# Langage C et Programmation Système
# TP n° 4 : Enregistrements, pointeurs, allocation

**Exercice 1 : Le dictionnaire** ***Enregistrements, tableaux, préprocesseur***

Dans cet exercice, on implémentera une bibliothèque pour un simple tableau associatif, le type abstrait qui représente un dictionnaire, c'est à dire un ensemble $D$ de pairs $(k, v)$ où $k$ est une clé, $v$ est une valeur, et pour chaque $k$ il existe une seule valeur $v$ t.q. $D(k) = v$. Avant de commencer, parcourez rapidement le texte de cet exercice pour avoir une idée globale du projet, et créez les fichiers source nécessaires **et leur Makefile**. Pendant toute l'écriture du code, créez un programme C séparé pour tester votre bibliothèque de fonctions.

1. **Types abstrait et mémoire statique.** La syntaxe pour déclarer et définir un enregistrement est donnée ci-dessous (faîtes attention au point-virgule à la fin) :

```
struct mon_enregistrement_s {
  int champ_entier;
  char* champ_pointeur_sur_caractere;
};
```

Lorsque l'on inclut un fichier d'entêtes contenant cette définition, on peut utiliser le nouveau type : `struct mon_enregistrement_s` . Pour éviter de taper `struct` à chaque utilisation, il est possible de rajouter un synonyme comme ceci :

```
typedef struct mon_enregistrement_s mon_type_t;
```

Le type `mon_type_t` est donc pareil que le type `struct mon_enregistrement_s`. Alternativement, on peut aussi définir `mon_type_t` directement :

```
typedef struct {
  int champ_entier;
  char* champ_pointeur_sur_caractere;
} mon_type_t;
```

Dans cet exercice, on supposera fixées les longueurs suivantes :
- `DICTENT_KEYLENGTH` sera la taille maximale d'une clé et vaudra 15,
- `DICTENT_VALLENGTH` sera la taille maximale d'une valeur et vaudra 63,
- `DICT_INITSIZE` sera le nombre maximal d'entrées dans le dictionnaire et vaudra 128.

Définissez ces trois longueurs fixées par des constantes symboliques dans le fichier `dict1.h` (**Indication :** se servir des macros du préprocesseur C avec `#define`).

Définissez dans le ficher `dict1.h` le type `dictent_t` pour une entrée du dictionnaire, de telle façon que l'enregistrement associé ait les champs :
- `key` : la chaîne de caractères de taille maximale `DICTENT_KEYLENGTH` (sans compter le `'\0'`) contenant la clé,
- `val` : la chaîne de caractères de taille maximale `DICTENT_VALLENGTH` (sans compter le `'\0'`) contenant la valeur.

Définissez également le type `dict_t` dont l'enregistrement contient :
- `elems` : le tableau d'entrées du dictionnaire, de taille maximale `DICT_INITSIZE`,
- `size` : un entier qui représente le nombre de cases utilisées dans le tableau.

**Indication :** Il faut bien sûr se servir des constantes précédemment définies.

2. **Initialisation.** La déclaration et l'utilisation d'une variable de type enregistrement ou de ses champs s'effectue comme d'habitude et avec une syntaxe pas surprenante aux yeux d'un codeur Java [1]. Par exemple :

```
mon_type_t enr1,enr2;
enr1.champ_entier = 1;
enr1.champ_pointeur_sur_caractere = "Un";
enr2.champ_entier = 2;
enr2.champ_pointeur_sur_caractere = "Deux";
enr1 = enr2;
```

Écrivez dans le ficher `dict1.c` les fonctions suivantes (sans oublier d'actualiser `dict1.h`) :
- `dictentcreate` qui prend en argument (un pointeur sur) une chaîne de caractères représentant une clé et (un pointeur sur) une chaîne de caractères représentant une valeur et renvoie une entrée de dictionnaire (si les tailles des arguments sont trop grandes par rapport à l'espace réservé, alors on éliminera les caractères qui "dépassent", la fonction `strncpy` pourra peut-être vous être utile),
- `dictcreate` renvoie un `dict_t` vide où la taille d'occupation est initialisée.

3. **Lecture.** Écrivez les fonctions suivantes :
- `dictdump` prend un dictionnaire comme argument et affiche son contenu sur le flux d'erreur standard.
- `dictlook` prend un dictionnaire $d$ et (un pointeur sur) une chaîne $k$ représentant une clé, et renvoie l'entier $i$ correspondant à l'indice de l'entrée de $d$ pour $k$. Si la clé $k$ n'est pas présente dans $d$, cette fonction renvoie la valeur spéciale `DICT_NOTFOUND`, une constante égale à $-1$ (que vous devez définir dans `dict1.h`).
- `dictget` prend (un pointeur sur) une chaîne de caractères $s$, un dictionnaire $d$ et (un pointeur sur) une chaîne $k$ représentant une clé, et dépose dans $s$ la valeur qui dans $d$ est associée a $k$.

4. **Modification.** Écrivez les fonctions suivantes :
- `dictadd` prend comme arguments un dictionnaire $d$, (un pointeur sur) une chaîne de caractères $k$ représentant une clé et (un pointeur sur) une chaîne de caractères $v$ représentant une valeur, et renvoie le dictionnaire obtenu en agrandissant $d$ par $(k, v)$. Cette fonction renvoie $d$. Si $k$ est présente dans $d$ ou si la taille maximale est atteinte, elle affiche un message d'erreur sur l'erreur standard.
- `dictup` prend comme arguments un dictionnaire $d$, une clé $k$ et une valeur $v$, et renvoie le dictionnaire $d$ où l'entrée de la clé $k$ est modifiée et est associée à la valeur $v$. Cette fonction renvoie $d$. Si la clé $k$ n'est pas presente dans $d$, cette fonction affiche un message d'erreur sur l'erreur standard.
- `dictrm` prend un dictionnaire $d$ et une clé $k$, et renvoie le dictionnaire $d$ où l'entrée de $k$ est enlevée. Si $k$ n'est pas présente dans $d$, elle renvoie $d$ et affiche un message d'erreur sur l'erreur standard.

1. Devinez quel langage de programmation a inspiré la syntaxe de Java.

### Exercice 2 : Le dictionnaire indirect *Utilisation de mémoire et pointeurs*

Dans cet exercice, on analysera l'utilisation de la mémoire de l'implémentation précédente et on la modifiera pour éviter de passer des enregistrements.

1. Calculer combien d'octets sont nécessaires pour allouer une entrée du dictionnaire, avec ses chaînes, et pour un dictionnaire complet, avec son tableau d'entrées.
2. Comparez précisément les tailles à l'aide de `sizeof` sur les deux types. Comprenez où les tableaux de nos types sont alloués, et par qui.
3. Copiez la bibliothèque dans des fichiers séparés (`dict2.c` et `dict2.h`). Modifiez les déclarations et definitions des fonctions des questions 3 et 4 de l'exercice 1 pour qu'elles prennent en argument et renvoient des pointeurs sur des dictionnaires, au lieu de dictionnaires. En outre, supprimez les affichages d'erreurs et substituez-les par des renvois de `NULL`.

### Exercice 3 : Le dictionnaire flexible I *Allocation dynamique*

Dans cet exercice, on modifiera l'implémentation précédente pour éviter la restriction de la taille des chaînes de caractères et du dictionnaire et pour améliorer l'utilisation de mémoire dédiée aux entrées du dictionnaire.

1. **Allocation sur le tas.** Pour gérer l'allocation d'espace de mémoire, la bibliothèque standard de C nous offre les fonctions suivantes :

   `malloc` prend le nombre d'octets à allouer et retourne un pointeur générique (pointeur sur `void`) qui correspond à l'adresse qui nous est réservée, ou `NULL` en cas d'erreur.

   `free` prend l'adresse de la mémoire à libérer.

   Par exemple, pour créer un tableau de `n` entiers dynamiquement, on peut faire : `int* tab = (int*) malloc(n * sizeof(int));`. Pour libérer l'espace réservé, lorsque l'on n'a plus besoin de ce tableau, on fera `free(tab);`.

   On rappelle que si `v` est une variable, l'expression `&v` correspond à l'adresse mémoire de `v`, et si `p` est un pointeur sur une valeur, l'expression `*p` permet d'accéder à cette valeur. Un autre point : si on a un pointeur `p` de type `mon_type_t*`, alors pour accéder au champ `champ_entier` de `*p`, on peut faire `p->champ_entier` qui correspond à `(*p).champ_entier`.

   Copiez la bibliothèque précédente dans des fichiers séparés (`dict3.c` et `dict3.h`).
   - Modifiez les types `dictent_t` et `dict_t` de manière à utiliser des pointeurs. Le tableau d'entrées du dictionnaire sera désormais un tableau de pointeurs sur `dictent_t`.
   - Modifiez la fonction `dictentcreate` pour qu'il n'y ait plus de restrictions sur la taille des clés et des valeurs.
   - Modifiez la fonction `dictcreate` pour qu'elle s'occupe de l'allocation du dictionnaire et ses données. On allouera au début encore une fois `DICT_INITSIZE` pour commencer

     Créez la procédure `dictdestroy` qui prend un pointeur vers un dictionnaire et libère la mémoire dédiée. (N'oubliez pas de libérer également la mémoire de chacune des entrées avant de libérer la mémoire du dictionnaire.)
2. **Taille dynamique.** Pour modifier la taille de la mémoire allouée préalablement par `malloc`, on a à disposition `realloc`, qui prend l'adresse du début de la zone mémoire dont on désire modifier la taille et la nouvelle taille en octet, et retourne le

pointeur vers le début de la nouvelle zone mémoire allouée (éventuellement changée, et éventuellement `NULL` en cas d'erreur).

Baissez la valeur de `DICT_INITSIZE` à 1. Rajoutez à l'enregistrement du dictionnaire un champ de type entier positif `maxsize`. Modifiez la fonction `dictcreate` pour que dans le champ `maxsize` on ait la taille allouée pour le tableau (au début on aura `maxsize` qui vaut `DICT_INITSIZE`). Modifiez les fonctions `dictadd` et `dictrm` de la façon suivante : quand le tableau est complètement rempli, doublez sa taille ; quand les trois quarts du tableau sont vides, diminuez sa taille de la moitié.

3. **Contrôle de cohérence dans l'utilisation mémoire.** Ajoutez l'option `-g` dans l'appel à `gcc` dans le Makefile, qui insert les informations de débuggage dans les fichiers objet. Lancez le profileur Valgrind sur votre executable (par exemple avec : `valgrind ./montest-dict`) et vérifiez qu'il ne détecte aucune fuite mémoire, ni dépassement. Plus d'informations dans la section «*Memcheck Options*» du manuel.

### Exercice 4 : Le dictionnaire flexible II *Liste chaînée*

On rappelle que pour faire une liste simplement chaînée d'entiers en C, on peut utiliser un enregistrement de la forme suivante :

```
typedef struct ma_liste_s {
    int champ_entier;
    struct ma_liste_s* suivant;
} ma_liste;
```

Ainsi un object de type `ma_liste*` sera un pointeur vers une cellule de liste qui a deux champs : `champ_entier` et `suivant`. Si l'on veut créer une liste de deux éléments on peut ainsi faire les instructions suivantes :

```
ma_liste* l2 = (ma_liste*) malloc(sizeof(ma_liste));
l2->champ_entier = 2;
l2->suivant = NULL;
ma_liste* l1 = (ma_liste*) malloc(sizeof(ma_liste));
l1->champ_entier = 1;
l1->suivant = l2;
```

L'instruction

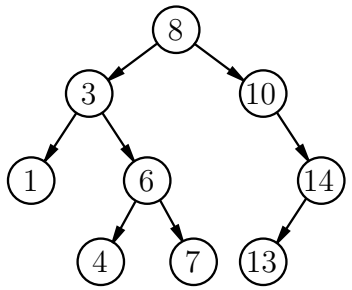
```
printf("(%d) -> (%d)\n", l1->champ_entier, l1->suivant->champ_entier);
```

affichera `(1) -> (2)`.

1. Reprenez dans des fichiers `dict4.c` et `dict4.h` votre implémentation du dictionnaire en utilisant une liste chaînée d'entrées de type `dictent_t` à la place d'un tableau d'entrées. Vous n'avez du coup plus besoin ni de `DICT_INITSIZE` ni de `maxsize`.
2. Ajoutez une fonction `dictaddord` qui insère les clés dans l'ordre lexicographique dans la liste (c'est à dire que le premier élément de la liste est le premier dans l'ordre et le dernier élément de la liste est le dernier).

**Exercice 5 : Le dictionnaire performant** ***Pointeurs++, allocation++***

Dans cet exercice, on modifiera de nouveau l'implémentation précédente pour que les opérations sur le dictionnaire aient une complexité en temps logarithmique dans le nombre d'entrées du dictionnaire (et non pas linéaire). Pour ce faire, on substituera le tableau des entrées par un arbre binaire de recherche, un arbre où chaque nœud $n$ à une clé supérieure à celle du chaque nœud du sous-arbre gauche et inférieure à celle du chaque nœud du sous-arbre droit. Un exemple où le contenu d'un nœud est un entier est montré sur la figure ci-contre.




1. **Type abstrait.** Copiez la bibliothèque dans des fichiers séparés (`dict5.c` et `dict5.h`). Modifiez le type `dictent_t` en rajoutant les champs :
   - `left` : un pointeur vers une entrée de dictionnaire, la racine de sous-arbre gauche, éventuellement `NULL`.
   - `right` : idem, mais pour le sous-arbre droit.

   Modifiez le type `dict_t` pour qu'il devienne synonyme de `dictent_t`.
2. **Initialisation et destruction.** Modifiez la fonction `dictcreate` pour qu'elle renvoie un pointeur `NULL`. Modifiez la fonction `dictdestroy` pour qu'elle s'occupe de la libération de la mémoire dédiée au nœud racine et à chaque nœud dans les sous arbres.
3. **Lecture.** Réécrivez la fonction d'affichage pour que sa sortie montre la structure arborescente (par example faire une visite en profondeur en gardant une indentation proportionnelle au niveau de profondeur). Récrivez la fonction de recherche (`dictlookup`) pour qu'elle renvoie le dernier nœud visité dans sa recherche, mais pas nécessairement ayant la clé en question. Modifiez la fonction de récupération par conséquent.
4. **Modification.** Adaptez la fonctions de mise à jour aux changements préalables. Modifiez la fonction d'insertion pour qu'elle s'occupe de l'allocation d'un nouveau nœud entrée, de l'initialisation des ses champs, et de sa liaison avec le nœud père. Modifiez la fonction de suppression de manière duale.
5. **Contrôle d'utilisation mémoire.** Relancez le profilage avec Valgrind pour vérifier la propreté de votre implémentation.