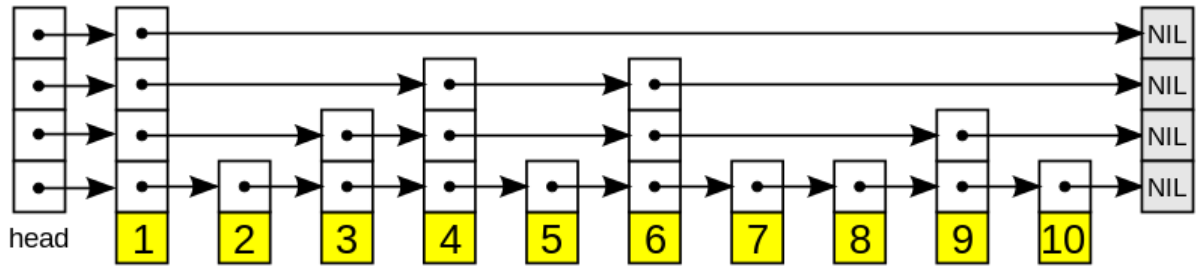


Projet de Gestion d'Agenda

Tristan GARNIER - Valentin GOURJON - Noé LACAILLE

Promo 2027 - Groupe A2 - Décembre 2023

Introduction et présentation du projet

L'évolution rapide des technologies a conduit à une explosion des données et à une demande croissante de solutions efficaces pour gérer ces informations de manière structurée. Dans ce contexte, notre projet se concentre sur le développement d'une application de gestion d'agenda en langage C, mettant en œuvre une approche novatrice basée sur les listes à niveaux.

L'objectif principal de cette application est de fournir une structure de données performante pour la gestion d'informations complexes telles que les contacts et les rendez-vous. Les listes à niveaux représentent le cœur de cette solution, offrant une alternative entre les listes chaînées traditionnelles et les arbres, avec pour résultat une meilleure complexité pour les opérations courantes telles que l'insertion, la recherche et la suppression.

Ce rapport détaillera les différentes étapes du projet, débutant par l'implémentation des listes à niveaux pour les entiers, puis étendant cette structure pour le stockage de contacts dans un agenda. Chaque section sera l'occasion d'explorer les choix de conception, les structures de données, et les algorithmes mis en œuvre.

La première partie du projet se concentre sur la création de listes à niveaux pour stocker des entiers. Nous présenterons les cellules à niveau et les listes à niveaux, accompagnées d'un programme illustratif pour mieux comprendre leur fonctionnement.

Ensuite, nous examinerons la complexité de la recherche dans une liste à niveau, en mettant en œuvre des recherches avec différents niveaux de la structure. La comparaison des temps d'exécution nous permettra d'évaluer l'efficacité de cette approche par rapport à une recherche classique effectuée uniquement au niveau 0.

Enfin, la dernière partie du projet étendra ces concepts au stockage de contacts dans un agenda. Nous aborderons la structuration des données pour les contacts, la logique de stockage sur plusieurs niveaux basée sur le nom des contacts, et enfin, la création d'une application de gestion d'agenda avec des fonctionnalités avancées telles que la complétion automatique et la sauvegarde des données.

Ce projet offre une opportunité unique d'explorer une approche innovante pour la gestion de données complexes, tout en fournissant des résultats tangibles à travers une application de gestion d'agenda fonctionnelle. La suite du rapport détaillera chaque partie du projet, mettant en lumière les choix de conception, les challenges rencontrés, et les performances obtenues.

Lien GitHub vers le projet : https://github.com/GourjonValentin/ProjetCalendar

---

Partie 1 - Implémentation des listes à niveau

Création de cellules à niveau

La création de cellules à niveau est une étape cruciale dans la mise en place de la structure de liste à niveaux. La fonction `create_cell1` a été conçue pour créer une nouvelle cellule avec une valeur spécifiée et le nombre de niveaux requis. En allouant dynamiquement la mémoire pour la cellule et ses pointeurs de niveau, cette fonction retourne un pointeur vers la cellule nouvellement créée, prête à être intégrée dans la liste à niveaux.

```c
t_d_cell1 * create_cell1(int value, int max_levels){
    t_d_cell1 *cell = malloc(sizeof(t_d_cell1));
    cell->value = value;
    cell->next = malloc(sizeof (t_d_cell1) * max_levels);
    cell->max_levels = max_levels;
    return cell;
}
```

Création de listes à niveau

La fonction `create_list1` permet la création d'une liste à niveaux en initialisant les pointeurs de tête pour chaque niveau. L'utilisateur spécifie le nombre maximal de niveaux que la liste peut avoir, et la fonction alloue dynamiquement l'espace nécessaire pour les pointeurs de tête.

```c
t_d_list1 * create_list1(int max_levels) {
    t_d_list1 *list = malloc(sizeof(t_d_list1));
    list->heads = malloc(sizeof(t_d_cell1 *) * max_levels);
    list->max_levels = max_levels;
    for (int i = 0; i < max_levels; i++) {
        list->heads[i] = NULL;
    }
    return list;
}
```

Insertion triée dans une liste à niveau

Une fonction cruciale pour la manipulation des listes à niveaux est l'insertion triée. La fonction `insert_sorted1` a été mise en place pour insérer une cellule à niveau dans la liste de manière à ce que celle-ci reste triée par ordre croissant à chaque niveau. Cette fonction utilise une stratégie optimisée pour tirer parti du tri existant dans les niveaux inférieurs.

```c
void insert_sorted_at_level1(t_d_list1 *list, t_d_cell1* cell, int level){
    if (level < 0 || level >= list->max_levels) {
        printf("Error: level %d does not exist\n", level);
        return;
    }
    if (list->heads[level] == NULL) {
        list->heads[level] = cell;
        cell->next[level] = NULL;
        return;
    }
    t_d_cell1 *temp = list->heads[level];
    if (temp->value > cell->value) {
        list->heads[level] = cell;
        cell->next[level] = temp;
        return;
    }
    while (temp->next[level] != NULL && temp->next[level]->value < cell->value) {
        temp = temp->next[level];
    }
    cell->next[level] = temp->next[level];
    temp->next[level] = cell;
}


void insert_sorted1(t_d_list1 *list, t_d_cell1* cell) {
    for (int i = 0; i < cell->max_levels; i++) {
        insert_sorted_at_level1(list, cell, i);
    }
}
```

Affichage des niveaux de la liste

L'affichage des niveaux de la liste est une fonctionnalité cruciale pour visualiser la structure de la liste à niveaux. La fonction `print_level1` permet d'afficher les éléments d'un niveau spécifié de la liste. Elle prend en compte le niveau donné et imprime les valeurs des cellules, montrant ainsi la distribution des valeurs à ce niveau.

```c
void print_level1(t_d_list1 *list, int level) {
    if (level < 0 || level >= list->max_levels) {
        printf("Error: level %d does not exist\n", level);
        return;
    }
    t_d_cell1 *cell = list->heads[level];
    printf("[list head_%d]", level);
    while (cell != NULL) {
        printf("-->[ %d|@-]", cell->value);
        cell = cell->next[level];
    }
    printf("-->NULL\n");
}

void print_list1(t_d_list1 *list) {
    for (int i = 0; i < list->max_levels; i++) {
        print_level1(list, i);
    }
}
```

Affichage aligné des niveaux de la liste

L'affichage aligné des niveaux de la liste améliore la lisibilité et la compréhension de la structure de la liste à niveaux. La fonction `print_aligned_level1` utilise des espaces et des tirets pour aligner visuellement les cellules à chaque niveau, offrant une représentation plus claire.

```c
void print_aligned_level1(t_d_list1 *list, int level) {
    if (level < 0 || level >= list->max_levels) {
        printf("Error: level %d does not exist\n", level);
        return;
    }
    int max_n = pow(2, list->max_levels)-1;
    int n_max = n_digit(max_n); // longueur max du chiffre
    int n_cell = 0;
    int i_prev = 0;
    int i_curr = 0;
    int delta_i = 0;
    t_d_cell1 *prev = NULL;
    t_d_cell1 *cell = list->heads[level];
    printf("[list head_%d]", level);
    while (cell != NULL) {
        if (prev == NULL) {
            i_prev = -1;
        } else {
            i_prev = index_of(prev->value, *list);
        }
        i_curr = index_of(cell->value, *list);
        delta_i = i_curr - i_prev - 1;
        for (int i = 0; i < delta_i * 9 + delta_i * n_max; i++) {
            printf("-");
        }
        n_cell = n_digit(cell->value);
        printf("-->[ ");
        for (int i = 0; i < n_max - n_cell; i++) {
            printf(" ");
        }
        printf("%d|@-]", cell->value);
        prev = cell;
        cell = cell->next[level];
    }
    delta_i = index_of(999999999999, *list) - i_curr - 1;
    for (int i = 0; i < delta_i * 9 + delta_i * n_max; i++) {
        printf("-");
    }
    printf("-->NULL\n");
}
```

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

La mise en œuvre de la recherche dans une liste à niveaux repose sur deux fonctions principales : la recherche classique (`classic_search1`) et la recherche optimisée (`optimized_search1`). Ces fonctions sont cruciales pour la gestion efficace des contacts et des rendez-vous dans l'agenda.

Recherche classique

La fonction `classic_search1` sert de référence pour évaluer la complexité de recherche dans une liste à niveaux. Elle effectue une recherche séquentielle à le niveau 0 de la liste, qui contient l'ensemble des valeurs, triées. Bien que cette approche garantisse une recherche exhaustive, elle peut devenir coûteuse pour des listes de grande taille.

```c
int classic_search1(t_d_list1 *list, int value) {
    int index = 0;
    t_d_cell1 *cell = list->heads[0];
    while (cell != NULL) {
        if (cell->value == value) {
            return index;
        }
        index++;
        cell = cell->next[0];
    }
    return -1;
}
```

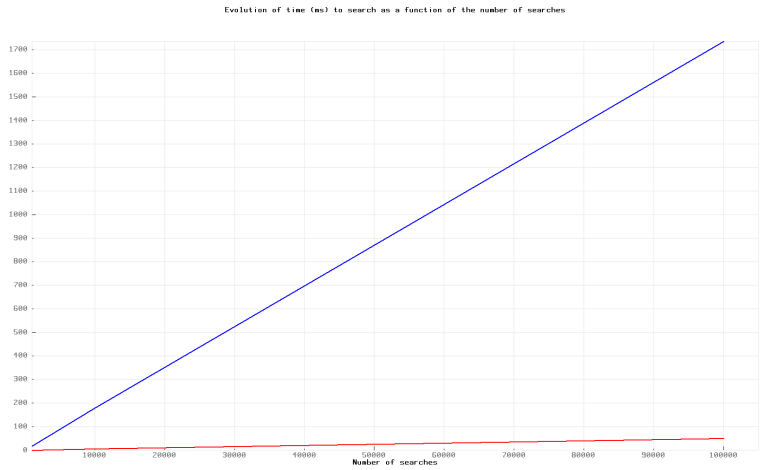
Recherche optimisée

La fonction `optimized_search1` vise à améliorer les performances de recherche en tirant parti de la structure à niveaux. Elle commence également par le niveau le plus élevé, mais elle utilise des informations sur la distribution des valeurs dans les niveaux pour déterminer si la valeur recherchée est susceptible d'être trouvée à un niveau inférieur. Cette approche permet de sauter certains niveaux, réduisant ainsi le nombre d'itérations nécessaires pour trouver la valeur cible.

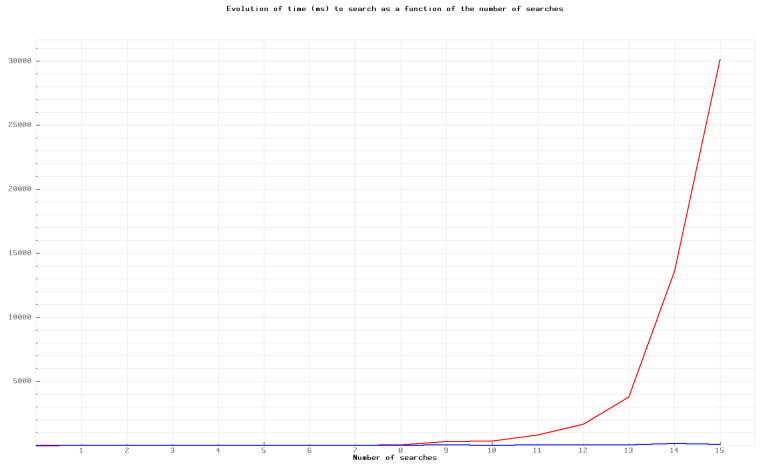
```c
int optimized_search1(t_d_list1 *list, int value) {
    int index = 0;
    int level = list->max_levels - 1;
    t_d_cell1 *prev = NULL;
    t_d_cell1 *cell = list->heads[level];
    while (level >= 0) {
        while (cell != NULL && cell->value < value) {
            index += pow(2, level);
            prev = cell;
            cell = cell->next[level];
        }
        if (cell != NULL && cell->value == value) {
            return index;
        }
        level--;
        if (prev != NULL) {
            cell = prev->next[level];
        } else {
            cell = list->heads[level];
        }
    }
    return -1;
}
```

Coût de complexité

Pour évaluer les performances de ces deux approches, des tests de complexité ont été effectués en mesurant le temps nécessaire pour effectuer la recherche pour différentes tailles de listes. Les résultats ont été générés et visualisés à l'aide de graphiques.



Les graphiques présentés illustrent clairement la différence de performance entre les deux approches. Le premier graphique, pour une liste à 16 niveaux, démontre la nette supériorité de la recherche optimisée (bleu) par rapport à la recherche classique (rouge).



Le deuxième graphique explore le temps de recherche en fonction du niveau maximal de la liste. La recherche classique (courbe rouge) montre une croissance exponentielle, soulignant son coût linéaire. À l'inverse, la recherche optimisée (courbe bleue) maintient une courbe logarithmique, assurant des performances constantes même avec une augmentation du niveau maximal.

En conclusion, la recherche optimisée, avec sa complexité logarithmique, offre une amélioration substantielle par rapport à la recherche classique, garantissant des temps de réponse rapides pour des listes de toutes tailles. Ces résultats soulignent l'efficacité essentielle de cette optimisation dans le contexte pratique de l'application. Les graphiques visuels renforcent cette compréhension en mettant en évidence la suprématie de la recherche optimisée.

Partie 3 - Stockage de contacts dans une liste à niveaux

L'application de gestion d'agenda propose un menu interactif offrant diverses fonctionnalités.

Rechercher un contact avec complétion automatique

Lors de la recherche d'un contact, l'application propose une complétion automatique à partir de la troisième lettre entrée pour le nom. Cela est réalisé en effectuant la saisie du nom caractère par caractère, offrant une expérience utilisateur améliorée.

```c
// Recherche avec complétion automatique
t_agenda_entry* search(t_d_list *agenda) {
    if (agenda == NULL) {
        printf("Error : agenda is NULL\n");
        return NULL;
    }
    int i = 1;
    int lvl_max = agenda->max_levels;
    t_d_cell *head = agenda->heads[lvl_max - 1];
    t_d_cell *cell = head;
    do {
        if (i == 1) {
            "Quelle est la première lettre du nom de la personne recherchée ?\n";
        } else if (i == 2) {
            "Quelles sont les deux premières lettres du nom de la personne recherchée ?\n";
        } else if (i == 3) {
            "Quelles sont les trois premières lettres du nom de la personne recherchée ?\n";
        } else {
            "Quel est le nom de la personne recherchée ?\n";
        }

        int j = 0;
        // liste possibilités
        while (cell != NULL  && (i == 0 ? 1 : strncmp(cell->ag_entry->contact->last_name, head->ag_entry->contact->last_name, i-1) == 0)) {
        char term;
        int choice = 0;

        // saisie sécurisée
        do {
        } while (choice < 0 || choice > j);

        for (int k = 0; k < choice - 1; k++) {
            head = head->next[lvl_max - i];
        }
        cell = head;
        if (i == 4) {
            return cell->ag_entry;
        }
        i++;

    } while (i < 5);
    return NULL; // should never happen
}
```

Afficher les rendez-vous d'un contact

L'application permet d'afficher les rendez-vous d'un contact spécifique, offrant une vue détaillée des événements associés à ce contact.

```c
void print_event_from_contact(t_agenda_entry *ag_entry){
    printf("%s ",ag_entry->contact->first_name);
    printf("%s\n",ag_entry->contact->last_name);
    if (ag_entry->events == NULL){
        printf("No events for this contact\n");
        return;
    }
    t_event_list *temp = ag_entry->events;
    while (temp != NULL){
        printf("Name : %s\n\r",temp->event->name);
        printf("Date : %d/%d/%d\n\r",temp->event->date.day,temp->event->date.month,temp->event->date.year);
        printf("Time : %dh%dm\n\r",temp->event->time.hour,temp->event->time.minute);
        printf("Duration : %dh%dm\n\r",temp->event->duration.hour,temp->event->duration.minute);
        temp = temp->next;
    }
}
```

Créer un contact et un rendez-vous

L'utilisateur peut créer un nouveau contact avec insertion automatique dans la liste. De plus, la création d'un rendez-vous pour un contact est également prise en charge, avec insertion dans la liste si le contact n'existe pas. La fonction étant trop longue, vous pouvez la consulter dans le dépôt GitHub.

Supprimer un rendez-vous

L'application permet à l'utilisateur de supprimer un rendez-vous spécifique associé à un contact donné.

```c
void del_event_from_contact(t_agenda_entry *agenda_entry, char *name){
    if (agenda_entry->events == NULL){
        printf("No events for this contact");
        return;
    }
    t_event_list *temp = agenda_entry->events;
    if (strcmp(temp->event->name,name) == 0){
        agenda_entry->events = temp->next;
        free(temp);
        return;
    }
    while (temp->next != NULL){
        if (strcmp(temp->next->event->name,name) == 0){
            t_event_list *temp2 = temp->next;
            temp->next = temp->next->next;
            free(temp2);
            return;
        }
        temp = temp->next;
    }
    printf("No event with this name");
}
```

Sauvegarder et charger un fichier de rendez-vous

L'application prend en charge la sauvegarde et le chargement des rendez-vous à partir de fichiers, offrant une fonctionnalité de persistance des données. La manipulation des fichiers texte est directement effectuée depuis le programme C.

```c
void save_calendar(t_d_list* agenda){
    printf("Quel nom voulez-vous donner au fichier ?(sans extension)\n");
    printf("ATTENTION : Si le fichier existe deja, il sera ecrase\n");
    char *name = scanString();
    char *path = malloc(sizeof(char) * 100);
    strcpy(path, name);
    strcat(path, ".txt");

    FILE *file = fopen(path, "w");
    if (file == NULL) {
        printf("Error opening file!\n");
        exit(1);
    }
    t_d_cell *temp = agenda->heads[0];
    while (temp != NULL) {
        fprintf(file, "User\n%s %s\n", temp->ag_entry->contact->first_name, temp->ag_entry->contact->last_name);
        t_event_list *temp_event = temp->ag_entry->events;
        while (temp_event != NULL) {
            fprintf(file, "Event\n%s\n", temp_event->event->name);
            fprintf(file, "%d/%d/%d\n", temp_event->event->date.day, temp_event->event->date.month, temp_event->event->date.year);
            fprintf(file, "%d/%d\n", temp_event->event->time.hour, temp_event->event->time.minute);
            fprintf(file, "%d/%d\n", temp_event->event->duration.hour, temp_event->event->duration.minute);
            temp_event = temp_event->next;
        }
        temp = temp->next[0];
    }
    fclose(file);
}
```

Conclusion

En conclusion, le développement de cette application de gestion d'agenda a été guidé par la conception d'une structure de données efficace pour la manipulation des contacts et des rendez-vous. L'implémentation d'une skip list a permis d'optimiser les opérations de recherche, insertion, et suppression, offrant une expérience utilisateur rapide et fluide.

La fonction de recherche, avec son mécanisme de complétion automatique à partir de la troisième lettre, améliore significativement l'efficacité de la recherche de contacts. De plus, la gestion des rendez-vous associés à chaque contact est transparente pour l'utilisateur, facilitant la visualisation et la modification des événements planifiés.

La création de contacts et d'événements a été simplifiée, et la suppression de rendez-vous est également accessible, garantissant une manipulation intuitive des données. Bien que la sauvegarde et le chargement de fichiers n'aient pas encore été implémentés dans la version actuelle, ils représentent des fonctionnalités potentielles pour étendre l'application.

Dans l'ensemble, l'application fournit une solution robuste et extensible pour la gestion d'agenda, mettant en avant l'efficacité de la Skip List dans la manipulation de grandes quantités de données. Les perspectives futures pourraient inclure le développement de fonctionnalités avancées, ainsi que l'intégration de la sauvegarde et du chargement de fichiers pour une expérience utilisateur complète.