ADT — abstract data type — List

data structure —— LinkedList

Dictionary ADT

2 type parameters K (key - lookup)
V (value - result)

like "T" from lists

```
void insert(K key, V value)   ← new mappings
void update(K key, V value)   ← changing mappings
void remove(K key)
List<K> getKeys()
V get(K key)
```

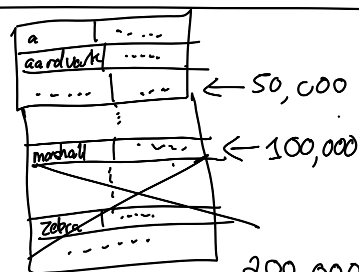
Guarantee:
for every mapping, K is unique

ArrayDictionary

uses an array of pairs of K and V

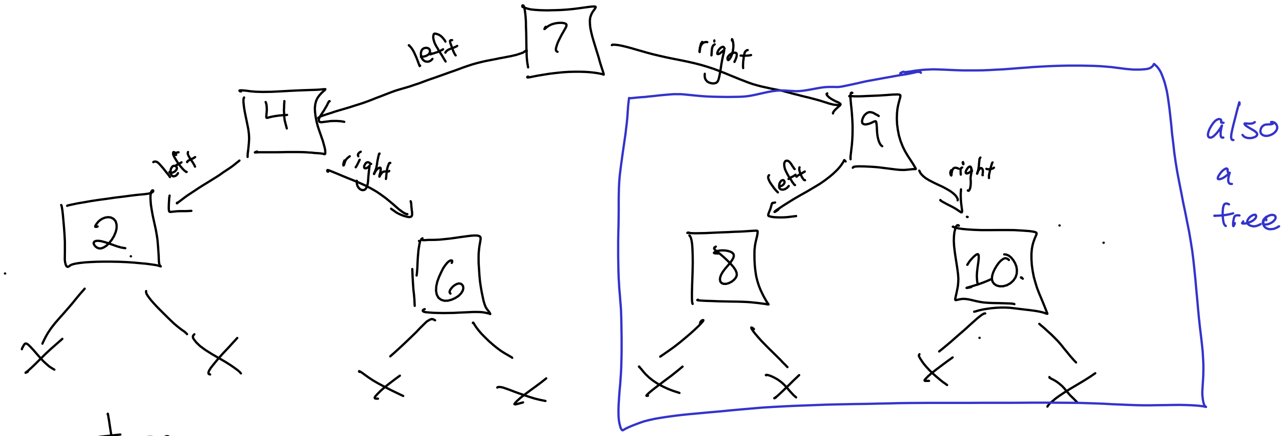
get(K) —— binary search $O(\log n)$

insert(K, V) —— $O(n)$

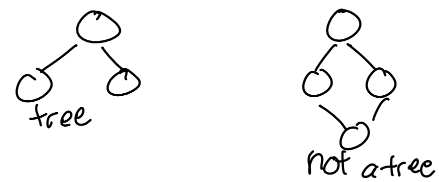


binary search

horn

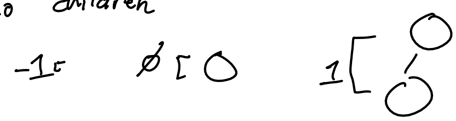


tree

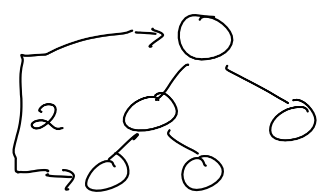


Vocab

- Node — container holding data; point in the tree
- Child — a node to which this node points
- Parent — the node pointing to this one (of which there is at most one)
- Root — the node in a tree with no parent
- Leaf — a node w/ no children
- Ancestor / Descendant



- Size — # nodes in tree
- Height — # of links between root & a deepest leaves



Tree : collection of nodes s.t.

- there is a root which has no parent
- all other nodes have a unique parent
- all nodes have the root as an ancestor.

or

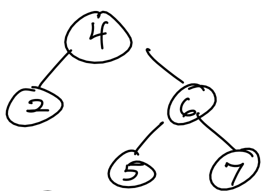
the empty tree

Binary Trees:

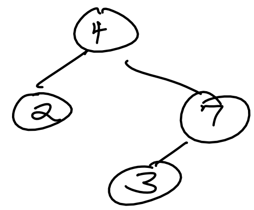
Trees in which each node has at most 1 left child and at most 1 right child

Binary Search Tree:

Every node to the left of a node N has key < the key of N and every node to the right of a node N has key > the key of N



BST



Binary tree not BST

BST data structure implements dictionary ADT interface

```
Method getSize( ) :
    Return sizeOfTree(this->root)
End Method

Method sizeOfTree(Node* n) :
    If n == nullptr :
        Return 0
    Else :
        sL <- sizeOfTree(n->left)
        sR <- sizeOfTree(n->right)
        Return sL + sR + 1
    End If
End Method
```