Polytech Nice Sophia
SI3 / 2021-2022
Erick Gallesio

Programmation Procédurale

Feuille 7

Pointeurs

Allocation dynamique

1 Pile de nombres

Écrire les fonctions permettant de gérer une pile d'entiers de taille quelconque. Les éléments de la pile seront du type :

```
struct element {
  int valeur;
  struct element *suivant;
};

typedef struct element Element;
```

Notes:

1. **Page Wikipedia sur la notion de pile en informatique**
2. Cette représentation n'est pas du tout efficace en termes d'occupation mémoire. Toutefois, ce qui nous intéresse ici c'est la manipulation de la liste chaînée permettant de représenter une pile.

Les fonctions à implémenter sont les suivantes:

- `push_item` pour ajouter un élément au sommet de la pile;
- `pop_item` pour enlever un élément de la pile (c'est-à-dire le dernier élément qui a été empilé) ;
- `top_value` qui renvoie l'entier en sommet de pile (sans l'enlever de la pile);
- `print_stack` pour imprimer le contenu de la pile.

Noter que c'est une erreur que d'essayer d'appeler la fonction `pop_item` sur une pile vide.

On peut écrire deux implémentations de la pile:

- une implémentation où les fonctions de gestion de la pile renvoient la nouvelle valeur de la pile;
- une implémentation où l'on passe la pile en paramètre par référence (ce qui nous conduira à avoir des pointeurs sur des pointeurs).

> La seconde écriture est un peu plus «compliquée». Elle sera donnée en corrigé, mais peut être sautée durant la séance de TD, si vous êtes en retard.

2 Listes chaînées

On veut pouvoir représenter des listes de nombres entiers en C. Pour cela, on réutilise le type d'éléments que l'on avait utilisé dans l'exercice précédent.

1. Définir le type `List` permettant de représenter une liste d'entiers.

2. Écrire les fonctions C suivantes:

```c
// Ajouter la valeur v au début de la liste lst
List prepend_element(List lst, int v);

// Ajouter la valeur v à la fin de la liste lst
List append_element(List lst, int v);

// Ajouter la valeur v dans la liste ordonnée lst
List insert_element(List lst, int v);

// Supprimer la (première) valeur égale à v de la liste lst
List delete_element(List lst, int v);

// Supprimer tous les éléments égaux à v dans la liste lst
List delete_elements(List lst, int v);

// Tester si la valeur v est dans liste lst
int find_element(List Lst, int v);

// Imprimer la liste lst
void print_list(List lst);
```

3. Modifier les fonctions précédentes qui ont un résultat de type `List` pour qu'elles soient maintenant de type `void` . Modifiez aussi, bien-sûr, votre programme de test en conséquence.

3 Lecture de chaînes de taille quelconque

Écrire une fonction permettant de lire une chaîne de caractères de taille quelconque sur le fichier standard d'entrée. Pour cela, on commencera par allouer une zone mémoire de taille `TBLOC` , cette zone mémoire étant étendue si nécessaire à l'aide de la fonction `realloc` . Cette fonction devra renvoyer une copie de la chaîne de caractères lue (i.e. dont la taille est égale au nombre de caractères lus). Lorsqu'on est en fin de fichier la fonction renverra `NULL` .

Un exemple de programme utilisant cette fonction est donné ci-dessous:

```c
int main(void) {
  char *s = NULL;

  do {
    printf("Entrer une chaîne: "); fflush(stdout);
    s = readline();
    if (s) {
      printf("Chaîne lue : '%s'\n", s);
      free(s);
    }
  } while (s);
  return 0;
}
```

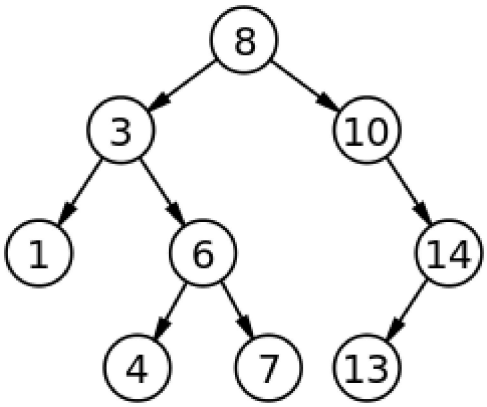
4 Arbre binaires d'entiers

Wikipédia définit un arbre binaire de la façon suivante:

> *Un **arbre binaire** est une structure de données qui peut se représenter sous la forme d'une hiérarchie dont chaque élément est appelé nœud, le nœud initial étant appelé racine. Dans un arbre binaire, chaque élément possède au plus deux éléments fils au niveau inférieur, habituellement appelés gauche et droit. Du point de vue de ces éléments fils, l'élément dont ils sont issus au niveau supérieur est appelé père.*
>
> *Au niveau le plus élevé il y a donc un nœud racine. Au niveau directement inférieur, il y a au plus deux nœuds fils. En continuant à descendre aux niveaux inférieurs, on peut en avoir quatre, puis huit, seize, etc. c'est-à-dire la suite des puissances de deux. Un nœud n'ayant aucun fils est appelé feuille. Le nombre de niveaux total est appelé hauteur de l'arbre.*
>
> *Le niveau d'un nœud, autrement dit la distance entre la feuille la plus éloignée et la racine, est appelé profondeur.*

Un **arbre binaire de recherche (ou ordonné)** est défini comme

> *un arbre binaire dans lequel chaque nœud possède une clé, telle que chaque nœud du sous-arbre gauche ait une clé inférieure ou égale à celle du nœud considéré, et que chaque nœud du sous-arbre droit possède une clé supérieure ou égale à celle-ci — selon la mise en œuvre de l'ABR, on pourra interdire ou non des clés de valeur égale. Les nœuds que l'on ajoute deviennent des feuilles de l'arbre.*



Arbre binaire de recherche (source Wikipédia)

On peut représenter un arbre binaire ordonné d'entiers avec la structure C suivant

```c
struct node {
    int value;
    struct node *left;
    struct node *right;
};

typedef struct node *tree;
```

Définir alors les fonctions suivantes

```c
tree create_empty_tree(void);  // créer un nouvel arbre vide

tree add_tree(tree t, int v);  // ajouter v dans l'arbre t. Cette fonction renvoie
                               // l'arbre dans lequel v a été ajouté.`{.c}

void print_tree(tree t);       // afficher l'arbre t (dans l'ordre)

tree find_tree(tree t, int v); // Renvoyer l'arbre de racine v (NULL si absent)

void free_tree(tree t);        // Libérer l'espace occupé par t
```

Un exemple de programme de tests:

```c
int main() {
  tree t = create_empty_tree();

  // Création de l'arbre du sujet
  t = add_tree(t, 8);
  t = add_tree(t, 3);
  t = add_tree(t, 10);
  t = add_tree(t, 1);
  t = add_tree(t, 6);
  t = add_tree(t, 7);
  t = add_tree(t, 4);
  t = add_tree(t, 14);
  t = add_tree(t, 13);

  // Impression de l'arbre (trié)
  printf("Valeurs dans l'arbre: ");
  print_tree(t);
  printf("\n");

  // Recherches
  printf("\nRecherches:\n");
  for (int i = 0; i < 20; i ++) {
    printf("%2d: %s, ", i, find_tree(t, i)? "oui": "non");
    if (i % 5 == 4) printf("\n");
  }

  // Free
  printf("\nLibération mémoire de l'arbre:\n");
  free_tree(t);

  return 0;
}
```

et la sortie obtenue à l'exécution (ici la fonction `free_tree` affiche les nœuds effacés pour comprendre dans quel ordre les choses se passent).

```
Valeurs dans l'arbre: 1 3 4 6 7 8 10 13 14

Recherches:
 0: non,  1: oui,  2: non,  3: oui,  4: oui,
 5: non,  6: oui,  7: oui,  8: oui,  9: non,
10: oui, 11: non, 12: non, 13: oui, 14: oui,
15: non, 16: non, 17: non, 18: non, 19: non,

Libération mémoire de l'arbre:
Freeing space for key 1
Freeing space for key 4
Freeing space for key 7
Freeing space for key 6
Freeing space for key 3
Freeing space for key 13
Freeing space for key 14
Freeing space for key 10
Freeing space for key 8
```

5 Facultatif: La fonction strtok

Écrire la fonction `strtok` dont le prototype est:

```
char *strtok(char *s, const char *delim);
```

Vérifier le comportement de cette fonction dans le manuel. Un exemple d'utilisation de cette fonction est donné ci-dessous.

```c
#include <stdio.h>
#include <string.h>
int main(void)
{
  char phrase[] = "Une phrase composée de mots."
  char *p;
  for (p = strtok(phrase, " ."); p ; p = strtok(NULL, " ."))
    printf("mot '%s'\n", p);
  return 0;
}
```

L'exécution de ce programme donnerait:

```
$ ./strtok
mot 'Une'
mot 'phrase'
mot 'composée'
mot 'de'
mot 'mots'
$
```