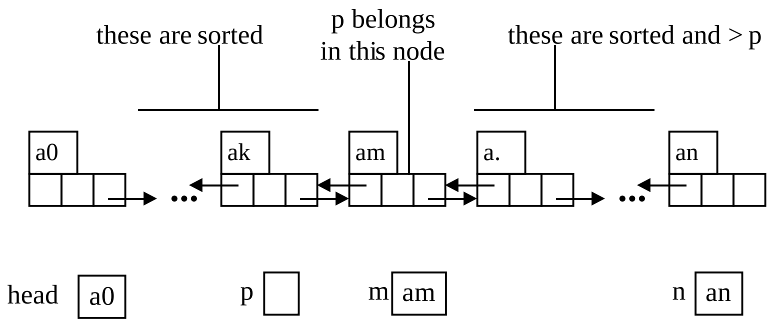
Recitation 9. Practice with linked lists

Introduction. This recitation will go over the linked list implementation given on the back of this sheet and show you some uses of it. The left column below gives an insertion sort algorithm. This recitation does not consider list iterators, which are a better structuring technique but in this case make the algorithms harder to write and perhaps to understand.

```
// Perform an insertion sort on b
public static void insertionsort(DoublyLinkedList b){
    ListNode p= b.first();
    // inv: p is the name of a node in b (or is the tail).
    //       The nodes before p are sorted.
    while (b.isValid(p)) {
        insert(b,p);
        p= p.next;
    }
}

// Nodes preceding node n in list b are sorted. Push n
// down to its sorted place in b.
// Precondition: b.isValid(p)
public static void insert(DoublyLinkedList b,
                               ListNode n) {
    ListNode m= n;
    int p= ((Integer)m.element).intValue();
    // invariant: value p belongs in node m.
    //   Nodes following m up to n are sorted and >= p
    //   Nodes preceding m are sorted
    while (b.hasPrec(m) ) {
        ListNode mprev= b.prev(m);
        int beforeP=
                ((Integer)mprev.element).intValue();
        if (beforeP <= p) {
            m.element= new Integer(p);
            return;
        }
        m.element= new Integer(beforeP);
        m= mprev;
    }
    m.element= new Integer(p);
}
```

invariant of insert's loop



```
/** An instance: doubly linked list with head, tail */
public class DoublyLinkedList {
   ListNode head; // the header node of this list.
                  // head.prev = null, always
   ListNode tail;  // the tail node of this list
                  // tail.next = null, always
   int size;      // number of nodes in this list

   /** Constructor: an empty linked list */
   public DoublyLinkedList() {
      head= new ListNode(null, null, null);
      tail= new ListNode(head, null, null);
      head.next= tail; size= 0;
   }

   /** = the linked list is empty */
   public boolean isEmpty()
      {  return size == 0;  }

   /** Remove all items from this linked list */
   public void makeEmpty() {
      head.next= tail;
      tail.prev= head;
      size= 0;
   }

   /** = size of this linked list */
   public int size()
      {  return size;  }

   /** = the first node in the list (tail if none) */
   public ListNode first()
     {  return head.next;  }

   /** = the last node in the list (head if none) */
   public ListNode last()
     {  return tail.prev;  }

   /** = "p is valid --is not one of head and last" */
   public boolean isValid(ListNode p)
      {  return p != head && p != tail; }

   /** = "a node of the list follows p" */
   public boolean hasNext(ListNode p)
      {  return p != tail && p.next != tail;  }

   /** = "a node of the list precedes p" */
   public boolean hasPrec(ListNode p)
      {  return p != head && p.prev != head;  }

   /** = node following node p (null if p is the tail)
        Precondition: p is not null */
   public ListNode next(ListNode p)
      {  return p.next;  }

    /** = node preceding node p (null if p is head)
        Precondition: p is not null*/
   public ListNode prev(ListNode p)
      {  return p.prev;  }

   /** if p is not tail, insert object x after node p. */
   public void insert(Object x, ListNode p) {
      if (p == tail)
         return;
      ListNode n= new ListNode(p, x, p.next);
      p.next.prev= n;
      p.next= n;
      size= size+1;
   }

   /** if p is not head or tail, delete p from list.  */
   public void delete(ListNode p) {
      if (!isValid(p))
         return;
      p.prev.next= p.next;
      p.next.prev= p.prev;
      size= size-1;
   }

   /** = a string that contains the elements of the list
      separated by commas and enclosed in "(" ")". */
   public String toString() {
      String s= "(";
      ListNode c= head.next;
      while (c != tail) {
         s= s + c.element;
         c= c.next;
         if (c != tail)
            s= s + ", ";
      }
      return s + ")";
   }
}

/** An instance is a node of a doubly linked list */
public class ListNode {
    /** name of previous node (null if none) */
    public ListNode prev;
    /** value in the node */
    public Object element;
    /** name of next node (null if none) */
    public ListNode next;  // name of next node
                           // (null if none)
    /** Constructor: a ListNode with element e,
                      prev field p, and next field n */
   public ListNode(ListNode p, Object e,
                      ListNode n)
     {  prev= p; element= e; next= n;  }
}
```