7. Tables de hachage

Clés

Une information complète se retrouve normalement à l'aide d'une clé qui l'identifie. Par exemple,

- Dans un annuaire téléphonique, connaître le nom et les prénoms d'un individu permet de retrouver son numéro de téléphone.

- Dans les sociétés modernes qui refusent le flou (et dans les ordinateurs) la clé doit identifier un individu unique, d'où par exemple, l'idée du numéro de sécurité sociale (AVS).

Annuaire téléphonique

Fred Solis, 20 rue du Milieu, ...	078 123 45 67
Anne Oroz, 4 av. de de la Harpe, ...	076 555 12 34
Chloé Rive, 5 chemin des Pelicans, ...	079 779 79 79

clé: nom et adresse (string)

valeur: numéro de téléphone (string)

Dictionnaire français -> anglais

rouge	red
bleu	blue
vert	green

clé: mot en français

valeur: mot en anglais

TDA tableau associatif

- Nous voulons un ensemble dynamique d'informations, c'est-à-dire aussi pouvoir ajouter ou supprimer un élément d'information.
- On en vient naturellement à définir un type abstrait de données, appelée table de symboles, table d'association, ou map, qui offre les opérations suivantes:
  - Ajouter une nouvelle association entre une clé et une valeur: insert / put
  - Trouver la valeur associée à une clé donnée: find / search / get
  - Retirer une clé de la table (avec la valeur associée): erase / delete

Mises en oeuvre possibles

	Tableau trié	Tableau non trié	Liste triée	Liste non triée	Arbre	Table de hachage
Insérer	O(n)	O(1)	O(n)	O(1)	$O(\log_2(n))$	O(1)
Rechercher	$O(\log_2(n))$	O(n)	O(n)	O(n)	$O(\log_2(n))$	O(1)
Supprimer	O(n)	O(n)	O(n)	O(n)	$O(\log_2(n))$	O(1)

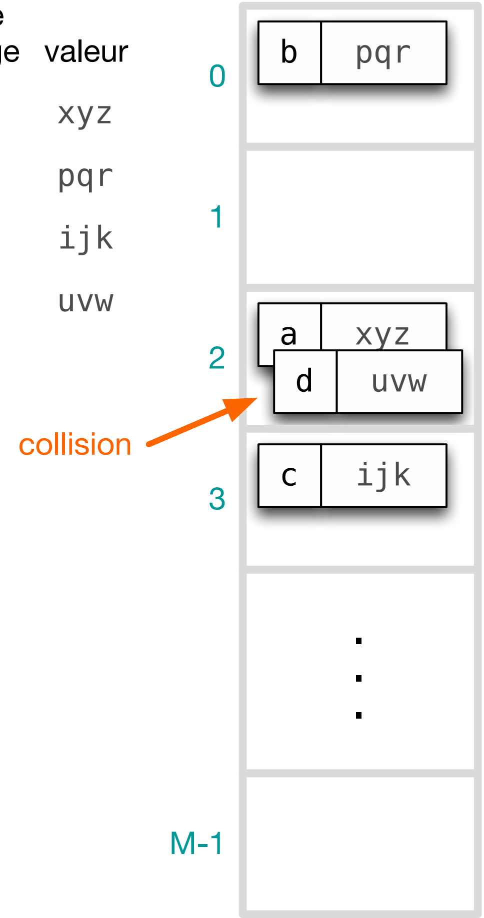
Principe du hachage

- Stocker les paires clé-valeur dans une table de hachage, i.e. un tableau accessible via des indices de 0 à $M-1$.

- La fonction de hachage $h : K \rightarrow \{0,1,2,\ldots,M-1\}$, tel que l'indice $h(k)$ - appelé adresse de hachage - nous dit où insérer / chercher l'élément dans la table

- Idéalement, des clés différentes produisent des adresses de hachage différentes : $k_1 \neq k_2 \Rightarrow h(k_1) \neq h(k_2)$.
  - Mais le nombre de clés possibles est en général plus grand que M, il est donc impossible de garantir cette condition
  - Quand plusieurs clés produisent la même adresse de hachage, on a une collision

clé	fct. de hachage	valeur
a	2	xyz
b	0	pqr
c	3	ijk
d	2	uvw



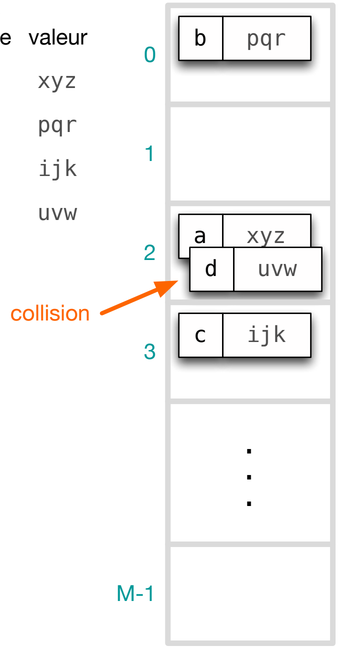
Restent à spécifier

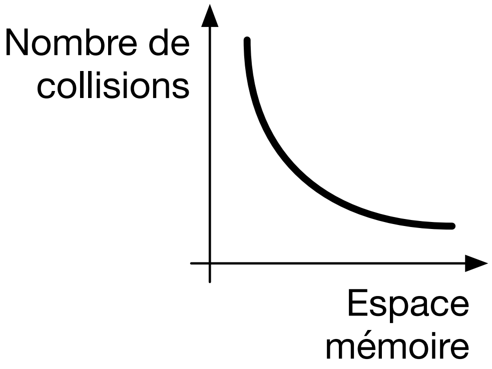
- Choix de la fonction de hachage

- Algorithme et structure de données pour gérer les collisions : chainage ou techniques d'adressage ouvert

- Choix de la taille M de la table de hachage : compromis entre nombre de collisions (donc vitesse) et mémoire utilisée

clé	fct. de hachage	valeur
a	2	xyz
b	0	pqr
c	3	ijk
d	2	uvw

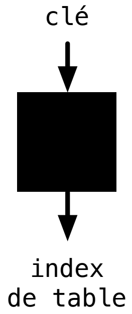




7.1. Fonctions de hachage

Fonction de hachage

- Une bonne fonction de hachage
  - est facile à calculer
  - minimise les collisions
  - distribue les adresses uniformément dans la table de hachage
  - utilise toute l'information fournie par la clé



- Par exemple, pour hacher des numéros de téléphone

  trois premiers chiffres : 021, 021, 021, 021, 079, 079, …

  trois derniers chiffres : 123, 367, 235, 974, 345, 267, …

Adressage par extraction

- Extraire des bits de la clé pour obtenir la valeur de hachage.

  - Exemple : bits 3, 10, 18 et 23 et un codage des bits par entier
    h("hello") = 110110 110000 110010 110010 110101$_b$ = 1011$_b$ =11$_d$

Facile à mettre en oeuvre

La valeur de hachage ne dépend pas de l'intégralité de la clé :

  - h("hello") = h("hello, world") = 11

Une bonne fonction de hachage doit faire intervenir tous les bits de la clé.

Adressage par division

- $h(k) = k$ modulo M , avec M premier et éloigné des puissances de 2

Bonne répartition

Multiplie les collisions

- On la combinera avec d'autres méthodes.

Adressage par multiplication

- $h(k) = \lfloor M \cdot ((k \cdot A) \mod 1) \rfloor$ avec $0 \leq A < 1$ réel

- Le choix de A doit éviter les accumulations aux extrémités de la table. Knuth a montré que la valeur $A = \dfrac{\sqrt{5}-1}{2}$ a de grande chance de bien marcher.

- Exemple avec $M = 10000$ et $A = \dfrac{\sqrt{5}-1}{2} = 0{,}61803...$

$$
\begin{aligned}
h(123456) &= \lfloor 10000 \cdot ((123456 \cdot 0.61803) \mod 1) \rfloor \\
&= \lfloor 10000 \cdot (76300.0041151 \mod 1) \rfloor \\
&= \lfloor 10000 \cdot 0.0041151 \rfloor \\
&= \lfloor 41.151 \rfloor = 41
\end{aligned}
$$

Adressage MAD

- Adressage par Multiplication, Addition et Division (MAD) via une fonction $h(k) = (a \cdot k + b) \mod M$ avec a et b entiers strictement positifs et a non multiple de M

  - Exemple avec $M = 7$, $a = 8$, et $b = 5$

    $$h(13) = (8 \cdot 13 + 5) \mod 7 = 109 \mod 7 = 4$$

  - Avec a, b, M connus, vulnérable à une attaque par choix de clés donnant toutes le même hash, ce qui entraine une complexité

- Hachage universel : choix aléatoire de la fonction de hachage indépendamment des clés pour annuler cet angle d'attaque.

  - $h(k) = ((a \cdot k + b) \mod p) \mod M$

  - En choisissant $p > k, \forall$clé k, $a \in [1,p-1]$, $b \in [1,p-1]$ avec $a \neq b$

Adressage par compression polynomiale

- On découpe la clé k en sous chaînes de longueurs de 8, 16, 32 ou 64 bits, ce qui donne des coefficients $k_0, k_1, k_2, \ldots, k_{n-1}$

- Pour une valeur de z fixe et non nulle, on calcule le polynôme $P(z) = k_0 + k_1 \cdot z + k_2 \cdot z^2 + \ldots + k_{n-1} \cdot z^{n-1}$

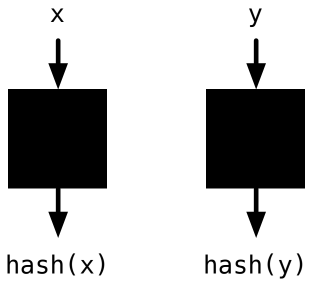
- La fonction de hachage est dès lors $h(k) = (k_0 + k_1 \cdot z + k_2 \cdot z^2 + \ldots + k_{n-1} \cdot z^{n-1}) \mod M$

- Très bonne méthode pour le type `string`. Le choix de $z = 33$ donne au plus 6 collisions sur un ensemble de 50'000 mots anglais.

Hachage en C++

La bibliothèque standard C++ définit un foncteur générique de hachage `std::hash<KeyType>` qui retourne un résultat de type `size_t` . Il est spécialisé selon les types :

- Types entiers : cast de la valeur vers `size_t`
- Autres types standards `(float, double, string, …)` : mises en oeuvre spécifiques
- Types utilisateur : à mettre en oeuvre par l'utilisateur en respectant …
  - Toujours : Si `x == y`, alors `hash(x) == hash(y)`
  - Le plus souvent possible : Si `x != y`, alors `hash(x) != hash(y)`



class template

std::hash

<functional>

`template <class T> struct hash;`

Default hash function object class

Unary function object class that defines the default hash function used by the standard library.

The functional call returns a hash value of its argument: A hash value is a value that depends solely on its argument, returning always the same value for the same argument (for a given execution of a program). The value returned shall have a small likelihood of being the same as the one returned for a different argument (with chances of collision approaching `1/numeric_limits<size_t>::max`).

Other function object types can be used as `Hash` for unordered containers provided they behave as defined above and they are at least *copy-constructible*, *destructible* function objects.

The default `hash` is a template class that is not defined for the general case. But all library implementations provide at least the following type-specific specializations:

header	types
<functional>	bool
	char
	signed char
	unsigned char
	char16_t
	char32_t
	wchar_t
	short
	unsigned short
	int
	unsigned int
	long
	unsigned long
	long long
	unsigned long long
	float
	double
	long double
	T* (for any type T)

header	types
<string>	string
	wstring
	u16string
	u32string
<memory>	unique_ptr
	shared_ptr
<vector>	vector<bool>
<bitset>	bitset
<system_error>	error_code
<typeindex>	type_index
<thread>	thread::id

Apart from being *callable* with an argument of the appropriate types, all objects of `hash` instantiations are *default-constructible*, *copy-constructible*, *copy-assignable*, *destructible* and *swappable*.

Users can provide custom specializations for this template with these same properties.

std::hash — Utilisation

```cpp
#include <functional>
#include <iostream>
#include <string>

using namespace std;

int main() {

    // création du foncteur a l'avance
    std::hash<int> int_hash;
    cout << "hash(" << 1 << ") = " << int_hash(1) << endl;

    // objet foncteur temporaire
    cout << "hash(" << 'a' << ") = " << std::hash<char>()('a') << endl;

    // hachage de plusieurs strings
    std::hash<string> str_hash;
    for( string s : { "A"s, "AAAAA"s, "AAAAB"s })
        cout << "hash(\"" << s << "\") = " << str_hash(s) << endl;
}
```

```
hash(1) = 1
hash(a) = 97
hash("A") = 3397809020744382953
hash("AAAAA") = 6458983483586025613
hash("AAAAB") = 3306170824323484213
```

HEIG-VD

Types définis par l'utilisateur

```
struct Transaction {
    std::string who;
    std::time_t when;
    long long amount;

    bool operator ==(const Transaction &o) const {
        return who == o.who && when == o.when && amount == o.amount;
    }
};

namespace std {
    template<> struct hash<Transaction> {
        std::size_t operator()(const Transaction& val) {
            using std::hash;
            size_t hashval = 17;
            hashval = 31*hashval + hash<std::string>()(val.who);
            hashval = 31*hashval + hash<std::time_t>()(val.when);
            hashval = 31*hashval + hash<long long>()(val.amount);
            return hashval;
        }
    };
}
```

Une classe définie par l'utilisateur

`operator==` est nécessaire pour distinguer les objets en cas de collision.

Mise en oeuvre d'une fonction de hachage par spécialisation de `std::hash`

Commencer avec une constante non nulle

Faire contribuer tous les attributs pertinents pour `operator==`

Utiliser un nombre premier petit

Hachage modulaire

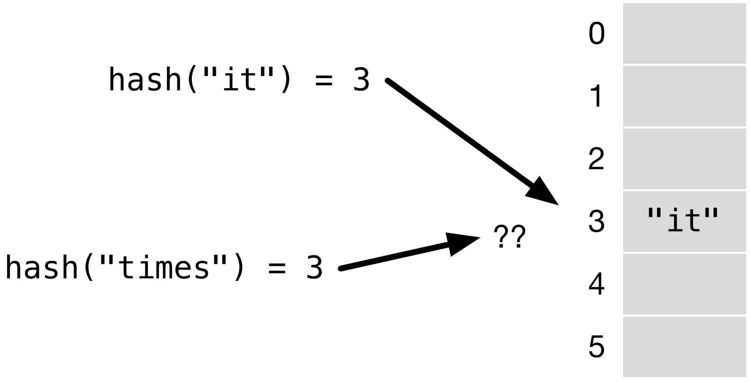
- `std::hash` retourne des valeurs de type `size_t` entre `0` et $2^{64}-1$.
- On veut des valeurs entre `0` et `M-1` pour une table de hachage de taille `M`.
- On réduit la plage de valeurs modulo `M`.

```
size_t hash(Key key) {
    return std::hash<Key>()(key) % M;
}
```

7.2. Gestion des collisions

Collisions

Deux clés $k_1 \neq k_2$ produisent le même hash $h(k_1) = h(k_2)$, et devraient dont être stockées dans la même case du tableau



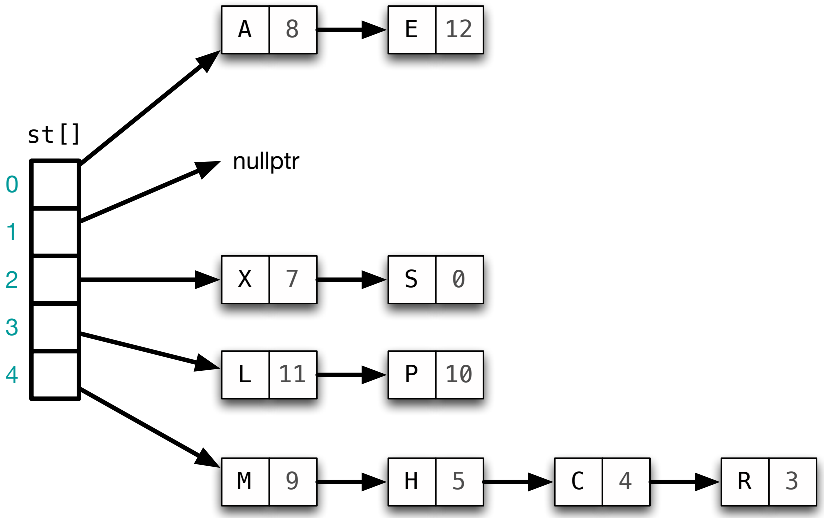
- Même avec une bonne distribution (uniforme) de la fonction de hachage, on a une haute probabilité de collision (problème des anniversaires)

- Challenge: Traiter les collisions efficacement

Résolution des collisions par chaînage

- Inventé par H.P. Luhn, IBM 1953
- Utilise un tableau de M listes simplement chaînées pour stocker N paires clé/valeur, avec $M < N$
- Hachage : transforme la clé k en un entier $0 \leq h(k) < M$
- Insertion : Insère l'élément au début de la liste d'index $h(k)$, s'il n'y est pas déjà présent.
- Recherche: Parcourt la liste d'indice $h(k)$ uniquement
- $\alpha = N/M$ est le taux d'occupation. Ici $\alpha = 2$

			fct. de hachage
put	S	0	2
put	E	1	0
put	A	2	0
put	R	3	4
put	C	4	4
put	H	5	4
put	E	6	0
put	X	7	2
put	A	8	0
put	M	9	4
put	P	10	3
put	L	11	3
put	E	12	0



Efficacité du chaînage

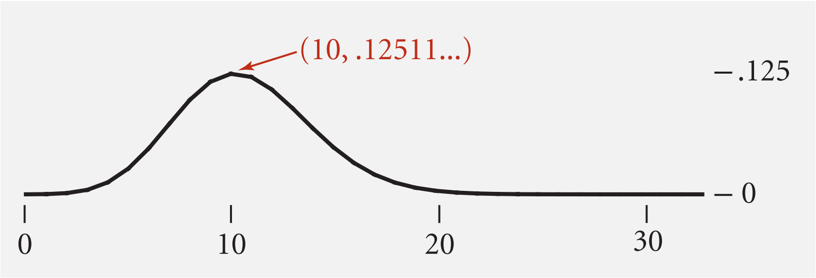
- Avec un hachage uniforme, la distribution des tailles des listes est une distribution binomiale.

- En pratique, la taille des listes - i.e. la complexité des opérations - est proche du taux d'occupation N/M

  - M trop grand ⇒ trop de listes vides

  - M trop petit ⇒ listes trop longues

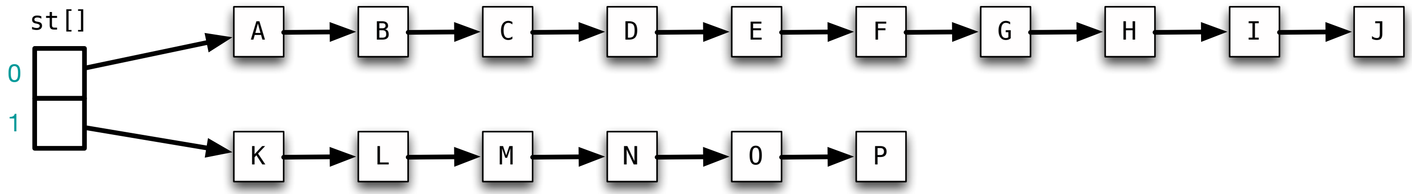
  - Choix typique: $M \approx \frac{N}{4}$ ⇒ opérations en temps constant



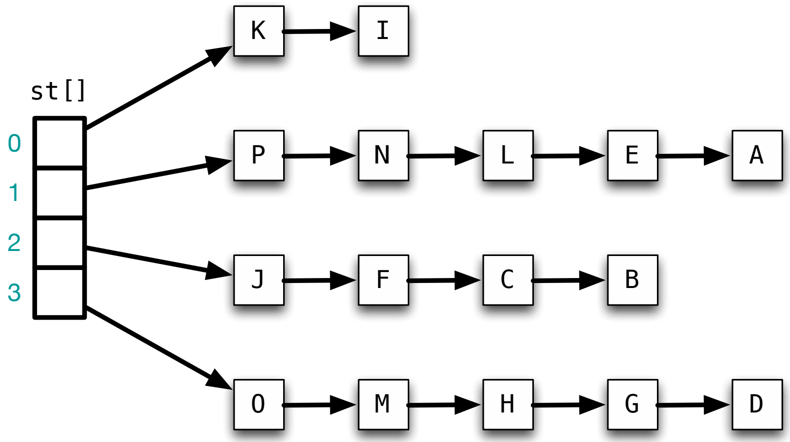
Distribution binomiale avec $N = 10^4$, $M = 10^3$, $\alpha = 10$

Redimensionner la table de hachage

Avant redimensionnement



Après redimensionnement



- Pour garder une longueur moyenne de liste N/M relativement constante, on va
  - doubler M quand $\dfrac{N}{M} \geq 8$
  - diviser M par deux quand $\dfrac{N}{M} \leq 2$
  - Attention, il faut re-hacher toutes les clés quand on change M

HE IG VD

Résolution par adressage ouvert

On stocke *N* paires clé-valeur dans une table de taille $M > N$ en utilisant les emplacements vides dans la table pour la résolution de collisions.

- Variantes:
  - Sondage linéaire *(linear probing)*
  - Double hachage *(double hashing)*
  - Hachage coalescent *(coalesced hashing)*
  - …

Sondage linéaire

Insérer une clé k : si l'emplacement à l'indice $h(k)$ est occupé, essayer $(h(k)+1) \mod M$, $(h(k)+2) \mod M$, ... jusqu'à trouver un emplacement vide

M = 16	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]																

insérer S (hash(S) = 6)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]							S									
							S									

insérer E (hash(E) = 10)

insérer A (hash(A) = 4)

insérer R (hash(R) = 14)

insérer C (hash(C) = 5)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S				E				R	
					A	C					E				R	

insérer H (hash(H) = 4)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S	H			E				R	

H

insérer X (hash(X) = 15)

insérer M (hash(M) = 1)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]		M			A	C	S	H			E				R	X

M X

insérer P (hash(P) = 14)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H			E				R	X

P

insérer L (hash(L) = 6)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

L

Sondage linéaire

- Chercher la clé k : si l'emplacement à l'indice $h(k)$ est occupé mais ne correspond pas à k, essayer $(h(k)+1) \mod M$, $(h(k)+2) \mod M$, … jusqu'à trouver soit k soit un emplacement vide

rechercher E (hash(E) = 10)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

succès

E

rechercher L (hash(L) = 6)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

succès

L

rechercher K (hash(K) = 5)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

échec

K

Analyse du sondage linéaire

- Donald Knuth montre - en 1962 - que pour une table de hachage avec M positions et $N = \alpha \cdot M$ clés, le nombre de tests à effectuer est de
  - $\approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$ si la recherche aboutit
  - $\approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$ si la recherche échoue
- Ces valeurs sont grandes quand α approche 1, mais entre 1.5 et 2.5 tests quand $\alpha < 0.5$
  - *M* trop grand ⇒ trop d'éléments de tableau vides
  - *M* trop petit ⇒ le temps de recherche explose
  - Choix typique: $\alpha = N/M \approx 1/2$

Redimensionner une table de hachage avec sondage linéaire

Pour garder le taux d'occupation $N/M \leq 1/2$

- Doubler la taille du tableau quand $N/M \geq 1/2$
- Réduire le tableau à la moitié quand $N/M \leq 1/8$
- On doit re-hacher toutes les clés quand on change la taille

Avant redimensionnement

M = 8	0	1	2	3	4	5	6	7
st[]		E	S			R	A	

Après redimensionnement

M = 16	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A		S				E				R	

Supprimer un élément

- On ne peut pas simplement enlever un élément du tableau
- Il faut ré-insérer tous les éléments qui suivent jusqu'au premier emplacement vide

Avant suppression de S

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

Après suppression de S ?

ne fonctionne pas, p. ex. si hash(H) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C		H	L		E				R	X

7.3. Hachage dans la STL

Tables de hachage dans la STL

Container class templates

Sequence containers:

array C++11	Array class (class template)	Structures linéaires (ASD1)
vector	Vector (class template)	
deque	Double ended queue (class template)	
forward_list C++11	Forward list (class template)	
list	List (class template)	

Container adaptors:

stack	LIFO stack (class template)	
queue	FIFO queue (class template)	
priority_queue	Priority queue (class template)	

Associative containers:

set	Set (class template)	Arbres (ASD1) équilibrés (ASD2)
multiset	Multiple-key set (class template)	
map	Map (class template)	
multimap	Multiple-key map (class template)	

Unordered associative containers:

unordered_set C++11	Unordered Set (class template)	Tables de hachage
unordered_multiset C++11	Unordered Multiset (class template)	
unordered_map C++11	Unordered Map (class template)	
unordered_multimap C++11	Unordered Multimap (class template)	

std::unordered_set

- TDA ensemble non trié
- Permet insertion, suppression et recherche en O(1) (amorti)
- Plus rapide qu'un `std::set` ordonné avec O(log(n))

std::unordered_set

<unordered_set>

```
template < class Key,                        // unordered_set::key_type/value_type
           class Hash = hash<Key>,           // unordered_set::hasher
           class Pred = equal_to<Key>,       // unordered_set::key_equal
           class Alloc = allocator<Key>      // unordered_set::allocator_type
           > class unordered_set;
```

Unordered Set

Unordered sets are containers that store unique elements in no particular order, and which allow for fast retrieval of individual elements based on their value.

In an `unordered_set`, the value of an element is at the same time its *key*, that identifies it uniquely. Keys are immutable, therefore, the elements in an `unordered_set` cannot be modified once in the container - they can be inserted and removed, though.

Internally, the elements in the `unordered_set` are not sorted in any particular order, but organized into *buckets* depending on their hash values to allow for fast access to individual elements directly by their *values* (with a constant average time complexity on average).

`unordered_set` containers are faster than set containers to access individual elements by their *key*, although they are generally less efficient for range iteration through a subset of their elements.

Iterators in the container are at least forward iterators.

std::unordered_map

std::unordered_map

<unordered_map>

```
template < class Key,                                    // unordered_map::key_type
           class T,                                      // unordered_map::mapped_type
           class Hash = hash<Key>,                       // unordered_map::hasher
           class Pred = equal_to<Key>,                   // unordered_map::key_equal
           class Alloc = allocator< pair<const Key,T> >  // unordered_map::allocator_type
           > class unordered_map;
```

Unordered Map

Unordered maps are associative containers that store elements formed by the combination of a *key value* and a *mapped value*, and which allows for fast retrieval of individual elements based on their keys.

In an `unordered_map`, the *key value* is generally used to uniquely identify the element, while the *mapped value* is an object with the content associated to this *key*. Types of *key* and *mapped* value may differ.

Internally, the elements in the `unordered_map` are not sorted in any particular order with respect to either their *key* or *mapped* values, but organized into *buckets* depending on their hash values to allow for fast access to individual elements directly by their *key values* (with a constant average time complexity on average).

`unordered_map` containers are faster than `map` containers to access individual elements by their *key*, although they are generally less efficient for range iteration through a subset of their elements.

Unordered maps implement the direct access operator (`operator[]`) which allows for direct access of the *mapped value* using its *key value* as argument.

Iterators in the container are at least forward iterators.

`unordered_…`

- Performance : `unordered_set / unordered_map` sont mises en oeuvre avec des tables de hachage et sont plus rapides pour l'insertion, suppression et recherche que leurs cousins `set / map`.

- Non-ordonnées : Il n'y a pas de notion d'ordre des clés.
  - Il n'y a pas de méthode pour accéder à la clé la plus petite ou la plus grande.
  - Les itérateurs parcourent les éléments du conteneur dans un ordre aléatoire.

- Exigences pour les éléments : Pour organiser les éléments les structures vont appliquer deux opérations aux éléments, qui doivent fournir des résultats valides:
  - Le class template `hash` appliqué à un élément doit fournir un code de hachage valide
  - `operator==` doit retourner `true` si deux éléments sont égaux
  - Par contre, pas besoin d'offrir `operator<` ni les autres opérateurs de comparaison