Informatique MP2I

Table des matières

1 Listes chaînées

1.1 C

Définition d'un type

```c
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 : premier élément d'une liste

```c
int fst(int_list* lst) {
    return int_list->value;
}
```

Description Retourne le premier élément d'une liste pointée par `lst`
Complexité $\Theta(1)$

Fonction à implémenter : dernier élément d'une liste

```c
int last(int_list* lst) {
    while (lst->nest != NULL) {lst = lst->next;}
    return int_list->value;
}
```

Description Retourne le dernier élément d'une liste pointée par `lst`
Complexité $\Theta(n)$

Fonction à implémenter : longueur d'une liste

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

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

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

```c
void add_l(int elem, int_list* lst) {
    int_list* new_p = (int_list*)malloc(sizeof(int_list));
```

```
    new_p->next = lst;
    new_p->value = elem;
}
```

Description Ajoute un élément `elem` à gauche de la liste pointée par `lst`
Complexité $\Theta(1)$

Fonction à implémenter : ajout d'un élément à droite d'une liste

```
void add_r(int elem, int_list* lst) {
    while (lst->next != NULL) {lst = lst->next;}
    int_list* new_p = (int_list*)malloc(sizeof(int_list));
    new_p->next = NULL;
    new_p->value = elem;
    lst->next = new_p;
}
```

Description Ajoute un élément `elem` à droite de la liste pointée par `lst`
Complexité $\Theta(n)$

Fonction à implémenter : appartenance à une liste

```
bool mem(int elem, int_list* lst) {
    if (lst->value == elem) {return true;}
    if (lst->next != NULL) {
        return mem(elem, lst->next);
    }
}
```

Description Vérifie si un élément `elem` est dans la liste pointée par `lst`
Complexité $O(n)$

Fonction à implémenter : retournement d'une liste

```
voir rev(int_list* lst) {
    int_list* rev_p = NULL;
    for (int_list* ptr = p_liste; ptr != NULL; ptr = ptr->next) {
        liste_int* tmp = (int_list*)malloc(sizeof(int_list));
        tmp->value = ptr->valeur;
        tmp->next = rev_p;
        rev_p = tmp;
    }
    return rev_p;
}
```

Description Retourne les éléments de la liste pointée par `lst`
Complexité $\Theta(n)$

Fonction à implémenter : suppression d'une liste

```c
void del(int_list* lst){
    while (lst->next != NULL) {
        list_int* tmp = lst;
        lst = tmp->next;
        free(tmp);
    }
}
```

Description	Supprime une liste pointée par `lst`
Complexité	$\Theta(n)$

1.2 OCaml

1.2.1 Type complexe

1.2.2 Le module `List`

Fonction disponible : `List.length`

Signature	`'a list -> int`
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	$\Theta(n)$

Fonction disponible : `List.map`

Signature	`('a -> 'b) -> 'a list -> 'b list`
Description	Applique une fonction à tous les éléments de la liste et renvoie une nouvelle liste avec les résultats
Complexité	$\Theta(n)$

Implémentation

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

Signature	`('a -> 'b) -> 'a list -> 'b list`
Description	Applique une fonction à tous les éléments de la liste et renvoie une nouvelle liste avec les résultats
Complexité	$\Theta(n)$

Fonction disponible : `List.rev`

Signature	`'a list -> 'a list`
Description	Inverse l'ordre des éléments d'une liste
Complexité	$\Theta(n)$

Implémentation

```
let rec rev l =
    let rec aux acc = function
        | [] -> acc
        | h::t -> aux (h::acc) t
    in aux [] l;;
```

Signature	`'a list -> 'a list`
Description	Inverse l'ordre des éléments d'une liste
Complexité	$\Theta(n)$

Fonction disponible : `List.mem`

Signature	`'a -> 'a list -> bool`
Description	Vérifie si un élément appartient à une liste
Complexité	$O(n)$

Implémentation

```
let rec mem elem = function
    | [] -> false
    | h::t -> elem = h || mem elem t;;
```

Signature	`'a -> 'a list -> bool`
Description	Vérifie si un élément appartient à une liste
Complexité	$O(n)$

2 Tris

Définition : Tri en place

Tri qui déplace les éléments à l'intérieur du tableau au lieu d'en créer un nouveau.

Définition : Tri stable

Tri qui préserve l'ordre des éléments (si `a` apparaît avant `b` dans le tableau initial, alors `a` apparaît avant `b` dans le tableau trié).

2.1 Différents algorithmes de tris

Définition : Tri par sélection

Tri qui sélectionne le plus petit élement du tableau non trié et l'insère au début de ce dernier.

Définition : Tri par insertion

Tri qui parcourt le tableau de gauche à droite, insérant chaque élément à sa place dans la partie triée du tableau.

Définition : Tri à bulles

Tri qui parcourt le tableau de gauche à droite en comparant chaque élément avec son voisin, et échange les deux éléments si le voisin est plus petit.

Définition : Tri rapide

Tri qui sélectionne un pivot et partitionne le tableau en deux sous-tableaux : les éléments plus petits que le pivot et ceux plus grands. Cette opération est répétée récursivement sur chaque sous-tableau.

Définition : Tri fusion

Tri qui divise récursivement le tableau en deux sous-tableaux, les trie et les fusionne pour obtenir le tableau trié.

2.2 C

Implémentation : échange de deux élements dans un tableau

```c
void array_swap(int* tab, int i, int j){
    int tmp = tab[i];
    tab[i] = tab[j];
    tab[j] = tmp;
}
```

Description	Échange les éléments `i` et `j` du tableau `tab`
Complexité	$\Theta(1)$

Implémentation : tri sélection

```
void tri_selection(int* tab, int n) {
    for (int i = 0; i < n-1; ++i) {
        int min = tab[i];
        int idx_min = i;
        for(int j = i+1; j < n; ++j) {
            if(tab[j] < min) {
                min = tab[j];
                idx_min = j;
            }
        }
        array_swap(tab, i, idx_min);
    }
}
```

Description	Tri le tableau `tab` avec un tri sélection
Complexité	$\Theta(n^2)$

Implémentation : tri insertion

```
void tri_insertion(int* tab, int n) {
    for(int i = 1; i < n; ++i) {
        for(int j = i; j > 0 && tab[j] < tab[j-1]; --j) {
            array_swap(tab, j, j-1);
        }
    }
}
```

Description	Tri le tableau `tab` avec un tri insertion
Complexité	$\Theta(n^2)$

Implémentation : tri à bulles

```
void tri_bulle(int* tab, int n) {
    for(int i = 0; i < n; ++i) {
        for(int j = n-1; j > i; --j) {
            if(tab[j] < tab[j-1]) {
                array_swap(tab, j, j-1);
            }
        }
    }
}
```

Description	Tri le tableau `tab` avec un tri à bulles
Complexité	$\Theta(n^2)$

Implémentation : partitionnement d'un tableau pour le tri rapide

```
int partition(int* tab, int n) {
    int pivot = tab[0];
    int d = n-1;
    int g = 1;
    while(d > g-1) {
        if(tab[g] < pivot) {++g;}
        else {array_swap(tab, g, d); --d;}
    }
    array_swap(tab, 0, g-1);
    return g;
}
```

Signature	`int* tab, int n`
Description	Partitionne le tableau `tab` de taille `n` pour le tri rapide
Complexité	$\Theta(n)$

Implémentation : Tri rapide

```
void tri_rapide(int* tab, int n) {
    int p = partition(tab, n);
    if(p > 1) {tri_rapide(tab, p);}
    if(n-p > 1) {tri_rapide(&tab[p], n-p);}
}
```

Description	Tri le tableau `tab` avec un tri rapide
Complexité	$\Theta(n\log(n))$ (en moyenne), $\Theta(n^2)$ (dans le pire des cas)

Implémentation : Tri fusion

```
void tri_fusion(int* tab, int n) {
    if(n < 11) {
        tri_bulle(tab,n);
    } else {
        int p = partition(tab, n);
        if(p > 1) {tri_fusion(tab, p);}
        if(n-p > 1) {tri_fusion(&tab[p], n-p);}
    }
}
```

Description	Tri le tableau `tab` avec un tri fusion
Complexité	$\Theta(n\log(n))$

2.3 OCaml

Implémentation : Minimum d'une liste

```
let min = function
    | [] -> failwith "Liste vide"
    | t::q ->
        let rec lst_min t = function
            | [] -> t;
            | t2::q2 when t < t2 -> lst_min t q2
            | t2::q2 -> lst_min t2 q2
        in lst_min t q;;
```

Signature	`'a list -> 'a`
Description	Renvoie le minimum de la liste fournie en paramètre
Complexité	$\Theta(n)$

Implémentation : Retire un élément d'une liste

```
let rec retire elem = function
    | [] -> failwith "Liste vide"
    | t::q when t = elem -> q
    | t::q -> t::retire elem q;;
```

Signature	`'a list -> 'a list`
Description	Retire l'élément `elem` de la liste fournie en paramètre
Complexité	$\Theta(n)$

Implémentation : Tri sélection

```
let rec tri_selection = function
    | [] -> []
    | lst -> let m = min lst in m::tri_selection (retire m lst);;
```

Signature	`'a list -> 'a list`
Description	Tri le tableau avec un tri sélection
Complexité	$\Theta(n^2)$

Implémentation : Tri insertion

```
let rec tri_insertion = function
    | [] -> []
    | t::q ->
        let rec insere elem = function
            | [] -> [elem]
            | t::q when t < elem -> t::insere elem q
            | lst -> elem::lst
        in insere t (tri_insertion q);;
```

Signature	`'a list -> 'a list`
Description	Tri le tableau avec un tri insertion
Complexité	$\Theta(n^2)$

Implémentation : Tri bulle

```
let rec tri_bulle = function
    | [] -> []
    | lst ->
        let rec bulle = function
            | [] -> []
            | t::[] -> [t]
            | t::q ->
                let b = bulle q in
                    if List.hd b < t then List.hd b::t::(List.tl b)
                    else t::b
        in let b = bulle lst in List.hd b::tri_bulle (List.tl b);;
```

Signature	`'a list -> 'a list`
Description	Tri le tableau avec un tri bulle
Complexité	$\Theta(n^2)$

Implémentation : Joint deux listes

```
let rec join lst = function
    | [] -> lst
    | t::q -> t::join lst q
```

Signature	`'a list -> 'a list -> 'a list`
Description	Joint une liste devant la liste `lst`
Complexité	$\Theta(n)$ (n est la longueur de la deuxième liste)

Implémentation : Partitionnement d'une liste

```
let rec partition p left right = function
    | [] -> (left,right)
    | t::q when t < p -> partition p (t::left) right q
    | t::q -> partition p left (t::right) q;;
```

Signature	`'a -> 'a list -> 'a list -> 'a list-> 'a list * 'a list`
Description	Partitionne les éléments de la liste donnée en paramètre `left` et `right` servent d'accumulateurs pour les éléments plus petits et plus grand que le pivot `p`
Complexité	$\Theta(n)$

Implémentation : Tri rapide

```
let rec tri_rapide = function
    | [] -> []
    | t::q ->
        let (l,r) = partition t [] [] q in
        join (t::tri_rapide r) (tri_rapide l);;
```

Signature	`'a list -> 'a list`
Description	Tri le tableau avec un tri rapide
Complexité	$\Theta(n \log(n))$ (en moyenne), $\Theta(n^2)$ (dans le pire des cas)

3 *Tablistes*

Définition : *Tabliste*

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

3.1 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 Créé une *tabliste* 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 élement `k` appartient à la *tabliste* `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 : supression 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` ($\mathcal{N} = \varnothing$)
Complexité : $\Theta(1)$

3.2 OCaml

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 élement `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	`tablist -> int -> unit`
Description	Ajoute un élément `k` à la *tabliste* pointée par `ptr_t`
Complexité	$\Theta(1)$

Fonction à implémenter : supression 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	`tablist -> int -> unit`
Description	Supprime l'élément `k` de la *tabliste* pointée par `ptr_t`
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	`tablist -> unit`
Description	Affiche les éléments de la *tabliste* `t` dans laquelle ils ont été ajoutés à celle-ci
Complexité	$\Theta(\lvert\mathcal{N}\rvert)$

Fonction à implémenter : vidage d'une *tabliste*

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

Signature	`tablist -> unit`
Description	Supprime les éléments de la *tabliste* pointée par `ptr_t` ($\mathcal{N} = \varnothing$)
Complexité	$\Theta(1)$