Algorithmique Avancée

Nicolas Labroche

Cours 3 : listes circulaires, doublement chaînées, piles et files

Université de Tours

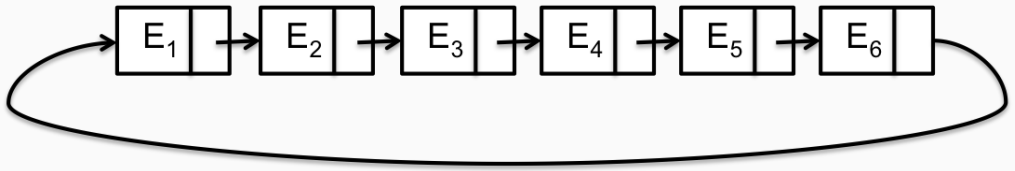
Objectifs du cours

- listes circulaires
- listes doublement chaînées
- opérations sur les piles et files

Listes circulaires

Cycles et listes circulaires

- Cycle : propriété d'une liste à former une chaîne fermée ou boucle
- Liste circulaire : liste formant un cycle
- Pas de début ou de fin dans une liste circulaire



- Différences avec les fonctions pour les listes classiques :
  - critère d'arrêt : comparer à la tête de liste au lieu de `null`
  - initialisation ou ajout en fin : penser au pointeur vers la tête de liste
  - ajouter un élément au début $\neq$ ajouter à la fin car l'élément de tête est utilisé pour tous les tests d'arrêt de parcours de liste

Constructeurs simples d'une liste circulaire

Création d'une classe `CList<T>` qui implémente `LList<T>`

- mais modification du pointeur de fin de liste (`null`) en tête de liste (`head`) pour les parcours
- pas de changement pour la création d'une liste circulaire vide par rapport à une liste

```
public CList(){
    head = null;
    last = null;
    size = 0;
}
```

- en revanche pour une liste à une valeur, le `Node<T>` doit pointer sur lui-même !
- voir la méthode d'ajout d'un élément dans une liste initialement vide

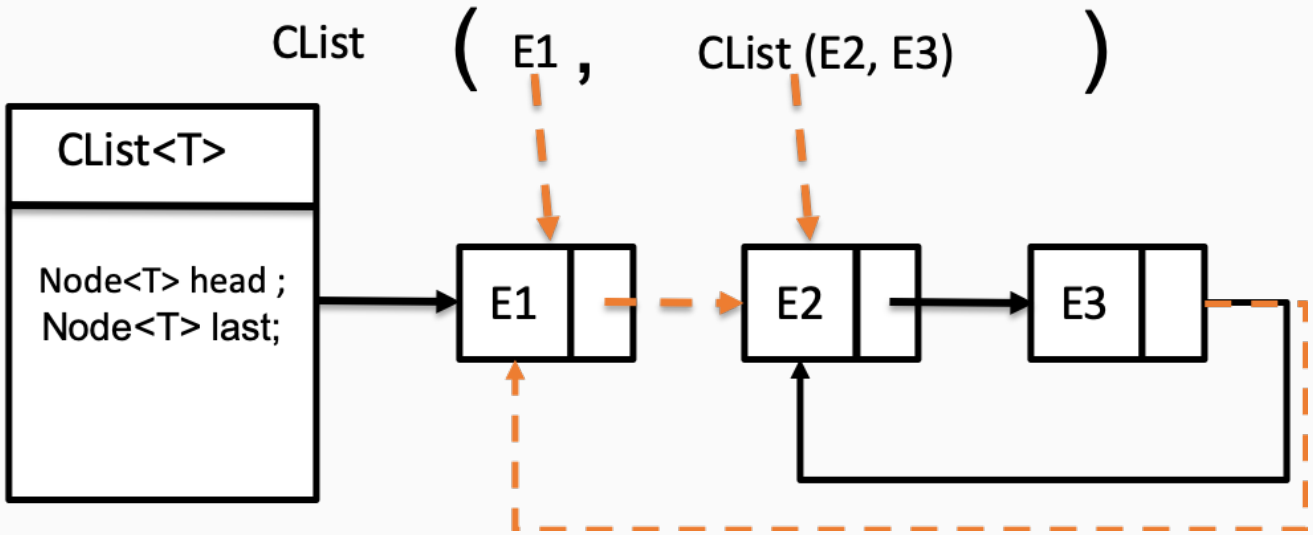
Constructeur à partir d'une autre liste (circulaire)

Tâche complexe

- il faut lier la tête à la liste circulaire existante comme pour une liste normale
- puis il faut changer la fin de liste pour pointer vers le nouveau début de liste

```
public CList(T val, CList<T> tail) {
    head = new Node<T>(val, tail.head);
    size = 1 + tail.size;
    last = tail.last;
    last.next = head; // circular reference here
}
```

Illustration du constructeur



- la difficulté est de remplacer le pointeur qui retourne intialement vers la tête de la liste ajoutée par la nouvelle tête de liste

Affichage d'une liste circulaire

Comme précédemment, parcours avec arrêt lors de la découverte de la tête de liste

- la difficulté vient des listes qui ne contiennent qu'un élément et où `head` pointe sur lui-même
- car dans ce cas, il est impossible d'écrire

```
Node<T> p = head;
while (p != head) { ...}
```

- il faut envisager un `do ... while` ou bien sortir la première itération de la boucle

Affichage d'une liste circulaire (2)

- avec sortie de la première itération de la boucle

```
public String toString() {
    StringBuilder sb = new StringBuilder("(");
    if (!isEmpty()){
        Node<T> p = head;
        do {
            sb.append(p.value);
            if (p.next != head) sb.append(",");
            p = p.next;
        } while (p != head);
    }
    sb.append(")");
    return sb.toString();
}
```

Ajout d'une valeur dans une liste circulaire

Cas 1 : ajout en tête de liste : penser à bien reconstruire les références

```
public void add(T elem, int index) {
    Node<T> p = this.head;
    Node<T> tmp = new Node<T>(elem);
    if (index == 0) { // insert in first position
        tmp.next = head;
        head = tmp;
        last.next = head; // circular
    }
    [...]
}
```

Ajout d'une valeur dans une liste circulaire

Cas 2 : ajout au milieu / fin de liste : pas de changement particulier

```
public void add(T elem, int index) {
    Node<T> p = this.head;
    Node<T> tmp = new Node<T>(elem);
    if (index == 0) {
    [...]
    } else { // idem normal linked list
        int i = 1;
        while (p.next != head && i < index){
            p = p.next;
            i++;
        }
        tmp.next = p.next;
        p.next = tmp;
    }
}
```

Suppression d'une valeur dans une liste circulaire

- suppression uniquement si l'index passé en argument fait sens

```
public void remove(int index) {
    if (index < size){
        if (index == 0) pop();
        else {
            int cpt = 0;
            Node<T> p = head;
            while (cpt < index - 1) {
                p = p.next; cpt++;
            }
            p.next = p.next.next;
        }
        size --;
    }
}
```

Suppression d'une valeur dans une liste circulaire

- attention : suppression du 1er élément nécessite de reconstruire la référence circulaire

```
private void pop(){
    if (size <= 1){
        head = null;
        last = null;
        size = 0;
    } else {
        head = head.next;
        last.next = head;
        size = size - 1;
    }
}
```

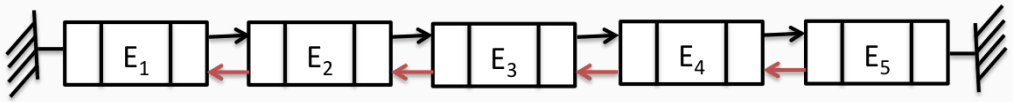
Listes doublement chaînées

Définition des listes doublement chaînées

Une liste doublement chaînée est :

- une structure auto-référentielle avec 2 structures de même type
  - à la différence de la liste chaînée qui n'en référence qu'une seule

- une liste qui possède une référence vers l'élément suivant et une référence vers l'élément précédent



Changement de la classe `Node`

- **nouvelle** classe de liste doublement chaînée `DNode<T>`
- une valeur + un pointeur suivant + un pointeur précédent

```
public class DNode<T>{
    T value;
    DNode<T> prev;
    DNode<T> next;

    public DNode(T elem) {
        this.value = elem; this.prev = null; this.next = null;
    }

}
```

Modifications par rapport à la classe `Node`

- ajout d'un constructeur avec 3 arguments (voir après)
- attention à la référence partant de `next` si `next` est vide !

```
public DNode(T elem, DNode<T> _prev, DNode<T> _next) {
    this.value = elem;
    this.prev = _prev;
    if (_prev != null) _prev.next = this;
    this.next = _next;
    if (_next != null) _next.prev = this;
}
```

Avantage des listes doublement chaînées

- Possibilité de parcourir la liste dans les deux sens
- Insertion avant ou après un élément de la liste sans disposer d'un pointeur sur un voisin
- Attention le constructeur `DNode` gère les liens avant / arrière

Avantage des listes doublement chaînées

```
public void add(T elem, int index) {
    DNode<T> p = this.head;

    if (index == 0) {
        DNode<T> tmp = new DNode<T>(elem, null, head);
        head = tmp;
    } else {
        int i = 1;
        while (p.next != null && i < index){
            p = p.next;
            i++;
        }
        // insert tmp with double linkage
        DNode<T> tmp = new DNode<T>(elem, p, p.next);
    }
}
```

Limites des listes doublement chaînées - 1

- Structure plus coûteuse en mémoire 2 références / élément
- Et en temps : 3 ou 4 opérations pour 1 insertion / suppression
- par exemple, réfléchir au code pour la suppression d'un élément

Limites des listes doublement chaînées - 2

```
public void remove(int index) {
    if (index < size){
        if (index == 0) {
            this.head = this.head.next;
            this.head.prev = null; // double linked list
        }
        else { // searching for the element to remove
            int cpt = 0;
            DNode<T> p = head;
            while (cpt < index - 1) {
                p = p.next;
                cpt++;
            }
            // found the element to remove!
            // see next slide for the code
        }
        size --;
    }
}
```

Limites des listes doublement chaînées - 3

Suppression de l'élément `q` identifié : plusieurs manipulations de pointeurs !

```
DNode<T> q = p.next; // q = the element to remove
if (q.next != null){ // q is not the last element
    p.next = q.next;
    q.next.prev = p;
} else { // in case q is the last element
    p.next = null;
    q.prev = null;
}
```

Pile et file

Pile et File

- Piles et Files sont des structures cruciales en informatique
  - Une pile permet de gérer le appels imbriqués de méthodes
  - Une file permet de gérer les priorités entre plusieurs opérations concurrentes
- D'un point de vue bas niveau, elles peuvent être implémentées comme des listes mais pas obligatoirement
- S'il y a beaucoup d'opérations d'entrée / sortie alors la structure de "type liste" est idéale
- Introduction d'une nouvelle interface minimaliste

```
public interface SimpleCollection<T> {
    public boolean add(T elem);
    public T remove();
}
```

Pile

- Une pile (stack en anglais) est une structure de données
- Fondée sur le principe du “dernier arrivé, premier sorti”
- Last-In, First-Out : LIFO
- Utilisation des piles
  - empilement de l’adresse de retour + paramètres
  - dans la fonction : dépilement de l’adresse de retour + paramètres
  - empilement de la valeur de retour de la fonction
- Les piles peuvent avoir optionnellement une taille limitée en mémoire
- Plusieurs implémentations possibles : tableau ou type liste
- Manipulation par les fonctions suivantes :
  - `boolean add(T valeur)` : empiler une valeur en tête de la pile

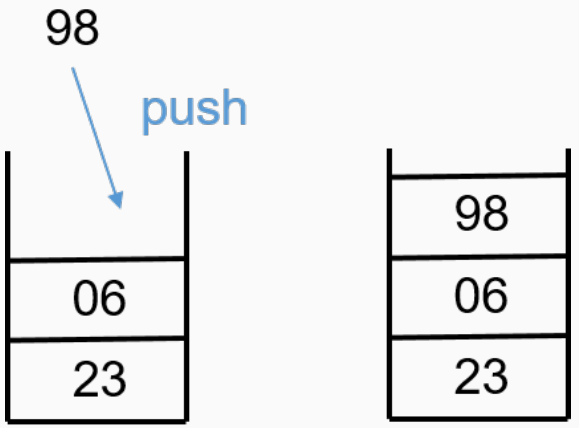
Constructeur de pile

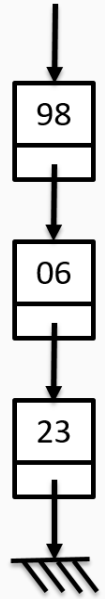
- 2 constructeurs : gestion ou pas de la taille maximale

```
public class Stack<T> implements SimpleCollection<T>{

    protected Node<T> head;
    protected Node<T> last;
    protected int size;
    int max_size;

    public Stack(){
        head = null;
        last = null;
        size = 0;
        max_size = -1;
    }

    public Stack(int maxSize){
        this();
        max_size = maxSize;
    }
```

Empiler : `add` (1)

- Méthode qui ajoute un élément en tête de pile
- Idem à insertion début pour les listes chaînées
- Retourne un booléen qui indique si l'insertion a été possible





Empiler : `add` (2)

- attention à la gestion de la taille maximale de la pile

```
public boolean add(T elem) {
    if (max_size == -1 || (max_size != -1 && size < ↩
        max_size)){
        if (head == null){
            head = new Node<>(elem);
            last = head;
            size = 1;
        } else {
            head = new Node(elem, head);
            size ++;
        }
        return true;
    } else return false;
}
```

Dépiler : `remove`

- Modifie la pile en supprimant la tête de pile (la 1ère valeur)
- Retourne la valeur contenue dans la tête

```
public T remove() {
    if (size > 0){
        T tmp = head.value;
        head = head.next;
        size --;
        return tmp;
    } else return null;
}
```

File

- Une file (queue en anglais) est une structure de données
- Fondée sur le principe du “premier arrivé, premier sorti”
- First-In, First-Out : FIFO

- Utilisation : gestion de files d’attentes
  - requêtes en attente sur un serveur (web, impression)
  - affectation équitable du temps processeur
  - liste des nœuds à visiter dans un parcours d’arbre en largeur

- Implémentation par une structure de type liste
- Manipulation par les méthodes de `SimpleCollection`
  - `boolean add(valeur)` : ajouter une valeur en fin de file
  - `T remove()` : renvoie l’élément en tête de la file et le supprime

Constructeur de file

- comme pour les piles, 2 constructeurs : gestion ou pas de la taille maximale

```
public class Queue<T> implements SimpleCollection<T>{

    protected Node<T> head;
    protected Node<T> last;
    protected int size;
    int max_size;

    public Queue(){
        head = null;
        last = null;
        size = 0;
        max_size = -1;
    }
}
```

Ajouter à une file : `add`

- la tête de la file est la tête de liste
- insertion d'un nouvel élément en fin de liste

```
public boolean add(T elem) {
    if (max_size == -1 || (max_size != -1 && size < ↩
        max_size)){
        if (head == null){
            head = new Node<>(elem);
            last = head;
            size = 1;
        } else {
            last.next = new Node(elem); // insert in the ↩
                end
            last = last.next;
            size ++;
        }
        return true;
    } else return false;
}
```

Enlever un élément de la file : `remove`

- modifie la file modifiée
- équivalent de la suppression du premier élément d'une liste chaînée

```
public T remove() {
    if (size > 0){
        T tmp = head.value;
        head = head.next;
        size --;
        return tmp;
    } else return null;
}
```

En Java

Pile et file en Java (1)

- Implémentation des piles LIFO avec l'interface `Stack`

```
Stack<Integer> pile1 = new Stack<Integer>();
pile1.pop();  // depiler
pile1.push(23); // empiler
boolean vide = pile1.empty();
```

- Implémentation des files FIFO avec la classe `LinkedList`

```
LinkedList<Integer> file = new LinkedList<Integer>();
Integer tete = file.poll(); // recuperation et ↩
    suppression de la tete de file
```

Pile et file en Java (2)

- Plusieurs classes implémentent les interfaces Java
  - `Deque<E>`
  - `Collection<E>`
  - `Queue<E>`
- $\Rightarrow$ classes mixtes pouvant être utilisées en tant que pile, file, liste

```
// Plus rapide que Stack pour les piles
Deque<Integer> pile = new ArrayDeque<Integer>();
// Plus rapide que LinkedList pour les files
Deque<Double> file = new ArrayDeque<Double>();
```