Algorithmique et structures de données

Structures de données dynamiques

Samar MOUCHAWRAB, PhD Eng

2A Cycle Préparatoire – Semestre 1
2017/2018

Les piles

Définition

Si on veut enlever un livre de la pile de livres ci-dessous, il faut commencer à dépiler du dernier élément ajouté dans la pile.

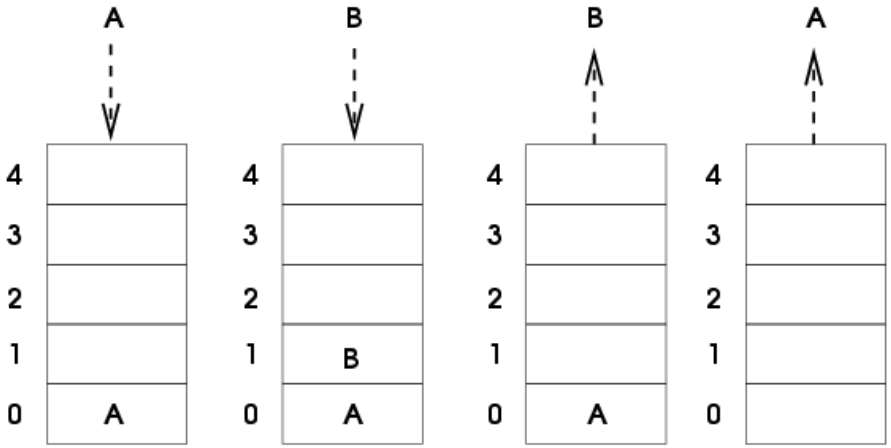
Le premier livre mis dans la pile, le bleu, est le dernier à être enlevé de la pile.

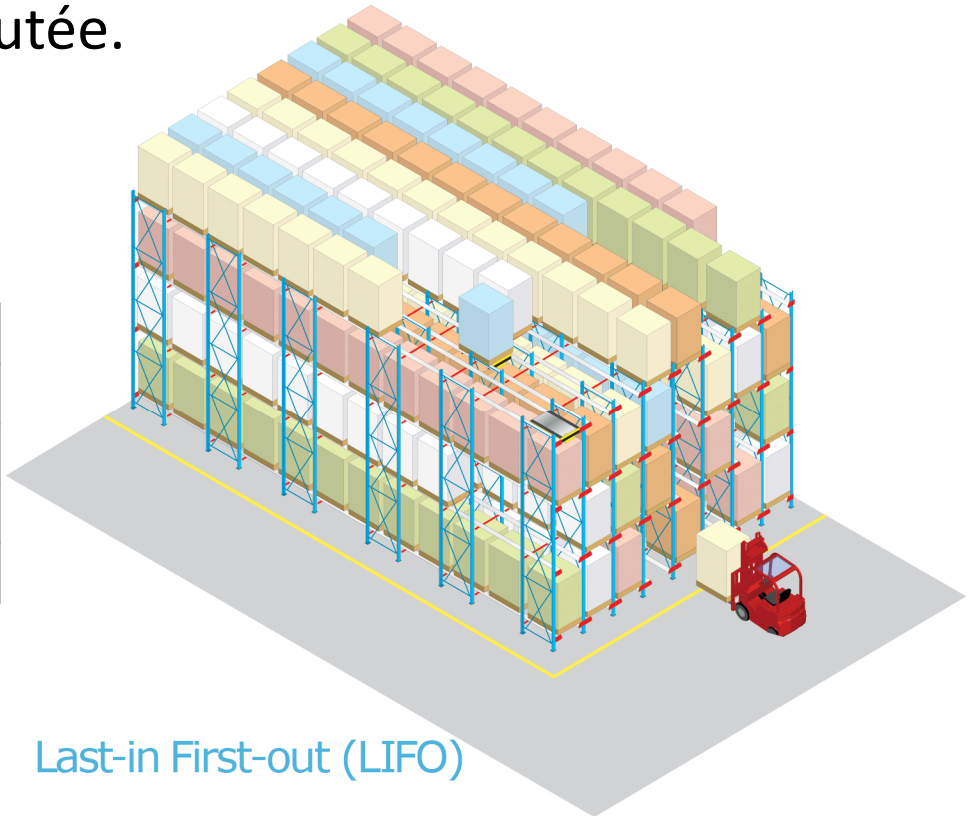
En informatique, une pile (en anglais stack) est une structure de données fondée sur le principe « dernier arrivé, premier sorti » (ou LIFO pour Last In, First Out), ce qui veut dire que les derniers éléments ajoutés à la pile seront les premiers à être récupérés.

Fonctionnement des piles

Le principe des piles en programmation est de stocker des données au fur et à mesure les unes au-dessus des autres pour pouvoir les récupérer plus tard.

Le fonctionnement est donc celui d'une pile d'assiettes : on ajoute des assiettes sur la pile, et on les récupère dans l'ordre inverse, en commençant par la dernière ajoutée.





Last-in First-out (LIFO)

Opérations

Les opérations principales sur les piles sont :

- Créer une pile vide
- Vérifier si la pile est vide ou non
- Empiler : ajouter un élément au sommet de la pile
- Retourner l'élément au sommet de la pile
- Dépiler : retirer l'élément du sommet de la pile

Applications

Dans un navigateur web, une pile sert à mémoriser les pages Web visitées. L'adresse de chaque nouvelle page visitée est empilée et l'utilisateur dépile l'adresse de la page précédente en cliquant le bouton « Afficher la page précédente ».

L'évaluation des expressions mathématiques en notation post-fixée utilise une pile.

La fonction « Annuler la frappe » (en anglais « Undo ») d'un traitement de texte mémorise les modifications apportées au texte dans une pile.

Un algorithme de recherche en profondeur utilise une pile pour mémoriser les nœuds visités.

Les algorithmes récursifs admis par certains langages utilisent implicitement une pile d'appel.

Implémentation

Chaque élément de la pile aura une structure identique à celle d'une liste chaînée :

```
typedef struct Element Element;
struct Element {
    int nombre;
    Element *suivant;
};
```

La structure de contrôle contiendra l'adresse du premier élément de la pile, celui qui se trouve tout en haut :

```
typedef struct Pile Pile;
struct Pile {
    Element *premier;
} ;
```

Contrairement aux listes chaînées, on ne parle pas d'ajout ni de suppression. On parle d'empilage et de dépilage car ces opérations sont limitées à un élément précis.

Initialisation

A l'exemple de la fonction initialisation d'une liste chaînée, cette fonction initialise la pile avec un nombre et retourne un pointeur sur la pile.

```
Pile * initialisation(int Nb) {
    Pile *pile = malloc(sizeof(*pile));
    Element *element = malloc(sizeof(*element));

    if (pile == NULL || element == NULL) {
        exit(EXIT_FAILURE);
    }

    element->nombre = Nb;
    element->suivant = NULL;
    pile->premier = element;

    return pile;
}
```

Empilage

La fonction empiler prendra en paramètre la structure de contrôle de la pile (de type Pile) ainsi que le nouveau nombre à stocker.

```
void empiler(Pile *pile, int nvNombre) {
    Element *nouveau = malloc(sizeof(*nouveau));
    if (pile == NULL || nouveau == NULL)
    {
        exit(EXIT_FAILURE);
    }

    nouveau->nombre = nvNombre;
    nouveau->suivant = pile->premier;
    pile->premier = nouveau;
}
```

L'ajout se fait en début de pile car, comme on l'a vu, il est impossible de le faire au milieu d'une pile. C'est le principe même de son fonctionnement, on ajoute toujours par le haut.

Dépilage

Le rôle de la fonction de dépilage est de supprimer l'élément tout en haut de la pile. Mais elle doit aussi retourner l'élément qu'elle dépile, c'est-à-dire le nombre qui était stocké en haut de la pile.

```
int depiler(Pile *pile){
    if (pile == NULL) {
        exit(EXIT_FAILURE);
    }

    int nombreDepile = 0;
    Element *elementDepile = pile->premier;

    if (pile != NULL && pile->premier != NULL) {
        nombreDepile = elementDepile->nombre;
        pile->premier = elementDepile->suivant;
        free(elementDepile);
    }

    return nombreDepile;
}
```

Affichage d'une pile

Cette fonction sert principalement pour tester le fonctionnement de notre pile et surtout pour « visualiser » le résultat.

```
void afficherPile(Pile *pile) {
    if (pile == NULL) {
        exit(EXIT_FAILURE);
    }
    Element *actuel = pile->premier;

    while (actuel != NULL) {
        printf("%d\n", actuel->nombre);
        actuel = actuel->suivant;
    }

    printf("\n");
}
```

Exercice

Ecrire une fonction CopiePile recevant une pile comme argument et renvoyant une copie de cette pile. Attention, la pile en paramètre doit être conservée !

Exercice

```
Pile * CopiePile (Pile *pile) {
    int n;
    if (pile == NULL) {
        exit(EXIT_FAILURE);
    }
    Pile *copie = malloc(sizeof(*copie));
    Pile *temp = malloc(sizeof(*temp));
    if (copie != NULL && temp != NULL) {
        n = depiler(pile);
        temp = initialisation(n);
        while (pile->premier != NULL) {
                n = depiler(pile);
                empiler(temp, n);
        }
        n = depiler(temp);
        pile = initialisation(n);
        copie = initialisation(n);
        while (temp->premier != NULL) {
                n = depiler(temp);
                empiler(pile, n);
                empiler(copie, n);
        }
    }
    free(temp);
    return copie;
}
```

Exercice

```
// pour tester la fonction CopiePile

void main() {
    Pile *pile, *copie;

    pile = initialisation(3);
    empiler(pile, 5);
    empiler(pile, 10);

    afficherPile(pile);

    copie = CopiePile(pile);

    afficherPile(copie);

    free(pile);
    free(copie);
}
```

Exercice

Ecrire un programme avec les fonctions nécessaires qui crée une pile de livres (un livre est défini par son étiquette : « math », « fr », « eng », « info », ou « socio ». La pile est initialisée et remplit avec plusieurs livres et ensuite dépilée un à un pour calculer le nombre de livres de chaque catégorie.

Exercice

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct Element Element;
struct Element
{
    char* nom;
    Element *suivant;
};

typedef struct Pile Pile;
struct Pile {
    Element *premier;
} ;


Pile * initialisation(char* str) {
    Pile *pile = malloc(sizeof(*pile));
    Element *element = malloc(sizeof(*element));

    if (pile == NULL || element == NULL) {
        exit(EXIT_FAILURE);
    }
```

Exercice

```
    element->nom = str;
    element->suivant = NULL;
    pile->premier = element;

    return pile;
}

void empiler(Pile *pile, char* str) {
    Element *nouveau = malloc(sizeof(*nouveau));
    if (pile == NULL || nouveau == NULL)
    {
        exit(EXIT_FAILURE);
    }

    nouveau->nom = str;
    nouveau->suivant = pile->premier;
    pile->premier = nouveau;
}

char* depiler(Pile *pile){
    if (pile == NULL) {
        exit(EXIT_FAILURE);
    }
```

Exercice

```
    char* str = "";
    Element *elementDepile = pile->premier;

    if (pile != NULL && pile->premier != NULL) {
        str = elementDepile->nom;
        pile->premier = elementDepile->suivant;
        free(elementDepile);
    }

    return str;
}

void afficherPile(Pile *pile) {
    if (pile == NULL) {
        exit(EXIT_FAILURE);
    }
    Element *actuel = pile->premier;

    while (actuel != NULL) {
        printf("%s\n", actuel->nom);
        actuel = actuel->suivant;
    }

    printf("\n");
}
```

Exercice

```
void main() {
        Pile *pile;
        char* str;
    pile = initialisation("math");
        empiler(pile, "info");
        empiler(pile, "fr");
        empiler(pile, "eng");
        empiler(pile, "math");
        empiler(pile, "math");
        empiler(pile, "info");
        afficherPile(pile);
        int math = 0;
        int fr = 0;
        int eng = 0;
        int info = 0;
        int socio = 0;

        Element* actuel = pile->premier;
```

Exercice

```
        while (actuel != NULL) {
                str = depiler(pile);
                if (! strcmp(str, "math")) math++;
                else if (! strcmp(str, "info")) info++;
                else if (! strcmp(str, "fr")) fr++;
                else if (! strcmp(str, "eng")) eng++;
                else if (! strcmp(str, "socio")) socio++;
                actuel = pile->premier;
        }
        printf("\nNombre livres de Maths : %d", math);
        printf("\nNombre livres d'Informatique : %d", info);
        printf("\nNombre livres de Francais : %d", fr);
        printf("\nNombre livres d'Anglais : %d", eng);
        printf("\nNombre livres de Sociologie : %d", socio);
}
```

Les files

Définition

Une file, queue en anglais, est une structure de données basée sur le principe « premier entré, premier sorti », en anglais First In First Out (FIFO), ce qui veut dire que les premiers éléments ajoutés à la file seront les premiers à être récupérés.

Le fonctionnement ressemble à une file d'attente : Les premières personnes à arriver sont les premières personnes à sortir de la file.

Opérations

Les opérations principales sur les files sont :

- Créer une file vide
- Vérifier si la file est vide ou non
- Ajouter un élément à la file (Enqueue)
- Enlever un élément de la file (Dequeue)
- Calculer la taille de la file (nombre d'éléments dans la file)

Applications

En général, on utilise des files pour mémoriser temporairement des transactions qui doivent attendre pour êtres traitées.

Les serveurs d'impression, qui doivent traiter les requêtes dans l'ordre dans lequel elles arrivent, et les insèrent dans une file d'attente (ou une queue).

En programmation, les files sont utiles pour mettre en attente des informations dans l'ordre dans lequel elles sont arrivées. Par exemple, dans un logiciel de chat (type messagerie instantanée), si vous recevez trois messages à peu de temps d'intervalle, vous les enfilez les uns à la suite des autres en mémoire. Vous vous occupez alors du premier message arrivé pour l'afficher à l'écran, puis vous passez au second, et ainsi de suite.

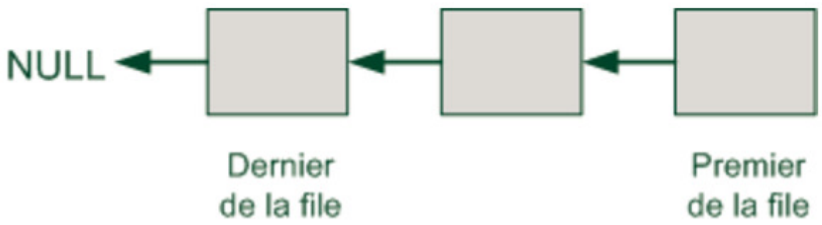
Implémentation

Comme une liste chaînée, une file est une liste où chaque élément pointe vers le suivant. Le dernier élément de la file pointe vers NULL.

```
typedef struct Element Element;
struct Element {
    int nombre;
    Element *suivant;
};
```

La structure de contrôle contiendra l'adresse du premier élément de la file, celui qui se trouve tout en haut :

```
typedef struct File File;
struct File {
    Element *premier;
} ;
```



Initialisation d'une file vide

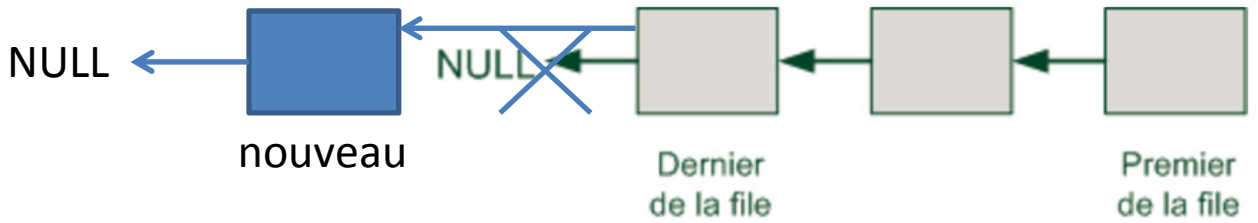
On initialise la file à NULL (file vide sans éléments)

```
File * initialisation() {
       File *file = malloc(sizeof(*file));
       file->premier = NULL;
       return file;
}
```

Enfilage

La fonction qui ajoute un élément à la file est appelée fonction d'enfilage. Il y a deux cas à gérer :

- soit la file est vide, dans ce cas on doit juste créer la file en faisant pointer *premier* vers le nouvel élément créé
- soit la file n'est pas vide, dans ce cas il faut parcourir toute la file en partant du premier élément jusqu'à arriver au dernier. On rajoutera notre nouvel élément après le dernier.



Enfilage

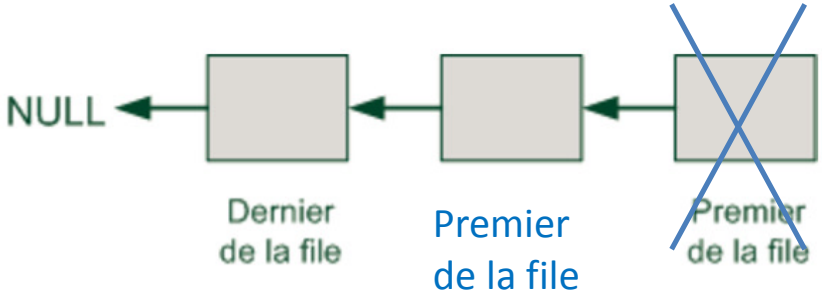
```
void enfiler(File *file, int nvNombre) {
    Element *nouveau = malloc(sizeof(*nouveau));
    if (file == NULL || nouveau == NULL) {
        exit(EXIT_FAILURE);
    }

    nouveau->nombre = nvNombre;
    nouveau->suivant = NULL;

    if (file->premier != NULL) /* La file n'est pas vide */{
        /* On se positionne à la fin de la file */
        Element *elementActuel = file->premier;
        while (elementActuel->suivant != NULL) {
            elementActuel = elementActuel->suivant;
        }
        elementActuel->suivant = nouveau;
    }
    else /* La file est vide, notre élément est le premier */ {
        file->premier = nouveau;
    }
}
```

Défilage

Le défilage ressemble au dépilage. Étant donné qu'on possède un pointeur vers le premier élément de la file, il nous suffit de l'enlever et de renvoyer sa valeur.



Défilage

```
int defiler(File *file)
{
    if (file == NULL)
    {
        exit(EXIT_FAILURE);
    }

    int nombreDefile = 0;

    /* On vérifie s'il y a quelque chose à défiler */
    if (file->premier != NULL)
    {
        Element *elementDefile = file->premier;

        nombreDefile = elementDefile->nombre;
        file->premier = elementDefile->suivant;
        free(elementDefile);
    }

    return nombreDefile;
}
```

Affichage de file

Cette fonction sert principalement pour tester le fonctionnement de la file et surtout pour « visualiser » le résultat.

```
void afficherFile(File *file) {
    if (file == NULL) {
        exit(EXIT_FAILURE);
    }
    Element *actuel = file->premier;

    while (actuel != NULL) {
        printf("%d->", actuel->nombre);
        actuel = actuel->suivant;
    }

    printf("\n");
}
```

Exercice

Ecrire un programme qui reçoit des messages de l'utilisateur et les stocke dans une file, puis les traite dans l'ordre.

Les messages peuvent être: 1 = Ouvrir porte, 2 = Fermer porte.

Au départ, on suppose que la porte est fermée.
Le traitement des messages se fait comme suit:

- Si message reçu = Ouvrir porte et la porte est fermée, afficher message : Demande traitée, la porte est ouverte
- Si message reçu = Ouvrir porte et la porte est ouverte, afficher message : La porte est déjà ouverte, pas d'actions
- Si message reçu = Fermer porte et la porte est fermée, afficher message : La porte est déjà fermée, pas d'actions
- Si message reçu = Fermer porte et la porte est ouverte, afficher message : Demande traitée, la porte est fermée

Exercice

```
void main() {
        int c;
        File *file = initialisation();

        printf("Entrer vos commandes: 1 pour Ouvrir Porte, 2 pour
          Fermer Porte. Quand vous voulez arreter, saisir 0:\n");
        do {
                scanf("%d", &c);
                if ( c==1 || c ==2 )
                        enfiler(file, c);
        } while(c!=0);

        afficherFile(file);

        printf("\n Traitement de commandes : La porte est
            initialement fermee \n");
        Element *actuel = file->premier;
        int etat = 1; // 1 pour porte fermee, 0 pour ouverte
```

Exercice

```
        while(actuel != NULL) {
            c = defiler(file);
            if (c == 1 && etat == 1) {
               printf("Demande traitee,la porte est ouverte\n");
               etat = 0;
            }
            else if (c == 2 && etat == 1)
               printf("La porte est deja fermee, pas d'actions\n");
            else if (c == 1 && etat == 0)
               printf("La porte est deja ouverte,pas d'actions\n");
            else if (c == 2 && etat == 0) {
               printf("Demande traitee, la porte est fermee\n");
               etat = 1;
            }
            actuel = file->premier;
        }

        printf("\n Fin des commandes \n");
        free(file);
}
```