CS106X
Autumn 2012

Handout 28
October 29th, 2012

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:

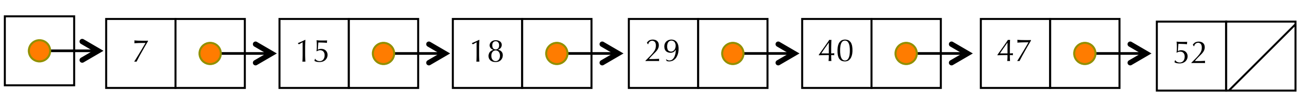
`list`	`list after call braid(list)`
1 → 4 → 2	1 → 2 → 4 → 4 → 2 → 1
3	3 → 3
1 → 3 → 6 → 10 → 15	1 → 15 → 3 → 10 → 6 → 6 → 10 → 3 → 15 → 1

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

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

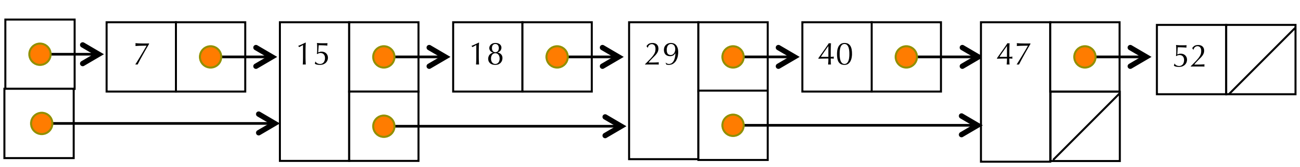
Discussion Problem 2: Searching Skip Lists

Imagine the sorted, singly linked list drawn below:



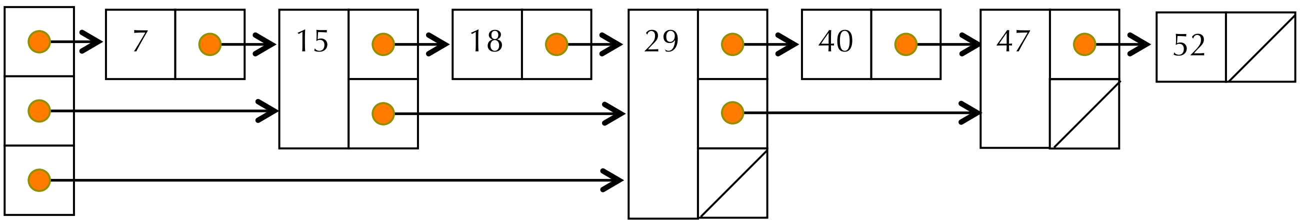
In spite of its being sorted, it still takes linear time to find the 52, or to confirm that something like 63 isn't present. We've discussed this very point in lecture: linked lists—even sorted ones—don't provide random access to arbitrary elements in the sequence, so we don't have anything close to binary search available to us.

Now imagine every other node in the list has **two** pointers, and the second pointer in each actually addresses the node two in front of it. Here's the new picture:



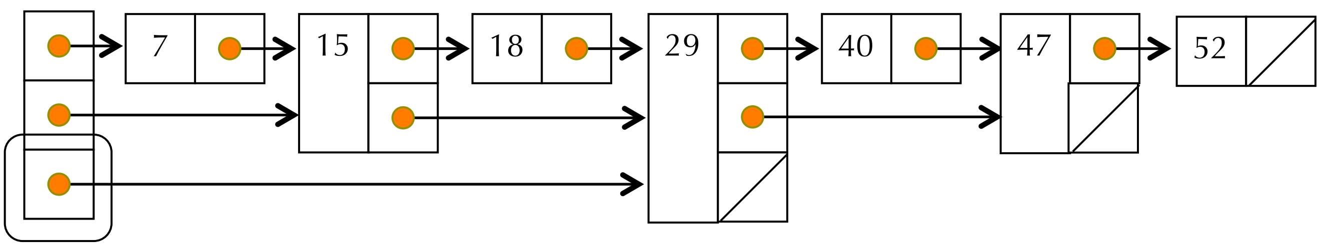
Initially, you traverse through the second level of links while you can, and then if need be, move to the original level of pointers and continue. So, a search for 47 would require we travel through three pointers. A search for 40 would require an examination of the same three pointers, except we'd detect the third one led to a node housing a number larger than the 40 and try a fourth: the one extending from the 29 to the 40. This doesn't technically improve the asymptotic running time of search, but it's a gesture in the right direction.

Take this one level further and introduce a third level of pointers:

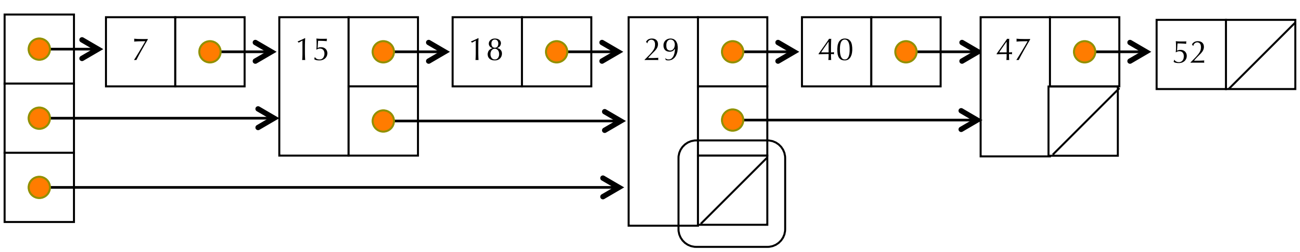


Search would start with the level-2 pointers, continuing with the level-1 pointer if the level-2 pointers dead end, and moving down to the level-0 pointers if the level-1 pointers dead end. Now we're working toward a data structure that, if extended to allow more and more levels of pointers, can support sub-linear search time.

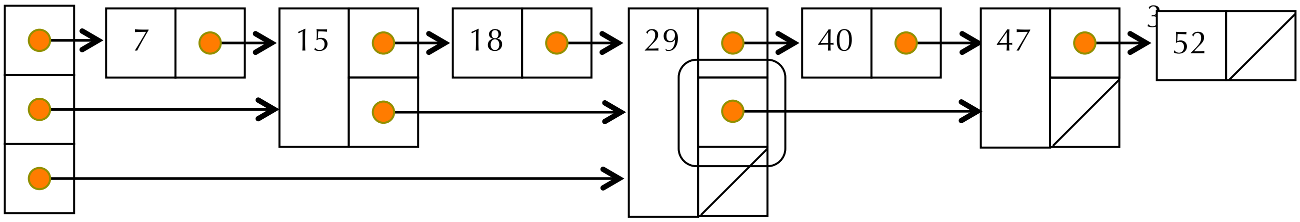
Want to know how we can search for the 40? Here's a short film identifying the node you visit and the pointers you dereference in order to hone in on that 40.



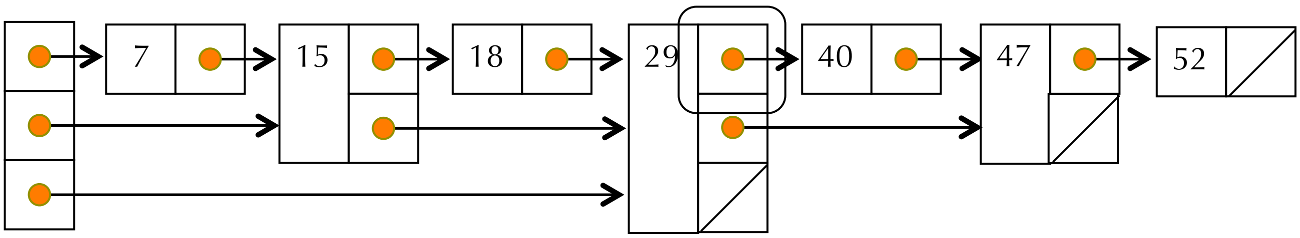
1.) Start with the last of the leading pointers, which identifies a node with a 29 inside of it. 29 is less than the 40 we're looking for, so advance one node at level 2.



2.) Now we're on the 29 node, and see that its level-2 pointer is **`NULL`**. *There are lower levels to explore, so we stay on the same node, but descend from level 2 to level 1.*



*3.) We're still on the 29 node, but we've demoted the search to using level-1 next pointers. The level-1 pointer is non-**NULL**, but it addresses a node with a 47 inside of it. That 47 is too big—certainly bigger than the 40, so we descend another level and continue with level-0 pointers.*



4.) We're still on the 29 node, working at level-0 now. The relevant pointer is addressing the node with a 40 inside, so we know the 40 exists. Yay, the 40 exists!

In a nutshell, you make as much progress as you can at the higher-level pointers, and as needed, you descend to lower-level pointers to explore the rest of the list while skipping less.

The generalization of all this is the **skip list**, which is a sorted, linked structure where each node contains a vector of next pointers—where the vector may be of length 1, 2, 3, or more. The only requirement is that the k^{th} pointer in the vector hop at least as far as the $k - 1^{th}$ pointer.

Assume that the skip list node is defined as follows:

```
struct skipListNode {
   int value;
   Vector<skipListNode *> links;
};
```

Write a function call **skipListContains**, which takes a **Vector<skipListNode *>** called **heads**, where the **skipListNode *** at index **k** contains the address of the first node with a level-**k** pointer, and returns **true** if and only if the supplied **key** is present somewhere in the list. Your search should make as much progress at level **k** before transitioning to level **k - 1**, because doing so will result in an average running time that's sub-linear.

```
static bool skipListContains(Vector<skipListNode *>& heads, int key);
```

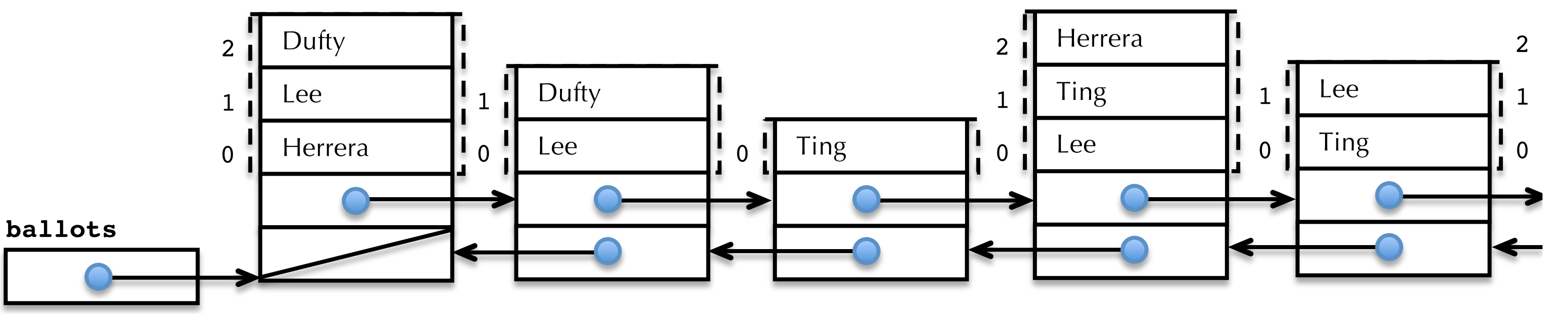
Discussion Problem 3: Ranked Choice Voting

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:

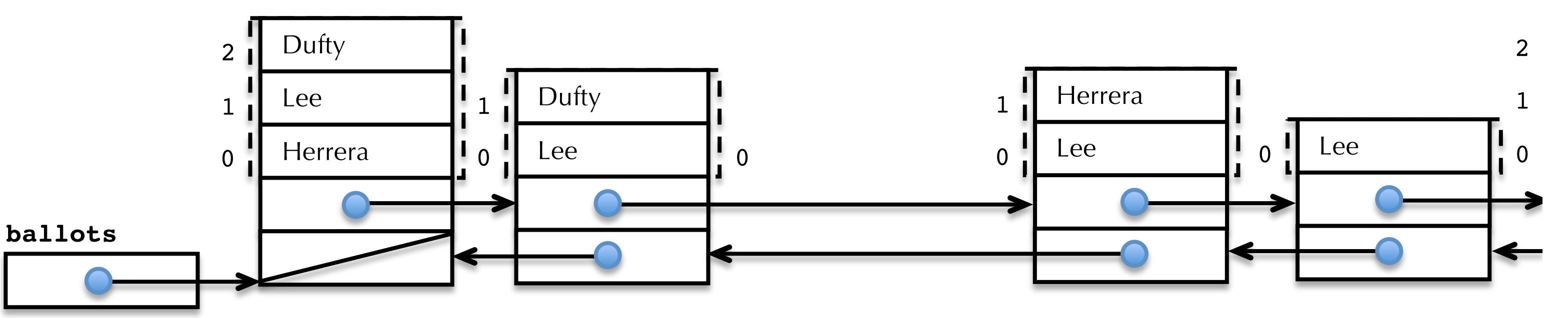
```
struct ballot {
   Vector<string> votes; // of size 1, 2, or 3; sorted by preference
   ballot *next;
   ballot *prev;
};
```

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.

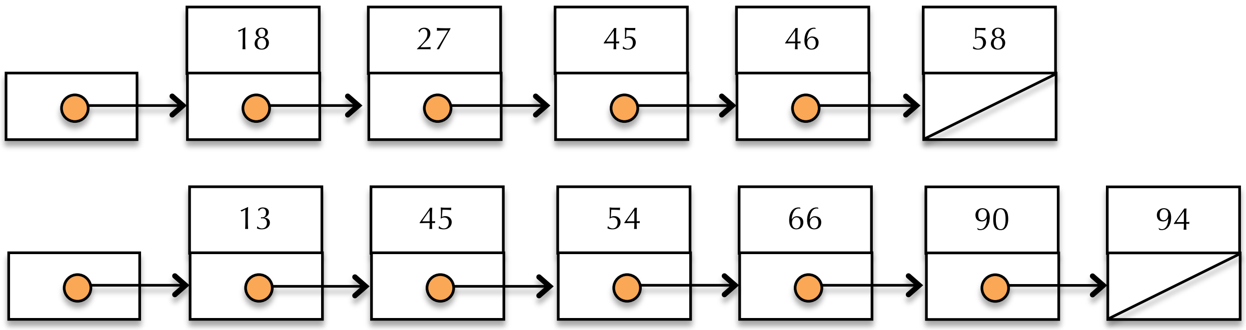
```
static string identifyLeastPopular(ballot *ballots);
```

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.

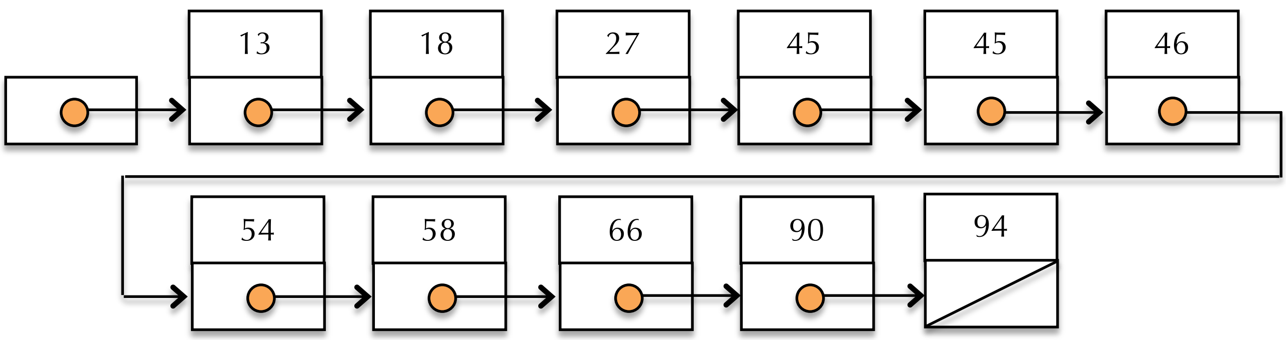
```
static void eliminateLeastPopular(ballot *& ballots, const string& name);
```

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);
```