Les files en C

par Nicolas Joseph (home) (Blog)

Date de publication : 10 Aout 2005

Les structures de données en C.
Quatrième partie : les files.

I - Introduction
II - Principe
III - En Pratique
    III-A - La structure d'un élément
    III-B - Initialisation
    III-C - Ajouter un élément
    III-D - Extraire un élément
    III-E - Suppression de la file
IV - Conclusion
V - Remerciements
VI - Code source complet

I - Introduction

Dernière structure de donnée basée sur les listes (enfin !), il s'agit des files. Pour nous faciliter la vie, et ne pas réinventer la roue, nous allons utiliser la bibliothèque de listes doublement chaînées pour créer celle des files.

II - Principe

Les files sont très similaires aux piles, la seule différence se situe au niveau de l'ajout de nouvel élément : il se fait en début de liste. Par contre la suppression se fait toujours en fin de liste. Les files sont aussi appelée liste FIFO (**F**irst **I**n **F**irst **O**ut).

Malgré un fonctionnement proche de celui des piles, la bibliothèque de gestion des files va être plus complexe, en effet, il faut se déplacer au début ou à la fin de la liste suivant que nous ajoutons ou retirons un élément.

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 queue
{
   struct queue *prev;
   struct queue *next;
   void *data;
} queue_s;
```

III-B - Initialisation

On initialise le pointeur qui servira de file :

```
queue_s *queue_new (void)
{
   return (NULL);
}
```

III-C - Ajouter un élément

Les éléments doivent être ajoutés au début, il faut donc insérer le nouvel élément avant le premier de la liste.

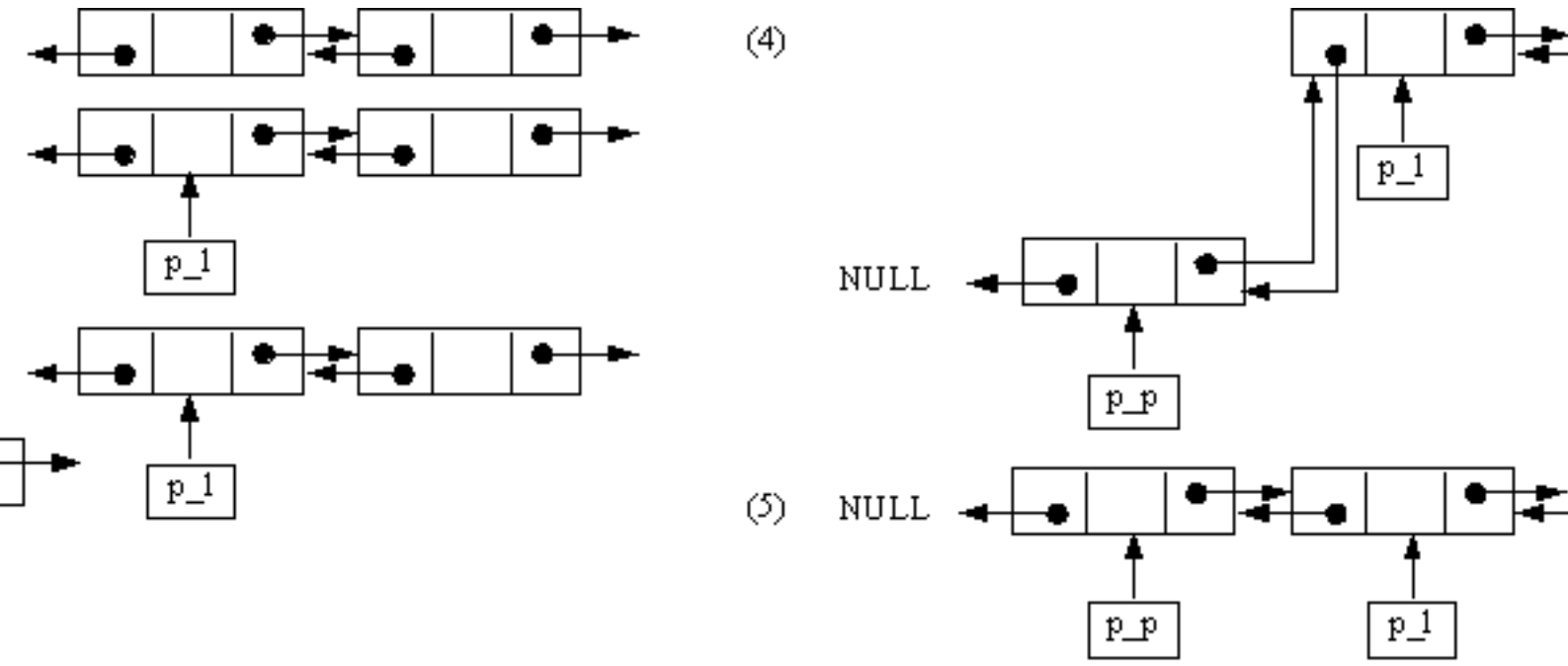

Figure 1 : ajout d'un élément à une file

1 Voici l'état de la file avant l'appel de la fonction
2 On se positionne au début de la file grâce au pointeur *p_l*, c'est donc avant qu'il faut ajouter un élément

3 Création d'un nouvel élément pointé par *p_p*
4 On fait pointer le nouvel élément sur le premier maillon de la file : *p_p* et inversement
5 Etat de la liste après l'appel de la fonction

Et maintenant la même chose mais en C :

```c
void queue_post (queue_s ** pp_queue, void *data)
{
   if (pp_queue != NULL)
   {
      queue_s *p_l = NULL;
      queue_s *p_p = NULL;

      queue_first (pp_queue);
/* (2) */
      p_l = *pp_queue;
/* (3) */
      p_p = malloc (sizeof (*p_p));
      if (p_p != NULL)
      {
         p_p->data = data;
/* (4) */
         p_p->next = p_l;
         p_p->prev = NULL;
         if (p_l != NULL)
            p_l->prev = p_p;
/* (5) */
         *pp_queue = p_p;
      }
      else
      {
         fprintf (stderr, "Memoire insuffisante\n");
         exit (EXIT_FAILURE);
      }
   }
   return;
}
```

Voici le détail de la fonction *queue_first* qui permet de se déplacer au début de la file :

```c
static void queue_first (queue_s ** pp_queue)
{
   if (pp_queue != NULL && *pp_queue != NULL)
   {
      while ((*pp_queue)->prev != NULL)
         queue_prev (pp_queue);
   }
   return;
}

static void queue_prev (queue_s ** pp_queue)
{
   if (pp_queue != NULL && *pp_queue != NULL)
      *pp_queue = (*pp_queue)->prev;
   return;
}
```

*Ces fonctions sont déclarées comme **static** puisqu'elles n'ont plus à être utilisées hors de la bibliothèque.*

III-D - Extraire un élément

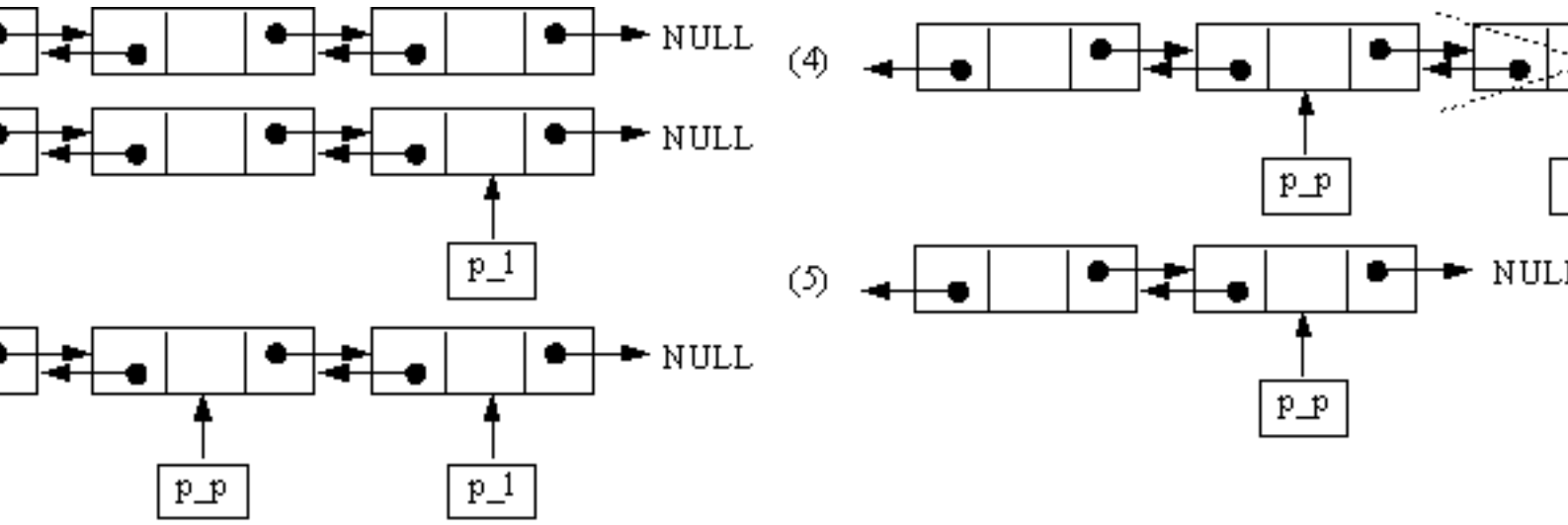

Figure 2 : suppression d'un élément de la file

1 Voici l'état de la file avant l'appel de la fonction
2 On se positionne à la fin de la file grâce au pointeur *p_l*, c'est l'élément à supprimer
3 On sauvegarde le futur dernier élément de la file grâce au pointeur *p_p*
4 Libération de la mémoire
5 On met le pointeur *next* de *p_p* à NULL pour marquer la fin de la file

```
void *queue_get (queue_s ** pp_queue)
{
   void *ret = NULL;

   if (pp_queue != NULL && *pp_queue != NULL)
   {
      queue_s *p_l = NULL;
      queue_s *p_p = NULL;

/* (2) */
      queue_last (pp_queue);
      p_l = *pp_queue;
/* (3) */
      if (p_l != NULL)
         p_p = p_l->prev;
      ret = p_l->data;
/* (4) */
      free (p_l);
      p_l = NULL;
/* (5) */
      if (p_p != NULL)
         p_p->next = NULL;
      *pp_queue = p_p;
   }
   return (ret);
}
```

Et bien sûr le code de la fonction *queue_last* :

```
static void queue_last (queue_s ** pp_queue)
{
   if (pp_queue != NULL && *pp_queue != NULL)
   {
      while ((*pp_queue)->next != NULL)
         queue_next (pp_queue);
   }
```

```
   return;
}

static void queue_next (queue_s ** pp_queue)
{
   if (pp_queue != NULL && *pp_queue != NULL)
      *pp_queue = (*pp_queue)->next;
   return;
}
```

III-E - Suppression de la file

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

```
void queue_delete (queue_s ** pp_queue)
{
   if (pp_queue != NULL && *pp_queue != NULL)
   {
      while (*pp_queue != NULL)
         queue_get (pp_queue);
   }
   return;
}
```

IV - Conclusion

Cet article termine le premier chapitre de cette série. Avant de passer à la seconde grosse partie (les arbres), nous allons faire une petite pause avec l'étude des tables de hachages.

V - Remerciements

Merci à Maximilian pour la relecture attentive de cet article.

VI - Code source complet

Téléchargez l'**archive zippée**.

file.h

```
#ifndef H_FILE
#define H_FILE

typedef struct queue queue_s;

queue_s *queue_new (void);
void queue_post (queue_s **, void *);
void *queue_get (queue_s **);
void queue_delete (queue_s **);

#endif /* not H_FILE */
```

file.c

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

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

static void queue_first (queue_s **);
static void queue_prev (queue_s **);
static void queue_last (queue_s **);
static void queue_next (queue_s **);

queue_s *queue_new (void)
{
   return (NULL);
}

void queue_post (queue_s ** pp_queue, void *data)
{
   if (pp_queue != NULL)
   {
      queue_s *p_l = NULL;
      queue_s *p_p = NULL;

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

file.c

```
   return;
}

void *queue_get (queue_s ** pp_queue)
{
   void *ret = NULL;

   if (pp_queue != NULL && *pp_queue != NULL)
   {
      queue_s *p_l = NULL;
      queue_s *p_p = NULL;

      queue_last (pp_queue);
      p_l = *pp_queue;
      if (p_l != NULL)
         p_p = p_l->prev;
      ret = p_l->data;
      free (p_l);
      p_l = NULL;
      if (p_p != NULL)
         p_p->next = NULL;
      *pp_queue = p_p;
   }
   return (ret);
}

void queue_delete (queue_s ** pp_queue)
{
   if (pp_queue != NULL && *pp_queue != NULL)
   {
      while (*pp_queue != NULL)
         queue_get (pp_queue);
   }
   return;
}

static void queue_first (queue_s ** pp_queue)
{
   if (pp_queue != NULL && *pp_queue != NULL)
   {
      while ((*pp_queue)->prev != NULL)
         queue_prev (pp_queue);
   }
   return;
}

static void queue_prev (queue_s ** pp_queue)
{
   if (pp_queue != NULL && *pp_queue != NULL)
      *pp_queue = (*pp_queue)->prev;
   return;
}

static void queue_last (queue_s ** pp_queue)
{
   if (pp_queue != NULL && *pp_queue != NULL)
   {
      while ((*pp_queue)->next != NULL)
         queue_next (pp_queue);
   }
   return;
}

static void queue_next (queue_s ** pp_queue)
{
   if (pp_queue != NULL && *pp_queue != NULL)
      *pp_queue = (*pp_queue)->next;
```

file.c
```
        return;
    }
```