Algorithm Design
M. T. Goodrich and R. Tamassia
John Wiley & Sons
Solution of Exercise C-3.26

First, observe that methods $\mathsf{SkipSearch}(k)$ uses only methods $\mathsf{below}(p)$ and $\mathsf{after}(p)$. It is a top-down, scan-forward traversal technique. We examine how an insertion and a removal of an item can be performed in a similar traversing manner.

Insertion: The main idea here is that we can do all the coin flips before we do any skip-list traversals or insertions. The insertion algorithm consists of the following steps:

- Flip the coin several times until heads come up to compute the height of the new tower.
- Compute the height of the skip-list if necessary (unless you store this information and update it during insertion/removal stage). We can always compute the height by going down from the topmost left element till we hit the rock bottom.
- If the number of coin tosses is more than the height of the skip-list, add new level(s), otherwise find the topmost level in which to start insertion (i.e., go down height-of-the-list - number-coin-tosses levels).
- Starting from the position found in the previous step, do something very similar to SkipSearch except that you also have to insert the new element at each level before you skip down.

Removal: For the removal algorithm, we perform a $\mathsf{SkipSearch}(k)$-like traversal down the skip-list for the topmost leftmost item such that the item to the right has key k (if such a key exists). We go on by deleting the whole tower in a top-down fashion.

In both cases, we need to restructure the references between elements (namely references $\mathsf{below}(p)$ and $\mathsf{after}(p)$). We give the pseudo-code that performs these operations. We assume topleft stores the topmost-left position of the skip-list.

Algorithm $\mathsf{insertItem}(k, e)$

```
{compute the height of the tower}
ctosses ← 0
while (random() < 1/2) do
  ctosses ← ctosses + 1
height ← −1
current ← topleft

{find height of skip list}
while (below(current) ≠ null) do
  current ← below(current)
  height ← height + 1

{ Insert new levels if necessary and find the first level to insert at}
if (height < ctosses) then
  for i ← 1 to ctosses − height do
    oldtopleft ← topleft
    topleft ← new Item(−∞,null)
    after(topleft) ← new Item(∞,null)
    below(topleft) ← oldtopleft
    below(after(topleft)) ← after(oldtopleft)
  insertat ← below(topleft)
else
  insertat ← topleft
  for i ← 0 to (height − ctosses − 1) do
        insertat ← below(insertat)

{ Now do SkipSearch, inserting before going down }
oldnewnode ← null
while (insertat ≠ null) do
  while(key(after(insertat)) ≤ k) do
        insertat ← after(insertat)
  newnode ← new Item(k,e)
  after(newnode) ← after(insertat)
  if (oldnewnode ≠ null) then
        below(oldnewnode) ← newnode
  after(insertat) ← newnode
  oldnewnode ← newnode
  insertat ← below(insertat)
```

```
Algorithm removeElement(k)
  current ← topleft
  while (below(current) ≠ null) do
      current ← below(current)
      while key(after(current)) < k do
              current ← after(current)
      if after(current) = k then
              tmp ← after(current)
              after(current) ← after(after(current))
              delete tmp
```

Note that the insertion has $O(\log(n))$ expected time (for the same reasons that SkipSearch has $O(\log(n))$ expected time.

An alternative realization of insertItem could involve the use of a stack. As we perform SkipSearch, we push into a stack every element that we access. Once we are at the bottom of the structure, we start tossing the coin and as long as the flip is coming up "heads" we insert a copy of the inserted element. We track any after(p) reference that needs be updated by performing a pop operation of the stack. Any below(p) reference is handled by storing the last copy added to the tower. Depending on when the flip is coming up "tails", the stack may not become empty or we may need add new levels. The details are left as an exercise.