IN103
Résolution de problèmes algorithmiques
une approche fondée sur les structures de données

Alexandre Chapoutot

Année académique 2023-2024

`https://perso.ensta-paris.fr/~chapoutot/teaching/in103/`

Plan du cours

Motivation pour une bibliothèque spécifique à IN103

- Par ce que le langage C n'est pas doté d'une bibliothèque de structure de données comme d'autres langages (p. ex., C++, OCaml, Java, ...)
- Il existe des bibliothèques de structures de données en C
  - GLib (associée à GNOME), Gnulib, SGLIB, liblfds
  - et plein de trucs sur GitHub

  Mais elles ne sont pas aisées à prendre en main ou incomplètes.
- La bibliothèque **libin103** a été concue pour simplifier l'utilisation et illustrer tous les concepts du cours, pas pour les performances

Remarque

Beaucoup d'algorithmes sont efficaces car ils ont une gestion des données efficaces. Cette gestion s'appuie sur des structures de données, c'est-à-dire, une représentation des données pertinentes pour résoudre la classe de problèmes considérée.

La bibliothèque libin103

Tour d'horizon de la bibliothèque `libin103`

Pour les besoins de l'enseignement IN103, une bibliothèque logicielle a été développée, mettant en œuvre les principales structures de données et algorithmes utilisés durant le cours.

Les structures de données présentes sont :

- **Listes chaînées** (`list`), les **Piles** (`stack`), les **Files** (`queue`) et les **Ensembles** (`set`), en plusieurs versions :
  Pour les `int`, `double`, `char` et une version générique avec `void*`
- Les **Arbres binaires** (`bitree`), les **Arbres binaires de recherche** (`bistree`) et les **Tas** (`heap`), en plusieurs versions :
  Pour les `int`, `double`, `char` et une version générique avec `void*`
- Les **Ensembles disjoints** (`union-find`) uniquement en version pour les `int`
- Les **Graphes** (`graph`) en plusieurs versions :
  Pour les `int` et une version générique avec `void*`

A chaque séance du cours, une présentation du principe de fonctionnement des structures de données et des fonctions de manipulation seront réalisées.

Tour d'horizon de la bibliothèque `libin103`

Répertoire `libin103`

```
libin103-1.4/
|- include/
|- Makefile
|- Makefile.common
|- README.md
|- source/
|- templates/
|- tests/
```

Explications

- Dans le répertoire `include` : les fichiers entêtes `.h`, interface de programmation des structures de données ;
- Dans le répertoire `source` : les fichiers définitions `.c`. C'est également dans ce répertoire que sera stocké le fichier **libin103.a** ;
- Dans le répertoire `templates` : les fichiers qui servent à générer des versions spécialisées (`int`, `double`, etc.) des structures de données ;
- Dans le répertoire `tests` : les programmes qui ont servi à la mise au point de la bibliothèque. Ils peuvent également être lus comme des exemples d'utilisation de la bibliothèque.

Documentation HTML

`https://perso.ensta-paris.fr/~chapoutot/teaching/in103/refman/index.html`

Exemple : pour utiliser les listes chaînées

Fichiers .h

Dans le répertoire include, on trouvera les fichiers suivants

list.h, character_list.h, generic_list.h, integer_list.h, real_list.h

Fichier list.h

```
#ifndef LIST_H
#define LIST_H

#include "integer_list.h"
#include "real_list.h"
#include "character_list.h"
#include "generic_list.h"

#endif
```

Les autres fichiers seront présentés dans le reste du cours.

Listes chaînées

Motivation pour une nouvelle structure de données

Pour le moment, les tableaux ont été étudiés. Ils sont

- de taille fixe
- mis en œuvre par une zone mémoire contiguë d'éléments du même type
- alloués statiquement ou dynamiquement

Pensez aux exemples de M. Pessaux avec des fichiers dont la première ligne est le nombre de données à lire ensuite !

Les tableaux

- ne s'adaptent pas facilement à des données qui bougent beaucoup (insertion/suppression) ;
- offrent une représentation adaptée à l'adressage directe.

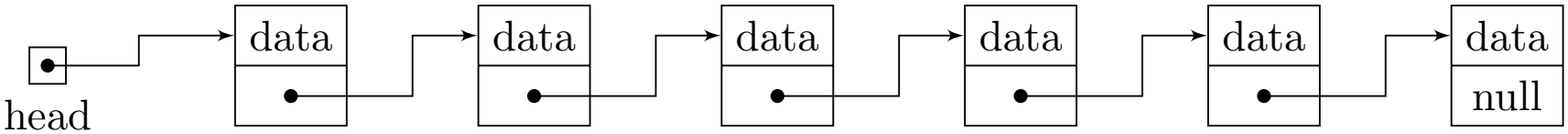
Implémentation des listes chaînées

Mathématiquement une `liste` peut être définie de manière inductive par :

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

$\Rightarrow$ type inductif de la forme : *list* ::= `Nil` | elem; *list*

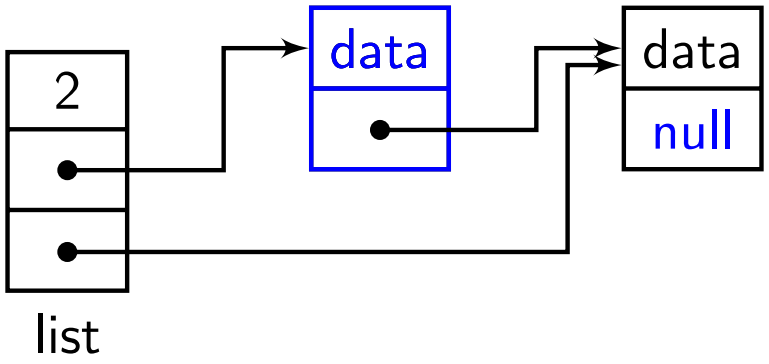
- Les langages fonctionnels (OCaml, Haskell, ...) permettent de définir des types inductifs mais pas le langage C.
- En langage C, les pointeurs sont utilisés pour créer un chaînage entre les éléments de la liste.



Zoologie des listes chaînées

- simplement/doublement chaînée ;
- circulaire, ...
- avec pointeur de fin, champs taille, etc.

Implémentation des listes chaînées

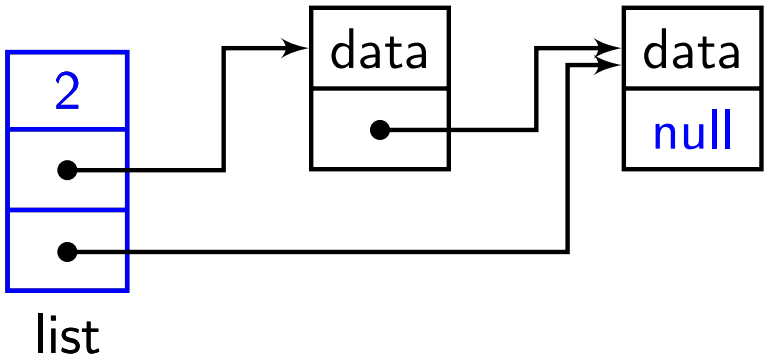


Une représentation des éléments de la liste

```
typedef struct integer_list_elmt_ {
  int data;  /* can be double or char or void*  */
  struct integer_list_elmt_t* next;
} integer_list_elmt_t;
```

Remarque : les listes permettent de gérer des données de manière dynamique (et les cellules de la chaîne sont répartis dans la mémoire) mais on perd l'accès direct aux éléments.

Implémentation des listes chaînées



Une représentation de la liste

```
typedef struct integer_list_ {
  int size;
  integer_list_elmt_t* head;
  integer_list_elmt_t* tail;
} integer_list_t;
```

Remarque : représentation choisie pour également faciliter la mise en œuvre d'autres structures de données (cf plus loin)

Interface de programmation [1]

Interface de programmation

Contient la liste des signatures des fonctions (ainsi qu'une courte description) permettant de manipuler une bibliothèque logicielle.

En général, l'API pour les structures de données contient

- les fonctions de créations/destructions de la structure de données
- les fonctions pour ajouter/supprimer des éléments dans la structure de données
- les fonctions permettant de récupérer de l'information (comme la taille)

1. En anglais, *Application Programming Interface* (API).

API des listes chaînées – 1

API des listes – Création/Destruction

```
void integer_list_init(integer_list_t*);

void integer_list_destroy(integer_list_t*);
```

API des listes – Accesseurs

```
int integer_list_size (integer_list_t*);

integer_list_elmt_t* integer_list_head(integer_list_t*);

integer_list_elmt_t* integer_list_tail(integer_list_t*);

integer_list_elmt_t* integer_list_next(integer_list_elmt_t*);

int integer_list_data(integer_list_elmt_t*);
```

API des listes chaînées – 2

API des listes – Prédicats

```
bool integer_list_is_head(integer_list_t*,
                          integer_list_elmt_t*);

bool integer_list_is_tail(integer_list_elmt_t*);
```

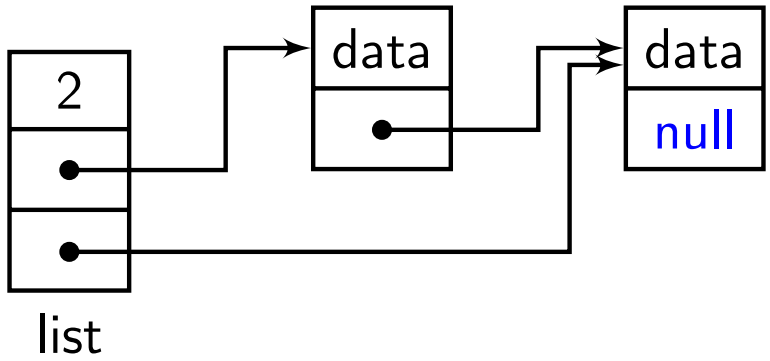
API des listes – Insertion/Suppression

```
int integer_list_ins_next(integer_list_t*,
                          integer_list_elmt_t*,
                          int data);

int integer_list_rem_next(integer_list_t*,
                          integer_list_elmt_t*,
                          int* data);
```

Insertion dans une liste chaînée

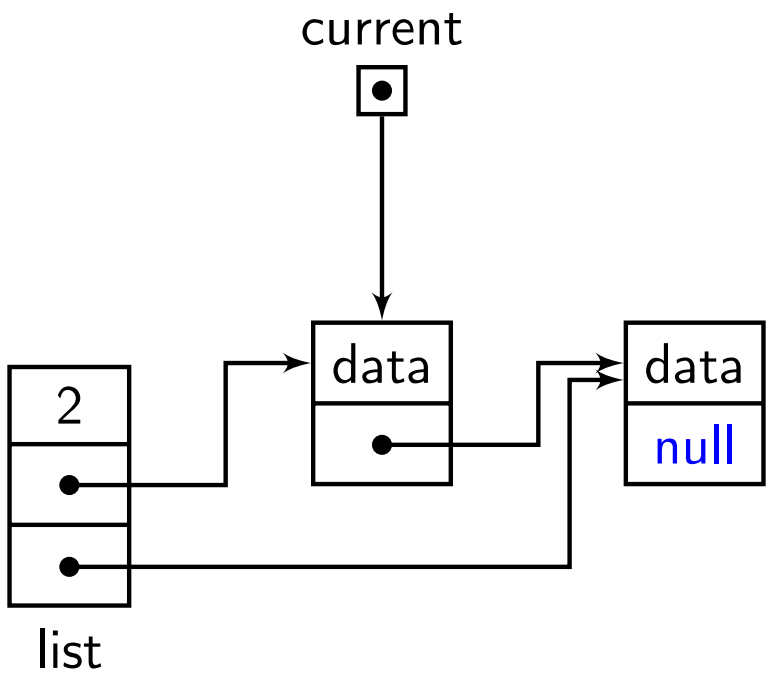
Suivant l'API, l'insertion dans une liste est réalisée à partir d'un élément.



Étapes d'insertion

Insertion dans une liste chaînée

Suivant l'API, l'insertion dans une liste est réalisée à partir d'un élément.

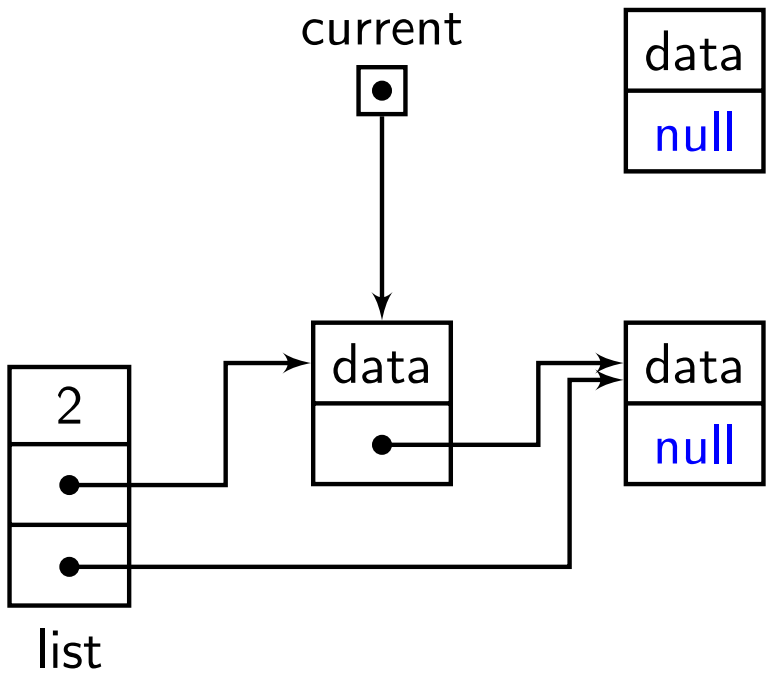


Étapes d'insertion

1. On donne l'élément après lequel insérer un nouvel élément

Insertion dans une liste chaînée

Suivant l'API, l'insertion dans une liste est réalisée à partir d'un élément.

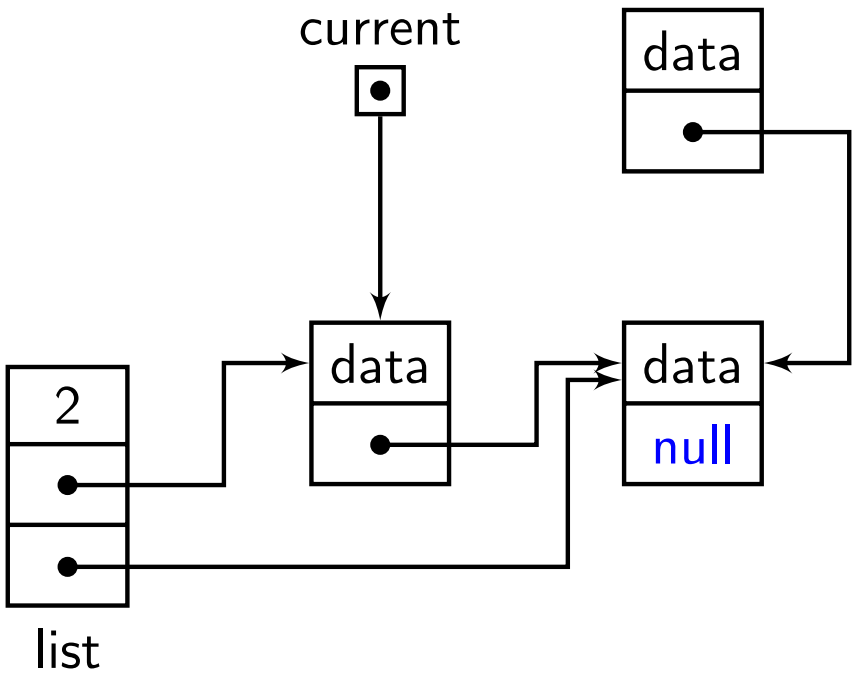


Étapes d'insertion

1. On donne l'élément après lequel insérer un nouvel élément
2. On crée une nouvelle cellule avec la nouvelle donnée

Insertion dans une liste chaînée

Suivant l'API, l'insertion dans une liste est réalisée à partir d'un élément.

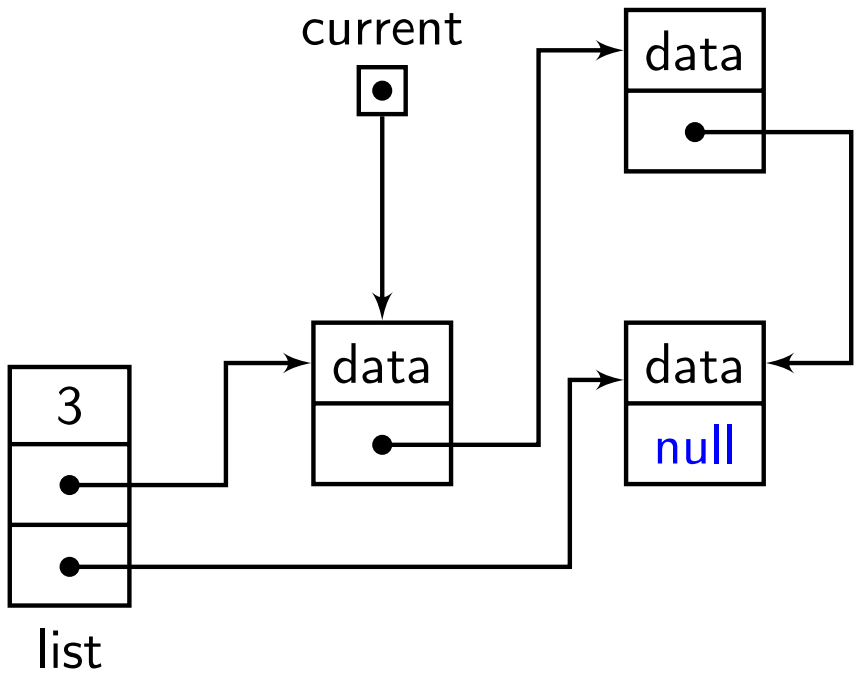


Étapes d'insertion

1. On donne l'élément après lequel insérer un nouvel élément
2. On crée une nouvelle cellule avec la nouvelle donnée
3. On fait le nouveau chaînage (attention à l'ordre des opérations)
4. On met à jour de la taille de la liste chaînée

Insertion dans une liste chaînée

Suivant l'API, l'insertion dans une liste est réalisée à partir d'un élément.



Étapes d'insertion

1. On donne l'élément après lequel insérer un nouvel élément
2. On crée une nouvelle cellule avec la nouvelle donnée
3. On fait le nouveau chaînage (attention à l'ordre des opérations)
4. On met à jour de la taille de la liste chaînée

Insertion dans une liste chaînée

Quelques comportements particuliers

- Que se passe-t-il si l'élément après lequel insérer est le pointeur `NULL` ?

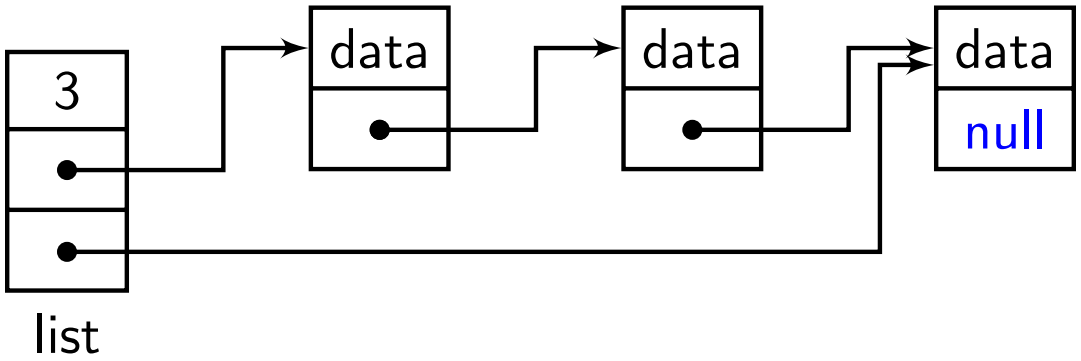
  Insertion en tête

- Comment faire une insertion en fin de liste ?

```
integer_list_ins_next(&list, integer_list_tail(&list), data);
```

Suppression dans une liste chaînée

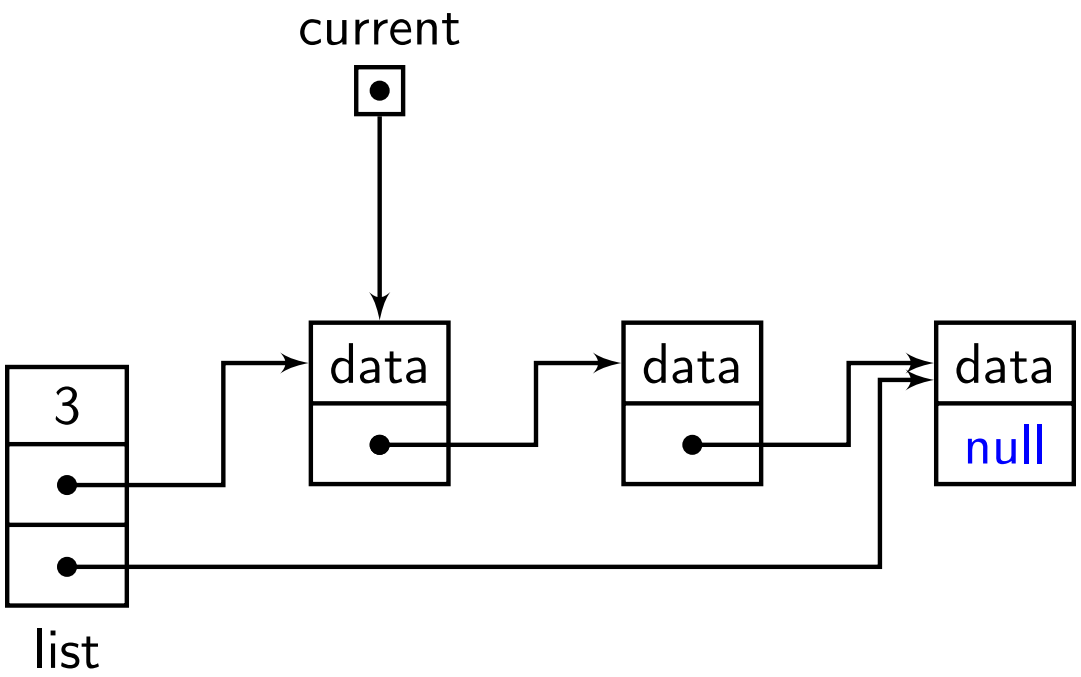
Suivant l'API, la suppression dans une liste est réalisée à partir d'un élément.



Étapes de suppression

Suppression dans une liste chaînée

Suivant l'API, la suppression dans une liste est réalisée à partir d'un élément.

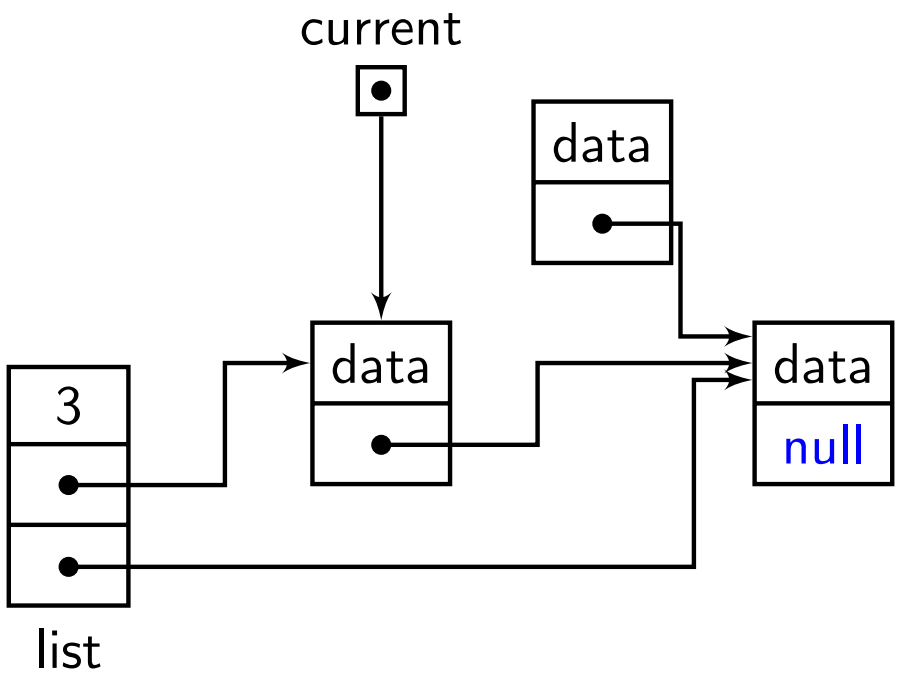


Étapes de suppression

1. On donne l'élément après lequel supprimer un nouvel élément

Suppression dans une liste chaînée

Suivant l'API, la suppression dans une liste est réalisée à partir d'un élément.

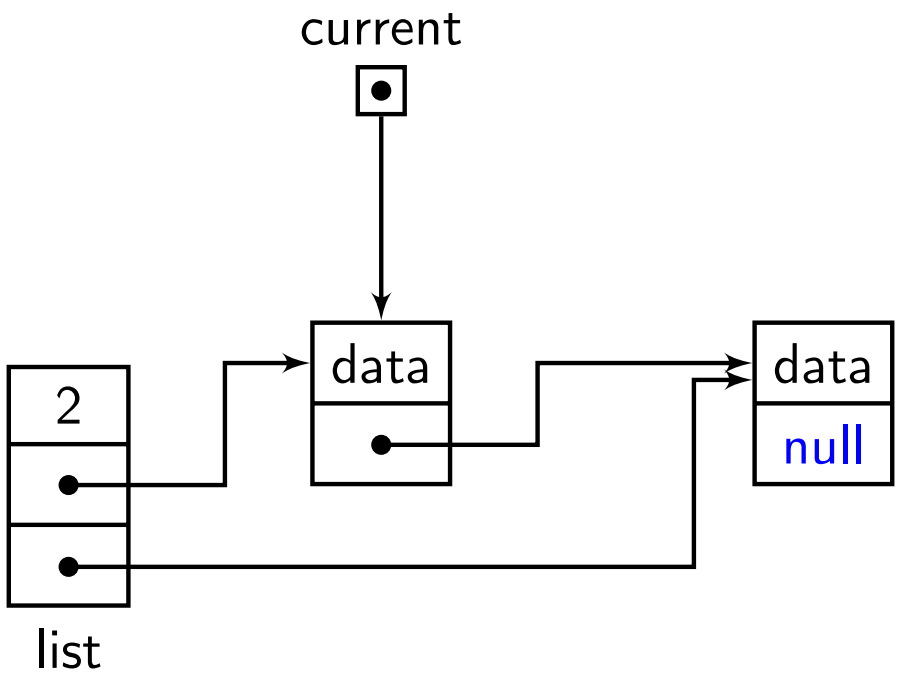


Étapes de suppression

1. On donne l'élément après lequel supprimer un nouvel élément
2. On fait le nouveau chaînage (une seule opération)

Suppression dans une liste chaînée

Suivant l'API, la suppression dans une liste est réalisée à partir d'un élément.



Étapes de suppression

1. On donne l'élément après lequel supprimer un nouvel élément
2. On fait le nouveau chaînage (une seule opération)
3. On désalloue la cellule
4. On met à jour de la taille de la liste chaînée

Suppression dans une liste chaînée

Quelques comportements particuliers

- Que se passe-t-il si l'élément après lequel supprimer est le pointeur `NULL` ?

  Suppression en tête

- Que se passe-t-il si on essaie de faire une suppression du dernier élément ?

  Pas possible, simplement code de retour égale à -1

  Note : pour supprimer le dernier élément il faut garder un pointeur sur le pénultième élément.

- Que se passe-t-il si on essaie de faire une suppression dans une liste vide ?

  Pas possible, simplement code de retour égale à -1

Complexité des opérations sur les listes

Fonction	**Complexité**	**Commentaire**
Initialisation	$\mathcal{O}(1)$	Mise de la mémoire à zéro
Destruction	$\mathcal{O}(n)$	Parcourir la liste pour supprimer les éléments
Accesseurs	$\mathcal{O}(1)$	
Insertion	$\mathcal{O}(1)$	**Hyp :** on sait où faire l'insertion
Suppression	$\mathcal{O}(1)$	**Hyp :** on sait où faire l'insertion
Recherche	$\mathcal{O}(n)$	Pas d'accès direct, il faut parcourir la liste

Remarque

La structure de liste a été pensée pour avoir des opérations avec des complexités basses.
Essentiellement car une liste est formée d'un pointeur de tête, d'un pointeur de queue, d'un champs taille.

Retour sur les types génériques

Liste chaînée générique

```
typedef struct generic_list_elmt_ {
  void *data;
  struct generic_list_elmt_  *next;
} generic_list_elmt_t;

typedef struct generic_list_ {
  int size;
  int (*compare)(const void *key1, const void *key2);
  void* (*build)(const void *data);
  void (*destroy)(void *data);
  generic_list_elmt_t *head;
  generic_list_elmt_t *tail;
} generic_list_t;
```

Conséquence :

- Comme `void*` est un pointeur vers une zone mémoire sans autre information, il faut indiquer comment créer des éléments, comment comparer des éléments et comment détruire des éléments.

Retour sur les types génériques

La principale modification concerne la fonction d'initialisation,

Fonction d'initialisation [a]

a. Le mot clef `const` indique au compilateur que la zone mémoire pointée ne doit pas changer (pas d'affectation possible sur les arguments).

```
void generic_list_init(generic_list_t *list,
            int (*compare) (const void *key1, const void *key2),
            void* (*build)(const void *data),
            void (*destroy)(void *data));
```

Dans ce contexte

- on utilise des **pointeurs sur des fonctions**.
- notez le prototype de ces pointeurs de fonction

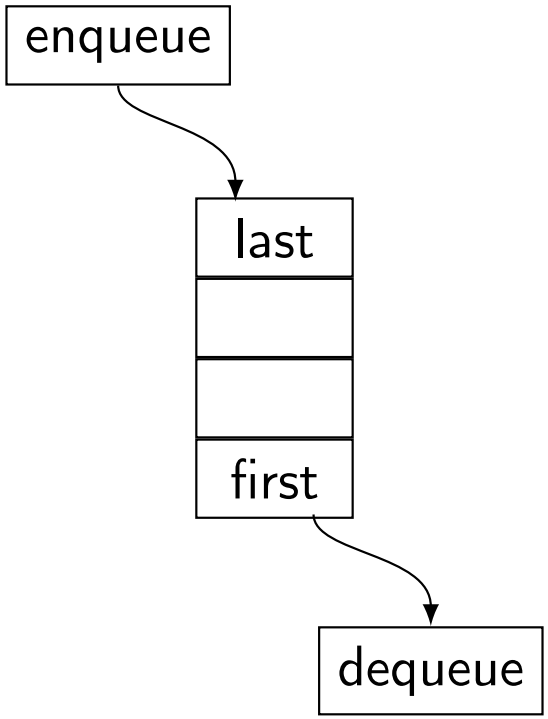
```
<type de retour> (*nom) (<types arguments>);
```

- Toutes autres fonctions suivent le même format **excepté que vous devez mémoriser** quel type de données est stocké ! !

Files

Structure de données : File

- Stocker des choses à traiter dans l'ordre d'arrivée ($\approx$ file d'attente).
- Comportement (*FIFO* : ***F**irst **I**n **F**irst **O**ut*) : premier inséré, premier retiré
- On peut consulter les éléments dans la file mais pas les retirer tant que ceux de devant sont dans la file.
- **On traite les éléments dans l'ordre d'arrivée (ou d'insertion dans la file)**



Applications des files

- Stocker des «transactions» à effectuer : mémoire tampon («buffer»), accès à des zones mémoires protégées.
- Ordonnancer les processus dans un OS multitâche.
- Parcours «en largeur » d'arbres ou de graphes (voir plus tard les cours sur les arbres et les graphes).
- Services client/serveur (Web).

Implémentations des files

Plusieurs façon de faire :

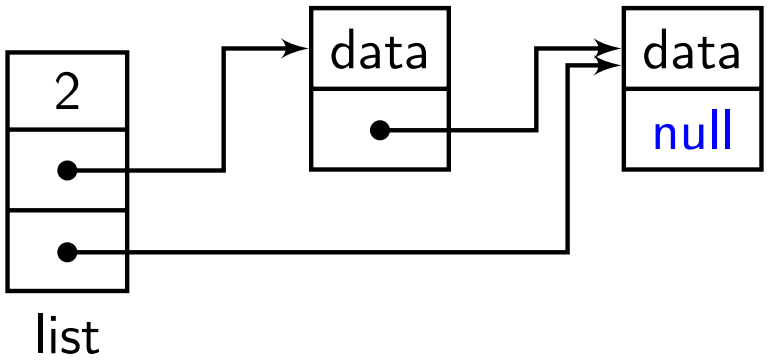
- A l'aide de tableaux de manière circulaire

```
struct queue {
  unsigned int max_nb;     /* Nombre max d'elements. */
  unsigned int cur_nb;     /* Nombre actuel d'elements. */
  unsigned int first;      /* Indice du premier element. */
  int *data;
};
```

Remarque : On doit mettre en œuvre des gestions des dépassements de capacité (exception/realloc)

- **A l'aide d'une liste chaînée** comme dans la bibliothèque `libin103`

Implémentation des files dans `libin103`



Une file est une liste

```
typedef integer_list_t integer_queue_t;
```

MAIS avec une API différente

API des files – 1

API des files – Création/Destruction

```
void integer_queue_init (integer_queue_t* queue);

void integer_queue_destroy (integer_queue_t* queue);
```

API des files – Accesseurs

```
int integer_queue_size (integer_queue_t* queue);

int integer_queue_peek(integer_queue_t* queue);
```

- peek, fonction pour récupérer (sans supprimer) le premier élément (le plus vieux) de la file

API des files – 2

API des files – Insertion/Suppression

```
int integer_queue_enqueue(integer_queue_t* queue, int data);

int integer_queue_dequeue(integer_queue_t* queue, int* data);
```

- enqueue, fonction pour insérer un nouvel élément dans la file
- dequeue, fonction pour supprimer le premier élément (le plus vieux) dans la file

Complexité des opérations sur les files

Fonction	**Complexité**	**Commentaire**
Initialisation	$\mathcal{O}(1)$	Mise de la mémoire à zéro
Destruction	$\mathcal{O}(n)$	Pareil que pour les listes
Accesseurs	$\mathcal{O}(1)$	
`Enqueue`	$\mathcal{O}(1)$	Équivalent à insertion en fin de la liste
`Dequeue`	$\mathcal{O}(1)$	Équivalent à suppression en début de la liste

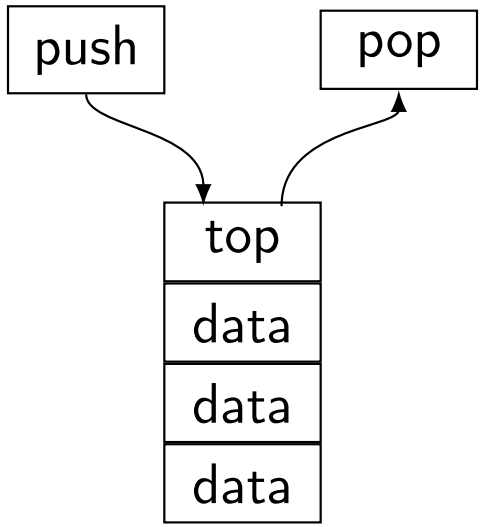
Remarque

La structure de liste a été pensée pour mettre en œuvre facilement des files d'où les classes de complexité basses

Piles

Structure de données : Pile

- Comment dans la vie courante : stocker des choses comme une pile d'assiettes.
- Comportement (*LIFO* : ***L**ast **I**n **F**irst **O**ut*) : dernier déposé, premier retiré
- On ne peut pas consulter les éléments sous le sommet de la pile.



Applications des piles

- Garder trace d'un historique de traitements (voir plus tard les cours sur les arbres et les graphes)
- Garder trace de traitements non complétés («bracktracking») : classique en programmation par contraintes

- Base de nombreuses machines virtuelles (Java par exemple).

Exemple, notation polonaise inversée (RPN)

- Notation postfixe "1 2 + 5 ×"
- Opérateurs après les opérandes (comme les anciennes calculatrices HP)
- Opérandes sur la pile, opérateur opérant sur les éléments de la pile.

- . . . Même structure que la pile d'exécution d'un programme C. . .

Implémentations des piles

Plusieurs façon de faire :

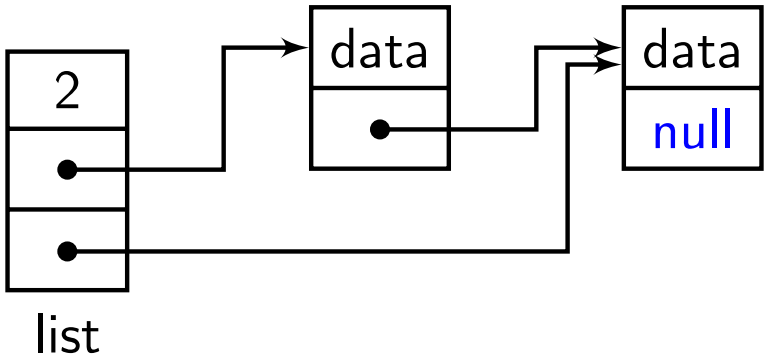
- A l'aide de tableaux et un pointeur de pile indiquant la prochaine case libre

```
#define MAX_SIZE 64
struct stack {
    unsigned int sp;      /* Pointeur de pile. */
    int data[MAX_SIZE];    /* Memoire de pile. */
};
```

  Remarque : On doit mettre en œuvre des gestions des dépassements de capacité (exception/realloc)

- **A l'aide d'une liste chaînée** comme dans la bibliothèque `libin103`

Implémentation des piles



Une pile est une liste

```
typedef integer_list_t integer_stack_t;
```

MAIS avec une API différente

API des piles – 1

API des piles – Création/Destruction

```
void integer_stack_init (integer_stack_t* stack);

void integer_stack_destroy (integer_stack_t *stack);
```

API des piles – Accesseurs

```
int integer_stack_size (integer_stack_t * stack);

int integer_stack_peek(integer_stack_t * stack);
```

- peek, fonction pour récupérer (sans supprimer) le premier élément de la pile (le sommet, la valeur la plus récente)

API des piles – 2

API des piles – Insertion/Suppression

```
int integer_stack_push(integer_stack_t *stack, int data);

int integer_stack_pop(integer_stack_t *stack, int* data);
```

- push, fonction pour insérer un nouvel élément au sommet de la file
- pop, fonction pour supprimer un le sommet de la file

Complexité des opérations sur les piles

Fonction	**Complexité**	**Commentaire**
Initialisation	$\mathcal{O}(1)$	Mise de la mémoire à zéro
Destruction	$\mathcal{O}(n)$	Même cas que les listes
Accesseurs	$\mathcal{O}(1)$	
`Push`	$\mathcal{O}(1)$	Équivalent à insertion en début de la liste
`Pop`	$\mathcal{O}(1)$	Équivalent à suppression en début de la liste

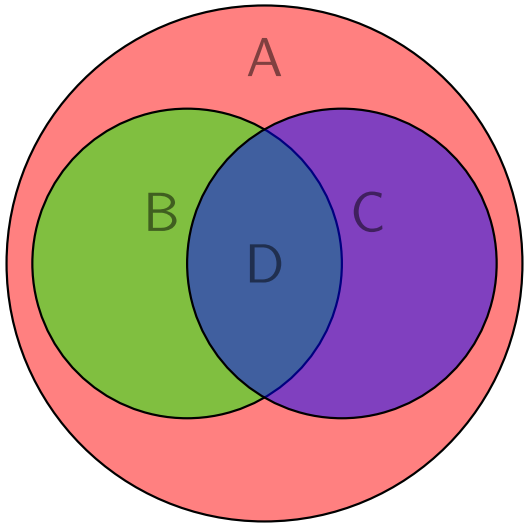
Remarque

La structure de liste a été pensée pour mettre en œuvre facilement ces piles d'où les classes de complexité basse.

Ensembles

Structure de données : ensembles

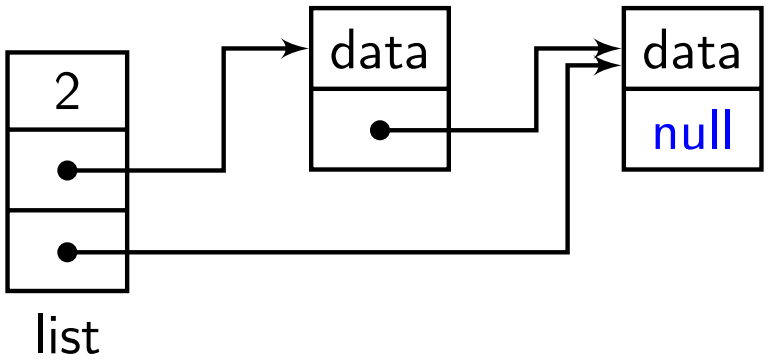
- Collection d'éléments sans multi occurrence
- Appliquer des opérations ensemblistes : union, intersection, différence



Quelques applications des ensembles

- Corrélation de données
- Couverture d'ensembles (cf TP)
- Opérations mathématiques sur les ensembles
- Les algèbres relationnelles comme pour le langage SQL

Implémentation des ensembles



Un ensemble est une liste

```
typedef integer_list_t integer_set_t;
```

mais avec une API différente

API des ensembles – 1

API des ensembles – Création/Destruction

```
void integer_set_init(integer_set_t *set);

void integer_set_destroy (integer_set_t *set);
```

API des ensembles – Accesseurs

```
int integer_set_size(integer_set_t*);
```

API des ensembles – Prédicats

```
bool integer_set_is_member(integer_set_t*, int);

bool integer_set_is_subset(integer_set_t*, integer_set_t*);

bool integer_set_is_equal(integer_set_t*, integer_set_t*);
```

API des ensembles – 2

API des ensembles – Insertion/Suppression

```
int integer_set_insert(integer_set_t *set, int data);

int integer_set_remove(integer_set_t *set, int *data);
```

- `insert`, fonction pour insérer un nouvel élément au dans l'ensemble en garantissant l'unicité

API des ensembles – Opérations ensemblistes

```
int integer_set_union(integer_set_t *setu,
    integer_set_t*, integer_set_t*);

int integer_set_intersection(integer_set_t *seti,
    integer_set_t*, integer_set_t*);

int integer_set_difference(integer_set_t *setd,
    integer_set_t*, integer_set_t*);
```

Intuition des algorithmes

On considère deux ensembles s_1 et s_2 de cardinalité respectives n_1 et n_2.

- Insertion : ajout d'un nouvel élément si pas déjà présent (test d'appartenance)
- Union : insérer les éléments de s_1 dans s_2
- Intersection : pour chaque élément de s_1, on vérifie sa présence dans s_2, si oui insertion
- Différence : pour chaque élément de s_1, on vérifie sa présence dans s_2, si non insertion
- Test sous ensemble : vérifie que tous les éléments de s_1 sont dans s_2
- Test égalité : tous les éléments de s_1 sont dans s_2 et tous les éléments de s_2 sont dans s_1

Complexité des opérations sur les ensembles

Fonction	**Complexité**	**Commentaire**
Initialisation	$\mathcal{O}(1)$	Mise de la mémoire à zéro
Destruction	$\mathcal{O}(n)$	Pareil que pour les listes
Accesseurs	$\mathcal{O}(1)$	
Test appartenance	$\mathcal{O}(n)$	Parcours de liste
Test sous ensemble	$\mathcal{O}(n_1 n_2)$	Parcours des deux listes
Test égalité	$\mathcal{O}(n_1 n_2)$	Parcours des deux listes (2 fois)
`Insert`	$\mathcal{O}(n)$	Vérification de l'appartenance
`Remove`	$\mathcal{O}(n)$	Recherche de l'élément à supprimer
Union	$\mathcal{O}(n_1 n_2)$	Parcours des deux listes
Intersection	$\mathcal{O}(n_1 n_2)$	Parcours des deux listes
Différence	$\mathcal{O}(n_1 n_2)$	Parcours des deux listes

Remarque

La structure de liste permet une mise en œuvre facile des ensembles mais avec une complexité importante. On peut faire mieux si on considère une structure sous jacente arborescente, cf plus tard dans le cours.