IMAC

TP Algorithmique 3

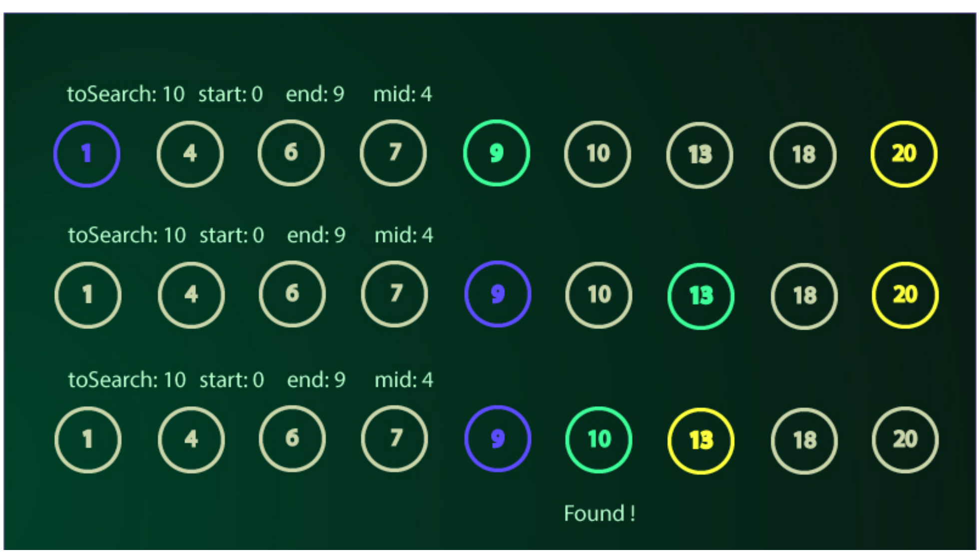
Recherche dichotomique et Arbre binaire

Les algorithmes les plus efficaces sont souvent en $O(log(n))$ ou en $O(nlog(n))$. Vous allez implémenter durant ce TP un algorithme de recherche en $O(log(n))$ et une structure de données qui facilite la recherche.

1 Recherche dichotomique

1.1 Principe

Cette algorithme fonctionne avec un tableau de nombre **triés**, lorsqu'on compare un nombre avec l'un de ceux du tableau, on peut déterminer dans quelle partie du tableau le nombre peut potentiellement se trouver. Si notre nombre est inférieur à celui comparé, il risque de se trouver dans la première partie du tableau, dans le cas contraire il devrait se trouver dans la seconde partie du tableau.
En partant de ce principe là, on peut cibler nos recherche. En particulier, en regardant la valeur du milieu de tableau, on divise le tableau en deux et en réitérant ce processus avec, à chaque fois, la partie qui contient potentiellement le nombre, on réduit le nombre de recherches total. Le temps que nous allons mettre avant de savoir si le nombre que l'on cherche est dans le tableau ou pas revient à se demander combien de fois nous pouvons diviser le tableau en 2, ou autrement-dit, combien de fois devons-nous multiplier 2 pour obtenir la taille du tableau $\Rightarrow 2^x = n \Leftrightarrow x = log_2(n)$



1.2 Algorithme

Algorithm 1 Binary Search

$t \Leftarrow$ tableau de n nombre aléatoire **triés**
toSearch $\Leftarrow$ nombre à chercher
start $\Leftarrow 0$
end $\Leftarrow n$
Tant que *start* $<$ *end* **faire**
    mid $\Leftarrow \frac{start+end}{2}$
    Si *toSearch* $> t[mid]$ **Alors**
        start $\Leftarrow mid + 1$
    Sinon Si *toSearch* $< t[mid]$ **Alors**
        end $\Leftarrow$ *mid*
    Sinon
        foundIndex $\Leftarrow$ *mid*
        Arrêt
    fin Si
fin Tant que

2 Arbre Binaire de Recherche (ABR)

Les arbres binaires de recherches sont des structures de données qui reposent sur cette même idée de diviser pour réduire la recherche de données. Un arbre est un ensemble de nœuds qui sont :

— Soit des **branches** : Des nœuds qui possèdent d'autres nœuds, leurs enfants. Dans le cas d'un arbre binaire, une branche possède au maximum deux enfants (gauche et droite).

— Soit des **feuilles** : Des nœuds sans enfants.

Une branche qui n'est l'enfant d'aucune autre branche est une **racine**.
L'arbre binaire de recherche est organisée de telle sorte que chaque nœud soit plus grand que son enfant de gauche et plus petit que son enfant de droite. Cette organisation permet de faciliter la recherche

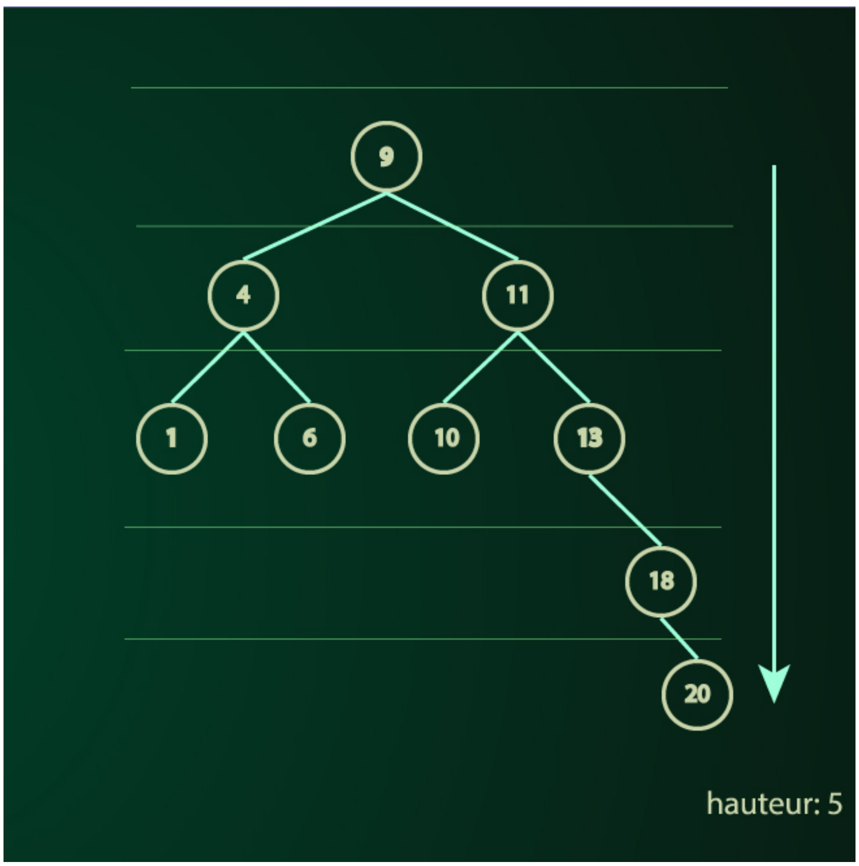


FIGURE 1 – Exemple d'arbre binaire de recherche

d'un élément, chaque nœud qui ne correspond pas à l'élément recherché délègue la recherche à son gauche ou droite, réduisant à chaque fois la recherche de moitié.

Propriété : La hauteur h d'un arbre est égale au nombre de nœuds se trouvant dans le chemin le plus long pour arriver à une feuille.

2.1 Arbre AVL - Arbre de recherche Adelson-Velsky Landis

Le problème de l'exemple donné ci-dessus est la recherche du nombre 20. On arrive au nœud 20 après 5 itérations au lieu de 3. Ce problème est dû au fait que l'arbre n'est pas correctement équilibré. Un arbre a est **équilibré** s'il répond à la propriété suivante :

$$\forall noeud \in arbre, -1 \leq f(noeud) \leq 1$$

$$f(n) = h(droit) - h(gauche)$$

Autrement-dit, aucun des nœuds ne doit être plus profond de deux nœuds ou plus que son frère.
l'arbre **AVL** ou arbre de recherche automatiquement équilibré est un arbre de recherche qui s'équilibre à chaque insertion d'une valeur. Cela permet de garantir une complexité d'insertion et de recherche en $o(log(n))$

2.1.1 Équilibrage/rotation d'un arbre

Après avoir inséré (ou supprimé) un nœud dans l'arbre si on se rend compte que celui-ce devient déséquilibré, il faut procédé à une rotation de l'arbre pour le rééquilibrer.
Voici les différentes rotations possibles :

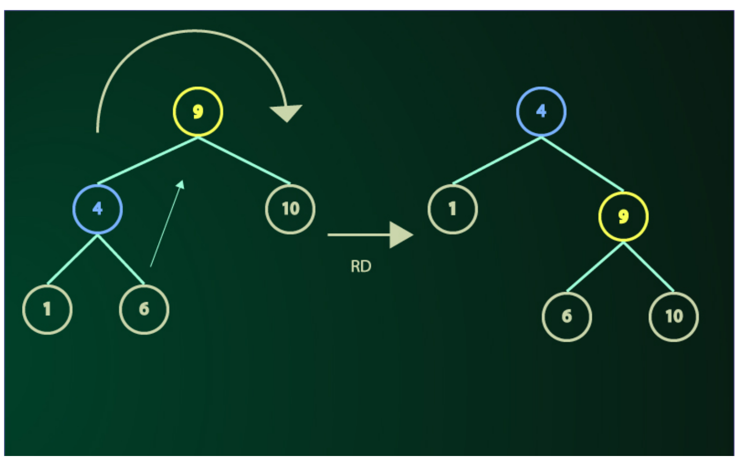


FIGURE 2 – Rotation Droite

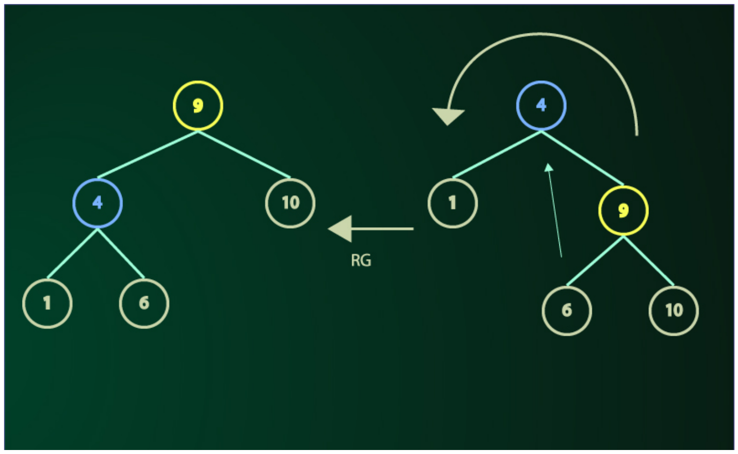


FIGURE 3 – Rotation Gauche

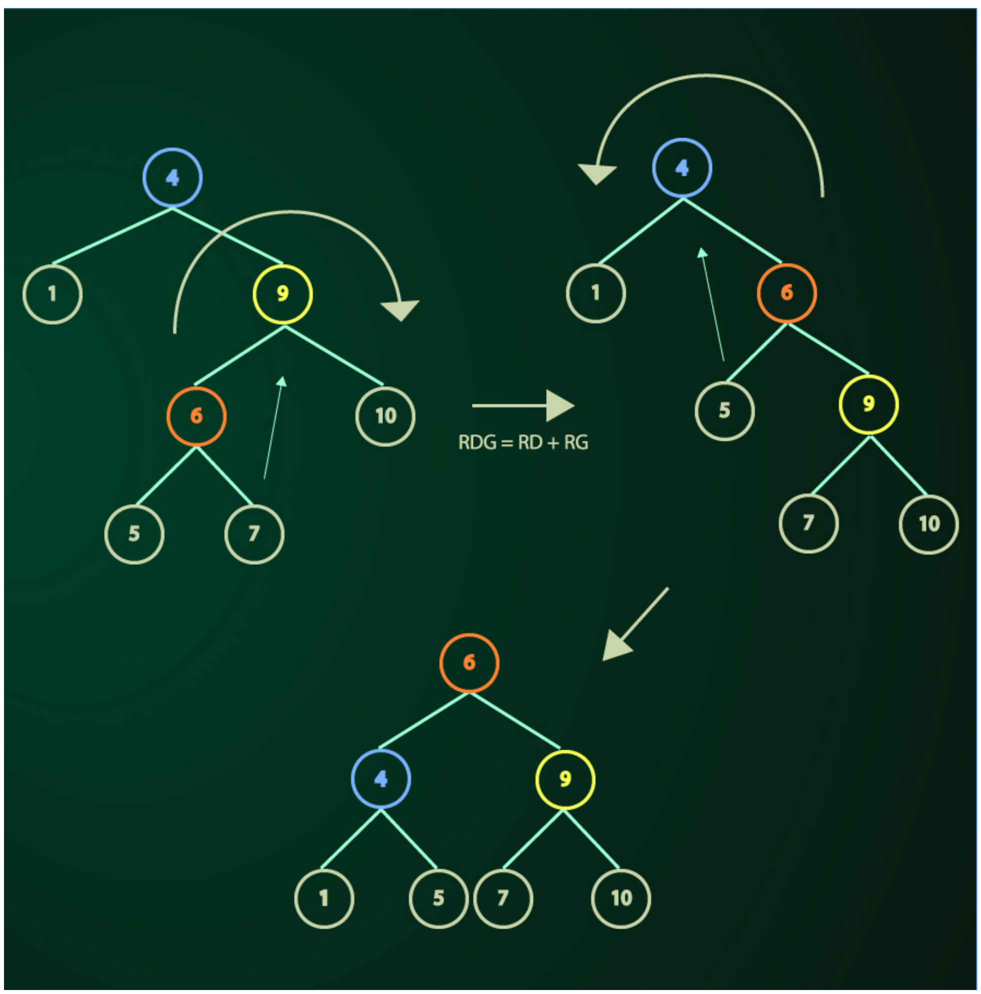


FIGURE 4 – Rotation Droite Gauche

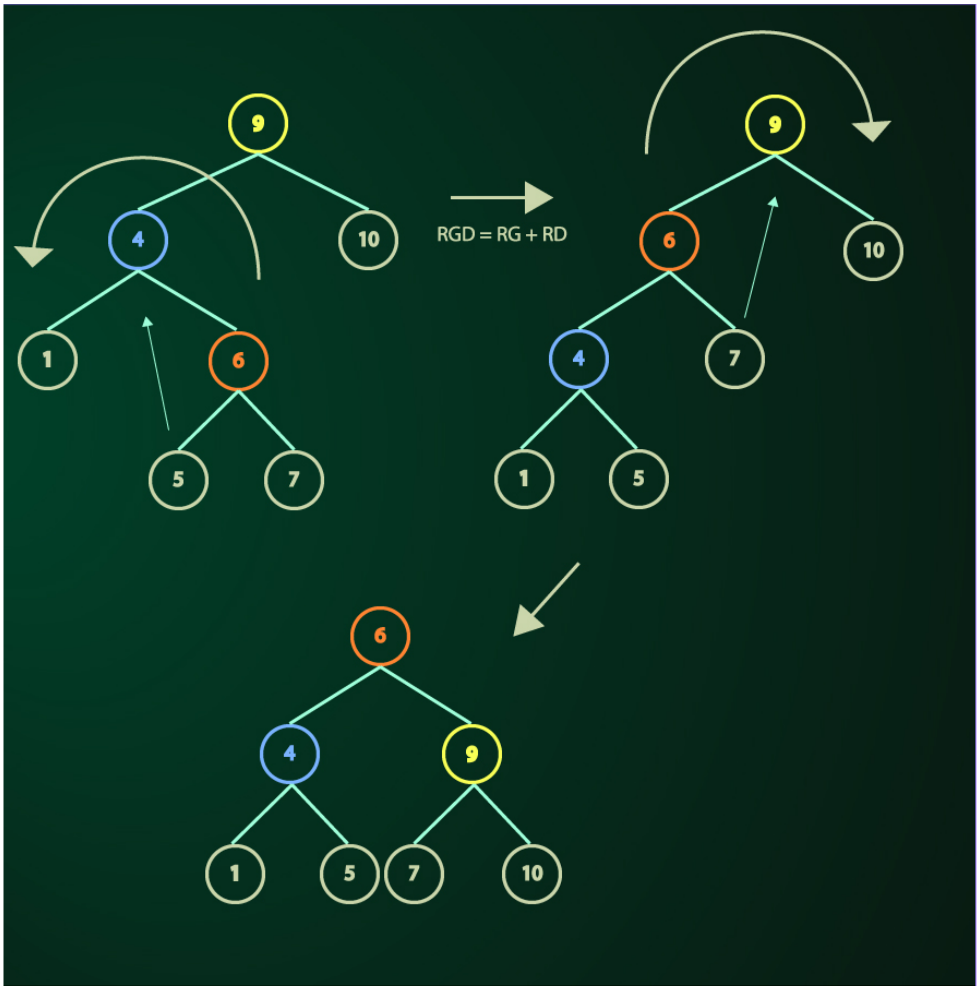


Figure 5 – Rotation Gauche Droite

2.1.2 Algorithme

Lors d'une insertion ou d'une suppression d'un nœud, si votre arbre se déséquilibre :
- Si le nœud x le plus bas déséquilibré a un déséquilibre -2 :
  - Son fils gauche a un déséquilibre $-1 \Rightarrow$ Rotation droite de x
  - Son fils gauche a un déséquilibre $+1 \Rightarrow$ Rotation gauche droite de x
- Sinon le nœud x a un déséquilibre $+2$:
  - Son fils droit a un déséquilibre $-1 \Rightarrow$ Rotation droite gauche de x
  - Son fils droit a un déséquilibre $+1 \Rightarrow$ Rotation gauche de x

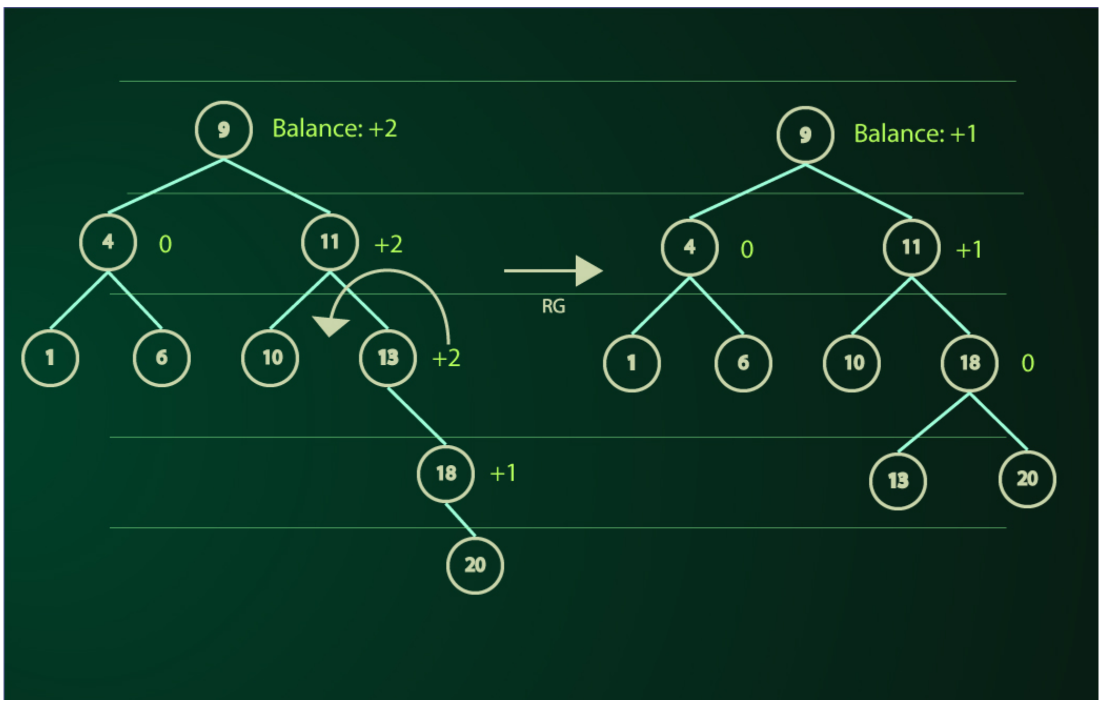


FIGURE 6 – Équilibrage d'un arbre

3 TP

Implémenter les fonctions suivantes à l'aide d'un algorithme récursif :
- **binarySearch**(Array $array$, int $toSearch$) : retourne l'index de la valeur $toSearch$ ou -1 si la valeur n'est pas dans le tableau.
- **binarySearchAll**(Array $array$, int $toSearch$, int indexMin, int indexMax) : rempli l'index minimum et maximum de la valeur $toSearch$. Si la valeur n'est pas dans le tableau rempli les deux index par -1.

Implémenter un arbre binaire de recherche avec les méthodes suivantes :
- **insertNumber**(int value) : insère un nouveau $noeud$ dans l'arbre avec la valeur $value$.
- **height**() : retourne la hauteur de l'arbre.

- **nodesCount**() : retourne le nombre de nœuds de l'arbre.
- **isLeaf**() : retourne vrai si l'arbre est une feuille, faux sinon.
- **allLeaves**(Node* $leaves[]$, int $leavesCount$) : rempli le tableau $leaves$ avec toutes les feuilles de l'arbre.
- **inorderTravel**(Node* $nodes[]$, int $nodesCount$) : rempli le tableau $nodes$ en parcourant l'arbre dans l'ordre infixe (fils gauche, parent, fils droit).
- **preorderTravel**(Node* $nodes[]$, int $nodesCount$) : rempli le tableau $nodes$ en parcourant l'arbre dans l'ordre préfixe (parent, fils gauche, fils droit).
- **postorderTravel**(Node* $nodes[]$, int $nodesCount$) : rempli le tableau $nodes$ en parcourant l'arbre dans l'ordre suffixe (fils gauche, fils droit, parent).
- **insertNumber**(int value) : insertNumber modifié pour garder un arbre équilibré.

Vous pouvez utiliser le langage que vous souhaitez.

3.1 C++

Le dossier *Algorithme_TP3/TP* contient un dossier *C++*. Vous trouverez dans ce dossier des fichiers *exo<i>.pro* à ouvrir avec *QtCreator*, chacun de ces fichiers projets sont associés à un fichier *exo<i>.cpp* à compléter pour implémenter les différentes fonctions ci-dessus.
L'exercice *exo3.cpp* implémente une structure *BinarySearchTree* possédant les différentes méthodes d'un arbre binaire de recherche à implémenter.

```
struct Node {

  Node* left;
  Node* right;
  int value;

  void print()
  {
    if (this->left != nullptr)
      printf("left: %d\n", this->left->value);
    if (this->right != nullptr)
      printf("right: %d\n", this->right->value);
    printf("this: %d\n", this->value);
  }

}
```

Notes :

— Dans une fonction C_{++}, si le type d'un paramètre est accompagné d'un '&' alors il s'agit d'un paramètre entré/sortie. La modification du paramètre se répercute en dehors de la fonction. Pour la fonction $binarySearchAll$, vous pouvez modifier $indexMin$ et $indexMax$ pour retourner les index à calculer.

— La structure $BinaryTree$ est un alias de la structure $Node$, il s'agit de la même structure.

— $BinarySearchTree$ est une spécialisation de $BinaryTree$, il possède donc les mêmes propriétés que la structure $BinaryTree$ et possède donc déjà les attributs $left$,$right$ et $value$

— Lorsque vous faites appel à $this$ dans une méthode d'une structure (ou d'une classe), vous pouvez accéder aux attributs de la structure en question, comme dans l'exemple ci-dessus.
— Vous pouvez utiliser la méthode createNode(int $value$) pour créer un nouveau nœud.