15.1. Skip Lists

15.1.1. Skip Lists

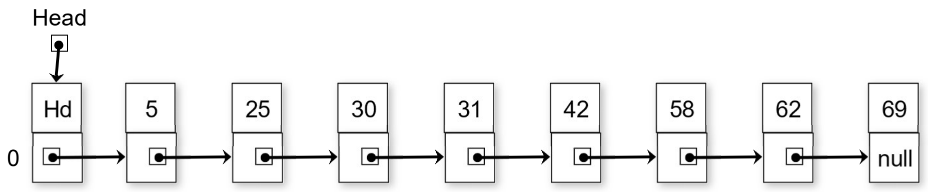
This module presents a probabilistic search structure called the ***skip list***. Like the ***BST***, skip lists are designed to overcome a basic limitation of array-based and linked lists: Either search or update operations require linear time. The skip list is an example of a ***probabilistic data structure***, because it makes some of its decisions at random.

Skip lists provide an alternative to the BST and related tree structures. The primary problem with the BST is that it may easily become unbalanced. The ***2-3 Tree*** is guaranteed to remain balanced regardless of the order in which data values are inserted, but it is rather complicated to implement. The ***AVL tree*** and the ***splay tree*** are also guaranteed to provide good performance, but at the cost of added complexity as compared to the BST. The skip list is easier to implement than known balanced tree structures. The skip list is not guaranteed to provide good performance (where good performance is defined as $\Theta(\log n)$ search, insertion, and deletion time), but it will provide good performance with extremely high probability (unlike the BST which has a good chance of performing poorly). As such it represents a good compromise between difficulty of implementation and performance.

1 / 18

<< < > >>

Here we will illustrate the skip list concept. A skip list can be viewed as a sorted linked list with some extra pointers. We start with a simple linked list whose nodes are ordered by key value. To search this list requires that we move down the list one node at a time, visiting O(n) nodes in the average case.



We can continue adding pointers to selected nodes in this way --- give a third pointer to every fourth node, give a fourth pointer to every eighth node, and so on—until we reach the ultimate of $\log n$ pointers in the first and middle nodes for a list of n nodes. To search, start with the bottom row of pointers, going as far as possible and skipping many nodes at a time. Then, shift up to shorter and shorter steps as required. With this arrangement, the worst-case number of accesses is $\Theta(\log n)$.

We will store with each skip list node an array named `forward` that stores the pointers. Position `forward[0]` stores a level 0 pointer, `forward[1]` stores a level 1 pointer, and so on. It also uses a KVPair to store the key and record for the node. The SkipNode class follows:

Java | **Processing** | **Java (Generic)**

```
class SkipNode {
   private KVPair rec;
   private SkipNode[] forward;

   public Object element() {
     return rec.value();
   }

   public Comparable key() {
     return rec.key();
   }

   public SkipNode(Comparable key, Object elem, int level) {
     rec = new KVPair(key, elem);
     forward = new SkipNode[level + 1];
     for (int i = 0; i < level; i++)
       forward[i] = null;
   }

   public String toString() {
     return rec.toString();
   }
 }
```

The skip list object includes data member `level` that stores the highest level for any node currently in the skip list. The skip list stores a header node named `head` with `level+1` pointers where the head level is initially 0 and the level is set to -1 for the empty list. The start of the SkipList class follows:

Java **Processing** **Java (Generic)**

```
class SkipList implements Dictionary {
  private SkipNode head;
  private int level;
  private int size;
  static private Random ran = new Random(); // Hold the Random class object

  public SkipList() {
    head = new SkipNode(null, null, 0);
    level = -1;
    size = 0;
  }
```

The `find` function works as follows.

Java **Processing** **Java (Generic)**

```
  // Return the (first) matching matching element if one exists, null otherwise
  public Object find(Comparable key) {
    SkipNode x = head; // Dummy header node
    for (int i = level; i >= 0; i--) // For each level...
      while ((x.forward[i] != null) && (x.forward[i].key().compareTo(key) < 0)) // go forward
        x = x.forward[i]; // Go one last step
    x = x.forward[0]; // Move to actual record, if it exists
    if ((x != null) && (x.key().compareTo(key) == 0)) return x.element(); // Got it
    else return null; // Its not there
  }
```

The ideal skip list is organized so that (if the head node is not counted) half of the nodes have only one pointer, one quarter have two, one eighth have three, and so on. And ideally, the distances would be equally spaced; in effect this is a "perfectly balanced" skip list. Maintaining such balance would be expensive during the normal process of insertions and deletions. The key to skip lists is that we do not worry about any of this. Whenever inserting a node, we assign it a level (i.e., some number of pointers). The assignment is random, using a geometric distribution yielding a 50%

probability that the node will have one pointer, a 25% probability that it will have two, and so on. The following function determines the level based on such a distribution.

Java **Processing** **Java (Generic)**

```
// Pick a level using a geometric distribution
int randomLevel() {
  int lev;
  for (lev = 0; Math.abs(ran.nextInt()) % 2 == 0; lev++) // ran is random generator
    ; // Do nothing
  return lev;
}
```

Once the proper level for the node has been determined, the next step is to find where the node should be inserted and link it in as appropriate at all of its levels. Here is an implementation for inserting a new value into the skip list followed by a visualization of the process. Note that we build an `update` array as we progress through the skip list, so that we can update the pointers for the nodes that will precede the one being inserted.

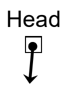
Java **Processing** **Java (Generic)**

```
/** Insert a key, element pair into the skip list */
public void insert(Comparable key, Object elem) {
  int newLevel = randomLevel(); // New node's level
  if (newLevel > level) // If new node is deeper
    adjustHead(newLevel); // adjust the header
  // Track end of level
  SkipNode[] update = new SkipNode[level + 1];
  SkipNode x = head; // Start at header node
  for (int i = level; i >= 0; i--) { // Find insert position
    while ((x.forward[i] != null) && (x.forward[i].key().compareTo(key) < 0))
      x = x.forward[i];
    update[i] = x; // Track end at level i
  }
  x = new SkipNode(key, elem, newLevel);
  for (int i = 0; i <= newLevel; i++) { // Splice into list
    x.forward[i] = update[i].forward[i]; // Who x points to
    update[i].forward[i] = x; // Who points to x
  }
  size++; // Increment dictionary size
}

private void adjustHead(int newLevel) {
  SkipNode temp = head;
  head = new SkipNode(null, null, newLevel);
  for (int i = 0; i <= level; i++)
    head.forward[i] = temp.forward[i];
  level = newLevel;
}
```

1 / 36 ≪ < > ≫

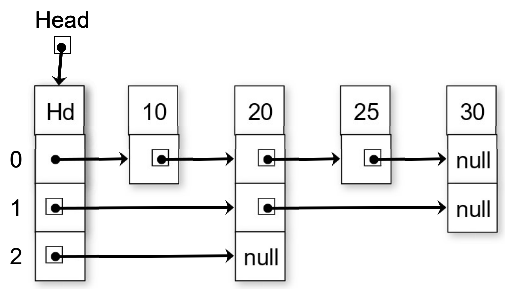
Now we will illustrate skip list insertion. The skip list is initialized with a header node of level 0, whose forward pointer is set to null. The top item shows the value associated with the skip list node. The head node is special so has the value "Hd".



The `remove` function is similar to insertion in that the `update` array is built as part of searching for the record to be deleted. Then those nodes specified by the update array have their forward pointers adjusted to point around the node being deleted.

1 / 42

The remove operation is similar to insertion in that it also uses an update array to search for the position of the key in the skip list before removing it.



A newly inserted node could have a high level generated by `randomLevel`, or a low level. It is possible that many nodes in the skip list could have many pointers, leading to unnecessary insert cost and yielding poor (i.e., $\Theta(n)$) performance during search, because not many nodes will be skipped. Conversely, too many nodes could have a low level. In the worst case, all nodes could be at level 0, equivalent to a regular linked list. If so, search will again require $\Theta(n)$ time. However, the probability that performance will be poor is quite low. There is only one chance in 1024 that ten nodes in a row will be at level 0. The motto of probabilistic data structures such as the skip list is "Don't worry, be happy". We simply accept the results of `randomLevel` and expect that probability will eventually work in our favor. The advantage of this approach is that the algorithms are simple, while requiring only $\Theta(\log n)$ time for all operations in the average case. For a skip list of size n, the expected memory usage is $2n$. This is because a level l node needs $l+1$ forward pointers, but occurs with probability $2^{(l+1)}$. So a skip list is expected to have $\sum_{l=0}^{l=\infty}(l+1)/2^{(l+1)}$ pointers, which is 2. Thus, the number of pointers needed by both the BST and the skip list are expected to be the same.

In practice, the skip list will probably have better performance than a BST storing the same data. The BST can have bad performance caused by the order in which data are inserted. For example, if n nodes are inserted into a BST in ascending order of their key values, then the BST will look like a linked list with the deepest node at depth $n-1$. If the data inserted over the life of the BST could be randomly ordered, then the probability distribution for the cost of the insert and search operations would be similar to that of the skip list. The problem for the BST is that this randomization does not happen in fact, but rather the BST is constrained by the actual order of inputs and searches.

In contrast, the skip list's performance does not depend on the order in which values are inserted into the list. In a sense, the data are "randomized" automatically as part of the skip list's probabilistic behavior when the depths of the nodes are selected. As the number of nodes in the skip list increases, the probability of encountering the worst case decreases geometrically. Thus, the skip list illustrates a tension between the theoretical worst case (in this case, $\Theta(n)$ for a skip list operation), and a rapidly increasing probability of average-case performance of $\Theta(\log n)$, that characterizes probabilistic data structures.