

Estructuras de Datos

Estructuras lineales:

Lista, Lista Ordenada y Cola de Prioridad

Contenidos

- El TAD Lista.
- Iteradores.
- El TAD Lista ordeanda.
- El TAD cola de prioridad.

EEDD - GRADO EN ING. INFORMÁTICA UCO

Listas

- Recuerda: la lista “simple”.

List[T]

Makers:

- **make():List[T]** //makes a empty list.
 - **post-c:** isEmpty() is True.

Observers:

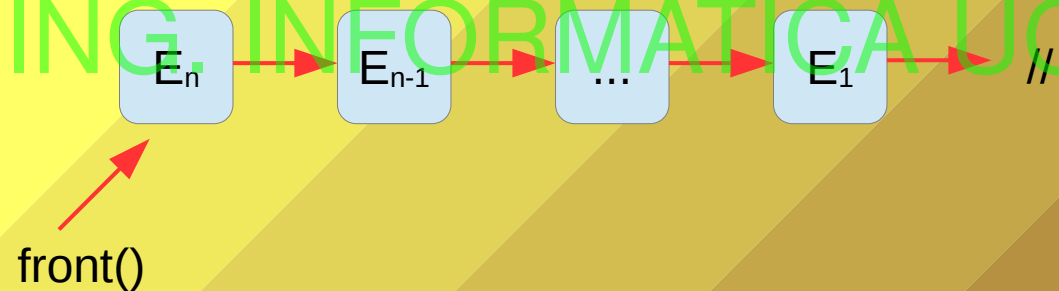
- **isEmpty():Boolean** //is the list empty?
- **size():Integer** //Number of items in the list.
- **front():T** //return the first Item of the list.
 - **pre-c:** not isEmpty()

Modifiers:

- **pushFront(item:T)** //insert item before the head.
 - **post-c:** front() == item
 - **post-c:** size()==old.size()+1
- **popFront()** //delete the first item of the list.
 - **pre-c:** not isEmpty()
 - **post-c:** size()==old.size()-1

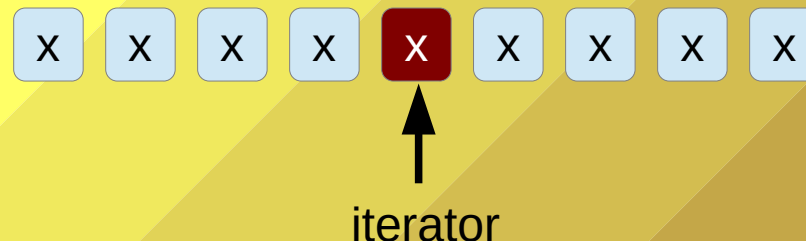
Invariants:

- isEmpty() or size()>0



Listas

- En la práctica, ¿qué otras operaciones sobre una lista necesitamos?
 - Poder acceder a todos los elementos de la lista, no sólo a la cabeza.
 - Poder realizar operaciones de edición (inserción/borrado) en todas las posiciones de la lista.
 - Solución:
 - usar un Iterador (nuevo TAD) para navegar e indicar la posición de la lista sobre la que queremos editar.



Listas

- TAD ListIterator[T]: especificación.

TAD ListIterator[T]:

Types:

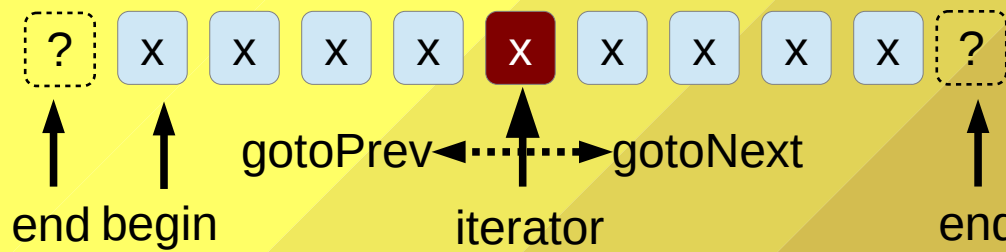
- iterator as ListIterator[T]

Observers:

- isValid():Bool
- operator=(o:iterator):Bool
- distance(o:iterator):Integer
- get():T
 - Pre-c: isValid().
- next(d:Integer):iterator
 - Pos-c: not retv.isValid() or this.distance(retv)=d
- prev(d:Integer):iterator
 - Pos-c: not retv.isValid() or retV.distance(this)=d

Modifiers:

- set(v:T)//Set the item.
 - Pre-c: isValid().
 - Post-c: get()==v
- gotoNext(d:Integer)
 - Pos-c: isValid() or old.this.distance(this)=d
- gotoPrev(d:Integer)
 - Pos-c: not isValid() or this.distance(old.this)=d



No es un iterador válido

Listas

push front, pop front, size, ... SList[T]

- **TAD List[T]: especificación.**

TAD List[T] extend SimpleList[T]

Types:

- iterator as ListIterator[T]

Observers:

- back():T
 - Pre-c: not isEmpty().
- begin():iterator
- end():iterator
- find(v:T, from:iterator):iterator
 - Post-c: not retv.isValid() or v=ret.get()
 - Post-c: retv.isValid() or retv = end()

Modifiers:

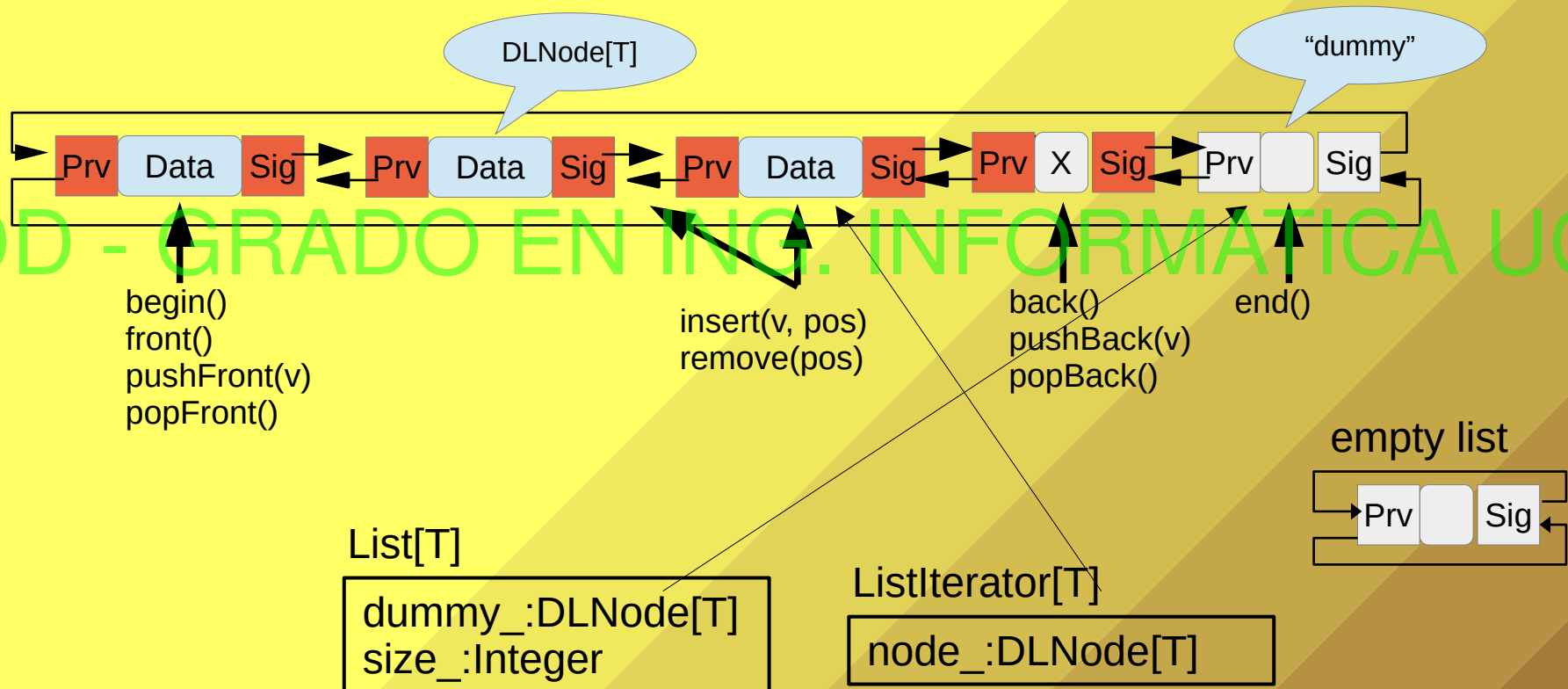
- pushBack(v:T)
 - Pre-c: size()=old.size()+1
 - Post-c: back()=v
- popBack()
 - Post-c: size()=old.size()-1
- insert(v:T, pos:iterator):iterator
 - Post-c: size()=old.size()+1
 - Post-c: retv.item()=v
 - Post-c: retv.next()=pos
- remove(pos:iterator):iterator
 - Pre-c: pos.isValid()
 - Pos-c: size()=old.size()-1
 - Pos-c: old.pos.next()=retv

Invariant:

Not isEmpty() Or begin()==end()

Listas

- TAD List[T]: diseño.



Listas

- TAD DLNode[T]: especificación.
 - Extiende LNode para tener un enlace al nodo previo que posibilita insertar/borrar con $O(1)$ en cualquier posición de la lista.

TAD DLNode[T]: extend LNode[T]

Makers:

- `create(it:T, prev:DLNode, next:DLNode): DLNode[T]`

Observers

- `prev():DLNode[T]` //Gets a link to next node.

Modifiers::

- `setPrev(p:DLNode)` //Sets the link to previous node.
 - **Post-c:** `prev()==p`

DLNode[T]:

`item_:T`
`prev_:DLNode[T]`
`next_:DLNode[T]`

Listas

- TAD ListIterator[T]: diseño.

```
isValid():Bool  
  Return node_ <> Void
```

```
get():T  
  Return node_.get()
```

```
operator=(o:ListIterator[T]):Bool  
  Return node_=o.node_
```

```
distance(o:ListIterator[T]):Int //0(n)  
  dist=0  
  aux:=This  
  While(aux<>o) Do  
    dist++  
    aux.gotoNext(1)  
  End-While  
  Return dist
```

```
next(d:Int):ListIterator[T] //0(1)  
  aux := This  
  aux.gotoNext(d)  
  Return aux
```

ListIterator[T]

node_:DLNode[T]

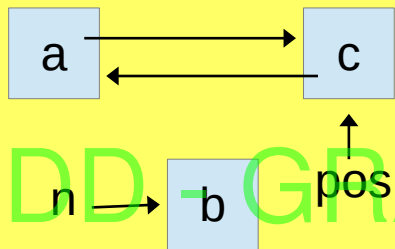
```
prev(d:Int):ListIterator[T] //0( )  
  aux := This  
  aux.gotoPrev(d)  
  Return aux
```

```
set(v:T)  
  node_.set(v)  
  
gotoNext(d:Int) //0( )  
  While d>0 Do  
    node_:=node_.next()  
    d--  
  End-While
```

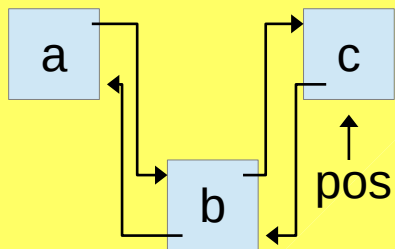
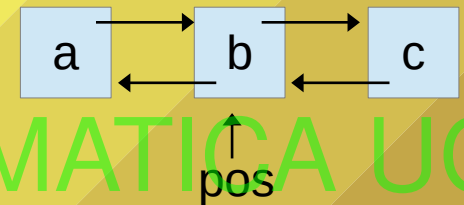
```
gotoPrev(d:Int) //0( )  
  While d>0 Do  
    node_:=node_.prev()  
    d--  
  End-While
```

Listas

- List[T]: diseño algoritmos hook y unhook.



```
Algorithm List[T]::hook(n:DLNode,  
    pos:DLNode[T])  
//O ( )  
Begin  
    n.setPrev(pos.prev());  
    n.setNext(pos);  
    pos.prev().setNext(n);  
    pos.setPrev(n);  
    size_ ← size_+1  
End.
```



```
Algorithm List[T]::unhook(pos:DLNode[T])  
//O ( )  
Begin  
    pos.prev().setNext(pos.next())  
    pos.next().setPrev(pos.prev())  
    size_ ← size_-1  
End.
```

Listas

- List[T]: diseño algoritmos insert/remove.

```
Algorithm List[T]::insert(v:T,  
    pos:iterator):iterator  
    ) //0 ( )  
Var  
    n:DLNode[T]  
Begin  
    n ← DLNode[T]::create(v)  
    hook(n, pos.node())  
    Return iterator(n)  
End.
```

```
Algorithm List[T]::remove(  
    pos:iterator):iterator  
    ) //0 ( )  
Var  
    i:iterator  
Begin  
    i ← pos.next()  
    unhook(pos.node())  
    Return i  
End.
```

Listas

- List[T]: diseño algoritmos push/pop.

```
Algorithm List[T]::pushFront(  
    v:T,  
    ) //0 (1)  
Begin  
    insert(v, begin())  
End.
```

```
Algorithm List[T]::popFront()//0 (1)  
Begin  
    remove(begin())  
End.
```

```
Algorithm List[T]::pushBack(  
    v:T,  
    ) //0 (1)  
Begin  
    insert(v, end())  
End.
```

```
Algorithm List[T]::popBack()//0 (1)  
Begin  
    remove(end().prev())  
End.
```

Listas

- TAD List[T]: diseño algoritmos fold/unfold.

El formato será
'['item1' 'item2' 'item3' ... 'item_n']'
donde item1 is cabeza de la lista.

Algorithm List[T]::fold(out:Stream)

Begin

```
out.write("[")
it ← begin()
While not it=end() Do
  out.write(" ", it.item())
  it.gotoNext()
End-While
out.write(" ]")
```

End.

Se asume la operación T::create(String) para desplegar valores de tipo T desde su representación en formato string.

Algorithm List[T]::create(in:Stream)

Var

tk: String #token

Begin

```
in.read(tk)
If token = "[" Then
  in.read(tk)
  While Not in.eof() And tk <> "]" Do
    pushBack( T::create(tk) )
    in.read(tk)
  End-While
  If tk <> "]" Then
    ERROR
```

Else

ERROR

End-If

End.

Listas

- TAD List[T]: diseño algoritmo find.

```
Algorithm List[T]::find(v:T, pos:iterator):iterator
//O ( )
Var
  i:iterator
Begin
  i ← pos
  While (i!=end() and i.item()!=v) Do
    i.gotoNext()
  Return i
End.
```

Listas

- Ordenar listas.
 - No tenemos acceso aleatorio, no podemos aplicar quicksort.
 - Algoritmo de selección: $O(N^2)$
 - Algoritmo mergeSort: $O(N \log N)$
 - Necesitamos poder fusionar listas ordenadas sin copiar los datos, sólo actualizando enlaces.

Listas

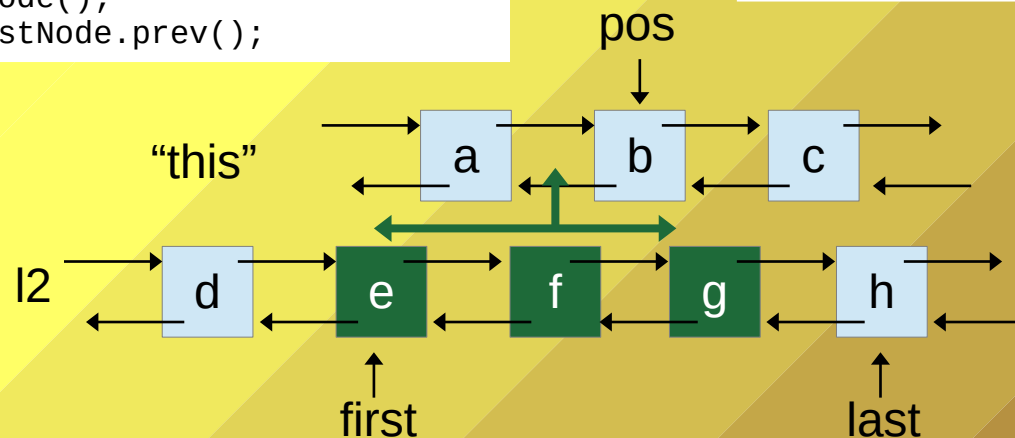
- TAD List[T]: diseño algoritmo splice.

```
Algorithm List[T]::splice(pos:iterator,  
l2:List[T], first:iterator, last:iterator)  
//O ( )  
Var  
    posNode, prevPosNode:DLNode[T]  
    firstNode, prevFirstNode:DLNode[T]  
    lastNode, prevLastNode[T]  
    rangeSize:Integer  
Begin  
    rangeSize ← first.distance(last)  
    posNode ← pos.node();  
    prevPosNode ← posNode.prev();  
    firstNode ← first.node();  
    prevFirstNode ← firstNode.prev();  
    lastNode ← last.node();  
    prevLastNode ← lastNode.prev();
```

```
// Splice l2 range into this.  
prevPosNode.setNext(firstNode)  
firstNode.setPrev(prevPosNode)  
posNode.setPrev(prevLastNode)  
prevLastNode.setNext(posNode)
```

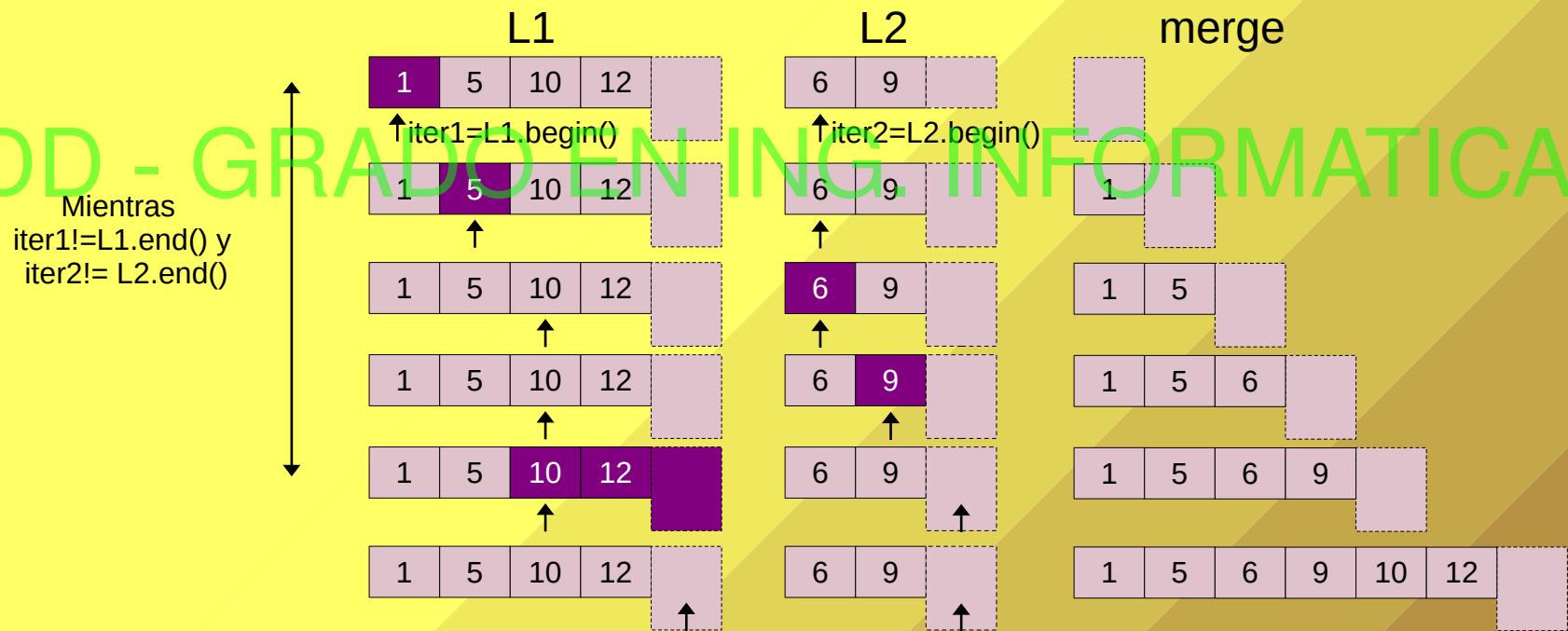
```
// remove range from list2  
prevFirstNode.setNext(lastNode)  
lastNode.setPrev(prevFirstNode)
```

```
// Update lists sizes.  
size_ <- size_ + rangeSize  
l2->size_ <- l2->size_ - rangeSize  
End.
```



Listas

- TAD List[T]: diseño algoritmo merge.



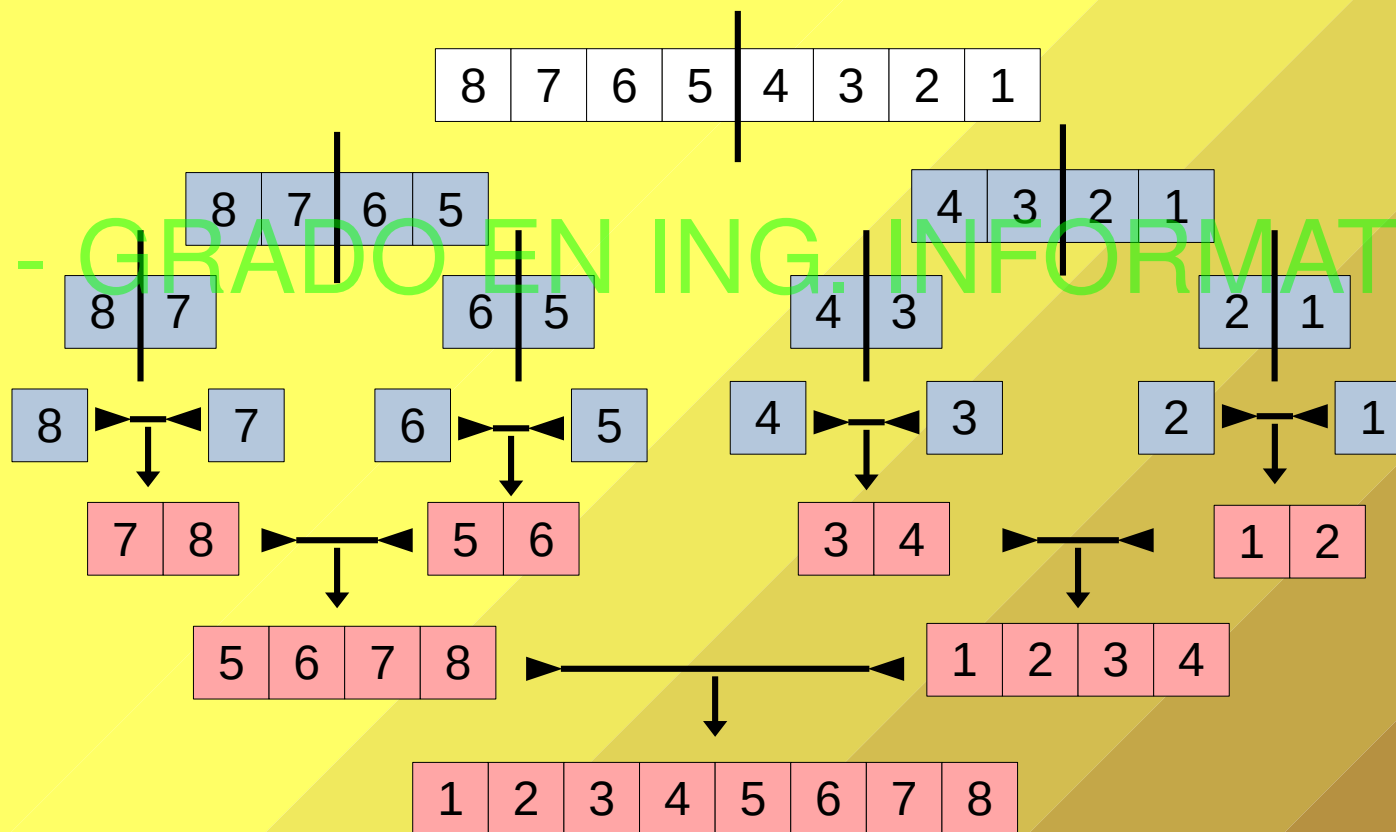
Listas

- TAD List[T]: diseño algoritmo merge.

```
Algorithm List[T]::merge(l2:List[T], cmp:Compare) //0 ( )
Var
  iter1, iter2, tmp: iterator
Begin
  iter1 <- begin()
  iter2 <- l2.begin()
  While iter1 != end() And iter2 != l2.end() Do
    If cmp(iter1.item(), iter2.item()) Then
      iter1.gotoNext()
    Else
      tmp <- iter2
      iter2.gotoNext()
      splice(iter1, l2, tmp, tmp.next())
    End-If
  End-While
  If iter2 != l2.end() Then
    splice(iter1, l2, l2.begin(), l2.end());
  End.
```

Listas

- TAD List[T]: diseño algoritmo mergeSort.



Listas

- TAD List[T]: diseño algoritmo mergeSort.

```
Algorithm List[T]::sort(cmp:Compare)
//0 ( )
Var
  tmp: List[T]
  midPoint: iterator
Begin
  If (size() > 1) Then
    tmp ← List<T>::create()
    midpoint ← begin().gotoNext(size() / 2)
    tmp.splice(tmp.begin(), this, midpoint, end())
    sort(cmp)
    tmp.sort(cmp)
    merge(tmp, cmp)
  End-If
End.
```

Listas

- TAD Queue[T]: Diseño usando una lista.

```
Queue::isEmpty():Boolean //0( )  
    return l_→ isEmpty()
```

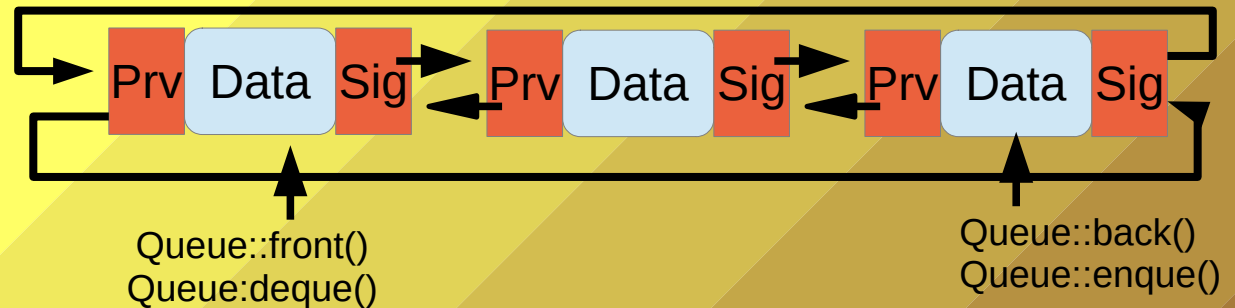
```
Queue::size():Integer //0( )  
    return l_.size()
```

```
Queue::front():T // 0( )  
    return l_.front()
```

```
Queue::back():T // 0( )  
    return l_.back()
```

```
Queue::enqueue(v:T) //0( )  
    l_.pushBack(v)
```

```
Queue::dequeue() //0( )  
    l_.popFront()
```



Lista Ordenada

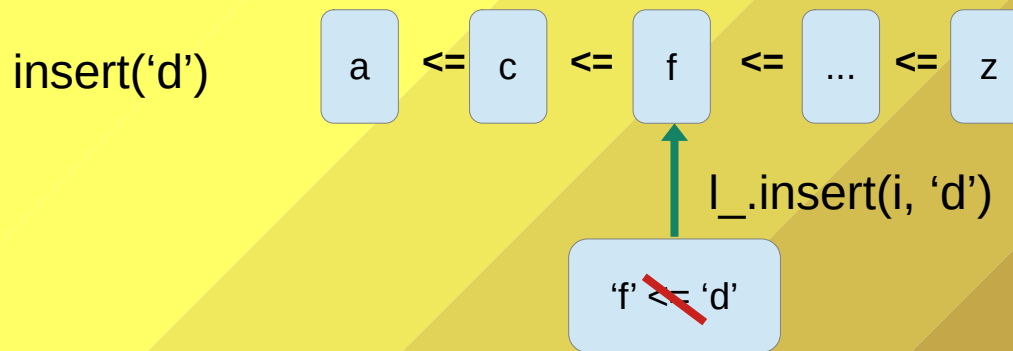
- TAD `OrderedList[T, Comp[T]]`.
 - Representa una lista con el invariante:
 - `isEmpty()` **or** **forall** `it in [begin(), end())`: `it.next()=end()` **or** `Comp(it.item(),it.next().item())`
 - Modificadores:
 - `insert(v:T):iterator` //Insert v in order.
 - Post-c: `size()=old.size()+1`
 - Post-c: `retv.item()=v`
 - Post-c: `not retv.next().isValid()` **or** `Comp(v, retv.next().item())`
 - Post-c: `not retv.prev().isValid()` **or** `Comp(retv.prev().item(), v)`
 - Ventajas:
 - Acceso secuencial en orden de clave con $O(n)$.
 - Recorrido secuencial en orden de clave con $O(n)$.
 - Acceso $O(1)$ al menor/mayor elemento de la secuencia.

`Comp('a','b')=True` si
“a” está delante de “b”
en el orden deseado”

Lista Ordenada

- OrderedList[T]: diseño usando una lista.

```
OrderedList[T, '<=']::insert(v:T):Iterator[T] //0( )
Begin
  i ← l_.begin()
  While (i != l_.end()) And Comp(i.item(), v) Do
    i.gotoNext()
  Return l_.insert(i, v)
End.
```



Cola de Prioridad

- ADT PQueue[T]
 - **Makers:**
 - create():PQueue[T]
 - post-c: isEmpty()
 - **Observers:**
 - isEmpty():Boolean
 - size():Integer
 - front():T
 - pre-c: not isEmpty().
 - post-c: “front is the oldest most prioritized item in the queue”
 - back():T
 - pre-c: not isEmpty().
 - post-c: “back is the newest less prioritized item in the queue”
 - **Modifiers:**
 - enqueue(it:T)
 - post-c: not isEmpty()
 - dequeue()
 - pre-c: not isEmpty()
 - **Invariants:**
 - isEmpty() or size()>0
 - isEmpty() or front()>=back()

PQueue[T]

l_:OrderedList[T, '>=']

```
PQueue::isEmpty():Boolean //0( )
return l_.isEmpty()
```

```
PQueue::size():Integer //0( )
return l_.size()
```

```
PQueue::front():T // 0( )
return l_.front()
```

```
PQueue::back():T // 0( )
return l_.back()
```

```
PQueue::enqueue(v:T) //0( )
l_.insert(v)
```

```
PQueue::dequeue() //0( )
l_.popFront()
```


Resumiendo

- La lista es una estructura de datos indicada cuando el proceso es secuencial y no es necesario realizar accesos aleatorios.
- Usamos iteradores para navegar y editar la lista.
- Utilizamos un diseño de cadena de nodos doblemente enlazados para:
 - Acceso $O(1)$ a la cabeza y a la cola de la lista.
 - Edición de la lista con $O(1)$ en cualquier posición.
 - Ordenación $O(N \log N)$ usando el algoritmo mergeSort.
- La lista ordenada permite acceso secuencial en orden clave con $O(1)$ y recorrido secuencial en orden de clave con $O(N)$.
- Una Cola de prioridad puede ser implementada con una lista ordenada.

Referencias

- Lecturas recomendadas:
 - Caps. 8 y 9 de “Estructuras de Datos”, A. Carmona y otros. U. de Córdoba. 1999.
 - Caps 6 y 7 de “*Data structures and software development in an object oriented domain*”, Tremblay J.P. y Cheston, G.A. Prentice-Hall, 2001.
 - Wikipedia.