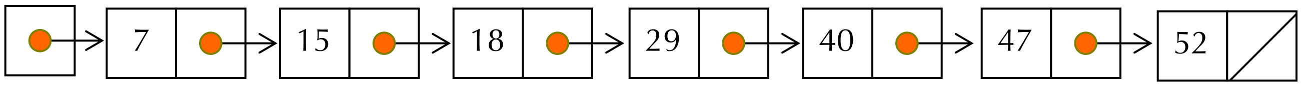
CS106X
Winter 2019

Handout 26
February 11th, 2019

# Section Handout

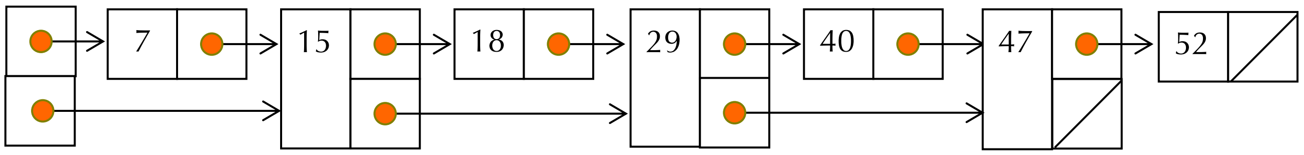
**Problem 1: Searching Skip Lists**

Imagine the sorted, singly linked list drawn below:



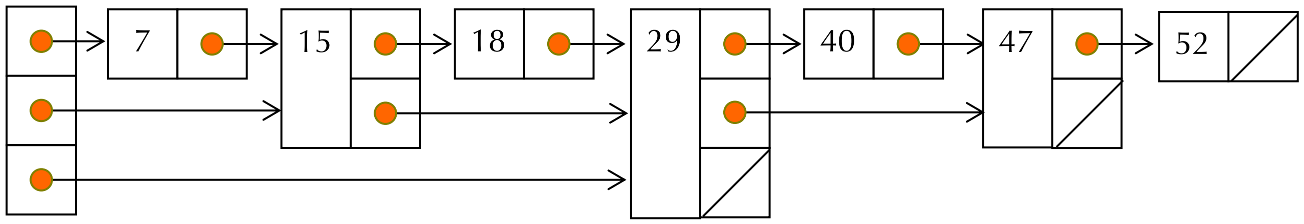
In spite of its being sorted, it still takes linear time to find the 52, or to confirm that something like 63 isn't present. We've discussed this very point in lecture: linked lists—even sorted ones—don't provide random access to arbitrary elements in the sequence, so we don't have anything close to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



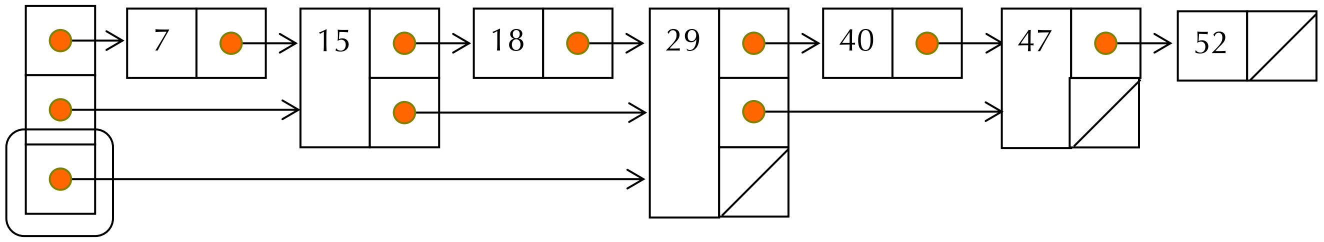
Initially, you traverse through the second level of links while you can, and then if need be, move to the original level of pointers and continue. So, a search for 47 would require we travel through three pointers. A search for 40 would require an examination of the same three pointers, except we'd detect 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:

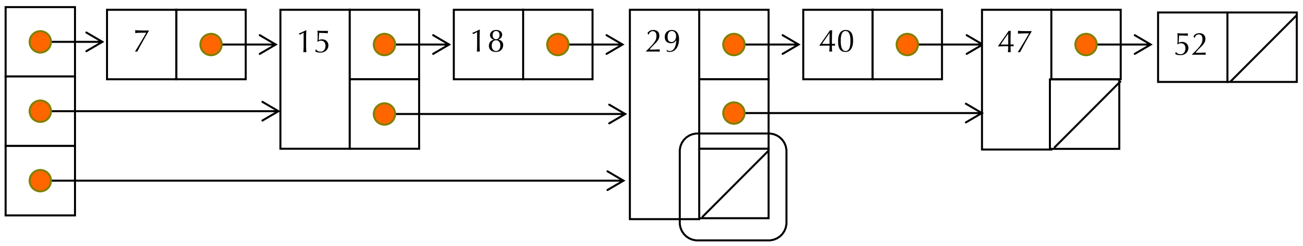


Search would start with the level-2 pointers, continuing with the level-1 pointer if the level-2 pointers read a dead end, and moving down to the level-0 pointers if the level-1 pointers dead end. Now we're working toward a data structure that, if extended to allow more and more levels of pointers, can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

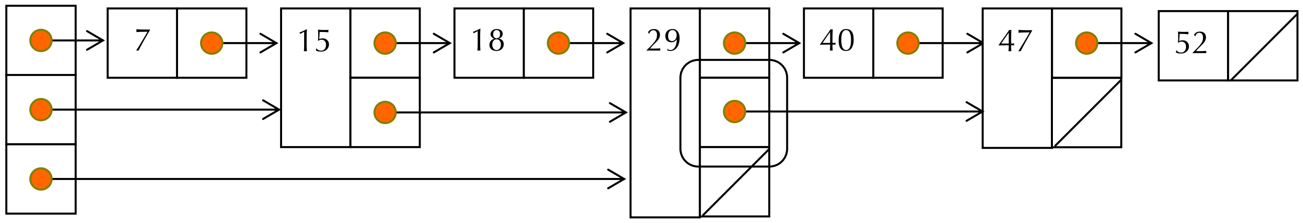
Want to know how we can search for the 40? Here's a short play-by-play identifying the node you visit and the pointers you dereference in order to hone in on that 40:



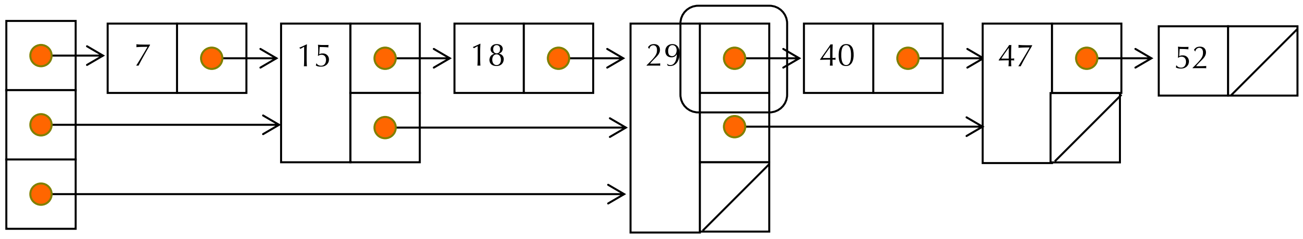
*1.) Start with the last of the leading pointers, which identifies a node with a 29 inside of it. 29 is less than the 40 we're looking for, so advance one node at level 2.*



*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 kth pointer in the vector hop at least as far as the k - 1th 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(const Vector<skipListNode *>& heads, int key);
```

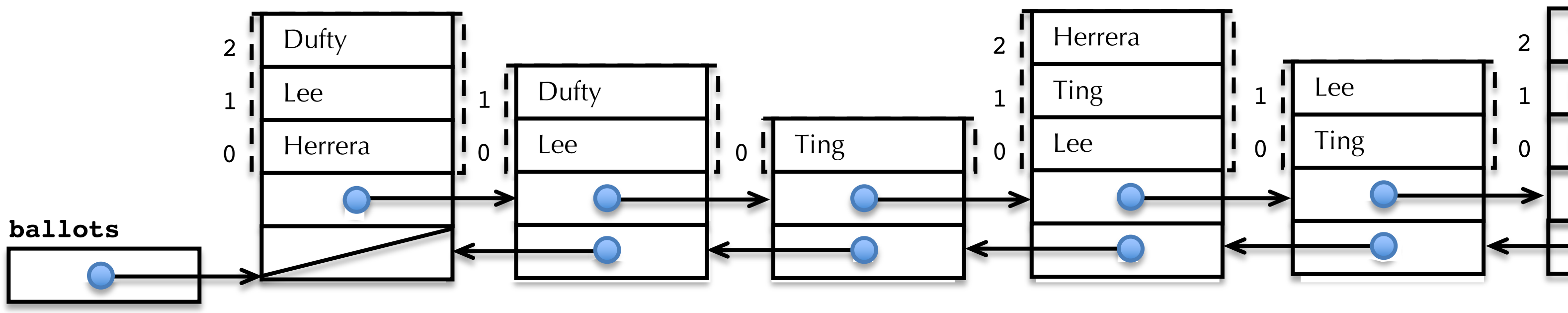
## Problem 2: Ranked Choice Voting

Ranked choice voting—also known 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:

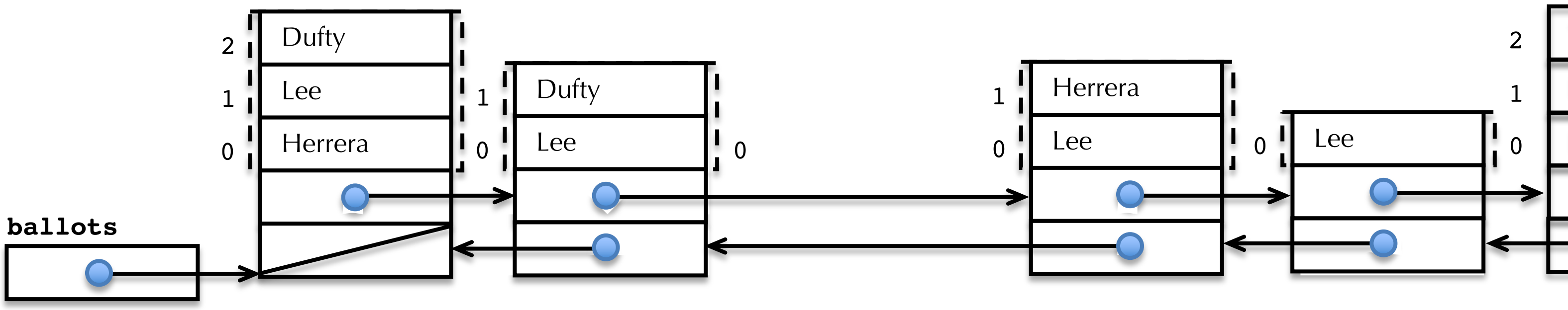
```
struct ballot {
   Vector<string> votes; // of size 1, 2, or 3; sorted by preference
   ballot *next;
   ballot *prev;
};
```

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.

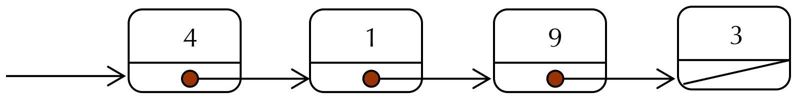
```
static string identifyLeastPopular(ballot *ballots);
```

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);
```

**Problem 3: Binary Tree Synthesis**

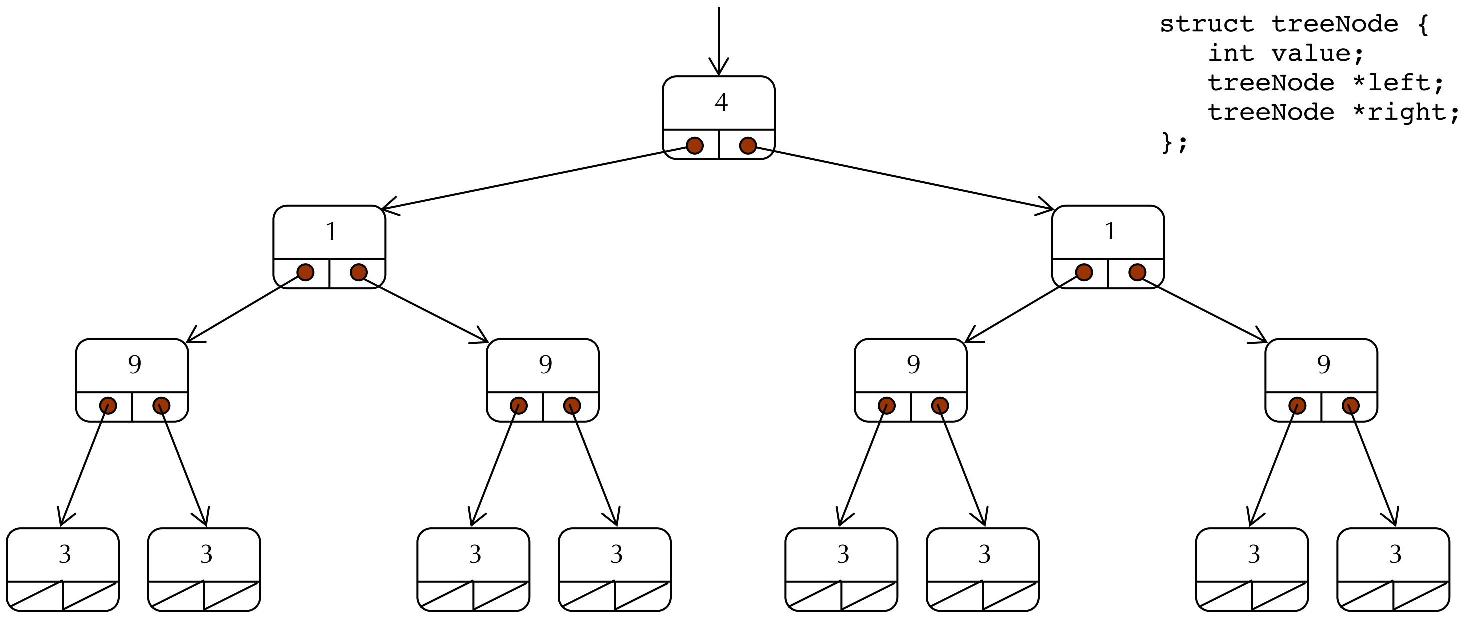
**`listToBinaryTree`** takes a singly linked list of numbers and constructs an independent binary tree where the nth item of the singly linked list occupies all positions at the nth level of the binary tree structure. So, if given the address of the list's front node



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

you would synthesize the following tree and return the address of its root:

```
struct treeNode {
   int value;
   treeNode *left;
   treeNode *right;
};
```



Had there been additional elements beyond the 3, then all of the 3s in the tree would have had two non-**`NULL`** children, and so forth. The product of the function you'll write will be a complete, balanced, binary tree (though not binary search) where every path from the root to a leaf sees the same sequence of numbers as seen in the original list.

For the discussion problem, you're to write two versions of this **`listToBinaryTree`** function: one recursive function, and one iterative one. The recursive function is very short and very clean and gives birth to the tree in a depth-first manner. The second version is iterative, uses a single **`Queue< treeNode **>`** as an auxiliary data structure, and builds up the tree of interest in a breadth-first manner. That means that the node housing the 4 is allocated first, then the two nodes housing the 1s are allocated, initialized and planted as children of the 4 node, and then all four of the nodes holding 9s are placed, and then the eight nodes surrounding 3s would be placed.

There are advantages to both versions, so it's instructive to be familiar with each.

a. Implement **`listToBinaryTree`** recursively. Do not destroy or otherwise modify the original list. Just use it as a read-only sequence of numbers used to recursively synthesize the corresponding binary tree.

```
static treeNode *listToBinaryTree(const listNode *head);
```

b. Now implement the iterative version, which should make use of a single **`Queue<treeNode **>`**. The **`Queue<treeNode **>`** is used to line up the locations of the **`treeNode *`**s that need to be considered during the next iteration. You have this and the next page for your solution. (Hint: your **`Queue`** template has a **`size`** method that comes in handy.)

```
static treeNode *listToBinaryTree(const listNode *head);
```