SSet implementations			
	find(x)	add(x)/remove(x)	
SkiplistSSet	$O(\log n)^{E}$	$O(\log n)^{E}$	§ 4.2
Treap	$O(\log n)^{E}$	$O(\log n)^{E}$	§ 7.2
ScapegoatTree	$O(\log n)$	$O(\log n)^{A}$	§ 8.1
RedBlackTree	$O(\log n)$	$O(\log n)$	§ 9.2
BinaryTrie^{I}	$O(w)$	$O(w)$	§ 13.1
XFastTrie^{I}	$O(\log w)^{A,E}$	$O(w)^{A,E}$	§ 13.2
YFastTrie^{I}	$O(\log w)^{A,E}$	$O(\log w)^{A,E}$	§ 13.3

(Priority) Queue implementations			
	findMin()	add(x)/remove()	
BinaryHeap	$O(1)$	$O(\log n)^{A}$	§ 10.1
MeldableHeap	$O(1)$	$O(\log n)^{E}$	§ 10.2

^{I} This structure can only store w-bit integer data.

Table 1.2: Summary of SSet and priority Queue implementations.

1.8 Discussion and Exercises

The List, USet, and SSet interfaces described in Section 1.2 are influenced by the Java Collections Framework [54]. These are essentially simplified versions of the List, Set, Map, SortedSet, and SortedMap interfaces found in the Java Collections Framework.

For a superb (and free) treatment of the mathematics discussed in this chapter, including asymptotic notation, logarithms, factorials, Stirling's approximation, basic probability, and lots more, see the textbook by Leyman, Leighton, and Meyer [50]. For a gentle calculus text that includes formal definitions of exponentials and logarithms, see the (freely available) classic text by Thompson [71].

For more information on basic probability, especially as it relates to computer science, see the textbook by Ross [63]. Another good reference, which covers both asymptotic notation and probability, is the textbook by Graham, Knuth, and Patashnik [37].

Exercise 1.1. This exercise is designed to help familiarize the reader with

or remove(i) is at least

$$\sum_{i=1}^{r}(n_i/2-1) \le m \ ,$$

which is equivalent to

$$\sum_{i=1}^{r} n_i \le 2m + 2r \ .$$

On the other hand, the total time spent during all calls to resize() is

$$\sum_{i=1}^{r} O(n_i) \le O(m+r) = O(m) \ ,$$

since r is not more than m. All that remains is to show that the number of calls to add(i,x) or remove(i) between the $(i-1)$th and the ith call to resize() is at least $n_i/2$.

There are two cases to consider. In the first case, resize() is being called by add(i,x) because the backing array a is full, i.e., a.length = n = n_i. Consider the previous call to resize(): after this previous call, the size of a was a.length, but the number of elements stored in a was at most a.length/2 = $n_i/2$. But now the number of elements stored in a is n_i = a.length, so there must have been at least $n_i/2$ calls to add(i,x) since the previous call to resize().

The second case occurs when resize() is being called by remove(i) because a.length $\ge 3n = 3n_i$. Again, after the previous call to resize() the number of elements stored in a was at least a.length/2 − 1.[1] Now there are $n_i \le$ a.length/3 elements stored in a. Therefore, the number of remove(i) operations since the last call to resize() is at least

$$\begin{aligned} R &\ge \texttt{a.length}/2 - 1 - \texttt{a.length}/3 \\ &= \texttt{a.length}/6 - 1 \\ &= (\texttt{a.length}/3)/2 - 1 \\ &\ge n_i/2 - 1 \ . \end{aligned}$$

In either case, the number of calls to add(i,x) or remove(i) that occur between the $(i-1)$th call to resize() and the ith call to resize() is at least $n_i/2-1$, as required to complete the proof. □

[1] The −1 in this formula accounts for the special case that occurs when n = 0 and a.length = 1.

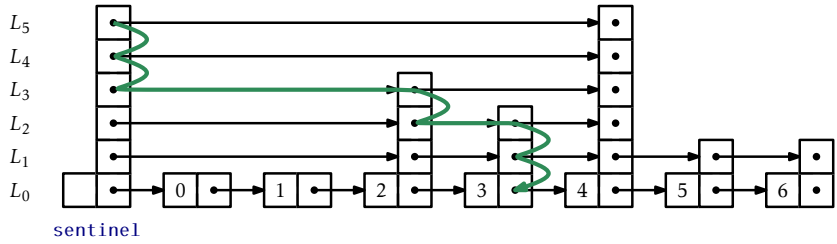


Figure 4.2: The search path for the node containing 4 in a skiplist.

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 stored only once, even though `x` may appear in several lists.

SkiplistSSet
```
struct Node {
  T x;
  int height;      // length of next
  Node *next[];
};
```

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` $\geq$ `x`:

SkiplistSSet
```
Node* findPredNode(T x) {
```