Les piles en C

par Nicolas Joseph

Date de publication : 27 Juillet 2005

Dernière mise à jour :

Les structures de données en C. Troisième partie : les piles.

I - Introduction

Pour poursuivre la découverte des différentes structures de données, nous allons maintenant nous attarder sur les piles.

II - Principe

Les piles peuvent être représentées comme une pile d'assiettes, vous pouvez ajouter des assiettes au sommet de la pile et lorsque vous voulez en enlever une, il s'agit de la dernière ajoutée : on parle de liste LIFO (**L**ast **I**n **F**irst **O**ut). Les piles ne sont que des cas particuliers de listes chaînées dont les éléments ne peuvent être ajoutés et supprimés qu'en fin de liste. De ce fait la manipulation s'en trouve grandement simplifiée puisqu'elle ne nécessite que deux fonctions :

- Une fonction pour ajouter un élément au sommet de la pile
- Une seconde pour le retirer

III - En pratique

III-A - La structure d'un élément

Pour représenter un élément de la pile, il nous suffit de reprendre la structure d'un élément d'une liste doublement chaînée.

```
typedef struct stack
{
   struct stack *prev;
   struct stack *next;
   void *data;
} stack_s;
```

III-B - Initialisation

Rien de nouveau côté initialisation, on s'assure juste que le pointeur qui servira de pile est bien mit à NULL.

```
stack_s *stack_new (void)
{
   return (NULL);
}
```

III-C - Empiler un élément

Les éléments ne peuvent être ajoutés qu'en fin de liste, il n'est donc plus nécessaire de se préoccuper d'un éventuel élément suivant. A part ce détail, c'est exactement le même code que pour les listes doublement chaînées.

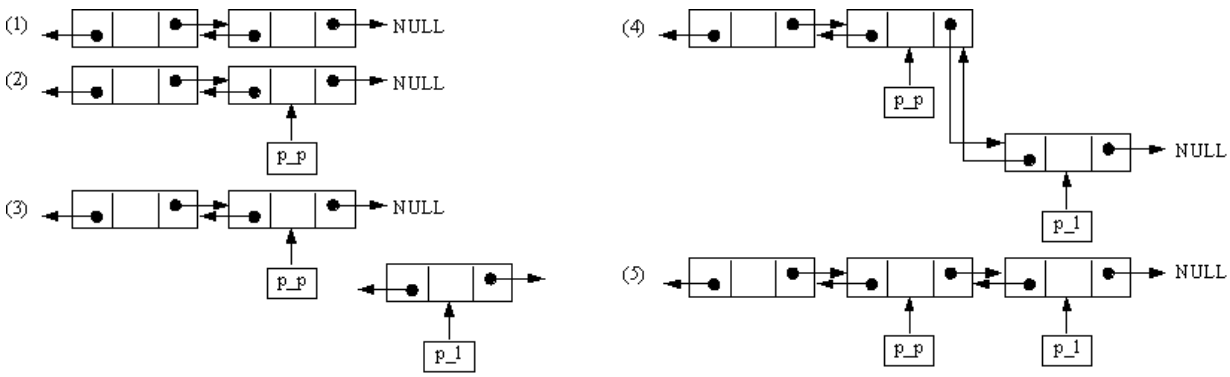

Figure 1: représentation de l'ajout d'un élément à une pile.

1 Voici l'état de la pile avant l'appel de la fonction
2 Le pointeur *p_p* représente l'argument de la fonction, c'est donc après lui qu'il faut insérer un élément
3 Création d'un nouvel élément pointé par *p_l*
4 On ajoute le nouvel élément à la fin de la pile
5 Dernier point qui n'est pas présent sur le schéma ci-dessus pourtant trés important : faire pointer *pp_pile* sur le nouvel élément

```
void stack_push (stack_s ** pp_stack, void *data)
{
   if (pp_stack != NULL)
   {
```

```
   /* (2) */
         stack_s *p_p = *pp_stack;
         stack_s *p_l = NULL;

   /* (3) */
         p_l = malloc (sizeof (*p_l));
         if (p_l != NULL)
         {
            p_l->data = data;
   /* (4) */
            p_l->next = NULL;
            p_l->prev = p_p;
            if (p_p != NULL)
               p_p->next = p_l;
   */ (5) */
            *pp_stack = p_l;
         }
         else
         {
            fprintf (stderr, "Memoire insuffisante\n");
            exit (EXIT_FAILURE);
         }
      }
      return;
   }
```

III-D - Dépiler un élément

L'élément à dépiler correspond obligatoirement au dernier ajouté qui se trouve au sommet de la pile. Généralement un élément est retiré de la pile pour pouvoir être utilisé, il faut donc retourner l'élément retiré.

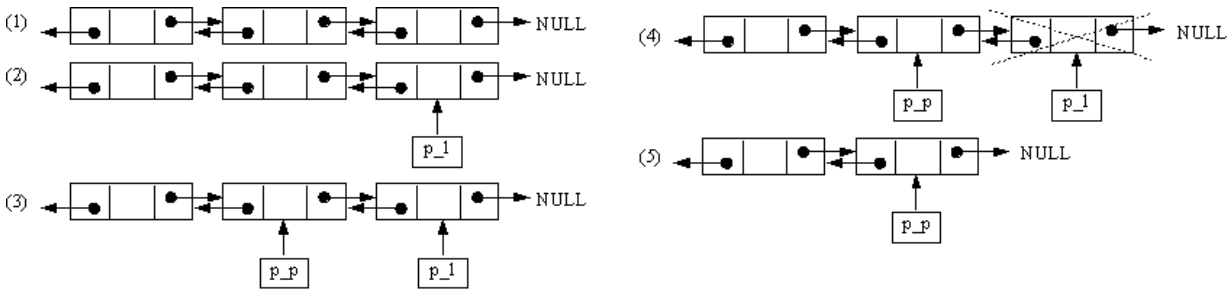

Figure 2: représentation de la suppression d'un élément d'une pile.

1 Voici l'état de la liste avant l'appel de la fonction
2 Le pointeur *p_l* représente l'argument de la fonction et donc l'élément à supprimer
3 Sauvegarde de l'élément précédent celui à supprimer grâce au pointeur *p_p*
4 Libération de la mémoire
5 On met le pointeur *next* de *p_p* à NULL
6 Même remarque que pour l'ajout d'un élément : ne pas oublier de se placer au sommet de la pile

```
   void *stack_pop (stack_s ** pp_stack)
   {
      void *ret = NULL;

      if (pp_stack != NULL && *pp_stack != NULL)
      {
   /* (2) */
         stack_s *p_l = *pp_stack;
   /* (3) */
         stack_s *p_p = p_l->prev;

         if (p_p != NULL)
            p_p->next = NULL;
         ret = p_l->data;
   /* (4) */
         free (p_l);
```

```
        p_l = NULL;
/* (5) */
        *pp_stack = p_p;
    }
    return (ret);
}
```

III-E - Suppression de la pile

Pour supprimer la liste, il suffit de dépiler tous ses éléments.

```
void stack_delete (stack_s ** pp_stack)
{
    if (pp_stack != NULL && *pp_stack != NULL)
    {
        while (*pp_stack != NULL)
            stack_pop (pp_stack);
    }
    return;
}
```

IV - Conclusion

La création de la bibliothèque de manipulation des piles est grandement simplifiée du fait qu'elle s'inspire des listes doublement chaînées. Il est tout à fait possible d'obtenir le même résultat avec les listes simplement chaînées, je vous laisse le faire en guise d'exercice.

VII - Remerciements

Merci à Gnux pour la relecture attentive de cet article.

VI - Code source complet

Téléchargez l'archive zippée.

pile.h

```
#ifndef H_PILE
#define H_PILE

typedef struct stack stack_s;

stack_s *stack_new (void);
void stack_push (stack_s **, void *);
void *stack_pop (stack_s **);
void stack_delete (stack_s **);

#endif /* not H_PILE */
```

pile.c

```
#include <stdio.h>
#include <stdlib.h>
#include "pile.h"

struct stack
{
   struct stack *prev;
   struct stack *next;
   void *data;
};

stack_s *stack_new (void)
{
   return (NULL);
}

void stack_push (stack_s ** pp_stack, void *data)
{
   if (pp_stack != NULL)
   {
      stack_s *p_p = *pp_stack;
      stack_s *p_l = NULL;

      p_l = malloc (sizeof (*p_l));
      if (p_l != NULL)
      {
         p_l->data = data;
         p_l->next = NULL;
         p_l->prev = p_p;
         if (p_p != NULL)
            p_p->next = p_l;
         *pp_stack = p_l;
      }
      else
      {
         fprintf (stderr, "Memoire insuffisante\n");
         exit (EXIT_FAILURE);
      }
   }
   return;
}

void *stack_pop (stack_s ** pp_stack)
{
   void *ret = NULL;

   if (pp_stack != NULL && *pp_stack != NULL)
   {
      stack_s *p_l = *pp_stack;
      stack_s *p_p = p_l->prev;

      if (p_p != NULL)
         p_p->next = NULL;
      ret = p_l->data;
      free (p_l);
      p_l = NULL;
```

pile.c

```
        *pp_stack = p_p;
     }
     return (ret);
  }

  void stack_delete (stack_s ** pp_stack)
  {
     if (pp_stack != NULL && *pp_stack != NULL)
     {
        while (*pp_stack != NULL)
           stack_pop (pp_stack);
     }
     return;
  }
```