

Ecole Publique d'Ingénieurs en 3 ans

Rapport

ALGORITHMIQUE AVANCEE

Le 02 juin 2024

LY Johann
OUAZZANI-CHADI Hamza
Année Universitaire 2023/2024
Informatique P2026

Encadrant TP : Patrick DUCROT
Professeur matière : Loïck Lhote



TABLE OF CONTENTS

1. Listes chaînées	7
1.1. Fonctions	7
1.1.1. Création d'une nouvelle liste	7
1.1.2. Libération de la mémoire	7
1.1.3. Affichage d'une liste	8
1.1.4. Trouver une clé donnée dans une liste	8
1.1.5. Supprimer une clé donnée dans une liste	9
1.1.6. Ajouter un couple donné (clé, valeur) dans une liste	10
1.2. Tests et affichage	11
2. Table de hachage	12
2.1. Fonctions	13
2.1.1. Génération de hachés	13
2.1.2. Affichage d'une table de hachage donnée	13
2.1.3. Création d'une table de hachage	13
2.1.4. Insertion d'un couple (clé, valeur) dans une table sans la redimensionnée	14
2.1.5. Destruction d'une table	14
2.1.6. Redimension d'une table de hachage donnée	15
2.1.7. Insertion d'un couple (clé, valeur) dans une table avec redimension si nécessaire	15
2.1.8. Recherche d'une valeur par clé donnée dans la table	15
2.1.9. Suppression d'un couple (clé, valeur) par clé donnée	15
2.2. Exemple	16
3. Piles	17
3.1. Fonctions	17
3.1.1. Création d'une pile	17
3.1.2. Ajout d'un nouvel élément au sommet de la pile	17
3.1.3. Suppression de l'élément au sommet de la pile	18
3.1.4. Consultation de la valeur du sommet de la pile	19

3.1.5.	Vérification de l'état de la pile	19
3.2.	Affichage de la pile	19
4.	Files	20
4.1.	Fonctions	20
4.1.1.	Création d'une nouvelle file vide	20
4.1.2.	Vérification de la file	21
4.1.3.	Ajout d'un élément dans une file	21
4.1.4.	Suppression de la tête de file	22
4.1.5.	Valeur de l'élément en tête de file	22
4.2.	Traces d'exécutions	23
5.	Arbres binaires de recherche (ABR)	24
5.1.	Définition de la structure	25
5.2.	Fonctions	25
5.2.1.	Création d'un arbre binaire de recherche vide	25
5.2.2.	Libération de la mémoire d'un ABR	25
5.2.3.	Ajout d'une valeur dans un ABR	25
5.2.4.	Hauteur d'un ABR	28
5.2.5.	Trouver une valeur dans un ABR	29
5.2.6.	Suppression de la racine d'un ABR	30
5.2.7.	Suppression d'une valeur d'un ABR	31
5.2.8.	Construire un ABR à partir d'une permutation	32
5.2.9.	Affichage d'un ABR	32
5.3.	Traces d'exécutions	32
5.4.	Résultats et critiques	33
6.	Arbres binaires de recherche randomisés (RBST)	37
6.1.	Définition de la structure	37
6.2.	Fonctions	38
6.2.1.	Taille d'un RBST	38
6.2.2.	Couper un RBST en fonction d'une valeur donnée	38
6.2.3.	Insertion d'une valeur donnée à la racine d'un RBST	40
6.2.4.	Insertion d'une valeur donnée dans un RBST	41
6.2.5.	Création d'un RBST à partir d'une permutation donnée	42

6.3.	Traces d'exécutions	42
6.4.	Résultats et critiques	43
7.	Arbres rouges-noirs	49
7.1.	Définition et propriétés de la structure de données	49
7.2.	Fonctions	50
7.2.1.	Rotation d'un ARN autour d'un nœud	50
7.2.2.	Équilibrage d'un ARN	52
7.2.3.	Insertion d'une valeur dans une ARN	55
7.2.4.	Calcul de la hauteur noire d'un ARN	56
7.2.5.	Vérification si un ARN possède une structure valide	56
7.3.	Traces d'exécutions	56
7.4.	Résultats et critiques	57
8.	File de priorité (Heap)	62
8.1.	Définition de la structure	62
8.2.	Fonctions	62
8.2.1.	Création d'un tas vide dont les clés sont $\{1, \dots, n\}$	62
8.2.2.	Affichage des données d'un tas	62
8.2.3.	Retourner l'élément avec la plus faible priorité dans le tas	63
8.2.4.	Supprimer l'élément de plus faible priorité dans le tas	63
9.	Graphes	65
9.1.	Fonctions	65
9.1.1.	Ajout d'une arête dans un graphe	65
9.1.2.	Création du graphe	65
9.1.3.	Affichage du graphe	65
9.1.4.	Parcours en profondeur dans le graphe	67
9.1.5.	Parcours en largeur dans le graphe	68
9.1.6.	Calcul du nombre de composantes connexes	69
10.	Arbre couvrant minimum (PRIM)	70
10.1.	Introduction	70
10.2.	Fonction Prim	70
10.2.1.	Traces d'exécution	71
11.	Graphes orientés acycliques	73

11.1. Introduction	73
11.2. Fonctions	73
11.2.1. topologicalSearch	73
11.2.2. computeEarliestStartDates	74
11.2.3. computeLatestStartDates	74
11.3. Complexité	75
11.4. Traces d'exécution	76

INTRODUCTION

Ce rapport présente l'implémentation de plusieurs structures de données et de méthodes algorithmiques fondamentales pour toute personne étudiant le développement logiciel. Cette implémentation s'inscrit dans le cadre du cours Algorithmique Avancée au sein de l'ENSICAEN.

Dans le cadre de cette implémentation, nous avons bénéficié d'une organisation méthodique facilitant notre progression. En effet, on nous a fourni pour chaque structure de données et méthodes algorithmiques, les prototypes de chaque fonction ainsi que de certains tests unit. Ce cadre préétabli nous amène donc à simplement implémenter le corps de ces fonctions.

Cette implémentation se divise en plusieurs travaux pratiques réalisés en langage de programmation C présentant des dépendances entre eux. Ces dépendances sont illustrées dans le schéma ci-dessous.

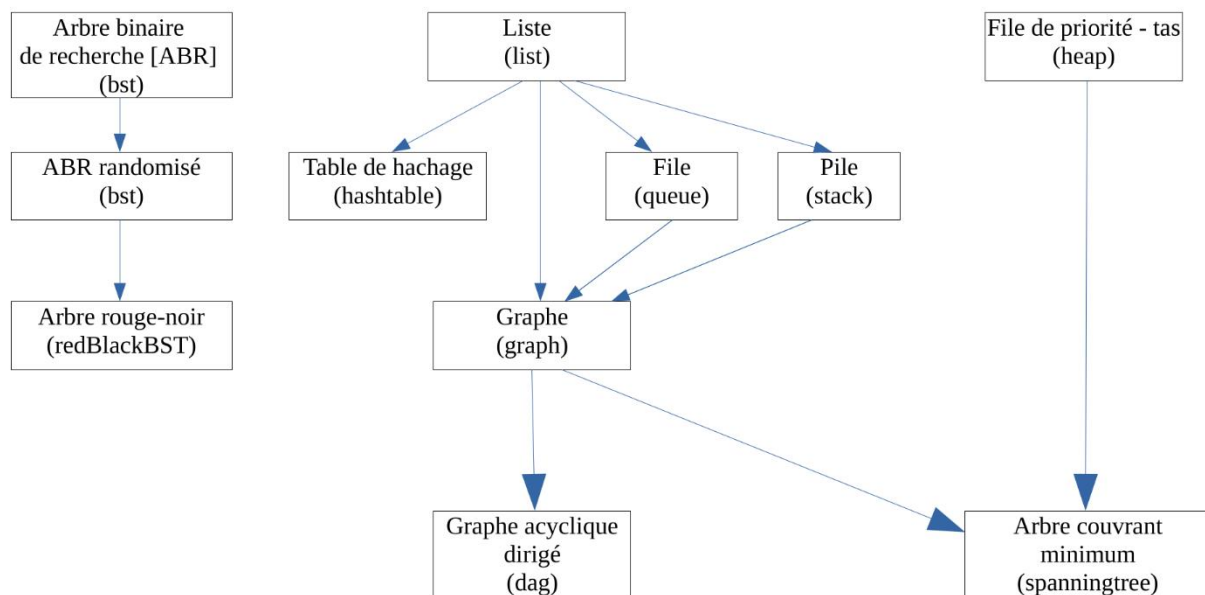


Figure 1 Organisation des Tps et leurs dépendances

IMPLEMENTATION

1. Listes chaînées

Les listes chaînées (qu'on va appeler liste pour faciliter les notations) vont nous permettre d'implémenter les piles, les files, les tables de hachage ainsi que les graphes. Pour couvrir l'ensemble de ces structures de données, les listes doivent contenir des couples (clé : string, valeur : int).

Une liste 'List' est définie comme un pointeur sur la première cellule ('Cell') de la liste (pointeur NULL si la liste est vide).

Une cellule est composée d'une clé, d'un entier et d'un pointeur vers la cellule suivante (pointeur NULL pour la dernière cellule de la liste).

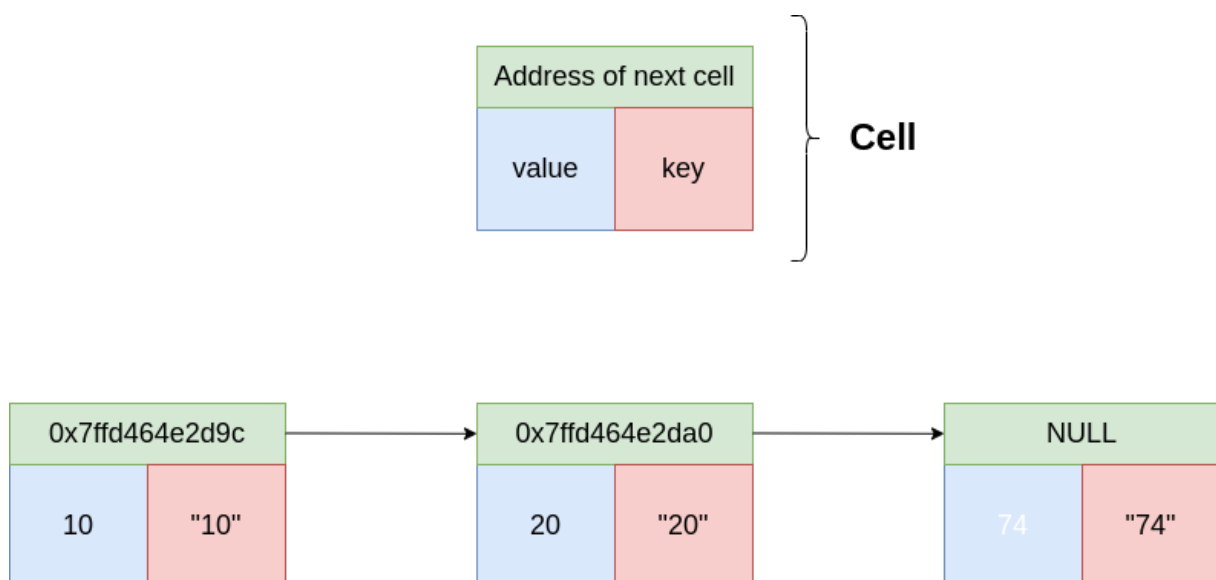


Figure 2 Structures de données Cell et List

1.1. Fonctions

1.1.1. Création d'une nouvelle liste

La fonction `newList()` de type `List` renvoie simplement une adresse nulle.

1.1.2. Libération de la mémoire

La fonction '`freeList()`' est essentielle afin de libérer la mémoire allouée à une liste. Elle est de type `void` et prend en paramètre un pointeur vers la première cellule de la liste.

Cette fonction procède à une libération récursive de la mémoire de chaque cellule de la liste. Dans un premier temps, on vérifie si le pointeur passé en paramètre est nul. Si c'est le cas, notre

liste est vide est donc aucune action n'est requise et on termine. Ensuite on vérifie si la liste ne contient qu'une cellule, et dans ce cas, on libère la mémoire allouée pour la clé de cette cellule puis la mémoire allouée pour la cellule elle-même. Si la liste contient plus d'une cellule, on effectue un appel récursif sur la cellule suivante de la liste et on libère la mémoire allouée de la clé de la cellule actuelle et la cellule actuelle elle-même. Ainsi on libère la mémoire de la dernière cellule à la première lorsque l'on dépile les appels récursifs.

Cette approche garantit que toutes les ressources allouées pour chaque cellule de la liste soient correctement libérées, évitant ainsi les fuites mémoires potentielles.

1.1.3. Affichage d'une liste

La fonction 'printList()' permet d'avoir un affichage console d'une liste. Elle prend en paramètre la liste que l'on veut afficher et un entier 'type'. La fonction imprime le contenu de la liste dans la console selon le type spécifié. Si type est égal à 0, seuls les valeurs des cellules sont imprimées. Sinon, la fonction imprime les couples (clé, valeur) des cellules.

On procède simplement de manière récursive où à chaque appel récursif on imprime les informations de la cellule pointée passée en paramètre et on appelle la fonction pour la cellule suivante. On s'arrête lorsque la cellule pointée passée en paramètre ne possède pas de cellule suivante (arrivé à la fin de la liste). On gère aussi le cas où l'on essaye d'afficher une liste vide.

1.1.4. Trouver une clé donnée dans une liste

La fonction 'findKeyInList' prend en paramètres une liste et une clé, et retourne un pointeur vers la première cellule contenant cette clé, ou un pointeur NULL si la clé n'est pas présente dans la liste.

Dans un premier temps, on vérifie que la liste en entrée n'est pas vide. Ensuite, on procède de manière récursive jusqu'à la dernière cellule de la liste.

On fait une disjonction de cas pour gérer les situations où clé à une adresse NULL ou non. Ceci est nécessaire car pour comparer deux chaînes de caractères et déterminer si elles sont identiques, nous utilisons la fonction 'strcmp' de la bibliothèque 'stdlib.h' qui possède un comportement indéterminé si l'un de ses paramètres est NULL.

A chaque appel récursif, on vérifie si la cellule pointée par le pointeur en entrée contient la clé. Si c'est le cas, on retourne l'adresse de cette cellule. Sinon on effectue un appel récursif sur la cellule suivante.

En vérifiant à chaque appel récursif si l'entrée est un pointeur nul ou non, on s'assure de retourner un pointeur nul si la clé n'est pas présente dans la liste.

Cette approche récursive permet d'effectuer une recherche efficace dans la liste, garantissant une couverture complète des cellules jusqu'à ce que la clé soit trouvée ou que la fin de la liste soit atteinte.

1.1.5..Supprimer une clé donnée dans une liste

La fonction 'delKeyInList()' supprime une clé donnée dans une liste. Elle prend en paramètre une liste et la clé à supprimer, et retourne un pointeur vers la première cellule de la liste après suppression. Seule la première occurrence de la clé est supprimée de la liste.

On vérifie d'abord si la liste est vide. Si c'est le cas, on retourne simplement un pointeur nul.

Ensuite, on utilise la fonction 'findKeyInList' pour trouver la cellule cible à supprimer. Si la clé n'est pas trouvée, on retourne simplement un pointeur vers la liste inchangée.

Si la clé est trouvée dans la liste, on procède à sa suppression en fonction de plusieurs cas :

- Si la liste contient seulement une cellule, à fortiori c'est celle à supprimer. On libère donc la mémoire allouée pour la clé de la cellule ainsi que celle de la cellule elle-même.
- Si la liste contient au moins deux cellules et que la cible est la première cellule de la liste, on libère l'espace mémoire nécessaire à cette cellule (la clé et la cellule elle-même) puis on retourne un pointeur vers la cellule suivante. On stockera donc l'adresse de la cellule suivante avant de libérer la cellule cible.
- Si la liste contient au moins deux cellules et que la clé à supprimer se trouve dans une cellule autre que la première, on parcourt la liste jusqu'à trouver la cellule précédant la cellule cible. Une fois la cellule précédente trouvée, on ajuste les pointeurs pour « contourner » la cellule cible et on libère les ressources utilisées par la cellule cible.

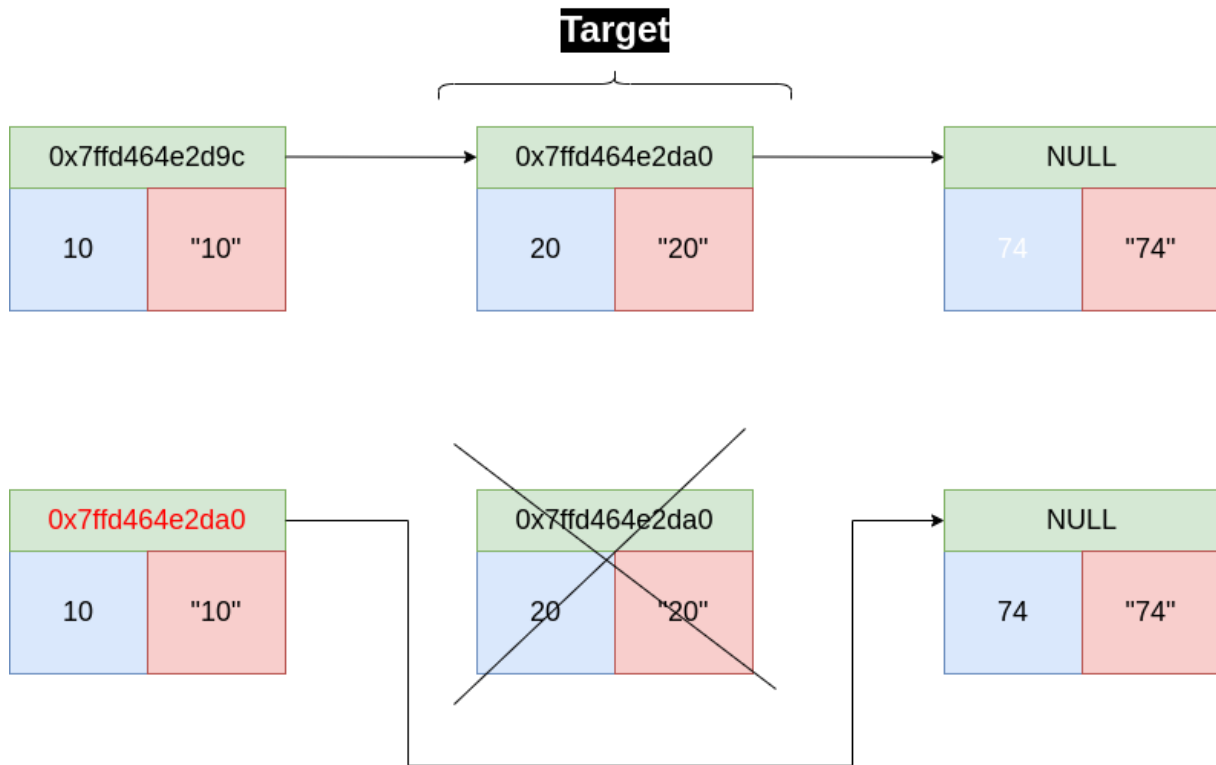


Figure 3 Exemple de suppression d'une clé donnée dans une liste

1.1.6. Ajouter un couple donné (clé, valeur) dans une liste

La fonction 'addKeyValueInList()' permet d'ajouter un couple (clé, valeur) au début d'une liste. Elle prend en paramètres un pointeur vers la liste, la clé à ajouter et la valeur correspondante. La fonction retourne un pointeur vers la première cellule de la liste modifiée.

Dans un premier temps on alloue dynamiquement la mémoire pour créer une nouvelle cellule qui va contenir le couple donné et on va lui affecter la clé, la valeur et l'adresse de la cellule suivante aux champs correspondants. On met le champ 'clé' et 'adresse de la cellule suivante' à NULL, et on affecte la valeur passée en paramètre au champ valeur.

On distinguera le cas où la clé est le pointeur nul. En effet, nous faisons appel à la fonction 'strcpy' de la bibliothèque 'stdlib.h' afin de remplir le champ clé de la nouvelle cellule, qui possède un comportement indéterminé dans le cas où un de ses paramètres est un pointeur nul.

Si la clé passée en paramètre est le pointeur nul, on ne fait rien. Sinon, on alloue la mémoire pour le champ clé et on copie à l'aide de 'strcpy' la clé passée en paramètre dans le champ clé de la nouvelle cellule.

Une fois les champs clé et valeur remplis on vérifie si la liste est vide ou non. Si la liste est vide, la nouvelle cellule devient la première cellule de la liste, ainsi on renvoie son adresse. Sinon, on

affecte l'adresse pointée passée en paramètres au champ 'adresse de la cellule suivante' de la nouvelle cellule puis on retourne l'adresse de la nouvelle cellule.

1.2. Tests et affichage

Pour les listes, on possède des tests exhaustifs qui après exécution nous donne l'affichage console suivant.

```
Is the list NULL (0=NO, 1=YES)? 1
Print empty list:[]
[]
Test findKeyList with empty list:
Key two not found

Test delKeyInList with empty list:
Print empty list after deletion:[]
Test add function:
[1]
[(one,1)]
[2,1]
[(two,2),(one,1)]
[3,2,1]
[(three,3),(two,2),(one,1)]
[0,3,2,1]
[(NULL,0),(three,3),(two,2),(one,1)]
[4,0,3,2,1]
[(four,4),(NULL,0),(three,3),(two,2),(one,1)]
Found key two with value 2
Key twelve not found
Found key (null) with value 0
Test delete function:NULL: [(four,4),(three,3),(two,2),(one,1)]
Test delete function:two: [(four,4),(three,3),(one,1)]
five: [(four,4),(three,3),(one,1)]
four: [(three,3),(one,1)]
one: [(three,3)]
three: []
three: []
```

Figure 4 Tests exhaustifs pour la structure de données List

Tests units

Étant donné que les listes sont fondamentales pour l'implémentation d'autres structures de données (graphes, tables de hachage, pile et file), il nous est fourni plusieurs fichiers de tests unitaires élaborés par Loïck Lhote. Ces tests unitaires couvrent un certain nombre de cas vérifiant que toutes les méthodes fonctionnent et examinent la bonne gestion mémoire.

```
Summary of the unit tests
-----
Summary: 2 successful tests over 2 tests for newList().
Summary: 5 successful tests over 5 tests for freeList().
Summary: 20 successful tests over 20 tests for printList().
Summary: 8 successful tests over 8 tests for findKeyInList().
Summary: 9 successful tests over 9 tests for delKeyInList().
Summary: 7 successful tests over 7 tests for addKeyValueInList().
```

Figure 5 Tests units List

2. Table de hachage

La structure de donnée table de hachage (*hashtable*) est largement utilisée en informatique pour stocker des couples (clé, valeur) de manière efficace et permettre un accès rapide aux données.

L'implémentation de la table de hachage repose sur le principe de fonction de hachage *MurmurHash*, une fonction reconnue pour sa robustesse et ses bonnes performances en termes de distribution de valeurs hachées.

Une table de hachage sera définie ici comme une structure (*struct hashtable*) contenant la taille de la table, le nombre de couples stockés et un tableau de liste (*List**), où chacune des listes peuvent stocker des couples.

hashtable
<ul style="list-style-type: none"> - sizeTable : size_t - numberOfPairs : size_t - table : List*

Figure 6 Structure représentant la notion de table de hachage

Par la suite on définira *Hashtable* comme un alias de *hashtable* et on doublera la taille de notre table de hachage lorsque le nombre d'éléments de celle-ci dépasse la taille du tableau.

Voici une illustration d'une table de hachage.

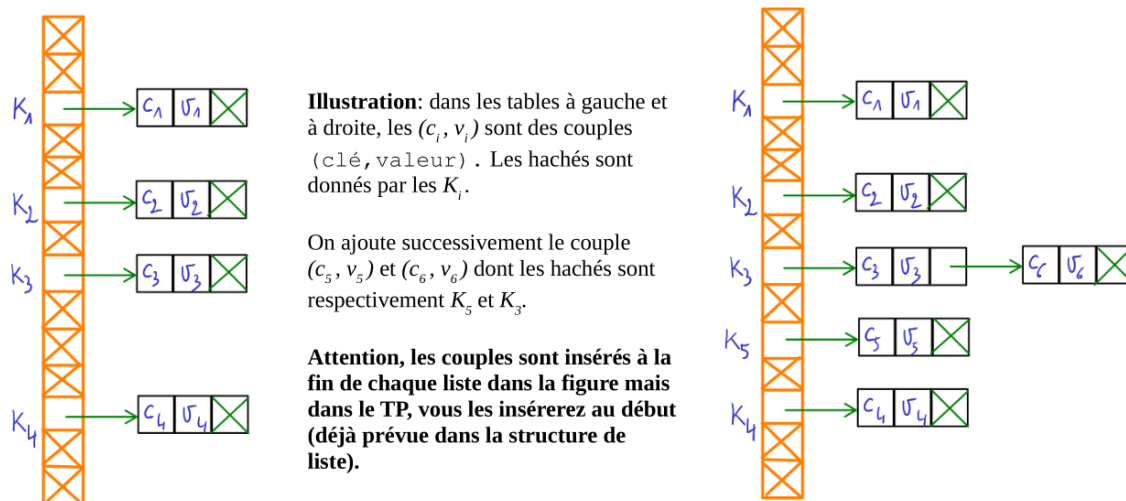


Figure 7 Exemple d'une table de hachage

On va implémenter un ensemble de fonctions permettant de **créer**, **détruire**, **insérer**, **rechercher**, **supprimer** et **imprimer** les données stockées dans la table de hachage. Ces fonctions offrent une interface conviviale pour interagir la structure de données, facilitant ainsi son utilisation.

2.1. Fonctions

2.1.1. Génération de hachés

La fonction de hachage non cryptographique MurmurHash nous est fournie. Elle prend en paramètre une chaîne de caractère représentant notre clé, un entier représentant la longueur de la clé ainsi qu'un entier max**Value**. Cette fonction retourne un entier compris entre 0 inclus et max**Value** exclus qui est le haché de la clé rentrée en paramètre.

2.1.2. Affichage d'une table de hachage donnée

La fonction '*hashtablePrint*' prend en paramètre une table de hachage et procède à un affichage console de la taille de cette table, du nombre de couples qu'elle contient ainsi que la table en elle-même.

Pour afficher le contenu de cette table, on va simplement appeler '*printList*' pour chaque liste de la table.

2.1.3. Création d'une table de hachage

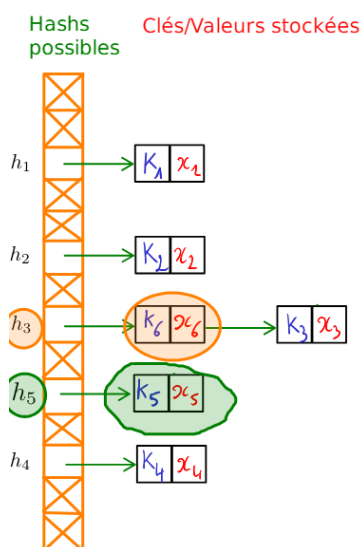
La fonction '*hashtableCreate*' instance une table de hachage de taille *n* passé en paramètre de la fonction. Elle initialise les champs de la table et retourne l'instance.

2.1.4. Insertion d'un couple (clé, valeur) dans une table sans la redimensionnée

La fonction 'hashtableInsertWithtoutResizing' insère un couple (clé, valeur) dans la table de hachage sans la redimensionnée. Si la clé est déjà présente dans la table de hachage, on remplace sa valeur associée par la nouvelle.

Cette fonction prend en paramètre un pointeur de table de hachage sur la table de hachage où l'on veut insérer le couple, la clé pour le nouveau couple ainsi que la valeur pour le nouveau couple. La fonction est de type void et ne retourne donc rien.

Dans un premier temps, on gère le cas où le pointeur passé en paramètre est nul. Ensuite on calcule le haché de la clé passé en paramètre. On appelle maintenant la fonction 'findKeyInList' sur la liste placée à la haché-ième position de la table de hachage. Si la clé n'est pas présente dans cette liste, on ajoute le nouveau couple à l'aide de 'addKeyValueInList' et on met à jour le nombre d'éléments de la table. Sinon, on met simplement à jour la valeur de la cellule qui contient la clé.



$h = \text{hash}(K) \leadsto$ retourne un entier appelé le hash de la clé K
Ajout de (K_6, v_6) :
 $\text{hash}(K_6) = h_3$
 \hookrightarrow différente de h_1, \dots, h_4
 \Rightarrow occupation d'une nouvelle case + ajout d'une liste
Ajout de (K_6, v_6) :
 $\text{hash}(K_6) = h_3$
 \Rightarrow ajout à la liste existante associée à $\text{hash}(K_6) = h_3$

Figure 8 Insertion d'un couple dans une table de hachage

2.1.5. Destruction d'une table

La fonction 'hashtableDestroy' libère la mémoire du tableau de listes de la table de hachage et mets les champs *sizeTable* et *numberOfPairs* à 0 ainsi que le champ du tableau de listes à nul. En revanche, la mémoire allouée par la structure en elle-même n'est pas libérée.

La fonction est de type void et prend en paramètre un pointeur pointant sur la table de hachage à détruire.

2.1.6. Redimension d'une table de hachage donnée

La fonction *'hashtableDoubleSize'* retourne une table de hachage deux fois plus grande que la table de hachage passée en paramètre et transfère toutes les paires clé-valeur de celle-ci dans la nouvelle table de hachage.

Cette opération est nécessaire lorsque la table devient trop chargée pour éviter les collisions excessives et maintenir de bonnes performances.

2.1.7. Insertion d'un couple (clé, valeur) dans une table avec redimension si nécessaire

La fonction *'hashtableInsert'* insère un couple (clé, valeur) dans la table de hachage avec redimension si nécessaire. Si la clé est déjà présente dans la table de hachage, on remplace sa valeur associée par la nouvelle.

Cette fonction prend en paramètre un pointeur de table de hachage sur la table de hachage où l'on veut insérer le couple, la clé pour le nouveau couple ainsi que la valeur pour le nouveau couple. La fonction est de type void et ne retourne donc rien.

On appelle simplement la fonction *'hashtableInsertWithtoutResizing'* puis si le nombre de couple (clé, valeur) est supérieur à la taille du tableau, on instance une nouvelle table de hachage en appelant *'hashtableDoubleSize'* et on détruit l'ancienne table à l'aide de la fonction *'hashtableDestroy'*.

2.1.8. Recherche d'une valeur par clé donnée dans la table

La fonction *'hashtableHaskey'* permet de vérifier la présence d'une clé donnée dans une table de hachage. Elle prend en paramètre la table de hachage à inspecter, ainsi qu'une chaîne de caractère représentant la clé à rechercher. De type *'int'*, cette fonction renvoie la valeur 1 si la clé est présente, et 0 dans le cas contraire.

Le processus de vérification consiste d'abord à calculer le haché de la clé fournie à l'aide de la fonction *'murmurhash'*. Ensuite, une recherche par clé est effectuée dans la liste associée à l'indice haché, qui correspond à la liste à la haché-ème position dans le tableau représentant la table de hachage. Cette recherche est réalisée grâce à la fonction *'findKeyInList'*.

2.1.9. Suppression d'un couple (clé, valeur) par clé donnée

La fonction *'hashtableRemove'* supprime par clé donnée le couple (clé, valeur) associé d'une liste. Elle prend en paramètre la table de hachage dans laquelle on effectue la suppression et la clé qui correspond au couple à supprimer. Cette fonction renvoie 0 si la clé n'est pas présente dans la table, 1 sinon.

On calcule le haché de la clé passée en paramètre, puis on appelle *'findKeyInList'* pour la haché-ème liste du tableau. Si la clé est présente, on supprime le couple à l'aide de *'delKeyInList'* et on retourne 1. Sinon, on ne fait rien et on retourne 0.

2.2. Exemple

La fonction *'countDistinctWordsInBook'* illustre l'utilisation pratique d'une table de hachage pour analyser un texte contenu dans un fichier. Cette fonction a pour objectif de compter le nombre de mots dans le fichier ainsi que le nombre de mots uniques tout en affichant le nombre d'occurrences de chaque mot unique.

Cette approche permet une analyse rapide et efficace du texte, en fournissant des informations précieuses sur la fréquence d'apparition des mots et la diversité du vocabulaire utilisé.

```

[idly,1),(six,15)]
[(hanging,13)]
[(alarming,1)]
[(impossible,4)]
[(wondered,14)]
[(blin,1)]
[(applause,2)]
[(keys,14)]
[(heartstrings,1)]
[(stantly,1),(sundays,1)]
[(wing,9)]
[(brilliantly,1)]
[(normally,3)]
[(buying,3)]
[(bang,1),(curtain,1)]
[(distant,4),(indeed,10),(stuck,9)]
[(nodded,10),(types,1)]
[(gathering,2)]
[(sniff,3),(trances,1)]
[(sports,2),(understand,16),(fallen,6)]
[(yorkshire,2),(nasty,14)]
[(ta,2)]
[(insultin,1),(dangerous,8),(jackets,2)]
[(miserable,2),(stunned,6)]
[(rid,6)]
[(risks,1),(knuts,6)]
[(improvement,1)]
[(serpent,2)]
[(streaked,5)]
[(Fluffy,27),(courage,6)]
[(seen,16),(classrooms,2),(protected,2)]
[(merry,2)]
[(alberic,1)]
[(torture,1),(switching,1),(wilder,2),(cheers,6)]
[(flickered,2)]
[(visitors,1),(memory,2)]
[(shivering,1),(sacked,2)]
[(realised,17),(awake,8)]
[(politer,1)]
[(deserted,1)]
[(rang,6),(said,791)]
[(excuses,1),(lop,1)]
[(flourish,1)]
[(barking,2),(blubber,1)]
[(straw,1)]
[(existence,1)]
[(not,238)]
[(grindelwald,2)]
[(proving,2)]
[(second,44)]
Nombre total de mots : 81669 et Nombre total de mots uniques : 5886
---- Fin Test countWordsInBook ----

```

Figure 9 Affichage CountDistinctWordsInBook

3. Piles

Les piles, souvent utilisées en informatique et dans de nombreux domaines, sont une structure de données fondamentale. Elles suivent le principe de Last-In, First-Out (LIFO), ce qui signifie que le dernier élément ajouté à la pile sera le premier à en sortir, similaire à une pile d'assiettes où vous ajoutez et retirez des assiettes du dessus.

On va ici représenter une pile par une structure de liste dans laquelle tous les champs key de toutes les cellules seront à NULL. On aura donc accès à la tête de la liste en temps constant. On ajoutera et supprimera les éléments en tête de liste.

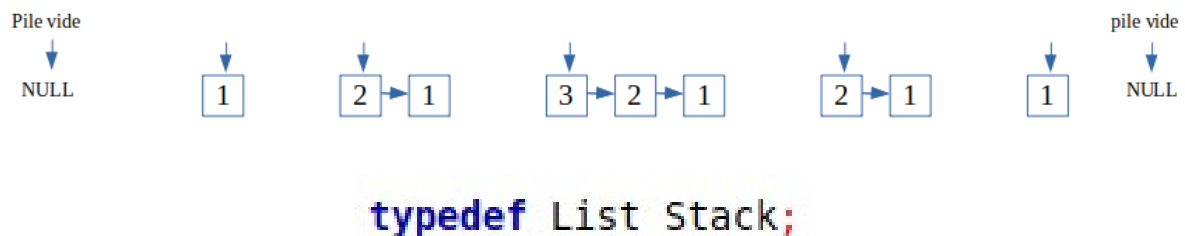


Figure 10 Structure de données Pile et fonctionnement

3.1. Fonctions

3.1.1. Création d'une pile

La fonction `'createStack'` est chargée de créer une nouvelle pile vide. Elle alloue dynamiquement la mémoire pour la structure de la pile et initialise la pile en créant une nouvelle liste vide. Elle renvoie un pointeur vers la nouvelle pile créée.

3.1.2. Ajout d'un nouvel élément au sommet de la pile

La fonction `'push'` ajoute un nouvel élément au sommet de la pile. Elle prend en argument un pointeur sur la pile à laquelle ajouter l'élément et la valeur de l'élément à ajouter. Pour cela, elle utilise la fonction `'addKeyValueInList'` pour ajouter la valeur à la pile pointée.

Ajout de 8 :

Pushing Value: 8

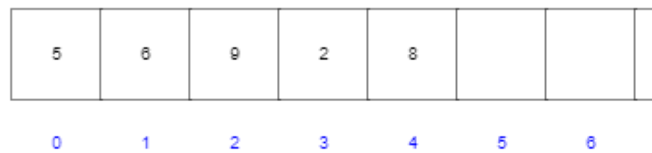
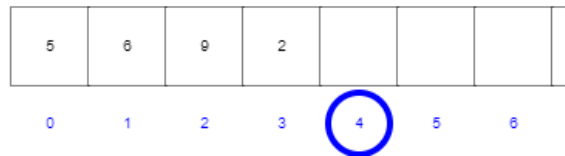


Figure 11 Empiler une valeur

3.1.3. Suppression de l'élément au sommet de la pile

La fonction 'pop' supprime l'élément situé au sommet de la pile et renvoie sa valeur. Elle prend en paramètre un pointeur vers la pile à partir de laquelle l'élément doit être supprimé. Elle vérifie d'abord si la pile est vide, puis elle extrait la valeur de l'élément supprimé et libère la mémoire associée.

Popped Value: 8

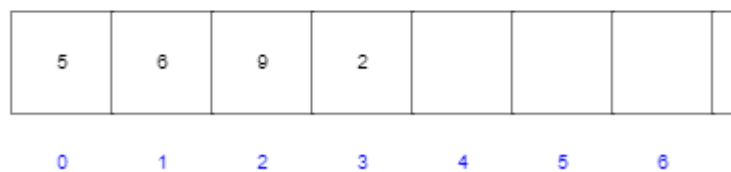


Figure 12 Dépiler le haut de d'une pile

3.1.4. Consultation de la valeur du sommet de la pile

La fonction 'peek' permet de consulter la valeur de l'élément situé au sommet de la pile sans le supprimer. Elle renvoie simplement la valeur de cet élément sans modifier la pile.

3.1.5. Vérification de l'état de la pile

La fonction 'isStackEmpty' vérifie si la pile est vide ou non. Si la pile est vide, elle renvoie 1, sinon elle renvoie 0.

3.2. Affichage de la pile

La fonction 'stackPrint' affiche le contenu de la pile sur la sortie standard. Elle utilise la fonction 'printList' pour imprimer les éléments de la pile.

```
[ ]
Add 1 to stack:
[1]
Add 2 to stack:
[2,1]
Add 3 to stack:
[3,2,1]
popped value: 3
Updated stack:
[2,1]
popped value: 2
Updated stack:
[1]
add 4 to stack:
[4,1]
Top value of the stack: 4
[4,1]
Is stack empty? 0

Final stack:
[ ]
Is stack empty? 1
```

Figure 13 Tests exhaustifs pour la structure de donnée Pile

4. Files

La file est une structure de donnée de type **FIFO : first in, first out**. Le premier élément à entrer dans la file sera le dernier élément à sortir. On va ici représenter la file par une structure de liste à laquelle on ajoute un pointeur vers la queue de la liste. Ainsi on accède en temps constant à la tête et à la queue de la liste.

Ici, on ajoutera les éléments en queue de liste et supprimera les éléments en tête de liste.

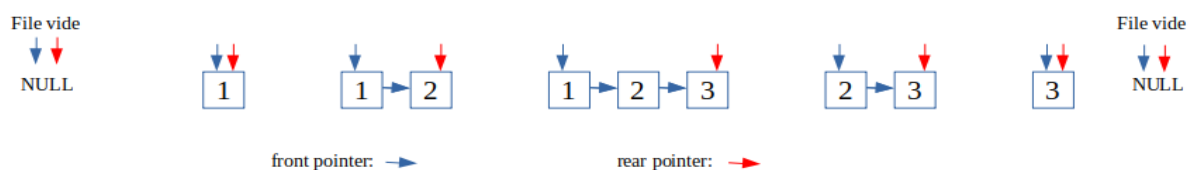


Figure 14 Illustration et fonctionnement d'une file

```

typedef struct queue {
    Cell* front; /** A pointer to the front (first) element in the queue. */
    Cell* rear;  /** A pointer to the rear (last) element in the queue. */
} Queue;

```

Figure 15 Structure de file en langage C

4.1. Fonctions

4.1.1. Création d'une nouvelle file vide

La fonction 'createQueue' alloue dynamiquement la mémoire de la nouvelle file créée. On initialise les champs 'front' et 'rear' à NULL puis on retourne le pointeur sur cette nouvelle file.



Figure 16 File vide

4.1.2. Vérification de la file

La fonction '*isQueueEmpty*' renvoie 1 si la file est vide et 0 dans le cas contraire. Elle prend en argument la file à vérifier.

4.1.3. Ajout d'un élément dans une file

La fonction '*enqueue*' ajoute un élément **en queue de file**. Elle prend en paramètre un pointeur sur la file dans laquelle on procède à l'ajout. La fonction ne retourne rien.

Dans un premier temps, on gère le cas où la file serait vide. Ensuite on alloue dynamiquement la mémoire pour la nouvelle cellule afin de stocker l'élément ajouter. On distinguera le cas si la file est vide ou non à l'aide de la fonction '*isQueueEmpty*'. Si elle est vide, on fait pointer '*front*' et '*rear*' sur la nouvelle cellule. Sinon, on fait pointer '*rear*' et le pointeur '*nextCell*' de la cellule située en queue de file vers la nouvelle cellule.

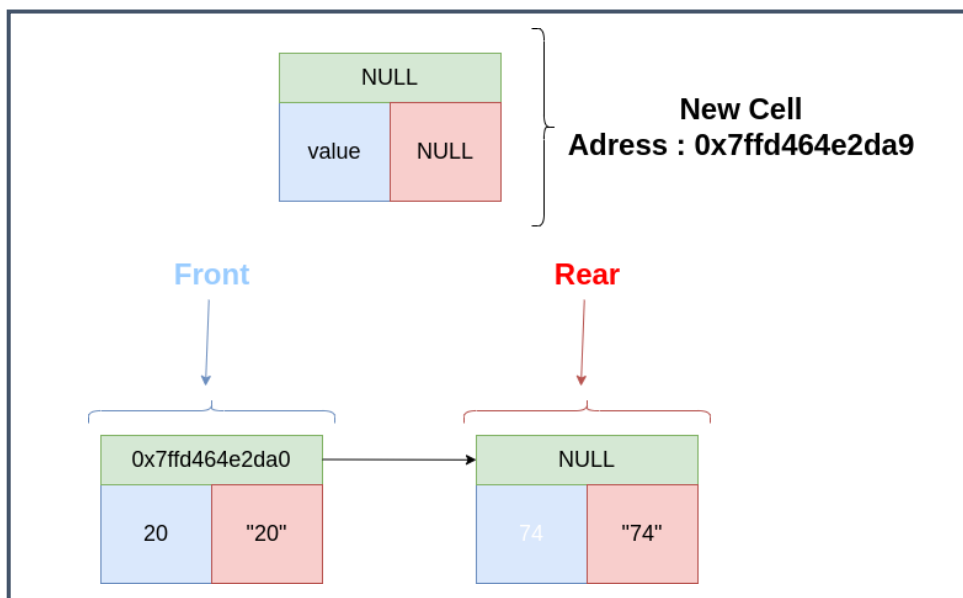


Figure 17 Insertion en queue de file 1/2

Après insertion :

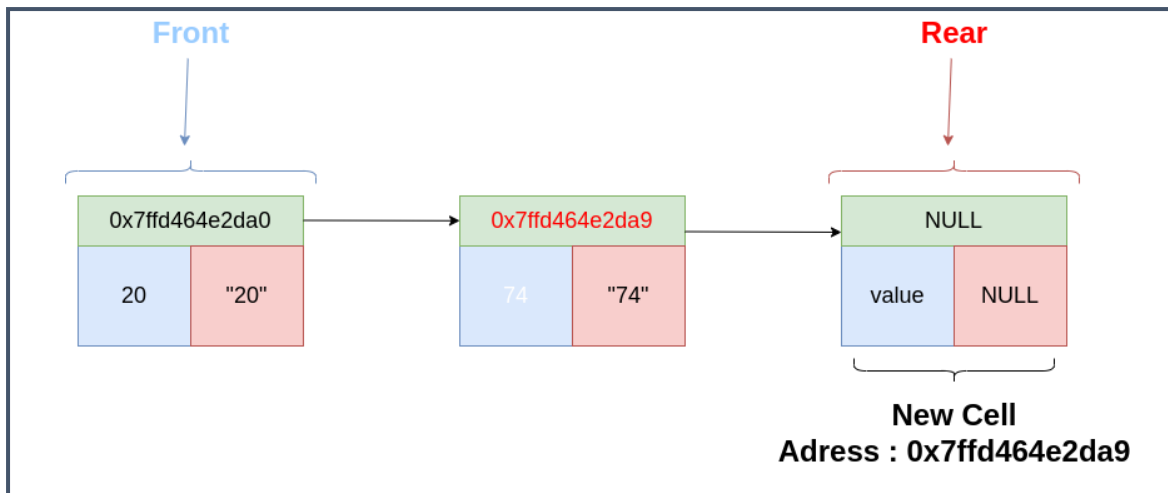


Figure 18 Insertion en queue de file 2/2

4.1.4. Suppression de la tête de file

La fonction *'dequeue'* est chargée de retirer l'élément situé en tête de file. Elle prend en paramètre un pointeur vers la file où cette opération sera effectuée. Cette fonction renvoie l'élément retiré de la file.

Dans un premier temps, la fonction traite le cas où la file pointée est vide ou où le pointeur serait NULL, assurant ainsi une gestion appropriée des erreurs.

Ensuite, si la file contient uniquement un élément, la fonction stocke la valeur de cet élément, libère la mémoire allouée pour la cellule associée, puis réinitialise les pointeurs *'rear'* et *'front'* à NULL pour indiquer que la file est désormais vide. Elle retourne ensuite la valeur de l'élément retiré.

Dans le cas où la file contient plus d'un élément, la fonction stocke la valeur de l'élément en tête de file, libère la mémoire associée à la cellule concernée. Ensuite, elle réajuste le pointeur *'front'* pour pointer vers la deuxième cellule de la file. Enfin, elle retourne la valeur de l'élément retiré. Ainsi, la fonction garantit un maintien de la file correcte après le retrait de l'élément en tête.

4.1.5. Valeur de l'élément en tête de file

La fonction *'queueGetFrontValue'* extrait la valeur de l'élément en tête de la file passée en paramètre (queue q). Elle retourne un entier représentant la valeur de cet élément.

Tout d'abord, la fonction vérifie si la file est vide ou non afin de gérer les erreurs potentielles.

Si la file n'est pas vide, la fonction procède à l'extraction de la valeur de l'élément en tête. Comme la file suit le principe FIFO (premier entré, premier sorti), l'élément en tête est toujours situé à l'adresse pointée par *'q.front'*. Ainsi la fonction retourne simplement la valeur de l'élément en tête de la file, sans nécessiter d'autres opérations.

Cette fonction assure donc une opération simple et efficace pour obtenir la valeur de

4.2. Traces d'exécutions

```
[ ]
Add 1 to queue:
[1]
Add 2 to queue:
[1,2]
Add 3 to queue:
[1,2,3]
Dequeued value: 1
Updated queue:
[2,3]
Dequeued value: 2
Updated queue:
[3]
Front value: 3
add 4 to queue:
[3,4]
Is queue empty? 0
Final queue:
[ ]
Is queue empty? 1
```

Figure 19 Tests exhaustifs File

5. Arbres binaires de recherche (ABR)

Les arbres binaires de recherche (ABR) constituent l'une des structures de données les plus fondamentales en informatique. Leur organisation hiérarchique et leur capacité à maintenir un ordre spécifique permettent une manipulation efficace des données, que ce soit pour le stockage, la recherche ou la récupération. Dans ce travail pratique nous allons nous pencher sur l'implémentation et l'utilisation des ABR à travers différentes fonctions telles que **l'insertion**, la **suppression** et la **recherche de valeurs** dans ces structures.

Avant de plonger dans les détails des fonctions et des opérations sur les ABR, il est crucial de comprendre leur concept fondamental. Un ABR est une structure de données arborescente dans laquelle chaque nœud possède au plus deux enfants : un nœud gauche et un nœud droit. De plus, pour chaque nœud, toutes les valeurs dans le sous-arbre gauche sont inférieures à la valeur du nœud, tandis que toutes les valeurs dans le sous-arbre droit sont supérieures à la valeur du nœud. Cette organisation permet des opérations de recherche efficaces, souvent en temps logarithmique en le nombre total de nœuds dans l'arbre.

Dans ce contexte, nous allons explorer les différentes fonctions implémentées pour manipuler les ABR, telles que la **création d'un arbre vide**, l'**ajout de valeurs**, la **recherche de valeurs spécifiques**, le **calcul de la hauteur de l'arbre**, ainsi que la **suppression de valeurs** et la **libération de la mémoire** associée à un ABR. En comprenant ces opérations et en analysant leurs performances, nous pourrions tirer des conclusions précieuses sur l'utilité et l'efficacité des ABR dans une variété d'applications informatiques.

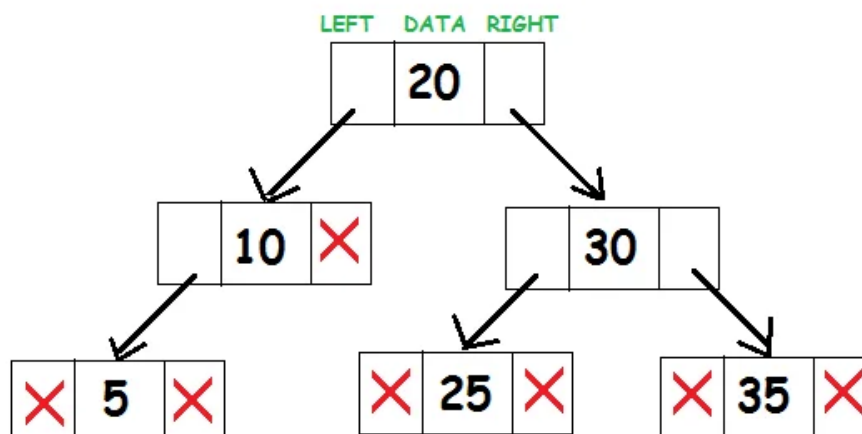


Figure 20 Exemple d'un ABR

5.1. Définition de la structure

NodeBST	<code>typedef NodeBST* BinarySearchTree;</code>
<ul style="list-style-type: none">- value : int- leftBST : NodeBST*- rightBST : NodeBST*	

Figure 21 Structure représentant un nœud d'un ABR et définition d'un ABR

5.2. Fonctions

5.2.1. Création d'un arbre binaire de recherche vide

La fonction '*createEmptyBST*' est chargée de créer un nouvel arbre binaire de recherche vide. Elle retourne simplement un pointeur vers la racine de cet arbre, initialisé à NULL.

5.2.2. Libération de la mémoire d'un ABR

La fonction '*freeBST*' a pour objectif de libérer la mémoire occupée par un arbre binaire de recherche donné. Elle prend en paramètre un pointeur vers la racine de l'arbre à libérer (*BinarySearchTree tree*). La première étape de la fonction consiste à vérifier si l'arbre est vide, c'est-à-dire si le pointeur vers la racine est nul. Si tel est le cas, la fonction se termine immédiatement sans effectuer aucune opération, car il n'y a pas de mémoire à libérer pour un arbre vide.

En revanche, si l'arbre n'est pas vide, la fonction procède à la **libération récursive de la mémoire** pour les sous-arbres gauche et droit de la racine. Pour cela, elle appelle récursivement la fonction '*freeBST*' sur les sous-arbres gauche et droit, garantissant ainsi que toute la mémoire associée à ces sous-arbres soit libérée.

Une fois que la mémoire pour les sous-arbres a été libérée, la fonction libère ensuite la mémoire pour la racine elle-même. Cela permet de libérer la mémoire allouée pour le nœud racine de l'arbre.

Enfin, la fonction se termine et retourne au programme appelant. Ainsi, la fonction '*freeBST*' assure une libération correcte et complète de la mémoire d'un arbre binaire de recherche, évitant tout risque de fuite mémoire.

5.2.3. Ajout d'une valeur dans un ABR

La fonction '*addToBST*' a pour rôle d'ajouter une valeur donnée à un arbre binaire de recherche. Elle prend en paramètres un pointeur vers la racine de l'arbre (*BinarySearchTree*

tree) ainsi que la valeur à ajouter (int value), puis retourne un pointeur vers la racine du nouvel arbre résultant.

Dans un premier temps, la fonction vérifie si l'arbre est vide. Si tel est le cas, la fonction alloue dynamiquement de la mémoire pour créer un nouveau nœud 'NodeBST', initialise sa valeur avec la valeur fournie, et ses sous-arbres gauche et droit à nul. Ensuite, elle retourne un pointeur vers ce nouveau nœud, qui devient ainsi la racine de l'arbre contenant uniquement cette valeur.

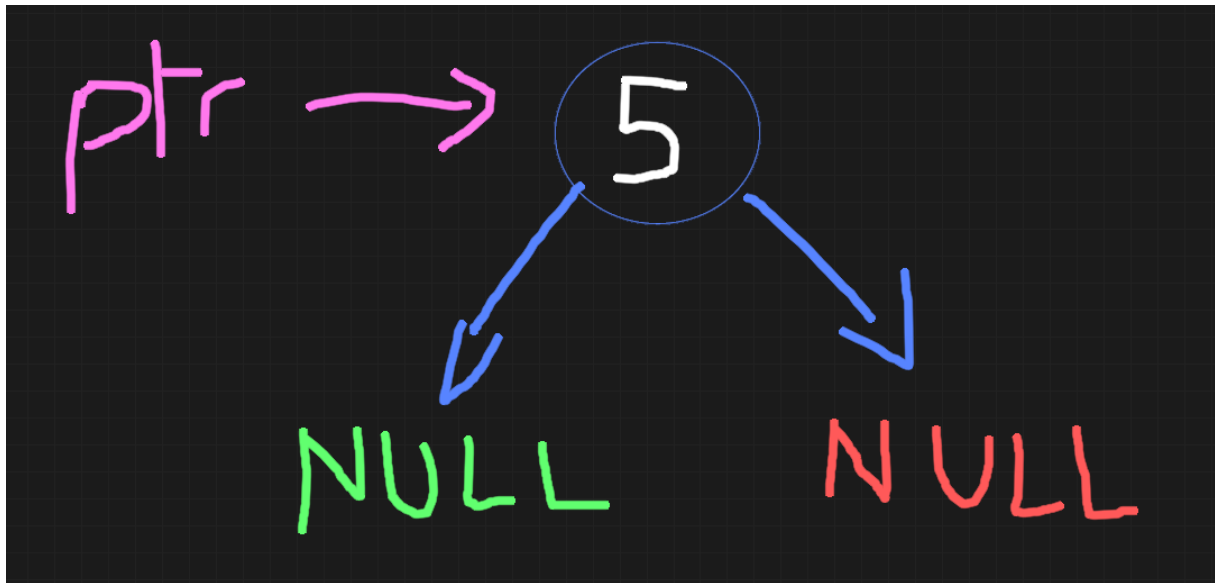
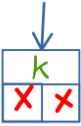


Figure 22 Insertion dans un ABR vide

Ensuite, si l'arbre n'est pas vide, la fonction compare la valeur à ajouter avec la valeur du nœud courant. Si la valeur à ajouter est supérieure à la valeur du nœud courant, le sous-arbre droit reçoit 'addToBST' appelé de manière réursive sur lui-même. De manière similaire la fonction effectue la même opération sur le sous-arbre gauche. Puis on retourne le nœud courant.

Finalement, la fonction retourne un pointeur vers la racine de l'arbre, qui n'a pas été modifiée si la valeur à ajouter était déjà présente dans l'arbre, ou bien qui a été mise à jour si la valeur a été ajoutée avec succès dans l'ABR.

```

ABR ajouter (clé K, ABR A)
    Si A est vide retourner 
    Si racine(A) < K alors
        | droite(A) ← ajouter (K, droite(A))
    Si racine(A) > K alors
        | gauche(A) ← ajouter (K, gauche(A))
    Retourner A

```

Figure 23 Algorithme d'insertion au sein d'un ABR

La complexité de l'ajout d'une valeur dans un ABR est proportionnelle à la hauteur de l'arbre, car le nombre d'opérations nécessaires pour insérer la valeur dépend du nombre d'étages à parcourir pour atteindre la feuille appropriée.

Complexité : $O(h)$

Figure 24 Complexité d'insertion dans un ABR

On remarque donc que dans le cas d'un arbre parfaitement équilibré, on sera en $O(\log n)$ avec n le nombre de nœuds de l'arbre.

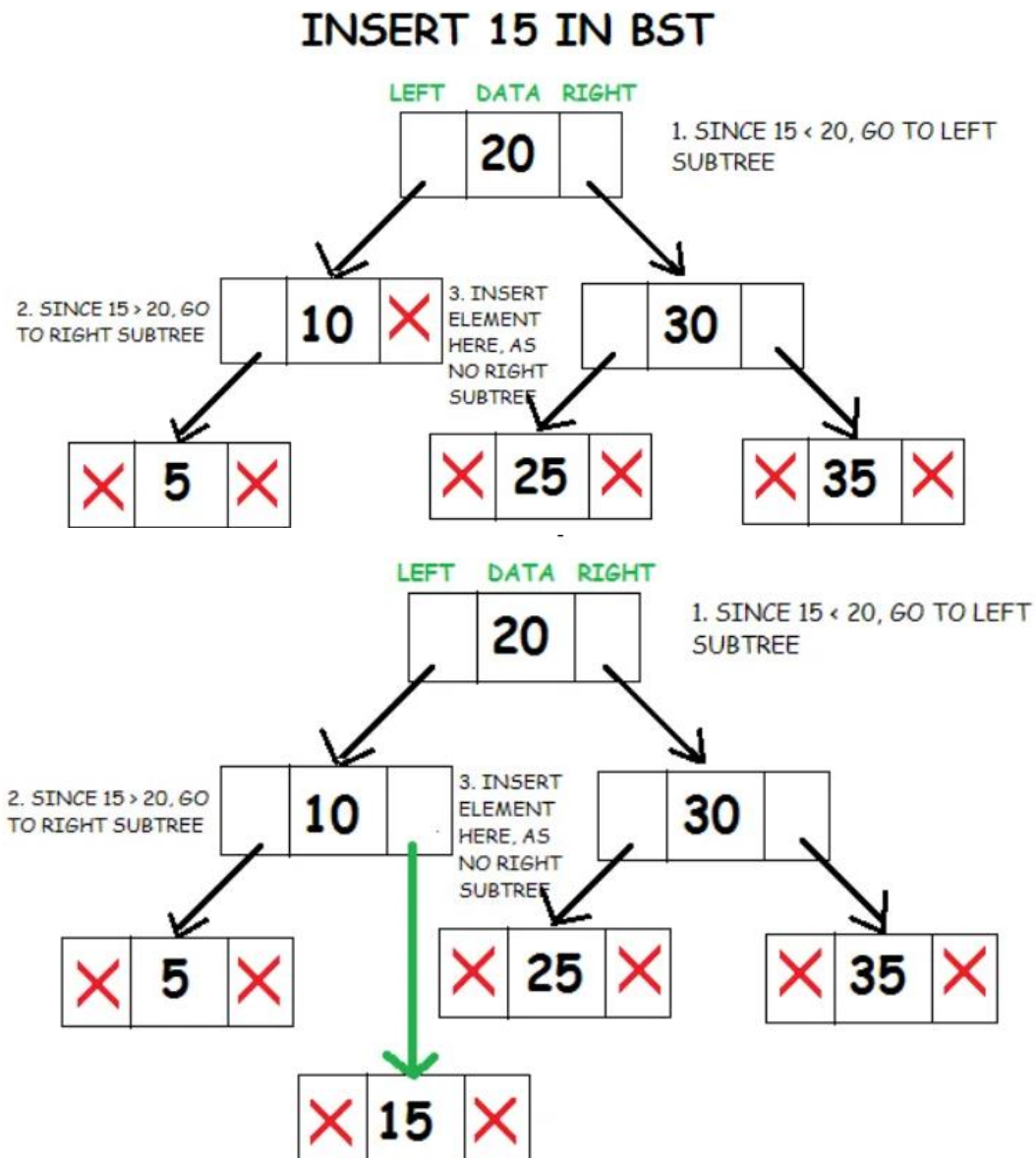


Figure 25 Exemple insertion ABR

5.2.4. Hauteur d'un ABR

La fonction '*heightBST*' est chargée de calculer la hauteur d'un arbre binaire de recherche donné. Elle prend en paramètre un pointeur vers la racine de l'arbre (*BinarySearchTree tree*) et retourne la hauteur de cet arbre.

Tout d'abord, la fonction gère le cas où le pointeur est nul. Si tel est le cas, on définit sa hauteur comme étant -1 (une convention pour représenter la hauteur d'un arbre vide).

Ensuite, si l'arbre n'est pas vide, la **fonction calcule récursivement la hauteur de ses sous-arbres** gauche et droit en s'appelant elle-même sur ses sous-arbres.

Ensuite on retourne 1 (hauteur du nœud courant) en ajoutant le maximum entre les hauteurs de ses sous-arbres.

Ainsi, grâce à cette approche récursive, la fonction explore de manière exhaustive tous les chemins de l'arbre, calculant la hauteur à chaque étape et retourne finalement la hauteur maximale entre les sous-arbre gauche et droit, plus un.

5.2.5. Trouver une valeur dans un ABR

La fonction '*searchBST*' est conçue pour rechercher une valeur donnée dans un ABR par une **méthode de recherche récursive**. Elle prend en paramètre un pointeur vers la racine de l'arbre (*BinarySearchTree tree*) et la valeur à rechercher (*int value*), et retourne l'adresse du nœud contenant la valeur (*NodeBST*) (NULL si la valeur n'est pas dans l'arbre).

Dans un premier temps, si le pointeur passé en paramètre est nul, on retourne une adresse nulle, indiquant que la valeur recherchée n'a pas été trouvée dans l'arbre.

Ensuite, la fonction vérifie si la valeur de la racine de l'arbre est égale à la valeur recherchée (value). Si c'est le cas, cela signifie que la valeur recherchée a été trouvée à la racine de l'arbre, et la fonction retourne un pointeur vers le nœud contenant cette valeur.

Si la valeur recherchée est supérieure à la valeur de la racine de l'arbre, la fonction continue la recherche dans le sous-arbre droit de la racine en appelant récursivement '*searchBST*' sur le sous-arbre droit.

De même, si la valeur de la racine de l'arbre est inférieure à la valeur recherchée, on effectue la même chose sur le sous-arbre gauche.

Ce processus de recherche récursive se poursuit jusqu'à ce que la valeur recherchée soit trouvée dans l'arbre ou jusqu'à ce que la fonction atteigne le bas de l'arbre où aucune recherche supplémentaire ne peut être effectuée. Dans ce dernier cas, on revient au cas où le pointeur passé en paramètre est nul et donc la fonction renvoie une adresse nulle indiquant que la valeur recherchée n'est pas présente dans l'arbre.

Cette approche récursive permet à la fonction de parcourir efficacement l'ABR grâce à l'organisation hiérarchique des valeurs dans ce dernier.

Algorithm 1: Search for a value in a BST

Input: A BST *tree* and an integer value *value*
Output: The node in the BST containing the value *value*, or NULL if the value is not found

```

if tree = NULL then
    | return NULL;
end
if tree → value = value then
    | return tree;
end
if value > tree → value then
    | return searchBST(tree → rightBST, value);
end
else
    | return searchBST(tree → leftBST, value);
end

```

Figure 26 Algorithme de recherche au sein d'un ABR

Complexité : $O(h)$

Figure 27 Complexité de la recherche au sein d'un ABR

5.2.6. Suppression de la racine d'un ABR

La fonction '*deleteRootBST*' est conçue pour supprimer le nœud racine d'un ABR tout en préservant la propriété de l'ABR. Elle prend en paramètre un pointeur sur la racine de l'arbre à supprimer.

Dans un premier temps, la fonction gère le cas où le pointeur est nul.

Ensuite la fonction vérifie si l'arbre ne contient que la racine, dans ce cas on retourne une adresse nulle (arbre désormais vide).

Si le nœud racine de l'arbre n'a qu'un seul sous-arbre, on retourne simplement l'adresse de son sous-arbre.

Si le nœud racine possède deux sous-arbre, la fonction recherche le nœud le plus à droite dans le sous-arbre gauche de la racine. Ce nœud possède la plus grande valeur du sous-arbre gauche et on peut donc changer la valeur du nœud racine tout en préservant les propriétés

d'un ABR. Une fois que le nœud le plus à droite est trouvé, sa valeur est copiée dans le nœud racine, puis on libère la mémoire allouée pour ce nœud tout en ajustant les liaisons dans l'arbre.

En résumé

La fonction '*deleteRootBST*' offre une méthode efficace pour supprimer le nœud racine d'un ABR tout en préservant les propriétés d'un ABR.

5.2.7. Suppression d'une valeur d'un ABR

La fonction '*deleteFromBST*' est une fonction conçue pour supprimer un nœud contenant une valeur spécifique tout en maintenant la bonne structure de l'ABR. Elle prend en paramètres un pointeur vers la racine de l'ABR (BinarySearchTree tree), une valeur entière (int value) et renvoie un pointeur sur la racine de l'arbre résultant

Dans un premier temps, si l'arbre est vide, on retourne une adresse nulle.

Ensuite si la valeur du nœud courant est égale à la valeur à supprimée, on renvoie l'adresse renvoyée par la fonction '*deleteFromBST*' appelée sur le nœud courant et on libère la mémoire associée au nœud courant.

Sinon, en fonction de la valeur du nœud courant on appelle récursivement '*deleteFromBST*' sur le sous-arbre droit ou gauche, affecte l'adresse retournée au sous-arbre et on retourne le nœud courant.

Cette approche récursive permet de parcourir efficacement l'arbre jusqu'à trouver (ou non) la valeur à supprimer et renvoie l'adresse de la racine de l'arbre résultant respectant les propriétés d'un ABR.

Algorithm 1: Delete a value from a BST

```
Input: A BST tree and an integer value value
Output: The resulting BST after deletion of the value value
if tree = NULL then
    | return NULL;
if tree → value = value then
    | BinarySearchTree tmp;
    | tmp = deleteRootBST(tree);
    | free(tree);
    | return tmp;
if value > tree → value then
    | tree → rightBST = deleteFromBST(tree → rightBST, value);
if value < tree → value then
    | tree → leftBST = deleteFromBST(tree → leftBST, value);
return tree;
```

Figure 28 Algorithme de suppression d'un nœud au sein d'un ABR

5.2.8. Construire un ABR à partir d'une permutation

La fonction '*buildBSTFromPermutation*' crée un ABR à partir d'une permutation donnée. Elle prend en argument un tableau d'entiers représentant la permutation et un entier représentant la taille du tableau.

5.2.9. Affichage d'un ABR

La fonction '*prettyPrintBST*' procède à un affichage console d'un ABR passé en paramètres.

5.3. Traces d'exécutions

Voici un affichage console de plusieurs tests exhaustifs.

```

          9
        8
      7
    6
  5
    4
  3
    2
  1
    root deleted
    *****
    *****
          9
        8
      7
    6
  5
    4
  3
    2
  1
    *****
    *****
    2 has been found !
    16 has not been found...
    9 has been found !
    *****
    The height of the tree is 3.
    *****
          9
        8
      7
    6
  5
    4
  3
    2
  1
    root deleted
  
```

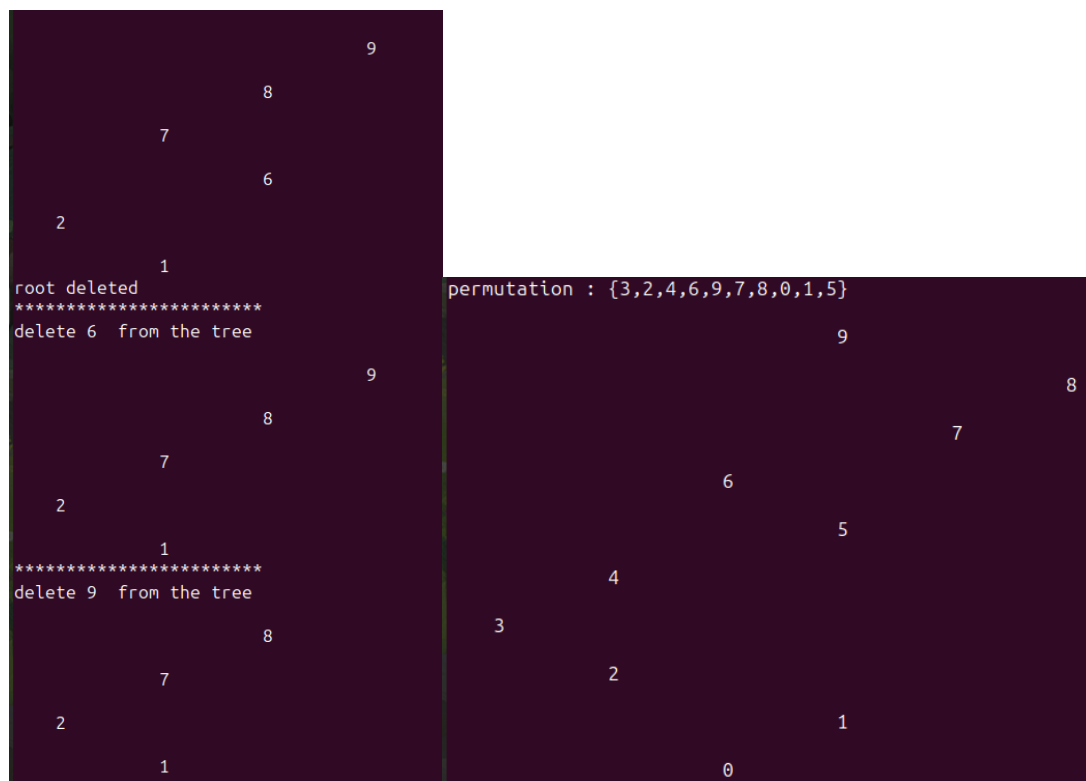



Figure 29 Tests exhaustifs ABR

5.4. Résultats et critiques

Le code fourni nous permet de mener un grand nombre d'expériences afin d'évaluer les performances des opérations sur les arbres de recherche binaire (ABR) en fonction de la distribution des données. Les résultats expérimentaux obtenus révèlent des différences importantes dans ces performances, notamment en ce qui concerne le temps de construction des arbres, le temps nécessaire pour effectuer des recherches et la hauteur des arbres.

Question : Quelles conclusions pouvons-nous tirer des résultats d'expériences que nous obtenons ?

Réponse :

En réalisant 1000 tests sur des permutations de données uniformes et non uniformes de taille 1000, nous constatons que, en moyenne, les temps de construction et de recherche, ainsi que les hauteurs des arbres sont notablement plus élevés dans le cas de distributions non uniformes.

Ici la distribution non uniforme sera une distribution uniforme de taille n telle que les n/10 premiers éléments seront triés.

```

Comparison between the data structures
size of the permutations: 1000
number of tests: 1000
Binary search tree with uniform distribution:
-> The average time to build is : 87.558000
-> The average height is : 21.106000
-> The average time to perform searches is : 85.197000
Binary search tree with non uniform distribution:
-> The average time to build is : 216.967000
-> The average height is : 105.941000
-> The average time to perform searches is : 225.080000
beginning graph....

```

Figure 30 Résultats sur distributions uniformes et non uniformes pour ABR

Ensuite, nous procédons à plusieurs expériences sur des échantillons de différentes tailles dans le but de tracer des graphiques afin d'étudier le comportement temporel des opérations de construction et de recherche ainsi que l'évolution de la hauteur des arbres en fonction de leur taille.

Premièrement, on observe que le temps requis pour construire un ABR en fonction de sa taille, dans le cas d'une distribution non uniforme, suit une tendance croissante avec une augmentation significative du temps de construction à mesure que la taille de l'arbre augmente. Tandis que pour une distribution uniforme, le temps de construction semble contrôlé logarithmiquement en la taille de l'arbre. Ceci est cohérent car comme mentionné dans le 5.2.3, la complexité de l'ajout d'un élément dans un arbre est en $O(\text{hauteur})$. **Dans le cas d'une distribution uniforme, l'arbre est généralement équilibré**, et a donc une **hauteur égale à $\log(n)$** (n la taille de l'arbre), et **dans le cas d'une distribution non uniforme**, l'arbre sera probablement déséquilibré et aura une hauteur plus grande que celle prévue dans le cas d'une distribution uniforme. **L'arbre aura une structure de liste chaînée de hauteur linéaire en sa taille.**

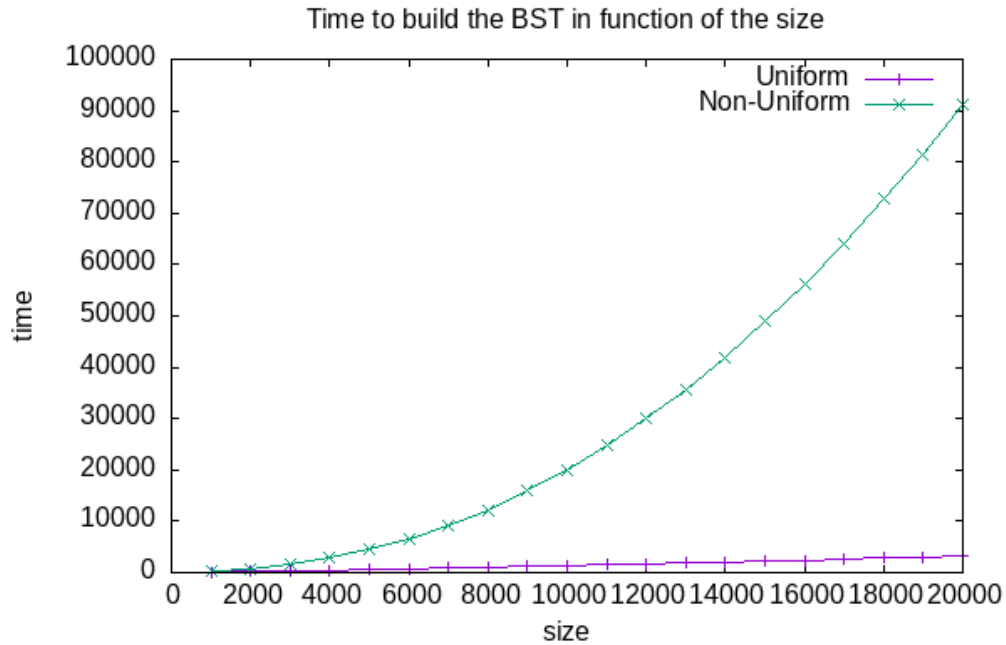


Figure 31 Temps de construction d'un ABR en fonction de sa taille (cas uniforme et non uniforme)

Deuxièmement, on observe que la hauteur des arbres dans le cas d'une distribution uniforme est en $\log(n)$ car les arbres sont dans ce cas, probablement bien équilibrés. Dans le cas d'une distribution non uniforme, **la hauteur des arbres est linéaire en leur tailles** (de coefficient directeur 1/10). Ceci est **cohérent** car les $n/10$ premiers éléments insérés sont triés c'est-à-dire que les arbres seront au moins de taille $n/10$.

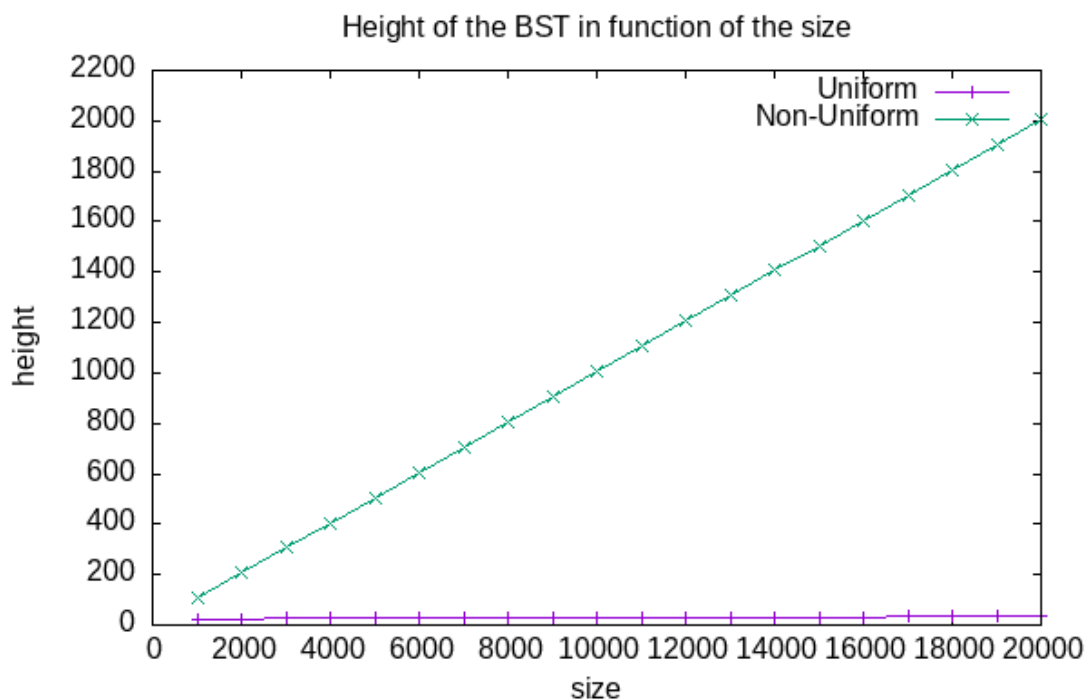


Figure 32 Hauteur d'un ABR en fonction de sa taille (cas uniforme et non uniforme)

Enfin, concernant le temps pour effectuer 1000 recherches en fonction de la taille de l'arbre, on observe un comportement linéaire en la taille de l'arbre dans le cas d'une distribution non uniforme et un comportement logarithmique en la taille de l'arbre dans le cas d'une distribution uniforme. **En effet, les opérations nécessaires à la recherche dépendent de la hauteur de l'arbre.**

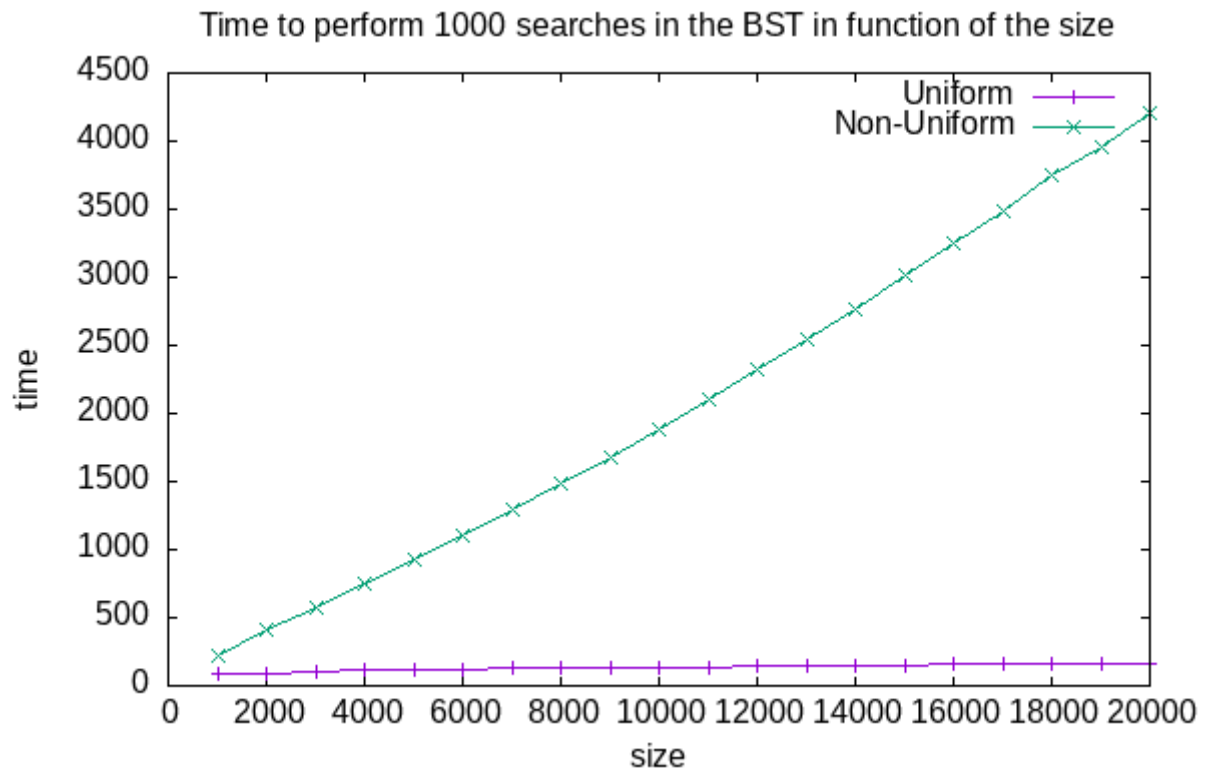


Figure 33 Temps pour effectuer 1000 recherches dans un ABR en fonction de sa taille (cas uniforme et non uniforme)

En conclusion, la hauteur de l'arbre est directement impactée par le type de distribution des nœuds à insérer. De plus, étant donné que la création et la recherche dépendent de la hauteur de l'arbre, on explique bien ces différences de performances entre une distribution uniforme et non uniforme.

6. Arbres binaires de recherche randomisés (RBST)

Les arbres binaires de recherche randomisés (RBST) représentent une structure de données importante dans le domaine de l'informatique. Cette structure de données s'inscrit dans la continuité des arbres binaires de recherche classiques en **introduisant un élément de randomisation**. Cela influence la construction et la manipulation de l'arbre de manière **non déterministe**. Dans ce travail pratique nous allons se pencher sur l'implémentation et l'utilisation des RBST à travers différentes fonctions clés telles que la création d'un arbre vide, l'insertion, la recherche de valeurs dans ces structures ainsi que les calculs de la taille et de la hauteur de l'arbre. Avant de plonger dans les détails des fonctions et des opérations, il est impératif de comprendre leur concept fondamental. **Un RBST possède la même structure d'un ABR où chaque nœud possède en plus sa taille**. Ce qui les distinguent par rapport à un ABR, c'est leur capacité à **introduire de l'aléatoire dans le processus de construction**, modifiant ainsi la disposition des nœuds dans l'arbre de manière non déterministe. En comprenant ces opérations et en analysant leurs performances, nous pourrions tirer des conclusions sur le caractère aléatoire de cette structure de données.

6.1. Définition de la structure

On représentera un RBST de la façon suivante :

NodeRBST	<code>typedef NodeRBST* RBinarySearchTree;</code>
<ul style="list-style-type: none">- value : int- leftBST : NodeBST*- rightBST : NodeBST*- size : int	

Figure 34 Définition de la structure d'un RBST

6.2. Fonctions

Nous ne procéderons pas à une analyse détaillée des fonctions telles que *'createEmptyRBST'*, *'freeRBST'*, *'heightRBST'*, *'searchRBST'* et *'prettyPrintRBST'* dans cette section, car elles suivent la même logique que celles utilisées pour les arbres binaires de recherche (ABR) (5.2).

6.2.1. Taille d'un RBST

On implémente simplement cet algorithme.

Algorithm 1: *sizeOfRBST*

Input: An *RBinarySearchTree tree*
Output: The number of nodes in the *RBinarySearchTree tree*
if *tree = NULL* **then**
 | **return** 0;
end
return 1 + *sizeOfRBST(tree → leftRBST)* +
 sizeOfRBST(tree → rightRBST);

Figure 35 Algorithme taille d'un RBST

6.2.2. Couper un RBST en fonction d'une valeur donnée

La fonction *'splitRBST'* a pour objectif de diviser un RBST en deux parties distinctes en fonction d'une valeur donnée. Cette fonction prend en entrée un arbre (*RBinarySearchTree tree*) et une valeur (*int value*) ainsi que deux pointeurs vers des arbres qui stockeront respectivement les parties inférieures (*RBinarySearchTree *inf*) et supérieures (*RBinarySearchTree *sup*) de l'arbre initial après division.

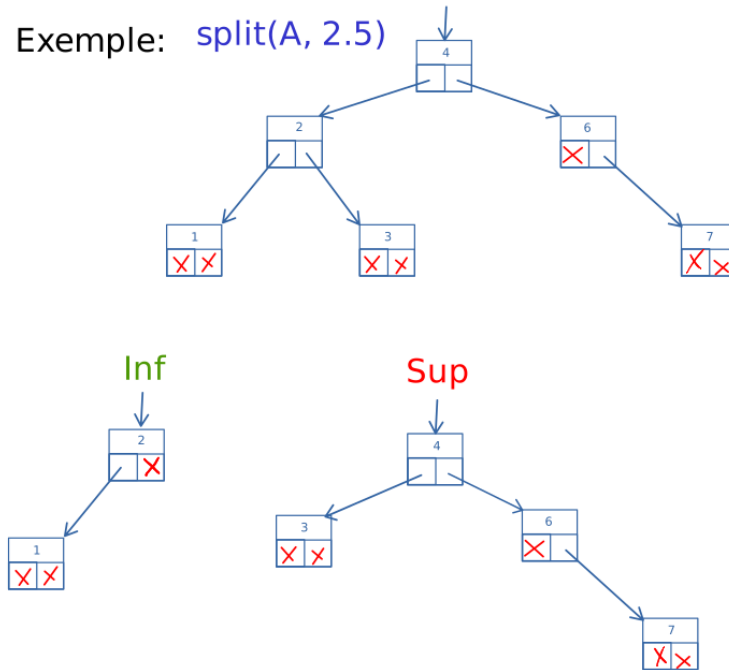


Figure 36 Exemple de coupe d'un RBST

On distinguera 3 cas :

Cas 1 : $value = tree \rightarrow value$

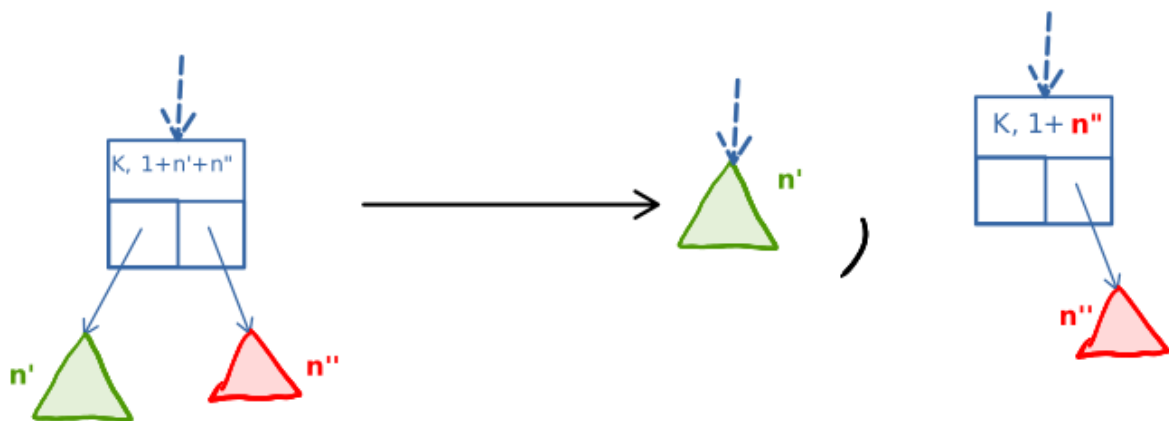


Figure 37 Cas 1 splitRBST

Cas 2 : $value > tree \rightarrow value$

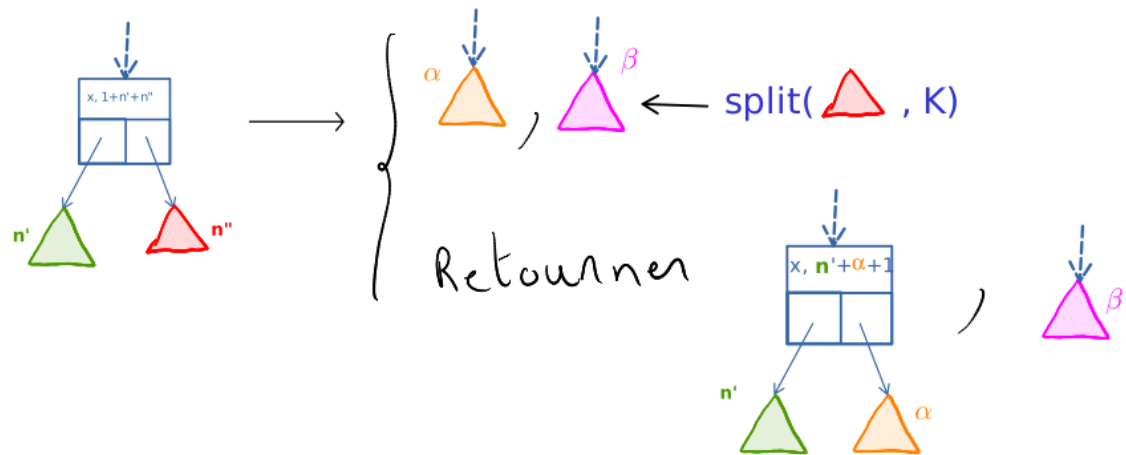


Figure 38 Cas 2 `splitRbst`

Cas 3 : $value < tree \rightarrow value$

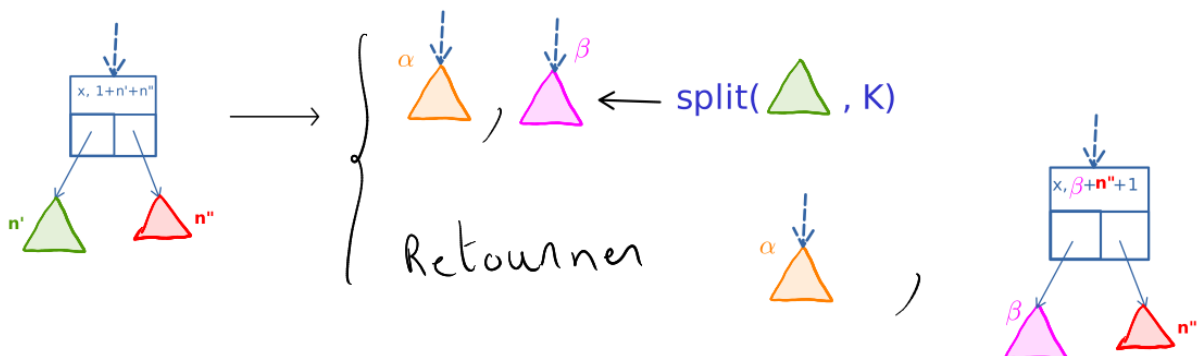


Figure 39 Cas 3 `splitRbst`

On implémente ces 3 cas **de manière récursive** tout en mettant à jour la taille des arbres `*inf` et `*sup` et en faisant attention à la bonne gestion mémoire.

6.2.3. Insertion d'une valeur donnée à la racine d'un RBST

La fonction `'insertAtRoot'` a pour but d'insérer une nouvelle valeur dans un RBST à la **racine de celui-ci**. Ceci implique de **diviser l'arbre existant** en appelant la fonction `'splitRbst'` sur celui-ci de manière à le couper par rapport à la nouvelle valeur à insérer.

La fonction va donc allouer dynamiquement de la mémoire pour deux pointeurs vers des arbres (`inf`, `sup`) qui vont par la suite contenir les deux parties résultantes l'appel de `'split'`.

Ensuite, on alloue dynamiquement de la mémoire pour le nouveau nœud contenant la nouvelle valeur et on remplit ses champs de la façon suivante :

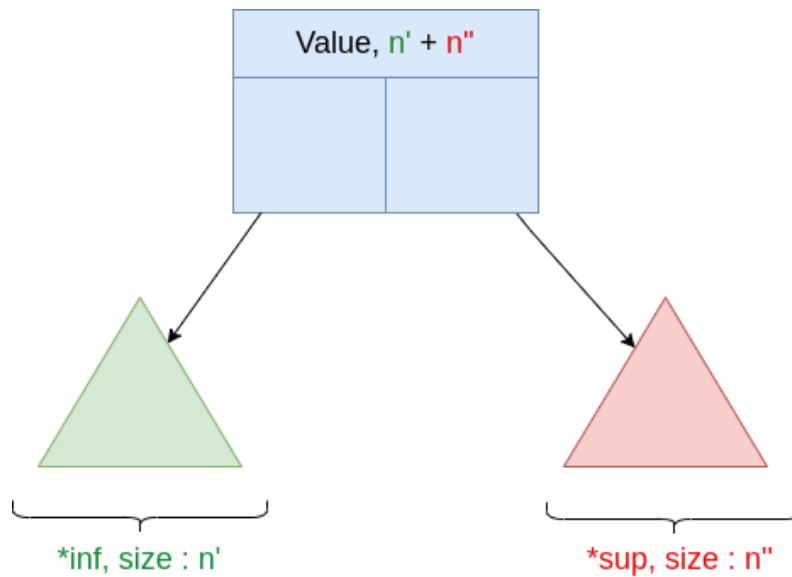


Figure 40 Illustration de la fonction *insertAtRoot*

6.2.4. Insertion d'une valeur donnée dans un RBST

La fonction '*addToRBST*' est conçue pour ajouter une nouvelle valeur à un RBST en ajoutant **une notion de probabilité**. Le principe d'ajout avec probabilité repose sur une approche aléatoire pour décider si la nouvelle valeur doit être insérée comme un nœud enfant du nœud courant ou comme nouvelle racine de l'arbre associé au nœud courant. Cette approche est basée sur la génération d'une probabilité aléatoire p qui est par la suite comparée à un seuil qui sera ici égal à l'inverse de la taille de l'arbre associé au nœud courant + 1.

Exemple :

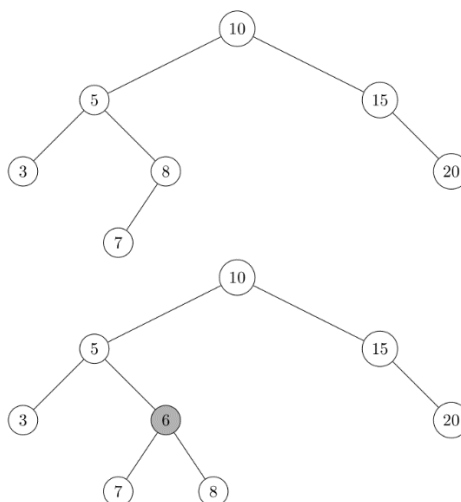


Figure 41 Exemple d'insertion dans un RBST

Étape 1 : nœud : 10

$p = 0.3$, seuil = $1/8 = 0.125$

$p > \text{seuil} \Rightarrow$ on ajoute pas à la racine

Étape 2 : nœud 5

$p = 0.4$, seuil = $1/5 = 0.200$

$p > \text{seuil} \Rightarrow$ on ajoute pas à la racine

Étape 3 : nœud 8

$p = 0.1$, seuil = $1/3 = 0.333$

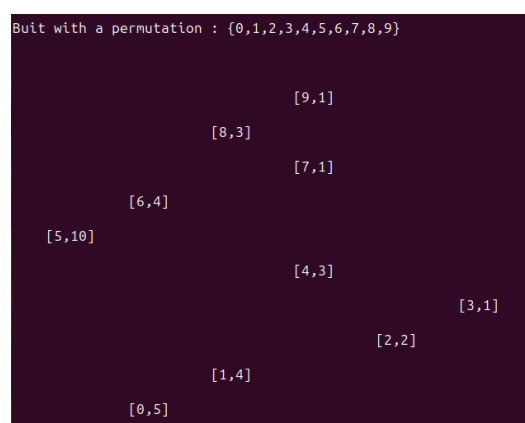
$p < \text{seuil} \Rightarrow$ **on ajoute à la racine**

En conclusion, cette approche permet d'obtenir plusieurs arbres différents pour une permutation donnée.

6.2.5. Création d'un RBST à partir d'une permutation donnée

La fonction '*buildRBSTFromPermutation*' est implémentée de la même façon que pour un ABR mais ici, **on insère au sens d'un RBST**.

6.3. Traces d'exécutions



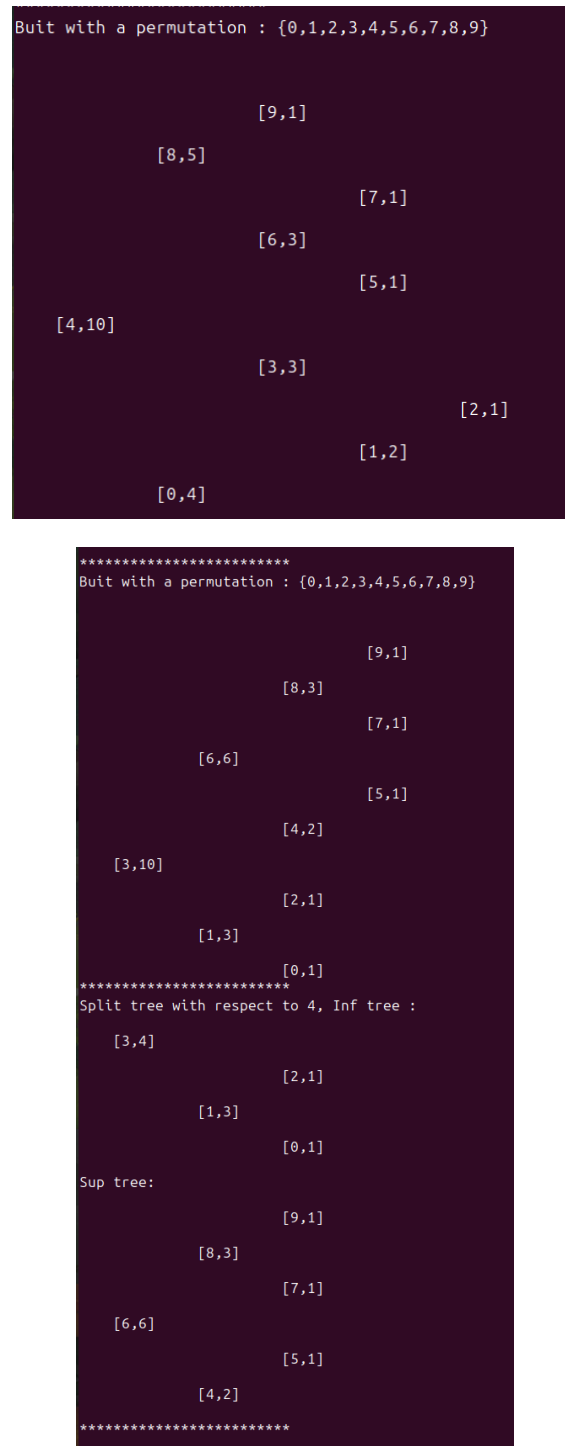


Figure 42 Tests exhaustifs pour RBST

6.4. Résultats et critiques

Le code fourni nous permet de mener un grand nombre d'expériences afin d'évaluer les performances des opérations sur les arbres de recherche binaire randomisés (RBST) en

fonction de la distribution des données. Les résultats expérimentaux obtenus révèlent des différences importantes sur ces performances par rapport aux ABR.

Question : Quelles conclusions pouvons-nous tirer des résultats d'expériences que nous obtenons ?

Réponse :

En réalisant 1000 tests sur des permutations de données uniformes et non uniformes de taille 1000, nous constatons que, en moyenne, les temps de construction et de recherche, ainsi que les hauteurs des arbres seront similaires pour le cas d'une distribution uniforme et non uniforme

Ici la distribution non uniforme sera encore une distribution uniforme de taille n telle que les $n/10$ premiers éléments seront triés.

```

Comparison between the data structures
size of the permutations: 1000
number of tests: 1000
Binary search tree with uniform distribution:
-> The average time to build is : 87.738000
-> The average height is : 21.165000
-> The average time to perform searches is : 83.930000
Binary search tree with non uniform distribution:
-> The average time to build is : 221.474000
-> The average height is : 105.876000
-> The average time to perform searches is : 225.838000
Randomized binary search tree with uniform distribution:
-> The average time to build is : 752.567000
-> The average height is : 21.014000
-> The average time to perform searches is : 85.060000
Randomized binary search tree with non uniform distribution
-> The average time to build is : 800.806000
-> The average height is : 20.924000
-> The average time to perform searches is : 96.664000

```

Figure 43 Résultats sur distributions uniformes et non uniformes pour RBST

Ensuite, nous procédons à plusieurs expériences sur des échantillons de différentes tailles dans le but de tracer des graphiques afin d'étudier le comportement temporel des opérations de construction et de recherche ainsi que l'évolution de la hauteur des arbres en fonction de leur taille.

Cas non uniforme :

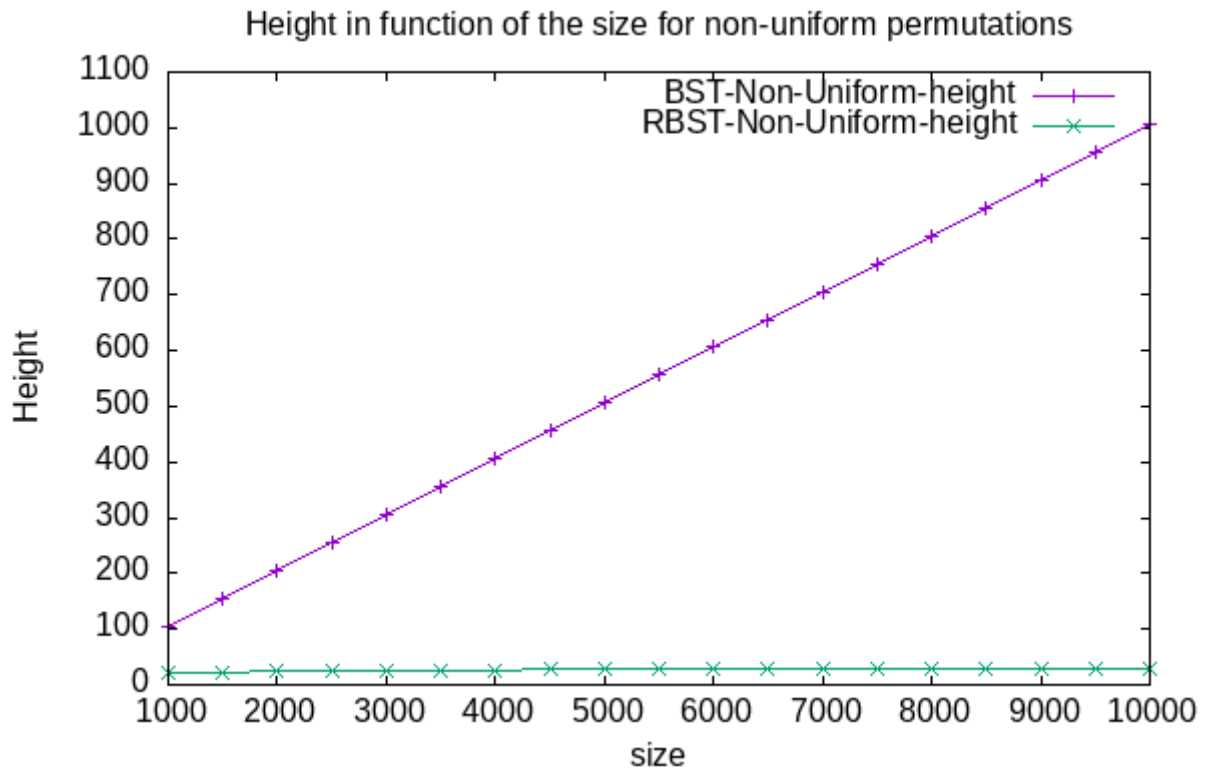


Figure 44 Hauteur RBST et ABR en fonction de leur taille (cas non uniforme)

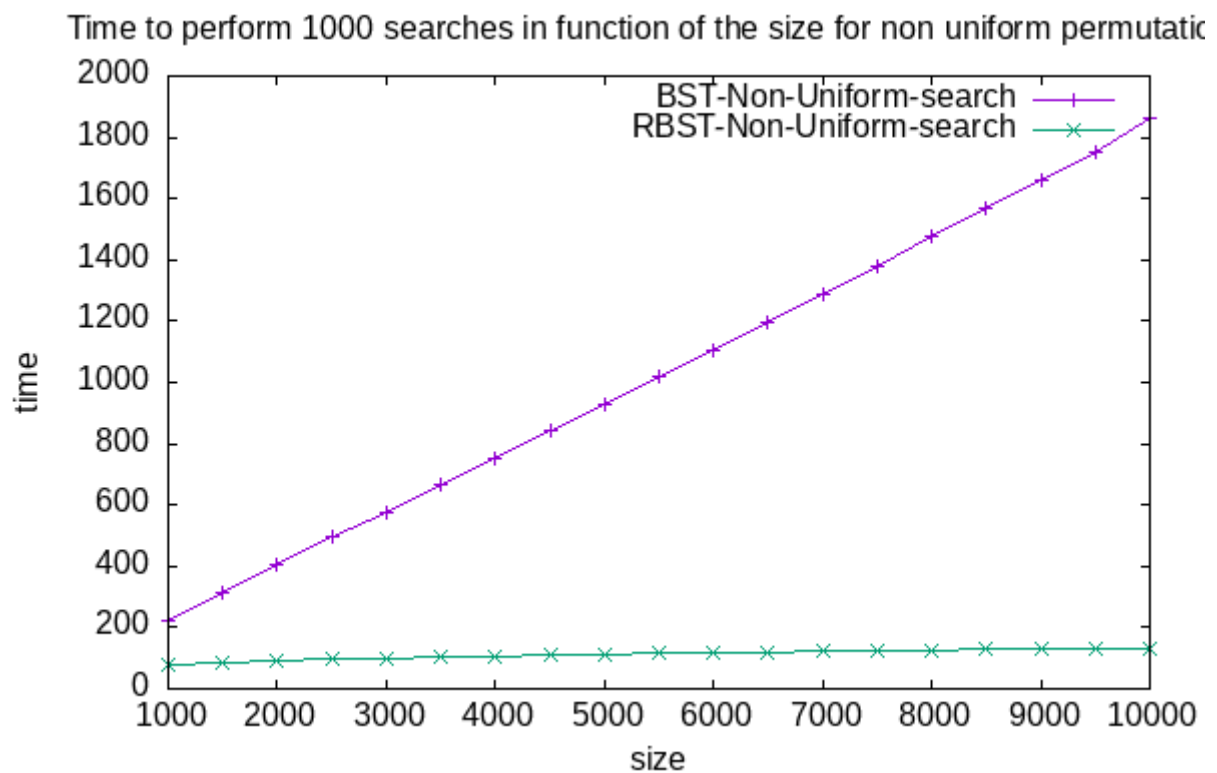


Figure 45 Temps pour effectuer 1000 recherches dans un RBST et un ABR en fonction de leur taille (cas non uniforme)

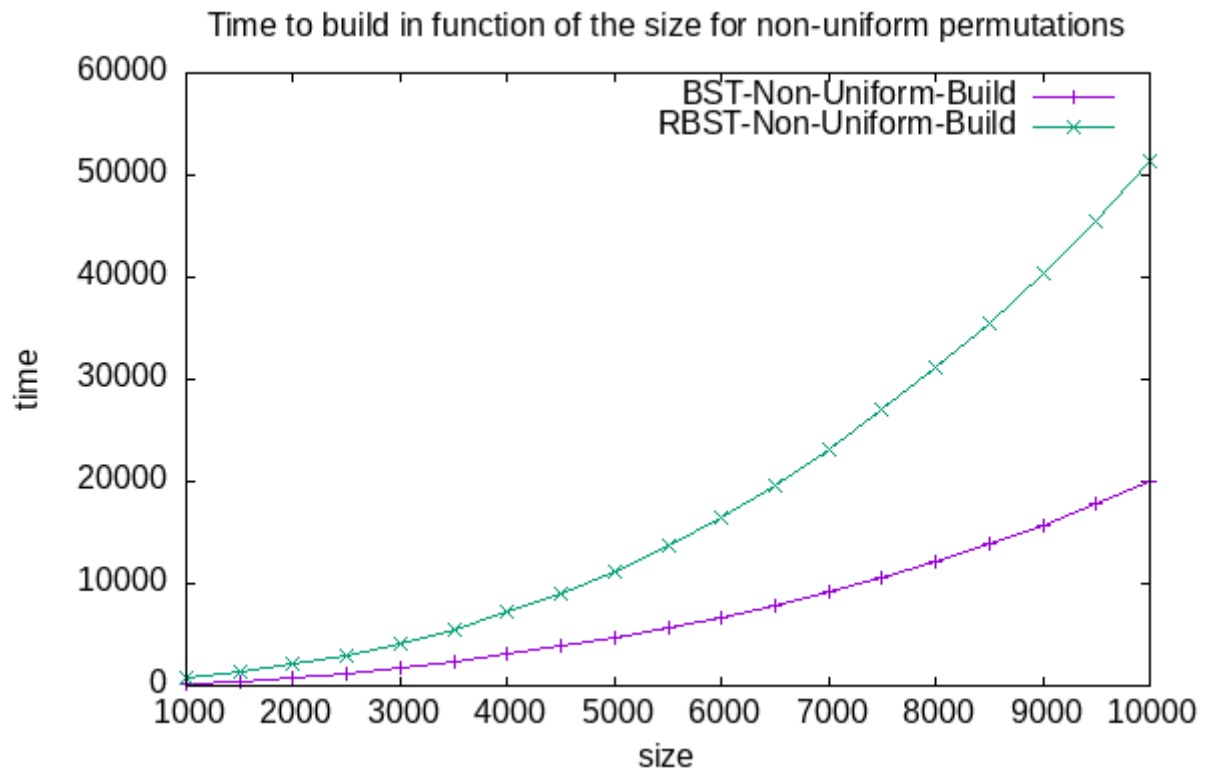


Figure 46 Temps pour construire un ABR et un RBST en fonction de leur taille (cas non uniforme)

Cas uniforme :

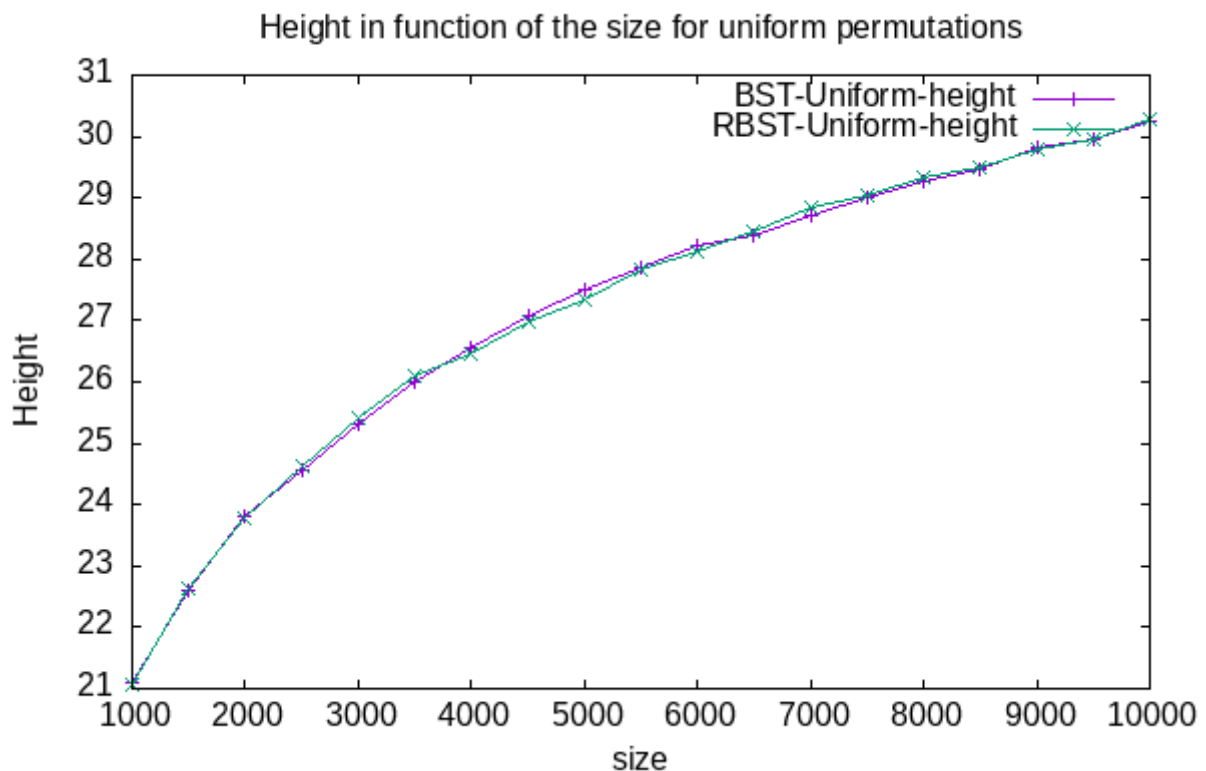


Figure 47 Hauteur RBST et ABR en fonction de leur taille (cas uniforme)

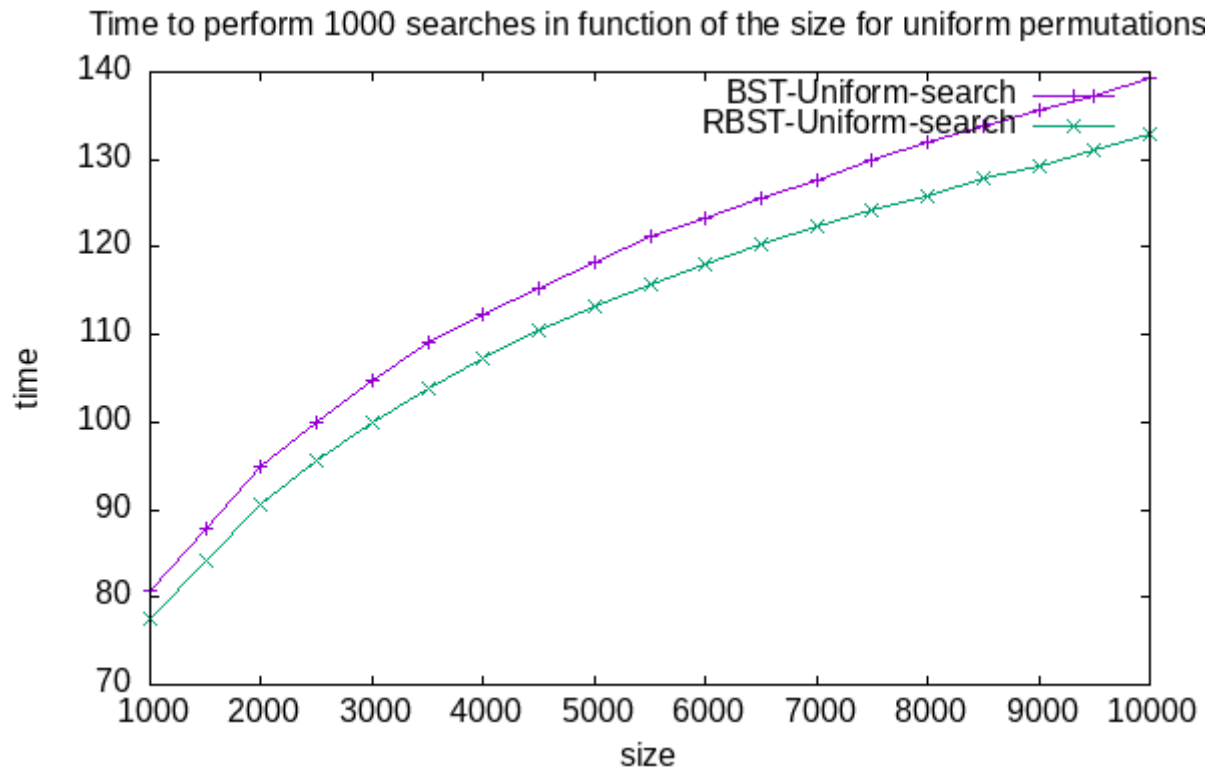


Figure 48 Temps pour effectuer 1000 recherches dans un RBST et un ABR en fonction de leur taille (cas uniforme)

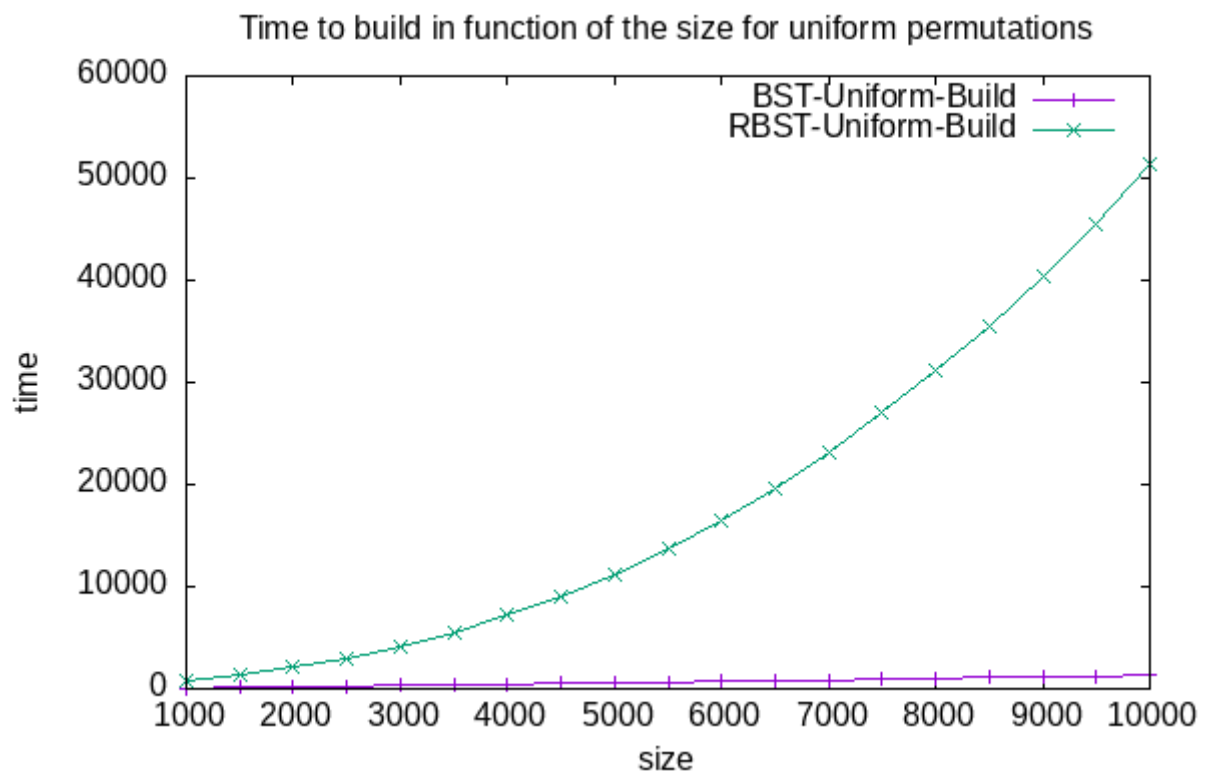


Figure 49 Temps pour construire un ABR et un RBST en fonction de leur taille (cas uniforme)

On constate grâce aux graphiques et au 5.4, que **dans le cas d'une distribution non uniforme, les RBST présentent un comportement similaire à celui des ABR lorsqu'ils sont confrontés à une distribution uniforme.** Cette similitude découle du fait que les RBST, par **leur nature aléatoire, ne sont pas sensibles à l'ordre initial des éléments.** Ainsi, même en présence d'une distribution non uniforme, les performances des opérations sur les RBST demeurent stables et prédictibles, offrant une efficacité comparable à celle des ABR dans des conditions uniformes. **Cette observation souligne l'atout des RBST, qui conservent leur efficacité quel que soit le schéma de distribution des données.**

7. Arbres rouges-noirs

Les arbres Rouges-Noirs (ARN) sont une variante des arbres binaires de recherche (ABR) qui apportent une structure supplémentaire pour garantir un équilibre relatif dans l'arbre. Cette structure de données a été inventée par Rudolf Bayer en 1972 puis étendue en 1978 par GuibS-Sedgewick.

Dans ce travail pratique nous allons se pencher sur l'implémentation et l'utilisation des ARN à travers différentes fonctions clés telles que la création d'un arbre vide, la rotation autour d'un nœud, l'équilibrage d'un ARN, l'insertion, la recherche de valeurs dans ces structures, les calculs de la taille et de la hauteur de l'arbre. Avant de plonger dans les détails des fonctions et des opérations, il est impératif de comprendre leur concept fondamental. Un **ARN** possède la même structure d'un ABR où **chaque nœud possède en plus une couleur et un pointeur vers son nœud parent**. Ils se distinguent des ABR par **leurs règles d'équilibrage**. En effet, alors que les ABR peuvent devenir déséquilibrés avec des séquences d'insertions ou de suppressions spécifiques, les ARN maintiennent un équilibre relatif défini par plusieurs propriétés. En comprenant ces opérations et en analysant leurs performances, nous pourrions tirer des conclusions sur le caractère dynamique de cette structure de données.

7.1. Définition et propriétés de la structure de données

NodeRedBlackBST	<code>typedef NodeRedBlackBST* RedBlackBST;</code>
<ul style="list-style-type: none">- value : int- leftBST : NodeRedBlackBST*- rightBST : NodeRedBlackBST*- color : int- father : NodeRedBlackBST*	

Figure 50 Structure d'un ARN

Un ARN est un ABR ayant les propriétés additionnelles suivantes :

- Chaque nœud est soit rouge, soit noir ;
- Tous les nœuds NULL ainsi que la racine sont noirs ;
- Un nœud rouge n'a pas d'enfant rouge ;
- Tous les chemins d'un nœud à ses descendants NULL traversent le même nombre de sommets (nœuds ou feuilles) noirs.

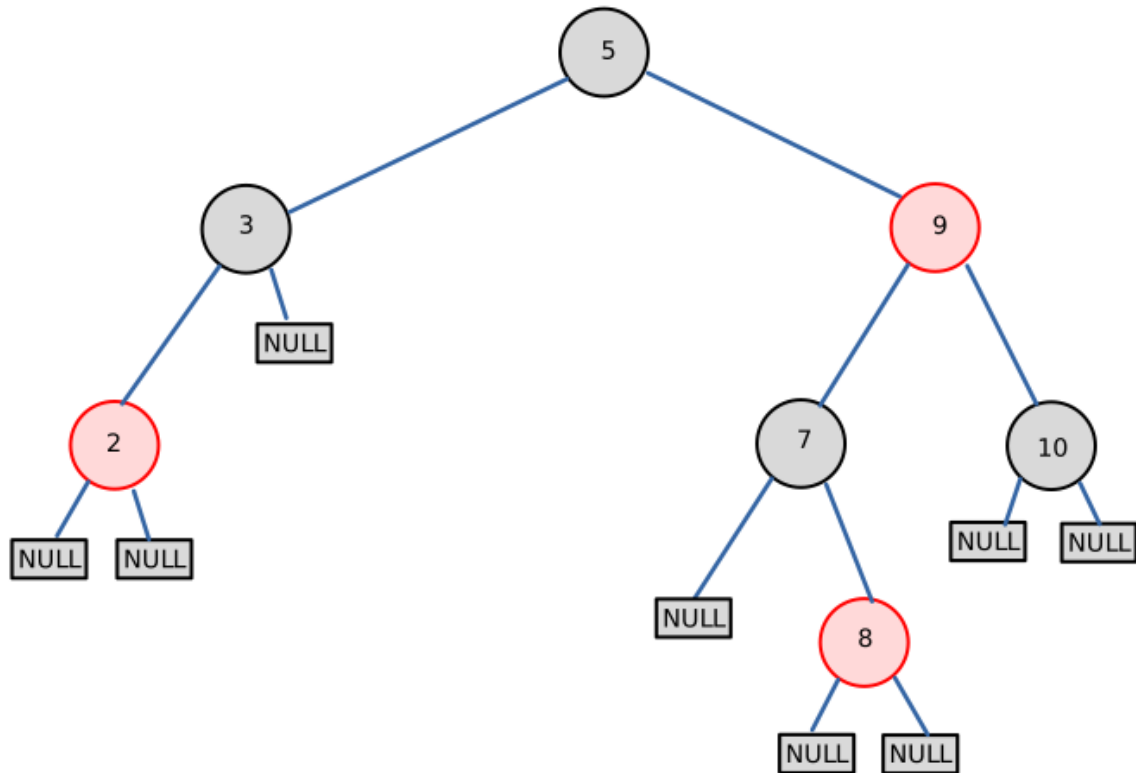


Figure 51 Exemple d'un ARN

7.2. Fonctions

Nous ne procéderons pas à une analyse détaillée des fonctions telles que 'createEmptyRedBlackBST', 'freeRedBlackBST', 'heightRedBlackBST', 'searchRedBlackBST', 'prettyPrintRedBlackBST' et 'buildRedBlackBSTFromPermutation' dans cette section, car elles suivent la même logique que celles utilisées pour les arbres binaires de recherche (ABR) (5.2) et les RBST (6.2).

7.2.1. Rotation d'un ARN autour d'un nœud

On nous fournit deux fonctions permettant d'effectuer une rotation gauche et droite autour d'un nœud au sein d'un ARN.

Rotations gauche/droite :

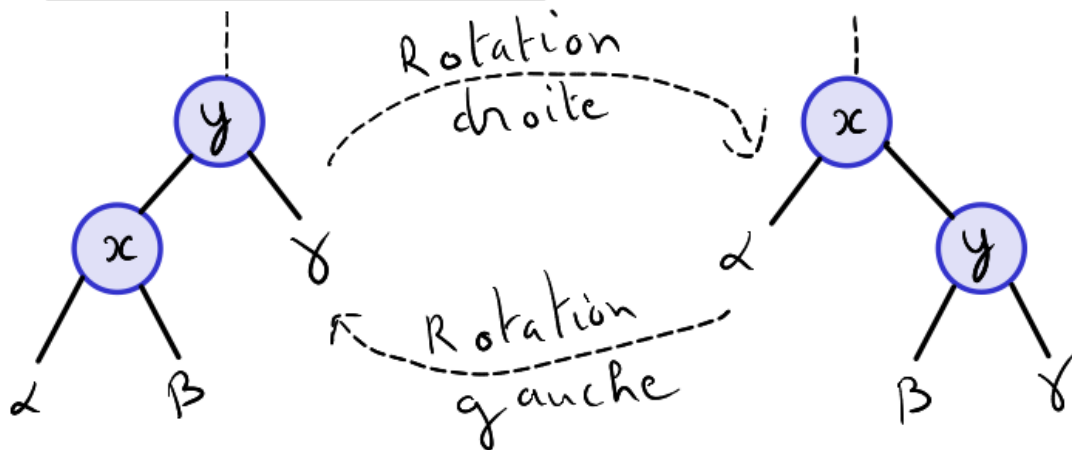


Figure 52 Rotation autour d'un nœud ARN

Ces rotations seront nécessaires pour réparer lorsqu'une règle n'est plus respectée après insertion ou suppression.

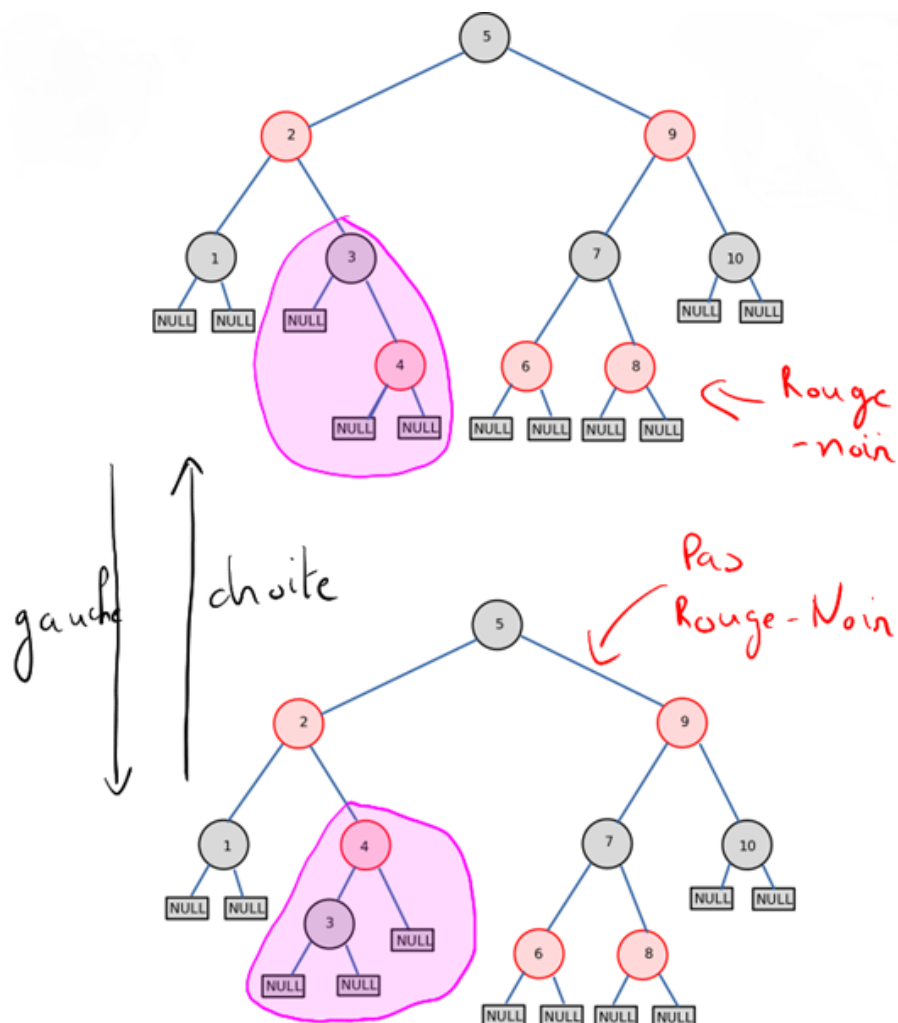


Figure 53 Exemple de rotation au sein d'un arbre

7.2.2. Équilibrage d'un ARN

Lorsqu'on insère un nouveau nœud dans un ARN, on insère un nœud rouge. Ainsi, après insertion d'un nouveau nœud, seuls 2 règles peuvent être non respectées mais 1 seule à la fois :

- tous les nœuds NULL ainsi que la racine sont noirs ;
- un nœud rouge n'a pas d'enfant rouge ;

On ne corrige que dans 4 cas :

- 1- lorsque la racine est rouge (règle 2 non respectée)
- 2- le nœud inséré et son père sont rouges (règle 3 non respectée)
 - 2-1- l'oncle est rouge
 - 2-2- l'oncle est noir et le nœud inséré est Gauche-Gauche ou Droite-Droite
 - 2-3- l'oncle est noir et le nœud inséré est Gauche-Droite ou Droite-Gauche

Dans tous les autres cas, on ne fait rien car l'arbre est Rouge-Noir.

Cas 1 :

Cas 1 : La racine est rouge
 (règle 2 non respectée)
 → on change la couleur du nœud

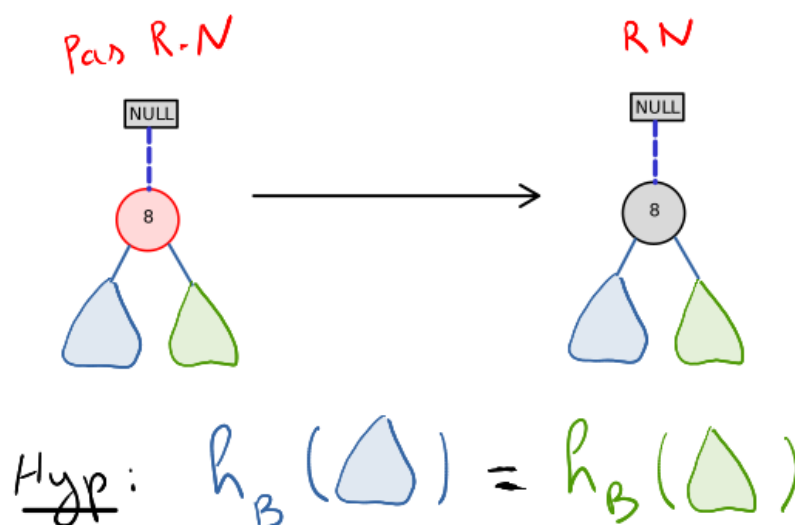


Figure 54 Cas 1 équilibrage ARN

Cas 2-1 : L'oncle est rouge

2-1 : L'oncle est rouge (B inséré)

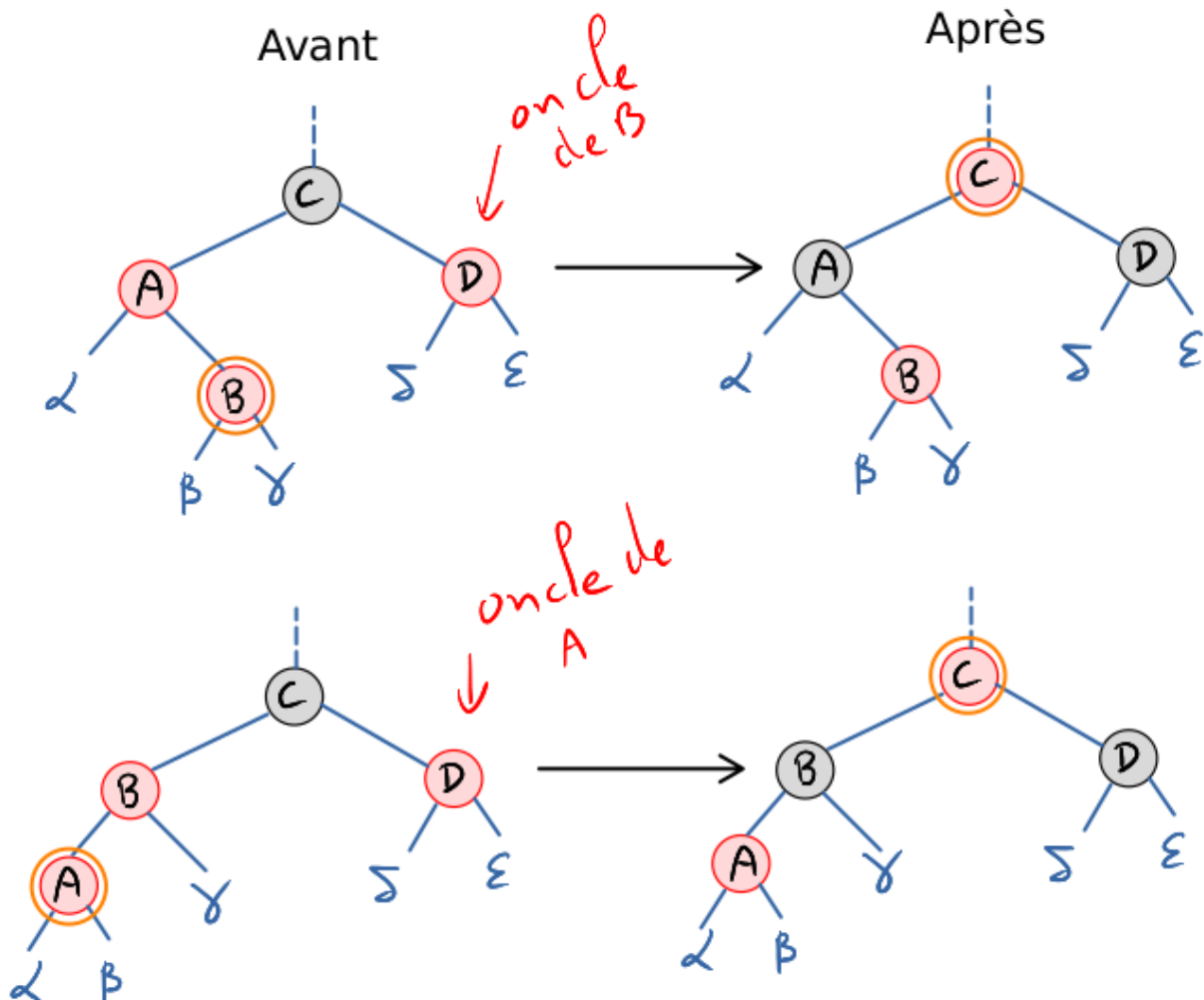


Figure 55 Cas 2-1 équilibrage ARN

Avec comme hypothèses : $hauteurNoire(alpha) = hauteurNoire(beta) = hauteurNoire(gamma) = hauteurNoire(delta) = hauteurNoire(epsilon)$

Concernant le cas 2-1, après opération, C devient rouge. On se retrouve donc dans un des cas suivants :

- Si C est la racine, on revient au cas 1.
- Si C n'est pas la racine et que son père est noir, alors on a un ARN
- Si C n'est pas la racine et que son père est rouge, alors son grand-père est noir, on revient à une des situations précédentes.

On réitère ce procédé de manière récursive jusqu'à la racine de l'arbre.

Cas 2-2 : L'oncle est noir et le nœud inséré est Gauche-Droite ou Droite-Gauche

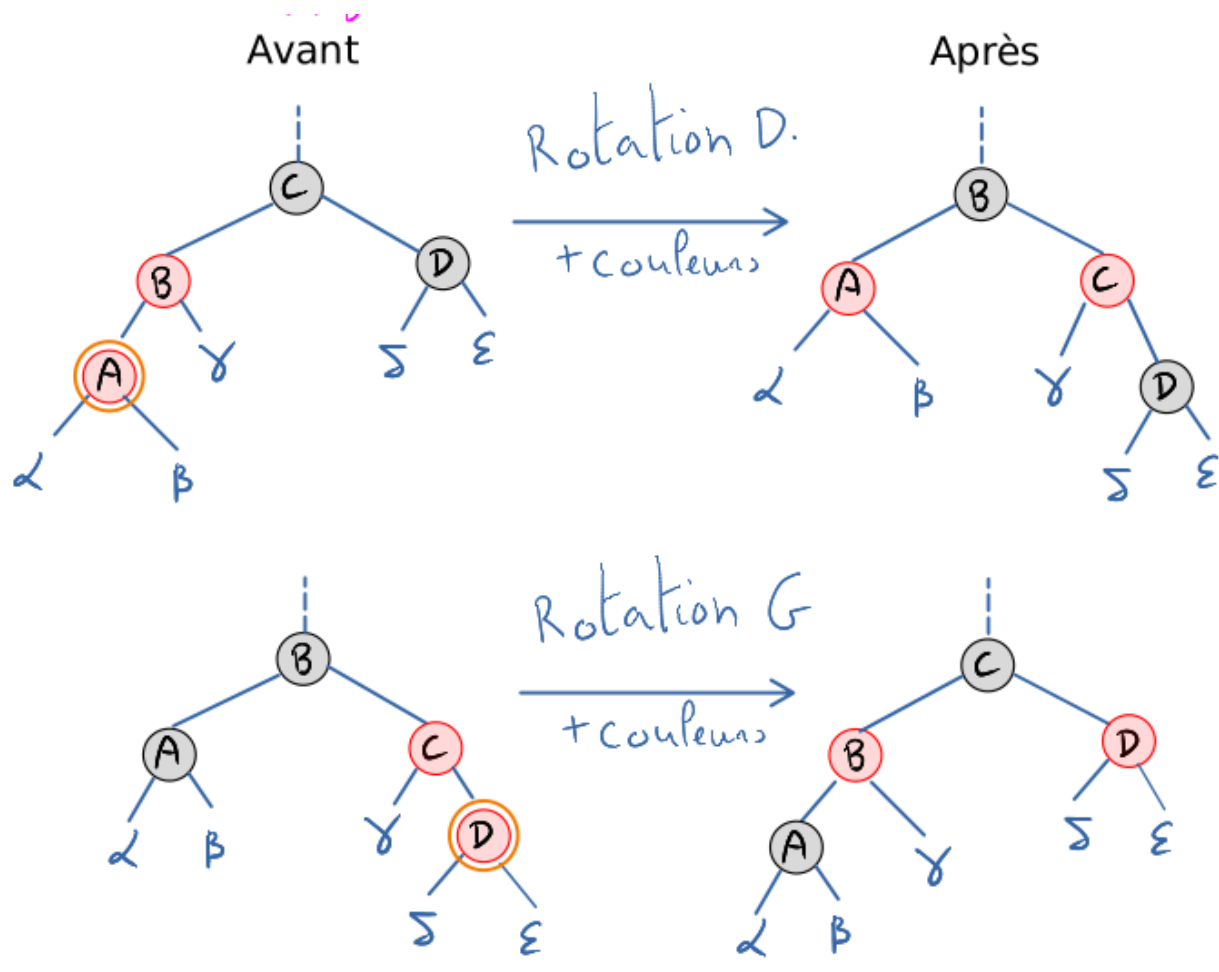


Figure 56 Cas 2-2 équilibrage ARN

Après opération, on obtient bien une structure d'ARN.

Cas 2-3 : L'oncle est noir et le nœud inséré est Gauche-Droite ou Droite-Gauche

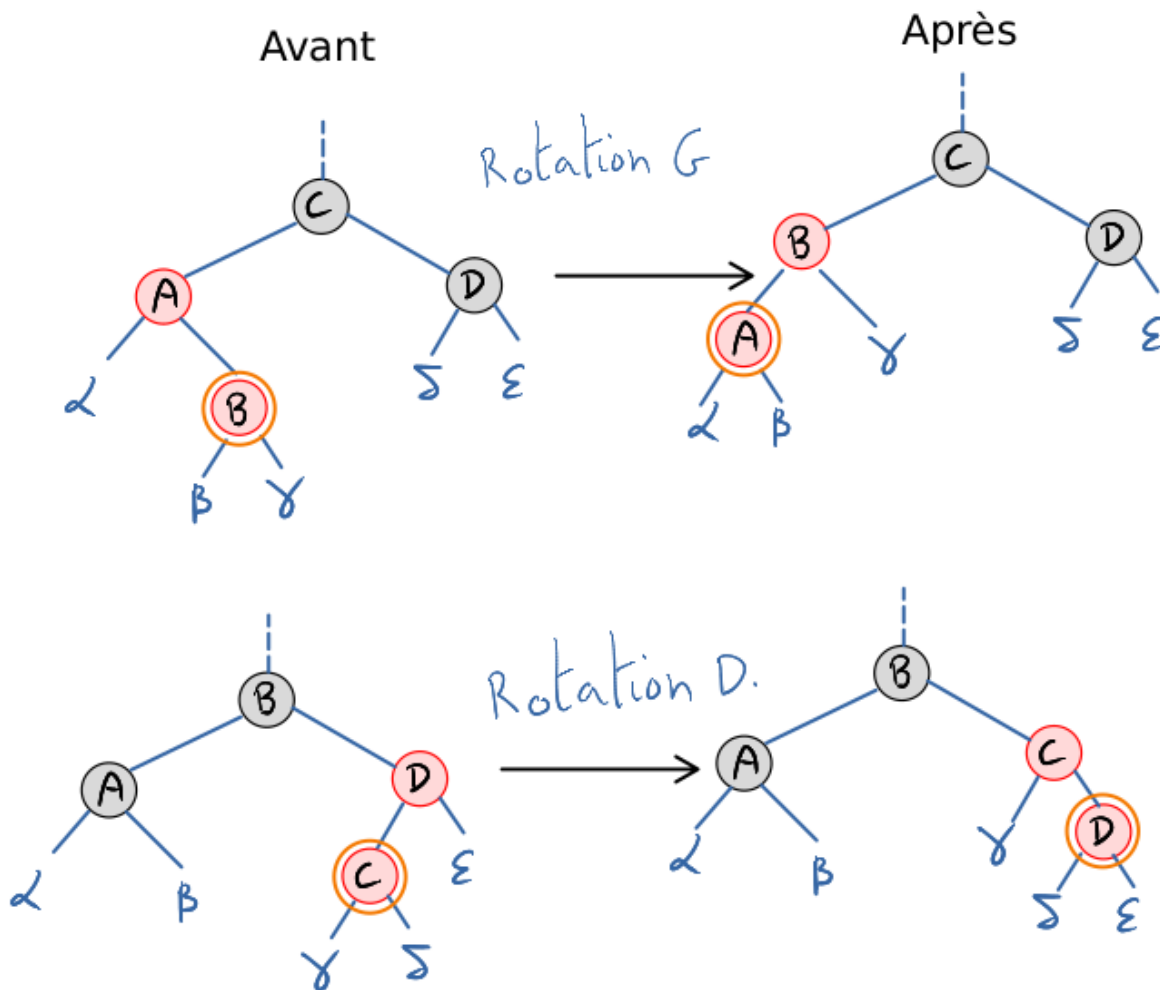


Figure 57 Cas 2-3 équilibrage ARN

Après opération, on se retrouve dans le cas 2-2.

En résumé

On implémente cela par une fonction récursive prenant en paramètre un pointeur sur l'ARN à équilibré (*RedBlackBST* tree*) ainsi qu'un pointeur vers le nœud venant d'être inséré (*NodeRedBlackBST *curr*).

7.2.3. Insertion d'une valeur dans une ARN

La fonction *insertNodeRedBlackBST* insère une valeur donnée dans un ARN. Elle prend en argument un pointeur vers l'ARN (*RedBlackBST* tree*) où l'on insère la valeur et la valeur à insérée (*int value*).

On suit la même logique que lors de l'insertion dans un ABR classique (5.2.3) mais après que la nouvelle valeur a été insérée, **on équilibre l'arbre** à l'aide de la fonction *balancedRedBlackBST*.

7.2.4. Calcul de la hauteur noire d'un ARN

La fonction *blackHeightRedBlackBST* retourne la hauteur noire d'un ARN (compte la racine si elle est noire) ou -1 si la hauteur noire de ses enfants ne sont pas égales ou si elles sont égales à -1. L'implémentation de cette fonction nous a été fournie. La fonction parcourt de manière récursive l'entièreté de l'arbre jusqu'à atteindre le bas de celui-ci. Ensuite en dépilant les appels, on retourne les valeurs adéquates en fonction de la couleur du nœud et de la hauteur noire de ses sous-arbres (de même hauteur noire ou non).

7.2.5. Vérification si un ARN possède une structure valide

La fonction *isRedBlackBST* renvoie 1 si l'arbre passé en paramètre possède bien une structure d'ARN et 0 sinon. La fonction appelle simplement *blackHeightRedBlackBST* et selon la valeur renvoyée, on renvoie 1 ou 0.

7.3. Traces d'exécutions

Voici un affichage console de quelques tests exhaustifs.

9 4 3 1 2 8 7 5 6 0 -----	
[9,1]	[9,0]
-----	[8,1]
[9,0]	[4,0]
[4,1]	[3,1]
-----	[2,0]
[9,1]	[1,1]
-----	[9,1]
[4,0]	[8,0]
[3,1]	[7,1]
-----	[4,0]
[9,0]	[3,1]
[4,0]	[2,0]
[3,0]	[1,1]
-----	[9,0]
[9,0]	[8,1]
[4,0]	[7,0]
[3,1]	[5,1]
[2,0]	[4,0]
[1,1]	[3,1]
	[2,0]
	[1,1]


```
hauteur = 3
searchRedBlackBST(tree, 7) != NULL : 1
searchRedBlackBST(tree, 11) != NULL : 0
isRedBlackBST(tree) == 1 : 1
ADRESS tree BEFORE : 0x5e482b6fb6e0
ADRESS tree AFTER : (nil)
```

Figure 58 Tests exhaustifs ARN

7.4. Résultats et critiques

Le code fourni nous permet de mener un grand nombre d'expériences afin d'évaluer les performances des opérations sur les arbres rouges-noirs en fonction de la distribution des données. Les résultats expérimentaux obtenus révèlent des différences importantes sur ces performances par rapport aux ABR.

Question : Quelles conclusions pouvons-nous tirer des résultats d'expériences que nous obtenons ?

Réponse :

En réalisant 1000 tests sur des permutations de données uniformes et non uniformes de taille 1000, nous constatons que, en moyenne, les temps de construction et de recherche, ainsi que les hauteurs des arbres seront similaires pour le cas d'une distribution uniforme et non uniforme

Ici la distribution non uniforme sera encore une distribution uniforme de taille n telle que les $n/10$ premiers éléments seront triés.

```

Comparison between the data structures
size of the permutations: 5000
number of tests: 1000
Binary search tree with uniform distribution:
-> The average time to build is : 578.978000
-> The average height is : 27.389000
-> The average time to perform searches is : 115.828000
Binary search tree with non uniform distribution:
-> The average time to build is : 4664.137000
-> The average height is : 505.988000
-> The average time to perform searches is : 928.002000
Randomized binary search tree with uniform distribution:
-> The average time to build is : 12136.728000
-> The average height is : 27.339000
-> The average time to perform searches is : 123.185000
Randomized binary search tree with non uniform distribution:
-> The average time to build is : 12003.052000
-> The average height is : 27.375000
-> The average time to perform searches is : 122.943000
Red-Black tree with uniform distribution:
-> The average time to build is : 774.855000
-> The average height is : 14.032000
-> The average time to perform searches is : 102.982000
Red-Black tree with non uniform distribution:
-> The average time to build is : 780.510000
-> The average height is : 15.000000
-> The average time to perform searches is : 106.800000

```

Figure 59 Résultats sur distributions uniformes et non uniformes pour ARN, ABR, RBST

Cas uniforme :

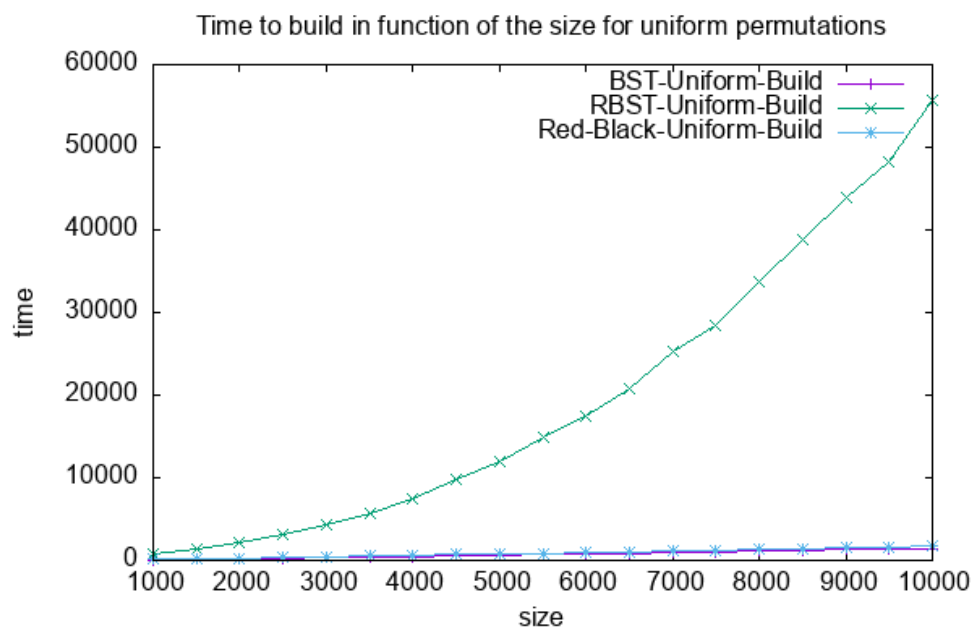


Figure 60 Temps pour construire ARN, RBST, ABR en fonction de leur taille (cas uniforme)

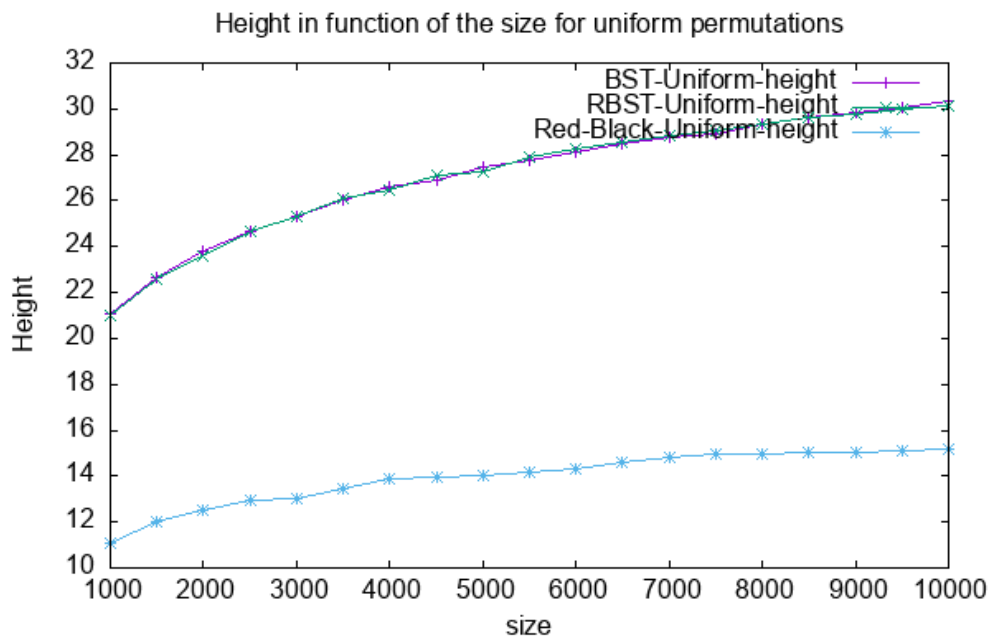


Figure 61 Hauteur d'un ARN, RBST, ABR en fonction de leurs tailles (cas uniforme)

on de leur taille (cas uniforme)

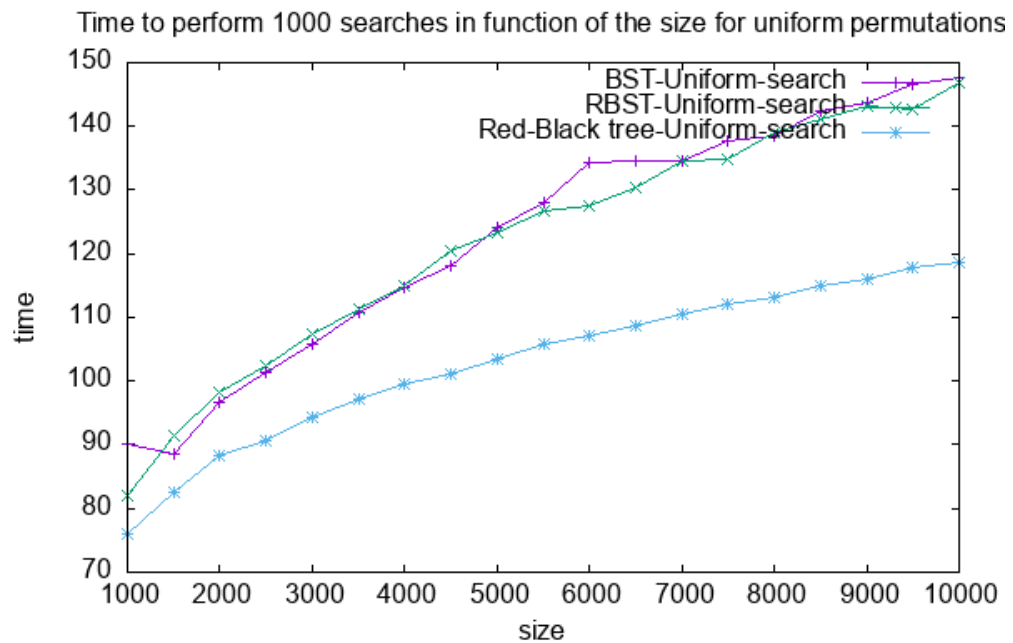


Figure 62 Temps pour effectuer 1000 recherches dans un RBST, ARN et un ABR en fonction de leurs tailles (cas uniforme)

Cas non uniforme :

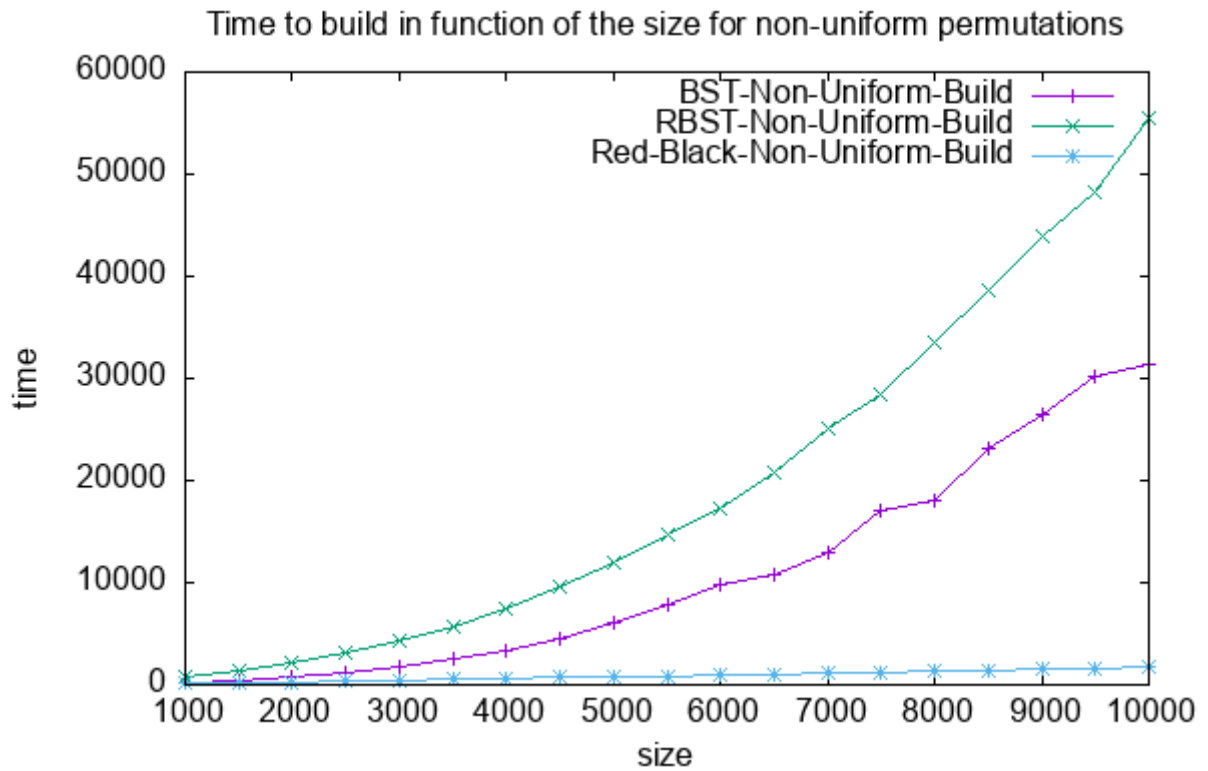


Figure 63 Temps pour construire ARN, RBST, ABR en fonction de leur taille (cas non uniforme)

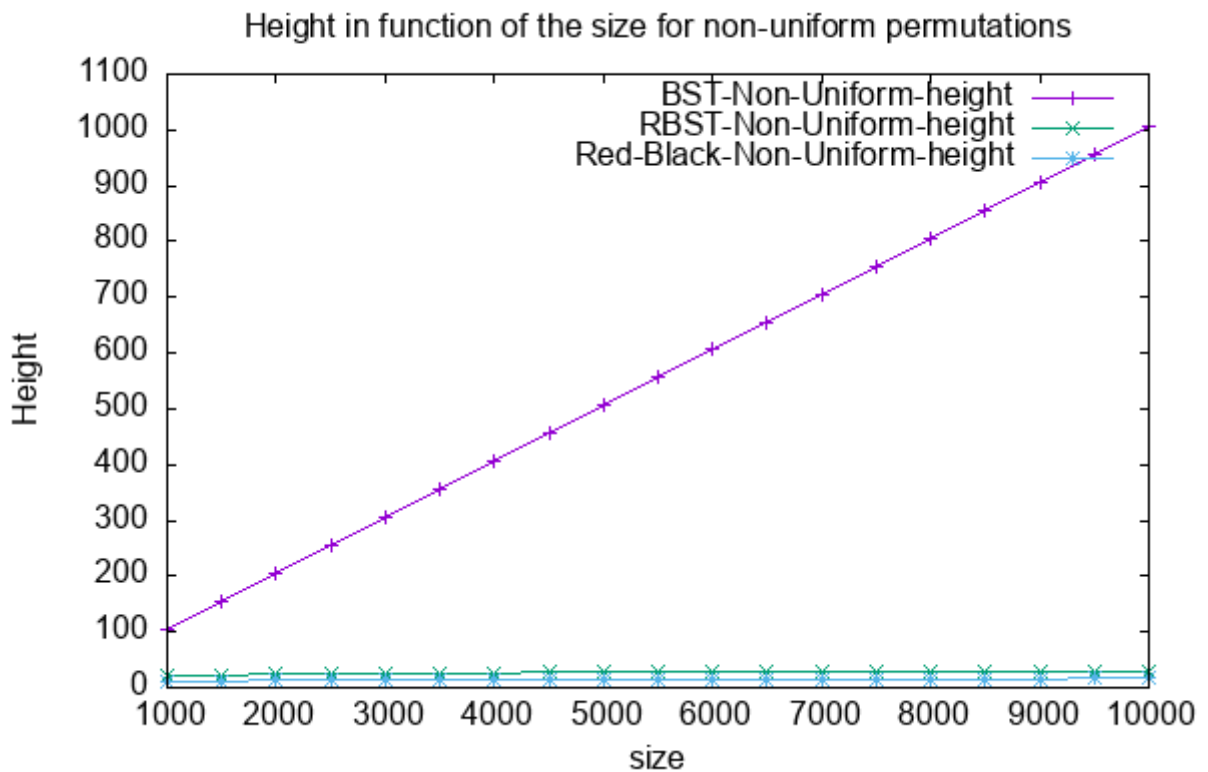


Figure 64 Hauteur d'un ARN, RBST, ABR en fonction de leur taille (cas non uniforme)

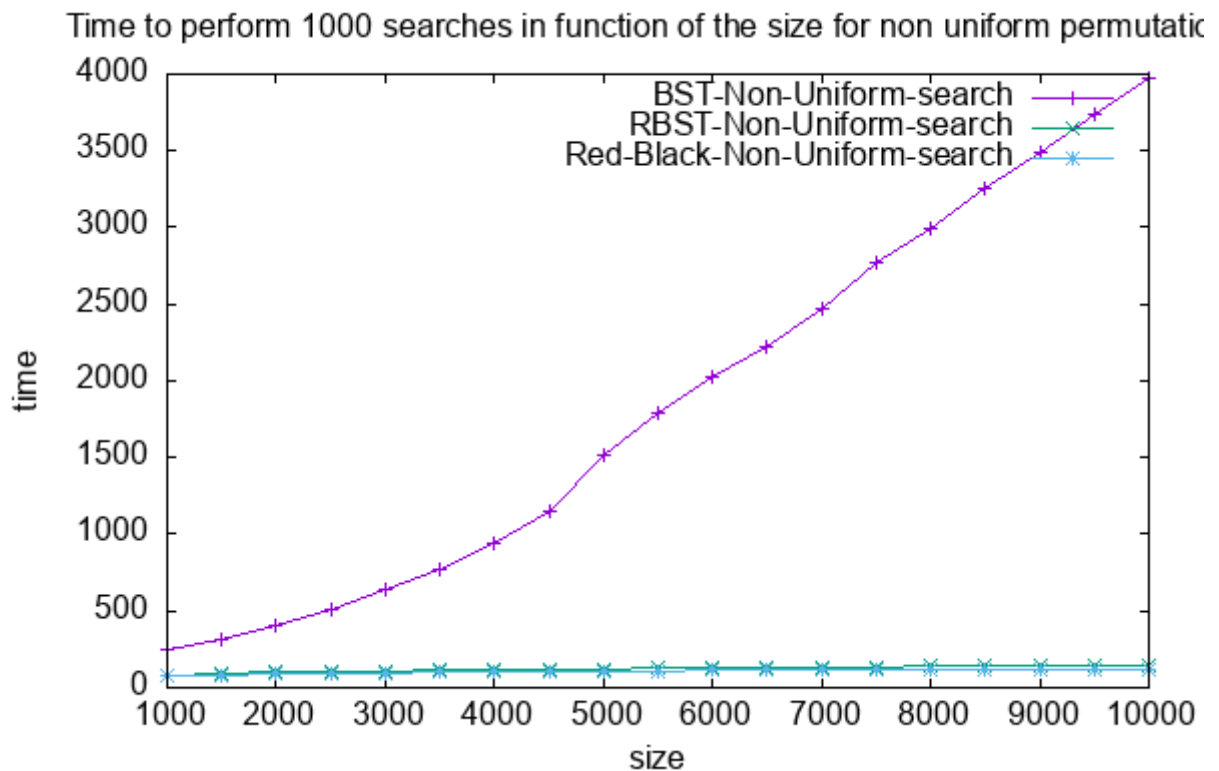


Figure 65 Temps pour effectuer 1000 recherches dans un RBST, ARN et un ABR en fonction de leurs tailles (cas non uniforme)

Les RBST, par leur caractère aléatoire et les ARN grâce à leurs propriétés d'équilibrage, ont globalement le même comportement dans les deux cas de distribution par rapport aux ABR, avec une meilleure performance des ARN pour la construction d'un arbre.

On remarque cependant que les ARN présentent de meilleures performances en termes de hauteur et de temps de construction.

En conclusion, les RBST et ARN montrent de meilleures performances par rapport aux ABR, mais par leurs propriétés d'équilibrage, les ARN se démarquent en termes de performances de recherche, de construction et de hauteur. Ainsi pour des applications nécessitant des opérations de recherches, on privilégiera les ARN puis les RBST et ensuite les ABR.

8. File de priorité (Heap)

Dans le monde de l'informatique, les files de priorité sont des outils précieux pour traiter les données en attribuant des priorités à certains éléments. Elles permettent d'insérer des éléments selon leur priorité, de récupérer l'élément avec la plus faible priorité, de le supprimer et de modifier sa priorité.

Une implémentation courante des files de priorité utilise des tas, des structures de données où chaque nœud a une priorité inférieure ou égale à celle de ses descendants. Cette approche offre des performances optimales pour les opérations de file de priorité.

Dans notre rapport, nous explorerons la mise en œuvre des files de priorité à l'aide de tas, en examinant en détail les opérations clés et en soulignant leur importance dans divers contextes, notamment dans des algorithmes tels que celui de Prim pour les arbres couvrants de poids minimum.

8.1. Définition de la structure

Cette structure représente un tas, un type de structure de données qui stocke une collection d'éléments avec des priorités associées. Chaque élément dans le tas est représenté par un index, et ses informations sont stockées dans les tableaux '*position*', '*heap*', et '*priority*'. Le champ *n* spécifie le nombre maximal d'éléments pouvant être stockés dans le tas, tandis que *nbElements* donne le nombre réel d'éléments présents dans le tas à un moment donné.

8.2. Fonctions

8.2.1. Création d'un tas vide dont les clés sont $\{1, \dots, n\}$

La fonction '*createHeap*' est chargée de créer un pointeur sur un tas vide, elle prend en paramètre '*n*' le nombre d'éléments maximale du tas. Le tas est alloué dynamiquement où en premier on a alloué la structure *Heap*, pour ensuite allouer les trois tableaux de cette même structure. On initialise les trois tableaux de la structure *Heap* à l'aide d'une boucle où on met initialement tout les cases à -1.

8.2.2. Affichage des données d'un tas

La fonction '*printHeap*' est utilisée pour afficher les propriétés et le contenu d'un tas, notamment son nombre maximal d'éléments, le nombre réel d'éléments présents, les positions des éléments, leurs priorités et les éléments eux-mêmes.

Pour l'affichage du contenu du tableau 'heap', on teste si le tas est non vide pour pouvoir faire le parcours et l'affichage de ce tableau, sinon on affiche un tas vide ([]). Cela permet de visualiser rapidement l'état du tas, ce qui est utile pour le débogage et la compréhension des données stockées.

```
Current heap:
n: 10
nbElements: 0
position: [-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 ]
priority: [-1.00 -1.00 -1.00 -1.00 -1.00 -1.00 -1.00 -1.00 -1.00 -1.00 ]
heap: [ ]
Current heap:
n: 10
nbElements: 2
position: [0 1 -1 -1 -1 -1 -1 -1 -1 -1 ]
priority: [0.00 1.00 -1.00 -1.00 -1.00 -1.00 -1.00 -1.00 -1.00 -1.00 ]
heap: [ 0 1 ]
Current heap:
n: 10
nbElements: 3
position: [0 1 2 -1 -1 -1 -1 -1 -1 -1 ]
priority: [0.00 1.00 2.00 -1.00 -1.00 -1.00 -1.00 -1.00 -1.00 -1.00 ]
heap: [ 0 1 2 ]
Current heap:
n: 10
nbElements: 4
position: [0 1 2 3 -1 -1 -1 -1 -1 -1 ]
priority: [0.00 1.00 2.00 3.00 -1.00 -1.00 -1.00 -1.00 -1.00 -1.00 ]
heap: [ 0 1 2 3 ]
```

Figure 66 Tests exhaustifs Heap

8.2.3. Retourner l'élément avec la plus faible priorité dans le tas

La fonction 'getElement' décrite vérifie si le tas est vide. Si c'est le cas, elle retourne -1, indiquant qu'il n'y a aucun élément dans le tas. Sinon, elle retourne le premier élément du tas, sachant que celui-ci possède la plus faible priorité parmi tous les éléments du tas.

Modifier la priorité d'un élément du tas

La fonction 'modifyPriorityHeap' est utilisée pour changer la priorité d'un élément spécifié dans un tas. Sa liste de paramètres comprend la nouvelle valeur de priorité et un autre élément dans le tas. Étant donné que la nouvelle priorité est inférieure à l'ancienne, la fonction déplace l'élément vers le haut du tas pour maintenir l'intégrité de la structure du tas. Sinon, si la priorité du nouvel élément est supérieure à l'ancienne, la fonction déplace l'élément vers le bas du tas.

8.2.4. Supprimer l'élément de plus faible priorité dans le tas

La fonction 'removeElement' supprime l'élément de priorité minimale d'un tas tout en maintenant la propriété du tas. La fonction 'getElement' est utilisée pour récupérer l'élément de priorité minimale situé à la racine du tas. Après cela, l'élément est échangé avec le dernier élément du tas afin de retirer l'élément racine tout en préservant la structure du tas. Une fois cet échange effectué, la position de l'élément retiré est définie comme étant -1 dans le tableau des positions, indiquant ainsi son absence dans le tas. Ensuite, la fonction se focalise sur la

restauration du tas en déplaçant l'élément échangé (qui est désormais à la racine) vers le bas du tas. On accomplit cela en parcourant le tas à partir de la racine et en échangeant l'élément avec son enfant le plus petit si nécessaire, jusqu'à ce que l'élément soit correctement positionné, c'est-à-dire que sa priorité soit inférieure ou égale à celle de ses enfants. Après avoir rétabli la propriété du tas et positionné correctement l'élément, la fonction renvoie l'élément retiré, qui est celui avec la plus faible priorité dans le tas. En assurant l'intégrité et la validité du tas, cette approche garantit une suppression efficace de l'élément de priorité minimale.

9. Graphes

Les graphes permettent de représenter les liens entre différentes entités dans de nombreuses applications informatiques. Les parcours en profondeur (DFS) et en largeur (BFS) sont deux algorithmes fondamentaux pour explorer les graphes de manière efficace. Le DFS explore en profondeur une branche du graphe jusqu'à atteindre une impasse, utilisant une pile, et est utile pour détecter les cycles et les composantes fortement connexes. Le BFS explore les sommets par niveaux successifs, utilisant une file, et garantit de trouver le chemin le plus court dans un graphe non pondéré. Ces deux algorithmes ont une complexité temporelle de $O(V+E)$ et sont essentiels pour résoudre divers problèmes liés aux graphes, comme la recherche de chemins optimaux ou la vérification de propriétés structurelles.

9.1. Fonctions

9.1.1. Ajout d'une arête dans un graphe

La méthode "addEdgeInGraph" est utilisée pour ajouter une arête dans un graphe. Elle prend en compte le point de départ et le point d'arrivée de cette arête, ainsi que le graphe dans lequel cette arête doit être ajoutée. Cette fonction met à jour uniquement le tableau de listes d'adjacence pour ce graphe en ajoutant la destination à la liste d'adjacence du sommet de départ. Après avoir effectué cette action, cette fonction se termine immédiatement sans renvoyer de valeur.

9.1.2. Création du graphe

La fonction 'createGraph' est utilisée pour créer un graphe avec un nombre spécifié de sommets, éventuellement orienté, et une distance maximale entre les sommets (sigma). Cette fonction initialise les différentes composantes du graphe, notamment le tableau de listes d'adjacence, les coordonnées spatiales des sommets et les parents des sommets. Ensuite, on parcourt chaque sommet et pour chaque paire de sommets distincts, si la distance entre eux est inférieure à la valeur de sigma, alors on ajoute une arête entre ces deux sommets dans le graphe. Pour réaliser cette opération, on utilise la fonction 'addEdgeInGraph'. Enfin, on retourne le graphe ainsi créé.

9.1.3. Affichage du graphe

Pour l'affichage du graphe, comme demandé, on a fait un parcours des vertex pour afficher leurs coordonnées. Ensuite, on a parcouru les composants de la liste d'adjacence pour afficher les cellules voisines à chaque vertex, et pour ce faire, un pointeur de type Cell est utilisé pour parcourir la liste d'adjacence du sommet.

Vertex 1 : (0.834000,0.502000)	Vertex 0 : 28->17->NULL.
Vertex 2 : (0.229000,0.477000)	Vertex 1 : 35->NULL.
Vertex 3 : (0.360000,0.165000)	Vertex 2 : 45->41->39->37->36->14->7->4->NULL.
Vertex 4 : (0.235000,0.428000)	Vertex 3 : 44->33->29->27->18->NULL.
Vertex 5 : (0.350000,0.815000)	Vertex 4 : 45->39->37->14->7->2->NULL.
Vertex 6 : (0.299000,0.898000)	Vertex 5 : 48->46->40->31->19->6->NULL.
Vertex 7 : (0.356000,0.450000)	Vertex 6 : 48->46->19->15->12->5->NULL.
Vertex 8 : (0.076000,0.101000)	Vertex 7 : 45->41->39->32->4->2->NULL.
Vertex 9 : (0.573000,0.797000)	Vertex 8 : 49->22->10->NULL.
Vertex 10 : (0.090000,0.084000)	Vertex 9 : 40->NULL.
Vertex 11 : (0.994000,0.750000)	Vertex 10 : 49->22->8->NULL.
Vertex 12 : (0.171000,0.882000)	Vertex 11 : 38->NULL.
Vertex 13 : (0.645000,0.196000)	Vertex 12 : 48->46->15->6->NULL.
Vertex 14 : (0.126000,0.401000)	Vertex 13 : 21->20->NULL.
Vertex 15 : (0.246000,0.957000)	Vertex 14 : 45->37->4->2->NULL.
Vertex 16 : (0.652000,0.432000)	Vertex 15 : 46->12->6->NULL.
Vertex 17 : (0.812000,0.234000)	Vertex 16 : 23->NULL.
Vertex 18 : (0.262000,0.172000)	Vertex 17 : 28->21->0->NULL.
Vertex 19 : (0.399000,0.849000)	Vertex 18 : 44->33->27->3->NULL.
Vertex 20 : (0.600000,0.102000)	Vertex 19 : 40->6->5->NULL.
Vertex 21 : (0.665000,0.251000)	Vertex 20 : 34->30->13->NULL.
Vertex 22 : (0.000000,0.021000)	Vertex 21 : 17->13->NULL.
Vertex 23 : (0.702000,0.428000)	Vertex 22 : 10->8->NULL.
Vertex 24 : (0.123000,0.627000)	

Figure 67 Trace d'exécution de la fonction d'affichage du graphe

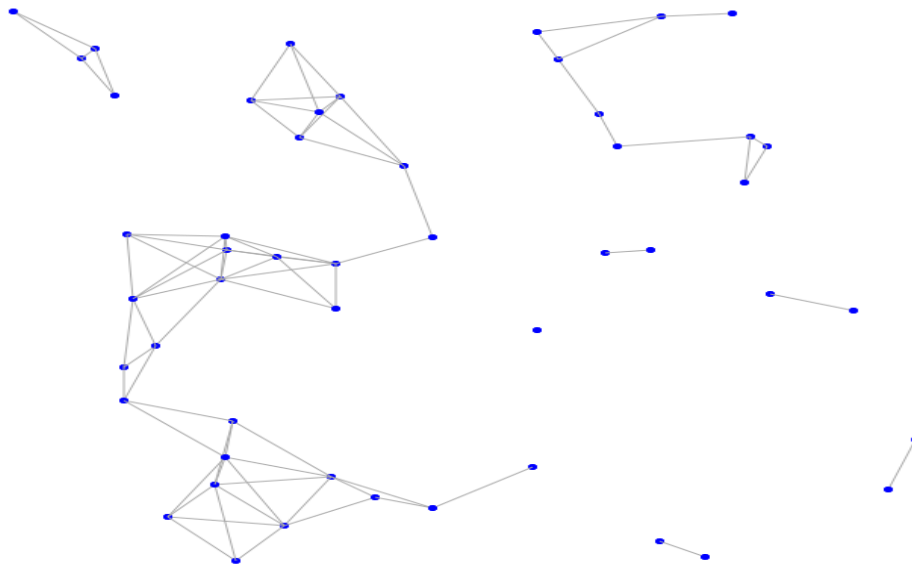


Figure 68 Résultat du graphe construit

9.1.4. Parcours en profondeur dans le graphe

La fonction 'graphDFS' implémente un parcours en profondeur (Depth-First Search) dans un graphe à partir d'un sommet spécifié. On utilise un tableau dynamiquement alloué pour suivre le statut de chaque sommet visité, initialisant chaque élément à 0 pour indiquer qu'aucun sommet n'a été visité. Ensuite, on crée une pile pour stocker les sommets à explorer, marquant le sommet de départ comme visité et l'ajoutant à la pile.

Tant que la pile n'est pas vide, on extrait le sommet courant de la pile, on l'affiche, puis on parcourt ses voisins. Pour chaque voisin non visité, on le marque comme visité, on le rajoute à la pile et on met à jour le tableau des parents pour refléter la relation parent-enfant dans l'arborescence de parcours en profondeur. Une fois que tous les voisins du sommet courant ont été explorés, le sommet courant est retiré de la pile.

Ce processus se répète jusqu'à ce que tous les sommets accessibles aient été visités. Enfin, la mémoire allouée pour le tableau de suivi des sommets est libérée. Cette fonction permet d'explorer efficacement le graphe en profondeur, offrant ainsi une méthode pour découvrir ses composants connectés et générer un arbre de parcours en profondeur.

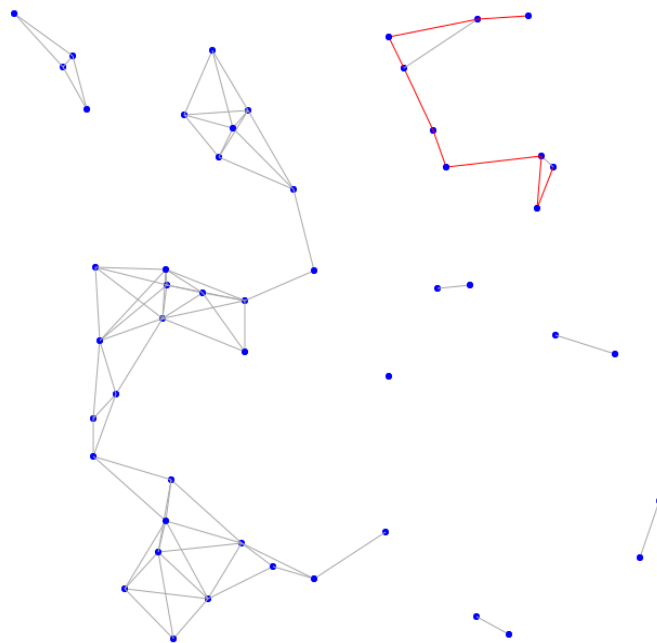


Figure 69 Graphe du parcours en profondeur à partir d'un nœud de départ

9.1.5. Parcours en largeur dans le graphe

La fonction 'graphBFS' implémente un parcours en largeur (Breadth-First Search) dans un graphe à partir d'un sommet spécifié. On alloue dynamiquement de la mémoire pour stocker le sommet courant extrait de la file, ainsi qu'un tableau usedVertices pour suivre les sommets déjà visités. Initialement, la taille du tableau 'usedVertices' est définie à -1.

Ensuite, on crée une file pour stocker les sommets à explorer, en plaçant le sommet de départ dedans. On ajoute également le sommet de départ au tableau 'usedVertices'.

Tant que la file n'est pas vide, on extrait un sommet de la file et on l'affiche. Ensuite, on explore ses voisins. Pour chaque voisin non visité, on l'ajoute à la file, on le marque comme visité en l'ajoutant au tableau 'usedVertices', et on met à jour le tableau des parents pour refléter la relation parent-enfant dans l'arborescence de parcours en largeur. On répète ce processus jusqu'à ce que tous les sommets accessibles aient été visités.

Finalement, on libère la mémoire allouée pour le pointeur 'current'. Cette fonction offre une manière efficace d'explorer le graphe en largeur, permettant de découvrir ses composants connectés et de générer un arbre de parcours en largeur.

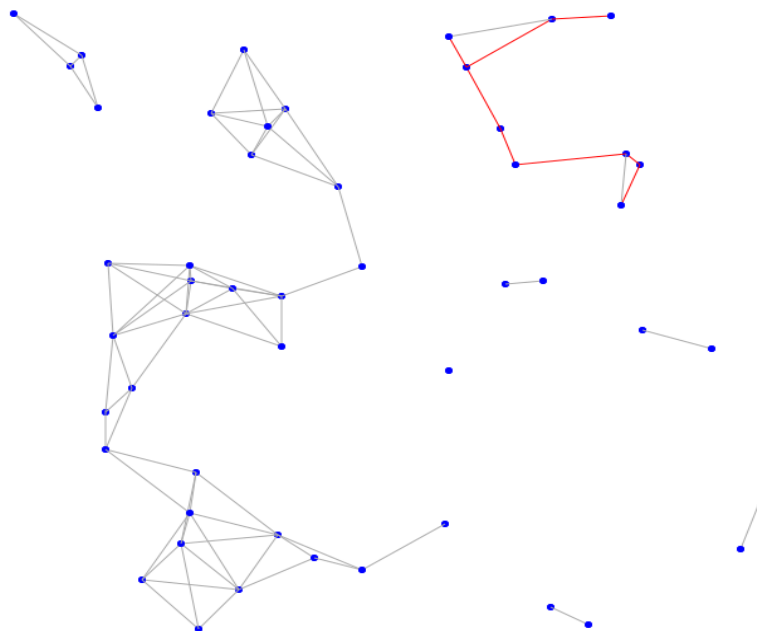


Figure 70 Graphe du parcours en largeur à partir d'un noeud de départ

9.1.6. Calcul du nombre de composantes connexes

La fonction 'numberOfComponents' permet de calculer le nombre de composantes connexes dans un graphe donné. On alloue dynamiquement de la mémoire pour un tableau 'visited' de taille égale au nombre de sommets du graphe, ainsi qu'un tableau parents pour stocker les parents des sommets. Initialement, tous les sommets sont marqués comme non visités (0) et leurs parents sont définis à -1.

Ensuite, on initialise le compteur de composantes à 0. On parcourt ensuite tous les sommets du graphe. Si un sommet n'a pas été visité, on incrémente le compteur de composantes, puis on utilise le parcours en largeur (graphBFS) à partir de ce sommet pour visiter tous les sommets de sa composante connexe.

Une fois que tous les sommets ont été visités, on libère la mémoire allouée pour le tableau visited. Enfin, on retourne le nombre total de composantes connexes dans le graphe.

Cette fonction offre une manière efficace de déterminer le nombre de composantes connexes dans un graphe, ce qui est utile pour comprendre sa structure et ses propriétés.

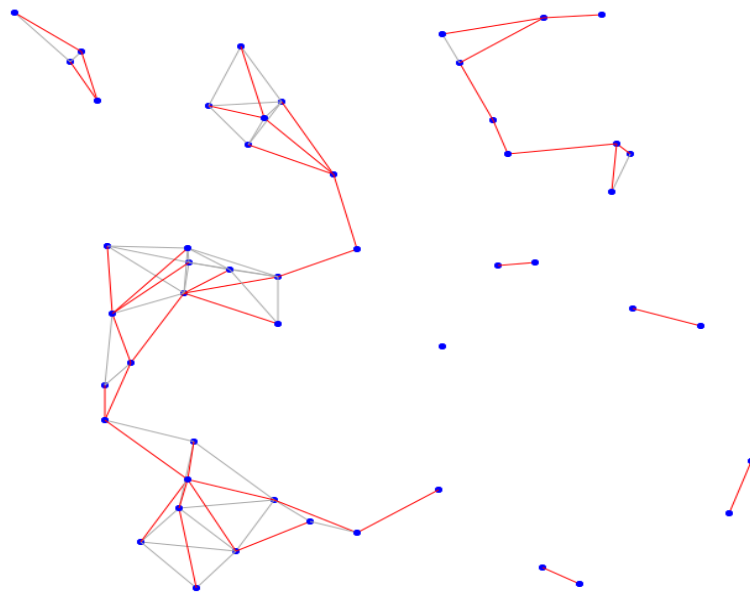


Figure 71 Graphe avec ses composantes connexes en rouge

10. Arbre couvrant minimum (PRIM)

10.1. Introduction

L'algorithme de Prim est un algorithme glouton classique utilisé pour trouver l'arbre couvrant minimal (ACM) d'un graphe pondéré non orienté. L'arbre couvrant minimal est un sous-ensemble des arêtes qui connecte tous les sommets du graphe sans former de cycles et avec le poids total des arêtes le plus faible possible. L'algorithme de Prim commence avec un seul sommet et développe l'ACM une arête à la fois, choisissant toujours la plus petite arête qui connecte un sommet à l'intérieur de l'ACM à un sommet à l'extérieur de l'ACM. Cela garantit que l'arbre reste acyclique, inclut tous les sommets et maintient le poids minimal des arêtes à chaque étape.

10.2. Fonction Prim

La fonction Prim implémente l'algorithme de Prim pour calculer l'arbre couvrant minimal d'un graphe donné g , en commençant par un sommet initial s . L'algorithme utilise une structure de tas pour gérer les priorités des sommets en fonction de la distance minimale depuis l'arbre couvrant en cours de construction. Au début, tous les sommets sont insérés dans le tas avec une priorité maximale (infinie), sauf le sommet de départ s qui reçoit une priorité de zéro. Ensuite, l'algorithme itère jusqu'à ce que le tas soit vide : à chaque étape, il extrait le sommet avec la plus petite priorité (la plus courte distance) et met à jour les priorités des sommets adjacents si une connexion plus courte est trouvée via le sommet extrait. Les parentés des sommets sont également mises à jour pour refléter la structure de l'ACM en construction.

10.2.1. Traces d'exécution

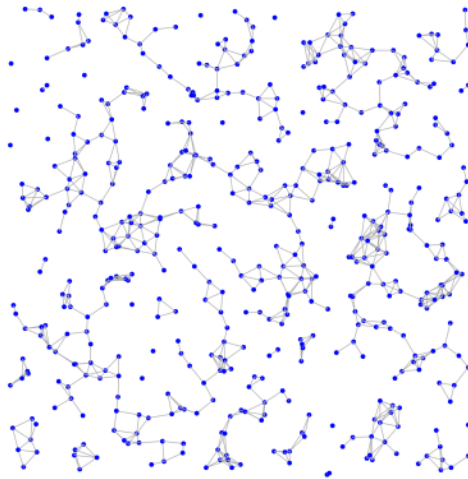


Figure 72 Graphe pour l'exemple Prim

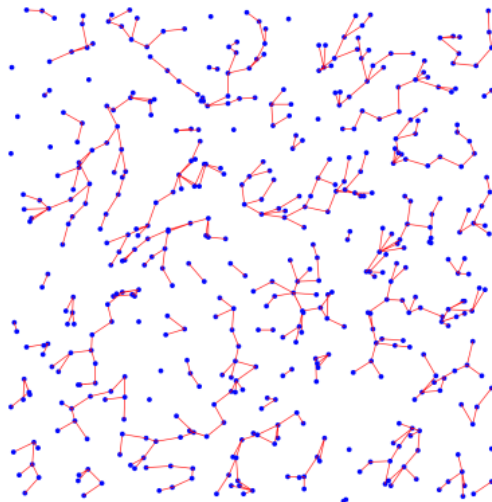


Figure 73 ACM via Prim

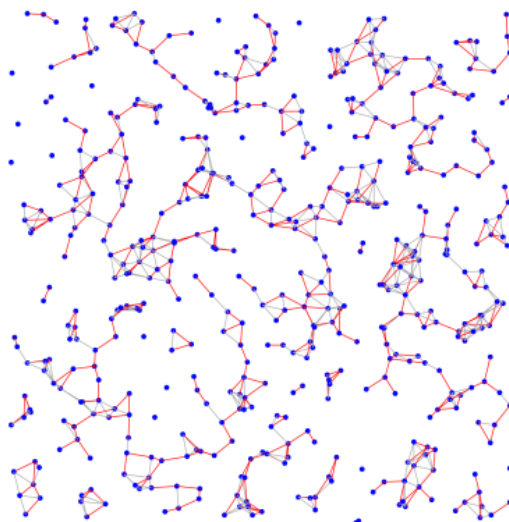


Figure 74 Graphe muni de son ACM grâce à Prim

Vertex 150	: (0.340324,0.826504)	Vertex 403 : 371 → 37 → NULL
Vertex 151	: (0.045542,0.424118)	Vertex 404 : 235 → 234 → 234 → NULL
Vertex 152	: (0.153031,0.988214)	Vertex 405 : 171 → 162 → NULL
Vertex 153	: (0.904451,0.109123)	Vertex 406 : 435 → 430 → NULL
Vertex 154	: (0.030746,0.750447)	Vertex 407 : 385 → 357 → 37 → NULL
Vertex 155	: (0.754758,0.222931)	Vertex 408 : 385 → 385 → NULL
Vertex 156	: (0.793789,0.207390)	Vertex 409 : 430 → 430 → 38 → NULL
Vertex 157	: (0.600133,0.441745)	Vertex 410 : 402 → 37 → NULL
Vertex 158	: (0.400039,0.459342)	Vertex 411 : 380 → 325 → NULL
Vertex 159	: (0.158807,0.077095)	Vertex 412 : 404 → 380 → 41 → 37 → 8 → NULL
Vertex 160	: (0.730744,0.615236)	Vertex 413 : 400 → 400 → 32 → 37 → 1 → NULL
Vertex 161	: (0.279403,0.930147)	Vertex 414 : 354 → 361 → 37 → NULL
Vertex 162	: (0.972710,0.340865)	Vertex 415 : 355 → 35 → 37 → NULL
Vertex 163	: (0.924650,0.592721)	Vertex 416 : 470 → 470 → 430 → 444 → 370 → 214 → NULL
Vertex 164	: (0.855565,0.472742)	Vertex 417 : 371 → 327 → NULL
Vertex 165	: (0.368938,0.739598)	Vertex 418 : 439 → 439 → 138 → NULL
Vertex 166	: (0.428428,0.585241)	Vertex 419 : 435 → NULL
Vertex 167	: (0.142935,0.311597)	Parents : 1 223 340 35 241 61 52 232 1 249 344 344 236 304 471 19 47 402 1 211 1 211 286 447 271 339 68 131 259 382
Vertex 168	: (0.004510,0.806901)	272 125 126 346 9 215 12 176 396 352 244 327 483 294 371 94 175 16 271 477 413 413 112 278 62 147 434 45 351 138 310 8
Vertex 169	: (0.591280,0.644049)	2 241 25 36 443 8 76 238 406 482 423 258 477 254 43 287 352 134 346 388 228 381 147 46 1 64 138 1 52 34 247 33 254 1
Vertex 170	: (0.539022,0.345091)	57 125 125 234 483 30 113 36 186 27 245 267 259 84 336 113 41 88 388 160 300 331 100 329 138 231 127 266 486 96 262
Vertex 171	: (0.947182,0.292580)	258 444 48 448 134 258 485 130 262 265 344 285 226 62 454 235 345 131 261 35 634 76 58 354 427 136 121 235 255 238
Vertex 172	: (0.387682,0.765017)	427 68 63 135 482 132 251 336 445 355 221 134 338 485 355 46 425 323 37 1 96 233 461 382 125 34 1 146 216 268 496 13
Vertex 173	: (0.643058,0.070750)	4 261 184 421 51 121 238 125 42 421 421 365 238 11 288 48 89 4 235 1 91 266 21 288 210 478 284 42 435 64 62 382 114
Vertex 174	: (0.688491,0.467395)	52 218 1 64 268 21 211 1 124 233 235 238 238 42 77 6 211 43 179 68 483 481 365 478 254 255 13 236 134 447 128 247 33
Vertex 175	: (0.504915,0.101665)	8 1 113 485 1 271 355 381 411 4 385 268 174 138 328 481 238 485 334 246 248 19 363 36 236 261 32 125 488 215 247 447
Vertex 176	: (0.686261,0.995318)	427 363 228 68 442 236 221 43 821 458 1 458 258 681 271 127 288 383 471 115 482 421 121 146 478 13 187 422 288 386 488
Vertex 177	: (0.389082,0.733789)	271 68 238 172 84 172 138 138 367 13 86 115 484 117 288 244 271 387 51 325 459 432 1 488 347 306 265 378 36 288 213 14
Vertex 178	: (0.794722,0.887364)	5 634 286 481 180 363 274 428 128 11 454 287 128 111 340 243 65 264 51 189 361 465 467 485 172 286 271 478 225 226 77 4
Vertex 179	: (0.927922,0.592656)	81 307 125 397 288 138 354 355 381 1 264 485 171 186 13 138 283 1 121 235 264 174 384 180 487 288 114 445 485 486 118
Vertex 180	: (0.766345,0.964152)	7 224 41 238 348 121 287 387 113 35 121 188 486 124 266 287 85 225 114 288 485 35 151 1 4 114 121 71 263 278 18
Vertex 181	: (0.509432,0.024788)	8 216 188 51 265 29 446 488 377 265 33 118 138 381 13 475 488 288 488 1 243 51 113 219 1 236 129 476 488 245 425 111 2
Vertex 182	: (0.613695,0.378482)	82 478 453 341 246 136 184 482 27 124 442 435 12 210 343 74 388 12 71 181 238 478 221 458 481
Vertex 183	: (0.802563,0.672356)	

Figure 75 Traces d'exécutions

11. Graphes orientés acycliques

11.1. Introduction

Un graphe orienté acyclique, ou DAG (Directed Acyclic Graph), est un graphe orienté ne contenant aucun cycle. Intuitivement, un DAG est composé de sommets « source » sans arête entrante, de sommets « puits » sans arête sortante, et de sommets internes qui servent de chemin entre les sommets « source » et les sommets « puits ». Les DAG sont omniprésents dans de nombreux domaines tels que la généalogie, la science des données, la planification, et l'informatique.

11.2. Fonctions

11.2.1. `topologicalSearch`

La fonction `topologicalSort` vise à effectuer un tri topologique sur le graphe. La démarche commence par initialiser une variable pour suivre le nombre de sommets visités. Pour chaque sommet du graphe, si le sommet n'a pas de prédécesseur (indiqué par une valeur particulière), on initie une recherche en profondeur (DFS) en utilisant une pile. On marque le sommet comme visité et on pousse ce sommet dans la pile DFS. Tant que la pile n'est pas vide, on examine le sommet au sommet de la pile. Si ce sommet a des successeurs non visités, on les marque comme visités et on les pousse dans la pile. Si tous les successeurs ont été visités, on retire le sommet de la pile, on l'ajoute à l'ordre topologique, et on continue jusqu'à ce que tous les sommets soient traités. Le résultat est un ordre dans lequel chaque sommet apparaît avant ses successeurs, stocké dans le tableau `topological_ordering` du graphe.

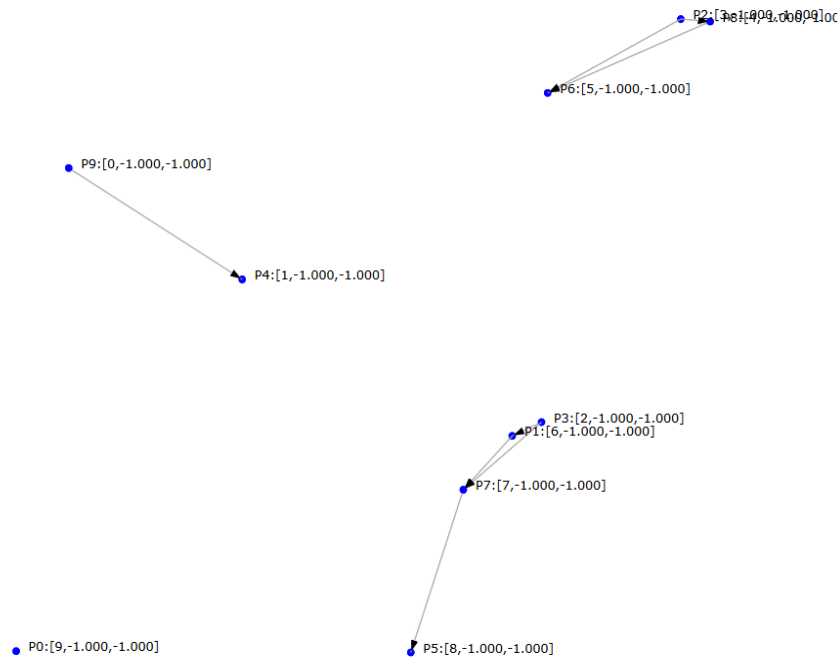


Figure 76 Exemple d'un DAG

11.2.2. computeEarliestStartDates

La fonction `computeEarliestStartDates` calcule la date de début la plus précoce pour chaque sommet en suivant l'ordre topologique du graphe. On commence par initialiser la date de début du premier sommet de l'ordre topologique à 0. Pour chaque sommet suivant dans cet ordre, on calcule la date de début la plus précoce en considérant les dates de début des prédécesseurs de ce sommet et les distances pondérées des arêtes les reliant. Pour chaque prédécesseur, on ajoute la distance pondérée à sa date de début pour obtenir une date possible pour le sommet en cours. La date de début la plus précoce pour le sommet en cours est le maximum de ces dates possibles. Cette valeur est stockée dans le tableau `earliest_start` du graphe.

11.2.3. computeLatestStartDates

La fonction `computeLatestStartDates` calcule la date de début la plus tardive pour chaque sommet, en partant de la fin de l'ordre topologique. On commence par fixer la date de début la plus tardive du dernier sommet (un puits) au maximum des dates de début les plus précoces des puits. Ensuite, pour chaque sommet précédent dans l'ordre topologique inversé, on calcule la date de début la plus tardive en considérant les dates de début des successeurs de ce sommet et les distances pondérées des arêtes les reliant. Pour chaque successeur, on soustrait

la distance pondérée de sa date de début pour obtenir une date possible pour le sommet en cours. La date de début la plus tardive pour le sommet en cours est le minimum de ces dates possibles. Cette valeur est stockée dans le tableau `latest_start` du graphe.

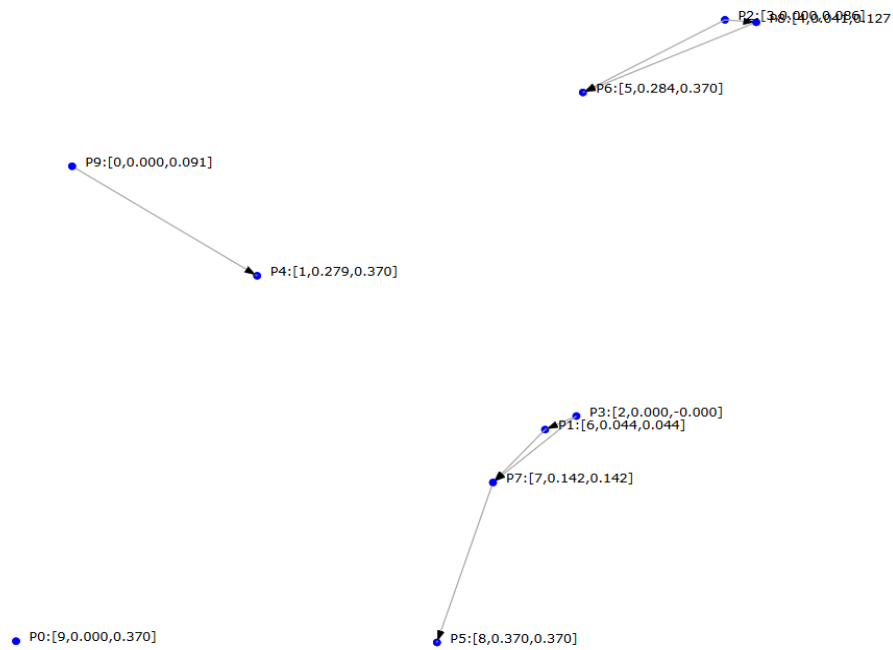


Figure 77 DAG avec dates au plus tôt et au plus tard

11.3. Complexité

Notons V le nombre de sommets et E le nombre d'arête du graphe.

1. `topologicalSort` :

- Parcours en profondeur (DFS) : La complexité de l'algorithme DFS dépend du nombre de sommets et du nombre d'arêtes du graphe. Pour chaque sommet, le DFS explore ses voisins non visités. Dans le pire des cas, chaque arête sera parcourue exactement deux fois (une fois pour chaque extrémité). Ainsi, la complexité de l'algorithme DFS est de $O(V + E)$, où V est le nombre de sommets et E est le nombre d'arêtes dans le graphe.
- Boucle externe : La boucle externe de `topologicalSort` itère sur chaque sommet du graphe, ce qui prend également $O(V)$ dans le pire des cas.

2. `computeEarliestStartDates` :

- a. Boucle externe : La boucle externe itère sur chaque sommet du graphe, ce qui prend $O(V)$ dans le pire des cas.
 - b. Boucle interne : À chaque itération de la boucle externe, une boucle interne parcourt les prédécesseurs du sommet actuel. Dans le pire des cas, chaque sommet a des prédécesseurs dans le graphe, donc cette boucle a une complexité de $O(E)$.
3. computeLatestStartDates :
- a. Boucle externe : La boucle externe itère sur chaque sommet du graphe, ce qui prend $O(V)$ dans le pire des cas.
 - b. Boucle interne : À chaque itération de la boucle externe, une boucle interne parcourt les successeurs du sommet actuel. Dans le pire des cas, chaque sommet a des successeurs dans le graphe, donc cette boucle a une complexité de $O(E)$.

En résumé, chaque fonction présente une complexité temporelle linéaire ou quasi-linéaire par rapport au nombre de sommets (V) et au nombre d'arêtes (E) du graphe. La complexité exacte dépendra de la structure spécifique du graphe et des opérations effectuées à chaque itération.

11.4. Traces d'exécution

```
-----
Print of the graph
Number of vertices : 15
Sigma : 0.300000
Coordinates of the vertices
Vertex 0 : (0.981750,0.923002)
Vertex 1 : (0.416425,0.124485)
Vertex 2 : (0.514756,0.061892)
Vertex 3 : (0.983154,0.882382)
Vertex 4 : (0.642537,0.148198)
Vertex 5 : (0.293680,0.306711)
Vertex 6 : (0.108219,0.115146)
Vertex 7 : (0.560686,0.223090)
Vertex 8 : (0.544969,0.233650)
Vertex 9 : (0.960051,0.637410)
Vertex 10 : (0.833979,0.663320)
Vertex 11 : (0.103305,0.944060)
Vertex 12 : (0.605121,0.575549)
Vertex 13 : (0.793146,0.284341)
Vertex 14 : (0.286355,0.630268)
Adjacency lists
Vertex 0 : NULL
Vertex 1 : 8 -> 7 -> 5 -> 4 -> NULL
Vertex 2 : 8 -> 7 -> 4 -> 1 -> NULL
Vertex 3 : 0 -> NULL
Vertex 4 : 13 -> 8 -> 7 -> NULL
Vertex 5 : NULL
Vertex 6 : 5 -> NULL
Vertex 7 : 13 -> 8 -> 5 -> NULL
Vertex 8 : 13 -> 5 -> NULL
Vertex 9 : 10 -> 3 -> 0 -> NULL
Vertex 10 : 3 -> 0 -> NULL
Vertex 11 : NULL
Vertex 12 : 10 -> NULL
Vertex 13 : NULL
Vertex 14 : NULL
Parents: -1 -1 -1 -1 1 8 -1 1 1 -1 9 -1 -1 8 -1
```

```
Topological ordering:
[14 12 11 9 10 6 3 2 1 4 7 8 5 13 0 ]
Print start dates:
Vertex 0 : [earliest start date=0.551, latest start date=0.735 ]
Vertex 1 : [earliest start date=0.117, latest start date=0.117 ]
Vertex 2 : [earliest start date=0.000, latest start date=0.000 ]
Vertex 3 : [earliest start date=0.510, latest start date=0.695 ]
Vertex 4 : [earliest start date=0.344, latest start date=0.344 ]
Vertex 5 : [earliest start date=0.735, latest start date=0.735 ]
Vertex 6 : [earliest start date=0.000, latest start date=0.469 ]
Vertex 7 : [earliest start date=0.455, latest start date=0.455 ]
Vertex 8 : [earliest start date=0.474, latest start date=0.474 ]
Vertex 9 : [earliest start date=0.000, latest start date=0.301 ]
Vertex 10 : [earliest start date=0.245, latest start date=0.430 ]
Vertex 11 : [earliest start date=0.000, latest start date=0.735 ]
Vertex 12 : [earliest start date=0.000, latest start date=0.185 ]
Vertex 13 : [earliest start date=0.727, latest start date=0.735 ]
Vertex 14 : [earliest start date=0.000, latest start date=0.735 ]
```

Figure 78 Trace d'exécutions pour topologiccal ordering

CONCLUSION

La réalisation de ces travaux pratiques nous a permis de nous familiariser avec diverses structures de données et leurs algorithmes associés, élément fondamental pour tout informaticien souhaitant optimiser le traitement de l'information. En implémentant successivement les listes, piles, files, arbres binaires de recherche (ABR), arbres rouge-noir, tables de hachage, graphes, graphes acycliques dirigés (DAG), tas et arbres couvrants minimaux, nous avons acquis une compréhension approfondie des principes sous-jacents à chaque structure.

Chaque TP a été conçu pour respecter les dépendances entre les structures, nous permettant de construire notre savoir de manière progressive et cohérente. Cette approche nous a offert une vision globale de l'interaction entre différentes structures de données et de leur application dans divers contextes algorithmiques.

L'implémentation des ABR, par exemple, nous a initiés aux concepts de récursivité et d'équilibrage d'arbres, tandis que les arbres rouge-noir ont approfondi notre compréhension des techniques d'équilibrage automatique. Les structures de hachage ont illustré l'efficacité de l'accès direct et des opérations de recherche, tandis que la manipulation des graphes et DAG a mis en lumière les algorithmes de parcours et de recherche de chemins optimaux.

En somme, ce parcours à travers les différentes structures de données a non seulement enrichi notre bagage technique mais nous a également préparés à aborder des problématiques algorithmiques plus complexes avec une approche méthodique et optimisée. Ce voyage à travers l'algorithmique avancée nous laisse ainsi mieux armés pour les défis futurs en informatique.

TABLE DES FIGURES

Figure 1 Organisation des Tps et leurs dépendances	6
Figure 2 Structures de données Cell et List	7
Figure 3 Exemple de suppression d'une clé donnée dans une liste	10
Figure 4 Tests exhaustifs pour la structure de données List	11
Figure 5 Tests unitaires List	11
Figure 6 Structure représentant la notion de table de hachage	12
Figure 7 Exemple d'une table de hachage	13
Figure 8 Insertion d'un couple dans une table de hachage	14
Figure 9 Affichage CountDistinctWordsInBook	16
Figure 10 Structure de données Pile et fonctionnement	17
Figure 11 Empiler une valeur	18
Figure 12 Dépiler le haut de d'une pile	18
Figure 13 Tests exhaustifs pour la structure de donnée Pile	19
Figure 14 Illustration et fonctionnement d'une file	20
Figure 15 Structure de file en langage C	20
Figure 16 File vide	20
Figure 17 Insertion en queue de file 1/2	21
Figure 18 Insertion en queue de file 2/2	22
Figure 19 Tests exhaustifs File	23
Figure 20 Exemple d'un ABR	24
Figure 21 Structure représentant un nœud d'un ABR et définition d'un ABR	25
Figure 22 Insertion dans un ABR vide	26
Figure 23 Algorithme d'insertion au sein d'un ABR	27
Figure 24 Complexité d'insertion dans un ABR	27
Figure 25 Exemple insertion ABR	28
Figure 26 Algorithme de recherche au sein d'un ABR	30
Figure 27 Complexité de la recherche au sein d'un ABR	30
Figure 28 Algorithme de suppression d'un nœud au sein d'un ABR	31
Figure 29 Tests exhaustifs ABR	33
Figure 30 Résultats sur distributions uniformes et non uniformes pour ABR	34
Figure 31 Temps de construction d'un ABR en fonction de sa taille (cas uniforme et non uniforme)	35
Figure 32 Hauteur d'un ABR en fonction de sa taille (cas uniforme et non uniforme)	35
Figure 33 Temps pour effectuer 1000 recherches dans un ABR en fonction de sa taille (cas uniforme et non uniforme)	36
Figure 34 Définition de la structure d'un RBST	37

Figure 35 Algorithme taille d'un RBST	38
Figure 36 Exemple de coupe d'un RBST	39
Figure 37 Cas 1 splitRBST	39
Figure 38 Cas 2 splitRbst	40
Figure 39 Cas 3 splitRBST	40
Figure 40 Illustration de la fonction insertAtRoot	41
Figure 41 Exemple d'insertion dans un RBST	41
Figure 42 Tests exhaustifs pour RBST	43
Figure 43 Résultats sur distributions uniformes et non uniformes pour RBST	44
Figure 44 Hauteur RBST et ABR en fonction de leur taille (cas non uniforme)	45
Figure 45 Temps pour effectuer 1000 recherches dans un RBST et un ABR en fonction de leur taille (cas non uniforme)	45
Figure 46 Temps pour construire un ABR et un RBST en fonction de leur taille (cas non uniforme)	46
Figure 47 Hauteur RBST et ABR en fonction de leur taille (cas uniforme)	46
Figure 48 Temps pour effectuer 1000 recherches dans un RBST et un ABR en fonction de leur taille (cas uniforme)	47
Figure 49 Temps pour construire un ABR et un RBST en fonction de leur taille (cas uniforme)	47
Figure 50 Structure d'un ARN	49
Figure 51 Exemple d'un ARN	50
Figure 52 Rotation autour d'un nœud ARN	51
Figure 53 Exemple de rotation au sein d'un arbre	51
Figure 54 Cas 1 équilibrage ARN	52
Figure 55 Cas 2-1 équilibrage ARN	53
Figure 56 Cas 2-2 équilibrage ARN	54
Figure 57 Cas 2-3 équilibrage ARN	55
Figure 58 Tests exhaustifs ARN	57
Figure 59 Résultats sur distributions uniformes et non uniformes pour ARN, ABR, RBST	58
Figure 60 Temps pour construire ARN, RBST, ABR en fonction de leur taille (cas uniforme)	58
Figure 61 Hauteur d'un ARN, RBST, ABR en fonction de leurs tailles (cas uniforme)	59
Figure 62 Temps pour effectuer 1000 recherches dans un RBST, ARN et un ABR en fonction de leurs tailles (cas uniforme)	59
Figure 63 Temps pour construire ARN, RBST, ABR en fonction de leur taille (cas non uniforme)	60
Figure 64 Hauteur d'un ARN, RBST, ABR en fonction de leur taille (cas non uniforme)	60
Figure 65 Temps pour effectuer 1000 recherches dans un RBST, ARN et un ABR en fonction de leurs tailles (cas non uniforme)	61
Figure 66 Tests exhaustifs Heap	63
Figure 67 Trace d'exécution de la fonction d'affichage du graphe	66
Figure 68 Résultat du graphe construit	66

Figure 69 Graphe du parcours en profondeur à partir d'un nœud de départ	67
Figure 70 Graphe du parcours en largeur à partir d'un noeud de départ	68
Figure 71 Graphe avec ses composantes connexes en rouge	69
Figure 72 Graphe pour l'exemple Prim	71
Figure 73 ACM via Prim	71
Figure 74 Graphe muni de son ACM grâce à Prim	71
Figure 75 Traces d'exécutions	72
Figure 76 Exemple d'un DAG	74
Figure 77 DAG avec dates au plus tôt et au plus tard	75
Figure 78 Trace d'exécutions pour topologiccal ordering	77

REFERENCES

- [1] Cours de Loïck Lhote pour de nombreuses illustrations, explications et implémentations
- [2] Aide de Patrick Ducrot lors des séances de TP



Ecole Publique d'Ingénieurs en 3 ans

6 boulevard Maréchal Juin, CS 45053
14050 CAEN cedex 04

