Les piles en C

par Nicolas Joseph

Date de publication : 03/05/2005

Dernière mise à jour : 03/05/2005

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 ajouter : on parle de liste FIFO (**F**isrt **I**n **F**irst **O**ut). Les piles ne sont que des cas particulier de listes chaînées dont les éléments ne peuvent être ajouté et supprimé 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 les retirer

III - En pratique

III-A - La strcuture 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 pile
{
   struct pile *prev;
   struct pile *next;
   void *data;
} pile_s;
```

III-B - Initialisation

Bien sûr, il est tout de même nécessaire d'initialiser le pointeur qui servira de pile :

```
ppile_s *pile_new (void)
{
   return (NULL);
}
```

III-C - Empiler un élément

Les éléments ne peuvent être ajouter 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.

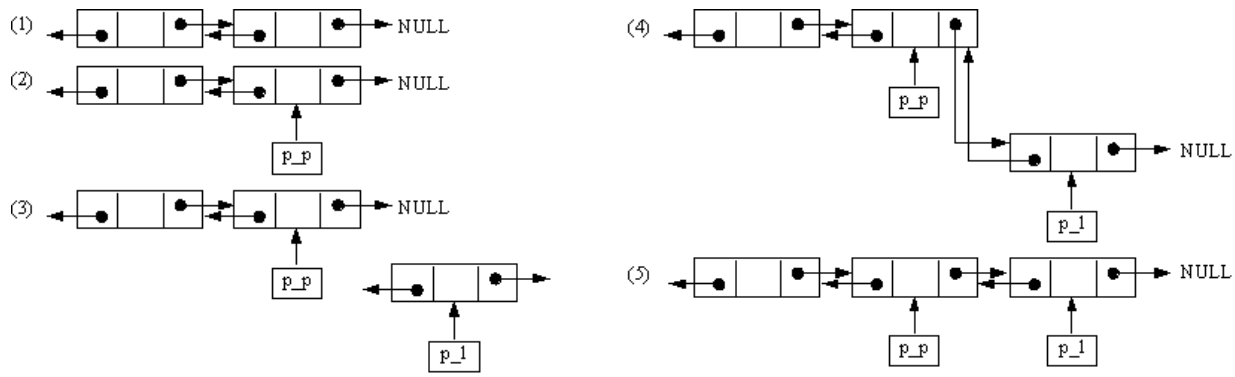

Figure 1: représentation de l'ajout d'un élément à une pile.

```
void pile_push (pile_s ** pp_pile, void *data)
{
   if (pp_pile != NULL)
   {
/* (2) */
      pile_s *p_p = *pp_pile;
      pile_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_pile = p_l;
         }
         else
         {
            fprintf (stderr, "Memoire insuffisante\n");
            exit (EXIT_FAILURE);
         }
      }
      return;
   }
```

III-D - Dépiler un élément

L'élément a 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.

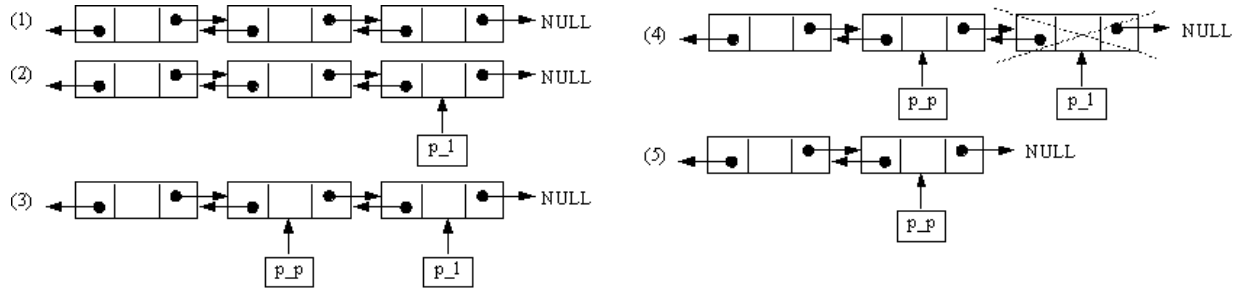

Figure 2: représentation de la suppression d'un élément à une pile.

```
   void *pile_pop (pile_s ** pp_pile)
   {
      void *ret = NULL;

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

         if (p_p != NULL)
            p_p->next = NULL;
         ret = pile_data (p_l);
/* (4) */
         free (p_l);
         p_l = NULL;
/* (5) */
         *pp_pile = p_p;
      }
      return (ret);
   }
```

III-E - Accés à la donnée stockée

```
   void *pile_data (pile_s * p_pile)
   {
      return ((p_pile != NULL) ? p_pile->data : NULL);
   }
```

III-F - Suppression de la pile

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

```
void pile_delete (pile_s ** pp_pile)
{
   if (pp_pile != NULL && *pp_pile != NULL)
   {
      while (*pp_pile != NULL)
         pile_pop (pp_pile);
   }
   return;
}
```

IV - Conclusion

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

V - Remerciements

VII - Code source complet

Télécharger l'archive zippée.

pile.h

```
#ifndef H_PILE
#define H_PILE

#include <stddef.h>             /* pour size_t */

typedef struct pile pile_s;

pile_s *pile_new (void);
void pile_push (pile_s **, void *);
void *pile_pop (pile_s **);
void *pile_data (pile_s *);
void pile_delete (pile_s **);

#endif /* not H_PILE */
```

pile.c

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

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

pile_s *pile_new (void)
{
   return (NULL);
}

void pile_push (pile_s ** pp_pile, void *data)
{
   if (pp_pile != NULL)
   {
      pile_s *p_p = *pp_pile;
      pile_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_pile = p_l;
      }
      else
      {
         fprintf (stderr, "Memoire insuffisante\n");
         exit (EXIT_FAILURE);
      }
   }
   return;
}

void *pile_pop (pile_s ** pp_pile)
{
   void *ret = NULL;

   if (pp_pile != NULL && *pp_pile != NULL)
   {
      pile_s *p_l = *pp_pile;
      pile_s *p_p = p_l->prev;

      if (p_p != NULL)
         p_p->next = NULL;
```

pile.c

```
        ret = pile_data (p_l);
        free (p_l);
        p_l = NULL;
        *pp_pile = p_p;
     }
     return (ret);
  }

  void *pile_data (pile_s * p_pile)
  {
     return ((p_pile != NULL) ? p_pile->data : NULL);
  }

  void pile_delete (pile_s ** pp_pile)
  {
     if (pp_pile != NULL && *pp_pile != NULL)
     {
        while (*pp_pile != NULL)
           pile_pop (pp_pile);
     }
     return;
  }
```