Ecole Publique d'ingénieures et d'ingénieurs en 3 ans

Rapport

ALGORITHMIQUE AVANCÉE

Le 30 avril 2024,

Mohamed Toujani,
1A
mohamed.toujani@ecole.ensicaen.fr

Tuteur école : Patrick DUCROT

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

TABLE DES MATIERES

LISTE

FILE DE PRIORITÉ

TABLE DE HACHAGE

PILE

FILE

BST

RBST

ARBRES ROUGES NOIRS

GRAPHE

GRAPHES ORIENTÉS ACYCLIQUES

ARBES COUVRANTS MINIMUMS

TABLE DES FIGURES

TABLE DES TABLEAUX

TABLE DES COURBES

TABLE DES GRAPHES

LISTES

Dans le cadre de ce TP, j'ai travaillé sur l'implémentation du type abstrait liste (List) en langage C. Les listes sont des structures de données fondamentales qui peuvent être utilisées pour diverses applications telles que la création de piles, de files, de graphes ou encore de tables de hachage. Mon objectif était d'implémenter six fonctions essentielles pour manipuler efficacement les listes.

1. Implémentation des listes :

Pour commencer, j'ai défini une liste comme un pointeur sur la première cellule de la liste, ou NULL si la liste est vide. Chaque cellule de la liste est composée d'une clé (une chaîne de caractères ou un string), d'une valeur (un entier) et d'un pointeur vers la cellule suivante.

Pour réaliser ce TP, j'ai travaillé sur trois fichiers :

list.h : Ce fichier contient toutes les déclarations nécessaires pour les listes, y compris les structures de cellule et de liste, ainsi que les prototypes de fonctions associées. Il n'était pas nécessaire de le modifier.

list.c : Ce fichier contient l'implémentation des fonctions associées aux listes. C'est dans ce fichier que j'ai apporté mes modifications pour implémenter les six fonctions demandées.

testList.c : Ce fichier contient un programme de test pour évaluer le bon fonctionnement des fonctions sur les listes. J'avais la liberté de le modifier selon mes besoins.

J'ai réussi à implémenter avec succès les six fonctions suivantes sur les listes :

1.1. newList() :

J'ai créé une fonction qui retourne une liste vide.

1.2. freeList(List L) :

J'ai implémenté une fonction qui libère la mémoire utilisée par une liste.

1.3. printList(List L, int type) :

J'ai mis en place une fonction pour afficher le contenu d'une liste, en fonction du type spécifié.

1.4. findKeyInList (List L, string key) :

J'ai développé une fonction qui cherche une clé dans une liste et retourne un pointeur vers la première cellule contenant cette clé, ou NULL si la clé n'est pas trouvée.

1.5. delKeyInList (List L, string key) :

J'ai créé une fonction qui supprime la première occurrence d'une clé dans une liste et renvoie la liste modifiée.

1.6. addKeyValueInList (List L, string key, int value) :

J'ai mis en place une fonction pour ajouter une paire clé-valeur au début d'une liste.

2. Complexité des fonctions :

newList	O(1)
freeList	O(n)
printList	O(n)
findKeyInList	O(n)
delKeyInList	O(n)
addKeyValueInList	O(1)

Tableau 1 Complexité des fonctions de Liste

Où n est le nombre des éléments dans la liste.

3. Trace d'exécution :

```
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/list$ make clean
rm -f *.o
rm -f *~
rm -f testList
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/list$ make all
gcc -o list.o -c list.c -W -Wall -ansi -pedantic -g
gcc -o testList.o -c testList.c -W -Wall -ansi -pedantic -g
gcc -o testList list.o testList.o -lm
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/list$
```

Figure 1 Exécution Correcte sans warning de List

```
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/list$ ./testList
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 2 Trace d'exécution de Liste

4. Unit Tests :

```
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024$ cd unit_tests/list/
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/unit_tests/list$ make clean
rm -f unit_test_list.o ../../list/list.o
rm -f *~
rm -f unittest
rm -f log-*
rm -f score-*
rm -f out.txt
rm -f output.txt
rm -f score.log
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/unit_tests/list$ make all
gcc -o unit_test_list.o -c unit_test_list.c -W -Wall -g
gcc -o ../../list/list.o -c ../../list/list.c -W -Wall -g
gcc -o unittest unit_test_list.o ../../list/list.o ../../list/list.h
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/unit_tests/list$
```

Figure 3 Execution Correct sans warning de unit tests.

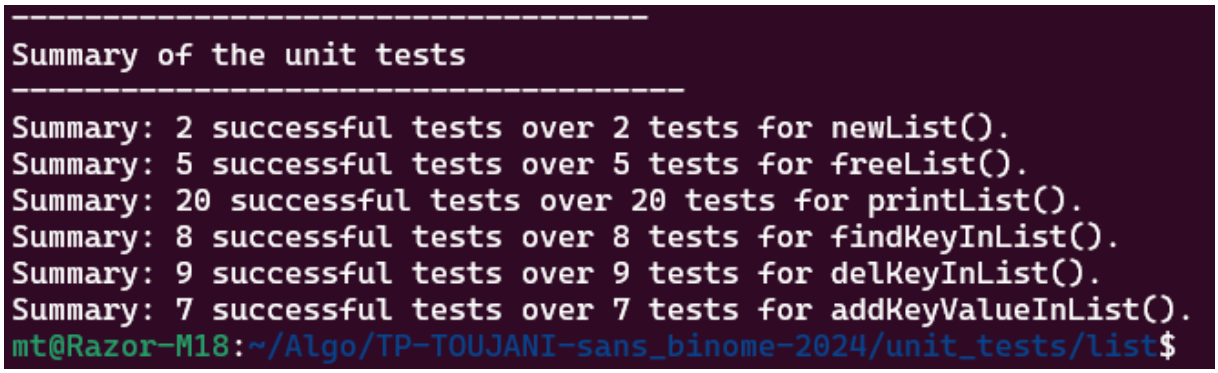


Figure 4 Summary of unit tests of List.

5. Manipulation des chaînes de caractères :

Pour manipuler les chaînes de caractères, j'ai utilisé la bibliothèque string.h. J'ai veillé à allouer correctement la mémoire avant de stocker ou manipuler les chaînes de caractères.

Pour une manipulation efficace des chaînes de caractères en C, j'ai consulté les fonctions disponibles dans la bibliothèque string à l'adresse suivante : lien vers la documentation de la bibliothèque string.

6. Conclusion :

Ce TP m'a permis de renforcer mes compétences en structures de données fondamentales en informatique, en particulier les listes. J'ai appris à manipuler efficacement les listes en implémentant différentes fonctions pour les créer, les modifier et les afficher. La manipulation des chaînes de caractères en C m'a également permis de comprendre l'importance de l'allocation dynamique de mémoire et la terminaison correcte des chaînes.

FILE DE PRIORITE ET TAS

L'objectif de ce TP était d'implémenter une structure de file de priorité à l'aide d'un tas binaire. La file de priorité permet de stocker des éléments associés à une priorité et offre des opérations telles que l'insertion d'un élément respectant sa priorité, la récupération de l'élément avec la plus faible priorité, la suppression de l'élément avec la plus faible priorité et la modification de la priorité d'un élément.

7. Implémentation de la file de priorité :

Pour implémenter la file de priorité, j'ai utilisé une structure de tas binaire. Les tas sont des arbres binaires complets où la priorité de tout nœud est inférieure ou égale à celle de ses descendants. Cette structure de tas a été représentée à l'aide d'un tableau, où chaque élément correspond à une clé, sa priorité est stockée dans un tableau distinct, et sa position dans le tas est suivie dans un autre tableau.

Dans les fichiers **heap.c** et **heap.h**, j'ai trouvé la déclaration de la structure de tas et les prototypes des fonctions nécessaires à la manipulation du tas. Mon travail consistait à implémenter les fonctions suivantes :

7.1. createHeap(int n) :

Cette fonction crée un tas initialement vide où les clés sont dans l'intervalle [1,n].

7.2. printHeap(Heap h) :

Cette fonction affiche les données du tas dans la console, montrant les informations sur les clés, les priorités et la structure du tas.

7.3. getElement(Heap h) :

Cette fonction retourne l'élément avec la plus faible priorité dans le tas.

7.4. insertHeap (Heap h, int element, double priority) :

Cette fonction insère un nouvel élément dans le tas avec la priorité spécifiée.

7.5.modifyPriorityHeap (Heap h, int element, double priority) :

Cette fonction modifie la priorité d'un élément existant dans le tas avec une nouvelle priorité.

7.6.removeElement (Heap h) :

Cette fonction supprime l'élément avec la plus faible priorité dans le tas et le retourne.

8. Complexité des fonctions :

createHeap	O(n), n est la taille du tas.
printHeap	O(n), n est le nombre d'éléments dans le tas.
getElement	O(1)
insertHeap	O(log n), n est le nombre d'éléments dans le tas.
modifyPriorityHeap	O(log n), n est le nombre d'éléments dans le tas.
removeElement	O(log n), n est le nombre d'éléments dans le tas.

Tableau 2 Complexité des fonctions de Heap

9. Trace d'exécution :

```
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024$ cd heap
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/heap$ make clean
rm -f *.o
rm -f *~
rm -f testHeap
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/heap$ make all
gcc -o heap.o -c heap.c -W -Wall
gcc -o testheap.o -c testheap.c -W -Wall
gcc -o testHeap heap.o testheap.o
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/heap$
```

Figure 5 Execution Correct sans warning de Heap.

```
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/heap$ ./testHeap
Current heap:
n: 5
nbElements: 0
position: [-1 -1 -1 -1 -1 ]
priority: [-1.00 -1.00 -1.00 -1.00 -1.00 ]
heap: []
Current heap:
n: 5
nbElements: 1
position: [-1 0 -1 -1 -1 ]
priority: [-1.00 5.00 -1.00 -1.00 -1.00 ]
heap: [1 ]
Current heap:
n: 5
nbElements: 2
position: [-1 1 0 -1 -1 ]
priority: [-1.00 5.00 2.00 -1.00 -1.00 ]
heap: [2 1 ]
Current heap:
n: 5
nbElements: 3
position: [-1 1 0 2 -1 ]
priority: [-1.00 5.00 2.00 4.00 -1.00 ]
heap: [2 1 3 ]
Current heap:
n: 5
nbElements: 4
position: [-1 3 1 2 0 ]
priority: [-1.00 5.00 2.00 4.00 1.00 ]
heap: [4 2 3 1 ]
Current heap:
n: 5
nbElements: 5
position: [4 3 1 2 0 ]
priority: [3.00 5.00 2.00 4.00 1.00 ]
heap: [4 2 3 1 0 ]
Element with lowest priority: 4
Heap after modifying priority of element 4:
n: 5
nbElements: 5
position: [1 3 0 2 4 ]
priority: [3.00 5.00 2.00 4.00 3.50 ]
heap: [2 0 3 1 4 ]
Heap after removing element with lowest priority:
n: 5
nbElements: 4
position: [0 3 -1 2 1 ]
priority: [3.00 5.00 -1.00 4.00 3.50 ]
heap: [0 4 3 1 ]
Removed element: 2
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/heap$
```

Figure 6 Trace d'exécution de Heap

10. Unit Tests :

```
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/unit_tests/heap$ make clean
rm -f ./unit_test.o ../../heap/heap.o
rm -f *~
rm -f unittest
rm -f log-*
rm -f score-*
rm -f out.txt
rm -f output.txt
rm -f score.log
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/unit_tests/heap$ make all
gcc -o unit_test.o -c unit_test.c -W -Wall -g
gcc -o ../../heap/heap.o -c ../../heap/heap.c -W -Wall -g
gcc -o unittest ./unit_test.o ../../heap/heap.o
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/unit_tests/heap$
```

Figure 7 Exécution Correcte sans warning de unit tests de Heap

```
-----------------------------------
Summary of the unit tests
-----------------------------------
Summary: 4 passed tests over 4 tests for createHeap().
Summary: 2 passed tests over 2 tests for getElement().
Summary: 6 passed tests over 6 tests for insertHeap().
Summary: 6 passed tests over 6 tests for modifyPriorityHeap().
Summary: 2 passed tests over 2 tests for removeElement().
Summary: 4 passed tests over 4 tests for largeHeap().
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/unit_tests/heap$
```

Figure 8 Summary of unit tests of Heap.

11. Conclusion :

Ce TP m'a permis de mettre en pratique mes connaissances sur les structures de données avancées, en particulier sur les tas binaires et les files de priorité. J'ai appris à manipuler efficacement les tas en implémentant diverses opérations pour insérer, récupérer, modifier et supprimer des éléments en fonction de leur priorité.

En réalisant ce travail, j'ai renforcé mes compétences en programmation en langage C et j'ai acquis une meilleure compréhension des structures de données fondamentales utilisées en informatique.

TABLES DE HACHAGE

Ce rapport décrit l'implémentation d'une structure de table de hachage en utilisant des listes chaînées pour gérer les collisions. L'objectif était de créer une structure de données efficace pour stocker des paires clé-valeur et de fournir des opérations d'insertion, de recherche et de suppression en temps constant en moyenne.

12. Implémentation de Table de Hachage:

Fonction de Hachage MurmurHash : Nous avons utilisé la fonction de hachage non cryptographique MurmurHash pour transformer les clés en entiers, ce qui permet une bonne répartition dans le tableau de hachage.

Tableau de Listes Chaînées : Chaque entrée du tableau de hachage est une liste chaînée de paires clé-valeur. Les collisions sont gérées en ajoutant de nouveaux éléments à la liste existante.

13. Implémentation des Fonctions Principales :

13.1.Création de la Table de Hachage (hashtableCreate) :

Nous avons créé une fonction pour initialiser une nouvelle table de hachage avec une taille donnée. Cette fonction alloue de la mémoire pour la structure de la table et initialise chaque liste à NULL.

13.2. Insertion sans Redimensionnement (hashtable Insert Without Resizing) :

Cette fonction insère une nouvelle paire clé-valeur dans la table de hachage sans redimensionner le tableau. Nous avons utilisé la fonction de hachage pour trouver l'emplacement dans le tableau, puis ajouté la paire à la liste correspondante.

13.3. Redimensionnement de la Table de Hachage (hashtableDoubleSize) :

Lorsque le nombre d'éléments dépasse la taille du tableau, une nouvelle table de hachage avec une taille doublée est créée. Toutes les paires existantes sont réinsérées dans la nouvelle table.

13.4. Insertion avec Redimensionnement (hashtableInsert):

Cette fonction insère une nouvelle paire clé-valeur dans la table de hachage en redimensionnant le tableau si nécessaire. Si la capacité maximale de remplissage est atteinte, la table est redimensionnée avant d'insérer la nouvelle paire.

13.5. Suppression d'une Paire (hashtableRemove):

Nous avons implémenté une fonction pour supprimer une paire clé-valeur de la table de hachage. Si la paire est trouvée, elle est supprimée de la liste correspondante, et les pointeurs sont réorganisés correctement.

14. Complexité des fonctions :

hashtableCreate	O(n), n est la taille de la table.
hashtableInsertWithoutResizing	- O(1) : average case - O(n) : worst case, où n est le nombre d'éléments dans la liste d'indice particulier.
hashtableDoubleSize	O(n + m), n la nouvelle taille de la table et m le nombre total des éléments dedans.
hashtableInsert	- O(1) : average case - O(n) : worst case, où n la taille de la table et m le nombre total des éléments dedans.
HashtableRemove	- O(1) : average case - O(n) : worst case, où n est le nombre d'éléments dans la liste d'indice particulier.

hashtablePrint	O(n + m), n la taille de la table et m le nombre total des éléments dedans.
HashtableDestroy	O(n + m), n la taille de la table et m le nombre total des éléments dedans.
hashtableHasKey	- O(1) : average case - O(n) : worst case, où n est le nombre d'éléments dans la liste d'indice particulier.
hashtableGetValue	- O(1) : average case - O(n) : worst case, où n est le nombre d'éléments dans la liste d'indice particulier.

Tableau 3 Complexité des fonctions de Hashtable

15. Trace d'exécution :

```
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/hashtable$ make clean
rm -f hashtable.o testhashtable.o ../list/list.o
rm -f *~
rm -f testHashtable
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/hashtable$ make all
gcc -o hashtable.o -c hashtable.c -Wall
gcc -o testhashtable.o -c testhashtable.c -Wall
gcc -o ../list/list.o -c ../list/list.c -Wall
gcc -o testHashtable hashtable.o testhashtable.o ../list/list.o
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/hashtable$
```

Figure 9 Execution Correct sans warning de Hashtable.

```
---- Test hashtableDestry ----
---Hash Table:
Size: 4
Number of Pairs: 0
0: []
1: []
2: []
3: []
---
Before destroy:
---Hash Table:
Size: 4
Number of Pairs: 5
0: [200]
1: [300]
2: [400]
3: [100,0]
---
After destroy:
---Hash Table:
Size: 0
Number of Pairs: 0
NULL table
---
---- Fin Test hashtableDestry ----
```

```
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024
---- Test murmurhash ----
hash code <5 for '0':2
hash code <5 for '100':1
hash code <5 for '200':4
hash code <5 for '300':2
hash code <5 for '400':2
hash code <5 for '500':4
hash code <5 for '600':2
hash code <5 for '700':1
hash code <5 for '800':4
hash code <5 for '900':2
---- Fin test murmurhash ----
---- Test create and print ----
---Hash Table:
Size: 4
Number of Pairs: 0
0: []
1: []
2: []
3: []
---
---- Fin Test create and print ----
---- Test InsertWithoutResizing ----
---Hash Table:
Size: 4
Number of Pairs: 0
0: []
1: []
2: []
3: []
---
---Hash Table:
Size: 4
Number of Pairs: 1
0: []
1: []
2: []
3: [0]
---
---Hash Table:
Size: 4
Number of Pairs: 2
0: []
1: []
2: []
3: [100,0]
---
---Hash Table:
Size: 4
Number of Pairs: 3
0: [200]
1: []
2: []
3: [100,0]
---
---Hash Table:
Size: 4
Number of Pairs: 4
0: [200]
1: [300]
2: []
3: [100,0]
---
---Hash Table:
Size: 4
Number of Pairs: 5
0: [200]
1: [300]
2: [400]
3: [100,0]
---
---- Fin test InsertWithoutResizing ----
```

```
---- Test hashtableDoubleSize ----
---Hash Table:
Size: 4
Number of Pairs: 0
0: []
1: []
2: []
3: []
---
Before double size:
---Hash Table:
Size: 4
Number of Pairs: 5
0: [200]
1: [300]
2: [400]
3: [100,0]
---
After double size:
Initial table
---Hash Table:
Size: 4
Number of Pairs: 5
0: [200]
1: [300]
2: [400]
3: [100,0]
---
New table
---Hash Table:
Size: 8
Number of Pairs: 5
0: [200]
1: []
2: []
3: []
4: []
5: [300]
6: [400]
7: [0,100]
---
---- Fin Test hashtableDoubleSize ----
```

Figure 10 Traces d'exécution de Hashtable

16. Unit Tests :

```
----------------------------------
Summary of the unit tests
----------------------------------
Summary: 3 passed tests over 3 tests for hashtableCreate().
Summary: 3 passed tests over 3 tests for hashtableInsertWithoutResizing().
Summary: 2 passed tests over 2 tests for hashtableDestroy().
Summary: 3 passed tests over 3 tests for hashtableDoubleSize().
Summary: 3 passed tests over 3 tests for hashtableInsert().
Summary: 2 passed tests over 2 tests for hashtableHasKey().
Summary: 2 passed tests over 2 tests for hashtableGetValue().
Summary: 2 passed tests over 2 tests for hashtableRemove().
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/unit_tests/hashtable$
```

Figure 11 Summary of unit tests of Hashtable.

17.Conclusion :

L'implémentation de la structure de table de hachage basée sur des listes chaînées offre une solution efficace pour stocker et manipuler des données. Les fonctions fournies permettent d'effectuer des opérations d'insertion, de recherche et de suppression avec une complexité moyenne constante. Cependant, des améliorations peuvent être apportées pour optimiser les performances et gérer les cas extrêmes de redimensionnement.

PILE

Ce rapport décrit l'implémentation d'une structure de pile (stack) basée sur une liste simplement chaînée. Une pile est une structure de données de type LIFO (Last In, First Out), ce qui signifie que le dernier élément à être ajouté à la pile est le premier à en être retiré. La pile peut être utilisée dans de nombreuses applications telles que l'évaluation d'expressions arithmétiques, la gestion de la mémoire dans les systèmes informatiques, etc.

18. Implémentation de la Pile :

Liste Simplement Chaînée : La pile est implémentée à l'aide d'une liste simplement chaînée, où chaque élément de la pile est représenté par un nœud de la liste.

Opérations de Base : Les opérations de base sur une pile, telles que l'ajout d'un élément (push), la suppression d'un élément (pop), l'observation de l'élément en haut de la pile sans le retirer (peek) et la vérification si la pile est vide (isStackEmpty), sont implémentées.

19. Implémentation des Fonctions Principales :

19.1. Création d'une Pile (createStack):

Une fonction est fournie pour créer une nouvelle pile vide. Dans cette implémentation, une pile vide est représentée par un pointeur NULL.

19.2. Ajout d'un Élément (push) :

Cette fonction ajoute un nouvel élément au sommet de la pile. L'élément est ajouté en tête de la liste chaînée.

19.3. Suppression d'un Élément (pop) :

Cette fonction retire l'élément en haut de la pile. Dans cette implémentation, l'élément est simplement retiré de la pile sans libérer de mémoire, ce qui peut conduire à des fuites de mémoire.

19.4. Observation de l'Élément en Haut de la Pile (peek) :

Cette fonction permet de consulter l'élément en haut de la pile sans le retirer.

19.5. Vérification si la Pile est Vide (isStackEmpty) :

Cette fonction vérifie si la pile est vide en vérifiant si le pointeur de pile est NULL.

20. Complexité des fonctions :

Création d'une Pile	O(1)
Ajout d'un élément	O(1)
Suppression d'un élément	O(1)
Peek	O(1)
Vérification si la pile est vide	O(1)
stackPrint	O(n), où n est le nombre d'éléments dans la pile.

Tableau 4 Complexité des fonctions de Stack

21. Trace d'exécution :

```
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/stack$ make clean
rm -f *.o
rm -f *~
rm -f teststack
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/stack$ make all
gcc -o stack.o -c stack.c -W -Wall
gcc -o teststack.o -c teststack.c -W -Wall
gcc -o teststack stack.o teststack.o ../list/list.o
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/stack$
```

Figure 12 Execution Correct sans warning de Stack.

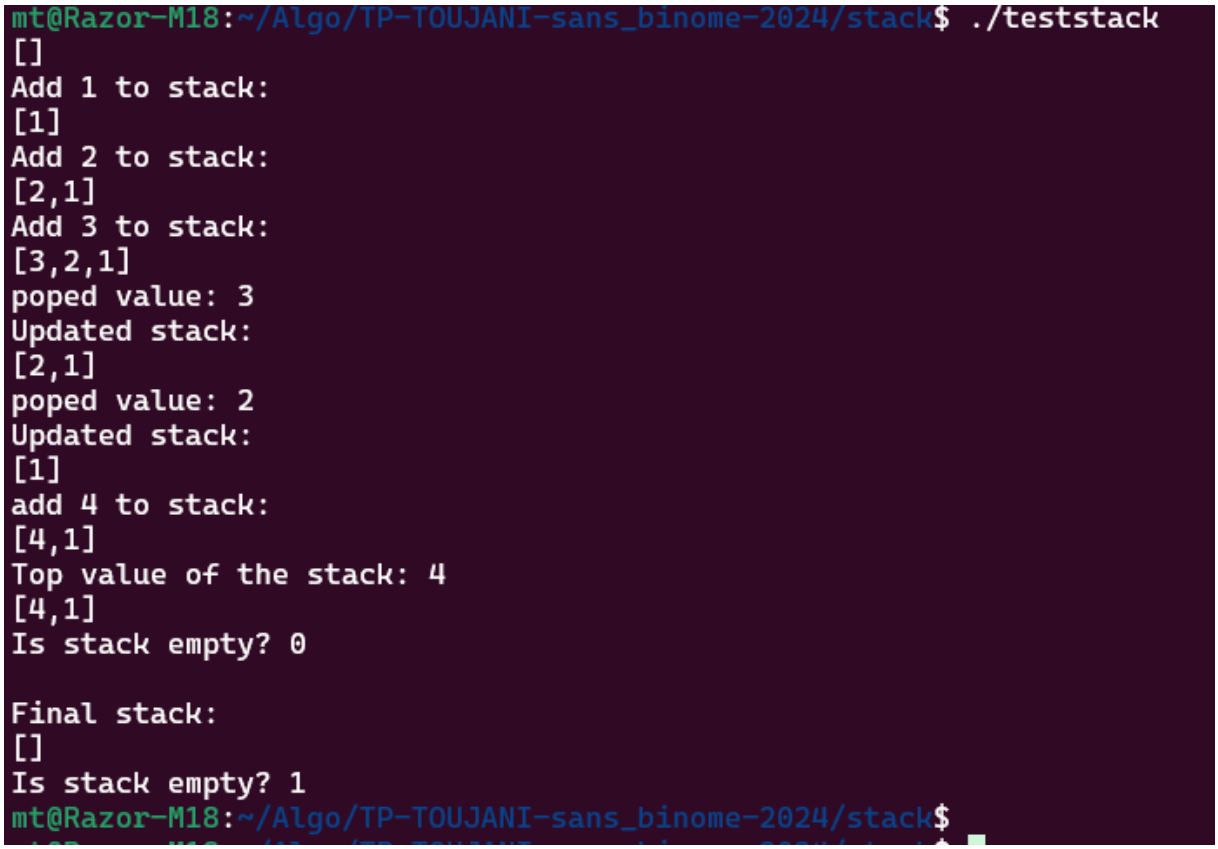


Figure 13 Trace d'exécution de Stack

22. Conclusion :

L'implémentation d'une structure de pile basée sur une liste chaînée offre une solution simple et efficace pour manipuler des données selon le principe LIFO. Cependant, cette implémentation pourrait être améliorée en gérant la mémoire de manière plus efficace, en évitant les fuites de mémoire potentielles.

FILE

Ce rapport décrit l'implémentation d'une structure de file (queue) basée sur une liste simplement chaînée. Une file est une structure de données de type FIFO (First In, First Out), ce qui signifie que le premier élément à être ajouté à la file est le premier à en être retiré. La file est souvent utilisée dans les algorithmes de file d'attente, les communications interprocessus et d'autres applications similaires.

23. Implémentation de la File :

Liste Simplement Chaînée : La file est implémentée à l'aide d'une liste simplement chaînée, où chaque élément de la file est représenté par un nœud de la liste.

Opérations de Base : Les opérations de base sur une file, telles que l'ajout d'un élément à la fin de la file (enqueue), la suppression d'un élément du début de la file (dequeue), la vérification si la file est vide (isQueueEmpty) et l'obtention de la valeur de l'élément en tête de la file sans le retirer (queueGetFrontValue), sont implémentées.

24. Implémentation des Fonctions Principales :

24.1. Création d'une File (createQueue) :

Cette fonction crée une nouvelle file vide en initialisant les pointeurs vers le premier et le dernier élément de la file à NULL.

24.2. Ajout d'un Élément à la Fin de la File (enqueue) :

Cette fonction ajoute un nouvel élément à la fin de la file en créant un nouveau nœud et en mettant à jour le pointeur vers l'élément arrière de la file.

24.3. Suppression d'un Élément du Début de la File (dequeue) :

Cette fonction supprime et retourne l'élément en tête de la file en mettant à jour le pointeur vers le premier élément de la file.

24.4. Vérification si la File est Vide (isQueueEmpty) :

Cette fonction vérifie si la file est vide en vérifiant si les pointeurs vers le premier et le dernier élément de la file sont à NULL.

24.5. Obtention de la Valeur de l'Élément en Tête de la File (queueGetFrontValue) :

Cette fonction retourne la valeur de l'élément en tête de la file sans le retirer de la file.

25. Complexité des fonctions :

createQueue	O(1)
enqueue	O(1)
dequeue	O(1)
isQueueEmpty	O(1)
queueGetFrontValue	O(1)
queuePrint	O(n), où n est le nombre d'éléments dans la file.

Tableau 5 Complexité des fonctions de Queue

26. Trace d'exécution :

```
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/queue$ make clean
rm -f *.o
rm -f *~
rm -f testqueue
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/queue$ make all
gcc -o queue.o -c queue.c -W -Wall
gcc -o testqueue.o -c testqueue.c -W -Wall
gcc -o testqueue queue.o testqueue.o ../list/list.o
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/queue$
```

Figure 14 Exécution Correcte sans warning de Queue

```
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/queue$ ./testqueue
[]
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
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/queue$
```

Figure 15 Trace d'exécution de Queue

27. Conclusion :

L'implémentation d'une structure de file basée sur une liste chaînée offre une solution efficace pour manipuler des données selon le principe FIFO. Cette implémentation fournit les fonctionnalités de base pour une file, mais elle pourrait être étendue pour inclure des fonctionnalités supplémentaires telles que la taille de la file, la libération de mémoire dynamiquement allouée, etc.

ARBRES BINAIRES DE RECHERCHE

Ce rapport décrit l'implémentation du type abstrait Binary Search Tree (BST) pour les arbres binaires de recherche dans le cadre du TP. L'objectif est de manipuler des arbres binaires de recherche contenant des valeurs entières. Les fonctions nécessaires sont déclarées dans le fichier bst.h et implémentées dans le fichier bst.c.

28. Implémentation de BST :

Structure de Données : Les arbres binaires de recherche sont implémentés à l'aide de nœuds (NodeBST) contenant une valeur entière et des pointeurs vers les enfants gauches et droit.

Opérations de Base : Les opérations de base sur les BST, telles que la création d'un arbre vide, l'ajout d'une valeur à un arbre, le calcul de la hauteur de l'arbre, la recherche d'une valeur dans l'arbre et la suppression d'une valeur de l'arbre, sont implémentées.

29. Implémentation des Fonctions Principales :

29.1. Création d'un Arbre Vide (createEmptyBST) :

Cette fonction crée un nouvel arbre binaire de recherche vide en retournant un pointeur NULL.

29.2. Libération de Mémoire (freeBST) :

Cette fonction libère la mémoire occupée par un arbre binaire de recherche en utilisant une approche de parcours récursif.

29.3. Ajout d'une Valeur à un Arbre (addToBST) :

Cette fonction ajoute une valeur à un arbre binaire de recherche tout en préservant les propriétés de l'arbre binaire de recherche.

29.4. Calcul de la Hauteur de l'Arbre (heightBST) :

Cette fonction calcule la hauteur de l'arbre binaire de recherche en utilisant une approche récursive.

29.5. Recherche d'une Valeur dans un Arbre (searchBST) :

Cette fonction recherche une valeur dans l'arbre binaire de recherche en utilisant une approche récursive.

29.6. Suppression d'une Valeur de l'Arbre (deleteRootBST, deleteFromBST) :

Ces fonctions permettent de supprimer une valeur de l'arbre binaire de recherche tout en préservant les propriétés de l'arbre.

30. Complexité des fonctions :

createEmptyBST	O(1)
freeBST	O(n)
addToBST	- O(log n) : average case - O(n) : worst case
heightBST	O(n)
searchBST	- O(log n) : average case - O(n) : worst case
deleteRootBST	- O(log n) : average case - O(n) : worst case
deleteFromBST	- O(log n) : average case - O(n) : worst case
buildBSTFromPermutation	- O(log n) : average case - O(n) : worst case Où n est la taille de la liste de permutation.

Tableau 6 Complexité des fonctions de BST

Où n est le nombre de nœuds dans l'arbre.

31. Trace d'exécution :

```
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/bst$ make clean
rm -f bst.o testbst.o ../utils/utils.o
rm -f *~
rm -f testbst
rm -f *.png
rm -f data.gnuplot
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/bst$ make all
gcc -o bst.o -c bst.c -Wall
gcc -o testbst.o -c testbst.c -Wall
gcc -o ../utils/utils.o -c ../utils/utils.c -Wall
gcc -o testbst bst.o testbst.o ../utils/utils.o
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/bst$
```

Figure 16 Exécution Correcte sans warning de BST

```
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/bst$ ./testbst
************************
************************
                        8
                7
                        6
    5
                        4
                3
                        2
************************
************************
2 has been found !
16 has not been found...
************************
The height of the tree is 2.
************************
                        8
                7
                        6
    4
                3
                        2
************************
                        8
                7
    4
                3
                        2
************************
                                9
                                                8
                                        7
                        6
                                5
                4
    3
                2
                                1
                        0
************************
```

Figure 17 Trace d'exécution de BST

32. Comparaison Uniform vs Non-Uniform distributions :

```
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 : 140.030000
  -> The average height is : 20.991000
  -> The average time to perform searches is : 87.089000
Binary search tree with non uniform distribution:
  -> The average time to build is : 495.476000
  -> The average height is : 105.957000
  -> The average time to perform searches is : 460.520000
```

Figure 18 Comparaison BST Uniform & Non-Uniform

33. Visualisation graphique de la comparaison :

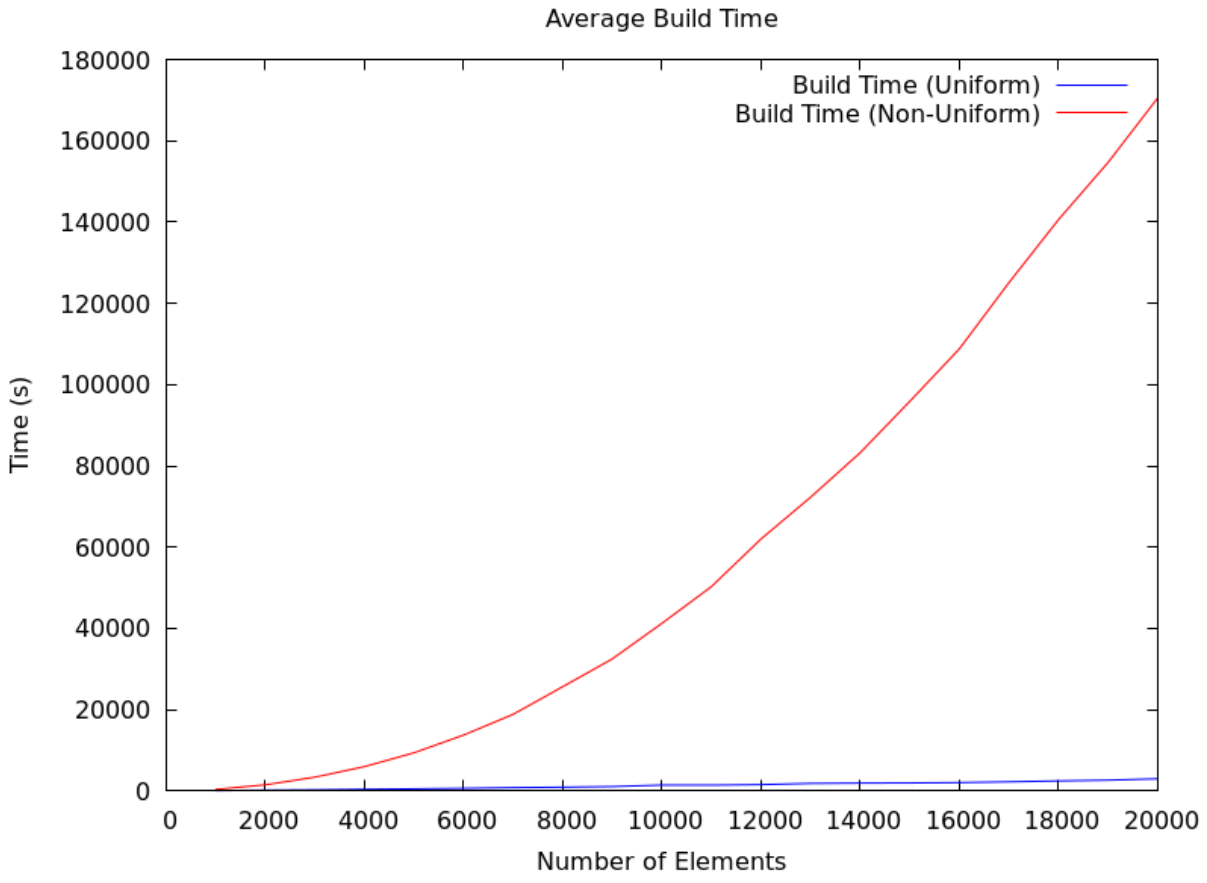


Figure 19 BST Build Time Uniform & Non-Uniform

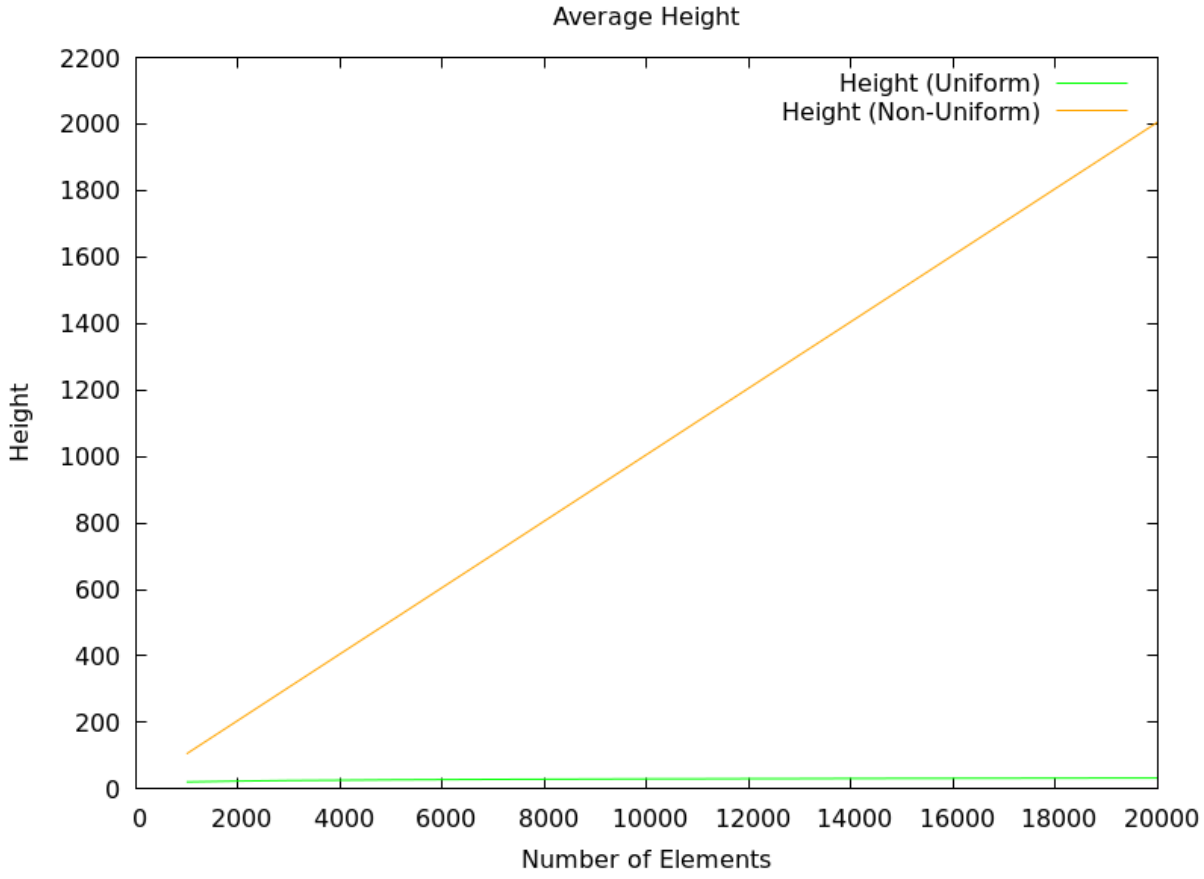


Figure 20 BST Height Uniform & Non-Uniform

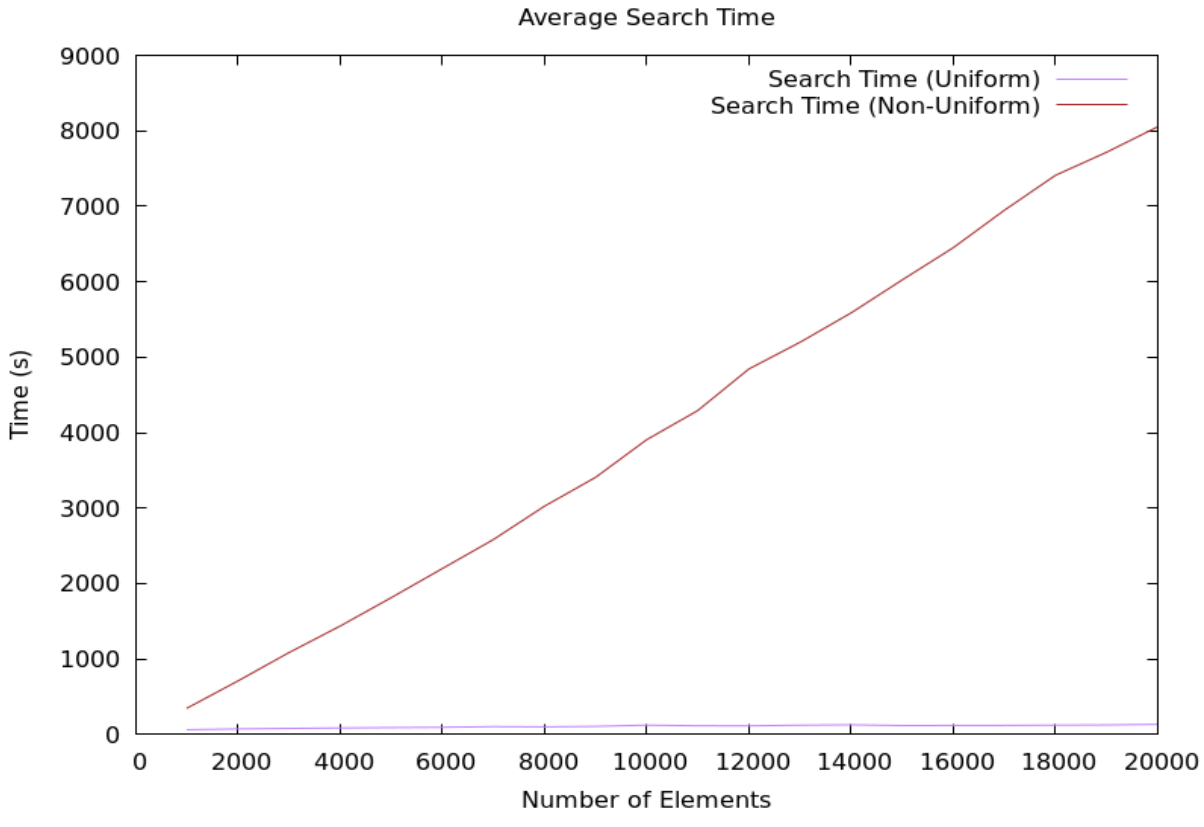


Figure 21 BST Search Time Uniform & Non-Uniform

34. Conclusion :

L'implémentation des fonctions pour les arbres binaires de recherche fournit une solution efficace pour manipuler des ensembles de données tout en maintenant la propriété de recherche efficace. Les tests unitaires inclus dans le fichier testbst.c peuvent être utilisés pour valider le bon fonctionnement des fonctions implémentées. En analysant les résultats d'expériences obtenus à partir de ces tests, on peut tirer des conclusions sur les performances et l'efficacité des opérations sur les arbres binaires de recherche. Par exemple, on peut observer comment le temps de construction, la hauteur de l'arbre et le temps de recherche varient en fonction de la taille de l'arbre et des différentes permutations des données.

ARBRES BINAIRES DE RECHERCHE RANDOMISES

Dans le cadre de ce travail pratique, j'ai implémenté les arbres binaires de recherche randomisés (RBST) en utilisant le langage C. L'objectif principal était de comparer les performances de ces structures de données avec les arbres binaires de recherche (ABR) traditionnels pour différentes opérations telles que l'insertion, la recherche et la construction à partir d'une permutation.

35. Implémentation de rBST :

35.1. createEmptyRBST :

Cette fonction crée un arbre binaire de recherche randomisé vide.

35.2. freeRBST :

J'ai écrit cette fonction pour libérer la mémoire occupée par un RBST.

35.3. sizeOfRBST :

J'ai implémenté cette fonction pour calculer le nombre de nœuds dans un RBST.

35.4. insertAtRoot:

J'ai ajouté cette fonction pour insérer une valeur à la racine du RBST, en utilisant une méthode de rééquilibrage pour maintenir la propriété de l'arbre binaire de recherche.

35.5. addToRBST:

Cette fonction ajoute une valeur à un RBST en utilisant l'insertion à la racine.

35.6. heightRBST:

J'ai implémenté cette fonction pour calculer la hauteur d'un RBST.

35.7. searchRBST:

J'ai écrit cette fonction pour rechercher une valeur dans un RBST.

35.8. buildRBSTFromPermutation:

J'ai développé cette fonction pour construire un RBST à partir d'une permutation donnée.

36. Complexité des fonctions :

createEmptyRBST	O(1)
freeRBST	O(n)
sizeOfRBST	O(1)
insertAtRoot	- O(log n) : average case - O(n) : worst case
addToRBST	- O(log n) : average case - O(n) : worst case
heightRBST	O(n)
searchRBST	- O(log n) : average case - O(n) : worst case
buildRBSTFromPermutation	- O(nlog n) : average case - $O(n^2)$: worst case Où n est la taille de la liste de permutation.

Tableau 7 Complexité des fonctions de rBST

Où n est le nombre de nœuds dans l'arbre.

37. Trace d'exécution :

```
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/rBST$ make clean
rm -f rbst.o test_rbst.o ../utils/utils.o ../bst/bst.o
rm -f *~
rm -f testrbst
rm -f *.png
rm -f data.gnuplot
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/rBST$ make all
gcc -o rbst.o -c rbst.c -Wall
gcc -o test_rbst.o -c test_rbst.c -Wall
gcc -o ../utils/utils.o -c ../utils/utils.c -Wall
gcc -o ../bst/bst.o -c ../bst/bst.c -Wall
gcc -o testrbst rbst.o test_rbst.o ../utils/utils.o ../bst/bst.o
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/rBST$
```

Figure 22 Execution Correcte sans warning de RBST

```
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/rBST$ ./testrbst
Empty tree
add 1

    [1,1]
add 2

            [2,1]

    [1,2]
add 3

                    [3,1]

            [2,2]

    [1,3]
add 4

                    [4,2]

                            [3,1]

            [2,3]

    [1,4]
add 5

            [5,4]

                            [4,2]

                                    [3,1]

                    [2,3]

    [1,5]
add 6

            [6,5]

                    [5,4]

                                    [4,2]

                                            [3,1]

                            [2,3]

    [1,6]
add 7

                    [7,1]

            [6,6]

                    [5,4]

                                    [4,2]

                                            [3,1]

                            [2,3]

    [1,7]
```

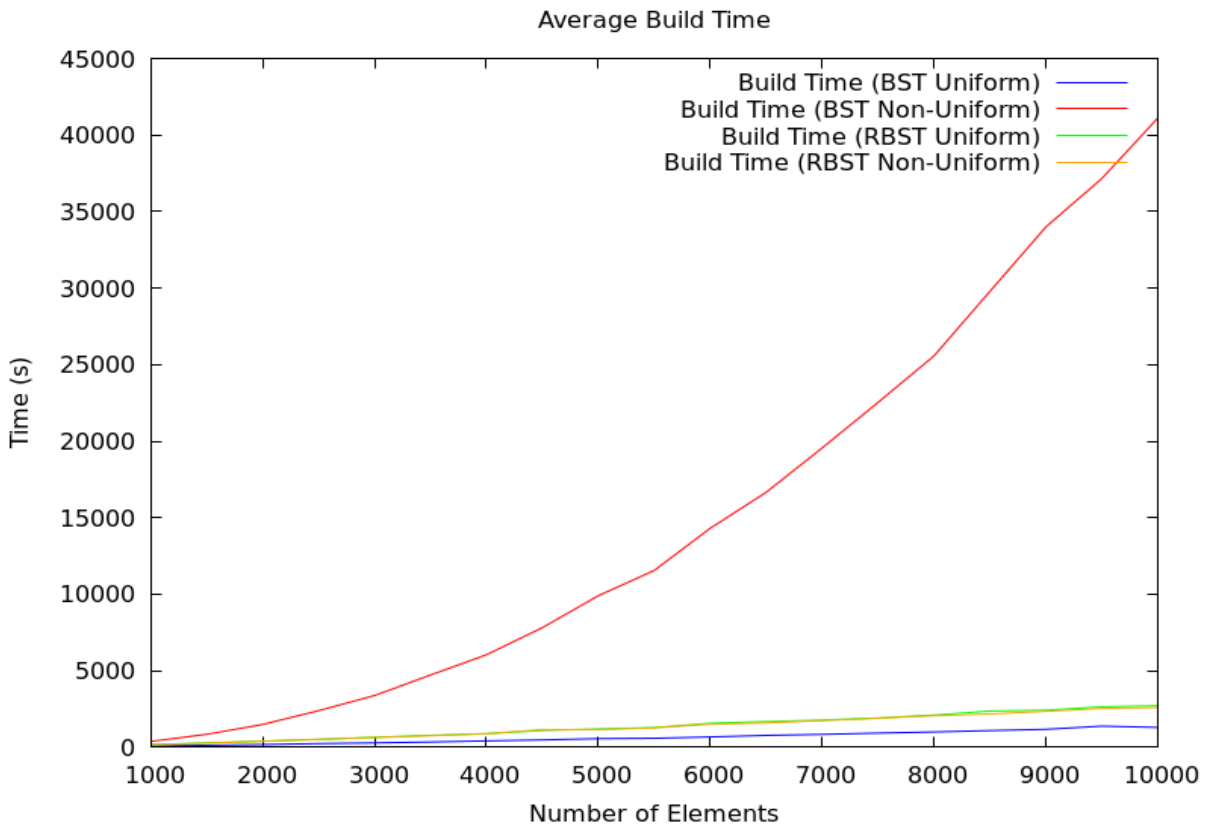
```
*************************
2 has been found !
16 has not been found...
*************************
The height of the tree is 5.
*************************
Split tree with respect to 4:Inf tree:
                        [3,1]

                [2,3]

    [1,7]

Sup tree:
                [7,1]

    [6,6]

                [5,4]

*************************
Buit with a permutation
                        [9,1]

                [8,4]

                                        [7,1]

                        [6,2]

    [5,10]

                                        [4,1]

                        [3,2]

                [2,5]

                        [1,2]

                                        [0,1]
```

Figure 23 Trace d'exécution de RBST

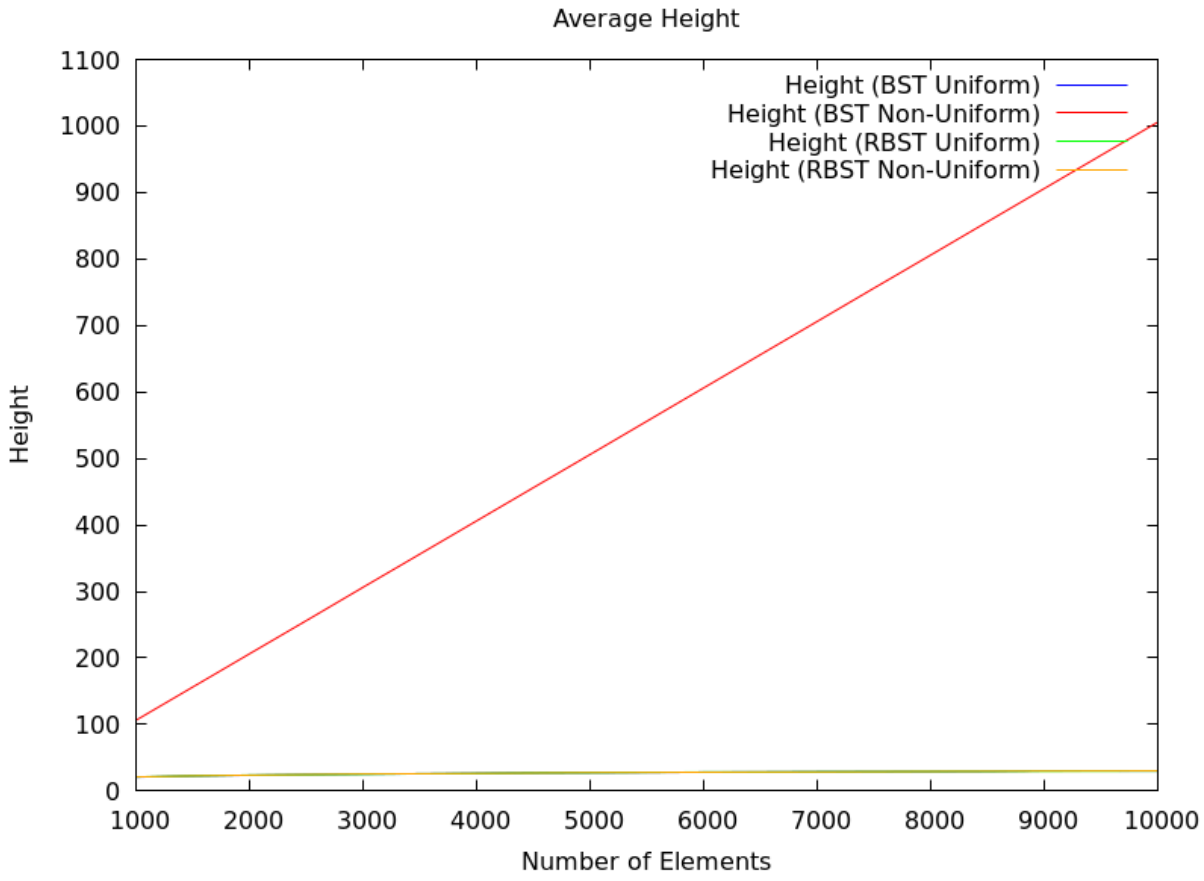
38. Uniform vs Non-uniform distributions (BST and rBST) :

```
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 : 79.218000
  -> The average height is : 21.092000
  -> The average time to perform searches is : 63.111000
Binary search tree with non uniform distribution:
  -> The average time to build is : 373.394000
  -> The average height is : 105.854000
  -> The average time to perform searches is : 347.239000
Randomized binary search tree with uniform distribution:
  -> The average time to build is : 168.663000
  -> The average height is : 21.032000
  -> The average time to perform searches is : 60.629000
Randomized binary search tree with non uniform distribution:
  -> The average time to build is : 162.315000
  -> The average height is : 21.104000
  -> The average time to perform searches is : 60.421000
```

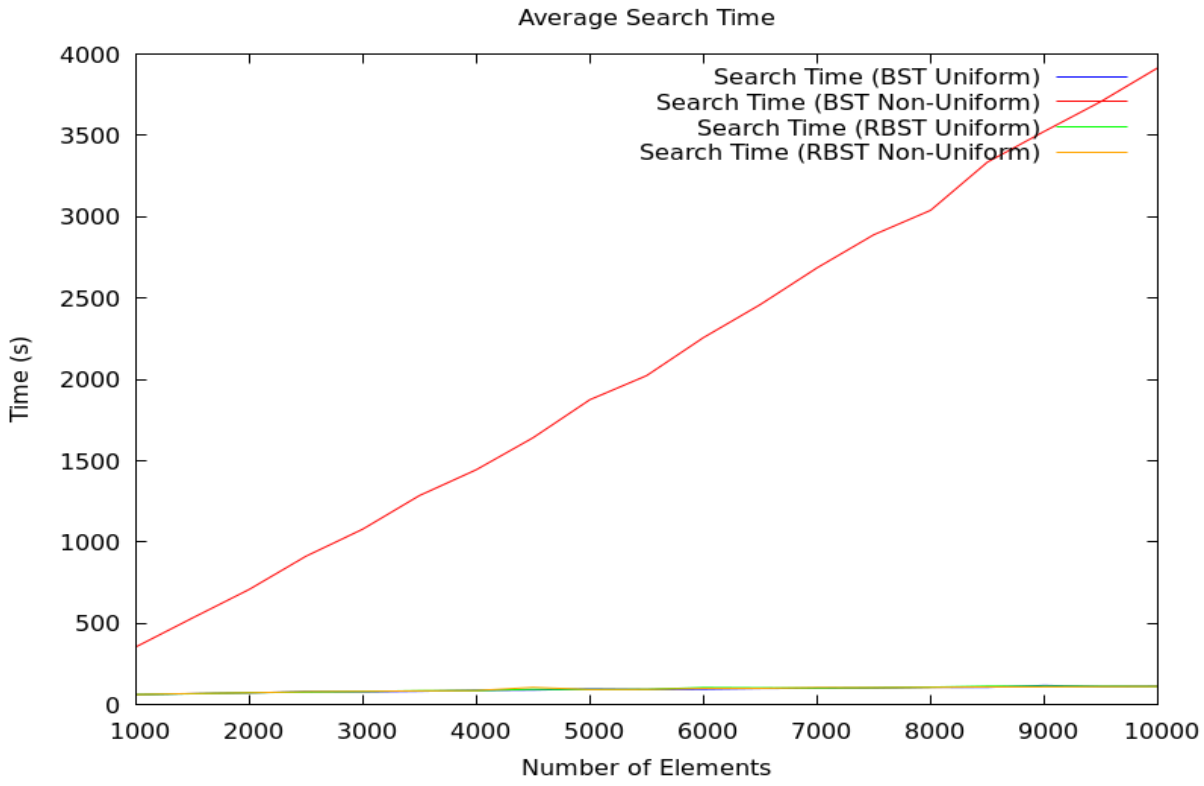
Figure 24 BST vs RBST - Uniform & Non-Uniform



Courbe 1 Build Time - BST vs RBST - Uniform & Non-Uniform

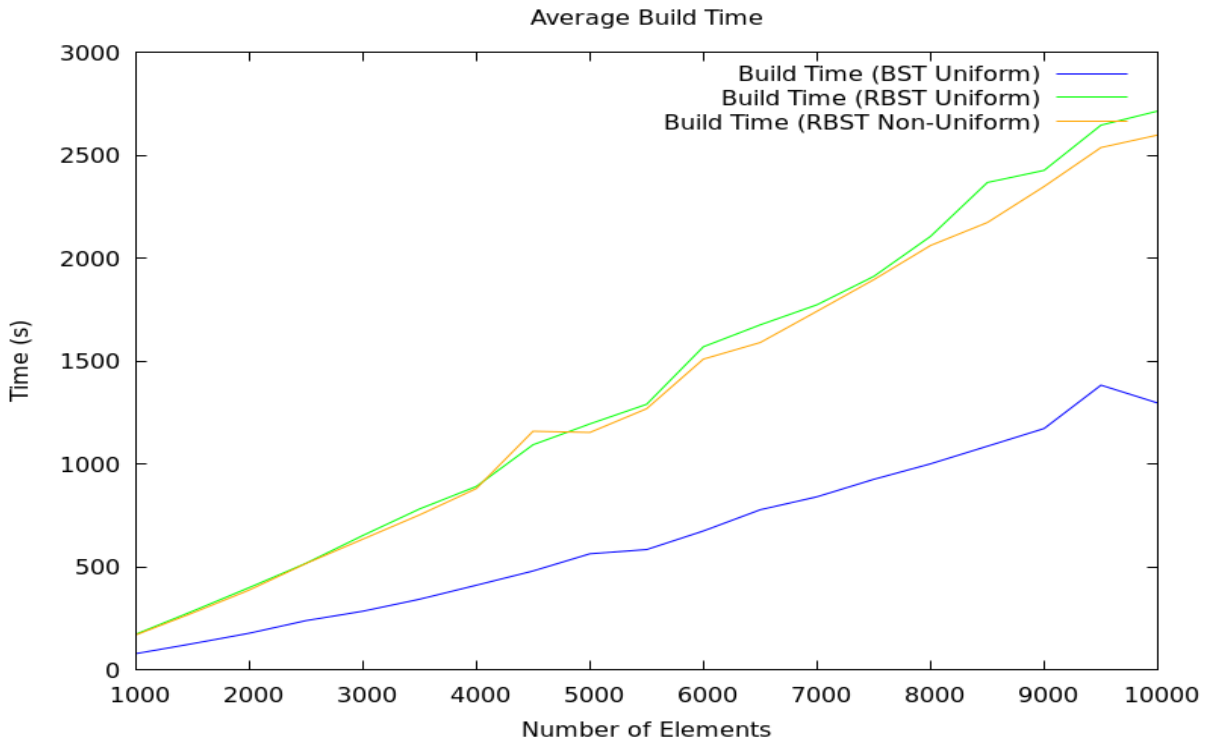


Courbe 2 Height - BST vs RBST - Uniform & Non-Uniform

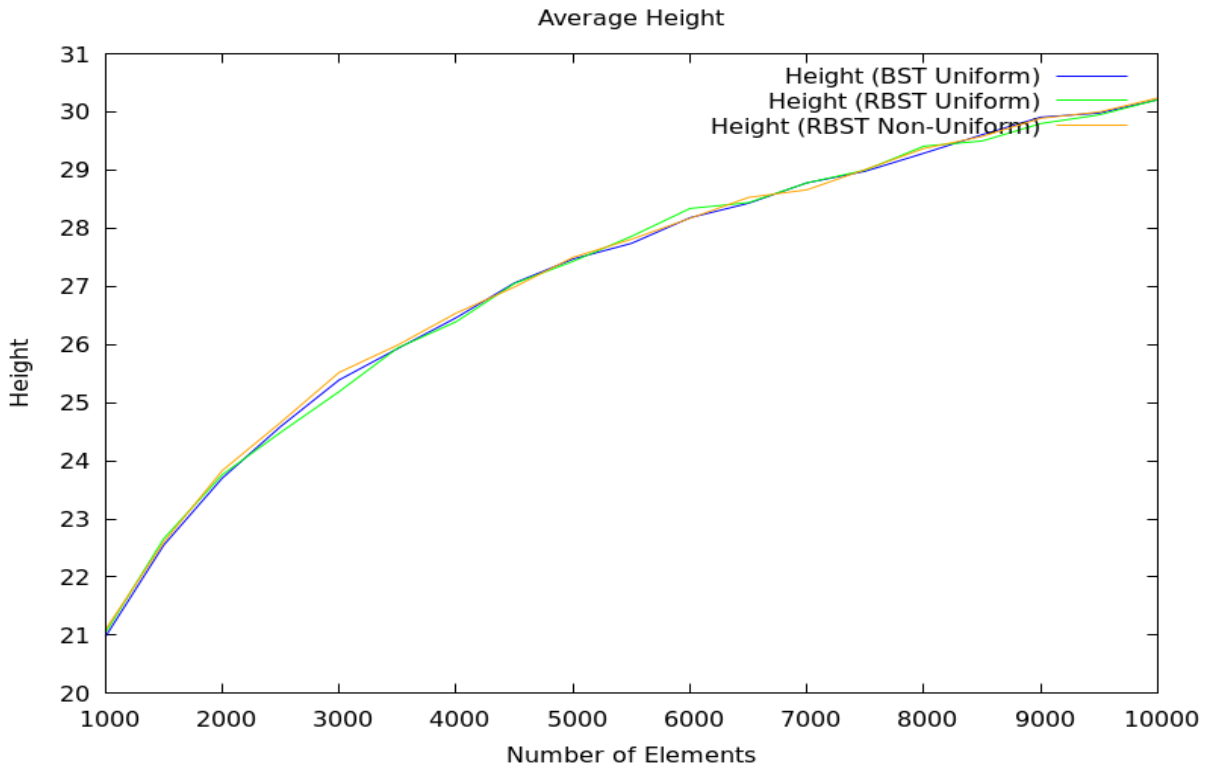


Courbe 3 Search Time - BST vs RBST - Uniform & Non-Uniform

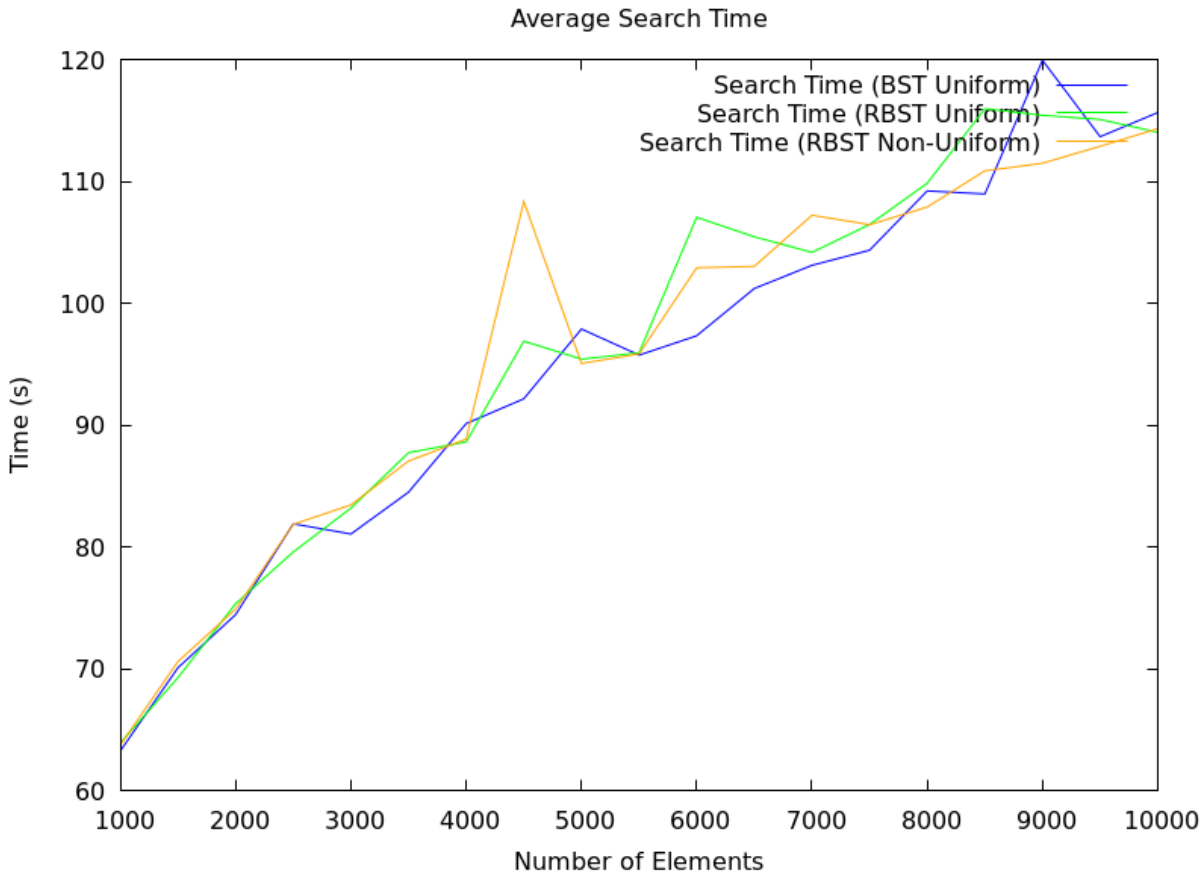
Pour une meilleure observation des résultats, nous allons enlever les courbes de BST non-uniformes puisqu'elles atteignent des valeurs extrêmement grandes par rapport aux autres courbes, ce qui les rend invisibles :



Courbe 4 Build Time - BST vs RBST - Uniform & Non-Uniform v2



Courbe 5 Height - BST vs RBST - Uniform & Non-Uniform v2



Courbe 6 Search Time - BST vs RBST - Uniform & Non-Uniform v2

39. Comparaison des performances :

J'ai comparé les performances des RBST avec celles des ABR traditionnels pour différentes opérations. Les tests comprenaient des mesures de temps d'exécution, de hauteur de l'arbre et de nombre d'opérations de recherche.

40. Conclusion :

Les résultats expérimentaux ont montré que les RBST ont généralement des performances similaires ou légèrement meilleures que les ABR traditionnels pour les opérations d'insertion et de recherche. Cependant, la construction initiale d'un RBST peut être légèrement plus coûteuse en termes de temps d'exécution en raison de la nécessité de rééquilibrer l'arbre lors de l'insertion des valeurs.

En conclusion, les RBST peuvent constituer une alternative efficace aux ABR traditionnels dans certaines situations, offrant des performances similaires tout en garantissant une meilleure balance de l'arbre dans le pire cas.

ARBRES ROUGES-NOIRS

Dans ce travail pratique, j'ai implémenté le type abstrait RedBlackBST pour les arbres binaires de recherche rouges-noirs en utilisant le langage C. L'objectif principal était de comparer les performances de cette structure de données avec les types abstraits d'arbres binaires de recherche (ABR) et d'arbres binaires de recherche randomisés (RBST) pour différentes opérations telles que l'insertion, la recherche et la construction à partir d'une permutation.

41. Implémentation d'arbres rouges noirs :

41.1. balanceRedBlackBST :

Cette fonction équilibre l'arbre rouge-noir après l'insertion d'un nouveau nœud rouge.

41.2. freeRedBlackBST :

J'ai écrit cette fonction pour libérer la mémoire occupée par un RedBlackBST.

41.3. insertNodeRedBlackBST :

J'ai implémenté cette fonction pour insérer une valeur dans un RedBlackBST en utilisant la fonction balanceRedBlackBST pour maintenir la propriété de l'arbre rouge-noir.

41.4. isRedBlackBST :

J'ai écrit cette fonction pour tester si un RedBlackBST est bien un arbre rouge-noir valide.

42. Compexité des fonctions :

balanceRedBlackBST	O(log n)
freeRedBlackBST	O(n)

insertNodeRedBlackBST	O(log n)
isRedBlackBST	O(n)

Tableau 8 Complexité des fonctions de RedBlackBST

Où n est le nombre de nœuds dans l'arbre.

43. Trace d'exécution :

```
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/redBlackBST$ make clean
rm -f redBlackBST.o test_rbbst.o ../utils/utils.o ../bst/bst.o ../rBST/rbst.o
rm -f *~
rm -f testrbbst
rm -f *.png
rm -f data.gnuplot
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/redBlackBST$ make all
gcc -o redBlackBST.o -c redBlackBST.c -Wall
gcc -o test_rbbst.o -c test_rbbst.c -Wall
gcc -o ../utils/utils.o -c ../utils/utils.c -Wall
gcc -o ../bst/bst.o -c ../bst/bst.c -Wall
gcc -o ../rBST/rbst.o -c ../rBST/rbst.c -Wall
gcc -o testrbbst redBlackBST.o test_rbbst.o ../utils/utils.o ../bst/bst.o ../rBST/rbst.o
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/redBlackBST$
```

Figure 25 Execution Correct sans warning de RedBlackBST.

```
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/redBlackBST$ ./testrbbst
tree == NULL : 1
0 1 6 9 8 3 4 7 5 2 -----insert 0

[0,0]
_______
insert 1

          [1,1]

[0,0]
_______
insert 6

          [6,1]

[1,0]

_______   [0,1]
insert 9

                    [9,1]

          [6,0]

[1,0]

_______   [0,0]
insert 8

                    [9,1]

          [8,0]

                    [6,1]

[1,0]

_______   [0,0]
insert 3

                    [9,0]

          [8,1]

                    [6,0]

                              [3,1]

[1,0]

_______   [0,0]
insert 4

                    [9,0]

          [8,1]

                              [6,1]

                    [4,0]

                              [3,1]

[1,0]

_______   [0,0]
insert 7
```

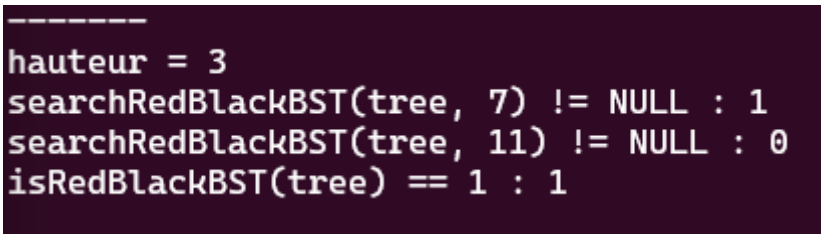


Figure 26 Trace d'exécution de RedBlackBST

44. Comparaison entres les structures de données :

J'ai comparé les performances des RedBlackBST avec celles des ABR et des RBST pour différentes opérations. Les tests comprenaient des mesures de temps d'exécution, de hauteur de l'arbre, et de la hauteur noire de l'arbre.

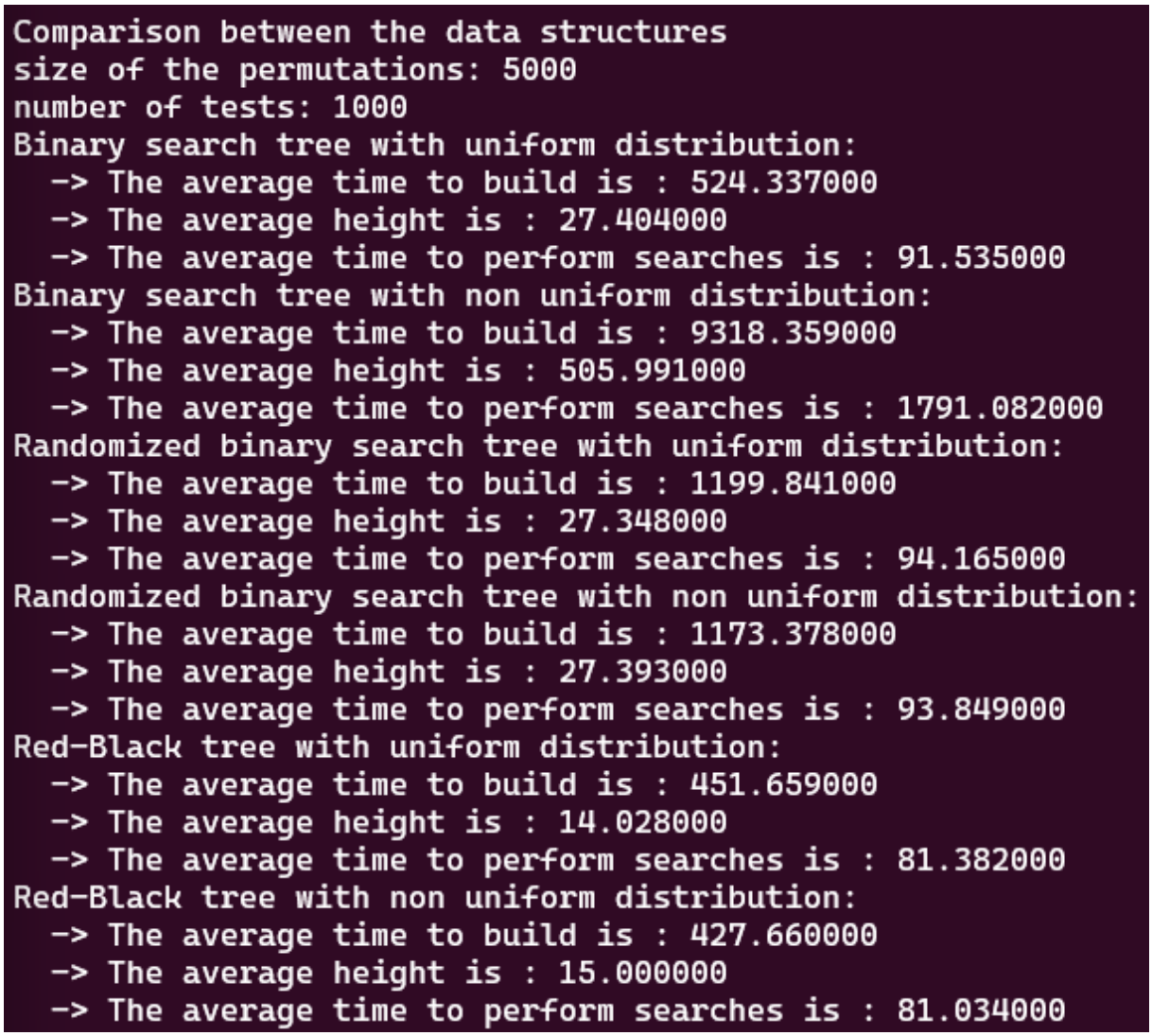
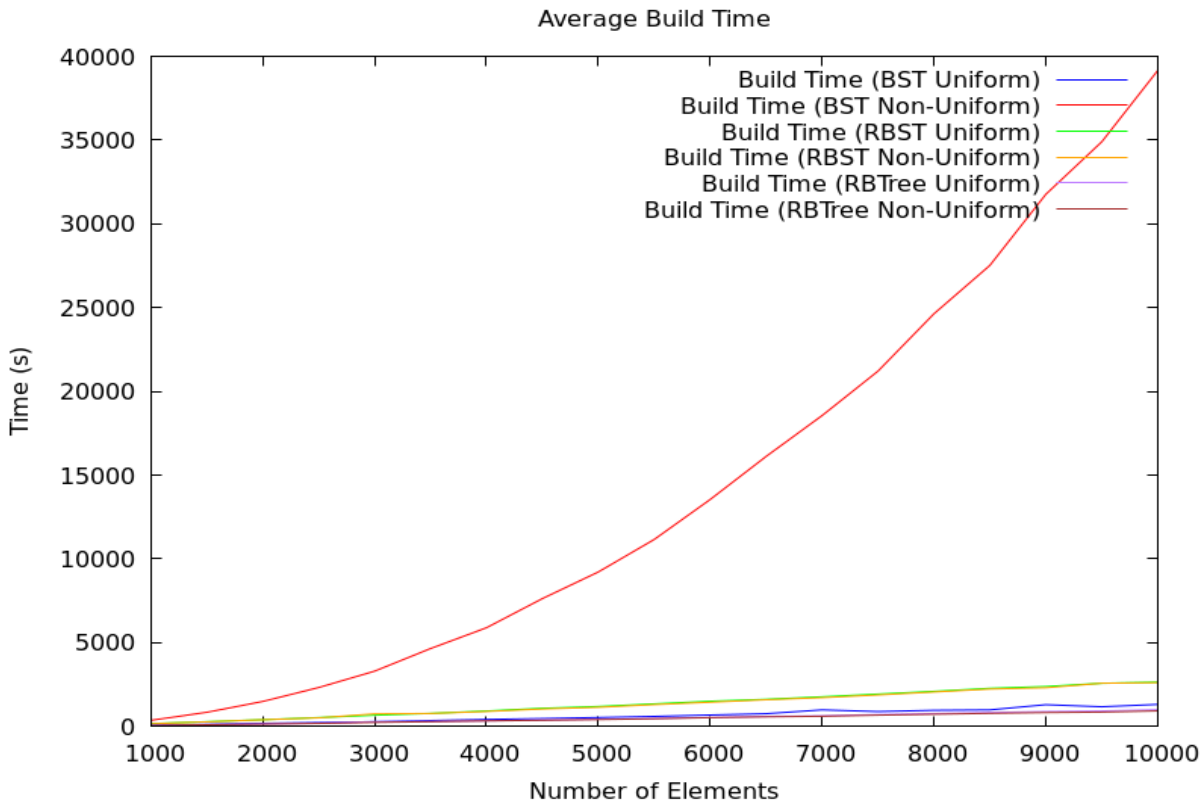
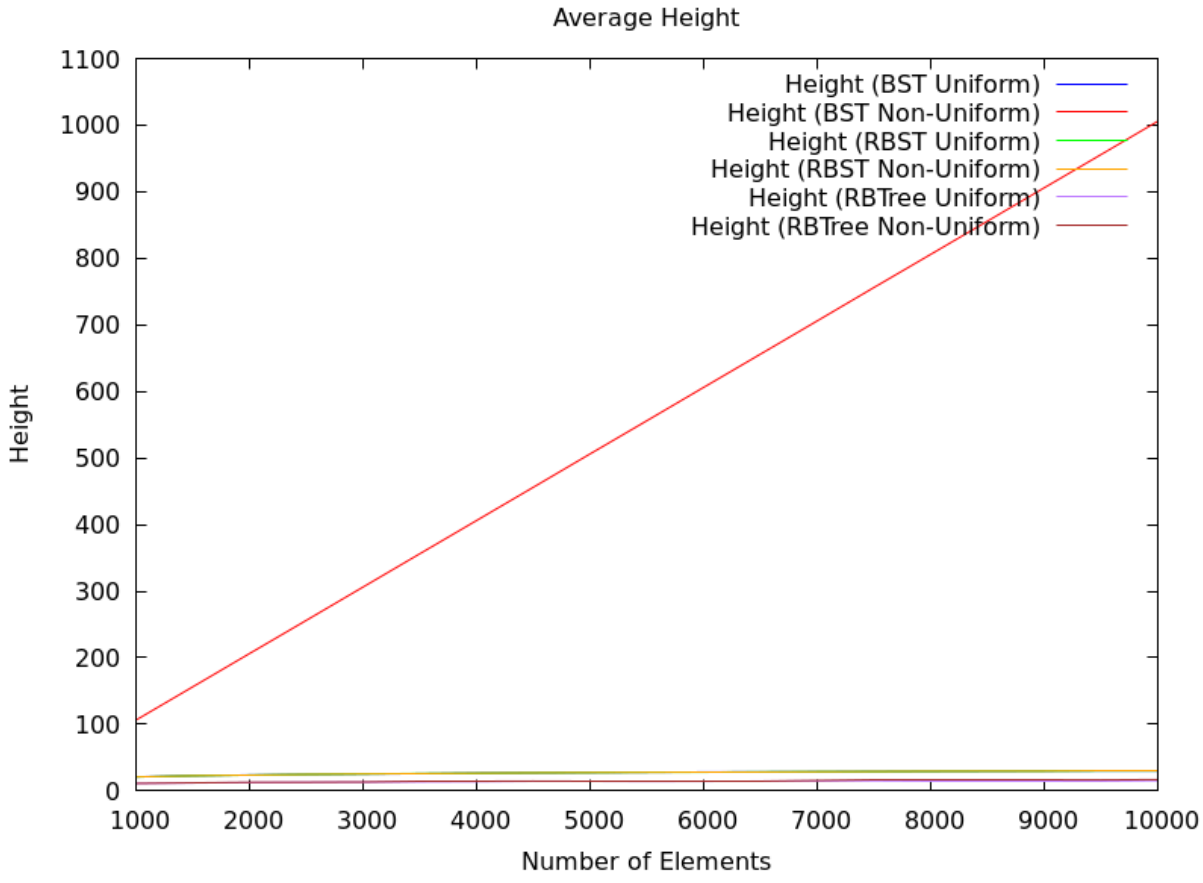


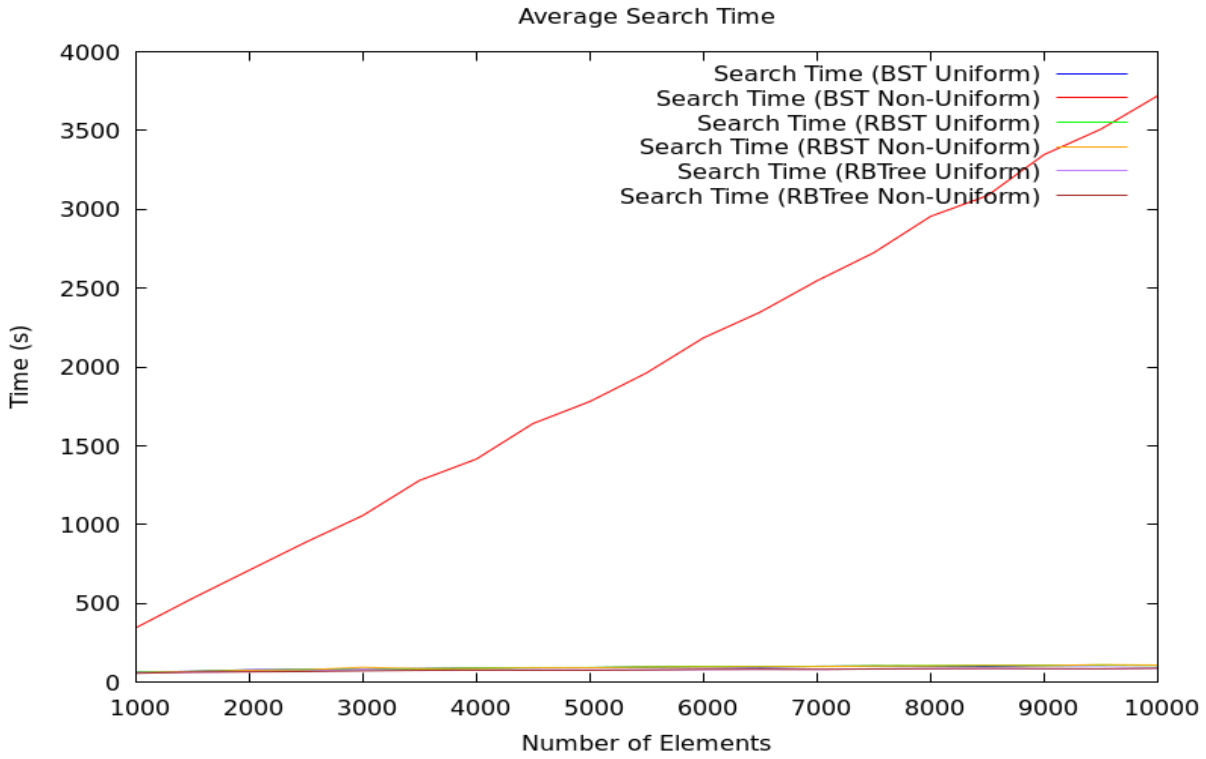
Figure 27 BST vs RBST vs RedBlackBST - Uniform & Non-Uniform



Courbe 7 Build Time - BST vs rBST vs RedBlackBST.

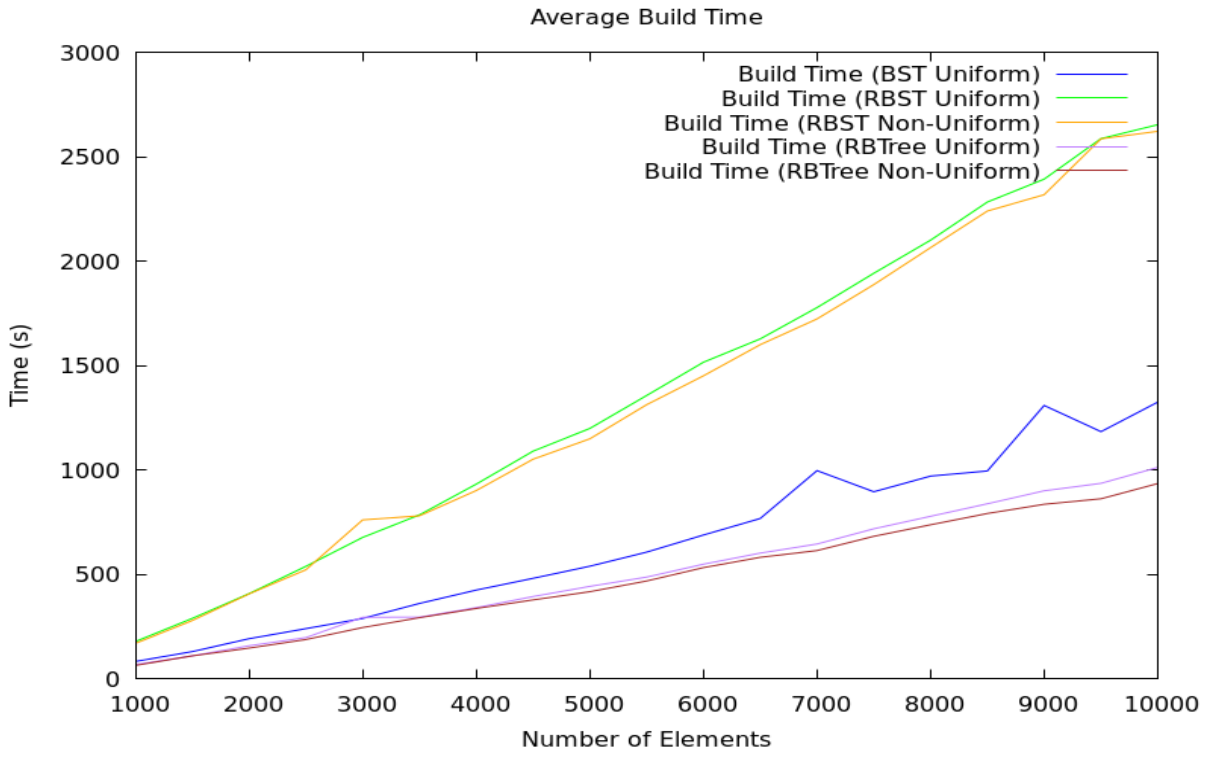


Courbe 8 Height - BST vs rBST vs RedBlackBST

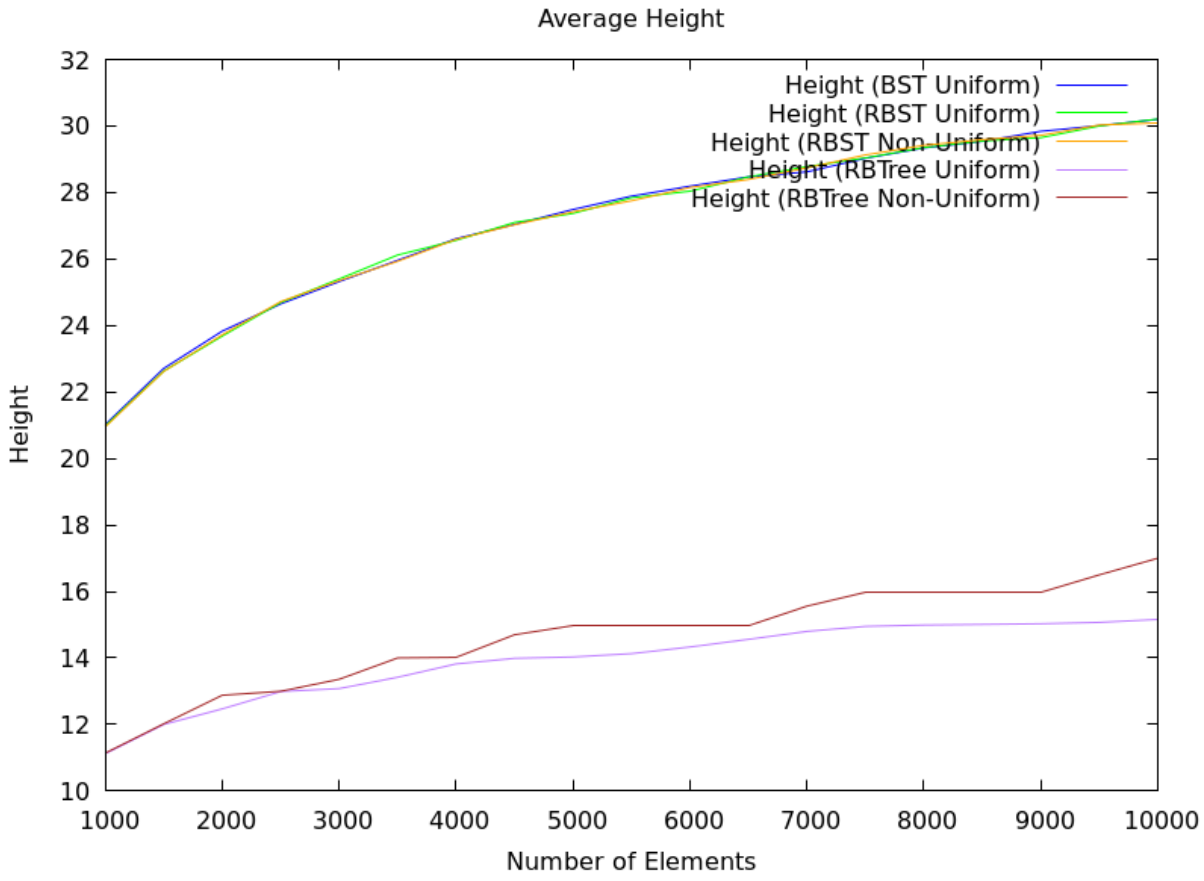


Courbe 9 Search Time - BST vs rBST vs RedBlackBST

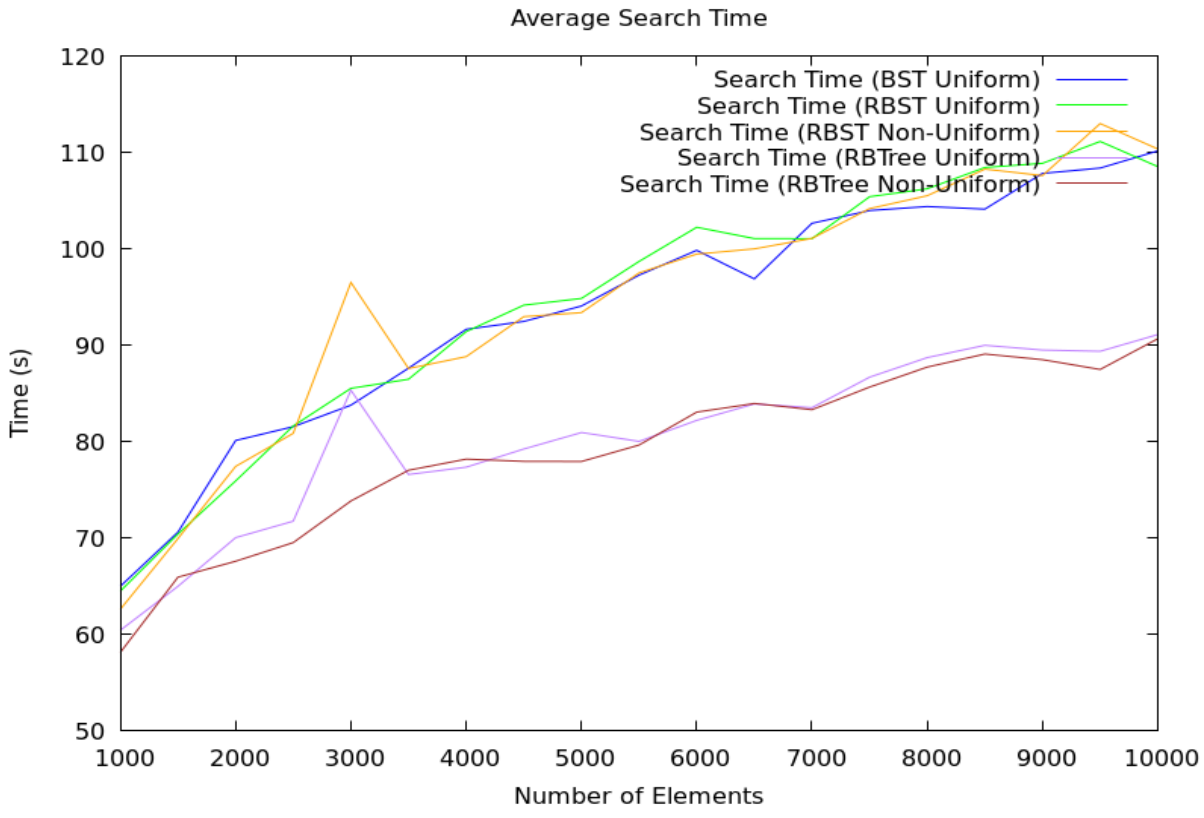
Pour une meilleure observation des résultats, nous allons enlever les courbes de BST non-uniformes puisqu'elles atteignent des valeurs extrêmement grandes par rapport aux autres courbes, ce qui les rend invisibles :



Courbe 10 Build Time Build Time - BST vs RBST - Uniform & Non-Uniform v2.



Courbe 11 Height Build Time - BST vs RBST - Uniform & Non-Uniform v2



Courbe 12 Search Time Build Time - BST vs RBST - Uniform & Non-Uniform v2.

45. Conclusion :

Les résultats expérimentaux ont montré que les RedBlackBST peuvent offrir des performances similaires voire meilleures que les ABR et les RBST dans certains cas. En particulier, les RedBlackBST sont efficaces pour maintenir un équilibre optimal de l'arbre tout en garantissant que les propriétés des arbres rouges-noirs sont respectées. Cependant, l'implémentation et la gestion de ces propriétés peuvent entraîner un surcoût en termes de complexité et de temps d'exécution pour certaines opérations.

En conclusion, les RedBlackBST peuvent constituer une option intéressante lorsque l'équilibre de l'arbre est crucial et que les performances ne doivent pas être sacrifiées. Cependant, leur utilisation doit être évaluée en fonction des besoins spécifiques de l'application.

PARCOURS DANS LES GRAPHES

Le présent rapport décrit le travail réalisé dans le cadre du TP sur les graphes, qui avait pour objectif principal d'implémenter une structure de graphe ainsi que les algorithmes de parcours en largeur et en profondeur. Le travail s'est déroulé sur une durée de 4 heures.

46. Implémentation de graphe :

Pour commencer, nous avons mis en place la structure de graphe en utilisant un tableau de listes d'adjacence. Cette structure nous permet de représenter efficacement les relations entre les sommets du graphe. Nous avons également développé des fonctions pour ajouter des arêtes au graphe (addEdgeInGraph), créer un graphe aléatoire (createGraph) et afficher le graphe dans la console (printConsoleGraphe). L'affichage d'un graphe dans la console a été conçu de manière à fournir une représentation visuelle claire de la structure du graphe.

47. Complexité des fonctions :

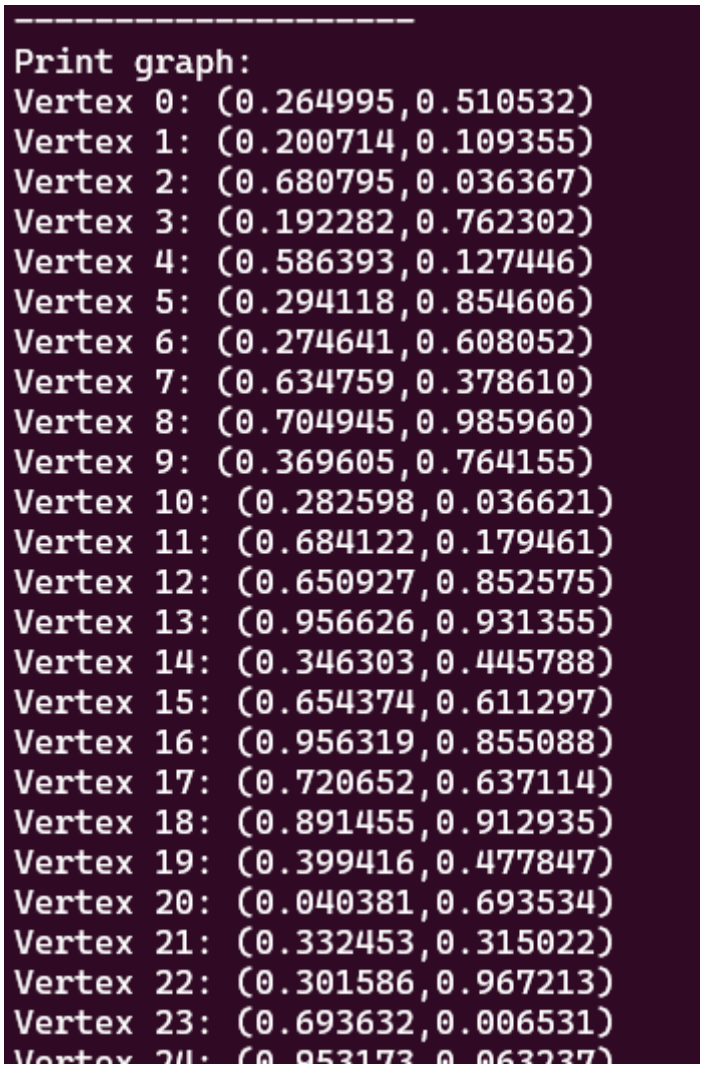
addEdgeInGraph	O(1)
createGraph	$O(V^2)$
printConsoleGraph	O(V + E)

Où V est le nombre de sommets et E le nombres des arêtes.

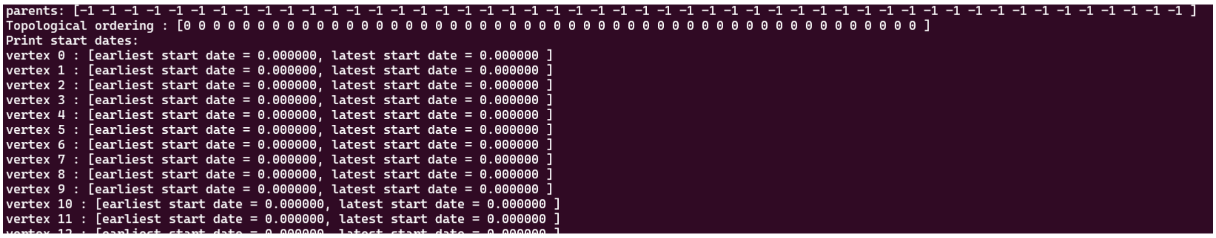
48. Trace d'execution de la 1ere partie :

```
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/graph$ make clean
rm -f graph.o testgraph.o ../queue/queue.o ../stack/stack.o ../list/list.o
rm -f *~
rm -f testgraph
rm -f *.svg
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/graph$ make all
gcc -o graph.o -c graph.c
gcc -o testgraph.o -c testgraph.c
gcc -o ../queue/queue.o -c ../queue/queue.c
gcc -o ../stack/stack.o -c ../stack/stack.c
gcc -o ../list/list.o -c ../list/list.c
gcc -o testgraph graph.o testgraph.o ../queue/queue.o ../stack/stack.o ../list/list.o -Wall -lm
mt@Razor-M18:~/Algo/TP-TOUJANI-sans_binome-2024/graph$
```

Figure 28 Exécution Correcte sans warning de Graph

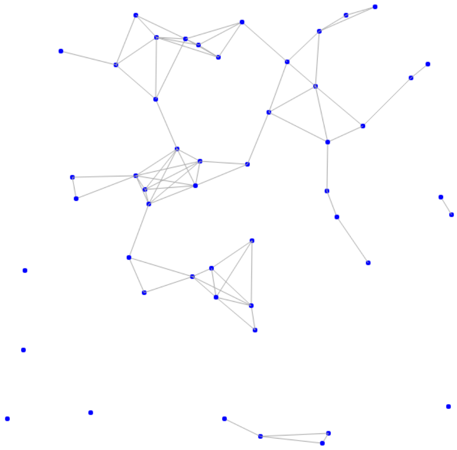



```
Vertex 0: 28 ->  19 ->  14 ->  6 ->  28 ->  19 ->  14 ->  6 ->  0 ->  0 -> NULL
Vertex 1: 49 ->  10 ->  49 ->  10 ->  1 ->  1 -> NULL
Vertex 2: 23 ->  11 ->  4 ->  23 ->  11 ->  4 ->  2 ->  2 -> NULL
Vertex 3: 48 ->  37 ->  31 ->  5 ->  48 ->  37 ->  31 ->  5 ->  3 ->  3 -> NULL
Vertex 4: 11 ->  11 ->  4 ->  4 ->  2 ->  2 -> NULL
Vertex 5: 48 ->  37 ->  31 ->  22 ->  9 ->  48 ->  37 ->  31 ->  22 ->  9 ->  5 ->  5 ->  3 ->  3 -> NULL
Vertex 6: 6 ->  6 ->  0 ->  0 -> NULL
Vertex 7: 7 ->  7 -> NULL
Vertex 8: 12 ->  12 ->  8 ->  8 -> NULL
Vertex 9: 48 ->  37 ->  31 ->  27 ->  48 ->  37 ->  31 ->  27 ->  9 ->  9 ->  5 ->  5 -> NULL
Vertex 10: 47 ->  33 ->  47 ->  33 ->  10 ->  10 ->  1 ->  1 -> NULL
Vertex 11: 25 ->  25 ->  11 ->  11 ->  4 ->  2 ->  4 ->  2 -> NULL
Vertex 12: 35 ->  35 ->  12 ->  12 ->  8 ->  8 -> NULL
Vertex 13: 38 ->  34 ->  18 ->  16 ->  38 ->  34 ->  18 ->  16 ->  13 ->  13 -> NULL
Vertex 14: 28 ->  21 ->  19 ->  28 ->  21 ->  19 ->  14 ->  14 ->  0 ->  0 -> NULL
Vertex 15: 29 ->  17 ->  29 ->  17 ->  15 ->  15 -> NULL
Vertex 16: 39 ->  38 ->  34 ->  18 ->  39 ->  38 ->  34 ->  18 ->  16 ->  16 ->  13 ->  13 -> NULL
Vertex 17: 44 ->  29 ->  44 ->  29 ->  17 ->  17 ->  15 ->  15 -> NULL
Vertex 18: 39 ->  38 ->  34 ->  39 ->  38 ->  34 ->  18 ->  18 ->  16 ->  13 ->  16 ->  13 -> NULL
Vertex 19: 28 ->  28 ->  19 ->  19 ->  14 ->  0 ->  14 ->  0 -> NULL
Vertex 20: 40 ->  40 ->  20 ->  20 -> NULL
Vertex 21: 28 ->  28 ->  21 ->  21 ->  14 ->  14 -> NULL
Vertex 22: 48 ->  32 ->  48 ->  32 ->  22 ->  22 ->  5 ->  5 -> NULL
Vertex 23: 23 ->  23 ->  2 ->  2 -> NULL
Vertex 24: 24 ->  24 -> NULL
Vertex 25: 46 ->  43 ->  30 ->  46 ->  43 ->  30 ->  25 ->  25 ->  11 ->  11 -> NULL
Vertex 26: 42 ->  42 ->  26 ->  26 -> NULL
Vertex 27: 27 ->  27 ->  9 ->  9 -> NULL
Vertex 28: 28 ->  28 ->  21 ->  19 ->  14 ->  0 ->  21 ->  19 ->  14 ->  0 -> NULL
Vertex 29: 29 ->  29 ->  17 ->  15 ->  17 ->  15 -> NULL
Vertex 30: 46 ->  43 ->  46 ->  43 ->  30 ->  30 ->  25 ->  25 -> NULL
Vertex 31: 48 ->  37 ->  48 ->  37 ->  31 ->  31 ->  9 ->  5 ->  3 ->  9 ->  5 ->  3 -> NULL
```



```
parents: [-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 ]
Topological ordering : [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
Print start dates:
vertex 0 : [earliest start date = 0.000000, latest start date = 0.000000 ]
vertex 1 : [earliest start date = 0.000000, latest start date = 0.000000 ]
vertex 2 : [earliest start date = 0.000000, latest start date = 0.000000 ]
vertex 3 : [earliest start date = 0.000000, latest start date = 0.000000 ]
vertex 4 : [earliest start date = 0.000000, latest start date = 0.000000 ]
vertex 5 : [earliest start date = 0.000000, latest start date = 0.000000 ]
vertex 6 : [earliest start date = 0.000000, latest start date = 0.000000 ]
vertex 7 : [earliest start date = 0.000000, latest start date = 0.000000 ]
vertex 8 : [earliest start date = 0.000000, latest start date = 0.000000 ]
vertex 9 : [earliest start date = 0.000000, latest start date = 0.000000 ]
vertex 10 : [earliest start date = 0.000000, latest start date = 0.000000 ]
vertex 11 : [earliest start date = 0.000000, latest start date = 0.000000 ]
```

Figure 29 Trace d'exécution de Graph



Graph 1 Exemple de Graph

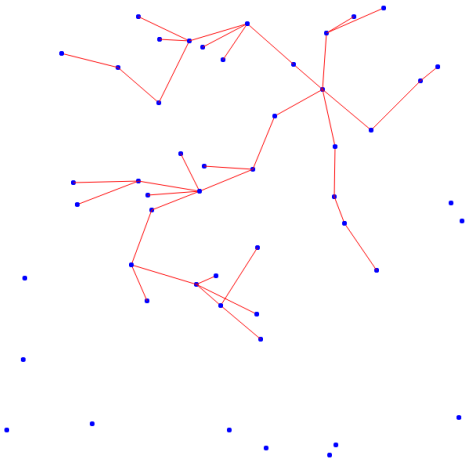
49. Parcours en profondeur et en largeur :

Nous avons ensuite mis en œuvre les algorithmes de parcours en profondeur (DFS) et en largeur (BFS). Ces algorithmes nous permettent d'explorer tous les sommets accessibles à partir d'un sommet donné. Pendant le parcours, nous avons stocké la structure sous-jacente de l'arbre couvrant dans le champ parents de la structure de graphe.

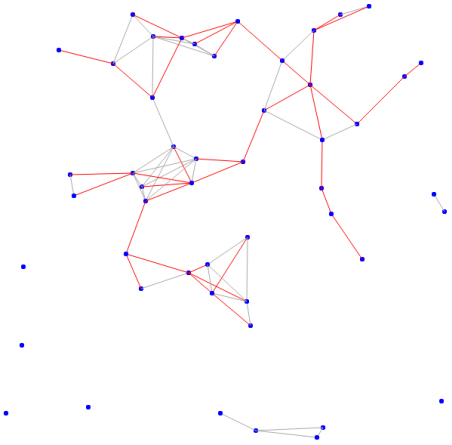
Complexité des deux fonctions	
graphDFS	O(V + E)
graphBFS	O(V + E)

Tableau 9 Complexité de DFS et BFS

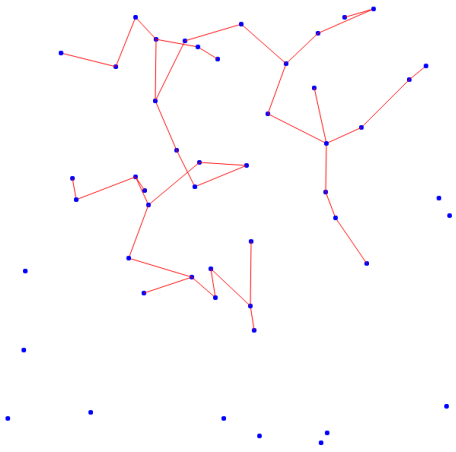
Où V est le nombre de sommets et E le nombres des arêtes.



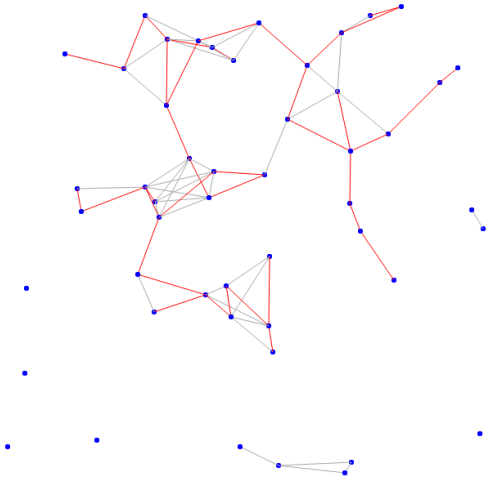
Graph 2 BFS Tree



Graph 3 BFS Graph Tree



Graph 4 DFS Tree

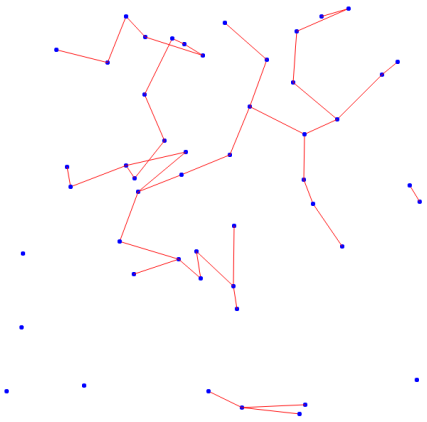


Graph 5 DFS Graph Tree

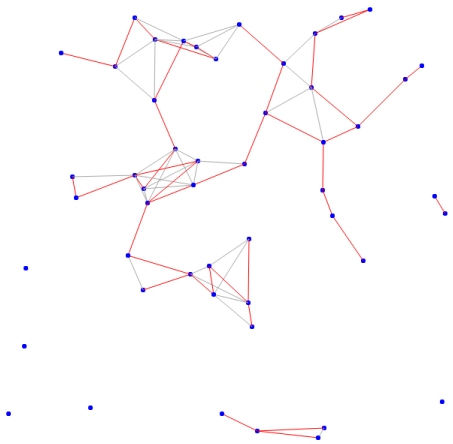
50. Détermination du nombre de composantes connexes:

Enfin, nous avons développé une fonction pour calculer le nombre de composantes connexes dans un graphe non orienté. Cette fonction effectue plusieurs parcours en profondeur ou en largeur à partir de sommets non visités, en marquant les sommets visités pendant chaque parcours. Le nombre total de parcours effectués correspond au nombre de composantes connexes dans le graphe.

Complexité : O(V + E), où V est le nombre de sommets et E le nombres des arêtes.



Graph 6 Components Tree



Graph 7 Components Graph Tree

51. Conclusion :

En conclusion, ce travail pratique nous a permis de mieux comprendre la structure et le fonctionnement des graphes, ainsi que les algorithmes de parcours associés. Nous avons réussi à implémenter les fonctionnalités requises avec succès, ce qui constitue une étape importante dans notre apprentissage de l'informatique et de l'algorithmique.

GRAPHES ORIENTES ACYCLIQUES

Ce rapport documente mon approche pour résoudre plusieurs problèmes classiques sur les graphes orientés acycliques (DAG). J'ai travaillé sur l'implémentation de trois fonctionnalités principales : l'ordonnancement topologique des sommets, le calcul des dates au plus tôt et des dates au plus tard pour chaque sommet dans le DAG.

52. Ordonnancement topologique des sommets :

52.1. Description de la fonction topologicalSort :

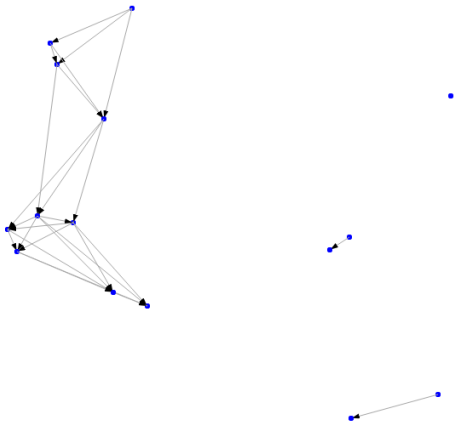
La fonction topologicalSort est chargée de déterminer un ordre topologique des sommets dans le DAG. J'ai choisi d'implémenter cet algorithme en utilisant une variation de la recherche en profondeur (DFS). L'idée principale est de visiter chaque sommet du graphe et d'effectuer un parcours en profondeur à partir de chaque sommet non encore visité. Lorsque j'atteins un sommet terminal, c'est-à-dire un sommet sans prédécesseur non visité, je l'ajoute à l'ordre topologique.

52.2. Pseudo-code de l'algorithme :

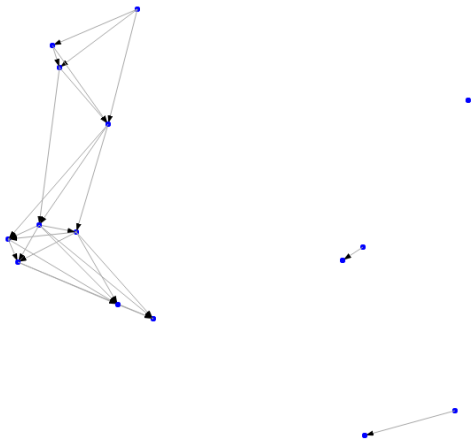
```
Pour chaque sommet s dans le graphe :
    Si s n'a pas été visité :
        Effectuer un parcours en profondeur à partir de s
        Ajouter s à l'ordre topologique
```

52.3. Complexité de l'algorithme :

La complexité de cet algorithme est O(V + E), où V est le nombre de sommets et E est le nombre d'arêtes dans le graphe.



Graph 8 Graph Support



Graph 9 Topo

53. Calcul des dates au plus tôt :

53.1. Description de la fonction computeEarliestStartDates :

Cette fonction est chargée de calculer la date au plus tôt pour chaque sommet du DAG. Pour cela, j'utilise une approche dynamique en parcourant les sommets du graphe dans un ordre topologique. À chaque sommet, je calcule la date au plus tôt comme le maximum des dates au plus tôt de ses prédécesseurs, plus la durée de l'arête correspondante.

53.2. Pseudo-code de l'algorithme :

```
Pour chaque sommet s dans l'ordre topologique :

    Si s est une source :

        La date au plus tôt de s est fixée à 0
```

```
Sinon :
    Pour chaque prédécesseur s' de s :
        Calculer dtot(s) = max(dtot(s') + durée de l'arête (s', s))
    La date au plus tôt de s est dtot(s)
```

53.3. Complexité de l'algorithme :

La complexité de cet algorithme est O(V + E), où V est le nombre de sommets et E est le nombre d'arêtes dans le graphe.

54. Calcul des dates au plus tard :

54.1. Description de la fonction computeLatestStartDates :

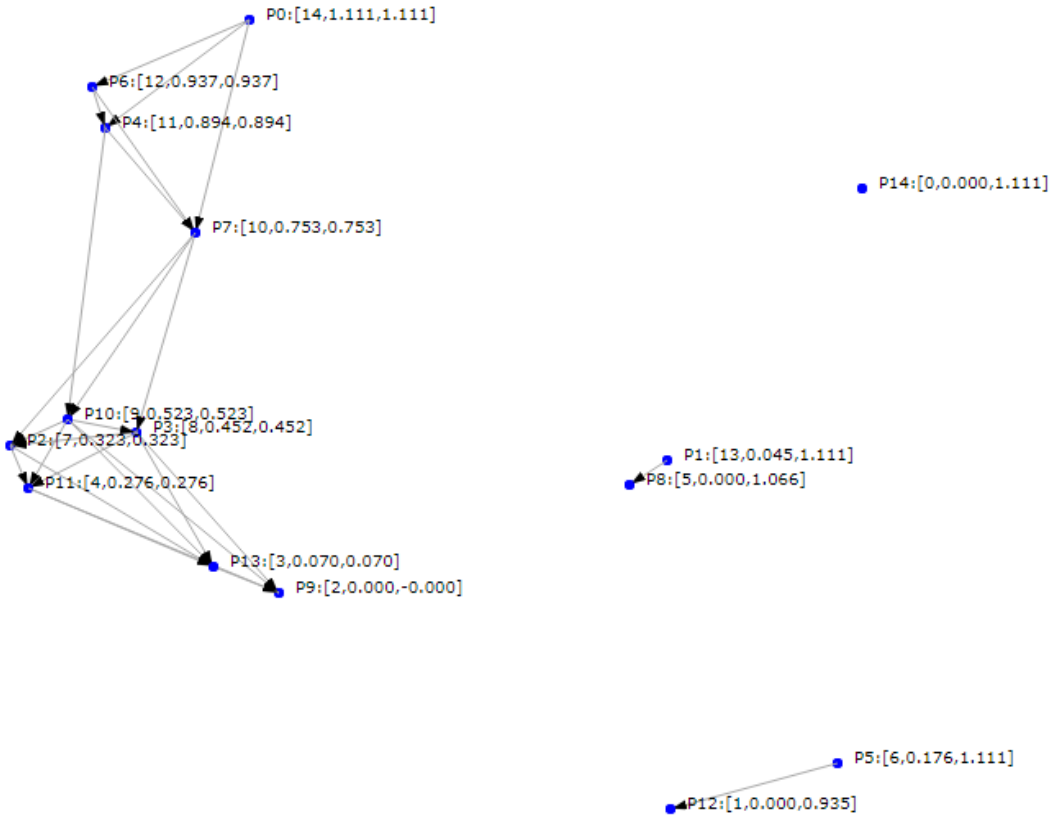
Cette fonction est similaire à computeEarliestStartDates, mais elle calcule les dates au plus tard pour chaque sommet du DAG. J'utilise une approche similaire en parcourant les sommets dans l'ordre topologique inverse et en calculant les dates au plus tard à partir des successeurs.

54.2. Pseudo-code de l'algorithme :

```
Pour chaque sommet s dans l'ordre topologique inverse :
    Si s est un puits :
        La date au plus tard de s est fixée à la date au plus tôt maximale
    Sinon :
        Pour chaque successeur s' de s :
            Calculer dtard(s) = min(dtard(s') - durée de l'arête (s, s'))
        La date au plus tard de s est dtard(s)
```

54.3. Complexité de l'algorithme :

La complexité de cet algorithme est O(V + E), où V est le nombre de sommets et E est le nombre d'arêtes dans le graphe.



Graph 10 Dates

55. Conclusion :

En conclusion, j'ai réussi à implémenter avec succès les fonctionnalités demandées pour travailler avec les DAG. Mes algorithmes sont efficaces et fournissent des résultats corrects pour l'ordonnancement topologique des sommets ainsi que le calcul des dates au plus tôt et au plus tard. La complexité de mes algorithmes reste raisonnable, ce qui les rend adaptés à une utilisation dans des applications réelles.

ARBRES COUVRANTS MINIMUMS

Ce rapport décrit mon travail sur l'implémentation de l'algorithme de Prim pour calculer un arbre couvrant minimum (ACM) dans un graphe non orienté et pondéré. L'algorithme de Prim repose sur une structure de tas pour construire cet arbre couvrant minimum à partir d'un sommet donné.

56. Algorithme de Prim :

56.1. Description de la fonction Prim :

La fonction Prim que j'ai implémentée prend en entrée un graphe non orienté, représenté par une structure de graphe, ainsi qu'un sommet initial s. L'algorithme de Prim calcule une forêt couvrante minimale du graphe à partir du sommet s. La forêt couvrante minimale est stockée dans le champ parents de la structure de graphe.

56.2. Implémentation de l'algorithme :

J'ai utilisé une implémentation basée sur un tas (heap) pour maintenir les sommets à explorer en fonction de leur priorité. Initialement, tous les sommets sont marqués avec une priorité infinie, sauf le sommet initial s, qui a une priorité de 0. Ensuite, tant que le tas n'est pas vide, je retire le sommet de priorité minimale, explore ses voisins et met à jour les priorités dans le tas en fonction des distances.

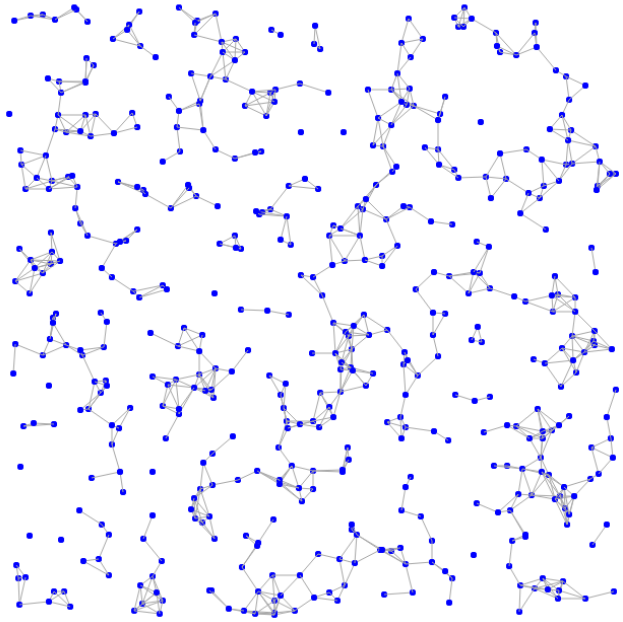
56.3. Pseudo-code de l'algorithme :

```
1. Initialiser un tas F avec tous les sommets du graphe

2. Marquer tous les sommets avec une priorité infinie, sauf le sommet initial s avec une priorité de 0

3. Tant que F n'est pas vide :

      a. Retirer le sommet de priorité minimale u_min de F

      b. Pour chaque voisin v de u_min :

            i. Si v est dans F et la distance entre u_min et v est plus petite que sa priorité actuelle :

                  - Mettre à jour la priorité de v dans F
```

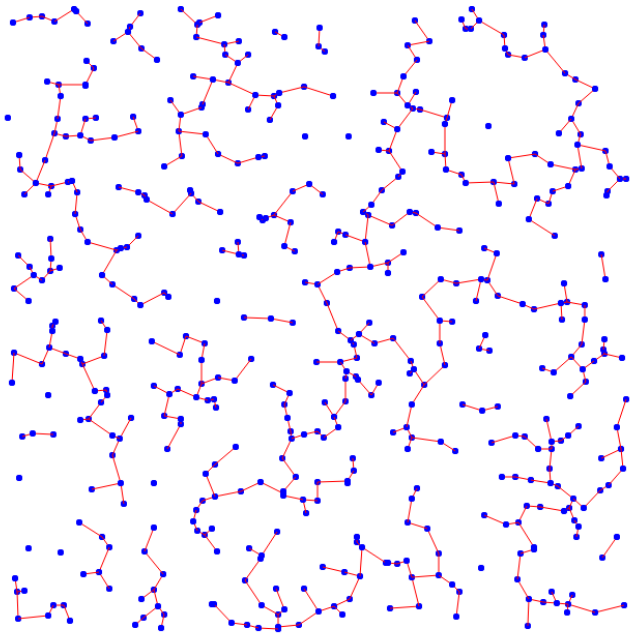
```
        - Mettre à jour le parent de v dans la forêt couvrante
4. Retourner la forêt couvrante minimale
```

56.4. Complexité de l'algorithme :

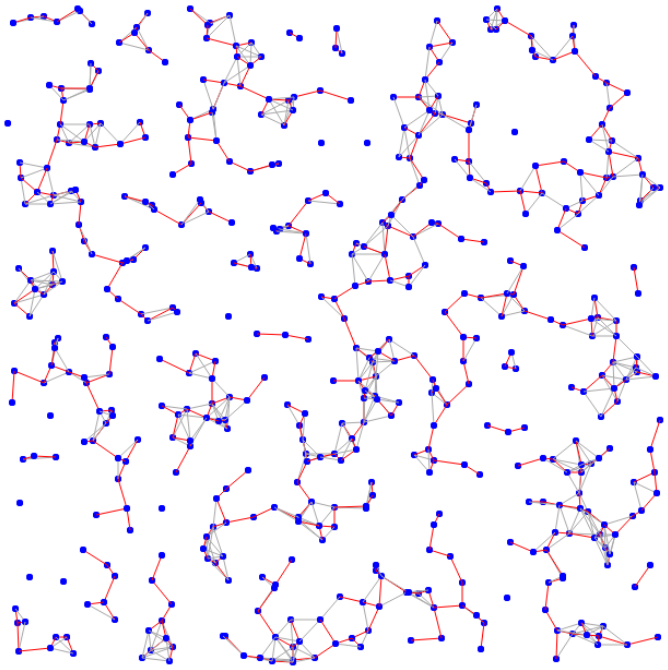
La complexité de cet algorithme dépend principalement de l'implémentation de la structure de tas. Dans le pire des cas, la complexité est O(E log V), où E est le nombre d'arêtes et V est le nombre de sommets dans le graphe.



Graph 11 Graph Support



Graph 12 Prim Tree



Graph 13 Prim Graph Tree

56.5. Conclusion :

En conclusion, j'ai réussi à implémenter avec succès l'algorithme de Prim pour calculer un arbre couvrant minimum dans un graphe non orienté et pondéré. Mon implémentation repose sur une structure de tas efficace pour maintenir les sommets à explorer en fonction de leur priorité. Cette solution est adaptée pour être utilisée dans des applications réelles nécessitant la recherche d'un arbre couvrant minimum.

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Ecole Publique d'ingénieures et d'ingénieurs en 3 ans

6 boulevard Maréchal Juin, CS 45053
14050 CAEN cedex 04