More Binary Search Tree Methods

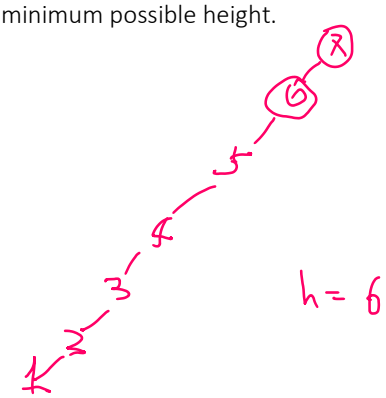
- Minimum and Maximum
  - Because of the binary-search-property, the minimum item is stored in the leftmost node in a binary search tree; and the maximum item is in the rightmost node.

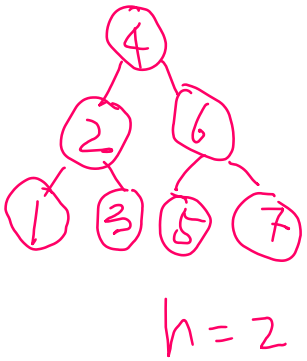
```
tree_minimum (x)
1  while (x.left is not None)
2        x = x.left
3  return x.val
```

```
tree_maximum (x)
1  while (x.right is not None)
2        x = x.right
3  return x.val
```

  - Since we need to scan from the root to a leaf, it is easy to see that **tree_minimum** and **tree_maximum** both have time complexity $O(h)$, where h is the height of the tree.
  - Actually, many binary search tree methods have time complexity $O(h)$, so we have a question that how tall and how short a binary tree can be.

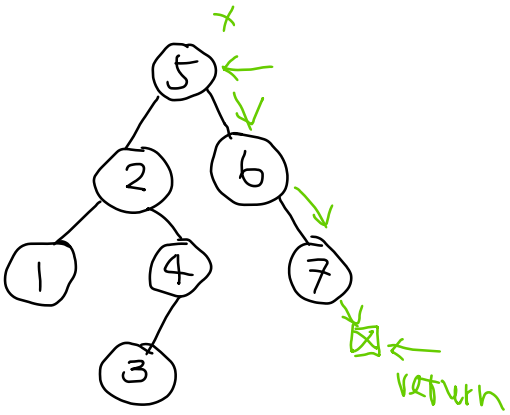
1. Draw a binary search tree with numbers $1\sim7$ of maximum possible height and a tree with same items of minimum possible height.





- In general, the height of a binary tree with n nodes is between $\lfloor \lg n \rfloor$ and $n-1$. When an algorithm has time complexity $O(h)$, and if there is not more information about the shape of the binary tree, we can only use $O(n)$ to bound $O(h)$.
  - When a binary tree has height $\Theta(\lg n)$, then we say this binary tree is ***balanced***. We have learned that a binary heap (a complete binary tree) has height $\lfloor \lg n \rfloor$, so it is a balanced binary tree.

- Search in Binary Search Tree
  - Given a pointer to the root of a subtree, **tree_search** searches for $item$ in the subtree rooted at x. It returns a pointer to the node that contains the $item$, or it returns a pointer points to a None node.
  - One can also implement dunder method $\_\_contains\_\_\,(self, item)$ following this pseudo-code.

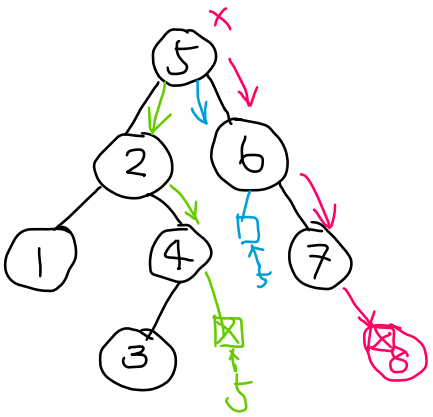



```
tree_search (x, item)
1  if x = None or item = x.val
2        return x
3  if item < x.val:
4        tree_search (x.left, item)
5  else  tree_search (x.right, item)
```

- In the worst case, we need to scan from the root all the way down to one of the leaves, so the worst-case time complexity for **tree_search** is $\Theta(h)$, where h is the height of the tree.

- Insertion
  - In this pseudo-code, **tree_insertion** insert a ***unique new*** item into tree T. Remind that, **tree_search** returns a pointer to a None node here since $item$ is not in T before.
  - As an aside, so that we can insert duplicate values into a binary search tree, we need to update the pseudo-code: in **tree_search**, we don't when we find $item$ now, we arbitrarily choose either left or right and keep going down.

tree_insertion $(T, item)$
1 $x = T.root$
2 $z =$ **tree_search** $(x, item)$
3 $z.val = item$

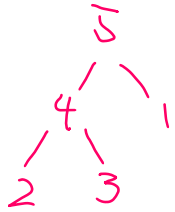


  - This algorithm works since **tree_search** can return the location where $item$ is supposed to be in a binary search tree.
  - Again, since we need to scan from the root to a leaf, the worst-case time complexity for **tree_insertion** is $O(h)$, where h is the height of the tree.

2. Insert 8 and 5 to the binary search tree.

- Constructing a Binary Search Tree using insertion
  - When constructing a binary heap, with same set of items and different adding orders, the outcome binary trees always have the same shape (items can be in different nodes). But this is not true for a binary search tree: insertion order affects the shape of the constructed binary search tree dramatically.

3. Draw the binary heap constructed by adding $1,2,3,4,5$ and the binary heap constructed by adding $4,2,1,3,5$.



4. Draw the binary search tree constructed by inserting $1,2,3,4,5,6,7$ and the binary search tree constructed by inserting $4,6,2,1,3,5,7$.