CS106X
Winter 2017

Chris Gregg

Section Handout #5

This week has practice with pointers, dynamic allocation, and Linked Lists.

1. A Series of Unfortunate References
What is the output of the following code snippet?

```
void VFD(int *duncan, int isadora, int& quigley) {
  isadora += *duncan;
  quigley *= isadora;
  (*duncan) = 1;
  isadora = *duncan;
}

int main() {
  int sunny = -6;
  int klaus = 21;
  int violet = 2;
  VFD(&sunny, violet, klaus);
  cout << sunny << " " << klaus << " " << violet << " " << endl; return 0;
}
```

2. Section Leaders, Then and Now
Analyze the following program and draw the state of memory at the two points indicated. Be sure to differentiate between memory on the stack and memory on the heap, note values that haven't been initialized, and identify where memory has been orphaned (heap values that no longer have a pointer to them in scope).

```
struct sectionleader {
  int leslie;
  sectionleader *colin;
  int *jason[2];
};

void jeanluc() {
  sectionleader chris[2];
  sectionleader *wesley;
  wesley = &chris[1];
  chris[0].leslie = 152;
  chris[0].jason[0] = new int[2];
  chris[0].jason[1] = &(wesley->leslie);
  wesley->jason[0] = chris[0].jason[1];
  wesley->jason[1] = &(chris[0].jason[0][1]);
  *(wesley->jason[1]) = 9189;
  chris[1].colin = wesley->colin = wesley;

  // <-- 1) draw the state of memory just prior to the call to tyler

  tyler(chris[1], wesley->colin);
}
```

Thanks to Marty Stepp, Victoria Kirst, Jerry Cain, and other past CS106 instructors for contributing content on this handout. Thanks to Colin Kincaid and Anupama Rajan for proofreading.

```
void tyler(sectionleader &anupama, sectionleader *&aaron) {
  anupama.leslie = 465;
  aaron->colin = &anupama;
  aaron->leslie = 830;
  aaron->colin[0].jason[1] = &(anupama.colin->leslie);
  aaron = &anupama;
  aaron->leslie = 507;
  aaron->colin = new sectionleader[2];

  // <-- 2) draw the state of memory just before tyler returns
}

int main() {
  jeanluc();
}
```

For the remaining problems, assume the following structures been declared:

ListNode

```
struct ListNode {
  int data;
  ListNode *next;
}
```

StringNode

```
struct StringNode {
  string data;
  StringNode *next;
}
```

DoubleNode

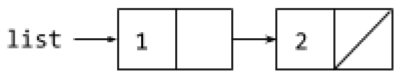
```
struct DoubleNode {
  double data;
  DoubleNode *next;
}
```

PointNode

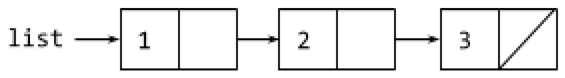
```
string PointNode {
  int x;
  int y;
  PointNode *next;
}
```

3. What's the Code Do?

For each of the following diagrams, draw a picture of what the given nodes would like like after the given line of code executes:



```
list->next = new ListNode;
list->next->data = 3;
```



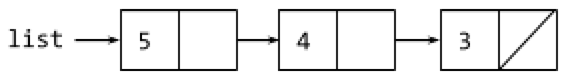
```
list->next->next = nullptr;
```

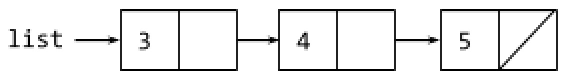
4. What's the Code?

For each of the following diagrams, write the code that will produce the given "after" result from the given "before" starting point by modifying the links between the nodes shown and/or creating new nodes as needed. There may be more than one way to write the code, but do **not** change the data field of any existing node. If a variable doesn't appear in the "after" picture, it doesn't matter what value it has after changes are made.

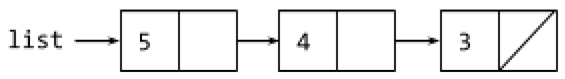
Before **After**

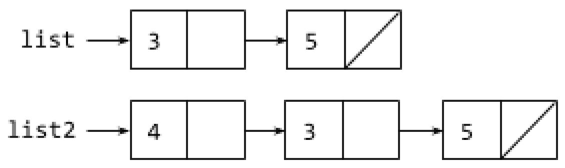
a)





b)





5. Is Sorted
Write a function that takes in a pointer to the front of a linked list of integers and returns whether or not the list that's pointed to is in sorted (nondecreasing) order. An empty list is considered to be sorted. Consider both recursive and non-recursive solutions.

6. Count Duplicate Strings
Write a function that takes in a pointer to the front of a sorted linked list of lowercase strings and returns the number of duplicates in the list. For example, suppose you are given the following list:

```
{"apple", "apple", "apple", "bat", "bat", "car", "car", "dog", "dog", "dog", "fox", "fox"}
```

Your function should return 7 (once for each of the underlined duplicates).

7. Remove All Threshold
Write a function that takes a pointer to the front of a linked list of doubles, as well as a target value and a threshold. It should remove all values in the list that are within the threshold of the target value. For example, suppose you are given the following list, a target of 3.0, and a threshold of 0.3:

```
{3.0, 9.0, 4.2, 2.1, 3.3, 2.3, 3.4, 4.0, 2.9, 2.7, 3.1, 18.2}
```

You should remove the underlined values, yielding the following list:

```
{9.0, 4.2, 2.1, 2.3, 3.4, 4.0, 18.2}
```

If the list is empty or values within the given range don't appear in the list, then the list should not be changed by your function. You should preserve the original order of the list.

8. Double List
Write a function that takes a pointer to the front of a linked list of integers and appends a copy of the original sequence to the end of the list. For example, suppose you're given the following list:

```
{1, 3, 2, 7}
```

After a call to your function, the list's contents would be:

```
{1, 3, 2, 7, 1, 3, 2, 7}
```

Do not use any auxiliary data structures to solve this problem. You should only construct one additional node for each element in the original list. Your function should run in O(n) time where n is the number of nodes in the list.

9. Split
Write a function that takes in a pointer to the front of a linked list of integers and rearranges the elements so that all the negative values appear before all the non-negatives, with each group in the same relative order as the original list. For example, suppose you are given the following list:

```
{8, 7, -4, 19, 0, 43, -8, -7, 2}
```

After a call to your function, the list's contents would be:

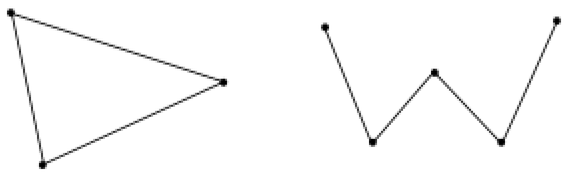
```
{-4, -8, -7, 8, 7, 19, 0, 43, 2}
```

You must solve this problem by rearranging the links of the list, not by editing data fields or creating new nodes. Do not use any auxiliary data structures to solve this problem.

10. Reverse Recurse
Write a recursive function that takes in a pointer to a list of integers and reverses the elements in the list. Return a pointer to the first element of the newly reversed list. Do not create new elements, edit data fields, or use any auxiliary data structures.

11. Merge
Write a function that given the pointers to two sorted lists of integers, and merges them into one sorted list. Return a pointer to the first element of the newly reversed list. Your implementation shouldn't allocate any new memory, but should use the nodes making up the two originals. Do not use any auxiliary data structures to solve this problem.

12. Draw Polygonal Path
Write a function that takes in a `GWindow` object and a pointer to a list of x/y coordinate pairs. The list represents a "polygonal path," or a series of connected line segments, and draws the path as a series of dots and lines on the window.



Each line should be 1 pixel thick and each dot should be 2 pixels wide. It might help to recall the `drawLine` and `drawOval` functions from the `GWindow` class:

```
gw.drawLine(x1, y1, x2, y2);
gw.drawOval(x, y, xr, yr);
```

Note that the polygonal path can be closed (as on the left above) or open (as on the right). In the case of an open path, the last node in the path will have a next value of nullptr. In the case of a closed polygonal path, the last node will link back to the first node in the path. Your function should also be able to draw a single point. In the case of a null list, you should draw nothing.

13. Braiding Lists
Write a function that takes in the pointer to a linked list of integers and modifies the list so that it alternates elements between the original list and the reverse of the original list. Here are a few examples:

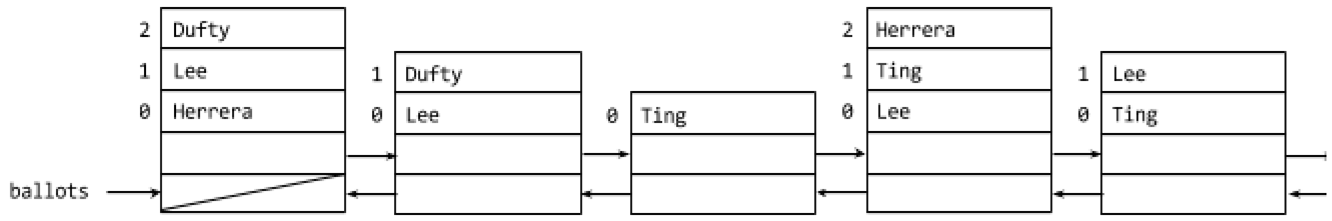
Before	After
`{1, 4, 2}`	`{1, 2, 4, 4, 2, 1}`
`{3}`	`{3, 3}`
`{1, 3, 6, 10, 15}`	`{1, 15, 3, 10, 6, 6, 10, 3, 15, 1}`

14. Instant Runoffs
In the city of San Francisco, they used ranked choice (or instant runoff) voting for mayoral elections. Instead of voting for a single candidate, those casting ballots vote for up to three candidates, ranking them 1, 2, and 3. Initially, only first rank votes matter – if a single candidate gets a majority (> 50%) of the first rank votes, then that candidate wins. If nobody has the a majority, the candidate with the least number of first rank votes is eliminated by removing that candidate from all ballots (or "exhausting" those votes), and promoting the other choices on those ballots to close any gaps. This process is repeated over and over again until one candidate has a majority of rank-one votes.

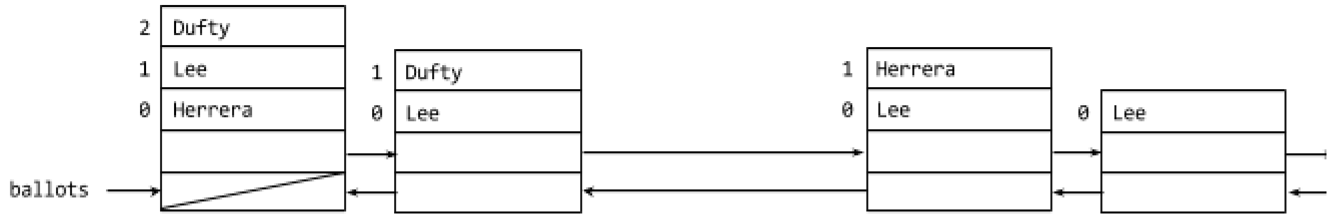
We're going to implement instant runoffs using a *doubly-linked list*. Remember that a doubly-linked list is similar to a linked list, but each node has a pointer to the previous node in the list as well as the next. For the purposes of this problem, assume we have the following struct:

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

Assume you have :



If nobody wins the majority of first-rank votes, you remove the last place candidate. Let's suppose after analyzing all of the ballots (including those not pictured), we determine Ting received the smallest number of first place votes. The ballots would be updated to look like this:



Note that when all of the choices are eliminated from a ballot, the ballot is removed from the list.

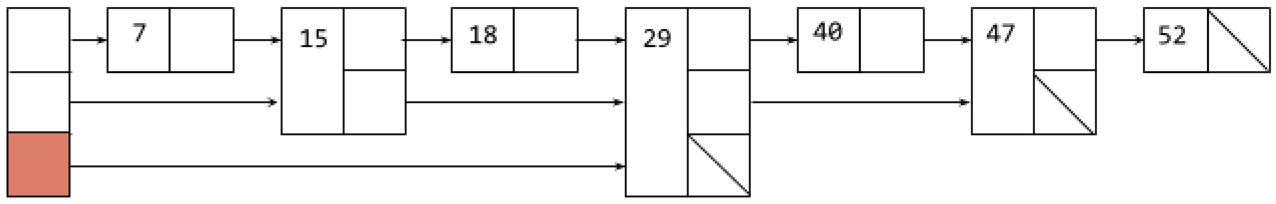
First, write a function that, given a pointer to a list of ballots, returns the candidate receiving the smallest number of first-choice votes. You can assume that all ballots include at least one vote, that no ballots ever include two votes for the same candidate, that each candidate received at least one first-rank vote, and if two or more candidates are tied for least popular, then any one of them can be returned.

Next, implement a 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 all ballots that are depleted of all votes.

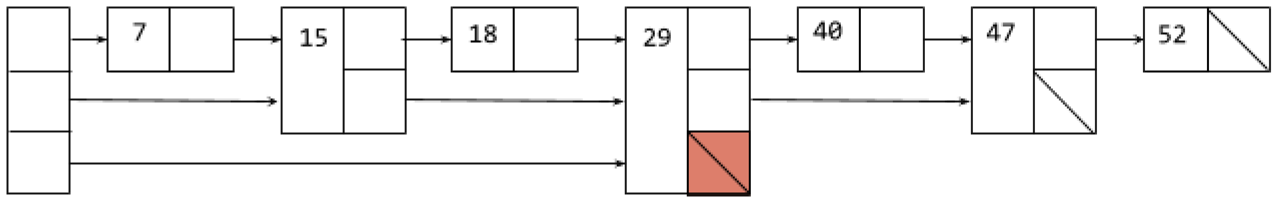
15. Skip a Dee Doo Dah

So far we've learned about singly- and doubly-linked lists, but both of them have similar performance failings – even with a sorted list, you can't find an element in the list in better than O(n) time. There are more complex list structures that allow you to find elements faster. One of those is called a skip list.

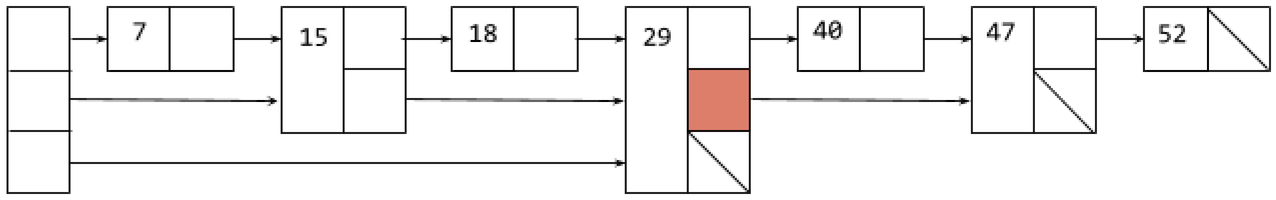
A skip list is a sorted linked list, based around the idea of being able to skip over nodes when searching. Instead of keeping a single pointer to the next element in the list, each node keeps can keep up to *n* pointers, where the *i*th pointer points to the element *i* elements ahead in the list. Here's an example of how a skip list might improve search performance.



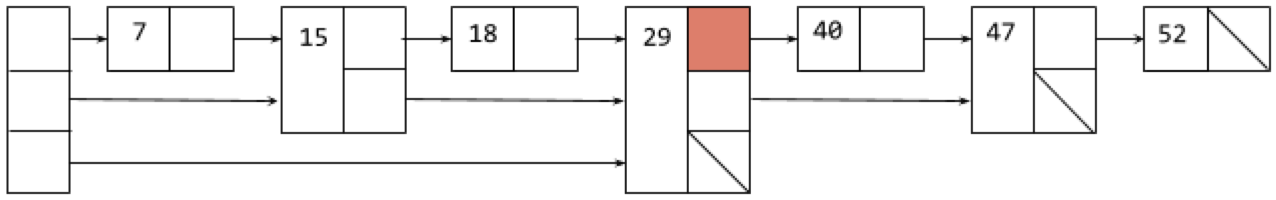
Let's say we're looking for the element 40 in this list. We start with the last of the leading pointers, which points furthest into the list. We see that the node it points to contains a 29, which is less than what we're looking for, so we continue on the level-2 pointer.



We see the level-2 pointer is null, so we move down a level, to the level-1 pointer.



We look to see where the level-1 pointer is pointing, and see that the value is 47, greater than the value we're looking for, so we move down one more level, to the level-0 pointer.



Finally, we see that the node pointed to by the level-0 pointer is the value we're searching for.

Assume you're given the following struct:

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

Write a function that, gvien a Vector of pointers to SkipListNodes that represents the heads of a skip list and a integer value, returns whether or not the value specified is contained in the skip list. Remember that a given node may have an arbitrary numbers to other nodes, and that you must make as much progress as possible at level k before moving down to level k - 1.