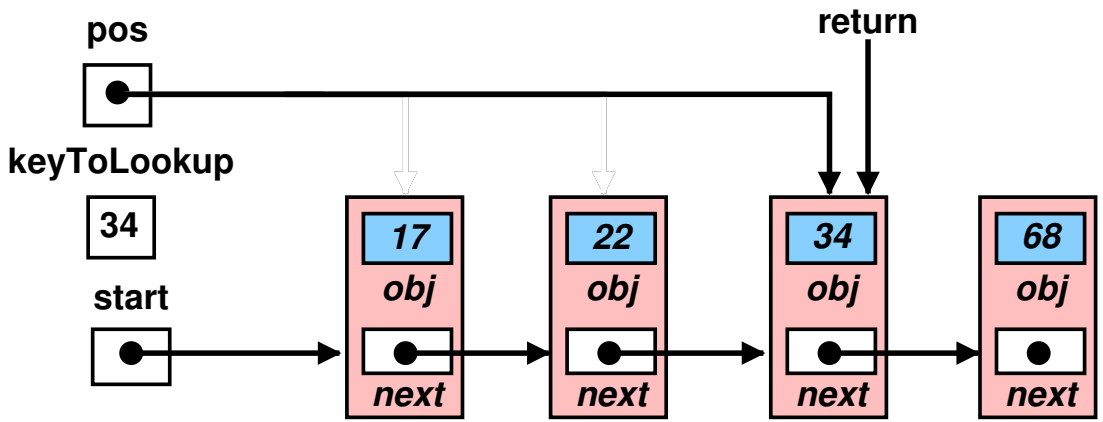
Lists:

- In contrast to Queue and Stack, a **List** allows to insert and delete at arbitrary positions.
- Variant we consider here: '**sorted List**', for simplicity based on Integer elements.
- this requires a sorting key, for example:

```
1   public static int getKey(Object obj) { // sorting key
2       return  (Integer)obj;    // only for Integer objects
3   }
```

```
1  public class SortedList {
2
3    public SortedList ()  { }   Constructor
4
5    private Elem start;   anchor of the list
6
7  methods for list accesses:
8    public static int getKey(Object obj)    { ... }
9    public Elem lookUp (int keyToLookup)    { ... }
10   public void sortIn (Elem newElem)       { ... }
11   public void delete (Elem elemToDelete)  { ... }
12
13   @Override public String toString () {  ...for debugging...
14     String str = "";
15     for (Elem pos = start; pos != null; pos = pos.getNext())
16       str += pos.getValue() + "␣␣";
17     return str;
18   }
19 }
```

```
1    public Elem lookUp (int keyToLookup)  {
2       Elem pos = start;
3       while ( pos != null ){
4          if ( getKey(pos.getObject()) == keyToLookup )
5             return pos;                          // found
6          else
7             pos = pos.getNext();
8       }
9       return null;                    //   not found
10   }
```



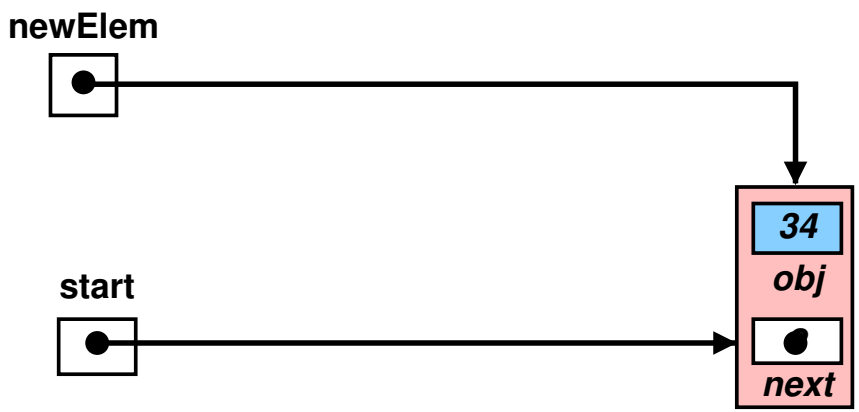
Attention: in this and the following slides only the key of the data object is shown (instead of a reference and the actual object).

Basic structure of `sortIn`:

```
1     public void sortIn (Object newObj) {
2       int key = getKey(newObj);
3       Elem newElem = new Elem(newObj);
4  // insert into empty list
5       if (start == null ) {
6         ...
7       }
8  // insert before first element:
9       if (getKey(start.getObject()) > key ) {
10       ...
11      }
12 // insert between two elements:
13      Elem pos = start;
14      while ( pos.getNext() != null)
15         if (getKey(pos.getNext().getObject()) > key)
16         ...  else ...
17 // insert at end of list:
18      pos.setNext(newElem);
19      newElem.setNext(null);
20    }
```

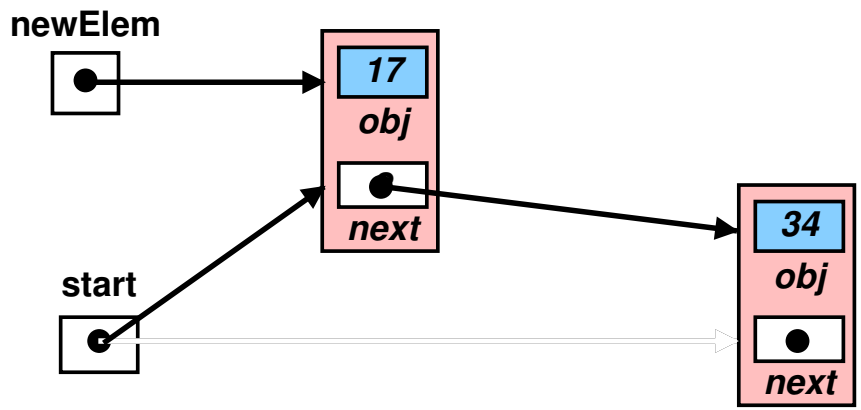
Insert into empty list:

```
1      int key = getKey(newObj);
2      Elem newElem = new Elem(newObj);
3
4      if (start == null ) {
5         newElem.setNext (null);     start = newElem;
6         return;
7      }
```



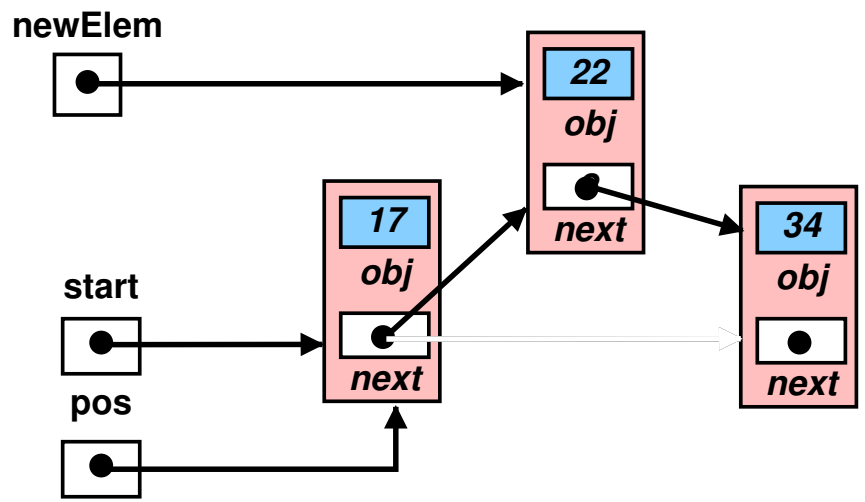
Insert before first element:

```
1       if (getKey(start.getObject()) > key ) {
2           newElem.setNext (start);     start = newElem;
3           return;
4       }
```



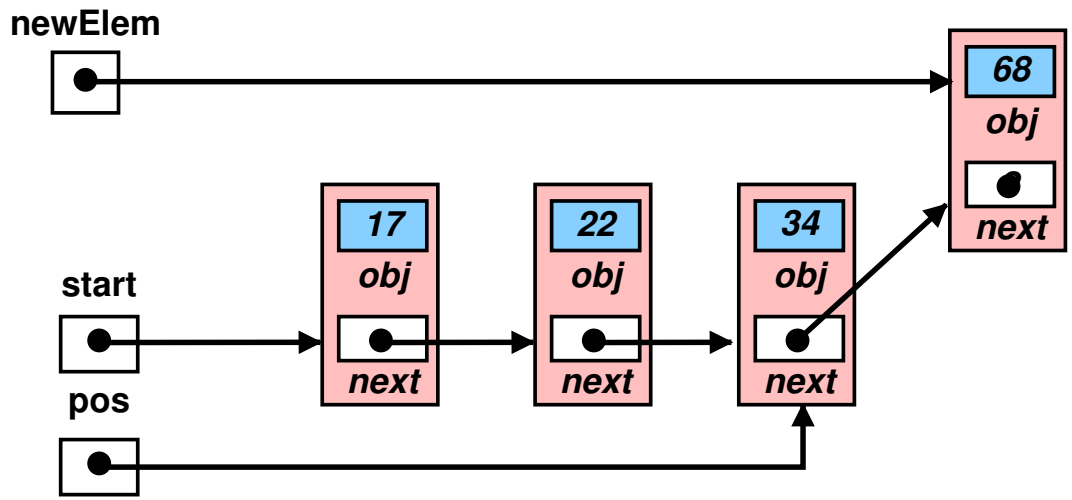
Insert between two elements:

```
1        Elem pos = start;
2        while ( pos.getNext() != null)
3          if (getKey(pos.getNext().getObject()) > key) {
4              newElem.setNext (pos.getNext());
5              pos.setNext (newElem);
6              return;
7          } else
8              pos = pos.getNext();
```



Insert at end of list: (`pos` is already set!)

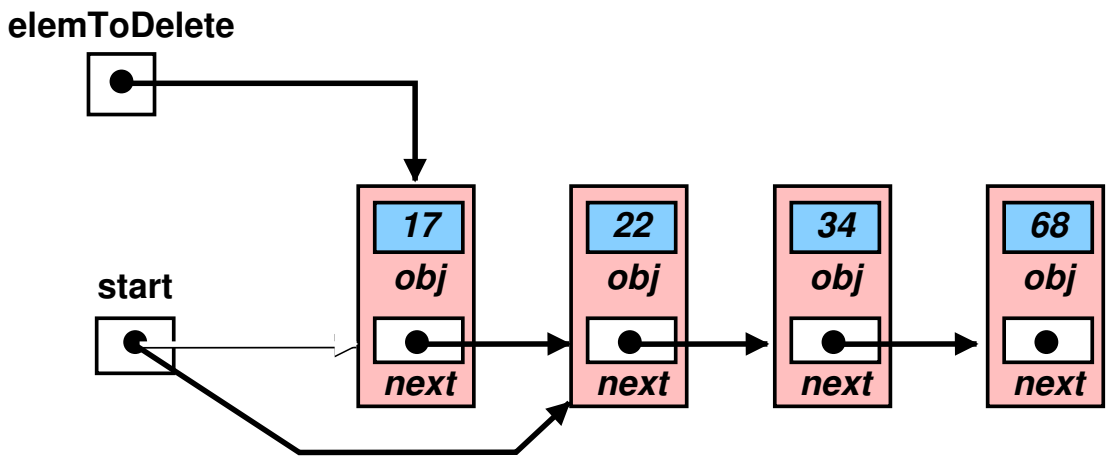
```
1       pos.setNext(newElem);
2       newElem.setNext(null);
```



```
1     public void delete (Elem elemToDelete) {
2  // empty list or nothing to delete
3        if (start == null|| elemToDelete == null)
4           return;
5  // delete the first element
6        if (start == elemToDelete) {
7           start = elemToDelete.getNext();
8           return;
9        }
10 // delete element at other position
11       Elem pos = start;
12       while ( pos.getNext() != null )
13          if ( pos.getNext() == elemToDelete ) {
14             pos.setNext(elemToDelete.getNext());
15          return;
16          }
17          else
18             pos = pos.getNext();
19    }
```

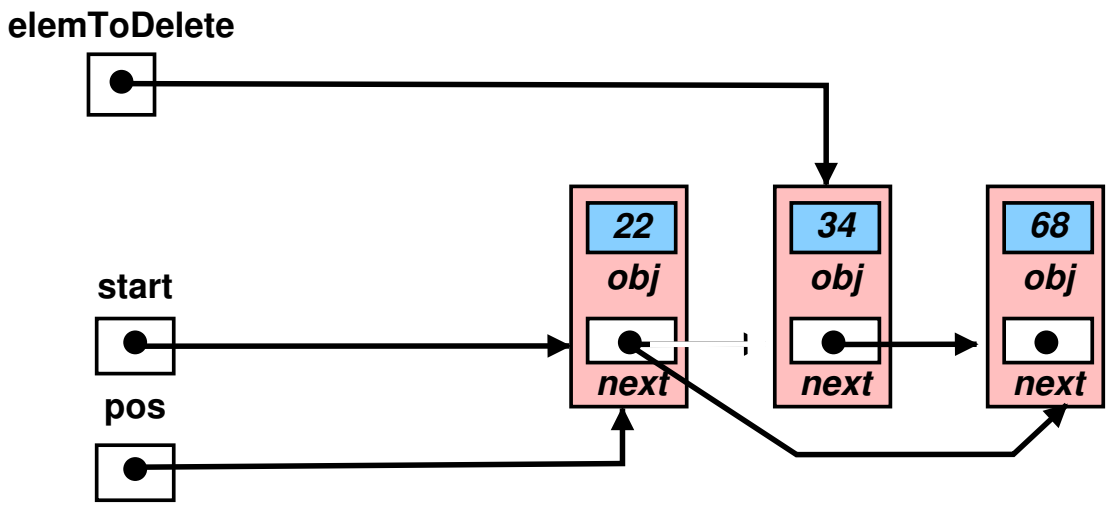
Delete first element:

```
1 if (start == elemToDelete) {
2           start = elemToDelete.getNext();
3           return;
4       }
```



Delete other elements:

```
1        Elem pos = start;
2        while ( pos.getNext() != null )
3           if ( pos.getNext() == elemToDelete ) {
4              pos.setNext(elemToDelete.getNext());
5              return;
6           }
7           else
8              pos = pos.getNext();
```



Test Program for `SortedList` in `K5B07E_SortedList`:

```
1  public class SortedListTest {
2    public static void main(String[] args) {
3      SortedList liste = new SortedList();
4      while ( true ) {
5        String str = System.console().readLine
6             ("Input [key|-key|0]: ");
7        if (str.equals("0")) {
8          return;
9        } else {
10         int key = Integer.parseInt(str);
11         if (key > 0) {
12           liste.sortIn (new Integer(key));
13           System.out.println("Insert:␣" + key);
14           System.out.println("List:␣␣␣" + liste);
15         } else {
16           key = - key;
17           liste.delete (liste.lookUp (key));
18           System.out.println("Delete:␣" + key);
19           System.out.println("List:␣␣␣" + liste);
20         }
21       }
22     }
23   }
24 }
```

Doubly linked lists:

- A list is processed following the references.
- With a singly linked list, a step backwards usually requires to start again at the head of the list.
- If backward steps are frequent, a **doubly linked list** should be preferred.

To demonstrate the use of references, we discuss a simplified version (e.g., without special methods for backward search).

In the example `K5B08E_SortedDoubleList` the test class `SortedListTest` is identical to the previous example, but the classes `Elem` and `SortedList` are modified.

Element with forward and backward linkage, with additional `prev` reference

```
1  public class DLElem {
2
3      private Object obj;
4      private DLElem next;
5      private DLElem prev;      reference to previous element
6
7      public DLElem () {}
8      public DLElem (Object obj) { setObject(obj); }
9
10     public void setObject (Object newObj) { obj = newObj;}
11     public Object getObject () { return obj; }
12
13     public void setNext (DLElem nextElem) { next = nextElem; }
14     public void setPrev (DLElem prevElem)  prev = prevElem;
15
16     public DLElem getNext () { return next; }
17     public DLElem getPrev ()  return prev;
18
19     @Override
20     public String toString () { return obj.toString ();  }
21 }
```

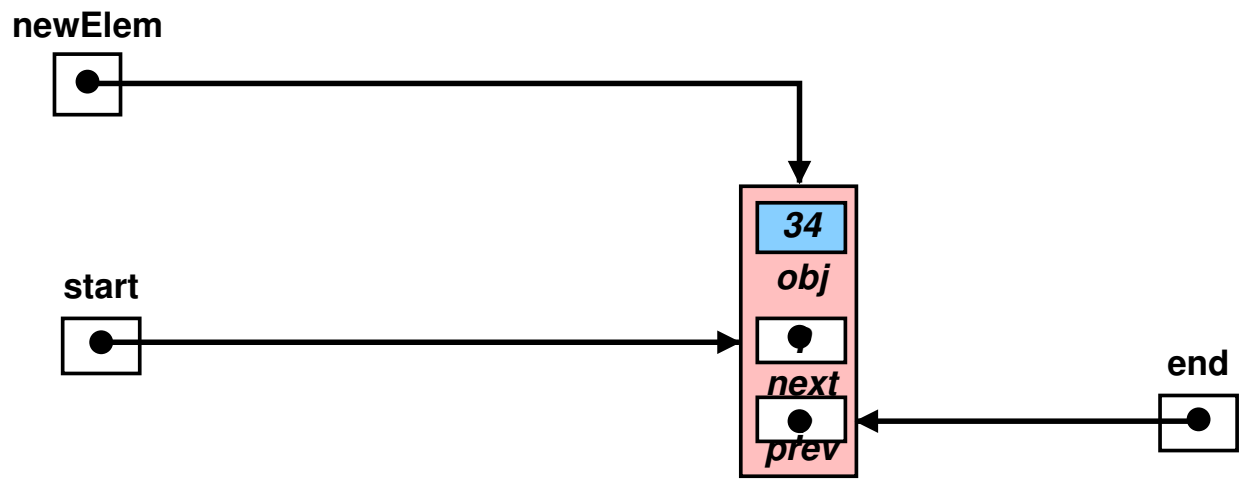
```
1  public class SortedDoubleList {
2
3   public SortedDoubleList ()  { }   constructor
4   private DLElem start;              start of list
5   private DLElem end;               end of list
6  sorting key as before:
7   public static int getKey(Object obj) { // sorting key
8        return  (Integer)obj;    // only for Integer objects
9   }
10 identical to single linkage:
11  public DLElem lookUp (int keyToLookup)  { ... }
12 more methods for list access
13  public void sortIn (Object newObj)      { ... }
14  public void delete delete (DLElem elemToDelete) { ... }
15
16 for debugging: list forward and backward...
17  @Override public String toString () {
18   String str = "␣--->␣";
19   for (DLElem pos = start; pos != null; pos = pos.getNext())
20       str += (Integer)pos.getObject() + "␣␣";
21   str += ",␣␣<---␣";
22   for (DLElem pos = end; pos != null; pos = pos.getPrev())
23       str += (Integer)pos.getObject() + "␣␣";
24   return str;
25 } }
```

The other methods are significantly different from the previous example:

```
1  public void sortIn (Object newObj) {
2    int key = getKey(newObj);
3    DLElem newElem = new DLElem(newObj);
4  //insert in empty list
5    if (start == null ) {
6       ...
7    }
8  // insert before first element
9    if ( getKey(start.getObject()) > key ) {
10      ...; return;
11   }
12 // insert between two elements
13   DLElem pos = start;
14   while ( pos.getNext() != null)
15      if ( getKey(pos.getNext().getObject()) > key ) {
16         ...;  return;
17      }
18      else pos = pos.getNext();
19 // insert at the end of the list
20   pos.setNext(newElem);     end = newElem;
21   newElem.setNext(null);    newElem.setPrev(pos);
22 }
```

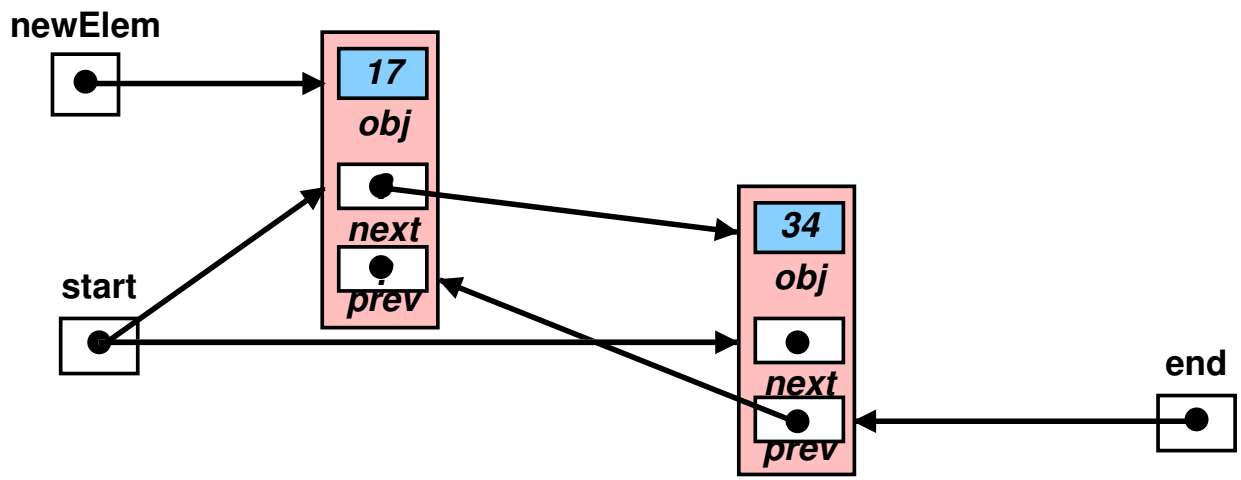
Insert in empty list:

```
1   if (start == null ) {
2       newElem.setNext (null);  newElem.setPrev (null);
3       start = newElem;  end = newElem;
4       return;
5   }
```



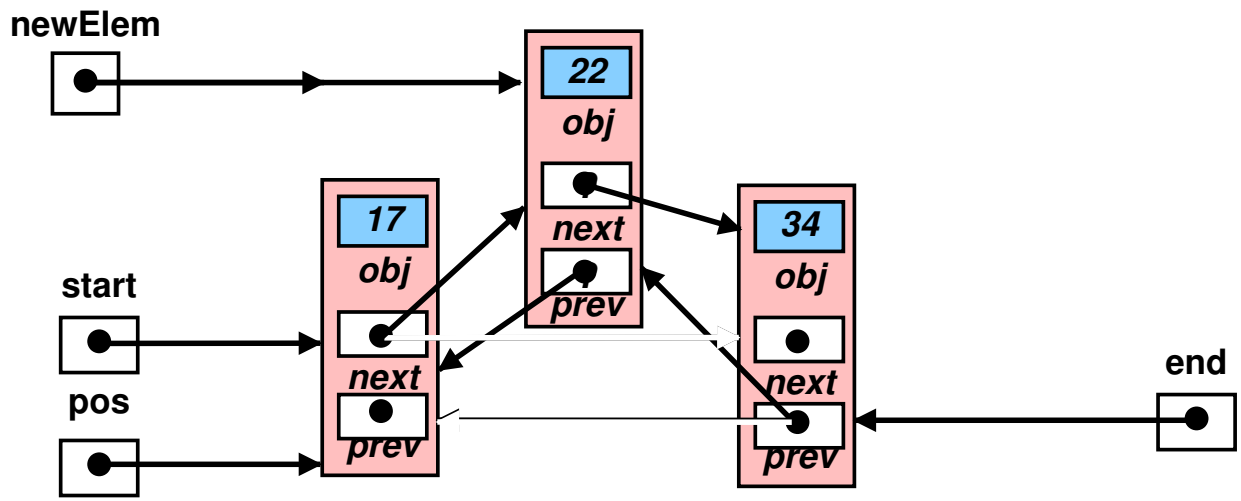
Insert before first element:

```
1   if ( getKey(start.getObject()) > key ) {
2       start.setPrev (newElem);  newElem.setNext (start);
3       newElem.setPrev (null);   start = newElem;
4       return;
5   }
```



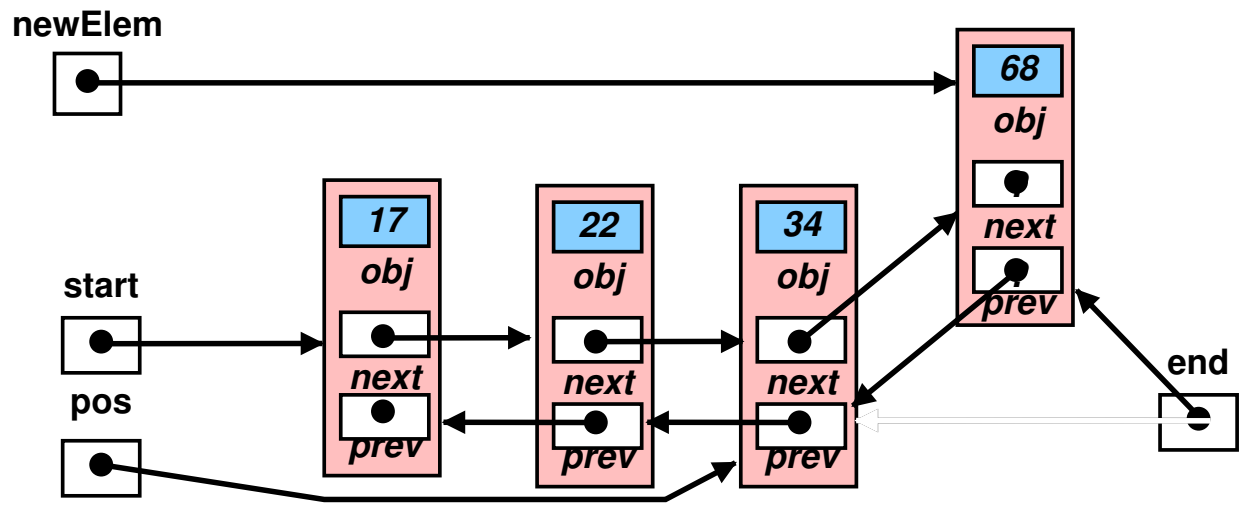
Insert between two elements:

```
1   DLElem pos = start;
2   while ( pos.getNext() != null)
3      if ( getKey(pos.getNext().getObject()) > key ) {
4         pos.getNext().setPrev(newElem);
5         newElem.setNext (pos.getNext());
6         newElem.setPrev (pos);
7         pos.setNext (newElem);
8         return;
9      }
10     else pos = pos.getNext();
```



Insert at end of list:

```
1    pos.setNext(newElem);     end = newElem;
2    newElem.setNext(null);    newElem.setPrev(pos);
```



Deletion of arbitrary element,
with special treatment for first/last element:

```
 1  public void delete (DLElem elemToDelete) {
 2    if (elemToDelete == null )
 3      return;
 4    if (elemToDelete.getPrev() != null)
 5      elemToDelete.getPrev().setNext(elemToDelete.getNext());
 6    else // special case first element:
 7      start = elemToDelete.getNext();
 8    if (elemToDelete.getNext() != null)
 9      elemToDelete.getNext().setPrev(elemToDelete.getPrev());
10    else // special case last element:
11      end = elemToDelete.getPrev();
12  }
```

Remove from double linkage:

```
1 ...
2 elemToDelete.getPrev().setNext(elemToDelete.getNext());
3 ...
4 elemToDelete.getNext().setPrev(elemToDelete.getPrev());
5 ...
```