ECE PARIS
ÉCOLE D'INGÉNIEURS

Robin FERCOQ
2018-2019

Conception et Programmation Orientée Objet C++

POO - C++

Sommaire général du semestre

COURS	Semaine suivante →	**TPs**
1. *Intro, concepts, 1 exemple*		1. *Organisation objet des données*
2. *Modélisation objet / UML*		2. *Diagrammes de classe UML*
3. *C++ pratique 1*		3. *C++ pratique, E/S, string, vector*
4. *C++ pratique 2*		4. *C++ pratique, type &, surcharge*
5. *Classes & C++ : bases*		5. *Date : une classe simple en C++*
6. *Classes & C++ : compléments*		6. *UML et C++, associations*
7. Conteneurs & C++ : la STL		7. *Gestion de collections complexes*
8. *Héritage / polymorphisme*		8. *Collections polymorphes*
9. *Modèles objets avancés*		9. *Modèle composite et graphismes*
10. *Exceptions, flots, fichiers ...*		10. *Persistance / fichiers / except.*
11. *Templates côté développeur*		11. *Développement de templates*
12. *Gestion mémoire / smarts ptrs*		12. *Soutenance de **projet** ...*

Conteneurs & C++ : la STL

COURS 7

A) Structures de données & STL
B) Itérateurs, parcours, algos
C) Conteneurs séquentiels
D) Piles et files
E) Conteneurs ensemblistes : set
F) Conteneurs associatifs : map
G) Arbre Binaire de Recherche
H) Table de hachage

COURS 7

A) Structures de données & STL
B) Itérateurs, parcours, algos
C) Conteneurs séquentiels
D) Piles et files
E) Conteneurs ensemblistes : set
F) Conteneurs associatifs : map
G) Arbre Binaire de Recherche
H) Table de hachage

Structures de données & STL

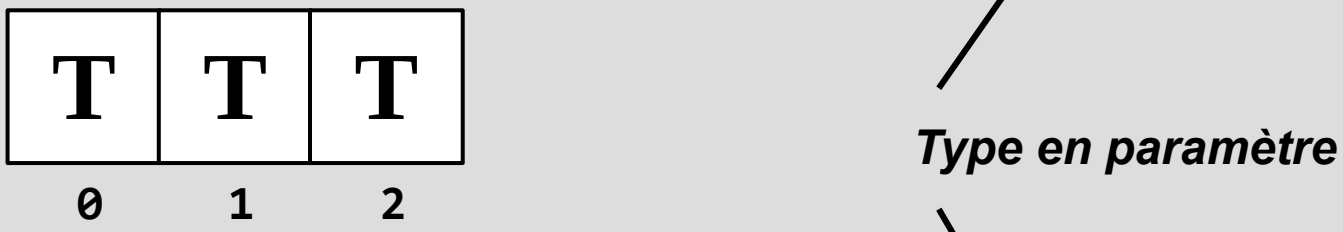
Structures de données & STL

La Standard Template Library fournit

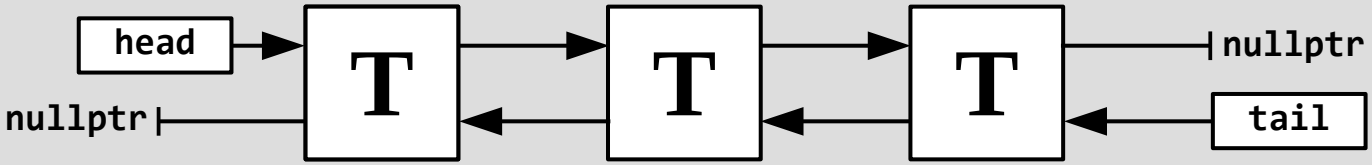
- *Des classes conteneurs génériques (templates)*
  générique/template = type paramétrable
  les conteneurs pourront contenir n'importe quel type
- *Des itérateurs pour désigner des emplacements*
  itérateur = pointeur amélioré spécialement conçu pour parcourir/désigner les « cases » des conteneurs
- *Des méthodes et algorithmes usuels pour*
  insertion / suppression / recherche / tri
- *C'est un standard ISO, plusieurs implémentations existent*
  (GNU)libstdc++ / (LLVM)libc++ / Microsoft STL / Apache stdcxx ...
- *Le succès du C++ comme langage industriel repose en grande partie sur la STL: productivité, fiabilité, performance*

Structures de données & STL

- *Qu'est-ce qu'une classe conteneur générique ? (generic/template container class)*
- *La classe générique* `std::vector< T >`

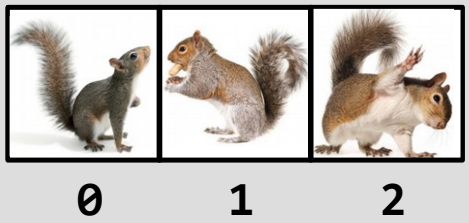


- *La classe générique* `std::list< T >`



Structures de données & STL

- *Qu'est-ce qu'une classe conteneur générique ? (generic/template container class)*
- *La classe concrète* `std::vector<Ecureuil>`

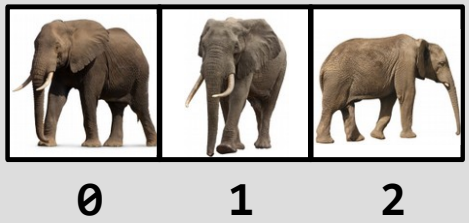


Type 16 octets

- *La classe concrète* `std::list<Ecureuil>`

Structures de données & STL

- *Qu'est-ce qu'une classe conteneur générique ? (generic/template container class)*
- *La classe concrète* `std::vector<Elephant>`



Type 4000000 octets

- *La classe concrète* `std::list<Elephant>`

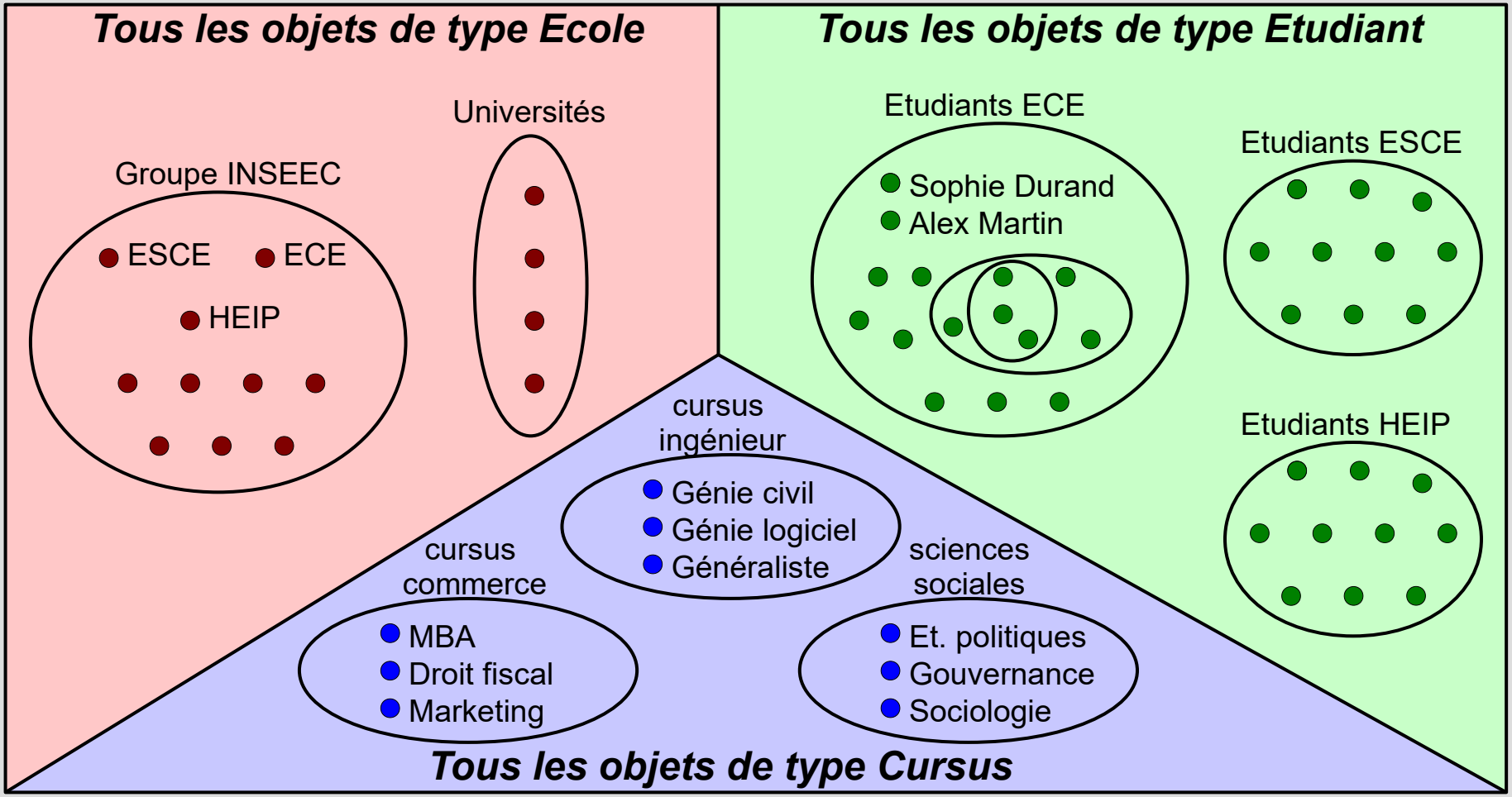
Structures de données & STL

- *Qu'est-ce qu'une classe conteneur générique ?*
  (generic/template container class)
- *C'est une classe « paramétrée en type »*
  Techniquement en C++ : classe ***template*** *(cours 11)*
- *Elle contient une collection d'éléments de type arbitraire ! Exemple :* `std::vector< T >`
- *Attention cependant, un même conteneur générique ne contient qu'un seul type à la fois, on ne mélange pas écureuils et éléphants !*
  (sauf polymorphisme, cours 8)

- *Ça sert à quoi ?* ***Ça répond à des besoins...***

Structures de données & STL

- *On a besoin de séparer les objets en groupes*
- *Appartenance aux classes : pas assez fin*



Structures de données & STL

- *Ces groupes ou « collections » doivent être*
  - *Accédés : 1 élément à la fois*
  - *Parcourus : traitements collectifs*
  - *Agrandis : ajout d'élément(s)*
  - *Diminués : retraits d'élément(s)*
- *Ils doivent permettre des opérations*

  Savoir taille / Trier / Trouver ...
- *Ils doivent pouvoir apporter des garanties*

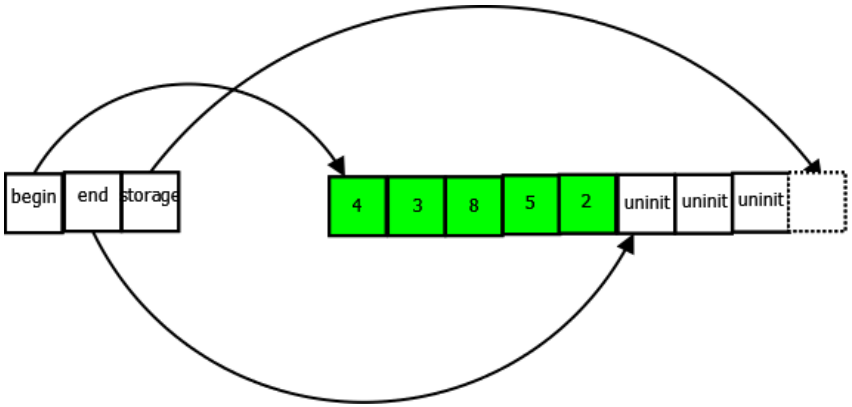
  Unicité / Performance / Stabilité

Structures de données & STL

- *Différentes façons d'organiser les données :*

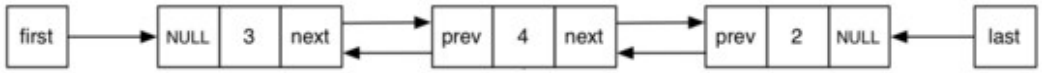
Structures de données

vecteur

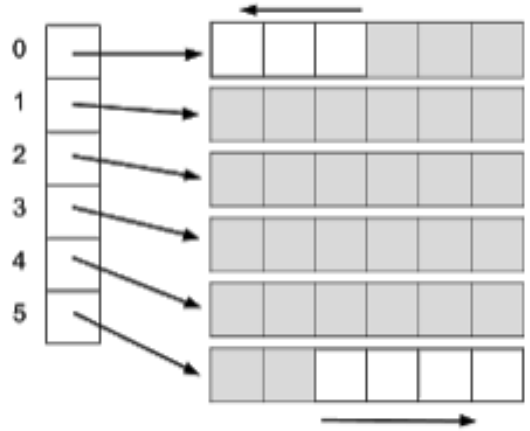


liste simple chaînage

liste double chaînage



file à 2 entrées



arbre binaire de recherche

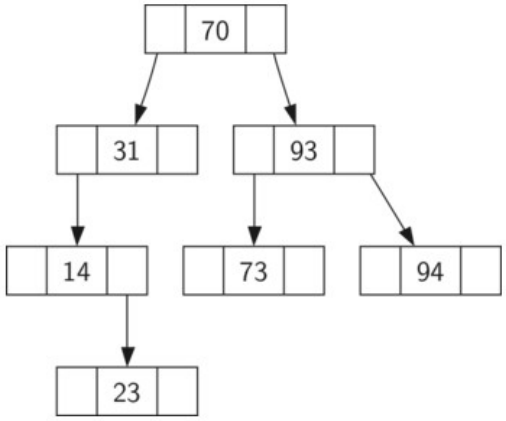
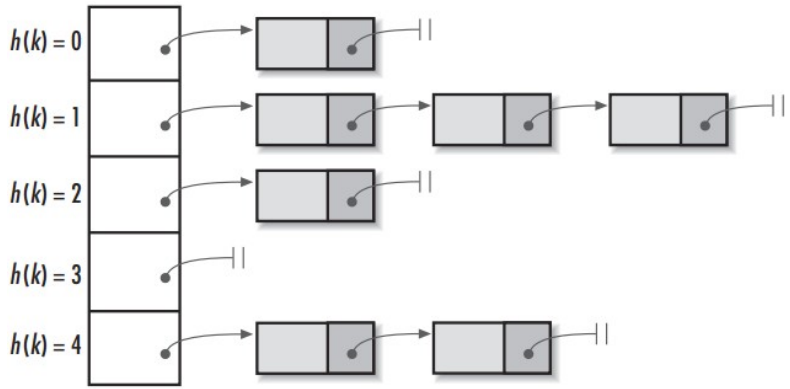


table de hachage



Structures de données & STL

- *Différentes façons d'accéder aux données :*

Conteneurs	***Structure(s) utilisée(s)***
`vector`	vecteur
`list`	liste double chaînage
`forward_list`	liste simple chaînage
`deque`	file à 2 entrées
`stack`	file à 2 entrées ou liste à double chaînage ou vecteur
`queue`	file à 2 entrées ou liste à double chaînage
`set`	arbre binaire de recherche
`unordered_set`	table de hachage
`map`	arbre binaire de recherche avec éléments clé/valeur
`unordered_map`	table de hachage avec éléments clé/valeur

Structures de données & STL

- *En fonction des **besoins d'accès** et des **contraintes et avantages** des différentes structures on choisit un de ces conteneurs...*

`vector`	vecteur
`list`	liste double chaînage
`forward_list`	liste simple chaînage
`deque`	file à 2 entrées
`stack`	file à 2 entrées ou liste à double chaînage ou vecteur
`queue`	file à 2 entrées ou liste à double chaînage
`set`	arbre binaire de recherche
`unordered_set`	table de hachage
`map`	arbre binaire de recherche avec éléments clé/valeur
`unordered_map`	table de hachage avec éléments clé/valeur

Structures de données & STL

- *En fonction des **besoins d'accès** et des **contraintes et avantages** des différentes structures on choisit un de ces conteneurs...*

vector	vecteur
list	liste double chaînage
forward_list	liste simple chaînage
deque	file à 2 entrées
stack	file à 2 entrées ou liste à double chaînage ou vecteur
queue	file à 2 entrées ou liste à double chaînage
set	arbre binaire de recherche
unordered_set	table de hachage
map	arbre binaire de recherche avec éléments clé/valeur
unordered_map	table de hachage avec éléments clé/valeur

Accès efficace au rang i
Insertion au milieu inefficace

Accès inefficace au rang i
Insertion au milieu efficace
(à condition d'y être)

Structures de données & STL

- *Le gros avantage des conteneurs **génériques** : pas besoin de recoder pour chaque type* **T** *!*

	int	double	string	Ecureuil	Elephant*	list<string>	...
vector	✔	✔	✔	✔	✔	✔	✔
list	✔	✔	✔	✔	✔	✔	✔
forward_list	✔	✔	✔	✔	✔	✔	✔
deque	✔	✔	✔	✔	✔	✔	✔
stack	✔	✔	✔	✔	✔	✔	✔
queue	✔	✔	✔	✔	✔	✔	✔
set	✔	✔	✔	✔	✔	✔	✔
unordered_set	✔	✔	✔	✔	✔	✔	✔
map<Key,Value>	✔[2]	✔[2]	✔[2]	✔[2]	✔[2]	✔[2]	✔[2]
unordered_map<K,V>	✔[2]	✔[2]	✔[2]	✔[2]	✔[2]	✔[2]	✔[2]

Structures de données & STL

- Les méthodes utilisables dépendent des conteneurs...

		Sequence containers					Associative containers				Unordered associative containers				Container adaptors		
Header		<array>	<vector>	<deque>	<forward_list>	<list>	<set>		<map>		<unordered_set>		<unordered_map>		<stack>	<queue>	
Container		array	vector	deque	forward_list	list	set	multiset	map	multimap	unordered_set	unordered_multiset	unordered_map	unordered_multimap	stack	queue	priority_queue
	(constructor)	(implicit)	vector	deque	forward_list	list	set	multiset	map	multimap	unordered_set	unordered_multiset	unordered_map	unordered_multimap	stack	queue	priority_queue
	(destructor)	(implicit)	~vector	~deque	~forward_list	~list	~set	~multiset	~map	~multimap	~unordered_set	~unordered_multiset	~unordered_map	~unordered_multimap	~stack	~queue	~priority_queue
	operator=	(implicit)	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=
	assign		assign	assign	assign	assign											
Iterators	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin			
	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin			
	end	end	end	end	end	end	end	end	end	end	end	end	end	end			
	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend			
	rbegin	rbegin	rbegin	rbegin		rbegin	rbegin	rbegin	rbegin	rbegin							
	crbegin	crbegin	crbegin	crbegin		crbegin	crbegin	crbegin	crbegin	crbegin							
	rend	rend	rend	rend		rend	rend	rend	rend	rend							
	crend	crend	crend	crend		crend	crend	crend	crend	crend							
Element access	at	at	at	at					at				at				
	operator[]	operator[]	operator[]	operator[]					operator[]				operator[]				
	data	data	data														
	front	front	front	front	front	front										front	top
	back	back	back	back		back									top	back	
Capacity	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty
	size	size	size	size		size	size	size	size	size	size	size	size	size	size	size	size
	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size			
	resize		resize	resize	resize	resize											
	capacity		capacity								bucket_count	bucket_count	bucket_count	bucket_count			
	reserve		reserve								reserve	reserve	reserve	reserve			
	shrink_to_fit		shrink_to_fit	shrink_to_fit													
Modifiers	clear		clear	clear	clear	clear	clear	clear	clear	clear	clear	clear	clear	clear			
	insert		insert	insert	insert_after	insert	insert	insert	insert	insert	insert	insert	insert	insert			
	insert_or_assign								insert_or_assign				insert_or_assign				
	emplace		emplace	emplace	emplace_after	emplace	emplace	emplace	emplace	emplace	emplace	emplace	emplace	emplace			
	emplace_hint						emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint			
	try_emplace								try_emplace				try_emplace				
	erase		erase	erase	erase_after	erase	erase	erase	erase	erase	erase	erase	erase	erase			
	push_front			push_front	push_front	push_front											
	emplace_front			emplace_front	emplace_front	emplace_front											
	pop_front			pop_front	pop_front	pop_front										pop	pop
	push_back		push_back	push_back		push_back									push	push	push
	emplace_back		emplace_back	emplace_back		emplace_back									emplace	emplace	emplace
	pop_back		pop_back	pop_back		pop_back									pop		
	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap
	merge				merge	merge	merge	merge	merge	merge	merge	merge	merge	merge			
	extract						extract	extract	extract	extract	extract	extract	extract	extract			
List operations	splice				splice_after	splice											
	remove				remove	remove											
	remove_if				remove_if	remove_if											
	reverse				reverse	reverse											
	unique				unique	unique											
	sort				sort	sort											
Lookup	count						count	count	count	count	count	count	count	count			
	find						find	find	find	find	find	find	find	find			
	contains						contains	contains	contains	contains	contains	contains	contains	contains			
	lower_bound						lower_bound	lower_bound	lower_bound	lower_bound							
	upper_bound						upper_bound	upper_bound	upper_bound	upper_bound							
	equal_range						equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range			
Observers	key_comp						key_comp	key_comp	key_comp	key_comp							
	value_comp						value_comp	value_comp	value_comp	value_comp							
	hash_function										hash_function	hash_function	hash_function	hash_function			
	key_eq										key_eq	key_eq	key_eq	key_eq			
Allocator	get_allocator		get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator			

Structures de données & STL

- *Ces méthodes sont complétées par des **fonctions génériques** (type `T` quelconque) qui implémentent des **algorithmes** usuels : trier, min, max, compter, trouver ...*
- *Les algorithmes utilisables dépendent des conteneurs Par exemple l'algorithme de tri* **`std::sort`** *nécessite un conteneur avec accès aléatoire :* **`operator[]`**
- *Les listes chaînée n'ont pas cet accès, mais elle ont une **méthode** de tri (au final ça revient au même...)*
- *Ces fonctions sont dans* **`#include <algorithm>`**
- *Il y en a beaucoup ! Autant à ne pas re-coder...*
- *Gain en productivité, fiabilité, performance*

Structures de données & STL

Non-modifying sequence operations
Defined in header <algorithm>
all_of (C++11)
any_of (C++11)
none_of (C++11)
for_each
for_each_n (C++17)
count
count_if
mismatch
find
find_if
find_if_not (C++11)
find_end
find_first_of
adjacent_find
search
search_n

Modifying sequence operations
Defined in header <algorithm>
copy
copy_if (C++11)
copy_n (C++11)
copy_backward
move (C++11)
move_backward (C++11)
fill
fill_n
transform
generate
generate_n
remove
remove_if
remove_copy
remove_copy_if
replace
replace_if
replace_copy
replace_copy_if
swap
swap_ranges
iter_swap
reverse
reverse_copy
rotate
rotate_copy
shift_left
shift_right (C++20)
random_shuffle (until C++17)
shuffle (C++11)
sample (C++17)
unique
unique_copy

Partitioning operations
Defined in header <algorithm>
is_partitioned (C++11)
partition
partition_copy (C++11)
stable_partition
partition_point (C++11)

Sorting operations
Defined in header <algorithm>
is_sorted (C++11)
is_sorted_until (C++11)
sort
partial_sort
partial_sort_copy
stable_sort
nth_element

Binary search operations (on sortec
Defined in header <algorithm>
lower_bound
upper_bound
binary_search
equal_range

Other operations on sorted ranges
Defined in header <algorithm>
merge
inplace_merge

Set operations (on sorted ranges)
Defined in header <algorithm>
includes
set_difference
set_intersection
set_symmetric_difference
set_union

Heap operations
Defined in header <algorithm>
is_heap (C++11)
is_heap_until (C++11)
make_heap
push_heap
pop_heap
sort_heap

Minimum/maximum operations
Defined in header <algorithm>
max
max_element
min
min_element
minmax (C++11)
minmax_element (C++11)
clamp (C++17)

Comparison operations
Defined in header <algorithm>
equal
lexicographical_compare
compare_3way (C++20)
lexicographical_compare_3way

Permutation operations
Defined in header <algorithm>
is_permutation (C++11)
next_permutation
prev_permutation

Numeric operations
Defined in header <numeric>
iota (C++11)
accumulate
inner_product
adjacent_difference
partial_sum
reduce (C++17)
exclusive_scan (C++17)
inclusive_scan (C++17)
transform_reduce (C++17)
transform_exclusive_scan (C++17)
transform_inclusive_scan (C++17)

Operations on uninitialized memory
Defined in header <memory>
uninitialized_copy
uninitialized_copy_n (C++11)
uninitialized_fill
uninitialized_fill_n
uninitialized_move (C++17)
uninitialized_move_n (C++17)
uninitialized_default_construct
uninitialized_default_construct_n
uninitialized_value_construct
uninitialized_value_construct_n
destroy_at (C++17)
destroy (C++17)
destroy_n (C++17)

C library
Defined in header <cstdlib>
qsort
bsearch

Les algorithmes de la STL

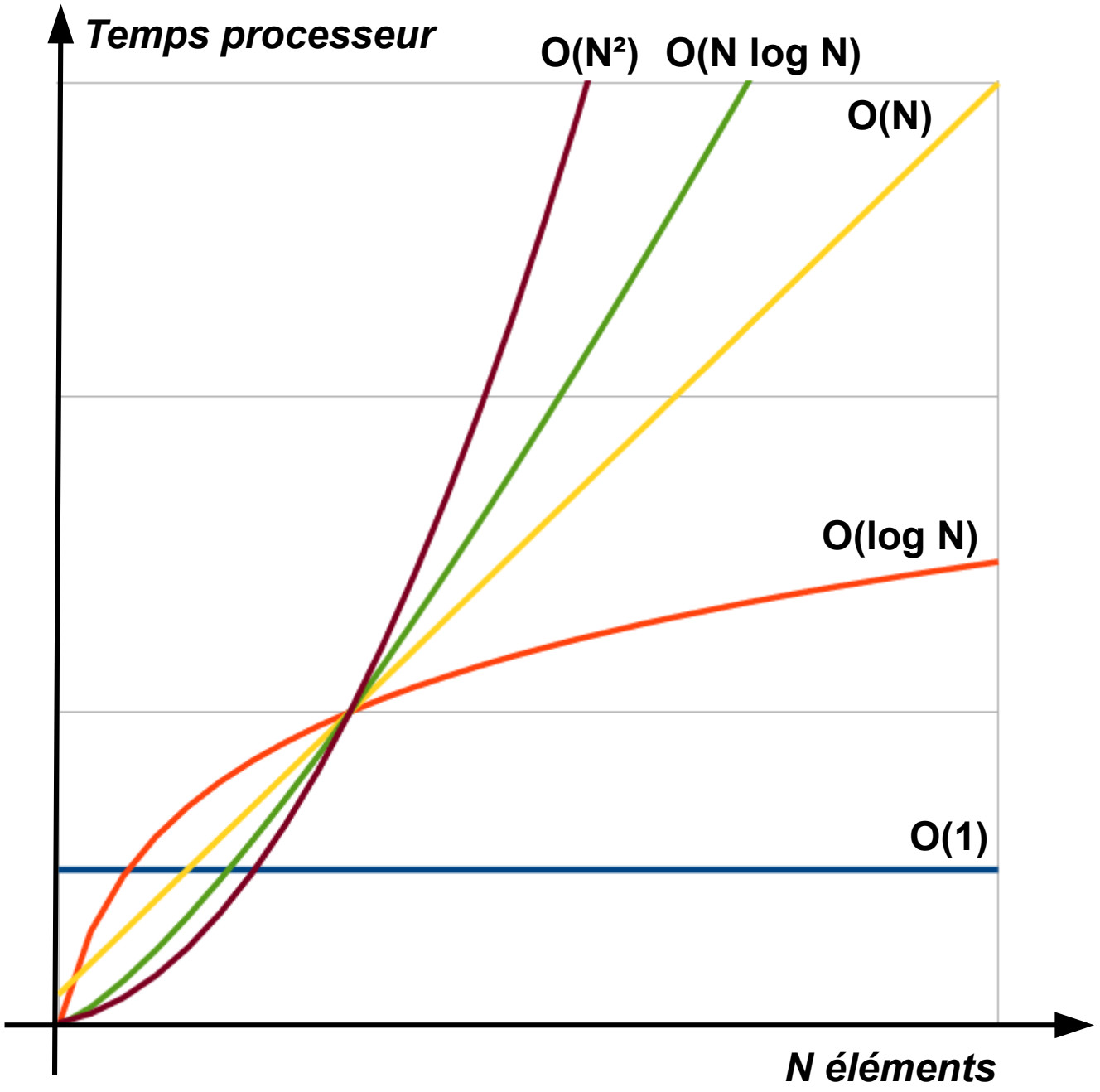
Structures de données & STL

- *Utiliser une méthode ou un algorithme a un coût*
- *Ce coût processeur (**temps d'exécution**) est précisé dans la documentation au paragraphe **complexité** (**complexity**)*
- *Le temps d'exécution exact dépend de la machine, des options d'optimisation, de la fragmentation des mémoires caches... La norme ne peut pas entrer dans ces détails*
- *L'indication de **complexité temporelle** est donnée comme une **proportionnalité** entre un nombre N d'éléments impliqués et le temps d'exécution...*

Structures de données & STL

- *Cette **proportionnalité** entre un nombre **N** d'éléments impliqués et le **temps d'exécution**... est exprimée en **notation grand O** :*
  - *O(1)* *temps constant*
  - *O(log N)* *temps logarithmique*
  - *O(N)* *temps linéaire*
  - *O(N log N)* *temps log-linéaire*
  - *O(N²)* *temps quadratique*

Structures de données & STL



Structures de données & STL

Exemples de complexités

- *L'insertion au milieu d'un vecteur implique de décaler N/2 éléments :* `insert` *est de complexité O(N) temps linéaire pour un vecteur*
- *L'insertion au milieu d'une liste juste après un élément qu'on est en train de visiter implique juste d'ajouter un maillon, ce temps ne dépend pas du nombre N d'éléments :* `insert` *est de complexité O(1) temps constant pour une liste*
- *Les tris que les débutants codent sont en $O(N^2)$*
- *Les tris de la STL sont en moyenne O(N log N)*

Structures de données & STL

- *On n'aurait pas besoin de toutes ces complications si une seule structure de donnée était tout le temps « la meilleure » !*
- ***Souvent** la structure par cases contiguës en mémoire avec ré-allocations est la meilleure : ceci correspond au* **`std::vector`** *qu'on utilisera « par défaut » dans les situations usuelles parce qu'à l'usage c'est un bon compromis*
- *Parfois on a **beaucoup d'insertions/délétions** ou alors on va **souvent chercher par valeur**, **alors d'autres conteneurs sont préférables...***

Structures de données & STL

- *Rappel technique : n'oubliez pas les includes*
- *Chaque conteneur a le sien*
- *Les algorithmes sont dans algorithm*

```
#include <iostream>
#include <string>
#include <vector>
#include <list>
#include <deque>
#include <set>
#include <unordered_set>
#include <map>
#include <algorithm>
...
```

COURS 7

A) Structures de données & STL
B) Itérateurs, parcours, algos
C) Conteneurs séquentiels
D) Piles et files
E) Conteneurs ensemblistes : set
F) Conteneurs associatifs : map
G) Arbre Binaire de Recherche
H) Table de hachage

Itérateurs, parcours, algos

Itérateurs, parcours, algos

- *Pour faire fonctionner toute cette machinerie* ***générique*** *(le* ***même*** *code trie des Écureuils ou des Éléphants) on a besoin d'un mécanisme de* ***désignation*** *des « cases » plus évolué et plus souple que les indices ou que les pointeurs*
- *La nouvelle catégorie d'objets introduite s'appelle les* ***itérateurs***
- ***1 itérateur désigne une case***
- ***2 itérateurs désignent un intervalle***
- *En gros c'est comme une sorte de pointeur...*

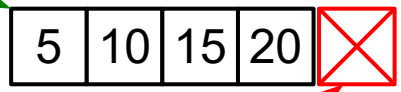
Itérateurs, parcours, algos

- *Chaque classe conteneur concrète a ses propres types d'itérateurs :*
- *La classe concrète* **`std::vector<int>`**
  a le type itérateur **`std::vector<int>::iterator`**
- *La classe concrète* **`std::set<Ecureuil>`**
  a le type itérateur **`std::set<Ecureuil>::iterator`**
- *Un itérateur est comme un pointeur sur un élément, il peut modifier l'élément sauf si on utilise la version const de l'itérateur :*
- *La classe concrète* **`std::list<Elephant>`**
  a le type itérateur constant (qui ne doit pas modifier)
  `std::list<Elephant>::const_iterator`

Itérateurs, parcours, algos

- *La méthode* **`begin()`** *retourne un itérateur sur le 1er élément*
- *La méthode* **`end()`** *retourne un itérateur sur un élément fictif qui est* ***après*** *le dernier élément*

```
std::vector<int> nbr{5, 10, 15, 20};

std::vector<int>::iterator it;
std::vector<int>::iterator debut;
std::vector<int>::iterator fin;

debut = nbr.begin();
fin = nbr.end();

for (it = debut; it!=fin; ++it)
    std::cout << *it << " ";
```

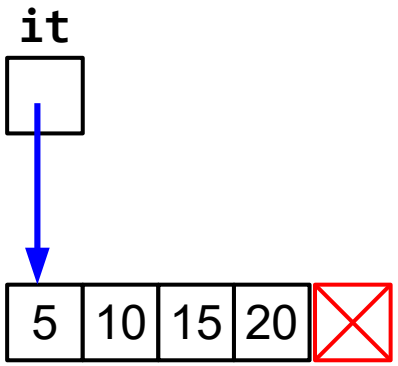



```
5 10 15 20
```

Itérateurs, parcours, algos

- *On peut faire **avancer** un itérateur (case suivante) en utilisant l'opérateur* **++**
- *On peut **accéder à l'élément** désigné par l'itérateur en le **déréférençant (préfixer avec ***)*

```
std::vector<int> nbr{5, 10, 15, 20};

std::vector<int>::iterator it;
std::vector<int>::iterator debut;
std::vector<int>::iterator fin;

debut = nbr.begin();

fin = nbr.end();

for (it = debut; it!=fin; ++it)
    std::cout << *it << " ";
```

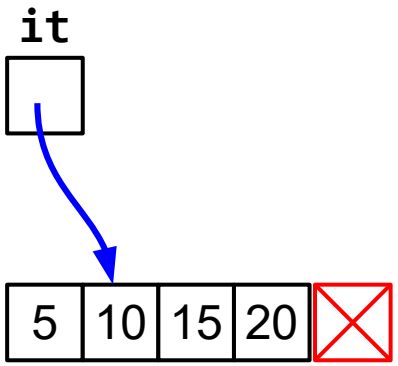



```
5 10 15 20
```

Itérateurs, parcours, algos

- *On peut faire **avancer** un itérateur (case suivante) en utilisant l'opérateur* **++**
- *On peut **accéder à l'élément** désigné par l'itérateur en le **déréférençant (préfixer avec ***)*

```
std::vector<int> nbr{5, 10, 15, 20};

std::vector<int>::iterator it;
std::vector<int>::iterator debut;
std::vector<int>::iterator fin;

debut = nbr.begin();

fin = nbr.end();

for (it = debut; it!=fin; ++it)
    std::cout << *it << " ";
```

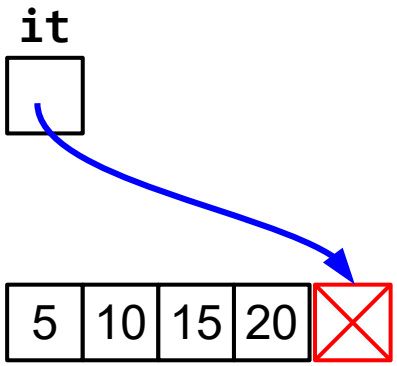



```
5 10 15 20
```

Itérateurs, parcours, algos

- *On peut faire **avancer** un itérateur (case suivante) en utilisant l'opérateur* **++**
- *On peut **accéder à l'élément** désigné par l'itérateur en le **déréférençant (préfixer avec ***)*

```
std::vector<int> nbr{5, 10, 15, 20};

std::vector<int>::iterator it;
std::vector<int>::iterator debut;
std::vector<int>::iterator fin;

debut = nbr.begin();

fin = nbr.end();

for (it = debut; it!=fin; ++it)
    std::cout << *it << " ";
```



Etc... Jusqu'à ce que la condition it!=fin soit fausse et on sort de la boucle

```
5 10 15 20
```

Itérateurs, parcours, algos

- *On n'est pas obligé de décomposer comme ça*
- *Une syntaxe normale pour parcourir :*

```
std::vector<int> nbr{5, 10, 15, 20};

for (std::vector<int>::iterator it = nbr.begin(); it!=nbr.end(); ++it)
    std::cout << *it << " ";
```

- **it autorise l'écriture (on peut modifier les éléments)*
- *Si on veut garantir que dans le corps de boucle on ne modifiera pas accidentellement les éléments on utilise la version const :*

```
std::vector<int> vec{5, 10, 15, 20};

for (std::vector<int>::const_iterator it = nbr.begin(); it!=nbr.end(); ++it)
    std::cout << *it << " ";
```

Itérateurs, parcours, algos

- *Pendant 15 ans les développeurs C++ ont usé leur clavier sur ces horreurs...*
- *Heureusement en C++11 on a* **auto**

```
std::vector<int> nbr{5, 10, 15, 20};

for (auto it = nbr.begin(); it!=nbr.end(); ++it)
    std::cout << *it << " ";
```

- *Mais ici on ne doit pas modifier les éléments... const auto ne marche pas, auto& non plus (on a déjà un « pointeur » avec l'itérateur) Il faut alors utiliser* **cbegin** *et* **cend**

```
std::vector<int> nbr{5, 10, 15, 20};

for (auto it = nbr.cbegin(); it!=nbr.cend(); ++it)
    std::cout << *it << " ";
```

Itérateurs, parcours, algos

- *Finalement voilà le C++ (presque) moderne*

```
std::vector<int> nbr{5, 10, 15, 20};


for (auto it = nbr.cbegin(); it!=nbr.cend(); ++it)
    std::cout << *it << " ";

for (auto it = nbr.begin(); it!=nbr.end(); ++it)
    *it *= 2;

for (auto it = nbr.cbegin(); it!=nbr.cend(); ++it)
    std::cout << *it << " ";
```

```
5 10 15 20
```

```
10 20 30 40
```

- *Pour comparaison, l'approche par indice :*

```
for (size_t i=0; i<nbr.size(); ++i)
    std::cout << nbr[i] << " ";

for (size_t i=0; i<nbr.size(); ++i)
    nbr[i] *= 2;

for (size_t i=0; i<nbr.size(); ++i)
    std::cout << nbr[i] << " ";
```

```
5 10 15 20
```

```
10 20 30 40
```

Itérateurs, parcours, algos

- *Et si une liste était mieux qu'un vecteur ?*

```
std::list <int> nbr{5, 10, 15, 20};

for (auto it = nbr.cbegin(); it!=nbr.cend(); ++it)
    std::cout << *it << " ";
```

```
5 10 15 20
```

```
for (auto it = nbr.begin(); it!=nbr.end(); ++it)
    *it *= 2;

for (auto it = nbr.cbegin(); it!=nbr.cend(); ++it)
    std::cout << *it << " ";
```

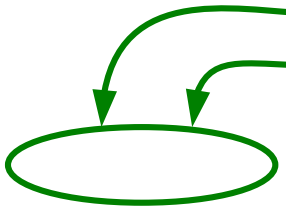
```
10 20 30 40
```

- *Pour comparaison, l'approche par indice casse !*

```
for (size_t i=0; i<nbr.size(); ++i)
    std::cout << nbr[i] << " ";

for (size_t i=0; i<nbr.size(); ++i)
    nbr[i] *= 2;

for (size_t i=0; i<nbr.size(); ++i)
    std::cout << nbr[i] << " ";
```

```
error:
no match for 'operator[]'
```

Itérateurs, parcours, algos



Au choix !

```
<int> nbr{5, 10, 15, 20};
```

```
for (auto it = nbr.cbegin(); it!=nbr.cend(); ++it)
    std::cout << *it << " ";

for (auto it = nbr.begin(); it!=nbr.end(); ++it)
    *it *= 2;

for (auto it = nbr.cbegin(); it!=nbr.cend(); ++it)
    std::cout << *it << " ";
```

Le même code

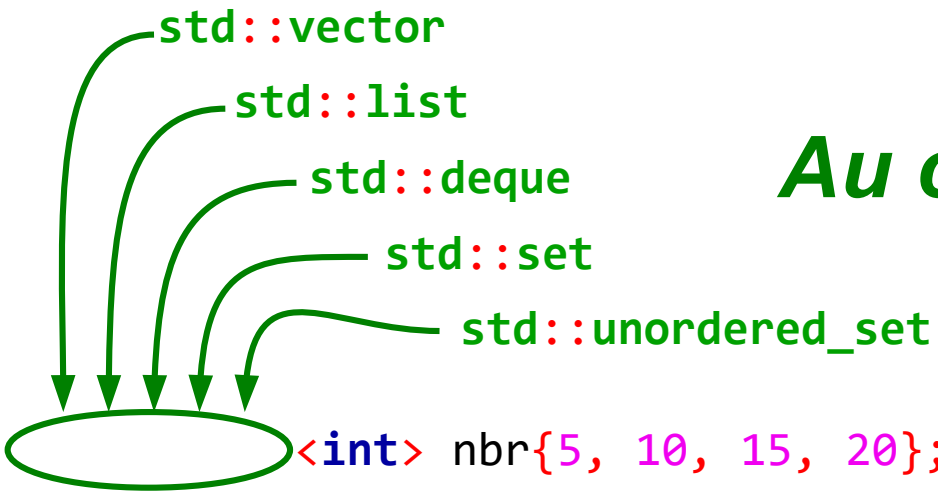
- *Les itérateurs permettent de **substituer** un conteneur à un autre sans casser le code : ils permettent une **abstraction** des algorithmes par rapport aux structures de données concrètes*
- *C'est le Graal du software, le découplage ultime*

Itérateurs, parcours, algos

- *Un tel résultat valait bien tous ces sacrifices...*
- *Désormais tous les algorithmes de la STL peuvent être codés sur la base des itérateurs : ils s'appliquent à plusieurs conteneurs à la fois (avec quelques restrictions selon les aptitudes de l'itérateur et du conteneur visé)*
- *C'est la raison pour laquelle les algorithmes et méthodes de la STL utilisent exclusivement des itérateurs*

Itérateurs, parcours, algos

- *Exemple d'algorithme non trivial rendu indépendant des structures de données*



Au choix !

```
<int> nbr{5, 10, 15, 20};

for (auto it = nbr.begin(); it != nbr.end(); )
{
    if (*it % 2 == 0)
        it = nbr.erase(it);
    else
        ++it;
}

for (auto it = nbr.cbegin(); it!=nbr.cend(); ++it)
    std::cout << *it << " ";
```

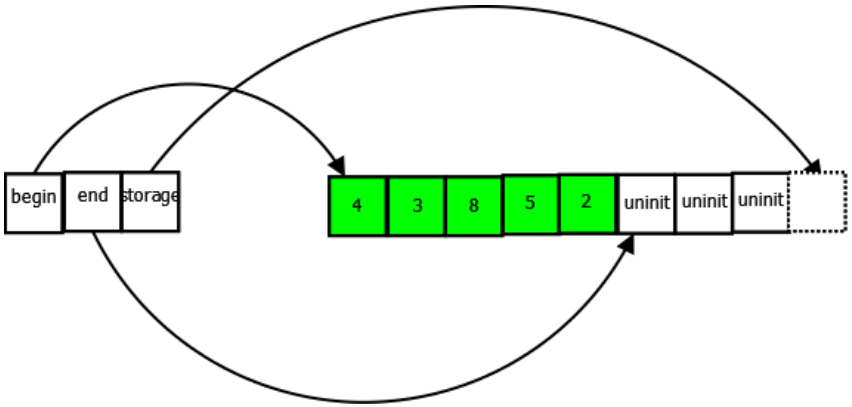
Ce code supprime les éléments pairs

```
5 15
```

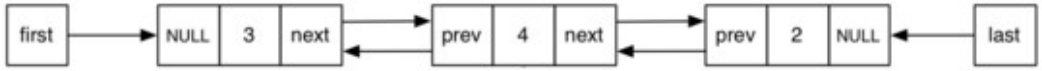
Itérateurs, parcours, algos

- *Le même code utilisateur peut fonctionner sur des « mécaniques » très différentes*

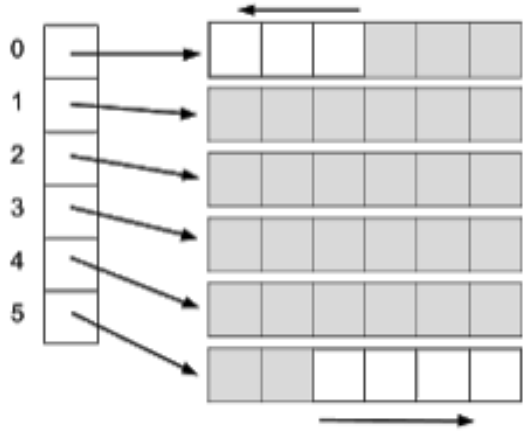
`std::vector`



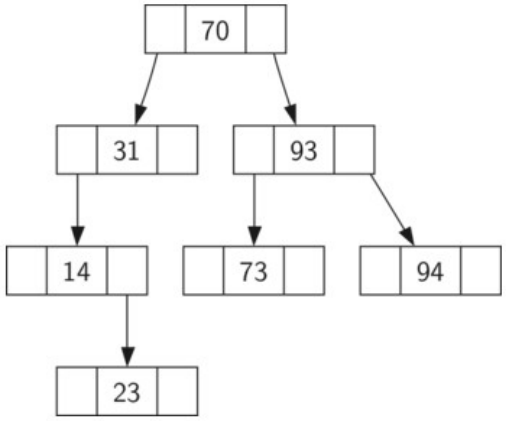
`std::list`



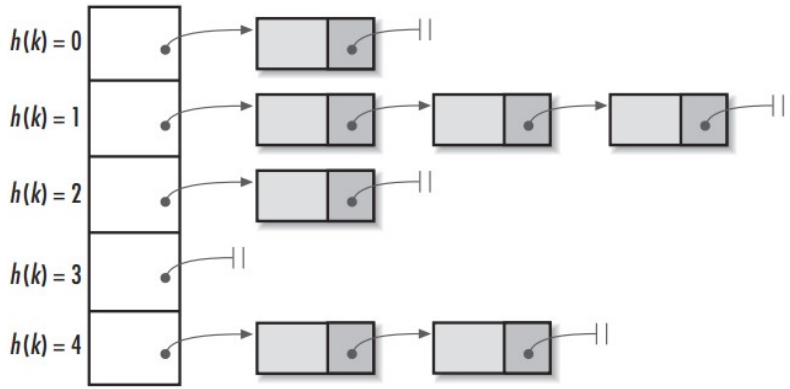
`std::deque`



`std::set`



`std::unordered_set`



Itérateurs, parcours, algos

- *En résumé les itérateurs fournissent un mécanisme **plus général** que `[i]` pour*
- *être utilisables avec tous les conteneurs*
- *servir d'interface de parcours (boucles)*
- *servir d'interface avec certaines méthodes (erase / insert / find)*
- *servir d'interface avec les algorithmes STL*

```
std::vector<int> nbr{10, 5, 20, 15};

std::sort(nbr.begin(), nbr.end());

for (auto it = nbr.cbegin(); it!=nbr.cend(); ++it)
    std::cout << *it << " ";
```

Trier !

```
5 10 15 20
```

Itérateurs, parcours, algos

- *Et si je veux trier à l'envers ou parcourir les éléments en sens inverse ?*
- *Utiliser les reverse iterators qui s'obtiennent en préfixant par r :*

```
std::vector<int> nbr{10, 5, 20, 15};

std::sort(nbr.rbegin(), nbr.rend());

for (auto it = nbr.cbegin(); it!=nbr.cend(); ++it)
    std::cout << *it << " ";
```

Tri inverse !

```
20 15 10 5
```

- *std::sort est en moyenne log-linéaire O(N log N)*
  trier 10^6 entiers *lui prend environ* ***20ms*** *à 1GHz*
  Le tri bulle que vous savez coder est $O(N^2)$
  il prendrait 10^{12} étapes, environ ***15 minutes*** *!*

Itérateurs, parcours, algos

- *Et si je veux trier des éléphants ?*
- *Il faudra fournir à l'algorithme un moyen de comparer 2 objets de type Elephant*
- *Soit en surchargeant operator<*
- *Soit en passant une fonction en paramètre*
- *Soit en passant un « foncteur » en paramètre*
- *Soit en utilisant une fonction lambda anonyme ...*
- *Voir lien ci dessus*
- *Dans tous les cas il faudra payer le prix de la permutation mémoire d'objets lourds. Il est préférable si possible de permuter des pointeurs*

Itérateurs, parcours, algos

- *Et si je veux juste parcourir les éléments et que je trouve la syntaxe des itérateurs trop lourde ?*
- *Utiliser la syntaxe « range-based for loop »*

```
std::vector<int> nbr{5, 10, 15, 20};

for (auto elem : nbr)
    std::cout << elem << " ";

for (auto& elem : nbr)
    elem *= 2;

for (const auto& elem : nbr)
    std::cout << elem << " ";
```

elem est une copie de chaque élément

```
5 10 15 20
```

elem est une référence de chaque élément

```
10 20 30 40
```

elem est une référence constante de chaque élément

- *C'est chic !*

COURS 7

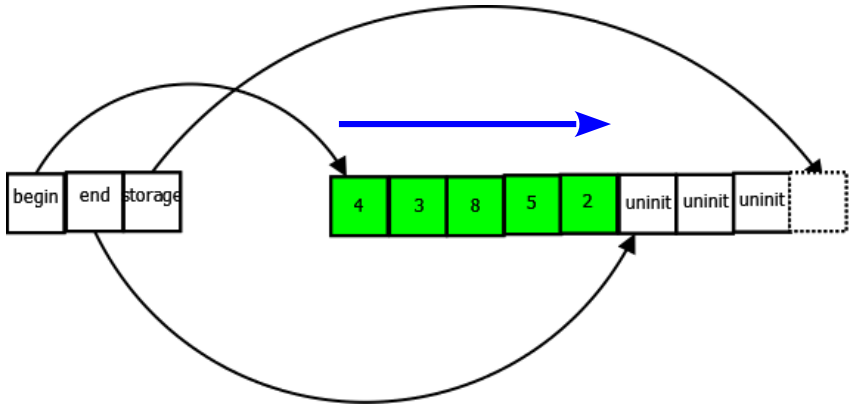
A) Structures de données & STL
B) Itérateurs, parcours, algos
C) Conteneurs séquentiels
D) Piles et files
E) Conteneurs ensemblistes : set
F) Conteneurs associatifs : map
G) Arbre Binaire de Recherche
H) Table de hachage

Conteneurs séquentiels

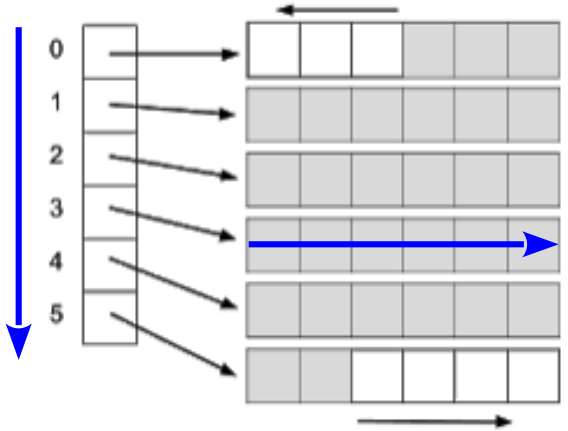
Conteneurs séquentiels

- *Les conteneurs « séquentiels » correspondent aux structures de données linéaires (1D)*

vector



deque = double-ended queue

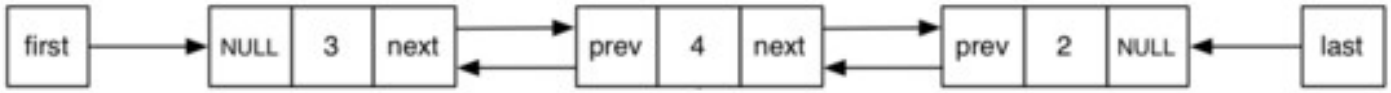


➔ *Les conteneurs séquentiels* ***vector*** *et* ***deque*** *sont organisés de façon* ***contiguë en mémoire*** *directement (vector) ou avec indirection (deque)*
=> ils disposent d'un ***accès "aléatoire" en*** `[i]`
efficace en O(1)

Conteneurs séquentiels

- *Les conteneurs « séquentiels » correspondent aux structures de données linéaires (1D)*

list



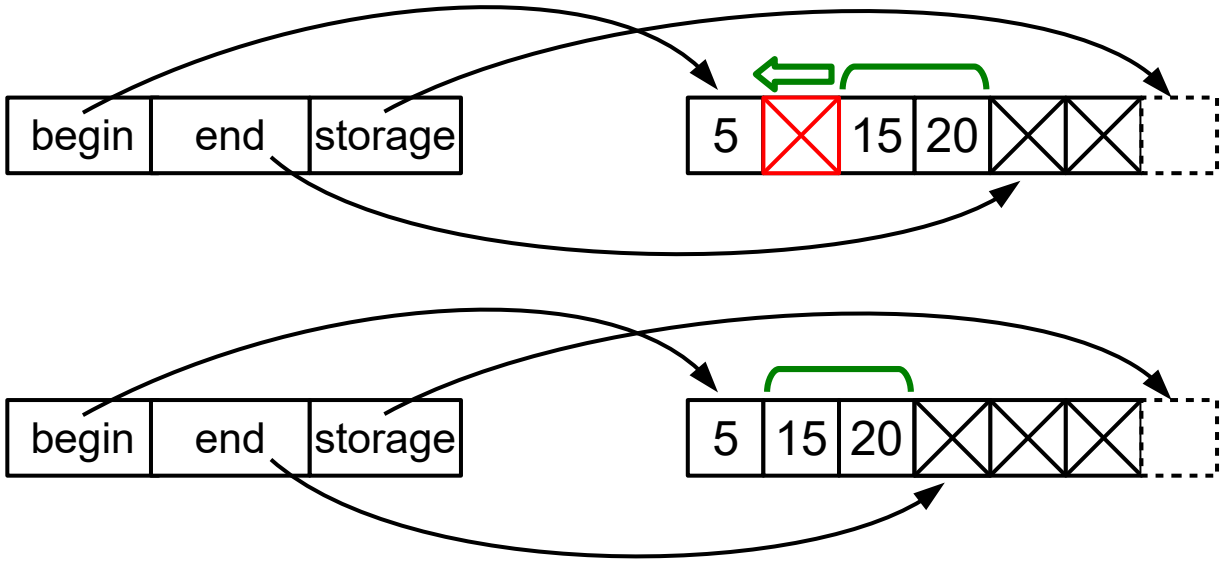
forward_list

- *Les conteneurs séquentiels **list** et **forward_list** relient les éléments par chaînage (pointeurs)*
  Pour accéder à un élément il faut parcourir
  => ils ne disposent pas d'un accès "aléatoire"
  *il n'y a **pas d'accès en** `[i]`*

Conteneurs séquentiels

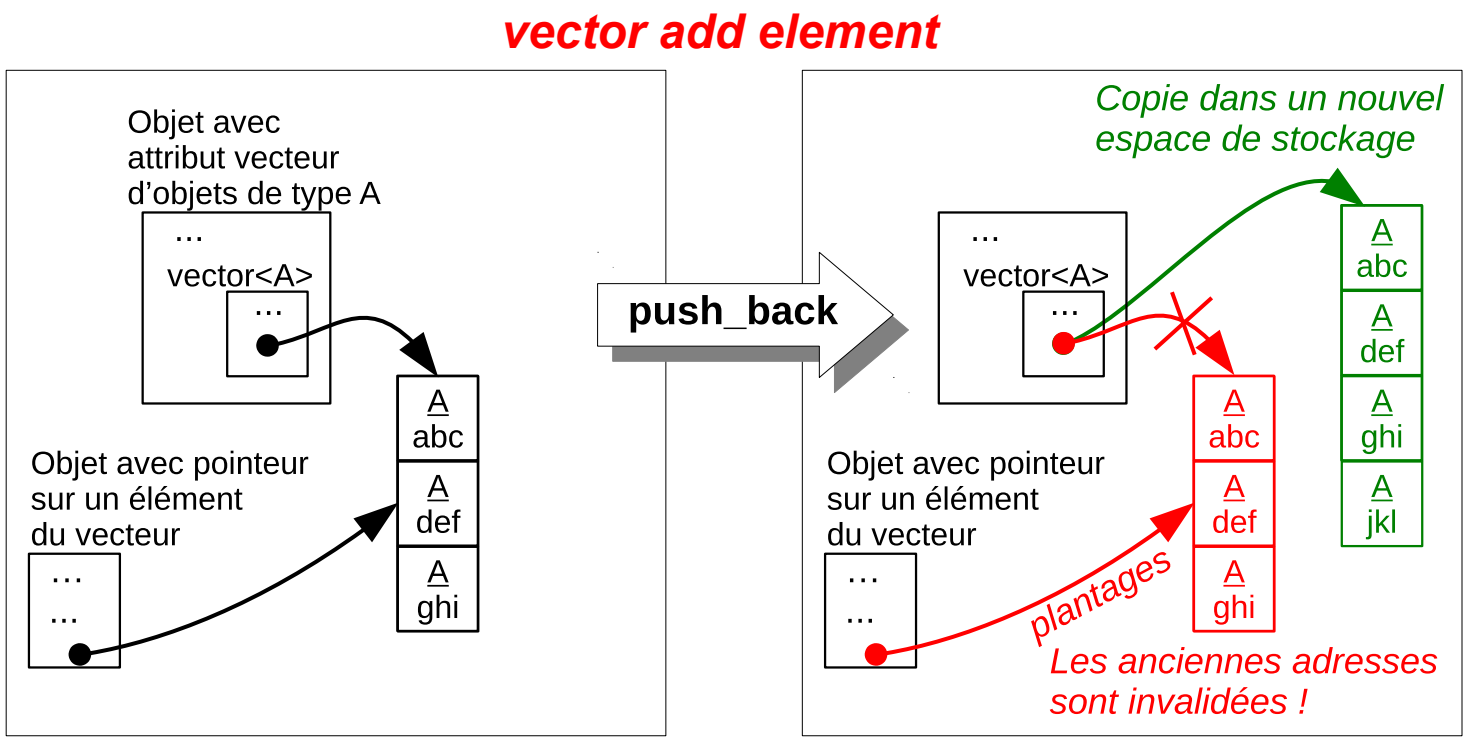
- *Les conteneurs séquentiels contiguës **vector** et **deque** ne stockent **pas** les éléments à des emplacements **stables** :*
  insert et erase peuvent bouger les éléments
- *Les éléments ne doivent pas être pointés !*

vector erase element



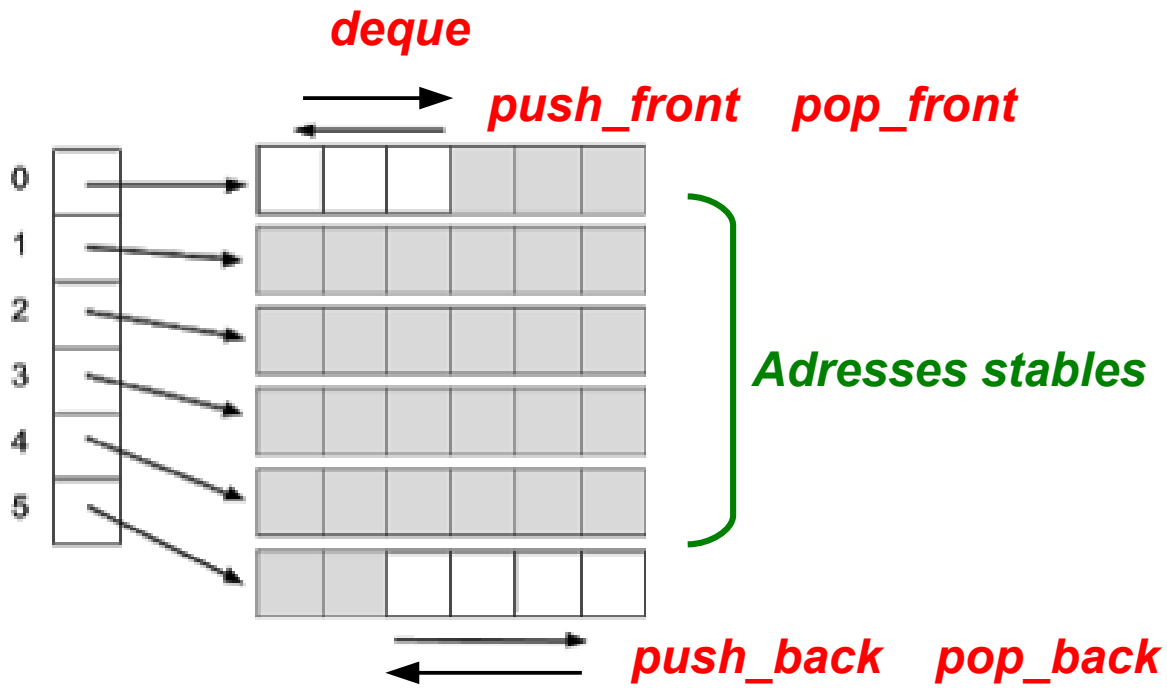
Conteneurs séquentiels

- *Les conteneurs séquentiels contiguës **vector** et **deque** ne stockent **pas** les éléments à des emplacements **stables** :*
  insert et erase peuvent bouger les éléments
- *Les éléments ne doivent pas être pointés !*



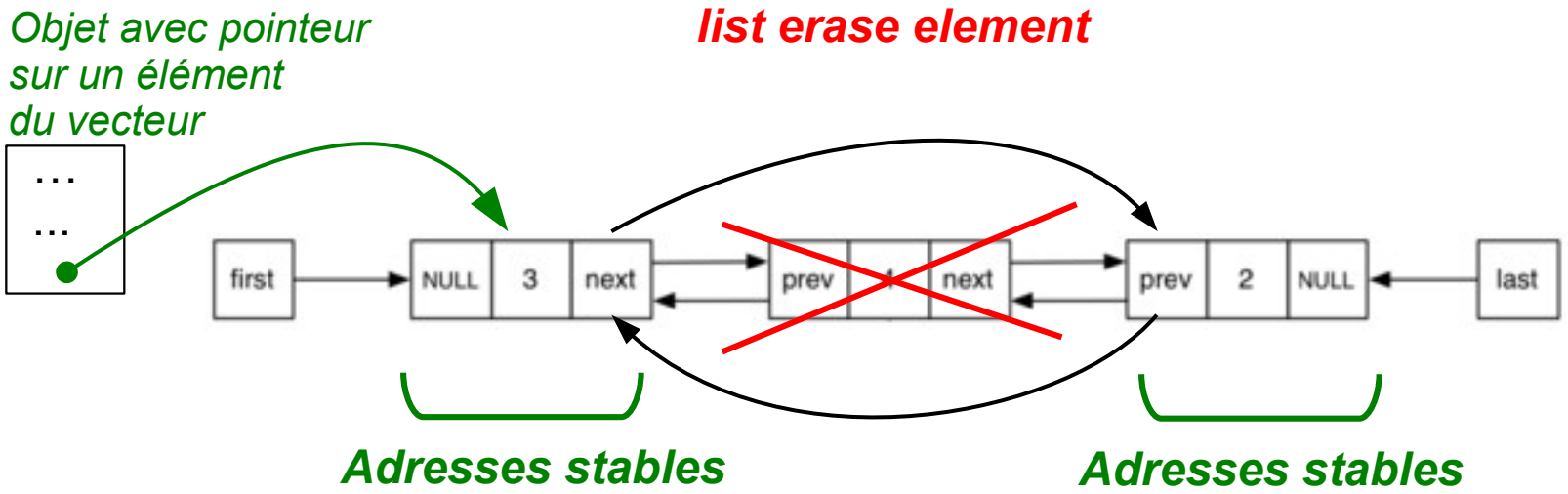
Conteneurs séquentiels

- *Les conteneurs séquentiels contiguës* ***vector*** *et* ***deque*** *ne stockent* ***pas*** *les éléments à des emplacements* ***stables*** *:*
  insert et erase peuvent bouger les éléments
- *Exception : deque avec ajout/retrait tête/queue*



Conteneurs séquentiels

- *Les conteneurs séquentiels non contiguës* ***list*** *et* ***forward_list*** *stockent les éléments à des emplacements* ***stables*** *:*
  insert et erase ne bougent pas les éléments
- *Les éléments peuvent être pointés !*



Conteneurs séquentiels

- *Le hardware (processeur/mémoire) préfère les données contiguës : ça va plus vite*
- ***Si il n'y a pas de contraintes fortes** :*
  - *Pas ou peu d'insertions/délétions*
  - *Et/ou peu d'éléments (N petit)*
  - *Pas de besoin de stabilité des adresses*

  alors le choix « par défaut » est le `vector`
- *Les autres conteneurs séquentiels présentent d'autres profils d'utilisation, `forward_list` stable, léger mais parcours que dans un sens etc...*

Conteneurs séquentiels

- *Si on peut connaître à l'avance le nombre d'éléments qui sera dans un vecteur alors il est préférable de le déclarer à cette taille*

```
/// Réserver 10 cases. Attention (10) pas {10} !
std::vector<int> impairs(10);

/// Remplir directement dans les cases qui existent
for (size_t i=0; i<impairs.size(); i++)
    impairs[i] = 2*i+1;

for (const auto& elem : impairs)
    std::cout << elem << " ";
```

Pré-allouer, préférable si possible

```
1 3 5 7 9 11 13 15 17 19
```

```
/// Vecteur initial vide
std::vector<int> impairs;

/// Remplir avec des push_back
for (int i=0; i<10; i++)
    impairs.push_back(2*i+1);

for (const auto& elem : impairs)
    std::cout << elem << " ";
```

push_back dans vecteur vide
Ça marche et c'est commode
mais ça impose des ré-allocations

```
1 3 5 7 9 11 13 15 17 19
```

Conteneurs séquentiels

- *Si on peut connaître à l'avance le nombre d'éléments qui sera dans un vecteur alors il est préférable de le déclarer à cette taille*

```
/// Réserver 10 cases. Attention (10) pas {10} !
std::vector<int> impairs(10);

/// Remplir directement dans les cases qui existent
for (size_t i=0; i<impairs.size(); i++)
    impairs[i] = 2*i+1;

for (const auto& elem : impairs)
    std::cout << elem << " ";
```

```
1 3 5 7 9 11 13 15 17 19
```

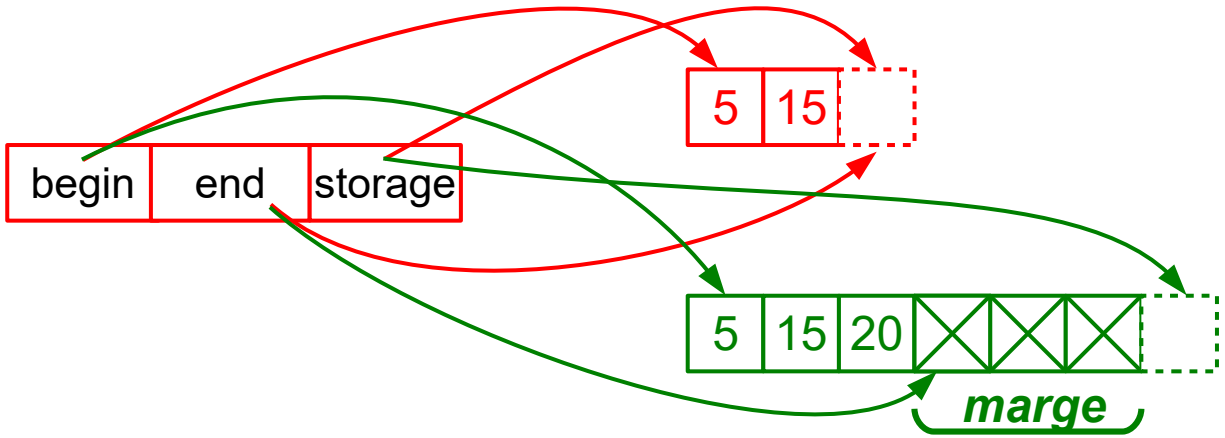
```
/// Vecteur initial vide
std::vector<int> impairs;

/// Remplir avec des push_back
for (int i=0; i<10; i++)
    impairs.push_back(2*i+1);

for (const auto& elem : impairs)
    std::cout << elem << " ";
```

CRASH : Attention à ne pas tout mélanger !

Conteneurs séquentiels

- *Sinon on remplit avec push_back quand on ne peut pas faire autrement*
- *La « complexité en temps amorti » est* ***O(1) par élément ajouté avec push_back***

push_back => ré-allocation



Mais il n'y a pas une ré-allocation à chaque push_back ! L'augmentation de la taille du vecteur se fait de façon exponentielle, par exemple 2, 4, 8 …
Quand on arrive à 1024 élément stockés on a copié 2+4+8...+512=1023 éléments
Donc pour stocker N éléments avec push_back on a fait environ 2N opérations :
2N opérations pour N éléments → proportion de 2N/N = 2 = temps constant = O(1)

COURS 7

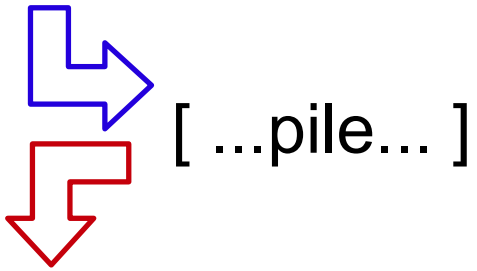
A) Structures de données & STL
B) Itérateurs, parcours, algos
C) Conteneurs séquentiels
D) Piles et files
E) Conteneurs ensemblistes : set
F) Conteneurs associatifs : map
G) Arbre Binaire de Recherche
H) Table de hachage

Piles et files

Piles et files

- *Les **piles** et **files** sont des conteneurs qui imposent des **protocoles stricts et restreints** sur l'ordre d'accès aux éléments stockés*
- *Pas d'itérateurs : pas de parcours ! Récupérer les données → les consommer*

Pile = **`std::stack`**
LIFO Last In First Out

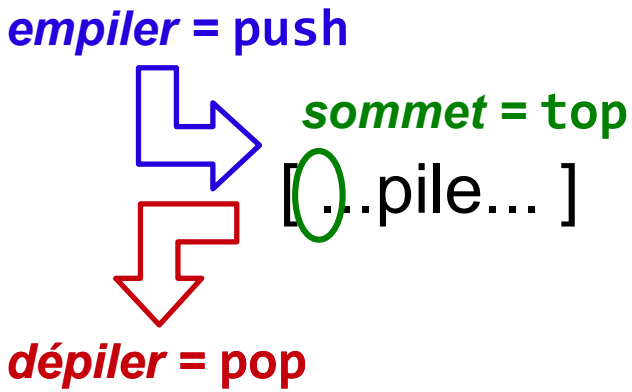


File = **`std::queue`**
FIFO First In First Out

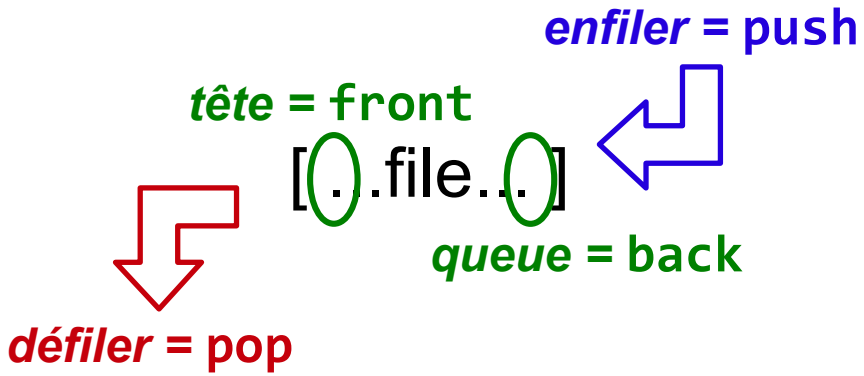
Piles et files

- *Les **piles** et **files** sont des conteneurs qui imposent des **protocoles stricts et restreints** sur l'ordre d'accès aux éléments stockés*
- *Pas d'itérateurs : pas de parcours !*
  Récupérer les données → les consommer: **pop** *!*

Pile = **`std::stack`**
LIFO Last In First Out



File = **`std::queue`**
FIFO First In First Out



Piles et files

Source : cppreference

```
#include <stack>
#include <iostream>

int main()
{
    std::stack<int>   s;

    s.push( 2 );
    s.push( 6 );
    s.push( 51 );

    std::cout << s.size() << " elements on stack\n";
    std::cout << "Top element: "
              << s.top()          // Leaves element on stack
              << "\n";
    std::cout << s.size() << " elements on stack\n";
    s.pop();
    std::cout << s.size() << " elements on stack\n";
    std::cout << "Top element: " << s.top() << "\n";

    return 0;
}
```

pile

```
3 elements on stack
Top element: 51
3 elements on stack
2 elements on stack
Top element: 6
```

Piles et files

- *Les **piles** et **files** sont des conteneurs qui sont à la base de nombreux algorithmes permettant de traiter les structures de données non séquentielles, en particulier les **arbres** et les **graphes** : second semestre =)*

Pile = **`std::stack`**
LIFO Last In First Out

Parcours en profondeur d'abord

Gestion de tâches, sous-tâches...
Exemple : appels de sous-programmes

Revenir en arrière - *backtracking* - pour explorer des alternatives
Exemple : Undo/Redo

File = **`std::queue`**
FIFO First In First Out

Parcours en largeur d'abord

Situations de communication entre processus producteur et consommateur asynchrones
Exemple : gestionnaire d'événement

Accès à des ressources partagées
Exemple : file d'impression

COURS 7

A) Structures de données & STL
B) Itérateurs, parcours, algos
C) Conteneurs séquentiels
D) Piles et files
E) Conteneurs ensemblistes : set
F) Conteneurs associatifs : map
G) Arbre Binaire de Recherche
H) Table de hachage

Conteneurs ensemblistes : set

Conteneurs ensemblistes : set

- *Les conteneurs « ensemblistes »*
  ***set** et **unordered_set** sont des conteneurs utilisés quand on veut souvent savoir si une **valeur** appartient à un **ensemble** de valeurs.*
- *Ils vont ajouter/enlever/(re)trouver des valeurs **efficacement***
- *Il vont pouvoir servir de dictionnaire...*
- *Méthode* `find` *pour* `set` *:* ***O(log N)***
- *Méthode* `find` *pour* `unordered_set` *:* ***O(1)***
- *Fonction* `find` *pour* `vector` *& séquentiels :* ***O(N)***
  L'algo pour trouver est de tester 1 par 1 !

Conteneurs ensemblistes : set

- **`set`** *utilise un arbre binaire de recherche*
- **`unordered_set`** *utilise une table de hachage*
- ➔ *Voir chapitres G et H*
- *Exemple d'utilisation en situation réelle : dans la classe Svgfile utilisée en TP on veut éviter d'ouvrir plusieurs fois le même fichier, un attribut « static » (variable globale de classe nous y reviendrons) contiendra l'**ensemble** des noms de fichiers en cours d'utilisation...*

```
// Pour éviter les ouverture multiples
static std::set<std::string> s_openfiles;
```

ligne 44 svgfile.h

Conteneurs ensemblistes : set

- *Exemple montrant l'utilisation de* **`find`**
- *Si la valeur demandée existe dans le conteneur* **`find`** *retourne l'itérateur sur cet élément sinon il retourne l'itérateur de fin*

Source : cppreference

```
Found 2
```

```cpp
#include <iostream>
#include <set>

int main()
{
    std::set<int> example = {1, 2, 3, 4};

    auto search = example.find(2);
    if (search != example.end()) {
        std::cout << "Found " << (*search) << '\n';
    } else {
        std::cout << "Not found\n";
    }
}
```

Conteneurs ensemblistes : set

- Autre exemple montrant les méthodes insert (ajout à l'ensemble) et erase (enlever)

```
void check(const std::set<int>& ensemble, int val)
{
    auto trouve = ensemble.find(val);
    if ( trouve!=ensemble.end() )
        std::cout << val << " est dans l'ensemble" << std::endl;
    else
        std::cout << val << " n'est pas dans l'ensemble" << std::endl;
}

int main()
{

    std::set<int> nbr;

    nbr.insert(5);
    check(nbr, 3);
    nbr.insert(3);
    check(nbr, 3);
    nbr.erase(3);
    check(nbr, 3);
```

```
3 n'est pas dans l'ensemble
3 est dans l'ensemble
3 n'est pas dans l'ensemble
```

Conteneurs ensemblistes : set

- **`set`** *marche avec tous les types comparables avec operator< ou foncteur less → chapitre G*
- **`unordered_set`** *marche avec tous les types hachables (avec foncteur hash) → chapitre H*
- *Types basics int, double... pointeurs... string sont utilisables par défaut*
- *Pour avoir* `std::set<Ecureuil>` *il faudra dire au système comment les comparer (idem que pour les algos de tri)*
- *Pour avoir* `std::unordered_set<Elephant>` *il faudra dire au système comment hacher un éléphant !*

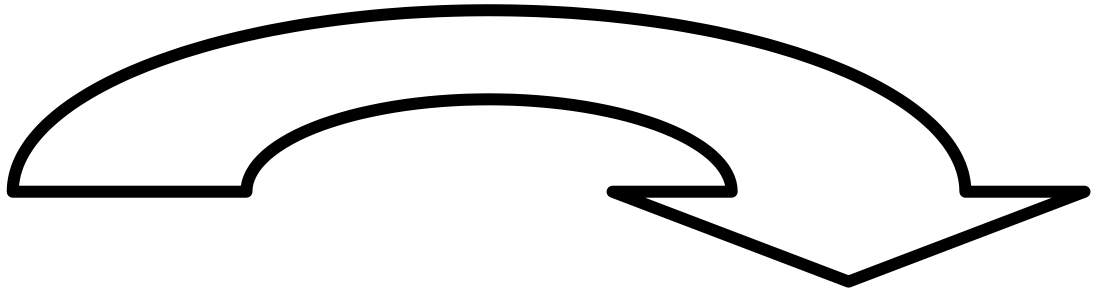
Conteneurs ensemblistes : set

- **`set`** et **`unordered_set`** *garantissent l'unicité des valeurs qui leur sont ajoutées : on peut ajouter (`insert`) plusieurs fois la même valeur sans créer de doublon !*
- *Utiliser les versions* **`multiset`** *et* **`unordered_multiset`** *si vous voulez des doublons !*
- **`set`** est **naturellement toujours trié** : son parcours donnera les élément dans l'ordre croissant (selon la méthode de comparaison...) Le tri se fait directement à chaque insertion (insertion à la bonne place)
- **`unordered_set`** n'est ni trié ni triable !

COURS 7

A) Structures de données & STL
B) Itérateurs, parcours, algos
C) Conteneurs séquentiels
D) Piles et files
E) Conteneurs ensemblistes : set
F) Conteneurs associatifs : map
G) Arbre Binaire de Recherche
H) Table de hachage

Conteneurs associatifs : map



Conteneurs associatifs : map

- **`map`** *et* **`unordered_map`** *sont des conteneurs qui contiennent des paires* ***clé-valeur (key-value)***
- *Ils se déclarent avec 2 paramètres de type :* **`std::map<K, V>`**
  - *K est le type clé*
    même contraintes que set & unordered_set
  - *V est le type valeur*
    besoin constructeur par défaut pour accès [key]
- *Un conteneur associatif permet de retrouver* ***très efficacement*** *la valeur associée à une clé*

  O(log N) *pour map,* ***O(1)*** *pour unordered_map*

Conteneurs associatifs : map

- *Exemple avec une map qui associe un entier(valeur) à une chaîne(clé) :* **`std::map<std::string, int>`**
- *Les paires sont rangées dans une struct générique* **`std::pair`** *avec attribut* **`first`** *(clé) et* **`second`** *(valeur)*

```cpp
/// Déclaration avec initialisation
std::map<std::string, int> asso
{
    {"Zürich",   143},
    {"Basel",    114},
    {"Lausanne", 116},
    {"Bern",      24}
};

/// Parcours par itérateur
for(auto it=asso.cbegin(); it!=asso.cend(); ++it)
    std::cout << it->first << " -> " << it->second << std::endl;

/// Parcours par "range-based for loop" (même résultat)
for(const auto& elem : asso)
    std::cout << elem.first << " -> " << elem.second << std::endl;
```

```
Basel -> 114
Bern -> 24
Lausanne -> 116
Zürich -> 143
```

Conteneurs associatifs : map

- *On peut appeler explicitement la méthode* **`insert`** *ou utiliser la notation accès direct par* **`[clé]`**
- *Noter que le conteneur map fait une insertion triée dans l'ordre des clés (et non dans l'ordre des insert)*

```
/// Déclaration avec remplissage ultérieur
std::map<std::string, int> asso;

/// Remplissage : noter la syntaxe make_pair
asso.insert(std::make_pair("Zürich", 143));

/// Remplissage : forme courte
asso.insert({"Basel", 114});
// Attention ceci ne passe pas
// asso.insert("Basel", 114);

/// Remplissage : forme directe
asso["Lausanne"] = 116;
asso["Bern"] = 24;

/// Parcours par "range-based for loop"
for(const auto& elem : asso)
    std::cout << elem.first << " -> " << elem.second << std::endl;
```

```
Basel -> 114
Bern -> 24
Lausanne -> 116
Zürich -> 143
```

Conteneurs associatifs : map

- *Attention avec la notation accès direct par* **`[clé]`**
  le seul fait de parler d'une clé créer une entrée !
- *Ça ne plante pas, mais c'est rarement bon signe...*

```cpp
/// Accès direct par clé en lecture et écriture
std::cout << asso["Bern"] << std::endl;
asso["Bern"] = 25;
std::cout << asso["Bern"] << std::endl;
++asso["Bern"];
std::cout << asso["Bern"] << std::endl;


/// Si la clé n'existe pas on a une valeur par défaut
/// Surprise un accès en lecture a ajouté un élément !
std::cout << asso["Mexico"] << std::endl;

for(const auto& elem : asso)
    std::cout << elem.first << " -> " << elem.second << std::endl;
```

```
24
25
26
0
Basel -> 114
Bern -> 26
Lausanne -> 116
Mexico -> 0
Zürich -> 143
```

Conteneurs associatifs : map

- *La fête est gâchée ! L'accès direct par* **`[clé]`**
  est une belle syntaxe mais il faut gérer les aléas...
- *Donc en général on doit se rappeler qu'une*
  clé peut ne pas exister (et qu'il vaut mieux le savoir)

```
/// l'accès avec [clé] est sympa mais dangereux ! Alternative...
auto trouve = asso.find("Tokyo");
if ( trouve!=asso.end() )
    std::cout << trouve->first << " -> " << trouve->second << std::endl;
else
    std::cout << "Tokyo pas trouvé" << std::endl;

trouve = asso.find("Lausanne");
if ( trouve!=asso.end() )
    std::cout << trouve->first << " -> " << trouve->second << std::endl;
else
    std::cout << "Lausanne pas trouvé" << std::endl;
```

```
Tokyo pas trouvé
Lausanne -> 116
```

Conteneurs associatifs : map

- *Enlever des éléments ne pose pas de problème...*
  Vérifier de ne pas utiliser un itérateur qui vaut end( ) !
- *L'objet valeur associé est détruit. Si la valeur est un pointeur c'est le pointeur qui est détruit, pas le pointé.*

```
/// Effacer par itérateur
trouve = asso.find("Bern");
if ( trouve!=asso.end() )
    asso.erase(trouve);

/// Effacer directement
asso.erase("Mexico");

/// Effacer directement
/// une clé inexistante = aucune conséquence !
asso.erase("Sidney");

/// Après les effacements
for(const auto& elem : asso)
    std::cout << elem.first << " -> " << elem.second << std::endl;
```

```
Basel -> 114
Lausanne -> 116
Zürich -> 143
```

Conteneurs associatifs : map

- **`map`** *et* **`unordered_map`** *s'utilisent pratiquement de la même façon (même interface de base)*
- **`unordered_map`** *ne donne aucune garantie d'ordre dans la séquence de parcours : ce n'est ni l'ordre d'ajout des éléments ni un ordre naturel*
- **`unordered_map`** *peut être plus rapide pour des grosses maps mais ça dépend de nombreux facteurs (voir implémentations chapitres G et H) Il faut faire des tests pour vraiment savoir*

O(log N) de **`map`** *donne des* ***bonnes performances***

$\log_2 10^3 \approx 10$ $\log_2 10^6 \approx 20$ $\log_2 10^9 \approx 30$

Conteneurs associatifs : map

- *Du moment qu'il est comparable (operator<) ou hachable n'importe quel type clé est possible*
- **`std::map<Ecureuil, Elephant>`**
  À chaque écureuil on associe un éléphant (indiquer au système operator< entre écureuils)
- **`std::unordered_map<Elephant, Ecureuil>`**
  À chaque éléphant on associe un écureuil (indiquer au système le hachage d'éléphant)
- *Mais le plus souvent on utilise un type clé de type-valeur (au sens value-type, voir cours 5)*

```
std::map<Date, std::string> saints;
```

Conteneurs associatifs : map

- *Autre exemple (atypique)*

```cpp
std::map<double, std::string> quoi;

quoi[3.1413] = "entre 3.141 et 3.142 ?";
quoi[3.14] = "Moins que Pi";
quoi[3.1411] = "Combien de choses ";
quoi[3.15] = "Plus que Pi";
quoi[3.141] = "Moins que Pi";
quoi[3.1412] = "peut-on ranger ";
quoi[3.142] = "Plus que Pi";

/// Parcours par "range-based for loop"
for(const auto& elem : quoi)
    std::cout << elem.first << " -> " << elem.second << std::endl;
```

```
3.14 -> Moins que Pi
3.141 -> Moins que Pi
3.1411 -> Combien de choses
3.1412 -> peut-on ranger
3.1413 -> entre 3.141 et 3.142 ?
3.142 -> Plus que Pi
3.15 -> Plus que Pi
```

Conteneurs associatifs : map

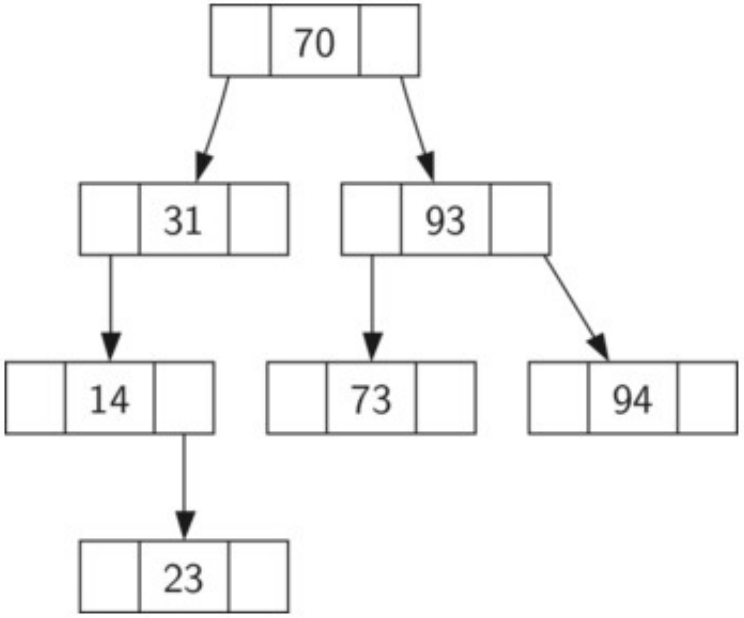
- *Les conteneurs associatifs trouvent un nombre considérable d'application en programmation*
- *Parmi les exemples d'usage : la sérialisation*
- *Sérialiser = transformer des données RAM en séquence d'octets pour sauvegarde fichier ou stockage base de données ou transfert réseau...*
- *Le problème : les objets des types-entités se désignent réciproquement par pointeurs, et les pointeurs ne se rechargent pas !*
- *Une solution possible: associer un int à chacun !* **`std::map<Elephant*, int>`** *et sauver ces indices réciproques*

COURS 7

A) Structures de données & STL
B) Itérateurs, parcours, algos
C) Conteneurs séquentiels
D) Piles et files
E) Conteneurs ensemblistes : set
F) Conteneurs associatifs : map
G) Arbre Binaire de Recherche
H) Table de hachage

Arbre Binaire de Recherche

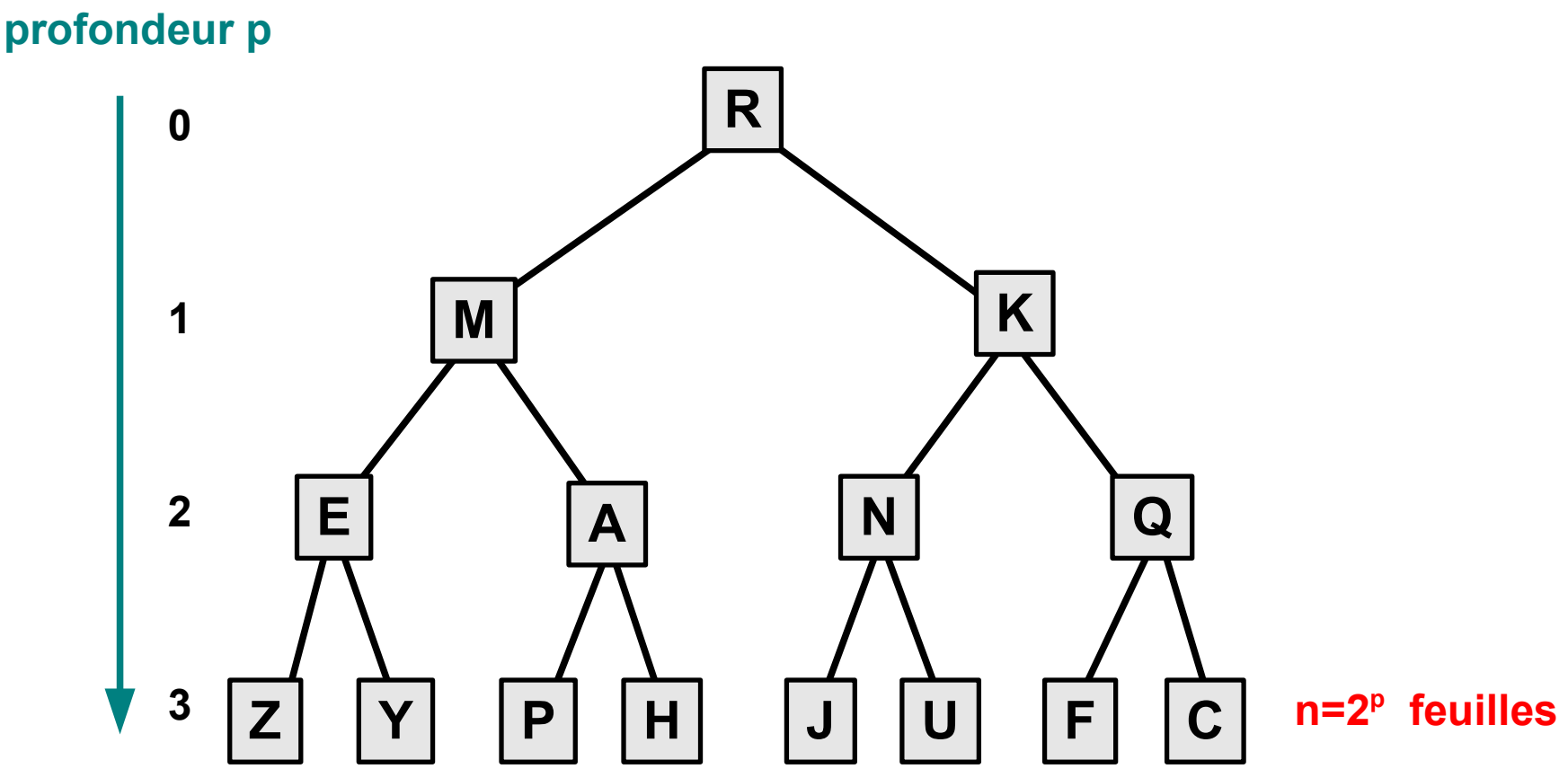
Arbre Binaire de Recherche



- `set` *et* `map` *utilisent une structure de donnée* ***arbre binaire de recherche***
- ***set y stocke juste une clé***
- ***map y stocke une paire clé-valeur***

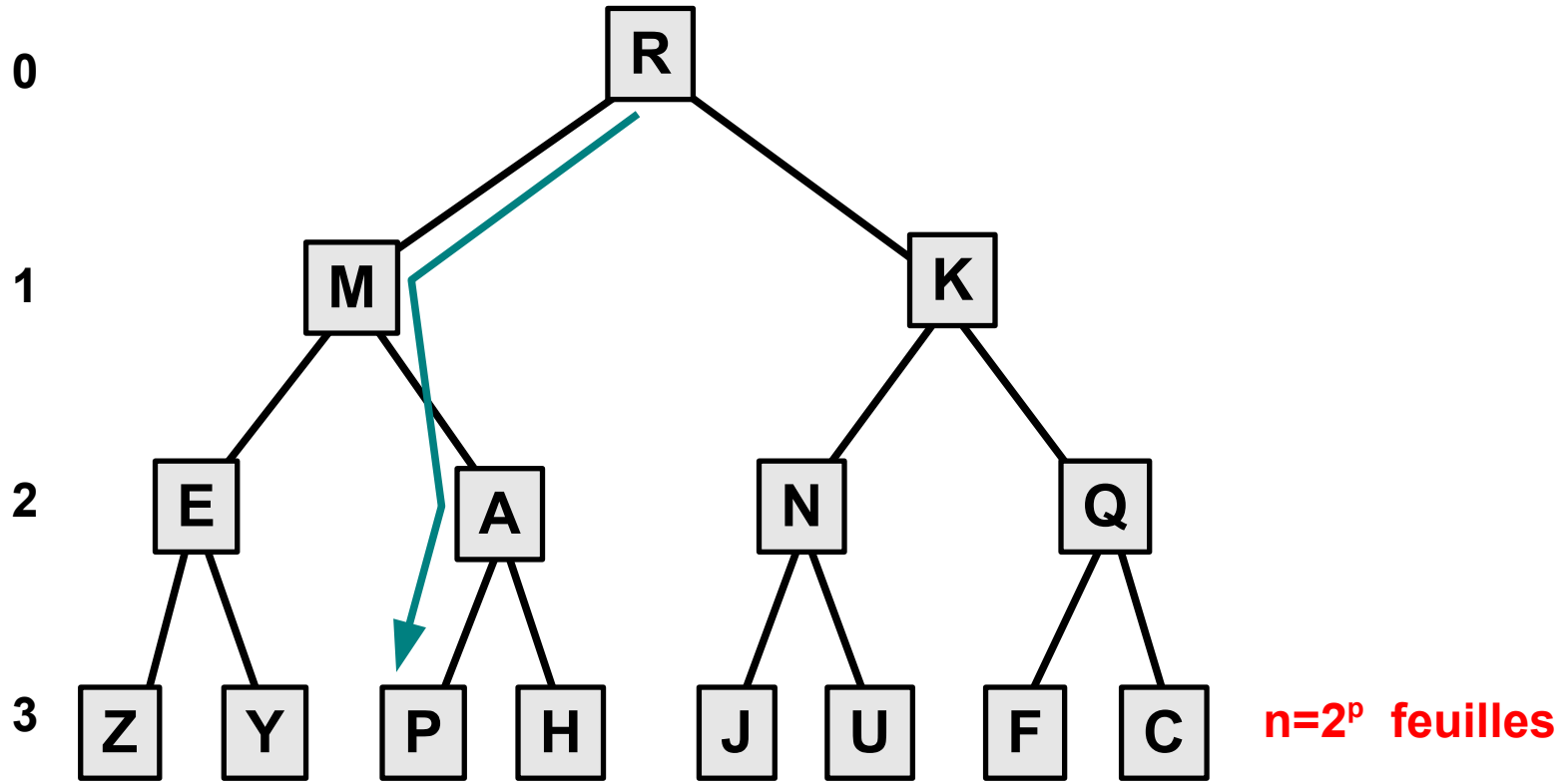
Arbre Binaire de Recherche

- **Caractéristiques des arbres binaires équilibrés : on peut atteindre une feuille parmi $n=2^p$ en p étapes**



Arbre Binaire de Recherche

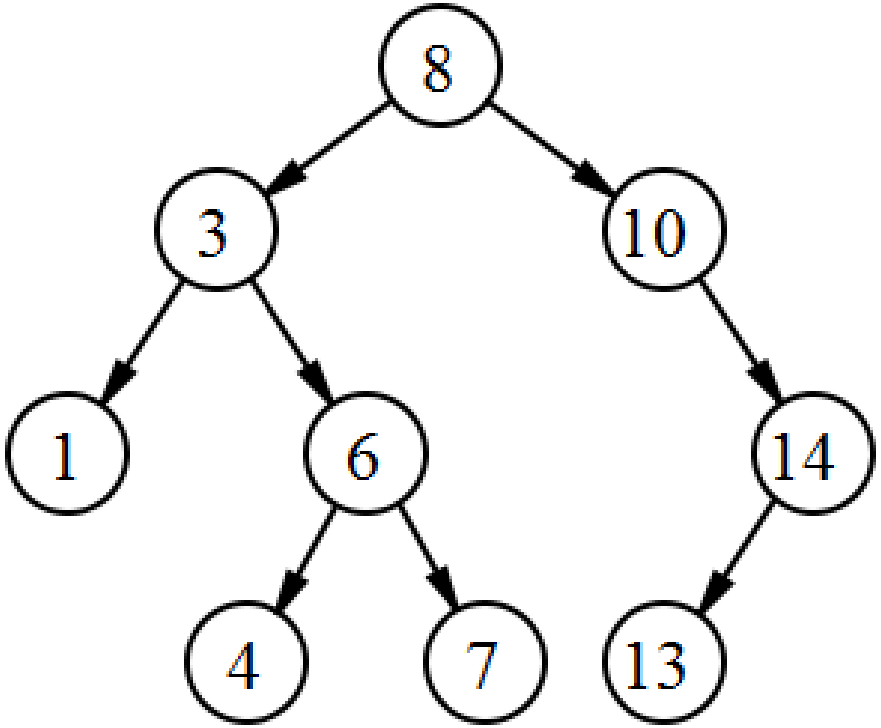
- **Caractéristiques des arbres binaires équilibrés : on peut atteindre une feuille parmi $n=2^p$ en p étapes**
- **Un** accès à un élément **parmi n** est efficace **$p=\log_2 n$**



Arbre Binaire de Recherche

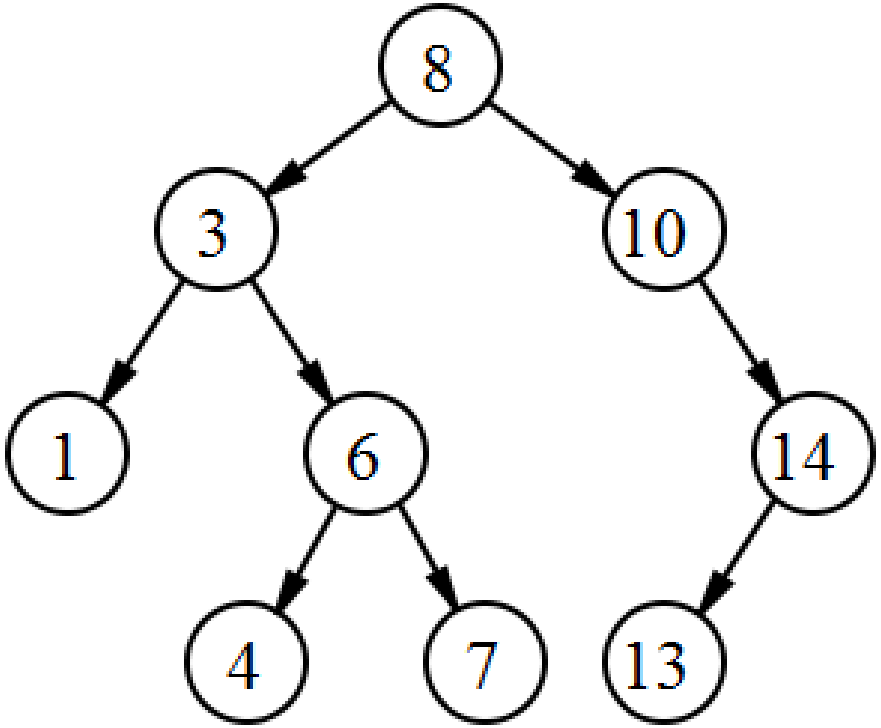
- Arbre Binaire **de Recherche** : insertion triée efficace

  chaque nœud porte une valeur unique : clé du tri...



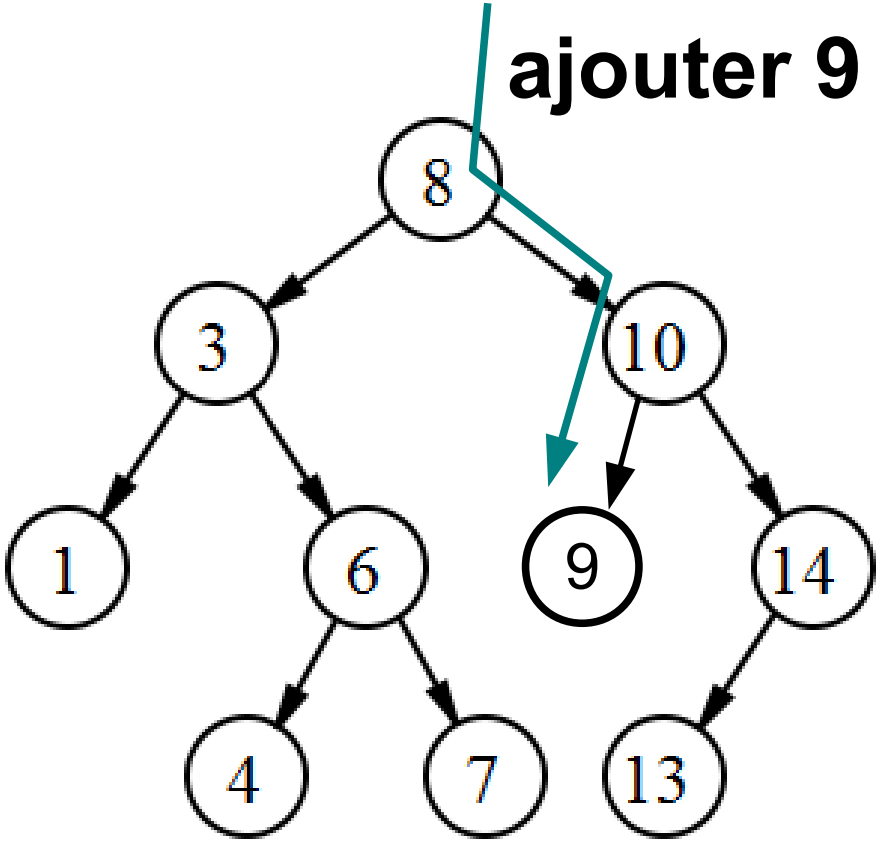
Arbre Binaire de Recherche

- Arbre Binaire **de Recherche** : insertion triée efficace
- **Propriété à vérifier** : pour tout nœud
  max(sous arbre gauche) < valeur du nœud
  min(sous arbre droit) > valeur du nœud



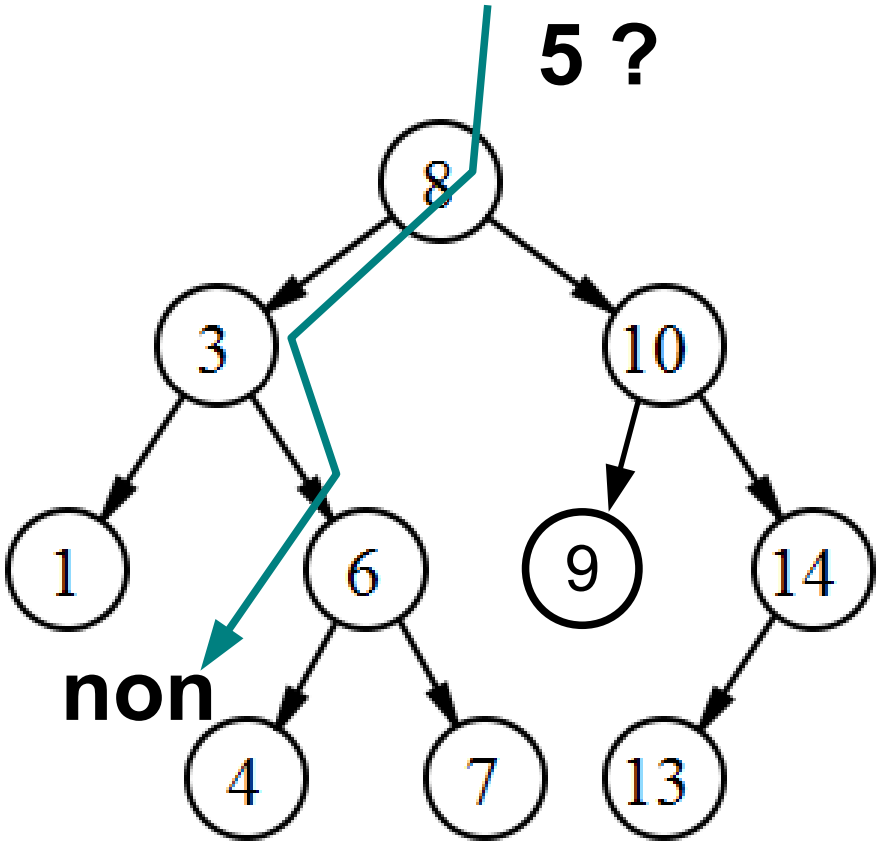
Arbre Binaire de Recherche

- Pour **insérer** : parcours en $\log_2 n \rightarrow$ efficace
- On aiguille à chaque niveau à gauche ou à droite selon la valeur rencontrée



Arbre Binaire de Recherche

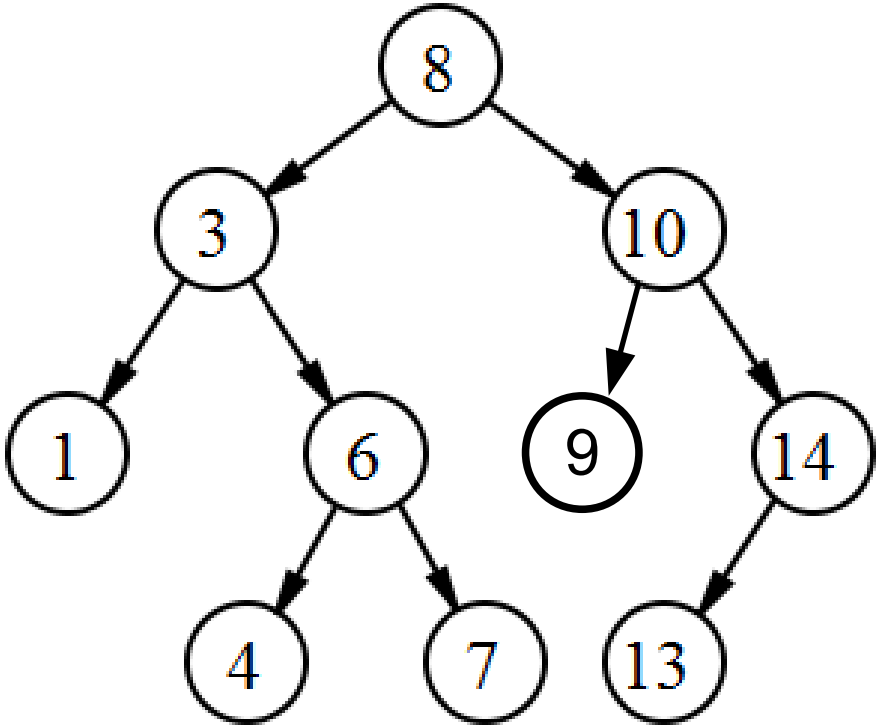
- Pour **rechercher :** parcours en $\log_2 n \rightarrow$ efficace
- On aiguille à chaque niveau à gauche ou à droite selon la valeur rencontrée



Arbre Binaire de Recherche

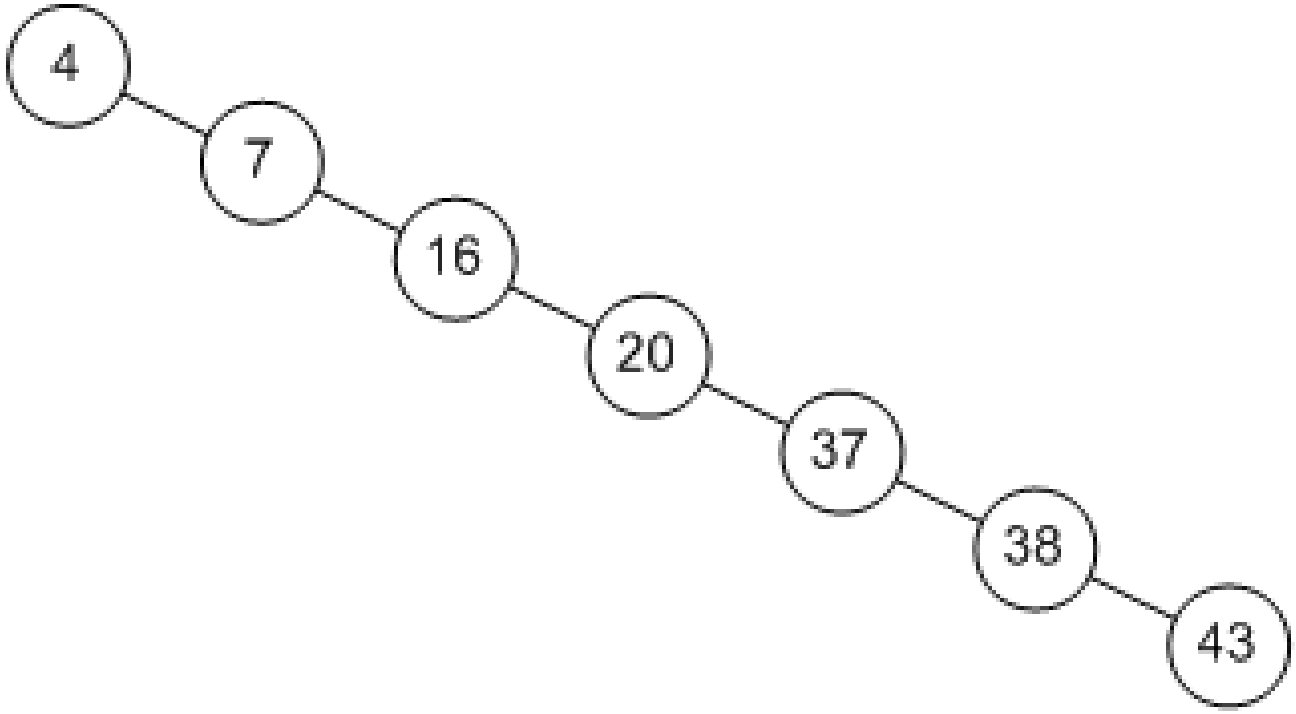
- **On obtient tous les nœuds dans l'ordre trié en utilisant un parcours en profondeur infixé**

1 3 4 6 7 8 9 10 13 14



Arbre Binaire de Recherche

- **L'insertion n'est efficace que si l'arbre binaire de recherche est équilibré**



- Un mécanisme de ré-équilibrage efficace doit être utilisé au fur et à mesure des insertions …

Voir introduction de Wikipedia

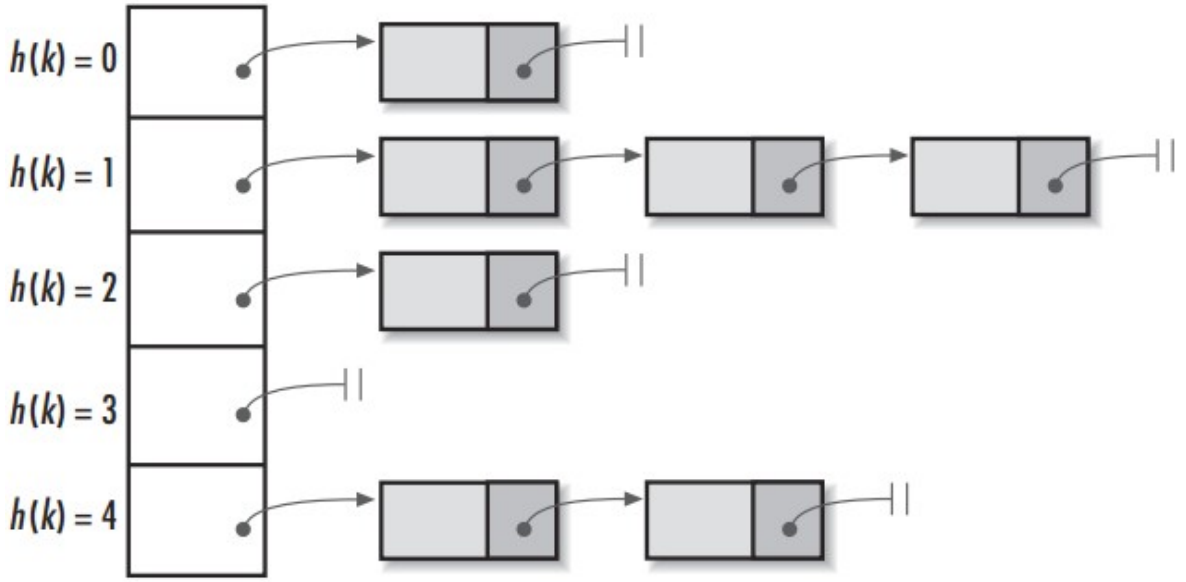
COURS 7

A) Structures de données & STL
B) Itérateurs, parcours, algos
C) Conteneurs séquentiels
D) Piles et files
E) Conteneurs ensemblistes : set
F) Conteneurs associatifs : map
G) Arbre Binaire de Recherche
H) Table de hachage

Table de hachage

Table de hachage

Voir introduction de Wikipedia



- **`unordered_set`** *et* **`unordered_map`** *utilisent une structure de donnée* ***table de hachage***
- **`unordered_set`** ***y stocke juste une clé***
- **`unordered_map`** ***y stocke une paire clé-valeur***