CS61B: Lecture 40
Wednesday, April 30, 2014

Generational Garbage Collection

Studies of memory allocation have shown that most objects allocated by most programs have short lifetimes, while a few go on to survive through many garbage collections. This observation has inspired generational garbage collectors, which separate old from new objects.

A generational collector has two or more generations, which are like the separate spaces used by copying collectors, except that the generations can be of different sizes, and can change size during a program's lifetime.

Sun's 1.3 JVM divides objects into an old generation and a young generation. Because old objects tend to last longer, the old generation doesn't need to be garbage collected nearly as often. Hence, the old generation uses a compacting mark-and-sweep collector, because speed is not critical, but memory efficiency might be. Because old objects are long-lived, and because mark and sweep only uses one memory space, the old generation tends to remain compact.

The young generation is itself divided into three areas. The largest area is called "Eden", and it is the space where all objects are born, and most die. Eden is large enough that most objects in it will become garbage long before it gets full. When Eden fills up, it is garbage collected and the surviving objects are copied into one of two _survivor_spaces_. The survivor spaces are just the two spaces of a copying garbage collector.

If an unexpectedly large number of objects survive Eden, the survivor spaces can expand if necessary to make room for additional objects.

Objects move back and forth between the two survivor spaces until they age enough to be _tenured_ - moved to the old generation. Young objects benefit from the speed of the copying collector while they're still wild and prone to die young.

Thus, the Sun JVM takes advantage of the best features of both the mark-and-sweep and copying garbage collection methods.

There are two types of garbage collection: minor collections, which happen frequently but only affect the young generation - thereby saving lots of time - and major collections, which happen much less often but cover all the objects in memory.

This introduces a problem. Suppose a young object is live only because an old object references it. How does the minor collection find this out, if it doesn't search the old generation?

References from old objects to young objects tend to be rare, because old objects are set in their ways and don't change much. Since references from old objects to young are so rare, the JVM keeps a special table of them, which it updates whenever such a reference is created. The table of references is added to the roots of the young generation's copying collector.

```
-------------------------------------------------------------------------
|                                                                       |
| old generation                                                        |
|                                                                       |
|                                                                       |
-------------------------------------------------------------------------
|                                                                       |
| young generation                                                      |
|                                                                       |
|  -------------------------------------------------------------------  |
|  | survivor space                                                  |  |
|  |                                                                 |  |
|  -------------------------------------------------------------------  |
|  | survivor space                                                  |  |
|  |                                                                 |  |
|  -------------------------------------------------------------------  |
|                                 _____                   ____          |
|      /----\               /----/     \/\/\         /---/    \_____    |
|    _/      \     -\      /                \____--___/            \    |
|   /         \____/ \__--/                                         |   |
|  |                                                               /    |
|  |                             Eden                              \    |
|   \                                                               |   |
|    \                                    _                ^       /    |
|     -\   /\_    _/--\     /\    /\     / \--\    /--\   / \___  /     |
|       --/   \__/     \____/ \/\_/  \__/      \/\/    --/     \_/      |
-------------------------------------------------------------------------
```

AUGMENTING DATA STRUCTURES

Once you know how to design one of the data structures taught in this class, it's sometimes easy to augment it to have "extra" abilities.

You've already augmented data structures in Project 3. For example, the set E of edges is stored as both a hash table and an adjacency list. The hash table allows you to test set membership in O(1) time, unlike the adjacency list. The adjacency list tells you the edges adjoining a vertex in O(degree) time, unlike the hash table.

2-3-4 Trees with Fast Neighbors

Suppose you have a 2-3-4 tree with no duplicate keys. Given a key k, you want to be able to determine whether k is in the tree, and what the next smaller and larger keys are, in O(1) time. You are allowed to change the insert() and remove() operations, but they still must take O(log n) time. Can you do it?

It's easy if you combine the 2-3-4 tree with a hash table. The hash table maps each key to a record that stores the next smaller and next larger keys in the tree.

```
                    ----------------      ---------------
                    |              |      | ----- ----- |
                    |  Hash table  |      | | 4 | | 9 | |
              5 ----+\/\/\/\/\/\/\/+----->| ----- ----- |
                    ----------------      | prev   next |
                                          ---------------
```

The trick is that when you insert a key into the tree, you can determine by tree search in O(log n) time what the next smaller and larger keys are. Then, you update all three keys' records in the hash table in O(1) time.

When you remove a key from the tree, you can learn its two neighboring keys from the hash table, then update the neighbor records for those two keys so they list each other instead of the removed key. You also remove the key's record from the hash table. The hash table updates take O(1) time, and it takes O(log n) time to remove the key from the 2-3-4 tree itself.

Splay Trees with Node Information

Sometimes it's useful for a binary search tree to record extra information in each node, like the size and height of each subtree at each node.

In splay trees, this is easy to maintain. Splaying is just a sequence of tree rotations. Each rotation changes the sizes of only two subtrees, and we can easily compute their new sizes after the rotation. Let size(Y) be the number of nodes in the subtree rooted at node Y. After a right rotation (for instance) you can recompute the information as follows:

```
size(Y) = 1 + size(B) + size(C)                  Y                       X
size(X) = 1 + size(A) + size(Y)                 / \                     / \
                                               X   ^                   ^   Y
height(Y) = 1 + max{height(B), height(C)}     / \ /C\                 /A\ / \
height(X) = 1 + max{height(A), height(Y)}    ^   ^      ------------>     ^  ^
(Note:  to make this work, we must say      /A\/B\      rotate right     /B\/C\
that the height of an empty tree is -1.)
```

Be forwarned that a rotation does not just change the heights of X and Y--it also can change the heights of all their ancestors. But X gets splayed all the way to the root, so all the ancestors' heights get fixed on the way up.

Likewise, inserting or removing an item changes the subtree sizes of all the ancestors of the affected item, and possibly their heights as well. But a newly inserted item gets splayed to the top; and a removed node's parent is splayed to the top. So again, all the sizes and heights will get fixed during the rotations. Let's watch the size fields as we insert a new node X into a splay tree. (The following numbers are sizes, _not_ keys.)

Note that the very first rotation is at the grandparent of node X (zig-zig).

```
    10              10              10                   10             [11]
   /  \            /  \            /  \                  /  \            / \
  8    1          8    1          8    1               [9]  1          6   4
 / \             / \             / \                  / \             /\   /\
1   6           1   6           1   6                6   2           1  4 2  1
   / \             / \             / \              / \   \             /   \
  4   1 =zig=>    5   1 =zig=>   [5]  1 =zig-zag=> 1  4    1 =zig=>   3     1
 / \             / \             /                   /               / \
1   2           3  [1]          4                   3               1   1
   / \         / \           /                   / \
  1  [X]      1   1           3                   1   1
                             / \
                            1   1
```

```
How can we use this information?  We can answer the query "How       3  find(4)
many keys are there between x and y?" in O(log n) amortized        / \
time if the splay tree has no duplicate keys and we label every    2   5
subtree with its size.  Our strategy is to set c = n, then        /     \
deduct from c the number of keys outside the range [x, y].       1       8
                                                                        / \
  find(x);  // After the splaying, the keys in the root's left         6   9
  // subtree are all less than x, so subtract their number from c.
  c = c - size(root's left subtree);                                 6  find(7)
  if (root key < x)  // Only possible if x is not in the tree--     / \
    c--;             // otherwise x was splayed to the root.       3   8
                                                                  / \   \
  find(y);  // After the splaying, the keys in the root's        2   5   9
            // right subtree all exceed y.                      /
  c = c - size(root's right subtree);                          1
  if (root key > y) c--;
                                                             Keys in [4, 7] =
Now, c is the number of keys in [x, y].                      7 - 2 - 1 - 2 = 2.
```