Cours 4

François Delobel, Anaïs Durand, Abdel Hasbani,
Laurent Provot, Carine Simon

C++ - S2 - 2022-2023

IUT CLERMONT AUVERGNE
Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

Comment choisir le bon conteneur ?

- Selon la nature des valeurs :
  - nombre fixe ou variable ?
  - valeurs uniques ?
  - clé ?
  - valeurs ordonnées ?

- Selon leur utilisation :
  - accès à un élément par sa position, par sa valeur ou par sa clé ?
  - fréquence des opérations d'insertion, suppression, accès ...

Tableau

Tableau

Taille fixe

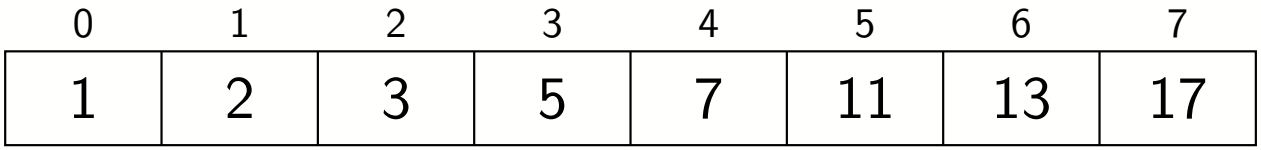
```
int t[8]{1,2,3,5,7,11,13,17};
```

0	1	2	3	4	5	6	7
1	2	3	5	7	11	13	17

Tableau

Taille fixe

```
int t[8]{1,2,3,5,7,11,13,17};
```



0	1	2	3	4	5	6	7
1	2	3	5	7	11	13	17

$\Rightarrow$ À privilégier pour des collections avec un nombre d'éléments fixé.

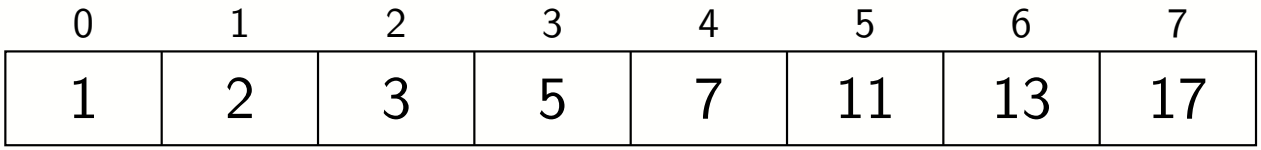
vector, list

vector vs. list

Rappel :

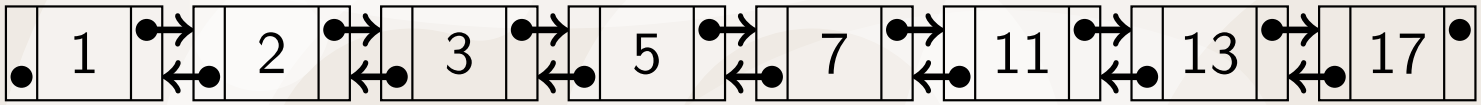
- `vector` : tableau dynamique

```
std::vector<int> v{1,2,3,5,7,11,13,17};
```



- `list` : liste doublement chaînée

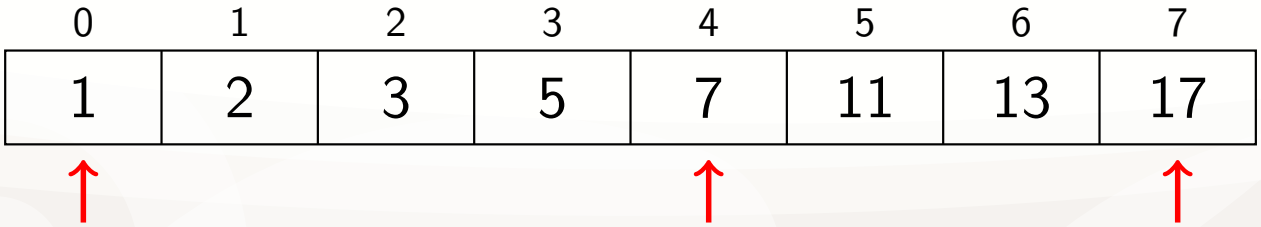
```
std::list<int> l{1,2,3,5,7,11,13,17};
```



Accès à un élément par sa position

Accès immédiat quelle que soit la position.

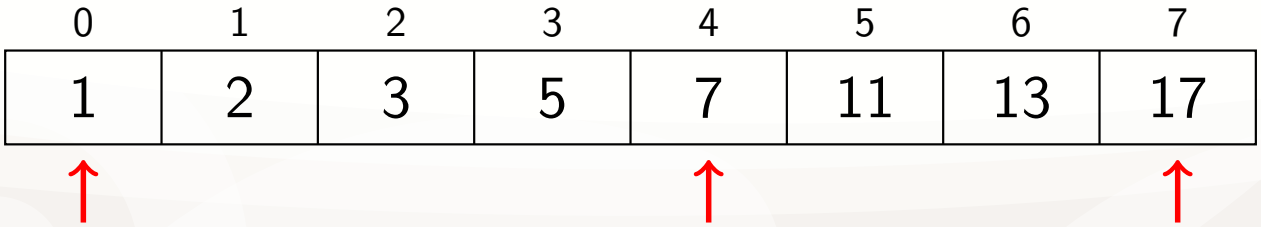
```
v[pos]
v.at(pos)
```



Accès à un élément par sa position

Accès immédiat quelle que soit la position.

```
v[pos]
v.at(pos)
```

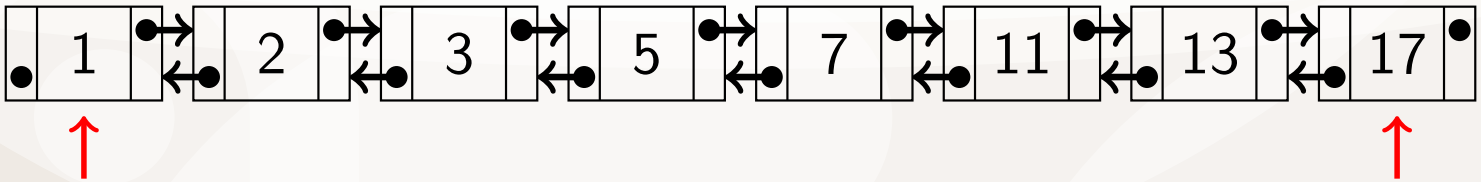


$\Rightarrow$ **1 opération**

Accès à un élément par sa position

Pas d'accès immédiat sauf pour le début et à la fin de la liste.

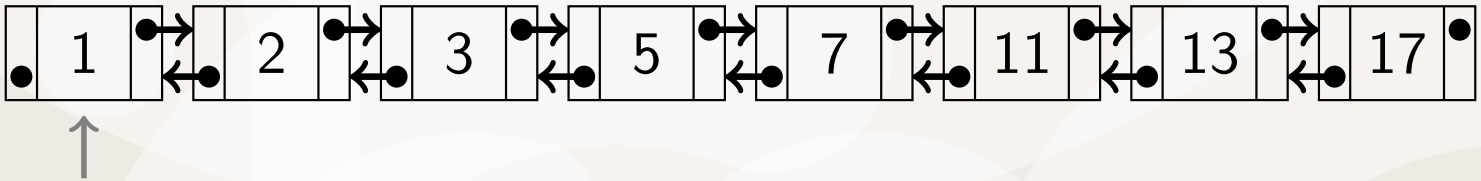
```
l.front()
l.back()
```



Accès à un élément par sa position

Besoin de parcourir la liste pour atteindre les autres positions.

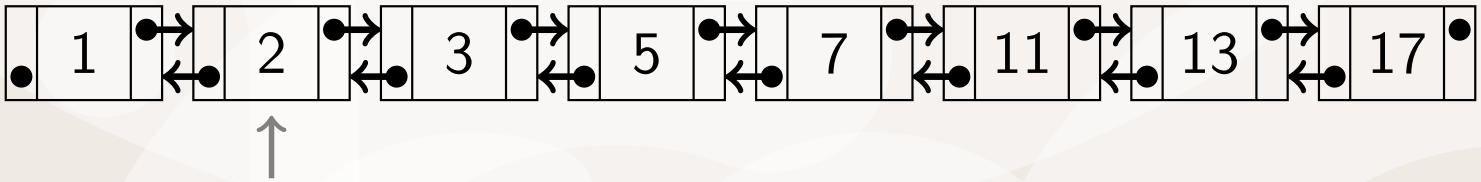
```
int i = 0;
for(auto it=l.begin(); it!=l.end(); it++){
  if(i == pos){
    //*it = élément en position pos
    break;
  }
  i++;
}
```



Accès à un élément par sa position

Besoin de parcourir la liste pour atteindre les autres positions.

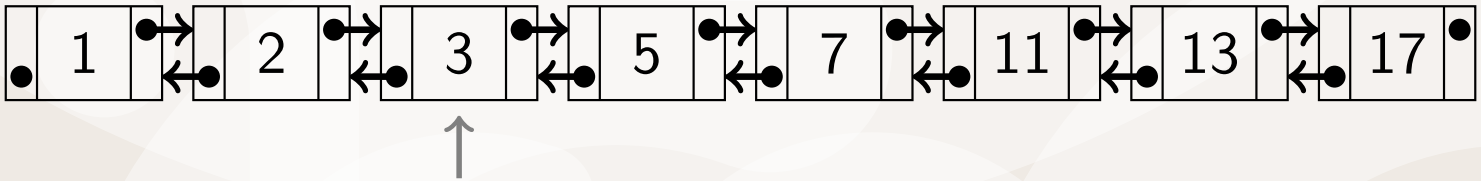
```
int i = 0;
for(auto it=l.begin(); it!=l.end(); it++){
  if(i == pos){
    //*it = élément en position pos
    break;
  }
  i++;
}
```



Accès à un élément par sa position

Besoin de parcourir la liste pour atteindre les autres positions.

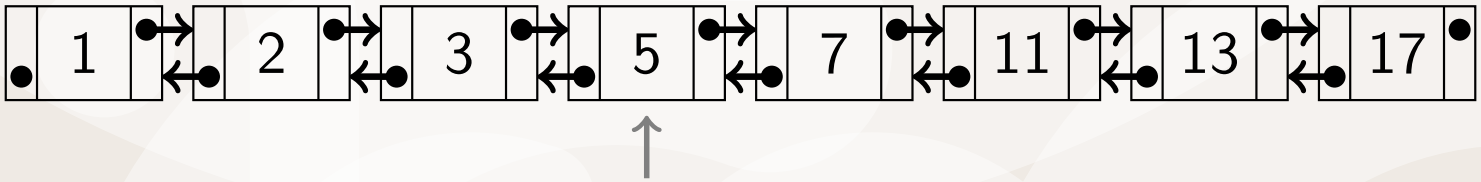
```
int i = 0;
for(auto it=l.begin(); it!=l.end(); it++){
  if(i == pos){
    //*it = élément en position pos
    break;
  }
  i++;
}
```



Accès à un élément par sa position

Besoin de parcourir la liste pour atteindre les autres positions.

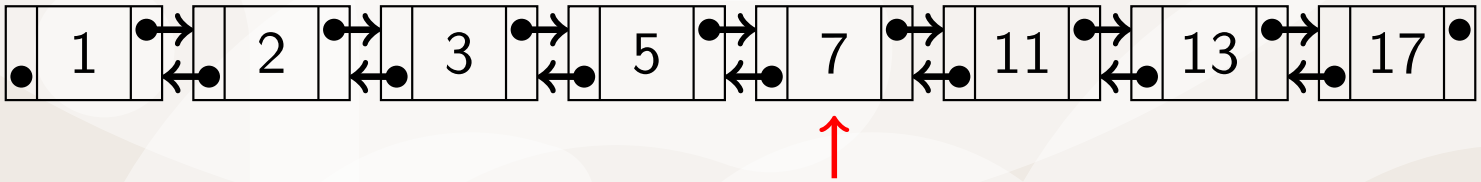
```
int i = 0;
for(auto it=l.begin(); it!=l.end(); it++){
  if(i == pos){
    //*it = élément en position pos
    break;
  }
  i++;
}
```



Accès à un élément par sa position

Besoin de parcourir la liste pour atteindre les autres positions.

```
int i = 0;
for(auto it=l.begin(); it!=l.end(); it++){
  if(i == pos){
    //*it = élément en position pos
    break;
  }
  i++;
}
```

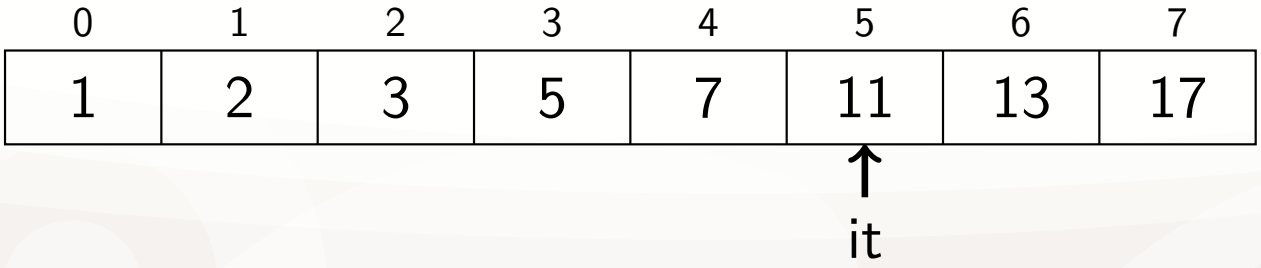


$\Rightarrow \leq n-1$ **opérations**

Insertion d'un élément

Toutes les valeurs placées après la position indiquée doivent être décalées.

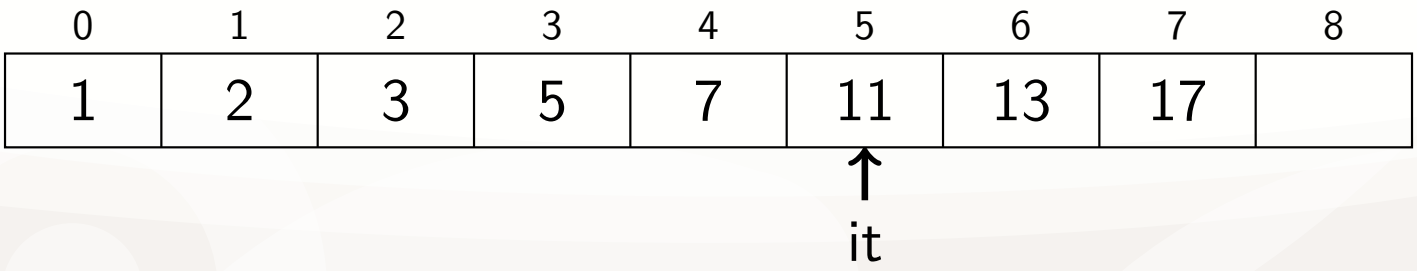
```
v.insert(it, 10);
```



Insertion d'un élément

Toutes les valeurs placées après la position indiquée doivent être décalées.

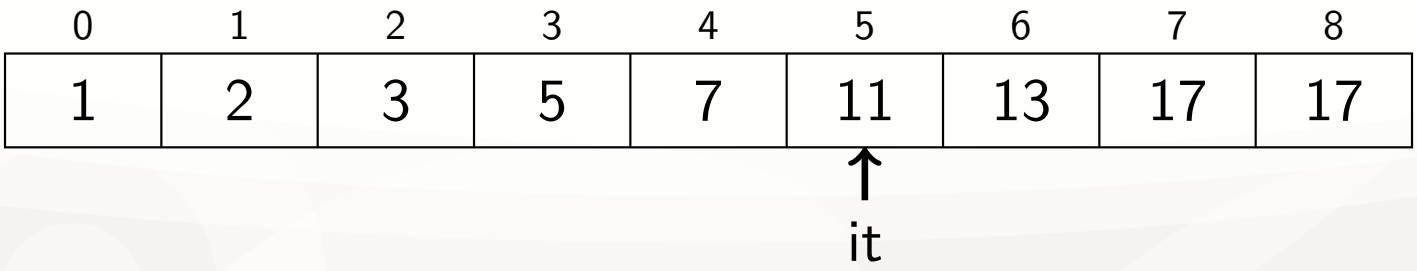
```
v.insert(it, 10);
```



Insertion d'un élément

Toutes les valeurs placées après la position indiquée doivent être décalées.

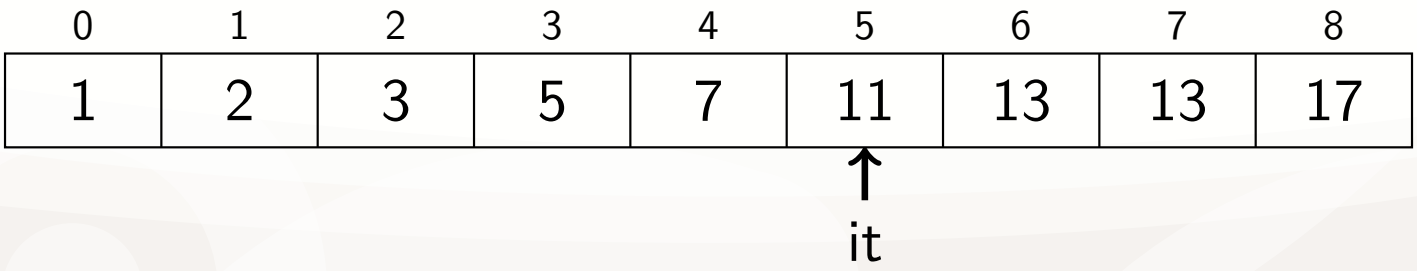
```
v.insert(it, 10);
```



Insertion d'un élément

Toutes les valeurs placées après la position indiquée doivent être décalées.

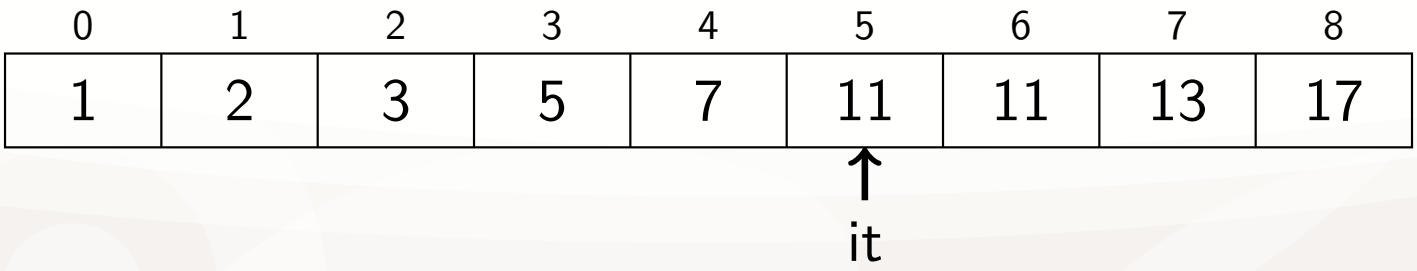
```
v.insert(it, 10);
```



Insertion d'un élément

Toutes les valeurs placées après la position indiquée doivent être décalées.

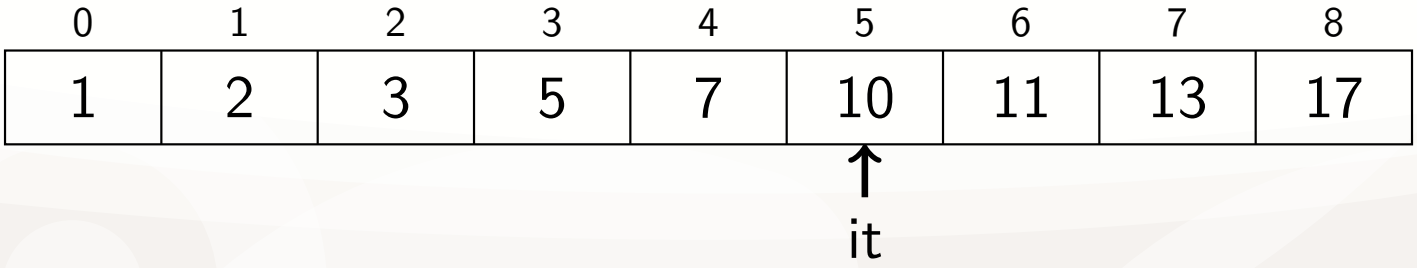
```
v.insert(it, 10);
```



Insertion d'un élément

Toutes les valeurs placées après la position indiquée doivent être décalées.

```
v.insert(it, 10);
```

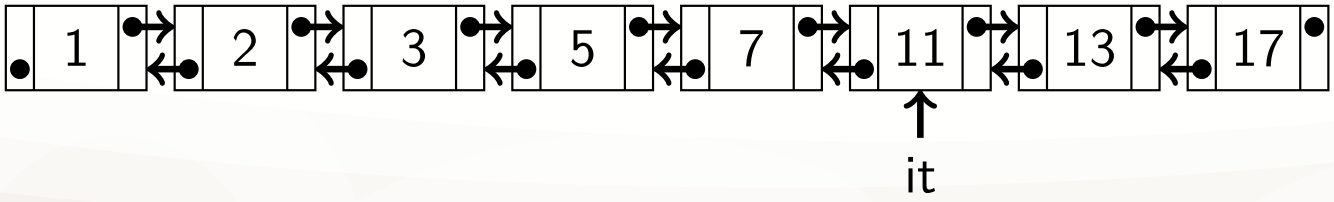


$\Rightarrow \leq n+1$ **opération**

Insertion d'un élément

Il suffit de modifier le chaînage.

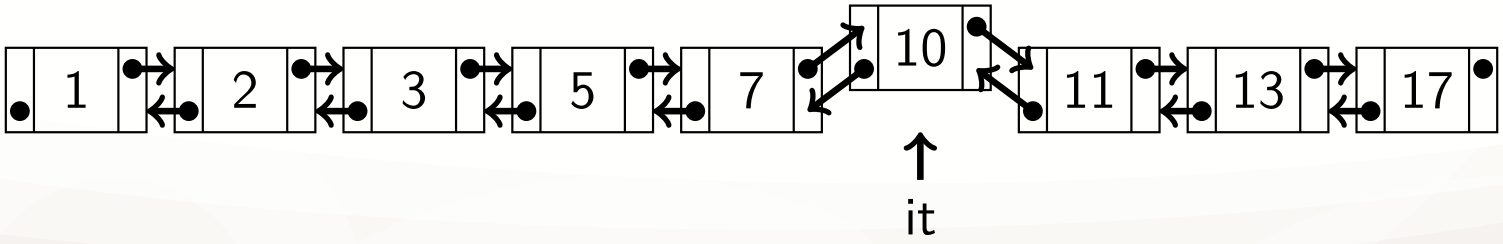
```
l.insert(it, 10);
```



Insertion d'un élément

Il suffit de modifier le chaînage.

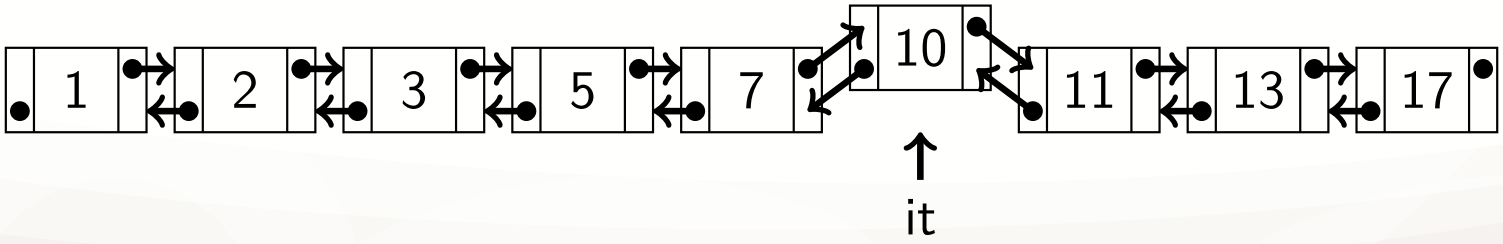
```
l.insert(it, 10);
```



Insertion d'un élément

Il suffit de modifier le chaînage.

```
l.insert(it, 10);
```

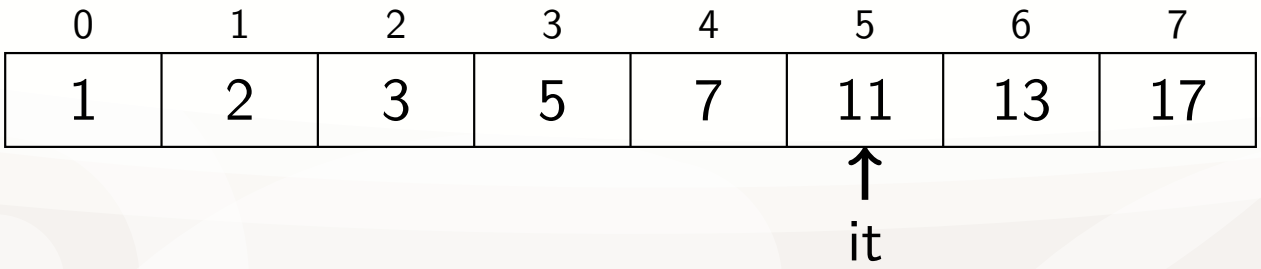


$\Rightarrow$ **3 opérations**

Suppression d'un élément

Toutes les valeurs placées après la position indiquée doivent être décalées.

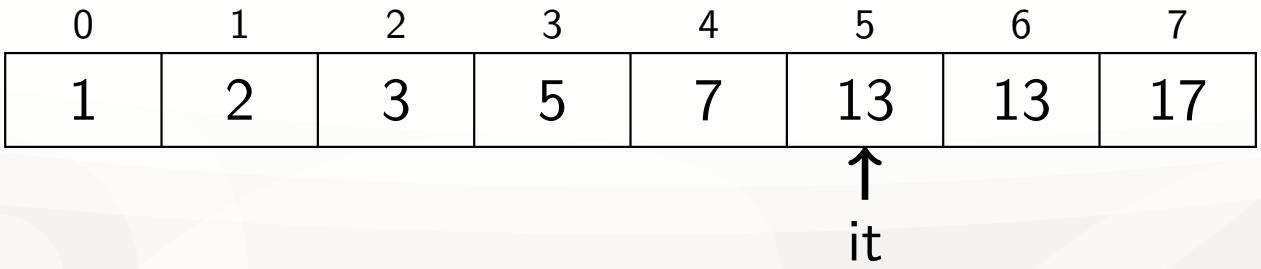
```
v.erase(it);
```



Suppression d'un élément

Toutes les valeurs placées après la position indiquée doivent être décalées.

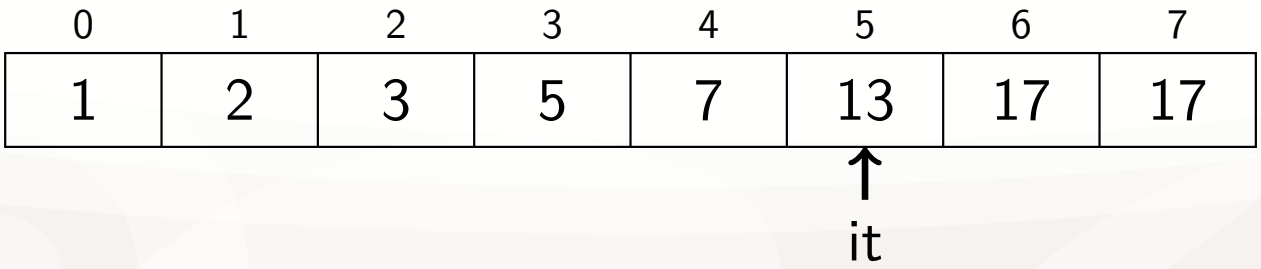
```
v.erase(it);
```



Suppression d'un élément

Toutes les valeurs placées après la position indiquée doivent être décalées.

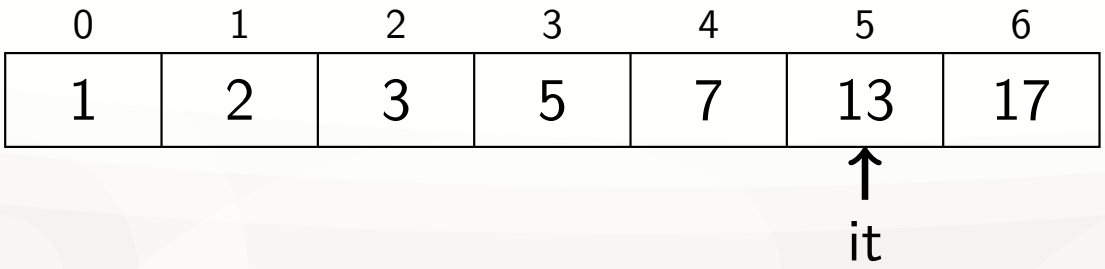
```
v.erase(it);
```



Suppression d'un élément

Toutes les valeurs placées après la position indiquée doivent être décalées.

```
v.erase(it);
```

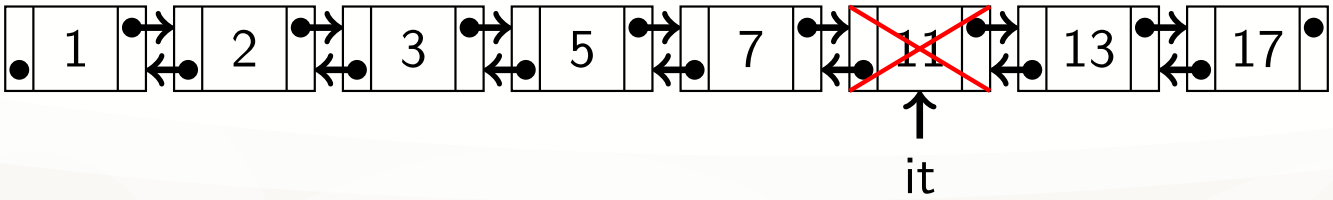


$\Rightarrow \leq n-1$ **opérations**

Suppression d'un élément

Il suffit de modifier le chaînage.

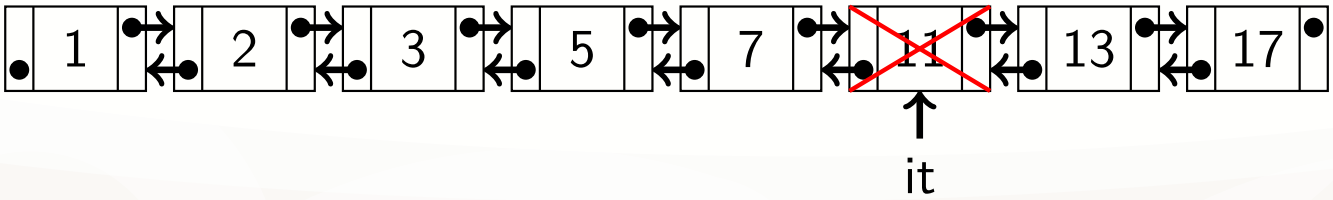
```
l.erase(it);
```



Suppression d'un élément

Il suffit de modifier le chaînage.

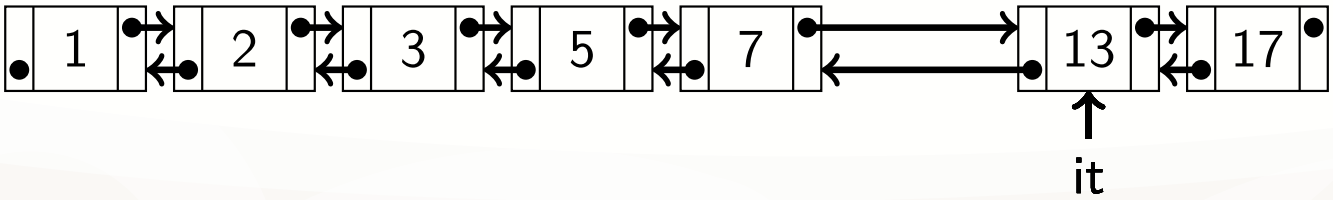
```
l.erase(it);
```



Suppression d'un élément

Il suffit de modifier le chaînage.

```
l.erase(it);
```

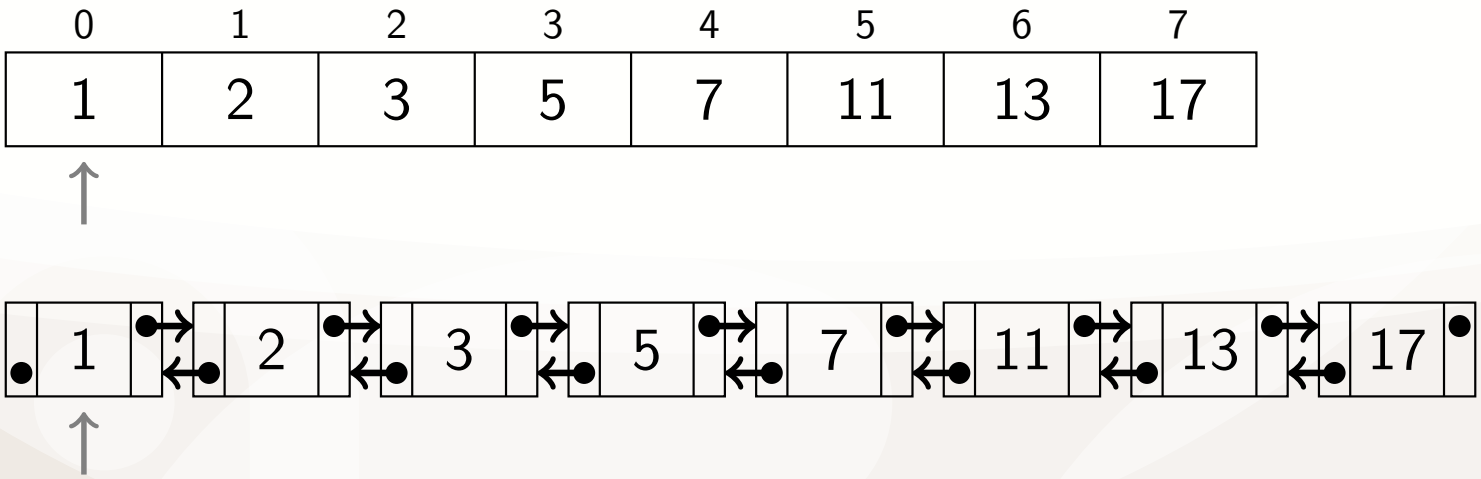


$\Rightarrow$ **3 opérations**

Recherche d'un élément

Il faut parcourir la collection jusqu'à trouver l'élément.

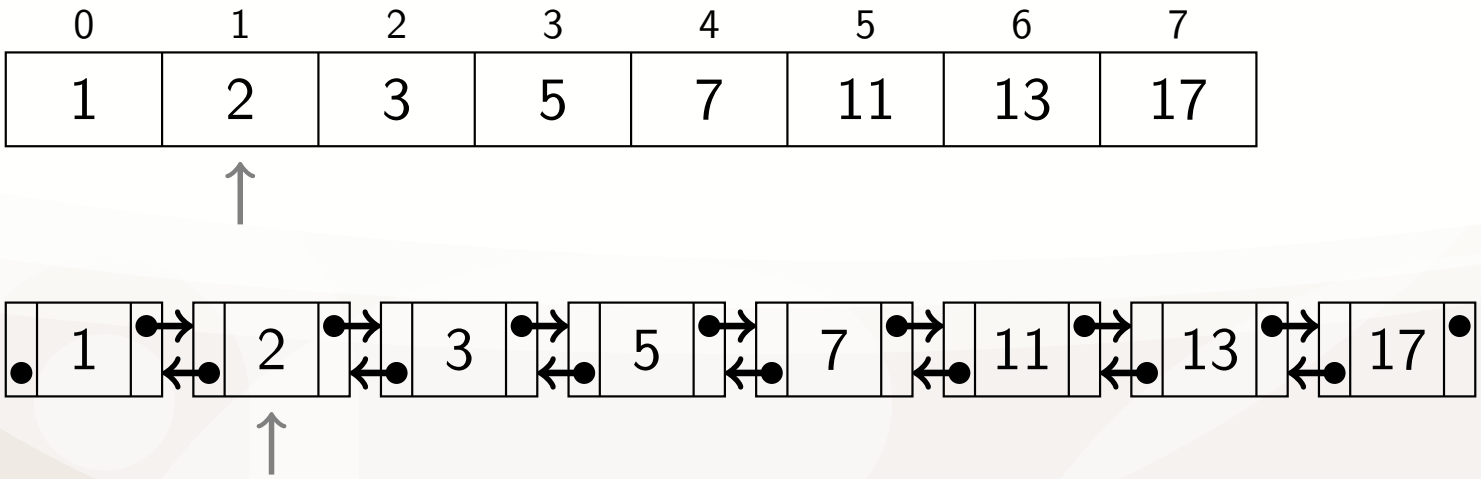
```
for(auto it=v.begin(); it!=l.end(); it++){
  if(*it == val){/*trouvé*/}
}
```



Recherche d'un élément

Il faut parcourir la collection jusqu'à trouver l'élément.

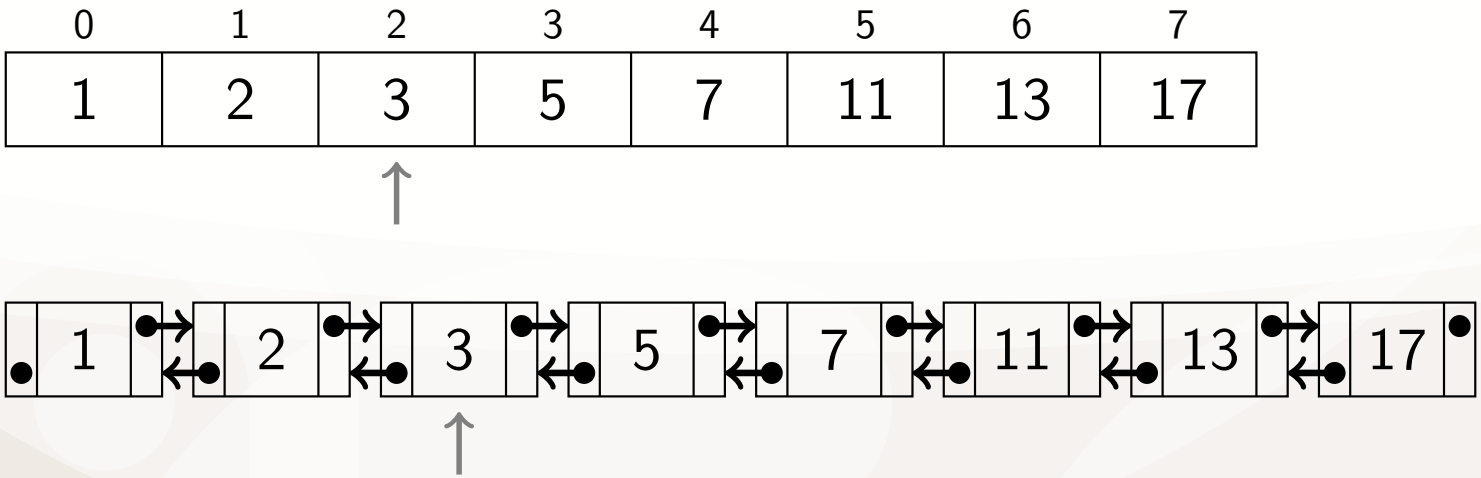
```
for(auto it=v.begin(); it!=l.end(); it++){
  if(*it == val){/*trouvé*/}
}
```



Recherche d'un élément

Il faut parcourir la collection jusqu'à trouver l'élément.

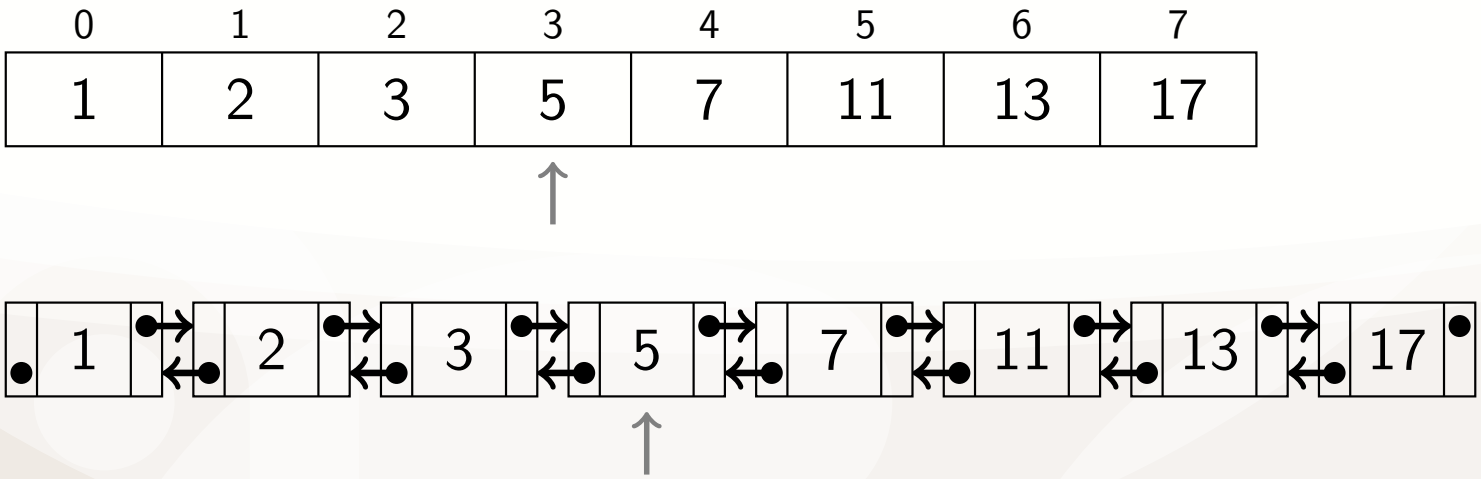
```
for(auto it=v.begin(); it!=l.end(); it++){
  if(*it == val){/*trouvé*/}
}
```



Recherche d'un élément

Il faut parcourir la collection jusqu'à trouver l'élément.

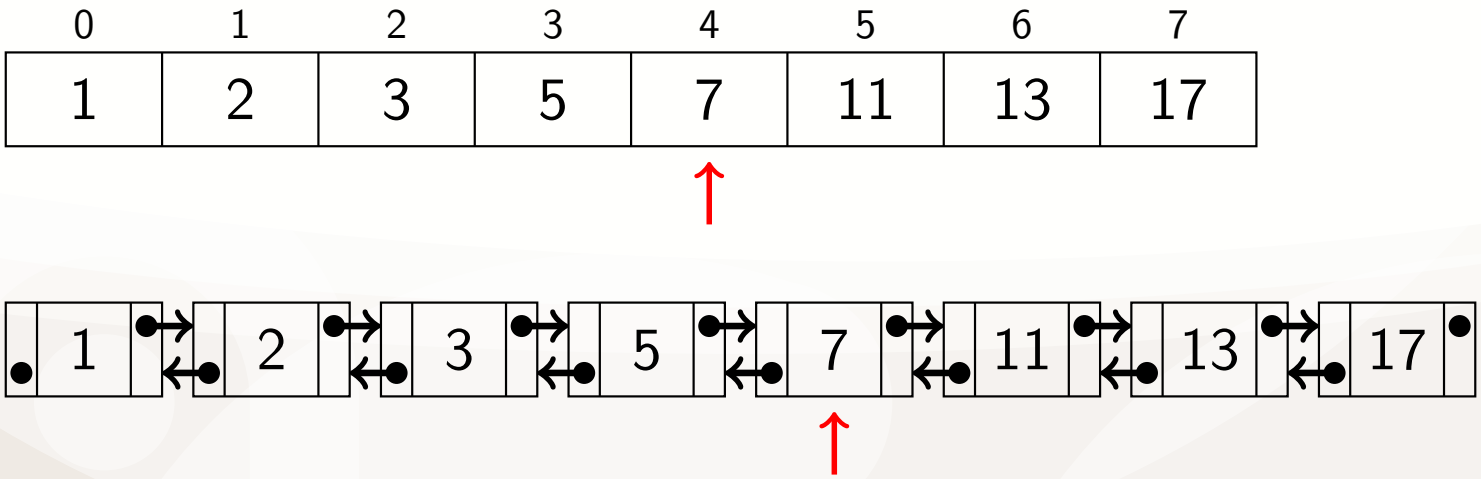
```
for(auto it=v.begin(); it!=l.end(); it++){
  if(*it == val){/*trouvé*/}
}
```



Recherche d'un élément

Il faut parcourir la collection jusqu'à trouver l'élément.

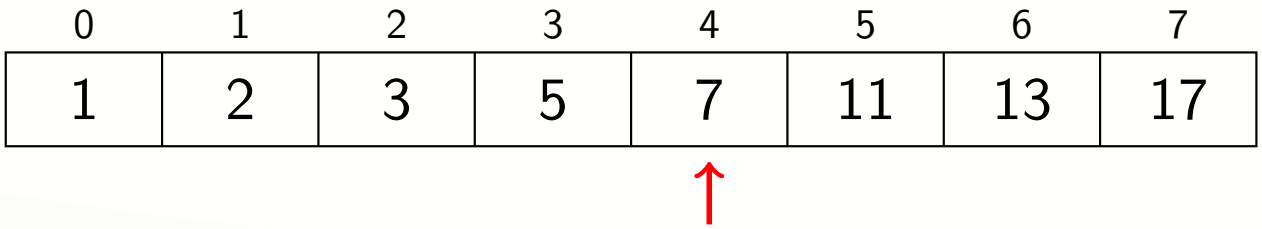
```
for(auto it=v.begin(); it!=l.end(); it++){
  if(*it == val){/*trouvé*/}
}
```

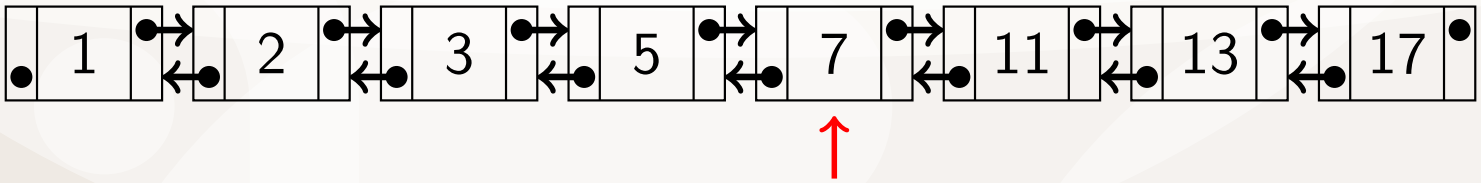


Recherche d'un élément

Il faut parcourir la collection jusqu'à trouver l'élément.

```
for(auto it=v.begin(); it!=l.end(); it++){
  if(*it == val){/*trouvé*/}
}
```





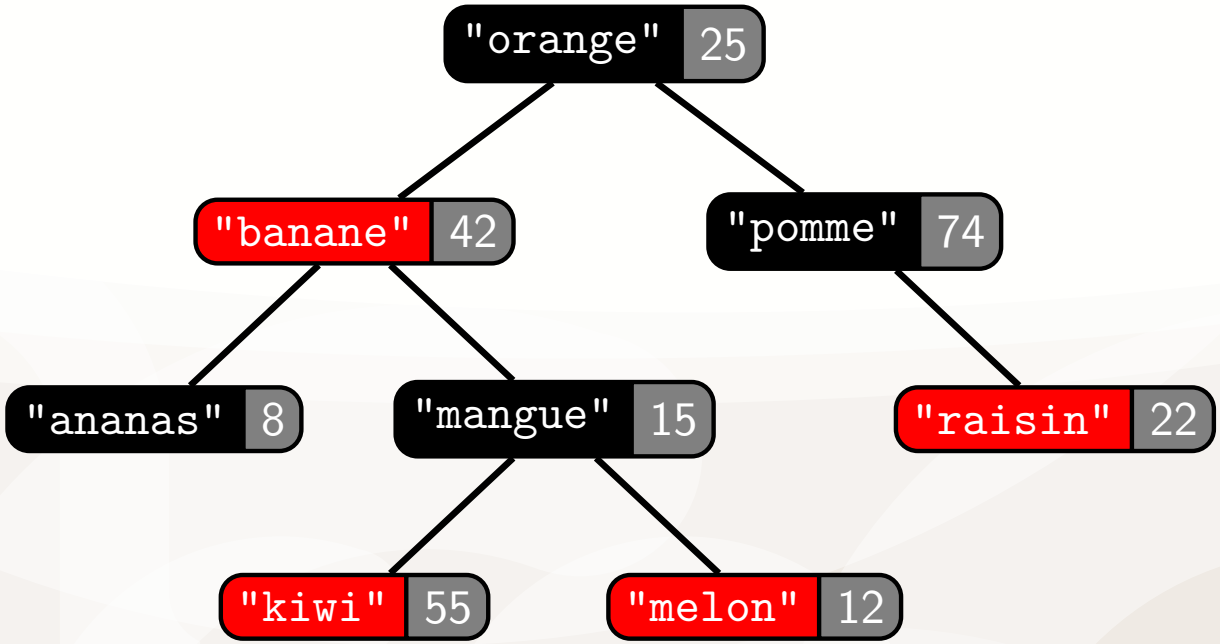
$\Rightarrow \leq n$ **opérations**

map, unordered_map, set, unordered_set

map vs. unordered_map

- `map` : conteneur associatif (paires clé, valeur) ordonné

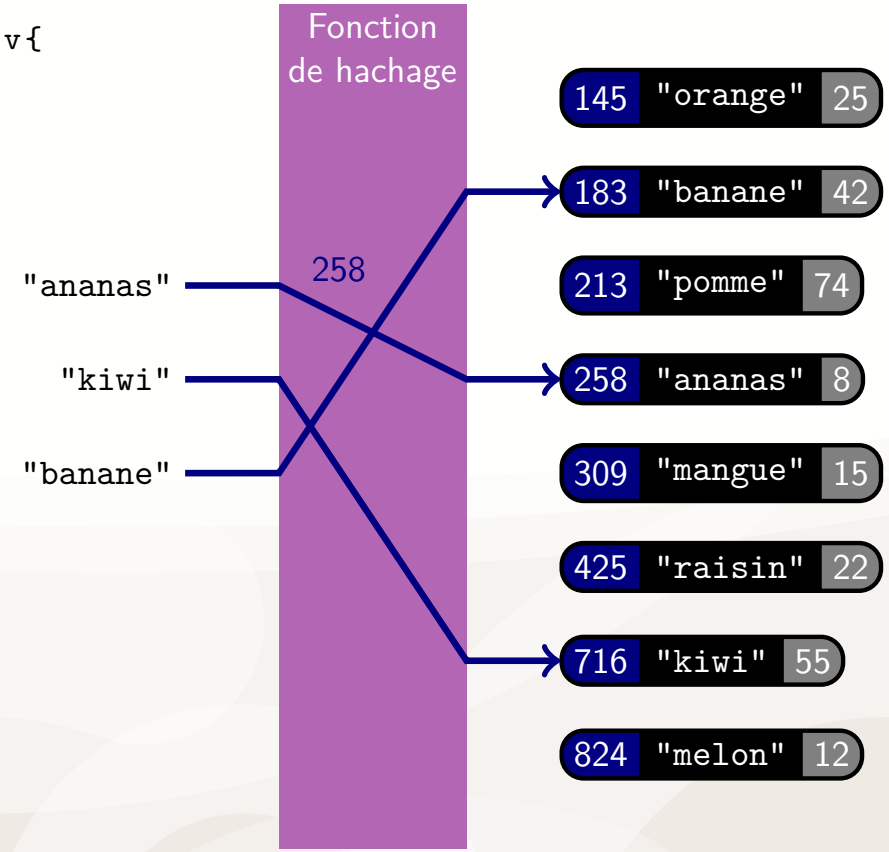
```
std::map<string, int> v{{"banane", 42}, {"orange", 25}, {"melon", 12},
  {"pomme", 74}, {"kiwi", 55}, {"ananas", 8}, {"mangue", 15}, {"raisin", 22}};
```



map vs. unordered_map

- `unordered_map` : conteneur associatif non ordonné

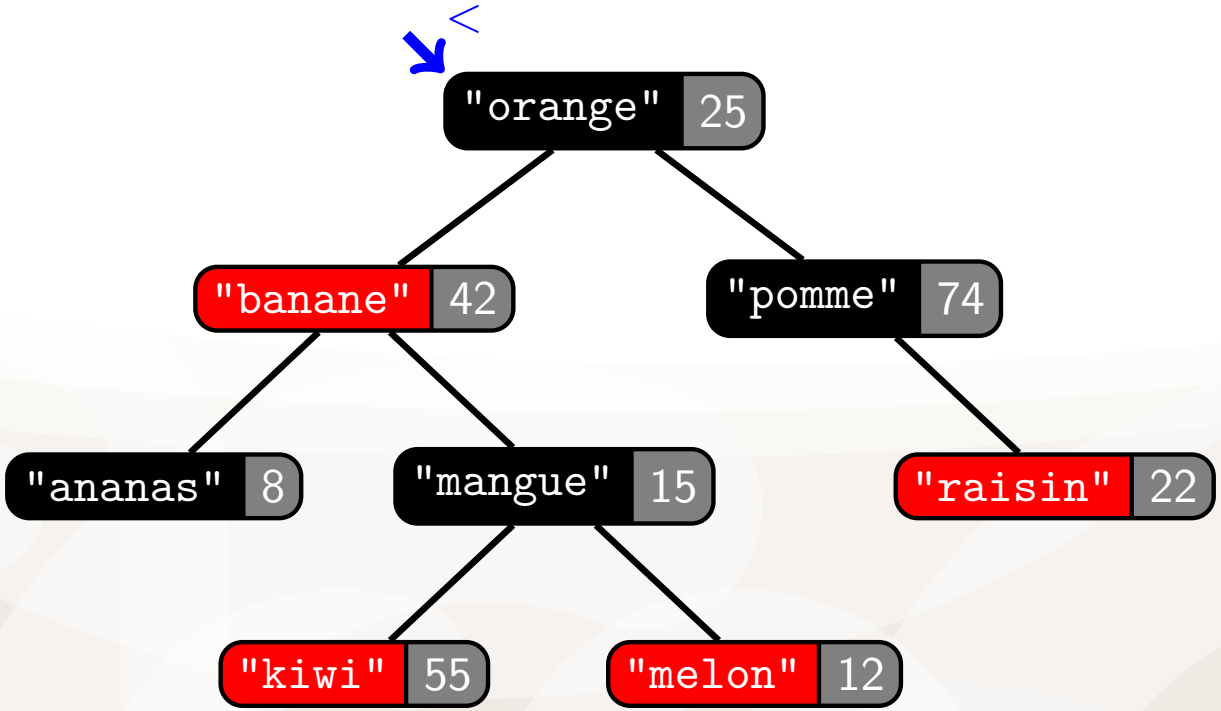
```
std::unordered_map<string, int> v{
  {"banane", 42},
  {"orange", 25},
  {"melon", 12},
  {"pomme", 74},
  {"kiwi", 55},
  {"ananas", 8},
  {"mangue", 15},
  {"raisin", 22}
};
```



Accès à un élément par sa clé

Paires (clé, valeur) ordonnées dans un arbre binaire de recherche.

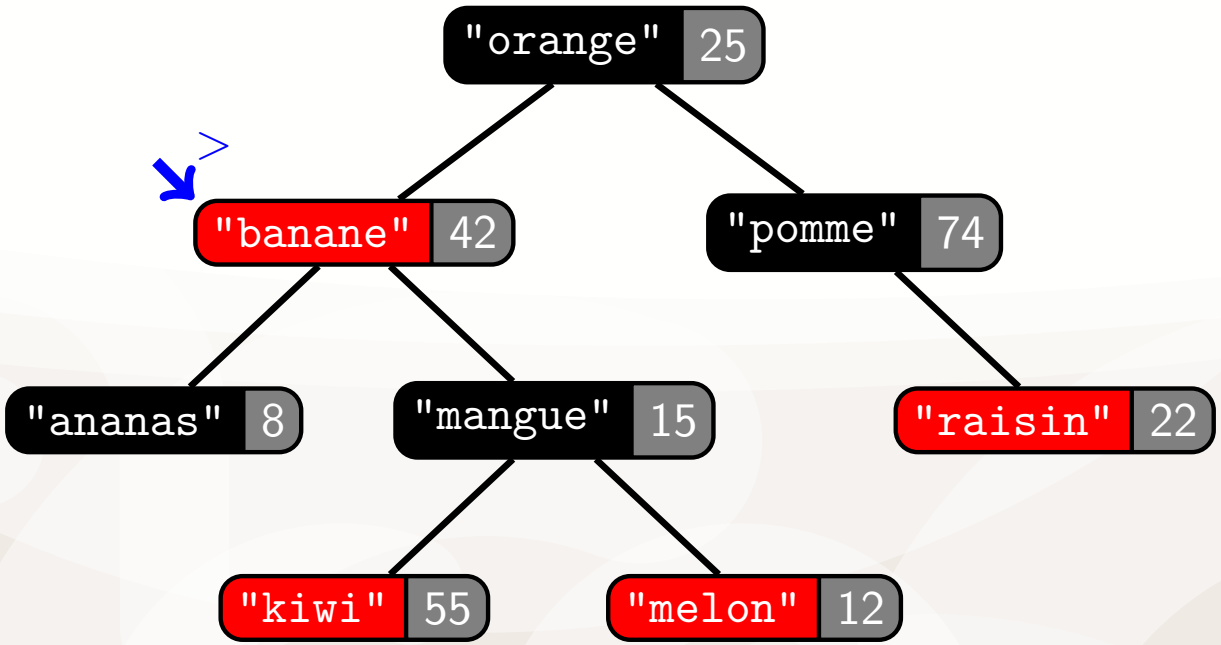
```
m["melon"]
```



Accès à un élément par sa clé

Paires (clé, valeur) ordonnées dans un arbre binaire de recherche.

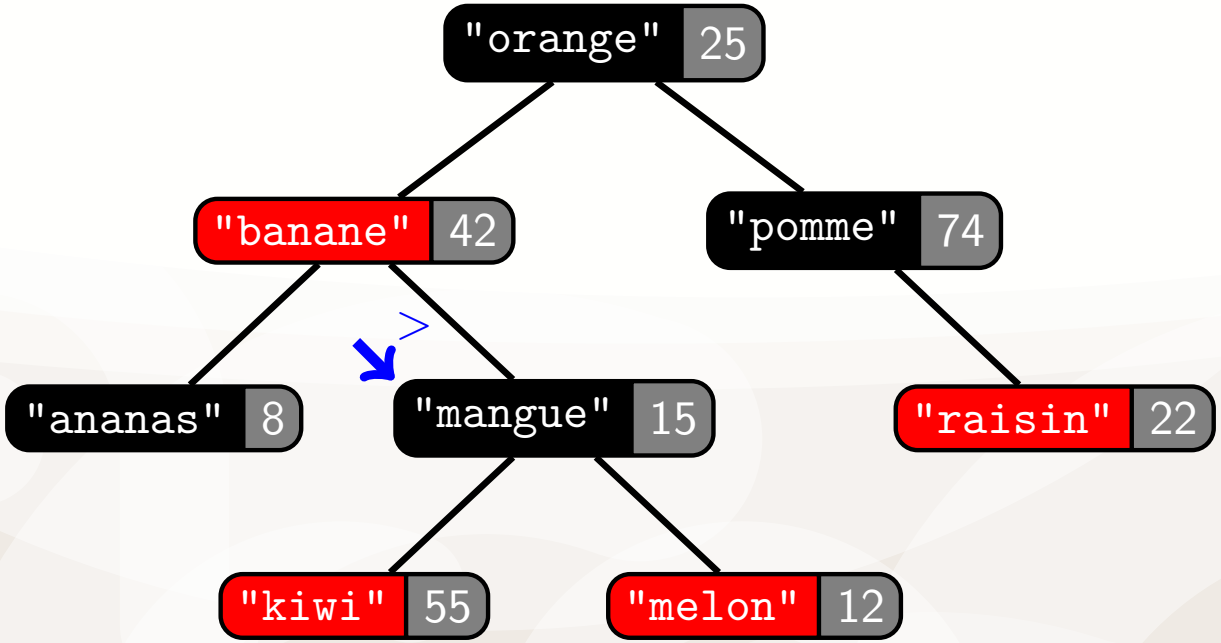
```
m["melon"]
```



Accès à un élément par sa clé

Paires (clé, valeur) ordonnées dans un arbre binaire de recherche.

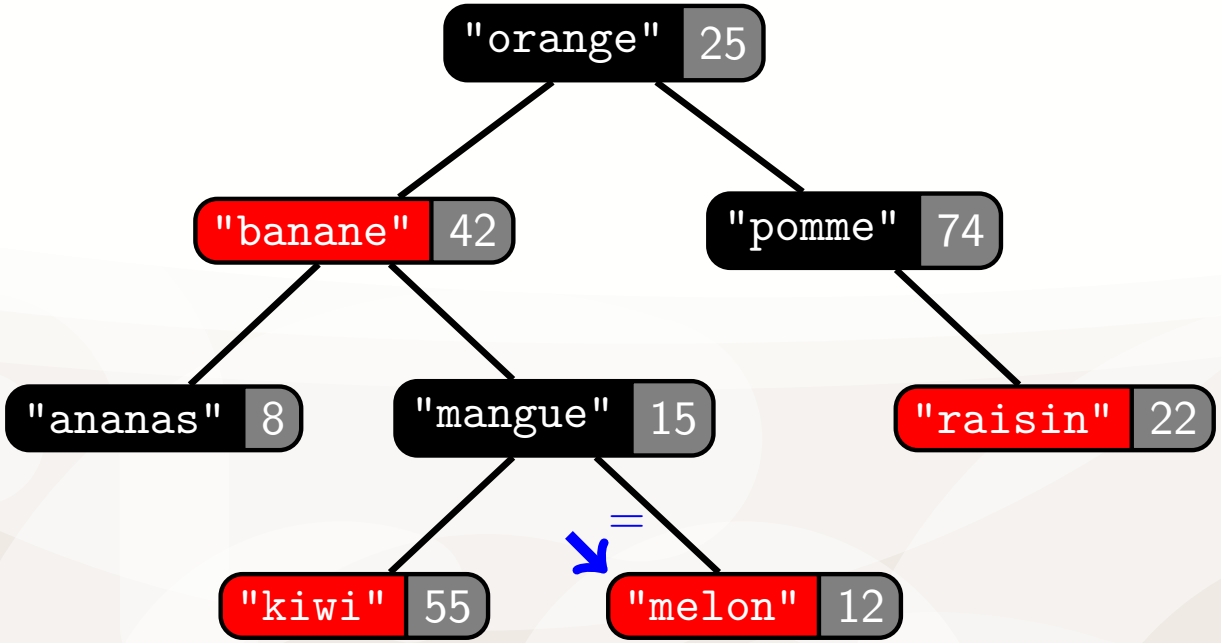
```
m["melon"]
```



Accès à un élément par sa clé

Paires (clé, valeur) ordonnées dans un arbre binaire de recherche.

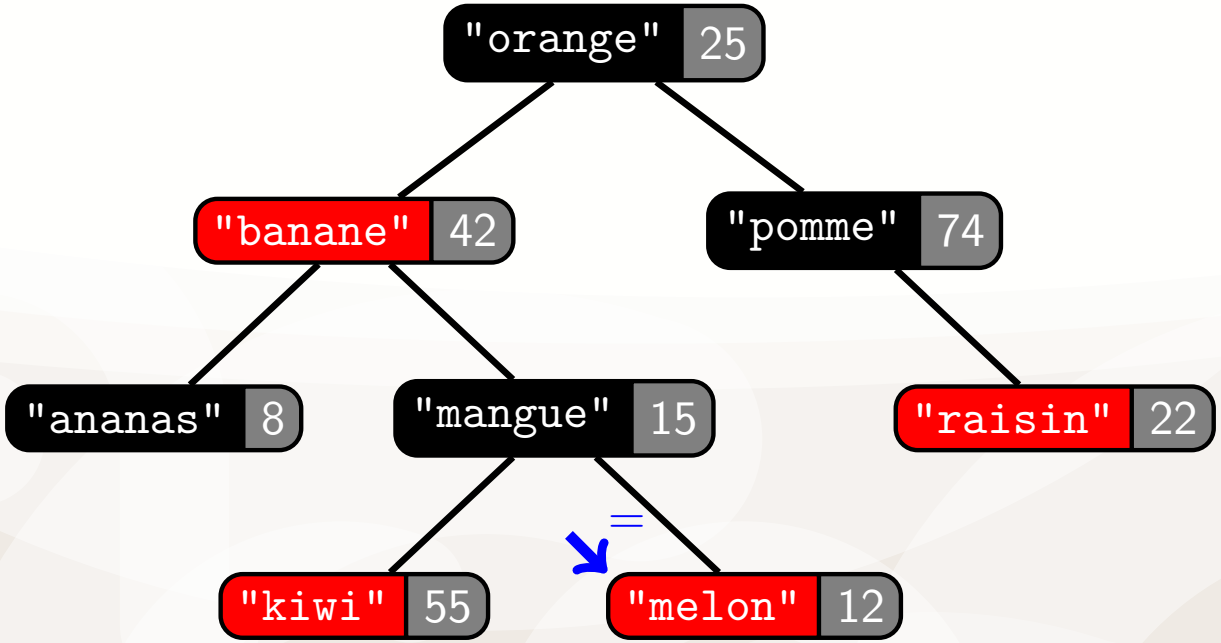
```
m["melon"]
```



Accès à un élément par sa clé

Paires (clé, valeur) ordonnées dans un arbre binaire de recherche.

```
m["melon"]
```

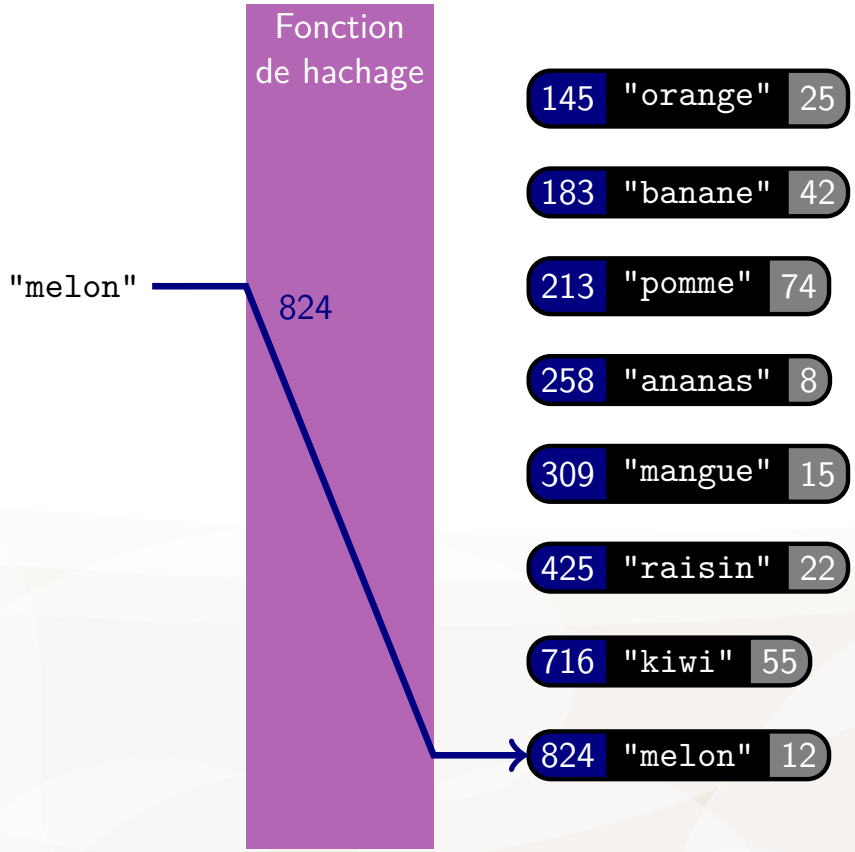


$\Rightarrow$ $O(\log(n))$ **opérations**

Accès à un élément par sa clé

Paires (clé, valeur) stockées en fonction du haché de la clé.

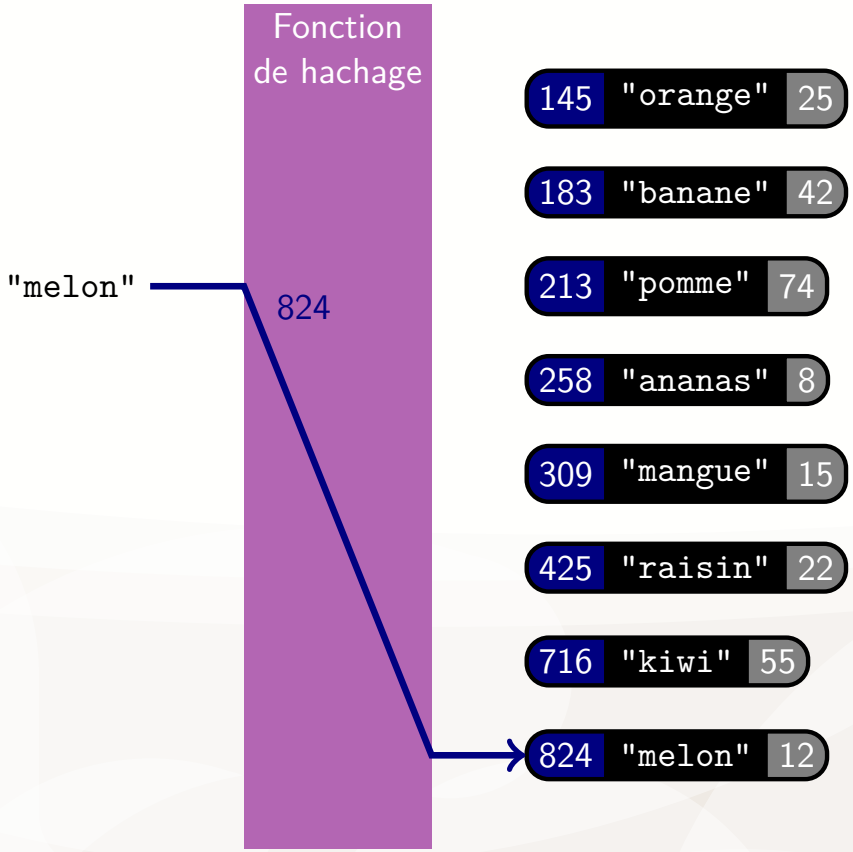
```
m["melon"]
```



Accès à un élément par sa clé

Paires (clé, valeur) stockées en fonction du haché de la clé.

```
m["melon"]
```

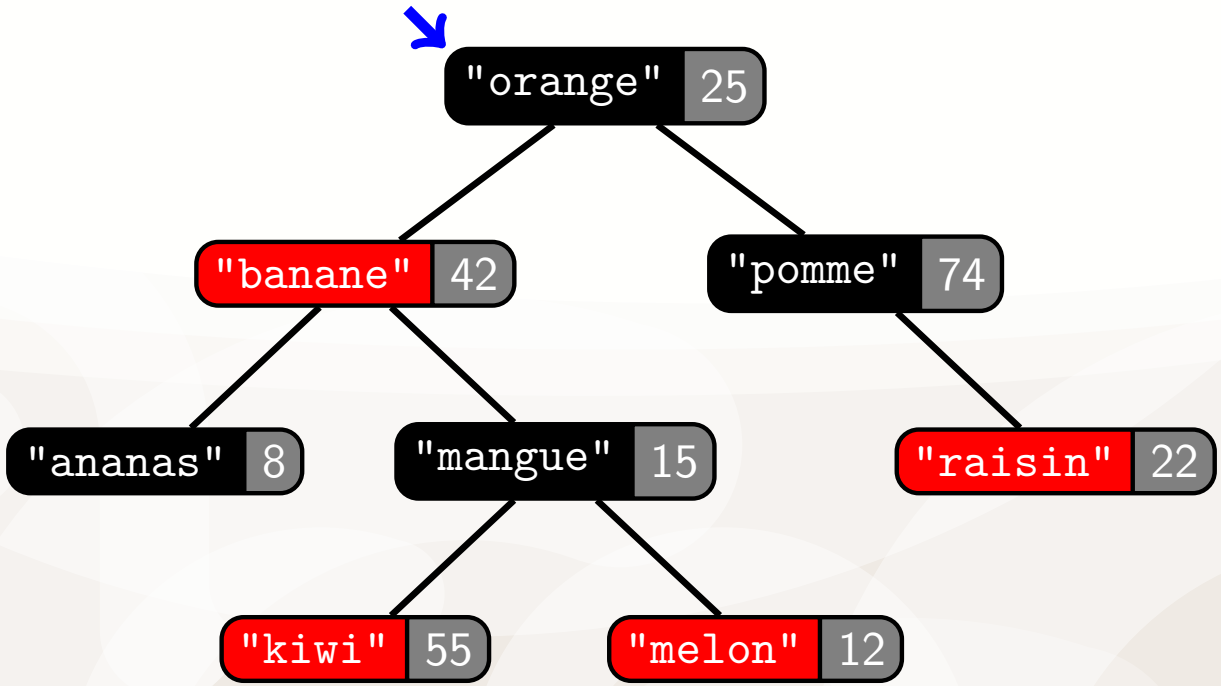


$\Rightarrow$ $O(1)$ **opérations**

Accès à un élément par sa valeur

Il faut parcourir la collection jusqu'à trouver l'élément.

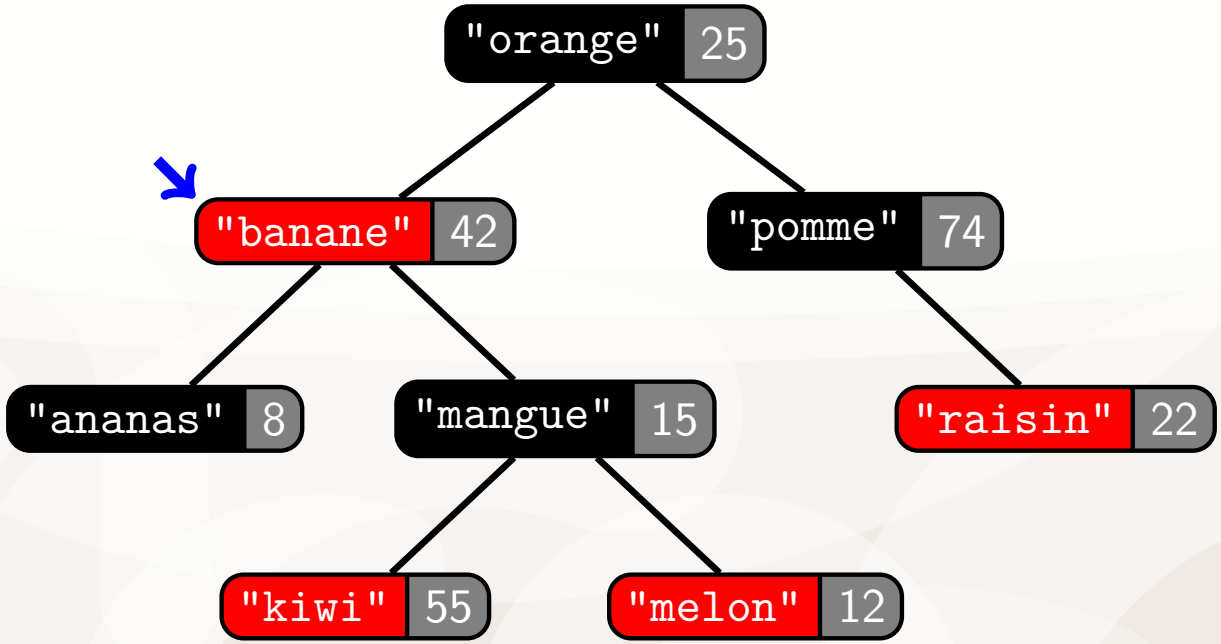
```
for(auto it=m.begin(); it!=m.end(); it++){
  if(it->second == val){/*trouvé*/}
}
```



Accès à un élément par sa valeur

Il faut parcourir la collection jusqu'à trouver l'élément.

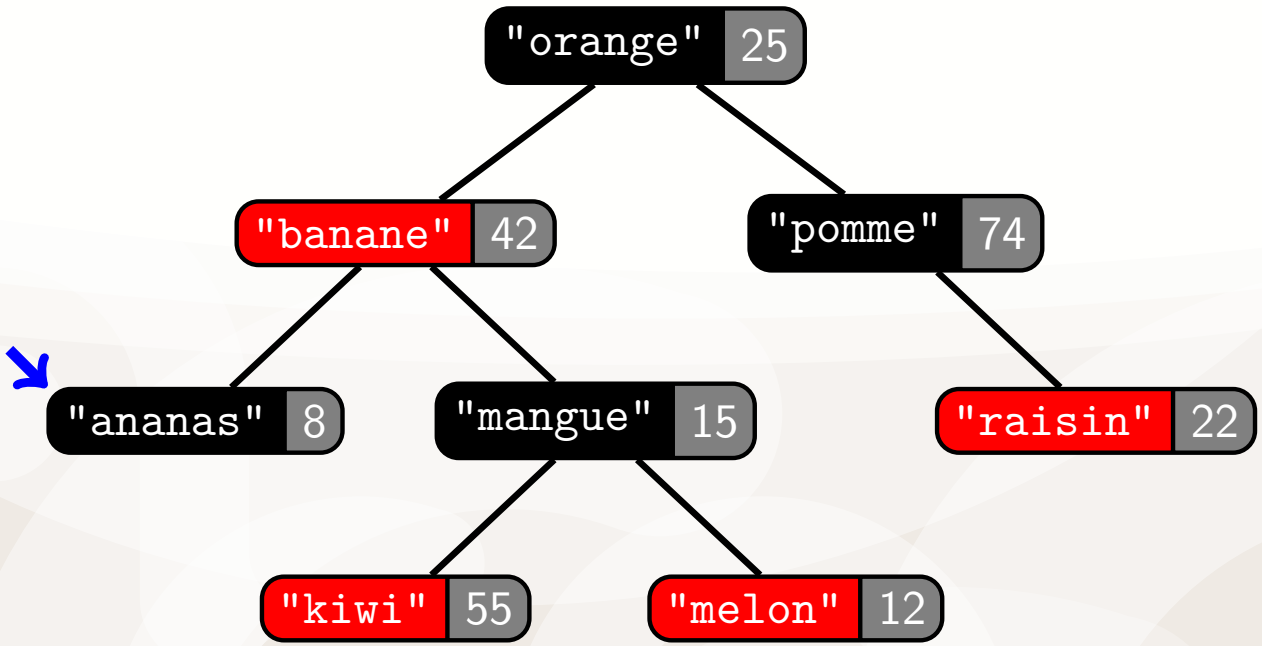
```
for(auto it=m.begin(); it!=m.end(); it++){
  if(it->second == val){/*trouvé*/}
}
```



Accès à un élément par sa valeur

Il faut parcourir la collection jusqu'à trouver l'élément.

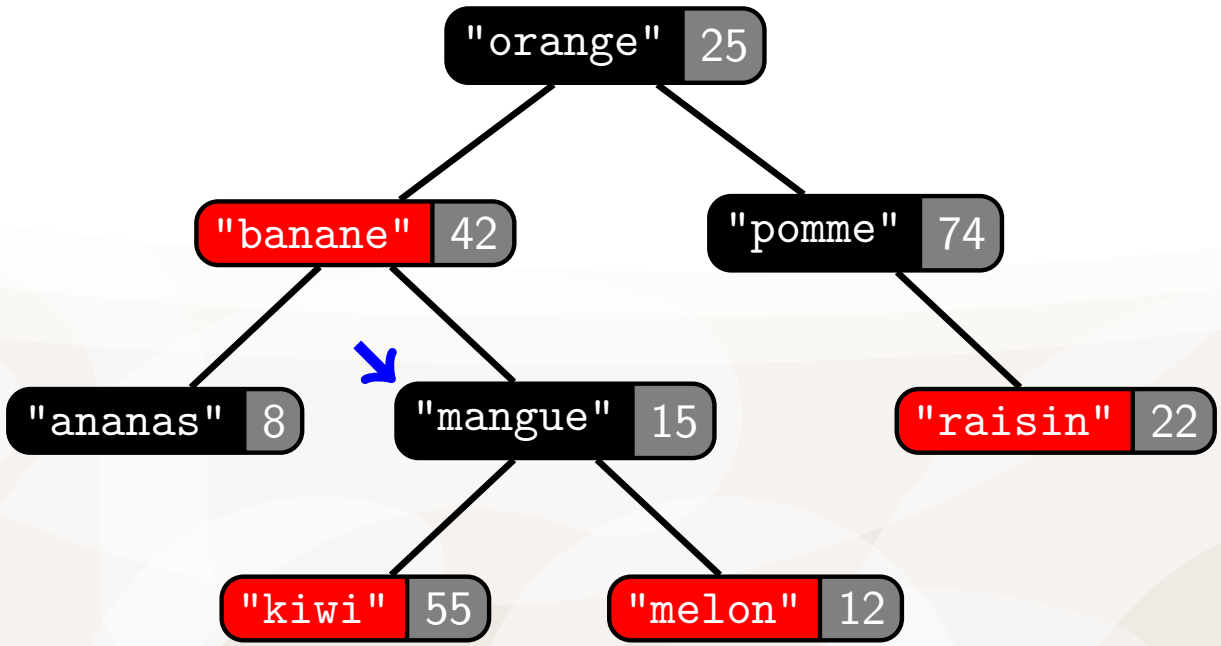
```
for(auto it=m.begin(); it!=m.end(); it++){
  if(it->second == val){/*trouvé*/}
}
```



Accès à un élément par sa valeur

Il faut parcourir la collection jusqu'à trouver l'élément.

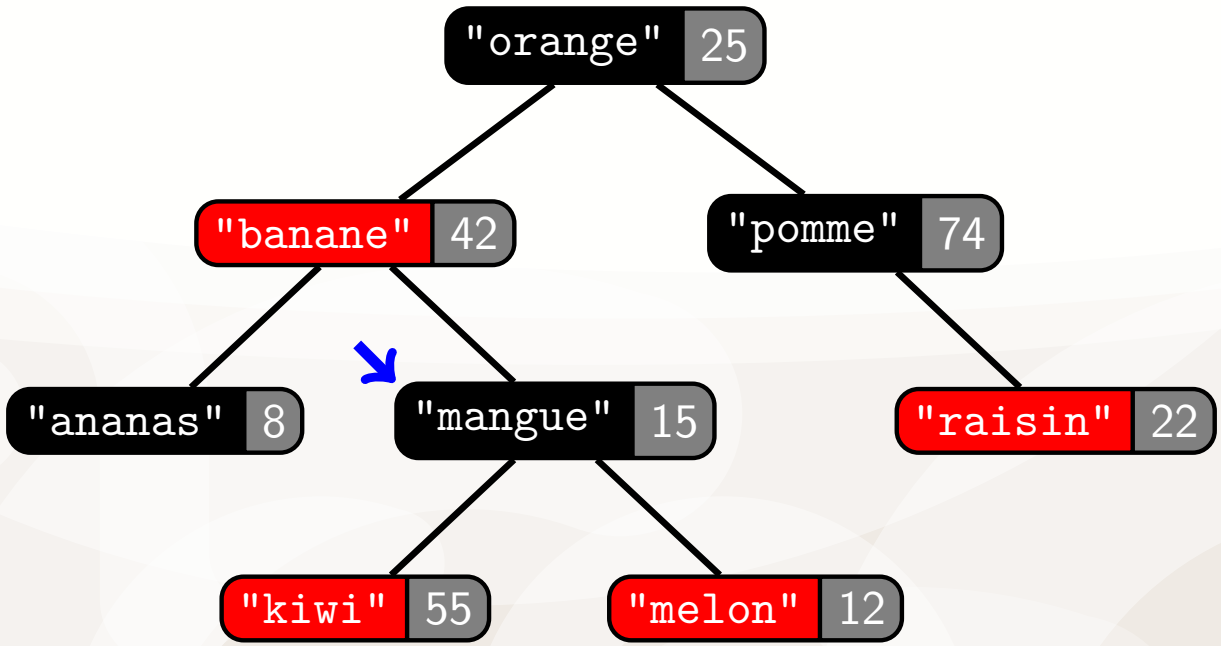
```
for(auto it=m.begin(); it!=m.end(); it++){
  if(it->second == val){/*trouvé*/}
}
```



Accès à un élément par sa valeur

Il faut parcourir la collection jusqu'à trouver l'élément.

```
for(auto it=m.begin(); it!=m.end(); it++){
  if(it->second == val){/*trouvé*/}
}
```



$\Rightarrow$ n **opérations**

Insertion/suppression

- `map` : besoin de réorganiser l'arbre

$\Rightarrow O(\log n)$ opérations

Insertion/suppression

- `map` : besoin de réorganiser l'arbre

  $\Rightarrow O(\log n)$ opérations

- `unordered_map` : besoin de calculer le haché

  $\Rightarrow O(1)$ opérations

`set` vs. `unordered_set`

- `set` : équivalent d'une `map` où les clés sont les valeurs elles-mêmes
  $\Rightarrow$ valeurs (uniques) triées dans un abre binaire de recherche

- `unordered_set` : équivalent d'une `unordered_map` où les clés sont les valeurs elles-mêmes
  $\Rightarrow$ valeurs (uniques) organisées en fonction de leur haché

Résumé

Résumé

		Accès par position	Accès par clé	Accès par valeur	Insertion	Suppression	Trié par
séquences	`vector`	$O(1)$	–	$O(n)$	$O(n)$	$O(n)$	ordre d'insertion
	`list`	$O(n)$	–	$O(n)$	$O(1)$	$O(1)$	
associatifs	`map`	–	$O(\log n)$	$O(n)$	$O(\log n)$	$O(\log n)$	clé
	`unordered_map`	–	$O(1)$	$O(n)$	$O(1)$	$O(1)$	X
	`set`	–	$O(\log n)$		$O(\log n)$	$O(\log n)$	valeur
	`unordered_set`	–	$O(1)$		$O(1)$	$O(1)$	X