CH.0 RAPPELS

- 0.1 Les types de base
- 0.2 Les listes
- 0.3 Les piles et les files
- 0.4 Le hachage

0.1 Les types de base

Objets indépendants du langage de programmation (types abstraits)
avec des opérations associées à ces objets.

Types de base

Exemples : nombres entiers, opérations arithmétiques sur les entiers,
nombres réels, opérations arithmétiques,
booléens, opérations logiques,
caractères,
pointeurs.

Implémentation des types de base dépendante de l'environnement matériel (processeur, mémoire) et de l'environnement logiciel (langage).

Entiers naturels – Entiers en pratique (sur deux octets…)
Opérations différentes, erreur sur dépassement…
Implémentation courante pour calcul : *complément à 2*

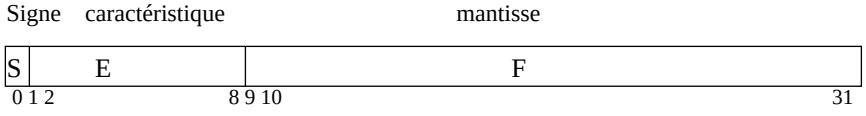
$r = 547$	Bin (r)	0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 1
		sur 16 bits
$-r = -547$	Bin ($2^{16} - r$)	1 1 1 1 1 1 0 1 1 1 0 1 1 1 0 1

L2-2 ch0 3

Nombres réels
Nombreuses opérations et fonctions de base.
Implémentation : objets très différents des nombres "mathématiques" : les axiomes ne sont plus satisfaits qu'"approximativement".
Plusieurs possibilités : virgule fixe, virgule flottante, …
Les résultats sont généralement dépendants de l'implémentation.

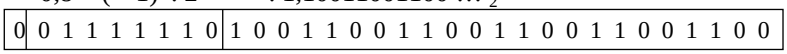
Implémentation (standard IEEE)

$x = (-1)^s \cdot 2^{E-127} \cdot 1, F$

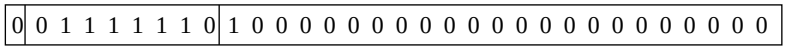


L2-2 ch0 4

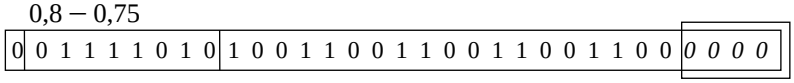
$0{,}8 = (-1)^0 \cdot 2^{126-127} \cdot 1{,}10011001100\ldots_2$



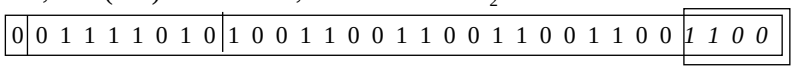
$0{,}75 = (-1)^0 \cdot 2^{126-127} \cdot 1{,}100\ldots_2$



$0{,}8 - 0{,}75$



$0{,}05 = (-1)^0 \cdot 2^{122-127} \cdot 1{,}10011001100\ldots_2$



L2-2 ch0 5

0.2 Les listes

Suite d'éléments $(e_1, e_2, e_3, \ldots, e_n)$ de même type.
Accès séquentiel $e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow \ldots \rightarrow e_n$
Parcours : tête, successeur, fin.
Fonction longueur de la liste.
Listes de base : liste vide (longueur 0),
liste de longueur 1.
Constructeur : concaténation de listes.
Opérations : recherche, effacement, insertion d'éléments.

Implémentations possibles :
- tableaux,
- pointeurs,
- curseurs,
- tables de hachage.

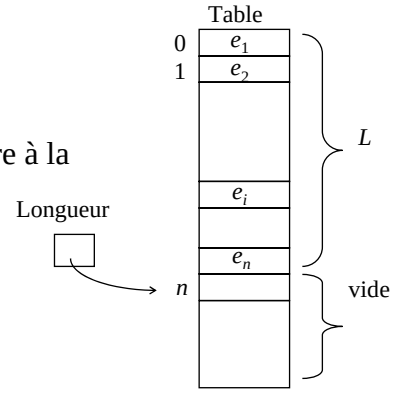
L2-2 ch0 6

Choix de l'implémentation.
  Dépend des opérations envisagées et du langage :
    compromis entre simplicité, rapidité et efficacité.

Implémentation par tableau :
  liste = structure (longueur, tableau)

$L = (e_1, e_2, \ldots, e_n)$

Avantages :
  simple à implémenter ;
  bon contrôle de la place ;
  nécessité de maintenir la longueur (inférieure à la taille du tableau !) ;
  accès rapide (fonction successeur = + 1) ;

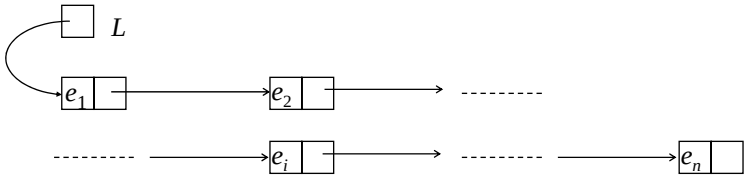


Inconvénients :
  concaténation difficile ;
  modifications lentes, nécessitant souvent la recopie (insertion ou suppression d'un élément).

Convient bien pour des listes statiques.

Variante :
  chaîne de caractères (en C, dans string.h) :
    tableau de caractères (adresses consécutives) dont la longueur est celle de la chaîne + 1, terminé par 0.

Implémentation par pointeurs : liste chaînée



Chaque cellule est une structure comportant un pointeur vers cellule (type récursif).
L'adresse d'une cellule et la liste sont du même type pointeur vers cellule (il peut être intéressant de les nommer différemment…)
L'accès se fait en suivant les pointeurs.

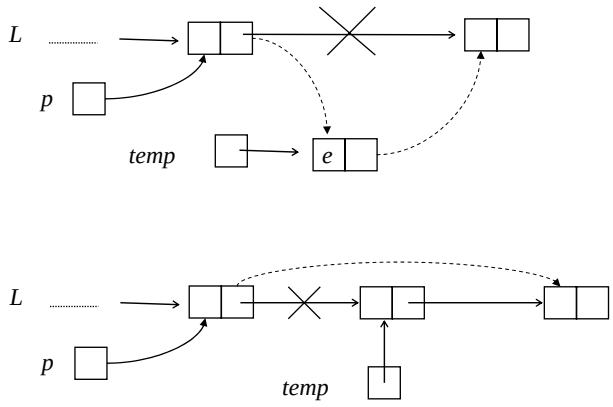
Avantages :
rapidité des opérations, car pas de recopie des données ;
adjonction, effacement, fusion de listes faciles ;
gestion dynamique de la mémoire ;
implémentation se prêtant bien à la récursivité (liste = tête de liste suivie du reste de la liste).

Inconvénients :
gestion dynamique de la mémoire (allocations et libérations explicites) ;
un peu plus gourmand en mémoire ;
pas d'accès direct et accès séquentiel dans une seule direction.

Par exemple, insertion d'un élément dans une liste

```
typedef struct CELLULE {
       element elt ;
       struct CELLULE  *  succ ;
} cellule ;
typedef  cellule  *  adresse ;
typedef  adresse  liste ;
...
liste ajouter ( liste L, adresse p, element e ) {
       temp = (adresse) malloc(sizeof(cellule)) ;
       if ( temp == NULL ) erreur() ;
       else {
              temp -> elt = e ;
              temp -> succ  =  p -> succ ;
              p -> succ  =  temp ;
       }
       return L ;
}
```


Insertion et suppression d'une cellule en temps constant:

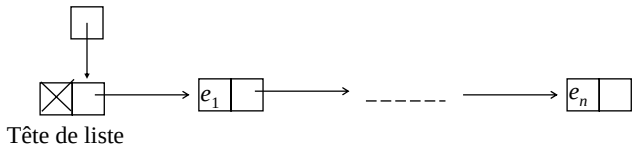


Seule la recherche est en temps linéaire.

Variantes et perfectionnements :

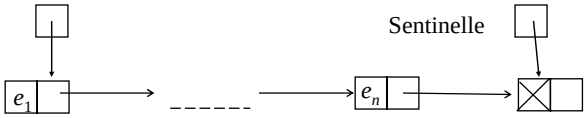
Tête de liste

$L = (e_1, e_2, \ldots, e_n)$



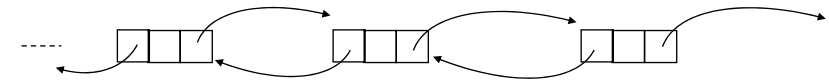
Permet de traiter le premier élément de la liste comme tous les autres (il a un prédécesseur) et l'adresse de la liste n'est pas modifiée par les insertions/suppressions.

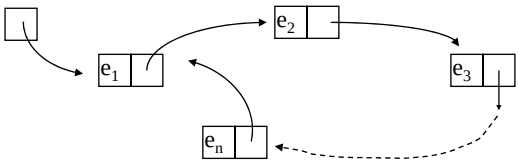
Sentinelle



Permet de donner un successeur au dernier élément de la liste. On ne teste plus si le pointeur est nul, mais s'il est égal à la sentinelle.
On peut utiliser la place pour mettre une information temporaire (cas de la recherche séquentielle).

Double chaînage





Liste circulaire

L2-2 ch0 15

Implémentation par curseurs

$L = (e_1, e_2, \ldots, e_n)$

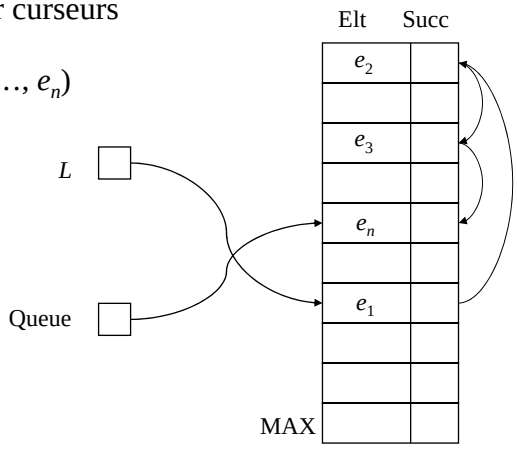


Tableau de taille MAX de structures dont le 2e champ est un indice.

L2-2 ch0 16

C’est une variante des pointeurs : les indices correspondent à des adresses.

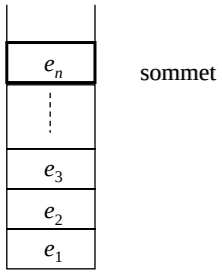
On contrôle la gestion de la mémoire (MAX ?)

Inconvénient : trouver un emplacement libre.
Solutions possibles :
placer une valeur conventionnelle (recherche en temps linéaire) ;
maintenir une liste des emplacements libres

0.3 Les piles et les files

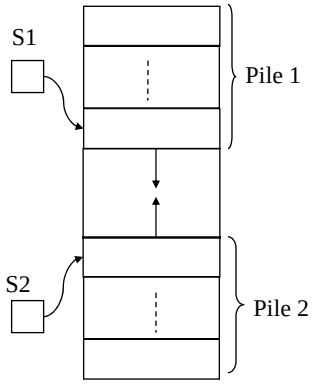
Pile = liste "last in-first out" LIFO.
Liste dans laquelle seul l’accès à une extrémité est possible.
Cette extrémité est appelée sommet de la pile.

Pile vide ;
Empiler un élément ;
Dépiler un élément si la pile n’est pas vide.



Implémentation possible :
tableau et valeur du sommet de la pile
(nécessite un indice spécial pour pile vide).
variante avec l’indice du sommet + 1 (premier emplacement libre).

Implémentation de 2 piles dans un unique tableau.
Tester que les pointeurs de sommet restent différents.



File = liste "first in-first out" FIFO.
Liste dans laquelle seul l'insertion et la lecture se font à des extrémités différentes.
Synonymes : file de priorité, queue.

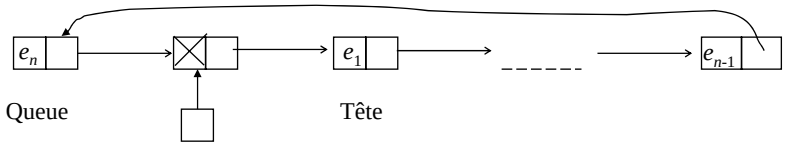
$$F = (e_1, e_2, \ldots, e_n)$$

On enlève par la gauche (tête) et on ajoute par la droite (queue).

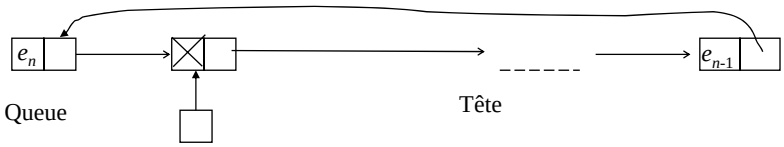
Les éléments vont se rapprocher de la tête en cours d'utilisation.

Idée pour l'implémentation : laisser les éléments fixes et déplacer la tête et la queue.

Implémentations possibles :
File chaînée traitée comme une liste chaînée circulaire.

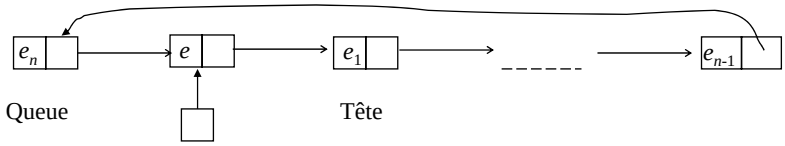


Accès et effacement de e_1 (sauf si file vide) :

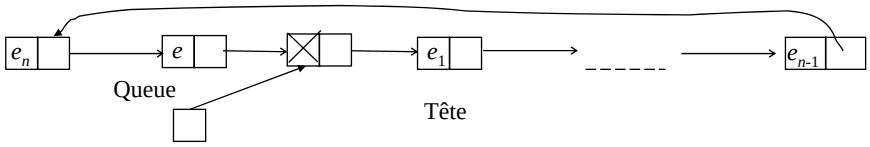


Actualisation de la tête par modification d'un pointeur.

Insertion de e :

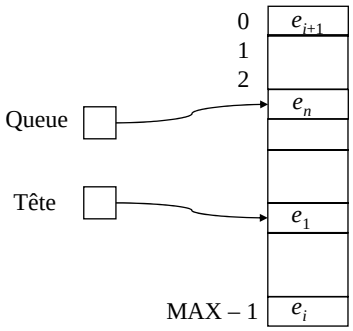


Suivi d'actualisation de la queue :



Deux opérations en temps constant.

Implémentation dans un tableau



Effacement de e_i par incrémentation de l'indice Tête ;
Insertion d'un nouvel élément par écriture en position Queue + 1 et incrémentation de Queue.
Tous les calculs doient être faits modulo MAX.
Il est utile de garder la longueur de la file.

Utilisation : mémoire tampon (buffer circulaire).

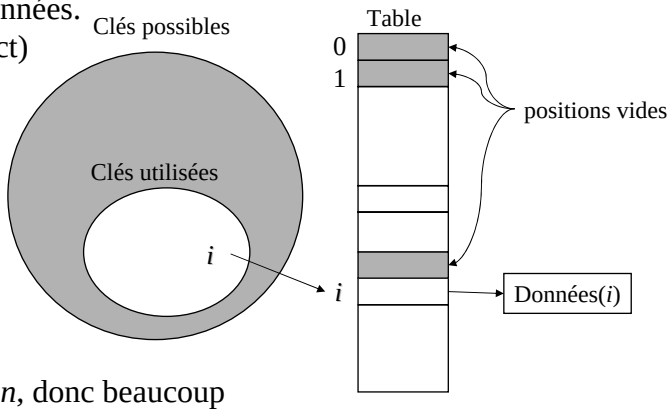
0.4 Le hachage

Données auxquelles on accède par des *clés*.
Par exemple, dictionnaire auquel on accède par la chaîne de caractères constituant le mot, ou informations sur des individus au moyen d'un numéro d'identification...

A deux données différentes correspondent deux clés différentes.
On manipule les données (consultation, insertion, effacement) à travers leur clé.

Supposons que les clés soient des nombres entiers (numéro national d'identification NNI, ...)
L'univers des clés est l'ensemble des clés potentiellement utilisables, contient n éléments (2^{13} pour le NNI).
Parmi ces clés, on en utilise effectivement k (65 millions pour le NNI).

Représentation de ces données.
Par table (adressage direct)

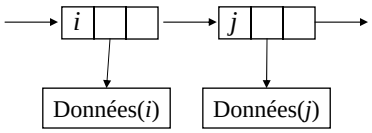


Avantages :
Consultation, insertion et effacement des données en temps constant $O(1)$.

Inconvénients :
Taille de la table égale à n, donc beaucoup de positions vides.

C'est comme ça qu'est gérée la table des fichiers. Les clés sont les i-noeuds.

Par liste chaînée triée



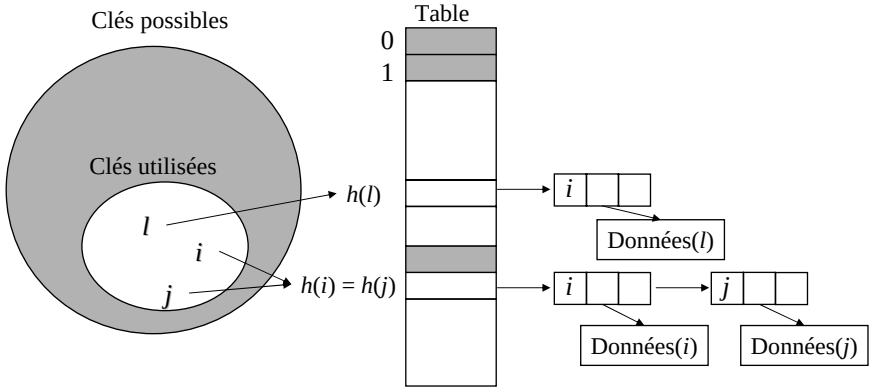
Avantages :
N'utilise que la place strictement nécessaire.

Inconvénients :
Consultation, insertion et effacement des données en temps moyen proportionnel à k, donc $O(k)$.

Cela peut devenir prohibitif lorsque k est grand.

Les deux techniques sont complètement opposées.
Une position médiane peut être utilisée.

Par tables de hachage



Une *fonction de hachage h* est utilisée. L'ensemble des valeurs prises est beaucoup plus petit que *n*. Deux clés distinctes peuvent donner le même résultat (*collision*). Les collisions peuvent être résolues par chaînage.

Problèmes :
Trouver une "bonne" fonction *h*.
Trouver la taille "optimale" de la table.

Si la fonction de hachage est uniforme, les listes chaînées auront la même longueur $\alpha = k$ / taille de la table, appelé *facteur de charge*.
Dans ce cas les opérations sur les données prendront un temps $O(1 + \alpha)$ en moyenne, et $O(1)$ pour l'insertion. Mais il s'agit de complexité **en moyenne**. Dans le pire des cas, la complexité est en $O(k)$.

La fonction *h* doit être aussi uniforme (et simple à calculer) que possible. La valeur retournée est en général un entier, ce qui permet d'utiliser la valeur comme indice de la table.
Choix possible : le reste dans la division par la taille de la table. Il faut éviter les puissances de 2 et les puissances de 2 moins 1. Un nombre premier est un bon choix.

Autre possibilité, multiplier la clé par une constante A, prendre la partie fractionnaire et multiplier par la taille de la table.
Cela permet d'utiliser une table dont la taille est une puissance de 2.
Knuth suggère d'utiliser $A = (\sqrt{5} - 1)/2 = 0{,}61803...$

Il faut aussi que la table soit de taille suffisante pour minimiser le nombre de collisions (paradoxe des anniversaires).

Il est aussi possible de prévoir de gérer les collisions par une table dont les éléments contiennent plusieurs pointeurs utilisables, ce qui évite le recours aux listes chaînées.

D'autres stratégies existent.