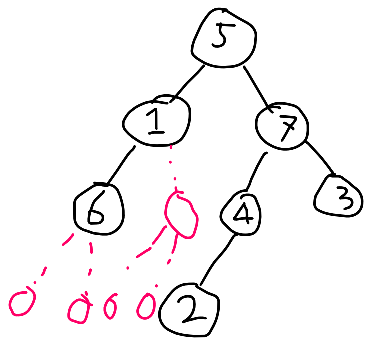
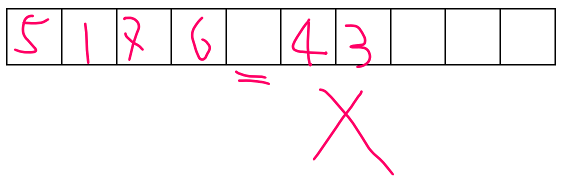
Binary Tree and Binary Search Tree

- We have seen binary trees (where binary means each node has at most two children) in the study of binary heap. Since a binary heap is a complete tree, we can implement it using a list or array. But in general, not all binary trees are complete; if we use the same way to implement any binary tree with a list, there will be empty spots in the list.



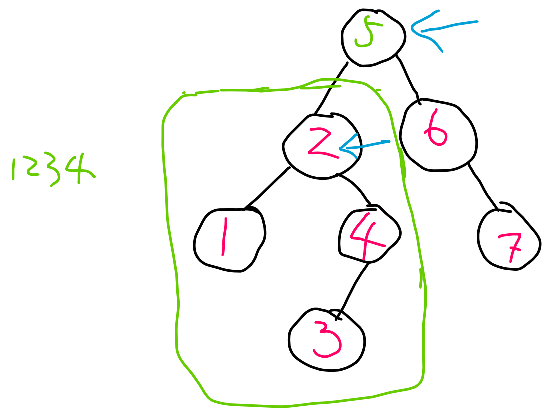


- We need to use a linked data structure to implement a binary tree, where each node contains three attributes:
  1) a piece of data: $self.val$,
  2) a pointer to left child: $self.left$,
  3) a pointer to right child: $self.right$.

  - Note that, we cannot access the parent of node directly.

- If we simply store items into a binary tree, we don't get any advantage from it. People invented the data structure **Binary Search Tree** (**bst**) which is implemented on a binary tree and satisfies **binary-search-property**: let x be a node in a binary search tree, if y is a node in the left subtree of x then $y.val \leq x.val$; if y is a node in the right subtree of x, then $y.val \geq x.val$.

1. Fill numbers 1 to 7 into the following binary search tree so that the binary search property is satisfied.



- What a binary search tree provides us is a binary-search-like quick search procedure.

search 4

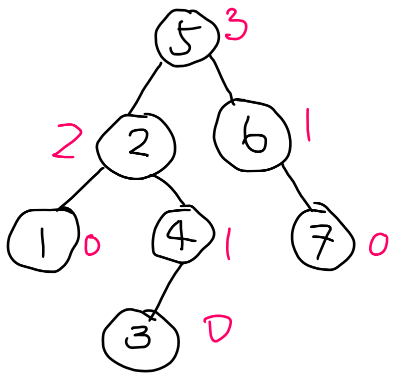
- As an ADT, Binary Search Tree provides at least the following methods:
  - $search(item)$
  - $insert(item)$
  - $delete(item)$

and more. Before introducing insertion and deletion, we will introduce some methods that do not change the data structure and at the same time we will see some nice properties of binary search trees.

Binary Search Tree methods

- Tree Height
  - We have learned that the height of a tree is the distance between the tree root and the farthest leave.
  - An observation is that the height of a node equals to 1 plus the larger height between left and right children. With this observation we can create a recursive height function.

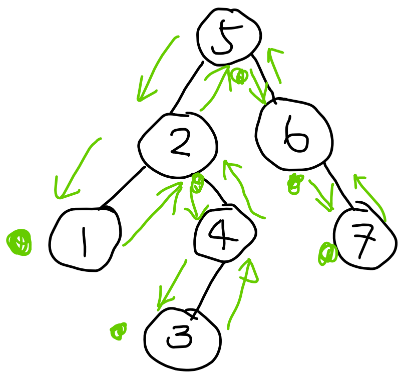
```
 height ($x$)        //$x$ is a node note tree
1 if $x$ is a leaf:
2         return 0
3 else:
4         return $1 + \max(height(x.left),\ height(x.right))$
```



  - The running time of this method can be found this way: at each node, we look at both of its children. Thus, if x is the root of an n-node tree, the time complexity of **height** (x) is $\Theta(n)$.

- Tree Traversal
  - Given a binary tree, it is natural that we want to see all the nodes in the tree. This inspired us to create method that we can traverse all nodes in the tree.
  - One way to traverse a tree rooted at node x is **inorder_tree_walk** (x).

```
 inorder_tree_walk ($x$)
1  if $x$ is not None:
2         inorder_tree_walk ($x.left$)
3         print $x.val$
4         inorder_tree_walk ($x.right$)
```



1, 2, 3, 4, 5, 6, 7

  - if x is the root of an n-node tree, the time complexity of **inorder_tree_walk** (x) is $\Theta(n)$.
  - An **inorder_tree_walk** from the root can print out values in a binary search tree in a non-decreasing order. Thus, a binary search tree keeps values sorted all the time.

  - There are other tree walks:

```
 preorder_tree_walk ($x$)
1  if $x$ is not None:
2     print $x.val$
3     preorder_tree_walk ($x.left$)
4     preorder_tree_walk ($x.right$)
```

```
 postorder_tree_walk ($x$)
1  if $x$ is not None:
2         postorder_tree_walk ($x.left$)
3         postorder_tree_walk ($x.right$)
4         print $x.val$
```

The prefixes $pre-$, $in-$ and $post-$ show at what time the root of a subtree is being visited. Similar to **inorder_tree_walk**, these two tree walks also have time complexity $\Theta(n)$.