

TC Informatique

PC N° 4
30 Novembre 2000

François Sillion

Structures dynamiques Listes chaînées

<http://w3.edu.polytechnique.fr/informatique>

Structures de données abstraites

- Représenter
 - un *ensemble* d'éléments
 - Muni de certaines *opérations*
 - Vérifiant certaines *propriétés*
- Exemple d'une *pile* (LIFO = Last In, First Out)
- Opérations :

```
static Pile CréePile();  
static Pile Empile( Pile p, Element e);  
static Element Sommet( Pile p );  
static Pile Depile( Pile p );  
static boolean PileVide( Pile p );
```

- Propriétés :

PileVide(CréePile()) = true

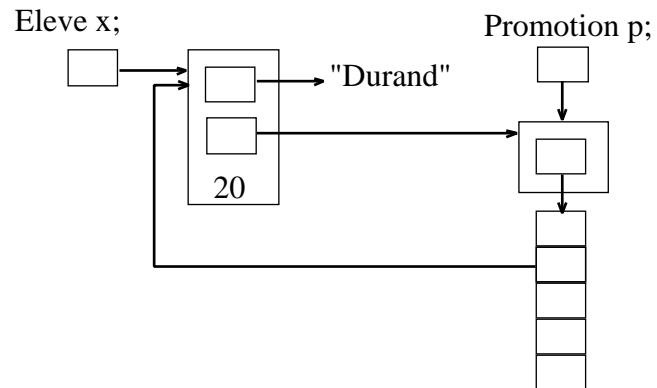
PileVide(Empile(p, e)) = false

Sommet(Empile(p, e)) = e

Depile(Empile(p, e)) = p

Rappel sur les objets « complexes »

```
Class Eleve {          Class Promotion {  
    String nom;        Eleve [] eleves;  
    Promotion p;      };  
    int age;  
};
```



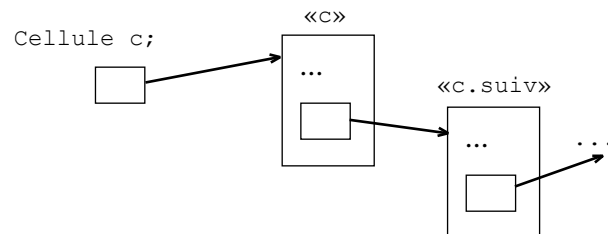
- Pas de restriction sur la nature des champs
- Objet = suite d'octets contigus en mémoire
- Les chaînes de caractères sont des objets

Structures de données dynamiques

- Les tableaux sont trop limités
 - Taille fixe
 - Structure fixe (linéaire)
- Structures plus flexibles ou complexes, de taille variable
 - Utilisation d'objets de type T contenant des références à d'autres objets du même type
 - Allocation de mémoire en fonction des besoins, i.e. *dynamique*

Structures de données dynamiques

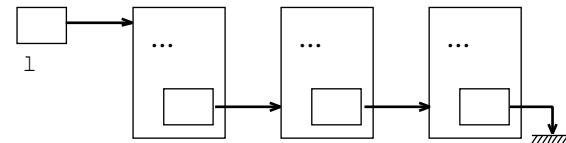
```
Class cellule {  
    ...  
    cellule suiv;  
};
```



- Exemples

- Listes (piles, files d'attente, listes de priorité)
- Arbres (arbres binaires de recherche)
- Graphes (plan de métro)

Liste chaînée



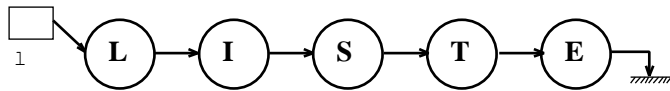
```
Class Liste {  
    int val;  
    Liste suiv;  
};
```

- Représentation de la liste vide

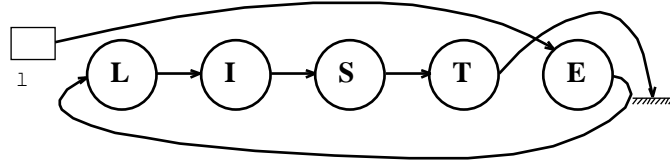
- Constante prédéfinie `null` (polymorphisme)
- Cellule particulière (variable globale)

```
Static Liste empty = new Liste();
```

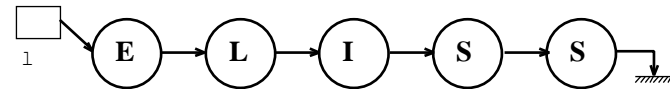
Liste chaînée : avantages



- Taille arbitraire (mais accès séquentiel)
- Facilité des opérations sur des données existantes
 - Exemple de reconfiguration : placer le dernier élément en tête de liste.



– Représentation identique :



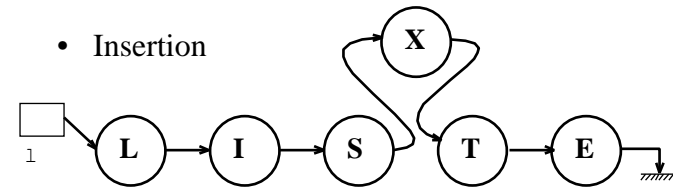
PC 4

François Sillion

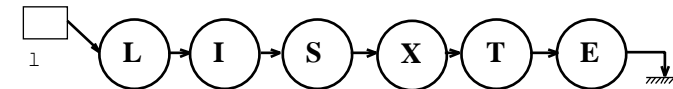
7

Liste chaînée : avantages

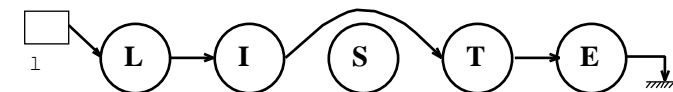
- Insertion



– Représentation identique :



- Suppression



PC 4

François Sillion

8

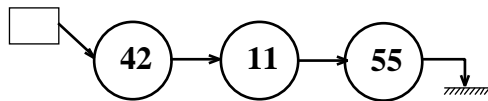
Liste chaînée de valeurs entières

```
public class Liste {
    int val;
    Liste suiv;

    Liste ( int i, Liste suite ) {
        val = i;
        suiv = suite;
    }
    static boolean EstVide( Liste l ) {
        return l == null;
    }
};
```

- **Insertion en tête**

```
New Liste(42, new Liste(11, new Liste(55,null)))
```



Utilisation en pile

- **Fonctionnalités de pile**

```
Static Liste Empile( int val, Liste l ){
    return new Liste( val, l );
}
Static Liste Depile( Liste l ) {
    return l.suiv;
}
Static int Sommet( Liste l ) {
    return l.val;
}
Static Liste CreePile() {
    return null;
}
Static boolean PileVide( Liste l ) {
    return l == null;
}
```

Utilisation en table

- Recherche (réursive)

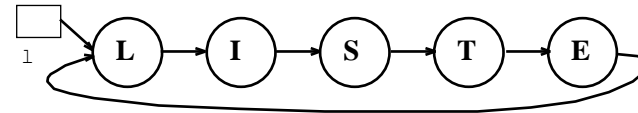
```
Static boolean Cherche( int val, Liste l ){
    if ( l == null ) {
        return false;
    } else if ( val == l.val ) {
        return true;
    } else {
        return Cherche( val, l.suiv );
    }
}
```

- Suppression (réursive)

```
Static Liste Supprime( int val, Liste l ) {
    if ( l == null ) {
        return l;
    } else if ( val == l.val ) {
        return l.suiv;
        // return Supprime( val, l.suiv );
    } else {
        l.suiv = Supprime( val, l.suiv );
        return l;
    }
}
```

Variations

- Liste circulaire



- Liste doublement chaînée

```
public class Liste {
    int val;
    Liste suiv;
    Liste prev;
    ...
}
```

- Utile lorsque l'on arrive sur un élément quelconque de la liste, et que l'on souhaite pouvoir supprimer

Listes et récursivité

- Impression d'une liste

- À l'endroit

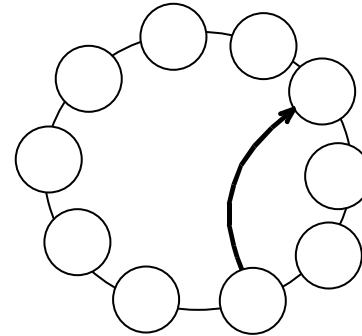
```
static void ImprimeEndroit( Liste l ){
    if ( l != null ) {
        write( l.val );
        ImprimeEndroit( l.suiv );
    }
}
```

- À l'envers

```
static void ImprimeEnvers( Liste l ){
    if ( l != null ) {
        ImprimeEnvers( l.suiv );
        write( l.val );
    }
}
```

Problème de Josephus

- Formulation macabre: suicide collectif



```
// exécution des éléments, un par un
while ( t != t.suiv ) {
    for ( i=1; i < M ; i++ ) t = t.suiv;
    writeln( t.suiv.val );
    t.suiv = t.suiv.suiv;
}
writeln( t.val );
```

Recherche

- Recherche séquentielle

- Tableau

```
for ( int i=0; i < a.length ; ++i )  
    if ( a[i] == val ) return true;
```

- Liste

```
while ( (t != null) && (t.val != val) )  
    t = t.suiv;  
return ( t != null );
```

- Complexité linéaire

Recherche dichotomique

- Travail dans un tableau, supposé trié

```
public static boolean cherche  
    ( int a[], int val, int g, int d ){  
    if ( g < d ) {  
        int m = (g+d)/2;  
        if ( val < a[m] ){  
            return cherche(a, val, g, m);  
        } else {  
            return cherche(a, val, m, d);  
        }  
    } else {  
        return ( a[g] == val );  
    }  
}
```

Appel de `cherche(a, val, 0, a.length)`

Recherche dichotomique

- Version non récursive

```
public static boolean cherche
    ( int a[], int val, int g, int d ){
    while ( g < d ) {
        int m = (g+d)/2;
        if ( val < a[m] ){
            d = m;
        } else {
            g = m;
        }
    }
    return ( a[g] == val );
}
```

Principe du hachage

- Recherche d'éléments quelconques (chaines de caractère...)
- On suppose qu'on dispose d'une fonction *h* qui transforme les objets recherchés en un (petit) nombre entier.
- Compromis entre temps de calcul et espace

```
public static boolean cherche
    ( int a[], Chose truc ){
    int key = h(truc);
    if ( a[key] == null ) {
        return false;
    } else if ( comparer( a[key], truc ) ) {
        return true;
    } else
        ...
}
```

Hachage avec tableau fini

- Adressage ouvert
 - En cas de collision, stocke dans la case suivante

```
public static boolean cherche
    ( int a[], Chose truc ){
    int key = h(truc) % a.length;
    while ( a[key] != null ) {
        if ( comparer( a[key], truc ) ) {
            return true;
        } else {
            key = (key+1) % a.length;
        }
    }
    return false;
}
```