Arbres bien équilibrés

Dictionnaire

(aussi appelés tableaux associatifs ou tables d'association)

On veut associer des clefs $k \in \mathcal{K}ey$ à des valeurs $v \in \mathcal{V}al$

Il est possible

- de créer un dictionnaire vide
- d'insérer une nouvelle association
- de rechercher à quelle valeur est associée une clef
- de supprimer une association

En général, $\mathcal{K}ey$ est supposé totalement ordonné.
Dans la suite, on supposera qu'à une clef n'est associée qu'une seule valeur au maximum (le dictionnaire est une fonction au sens mathématique)

Exemples d'utilisation

Omniprésent en informatique :

- Table des symboles dans un compilateur
  $\mathcal{K}ey=$ symboles, $\mathcal{V}al=$ informations (type, visibilité, ...)
- Système de fichier
  $\mathcal{K}ey=$ chemins, $\mathcal{V}al=$ emplacements disque
- Mémoïsation
  $\mathcal{K}ey=$ arguments, $\mathcal{V}al=$ résultats
- Moteur de recherche
  $\mathcal{K}ey=$ mots-clefs, $\mathcal{V}al=$ pages associées
- Représentation d'un ensemble
  $\mathcal{K}ey=$ élément, $\mathcal{V}al=\{est\_dedans\}$
- ...

Spécification d'un dictionnaire – Interface

```
type ('k,'v) dict

val create :
  unit -> ('k,'v) dict
val add :
  ('k,'v) dict -> 'k -> 'v -> ('k,'v) dict
val find :
  ('k,'v) dict -> 'k -> 'v
val remove :
  ('k,'v) dict -> 'k -> ('k,'v) dict
```

Spécification d'un dictionnaire – Propriétés

- une recherche dans un dictionnaire vide renvoie une erreur
- une recherche de la clef `k` dans le dictionnaire où l'on vient d'ajouter l'association $\texttt{k} \mapsto \texttt{v}$ renvoie `v`
- une recherche de la clef `k` dans le dictionnaire où l'on vient d'ajouter l'association $\texttt{k'} \mapsto \texttt{v}$ renvoie avec $\texttt{k} \neq \texttt{k'}$ renvoie la même valeur que la recherche avant l'ajout
- une recherche de la clef `k` dans le dictionnaire où l'on vient de retirer l'association à `k` renvoie une erreur
- une recherche de la clef `k` dans le dictionnaire où l'on vient de retirer l'association à `k'` avec $\texttt{k} \neq \texttt{k'}$ renvoie la même valeur que la recherche avant la suppression

Implémentation à l'aide de listes d'associations

```
type ('k,'v) dict = ('k * 'v) list

let create _ = []                                    O(1)

let add d k v = (k,v)::d                             O(1)

let find d k = snd (                                 O(n)
  List.find (fun (k', _) -> k = k') d )

let remove d k =                                     O(n)
  List.filter (fun (k', _) -> k <> k') d
```

Recherche par dichotomie

on aimerait avoir des opérations de recherche, d'insertion et de suppression efficaces en moyenne et dans le pire des cas

tableau trié

- recherche en $O(\log n)$ par dichotomie

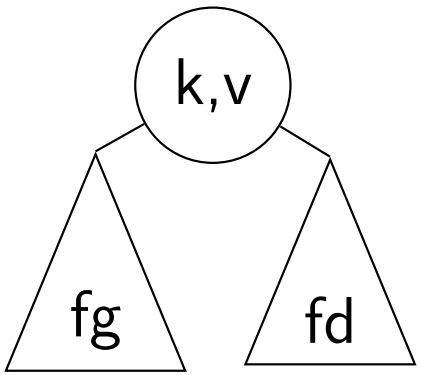
mais

- tri du tableau en $O(n \log n)$
- insertion et suppression coûteuses ($O(n)$, décalages)

Arbres binaires de recherche (ABR)

Structure de données pour exploiter la recherche par dichotomie

Arbre dont les nœuds contiennent des paires (clef, valeur), possédant au plus deux fils.

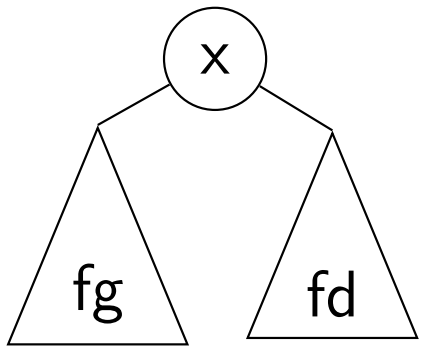


Invariant : pour un nœud

- les clefs dans le sous-arbre gauche fg sont toutes plus petites que k
- les clefs dans le sous-arbre droit fd sont toutes plus grandes que k
- fg et fd vérifie l'invariant

Preuve par induction

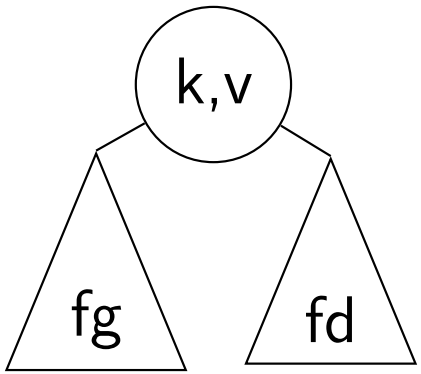
Pour montrer que P est vrai pour tout arbre binaire, il suffit de montrer que

- P est vrai pour l'arbre vide
- P est vrai pour 

si on suppose que P est vrai pour fg et fd

Plus approprié qu'une récurrence sur la hauteur ou la taille de l'arbre

Recherche

rechercher k' dans 

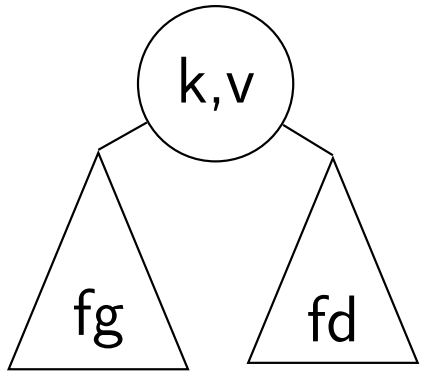
- si k' $=$ k, retourner v
- si k' $<$ k et fg est non-vide, rechercher k' dans fg
- si k' $>$ k et fd est non-vide, rechercher k' dans fd

Complexité : $O(h)$

- à chaque appel récursif, on diminue la hauteur de l'arbre d'au moins 1
- si la bonne clef se trouve au plus bas de l'arbre, il faudra h étapes, où h est la hauteur de l'arbre

Insertion

insérer k',v' dans l'arbre vide : retourner (k',v')

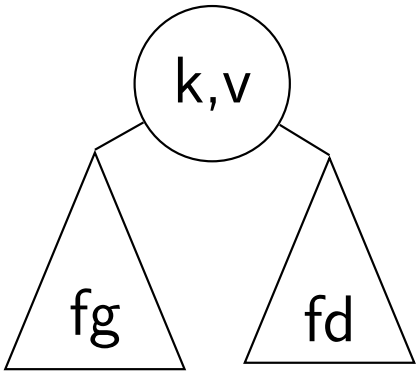
insérer k',v' dans 

- si k' $<$ k, insérer k',v' dans fg
- si k' $>$ k, insérer k',v' dans fd

Complexité : $O(h)$

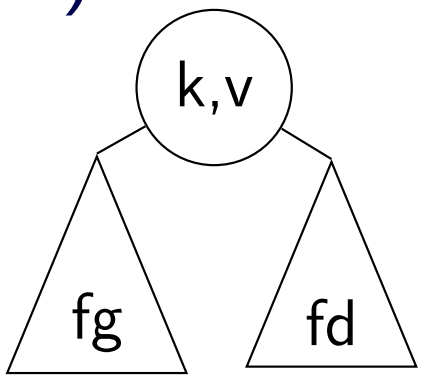
- à chaque appel récursif, on diminue la hauteur de l'arbre d'au moins 1
- on peut avoir à descendre jusqu'au nœud le plus bas

Suppression

Supprimer k dans 

- si fg vide, on retourne fd
- si fd vide, on retourne fg
- sinon,
  - on recherche l'association (k',v') avec la plus grande clef dans fg (celle la plus à droite)
  - on la met à la place de la racine
  - on supprime k' dans fg

Suppression (suite)

Supprimer k' dans 

avec k $\neq$ k' :

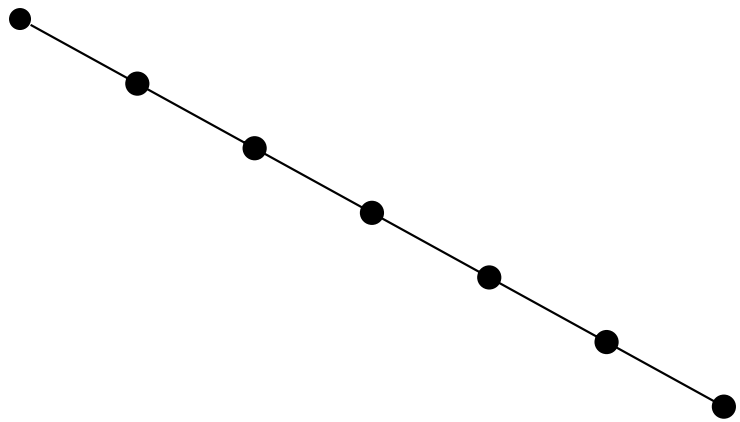
- si k' $<$ k, supprimer k' dans fg
- si k' $>$ k, supprimer k' dans fd

Complexité : $O(h)$

- si la clef à supprimer est dans le nœud le plus à droite, $O(h)$
- donc si la clef à supprimer est à la racine, recherche du plus à droite en $O(h)$ et suppression en $O(h) \Rightarrow O(h)$
- si la clef n'est pas à la racine, on fait un appel récursif sur un arbre de hauteur $h-1$ au maximum $\Rightarrow O(h)$

Complexité dans le pire des cas

La hauteur d'un arbre à n nœuds est n dans le pire des cas :



En particulier, on obtient un arbre de ce type si on insère les éléments par ordre croissant

La complexité dans le pire des cas est $O(n)$ pour rechercher, insérer et supprimer

Complexité en moyenne

La hauteur moyenne d'un arbre à n nœuds est $\sqrt{n}$

Toutefois, si on considère les arbres obtenus en insérant les éléments de $[0 \cdots n-1]$ suivant toutes les permutations possibles, la hauteur moyenne d'un arbre est $\log n$

La complexité en moyenne est donc $O(\log n)$ pour rechercher, insérer et supprimer

Implémentation

```
type ('k,'v) dict =
    Nil
  | Bin of ('k,'v) dict * ('k * 'v) * ('k,'v)

let create _ = Nil

let rec find d k =
  match d with
     Nil -> raise Not_found
  | Bin(_, (k',v), _) when k = k' -> v
  | Bin(fg,(k',_), _) when k < k' ->
      find fg k
  | Bin(_, (k',_),fd)  -> find fd k
```

Implémentation (suite)

```
let rec add d k v =
  match d with
      Nil -> Bin(Nil,(k,v),Nil)
    | Bin(_, (k', _), _) when} k = k' ->
        failwith "cas␣non␣traité"
    | Bin(fg,(k',v'),fd) when} k < k' ->
        Bin(add fg k v,(k',v'),fd)
    | Bin(fg,(k',v'),fd) ->
        Bin(fg,(k',v'),add fd k v)
```

Correction

`find(add(d,k,v),k) = v` :
Pour pouvoir continuer, il faut connaître la forme de `d`
On procède par induction sur `d`

- `d = Nil` : alors `add d k v = Bin(Nil,(k,v),Nil)`
  `find (Bin(Nil,(k,v),Nil)) k = v` CQFD

- `d = Bin(fg, k',fd)`
  Par hypothèse d'induction, `find(add(fg,k,v),k) = v`
  et `find(add(fd,k,v),k) = v`
  - si $k < k'$,
    `add d k v = Bin(add fg k v, k', fd)`
    `find (Bin(add fg k v, k', fd)) = find fg k v`
  - si $k > k'$, symétrique
  - $k \neq k'$ par hypothèse