Le but de ce projet est de réaliser une application qui permet de gérer un agenda en utilisant une structure de données intermédiaire entre les listes chaînées et les arbres.

Cette structure de données permettra, dans le même temps, de :

1. Conserver un ordre entre ses éléments :

Ainsi, on pourra comparer les éléments deux à deux, et déterminer lequel des deux doit être rangé « avant l'autre » dans cette structure de données

2. Effectuer les opérations standard (insertion, recherche, suppression) avec une complexité qui reste faible, notamment plus faible qu'avec une liste simplement chaînée.

L'idée est que cette structure permette d'implémenter des traitements utilisant la dichotomie, pour tirer profit du fait que les valeurs sont stockées suivant un critère de tri.

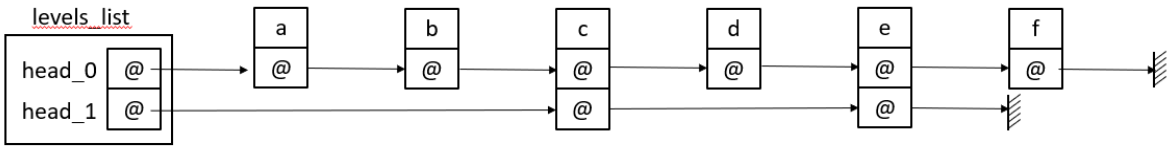
Etape 1 : illustration avec des entiers, liste à deux niveaux

On souhaite avoir une structure de données, stockant des entiers, et triée par ordre décroissant. Cette structure est une liste chaînée avec deux niveaux :

Un niveau où chaque cellule pointe vers la suivante, comme dans une liste chaînée classique, et un second niveau où la cellule pointe **ou non** vers une autre plus loin dans la liste.

Chaque cellule aura donc un, ou deux, pointeurs vers une suivante, en fonction des chaînages.

Illustration : pour simplifier l'interprétation, on représentera une cellule de manière verticale : en haut, la valeur stockée, puis en dessous les différents niveaux de pointage, le premier étant celui d'une liste simplement chaînée.



La variable **`levels_list`** stocke deux pointeurs, un sur chaque niveau de liste. Certaines cellules n'ont qu'une suivante, d'autre deux, la dernière de la liste de niveau 0 n'en a aucune.

Ainsi, on définit une cellule de la manière suivante : elle stocke une valeur (int), et a deux pointeurs next_0 et next_1.

```
typedef struct s_d_cell
{
     int value;
     struct s_d_cell *next_0;
     struct s_d_cell *next_1;
} t_d_cell;
```

Et la liste correspondante :

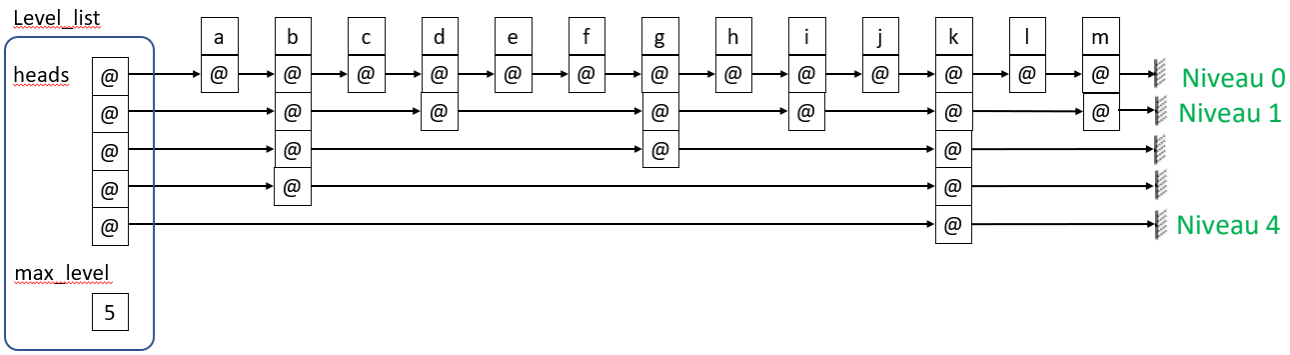
```
typedef struct s_d_list
{
     t_d_cell *head_0;
     t_d_cell *head_1;
} t_d_list;
```

Liste à plusieurs niveaux

Listes et cellules à plusieurs niveaux : on souhaite généraliser ce principe pour plusieurs niveaux. On ne dispose que d'une seule information à ce propos : le nombre maximum de niveau, que l'on appellera **hauteur** de la liste.

Par exemple, si la hauteur de la liste est égale à 5, chaque cellule pourra pointer de 0 (uniquement pour la dernière) à 5 cellules. Les différents niveaux seront numérotés à partir de 0, ce niveau 0 étant celui où toutes les cellules sont chaînées (comme pour une liste chaînée classique)

Illustration :



Et les cellules par niveau :

Niveau 0 : **a b c d e f g h i j k l m**

Niveau 1 : **b d g i k m**

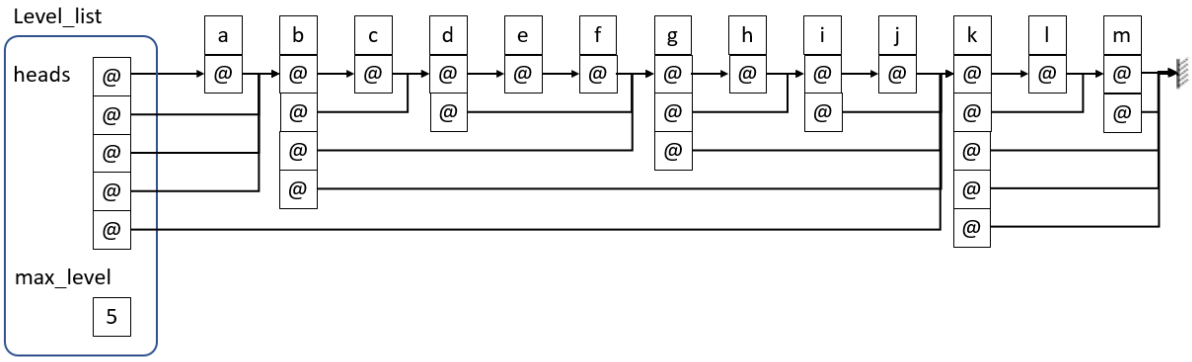
Niveau 2 : **b g k**

Niveau 3 : **b k**

Niveau 4 : **k**

Illustration équivalente : en réalité, chaque pointeur, quel que soit son niveau, pointe sur une cellule, et non sur un autre pointeur de même niveau :

Ainsi, une illustration plus fidèle de la structure de données est la suivante :



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'

Ainsi, on peut définir, pour chaque cellule, de manière analogue à la liste, le nombre '`level`' de suivantes.

Ainsi, **une liste est représentée comme un ensemble de pointeurs** '`heads`' pointant sur chaque niveau : **on utilisera un tableau dynamique** contenant `max_level` pointeurs

De même, **chaque cellule contiendra un ensemble de pointeurs** '`next`' pointant sur le ou les niveaux concernés par la cellule : **on utilisera un tableau dynamique** de `level` pointeurs.

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

Cahier des charges :

Vous créerez des modules (ensemble fichier.c / fichier.h) pour :

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

     Pour cette dernière fonctionnalité, il est possible de faire une 'simple' insertion à chaque niveau en partant du début de la liste, mais il est possible d'être plus efficace, à vous de trouver comment.

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

Exemples de résultats attendus, pour une liste à 5 niveaux

Liste à niveau vide

```
[list head_0 @-]--> NULL
[list head_1 @-]--> NULL
[list head_2 @-]--> NULL
[list head_3 @-]--> NULL
[list head_4 @-]--> NULL
```

Après insertion de cellules à niveau, affichage simple

```
[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
```

Après insertion de cellules à niveau, affichage aligné

```
[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 2 – Complexité de la recherche dans une liste à niveau

L'utilisation de liste à niveaux permet d'accélérer les recherches pour obtenir une complexité qui s'approche de celle de la recherche par dichotomie. Pour l'illustrer, nous allons construire une grande liste à niveau, et comparer :

Le temps mis pour des recherches en utilisant uniquement le niveau 0 (donc comme une liste chaînée simple)

Le temps mis pour des recherches en utilisant tous les niveaux.

Le principe de recherche est le suivant : on commence la recherche au niveau le plus haut. Si on ne trouve pas la valeur à ce niveau, on continue la recherche au niveau précédent en repartant de la cellule à laquelle on était arrivé.

Pour cela, nous allons construire des listes à niveau avec le principe suivant.

Soit un entier n :

La liste stockera $2^n - 1$ cellules, avec les valeurs de 1 à $2^n - 1$

Chaque niveau pointera une cellule sur deux du niveau précédent. Il y aura donc n niveaux

Illustration pour n=3

On stockera $2^3 - 1 = 7$ cellules, avec les valeurs de 1 à 7, cette liste aura $n = 3$ niveaux

La liste voulue est la suivante

```
[list head_0 @-]-->[  1|@-]-->[  2|@-]-->[  3|@-]-->[  4|@-]-->[  5|@-]-->[  6|@-]-->[  7|@-]-->NULL
[list head_1 @-]------------->[  2|@-]------------->[  4|@-]------------->[  6|@-]------------->NULL
[list head_2 @-]----------------------------------->[  4|@-]----------------------------------->NULL
```

Exemples

Recherche de 3

Au niveau 2 : on ne le trouve pas, on descend au niveau 1 à partir du début (car si 3 est dans la liste, il est avant 4)

Au niveau 1 : on ne le trouve pas, on descend au niveau 0 à partir de la cellule stockant 2 (car si 3 est dans la liste, il est après 2)

Au niveau 0 : on trouve 3

Recherche de 7

Au niveau 2, on ne le trouve pas, on descend au niveau 1 à partir de la cellule stockant 4 (car si 7 est dans la liste, il est après 4)

Au niveau 1, on ne le trouve pas, on descend au niveau 0 à partir de la cellule stockant 6 (car si 7 est dans la liste, il est après 6)

Au niveau 0, on le trouve.

Cahier des charges

À partir d'une valeur de n, créer la liste à niveau correspondante. Une solution, pour calculer les niveaux des cellules, est de créer un tableau **`levels`** de $2^n - 1$ entiers, de l'initialiser à 0, puis ajouter 1 dans une case sur 2, puis ajouter 1 dans une case sur 4, etc…

Exemple

pour $n = 3$: un tableau **`levels`** de 7 entiers, initialisé avec 7 fois la valeur 0

0	0	0	0	0	0	0

On ajoute 1 dans une case sur 2

0	1	0	1	0	1	0

On ajoute 1 dans une case sur 4

0	1	0	2	0	1	0

On peut maintenant ajouter les cellules dans la liste avec une boucle : pour un indice **`i`**, on ajoute la cellule de valeur **`i+1`** au niveau **`levels[i]`**

Ensuite, écrire deux fonctions de recherche :

- Une fonction de recherche 'classique' uniquement dans le niveau 0
- Une 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 n croissantes.

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

Vous produirez des graphiques de comparaison de complexité de ces deux types de recherche