Data Structures and Algorithms — CSCI 230
Chapter 3 — Review Solutions

1. The standard `vector` container class provides neither `push_front` nor `pop_front` operations because these are inefficient operations on an array: all remaining values would have to be shifted over by one location to fill the empty space — a linear time operation. The standard library is designed to explicit prevent such inefficiencies by not providing such operations. Hence, the `vector` container class should not be used for a queue.

   Note: the buffer class we investigated in lab 1 used a fixed sized array for efficient queue operations. Requiring such a fixed sized is too restrictive, however.

2. (a) Assuming k is small relative to n, the vector would be kept in increasing order of id number. Each insert operation then would be $O(k)$ worst-case because at most k `foos` at the end of the vector would need to be shifted over to insert a new `foo`. The search operation could then use binary search because of the sorted nature, requiring $O(\log n)$ time. The print operation would just be $O(n)$ because the `foos` would already be sorted.

   If $k = \theta(n)$ and search and print operations are extremely rare (occurring, say $O(1)$ times total), then keeping the vector unsorted and just inserting at the end would make sense. You really need not have considered this case.

   (b) For linked list, the choice of sorted vs. unsorted, which determines the efficiency of various operations, is less clear. When the list is sorted, insert requires $O(k)$ time, search $O(n)$ and print would also be $O(n)$. When the list is unsorted, insert requires $O(1)$, search $O(n)$ (though the constant multiplier on n is twice that of the sorted case), and print would be $O(n \log n)$. Thus, the cost (and the choice) depends on the relative frequency of insert vs. print operations.

   (c) For small k the vector is the better choice. For larger k, the unsorted list might be better if insert is much more frequent than search or print.

3. Here is the function for `lists`:

```
void
every_nth( istream & instr, int n )
{
  list<float> values;
  float in_val;
  while ( instr >> in_val ) values.push_back(in_val);
  list<float>::reverse_iterator p = values.rbegin();
```

```
  int i=0;
  for ( ; p != values.rend(); ++p, ++i )
    if ( i%n == 0 ) cout << *p << endl;
}
```

Note that you might have used two nested loops for the printing operation. That is likely to be an equally valid solution. Also, you could have called the `reverse` function and then just used an ordinary iterator.

The solution for `vector` is exactly the same, with the word `list` replaced by the word `vector`!

4. Here is the solution for singly-linked list. Note that at the end of each loop iteration, `p` points to the head of the reversed part of the list and `q` points to the head of the unreversed part and `*p` is the former predecessor of `*q`.

```
void Reverse( Node* & head )
{
  if ( !head ) return;
  Node * p = head;
  Node * q = head->next;
  while (q) {
    Node * r = q->next;
    q->next = p;   // reverse the direction of node *q
    p = q;
    q = r;
  }
  head = p;
}
```

Here is the solution for doubly-linked list. Note that fewer scratch pointers are needed because each node knows both its predecessor and successor.

```
void Reverse( Node* & head )
{
  if ( !head ) return;
  do {
    Node * old_next = head->next;
    head->next = head->prev;
    head->prev = old_next;
    if ( !old_next ) break;  // end of list has been reached
    head = old_next;
  } while (true);
}
```