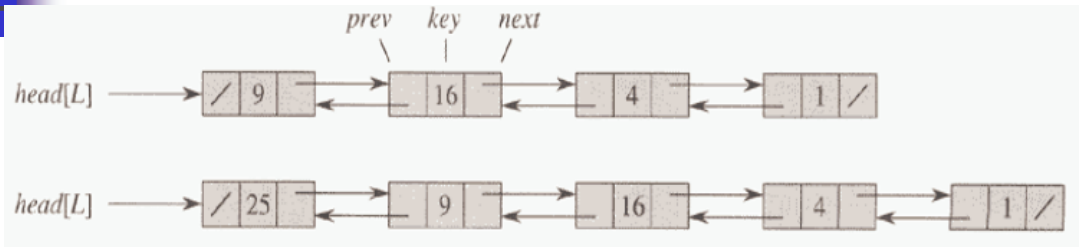
ADTs

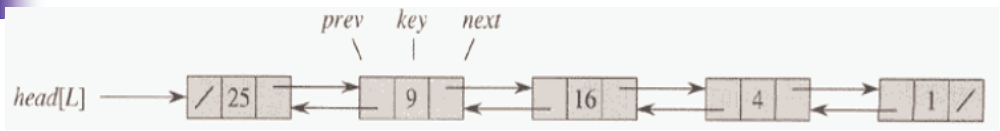
- ADTs permitem programação modular:
  - separam o programa em módulos mais pequenos;
  - clientes diferentes podem partilhar a mesma ADT.
- ADTs permitem o encapsulamento:
  - mantêm-se os módulos independentes;
  - podem-se substituir as várias classes que implementam a mesma interface, sem alterar o cliente.
- Aspectos do desenho de uma ADT:
  - especificação formal do problema;
  - a implementação tem tendência a tornar-se obsoleta.

Lista duplamente ligada Inserção



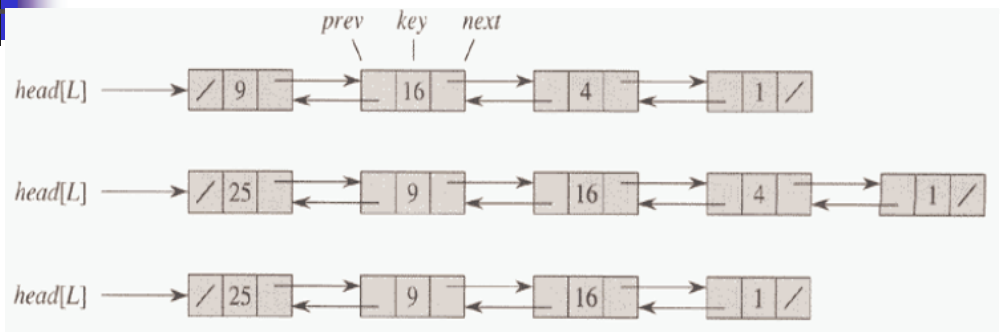

```
LIST-INSERT(L, x)
1  next[x] ← head[L]
2  if head[L] ≠ NIL
3     then prev[head[L]] ← x
4  head[L] ← x
5  prev[x] ← NIL
```

Lista duplamente ligada
Pesquisa




```
LIST-SEARCH(L, k)
1  x ← head[L]
2  while x ≠ NIL and key[x] ≠ k
3        do x ← next[x]
4  return x
```


Lista duplamente ligada
Remoção



Lista Duplamente Ligada

```
public interface DList<E> {

  public int size();
  public boolean isEmpty();

  public void add(E e);
  public boolean remove(E e);

}
```


Lista Duplamente Ligada

```
public class DLinkedList1<E> implements DList<E> {

private static class DNode<T> {
  public DNode<T> next, prev;
    public T key;
    public DNode()   { }
    public DNode(T e ) { key=e; }
}

protected int size;
protected  DNode<E> head;
public int size()    { return size;}
public boolean isEmpty(){ return head == null;}

public void add(E e) { // Adiciona o elemento no inicio da lista ligada
   DNode<E> n=new DNode<E>(e);
   if(head!=null){ n.next=head;  head.prev=n;}
   head=n; }
//(...)
}
```

Lista Duplamente Ligada

```
public class DLinkedList1<E> implements DList<E> {
(...)
// Remove o primeiro elemento da lista igual a e
  public boolean remove (E e){
    DNode aux=search(e);
    if(aux!=null){
      if(aux.prev != null) {aux.prev.next = aux.next;}
      else{head = aux.next;}
      if(aux.next != null){ aux.next.prev = aux.prev;}
      return true;
    }
    else return false;
}
  protected DNode<E> search( E e ){
    DNode<E> aux = head;
    while( aux != null && !aux.key.equals(e))
      aux=aux.next;
      return aux;
    }
//(...)
}
```


Lista Duplamente Ligada

```
public class DLinkedList1<E> implements DList<E> {
(...)
  public static <E> void show(DNode<E> l){
    System.out.print("< ");
    while(l != null){ System.out.print(l.key+" "); l = l.next;}
    System.out.println(">");
}
  public void reverse() {
    DNode<E> iter = head, aux;
    while( iter != null ) {
      head = iter;
      aux = iter.next;
      iter.next = iter.prev;
      iter.prev = aux;
      iter = aux;
   }
}
(...)
}
```

Iteradores

```
public interface Iterable<E> {

 public Iterator<E> iterator();

}
```

```
public interface Iterator<E> {

 public boolean hasNext();

public E next();

 public void remove();

}
```


Lista Duplamente Ligada- Iteradores

```
public class DLinkedList1<E> implements DList<E>,Iterable<E> {
  private class LinkedIterator implements Iterator<E> {
     // Nó dos dados retornados pelo next() seguinte.
  protected DNode<E> node = head;
        /* Nó retornado pelo next() anterior. Como inicialmente não existiu next()
   referecia null. Depois de um remove() permite assinalar que não existe next()
   anterior.*/
  protected DNode<E> prev = null;
     // Retorna true se existem mais elementos para iterar.
  public boolean hasNext() { return node != null; }
     // Retorna o elemento da posição corrente e avança.
  public E next() {
   if ( node == null )throw new NoSuchElementException();
   prev = node; node = node.next; // Avança
   return prev.key;
}
(...)
```

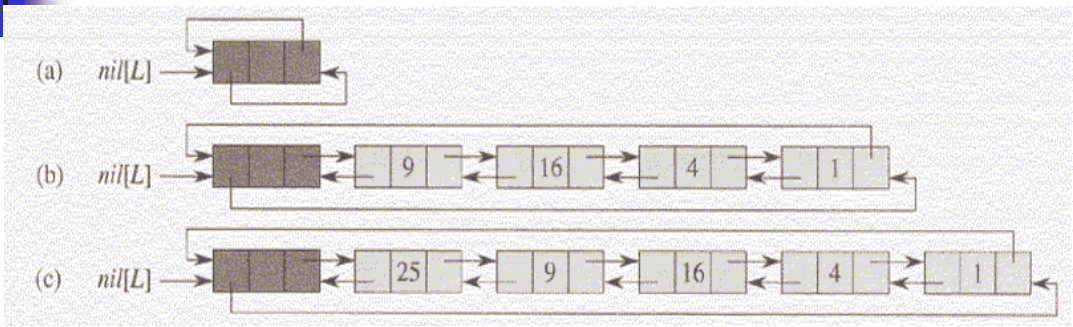
Lista Duplamente Ligada- Iteradores

```
(...)
 public void remove() {
  // Se a última operação não foi next() lança excepção
  if ( prev == null ) throw new IllegalStateException();
  // Remove o nó do elemnto retornado pelo último next
  DLinkedList1.this.remove(prev);
  prev = null; // Assinala que a operação anterior não foi um next()
 }
}//termina a classe LinkedIterator

public Iterator<E> iterator() {return new LinkedIterator(); }

protected void remove( DNode<E> rem ){
  if ( rem.next != null) rem.next.prev=rem.prev;
  if ( rem.prev != null )
      rem.prev.next=rem.next;
  else
    head = rem.next;
  --size;
}
(...)}
```


Lista com sentinela Inserção



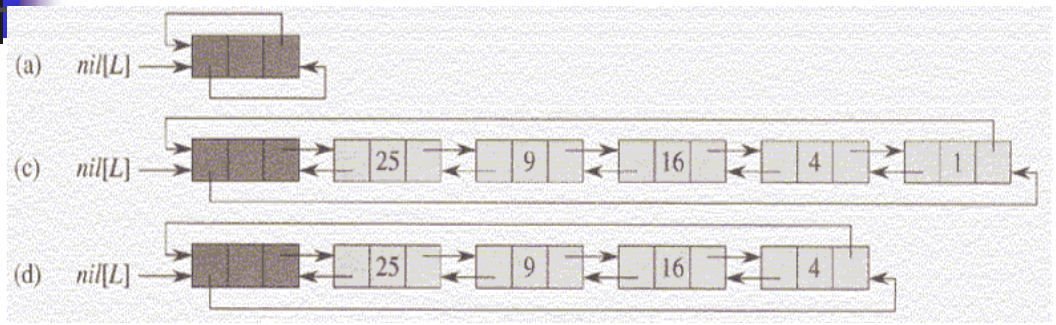
LIST-INSERT′(L, x)

1 $next[x] \leftarrow next[nil[L]]$
2 $prev[next[nil[L]]] \leftarrow x$
3 $next[nil[L]] \leftarrow x$
4 $prev[x] \leftarrow nil[L]$

LIST-INSERT(L, x)

1 $next[x] \leftarrow head[L]$
2 **if** $head[L] \neq$ NIL
3 **then** $prev[head[L]] \leftarrow x$
4 $head[L] \leftarrow x$
5 $prev[x] \leftarrow$ NIL

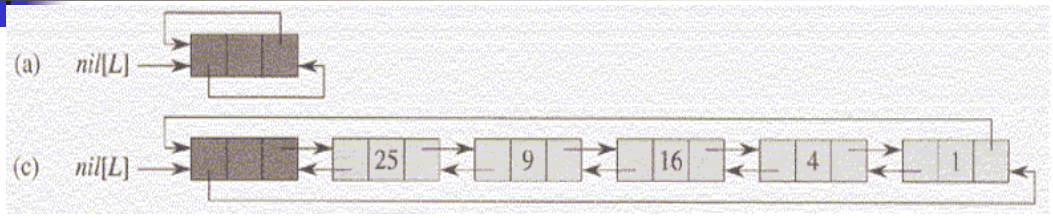
Lista com sentinela
Remoção




```
LIST-DELETE′(L, x)
1  next[prev[x]] ← next[x]
2  prev[next[x]] ← prev[x]
```

```
LIST-DELETE(L, x)
1  if prev[x] ≠ NIL
2     then next[prev[x]] ← next[x]
3     else head[L] ← next[x]
4  if next[x] ≠ NIL
5     then prev[next[x]] ← prev[x]
```


Lista com sentinela
Pesquisa




```
LIST-SEARCH′(L, k)
1  x ← next[nil[L]]
2  while x ≠ nil[L] and key[x] ≠ k
3        do x ← next[x]
4  return x
```

```
LIST-SEARCH(L, k)
1  x ← head[L]
2  while x ≠ NIL and key[x] ≠ k
3        do x ← next[x]
4  return x
```

Lista Duplamente Ligada com Sentinela e Circular

```
public class DLinkedList3<E> implements DList<E>{
  private static class DNode<E> {
    public DNode<E> next, prev;
    public E key;
    public DNode()   { }
    public DNode(E e ) { key=e; }
    }
}
    private int size;
    protected DNode<E> dummy;

    public DLinkedList3() {
       dummy=new DNode<E>(); dummy.next= dummy.prev = dummy;}

public int size()    { return size;}
public boolean isEmpty() { return dummy.next == dummy;}
//(...)
}
```


Lista Duplamente Ligada com Sentinela e Circular

```
public class DLinkedList3<E> implements DList<E>{
  //(...)
  public E getFirst() { return dummy.next.key; }
  public E getLast()  { return dummy.prev.key; }

// Adiciona o elemento no fim da lista ligada
  public void add(E e) {
    DNode<E> n=new DNode<E>(e); ++size;
    n.prev=dummy.prev;
    n.next=dummy;
    dummy.prev.next=n;
    dummy.prev=n;
}

public boolean remove(E e){
   DNode<E> aux=search(e);
   if(aux!=null){
       aux.next.prev=aux.prev; aux.prev.next=aux.next; return true;}
   return false;
   }
//(...)
}
```

Lista Duplamente Ligada com Sentinela e Circular

```
public class DLinkedList3<E> implements DList<E>{
  //(...)
  public DNode<E> search(E e){
    DNode<E> aux=dummy.next;
    while(aux!=dummy){
      if(aux.key.equals(e)){ return aux;}
      aux=aux.next;
    }
    return null;
  }

  public boolean remove(E e){
    DNode<E> aux=search(e);
    if(aux!=null){
      aux.next.prev=aux.prev;
      aux.prev.next=aux.next;
      return true;
    }
    return false;
}
//(...)
}
```


Lista Duplamente Ligada com Sentinela e Circular

```
public class DLinkedList3<E> implements DList<E>{
  //(...)/*Move os nós da lista 2 para a lista 1. Assume que as duas listas estão ordenadas
  *pela ordem dada pelo comparador. A lista 1 permanece ordenada e a lista 2 fica vazia.*/
 public static < E >  DNode<E> merge( DNode<E> dummy1, DNode<E> dummy2,
   Comparator<E> cmp) {
     DNode<E> head1 = dummy1.next; DNode<E> head2 = dummy2.next;
      while (head1 != dummy1 && head2 != dummy2 ) {
         if ( cmp.compare( head1.key, head2.key) <= 0 ) head1= head1.next;
         else {
                  head2.prev = head1.prev;
                  head1.prev.next = head2;
                  head1.prev=head2;
                  head2 = head2.next;
                  head1.prev.next = head1;  }
      }
        if ( head2 != dummy2 ) {
                 head1.prev.next = head2;
                 head2.prev = head1.prev;
                 head1.prev = dummy2.prev;
                 dummy2.prev.next = head1;
        }
        dummy2.next = dummy2.prev = dummy2; return dummy1;
   }}
```

Lista Duplamente Ligada com Sentinela e Circular

```
public class DLinkedList3<E> implements DList<E>{
  //(...)/*Move os nós da lista 2 para a lista 1. Assume que as duas listas estão ordenadas
  *pela ordem dada pelo comparador. A lista 1 permanece ordenada e a lista 2 fica vazia.*/
 public static < E >  DNode<E> merge( DNode<E> dummy1, DNode<E> dummy2,
   Comparator<E> cmp) {
     DNode<E> head1 = dummy1.next; DNode<E> head2 = dummy2.next;
      while (head1 != dummy1 && head2 != dummy2 ) {
          if ( cmp.compare( head1.key, head2.key) <= 0 ) head1= head1.next;
          else {
                   head2.prev = head1.prev;
                   head1.prev.next = head2;
                   head1.prev=head2;
                   head2 = head2.next;
                   head1.prev.next = head1;  }
      }
        if ( head2 != dummy2 ) {
                 head1.prev.next = head2;
                 head2.prev = head1.prev;
                 head1.prev = dummy2.prev;
                 dummy2.prev.next = head1;
        }
        dummy2.next = dummy2.prev = dummy2; return dummy1;
   }}
```


Exercício

- Implementar o algoritmo *merge sort* para duas listas duplamente ligadas.