# Le hachage

# Principe général (1)

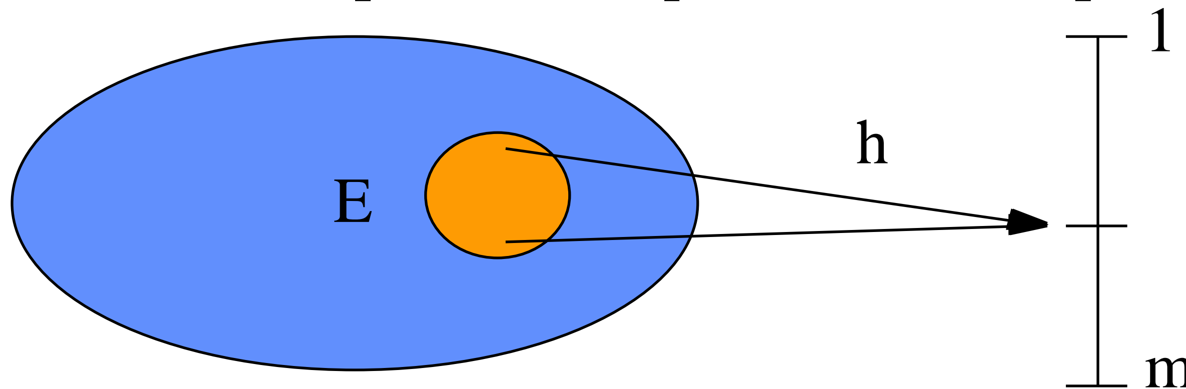


- Arbres de recherche => élément placé par rapport aux autres
- Hachage => élément placé à partir d'un calcul sur sa clé
  - complexité moyenne constante (indépendant du nombre d'éléments)
  - adaptation facile sur disque
- Remarque: cardinalité {c| c    E} très inférieur au nombre de clés différentes:
  - 500 000 personnes, clés sur 30 octets => 3 $10^{42}$ clés
  - algorithme de placement non injectif

# Principe général (2)



- h : clé [1..m]
- tableau T [1..m]
- si pour tout c, c' E, h(c) h(c'), placer l'élément de clé c en h(c) => T[h(la_clé(e))] = e
- sinon => collisions,
- h localise une partie de E qui est haché en petits morceaux



# Principe général (3)



- Il n'est pas question d'éliminer les collisions

    groupe de 23 personnes, la probabilité que deux personnes aient leur anniversaire le même jour est de 0,5

    en prenant m = 365, placer 23 personnes => 1 chance sur 2 collision

- choix judicieux de h limite les collisions
  probabilité pour que #{c | c    E et h(c) = i} = k

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| #E = 0,5m | 0,60 | 0,30 | 0,08 | 0,01 | 0,002 | 0,0002 |
| #E = m | 0,37 | 0,37 | 0,18 | 0,06 | 0,01 | 0,003 |

# Spécification du hachage

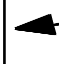


**type** ens-hach **dérive de** table [ensemble [élément, clé]]
**paramètre** élément, clé **utilise** booléen
**avec** la_clé: élément   clé
  h: clé   1 .. m

**opérations** **préconditions**

vide-h : ens-hach
ajout-h : élément × ens-hach / ens-hach   ¬(la_clé (e) $_h$ E)
supprim-h : clé × ens-hach / ens-hach   c $_h$ E
recherch-h : clé × ens-hach / élément   c $_h$ E
_ $_h$ _ : clé × ens-hach booléen

**axiomes**

borneinf(vide-h) = 1  bornesup(vide-h) = m  *initialement table de m ensembles vides*
1  i  m  ième (vide-h, i) = vide
c $_h$ E  c  ième(E, h(c))
ajout-h(e, E) = **soit** i = h(la_clé(e)); changer-ième(E, i, ajouter(e, ième(E, i)))
supprim-h(c, E) = **soit** i = h(c); changer-ième(E, i, supprimer(c, ième(E, i)))
recherch-h(c, E) = recherche(c, ième(E, h(c)))

# Fonction de hachage



- Elle doit être:

  équiprobable => probabilité que h(c) = i est 1/m

  déterministe => pour c donnée, h(c) donne toujours la même valeur

  facilement calculable => bonnes performances

- Idées générales

  extraction de bits => extraire p bits de la clé, avec $m = 2^p$

  compression de bits => découpage de la clé en q tranches de p bits puis combiner les tranches par xor et rotations

  division => h(c) = c mod m

  multiplication => h(c) = (( c * r ) mod 1) * m .

- Pas de fonction universelle => essayer sur échantillon

# Résolution des collisions



- Ajout de e, tel que h(la_clé(e)) = i et T[i] occupé
- Où?

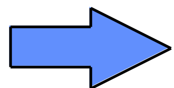
  à une autre place libre du tableau T

  dans une zone à part, zone de débordement

- Comment le relier à i?

  par pointeur => liste chaînée des collisions

  par calcul => algorithme déterminant la suite $\{i_p\}$



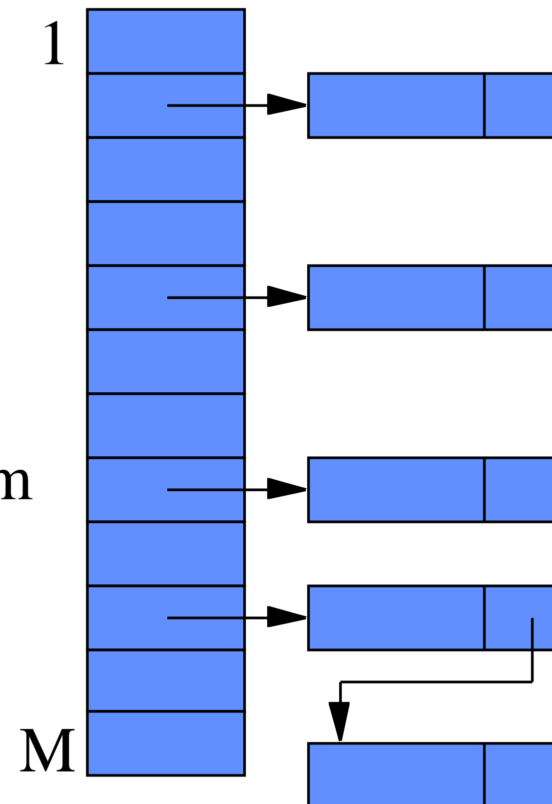
*présentation de la méthode de chaînage en zone de dépordement*

simple, efficace et adaptée au disque

# Exemple d'implantation



```
type Place; type Pt_Place is access Place;
type Place is record
        Contenu : Element;
        Suivant : Pt_Place := null;
    end record;
type Ens_Hach is array (1 .. M) of Pt_Place;
```

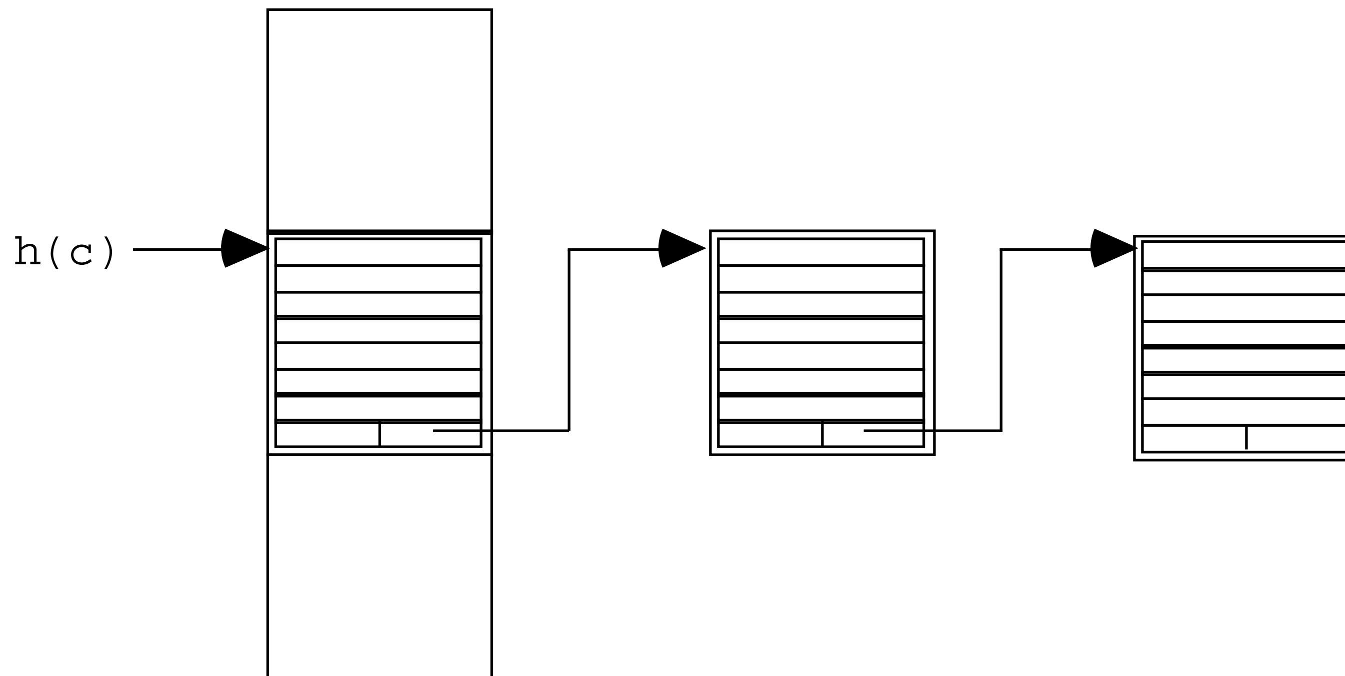



- si fonction hachage correcte, listes sensiblement de même longueur: n/m
- m grand
    - complexité en temps faible
    - perte de place dans T

# Implantation sur disque



- 1 bloc = p enregistrements de la même liste
- Chaînage des blocs d'une liste



# Implantation en Ada



```
package Hachage_IO is
    type File_Type is limited private;
private
    package L_E is new Listes_Contigues (Element);
    type T_Bloc is record
        Contenu : L_E.Liste (Taille); Suivant : Natural := 0;
    end record;
    package IO is new Ada.Direct_IO (Element_Type => T_Bloc);
    subtype T_Num_B_Hach is Integer range 1..Espace'pos(Espace'Last)
                                    - Espace'pos (Espace'First) + 2;
    type File_Type is  record
        File : IO.File_Type;
        Bloc: T_Bloc; Num_Bloc : Natural := 0;
        Indice : Positive;
        Hachage_Courant : T_Num_B_Hach := T_Num_B_Hach'Last;
    end record;
end Hachage_IO;
```

*chaînage des blocs sur disque*

*transformation espace de hachage en numéros de blocs*

# Les opérations



- Recherche => parcours de la liste des blocs
- Adjonction => parcours jusqu'à la fin (contrôle) et ajout au bout
- Suppression => recherche et tassement ou marquage
- complexité => nombre de blocs de la liste q

  au pire une seule liste => q    n/p

  en moyenne q = n/mp + 0,5

  expace inoccupé: 1/2q
- Bon conpromis: q    2,5, donne 2,5 accès disques et 20% inoccupé => m = n/2p

# Réorganisation



- Quand n augmente, la taille des listes augmente et il y a dégradation des performances
- Réorganisation par augmentation de m, avec recopie du contenu du fichier
- Réorganisation dynamique

  fonction H: clé 0..N, et $h(c) = H(c) \bmod m$

  augmentation: $m' = 2m \Rightarrow h'(c) = h(c)$ ou $h(c) + m$

  Il est possible de continuer à utiliser le fichier pendant sa réorganisation
  - marquer les listes m+1 à 2m
  - accès liste i marquée => demander éclatement liste i-m
  - éclatement liste i => démarquer liste i+m