Tom GUERRIER

Anys Hadjidj

Mael LECARPENTIER

Projet semestriel: Gestion d'un Agenda

L2-Plus 2023-2024

Sommaire :

1) Introduction

Dans l'ère numérique actuelle, la gestion efficace du temps et des tâches est devenue un défi quotidien pour de nombreuses personnes. Dans cette perspective, notre projet vise à répondre à cette exigence croissante en développant un programme innovant dédié à la gestion d'agenda. Cependant, plutôt que de se limiter aux approches classiques des listes chaînées ou des arbres, notre vision repose sur l'implémentation d'une structure de données intermédiaire en implémentant plusieurs niveaux. Les listes à niveaux constituent une extension évoluée des structures de données, offrant une approche plus complète pour le stockage d'entiers triés par ordre croissant. Cette généralisation inclut la possibilité d'avoir plusieurs niveaux de pointage, avec la hauteur de la liste définissant le nombre maximum de niveaux. Chaque cellule de la liste comporte un ensemble de pointeurs, permettant des connexions plus complexes que celles d'une liste chaînée traditionnelle. La règle générale pour le pointage des cellules est la suivante : Si une cellule pointe sur une autre cellule à un certain niveau, elle pointe sur d'autres cellules à tous les niveaux inférieurs.

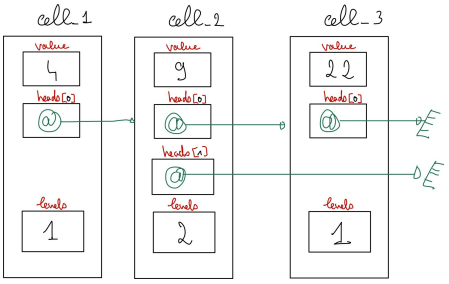
Représentation d'une liste chaînée traditionnelle (avec un seul niveau pour chaque cellules)

```
[list head_0 @-]-->[1|@-]-->[2|@-]-->[3|@-]-->[4|@-]-->[5|@-]-->[6|@-]-->[7|@-]-->[8|@-]-->[9|@-]-->[10|@-]-->[11|@-]-->[12|@-]-->[13|@-]-->[14|@-]-->[15|@-]-->NULL
```

2) Représentation des structures de données choisies

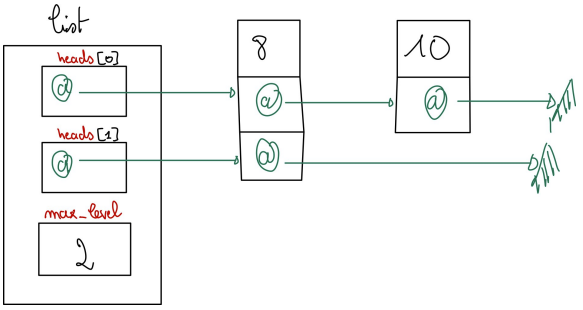
Cette structure représente chaque cellule de la liste. Elle contient le champ "value" qui stock l'entier de la cellule. Le champ "levels" contient le nombre de niveaux que comporte la cellule. Le champ "nexts" est un tableau de pointeurs qui peut avoir entre 1 niveau et le nombre de niveaux que la liste possède.

```c
typedef struct s_d_cell
{
    int value;
    int levels;
    struct s_d_cell** nexts;
}t_d_cell;
```



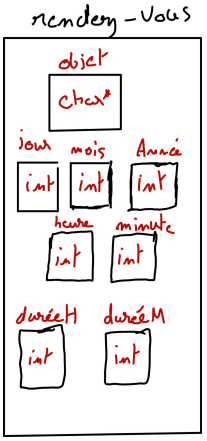
Cette structure représente la liste. Elle comporte le champ “heads” qui est un tableau de pointeurs qui peut pointer vers NULL ou une cellule. Ce tableau peut avoir entre 1 et n niveaux.

```c
typedef struct s_d_list
{
    t_d_cell** heads;
    int max_level;
} t_d_list;
```



Cette structure représente un rendez-vous, les champs représentent toutes les informations a savoir sur un rendez-vous tel que l'horaire (année, mois, jour, heure, minute) ainsi que sa durée (duréeH, duréeM) et l’objet du rendez-vous (objet).

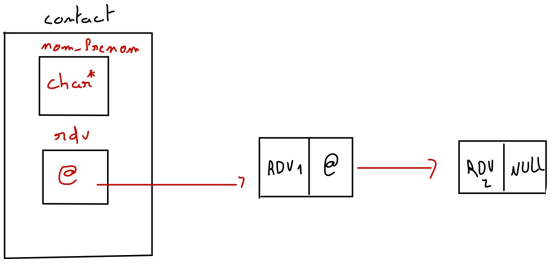
```c
typedef struct rendezvous
{
    int jour, mois, année;
    int heure, minute;
    int duréeH, duréeM;
    char* objet;
}rendezvous;
```



Cette structure représente un contact. Les champs représentent les informations du contact (nom_prenom) ainsi qu’un pointeur pointant vers la cellule de son premier rendez-vous. Cette structure prend le rôle d’une liste (son champ *rdv est équivalent à un champ *rdv)

```c
typedef struct contact
{
    char* nom_prenom;
    cellrdv* rdv

}contacts;
```



3) Explications des différentes fonctions principales et exécutions

a) Insertion d'une cellule à niveau

L'insertion d'une cellule dans une liste à niveau demande un peu plus réflexion qu'une insertion dans une simple liste chaînée. Nous devons commencer l'insertion dans le niveau le plus haut de la cellule et descendre au niveau le plus bas. Nous devons vérifier si la cellule que nous devons insérer est supérieure à la cellule précédente et inférieure à la cellule suivante pour ensuite aller au niveau inférieur. Il peut y avoir certains cas particuliers:

- La liste est vide: Nous appelons donc la fonction pour insérer la cellule en tête de liste.
- La valeur de la cellule est inférieur à la tête d'un niveau: on insère alors la cellule en tête de niveau
- La valeur de la cellule est supérieur à la valeur de la dernière cellule d'un niveau: on insert la cellule en fin de la liste du niveau

```
List a 1 niveaux vide :
[list head_0 @-]-->NULL
[list head_1 @-]-->NULL
Insertion de la valeur 3 avec 1 niveaux :
[list head_0 @-]-->[3|@-]-->NULL
[list head_1 @-]----------->NULL
Insertion de la valeur 2 avec 2 niveaux :
[list head_0 @-]-->[2|@-]-->[3|@-]-->NULL
[list head_1 @-]-->[2|@-]----------->NULL
Insertion de la valeur 6 avec 1 niveaux :
[list head_0 @-]-->[2|@-]-->[3|@-]-->[6|@-]-->NULL
[list head_1 @-]-->[2|@-]--------------------->NULL
```

b) Recherche dichotomique

Le principe de liste à niveau de niveau n'a aucun sens sans une recherche dichotomique. Pour effectuer une recherche dichotomique, nous devons commencer la recherche dans le niveau le plus élevé. Si notre valeur est supérieure à la valeur dans la cellule parcourue, nous descendons d'un niveau et commençons la recherche à la tête de la liste.
Si la valeur est inférieure à la cellule parcourue mais est supérieure à la valeur de la cellule suivante, alors nous descendons d'un niveau et reprenons la recherche à la cellule suivante (donc le next du niveau précédent). La fonction nous retourne 1 si la valeur a été trouvée ou 0 si elle n'a pas été trouvée.

```
[list head_0 @-]-->[1|@-]-->[2|@-]-->[3|@-]-->[4|@-]-->[5|@-]-->[6|@-]-->[7|@-]-->NULL
[list head_1 @-]----------->[2|@-]----------->[4|@-]----------->[6|@-]----------->NULL
[list head_2 @-]----------------------------->[4|@-]----------------------------->NULL
Recherche de la valeur 4 : La valeur 4 a ete trouvee
Recherche de la valeur 12 : La valeur 12 n'a pas ete trouvee
```

c) Création d'une liste à partir de n niveaux

Cette fonction nous permet de créer une liste déjà remplie avec chacun de ses niveaux remplis. Pour cette fonction, nous avons eu besoin d'un tableau qui nous permet de déterminer l'endroit où insérer chaque cellule.

```
int* CreateTab(int n);
```

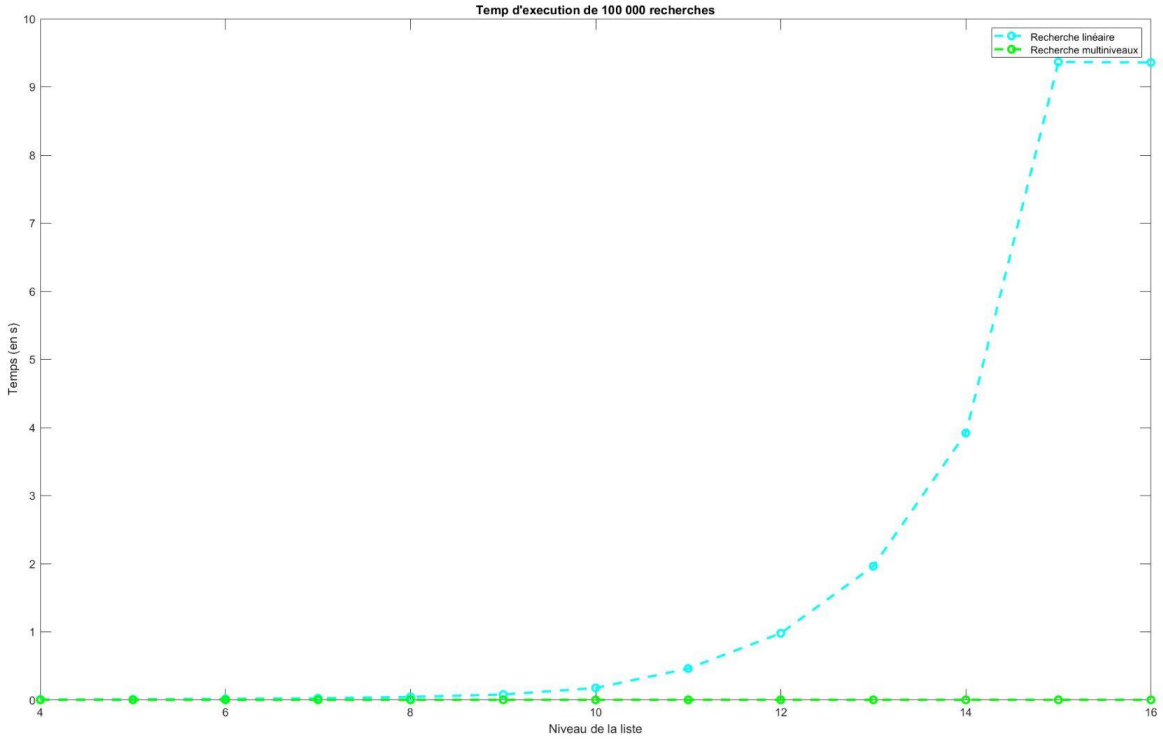
Cette fonction est indispensable pour effectuer nos tests sur notre fonction sur la recherche dichotomique.

d) Affichage ordonnée

L'affichage ordonné nous permet d'avoir une meilleure lisibilité de notre liste. En effet, elle nous permet d'aligner les différents niveaux de chaque cellules. Pour réaliser ce programme nous devons détecter les décalages présents pour chaque niveau. Pour ce faire, nous comparons la liste de chaque niveau avec celle du niveau 0 qui est forcément complète, si les valeurs diffèrent, nous devons faire un décalage. Voici un exemple d'affichage ordonné pour une liste à 4 niveaux :

```
[list head_0 @-]-->[1|@-]-->[2|@-]-->[3|@-]-->[4|@-]-->[5|@-]-->[6|@-]-->[7|@-]-->[8|@-]-->[9|@-]-->[10|@-]-->[11|@-]-->[12|@-]-->[13|@-]-->[14|@-]-->[15|@-]-->NULL
[list head_1 @-]----------->[2|@-]----------->[4|@-]----------->[6|@-]----------->[8|@-]----------->[10|@-]----------->[12|@-]----------->[14|@-]----------->NULL
[list head_2 @-]----------------------------->[4|@-]----------------------------->[8|@-]----------------------------->[12|@-]----------------------------->NULL
[list head_3 @-]------------------------------------------------------------------>[8|@-]------------------------------------------------------------------>NULL
```

4) Calcul de l'efficacité des fonctions de recherche



Nous avons effectué le graphe modélisant le temps mis à l'exécution de 100000 recherches de valeurs aléatoire dans notre liste. Nous comparons ici deux modes de recherche : la recherche dichotomique (recherche multi-niveaux) et la recherche linéaire qui utilise seulement le niveau le plus bas (comme dans une liste chaînée classique). Nous pouvons remarquer que pour les listes ayant jusqu'à 2^7 - 1 valeurs, le temps obtenu pour effectuer toutes les recherches est sensiblement le même pour les deux modes. Cependant à partir de 2^8 - 1 valeurs, le temps mis à faire ces recherches pour la recherche linéaire augmente de manière exponentielle, tandis que le temps mis avec la recherche dichotomique reste le même. On peut donc en conclure que pour des grandes listes avec de nombreuses valeurs, il est préférable d'utiliser une recherche multi-niveaux pour faire des recherches afin de minimiser la complexité et le temps d'exécution.

5) Parties effectuées et non effectuées

Partie I et II :
Fonctions réalisées :

- Création de cellule (CreateCell)
- Création d'une liste à niveau vide (CreateEmptyList)
- Insertion d'une cellule à niveau en tête de liste (insertHeadCell)
- Afficher l'ensembles des cellules de la liste pour un niveau donné (displayListLevel)
- Afficher tous les niveaux de la liste en alignant les cellules (displayAlignedList)
- Insérer une cellule à niveau dans la liste, au bon endroit (insertCell)
- Fonction de calcul du temps d'exécution pour un nombre de recherches (CalculTimer)
- Création d'une liste à n niveaux (CreateListNvalue)

Partie III :
Fonctions réalisées :

- Création des structures (rendez-vous, contacts, …)
- Fonction pour les saisies utilisateurs (scanString)
- Création d'un contact (CreateContact)

Fonctions à finaliser :

- Création d'un rendez-vous (Createrdv)
- Insertion d'un rendez-vous pour un contact (InsertRdv)
- Affichage des rendez-vous d'un contact (displayRDV)
- Affichage du menu (PrintMenu)

Fonctions à créer :

- Recherche d'un contact
- Suppression d'un rendez-vous
- Sauvegarder le fichier de tous les rendez-vous
- Charger un fichier de rendez-vous

6) Conclusion

En conclusion, notre projet de gestion d'agenda, basé sur une structure de données à plusieurs niveaux, représente une avancée significative dans la recherche de solutions innovantes pour la gestion du temps. Malgré certaines parties encore à finaliser, notre projet nous a permis d'approfondir notre compréhension des structures de données complexes et d'améliorer notre collaboration d'équipe. Les fonctions clés, telles que l'insertion à niveau et la recherche dichotomique, ont démontré leur efficacité.

Lien de notre espace GIT-Hub : https://github.com/Maeell04/Projet-Gestion-Agenda.git