REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

Université de M'sila
Faculté des Mathématiques et de l'Informatique
Département d'informatique

جامعة المسيلة
كلية الرياضيات والإعلام الآلي
قسم الإعلام الآلي

Chapitre 3: Partie 2

Les pointeurs et les listes chaînées

Algorithmique et structure de données 2

Présenté par: Dr. Benazi Makhlouf
Année universitaire: 2022/2023

Contenu du chapitre 03 part 2:

5. Listes chaînées
6. Opérations sur les listes chaînées
7. Liste doublement chaînée
8. Listes chainées spéciales
    1. Les files (Queue)
    2. Les piles (Stack)

1. Introduction

Les tableaux représentent un concept important dans tout langage de programmation car ils permettent un accès rapide à leurs éléments. Cependant, ils présentent deux inconvénients :

- Les éléments du tableau doivent être contigus (adjacents) en mémoire.
- Il n'est pas possible d'insérer ou de supprimer des éléments dans le tableau sans recréer le tableau à nouveau.

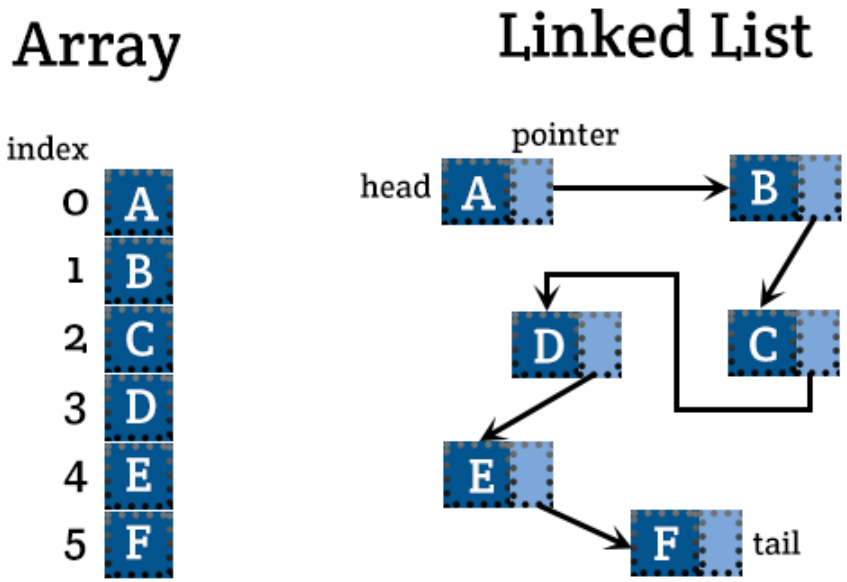
Nous avons donc besoin d'une autre structure de données connue sous le nom de **liste chaînée**.

Définition

- Les **listes chaînées (linked list)** sont une structure de données récursive.
- Elles sont composées de nœuds du même type crées dynamiquement, reliés les uns aux autres par des pointeurs.
- Contrairement aux tableaux, les nœuds des listes chaînées peuvent se trouver dans des emplacements non contigus de la mémoire.
- Les listes chaînées sont constituées d'éléments appelés enregistrements, élément, nœuds ou cellules.
- Chaque élément contient:
  - un ou plusieurs champs pour stocker des données.
  - un pointeur (lien) vers l'élément suivant de la liste.

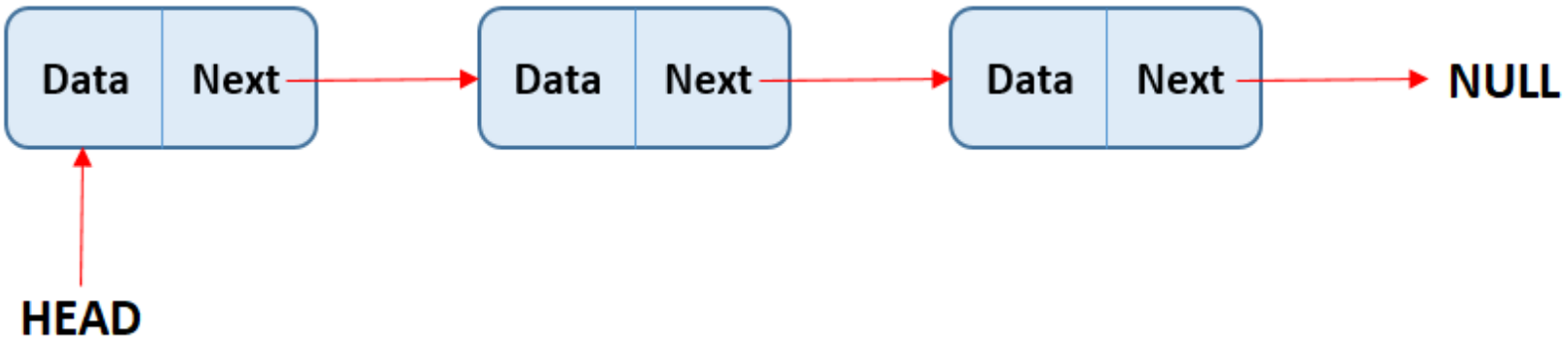
Listes chaînées vs Tableaux

- Les listes chaînées permettent de modifier leur dimension en insérant ou supprimant des éléments de n'importe quelle position dans la liste.
- Pour accéder à un élément de la liste, il faut partir de son en-tête et parcourir tous les éléments qui le précèdent, ce qui peut prendre plus de temps que l'accès direct aux éléments d'un tableau.
- Les listes chaînées sont donc une structure de données linéaire, en opposition à la structure de tableau qui permet un accès aléatoire.



La représentation

- En langage C, un nœud est représenté à l'aide de structures "struct", tandis qu'un en-tête est représenté par un pointeur.
- Pour simplifier l'explication, nous utilisons un champ de données unique de type entier appelé "data" pour tous les enregistrements dans la liste, au lieu d'utiliser des champs de données spécifiques pour chaque type d'enregistrement (comme les informations sur les étudiants, le nom, le prénom, la date, etc.).



Déclaration

Déclaration du type des éléments ou des nœuds

```
typedef struct Node {
  int data;
  struct Node * next;
} Node;
```

```
Node structure
  data:entier
  next :^ Node
fin_structure
```

- "**data**" représente les données stockées dans la liste, telles que le nom et le prénom d'un étudiant, la date d'un événement, etc. Ce champ peut être remplacé par toute autre variable correspondant au type de données que vous souhaitez stocker dans la liste.
- "**next**" est un pointeur qui contient l'adresse de l'élément suivant dans la liste, ou **NULL** s'il ne pointe sur aucun élément. Ce champ est important car il permet de relier les nœuds les uns aux autres pour former la liste chaînée.

Déclaration de type de la tête `typedef Node* List;`

Premier élément

```
List head;  // un pointeur simple vers une structure
Node e1,e2; // une structure complexe
head=&e1;
e1.data=1;          ⇔ (*head).data=1;
e1.next=&e1;        ⇔ (*head).next= &e1;
e2.next= NULL;      ⇔ (*(* head ).next).next = NULL;
```

On peut créer un élément Node dynamiques comme suit:

```
head=(Node*) malloc(sizeof Node);
```

Ou en C++

```
head=new Node ;
```

L’opérateur de sélection -> en C

L'expression « head=&e1; » signifie que le pointeur « head » pointe vers l'adresse mémoire de l'objet « e1 ». Pour accéder aux champs de cet objet, tels que « e1.data » ou « e1.next », on peut utiliser la syntaxe « (*head).Data » et « (head).next ». Cependant, en langage C, il est plus courant d'utiliser l'opérateur de sélection de champ ->, ce qui donne « head->Data » et « head->next ». Cela permet d'éviter l'utilisation de l'opérateur de déréférencement (), ce qui rend le code plus concis et plus lisible.

```
head->data=1;          ⇔ (*head).data=1;
head->next= NULL;
```

Observations

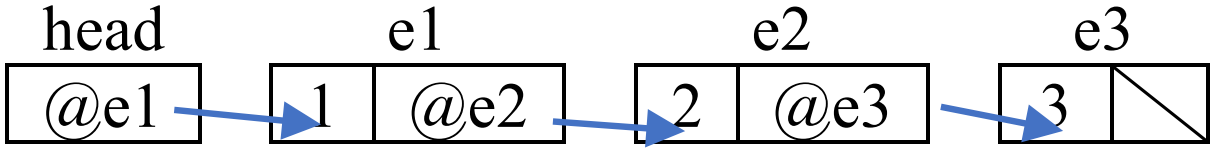
- « head.data » est **incorrect** car « head » est un **pointeur**, et pas une **structure**.
- Les pointeurs 'head', 'e1.next' sont tous de même type, ce qui signifie qu'il est possible de faire des affectations entre eux.
- -> est prioritaire de *

Le parcours d'une liste chaînée

Le dernier élément

Le dernier élément de la liste n'a pas d'élément suivant, donc NULL est affecté à son pointeur 'next'. Lors du parcours de la liste, NULL est utilisé pour vérifier si le dernier élément a été atteint ou non.

Le parcours: L'exemple suivant montre comment parcourir les éléments d'une liste chaînée. Supposons que nous ayons la liste suivante :



Comme "e1.next" pointe vers "e2", on peut faire pointer "head" vers "e2" en effectuant l'opération suivante : "head = e1.next;".

Maintenant que "head" pointe vers "e2", alors "head->next" équivaut à "e2.next".

`head= &e2⇔ head= e1.next⇔ head= head->next`

Donc, pour passer d'un nœud à l'autre, on utilise « head=head->next »

parcours

```
while (head!= NULL){
    //do something
    head = head->next;
}
```

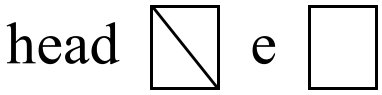
```
 while (head){
     //do something
     head = head->next;
 }
```

```
TQ (head≠NULL)
faire
    //faire qqs
chose
    head←head^.next
FTQ
```

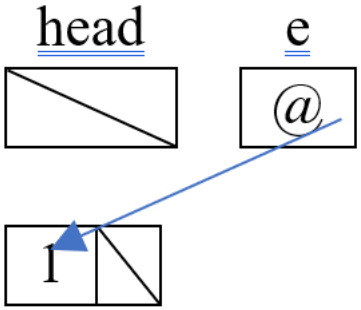
La Création

Supposons que nous ayons une liste vide avec head=NULL; Pour créer un nouvel élément, on utilise la fonction d'allocation dynamique de mémoire malloc().

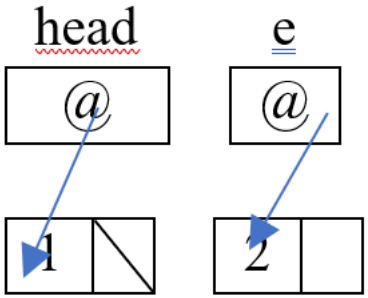
```
List e, head= NULL;
```




```
e = malloc( sizeof(Node));
e->data=1;
e->next= NULL;

head = e;
```




```
e = malloc( sizeof(Node));
e->data=2;
```



élément « e » a été créé et peut être ajouté en tête de la liste en le reliant au premier élément de la liste grâce à son champ next.

e->next=head; head=e;

ou à la fin head->next=e

6. Opérations sur les listes chaînées

Il existe plusieurs façons de créer des fonctions pour ajouter ou supprimer un élément de la liste :

- En utilisant des fonctions qui prennent une liste en paramètre et renvoient une liste. Dans ce cas, la liste peut être passée par valeur.
- En utilisant des procédures et un élément auxiliaire (sentinelle) pour éviter le passage par adresse. Dans ce cas, la liste peut être passée par valeur.
- En utilisant des procédures sans auxiliaire. Dans ce cas, la liste doit être passée par adresse.
- En utilisant des fonctions qui prennent une liste en paramètre et retournent une valeur booléenne (bool) pour nous dire si l'opération a réussi (true) ou non (false). Dans ce cas, la liste doit être passée par adresse. Nous utiliserons cette dernière méthode.

6.1. L'Affichage d'une liste

```
void display_list(List head) {
  while (head != NULL) {
    printf("%d->", head->data);
    head = head->next;
  }
  printf("fin\n");
}
```

```
void display_list(List head) {
  if (head){
     printf("%d->", head->data);
     display_list(head->next);
  } else printf("fin\n");
}
```

6.2. Taille de la liste

```
int size_list(List head){
  int n=0;
  while (head != NULL) {
    head= head->next;
    n++;
  }
  return n;
}
```

```
int size_list(List head){
if (!head)  return 0;
return 1+ size_list(head->next);
}
```

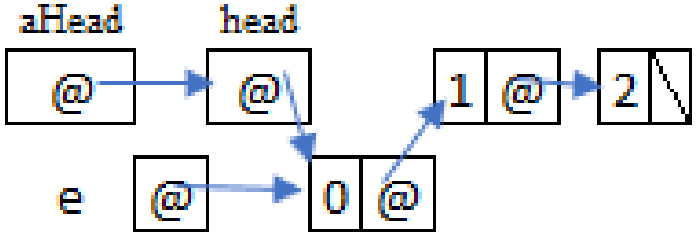
6.3. Ajouter un élément à la liste

Le processus d'ajout d'un élément à une liste chaînée se déroule en 3 étapes :

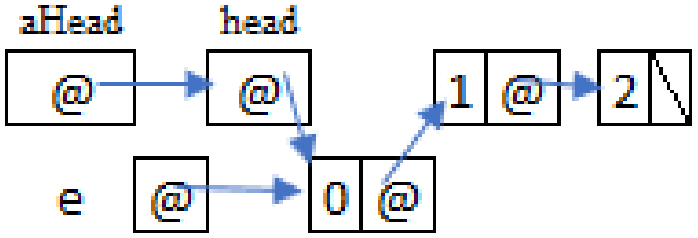
1. Créer et initialiser un nœud
2. Déterminer l'emplacement du nœud.
3. Ajoutez le nœud à la liste en réaffectant les pointeurs.

Ajouter un élément au début de la liste (en tête)

1. L'ajout d'un élément au début de la liste nécessite de changer la tête de la liste, il est donc important de passer la liste par adresse pour que cette modification soit visible dans l'appelant.
2. Créer un nouvel élément, et s'il échoue, renvoyer false à l'appelant
3. Initialiser l'élément
4. Remplacer « next » de « e » pour pointer vers le premier élément de la liste
5. Modifier la tête de la liste pour pointer vers le nouvel élément, « aHead et e » sont deux variables locales qui sont supprimées immédiatement après l'exécution de la procédure

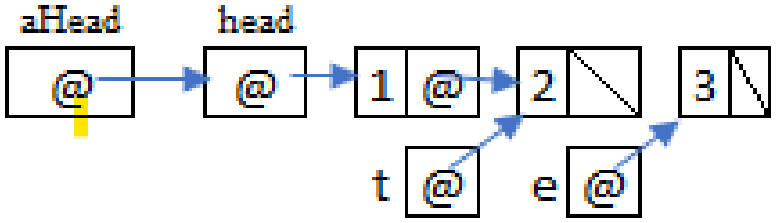


```
int add_head(List* aHead, int d){

 List e = malloc(sizeof(Node));
  if (e == NULL) {
    return 0;
  }
 e-> data = d;
 e-> next = *aHead;

  *aHead=e;
   return 1;
 }
```

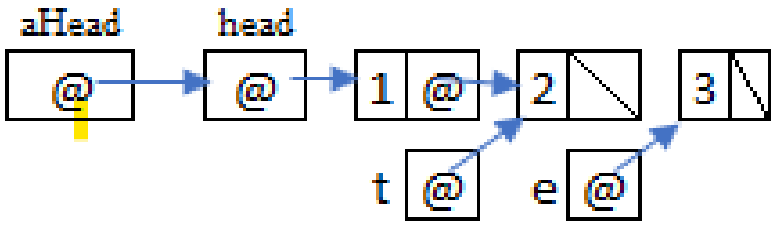


Ajouter un élément à la fin

1. Il est possible que nous ajoutions un élément dans l'en-tête, nous devons donc la passer par adresse. Créer un nouvel élément
2. Initialiser l'élément et affectez NULL au « next » car ce sera le dernier élément
3. Si la liste est vide, on va l'insérer dans la tête
4. Si la liste contient au moins un élément, on cherche le dernier élément
5. Insérer l'élément en dernier.



```
int append_end(List*aHead, int d){
  List t;
  List e = malloc(sizeof(Node));
  if (e == NULL) {
    return 0;
  }
  e->data = d;
  e->next = NULL;
 if (*aHead == NULL)
    *aHead = e;

 else {
    t= *aHead;
    while (t-> next != NULL)
      t= t-> next;

    t-> next=e;
   }
   return 1;}
```



6.4. Supprimer un élément de la liste

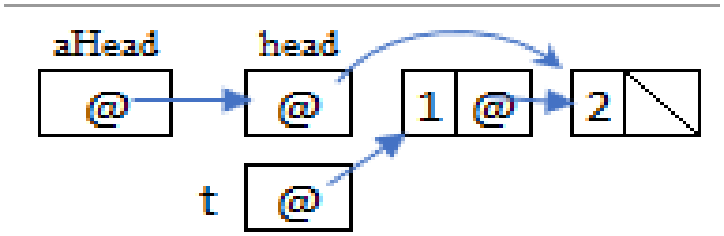
Le processus de suppression d'un nœud d’une liste se déroule en 4 étapes :

1. Déterminez le nœud précédent du nœud à supprimer.
2. Garder l'adresse du nœud à supprimer dans une variable
3. Connectez le nœud précédent au nœud suivant du nœud à supprimer.
4. Videz la mémoire réservée par le nœud à supprimer.

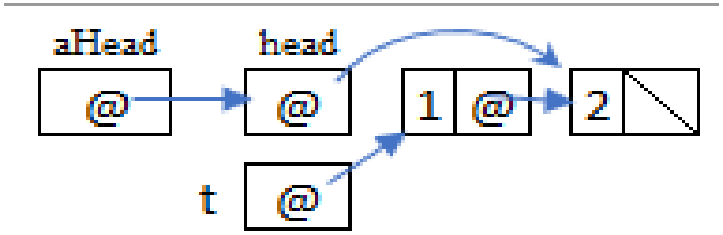
Il y a donc 3 cas, soit la liste est vide, contient un seul élément, ou contient plus d'un élément.

Supprimer l'élément de la tête

1. Il est possible que nous supprimions un élément de l'en-tête, nous devons donc passer la liste par adresse
2. Si la liste est vide, il n'y a pas d'élément à supprimer, donc on renvoie false.
3. Stocke l'adresse du premier élément à supprimer dans t
4. Connecter la tête avec le deuxième élément
5. Vider la mémoire réservée par le premier élément

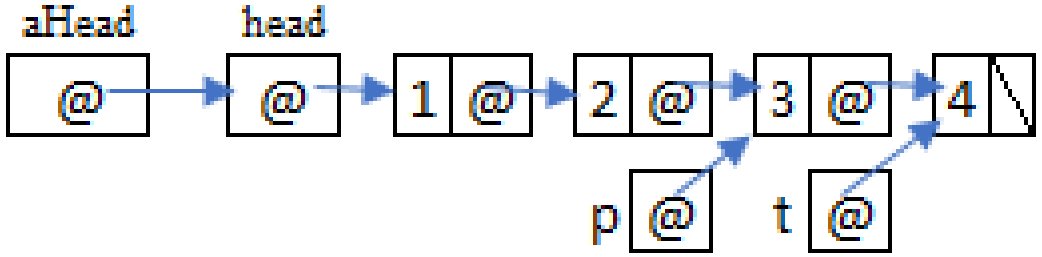


```
int delete_head(List*aHead)
{
   List t;


   if (*aHead== NULL)
      return 0;

   t = *aHead;

   *aHead =t-> next;

   free(t);
   return 1;
}
```

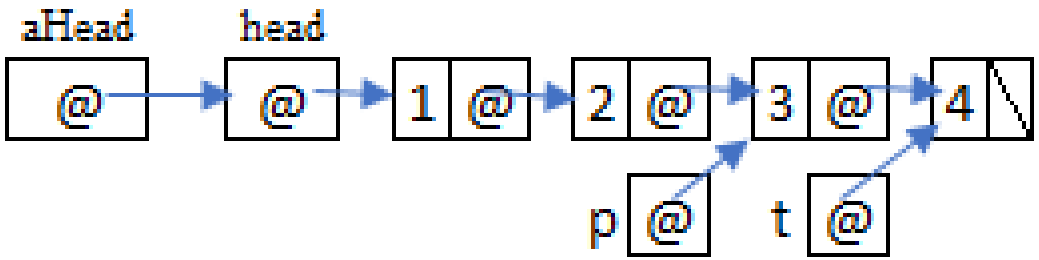


Supprimer un élément de la fin

1. Il est possible que nous supprimions un élément de la tête, nous devons donc passer la liste par adresse. t contient le dernier élément et p est l'avant-dernier élément
2. Si la liste est vide, il n'y a pas d'élément à supprimer, donc on retourne false
3. Si la liste ne contient qu'un seul élément, supprimez-le directement de la tête
4. Si la liste contient plus d'un élément, on cherche le dernier élément t et l'avant-dernier p
5. On affecte NULL à « next » de l'avant-dernier p, car il est devenu le dernier et on supprime le dernier t de la mémoire.



```
int delete_end (List*aHead) {
  List t, p;

  if (*aHead== NULL)
     return 0;

 if ((*aHead)->next ==NULL) {
    free(*aHead);
    *aHead = NULL;
    return 1;
 }
  t = *aHead;
  while (t->next != NULL) {
    p=t ;
    t= t->next;
  }

  p-> next=NULL;
  free(t);
  return 1;
}
```



6.5. Supprimer la liste

1. Nous supprimons de l'en-tête jusqu'à ce que la liste devienne vide
2. Ou en utilisant la fonction delete_head jusqu'à ce qu'elle renvoie false

```
void delete_list(List*aHead) {
  List t;
  while(*aHead!= NULL) {
    t = *aHead;
    *aHead =t-> next;
    free(t);
  }
}
```

```
void delete_list(List*aHead) {
  while (delete_head(aHead));
}
```

6.6. Programme principal (utilisation)

```
int main() {
  List head =NULL;
  add_head(&head, 3);
  add_head(&head, 2);
  append_end(&head, 4);
  add_head(&head, 1);
  append_end(&head, 5);
  printf("size=%d\n", size_list(head));
  display_list(head);
  delete_head(&head);
  delete_end(&head);
  printf("size=%d\n", size_list(head));
  display_list(head);
  delete_list(&head);
  printf("size=%d\n", size_list(head));
  display_list(head);
  return 0;
}
```

L’ecran

- Le programme affichera
  - size=5
  - 1->2->3->4->5->fin
- Puis il affichera
  - size=3
  - 2->3->4->fin
- A la fin il affichera
  - size=0
  - fin

Fin partie 2 du Chapitre 03