Laboratoire : Parcours des arbres binaires

1 Objectifs

Le but de ce laboratoire est de découvrir les arbres binaires en implémentant les principales opérations de parcours de cette structure de données.

2 Travail à faire

Vous devez implémenter la classe `Arbre` en respectant l'interface que nous vous fournissons. Vous remarquerez que la classe ne permet pas l'ajout ou le retrait de noeuds, car nous voulons que vous vous concentriez sur les parcours dans un arbre binaire. Il n'est alors possible d'initialiser un `Arbre` que via le constructeur. Afin de construire un `Arbre`, vous devez fournir le tableau des parcours symétrique (en ordre) et priorité au père (pré ordre). Nous vous fournissons la construction utilisant un `std::vector` pour vous aider à implémenter celui utilisant des tableaux dynamiques. Finalement, la classe offre 4 types de parcours : pré ordre, post ordre, en ordre et par niveau. Les trois premiers parcours nécessitent l'utilisation d'une méthode privée récursive.

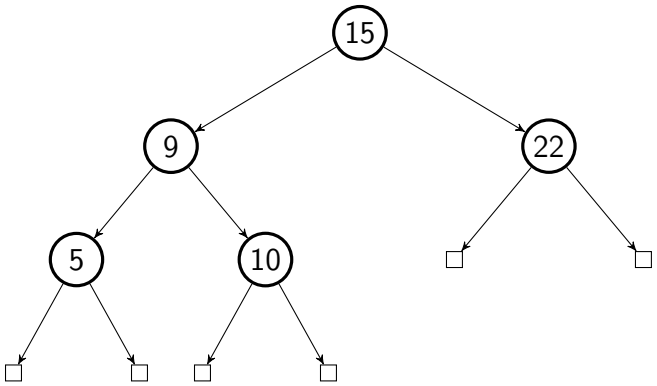
2.1 À propos du constructeur

Le constructeur à implémenter a la signature suivante :

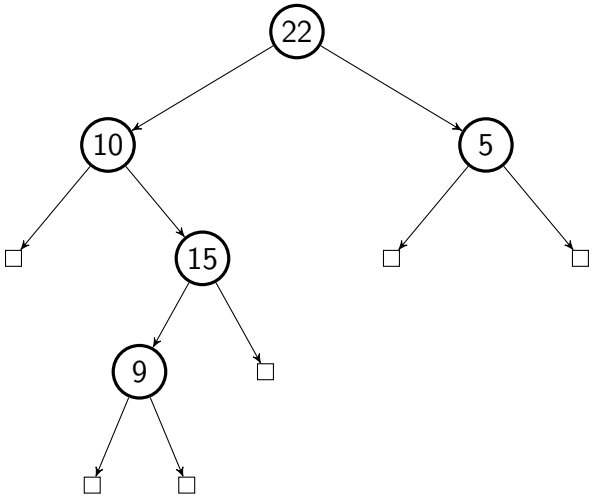
```
Arbre(E * p_visiteSymetrique, int p_debut, int p_fin, E **p_visitePere);
```

Soit A un arbre binaire, soit S l'ordre dans lequel A est parcouru quand on fait un parcours symétrique, et soit P l'ordre dans lequel A est parcouru quand on fait un parcours à priorité aux pères. Si on connaît S et P, il est toujours possible de reconstruire l'arbre A.

Exemple #1 Par exemple, si $S = [5, 9, 10, 15, 22]$ et $P = [15, 9, 5, 10, 22]$. Alors, le seul arbre A possible est l'arbre suivant :



Exemple #2 Si $S = [10, 9, 15, 22, 5]$ et $P = [22, 10, 15, 9, 5]$. Alors, le seul arbre A possible est l'arbre suivant :



Remarquez que dans certains cas, aucun arbre n'existe. Par exemple, si $S = [10, 15, 22, 9, 5]$ et $P = [10, 22, 9, 5, 15]$, il est impossible de construire un arbre A qui y correspond. **Vous n'avez PAS à traiter le cas où l'arbre est impossible à reconstruire** (pas de code d'erreur non plus, ni cas spécial à traiter, faites juste supposer que ça n'arrivera pas).

Vous avez à réaliser un constructeur qui reconstruit A à partir de S et P. La méthode auxiliaire (récursive) associée à ce constructeur devra créer les noeuds de cet arbre à partir de rien (avec des `new`), les connecter entre eux de la bonne façon, et retourner la racine de cet arbre. Le prototype obligatoire de cette méthode auxiliaire est le suivant :

```
Noeud * _auxPereSym(E *p_visiteSymetrique, int p_debut, int p_fin,
                    E **p_visitePere);
```

où `p_visiteSymetrique` est un tableau dynamique contenant les éléments de S, les paramètres `p_debut` et `p_fin` contiendront les indices de début et de fin du tableau

`p_visiteSymetrique`, et `p_visitePere` correspond au tableau dynamique contenant les éléments de P, passé par adresse.

2.2 À propos des parcours

Parcours pré ordre

1. Visiter la racine
2. Parcourir récursivement le sous-arbre de gauche
3. Parcourir récursivement le sous-arbre de droite

Parcours en ordre

1. Parcourir récursivement le sous-arbre de gauche
2. Visiter la racine
3. Parcourir récursivement le sous-arbre de droite

Parcours post ordre

1. Parcourir récursivement le sous-arbre de gauche
2. Parcourir récursivement le sous-arbre de droite
3. Visiter la racine

Parcours par niveau

1. Mettre la racine dans une file
2. Retirer le noeud du début de la file pour l'examiner
3. Mettre tous les voisins (fils) non explorés dans la file (à la fin)
4. Si la file n'est pas vide, reprendre à l'étape 2

3 Documentation

Voir la section Documentation/Normes sur le site Web du cours. Vous y trouverez la description des commentaires attendus dans un programme ainsi que des normes de programmation en vigueur dans notre cours.

4 Important

1. Nous vous fournissons des tests unitaires *Google Test* que vous pouvez utiliser pour vérifier votre implémentation.
2. Vous êtes tenu de faire la gestion des exceptions dans les méthodes que vous avez à implémenter. Référez-vous à la documentation *Doxygen* fournie avec l'énoncé pour connaître les types d'exception que vous avez à gérer pour chacune d'elles. Vous devez utiliser le cadre de la théorie du contrat que nous avons préparé dans les fichiers `ContratException.cpp` et `ContratException.h` disponibles sur le site du cours.
3. Vous devez documenter chaque méthode, que vous avez à implémenter, avec les commentaires de *Doxygen*. Référez-vous à section de Doxygen sur le site Web du cours ainsi qu'aux exemples de cette semaine pour découvrir les commentaires que vous avez à écrire. Assurez-vous de respecter les normes de programmation en vigueur disponibles également sur le site du cours.
4. Vous devez générer la documentation en format HTML. À cette fin, nous vous fournissons un fichier de configuration `sdd.doxyfile` que vous pouvez utiliser tel quel.
5. Nous vous encourageons à utiliser au maximum la partie privée de la classe afin d'y ajouter des fonctions utilitaires privées. Elles permettent d'augmenter la lisibilité du code et de réduire la duplication de code identique.

Bon travail !