Structures de données : cours 2a – Structures de données linéaires

Elena Leroux

Structures de données

- **Les algorithmes emploient des données :**
  - comme argument (en entrée),
  - pour représenter et conserver des résultats intermédiaires,
  - comme espace de recherche,
  - pour représenter ou stocker un résultat.

- **La performance d'un algorithme varie selon la manière dont les données qu'il emploie sont organisées et ceci en fonction :**
  - du coût des accès (ressources système, méthodes d'accès).
  - de la complexité de la recherche dans l'espace d'adressage,
  - de la complexité des opérations d'insertion, de marquage, ou d'extraction dans l'espace d'adressage,
  - de la nécessité répétée de traiter des cas partiellement résolus ou des états intermédiaires.

Plan

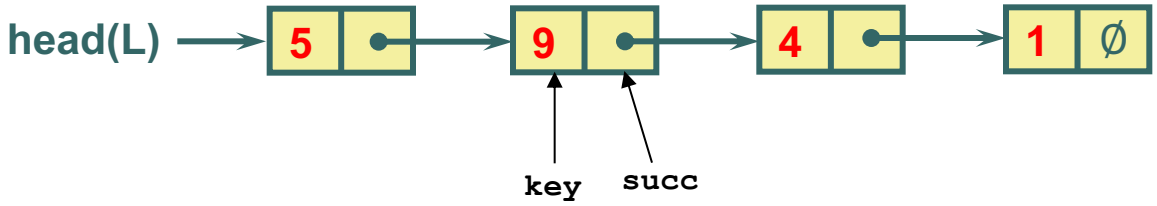
- Structures de données basiques :
  - tableaux
  - listes chaînées :
    - définition des listes chaînées et leurs différents types
    - réalisation d'une liste doublement chaînée
    - opérations de base : recherche, insertion et suppression
  - complexité
- Piles et files
- Hachage

Types de listes chaînées (1/3)

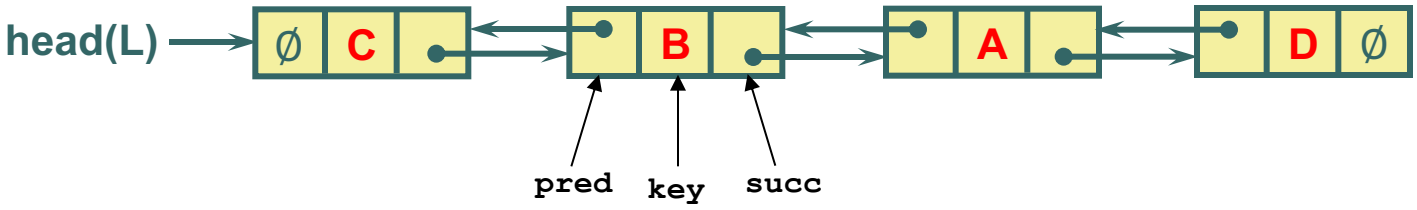
- **Définition :**

  Une liste chaînée (*linked list*) est une structure de données concrète constituée d'une séquence de nœuds et de liens entre les nœuds.

- **Types de listes chaînées :**
  - **Liste simplement chaînée** (*singly linked list*) **:**

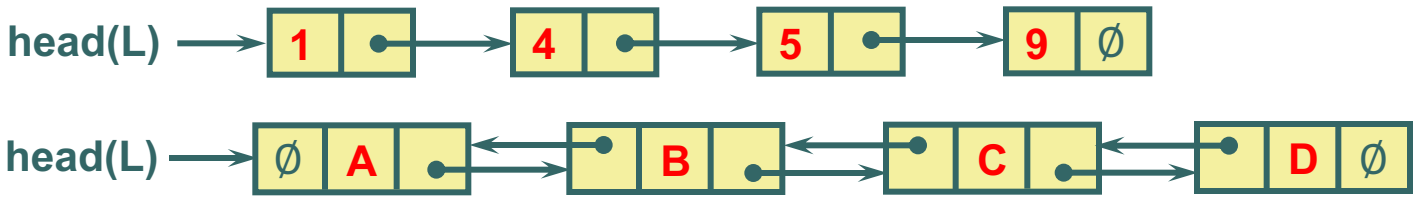


  - **Liste doublement chaînée** (*double linked list*) **:**

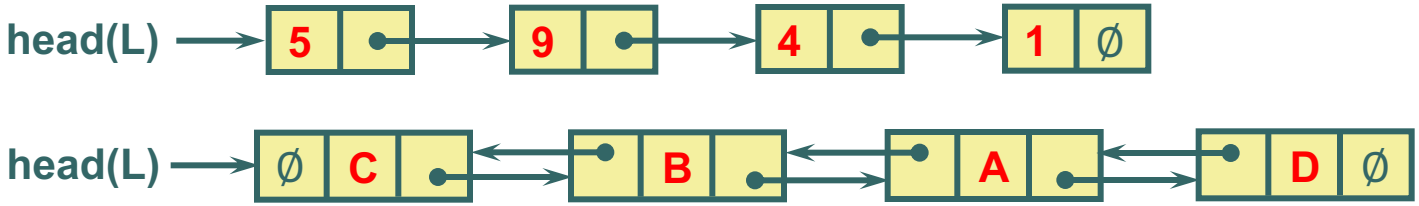


Types de listes chaînées (2/3)

- **Liste triée :**
  - dans une liste triée l'ordre linéaire de la liste correspond à l'ordre linéaire des clés stockées dans les nœuds de la liste.
  - Exemples :

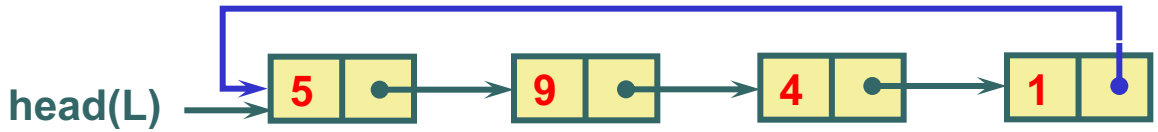


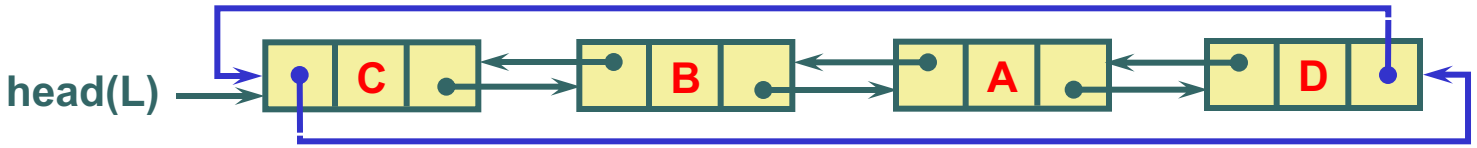
- **Liste non-triée :**
  - dans une liste non-triée les nœuds peuvent apparaître dans n'importe quel ordre.
  - Exemples :



Types de listes chaînées (3/3)

- **Liste circulaire :**
  - dans une liste circulaire, le pointeur `pred` (s'il existe) de la tête de liste pointe sur la queue, et pointeur `succ` de la queue de liste pointe sur la tête.
  - Exemples :





- **Liste non-circulaire :**

Plan

- Structures de données basiques :
  - tableaux
  - listes chaînées :
    - définition des listes chaînées et leurs différents types
    - réalisation d'une liste doublement chaînée
    - opérations de base : recherche, insertion et suppression
  - complexité
- Piles et files
- Hachage

Réalisation des listes doublement chaînées

- **Implémenter la classe `Node` :**

```
public class Node {
  // Instance variables
  private Object key;
  private Node succ;
  private Node pred;

  // Constructors of the class
  public Node() {…}

 // Accessor methods
  public Object getKey() {…}
  public Node getSucc() {…}
  public Node getPred() {…}

 // Modifier methods
  public void setKey(Object newKey) {…}
  public void setSucc(Node newSucc) {…}
  public void setPred(Node newPred) {…}
}
```

- **Implémenter la classe `Liste` :**

```
public class List {
  // Instance variables
  private Node head;

  // Constructors of the class
  public List() {…}

 // Methods
  public void insert(List L, Node n) {…}
  public void delete(List L, Node n) {…}
  public Node search(List L, Node n) {…}
}
```

- **Supposition :**
  - On suppose que les listes sur lesquelles on travaille sont doublement chaînées, non-triées et non-circulaires.

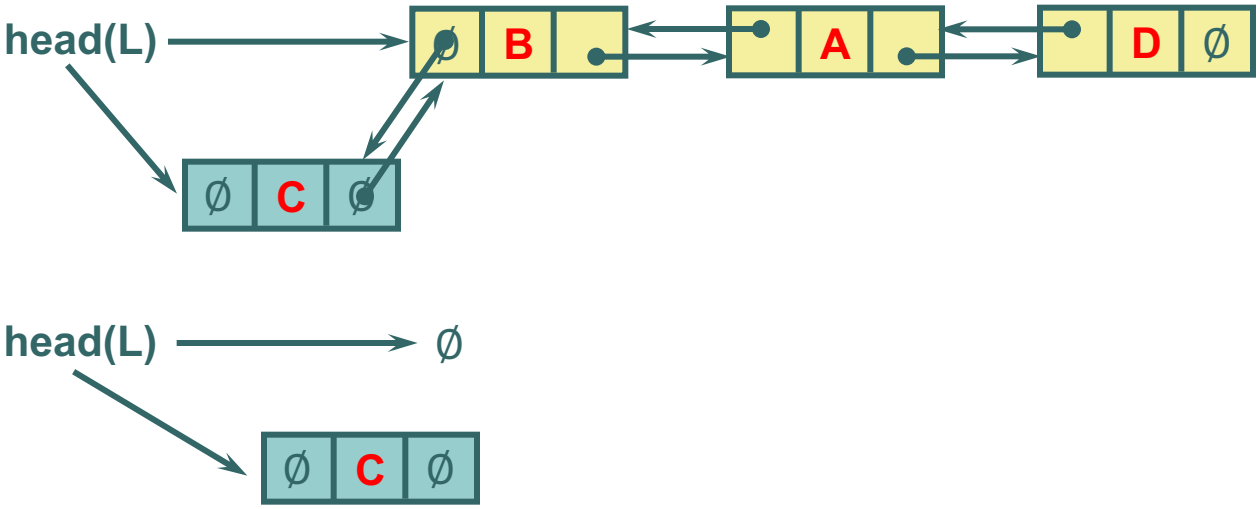
Plan

- Structures de données basiques :
  - tableaux
  - listes chaînées :
    - définition des listes chaînées et leurs différents types
    - réalisation d'une liste doublement chaînée
    - opérations de base : recherche, insertion et suppression
  - complexité
- Piles et files
- Hachage

Insertion dans une liste doublement chaînée

```
Algorithm insert(List L, Node n)
Begin
  if head(L)≠null then
    succ(n) ← head(L);
    pred(head(L)) ← n;
  end;
  head(L) ← n;
End.
```

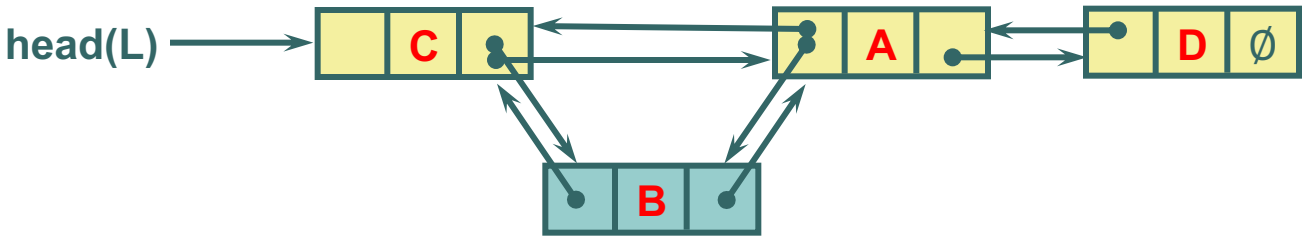
La complexité de l'algorithme `insert` est `O(1)`.



Suppression dans une liste doublement chaînée

```
Algorithm delete(List L, Node n)
Begin
  if pred(n)≠ null then
    succ(pred(n)) ← succ(n);
  else
    head(L) ← succ(n);
  end;
  if succ(n) ≠ null then
    pred(succ(n)) ← pred(n);
  end;
  n ← null;
End.
```

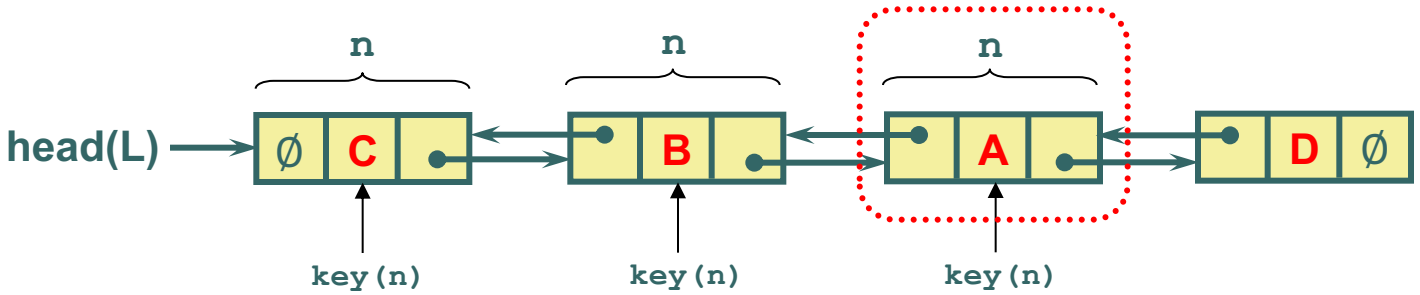
La complexité de l'algorithme `delete` est `O(1)`.



Recherche dans une liste doublement chaînée

```
Algorithm search(List L, Object k) : Node
Begin
  n ← head(L);
  while n ≠ null et key(n) ≠ k do
    n ← succ(n);
  end;
  return n;
End.
```

La complexité de l'algorithme **search** est **O(n)**.



k = A

Plan

- Structures de données basiques :
  - tableaux
  - listes chaînées
  - complexité
- Piles et files
- Hachage

Complexité

- Soient
  - `S` une structure de données (tableau, liste chaînée ou liste doublement chaînée),
  - `k` un indice de tableau,
  - `e` un élément et
  - `x` un indice de tableau ou un pointeur de liste,

Alors la complexité dans pire des cas des trois opérations principales est :

	`recherche(S,k)`	`recherche(S,e)`	`insertion(S,e)`	`suppression(S,x)`
cas non trié : tableau	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
cas non trié : liste chaînée	---	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
cas non trié : liste doublement chaînée	---	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
cas trié : tableau	$\Theta(1)$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$
cas trié : liste chaînée	---	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
cas trié : liste doublement chaînée	---	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$