University of Washington

Department of Computer Science and Engineering

CSE 373

Problem Set 2 Solution

1. (2 points) Linked List vs. Array for Polynomials

   The main advantage of a linked list is that space is only used to store the non-zero coefficients, whereas with an array one typically stores the coefficients for all terms up to the degree of the exponent. For example, the representation of $1 + x^{1000}$ would be a two-element linked list, but a thousand-element array filled mainly with zeroes.

2. (2 points) Singly- vs. Doubly-linked Lists

   The disadvantages of a doubly-linked list are operations which are slightly more complicated to implement, and an increase in memory consumption (there is an extra pointer stored for every element of the list; in the case of lists of "small" types, this may be a significant percentage increase).

   On the other hand, the "FIND_PREVIOUS" operation becomes trivial (O(1)) for the doubly-linked list, whereas it is $O(n)$ for the singly-linked list. In particular, this means that REMOVE for doubly linked lists is O(1) instead of $O(n)$.

3. (4 points) List Intersection

   Assume that lists are represented by pointers to their first elements, are sorted in increasing order, and that neither has duplicate elements. If L_1 and L_2 have lengths m and n respectively and are in sorted order, we can use a merge-like operation to find their intersection in $O(m + n)$ time as shown below; if the lists are unsorted, we could either compare all elements in $O(mn)$ time, or — even better — sort them and then use the algorithm below, for a total time of $O(m \log m + n \log n)$.

```
ListP intersection(ListP L1, ListP L2)
{
  ListP result;

  while (L1 and L2 are not empty) do {
    if (L1.value < L2.value) then
      L1 = L1.next;
    else if (L2.value < L1.value) then
      L2 = L2.next;
    else { /* They must be equal, so the element is in
              the intersection */
```

```
      /* Here, I'm just assuming list_insert places L1.value
         at the head of the list.  It's okay to assume ``obvious''
         operations like this because it's more clear than
         placing all the pointer operations for an insertion
         into the pseudocode.  Note that if we want to get
         the intersection in sorted order, we would also want
         to keep a tail pointer to the result list and use
         it to do a list_append instead of inserting at the head.
      */
      list_insert(result, L1.value);
      L1 = L1.next;
      L2 = L2.next;
    }
  }
  return result;
}
```

4. (6 points) Polynomial Addition

I think we've seen this code before; polynomial addition looks a lot like the previous routine in flavor, and takes $O(m+n)$ time. Here, I've demonstrated a recursive version in contrast to the previous solution's iterative version.

```
/* Assumes that p1 and p2 are in decreasing order
   by exponent, and that p1 and p2 do *not* have
   duplicated exponents.  The result is a sorted
   list with no duplicated exponents. */
ListP merge(ListP p1, ListP p2)
{
   ListP result;

   if (p1 == null) then
     return copy_list(p2);

   if (p2 == null) then
     return copy_list(p1);

   if (p1.exp > p2.exp) then {
     result = duplicate_node(p1);
     result.next = merge(p1.next, p2);
   }
   else if (p1.exp < p2.exp) then {
```

```
    result = duplicate_node(p2);
    result.next = merge(p1, p2.next);
  }
  else /* p1.exp == p2.exp */
    /* We have an if here to avoid inserting a 0 coef. */
    if (p1.coef + p2.coef != 0) {
      result = new node(p1.coef + p2.coef, p1.exp);
      result.next = merge(p1.next, p2.next);
    } else {
      result = merge(p1.next, p2.next);
    }
  return result;
}
```

5. (2 points) Lazy Deletion

   The big advantage of lazy deletion is the time savings. It takes only O(1) time to mark an element for deletion, but in a singly-linked list, the time for a single "standard" deletion is $O(n)$ because we must search for the previous element. However, notice that in $O(n)$ steps we can delete as many elements as we want. Hence, by "batching" deletions together we can effectively make each one cheaper. [Note: later in the quarter we will be able to show that this scheme gives deletion an "amortized cost" of O(1).]

   Of course, the disadvantage is that the list may become cluttered with lots of items marked for deletion. This does not raise the time complexity of list operations in the big-oh sense as long as we never allow the number of deleted elements to be more than linear with respect to the number of non-deleted elements, but it still wastes memory and makes the operations slower in practice.

6. (4 points) List Representations and Queues

   If all we have is a pointer to the head of the list, then DEQUEUE still takes O(1) time by removing from the head of the list. However, ENQUEUE now takes $O(n)$ time for a queue of size n, since we must traverse to the end of the list to perform the queue insertion.

   To correct this problem, we should also keep a "tail pointer" that points to the last element in the list. This allows us to do the ENQUEUE operation in O(1) time as well. This strategy is helpful not just for queues, but for any list in which appending is a common operation.

   As for the advantage of the array representation, it's mainly in ease of implementation and practical speed. The array implementation might be slightly faster in practice, but not so much that it would be noticed unless you were doing millions of queue operations. The chief disadvantage is that the maximum size is fixed.