Algo 18 – Arbre Binaire de Recherche (ABR)

I – Caractéristiques d'un arbre binaire de recherche (ABR)

1.1 - Description

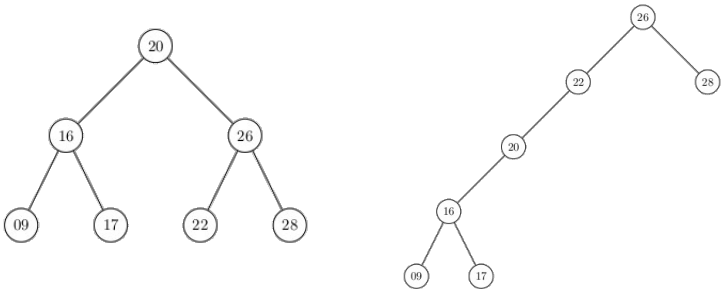
Un **arbre binaire de recherche** est un Arbre Binaire qui respecte une organisation particulière.
[1] Les **Clés des Noeuds doivent toutes pouvoir être comparées entre elles**.
[2] Les noeuds sont placés **récursivement** de telle façon que :

1. **Toutes les clés du sous-arbre gauche sont plus petites*** que la clé de la racine de l'Arbre étudié
2. **Toutes les clés du sous-arbre de droite sont plus grandes*** que la clé de la racine de l'Arbre étudié

*La présence du * indique que l'égalité peut être traitée comme vous le voulez.*

Récursivement indique que, quelque soit le nœud choisi dans l'ABR, les nœuds du sous-arbre gauche sont plus petits et les nœuds du sous-arbre droite sont plus grands.

Exemples : deux ABR différents portant les mêmes clés
A gauche, un Arbre Binaire de Recherche Parfait puis, à droite, les mêmes noeuds avec les mêmes clés dans une configuration moins optimale pour la recherche.

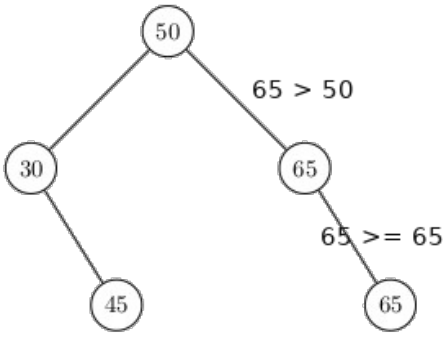


1.2 - Création progressive d'un ABR (version naïve : aucun contrôle sur la forme obtenue)

https://www.cs.usfca.edu/~galles/visualization/BST.html

La première valeur reçue (50 ici) devient la clé de la racine puis on positionne progressivement les autres éléments de cette façon :

1. Partir de la racine
2. Si la clé à placer est i**nférieure à celle du noeud actuel**, on part **à gauche**. Deux cas : si le sous-arbre gauche est vide, on lui attribue cette clé, sinon on répète l'opération récursivement en partant de la racine du sous-arbre gauche.
3. Si la clé à placer est **supérieure à celle du noeud actuel**, on part **à droite**. Deux cas : si le sous-arbre droite est vide, on lui attribue cette clé, sinon on répète l'opération récursivement en partant de la racine du sous-arbre droite.



Exemple (complet sur le site) :

On part de 50 et on rajoute 30. On rajoute ensuite 45. Puis 65. Puis éventuellement à nouveau 65.

1.3 Coût d'une recherche dans un Arbre binaire

La forme de l'ABR influe fortement sur le coût de la recherche. Il existe bien entendu des algorithmes d'insertion plus efficace qui cherche constamment à se rapprocher de la forme d'un arbre parfait, quitte à transformer en partie l'architecture de l'arbre.

Résumé à connaître par coeur

Recherche **non ciblée** est **linéaire en n** : $\mathcal{O}$**(n)**
Recherche **ciblée** est **linéaire en h** (et h dépend de la forme de l'arbre) :

- Sur un arbre binaire **parfait** : $\mathcal{O}$(h) donc $\mathcal{O}(\log_2(n))$
- Sur un arbre binaire **filiforme** : $\mathcal{O}$(h) donc $\mathcal{O}$**(n)**
- Sur un arbre binaire **quelconque** : $\mathcal{O}$(h) donc $\mathcal{O}$(n)

II – Algorithme d'insertion

2.1 Description de l'algorithme

On utilisera les 7 primitives habituelles :

nvAB() nvABV() estABV() racine() contenu() gauche() droite()

On rajoute une nouvelle primitive capable de « remplir » un AB VIDE pour en faire un NON VIDE :

initialiser(arbre:AB VIDE, cle:"Elément") -> None

Transforme un AB VIDE en AB NON VIDE en lui attribuant une clé et deux sous-arbres vides.

Algorithme d'insertion d'une clé dans un ABR

ENTREE 1 : l'ABR arbre avec lequel on veut travailler
ENTREE 2 : la clé qu'on veut rajouter à notre arbre.
SORTIE : Vide, on modifie ici l'arbre sur place

```
inserer_cle_dans_ABR(arbre:ABR, cle:Elément) -> Vide

    # étape 1 : on explore l'arbre jusqu'à trouver un sous-arbre de destination vide
    TANT QUE NON estABV(arbre)
        SI cle < contenu(racine(arbre))
            arbre ← gauche(arbre)       # on part à gauche
        SINON
            arbre ← droite(arbre)       # on part à droite
        Fin SI
    Fin Tant Que

    # étape 2 : on transforme notre arbre qui est un ABR VIDE en ABR NON VIDE
    initialiser(arbre, cle)

    Renvoyer VIDE
```

2.2 Implémentation avec l'AB utilisant notre modèle objet

Remarquez bien que :
- **initialiser()** est une réelle primitive : on manipule directement le modèle de l'AB.
- **inserer_cle_dans_ABR()** est une fonction construite à l'aide des 8 primitives.

```
def initialiser(arbre:'AB VIDE', cle:'Elément') -> None:
    """Transforme l'AB VIDE en AB NON VIDE en lui attribuant une clé et 2 sous-arbres vides"""

    arbre.v = cle
    arbre.g = nv_ABV()
    arbre.d = nv_ABV()


def inserer_cle_dans_ABR(arbre:'ABR', cle:'Elément') -> None:
    """Place la clé au bon endroit dans l'ABR"""

    while not est_ABV(arbre):
        if cle < contenu(racine(arbre)):
            arbre = gauche(arbre)
        else:
            arbre = droite(arbre)
    initialiser(arbre, cle)
```

2.3 Coût d'une insertion dans un ABR

L'analyse de la fonction précédente montre que c'est exactement comme le coût d'une recherche.

III – Algorithme de recherche

3.1 Description de l'algorithme

Algorithme de recherche dans un ABR : version itérative

```
      ENTREE 1 : un Arbre Binaire de Recherche arbre
      ENTREE 2 : une Clé cr
      SORTIE : un ABR NON VIDE portant cette clé ou un ABR VIDE si échec

recherche(arbre:ABR, cr:Cle) -> ABR
      TANT QUE NON estABV(arbre)
          SI cr == contenu(racine(arbre))
              Renvoyer racine(arbre)
          SINON SI cr < cle(racine(arbre))
              arbre ← gauche(arbre)
          SINON SI cr > cle(racine(arbre))
              arbre ← droite(arbre)
          Fin SI
      Fin Tant Que
      Renvoyer arbre   # C'est un ABR vide si on arrive ici
```

3.2 Implémentation avec l'AB utilisant notre modèle objet

```
  def recherche(arbre:'ABR', cr:'Elément') -> 'ABR':
    """Renvoie l'ABR portant la clé comme racine, ABR VIDE si échec de la recherche"""

    while not est_ABV(arbre):
        if cr == contenu(racine(arbre)):
            return racine(arbre)
        elif cr < contenu(racine(arbre)):
            arbre = gauche(arbre)
        elif cr > contenu(racine(arbre)):
            arbre = droite(arbre)
    return arbre  # si on arrive ici, c'est l'arbre est VIDE
```

Version ITERATIVE UNE SORTIE

Version RECURSIVE

3.3 Coût d'une recherche dans un ABR

Voir 1.3

REMARQUE : Le cas parfait est assez rare. On a donc cherché à voir dans quel cas, l'algorithme contenue à se comporter en terme de coût plutôt comme un arbre parfait que comme un arbre filiforme. Cela permet d'appliquer les coûts théoriques théoriques du parfait même s'il n'est pas réellement un arbre parfait : c'est la notion d'**arbre équilibré**.

ABR parfait et et ABR équilibré

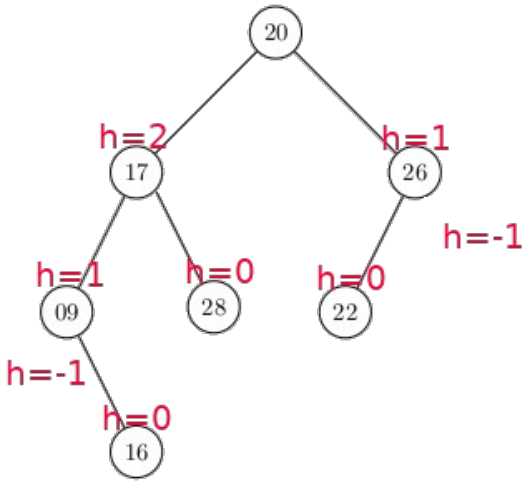
Vous savez qu'un ABR est parfait si le dernier étage est composé uniquement des feuilles de l'arbre.

La notion d'arbre équilibré possède plusieurs variantes.

L'important est de comprendre qu'il s'agit d'un arbre pour lequel les recherches peuvent toutes se ramener à peu près au cas de l'arbre parfait.

C'est le "à peu près" qui fera la différence entre les définitions. En 1962 que deux Russes, **Adel'son-Vel'skii et Landis**, introduisent des critères définissant un équilibre. Les arbres vérifiant ces critères sont connus sous le nom d'**arbres AVL**.

Voici le critère qu'ils ont définis pour savoir si un arbre est équilibré : **pour tout nœud de l'arbre, la différence entre les hauteurs de sous-arbres ne peut excéder 1.**



Sur l'exemple, on voit bien que tous les arbres présents dans notre arbre respectent le critère AVL : on peut donc supposer qu'une recherche sur cet arbre se fait à coût logarithmique.