Complexité algorithmique d'un arbre binaire

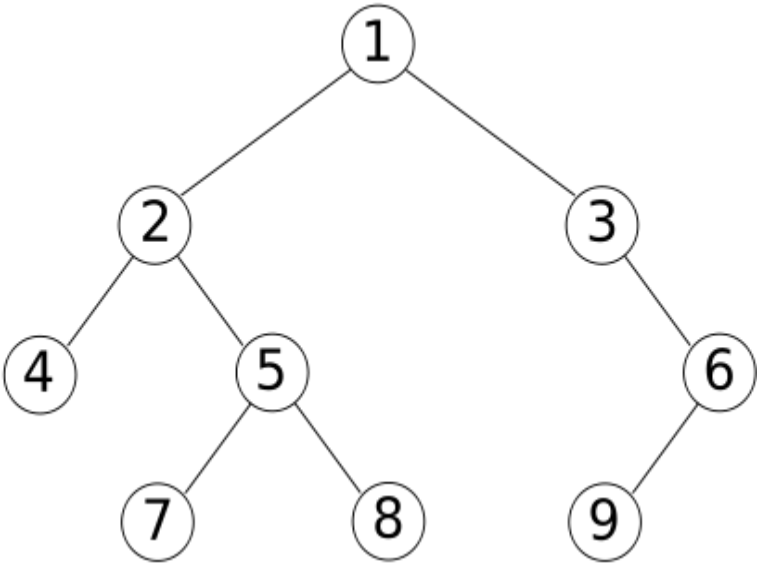
Introduction :

*En informatique, un **arbre binaire** est une structure de données qui peut se représenter sous la forme d'une hiérarchie dont chaque élément est appelé nœud, le nœud initial étant appelé racine. Dans un arbre binaire, chaque élément possède au plus deux éléments fils au niveau inférieur, habituellement appelés gauche et droit. Du point de vue de ces éléments fils, l'élément dont ils sont issus au niveau supérieur est appelé père.*

Au niveau le plus élevé il y a donc un nœud racine. Au niveau directement inférieur, il y a au plus deux nœuds fils. En continuant à descendre aux niveaux inférieurs, on peut en avoir quatre, puis huit, seize, etc. c'est-à-dire la suite des puissances de deux. Un nœud n'ayant aucun fils est appelé feuille. Le nombre de niveaux total, autrement dit la distance entre la feuille la plus éloignée et la racine, est appelé hauteur de l'arbre.

Le niveau d'un nœud est appelé profondeur.

Les arbres binaires peuvent notamment être utilisés en tant qu'arbre binaire de recherche ou en tant que tas binaire.



Enoncé :

Dans ce rapport on va traiter la complexité algorithmique d'un arbre binaire

Un arbre binaire qui n'est pas vide est formé d'un nœud, sa racine, et de deux sous-arbres binaires, l'un appelé le fils gauche, l'autre le fils droit. Nous nous intéressons aux arbres contenant des informations. Chaque nœud porte une information, appelée son contenu. Un arbre non vide est donc entièrement décrit par le triplet (fils gauche, contenu de la racine, fils droit). Cette définition récursive se traduit en une spécification de programmation. Il suffit de préciser la nature du contenu d'un nœud. Pour simplifier, nous supposons que le contenu est un entier. On obtient alors la définition.

```
class Arbre
{
  int contenu;
  Arbre filsG, filsD;
  Arbre(Arbre g, int c, Arbre d)
  {
    filsG = g;
    contenu = c;
    filsD = d;
  }
}
```

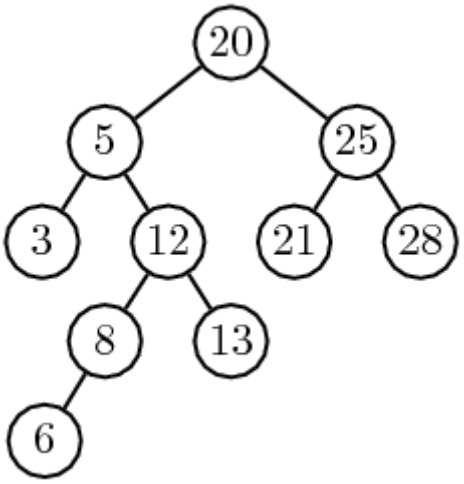
L'arbre vide est, comme d'habitude, représenté par null*. Un arbre réduit à une feuille, de contenu* x*, est créé par*

```
new Arbre(null, x, null)
```

L'arbre de la figure 5.1 est créé par

```
new Arbre(
  new Arbre(
    new Arbre(null, 3, null),
    5,
    new Arbre(
      new Arbre(
       new Arbre(null, 6, null),
       8,
       null)
      12,
      new Arbre(null, 13, null))),
  20,
```

```
  new Arbre(
    new Arbre(null, 21, null),
    25,
    new Arbre(null, 28, null)))
```



Un arbre binaire portant des informations aux nœuds.

Avant de poursuivre, reprenons le schéma déjà utilisé pour les listes. Quatre fonctions caractérisent les arbres: composer, cle, filsGauche, filsDroit. *Elles s'implantent facilement:*

```
static Arbre composer(Arbre g, int c, Arbre d)
{
  return new Arbre(g, c, d);
}
static int cle(Arbre a)
{
  return a.contenu;
}
static Arbre filsGauche(Arbre a)
{
  return a.filsG;
}
static Arbre filsDroit(Arbre a)
{
  return a.filsD;
}
```

Les quatre fonctions sont liées par les équations suivantes:

$$cle(composer(g, c, d)) = c$$

$$\mathit{filsGauche}(\mathit{composer}(g, c, d)) = g$$

$$\mathit{filsDroit}(\mathit{composer}(g, c, d)) = d$$

$$\mathit{composer}(\mathit{filsGauche}(a), \mathit{cle}(a), \mathit{filsDroit}(a)) = a; \quad (a \neq \mathit{null})$$

Comme pour les listes, ces quatre fonctions sont à la base d'opérations non destructives.

La définition récursive des arbres binaires conduit naturellement à une programmation récursive, comme pour les listes. Voici quelques exemples: la taille d'un arbre, c'est-à-dire le nombre t(a) de ses nœuds, s'obtient par la formule

$$t(a) = \begin{cases} 0 & \text{si } a \text{ est vide} \\ 1 + t(a_g) + t(a_d) & \text{sinon.} \end{cases}$$

où sont notés a_g et a_d les sous-arbres gauche et droit de a. D'où la méthode

```
static int taille(Arbre a)
{
  if (a == null)
    return 0;
  return 1 + taille(a.filsG) + taille(a.filsD);
}
```

Des formules semblables donnent le nombre de feuilles ou la hauteur d'un arbre. Nous illustrons ce style de programmation par les parcours d'arbre définis page ??. Les trois parcours en profondeur s'écrivent:

```
static void parcoursPréfixe(Arbre a)
{
  if (a == null)
    return;
  System.out.print(a.contenu + " ");
  parcoursPréfixe(a.filsG);
  parcoursPréfixe(a.filsD);
}
static void parcoursInfixe(Arbre a)
```

```
{
  if (a == null)
    return;
  parcoursInfixe(a.filsG);
  System.out.print(a.contenu + " ");
  parcoursInfixe(a.filsD);
}
static void parcoursSuffixe(Arbre a)
{
  if (a == null)
    return;
  parcoursSuffixe(a.filsG);
  parcoursSuffixe(a.filsD);
  System.out.print(a.contenu + " ");
}
```

Le parcours en largeur d'un arbre binaire s'écrit simplement avec une file. Le parcours préfixe s'écrit lui aussi simplement de manière itérative, avec une pile. Nous reprenons les classes Pile *et* File *du chapitre 1, sauf que ce sont, cette fois-ci, des piles ou des files d'arbres. On écrit alors*

```
static void parcoursPréfixeI(Arbre a)
{
  if (a == null)
    return;
  Pile p = new Pile();
  p.ajouter(a);
  while (!p.estVide())
  {
    a = p.valeur();
    p.supprimer();
    System.out.print(a.contenu + " ");
    if (a.filsD != null)
      p.ajouter(a.filsD);
    if (a.filsG != null)
      p.ajouter(a.filsG);
  }
}
static void parcoursLargeurI(Arbre a)
{
  if (a == null)
    return;
  File f = new File();
  f.ajouter(a);
  while (!f.estVide())
  {
```

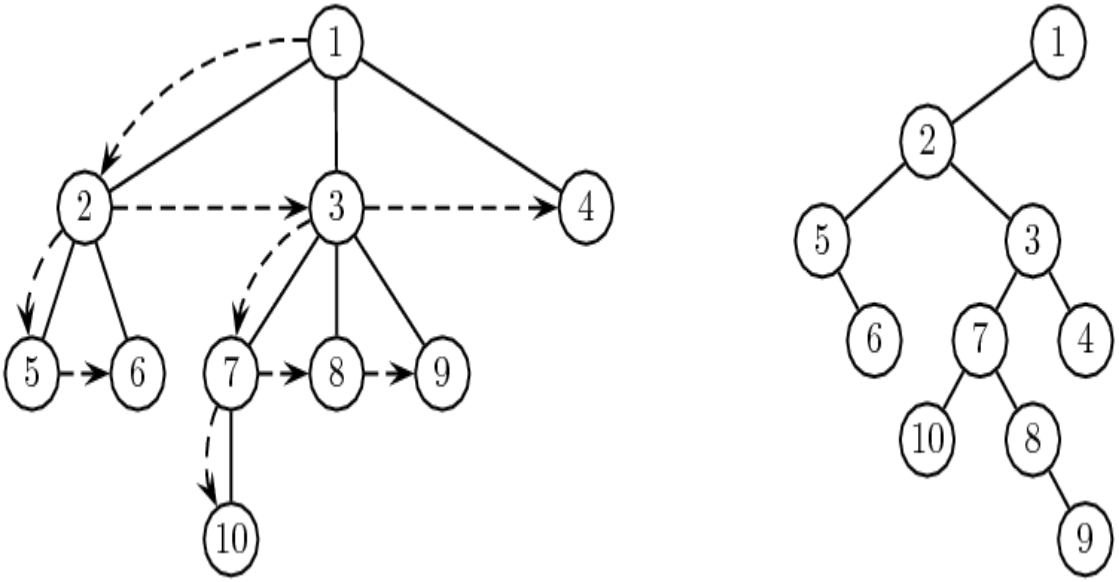
```
    a = f.valeur();
    f.supprimer();
    System.out.print(a.contenu + " ");
    if (a.filsG != null)
      f.ajouter(a.filsG);
    if (a.filsD != null)
      f.ajouter(a.filsD);
  }
}
```

Implantation des arbres ordonnés par arbres binaires

Rappelons qu'un arbre est ordonné si la suite des fils de chaque nœud est ordonnée. Il est donc naturel de représenter les fils dans une liste chaînée. Un nœud contient aussi la référence à son fils aîné, c'est-à-dire à la tête de la liste de ses fils. Ainsi, chaque nœud contient deux références, celle à son fils aîné, et celle à son frère cadet. En d'autres termes, la structure des arbres binaires convient parfaitement, sous réserve de rebaptiser filsAine *le champ* filsG *et* frereCadet *le champ* filsD*.*

```
class ArbreOrdonne
{
  int contenu;
  Arbre filsAine, frereCadet;
  Arbre(Arbre g, int c, Arbre d)
  {
    filsAine = g;
    contenu = c;
    frereCadet = d;
  }
}
```

Cette représentation est aussi appelée « fils gauche — frère droit » (voir figure).



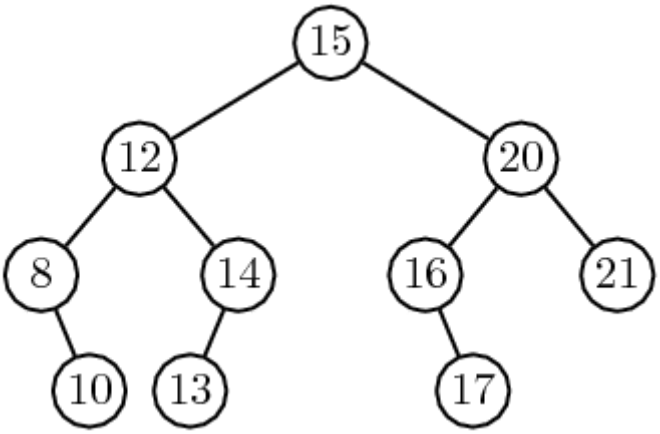
Représentation d'un arbre ordonné par un arbre binaire.

Noter que la racine de l'arbre binaire n'a pas de fils droit. En fait, cette représentation s'étend à la représentation, par un seul arbre binaire, d'une forêt ordonnée d'arbres ordonnés.

Arbres binaires de recherche

Les arbres binaires servent à gérer des informations. Chaque nœud contient une donnée prise dans un certain ensemble. Nous supposons dans cette section que cet ensemble est totalement ordonné. Ceci est le cas par exemple pour les entiers et pour les mots.

Un arbre binaire a est un arbre binaire de recherche si, pour tout nœud s de a, les contenus des nœuds du sous-arbre gauche de s sont strictement inférieurs au contenu de s, et que les contenus des nœuds du sous-arbre droit de s sont strictement supérieurs au contenu de s (cf. figure 5.3).



Un arbre binaire de recherche.

Une petite mise en garde: contrairement à ce que l'analogie avec les tas pourrait laisser croire, il ne suffit pas de supposer que, pour tout nœud s de l'arbre, le contenu du fils gauche de s soit strictement inférieur au contenu de s, et que le contenu du fils droit de s soit supérieur au contenu de s. Ainsi, dans l'arbre de la figure 5.3, si on change la valeur 13 en 11, on n'a plus un arbre de recherche...

Une conséquence directe de la définition est la règle suivante:

***Règle.** Dans un arbre binaire de recherche, le parcours infixe fournit les contenus des nœuds en ordre croissant.*

Une seconde règle permet de déterminer dans certains cas le nœud qui précède un nœud donné dans le parcours infixe.

***Règle.** Si un nœud possède un fils gauche, son prédécesseur dans le parcours infixe est le nœud le plus à droite dans son sous-arbre gauche. Ce nœud n'est pas nécessairement une feuille, mais il n'a pas de fils droit.*

Ainsi, dans l'arbre de la figure 5.3, la racine possède un fils gauche, et son prédécesseur est le nœud de contenu 14, qui n'a pas de fils droit...

Les arbres binaires de recherche fournissent, comme on le verra, une implantation souvent efficace d'un type abstrait de données, appelé dictionnaire, qui opère sur un ensemble totalement ordonné à l'aide des opérations suivantes:

- *rechercher un élément*
- *insérer un élément*
- *supprimer un élément*

Le dictionnaire est simplement une table d'associations (chapitre 3) dont les informations sont inexistantes. Plusieurs implantations d'un dictionnaire sont envisageables: par tableau, par liste ordonnés ou non, par tas, et par arbre binaire de recherche. La table 5.1 rassemble la complexité des opérations de dictionnaire selon la structure de données choisie.

Implantation	*Rechercher*	*Insérer*	*Supprimer*
Tableau non ordonné	*O(n)*	*O(1)*[a]	*O(n)*
Liste non ordonnée	*O(n)*	*O(1)*[a]	*O(1)*[b]
Tableau ordonné	*O(logn)*	*O(n)*	*O(n)*
Liste ordonnée	*O(n)*	*O(n)*	*O(1)*[b]
Tas	*O(n)*	*O(logn)*	*O(n)*
Arbre de recherche	*O(h)*	*O(h)*	*O(h)*

Ces coûts supposent que l'on sait que l'élément inséré n'est pas dans le dictionnaire.

Ces coûts supposent l'élément supprimé déjà trouvé, ainsi que des opérations destructives.

Complexité des implantations de dictionnaires

L'entier h désigne la hauteur de l'arbre. On voit que lorsque l'arbre est bien équilibré, c'est-à-dire lorsque la hauteur est proche du logarithme de la taille, les opérations sont réalisables de manière particulièrement efficace.