Binary Search Trees

R. K. Ghosh

IIT Kanpur

Binary Search Trees

Binary Search Trees

Definition

It is a symmetric ordered binary tree is such that for each node

1. All the values stored in the left subtree are less than or equal (allows duplicates) to the value stored at the node.
2. All the values stored in the right subtree are greater than the value stored at the node.

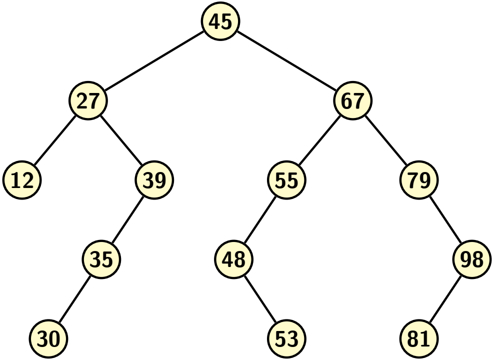
Searching is the Main Operation

- BST is an extension of a basic binary tree having a special attribute called key associated with the information stored each node.
- The main operation in a BST is membership search.
- The value of the key uniquely idenfies a node.
- Besides search there are other binary tree operations: **delete()**, **insert()**, **deleteMax()**, **deleteMin()**

Use of BST

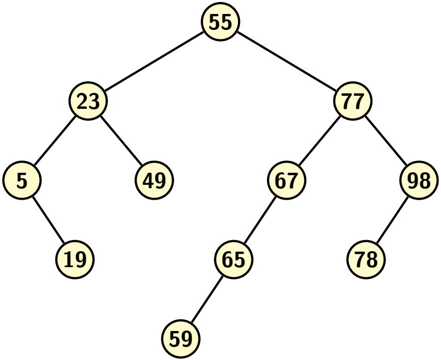
- Used where data enter and leave in random order in a regular basis (in a dynamic environment).
- Still, one could argue non balanced BSTs are of theoretical interest.
- In an average BSTs perform pretty well because data arrival is in random order.
- It forms the basis of balanced binary trees which are generally used for dictionary applications.
- Dictionary is a data structure for implementation of key-value kind store.
  - More precisely, a key is associated with each value.
  - Given a key, retrieve, store, or delete the data from store.

Example 1



- BST property is preserved at each node.

Example 2



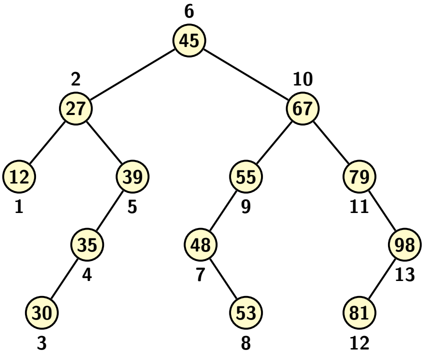
- BST property is preserved at each node.

Some Observations

- The leftmost node in a BST has the minimum key.
- The rightmost node has the maximum key.
- Membership search for a key k performed as follows:
  - If tree is empty then report ”NO” (k is not present), otherwise start at the root.
  - Compare the value stored at the root of the (sub)tree.
  - If key k_r at the root equal to k then report ”YES”.
  - If $k_r < k$ then recursively search right subtree.
  - Else if $k_r > k$ then recursively search left subtree.

Observations Regarding Traversal BST

- In order traversal of a BST produces the sorted list of keys.

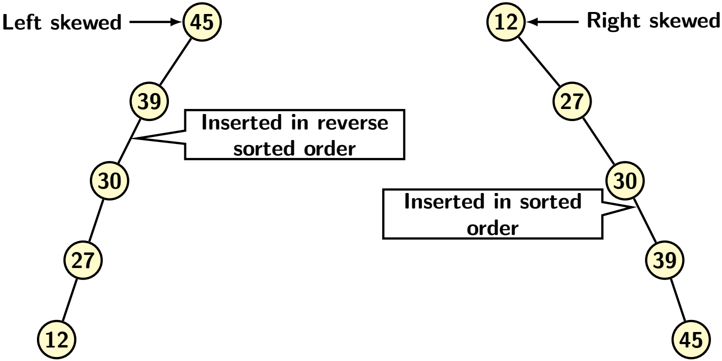


Inorder list: 12, 27, 30, 35, 39, 45, 48, 53, 55, 67, 79, 81, 98.

Observations Regarding Building BST

- If keys arrive in ascending order then a right skewed BST results.
- Similarly, if insertions are performed in the reverse sorted order then a left skewed BST is obtained.
- However, when the insertions are performed randomly then most likely the tree would be balanced.
- Membership search is fast unless you have a left skewed or a right skewed BST.

Left/Right Skewed BST



Important Operations on BST

- **makeNull()**: Creates T as an empty BST.
- **isEmpty**(): Returns true of BST is empty.
- **insert**(x): Insert x into T.
- **delete**(x): Delete x from T.
- **deleteMin**(): Removes minimum element from T.
- **deleteMax**(): Removes maximum element from T.
- **findMin**(): Returns minimum element in T.
- **findMax**(): Returns maximum element in T.
- **find**(x): Returns true if T contains x. given node.

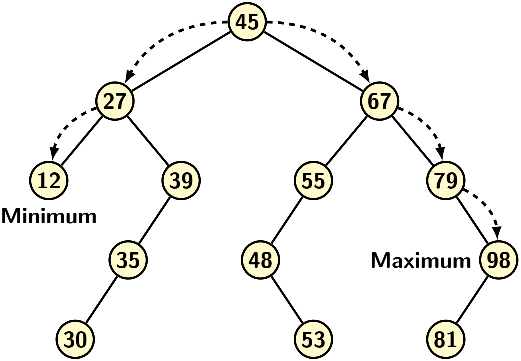
Minimum & Maximum

- Minimum is the leftmost node & maximum is the rightmost node.

```
node * findMin(BST T) { // Leftmost node
    x = getRoot(T);
    while (x->left != NULL)
        x = x->left;
    return x;
}

node * findMax(node *x) { // Rightmost node
    x = getRoot(T);
    while (x->right != NULL)
        x = x->right;
    return x;
}
```

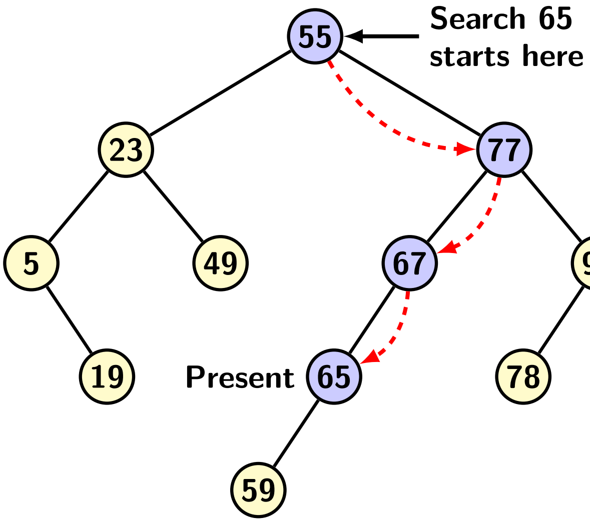
Example for Min & Max



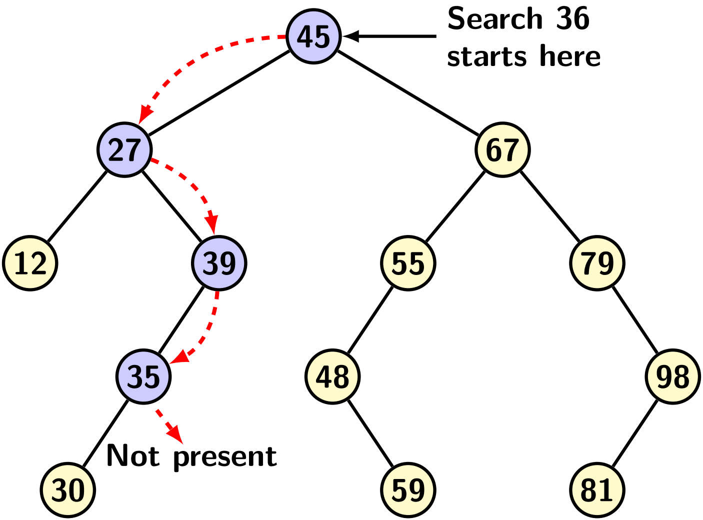
Pseudo Code for Search

```
node * Search(BST T, Val k) {
    x = getRoot(T);
    while (x != NULL && k != x->key) {
        if (k < x->key)
            x = x->left;
        else
            x = x->right;
    }
    return x;
}
```

Search for Element Present



Search for Element not Present



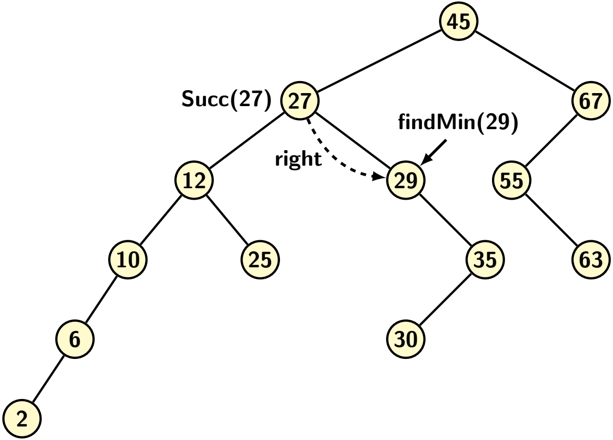
Successor & Predecessor

- An important operation is to locate the inorder successor and the inorder predecessor of a node.
- It is a bit harder than plain membership search.
  - If a node x has a nonempty RST then its succ(x) is the smallest key in RST(x).
  - If x has an empty RST then its succ(x) is the lowest anscestor of x whose left child is also an ancestor of x (it could be x itself).
  - For finding the predecessor you need to apply symmetric rules.

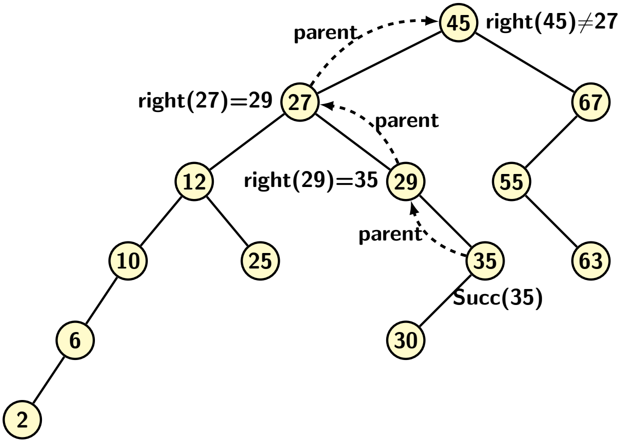
Pseudo Code for Successor

```
node * successor(node *x) {
    if (x->right != NULL)
        return findMin(x->right);
    y = x->parent;
    while (y != NULL && x == y->right) {
        x = y;
        y = y->parent;
    }
    return y;
}
```

Successor Example 1



Successor Example 2



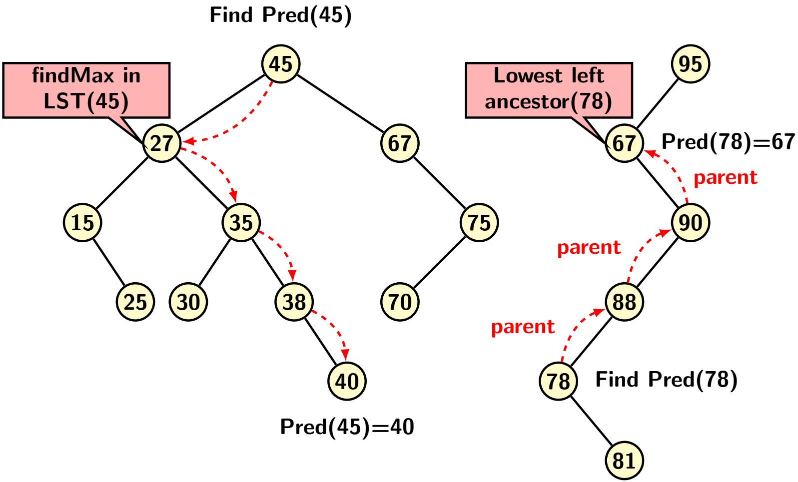
Predecessor

- If x has nonempty LST, then $pred(x)$ = $\max\{y|y \in LST(x)\}$.
- if x does not have a left child, i.e. LST(x) = NULL, then $pred(x)$ is the lowest (first) left ancestor of x.

```
node * predecessor(node *x) {
    if (x->left != NULL)
        return findMax(x->right);

    // Find lowest left ancestor
    y = x->parent;
    while (y != NULL && x == y->left) {
        x = y;
        y = y->parent;
    }
    return y;
}
```

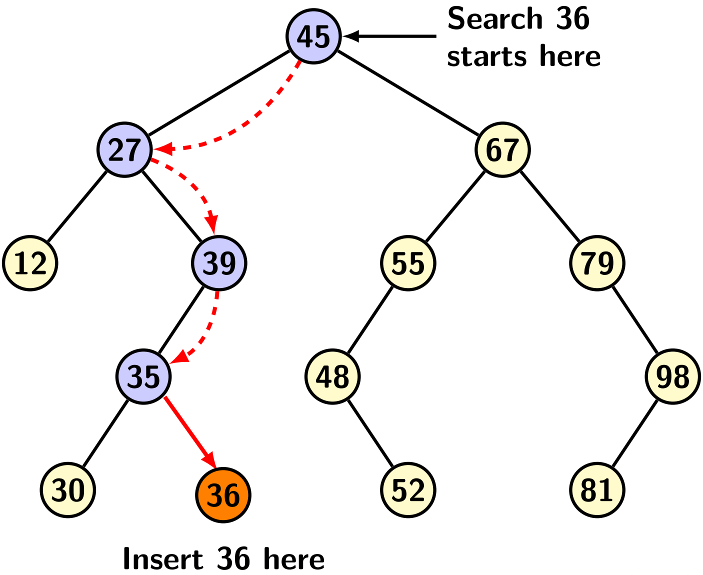
Predecessor Example



Inserting a New Node

- Use the membership search for the new value.
- If the new value is not present you will reach a node with no child pointer.
- Insert a new node at that point with the input value, and create a pointer for this node.
- The new node will always be a leaf node.

Inserting 36



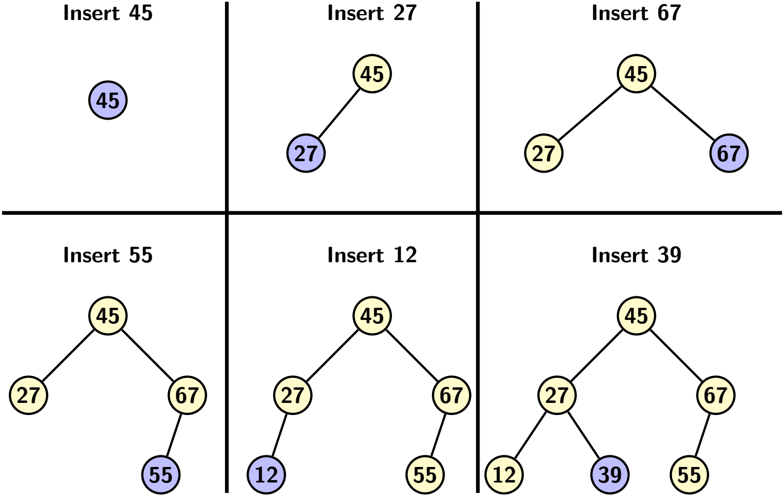
Insertion into Left Subtree

```
kr = getKey(root(T));
if (key < kr )) {
    // Insertion into left subtree of the root
    if leftChild(root(T)) == NULL {
        Create a new node leftchild(root(T))
            with value key;
        return T;
    } else
        Insert(leftChild(root(T)), key);
}
```

Insertion into Right Subtree

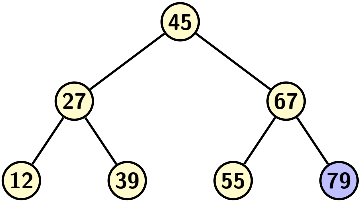
```
kr = getKey(root(T));
if (key > kr) {
    // Insert in right subtree.
    if rightChild(T) == NULL {
            Create a new node rightchild(root(T
                )) with value key;
            return T;
    } else
         Insert(rightChild(root(T)), key);
}
```

Insertion Example

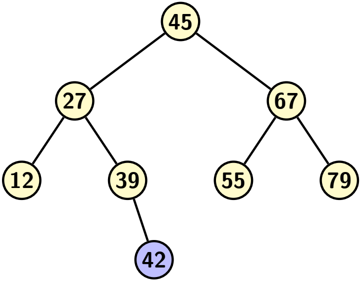


Insertion Example (contd.)

Insert 79



Insert 42



Deletion from BST

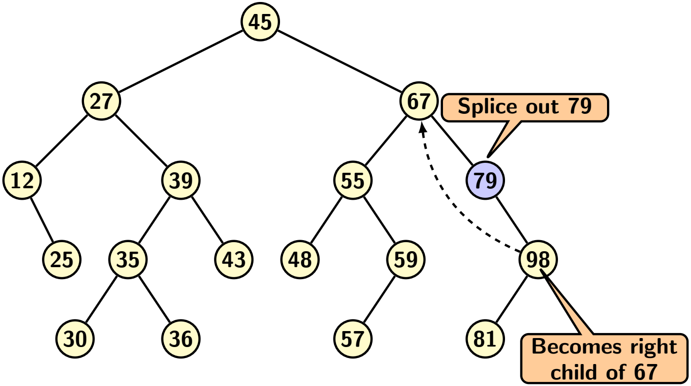
The key to be removed may belong either to a leaf node or to an internal node.

- **Case 1**: Deleting a leaf node. No readjustment needed. It can just be removed.
- **Case 2**: Deleting an internal node x could be achieved by replacing x by its inorder predecessor or successor in BST.
- We analyze the deletion scenarion under two subcases:
  - **Case 2.1**: Node has only one child.
  - **Case 2.2**: Node has two children.

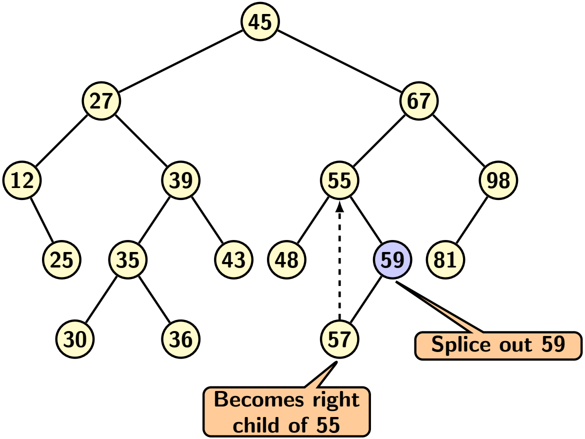
Case 2.2: Deletion from BST

- If node x has just on child, set **child**(x) as **child**(**parent**(x)). This amounts to splicing out x from the tree.
- If x has two children find the inorder predecessor **inpred**(x).
  - Replace key in x by the key in **inpred**(x).
  - If **inpred**(x) is a leaf node, just delete it.
  - Otherwise, **inpred**(x) can have only a left child (why?)
  - Splice out **inpred**(x) from the tree and make left child of **inpred**(x) as right child of **parent**(**inpred**(x))

Case 2.1: Node Having One Child

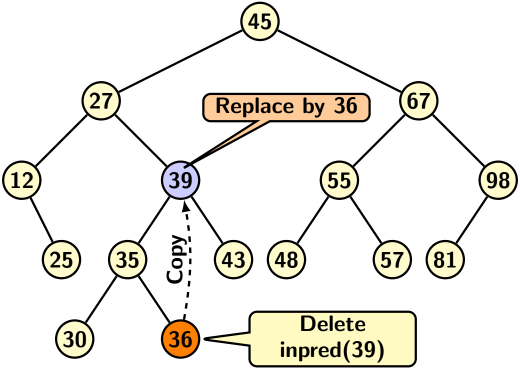


Case 2.1: Node Having One Child



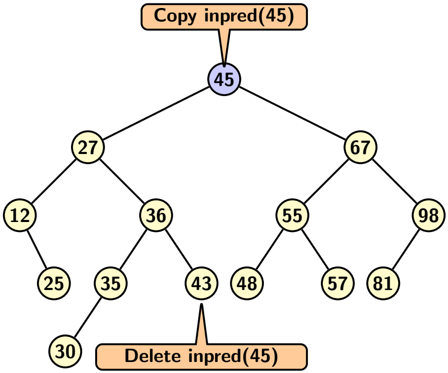
Case 2.2: Node Having Two Children

- **inpred**(39) = 36 which is a leaf node.



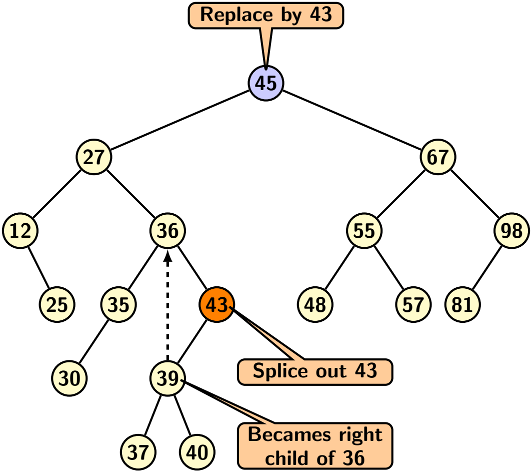
Case 2.2: Node Having Two Children

- **inpred**(45) = 43, and 43 is a leaf node.



Case 2.2: Node Having Two Children

- Node 43 may only have a left subtree.
- In that case, splice out 43 after copying into the root.



Analysis of BST Operations

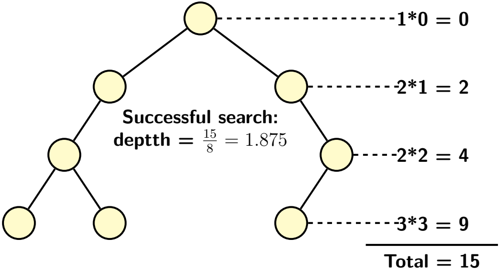
- The worst case scenario occurs when BST is completely skewed.
- So, insertion may require time up to O(n).
- The best case scenario occurs when BST is balanced.
- In this case, insertion requires time of O($\log n$).
- For average case scenario, estimate the number of links to be traversed in an average.

Internal Path Length

Total Internal Path Length

It is the sum of depth of all its nodes.

In the tree shown below the total internal path length is: 15



Average Case Analysis of BST Operation

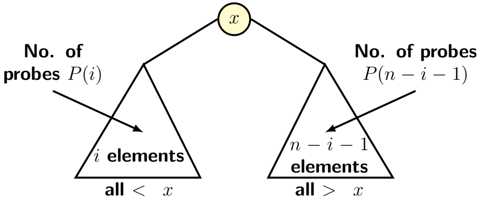
Lemma

If n elements are inserted in random order into an initially empty BST then average path length is O($\log n$).

- Before trying presenting the proof, let analyze how internal path length can be computed.
- BST is formed by only insertions, we assume all order of insertions is equally likely.
- Let $P(n)$ be the average path length to a node in BST with n nodes.
- Let x be the first element to be inserted, it is the root.
- x can be equally likely to be 1st, second, third or nth in sorted order. So prob($x = i$) = $\frac{1}{n}$

Average Case Analysis of BST Operation

- $P(0) = 0$, and $P(1) = 1$.
- Now consider a fixed $i,\ 0 \leq i \leq n-1$.
- Let us see how the next insertion occur.



Average Case Analysis of BST Operation

- If the root is searched number of probes = 1
- If an element in LST(root) is searched, average number probes = $P(i)$
- If an element in RST(root) is searched, average number probes = $P(n-i-1)$
- Probability of seeking any element = $\frac{1}{n}$.
- So, average path length for a fixed i is given by:

$$
\begin{aligned}
P(n,i) &= \frac{1}{n}(1 + i(1 + P(i)) + (n-i-1)(1 + P(n-i-1)) \\
&= 1 + \frac{i}{n}P(i) + \frac{n-i-1}{n}P(n-i-1)
\end{aligned}
$$

Average Case Analysis of BST Operation

Lemma

Prove that $P(n) = 1 + 4\log n$.

Proof:

- $P(n) = \sum_{i=0}^{n-1} P(n,i) \times$ Prob{LST has i nodes}.
- LST has i element means that $i+1$ element must be x probability of which is $\frac{1}{n}$. So, in other words,

$$P(n) = \frac{1}{n}\sum_{0}^{n-1} P(n,i)$$
$$= \frac{1}{n}\sum_{0}^{n-1}\left(1 + \frac{i}{n}P(i) + \frac{n-i-1}{n}P(n-i-1)\right)$$

Average Case Analysis of BST Operation

Proof (contd):

$$= 1 + \frac{1}{n^2} \sum_{i=0}^{n-1} (iP(i) + (n-i-1)P(n-i-1))$$

$$= 1 + \frac{2}{n^2} \sum_{i=0}^{n-1} iP(i)$$

- Now use induction to prove that above expression $\leq 1 + 4\log n$.

Proof (contd):

- Base case: $P(1) = 1$ and also expression $1 + \frac{2}{n^2}\sum_{i=0}^{n-1} iP(i) = 1$.
- Induction hypothesis: assume that $P(i) = 1 + 4\log i$ for $0 \leq i < n$.
- Induction step:

$$
\begin{aligned}
P(n) &\leq 1 + \frac{2}{n^2}\sum_{i=0}^{n-1} i(1 + 4\log i) \\
&= 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} 4i\log i + \frac{2}{n^2}\sum_{i=0}^{n-1} i \\
&\leq 2 + \left(\frac{8}{n^2}\sum_{i=1}^{n-1} i\log i\right), \text{ since } \sum_{i=1}^{n-1} i \leq \frac{n^2}{2}
\end{aligned}
$$

Average Case Analysis of BST Operation

Proof (contd):

Therefore, $P(n) \leq 2 + \frac{8}{n^2} \sum_{i=1}^{n-1} i \log i$.

Now consider the expression $\sum_{i=1}^{n-1} i \log i$

$$\begin{aligned}
\sum_{i=1}^{n-1} i \log i &= \sum_{i=1}^{\lceil \frac{n}{2} \rceil - 1} i \log i + \sum_{\lceil \frac{n}{2} \rceil}^{n-1} i \log i \\
&\leq \sum_{i=1}^{\lceil \frac{n}{2} \rceil - 1} i \log \frac{n}{2} + \sum_{\lceil \frac{n}{2} \rceil}^{n-1} i \log n \\
&\leq \frac{n^2}{8} \log \frac{n}{2} + \frac{3n^2}{8} \log n
\end{aligned}$$

Average Case Analysis of BST Operation

Proof (contd):

Then simplifying from the last expression we have

$$\sum_{i=1}^{n-1} i \log i = \frac{n^2}{2} \log n - \frac{n^2}{8}$$

Therefore,

$$\begin{aligned} P(n) &\leq 2 + \frac{8}{n^2} \sum_{i=1}^{n-1} i \log i \\ &\leq 2 + \frac{8}{n^2} \left(\frac{n^2}{2} \log n - \frac{n^2}{8} \right) \\ &= 1 + 4 \log n. \end{aligned}$$

Summary

- We discussed about both structural properties and BST properties of Binary Search Trees.
- Applications of BST discussed in context of a dynamic environment where data enter and leave on continuous basis.
  - For example, in Dictionary type operations.
  - Or more precisely for (key, value) kind of store.
- We also analyzed average case time complexity for BST operation.