1 Introduction to trees

A tree T is made up of a set of nodes endowed with parent-child relationship with following properties:

- If T is non-empty, it has a special node called the root that has no parent
- Every node v of T other than the root has a unique parent
- Following the parent relation always leads to the root (i.e., the parent-child relation does not have "cycles")

1.1 Terminology

- Root: node without parent.
- Internal node: node with at least one child.
- External/leaf node: node without children.
- Ancestors: parent, grandparent, great-grandparent, etc.
- Descendants: child, grandchild, great-grandchild, etc.
- Two nodes with the same parent are siblings.
- Depth of a node: number of ancestors not including itself.
- Level: set of nodes with given depth.
- Height of a tree: maximum depth.
- Subtree: tree made up of some node and its descendants.
- Edge: a pair of nodes (u, v) such that one is the parent of the other.
- Path: sequence of nodes such that consecutive nodes in the sequence have an edge

Tree facts

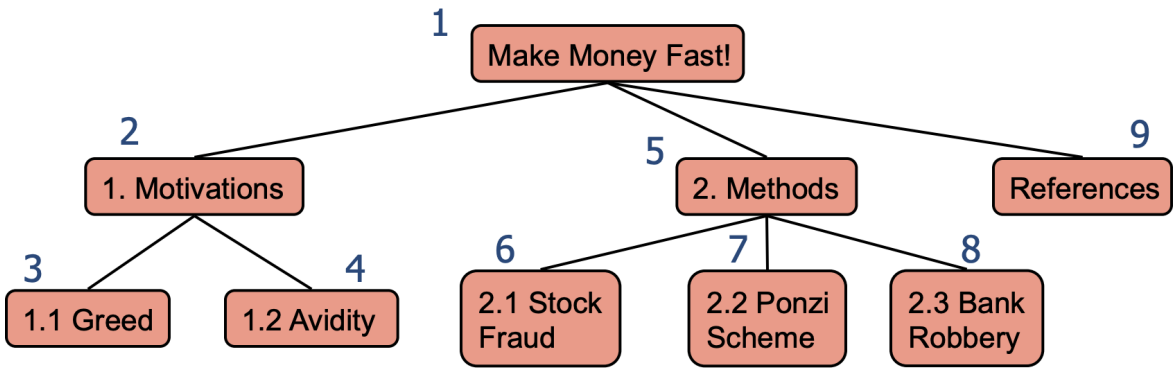
- If node X is an ancestor of node Y, then Y is a descendant of X.
- Ancestor/descendant relations are transitive.
- Every node is a descendant of the root.
- There may be nodes where neither is an ancestor of the other.
- Every pair of nodes has at least one common ancestor.
- The lowest common ancestor (LCA) of x and y is a node z such that z is the ancestor of x and y and no descendant of z has that property.
- In an ordered tree there is a prescribed order for each node's children.

2 Traversal

2.1 Preorder Traversal

To do a preorder traversal starting at a given node, we visit the node before visiting its descendants If tree is ordered visit the child subtrees in the prescribed order. Visit does some work on the node such as print node data, aggregate node data or modify node data. The example shows a pre_order traversal called at root.

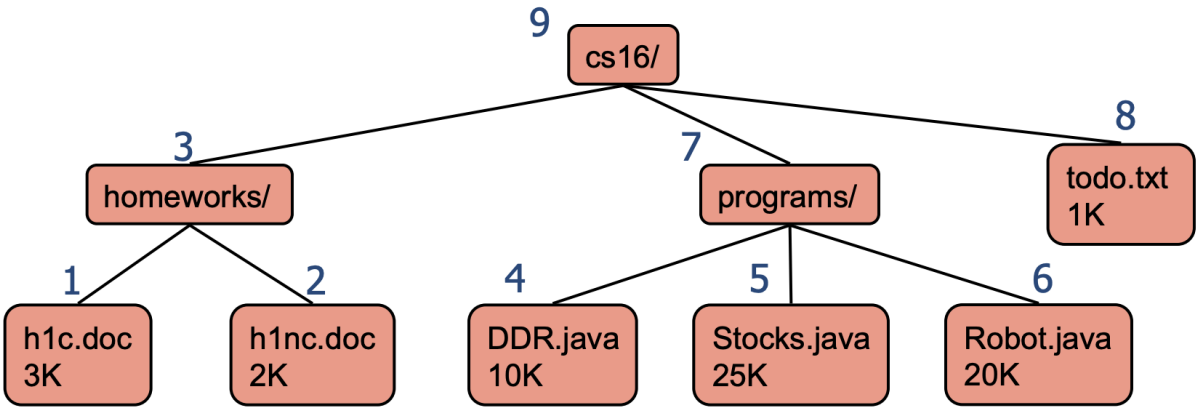
```
1. def pre_order(v):
2.     visit(v)
3.     for each child w of v do
4.         pre_order(w)
```



2.2 Postorder Traversal

To do a postorder traversal starting at a given node, we visit the node after its descendants If tree is ordered visit the child subtrees in the prescribed order.

```
1. def pre_order(v):
2.     for each child w of v do
3.         pre_order(w)
4.     visit(v)
```



3 Binary Trees

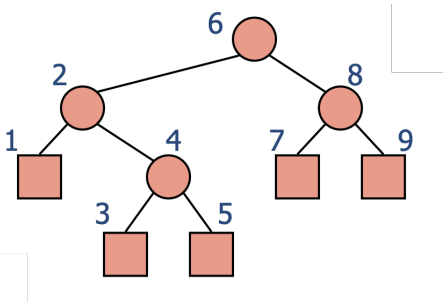
A binary tree is an ordered tree with the following properties: Each internal node has at most two children. Each child node is labeled as a left child or a right child. Child ordering is left followed by right. The right/left subtree is the subtree root at the right/left child. We say the tree is proper if every internal node has two children.

```
1. def is_external(v):                          # Tests if v is a leaf
2.     return v.left = null and v.right = null
```

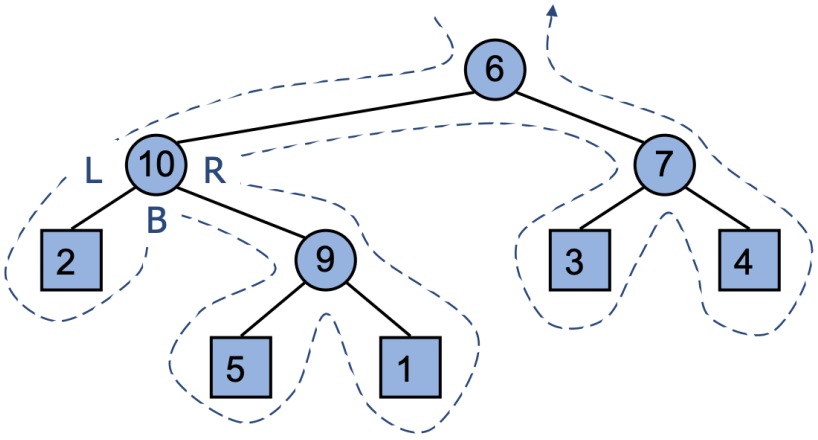
3.1 Inorder Traversal

To do an inorder traversal starting at a given node, the node is visited after its left subtree but before its right subtree.

```
1. def in_order(v):
2.     if v.left != null then
3.         in_order(v.left)
4.     visit(v)
5.     if v.right != null then
6.         in_order(v.right)
```



4 Euler Tour Traversal & Some code



6,10,2,2,2,10,9,5,5,5,9,1,1,1,9,10,6,7,3,3,3,7,4,4,4,7,6

Preorder (first visit): 6, 10, 2, 9, 5, 1, 7, 3, 4

Inorder (second visit): 2, 10, 5, 9, 1, 6, 3, 7, 4

Postorder (third visit): 2, 5, 1, 9, 10, 3, 4, 7, 6

```
1. def height(v):                          # compute height of subtree at v
2.     if v.parent = null then             # root's depth is 0
3.         return 0
4.     else
5.         return depth(v.parent) + 1
```

```
1. def depth(v):                           # compute height of subtree at v
2.     if v.isExternal() then              # a leave's height is 0
3.         return 0
4.     else
5.         h ← 0
6.         for each child w of v do
7.             h ← max(h, height(w))
8.         return h + 1
```

5 Binary Search Trees

A binary search tree is a binary tree storing keys (or key-value pairs) satisfying the following BST property. **"For any node v in the tree and any node u in the left subtree of v and any node w in the right subtree of v, key(u) < key(v) < key(w)"**

5.1 BST Implementation

To simplify BST implementation, keys (or key-value pairs) are only stored in internal noves and not extreenal nodes, with external nodes being null. To search for a key k, we trace a downward path starting at the root. To decide whether to go left or right, we compare the key of the current node v with k. If we reach an external node, this means that the key is not in the data structure.

```
1. def search(k, v):
2.     if v.isExternal() then
3.         return v                                  # unsuccessful search
4.     if k = v.key() then
5.         return v                                  # successful search
6.     if k < v.key() then
7.         return search(k, v.left)
8.     else                                          # k > v.key()
9.         return search(k, v.right)
```

Runs in O(h) time, where h is the height of the tree. In the wort case h = n-1 and in the best case, $h <= log_2 n$
To perform operation put(k, o), we search for key k (using search). If k is found in the tree, replace the corresponding value by o. If k is not found, let w be the external node reached by the seached by the search. We replace w with an internal node storing key k and value o and two external nodes as children. The new node is a leaf and the tree is proper. The operation put runs in O(h) time.
To perform operation remove(k), we search for key k (using search) to find the node w holding k. We distinguish between two cases being, w has one external child and w having two internal children. If k is not in the tree, we can either throw an exception or do nothing depending on the ADT specs.

- **Deletion Case 1**: Suppose that the node w we want to remove has an external child, which we call z. To remove w we remove w and z from the tree. We then promote the other child of w to take w's place. This preserves the BST property.
- **Deletion Case 2**: Suppose that the node w we want to remove has two internal children. To remove w, we find the internal node y following w in an inorder traversal (i.e., y has the smallest key among the right subtree under w). we copy the entry from y into node w. we remove node y and its left child z, whcih must be external. This preserves the BST property.

```
1. def remove(k):
2.     w ← search(k, root)
3.     if w.isExternal() then
4.         return null                                  # key not found
5.     else if w has at least one external child z then
6.         remove z
7.         promote the other hcild of w to take w's place
8.         remove w
9.     else                                             # w has two internal children
10.        y ← w's successor in an inorder traversal
11.        replace contents of w with entry from y
12.        remove y as above
```

Complexity Consider a binary search tree with n items and height h. The sapce used is O(n) and get, put and remove take O(h). The height h can be n in the worst case and log(n) in the best case. Therefore, using better insertion routines can drastically cut running times.

5.2 Duplicate key values in BST

It is possible to allow duplicates, but it comes at the cost of additional compleixty.

- Allowing left descendants to be equal to the parent. i.e. $key(leftdescendant) <= key(node) <= key(rightdescendant)$
- Using a list to store duplicates.

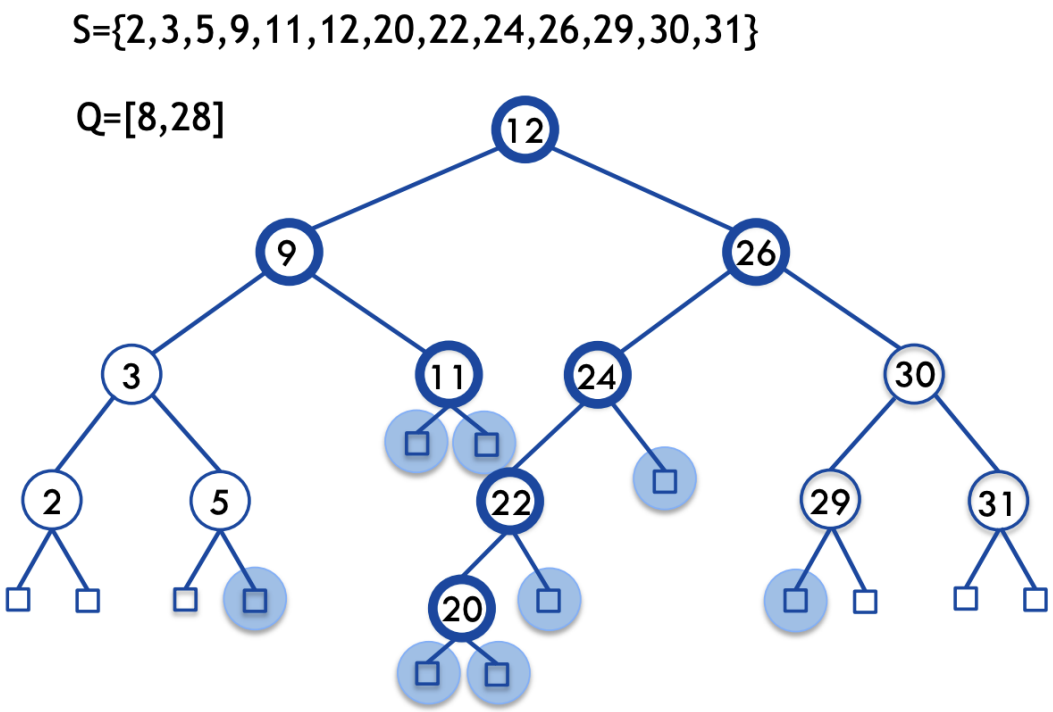
5.3 Range Queries

A range querey is defined by two values k_1 and k_2. We are to find all ekys k stored in T such that $k_1 <= k <= k_2$. The algorithm is a restricted version of inorder traversal. When at node v:

- If $key(v) < k_1$: Recursively search right subtree
- If $k_1 <= key(v) <= k_2$: Recursively search left subtree, add v to range output, search right subtree.
- If $k_2 < key(v)$: REcursively search left subtree

```
1. def range_search(T, k1, k2):
2.     output ← []
3.     range(T.root, k1, k2)
```

```
1. def range(v, k1, k2):
2.     if v is external then
3.         return null
4.     if key(v) > k2 then
5.         range(v.right, k1, k2)
6.     else if key(v) < k1 then
7.         range(v.right, k1, k2)
8.     else
9.         range(v.left, k1, k2)
10.        output.append(v)
11.        range(v.right, k1, k2)
```



Performance Let P_1 and P_2 be the binary search paths to k_1 and k_2. We say a node v is a:

- Boundary node if v in P_1 or P_2
- Inside node if key(v) in [k_1, k_2] but not in P_1 or P_2
- Outside node if key(v) not in [k_1, k_2] and not in P_1 or P_2

The alogirithm only vists boundary and inside nodes and $|insidenodes| <= |output|$ and $|boundarynodes| <= 2 * treeheight$. Therefore, since we only spend O(1) time per node we visit, the total running time of range search is O($|output| + treeheight$)

6 Maintaining a balanced BST

Ranked balanced trees are a family of balanced BST implementations that use the idea of keeping a "rank" for every node, where r(v) acts as a proxy measure of the size of the subtree rooted at v. This is primarily done to reduce the discrepancy between the ranks of left and right subtrees.

6.1 AVL trees

AVL trees are rank-balanced trees, where r(v) is its height of the subtree rooted at v. Balance constraint: The ranks of the two children of every internal-node differ by at most 1

Proof (by induction): The height of an AVL tree stroing n keys is O(logn)

- Let N(h) be the minimum number of keys of an AVL tree of height h.
- We can easily see that N(1) = 1 and N(2) = 2
- Clearly N(h) > N(h-1) for any h >= 2.
- For For h > 2, the smallest AVL tree of height h contains the root node, one AVL subtree of height h-1 and another of height at least h-2:
- N(h) >= 1 + N(h-1) + N(h-2) > 2 N(h-2)
- By induction we can show that for h even N(h) >= $2^{h/2}$
- Taking logarithms: h < 2 log N(h)
- Thus the height of an AVL tree is O(log n)

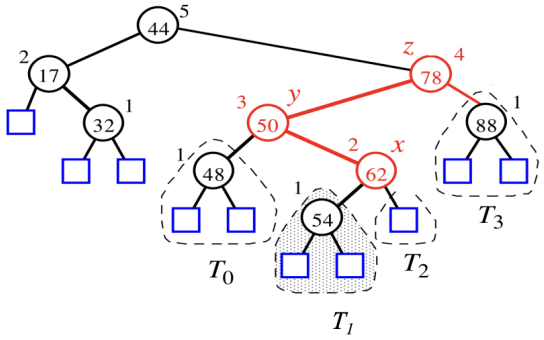
6.2 Insertion in AVL trees

Suppose we are to insert a key k into our tree:

- If k is in the tree, search for k ends at node holding k. There is nothing to do so tree structure does not change
- If k is not in the tree, search for k ends at external node w. Make this be a new internal node containing key k
- The new tree has BST property, but it may not have AVL balance property at some ancestor of w since some ancestors of w may have increased their height by 1 and every node that is not an ancestor of w hasn't changed its height.
- We use rotations to re-arrange tree to re-establish AVL property, while keeping BST property.

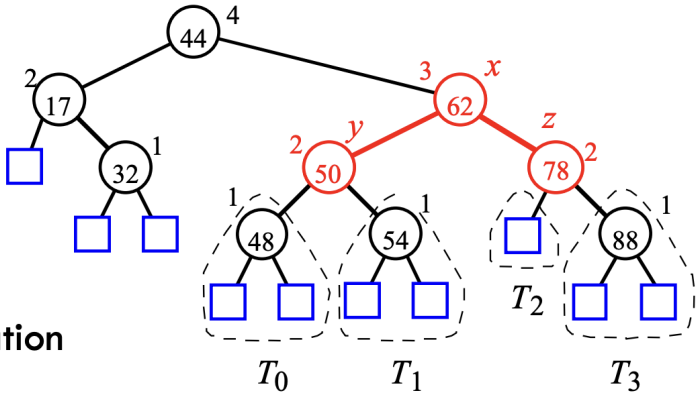
7 Re-establishing AVL property

- Let w be the location of a newly inserted node. (54)
- Let z be the lowest ancestor of w, whose children heights differ by 2.
- Let y be the child of z that is the ancestor of w.
- Let x be the child of y that is is the ancestor of w.



If tree does not have AVL property, do a trinode restructure at x, y, z

It can be argued that tree has AVL property after operation

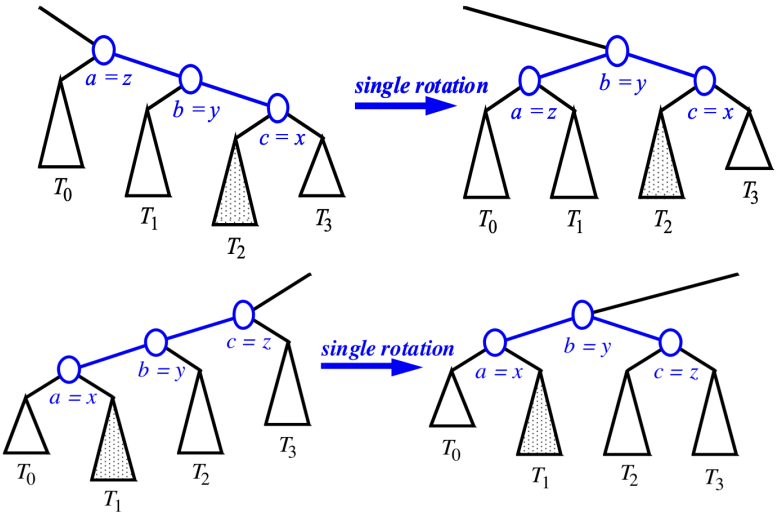


7.1 Augmenting BST with a height attribute

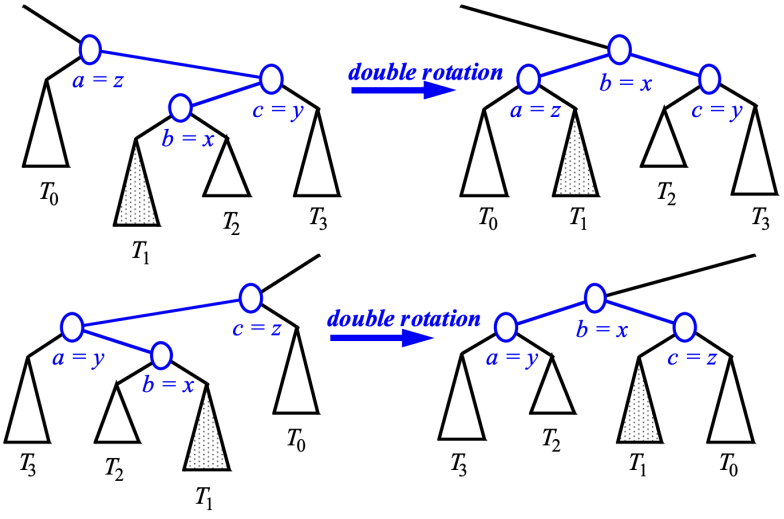
But how do we know the height of each node? If we had to compute this from scratch it would take O(n) time Therefore, we need to have this pre-computed and update the height value after each insertion and rebalancing operation: After we create a node w, we should set its height to be 1, and then update the height of its ancestors. After we rotate (z, y, x) we should update their height and that of their ancestors. Thus, we can maintain the height only using O(h) work per insert

1. **def** reststructure(x): *# O1*
   Input: A node x of a binary tree T that has boath a parent y and a grandparent z.
   Output: The tree T after a trinode restrcucting (single or double roation)
2. Let (a,b,c) be the left-to-right (inorder) listing of the nodes x, y, and z, and let (To,T1,T2,T3) be the left-to-right (inorder) listing of the four subtrees of x, y, and z that are not rooted at x, y, and z.
3. Replace the subtree with a new subtree rooted at b.
4. Let a be the left child of b and let To and T1 be the left and right subtrees of a.
5. Let c be the right child of b and let T2 and T3 be the left and right subtrees of c.
6. Recalculate the heights of a, b, and c from the values stored at their children
7. **return** b

Single Rotations:



Double rotations:



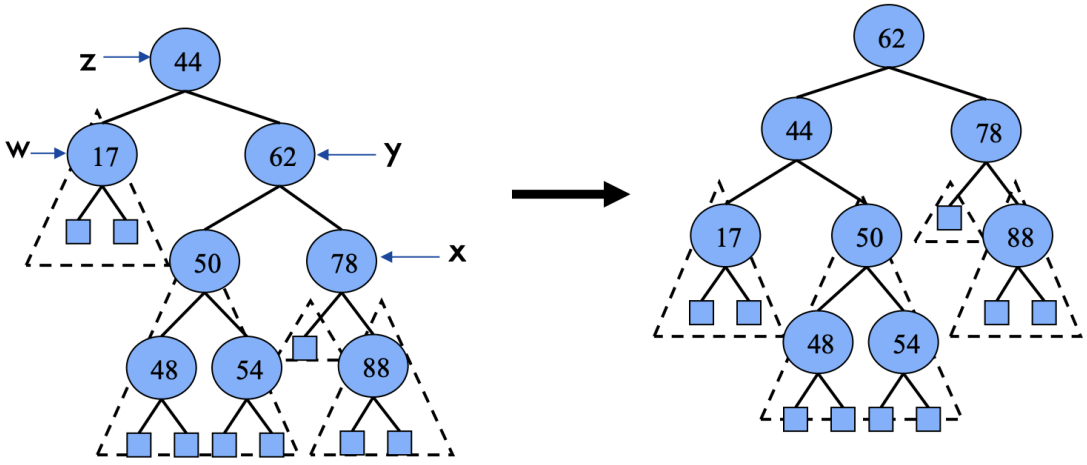
7.2 Removal in AVL trees

Suppose we are to remove a key k from our tree:

- If k is not in the tree, search for k ends at external node. There is nothing to do so tree structure does not change.
- If k is in the tree, search for k performs usual BST removal leading to removing a node with an external child and promoting its other child, which we call w.
- The new tree has BST property, but it may not have AVL balance property at some ancestor of w since some ancestors of w may have decreased their height by 1 and every node that is not an ancestor of w hasn't changed its heights.
- We use rotations to rearrange tree and re-establish AVL property, while keeping BST property.

Re-establishing AVL property

- Let w be the parent of deleted node.
- Let z be the lowest ancestor of w, whose children heights differ by 2.
- Let y be the child of z with larger height (y is not an ancestor of w).
- Let x be child of y with larger height.



This restores the AVL property at z, but it may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached.

7.3 AVL Tree Performance

The data structure uses O(n) space. Height of the tree O(log n). Searching takes O(log n) time. Insertion takes O(log n) time. Removal takes O(log n) time

8 Map ADT

- get(k): if the map M has an entry with key k, return its associated value.
- put(k, v): if key k is not in M, then insert (k, v) into the map M; else, replace the existing value associated to k with v.
- remove(k): if the map M has an entry with key k, remove it.
- size(), isEmpty()
- entrySet(): return an iterable collection of the entries in M.
- values(): return an iterable collection of the values in M.
- keySet(): return an iterable collection of the keys in M.

8.1 Sorted map ADT (extra methods)

- firstEntry() returns the entry with the smallest key; if map is empty, returns null.
- lastEntry() returns the entry with the largest key; if map is empty, returns null.
- ceilingEntry(k) returns the entry with the least key that is greater than or equal to k (or null, if no such entry exists).
- floorEntry(k) returns the entry with the greatest key that is less than or equal to k (or null, if no such entry exists).
- lowerEntry(k) returns the entry with the greatest key that is strictly less than k (or null, if no such entry exists).
- higherEntry(k) returns the entry with least key that is strictly greater than k (or null, if no such entry exists).
- subMap(k1,k2) returns an iteration of all the entries with key greater than or equal to k1 and strictly less than k2.

8.2 List-Based (unsorted) Map

We can implement a map using an unsorted list of key-item pairs To do a get and put we may have to traverse the whole list, so those operations take O(n) time. Only feasible if map is very small or if we put things at the end and do not need to perform many gets (i.e., system log)

8.3 Tree-Based (sorted) Map

We can implement a sorted map using an AVL tree, where each node stores a key-item pair To do a get or a put we search for the key in the tree, so these operations take O(h) time, which can be O(log n) if the tree is balanced. Only feasible if there is a total ordering on the keys