Chapitre 2
Les ensembles dynamiques et les structures de données

Michaël Krajecki

Université de Reims Champagne-Ardenne
michael.krajecki@univ-reims.fr
http://www.univ-reims.fr/crestic

Graphes et algorithmes

Les ensembles dynamiques

- La notion d'*ensemble* est fondamentale en informatique (comme en mathématiques)
- Ils peuvent être modifiés durant l'exécution d'un algorithme, on parle alors d'ensemble *dynamique*
- Un ensemble dynamique qui supporte l'insertion, la suppression et la recherche d'une valeur est appelée un *dictionnaire*
- Certains problèmes nécessitent d'autres opérations.

Eléments d'un ensemble dynamique

- Un ensemble regroupe plusieurs éléments
- Chaque élément peut comporter plusieurs valeurs qui peuvent être regroupées :
  - dans des champs d'un enregistrement
  - dans des propriétés d'un objet
- Certains ensembles dynamiques supposent l'existance d'une valeur particulière appelée *clé*
- Chaque élément regroupe alors une clé et des *données satellites*
- Il peut être utile de disposer d'un ordre total sur les clés

Les opérations

- **Rechercher(E,k)** : renvoie l'élément de E dont la clé est k où la valeur particulière *nil* sinon
- **Insertion(E,v)** : insertion de l'élément v dans l'ensemble E
- **Suppression(E,v)** : suppression de la valeur v de l'ensemble E
- **Minimum(E)** : calcule la valeur minimale de l'ensemble E (nécessite un ordre total), idem pour **Maximum(E)**
- **Successeur(E,v)** : renvoie la valeur qui suit v dans E
- **Predecesseur(E,v)** : renvoie la valeur qui précède v dans E

Récursivité et listes

La récursivité :

- permet une écriture plus claire des algorithmes ;
- les algorithmes sont généralement plus courts ;
- il est plus facile de prouver la terminaison et la correction ;
- le calcul de la complexité est simplifié.

Récursivité et listes

La récursivité :

- permet une écriture plus claire des algorithmes ;
- les algorithmes sont généralement plus courts ;
- il est plus facile de prouver la terminaison et la correction ;
- le calcul de la complexité est simplifié.

Les listes :

- c'est un type de données naturellement récursif ;
- la gestion de la mémoire est généralement dynamique ;
- très utiles pour représenter des problèmes irréguliers.

Plan

La récursivité

Définition (Récursivité)

Une fonction (ou une procédure) R est dite récursive, si pour la définir (écrire son algorithme), au moins une référence à elle même (un appel à R) est utilisée.

Par exemple, il est possible de définir $n!$ ainsi :

$$0! = 1 \text{ et } \forall n > 0, n! = n \times (n-1)!$$

```
Fonction Fact(n : entier) : entier.
Début
  Si n=0 alors
    Fact ← 1
  sinon Fact ← n × Fact(n-1)
  Fsi
Fin.
```

Récursivité terminale

Fonction *Pgcd(a,b : entier) : entier.*
Début
  Si *a=b* ***alors***
    Pgcd $\leftarrow$ *a*
  sinon si *a>b* ***alors*** *Pgcd* $\leftarrow$ *Pgcd(a-b,a)*
  sinon *Pgcd* $\leftarrow$ *Pgcd(a,b-a)*
  Fsi
Fin.

- Vous pouvez remarquer que l'appel récursif est la dernière instruction exécutée
- On parle dans ce cas de récursivité *terminale*.

Recherche dichotomique

Voici un autre exemple de récursivité terminale.

```
Fonction Dicho(t : tableau d'entiers, binf, bsup, v : entier) : entier.
variable : p : entier
Début
  Si binf>bsup alors Dicho ← -1
  sinon      p ← (binf+bsup)/2
    Si t[p]=v alors Dicho ← p
    Sinon si t[p]>v alors Dicho ← Dicho(t, binf, p-1, v)
    Sinon Dicho ← Dicho(t, p+1, bsup, v)
    Fsi
  Fsi
Fin.
```

La suite de Fibonacci

Soit la suite de Fibonacci suivante :

$$\begin{aligned} u_0 &= 1, u_1 = 1 \\ u_n &= u_{n-1} + u_{n-2}, \forall n \geq 2. \end{aligned}$$

Fonction *Fib(n : entier) : entier.*
Début
  Si *n=0* ***alors*** *Fib* $\leftarrow$ *1*
  sinon si *n=1* ***alors*** *Fib* $\leftarrow$ *1*
  sinon *Fib* $\leftarrow$ *Fib(n-1) + Fib(n-2)*
  Fsi
Fin.

La suite de Fibonacci

Il est possible d'éviter de calculer plusieurs fois chaque élément de la suite

```
Procédure Fib2(n,u,v : entier).
variable : u1, v1 : entier
Début
  Si n=1 alors
  u ← 1
  v ← 1
  sinon
    Fib2(n-1,u1,v1)
    u=u1+v1
    v=u1
  Fsi
Fin.
```

Récursivité non terminale et croisée

- Quand, dans la définition d'une fonction récursive, l'appel récursif ne termine pas la fonction, on parle de récursivité non terminale.
- Quand une fonction A est définie à l'aide du fonction B, elle même définie à l'aide de A, on parle de récursivité croisée

Elimination de la récursivité

- Pourquoi éliminier la récursivité ?

Elimination de la récursivité

- Pourquoi éliminier la récursivité ?
  - la gestion de la récursivité par le système est lourde
  - elle peut pénaliser les performances du système

Elimination de la récursivité

- Pourquoi éliminier la récursivité ?
  - la gestion de la récursivité par le système est lourde
  - elle peut pénaliser les performances du système
- La suppression de la récursivité terminale est triviale

Elimination de la récursivité

- Pourquoi éliminier la récursivité ?
  - la gestion de la récursivité par le système est lourde
  - elle peut pénaliser les performances du système
- La suppression de la récursivité terminale est triviale
- La suppression de la récursivité non terminale est plus délicate : une pile est alors nécessaire pour mémoriser l'état de la mémoire avant l'appel récursif.

Elimination de la récursivité

- Pourquoi éliminier la récursivité ?
  - la gestion de la récursivité par le système est lourde
  - elle peut pénaliser les performances du système
- La suppression de la récursivité terminale est triviale
- La suppression de la récursivité non terminale est plus délicate : une pile est alors nécessaire pour mémoriser l'état de la mémoire avant l'appel récursif.
- La suppression de la récursivité croisée, comme pour la récursivité non terminale, est généralement délicate.

Elimination de la récursivité terminale

- Il est facile de rendre une fonction récursive terminale en une fonction non récursive.
- Il suffit de simuler la récursivité par une boucle *tant que*.
- La négation du test d'arrêt de la récursivité étant utilisée pour le test de la boucle *tant que*.
- Voici par exemple, la fonction itérative *pgcd*...

Elimination de la récursivité terminale

Fonction *Pgcd(a,b : entier) : entier.*
Début
  Tant que $a \neq b$ ***faire***
    sinon si $a > b$ ***alors*** $a \leftarrow a-b$
    sinon $b \leftarrow b-a$
  Ftant
  $Pgcd \leftarrow a$
Fin.

Plan

Les listes chainées

Définition (Liste)

Une liste est une succession de valeurs d'un même type pour lesquelles un accès direct n'est pas fréquent.

- Le type liste peut être défini récursivement :

$$L(V) = \{()\} \cup \{(v, l) \text{ où } v \in V \text{ et } l \in L(V)\}.$$

Les listes chainées

Définition (Liste)

Une liste est une succession de valeurs d'un même type pour lesquelles un accès direct n'est pas fréquent.

- Le type liste peut être défini récursivement :

$$L(V) = \{()\} \cup \{(v, l) \text{ où } v \in V \text{ et } l \in L(V)\}.$$

- Remarque : si nous acceptons cette définition, $(v_1, v_2, ..., v_n)$ devrait être notée par $(v_1, (v_2, (..., (v_n, ()))...))$.

Fonctions de base

- Les actions de bases sur les listes sont définies par les 5 fonctions suivantes :
  1. Vide : $L(V) \rightarrow \mathcal{B}$
  2. Tête : $L(V) - \{()\} \rightarrow V$
  3. Queue : $L(V) - \{()\} \rightarrow L(V)$
  4. ConsVide : $\rightarrow \{()\}$
  5. Cons : $V \times L(V) \rightarrow L(V)$

Fonctions de base

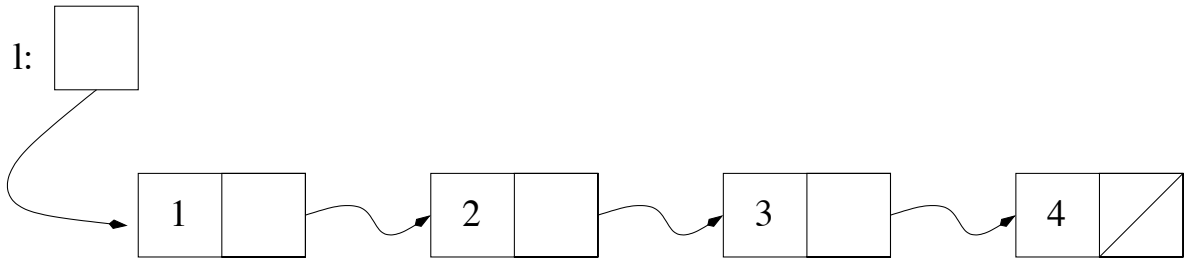
- Les actions de bases sur les listes sont définies par les 5 fonctions suivantes :
  1. Vide : $L(V) \rightarrow \mathcal{B}$
  2. Tête : $L(V) - \{()\} \rightarrow V$
  3. Queue : $L(V) - \{()\} \rightarrow L(V)$
  4. ConsVide : $\rightarrow \{()\}$
  5. Cons : $V \times L(V) \rightarrow L(V)$
- Il est possible de montrer que tous les traitements sur les listes linéaires peuvent être écrits à l'aide de ces 5 primitives.

Fonctions de base

- Les actions de bases sur les listes sont définies par les 5 fonctions suivantes :
  1. Vide : $L(V) \rightarrow \mathcal{B}$
  2. Tête : $L(V) - \{()\} \rightarrow V$
  3. Queue : $L(V) - \{()\} \rightarrow L(V)$
  4. ConsVide : $\rightarrow \{()\}$
  5. Cons : $V \times L(V) \rightarrow L(V)$
- Il est possible de montrer que tous les traitements sur les listes linéaires peuvent être écrits à l'aide de ces 5 primitives.
- Cependant, pour des problèmes d'efficacité, il est parfois nécessaire d'étendre cet ensemble de primitives.

Comparaison entre listes et tableaux

- La représentation des listes par chaînages est classique
- Elle permet :
  - une gestion dynamique de la mémoire
  - facilite l'insertion et la suppression de valeurs



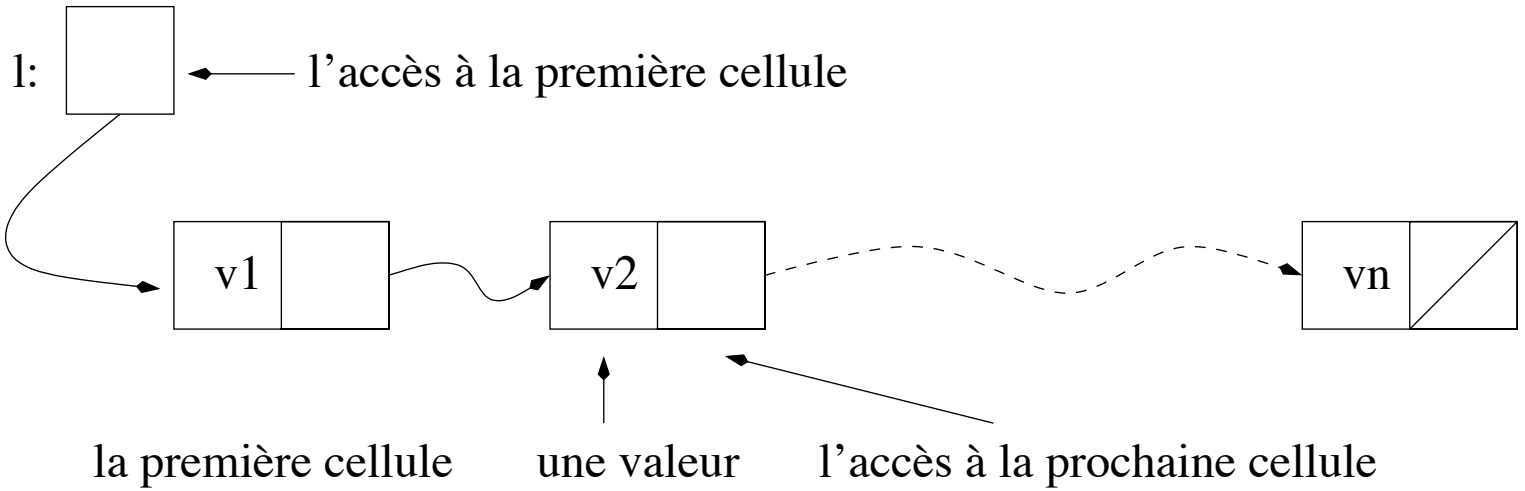
	insertion / suppression	mémoire	accès aléatoire
Tables	mauvais (décalage)	a priori mauvais	bon
Listes	rapide (chaînage)	a priori bon	mauvais

Représentation par chaînage

Afin de représenter les listes par chaînages, nous allons définir la notion de cellule. Une cellule comporte deux informations :

- une valeur de la liste ;
- un accès à la prochaine cellule de la liste ;

Une liste est alors définie par l'accès à la première cellule qui la compose.



Utilisation des structures et des pointeurs

Pour représenter une cellule en C, l'utilisation de structures est conseillée.

Le deuxième champs de la structure (l'accès à la prochaine cellule) sera défini comme un pointeur sur une cellule. Voici la définition en C d'une cellule pour une liste de caractères :

```
struct cellule {
   char valeur;
   struct cellule *suivant;
};
```

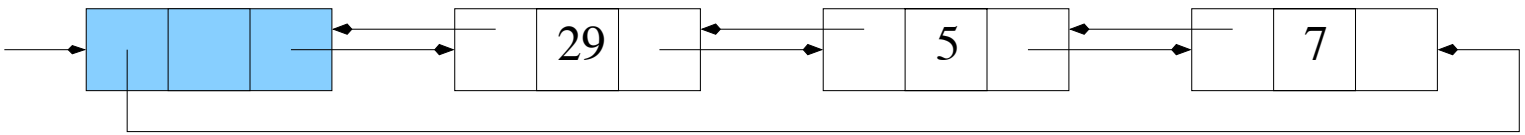
De même, une liste sera définie comme un pointeur sur une cellule...

Liste doublement chaînée

- Une liste doublement chaînée possède :
  - un lien vers son successeur dans la liste
  - un lien vers son prédécesseur
- L'élément dont le prédécesseur ne désigne aucun élément est appelé *tête*
- L'élément dont le successeur n'est pas défini est la *queue* de la liste

Liste circulaire et sentinelle

- Une liste peut être circulaire :
  - le prédécesseur de la tête désigne la queue
  - le successeur de la queue désigne la tête
- Une liste peut aussi être triée :
  - la tête est l'élément minimum
  - la tête est l'élément maximum
- L'utilisation d'une sentinelle est possible



Plan

Les piles et les files

Il est souhaitable de définir le comportement de chaque fonction de base (primitive) à l'aide d'axiomes pour lever toute ambiguité.

Définition (Pile)

Une pile (FILO : First In Last Out) est une liste dont le seul élément disponible est la tête.

Type **Pile**, *Opérations :*

$$
\begin{array}{ll}
\mathit{PileVide} & : \emptyset \rightarrow \mathit{Pile} \\
\mathit{Empiler} & : \mathit{Pile} \times \mathit{Element} \rightarrow \mathit{Pile} \\
\mathit{Depiler} & : \mathit{Pile} \rightarrow \mathit{Pile} \\
\mathit{Sommet} & : \mathit{Pile} \rightarrow \mathit{Element} \\
\mathit{EstVide} & : \mathit{Pile} \rightarrow \mathit{Booleen}
\end{array}
$$

Les piles

- *Sommet* et *Depiler* sont définies :
  - si la pile n'est pas vide
  - et satisfont les axiomes suivants pour toute pile p et tout élément e :
    - $Sommet(Empiler(p,e))=e$
    - $Depiler(Empiler(p,e))=p$

Les piles

- *Sommet* et *Depiler* sont définies :
  - si la pile n'est pas vide
  - et satisfont les axiomes suivants pour toute pile p et tout élément e :
    - $Sommet(Empiler(p,e))=e$
    - $Depiler(Empiler(p,e))=p$
- L'opération *EstVide* vérifie :
  - $EstVide(PileVide()) = Vrai$
  - $EstVide(Empiler(p,e)) = Faux$

Évaluation d'une expression postfixée

- La notation "naturelle" d'une expression arithmétique est dite infixée : quand on lit une expression de gauche à droite, on rencontre l'opérande gauche, puis l'opérateur et enfin l'opérande droite.
- Voici un exemple : 2×3.
- Il existe 2 autres notations possibles : une notation préfixée et une notation postfixée.

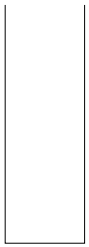
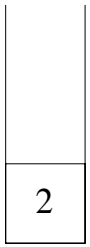
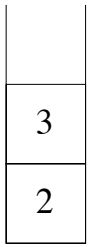
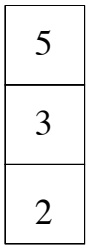
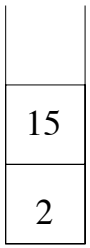
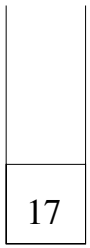
  Notation préfixée : on rencontre en premier lieu, l'opérateur, puis l'opérande gauche et enfin l'opérande droite. Par exemple : $\times$ 2 3.

  Notation postfixée : opérande gauche, opérande droite et enfin l'opérateur. Par exemple : 2 3 $\times$

Évaluation d'une expression postfixée

- L'évaluation d'une expression infixée est généralement réalisée en utilisant des arbres (voir chapitre suivant) : Les nœuds intérieurs sont des opérateurs et les feuilles des valeurs (constantes ou variables).
- L'évaluation d'une expression postfixée est plus simple. Une expression peut être évaluée en utilisant une pile :
  - Quand on rencontre une valeur, on l'empile ;
  - Quand on rencontre un opérateur, il suffit de consulter le sommet de la pile pour obtenir les deux opérandes. On empile le résultat de cette évaluation partielle.
  - Quand on rencontre le symbole fin de chaîne, le résultat est au sommet de la pile.
- Exemple pour l'expression $2 + 3 \times 5 \rightsquigarrow 2\ 3\ 5\ \times\ +$

Évaluation d'une expression postfixée

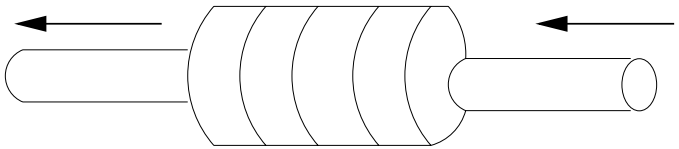
Etape 0 :	Etape 1 :	Etape 2 :
pile  expression : 2 3 5 * + #	pile  expression : 3 5 * + #	pile  expression : 5 * + #
Etape 3 :	**Etape 4 :**	**Etape 5 :**
pile  expression : * + #	pile  expression : + #	pile  expression : #

Plan

Les files

Définition (File)

Une file (FIFO : First In First Out) est une liste dont seule la tête est accesible et dans laquelle l'insertion est réalisée en queue.



Type **File**, *Opérations :*

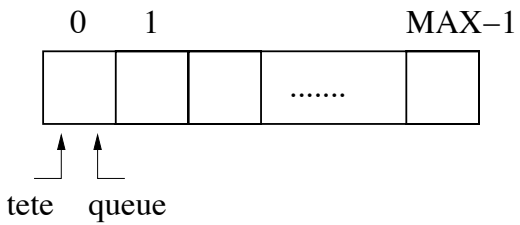
$$
\begin{array}{ll}
\mathit{FileVide} & : \emptyset \rightarrow \mathit{File} \\
\mathit{Enfiler} & : \mathit{File} \times \mathit{Element} \rightarrow \mathit{File} \\
\mathit{Defiler} & : \mathit{File} \rightarrow \mathit{File} \\
\mathit{Tete} & : \mathit{File} \rightarrow \mathit{Element} \\
\mathit{EstVide} & : \mathit{File} \rightarrow \mathit{Booleen}
\end{array}
$$

Implantation des files à l'aide de tableaux

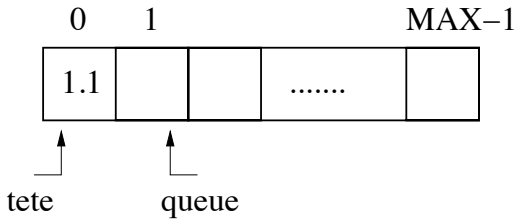
- Il faut conserver, en plus des valeurs, les indices de la tête de la file et de sa queue.
- La création d'une file consiste à initialiser les champs *tete* et *queue* de la structure avec la valeur 0
- Enfiler une valeur consiste à placer cette valeur en position *queue* du tableau. Il faut de plus incrémenter *queue* de 1 en modulo
- Une file est vide quand *tete*=*queue*
- Défiler une valeur est une opération simple : il suffit d'incrémenter la variable *tete*

Implantation des files à l'aide de tableaux

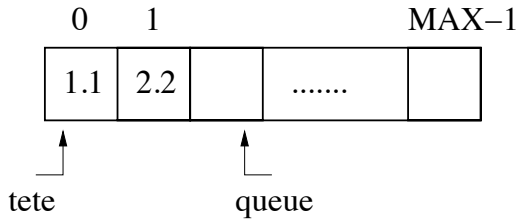
fi = FileVide();



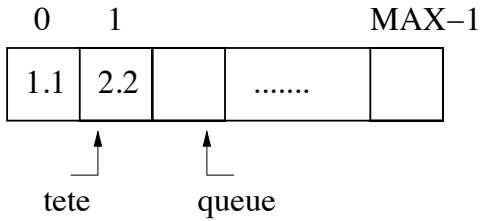
fi = Enfiler(1.1,fi);



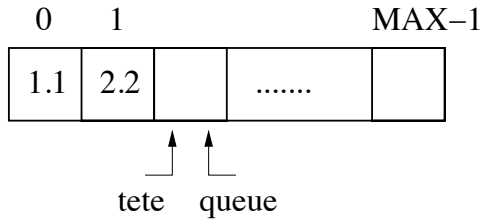
fi =Enfiler(2.2,fi);



fi = Defiler(fi);



fi = Defiler(fi);



Implantation des files à l'aide de tableaux

Attention :

1. Il n'y a pas de gestion des erreurs :
   - Que se passe-t-il en cas d'enfilement dans une file pleine ?
   - Défiler une file vide ?

2. La fonction *EstVide(f)* renvoie une réponse positive si :
   - La file est vide ;
   - mais aussi si la file est pleine...

Plan

Les tables de hachage

- De nombreuses applications en informatique nécessitent des structures de données où les opérations de dictionnaire sont possibles :
  1. Insérer.
  2. Rechercher.
  3. Supprimer.

Les tables de hachage

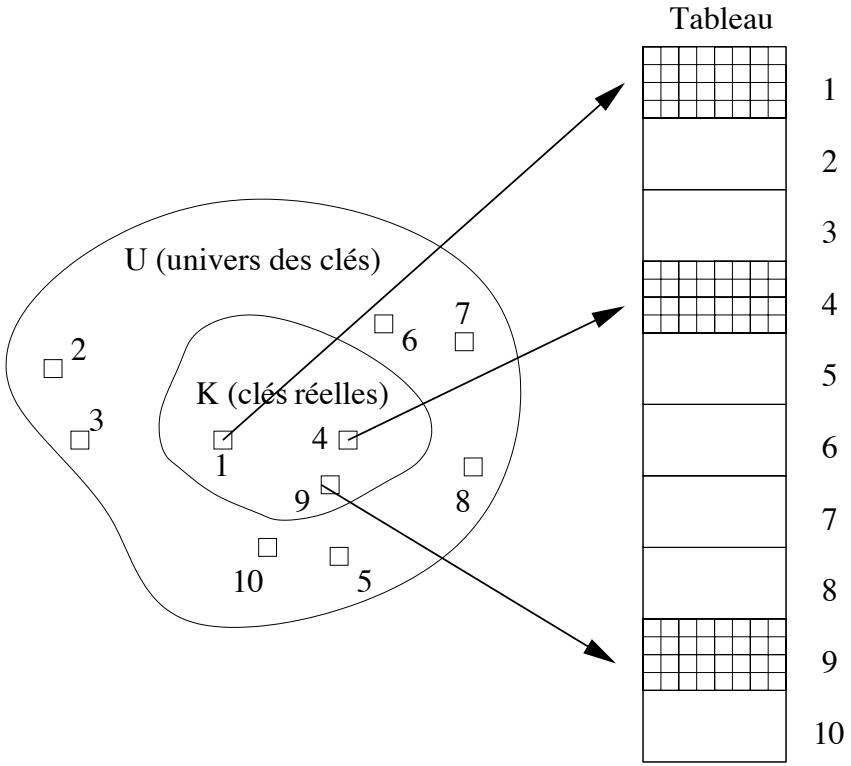
- De nombreuses applications en informatique nécessitent des structures de données où les opérations de dictionnaire sont possibles :
  1. Insérer.
  2. Rechercher.
  3. Supprimer.
- Les *tables de hachage* permettent une implantation efficace de ces 3 opérations.

Les tables de hachage

- De nombreuses applications en informatique nécessitent des structures de données où les opérations de dictionnaire sont possibles :
  1. Insérer.
  2. Rechercher.
  3. Supprimer.
- Les *tables de hachage* permettent une implantation efficace de ces 3 opérations.
- Une table de hachage est une généralisation de la notion plus simple de tableau ordinaire.

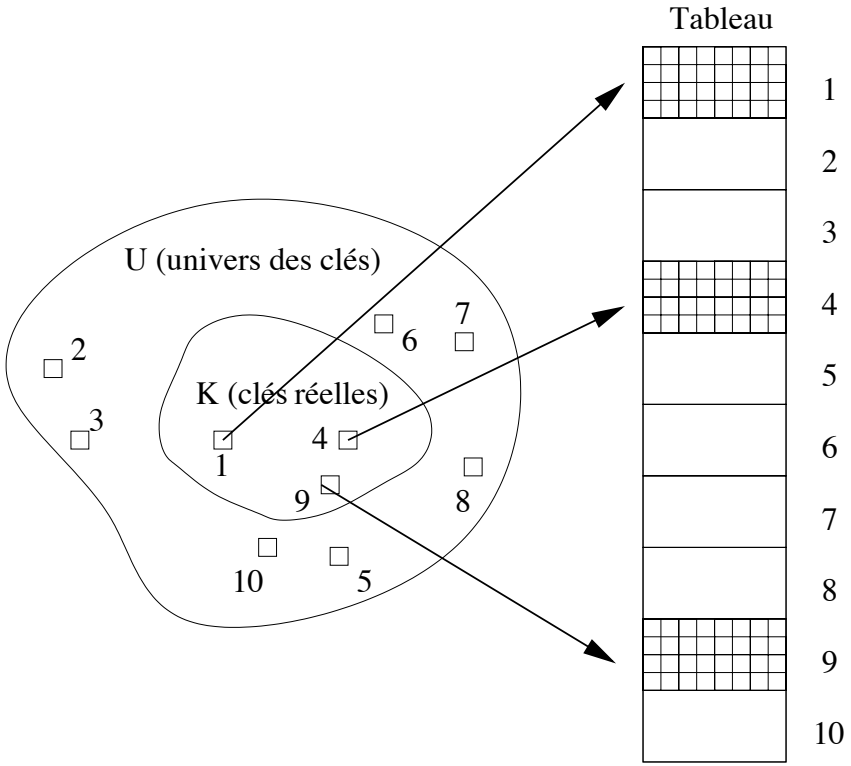
Retour sur les tableaux

- L'adressage direct permet d'atteindre efficacement tout élément d'un tableau (coût en $O(1)$)



Retour sur les tableaux

- L'adressage direct permet d'atteindre efficacement tout élément d'un tableau (coût en $O(1)$)



- mais ce n'est pas toujours possible...

Les tables de hachage

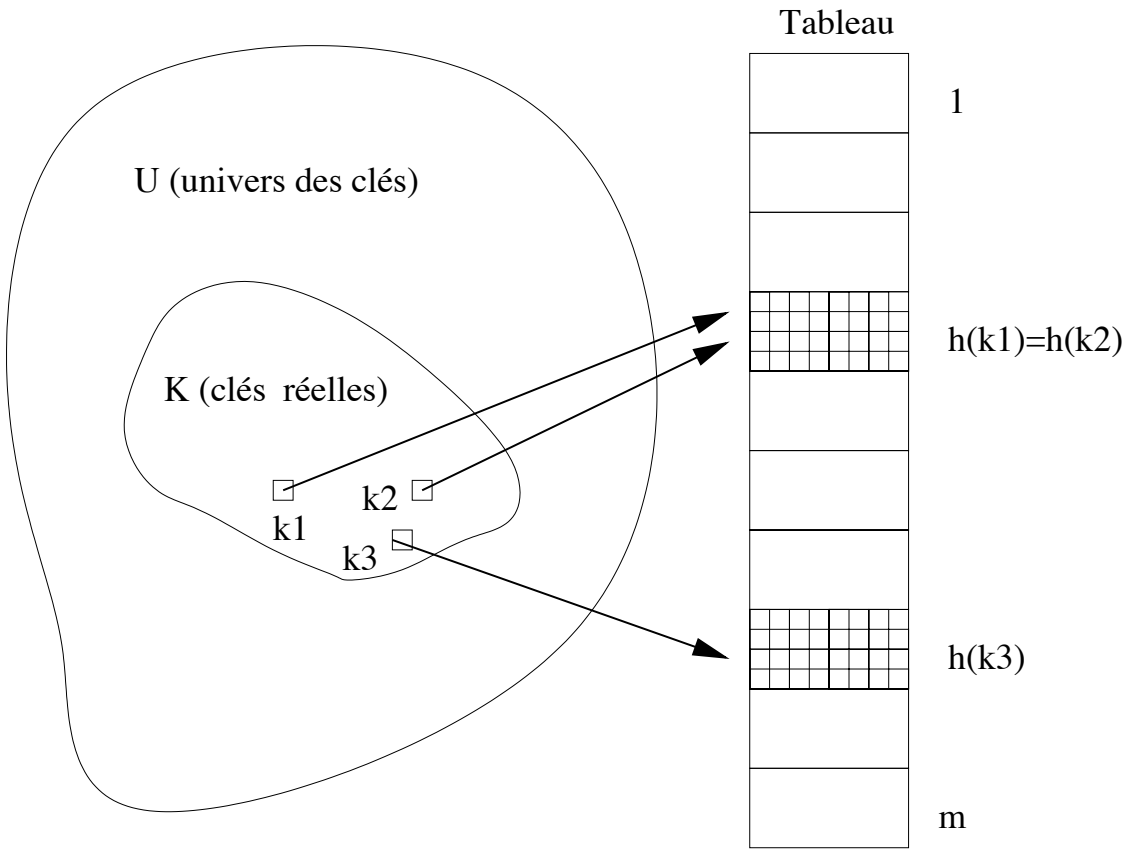
- Quand le nombre de clés possibles est largement supérieur au nombre de clés utilisés ($U >> K$), cette technique se montre mal adaptée :
  - l'adressage direct gaspille une partie importante de la mémoire réservée ;
  - il est parfois impossible d'allouer l'espace mémoire nécessaire au tableau.

Les tables de hachage

- Quand le nombre de clés possibles est largement supérieur au nombre de clés utilisés ($U >> K$), cette technique se montre mal adaptée :
  - l'adressage direct gaspille une partie importante de la mémoire réservée ;
  - il est parfois impossible d'allouer l'espace mémoire nécessaire au tableau.
- Les tables de hachage proposent de calculer l'indice où doit être stockée une valeur en fonction de sa clé.
- L'objectif est double :
  1. conserver un accès efficace à une donnée (en moyenne)
  2. réduire l'espace mémoire gaspillé.

Plan

Les tables de hachage



Les tables de hachage

- La *fonction de hachage* h est chargée de calculer l'*alvéole* $h(k)$ (la position dans la table de hachage) ou sera rangée la valeur correspondant à la clé k.
- $h : U \to \{0, 1, ..., m-1\}$ où m est la taille de la table de hachage.

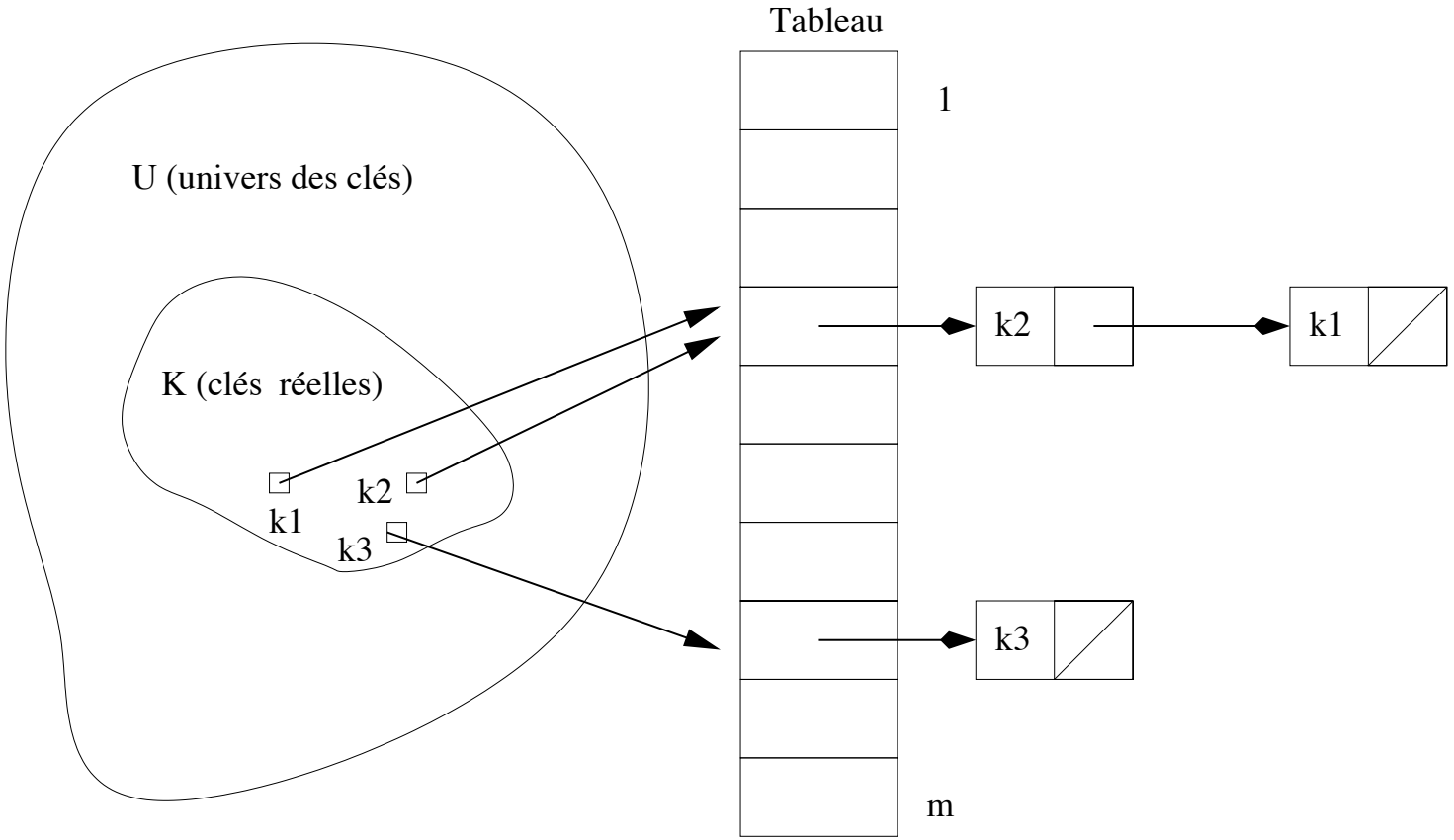
Les tables de hachage

- La *fonction de hachage* h est chargée de calculer l'*alvéole* $h(k)$ (la position dans la table de hachage) ou sera rangée la valeur correspondant à la clé k.
- $h : U \to \{0, 1, ..., m-1\}$ où m est la taille de la table de hachage.
- **Inconvénient** : 2 valeurs de clés différentes peuvent êtres hachées dans la même alvéole, on parle alors de *collision*.

  ⇝ Une bonne fonction de hachage minimise les collisions.
  ⇝ Il existe des techniques efficaces pour résoudre ces collisions.

Plan

Résolution des collisions par chaînage



Résolution des collisions par chaînage

- m : la taille de la table de hachage (le nombre d'alvéoles) ;
- n : le nombre de valeurs à stocker ;
- $\alpha = \frac{n}{m}$: le facteur de remplissage.

Comportement dans le pire des cas :

- très mauvais, tous les éléments sont hachés dans la même alvéole ;
- la longueur de la liste est en $O(n)$.

Comportement en moyenne (sous hypothèse de *hachage uniforme simple*) :

- chaque valeur a la même probabilité d'être hachée dans une alvéole quelconque ;
- coût d'une recherche : $\Theta(1 + \alpha)$ (à démontrer)

Recherche d'une alvéole libre

- Si une collision se produit au niveau de l'alvéole a, il est possible d'insérer non pas la valeur de clé k dans l'alévole calculée par la fonction $h(k) = a$, mais dans la première alvéole b libre en vérifiant $b > a$.

- Si une valeur $v_1$de clé k_1 est hachée dans une alvéole $h(k_1)$ occupée par une valeur v_2 de clé k_2 et $h(k_1) \neq h(k_2)$ (v_2 n'est pas à sa place), on cherche une nouvelle alvéole libre pour v_2 et on place v_1 dans son alvéole ($h(k_1)$).

- Dans ce cas, la table de hachage ne peut contenir plus de m valeurs.

Plan

Fonctions de hachage

Une bonne fonction de hachage vérifie l'hypothèse de hachage uniforme :

- soit $P(k)$ la probabilité pour que k soit tirée ;
- hypothèse de hachage uniforme :

$$\sum_{k|h(k)=j} P(k) = \frac{1}{m} \forall j = 0, 1, ..., m-1.$$

En pratique P est souvent inconnue, on utilise alors des *heuristiques*.

La plupart des fonctions de hachage supposent que l'univers des clés U est l'ensemble des entiers naturels $\mathbb{N}$.

La méthode de la division

Pour trouver l'alvéole où sera hachée la valeur de clé k, on considère le reste de la division entière par m, la taille de la table :

$$h(k) = k\%m.$$

Par exemple, si $m = 11, k = 15$ alors $h(k) = 15\%11 = 4$.

La méthode de la division

Pour trouver l'alvéole où sera hachée la valeur de clé k, on considère le reste de la division entière par m, la taille de la table :

$$h(k) = k\%m.$$

Par exemple, si $m = 11, k = 15$ alors $h(k) = 15\%11 = 4$.

Cette fonction de hachage est très simple et obtient de bons résultats si la valeur de m est bien choisie :

- il faut éviter les nombres paires et les puissances de 2 ;
- il est préférable de choisir un nombre premier proche de la taille de la table de hachage souhaitée ;
- par exemple, si on souhaite une table contenant au moins 700 alvéoles, la valeur $m = 701$ est préférable.

La méthode de la multiplication

- Le principe :
  - on choisit A une constante telle que $0 < A < 1$;
  - on calcule la partie fractionnaire de $k \times A$, notée $\lfloor kA \rfloor$ (k est la clé de la valeur à hacher) ;
  - $h(k)$ sera alors égal à la partie entière de $m \times \lfloor kA \rfloor$.
- Ce principe permet de choisir une valeur quelconque de m.
- En pratique la valeur $A = \frac{(\sqrt{5}-1)}{2} = 0,61803...$ fonctionne bien.