

# 16. TDA Ensemble





Un ensemble stocke ses données *sans répétition*.

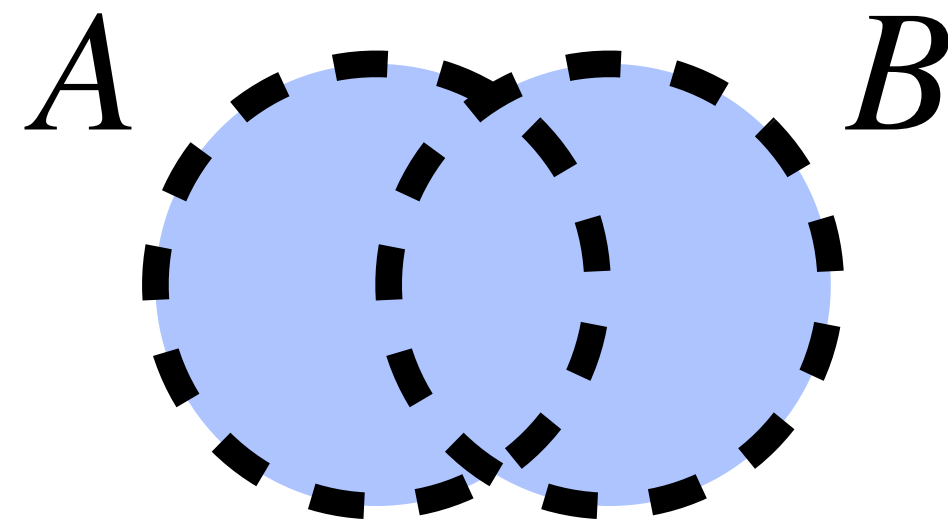
## Opérations

- **Insertion** d'un élément
- **Recherche** de la présence d'un élément
- **Suppression** d'un élément
- **Parcours, itération, ...**

Un ensemble est dit **ordonné** si on peut le parcourir dans l'ordre.  
(*Par exemple s'il est mis en oeuvre avec un ABR*)

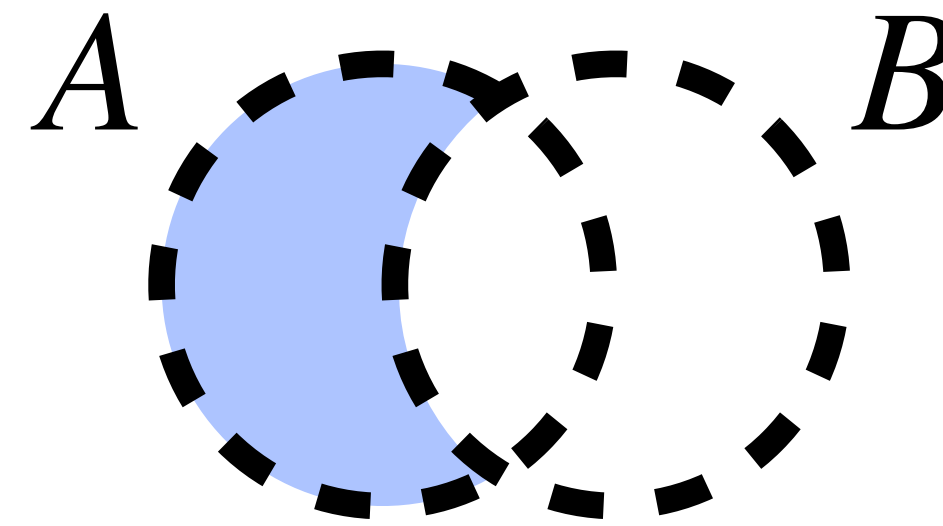


## Union



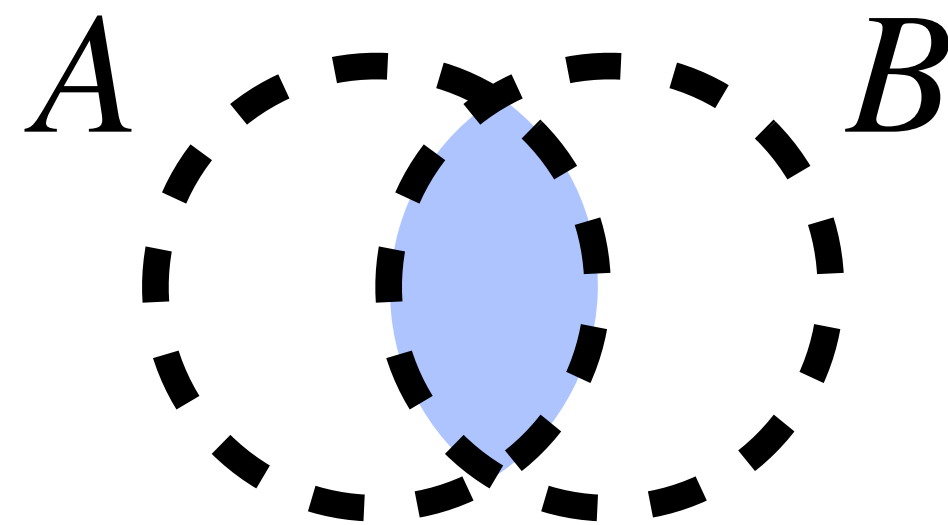
$$A \cup B = \{x : x \in A \vee x \in B\}$$

## Différence



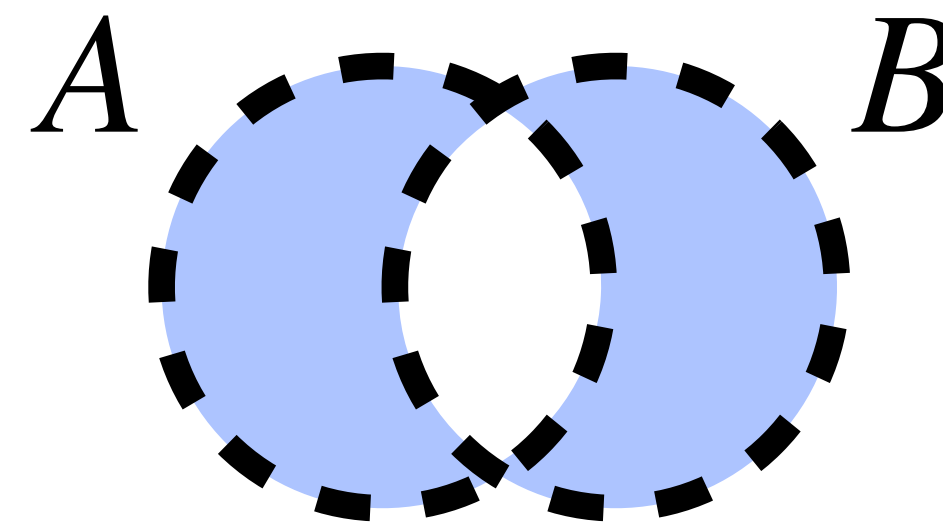
$$A \setminus B = \{x : x \in A \wedge x \notin B\}$$

## Intersection

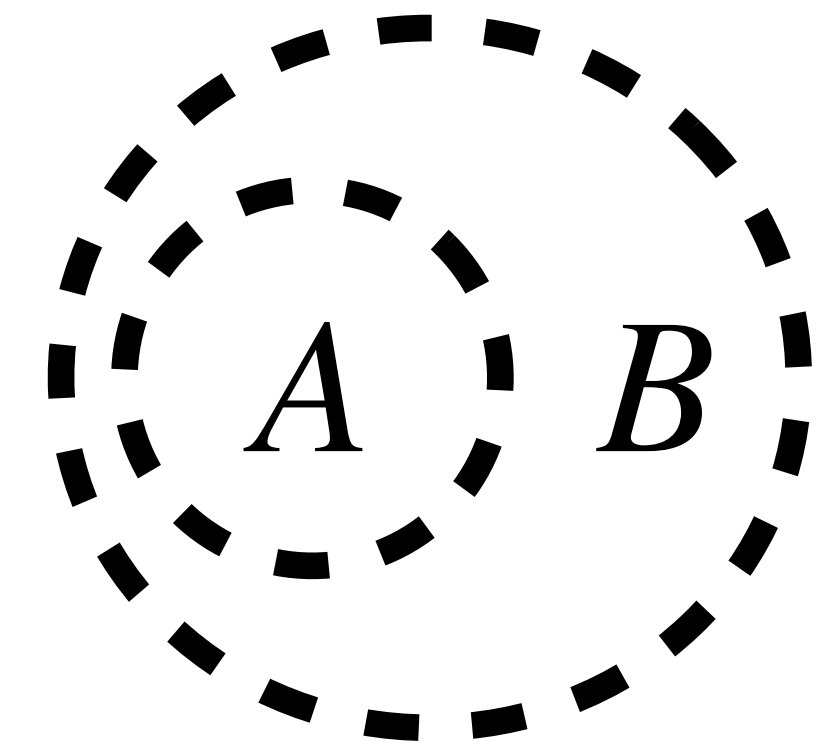


$$A \cap B = \{x : x \in A \wedge x \in B\}$$

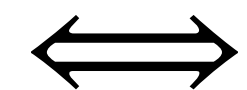
## Différence symétrique



$$A \Delta B = (A \cup B) \setminus (A \cap B)$$



$$A \subseteq B$$

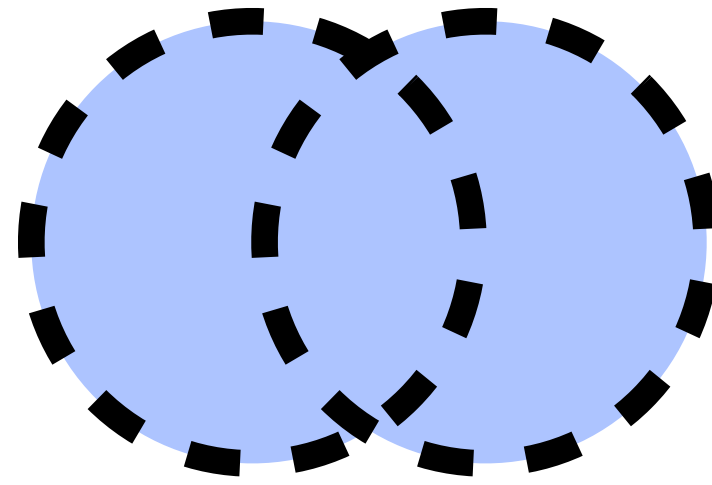


$$\forall x \in A, x \in B$$



# Union

$$A \cup B$$



```
fonction union (A, B)
  U ← copier(A)
  b ← début(B)
  tant que b != fin(B),
    insérer(U, b.clé)
    b ← suivant(b)
  retourner U
```

## Algorithme naïf

- Copier un ensemble dans un ensemble de sortie
- Y insérer les éléments de l'autre ensemble

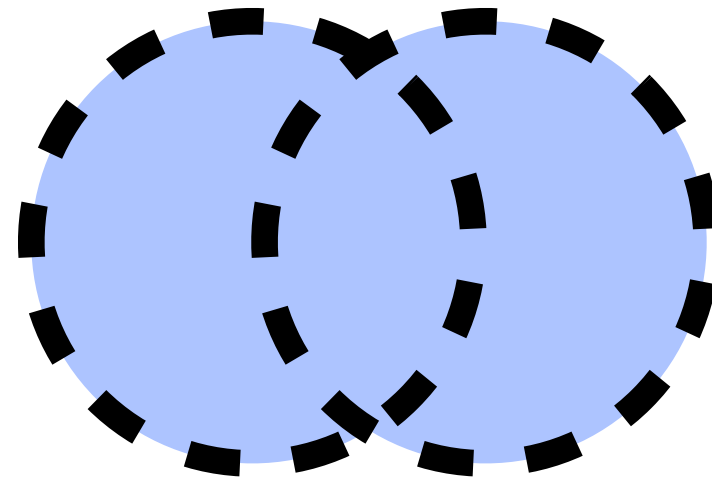
## Complexité

$O(n + m \cdot \log(n + m))$  si les ensembles ont  $n$  et  $m$  éléments.



# Union

$$A \cup B$$

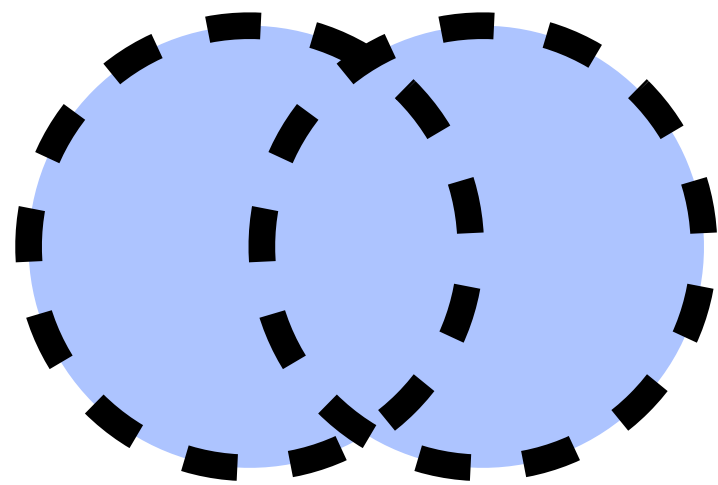


## Algorithme optimisé

**Idée** : similaire à la fusion du tri fusion :

- $O(n + m)$  [
- Itérer sur les deux ensembles dans l'ordre croissant,
  - en extrayant à chaque fois le plus petit des deux minimums,
  - et l'insérant dans l'ensemble de sortie.
- $O(n + m)$  [
- Le résultat est un ABR dégénéré qu'il suffit d'arboriser.

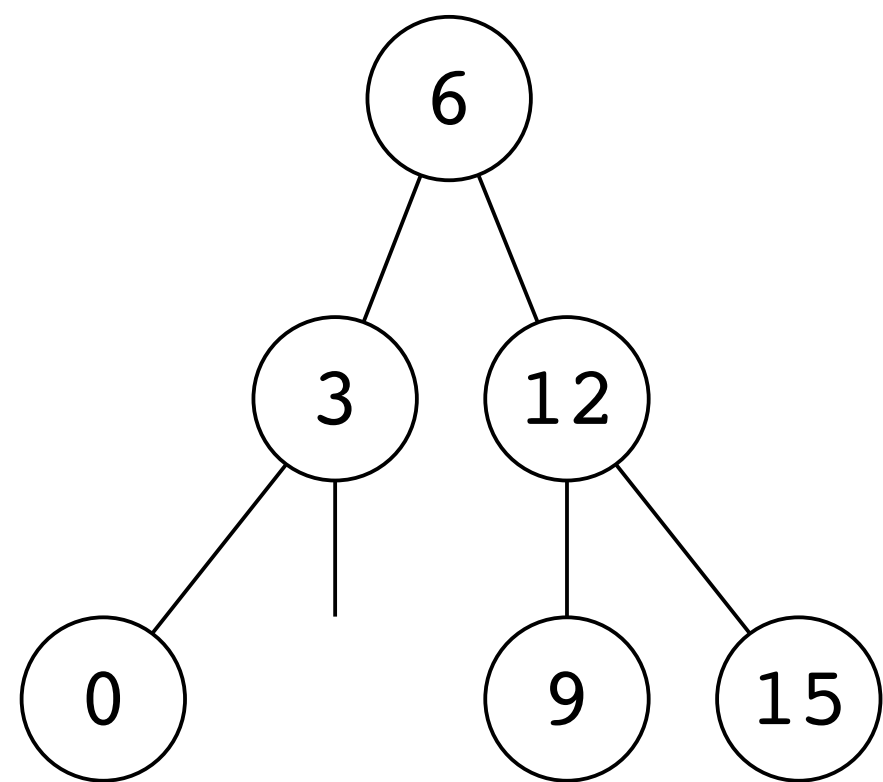
# Union

 $A \cup B$ 

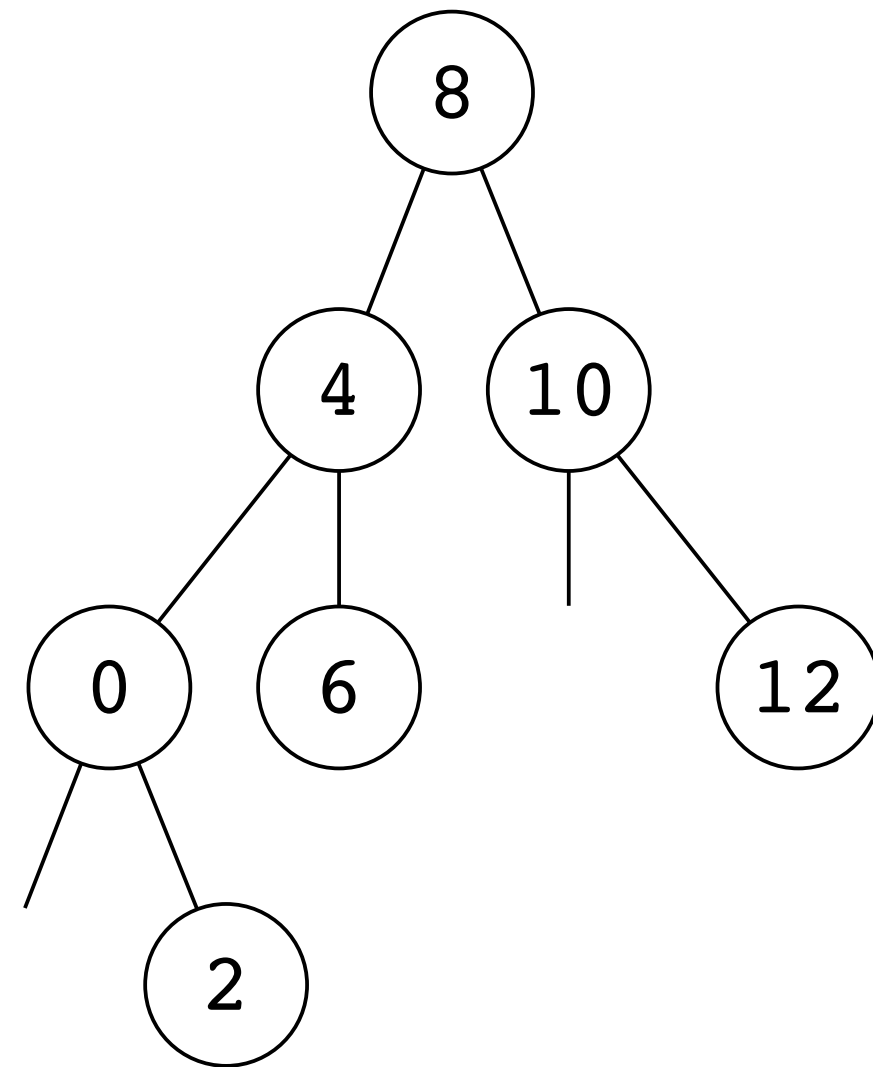
```
fonction union (A, B)
    U ← ∅
    a ← début(A), b ← début(B)
    tant que a != fin(A) et b != fin(B),
        si a.clé < b.clé,
            k ← a.clé, a ← suivant(a)
        sinon, si a.clé > b.clé,
            k ← b.clé, b ← suivant(b)
        sinon,
            k ← a.clé, a ← suivant(a), b ← suivant(b)
        n ← nouveau noeud de clé k
        n.gauche ← U, U ← n
    tant que a != fin(A),
        n ← nouveau noeud de clé a.clé
        n.gauche ← U, U ← n, a ← suivant(a)
    tant que b != fin(B),
        n ← nouveau noeud de clé b.clé
        n.gauche ← U, U ← n, b ← suivant(b)
    retourner U
```



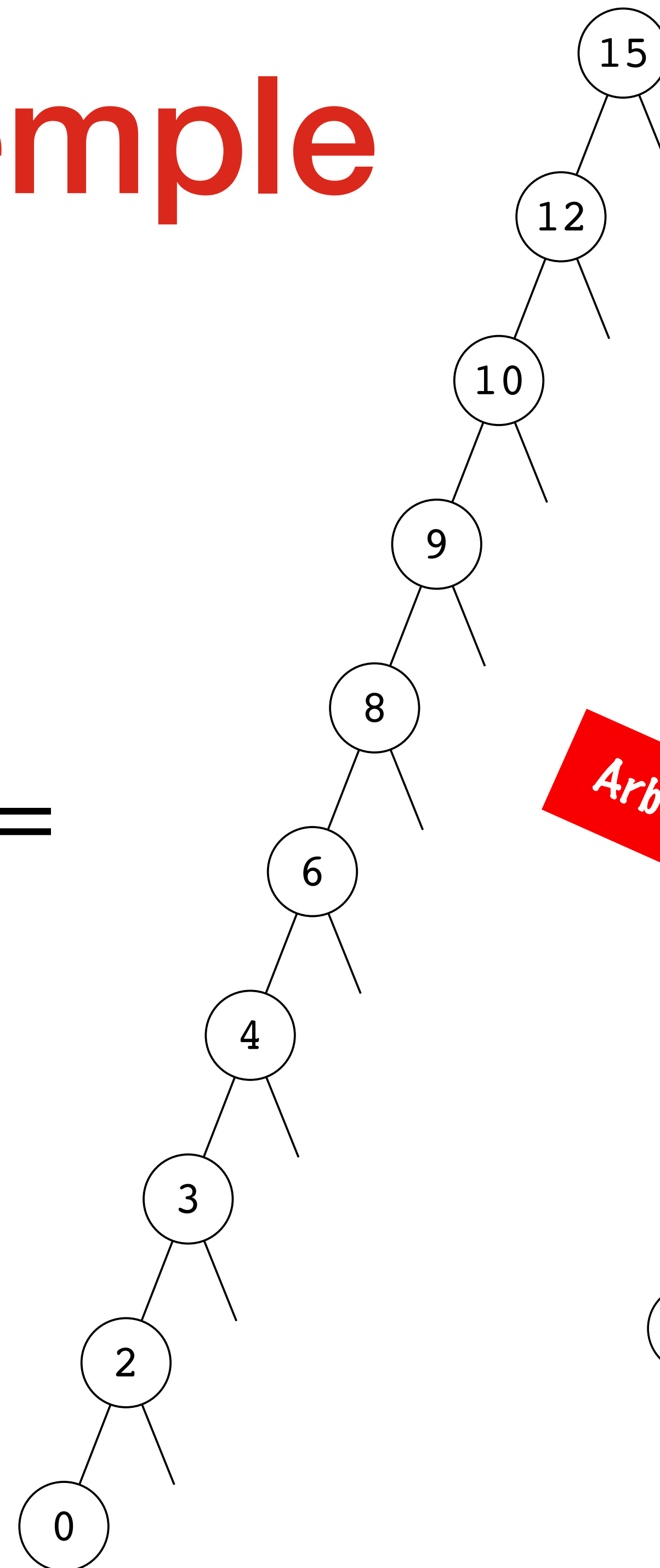
## Exemple



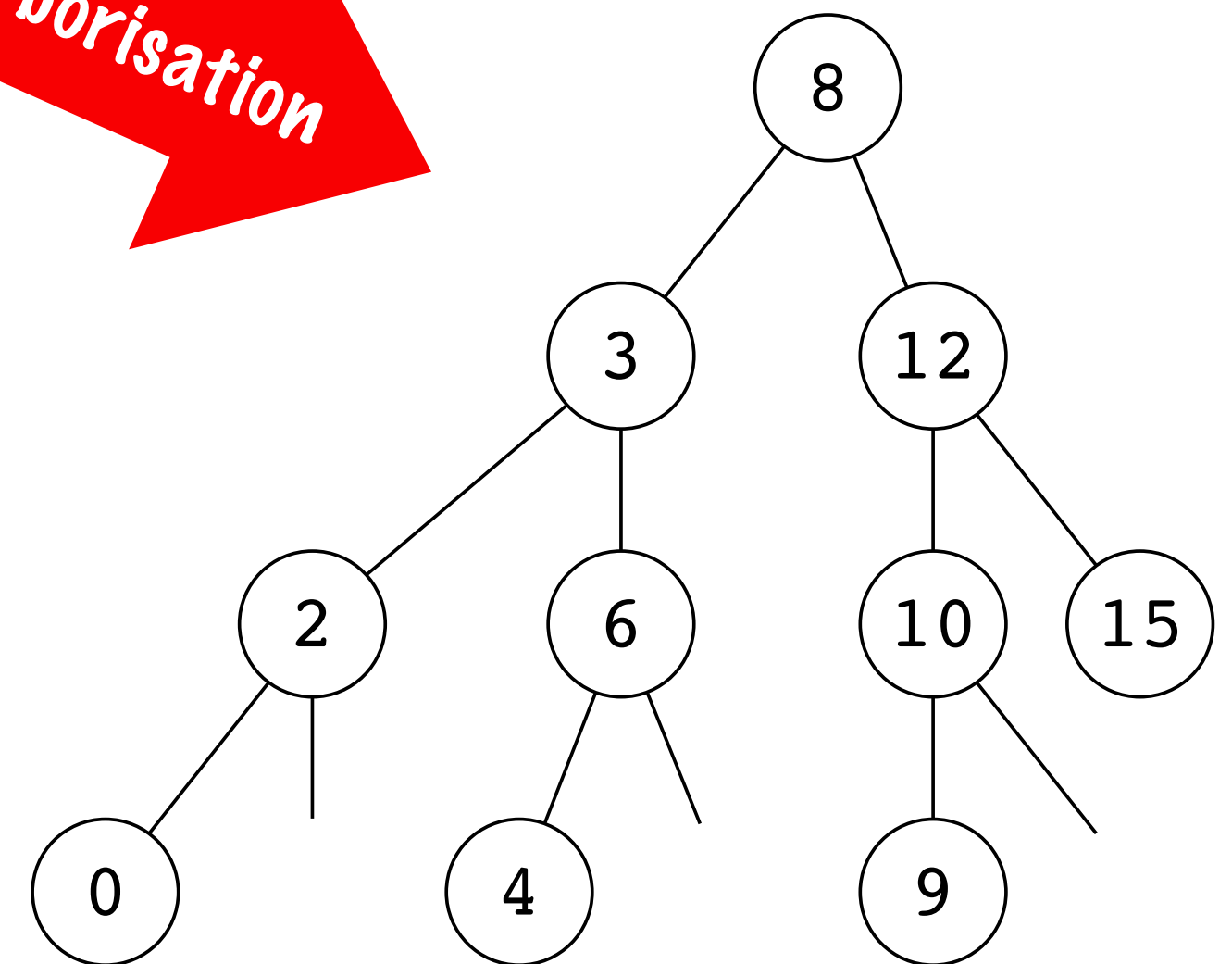
U



=



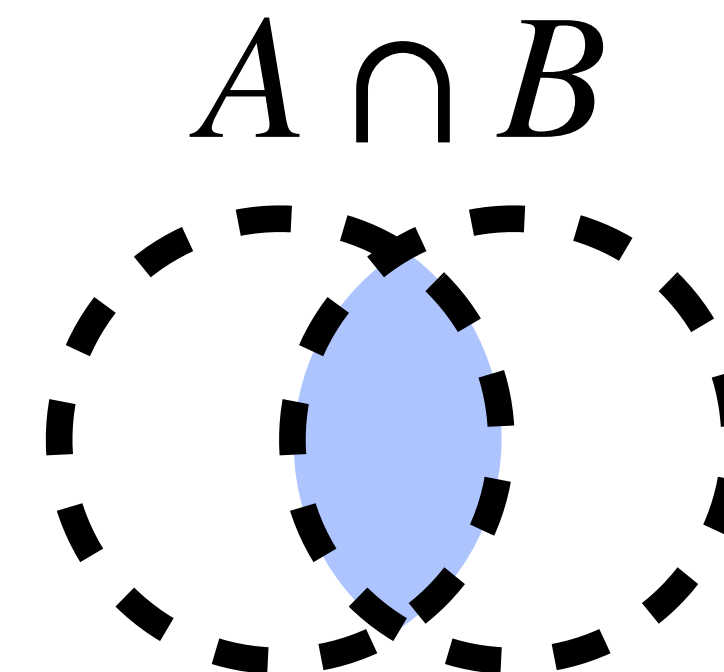
Arborisation



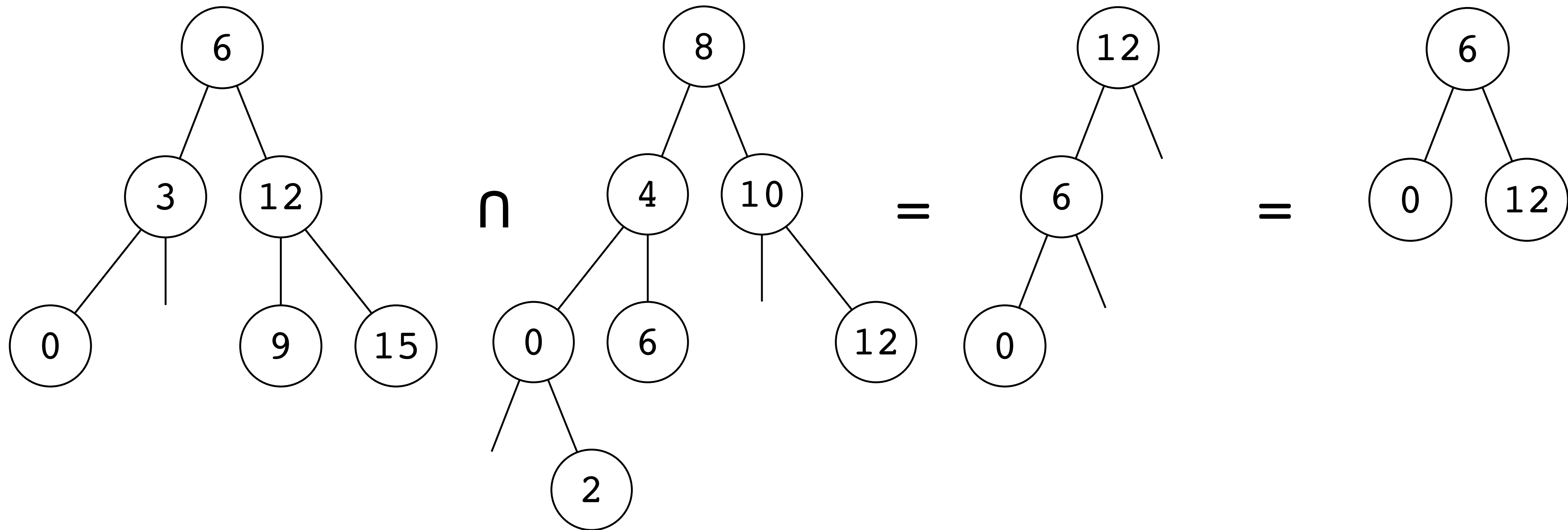
# HE<sup>VD</sup> IG Intersection

- Algorithme similaire en  $O(m + n)$
- Itération simultanée sur les deux arbres par ordre croissant
- N'insérer que les clés présentent dans les deux arbres
- Arboriser le résultat final

```
fonction intersection (A, B)
  I ← ∅
  a ← début(A), b ← début(B)
  tant que a != fin(A) et b != fin(B),
    si a.clé < b.clé,
      a ← suivant(a)
    sinon, si a.clé > b.clé,
      b ← suivant(b)
    sinon,
      n ← nouveau noeud de clé a.clé
      n.gauche ← I, I ← n
      a ← suivant(a), b ← suivant(b)
  retourner I
```



## Exemple



# 17. TDA Tableau Associatif



# TDA Tableau associatif

*(a.k.a. dictionnaire)*



Associe à chaque **clé** unique une **valeur**.

*Peut être vu comme une généralisation du TDA tableau avec des indices de type arbitraire.*

Permet d'écrire

```
age[ "Chiara" ] = 7;
```

Expression correcte en C++ si age est de type

```
std::map < std::string, unsigned >
```



- **Ajouter** - associer une valeur à une nouvelle clé
- **Modifier** - associer une nouvelle valeur à une clé existante
- **Supprimer** - une clé et la valeur associée
- **Chercher** - si une clé existe et quelle est la valeur associée
- **Parcourir, itérer, ...**



ABR dont les noeuds stockent une **valeur** en plus de la **clé**.

- Version **ordonnée** du tableau associatif
- Complexités en  $O(\log(n))$

Tables de hachage

- Version **non-ordonnée** du tableau associatif
- Complexité moyenne en  $O(1)$ , mais  $O(n)$  au pire

## 18. `std::set`



# Conteneurs et algorithmes



Container adapters:

|                       |                                  |
|-----------------------|----------------------------------|
| <b>stack</b>          | LIFO stack (class template )     |
| <b>queue</b>          | FIFO queue (class template )     |
| <b>priority_queue</b> | Priority queue (class template ) |

## Associative containers:

|                 |                                    |
|-----------------|------------------------------------|
| <b>set</b>      | Set (class template )              |
| <b>multiset</b> | Multiple-key set (class template ) |
| <b>map</b>      | Map (class template )              |
| <b>multimap</b> | Multiple-key map (class template ) |

## Unordered associative containers:

|                                                |                                      |
|------------------------------------------------|--------------------------------------|
| <b>unordered_set</b> <small>C++11</small>      | Unordered Set (class template )      |
| <b>unordered_multiset</b> <small>C++11</small> | Unordered Multiset (class template ) |
| <b>unordered_map</b> <small>C++11</small>      | Unordered Map (class template )      |

# `std::set<T, Compare>`

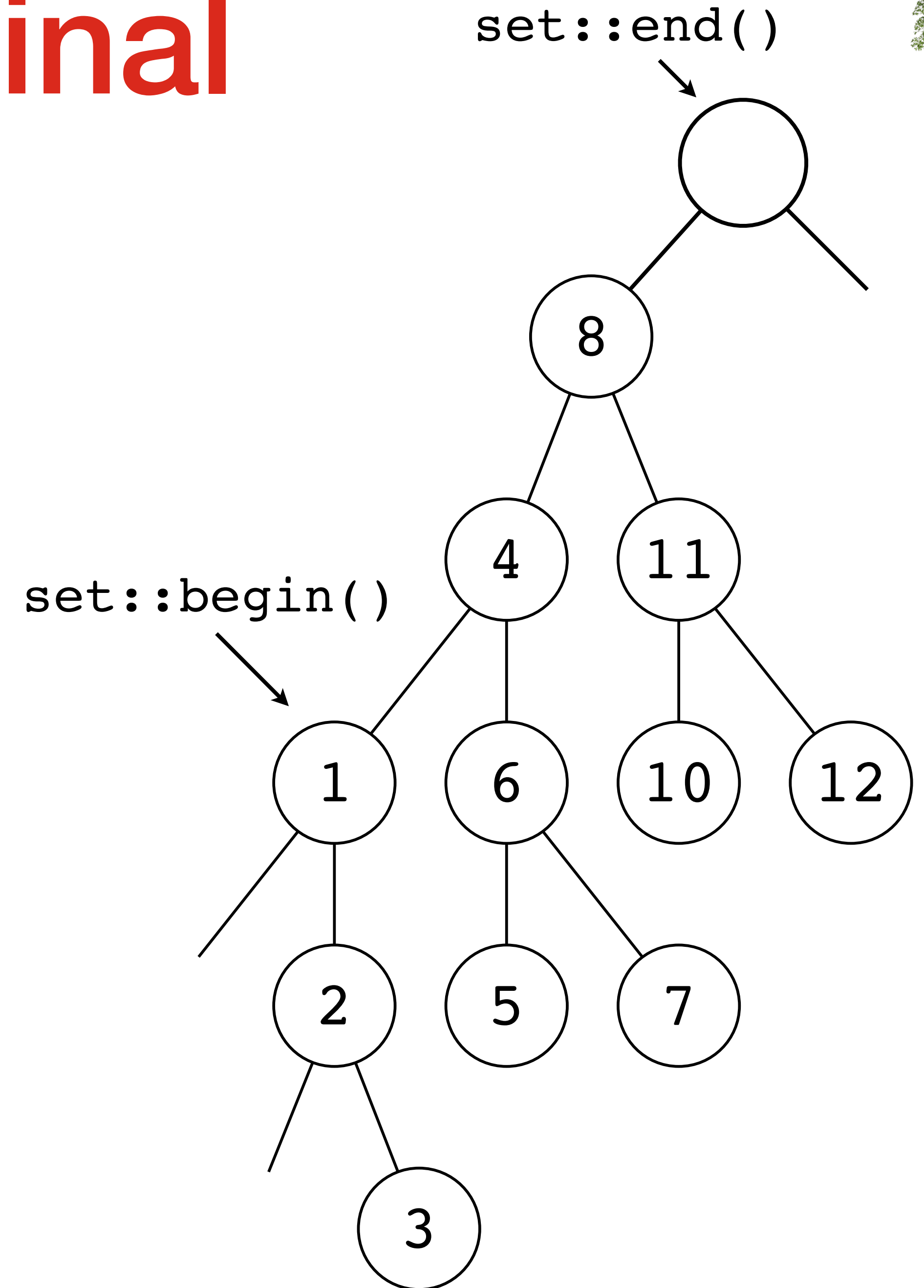


- TDA ensemble trié
  - Éléments non modifiables, seulement insertion, recherche et suppression
- Arbre rouge-noir, une sorte d'arbre binaire de recherche équilibré
- 3 pointeurs par noeud (parent, gauche, droite) pour permettre une itération efficace
- Ne stocke pas la taille dans les noeuds, donc n'offre pas le TDA order statistics tree

# Noeud final



- Pour simplifier la mise en oeuvre des itérateurs
- La racine est à gauche d'un noeud final vide
- L'itérateur `set::end()` pointe vers ce noeud final
- Rend possible une opération telle que `std::prev(s.end())`



# Insertion



```
(1) pair<iterator,bool> insert (const value_type& val);  
    pair<iterator,bool> insert (value_type&& val);  
    pair<iterator,bool> emplace (Args&&... args);  
  
(2) iterator insert (const_iterator p, const value_type& val);  
    iterator insert (const_iterator p, value_type&& val);  
    iterator emplace_hint (const_iterator p, Args&&... args);  
  
(3) void insert (InputIterator first, InputIterator last);  
  
(4) void insert (initializer_list<value_type> il);
```



# Insertion

```
set<int> s {4, 2, 1, 3};  
pair<set<int>::iterator, bool> p;  
p = s.insert(5); // *(p.first) = 5  
                // p.second = true  
p = s.insert(3); // *(p.first) = 3  
                // p.second = false  
for (auto e : s) cout << e; // 12345
```

```
set<int> s {1, 2, 3};  
set<int>::iterator i;  
i = s.insert(s.end(), 5); // *i = 5  
i = s.insert(i, 4);      // *i = 4  
i = s.insert(s.begin(), 0); // *i = 0  
for (auto e : s) cout << e; // 012345
```

```
set<int> s {0, 6, 3};  
list<int> L{4, 1, 2, 3};  
s.insert(L.begin(), L.end());  
for (auto e : s) cout << e; // 012346
```

```
set<int> s {0, 3, 6};  
s.insert({1, 3, 5});  
for (auto e : s) cout << e; // 01356
```

Insertion par la clé en  $O(\log(n))$

Insertion avec indice en  $O(1)$   
s'il est correct

Insertion d'une plage de  $n$  clés dans un  
set de taille  $s$  en  $O(n \cdot \log(n + s))$

Insertion d'une liste d'initialisation



## 3 fonctions disponibles

```
const_iterator find (const value_type& val) const;  
const_iterator lower_bound (const value_type& val) const;  
const_iterator upper_bound (const value_type& val) const;
```

lower\_bound retourne  
“le premier élément  
qui ne va pas à gauche de val”

upper\_bound retourne  
“le premier élément  
qui va à droite de val”

```
set<int> s { 0, 2, 4, 6, 8 };  
set<int>::iterator i;
```

```
i = s.find(2);           // *f = 2  
i = s.lower_bound(2);    // *f = 2  
i = s.upper_bound(2);    // *f = 4
```

```
i = s.find(3);           // f = s.end()  
i = s.lower_bound(3);    // *f = 4  
i = s.upper_bound(3);    // *f = 4
```



## 3 interfaces

```
iterator erase (const_iterator position);  
size_type erase (const value_type& val);  
iterator erase (const_iterator first, const_iterator last);
```

## Complexités en ...

- $O(1)$
- $O(\log(n))$
- $O(\text{distance}(\text{first}, \text{last}))$

```
set<int> s{0,2,4,6,7,8,9};  
  
s.erase(next(s.begin(),3));  
for (auto e : s) cout << e; // 024789  
  
s.erase(4);  
for (auto e : s) cout << e; // 02789  
  
s.erase(s.lower_bound(2),s.upper_bound(8));  
for (auto e : s) cout << e; // 09
```

# 19. `std::map`



`std::map <Key, Value, Compare>`



- Met en oeuvre le TDA tableau associatif
- Essentiellement un `std::set` contenant des éléments de type

`std::pair<const Key, Value>`

dont la fonction de comparaison ne tient compte que des clés

- Et qui fournit l'opérateur crochet

`Value& map<Key, Value>::operator[] (const Key& k)`

# map<K, V> :: operator[]



Selon la documentation, l'opérateur s'écrit...

```
template <typename K, typename V>
V& map<K,V>::operator[](const K& k)
{
    return (*((this->insert(make_pair(k,V()))).first)).second;
}
```

## map&lt;K, V&gt;::operator[] en détail



```
return (*(this->insert(make_pair(k, V()))).first).second;
```

```
template <typename K, typename V>
V& map<K, V>::operator[](const K& k)
{
    V v = V();
    pair<K, V> p = make_pair(k, v);
    pair<map<K, V>::iterator, bool> r = this->insert(p);
    map<K, V>::iterator i = r.first;
    pair<K, V>& q = *i;
    V& w = q.second;
    return w;
}
```



```
string chaine = "abracadabra";  
std::map<char, size_t> m;  
  
size_t i = 0;  
for(char c : chaine) m[c] = i++;  
  
for(auto p : m)  
    cout << "m[" << p.first << "]= " << p.second << "\n";
```

```
m[ a ]=10  
m[ b ]=8  
m[ c ]=4  
m[ d ]=6  
m[ r ]=9
```

Attention, lire `m[ c ]` insère un élément si la clé `c` est absente



```
string chaine = "abracadabra";  
std::map<char, size_t> m;  
size_t i = 0;  
for(char c : chaine) m[c] = i++;  
  
cout << "m.size(): " << m.size() << "\n";
```

`m.size(): 5`

Attention, lire `m[ c ]` insère un élément si la clé `c` est absente



```
string chaine = "abracadabra";  
std::map<char, size_t> m;  
size_t i = 0;  
for(char c : chaine) m[c] = i++;  
  
cout << "m.size(): " << m.size() << "\n";  
  
for(char c = 'c'; c <= 'e'; ++c)  
    cout << "m[" << c << "]= " << m[c] << "\n";
```

`m.size() : 5`

`m[c]=4`

`m[d]=6`

`m[e]=0`

Attention, lire `m[ c ]` insère un élément si la clé `c` est absente



```
string chaine = "abracadabra";
std::map<char, size_t> m;
size_t i = 0;
for(char c : chaine) m[c] = i++;

cout << "m.size(): " << m.size() << "\n";

for(char c = 'c'; c <= 'e'; ++c)
    cout << "m[" << c << "]= " << m[c] << "\n";

cout << "m.size(): " << m.size() << "\n";
```

`m.size() : 5`

`m[c]=4`

`m[d]=6`

`m[e]=0`

`m.size() : 6`

# map<K, V>::find(K const& k)



*// On peut l'éviter en utilisant la fonction find.*

```
std::cout << "m.size(): " << m.size() << "\n";
```

```
for(char c = 'd'; c <= 'f'; ++c)
{
```

```
    auto it = m.find(c);
```

```
    if(it != m.end())
```

```
        std::cout << "m[" << c << "]= " << (*it).second << "\n";
```

```
    else
```

```
        std::cout << "m[" << c << "] est absent\n";
```

```
}
```

m.size(): 6

m[d]=6

m[e]=0

m[f] est absent



*// On peut l'éviter en utilisant la fonction find.*

```
std::cout << "m.size(): " << m.size() << "\n";
```

```
for(char c = 'd'; c <= 'f'; ++c)  
{
```

```
    auto it = m.find(c);
```

```
    if(it != m.end())
```

```
        std::cout << "m[" << c << "]= " << (*it).second << "\n";
```

```
    else
```

```
        std::cout << "m[" << c << "] est absent\n";
```

```
}
```

```
std::cout << "m.size(): " << m.size() << "\n";
```

```
m.size(): 6
```

```
m[d]=6
```

```
m[e]=0
```

```
m[f] est absent
```

```
m.size(): 6
```

# multiset et multimap



- Comme set et map, mais permettent de stocker plusieurs fois la même clé
- Pas d'operator[ ] pour multimap ; il serait ambigu
- La méthode equal\_range retourne toutes les clés équivalentes

```
template <typename T>
pair<multiset<T>::iterator, multiset<T>::iterator>
multiset<T>::equal_range (const T& k)
{
    return make_pair(lower_bound(k), upper_bound(k));
}
```



Coût fixe :

- 1 pointeur vers le noeud minimum `set::begin()`
- 1 pointeur vers le noeud final `set::end()`
- 1 `size_t` pour stocker la taille

$2 * \text{sizeof}(T^*) +$   
 $1 * \text{sizeof}(\text{size\_t})$

Coût par noeud :

- 2 pointeurs vers les enfants
- 1 pointeur vers le parent
- Pour l'information d'équilibrage :  
1 bit pour les arbres rouge-noir
- Évidemment la mémoire pour stocker la clé  
(et la valeur pour `std::map`)
- Plus éventuellement des bytes d'alignement,  
mais nous n'en tenons pas compte en ASD

$3 * \text{sizeof}(\text{void}^*) + \dots + \text{sizeof}(\text{Key}) +$   
 $\text{sizeof}(\text{Value})$

# 19 . STL set theory operations + output iterators





# <algorithm>

**Merge** (operating on sorted ranges):

|                                 |                                                                              |
|---------------------------------|------------------------------------------------------------------------------|
| <b>merge</b>                    | Merge sorted ranges (function template )                                     |
| <b>inplace_merge</b>            | Merge consecutive sorted ranges (function template )                         |
| <b>includes</b>                 | Test whether sorted range includes another sorted range (function template ) |
| <b>set_union</b>                | Union of two sorted ranges (function template )                              |
| <b>set_intersection</b>         | Intersection of two sorted ranges (function template )                       |
| <b>set_difference</b>           | Difference of two sorted ranges (function template )                         |
| <b>set_symmetric_difference</b> | Symmetric difference of two sorted ranges (function template )               |



```
template <class InputIterator1, class InputIterator2,  
          class OutputIterator>  
OutputIterator set_xxx ( InputIterator1 first1, InputIterator1 last1,  
                        InputIterator2 first2, InputIterator2 last2,  
                        OutputIterator result);
```

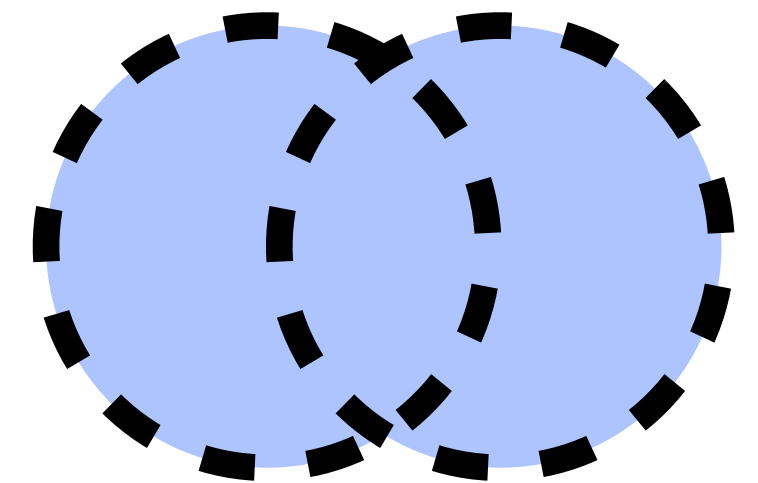
- En entrée, des plages d'éléments triés
- Ecrivent dans un itérateur de sortie
- Retournent l'itérateur suivant le dernier écrit

## set\_union



```
template <class InputIter1, class InputIter2, class OutputIter>
OutputIter set_union ( InputIter1 first1, InputIter1 last1,
                      InputIter2 first2, InputIter2 last2,
                      OutputIter result)
{
    while (true)
    {
        if (first1==last1) return std::copy(first2,last2,result);
        if (first2==last2) return std::copy(first1,last1,result);

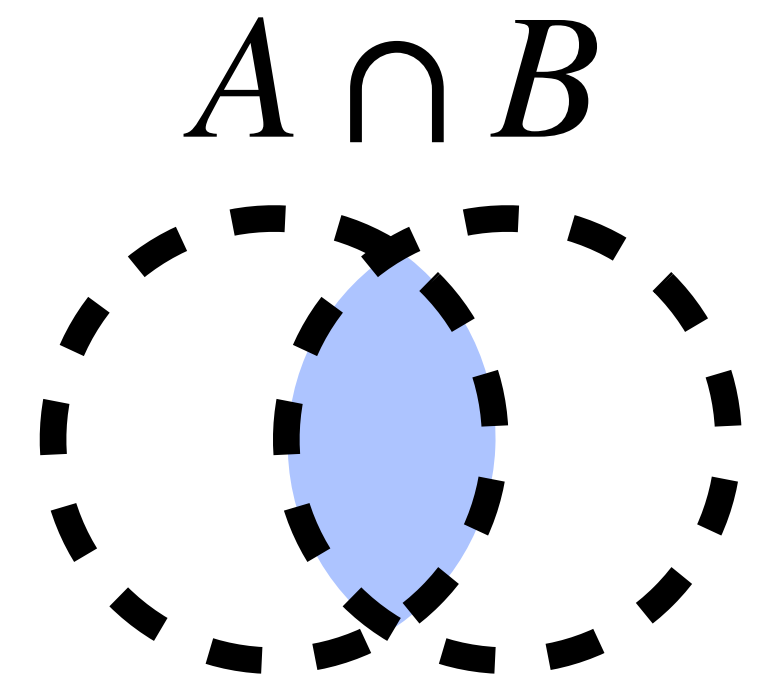
        if (*first1<*first2) { *result = *first1; ++first1; }
        else if (*first2<*first1) { *result = *first2; ++first2; }
        else { *result = *first1; ++first1; ++first2; }
        ++result;
    }
}
```

 $A \cup B$ 

## set\_intersection



```
template <class InputIter1, class InputIter2, class OutputIter>
OutputIter set_intersection (InputIter1 first1, InputIter1 last1,
                             InputIter2 first2, InputIter2 last2,
                             OutputIter result)
{
    while (first1!=last1 && first2!=last2)
    {
        if (*first1<*first2) ++first1;
        else if (*first2<*first1) ++first2;
        else {
            *result = *first1;
            ++result; ++first1; ++first2;
        }
    }
    return result;
}
```



# En entrée ?



```
#include <set>
std::set<int> E1 = {5, 10, 15, 20, 25, 5, 10};
// E1 contient {5, 10, 15, 20, 25}
```

```
#include <list>
std::list<int> E2 = { 50, 40, 10, 20, 30, 10, 40 };
E2.sort();
E2.unique(); // E2 contient {10, 20, 30, 40, 50}
```

```
#include <forward_list>
std::forward_list<int> E3 = { 5, 10, 15, 20, 25, 5, 10};
E3.sort();
E3.unique(); // E3 contient {5, 10, 15, 20, 25}
```

```
#include <vector>
#include <algorithm>
std::vector<int> E4 = { 50, 40, 10, 20, 30, 10, 40 };
std::sort(E4.begin(), E4.end());
auto last = std::unique(E4.begin(), E4.end());
E4.erase(last, E4.end()); // E4 contient {10, 20, 30, 40, 50}
```

# En sortie ?



- La seule approche que nous connaissons consiste à allouer suffisamment (trop) avant l'appel et à effacer l'excédent après.

```
vector<int> U(E1.size()+E2.size());  
auto lastU = std::set_union(E1.begin(), E1.end(),  
                           E2.begin(), E2.end(),  
                           U.begin());  
U.erase(lastU, U.end());
```

- Comment faire avec `list`, `forward_list`, `set` ?



- `back_inserter`,  
`front_inserter`,  
`inserter`
- Conservent une référence  
au conteneur qu'ils itèrent
- Appèlent `push_back`,  
`push_front` ou `insert`  
quand ils sont référencés à  
gauche d'une affectation

| Expression                                                       | Propriété                                                                                                   |
|------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------|
| <code>x b(a);</code><br><code>b = a;</code>                      | Constructible par copie, assignable et destructible                                                         |
| <code>*a = t;</code>                                             | Déréférençable à gauche, seulement à gauche d'une affectation, et une seule fois entre deux incrémentations |
| <code>++a;</code><br><code>a++;</code><br><code>*a++ = t;</code> | Incrémentable                                                                                               |
| <code>swap(a, b)</code>                                          | Les lvalues sont échangeables                                                                               |

# Itérateurs de sortie



```
#include <iterator>
list<int> I;    // ou vector<int> ou deque<int>
std::set_intersection(E1.begin(), E1.end(),
                     E2.begin(), E2.end(),
                     std::back_inserter(I));
```

```
forward_list<int> D;
std::set_difference(E1.begin(), E1.end(),
                  E2.begin(), E2.end(),
                  std::front_inserter(D));
D.reverse();    (insérer au début inverse le résultat)
```

```
set<int> SD;
std::set_symmetric_difference(E1.begin(), E1.end(),
                             E2.begin(), E2.end(),
                             std::inserter(SD, SD.end()));
```