# Structure dynamique de données

## I3 - Algorithmique et programmation

Nicolas Delestre, Nicolas Malandain

# Plan...



Architecture des Systèmes d'Information

# Compilation et édition des liens 1 / 2

## Définitions (Wikipédia)

« *Un* **compilateur** *est un programme informatique qui traduit un langage (appelé le langage source) en un autre (le langage cible), généralement dans le but de créer un exécutable.*

*Un compilateur sert le plus souvent à traduire un code source écrit dans un langage de programmation en un autre langage, habituellement un langage d'assemblage ou un langage machine. Le programme en langage machine produit par un compilateur est appelé code objet.* »

« *Lors d'un développement informatique, l'***édition des liens** *est un processus qui permet de créer des fichiers exécutables ou des bibliothèques dynamiques ou statiques, à partir de fichiers objets.* »

Architecture des Systèmes d'Information

# Compilation et édition des liens 2 / 2

## En complément

- Un compilateur permet souvent de changer de paradigme
- Un compilateur ajoute du code non présent dans le programme source

# Mémoire et allocation 1 / 2

## Allocation de mémoire

- Les entités utilisées (variables, constantes, (sous-)programmes) par le programme source sont représentées en mémoire (RAM de l'ordinateur)
- Il y a une relation directe entre l'identifiant que l'on utilise et un espace mémoire qui stocke l'information correspondante

## Plusieurs emplacements dans la mémoire vive (segment)

statique ou *text* emplacement pour les programmes, sous-programmes

*bss* emplacement pour les variables globales

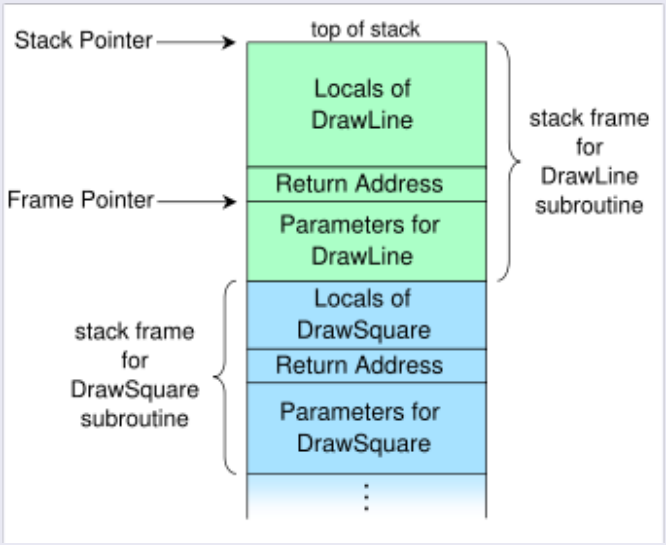
*data* emplacement pour les constantes

tas ou *heap* emplacement où sont stockés des espaces mémoires alloués dynamiquement. La taille du tas varie en fonction de l'exécution du programme

# Mémoire et allocation 2 / 2

## Plusieurs emplacements dans la mémoire vive (segment)

pile ou *stack* emplacement où sont stockées les variables locales et les paramètres formels des sous-programmes. La taille de la pile varie en fonction de l'exécution du programme




Wikipédia

## Registre

- Le microprocesseur possède lui aussi des espaces mémoires, nommés registres, qui sont indépendants de tout programme
  - Les registres sont en nombre limité
  - Les registres servent à faire des calculs
  - Il y a des copies régulières entre la mémoire vive et les registres

# Allocation statique

## Définition

- Allocation de mémoire effectuée lors de l'exécution mais prévue lors de la compilation
  - À chaque type de données correspond une taille mémoire et une façon de représenter l'information
  - À chaque variable ou paramètre formel déclarés correspond un espace mémoire dont la taille est fonction du type
- Le compilateur ajoute donc automatiquement du code pour réserver de l'espace mémoire avant utilisation (au niveau de la déclaration) et pour libérer si besoin est (dans la pile)

# Allocation dynamique

## Définition

- Allocation de mémoire effectuée lors de l'exécution mais non prévue lors de la compilation
  - Allocation qui se fait uniquement dans le tas
  - Cette allocation est à la charge du programmeur, il lui faut donc :
    - une procédure permettant de réserver une zone mémoire (**allouer**() en algorithmique et `new` en Pascal)
    - une procédure permettant de libérer une zone mémoire (**liberer**() en algorithmique et `dispose` en Pascal)
    - une variable (et donc un type) permettant de référencer cette zone mémoire allouée

# Pointeur

## Définition

- On nomme un « pointeur » $p$ une variable permettant de référencer une zone mémoire permettant de stocker une information de type $T$
- Le type de $p$ est nommé « pointeur sur $T$ ». Il est noté (en algorithmique et en pascal) $\hat{}T$
- Lorsqu'une variable ne pointe sur aucune zone mémoire, il faut l'initialiser avec la valeur `NIL`

## Opérateurs sur les pointeurs (en algorithmique et en pascal)

- $\hat{}$ opérateur unaire (opérande à gauche de l'opérateur) permettant de « déréférencer » un pointeur (accéder à la valeur de la zone mémoire pointée)
- @ opérateur unaire (opérande à droite de l'opérateur) permettant d'obtenir un pointeur sur une variable

# Allocation dynamique et pointeur

## Exercice : que fait la procédure `mystere` ?

```
procédure mystere ()
    Déclaration  i : Naturel
                 pi : ^Naturel
debut
    i ← 0
    ecrire(i)
    pi ← @i
    pi^ ← pi^+1
    ecrire(i)
    allouer(pi)
    pi^ ← i
    ecrire(pi^)
    liberer(pi)
    ecrire(pi^)
fin
```

# Structures de données dynamiques 1 / 3

## Contexte

- Lorsque l'on veut stocker en mémoire $n$ fois le même type d'élément, on utilise jusqu'à présent les tableaux
- Les tableaux sont des allocations statiques (la taille du tableau est définie à la compilation), on ne peut pas l'adapter au contexte. Le fait de réserver `MAX` éléments :
  - consomme beaucoup de mémoire si peu d'éléments réellement utilisés ($n << MAX$)
  - pose problème si on a besoin de plus de $MAX$ éléments à stocker ($n > MAX$)
- Il faudrait pouvoir stocker en mémoire autant de données dont on a besoin et pas plus
- Mais ce nombre de données ne peut être déterminé à la compilation, il ne peut être déterminé qu'à l'exécution

C'est le rôle des structures de données dynamiques

# Structures de données dynamiques 2 / 3

## Comment les concevoir ?

- Il faut donc que la mémoire soit réservée à l'exécution ⇒ besoin d'allocations dynamiques
- Mais il faut pouvoir référencer ces allocations dynamiques ⇒ besoins de pointeurs (les pointeurs sont des variables donc allocation statique)
- Ainsi le nombre de pointeurs est fonction du nombre d'éléments que l'on veut stocker, ce qui est contradictoire avec notre objectif
- Il faut donc que les futures espaces mémoires alloués ne soient pas référencés par des variables mais par les espaces mémoires déjà alloués
- Ainsi **les espaces mémoires déjà alloués** stockent l'information à réellement stocker *et* également une référence vers **les autres espaces mémoires alloués ou à allouer** (définition récursive)

# Structures de données dynamiques 3 / 3

## Comment les concevoir ? (suite)

- Une structure dynamique est donc au départ une variable pointeur (allocation statique) qui fait référence à une zone mémoire allouée dynamiquement qui stocke à la fois un élément à réellement stocker et une ou plusieurs références vers le même type de zone mémoire
- Lorsque dans cette zone mémoire, il y a une seule référence, la structure dynamique est une **liste chaînée**
- Il existe d'autres types de structures dynamiques : liste chaînée circulaire, liste doublement chaînée, arbre binaire, etc.

# Liste chaînée

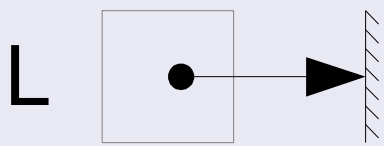
## Définition

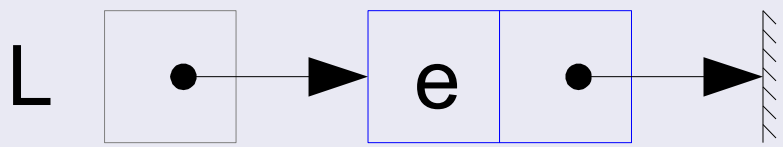
Une liste chaînée est soit :

- une liste vide
- un élément suivi d'une liste chaînée

## Graphiquement

On représente une liste $L$ de la façon suivante :




On voit donc apparaître le concept de *pointeur* et de *noeud*

# Liste chaînée d'entiers 1 / 10

## Conception préliminaire

- **fonction** listeVide **() :** ListeChaineeDEntiers
- **fonction** estVide **(**uneListe : ListeChaineeDEntiers**) : Booleen**
- **procédure** ajouter **(E/S** uneListe : ListeChaineeDEntiers,**E** element : Entier**)**
- **fonction** obtenirEntier **(**uneListe : ListeChaineeDEntiers**) :** Entier
  - ⌊**précondition(s)** *non*(*estVide*(*uneListe*))
- **fonction** obtenirListeSuivante **(**uneListe : ListeChaineeDEntiers**) :** ListeChaineeDEntiers
  - ⌊**précondition(s)** *non*(*estVide*(*uneListe*))
- **procédure** fixerListeSuivante **(E/S** uneListe : ListeChaineeDEntiers,**E** nelleSuite : ListeChaineeDEntiers**)**
  - ⌊**précondition(s)** *non*(*estVide*(*uneListe*))
- **procédure** supprimerTete **(E/S** l :ListeChaineeDEntiers**)**
  - ⌊**précondition(s)** non estVide(l)
- **procédure** supprimer **(E/S** uneListe : ListeChaineeDEntiers**)**

Architecture des Systèmes d'Information

# Liste chaînée d'entiers 2 / 10

## Conception détaillée

**Type** ListeChaineeDEntiers = ^ NoeudDEntier
**Type** NoeudDEntier = **Structure**
    entier : **Entier**
    listeSuivante : ListeChaineeDEntiers
**finstructure**

**fonction** listeVide **() :** ListeChaineeDEntiers
**debut**
    **retourner NIL**
**fin**
**fonction** estVide **(**l :ListeChaineeDEntiers**) : Booleen**
**debut**
    **retourner** l=**NIL**
**fin**
**procédure** ajouter **(E/S** l :ListeChaineeDEntiers,**E** e :**Entier)**
    **Déclaration** temp : ListeChaineeDEntiers
**debut**
    temp ← l
    **allouer**(l)
    l^.entier ← e
    fixerListeSuivante(l,temp)
**fin**

# Liste chaînée d'entiers 3 / 10

## Conception détaillée (suite)

**fonction** obtenirEntier **(**l :ListeChaineeDEntiers**) : Entier**

⌊**précondition(s)** non estVide(l)

**debut**

**retourner** l^.entier

**fin**

**procédure** fixerListeSuivante **(E/S** l :ListeChaineeDEntiers,**E** l' :ListeChaineeDEntiers**)**

⌊**précondition(s)** non estVide(l)

**debut**

l^.listeSuivante ← l'

**fin**

**fonction** obtenirListeSuivante **(**l :ListeChaineeDEntiers**) :** ListeChaineeDEntiers

⌊**précondition(s)** non estVide(l)

**debut**

**retourner** l^.listeSuivante

**fin**

Architecture des Systèmes d'Information

# Liste chaînée d'entiers 4 / 10

## Conception détaillée (fin)

**procédure** supprimerTete **(E/S** l :ListeChaineeDEntiers**)**

⌊**précondition(s)** non estVide(l)

**Déclaration** temp : ListeChaineeDEntiers

**debut**
    temp ← l
    l ← obtenirListeSuivante(l)
    **liberer**(temp)
**fin**
**procédure** supprimer **(E/S** l :ListeChaineeDEntiers**)**
**debut**
    **tant que** non estVide(l) **faire**
        supprimerTete(l)
    **fintantque**
**fin**

# Liste chaînée d'entiers 5 / 10

## Exemple d'utilisation : parcours de liste (version itérative)

```
procédure afficher (E l :ListeChaineeDEntiers)
debut
    tant que non estVide(l) faire
        ecrire(obtenirEntier(l))
        l ← obtenirListeSuivante(l)
    fintantque
fin
```

## Exemple d'utilisation : parcours de liste (version récursive)

```
procédure afficher (E l :ListeChaineeDEntiers)
debut
    si non estVide(l) alors
        ecrire(obtenirEntier(l))
        afficher(obtenirListeSuivante(l))
    finsi
fin
```

Architecture des Systèmes d'Information

# Liste chaînée d'entiers 6 / 10

## Developpement

```
unit ListeChaineeDEntiers;

interface

type
   TListeChaineeDEntiers = ^TNoeudDEntiers;
   TNoeudDEntiers        = record
                             entier        : Integer;
                             listeSuivante : TListeChaineeDEntiers;
                           end;

function listeVide() : TListeChaineeDEntiers;
function estVide(l : TListeChaineeDEntiers) : boolean;
procedure ajouter(var l : TListeChaineeDEntiers; e : Integer);
function obtenirEntier(l :  TListeChaineeDEntiers) : Integer;
function obtenirListeSuivante(l :  TListeChaineeDEntiers) : TListeChaineeDEntiers;
procedure fixerListeSuivante(var l : TListeChaineeDEntiers; ls :
TListeChaineeDEntiers);
procedure supprimerTete(var l :  TListeChaineeDEntiers);
procedure supprimer(var l :  TListeChaineeDEntiers);
```

# Liste chaînée d'entiers 7 / 10

## Developpement (suite)

```
implementation

function listeVide() : TListeChaineeDEntiers;
begin
   listeVide:=NIL
end; { listeVide }

function estVide(l : TListeChaineeDEntiers) : boolean;
begin
   estVide:=l=NIL
end; { estVide }

procedure ajouter(var l : TListeChaineeDEntiers; e : Integer);
var temp :  TListeChaineeDEntiers;
begin
   temp:=l;
   new(l);
   l^.entier:=e;
   l^.listeSuivante:=temp
end; { ajouter }
```

# Liste chaînée d'entiers 8 / 10

## Developpement (suite)

```
function obtenirEntier(l :  TListeChaineeDEntiers) : Integer;
begin
   obtenirEntier:=l^.entier
end; { obtenirEntier }

function obtenirListeSuivante(l :  TListeChaineeDEntiers) : TListeChaineeDEntiers;
begin
   obtenirListeSuivante:=l^.listeSuivante
end; { obtenirListeSuivante }

procedure fixerListeSuivante(var l : TListeChaineeDEntiers; ls :
TListeChaineeDEntiers);
begin
   l^.listeSuivante:=ls
end;

procedure supprimerTete(var l :  TListeChaineeDEntiers);
var temp : TListeChaineeDEntiers;
begin
   temp:=l;
   l:=obtenirListeSuivante(l);
   dispose(temp)
end; { supprimerTete }
```

# Liste chaînée d'entiers 9 / 10

## Developpement (suite)

```
procedure supprimer(var l : TListeChaineeDEntiers);
var temp : TListeChaineeDEntiers;
begin
   while not estVide(l) do
      supprimerTete(l)
end; { supprimer }

end.
```

## Exemple d'utilisation

```
program testListeChainee;

uses ListeChaineeDEntiers;

procedure afficher(l : TListeChaineeDEntiers);
begin
   while not estVide(l) do
   begin
      write(obtenirEntier(l),' ');
      l:=obtenirListeSuivante(l);
   end;
   writeln()
end;
```

# Liste chaînée d'entiers 10 / 10

## Exemple d'utilisation (fin)

```
var l : TListeChaineeDEntiers;

begin
   l:=listeVide();
   ajouter(l,1);
   ajouter(l,2);
   ajouter(l,3);
   afficher(l);
   supprimer(l)
end.
```

## Exercice

- Qu'affiche ce programme ?

# Utilisation des listes chaînées

## Des algorithmes naturellement récursifs

- De part la définition récursive des listes chaînées, les algorithmes les utilisant sont naturellement récursifs

## Exemples d'algorithmes

- obtenir le nombre d'éléments, le ième élément
- savoir si un élément est présent
- concaténer deux listes chaînées
- insérer un élément à la ième position
- insérer un élément dans une liste chaînée ordonnée
- fusionner deux listes chaînées ordonnées
- inverser une liste chaînée
- ...