Chapter 4

Skiplists

In this chapter, we discuss a beautiful data structure: the skiplist, which has a variety of applications. Using a skiplist we can implement a `List` that has $O(\log n)$ time implementations of `get(i)`, `set(i,x)`, `add(i,x)`, and `remove(i)`. We can also implement an `SSet` in which all operations run in $O(\log \mathtt{n})$ expected time.

The efficiency of skiplists relies on their use of randomization. When a new element is added to a skiplist, the skiplist uses random coin tosses to determine the height of the new element. The performance of skiplists is expressed in terms of expected running times and path lengths. This expectation is taken over the random coin tosses used by the skiplist. In the implementation, the random coin tosses used by a skiplist are simulated using a pseudo-random number (or bit) generator.

4.1 The Basic Structure

Conceptually, a skiplist is a sequence of singly-linked lists $L_0, \ldots, L_h$. Each list L_r contains a subset of the items in L_{r-1}. We start with the input list L_0 that contains `n` items and construct L_1 from L_0, L_2 from L_1, and so on. The items in L_r are obtained by tossing a coin for each element, `x`, in L_{r-1} and including `x` in L_r if the coin turns up as heads. This process ends when we create a list L_r that is empty. An example of a skiplist is shown in Figure 4.1.

For an element, `x`, in a skiplist, we call the *height* of `x` the largest value

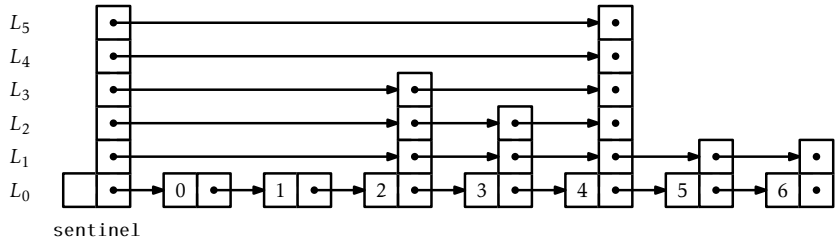

Figure 4.1: A skiplist containing seven elements.

r such that x appears in L_r. Thus, for example, elements that only appear in L_0 have height 0. If we spend a few moments thinking about it, we notice that the height of x corresponds to the following experiment: Toss a coin repeatedly until it comes up as tails. How many times did it come up as heads? The answer, not surprisingly, is that the expected height of a node is 1. (We expect to toss the coin twice before getting tails, but we don't count the last toss.) The *height* of a skiplist is the height of its tallest node.

At the head of every list is a special node, called the *sentinel*, that acts as a dummy node for the list. The key property of skiplists is that there is a short path, called the *search path*, from the sentinel in L_h to every node in L_0. Remembering how to construct a search path for a node, u, is easy (see Figure 4.2) : Start at the top left corner of your skiplist (the sentinel in L_h) and always go right unless that would overshoot u, in which case you should take a step down into the list below.

More precisely, to construct the search path for the node u in L_0, we start at the sentinel, w, in L_h. Next, we examine w.next. If w.next contains an item that appears before u in L_0, then we set w = w.next. Otherwise, we move down and continue the search at the occurrence of w in the list L_{h-1}. We continue this way until we reach the predecessor of u in L_0.

The following result, which we will prove in Section 4.4, shows that the search path is quite short:

Lemma 4.1. *The expected length of the search path for any node,* u, *in* L_0 *is at most* $2\log n + O(1) = O(\log n)$.

A space-efficient way to implement a skiplist is to define a Node, u,

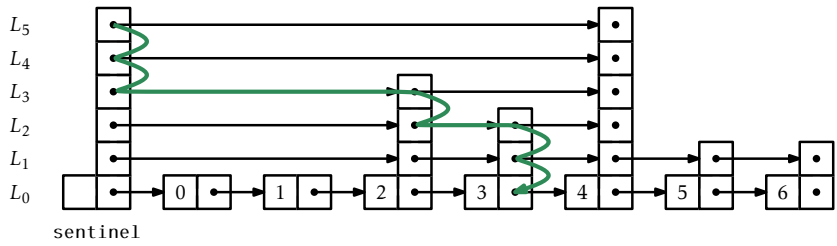

Figure 4.2: The search path for the node containing 4 in a skiplist.

as consisting of a data value, `x`, and an array, `next`, of pointers, where `u.next[i]` points to `u`'s successor in the list L_i. In this way, the data, `x`, in a node is referenced only once, even though `x` may appear in several lists.

SkiplistSSet

```
class Node<T> {
  T x;
  Node<T>[] next;
  Node(T ix, int h) {
    x = ix;
    next = Array.newInstance(Node.class, h+1);
  }
  int height() {
    return next.length - 1;
  }
}
```

The next two sections of this chapter discuss two different applications of skiplists. In each of these applications, L_0 stores the main structure (a list of elements or a sorted set of elements). The primary difference between these structures is in how a search path is navigated; in particular, they differ in how they decide if a search path should go down into L_{r-1} or go right within L_r.

4.2 SkiplistSSet: An Efficient SSet

A SkiplistSSet uses a skiplist structure to implement the SSet interface. When used in this way, the list L_0 stores the elements of the SSet in sorted order. The find(x) method works by following the search path for the smallest value y such that y ≥ x:

SkiplistSSet
```
Node<T> findPredNode(T x) {
  Node<T> u = sentinel;
  int r = h;
  while (r >= 0) {
    while (u.next[r] != null && compare(u.next[r].x,x) < 0)
      u = u.next[r];   // go right in list r
    r--;               // go down into list r-1
  }
  return u;
}
T find(T x) {
  Node<T> u = findPredNode(x);
  return u.next[0] == null ? null : u.next[0].x;
}
```

Following the search path for y is easy: when situated at some node, u, in L_r, we look right to u.next[r].x. If x > u.next[r].x, then we take a step to the right in L_r; otherwise, we move down into L_{r-1}. Each step (right or down) in this search takes only constant time; thus, by Lemma 4.1, the expected running time of find(x) is $O(\log n)$.

Before we can add an element to a SkipListSSet, we need a method to simulate tossing coins to determine the height, k, of a new node. We do so by picking a random integer, z, and counting the number of trailing 1s in the binary representation of z:[1]

SkiplistSSet
```
int pickHeight() {
  int z = rand.nextInt();
```

[1] This method does not exactly replicate the coin-tossing experiment since the value of k will always be less than the number of bits in an int. However, this will have negligible impact unless the number of elements in the structure is much greater than $2^{32} = 4294967296$.

```
  int k = 0;
  int m = 1;
  while ((z & m) != 0) {
    k++;
    m <<= 1;
  }
  return k;
}
```

To implement the `add(x)` method in a `SkiplistSSet` we search for `x` and then splice `x` into a few lists $L_0,\ldots,L_k$, where `k` is selected using the `pickHeight()` method. The easiest way to do this is to use an array, `stack`, that keeps track of the nodes at which the search path goes down from some list L_r into L_{r-1}. More precisely, `stack[r]` is the node in L_r where the search path proceeded down into L_{r-1}. The nodes that we modify to insert `x` are precisely the nodes `stack[0]`,...,`stack[k]`. The following code implements this algorithm for `add(x)`:

SkiplistSSet

```
boolean add(T x) {
  Node<T> u = sentinel;
  int r = h;
  int comp = 0;
  while (r >= 0) {
    while (u.next[r] != null
           && (comp = compare(u.next[r].x,x)) < 0)
      u = u.next[r];
    if (u.next[r] != null && comp == 0) return false;
    stack[r--] = u;          // going down, store u
  }
  Node<T> w = new Node<T>(x, pickHeight());
  while (h < w.height())
    stack[++h] = sentinel;   // height increased
  for (int i = 0; i < w.next.length; i++) {
    w.next[i] = stack[i].next[i];
    stack[i].next[i] = w;
  }
  n++;
  return true;
}
```

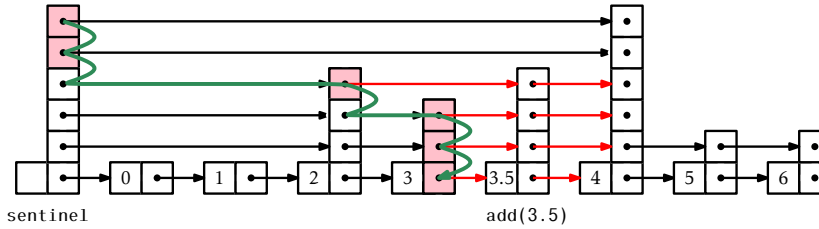

Figure 4.3: Adding the node containing 3.5 to a skiplist. The nodes stored in `stack` are highlighted.

Removing an element, `x`, is done in a similar way, except that there is no need for `stack` to keep track of the search path. The removal can be done as we are following the search path. We search for `x` and each time the search moves downward from a node `u`, we check if `u.next.x = x` and if so, we splice `u` out of the list:

SkiplistSSet

```
boolean remove(T x) {
  boolean removed = false;
  Node<T> u = sentinel;
  int r = h;
  int comp = 0;
  while (r >= 0) {
    while (u.next[r] != null
           && (comp = compare(u.next[r].x, x)) < 0) {
      u = u.next[r];
    }
    if (u.next[r] != null && comp == 0) {
      removed = true;
      u.next[r] = u.next[r].next[r];
      if (u == sentinel && u.next[r] == null)
        h--;  // height has gone down
    }
    r--;
  }
  if (removed) n--;
  return removed;
}
```

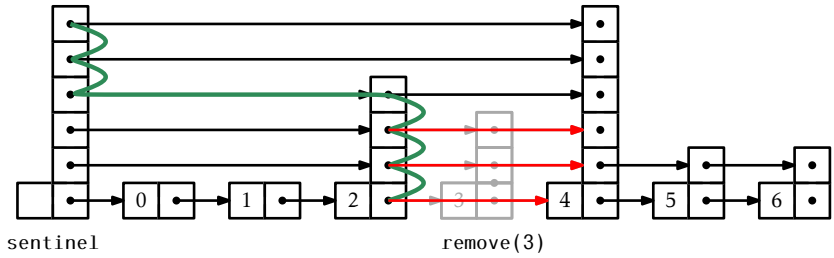

Figure 4.4: Removing the node containing 3 from a skiplist.

4.2.1 Summary

The following theorem summarizes the performance of skiplists when used to implement sorted sets:

Theorem 4.1. *`SkiplistSSet` implements the `SSet` interface. A `SkiplistSSet` supports the operations* `add(x)`, `remove(x)`, *and* `find(x)` *in* $O(\log n)$ *expected time per operation.*

4.3 SkiplistList: An Efficient Random-Access List

A `SkiplistList` implements the `List` interface using a skiplist structure. In a `SkiplistList`, L_0 contains the elements of the list in the order in which they appear in the list. As in a `SkiplistSSet`, elements can be added, removed, and accessed in $O(\log n)$ time.

For this to be possible, we need a way to follow the search path for the ith element in L_0. The easiest way to do this is to define the notion of the *length* of an edge in some list, L_r. We define the length of every edge in L_0 as 1. The length of an edge, e, in L_r, $r > 0$, is defined as the sum of the lengths of the edges below e in L_{r-1}. Equivalently, the length of e is the number of edges in L_0 below e. See Figure 4.5 for an example of a skiplist with the lengths of its edges shown. Since the edges of skiplists are stored in arrays, the lengths can be stored the same way:

SkiplistList

```
class Node {
```

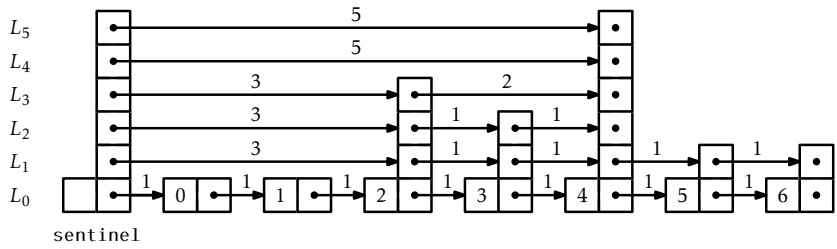

Figure 4.5: The lengths of the edges in a skiplist.

```
  T x;
  Node[] next;
  int[] length;
  Node(T ix, int h) {
    x = ix;
    next = Array.newInstance(Node.class, h+1);
    length = new int[h+1];
  }
  int height() {
    return next.length - 1;
  }
}
```

The useful property of this definition of length is that, if we are currently at a node that is at position `j` in L_0 and we follow an edge of length ℓ, then we move to a node whose position, in L_0, is $\texttt{j} + \ell$. In this way, while following a search path, we can keep track of the position, `j`, of the current node in L_0. When at a node, `u`, in L_r, we go right if `j` plus the length of the edge `u.next[r]` is less than `i`. Otherwise, we go down into L_{r-1}.

SkiplistList

```
Node findPred(int i) {
  Node u = sentinel;
  int r = h;
  int j = -1;   // index of the current node in list 0
  while (r >= 0) {
    while (u.next[r] != null && j + u.length[r] < i) {
      j += u.length[r];
```

```
      u = u.next[r];
    }
    r--;
  }
  return u;
}
```

SkiplistList

```
T get(int i) {
  return findPred(i).next[0].x;
}
T set(int i, T x) {
  Node u = findPred(i).next[0];
  T y = u.x;
  u.x = x;
  return y;
}
```

Since the hardest part of the operations `get(i)` and `set(i,x)` is finding the `i`th node in L_0, these operations run in $O(\log n)$ time.

Adding an element to a `SkiplistList` at a position, `i`, is fairly simple. Unlike in a `SkiplistSSet`, we are sure that a new node will actually be added, so we can do the addition at the same time as we search for the new node's location. We first pick the height, `k`, of the newly inserted node, `w`, and then follow the search path for `i`. Any time the search path moves down from L_r with $r \le k$, we splice `w` into L_r. The only extra care needed is to ensure that the lengths of edges are updated properly. See Figure 4.6.

Note that, each time the search path goes down at a node, `u`, in L_r, the length of the edge `u.next[r]` increases by one, since we are adding an element below that edge at position `i`. Splicing the node `w` between two nodes, `u` and `z`, works as shown in Figure 4.7. While following the search path we are already keeping track of the position, `j`, of `u` in L_0. Therefore, we know that the length of the edge from `u` to `w` is `i - j`. We can also deduce the length of the edge from `w` to `z` from the length, ℓ, of the edge from `u` to `z`. Therefore, we can splice in `w` and update the lengths of the edges in constant time.

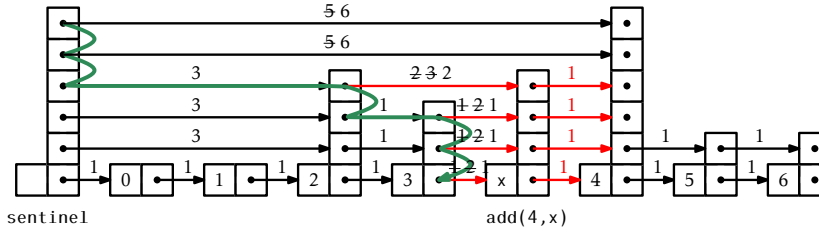

Figure 4.6: Adding an element to a `SkiplistList`.

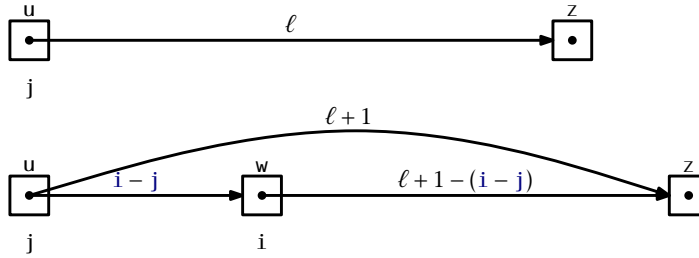

Figure 4.7: Updating the lengths of edges while splicing a node `w` into a skiplist.

This sounds more complicated than it is, for the code is actually quite simple:

SkiplistList
```
void add(int i, T x) {
  Node w = new Node(x, pickHeight());
  if (w.height() > h)
    h = w.height();
  add(i, w);
}
```

SkiplistList
```
Node add(int i, Node w) {
  Node u = sentinel;
  int k = w.height();
  int r = h;
  int j = -1; // index of u
  while (r >= 0) {
```

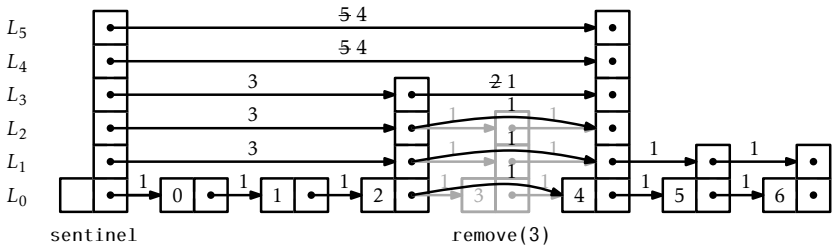

Figure 4.8: Removing an element from a SkiplistList.

```
    while (u.next[r] != null && j+u.length[r] < i) {
      j += u.length[r];
      u = u.next[r];
    }
    u.length[r]++; // accounts for new node in list 0
    if (r <= k) {
      w.next[r] = u.next[r];
      u.next[r] = w;
      w.length[r] = u.length[r] - (i - j);
      u.length[r] = i - j;
    }
    r--;
  }
  n++;
  return u;
}
```

By now, the implementation of the `remove(i)` operation in a `SkiplistList` should be obvious. We follow the search path for the node at position `i`. Each time the search path takes a step down from a node, `u`, at level `r` we decrement the length of the edge leaving `u` at that level. We also check if `u.next[r]` is the element of rank `i` and, if so, splice it out of the list at that level. An example is shown in Figure 4.8.

SkiplistList

```
T remove(int i) {
  T x = null;
  Node u = sentinel;
  int r = h;
```

```
  int j = -1; // index of node u
  while (r >= 0) {
    while (u.next[r] != null && j+u.length[r] < i) {
      j += u.length[r];
      u = u.next[r];
    }
    u.length[r]--;  // for the node we are removing
    if (j + u.length[r] + 1 == i && u.next[r] != null) {
      x = u.next[r].x;
      u.length[r] += u.next[r].length[r];
      u.next[r] = u.next[r].next[r];
      if (u == sentinel && u.next[r] == null)
        h--;
    }
    r--;
  }
  n--;
  return x;
}
```

4.3.1 Summary

The following theorem summarizes the performance of the `SkiplistList` data structure:

Theorem 4.2. *A `SkiplistList` implements the `List` interface. A `SkiplistList` supports the operations* `get(i)`, `set(i,x)`, `add(i,x)`, *and* `remove(i)` *in* $O(\log n)$ *expected time per operation.*

4.4 Analysis of Skiplists

In this section, we analyze the expected height, size, and length of the search path in a skiplist. This section requires a background in basic probability. Several proofs are based on the following basic observation about coin tosses.

Lemma 4.2. *Let T be the number of times a fair coin is tossed up to and including the first time the coin comes up heads. Then* $\mathrm{E}[T] = 2$.

Proof. Suppose we stop tossing the coin the first time it comes up heads. Define the indicator variable

$$I_i = \begin{cases} 0 & \text{if the coin is tossed less than } i \text{ times} \\ 1 & \text{if the coin is tossed } i \text{ or more times} \end{cases}$$

Note that $I_i = 1$ if and only if the first $i-1$ coin tosses are tails, so $\mathrm{E}[I_i] = \Pr\{I_i = 1\} = 1/2^{i-1}$. Observe that T, the total number of coin tosses, can be written as $T = \sum_{i=1}^{\infty} I_i$. Therefore,

$$\begin{aligned} \mathrm{E}[T] &= \mathrm{E}\left[\sum_{i=1}^{\infty} I_i\right] \\ &= \sum_{i=1}^{\infty} \mathrm{E}[I_i] \\ &= \sum_{i=1}^{\infty} 1/2^{i-1} \\ &= 1 + 1/2 + 1/4 + 1/8 + \cdots \\ &= 2 \ . \end{aligned}$$

□

The next two lemmata tell us that skiplists have linear size:

Lemma 4.3. *The expected number of nodes in a skiplist containing* n *elements, not including occurrences of the sentinel, is* $2n$.

Proof. The probability that any particular element, x, is included in list L_r is $1/2^r$, so the expected number of nodes in L_r is $n/2^r$.[2] Therefore, the total expected number of nodes in all lists is

$$\sum_{r=0}^{\infty} n/2^r = n(1 + 1/2 + 1/4 + 1/8 + \cdots) = 2n \ .$$

□

Lemma 4.4. *The expected height of a skiplist containing* n *elements is at most* $\log n + 2$.

Proof. For each $r \in \{1, 2, 3, \ldots, \infty\}$, define the indicator random variable

$$I_r = \begin{cases} 0 & \text{if } L_r \text{ is empty} \\ 1 & \text{if } L_r \text{ is non-empty} \end{cases}$$

[2] See Section 1.3.4 to see how this is derived using indicator variables and linearity of expectation.

The height, h, of the skiplist is then given by

$$h = \sum_{i=1}^{\infty} I_r \ .$$

Note that I_r is never more than the length, $|L_r|$, of L_r, so

$$E[I_r] \leq E[|L_r|] = n/2^r \ .$$

Therefore, we have

$$\begin{aligned} E[h] &= E\left[\sum_{r=1}^{\infty} I_r\right] \\ &= \sum_{r=1}^{\infty} E[I_r] \\ &= \sum_{r=1}^{\lfloor \log n \rfloor} E[I_r] + \sum_{r=\lfloor \log n \rfloor + 1}^{\infty} E[I_r] \\ &\leq \sum_{r=1}^{\lfloor \log n \rfloor} 1 + \sum_{r=\lfloor \log n \rfloor + 1}^{\infty} n/2^r \\ &\leq \log n + \sum_{r=0}^{\infty} 1/2^r \\ &= \log n + 2 \ . \end{aligned}$$

□

Lemma 4.5. *The expected number of nodes in a skiplist containing n elements, including all occurrences of the sentinel, is $2n + O(\log n)$.*

Proof. By Lemma 4.3, the expected number of nodes, not including the sentinel, is $2n$. The number of occurrences of the sentinel is equal to the height, h, of the skiplist so, by Lemma 4.4 the expected number of occurrences of the sentinel is at most $\log n + 2 = O(\log n)$. □

Lemma 4.6. *The expected length of a search path in a skiplist is at most $2 \log n + O(1)$.*

Proof. The easiest way to see this is to consider the *reverse search path* for a node, x. This path starts at the predecessor of x in L_0. At any point in

time, if the path can go up a level, then it does. If it cannot go up a level then it goes left. Thinking about this for a few moments will convince us that the reverse search path for x is identical to the search path for x, except that it is reversed.

The number of nodes that the reverse search path visits at a particular level, r, is related to the following experiment: Toss a coin. If the coin comes up as heads, then move up and stop. Otherwise, move left and repeat the experiment. The number of coin tosses before the heads represents the number of steps to the left that a reverse search path takes at a particular level.[3] Lemma 4.2 tells us that the expected number of coin tosses before the first heads is 1.

Let S_r denote the number of steps the forward search path takes at level r that go to the right. We have just argued that $E[S_r] \leq 1$. Furthermore, $S_r \leq |L_r|$, since we can't take more steps in L_r than the length of L_r, so

$$E[S_r] \leq E[|L_r|] = n/2^r \ .$$

We can now finish as in the proof of Lemma 4.4. Let S be the length of the search path for some node, u, in a skiplist, and let h be the height of the skiplist. Then

$$\begin{aligned}
E[S] &= E\left[h + \sum_{r=0}^{\infty} S_r\right] \\
&= E[h] + \sum_{r=0}^{\infty} E[S_r] \\
&= E[h] + \sum_{r=0}^{\lfloor \log n \rfloor} E[S_r] + \sum_{r=\lfloor \log n \rfloor + 1}^{\infty} E[S_r] \\
&\leq E[h] + \sum_{r=0}^{\lfloor \log n \rfloor} 1 + \sum_{r=\lfloor \log n \rfloor + 1}^{\infty} n/2^r \\
&\leq E[h] + \sum_{r=0}^{\lfloor \log n \rfloor} 1 + \sum_{r=0}^{\infty} 1/2^r
\end{aligned}$$

[3] Note that this might overcount the number of steps to the left, since the experiment should end either at the first heads or when the search path reaches the sentinel, whichever comes first. This is not a problem since the lemma is only stating an upper bound.

$$\begin{aligned}
&\leq \mathrm{E}[h] + \sum_{r=0}^{\lfloor \log n \rfloor} 1 + \sum_{r=0}^{\infty} 1/2^r \\
&\leq \mathrm{E}[h] + \log n + 3 \\
&\leq 2\log n + 5 \ .
\end{aligned}$$

□

The following theorem summarizes the results in this section:

Theorem 4.3. *A skiplist containing* n *elements has expected size* $O(n)$ *and the expected length of the search path for any particular element is at most* $2\log n + O(1)$.

4.5 Discussion and Exercises

Skiplists were introduced by Pugh [62] who also presented a number of applications and extensions of skiplists [61]. Since then they have been studied extensively. Several researchers have done very precise analyses of the expected length and variance of the length of the search path for the ith element in a skiplist [45, 44, 58]. Deterministic versions [53], biased versions [8, 26], and self-adjusting versions [12] of skiplists have all been developed. Skiplist implementations have been written for various languages and frameworks and have been used in open-source database systems [71, 63]. A variant of skiplists is used in the HP-UX operating system kernel's process management structures [42]. Skiplists are even part of the Java 1.6 API [55].

Exercise 4.1. Illustrate the search paths for 2.5 and 5.5 on the skiplist in Figure 4.1.

Exercise 4.2. Illustrate the addition of the values 0.5 (with a height of 1) and then 3.5 (with a height of 2) to the skiplist in Figure 4.1.

Exercise 4.3. Illustrate the removal of the values 1 and then 3 from the skiplist in Figure 4.1.

Exercise 4.4. Illustrate the execution of `remove(2)` on the `SkiplistList` in Figure 4.5.

Exercise 4.5. Illustrate the execution of add(3, x) on the SkiplistList in Figure 4.5. Assume that pickHeight() selects a height of 4 for the newly created node.

Exercise 4.6. Show that, during an add(x) or a remove(x) operation, the expected number of pointers in a SkiplistSet that get changed is constant.

Exercise 4.7. Suppose that, instead of promoting an element from L_{i-1} into L_i based on a coin toss, we promote it with some probability p, $0 < p < 1$.

1. Show that, with this modification, the expected length of a search path is at most $(1/p)\log_{1/p} n + O(1)$.
2. What is the value of p that minimizes the preceding expression?
3. What is the expected height of the skiplist?
4. What is the expected number of nodes in the skiplist?

Exercise 4.8. The find(x) method in a SkiplistSet sometimes performs *redundant comparisons*; these occur when x is compared to the same value more than once. They can occur when, for some node, u, u.next[r] = u.next[r − 1]. Show how these redundant comparisons happen and modify find(x) so that they are avoided. Analyze the expected number of comparisons done by your modified find(x) method.

Exercise 4.9. Design and implement a version of a skiplist that implements the SSet interface, but also allows fast access to elements by rank. That is, it also supports the function get(i), which returns the element whose rank is i in $O(\log n)$ expected time. (The rank of an element x in an SSet is the number of elements in the SSet that are less than x.)

Exercise 4.10. A *finger* in a skiplist is an array that stores the sequence of nodes on a search path at which the search path goes down. (The variable stack in the add(x) code on page 91 is a finger; the shaded nodes in Figure 4.3 show the contents of the finger.) One can think of a finger as pointing out the path to a node in the lowest list, L_0.

A *finger search* implements the `find(x)` operation using a finger, by walking up the list using the finger until reaching a node `u` such that `u.x < x` and `u.next = null` or `u.next.x > x` and then performing a normal search for `x` starting from `u`. It is possible to prove that the expected number of steps required for a finger search is $O(1+\log r)$, where r is the number values in L_0 between `x` and the value pointed to by the finger.

Implement a subclass of `Skiplist` called `SkiplistWithFinger` that implements `find(x)` operations using an internal finger. This subclass stores a finger, which is then used so that every `find(x)` operation is implemented as a finger search. During each `find(x)` operation the finger is updated so that each `find(x)` operation uses, as a starting point, a finger that points to the result of the previous `find(x)` operation.

Exercise 4.11. Write a method, `truncate(i)`, that truncates a `SkiplistList` at position `i`. After the execution of this method, the size of the list is `i` and it contains only the elements at indices $0,\ldots,\texttt{i}-1$. The return value is another `SkiplistList` that contains the elements at indices $\texttt{i},\ldots,\texttt{n}-1$. This method should run in $O(\log \texttt{n})$ time.

Exercise 4.12. Write a `SkiplistList` method, `absorb(l2)`, that takes as an argument a `SkiplistList`, `l2`, empties it and appends its contents, in order, to the receiver. For example, if `l1` contains a,b,c and `l2` contains d,e,f, then after calling `l1.absorb(l2)`, `l1` will contain a,b,c,d,e,f and `l2` will be empty. This method should run in $O(\log \texttt{n})$ time.

Exercise 4.13. Using the ideas from the space-efficient list, `SEList`, design and implement a space-efficient `SSet`, `SESSet`. To do this, store the data, in order, in an `SEList`, and store the blocks of this `SEList` in an `SSet`. If the original `SSet` implementation uses $O(\texttt{n})$ space to store `n` elements, then the `SESSet` will use enough space for `n` elements plus $O(\texttt{n}/\texttt{b}+\texttt{b})$ wasted space.

Exercise 4.14. Using an `SSet` as your underlying structure, design and implement an application that reads a (large) text file and allows you to search, interactively, for any substring contained in the text. As the user types their query, a matching part of the text (if any) should appear as a result.

Hint 1: Every substring is a prefix of some suffix, so it suffices to store all suffixes of the text file.
Hint 2: Any suffix can be represented compactly as a single integer indicating where the suffix begins in the text.
Test your application on some large texts, such as some of the books available at Project Gutenberg [1]. If done correctly, your applications will be very responsive; there should be no noticeable lag between typing keystrokes and seeing the results.

Exercise 4.15. (This excercise should be done after reading about binary search trees, in Section 6.2.) Compare skiplists with binary search trees in the following ways:

1. Explain how removing some edges of a skiplist leads to a structure that looks like a binary tree and is similar to a binary search tree.

2. Skiplists and binary search trees each use about the same number of pointers (2 per node). Skiplists make better use of those pointers, though. Explain why.