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 3

Les pointeurs et les listes chaînées

Algorithmique et structure de données 2

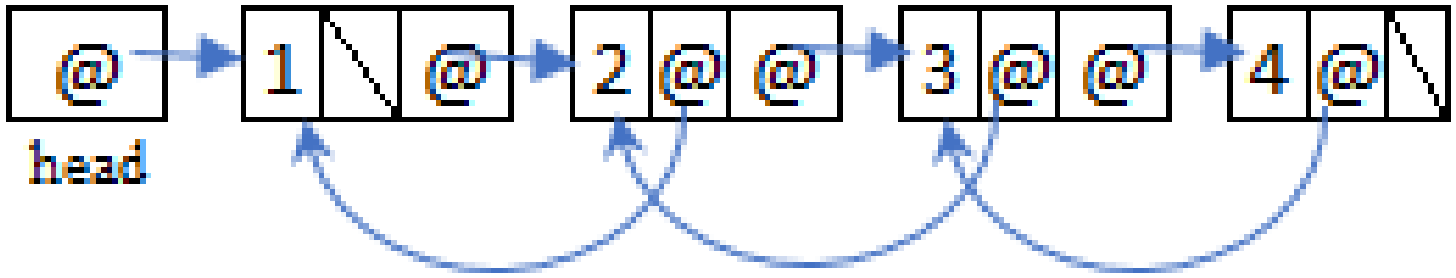
Présenté par: Dr. Benazi Makhlouf
Année universitaire: 2022/2023

Contenu du chapitre 03
part 3:

7. Liste doublement chaînée
8. Listes chainées spéciales
    1. Les piles (Stack)
    2. Les files (Queue)

7. Liste doublement chaînée

En plus des données et du pointeur qui pointe vers l'élément suivant, une liste doublement liée contient un autre pointeur, généralement appelé "prev", qui pointe vers l'élément précédent. Ce pointeur facilite le parcours de la liste dans les deux sens et simplifie ainsi le processus de suppression ou d'insertion d'un élément avant celui sélectionné.



Déclaration

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

"next" est un pointeur qui contient l'adresse de l'élément suivant,

"prev" est un pointeur qui contient l'adresse de l'élément précédent.

Le « prev » du premier élément peut être utilisé pour faire référence au dernier élément de la liste, ce qui accélère le processus d'accès au dernier élément pour l'ajout ou la suppression.

Ajouter un élément au début (tête)

```
e-> next = *aHead; //Changer next de « e » pour qu'il pointe vers le premier
e-> prev = NULL;

 if(*aHead!= NULL)   //prev du premier élément, s'il existe, pointe vers le nouvel.
   (*aHead)->prev = e;
*aHead=e;   //la tête de la liste pointe vers le nouvel élément
```

Supprimer l'élément du début (la tête)

```
 t = *aHead;   //Stocke l'adresse de l'élément à supprimer
*aHead =t-> next; //Se lier avec le deuxième élément

 if(*aHead!= NULL)
   (*aHead)->prev = NULL;
 free(t); //Vider la mémoire réservée
```

Ajouter un élément à la fin

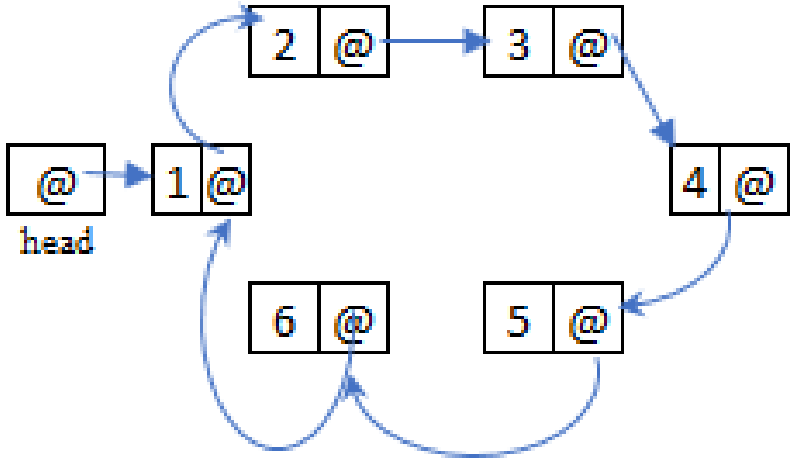
```
  e-> next = NULL;   //car ce sera le dernier
 if (*aHead == NULL) {
    e-> prev = NULL;
    *aHead = e;
  }
 else {
    t= *aHead;
    while (t-> next != NULL)  //le dernier élément est recherché
      t= t-> next;
    e-> prev = t; //relier le nouveau avec le dernier
    t-> next=e;
  }
```

Supprimer un élément du dernier

```
t = *aHead;

while (t->next != NULL) //le dernier élément est recherché

  t= t->next;
p=t->prev; //récupérer l'adresse de l'avant dernier

p ->next=NULL; // devein le dernier

free(t);
```

8. Listes chainées spéciales

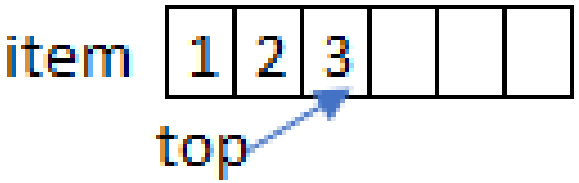
En plus des listes simples et doubles liées, il existe des listes circulaires simples et doubles liées
La liste **circulaire** est une liste chaînée normale, sauf que le dernier élément ne porte pas NULL mais fait référence au premier élément de la li



8.1. Les piles (Stack):

Pile:

- une structure de données abstraite a un ensemble d'enregistrements de même type.
- Les deux opérations possibles sont le push (empiler) et le pop (depiler).
- Les opérations ont lieu à une seule extrémité du groupe appelée le sommet (ou top en anglais).
- La structure de données suit le principe LIFO (Last In First Out).
- Le dernier élément ajouté est le premier à être supprimé, et l'ordre de sortie est l'opposé de l'ordre d'entrée.
- On peut l'implémenter a l'aide des tableaux ou a l'aide



Exemple :

- L'historique du navigateur Web
- La liste des opérations dans Word pour être annuler,

8.1.1. Utilisation du tableau

1. **Déclaration** : Une structure est créée qui contient une table d'éléments item allouée dynamiquement, et top l'endroit d'ajout ou de suppression et capacity qui représente la taille.

```
typedef struct Stack{
  int *item;
  int top, capacity;
} Stack;
```

2. **init**(): La table est créée et la valeur 0 est affectée à top pour indiquer que la pile est vide.

```
Stack init(int size) {
  Stack s;
  s.top = 0;
  s.capacity = size;
  s.item=(int*)malloc(size* sizeof(int));
  return s;
}
```

3. **isEmpty**():La pile est vide si la valeur de top est 0.

```
bool isEmpty(Stack s){
    return s.top==0;
}
```

4. isFull(): Si le tableau est plein, top est égal à capacity.

```
bool isFull(Stack s){
    return s.top==s.capacity;
}
```

5. **Pop**(): La fonction Pop permet de décrémenter 'top' et de renvoyer le dernier élément qu'il pointe.

```
int Pop(Stack &s){
  int x;
  if(isEmpty(s)){
    printf("error: Stack is empty");
    exit(1);
  }
  s.top--;
  x=s.item[s.top];
  return x;
}
```

6. **Push** (): La fonction Push permet d'ajouter l'élément x à la table et d'incrémenter de 1 le pointeur 'top'. Vous devez vous assurer que la pile (table) n'est pas pleine.

```
void Push(Stack &s, int x){
  if(isFull(s)){
    printf("error: Stack is full");
    exit(1);
  }
  s.item[s.top]=x;
  s.top++;
}
```

8.1.2. En Utilisant des listes chainées :

Pour simuler une pile à l'aide de listes, l'ajout et la suppression doivent se faire du même côté (au début ou à la fin).

1. **Push** ():La fonction push est la même que la fonction add_head

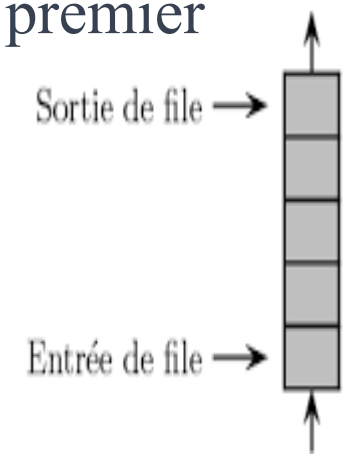
```
void Push(List &l, int x) {
  List e = new Node;
  e->data = x;
  e->next = l;
  l=e;
}
```

2. **Pop**(): La fonction pop est la même que la fonction delete_head sauf que la fonction pop renvoie l'élément qui a été supprimé. Donc, avant de supprimer le premier élément t, nous sauvegardons t-> data dans x, puis le supprimons et renvoyons la valeur de x.

```
int Pop(List &l) {
  List t; int x;
  if(l==NULL){
    printf("error: Stack is empty");
    exit(1);
  }
  t = l;
  l =t-> next;
  x=t->data;
  free(t);
  return x;
}
```

8.2. Les files (Queue)

- La file est une structure de données abstraite pour stocker des enregistrements du même type.
- Elle offre deux opérations essentielles : l'ajout d'un nouvel élément (enQueue ou enfiler) et la suppression d'un élément (deQueue ou défiler).
- Elle respecte la propriété FIFO (First In First Out), ce qui signifie que le premier élément ajouté est le premier élément à être supprimé.
- En d'autres termes, l'ordre de sortie est identique à l'ordre d'entrée.
- On peut l'implémenter a l'aide des tableaux ou a l'aide des listes



Exemple : liste d'événements, file d'attente, liste des fichiers envoyés à l'imprimante...

8.2.1. Utilisation de listes chainées :

- Pour simuler une file à l'aide de listes, il faut ajouter et supprimer des éléments à deux extrémités différentes de la liste.
- On peut ajouter de nouveaux éléments à la fin de la liste et supprimer les éléments au début de la liste.
- Cette approche peut être inversée en ajoutant des éléments au début de la liste et en supprimant les éléments à la fin.
- La structure de liste permet des insertions et des suppressions rapides et efficaces, sans nécessiter de déplacements coûteux d'éléments comme avec l'implémentation à l'aide de tableaux.

1. **Déclaration** : Nous créons une structure qui contient deux champs, le premier fait référence au premier élément de la liste et le second au dernier élément de la liste.

```
typedef struct Queue{
  struct Node* first, *last;
  int size;
} Queue;
```

initQ() initialise la file en affectant **NULL** aux champs first et last de la structure.

```
Queue initQ(){
 Queue Q;
 Q.first =NULL;
 Q.last =NULL;
 Q.size =0;
 return Q
}
```

```
bool isEmpty(Queue Q){
    return Q.size==0;
}
```

enQueue():est identique « append_end »

```
void enQueue(Queue &Q, int x) {
  Node *e = malloc(sizeof(Node));
  e->data = x;
  e->next = NULL;
  if(isEmpty(Q))
    Q.first =e;
  else
    Q.last->next =e;
  Q.last =e;
  Q.size++;
}
```

deQueue():est identique « delete_head »

```
int deQueue(Queue &Q) {
  Node* t; int x;
  if(isEmpty(Q)){
    printf("error: is empty");
    exit(1);
  }
  t = Q.first;
  x = t ->data;
  Q.first = t-> next;
  free(t);
  Q.size--;
  if (Q.size==0) Q.last =NULL;
  return x;
}
```

8.2.2. Utilisation du tableau

1. **Déclaration** : création d'une structure qui contient une table d'éléments allouée dynamiquement en mémoire, un emplacement de début "start" pour l'ajout, un emplacement "end" pour la suppression et "capacity" qui contient le nombre d'éléments qu'on peut ajouter dans la table.

```
typedef struct Queue{
   int *item;
   int start, end, capacity;
} Queue;
```

2. **init**() : création de la table et affectation de la valeur -1 à start et end pour indiquer que la file est vide. Si la création échoue, la fonction renvoie false.


```
Queue init( int size) {
   Queue Q;
   Q.start = -1;
   Q.end = -1;
   Q.capacity= size;
   Q.item= malloc(size * sizeof(int));
   return Q;
}
```

3. **isEmpty**() : vérification si la file est vide en comparant si start =-1

```
bool isEmpty(Queue Q){
    return Q.start ==-1 && Q.end==-1;
}
```

4. **isFull**() : vérification si la file est pleine en comparant si la valeur de start+1 est la même que la valeur de end en utilisant mod "%" pour ramener la table au début.

```
bool isFull(Queue Q){
 return (Q.start+1)% Q.capacity == Q.end;
}
```

5. **enQueue**() : vérification si la file n'est pas pleine, puis ajout 1 à start pour référencer le premier élément vide et ajout x à la table.

```
void enQueue (Queue &Q,int x){
  if(isFull(Q)){
    printf("error: Queue is full");
    return;
  }
  if(isEmpty(Q))
    Q.end=0;

  Q.start= (Q.start +1)% Q.capacity;
  Q.item[Q.start]=x;
}
```

6. **deQueue**() : renvoie le premier élément de la table pointé par end, Si la file est vide, elle informe l'utilisateur.

Si la file devein vide , met -1 dans start sinon ajoute 1 à end.

```
int deQueue(Queue &Q){
  if(isEmpty(Q)){
    printf("error: Queue is empty");
    return;
  }
  int x= Q.item[Q.end];
  if (Q.start == Q.end)
    Q.start = Q.end= -1;
  else
    Q.end= (Q.end+1)% Q.capacity;
  return x;
}
```

Fin du Chapitre 03