STRUCTURES DE DONNÉES ET ALGORITHMES

Esial 1ère année

Année 2006-2007

Plan

- Conception d'une solution
- Type Abstrait
- Type Liste
- Type Pile
- Type File

Conception d'une solution

- Formuler le problème
  - modélisation mathématique
  - algorithme informel
- Spécifier les données
  - types de données abstraits
  - algorithme pseudo-langage
- Construire une solution
  - structures de données et programme

Exemple : corriger des copies
Formuler le problème

- faire un corrigé des exercices
- faire un tas de copies
- pour chaque copie du tas, corriger et noter chaque réponse en fonctions du corrigé
- mettre la copie sur un au tas
- calculer la moyenne du nouveau tas

Spécifier les données

- utilisation d’une liste d ’exercices corrigés
- utilisation d’une pile de copies
- possibilité d’enlever une copie de la pile
- possibilité d’accéder aux corrigés de la liste de corrigés

Algorithme pseudo langage (1)

```
corriger(c : Copie) = note : Entier
        total ← 0
        L ← exercices_corrigés
        pourchaque élément de exercices_rendu(c)
            chercher l'élément correspondant dans L
            comparer et noter l 'exercice
            total ← total + note
        finpourchaque
        note ← total
```

Algorithme pseudo langage (2)

```
corriger(tas : Pile) = nouveau_tas : Pile
    tantque ¬vide(tas)
        copie ← dépiler(tas)
        note ← corriger(copie)
        recopier la note dans la base de notes
        ajouter(nouveau_tas,copie)
    fintantque
```

Construire une solution

```
List exercicesRendus(List ex) {
      List corrigé;
      for(Iterator It=ex.iterator() ; It.hasNext() ; ) {
         chercher( corrigé, it.next() )
         ...
      }
      return corrigé;
}
```

Type Abstrait

Définition

- Les données sont considérées de manière abstraite
- On se donne :
  - une notation
  - un ensemble d'opérations
  - les propriétés de ces opérations
- Voir cours IB1

Type Liste

Séquence finie, éventuellement vide, d’éléments de même type

Exemples de listes

- Gagnants au Loto dans l'amphi
  - ()
- Cours IB
  - (IB1, IB2, IB3)
- Nombres premiers inférieurs à 20
  - (2,3,5,7,11,13,17,19)
- chaîne de caractères
  - « hello »
  - Peut être vue comme une liste de caractères
  - ('h', 'e', 'l', 'l', 'o')

Terminologie

- longueur : nombre d'éléments composant la liste
- une liste de longueur zéro est une liste vide
- une liste non vide comprend un premier élément appelé tête.
- Le reste de la liste est appelé queue

Terminologie

- À chaque élément d'une liste est associée une position (un entier)
- une sous-liste est une liste composée des éléments dont la position $\in$ [i, j]
- un préfixe est une sous-liste qui commence au début de la liste
- un suffixe est une sous-liste qui se termine à la fin de la liste

Exercice : (2,7,1,8,2)

- Longueur 5
- tête 2
- queue (7,1,8,2)
- préfixes (), (2), (2,7), (2,7,1), ..., (2,7,1,8,2)
- suffixes (2,7,1,8,2), (7,1,8,2), …, (8,2), (2), ()
- sous-listes (), (2), (7), …, (2,7), (7,1), …, (2,7,1),...

Opérations sur les listes

Sorte Liste

Opérations

liste_vide	:	→ Liste
appartient	:	Liste x Elément → Booléen
supprimer	:	Liste x Elément → Liste
longueur	:	Liste → Entier
est_vide	:	Liste → Booléen
ième	:	Liste x Entier → Elément
concaténer	:	Liste x Liste → Liste

Opérations (suite)

tête : Liste → Elément

queue : Liste → Liste

chercher : Liste x Elément → Entier

insérer_tête : Liste x Elément → Liste

insérer_queue: Liste x Elément → Liste

insérer : Liste x Entier x Elément → Liste

...

Axiomatisation

- Choix des constructeurs

  liste_vide : → Liste

  cons : Elément x Liste → Liste

- Les constructeurs sont les opérateurs élémentaires permettant de construire une liste
- Cela correspond à la création d'un objet en Java
- Et plus généralement à de l'allocation mémoire

Définition des fonctions

longueur(liste_vide) = 0

longueur(cons(e,L)) = 1 + longueur(L)

- Exemple :

  longueur(cons(a,cons(b,liste_vide)))

  1 + longueur(cons(b,liste_vide))

  1 + 1 + longueur(liste_vide)

  1 + 1 + 0

  2

Autre exemple

appartient(liste_vide, e) = faux
appartient(cons(e,L), f) = si e=f
alors vrai
sinon appartient(L, f)

appartient(cons(1,cons(2,liste_vide)), 2)
appartient(cons(2,liste_vide), 2)
vrai

Axiomatisation (suite)

tête(cons(e,L)) = e
queue(cons(e,L)) = L

concaténer(liste_vide, L') = L'
concaténer(cons(e,L), L') = cons(e, concaténer(L,L'))

insérer_tête(L,e) = cons(e,L)
insérer_queue(L,e) = concaténer(L,cons(e,liste_vide))

ième(cons(e,L), 0) = e
ième(cons(e,L), i) = ième(L, i-1)
si $1 \leq i \leq$ longueur(L)

…

Passage à l’implantation

Interface List

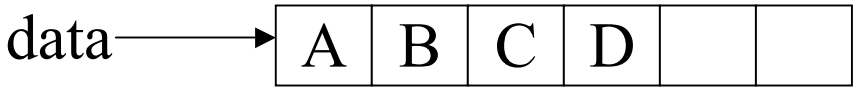
```
public interface List extends Collection {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    boolean add(Object o);
    boolean remove(Object o);
    Object get(int index);
    Object set(int index, Object element);
    Object add(int index, Object element);
    int indexOf(Object o);
    …
}
```

Comment représenter une liste?

- Par cons et liste_vide ?
- Par une liste chaînée ?
- Par un tableau ?
- Par une liste doublement chaînée ?
- Par un tableau et des index vers les suivants ?
- …

Implantation par un tableau

- data : tableau mémorisant les éléments
- length : capacité courante
- size : longueur de la liste



size = 4

length = 6

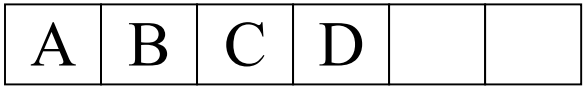
Classe ArrayList

```
public class ArrayList extends AbstractList implements List … {
    private Object data[];
    private int size;
    …
}
```

Recherche séquentielle

```
public int indexOf(Object e) {
    for (int i = 0; i < size; i++)
      if (equals(e, data[i]))
        return i;
    return -1;
  }

private boolean equals(Object o1, Object o2)  {
  return o1 == null ? o2 == null : o1.equals(o2);
}
```



indexOf(C)

i = 0 cond = faux

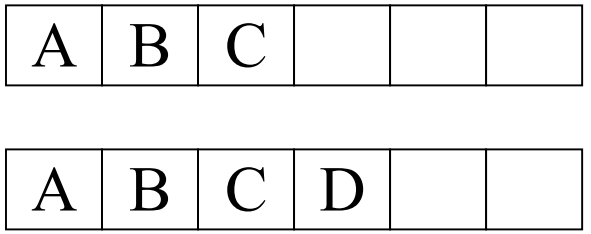
i = 1 cond = faux

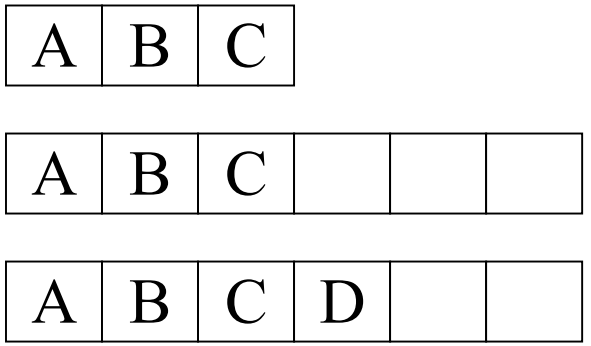
i = 2 cond = vrai

Result = 2

Insertion en queue : add(D)

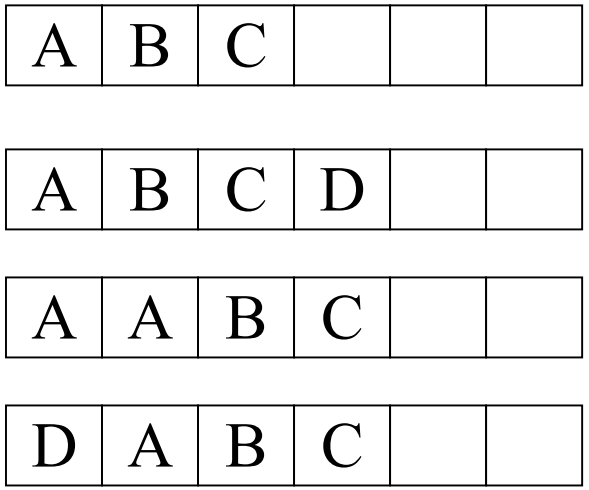
```
public boolean add(Object e) {
   if (size == data.length)
     ensureCapacity(size + 1);
   data[size++] = e;
   return true;
 }
 public void ensureCapacity(int minCapacity) {
   int current = data.length;
   if (minCapacity > current) {
      Object[] newData =
           new Object[Math.max(current * 2, minCapacity)];
      System.arraycopy(data, 0, newData, 0, size);
      data = newData;
   }
 }
```





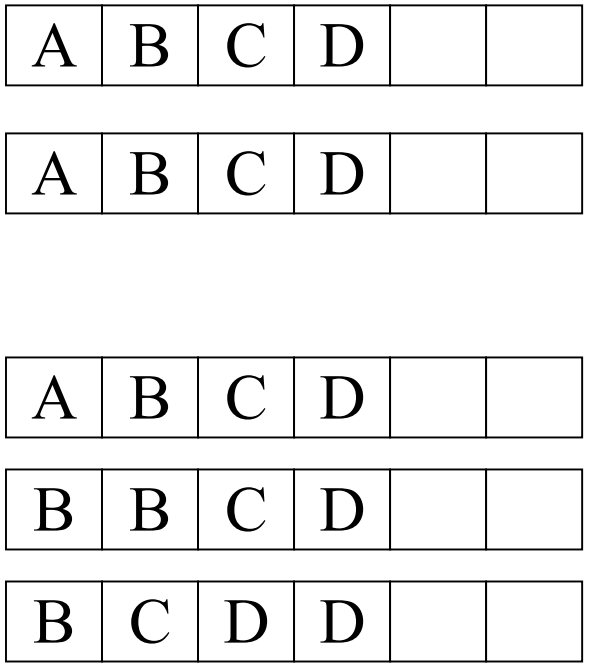
Insertion en tête : add (0,D)

```
public void add(int index, Object e) {
  checkBoundInclusive(index);
  if (size == data.length)
    ensureCapacity(size + 1);
  if (index != size)
    System.arraycopy(data, index,
                     data, index + 1, size - index);
  data[index] = e;
  size++;
}
private void checkBoundInclusive(int index) {
   if (index > size) {
     throw new IndexOutOfBoundsException(
     "Index: " + index + ", Size: " + size);
   }
}
```



Suppression

```
public Object remove(int index) {
   checkBoundExclusive(index);
   Object r = data[index];
   if (index != --size)
     System.arraycopy(data, index + 1,
                              data, index, size - index);
   // Aid for garbage collection
   data[size] = null;
   return r;
 }
private void checkBoundExclusive(int index) {
  if (index >= size)
     throw new IndexOutOfBoundsException(
     "Index: " + index + ", Size: " + size);
 }
```

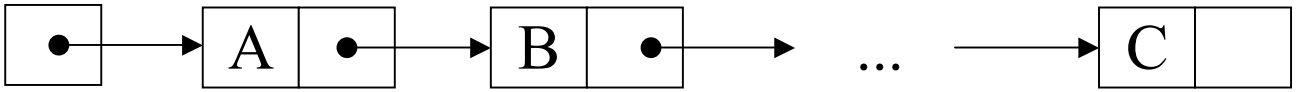


Résumé

- Recherche : linéaire
- Insertion en queue : temps constant
- Insertion en tête : linéaire
- Suppression en queue : temps constant
- Suppression en tête : linéaire

Implantation par une liste chaînée

- Liste
  - first : première cellule de la liste
  - size : taile de la liste
- Cellule
  - data : élément de la cellule
  - next : prochaine cellule

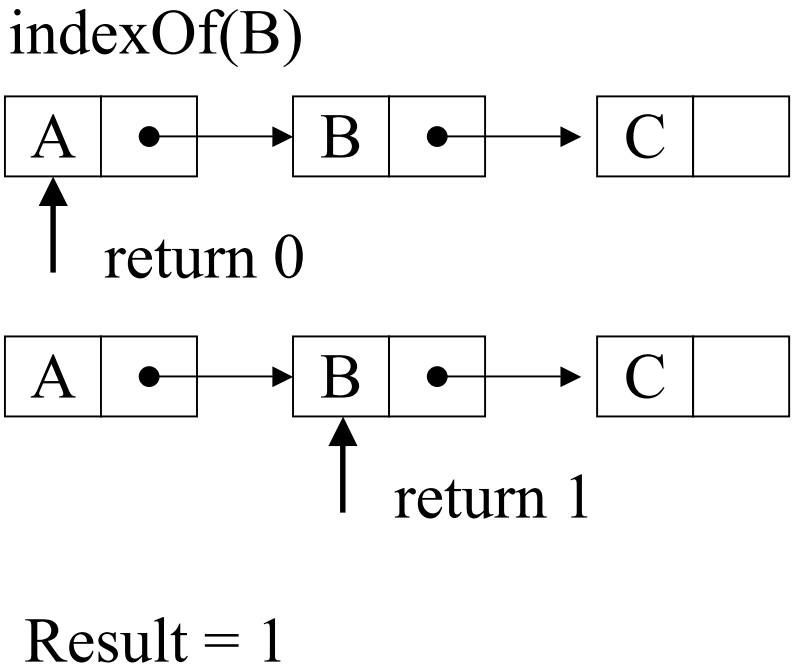


Classes LinkedList et Entry

```
public class LinkedList extends AbstractSequentialList implements List {
  private Entry first;
  private int size = 0;
  public int indexOf(Object o) { … }
}
private static class Entry {
  Object data;
  Entry next;
  Entry(Object data) {
    this.data = data;
  }
}
```

Recherche séquentielle

```
public int indexOf(Object o) {
  int index = 0;
  Entry e = first;
  while (e != null) {
      if (equals(o, e.data)) {
        return index;
      }
      index++;
      e = e.next;
    }
  return -1;
}
```

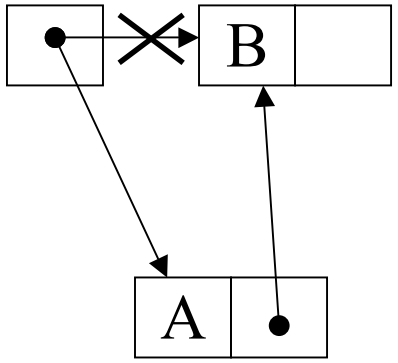


Insertion en tête

```
public void addFirst(Object o) {
   Entry e = new Entry(o);
   if (size == 0) {
     first = e;
   } else {
     e.next = first;
     first = e;
   }
   size++;
}
```

addFirst(B)

addFirst(A)

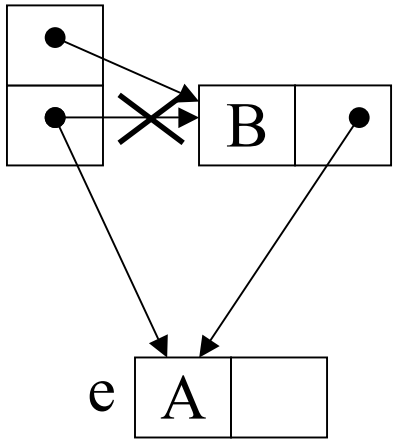


Insertion en queue

```
private void addLast(Object o) {
   Entry e = new Entry(o);
   if (size == 0) {
     first = last = e;
   } else {
     last.next = e;
     last = e;
   }
   size++;
 }
```

addLast(B)

addLast(A)

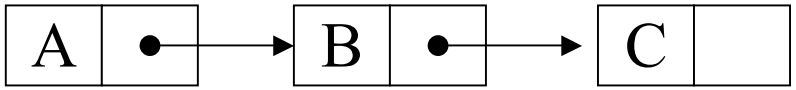


Résumé

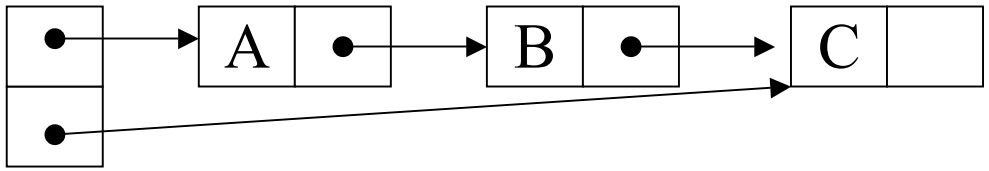
- Recherche : linéaire
- Insertion en queue : temps constant
- Insertion en tête : temps constant
- Suppression en queue : temps constant
- Suppression en tête : temps constant

Question

- Peut-on se passer de LinkedList et utiliser directement Entry ?

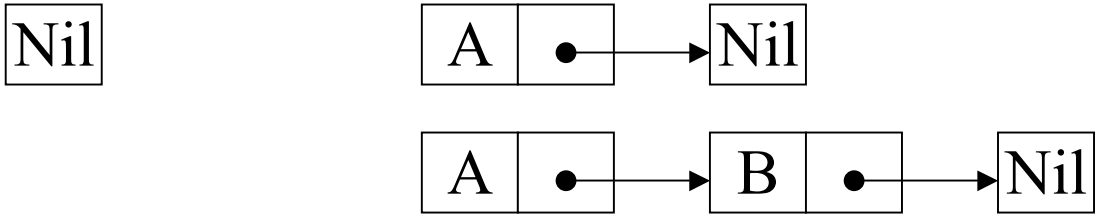


A la place de :



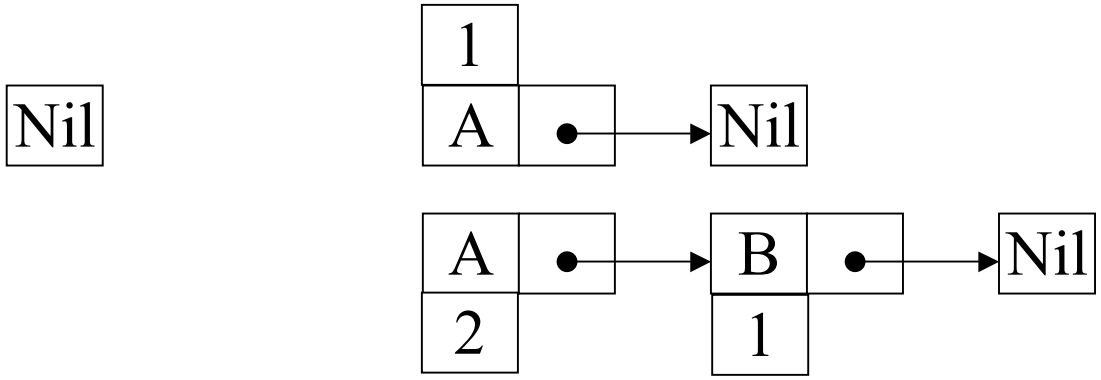
Questions

- Comment représenter la liste vide ?
- Comment représenter une liste avec 1 élément ?
- Comment représenter une liste avec 2 éléments ?



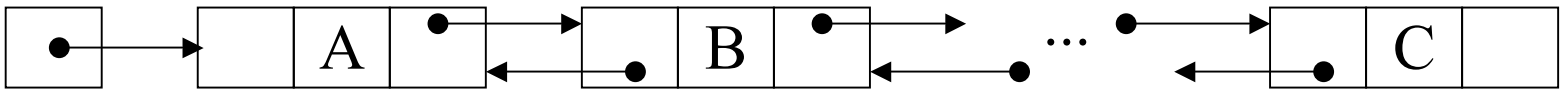
Intérêts / Problèmes

- Ressemble aux types abstraits (head,tail)
- Accès à la $i^{ème}$ position peu efficace
- Nombre d'éléments peu efficace
- On pourrait stocker de l'information dans les cellules, mais c'est peu pratique



Double chaînage

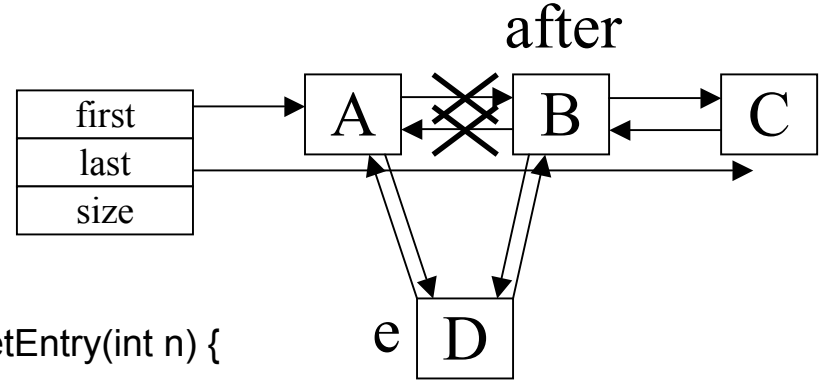
- Liste
  - first : première cellule de la liste
  - last : dernière cellule de la liste
  - size
- Cellule
  - data : élément de la cellule
  - next : prochaine cellule
  - previous : cellule précédente



Insertion au milieu

```
public void add(int index, Object o) {
  checkBoundsInclusive(index);
  Entry e = new Entry(o);
  if (index < size) {
     Entry after = getEntry(index);
     e.next = after;
     e.previous = after.previous;
     if (after.previous == null) {
       first = e;
     } else {
       after.previous.next = e;
     }
     after.previous = e;
     size++;
  } else {
    addLastEntry(e);
  }
}
```

addLast(D,1)




```
Entry getEntry(int n) {
  Entry e;
  if (n < size / 2) {
     e = first; // n less than size/2, iterate from start
     while (n-- > 0) { e = e.next; }
  } else {
     e = last; // n greater than size/2, iterate from end
     while (++n < size) { e = e.previous; }
    }
  return e;
}
```

Résumé

- Ajout et suppression en tête ou queue presque aussi efficace que dans le cas simplement chaîné
- Avantages :
  - Déplacement de la droite vers la gauche
  - Accès indexé plus efficace
- Inconvénients
  - Taille des cellules plus importante
  - Administration des pointeurs plus complexe

Pour en finir avec les listes

- Les types abstraits permettent de raisonner
- Exemples de termes

```
t_1 =   insérer_tête(
                insérer_tête(liste_vide,2),
                1)
t_2 =   insérer_ queue(
                insérer_ queue(liste_vide,1),
                2)
```

- A-t-on t_1 = t_2 ?

Preuve

t_1 = insérer_tête(insérer_tête(liste_vide,2),1)
= insérer_tête(cons(2,liste_vide),1)
= cons(1,cons(2,liste_vide))

- insérer_queue(liste_vide,e) = cons(e,liste_vide)
- insérer_queue(cons(e,L),e') = cons(e, insérer_queue(L,e')

t_2 = insérer_ queue(insérer_ queue(liste_vide,1),2)

insérer_ queue(liste_vide,1) = cons(1,liste_vide)

= insérer_ queue(cons(1,liste_vide),2)
= cons(1,insérer_queue(liste_vide,2))
= cons(1,cons(2,liste_vide)))

- d 'où t_1 = t_2

Type Pile

LIFO : Last In, First Out

Les insertions et les suppressions se font à une seule extrémité : le sommet

Opérations sur les piles

Sorte Pile

Opérations

pile_vide	:	→ Pile
sommet	:	Pile → Elément
empiler	:	Pile x Elément → Pile
dépiler	:	Pile → Pile
est_vide	:	Pile → Booléen

Axiomatisation

- Constructeurs

  pile_vide : → Pile

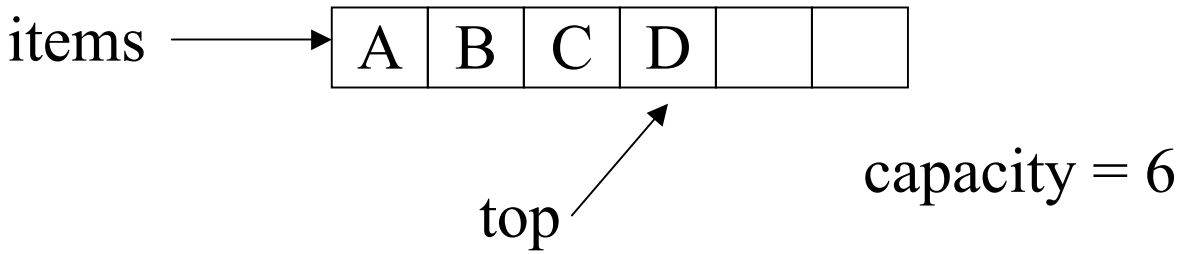
  empiler : Pile x Elément → Pile

- Définition des fonctions

  sommet(empiler(P,e)) = e
  dépiler(empiler(P,e)) = P
  est_vide(pile_vide) = vrai
  est_vide(empiler(P,e))= faux

Implantation par un tableau

- items : tableau mémorisant les éléments
- capacity : capacité courante
- top : sommet de la pile



Implantation

- D’une manière générale, on peut implanter les piles en utilisant des listes
  - Représentées par un tableau
  - Chaînées
- Intérêts
  - Réduit les efforts d’implantation
  - Permet d’aller lire dans la pile
  - En Java, la classe Stack est implantée en utilisant Vector (version synchronisée de ArrayList)

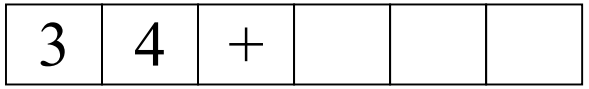
Exemples d'utilisation de piles

- évaluation d'expression
  - (3+4)*2 se note également 3 4 + 2 *

empiler(3), évaluer(sommet())

empiler(4), évaluer(sommet())

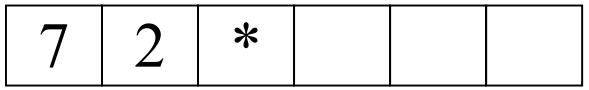
empiler(+), évaluer(sommet())



dépiler(), a ← sommet(), depiler()

b ← sommet(), depiler()

empiler(a+b)



empiler(2), ...

Comparaison

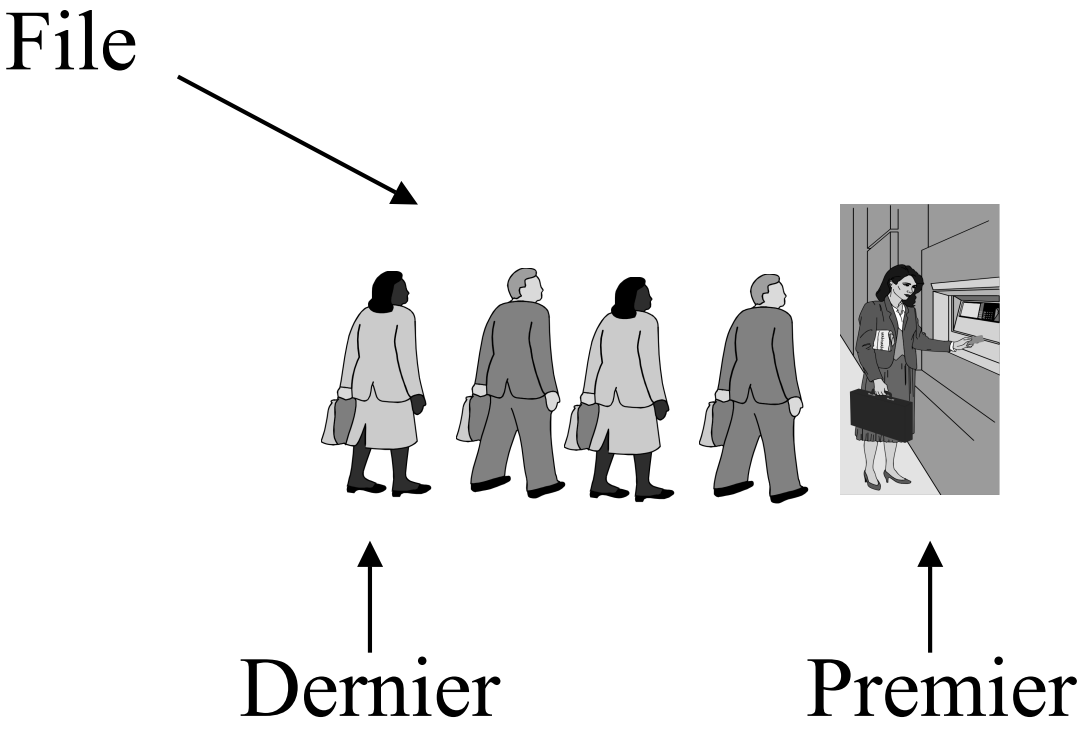
- Liste chaînée
  - extension peu coûteuse
  - pas de déplacement des éléments
- Tableau
  - gain d'espace
  - mémoire consécutive
- Le choix dépend de l'utilisation
  - connaît-on la taille maximale de la pile ?
  - veut-on une efficacité maximale ?

Type File

FIFO : First In, First Out

Les adjonctions se font à une extrémité et les suppressions à l’autre

Exemple



Opérations sur les files

Sorte File

Opérations

file_vide	:	→ File
premier	:	File → Elément
ajouter	:	File x Elément → File
retirer	:	File → File
est_vide	:	File → Booléen

Axiomatisation

- Constructeurs

  file_vide : → File

  ajouter : File x Elément → File

- Définition des fonctions

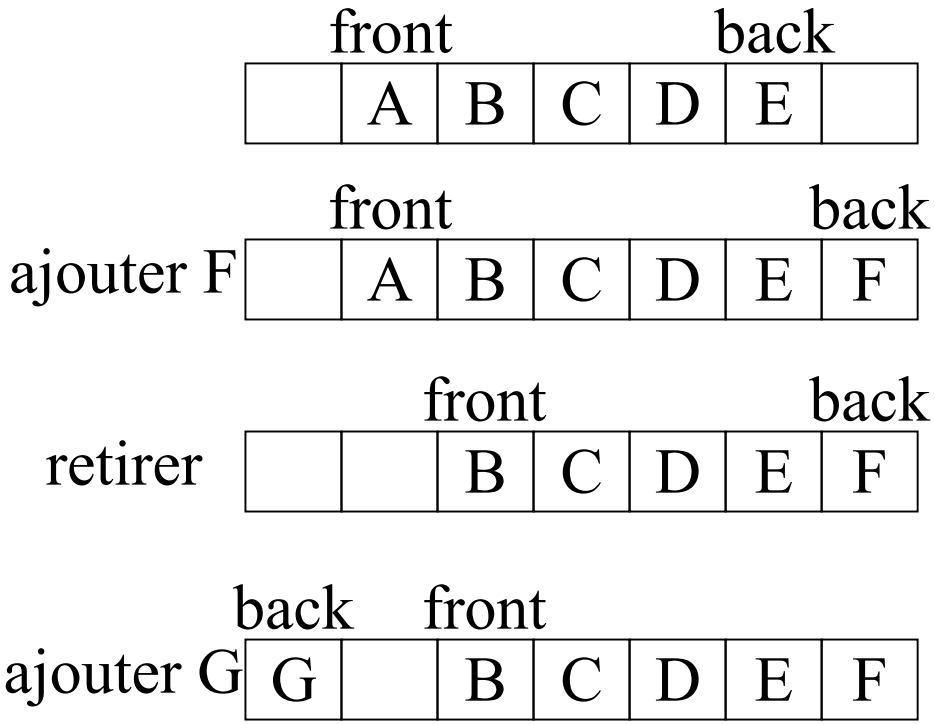
premier(ajouter(F,e)) = e si est_vide(F) = vrai
premier(ajouter(F,e)) = premier(F) si est_vide(F) = faux
retirer(ajouter(F,e)) = file_vide si est_vide(F) = vrai
retirer(ajouter(F,e)) = ajouter(retirer(F),e)
si est_vide(F) = faux

Implantation

- On peut utiliser LinkedList, en se limitant à :
  - addLast
  - removeFirst
- On peut également utiliser ArrayList, mais :
  - remove(0) est une opération coûteuse
  - Il faut décaler tous les éléments vers la gauche
- Y-a-t-il une autre implantation possible ?
- Oui, les Queue (depuis Java 1.5)

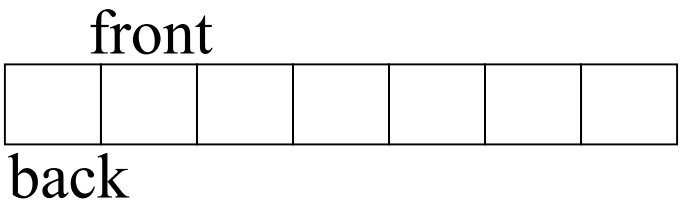
Implantation par un tableau

- front : index de la tête de la file
- back : index du dernier élément de la file

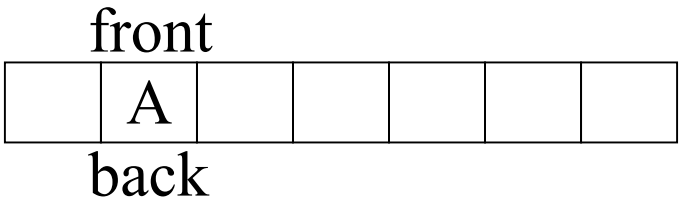


Implantation par un tableau

```
void wipeOut() {
  front = 1;
  back = 0;
}
```




```
void enqueue(Object o) {
  back = (back+1) % size;
  items.set(back, o);
}
```




```
boolean isEmpty {
  return (front == back+1) % size);
}
```

```
boolean isFull() {
  return (front == (back+2) % size);
}
```

Exercice

- Implanter les Queue
- Comparer les performances avec
  - Les LinkedList
  - Les ArrayList

The End