EPFL

Structuration des données allouées dynamiquement

Pourquoi aurions-nous besoin d'autre chose que des **vector** ?

Ouverture vers d'autres horizons avec la notion de **liste**

Objectifs:

- Comparer deux approches d'organisation des données en terme de flexibilité et de coût calcul

Plan:

- Qu'est-ce qu'une liste chaînée ?
- Coût calcul des operations d'accès, d'ajout et de suppression
- Comparaison liste / tableau

Rappels allocation dynamique

Allocation dynamique (à-la-main) :

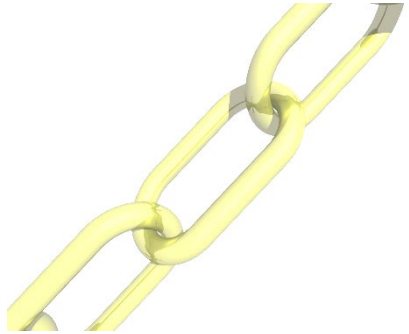
- L'opérateur **new** permet d'allouer **l'espace mémoire** pour une seule instance
    - Ex: `Nom_de_type* p(new Nom_de_type);`
- L'opérateur **new [N]** permet d'allouer **l'espace mémoire** pour N instances
    - Ex: `Nom_de_type* tab(new Nom_de_type[N]);`

Allocation dynamique avec taille adaptative **en toute sécurité**:

- **vector** adapte l'espace réservé selon les besoins
    - (revoir ce cours)

EPFL

Qu'est-ce qu'une liste chaînée ?



Avantage d'un tableau/vector:

les éléments sont consécutifs en mémoire : ***accès*** à coût calcul constant avec **[** i **]**
Indépendamment de la taille du tableau/vector

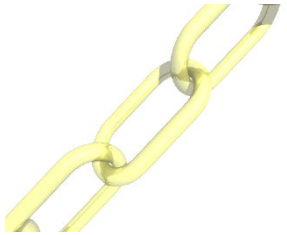
Inconvénient : si le tableau est trié et qu'il doit rester trié quand on ***ajoute*** ou ***enlève*** un élément, cela implique un coût calcul en fonction de la taille N du tableau :

Alternative au tableau/vector : la liste chaînée

Repose sur ***la gestion*** (allocation/libération) ***d'un élément à la fois***
Les éléments ne sont pas obligatoirement consécutifs en mémoire

Coût calcul moindre pour les opération d'ajouter et d'enlever un élément sous certaines conditions.

Liste simplement chaînée (1)



Principe de la mise en œuvre d'une liste simplement chaînée :

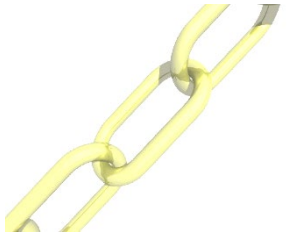
- Une **instance** de la classe sert à représenter UN élément de la liste chaînée

- un des champs est un pointeur qui pointe vers l'élément ***suivant*** de la liste chaînée (liste simplement chaînée)

```
class Element
{
    ...  données stockées dans un élément

    Element * suivant;
};
```

Liste simplement chaînée (2)



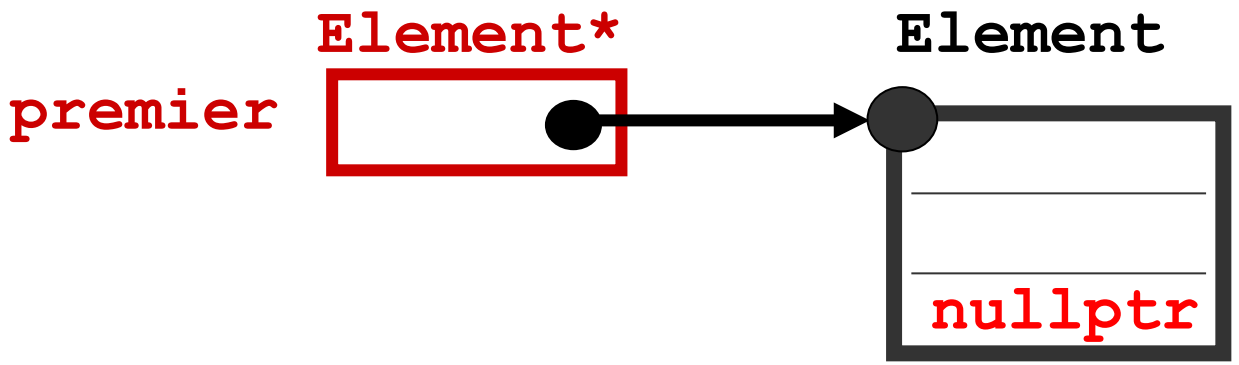
Pour gérer une liste chaînée il faut en plus prévoir un pointeur qui va mémoriser l'adresse du premier élément de la chaîne.

Exemple: **`premier`** pointe sur une liste vide

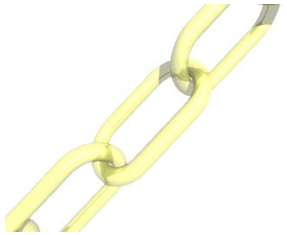
```
Element * premier = nullptr ;
```

Des fonctions se chargent d'ajouter ou d'enlever un élément pour une liste identifiée par le pointeur sur son premier élément.

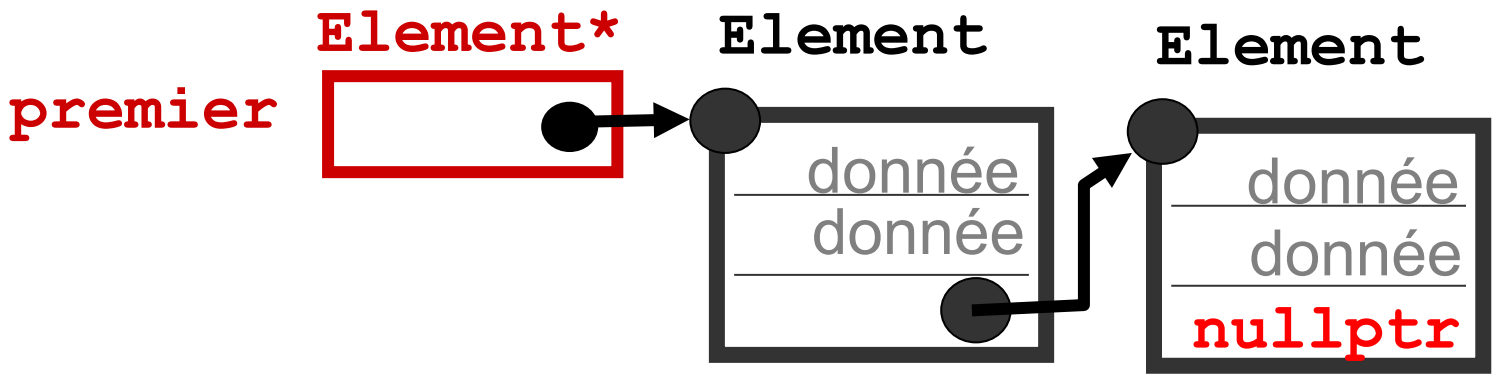
Exemple: **`premier`** pointe sur une liste possédant un seul élément



Liste simplement chaînée (3)



Le champ **`suivant`** indique l'adresse de l'élément suivant ou NULL s'il est le dernier.

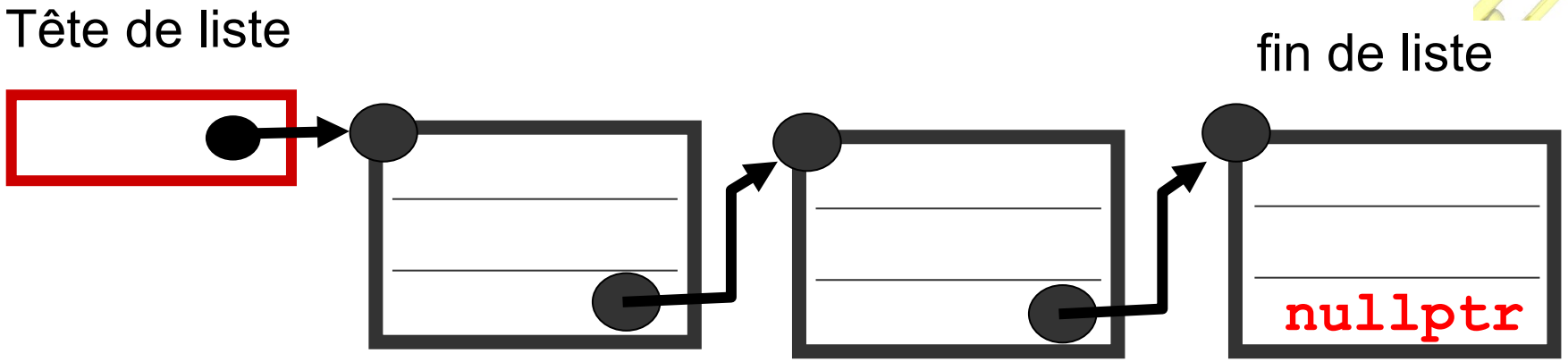


L'ajout d'un nouvel élément peut se faire "en tête" ou "en fin" de liste, ou "au bon endroit" si elle est triée.

Si la liste n’a pas besoin triée, l'ajout le moins coûteux en temps calcul est "en tête"

Coût calcul des opérations de base

Accès à un élément d'une liste

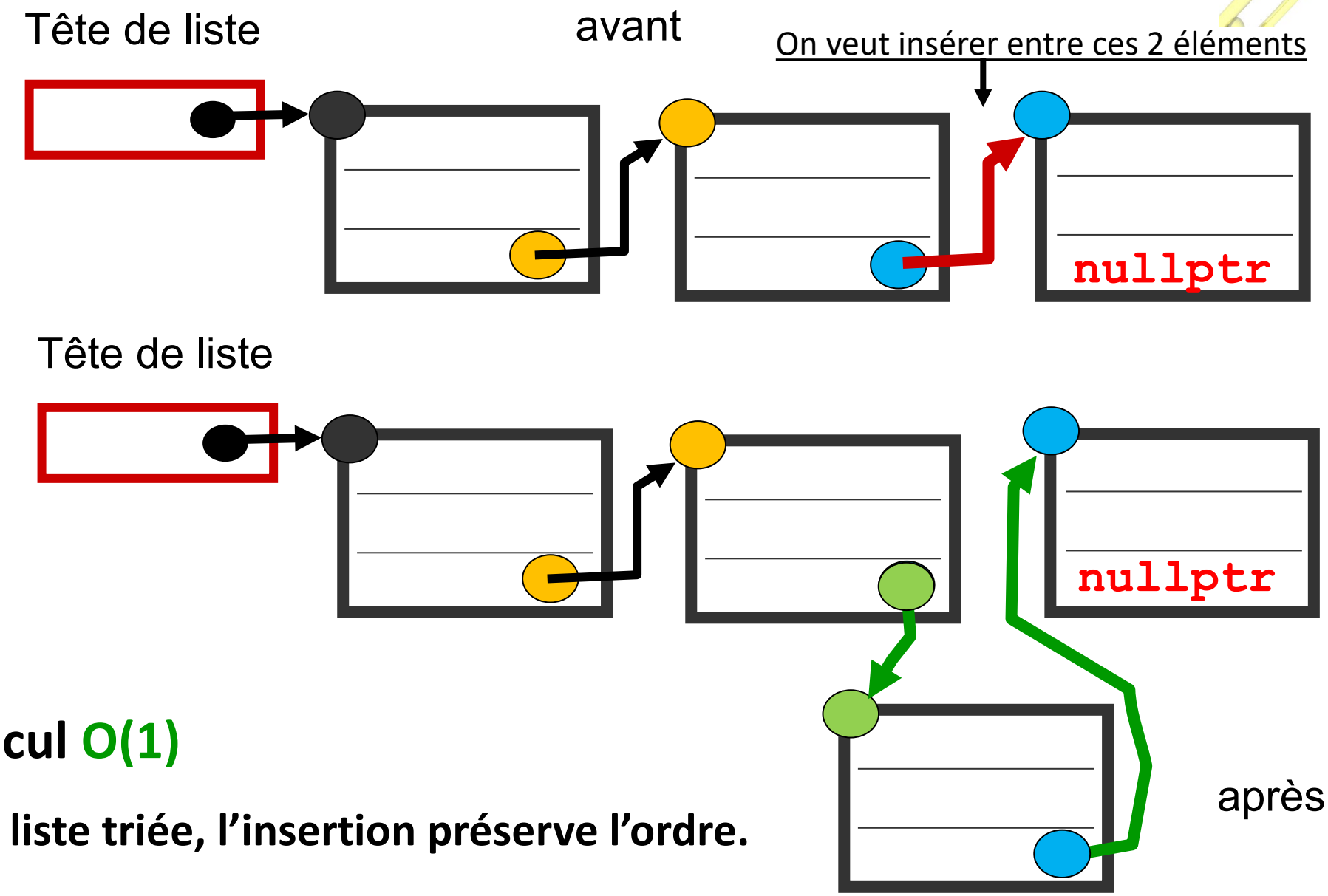


L'accès à un élément n'est pas aussi efficace que pour un tableau/vector:
il se fait séquentiellement en partant de la tête de liste et en parcourant la liste jusqu'à atteindre l'élément recherché.

Le coût calcul est O(N) pour une liste de N éléments.

Autre inconvénient: même en cas de liste triée, on ne peut PAS mettre en oeuvre une recherche dichotomique. Le coût reste O(N).

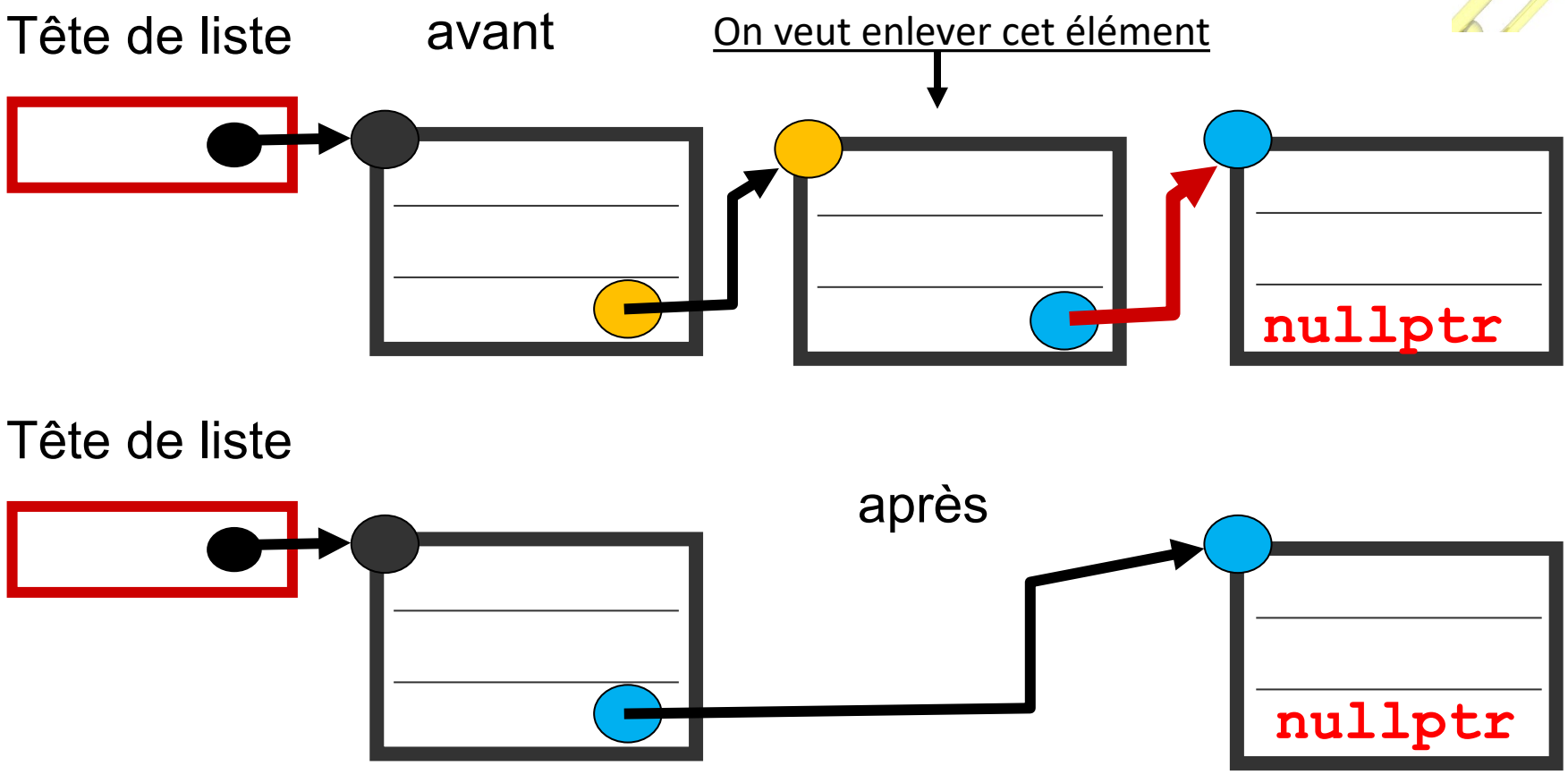
Coût calcul des opérations de base : **ajouter**



Coût calcul O(1)

En cas de liste triée, l'insertion préserve l'ordre.

Coût calcul des opérations de base : **enlever**



Coût calcul O(1)

En cas de liste triée, on préserve aussi l'ordre.

Comparaison liste <-> tableau/vector (1)

- Accès à un élément d'un tableau de N éléments: **O(1)**
  - l'accès est direct avec l'indice de l'élément : tab[i]

- Recherche d'un élément dans un tableau trié : **O(log2(N))**
  - algorithme de la recherche dichotomique

- Recherche dans un tableau non trié : **O(N)**
- **Ajouter** ou **enlever** un élément *(cf ficher vector Week4)* :
  - dans un tableau trié: **O(N)**
  - dans un tableau non-trié: **O(1)**

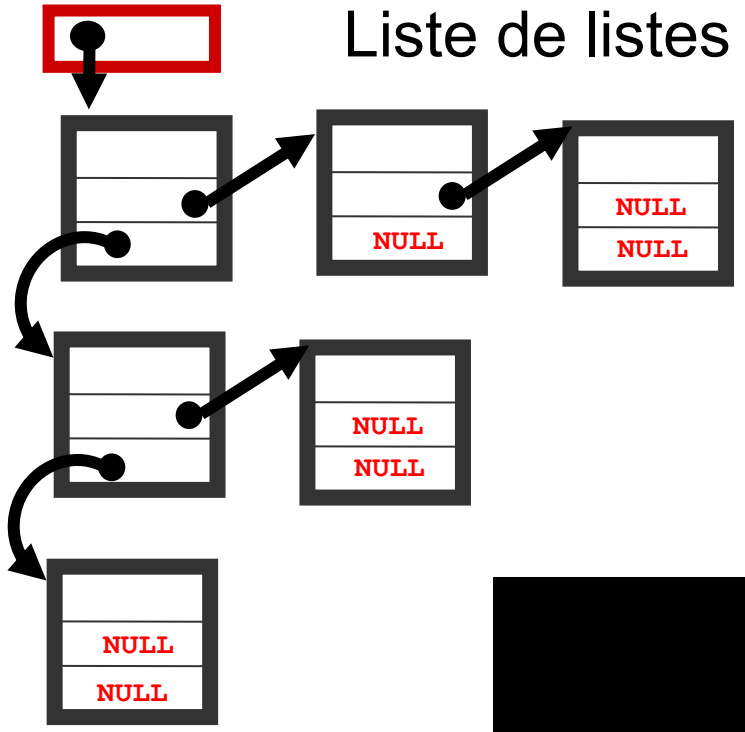
Comparaison liste <-> tableau (2)

- Effectuer une opération *simple* sur tous les éléments d'un tableau ou d'une liste : même coût pour liste et tableau = **O(N)**
  - ex: dessin, écriture sur fichier, etc…

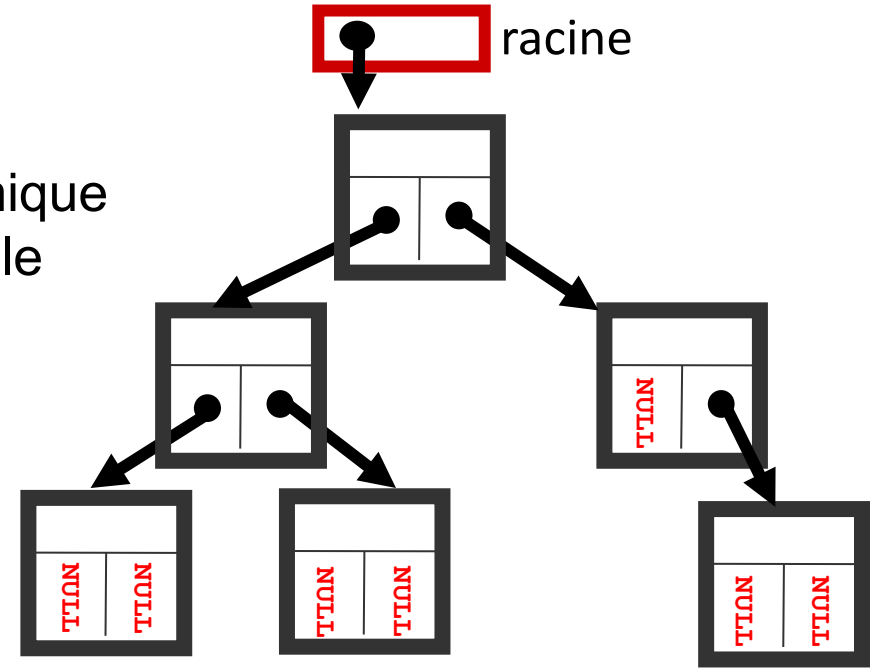
- Comportement vis à vis de la mémoire cache:
  - accès **ligne par ligne** d'un tableau est beaucoup plus efficace que **colonne par colonne**
  - le parcours d'une liste est aussi efficace qu'un parcours ligne par ligne d'un tableau Si la liste vient juste d'être créée (ex. Lecture de fichier).
  - cette efficacité diminue au fur et à mesure qu'on enlève puis ajoute des éléments car les éléments consécutifs de la liste ne sont plus consécutifs en mémoire.

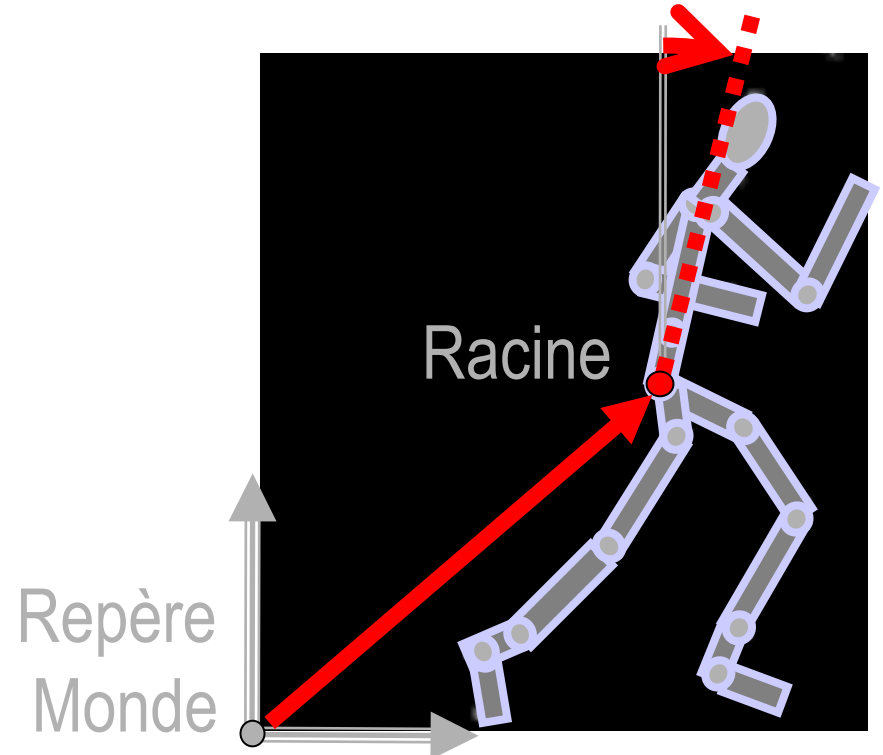
Quelques autres possibilités de structuration des données

Liste de listes (~arbre binaire)



Recherche dichotomique
à nouveau possible
en $O(\log_2(N))$





<u>arbre N-aire</u>: un élément possède

- un seul parent
- de zero à N descendants
- *pour cet exemple*:
  - position/orientation relative au parent

Résumé

- **La liste chaînée offre une grande flexibilité pour la mémorisation d'une grande quantité de données produite dynamiquement (ajout/retrait).**

- **Son coût calcul de O(1) est meilleur que le tableau pour les opérations d'ajout et de retrait d'un élément dans un ensemble trié.**

- **Son parcours séquentiel implique un coût calcul en O(N) pour l'accès à un élément alors qu'il est constant pour un tableau.**

- **La liste chaîne présente des limitations qui peuvent être surmontées avec des structures de données telles que l'arbre binaire qui permet la recherche dichotomique en $O(\log_2(N))$**

- **Ce type de structuration des données est très riche : arbre, graphe...**