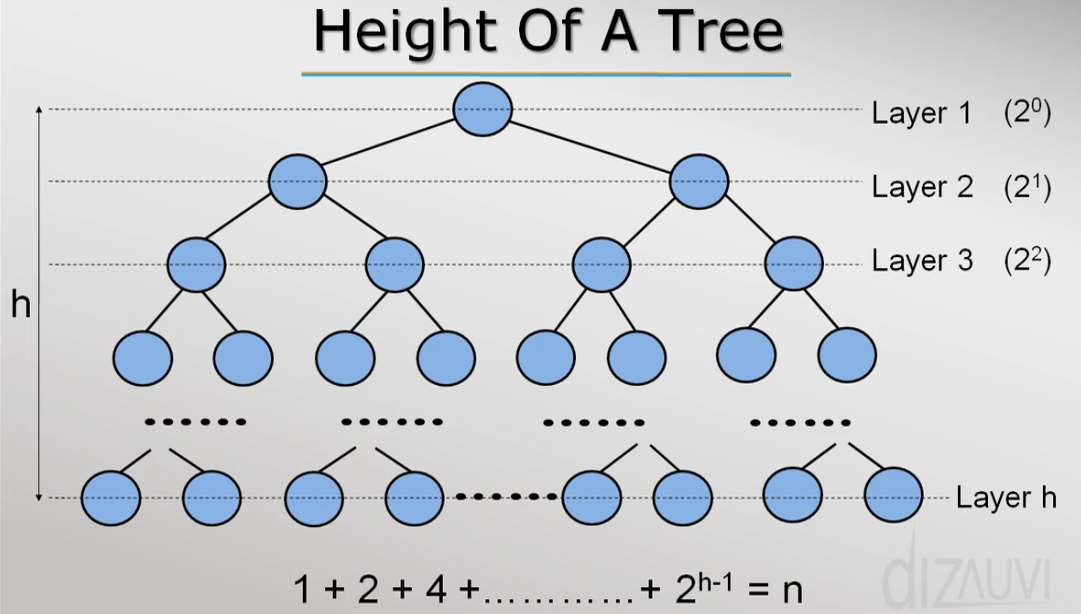


Các operations mà chúng ta thực hiện trên một Tree sẽ đạt hiệu quả tốt nhất trên một Balanced Tree, do đó lý tưởng nhất, chúng ta luôn luôn muốn có một balanced tree, tuy nhiên điều này phụ thuộc vào data mà chúng ta dùng để build nên Tree đó.

**--------------- --------------- --------------- ---------------**

**Height Of a Tree**

Giả sử chúng ta có một Balanced Tree, chúng ta sẽ xác định height của nó.



**--------------- --------------- --------------- ---------------**

**Time Complexity of Operations on Binary Search Tree**

Giả sử chúng ta xem xét trên một Balanced Search Tree, ta có n là số lượng các Node của Tree.

Search (item) = O(log2n), tương tự với Delete, vì chúng ta cũng phải tìm kiếm Node trước đó và thực hiện 1 vài tác vụ trên Node để thực hiên xóa Node đó đi, việc insert item cũng là tương tự.

**--------------- --------------- --------------- --------------- --------------- ---------------**

**Bài 7: đồ thị**

Là khái niệm về sự biểu diễn của các connections giữa các object, miêu tả rất nhiều sự vật, bối cảnh, các vấn đề trogn cuộc sống cũng như khoa học máy tính.

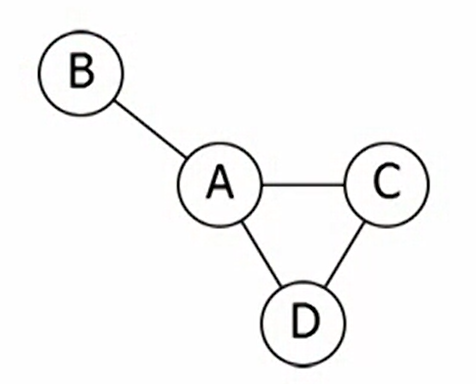
Graph: là tập hợp các các V (of vertices – các đỉnh), và các E (of Edge – các cạnh) sẽ kết nổi các đỉnh với nhau

**--------------- --------------- --------------- ---------------**

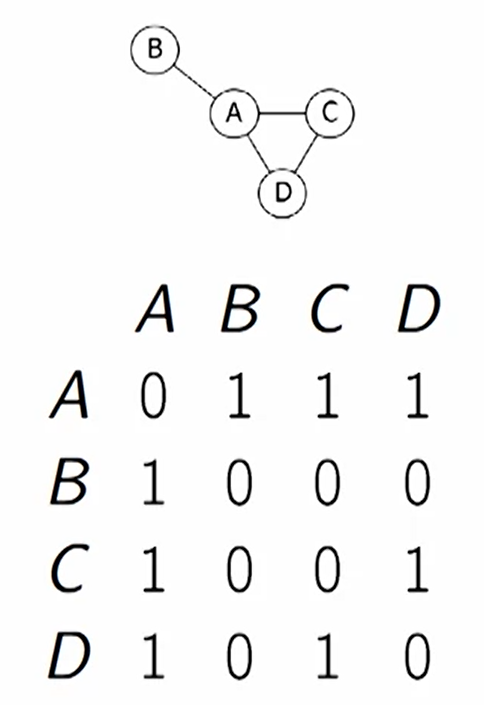
**Biểu diễn đồ thị**

Sử dụng Edge List:

* Chúng ta có các node: A, B, C, D
* Chúng ta có các cạnh: (A, B), (A, C), (A, D), (C, D)



Sử dụng ma trận gần kề: mỗi entries sẽ bằng 1 nếu như 2 edge có thể connect được đến nhau



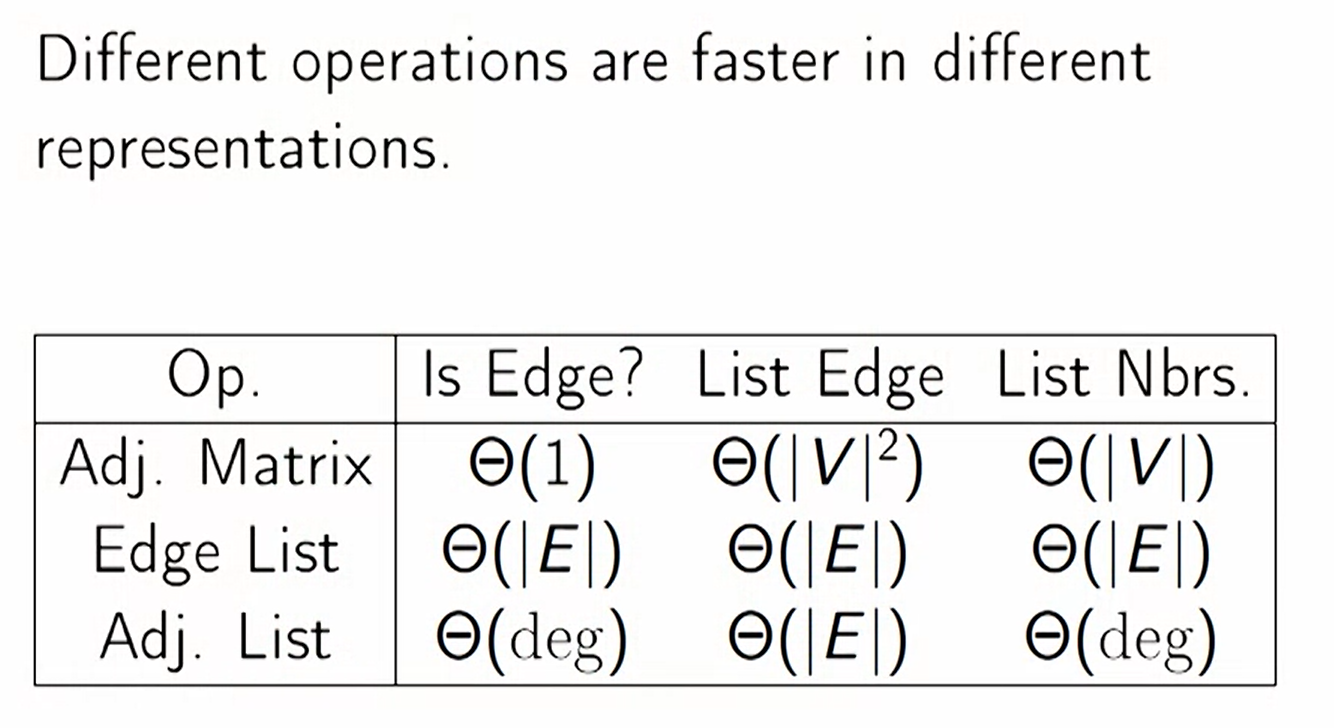
Sử dụng các danh sách liền kề: mỗi một đỉnh sẽ có một list các đỉnh mà nó có connect được đến, do đó ta sẽ có list như sau đối với ma trận ở trên:

A adjacent to B, C, D

B adjacent to A

C adjacent to A, D

D adjacent to A, C



Thời gian chạy của thuật toán Graph:

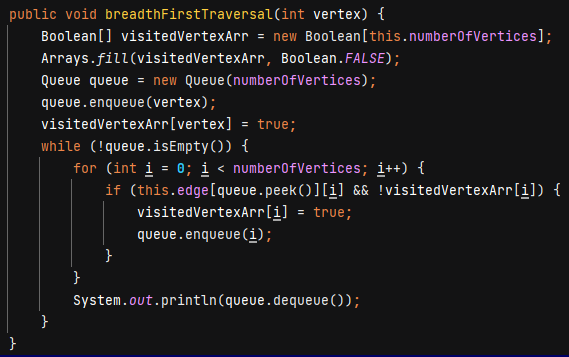
* phụ thuộc vào số lượng các Đỉnh |V| và các Cạnh |E|

**--------------- --------------- --------------- ---------------**

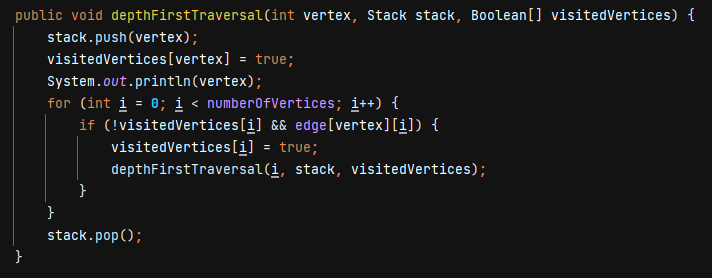
**Duyệt đồ thị theo chiều rộng**

Có 2 phương pháp để duyệt toàn bộ các đỉnh của một đồ thị:

1. Duyệt (tìm kiếm) theo chiều rộng (breadth first search): thuật toán BFS bắt đầu từ đỉnh V (được xác định trước) và lần lượt nhìn các đỉnh liền kề với đỉnh V. Sau đó với mỗi đỉnh trong số đó, thuật toán lại lần lượt nhìn các đỉnh liền kề với nó mà chưa được quan sát trước đó và lặp lại cho đến khi tất cả các đỉnh đều được quan sát.



1. Duyệt theo chiều sâu (depth first search): cũng bắt đầu tại một đỉnh V cho trước, quá trình tìm kiếm được phát triển tới đỉnh con đầu tiên của nút đang tìm kiếm và cứ như thế tiếp tục như thế cho tới khi gặp được đỉnh cần tìm hoặc tới 1 đỉnh không có con, khi đó giải thuật quay lui về đỉnh vừa mới tìm kiếm ở bước trước đó để tìm một khả năng khác.



**Đồ thị vô hướng**

Đồ thị mà trong đó tất cả các cạnh đều vô hướng (không có hướng) thì được gọi là đồ thị vô hướng, mỗi cạnh luôn là một mối quan hệ hai chiều và có thể duyệt theo hai hướng. Đồ thì có hướng là trường hợp ngược lại của đồ thị vô hướng, tức là từ một đỉnh A có thể đến đỉnh B, nhưng điều ngược lại chưa chắc xảy ra.

**Tinh liên thông**

Một đồ thị được gọi là liên thông nếu như mọi cặp đỉnh trong đồ thị đều có đường đi đến nhau, nếu tồn tại một cặp đỉnh không có đường đi đến nhau, ta gọi đồ thị đó là không liên thông.

**Giới thiệu về bài toán Định tuyến nhanh nhất**

Định tuyến là xác định tuyến đường đi theo một điều kiện nhất định

Khi định nghĩa về chiều dài đường đi sẽ có hai cách:

* Chiều dài đường đi chính là số các cạnh mà nó đi qua
* Chiều dài đường đi chính là tổng số trọng số của các đoạn mà nó đi qua, áp dụng với đồ thị có trọng số

**Thuật toán Dijkstra**

Khi đi từ một đỉnh A đến đỉnh B trong đồ thị, chúng ta có thể có rất nhiều đường đi, thường ra sẽ chọn đường đi có độ dài ngắn nhất. Thuật toán Dijkstra là một trong số những thuật toán có thể giải được bài toán này.

Ý tưởng của thuật toán này là lưu một tập hợp các vertices mà khoảng cách đã được đặt chính xác (known region)

Đỉnh đầu tiên được added vào R là S (source)

Sau mỗi lần lặp chúng ta sẽ lấy 1 đỉnh không thuộc R với dist-value nhỏ nhất, gán nó vào R, và tiếp tục relax những cạnh tiếp theo.

public void dijkstra(char v1**,** char v2) {  
 /\* Xác định điểm bắt đầu và kết thúc \*/  
 int startVertex = this.mapStringToIndex.get(v1)**;** int finalVertext = this.mapStringToIndex.get(v2)**;** /\* Tạo một array rỗng, tính toán đường đi ngắn nhất từ mọi đỉnh trong graph đến đỉnh v1. Mỗi phần tử sẽ store tổng trọng số của đường đi ngắn nhất và đỉnh trỏ đến nó ở ngay trước đó. Ban đầu ta sẽ gán trọng số là MAX\_INT, và tracked node trước đó là -1 \*/  
 TrackingNode[] trackingNodes = new TrackingNode[this.numberOfVertices]**;** for (int i = **0;** i < trackingNodes.length**;** i++) {  
 trackingNodes[i] = new TrackingNode()**;** trackingNodes[i].setWeight(Integer.*MAX\_VALUE*)**;** trackingNodes[i].setPreviousVertex(-**1**)**;** trackingNodes[i].setVisited(false)**;** }  
  
 trackingNodes[startVertex].setWeight(**0**)**;** trackingNodes[startVertex].setPreviousVertex(startVertex)**;** /\* thực hiện đi qua tất cả các đinh để tìm cách rút ngắn đường đi, loop đến khi nào tất cả các đỉnh đều đã được xét \*/  
 while(!*isVisitedALlVertex*(trackingNodes)) {  
 int minValue = Integer.*MAX\_VALUE***;** int minIndex = -**1;** for (int i = **0;** i < trackingNodes.length**;** i++) {  
 /\* Tìm đỉnh hiện tại có trọng số bé nhất để xét \*/  
 if (!trackingNodes[i].isVisited() && trackingNodes[i].getWeight() < minValue) {  
 minValue = trackingNodes[i].getWeight()**;** minIndex = i**;** }  
 }  
  
 /\* Sau khi đã tìm được đỉnh thích hợp để xét, bắt đầu thực hiện tìm đường đi từ đỉnh này đến các đỉnh còn lại để xem xem có thể rút ngắn được đường đi không \*/  
 for (int j = **0;** j < numberOfVertices**;** j++) {  
 if (edge[minIndex][j] > **0** && edge[minIndex][j] < *EDGE\_NOT\_FOUND\_WEIGHT* && (edge[minIndex][j] + trackingNodes[minIndex].getWeight() < trackingNodes[j].getWeight())) {  
 trackingNodes[j].setWeight(edge[minIndex][j] + trackingNodes[minIndex].getWeight())**;** trackingNodes[j].setPreviousVertex(minIndex)**;** }  
 }  
  
 /\* Đánh dấu node này đã được duyệt \*/  
 trackingNodes[minIndex].setVisited(true)**;** }  
  
 /\* Kết thúc thuật toán, chúng ta có được mảng trackingNodes lưu trọng số của đường đi ngắn nhất từ đỉnh v1 tới các đỉnh còn lại, và đỉnh trước đó để đến mỗi đỉnh, giờ ta sẽ tracking ngược lại từ đỉnh v2 về đỉnh v1, ta sẽ dùng stack vì chúng ta sẽ duyệt ngược lại \*/  
 Stack result = new Stack(numberOfVertices)**;** int currentVertexIndex = finalVertext**;** while (currentVertexIndex != startVertex) {  
 result.push(currentVertexIndex)**;** currentVertexIndex = trackingNodes[currentVertexIndex].getPreviousVertex()**;** }  
 result.push(startVertex)**;** while (!result.isEmpty()) {  
 System.*out*.print(this.mapIndexToString.get(result.pop()) + " ")**;** }  
 System.*out*.println()**;** System.*out*.println("Total weighted of shortest path from " + v1 + " to " + v2 + ": " + trackingNodes[this.mapStringToIndex.get(v2)].getWeight())**;** System.*out*.println("Dijkstra algorithm for shortest path from " + v1 + " to other vertex:")**;** for (int i = **0;** i < trackingNodes.length**;** i++) {  
 System.*out*.println("From " + v1 + " to " + this.mapIndexToString.get(i) + " is " + trackingNodes[i].getWeight())**;** }  
}

**Cây khung nhỏ nhất**

**Khái niệm:** cây khung là một tập con của đồ thị G, trong đó tất cả các đỉnh được phủ bởi số cạnh tối thiểu có thể, do đó 1 cây khung không có các kết nối vòng và không thể thuộc kiểu bất liên thông

**Giới thiệu về thuật toán tham lam**

**Ý tưởng của thuật toán Kruskal:**

tìm cây khung nhỏ nhất nhưng không phụ thuộc vào điểm bắt đầu để tìm cây khung nhỏ nhất của một đồ thị liên thông có trọng số, nói cách khác là nó tìm một cây chứa tất cả các đỉnh (đi qua tất cả các đỉnh) và có tổng trọng số các cạnh là bé nhất.

Cách thực hiện thuật toán **Kruskal:**

* Liệt kê các cạnh theo thứ tự tăng dần
* Thực hiện thêm vào các cạnh nhưng đảm bảo để không tạo thành 1 chu trình kín
* Thực hiện liên tục cho đến khi nào tất cả các đỉnh đều được đi qua

**Ý tưởng của thuật toán Prim:** tương tự với Kruskal, thuật toán Prim là một thuật toán tham lam để tìm cây trùm nhỏ nhất của một đồ thị vô hướng có trọng số liên thông, với mục đích cuối cùng tương tự với **thuật toán Kruskal**

Cách thực hiện thuật toán **Prim:**

* Gần tương tự dijitra

**--------------- --------------- --------------- --------------- --------------- --------------- --------------- --------------- --------------- --------------- --------------- ---------------**

**Phần 4: Các phương pháp tìm kiếm và sắp xếp nâng cao**

**Bài 8: các thuật toán nâng cao**

**Thuật toán tìm kiếm nhanh (Quick Sort)**

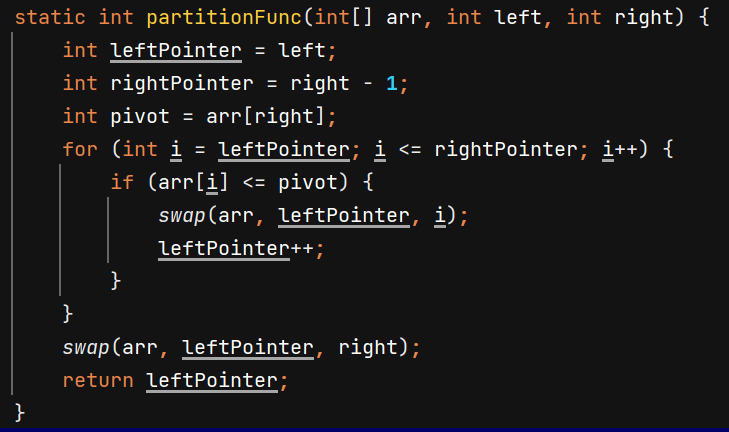
Thuật toán Quick Sort là một thuật toán sắp xếp nhanh trên một danh sách cho trước, gồm hai giai đoạn:

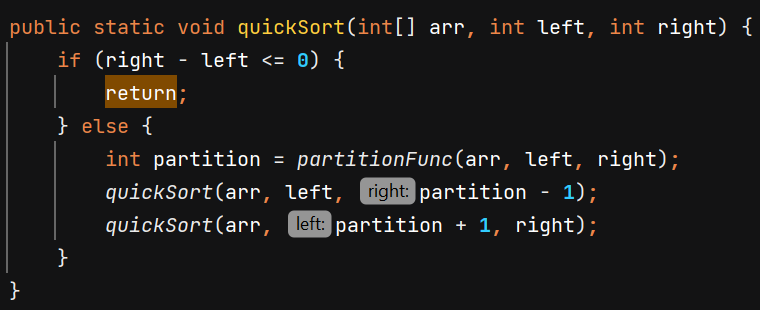
Giai đoạn một chia nhỏ danh sách bằng cách so sánh từng phần tử của danh sách với một phần tử được chọn, những phần tử nhỏ hơn hoặc bằng phần tử chốt được đưa về phía trước và nằm trong danh sách con thứ nhất, các phần tử lớn hơn chốt được đưa về phía sau. Cứ tiếp tục chia như vậy tới khi các danh sách con có độ dài bằng 1. Đây là một thuật toán In Place Sorting.

Worst Case: O (n2)

Average Case: O (n log n)

Mấu chốt của thuật toán Quick sort là việc phân đoạn dãy số. Mục tiêu của công việc này là: cho một mảng và một phần tử X là pivot, đặt X vào đúng vị trí của mảng đã sắp xếp, di chuyển tất cả phần tử của mảng mà nhỏ hơn X sang bên trái vị trí của X, và di chuyển tất cả các phần tử của mảng mà lớn hơn X sang bên phải của X. Khi đó ta sẽ có 2 mảng con, mảng bên phải và mảng bên trái của X, tiếp tục công việc với mỗi mảng con (chọn Pivot, phân đoạn) cho tới hi được mảng sắp xếp.





**--------------- --------------- --------------- ---------------**

**Thuật toán Shell Sort**

Shell Sort là một giải thuật sắp xếp dựa trên sắp xếp chèn, giải thuật này tránh được các trường hợp phải tráo đổi vị trí của hai phần tử quá xa nhau trong giải thuật sắp xếp chọn.

<https://viettuts.vn/cau-truc-du-lieu-va-giai-thuat/giai-thuat-sap-xep-shell-sort>

public static void shellSort(int[] arr) {  
 int interval = **0;** /\*  
 Tính giá trị khoảng (interval)  
 \*/  
 while (interval < arr.length / **3**) {  
 interval = interval \* **3** + **1;** }  
  
 while (interval > **0**) {  
 for (int outer = interval**;** outer < arr.length**;** outer++) {  
 /\* chọn giá trị để chèn \*/  
 int valueToInsert = arr[outer]**;** int inner = outer**;** /\* Dịch chuyển phần tử sang phải \*/  
 while (inner > interval - **1** && arr[inner - interval] >= valueToInsert) {  
 arr[inner] = arr[inner - interval]**;** inner = inner - interval**;** }  
 /\* Chèn giá trị vào vị trí trên \*/  
 arr[inner] = valueToInsert**;** }  
 /\* Tinhs toán lại giá trị của interval \*/  
 interval = (interval - **1**) / **3;** }  
}

**--------------- --------------- --------------- ---------------**

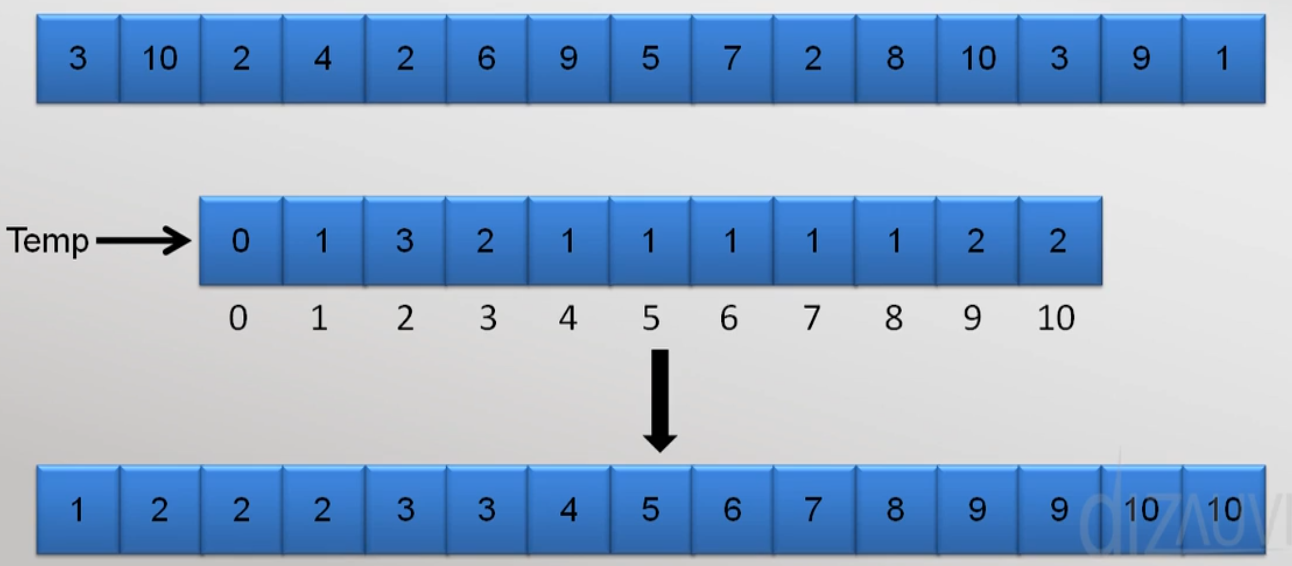
**Thuật toán sắp xếp đếm**

Đây là thuật toán sắp xếp đếm số lần xuất hiện khác nhau của các giá trị khác nhau trong mảng ban đầu và gán giá trị vừa đếm vào vị trí tương ứng của một mảng có độ dài là giá trị lớn nhất của mảng. Thuật toán này hoạt động hiệu quả dựa trên việc chúng ta biết range của array mà chúng ta muốn sort, và cách chúng ta sort sẽ minh họa như sau:









Thuật toán này tương đối dễ hiểu và dễ thực hiện.

**--------------- --------------- --------------- ---------------**

**Thuật toán sắp xếp theo cơ số (Radix Sort)**

**--------------- --------------- --------------- ---------------**

**Thuật toán sắp xếp trộn (merged sort)**

Thuật toán Merged Sort và quicksort là 2 thuật toán dựa vào tư tưởng chia để trị. Thủ tục cơ bản là việc trộn 2 danh sách đã được sắp xếp thành một danh sách mới theo thứ tự. Độ phức tạp của thuật toán là **Ο(n log n).** Đây là một trong những thuật toán sắp xếp rất tốt. Để hiểu về merged sort, chúng ta sẽ bắt đầu với một mảng chưa sort



Chúng ta biết rằng merged sort sẽ chia mảng thành các mảng con có số phần tử bằng nhau, cho đến khi mảng đó chỉ còn 1 phần tử





Bây giờ chúng ta sẽ “ghép” các phần tử tương tự như lúc chúng ta tách chúng ra. Bắt đầu bằng việc so sánh element của mỗi list và combine chúng thành một mảng được sort. Dễ dàng nhận thấy 14 và 33 đã được sorted, so sánh 27 và 10, chúng ta đưa 10 ra trước 7, tương tự đối với 35 và 19



Ở lần lặp tiếp theo của việc combinem so sánh các list array có 2 data values, và thực hiện merge chúng theo thứ tự đã sort



Mảng kết quả sẽ như sau:



Implement thuật toán:

/\* Method to merged 2 arrays \*/  
static int[] merge(int[] left**,** int[] right) {  
 int[] result = new int[left.length + right.length]**;** Arrays.*fill*(result**,** Integer.*MAX\_VALUE*)**;** while (left[**0**] != Integer.*MAX\_VALUE* && right[**0**] != Integer.*MAX\_VALUE*) {  
 if (left[**0**] > right[**0**]) {  
 *addItem*(result**,** right[**0**])**;** *removeFirstItem*(right)**;** } else {  
 *addItem*(result**,** left[**0**])**;** *removeFirstItem*(left)**;** }  
 }  
  
 while (left[**0**] != Integer.*MAX\_VALUE*) {  
 *addItem*(result**,** left[**0**])**;** *removeFirstItem*(left)**;** }  
  
 while (right[**0**] != Integer.*MAX\_VALUE*) {  
 *addItem*(result**,** right[**0**])**;** *removeFirstItem*(right)**;** }  
 return result**;**}

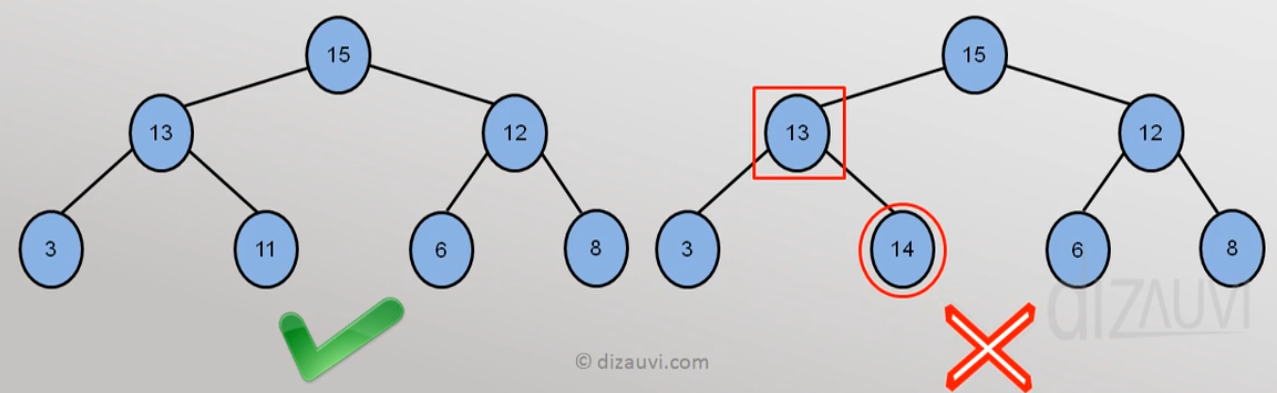
public static int[] mergeSort(int[] arr) {  
 if (arr.length == **1**) {  
 return arr**;** }  
 int[] leftArr**,** rightArr**;** if (arr.length > **2**) {  
 leftArr = Arrays.*copyOfRange*(arr**, 0,** arr.length / **2** + **1**)**;** rightArr = Arrays.*copyOfRange*(arr**,** arr.length / **2** + **1,** arr.length)**;** } else {  
 leftArr = Arrays.*copyOfRange*(arr**, 0, 1**)**;** rightArr = Arrays.*copyOfRange*(arr**, 1, 2**)**;** }  
 leftArr = *mergeSort*(leftArr)**;** rightArr = *mergeSort*(rightArr)**;** return *merge*(leftArr**,** rightArr)**;**}

**--------------- --------------- --------------- --------------- --------------- --------------- --------------- --------------- --------------- --------------- --------------- ---------------**

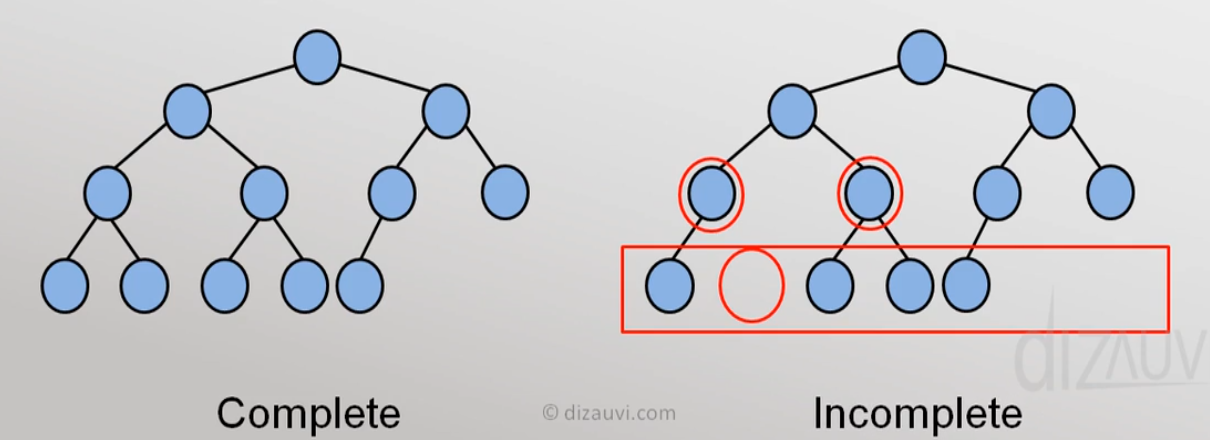
**Bài 9: Heap**

**Giới thiệu về Heap**

**Heap** là một cấu trúc dữ liệu dựa trên Tree thỏaa mãn tính chất Heap, là một trường hợp của cây nhị phân thỏa mãn: nếu B là nút con của A thì giá trị của khóa A **phải lớn hơn bằng bằng** giá trị của khóa B, và hệ quả của tính chất này đó là giá trị khóa lớn nhất luôn luôn nằm ở nút gốc, một Heap như vậy được gọi là **Max Heap.** Nếu mọi phép so sánh bị đảo ngược khiến cho khóa có giá trị nhỏ nhất luôn nằm ở nút gốc thì Heap đó được gọi là **Min Heap.**

****

Một Heap phải đảm bảo là complete:



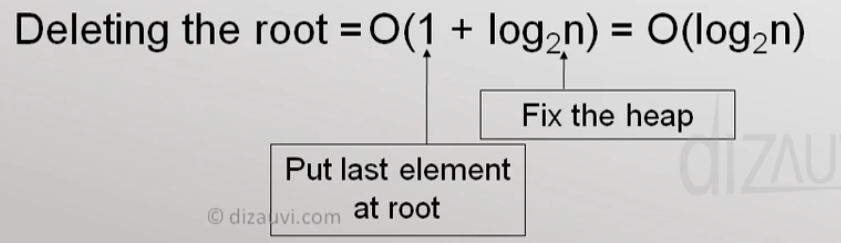
**Heap** có vai trò quan trọng trong nhiều thuật toán cho đồ thị chẳng hạn như Dijkstra hay thuật toán sắp xếp Heap Sort

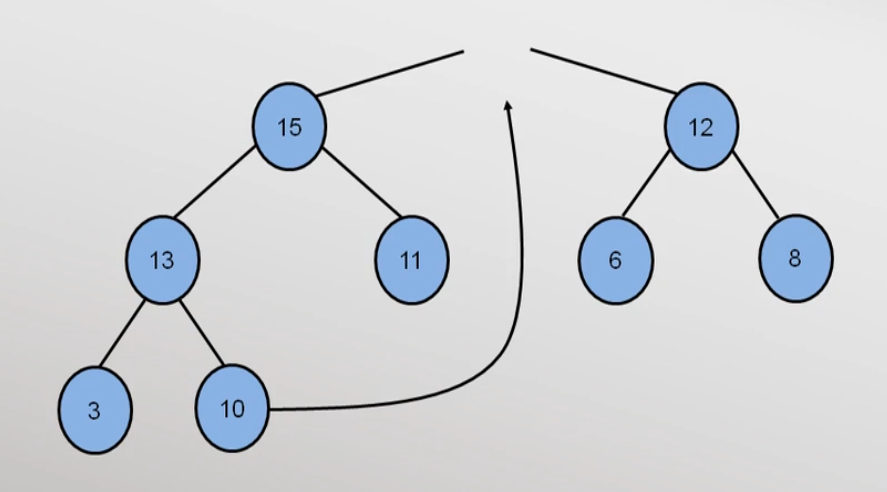
**--------------- --------------- --------------- ---------------**

**Xóa Root node**

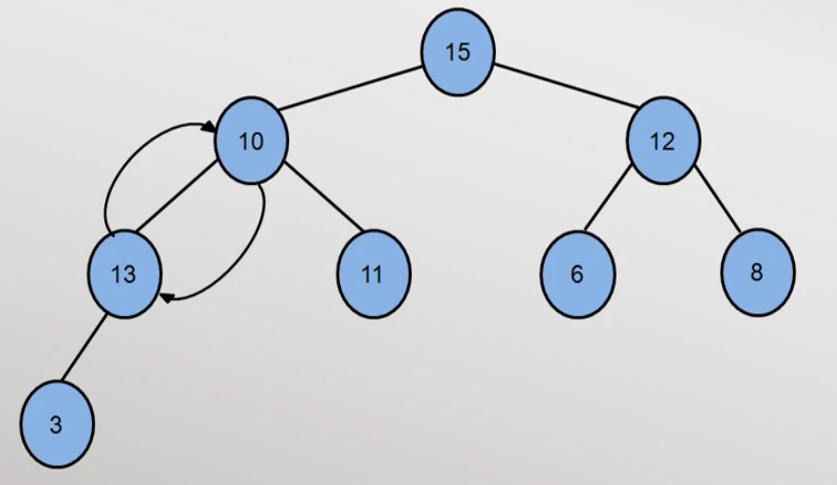
Đây là một operator chúng ta hay thực hiện trên Heap, tương tự như chúng ta làm với **Queue** với chức năng dequeue

Độ phức tạp của thuật toán deleting the Root: O (log2n)





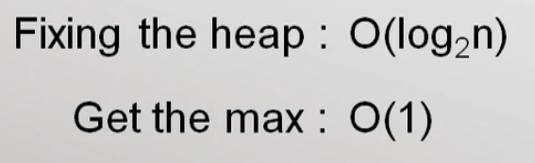




**--------------- --------------- --------------- ---------------**

**Thêm mới một Node**

Việc thêm mới một node, chúng ta vẫn phải đảm bảo Heap của chúng ta complete, do đó sau khi thêm Node mới vào vị trí để đảm bảo tính complete của Heap, nếu như tính chất của Heap bị phá vỡ, ta phải tiếp tục “bubble” Node mới này để đưa nó về vị trí thích hợp để đảm bảo tính chất của Heap. Chúng ta gọi operation này là **fixing the heap.**



**--------------- --------------- --------------- --------------- --------------- --------------- --------------- --------------- --------------- --------------- --------------- ---------------**

**Bài 10: HashTable và Hashing**

**Bảng băm** là một cấu trúc dữ liệu theo cách thức liên hợp, trong một bảng băm dữ liệu sẽ được lưu trữ theo dạng mảng, mỗi giá trị dữ liệu sẽ có một chỉ số riêng **index** khác nhau. Việc truy cập sẽ rất nhanh nếu như chúng ta biết được khóa của dữ liệu

Hashing là một kỹ thuật dùng để xác định duy nhất một đối tượng cụ thể từ một nhóm các đối tượng tương tự nó, ví du:

* Mỗi sinh viên trong trường được gán với một số ID, từ ID này có thể truy cập để tìm kiếm các thông tin liên quan tói sinh viên
* Mỗi đầu sách trong thư viện được gán cho một ID duy nhất, từ ID này có thể biết được vị trí chính xác của nó trong thư viện hoặc người dùng đã mượn nó

Trong các ví dụ trên, sách và sinh viên đã được mã hóa với một ID duy nhất, đó gọi là kỹ thuật Hashing

Như vậy ta cũng có thể định nghĩa Hashing là kỹ thuật biến đổi một khóa (key) trở thành một số nguyên bằng cách dùng các công thức toán học, công thức toán học này được thực hiện trong một hàm gọi là hash function (hàm băm). Một key sau khi xử lý của hàm băm sẽ sinh ra một số nguyên duy nhất, được gọi là mã băm (hash code)

Hashing có thể được thực hiện trong hai bước

1. Chuyển đổi khóa (key) thành số nguyên (khóa nhỏ hơn) dùng hàm băm, khóa sau khi được chuyển đổi có thể được dùng như chỉ số để chứa phần tử ban đầu nằm trong mảng băm
2. Phần tử được chứa trong mảng băm có thể được truy cập nhanh chóng qua khóa đã được chuyển đổi.

Bảng băm là một kiểu cấu trúc dữ liệu lưu giữ dữ liệu theo cách thức liên hợp, dưới dạng cặp (key, value). Nó sử dụng hàm băm để tính toán chỉ số thông qua key, chỉ số này được dùng cho việc chèn hoặc tìm kiếm dữ liệu dễ dàng hơn. Với một bảng băm có một hàm băm được thực hiện tốt thì trong trường hợp lý tưởng, việc tìm kiếm các phần tử trong bảng có thời gian là O(1)

Bởi vì hàm băm trả về một key có value < khóa gốc (là một số nguyên lớn hoặc một chuỗi dài), dẫn đến có thể có 2 khóa có thể có cùng một giá trị. Tình huống là một phần tử mới được thêm vào bảng băm có khóa ánh xạ với một vị trí đã có dữ liệu, gọi là sự xung đột. Chúng ta sẽ phải giải quyết các xung đột này bằng các cách phổ biến sau:

1. Phương pháp kết nối (chaining)
2. Phương pháp open addressing

Một hàm băm tốt nếu đạt được các yêu cầu cơ bản sau:

1. Dễ tính toán
2. Phân bố đồng đều
3. Ít xảy ra va chạm

**--------------- --------------- --------------- ---------------**

**Giải quyết xung đột (collision)**

Gọi h(k) là mộthàm băm, khi đưa thêm value vào hash table, nếu ta có một hash function và các key A và B khác nhau nhưng h(A) = h(B), rõ ràng lúc này 2 value sẽ được đưa tới cùng một vị trí, đây gọi là đụng độ (collision).

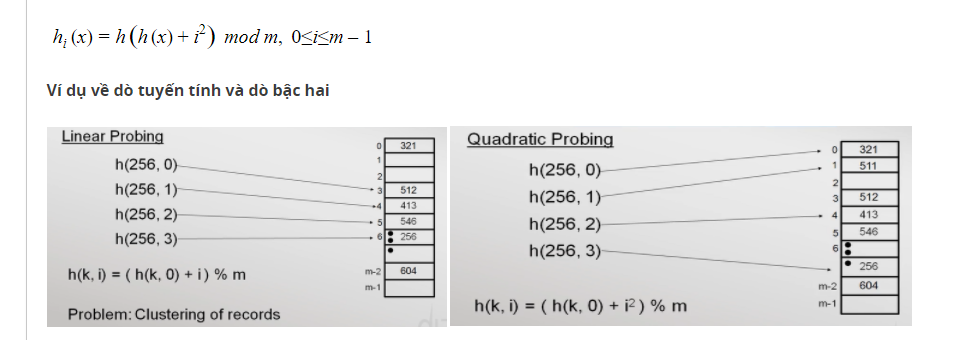
Giải quyết bằng chaining: thay vì chúng ta chèn trực tiếp vào một vị trí trong bảng, chúng ta sẽ coi mỗi vị trí là một linked list, và chúng ta sẽ chèn data mới vào linked list đó. Trường hợp worst case là khi tất cả các khóa đều ánh xạ đến một vị trí, đó là khi áp dụng hàm băm cho bất kỳ khóa nào ta cũng đều thu về cùng một key. Như vậy mọi bản ghi sẽ được lưu trữ trong một linked list, do đó thời gian để tìm một bản ghi sẽ là O(n).

**--------------- --------------- --------------- ---------------**

**Giải quyết xung đột bằng Open Addressing**

Dò tuyến tính

Dò theo hàm bậc 2:



Dò nhị phân

Băm kép