L2

Gestion d'un agenda

Rapport Projet

Jung-hoon KIM
Maxime GIMER
Hugo DELWAIL

Sommaire

Introduction

Dans le cadre de notre deuxième année en cycle préparatoire à l'école d'ingénieur EFREI Paris, nous avons eu pour tâche la réalisation d'un projet informatique dans la matière **Algorithmique et structure de données 2** .

Notre objectif ce semestre était de créer une application qui permet de gérer un agenda en utilisant une structure de données intermédiaires entre les listes chaînées et les arbres . Une structure qui permettra ,dans le même temps,de:

- Conserver un ordre entre ses éléments
- Effectuer les opérations standard avec une complexité qui reste faible, notamment plus faible qu'avec une liste simplement chaînée.

Pour cela nous allons effectuer ce projet en 3 parties , premièrement par l'implémentation des classes à niveaux qui stocke des entiers ensuite étudier sur la complexité de la recherche dans une liste à niveau et enfin la création du programme de gestion de l'agenda.

Partie 1 Implémentation des listes à niveaux – stockage d'entiers

1.Créer une cellule : on donne sa valeur et le nombre de niveaux que possède cette cellule, pour obtenir un pointeur vers cette cellule

```h
#ifndef CELL_H
#define CELL_H

typedef struct s_cell
{
    int value;
    int levels;
    struct s_cell **next;
}t_cell;


t_cell *create_level_cell(int value, int levels);

#endif
```

Création de la Structure de la cellule , *value* pour valeurs stockées dans la cellule, et levels pour les niveaux de la cellule et un pointeur next vers les cellules suivantes à différents niveaux

On crée donc une fonction qui prend en paramètre des valeurs et des niveaux pour pouvoir retourner une cellule

```c
t_cell *create_level_cell(int value, int levels) {

    t_cell *cell = malloc(sizeof(t_cell));

    cell->value = value;
    cell->levels = levels;

    cell->next = malloc(sizeof(t_cell*) * levels);

    for (int i = 0; i < levels; i++) {
        cell->next[i] = NULL;
    }

    return cell;
}
```

malloc pour allouer la mémoire pour une nouvelle instance de t_cell.

On initialise les champs de la structure.

On alloue de la mémoire pour le tab de pointeur t_cell* avec un pointeur à chaque niveau (* levels)

Pour enfin initialiser chaque pointeur du tableau (ici a NULL)

2

```c
typedef struct s_level_list {
    t_cell *head;
    int max_levels;
} t_level_list;
```

un pointeur head qui va pointer vers la tête

max levels qui va définir le niveau maximal que la liste pourra avoir.

2.1 Créer une liste à niveau vide : on donnera le nombre maximal de niveaux que possède cette liste.

```c
t_level_list *create_level_list(int max_levels) {

    t_level_list *list = malloc(sizeof(t_level_list));
    list->max_levels = max_levels;
    list->head = create_level_cell(-1, max_levels);

    return list;
}
```

La create_level_list fait appel a la fonction create_level_cell pour initialiser la tête de la liste avec un -1 et son nombre de niveaux a une valeur de max levels.-1 indiquera une valeur sentinelle .

2.2 Insérer une cellule à niveaux en tête de liste (attention à bien tenir compte du nombre de niveaux de la cellule)

```c
void insert_cell_at_head(t_level_list *list, t_cell *new_cell) {

    for (int i = 0; i < new_cell->levels; i++) {
        if (i < list->max_levels) {
            new_cell->next[i] = list->head->next[i];
            list->head->next[i] = new_cell;
        }
    }
}
```

La fonction fait appel à une liste et à une cellule : Pour chaque niveau i de la nouvelle cellule, le code vérifie si i est inférieur au nombre maximum de niveaux dans la liste (list->max_levels). Cette vérification est nécessaire pour éviter d'ajouter des niveaux qui n'existent pas dans la liste.
Si le niveau est valide, la cellule est insérée à ce niveau. On fait pointer le next[i] de la nouvelle cellule vers la cellule qui était précédemment en tête de liste à ce niveau. Ensuite, on met à jour la tête de liste au niveau i pour qu'elle pointe vers la nouvelle cellule

2.3 Afficher l’ensembles des cellules de la liste pour un niveau donné

```c
void display_cells_at_level(t_level_list *list, int level) {
   // Affichage des cellules du niveau donné
    t_cell *current = list->head->next[level];
    printf("Cellules au niveau %d: ", level);

    while (current != NULL) {
        printf("%d ", current->value);
        current = current->next[level];
    }

    printf("\n");
}
```

Ici nous allons parcourir les cellules en commençant par la tête `t_cell *current = list->head->next[level];`
Et faire une boucle while pour afficher la cellule du niveau puis la cellule suivante tant que le niveau n’est pas vide.

demonstration:

```
[list head_0 @-]-->[ 18|@-]-->[ 25|@-]-->[ 31|@-]-->[ 32|@-]-->[ 56|@-]-->[ 59|@-]-->[ 59|@-]-->[ 91|@-]-->NULL
[list head_1 @-]-->[ 18|@-]------------->[ 31|@-]-->[ 32|@-]-->[ 56|@-]-->[ 59|@-]------------->[ 91|@-]-->NULL
[list head_2 @-]-->[ 18|@-]----------------------->[ 32|@-]-->[ 56|@-]-->[ 59|@-]------------->[ 91|@-]-->NULL
[list head_3 @-]-->[ 18|@-]----------------------->[ 32|@-]------------->[ 59|@-]----------------------->NULL
[list head_4 @-]---------------------------------->[ 32|@-]------------->[ 59|@-]----------------------->NULL
```

```c
    insert_sorted(myList, create_level_cell(18, 3));
    insert_sorted(myList, create_level_cell(25, 1));
    insert_sorted(myList, create_level_cell(31, 2));
    insert_sorted(myList, create_level_cell(32, 5));
    insert_sorted(myList, create_level_cell(56, 3));
    insert_sorted(myList, create_level_cell(59, 5));
    insert_sorted(myList, create_level_cell(59, 1));
    insert_sorted(myList, create_level_cell(91, 3));
```

Ici par exemple au niveau 3 nous avons les cellules 18, 32,59:

```c
printf("Affichage de tous les cellules a un nv donne :\n");
display_cells_at_level(myList,3);
```

```
Affichage de tous les cellules a un nv donne :
Cellules au niveau 3: 18 32 59
```

2.4 Afficher tous les niveaux de la liste

```c
void display_all_levels(t_level_list *list) {
    // Vérification de la validité de la liste
    if (list == NULL) {
        printf("Liste non valide.\n");
        return;
    }
    // Affichage des cellules pour chaque niveau
    for (int level = 0; level < list->max_levels; level++) {
        t_cell *current = list->head->next[level];
        printf("[list head_%d @-]", level);

        while (current != NULL) {
            printf("-->[%d|@]", current->value);
            current = current->next[level];
        }

        printf("-->NULL\n");
    }
}
```

On commence par vérifier que la liste n'est pas vide
On fait une boucle for pour parcourir les niveaux en commençant par le niveau 0 , dans cette boucle nous allons créer un pointeur current qui pointe vers le premier élément (après la tête) de la liste au niveau level.
Et enfin une boucle while pour parcourir les cellules de chaque niveaux et afficher les valeurs de chaque cellules.

Demonstration :
Nous allons reprendre la liste précédente

```c
printf("Affichage de tous les niveaux de la liste :\n");
display_all_levels(myList);
```

```
Affichage de tous les niveaux de la liste :
[list head_0 @-]-->[18|@]-->[25|@]-->[31|@]-->[32|@]-->[56|@]-->[59|@]-->[59|@]-->[91|@]-->NULL
[list head_1 @-]-->[18|@]-->[31|@]-->[32|@]-->[56|@]-->[59|@]-->[91|@]-->NULL
[list head_2 @-]-->[18|@]-->[32|@]-->[56|@]-->[59|@]-->[91|@]-->NULL
[list head_3 @-]-->[18|@]-->[32|@]-->[59|@]-->NULL
[list head_4 @-]-->[32|@]-->[59|@]-->NULL
```

2.5 En option : afficher tous les niveaux de la liste, en alignant les cellules

Pour l'affichage en alignant les cellules , nous allons garder le code précédent sauf que pour l'affichage des valeurs de chaque cellules a chaque niveau si la cellule ne pointent pas vers des cellules valides nous allons afficher un espace à la place .

```c
t_cell *current_level_0 = list->head->next[0];
for (int i = 0; i < total_cells; i++) {
    if (current != NULL && current_level_0 != NULL && current->value == current_level_0->value) {
        printf("-->[ %d|@- ]", current->value);
        current = current->next[level];
        current_level_0 = current_level_0->next[0];
    } else {
        printf("------------");
        if (current_level_0 != NULL) {
            current_level_0 = current_level_0->next[0];
        }
    }
}
printf("-->NULL\n");
```

```c
printf("\nAffichage de tous les niveaux de la liste avec alignement :\n");
display_all_levels_aligned(myList);
```

```
Affichage de tous les niveaux de la liste avec alignement :
[list head_0 @-]-->[ 18|@- ]-->[ 25|@- ]-->[ 31|@- ]-->[ 32|@- ]-->[ 56|@- ]-->[ 59|@- ]-->[ 59|@- ]-->[ 91|@- ]-->NULL
[list head_1 @-]-->[ 18|@- ]-------------->[ 31|@- ]-->[ 32|@- ]-->[ 56|@- ]-->[ 59|@- ]-------------->[ 91|@- ]-->NULL
[list head_2 @-]-->[ 18|@- ]------------------------->[ 32|@- ]-->[ 56|@- ]-->[ 59|@- ]-------------->[ 91|@- ]-->NULL
[list head_3 @-]-->[ 18|@- ]------------------------->[ 32|@- ]-------------->[ 59|@- ]------------------------->NULL
[list head_4 @-]------------------------------------->[ 32|@- ]-------------->[ 59|@- ]------------------------->NULL
```

2.6 Insérer une cellule à niveau dans la liste, au bon endroit, de manière à ce que la liste reste triée par ordre croissant

```c
void insert_sorted(t_level_list *list, t_cell *new_cell) {
    // Vérification de la validité de la liste et de la nouvelle cellule
    if (list == NULL || new_cell == NULL) {
        return;
    }
    // Initialisation du tableau pour stocker les cellules précédentes
    t_cell *update[list->max_levels];
    t_cell *current = list->head;

    // Recherche de la position d'insertion pour chaque niveau
    for (int level = list->max_levels - 1; level >= 0; level--) {
        while (current->next[level] != NULL && current->next[level]->value < new_cell->value) {
            current = current->next[level];
        }
        update[level] = current;
    }
    // Insertion de la nouvelle cellule à chaque niveau concerné
    for (int level = 0; level < new_cell->levels; level++) {
        new_cell->next[level] = update[level]->next[level];
        update[level]->next[level] = new_cell;
    }
}
```

On commence par initialiser les cellules de mise à jour ,Un tableau update est initialisé pour stocker les cellules qui précéderont directement la nouvelle cellule à chaque niveau de la liste. Nous allons ensuite chercher sa position pour cela, nous allons parcourir la liste à partir du niveau le plus élevé jusqu'au niveau le plus bas. Cette position est déterminée en trouvant la dernière cellule dont la valeur est inférieure à celle de la nouvelle cellule.
La valeur de cette cellule va être stockée dans le tableau ‘update’.

Demonstration :

```
t_level_list *myList = create_level_list(5);
insert_sorted(myList, create_level_cell(32, 5));
insert_sorted(myList, create_level_cell(56, 3));
insert_sorted(myList, create_level_cell(18, 4));
insert_sorted(myList, create_level_cell(25, 1));
insert_sorted(myList, create_level_cell(31, 2));
insert_sorted(myList, create_level_cell(59, 5));
insert_sorted(myList, create_level_cell(59, 1));
insert_sorted(myList, create_level_cell(91, 3));
```

les cellules sont insérées dans l'ordre 32 56 18 25 31 59 59 91
mais sont affichés dans l'ordre croissant.

```
Affichage de tous les niveaux de la liste avec alignement :
[list head_0 @-]-->[ 18|@- ]-->[ 25|@- ]-->[ 31|@- ]-->[ 32|@- ]-->[ 56|@- ]-->[ 59|@- ]-->[ 59|@- ]-->[ 91|@- ]-->NULL
[list head_1 @-]-->[ 18|@- ]-------------->[ 31|@- ]-->[ 32|@- ]-->[ 56|@- ]-->[ 59|@- ]-------------->[ 91|@- ]-->NULL
[list head_2 @-]-->[ 18|@- ]------------------------->[ 32|@- ]-->[ 56|@- ]-->[ 59|@- ]-------------->[ 91|@- ]-->NULL
[list head_3 @-]-->[ 18|@- ]------------------------->[ 32|@- ]-------------->[ 59|@- ]------------------------->NULL
[list head_4 @-]----------------------------------->[ 32|@- ]-------------->[ 59|@- ]------------------------->NULL
```

Partie 1 - Implémentation des listes à niveaux – stockage d'entiers

- ☑ ~~1. Les cellules à niveau, avec les fonctionnalités suivantes :~~
- ☑ ~~- Créer une cellule : on donne sa valeur et le nombre de niveaux que possède cette cellule, pour obtenir un pointeur vers cette cellule~~
- ☑ ~~2. Les listes à niveau, avec les fonctionnalités suivantes :~~
- ☑ ~~- Créer une liste à niveau vide : on donnera le nombre maximal de niveaux que possède cette liste~~
- ☑ ~~- Insérer une cellule à niveaux en tête de liste (attention à bien tenir compte du nombre de niveaux de la cellule)~~
- ☑ ~~- Afficher l'ensembles des cellules de la liste pour un niveau donné~~
- ☑ ~~- Afficher tous les niveaux de la liste~~
- ☑ ~~- En option : afficher tous les niveaux de la liste, en alignant les cellules~~
- ☑ ~~- Insérer une cellule à niveau dans la liste, au bon endroit, de manière à ce que la liste reste triée par ordre croissant~~

- ☑ ~~Vous écrirez un programme (fonction main()) qui illustre toutes ces fonctionnalités~~

Partie 2 – Complexité de la recherche dans une liste à niveau

Ecrire deux fonctions de recherche :

· Une fonction de recherche ‘classique’ uniquement dans le niveau 0

```
t_cell *search_level_0(t_level_list *list, int value) {
    // Commence au début de la liste, qui est au niveau 0
    t_cell *current = list->head;
    // Parcourt la liste au niveau 0
    while (current != NULL) {
        // Vérifie si la cellule courante a la valeur recherchée
        if (current->value == value) {
            return current;
        }
        // Passe à la cellule suivante au niveau 0
        current = current->next[0];
    }

    return NULL;
}
```

On commence par initialiser un pointeur ‘current’ qui pointe vers la tête de la liste
Ensuite, tant que current n’est pas vide nous allons comparer si la valeur de current est égal à la valeur recherchée si oui elle retourne current sinon elle pointe vers le prochain.

· Une fonction de recherche à partir du niveau le plus haut

```
t_cell* search_all_levels(t_level_list *list, int value) {
    if (list == NULL || list->head == NULL) {
        return NULL;
    }
    t_cell *current = list->head;
    int level = list->max_levels - 1;
    // Parcourt tous les niveaux de la liste de saut
    while (level >= 0) {
        // Parcourt le niveau courant jusqu'à trouver un élément plus grand ou égal à la valeur recherchée
        while (current->next[level] != NULL && current->next[level]->value < value) {
            current = current->next[level];
        }
        // Vérifie si la valeur recherchée est trouvée au niveau courant
        if (current->next[level] != NULL && current->next[level]->value == value) {
            return current->next[level];  // Retourne l'élément trouvé
        }
        // Descend au niveau inférieur
        level--;
    }

    return NULL; // Retourne NULL si la valeur n'est pas trouvée
}
```

Pareil, on commence par initialiser un pointeur ‘current’ qui pointe vers la tête de la liste

On va parcourir tous les niveaux, et chaque niveau va être parcouru jusqu'à trouver un élément plus grand ou égal à la valeur recherchée. Si la valeur recherchée est trouvée au niveau courant nous allons retourner l'élément trouvée sinon on descend d'un niveau .

- Comparer le temps mis pour faire 1 000, 10 000, 100 000 recherches (rechercher une valeur au hasard) avec des valeurs de croissantes.

Pour effectuer cette comparaison, vous pourrez utiliser les fonctions de chronométrage fournies dans le module timer (timer.c / timer.h) disponible sur Moodle.

```
int initial_level = 7;
int final_level = 16;
int searches_per_level = 10000; // Nombre de recherches par niveau
```

On commence par initialiser le niveau de recherche initial et final et le nombre de recherche

```
    for (int level = initial_level; level <= final_level; level++) {
        int max_value = pow(2, level) - 1;
```

donc pour pour chaque niveau

```
  t_level_list *myList = create_level_list(level);

        // Création des cellules dans la liste
        for (int i = 0; i < max_value; i++) {
            insert_sorted(myList, create_level_cell(i, rand() % level +
1));
        }
```

Nous allons créer une liste contenant des cellules avec des valeurs aléatoires

Recherche au Niveau 0:

```
        startTimer();
        for (int i = 0; i < searches_per_level; i++) {
            search_level_0(myList, rand() % max_value);
        }
        stopTimer();

        char *time_level_0 = getTimeAsString();
```

Pour mesurer le temp on commence par startTimer pour commencer le timer, on fait une boucle for pour effectuer n fois la recherche ici dans notre cas n =10 000 et on cherche la valeur random générée aléatoirement .
le timer sera arrêté avec stop Timer, et sera stocké dans time level_0 avec la fonction getTime AsString en ms

Et on fait de même avec la recherche à tous les niveaux

```
    startTimer();
        for (int i = 0; i < searches_per_level; i++) {
            search_all_levels(myList, rand() % max_value);
        }
        stopTimer();

        char *time_all_levels = getTimeAsString();
```

Recherche d'une valeur inexistante :

```
int hors_list = max_value +1 ;
```

On crée une valeur supérieure à la valeur maximale .

```
search_level_0(myList, hors_list);
```

```
search_all_levels(myList, hors_list);
```

que nous allons rechercher.

1000 recherches :
val au hasard

```
Niveau: 7, Temps Niveau 0: 000.000 msTemps Tous Niveaux: 000.000 ms
Niveau: 8, Temps Niveau 0: 000.000 msTemps Tous Niveaux: 000.000 ms
Niveau: 9, Temps Niveau 0: 000.000 msTemps Tous Niveaux: 000.000 ms
Niveau: 10, Temps Niveau 0: 000.001 msTemps Tous Niveaux: 000.000 ms
Niveau: 11, Temps Niveau 0: 000.004 msTemps Tous Niveaux: 000.000 ms
Niveau: 12, Temps Niveau 0: 000.008 msTemps Tous Niveaux: 000.000 ms
Niveau: 13, Temps Niveau 0: 000.016 msTemps Tous Niveaux: 000.003 ms
Niveau: 14, Temps Niveau 0: 000.033 msTemps Tous Niveaux: 000.004 ms
Niveau: 15, Temps Niveau 0: 000.068 msTemps Tous Niveaux: 000.008 ms
Niveau: 16, Temps Niveau 0: 000.070 msTemps Tous Niveaux: 000.006 ms
```

val inexistente

```
Niveau: 7, Temps Niveau 0: 000.000 msTemps Tous Niveaux: 000.000 ms
Niveau: 8, Temps Niveau 0: 000.000 msTemps Tous Niveaux: 000.000 ms
Niveau: 9, Temps Niveau 0: 000.002 msTemps Tous Niveaux: 000.000 ms
Niveau: 10, Temps Niveau 0: 000.003 msTemps Tous Niveaux: 000.001 ms
Niveau: 11, Temps Niveau 0: 000.008 msTemps Tous Niveaux: 000.000 ms
Niveau: 12, Temps Niveau 0: 000.015 msTemps Tous Niveaux: 000.002 ms
Niveau: 13, Temps Niveau 0: 000.031 msTemps Tous Niveaux: 000.004 ms
Niveau: 14, Temps Niveau 0: 000.066 msTemps Tous Niveaux: 000.007 ms
Niveau: 15, Temps Niveau 0: 000.151 msTemps Tous Niveaux: 000.013 ms
Niveau: 16, Temps Niveau 0: 000.311 msTemps Tous Niveaux: 000.026 ms
```

10 000 recherches:
val au hasard

```
Niveau: 7, Temps Niveau 0: 000.000 msTemps Tous Niveaux: 000.002 ms
Niveau: 8, Temps Niveau 0: 000.002 msTemps Tous Niveaux: 000.001 ms
Niveau: 9, Temps Niveau 0: 000.006 msTemps Tous Niveaux: 000.001 ms
Niveau: 10, Temps Niveau 0: 000.014 msTemps Tous Niveaux: 000.001 ms
Niveau: 11, Temps Niveau 0: 000.036 msTemps Tous Niveaux: 000.003 ms
Niveau: 12, Temps Niveau 0: 000.079 msTemps Tous Niveaux: 000.005 ms
Niveau: 13, Temps Niveau 0: 000.164 msTemps Tous Niveaux: 000.019 ms
Niveau: 14, Temps Niveau 0: 000.322 msTemps Tous Niveaux: 000.035 ms
Niveau: 15, Temps Niveau 0: 000.692 msTemps Tous Niveaux: 000.067 ms
Niveau: 16, Temps Niveau 0: 000.709 msTemps Tous Niveaux: 000.064 ms
```

val inexistente

```
Niveau: 7, Temps Niveau 0: 000.002 msTemps Tous Niveaux: 000.001 ms
Niveau: 8, Temps Niveau 0: 000.005 msTemps Tous Niveaux: 000.001 ms
Niveau: 9, Temps Niveau 0: 000.010 msTemps Tous Niveaux: 000.003 ms
Niveau: 10, Temps Niveau 0: 000.036 msTemps Tous Niveaux: 000.003 ms
Niveau: 11, Temps Niveau 0: 000.083 msTemps Tous Niveaux: 000.004 ms
Niveau: 12, Temps Niveau 0: 000.161 msTemps Tous Niveaux: 000.012 ms
Niveau: 13, Temps Niveau 0: 000.325 msTemps Tous Niveaux: 000.043 ms
Niveau: 14, Temps Niveau 0: 000.673 msTemps Tous Niveaux: 000.077 ms
Niveau: 15, Temps Niveau 0: 001.434 msTemps Tous Niveaux: 000.123 ms
Niveau: 16, Temps Niveau 0: 002.971 msTemps Tous Niveaux: 000.255 ms
```

100 000 recherches:
val au hasard

```
Niveau: 7, Temps Niveau 0: 000.015 msTemps Tous Niveaux: 000.006 ms
Niveau: 8, Temps Niveau 0: 000.028 msTemps Tous Niveaux: 000.008 ms
Niveau: 9, Temps Niveau 0: 000.060 msTemps Tous Niveaux: 000.011 ms
Niveau: 10, Temps Niveau 0: 000.132 msTemps Tous Niveaux: 000.015 ms
Niveau: 11, Temps Niveau 0: 000.367 msTemps Tous Niveaux: 000.029 ms
Niveau: 12, Temps Niveau 0: 000.788 msTemps Tous Niveaux: 000.064 ms
Niveau: 13, Temps Niveau 0: 001.620 msTemps Tous Niveaux: 000.156 ms
Niveau: 14, Temps Niveau 0: 003.378 msTemps Tous Niveaux: 000.361 ms
Niveau: 15, Temps Niveau 0: 006.862 msTemps Tous Niveaux: 000.676 ms
Niveau: 16, Temps Niveau 0: 006.926 msTemps Tous Niveaux: 000.633 ms
```

val inexistente

```
Niveau: 7, Temps Niveau 0: 000.028 msTemps Tous Niveaux: 000.003 ms
Niveau: 8, Temps Niveau 0: 000.053 msTemps Tous Niveaux: 000.007 ms
Niveau: 9, Temps Niveau 0: 000.105 msTemps Tous Niveaux: 000.016 ms
Niveau: 10, Temps Niveau 0: 000.345 msTemps Tous Niveaux: 000.029 ms
Niveau: 11, Temps Niveau 0: 000.766 msTemps Tous Niveaux: 000.048 ms
Niveau: 12, Temps Niveau 0: 001.526 msTemps Tous Niveaux: 000.157 ms
Niveau: 13, Temps Niveau 0: 003.182 msTemps Tous Niveaux: 000.387 ms
Niveau: 14, Temps Niveau 0: 006.515 msTemps Tous Niveaux: 000.686 ms
Niveau: 15, Temps Niveau 0: 014.274 msTemps Tous Niveaux: 001.320 ms
Niveau: 16, Temps Niveau 0: 031.024 msTemps Tous Niveaux: 002.478 ms
```

Partie 2 – Complexité de la recherche dans une liste à niveau

- ☑ ~~fonction de recherche 'classique' uniquement dans le niveau 0~~
- ☑ ~~fonction de recherche à partir du niveau le plus haut~~
- ☑ ~~Comparer le temps mis pour faire 1 000, 10 000, 100 000 recherches (rechercher une valeur au hasard) avec des valeurs de croissantes.~~
- ☐ Affichage graphique des valeurs
  (non fait problème due à Excel)

Partie 3 : Stockage de contacts dans une liste à niveaux

1. Rechercher un contact, et proposer une complétion automatique à partir de la 3ème lettre entrée
pour le nom (il faudra donc faire la saisie du nom de recherche caractère par caractère) ;
2. Afficher les rendez-vous d'un contact ;
3. Créer un contact (avec insertion dans la liste) ;
4. Créer un rendez-vous pour un contact (avec insertion dans la liste si le contact n'existe pas) ;
5. Supprimer un rendez-vous ;
6. Sauvegarder le fichier de tous les rendez-vous ;
7. Charger un fichier de rendez-vous ;

Nous allons commencer par créer un contact:

```
void addContact(const char *nom, const char *prenom) {
    if (contactCount < MAX_CONTACTS) {
        ContactList[contactCount].nom = strdup(nom);
        ContactList[contactCount].prenom = strdup(prenom);
        contactCount++;
    }
}
```

On crée un nouveau contact en allouant dynamiquement de la mémoire pour stocker les noms et prénoms du nouveau contact. Elle va ensuite augmenter le nombre de contacts .

Nous allons ensuite créer un rendez-vous pour un contact(avec insertion dans la liste si le contact n'existe pas) ;

```
void addRendezVousToContact(Contact *contact, RendezVous rv) {
    if (contact->nombreRendezVous < 10) { // 10 est le nombre de
rendez-vous par contact(on pourrais prendre n'inporte quel chiffre)
        contact->rendezVous[contact->nombreRendezVous++] = rv;
    }
}
```

On commence par créer un rendez-vous au contact
avec comme cellule:

```
typedef struct RendezVous {
    int jour;
    int mois;
    int annee;
    int heure;
    int minute;
    char objet[100];
} RendezVous;
```

```
void creerRendezVousPourContact(const char *nom, const char *prenom,
RendezVous rv) {
    for (int i = 0; i < contactCount; i++) {
        if (strcmp(ContactList[i].nom, nom) == 0 &&
strcmp(ContactList[i].prenom, prenom) == 0) {
            addRendezVousToContact(&ContactList[i], rv);
            return;
        }
    }

    // Si le contact n'existe pas, le créer et ajouter le rendez-vous
    creerEtInsererContact(nom, prenom);
    addRendezVousToContact(&ContactList[contactCount - 1], rv);
}
```

Si le contact est trouvé, un rendez-vous est ajouté. Si le contact n'existe pas, un nouveau contact est créé avec le rendez-vous spécifié.

On peut ensuite faire un programme qui recherche un contact, et propose une complétion automatique à partir de la 3ème lettre entrée

```
void autoCompleteContact(char *input) {
    int length = strlen(input);
    if (length < 3) {
        printf("Continuez à entrer des caractères...\n");
        return;
    }

    printf("Suggestions:\n");
    bool found = false;
    for (int i = 0; i < contactCount; i++) {
        if (strncmp(ContactList[i].nom, input, length) == 0) {
            printf("%s %s\n", ContactList[i].nom,
ContactList[i].prenom);
            found = true;
        }
    }

    if (!found) {
        printf("Aucun contact trouvé.\n");
    }
}
```

Si l'entrée est inférieure à 3 l'autocompletion ne sera pas activée , si l'entrée possède plus de 3 caractères on va parcourir la liste de contacts existants (ContactList). Pour chaque contact, on va comparer le début du nom du contact avec la chaîne input en utilisant strncmp. Si elle est trouvée nous allons afficher le nom et le prénom de ce contact .

Nous allons maintenant voir le code pour afficher le contact

```
void afficherRendezVousDuContact(const char *nom, const char *prenom) {
    for (int i = 0; i < contactCount; i++) {
        if (strcmp(ContactList[i].nom, nom) == 0 &&
strcmp(ContactList[i].prenom, prenom) == 0) {
            printf("Rendez-vous pour %s %s:\n", nom, prenom);
            for (int j = 0; j < ContactList[i].nombreRendezVous; j++) {
                RendezVous rv = ContactList[i].rendezVous[j];
                printf("%02d/%02d/%04d à %02d:%02d - %s\n", rv.jour,
rv.mois, rv.annee, rv.heure, rv.minute, rv.objet);
            }
            return;
        }
    }
    printf("Contact non trouvé.\n");
}
```

On commence par parcourir la liste des contacts.Pour chaque contact dans la liste, la fonction compare le nom et le prénom du contact avec les valeurs fournies en arguments (nom et prénom). .
Une fois le contact trouvé, on parcourt ensuite tous les rendez-vous enregistrés pour ce contact et on printf les détails de chaque rendez-vous, y compris la date, l'heure, et l'objet du rendez-vous.

Nous allons maintenant voir pour la suppression d'un rendez-vous

```
void supprimerRendezVous(Contact *contact, int jour, int mois, int
annee, int heure, int minute) {
    for (int i = 0; i < contact->nombreRendezVous; i++) {
        RendezVous rv = contact->rendezVous[i];
        if (rv.jour == jour && rv.mois == mois && rv.annee == annee &&
rv.heure == heure && rv.minute == minute) {
            // Décalage des rendez-vous suivants
            for (int j = i; j < contact->nombreRendezVous - 1; j++) {
                contact->rendezVous[j] = contact->rendezVous[j + 1];
            }
            contact->nombreRendezVous--;
            return;
        }
```

```
    }
    printf("Rendez-vous non trouvé.\n");
}
```

La fonction parcourt tous les rendez-vous du contact spécifié. Pour chaque rendez-vous, elle compare les détails du rendez-vous (jour, mois, année, heure, minute) avec ceux fournis en paramètres.

Lorsqu'un rendez-vous correspondant aux détails fournis est trouvé, la fonction procède à sa suppression. Elle le fait en décalant tous les rendez-vous qui suivent dans la liste d'une position vers le début, écrasant ainsi le rendez-vous à supprimer .

Après la suppression, elle décrémente le compteur de rendez-vous (nombreRendezVous) du contact, reflétant ainsi la suppression d'un rendez-vous dans son agenda.

Enfin nous allons voir pour l'ajout des rdv dans un fichier et l'appel de ces rdv à partir d'un fichier:

Ajout des Rendez-vous:

```
void sauvegarderRendezVous(const char *nomFichier) {
    FILE *fichier = fopen("rdv.txt", "w");
    if (fichier == NULL) {
        perror("Erreur lors de l'ouverture du fichier");
        return;
    }

    for (int i = 0; i < contactCount; i++) {
        Contact contact = ContactList[i];
        for (int j = 0; j < contact.nombreRendezVous; j++) {
            RendezVous rv = contact.rendezVous[j];
            fprintf(fichier, "%s %s - %02d/%02d/%04d %02d:%02d - %s\n",
                    contact.nom, contact.prenom,
                    rv.jour, rv.mois, rv.annee, rv.heure, rv.minute,
rv.objet);
        }
    }

    fclose(fichier);
}
```

Nous allons d'abord créer le fichier rdv.txt , ensuite en fonction du nombre de contact nous allons parcourir tout les contact et en fonction du nombre de rdv de chaque contact nous allons parcourir tous les rendez-vous de chaque contact que nous allons implémenter dans le fichier avec les caractéristiques du rdv (jour mois année heure minute objet)mais aussi par le nom et le prénom de la personne avec le rdv concernée pour pouvoir le retrouver .

Appel des rendez-vous à partir du fichier:

```
void chargerRendezVous(const char *nomFichier) {
    FILE *fichier = fopen("rdv.txt", "r");
    if (fichier == NULL) {
        perror("Erreur lors de l'ouverture du fichier");
        return;
    }

    char nom[50], prenom[50], objet[100];
    int jour, mois, annee, heure, minute;
    while (fscanf(fichier, "%s %s - %d/%d/%d %d:%d - %[^\n]",
                  nom, prenom, &jour, &mois, &annee, &heure, &minute,
objet) == 8) {
        RendezVous rv = {jour, mois, annee, heure, minute};
        strcpy(rv.objet, objet);
        creerRendezVousPourContact(nom, prenom, rv);
    }

    fclose(fichier);
}
```

Nous allons faire l'inverse de la fonction précédente , nous allons appeler le fichier contenant les rendez-vous. Nous allons ensuite lire ligne par ligne avec fscanf.Pour chaque ligne lue, nous allons créer une structure dont les champs seront remplis avec les données extraites. Puis, on fait appelle creerRendezVousPourContact pour l'implementer avec le nom et prénom du contact ainsi que le rendez-vous extrait.

Partie 3 : Stockage de contacts dans une liste à niveaux

- ☑ ~~1. Rechercher un contact, et proposer une complétion automatique à partir de la 3ème lettre entrée pour le nom (il faudra donc faire la saisie du nom de recherche caractère par caractère) ;~~
- ☑ ~~2. Afficher les rendez-vous d'un contact ;~~
- ☑ ~~3. Créer un contact (avec insertion dans la liste) ;~~
- ☑ ~~4. Créer un rendez-vous pour un contact (avec insertion dans la liste si le contact n'existe pas) ;~~
- ☑ ~~5. Supprimer un rendez-vous ;~~
- ☑ ~~6. Sauvegarder le fichier de tous les rendez-vous ;~~
- ☑ ~~7. Charger un fichier de rendez-vous~~

- ☐ 8. Fournir les temps de calcul pour une insertion de nouveau contact : voir point 2) ci –dessous.

2) Tracer la complexité de l'application :

Comme pour la liste à niveau comportant des entiers, vous devez produire un graphique de comparaison de temps d'exécution entre :

- ☐ i) Les opérations de recherche / insertion faites uniquement au niveau 0
- ☐ ii) Ces mêmes opérations faites à partir du niveau le plus élevé.

Conclusion :

En conclusion, ce projet a permis de mettre en évidence l'efficacité et la flexibilité des structures de données complexes telles que les listes à niveaux dans la gestion des agendas et des contacts. L'implémentation des différentes fonctionnalités, de la création de cellules à l'affichage des niveaux, ainsi que la gestion des contacts et des rendez-vous, a démontré une utilisation pratique de ces concepts algorithmiques. De plus, l'analyse de la complexité des opérations de recherche a révélé l'importance de choisir la structure de données adaptée selon le contexte d'utilisation.