Project Diary management

TI301I - Algorithms and Data Structures 2

By CHARTIER Max,BORDEL Tristan,PICHARD Tristan

Promo 2027 int-2

SUMMARY

Project Diary management

I. Introduction

When starting to code in C, we were convinced that there only exists so many types, for example int, floats and char, but at the end of last year and during this year we discovered that it is possible for the writer of the code to create new types, called structures, those can store different type of values, which is practical when we want to create more complex systems.

In our project, the “diary management”, we had to handle structures, and use in particular a mixture of structures such as linked lists and trees.

II. Objectives of the project

The main objectives of this project is to create a Diary management program, it is composed of 3 parts, that we are gonna detail in this report. We are first gonna focus on how to create a multilevel list, how to insert values into it, from the head and how to display them, the second part will be more focused on the complexity of searching in the level list, using dichotomy and normal search and finally we are gonna implement our code so that it becomes a real diary, with contacts and appoitments. It will make us use our knowledge concerning the creation of structures, of functions and of course on the use of pointers.

III. Structures and functions

A. Part I

As mentioned before, the first part is mainly focused on the understanding of Multilevel lists, we will therefore have to use the following structures:

The code simply creates the t_d_cell structure which has a value and a level both of type int and a double pointer to the following t_d_cell.=>

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

The t_d_list is a list that will contain the first cell: head which will be a t_d_cell.Then we have the maxlevel, which will be indicating the maximum number of levels it can have.

```
typedef struct s_d_list
{
    int maxlevel;
    t_d_cell **head;
} t_d_list;
```

We now need a function that creates a t_d_list, with the maximum level number , it will allocate space for the t_d_list considering the maxlevel variable given as parameter and will return the address of the t_d_list.

```
t_d_list* createEmptyList(int maxlevel)
{
    t_d_list* list;
    list = (t_d_list*) malloc(sizeof(t_d_list));
    list->maxlevel = maxlevel;
    list->head = (t_d_cell**) malloc(sizeof(t_d_cell*) * maxlevel);
    for (int i = 0; i < maxlevel; i++)
        list->head[i] = NULL;
    return list;
}
```

We now need to work on the insertions: there are gonna be

```
void insertHead(t_d_list* list, int value, int levels)
{
    t_d_cell* cell = createCell(value, levels);
    for (int curlvl = 0; curlvl < levels; curlvl++)
    {
        if (list->head[curlvl] != NULL)
        {
            cell->next[curlvl] = list->head[curlvl];
```

two types, first we want to insert a head, it will be done using the following function. It will take as parameters a pointer to a t_d_list, a value and a level. We are gonna create a cell, and then we are gonna insert it at the head of our list at the level specified.

It gets more complicated when we want to insert a value and make sure that the level is still correctly ordered, we will need to go through the list and verify each value, to know at what place we need to insert it. We will do it with the following function. The loop will be there to verify that we are below our maxlevel and the second one will verify all the values, if it's not bigger then the value we insert, we go to the next.

```c
void insertSortedCell(t_d_list* list, int value, int level)
{
    t_d_cell *cell = createCell(value, level);
    t_d_cell *tmp = NULL;
    for (int i = list->maxlevel - 1; i >= 0; i--)
    {
        if (cell->level > i)
        {
            if (list->head[i] == NULL)
            {
                cell->next[i] = list->head[i];
                list->head[i] = cell;
            }
            else
            {
                tmp = list->head[i];
                while ((tmp->next[i] != NULL))
                {
                    if (tmp->next[i]->value > value)
                        break;
                    tmp = tmp->next[i];
                }
                cell->next[i] = tmp->next[i];
                tmp->next[i] = cell;
            }
        }
    }
}
```

```c
void displayLevel(t_d_list* list, int level)
{
    t_d_cell* tmp = list->head[level];
    t_d_cell* tmp2 = list->head[0];

    printf("[list head_%d @-]--", level);

    while (tmp != NULL && tmp2 != NULL)
    {
        if (tmp2 != NULL && (tmp->value == tmp2->value))
        {
            printf(">[ %d|@]--", tmp->value);
            tmp = tmp->next[level];
        }
        else
        {
            if (tmp2->value > 9)
                printf("----------");
            else
                printf("---------");
        }
        tmp2 = tmp2->next[0];
    }
```

Now that we have the insertions and the general structure of our multilist, we need to be able to visualize it in order to verify it's functioning. We directly created the function that displays the cells aligned, cause we thought that it was important for a diary. The function on itself is long but it works on this algorithm: we will have one pointer on the first level, and one pointer on the level we want to display, if the value are the same, it prints the one on our level, if they are different, we will print a certain amount of "-" to get bellow the next value of the level 0. The rest of the functions recreate these algorithms but for the case in which our level is pointing to null but not the first. The printing of the lines depends on the value we are at, if it's more then 9, we need to print one more "-" than before.

The display all levels reuses this function but repeats it for every level of the list.

```c
void displayAllLevel(t_d_list* list)
{
    int nbr = list->maxlevel;
    for (int i=0;i<nbr;i++)
        displayLevel(list,i);
}
```

Now let's demonstrate the code: We are using the following main:

```c
int main()
{
    t_d_list* list = createEmptyList(SIZE);

    insertHead(list, 9, 1);
    insertHead(list, 8, 2);
    insertHead(list, 4, 3);
    insertSortedCell(list, 6, 1);
    insertSortedCell(list, 12, 4);


    printf("\n");

    displayAllLevel(list);

    free(list);
    return 0;
}
```

We can see on our output, that if first displays the level 2, then displays all the levels, we can see that the values inserted with the insertSortedCell are sorted, and the display is what we expected, with the values being aligned.

```
[list head_2 @-]-->[ 4|@]----------------------------->[ 12|@]-->NULL

[list head_0 @-]-->[ 4|@]-->[ 6|@]-->[ 8|@]-->[ 9|@]-->[ 12|@]-->NULL
[list head_1 @-]-->[ 4|@]------------>[ 8|@]------------>[ 12|@]-->NULL
[list head_2 @-]-->[ 4|@]----------------------------->[ 12|@]-->NULL
[list head_3 @-]------------------------------------->[ 12|@]-->NULL
```

B. Part II

We will for this part consider a new way of creating cells in our list, we will have n levels, with 2^^n-1 cells for the level 0. To have an example for this let's take the value n=3, we will have 7 cells and 3 levels:

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

Now we are gonna start looking into complexity, but before talking about it we will create our research functions, we are asked to create 2, a classic one and a dichotomous one. For the classical one, we are gonna search at the level 0, that contains all the values, we are gonna initialize a variable that will keep track of its place in this list (n). It will take as parameters a list and a

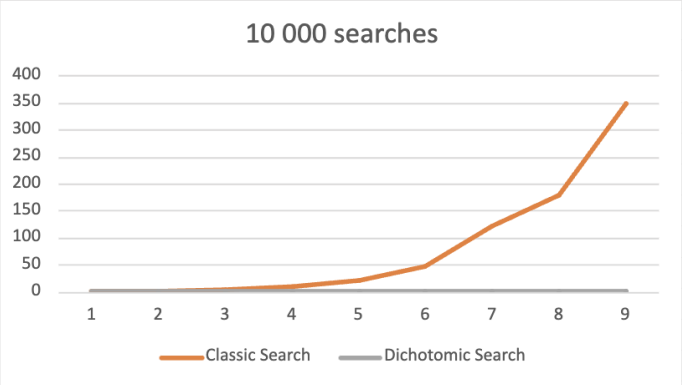
```c
int searchValue(t_d_list list, int value)
{
    t_d_cell *cell = list.head[0];
    int n = 0;
    my_bool found = false;
    while ((found == false) && (cell != NULL))
    {
        if (cell->value == value)
            found = true;
        cell = cell->next[0];
        n++;
    }
    if (found == true)
        return n;
    else
        return -1;
}
```

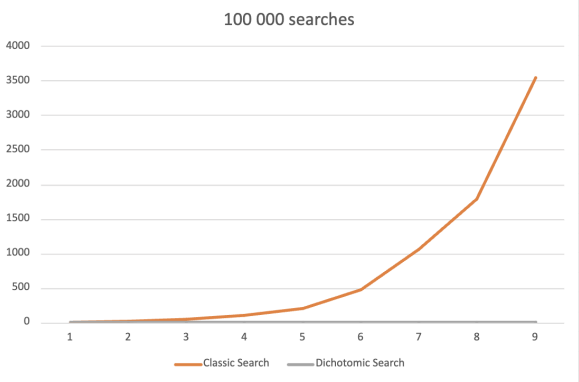
value, that we are gonna search. If the value is not found, we are gonna return -1 as an error code.

Before looking at the dichotomous one, let's remember what dichotomy is : it is an algorithm that decides between two distinct alternatives. In our case it will go to the highest level, it will look at the highest level, compare the value it has with the value n, and then go down a level and look at the cell minus the number of the value n divided by 2. Which gives us the following code:it will therefore search in a more efficient way in the list, without having to go through all of it. It will like the previous one return the position n.

```c
int dichoSearchValue(t_d_list list, int value)
{
    int cur_level = list.maxlevel - 1;
    if (cur_level < 0 || cur_level >= list.maxlevel) {
        return -2;
    }

    t_d_cell *cell = list.head[cur_level];
    if (!cell) {
        return -2;
    }

    int taille = (int)(pow(2, SIZE) - 1);
    if (taille <= 0) {
    while (cell->value != value && cur_level >= 0) {
        if (value > cell->value) {
            cell = cell->next[--cur_level];
            if (cell) {
                n += mid_val;
            } else {
                return -2;
            }
        } else {
            cell = list.head[--cur_level];
            if (cell) {
                n -= mid_val;
            } else {
                return -2;
            }
        }
        mid_val /= 2;
    }

    return n;
}
```

When we compare the efficiency of those techniques we find that the dichotomic search is exponentially faster, as we can see on the graphics. For 10 000 searches and for 100 000 searches.





To create those graphics, we used the time.h header, that gave us the time of each research, we initialized multilevel lists in order to go through them.

C. Part 3

For the last part, since we are trying to put all of our precedent work into a real diary management, we need to create new structures that will work like the previous ones, but that store other information. We have s_date that will store a date with it's day,year and month, then s_time that will store the minutes and hours, the s_appoitment structure, that will be composed of a date, a time, a length and a char type purpose: to be able to specify all the needs.

```
typedef struct s_appointment
{
    t_date date;
    t_time start;
    t_time length;
    char *purpose;
}t_appointment;
```

```
typedef struct s_date
{
    int year;
    int month;
    int day;
}t_date;

typedef struct s_time
{
    int hour;
    int minute;
}t_time;
```

We now need to store those new types in our lists that will compose the diary, the type s_cell_apt, will correspond to the new cell type, it will store an appointment type, a level, and the address of the following appointment cell, those will be store in the s_list_apt type.

```
typedef struct s_cell_apt
{
    t_appointment apt;
    int level;
    struct s_cell_apt **next;
}t_cell_apt;
```

```
typedef struct s_list_apt
{
    int maxlevel;
    t_cell_apt **head;
} t_list_apt;
```

We also have the structure creating a contact, with the address of its surname,name and concatenated name with these lists in which he belongs. Which is stored in a new type of t_cell declared as t_cell_cnt, composed of a t_contact, a level and a pointer to the next cell. Which is itself used in the t_list_cnt that will create a list but not for the appointments but for the contacts. FInally we have the t_calender type, that will simply store a t_list contact.

```
typedef struct s_contact
{
    char *surname;
    char *firstname;
    char *conc_name;
    t_list_apt appointments;
}t_contact;
```

```
typedef struct s_cell_cnt
{
    t_contact *contact;
    int level;
    struct s_cell_cnt **next;
}t_cell_cnt;
```

```
typedef struct s_calendar
{
    t_list_contact *list_cnt;
}t_calendar;
```

To be able to recognize a new contact, we created the following function:

```
t_contact* scanContact()
{
    t_contact *contact;
    contact = (t_contact*) malloc(sizeof(t_contact));
    printf("Enter your first name : ");
    contact->firstname = scanString();
    printf("Enter your surname : ");
    contact->surname = scanString();
    contact->conc_name = concUnderscore(contact->surname, contact->firstname);
    return contact;
}
```

Which will create a variable of type t_contact, create space and then ask for the user's name, surname and will create the concatenated name. It will then return the contact created. To be able to see how this function works, we created the display contact function, that basically returns the name, the surname and the concatenated form, which gives the following output:

```
PS C:\Users\Max Chartier\Desktop\Projet C\ProjectF\ProjectDiary> .\diary3
Enter your first name : Tristan
Enter your surname : Bordel
First Name : tristan
Surname : bordel
Known as : ¿$┬bordel_tristan
```

We have also converted all of our previous functions so that they can handle the new structures, meaning they were all modified. We divided our code in three different mains so that all of the parts are visible without having to modify anything.

IV. Conclusion

In conclusion, we can say that this project was a validation of our knowledge concerning this semester's work in Data structure and Algorithmics. We worked on a more complex project that gathered almost every new aspect of C we have seen this year. The combination of all of those concepts was not necessarily easy for us, but with the help of our teachers, of the internet in some cases and finally of every member of this group we managed to go as far as we could. We also had to work in a bigger group than usual, which was not always easy but taught us new ways of work using github and the decomposition of the tasks. We didn't succeed at implementing the last part of the project, meaning the 3rd, even though we managed to convert our functions to new structures, we didn't create all the functions displaying the dairy. In the future, we would like to finish this part, to truly have the satisfaction of seeing all our work in a clean form, it would also be interesting to add different features, such as a more developed interface, using colors and different ways to save your appointments depending on their importance.