Section Handout

Discussion Problem 1: Braided Lists

Write a function called **`braid`** that takes the leading address of a singly linked list, and weaves the reverse of that list into the original.

```
struct node {
   int value;
   node *next;
};
```

Here are some examples:

`list`	**`list after call braid(list)`**
1 → 4 → 2	1 → 2 → 4 → 4 → 2 → 1
3	3 → 3
1 → 3 → 6 → 10 → 15	1 → 15 → 3 → 10 → 6 → 6 → 10 → 3 → 15 → 1

You have this page and the next page to present your solution.

```
static void braid(node *list);
```

Discussion Problem 2: `append` Two Ways

Assume the following node definition:

```
struct node {
   int value;
   node *next;
};
```

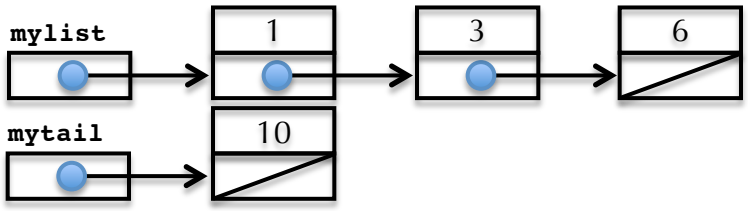
and consider the following two recursive functions, which differ only by the placement of a single **`&`**. (Normally tail recursion is a no-no, but I'm using here to exercise your understanding of memory and references.)

```
void append1(node *list, node *tail) {
   if (list == NULL) {
      list = tail;
   } else {
      append1(list->next, tail);
   }
}
```

```
void append2(node *& list, node *tail) {
   if (list == NULL) {
      list = tail;
   } else {
      append2(list->next, tail);
   }
}
```

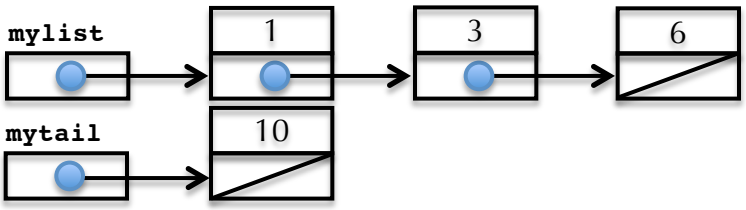
Presumably, each exists with the intent of tacking the node addressed by **`tail`** to the end of the full list addressed by **`list`**. **`append2`** works, and **`append1`** doesn't.

Assume the following illustration captures exactly how variables **mylist** and **mytail** [each of type **node ***] have been initialized just prior to a call to one of the two append functions:

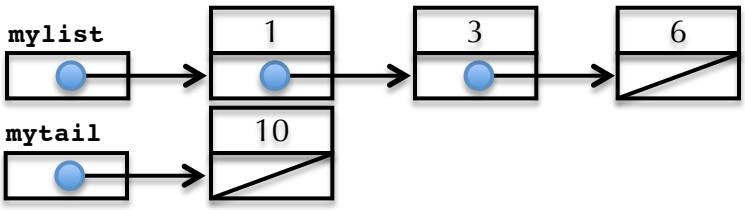


We want you to draw the state of memory just before **list = tail** executes to see exactly how it is that **append1(mylist, mytail)** fails even though **append2(mylist, mytail)** succeeds.

a. First draw the state of memory as the primary call to **append1(mylist, mytail)** bottoms out and just before its **list = tail** statement executes. You'll want to draw all of the parameters for all four function calls, being clear what each of the parameters associated with each of the recursive calls contains.



b. Do the same thing again, this time for a primary call to **append2(mylist, mytail)** [again, just before its **list = tail** statement executes]. Again, draw all of the parameters for all four function calls, being clear what each of the parameters associated with each of the recursive calls contains.



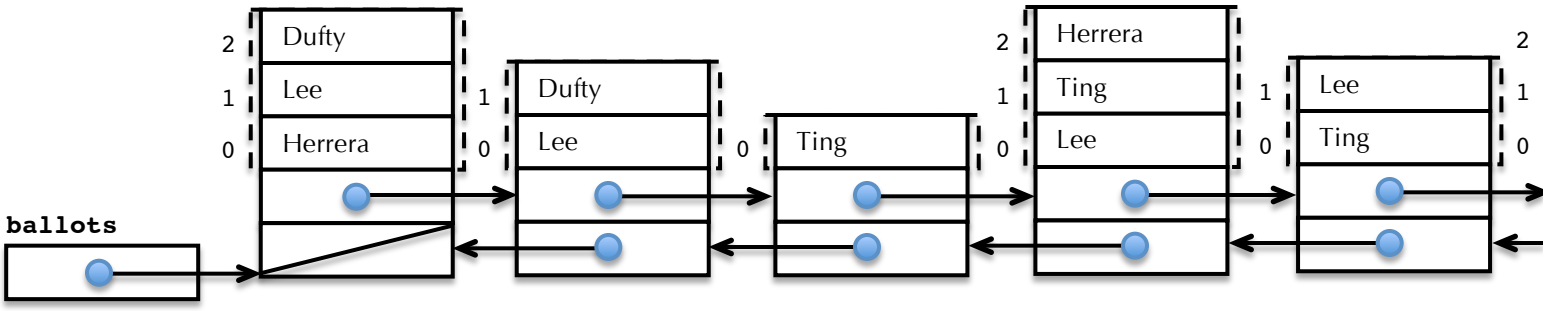
Discussion Problem 3: Ranked Choice Voting

Ranked choice voting—also knows as instant runoff voting—is used in San Francisco for mayoral elections. Rather than voting for a single candidate, those casting ballots vote for up to **three** candidates, ranking them 1, 2, and 3 (or, in computer science speak: 0, 1, and 2.)

Assume you are given the following to represent a single ballot:

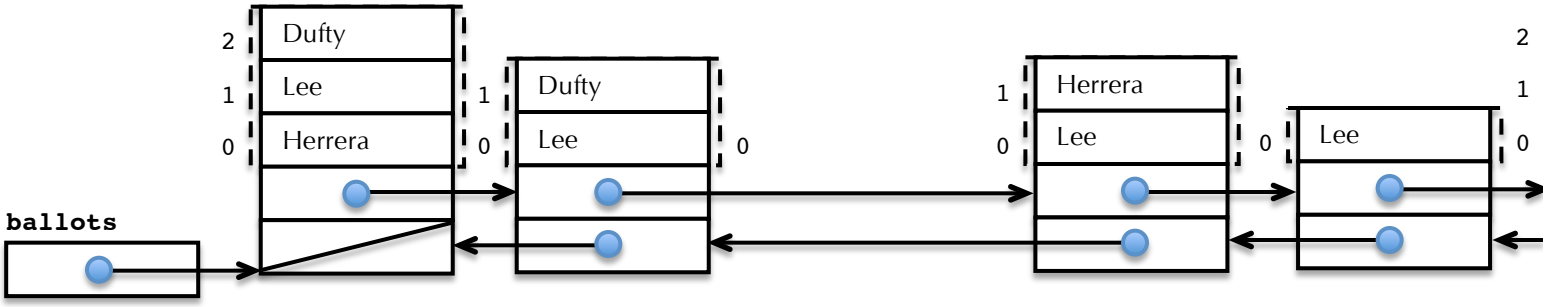
```
struct ballot {
   Vector<string> votes; // of size 1, 2, or 3; sorted by preference
   ballot *next;
   ballot *prev;
};
```

and that the collection of all ballots is represented as a doubly linked list. The first five of what in practice would be tens of thousands of ballots in a real San Francisco election might look like this:



Initially, only first place votes matter, and if a single candidate gets the majority of all first place votes, then that candidate wins. Often, no one gets a majority of all first place votes [There were, for instance, 16 official candidates in San Francisco's mayoral election on November 8th, 2011 and Ed Lee, who eventually won, only got only 31% of the first choice votes.] In that case, the candidate with the least number of first place votes is eliminated by effectively removing that candidate from all ballots everywhere (the rank choice voting literature says these votes are **exhausted**) and promoting all second and third place votes to be first and second place votes to close any gaps.

If, after an analysis of the ballots list above it's determined that Phil Ting received the smallest number of first place votes, the ballots list would be updated to look like this:



The first two ballots were left alone, but the next three were updated to reflect Ting's elimination. Note the one node that included a standalone vote for Phil Ting was removed from the list, since it no longer contains any valid votes. The two other impacted nodes each saw candidates Dennis Herrera and Ed Lee advance from third and second to second and first, respectively.

The process is repeated over and over again until it leaves one candidate with a majority of rank-one votes. [On November 8th, 2011, this very process was applied 12 times before Ed Lee prevailed with 61% of all remaining first choice votes.]

a. Implement the **`identifyLeastPopular`** function that, given a doubly linked list of ballots called **`ballots`**, returns the name of the candidate receiving the smallest number of first-choice votes. You may assume all ballots include at least one vote, that no ballots ever include two votes for the same candidate, and that if two or more candidates are tied for least popular [maybe Phil Ting and Bevan Dufty, for instance, each get only two first-choice votes and no one got only one], then any one of them can be returned.

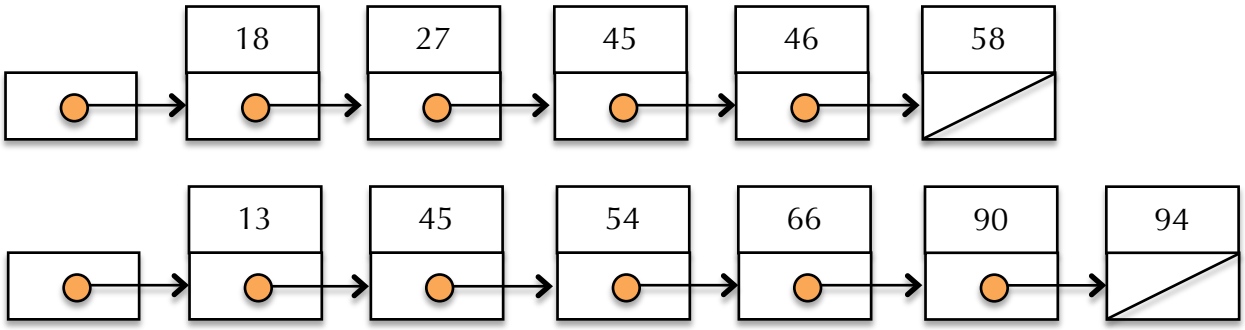
```
static string identifyLeastPopular(ballot *ballots);
```

b. Next implement the **`eliminateLeastPopular`** function which, given a doubly linked list of **`ballots`** and the **`name`** of the candidate to be eliminated, removes all traces of the candidate from the list of ballots, removing and properly disposing of any ballots depleted of all votes. Ensure that you properly handle the case where the first or last ballot (or both) is removed.

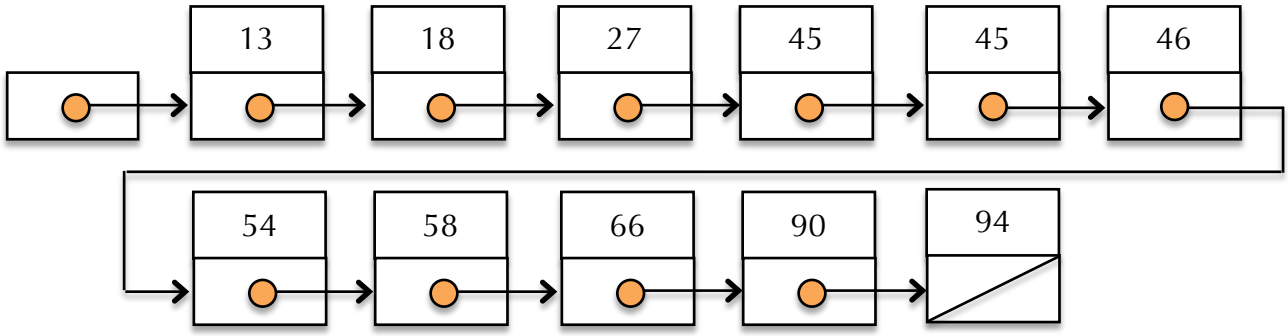
```
static void eliminateLeastPopular(ballot *& ballots, const string& name);
```

Lab Problem 1: Merging Lists

Write a function called **`mergeLists`** that, given two sorted linked lists of potentially different lengths, merges the two into a single list. Your implementation shouldn't allocate any new memory, but should instead use the nodes making up the two originals. So, given the following lists,



`mergeLists` would synthesize the following and return the address of the node surrounding the 13.



Implement to the following record definition and prototype. Note that your implementation shouldn't allocate or free any nodes at all, and it should run in time that's proportional to the length of the final list. We've given you a test harness that creates two sorted lists of varying lengths, and your job is to complete the implementation of the **`mergeLists`** function.

```
struct node {
   int value;
   node *next;
};

static node *mergeLists(node *one, node *two);
```