Splay trees

* A splay tree is a self-adjusting BST with the property that recently accessed node splaying to root.
  + Self adjusting is to make recent accessed node the root of the tree by a sequence of rotation (also called splaying)
* In a splay tree, search, insert and delete operations are first done as BST operations, then followed by combined splaying to make the relevant node root of the tree.
* Why splay tree? The practical principle: if a data element is accessed, it is likely that it will be accessed again. Since the most frequently accessed nodes are moved closer to the root node, these nodes can be accessed faster.

**Amortized analysis**

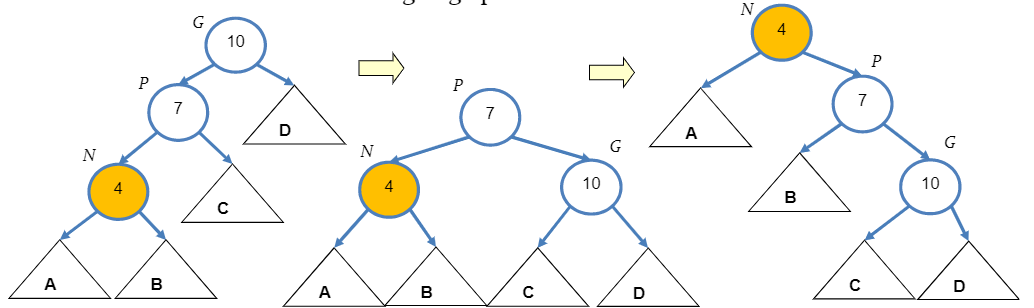
Worst case analysis: Splay trees are not balanced BSTs, worst case time per operation is O(n)

* Amortized analysis is to value to worst case average resource usage of an algorithm in a sequence of runs. If a sequence of M operations takes O(M f(n)) time, we say the amortized runtime is O(f(n))
* For splay trees, amortized time per operation for search, insertion and deletion is O(log n)

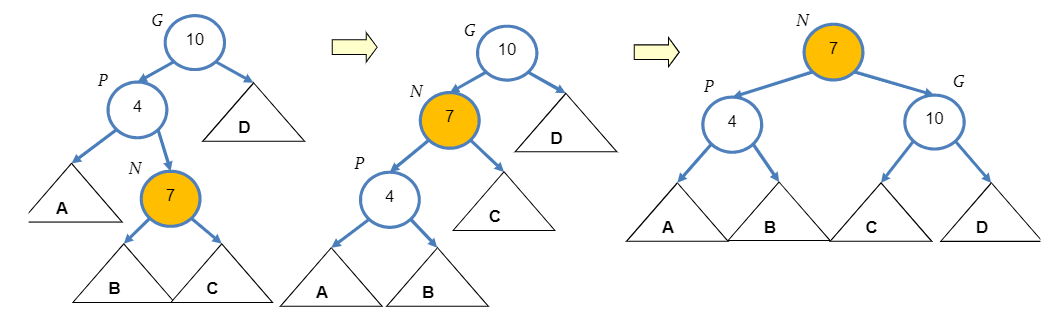
**Splay tree operations**

* Last accessed node can be moved to root by a sequence of left or right rotations in many ways.
* Same as BST, we consider the search, insert, delete operations on splay tree
* The principle of these operations is do the corresponding BST operations and flowed by splaying operations
* Splay operations move a node up to root by a sequence of zig-zig, zig-zag, and zig operations (in bottom-up or top-down style)
  + Bottom-up style requires a stack for backtracking to root, O(height)
  + Top-down style does not require stack, use iteration and constant number of variables. Space O(1)

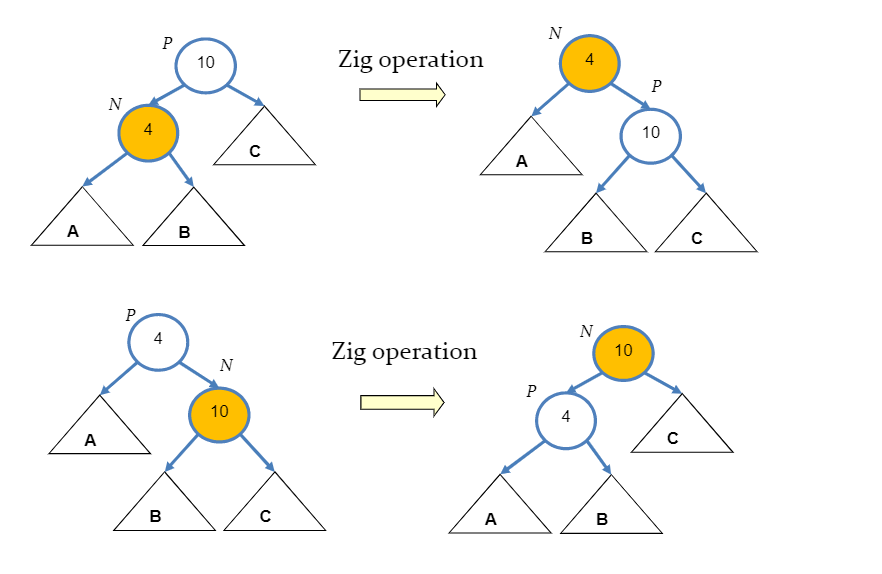
**Zig-zig operation: (right-right rotations or left-left rotations)**



**Zig-zag operation (left-right or right-left rotations)**

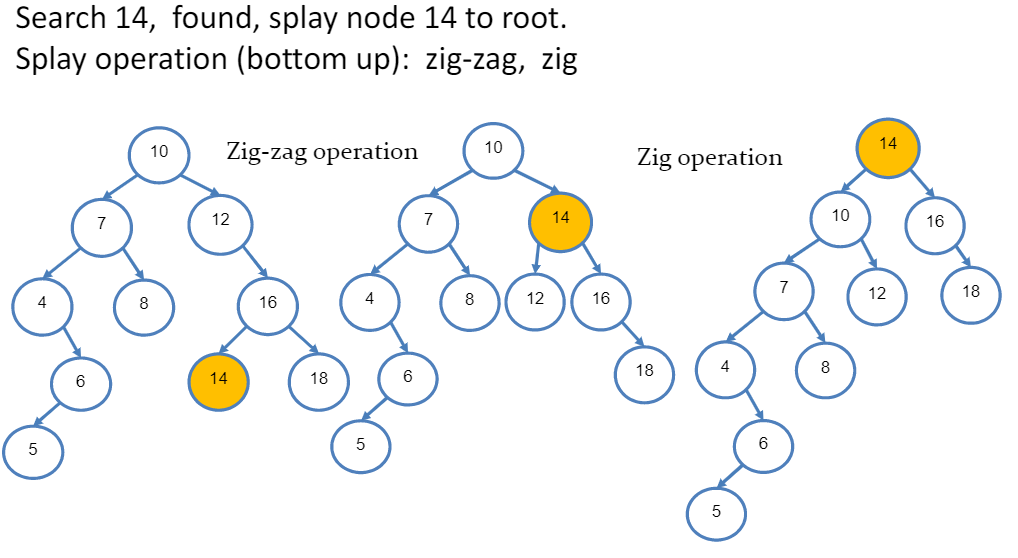


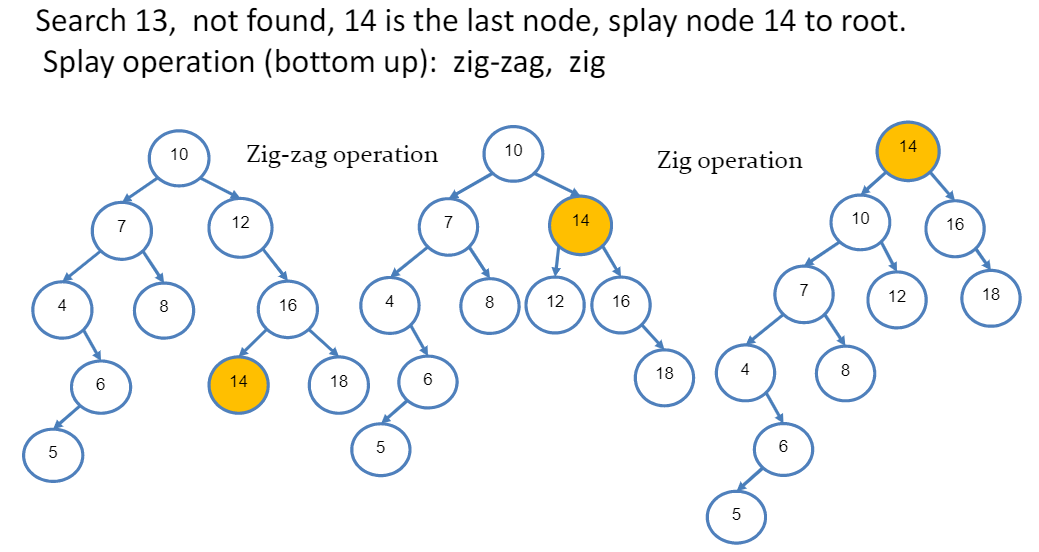
**Zig operation is just a single right or left rotation**



Same as BST, we consider the search, insert and delete operations on splay tree. The principle of these operations is do the corresponding BST operations and followed by splaying operations.

**Example of splay tree search**

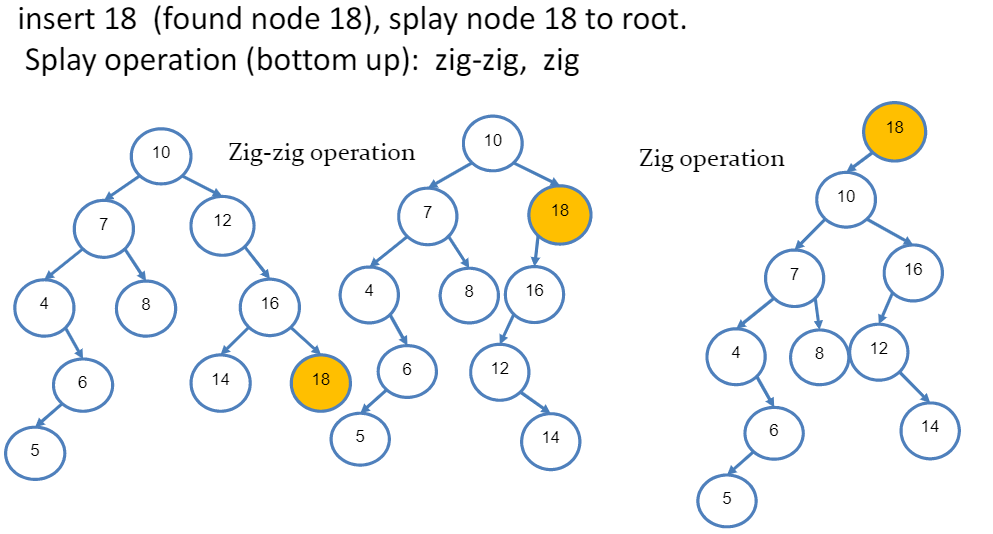




**Splay tree insertion**

Algorithm:

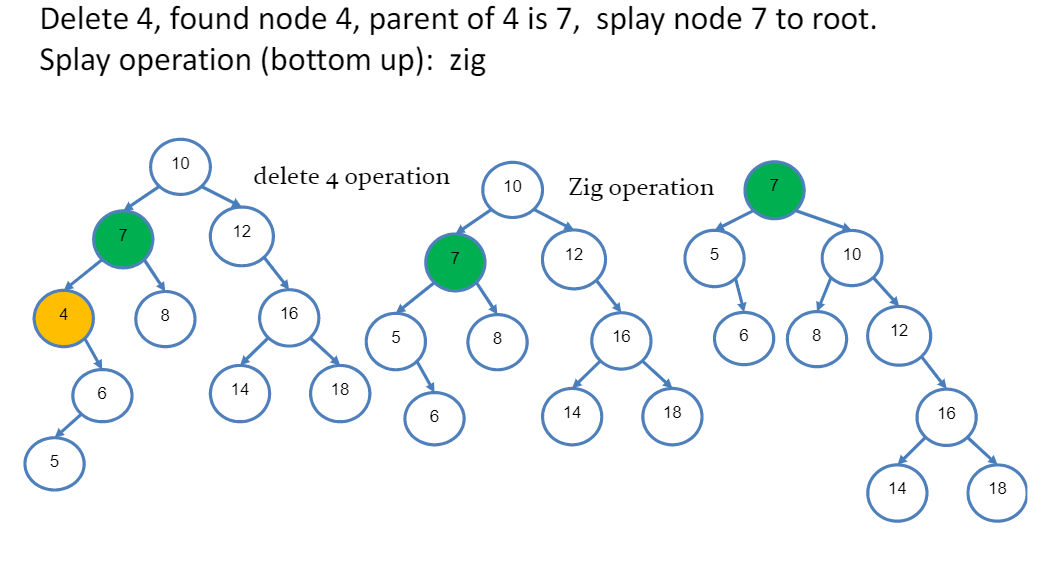
1. Do BST search for the key value to be inserted. If found N, splay N to root
2. Otherwise, do BST insertion
3. Let N be the new inserted node, splay N to root



**Splay tree deletion**

Algorithm:

1. Search for the node N to be deleted
2. If not found, splay the last accessed node P
3. If found, let PP be the parent of N. Replace N by an appropriate descendant of P. If N is not root, splay the tree at P.

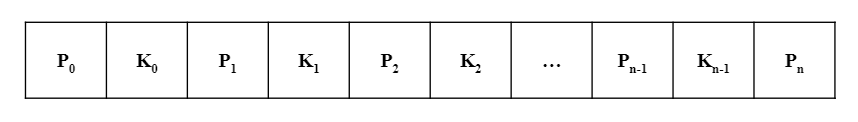


**Splay tree summary**

1. Splay trees are efficient to access frequently accessed keys
2. Splaying can be done in bottom up or top down styles
3. Splay tree implementation does not need extra field
4. All operations are in amortized O(log n) time.

**Multiway Search Tree**

* M-way (m-ary) search tree is generalization of binary search tree, allowing at most m-1 keys and m children



* In the node structure P0, P1, P2,…Pn are pointers pointing to the root of subtrees of the node, and K0, K1, K2, …, Kn-1 are the key values of the node.
  + Node has n key values, and n+1 pointers, n+1 <=m
  + All the key values are stored in ascending order:
    - Ki < Ki+1 for 0<=i<=n-2
  + All key values in subtree pointed by Pi are less then Ki
  + All key values in subtree pointed by Pi+1 are bigger than Ki
* M is the maximum number of subtrees (children)
  + The out-degree of an m-way search tree is at most m
* M-1 is the upper limit that defined how many key values can be stored in the node
* M-way search tree is a generalization of binary search tree i.e. when m=2, a 2-way search tree is a BST

**Node structure**

Typedef struct node{

Int data;

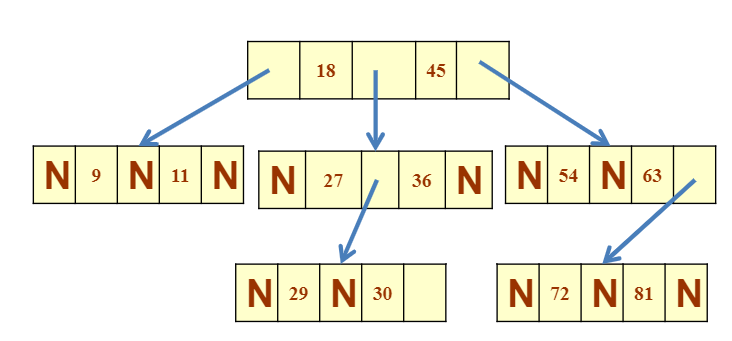
Int count; //number of key values

Int key[m-1]; //key arrays, one more for operation

Struct node \*child[m]; subtree pointer array

} tnode;

**Example of 3-way search tree**



**MWST traversal**

Similar to the binary tree traversal, we can traverse MWST in depth-first order (in-order) or breadth first order

/\*in-order traversal of multi-way tree\*/

Void print\_inorder(tnode \*root){

If(root!=NULL){

Int I;

For(i=0;i<root->count;i++){

Printf(“%d”, root->key[i]);

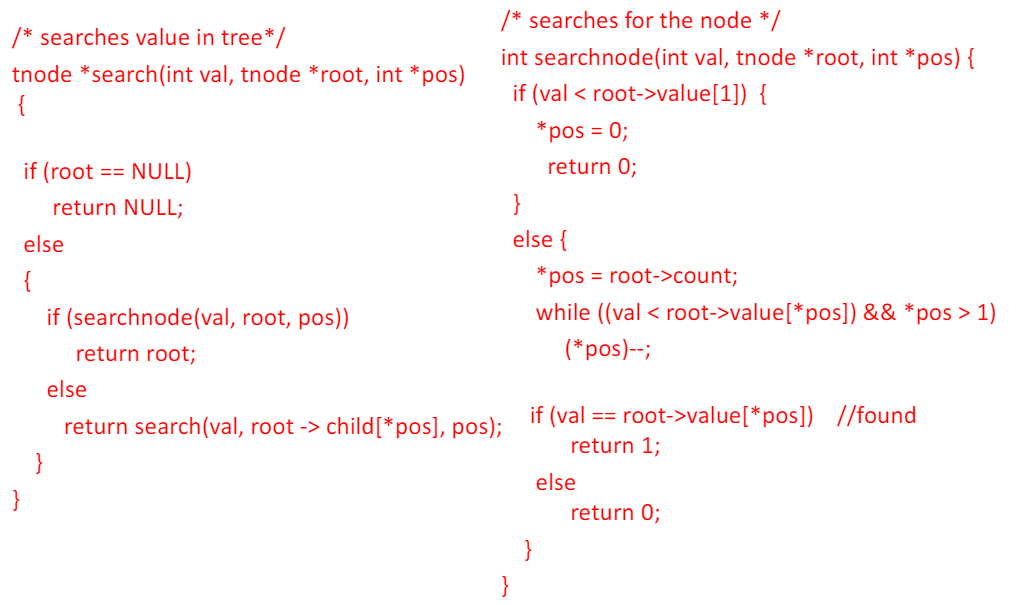
Print\_inorder(child[i+1]);

}

}

}

**MWST search**

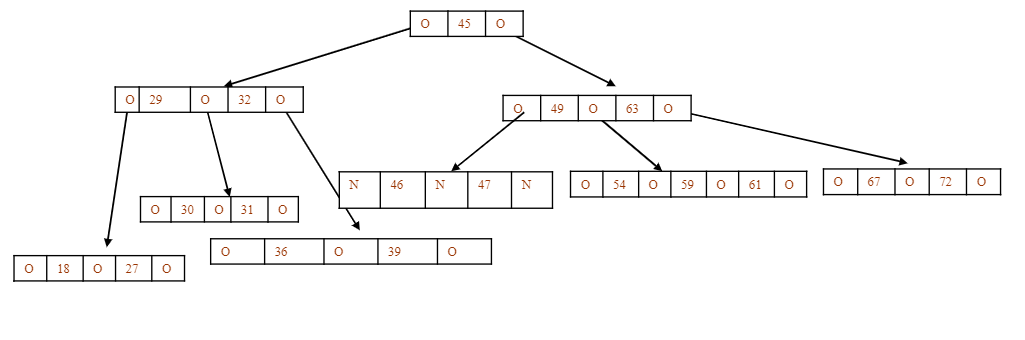


**B Trees**

A B-tree of order m is a self-balanced m-way MWST with the following properties

1. Every node has at most m children
2. Every node in the B-tree except the root node and leaf nodes have at least (minimum) m/2 children
3. The root node has at least two children if it is not a leaf node
4. All leaf nodes are at the same level

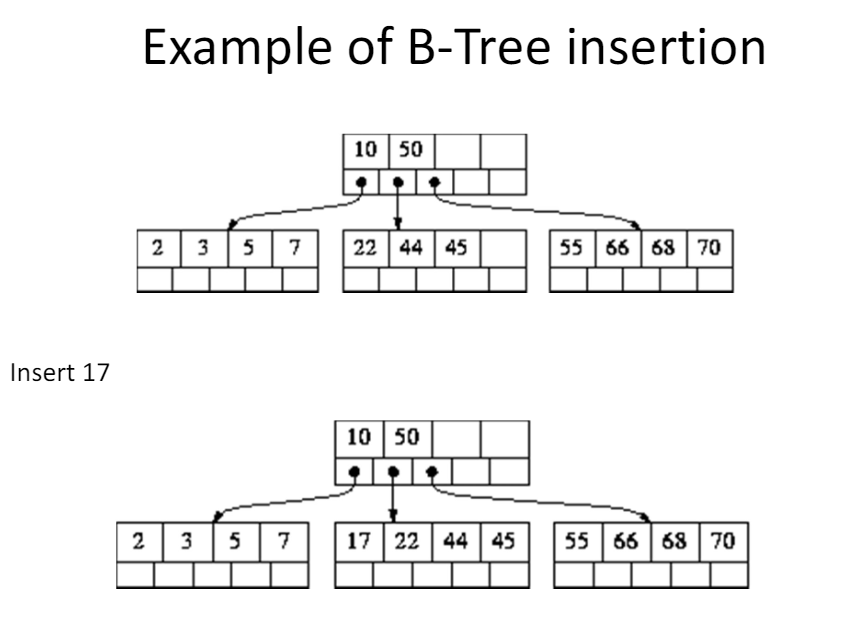
**Example of B-tree with m=4**

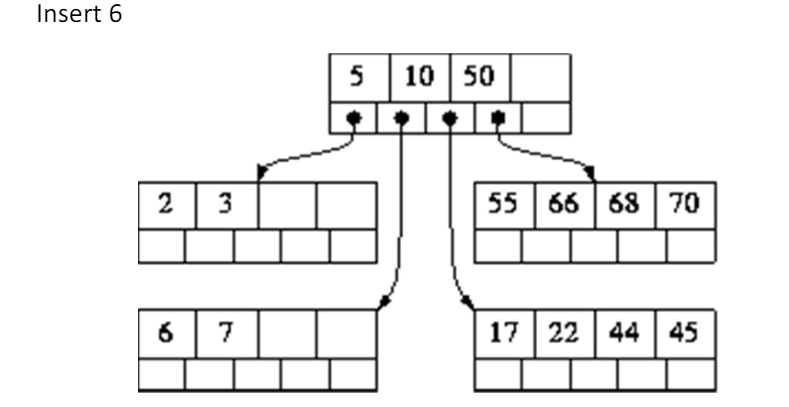


**Insertion operation on B-trees**

Algorithm:

1. Search the B tree to find the leaf node where the new key value should be inserted
2. If the node is not full, that is it contains less than m-1 key values then insert the new element in the node, keeping the node’s element ordered
3. If the node is full, insert the new value in order into the existing set of keys, split the node at its median into two nodes. Note that the split nodes are half full. Insert the median element to is parent node. Goto step 2.





**B-Tree deletion**

Algorithm:

1. Locate the lead node which has to be deleted
2. If the leaf node contains more than minimum number of key values (more than m/2 elements), then delete the value.
3. Else if the leaf node does not contain even m/2 elements then, fill the node by taking an element either from the left or from the right sibling
   1. If the left sibling has more than the minimum number of key values (elements), push its largest key into its parent’s node and pull down the intervening element from the parent node to the leaf node where the key is deleted.
   2. Else if the right sibling has more than the minimum number of key values (elements), push its smallest key into its parent node and pull down the intervening element from the parent node to the leaf node where the key is deleted.
4. Else if both left and right siblings contain only minimum number of elements, then create a new leaf node by combining the two leaf nodes and the intervening element of the parent node (ensuring that the number of elements do not exceed the maximum number of elements a node can have, that is, m). If pulling the intervening element from the parent node leaves it with less than minimum number of keys in the node, then propagate the process upwards thereby reducing the height of the B tree.

* To delete an internal node, promote the successor or predecessor of the key to be deleted to occupy the position of the deleted key. This predecessor or successor will always be in the leaf node. So further the processing will be done as if a value from the leaf node has been deleted.