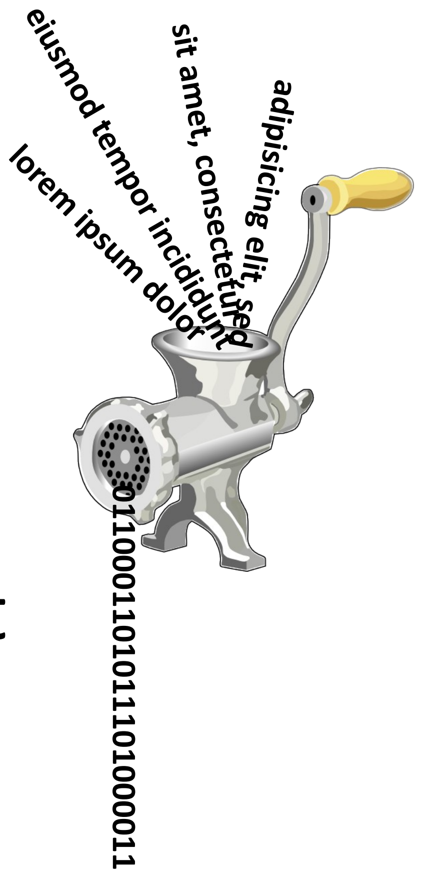
ÉCOLE NATIONALE DES PONTS ET CHAUSSÉES

IP PARIS

PRALG : séance 7

Ensembles, tables associatives, et hachage



Pascal Monasse/Renaud Marlet
Laboratoire LIGM-IMAGINE

Motivation

- Manipuler efficacement des **grands ensembles** (≠ vecteur)
  - test rapide d'appartenance, intersection, union...
- Implémenter des **tableaux associatif** (indices ≠ entier)
  - ```
    population["Paris"] = 2243739;
    cout << population[ville];
    ```
- Notions de **fonctions de hachage**
  - probabilités et théorie des nombres
- Prétexte à des **compléments en C++**
  - types de données de la STL, surcharge d'opérateurs, spécialisation de template

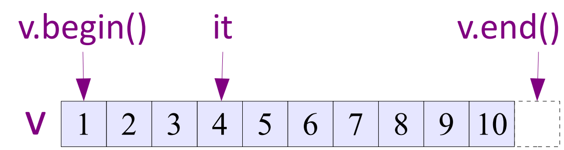
= usages très courants
    ```

Notations pour mesurer la complexité

- **Borne supérieure asymptotique : $O(\ldots)$**
  - $O(g(n)) = \{ f \mid \exists\, n_0, c > 0 \text{ tq } \forall\, n \geq n_0,\ 0 \leq f(n) \leq c\, g(n) \}$
- **Borne inférieure asymptotique : $\Omega(\ldots)$**
  - $\Omega(g(n)) = \{ f \mid \exists\, n_0, c > 0 \text{ tq } \forall\, n \geq n_0,\ 0 \leq c\, g(n) \leq f(n) \}$
- **Encadrement asymptotique : $\Theta(\ldots)$**
  - $$\begin{aligned}\Theta(g(n)) &= O(g(n)) \cap \Omega(g(n)) \\ &= \{ f \mid \exists\, n_0, c_1, c_2 > 0 \text{ tq } \forall\, n \geq n_0, \\ &\qquad 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \}\end{aligned}$$

Rappels C++ : itérateurs de la STL

- **C++98, C++03**




```
vector<int> v(10);
for(int i=0; i<v.size(); i++)
  v[i]=i+1;                 // v : 1 2 3 4 5 6 7 8 9 10
for(vector<int>::iterator it = v.begin();
    it != v.end(); ++it) {
  cout << *it << ' '; // Affiche 1 2 3 4 5 6 7 8 9 10
  *it = (*it)/2;            // v : 0 1 1 2 2 3 3 4 4 5
}
```

- **C++11** auto-typage et for-range

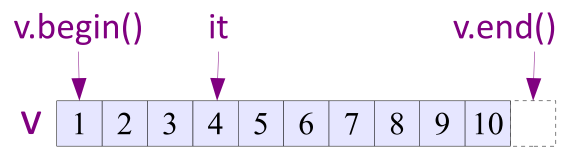
```
for(auto it=v.begin(); it!=v.end();++it)
```

```
for(auto it : v)
```

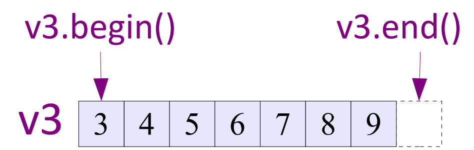
Mais attention, typage plus faible.
Plus on spécifie (des types), plus
on peut détecter des incohérences.

Rappels C++ : itérateurs et initialisations

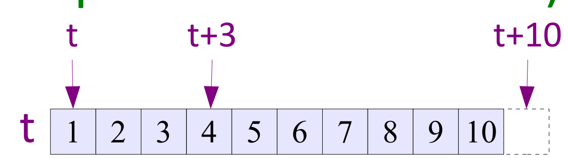
- **C++98, C++03, C++11**




```
vector<int> v(10);
// v : 0 0 0 0 0 0 0 0 0 0   // initialisation à 0 pour les types de base (ici int)
for(int i=0;i<v.size();i++) v[i]=i+1;
// v : 1 2 3 4 5 6 7 8 9 10
vector<int> v2(v.begin(),v.end());
// v2 : 1 2 3 4 5 6 7 8 9 10
vector<int> v3(v.begin()+2,v.end()-1);
// v3 : 3 4 5 6 7 8 9
```




```
vector<int> v4(v.begin(),v3.end());  // Pas de sens !
// v4 : indéterminé, erreur possible à l'exécution (mais pas nécessairement)
int t[] = {1,2,3,4,5,6,7,8,9,10};
// t : 1 2 3 4 5 6 7 8 9 10
vector<int> v5(t+1,t+6);
// v5 : 2 3 4 5 6
```




```
vector<int> v6(t,t+sizeof(t)/sizeof(t[0]));  // t+10
// v6 : 1 2 3 4 5 6 7 8 9 10
```

40

sizeof(int) = 4

Rappels C++ : construction avec initialisation

- **C++98, C++03**

```
int t[] = {1, 2, 3, 4, 5};                  //t:1 2 3 4 5
vector<int> v(t,t+5);                       //v:1 2 3 4 5
vector<int> v(t,t+sizeof(t)/sizeof(t[0]));
```

- **C++11** (uniform initialization et initializer_list)

```
vector<int> v({1, 2, 3, 4, 5});   //v:1 2 3 4 5
vector<int> v {1, 2, 3, 4, 5};    //v:1 2 3 4 5
vector<int> v={1, 2, 3, 4, 5};    //v:1 2 3 4 5
```

- Idem pour `list<T>`, etc.

Type de données « ensemble »

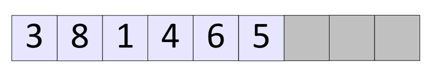
- Opérations de base (élémentaires), outre la création
  - chercher : $x \in X$?
  - insérer : $X \leftarrow X \cup \{x\}$
  - supprimer : $X \leftarrow X \setminus \{x\}$
- Opérations générales (auxiliaires)
  - intersection : $X \cap Y$
  - union : $X \cup Y$
  - complément (par rapport à un surensemble donné) : $\overline{X}$
  - cardinal : $|X|$, etc.

Type de données « ensemble »

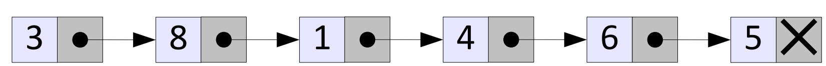
- Opérations de base
  - chercher : $x \in X$?
  - insérer : $X \leftarrow X \cup \{x\}$
  - supprimer : $X \leftarrow X \setminus \{x\}$
- Implémentation
  - ???

Type de données « ensemble »

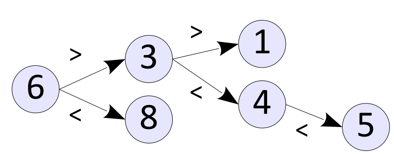
- Opérations de base
  - chercher : $x \in X$?
  - insérer : $X \leftarrow X \cup \{x\}$
  - supprimer : $X \leftarrow X \setminus \{x\}$
- Implémentation
  - tableau, vecteur



  - liste chaînée



  - arbre binaire

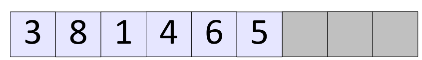


Type de données « ensemble »

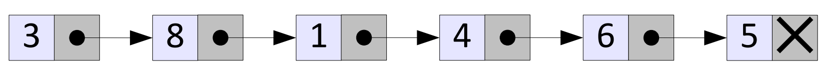
- Opérations de base
  - chercher : $x \in X$?
  - insérer : $X \leftarrow X \cup \{x\}$
  - supprimer : $X \leftarrow X \setminus \{x\}$

Complexité de ces opérations ?

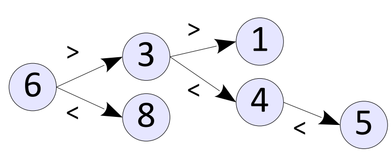
- Implémentation
  - tableau, vecteur



  - liste chaînée



  - arbre binaire

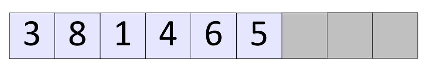


Type de données « ensemble »

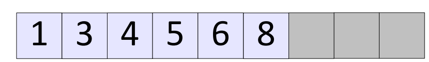
- Opérations de base
  - chercher : $x \in X$?
  - insérer : $X \leftarrow X \cup \{x\}$
  - supprimer : $X \leftarrow X \setminus \{x\}$

Complexité de ces opérations ?

- Implémentation
  - tableau, vecteur

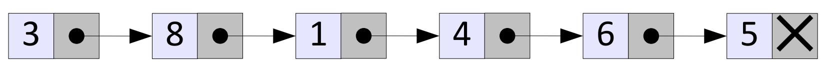


[si vrac]

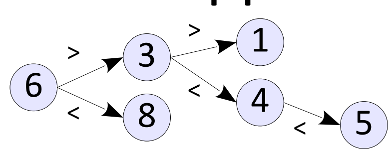


recherche $O(\log n)$ si ordonné

    - chercher, insérer et supprimer : $O(n)$ moyenne et pire cas
  - liste chaînée



    - chercher, insérer et supprimer : $O'(n)$ moyenne et pire cas
  - arbre binaire



    - chercher, insérer et supprimer : $O(\log n)$ moyenne, $O(n)$ pire cas

$O(\log n)$ si équilibré (red-black tree, AVL, ...)

Type de données « ensemble » en C++, dans la STL

- Classe et opérations
  - `set<T> s;`
  - `r=s.insert(x);`
    - `r.second` : booléen qui dit si `x` est nouveau dans `s`

r.first : iterator sur élément inséré ou déjà présent

  - `r=s.erase(x);`
    - `r` : nombre d'éléments réellement supprimés (0 si `x` déjà dans `s`)
  -
```
for(set<T>::iterator it = s.begin();
    it != s.end(); ++it)  // élément : x=*it
  Operation(*it);
```
  - `it=s.find(x);`
    - si `it == s.end()`, alors `x` n'a pas été trouvé
    - sinon `x` est trouvé, `*it == x` et `s.erase(it)` possible

Type de données « ensemble » en C++, dans la STL

- Classe et opérations : complexité (moyenne, pire cas)
  - `set<T> s;` : $O(1)$
  - `r=s.insert(x);` : $O(\log n)$ où n est la taille de `s`
    - `r.second` : booléen qui dit si `x` est nouveau dans `s`
  - `r=s.erase(x);` : $O(\log n)$
    - `r` : nombre d'éléments réellement supprimés (0 si `x` déjà dans `s`)
  - ```
    for(set<T>::iterator it = s.begin();
        it != s.end(); ++it) // x=*it
      Operation(*it);
    ```
    : $O(n)$
  - `it=s.find(x);` : $O(\log n)$
    - si `it == s.end()`, alors `x` n'a pas été trouvé
    - sinon `x` est trouvé, `*it == x` et `s.erase(it)` possible : $O(1)$
    ```

Type de données « ensemble » en C++, dans la STL

- Classe et opérations : complexité (moyenne, pire cas)
  - `set<T> s;` : $O(1)$
  - `r=s.insert(x);` : $O(\log n)$ où n est la taille de `s`
    - `r.second` : booléen qui dit si `x` est nouveau dans `s`
  - `r=s.erase(x);` : $O(\log n)$
    - `r` : nombre d'éléments réellement supprimés (0 si `x` déjà dans `s`)
  - ```
    for(set<T>::iterator it = s.begin();
        it != s.end(); ++it) // x=*it
      Operation(*it);
    ```
    : $O(n)$
  - `it=s.find(x);` : $O(\log n)$
    - si `it == s.end()`, alors `x` n'a pas été trouvé
    - sinon `x` est trouvé, `*it == x` et `s.erase(it)` possible : $O(1)$

Comment savoir ces complexités ?
    ```

Où trouver l'information : RTFM

Beaucoup d'informations disponibles sur le web :
- ex. ici pour la STL sur **cplusplus.com**

Autres sites fiables et riches :
- **stackoverflow.com**
- **cppreference.com**
- **cprogramming.com**

```
  std::cout << "myset contains:";
  for (it=myset.begin(); it!=myset.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Output:

```
myset contains: 10 30 50
```

Complexity

For the first version (`erase(position)`), amortized constant.
For the second version (`erase(val)`), logarithmic in container size.
For the last version (`erase(first,last)`), logarithmic in container set::size plus linear in the distance between *first* and *last*.

Iterator validity

Iterators, pointers and references referring to elements removed by the function are invalidated.
All other iterators, pointers and references keep their validity.

Data races

The container is modified.
The elements removed are modified. Concurrently accessing other elements is safe, although iterating ranges in the

Où trouver l'information : RTFM

Beaucoup d'informations disponibles sur le web :
- ex. ici pour la STL sur **cplusplus.com**

Autres sites fiables et riches :
- **stackoverflow.com**
- **cppreference.com**
- **cprogramming.com**

☛ Pensez à inclure ces informations quand vous écrivez vos propres bibliothèques

```
  std::cout << "myset contains:";
  for (it=myset.begin(); it!=myset.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Output:

```
myset contains: 10 30 50
```

Complexity

For the first version (`erase(position)`), amortized constant.
For the second version (`erase(val)`), logarithmic in container size.
For the last version (`erase(first,last)`), logarithmic in container set::size plus linear in the distance between *first* and *last*.

Iterator validity

Iterators, pointers and references referring to elements removed by the function are invalidated.
All other iterators, pointers and references keep their validity.

Data races

The container is modified.
The elements removed are modified. Concurrently accessing other elements is safe, although iterating ranges in the

Type de données « ensemble » : Construction avec initialisation

- **C++98, C++03**

```
int t[] = {1, 4, 2, 3, 2};          // t: 1 4 2 3 2
set<int> s(t,t+5);                  // s: 1 2 3 4
set<int> s(t,t+sizeof(t)/sizeof(t[0]));// s: 1 2 3 4
```

- **C++11**

```
set<int> s({1, 4, 2, 3, 2});        // s: 1 2 3 4
set<int> s {1, 4, 2, 3, 2};         // s: 1 2 3 4
set<int> s={1, 4, 2, 3, 2};         // s: 1 2 3 4
```

☛ Équivalent à des insertions successives

```
set<int> s; s.insert(1); s.insert(4); …
```

Type de données « ensemble » : Opérations générales

- Opérations générales **via les opérations élémentaires**
  - insertion d'un élément : $S \leftarrow S \cup \{x\}$
  - suppression d'un élément : $S \leftarrow S \setminus \{x\}$
- Union : $S_2 \leftarrow S_2 \cup S_1$
  - ???
- Différence : $S_3 \leftarrow S_3 \setminus S_1$
  - ???
- Intersection : $S_1 \leftarrow S_1 \cap S_4$
  - ???

Type de données « ensemble » : Opérations générales

- Opérations générales **via les opérations élémentaires**
  - insertion d'un élément : $S \leftarrow S \cup \{x\}$
  - suppression d'un élément : $S \leftarrow S \setminus \{x\}$
- Union : $S_2 \leftarrow S_2 \cup S_1$
  - $\forall x \in S_1,\ S_2 \leftarrow S_2 \cup \{x\}$
- Différence : $S_3 \leftarrow S_3 \setminus S_1$
  - $\forall x \in S_1,\ S_3 \leftarrow S_3 \setminus \{x\}$
- Intersection : $S_1 \leftarrow S_1 \cap S_4$
  - $\forall x \in S_1,$ if $x \notin S_4$ then $S_1 \leftarrow S_1 \setminus \{x\}$

Type de données « ensemble »
Opérations générales en C++, dans la STL

- Opérations générales **via les itérateurs**

```
for(set<T>::iterator it1=s1.begin();
    it1 != s1.end(); ++it1) {
  // Union : s2 ← s2 ∪ s1
  s2.insert(*it1);

  // Différence : s3 ← s3 \ s1
  s3.erase(*it1);

  // Intersection : s1 ← s1 ∩ s4
  if(s4.find(*it1)==s4.end()) // Si pas trouvé,
    s1.erase(it1);            // le supprimer.
}  // ☛ L'argument est l'itérateur it1, pas l'élément *it1 !
```

Type de données « ensemble »
Opérations générales en C++, dans la STL

- Opérations générales **via les itérateurs**

```
for(set<T>::iterator it1=s1.begin();
    it1 != s1.end(); ++it1) {

  // Union : s2 ← s2 ∪ s1
  s2.insert(*it1);

  // Différence : s3 ← s3 \ s1
  s3.erase(*it1);

  // Intersection : s1 ← s1 ∩ s4
  if(s4.find(*it1)==s4.end()) // Si pas trouvé,
    s1.erase(it1);            // le supprimer.
}   // ☛ L'argument est l'itérateur it1, pas l'élément *it1 !
```

Complexité de ces opérations générales ?

Type de données « ensemble » Opérations générales en C++, dans la STL

- Opérations générales **via les itérateurs**

```
for(set<T>::iterator it1=s1.begin();
    it1 != s1.end(); ++it1) {

  // Union : s2 ← s2 ∪ s1                 : O(n1 log n2)
  s2.insert(*it1);

  // Différence : s3 ← s3 \ s1            : O(n1 log n3)
  s3.erase(*it1);

  // Intersection : s1 ← s1 ∩ s4          : O(n1 log n4)
  if(s4.find(*it1)==s4.end()) // Si pas trouvé,
    s1.erase(it1);            // le supprimer.
}   // ☛ L'argument est l'itérateur it1, pas l'élément *it1 !
```

Complexités : Union $O(n_1 \log n_2)$, Différence $O(n_1 \log n_3)$, Intersection $O(n_1 \log n_4)$.

Type de données « ensemble » Opérations générales en C++, dans la STL

- Opérations générales **via les itérateurs**

```
for(set<T>::iterator it1=s1.begin();
    it1 != s1.end(); ++it1) {
  // Union : s2 ← s2 ∪ s1                : O(n1 log n2)
  s2.insert(*it1);
  // Différence : s3 ← s3 \ s1           : O(n1 log n3)
  s3.erase(*it1);
  // Intersection : s1 ← s1 ∩ s4         : O(n1 log n4)
  if(s4.find(*it1)==s4.end()) // Si pas trouvé,
    s1.erase(it1);            // le supprimer.
} // Vaut-il mieux itérer sur s1, s2, s3 ou s4 ?
```

Complexités : Union $O(n_1 \log n_2)$; Différence $O(n_1 \log n_3)$; Intersection $O(n_1 \log n_4)$

Type de données « ensemble » Opérations générales en C++, dans la STL

- Opérations générales **via les itérateurs**

```
for(set<T>::iterator it1=s1.begin();
    it1 != s1.end(); ++it1) {
  // Union : s2 ← s2 ∪ s1            : O(n1 log n2)
  s2.insert(*it1);

  // Différence : s3 ← s3 \ s1       : O(n1 log n3)
  s3.erase(*it1);

  // Intersection : s1 ← s1 ∩ s4     : O(n1 log n4)
  if(s4.find(*it1)==s4.end()) // Si pas trouvé,
    s1.erase(it1);            // le supprimer.
} // ☛ Choisir d'itérer sur le plus petit ensemble (hyp. n1)
```

Complexités : Union $O(n_1 \log n_2)$; Différence $O(n_1 \log n_3)$; Intersection $O(n_1 \log n_4)$; hyp. n_1.

Type de données « ensemble » Opérations générales en C++, dans la STL

- Opérations générales **via les itérateurs**

```
for(set<T>::iterator it1=s1.begin();
    it1 != s1.end(); ++it1) {
  // Union : s2 ← s2 ∪ s1              : O(n1 log n2)
  s2.insert(*it1);

  // Différence : s3 ← s3 \ s1          : O(n1 log n3)
  s3.erase(*it1);

  // Intersection : s1 ← s1 ∩ s4        : O(n1 log n4)
  if(s4.find(*it1)==s4.end()) // Si pas trouvé,
    s1.erase(it1);            // le supprimer.
}
// ☛ Choisir d'itérer sur le plus petit ensemble (hyp. n1)
```

Complexités : Union $O(n_1 \log n_2)$; Différence $O(n_1 \log n_3)$; Intersection $O(n_1 \log n_4)$; (hyp. n_1)

Rien ne vous choque ?

Où trouver l'information : RTFM

Doc de set<T>::erase sur **cplusplus.com**

```
  std::cout << "myset contains:";
  for (it=myset.begin(); it!=myset.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Output:

```
myset contains: 10 30 50
```

☛ Attention : it1 non valide après erase
donc ++it1 est un bug

Complexity

For the first version (`erase(position)`), amortized constant.
For the second version (`erase(val)`), logarithmic in container size.
For the last version (`erase(first,last)`), logarithmic in container set::size plus linear in the distance between *first* and *last*.

Iterator validity

Iterators, pointers and references referring to elements removed by the function are invalidated.
All other iterators, pointers and references keep their validity.

Data races

The container is modified.
The elements removed are modified. Concurrently accessing other elements is safe, although iterating ranges in the

Type de données « ensemble »
Opérations générales en C++, dans la STL

- Opérations générales **via les itérateurs**
  - // Intersection : s1 ← s1 ∩ s4

```
for(set<T>::iterator it1=s1.begin();
      it1 != s1.end();)
    if(s4.find(*it1)==s4.end()){ // Si pas trouvé,
      set<T>::iterator it2=it1++;
      s1.erase(it2);              // le supprimer.
    } else
      ++it1;
}
```

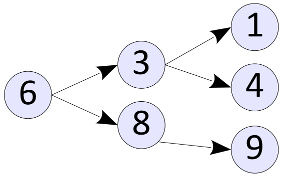
Type des éléments d'un ensemble

- Type arbitraire en paramètre du template (STL)

```
template<T> class set { … }

set<T> s;
T x = …;
s.insert(x);
```

- Rappel : implémentation efficace
  - ex. arbre binaire équilibré



Type des éléments d'un ensemble

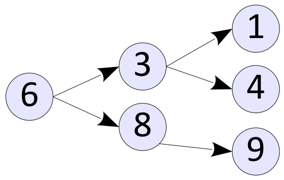
- Type arbitraire en paramètre du template (STL)

```
template<T> class set { … }
```

```
set<T> s;
T x = …;
s.insert(x);
```

Il ne manque rien ?

- Rappel : implémentation efficace
  - ex. arbre binaire équilibré

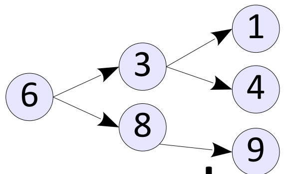


Type des éléments d'un ensemble

- Type **+/- arbitraire** en paramètre du template (STL)

```
template<T> class set { … }
```

```
set<T> s;
T x = …;
s.insert(x);
```

- Rappel : implémentation efficace
  - ex. arbre binaire équilibré 

  - ça suppose de pouvoir **ordonner** les éléments de l'ensemble
    - N.B. ordre total (pas partiel)

Ordre des éléments d'un ensemble

- Ordre par défaut sur les types C++ : via « < »
- Type scalaire : ex. entiers (idem `float`, etc.)

```
set<int> s; // Initialement vide
s.insert(4);
s.insert(1);
s.insert(3);
s.insert(4);
s.insert(2);

for (set<int>::iterator it = s.begin();
     it != s.end(); ++it)
  cout << *it << ' '; // Affiche : 1 2 3 4
cout << endl;      //   (pas 4 1 3 2, ni 2 3 1 4)
```

Ordre des éléments d'un ensemble

- Caractères

```
set<char> s;
s2.insert('O');
s2.insert('o');
s2.insert('0');

for (set<char>::iterator it = s.begin();
     it != s.end(); ++it)
  cout<< *it <<' '; // quel ordre ???
```

- String

```
set<string> s;
s.insert("foo");
s.insert("bar");
s.insert("baz");
```

Ordre des éléments d'un ensemble

- Caractères

```
set<char> s;
s2.insert('O');
s2.insert('o');
s2.insert('0');
```

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

```
for (set<char>::iterator it = s.begin();
     it != s.end(); ++it)
  cout<< *it <<' '; //en général,ordre ASCII: 0 O o
```

- String

```
set<string> s;
s.insert("foo");
s.insert("bar");
s.insert("baz"); // quel ordre ???
```

Pas défini dans le standard C++.
Seules certitudes :

- '0' < '1' < … < '9'
- 'A' < 'B' < … < 'Z'
- 'a' < 'b' < … < 'z'

Ordre des éléments d'un ensemble

- Caractères

```
set<char> s;
s2.insert('O');
s2.insert('o');
s2.insert('0');
```

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

```
for (set<char>::iterator it = s.begin();
     it != s.end(); ++it)
  cout<< *it <<' '; //en général,ordre ASCII: 0 O o
```

- String

```
- set<string> s;
  s.insert("foo");
  s.insert("bar");
  s.insert("baz"); // Ordre lexicographique : bar baz foo
```

Pas défini dans le standard C++.
Seules certitudes :
- '0' < '1' < … < '9'
- 'A' < 'B' < … < 'Z'
- 'a' < 'b' < … < 'z'

Ordre des éléments d'un ensemble

- Pointeur : basé sur l'adresse mémoire ≈ scalaire

```
set <int*> s;
int x[] = {1,2,3};
int y[] = {4,5,6};
s.insert(x);
s.insert(y);
cout << s.size(); // combien d'éléments ???
cout << (x < y);  // quel ordre ???

int z[] = {1,2,3};
s.insert(z);
cout << s.size(); // combien d'éléments ???
cout << (x == z); // égaux ???
```

Ordre des éléments d'un ensemble

- Pointeur : basé sur l'adresse mémoire ≈ scalaire

```
set <int*> s;
int x[] = {1,2,3};
int y[] = {4,5,6};
s.insert(x);
s.insert(y);
cout << s.size(); // 2 éléments
cout << (x < y);  // 0 ou 1 selon mémoire

int z[] = {1,2,3};
s.insert(z);
cout << s.size(); // 3 éléments
cout << (x == z); // 0 car pointeurs différents
```

Ordre des éléments d'un ensemble

- Types définis par l'utilisateur ?
- Essai :

```
class Point2D {
    int x, y;
public:
    Point2D(int u,int v): x(u),y(v) {}
};

set<Point2D> s;
Point2D p(1,4);
Point2D q(2,3);
s.insert(p); // Pb à la compilation
s.insert(q); // Pb à la compilation
```

Qu'arrive-t-il ?

Ordre des éléments d'un ensemble

```
/usr/include/c++/4.2.1/bits/stl_function.h: In member function
'bool std::less<_Tp>::operator()(const _Tp&, const _Tp&) const
[with _Tp = Point2D]':
/usr/include/c++/4.2.1/bits/stl_tree.h:982:   instantiated
from 'std::pair<typename std::_Rb_tree<_Key, _Val,
_KeyOfValue, _Compare, _Alloc>::iterator, bool>
std::_Rb_tree<_Key, _Val, _KeyOfValue, _Compare,
_Alloc>::_M_insert_unique(const _Val&) [with _Key = Point2D,
_Val = Point2D, _KeyOfValue = std::_Identity<Point2D>,
_Compare = std::less<Point2D>, _Alloc =
std::allocator<Point2D>]'
/usr/include/c++/4.2.1/bits/stl_set.h:307:   instantiated from
'std::pair<typename std::_Rb_tree<_Key, _Key,
std::_Identity<_Key>, _Compare, typename
_Alloc::rebind<_Key>::other>::const_iterator, bool>
std::set<_Key, _Compare, _Alloc>::insert(const _Key&) [with
_Key = Point2D, _Compare = std::less<Point2D>, _Alloc =
std::allocator<Point2D>]'
cmp.cpp:28:   instantiated from here
/usr/include/c++/4.2.1/bits/stl_function.h:227: error: no
match for 'operator<' in '__x < __y'
```

Ordre des éléments d'un ensemble

```
/usr/include/c++/4.2.1/bits/stl_function.h: In member function
'bool std::less<_Tp>::operator()(const _Tp&, const _Tp&) const
[with _Tp = Point2D]':
/usr/include/c++/4.2.1/bits/stl_tree.h:982:   instantiated
from 'std::pair<typename std::_Rb_tree<_Key, _Val,
_KeyOfValue, _Compare, _Alloc>::iterator, bool>
std::_Rb_tree<_Key, _Val, _KeyOfValue, _Compare,
_Alloc>::_M_insert_unique(const _Val&) [with _Key = Point2D,
_Val = Point2D, _KeyOfValue = std::_Identity<Point2D>,
_Compare = std::less<Point2D>, _Alloc =
std::allocator<Point2D>]'
/usr/include/c++/4.2.1/bits/stl_set.h:307:   instantiated from
'std::pair<typename std::_Rb_tree<_Key, _Key,
std::_Identity<_Key>, _Compare, typename
_Alloc::rebind<_Key>::other>::const_iterator, bool>
std::set<_Key, _Compare, _Alloc>::insert(const _Key&) [with
_Key = Point2D, _Compare = std::less<Point2D>, _Alloc =
std::allocator<Point2D>]'
cmp.cpp:28:   instantiated from here
/usr/include/c++/4.2.1/bits/stl_function.h:227: error: no
match for 'operator<' in '__x < __y'
```

Info pertinente souvent à la fin

Ordonner les éléments d'un ensemble : Surcharge d'opérateur

Pas d'ordre par défaut → il faut en définir un explicitement

```
class Point2D {
    int x, y;
public:
    Point2D(int u,int v): x(u), y(v) {}
    bool operator<(const Point2D &pt) const {
        return x < pt.x ||
              (x == pt.x && y < pt.y);
    } //Ordre lexicographique sur coordonnées : x, puis y
};

set<Point2D> s;
Point2D p(1,4);
Point2D q(2,3);
cout << (p < q); // Affiche 1
s.insert(p);     // OK
s.insert(q);     // OK
```

Ordonner les éléments d'un ensemble : Surcharge d'opérateur

```
class Point2D {
    int x, y;
public:
    Point2D(int u,int v): x(u), y(v) {}
    bool operator<(const Point2D &pt) const {
        return x < pt.x ||
               (x == pt.x && y < pt.y);
    }
};

set<Point2D> s;
Point2D p(1,4);
Point2D q(2,3);
cout << (p < q); // Affiche 1
s.insert(p);     // OK
s.insert(q);     // OK
```

???

???

Ordonner les éléments d'un ensemble : Surcharge d'opérateur

ne modifie pas l'argument

ne modifie pas les variables de classe

```
class Point2D {
    int x, y;
public:
    Point2D(int u,int v): x(u), y(v) {}
    bool operator<(const Point2D &pt) const {
        return x < pt.x ||
               (x == pt.x && y < pt.y);
    }
};

set<Point2D> s;
Point2D p(1,4);
Point2D q(2,3);
cout << (p < q); // Affiche 1
s.insert(p);     // OK
s.insert(q);     // OK
```

vérification automatique à la compilation

Ordonner les éléments d'un ensemble : Surcharge d'opérateur

ne modifie pas l'argument

ne modifie pas les variables de classe

```
class Point2D {
    int x, y;
public:
    Point2D(int u,int v): x(u), y(v) {}
    bool operator<(const Point2D &pt) const {
        return x < pt.x ||
              (x == pt.x && y < pt.y);
    }
};

set<Point2D> s;
Point2D p(1,4);
Point2D q(2,3);
cout << (p < q); // Affiche 1
s.insert(p);     // OK
s.insert(q);     // OK
```

vérification automatique à la compilation

☛ deuxième **const obligatoire** pour définir <, sinon ne compile pas

Ordonner les éléments d'un ensemble : Surcharge d'opérateur

```
class Point2D {
    int x, y;
public:
    Point2D(int u,int v): x(u), y(v) {}
    bool operator<(const Point2D &pt) const {
        return x < pt.x ||
               (x == pt.x && y < pt.y);
    }
};
```

ne modifie pas l'argument

ne modifie pas les variables de classe

vérification automatique à la compilation

Propriété nécessaire de « < »

p < q doit être « neutre », sans effet sur les données, car il est testé (appelé) de multiples fois par insert, find, erase..., sans contrôle

Quoi exiger d'autre ?

Ordonner les éléments d'un ensemble : Surcharge d'opérateur

```
class Point2D {
    int x, y;
public:
    Point2D(int u,int v): x(u), y(v) {}
    bool operator<(const Point2D &pt) const {
        return x < pt.x ||
               (x == pt.x && y < pt.y);
    }
};
```

ne modifie pas l'argument

ne modifie pas les variables de classe

vérification automatique à la compilation

Propriété nécessaire de « < »… mais pas toutes vérifiables !

- p < q doit être « neutre », sans effet sur les données, car il est testé (appelé) de multiples fois par insert, find, erase..., sans contrôle
- p < q doit renvoyer toujours la même valeur [si plusieurs appels]
- not(p < q) et not(q < p) ⇒ p==q [ordre strict total : élimine doublons]

Ordonner les éléments d'un ensemble : Surcharge d'opérateur

Tentative ☹

```
class Point2D {
    int x, y;
public:
    Point2D(int u,int v) {x=u; y=v;}
    bool operator<(const Point2D &pt) const {
        return x+y < pt.x+pt.y;
// Tri suivant la distance signée à la diagonale (x,-x)
    }
// Garanties:  p<q  immuable mais !(p<q) && !(q<p) ⇒ p==q
};

set<Point2D> s;
Point2D p(1,4);
Point2D q(2,3);
s.insert(p);       // OK
s.insert(q);       // Pb (silencieux) à l'exécution
cout << s.size(); // 1 et non 2 !
```

Pourquoi ?

Ordonner les éléments d'un ensemble : Surcharge d'opérateur

```
class Point2D {
    int x, y;
public:
    Point2D(int u,int v) {x=u; y=v;}
    bool operator<(const Point2D &pt) const {
        return x+y < pt.x+pt.y;
// Tri suivant la distance signée à la diagonale (x,-x)
    }
// Garanties : p<q  immuable mais !(p<q) && !(q<p) ⇒ p==q
};

set<Point2D> s;
Point2D p(1,4);
Point2D q(2,3);
s.insert(p);        // OK
s.insert(q);        // !(p<q), !(q<p), donc p==q et q doublon
cout << s.size(); // 1 et non 2 !
```

Tentative ☹

Pourquoi ?

Ordre sur les contenants (containers)

- Ordre prédéfini sur types dans la STL : **lexicographique**
  - **pair<T1,T2>**

```
template <class T1, class T2> struct pair {
  T1 first;
  T2 second;
  pair() : first(T1()), second(T2()) {}
  bool operator<(const pair &p) const {
    return first < p.first ||
      (!(p.first < first) && second < p.y);
}}
```

n'utilise que < sur T1 et sur T2, pas ==

  - **tuple<T1,...,Tn>**
    - `tie(x1,...,xn) < tie(y1,...,yn)` C++11
  - **list<T>**, **vector<T>**, **set<T>**, etc. → ex. set< set<T>> OK
- Choix dynamique de l'ordre (= à l'exécution, ≠ à la compilation)
  - argument optionnel du constructeur : set<T> s(...,***compar***);

Type de données « ensemble » en C++, dans la STL

rappel

- Opérations générales via les itérateurs

```
for(set<T>::iterator it1=s1.begin();
    it1 != s1.end(); ++it1) {
  // Union : s2←s2 ∪ s1              : O(n1 log n2)
  s2.insert(*it1);
  // Différence : s3←s3 \ s1         : O(n1 log n3)
  s3.erase(*it1);
  // Intersection : s1←s1 ∩ s4       : O(n1 log n4)
  if(s4.find(*it1)==s4.end())
    s1.erase(it1);
} // Choisir d'itérer sur le plus petit
ensemble (hyp. n1)
```

Type de données « ensemble » en C++, dans la STL

- Opérations générales via les itérateurs

```
for(set<T>::iterator it1=s1.begin();
      it1 != s1.end(); ++it1) {
    // Union : s2 ← s2 ∪ s1              : O(n1 log n2)
    s2.insert(*it1);
    // Différence : s3 ← s3 \ s1         : O(n1 log n3)
    s3.erase(*it1);
}   // ☛ Mais opérations « destructrices » pour s2, s3
    // Comment faire des opérations non destructrices ?
```

Complexités : Union $O(n_1 \log n_2)$; Différence $O(n_1 \log n_3)$

Type de données « ensemble » en C++, dans la STL

- Opérations **non destructrices** par copie préalable

```
set<T> s(s2); // ou s3
for(set<T>::iterator it1=s1.begin();
      it1 != s1.end(); ++it1) {
    // Union : s2 ← s2 ∪ s1              : O(n1 log n2)
    s.insert(*it1);
    // Différence : s3 ← s3 \ s1          : O(n1 log n3)
    s.erase(*it1);
} // ☛ Mais coût de la copie non négligeable
```

« Algorithmes » dans la STL

- Opérations sur des **intervalles <u>triés</u>** (ex. sous-ensembles)
  - ex. **`set_intersection`** pour implémenter s3 ← s1 ∪ s2

Rappel : constructeurs C++11

```
set<int> s1 = {1, 2, 3, 4, 5};                          // s1 : 1 2 3 4 5
set<int> s2 = {2, 4, 6, 8, 10};                         // s2 : 2 4 6 8 10
vector<int> v(min(s1.size(),s2.size()));     // v : 0 0 0 0 0 [size 5]
vector<int>::iterator end =                       // fin d'intersection
    set_intersection(s1.begin(), s1.end(), s2.begin(), s2.end(),
                                  // 2 intervalles
                     v.begin());// itérateur sur zone de stockage

// v : 2 4 0 0 0  [size 5]
v.resize(end - v.begin());                 // v : 2 4  [size 2]

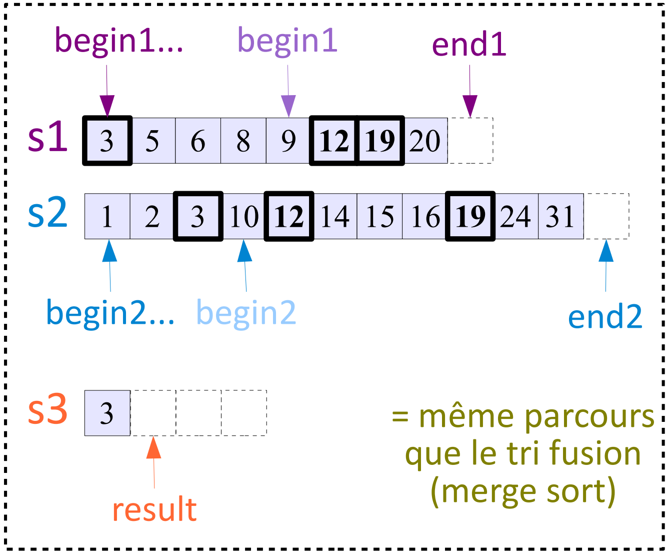
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it)
    cout << *it << ' '; cout << endl;    // affiche 2 4

set<int> s3(v.begin(), v.end());           // s3 : 2 4  [size 2]
```

« Algorithmes » dans la STL

- Opérations sur des **intervalles triés**
  - ex. **`set_intersection`** : comportement interne équiv.

```
OutputIterator set_intersection(InputIterator1 begin1, InputIterator1 end1,
        InputIterator2 begin2, InputIterator2 end2, OutputIterator result) {
    while (begin1 != end1 && begin2 != end2) {
        if (*begin1 < *begin2) ++begin1;
        else if (*begin2 < *begin1) ++begin2;
        else { *result = *begin1;   // Ou *begin2
               ++result; ++begin1; ++begin2; }
    }
    return result;
}
```

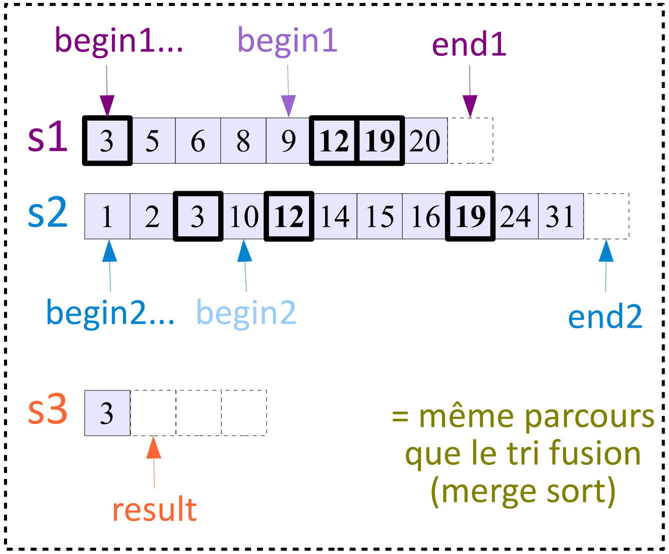


  - complexité : au plus $2(n_1+n_2)-1$ itérations, c.-à-d. $O(n_1+n_2)$
  - copies réduites au strict nécessaire

« Algorithmes » dans la STL

- Opérations sur des **intervalles triés**
  - ex. **`set_intersection`** : comportement interne équiv. ???

```
OutputIterator set_intersection(InputIterator1 begin1, InputIterator1 end1,
        InputIterator2 begin2, InputIterator2 end2, OutputIterator result) {
    while (begin1 != end1 && begin2 != end2) {
        if (*begin1 < *begin2) ++begin1;
        else if (*begin2 < *begin1) ++begin2;
        else { *result = *begin1;    // Ou *begin2
               ++result; ++begin1; ++begin2; }
    }
    return result;
}
```



  - complexité : au plus $2(n_1+n_2)-1$ itérations, c.-à-d. $O(n_1+n_2)$
  - copies réduites au strict nécessaire

Hiérarchie d'itérateurs dans la STL

- **Input iterator** : une seule passe (lecture-incrément)
  - copie : *IteratorType* it1(it2), it1=it2
  - comparaison : it1==it2, it1!=it2
  - déréférence pour lecture seule (rvalue) : *it, it->member
  - incrément : it++, ++it, *it++
  - exemples
    - pointeurs : *type**
    - itérateurs de list, vector, set... (bien que plus puissants)

Hiérarchie d'itérateurs dans la STL

- **Input iterator** : une seule passe (lecture-incrément)
  - *ItType* it1(it2), it1=it2, *it, it->m, it++, ++it, it1==it2, it1!=it2
- **Forward iterator** : + écriture possible
  - idem plus *it=v
- **Bidirectional iterator** : + retour arrière possible
  - idem plus it--, --it
- **Random-access iterator** : + accès arbitraire
  - idem plus it+n, it-n, it[n] (équivalent à *(it+n)), it+=n, it-=n, it1-it2, it1<it2, it2<=it2, it1>it2, it2>=it2
- **Output iterator** : une seule passe (écriture-incrément)
  - *ItType* it1(it2), it1=it2, *it=x, it++, ++it // sans test d'égalité

Intersection d'intervalles triés : Ex. vecteurs vers vecteur

```
#include <algorithm>          // Pour utiliser les algos de la STL

vector<int> v1 = {  5, 10, 15, 20, 25}; int n1 = v1.size(t);   // 5
vector<int> v2 = {50, 40, 30, 20, 10}; int n2 = v2.size();     // 5
vector<int> v(min(n1,n2));// 0  0  0  0  0  : init. par défault
vector<int>::iterator end;// pour la fin de l'intersection

sort(v1.begin(), v1.end());   // v1 :   5 10 15 20 25 intervalle trié
sort(v2.begin(), v2.end());   // v2 :  10 20 30 40 50 intervalle trié
// Car intersection indéfinie si [v1,v1+n1[ et [v2,v2+n2[ non triés

end = set_intersection(v1.begin(),v1.end(),v2.begin(),v2.end(),
                                             v.begin());// v:10 20 0 0 0

v.resize(end-v.begin());      // 10 20
```

Intersection d'intervalles triés : Ex. tableaux vers vecteur

```
#include <algorithm>        // Pour utiliser les algos de la STL

int t1[] = {  5, 10, 15, 20, 25}; int n1 =
sizeof(t1)/sizeof(t1[0]);  // 5
int t2[] = {50, 40, 30, 20, 10}; int n2 =
sizeof(t2)/sizeof(t2[0]);  // 5
vector<int> v(min(n1,n2)); // 0  0  0  0  0  : init. par
défault
vector<int>::iterator end; // pour la fin de l'intersection

sort(t1, t1+n1);        // t1 :  5 10 15 20 25 intervalle trié
sort(t2, t2+n2);        // t2 : 10 20 30 40 50 intervalle trié
// Car intersection indéfinie si [t1,t1+n1[ et [t2,t2+n2[ non
triés

end = set_intersection(t1, t1+n1, t2, t2+n2,
                       v.begin()); // v: 10 20 0 0 0

v.resize(end-v.begin());      // 10 20
```

Intersection d'intervalles triés : Ex. tableaux vers tableau

```
#include <algorithm>        // Pour utiliser les algos de la STL

int t1[] = {  5, 10, 15, 20, 25}; int n1 =
sizeof(t1)/sizeof(int); // 5
int t2[] = {50, 40, 30, 20, 10}; int n2 =
sizeof(t2)/sizeof(int); // 5
int t[min(n1,n2)];          // ☒ ☒ ☒ ☒ ☒: pas d'init. par défaut
int* end;                   // Pour la fin de l'intersection

sort(t1, t1+n1);            // t1 :  5 10 15 20 25 intervalle trié
sort(t2, t2+n2);            // t2 : 10 20 30 40 50 intervalle trié

end = set_intersection(t1,t1+n1,t2,t2+n2,t);// t: 10 20 ☒ ☒ ☒

for (int i=0; i<(end-t); i++) cout << t[i] << " ";   //10 20
for (int* it=t; it!=end; it++) cout << *it << " ";   //10 20
```

Algorithmes dans la STL

- Opérations sur des intervalles triés
  - `#input <algorithm>`

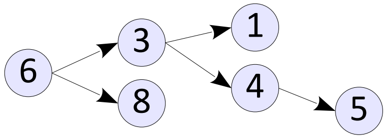
```
set_intersection(...)
set_union(...)
set_difference(...)
set_symmetric_difference(...)
```

```
OutputIterator set_xxxxx(InputIterator1 first1,
                         InputIterator1 last1,
                         InputIterator2 first2,
                         InputIterator2 last2,
                         OutputIterator result)
```

  - complexité : au pire $O(n_1+n_2)$
  - variantes avec comparateur en argument supplémentaire

Type de données « ensemble » En résumé :

- Opérations élémentaires :
  - chercher : $x \in X$?
  - insérer : $X \leftarrow X \cup \{x\}$
  - supprimer : $X \leftarrow X \setminus \{x\}$



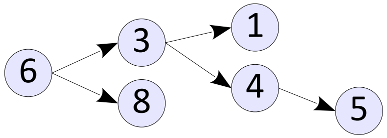
- Opérations générales : $\cup$, $\cap$, $\setminus$

Complexité ?

- Implémentations
  - avec arbre binaire (équilibré) si l'ensemble est ordonné
    - chercher, insérer et supprimer : ???
    - intersection, union, différence : ???

Type de données « ensemble »
En résumé :

- Opérations élémentaires :
  - chercher : $x \in X$?
  - insérer : $X \leftarrow X \cup \{x\}$
  - supprimer : $X \leftarrow X \setminus \{x\}$



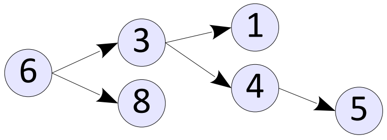
- Opérations générales : $\cup$, $\cap$, $\setminus$
- Implémentations

Complexité ?

  - avec arbre binaire (équilibré) si l'ensemble est ordonné
    - chercher, insérer et supprimer : $O(\log n)$ moyenne et pire cas
    - intersection, union, différence : $O(n_1 + n_2)$ moyenne et pire cas

Type de données « ensemble » En résumé :

- Opérations élémentaires :
  - chercher : $x \in X$?
  - insérer : $X \leftarrow X \cup \{x\}$
  - supprimer : $X \leftarrow X \setminus \{x\}$



- Opérations générales : $\cup$, $\cap$, $\setminus$
- Implémentations

Complexité ?

  - avec arbre binaire (équilibré) si l'ensemble est ordonné
    - chercher, insérer et supprimer : $O(\log n)$ moyenne et pire cas
    - intersection, union, différence : $O(n_1 + n_2)$ moyenne et pire cas

Peut-on faire plus rapide?

Peut-on faire sans ordonner ?

Bitset

- Vecteur de n bits

0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	1	0	1	0	0	0	1	0	1	1	1	0	...

  - codage compact dans un tableau de mots machine (32, 64 bits)
  - interprétables comme 0/1, false/true, présent/absent...
- Usage
  - ensemble fini d'entiers, d'objets, etc.
  - interprétation **fixée par le programmeur (ordre arbitraire)**

Codage binaire

0	1	1	0
3	2	1	0
figue	raisin	banane	noix

- $1000_b = 2^3 = 8 \Leftrightarrow \{3\} \Leftrightarrow$ {figue}
- $1011_b = 2^3 + 0 + 2^1 + 2^0 = 11 \Leftrightarrow \{3,1,0\} \Leftrightarrow$ {figue, banane, noix}
- $0000_b = \; = 0 \Leftrightarrow \{\} \Leftrightarrow$ {}

Bitset

- Vecteur de n bits

0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	1	0	1	0	0	0	1	0	1	1	1	0	...

  - codage compact dans un tableau de mots machine (32, 64 bits)
  - interprétables comme 0/1, false/true, présent/absent...

- Opérations principales (efficaces via le microprocesseur)

- set(s,i,b) : positionne le i-ème bit de s à b $\Leftrightarrow \begin{cases} s \leftarrow s \cup \{i\} \\ s \leftarrow s \setminus \{i\} \end{cases}$
- test(s,i) : retourne le i-ème bit de s $\Leftrightarrow$ $i \in s$?
- $s \wedge t$: et logique (and, &) $\Leftrightarrow$ $s \cap t$ (intersection)
- $s \vee t$: ou logique (or, |) $\Leftrightarrow$ $s \cup t$ (union)
- $\neg s$: négation (not, !) $\Leftrightarrow$ s (complément)
- shl(s,m) : décalage à gauche de m positions [shift left]
- shr(s,m) : décalage à droite de m positions [shift right]

Bitset en C++

- **Taille statique** (connue à la compilation)
  - **`bitset<size>`** `s`
  - `s.set(pos,val)` ou `s[pos]=val`
  - `s.test(pos)` ou `s[pos]`
  - `s1&s2 (and), s1|s2 (or), s1^s2 (xor), ~s (not), s<<m (shl), s>>m (shr), s1==s2 (eq), s1!=s2 (neq)`
- **Taille dynamique** (connue à l'exécution)
  - **`vector<bool>`** : version spécialisée de `vector<T>`
- Complexité de bitset (moyenne & pire cas) = celle d'un tableau
  - set/test : $O(1)$, and/or/not/etc. : $O(n)$ [avec faible constante]

Bitset en C++

- **Taille statique** (connue à la compilation)
  - **`bitset<size>`** `s`
  - `s.set(pos,val)` **ou** `s[pos]=val`
  - `s.test(pos)` **ou** `s[pos]`
  - `s1&s2 (and), s1|s2 (or), s1^s2 (xor), ~s (not), s<<m (shl), s>>m (shr), s1==s2 (eq), s1!=s2 (neq)`
- **Taille dynamique** (connue à l'exécution)
  - **`vector<bool>`** : version spécialisée de `vector<T>`
- Complexité de bitset (moyenne & pire cas) = celle d'un tableau
  - set/test : $O(1)$, and/or/not/etc. : $O(n)$ [avec faible constante]

Mieux que set<T>, mais peut-on faire encore mieux ?

Bitset en C++

- Difficile de faire plus rapide ou plus compact
  - sauf en travaillant sur la constante en facteur de $O(1)$, $O(n)$
    - compromis de taille (x 8, x 32/64), alignement des données, cache
      - ex. vector<char> ou vector<int> en stockant 0/1 pour false/true
    - stockage inhomogène selon distribution des (sous-)ensembles
- Mais limitation de représentation
  - représente $\wp(\{1,\ldots,n\})$, pas un ensemble arbitraire S
  - suppose une convention (bijection) : $\wp(\{1,\ldots,n\}) \leftrightarrow S$
- Peu adapté à des changements fréquents de n
  - bitset<n> : utilisation la plus commune, n fixe (connu à la compilation)
  - vector<bool> si taille inconnue avant exécution et **constante pendant**

Spécialisation de template (totale)

```
template <typename T>
class vector             // Version générique – doit être définie en
premier
{
  private:
    T* data;             // Blocs de mémoire alloués dynamiquement
    int capacity;        // Taille de data[]
    int size;            // Nombre d'éléments effectivement utilisés
  public:                   // On a :  size ≤ capacity
    …
```

```
template <>
class vector <bool>  // Version spécialisée – doit être définie
ensuite
{
  private:
    unsigned* data;  // Mots mémoire contigus alloués dynamiquement
    int capacity;    // Taille de data[]
    int size;        // Nombre de bits/bool effectivement utilisés
  public:               // On a :  size ≤ capacity*sizeof(unsigned)
    …
```

Spécialisation de template (partielle)

```
template <typename T, int size>
class fixedSizeVector  // Version générique – définie en premier
{
  private:
    T data[size];       // Tableau de taille statique
                        // Pas de variable capacity car capacity =
size
  public:
    …
```

```
template <int size>
class fixedSizeVector <bool,size>  // Spécialisation partielle
{
  private:
    unsigned data[size/sizeof(unsigned)]; // Bits contigus
                        // Pas de variable capacity car capacity=size
  public:
    …
```

Bilan

- Ensemble quelconque muni d'un ordre
  - opérations élémentaires (élément-ensemble) : $O(\log n)$
  - opérations globales (2 ensembles) : $O(n)$
- Ensemble en bijection « instantanée » avec $\wp(\{1,\ldots,n\})$
  - opérations élémentaires (élément-ensemble) : $O(1)$
  - opérations globales (2 ensembles) : $O(n)$ [faible constante]

Peut-on faire mieux ?
(rapide **et** sans ordre imposé)

Tableau associatif
(aussi appelé table d'association)

- Tableau associatif ≈ **fonction**

$t : K \rightarrow X$ K : ensemble de clés (keys) qcq

$k_1 \rightarrow x_1$ X : ensemble de valeurs qcq

$\dots$ $\mathrm{Dom}(t) = \{k_1,\dots,k_n\}$

$k_n \rightarrow x_n$

- Tab. asso. ≈ **ensemble** d'associations avec clés uniques

$$t = \{(k_1,x_1), \dots, (k_n,x_n)\}$$

Tableau vs Tableau associatif

n : nb d'entrées

- **Tableau**
  - représentation modifiable d'une fonction $t : \underbrace{I}_{\text{entiers contigus}} \rightarrow \underbrace{X}_{\text{ensemble quelconque}}$ avec $I = \{0,\dots,n-1\}$
  - opérations de base : lire, écrire
    - `x=t[i]` : déterminer la valeur x associée à l'**indice *i*** $\in I$
      - $x \leftarrow t(i)$
    - `t[i]=x` : associer la valeur x à l'indice i de t
      - $t \leftarrow t'$ tel que $t'(i) = x$ et $t'(j) = t(j)$ pour $j \neq i$
- **Tableau associatif** = généralisation
  - représentation modifiable d'une fonction $t : \underbrace{K}_{\text{ensembles quelconques}} \rightarrow \underbrace{X}_{\text{ensembles quelconques}}$
  - opérations de base : lire, écrire
    - `x=t[k]` : déterminer la valeur x associée à la **clé *k*** $\in K$
      - $x \leftarrow t(k)$
    - `t[k]=x` : associer la valeur x à la clé k de t
      - $t \leftarrow t'$ tel que $t'(k) = x$ et $t'(j) = t(j)$ pour $j \neq k$

Tableau associatif

- Opérations additionnelles
  - changer le **domaine de définition**
    - insérer dans t l'association (k,x) :
      $t \leftarrow t'$ tq $\mathrm{Dom}(t') = \mathrm{Dom}(t) \cup \{k\}$, $t'(k) = x$ et $t'(j) = t(j)$ pour $j \neq k$
    - supprimer de t l'association de la clé k, si elle existe :
      $t \leftarrow t'$ tq $\mathrm{Dom}(t') = \mathrm{Dom}(t) \setminus \{k\}$, $t'(j) = t(j)$ pour $j \in \mathrm{Dom}(t')$
  - interroger sur le **domaine de définition**
    - chercher si une association avec la clé k existe dans t :
      $k \in \mathrm{Dom}(t)$?
- Notations usuelles
  - `t` (tableau), `a` (associatif), `m` (map), ou nom de l'association

Tableau associatif

- Usages courants
  - tableaux creux
    - ex. répertoire téléphonique : nom ⟶ numéro
    - ex. dictionnaire : mots ⟶ définitions
  - ajout d'informations sur des objets
    - instance de classe ⟶ informations (≈ champ supplémentaire)
      - au lieu de : `obj.info = … ; f(obj.info) ;`
      - on fait : `info[obj] = … ; f(info[obj]);`
    - ex. objets externes qu'on ne peut pas étendre (sous-classer)
    - ex. information temporaire en cours d'exécution
      - mémoire libérable (à la différence d'un champ supplémentaire)

Tableau associatif en C++, dans la STL

n : nb d'entrées

- Classe et opérations : complexité (moyenne, pire cas)
  - `map<Key,T> a;` : $O(1)$
  - `a.insert(pair<Key,T>(k,x))` : $O(\log n)$
  - `a.erase(k)` : $O(\log n)$
  - `for(map<Key,T>::iterator it=a.begin(); it!=a.end(); ++it)` : $O(n)$
    `Operation(it->first,it->second);` // Oper(k,x)
  - `it=a.find(k)` : $O(\log n)$
    - ne retourne pas un `bool` mais un itérateur `it` tel que
      - si `it == a.end()`, alors `k` n'a pas été trouvée dans `a`
      - sinon l'association de `k` à `x` a été trouvée, `*it == (k,x)` et `a.erase(it)` utilisable pour la supprimer : $O(1)$, $O(\log n)$

Tableau associatif en C++, dans la STL

n : nb d'entrées

- Classe et opérations : complexité (moyenne, pire cas)
  - `map<Key,T> a;` : $O(1)$
  - `a[k]=x;` : $O(\log n)$
  - `x=a[k];` : $O(\log n)$

    ☛ crée automatiquement une association (k,z) si k $\notin$ Dom(a) où z $=$ constructeur par défaut de T, ex. a[k]++ ⇔ a[k]=0; a[k]++
  - `a.at(k)=x;` : $O(\log n)$
  - `x=a.at(k);` : $O(\log n)$

    ☛ lève une exception `out_of_range` si une association pour k n'existe pas déjà (comme pour vector<T> v.at(i))
- Comportement similaire aux tableaux/vecteurs C++

Tableau associatif en C++, dans la STL

n : nb d'entrées

they allow the direct iteration on subsets based on their order.

Maps are typically implemented as *binary search trees*.

- **map<Key,T>**
  - suppose un ordre total <u>sur les clés</u>
  - implémenté avec un arbre binaire
  - opérations élémentaires en $O(\log n)$, moyenne et pire cas
- **unordered_map<Key,T>**

  C++11 seulement

  - ne suppose pas d'ordre sur les clés
  - implémenté avec une **table de hachage** (hash table)
  - opérations élémentaire en $O(1)$ en moyenne et $O(n)$ pire cas
    - contrôle du pire cas pour qu'il soit très peu probable

Table de hachage

univers (ensemble des clés)

- Objectif = fonction modifiable $a : U \to X$ ... $a[k] = x$

quelconques

- Idée = composition d'opérations : $a[k] \approx t[h[k]]$
  - une fonction de hachage $h : U \to \{0,\ldots,m-1\}$
  - un adressage par tableau $t : \{0,\ldots,m-1\} \to X$
  - une gestion de **collisions**, pour le cas $h(k) = h(k')$ avec $k \neq k'$

  ☛ $a \approx t \circ h$

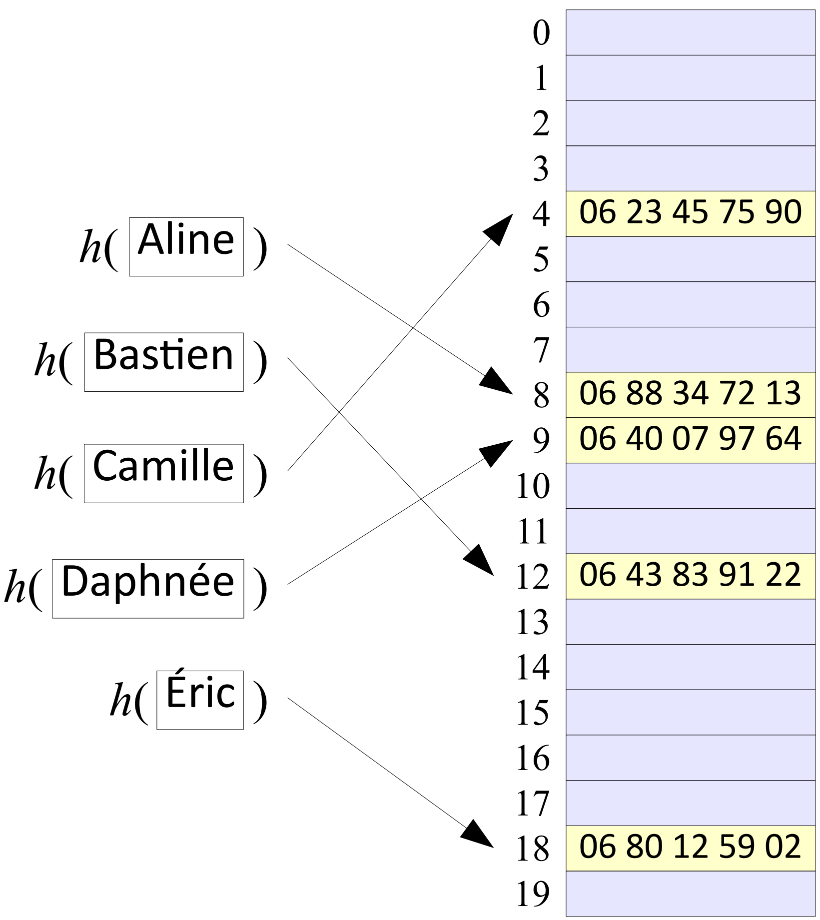
S = Dom(a) : support de la fonction
n = |S| : nombre d'associations dans a
m = |Dom(t)| : longueur du tableau t

- Contraintes
  - hachage rapide : $O(1)$ → perf. de a ≈ perf. de $t = O(1)$
  - collisions très peu fréquentes → pire cas $O(n)$ acceptable

Table de hachage sans collision

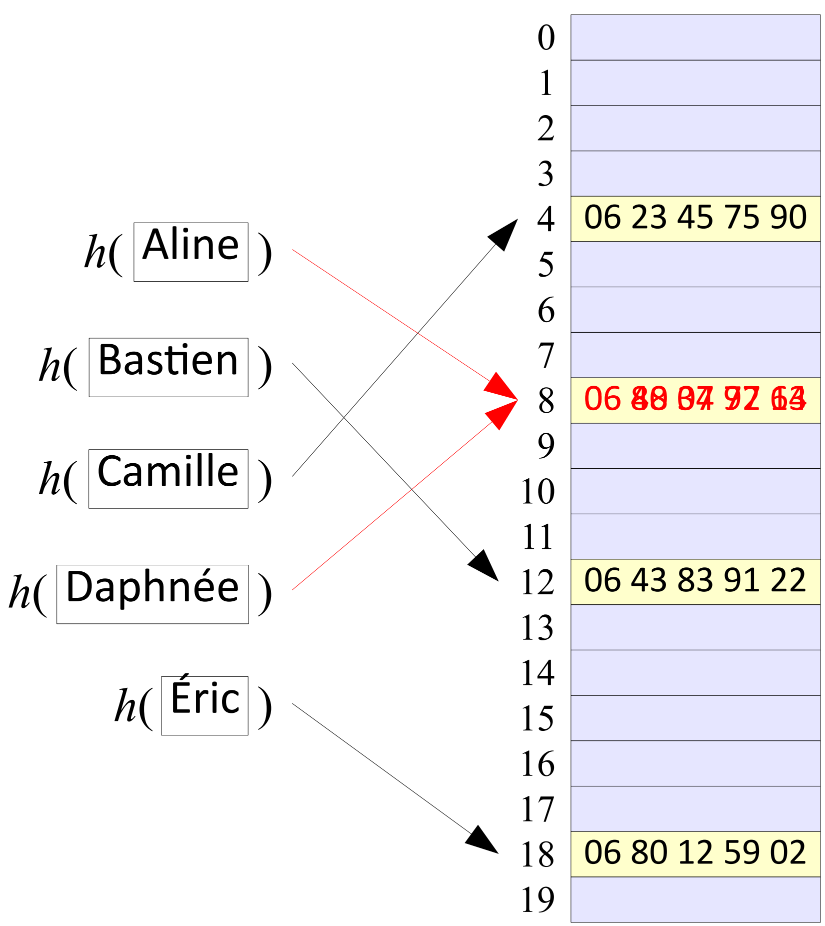
- Cas idéal = $h : U \to \{0,\ldots,m-1\}$ injection

n : nb d'entrées
m : taille du tableau



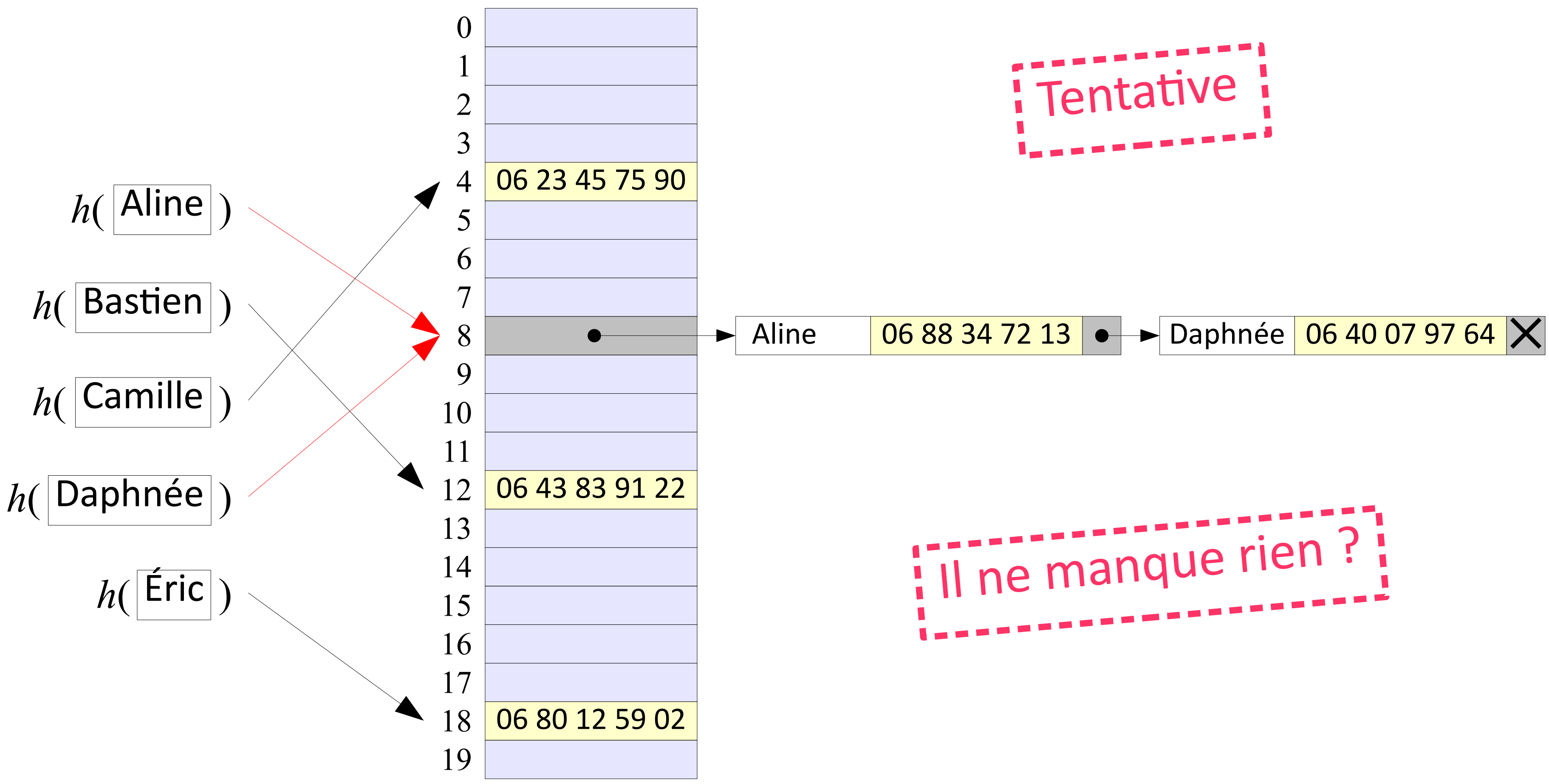
Collision

- En pratique, il existe des cas où $h(k) = h(k')$ avec $k \neq k'$



Gestion de collision par liste chaînée

- Recherche linéaire en cas de collision

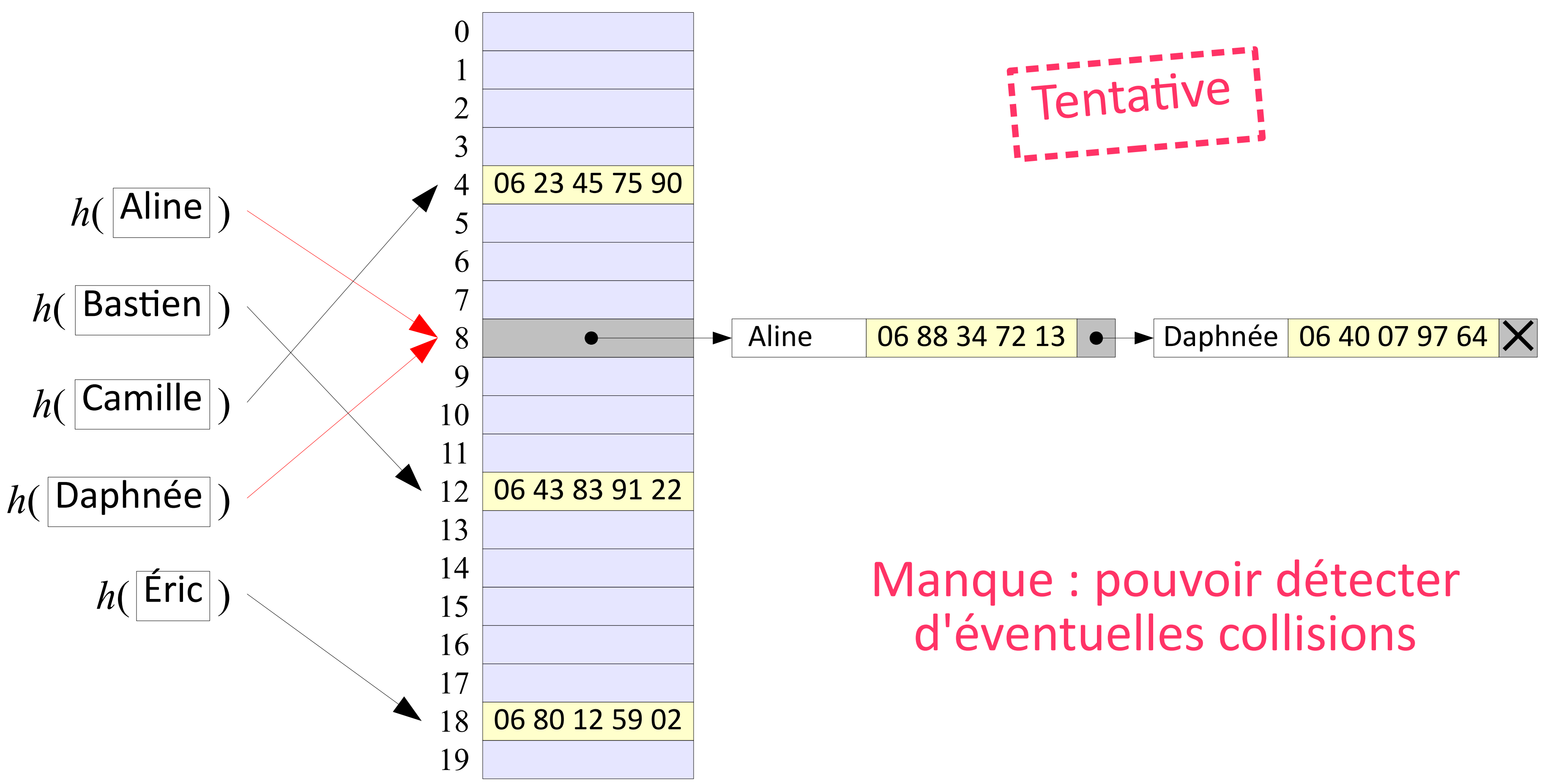


Tentative

Il ne manque rien ?

Gestion de collision par liste chaînée

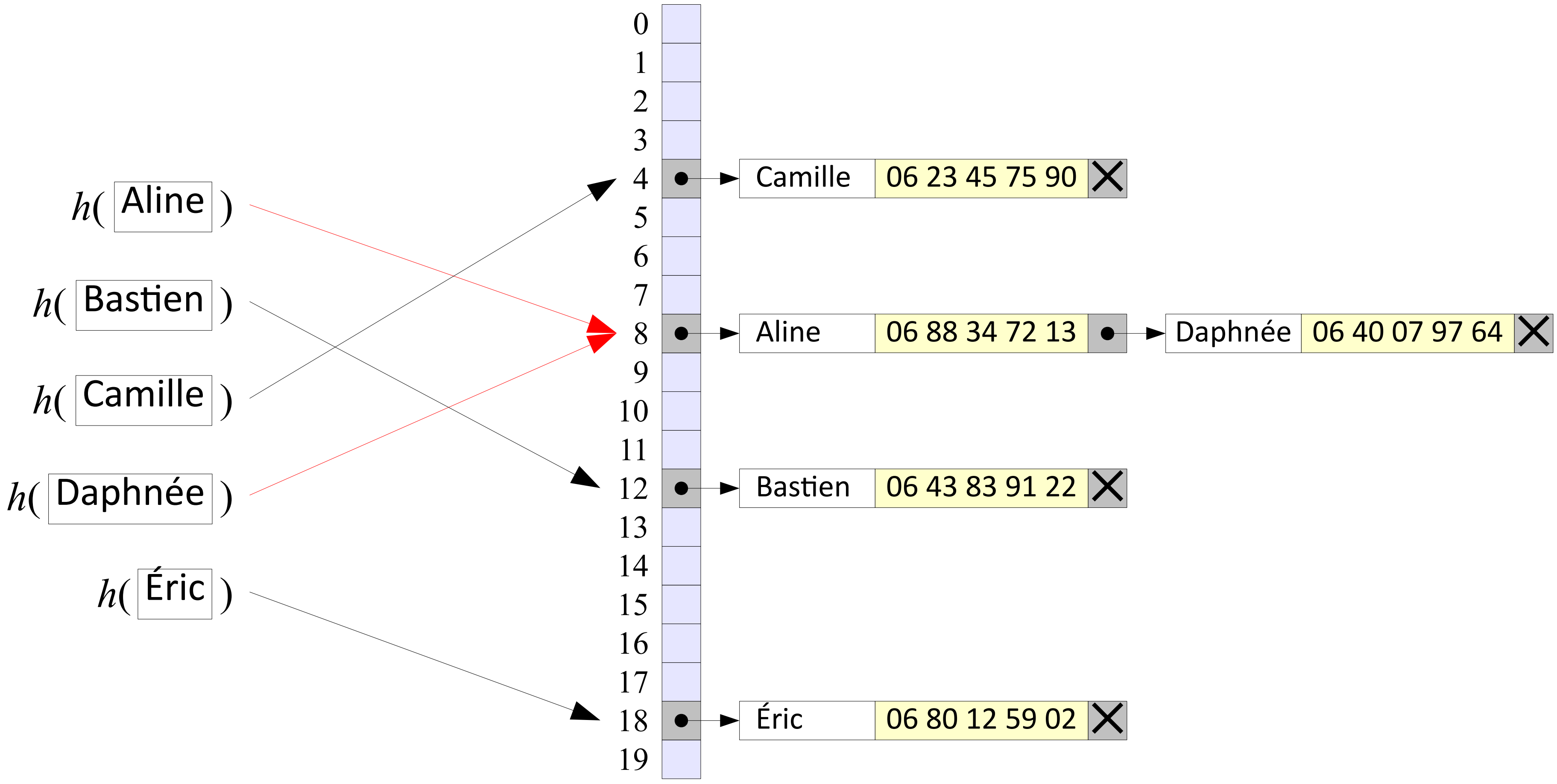
- Recherche linéaire en cas de collision



Manque : pouvoir détecter d'éventuelles collisions

Gestion de collision par liste chaînée

- Table : pointeurs sur un chaînage d'associations



Gestion de collision : Complexité et alternatives

n : nb d'entrées
m : taille du tableau

- $h : U \to \{0,\ldots,m-1\}$ utilisée sur $S \subset U$ inconnu, $|S| = n$
- Hypothèse d'uniformité : $\Pr(h(k) = i)$ indép. de k et i
- Facteur de charge d'une table de taille m : $\alpha = n / m$
  = nb moyen de collisions par entrée du tableau (= long. moy. chaînes)
  - si $n = O(m)$ (hypothèse raisonnable), $\alpha = O(m) / m = O(1)$
- Complexité si collisions avec liste chaînée
  - moyenne : $\Theta(1+\alpha)$ [$=O(1)$ si $n=O(m)$], pire cas : $O(n)$
  - ☛ le plus simple/efficace car α supposé petit
- Complexité si collisions avec arbre binaire équilibré
  - moyenne : $\Theta(1+\log \alpha)$ [$=O(1)$ si $n=O(m)$], pire cas : $O(\log n)$

Gestion de collision : Complexité et alternatives

n : nb d'entrées
m : taille du tableau

Pertinent si $\alpha \ll 1$
Taille $O(m)$ vs $O(n)$ = gâchis d'espace
→ Compromis espace-temps

- $h : U \to \{0,\dots$ … nnu, $|S| = n$
- Hypothèse d … p. de k et i
- Facteur de charge d'une table de taille m : $\alpha = n / m$
  = nb moyen de collisions par entrée du tableau (= long. moy. chaînes)
  - si $n = O(m)$ (hypothèse raisonnable), $\alpha = O(m) / m = O(1)$
- Complexité si collisions avec liste chaînée
  - moyenne : $\Theta(1+\alpha)$ [$= O(1)$ si $n = O(m)$], pire cas : $O(n)$
  - ☛ le plus simple/efficace car α supposé petit
- Complexité si collisions avec arbre binaire équilibré
  - moyenne : $\Theta(1+\log \alpha)$ [$= O(1)$ si $n = O(m)$], pire cas : $O(\log n)$

Qu'est-ce qui fait une bonne fonction de hachage ?

n : nb d'entrées
m : taille du tableau

- Fonction de hachage $h : U \to \{0,\ldots,m-1\}$
- Uniformité
  - chaque clé $k \in U$ a autant de chance d'être hachée en n'importe quel indice $i \in \{0,\ldots,m-1\}$
    $\Leftrightarrow \Pr(h(k) = i)$ indépendante de k et i
  - hypothèse fréquente, rarement possible à vérifier car distribution sur $S \subset U$ en général inconnue → approximation
  - en pratique : éviter d'être sensible à des points communs entre différentes clés plausibles (ex. mots proches)
- Calcul rapide

Hacher vers les entiers

n : nb d'entrées
m : taille du tableau

- Problème :
  - trouver une bonne fonction $h : U \longrightarrow \{0,\dots,m-1\}$
- Idée = on passe par $\mathbb{N}$: on décompose $h = h' \circ h''$
  - $h'' : U \longrightarrow \mathbb{N}$ sans biais (ou peu), ex. bijection
  - $h' : \mathbb{N} \longrightarrow \{0,\dots,m-1\}$
- Exemple : U chaînes de caractères
  - $k = (c_j)_{1 \leq j \leq l} \in U$ avec $c_j \in \{0,\dots,255\}$
  - $h''(k) = \sum_{j=1}^{l} c_j \, 256^{j-1}$ (≈ décomposition en base 256)
  - $h = h' \circ h''$

Hacher les entiers : Méthode de la division

n : nb d'entrées
m : taille du tableau

- On cherche de « bonnes » fonctions $h' : \mathbb{N} \rightarrow \{0,\ldots,m-1\}$
- **Méthode de la division :** $h'(k) = k \bmod m$

pb d'uniformité

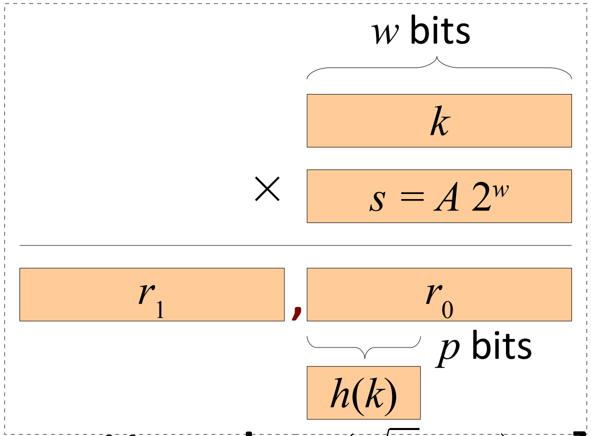
  - si $m = 2^p$ → ne considère que les p bits de poids faible → ☹
  - si $m \neq 2^p$ → tous les bits de k contribuent → ☺
    - mais pas suffisant : ex. si $k = (c_j)_{1 \leq j \leq l}$ chaîne de caract., $\mathrm{m} = 2^p - 1$ et $h(k) = \sum_j c_j \, (2^p - 1)^j$, alors $h(k)$ invariant par permutation des c_i
  - si m nombre premier → « patterns » courants dans k évités
- En pratique :
  - choisir un α acceptable, puis m nb premier proche de n / α, mais pas trop proche d'un 2^p (≠ d'un nombre de Mersenne $2^p - 1$)

Hacher les entiers : Méthode de la multiplication

n : nb d'entrées
m : taille du tableau

- On cherche de « bonnes » fonctions $h' : \mathbb{N} \to \{0,\ldots,m-1\}$
- **Méthode de la multiplication :** $h'(k) = \lfloor m\ \{A\,k\} \rfloor$
  - pour $0 < A < 1$
  - avec $\{v\} = v - \lfloor v \rfloor$ partie fractionnaire de v (aussi notée $v \bmod 1$)

    en C++ : `fmod(v,1)`
  - pas d'influence de m, choisi indépendamment
    - $m = 2^p$ est OK, et même plus efficace :
    - si $k < 2^w$ où w est la taille du mot machine
      si A est de la forme $s / 2^w$ où $0 < s < 2^w$
      alors $k\,s = k\,A\,2^w$ est sur 2 mots machine
      $h(k) = p$ premiers bits du 2e mot (⇔ frac.)



Tout ça, c'est pour réduire la constante du $O(1)$ …

  - certains A meilleurs que d'autres [recomm. Knuth $= (\sqrt{5}-1)/2$]

Hacher vers les entiers

n : nb d'entrées
m : taille du tableau

- Problème :
  - hyp. : on connaît un bon hachage $h' : \mathbb{N} \to \{0,\dots,m-1\}$
  - trouver un bon prétraitement $h'' : U \to \mathbb{N}$ tel que $h = h' \circ h''$ rapide et peu consommateur d'espace
- Hyp. générale : $\forall k \in U$ décomposable en fragments
  - $k = (w_j)_{1 \le j \le l}$ avec des w_j petits (ex. dans mot machine)
- Solution : itérer sur les fragments pour les combiner :

$s \leftarrow 0$ // Valeur initiale (peu importante)
for $j \leftarrow 1$ to l // Parcourir chaque portion de la clé k
  $s \leftarrow f(s, w_j)$ // Avec f rapide, s et w_j petits
$i \leftarrow g(s,n)$ // c.-à-d. $i = g(f(\dots f(f(0, w_1), w_2) \dots, w_l), n)$

Hachage universel

n : nb d'entrées
m : taille du tableau

- Problème :
  - adversaire malicieux qui choisit les clés exprès pour créer des collisions (ex. → déni de service)
- Solution :
  - au début de l'exécution, choix aléatoire de la fonction de hachage dans $H=(h_j)_{1\leq j\leq l}$ avec $h_j : U \longrightarrow \{0,\ldots,m-1\}$
  - par ailleurs, H est **universel** ssi $\forall k,k'\in U$
    - $|\{h\in H \text{ tq } h(k)=h(k')\}| \leq |H|/m=l/m$, ou bien
    - $\forall h\in H, \Pr(h(k)=h(k'))\leq 1/m$

  ≈ tirage aléatoire de $h(k)$ et $h(k')$
    - pas d'entrée qui conduit <u>toujours</u> au pire cas (ex. ≠ quicksort)

Exemple de hachage universel

n : nb d'entrées
m : taille du tableau

- Soient
  - p premier tq $|U| \leq p$ et $p > m$
  - $\mathbb{Z}_p = \{0,\ldots,p-1\}$ et $\mathbb{Z}_p^* = \{1,\ldots,p-1\}$
  - $h_{a,b}(k) = ((ak+b) \bmod p) \bmod m$
  - $H_{p,m} = \{h_{a,b}(k) : a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$ → $p(p-1)$ fonctions
- Théorème
  - la classe des fonctions de hachage de $H_{p,m}$ est universelle
- Preuve
  - théorie des nombres + probabilité, cf. bibliographie

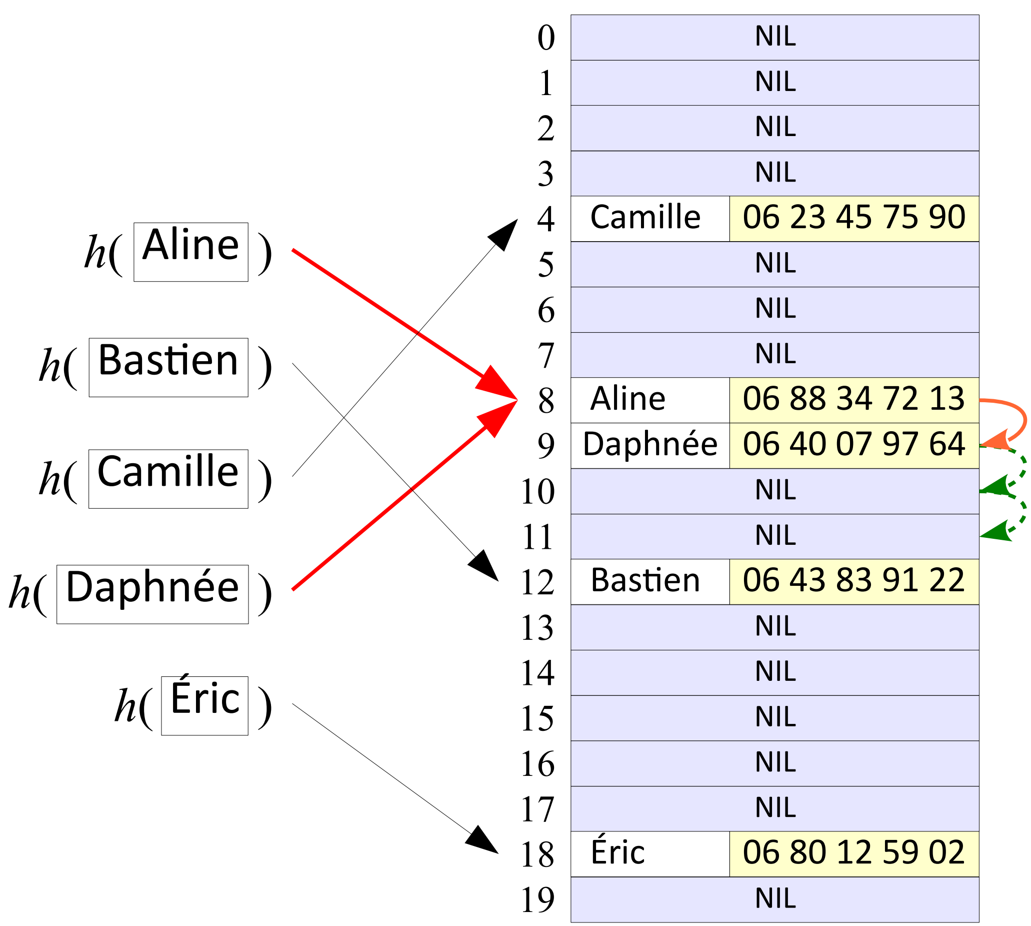
Adressage ouvert (open adressing)

- Idée : économiser de l'espace en stockant les collisions dans des cases encore libres du tableau
  - chaque case de t contient $(k,x) \in U \times X$ ou « NIL » (⇔ vide)
- Gain de place
  - pas d'allocation spécifique pour chaque association (k,x)
  - absence de pointeurs
- Principe
  - examen d'une série de cases ne dépendant que de la clé k, jusqu'à trouver une case vide (≈ gestion de collision)

Adressage ouvert

n : nb d'entrées
m : taille du tableau

- Ex. si collision, examiner la case suivante



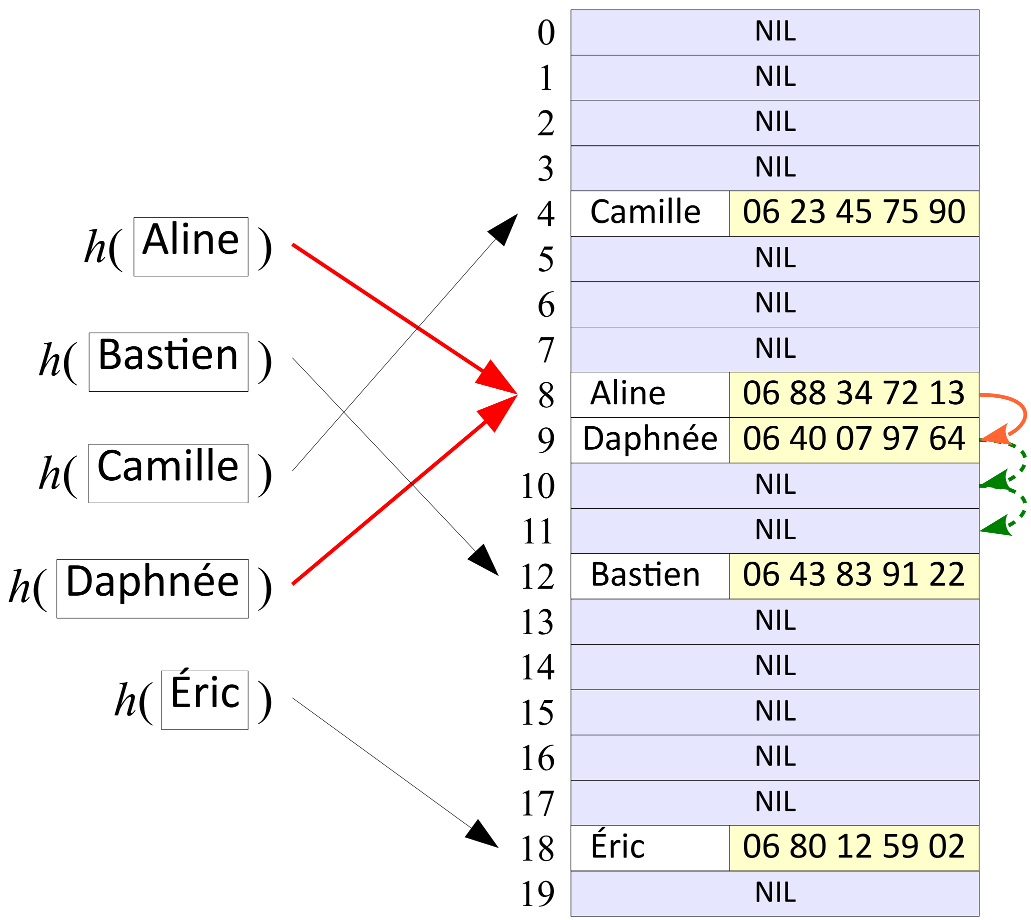
Plus rapide :
- pas d'allocation mémoire pour une nouvelle association

Plus compact :
- pas de pointeurs

Adressage ouvert

n : nb d'entrées
m : taille du tableau

- Ex. si collision, examiner la case suivante



Plus rapide :
- pas d'allocation mémoire pour une nouvelle association

Plus compact :
- pas de pointeurs

Sauf si chaque case (contenant NIL ou pas) prend de la place, comme ici : cf. taille(k,x) vs taille(ptr)

Adressage ouvert

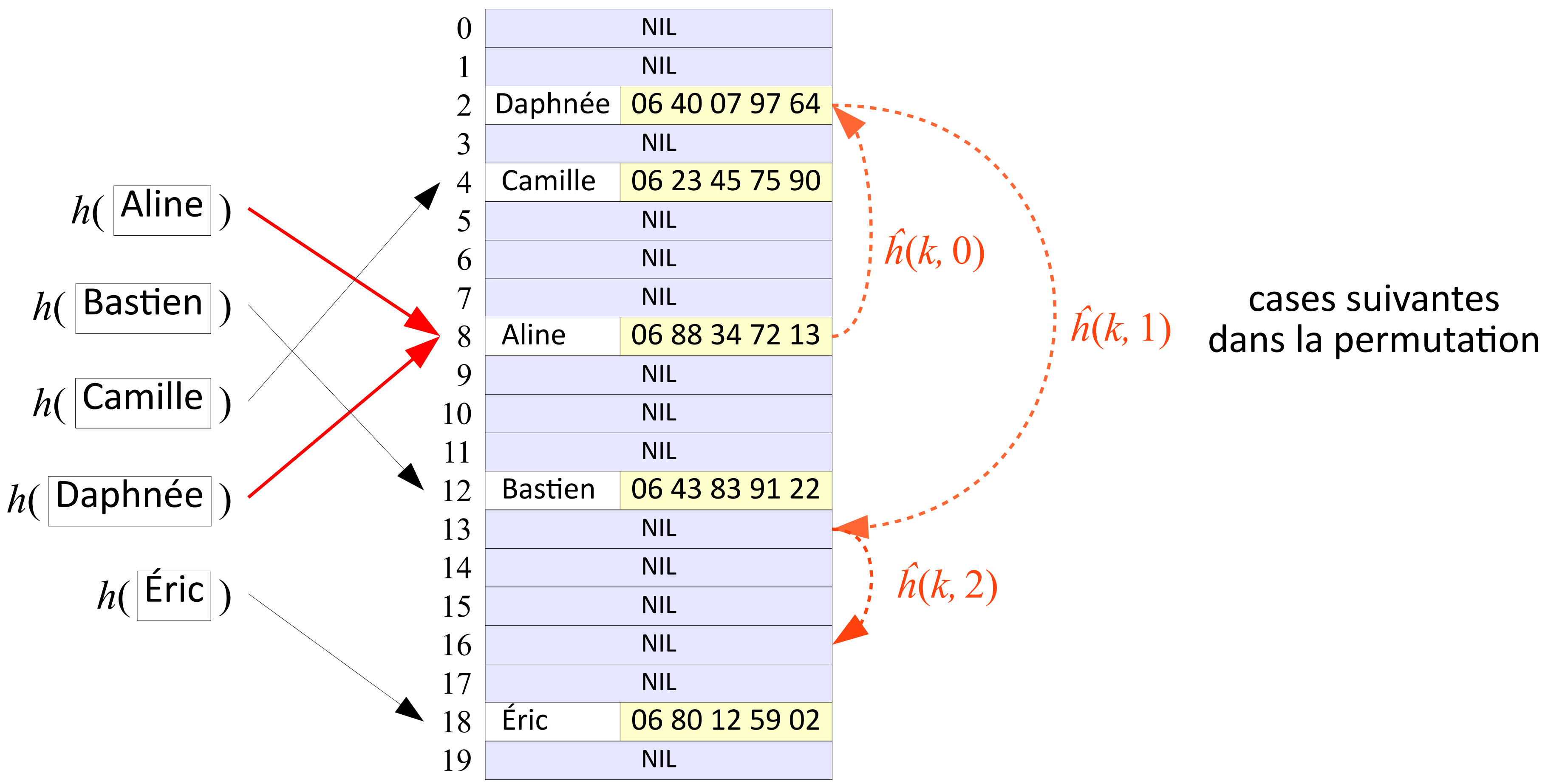
n : nb d'entrées
m : taille du tableau

- Se donner $\hat{h} : U \times \{0,\ldots,m-1\} \to \{0,\ldots,m-1\}$
  - tq $\forall k \in U$ la séquence de sondage (probe sequence) $\langle \hat{h}(k,0),\ldots,\hat{h}(k,m-1) \rangle$ est une permutation de $\langle 0,\ldots,m-1 \rangle$
  - garantit que tout t est sondé → pas de gâchis
- Insertion, recherche [→]
  - itération sur la séquence de sondage $\hat{h}(k,i)_{i\geq 0}$ si collision
- Suppression
  - pose d'une marque DELETED (≠ NIL)
  - mais complexité de recherche ne dépend pas que de $\alpha=n/m$
    - inefficace → pas utilisé en gén. s'il faut pouvoir supprimer des clés

Adressage ouvert

n : nb d'entrées
m : taille du tableau

- Si collision, examiner la case « suivante » dans la séquence de sondage



Adressage ouvert

n : nb d'entrées
m : taille du tableau

- Sondage linéaire (linear probing)
  - $\hat{h}(k,i) = (h'(k) + i) \bmod m$
  - avec fonction de hachage auxiliaire $h' : U \longrightarrow \{0,\ldots,m-1\}$
  - i et m sans diviseurs communs → exploite toute la table
  - effet d'amas possible (clustering) quand $\hat{h}(k,i) = \hat{h}(k',j)$: longue séquence de cases occupées
- Sondage quadratique (quadratic probing)
  - $\hat{h}(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
  - ex. si $m=2^p$, $c_1=c_2=½$ bon choix car parcourt toute la table
  - effet d'amas moindre mais possible

Adressage ouvert

n : nb d'entrées
m : taille du tableau
α : taux de remplissage

- Double hachage (double hashing)
  - deux fonctions de hachage auxiliaire $h_1, h_2 : U \longrightarrow \{0,\ldots,m-1\}$
  - $\hat{h}(k,i) = (h_1(k) + i\, h_2(k)) \bmod m$
  - m^2 séquences de sondage possibles (au lieu de m)
  - effet d'amas dissipé
  - complexité moyenne du nb de sondages pour $\alpha = n/m \ (< 1)$
    - pour une recherche qui échoue (clé non trouvée) : $\dfrac{1}{1-\alpha}$
      - ex. $\alpha = 50\%$ → 2 sondages | $\alpha = 90\%$ → 10 sondages
    - pour une recherche qui réussit (clé trouvée) :
      - ex. $\alpha = 50\%$ → $1{,}4$ sondages | $\alpha = 90\%$ → $2{,}6$ sondages $\dfrac{1}{\alpha}\log\left(\dfrac{1}{1-\alpha}\right)$

Notion de hachage parfait

n : nb d'entrées
m : taille du tableau

- Bonne performance en moyenne : $O(1)$
  modérée dans le (rare) pire cas : $O(n)$ ou $O(\log n)$
- Si l'ensemble des clés utilisées $S \subset U$ est fixé (statique) on peut garantir la performance dans le pire cas : $O(1)$
- Construction :
  - premier niveau de hachage ordinaire avec fonction bien choisie dans famille de fonctions de hachage universelle
  - pour les collisions, pas de liste ou arbre binaire mais un 2e niveau de hachage choisi pour garantir l'absence de collision
  - difficulté : garantir $O(n)$, cf. preuve dans la bibliographie

Tableau associatif en C++, dans la STL avec arbre binaire

Rappel (C++98)

n : nb d'entrées

- Classe et opérations : complexité (moyenne, pire cas)
  - `map<Key,T> a;` : $O(1)$
  - `a[k]=x;` : $O(\log n)$
  - `x=a[k];` : $O(\log n)$

    ☛ crée automatiquement une association (k,z) si k $\notin$ Dom(a) où z = constructeur par défaut de T, ex. a[k]++ ⇔ a[k]=0; a[k]++
  - `a.at(k)=x;` : $O(\log n)$
  - `x=a.at(k);` : $O(\log n)$

    ☛ lève une exception `out_of_range` si une association pour k n'existe pas déjà (comme pour vector<T> v.at(i))
- Comportement similaire aux tableaux/vecteurs C++

Tableau associatif en C++, dans la STL avec table de hachage

C++11 seulement

n : nb d'entrées

- Classe et opérations : complexité (moyenne, pire cas)
  - `unordered_map<Key,T> a;` : $O(1)$
  - `a[k]=x;` : $O(1+\alpha)$, $O(n)$
  - `x=a[k];` : $O(1+\alpha)$, $O(n)$

    en pratique $O(1+\alpha) = O(1)$

    ☛ crée automatiquement une association (k,z) si k ∉ Dom(a) où z = constructeur par défaut de T, ex. a[k]++ ⇔ a[k]=0; a[k]++
  - `a.at(k)=x;` : $O(1+\alpha)$, $O(n)$
  - `x=a.at(k);` : $O(1+\alpha)$, $O(n)$

    ☛ lève une exception `out_of_range` si une association pour k n'existe pas déjà (comme pour vector<T> v.at(i))
- Comportement similaire aux tableaux/vecteurs C++

Tableau associatif en C++, dans la STL avec arbre binaire

Rappel (C++98)

- Classe et opérations : complexité (moyenne, pire cas)
  - `map<Key,T> a;` : $O(1)$
  - `a.insert(pair<Key,T>(k,x))` : $O(\log n)$
  - `a.erase(k)` : $O(\log n)$
  - `for(map<Key,T>::iterator it=a.begin(); it!=a.end(); ++it)` : $O(n)$
    `Operation(it->first,it->second);`
  - `it=a.find(k)` : $O(\log n)$
    - ne retourne pas un `bool` mais un itérateur `it` tel que
      - si `it == a.end()`, alors `k` n'a pas été trouvée dans `m`
      - sinon l'association de `k` à `x` a été trouvée, `*it == (k,x)` et `a.erase(it)` utilisable pour la supprimer : $O(1)$, $O(\log n)$

Tableau associatif en C++, dans la STL avec table de hachage

C++11 seulement

- Classe et opérations : complexité (moyenne, pire cas)
  - `unordered_map<Key,T> a;` : $O(1)$
  - `a.insert(pair<Key,T>(k,x))` : $O(1+\alpha)$, $O(n)$
  - `a.erase(k)` : $O(1+\alpha)$, $O(n)$
  - `for(unordered_map<Key,T>::iterator it=a.begin(); it!=a.end(); ++it)` : $O(n)$
    `Operation(it->first,it->second);`
  - `it=a.find(k)` : $O(1+\alpha)$, $O(n)$
    - ne retourne pas un `bool` mais un itérateur `it` tel que
      - si `it == a.end()`, alors `k` n'a pas été trouvée dans `m`
      - sinon l'association de `k` à `x` a été trouvée, `*it == (k,x)` et `a.erase(it)` utilisable pour la supprimer : $O(1+\alpha)$, $O(n)$

en pratique $O(1+\alpha) = O(1)$

Pour définir un objet « hachable »

(1) Fournir une fonction d'égalité (pas un ordre)

- plus léger, moins contraint que « < »

pour test de collision dans t après calcul de $i = h(k)$

- besoin de tester les clés « $k = k'$? » pour savoir si collision
- comparer des types arbitraires, définis par le programmeur

(N.B. test de collision « $h(k) = h(k')$? » toujours OK car sur entiers)

En C++ :

- définir **`operator==`** pour le type considéré
- ou donner une fonction d'égalité en argument
  à la table de hachage lors de sa construction

Égalité entre éléments

```
class Point2D {
    int x, y;
public:
    ...
    bool operator==(const Point2D &pt) const {
        return x == pt.x && y == pt.y;
    }
};

unordered_map<Point2D,string> a;
Point2D p(1,4);
Point2D q(2,3);
cout << (p == q); // 0
a[p] = "foo";     // OK
a[q] = "bar";     // OK
cout << a[p] << a.size(); // foo2
```

C++11 seulement

Égalité entre éléments

```
class HPoint2D { // Coordonnées homogènes
    float x, y, w;
public:
    ...
    bool operator==(const HPoint2D &pt) const {
        return w*pt.x==pt.w*x && w*pt.y==pt.w*y;
    }
};
```

$(x,y,w) \equiv (cx,cy,cw)$
$\equiv (x/w,\ y/w,\ 1)$ si $w \neq 0$

```
unordered_map<HPoint2D,string> a;
HPoint2D p(1.,3.,2.);
HPoint2D q(2.,6.,4.);
cout << (p == q); // 1
a[p] = "foo";      // OK
a[q] = "bar";      // OK
cout << a[p] << a.size(); // bar1
```

C++11 seulement

Pour définir un objet « hachable »

(2) Fournir une fonction de hachage $h : U \to \{0,\ldots,m-1\}$

En C++ : définir une spécialisation de **struct hash<T>**

- namespace std {
  template <>
  struct hash<MyClass> {
  std::size_t operator()(const **MyClass**& key) const {…}
  };
  }

  size_t : grand entier non signé (taille d'un objet en octets)
  - ex. type retourné par sizeof(T)
- ou donner une fonction en argument qd la table est créée

Pour définir un objet « hachable »

- Fabriquer une valeur de type size_t
  - hacher les champs de type élémentaire : int, float, string...
  - combiner ces valeurs avec des opérations logiques : ^, <<, ...
- Briques de base pour retourner une valeur size_t
  - types élémentaires : fonctions **std::hash<*type_elem*>()(*x*)**
    - hash<float>(2*3.14)
    - hash<string>("Ponts"), ...
  - opérateurs logiques
    - $x \wedge y$: ou exclusif (xor)
    - $x << n$: décalage à gauche de n bits (shl), ...

Exemple de fonction de hachage

C++11 seulement

```
namespace std {
   using std::size_t;
   using std::hash;
   template <>
   struct hash<Point2D>{
      size_t operator()(const Point2D& p) const {
        size_t const hx (hash<float>()(p.x()));
        size_t const hy (hash<float>()(p.y()));
        return hx ^ (hy << 1);
   }}
};
unordered_map<Point2D,string> a;
Point2D p(1,4), q(2,3);
a[p] = "foo";
a[q] = "bar";
cout << a[p] << a.size(); // foo2
```

décalage hy << 1 pour éviter h(x,y)=h(y,x)

Ensembles : et aussi...

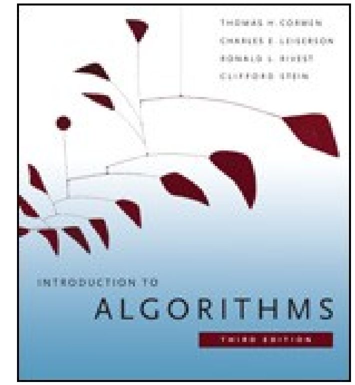
- Ensembles pas toujours nécessaires
  - un vecteur suffit souvent
    - si unicité gérée par le reste du programme
    - si seules opérations = ajouts d'éléments et itérations sans ordre
- STL
  - `unordered_set<T>` C++11 seulement
    - ensemble sans ordre (mais avec égalité), basé sur le hachage
  - `multiset<T>`
    - plusieurs occurrences possibles d'un même élément
  - `unordered_multiset<T>`

C++11 seulement

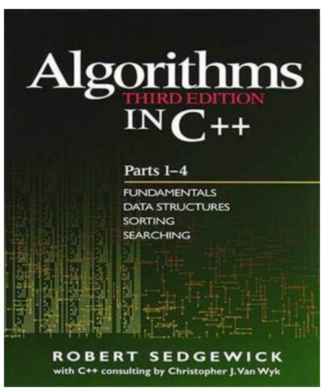
Hachage : et aussi...

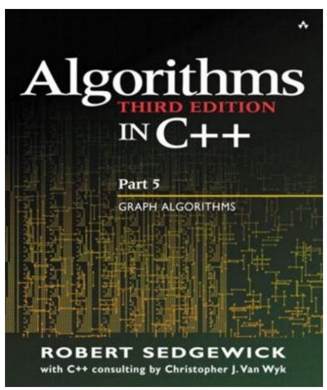
- STL
  - **`unordered_set<T>`**
  - **`unordered_multiset<T>`**
  - **`unordered_multimap<T>`** (≈ map vers multiset)
- Rehachage si le hachage initial a trop de collisions
- Utilisation en cryptographie
  - similarité avec les sommes de contrôle (checksum)
  - chute de performance pour se rapprocher de l'uniformité
    → seulement pour les applications sécuritaires (clés malicieuses)
- Domaine toujours très actif

Bibliographie



- *Introduction to Algorithms.* T. H. Cormen, Ch. E. Leiserson, R. L. Rivest, C. Stein. The MIT Press. 3rd edition, 2009.

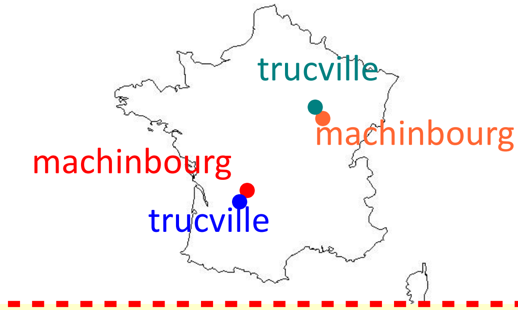




- *Algorithms in C++.* R. Sedgewick. Addison-Wesley, 3rd edition, 2002.

- Bibliothèque STL (structures de données courantes)
  - commencez toujours par là !
  - ex. http://www.cplusplus.com/reference/stl/

Exercice : isotoponymes et villes confondues



Tester d'abord sur un petit exemple artificiel

Ne pas utiliser **find** ni **sort**. Ne pas modifier les structures de données sauf si c'est indispensable pour les opérations présentées en cours

map et **set** suffisants, mais **unordered_xxx** aussi possibles

0) Récupérer l'archive avec les noms et coordonnées de 35180 villes de France métropolitaine. Les charger dans un **vector<Town>** (code fourni).
Il se trouve que certaines villes ont des **noms identiques**. De plus, du fait d'approximations, certaines villes ont aussi des **coordonnées identiques**.

1) Utiliser une table associative pour compter, construire et afficher sous forme de texte l'**histogramme des répétitions de noms de villes** [= le nb de noms de villes utilisés par 1 ville exactement, 2 villes exactement, 3 villes exactement, …]. [≈ 15 LOC]

2) Afficher l'**histogramme du nb de villes de mêmes coordonnées (Point2D)**. [≈ 15 LOC]

3) Calculer l'ensemble N des villes qui ont **une autre ville de même nom** et l'ensemble C des villes qui ont **une autre ville de mêmes coordonnées** [Étendre **Town** pour pouvoir définir **set<Town>**]. Calculer N∩C avec une complexité O(|N|+|C|) [avec fonction de la STL dédiée à ça]. Combien de villes ont cette propriété conjointe [=|N∩C|] ? [≈ 20 LOC]

4) Calculer efficacement pour combien de villes on peut se tromper en entendant parler d'**une ville A toute proche d'une ville B** ? [= nb de villes v1 tq ∃ v2, v3, v4 tq coord(v1)=coord(v2), nom(v1)=nom(v3), coord(v3)=coord(v4), nom(v2)=nom(v4)] [≈ 30 LOC]
OK si vous n'en trouvez pas, mais testez aussi sur un exemple artificiel présentant une occurrence. Ne pas répéter les solutions symétriques : (v2,v1,v4,v3), (v3,v4,v1,v2) et (v4,v3,v2,v1).

5) Comparer le temps de calcul par rapport à l'approche naïve prenant en compte toutes les villes : quel est le gain de temps ? [> × 100]