```c
#include <stdio.h>
int main (void)
{
  printf ("Hello world!\n");
  return 0;
}
```

Langage C

École d'ingénieurs Sup Galilée
Formation Télécom et Réseaux – 1ère année
Université Paris Nord

Annexe A

Complément sur les listes

On reprend les définitions du cours sur les listes chaînées. On va donner l'exemple ici de la gestion d'une liste doublement chaînée, que l'on peut adapter facilement au cas d'une liste simplement chaînée.

A.1 Pré-requis

On suppose que l'on a défini préalablement le type `DATA` qui représente les données que l'on souhaite stocker. De plus on dispose des fonctions suivantes :

```
void printData (DATA data);
void saveData (FILE *file, DATA data);
DATA loadData (char *str);
void copyData (DATA *dest, DATA src);
```

Ces fonctions permettent de réaliser les tâches suivantes :

- `printData` : affiche le contenu de l'objet `data` sur la console,
- `saveData` : écrit le contenu de l'objet `data` dans le fichier `file` (une ligne au format ASCII / CSV),
- `loadData` : convertit la chaîne de caractères `str` en un objet de type `DATA`,
- `copyData` : copie le contenu de l'objet `src` dans l'objet `dest`.

Il est impératif de bien isoler la gestion de la liste de la gestion des données !

A.2 Rappels

La liste en elle-même est implémentée par la structure `NODE` :

```
typedef struct _NODE
{
  DATA data;
  struct _NODE *next;
  struct _NODE *prev;
} NODE;
```

et la structure `LIST` :

```
typedef struct
{
  NODE *head;
  NODE *tail;
} LIST;
```

Il faut se représenter un objet de type `LIST` comme le schéma présenté dans la figure A.1.

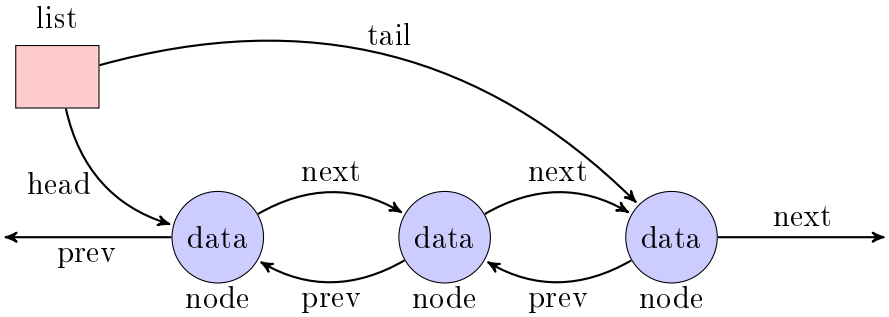

FIGURE A.1 – Représentation visuelle d'une liste doublement chaînée comportant trois éléments. Les flèches symbolisent les pointeurs.

Pour définir une nouvelle liste vide, on peut utiliser l'instruction suivante :

```
LIST mylist = {NULL, NULL};
```

qui déclare la liste `mylist`, où les pointeurs `head` et `tail` sont initialisés à la valeur `NULL`.

A.3 Manipulation de liste

Afin de manipuler la liste on va définir une interface de gestion de liste, composée des fonctions suivantes, dont on détaillera le fonctionnement en *pseudo-code* :

```
void appendToList (LIST *list, DATA data);
void prependToList (LIST *list, DATA data);
DATA removeHead (LIST *list);
DATA removeTail (LIST *list);
void printList (LIST list);
```

Ces fonctions permettent de réaliser les tâches suivantes :

- `appendToList` : ajoute un élément contenant l'objet `data` à la fin de la liste `list`,
- `prependToList` : ajoute un élément contenant l'objet `data` au début de la liste `list`,
- `removeHead` : enlève le premier élément de la liste et retourne son contenu,
- `removeTail` : enlève le dernier élément de la liste et retourne son contenu,
- `printList` : affiche le contenu de la liste (c'est à dire de chaque élément de la liste).

Ces cinq fonctions fournissent une base permettant de manipuler les listes doublement chaînées.

Dans la suite on propose un pseudo-code pour chacune de ces fonctions. Pour obtenir le code en C il suffit de remplacer chaque ligne du pseudo-code par une instruction bien choisie, éventuellement en appelant une des fonctions définies dans la partie A.1, ou bien une fonction de la bibliothèque standard.

On laisse en exercice l'écriture d'un pseudo-code (et du code en langage C) pour les fonctions :

```
LIST loadList (FILE *file);
```

qui charge une liste à partir d'un fichier `file`, comportant des données au format ASCII / CSV et :

```
void saveList (FILE *file, LIST list);
```

qui sauvegarde le contenu de la liste `list` dans le fichier `file`, au format ASCII / CSV.

A.3.1 Pseudo-code `appendToList`

```
void appendToList (LIST *list, DATA data);
```

1. on déclare un pointeur `node` vers un objet de type `NODE`,
2. on alloue dynamiquement la mémoire de l'objet `node` et on l'initialise à 0,
3. on copie le contenu du paramètre `data` dans le champ `data` de `node`,
4. si le dernier élément de `list` existe :
   (a) on affecte `node` au suivant du dernier élément de `list`,
   (b) on affecte le dernier élément de `list` comme précédent de `node`,
   (c) on affecte `node` au dernier élément de `list`,
5. sinon on affecte `node` au premier et au dernier élément de `list`.

Le dernier élément de la liste `list` contient maintenant les données de l'objet `data` passé en paramètre.

A.3.2 Pseudo-code `prependToList`

```
void prependToList (LIST *list, DATA data);
```

1. on déclare un pointeur `node` vers un objet de type `NODE`,
2. on alloue dynamiquement la mémoire de l'objet `node` et on l'initialise à 0,
3. on copie le contenu du paramètre `data` dans le champ `data` de `node`,
4. si le premier élément de `list` existe :
   (a) on affecte `node` au précédent du premier élément de `list`,
   (b) on affecte le premier élément de `list` comme suivant de `node`,
   (c) on affecte `node` au premier élément de `list`,
5. sinon on affecte `node` au premier et au dernier élément de `list`.

Le premier élément de la liste `list` contient maintenant les données de l'objet `data` passé en paramètre.

A.3.3 Pseudo-code `removeHead`

```
DATA removeHead (LIST *list);
```

1. on déclare un objet de `data` de type `DATA`, initialisé à 0,
2. si la liste `list` est vide, on affiche une erreur,
3. sinon :
   (a) on déclare un pointeur `node` vers un objet de type `NODE`,
   (b) on affecte le premier élément de `list` à `node`,
   (c) on copie le contenu du champ `data` de `node` dans l'objet `data`,
   (d) on affecte le suivant de `node` au premier élément de `list`,
   (e) si le premier élément de `list` existe, on affecte la valeur `NULL` à son précédent,
   (f) sinon la liste est vide et on affecte la valeur `NULL` au premier et au dernier élément de `list`.
   (g) on libère la mémoire allouée au pointeur `node`,
4. on retourne `data`.

A.3.4 Pseudo-code `removeTail`

```
DATA removeTail (LIST *list);
```

1. on déclare un objet de `data` de type `DATA`, initialisé à 0,
2. si la liste `list` est vide, on affiche une erreur,
3. sinon :
   (a) on déclare un pointeur `node` vers un objet de type `NODE`,
   (b) on affecte le dernier élément de `list` à `node`,
   (c) on copie le contenu du champ `data` de `node` dans l'objet `data`,
   (d) on affecte le précédent de `node` au dernier élément de `list`,
   (e) si le dernier élément de `list` existe, on affecte la valeur `NULL` à son suivant,
   (f) sinon la liste est vide et on affecte la valeur `NULL` au premier et au dernier élément de `list`.
   (g) on libère la mémoire allouée au pointeur `node`,
4. on retourne `data`.

A.3.5 Pseudo-code `printList`

```
void printList (LIST list);
```

1. si la liste `list` est vide on affiche un avertissement,
2. sinon :
   (a) on déclare un pointeur `node` vers un objet de type `NODE`,
   (b) on affecte le premier élément de `list` à `node`,
   (c) tant que `node` est différent du dernier élément de `list`,
      i. on affiche le contenu du champ `data` de `node`,
      ii. on affiche un séparateur (`"->"` par exemple),
      iii. on affecte le suivant de `node` à `node`,
   (d) on affiche le contenu du champ `data` du dernier élément de `list`,
3. on affiche un retour à la ligne.