As we have described in class, writing multithreaded workloads can improve application performance by utilizing *thread-level parallelism*. As you saw in CS105 and as we have reiterated in class, the benefit of using multiple threads can be significant but it is unlikely that we can fully realize ideal speedups in practice. This is because of *Ahmdahl's Law*, which is fomalized below:

$$S = \frac{1}{(1 - \%p) + \frac{\%p}{I}} \tag{1}$$

Where S defines the overall speedup, $\%p$ describes the percentage of the application that is parallelizable, and I is the maximum ideal speedup.

It may be difficult to deduce what the percentage of an application is parallel! Unlike O or Θ, there is no good mechanism to define exactly how much of a program is inherently parallel or sequential relative to the overall execution. As such, we will empirically attempt to deduce the value p for various applications and, in the process, implement different strategies for writing a concurrent data structure.

Getting Started Start this lab by obtaining the source from the course website:

```
wget https://cs.pomona.edu/classes/cs181ca/labs/lab10.zip
unzip lab10.zip
```

In this directory, you will see that there are source files to implement a data structure called a *skip list*. Skip lists are often used in the literature due to their natural parallelizability while still promising $O(\log n)$ search, insertion, and removal operation. We will spend the next section overviewing skip lists before we dive into making them concurrent!

1 Skip Lists

A skip list is a probabilistic, sorted data structure in which data is stored in nodes in a collection of hierarchical linked lists. With a 50% probability, a node may be *promoted* to the next level of the skip list. As a result, each level up in the skip list has half as many nodes as the level below.

By arranging nodes in this way, searching for a node in a skip list works by starting from the minimum node and scanning along the uppermost linked list for the key. If the node is found, the search is done! Otherwise, track the node with the closest key less than the searched key and begin the search from *this node* at the level below.

To exemplify this procedure, an example skip list is shown in Fig. 1.

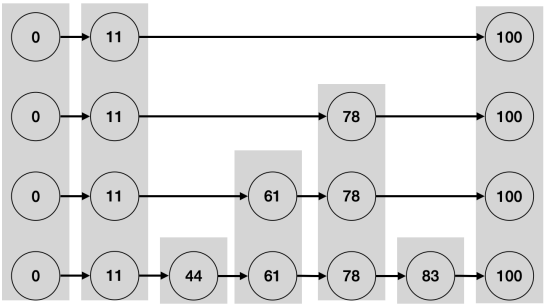

Figure 1: A skip list data structure.

Suppose we wanted to search for the element `61` in this skip list. To do so, we would start at the minimum element in the upper-most list and traverse from left to right looking for the element. In this case, this would mean that traversing the upper-most list would go from $0 \to 11 \to 100$. As soon as we reach a node greater than the key we are searching for (`100`) we are done searching at this level, so we go down to the next level and start the search from the node *right before* the biggest value at the level above. In this case, this means at the next level down, we begin our search from `11`. At the next level down we would search $11 \to 78$ and, again stop our search as we have reached a node greater than the key we are searching for. From here, we would continue our search from `11` at the next level down. Our search would go $11 \to 61$, and we have found our element! If we get to the lowest level of the skip list and do not find the element, then our search has failed.

Skip lists promise $O(\log n)$ due to the fact that there will be $\log n$ levels of the skip list data structure and because the expected number of nodes to traverse at a particular level is constant. For further reading, you can look at the formal analysis of the skip list complexity classes here!

If we want to insert a node into the skip list, we need to start by finding the right place to link it into the data structure. To do so, we start by searching for the key that we would like to insert but maintain a list of *predecessors* and *successors* at every level of list. That way, when we determine how many levels we want to insert the node to, we can perform a linked list insertion at each of these levels. Similarly, if we want to remove an element from the skip list, we do so by unlinking the node from each list it belongs to.

Each of these procedures are provided to you in the `src/sequential/skip_list.hh` file.

2 Parallelization Strategies

In this lab, we will be implementing two different locking strategies to analyze the parallelizability of different approaches: coarse-grained locking and fine-grained locking. We will implement these strategies together in lab!

3 Analyzing Outputs

To understand the differences in performance, we should compare the speedup/slow down of an approach relative to the single threaded case! From this, we can come up with a speedup factor relative to the single threaded version of that run.