LU2IN006

Cours 6 - SDA Arbre Binaire de Recherche

"Structures de données"

Pierre-Henri Wuillemin
2025fev
pierre-henri.wuillemin@lip6.fr

1. Arbre binaire de recherche

- Tous les algorithmes de recherche actuellement vus sur les arbres sont des parcours complets sur l'arbre : $O(n)$ où n est le nombre d'éléments de l'arbre.

  PS : c'est parfois obligatoire : copie d'un arbre !
- Pour des algo. en $O(h)$ où h hauteur, il faudrait choisir une appel récursif vers `fg` ou `fd` mais pas vers les 2 !
- Quels algorithmes de recherche pourraient être en $O(h)$? En particulier, comment faire une fonction `exists` en $O(h)$?

Question : Comment rechercher facilement dans un ensemble de données totalement ordonnées ?

- Trouver un élément rapidement,
- Trouver le min/max rapidement,
- Insérer/supprimer un élément rapidement.

Définition

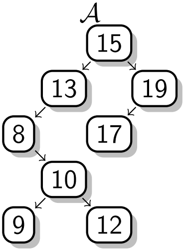
Question : Comment rechercher facilement dans un ensemble de données totalement ordonnées ?

Dans ce qui suit, on utilisera $c(x)$ pour indiquer la clé du nœud x.

Arbre Binaire de Recherche (ABR)

Un Arbre Binaire de Recherche est :

- un arbre binaire A
- $\forall x \in A, \forall y \in A_g(x), \forall z \in A_d(x), c(y) < c(x) < c(z)$



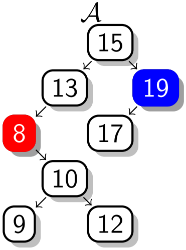
Le parcours **infixe** d'un ABR donne la liste triée des clés.

`infixe` sur $\mathcal{A}$: [8 9 10 12 13 15 17 19]

Recherche du plus petit/grand élément d'un ABR

Un ABR est un arbre binaire : la structure de données C `btree` est suffisante. Il s'agit de s'assurer que la logique des algorithmes permettra d'assurer la cohérence de l'ABR.

```c
typedef struct s_btree {
  int cle;

  struct s_btree* fd;
  struct s_btree* fg;
} btree;
```



Minimum et Maximum dans un ABR

Dans un ABR,

- Le plus petit élément se trouve dans le nœud le plus à gauche.
- Le plus grand élément se trouve dans le nœud le plus à droite.

Minimum et maximum dans un ABR

Trouver le minimum d'un ABR : complexité en $O(h)$

```
int minABR(btree* abr) {
  if (abr==NULL)
    return -1;

  if (abr->fg==NULL)
    return abr->cle;

  return minABR(abr->fg);
}
```

Trouver le maximum d'un ABR : complexité en $O(h)$

```
int maxABR(btree* abr) {
  if (abr==NULL)
    return -1;

  if (abr->fd==NULL)
    return abr->cle;

  return maxABR(abr->fd);
}
```

Vérifier qu'un arbre binaire est un ABR

Propriété récursive des ABR

Un ABR est un arbre binaire $A = (r, A_g, A_d)$ qui vérifie :

- A_g et A_d sont des ABR,
- $\max A_g < c(r) < \min A_d$

```c
int checkABR(btree *b) {
  if (b==NULL) return 1;

  if (b->fg!=NULL) {
    if (checkABR(b->fg)==0) return 0;
    if (maxABR(b->fg)>=b->cle) return 0;
  }

  if (b->fd!=NULL) {
    if (checkABR(b->fd)==0) return 0;
    if (minABR(b->fd)<=b->cle) return 0;
  }

  return 1;
}
```

Rechercher un élément dans un arbre (rappel)

Rechercher un élément dans un arbre implique de devoir chercher dans l'arbre entier : Recherche en $O(n)$

Recherche version arbre

```c
btree* exists(btree* t, int val) {
  if (t!=NULL) {
    if (t->cle==val)
      return t;
    else {
      btree* tmp;

      tmp=exists(t->fg,val);
      if (tmp!=NULL)
        return tmp;

      tmp=exists(t->fd,val);
      if (tmp!=NULL)
        return tmp;
    }
  }

  return NULL;
}
```

Recherche un élément dans un ABR

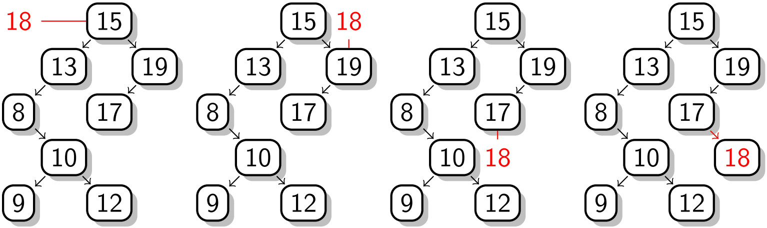
La structure d'un ABR permet de s'assurer à tout moment dans quel sous-arbre il faut chercher plutôt que devoir chercher dans l'arbre entier : Recherche en $O(h)$

Recherche version ABR

```
btree* existsABR(btree* t,int val) {
  if (t==NULL) return NULL;

  if (val==t->cle) return t;

  if (val<t->cle)
    return existsABR(t->fg,val);
  else
    return existsABR(t->fd,val);
}
```

Insertion d'un élément dans un ABR

But : insérer un nouvel élément (supposé non existant dans l'ABR)



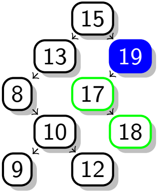
Insérer un élément dans un ABR consiste à l'insérer dans le sous-ABR adéquat : $O(h)$.

La forme de l'ABR dépend de l'ordre d'insertion ainsi que des données saisies !

Insertion d'un élément dans un ABR

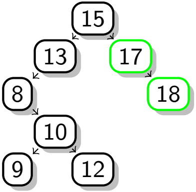

```c
btree* insererABR(btree* b,int val) {
  if (b==NULL) {            /* premier element de l'ABR*/
    return cree(val,NULL,NULL);
  }

  if (val<b->cle) {     /* dans Ag */
    if (b->fg==NULL) {     /* nouveau fils gauche */
      btree* nv=cree(val,NULL,NULL);
      b->fg=nv;
    } else {
      insererABR(b->fg,val);
    }
  } else {                  /* dans Ad */
    if (b->fd==NULL) {     /* nouveau fils droit */
        btree* nv=cree(val,NULL,NULL);
        b->fd=nv;
    } else {
        insererABR(b->fd,val);
    }
  }

  return b;
}
```

Suppression d’un élément (1)



Sous-problème :
Suppression du **max** d’un ABR

- la fonction va modifier l’ABR,
- la fonction va rendre la racine du nouvel arbre,
- la fonction va calculer le max dans un argument,
- propriété utilisée :
  Le max n’a pas de sous-arbre droit




```c
btree* supprimeMaxABR(btree* abr,int *pMax) {
  if (abr->fd==NULL) {
    *pMax=abr->cle;       /* sauvegarde du max */
    btree* resultat=abr->fg;/* sauvegarde du nouveau */
    free(abr);              /* suppression de l'ancien */
    return resultat;
  } else {
    /* suppression dans le fils droit */
    abr->fd=supprimeMaxABR(abr->fd,pMax);
    return abr;
  }
}
```

Suppression d'un élément (2)

Idées de l'algorithme :

- Si l'élément n'est pas la racine de l'arbre, il suffit de le supprimer dans le bon fils (gauche ou droit) et de recréer l'arbre complet à partir de ce nouveau fils.
- Si l'élément est la racine, alors le bon candidat à son remplacement est soit son fils s'il n'en a qu'un, soit le max de son fils gauche (ou le min de son fils droit)

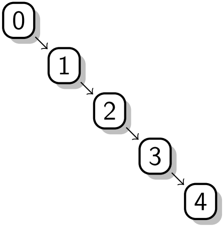
```
    btree* supprimeABR(btree* abr,int value) {
      if (abr->cle<value) {
        /* suppression dans le fils gauche */
        abr->fg=supprimeABR(abr->fg,value);

      } else if (abr->cle>value) {
        /* suppresion dans le fils droit */
        abr->fd=supprimeABR(abr->fd,value);

      } else {
        /* suppression de la racine */
      }
  }
```

Suppression d'un élément (3)

```
btree* supprimeABR(btree* abr,int value) {
  /* nouvelle racine (si besoin) */
  btree *resultat=abr;

  if (abr->cle>value) {
    /* suppression dans le fils gauche */
    abr->fg=supprimeABR(abr->fg,value);
  } else if (abr->cle<value) {
    /* suppression dans le fils droit  */
    abr->fd=supprimeABR(abr->fd,value);
  } else {
    if (abr->fg==NULL) {
      /* nouvelle racine : fils droit   */
      resultat=abr->fd;
      free(abr);
    } else { /* supprimer max de fils gauche   */
      int max;
      abr->fg=supprimeMaxABR(abr->fg,&max);
      abr->cle=max;
    }
  }

  return resultat;
}
```

Arrgh

```c
btree *abr=NULL;

int i;
for(i=0;i<5;i++)
  abr=insererABR(abr,i);
```



Inconvénients des ABR

On crée facilement des ABR dont la hauteur h est égale au nombre d'éléments n. Les algorithmes dans de tels ABR seront donc en $O(n)$

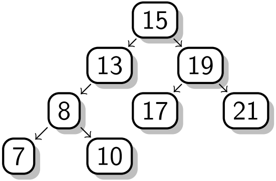
Intérêt des ABR

- Les Arbres Binaires de Recherche sont une structure de données qui permet de représenter correctement un ensemble ordonné de clés.
- Les alternatives (tableaux ou listes) ont de bien moins bons comportements dans les opérations usuelles : ajout / suppression / recherche.
- Pour tous les algorithmes des ABRs, le temps de calcul est proportionnel à la hauteur de l'arbre (et non au nombre de clés comme pour les autres alternatives)
- **Toutefois**, dans le pire des cas, la hauteur de l'arbre est le nombre de clés dans l'arbre.
- En moyenne, les ABRs sont quand même souvent intéressants.
- Pour éviter les mauvais cas : **équilibrage des arbres**.

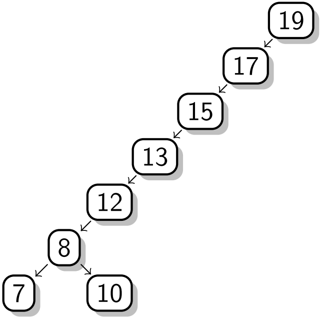
Retour sur les ABR

Les Arbres Binaires de Recherche permettent d'opérer des opérations d'Insertion, Suppression, Recherche en un temps proportionnel à la hauteur de l'arbre ... **c'est mieux que proportionnel au nombre de nœuds !**

Problème des arbres mal équilibrées



Opérations en 4 étapes au plus



Opérations en 8 étapes

Pourquoi équilibrer ?

Si les procédures d'Insertion et Suppression permettaient de garder un arbre équilibré, la hauteur de l'arbre (et le temps de calcul) serait minimisée.

Comment équilibrer ? *Vœux pieux ...*

- $\forall n, |F_g(n)| = |F_d(n)|$: même nombre de nœuds à gauche et à droite.
- $\forall n, h(F_g(n)) = h(F_d(n))$: même hauteur des sous-arbres gauche et droit.

Les propriétés précédentes sont difficiles à maintenir dans un ABR. Donc, une version relaxée :

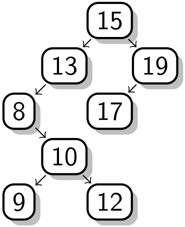
- $\forall n, |h(F_g(n)) - h(F_d(n))| < 2$: les 2 hauteurs sont les mêmes, à 1 près.

En passant : dichotomie et ABR

Une recherche par dichotomie revient à une recherche dans un ABR balancé "au mieux" suivant le premier critère d'équilibrage : même nombre de nœuds à gauche et à droite.

Arbre AVL [1962 - Adel'son-Vel'skii et Landiis]

Un arbre AVL vérifie la propriété : **la différence des hauteurs des fils gauche et droit de tout nœud ne peut excéder** 1**.**



- Cet arbre n'est pas AVL : 15, 13 et 8 violent la propriété.
- Un arbre binaire complet est AVL.
- **Pour maintenir un arbre AVL, il faudra garder en tout nœud cette différence de hauteur.**

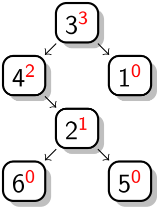
Propriété des arbres AVL

La hauteur d'un arbre AVL est de l'ordre de **ln n**.

Implémentation d'un arbre AVL en C

Implémentation : chaque nœud doit connaître sa hauteur

```c
typedef struct s_avltree {
  int cle;
  int hauteur; /* hauteur de l'arbre */

  struct s_avltree* fd;
  struct s_avltree* fg;
} AVL;
```

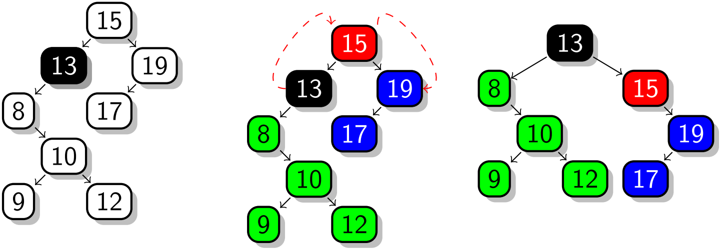


Création d'un nœud AVL

```c
AVL* cree(int val, AVL* fd,AVL* fg) {
  AVL* n=(AVL *)malloc(sizeof(AVL));

  n->cle=val;
  n->fg=fg;
  n->fd=fd;

  n->hauteur=1+max(
       fg==NULL?-1:fg->hauteur,
       fd==NULL?-1:fd->hauteur
  );

  return n;
}
```

Comment rééquilibrer un ABR ?

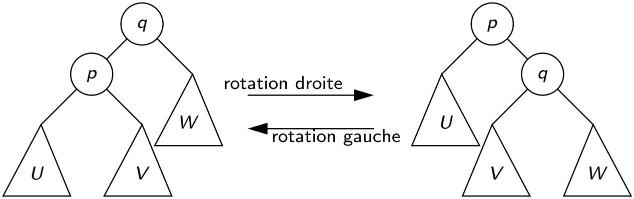
Rééquilibrage de 13



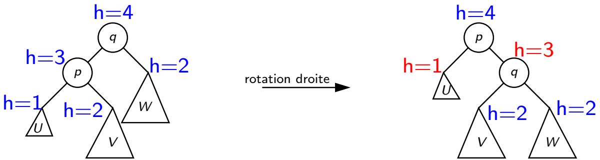
Cette opération s'appelle une rotation.

Rééquilibrage de l'arbre : rotations

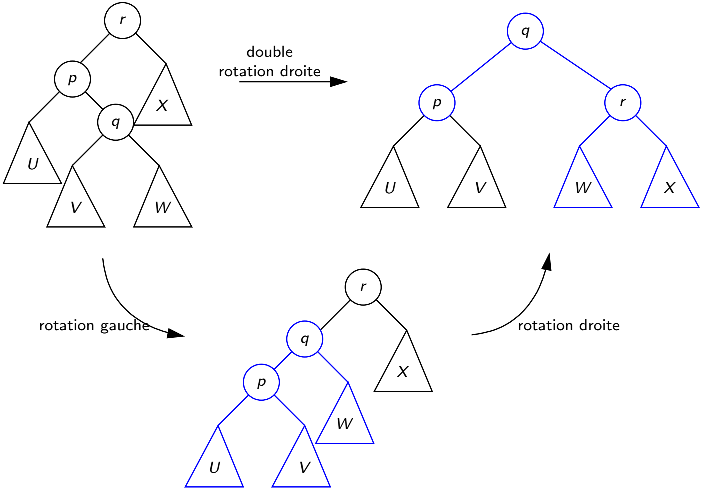
Les transformations suivantes seront à utiliser :



- Un ABR est transformé en ABR par rotation. Les rotations conservent l'ordre infixe.
- **La propriété AVL n'est pas conservée par une rotation.**



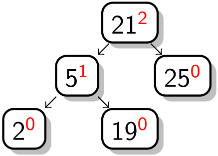
Rééquilibrage de l'arbre (2) : rotations doubles



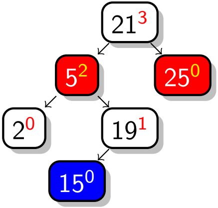
La double rotation gauche est définie de la même façon.

Rééquilibrage de l'arbre (3) : À quoi ça sert ?

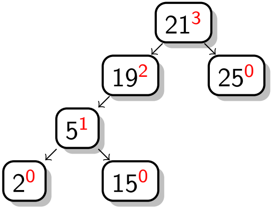
(1) Soit l'ABR AVL suivant :



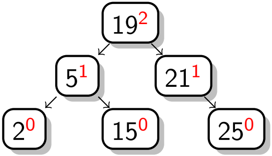
(2) Insertion de 15 :
ABR non AVL



(3) Rotation gauche 19-5 :
ABR non AVL

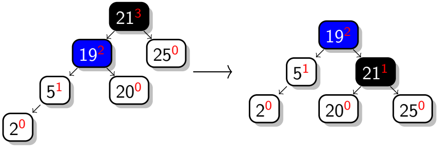


(4) Rotation droite 19-21 :
ABR AVL



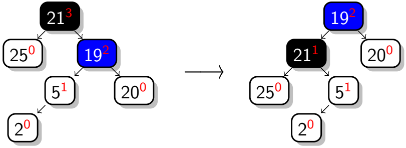
Implémentation des rotations

rotDroite(21)




```c
AVL* rotDroite(AVL* rac) {
  AVL* nvrac=rac->fg;

  rac->fg=nvrac->fd;
  nvrac->fd=rac;

  majHauteur(rac);
  majHauteur(nvrac);

  return nvrac;
}
```

rotGauche(21)

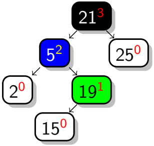



```c
AVL* rotGauche(AVL* rac){
  AVL* nvrac=rac->fd;

  rac->fd=nvrac->fg;
  nvrac->fg=rac;

  majHauteur(rac);
  majHauteur(nvrac);

  return nvrac;
}
```

Les rotations sont en $O(1)$!

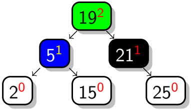
Implémentation des doubles rotations

doubleRotDroite(21)



❶ RotGauche(5)
❷ RotDroite(21)

⟶




```c
AVL* doubleRotDroite(AVL* rac) {
  rac->fg=rotGauche(rac->fg);
  /* le fils gauche a chang'e */
  majHauteur(rac);

  return rotDroite(rac);
}
```

Les doubles rotations sont en $O(1)$!

Insertion/Suppression dans un ABR AVL

Algorithme `bienEquilibrer`

Après une insertion/suppression classique, le long du chemin vers la racine, on vérifie :

- Soit un arbre A,
- G et D ses sous-arbres gauche et droit,
- g et d, les sous-arbres gauche et droit de G.

```
Si H(G)-H(D)=2 Alors
  Si H(g)<H(d) alors rotation gauche de G
  rotation droite de A
FinSi

Si H(G)-H(D)=-2 ... (symétrique dans le fils droit)
```

Insertion d'un élément dans un ABR


```
AVL* insererABR(AVL* b,val v) {
  if (b==NULL) {              /* premier element de l'ABR*/
    return cree(val,NULL,NULL);
  }
  if (val<b->cle) {     /* dans Ag */
    if (b->fg==NULL) {     /* nouveau fils gauche */
      AVL* nv=cree(val,NULL,NULL);
      b->fg=nv;
    } else {
      b->fg=insererABR(b->fg,val);
    }
  } else {                  /* dans Ad */
    if (b->fd==NULL) {     /* nouveau fils droit */
      AVL* nv=cree(val,NULL,NULL);
      b->fd=nv;
    } else {
      b->fd=insererABR(b->fd,val);
    }
  }

  majHauteur(b);
  b=bienEquilibrer(b);
  return b;
}
```