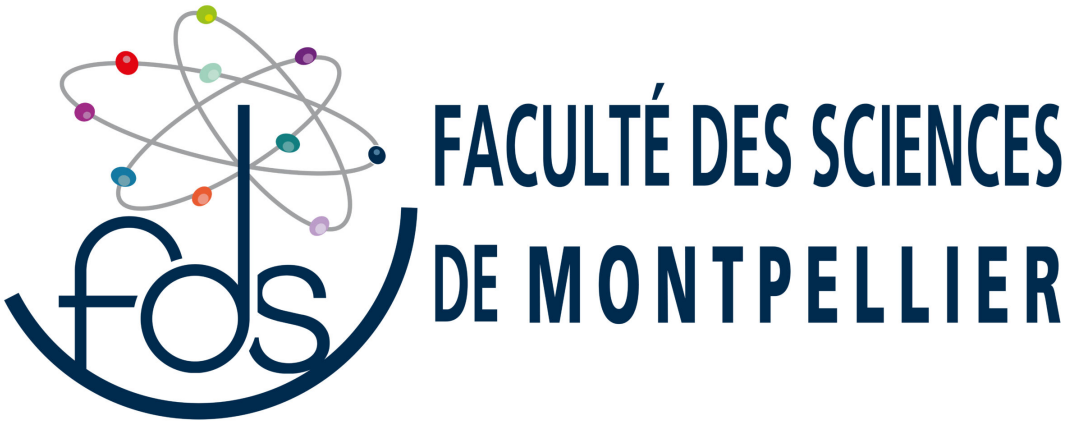
Université de Montpellier

M1 IMAGINE - Algorithme d'exploration et mouvement
Compte Rendu 1 - Structure et parcours d'un arbre binaire

Etudiant :
Guillaume Bataille

Encadrant :
Madalina Croitoru

Année 2022-2023



0.1 Contexte et objectif

Lors du cours, nous avons vu l'importance et la pertinence des arches binaires pour gérer des donnéees mais surtout des parcours.
L'objectif est donc de créer une structure d'arbre en C++ et d'y implémenter deux méthodes de parcours : En **largeur** et en **profondeur**.

Sommaire

1. Illustration et struct de l'arbre

On stocke notre arbre de cette façon :

```
0 1 2 3
1 4 5
2
3 6
```

Chaque ligne représente un noeud suivi de ses fils

Et voici l'idée de l'arbre binaire via un dessin :

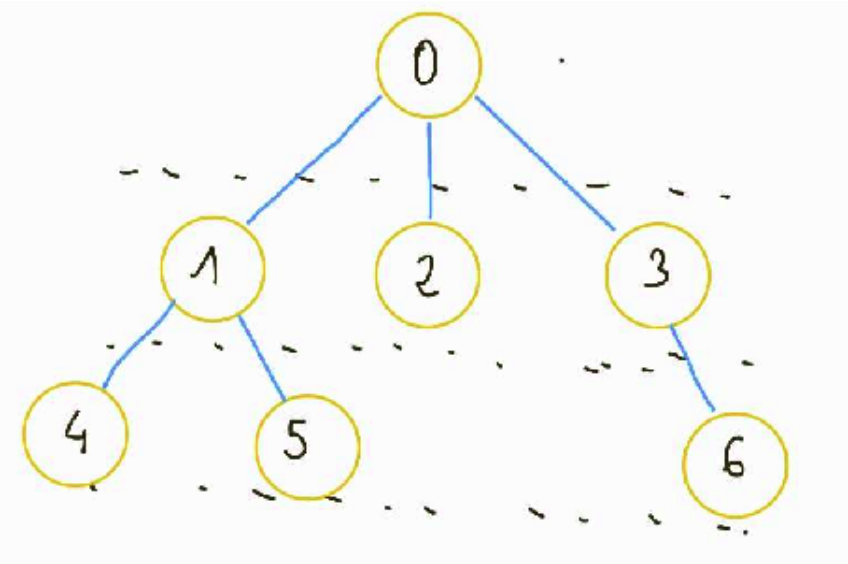


FIGURE 1.1 – Représentation de l'arbre représentant les données du dessus

2. Code

Dans notre code, nous avons deux classes , la classe Tree et la classe Node.
La classe Node contient un pointeur parent privé, un vecteur privé de pointeurs d'enfants, et des variables publiques id et size. La classe dispose de méthodes pour créer un enfant, tester s'il y a des enfants, obtenir le nombre d'enfants et obtenir un pointeur vers les enfants.

La classe Tree a un pointeur vers le noeud racine et des méthodes pour créer un arbre à partir d'un fichier, effectuer une recherche en largeur d'abord et effectuer une recherche en profondeur d'abord. La méthode createFromFile lit un fichier donné et crée un arbre en utilisant les informations contenues dans le fichier. Les méthodes breadthFirstSearch et depthFirstSearch parcourent l'arbre en utilisant une file d'attente et une pile respectivement, en imprimant l'ID de chaque noeud visité.

```
1
2   //------------------------ classe Node -----------------------------
3
4   class Node
5   {
6   private:
7       Node *parent = nullptr;      // ptr vers parent
8       std::vector<Node *> childs; //liste de noeud enfant
9
10  public:
11      int id;
12      int size = 0;
13
14      Node(){};
15      Node(Node *parentPtr, int id)
16      {
17          this->id = id;
18          parent = parentPtr;
19      }
20      Node(int id)
21      {
22          this->id = id;
23      }
24      ~Node(){};
25
26      Node *createChild(int id) //Creation d'enfant
27      {
28          Node *nodePtr = new Node(this, id);
29          childs.push_back(nodePtr);
30          size++;
31          return nodePtr;
32      }
```

```
33
34      bool hasChild() // Teste si y'a un enfant
35      {
36          if (childs.size() == 0)
37          {
38              return false;
39          }
40          return true;
41      }
42
43      int getSize() // Retour du nbr d'enfant
44      {
45          return childs.size();
46      }
47
48      std::vector<Node *> *getChilds()
49      {
50          return &childs;
51      }
52  };
53
54
55
56  // ----------------------------- class Tree ----------------------------------------
57
58  class Tree
59  {
60  private:
61      Node *root; //Noeud initial
62
63  public:
64      Tree(){};
65      ~Tree(){};
66
67      void createFromFile(std::string path)
68      {
69          // Ouverture du fichier en lecture
70          std::ifstream file;
71          file.open(path);
72
73          // Vérification de l'ouverture du fichier
74          if (!file.is_open())
75          {
76              std::cout << "Erreur lors de la lecture du fichier" << std::endl;
77              return;
78          }
79
80          // Map pour stocker les noeuds créés en utilisant leur ID comme clé
81          std::map<int, Node *> created;
82          std::string line;
83
```

```
84          // Lecture de chaque ligne du fichier
85          while (std::getline(file, line))
86          {
87              // Parsing de la ligne pour obtenir les ID des parents et des enfants
88              std::stringstream ss(line);
89              std::vector<int> parsedLine;
90
91              std::string word;
92              while (std::getline(ss, word, ' '))
93              {
94                  parsedLine.push_back(std::stoi(word));
95              }
96
97              // Récupération de l'ID du parent
98              int idParent = parsedLine[0];
99              assert(!parsedLine.empty());
100             parsedLine.erase(parsedLine.begin()); // suppression de la première valeur
101
102             // Récupération du noeud parent s'il a déjà été créé, sinon création d'un nouveau noeud
103             Node *actualNode;
104             if (created.count(idParent))
105             {
106                 actualNode = created.at(idParent); // récupération du pointeur vers le noeud parent
107             }
108             else
109             {
110                 actualNode = new Node(idParent);
111                 created.insert(std::pair<int, Node *>(idParent, actualNode)); // insertion dans la map
112                 root = actualNode;
113             }
114
115             // Parcours des enfants et création de chaque noeud enfant
116             for (int idChild : parsedLine)
117             {
118                 Node *childPtr = actualNode->createChild(idChild);
119                 created.insert(std::pair<int, Node *>(idChild, childPtr));
120             }
121         }
122
123         // Fermeture du fichier
124         file.close();
125     }
126
127     // ----------------------------parcours en largeur ----------------------------
128     void breadthFirstSearch()
129     {
130         // Initialisation d'une file d'attente pour stocker les noeuds à visiter
131         std::queue<Node *> idStack;
132         // Vérification que le noeud racine existe
133         if (root != nullptr)
134         {
```

```
135             // Ajout du noeud racine à la file d'attente
136             idStack.push(root);
137         }
138 
139         // Boucle while qui continue tant que la file d'attente n'est pas vide
140         while (!idStack.empty())
141         {
142             // Récupération du premier élément (le prochain noeud à visiter) de la file d'attente
143             Node *actualNode = idStack.front();
144             // Suppression de ce noeud de la file d'attente
145             idStack.pop();
146 
147             // Affichage de l'ID du noeud courant
148             std::cout << actualNode->id << " |--> ";
149 
150             // Parcours des enfants du noeud courant
151             for (Node *child : *actualNode->getChilds())
152             {
153                 // Ajout des enfants à la file d'attente pour qu'ils soient visités plus tard
154                 idStack.push(child);
155             }
156         }
157 
158         // Affichage de "End." pour indiquer la fin de la recherche
159         std::cout << "End." << std::endl;
160     }
161 
162     // ----------------------------parcours en profondeur ----------------------------
163     void depthFirstSearch()
164     {
165         // Initialisation d'une pile pour stocker les noeuds à visiter
166         std::deque<Node *> idStack;
167         // Vérification que le noeud racine existe
168         if (root != nullptr)
169         {
170             // Ajout du noeud racine à la pile
171             idStack.push_front(root);
172         }
173 
174         // Boucle while qui continue tant que la pile n'est pas vide
175         while (!idStack.empty())
176         {
177             // Récupération du premier élément (le prochain noeud à visiter) de la pile
178             Node *actualNode = idStack.front();
179             // Suppression de ce noeud de la pile
180             idStack.pop_front();
181 
182             // Affichage de l'ID du noeud courant
183             std::cout << actualNode->id << " |--> ";
184 
185             // Récupération des enfants du noeud courant
```

```
186             std::vector<Node *> *childs = actualNode->getChilds();
187 
188             // Parcours des enfants en utilisant un itérateur
189 
190             for (auto it = childs->rbegin(); it != childs->rend(); it++)
191             {
192                 idStack.push_front(*it);
193             }
194         }
195 
196         std::cout << "End." << std::endl;
197     }
198 };
199 
200 #endif
201 
```

---

3. Resultat

Voici ce qu'on obtient avec nos deux parcours pour l'arbre donnée au dessus :

```
reading...
Breadth-First-Search : //Recherche en largeur
0 |--> 1 |--> 2 |--> 3 |--> 4 |--> 5 |--> 6 |--> End.
Depth-First-Search : //Recherche en profondeur
0 |--> 1 |--> 4 |--> 5 |--> 2 |--> 3 |--> 6 |--> End.
```