PC 6
Arbres binaires de recherche

- Définition
- Représentation
- Insertion
- Suppression
- Complexité
- Autres arbres

Comparaison des algorithmes de recherche

structure	recherche	insertion
tableau	$O(n)$	$O(1)$
liste	$O(n)$	$O(1)$
table ordonnée	$O(\log n)$	$O(n)$
arbres de recherche	$O(\log n)^*$	$O(\log n)^*$

$*$: en moyenne.

Les arbres binaires de recherche fournissent une structure de données *dynamique* de relativement basse complexité.

Arbres de recherche

Arbres binaires satisfaisant la propriété suivante: Si v est l'étiquette d'un noeud, on a $w < v$ pour tout noeud du sous-arbre gauche et $v < w$ pour tout noeud du sous-arbre droit.

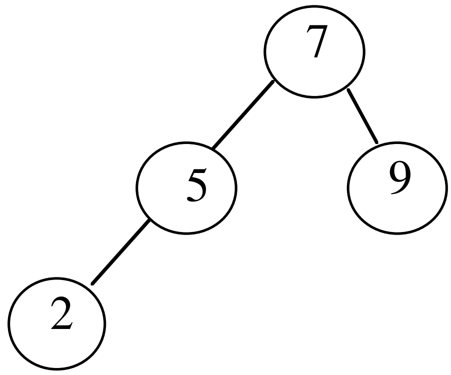

Figure 1:

Propriété caractéristique: la suite des étiquettes des noeuds dans l'ordre interne est croissante.

Opérations de base: recherche,insertion, suppression.

La recherche est une généralisation de la recherche dichotomique.

Représentation

On représente les arbres avec des pointeurs.

Déclaration de type en Pascal:

```
type
  Arbre = ^Noeud;
  Noeud = record
        contenu: Element;
        filsg: Arbre;
        filsd: Arbre;
        end;
```

Ou en C:

```
typedef struct Noeud{
    Element        contenu;
    struct Noeud   *filsG;
    struct Noeud   *filsD;
}  *Arbre;
```

Recherche d'un élément

La recherche est une généralisation de la recherche dichotomique. En Pascal:

```
function Recherche (v: Element; a: Arbre):Arbre;
 var r: Arbre;
 begin
  if a = nil then r := nil
  else if v = a^.contenu then
         r := a
  else if v < a^.contenu then
         r := Recherche (v, a^.filsg)
  else
         r := Recherche (v, a^.filsd);
  Recherche := r;
 end;
```

En C:

```
Arbre Recherche (Element v, Arbre a)
{
  if (a == NULL || v == a -> contenu)
       return a;
  else
    if (v < a -> contenu)
       return Recherche (v, a -> filsG);
    else
       return Recherche (v, a -> filsD);
}
```

Insertion

```
procedure Ajouter (v : Element; var a : Arbre);
  begin if a = nil then
           a := NouvelArbre(v, nil, nil)
        else if v <= a^.contenu then
           Ajouter (v, a^.filsG)
        else Ajouter (v, a^.filsD);
  end;
```

ou en C:

```
void Ajouter(Element v, Arbre *ap)
{
 Arbre a=*ap;
 if (a==NULL)
  a = NouvlArbre(v,NULL,NULL);
 else if (v <= a -> contenu)
  Ajouter (v, &a -> filsG);
 else
  Ajouter (v, &a -> filsD);
 *ap = a;
}
```

Suppression

La suppression d'un élément nécessite une précaution : il ne faut pas créer un "trou". On choisit de remplacer le noeud à supprimer par le plus grand des descendants de son fils gauche (ou par son fils droit s'il n'a pas de fils gauche).

La première fonction supprime le plus grand élément d'un arbre de recherche et rend sa valeur:

```
function max(var a: Arbre): Element;

begin
 if a^.filsd = nil then begin
  max := a^.contenu;
  a := a^.filsg
 end
 else
  max := max(a^.filsd)
end;
```

```
procedure Supprimer(v: Element, var a: Arbre);
begin
 if a <> nil then
  if v < a^.contenu then
   Supprimer(v, a^.filsg)
  else if v > a^.contenu then
   Supprimer(v, a^.filsd)
  else if (a^. filsg = nil) then
   a := a^.filsd
  else
   a^.contenu := max(a^.filsg)
end;
```

Complexité

Toutes les opérations sur les arbres binaires de recherche se réalisent le long d'un chemin de l'arbre.

Leur complexité est donc $O(h)$ où h est la hauteur de l'arbre.

La complexité moyenne est la hauteur moyenne $h(n)$ d'un noeud. Pour montrer que $h(n) = O(\log n)$, on procède en deux étapes.

Relation de récurrence

Supposons que l'arbre est obtenu par n insertions successives de n éléments arrivant dans un ordre aléatoire. On a en raisonnant sur le premier arrivé i

$$h(n) = \sum_{i=1}^{n} \frac{1}{n} h(n,i)$$

avec, en raisonnant sur chaque noeud

$$\begin{aligned} h(n,i) &= \frac{1}{n}[1 + (i-1)(h(i-1)+1) \\ &\quad + (n-i)(h(n-i)+1)] \end{aligned}$$

d'où

$$h(n) = 1 + \frac{2}{n^2} \sum_{i=1}^{n-1} i h(i)$$

Résolution

On va montrer que

$$h(n) = 2(1 + \frac{1}{n})H_n - 3$$

où $H_n = 1 + \frac{1}{2} + \ldots + \frac{1}{n}$ est le n-ième *nombre harmonique.* Comme $H_n = O(\log n)$, la conclusion suit.

On écrit

$$h(n) = 1 + \frac{2}{n^2} \sum_{i=1}^{n-2} ih(i) + \frac{2n-2}{n^2} h(n-1)$$

Et on soustrait $\frac{(n-1)^2}{n^2} h(n-1)$. On obtient

$$h(n) = \frac{n^2-1}{n^2} h(n-1) + \frac{2n-1}{n^2}$$

ou encore

$$\frac{n}{n+1}(h(n)+3) = \frac{n-1}{n}(h(n-1)+3) + \frac{2}{n}$$

Comme l'expression ci-dessus satisfait cette relation de récurrence et que $h(1) = 1$, le résultat suit.

Autres arbres

On peut obtenir une complexité logarithmique non pas en moyenne mais dans le pire des cas en utilisant des structures plus compliquées comme :

- Arbres AVL
- Arbres 2-3
- ...

Arbres AVL

Un arbre AVL ou arbre équilibré est un arbre binaire tel que pour tout noeud n, les hauteurs des sous-arbres gauche et droit en n diffèrent d'au plus 1.

On peut vérifier que la hauteur h d'un arbre AVL ayant N noeuds vérifie

$$N \geq F_{h+2} - 1$$

où F_h est la suite de Fibonacci définie par $F_0 = 0, F_1 = 1$ et $F_{h+1} = F_h + F_{h-1}$. Ainsi

$$h \leq 1.5 \log N$$

Insertion dans un arbre AVL

L'insertion dans un arbre AVL se fait en suivant le même principe que dans les arbres binaires de recherche mais en ajoutant une fonction de *réequilibrage*. Celle-ci est définie en utilisant des opérations de *rotation* sur les arbres binaires.

La *rotation droite* est définie par

$$rd(r, (s, R, S), T) = (s, R, (r, S, T))$$

La rotation gauche est l'opération inverse. Toutes deux s'effectuent en temps constant.