

Informatique MP2I

Table des matières

1	Listes chaînées	1
1.1	C	1
1.2	OCaml	1
1.2.1	Type complexe	1
1.2.2	Le module <code>List</code>	1
2	<i>Tablistes</i>	3
2.1	C	3
2.2	OCaml	5

1 Listes chaînées

1.1 C



Définition d'un type

```
struct cell {
    int value;
    struct cell* next;
};
typedef struct cell int_list;
/* Cette définition de listes chaînées ainsi que la majorité des
   fonctions à suivre s'adapte également pour les autres types */
```

Fonction à implémenter : longueur d'une liste

```
int length(int_list* lst) {
    if (lst == NULL) {
        return 0;
    }
    return 1 + length(int_list->next);
}
```

Description	Renvoie la longueur de la liste
Complexité	$\Theta(n)$

Fonction à implémenter : ajoute un élément à gauche d'une liste

```
void add_l(int elem, int_list* lst) {
    list_int* new_p = (list_int*)malloc(n*sizeof(list_int));
    new_p->next = lst;
    new_p->value = elem;
    return new_p;
}
```

Description	Ajoute un élément à gauche de la liste
Complexité	$\Theta(1)$

1.2 OCaml



1.2.1 Type complexe

1.2.2 Le module List

Fonction disponible : List.length

Signature	<code>'a list -> int</code>
Description	Renvoie la longueur de la liste
Complexité	$\Theta(n)$

Implémentation

```
let rec length = function
  | [] -> 0
  | h::t -> 1 + length t;;
```

Signature `'a list -> int`
Description Renvoie la longueur de la liste
Complexité $\Theta(n)$

Fonction disponible : `List.iter`

Signature `('a -> unit) -> 'a list -> unit`
Description Applique une fonction à tous les éléments de la liste
Complexité $\Theta(n)$

Implémentation

```
let rec iter f = function
  | [] -> ()
  | h::t -> f h; iter f t;;
```

Signature `('a -> unit) -> 'a list -> unit`
Description Applique une fonction à tous les éléments de la liste
Complexité $\Theta(n)$

Fonction disponible : `List.fold_left`

Signature `('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
Description Applique une fonction successivement à un élément de la liste et au résultat de l'itération précédente en partant de la fin de la liste
Complexité $\Theta(n)$

Implémentation

```
let rec fold_left f acc = function
  | [] -> acc
  | h::t -> fold_left f (f acc h) t;;
```

Signature `('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
Description Applique une fonction successivement à un élément de la liste et au résultat de l'itération précédente en partant de la fin de la liste
Complexité $\Theta(n)$

2 *Tablistes*

Définition : *Tabliste*

Structure de données qui permet de stocker un sous-ensemble \mathcal{N} fini de \mathbb{N} (de la forme $\llbracket 1, n-1 \rrbracket$), et d'effectuer des opérations sur celle-ci en $\Theta(1)$.

2.1 C

C

Définition d'un type

```
typedef struct {int* pos; int* values; int size;} Tablist;
```

La champ `values` correspond à une liste des entiers de \mathcal{N} sous la forme d'un tableau de taille n dont les `size` premières cases contiennent les éléments présents dans \mathcal{N} . Le champ `pos` est un tableau de taille n tel que : si $k \in \mathcal{N}$, `pos[k]` contient la position de k dans la liste `values` et une valeur quelconque sinon.

Fonction à implémenter : création d'une *tabliste*

```
Tablist init(int n) {
    Tablist t = {
        .pos = malloc(n*sizeof(int)),
        .values = malloc(n*sizeof(int)),
        .size = 0
    };
    for (int i = 0; i < n; ++i) {
        t.pos[i] = 0; // nécessaire à cause des nouvelles normes
        t.values[i] = 0; // facultatif
    }
    return t;
}
```

Description	Crée une <i>tabliste</i> vide
Complexité	$\Theta(n)$

Fonction à implémenter : appartenance à une *tabliste*

```
bool mem(Tablist t, int k) {
    int p = t.pos[k];
    return (p > 0 && p <= t.size && t.values[p]==k);
}
```

Description	Vérifie si un élément k appartient à la <i>tabliste</i> t
Complexité	$\Theta(1)$

Fonction à implémenter : ajout à une *tabliste*

```
void add(Tablist* ptr_t, int k) {  
    if (!mem(*ptr_t, k)) {  
        ptr_t->values[ptr_t->size] = k;  
        ptr_t->pos[k] = ptr_t->size;  
        ++ptr_t->size;  
    };  
}
```

Description Ajoute un élément k à la *tabliste* pointée par `ptr_t`
Complexité $\Theta(1)$

Fonction à implémenter : suppression d'une *tabliste*

```
void t_remove(Tablist* ptr_t, int k) {  
    if (mem(*ptr_t, k)) {  
        int i = ptr_t->values[ptr_t->size-1];  
        int p = ptr_t->pos[k];  
        ptr_t->values[p] = i;  
        ptr_t->pos[i] = p;  
        --ptr_t->size;  
    }  
}
```

Description Supprime l'élément k de la *tabliste* pointée par `ptr_t`
Complexité $\Theta(1)$

Fonction à implémenter : affichage d'une *tabliste*

```
void print(Tablist t) {  
    for (int i = 0; i < t.size-1; ++i) {  
        printf("%d, ", t.values[i]);  
    }  
    printf("%d\n", t.values[t.size-1]);  
}
```

Description Affiche les éléments de la *tabliste* t dans laquelle ils ont été ajoutés à celle-ci
Complexité $\Theta(|\mathcal{N}|)$

Fonction à implémenter : vidage d'une *tabliste*

```
void empty(Tablist* ptr_t) {  
    ptr_t->size = 0;  
}
```

Description Supprime les éléments de la *tabliste* pointée par `ptr_t` ($\mathcal{N} = \emptyset$)
Complexité $\Theta(1)$

Définition d'un type

```
type tablist = {
  pos : int array;
  values : int array;
  mutable size : int
};;
```

La champ `values` correspond à une liste des entiers de \mathcal{N} sous la forme d'un tableau de taille n dont les `size` premières cases contiennent les éléments présents dans \mathcal{N} . Le champ `pos` est un tableau de taille n tel que : si $k \in \mathcal{N}$, `pos[k]` contient la position de k dans la liste `values` et une valeur quelconque sinon.

Fonction à implémenter : création d'une *tabliste*

```
let init n = {
  pos = Array.make n 0;
  values = Array.make n 0;
  size = 0
};;
```

Signature `int -> tablist`
 Description Créé une *tabliste* vide
 Complexité $\Theta(n)$

Fonction à implémenter : appartenance à une *tabliste*

```
let mem t k =
  let p = t.pos.(k) in
  p > 0 && p < t.size && t.values.(p) = k;;
```

Signature `tablist -> int -> bool`
 Description Vérifie si un élément k appartient à la *tabliste* t
 Complexité $\Theta(1)$

Fonction à implémenter : ajout à une *tabliste*

```
let add t k =
  if not (mem t k) then
    begin
      t.values.(t.size) <- k;
      t.pos.(k) <- t.size;
      t.size <- t.size + 1;
    end;
  ;;
```

Signature	<code>tablist -> int -> unit</code>
Description	Ajoute un élément <code>k</code> à la <i>tabliste</i> pointée par <code>ptr_t</code>
Complexité	$\Theta(1)$

Fonction à implémenter : suppression d'une *tabliste*

```
let remove t k =
  if mem t k then
    begin
      let i = t.values.(t.size - 1)
      and p = t.pos.(k) in
      t.values.(p) <- i;
      t.pos.(i) <- p;
      t.size <- t.size - 1;
    end;
  ;;
```

Signature	<code>tablist -> int -> unit</code>
Description	Supprime l'élément <code>k</code> de la <i>tabliste</i> pointée par <code>ptr_t</code>
Complexité	$\Theta(1)$

Fonction à implémenter : affichage d'une *tabliste*

```
let print t =
  for i = 0 to t.size - 2 do
    print_int t.values.(i);
    print_string ", ";
  done;
  print_int t.values.(t.size - 1);
  print_newline ();;
```

Signature	<code>tablist -> unit</code>
Description	Affiche les éléments de la <i>tabliste</i> <code>t</code> dans laquelle ils ont été ajoutés à celle-ci
Complexité	$\Theta(\mathcal{N})$

Fonction à implémenter : vidage d'une *tabliste*

```
let empty t = t.size <- 0;;
```

Signature	<code>tablist -> unit</code>
Description	Supprime les éléments de la <i>tabliste</i> pointée par <code>ptr_t</code> ($\mathcal{N} = \emptyset$)
Complexité	$\Theta(1)$

