Sujet TD n°02

Les arbres binaires de recherche

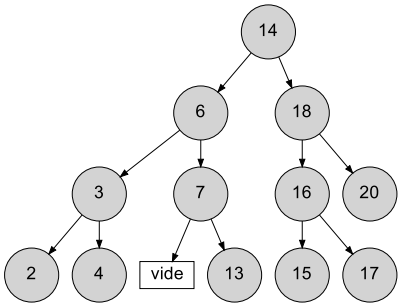
Notions abordées

- ✓ Arbre binaire de recherche
- ✓ Opérations sur les arbres (ajout, suppression de nœuds, parcours)

Les arbres binaires de recherche

Nous allons implémenter des arbres binaires de recherche (ou ABR) et des fonctions pour les manipuler. Un ABR est tout d'abord "binaire" ce qui implique que chacun de ses nœuds peut comporter au maximum deux fils. Il est aussi construit de façon qu'il soit facile de faire une recherche sur les valeurs qu'il contient. Pour cela, on définit les deux règles suivantes :

- Toutes les valeurs inférieures ou égales à celle de la racine sont stockées dans le sous-arbre gauche.
- Toutes les valeurs strictement supérieures à celle de la racine sont stockées dans le sous-arbre droit.

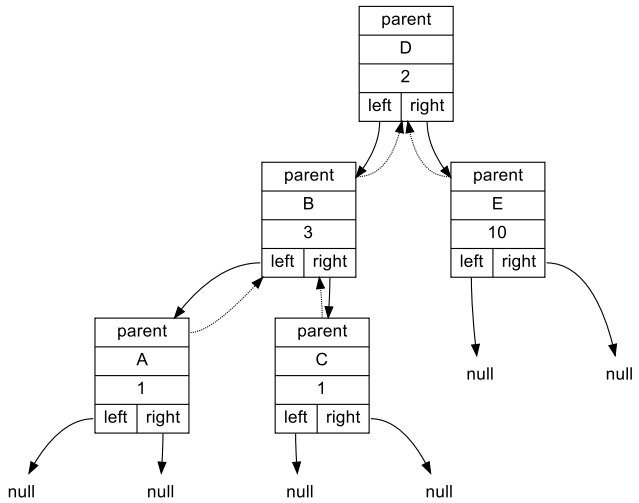


Les sous-arbres gauche et droit de la racine sont aussi des ABR. La figure ci-contre donne un exemple d'ABR contenant des entiers.

Structure de données

Dans la suite de ce TD nous allons utiliser un ABR pour implanter une table de symboles. Tout comme pour la table de hachage, nous allons donc stocker des paires <clés/valeurs>.

L'illustration de droite montre l'exemple d'un arbre binaire de recherche dans lequel on a inséré successivement les paires suivantes : <D,2>, <B,3>, <C,1>, <A,1> et <E,10>.



Pour représenter et manipuler les allocations de mémoire, vous utiliserez le TAD Pointeur décrit ci-dessous.

TA $Pointeur < T >$

Utilise $T,\ Booléen$
Opérations
null: $\rightarrow Pointeur < T >$
contenu: $Pointeur < T > \ \rightarrow T$
adresse_de: $T \rightarrow Pointeur < T >$
estNull: $Pointeur < T > \ \rightarrow Booléen$
creer $< T > : \rightarrow Pointeur < T >$
détruire: $Pointeur < T > \ \rightarrow Pointeur < T >$

Pré-conditions : $p \in Pointeur < T >$
$\text{défini}(\text{contenu}(p)) \Rightarrow \neg\text{estNull}(p)$
$\text{défini}(\text{detruire}(p)) \Rightarrow \neg\text{estNull}(p)$

Axiome : $\forall p \in Pointeur < T >; \forall t \in T$
$\text{estNull}(null) = vrai$
$\text{estNull}(\text{creer} < T >) = faux$
$\text{contenu}(\text{adresse\_de}(t)) = t$

Vous trouverez en complément ci-dessous les définitions partielles de deux TAD qui vous seront utiles pour la suite.

TA $Paire$

Utilise $Clé, Valeur$
Opérations
creerPaire: Clé × Valeur → $Paire$
lireClé: $Paire \rightarrow Clé$
lireValeur: $Paire \rightarrow Valeur$

Pré-conditions :
…
Axiome : $\forall c \in Clé; \forall v \in Valeur$
$\text{lireClé}\big(\text{creerPaire}(c, v)\big) = c$
$\text{lireValeur}\big(\text{creerPaire}(\text{c}, \text{v})\big) = \text{v}$

TA $Noeud$

Utilise: $Paire,\ Booléen, Pointeur$
Opération(s):
$creerNoeud$:
$Paire \times Pointeur < Noeud > \times Pointeur < Noeud > \times Pointeur < Noeud > \rightarrow Pointeur < Noeud >$
$contenuNoeud$: $Pointeur < Noeud > \rightharpoonup Paire$
$estRacine$: $Pointeur < Noeud > \rightharpoonup Booléen$
$estExterne$: $Pointeur < Noeud > \rightharpoonup Booléen$
$estInterne$: $Pointeur < Noeud > \rightharpoonup Booléen$
$gauche$: $Pointeur < Noeud > \rightharpoonup Pointeur < Noeud >$
$droit$: $Pointeur < Noeud > \rightharpoonup Pointeur < Noeud >$
$père$: $Pointeur < Noeud > \rightharpoonup Pointeur < Noeud >$

Pré-condition(s): $\forall n \in Pointeur < Noeud >$
$\text{défini}\big(contenuNoeud(n)\big) \Rightarrow \neg\ \text{estNull(n)}$
$\text{défini}\big(estRacine(n)\big) \Rightarrow \neg\ \text{estNull(n)}$
$\text{défini}\big(estExterne(n)\big) \Rightarrow \neg\ \text{estNull(n)}$
$\text{défini}\big(estInterne(n)\big) \Rightarrow \neg\ \text{estNull(n)}$
$\text{défini}\big(gauche(n)\big) \Rightarrow \neg\ \text{estNull(n)}$
$\text{défini}\big(droit(n)\big) \Rightarrow \neg\ \text{estNull(n)}$
$\text{défini}\big(père(n)\big) \Rightarrow \neg\ \text{estNull(n)}$

Axiomes: $\forall e \in Paire, \forall n, p, g, d \in Pointeur < Noeud >$
(A01) $\text{contenuNoeud}\big(\text{creerNoeud(e, p, g, d)}\big) = \text{e}$
(A02) $\text{pere}\big(\text{creerNoeud(e, p, g, d)}\big) = \text{p}$
(A03) $\text{gauche(creerNoeud(e, p, g, d))} = \text{g}$
(A04) $\text{droit(creerNoeud(e, p, g, d))} = \text{d}$
(A05) estRacine(n)=estNull(pere(n))
(A06) $\text{estExterne}(n) = \neg estRacine(n) \wedge estNull\big(gauche(n)\big) \wedge estNull\big(droit(n)\big)$
(A07) $\text{estInterne}(n) = \neg estRacine(n) \wedge \big(\neg estNull(gauche(n)) \vee \neg estNull(droit(n))\big)$

TA $ArbreBinaireDeRecherche\ (ABR)$

Utilise: $Noeud, Paire, Clé, Booléen, Pointeur$

Opération(s):
$creerABR\ :\ \rightarrow ABR$
$ajouterElement$: $ABR \times Pointeur < Paire > \rightarrow ABR$
$supprimerElement$: $ABR \times Clé \rightharpoonup ABR$
$chercherElement$: $ABR \times Clé \rightarrow Pointeur < Paire >$
$estVide$: $ABR \rightarrow Booléen$
$racine$: $ABR \rightarrow Pointeur < Noeud >$

Pré-condition(s): $\forall a \in Arbre, \forall c \in Clé$
$\text{défini}\big(racine(a)\big) \Rightarrow \neg\ \text{estVide(a)}$
$\text{défini}\big(supprimerElement(a, c)\big) \Rightarrow \neg\ \text{estVide(a)}$

Axiomes:
…

Remarque : comme pour les tables de hachage, une Paire est un tuple <clé/valeur>.

Question 01 : Ecrire, à partir de ce qui précède, et, en utilisant la syntaxe algorithmique étudiée en M1103, la structure de données permettant de représenter et manipuler un arbre binaire de recherche tel qu'illustré plus haut.

Algorithmes sur les arbres binaires de recherche

Question 2 : Écrivez le sous-programme *creerNoeud* du TAD $Noeud$.

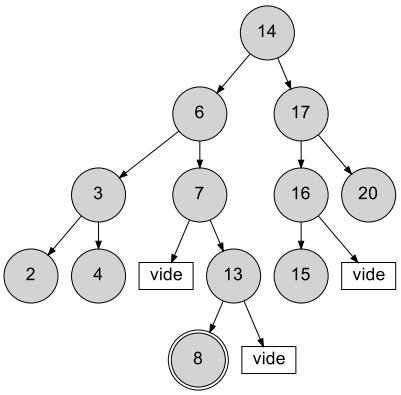
Question 3 : Écrivez le sous-programme *creerABR* du TAD ABR.

Question 4 : Ecrivez les sous-programme estInterne, estExterne et estRacine du TAD $Noeud$.

Question 5 : Écrivez un sous-programme a*fficherCroissant* qui affiche les paires des nœuds d'un arbre par valeur croissante de leur clé.

Question 6 : Écrivez un sous-programme *chercherElement* qui renvoie l'adresse d'un nœud de l'arbre contenant l'élément recherché, *null* s'il n'est pas trouvé.

Question 7 : Écrivez un sous-programme *ajouterElement* qui permet d'insérer un élément dans l'arbre. Pour cela vous utiliserez l'algorithme qui consiste à ajouter aux feuilles : On recherche la valeur x à insérer dans l'arbre. Lorsque l'on arrive au niveau d'une feuille f, si x est inférieur au contenu de f, on ajoute un nouveau nœud de valeur x comme fils gauche, sinon, on ajoute un nouveau nœud de valeur x comme fils droit.

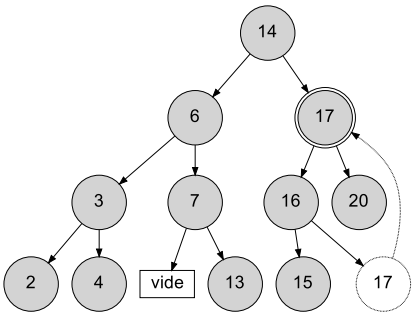


L'illustration de droite montre ce qui se passe lorsque l'on ajoute en feuille l'élément 8 dans l'arbre.

Question 8 : On souhaite écrire une fonction *supprimerElement* qui permet de retirer un élément de l'arbre.

Pour rappel, l'algorithme général est le suivant :

1. Si le nœud à enlever est une feuille, on l'enlève ;
2. Si c'est un nœud qui n'a qu'un fils, on le remplace par ce fils ;
3. Si c'est un nœud qui a deux fils, le remplacer par la feuille de plus grande valeur dans le sous-arbre gauche ou par la feuille de plus petite valeur dans le sous arbre droite.



La figure de droite illustre ce qu'il se passe lorsque l'on supprime un nœud qui contient la valeur 18.

Question 8.1 : Écrivez un sous-programme *supprimerFeuille* permettant de supprimer une feuille de l'arbre. ***Attention,*** *on ne supprime que s'il s'agit d'une feuille, cad un nœud qui n'a pas de successeur (pas de fils).*

Question 8.2 : Écrivez une fonction s*upprimeNoeudInterneUnFils* permettant de supprimer un noeud de l'arbre uniquement s'il s'agit d'un nœud interne n'ayant qu'un seul fils.

Question 8.3 : Écrivez une fonction s*upprimeNoeudInterneDeuxFils* permettant de supprimer un noeud de l'arbre uniquement s'il s'agit d'un nœud interne n'ayant qu'un seul fils.

Question 8.4 : Écrivez la fonction *supprimerElement*. Cette fonction devra utiliser les fonctions ci-dessus pour traiter les cas particuliers.