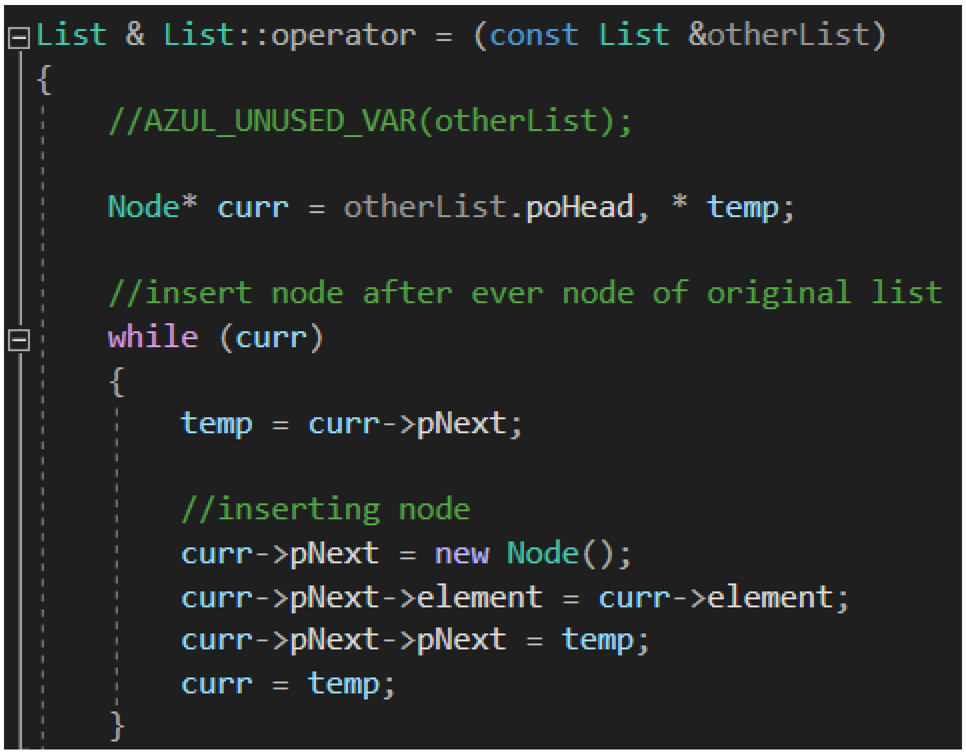
Parallelization

Joey Domino

Last paper! Ok so we are discussing parallelization complexity in Big O notation. I have built out a system that will take a list of given nodes that have a piece of data (string in this case) and a random and next pointer to track the nodes in the list and copy them into a new list via a deep copy. Parallelization comes in when we discuss how to best copy the data to the new list.

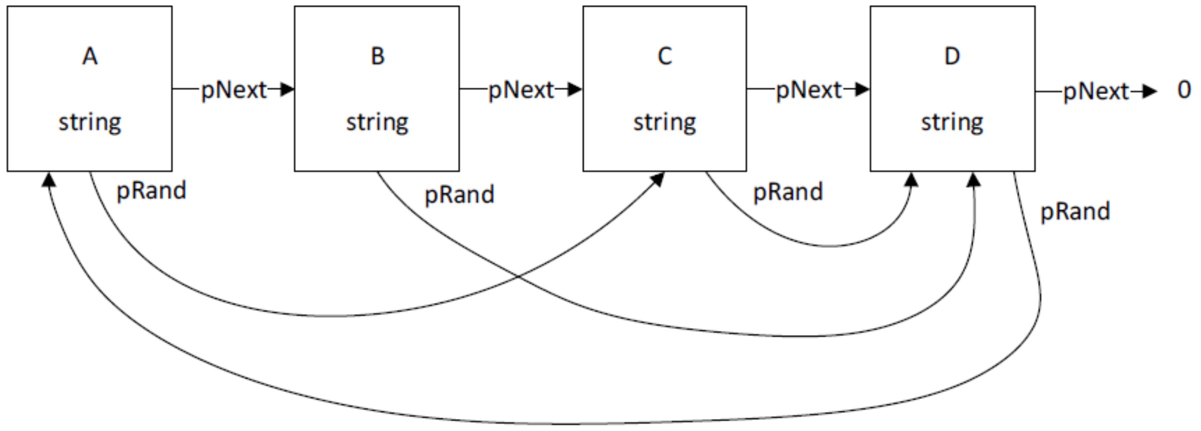
Normally when we deep copy items, there tends to be a temporary variable of the same type that is used to transfer the item to the new list of nodes, but the problem with that is it takes extra auxiliary space to complete. In other words: it takes up more space on the stack to clone the items than is actually necessary.

To get around this, we can create a new list of nodes in the existing list beside the node we are copying. Then copy the contents and next pointer to the new node and move on. After replicating the list, we must go through it once more to set the random pointers appropriately. We can't set the random pointers on the first pass because the nodes that random pointers point to don't exist yet.

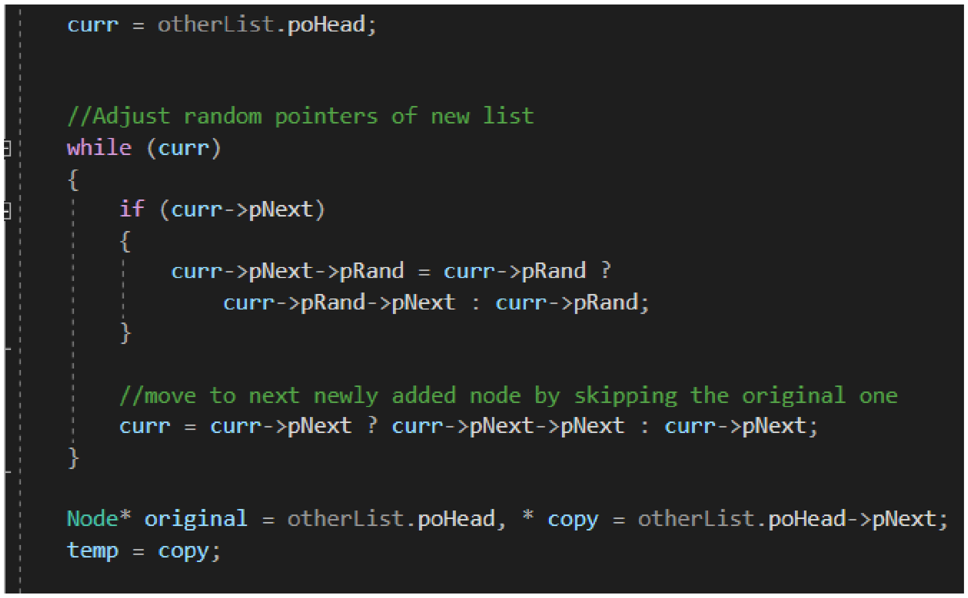


```
List & List::operator = (const List &otherList)
{
    //AZUL_UNUSED_VAR(otherList);

    Node* curr = otherList.poHead, * temp;

    //insert node after ever node of original list
    while (curr)
    {
        temp = curr->pNext;

        //inserting node
        curr->pNext = new Node();
        curr->pNext->element = curr->element;
        curr->pNext->pNext = temp;
        curr = temp;
    }
```

So we loop through the list one more time and set the copied node's random pointer to the original node's random pointer's next node. That's a lot of pointing in one sentence. It's easier to look at this diagram.

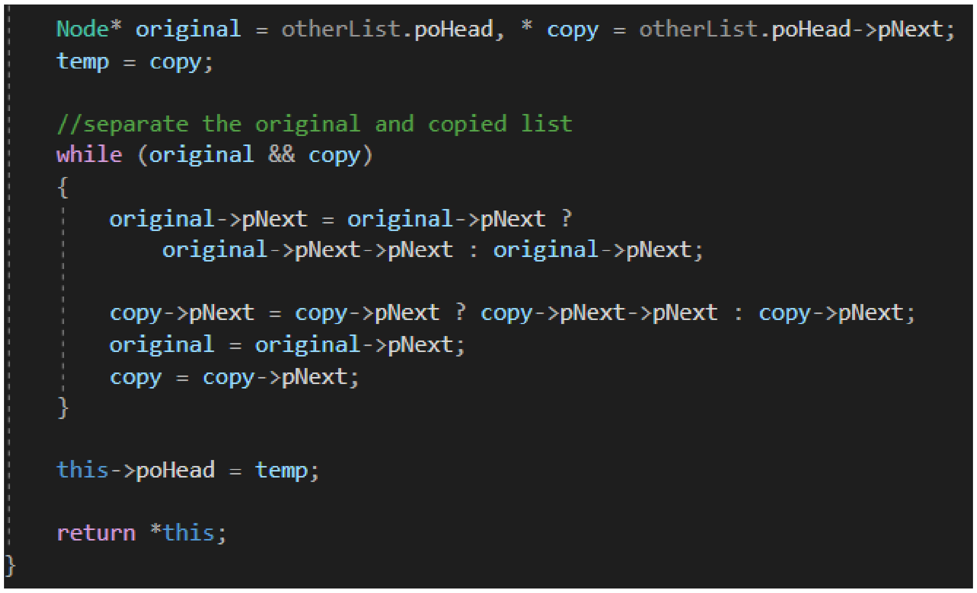


This is the existing list we are copying from. Imagine that we duplicate every node and place them beside their original version. We set the pointers so that the list consists of every node, then step through one more time to tell the new nodes "look at your creator node and copy that node's random->pNext location and set it to your pRand value.



```
curr = otherList.poHead;


//Adjust random pointers of new list
while (curr)
{
    if (curr->pNext)
    {
        curr->pNext->pRand = curr->pRand ?
            curr->pRand->pNext : curr->pRand;
    }

    //move to next newly added node by skipping the original one
    curr = curr->pNext ? curr->pNext->pNext : curr->pNext;
}

Node* original = otherList.poHead, * copy = otherList.poHead->pNext;
temp = copy;
```

If we reference our diagram above, this would mean that A's copy, let's call it A2, is now A's new pNext pointer. A2 now points to B. the system tells A2 to set its pRand pointer by looking at A's pRand->pNext value. In this case, it would be C's->pNext, which is C2. Do this for every node until complete and we have a newly deep copied list of nodes ready to go. There's just one problem: they still exist in the same list!



```
    Node* original = otherList.poHead, * copy = otherList.poHead->pNext;
    temp = copy;

    //separate the original and copied list
    while (original && copy)
    {
        original->pNext = original->pNext ?
            original->pNext->pNext : original->pNext;

        copy->pNext = copy->pNext ? copy->pNext->pNext : copy->pNext;
        original = original->pNext;
        copy = copy->pNext;
    }

    this->poHead = temp;

    return *this;
}
```

The next and final step of the process is to take every copy node and split it from the original list and into a new one. This requires one more loop through the list. This time, we update the next pointers and split the nodes into two lists: original and copy. Once this is done, we set the poHead of List to the top of the copied list and return the newly formed List.

Sounds complicated because it is. I'm far more used to creating a temp list that duplicates values before placing them. Doing things this way is faster, sure, but it can tend to be harder to read. As for speed, though, our time complexity reaches O(n) and auxiliary Space of O(1).

O(n) time complexity simply means we are looping through the list n number of times to complete the task. In this case, it's really O(3n), but constants aren't necessary when referring to Big O notation.

O(1) auxiliary space is incredible for this type of operation in my opinion. Auxiliary space is the extra or temporary space used by an algorithm to complete its task. Being O(1) means we did not use any extra or temporary space to complete this task. That is huge when dealing with lists of any type, because they can be very large. You may think that copying the nodes would increase our temporary space usage, but because we are using the copied list outside the method, the copied list is actually considered required. Because it's required, we don't consider the new list as part of the calculation. If we instead copied the original list into a buffer and then passed that buffer to a new list, then we'd increase our auxiliary space.

This is a very interesting way to copy a list. I'll have to remember this for future uses.