Comment représenter un graphe à l'aide de vector(s)

Buts:

- Comparer les forces/faiblesses de différents choix pour représenter les liens entre les noeuds d'un graphe

Plan:

- Rappel sur la mise en oeuvre de vector (sem1 Topic12)
- Représentation d'un graphe par un ensemble de Noeuds
- Liens: 1ière stratégie avec des pointeurs (catastrophique)
- Liens: Mise en oeuvre avec des indices (gérable mais gros risques d'erreur)
- Liens: 2ième stratégie avec des pointeurs (gérable)

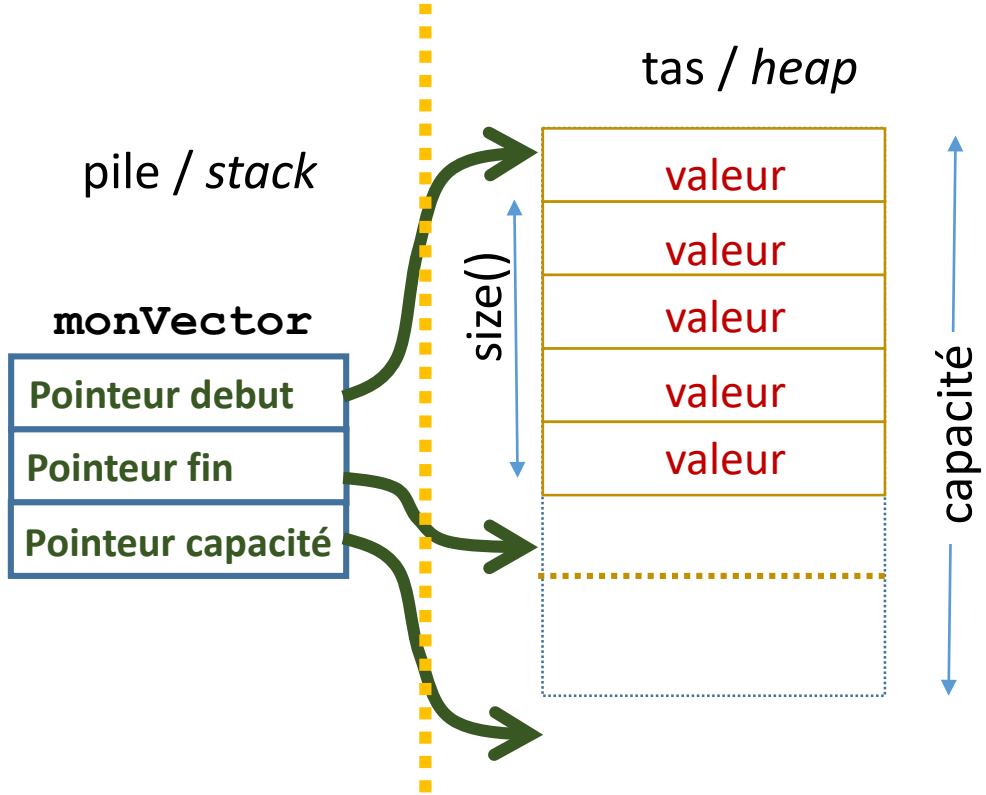
Rappel sur la mise en oeuvre de **vector** (sem1 Topic12)

Rappel : l'opérateur **sizeof(`type`)** appliqué à un **`type`** de donnée nous fournit le nombre d'octets occupés par une variable de ce **`type`** en mémoire.

Observation intéressante: **sizeof(vector<`type`>)** renvoie toujours le même nombre d'octets quel que soit le **`type`** de donnée. De même, **sizeof(`variable`),** où **`variable`** est une instance de **vector** d'un type donnée, donne toujours le même nombre d'octets quel que soit le nombre d'éléments du **vector**.

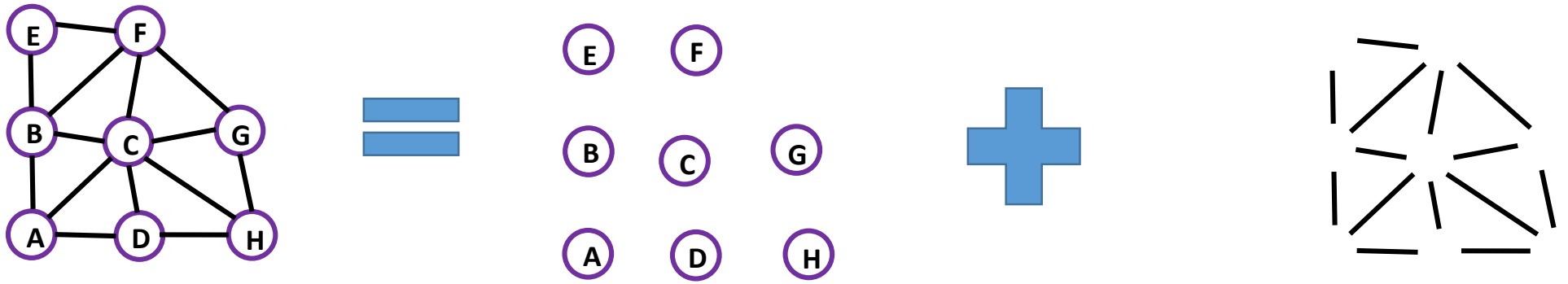
Comment est-ce possible ? L'implémentation de **vector** n'est pas documentée mais de nombreux ouvrages suggèrent qu'une instance de **vector** contient seulement *3 pointeurs*. Ces pointeurs mémorisent les adresses suivantes du tas (*heap*):

- Le **début des données**
- Juste après la **fin des données**
- Juste après la fin de la **capacité** réservée



Représentation d'un graphe par un ensemble d'instance de Noeuds

Un graphe est représenté au moins par un **ensemble de nœud** et par la **connectivité** de ces noeuds



Chaque nœud mémorise des *attributs spécifiques à l'application*

- Cet ensemble est *dynamique* car on veut pouvoir ajouter/enlever des nœuds.
- On pose aussi qu'il n'a pas besoin d'être trié

1ière conclusion: un **vector** convient parfaitement pour mémoriser **l'ensemble des noeuds.**

Il existe deux approches pour représenter la **connectivité** du graphe (l'information des *nœuds voisins*). Supposons qu'on ait N nœuds:

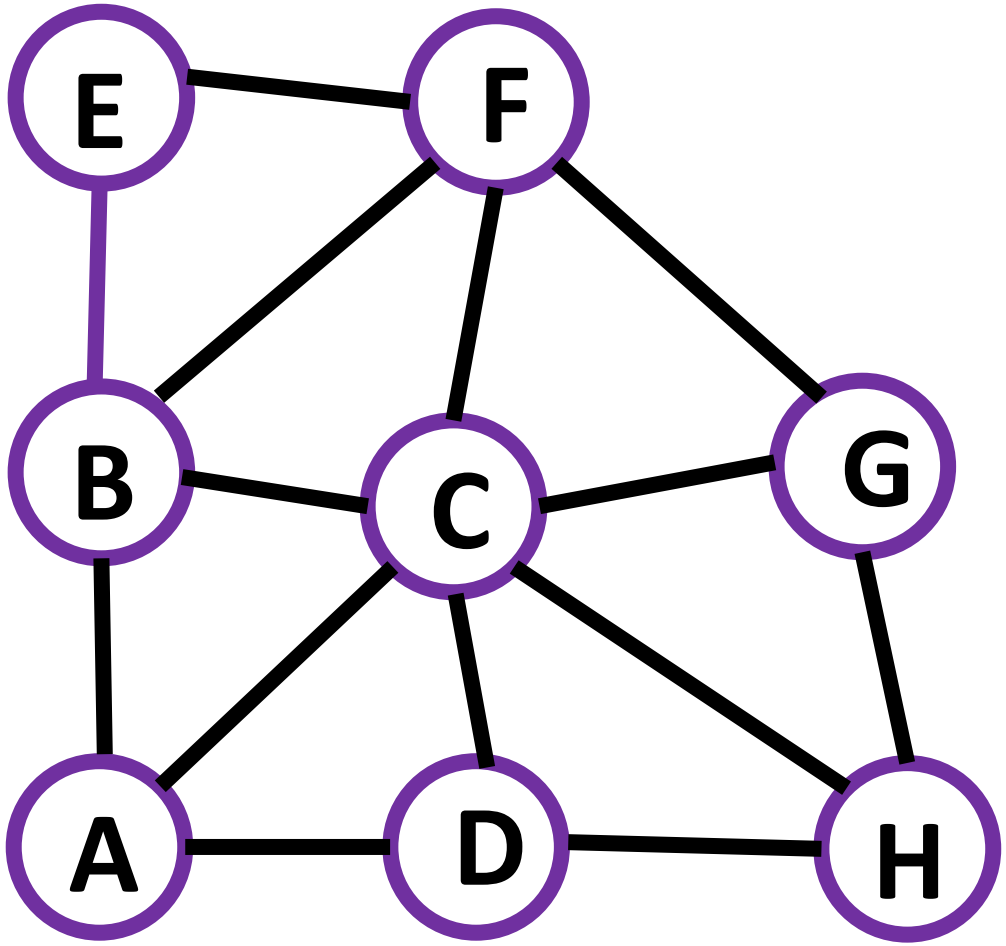
1. Une matrice NxN de booléens est trop coûteuse en mémoire
2. Nous choisissons de construire une **liste d'adjacence** *= les voisins* pour chaque nœud.

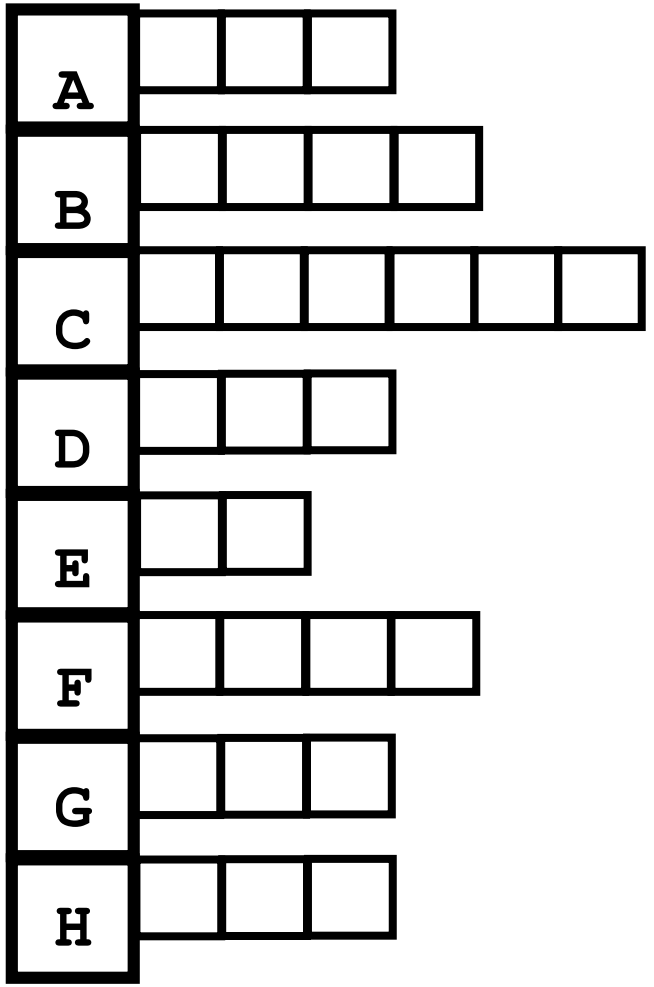
2ième conclusion: un **vector** convient parfaitement pour mémoriser **chaque liste d'adjacence.**

3ième conclusion: chaque nœud mémorise aussi un **vector** pour sa **liste d'adjacence.**

Représentation par vector de noeuds, chaque noeud ayant un vector de liens (liste d'adjacence)

*un nœud = attributs dont **1 liste d'adjacence = 1 vector***





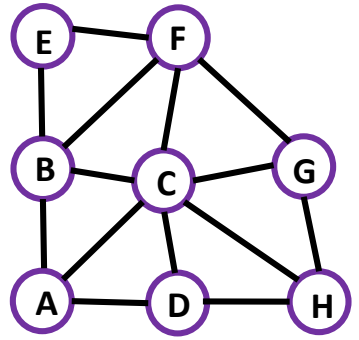
Quelle est la nature de l'information définissant un lien vers un nœud voisin ?

Liens: 1ière stratégie avec des pointeurs (catastrophique)

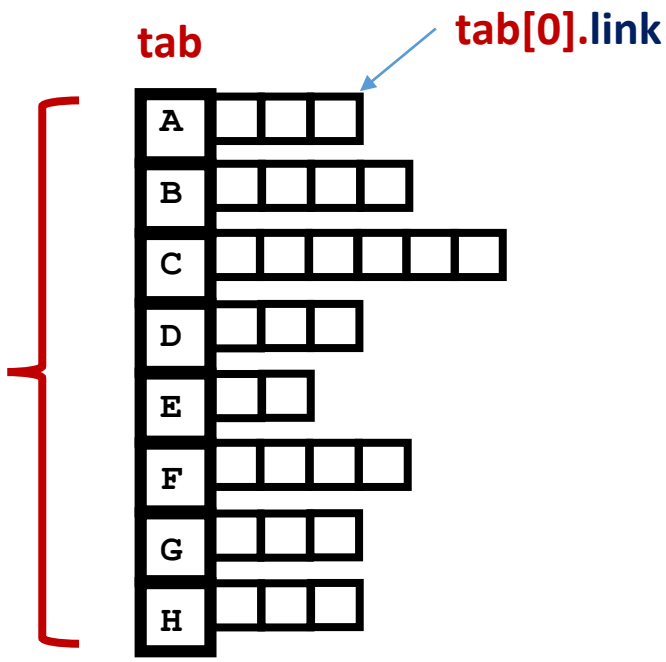
1ière stratégie: (cours du sem1) Pourquoi pas un pointeur vers un nœud ?

Supposons que nous disposons de la classe **Nœud** qui mémorise

1. les attributs du nœud
2. dont un **vector<Nœud*> link** des pointeurs vers les nœuds voisins



un **vector<Nœud> tab** mémorise l'ensemble des nœuds.



Chaque élément du vector **link** d'un nœud mémorise l'adresse des nœuds voisin.
Par exemple pour le nœud **D**,
dont les nœuds voisins sont **A**, **C** et **H**:

```
tab[3].link[0] = &tab[0]; //adresse de A
tab[3].link[1] = &tab[2]; //adresse de C
tab[3].link[2] = &tab[7]; //adresse de H
```

Question: pourquoi ce type de lien va-t-il produire des bugs à plus ou moins long terme ?

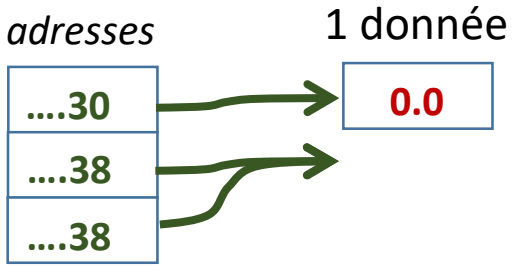
Question: pourquoi ce type de lien va-t-il produire des bugs à plus ou moins long terme ? (suite)

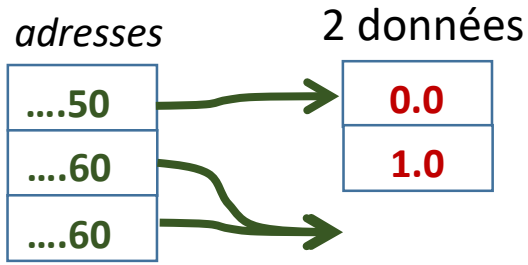
Examinons cet exemple d'exécution d'un programme qui agrandit plusieurs fois un **vector** avec **push_back()** et qui examine où sont rangées les données en mémoire.

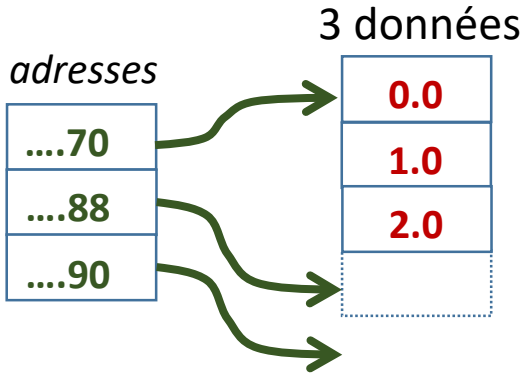
Init vide

nullptr
nullptr
nullptr

Revoir le dessin du slide 2 pour avoir la signification des 3 adresses







En examinant attentivement les groupes de 3 adresses on constate que le bloc des donnée est déplacé plusieurs fois !

Conséquence: les adresses précédentes des données sont toutes **invalides** !!!

Un tel choix pour mémoriser des liens va créer des ***bugs dès le premier déplacement des données sur le tas.***

```
0x7ffc7c16aff0
0
0
0
0x153d030
0x153d038
0x153d038

0x153d050
0x153d060
0x153d060

0x153d070
0x153d088
0x153d090

0x153d070
0x153d090
0x153d090

0x153d0a0
0x153d0c8
0x153d0e0

0x153d0a0
0x153d0d0
0x153d0e0

0x153d0a0
0x153d0d8
0x153d0e0

0x153d0a0
0x153d0e0
0x153d0e0

0x153d0f0
0x153d138
0x153d170

0x153d0f0
0x153d140
0x153d170
```

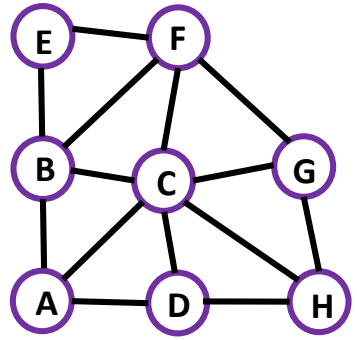
Performances : Déclarer le **vector** avec sa **dimension** dès qu'elle est connue à l'exécution *puis travailler avec des indices sur l'espace réservé* au lieu de faire de nombreux push_back.

Liens: Mise en oeuvre avec des indices (gérable mais gros risques d'erreur)

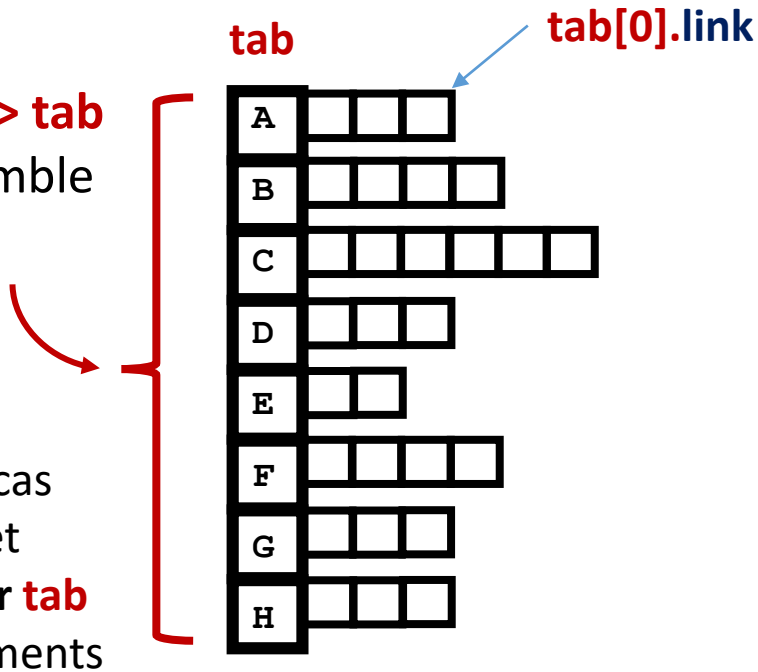
motivation: un indice d'élément de vector peut faire office de lien

La définition de la classe **Nœud** change légèrement:

1. les attributs du nœud
2. dont un **vector<unsigned> link** des indices des nœuds voisins



un **vector<Nœud> tab** mémorise l'ensemble des nœuds.



Pas de problème en cas d'agrandissement et déplacement du **vector tab** car la valeur de ses éléments reste inchangée.

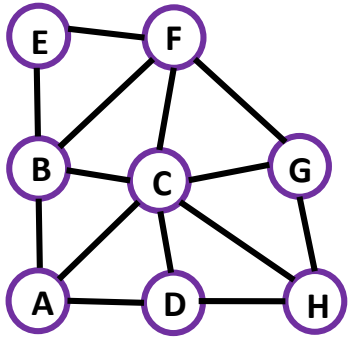
Chaque élément du vector **link** d'un nœud mémorise l'indice des nœuds voisin.
Par exemple pour le nœud **D**,
dont les nœuds voisins sont **A**, **C** et **H**:

```
tab[3].link[0] = 0; // indice de A
tab[3].link[1] = 2; // indice de C
tab[3].link[2] = 7; // indice de H
```

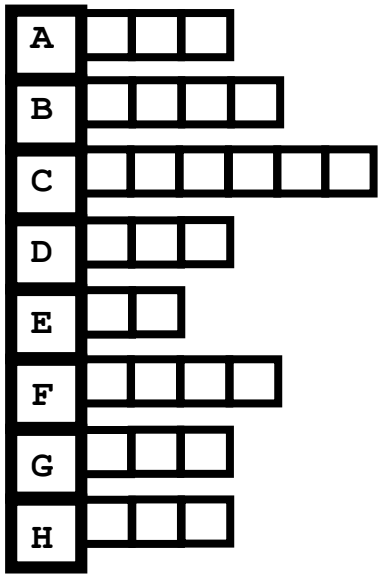
Question: dans quel contexte ce type de lien demande-t-il une extrême rigueur dans la mise à jour du graphe?

Question: dans quel contexte ce type de lien demande-t-il une extrême rigueur dans la mise à jour du graphe?
(suite)

AVANT



tab



Scénario: examinons ce qui se passe lorsqu'on détruit un élément du **vector** tab, par exemple le nœud **D**.

Sachant que ce **vector** n'a pas besoin d'être trié nous adoptons l'approche à coût constant qui échange le nœud **D** à détruire avec le dernier nœud **H** du **vector** puis qui appelle **pop_back().**

Question: est-ce suffisant de mettre à jour les listes d'adjacence des 3 voisins de **D**, c'est-à-dire pour **A**, **C** et **H** ?

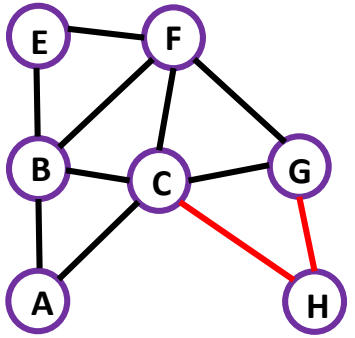
Non car en déplaçant **H** avec l'échange nous avons invalidé son ancien indice qui est mémorisé dans les listes d'adjacences des voisins (restants) de **H**, c'est-à-dire pour **C** et **G**.

Cette seconde classe de mise à jour ne doit pas être oubliée.

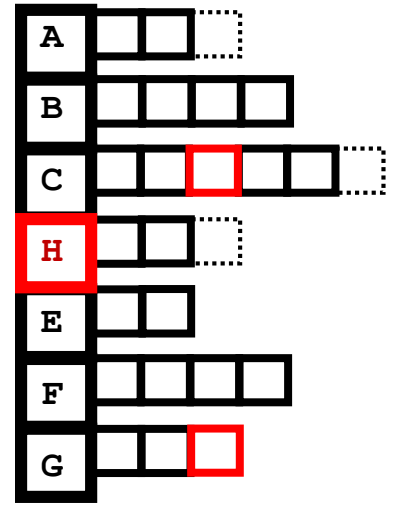
Alternative: on pourrait adopter une autre approche pour la destruction d'un nœud qui n'enlève pas le nœud détruit MAIS qui le marque comme *inactif*. Il n'y a pas d'échange ni d'emplacement libéré dans le vector dans ce cas. La seconde classe de mise à jour n'a plus lieu d'être.

Cependant cette alternative induit un surcoût d'occupation mémoire.

APRES



tab



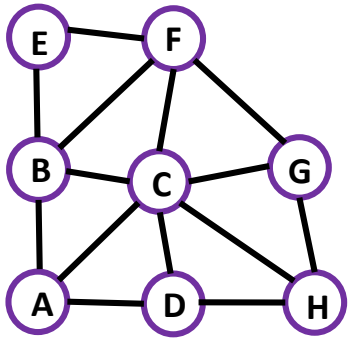
Liens: 2ième stratégie avec des pointeurs (gérable)

2ième stratégie: un lien est représenté par un pointeur vers un nœud

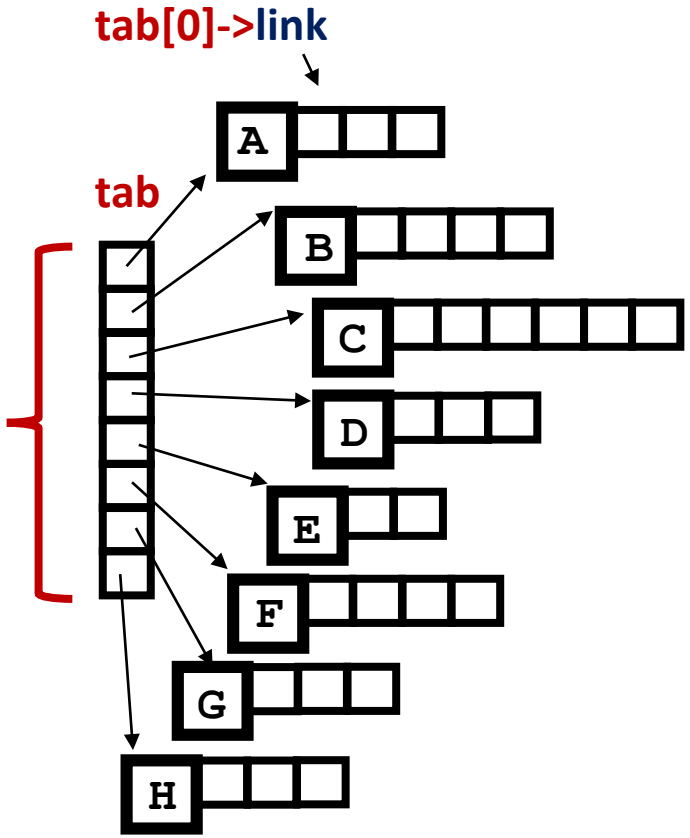
La définition de la classe **Nœud** reste la même qu'avec les pointeurs.

Ce qui change c'est la manière de créer les nœuds :

- Chaque nœud est **alloué dynamiquement** avec **`new`**
- C'est ***l'adresse*** ainsi obtenue qui est mémorisée dans **tab**



Le **vector<Nœud*> tab** mémorise l'ensemble des **pointeurs vers les nœuds** alloués dynamiquement.



Pas de problème en cas d'agrandissement et déplacement du **vector tab** car la valeur de ses éléments reste inchangée.

Chaque élément du vector **link** d'un nœud mémorise l'adresse des nœuds voisin.
Par exemple pour le nœud **D**, dont les nœuds voisins sont **A**, **C** et **H**:

```
tab[3]->link[0] = tab[0]; //adresse de A
tab[3]->link[1] = tab[2]; //adresse de C
tab[3]->link[2] = tab[7]; //adresse de H
```

Attention quand on détruit un noeud:
bien veiller à mettre à jour les **vector link** de ***tous ses voisins***