

Informatique MP2I

Table des matières

1	Listes chaînées	1
1.1	C	1
1.2	OCaml – Le module <code>List</code>	3
2	Tableaux	6
2.1	C	6
2.2	OCaml – Le module <code>Array</code>	6
3	Piles	9
3.1	C	9
3.2	OCaml – Le module <code>Stack</code>	9
4	Tris	12
4.1	Différents algorithmes de tris	12
4.2	C	13
4.3	OCaml	15
5	<i>Tablistes</i>	18
5.1	C	18
5.2	OCaml	20

1 Listes chaînées

1.1 C

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'adaptent également pour les autres types
*/
```

Fonction à implémenter : premier élément d'une liste

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

Description	Retourne le premier élément d'une liste pointée par <code>lst</code>
Complexité	$\Theta(1)$

Fonction à implémenter : dernier élément d'une liste

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

Description	Retourne le dernier élément d'une liste pointée par <code>lst</code>
Complexité	$\Theta(n)$

Fonction à implémenter : longueur d'une liste

```
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

```
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

```
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 – 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 Tableaux

2.1 C



2.2 OCaml – Le module Array



Fonction disponible : `Array.length`

Signature	<code>'a array -> int</code>
Description	Renvoie la longueur du tableau
Complexité	$\Theta(1)$

Implémentation

```
let length arr =  
  let l = ref 0 and count = ref true in  
  while !count do  
    try  
      let _ = arr.(!l) in incr l  
    with  
      | Invalid_argument _ -> count := false  
  done;  
  !l;;
```

Signature	<code>'a array -> int</code>
Description	Renvoie la longueur du tableau
Complexité	$\Theta(n)$

Fonction disponible : `Array.iter`

Signature	<code>('a -> unit) -> 'a array -> unit</code>
Description	Applique une fonction à tous les éléments du tableau
Complexité	$\Theta(n)$

Implémentation

```
let iter f arr =  
  for i = 0 to length arr-1 do  
    arr.(i) <- (f arr.(i))  
  done;;
```

Signature	<code>('a -> unit) -> 'a array -> unit</code>
Description	Applique une fonction à tous les éléments du tableau
Complexité	$\Theta(n)$

Fonction disponible : `Array.fold_left`

Signature	<code>('a -> 'b -> 'a) -> 'a -> 'b array -> 'a</code>
Description	Applique une fonction successivement à un élément du tableau et au résultat de l'itération précédente en partant du début du tableau
Complexité	$\Theta(n)$

Implémentation

```
let fold_left f def arr =  
  let acc = ref def in  
  for i = 0 to length arr-1 do  
    acc := f !acc arr.(i)  
  done;  
  !acc;;
```

Signature	<code>('a -> 'b -> 'a) -> 'a -> 'b array -> 'a</code>
Description	Applique une fonction successivement à un élément du tableau et au résultat de l'itération précédente en partant du début du tableau
Complexité	$\Theta(n)$

Fonction disponible : `Array.fold_right`

Signature	<code>('a -> 'b -> 'b) -> 'a array -> 'b -> 'b</code>
Description	Applique une fonction binaire de pliage à tous les éléments du tableau, en partant de la droite, et renvoie un résultat plié
Complexité	$\Theta(n)$

Implémentation

```
let fold_right f arr def =  
  let acc = ref def in  
  for i = length arr-1 downto 0 do  
    acc := f arr.(i) !acc  
  done;  
  !acc;;
```

Signature	<code>('a -> 'b -> 'b) -> 'a array -> 'b -> 'b</code>
Description	Applique une fonction binaire de pliage à tous les éléments du tableau, en partant de la droite, et renvoie un résultat plié
Complexité	$\Theta(n)$

Fonction disponible : `Array.map`

Signature	<code>('a -> 'b) -> 'a array -> 'b array</code>
Description	Applique une fonction à tous les éléments du tableau et renvoie un nouveau tableau avec les résultats
Complexité	$\Theta(n)$

Implémentation

```
let map f arr =  
  let new_arr = Array.make (Array.length arr) (f arr.(0)) in  
  for i = 1 to length arr-1 do  
    new_arr.(i) <- f arr.(i)  
  done;  
  new_arr;;
```

Signature	<code>('a -> 'b) -> 'a array -> 'b array</code>
Description	Applique une fonction à tous les éléments du tableau et renvoie un nouveau tableau avec les résultats
Complexité	$\Theta(n)$

3 Piles

3.1 C



3.2 OCaml – Le module Stack



Définition d'un type

```
type 'a t = {len: int ref; mutable contents: 'a list};;  
  
exception Empty;;
```

Fonction disponible : `Stack.create`

Signature `unit -> 'a t`
Description Crée une nouvelle pile vide
Complexité $\Theta(1)$

Implémentation

```
let create () = {len = ref 0; contents = []};;
```

Signature `unit -> 'a t`
Description Crée une nouvelle pile vide
Complexité $\Theta(1)$

Fonction disponible : `Stack.push`

Signature `'a -> 'a t -> unit`
Description Ajoute un élément au sommet de la pile
Complexité $\Theta(1)$

Implémentation

```
let push elem t = t.contents <- elem::t.contents; incr t.len;;
```

Signature `'a -> 'a t -> unit`
Description Ajoute un élément au sommet de la pile
Complexité $\Theta(1)$

Fonction disponible : `Stack.pop`

Signature `'a t -> 'a`
Description Retire et renvoie l'élément au sommet de la pile
Complexité $\Theta(1)$

Implémentation

```
let pop t = match t.contents with  
  | [] -> raise Empty  
  | hd::tl -> t.contents <- tl; decr t.len; hd;;
```

Signature `'a t -> 'a`

Description Retire et renvoie l'élément au sommet de la pile

Complexité $\Theta(1)$

Fonction disponible : `Stack.top`

Signature `'a t -> 'a`

Description Renvoie l'élément au sommet de la pile sans le retirer

Complexité $\Theta(1)$

Implémentation

```
let top t = let elem = pop t in push elem t; elem;;
```

Signature `'a t -> 'a`

Description Renvoie l'élément au sommet de la pile sans le retirer

Complexité $\Theta(1)$

Fonction disponible : `Stack.is_empty`

Signature `'a t -> bool`

Description Renvoie vrai si la pile est vide, faux sinon

Complexité $\Theta(1)$

Implémentation

```
let is_empty t = if !(t.len) = 0 then true else false;;
```

Signature `'a t -> bool`

Description Renvoie vrai si la pile est vide, faux sinon

Complexité $\Theta(1)$

Fonction disponible : `Stack.length`

Signature `'a t -> int`

Description Renvoie la longueur de la pile

Complexité $\Theta(1)$

Implémentation

```
let length t = !(t.len);;
```

Signature `'a t -> int`

Description Renvoie la longueur de la pile

Complexité $\Theta(1)$

4 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).

4.1 Différents algorithmes de tris

Définition : Tri par sélection

Tri qui sélectionne le plus petit élément 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.

Propriété/Théorème : Tri à bulles

Le tri à bulles est stable.

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 mixte

Tri qui utilise plusieurs algorithmes de tri pour avoir une meilleure complexité.

Implémentation : échange de deux éléments dans un tableau

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

Implémentation : tri rapide

```
void tri_mixte(int* tab, int n) {
    if (n < 10) {
        tri_bulle(tab,n);
    } else {
        int p = partition(tab, n);
        if (p > 1) {tri_mixte(tab, p);}
        if (n-p > 1) {tri_mixte(&tab[p], n-p);}
    }
}
```

Description Trie le tableau `tab` avec un tri rapide ou un tri à bulles si la liste est petite

Complexité $\Theta(n \log(n))$ (en moyenne), $\Theta(n^2)$ (dans le pire des cas)

4.3 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 Trie 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 Trie 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 Trie 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 Trie le tableau avec un tri rapide
Complexité $\Theta(n \log(n))$ (en moyenne), $\Theta(n^2)$ (dans le pire des cas)

5 *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 $\Theta(1)$.

5.1 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	<code>int -> tablist</code>
Description	Crée une <i>tabliste</i> 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	<code>tablist -> int -> bool</code>
Description	Vérifie si un élément k appartient à la <i>tabliste</i> 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 : 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 `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 $\Theta(|\mathcal{N}|)$

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` ($\mathcal{N} = \emptyset$)

Complexité $\Theta(1)$