Structures de données

MP/MP* Option informatique

Structure abstraite vs structure concrète

Une structure de donnée est un moyen de stocker un ensemble d'éléments.

1. **Type abstrait**: description des opérations permises sur une structure de donnée.
   Exemple: pile, file...

Structure abstraite vs structure concrète

Une structure de donnée est un moyen de stocker un ensemble d'éléments.

1. **Type abstrait**: description des opérations permises sur une structure de donnée.
   Exemple: pile, file...
2. **Réalisation concrète**: implémentation de ces opérations dans un langage de programmation, qui en détermine la complexité:
   $\Theta(n)$: mauvais
   $\Theta(\log(n))$: bien
   $\Theta(1)$: parfait.

Il peut exister plusieurs réalisations concrètes de la même structure abstraite.

Structure persistante vs structure mutable

1. Structure **persistante**: ne peut pas être modifiée, seules de nouvelles valeurs sont renvoyées.
   Exemples: `list`, `arbre`...
2. Structure **mutable**: peut être modifiée.
   Exemples: `array`, `string`, `ref`, `mutable`...

Structure persistante vs structure mutable

1. Structure **persistante**: ne peut pas être modifiée, seules de nouvelles valeurs sont renvoyées.
   Exemples: `list`, `arbre`...
2. Structure **mutable**: peut être modifiée.
   Exemples: `array`, `string`, `ref`, `mutable`...

Avantage des structures persistantes: moins de risques de bugs, programme plus facile à prouver par **induction structurelle**, backtracking (retour en arrière plus aisé).

Structure persistante vs structure mutable

On peut souvent voir, avec le type d'une fonction, si la structure utilisée est persistante.

Exemple: un algorithme de tri sera de type `'a list -> 'a list` pour une liste et `'a array -> unit` pour un tableau.

Structure persistante vs structure mutable

On peut souvent voir, avec le type d'une fonction, si la structure utilisée est persistante.

Exemple: un algorithme de tri sera de type `'a list -> 'a list` pour une liste et `'a array -> unit` pour un tableau.

Les structures persistantes sont plus adaptées à la programmation fonctionnelle (récursive) et les structures mutables à la programmation impérative (boucles, références).

Structure de données

Structures de données que nous allons voir:

1. Liste (doublement chaînée)
2. Tableau (dynamique)
3. Pile, file
4. Arbre binaire de recherche (TD: AVL, rouge/noir...)
5. Dictionnaire (avec ABR, table de hachage)
6. File de priorité (avec tas ou ABR)
7. Graphe (plus tard)

Liste

Les listes pourraient être redéfinies de la façon suivante (chaque élément a accès aux éléments suivant de la liste):

```
type 'a liste = Vide | Cons of 'a * 'a liste;;
```

On peut ainsi prouver qu'une proposition/un programme $\mathcal{P}$ est correct sur les listes en montrant:

1. $\mathcal{P}(\texttt{[]})$
2. $\mathcal{P}(\texttt{l}) \implies \forall\ \texttt{e},\ \mathcal{P}(\texttt{e::l})$

Liste

Pour une liste de taille n:

Opération	Complexité
taille	
test liste vide	
accéder/supprimer/ajouter au début	
accéder/supprimer/ajouter en position qcq	
recherche élément	
`l1 @ l2`	

Liste

Pour une liste de taille n:

Opération	Complexité
taille	$\Theta(n)$
test liste vide	$O(1)$
accéder/supprimer/ajouter au début	$O(1)$
accéder/supprimer/ajouter en position qcq	$O(n)$
recherche élément	$O(n)$
`l1 @ l2`	Θ(taille de l1)

Tableau

Pour un tableau de taille n:

Opération	Tableau	Tableau trié
taille	$O(1)$	$O(1)$
`Array.make n x`	$\Theta(n)$	$\Theta(n)$
`t.(i)`	$O(1)$	$O(1)$
`t.(i) <- ...`	$O(1)$	$O(n)$
recherche élément	$O(n)$	$O(\log(n))$

Il est impossible d'ajouter un élément à un tableau (la taille est fixée à la création du tableau).

Tableau

```
let t1 = [|1; 2|];;
let t2 = t1;;
t2.(0) <- 3;;
```

Que vaut `t1`?

Tableau

```
let t1 = [|1; 2|];;
let t2 = t1;;
t2.(0) <- 3;;
```

Que vaut `t1`?

`t1` vaut alors `[|3; 2|]`: la modification de `t1` modifie aussi `t2`.

Tableau de tableaux

Ne surtout pas créer une matrice $n \times p$ en écrivant:

```
Array.make n (Array.make p 0)
```

Tableau de tableaux

Ne surtout pas créer une matrice $n \times p$ en écrivant:

```
Array.make n (Array.make p 0)
```

Utiliser `Array.make_matrix n p x` à la place, ou le recoder:

```
let make_matrix n p x =
  let res = Array.make n [||] in
  for i = 0 to n - 1 do
    res.(i) <- Array.make p x
  done;
  res;;
```

Tableau dynamique

La taille n d'un tableau est fixée à sa création. On voudrait lui rajouter un $n+1$ème élément.

Tableau dynamique

La taille n d'un tableau est fixée à sa création. On voudrait lui rajouter un $n+1$ème élément.

1. Idée 1: créer un nouveau tableau de taille $n+1$ et recopier les éléments. Complexité:

Tableau dynamique

La taille n d'un tableau est fixée à sa création. On voudrait lui rajouter un $n+1$ème élément.

1. Idée 1: créer un nouveau tableau de taille $n+1$ et recopier les éléments. Complexité: $\Theta(n)$
2. Idée 2 (tableau dynamique):

Tableau dynamique

La taille n d'un tableau est fixée à sa création. On voudrait lui rajouter un $n+1$ème élément.

1. Idée 1: créer un nouveau tableau de taille $n+1$ et recopier les éléments. Complexité: $\Theta(n)$
2. Idée 2 (tableau dynamique): créer un nouveau tableau de taille $\boldsymbol{2n}$ et recopier les éléments.

Tableau dynamique

La taille n d'un tableau est fixée à sa création. On voudrait lui rajouter un $n+1$ème élément.

1. Idée 1: créer un nouveau tableau de taille $n+1$ et recopier les éléments. Complexité: $\Theta(n)$
2. Idée 2 (tableau dynamique): créer un nouveau tableau de taille $\boldsymbol{2n}$ et recopier les éléments. Si l'on réalise n ajouts à partir d'un tableau de taille 1, il y a des recopies pour les tailles 1, 2, 4, ..., $2^{\lfloor \log_2(n) \rfloor}$ et le nombre total de valeurs recopiées est

$$\sum_{k=0}^{\lfloor \log_2(n) \rfloor} 2^k = \Theta(2^{\lfloor \log_2(n) \rfloor}) = \Theta(n)$$

Complexité **amortie** (moyenné sur n opérations):

Tableau dynamique

La taille n d'un tableau est fixée à sa création. On voudrait lui rajouter un $n+1$ème élément.

1. Idée 1: créer un nouveau tableau de taille $n+1$ et recopier les éléments. Complexité: $\Theta(n)$
2. Idée 2 (tableau dynamique): créer un nouveau tableau de taille $\boldsymbol{2n}$ et recopier les éléments. Si l'on réalise n ajouts à partir d'un tableau de taille 1, il y a des recopies pour les tailles 1, 2, 4, ..., $2^{\lfloor \log_2(n) \rfloor}$ et le nombre total de valeurs recopiées est

$$\sum_{k=0}^{\lfloor \log_2(n) \rfloor} 2^k = \Theta(2^{\lfloor \log_2(n) \rfloor}) = \Theta(n)$$

Complexité **amortie** (moyenné sur n opérations): $O(1)$
(Une « list » de Python est en fait un tableau dynamique!)

Complexité amortie vs complexité en moyenne

Ne pas confondre:

1. Complexité **amortie**: complexité moyennée sur un certain nombre d'appels de la fonction.
2. Complexité **en moyenne**: complexité moyennée sur toutes les entrées possibles.

Pile

Une pile est aussi appelée FILO: First In, Last Out.

1. Implémentation persistante:

Pile

Une pile est aussi appelée FILO: First In, Last Out.

1. Implémentation persistante: avec une liste, on ajoute/supprime en $O(1)$ la tête de liste.
2. Implémentation impérative:

Pile

Une pile est aussi appelée FILO: First In, Last Out.

1. Implémentation persistante: avec une liste, on ajoute/supprime en $O(1)$ la tête de liste.
2. Implémentation impérative: avec un tableau (dynamique), on conserve l'indice du haut de la pile.

Pile

Il est possible de définir un type abstrait pile, ce qui permet ensuite d'avoir des algorithmes indépendants de l'implémentation:

```
type 'a pile =
  { push : 'a -> unit;
    pop : unit -> 'a;
    is_empty : unit -> bool };;

let of_list l =
  let p = ref l in
  { push = (fun e -> p := e::!p);
    pop = (fun () -> let res = List.hd !p in
                     p := List.tl !p;
                     res);
    is_empty = (fun () -> !p = []) };;
```

On créé ensuite une fonction de création `of_...` par implémentation concrète.

Pile

Application:

Pile

Application: suppression de la récursivité.

On peut simuler les appels récursifs d'une fonction en empilant les arguments correspondants.

Hanoï sans récursivité

```
let hanoi n =
  let p = of_list [] in
  p.push (n, 0, 2);
  while not p.is_empty ()
  do
    let (nb, deb, fin) = p.pop () in
    if nb = 1 then (* afficher deplacement *)
    else begin
      let mid = 3 - deb - fin in
      p.push (nb - 1, mid, fin);
      p.push (1, deb, fin);
      p.push (nb - 1, deb, mid);
    end
  done;;
```

Complexité (= nombre de déplacements):

Hanoï sans récursivité

```
let hanoi n =
  let p = of_list [] in
  p.push (n, 0, 2);
  while not p.is_empty ()
  do
    let (nb, deb, fin) = p.pop () in
    if nb = 1 then (* afficher deplacement *)
    else begin
      let mid = 3 - deb - fin in
      p.push (nb - 1, mid, fin);
      p.push (1, deb, fin);
      p.push (nb - 1, deb, mid);
    end
  done;;
```

Complexité (= nombre de déplacements):
$C(n) = 2C(n-1) + 1$

Hanoï sans récursivité

```
let hanoi n =
  let p = of_list [] in
  p.push (n, 0, 2);
  while not p.is_empty ()
  do
    let (nb, deb, fin) = p.pop () in
    if nb = 1 then (* afficher deplacement *)
    else begin
      let mid = 3 - deb - fin in
      p.push (nb - 1, mid, fin);
      p.push (1, deb, fin);
      p.push (nb - 1, deb, mid);
    end
  done;;
```

Complexité (= nombre de déplacements):

$C(n) = 2C(n-1) + 1 = 2^2 C(n-2) + 2 + 1$

$= 2^{n-1} C(n-(n-1)) + 2^{n-2} + ... + 1 = \sum_{k=0}^{n-1} 2^k = \boxed{2^n - 1}$

File

Une file est aussi appelée FIFO: First In, First Out.

1ère idée d'implémentation:

File

Une file est aussi appelée FIFO: First In, First Out.

1ère idée d'implémentation: avec une liste, on supprime en tête en $O(1)$ et on ajoute en fin de liste

File

Une file est aussi appelée FIFO: First In, First Out.

1ère idée d'implémentation: avec une liste, on supprime en tête en O(1) et on ajoute en fin de liste en $\Theta(n)$, bof!

File persistante

Comment créer une file persistante efficace?

File persistante

Comment créer une file persistante efficace?

Implémentation au programme, avec deux listes:

```
type 'a file = { debut : 'a list; fin : 'a list };;

let create () = { debut = []; fin = [] };;

let add e f = { debut = f.debut; fin = e::f.fin };;

let rec take f = match f.debut with
  | [] -> take { debut = List.rev f.fin; fin = [] }
  | e::q -> (e, { debut = q; fin = f.fin });;

let is_empty f = f.debut = [] && f.fin = [];;
```

File persistante avec 2 piles (listes)

```ocaml
type 'a file = { debut : 'a list; fin : 'a list };;

let create () = { debut = []; fin = [] };;

let add e f = { debut = f.debut; fin = e::f.fin };;

let rec take f = match f.debut with
  | [] -> take { debut = List.rev f.fin; fin = [] }
  | e::q -> (e, { debut = q; fin = f.fin });;

let is_empty f = f.debut = [] && f.fin = [];;
```

Complexité dans le pire des cas de:

1. `add`, `create`, `is_empty`:

File persistante avec 2 piles (listes)

```
type 'a file = { debut : 'a list; fin : 'a list };;

let create () = { debut = []; fin = [] };;

let add e f = { debut = f.debut; fin = e::f.fin };;

let rec take f = match f.debut with
  | [] -> take { debut = List.rev f.fin; fin = [] }
  | e::q -> (e, { debut = q; fin = f.fin });;

let is_empty f = f.debut = [] && f.fin = [];;
```

Complexité dans le pire des cas de:

1. `add`, `create`, `is_empty`: $O(1)$
2. `take`:

File persistante avec 2 piles (listes)

```
type 'a file = { debut : 'a list; fin : 'a list };;

let create () = { debut = []; fin = [] };;

let add e f = { debut = f.debut; fin = e::f.fin };;

let rec take f = match f.debut with
  | [] -> take { debut = List.rev f.fin; fin = [] }
  | e::q -> (e, { debut = q; fin = f.fin });;

let is_empty f = f.debut = [] && f.fin = [];;
```

Complexité dans le pire des cas de:

1. `add`, `create`, `is_empty`: $O(1)$
2. `take`: $O(n)$, où n est le nombre d'éléments de la file

File persistante avec 2 listes

Si on effectue n `add` et n `take` dans un ordre quelconque (en partant d'une file vide), quelle sera la complexité totale des n `take`?

File persistante avec 2 listes

Si on effectue n `add` et n `take` dans un ordre quelconque (en partant d'une file vide), quelle sera la complexité totale des n `take`?

Chaque élément est « renversé » exactement une fois, donc la complexité totale des `rev` est $O(n)$. Donc la complexité totale des n `take` est $O(n)$.
La **complexité amortie** d'un `take` est donc O(1)

Arbre persistant

Arbre binaire:

```
type 'a arbre_b = V | N of 'a * 'a arbre_b * 'a arbre_b;;
```

On peut prouver qu'une proposition/un programme $\mathcal{P}$ est correct sur les arbres binaires en montrant:

1. $\mathcal{P}(\texttt{V})$
2. $\mathcal{P}(\texttt{g}) \land \mathcal{P}(\texttt{d}) \implies \forall\ \texttt{r}, \mathcal{P}(\texttt{N(r, g, d)})$

Exemple: prouver que $f \leq 2^h$.

Arbre persistant

Arbre binaire:

```
type 'a arbre_b = V | N of 'a * 'a arbre_b * 'a arbre_b;;
```

On peut prouver qu'une proposition/un programme $\mathcal{P}$ est correct sur les arbres binaires en montrant:

1. $\mathcal{P}(\texttt{V})$
2. $\mathcal{P}(\texttt{g}) \land \mathcal{P}(\texttt{d}) \implies \forall\ \texttt{r},\ \mathcal{P}(\texttt{N(r, g, d)})$

Exemple: prouver que $f \leq 2^h$.

Arbre quelconque: `type 'a arbre = N of 'a * 'a arbre list`

Accès aux fils: O(1)
Accès au père:

Arbre persistant

Arbre binaire:

```
type 'a arbre_b = V | N of 'a * 'a arbre_b * 'a arbre_b;;
```

On peut prouver qu'une proposition/un programme $\mathcal{P}$ est correct sur les arbres binaires en montrant:

1. $\mathcal{P}(\texttt{V})$
2. $\mathcal{P}(\texttt{g}) \land \mathcal{P}(\texttt{d}) \implies \forall\ \texttt{r},\ \mathcal{P}(\texttt{N(r, g, d)})$

Exemple: prouver que $f \leq 2^h$.

Arbre quelconque: `type 'a arbre = N of 'a * 'a arbre list`

Accès aux fils: O(1)
Accès au père: impossible

Arbre impératif avec tableau

On peut aussi représenter un arbre à n sommets par un tableau `t` de taille $\geq n$ où `t.(i)` est le père du $i$ème sommet.

Accès aux fils:

Arbre impératif avec tableau

On peut aussi représenter un arbre à n sommets par un tableau `t` de taille $\geq n$ où `t.(i)` est le père du $i$ème sommet.

Accès aux fils: $O(n)$
Accès au père: $O(1)$
Ajout de sommet: impossible (sauf si tableau dynamique)

Hauteur d'un arbre

La **hauteur** d'un arbre est la longueur maximum (en nombre d'arêtes = « traits ») d'un chemin de la racine à une feuille.

Hauteur maximum d'un arbre binaire à n sommets:

Hauteur d'un arbre

La **hauteur** d'un arbre est la longueur maximum (en nombre d'arêtes $=$ « traits ») d'un chemin de la racine à une feuille.

Hauteur maximum d'un arbre binaire à n sommets: $n-1$.
Hauteur minimum d'un arbre binaire à n sommets:

Hauteur d'un arbre

La **hauteur** d'un arbre est la longueur maximum (en nombre d'arêtes $=$ « traits ») d'un chemin de la racine à une feuille.

Hauteur maximum d'un arbre binaire à n sommets: $n-1$.
Hauteur minimum d'un arbre binaire à n sommets: $\lfloor \log_2(n) \rfloor$.

Diamètre d'un arbre

Exercice: comment calculer le **diamètre** (la longueur maximum d'un chemin entre deux sommets) d'un arbre à n sommets?

Diamètre d'un arbre

Exercice: comment calculer le **diamètre** (la longueur maximum d'un chemin entre deux sommets) d'un arbre à n sommets?

```
let rec diam a = match a with
  | V -> -1
  | N(r, g, d) -> max (ht g + ht d + 2) (max (diam g) (diam d));;
```

Complexité:

Diamètre d'un arbre

Exercice: comment calculer le **diamètre** (la longueur maximum d'un chemin entre deux sommets) d'un arbre à n sommets?

```
let rec diam a = match a with
  | V -> -1
  | N(r, g, d) -> max (ht g + ht d + 2) (max (diam g) (diam d));;
```

Complexité: $O(n)$ appels à `ht`, donc $O(n^2)$.

Diamètre d'un arbre

Exercice: comment calculer le **diamètre** (la longueur maximum d'un chemin entre deux sommets) d'un arbre à n sommets?

```
let rec diam a = match a with
  | V -> -1
  | N(r, g, d) -> max (ht g + ht d + 2) (max (diam g) (diam d));;
```

Complexité: $O(n)$ appels à `ht`, donc $O(n^2)$.

```
let rec diam a = match a with (* renvoie (diamètre, hauteur) *)
  | V -> -1, -1
  | N(r, g, d) -> let dg, hg = diam g in
                  let dd, hd = diam d in
                  max (hg + hd + 2) (max dg dd), 1 + max hg hd;;
```

Complexité:

Diamètre d'un arbre

Exercice: comment calculer le **diamètre** (la longueur maximum d'un chemin entre deux sommets) d'un arbre à n sommets?

```
let rec diam a = match a with
  | V -> -1
  | N(r, g, d) -> max (ht g + ht d + 2) (max (diam g) (diam d));;
```

Complexité: $O(n)$ appels à `ht`, donc $O(n^2)$.

```
let rec diam a = match a with (* renvoie (diamètre, hauteur) *)
  | V -> -1, -1
  | N(r, g, d) -> let dg, hg = diam g in
                  let dd, hd = diam d in
                  max (hg + hd + 2) (max dg dd), 1 + max hg hd;;
```

Complexité: $O(n)$

Arbre binaire de recherche

Un arbre binaire `a` est un ABR si et seulement si...

1. ... pour chaque noeud `N(r, g, d)` de `a`, `r` est supérieur à toutes les étiquettes de `g` et inférieur à celles de `d`?
2. ... chaque noeud de `a` possède une étiquette r supérieure à celle de son fils gauche et inférieure à celle de son fils droit?
3. ... `a = V` ou `a = N(r, g, d)` avec `g`, `d` ABR et r est supérieur à l'étiquette de son fils gauche et inférieur à celle de son fils droit?
4. ... la liste du parcours infixe de `a` est croissante?

Arbre binaire de recherche

Un arbre binaire `a` est un ABR si et seulement si...

1. ... pour chaque noeud interne `N(r, g, d)` de `a`, `r` est supérieur à toutes les étiquettes de `g` et inférieur à celles de `d` ✓
2. ... ~~chaque noeud interne de a possède une étiquette r supérieure à celle de son fils gauche et inférieure à celle de son fils droit~~
3. ... ~~a = V ou a = N(r, g, d) avec g, d ABR et r est supérieur à l'étiquette de son fils gauche et inférieur à celle de son fils droit~~
4. ... la liste du parcours infixe de `a` est croissante ✓
   $\Rightarrow$ donne **algorithme de tri**

Arbre binaire de recherche

$$\texttt{N(r, g, d)}\ \text{ABR} \implies \texttt{g}\ \text{ABR et}\ \texttt{d}\ \text{ABR}$$

On dit que la propriété d'ABR est **héréditaire**: si elle vraie sur un arbre, elle est vraie pour tous ses sous-arbres.

Arbre binaire de recherche

L'utilisation d'un ABR suppose que l'on dispose d'une **relation d'ordre** sur les éléments, souvent sous forme d'une fonction de comparaison.

Pour simplifier l'écriture des opérations sur un ABR, on utilise ici <, qui ne fonctionne qu'avec des entiers, flottants, string et tuples (l'ordre lexicographique est utilisé pour les tuples).

Arbre binaire de recherche

Recherche d'un élément e dans un ABR:

Arbre binaire de recherche

Recherche d'un élément e dans un ABR:

```
let rec mem e abr = match abr with
  | V -> false
  | N(r, g, d) when e = r -> true
  | N(r, g, d) when e < r -> mem e g
  | N(r, g, d) -> mem e d;;
```

Complexité:

Arbre binaire de recherche

Recherche d'un élément e dans un ABR:

```
let rec mem e abr = match abr with
  | V -> false
  | N(r, g, d) when e = r -> true
  | N(r, g, d) when e < r -> mem e g
  | N(r, g, d) -> mem e d;;
```

Complexité: $\Theta(h)$ dans le pire des cas (la profondeur du sommet visité augmente à chaque appel récursif, et h est la profondeur maximum).

Arbre binaire de recherche

Ajouter un élément e (en tant que feuille) dans un ABR:

Arbre binaire de recherche

Ajouter un élément e (en tant que feuille) dans un ABR:

```
let rec add e abr = match abr with
  | V -> N(e, V, V)
  | N(r, g, d) when e < r -> N(r, add e g, d)
  | N(r, g, d) -> N(r, g, add e d);;
```

Complexité:

Arbre binaire de recherche

Ajouter un élément e (en tant que feuille) dans un ABR:

```
let rec add e abr = match abr with
  | V -> N(e, V, V)
  | N(r, g, d) when e < r -> N(r, add e g, d)
  | N(r, g, d) -> N(r, g, add e d);;
```

Complexité: $\Theta(h)$ dans le pire des cas (la profondeur du sommet visité augmente à chaque appel récursif, et h est la profondeur maximum).

Arbre binaire de recherche

Comment créer un ABR à partir d'une liste?

Arbre binaire de recherche

Comment créer un ABR à partir d'une liste?

```
let rec abr_of_list = function
  | [] -> V
  | e::q -> add e (abr_of_list q);;
```

Complexité:

Arbre binaire de recherche

Comment créer un ABR à partir d'une liste?

```
let rec abr_of_list = function
  | [] -> V
  | e::q -> add e (abr_of_list q);;
```

Complexité: $\Theta(n^2)$ dans le pire des cas (on effectue n `add` chacun en $\Theta(n)$ au pire).

On en déduit un tri:

Arbre binaire de recherche

Comment créer un ABR à partir d'une liste?

```
let rec abr_of_list = function
  | [] -> V
  | e::q -> add e (abr_of_list q);;
```

Complexité: $\Theta(n^2)$ dans le pire des cas (on effectue n `add` chacun en $\Theta(n)$ au pire).

On en déduit un tri:

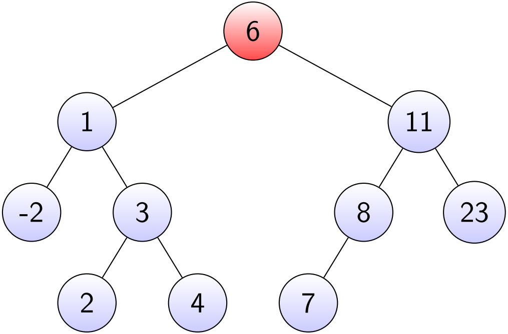
```
let tri l = infixe (abr_of_list l);;
```

Arbre binaire de recherche

Supprimer un élément e (ici 6) dans un ABR:

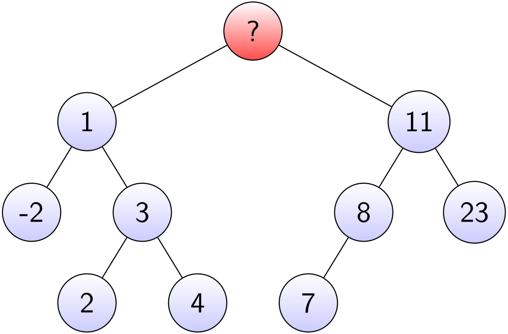
Arbre binaire de recherche

Supprimer un élément e (ici 6) dans un ABR:



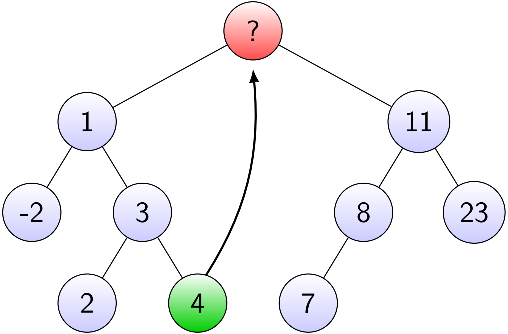
Arbre binaire de recherche

Supprimer un élément e (ici 6) dans un ABR:



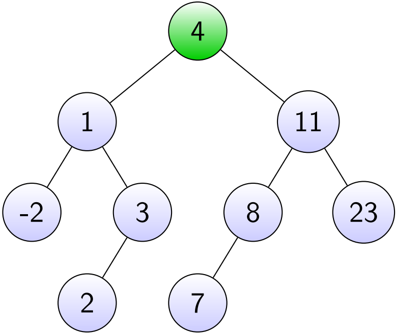
Arbre binaire de recherche (ABR)

Supprimer un élément e (ici 6) dans un ABR:



Arbre binaire de recherche

Supprimer un élément e (ici 6) dans un ABR:



Arbre binaire de recherche

Pour supprimer et renvoyer le maximum d'un ABR:

Arbre binaire de recherche

Pour supprimer et renvoyer le maximum d'un ABR:

```
let rec del_max abr = match abr with
  | N(r, g, V) -> r, g
  | N(r, g, d) -> let m, d' = del_max d in
                  m, N(r, g, d');;
```

Pour supprimer un élément e d'un ABR:

Arbre binaire de recherche

Pour supprimer et renvoyer le maximum d'un ABR:

```
let rec del_max abr = match abr with
  | N(r, g, V) -> r, g
  | N(r, g, d) -> let m, d' = del_max d in
                  m, N(r, g, d');;
```

Pour supprimer un élément e d'un ABR:

```
let rec del e abr = match abr with
  | N(r, V, d) when e = r -> d
  | N(r, g, d) when e = r -> let m, g' = del_max g in
                             N(m, g', d)
  | N(r, g, d) when e < r -> N(r, del e g, d)
  | N(r, g, d) -> N(r, g, del e d);;
```

Complexité:

Arbre binaire de recherche

Pour supprimer et renvoyer le maximum d'un ABR:

```
let rec del_max abr = match abr with
  | N(r, g, V) -> r, g
  | N(r, g, d) -> let m, d' = del_max d in
                  m, N(r, g, d');;
```

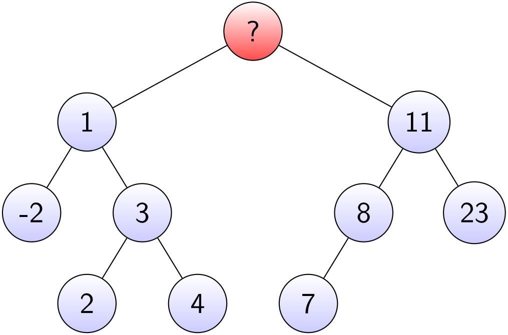
Pour supprimer un élément e d'un ABR:

```
let rec del e abr = match abr with
  | N(r, V, d) when e = r -> d
  | N(r, g, d) when e = r -> let m, g' = del_max g in
                             N(m, g', d)
  | N(r, g, d) when e < r -> N(r, del e g, d)
  | N(r, g, d) -> N(r, g, del e d);;
```

Complexité: chacune en $\Theta(h)$ dans le pire des cas.

Arbre binaire de recherche

Supprimer un élément e (ici 6) dans un ABR:



Autre possibilité (TD): fusionner les sous-arbres gauches et droits.

Arbre binaire de recherche

Opération	ABR	AVL (TD)...
ajouter élément	O(h) (pire cas: $\Theta(n)$)	O($\log(n)$)
supprimer élément	O(h) (pire cas: $\Theta(n)$)	O($\log(n)$)
rechercher élément	O(h) (pire cas: $\Theta(n)$)	O($\log(n)$)

Il existe de nombreuses façons de garantir un ABR *équilibré* (de hauteur O($\log(n)$)): AVL, arbres rouge-noir, arbres 2-3...

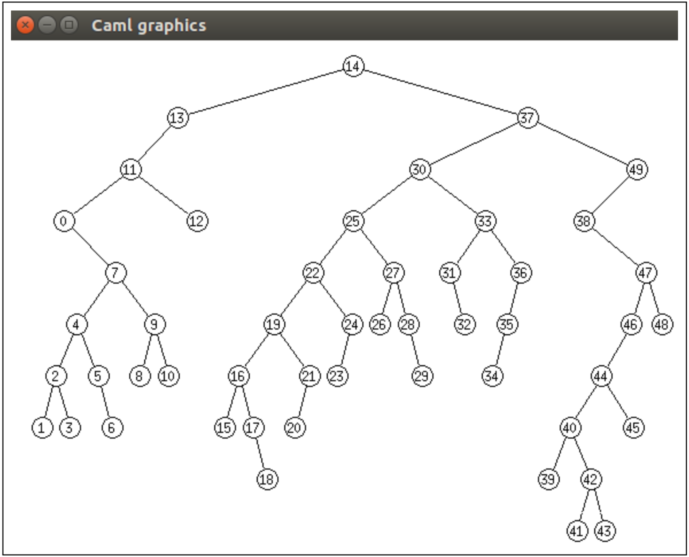
Arbre binaire de recherche aléatoire

1. On peut montrer que la hauteur moyenne d'un ABR à n sommets construit aléatoirement est $\Theta(\log(n))$, donc que les opérations d'ABR sont en moyenne $\Theta(\log(n))$.
   Exercice de programmation: le vérifier expérimentalement.
   (Exercice difficile: le prouver mathématiquement.)

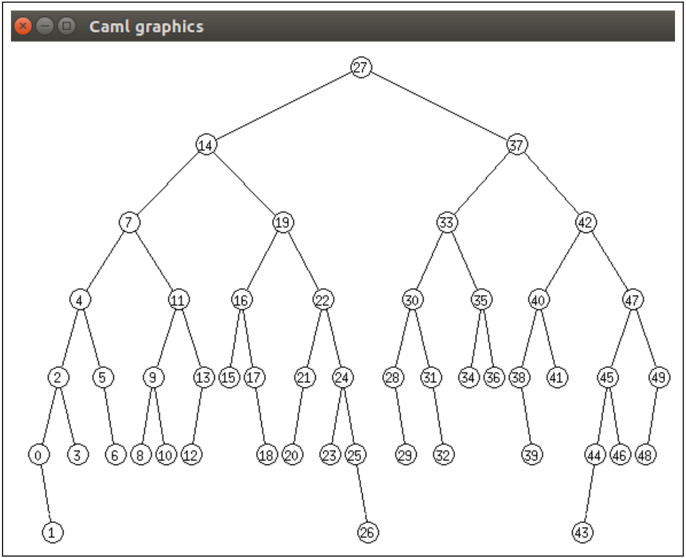
Arbre binaire de recherche aléatoire

1. On peut montrer que la hauteur moyenne d'un ABR à n sommets construit aléatoirement est $\Theta(\log(n))$, donc que les opérations d'ABR sont en moyenne $\Theta(\log(n))$.
   Exercice de programmation: le vérifier expérimentalement.
   (Exercice difficile: le prouver mathématiquement.)
2. Cependant, la hauteur moyenne d'un arbre binaire à n sommets choisi uniformément au hasard parmi tous les arbres binaires est $\sim 2\sqrt{\pi n}$.

ABR construit aléatoirement



AVL construit aléatoirement



Ensemble (set)

La structure d'ensemble possède les opérations:

1. ajouter un élément
2. supprimer un élément
3. test d'appartenance

Ensemble (set)

On peut définir un type abstrait d'ensemble en OCaml:

```
type 'a set =
  { add : 'a -> unit;
    del : 'a -> unit;
    has : 'a -> bool };;
```

Ensemble (set)

On peut définir un type abstrait d'ensemble en OCaml:

```
type 'a set =
  { add : 'a -> unit;
    del : 'a -> unit;
    has : 'a -> bool };;
```

Quelques implémentations possibles d'ensemble:

Ensemble (set)

On peut définir un type abstrait d'ensemble en OCaml:

```
type 'a set =
  { add : 'a -> unit;
    del : 'a -> unit;
    has : 'a -> bool };;
```

Quelques implémentations possibles d'ensemble:

Opération	Liste	ABR
ajouter	$O(1)$	$O(h)$
supprimer	$O(n)$	$O(h)$
rechercher	$O(n)$	$O(h)$

Si on utilise un ABR équilibré (par exemple un AVL), on obtient des complexités $O(\log(n))$.

Application

Question

Comment supprimer les doublons dans une liste de taille n?

Application

Question

Comment supprimer les doublons dans une liste de taille n?

On peut faire mieux que la méthode naïve en $\Theta(n^2)$:

Application

Question

Comment supprimer les doublons dans une liste de taille n?

On peut faire mieux que la méthode naïve en $\Theta(n^2)$:

1. trier en $O(n \log(n))$ puis supprimer les éléments consécutifs égaux en $O(n)$

Application

Question

Comment supprimer les doublons dans une liste de taille n?

On peut faire mieux que la méthode naïve en $\Theta(n^2)$:

1. trier en $O(n \log(n))$ puis supprimer les éléments consécutifs égaux en $O(n)$
2. (sans modifier l'ordre des éléments)

Application

Question

Comment supprimer les doublons dans une liste de taille n?

On peut faire mieux que la méthode naïve en $\Theta(n^2)$:

1. trier en $O(n \log(n))$ puis supprimer les éléments consécutifs égaux en $O(n)$
2. (sans modifier l'ordre des éléments) stocker chaque élément dans une structure d'ensemble (`set`) et le supprimer si il y est déjà présent:

Application

Question

Comment supprimer les doublons dans une liste de taille n?

On peut faire mieux que la méthode naïve en $\Theta(n^2)$:

1. trier en $O(n \log(n))$ puis supprimer les éléments consécutifs égaux en $O(n)$
2. (sans modifier l'ordre des éléments) stocker chaque élément dans une structure d'ensemble (`set`) et le supprimer si il y est déjà présent: $O(n \log(n))$ avec AVL.

Suppression des doublons avec un ensemble (set)

```
let doublons l =
  let s = new_set () in
  let rec aux = function
    | [] -> []
    | e::q -> if s.has e then aux q
              else (s.add e;
                    e::aux q)
  in aux l;;
```

Dictionnaire

Un dictionnaire gère un ensemble de couples (clé, valeur) et possède les opérations:

1. ajouter un couple (clé, valeur)
2. rechercher les valeurs associées à une clé
3. supprimer un couple (clé, valeur)

Il peut y avoir plusieurs valeurs pour la même clé.

Dictionnaire

Un dictionnaire gère un ensemble de couples (clé, valeur) et possède les opérations:

1. ajouter un couple (clé, valeur)
2. rechercher les valeurs associées à une clé
3. supprimer un couple (clé, valeur)

Il peut y avoir plusieurs valeurs pour la même clé.

Dans le cas où les clés sont des entiers consécutifs, on peut utiliser

Dictionnaire

Un dictionnaire gère un ensemble de couples (clé, valeur) et possède les opérations:

1. ajouter un couple (clé, valeur)
2. rechercher les valeurs associées à une clé
3. supprimer un couple (clé, valeur)

Il peut y avoir plusieurs valeurs pour la même clé.

Dans le cas où les clés sont des entiers consécutifs, on peut utiliser un tableau.

Dictionnaire

On peut définir un type abstrait de dictionnaire en Caml:

```
type ('key, 'value) dico =
  { add : 'key -> 'value -> unit;
    get : 'key -> 'value list;
    del : 'key -> unit };;
```

(`get k` renvoie `[]` si la clé *k* n'existe pas)

Dictionnaire

Puis on peut définir une fonction de création de dictionnaire avec une implémentation concrète (ici une liste de couples (clé, valeur)):

```
let of_list l =
  let di = ref l in
  { add = (fun k v -> di := (k, v)::!di);
    get = (fun k ->
      let rec aux = function
        | [] -> []
        | (k', v)::q when k = k' -> v::aux q
        | e::q -> aux q in
      aux !di);
    del = (fun k ->
      let rec aux = function
        | [] -> []
        | (k', v)::q when k' = k -> aux q
        | e::q -> e::aux q in
      di := aux !di)
  };;
```

Dictionnaire

Quelques implémentations possibles de dictionnaire:

Opération	Liste de couples	ABR équilibré	Table de hachage
ajouter	$O(1)$	$O(\log(n))$	$O(1)$ en moyenne
supprimer	$O(n)$	$O(\log(n))$	$O(1)$ en moyenne
rechercher	$O(n)$	$O(\log(n))$	$O(1)$ en moyenne

L'utilisation d'un ABR demande une relation d'ordre sur les clés.
L'utilisation d'une table de hachage demande une fonction de hachage sur les clés.

Table de hachage

Une table de hachage est constituée:

1. d'un tableau (dynamique) `t` contenant les valeurs
2. d'une fonction de hachage h de l'ensemble des clés vers les indices de `t`

La valeur associée à une clé c est stockée à l'indice $h(c)$ du tableau `t`.

Table de hachage

Une table de hachage est constituée:

1. d'un tableau (dynamique) `t` contenant les valeurs
2. d'une fonction de hachage h de l'ensemble des clés vers les indices de `t`

La valeur associée à une clé c est stockée à l'indice $h(c)$ du tableau `t`.

Si la même clé est associée à plusieurs valeurs alors `t` doit être un tableau de listes.

Table de hachage

Une table de hachage est constituée:

1. d'un tableau (dynamique) `t` contenant les valeurs
2. d'une fonction de hachage h de l'ensemble des clés vers les indices de `t`

La valeur associée à une clé c est stockée à l'indice $h(c)$ du tableau `t`.

Si la même clé est associée à plusieurs valeurs alors `t` doit être un tableau de listes.

Si les clés sont des entiers (non consécutifs), on peut choisir $h : x \longmapsto x \mod n$.

Sous quelques hypothèses, on peut montrer que les opérations de table de hachage sont en complexité moyenne O(1).

Application des dictionnaires

La suite de Syracuse est définie par:

$$u_{n+1} = \begin{cases} \dfrac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases}$$

$$u_0 = a$$

Application des dictionnaires

La suite de Syracuse est définie par:

$$u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases}$$

$$u_0 = a$$

Conjecture de Syracuse: $\forall a \geq 1$, $\exists t$, $u_t = 1$.

On note $t(a)$ le plus petit t tel que $u_t = 1$ et on veut calculer $t(a)$, $\forall a \leq \texttt{nmax}$.

Application des dictionnaires

La suite de Syracuse est définie par:

$$u_{n+1} = \begin{cases} \dfrac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases}$$

$$u_0 = a$$

Conjecture de Syracuse: $\forall a \geq 1$, $\exists t$, $u_t = 1$.

On note $t(a)$ le plus petit t tel que $u_t = 1$ et on veut calculer $t(a)$, $\forall a \leq \texttt{nmax}$.

Pour $a = 3$: $(u_n) = (3, 10, 5, 16, 8, 4, 2, 1, ...)$ et $t(3) = 7$.

Application des dictionnaires

La suite de Syracuse est définie par:

$$u_{n+1} = \begin{cases} \dfrac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases}$$

$$u_0 = a$$

Conjecture de Syracuse: $\forall a \geq 1$, $\exists t$, $u_t = 1$.

On note $t(a)$ le plus petit t tel que $u_t = 1$ et on veut calculer $t(a)$, $\forall a \leq$ `nmax`.

Pour $a = 3$: $(u_n) = (3, 10, 5, 16, 8, 4, 2, 1, ...)$ et $t(3) = 7$.
On a donc déjà calculé $t(10) = 6$, $t(5) = 5$, ...

Application des dictionnaires

On aimerait faire de la programmation dynamique mais les termes de la suite sont « imprévisibles »: on ne sait pas dans quel ordre les calculer.

Application des dictionnaires

On aimerait faire de la programmation dynamique mais les termes de la suite sont « imprévisibles »: on ne sait pas dans quel ordre les calculer.

Mémoïsation: on stocke tous les $t(a)$ calculés dans un dictionnaire. Si on doit calculer un certain $t(a)$ on vérifie s'il n'a pas déjà été calculé.

Application des dictionnaires

```
let rec t a = match di.get a with (* t a renvoie le temps de vol pour u0 = a *)
  | [] -> let res = 1 + t (if a mod 2 = 0 then a/2 else 3*a + 1) in
          di.add a res;
          res
  | e::q -> e;; (* le temps de vol a déjà été calculé *)

for a = 2 to 100 do (* stocke les temps de vol dans di *)
  t a
done;;
```

Ensemble en Python

```
In [12]: s = {2, 3, 5, 7, 11, 13}

In [13]: 3 in s
Out[13]: True

In [14]: len(s)
Out[14]: 6

In [15]: s.add(17)

In [16]: s
Out[16]: {2, 3, 5, 7, 11, 13, 17}
```

Dictionnaire en Python

```
In [4]: d = {"red" : (255, 0, 0), "green" : (0, 255, 0), "blue" : (0, 0, 255)}

In [5]: d["green"]
Out[5]: (0, 255, 0)

In [6]: d["yellow"] = (255,255,0) # ajout d'une valeur au dictionnaire

In [7]: d["yellow"]
Out[7]: (255, 255, 0)

In [8]: len(d)
Out[8]: 4

In [9]: "pink" in d
Out[9]: False

In [10]: "blue" in d
Out[10]: True
```