CST Part IA: Algorithm, SV 3
Joe Yan
2017-2-17

1 2007P11Q9

(a)
- Every node other than root has at least $t-1$ keys and thus at least t children. For all non-empty B-tree, the root must have at least one element.
- Every node has at most $2t-1$ keys and thus at most $2t$ children.
- All leaves have the same depth h which is the depth of the B-tree.
- Every node should have a method or auxiliary boolean value to determine whether it is a leaf which means it has no child and at the depth of the tree's height h and also an integer value which shows the number of keys currently stored in the node.
- Every node except leaves should have $k+1$ children if they have k keys.
- $\forall 0 < k < t-1$. $key_k \leq elements\ in\ child_{k+1} \leq key_{k+1}$
- $\forall 0 < k < t$. $elements\ in\ child_k \leq key_k \leq elements\ in\ child_{k+1}$

(b) Define the $split(x, i)$ method which will split the child with index i of node x. This method will only be called when such child is full and the node x is not full.
Assuming the minimum degree of the B-tree is t.The index of the middle key will be t-1. Store the middle key and split the child to two nodes which respectively have keys with index from 0 to $t-2$ and from t to $2t-2$ and children with index from 0 to $t-1$and from$t$ to $2t-1$. Then push the middle key to the node s at the position with index i. This will move the origin keys and children with index larger or equal to i to right by 1 index and put the middle key to the position with index i in the key-array and assign the previous created two new child nodes to positions with i and $i+1$ in the child-array. Note split method creates two nodes with t-1 keys and t children and adds one more keys and two more children to the non-full node x which holds the minimum degree of B-tree. When we insert a key to the B-tree. First, check whether the root is full. Root has no parent so we create a new one-key node as new root and do similarly as the split method. Create a node reference p assigning to root then iterate as following.

If p is a leaf, put the key to the right position and the insertion is done.
Else p is not a leaf. Search which child the new key should be then do following.
—If the child is full call method $split(p, i)$ where i is the index of the child the new key should be, remain the reference p and return to the start of the iteration.
—Else the child is not full, reassign the reference p to this child and return to the start of the iteration.
-C
-AC
-ACM
Insert B
- C
-A M
- C
-AB M
-Insert R
- C
-AB MR
Insert I

- C
-AB IMR
Insert D
- C-M
-AB I R
- C—M
-AB DI R
Insert G
- C——M
-AB DGI R
Insert E
- C–G-M
-AB D I R
- C—G-M
-AB DE I R
Insert X
- C—G-M
-AB DE I RX

(c) Here the "bottom node" is leaf.
Assume looking for the successor for key k with index i in the node x and there is no duplication in the B-tree.
If the key k is in a leaf. Done.
Else the key k is not in a leaf.
Claim the successor must be in a leaf.
Proof:
The successor cannot be in the sub-tree of x with index smaller than $i+1$ because they contain elements smaller than the k.
The successor cannot be in the sub-tree of x with index larger than $i+1$ because all keys in such sub-tree will be larger than keys in sub-tree of x with index i which reverse the definition of successor such that successor is the smallest element larger than k.
The successor cannot be in the parents of the child because for all keys in the parent smaller we do not consider. For keys in the parent larger than k they must larger than all keys in the sub-tree of node x so for the same reason they are too large and reverse the definition of successor.
All other situations is impossible to find the successor of k. So the successor of k must be in the sub-tree of x with index $i+1$. Looking for the successor means looking for the minimum in such sub-tree. And the minimum must be the most left key in such sub-tree which means it is in a leaf. Done.
Overall for all cases, either the key or its successor is in a bottom node.

(d1) Define several operations on the B-tree that conserves the property of B-tree.

Merge If the sum of numbers of keys in two adjacent brother nodes plus one is smaller or equal to the maximum allowed number of keys in a node. Remove the key between two brother nodes in their parents and form a new node by merging these three parts as a new child of the parent. Basically, this is the inverse of spilt.

Split Discussed in (b).

Redistribute Redistribute keys in the adjacent children with a change of the key between these two children in the parents. Meanwhile move the child either if targets of the redistribution are not leaves. The number of keys in the parent remains constant.

Assume k is the key going to be deleted.
If k is in a leaf node
—If k can lose a key without being too small then remove the key. Done.

—Else k can not lose a key without being too small.
——If k has an adjacent brother that can lose a key without being too small then redistribute one key from the brother to the node with key k and remove k.
——Else if k has no such adjacent brother which means it must satisfy the postcondition of the merge. We merge them with the number of keys in the parent reducing one. If the parents get too small, run this process for parents also to refill the parents to converse the property of B-tree.(That is a recursion here.) Finally, remove k in the leaf node.
Else k is not in a leaf node. As proved in (c), either k or it successor will be in a leaf node. Swap k and its successor then run the process for if k is a leaf node. Done.

(d2) Delete M

```
-        J————————————-V
-    C——G       M——Q—T    Y
- AB DEF HI KL NOP R U X Z
-        J————————————-V
-    C——G       N——Q—T    Y
- AB DEF HI KL MOP R U X Z
-        J————————————V
-    C——G       N—Q—T    Y
- AB DEF HI KL OP R U X Z
```

Delete Q

```
-        J————————————V
-    C——G       N—R—T    Y
- AB DEF HI KL OP Q U X Z
-        J————————————V
-    C——G       N—P—T    Y
- AB DEF HI KL O RQ U X Z
-        J————————————V
-    C——G       N—P-T    Y
- AB DEF HI KL O R U X Z
```

Delete Y

```
-        J————————————V
-    C——G       N—P-T    Z
- AB DEF HI KL O R U X Y
-        J————————————-T
-    C——G       N—P    V-Z
- AB DEF HI KL O R U X Y
-        J————————————-T
-    C——G       N—P    V
- AB DEF HI KL O R U XZY
-        J————————————-T
-    C——G       N—P    V
- AB DEF HI KL O R U XZ
```

2 2006P3Q2

(a) It is not a binary search tree. Because S is in the left sub-tree of R, but it is larger than R which reverses the property of binary search tree.

(b) Assume looking for the predecessor of key k in node x.
Assume there is no duplication of keys in the tree.
Claim: If there is a left sub-tree for x then the predecessor must be in it and a maximum of the sub-tree. Otherwise, going up to the root from x, predecessor will be in the first

node with key smaller than k. That is the first time "going left" to find its parent.(i.e. The first super-node of x where x is in its right sub-tree.)
Proof:
If there is a left sub-tree
—If the predecessor of x is not in its sub-tree.
——If x is in some left sub-tree of its super-tree then all keys in that super-tree except the branch going to x cannot be the predecessor because they are all larger than k.
——Else if x is in some right sub-tree of its super-tree then all keys in that super-tree except the branch going to x cannot be the predecessor because they are all smaller than all keys in the sub-tree of x but there is a left sub-tree for x. This reverses the property that the predecessor of k is the largest key smaller than k.
—Else if the predecessor of x is in its sub-tree then it must be the maximum of left sub-tree which is the maximum element smaller than k. Done.
Else there is no left sub-tree. As argued before, the remaining maximum choice can be found by the following process: going up to the root from x, predecessor will be in the first node with key smaller than k.

The proof of claim is the idea of how the algorithm works.
If there is a left sub-tree of x, then the predecessor of k will be the maximum of such left sub-tree.
Else the predecessor will be in the first super node of x where x is in its right sub-tree.)
Otherwise k is the minimum of the binary search tree who has no predecessor.

(c) Because neither of d's sub-tree is empty the predecessor must be the maximum of the left sub-tree of d.
Rewrite the key in d as its predecessor and remove the node the predecessor in.
Claim the predecessor node of the key in d has no right sub-tree.
Proof: If there is a right sub-tree then the maximum of the left sub-tree of d must be in it. Contradiction. So the claim must be true.
As there are no right sub-tree, link the left sub-tree (if there is one) to the parent of the node originally contains the predecessor of the key in node d. Done.
Claim it is still a BST.
proof: First we rewrite the key in d as the predecessor of it. BST property holds because the predecessor is in the left sub-tree of d so smaller than all keys in the right sub-tree of d and it is larger than all the keys in the left sub-tree of d as it is the maximum of it. Then we link the left sub-tree (if there is one) to the parent of the node originally contains the predecessor of the key in node d. BST property holds because all keys in such left sub-tree smaller than its grandparent as it is a child of the parent which is a right child by how we find the predecessor. (i.e. find the maximum) Done.

(d) Prove by contradiction here.
Assume the k_p is neither the smallest key greater than k_l nor the largest key smaller than k_l.
Without lost of generality, assume l is the left child of p. That is $k_l < k_p$ and assuming k_p is not the smallest key greater than k_l.(i.e. $\exists m.\ k_l < k_m < k_p$) Otherwise we reach contradiction directly.
If k_m is in the right sub-tree. Contradiction by $k_m > k_p$.
Else if p and l are in the left sub-tree of any super-tree. Then m can not be in the root or the right sub-tree of that super-tree. Contradiction by $k_m > k_p$.
Else p and l are in the right sub-tree of any super-tree. Then m can not be in the root or the left sub-tree of that super-tree. Contradiction by $k_m < k_l$.
For the previous super-tree, if consider a path from p to the root of the whole BST and let all node in such path be the root of such super-tree then it covers all nodes in the whole BST other than p,l. Done.

By contradiction, the k_p is either the smallest key greater than k_l or the largest key smaller than k_l.