Huitième partie

Les structures

Notion de structure

- *Il est habituel en programmation d'avoir besoin d'un mécanisme permettant de regrouper un certain nombre de variables.*
- *On a déjà vu que les tableaux permettaient de regrouper plusieurs éléments. Toutefois, ceux-ci devaient être tous du même type.*
- *Il est parfois nécessaire de regrouper des variables de types différents.*

Notion de structure

- ***Par exemple, pour créer un fichier de personnes on aurait besoin d'une structure de la forme :***
  - ***NOM*** *: chaîne de caractères*
  - ***AGE*** *: entier*
  - ***TAILLE*** *: réel*
- ***La solution est le concept de structure qui est un ensemble d'éléments de types différents repérés par un nom.***
- ***Les éléments d'une structure sont appelés membres ou champs de la structure.***

Déclaration des structures

- *Syntaxe :*

```
struct{
   liste des membres
}
```

- *Exemple :*

```
struct{
    char *Nom;
    int Age;
    float Taille;
} p1,p2;
```

Déclaration des structures

- ***La déclaration précédente déclare deux variables de noms p1 et p2 comme deux structures contenant trois membres.***
- ***L'espace mémoire nécessaire a été réservé pour que les deux variables contiennent chacune trois membres.***

p1	
Nom	char *
Age	int
Taille	float

p2	
Nom	char *
Age	int
Taille	float

Déclaration des structures

- *L'inconvénient de cette méthode est qu'il ne sera plus possible de déclarer d'autres variables du même type.*
- *Si nous déclarons ensuite :*

```
 struct{
    char *Nom;
    int Age;
    float Taille;
} p3;
```

- *p3 ne sera pas considérée du même type, il sera impossible en particulier d'écrire p1 = p3;.*

Déclaration des structures

- *Heureusement, il est possible de définir des modèle de structure.*
- *Syntaxe :*

```
struct identificateur{
  liste des membres
}
```

- *Exemple :*

```
struct personne{
   char *Nom;
   int Age;
   float Taille;
};
```

Déclaration des structures

- *Une telle déclaration n'engendre pas de réservation mémoire.*
- *Le modèle ainsi défini peut être utilisé dans une déclaration de variable.*
- *Exemple :*
  struct personne Michel, Anne;
  / Déclare deux variables de type "struct personne" */*

Déclaration des structures

- ***Dans une structure, tous les noms de champs doivent être distincts.***
- ***Par contre rien n'empêche d'avoir 2 structures avec des noms de champs en commun : l'ambiguïté sera levée par la présence du nom de la structure concernée.***

```
struct personne{
   char *Nom;
   int Age;
   float Taille;
}
```

```
struct pays{
   char *Nom;
   int nb_habitants;
   int superficie;
}
```

Initialisation de structures

- *Une déclaration de structure peut contenir une initialisation par une liste d'expressions constantes à la manière des initialisations de tableau.*
- *La valeur initiale est constituée d'une liste d'éléments initiaux (1 par champ) placée entre accolades.*
- *Exemple :*
  struct personne p = {"Dupond", 25, 1.80};

Accès à un champ

- *L'opérateur d'accès est le symbole "." (point) placé entre l'identificateur de la structure et l'identificateur du champ désigné.*
- *Syntaxe :*

  <ident_objet_struct>.<ident_champ>
- *Exemples :*

```
p1.Age = 25;
if(p2.Nom[0]=='D')
...
```

Affectation des structures

- *On peut affecter l'intégralité des valeurs d'une structure à une seconde structure ayant impérativement le même type.*
- *Exemple :*

```
struct personne p1,p2;
p1 = p2;
```

- *L'affectation d'une structure recopie tous les champs un à un.*
- *Attention la recopie est superficielle.*

Tableau de structures

- *Une déclaration de tableau de structures se fait selon le même modèle que la déclaration d'un tableau dont les éléments sont de type simple.*
- *Pour déclarer un tableau de 100 structures personne, on écrira :*
  ```
  struct personne t[100];
  ```
- *Pour référencer le nom de la personne qui a l'index i dans t, on écrira :*
  ```
  t[i].Nom = "Dupond";
  ```

Structures composées

- *Puisqu'une structure définit un type, ce type peut être utilisé dans une déclaration de variable mais aussi dans la déclaration d'une autre structure comme type de l'un de ses champs.*
- *Exemple :*

```
struct Date
{
   int Jour, Mois, Annee;
};
struct personne
{
   char *Nom;
   struct Date Naissance;
};
```

Structures composées

- *Exemple d'initialisation :*

```
struct personne p = {"Dupond",
{21,05,1980}};
```

- *Exemple d'accès aux membres :*

```
if (p.Naissance.Annee < 1985)
    if(p.Naissance.Mois == 5)
```

Pointeurs vers une structure

- *Supposons la structure personne déclarée, nous pouvons déclarer une variable de type pointeur vers cette structure de la façon suivante :*
  ```
  struct personne *p;
  ```
- *Nous pouvons alors affecter à p des adresses de struct personne.*

Pointeurs vers une structure

Exemple :

```
struct personne
{
    char *Nom;
    int Age;
};

int main()
{
  struct personne pers;
  struct personne *p;

  p = &pers;
}
```

Accès aux champs d'une structure pointée

- *Soit p un pointeur vers une structure personne, *p désigne la structure pointée par p.*
- *Mais *p.Nom ne référence pas le champ Nom de la structure car l'opérateur d'accès aux champs a une priorité supérieure à l'opérateur d'indirection.*
- **p.Nom est équivalent à *(p.Nom)*
- *Il faut donc écrire (*p).Nom pour forcer l'indirection à se faire avant l'accès au champ.*

Accès aux champs d'une structure pointée

- *Pour alléger la notation, le langage C a prévu un nouvel opérateur noté -> qui réalise à la fois l'indirection et la sélection.*
- *p->Nom est équivalent à (*p).Nom*
- *Exemple :*

```
struct personne pers;
struct personne *p;

p = &pers;
p->Nom="Dupont";
```

Champ pointant vers une structure

Exemple :

```
struct Date
{
    int Jour, Mois, Annee;
};

struct personne
{
    char *Nom;
    struct Date *Naissance;
};
```

Champ pointant vers une structure

- *Accès aux champs :*

```
pers.Nom = "Dupont";
pers.Naissance->Jour = 21;
pers.Naissance->Mois = 05;
pers.Naissance->Annee = 1985;
```

- *Si p est un pointeur sur pers :*

```
p->Nom = "Dupont";
p->Naissance->Jour = 21;
p->Naissance->Mois = 05;
p->Naissance->Annee = 1985;
```

Structures récursives

- *Ce genre de structures est fondamental en programmation car il permet d'implémenter la plupart des structures de données employées en informatique (listes, files, arbres, etc...).*
- *Ces structures contiennent un ou plusieurs champs du type pointeur vers une structure du même type et non de simples champs du type structure.*

Structures récursives

- *Exemple :*

```
struct personne
{
  char *Nom;
  int Age;
  struct personne *Pere, *Mere;
};
```

- *Déclaration incorrecte car la taille de Pere et Mere n'est pas connue :*

```
struct personne
{
  char *Nom;
  int Age;
  struct personne Pere, Mere;
};
```

Passage comme argument

- *La norme ANSI a introduit le transfert direct d'une structure entière comme argument d'une fonction, ainsi que la remontée de cette structure depuis une fonction appelée via une instruction return.*
- *Le passage de paramètres de type structure se fait par valeur.*
- *Pour les compilateurs antérieur à la norme ANSI, comme le C K&R, il n'est pas possible de passer en paramètre une structure, il faut utiliser un pointeur sur cette structure.*

Passage comme argument

```
struct personne
{
  int jour, mois, annee;
}

int cmp_date(struct date d1, struct date d2)
{
  if (d1.annee > d2.annee)
            return (APRES);
  if(d1.annee < d2.annee)
            return (AVANT);
… /* comparaison portant sur mois et jour */

}
int main()
{
  struct date1, date2;

  if(cmp_date(date1,date2)==AVANT)
     …
}
```

typedef

- *Pour simplifier les écritures et éviter de répéter systématiquement struct personne, on utilise typedef pour définir les structure ou union comme des nouveaux types.*
- ***Exemple :***

```
typedef struct personne
{
    char * Nom;
    int Age;
} Personne;

int main()
{
  Personne p;
  …
}
```

Structures de données élémentaires

Introduction

Dans cette partie, nous allons voir quelques structures utilisées de façon très intensive en programmation.

Leur but est de gérer un ensemble fini d'éléments dont le nombre n'est pas fixé a priori.

Les éléments de cet ensemble peuvent être de différentes sortes:

nombres entiers ou réels ;

chaînes de caractères ;

ou des objets informatiques plus complexes.

Introduction

On ne s'intéressera pas aux éléments de l'ensemble en question mais aux opérations que l'on effectue sur cet ensemble, indépendamment de la nature de ses éléments.

Les ensembles que l'on utilise en programmation sont des objets dynamiques.

Le nombre de leurs éléments varie au cours de l'exécution du programme, puisqu'on peut y ajouter et supprimer des éléments en cours de traitement.

Introduction

Les opérations minimales agissant sur ces ensembles sont les suivantes :

tester si l'ensemble E est vide.

ajouter l'élément x à l'ensemble E.

vérifier si l'élément x appartient à l'ensemble E.

supprimer l'élément x de l'ensemble E.

Introduction

Les Listes

Les Files

Les Piles

Les Arbres

Les listes chaînées

La liste est une structure de base de la programmation

Le langage LISP (LISt Processing), conçu par John MacCarthy en 1960, utilise principalement cette structure qui se révèle utile pour le calcul symbolique.

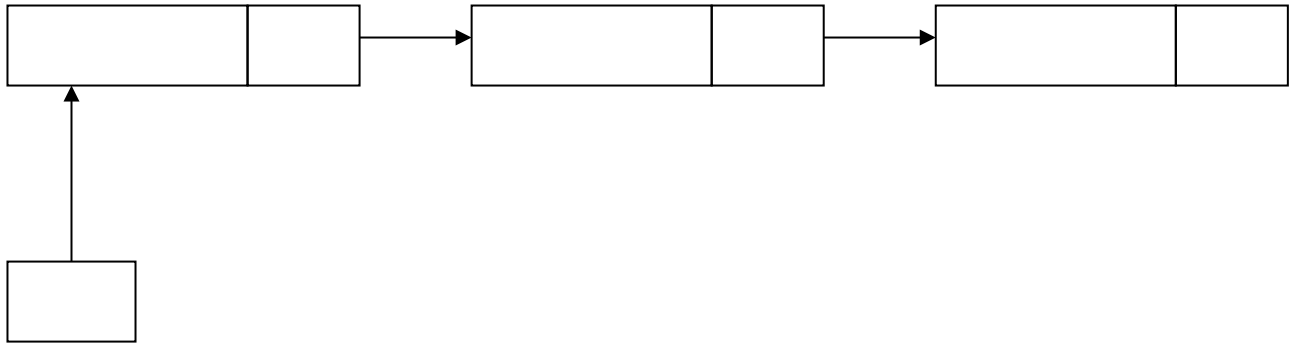
Une liste chaînée est un regroupement ordonné de données effectué de manière à ce que chaque composante (ou cellule) sache où se trouve la suivante.

Les listes chaînées

Chaque cellule contient un élément et un lien vers la cellule suivante.

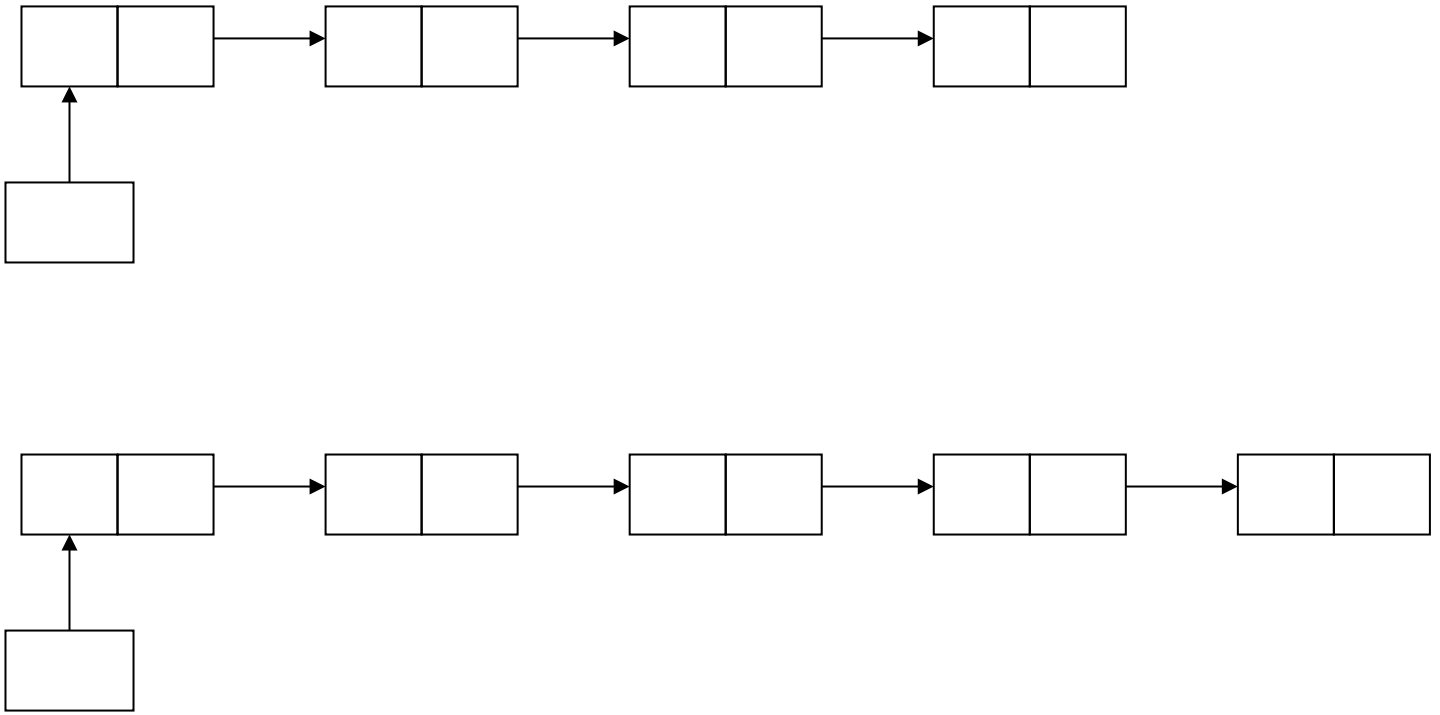
La référence vers la première cellule est contenue dans une variable de tête de liste.

Pour rechercher un élément de la liste, il faut parcourir les cellules depuis la tête de la liste.



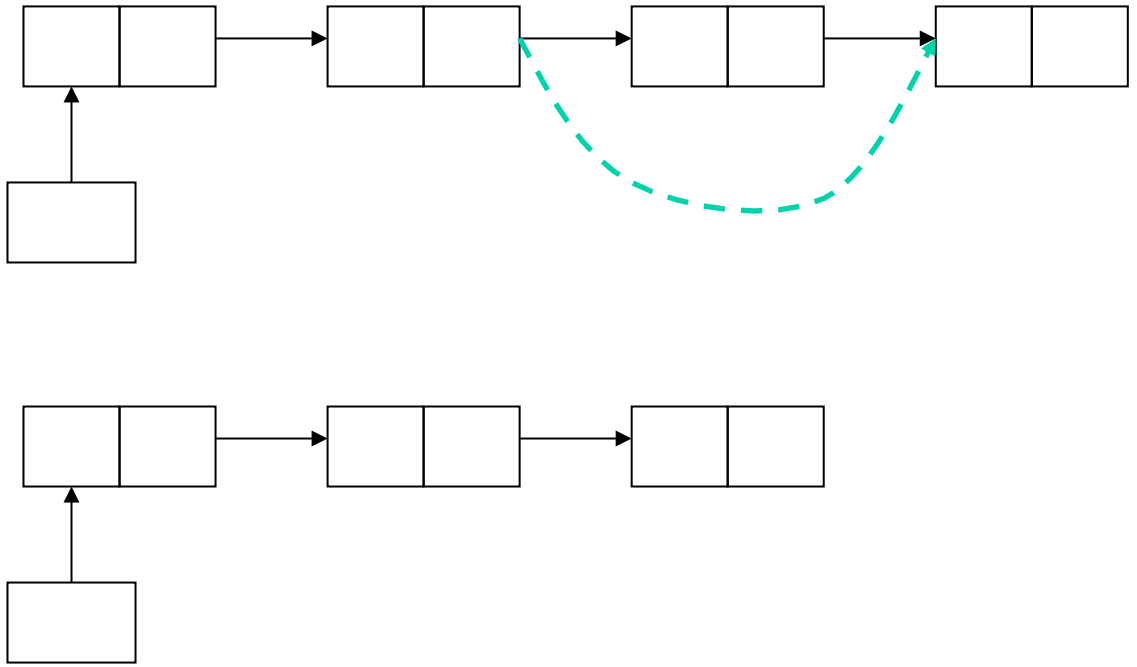
Les listes chaînées

Insertion d'une cellule en tête de liste :



Les listes chaînées

Suppression d'un élément de la liste :



Les listes chaînées

Les cellules sont généralement des structures contenant l'élément et le lien sur la cellule suivante.

Les listes peuvent être représentées de différentes manières :

sous forme de tableau, le lien est alors un entier représentant l'indice de la cellule suivante ;

avec les pointeurs, le lien est un pointeur désignant l'adresse de la cellule suivante.

Les files

- ***Les files sont utilisées en programmation pour gérer des objets qui sont en attente d'un traitement ultérieur, par exemple :***
  - *des processus en attente d'une ressource du système ;*
  - *des sommets d'un graphe ;*
  - *des nombres entiers en cours d'examen de certaines de leur propriétés ;*
  - *etc ...*

Les files

- *Dans une file les éléments sont systématiquement ajoutés en queue et supprimés en tête, la valeur d'une file est par convention celle de l'élément de tête.*
- *En anglais, on parle de stratégie FIFO (First In First Out), le premier élément entré sera le premier sortie.*

Les files

- ***Les opérations sur les files sont estvide, ajouter, valeur, supprimer:***

  estVide(F) *est vrai si et seulement si la file F est vide*

  ajouter(x,F) *est la file obtenue à partir de la file F en insérant l'élément x à la fin de celle-ci.*

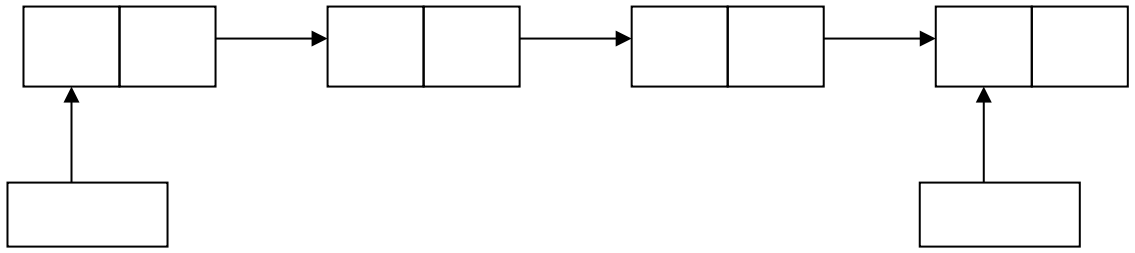
  valeur(F) *appliquée à une file non vide donne l'élément se trouvant en tête.*

  supprimer(F) *supprime le premier élément de la file F.*

Les files

- ***Deux réalisations possibles des files : avec des tableaux ou des listes chaînées.***
  - *L'écriture de programmes consiste à faire des choix pour représenter les structures de données.*
  - *L'ensemble des fonctions sur les files peut être indistinctement un module manipulant des tableaux ou un module manipulant des listes.*

  L'utilisation des files se fait uniquement à travers les fonctions estvide, ajouter, valeur, supprimer.

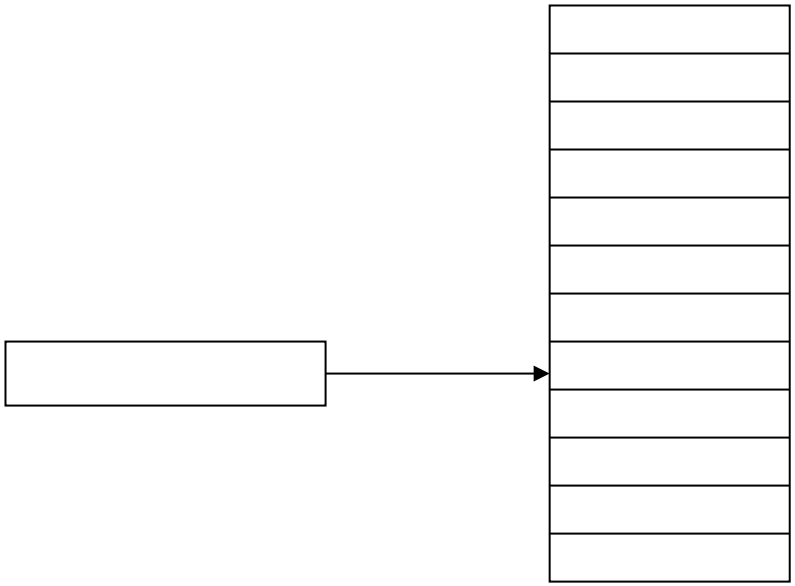


Les piles

- *La notion de pile intervient couramment en programmation, son rôle principal consiste à implémenter les appels de procédures.*
- *LIFO (Last In First Out), le dernier élément entré sera le premier sortie.*
- *Les opérations sur les piles sont estvide, ajouter, valeur, supprimer comme sur les files.*

Les piles

La réalisation des opérations sur les piles peut s'effectuer aussi bien en utilisant un tableau qui contient les éléments et un indice qui indiquera la position du sommet de la pile, qu'en utilisant une liste chaînée.



Les arbres

Les listes, les files et les piles sont des structures dynamiques unidimensionnelles.

Les arbres sont leur généralisation multidimensionnelle.

Chaque composante d'un arbre (un nœud) contient un élément et des liens vers ses fils.

Tous les nœuds ont un et un seul père sauf la racine qui n'en a pas.

Une feuille est un nœud qui n'a pas de fils.

Les arbres

Un arbre N-aire ne contient que des nœuds à N fils.

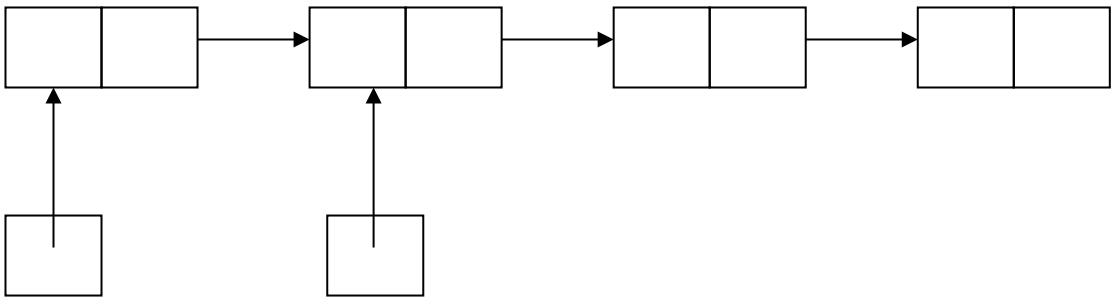
Une liste est un arbre dont chaque nœud a un et un seul fils. Une liste est donc un arbre unaire.

Comme pour les listes, un arbre est identifié par le premier de ces nœuds c'est-à-dire la racine.

De nombreux algorithmes ont été développé pour les arbres binaires (tout arbre peut être représenté par un arbre binaire).

Opérations courantes sur les listes

La fonction Tail est une primitive classique, elle supprime le premier élément.



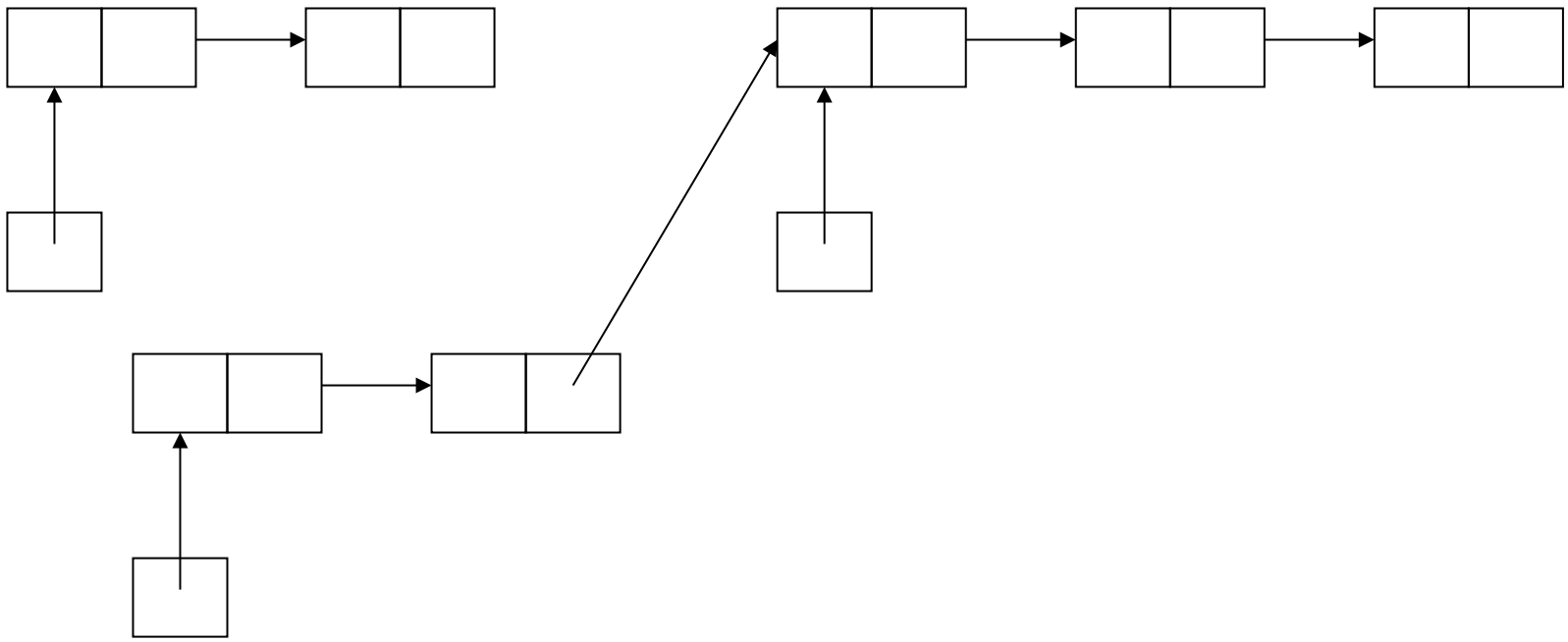
Opérations courantes sur les listes

- *Des procédures sur les listes construisent une liste à partir de deux autres.*
- *Il s'agit de mettre deux listes bout à bout pour en construire une dont la longueur est égale à la somme des longueurs des deux autres.*

Opérations courantes sur les listes

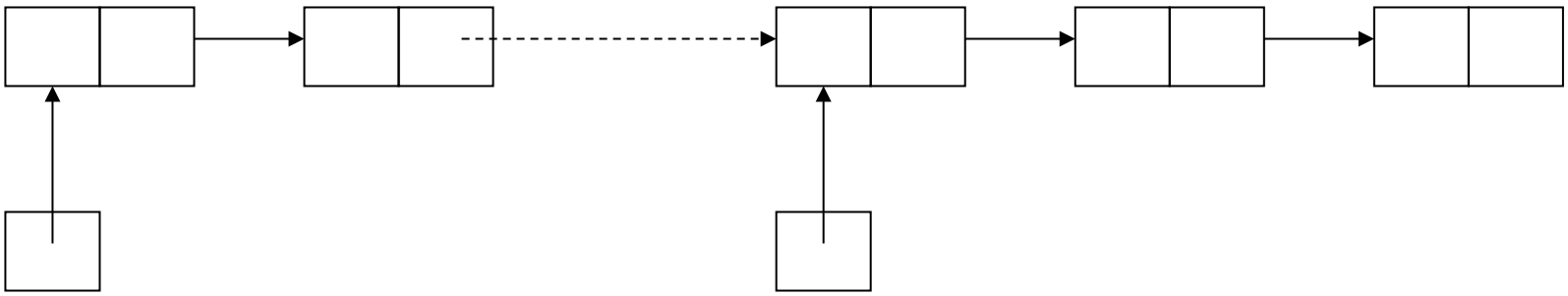
Dans la première procédure append, les deux listes ne sont pas modifiées.

Toutefois, on remarquera que, si append copie son premier argument, il partage la fin de liste de son résultat avec son deuxième argument



Opérations courantes sur les listes

Dans la seconde nConc, la première liste est transformée pour donner le résultat...



Opérations classiques sur les listes

opérations d'allocation, libération de mémoire

ListeCreer, ListeDetruire

opérations de gestion

ListeAccèder, ListeInsérer, ListeSupprimer

opérations de parcours

ListePremier, ListeSuivant

opération de test

ListeVide

Opérations classiques sur les listes

ListeCréer : 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

ListeAccèder : 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 p est erronée)

Opérations classiques sur les listes

ListeInsérer : 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

ListeSupprimer : deux 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 : deux 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)

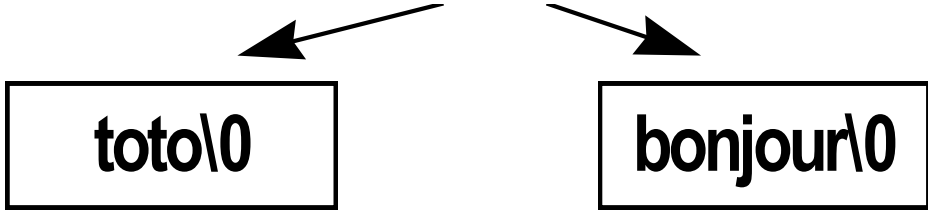
Eléments contenus dans les listes

Un élément de la liste peut être

- *un objet (stockage direct)*
- *une adresse (stockage indirect)*

Exemples:

- *stockage direct : l = (1 2 3 4)*
- *stockage indirect : l = (ad1 ad2)*



Différentes mises en œuvre des listes

Concretement, le concept de liste peut être implémenté de différentes manières :

Listes simplement chaînées sans en-tête

Listes simplement chaînées avec en-tête

Listes doublement chaînées

Listes circulaires

Listes par cellules contiguës

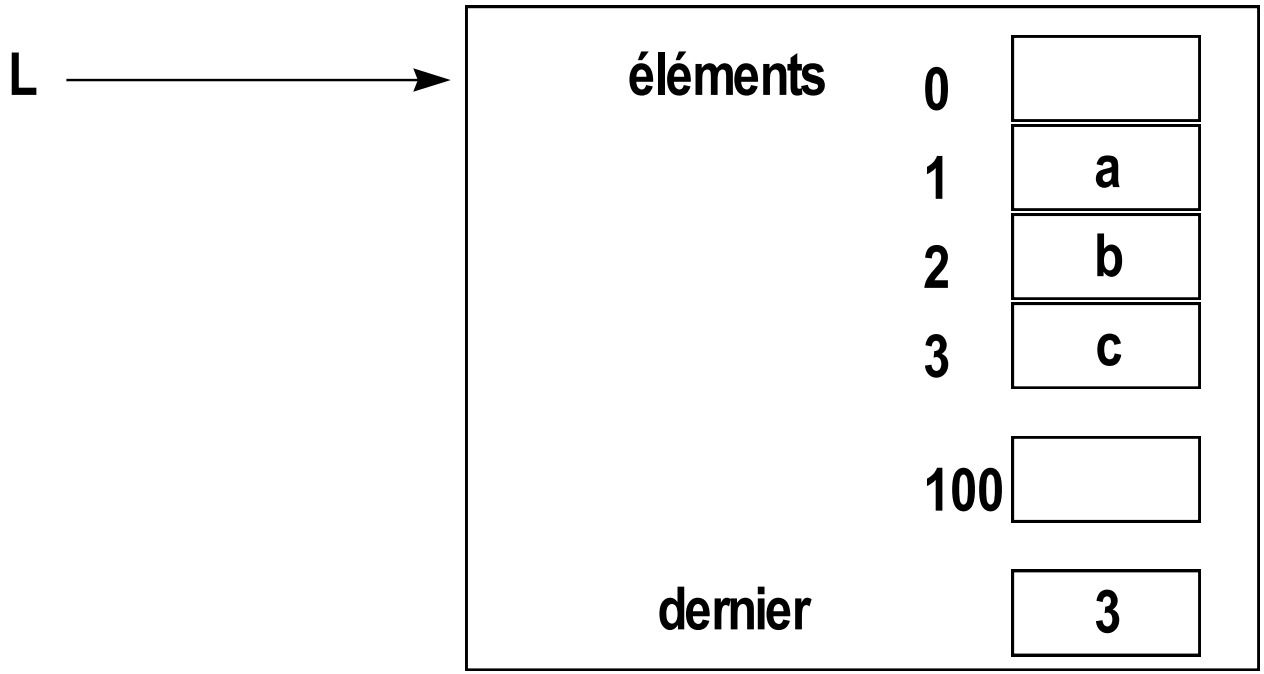
Listes par curseurs (faux-pointeurs)

Mise en œuvre des listes par cellules contiguës

plusieurs représentations sont possibles nous choisissons : une liste est un pointeur sur une structure à deux champs

un tableau qui contient les éléments de la liste

un entier indiquant l'indice du dernier élément de la liste



Mise en œuvre des listes par cellules contiguës

Analyse :

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 (ici 4)

la liste est vide si le dernier élément du tableau est 0

quand la liste est vide :

la position du premier élément de la liste est 1

Mise en œuvre des listes par cellules contiguës

Primitives

ListeCreer : 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.

ListeDétruire : il suffit d'appeler free sur la liste L

*Les **fonctions de parcours et d'accès** sont immédiates*

ListeInserer : pour insérer à la position p il faut commencer par décaler à partir du dernier élément jusqu'au $p^{ème}$; puis on insère dans la $p^{ème}$ 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

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 de décalages

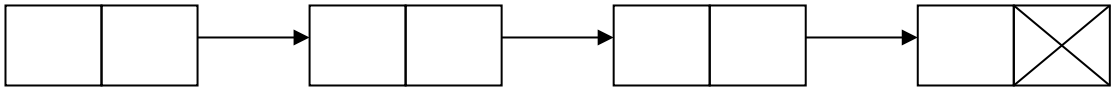
toutes les autres opération sont o(1) (en temps constant)

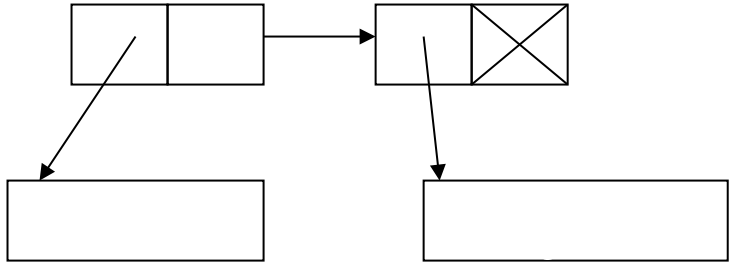
insérer et supprimer des éléments en fin de liste est en o(1) puisque dans ce cas, on a pas besoin de décaler.

Mise en œuvre des listes par cellules chaînées

Une cellule est composée de deux champs :

- *élément, contenant un élément de la liste*
- *suivant, contenant un pointeur sur une autre cellule*





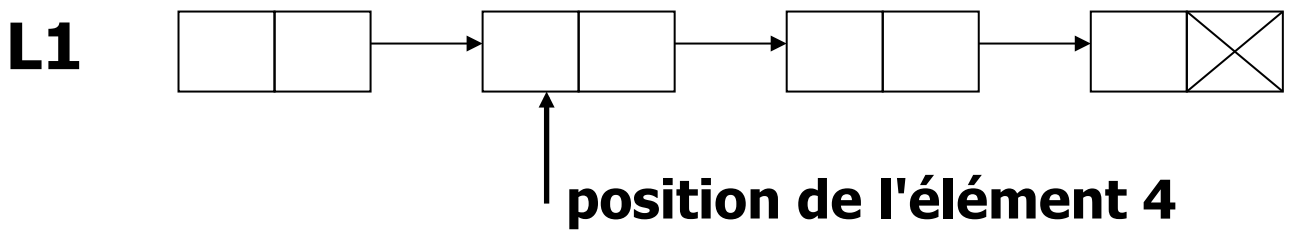
Mise en œuvre des listes par cellules chaînées

- *Les cellules ne sont pas rangés 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, *LISTE;
```

Listes simplement chaînées sans en-tête

- *Une liste est un pointeur sur la cellule qui contient le premier élément de la liste*
- *La position d'un élément est le pointeur sur la cellule qui contient cet élément*
- *La liste vide est le pointeur NULL*



Représentation des listes chaînées sans entête

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 10

   typedef struct noeud *ptrNoeud;
   typedef struct nœud
   {
      int valeur ;
      ptrNoeud suivant;
   }
   Noeud ;

   typedef ptrNoeud Liste ;
```

Représentation des listes chaînées sans entête

```
int valeur(Liste l)
{
      return (l->valeur);
}
int estVide(Liste l)
{
     return (l==NULL);
}
```

Représentation des listes chaînées sans entête

```
void afficher (Liste l)
{
      int i=0;
      printf("\n");
      if (l)
      {
         do
         {
            printf("Elément %d : %d\n",++i,l->valeur);
            l=l->suivant;
         }
         while (l);
      }
      else printf("La liste est vide");
 }
```

Représentation des listes chaînées sans entête

```
void supprimerPrem(Liste *l)
{
      ptrNoeud aSupp=*l;
      *l=(*l)->suivant;
      printf("Je détruis:%d\n",(aSupp)->valeur);
      free(aSupp);
 }
```

Représentation des listes chaînées sans entête

```
void ajouterFin (int v, Liste *l)
{
      ptrNoeud nouv, courant;
      if (nouv=(ptrNoeud) malloc (sizeof(Noeud)))
      {
         nouv->valeur=v;
         nouv->suivant=NULL;
      /* Premier cas L est vide */
         if (!*l)
            *l=nouv;
         else
         {
            courant=*l;
            while (courant->suivant)
              courant=courant->suivant;
              courant->suivant=nouv;
         }
      }
      else
       printf("Erreur d'allocation");
}
```

Représentation des listes chaînées sans entête

```
int main ()
{
      Liste l1 ;
      int i;
      for (i=0; i<10; i++)
         ajouterFin(i,&l1);
      afficher(l1);

      while (!estVide(l1))
      {
         int x=valeur (l1);
         printf("%d, ",x);
         supprimerPrem(&l1);
      }
      afficher(l1);
      printf("\n");
      return 0;
}
```

Listes simplement chaînées sans en-tête

- ***Deux inconvénients majeurs pour les listes chaînées sans en-tête :***
  - *problèmes lors de l'insertion ou de la suppression d'un élément au début de la liste*
  - *suppression d'un élément en o(n)*

Listes simplement chaînées sans en-tête

- ***Problème 1 : quand on veut insérer ou supprimer au début de la liste, ce n'est pas pareil qu'au milieu : donc à chaque fois il faut tester (si p = Premier(L)...)***
- ***De plus :***
  - *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) aïe aïe_ les bugs prévisibles*

Listes simplement chaînées sans en-tête

- ***Exemple : insérer 42 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 42 et un pointeur sur la première cellule de la liste L1***

  cell->element = 42 ;

  cell->suivant = p ->suivant ;

  3/ ***accrocher la cellule créée à la liste : modifier le pointeur L1 qui doit donc être passé par adresse***

  *int ListeInserer(ELEMENT x, POSITION p, LISTE * ptrl)*

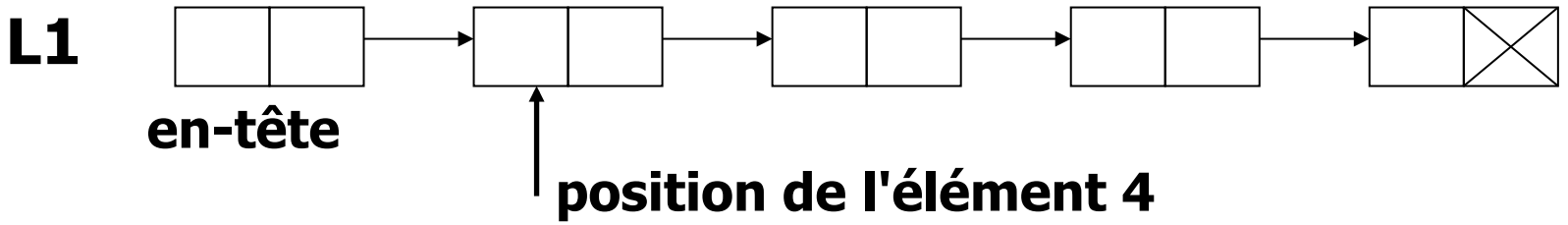
  ** ptrl = &cell ;*

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

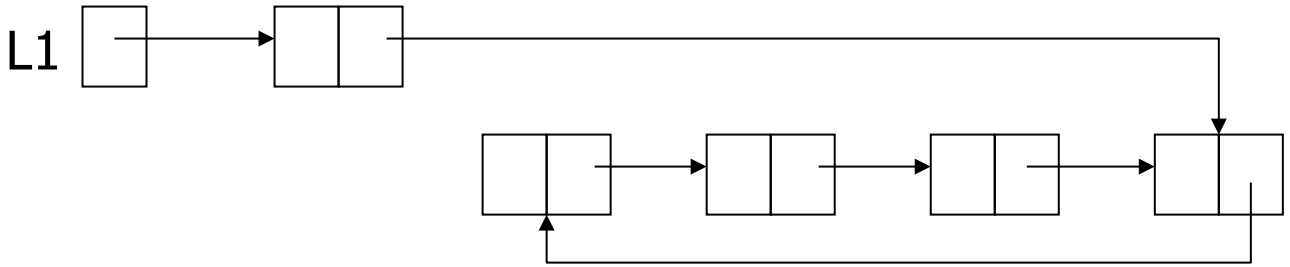
Listes simplement chaînées avec en-tête

- *Une cellule d'en-tête ne contient pas d'élément et pointe sur la cellule qui contient le premier élément de la liste*
- *Une liste est un pointeur sur la cellule d'en-tête*
- *La position d'un élément est un pointeur sur la cellule qui précède celle qui contient l'élément*
- *Une liste vide ne contient que sa cellule d'en-tête*
- *Avantages : les insertions et suppressions sont en o(1)*



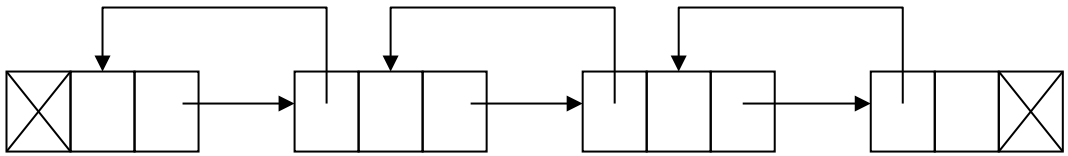
Les listes simplement chaînées circulaires

- ***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. représentation utilisée pour représenter des files on entre à la fin (en o(1)) et on sort au début (en o(1) aussi)*
  - *sauf suppression en fin de liste (et évidemment la recherche linéaire...) en o(n)*



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*



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*
- *Nécessite de gérer les cellules disponibles*

Listes par curseurs (faux-pointeurs)

(a b c d) est représentée par

l = 4

	éléments	suivant
0	d	-1
1		a
2	b	3
3	c	0
4	a	2
100		3

Discussion

Réalisations par 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))

Discussion

Réalisations par cellules chaînées

- *Avantages*
  - ♦ économise la mémoire pour des listes de taille très variables
  - ♦ facilite les opérations insérer et supprimer en o(1)
  - ♦ intéressant pour les listes où on fait beaucoup d'insertion et de suppression
- *Inconvénients*
  - ♦ risque de mauvaise manipulation sur les pointeurs (contrôle de la validité des positions qui augmente la complexité)
  - ♦ 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))