Algorithmique et Structures de données 1

L2 2021-2022

Travaux Pratiques 7

Site du cours : `https://defelice.up8.site/algo-struct.html`
Les exercices marqués de (@) sont à faire dans un second temps.
Un fichier écrit en langage C se termine conventionnellement par `.c`.
Une commande de compilation est `gcc fichier_source1.c fichier_source2.c fichier_source3.c`.
Voici des options de cette commande.

- `-o nom_sortie` pour donner un nom au fichier de sortie (par défaut `a.out`).
- `-Wall -Wextra` pour demander au compilateur d'afficher plus de Warnings
- `-std=c11` pour compiler selon la norme C11
- `-g -fsanitize=address` pour compiler avec information de débogage et en interdisant la plupart des accès a une zone mémoire non reservée.

Exemple : `gcc -Wall fichier1.c -o monprogramme`

Exercice 1. *Pile*

Implanter les fonctions suivantes manipulant des piles d'entiers de type `pileI_t` (I pour int, pour la définition de `pileI_t` voir plus bas) :

- `pileI_t* initialiserPI(void);` renvoie l'adresse d'une pile d'entier après l'avoir initialisée
- `void emPilerI(pileI_t* p,int a);`.
- `int dePilerI(pileI_t* p);`.
- `int estVidePI(pileI_t* p);` qui renvoie 1 si la pile est vide
- `void detruirePI(pileI_t* p);`.

```
typedef struct
{
   int n;  //nombre d'élément actuel de la pile
   int* tab; // tableau de taille tMax destiné à contenir des int
   int tMax; // capacité actuelle maximum de la pile (taille de tab)
} pileI_t;
```

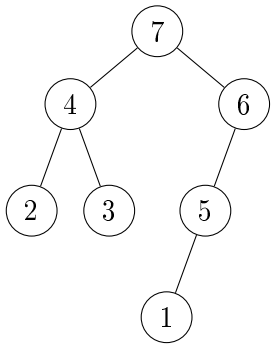
On doit veiller à agrandir la pile (par exemple de 100 cases supplémentaires) lorsque la pile est pleine et que l'on souhaite empiler un nouvel élément.

La suite du TP utilise une structure d'arbre binaire étiqueté `noeud_t` dont voici la définition :

```
typedef struct s_noeud_t
{
   int v; // étiquette du noeud (v pour valeur)
   struct s_noeud_t* g; // pointeur vers la racine du sous-arbre  gauche
   struct s_noeud_t* d; // pointeur vers la racine du sous-arbre droit
} noeud_t;
```

Exercice 2. *En largeur*

Implanter `void parcoursLargeur(noeud_t* a)` qui parcourt un arbre en largeur en affichant les étiquettes des nœuds. Par exemple un parcours en largeur de l'arbre suivant donnera : 7 4 6 2 3 5 1.



Pour effectuer un parcours en largeur il est nécessaire d'implanter une file qui mémorise les adresses des noeuds. Une idée de l'algorithme est le suivant.

1. On initialise en enfilant la racine.
2. Ensuite tant que la file n'est pas vide :
   (a) On retire le prochain noeud.
   (b) On affiche son étiquette.
   (c) On enfile ses deux fils, d'abords le gauche, puis le droit

Pour implanter la file vous pouvez utiliser cette structure.

```
typedef struct
{
   int queue; // place de la queue de la file (la dernière valeure enfilée)
   int tete;  // place de la tête de file (la plus ancienne valeur contenue)
   noeud_t** tab; // tableau d'adresse noeud_t* de taille tMax (il y a bien 2 étoile)
   int tMax; // capacité maximum de la file
} fileA_t;
```

Exercice 3. *@ Profondeur sans récursion*

Écrire une fonction `void parcoursPrefixe(cellule_t* a)` SANS appel récursif qui affiche les étiquettes des noeuds de l'arbre `a` en un parcours en profondeur préfixe.

Exercice 4. *@ Construction suffixe*

Écrire une fonction `cellule_t* construireSuffixe(int* tab)` qui construit un arbre étiqueté par des entiers positifs à partir d'un parcours suffixe de ses noeuds. **La fonction ne doit pas utiliser d'appel récursif.** L'arbre vide sera codé par -1 et la fin du parcours par un entier inférieur ou égale à -2. (Aide : s'inspirer de l'exercice 3 du TD8). Une suite invalide (qui ne correspond pas à un parcours suffixe) renverra NULL et n'allouera pas de zones mémoires supplémentaires.

Exemple :

```
int T[]={-1,-1,2,-1,-1,3,4,-1,-1,5,-1,6,7,-2,-1,412,44};
cellule_t a=construireSuffixe(T);
```

Construira l'arbre suivant :