Les algorithmes des fonctions de service

Remarque : Dans notre projet, on utilise data[0]='\0' pour indiquer un élément vide afin de séparer les nombres.

Void initialize(List *list)

On met le head et tail à NULL

Complexité: O(1). car on a deux opérations élémentaires.

void insert_empty_list(List *list,char *str)

```
Element *tmp;    // pointeur sur l'élément après lequel on ajoute notre nouvel élément
 Si list est vide
        calcul de NB : le nombre d'éléments nécessaires pour fabriquer str en fonction de la taille du data N
        createElemFromString(elemFromString);
        HeadDeLaListe=elemFromString;
        tmp=HeadDeLaListe ;
        Pour i de 2 à NB
                tmp->next=partieDuStringNumero(i); ;
               tmp= tmp->next ;
        Fin pour
        elemVide = creatElem;
        elemVide->next=NULL ;
        tmp=elemVide ;
        list->tail=elemVide ;
Sinon
        afficheMessageErreur
```

Complexité: O(1). Le nombre d'opérations de la boucle for est en fonction du nombre d'élément nécessaire pour stocker le nombre, ce nombre d'élement étant fini.

void insert_begining_list(List *list, char *str)

```
List *dataList;
si list n'est pas vide
        initialize(dataList); //On crée une liste temporaire
        insert_empty_list(dataList,str);  //On ajoute l'élément dans cette liste temporaire

        dataList->tail->next=list->head;
        list->head=dataList->head;
        free(dataList);
Sinon
        Afficher le message d'erreur
```

Complexité: O(1). La fonction insert_empty_list coûte O(1) et nous n'avons que des opérations élémentaires.

void insert_end_list(List *list, char *str)

```
List *dataList;
si list n'est pas vide
        allouer dynamiquement des mémoire en dataList ;
        initialize(dataList);
        insert_empty_list(dataList,str);       //création d'une liste contenant uniquement la donnée str
        list->tail->next=dataList->head;
        list->tail=dataList->tail;
        free(dataList);
Sinon
```

Afficher le message d'erreur

Complexité: O(1). La fonction insert_empty_list coûte O(1) et nous n'avons que des opérations élémentaires.

int insert_after_position(List *list, char *str, int p)

```
int cpt=0;        //compter le nb d'entiers déjà parcouru
// pos permet de mémoriser la position courante dans le parcours de Tant que
Element *pos= list->head;
List *dataList;

Si list est vide OU p<0
        Afficher le message d'erreur ;
        Sortir de fonction avec return ;
Fin si
Si p=0          // c.à.d. ajouter au début de la liste
        appeler la fonction insert_begining_list(list, str) ;
        Sortir de fonction avec return ;
Fin si

//Dans tout les autres cas(insertion à la fin de list inclus), on parcourt les éléments de list en utiisant pos
Tant que on n'est pas à la fin de liste(pos n'est pas NULL)
        Si élément i est vide
                incrémenter le compteur cpt ;
        Fin si
        Si cpt=p   // trouver la bonne position après lequel on va insérer la nouvelle donnée
                Créer une nouvelle list dataList ; //contenant uniquement la donnée str
                initialize(dataList);
                insert_empty_list(dataList, str);
                dataList->tail->next=pos->next;
                pos->next=dataList->head;
                free(dataList);
                sortir de fonction avec return;
        Fin si
        pos=pos->next ; // déplacer vers l'élément suivant
Fin Tant que


Si cpt < p       // la position indiquée par l'utilisateur n'existe pas dans la liste
        Afficher le message d'erreur
```

Complexité: O(n). Dans le meilleur de cas, soit la liste est vide, soit on insère au début, on a **Ω(1).** Et dans le pire de cas, on parcourt toute la liste, ce que nous donne O(n).

int delete_data(List* list, int p)

```
int cpt=0;        //compter le nb de nombre passé
Element *pos= list->head;     // permet de parcourir la liste
Element *posPrec;     //mémoriser la position précédente de data à supprimer
Element *dataBegin;  //permet d'indiquer le début de data à supprimer

Si list est vide OU p<0
        Afficher le message d'erreur ;
        Sortir de fonction avec return ;
```

Tant que on n'est pas à la fin de **list OU** on ne trouve pas la bonne position(cpt !=p)
pos=pos->next; //pointer vers l'élément suivant
Si **pos**->data est vide //l'élément vide
Incrémenter le compteur **cpt** ;
Si **cpt=p-1** ; //on arrive à la fin de (n-1) élément
//mémoriser la position qui se situe juste avant la donnée à supprimer
posPrec=pos;
//mémoriser le début de donnée à supprimer
dataBegin=pos->next;
Fin si
Fin si
Fin tant que
Si **cpt=p**
Si **p**=1 //Au cas ou l'on souhaite supprimer le premier data
Mémoriser la position du début de data en faisant **dataBegin**=list->head;
list->head=pos->next;
Sinon
Si on veut supprimer la dernière donnée (pos->next==NULL)
list->tail=posPrec; //bien positionner le queue de list
Fin si
posPrec->next=pos->next; //établir la liason entre l'élément précedante de data et l'élément suivant de data
Fin si
Supprimer la donnée en utilisant **dataBegin**
Fin si
Si **cpt<p**
Afficher le message d'erreur

Complexité: O(n). Pour trouver la bonne position de donnée à détruit, on doit parcourir la liste de taille n.

int compare(char *str1,char *str2)
On utilise la fonction **atoi** pour convertir une chaîne de caractère en entier.

Si **atoi**(str1)>**atoi(**str2)
Return 1 ;
Sinon
return 2 ;

Complexité: O(1). On a un test simple et dans chaque cas une opération tient en compte.

int sort(List *list);

Tant que la liste n'est pas vide
Le minimum = premier élément de la liste
On retrouve chaque nombre de la liste sous la forme d'un string
On convertit ce string en int
On apppelle compare avec le minimum et l'int qu'on vient de trouver
Si notre nouveau int est plus petit
alors il devient le minimum

Une fois la liste parcouru, le est minimum obtenu
on ajoute l'élément dans la Liste2 à la suite

On enleve cet élément de la Liste1
FinTantQue
A la fin on obtient une Liste2 trié, on fait pointé le head de Liste1 sur le head de Liste2 et le tail de Liste1 sur le tail de Liste2

Complexité: O(n²). On cherche le min de la liste n fois.

void display(List *list)
On parcourt **list** en utilisant **pos** qui désigne la position courante des éléments
Si **pos**->data n'est pas vide
Afficher **pos**->data ; //afficher le contenu
Sinon
Afficher un espace pour montrer le passage à un autre élément ;
Complexité: O(n). Il faut parcourir la liste de taille n.

void destruct(List *list)
On parcourt la liste et supprime des éléments l'un après l'autre de structure Element. Ensuite on détruit la liste tout entière.
Complexité: O(n). Il faut parcourir la liste de taille n.

Bonus

void somme(List *list){
On parcourt chaque élément de la liste, on le convertit en int
On incrémente une variable somme de la valeur de cet élément
On ajoute à la fin de la liste (O(1)) comme on a un poiteur sur tail.
}
Complexité: O(n). Il faut parcourir la liste de taille n.

Idées d'amélioration
Au niveau des possibilités d'amélioration, on peut imaginer la possibilité de faire une calculatrice avec ces nombres et cette structure comportant tous les calculs élémentaires , mais aussi des calculs plus complexes. On peut imaginer étendre ce projet avec des nombres floattant.