1.1

Algorithmique et Structures de Données

Jean-Charles Régin

Licence Informatique 2ème année

1.2

Listes et variantes

Jean-Charles Régin

Licence Informatique 2ème année

Listes

- Une **liste chaînée** désigne une structure de données représentant une collection ordonnée et de taille arbitraire d'éléments.
- L'accès aux éléments d'une liste se fait de manière séquentielle
  - chaque élément permet l'accès au suivant (contrairement au cas du tableau dans lequel l'accès se fait de manière absolue, par adressage direct de chaque cellule dudit tableau).
- **Un élément contient un accès vers une donnée**

Liste

- Le principe de la liste chaînée est que chaque élément possède, en plus de la donnée, des pointeurs vers les éléments qui lui sont logiquement adjacents dans la liste.

- **premier(L)** désigne le premier élément de la liste
- **nil** désigne l'absence d'élément

- **Liste simplement chaînée :**
  - **donnée(elt)** désigne la donnée associée à l'élément elt
  - **suivant(elt)** désigne l'élément suivant elt

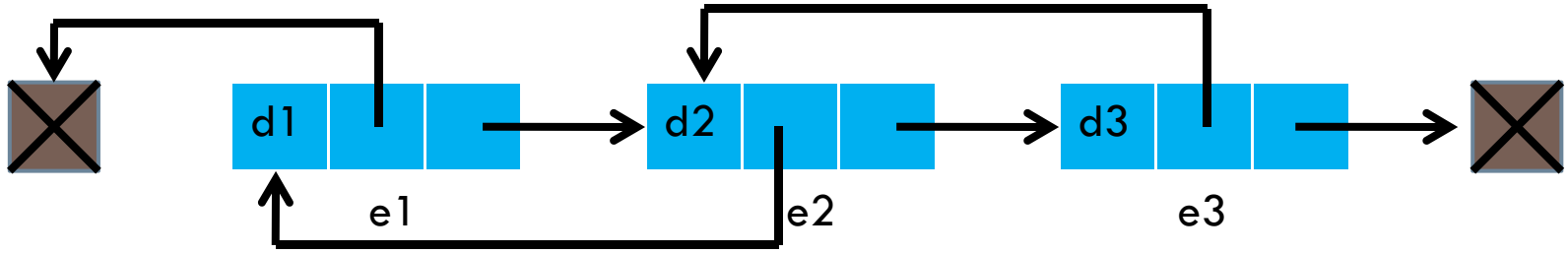
Liste

- Le principe de la liste chaînée est que chaque élément possède, en plus de la donnée, des pointeurs vers les éléments qui lui sont logiquement adjacents dans la liste.

- **Liste doublement chaînée :**
  - **donnée(elt)** désigne la donnée associée à l'élément elt
  - **suivant(elt)** désigne l'élément suivant elt
  - **précédent(elt)** désigne l'élément précédant elt

Liste doublement chaînée

- Représentation : ⟶ lien



- premier(L)=e1
- suivant(e1) = e2; précédent(e1) = nil
- suivant(e2) = e3; précédent(e2) = e1
- suivant(e3) = nil; précédent(e3) = e2

Liste : opérations

- Trois opérations principales
  - Parcours de la liste
  - Ajout d'un élément
  - Suppression d'un élément

- A partir de là d'autres opérations vont être obtenues : recherche d'une donnée, remplacement, concaténation de liste, fusion de listes, etc.

Insertion sous condition d'un élément

- Liste L doublement chaînée
- On veut insérer l'élément elt dans la liste avant le premier élément de la liste qui est associée à une donnée > 8

- Exemple : L : 5 7 4 9 6 5, on insère 5 7 4 elt 9 6 5

Insertion sous condition d'un élément


```
inserer(L,elt,val)
// on suppose que L n'est pas vide
e ← premier(L);
// on cherche la position
tant que(e ≠ nil et donnee(e)<=val){
   e ← suivant(e);
}
if (e = nil){
// on insère en fin
```

- **Zut je n'ai pas la fin !!!**

Insertion sous condition d'un élément


```
inserer(L,elt,val)
// on suppose que L n'est pas vide
e ← premier(L);
// on cherche la position
dernier ← e;
tant que (e ≠ nil et donnee(e) <= val){
    dernier ← e;
    e ← suivant(e);
}
if (e = nil){
// on insère en fin
    suivant(dernier) ← elt
    précédent(elt)← dernier
    suivant(elt)← nil
} else {
    // on insère avant e
```

- **Pb : faut tester avec le premier !!!**

Insertion sous condition d'un élément


```
inserer(L,elt,val)
// on suppose que L n'est pas vide
e ← premier(L);
// on cherche la position
dernier ← e;
tant que (e ≠ nil et donnee(e) <= val){
      dernier ← e;
      e ← suivant(e);
}
if (e = nil){
      // on insère en fin
      suivant(dernier) ← elt
      précédent(elt)← dernier
      suivant(elt)← nil
} else {
      // on insère avant e
      prec ← précédant(e)
      précédent(e) ← elt
      suivant(elt) ← e
      précédent(elt) ← prec
      if (prec = nil){
          premier(L) ← elt
      } else {
          suivant(prec) ← elt
      }
}
```

Listes avec sentinelles

- On introduit deux éléments « bidon », appelé sentinelles
  - A la fois comme premier et comme dernier.
- Ces éléments sont cachés
  - Le vrai premier est le suivant de la sentinelle
  - Le vrai dernier est le précédent de la sentinelle
- Cela évite les problèmes avec les tests avec la valeur nil, puisqu'il y a toujours un suivant ou un précédant pour les éléments visibles dans la liste.

- Insertion après e de elt

```
suivant(elt) ← suivant(e)
précédent(elt) ← e
précédent(suivant(e)) ← elt
suivant(e) ← elt
```

- Marche toujours ! Plus besoin de tests !

Listes avec sentinelles

- On introduit deux éléments « bidon », appelé sentinelles
  - À la fois comme premier et comme dernier.
- Cet élément est caché
  - Le vrai premier et le suivant de la sentinelle
  - Le vrai dernier est le précédent de la sentinelle
- Cela évite les problèmes avec les tests avec la valeur nil, puisqu'il y a toujours un suivant ou un précédent pour les éléments visibles dans la liste.

- Suppression d'un elt non bidon

```
précédant(suivant(elt)) ← précédant(elt)
suivant(précédant(elt)) ← suivant(elt)
```

- Marche toujours ! Plus besoin de tests !

Insertion sous condition d'un élément


```
inserer(L,elt,val)
// on suppose que L n'est pas vide
e ← suivant(premier(L));
// on cherche la position
dernier ← e;
tant que (e ≠ sentinelle(L) et donnee(e) <= val){
        dernier ← e;
        e ← suivant(e);
}
if (e = nil){
        // on insère en fin
        suivant(dernier) ← elt
        précédent(elt)← dernier
        suivant(elt)← nil
} else {
        // on insère avant e
        prec ← précédent(e)
        précédent(e) ← elt
        suivant(elt) ← e
        précédent(elt) ← prec
        if (prec = nil){
            premier(L) ← elt
        } else {
            suivant(prec) ← elt
        }
}
```

Insertion sous condition d'un élément


```
inserer(L,elt,val)
// on suppose que L n'est pas vide
e ← suivant(premier(L));
// on cherche la position
tant que(e ≠ sentinelle(L)et donnee(e)<=val){
    e ← suivant(e);
}
// on insère avant e
prec ← précédent(e)
précédent(e) ← elt
suivant(elt) ← e
précédent(elt) ← prec
suivant(prec) ← elt
```

Listes avec sentinelles

- On peut utiliser les données des sentinelles pour simplifier les algorithmes
- On traverse la liste et on s'arrête
  - Si on a trouvé un élément plus grand
  - Si on a atteint le dernier
- On peut éliminer un test en affectant à la donnée de la sentinelle, une donnée qui arrête le test.
  - On recherche un élément dont la donnée est < 9. On met 9 dans la donnée de la sentinelle

Insertion sous condition d'un élément


```
inserer(L,elt,val)
// on suppose que L n'est pas vide
e ← suivant(premier(L));
// on cherche la position
donnée(sentinelleFin(L)) ← val+1
tant que(e ≠ sentinelle(L) et donnee(e)<=val){
    e ← suivant(e);
}
// on insère avant e
prec ← précédent(e)
précédent(e) ← elt
suivant(elt) ← e
précédent(elt) ← prec
suivant(prec) ← elt
```

Insertion sous condition d'un élément


```
inserer(L,elt,val)
// on suppose que L n'est pas vide
e ← suivant(premier(L));
// on cherche la position
donnée(sentinelleFin(L)) ← val+1
tant que (donnee(e) <= val){
    e ← suivant(e);
}
// on insère avant e
prec ← précédent(e)
précédent(e) ← elt
suivant(elt) ← e
précédent(elt) ← prec
suivant(prec) ← elt
```

Listes circulaires

- Le suivant de la fin est le début
  Le précédent du début est la fin

- Une seule sentinelle est suffisant
- Quand elle est vide, elle pointe sur elle-même
  - `suivant(sentinelle)` ← `sentinelle`
  - `précédent(sentinelle)` ← `sentinelle`
- premier(L) : suivant(sentinelle)
- dernier(L) : précédent(sentinelle)
- Elt d'arrêt : sentinelle

Listes: stockage interne et externe

- Quand on crée une liste, on est face à un choix :
  - **Stockage externe** : on crée une structure spéciale pour les éléments (ListElement par exemple) qui contient les pointeurs de la liste + un pointeur vers les données
  - **Stockage interne** : on ajoute aux données les pointeurs de la liste

- **Stockage externe**

```
ListElement {
        MaClasseDonnee _obj;
        ListElement _suiv;
        ListElement _prec;
}
```

Listes: stockage interne et externe

- Quand on crée une liste, on est face à un choix :
  - **Stockage externe** : on crée une structure spéciale pour les éléments (ListElement par exemple) qui contient les pointeurs de la liste + un pointeur vers les données
  - **Stockage interne** : on ajoute aux données les pointeurs de la liste

- **Stockage interne**

```
MaClasseDonnee {
        MaDonnee _data;
        int _unEntier;
        MaClasseDonnee _suiv;
        MaClasseDonnee _prec;
}
```

Listes: stockage interne et externe

- **Stockage externe**

```
ListElement {
        MaClasseDonnee _obj;
        ListElement _suiv;
        ListElement _prec;
}
```

- Avantages :
  - Très générique
  - Indifférent aux types et à la taille des données (MaClasseDonnee peut être Object)
  - Très souple
- Inconvénients :
  - Allocation séparée
  - Code séparé (liste vs utilisation des données)

Listes: stockage interne et externe

- Stockage interne

```
MaClasseDonnee {
        MaDonnee _data;
        int _unEntier;
        MaClasseDonnee _suiv;
        MaClasseDonnee _prec;
}
```

- Avantages :
  - Accès aux données plus efficace
  - Moins de mémoire requise
  - Meilleure localité
  - Simplification de la gestion mémoire (les données des éléments de la liste sont détruites en même temps que les éléments)
- Inconvénients :
  - On doit connaître à l'avance le nombre de listes auxquelles un donnée peut appartenir simultanément
  - Perte de mémoire si on prévoit une appartenance qui n'est pas réalisée

Sdd : modifications et itérations

- De façon générale, il faut faire attention à la modification des structures de données pendant une itération
  - l'élément courant peut-être supprimé
  - le prochain peut-être supprimé
- Pas de solution générale : il faut être vigilant
- Parfois il est interdit de modifier pendant une itération. On peut alors procéder comme suit :
  - Pendant l'itération, on sauve les modifications
  - On effectue les modifications après l'itération

Listes : modifications et itérations

- Pour les listes, c'est très visible
- Ex : je supprime les éléments dont la donnée est paire. On considère une liste circulaire avec sentinelle.
- Voici **un algorithme faux** :

```
supprimerPairs(L)
    elt ←premier(L)
    tant que (elt != bidon(L)){
          if (donnee(elt) est pair){
                supprimer(elt);
          }
          elt ← suivant(elt);
    }
```

- elt a été supprimé : que vaut le suivant ?

Listes : modifications et itérations

- Pour les listes, c'est très visible
- Ex : je supprime les éléments dont la donnée est paire. On considère une liste circulaire avec sentinelle.
- Voici **un algorithme juste** :

```
supprimerPairs(L)
    elt ←premier(L)
    tant que (elt != bidon(L)){
          suiv ← suivant(elt);
          if (donnee(elt) est pair){
                supprimer(elt);
          }
          elt ← suiv;
    }
```

- On prend le suivant avant la suppression

Sdd : intégrité des données

- Il est important que l'organisation interne des données dans une Sdd soit cohérente.
- Certaines propriétés doivent être respectées
- Ces propriétés sont dépendantes de la Sdd
- Pour une liste doublement chaînée circulaire avec sentinelle, il est important que tout élément elt de la liste vérifie
  - précédent (suivant(elt)) = elt
  - suivant(précédent(elt)) = elt

Sdd : intégrité des données

- Afin d'assurer le bon fonctionnement interne de la Sdd on a intérêt
  - à éviter d'utiliser la structure interne de la Sdd
  - à préférer utiliser des accesseurs/modificateurs portant sur la Sdd et non sur ses éléments
- Cas de liste :
  - Excepté pour définir des fonctions internes de la Sdd, comme supprimer, ajouter, rechercher etc… :
    - on n'utilisera pas les primitives suivant(elt), précédent(elt) …
    - on préférera toujours passer par la structure de données : supprimer (L,elt), ajouerEnTete(L,elt) etc…

Sdd : intégrité des données

- L'utilisation de fonction impliquant toujours la structure de données se contrôle aisément avec les langages à objets
- Aucun modificateur et certains accesseurs de la classe EltList ne sont accessibles. Seule la liste a le droit d'accéder à ces méthodes
- La class List fournit l'API (advance programming interface) de la classe List et on doit passer par elle