Cours n°1 : LE TDA LISTE

- Structures de données et algorithmes en C
- Cours n°1 : le TDA LISTE
    1. **Qualités d'un logiciel**
    2. Les types de données abstraits
    3. Le TDA LISTE (TDA)
    4. Opérations fondamentales sur les listes
    5. Algorithmes abstraits sur les listes
    6. Réalisations du TDA LISTE
    7. Comparaison des réalisations

Bibliographie

- Aucun livre ne correspond exactement à cette partie du cours, mais ceslivres peuvent vous être particulièrement utiles en particulier pour le cours n°1 :

- Mon préféré (très bien sur les TDA, mais il est en Pascal)
  - **C. FROIDEVAUX, M.-C. GAUDEL, M. Soria, Types de données et algorithmes, Ediscience international, 1993, 577 p.**
- En C mais pas très orienté TDA
  - **HOROWITZ, SAHNI, ANDERSON, FREED, L'essentiel des structures de données en C, Dunod, 1993, 550 p.**
- Pour ceux qui ont encore du mal en C, indispensable
  - **KERNIGHAN, RITCHIE, Le langage C : Norme ANSI, (2nde ed), Dunod, 2004, 280 p.**

Attention

- Nous adopterons une perspective Prog Objet, génie logiciel
  - Cela nous amène à nous intéresser aux concepts **d'abstraction**, **d'encapsulation**, **d'interface**.
- Dans d'autres modules vous aborderez le C sous une autre perspective (système, compilation)
  - vous y aborderez les problèmes d'optimisation, d'arithmétique de pointeurs, utilisations de variables globales, de variables de modules…
  - Ces perspectives sont plus complémentaires que contradictoires :
    - il faut raisonner abstrait à certains moment du cycle de développement d'un logiciel et se préoccuper d'optimisation et de performance pratique à d'autres.
    - Si ces différences de perspectives vous posent problème, n'hésitez pas à poser des questions et à en discuter avec les enseignants concernés

Préoccupations de génie logiciel

- Ce cours est **un cours de programmation**, ce n'est pas un cours de Génie Logiciel, mais ces préoccupations doivent être présentes quand on programme

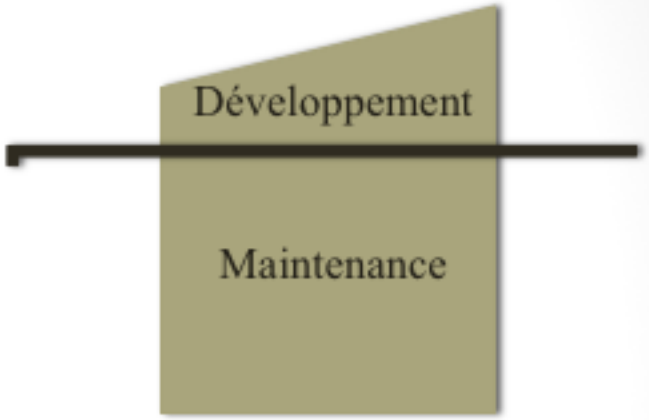
- Qualités d'un logiciel
  - **Valide** : Réalise exactement les tâches définies par la spécification
  - **Robuste** : Fonctionne même dans des conditions anormales
  - **Fiable** : Valide + robuste
  - **Extensible** : S'adapte aux changements de spécification
  - **Réutilisable** : En tout ou en partie dans de nouvelles applications
  - **Compatible** : peut être combiné avec d'autres

  - Autres : **Efficace**, **portable**, **vérifiable**, facile d'utilisation

Préoccupations de génie logiciel

- En C (et aussi en C++) les problèmes de **validité**, de **robustesse** et de fiabilité sont souvent liés aux **pointeurs**
  - i.e. aux problèmes de gestion de la mémoire dynamique
- **L'Efficacité** (obtenir un programme dont le temps d'exécution est minimal) est une préoccupation :
  - primordiale pour les applications temps réel, pas dans ce module (et en GL)
    - On s'intéresse à l'efficacité théorique des algorithmes (en o(n) en o(nLog n))
      - pas aux problèmes d'optimisation liés à la compilation, aux astuces du C (arithmétique de pointeur), aux systèmes hôtes
    - On ne dit pas que ces problèmes n'ont pas leur importance, mais
      - ils sont enseignés dans d'autres modules
      - d'autres problèmes plus importants dans une perspective GL ou POO sont au programme de ce module
- **Extensibilité**, **réutilisabilité**, **compatibilité** : préoccupations très importantes de la programmation objet

Préoccupations de génie logiciel



- La **Maintenance**
  - Partie immergée de l'iceberg

- Estimations pour donner des ordres de grandeurs **en temps consacré (**pas une hiérarchie d'importance)
  - Adaptation à l'évolution des besoins (42 %)
  - Modification des formats de données (17 %)
  - Corrections d'erreurs urgentes (12 %)
  - Corrections d'erreurs mineures (9 %)
  - Modification de matériel (6 %)
  - Documentation (5 %)
  - Amélioration de l'efficacité (4 %)

Le Programmeur du 21ème siècle

- Membre d'une équipe
  - Doit maintenir ou faire évoluer du code écrit par d'autres
    - **"Le meilleur code est celui qui est déjà écrit »**
  - Doit s'adapter aux utilisateurs
  - Doit choisir et utiliser sur des gros projets
    - Des bibliothèques de code déjà écrit
    - Des ateliers de génie logiciel
    - Des normes de qualité, guides de style, "design pattern »

- Attention au Bidouilleur génial isolé
  - Quelque fois utile
  - Souvent considéré comme une calamité
  - En un mot : **ringard**

Le meilleur code est déjà écrit

- Dans ce cas pourquoi enseigner encore la programmation ?
  - pour pouvoir choisir, il faut connaître
  - vous ne travaillerez pas tous sur des gros projets, dans des grosses boîtes
  - les ateliers de génie logiciel c'est certainement l'avenir mais ils sont encore bien lourds et complexes,
    - le code généré n'est pas toujours des plus efficaces
  - tant que les entreprises préféreront des langages de bas niveau comme C et C++ à de beaux langages comme ADA, Eiffel, Caml ou Ruby vous serez tranquilles, y aura du boulot pour vous !

- L'objectif est de vous préparer à être **les informaticiens de demain** mais vous insérant **dans le monde d'aujourd'hui**

Principes

- Se placer à un niveau d'abstraction élevé
  - Éviter les erreurs de conception
- Programmer avec des opérations de haut niveau
  - Qui ne dépendent pas de la représentation interne
  - Qui permettent de changer de représentation
  - Qui permettent à l'utilisateur de se concentrer sur les problèmes de conception en ignorant les détails de réalisation
- Encapsuler les données
  - N'accéder à la représentation interne que via des fonctions
  - L'utilisateur ne voit que les services pas la représentation interne
    - exemple :
      - on manipule les entiers par les opérations + - % / < > =
      - exceptionnellement (efficacité) en utilisant leur représentation interne

Principes

- A retenir jusqu'à la fin de vos jours : **Règle empirique des 90-10** (80-20) :
  - Un programme passe 90 % de son temps sur 10 % du code.
    - Ce sont ces 10 % là sur lesquels l'efficacité est cruciale où l'on s'intéressera à la représentation interne.
    - Pour le reste du programme les autres critères (lisibilité, maintenabilité etc.) sont primordiaux et on restera à un niveau d'abstraction élevé.
- **Utilisateur** ? : Pour ce cours, L'utilisateur est le programmeur qui utilisera les fonctions que vous avez écrites, vous concepteur du TDA.
  - Nous programmons pour d'autres programmeurs.
  - En IHM (Interface Homme Machine) l'utilisateur est soit le client soit la personne qui manipulera le logiciel interactif

Cours n°1 : LE TDA LISTE

- Structures de données et algorithmes en C
- Cours n°1 : le TDA LISTE
    1. Qualités d'un logiciel
    2. **Les types de données abstraits**
    3. Le TDA LISTE (TDA)
    4. Opérations fondamentales sur les listes
    5. Algorithmes abstraits sur les listes
    6. Réalisations du TDA LISTE
    7. Comparaison des réalisations

- **Orientation du cours** :
    - Premier temps : programmer à un haut niveau d'abstraction
    - Deuxième temps : passer à la réalisation dans un langage et se poser, alors seulement les problèmes techniques et d'efficacité.

Les types de données abstraits (TDA)

- **Définition** 1 :
  - Un ensemble de données et d'opérations sur ces données
- **Définition** 2 :
  - Un ensemble de données organisé de sorte que les spécifications des objets et des opérations sur ces objets soient séparées de la représentation interne des objets et de la mise en œuvre des opérations
- Nés de préoccupation de génie logiciel
  - **Abstraction**
  - **Encapsulation**
  - Vérification de types

Les TDA Exemples

- Le type entier
  - muni des opérations + - % / < > = est un TDA prédéfini
- Une pile
  - munie des opérations initialiser, empiler, dépiler, consulter le sommet de pile) est un TDA
- Les TDA généralisent les types prédéfinis.
  - ADA, C++ sont conçus pour supporter la réalisation de TDA
  - Smalltalk supporte l'abstraction et l'encapsulation mais pas la vérification de type à la compilation
  - en C comme vous pourrez le constater, utiliser les TDA c'est assez acrobatique et même contre nature car C est un langage inventé pour écrire UNIX pas pour écrire des TDA

Incarnation d'un TDA

- **Définition** :
  - Une incarnation (une réalisation, une mise en œuvre) d'un TDA est
    - La déclaration de la structure de données particulière retenue pour représenter le TDA
    - Et la définition (i.E le code)des opérations primitives dans un langage particulier
- On parle aussi de type concret de données
  - Exemple : on peut mettre en œuvre le TDA PILE en utilisant en mémoire des cellules contiguës (tableau) ou des cellules chaînées (Pointeurs).
  - On a donc plusieurs incarnations possibles pour un même TDA
- **Attention** : Définir le jeu des opérations primitives peut s'avérer un problème délicat.
  - **Opérations primitives** : la définition (le code) des fonctions utilise la représentation particulière choisie pour "incarner" le TDA
  - **Opérations non primitives** : la définition des opérations non primitives utilise les fonctions primitives mais est indépendante de la représentation particulière choisie

Incarnation d'un TDA

- **Remarques :** Pourquoi le langage C ? : C'est le plus répandu
  - On ne peut pas sortir d'une filière informatique sans maîtriser le C.
  - Tirer la langue pour mettre en œuvre les TDA en C vous permettra aussi de mieux comprendre les problèmes résolus par les langages Objets
  - pour obtenir une plus grande sécurité de programmation
- **Programmation à l'aide des TDA:**
  - Les programmeurs ont deux casquettes
    - le **concepteur** du TDA qui met en œuvre (écrit le code) les primitives et doit connaître la représentation interne adoptée
    - l'**utilisateur** du TDA qui ne connaît que les services (les opérations) et n'accède jamais à la représentation interne
  - en C :
    - l'utilisateur n'accède qu'au fichier .h (interface)
    - le programmeur accède aussi au fichier .c (réalisation)

Avantages des TDA

- Résumé :

TDA	Algorithme Abstrait	Utilisateur	Interface
Incarnation	Réalisation dans un langage Informatique	Programmeur	Réalisation

- Écriture de programmes en couches :
  - On écrira : **empiler(x, P)** *et pas* : **tab[sp++]**
    - La couche supérieure traite le problème dans les termes du domaine de problèmes (empiler(x, P))
    - La couche inférieure entre dans les détails du langage utilisé (tab[sp++])
- Séparation claire des offres de service et du codage
- Facilité de compréhension et d'utilisation
- Prise en compte de types complexes
- Briques d'une structuration modulaire rigoureuse

Inconvénients des TDA

- Le cout
  - L'utilisateur d'un TDA connaît les services mais ne connaît pas leur coût
  - Le concepteur du TDA connaît le coût des services mais ne connaît pas leurs conditions d'utilisation
  - Le coût des opérations dépend des réalisations : selon la réalisation certaines opérations peuvent être lentes ou efficaces mais il n'existe pas de réalisation plus efficace dans tous les cas.
    - Selon que l'on utilise plus fréquemment certaines opérations ou d'autres on préférera une représentation ou une autre.
- Le choix des primitives est quelque fois difficile à faire

Cours n°1 : LE TDA LISTE

- Structures de données et algorithmes en C
- Cours n°1 : le TDA LISTE
    1. Qualités d'un logiciel
    2. Les types de données abstraits
    3. **Le TDA LISTE (TDA)**
    4. Opérations fondamentales sur les listes
    5. Algorithmes abstraits sur les listes
    6. Réalisations du TDA LISTE
    7. Comparaison des réalisations

Le TDA LISTE

- Définition.
  - Une liste est une suite finie (éventuellement vide) d'éléments.

- Liste **homogène** : tous les éléments sont du même type
  - une liste d'entiers : l = (2 4 6 8)
  - liste de char * : l = ("bonjour " "toto" "vive" "le" "C")
  - Ada, C++, Caml, Pascal, C, Java ne tolèrent que des listes homogènes

- Liste **hétérogène** : les éléments sont de type différents.
  - l = ("toto" 2 (4 10 13) 'a')
  - Smalltalk, Lisp, Ruby tolèrent des listes hétérogènes

Hypothèses

- Dans la suite nous intéressons aux listes homogènes
- Tous les éléments appartiennent à un TDA ELEMENT qui possède un élément vide et sur lequel on peut :
  - Tester si deux éléments sont identiques (**ElementIdentique**)
  - Affecter dans un élément un autre élément (**ElementAffecter**)
  - Allouer la mémoire ou initialiser un élément (**ElementCreer**)
  - Libérer la mémoire allouée pour un élément (**ElementDetruire**)
  - Saisir un élément (**ElementLire**)
  - Afficher un élément (**ElementAfficher**)
- On dispose d'un TDA POSITION permettant de repérer un élément dans la liste sur lequel on peut :
  - Tester si deux positions sont égales (**PositionIdentique**)
  - Affecter dans une position une autre position (**PositionAffecter**)

Hypothèses

- Conventions de dénomination:
  - nous préfixons l'opération par le nom du TDA (en attendant mieux) pour distinguer par exemple **ListeLire** et **ElementLire**
    - pas d'utilisation du souligné mais *Camelcase*
    - procédures (verbes à l'infinitif) et fonctions (résultat du calcul)
- ELEMENT est d'un type quelconque (entier, caractère, chaîne ou tout type utilisateur muni des 4 opérations (**affecter**, **tester** l'égalité, **lire** et **afficher**)
  - Pour les éléments qui utilisent de la mémoire dynamique ou qui ont besoin d'être initialisés, on a besoin de primitives pour allouer et désallouer la mémoire dynamique (**créer** et **détruire**).
- POSITION peut être le rang de l'élément ou un pointeur sur la cellule contenant un élément (ou sur celle d'avant)
  - Lors de l'incarnation du TDA, une POSITION sera un entier ou un pointeur (mais ici ce n'est pas le problème)

Signature du TDA Liste

- Élément particulier : liste vide
- Utilise : ELEMENT, POSITION
- Fonctions primitives :
  - Allocation, libération de mémoire
    - **ListeCreer**, **ListeDetruire**
  - Gestion de la liste (consultation, modification)
    - **ListeAccèder**, **ListeInsérer**, **ListeSupprimer**
  - Parcours
    - **ListePremier**, **ListeSuivant**, **ListeSentinelle**
  - Test
    - **ListeVide**

- On définit ainsi un jeu de primitives possibles sur les listes.
  - Vous en étudierez un autre en TD qui utilise une définition récursive des listes.
    - Le choix des primitives dépend des opérations qui seront réalisées le plus souvent.

Primitives du TDA Liste

- **ListeCreer** : pas d'arguments ; alloue dynamiquement de la mémoire pour une liste vide et retourne cette liste
- **ListeDetruire** : un argument (la liste à détruire) ; libère la mémoire allouée dynamiquement pour la liste
- **ListeVide** : un argument (la liste à tester) ; retourne vrai si la liste est vide et faux sinon
- **ListeAcceder** : deux arguments (une position p et une liste l) ; retourne l'élément à la position p dans l (ou élément vide si l est vide ou si la position p est erronée)
- **ListeInserer** : trois arguments (un élément x, une position p et une liste l); modifie la liste l en insérant x à la position p dans l. Retourne vrai si l'insertion s'est bien passée et faux sinon

Primitives du TDA Liste

- **ListeSupprimer** : 2 arguments (une position p et une liste l) ; supprime l'élément à la position p dans la liste l et retourne vrai si la suppression s'est bien passée (et faux sinon)
- **ListeSuivant** : 2 arguments (une position p et une liste l) ; retourne la position qui suit p dans la liste (ou la sentinelle si la liste est vide ou si p n'est pas une position valide)
- **ListePremier** : 1 argument (la liste l) et retourne la première position dans la liste
- **ListeSentinelle** : 1 argument (la liste l) et retourne la position qui suit celle du dernier élément de la liste
  - une sentinelle est une astuce de programmation qui consiste à introduire un élément factice pour faciliter les recherches.
    - La sentinelle est une position valide mais n'est la position d'aucun élément de la liste. Lorsqu'une liste est vide Premier(l) = Sentinelle(l).

Déjà on triche, ici on ne devrait pas parler de réalisation

Cours n°1 : LE TDA LISTE

- Structures de données et algorithmes en C
- Cours n°1 : le TDA LISTE
    1. Qualités d'un logiciel
    2. Les types de données abstraits
    3. Le TDA LISTE (TDA)
    4. **Opérations fondamentales sur les listes**
    5. **Algorithmes abstraits sur les listes**
    6. Réalisations du TDA LISTE
    7. Comparaison des réalisations

Opérations classiques sur les listes

- À l'aide de ces quelques primitives,
  - On peut exprimer les opérations classiques sur les listes,et ce,
    - Sans se soucier de la réalisation concrète des listes
    - Sans manipuler ni pointeurs, ni tableaux, ni faux curseur
    - Sans mettre les mains dans le « cambouis »

- Deux exemples :
  - **ListePrecedent**
  - **ListeLocaliser**
- D'autres exemples en TD et en TP
  - **raz**, **nettoyer, afficher**, **longueur**, **insérer** au début, à la fin, **supprimer** au début, à la fin ...

ListePrecedent

- Signature:
  - Deux arguments : une position p et une liste l
  - Retourne la position précédent p dans l si elle existe,
    - sinon retourne la sentinelle
- Algorithme informel
  - Parcourir l en faisant évoluer 2 positions
    - la position courante et celle d'avant
  - Arrêt du parcours
    - quand la position courante est égale à p ou à la sentinelle
  - Retourne la position avant ou la sentinelle

- Rappel des primitives des TDA ELEMENT et POSITION : Trans 20

ListeLocaliser

- Signature
  - Deux arguments : un élément x et une liste l
  - Retourne la première position où on rencontre x dans l
    - Sinon retourne la sentinelle
- Algorithme informel de recherche linéaire
  - Parcourir l avec une position courante
  - Arrêt
    - quand l'élément courant est égal à x ou quand on atteint la fin de la liste
  - Retourne la position courante ou la sentinelle

- Rappel des primitives des TDA ELEMENT et POSITION : Trans 20

Algorithmes abstraits sur le TDA LISTE

- Algorithmes utilisant les opérations sur les listes
  - Exemple : **ListePurger** qui élimine les répétitions dans une liste
- Signature : trois signatures sont possibles
  - **ListePurger** modifie physiquement la liste initiale (qui est donc perdue) et qui est un argument passé par adresse. En C :
    - `ListePurger (&l)  /* Style Impératif */`
  - **ListePurger** retourne le résultat de la purge dans un argument passé par adresse. En C :
    - `ListePurger (&l, l1)  /* Style Impératif */`
  - **ListePurger** retourne une copie de la liste initiale, copie où les répétitions ont été enlevées. En C :
    - `l1=ListePurger(l)  /* Style Fonctionnel */`

Style de programmation impérative

- Le résultat est
  - Un paramètre de la procédure modifié physiquement à l'exécution, **en C : paramètre passé par adresse**

- Version 1 : purge sur place
  - `ListePurger(l)` modifie physiquement la liste l en supprimant les répétitions
- Version 2 : purge à côté
  - *`ListePurger(l, ll)`* modifie physiquement l qui est vide au départ et après contient une version purgée de ll (qui elle est inchangée)
- en C on est déjà moins abstrait dans ces deux premières versions :
  - pour que le paramètre soit modifié par la fonction `ListePurger` il faut qu'une liste soit une adresse (un pointeur sur quelque chose).

Style de programmation fonctionnelle

- Le résultat est retourné par la fonction au programme qui l'a appelée
- Ce type de programmation extrêmement puissant est mal supporté par le langage C
  - Aucun problème pour les résultats de type simple retourné par valeur
  - Deux problèmes pour les résultats locaux retournés par adresse :
    - Le choléra : pointeur fou
      - Exemple : retourner l'adresse d'un objet créé dans la pile lors de l'appel d'une fonction
    - La peste : fuite de mémoire
      - Dans notre Purge fonctionnelle La liste résultat est créée localement à la fonction ListePurger.
      - Pour qu'elle puisse être retournée il faut que la liste soit allouée dynamiquement dans la fonction.
      - Pas de problème nous avons ListeCreer pour cela. Oui mais qui la libérera ?
      - L'utilisateur ? Oui bien sûr, s'il y pense et s'il le peut.

Style de programmation fonctionnelle

```
LISTE ListePurger (LISTE L) {
        LISTE LL ;
        ELEMENT x ;
        POSITION p, q, fin ;
        LL = ListeCreer() ;
        fin = ListeSentinelle (L) ;
        p = ListePremier (L); q =  ListePremier (L);
        for ( ; p != fin ; p = ListeSuivant(p, L) ){
                x = ListeAcceder(p, L) ;
                if (ListeLocaliser(x, LL) == ListeSentinelle(LL)) {
                        ListeInserer (x, q, LL) ;
                        q = ListeSuivant(q, LL);
                        }
                }
        return LL ;
/* attention retour d'un pointeur déclaré dans la fonction
   La zone pointée a-t-elle été allouée dynamiquement (sinon pointeur
   fou) ? Si oui, Qui la libérera? (fuites de mémoires possibles
*/
}
```

Moralité : En C si l'on veut retourner par adresse un résultat créé localement dans une fonction, pour éviter la catastrophe du pointeur fou il faut que ce résultat ait été alloué dans le tas (en clair par un malloc) et dans ce cas le programme qui utilise la fonction court de gros risque de fuite de mémoire

Utilisation des TDA en C

- **Difficultés**: on voit apparaître sur l'exemple de ListePurger deux séries de difficultés:
  - la première est liée au langage C ;
    - on ne se soucie pas d'implémentation
      - **mais** pb du passage des arguments par valeur : notre algorithme "abstrait " (solution 1 et 2) ne fonctionne que si la liste est un pointeur
      - **mais** problème de la gestion de mémoire dynamique : la solution 3 (et dans une moindre mesure la 2) sont susceptibles de provoquer des problèmes de fuites de mémoire (on alloue de la mémoire et on laisse le soin à l'utilisateur de la fonction de libérer cette mémoire)
  - la deuxième est intrinsèque au TDA ;
    - tant qu'on ne connaît pas les coûts comparés de l'insertion au début ou à la fin de la liste on prend le risque de faire des purges douloureuses pour l'efficacité.
- **Retenez l'idée :**
  - **lors de la conception pensez abstrait ; lors de la mise au point il faut se soucier des représentations, du langage et de l'efficacité ; et en C de la gestion de mémoire**

Cours n°1 : LE TDA LISTE

- Structures de données et algorithmes en C
- Cours n°1 : le TDA LISTE
    1. Qualités d'un logiciel
    2. Les types de données abstraits
    3. Le TDA LISTE (TDA)
    4. Opérations fondamentales sur les listes
    5. Algorithmes abstraits sur les listes
    6. **Réalisations du TDA LISTE**
    7. Comparaison des réalisations

Réalisations du TDA LISTE

- Réaliser le TDA LISTE consiste à
  - Définir le TDA ELEMENT, cela peut être
    - Un objet (stockage direct)
    - Une adresse (stockage indirect)
  - Choisir une structure de données pour la représentation interne de la liste
    - Cellules contiguës (tableau)
    - Cellules chaînées (pointeurs)
    - Faux pointeurs (tableaux pour simuler les pointeurs)
  - Définir les primitives dans un langage de programmation

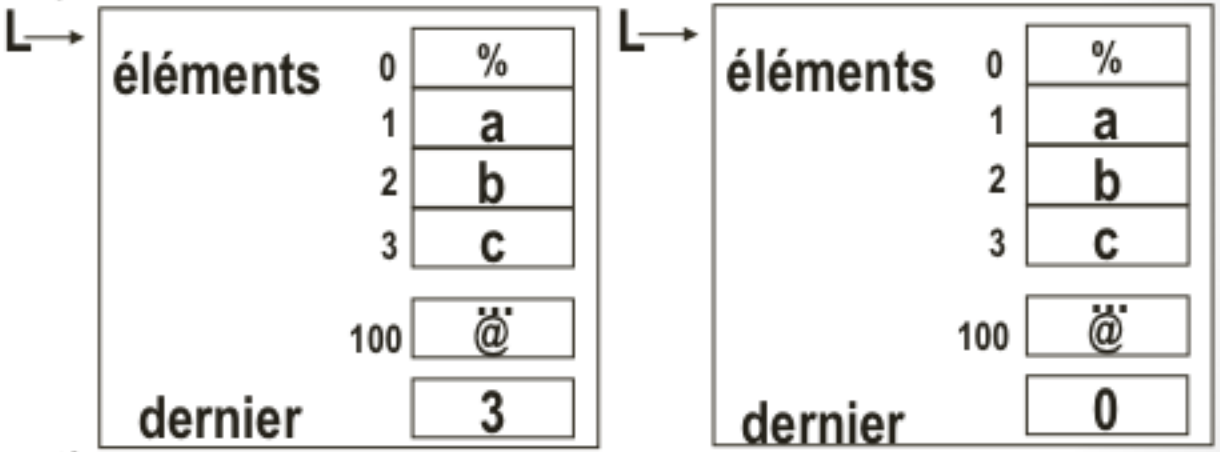
Stockage Direct et Indirect

- Exemples:
  - Stockage direct : l = (1 2 3 4)
  - Stockage indirect : l = (ad1 ad2)

- dans le listing joint au cours : le TDA ELEMENT est réalisé avec des entiers (stockage direct). En TD vous le réaliserez avec des fiches (stockage indirect) ; nous reviendrons sur le stockage indirect au cours 2 et en TP
- Pour l'instant **retenez** qu'en général un conteneur en stockage indirect n'est pas responsable de l'allocation et de la désallocation des objets qu'il contient ; il ne s'occupe que des pointeurs pas de objets pointés

Mise en œuvre des listes par cellules contiguës

- Une liste est un pointeur sur une structure à deux champs
  - Un tableau automatique qui contient les éléments de la liste
  - Un entier indiquant l'indice du dernier élément de la liste



- Choix de conception
  - Une liste est l'adresse d'une structure à modifier (indispensable pour les opérations sur les listes en style impératif)
  - Une position est un entier : l'indice de l'élément dans le tableau
  - Le premier élément de la liste est dans la case n°1 du tableau (pas dans la case 0)
  - La sentinelle est la position qui suit celle du dernier

Mise en œuvre des listes par cellules contiguës

- La liste est vide si le dernier élément du tableau est à la position 0
  - quand la liste est vide : la position du premier élément de la liste est 1 et la sentinelle est aussi 1

- PRIMITIVES
  - `ListeCréer` : allouer de la mémoire pour une telle structure et si l'allocation réussit mettre à 0 le champ dernier avant de retourner l'adresse de la structure
  - `ListeDetruire` : appeler free sur la liste L puisque la liste n'a pas de champ automatique (sauf si les éléments sont des pointeurs mais `LIsteDétruire` dans ce cas ne détruits pas les objets pointés)
  - Les fonctions de parcours et d'accès sont immédiates
  - Les fonctions de gestion de la liste :
    - `ListeInserer` : pour insérer à la position p il faut commencer par décaler à partir du dernier élément jusqu'au p ième; puis on insère dans la p ième case du tableau et on incrémente dernier
    - `ListeSupprimer` : consiste à tasser à partir de la position p jusqu'au dernier et à décrémenter dernier

Mise en œuvre des listes par cellules contiguës

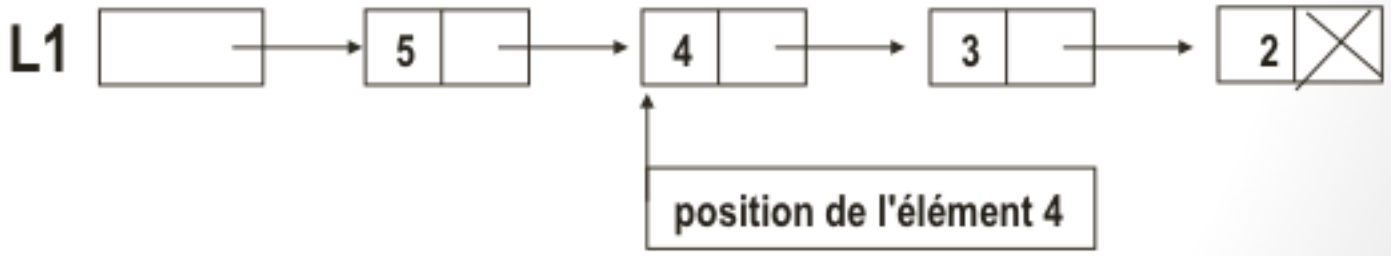
- En cas de stockage indirect, l'utilisateur doit se préoccuper du sort des objets pointés
  - qui n'est pas géré par `ListeSupprimer` ou `ListeDetruire` (et ce n'est ni un hasard ni un oubli)
- La Complexité :
  - `ListeInserer` et `ListeSupprimer` sont au pire des cas en o(n) (de l'ordre de n si n est le nombre d'éléments dans la liste) à cause des décalages
    - toutes les autres opérations sont en o(1) (en temps constant)
    - insérer et supprimer des éléments en fin de liste est en o(1) puisqu'alors on n'a pas besoin de décaler
- RÉALISATION EN C :
  - Cf. annexe du cours 1 fichier lsttab.h et lsttab.c

Mise en œuvre des listes par cellules chaînées

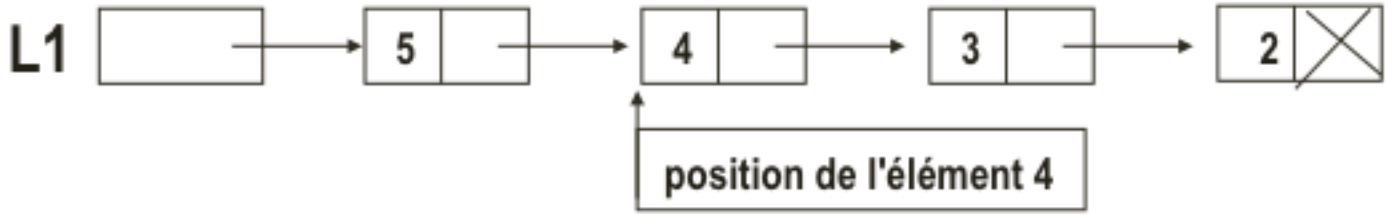
- Listes simplement chaînées sans en-tête
- Listes simplement chaînées avec en-tête
- Listes simplement chaînées circulaires
- Listes doublement chaînées

Listes simplement chaînées sans en-tête

- Une cellule est composée de deux champs :
  - Élément, contenant un élément de la liste
  - Suivant, contenant un pointeur sur une autre cellule
- LISTE : pointeur sur la cellule qui contient le premier élément
- POSITION : pointeur sur la cellule qui contient l'élément
- La liste vide est le pointeur NULL
- Inconvénients en C :
  - Insertion ou suppression d'un élément au début de la liste
  - Suppression d'un élément en **o(n)**



Listes simplement chaînées sans en-tête



- Les cellules ne sont pas rangées séquentiellement en mémoire ; d'une exécution à l'autre leur localisation peut changer
- Une position est un pointeur sur une cellule
- On a donc les déclarations suivantes :

```
typedef struct cell{
    ELEMENT element;
    struct cell * suivant;
    } cellule;
typedef cellule * POSITION, *LISTE;
```

Listes simplement chaînées sans en-tête

- **Problème 1** : quand on veut insérer ou supprimer au début de liste
  - ce n'est pas pareil qu'au milieu : donc à chaque fois il faut tester (si p = Premier(L)...)
  - de plus, on modifie physiquement la liste abstraite et on modifie le pointeur sur la première cellule ;
    - donc il faut passer un pointeur sur la liste en paramètre à ces procédures :
      - **ListeInserer(x, p, &L)** idem pour **ListeSupprimer (p, &L)**
- Ce n'est pas conforme à notre définition du TDA LISTE dans laquelle
  - le prototype de ListeInserer est `ListeInserer(x,P,L)`
  - Le prototype de ListeSuprimer est `ListeSupprimer(p,L)`
  - (cf. Transparent 23 et 24)

Listes simplement chaînées sans en-tête

- **Exemple** : insérer x au début de la liste L1

1. créer une cellule

```
cell = (cellule *) malloc (sizeof(cellule))
    (et tester si tout va bien)
```

2. initialiser la cellule avec x et un pointeur sur la cellule qui contient

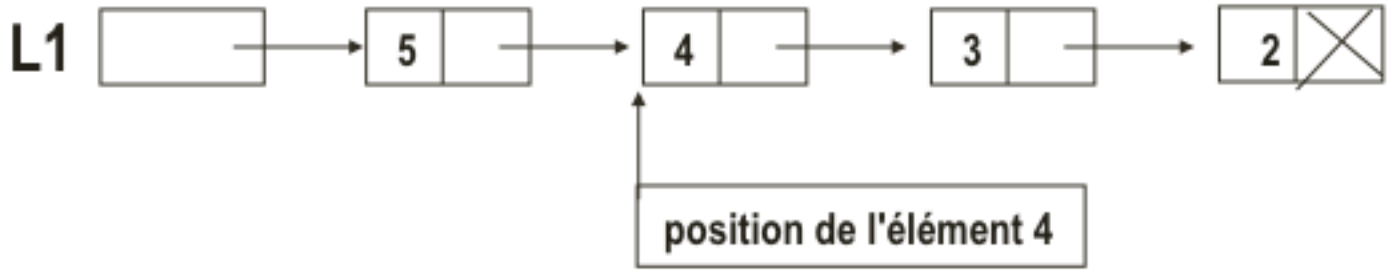
```
ElementAffecter(&cell->element, x) ;
cell->suivant = p ->suivant ;
```

3. accrocher la cellule créer à la liste : modifier le pointeur L1 qui doit donc être passé par adresse

```
* ptrl = cell ;
```

- le prototype est donc
  - `bool ListeInserer(ELEMENT x, POSITION p, LISTE * ptrl)`

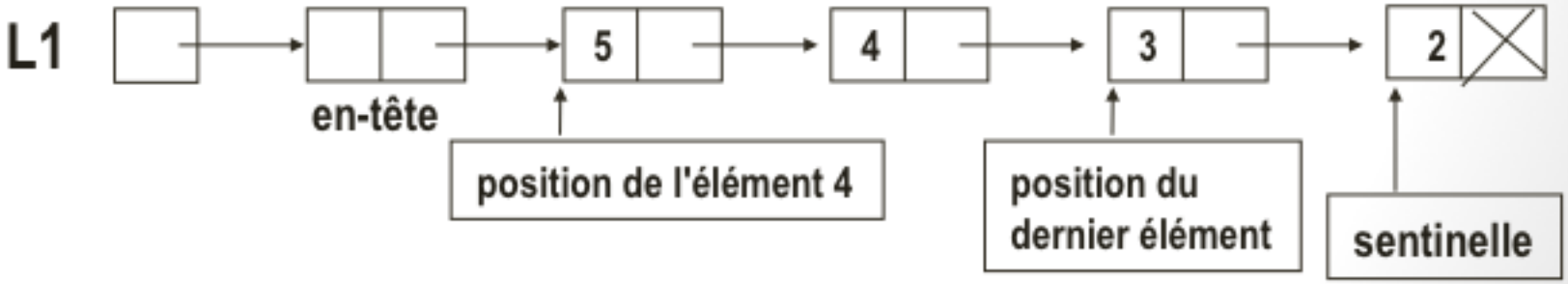
Listes simplement chaînées sans en-tête



- **Problème 2** : pour supprimer l'élément à la position p il faut d'abord trouver le précédent (recherche en **o(n)** dans le pire des cas) avant de pouvoir raccrocher la cellule suivante

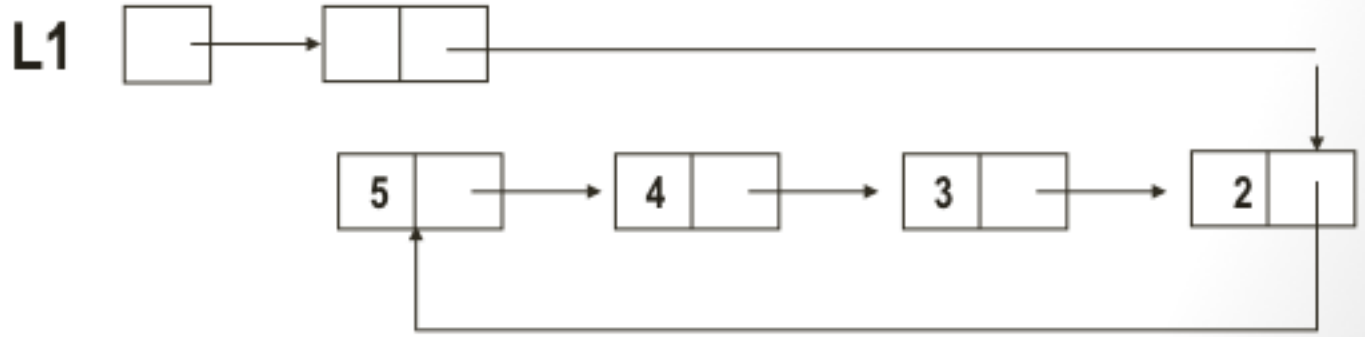
Listes simplement chaînées AVEC en-tête

- Une cellule d'en-tête ne contient pas d'élément et pointe sur la cellule qui contient le premier élément de la liste
- LISTE : pointeur sur la cellule d'en-tête
- POSITION : pointeur sur la cellule qui précède celle qui contient l'élément
- Liste vide : ne contient que sa cellule d'en-tête
- Avantages : les insertions et suppressions en tête sont en o(1)
- **Attention** : Regarder de près cette réalisation qui constitue une excellente révision de la manipulation de pointeurs



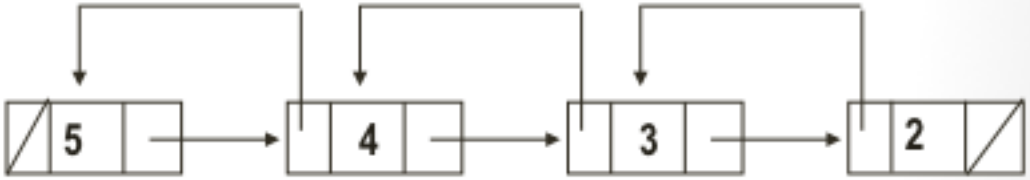
Listes simplement chaînées circulaires

- Une liste est un pointeur sur une cellule d'en-tête
  - La cellule d'en-tête pointe sur la cellule qui contient le dernier élément et celle-ci pointe sur le premier élément de la liste
- Avantages
  - Toutes les opérations sont en o(1) : insertion au début et à la fin, suppression au début, sauf la suppression en fin de liste (et évidemment la recherche linéaire...) en o(n).
- Représentation utilisée pour représenter des files (cf. TD)
  - on entre à la fin (en o(1)) et on sort au début (en o(1) aussi).



Listes doublement chaînées

- Pour parcourir facilement la liste dans les deux sens, on utilise des cellules qui gèrent deux pointeurs :
  - Un pointeur avant : champ précédent de la cellule
  - Un pointeur arrière : champ suivant de la cellule
- Les listes doublement chaînées peuvent être
  - Avec ou sans en-tête
  - Circulaires ou non



- avantages :
  - parcours dans les deux sens, précédent en o(1)
  - insérer avant et après une position
- inconvénients :
  - gérer deux pointeurs c'est pénible et en plus ça occupe de la place

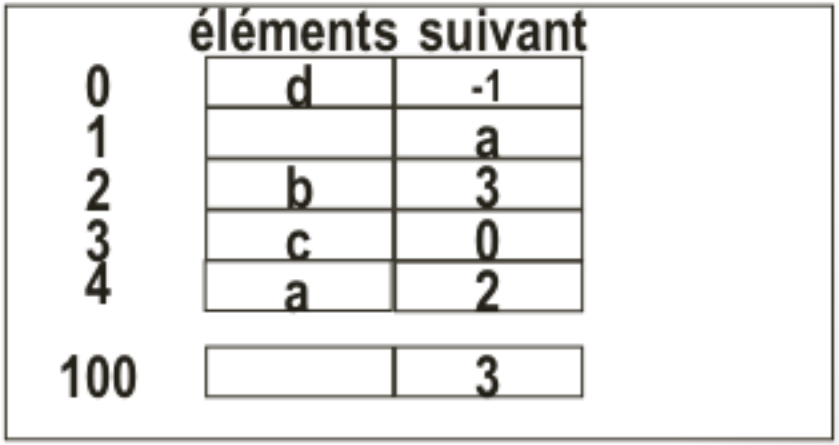
Mise en œuvre des listes par curseurs (faux-pointeurs)

- **Idée** : simuler les pointeurs avec un tableau qui peut contenir plusieurs listes
  - On dispose d'un grand tableau (simulant la mémoire, le tas) de cellules dont le premier champ contient l'élément et le deuxième champ l'indice dans le tableau du suivant
  - Une liste est alors l'indice du premier élément de la liste
  - Nécessité de gérer les cellules disponibles

(a b c d) est représentée par

l = 4

Mémoire



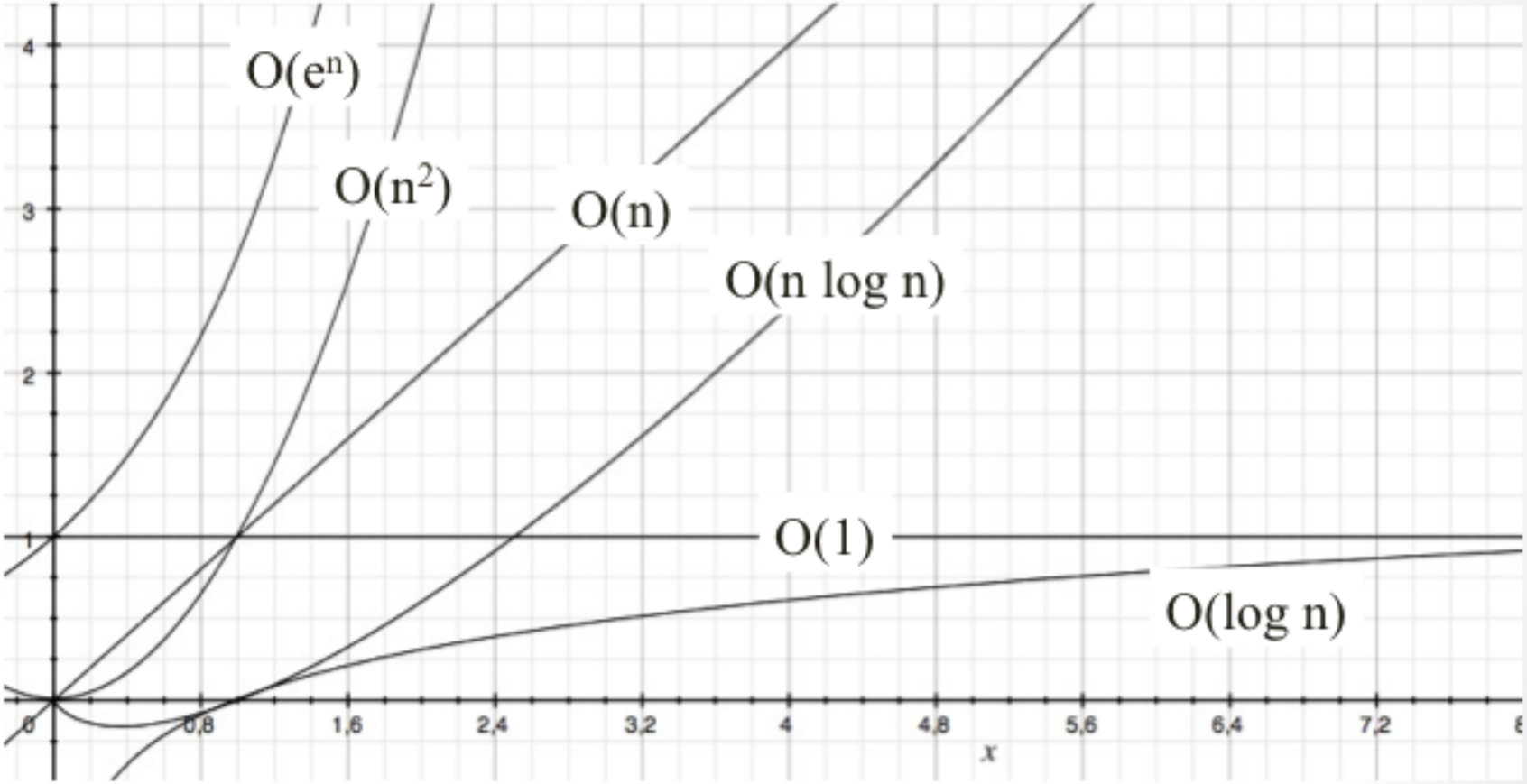
Cours n°1 : LE TDA LISTE

- Structures de données et algorithmes en C
- Cours n°1 : le TDA LISTE
    1. Qualités d'un logiciel
    2. Les types de données abstraits
    3. Le TDA LISTE (TDA)
    4. Opérations fondamentales sur les listes
    5. Algorithmes abstraits sur les listes
    6. Réalisations du TDA LISTE
    7. **Comparaison des réalisations**

Complexité des Algorithmes

- Pour étudier a priori l'efficacité d'un algorithme, on utilise les mesures de complexité.
  - **o(1)** : temps d'exécution constant (indépendant du nombre d'éléments de la liste)
  - **o(log n)** : complexité logarithmique(temps d'exécution proportionnel au log du nombre d'éléments de la liste)
  - **o(n)** : complexité linéaire (temps d'exécution proportionnel au nombre d'éléments de la liste)
  - **o(n log n)** : complexité de l'ordre de n log n
  - **o(n^2)** : complexité quadratique
  - **o(e^n)** : complexité exponentielle

Complexité des Algorithmes



Cellules contiguës

- Avantages
  - Simples à programmer
  - Facilite les opérations insérer en fin, longueur, précédent, sentinelle en o(1)
  - Intéressant si les listes ont une taille qui varie peu
- Inconvénients
  - Nécessite de connaître à l'avance la taille maximum de la liste
  - Coûteux en mémoire si on a des listes de taille très variable
  - Rend coûteuses les opérations d'insertion et de suppression (en o(n))

Cellules chaînées

- Avantages
  - Économise la mémoire pour des listes de taille très variable
  - Facilite les opérations insérer et supprimer en o(1)
  - Intéressant pour les listes où on fait beaucoup d'insertions et de suppressions
- Inconvénients
  - Risque de mauvaise manipulation sur les pointeurs (contrôle de la validité des positions qui augmente la complexité ou fragilise le TDA)
  - Coûteux en mémoire si beaucoup de grosses listes (un pointeur par cellules)
  - Rend coûteuses les opérations longueur, précédent, sentinelle, insérer en fin (en o(n))

Tableau récapitulatif

	listes chaînées avec en-tête	tableau
insérer/ supprimer à la position p	o(1)	o(n)
sentinelle, insérer en fin, longueur, précédent	o(n)	o(1)
accéder, premier, suivant	o(1)	o(1)
localiser, afficher, copier	o(n)	o(n)