Karan Khosla(91725095) CS261P - Data Structures

Binary Search Tree Alternatives Analysis

Individual Analysis of Trees

1. Binary Search Tree(unbalanced)(duplicates allowed)

Theoretical analysis

a) Left subtree contains nodes whole key value is less than root's key value
b) Right subtree contains nodes whole key value is greater than or equal to root's key value
c) Useful if insert and delete operations need to be performed in sorted data.
d) All operations take almost the same time.

Pros

a) Easy to implement and works
b) All operations take O(lgn) average case. (Insert,Search,Delete,Successor,Predecessor,Find_Min,Find_Max etc.)
c) Beats binary heap DS in search, delete(not delete_min) operation times.
d) Beat sorted array DS in insert,delete operation times.(O(n) for sorted array)

Cons

a) Worst case time can be bad (O(n))(when input is sorted)
b) Takes up more space than sorted array and binary heap DS.(because of pointers)
c) Search is guaranteed O(lgn) in sorted array, but in BST may be O(n).

Pseudo-Code

```
searchKey(k)
{
    while(node exists)
    {
        if(node->key == k)
            return node;
        elif(node->key > k)
            return search(node->left,k);
        else
            return search(node->right,k);
    }
    return NULL;
}
```

```
insertKey(k)
{
    if(k < node->key)
    {
        if(node->left does not exist)
            Insert new node with key;
        Else
            insertKey(node->left,k);
    }
    else
    {
        if(node->right does not exist)
            Insert new node with key
        else
            insertKey(node->right,key);
    }
}
deleteKey(k)
{
    if(node == NULL)
        return;         //If key does not exist
    if(node->key == k)                //Found the key to delete
    {
        If node has at most one child
            Connect the parent of node with that child
        else if node has two children
        {
            Let x = inorder successor of node.
            Copy contents of x into node.
            deleteKey(node->right,x->key);.
        }
    }
    else if(k < node->key)
        deletKey(node->left,k);
    else
        deleteKey(node->right,k);
}
```

2. AVL tree(duplicates allowed)

Theoretical analysis

a) All operations ensure a O(lgn) worst case time bound.
b) All operations take almost the same time.
c) Requires rotations(left and right) to balance the tree.
d) Difficult to implement as we have to traverse back up the tree to balance it using rotations(4 cases - left-left, left-right,right-left,right-right heavy)

Pros

a) All operations take O(lgn) **worst** case. As compared to O(n) worst case for BST (Insert,Search,Delete,Successor,Predecessor,Find_Min,Find_Max etc.)
b) Usually very good for searching keys as compared to standard BST.
c) Beats binary heap DS in search, delete(not delete_min) operation times.
d) Beat sorted array DS in insert,delete operation times.(O(n) for sorted array)

Cons

a) Difficult to implement because of rotations
b) Takes up more space than BST because each node has height variable.
c) Takes up more space than sorted array and binary heap DS.(because of pointers)

Pseudo-Code

```
searchKey(k)                                  // Same as BST
{
        node = root
        while node != null
                // compare k with node
                if k < node.value: node = left child
                else if k > node.value: node = right child
                else: return node
        k is not in data
}
insertKey(k)
{
        Do standard BST insertion.            //Node gets inserted at leaf.
        Follow a path from the leaf back up to the root, updating heights
        If a node has become unbalanced, do a rotation to rebalance it.
        //Case analysis shows: never need more than two rotations.
}
deleteKey(k)
{
        Do standard BST deletion.  //Node deleted in the end is one with not two children.
        Follow a path from the leaf back up to the root, updating heights
        If a node has become unbalanced, do a rotation to rebalance it.
        //Case analysis shows: never need more than two rotations.

}
```

3. Treap(duplicates allowed)

Theoretical analysis

a) Randomized alternative to BST.
b) Gets its name from Tree + Heaps.
c) Each node has priority value with key value.
d) Organized as : BST w.r.t. Key values and Max-Heap w.r.t. priorities.
e) All operations occur in O(lgn) **average** case time.
f) Requires rotations(left and right) to balance the tree.

Pros

a) Easier than AVL tree to implement as requires rotations to heapify the priority.
b) Very few rotations(expected <= 2) as compared to AVL tree(lgn)
c) Worst case is pretty rare as compared to standard BST.
d) Usually pretty good for searching keys as compared to standard BST.
e) Beats binary heap DS in search, delete(not delete_min) operation times.
f) Beat sorted array DS in insert,delete operation times.(O(n) for sorted array)

Cons

a) Worst case for an operation can be O(n).
d) Difficult to implement because of rotations.
e) Takes up more space than BST because each node has priority variable.
f) Takes up more space than sorted array and binary heap DS.(because of pointers)

Pseudo-Code

```
searchKey(k)
{
    //same as BST search
}
insertKey(k)
{
    v = Insert k using standard binary tree insertion // v is a leaf node
    v.priority = random()
    while (v is not the root and v.priority > v.parent.priority):
        if v is a left child: rotate right(v.parent)
        else: rotate left(v.parent)
}
deleteKey(k)
{
    while v is not a leaf:
         if left(v) == None: rotate left(v)
        elif (right(v) == None or v.left.priority > v.right.priority): rotate right(v)
        else: rotate left(v)
    Delete the leaf node v
}
```

4. Splay Trees(no duplicates allowed)

Reasons why this algorithm is chosen

a) Practically used in many applications(Windows NT, GCC compiler, GNU C++ library) etc.
b) Use locality of reference property. (usually 80 % accesses are to 20% of items.)
c) Outperforms other BST alternatives in many scenarios due to b)
d) In many cases provides O(1) search due to frequently access elements.
e) Dynamic Optimality Conjecture: Competitive ratio is O(1).

Theoretical analysis

a) Basic operation is splay. Splay a node means taking it to the root by using rotations.
b) Main idea of splay tree is to bring the recently accessed item to root.
c) 3 cases decide what sort of rotations-:
    1. x has no grandparent (zig)
    2. x is LL or RR grandchild (zig-zig)
    3. x is LR or RL grandchild (zig-zag)

d) Amortized(not worst case) time bound on operations.

Pros

a) Outperforms other BST alternatives in many scenarios due to locality of reference.
b) In many cases provides O(1) search due to frequently access elements.
c) Practically used in many applications(Windows NT, GCC compiler, GNU C++ library) etc.

Cons

a) Difficult to implement as each operation has to splay the recently used key to root.
b) Even search requires splaying and changing the tree(as compared to other BST alt.)
c) Performs very bad if data is non-repetitive.
d) Worst case time bound is O(n).(AVL has worst case O(lgn)).

Pseudo-code

```
splay(node ,key)
{
    //traverses the tree trying to find node with key k.
    // if found, then uses rotations based on three cases above to bring node to top
    //if not found, then brings last accessed node to the top using rotations.
}
searchKey(k)
    return splay(node,k);  //splays the node with key k or closest to the root.
insertKey(k)
{
    if(!root)
        return new Node(k);                // make new root
    Node n = splay(root,k);
    if(n->key == k)
        Return node;                       //if key k already exists, do nothing
    else
        if(k < n->key)
        {
                Make new node with key k as root;
                new Node’s right subtree is n;
                New Node’s left subtree is n->left subtree;
                n->left = NULL;
        }
        Else
        {
                Make new node with key k as root;
                New Node’s left subtree is n;
                New node’s right subtree is n->right subtree;
                n->right = NULL;
        }
    Return new Node;
}
```

}

deleteKey(k)

{

If Root is NULL: We simply return the root.

1. Else splay the given key k. If k is present, then it becomes the new root. If not present, then last accessed leaf node becomes the new root.
2. If new root’s key is not same as k, then return the root as k is not present.
3. Else the key k is present.
    - Split the tree into two trees Tree1 = root’s left subtree and Tree2 = root’s right subtree and delete the root node.
    - Let the root’s of Tree1 and Tree2 be Root1 and Root2 respectively.
    - If Root1 is NULL: Return Root2.
    - Else, Splay the maximum node (node having the maximum value) of Tree1.
    - After the Splay procedure, make Root2 as the right child of Root1 and return Root1.

}

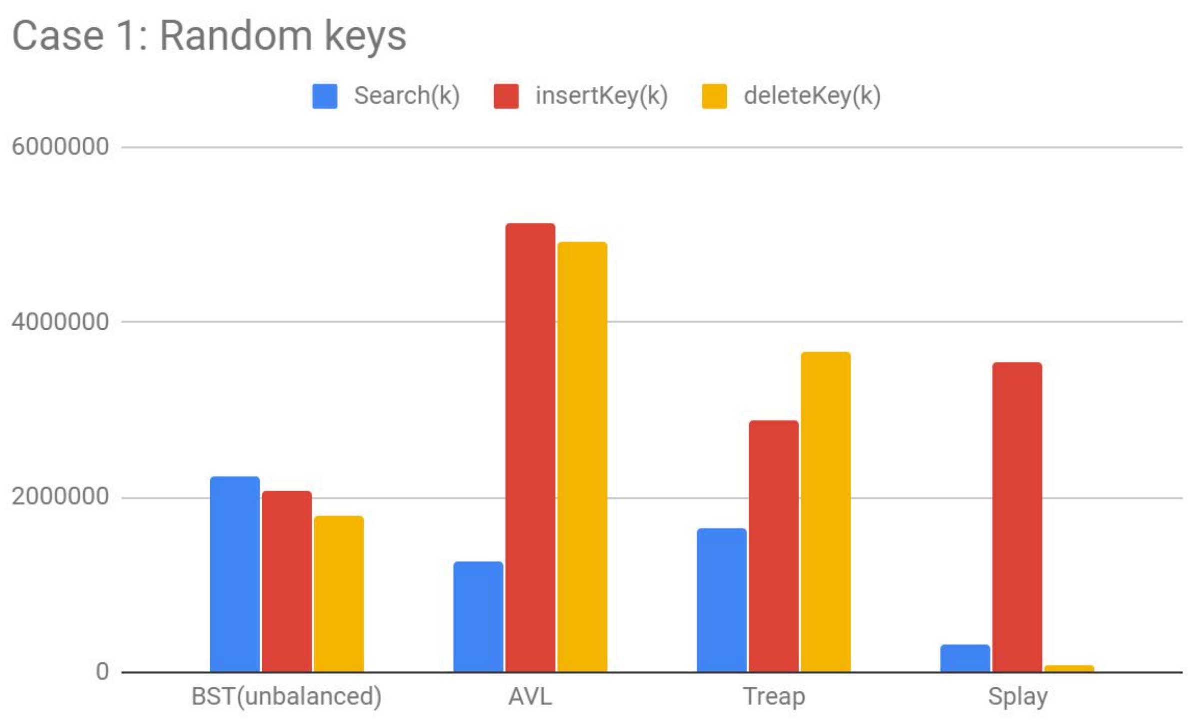
2. Comparative Analysis of BST alternatives

Here we analyze the various performances of the 4 above mentioned BST alternatives based on various inputs and see how the individual operations of the 4 trees perform.
We are using 10000 keys in all cases, only the ordering of keys is different.

Case 1(Random keys): 10000 random keys are inserted, searched and deleted from the trees. This is just to set the reference for the future test cases and see how they perform as compared to this base case. It itself is not very helpful when it comes to analyzing these 4 trees.

(All times are in nanoseconds)

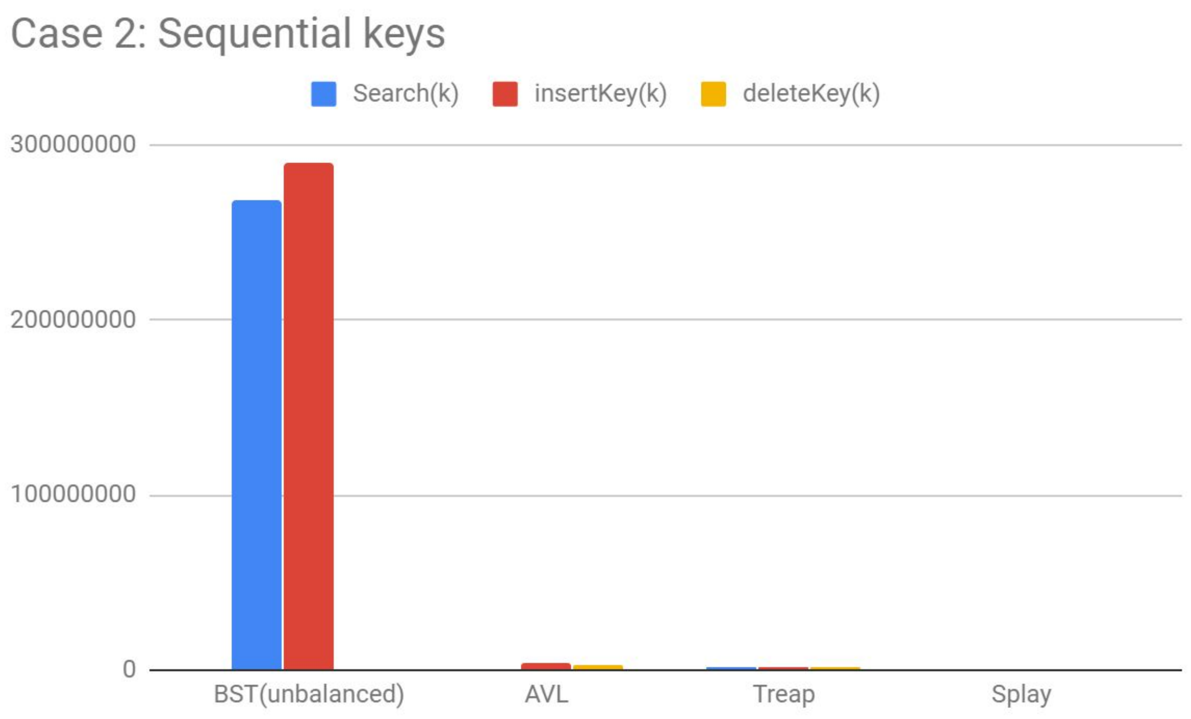
Operations -> Trees |	Search(k)	insertKey(k)	deleteKey(k)
BST(unbalanced)	2239118	2083261	1790941
AVL tree	1276826	5126347	4923944
Treap	1649543	2885112	3653112
Splay tree	320881	3543096	72287



- Even though, keys are randomized, we still see the splay tree has the best performance when it comes to search and delete because of locality of reference.
- Most consistent performer with all operations taking almost the same time is BST.
- AVL is the worst performer in this case because of the number of rotations(lgn) that need to be done each time an item is inserted or deleted.
- Treap performs better than AVL as the number of rotations is less than AVL.

Case 2(Sequential keys) : The keys to be inserted in this case are 1-10000 in sorted order. We use this case to show the shortcomings of BST(because of unbalanced, so all keys are inserted linearly and worst case is achieved at O(n)) and how the alternatives are useful in this case. Let's see how they perform.

Operations-> Trees ||	Search(k)	insertKey(k)	deleteKey(k)
BST(unbalanced)	268020669	289122682	346622
AVL	908342	4366104	3060716
Treap	1461245	1453487	1264132
Splay trees	486964	586755	111074

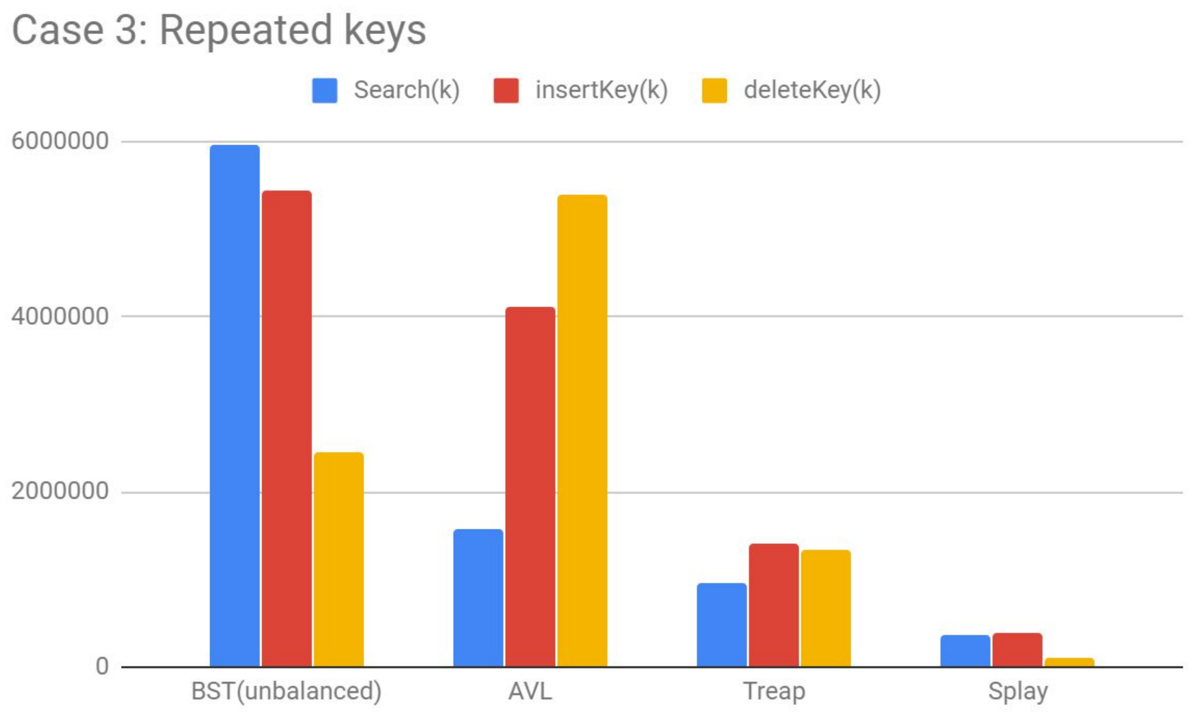


-As we can see from the graph that BST search and insert operations perform terribly(O(n)) if data is sorted(as expected). Delete is good because always the root is deleted.
-AVL tree performance does not really depend on what is the ordering in the data.
- Treap also performs well in this case.
- Splay trees performs the best as expected, this is why it is used most often in practical applications mentioned above.

So, from the above, we can conclude that if data is ordered or nearly ordered, don't use a standard BST. Best alternative is splay tree.

Case 3: Repeated keys
The keys in this set are repeated to a great extent, as is usually the case in most applications due to locality of reference(80% of accesses are to 20% items usually). We have 100 distinct keys that are repeated to reach the 10000 keys point.

Operations-> Trees ||	Search(k)	insertKey(k)	deleteKey(k)
BST(unbalanced)	5952171	5426070	2451360
AVL	1563482	4117861	5395393
Treap	964187	1416848	1336198
Splay trees	358049	396628	105432



- BST is pretty bad due to repeated keys and no balancing done.
- AVL is also not the ideal candidate when there is repetition in the keys.
- Treaps have decent performance due to less number of rotations.
- Splay trees outperform the others easily to locality of reference.

In this case, the clear winner is splay tree.

Conclusion

Usually, the choice of tree depends on what kind of data is used. But there are some cases where some trees should be avoided or used as follows-:

1. If data is random to a great extent, then we can use BST or splay trees depending on the number of operations that are to be performed. If the number of inserts is a lot, then prefer BST, Otherwise if there are loads of search and deletes as compared to inserted, then pick splay trees.
2. If data is ordered or nearly ordered, then BST should be avoided at all costs, as it performs terribly in this case. A good alternative is splay trees as it performs the best relatively.
3. If data is repetitive, then splay tree easily outperforms other trees in all operations. So, splay trees should be used.

From the above, we can see that splay trees are great to use, irrespective of the data or number of each type of operations to be performed in most cases.
This is why they are found in many practical applications and softwares like Windows NT, GNU GCC compiler, C++ library etc.