Cours d'Algorithmique et Complexité

Structures de données (2e suite)

Catalin Dima

Arbres binaires de recherche

Propriété de base des arbres binaires de recherche

Soit x un noeud de l'arbre. Alors :

1. pour tout y noeud dans le sous-arbre gauche de x, `x.clé` $\geq$ `y.clé`
2. pour tout y noeud dans le sous-arbre droite de x, `x.clé` $\leq$ `y.clé`

- Structure de données implémentant les opérations `rechercher, min, max, prédécesseur, successeur, insérer, supprimer`.
- Complexité (moyenne) des opérations $\Theta(\log n) = \Theta(h)$ où h est la hauteur de l'arbre.
- Mais complexité pire cas $\Theta(n)$...

Parcours en infix

Proposition

Si `t` est un arbre binaire de recherche, alors le parcours en infix avec affichage produit les clés en ordre croissant.

Théorème

L'algorithme de parcours en infix d'un arbre binaire à n noeuds prend un temps $\Theta(n)$.

Recherche d'un noeud possédant une clé

- On suppose que les noeuds sont de la classe `Noeud` possédant les champs `cle, droit, gauche` et `parent`.

```
class ArbreBinRech{
Noeud racine;
Noeud recherche(int cle){
   Noeud x=racine;
   while((x!=null) && (x.cle != cle)){
      if (x.cle > x) x=x.gauche;
         else x=x.droite;
   }
   return x;
}
```

(Variante récursive possible – mais alors méthode statique !)

Minimum

Propriété

Le noeud de clé minimale dans un arbre binaire de recherche est celui qui est “le plus à gauche” = dernier fils gauche du fils gauche du .... du fils gauche de la racine.

Variante qui cherche le noeud min dans un *sous-arbre* :

```
Noeud min(Noeud x){
   while(x.gauche!=null)
      x=x.gauche;
   return x;
}
```

Propriété

Le noeud de clé maximale dans un arbre binaire de recherche est celui qui est “le plus à droite” = dernier fils droit du fils droit du .... du fils droit de la racine.

Successeur

Propriété

Le successeur d'un noeud x dans un arbre binaire de recherche est soit son descendant le plus à gauche de son sous-arbre droit, s'il possède un sous-arbre droit, soit le plus proche ancêtre de x pour qui x se trouve dans le sous-arbre gauche.

- Noter qu'on a même pas besoin d'utiliser des comparaisons des clés !
- Évidemment, il n'y a pas de successeur du noeud qui se trouve "le plus à droite" dans l'arbre !

```
Noeud successeur(Noeud x){
   if(x.droite != null) return min(x);
   while(x.parent!=null)
      x=x.parent;
   return x;
}
```

Insertion

- Supposons que l'objet à insérer x (avec sa clé) est bien créé.
- La découverte de l'endroit où il faut insérer x dans un arbre se base sur la recherche d'une clé dans l'arbre : il faut trouver la "feuille" où x devrait être placé.

```
void inserer(Noeud x){
   Noeud y=racine;
   Noeud z=null;
   while(y!=null){
      z=y;
      if (x.cle>y.cle)      // c'est ici que ça ressemble
            y=y.gauche;     // à la recherche d'une clé !
         else y=y.droite;
   }
   x.parent=z;
   if(z==null)       // l'arbre était vide !
       racine=x;
      else if(x.cle<z.cle)
                  z.gauche=x;
        else z.droite=x;
}
```

Suppression

Pour supprimer un noeud x (après l'avoir identifié) :

- Si x n'a pas d'enfants : défaire les liens vers son père et le supprimer (ou laisser le *Garbage Collector* se charger de libérer l'espace).
- Si x a un seul fils : ce fils prendra la place de x.
- Sinon : on trouve le successeur y de x et c'est lui qui prendra la place de x.
  - Remarquer que, dans ce cas, x a un successeur !
  - Remarquer aussi que y n'a pas de sous-arbre gauche.
  - Le sous-arbre gauche de x devient sous-arbre gauche de y.
  - Les liens entre y et son sous-arbre droit restent en place.
  - Enfin, x peut dégager !

Suppression

```
void transplanter(Noeud x, Noeud y){
   if(x==racine){
      racine=y;
      return;
   }
   if (x==x.parent.gauche)  x.parent.gauche=y;
      else x.parent.droite = y;
   }
   if(y!=null) y.parent=x.parent;
   x.parent=x.gauche=y.gauche=null;
          // et le GC pourra faire son travail !
}
void supprimer(Noeud x){
   if(x.gauche=null){
      transplanter(x,x.droite);
      return;
   }
   if(x.droite==null){
      transplanter(x,x.gauche);
      return;
   }
   Noeud y = min(x.droite);
   if(y.parent!=x){
      transplanter(y,y.droite)
      y.droite = x.droite;
      y.droite.parent = y;
   }
   transplanter(x,y);
   y.gauche = x.gauche;
   x.gauche.parent = y;
}
```

Désavantages des arbres de recherche binaire

- Les différentes opérations dans un arbre de recherche binaire s'exécutent en temps $\Theta(h) = \Theta(\log n)$ *en moyenne* !
- Mais si l'arbre binaire est très déséquilibré, alors le temps d'exécution devient $\Theta(n)$!
  - Lorsqu'on crée un arbre vide, puis on insère des noeuds dons les clés sont triées !
  - Exemple avec $10, 15, 41, 56, 89$.
- Déséquilibré : il y a des chemins très courts et d'autres très longs.
  - L'arbre obtenu pour toute suite d'insertions de clés triées est en fait "linéaire" !
  - Le chemin le plus court a la longueur 1.
  - Le plus long a la longueur n = le nombre de clés = le nombre de noeuds.
- Solution : chercher à créer toujours des arbres pour lesquels la différence entre le chemin le plus court et le chemin le plus long reste raisonnable.

Arbres rouge-noir

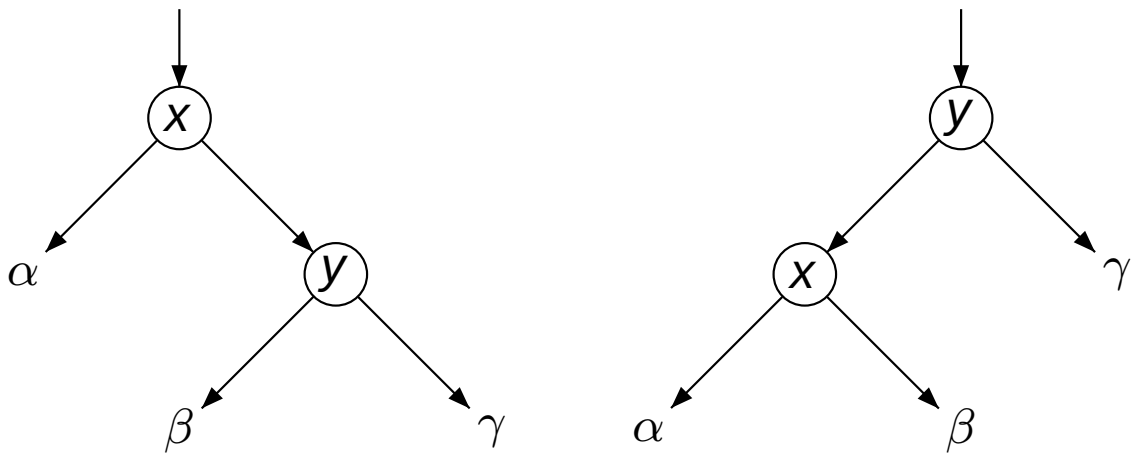
- C'est des arbres binaires de recherche d'un type particulier.
- Construits de telle sorte que *le chemin le plus long est moins de 2 fois plus long que le chemin le plus court.*
- Chaque noeud possède un bit supplémentaire (champs) : rouge/noir, avec les propriétés suivantes :
  - La racine est noire.
  - Les feuilles `null` sont noires.
  - Si un noeud est rouge alors ses deux enfants sont noirs.
  - Pour chaque noeud `x`, tous les chemins qui mènent à leurs feuilles contiennent le même nombre de noeuds noirs.
- Convention : les feuilles ne seront plus des références `null`, mais des noeuds contenant une clé "feuille" (façon sentinelles dans les listes chaînées).

Lemme

La hauteur d'un arbre rouge-noir ayant n noeuds (sans compter les feuilles-sentinelles) est au maximum $2\log(n+1)$.

Opérations sur les arbres rouge-noir

- Les opérations sur les arbres binaires de recherche sont aussi applicables aux arbres rouge-noir...
- ... sauf qu'elles peuvent détruire la propriété définissante !
- Il nous faut un algorithme de rééquilibrage.
- Opération de rotation :
  - Échange de deux noeuds et de leurs sous-arbres, dans le sens suivant :



- Remarquer que, à l'issue de cette rotation, l'arbre reste un arbre binaire de recherche !

Rotation

- On suppose que les arbres possèdent un 2e membre, `sentinelle`, qui est l'unique noeud représentant les feuilles (et donc remplace `null`!).

```
void rotation_gauche(noeud x){
   y = x.droite;
   x.droite = y.gauche;
   if (y.gauche != sentinelle) y.gauche.parent = x;
   y.parent = x.parent;
   if (x.parent == sentinelle){
         racine = y;
         return;
      }
   if (x==x.parent.gauche){
         x.parent.gauche = y;
         return;
      }
   x.parent.droite = y;
   y.gauche = x;
   x.parent = y;
}
```

Insertion dans un arbre rouge-noir

- Insertion comme dans un arbre de recherche normal.
- Affectation de couleur pour le noeud inséré : rouge.
- Rééquilibrage :
  - L'arbre peut avoir maintenant deux noeuds rouge adjacents (père-fils).
  - Il faut appliquer des rotations, combinées éventuellement avec des "ré-coloriages".
  - À l'issue d'une rotation qui n'aboutit pas a garantir la propriété rouge-noir, les paires de noeuds père-fils rouges sont plus proches de la racine.
- On décide du type de rotation en fonction de la couleur de l'oncle d'un noeud x.
- $oncle(x) = x.parent.parent.gauche$ ou $x.parent.parent.droite$.

Insertion dans un arbre rouge-noir

Situation 1 contredisant la propriété rouge-noir et opérations à appliquer

- x, $x.parent$ et $oncle(x)$ sont rouges.
- Et $x.parent.parent$, $frere(x)$, $oncle(x).gauche$ et $oncle(x).droite$ sont tous noirs.
- On va ré-colorier $x.parent$, $x.parent.parent$ et $oncle(x)$:
  - $x.parent$ et $oncle(x)$ deviennent noirs.
  - $x.parent.parent$ devient rouge.
- À l'issue de cela, soit l'arbre est rouge-noir, soit $x.parent.parent$ et $x.parent.parent.parent$ sont tous les deux rouges.

Et donc on peut reprendre avec $x = x.parent.parent$.

Insertion dans un arbre rouge-noir

Situation 2 contredisant la propriété rouge-noir et opérations à appliquer

- x et $x.parent$ sont rouges, $oncle(x)$ est noir.
- Et $x.parent.parent$, $frere(x)$ sont noirs,
- ... mais $oncle(x).gauche$ et $oncle(x).droite$ peuvent être soit rouges, soit noirs.
- En fonction du fait que x soit le fils gauche de son père, on applique une ou deux rotations :
- Si $x = x.parent.droite$ alors on applique une rotation pour x et $x.parent$ sans ré-coloriage.
- Puis une rotation pour $x.parent$ et $x.parent.parent$ et $x.parent$ et $x.parent.parent$ echangent leur couleur.
- Et l'arbre devient rouge-noir ! Exemple au tableau.

Algorithme d'insertion avec appel au rééquilibrage

```
void insererRN(Noeud x){
   Noeud y=racine;
   Noeud z=sentinelle;
   while(y!=sentinelle){
      z=y;
      if (x.cle>y.cle)
            y=y.gauche;
         else y=y.droite;
   }
   x.parent=z;
   if(z==sentinelle)       // l'arbre était vide !
       racine=x;
      else if(x.cle<z.cle)
                  z.gauche=x;
        else z.droite=x;
   x.gauche=x.droite=sentinelle;
   x.couleur=rouge;
   reequilibrage(x);
}
```

Algorithme de rééquilibrage

```
void reequilibrage(Noeud x){
   while(x.parent.couleur==rouge){
      if (x.parent== x.parent.parent.gauche){
         Noeud y=x.parent.parent.droite; //l'oncle !
         if (y.couleur==rouge){
            x.parent.couleur=rouge;
            y.couleur=noir;
            x.parent.parent.couleur=rouge;
            x=x.parent.parent;
         }
         else{
            if (x=x.parent.droite){
                x=x.parent;
                rotation_gauche(x);
            }
            x.parent.couleur=noir;
            x.parent.parent.couleur=rouge;
            rotation_droite(x.parent.parent);
          }
      }
      else // mêmes opérations mais en échangeant gauche et droite !
   }
   racine.couleur=noir;
}
```

Suppression dans un arbre rouge-noir

- D'abord on supprime le noeud comme dans un arbre de recherche normal.
- Rappel :
  - Soit x n'a pas de fils, donc on le supprime purement et simplement.
  - Lorsqu'on supprime x, on cherche le min(droite) ou, à défaut, le max(gauche) pour le remplacer.
  - Le min(droite) n'ayant pas de fils gauche, son père pointera desormais sur son fils droite.
  - Pareil pour le cas max(gauche).
  - Mettons y le noeud qu'on retrouve et qui remplacera x.
- Lorsque x est rouge et y est route aussi c'est pas grave : les propriétés rouge-noir restent satisfaites (à vérifier pour s'assurer !).
- Sinon on a un problème !
- Premier pas vers la solution :
  - x est noir et y est rouge : y (à sa nouvelle place) prendra la couleur noire et c'est ok.
  - y est noir – alors l'unique (!) fils de y aura son `compteur de noirs` incrémenté – et il peut devenir **deux fois noir** !
  - Et on cherche à appliquer des rotations avec re-coloriages qui résolvent ça.
- Il faut aussi faire attention si x était la racine : alors y, qui le remplace, doit devenir noir !

Résoudre le problème du deux-fois-noir

Un noeud t deux-fois-noir et bcp. de situations à traiter ; on suppose $t = t.pere.gauche$:

1. Le frère u de t est noir, $u.gauche$ et $u.droite$ sont noirs : on décrémente le compteur de noir de t, incrémente celui de $u.pere = t.pere$, u devient rouge et si le compteur de $u.pere$ est 2 on reprend avec $t \mapsto t.pere$.
2. u est noir, $u.pere = t.pere$ et $u.gauche$ sont noirs, $u.droite$ est rouge : rotation à gauche entre $t.pere$ et u, tous deviennent simplement noirs et c'est bon !
3. u est noir, pareil $u.pere = t.pere$, $u.gauche$ et $u.droite$ sont rouges : rotation à gauche entre $t.pere$ et u, tous deviennent simplement noirs sauf $u.gauche$ qui reste rouge (à sa nouvelle place) et c'est bon !
4. u est noir, $u.droite$ est noir, $u.gauche$ est rouge, peu importe la couleur de $t.pere = u.pere$: rotation droite entre u et $u.gauche$, u devient rouge, $u.gauche$ devient noir et on reprend sans changer t.
5. u est noir, $t.pere$ est rouge et $u.gauche$ est noir : rotaton gauche pour $t.pere$ et u, tous les noeuds gardent leur couleur sauf t qui devient simplement noir et c'est bon !
6. u est noir, $t.pere$ et $u.gauche$ sont rouge, $u.droit$ est noir : rotation droite entre u et $u.gauche$, u devient rouge et $u.gauche$ noir et on reprend sans changer t.
7. u est noir, $t.pere$ est rouge et $u.gauche$, $u.droite$ sont rouge : rotation gauche entre $t.pere$ et u, $u.droite$ et $t.pere$ deviennent noir, u devient rouge et t devient simplement noir et c'est bon !
8. u est rouge (alors $t.pere$, $u.droite$ et $u.gauche$ doivent être noirs !) : rotation gauche entre $t.pere$ et u, $t.pere$ devient rouge et u devient noir, tous les autres gardent leur couleur (y compris t qui reste deux-fois-noir) et on reprend sans changer t.

Et c'est tout !...