R5.04 - Qualité algorithmique

TD2 - Conception d'une liste chaînée

On se propose de réfléchir à l'implémentation d'une liste chaînée en C. On se concentrera sur les listes d'entiers (`int`).

On souhaite implémenter les opérations suivantes:

- insertion en tête d'un élément
- suppression d'un élément en tête de liste
- recherche d'un élément

En bonus, on pourra également implémenter les opérations suivantes:

- insertion d'un élément en position `i` dans la liste
- suppression d'un élément en position i dans la liste

Une implémentation `structurée`

Une liste chainée peut être représentée de manière abstraite par des **maillons** reliés les uns aux autres. Plus précisément, un maillon contient une *valeur* ainsi qu'une *référence* vers le maillon suivant dans la liste. Le dernier maillon de la liste possède donc une référence *nulle*.

En C, il est possible justement possible de créer des types de données complexes. Pour cela, nous utilisons les `struct`.

Les struct

Un struct se définit de la manière suivante:

```
struct monStruct {
    type1 champs1;
    type2 champs2;
    ...
}
```

La valeur associées au champs d'un `struc` peut être récupéré en utilisant la syntaxe `monStruct.champsi`.

Remarques

- Les champs sont stockés les un après les autres dans la mémoire.
- Des alignements mémoire peuvent avoir lieu.

Il est donc possible de représenter nos maillons de la manière suivante.

```c
struct maillon {
    int val;
    struct maillon * suivant;
};

struct maillon depart = {0,NULL};
```

Le lecteur attentif aura bien sur remarqué l'utilisation de la notation `maillon *`. C'est ce que l'on appelle un **pointeur** vers la zone mémoire contenant le prochain maillon. Ce pointeur n'est ni plus ni moins qu'une *adresse mémoire*.

> A noter
>
> Lorsque l'on utilise un pointeur vers un struct, l'accès aux champs se fait grâce au symbole `->`.

Quelques précisions sur les pointeurs

Il est possible d'obtenir l'adresse mémoire d'une variable en utilisant le symbole `&`. A l'inverse, on peut récupérer la valeur pointée en utilisant le symbole `*`. Voici un exemple:

```
int x = 3;
int *px = &x
*px = *px + 1
// quelle sera la valeur de x?
```

L'expression `*px = ...` permet de déréférencer le pointeur `px` et ainsi modifier directement la mémoire allouée pour la variable `x`.

Attention toutefois! Souvenons-nous qu'un pointeur est une adresse mémoire. Considérons l'exemple suivant.

```
int x = 3;
int *px = &x
*px = px + 1
printf("%d\n",x); // stdout: "-1510730656" ?!
```

Etrange non... En fait, pas tant que ca, `px` étant une adresse mémoire. Lancons la commande suivante:

```
printf("%p",(void *)(px+1));
```

L'affichage obtenu sera par exemple `0x7ffda5f41460`, ce qui correspond à l'adresse mémoire de la variable `x` (représentée par `px`) à laquelle nous avons additionné la mémoire nécessaire pour stocker un `int`, soit 4 octets. On peut donc en déduire que l'adresse de `x` est en fait `0x7ffda5f4145c` sur cet exemple.

Le phénomène que nous venons d'observer est une illustation de **l'arithmétique des pointeurs**.

A vous de jouer!

1. Pour chaque opération, faites un schéma décrivant leur fonctionnement.
2. A votre avis, laquelle de ces opérations aura un temps d'exécution proportionnelle à la taille de la liste (à savoir le nombre d'éléments qu'elle contient)? Justifiez informellement votre réponse.
3. Proposez ensuite une implémentation en C des différentes opérations sur la liste chainée.
4. Analyser la complexité en temps de chacune de ces opérations.

Complexité

La liste chaînée est ici implémentée avec des pointeurs vers des structures. Cependant il est également possible d'utiliser des **tableaux** pour implémenter une telle liste.

1. Comment pourrions-nous faire?
2. Cela changerait-il le temps d'exécution des différentes opérations sur la liste?

Exercices de consolidation

Représentation mémoire

Voici l'état d'une partie de la mémoire sur une architecture 32 bits.

Adresses	0	1	2	3
0x1110	01010010	00000000	00000000	00000000
0x1114	00000000	00000000	00000001	11111000

1. Supposons que le première élément en mémoire soit de type `char`. En supposant que l'encodage utilisé est celui du code ascii et que le code pour le `A` est 65, en déduire la lettre représentée.

Un autre type d'objet composé en `C` est l'objet `union`. La syntaxe est très similaire aux `struct`. La différence se fait au niveau de la représentation en mémoire. En effet, tous les champs commencent à la même adresse mémoire, à la différence de `struct` où les champs se suivent de manière linéraire.

2. Selon les types d'objets suivants, et en supposant que l'adresse `0x1110` corresponde à celle du premier champ, donnez les valeurs de chacun des champs:
   1. `struct obj1 {char c; unsigned int i};`
   2. `union obj2 {char c; unsigned int i};`

Variables et pointeurs

Donnez l'affichage pour les exemples suivants

```c
int x = 0;
int *ptr_x = &x;
*ptr_x += 3;
printf("%d\n",x);
```

Passage par valeur et passage par référence

Considérons le code suivant:

```c
int
multiplication(int x, int y){
    return x*y;
}

int
main(void)
{
    int x, y;
    x = 2;
    y = 3;
    printf("%d\n",multiplication(x,y));
}
```

Donnez un code équivalent en utilisant le concept de passage par référence.

Arithmétique des pointeurs

Qu'affiche le programme suivant?

```c
int t[3] = {0, 1, 2};
int *p = &t[0];
(*(p+1))--;
printf("[%d, %d, %d]\n",t[0],t[1],t[2]);
*(p+2) -= *p + 1;
printf("[%d, %d, %d]\n",t[0],t[1],t[2]);
t[t[t[0]]] *= 1;
printf("[%d, %d, %d]\n",t[0],t[1],t[2]);
```