Recursive Operations, Reading

There are two ways to write loops: (1) iterative, like the for-loops we're used to using, and (2) recursive. Your prerequisite preparation for this course should have exposed you to both, although your working knowledge of recursive loops may not be as strong as that of iterative loops.

Iterative loops have one set of local variables that come and go with each cycle -- the previous cycles' values for these variables are completely forgotten and only the present matters. The cycles are executed **in series**. But in recursive loops, the cycles all coexist at the same time, each with their own set of local variables. The cycles are executed **in parallel**. There's no good reason to write recursive loops unless you need their ability to run simultaneously in parallel. "Simultaneously", not "independently" -- it's not like they can go off and run separately in their own threads...

Recursive loops can be implemented in two different ways: (1) functions that call themselves, with which you are probably already familiar, and (2) iterative loops with data structures (stacks, queues, etc) to store the cycles' states while they await their turn to be processed. *(Actually we used that in the form of a "to-do" list in the quicksort algorithm in an earlier module.)* Both *do* the same thing -- there are just two very different ways to *program* it.

We wouldn't be mentioning all this now if we weren't going to be applying it real soon. The rest of the reading uses the first kind of recursive loop -- functions that call themselves.

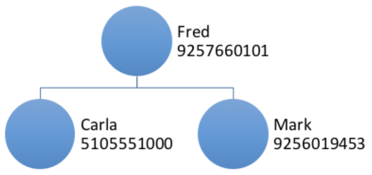
The keys Getter

Associative arrays usually have some function that can tell the main program what are the keys stored in it. In our AssociativeArray and HashTable we introduced a public function with this prototype:

```
queue<K> keys() const;
```

What makes this more involved than the search functions in the previous reading is that **keys** has to search out *all* keys in the tree simultaneously -- not just one. Each time it finds a node, it will have to save what it knows about that node while another cycle follows down is left-side decedents. When those cycles complete, it has to do the same with its right-side decedents before it can complete its cycle's work. The logic for a single cycle (to be coded into a recursive function) is like this:

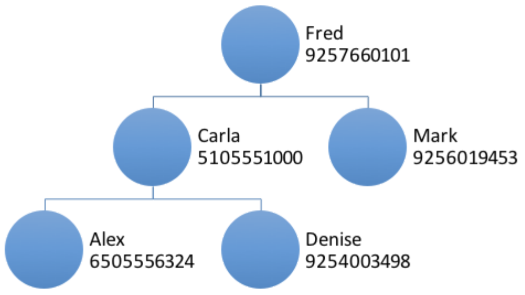
```
create a queue
p = rootNode
push p->left's key
push p's key
push p->right's key
```



The order matters, if we want the keys to be listed in sorted order -- it's left, middle, right.

But what if the left child has children? In that case, there's bit more to the **push p->left's key** step. We really should push its left, middle, and right, and not just its middle:

```
create a queue
p = rootNode
"push p->left's key"
{
  p₂ = p->left
  push p₂->left's key
  push p₂'s key
  push p₂->right's key
}
push p's key
push p->right's key
```



See how "push p->left's key" got replaced with a code block that looks just like the original code but with a different "p"? And see how the original "p" is set aside while the new code block with its own "p_2" runs? Also note a more subtle feature of this logic -- there is just *one* queue. Each cycle does *not* have it's own queue. They all share the one. For this to happen, the queue either has to be global (and we surely want to avoid global variables) or shared by reference as a parameter.

With all that, our function looks like this (without the template syntax):

```
void keys(const Node* p, queue<K>& q) const
{
  if (!p) return; // detect when the bottom of the tree is reached
  keys(p->left, q);
  q.push(p->key);
  keys(p->right, q);
}
```

The only problem is that for this to be called from the main program, the main program will have to (1) declare and share the queue, and (2) have access to the **struct Node** and the rootNode pointer. *None of that is going to happen*. In the main program we expect to just do something like this:

```
  queue<string> names = phoneBook.keys();
```

The solution is to create a second, public member function called "keys", create and return the queue from it, and start the recursive loop (since *it* has access to the root node). Something like this:

```
queue<K> keys() const
{
  queue<K> result;
  keys(rootNode, result); // calls the other version of the function
  return result;
}
```

Since the main program cannot use the recursive function, that should be private, like this:

```
template <typename K, typename V>
class BinarySearchTree
{
  ...

  public:
  ...
  queue<K> keys() const;

  private:
  void keys(const Node*, queue<K>&) const;
};
```

This is a very common structure for recursive functions, because the parameters needed in the recursive loop are often not available to the main program. So a simpler public function is supplied as an entry point to the process, and a private function handles the loop.

The clear Setter

The clear function is similar to keys, because it has to search out every node and do something with it. But that something is to deallocate the node instead of retrieve its key. So starting with the same algorithm used above, modified a bit, we get:

```
  p = rootNode
  deallocate p->left's key
  deallocate p->right's key
  deallocate p's key
```

Note how the order changed -- we cannot deallocate the middle before we deal with both the left *and* right decedents. You'll see this in recursive implementations of binary tree features, with deliberate order. "keys" was left, middle, right. "clear" is left, right, middle. More on these options in a later reading in this module.

Adapting the "keys" private function code (with no queue to pass around):

```
void clear(Node* p)
{
  if (!p) return; // detect when the bottom of the tree is reached
  clear(p->left);
  clear(p->right);
  delete p;
}
```

Of course, the main program cannot call this because of the pointer in the parameter list, so the public function launches the loop:

```
void clear()
{
  clear(rootNode);
  rootNode = 0;
  siz = 0;
}
```

Note the cleanup steps -- zeroing out the root and the size.