

ENPC – PRALG

TP : structure de données d'arbre

Renaud Marlet
Laboratoire LIGM-IMAGINE

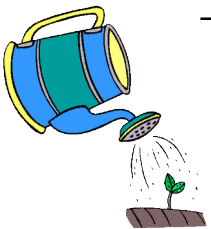
<http://imagine.enpc.fr/~marletr>

Structure de données

cf. cours correspondant

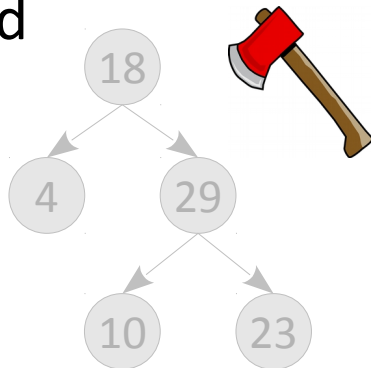
• Création

- nœud / arbre
- constructeur
 - avec info initiale ou valeur par défaut



• Destruction

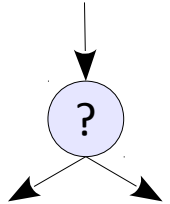
- sous-arbre (recursive)
- nœud



• Opérations :

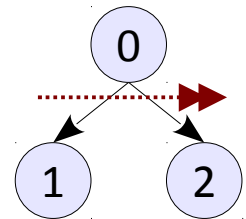


- accéder à l'info d'un nœud
 - lecture, écriture



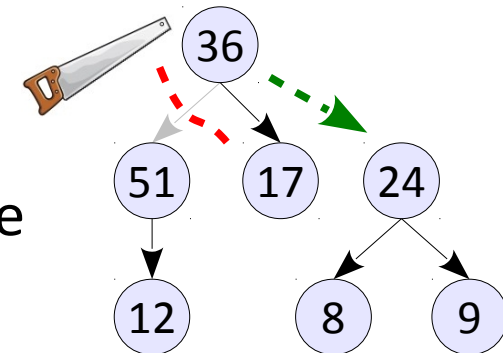
- parcourir (ex. itérateur)

- les fils d'un nœud
- les nœuds d'un arbre



- ajouter/supprimer

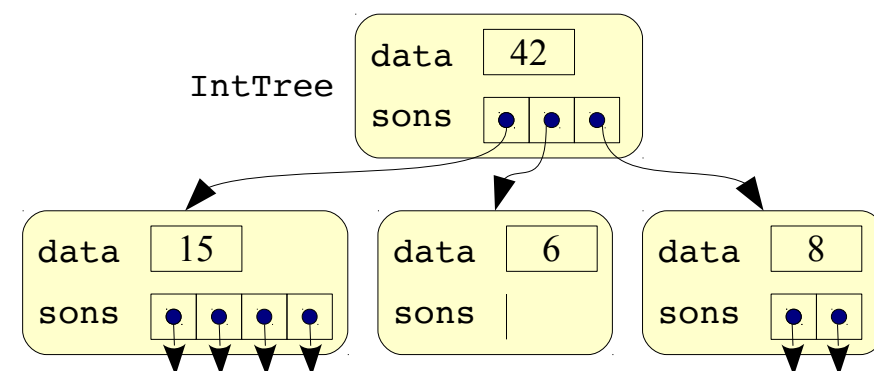
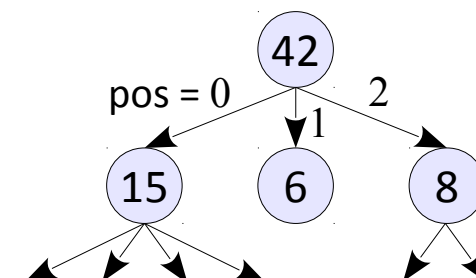
- un fils (sous-arbre)
 - nouvelle branche
 - à une position ou un rang donné



Exemple d'interface

// Node of a tree containing an integer at each node

```
class IntTree {
    // Node information
    int data;
    // Sequence of sons (empty if none)
    vector<IntTree*> sons;
public:
    // Create a node with given information
    IntTree(int d);
    // Destruct a node and all its descendants
    ~IntTree();
    // Return information of this node
    int getData();
    // Set information of this node
    void setData(int d);
    // Return the number of sons of this node
    int nbSons();
    // Return the son at position pos, if any (considering left-most son is at position 0)
    IntTree* getSon(int pos);
    // Replace the existing son at position pos with newSon (left-most son at position 0)
    void setSon(int pos, IntTree* newSon);
    // Add newSon as supplementary right-most son of this node
    void addAsLastSon(IntTree* newSon);
    // Remove right-most son of this node
    void removeLastSon();
};
```



Indice :
opérations déjà
+/- disponibles dans
la classe vector <T>

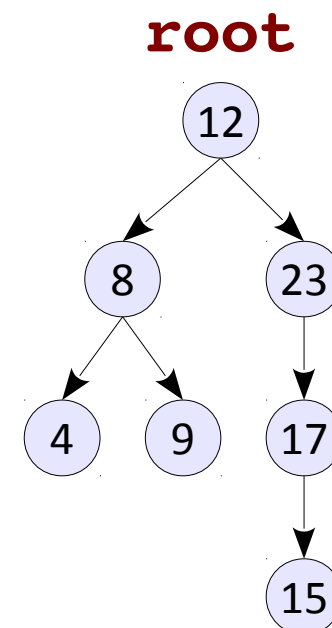
Exercice 1 : classe d'arbre (d'entiers)

1.1) Implémenter **IntTree** (*≈ 1 ligne par fonction !*)

- séparer l'implémentation en **IntTree.h** (*≈ p.18*) et **IntTree.cpp**
- destructeur (libération de la mémoire) :
 - arbre parcouru en profondeur d'abord, libération de chaque nœud en remontant
 - hyp. : on ne libère que des racines, pas des sous-arbres, et ils ne sont pas partagés
- ignorer de la gestion d'erreur pour le moment (voir exo 3)

1.2) Construisez l'arbre ci-contre dans une variable :

```
IntTree* root = new IntTree(12);
root->addAsLastSon(new IntTree(8));
root->getSon(0)->addAsLastSon(new IntTree(4));
root->getSon(0)->addAsLastSon(new IntTree(9));
root->addAsLastSon(new IntTree(23));
root->getSon(1)->addAsLastSon(new IntTree(17));
root->getSon(1)->getSon(0)->addAsLastSon(new IntTree(15));
```



Exercice 2 : affichage d'un arbre

2.1) À quel parcours de l'arbre correspond la suite 12 8 4 9 23 17 15 ?

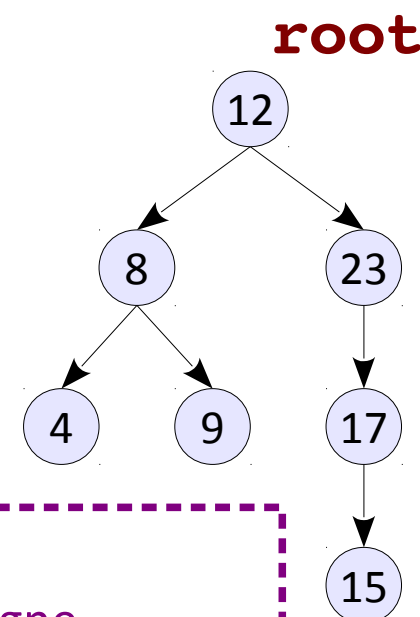
2.2) Ajouter une méthode récursive **void display()** telle que **root->display()** affiche : 12 8 4 9 23 17 15

2.3) Modifier la méthode d'affichage en

```
void display(string prefix = "",  
             string indent = " ")
```

pour que **root->display(" * ")** affiche :

```
* 12
*   8
*     4
*     9
*   23
*     17
*       15
```



Indications

prefix : affiché au début de chaque ligne
avant d'afficher la valeur du nœud

indent : ajouté à **prefix** à chaque niveau de
profondeur supplémentaire (affichage d'un fils)

Exercice 3 : gestion d'erreur

- 3.1) Lister tous les cas d'erreur pour les fonctions de **IntTree**
- 3.2) Pour lesquelles peut-on signaler l'erreur par valeur de retour ?
Auxquelles peut-on facilement ajouter un statut d'erreur ?
- 3.2) Pour lesquelles peut-on signaler l'erreur par exception ?
- 3.4) Choisir un mode de signalement d'erreur et le justifier
- 3.5) Implémenter le signalement d'erreur (0-3 lignes par fonction)
- 3.6) Documenter le signalement d'erreur [Optionel : en Doxygen]
→ compléter le commentaire en tête de chaque fonction
- 3.7) Tester la gestion d'erreur de chaque fonction de **IntTree**
→ le code de test du rattrapage d'erreur dans le programme principal (main.cpp) doit afficher les erreurs rencontrées

Doxygen

```
/**
 * Node of a tree containing an integer at each node.
 * @author Marc Ottage
 */
class IntTree { ...
    /**
     * Constructor. Create a node with given information.
     * @param d information on this node
     */
    IntTree(int d);
    /**
     * Return the son at position pos, if any.
     * @param pos position of the son (considering left-most son is at position 0)
     * @return son at position pos if pos is valid, 0 otherwise (= NULL)
     */
    IntTree* getSon(int pos); // Alternative 1 (gestion d'erreur par valeur de retour)
    /**
     * Return the son at position pos, if any.
     * @param pos position of the son (considering left-most son is at position 0)
     * @return son at position pos
     * @throws out_of_range if pos is not a valid position (between 0 and nbSons-1)
     */
    IntTree* getSon(int pos); // Alternative 2 (gestion d'erreur par exception)
};
```

cf. doxygen.org

Exercice 4 : templatisation

- 4.1) Rendre **IntTree** générique pour le type des données :
Écrire une classe **Tree<T>** qui prend le type en argument
- 4.2) Peut-on séparer **Tree<T>** en 2 fichiers **Tree.h** et **Tree.cpp** pour la compilation séparée?
Si oui, le faire ; sinon expliquer.
- 4.3) Faut-il changer la gestion d'erreur ?
Si oui, le faire ; sinon expliquer.
- 4.4) Définir dans **Tree<T>** les fonctions suivantes, avec leur gestion d'erreur (≈ 3 lignes) :
- ```
// Insert extra son at position pos, if pos exists
void insertSon(int pos, Tree<T>* son);
// Remove son at position pos, thus reducing nbSons
void removeSon(int pos);
```

indice : utiliser  
vect.insert(...)  
vect.begin()+pos  
vect.erase(...)



# Exercice 5 : différents parcours d'arbre

## [Optionnel : points supplémentaires]

- 5.1) Ajouter à **Tree** des fonctions de parcours qui affichent les infos
- en profondeur d'abord en entrant (= en descendant, ~ display)
  - en profondeur d'abord en sortant (= en remontant)
  - en largeur d'abord
- 5.2) Implémenter dans **Tree<T>** une fonction **int maxDepth()**
- qui calcule vite et sans allouer de mémoire la profondeur maximale d'un arbre (= de la feuille la plus profonde)
  - quel type de parcours utiliser et pourquoi ?
- 5.3) Implémenter dans **Tree** une fonction **int minDepth()**
- qui calcule rapidement la profondeur minimale d'un arbre (= profondeur de la feuille la moins profonde)
  - quel type de parcours utiliser et pourquoi ?