Université Ibn Tofaïl
Faculté des Sciences
Kénitra

A.U. 2021/2022

Programmation II

Licence SMI - S4

Pr. El B. AMEUR

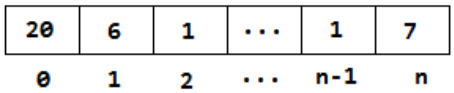
Plan du cours

Partie 2 : Implémentation des Types de Données Abstraits en C

1 LES LISTES CHAINÉES (*LINKED LIST*)

1.1 Généralités

- Les éléments d'un tableau sont placés de façon adjacent en mémoire.



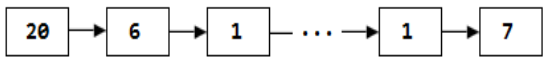
- Pour créer un tableau, il faut connaître sa taille.
- Si vous voulez supprimer un élément au milieu du tableau,
  - il vous faut recopier les éléments temporairement,
  - réallouer de la mémoire pour le tableau,
  - puis le remplir à partir de l'élément supprimé.

➔Bref, beaucoup de manipulations coûteuses en ressources.

5

1.1 Généralités

- Une liste chaînée est différente dans le sens où les éléments sont répartis dans la mémoire et reliés entre eux par des liens (pointeurs).



- L'ajout et la suppression des éléments d'une liste chaînée peut être à n'importe quel endroit et à n'importe quel instant, sans devoir recréer la liste entière.

6

1.1 Généralités

- Une liste chaînée donc est un ensemble de n éléments notée

  $L = e_1, e_{2, \ldots,} e_n$
  - Ou e_1 est le premier élément, e_2 le deuxième, etc...
  - Lorsque n=0 on dit que la liste est vide.
- Les listes servent à gérer un ensemble de données, un peu comme les tableaux :
  - Les listes sont cependant plus efficaces pour réaliser des opérations comme l'insertion et la suppression d'éléments.
  - Les listes utilisent par ailleurs l'allocation dynamique de mémoire et peuvent avoir une taille qui varie pendant l'exécution.

7

1.1 Généralités

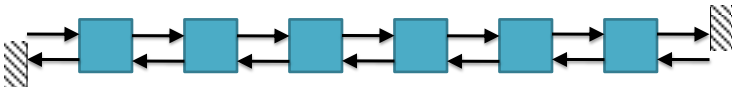
- **Remarque :**
  - Un tableau peut aussi être défini dynamiquement mais pour modifier sa taille, il faut en créer un nouveau, transférer les données puis supprimer l'ancien.
  - L'allocation (ou la libération) se fait élément par élément.

- Les opérations sur une liste peuvent être:
  - Créer une liste
  - Supprimer une liste
  - Rechercher un élément particulier
  - Insérer un élément (en début, en fin ou au milieu)
  - Supprimer un élément particulier
  - Permuter deux éléments
  - Concaténer deux listes
  - ...

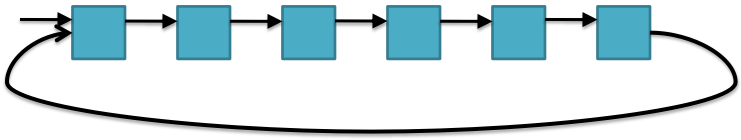
1.2 Types de listes chainées

- Les listes peuvent être :
  - **Simplement chaînées**

- **Doublement chaînées**



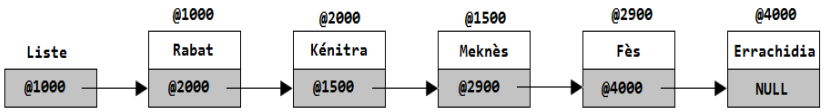
- **Circulaires (chaînage simple ou double)**



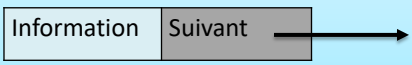
9

1.3.1 Listes simplement chaînées

- Une liste simplement chaînée est composée d'éléments distincts liés par un simple pointeur.



- Chaque élément (**Noeud**) d'une liste simplement chaînée est formé de deux parties:
  - **un champ (ou plusieurs champs) :** contenant la **donnée** (ou un pointeur vers celle-ci),
  - **un pointeur :** vers l'élément **suivant** de la liste.



Données **Pointeur**

10

1.3.1 Listes simplement chaînées

- On implémente un élément d'une liste simple contenant une donnée de type **caractère** (**char**) sous forme d'une structure **C**:

```
typedef char Type;
```

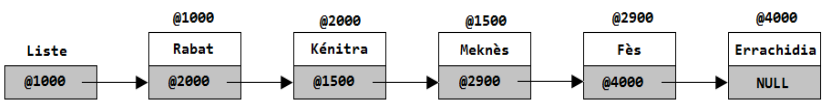
```
typedef struct Noeud * Liste;
```

```
typedef struct Noeud{
    Type info;
    Liste suivant;
} Noeud;
```

11

1.3.1 Listes simplement chaînées

- Le premier élément d'une liste est sa tête
- le dernier élément d'une liste est sa queue.
- Le pointeur du dernier élément est initialisé à la valeur **NULL** en **C**.



12

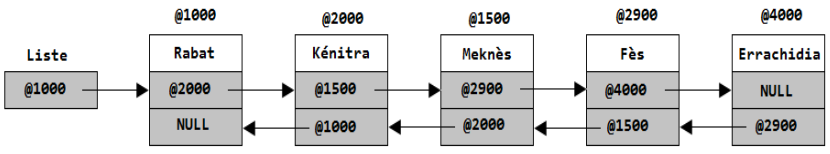
1.3.1 Listes simplement chaînées

- Pour accéder à un élément d'une liste simplement chaînée, on part de la tête et on passe d'un élément à l'autre à l'aide du pointeur **suivant** associé à chaque élément.
- En pratique, les éléments étant créés par allocation dynamique.
  - ➔Ne sont pas adjacents en mémoire contrairement à un tableau.
- La suppression d'un élément sans précaution ne permet plus d'accéder aux éléments suivants.
- D'autre part, une liste simplement chaînée ne peut être parcourue que dans un sens (de la tête vers la queue).

13

1.3.2 Listes doublement chaînées

- Les listes doublement chaînées sont constituées d'éléments comportant trois champs:
  - **un champ :** contenant l'information (**donnée** ou **pointeur** vers celle-ci).
  - **un pointeur :** vers l'élément **suivant** de la liste.
  - **un pointeur :** vers l'élément **précèdent** de la liste.
- Elles peuvent donc être parcourues dans les deux sens.



14

1.3.2 Listes doublement chaînées

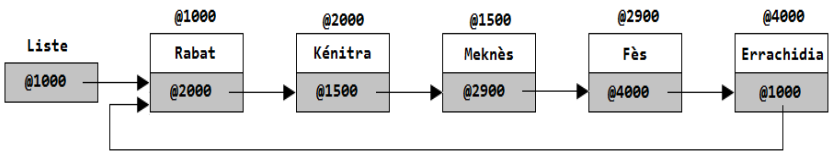
- En **C**, on implémente un élément d'une liste doublement chainée contenant une donnée entière sous forme d'une structure :

```
typedef int Type;
typedef struct Noeud * Liste;
typedef struct Noeud {
    Type info;
    Liste suivant;
    Liste precedent;
}Noeud;
```

15

1.3.3 Listes circulaires

- Une liste circulaire peut être simplement ou doublement chaînée.
- Sa particularité est de ne pas comporter de queue.
- Le dernier élément de la liste pointe vers le premier.
- Un élément possède donc toujours un suivant.



16

1.4 Opérations sur une liste simplement chainée

- La liste chaînée de caractères utilisée sera vide au départ.
- L'exemple suivant défini le type abstrait **Noeud** permettant de créer le type **Liste** représentant une liste chaînée de **caractères** (**char**).

```
typedef char Type;
```

```
typedef struct Noeud * Liste;
```

```
typedef struct Noeud{
    Type info;
    Liste suivant;
}Noeud;
```

17

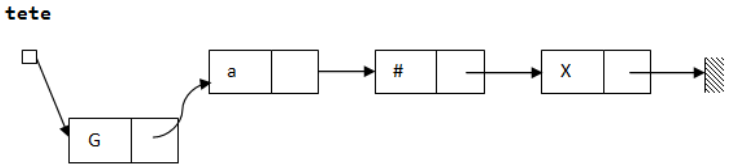
1.4.1 Insertion d'un élément

- Pour insérer un élément dans une liste chaînée, il faut savoir où l'insérer.
- Les trois insertions possibles dans une liste chaînée sont :
  - en tête de liste
  - en fin de liste
  - à une position donnée

18

1.4.1.1 Insertion de nœuds en tête de liste chaînée

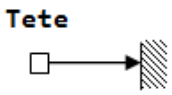
- L'insertion en tête, se fait en créant un élément, lui assigner la valeur que l'on veut insérer, puis pour terminer, raccorder cet élément à la liste.
- Lors d'une insertion en tête, on devra donc assigner au **suivant** l'adresse du premier élément de la liste.



19

1.4.1.1 Insertion de nœuds en tête de liste chaînée

- Illustrons un chaînage en tête : au départ, la liste chaînée est vide, et on suppose que l'on saisit, dans l'ordre, les caractères 'X', '#' et 'a'.
  - Initialisation de la liste à **NULL** :

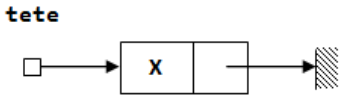



```
// ...
main(){
    Liste tete;
    tete = NULL;
}
```

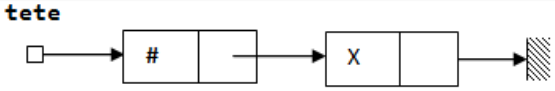
20

1.4.1.1 Insertion de nœuds en tête de liste chaînée

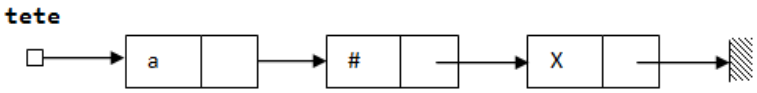
– Ajout de 'X' :



- Ajout de '#' :



- Ajout de 'a' :



21

1.4.1.1 Insertion de nœuds en tête de liste chaînée

- Le programme suivant implémente les différentes étapes illustrées :

```
// ...
main(){
    // ...
    // Création du noeud contenant 'X'
    Noeud * nouveau = (Noeud*)malloc(sizeof(Noeud));
    nouveau->info = 'X';
    nouveau->suivant = tete;
    tete = nouveau;
    // Création du noeud contenant '#'
    nouveau = (Noeud*)malloc(sizeof(Noeud));
    nouveau->info = '#';
    nouveau->suivant = tete;
    tete = nouveau;
    // Création du noeud contenant 'a'
    nouveau = (Noeud*)malloc(sizeof(Noeud));
    nouveau->info = 'a';
    nouveau->suivant = tete;
    tete = nouveau;
}
```

22

1.4.1.1 Insertion de nœuds en tête de liste chaînée

- Pour afficher le contenu de la liste, on doit parcourir la liste avec un pointeur, pour ne pas perdre la tête de la liste, jusqu'au bout et afficher toutes les valeurs qu'elle contient :

```
// ...
main(){
    // ...
    Liste tmp = tete;
    while(tmp != NULL){
        printf("%c ", tmp->info);
        tmp = tmp->suivant;
    }
}
```

```
a # X
```

23

1.4.1.1 Insertion de nœuds en tête de liste chaînée

- A ce niveau, on a tout intérêt d'améliorer notre programme en créant des fonctions réalisant les opérations sur la liste simplement chainée crée:
  - **initialiserListe :** fonction d'initialisation de la liste
  - **insererEnTete :** fonction d'insertion en tête
  - **afficherListe :** fonction d'affichage de la liste

24

1.4.1.1 Insertion de nœuds en tête de liste chaînée

- La définition des fonction est la suivante :

```
void initialiserListe(Liste * adrListe){
    (*adrListe) = NULL ;
}
```

```
void insererEnTete(Liste * adrListe, Type valElement){
    Noeud * nouveau ;
    nouveau = (Noeud*)malloc(sizeof(Noeud));
    if(nouveau != NULL ){
        nouveau->info = valElement;
        nouveau->suivant = (*adrListe);
        (*adrListe) = nouveau;
    }
}
```

```
void afficherListe(Liste L){
    while(L!=NULL){
        printf("%c ", L>info);
        L= L>suivant;
    }
    printf("\n");
}
```

25

1.4.1.1 Insertion de nœuds en tête de liste chaînée

- Le programme **main** sera ainsi :

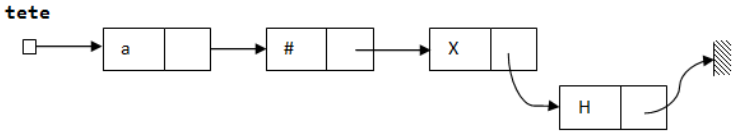
```
main(){
    Liste tete;
    initialiserListe(&tete);
    insererEnTete(&tete, 'X');
    insererEnTete(&tete, '#');
    insererEnTete(&tete, 'a');
    afficherListe(tete);
}
```

```
a # X
```

26

1.4.1.2 Insertion de nœuds en fin de la liste chaînée

- L'insertion en fin de liste est comme suit:
  - Créer un nouvel élément, lui assigner sa valeur, et mettre l'adresse de l'élément suivant à **NULL**.
  - Faire pointer le dernier élément de liste originale sur le nouvel élément que nous venons de créer
    - Créer un pointeur temporaire sur l'élément qui va se déplacer d'élément en élément, et regarder si cet élément est le dernier de la liste.
    - Un élément sera forcément le dernier de la liste si **NULL** est assigné à son champ suivant.



27

1.4.1.2 Insertion de nœuds en fin de la liste chaînée

- Le code **C** de la fonction d'insertion en fin est le suivant :

```
void insererEnQueue(Liste * adrListe, Type valElement){
    Noeud * nouveau = (Noeud*)malloc(sizeof(Noeud));
    nouveau->info = valElement;
    nouveau->suivant = NULL;
    if((*adrListe) == NULL){
        (*adrListe) = nouveau;
    }
    else{
        Liste tmp = (*adrListe);
        while(tmp->suivant != NULL){
            tmp = tmp->suivant;
        }
        tmp->suivant = nouveau;
    }
}
```

28

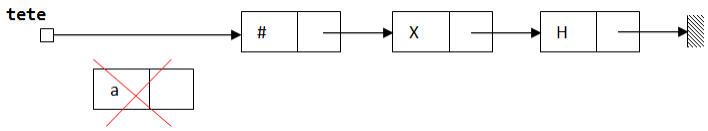
1.4.2 Suppression d'un élément

- Supprimer un élément en tête
- Supprimer un élément en fin de liste

29

1.4.2.1 Supprimer un élément en tête

- Pour supprimer le premier élément de la liste, il faut utiliser la fonction `free`.
- Si la liste n'est pas vide, on stocke l'adresse du premier élément de la liste après suppression (i.e. l'adresse du 2ème élément de la liste originale), on supprime le premier élément, et on renvoie la nouvelle liste.
- Attention quand même à ne pas libérer le premier élément avant d'avoir stocké l'adresse du second, sans quoi il sera impossible de la récupérer.



30

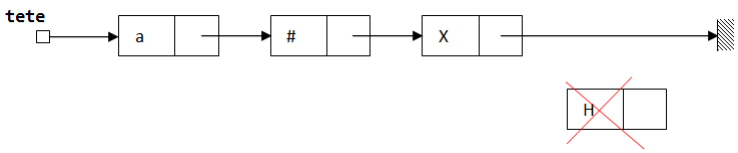
1.4.2.1 Supprimer un élément en tête

```
void supprimerEnTete(Liste * adrListe){
    if((*adrListe) != NULL){
        Liste tmp = (*adrListe)->suivant;
        free(*adrListe);
        (*adrListe) = tmp;
    }
}
```

31

1.4.2.2 Supprimer un élément en fin de liste

- Pour Supprimer un élément en fin de lise:
- il va falloir parcourir la liste jusqu'à son dernier élément,
- indiquer que l'avant-dernier élément va devenir le dernier de la liste
- et libérer le dernier élément pour enfin retourner le pointeur sur le premier élément de la liste d'origine.



32

1.4.2.2 Supprimer un élément en fin de liste

```
void supprimerEnFin(Liste * adrListe){
    if((*adrListe) != NULL){
        if((*adrListe)->suivant == NULL){
            free(*adrListe);
            (*adrListe) = NULL;
        }
        Liste tmp = (*adrListe);
        Liste ptmp = (*adrListe);
        while(tmp->suivant != NULL){
            ptmp = tmp;
            tmp = tmp->suivant;
        }
        ptmp->suivant = NULL;
        free(tmp);
    }
}
```

33

1.4.3 Rechercher un élément dans une liste

- Le but est de renvoyer l'adresse du premier élément trouvé ayant une certaine valeur.
  - Si aucun élément n'est trouvé, on renverra **NULL**.
- L'intérêt est de pouvoir, une fois le premier élément trouvé, chercher la prochaine occurrence en recherchant à partir de **elementTrouve->suivant.**
- On parcourt donc la liste jusqu'au bout, et dès qu'on trouve un élément qui correspond à ce que l'on recherche, on renvoie son adresse.

34

1.4.3 Rechercher un élément dans une liste

```
Noeud * rechercherElement(Liste L, Type valElement){
    Noeud * tmp = L;
    // Tant que l'on n'est pas au bout de la liste
    while(tmp != NULL){
        if(tmp->info == valElement){
            // Si l'élément a la valeur recherchée,
            // on renvoie son adresse
            return tmp;
        }
        tmp = tmp->suivant;
    }
    return NULL;
}
```

35

1.4.4 Compter le nombre d'occurrences d'une valeur

- Nous allons utiliser la fonction précédente permettant de rechercher un élément.
- On cherche une première occurrence : si on la trouve, alors on continue la recherche à partir de l'élément suivant, et ce tant qu'il reste des occurrences de la valeur recherchée.
- Il est aussi possible d'écrire cette fonction sans utiliser la précédente bien entendu, en parcourant l'ensemble de la liste avec un compteur que l'on incrémente à chaque fois que l'on passe sur un élément ayant la valeur recherchée.
- Cette fonction n'est pas beaucoup plus compliquée, mais il est intéressant d'un point de vue algorithmique de réutiliser des fonctions pour simplifier nos codes.

36

1.4.4 Compter le nombre d'occurrences d'une valeur

```
int nombreOccurences(Liste L, Type valElement){
   int i = 0;
   // Si la liste est vide, on renvoie 0
   if(L == NULL)
       return 0;
   // Sinon, tant qu'il y a encore un élément ayant la val = valeur
   while((L= rechercherElement(L, valElement)) != NULL){
       // On incrémente
       L= L>suivant;
       i++;
   }
   // Et on retourne le nombre d'occurrences
   return i;
}
```

37

1.4.5 Compter le nombre d'éléments d'une liste chaîné

- On parcoure la liste de bout en bout et incrémentez un compteur pour chaque nouvel élément trouvé.
- Jusqu'à maintenant, nous n'avons utilisé que des algorithmes itératifs qui consistent à boucler tant que l'on n'est pas au bout.
- Cette fois-ci, on va créer un algorithme récursif.

38

1.4.5 Compter le nombre d'éléments d'une liste chaîné

```
int nombreElements(Liste L){
    // Si la liste est vide, il y a 0 élément
    if(L == NULL)
        return 0;
    // Sinon, il y a un élément (celui que l'on est en train de traiter)
    // plus le nombre d'éléments contenus dans le reste de la liste
    return nombreElements(L->suivant)+1;
}
```

39

1.4.1.7 Recherche du k^{ème} élément

- Il suffit de se déplacer **k** fois à l'aide du pointeur **tmp** le long de la liste chaînée et de renvoyer l'élément à l'indice **k**.
- Si la liste contient moins de **i** élément(s), alors nous renverrons **NULL**.

40

1.4.1.7 Recherche du k^{ème} élément

```
Noeud * kiemeNoeud(Liste L, int k){
    int i;
    // On se déplace de k cases, tant que c'est possible
    for(i=0; i<k && L != NULL; i++){
        L = L->suivant;
    }
    // Si l'élément est NULL, c'est que la liste contient
    // moins de i éléments
    if(L == NULL){
        return NULL;
    }
    else{
        // Sinon on renvoie l'adresse de l'élément i
        return L;
    }
}
```

41

1.4.7 Effacer tous les éléments ayant une certaine valeur

- Dans cette dernière fonction, nous allons encore une fois utiliser un algorithme récursif.

```
void supprimerElement(Liste * adrListe, Type valElement){
    if((*adrListe) != NULL){
        if((*adrListe)->info == valElement){
            Noeud * tmp = (*adrListe)->suivant;
            free(*adrListe);
            (*adrListe) = tmp;
            supprimerElement(&tmp, valElement);
        }
        else{
            supprimerElement(&((*adrListe)->suivant), valElement);
        }
    }
}
```

42

2 PILES ET FILES (*STACK & QUEUE*)