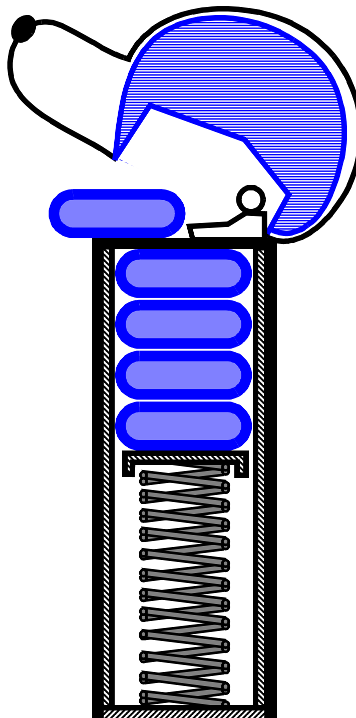
PILES, FILES ET LISTES CHAÎNÉES

- Types abstraits de données (TAD)
- Piles
- Exemple: Analyse boursière
- Files
- Listes chaînées
- Files à deux bouts (*deques*)

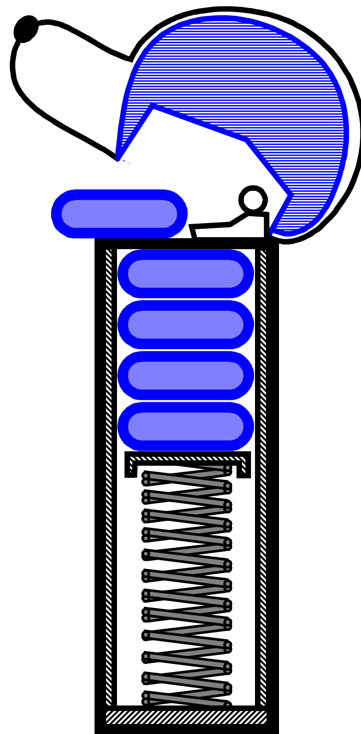


Types abstraits de données (TAD)

- Un **type abstrait de données** (*Abstract Data Type —ADT*) est une abstraction de structure de données: aucun codage n'est impliqué.
- Un TAD spécifie:
  - ce qui est contenu dans le TAD
  - les opérations qui peuvent être effectuées sur ou par le TAD.
- Par exemple, si nous cherchons à modéliser un sac de billes avec un TAD, nous pourrions spécifier que:
  - ce TAD contient des billes
  - ce TAD supporte l'insertion d'une bille et le retrait d'une bille.
- Il y a beaucoup de TAD standards et formalisés. Un sac de billes n'est pas l'un d'entre eux.
- Dans ce cours, nous apprendrons différents TAD standards (piles, files, listes...).

Piles (*Stacks*)

- Une pile est un contenant pour des objets insérés et retirés selon le principe dernier entré, premier sorti (*last-in-first-out*, ou *LIFO*).
- Les objets peuvent être insérés à tout moment, mais seulement le dernier (le plus récemment inséré) peut être retiré.
- Insérer un item correspond à empiler l'item (*pushing*). Dépiler la pile (*popping*) correspond au retrait d'un item.
- Analogie: distributeur de bonbons PEZ®

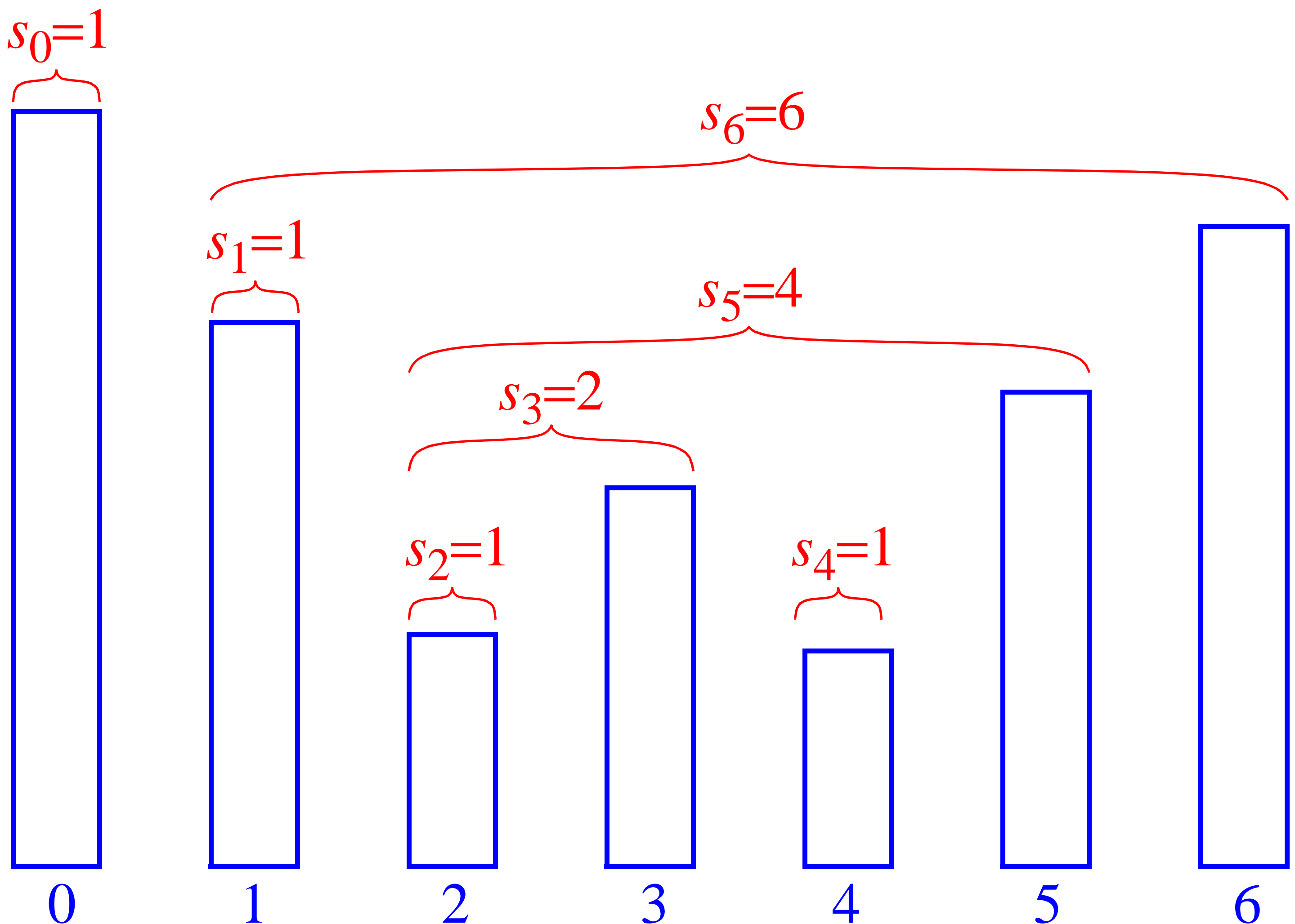


Le TAD Pile (ou Stack)

- Une pile est un type abstrait de données (TAD) qui supporte deux méthodes principales:
  - push(*o*): Insère l'objet *o* sur le dessus de la pile.
  - pop(): Retire l'objet du dessus de la pile et retourne-le; si la pile est vide, alors une erreur survient.
- Les méthodes secondaires suivantes devraient aussi être définies:
  - size(): Retourne le nombre d'objets dans la pile.
  - isEmpty(): Retourne un booléen indiquant si la pile est vide.
  - top(): Retourne l'objet du dessus de la pile, sans le retirer; si la pile est vide, alors une erreur survient.

Exemple

- L'**étendue** (*span*) du prix d'une action à un certain jour, *d*, est le nombre maximum de jours consécutifs (jusqu'à aujourd'hui) où le prix de l'action a été plus bas ou égal à son prix au jour *d*.



Un algorithme inéfficace

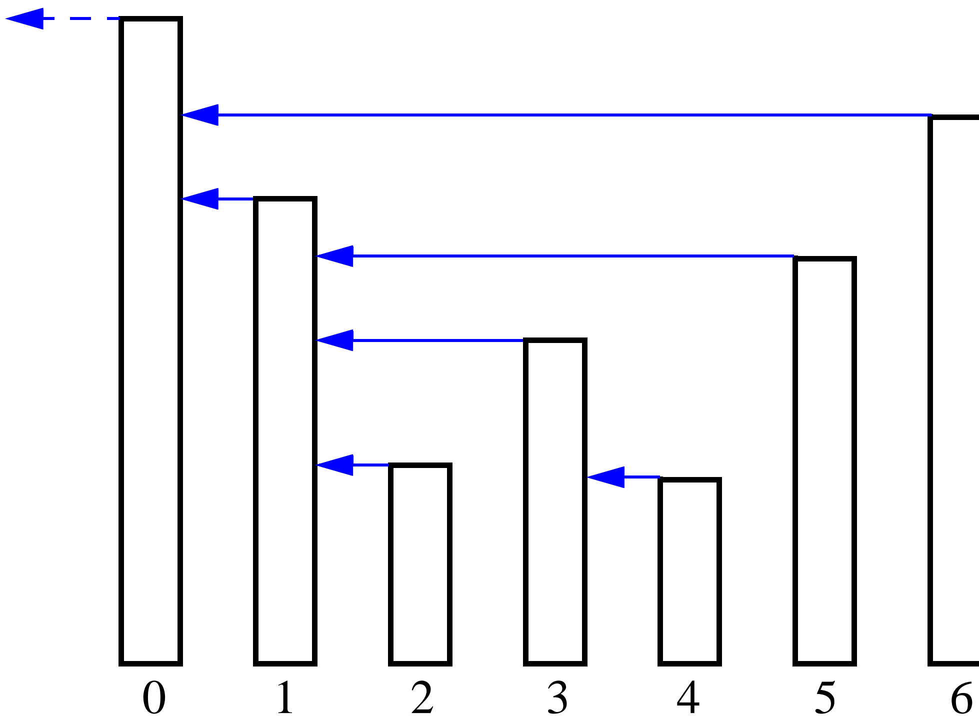
- Il y a une façon directe de calculer l'étendue d'une action à un jour donné pour n jours:

```
Algorithm computeSpans1(P):
  Entrée: Un vecteur de nombres P à n éléments.
  Sortie: Un vecteur de nombres A à n éléments tel que
              S[i] est l'étendue de l'action au jour i
  Soit S un vecteur de n nombres
  for i=0 to n−1 do
     k ←0
     done←false
     repeat
        if P[i-k] ≤P[i] then
           k←k+1
        else
           done←true
     until (k=i) or done
     S[i]←k
  return array S
```

- Le temps d'exécution de cet algorithme est (ouf!) $O(n^2)$. Pourquoi?

Une pile peut aider!

- Nous voyons que s_i au jour i peut être calculé facilement si nous connaissons le jour le plus proche avant i où le prix est plus haut lors de ce jour que le prix au jours i. Si un tel jour existe, appelons-le h_i.
- L'étendue est maintenant définie par $s_i = i - h_i$



Nous utilisons une ***pile*** pour calculer h_i

Étude de cas: Une *applet* pour analyse boursière (suite)

- Le pseudo-code pour notre nouvel algorithme:

```
Algorithm computeSpan2(P):
  Entrée: Un vecteur de nombres P à n éléments.
  Sortie: Un vecteur de nombres A à n éléments tel que
          S[i] est l'étendue de l'action au jour i
  Soit S un vecteur de n nombres et D une pile vide
  for i=0 to n-1 do
    done←false
    while not(D.isEmpty() or done) do
      if P[i]≥P[D.top()] then
        D.pop()
      else
        done←true
      if D.isEmpty() then
        h← -1
      else
        h←D.top()
      S[i]←i-h
      D.push(i)
    return array S
```

- Analysons le temps d'exécution de computeSpan2...

À propos de Java

- Étant donné le TAD pile, nous devons coder cet ADT afin de l'utiliser dans nos programmes.
- Vous devez comprendre deux concepts de programmation: les **interfaces** et les **exceptions**.
- Une interface est une façon de déclarer ce qu'une classe peut faire. Elle n'indique pas comment le faire.
- Pour une interface, vous écrivez simplement les noms de méthodes et leurs paramètres. Ce qui est important dans un paramètre est son **type**.
- Plus tard, quand vous écrirez une classe pour cette interface, vous coderez alors le contenu de ces méthodes.
- Séparer l'interface de la réalisation est une technique de programmation très utile. Exemple d'interface:

```
public interface radio {
   public void play();
   public void stop();
}
```

Une interface de pile en Java

- Même si la structure de donnée pile est déjà incluse comme classe Java dans le "*package*" java.util, il est possible, et parfois même préférable, de définir votre propre pile spécifique, comme ceci:

```
public interface Stack {

// accessor methods

  public int size(); // return the number of
                     // elements in the stack

  public boolean isEmpty();  // see if the stack
                             // is empty

  public Object top() // return the top element

    throws StackEmptyException; // if called on
                                // an empty stack


// update methods

  public void push (Object element); // push an
        // element onto the stack. Note that
        // the type of the parameter is
        // specified as an Object

  public Object pop() // return and remove the
                      // top element of the stack
    throws StackEmptyException; // if called on
                                // an empty stack

}
```

Exceptions

- Les **exceptions** sont un autre concept de programmation très utile, surtout dans un contexte de gestion d'erreurs.
- Quand vous détectez une erreur (ou un cas ***exception*el**), vous lancez (***throw***) une exception.
- Exemple

```
public void mangePizza() throws MalAuVentreException
{
   ...

   if (tropMangé)
      throw new MalAuVentreException("Ouch");

   ...
}
```

- Aussitôt l'exception lancée, le flux de contrôle sort de la méthode en cours d'exécution.
- Alors quand MalAuVentreException est lancée, nous sortons de la méthode mangePizza() pour aller là où cette méthode a été appelée.

Encore des exceptions

- Supposons que le fragment de code suivant ait appelé la méthode mangePizza() en premier lieu.

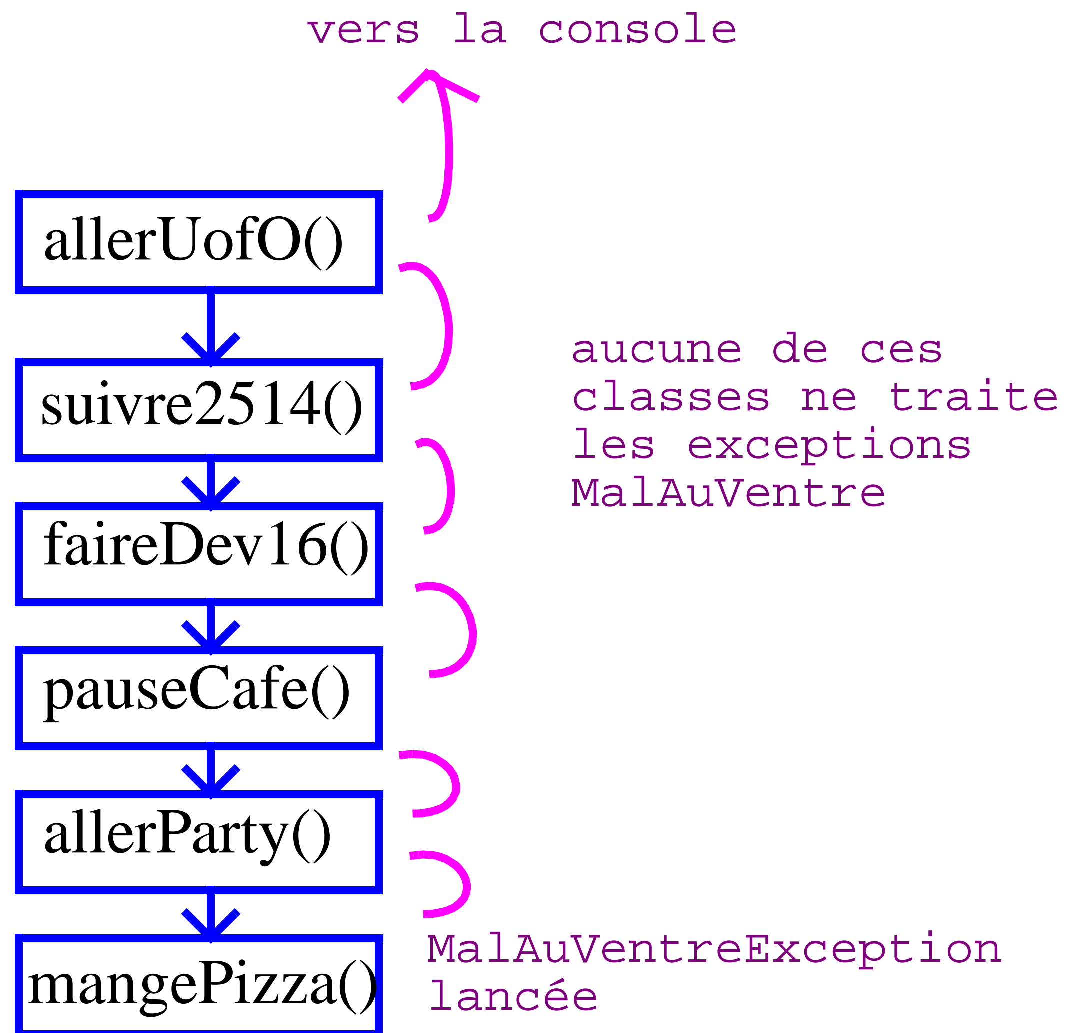
```
private void simuleRencontre()
{
   ...
   try
      {
         unStupideAE.mangePizza();
      }
   catch(MalAuVentreException e)
      {
         System.out.println(“quelqu’un a mal au ventre”);
      }
   ...
}
```

Toujours des exceptions

- Nous retournerons à `unStupideAE.mangePizza();` parce que, souvenez-vous, `mangePizza()` lança l'exception.
- Le bloc `try` et le bloc `catch` indiquent que nous sommes à l'écoute des exceptions qui sont spécifiées dans le paramètre de `catch`.
- Parce que `catch` est à l'écoute de `MalAuVentreException`, le contrôle ira au bloc `catch`, et `System.out.println` sera alors exécuté.
- Notez que le bloc `catch` peut contenir n'importe quoi, pas seulement un `System.out.println`. Vous pouvez gérer les erreurs détectées comme bon vous semble, et vous pouvez même les relancer.
- Notez aussi que si vous lancez une exception dans votre méthode, vous devez ajouter une clause `throws` à la suite du nom de votre méthode.
- Pourquoi utiliser les exceptions? Vous pouvez déléguer vers le haut la responsabilité de traiter les erreurs, c'est-à-dire que le code qui a appelé la méthode en cours aura à gérer le problème.

Toujours des exceptions

- Si vous ne traitez pas une exception (avec catch), elle sera propagée vers le haut le long de la chaîne d'appels de méthodes jusqu'à ce que l'utilisateur l'observe.



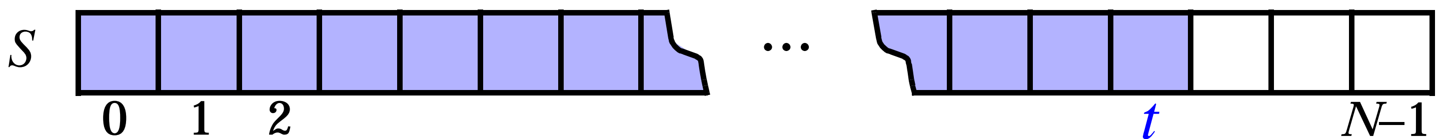
Exceptions finales

- Ainsi, nous savons comment lancer et traiter des exceptions. Mais que sont-elles exactement en Java? Des classes!
- Observez MalAuVentreException.

```
public class MalAuVentreException extends
   RuntimeException {
   public MalAuVentreException(String err)
      {
         super(err);
      }
}
```

Pile à base de vecteur

- Créez une pile en utilisant un vecteur et en spécifiant une taille maximale N, par ex. $N = 1\ 024$.
- La pile est composée d'un vecteur de N éléments S et d'une variable entière t, l'index de l'élément au-dessus de la pile S.



- Les indices acceptables pour ce vecteur commencent à 0, alors nous initialisons t à -1.
- Pseudo-code

Algorithm size():
  return t +1

Algorithm isEmpty():
  return ($t < 0$)

Algorithm top():
  if isEmpty() **then**
    throw a StackEmptyException
  return $S[t]$
...

Pile à base de vecteur (suite)

- Pseudo-Code (suite)

Algorithm push(o):
  if size() = N **then**
    throw a StackFullException
  $t \leftarrow t + 1$
  $S[t] \leftarrow o$

Algorithm pop():
  if isEmpty() **then**
    throw a StackEmptyException
  $e \leftarrow S[t]$
  $S[t] \leftarrow$ **null**
  $t \leftarrow t\text{-}1$
  return e

- Chacune des méthodes ci-haut a un temps d'exécution constant ($O(1)$)
- La réalisation avec vecteur est simple et efficace.
- Il y a une limite supérieure, N, pour la taille de la pile. Une valeur arbitraire N pourrait être trop petite pour une application, ou gaspiller de la mémoire.

Pile à base de vecteur: Une réalisation en Java

```
public class ArrayStack implements Stack {
  // Implementation of the Stack interface
  // using an array.

  public static final int CAPACITY = 1000; // default
                       // capacity of the stack
  private int capacity; // maximum capacity of the
                       // stack.
  private Object S[ ]; // S holds the elements of
                      // the stack
  private int top = -1; // the top element of the
                       // stack.

  public ArrayStack( ) { // Initialize the stack
      this(CAPACITY);// with default capacity
  }

  public ArrayStack(int cap) { // Initialize the
                  // stack with given capacity
      capacity = cap;
      S = new Object[capacity];
  }
```

Pile à base de vecteur — Réalisation en Java (suite)

```
public int size( ) { //Return the current stack
                    // size
    return (top + 1);
}
public boolean isEmpty( ) {  // Return true iff
                             // the stack is empty
    return (top < 0);
}


public void push(Object obj) { // Push a new
                             // object on the stack
    if (size() == capacity) {
      throw new StackFullException("Stack overflow.");
    }
    S[++top] = obj;
}


public Object top( ) // Return the top stack
                    // element
    throws StackEmptyException {
    if (isEmpty( )) {
      throw new StackEmptyException("Stack is
        empty.");
      }
    return S[top];
}
```

Pile à base de vecteur — Réalisation en Java (suite)

```
public Object pop() // Pop off the stack element
    throws StackEmptyException {
  Object elem;
  if (isEmpty( )) {
    throw new StackEmptyException(“Stack is Empty.”);
  elem = S[top];
  S[top--] = null; // Dereference S[top] and
                   // decrement top
  return elem;
  }
}
```

Pile extensible à base de vecteur

- Au lieu d'abandonner avec StackFullException, nous pouvons remplacer le vecteur S par un plus grand vecteur et continuer à traiter les opérations *push*.

Algorithm push(o):
  if size() = N **then**
    $A \leftarrow$ *new array of length f(N)*
  for $i \leftarrow 0$ **to** $N - 1$
    $A[i] \leftarrow S[i]$
  $S \leftarrow A$
  $t \leftarrow t + 1$
  $S[t] \leftarrow o$

- ***De quelle taille devrait être le nouveau vecteur?***
  - ***stratégie ajustée*** (additionner c): $f(N) = N + c$
  - ***stratégie de croissance*** (doubler): $f(N) = 2N$

- Afin de comparer ces deux stratégies, nous utiliserons le modèle de coût suivant:

opération *push* régulière: ajouter un élément	1
opération *push* spéciale: créer un vecteur de taille $f(N)$, copier N éléments, et ajouter un élément	$f(N) + N + 1$

Stratégie ajustée (c=4)

- Débuter avec un vecteur de taille 0
- Le coût d'une opération *push* spéciale est $2N + 5$

push	phase	n	N	coût
1	1	0	0	5
2	1	1	4	1
3	1	2	4	1
4	1	3	4	1
5	2	4	4	13
6	2	5	8	1
7	2	6	8	1
8	2	7	8	1
9	3	8	8	21
10	3	9	12	1
11	3	10	12	1
12	3	11	12	1
13	4	12	12	29

Performance de la stratégie ajustée

- Nous considérons k phases, où $k = n/c$
- Chaque phase correspond à une nouvelle taille de vecteur
- Le coût d'une phase i est de $2ci$
- le coût total de n opérations *push* est le coût total de k phases, avec $k = n/c$:

  $2c\ (1 + 2 + 3 + ... + k)$,

  qui est $O(k^2)$ et $\mathbf{O(n^2)}$.

Stratégie de croissance

- Débuter avec un vecteur de taille 0, ensuite 1, 2, 4, ...
- Le coût d'un *push* spécial est de $3N + 1$, où $N > 0$

push	phase	n	N	coût
1	0	0	0	2
2	1	1	1	4
3	2	2	2	7
4	2	3	4	1
5	3	4	4	13
6	3	5	8	1
7	3	6	8	1
8	3	7	8	1
9	4	8	8	25
10	4	9	16	1
11	4	10	16	1
12	4	11	16	1
...	...	...	...	...
16	4	15	16	1
17	5	16	16	49

Performance de la stratégie de croissance

- Nous considérons k phases, où $k = \log n$
- Chaque phase correspond à une nouvelle taille de vecteur
- Le coût d'une phase i est de 2^{i+1}
- le coût total de n opérations *push* est le coût total de k phases, avec $k = \log n$

  $2 + 4 + 8 + ... + 2^{\log n + 1} =$

  $2n + n + n/2 + n/4 + ... + 8 + 4 + 2 = 4n - 1$

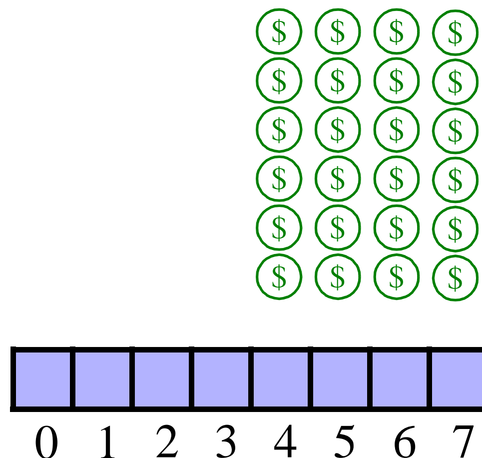
- La stratégie de croissance gagne!

Analyse amortie

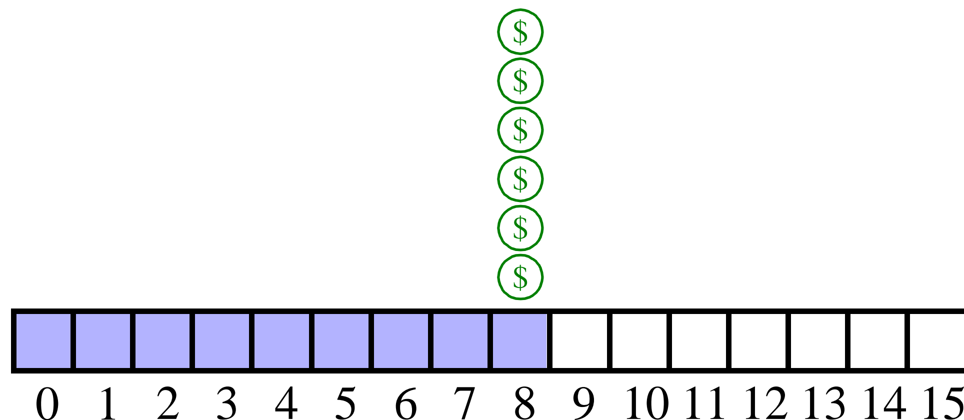
- Le temps d'exécution amorti d'une opération parmi une série d'opérations est le temps d'exécution du pire des cas de la série d'opérations toute entière divisé par le nombre d'opérations.
- La méthode de comptabilité détermine le temps d'exécution amorti à l'aide d'un système de crédits et de débits.
- Nous considérons l'ordinateur comme un appareil à sous qui exige un cyber-dollar pour une quantité constante de temps de calcul.
- Nous fixons un procédé pour facturer les opérations. Il s'agit là d'un ***procédé d'amortissement***.
- Nous pouvons surfacturer certaines opérations et en sousfacturer d'autres. Par exemple, nous pouvons ***facturer un même montant pour chaque opération***.
- Le procédé doit toujours nous procurer suffisament d'argent pour payer le coût réel de l'opération.
- Le coût total de la série d'opérations n'est pas plus élevé que le montant total facturé.
- (temps amorti) ≤ (total $ facturé) / (# opérations)

Procédé d'amortissement pour la stratégie de croissance

- À la fin d'une phase, nous devons avoir assez économisé pour payer le *push* spécial de la phase suivante.
- À la fin de la phase 3, il faut avoir économisé $24.



- Les économies payent pour la croissance du vecteur.



- Nous facturons **$7** pour un *push*. Les $6 économisés par *push* régulier sont "conservés" dans la seconde moitié du vecteur.

Analyse d'amortissement pour la stratégie de croissance

- Nous facturons **$5** (offre spéciale de lancement) pour le premier *push* et **$7** pour les suivants.

push	*n*	*N*	***solde***	***facture***	coût
1	0	0	$0	$5	$2
2	1	1	$3	$7	$4
3	2	2	$6	$7	$7
4	3	4	$6	$7	$1
5	4	4	$12	$7	$13
6	5	8	$6	$7	$1
7	6	8	$12	$7	$1
8	7	8	$18	$7	$1
9	8	8	$24	$7	$25
10	9	16	$6	$7	$1
11	10	16	$12	$7	$1
12	11	16	$18	$7	$1
...	...	...	...	...	...
16	15	16	$42	$7	$1
17	16	16	$48	$7	$49

“Casting” avec une pile générique

- Avoir un ArrayStack qui peut contenir seulement des objets Entier ou des objets Étudiant.
- Afin de réaliser ceci à l’aide d’une pile générique, les objets retournés doivent être “moulés” (*cast*) dans le bon type de donnée.
- Un exemple en Java:

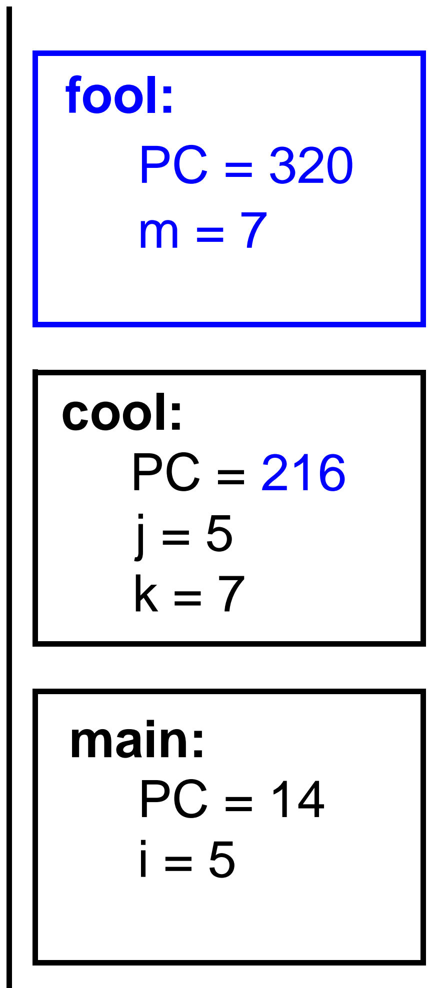
```
public static Integer[] reverse(Integer[] a) {
   ArrayStack S = new ArrayStack(a.length);
   Integer[] b = new Integer[a.length];
   for (int i = 0; i < a.length; i++)
      S.push(a[i]);
   for (int i = 0; i < a.length; i++)
      b[i] = (Integer)(S.pop()); // the popping
        // operation gave us an Object, and we
        // casted it to an Integer before
        // assigning it to b[i].
   return b;
}
```

Piles dans la Machine Virtuelle Java (JVM)

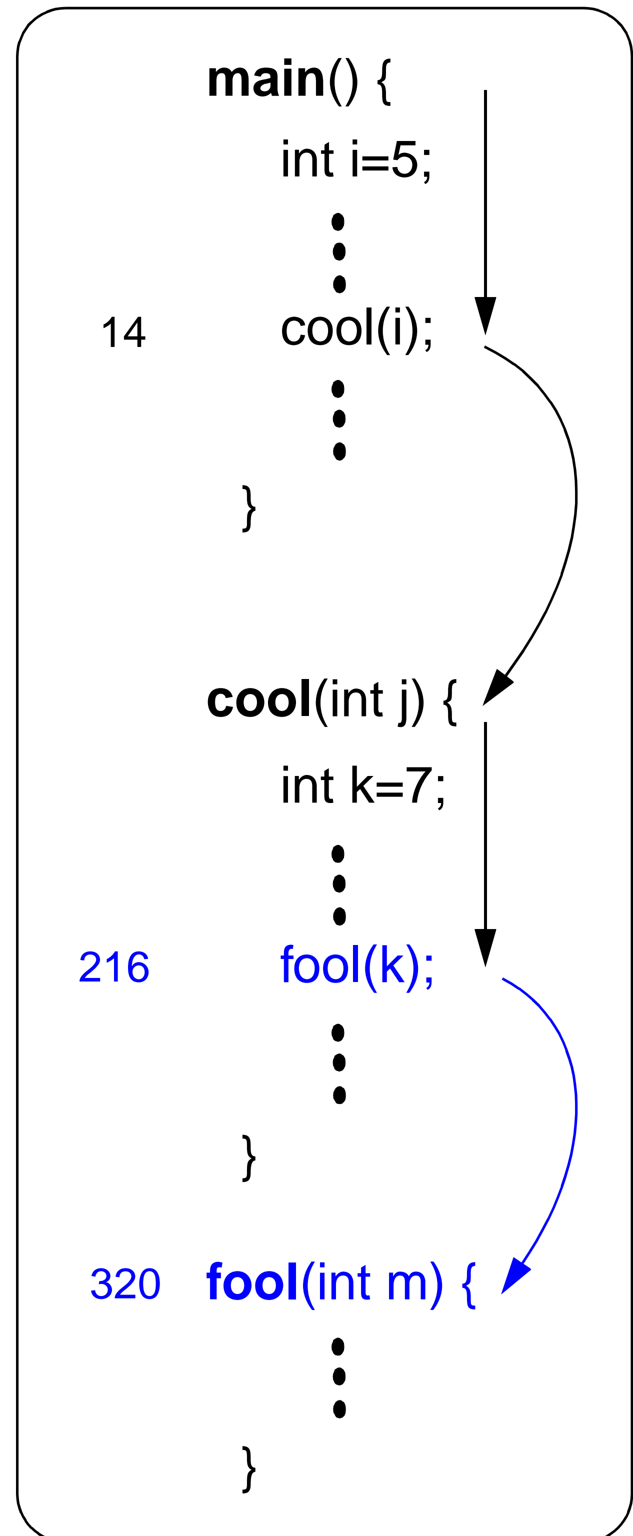
- Chaque processus en exécution dans un programme Java a sa propre pile de méthodes (Method Stack).
- Chaque fois qu'une méthode est appelée, elle est empilée sur une telle pile.
- L'utilisation d'une pile pour cette opération permet à Java de faire plusieurs choses utiles:
  - Exécuter des appels récursifs de méthode
  - Afficher la trace d'une pile pour localiser une erreur.
- Java inclut aussi une pile d'opérandes qui est utilisée pour évaluer les instructions arithmétiques:

```
Integer add(a, b):
  OperandStack Op
  Op.push(a)
  Op.push(b)
  temp1 ← Op.pop()
  temp2 ← Op.pop()
  Op.push(temp1 + temp2)
  return Op.pop()
```

Pile de méthodes Java



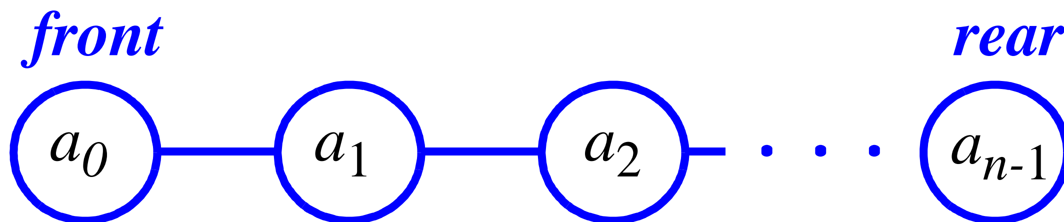
Pile Java



Programme Java

Files (*Queues*)

- Une file se distingue d'une pile par ses routines d'insertion et de retrait qui suivent le principe premier entré, premier sorti (*first-in-first-out*, ou *FIFO*).
- Des éléments peuvent être insérés à tout moment, mais seulement l'élément qui a été le plus longtemps dans la file peut être retiré.
- Les éléments sont enfilés (*enqueued*) par l'arrière (*rear*) et défilé (*dequeued*) par l'avant (*front*)

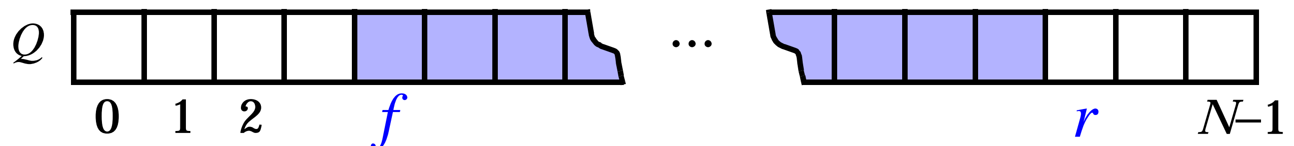


Le TAD File (ou Queue)

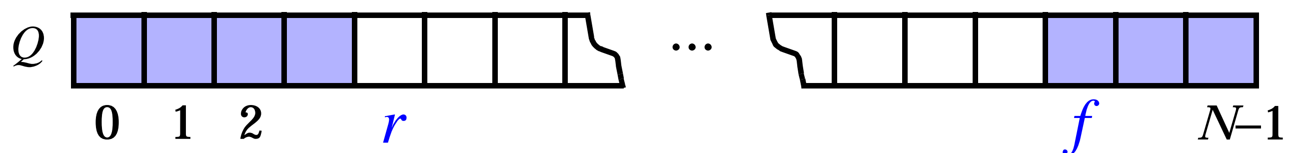
- La file supporte deux méthodes fondamentales:
  - enqueue(*o*): Insère l'objet *o* à l'arrière de la file
  - dequeue(): Retire l'objet du devant de la file et retourne-le; une erreur survient lorsque la file est vide
- Les méthodes secondaires suivantes devraient aussi être définies:
  - size(): Retourne le nombre d'objets dans la file
  - isEmpty(): Retourne un booléen indiquant si la pile est vide
  - front(): Retourne, sans le retirer, l'objet au devant de la file; si la pile est vide, alors une erreur survient

File à base de vecteur

- Créez une file en utilisant un vecteur circulaire.
- Spécifiez une taille maximale N, par ex. $N = 1\ 000$.
- La file est composée d'un vecteur de N éléments Q et de deux variables entières:
  - f, l'index de l'élément du devant
  - r, l'index de l'élément suivant celui de l'arrière
- Configuration "normale"



- Configuration circulaire ("*wrapped around*")



- Que veut dire $f=r$?

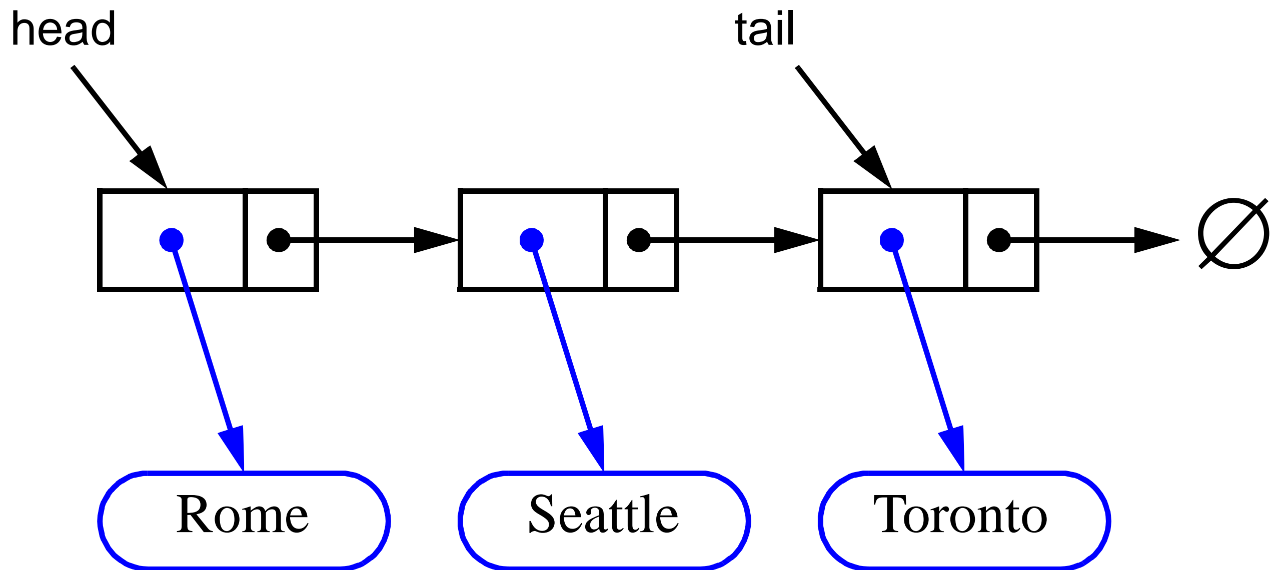
File à base de vecteur (suite)

- Pseudo-code

```
Algorithm size():
  return (N - f + r) mod N

Algorithm isEmpty():
  return (f = r)

Algorithm front():
  if isEmpty() then
    throw a QueueEmptyException
  return Q[f]

Algorithm dequeue():
  if isEmpty() then
    throw a QueueEmptyException
  temp ← Q[f]
  Q[f] ← null
  f ← (f + 1) mod N
  return temp

Algorithm enqueue(o):
  if size = N - 1 then
    throw a QueueFullException
  Q[r] ← o
  r ← (r +1) mod N
```

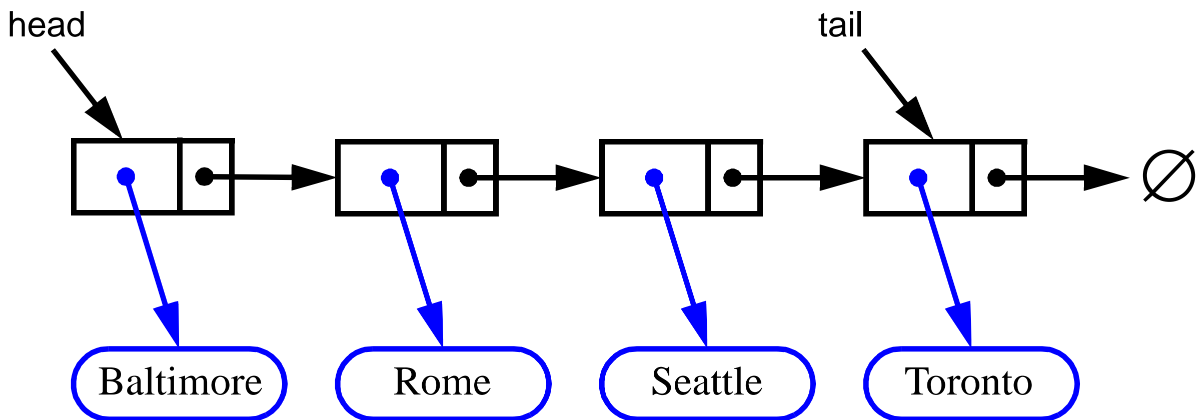
Réalisation d'une file à l'aide d'une liste simplement chaînée

- nœuds connectés en chaîne par des liens (*links*)

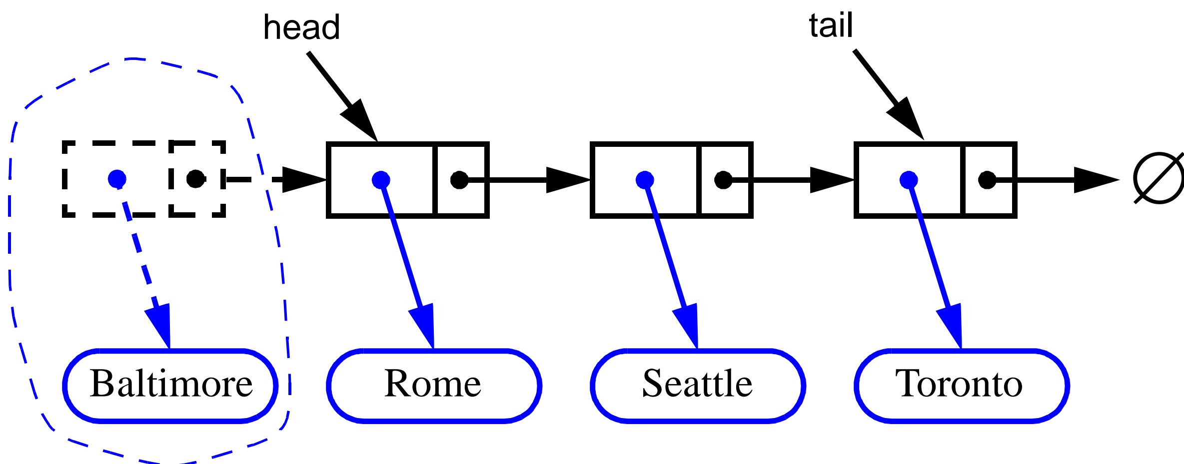


- la tête (*head*) de la liste est le devant de la file, la queue de la liste (*tail*) est le derrière de la file.
- pourquoi pas le contraire?

Retirer l'élément de tête



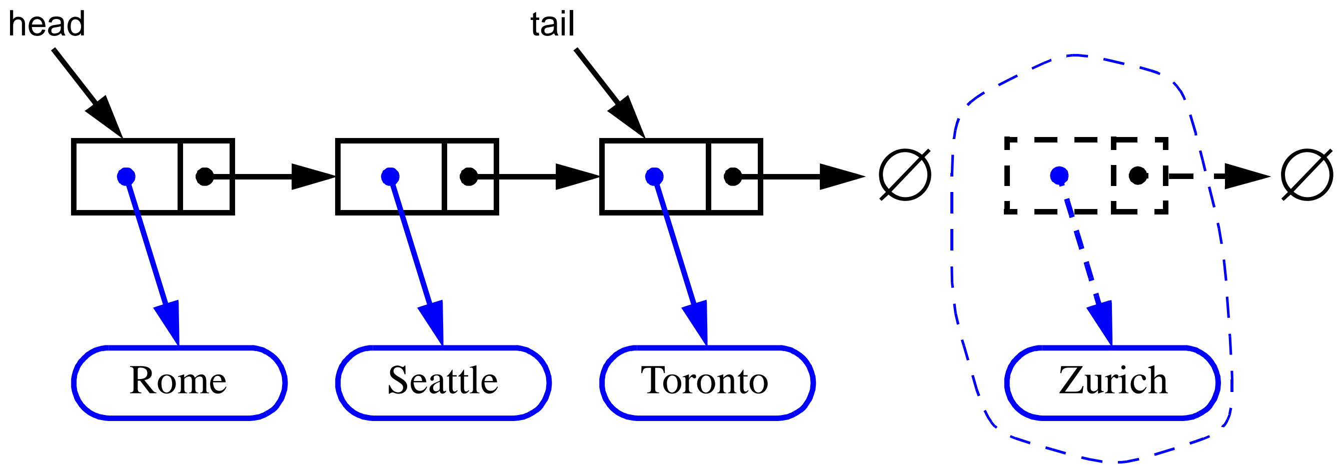
- avancez la référence de la tête



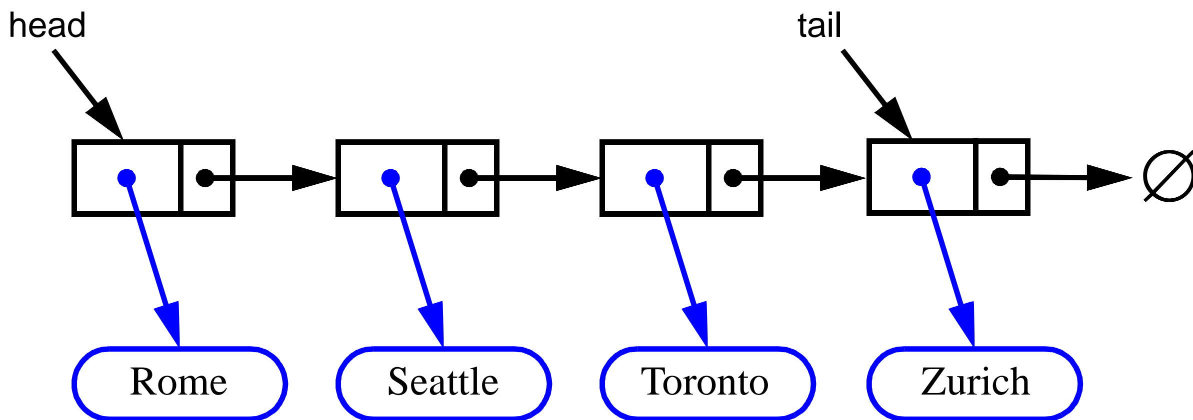
- insérer un élément à la tête est tout aussi facile.

Insérer un élément à la queue

- créez un nouveau nœud



- enchaînez-le et déplacez la référence à la queue



- comment retirer l'élément de queue?

Files à deux bouts (Double-Ended Queues)

- une file à deux bouts, ou deque, supporte l'insertion et le retrait à l'avant comme à l'arrière.
- Le TAD Deque:
  - insertFirst(*e*): Insère *e* au début de la deque
  - insertLast(*e*): Insère *e* à la fin de la deque
  - removeFirst(): retire et retourne le premier élément
  - removeLast(): retire et retourne le dernier élément
- Les méthodes secondaires incluent:
  - first()
  - last()
  - size()
  - isEmpty()

Réalisations de piles et de files à l'aide de Deques

- Piles avec Deques:

Méthode de Pile	Réalisation avec Deque
size()	size()
isEmpty()	isEmpty()
top()	last()
push(e)	insertLast(e)
pop()	removeLast()

- Files avec Deques:

Méthode de File	Réalisation avec Deque
size()	size()
isEmpty()	isEmpty()
front()	first()
enqueue()	insertLast(e)
dequeue()	removeFirst()

Le patron de conception Adaptateur (*Adaptor Pattern*)

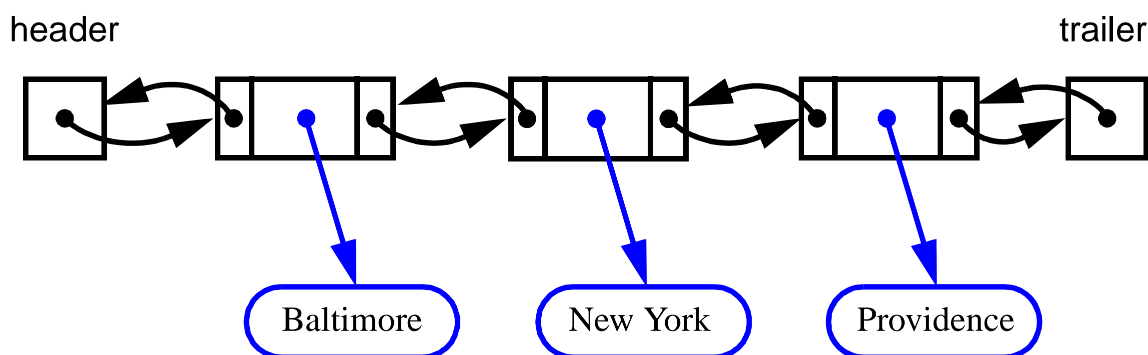
- L'utilisation d'une deque pour réaliser une pile ou une file est un exemple du patron de conception adaptateur (*adoptor pattern*). Ce patron réalise une classe en utilisant des méthodes d'une autre classe.

- Souvent, les classes *adaptateur* spécialisent des classes générales.

- Voici deux applications:
  - Spécialisation d'une classe générale en changeant quelques méthodes:
    Ex: réalisation d'une pile avec une deque.
  - Spécialisation de types d'objets utilisés par une classe générale:
    Ex: définir une classe IntegerArrayStack qui adapte ArrayStack pour ne contenir que des entiers.

Réalisation de deques à l'aide de listes doublement chaînées

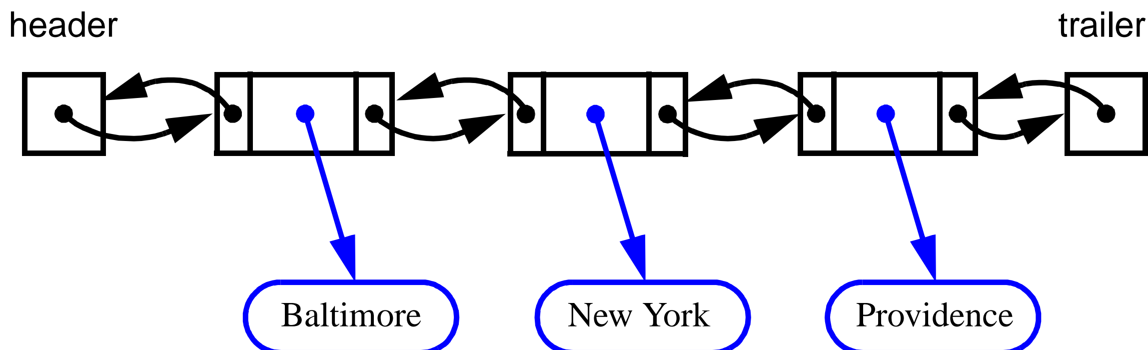
- Effacer l'élément de queue d'une liste simplement chaînée ne peut pas être fait en un temps constant.
- Pour réaliser une deque, nous utilisons une liste doublement chaînée avec des nœuds spéciaux pour l'avant (*header*) et l'arrière (*trailer*).



- Un nœud de liste doublement chaînée a un lien suivant (*next*) et un lien précédent (*prev*). Ce nœud supporte les méthodes suivantes:
  - setElement(Object e)
  - setNext(Object newNext),
  - setPrev(Object newPrev)
  - getElement(), getNext(), getPrev()
- En utilisant une liste doublement chaînée, toutes les méthodes de deque ont un temps d'exécution constant (c'est-à-dire,O(1))!

Réalisation de deques à l'aide de listes doublement chaînées (suite)

- En réalisant une liste doublement chaînée, nous ajoutons deux nœuds spéciaux aux extrémités: les nœuds *header* et *trailer.*
  - Le nœud *header* est placé avant le premier élément de la liste. Il a un prochain lien valide, mais un lien précédent vide.
  - Le nœud *trailer* est placé après le dernier élément de la liste. Il a un lien précédent valide, mais un prochain lien vide.
- les nœuds *header* et *trailer* sont des sentinelles ou nœuds "bidon" parce qu'ils ne contiennent pas d'éléments.
- Diagramme de notre liste doublement chaînée:



Réalisation de deques à l’aide de listes doublement chaînées (suite)

- Visualisons le code de removeLast().