(a) To complete the meld in linear time, we use two chain iterators `a` and `b` to march through the chains `A` and `B` respectively. When we fall off of one chain, the balance of the remaining chain is copied over to `C`. Since elements from `A` and `B` are to be added to the end of the chain `C`, we use the `Append` function defined in the file `echain.h`. The code is

```
template<class T> void Alternate(const Chain<T>&
A,
          const Chain<T>& B, Chain<T>& C) {//
Meld alternately from A and B to get C.
   // initialize
   ChainIterator<T> a,  // iterator for A
                    b;  // iterator for B
   T *DataA = a.Initialize(A); // first element
of A
   T *DataB = b.Initialize(B); // first of B
   C.Erase(); // empty C

   // create result
   while (DataA && DataB) {
      C.Append(*DataA);
      C.Append(*DataB);
      DataA = a.Next();
      DataB = b.Next();
      }
   // append the rest
   // at most one of A and B can be nonempty now
   while(DataA) {
      C.Append(*DataA);
      DataA = a.Next();
      }
   while(DataB) {
      C.Append(*DataB);
      DataB = b.Next();
      } }
```

(b) The call to `Erase` takes an amount of time that is linear in the length of the initial `C`. Each call to `Initialize`, `Append`, and `Next` takes $\Theta(1)$ time. So the time spent in all of the `while` loops is linear in the sum of the lenths of the chains `A`, `B`, and `C`. As a result, the complexity of `Alternate` is linear in the sum of the lengths of the three inital chains `A`, `B`, and `C`. The dependence on the initial length of `C` can be removed by using `Zero()` in place of *Erase*. If we do this, the nodes initially in *C* will not be deleted and the space will not become available for reuse by the program.

(c) and `caltern1.out`.