

## 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:

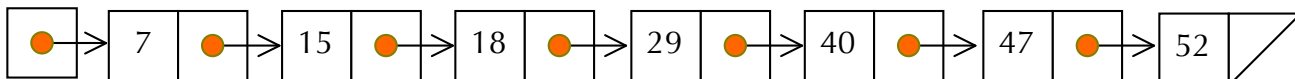
<b>list</b>	<b>list after call braid(list)</b>
1 → 4 → 2	1 → 2 → 4 → 4 → 2 → 1
3	3 → 3
1 → 3 → 6 → 10 → 15	1 → 15 → 3 → 10 → 6 → 6 → 10 → 3 → 15 → 1

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

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

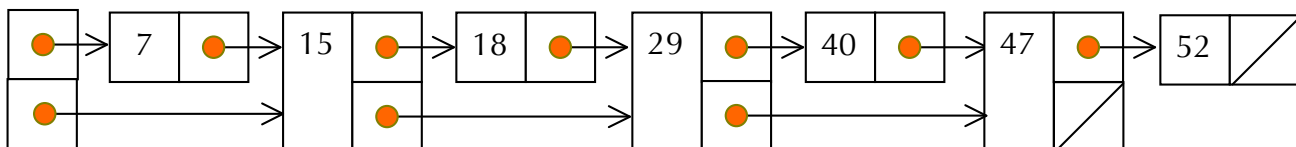
### Discussion Problem 2: Searching Skip Lists

Imagine the sorted, singly linked list drawn below:



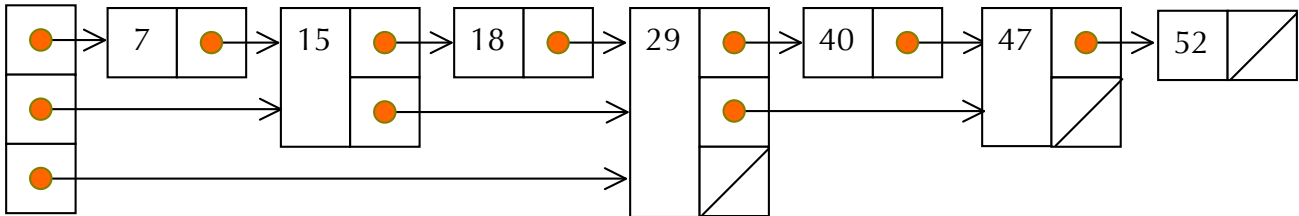
In spite of its being sorted, it still takes linear time to find the 52, or to confirm that something like 63 isn't present. We've discussed this very point in lecture: linked lists—even sorted ones—don't provide random access to arbitrary elements in the sequence, so we don't have anything close to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



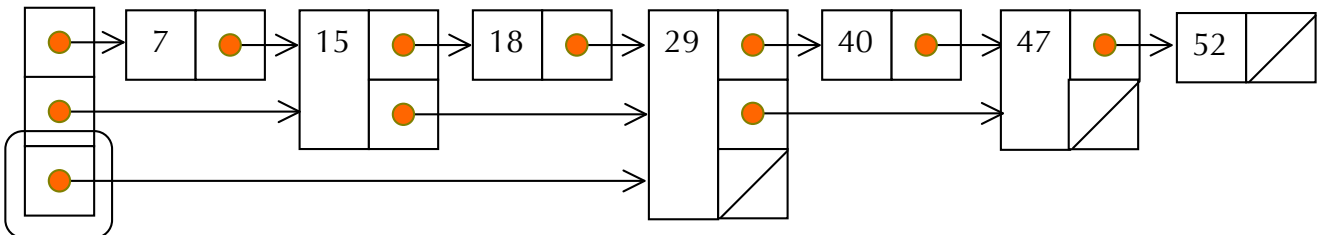
Initially, you traverse through the second level of links while you can, and then if need be, move to the original level of pointers and continue. So, a search for 47 would require we travel through three pointers. A search for 40 would require an examination of the same three pointers, except we'd detect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

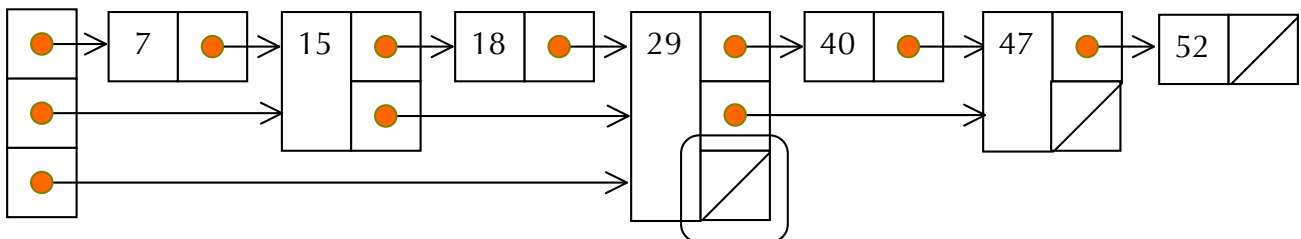


Search would start with the level-2 pointers, continuing with the level-1 pointer if the level-2 pointers dead end, and moving down to the level-0 pointers if the level-1 pointers dead end. Now we're working toward a data structure that, if extended to allow more and more levels of pointers, can support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

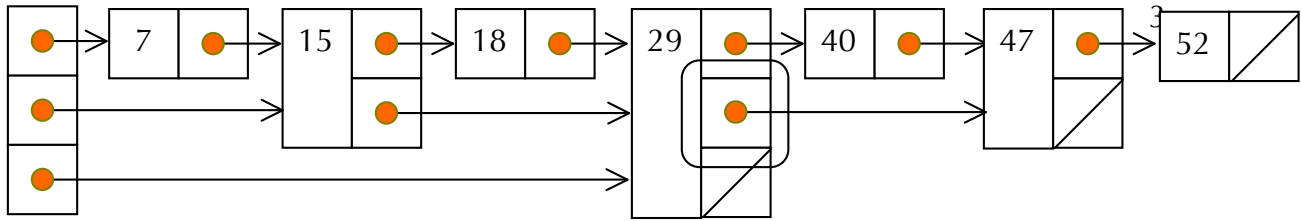
Want to know how we can search for the 40? Here's a short film identifying the node you visit and the pointers you dereference in order to hone in on that 40.



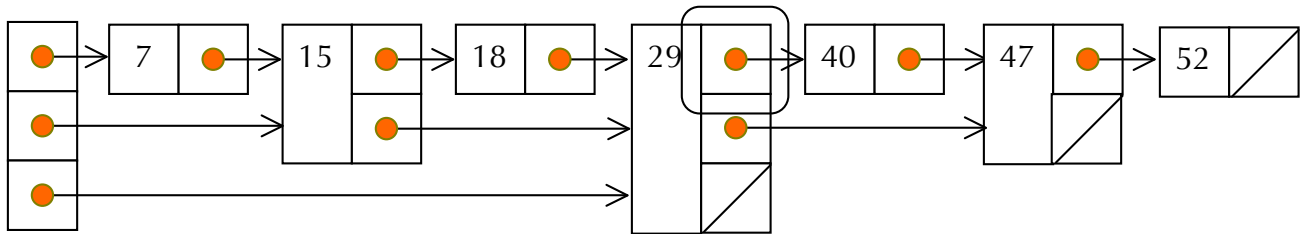
1.) Start with the last of the leading pointers, which identifies a node with a 29 inside of it. 29 is less than the 40 we're looking for, so advance one node at level 2.



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  $k^{\text{th}}$  pointer in the vector hop at least as far as the  $k - 1^{\text{th}}$  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.

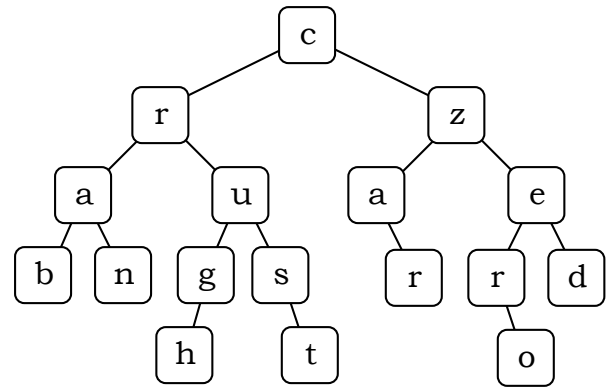
```
bool skipListContains(Vector<skipListNode *>& heads, int key);
```

## Lab Problem 1: Finding Words In Character Trees

You're given the following node definition for a general binary tree<sup>i</sup> of characters.

```
struct node {
    char ch;
    node *left, *right;
};
```

You're interested in writing a function that determines whether or not a particular word can be found along any path from the root to the fringe. So, given the binary tree on the right, your function should be able to confirm that presence of words like **"crab"**, **"ran"**, **"rug"**, **"crust"**, **"rust"**, **"us"**, **"ugh"**, **"czar"**, **"zero"**, and **"zed"**. **"rug"** and **"us"** are noteworthy, because they neither start at the root nor end along the fringe—they're completely internal. (**"arc"** doesn't count, because it's going the wrong way.)



Write a function called **wordExists**, which when given the root of a character tree and a string confirms whether or not the string can be found along some downward path from the root to any single leaf. In particular, it returns **true** when the word is present and **false** otherwise. You should return as soon as an answer can be determined, which means you should use recursive backtracking to prune the search.

```
bool wordExists(node *tree, string str);
```

Update the **character-tree.cpp** file with your implementation of the **wordExists** function. (You should ignore the **character-tree-test.cpp** file unless you're stumped and want to understand why things aren't working. And don't worry about freeing these trees either.)

## Lab Problem 2: Implementing the SkipSet [optional, and challenging, but good to know]

Let's revisit the skip list, which is quite arguably one of the cooler data structures invented in the past 20 or so years.. Hopefully, after working through the **skipListContains** as a group on the board, you understand why skip lists enable a fairly convincing imitation of binary search.

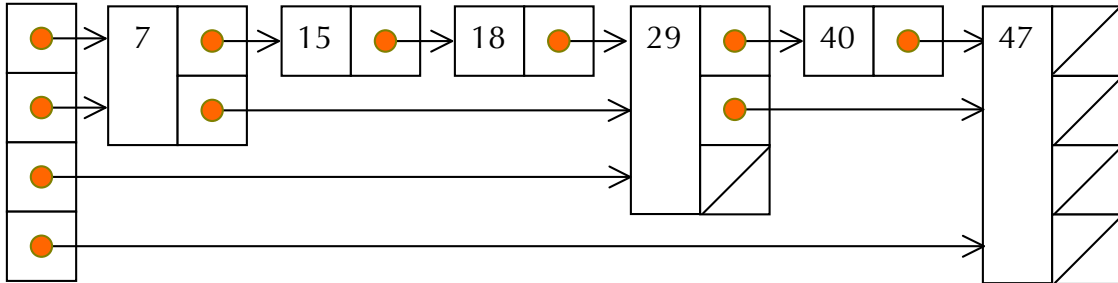
Confession: In practice, it's impossible to maintain such a regular, ruler-like structure where every other element has one level of pointers, every other intervening element has two levels of pointers, etc, if we're to freely insert and remove numbers. We shouldn't even pretend to try if insertion is to be fast.

---

<sup>i</sup> We haven't formally covered trees just yet, but just think of a tree—at least in this problem—as a linked list where there are always two forks in the road.

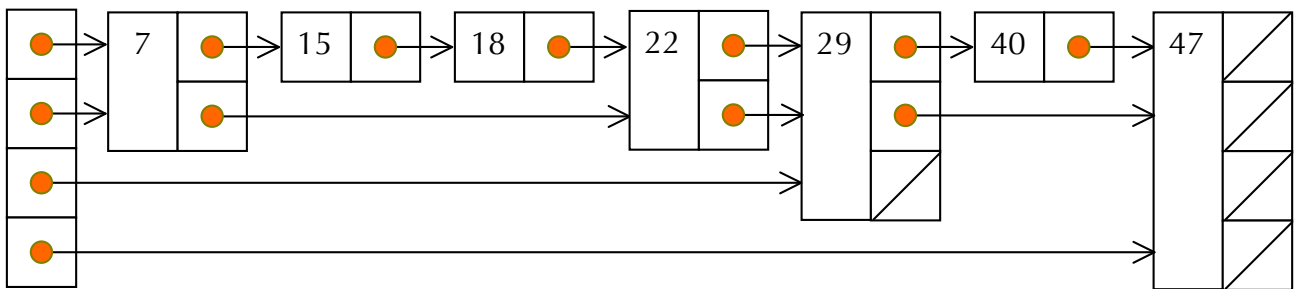
Given an arbitrary skip list where most are level 0, some but fewer are level 1, even fewer are level 3, and so forth, we can insert an arbitrary element by flipping a coin repeatedly until we flip a head. The number of simulated coin flips needed dictates how many levels of pointers will travel with the number being inserted. If the simulated coin is fair, we expect the node around the inserted number to have just one pointer about 50% of the time, two pointers 25% of the time, etc. In practice, the coin is biased ( $p = 0.25$  is common) so that multiple levels are even less common.

Consider the following not-so-symmetric skip list:

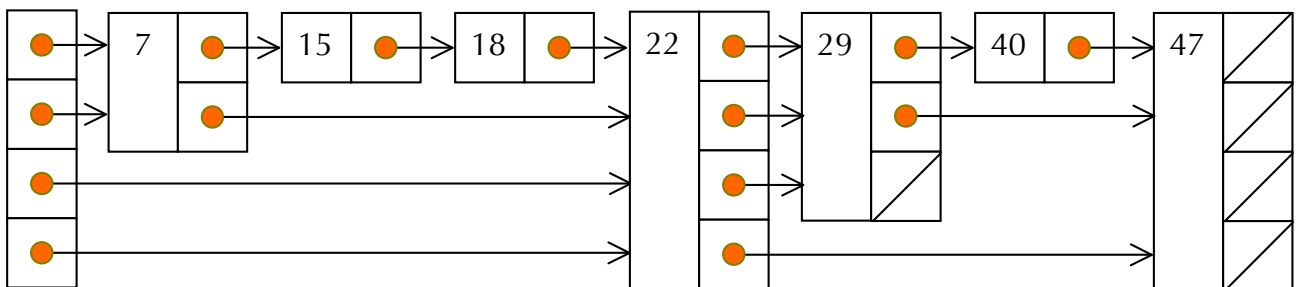


The six numbers could have been inserted in any order, and we really can't tell with any certainty what that order is. We do know this: at the time the 18 was inserted, we flipped some coin just once before getting a head. The same can be said about the insertion of the 15 and the 40. At the time the 7 was inserted, it took two flips. 29, three flips. 47, four flips.

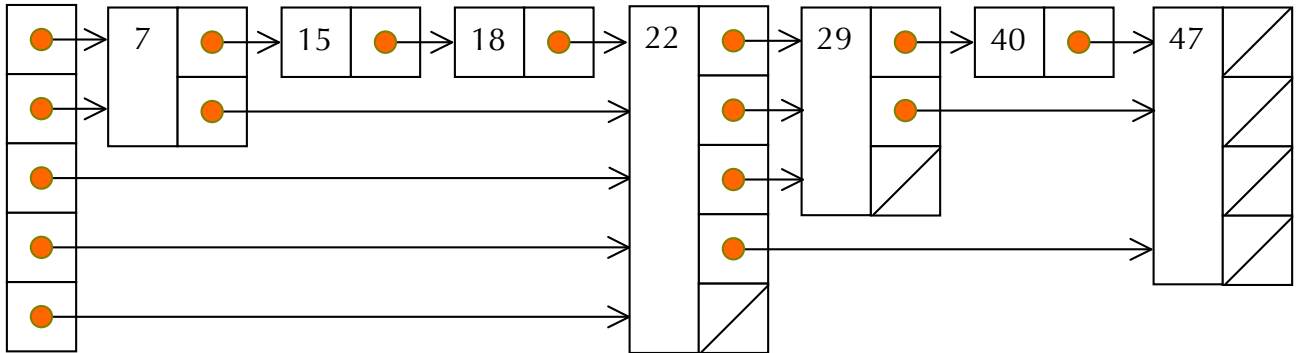
The height of the left spine—the array of leading pointers at each of the levels—needs to be as tall as the tallest node anywhere in the list. If we were to insert the number 22 at this point, and the luck of the flip said it should include level 0 and level 1 pointer, the picture would be updated as follows:



Had the number of coin flips been 4 instead of 2, the picture would look like this instead:



Had the number of coin flips been 5—thereby introducing a level of pointers not seen prior—the spine itself would need to be extended to stay in sync with the largest node height seen anywhere in the list.



In spite of the randomization, the generic algorithm for searching remains precisely the same as it was where we pretended there was a regular pattern of levels. By trusting that randomization produces level 0s most of the time, and higher levels much less often (and that the higher levels are sprinkled somewhat uniformly throughout the list), we can trust that, in most cases, the running time for search is still  $O(\lg n)$  on average.

One not uncommon use of the skip list is its use as the internal representation of a sorted set or a sorted map (and by sorted, I mean that iteration order is sorted order.) By leveraging the above descriptions of search and insertion, one can implement a set—we'll call it the **SkipSet** so it doesn't clash with our preexisting **Set** class—in terms of a skip list.

Here is the **skip-set.h** interface for the **SkipSet** class:

```
class SkipSet {
public:
    SkipSet();
    ~SkipSet();

    int size() { return numElems; }
    bool isEmpty() { return size() == 0; }
    bool contains(int value);
    void add(int value); // adds only if not already present

private:
    struct node {
        int value;
        Vector<node *> links;
    };

    int numElems;
    Vector<node *> spine;
};
```

Notice it's not a template, but rather an integer-specific set. To simplify things, I've gone ahead and defined the inner node class and the private data section for you, just to make sure you have enough to work with and don't feel the need to fuss over how to represent a skip list.

I've written a test harness (**skip-set-test.cpp**) to aggressively exercise the **SkipSet** you'll be implementing. You're concerned only with the implementation of the constructor, the destructor, **contains**, and **add**. I would implement **contains** before **add** (in spite of the fact that you can't test it until **add** is written), because **contains** is conceptually a subset of the **add** operation.