Rapport

Gestionnaire d’agenda

Algorithme et Structure de donnée 2

Corentin KERVAGORET, Nathan PEGE, Bastien PELTIER
Semestre 3 – 2023 – EFREI Paris

Table des matières

Introduction

Le projet permet d'approfondir les notions des liste chaînées vu en classe afin de réaliser un gestionnaire d'agenda. De projet permet d'introduire la notion de liste à niveau. Une liste à niveau est une liste simplement chainée ou chaque cellule aura un nombre x déterminé de niveaux et sera présent sur x niveaux. Cette liste permettra la recherche plus rapide dans une liste comme nous le verrons dans la partie 2. On utilisera ainsi le modèle de cette liste à niveau dans la partie 3 pour trier les noms des contacts dans une liste et faire une recherche des contacts plus rapide.

Lien du github : https://github.com/Corentin-k/Gestionnaire_Agenda

TOTAL	25	(dont 20 point hors bonus et options)
Parties 1 et 2	13	
les applications sont compilables	1	✓
insertion en tête	1	✓
affichage d'une liste à niveau	1	✓
insertion dans l'ordre de la liste	2	✓
creation de la liste por test partie 2	2	✓
recherche au niveau 0	1	✓
Recherche multi niveaux	2	✓
tests et timing	2	✓
graphique de performance expliqué	1	✓
Partie 3	7	
menu des choix	1	✓
créer un rendez-vous pour un contact	2	✓
afficher les rendez-vous d'un contact	2	✓
tests et timing pour la complexité	2	
Bonus & options	5	
affichage aligné (partie 1)	1	✓
complétion automatique pour le nom	2	
gestion des fichiers pour les rendez-vous	2	

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

Dans cette première partie nous avions comme objectif de créer une liste à niveau. Pour cela nous avons créé deux structures :

```c
typedef struct s_d_cell
{
    int value;
    int level;
    struct s_d_cell **next;
} t_d_cell;


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

- Une structure qui définit un nouveau type t_d_cell qui permettra de gérer une liste de cellule : son nom et son niveau et la cellule suivante. Cette structure permet de créer une liste simplement chaînée avec chaque cellule qui a un pointeur vers sa cellule voisine

- Et une structure t_d_list qui permettra de gérer une liste à niveau. Cette liste peut être représenter sous la forme d'un tableau qui contiendra n cases et chaque case contiendra le pointeur de la première cellule (t_d_cell) de cellule grâce

Pour gérer nous avons dû réaliser 3 fonctions pour gérer les cellules et 5 fonctions pour la liste :

```c
////////////////CELLULE////////////////////

//Créer une cellule : on donne sa valeur et le nomb
t_d_cell *createCell(int value, int num_levels);

//Affiche une cellule
void displayCell(t_d_cell *cell);


//Ajoute une cellule dans une liste en la triant pa
void addCellInList(t_d_list *list, t_d_cell *cell);
```

- Une fonction qui permet d'initialiser une cellule avec une valeur et un nombre qui déterminera sur combien de niveau elle devra être dans la liste.
- Une fonction qui permet d'afficher une cellule

- Une fonction qui ajoute une cellule dans la liste principale et qui permet de garder la liste triée de façon croissante (1)

```c
////////////////LISTE////////////////////
//Créer une liste : on donne le nombre de niveaux que possède cette li
t_d_list *createEmptyList(int max_levels);

//Créer une liste : de 2^n cellules
t_d_list createList(int n);

//Affiche une liste
void displayList(t_d_list *list);

//Ajoute une cellule dans une liste en la triant par ordre croissant
t_d_list* createListTrie(int n);

//Ajoute une cellule dans une liste en la triant par ordre croissant
void ajoutreCursifList(t_d_list* list, int val, int niv,int puiss);
```

- Une fonction qui permet d'initialiser une liste à niveau vide.

- Une fonction itérative qui permet de créer une liste de 2^n cellules pour la deuxième partie

- Une fonction qui permet d'afficher une liste de façon que chaque cellule soit affichée en colonne en fonction de son niveau (2)

- Et une autre fonction pour la création d'une liste 2^n mais de façon récursive (3)

(1)

```
Ajout de la valeur -59
[list head_0 @-]--->[ -79|@-]--->[ -59|@-]--->NULL
[list head_1 @-]--->[ -79|@-]--->[ -59|@-]--->NULL
[list head_2 @-]--->[ -79|@-]--->[ -59|@-]--->NULL
[list head_3 @-]--->[ -79|@-]---------------->NULL
[list head_4 @-]--->[ -79|@-]---------------->NULL
Ajout de la valeur 76
[list head_0 @-]--->[ -79|@-]--->[ -59|@-]--->[ 76|@-]--->NULL
[list head_1 @-]--->[ -79|@-]--->[ -59|@-]--->[ 76|@-]--->NULL
[list head_2 @-]--->[ -79|@-]--->[ -59|@-]---------------->NULL
[list head_3 @-]--->[ -79|@-]---------------------------->NULL
[list head_4 @-]--->[ -79|@-]---------------------------->NULL
Ajout de la valeur 3
[list head_0 @-]--->[ -79|@-]--->[ -59|@-]--->[ 3|@-]--->[ 76|@-]--->NULL
[list head_1 @-]--->[ -79|@-]--->[ -59|@-]--->[ 3|@-]--->[ 76|@-]--->NULL
[list head_2 @-]--->[ -79|@-]--->[ -59|@-]--->[ 3|@-]---------------->NULL
[list head_3 @-]--->[ -79|@-]---------------->[ 3|@-]---------------->NULL
[list head_4 @-]--->[ -79|@-]---------------------------------------->NULL
```

(2)

```
[list head_0 @-]--->[ -99831|@-]--->[ -79|@-]--->[ -59|@-]--->[ -54|@-]--->[ -26|@-]--->[ 3|@-]--->[ 76|@-]--->NULL
[list head_1 @-]--->[ -99831|@-]--->[ -79|@-]--->[ -59|@-]--->[ -54|@-]--->[ -26|@-]--->[ 3|@-]--->[ 76|@-]--->NULL
[list head_2 @-]--->[ -99831|@-]--->[ -79|@-]--->[ -59|@-]--->[ -54|@-]--->[ -26|@-]--->[ 3|@-]---------------->NULL
[list head_3 @-]--->[ -99831|@-]--->[ -79|@-]---------------->[ -54|@-]--->[ -26|@-]--->[ 3|@-]---------------->NULL
[list head_4 @-]-------------------->[ -79|@-]---------------->[ -54|@-]--->[ -26|@-]------------------------->NULL
```

(3)

```
///<summary>
///Ajoute une cellule dans une liste en la triant par ordre croissant
///</summary>
///<param name="n">niveau de la liste</param>
///<returns>void</returns>
t_d_list*  createListTrie(int n){
    t_d_list* list = createEmptyList( max_levels: n);
    ajoutreCursifList(list,  val: pow( X: 2, Y: n)/2, niv: n, puiss: n-1);
    return list;
}

void ajoutreCursifList(t_d_list* list, int val, int niv,int puiss)
{// le principe est d'ajouter du niveau le plus profond et de remonter comme un arbre avec ses voisins
    if (niv==0) return; // condition d'arrêt
    addCellInList(list, cell: createCell( value: val, nb_levels: niv));

    ajoutreCursifList(list, val: val+ pow( X: 2, Y: puiss-1), niv: niv-1, puiss: puiss-1);
    ajoutreCursifList(list, val: val- pow( X: 2, Y: puiss-1), niv: niv-1, puiss: puiss-1);
}
```

Cette fonction permet la création d'une liste à N niveau pour 2**n – 1 valeurs. Après avoir examiné la solution proposée dans le sujet, nous aurions donc un tableau de N valeurs, pour déterminer les niveaux et le nombre d'opérations qui va avec, un algorithme donc très couteux. Après réflexion, nous avons réalisé que le principe était similaire à celui d'un arbre binaire : les fils à gauche et à droite ont leurs niveaux max qui est égal à celui de leur père - 1 (exemple de 0 à 7, 4 possède 2 " fils", 2 et 6, qui sont au niveau –1 par rapport à lui, c'est à dire 1 – 4 étant au niveau max, 2).

Ainsi, nous pouvons parcourir toutes les valeurs de 0 à 2**n – 1 en passant par les moitiés : on part de la moitié, on l'ajoute au niveau max, et on passe à ses fils : lui-même + la moitié et lui-même moins la moitié, les 2 au niveau en dessous. La condition d'arrêt est simple : on ajoute jusqu'à atteindre le dernier niveau, c'est à dire 0 : ainsi, toutes les valeurs sont bien rajoutées de manière trié aux niveaux correspondants. Une manière beaucoup moins complexe qui nous permet d'aller jusqu'à N = 27 lors des tests !

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

Utilisation de la fonction timer.h et création d'un nouveau fichier research.h/.C réserver au temps de la recherche d'une taille de liste très grande avec une recherche classique dans un liste chainée à partir du niveau 0 de la liste à niveaux et une recherche plus adaptée en regardant sur chaque niveau si la valeur est présente et remonter jusqu'au niveau 0

Temps de recherche pour n valeurs en cherchant des valeurs de -n à +n

Pour n=1000

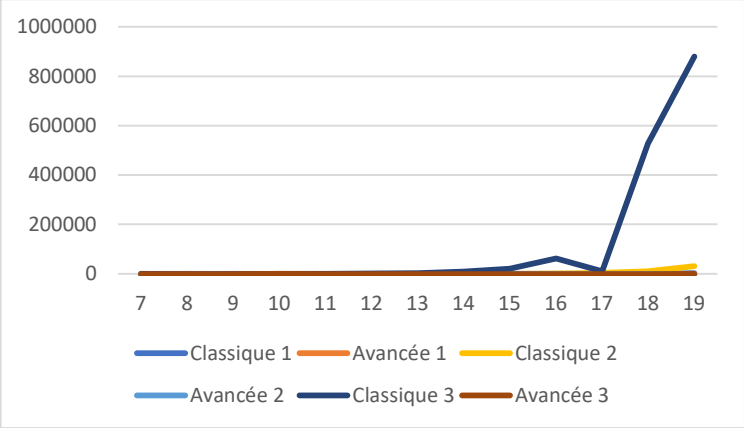
```
7   [1] 000,001 [0] 000,000
8   [0] 000,000 [0] 000,000
9   [2] 000,002 [1] 000,001
10  [4] 000,004 [0] 000,000
11  [8] 000,008 [0] 000,000
12  [13] 000,013    [0] 000,000
13  [29] 000,029    [0] 000,000
14  [75] 000,075    [0] 000,000
15  [160] 000,160   [0] 000,000
16  [229] 000,229   [0] 000,000
17  [463] 000,463   [1] 000,001
18  [1022] 001,022  [0] 000,000
19  [2860] 002,860  [0] 000,000
20  [10593] 010,593 [0] 000,000
21  [27634] 027,634 [0] 000,000
22  [56500] 056,500 [2] 000,002
23  [103539] 103,539    [0] 000,000
24  [121435] 121,435    [0] 000,000
25  [203427] 203,427    [0] 000,000
```

Pour n=10000

```
7   [3] 000,003 [1] 000,001
8   [6] 000,006 [1] 000,001
9   [13] 000,013    [1] 000,001
10  [54] 000,054    [0] 000,000
11  [101] 000,101   [1] 000,001
12  [161] 000,161   [1] 000,001
13  [294] 000,294   [1] 000,001
14  [500] 000,500   [1] 000,001
15  [1073] 001,073  [0] 000,000
16  [2110] 002,110  [2] 000,002
17  [4217] 004,217  [1] 000,001
18  [10877] 010,877 [0] 000,000
19  [31866] 031,866 [1] 000,001
```

Pour n=100000 :

```
7   [35] 000,035    [3] 000,003
8   [75] 000,075    [6] 000,006
9   [140] 000,140   [2] 000,002
10  [452] 000,452   [3] 000,003
11  [915] 000,915   [4] 000,004
12  [1901] 001,901  [3] 000,003
13  [3590] 003,590  [4] 000,004
14  [9632] 009,632  [8] 000,008
15  [21697] 021,697 [10] 000,010
16  [63011] 063,011 [3] 000,003
17  [102350] 102,350    [7] 000,007
18  [528274] 528,274    [8] 000,008
19  [879550] 879,550    [27] 000,027
```



On remarque donc que plus on a de valeurs dans la liste plus la recherche dans la liste à niveau est efficace. En effet dans la liste à niveau on effectue une recherche qui se rapproche à une recherche dichotomique. On parcourt le plus bas niveau où chaque valeur est écartée de n valeurs on trouve un encadrement de la position. On peut donc directement aller au bon endroit au niveau n-1 ou chaque valeur et espacé de n-x valeurs et on recommence. On a donc plus une complexité de O(n) mais de O(log(n+k))) avec k le nombre de niveaux ici 4.

Partie 3 – Stockage de contacts dans une liste à niveaux

Dans cette partie du projet, nous nous intéressons maintenant dans la conception d'un agenda dont le programme utilisera les notions vues dans les deux parties précédentes, c'est-à-dire l'emploi des listes à niveaux.

Notre agenda se compose en 2 grandes parties qui sont d'ailleurs liés entre elles. D'un côté on va s'occuper des contacts, et d'un autre des rendez-vous. On note cependant une grande différence, le stockage des contacts devra se faire à l'aide d'une liste à niveaux contrairement aux rendez-vous qui eux seront stockés dans des listes simplement chaînées.

Voyons maintenant en revue notre travail réalisé sur la partie 3 :

LES RENDEZ VOUS

1. Stockage des rendez-vous:

Commençons d'abord par la structure choisie pour gérer les rdvs :

```
typedef struct  rendez_vous{
    struct Date date;
    struct Heure heure_rendez_vous;
    struct Heure duree;
    struct rendez_vous *next;
    char* objet;

}*Rendez_vous;
```

Avec les structures Date et Heure suivante :

```
struct  Date{
    int jour;
    int mois;
    int annee;
};
```

```
struct Heure{
    int heure;
    int minute;
};
```

On sait que les rdv sont stockés dans des listes simples alors il est nécessaire que chaque rdv stocke le rdv suivant dans sa structure

2. Créer des rendez-vous :

Pour cette fonctionnalité, on a décidé de créer plusieurs fonctions qui vont nous aider à accomplir la tâche. Pour préciser, les rendez-vous sont stockés dans la structure des contacts que nous verrons juste après. Nous avons décidé aussi de les trier par ordre croissant de date d'apparition pour permettre un affichage plus cohérent et intuitif. On va retrouver 3 fonctions principales pour cette fonctionnalité :

```
// Fonction pour créer un nouveau rendez-vous pour un contact
Rendez_vous createRendezVous();
```

Cette fonction permet de recueillir les informés saisis par l'utilisateur dans la console et de retourner un rdv qui sera ensuite trier

```
// Fonction pour mettre au bon endroit le rdv
void addNewRendezVous(Contact *personne);
```

Celle-ci permet de stocker sous forme de liste simplement chaîné les rendez-vous dans la structure d'un contact particulier.

Pour que cette fonction puisse être plus efficace,

```
// Fonction pour comparer 2 rendez-vous pour savoir qui est le plus tôt
int compareRendezVous(Rendez_vous rdv1, Rendez_vous rdv2);
```

La fonction ci-dessus nous permet de trier les rendez-vous entre eux pour mieux les ranger. Elle renvoi 1 quand le rdv à placer est plus tard que le rdv testé et 0 quand le rdv est à la bonne place.

```
void addNewRendezVous(Contact *personne){
    Rendez_vous newRV = createRendezVous();

    Rendez_vous temp = personne->rendez_vous;

    // S'il n'y a pas déjà de rdv, celui créer sera direct en tête
    if (temp == NULL){
        printf( format: "test2");
        personne->rendez_vous= newRV;
        return;}

    // Si le premier rdv est plus grand alors il prend la première place
    if(compareRendezVous( rdv1: personne->rendez_vous, rdv2: newRV) == 0){
        newRV->next = personne->rendez_vous;
        personne->rendez_vous = newRV;
        return;
    }

    // on parcourt la liste pour savoir où placer le rdv
    while (temp != NULL && compareRendezVous( rdv1: temp->next, rdv2: newRV) != 0)
        temp = temp->next;

    newRV->next = temp->next;
    temp->next = newRV;

    return;


}
```

Le schéma est simple, on crée d'abord un rendez-vous avec la fonction createRendezVous() et on le stock dans une variable de type Rendez-vous qui est un pointeur. On prend en compte les différents cas possibles de l'insertion de notre rdv dans la liste chaîné grâce à l'utilisation de la fonction compareRendezVous() et on insère ensuite notre rdv.

Une particularité, lors de la création du rdv, l'utilisateur est invité à remplir de nombreux champs concernant les informations de ce rdv. Ces informations sont recueillies grâce aux fonctions :

```
char *scanString() {
    char texte[201];

    if (scanf( format: "%s", texte) ==1 ) {
        // Lire au plus 199 caractères pour éviter le dépassement de tampon
        char *copie = (char *)malloc( Size: strlen( Str: texte) + 1);

            strcpy( Dest: copie,  Source: texte);
            return copie;

    }

}
```

```
int scanInt(int max) {
    int num;
    while (scanf( format: "%d", &num) != 1 || num <0 || num >= max) {
        // Si la saisie n'est pas un entier, vider le tampon d'entrée
        while (getchar() != '\n');
        printf( format: "\nVeuillez saisir un entier valide :\n>>>");
    }
    return num;
}
```

Ces fonctions font appel à d'autre fonctions et permettent de "scanner" la réponse de l'utilisateur dans la console lorsque celui sera amené à répondre

3. Afficher des rendez-vous :

Cette fonctionnalité est assurée dans notre programme par une seule fonction :

```c
void displayRendezVous(Contact *personne)
{
    Rendez_vous temp = personne->rendez_vous;
    int conteur = 0;
    if(temp == NULL){
        printf( format: "Ce contact ne possede pas encore de rendez-vous ! \n");
        return;
    }
    while(temp != NULL){
        conteur ++;
        printf( format: "Rendez-vous n%d\n",conteur);
        printf( format: "------ Objet: %s ------",temp->objet);
        printf( format: "\n| Date : %d/%d/%d",temp->date.jour,temp->date.mois,temp->date.annee);
        printf( format: "\n| Heure: %d:%d",temp->heure_rendez_vous.minute,temp->heure_rendez_vous.heure);
        printf( format: "\n| Duree: %d:%d",temp->duree.heure,temp->duree.minute);
        printf( format: "\n--------------------------\n");
        temp= temp->next;
    }
    return;
}
```

```
Rendez-vous cree !
Rendez-vous n1
------ Objet: 1 ------
| Date : 1/1/1
| Heure: 1:1
| Duree: 1:1
--------------------------
Rendez-vous n2
------ Objet: 1 ------
| Date : 1/1/1
| Heure: 1:1
| Duree: 1:1
--------------------------
```

Dans notre agenda, vous ne pourrez pas voir tous les rendez-vous de tout le monde d'un coup mais seulement tous les rendez-vous d'un contact en particulier. Sur la gauche vous avec un aperçu de ce que donne cette fonction.

4. Supprimer les des rendez-vous:

Cette fonctionnalité est aussi assurée dans notre programme par une seule fonction :

```c
void deleteRendezVous(Contact *personne,int indexe){
    Rendez_vous temp = personne->rendez_vous;
    Rendez_vous prevtemp = personne->rendez_vous;
    if (indexe==0){
        personne->rendez_vous = temp->next;
        free( Memory: temp->objet);
        free( Memory: temp);
    }
    for(int i = 0; i < indexe ; i++){
        if(temp->next == NULL && i < indexe){
            printf( format: "\n--------------------------\n");
            printf( format: "Vous avez saisie un Rendez-vous qui n'existe pas \n");
            printf( format: "--------------------------\n\n");

            return;
        }
        temp = temp->next;
        if(i+1 == indexe){
            prevtemp->next = temp->next;
            free( Memory: temp->objet);
            free( Memory: temp);
        }
        prevtemp = prevtemp->next;
    }
    return;
}
```

C'est une fonction non originale, elle reprend simplement la structure générale d'une suppression de cellule d'une liste chaîné simple. Elle envisage les cas possibles et agit en conséquence sur la liste de rendez-vous stocker dans un contact. L'utilisateur doit simplement rentrer le numéro du rdv qu'il souhaite supprimer et le programme s'en charge.

LES CONTACTS

1. Structure des contacts :

Voici comment nous avons structuré nos contacts :

```
typedef struct  s_contact{
    char * nom;
    char * prenom;
    Rendez_vous rendez_vous;
    struct s_contact **next;
}Contact;
```

On stocke ici 2 chaînes de caractères qui correspondent au nom et au prénom du contact qui seront affectés lors de la demande de création d'un contact grâce à la fonction scanString() vue précédemment. Elle stocke aussi donc une liste de rdv qui se modélise par un pointeur qui pointe sur la première cellule de la liste de rdv.

Cependant, le stockage des contacts possède une particularité imposée, la liste qui stocke la totalité des contacts doit être une liste chaînée à niveaux. C'est pour ça que chaque contact possède en fait non pas un pointeur qui pointe vers le prochain contact mais un tableau de pointeur qui pointent sur différents niveaux de contact.

```
typedef struct  list_contact{
    Contact **contact; // Tableau de pointeurs vers les têtes de chaque niveau
    int max_contact;
}List_contact;
```

On a ci-dessus la liste à niveaux des contacts avec le tableau de pointeurs HEAD qui stocke les premiers contacts des différents niveaux de la liste.

2. Création des contacts :

```
Contact *createContact(){
    Contact *newContact = malloc( Size: sizeof(Contact));
    printf( format: "Saisir le nom du contact :\n>>>");
    newContact->nom = scanString();
    conversionminuscule( str: newContact->nom);

    printf( format: "Saisir le prenom du contact :\n>>>");
    newContact->prenom = scanString();conversionminuscule( str: newContact->prenom);
    newContact->rendez_vous = NULL;
    newContact->next = malloc( Size: 4*sizeof(*newContact));
    for (int x=0;x<4;x++) newContact->next[0]=NULL;
    return newContact;

}
```

Cette fonction permet la création d'un contact, on réserve la mémoire correspondante, et on récupère le nom et le prénom du contact, après les avoir convertis, et on n'oublie pas d'initialiser les next pour chaque niveau : de 1 à 3 (après avoir réservé également la mémoire pour ces 4 pointeurs).

3. Recherche d'un contact

On a également créé une fonction qui permet de chercher un contact dans la liste et qui le renvoie. Cette fonction sera utilisée dans le menu pour afficher un contact, supprimer un rendez-vous d'un contact, etc.

```
Contact* askContact(List_contact* listContact){
    Contact *contact = createEmptyContact();
    printf( format: "Veuillez entrer le nom du contact :\n>>>");
    contact->nom = scanString();
    printf( format: "\nVeuillez entrer le prenom du contact :\n>>>");
    contact->prenom = scanString();
    if(contactExists(listContact,contact)==NULL){
        printf( format: "Le contact n'existe pas\n");
        free( Memory: contact->nom);
        free( Memory: contact->prenom);
        free( Memory: contact);
        return NULL;
    }
    return contact;
}
```

Cette fonction créée donc un nouveau contact sans que l'utilisateur le sache et cherchera grâce à la fonction contactExist qui prend en paramètre une variable de type Contact et s'il n'existe pas il retourne un pointeur de type NULL.

4. LA RECUPERATION DES CONTACTS ET LA SAUVEGARDE

Nous avons réalisé deux fonctions qui permettent de récupérer une liste des contacts et de les sauvegarder à la fin du programme. Pour l'instant nous stockons la sauvegarde séparément de la récupération pour pouvoir réaliser des tests.

Il faut noter qu'il faut également ajouter une ligne dans le fichier cmakelist.txt afin de copier le document noms2008nat_txt.txt à l'endroit où sera mis l'exécutable

```
cmake_minimum_required(VERSION 3.17)
project(Gestionnaire_Agenda C)

set(CMAKE_C_STANDARD 23)

add_executable(Gestionnaire_Agenda main.c fichier.c fichier.h
        timer.h
        timer.c
        research.c
        research.h
        research.c
        contact.c
        contact.h
)
configure_file(./noms2008nat_txt.txt noms2008nat_txt.txt COPYONLY)
```

Voici les deux fonctions

```c
void readNamesFromFile( List_contact *listContact){
    printf( format: "Ouverture des contact ...\n");
    FILE* file = fopen( Filename: "noms2008nat_txt.txt",  Mode: "r");
    if (file == NULL) {
        printf( format: "Impossible d'ouvrir le fichier.\n");
        exit( Code: EXIT_FAILURE);
    }
    printf( format: "Recuperation des contact en cours ...");
    char name[50];
    char chaine[50] = ""; // Chaîne vide de taille TAILLE_MAX
    int i=0;
    while (fgets( Buf: chaine,  MaxCount: 50, file) != NULL && i!=100) // On lit le fichier tant qu'on ne reçoit pas d'erreur (NULL)
    {
        chaine[strcspn( Str: chaine,  Control: "\n")] = '\0'; //Pour retirer le \n à la fin de la chaine
        //printf("%s", chaine);
        Contact* new =createEmptyContact();
        printf( format: "%s", chaine);
        strcpy( Dest: name,  Source: chaine); //on convertit chaque chaine en minuscule
        conversionminuscule( str: name);
        printf( format: "%s", name);

        new->nom = (char*)malloc( Size: strlen( Str: name) + 1); //on alloue de l'espace pour ajouter la chaine dans le champ new->nom
        strcpy( Dest: new->nom ,  Source: name);
        new->prenom = NULL;

        addNewContacttemp(listContact,  newContact: new); // on ajoute le contact à la ligne
        i++;
    }printf( format: "\nRecuperation terminée !\n");

    fclose(file);//on ferme le fichier
}
```

Le principe est le même quand python on ouvre un fichier avec la fonction fopen en mode lecture « r » et on regarde chaque ligne le prénom qui y est inscrit. On transforme la chaine en minuscule et on ajoute le prénom. Il reste plus qu'à fermer le fichier avec la fonction fclose.

On fait de même pour la sauvegarde mais on inverse le procéder : on récupère chaque nom dans la liste des contacts que l'on ajoute au fichier nom.txt, ouvert en mode écriture « w », avec la fonction fprintf.

```c
void saveInFile(List_contact listContact){
    printf( format: "Ouverture du fichier Contact ou Recuperation du fichier Contact ...\n");
    FILE* file = fopen( Filename: "noms.txt",  Mode: "w");
    if (file == NULL) {
        printf( format: "Impossible d'ouvrir le fichier.\n");
        exit( Code: EXIT_FAILURE);
    }
    Contact *temp = listContact.contact[0];
    printf( format: "Sauvegarde des contact ...");

    while (temp != NULL ) {
        fprintf( stream: file,  format: "%s\n", temp->nom);
        temp = temp->next[0];

    }
    fclose(file);
    printf( format: "\nSauvegarde terminee !\n");
}
```