Algorithmique et Structures de données

Mouhamadou GAYE

Enseignant-Chercheur
Département d'Informatique
Licence 2 en Ingénierie Informatique

3 juillet 2022

Chapitre 2 : Structures de données linéaires

Introduction

- Une **structure de données** spécifie la façon de représenter des données en mémoire en décrivant l'attribution de la mémoire et les méthodes d'accès aux données. Une structure de données peut être :
  - **statique** lorsque l'espace mémoire attribué est fixée au moment de la création de la structure ;
  - **dynamique** lorsque l'espace mémoire attribué peut varier au cours du temps ;
  - **mutable** lorsque le contenu de la structure est modifiable.

Tableaux

Définition

Un tableau est objet composé des éléments de même type stockés de manière contiguë en mémoire. Chaque élément du tableau est repéré dans l'ensemble par un numéro appelé indice. Si le tableau est multidimensionnel, il s'agira d'un couple d'indices.

- **Déclaration :**

  type nom_tableau [TAILLE] ; //pour un tableau à une dimension

  type nom_tableau [NOMBRE_LIGNE][NOMBRE_COLONNE] ; //pour un tableau à deux dimensions

Tableaux

- **Accès aux éléments :**
  nom_tableau[i] ou *(nom_tableau + i) désigne l'élément à l'indice i et nom_tableau [i][j] l'élément aux indices (i, j) s'il s'agit d'une matrice.
  Les indices commencent à 0.
- **Initialisation à la déclaration :**
  type nom_tableau [N] = {val_1, val_2, ..., val_N} ;
  ou
  type nom_tableau [] = {val_1, val_2, ..., val_N} ;
  ou lorsqu'on veut initialiser quelques éléments
  type nom_tableau [] = { [i] = val_i, [j] = val_j} ;

Tableaux

- **Mininum / Maximun :**

```
type minimum(type t[], int n){
    type min = t[0] ;
    for(i = 0 ; i< n, i++){
        if (t[i] < min){
            min = t[i] ;
        }
    }
    return min ;
}
```

- **Recherche séquentielle d'un élément :**

```
type recherche(type t[], int n, type elt){
    int i = 0 ;
    while(i< n && t[i] != elt{
        i++ ;
    }
    return i< n ;
}
```

Tableaux

- **Insertion d'un élément dans un tableau trié :**

```
void insertion(type t[], int n, type elt){
    int i = 0, j ;
    while(i< n && elt < t[i]{
        i++ ;
    }
    if(i < n){
        for(j = n - 1 ; j > i ; i- -){
            t[j] = t[j - 1] ;
        }
        t[i] = elt ;
    }
}
```

Tableaux

- **Suppression d'un élément dans un tableau :**

```
int suppression(type t[], int n, type elt){
    int i = 0, j ;
    while(i < n && elt != t[i]{
        i++ ;
    }
    if(i < n){
        for( ; i < n - 1 ; i++){
            t[i] = t[i + 1] ;
        }
        n- - ;
    }
    return n ;
}
```

Listes chainées

Définition

Une liste chainée est une structure contenant des éléments de même type appelés aussi maillons ou cellules dans laquelle chaque élément contient :

- des informations caractéristiques de l'application (caractéristiques d'une personne par exemple)
- l'adresse de l'élément suivant et / ou précédent ou une marque de fin (pointeur NULL).

Une liste peut être simplement chainée, doublement chainée ou circulaire.

Listes simplement chainées

- **Déclaration**
  1. On déclare d'abord le maillon qui est une structure

```
typedef struct maillon{
    type info ;
    struct maillon *suivant ;
}Maillon ;
```

  2. Puis la liste qui est un pointeur sur le premier maillon.

```
Maillon *L ;
```

L'allocation de la mémoire se fait ici de façon dynamique avec la fonction malloc.

Listes simplement chainées

- **Ajout d'un élément en tête de liste**

```
Maillon * ajoutTete(Maillon *L, type elt){
    Maillon *nouveau = (Maillon*)malloc(sizeof(Maillon)) ;
    if(nouveau){
        nouveau->info = elt ;
        nouveau->suivant = L ;
    }
    return nouveau ;
}
```

Listes simplement chainées

- **Ajout d'un élément en fin de liste**

```
Maillon * ajoutFin(Maillon *L, type x){
    Maillon *pc, *nouveau = (Maillon*)malloc(sizeof(Maillon));
    if(nouveau != NULL){
        nouveau->info = x;
        nouveau->suivant = NULL;
    }
    if(L != NULL){
        pc = L;
        while(pc->suivant != NULL){
            pc = pc->suivant;
        }
        pc->suivant = nouveau;
    } else L = nouveau;
    return L;
}
```

Listes simplement chainées

- **Ajout d'un élément à une position donnée de la liste**

```
Maillon * ajoutApresElement(Maillon *L, type x, type elt){
    Maillon *pc, *nouveau = (Maillon*)malloc(sizeof(Maillon));
    if(nouveau != NULL){
        nouveau->info = x;
    }
    if(L != NULL){
        pc = L;
        while(pc->suivant != NULL && pc->info != elt){
            pc = pc->suivant;
        }
        nouveau->suivant = pc->suivant;
        pc->suivant = nouveau;
    }
    return L;}
```

Listes simplement chainées

- **Suppression d'un élément en tête de liste**

```
Maillon * suppressionTete(Maillon *L){
    Maillon *pc = L ;
    if(L != NULL){
        L = L->suivant ;
        free(pc) ;
    }
    return L ;
}
```

Listes simplement chainées

- **Suppression d'un élément en fin de liste**

```
Maillon * suppressionFin(Maillon *L){
    Maillon *pr, *pc ;
    if(L == NULL)
        return NULL ;
    if(L ->suivant == NULL){
        free(L) ; return NULL ;
    }
    pr = L ; pc = L->suivant ;
    while(pc->suivant != NULL){
        pr = pc ; pc = pc->suivant ;
    }
    pr->suivant = NULL ;
    free(pc) ;
    return L ;
}
```

Listes simplement chainées

- **Suppression d'un élément à une position donnée de la liste**

```
Maillon * suppressionElement(Maillon *L, type elt){
    Maillon *pr = *pc = L ;
    while(pc != NULL && pc->info != elt){
        pr = pc ;
        pc = pc->suivant ;
    }
    if(pc != NULL){
        pr->suivant = pc->suivant ;
        free(pc) ;
    }
    return L ;
}
```

Listes doublement chainées

- **Déclaration**

```
typedef struct maillon{
    type info ;
    struct maillon *precedent ;
    struct maillon *suivant ;
}Maillon ;
```

Listes doublement chainées

- **Ajout d'un élément en tête de liste**

```
Maillon * ajoutTete(Maillon *L, type elt){
    Maillon *nouveau = (Maillon*)malloc(sizeof(Maillon)) ;
    if(nouveau){
        nouveau->info = elt ;
        nouveau->precedent = NULL ;
        L->precedent = nouveau ;
        nouveau->suivant = L ;
    }
    return nouveau ;
}
```

Listes doublement chainées

- **Ajout d'un élément en fin de liste**

```
Maillon * ajoutFin(Maillon *L, type x){
    Maillon *pc, *nouveau = (Maillon*)malloc(sizeof(Maillon)) ;
    if(nouveau != NULL){
        nouveau->info = x ;
        nouveau->suivant = NULL ;
    }
    if(L != NULL){
        pc = L ;
        while(pc->suivant != NULL){
            pc = pc->suivant ;
        }
        pc->suivant = nouveau ;
        nouveau->precedent = pc ;
    } else {nouveau->precedent = NULL ; L = nouveau ;}
    return L ;}
```

Listes doublement chainées

- **Ajout d'un élément à une position donnée de la liste**

```
Maillon * ajoutApresElement(Maillon *L, type x, type elt){
    Maillon *pc, *nouveau = (Maillon*)malloc(sizeof(Maillon));
    if(nouveau != NULL){
        nouveau->info = x;
    }
    if(L != NULL){
        pc = L;
        while(pc->suivant != NULL && pc->info != elt){
            pc = pc->suivant;
        }
        nouveau->suivant = pc->suivant;
        nouveau->precedent = pc;
        pc->suivant->precedent = nouveau;
        pc->suivant = nouveau;
    }return L;}
```

Listes doublement chainées

- **Suppression d'un élément en tête de liste**

```
Maillon * suppressionTete(Maillon *L){
    Maillon *pc = L ;
    if(L != NULL){
        L = L->suivant ;
        L->precedent = NULL ;
        free(pc) ;
    }
    return L ;
}
```

Listes doublement chainées

- **Suppression d'un élément en fin de liste**

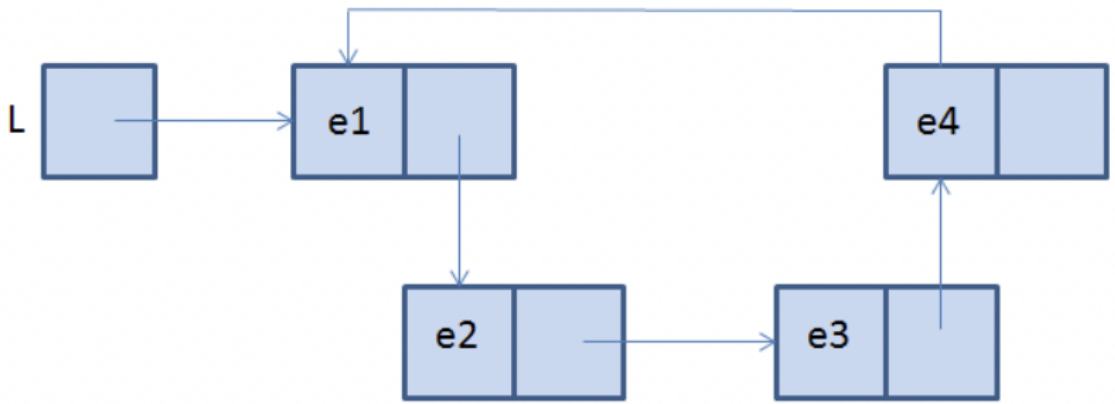
```
Maillon * suppressionFin(Maillon *L){
    Maillon *pr, *pc ;
    if(L == NULL)
        return NULL ;
    if(L ->suivant == NULL){
        free(L) ; return NULL ;
    }
    pr = L ; pc = L->suivant ;
    while(pc->suivant != NULL){
        pr = pc ; pc = pc->suivant ;
    }
    pr->suivant = NULL ;
    free(pc) ;
    return L ;
}
```

Listes doublement chainées

- **Suppression d'un élément à une position donnée de la liste**

```
Maillon * suppressionElement(Maillon *L, type elt){
    Maillon *pr = *pc = L ;
    if(L != NULL){
        while(pc != NULL && pc->info != elt){
            pr = pc ;
            pc = pc->suivant ;
        }
        if(pc != NULL){
            pr->suivant = pc->suivant ;
            if(pc->suivant != NULL)
                pc->suivant->precedent = pr ;
            free(pc) ;
        }
    }
    return L ;}
```

Listes circulaires



liste chaînée circulaire par un pointeur