Cours n° 6 : piles et files

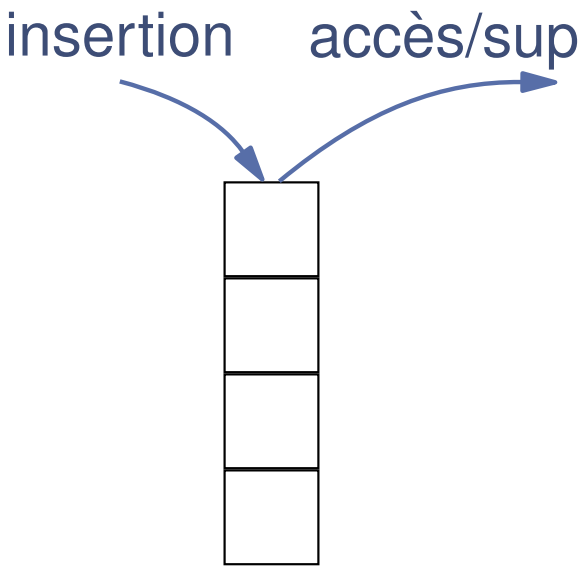
Christophe Gonzales

HUGO3 — Algorithmique

Les piles

Définition d'une pile

- Structure de données contenant un ensemble d'éléments de même type
- Insertion aisée de nouveaux éléments en O(1)
- Accès au dernier élément inséré en O(1)
- Suppression du dernier élément inséré en O(1)
- Pas d'autre opérateur



- structure en « pile d'assiettes »
- appelée **LIFO** : Last In, First Out
- Fréquemment utilisée en informatique :
  - Pile d'exécution (pile d'appels des fonctions)
  - Évaluation d'expression arithmétique
  - "Undo/Redo" dans un logiciel

Opérations usuelles sur les piles

Quelles opérations implanter sur une pile ?

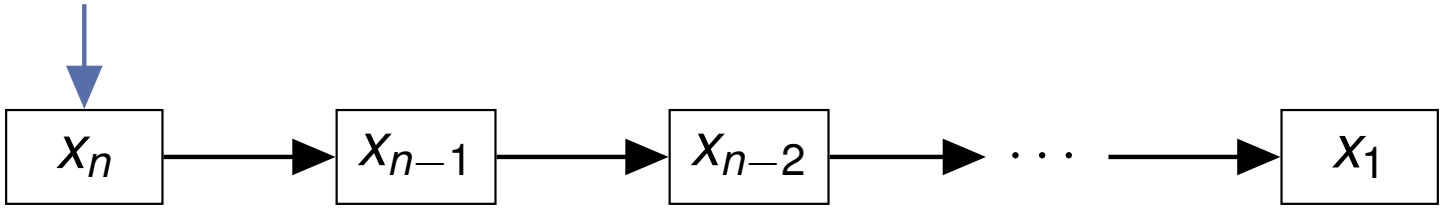
Opérations usuelles

- **Créer** une pile vide
- **Empiler** un nouvel élément dans la pile : `push`
- **Dépiler** un élément de la pile : `pop`
- **Récupérer** l'élément en tête de pile : `top`
- **Tester** si une pile est vide : `empty`

Implantations possibles

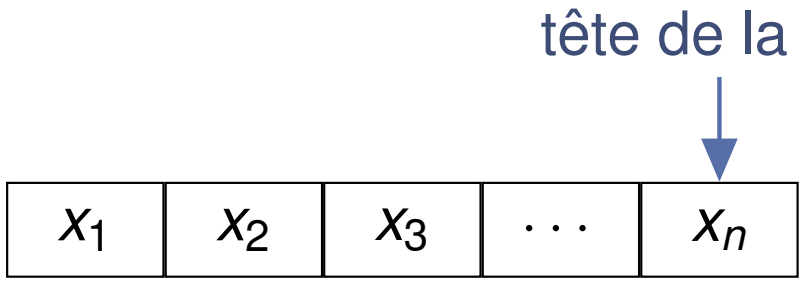
- **Implantation avec des listes simplement chaînées :**

tête de la pile



$\Longrightarrow$ ajout/accès/suppression en O(1)

- **Implantation avec des tableaux :**



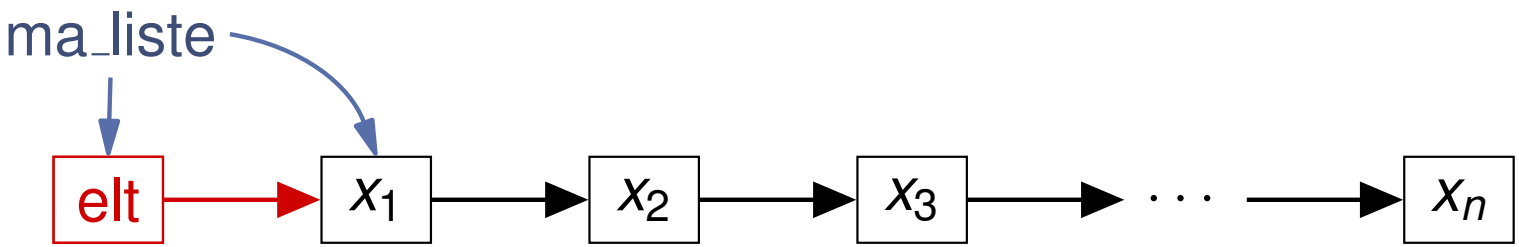
$\Longrightarrow$ ajout/accès/suppression en O(1)

Implantation par liste simplement chaînée (1/6)

```
#include <stdio.h>
#include <stdlib.h>
#include <exception> // exception du C++

// la structure des boites de la liste chaînée
typedef struct list_box {
  int value;
  struct list_box* next;
} list_box_t;

// la structure définissant une liste
// ⟹ permet d'avoir un pointeur sur un list_t
// et de rajouter un élément en tête de liste sans
// modifier le pointeur sur le list_t
typedef struct {
  list_box_t* tete;
} list_t;


// création d'une liste : ne contient aucune boite
list_t list_create () {
  list_t ma_liste { NULL };
  return ma_liste;
}
```

Implantation par liste simplement chaînée (2/6)

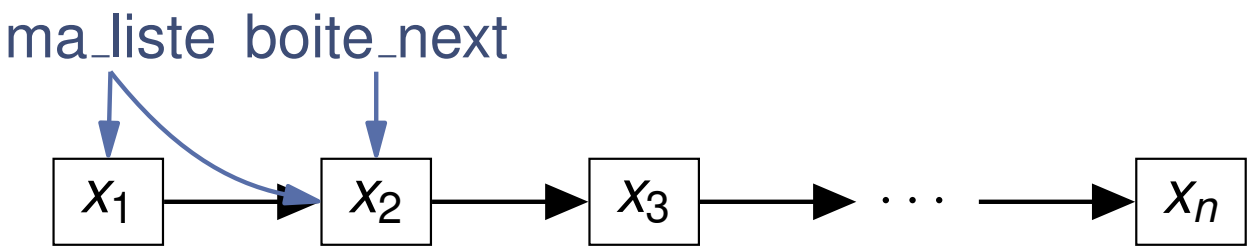
```
// rajoute un élément en tête de liste
void list_insert (list_t* ma_liste, int elt) {
  // création d'une nouvelle boite
  list_box_t* boite =
    (list_box_t*) malloc (sizeof (list_box_t));
  if (boite == NULL) throw std::exception ();

  // remplissage de la boite
  boite->value = elt;
  boite->next = ma_liste->tete;

  // rechaînage
  ma_liste->tete = boite;
}
```



`throw` $\Longrightarrow$ arrête l'exécution du programme (C++)

Implantation par liste simplement chaînée (3/6)

```
// suppression de l'élément en tête de liste s'il existe
// sinon on ``lève une exception''
void list_remove (list_t* ma_liste) {
  // si la liste est vide, c'est une erreur
  if (ma_liste->tete == NULL) throw std::exception ();

  // on conserve un pointeur sur la 2eme boite
  list_box_t* boite_next = ma_liste->tete->next;

  // on supprime le 1er element et on rechaîne
  free (ma_liste->tete);
  ma_liste->tete = boite_next;
}
```



Implantation par liste simplement chaînée (4/6)

```
// on récupère le premier élément de la liste
int list_front (list_t* ma_liste) {
  // si la liste est vide, c'est une erreur
  if (ma_liste->tete == NULL) throw std::exception ();

  return ma_liste->tete->value;
}

// teste si la liste est vide ou non
bool list_empty (list_t* ma_liste) {
  return ma_liste->tete == NULL;
}

// supprime toutes les boites de la liste
void list_clear (list_t* ma_liste) {
  list_box_t* boite = ma_liste->tete;
  while (boite != NULL) {
    list_box_t* boite_next = boite->next;
    free (boite);
    boite = boite_next;
  }
  ma_liste->tete = NULL;
}
```

Implantation par liste simplement chaînée (5/6)

```
// la structure de pile, contenant son implantation
// Le fait de cacher l'implantation (la liste) permet
// un ``haut'' niveau d'abstraction
typedef struct {
  list_t liste; // en orienté objet : composition
} pile_t;

// fonction qui initialise et renvoie une pile vide
pile_t create () {
  pile_t pile;
  pile.liste = list_create ();
  return pile;
}

// rajoute un élément dans la pile
void push (pile_t* pile, int elt) {
  list_insert (&(pile->liste), elt);
}

// supprime le dernier élément entré dans la pile
int pop (pile_t* pile) {
  int elt = list_front (&(pile->liste));
  list_remove (&(pile->liste));
  return elt;
}
```

Implantation par liste simplement chaînée (6/6)

```
// récupère le dernier élément entré dans la pile
int top (pile_t* pile) {
  return list_front (&(pile->liste));
}

// teste si la pile est vide
bool empty (pile_t* pile) {
  return list_empty (&(pile->liste));
}

// supprime tous les éléments de la pile
void clear (pile_t* pile) {
  list_clear (&(pile->liste));
}
```

Cacher à l'utilisateur la structure de données interne

$\Longrightarrow$ il n'utilisera que les spécificités des piles

$\Longrightarrow$ Si vous changez l'implantation des piles, l'utilisateur n'aura rien à modifier dans son programme

Implantation par tableau (1/6)

La structure « interne » de la pile

```c
#include <stdio.h>
#include <stdlib.h>
#include <exception> // exception du C++

// une structure de tableau dans lequel on peut
// rajouter des éléments en fin, et qui se réalloue
// si besoin lors de ces ajouts
typedef struct {
  int* data;     // l'espace contenant les données
  int  taille;   // la taille du tableau
  int  nb_elts; // nb éléments stockés dans le tableau
} array_t;
```

- équivalent aux `std::vector` du C++

Implantation par tableau (2/6)

```
// la taille par défaut des tableaux
#define TAILLE 2

// fonction qui initialise et renvoie un tableau vide
array_t array_create () {
  array_t tableau;

  // allocation du tableau
  tableau.data = (int*) malloc (TAILLE * sizeof(int));
  if (tableau.data == NULL) throw std::exception ();

  // pas d'éléments dans le tableau pour l'instant
  tableau.nb_elts = 0;
  tableau.taille = TAILLE;

  return tableau;
}
```

Implantation par tableau (3/6)

```
// rajoute un élément en fin de tableau
void array_insert (array_t* tableau, int elt) {
  // si le tableau est plein, doubler sa taille
  if ( tableau->nb_elts >= tableau->taille) {
    // on alloue un tableau 2 fois plus grand
    int* new_tab =
      (int*) malloc (2 * tableau->taille * sizeof(int));
    if (new_tab == NULL) throw std::exception ();

    // on recopie le tableau tab dans new_tab
    for (int i = 0; i < tableau->taille; i++) {
      new_tab[i] = tableau->data[i];
    }

    // on affecte new_tab à la pile
    free (tableau->data);
    tableau->data = new_tab;
    tableau->taille *= 2;
  }

  // on rajoute l'élément au tableau
  tableau->data[tableau->nb_elts] = elt;
  tableau->nb_elts++;
}
```

Implantation par tableau (4/6)

```
// suppression de l'élément en fin de tableau s'il existe
void array_remove (array_t* tableau) {
  // si le tableau est vide, c'est une erreur
  if (tableau->nb_elts == 0) throw std::exception ();

  // on supprime l'élément
  tableau->nb_elts--;
}

// on récupère le premier élément du tableau
int array_back (array_t* tableau) {
  // si le tableau est vide, c'est une erreur
  if (tableau->nb_elts == 0) throw std::exception ();
  return tableau->data[tableau->nb_elts - 1];
}

// teste si la liste est vide ou non
bool array_empty (array_t* tableau) {
  return (tableau->nb_elts == 0);
}

// supprime toutes les boites de la liste
void array_clear (array_t* tableau) {
  tableau->nb_elts = 0;
}
```

Implantation par tableau (5/6)

```
// la structure de pile, contenant son implantation
typedef struct {
  array_t tab;  // implantation de la pile : un tableau
} pile_t;

// fonction qui initialise et renvoie une pile vide
pile_t create () {
  pile_t pile;
  pile.tab = array_create ();
  return pile;
}

// rajoute un élément dans la pile
void push (pile_t* pile, int elt) {
  array_insert (&(pile->tab), elt);
}

// supprime le dernier élément entré dans la pile
int pop (pile_t* pile) {
  int elt = array_back (&(pile->tab));
  array_remove (&(pile->tab));
  return elt;
}
```

Implantation par tableau (6/6)

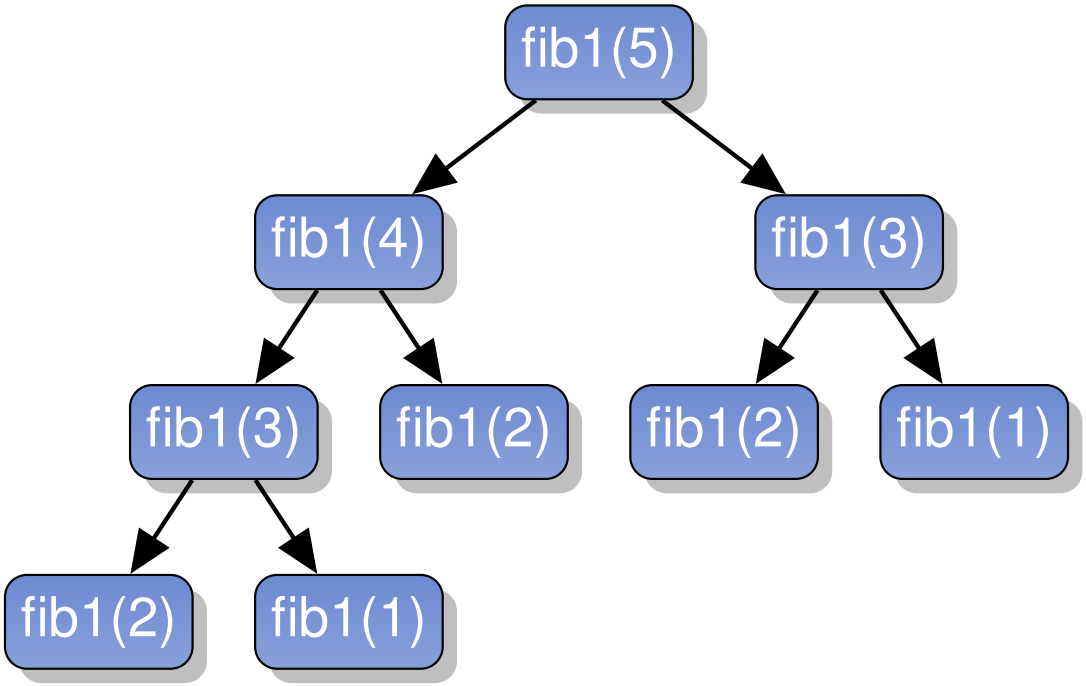
```
// récupère le dernier élément entré dans la pile
int top (pile_t* pile) {
  return array_back (&(pile->tab));
}

bool empty (pile_t* pile) {
  return array_empty (&(pile->tab));
}

void clear (pile_t* pile) {
  array_clear (&(pile->tab));
}
```

$\Longrightarrow$ Code très similaire aux piles avec listes !

Pile d'exécution et récursivité (1/2)

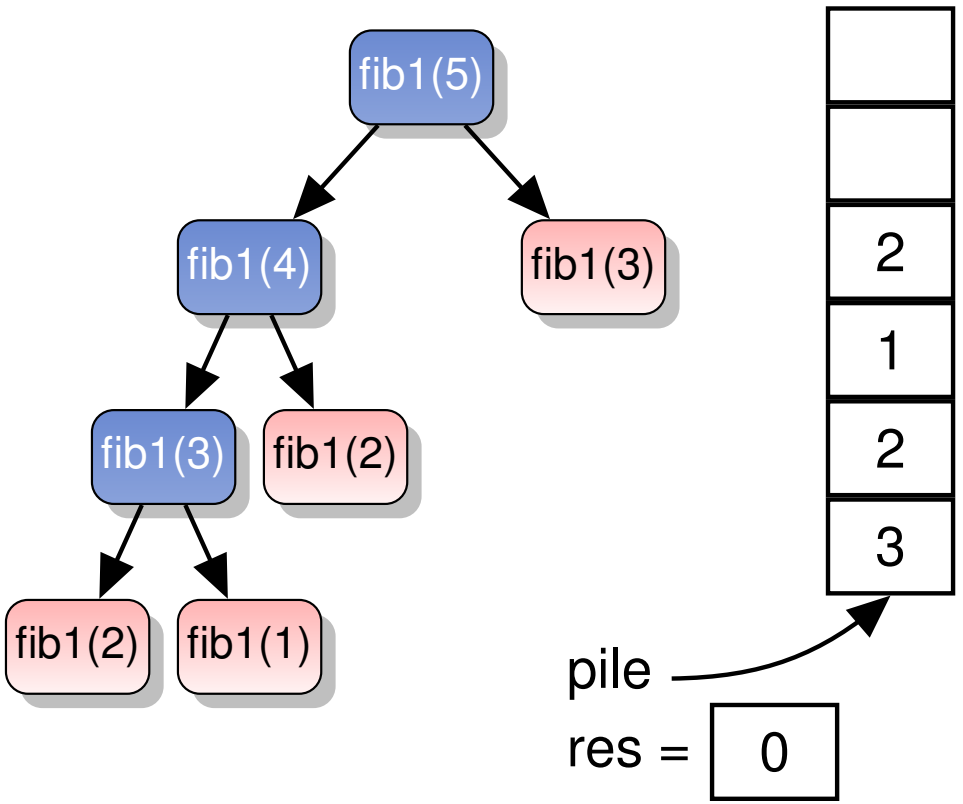
```
int fib1 (int n) {
  if (n <= 2) return 1;
  return fib1(n-1) + fib1(n-2);
}
```

Exécution de fib1(5) :



Pile d'exécution et récursivité (2/2)

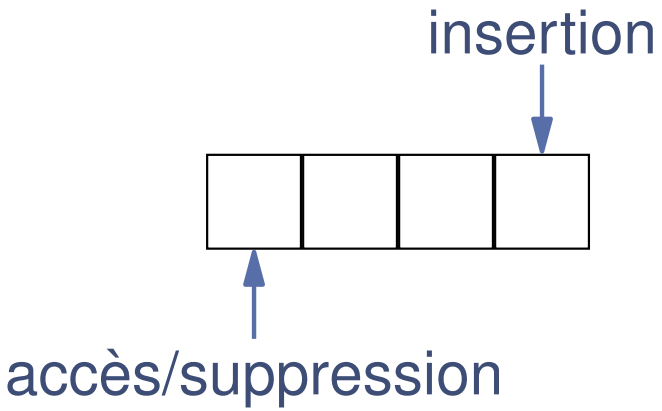
```
int fib3 (int n) {
  pile_t pile = create ();
  int res = 0;
  push (&pile, n);

  while (! empty(&pile) ) {
    int nb = pop (&pile);
    if (nb <= 2)
      res += 1;
    else {
      push (&pile, nb-2);
      push (&pile, nb-1);
    }
  }

  return res;
}
```



Les files

Définition d'une file

- Structure de données contenant un ensemble d'éléments de même type
- Insertion aisée de nouveaux éléments (`push`) en O(1)
- Accès au **premier** élément inséré (`top`) en O(1)
- Suppression du **premier** élément inséré (`pop`) en O(1)
- Pas d'autre opérateur



- appelée **FIFO** : First In, First Out
- Fréquemment utilisée en informatique :
  - files d'attente
  - *pipe* de Linux

Opérations usuelles sur les files

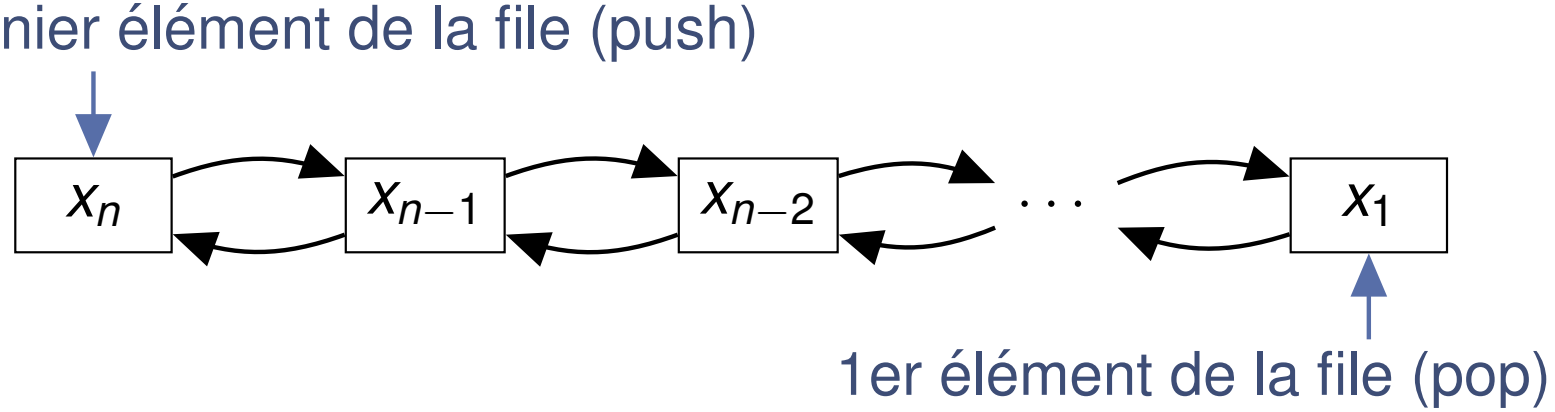
Quelles opérations implanter sur une file ?

Opérations usuelles

- **Créer** une file vide
- **Enfiler** un nouvel élément dans la file : `push`
- **Défiler** l'élément le plus ancien de la file : `pop`
- **Récupérer** l'élément le plus ancien de la file : `top`
- **Tester** si une file est vide : `empty`

Implantation des files

- Liste doublement chaînée, accès aux premier et dernier élément



- Tableau circulaire :

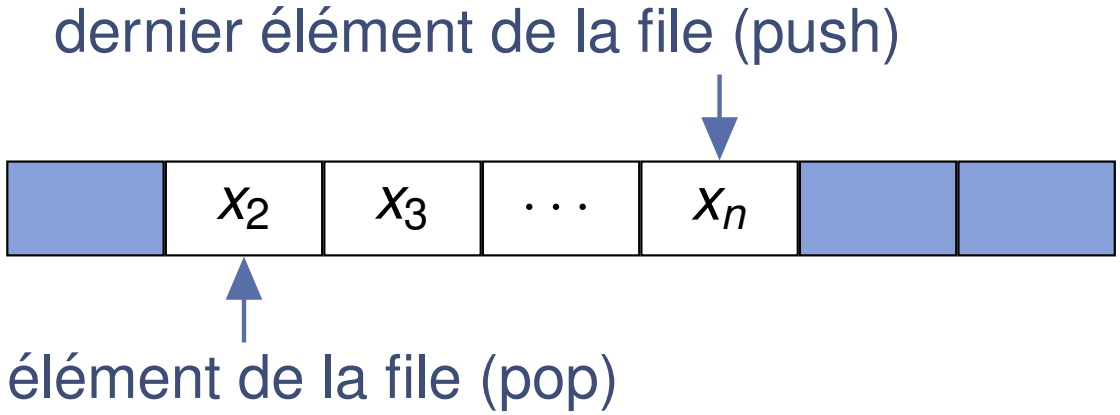


tableau circulaire : arrivé en fin, on repart au début du tableau pour insérer de nouveaux éléments !

Synthèse des piles et files

Opération	pile	file
Insertion en début de structure	✖	✖
Insertion en fin de structure	✔	✔
Insertion en cours de structure	✖	✖
Suppression en début de structure	✖	✔
Suppression en fin de structure	✔	✖
Suppression en cours de structure	✖	✖
Accéder au ième élément	✖	✖
Accéder au premier élément	✖	✔
Accéder au dernier élément	✔	✖
Accéder à l'élément le plus petit	✖	✖
Accéder à l'élément le plus grand	✖	✖
Chercher si un élément existe	✖	✖