ITI 1521. Introduction à l'informatique II*

Marcel Turcotte
École d'ingénierie et de technologie de l'information

Version du 2 mars 2011

Résumé

– Files
  – LinkedQueue

*. Ces notes de cours ont été conçues afin d'être visualiser sur un écran d'ordinateur.

Définitions

Une **file** (*queue*) est un **type abstrait de données** linéaire tel que l'ajout de données se fait à une extrémité, l'**arrière** (*rear*) de la file, et le retrait à l'autre, l'**avant** (*front*).

Ces structures de données sont dites FIFO : *first-in first-out*.

`enqueue()` ⇒ Queue ⇒ `dequeue()`

Les deux opérations de base sont :

enqueue : l'ajout d'un élément à l'arrière de la file,

dequeue : le retrait d'un élément à l'avant de la file.

⇒ Les files sont donc des structures de données semblables aux files d'attente au supermarché, à la banque, au cinéma, etc.

Applications des files

– Gestion de ressources partagées :
  – Accès au processeur ;
  – Accès à un disque ou autres périphériques, ex. imprimante ;
– Algorithmes à base de files :
  – Simulations ;
  – Génération de séquences de longueur croissantes sur un alphabet de taille fini ;
  – Trouver la sortie dans un labyrinthe.

Exemple simple d'utilisation

```
public class Test {
    public static void main( String[] args ) {

        Queue<Integer> q = new QueueImplementation<Integer>();

        for ( int i=0; i<10; i++ )
            q.enqueue( new Integer( i ) );

        while ( ! q.isEmpty() )
            System.out.println( q.dequeue() );

    }
}
```

⇒ Imprime ? 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

```
q = new Q();
q.enqueue( a );
q.enqueue( b );
q.enqueue( c );
q.dequeue( );
-> a
q.dequeue( );
-> b
q.enqueue( d );
q.dequeue( );
-> c
q.dequeue( );
-> d
```

⇒ Les éléments sont traités dans le même ordre qu'ils ont été insérés dans la file, ici l'élément **a** est le premier à joindre la file et c'est aussi le premier à quitter la file (*first-come first-serve*).

Implémentations

Tout comme les piles, il y a deux familles d'implémentations :
– Listes chaînées ;
– À l'aide d'un tableau.

TAD

```
public interface Queue {
    public abstract boolean isEmpty();
    public abstract void enqueue( Object o );
    public abstract Object dequeue();
}
```

TAD

```
public interface Queue<E> {
    public abstract boolean isEmpty();
    public abstract void enqueue( E o );
    public abstract E dequeue();
}
```

Implémentation à l'aide d'éléments chaînés

```
public class LinkedQueue<E> implements Queue<E> {

    public boolean isEmpty() { ... }
    public void enqueue( E o ) { ... }
    public E dequeue() { ... }

}
```

Implémentation à l'aide d'éléments chaînés

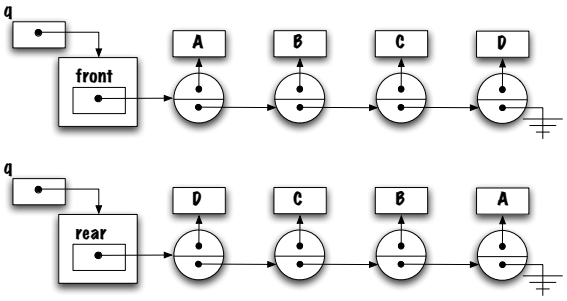
```
public class LinkedQueue<T> implements Queue<T> {

    private static class Elem<E> {
        private E value;
        private Elem<E> next;
        private Elem( E value, Elem<E> next ) {
            this.value = value;
            this.next = next;
        }
    }

    public boolean isEmpty() { ... }
    public void enqueue( T o ) { ... }
    public T dequeue() { ... }
}
```

Implémentation à l'aide d'éléments chaînés

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

    private static class Elem<E> {
        private E value;
        private Elem<E> next;
        private Elem( E value, Elem<E> next ) {
            this.value = value;
            this.next = next;
        }
    }

    private Elem<T> front; // or rear?

    public boolean isEmpty() { ... }
    public void enqueue( T o ) { ... }
    public T dequeue() { ... }
}
```

Implémentation à l'aide d'éléments chaînés

Quelle représentation vous semble préférable et pourquoi ?



Implémentation à l'aide d'éléments chaînés

Si nous choisissons la première implémentation, alors le retrait d'un élément sera facile (et rapide) mais l'ajout à l'arrière de la file sera difficile (et lent).

L'autre implémentation ne fait qu'inverser les situations, le retrait devient coûteux alors que l'ajout est rapide.

Est-ce l'impasse ?

Que faut-il pour faciliter le retrait ? Une référence **front**.

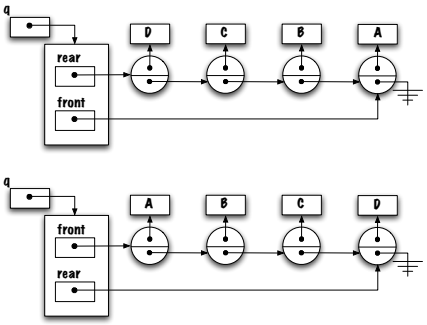
Que faut-il pour faciliter l'ajout ? Une référence **rear**.

Implémentation à l'aide d'éléments chaînés

Solution : la classe **LinkedQueue** aura une référence vers l'élément avant de la file et une référence vers l'élément arrière. Ainsi, les deux opérations, ajout et retrait, seront rapides.

Implémentation à l'aide d'éléments chaînés

Est-ce ces deux implémentations seront aussi efficaces l'une que l'autre ?



Discussion

Quels seront les impacts de cette modification ?

La quantité de mémoire supplémentaire est négligeable.

L'implémentation des méthodes sera plus complexe.

Ajout d'un élément

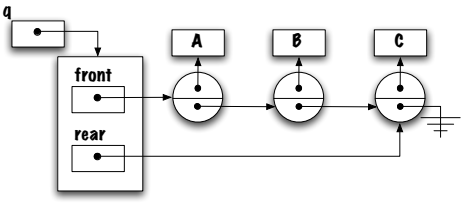
Identifiez le cas général ainsi que le ou les cas spéciaux.

Cas général, considérez un nombre suffisant d'éléments afin qu'il représente la majorité des cas.

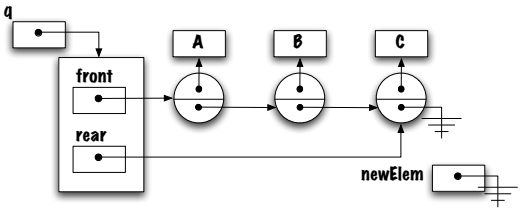
Les cas spéciaux sont les cas ou la stratégie employée pour le cas général ne s'appliquerait pas.

Files/Piles/Listes vides ou ayant un élément sont souvent les cas spéciaux.

Ajout d'un élément (cas général)

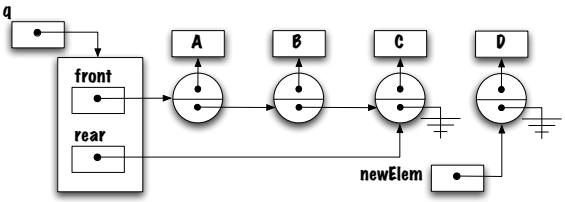


Ajout d'un élément (cas général)

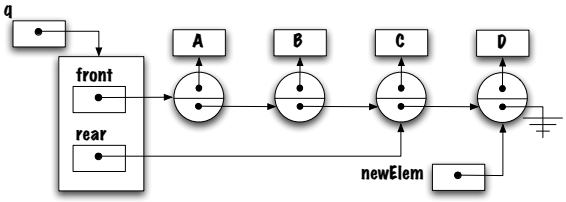


L'utilisation d'une variable locale facilitera le travail.

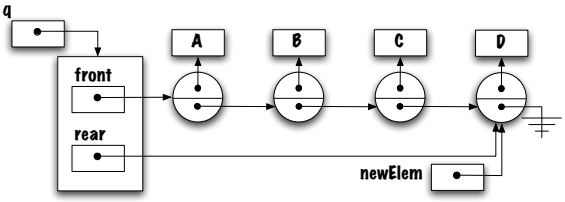
Ajout d'un élément (cas général)



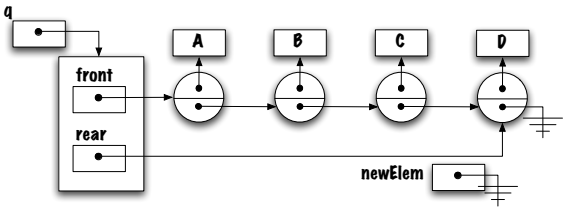
Ajout d'un élément (cas général)



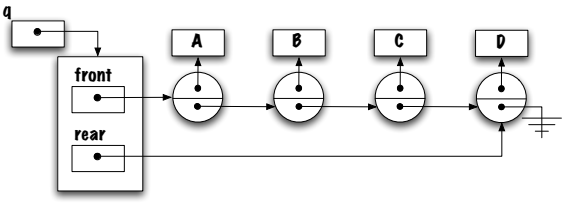
Ajout d'un élément (cas général)



Ajout d'un élément (cas général)

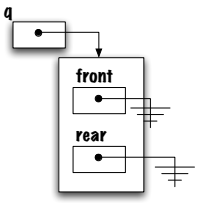


Ajout d'un élément (cas général)



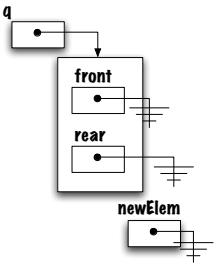
Ajout d'un élément (cas spécial)

Dessinez le diagramme de mémoire représentant la file vide.

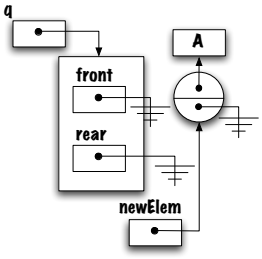


Quelle expression permet de reconnaitre (identifier) la file vide ?

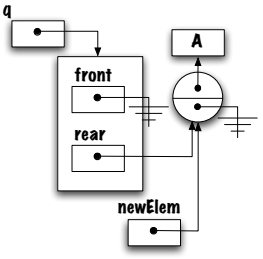
Ajout d'un élément (cas spécial)



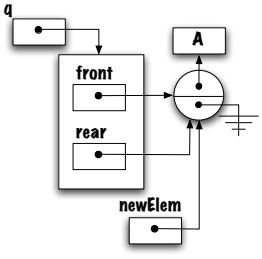
Ajout d'un élément (cas spécial)



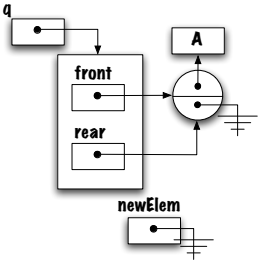
Ajout d'un élément (cas spécial)



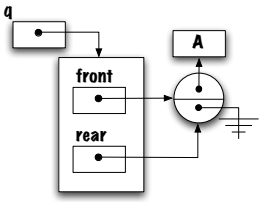
Ajout d'un élément (cas spécial)



Ajout d'un élément (cas spécial)



Ajout d'un élément (cas spécial)



Retrait d'un élément

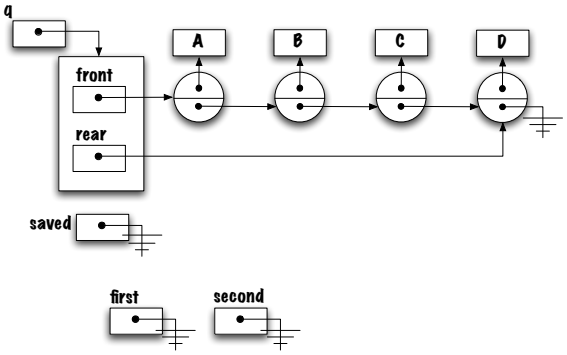
Identifiez le cas général ainsi que le ou les cas spéciaux.

Est-ce que la file vide est un cas spécial ?

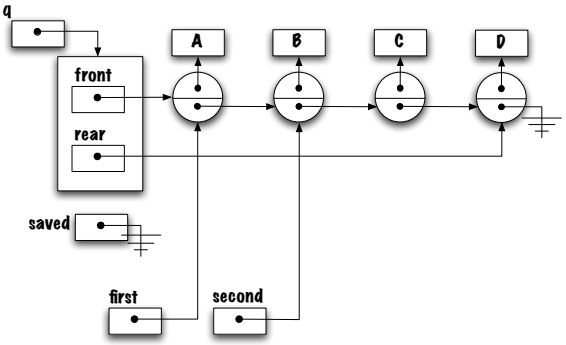
Non, c'est un cas illégal, pour lequel il faudra lancer une exception.

La file contenant un seul élément est le cas spécial.

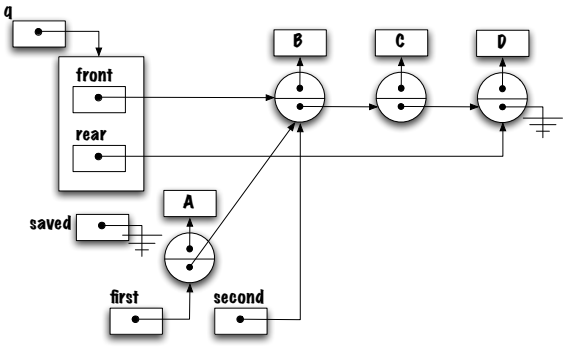
Retrait d'un élément (cas général)



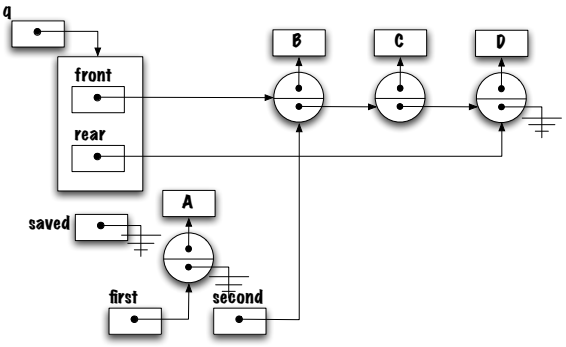
Retrait d'un élément (cas général)



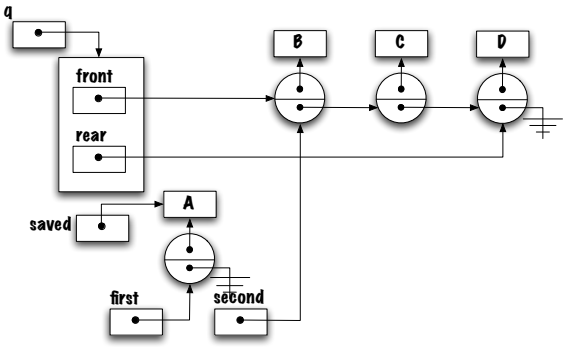
Retrait d'un élément (cas général)



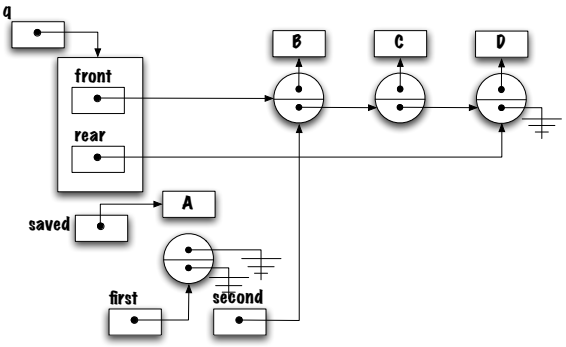
Retrait d'un élément (cas général)



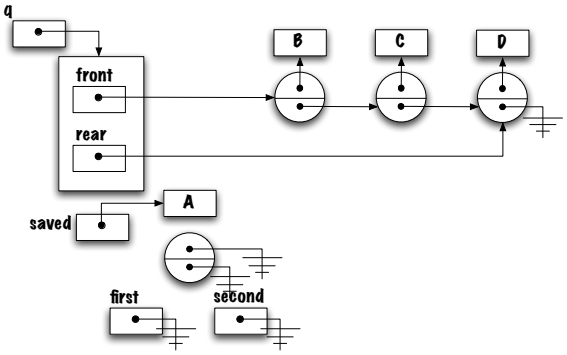
Retrait d'un élément (cas général)



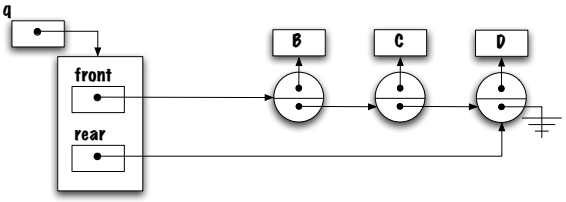
Retrait d'un élément (cas général)



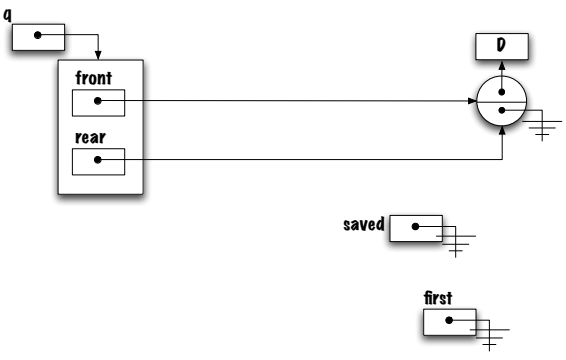
Retrait d'un élément (cas général)



Retrait d'un élément (cas général)

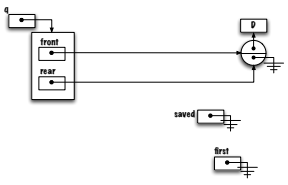


Retrait d'un élément (cas spécial)



Quelle expression permet de reconnaître une file contenant un seul élément ?

Retrait d'un élément (cas spécial)



Quelle expression permet de reconnaître une file contenant un seul élément ?

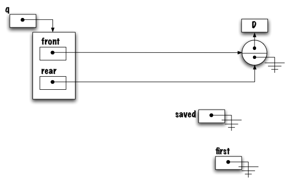
```
font != null && front.next == null
```

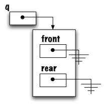
Que pensez-vous de ce qui suit ?

```
front == rear
```

Retrait d'un élément (cas spécial)

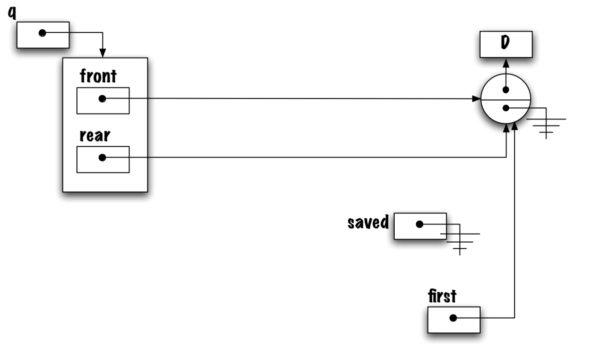
`front == rear` est vrai pour la file contenant un élément mais aussi la file vide ! !



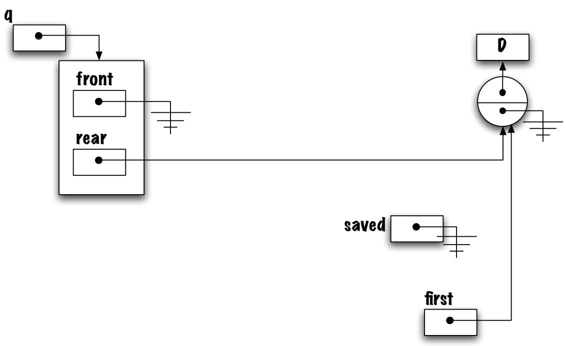


Une solution : `front != null && front == rear`

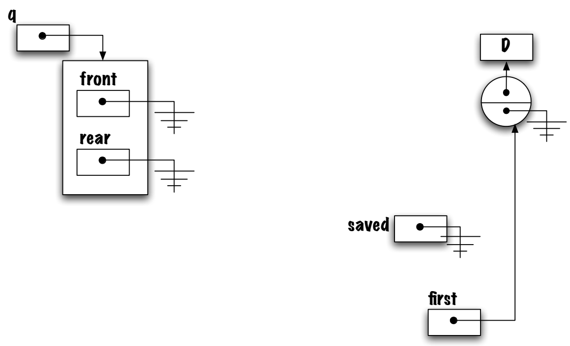
Retrait d'un élément (cas spécial)



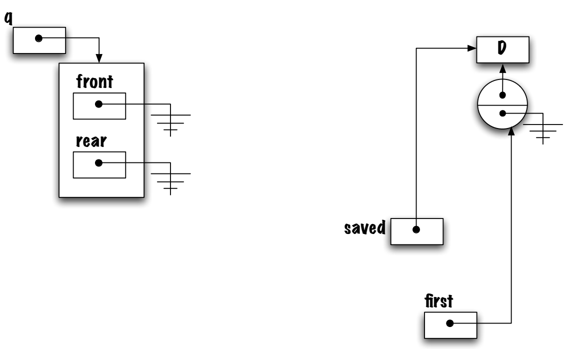
Retrait d'un élément (cas spécial)



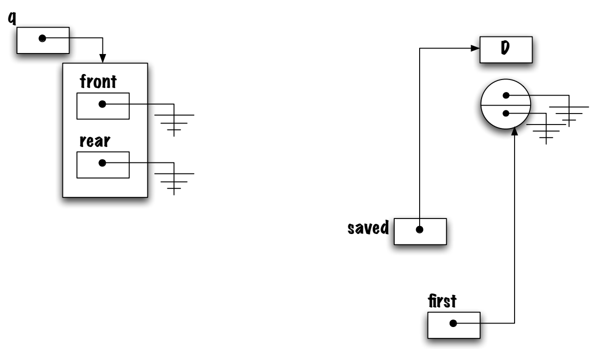
Retrait d'un élément (cas spécial)



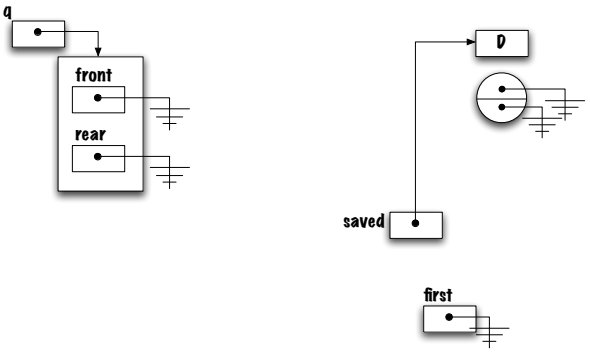
Retrait d'un élément (cas spécial)



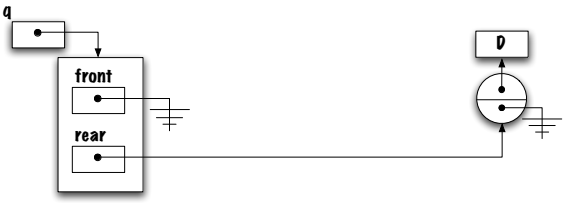
Retrait d'un élément (cas spécial)



Retrait d'un élément (cas spécial)



Piège !



Voici un problème commun, le lien vers l'élément arrière n'a pas été brisé.

Quels types de problèmes pourraient survenir ?

Qu'arrivera-t-il si l'on utilise l'expression suivante afin de détecter la file vide ? `front == null && rear == null`.