1 Introduction

Nous avons déjà rencontré deux structures de données assez simples :

— Les tableaux

— Les structs C

Ces deux types d'objets permettent de stocker des données d'une manière **structurée** : les tableaux comme les structs sont munis d'opérations bien spécifiques pour écrire ou lire des valeurs : accès à une case, à un attribut...

Définition 1

La spécification d'une ***structure de données abstraite*** est constituée de :

— son type/format, c'est à dire le genre d'informations que l'on peut enregistrer ;

— ses **opérations**, c'est à dire la manière dont on interagit avec la structure, la manière dont on y lit, écrit et modifie des données.

Exemple 1

Un tableau d'éléments de type `T` est une structure de données abstraite. C'est une suite finie d'éléments de type `T`, et ses opérations sont :

— `creer_tab(n)` : crée et renvoie un tableau d'une taille n

— `ecrire(t, i, x)` : écrit x à la i-ème case du tableau t

— `lire(t, i)` : renvoie la i-ème case du tableau t

— `taille(t)` : envoie la taille du tableau t

En C, si l'on utilise directement les tableaux fournis par le langage, comme on l'a fait jusqu'à maintenant, on ne peut pas obtenir la taille d'un tableau donné, on doit s'en rappeler dans une variable à part. On peut néanmoins implémenter notre propre version des tableaux, en utilisant les structs :

```
1 struct tableau{
2   int taille;
3   float* valeurs;
4 };
5 typedef struct tableau tableau_t;
```

Ensuite, on peut implémenter les opérations données par la spécification des tableaux :

```
1 tableau_t* creer_tab(int taille){
2   tableau_t* res = malloc(sizeof(tableau_t));
3   res->taille = taille;
4
5   res->valeurs = malloc(taille*sizeof(float));
6   for (int i = 0; i < taille; i++){
7     res->valeurs[i] = 0;
8   }
9 }
```

(On choisit arbitrairement de mettre tous les éléments à 0 initialement.)

```
1 void ecrire(struct tableau_t* t, int i, float x){
2   assert(0 <= i && i < t->taille);
3   t->valeurs[i] = x;
4 }
```

On remarque qu'implémenter nous même la structure de tableau permet de rajouter une couche de sécurité : on vérifie systématiquement qu'on accède bien à une case valide du tableau.

```c
float lire(struct tableau_t* t, int i){
  assert(0 <= i && i < t->taille);
  return t->valeurs[i];
}
```

```c
int taille(struct tableau_t* t){
  return t->taille;
}
```

En ayant écrit les fonction précédentes, on a **implémenté** la structure de données abstraite appelée "tableau". On dit que l'on a **implémenté** une **structure de données concrètes**.

Dans le reste du programme, peut alors manipuler les tableaux de la manière suivante :

```c
void echanger(tableau_t* t, int i, int j){
  float x = lire(t, i);
  float y = lire(t, j);
  ecrire(t, i, y);
  ecrire(t, j, i);
}

int main(){
  tableau_t* t = creer_tab(15);
  printf("Tableau créé, de taille %d\n", taille(t);
  ecrire(t, 7, 0.25);
  ecrire(t, 2, 11.5);
  echanger(t, 2, 7);
}
```

Notons que le code ci-dessus fonctionnerait peu importe comment est implémenté le type `tableau_t`, du moment qu'il existe et que les quatre opérations existent aussi : on a seulement besoin de la spécification de la SDA de tableau.

Définition 2

Une ***structure de données concrète*** est l'implémentation d'une structure de données abstraite.

Dans un programme, lorsque l'on **implémente** une structure de données, on prend le point de vue de la SDC : on choisit une implémentation, on crée les opérations concrètes. Cependant, lorsque l'on **utilise** une structure de données dans du code, on considère la structure abstraite.

Par exemple, avec la structure de tableaux implémentée plus haut, on pourrait uniquement utiliser les 4 opérations de la spécification :

```c
tableau_t* t_1 = creer_tableau(5); // VALIDE
tableau_t* t_2 = malloc(sizeof(tableau_t)); // INVALIDE

float x = lire(t_1, 2); // VALIDE
float y = t_1->valeurs[2]; // INVALIDE
```

En pratique, il sera courant d'implémenter une structure de données en écrivant un couple de fichiers `.h/.c`. Le fichier header ne contient que les déclarations, y compris pour la structure, qui est déclarée sans remplir les attributs :

```c
typedef struct tableau tableau_t;

// Crée un tableau de taille `taille` non initialisé
tableau_t* creer_tab(int taille);

// Renvoie la case i de t. Précondition: i est un indice valide
float lire(struct tableau_t* t, int i);

// Stocke x dans la case i de t. Précondition: i est un indice valide
void ecrire(struct tableau_t* t, int i, float x);

// Renvoie le nombre de cases de t
int taille(struct tableau_t* t);
```

Le fichier C contiendra les définitions de la structure et des fonctions. Ainsi, les fichiers C qui utiliseront la structure n'auront aucun moyen d'accéder aux attributs de la structure, car ils n'ont accès qu'au fichier header. On parle d'utilisation en **boîte noire**, car on utilise la structure sans connaître son fonctionnement interne.

La documentation d'une structure de données concrète est très importante, car c'est elle qui explique à l'utilisateur comment utiliser les différentes opérations. Autrement dit, la documentation **est** la spécification. Plus que jamais, lorsque l'on implémente des structures de données, on doit ***commenter*** le code.

Définition 3

Dans la suite, on distinguera trois familles d'opérations possibles pour les SDA :

- Les **constructeurs**, qui servent à créer et initialiser une structure
- Les **accesseurs**, qui permettent de lire une information dans la structure
- Les **transformateurs**, qui permettent de modifier la structure : en changeant une valeur, en ajoutant ou supprimant un élément, etc...

Certaines opérations peuvent être dans deux (ou plus) familles à la fois.

Par exemple pour la SDA de tableau :

- `creer_tab` est un constructeur ;
- `lire` est un accesseur ;
- `ecrire` est un transformateur ;
- `taille` est un accesseur.

On pourrait imaginer une cinquième opération, `copie(t)` qui renvoie une copie du tableau t : c'est à la fois un constructeur ET un accesseur.

Destructeur Lorsque l'on crée une SDC, on rajoute parfois un quatrième type d'opérations, les **destructeurs**, qui servent à détruire une structure et libérer toutes les ressources qu'elle utilisait.
On pourrait rajouter un destructeur `free_tab` aux tableaux comme suit :

```
1 void free_tab(struct tableau* t){
2   free(t->valeurs);
3   free(t);
4 }
```

Dans la suite du chapitre, on fixe `T` un type, et on considèrera des structures permettant de stocker des éléments de ce type. On omettra généralement les destructeurs dans la description des SDA, mais il faudra systématiquement penser à les programmer pour les SDC !

2 Pile

Une pile est une structure de données se comportant comme une pile d'assiette : on peut enlever l'assiette du dessus, ajouter une assiette au sommet de la pile, mais il est difficile d'insérer ou de retirer une assiette au milieu.

Cette structure fonctionne donc selon le principe "Dernier arrivé, premier sorti", ou ***LIFO*** (Last In First Out) : la dernière donnée que l'on ajoute sera toujours la première donnée que l'on devra retirer : c'est le **sommet** de la pile. Inversement, la première valeur que l'on ajoute dans une pile ne pourra être lue qu'une fois que toutes les valeurs suivantes ont été enlevées : c'est la **base** de la pile.

Exemple 2

Voici un exemple d'utilisation d'une pile :

A Structure de données abstraite

Une pile représente une suite finie d'éléments de type `T`, dont la taille peut varier. Ses opérations sont :

- **pile_vide**() crée une nouvelle pile vide (**Constructeur**) ;
- **empiler**(P, x) ajoute un nouvel élément x sur le sommet de la pile P (**Transformateur**) ;
- **depiler**(P) enlève le sommet de la pile, et le renvoie (**Transformateur** et **Accesseur**) ;
- **est_vide**(P) détermine si la pile P est vide (**Accesseur**).

Remarque 1

Dans certaines définitions, on a deux opérations séparées pour lire le sommet de pile et pour le supprimer. Ici, l'opération de dépilage est à la fois un accesseur et un transformateur.

Exercice 1

Exécuter l'algorithme suivant sur le tableau $T = [3, 1, 4, 1, 5]$: que fait-il ?

Algorithme 1 : ???

Entrée(s) : T un tableau de taille n
Sortie(s) : ???

```
1 P ← pile_vide();
2 pour i = 0 à n − 1 faire
3     empiler(P, T[i]);
4 i ← 0;
5 tant que non est_vide(P) faire
6     T[i] ← depiler(P);
7     i ← i + 1;
```

B Implémentation par tableau

Pour commencer, on s'autorise à avoir une taille limite pour la pile. On se fixe N_{max} un entier, et on implémente une pile avec un tableau de taille N_{max}.

Considérons la structure suivante :

```c
#define Nmax 10000
struct pile{
  int nb_elem;
  T tab[Nmax];
};
typedef struct pile pile_t;
```

Le principe de cette implémentation est que pour une pile `p`, seuls les `p->nb_elem` premiers éléments de `p->tab` ont un sens, et les cases d'indice `p->nb_elem` et au-delà peuvent contenir n'importe quoi. La case `p->tab[0]` contient la base de la pile, et `p->tab[p->nb_elem-1]` contient le sommet de la pile.

Exemple 3

Considérons une pile initialement vide, et dessinons la en tant que SDA et que SDC après quelques opérations.

L'implémentation des fonctions `pile_vide` et `est_vide` sont assez élémentaires vu la structure :

```c
pile_t* pile_vide(){
  pile_t* p = malloc(sizeof(pile_t));
  p->nb_elem = 0;
  return p;
}

/* Libère la mémoire allouée pour p (destructeur) */
void free_pile(pile_t* p){
  free(p);
}

bool est_vide(pile_t* p){
  return (p->nb_elem == 0);
}
```

Empilage, dépilage Pour empiler, on devra écrire dans la case `p->tab[p->nb_elem]` et incrémenter le nombre d'éléments. Pour dépiler, c'est l'inverse :

```
void empiler(pile_t* p, T x){
  assert(p->nb_elem < Nmax); // impossible d'empiler sur une pile pleine
  p->tab[p->nb_elem] = x;
  p->nb_elem++;
}

T depiler(pile_t* p){
  assert(!est_vide(p)); // impossible de dépiler une pile vide
  T res = p->tab[p->nb_elem-1];
  p->nb_elem--;
  return res;
}
```

Toutes ces opérations ont une complexité $\mathcal{O}(1)$.

C Implémentation par liste chaînée

L'implémentation basique par tableau est efficace, mais oblige les piles à avoir une taille limite. L'implémentation par liste chaînée permet d'éviter cette contrainte. Une liste chaînée est une structure composée de maillons reliés les uns aux autres en une unique chaîne. Chaque maillon contient un élément ainsi qu'une référence vers le maillon suivant. On stocke les éléments dans l'ordre du sommet vers la base :

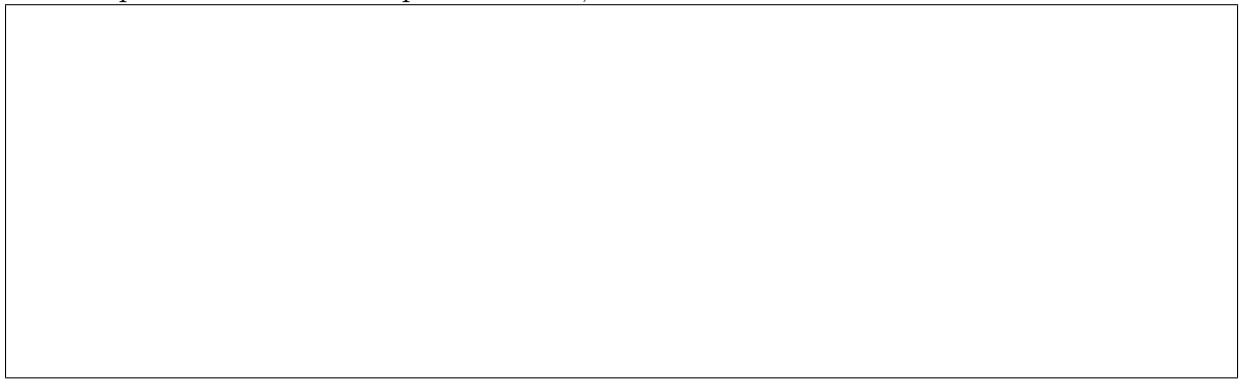
```
typedef struct maillon {
  T elem;
  struct maillon* suivant;
} maillon_t;


typedef struct pile{
  maillon_t* sommet;
} pile_t;
```

Exemple 4

Supposons que l'on a créé la pile suivante dans le main :

```
  pile_t* P = pile_vide();
  empiler(P, 6);
  empiler(P, 7);
  empiler(P, 8);
```

Voici à quoi ressemblerait la pile abstraite, et l'état réel de la mémoire :



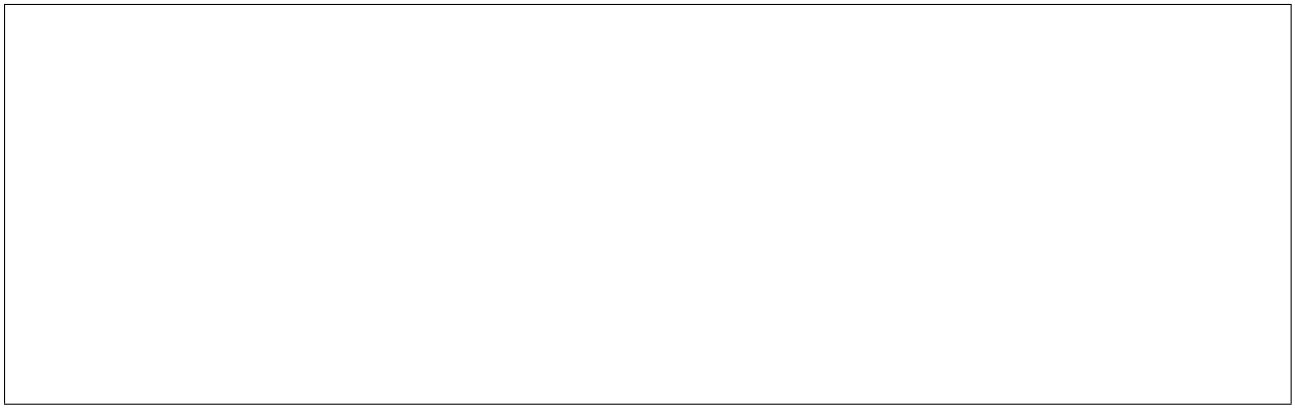
Dans cette implémentation, on utilise le pointeur `NULL` pour signaler qu'un maillon est le dernier, i.e. que c'est la base de la pile. Implémentons les différentes opérations.

Création de pile, déterminer si une pile est vide Une pile vide ne contient aucun maillon :

```
1 pile_t* pile_vide(){
2   pile_t* p = malloc(sizeof(pile_t));
3   p->sommet = NULL;
4   return p;
5 }
6
7 bool est_pile_vide(pile_t* p){
8   return (p->sommet == NULL);
9 }
```

Les deux opérations sont en $\mathcal{O}(1)$.

Empiler un élément au sommet Pour empiler un élément, on crée un nouveau maillon contenant l'élément, puis il faut raccorder les pointeurs de la pile et du nouveau maillon pour satisfaire le schéma suivant :



En tenant compte de tous les liens qui sont créés / modifiés / supprimés, on obtient le code suivant :

```
1 void empiler(pile_t* p, T x){
2   maillon_t* nouv_sommet = malloc(sizeof(maillon_t));
3
4   nouv_sommet->elem = x;
5   nouv_sommet->suivant = p->sommet;
6   p->sommet = nouv_sommet;
7 }
```

Complexité : $\mathcal{O}(1)$

Dépiler le sommet de pile Cette opération est essentiellement l'inverse de celle d'empilage : on doit extraire le maillon correspondant au sommet et recoller les liens. Il ne faut pas oublier de libérer la mémoire du maillon extrait afin d'éviter les fuites mémoire :

```
1  T depiler(pile_t* p){
2    assert(!est_pile_vide(p));
3
4    T res = p->sommet->elem;
5    maillon_t* nouveau_sommet = p->sommet->suivant;
6
7    free(p->sommet);
8    p->sommet = nouveau_sommet;
9    return res;
10 }
```

Complexité : $\mathcal{O}(1)$

Afficher une pile Lorsqu'on implémente une structure en C, il peut être utile d'ajouter des opérations qui ne font pas partie de la SDA, mais qui sont utiles pour le débug.
Pour afficher une pile, on parcourt tous ses éléments et on les affiche un par un. Si on le fait dans l'ordre naturel, on affichera le sommet en premier :

```
1  void print_pile(pile_t* p){
2    maillon_t* m = p->sommet;
3    while (m != NULL){
4      afficher m->elem; // dépend du type des données stockées
5      m = m->suivant;
6    }
7  }
```

Quand on manipule des listes chaînées, ce type de boucles revient souvent. En C, il est assez courant d'écrire ces boucles avec des for plutôt que des while, comme suit :

```
1  for(maillon_t* m = p->sommet; m != NULL; m = m->suivant){
2      ...
3  }
```

Une telle boucle se lirait "pour chaque maillon m de p, faire ..."

Exercice 2

Réécrire la fonction d'affichage en utilisant une boucle for.

Libérer une pile On parcourt la liste chaînée pour libérer un à un les maillons. Il faut faire attention à l'ordre des opérations : une fois qu'on a libéré un maillon, on ne peut plus accéder à ses attributs. Une solution est d'utiliser un pointeur pour se rappeler à chaque tour de l'adresse du maillon à libérer, et de le libérer après être passé au suivant :

```
1  void free_pile(pile_t* p){
2    maillon_t* m = p->sommet;
3    // Invariant: tous les maillons précédant strictement m
4    // ont été libérés
5    while (m != NULL){
6      maillon_t* a_liberer = m;
7      m = m->suivant;
8      free(a_liberer);
9    }
10   free(p);
11 }
```

Exercice 3

Dessiner l'état de la mémoire au fil de l'exécution de `free_pile`, et vérifier que toute la mémoire est libérée.

Complexité : $\mathcal{O}(n)$

D Utilisation de la SDA

Supposons que l'on a écrit un header `pile.h` avec la spécification de la SDA de pile. Supposons que l'on a aussi créé deux fichiers indépendants `pile_tab.c` et `pile_chaine.c` implémentant les piles selon les deux SDC vues précédemment. Chacun commencerait par :

```
1  #include "pile.h"
```

Lorsque l'on écrit un programme utilisant une pile, on doit inclure le header afin de pouvoir faire référence au type de la structure et à ses opérations :

```
1  #include "pile.h"
2
3  int main(){
4    pile_t* p = pile_vide();
5    empiler(p, "bla");
6    ...
7    free_pile(p);
8    return 0;
9  }
```

De plus, à la compilation, on peut **choisir** quelle structure concrète utilisée, simplement en changeant le fichier C utilisé :

```
gcc main.c pile_tab.c -o prog_avec_tab
gcc main.c pile_chaine.c -o prog_avec_chaine
```

Le fait d'avoir séparé la SDA et la SDC fait que l'on peut facilement changer la SDC, car seule la spécification importe. Le fichier `main.c` reste identique car il utilise seulement l'interface donnée par le header.

E Application

Voyons un exemple d'utilisation de la pile : les mots biens parenthésés.

Un mot sur l'alphabet { `[`, `(`,`]`,`)` } est dit bien parenthésé si chaque parenthèse ouvrante est associée à une parenthèse fermante correspondante de même type qui la suit dans le mot.

Par exemple, `([]([]))` est bien parenthésé, mais `([]` et `([)]` ne le sont pas.

On considère le problème de décision **Bien-Parenthésé** :

Étant donné $s = s_0 s_1 \dots s_{n-1}$ un mot sur { `[`, `(`,`]`,`)` }, s est-il bien parenthésé ?

Un algorithme qui apparaît naturellement pour tester si un mot est bien parenthésé est celui qui utilise une pile : lorsqu'on lit un symbole ouvrant on l'empile, et lorsqu'on lit un symbole fermant, on vérifie qu'il correspond bien au sommet de la pile.

Exemple 5

Déterminer si le mot `([]([()[]]])())` est bien parenthésé.

Plus précisément :

Entrée(s) : $s = s_0 s_1 \dots s_{n-1}$ un mot sur { `[`, `(`,`]`,`)` }
Sortie(s) : "Oui" si le mot est bien parenthésé, "Non" sinon

```
1  P ← pile_vide();
2  i ← 0;
3  tant que i < n faire
4      si s_i = (  ou  s_i = [ alors
5          empiler(P, s_i);
6      sinon
7          si est_vide(P) alors
8              retourner Non ;
9          sinon
10             b ← dépiler(P);
11             si b et s_i ne correspondent pas alors
12                 retourner Non ;
13     i ← i + 1 ;
14 retourner est_vide(P);
```

Terminaison Le programme termine, car sa seule boucle while est en réalité une boucle for.

Complexité La complexité de l'algorithme est en $\mathcal{O}(n)$ car toutes les opérations des piles sont en $\mathcal{O}(1)$ avec les implémentations vues précédemment.

Correction On veut montrer que l'algorithme renvoie Oui si et seulement si s est bien parenthésé. Le principe de l'algorithme est que la pile P stocke des parenthèses ouvrantes pour lesquelles on n'a pas encore trouvé de parenthèse fermante. Autrement dit, en lisant $s_0 \dots s_{i-1}$ on a éliminé les parties bien-parenthésées et stocké le trop-plein de parenthèses ouvrantes dans P.

Cela nous pousse vers l'invariant de boucle suivant :

$$\text{I : “}s \text{ est bien-parenthésé} \Leftrightarrow \mathbf{mot}(P)s_i s_{i+1} \dots s_{n-1} \text{ est bien-parenthésé”}$$

avec $\mathbf{mot}(P) = x_{k-1} \dots x_1 x_0$, où x_0 est le sommet de P, x_1 l'élément sous le sommet, etc...

Montrons que la propriété ci-dessus est bien un invariant de boucle. On note P_k, i_k les valeurs de P et i après k tours de boucles. Les autres variables (s, n) sont constantes.

- Pour $k = 0$, $i_0 = 0$ et P_0 est vide, donc $\mathbf{mot}(P_0) = \varepsilon$ (le mot vide). Ainsi, $\mathbf{mot}(P_0)s_{i_0} s_{i_0+1} \dots s_{n-1} = s$ et la propriété I est trivialement vraie pour $k = 0$.
- Soit $k \in \mathbb{N}$ tel que $I(k)$ est vraie. On considère un $k+1$ passage.
  On a $i_{k+1} = i_k + 1$, et on distinque deux cas :
  - Si s_{i_k} est ouvrante, alors : $P_{k+1} = \boxed{\dfrac{s_{i_k}}{P_k}}$, et donc $\mathbf{mot}(P_{k+1}) = \mathbf{mot}(P_k)s_{i_k}$. donc :

    $$\begin{aligned} s \text{ est } \mathbf{BP} &\Leftrightarrow \mathbf{mot}(P_k)s_{i_k} s_{i_k+1} \dots s_{n-1} \text{ est } \mathbf{BP} && \text{(par HR)} \\ &\Leftrightarrow \mathbf{mot}(P_k)s_{i_k} s_{i_{k+1}} \dots s_{n-1} \text{ est } \mathbf{BP} && \text{(car } i_{k+1} = i_k + 1) \\ &\Leftrightarrow \mathbf{mot}(P_{k+1})s_{i_{k+1}} \dots s_{n-1} \text{ est } \mathbf{BP} && \text{(car } \mathbf{mot}(P_{k+1}) = \mathbf{mot}(P_k)s_{i_k}) \end{aligned}$$

    et donc la propriété est vraie à la fin du passage.
  - Sinon, s_{i_k} est fermante. Sans perte de généralité, supposons que s_{i_k} vaut `')'`.
    - Si P_k est vide, alors $\mathbf{mot}(P_k) = \varepsilon$ et donc $\mathbf{mot}(P_k)s_{i_k} \dots s_{n-1}$ n'est pas bien-parenthésé puisqu'il commence par une parenthèse. Donc, par HR, s n'est pas bien parenthésé, et l'algorithme renvoie bien Non.
    - Sinon : on note b le sommet de P_k. On a : $P_k = \boxed{\dfrac{b}{P_{k+1}}}$, et donc $\mathbf{mot}(P_k) = \mathbf{mot}(P_{k+1})b$.

      Donc :

      $$\begin{aligned} s \text{ est bien-parenthésé} &\Leftrightarrow \mathbf{mot}(P_k)s_{i_k} s_{i_k+1} \dots s_{n-1} \text{ est bien-parenthésé} \\ &\Leftrightarrow \mathbf{mot}(P_{k+1})b s_{i_k} s_{i_{k+1}} \dots s_{n-1} \text{ est bien-parenthésé} \end{aligned}$$

      Or, b est ouvrante, et s_{i_k} est fermante, donc $\mathbf{mot}(P_{k+1})b s_{i_k} s_{i_{k+1}} \dots s_{n-1}$ est bien-parenthésé si et seulement si les deux conditions suivantes sont réalisées :

      - ($\mathbf{mot}(P_{k+1})s_{i_{k+1}} \dots s_{n-1}$ est bien-parenthésé ;
      - s_{i_k} correspond à b (c'est à dire $s_{i_k} =$ `'('`).

      Ainsi, si s_{i_k} correspond à b, on a bien

      $$s \text{ est bien-parenthésé} \Leftrightarrow \mathbf{mot}(P_{k+1})s_{i_{k+1}} \dots s_{n-1} \text{ est bien-parenthésé}$$

      et l'invariant de boucle est bien préservé. Sinon, s n'est pas bien parenthésé (car il contient `[)`, et l'algorithme renvoie Non et est correct.

Nous avons donc montré que I est un IdB, et qu'en plus, si l'algorithme renvoie Non, alors le mot de base n'est pas bien parenthésé.
En particulier, à la sortie de la boucle, $i = n$ et donc s est bien parenthésé si et seulement si $\mathbf{mot}(P)$ l'est. Mais P ne contient que des parenthèses ouvrantes : donc s est bien parenthésé si et seulement si P est vide, ce qui correspond bien à la valeur renvoyée par l'algorithme.

3 File

La structure de file stocke des données selon le principe "Dernier arrivé, dernier sorti", ou ***LILO*** (Last In Last Out), ou encore ***FIFO*** (First In First Out) Ainsi, les données sont lues dans l'ordre où elles sont ajoutées,

Remarque 2

Visuellement, c'est une file d'attente au supermarché !

Exemple 6

Voici un exemple d'utilisation d'une file :

A Structure de données abstraite

Une file représente une suite finie d'éléments de type `T`, dont la taille peut varier. Une file a une tête et une queue, et ses éléments sont rangés, de telle sorte que l'on enfile les nouveaux éléments à la queue de la file, et que l'on défile les éléments par la tête de la file. Ses opérations sont :

— Créer une file vide : **file_vide**() ;

— Enfiler un nouvel élément x à la queue d'une file F : **enfiler**(F, x) ;

— Défiler l'élément à la tête d'une file F et le renvoyer : **defiler**(F) ;

— Déterminer si une file est vide : **est_vide**(F).

Exercice 4

Pour chaque opération, dire si c'est un accesseur, un transformateur ou un constructeur.

B Implémentation par liste chaînée

On utilise presque les mêmes structures que pour les piles, mais on doit stocker deux pointeurs : celui de la tête et celui de la queue :

```
typedef struct maillon {
  T elem;
  struct maillon* suiv; // de la tete vers la queue
} maillon_t;

typedef struct file_{
  maillon_t* tete;
  maillon_t* queue;
} file_t;
```

Attention, le sens des pointeurs `suiv` va **contre** le sens de la file.

Exemple 7

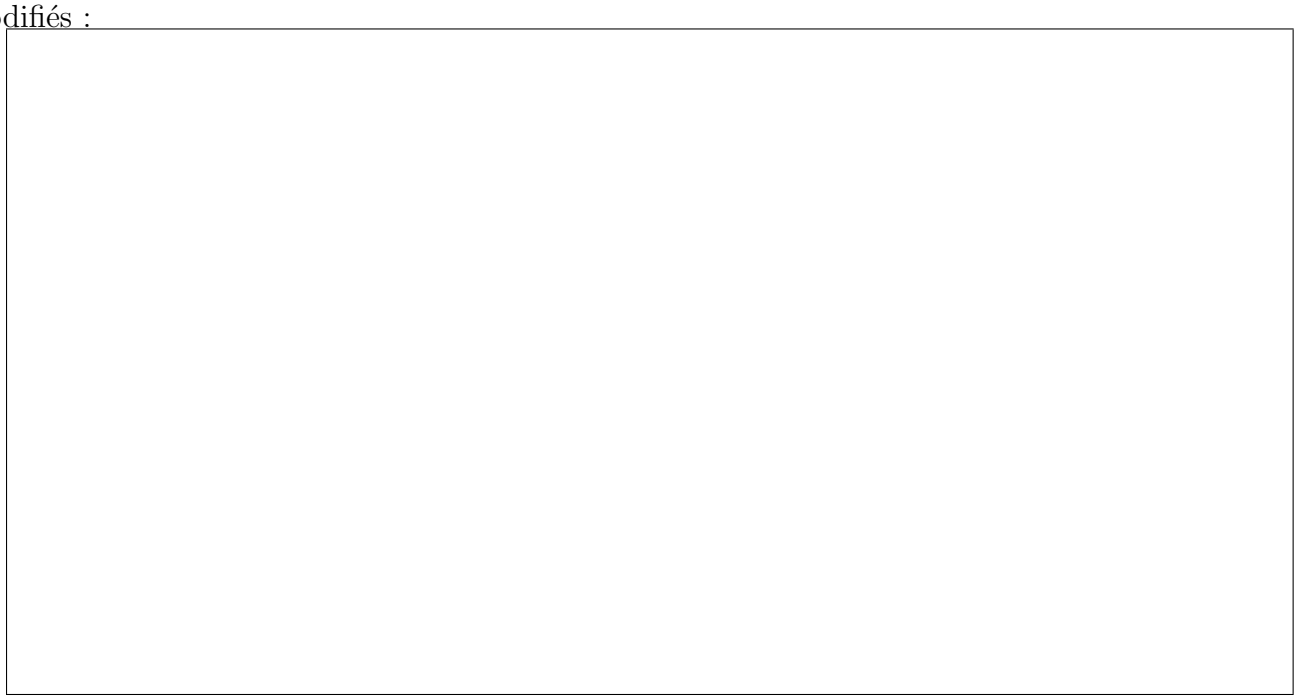
Voici une file abstraite et le schéma qu'on aurait en mémoire avec une pile concrète par liste chaînée :

Pour créer une file vide et déterminer si une file est vide, on procède exactement comme pour les piles :

```
file_t* file_vide(){
  file_t* f = malloc(sizeof(file_t));
  f->tete = NULL;
  f->queue = NULL;
}

bool est_file_vide(file_t* f){
  return (f->tete == NULL); // ou bien: return (f->queue == NULL);
}
```

Enfilage Enfiler un élément est un peu technique. Selon si la file était vide ou non, le schéma est un peu différent. Dans les deux cas, on crée nouveau maillon qui devient la queue de la file, mais si la file était vide alors ce nouveau maillon devient aussi la tête de file !

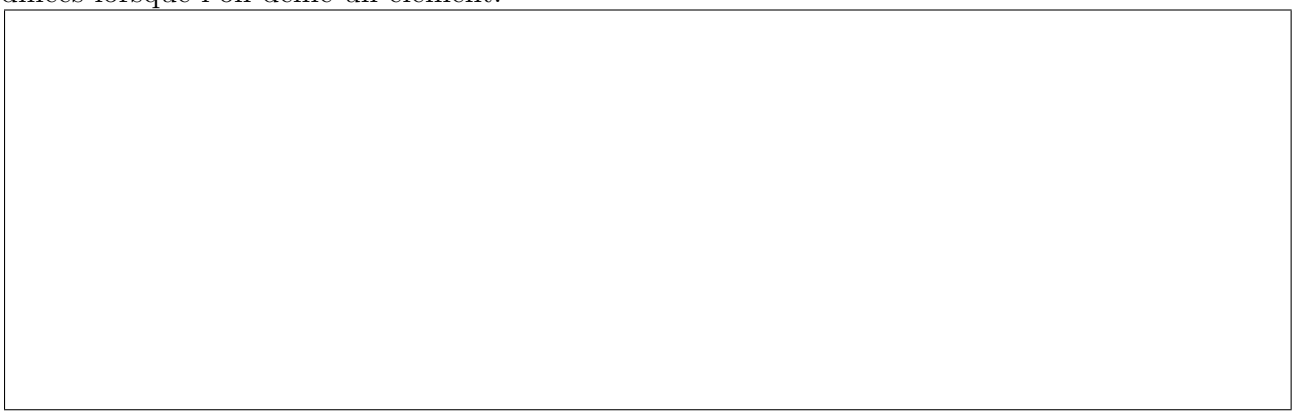
Dessinons les deux situations, en notant à chaque fois les liens qui sont ajoutés, supprimés, modifiés :



On en déduit le code C suivant :

```c
void enfiler(file_t* f, T x){
  maillon_t* anc_q = f->queue; // ancienne queue

  // création de la nouvelle queue de file
  maillon_t* nouv_q = malloc(sizeof(maillon_t));
  nouv_q->elem = x;
  nouv_q->suivant = NULL;

  // mise à jour de la queue
  f->queue = nouv_q;
  if (anc_q == NULL){ //la file était vide: mettre à jour la tete
    f->tete = nouv_q;
  } else { // anc_q a maintenant un maillon suivant: nouv_q
    anc_q->suiv = nouv_q
  }
}
```

Défilage A nouveau, faisons des schémas pour étudier comment les différentes liaisons sont modifiées lorsque l'on défile un élément.



On obtient le code C suivant :

```
1  T defiler(file_t* f){
2    assert(!est_file_vide(f));
3
4    maillon_t* anc_tete = f->fete; // ancienne tete
5    T res = anc_tete->elem;
6
7    maillon_t* nouv_tete = ancienne_tete->suivant;
8    f->tete = nouv_tete;
9
10   if (f->tete == NULL){ // la file est vide
11     f->queue = NULL;
12   }
13   return res;
14 }
```

Exercice 5

Reprendre les deux fonctions précédentes si les pointeurs au sein de la chaîne vont maintenant **de la queue vers la tête**. Que se passe t-il ?

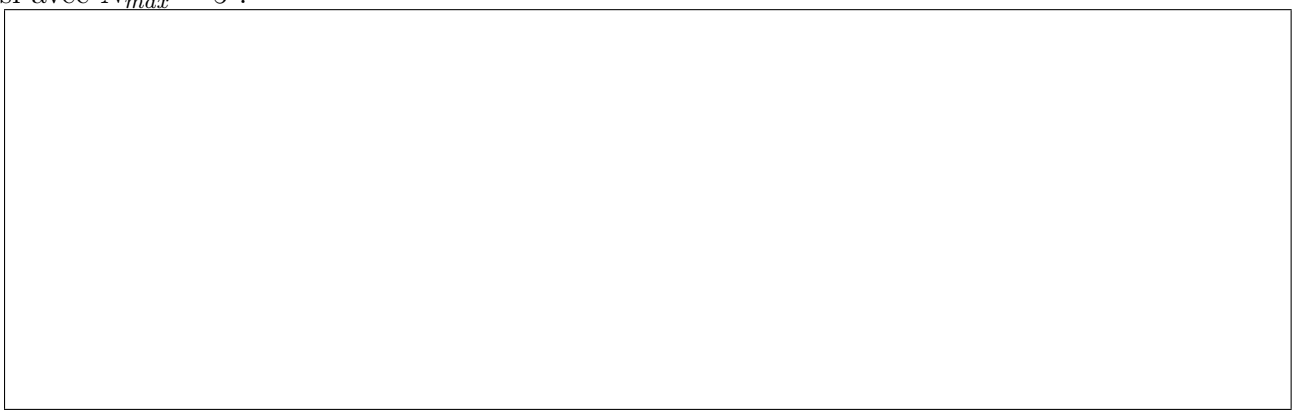
L'affichage et la libération de mémoire se font de manière analogue à ce que l'on a fait pour les piles.

C Implémentation par tableau

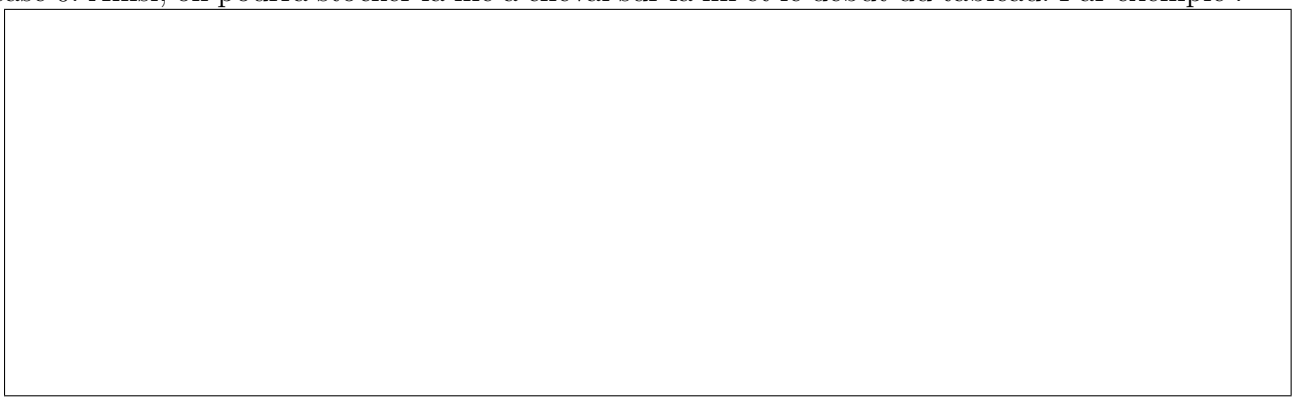
Comme pour les piles, on peut implémenter une file en utilisant un tableau contenant les valeurs de la file, de la tête vers la queue. Plus précisément, on utilise la structure suivante :

```c
#define Nmax 10000
typedef struct file {
  int queue; //indice de la prochaine case à remplir si l'on enfile
  int nb_elem; // nombre d'éléments dans la file
  T tab[Nmax];
} file_t;
```

Dans une file `f` de ce type, les cases utiles sont les cases d'indice `f->queue - i` avec i allant de 1 à `f->nb_elem`. La valeur en queue de file est donc à la case `f->queue - 1`, et la valeur en tête de file est à la case `f->queue - f->nb_elem`. Simulons l'évolution d'une file implémentée ainsi avec $N_{max} = 5$:



On voit donc qu'en utilisant directement cette implémentation, on se heurte à un problème : la file peut devenir inutilisable alors qu'elle n'est pas pleine. On va donc travailler dans des **tableaux cycliques**. Autrement dit, on considère que la case $N_{max} - 1$ du tableau est collée à la case 0. Ainsi, on pourra stocker la file à cheval sur la fin et le début du tableau. Par exemple :



Notons que cela revient à prendre les indices modulo N_{max}.

Pour créer une file vide, et déterminer si une file est vide :

```c
file_t* file_vide(){
  file_t* f = malloc(sizeof(file_t));
  f->nb_elem = 0;
  f->queue = 0;
  return f;
}

bool est_vide(file_t* f){
  return (f->nb_elem == 0);
}
```

L'opération pour enfiler est similaire à celle d'empilage de la pile, mais on travaille modulo la taille du tableau :

```
1 void enfiler(file_t* f, T x){
2   assert(f->nb_elem < Nmax);
3   f->tab[f->queue] = x;
4   f->queue = (f->queue + 1) % Nmax;
5   f->nb_elem++;
6 }
```

Pour défiler, il faut commencer par trouver l'indice de la case correspondant à la tête de file :

```
1 T defiler(file_t* f){
2   assert(!est_vide(f));
3   T res = f->tab[(Nmax + f->queue - f->nb_elem)%Nmax];
4   f->nb_elem--;
5   return res;
6 }
```

Exercice 6

On propose une variante de l'implémentation par tableau cyclique, où l'on stocke un indice de queue et un indice de tête :

```
1 #define Nmax 10000
2 typedef struct file {
3   int queue; //indice de la prochaine case à remplir si l'on enfile
4   int tete; // indice de la case à défiler
5   T tab[Nmax];
6 } file_t;
```

Implémenter les opérations de file avec cette SDC. Que remarquez-vous ?

4 Tableau redimensionnable

A Principe

Une limite des implémentations par tableau des piles et des files est que l'on a une taille limite à nos structures. On s'intéresse à l'implémentation d'une structure que l'on appellera **vecteur**, où les opérations sont celles des tableaux (lecture d'une case, écriture d'une case) et celles d'une pile (ajouter ou supprimer un élément à la fin), sans limite de taille L'implémentation sera facilement adaptable aux implémentations des piles et files par tableau.

L'idée est de prendre un tableau d'une taille arbitraire (4 dans le code ci-dessous), et de **redimensionner** ce tableau à chaque fois qu'il est rempli.

```
1  typedef struct vect{
2    T* tab; // éléments
3    int taille_max; // nombre de cases allouées pour tab
4    int nb_elem; // nombre de cases utilisées
5  } vect_t;
6
7  vect_t* creer_vecteur(){
8    vect_t* v = malloc(sizeof(vect_t));
9    v->nb_elem = 0;
10   v->tab = malloc(4 * sizeof(T));
11   v->taille_max = 4;
12   return v;
13 }
```

Lorsque l'on ajoute un élément à la fin du vecteur, s'il ne reste plus de place, i.e. si `nb_elem == taille_max`, on alloue un nouveau tableau, plus grand, et on y recopie tous les éléments.

Exercice 7

En décidant arbitrairement d'augmenter la taille de 10 à chaque fois qu'on réalloue, représenter l'évolution du tableau si l'on ajoute les éléments 1, 2, 3, 4, 5, etc ... à partir d'un vecteur vide. Même question si l'on double la taille à chaque fois qu'on réalloue.

En C, la fonction `realloc` sert à réallouer de la mémoire. Par exemple :

```
1  int* p = malloc(10 * sizeof(int));
2  ...
3  p = realloc(p, 15*sizeof(int));
```

Dans le code précédent, la dernière ligne désalloue les 10 cases int réservées à la ligne d'au dessus, et alloue 15 nouvelles cases. Elle recopie dans les nouvelles cases le contenu des anciennes. Cette opération prend un temps linéaire en l'ancienne taille mémoire car elle doit recopier chaque élément un à un.

On peut donc implémenter l'opération d'ajout comme suit :

```
1  void ajouter(vect_t* v, T x){
2    // redimensionner si nécessaire
3    if (v->nb_elem == v->taille_max){
4      int nouvelle_taille = ???;
5      v->tab = realloc(v->tab, nouvelle_taille);
6      v->taille_max = nouvelle_taille;
7    }
8    v->tab[v->nb_elem] = x;
9    v->nb_elem++;
10 }
```

B Performances : analyse de complexité amortie

Il reste à choisir de combien augmenter la taille lorsque l'on redimensionne. Notons que dans tous les cas, la complexité de l'ajout dans un vecteur de taille n sera en $\Omega(n)$ car dans le pire cas (c'est à dire en cas de redimensionnement), il faut recopier tous les éléments. Nous allons étudier la **complexité amortie** de l'ajout : si l'on effectue n ajouts depuis un vecteur vide, en notant le coût total C_n, on appellera **coût amorti** de l'ajout la valeur moyenne $\frac{C_n}{n}$. La complexité amortie d'une opération permet donc d'évaluer son coût moyenné sur un grand nombre d'utilisations. L'idée est que même si certains cas peuvent avoir une très mauvaise complexité, dans un programme long, ils arriveront rarement et seront compensés par un grand nombre d'opérations rapides.

Première idée A chaque ajout, on augmente de 1 la taille du tableau. Alors, à partir du premier redimensionnement, chaque ajout cause un redimensionnement et coûte $\Theta(i)$ où i est la taille du vecteur au moment de l'ajout. Ainsi, si l'on ajoute n éléments d'affilée, la complexité totale est en $\Theta(n^2)$.

Deuxième idée Plutôt que de réallouer un tableau avec une seule place de plus, on fixe $K \in \mathbb{N}^*$ et on alloue le tableau par blocs de K cases : Initialement, le tableau possède K cases, puis lorsque l'on doit ajouter un $K+1$-ème élément, on réalloue un tableau de taille $2K$, puis de taille $3K$, etc...

Ainsi, si l'on fait plusieurs ajouts, alors les K premiers se feront en temps constant, puis le $K+1$-ème en temps $K+1$, puis les $K-1$ suivants en temps constant, puis le $2K+1$-ème en temps $2K+1$, etc...

Le coût total combiné de n ajouts dans un vecteur vide est de l'ordre de :

$$\begin{aligned}
& \textstyle\sum_{i=0}^{n-1}(i+1 \text{ si } i \text{ est multiple de } K,\ 1 \text{ sinon}) \\
= & \sum_{i=0}^{n-1} 1 + \sum_{i=0}^{n-1}(i \text{ si } i \text{ multiple de } K,\ 0 \text{ sinon}) \\
= & \sum_{i=0}^{n-1} 1 + \sum_{j=0}^{\lfloor \frac{n-1}{K} \rfloor} Kj \\
= & \ n + K\frac{\lfloor \frac{n-1}{K} \rfloor (\lfloor \frac{n-1}{K} \rfloor + 1)}{2} \\
= & \ \mathcal{O}(n^2)
\end{aligned}$$

Donc, en moyenne, un ajout prend de l'ordre de $\mathcal{O}(n)$, comme pour la première idée. Notons que la constante se cachant dans le $\mathcal{O}$ est beaucoup plus faible.

Troisème idée On double la taille du tableau à chaque fois. Si l'on calcule à nouveau le coût total de n ajouts successifs, on obtient :

$$\begin{aligned}
& \textstyle\sum_{i=0}^{n-1}(i+1 \text{ si } i \text{ puissance de 2},\ 1 \text{ sinon}) \\
\leq & \sum_{i=0}^{n-1} 1 + \sum_{j=0}^{\lfloor \log_2(n-1) \rfloor} 2^j \\
\leq & \ n + 2^{\lfloor \log_2(n) \rfloor + 1} \\
\leq & \ 3n
\end{aligned}$$

Donc, le coût moyen d'un ajout est constant ! On dit que l'ajout a un coût **amorti** $\mathcal{O}(1)$.

Notons que ce faible coût en temps n'est pas gratuit : cette méthode nécessite de réserver jusqu'au double de la taille réelle du tableau. Le coût **spatial** est élevé, si l'on programme sur un système où la mémoire est précieuse, comme un micro-ordinateur, il faudra faire un compromis entre le coût temporel et le coût spatial.

Dans l'implémentation classique C de Python, c'est ce système de tableaux redimensionnables qui est utilisé pour implémenter les listes. Si l'on inspecte la taille mémoire d'une liste après plusieurs ajouts, on obtient le graphe suivant :

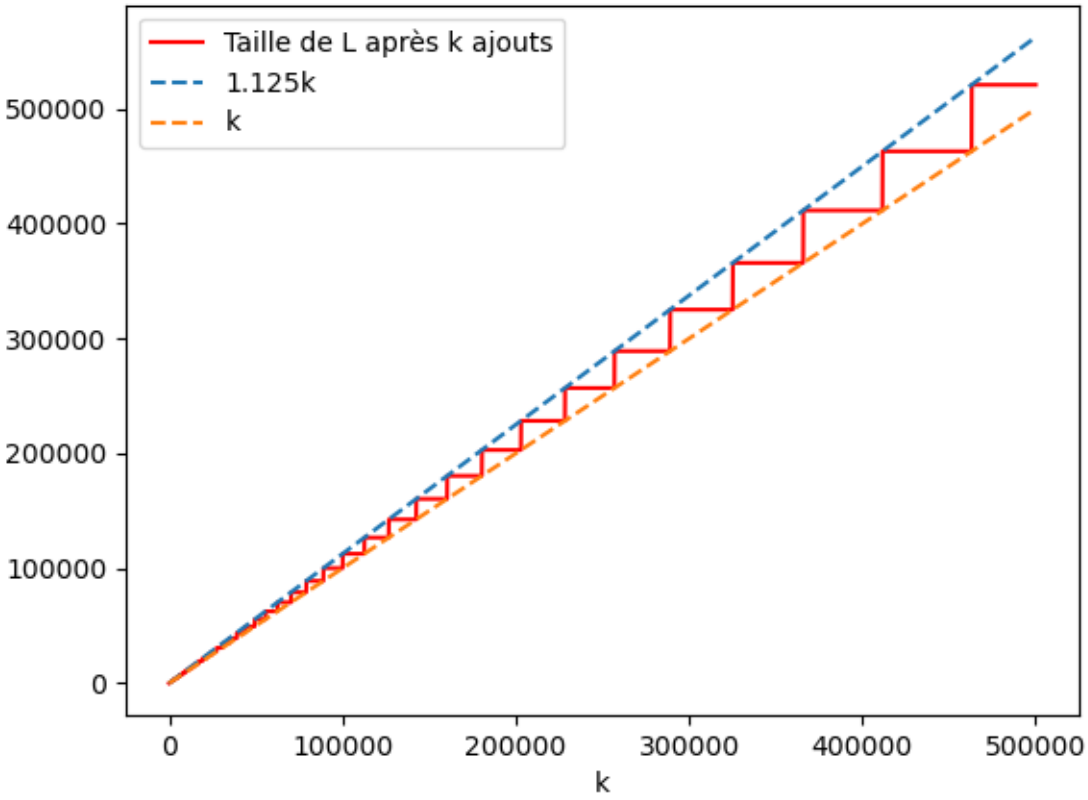


FIGURE 1 – Taille d'une liste Python L après k ajouts.

On voit qu'à chaque fois qu'un ajout dépasse la taille mémoire, la liste est réallouée en multipliant la taille par 1.125. On peut montrer avec les même calculs que précédemment que ceci donne des ajouts en temps amorti constant.

Exercice 8

Imaginer d'autres stratégies de redimensionnement (multiplier la taille par un réel α autre que 2, mettre la taille au carré, multiplier la taille par sa racine, etc...), et étudier leurs performances du point de vue du temps et de l'espace.

5 Dictionnaires

Les dictionnaires font partie des structures de données les plus importantes en informatique. Un dictionnaire est comme un tableau, indexé par des éléments d'un type quelconque plutôt que par des entiers consécutifs.

Exemple 8

Python propose un type natif de dictionnaires, que l'on peut manipuler avec la même syntaxe que les tableaux :

```
1 cri = dict() # associe à chaque animal son cri
2 cri["chat"] = "miaou"
3 cri["chien"] = "waf"
4 cri["carpe"] = ""
5
6 s = cri["chat"]
```

On fixe K et V deux ensembles. Un dictionnaire stocke une fonction de K dans V partielle (c'est à dire qu'elle n'est pas forcément définie sur tout K). Autrement dit, un dictionnaire stocke des couples $(k, v) \in K \times V$, où l'on appelle k la clé et v la valeur, tels que pour un $k \in K$ donné, il y a au plus un couple dans le dictionnaire dont k est la clé. Un dictionnaire permet de chercher la valeur associée à une clé, et de modifier cette valeur. On considère les opérations suivantes :

- **dict**() crée un dictionnaire vide ;
- **ajout**(D, k, v) associe la valeur v à la clé k dans le dictionnaire D.
- **contient**(D, k) détermine si la clé k est dans le dictionnaire D.
- **recherche**(D, k) renvoie la valeur associée à la clé k dans le dictionnaire D. k doit être une clé valable.
- **supprime**(D, k) supprime la clé k du dictionnaire D, ainsi que la valeur qui y était associée.

Lorsque l'on associe une valeur $v \in V$ à une clé $k \in K$, si k est déjà dans le dictionnaire, alors on **écrase** sa valeur associée pour y écrire v.

Exercice 9

Donner le type de chaque opération (**C**onstructeur, **A**ccesseur, **T**ransformateur).

En pseudo-code, on utilisera parfois une notation proche des tableaux, comme celle de Python : si D est un dictionnaire et k une clé, alors $D[k]$ est la valeur associée à k dans D, i.e. **recherche(D, k)**. De même, écrire $D[k] \leftarrow v$ revient à écrire **ajout**(D, k, v).

A Exemples d'applications des dictionnaires

Nombre d'occurrences, élément majoritaire Étant donné un tableau T d'éléments, on se demande quel est l'élément majoritaire, i.e. l'élément ayant le plus d'occurrences dans T. Un premier algorithme naïf consiste à regarder, pour chaque case, le nombre de cases qui contiennent la même valeur, et de garder en mémoire la meilleure case vue jusqu'à maintenant :

Algorithme 2 : `majoritaire`(T, n)

Entrée(s) : T tableau de taille n
Sortie(s) : $x \in T$ tel que $\{i \in [\![0, n-1]\!] \mid T[i] = x\}$ est de cardinal maximal

```
1  x_m ← Rien // élément le plus fréquent vu jusqu'à maintenant
2  o_m ← 0 // nombre d'occurrences de x_m dans T
3  pour i = 0 à n − 1 faire
4      o ← 0// nombre d'occurrences de T[i] dans T
5      pour j = 0 à n − 1 faire
6          si T[i] = T[j] alors
7              o ← o + 1;
8      si o > o_m alors
9          o_m ← o;
10         x_m ← T[i];
11 retourner x_m
```

La complexité de cet algorithme est $\mathcal{O}(n^2)$. En commençant par trier le tableau, on pourrait améliorer cette complexité en $\mathcal{O}(n \log n)$.

Exercice 10

Expliquer comment trouver un élément majoritaire dans un tableau trié plus efficacement qu'avec l'algorithme `majoritaire`(T, n).

Avec un dictionnaire, on peut trouver une solution encore plus efficace. L'idée est de créer un dictionnaire dont les clés sont les éléments du tableau, tel que la valeur associée à un élément est son nombre d'occurrences dans T. On commence par remplir ce dictionnaire en lisant une fois le tableau, puis on regarde la clé du dictionnaire ayant la plus grande valeur :

Algorithme 3 : Élément majoritaire

Entrée(s) : T tableau de taille n
Sortie(s) : $x \in T$ tel que $\{i \in [\![0, n-1]\!] \mid T[i] = x\}$ est de cardinal maximal

```
// Calcul du dictionnaire des occurrences
O ← Dictionnaire vide // x ↦ nombre d'occurrences de x
pour i = 0 à n − 1 faire
    si T[i] n'est pas une clé de O alors
        O[T[i]] ← 0;
    O[T[i]] ← O[T[i]] + 1;
// Recherche de la clé de valeur maximale
x_m ← Rien ;
o_m ← 0;
pour x clé de D faire
    si O[x] > o_m alors
        o_m ← o;
        x_m ← x;
retourner x_m
```

Alors, le en notant $A(i)$ le coût de l'ajout dans un dictionnaire à i éléments et $R(i)$ le coût de la recherche dans un dictionnaire à i éléments, le coût total de l'algorithme est :

$$\sum_{i=0}^{n-1}(R(i) + A(i)) + nR(n)$$

Nous allons voir une implémentation des dictionnaires où toutes les opérations sont en temps **constant** $\mathcal{O}(1)$ [1]. L'algorithme précédent est alors en $\mathcal{O}(n)$: on a gagné un facteur n par rapport à l'algorithme naïf !

1. En moyenne.

Implémentation par liste chaînée On commence par une implémentation plus naïve, où la recherche sera en complexité linéaire.

L'idée de cette implémentation est d'avoir une liste doublement chaînée dont les maillons contiennent des couples (k, v) clé-valeur :

```
typedef struct maillon{
  K key;
  V value;
  struct maillon* suiv;
  struct maillon* prec;
} maillon_t;

typedef struct dict{
  maillon_t* tete;
} dict_t;
```

Pour faire une recherche, une modification ou une suppression, le principe est le même : faire une recherche linéaire en parcourant les maillons un à un. Par exemple :

```
/* Stocke dans *result la valeur associée à K dans d,
   et renvoie un booléen indiquant si K a été trouvée */
bool recherche(dict_t* d, K key, V* result){
  for (maillon_t* m = d->tete; m != NULL; m = m->suiv){
    if (m->key == K){
      *result = m->value;
      return true;
    }
  }
  return false;
}
```

6 Dictionnaires : tables de hachage

Les dictionnaires sont une généralisation des tableaux. Si l'ensemble K des clés est de la forme $[\![0, m-1]\!]$ pour un $m \in \mathbb{N}$, alors un dictionnaire ayant K comme clés est presque exactement un ***tableau***. La différence est que pour un dictionnaire, il faut pouvoir indiquer si une case est utilisée ou pas, i.e. si l'indice de la case est présent dans le dictionnaire ou pas.
Pour représenter un tel dictionnaire, on pourrait donc utiliser un tableau dans lequel on stocke des éléments de l'ensemble $V \cup \{\textbf{NIL}\}$, où **NIL** est un élément particulier signifiant "pas de valeur".
Supposons maintenant que l'ensemble des clés K est fini de cardinal m, mais que ce n'est pas forcément un intervalle de la forme $[\![0, m-1]\!]$ (par exemple l'ensemble des prénoms de la classe). Si l'on dispose d'une fonction bijective de numérotation $f : K \to [\![0, m-1]\!]$, on peut implémenter un dictionnaire en utilisant un tableau T comme décrit au dessus, en stockant chaque clé k dans la case $T[i]$ avec $i = f(k)$.

— Création d'un dictionnaire vide : renvoyer un tableau T de taille m, avec $T[i] = \textbf{NIL}$ pour $i \in [\![0, m-1]\!]$.
— Écriture d'un couple (k, v) : Effectuer $T[f(k)] \leftarrow v$.
— Recherche de la valeur associée à k : Renvoyer $T[f(k)]$.
— Suppression de la clé k : Effectuer $T[f(k)] \leftarrow \textbf{NIL}$.

Cependant, cette solution ne fonctionne plus dès que l'ensemble K des clés possibles est trop grand. Prenons par exemple comme clés les chaînes de caractères de taille au plus 10. En utilisant le code ASCII, on peut voir une telle chaîne comme un entier de 10 chiffres en base 256, ce qui permettrait de numéroter l'ensemble des chaînes. On pourrait implémenter un dictionnaire comme expliqué plus haut, mais cela nécessiterait de réserver un tableau de taille $256^{10} \approx 10^{24}$, alors qu'en pratique seul une infime fraction du tableau serait réellement utile...

Le principe des tables de hachage est de ramener n'importe quel ensemble de clé à un ensemble de la forme $[\![0, m-1]\!]$ pour un $m \in \mathbb{N}$, avec m raisonnablement petit.

Définition 4

Soit K un ensemble de clés, et $m \in \mathbb{N}^*$. Une fonction de hachage est une fonction $h : K \to [\![0, m-1]\!]$ **pas forcément bijective**.
Pour $k \in K$, on appelle $h(k)$ le **hash** de k.

Si h est une fonction de hachage à valeurs dans $[\![0, m-1]\!]$, alors on peut implémenter un dictionnaire à clés dans K par un tableau de taille m. Pour insérer, modifier ou supprimer une clé $k \in K$ donnée, on calcule $i = h(k)$, et on regarde à la i-ème case du tableau. On appelle un tel tableau une ***table de hachage***.

Exemple 9

Soit K l'ensemble des mots sur l'alphabet `a, b, ..., z`. On considère une partie de jeu à plusieurs joueurs. On voudrait représenter le résultat de la partie par un dictionnaire qui à chaque joueur associe son score.
On considère la fonction de hachage $h_0 : K \to [\![0, 25]\!]$ qui à chaque chaîne de caractère associe la première lettre de son prénom (vu comme un entier entre 0 et 25). Cette fonction permet d'implémenter une table de hachage basique à 25 cases. Par exemple, pour les clés suivantes :

`adi`, `guigui`, `max`, `oli`

On obtiendrait la table de hachage suivante :

Exercice 11

On prend maintenant comme fonction de hachage

$$h_5(k) = \sum_{x \text{ lettre de } k} x \mod 5$$

(On considère toujours que $a = 0, b = 1, \ldots, z = 25$). Avec le même ensemble de clés qu'au dessus, représenter la nouvelle table de hachage :

On peut donc se retrouver dans une situation ou deux clés sont hachées vers la même valeur. En fait, dès que la taille m de la table est strictement inférieure à $|K|$, le lemme des tiroirs dit que l'on trouvera forcément deux clés ayant le même hash, car on n'aura pas injectivité de la fonction de hachage.

Définition 5

Soit $h : K \to [\![0, m-1]\!]$ une fonction de hachage et $k \neq k' \in K$. On dit que k et k' forment une ***collision*** si $h(k) = h(k')$

De plus, si l'on reprend la première fonction de hachage, celle qui prend seulement la première lettre du prénom, on remarque que peu de prénoms commencent par k, w, x, y, z par rapport aux autres lettres. Donc, on est non-seulement sur d'avoir des collisions, mais on sait même que certaines cases seront très demandées, et d'autres pas du tout.
On est donc confrontés à deux problèmes :

— Que faire en cas de collision ?

— Comment choisir une fonction de hachage qui évite le plus possible les collisions, et les répartit le mieux possible ?

On voudrait idéalement que la valeur de hachage fabriquée par la fonction soit "aléatoire", c'est à dire qu'on ne puisse pas a priori prévoir son comportement global. Si on reprend la deuxième fonction vue en exemple, qui fait la somme des lettres et prend le résultat modulo m, on peut imaginer qu'il n'y aura pas de case privilégiée : a priori, il n'y a pas de raison particulière pour que beaucoup de prénoms aient les sommes de leurs lettres congrues à 3 modulo 19.

Dans la suite, on considère T un tableau de taille m utilisé comme table de hachage pour un dictionnaire. On note $\{k_0, \dots, k_{n-1}\}$ les clés utilisées, n est donc la taille du dictionnaire (ne pas confondre la taille du dictionnaire et la taille du tableau utilisé pour implémenter le dictionnaire).

Il existe deux classes de méthodes pour résoudre les collisions dans une table de hachage :

— Les méthodes de résolution par chaînage : les éléments dont les clés ont la même valeur par la fonction de hachage sont rangés dans une seule liste chaînée. On parle de hachage indirect.

— Les méthodes de résolution par calcul : Lorsqu'il y a une collision, on calcule un nouvel emplacement où stocker la nouvelle clé. On parle de hachage direct.

Dans ce cours, on se contentera d'étudier la gestion par chaînage.

A Gestion des collisions par chaînage

Le principe est de stocker dans chaque case de la table de hachage une **liste chaînée** de tous les éléments qui y ont été mis par hachage. Ainsi, $T[i]$ contiendra la liste chaînée de toutes les clés $k \in \{k_0, \dots, k_{n-1}\}$ telles que $h(k) = i$. Les cases de T sont appelées des ***alvéoles*** (et parfois *buckets* en anglais).

On se ramène donc à la manipulation de dictionnaires par liste chaînée étudiée au début de cette section. Pour insérer, supprimer, modifier ou rechercher dans une table de hachage, on calcule le hash de la clé k à traiter notons le i, et on effectue l'opération sur la liste chaînée stockée dans l'alvéole i du tableau.

On implémente donc les opérations comme suit :

— Création d'un dictionnaire vide : Renvoyer un tableau T de taille m, avec chaque case contenant une liste vide. Les maillons de ces listes stockeront des couples clé-valeur.

— Écriture d'un couple (k, v) :

Algorithme 4 : Écriture dans une table de hachage avec chaînage

Entrée(s) : T table de hachage, h fonction de hachage à utiliser, $(k, v) \in K \times V$ couple à insérer

```
1 i ← h(k);
2 m ← recherche(T[i], k) // Maillon contenant la clé k
3 si m = NULL alors
4     ajouter(T[i], (k, v));
5 sinon
6     m.v = v;
```

— Recherche, Suppression : même principe

Exemple 10

On prend $m = 9$, et on considère les clés $k_0, \dots, k_{12}$ et leurs valeurs de hachage :

clé	:	k_0	k_1	k_2	k_3	k_4	k_5	k_6	k_7	k_8	k_9	k_{10}	k_{11}	k_{12}
hash	:	3	1	4	1	4	1	5	9	2	6	5	3	5

Alors la table de hachage ressemblera à :

Le pire cas pour ces opérations est atteint lorsque toutes les associations sont stockées dans la même alvéole. La complexité est alors en $\mathcal{O}(n)$. En pratique, lorsque l'on choisit une bonne fonction de hachage, les clés sont bien réparties, et le coût moyen des différentes opérations est $\mathcal{O}(1+\alpha)$, où $\alpha = \frac{n}{m}$ est le taux de remplissage, i.e. la taille moyenne des listes chaînées stockées dans les alvéoles.

Si l'on peut estimer à l'avance le nombre maximal n de clés que contiendra le dictionnaire, on peut fixer $m = n$ à la création et donc avoir des opérations en $\mathcal{O}(1)$ en moyenne. On peut également redimensionner le dictionnaire au cours de l'exécution, afin de contrôler α. Le taux de remplissage représente un ***compromis temps-mémoire*** : Si α est très petit, alors la table est peu remplie, donc il y a peu de collisions et les opérations sont rapides, mais la mémoire utilisée est très supérieure à l'espace nécessaire. Si α est grand, alors le tableau est très rempli, avec de nombreuses collisions, et les opérations sont lentes, mais le gaspillage de mémoire est minime.

En pratique, en redimensionnant et en gardant α borné convenablement, on obtient des bonnes performances en mémoire et en temps. Dans la plupart des cas pratiques, on peut considérer que les tables de hachage ont des complexités en temps constant : c'est une structure extrêmement efficace !

Par exemple, dans l'implémentation en C de Python, les dictionnaires sont implémentés par des tables de hachage avec gestion des collisions par chaînage. Si on regarde dans le code source [2], on peut voir que la taille des tables n'est pas fixée : elle est de 16 initialement, et le taux de remplissage α est gardé entre 10% et 50%, en redimensionnant la table lorsqu'elle est trop ou pas assez remplie.

2. github.com/python/cpython/blob/main/Python/hashtable.c

B Choix d'une fonction de hachage

Exemple 11

Si K est un ensemble de chaînes de caractères, on peut considérer la fonction :

$$h : k = k_0 k_1 \dots k_{l-1} \mapsto \sum_{i=0}^{l-1} k_i 256^i \mod m$$

Autrement dit, on considère, comme évoqué plus haut, que k représente un entier en base 256, et on prend le résultat modulo m.

Un inconvénient de cette fonction est que si m est pair, alors k et $h(k)$ auront toujours la même parité. Par exemple, les mots finissant par 'e' seront toujours hachés vers les cases d'indice impair, car 'e' vaut 101 en ASCII. De manière plus générale, si m est multiple de 2^p, alors les p derniers bits de $h(k)$ seront les mêmes que k.

On souhaite trouver des fonctions de hachage étant "assez aléatoires". De bons critères (liste non exhaustive) pour une fonction de hachage sont :

— Peu de collisions, et difficile de les trouver

— Fait intervenir tous les bits de la représentation de manière équitable

— Temps d'exécution court

Il existe de très nombreux algorithmes de hachage [3] dont certains servent plutôt en cryptographie que pour les tables de hachage.

La méthode de l'exemple précédent, consistant à considérer la clé comme un entier, et à prendre le reste modulo m, s'appelle méthode par ***division***.

3. en.wikipedia.org/wiki/ List_of_hash_functions

C Opérations additionnelles

On peut vouloir munir la structure abstraite de dictionnaire d'opérations supplémentaires. Par exemple, si l'ensemble K des clés est muni d'un ordre, étant donné une borne $k_0 \in K$, on pourrait avoir une opération permettant d'obtenir la liste des couples $(k, v) \in D$ tels que $k < k_0$.

De manière plus générale, on voudrait être capable d'***itérer*** sur les valeurs d'un dictionnaire D, soit dans un ordre arbitraire, soit selon un ordre précis défini sur K, soit par ordre d'insertion des clés-valeurs, etc....

L'implémentation des dictionnaires par liste chaînée permet de réaliser facilement les deux premières opérations, car la structure chaînée permet naturellement de parcourir les entrées du dictionnaire une à une. Cependant, pour les tables de hachage, la position des couples dans la table n'est pas corrélée à l'ordre d'insertion, où à un quelconque ordre sur K. Si l'on veut chercher une clé vérifiant une certaine propriété mais sans connaître sa valeur, on doit vérifier **chaque** alvéole une par une :

Algorithme 5 : `cle_inferieure`(T, k_0)

Entrée(s) : T table de hachage par chainage à m alvéoles, k_0 valeur limite
Entrée(s) : Une clé de T plus petite que k_0, ou NIL s'il n'y en a pas

```
1 pour i = 0 à m − 1 faire
2     m ← tete(T[i]);
3     tant que m ≠ NULL faire
4         si m.k < k_0 alors
5             retourner k
6         m ← suivant(m);
7 retourner NIL
```

La boucle extérieure permet d'itérer sur les alvéoles, et la boucle intérieure permet de parcourir l'alvéole $T[i]$.

Ce schéma algorithmique, consistant à parcourir chaque liste chaînée de la table, permet de répondre à de nombreuses requêtes : afficher le contenu du dictionnaire, renvoyer la liste des clés, la liste des valeurs, copier un dictionnaire, etc...

En utilisant d'autres types de structures, les ***arbres*** (voir chapitre 8), on peut obtenir des dictionnaires où les couples (clé, valeur) sont stockés d'une manière directement liée à l'ordre des clés. Avec ces implémentations, certaines requêtes (trouver la clé minimale, renvoyer les clés dans un certain intervalle), se font avec une bien meilleure complexité que les tables de hachage.