2.1 Listes

Sauf si autrement spécifié, les exercices suivants représentent une liste comme un ensemble de noeuds: chaque noeud x (sauf le noeud terminal x=null) contient les variables x.next (prochain élément) et x.data (contenu: élément stocké).

2.1.1 Échange d'éléments

Sauf si autrement spécifié, les exercices suivants représentent une liste comme un ensemble de noeuds: chaque noeud x (sauf le noeud terminal x=null) contient les variables x.next (prochain élément) et x.data (contenu: élément stocké).

- Développer le code pour exchange(n) qui échange deux noeuds suivant n.
- Développer le code pour exchangeData(n) qui échange le contenu des deux noeuds suivant n.

2.1.1 Échange d'éléments

Solution

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

typedef struct _node node;
struct _node {
    int data;
    node *next;
};
typedef struct _list list;
struct _list {
    node *first;
};
// l = a->b->c->d
int exchange(node *n){
    if (n->next == NULL )
        return 0;
    if (n->next->next == NULL)
        return 0;
    node *tmp = n->next->next; //tmp = c, l = a->b->c->d
    n->next->next = tmp->next; //b-> d, l = a->b->d
    tmp->next = n->next;       //c-> b, l = a->b->d
    n->next = tmp;             //a-> c, l = a->c->b->d
    return 1;
}
```

2.1.1 Échange d'éléments

Data Structures

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

typedef struct _node node;
struct _node {
    int data;
    node *next;
};
typedef struct _list list;
struct _list {
    node *first;
};
```

Solution

```c
int exchangeData(node *n){
    if (n->next == NULL )
        return 0;
    if (n->next->next == NULL)
        return 0;
    int tmp = n->next->next->data;
    n->next->next->data =  n->next->data;
    n->next->data    = tmp;
    return 1;
}
```

2.1.2 Inversion de liste

Proposer des algorithmes pour renverser une liste chaînée par itération (en un seul parcours), ou par récursion.

2.1.2 Inversion de liste

Solution

```
node* pop(list *l){
    node *tmp = l->first;
    if (tmp)
        l->first = tmp->next;
    return tmp;
}
void push(list *l,node *n){
    n->next = l->first;
    l->first = n;
}

void reverseit(list *l){
    list newlist= { .first = NULL};
    node *temp;
    while((temp = pop(l))){
        push(&newlist,temp);
    }
    l->first = newlist.first;
}
```

2.1.2 Inversion de liste

Solution

```
list* reverseRecursive(list *l, list *newlist){
    if (newlist == NULL)
        exit(1);
    if (l->first == NULL){
        l->first = newlist->first;
        return l;
    }
    push(newlist,pop(l));
    return reverseRecursive(l, newlist);
}
list* reverseR(list *l){
    list newlist =  { .first = NULL };
    return reverseRecursive(l, &newlist);
}
```

2.1.3 Liste circulaire

- Montrer comment implémenter les méthodes de l'interface file FIFO avec une liste circulaire (on maintient une référence au dernier noeud sur la liste).
- Montrer comment faire la concaténation de deux listes circulaires.

2.1.3 Liste circulaire

Solution

```
typedef struct _node node;
struct _node {
    int data;
    node *next;
};
typedef struct _listCirc listCirc;
struct _listCirc {
    node *first;
    node *last;
};

node* removeFifo(listCirc *l){
    node *tmp = l->first;
    if (tmp) {
        l->first = tmp->next;
        l->last->next = l->first;
    }
    return tmp;
}
void insertFifo(listCirc *l,node *n){
    n->next = l->first;
    l->first = n;
    l->last->next = l->first;
}
```

2.1.3 Liste circulaire

Data Structures

```
typedef struct _node node;
struct _node {
    int data;
    node *next;
};
typedef struct _listCirc listCirc;
struct _listCirc {
    node *first;
    node *last;
};
```

Solution

```
listCirc* concatenateCirc(listCirc *l0,listCirc *l1){
    l0->last->next = l1->first;
    l1->last->next = l0->first;
    return l0;
}
```