

Arbres bien équilibrés

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

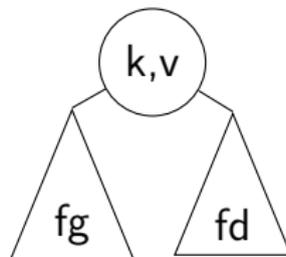
mais

- ▶ 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.



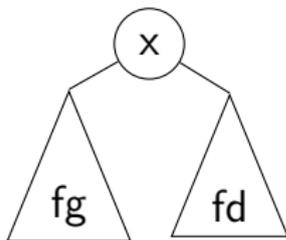
Invariant : pour un nœud

- ▶ 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

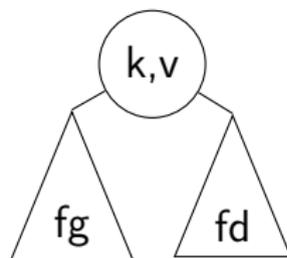
- ▶ P est vrai pour l'arbre vide



- ▶ P est vrai pour fg et fd 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

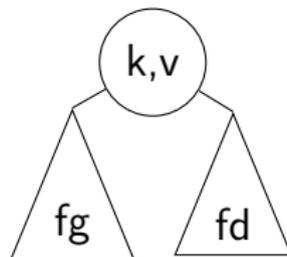
- ▶ si $k' = k$, retourner v
- ▶ si $k' < k$ et fg est non-vide, rechercher k' dans fg
- ▶ si $k' > k$ et fd est non-vide, rechercher k' dans fd

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')



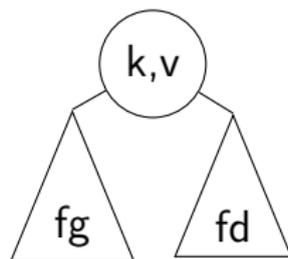
insérer k',v' dans

- ▶ 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

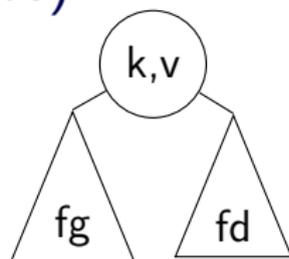
Suppression



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 fg fd avec $k \neq k'$:

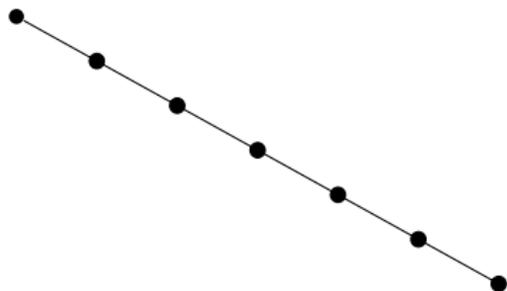
- ▶ si $k' < k$, supprimer k' dans fg
- ▶ si $k' > k$, supprimer k' dans fd

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 (OCaml)

```
type ('k,'v) dict =  
  Nil  
  | Bin of ('k,'v) dict * ('k * 'v) * ('k,'v) dict  
  
let creer _ = Nil  
  
let rec rechercher d k =  
  match d with  
  | Nil -> raise Not_found  
  | Bin(_, (k',v), _) when k = k' -> v  
  | Bin(fg,(k',_), _) when k < k' -> rechercher fg k  
  | Bin(_, (k',_),fd) -> rechercher fd k
```

Implémentation (OCaml, suite)

```
let rec inserer 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(inserer fg k v,(k',v'),fd)  
  | Bin(fg,(k',v'),fd) ->  
    Bin(fg,(k',v'),inserer fd k v)
```

Correction (OCaml)

`rechercher(inserer(d,k,v),k) = v :`

Pour pouvoir continuer, il faut connaître la forme de `d`

On procède par induction sur `d`

- ▶ `d = Nil` : alors `inserer d k v = Bin(Nil,(k,v),Nil)`
`rechercher (Bin(Nil,(k,v),Nil)) k = v` CQFD

► $d = \text{Bin}(fg, k', fd)$

Par hypothèse d'induction,

$\text{rechercher}(\text{insérer}(fg, k, v), k) = v$ et

$\text{rechercher}(\text{insérer}(fd, k, v), k) = v$

- si $k < k'$,
 $\text{insérer } d \text{ } k \text{ } v = \text{Bin}(\text{insérer } fg \text{ } k \text{ } v, k', fd)$
 $\text{rechercher}(\text{Bin}(\text{insérer } fg \text{ } k \text{ } v, k', fd)) =$
 $\text{rechercher } fg \text{ } k \text{ } v$
- si $k > k'$, symétrique
- $k \neq k'$ par hypothèse

Implémentation (C)

```
struct dict_base { key key;
                  value val;
                  dict fg;
                  dict fd; };

dict creer(int size) { return NULL; }

dict bin(dict fg, key k, value v, dict fd) {
    dict p = malloc(sizeof(dict_base));
    p->key = k;      p->val = v;
    p->fg = fg;     p->fd = fd;
    return p;
}
```

Implémentation (C, suite)

```
value rechercher(dict d, key k) {  
    while(d != NULL) {  
        if (d->key == k) return d->val;  
        if (d->key > k) d = d->fg;  
        else d = d->fd;    };  
    return NULL;  
}
```

Implémentation (C, version destructive)

```
dict inserer(dict d, key k, value v) {  
    if (!d) return bin(NULL,k,v,NULL);  
    if (d->key > k)  
        d->fg = inserer(d->fg, k, v);  
    if (d->key < k)  
        d->fd = inserer(d->fd, k, v);  
    return d;  
}
```

Implémentation (C, version non destructive)

```
dict inserer(dict d, key k, value v) {
    dict res;
    if (!d) res = bin(NULL,k,v,NULL);
    else {
        if (d->key > k)
            res = bin(inserer(d->fg, k, v),
                    d->key, d->val, d->fd);
        if (d->key < k)
            res = bin(d->fd, d->key, d->val,
                    inserer(d->fg, k, v));
    };
    return res;
}
```

Arbres bien équilibrés

Pour rendre les ABR efficaces, il faut minimiser la hauteur par rapport au nombre de nœuds

Définition

*Un arbre est **bien équilibré** si :*

- ▶ *la différence de hauteur entre ses deux sous-arbres est d'au plus 1*
- ▶ *chacun de ses sous-arbres est bien équilibré*

Hauteur d'un arbre bien équilibré

Théorème

Pour un arbre bien équilibré à n nœuds et de hauteur h

$$\log_2(1 + n) \leq h \leq \alpha \log_2(2 + n)$$

avec $\alpha = \frac{1}{\log_2\left(\frac{1+\sqrt{5}}{2}\right)} < 1,44$

Par conséquent, rechercher, insérer et supprimer sont en $O(\log n)$ dans les arbres bien équilibrés

Arbres auto-équilibrants

Problème : l'insertion et la suppression peuvent déséquilibrer un arbre bien équilibré

- ▶ Modifier les fonctions d'insertion et de suppression pour garantir un invariant de bon équilibre

Arbres AVL (Adelson-Velskii et Landis, 1962) : rééquilibrage par rotations

Encapsulation

On veut ne manipuler que des arbres bien équilibrés

- ▶ Il faut interdire la création d'arbres non équilibrés à l'extérieur du module

On utilise l'encapsulation liée à la modularité : la seule façon de construire des AVL sera via le constructeur intelligent `ba1`

Intérêt : dans `ba1` on peut supposer que les arguments sont bien équilibrés, puisqu'ils proviennent des appels aux constructeurs intelligents

Types privés en OCaml

```
type t = private A of u | B of v
```

On ne peut utiliser les constructeurs A et B qu'à l'intérieur du module

Toujours possible de filtrer A et B en dehors du module

Intérêt : on définit des constructeurs intelligents

```
let a (x : u) = let x' = ... in A x'
```

```
let b (y : v) = let y' = ... in B y'
```

En dehors du module, obligé de passer par les constructeurs intelligents pour faire un t

Toutefois, on peut toujours discriminer les objets de type t

Complexité

Après une insertion, une seule rotation (éventuellement double) est nécessaire

Après une suppression, la rotation à effectuer peut entraîner un déséquilibre au-dessus, il y a donc au plus h rotations à effectuer

Comme les rotations sont effectuées en temps constant, les opérations pour rééquilibrer l'arbre sont en $O(\log n)$

Par conséquent, dans un arbre AVL, la recherche, l'insertion et la suppression sont en $O(\log n)$, y compris dans le pire des cas