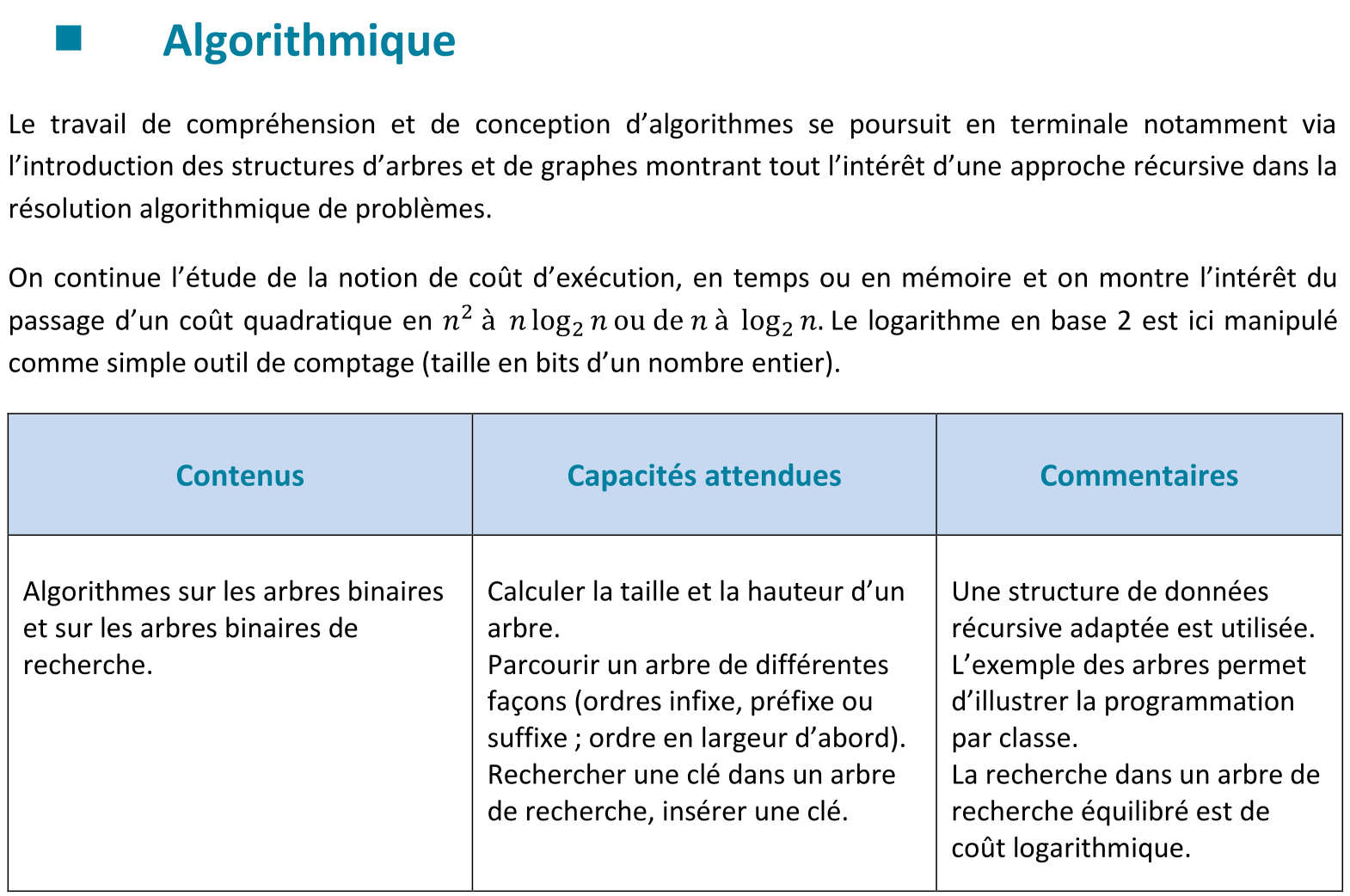
|  |  |
| --- | --- |
| Term NSI | Cours Arbres binaires algorithmes |
| COURS/AD : Cours Arbres binaires algorithmes [Extrait de pixees](https://pixees.fr/informatiquelycee/n_site/nsi_term.html), Ouvrage Ellipses, Nathan et DUI Toulouse | |



**1) Parcours d’arbres binaires :**

**Parcours d’arbre :** Notre objectif est de visiter tous les nœuds d’un arbre, sans pour autant visiter deux fois le même nœud. Dans une liste, il suffit de lire les éléments l’un après l’autre ; mais comme l’arbre est une structure de données non séquentielle, c’est un peu plus complexe.

**Parcours en largeur :**

|  |  |
| --- | --- |
| Le parcours en largeur consiste, simplement, à lire les nœuds dans le sens de la lecture. Si l’arbre est représenté par un tableau, cela revient simplement à lire les cases de celui-ci, l’une après l’autre.  parcours en largeur = A B C D E F |  |

***Ex 1*** : Parcours en largeur  **AD2 Arbres binaires algorithmes  
Remarque :** Le parcours en largeur sera étudié plus en détail, dans le cadre plus général, de l’étude des Graphes.

**Parcours en profondeur d’abord :** Le parcours en profondeur est plus aventureux, il consiste à explorer un chemin jusqu’au bout, avant de revenir en arrière pour emprunter un autre chemin. Cet algorithme s’écrira récursivement, pour s’adapter facilement à notre structure. Remarquons que nous passons sur chaque nœud à trois reprises.

|  |  |  |
| --- | --- | --- |
| –Lorsqu’on traite un nœud dès sa première visite (couleur orange), nous obtenons ***l’ordre préﬁxe*** des nœuds.  Algorithmiquement, cela revient à faire  [noeud] + appelRécursif sur sag + appelRécursif sur sad | Parcours préfixes : ABDECF | |
| Lorsqu’on traite un nœud lors de sa deuxième visite (couleur violette), nous obtenons ***l’ordre inﬁxe*** des nœuds.  Algorithmiquement, cela revient à faire  appelRécursif sur sag + [noeud] + appelRécursif sur sad | Parcours infixes : DBEACF | |
| Lorsqu’on traite un nœud lors de sa dernière visite (couleur verte), nous obtenons ***l’ordre postﬁxe*** des nœuds.  Algorithmiquement, cela revient à faire  appelRécursif sur sag + appelRécursif sur sad + [noeud] | Parcours postfixes : DEBFCA | |
| – **Parcours préﬁxe** : ABDECF (couleur orange)  – **Parcours inﬁxe** : DBEACF (couleur violette)  – **Parcours postﬁxe** : DEBFCA (couleur verte) | |  |

***Ex 2*** : Parcours en largeur**AD2 Arbres binaires algorithmes**

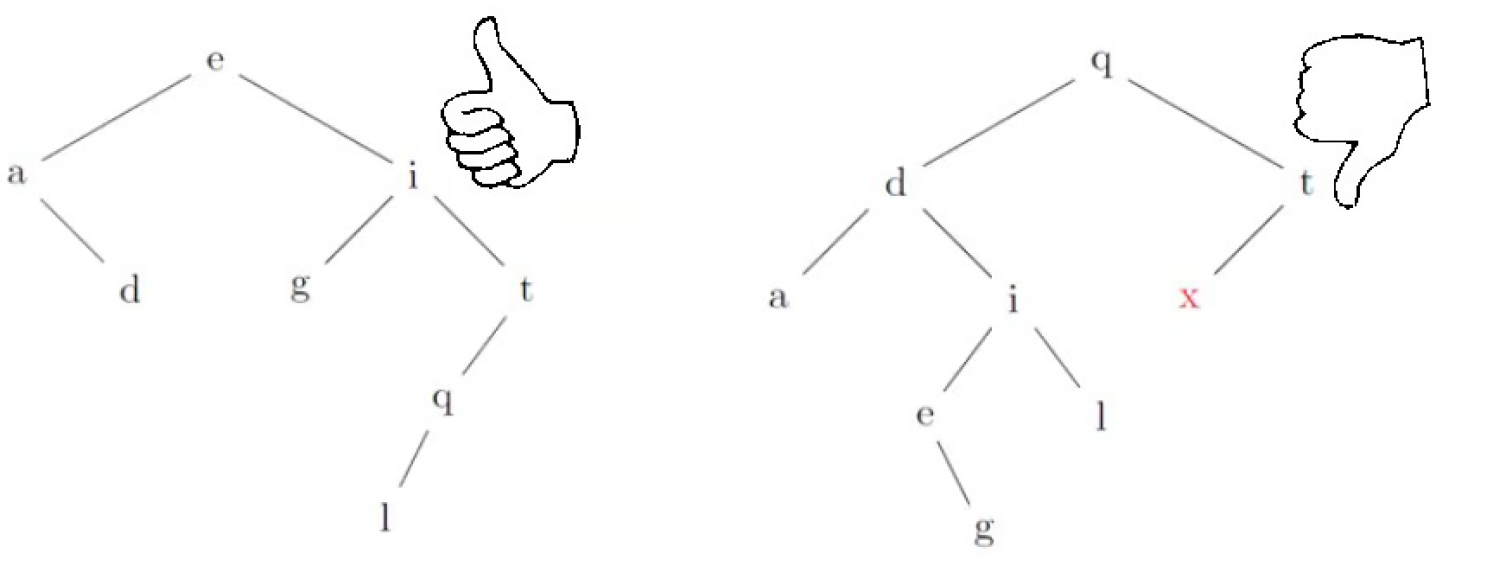
**2) Arbres binaires de recherche :** Un arbre binaire aux propriétés remarquables

**Déﬁnition :** Un arbre binaire de recherche est un arbre binaire dont l’ensemble des nœuds vériﬁent les propriétés suivantes :

– Les valeurs du sous-arbre gauche sont inférieures ou égales à la valeur du nœud.

– Les valeurs du sous-arbre droit sont strictement supérieures à la valeur du nœud.

Exemple :

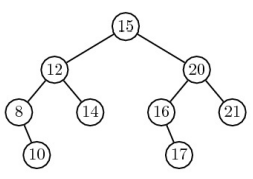
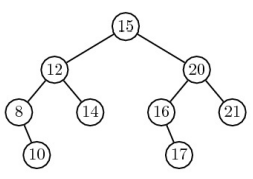


Spéciﬁcation du TAD ABR

Le TAD Arbre Binaire de Recherche (ABR) est une extension de l’Arbre Binaire. Il possède donc les mêmes opérations/préconditions/axiomes auxquels on ajoute :

|  |  |  |
| --- | --- | --- |
| ***Opérations*** **creerArbreVide** : → ArbreBinaire[E] **estVide :** ArbreBinaire[E] → Booléenracine : **ArbreBinaire[E] → E sag, sad :** ArbreBinaire[E] → ArbreBinaire[E] **assembler :** E × ArbreBinaire[E] ×  arbreBinaire[E] → ArbreBinaire[E]  **ajouter :** ABR[E] × E → ABR[E] **retirer :** ABR[E] × E → ABR[E] **rechercher :** ABR[E] × E → Booléen | ***Préconditions***  Soit a un ArbreBinaire. racine(a), sag(a) et sad(a) si et seulement si  **non estVide(a)** | ***Axiomes***  Soient a, g,d des ArbreBinaire[E], r un E.  estVide(arbreVide) = Vrai  estVide(assembler(r, g,d)) = Faux  racine(assembler(r, g,d)) = r |

Ajouter un élément dans un ABR : L’ajout d’un élément dans un ABR se fait au niveau des feuilles.

  
**Arbre A=15**

13

13

→

Nous allons donc naviguer dans l’arbre, en partant de la racine jusqu’à trouver un emplacement qui permettra de conserver la caractérisation d’un ABR. Nous obtenons alors l’algorithme suivant :

**Ajouter(A,x) :   
 Si x > racine(A)   
 Si estVide(sad(A)) alors sad(A) = assembler(x,creerArbre,creerArbre)  
 Sinon ajouter(sad(A),x)  
 Sinon   
 Si estVide(sag(A)) alors sag(A) = assembler(x,creerArbre, creerArbre)  
 Sinon ajouter(sag(A),x)**

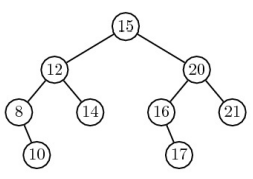
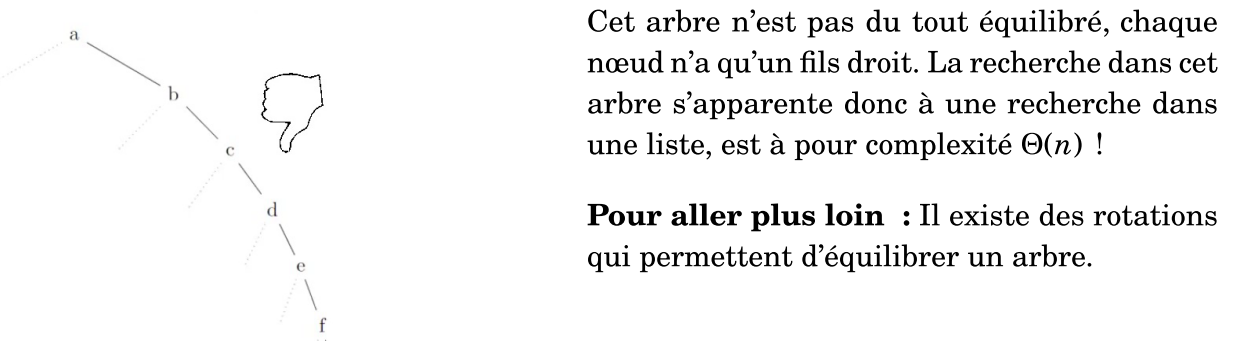
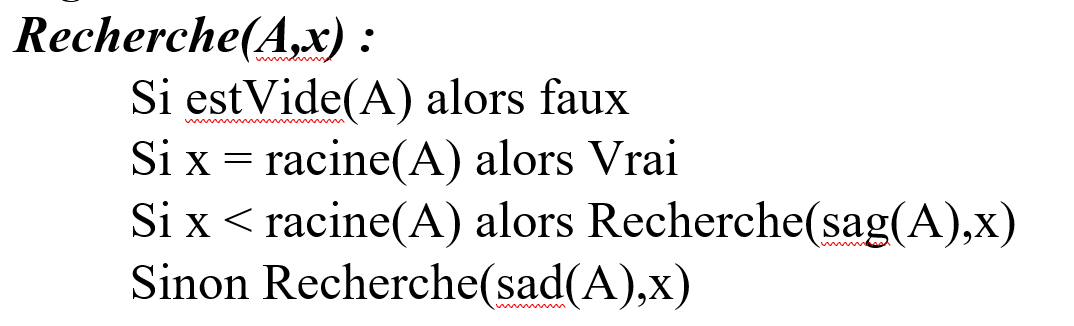
Exemple figure ci-dessus ajouter un nœud 13, appliquer l’algorithme (***compléter les étapes***)

|  |  |  |
| --- | --- | --- |
| **Ligne** | **Programme** | **Etapes (A,B…) d’exécution** |
| **1 2 3 4 5 6 7 8 9 10 11** | **Ajouter(A,x)   Si x > racine(A) alors  Si estVide(sad(A)) alors   sad(A) = assembler(x,creerArbre,creerArbre)  Sinon (estVide(sad(A)=False) alors  Ajouter(sad(A),x)  Sinon ( x <= racine(A) ) alors  Si estVide(sag(A)) alors  sag(A) =assembler(x,creerArbre, creerArbre)  Sinon (estVide(sag(A))=False) alors  Ajouter(sag(A),x)**   |  |  | | --- | --- | | **14** | **14** | | **sag** | **none** | | **sad** | **none** |      |  |  | | --- | --- | | **13** | **13** | | **sag** | **none** | | **sad** | **none** |  |  |  | | --- | --- | | **14** | **14** | | **sag** | **13** | | **sad** | **none** | | **A 1\_Ajouter(A=15,13)  B 2\_Si 13 > racine(A=15)=15 (False) C 7\_Sinon 13 <= racine(A=15)=15 (True) D 8\_Si estVide(sag(15)) (False 12) E 9\_Sinon (estVide(sag(15))=False 12) = True F 11\_Ajouter(sag(15)=12,13) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  G 1\_Ajouter(A=12,13) H 2\_Si 13 > racine(A=12) (True) I 3\_Si estVide(sad(12))= 14 (False) J 5\_ Sinon (estVide(sad(12)=False 14) = True K 11\_Ajouter(sag(12)=14,13) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ L 1\_Ajouter(A=14,13) M 2\_Si 15 > racine(4) (False) N 7\_Sinon ( 15 <= racine(A)=14 ) (True)  O 8\_Si estVide(sag(14)) (True) alors P 9\_sag(14) =assembler(13,creerArbre, creerArbre)** |

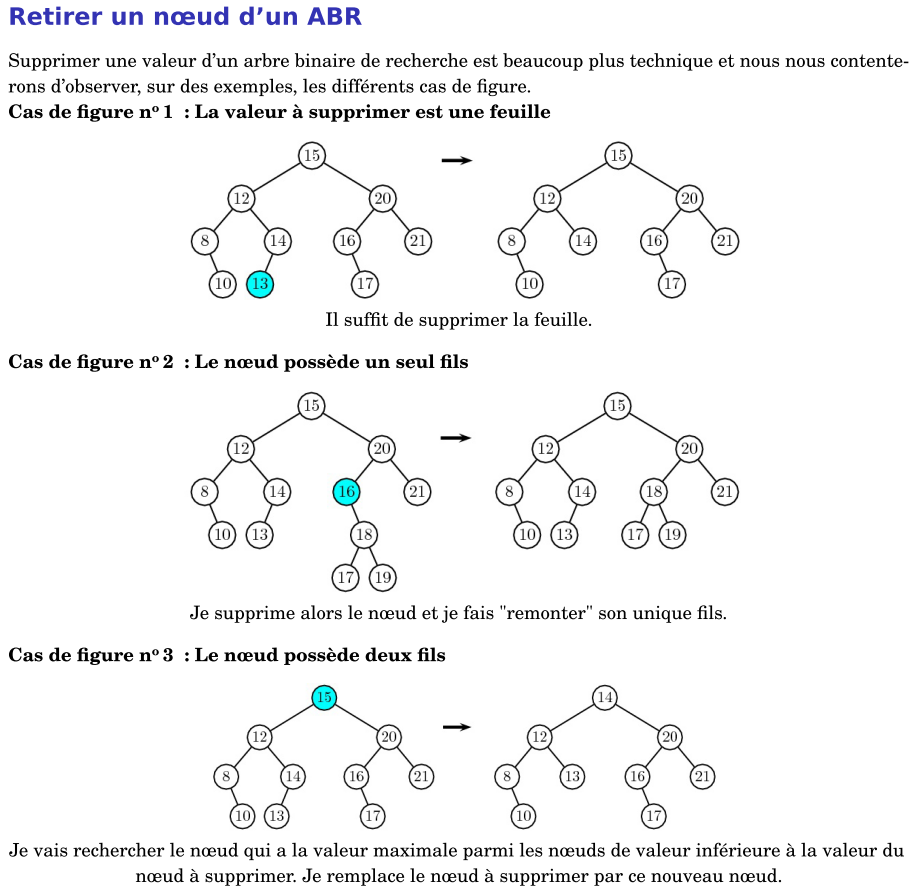
***La recherche dans un ABR*** Algorithme

La recherche dans un arbre binaire de recherche est équivalente à une recherche par dichotomie dans une liste triée. (coût ?? = logarithmique)

***Algorithme   
Recherche(A,x) :*** Si estVide(A) alors faux  
 Si x = racine(A) alors Vrai  
 Si x < racine(A) alors Recherche(sag(A),x)  
 Sinon Recherche(sad(A),x)  
  
En moyenne, la complexité de la recherche dans un ABR est en ϴ(ln(n)).  
Rappelons que rechercher un élément dans une liste quelconque a pour complexité ϴ(n). La recherche dans un **ABR** est donc plus rapide.

***Attention !*** Ces performances dépendent de l’équilibre de l’ABR.  
  
  
  
Rechercher 17 :Appliquer ***Algorithme Recherche(A,x) à l’arbre ci -dessous***  


|  |  |
| --- | --- |
|  | ***Recherche(A=15,x=17) :*** Si estVide(A=15) alors faux Si x=17 = racine(A)=15 alors Vrai Si x= 17 < racine(A)=15 alors Recherche(sag(A),x) Sinon Recherche(sad(A=15)=20,17) ----------------------- ***Recherche(A= 20 , x= 17 ) :*** Si estVide(A= 20 ) alors faux Si x= 17 = racine(A)= 20 alors Vrai Si x= 17 < racine(A)= 20 alors Recherche(sag(A= 20 ) = 16 , x= 17 ) Sinon Recherche(sad(A= 20 )= 21 , x = 17) ----------------------- ***Recherche(A= 16 , x= 17 ) :*** Si estVide(A= 16 ) alors faux Si x= 17 = racine(A)= 16 alors Vrai Si x= 17 < racine(A)= 16 alors Recherche(sag(A= 16 )= 17, x = 17) Sinon Recherche(sad(A= 16)= 17 , x= 17 ) ----------------------- ***Recherche(A= 17 , x= 17 ) :*** Si estVide(A=17) alors faux Si x= 17 = racine(A)= 17 alors Vrai Si x= 17 < racine(A)=20 alors Recherche(sag(A= 16 )= 17 , x= 17 ) Sinon Recherche(sad(A= 16 )= none , x= 17) |

  
  
***Ex 3*** : **Arbre binaire de recherche AD2 Arbres binaires algorithmes  
*Ex 4*** : **Compléter la classe Arbre AD2 Arbres binaires algorithmes**