CPS222 Lecture: Skip Lists

last revised 3/11/2015

Objectives:

1. To introduce skip lists

Materials:

1. Projectable of Figure 9.9
2. Graphical demo program and demo files animals.slist and animals.txt

I. Introduction

A. We have looked at two basic approaches to implementing search structures (which implement collections such as sets or maps)

1. A hashtable achieves performance that is probable - but not guaranteed - to be O(1).

   a) However, entries in a hashtable are not maintained in any particular order.

   b) Thus, a hashtable is useful for telling whether or not a given item is in the collection, or for locating the value associated with a given key, but is not useful for printing the keys in order.

2. A binary search tree does maintain keys in order, but has performance that cannot be guaranteed to be better than O(log n) (assuming we use some sort of balanced tree).

   a) A binary search tree does maintain keys in natural order; therefore, an inorder traversal of the tree can be used to visit the keys in natural order.

b) However, this requires that the code be aware of the structure of the tree. It is not easily possible to create a conventional iterator that supports natural order iteration through the tree.

3. Today, we look at a third type of search structure which - like the binary search tree - maintains keys in order and has O(log n) performance.

4. Since a skip list doesn't offer any capabilities not offered by a binary search tree, one may ask why we are spending time on it. There are two answers to this question.

   a) It is easy to iterate through the entries in the structure in natural order using a conventional iterator.

   b) The probabilistic strategy it uses is interesting, in that it maintains its performance by taking advantage of the behavior of random numbers, rather than by taking explicit steps to maintain a balanced tree,

B. A skip list - like a binary search tree - is based on keys of some ordered type (e.g. integers, strings). It makes use of two special values of key - neither of which can occur in actual data.

1. One value - conventionally called $-\infty$ must test less than any legitimate key. (If we are using strings, we can use the empty string - provided that no legitimate key can be "". If we are using integers, there must be a very small value that cannot legitimately occur).

2. The other value - conventionally called $+\infty$ must test greater than any legitimate key. (If we are using strings, we can use a string consisting of only the delete character - whose ASCII code is 127 - and which cannot be a legitimate key since it cannot be typed conventionally on the keyboard. If we are using integers, there must be a very large value that cannot legitimately occur).

C. A skip list consists of a set of lists of nodes, regarded as a series of levels. No keys occurs twice on the same level. The special keys -∞, and +∞ occur on every level.

1. The bottom level list includes -∞, every actual key, and +∞ in proper order. If the skip list is used for a map, it also contains the values associated with each key. (The values need not appear on the higher levels).

   a) Since every key is present on the bottom level, of course, every value is also present there.

   b) It is easy to construct an iterator that iterates over this list, visiting every key exactly once in natural order.

2. Each subsequent level contains -∞, a subset of the keys, and +∞ in proper order. Probabilistic techniques are used to provide that each level contain about half as many legitimate keys as the one below it.

3. This means that (if we number levels starting at 0 for the bottom level.

   a) Level 1 contains about 1/2 the keys

   b) Level 2 contains about 1/4 of the keys
      ...
   c) Level k contains about $1/2^k$ of the keys
      ...
   d) The level just below the top contains about 1 key. From this it follows that there are about log n levels in all containing real keys

4. The highest level contains just -∞ and +∞, but no keys.

5. Each node includes a link to the node on its right and to the node directly below it. (The node to the right will contain a greater key; that below it will contain the same key).

a) We call the set of nodes that lie in a vertical line (all of which contain the same key) a tower.

b) Bottom-level nodes contains null as their down link.

PROJECT Figure 9.9 from book

II. Operations on a Skip List

A. It turns out that the operations lookup, insert, and remove can be done on a skip list in O(log n) time.

B. We begin with lookup. To perform a lookup, we use the following approach. (We will call the node we are currently with `current`)

1. Set current to be the leftmost node in the top level list (which will be -∞)

2. While we are not yet at the bottom level:

   a) Set current to the node just below current (which will contain the same key.

   b) Advance current to be the node in its level having the largest key that is not greater than what we are looking for. (Will be = to the key we are looking for if it occurs in this level, else <)

3. When we get to the bottom level, either current will contain the key we are looking for or the key doesn't occur in the list (since all keys occur on the bottom level)

4. Observe that this algorithm only examines a few (ideally just 1 or 2) nodes on each level.

   a) The first level where we actually look at keys is one below the top level, and it contains approximately 1 key.

b) Levels further down contain more keys, but the search at the higher level has reduced the range of keys we need to consider.

c) If there log n levels, then we examine O(log n) nodes

5. DEMO using demo program on `animals.slist`

C. Now we consider insert.

1. Assume that we require that the key we insert be one that is not currently in the list.

2. We use same basic approach as with lookup, except that we keep track of the last value of current for each level. (We can do this with a simple array of node pointers having one element per level).

   Note that if the key is not yet present in the list then these nodes will be the the predecessors of the key we are inserting.

3. We now insert a new node at the bottom level, and then we work our way up, choosing a true/false value randomly at each level.

   a) If this value is true, we insert a copy of the key at this level as well - which is easy to do, since we noted the predecessor on the way down.

   b) As soon as we get a false value, we stop inserting copies.

   c) If we end up inserting a new node in the top level (which should just contain $-\infty$, and $+\infty$), we add a new top level containing just those keys and no others.

4. Observe the probability that we insert the key at k levels is simply $1/2^{(k-1)}$, which becomes very small as k becomes large. So in most cases we just insert the key at a small number of levels. The overall cost is therefore (cost of going down) + (cost of going up), and the amortized cost is O(log n).

5. DEMO using demo program.

    a) Load `animals.slist`

    b) Do some manual insertions

    c) Do bulk insert using `animals.txt` - note how a different list results each time

D. Remove is similar to lookup - except

1. When working our way across the level, we stop at the last node whose key is less than what we are looking for (rather than <=)

2. If we encounter the key we are looking for on any level, we remove it from that level, and then go down its tower. (This requires that, at each level, we keep track of the predecessor of teh current node, which is easily done.)