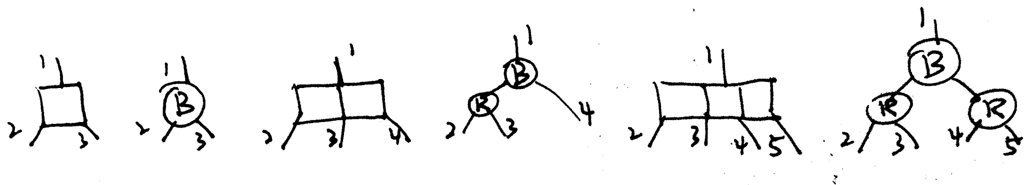
CST Part IA: Algorithm, SV 4
Joe Yan
2017-2-24

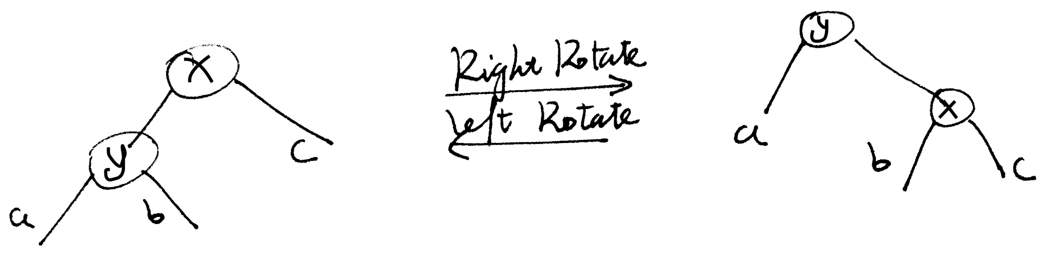
1 2009P1Q5

(a)
- The root node only has a left child and a right child. The leaf node (nil) only has a parent. All other nodes have a parent and a left child and a right child.
- Fix a parent node. All keys in the left subtree are smaller than the key in the parent node. All keys in the right subtree are larger than the key in the parent node.
- The color of each node is either red or black.
- The color of the left and right children of a red node must be black.
- All leaf nodes (nil) are black.
- And so all paths which directly start from the root to a leaf node contain the same number of black nodes.

By the color property, RB tree is approximately balanced because the longest path which directly starts from the root to a leaf node cannot be more than two times longer than the shortest such path. This is an advantage of RBT over BST.
However, maintaining the color property takes lots of re-color and reference-resign effort during insertions and deletions which is a disadvantage of RBT.

(b) 

(c) * The black-height does not count leaf nodes (nil) as black and all leaf nodes in the answer strictly mean the nil node which is used as a label of the end of the tree and it is not counted as a node.
The minimum possible numbers of nodes happens when the RB tree has the smallest possible height h which is the same as the black-height. By definition which all paths which directly start from the root to a leaf node contain the same number of black nodes, such RB tree is a full balanced tree which contains only black nodes with height h. Overall, it contains at least $2^h - 1$ nodes.
The maximum possible numbers of nodes happens when the RB tree has the largest possible height $2h$. Because the color of the left and right children of a red node must be black and the root node must be black the longest which directly starts from the root to a leaf node contains $2h$ nodes with color "Black-Red-Black-Red-...Black-Red" exclude the nil leaf node. Such RB tree is a full balanced tree with $2^{2h} - 1$ nodes.
Overall, $2^h - 1 \leq number\ of\ nodes\ in\ a\ RBT\ with\ black - height\ h \leq 2^{2h} - 1$

(d) 

Rotation is a restructure operation on trees with binary search tree property. It preserves the binary search tree property but may violate RB color property. How it is done is shown in the picture.

(e) As the question describes, the algorithm should know exactly how the BST looks like before and after the reshape and there is no duplication of keys in the BST.
Property of rotation (used later in the proof)

- The rotation preserves the BST property.
- The (right) rotation increases the depth of the node x and all nodes in subtrees c by 1 and decreases the depth of node x and all nodes in subtree a by 1. The depth of all nodes in sub-tree b keeps constant. The left rotation has similar argument and so for all nodes except the root node in a BST the algorithm can always decrease its depth by 1 which makes it closer to one of its super-parent (except parent).

Prove by induction and also the description of how such algorithm is implemented.
Base case:
Algorithm can always find where the root node of the target BST is in the prototype BST by BST search. By the second property of rotation, the algorithm can always do rotation to raise the node until it reaches the root. Now the modified BST has exactly the same shape as the target BST beyond depth 1. By the first property of rotation, the BST property holds for the modified BST and so the left and right sub-trees of the root in the modified BST contains exactly the same nodes as target element otherwise the BST property will not hold. This property is crucial for whether the induction can proceed and so call it property Q for the root node holds.
Induction case:
Suppose the modified BST has exactly the same shape as the target BST beyond depth k and property Q holds for all nodes beyond depth k.
Pick arbitrary nodes with depth k+1 in the target node and search it in the modified BST. Because property Q holds for the super-parent in depth k of such node in the modified BST the algorithm can raise such node in the modified BST to the root of the sub-tree of its super-parent in depth k by the same argument as the base case by doing a series of rotations. Now this node is in the same position as the target BST. Also by the second property of rotation, property Q holds for this node which is now in depth k+1.
Repeat the same procedure for all nodes in depth $k + 1$ in the target tree.
Then all nodes beyond depth $k + 1$ in the modified tree will be the same as the target tree and property Q holds for all such nodes.
Overall, by mathematical induction all nodes in the modified tree will be the same as the target tree.
Finally such algorithm must terminate and the induction will stop because the height of the target tree is finite.

2 2008P10Q9

(a)

0	35	10	5
1	6		
2	2		
3	18	3	8
4			

(b)

0	10
1	
2	2
3	3
4	
5	35
6	6
7	5
8	18
9	8

(c)i Generally there are three steps for a insertion to a chain structured table. FIRST step is using hash function to find the head of the chain in the array and SECOND step is try to putting the key-value pair into the chain with a searching for the first free entry in the array which is THIRD step.
FIRST, SECOND and THIRD are just labels which do not mean the step must be proceeded or proceeded in this sequence.
Each of these steps is required to have constant time complexity.

THIRD By the hint from the question, there should be an auxiliary double linked list for storing the free entry and so we can find the first free entry in the array in a constant time cost. The algorithm also need to maintain this auxiliary data structure which means it should take constant time to remove arbitrary entry from any position in the list. This demonstrates every free entry in this structure should have a reference to the previous and next free entries. i.e. It must be a double linked list as the hint describes. Also the algorithm should take constant time to reach any node in the double linked list because the algorithm will also delete some node in the middle of the double linked list so the reference of each double linked node should also be stored in a array to achieve the random access and so delete arbitrary specific node in a constant time.

SECOND When we use hash value of the key to get a entry in the array this entry must be the head of the chain if such chain has at least one key-value pair already. SECOND only happens under this assumption. So the algorithm spends constant time to find the head of the chain which means prepending the key-value pair to the chain spends least time. The algorithm move the previous head to the first free entry in the array and then put the new head to the original head position and make the pointer of the new head point to the previous head.

FIRST Hash function calculation and array reference should both take a constant time.

MOREOVER After I finish the pseudo code, I find it is also necessary to use double linked structure for the chain because the algorithm needs to know the parent of the moved entry when it is not the head of a chain otherwise there is no way to fix the chain in a constant time. (The algorithm needs to start from the head of the chain to find the moved entry's parent if using single linked list.)

* If you find this unreadable try the human-language version because fixing the auxiliary structure is clumsy (also run out of letter in define...) and not quite the point of this question... Define A[] to be the entry array, each entry stores ((KEY,VALUE),X,Z).
POINTER X points to the previous entry in the chain.
POINTER Z points to the next entry in the chain.
Define B[] to store a reference of every node in double linked list C if the entry is free. When the corresponding entry is not free and is the head of a chain, assign a label node Y to it. Otherwise which is not a head of a chain, assign a label node N to it.
Define C is a double linked list store a node corresponding to an entry in the entry array if and only if the entry is free. Each node has structure (X,KEY,Z).

POINTER X points to the previous free entry in the chain.
POINTER Z points to the next free entry in the chain.
Define H to be the hash function.
Define (K,V) to be the key-value pair.
Pseudo code:

```
If B[H(K)] != Y or N //equivalent to A[H(K)] is free
—A[H(K)] = ((K,V),null)
—C.remove(B[H(K)])
—B[H(K)] = null
Else
—If C.empty()
——Raise HashTableOverflow
—If A[H(K)].(KEY,VALUE).KEY == K
——A[H(K)] = ((K,V),null,A[H(K)].Z)
——RETURN;
—int temp = C.first().key() //move original entry to the first free entry
—A[temp] = A[H(K)]
—C.remove(B[temp])
—B[temp] = N
—If B(H(K)) == Y //build the new head of the old chain
——A[H(K)] = ((K,V),null,temp)
——A[temp].X = H(K) //fix the original chain head entry
—Else //build the new head of a new chain
——A[H(K)] = ((K,V),null,null)
——B[H(K)] = Y
——A[A[temp].X].Z = temp //fix the reference to the moved entry
——A[A[temp].Z].X = temp
RETURN
```

Pseudo code (human-friendly version, ignoring all auxiliary structure):

```
If A[H(K)] is free
—A[H(K)] = (K,V)
Else
—If no free entry in A
——Raise HashTableOverflow
—If it is a key duplication
——overwrite the entry and RETURN
—Move original entry to the first free entry
—If the moved entry was a head
——Put in the new chain head of the old chain
——Fix auxiliary structure
—Else
——put in the new head of a new chain
——Fix auxiliary structure
RETURN
```

(c)ii

0			((2,C),2,null)	((5,Z),null,null)
1				((2,C),2,null)
2	((2,A),null,null)	((2,C),null,null)	((12,T),null,0)	((12,T),null,1)
3				
4				