Lecture 12:
AVL Trees

Pure Consciousness is the field of perfect order, balance, and efficiency

1

1

Hash Tables

- Hash tables are a highly efficient O(1) implementation of the Map/Dictionary interface
  - findElement(k)
  - insertItem(k, e)
  - removeElement(k)
  - What are its disadvantages?

2

2

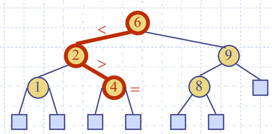
Wholeness Statement

A binary search tree allows users to associate keys to elements, then to access or remove those elements by key. An AVL tree supports efficient implementation of the binary search tree by maintaining balance and order through restructuring (rebalancing) when key-element pairs are inserted and removed. *Science of Consciousness:* Through the TM technique, each of us has access to the source of thought which is a field of perfect order, balance, and efficiency; contact with this field restructures and rebalances our physiology.

3

3

Binary Search Trees

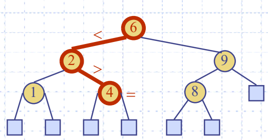


4

4

Binary Search Tree

- A binary search tree is a binary tree storing keys (or key-element items) at its internal nodes and satisfying the following property:
  - Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) \le key(v) \le key(w)$
- External nodes do not store items
- An in-order traversal of a binary search tree visits the keys in increasing order



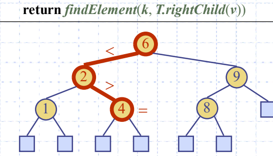
5

5

Search

- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of k with the key of the current node
- If we reach a leaf, the key is not found and we return NO_SUCH_KEY
- Example: findElement(4)

```
Algorithm findElement(k, v)
    if T.isExternal (v)
        return NO_SUCH_KEY
    if k < key(v) then
        return findElement(k, T.leftChild(v))
    else if k = key(v) then
        return element(v)
    else { k > key(v) }
        return findElement(k, T.rightChild(v))
```

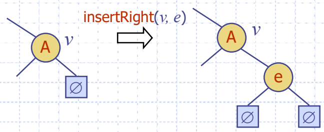


6

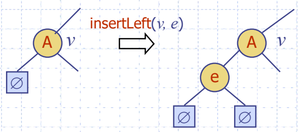
6

Recall: Insert Methods

insertRight(v, e)
Right child must be external!



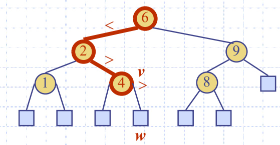
insertLeft(v, e)
Left child must be external.

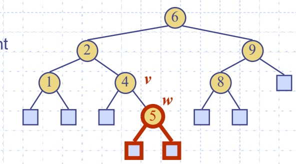


7

Insertion

- To perform operation insertItem(k, o), we search for key k
- If k is not already in the tree, then let w be the leaf reached by the search and v the parent
- We insert k at node w by inserting a node to the right of internal node v using insertRight(v)
- Example: insert 5

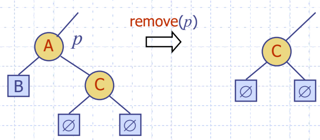


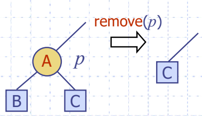


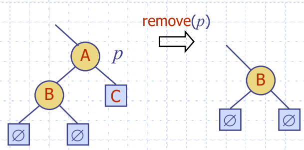
8

Recall: Remove Method

remove(p)
Either the left or right child must be external!
We can only remove the node above an external node.



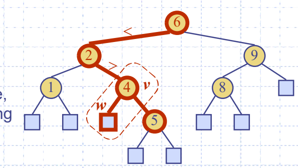


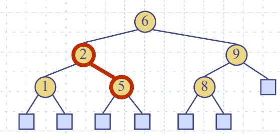


9

Deletion (Case 1)

- To perform operation removeElement(k), we search for key k
- Assume key k is in the tree, and let v be the node storing k
- Two cases:
  - Node v has a leaf child w
  - Node v has no leaf child
- If node v has a leaf child w, we remove v and w from the tree with operation remove(v)
- Example: remove 4

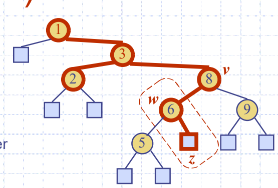




10

Deletion (Case 2)

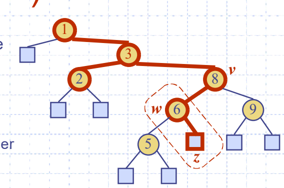
- We consider the case where the key k to be removed is stored at a node v whose children are both internal
  - we find the internal node w that precedes v in an in-order traversal
  - we copy $key(w)$ into node v
  - we remove node w and its right child z (which must be a leaf) by means of operation remove(w)
- Example: remove 8

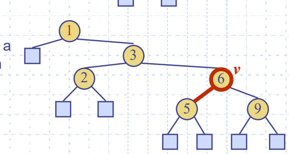


11

Deletion (Case 2)

- We consider the case where the key k to be removed is stored at a node v whose children are both internal
  - we find the internal node w that precedes v in an in-order traversal
  - we copy $key(w)$ into node v
  - we remove node w and its right child z (which must be a leaf) by means of operation remove(w)
- Example: remove 8





12

Binary Search Trees

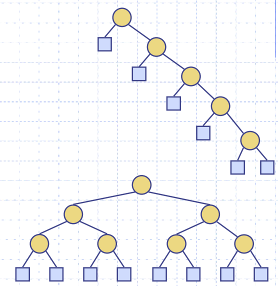
- What are the problems with binary search trees as the basis of the search, insert, and delete operations?

13

13

Performance

- Consider a dictionary with n items implemented by means of a binary search tree of height h
  - the space used is $O(n)$
  - methods findElement , insertItem and removeElement take $O(h)$ time
- The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case



14

14

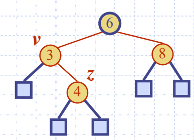
Main Point

1. A binary search tree is a binary tree with the property that the value at each node is greater than the values in the nodes of its left subtree (child) and less than the values in the nodes of its right subtree. When implemented properly, the operations (search, insert, and remove) can be efficiently accomplished.
   Science of Consciousness: Such data structures and algorithms reflect the following SCI principles: law of least action, principle of diving, perfect order, perfect balance.

15

15

AVL Trees

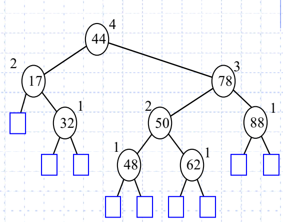


16

16

AVL Tree Definition

- **AVL trees are balanced.**
- An AVL Tree is a ***binary search tree*** such that for every internal node v of T, the *heights of the children of v can differ by at most 1.*

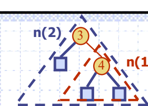


An example of an AVL tree where the heights are shown next to the nodes:

17

17

Height of an AVL Tree



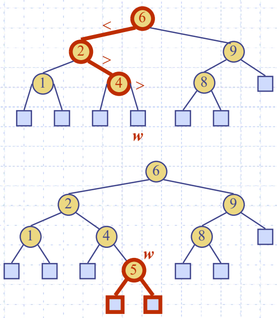
- **Fact**: The ***height*** of an AVL tree storing n keys is O(log n).
- **Proof**: Let us bound **n(h):** the minimum number of internal nodes of an AVL tree of height h.
- We easily see that n(1) = 1 and n(2) = 2
- For n > 2, an AVL tree of height h contains the root node, one AVL subtree of height h-1 and another of height h-2.
- That is, n(h) = 1 + n(h-1) + n(h-2)
- Knowing n(h-1) > n(h-2), we get n(h) > 2n(h-2). So
  n(h) > 2n(h-2), n(h) > 4n(h-4), n(h) > 8n(h-6), … (by induction),
  $n(h) > 2^i n(h-2i)$
- Solving the base case
  - Pick i = $\lceil h/2 \rceil$ - 1 since value of the base cases are n(1) = 1 and n(2) = 2
    - i.e., pick i such that $1 \le h-2i \le 2$
  - Thus we get: $n(h) > 2^{h/2-1}$
- Taking logarithms: h < 2log n(h) +2
- Thus the height of an AVL tree is O(log n)

18

18

Recall
Insertion into a BST

- To perform operation insertItem(k, o), we search for key k
- Assume k is not already in the tree, and let let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node
- Example: insert 5

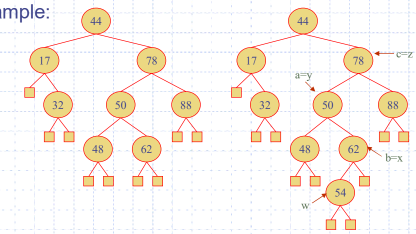


19

19

Insertion in an AVL Tree

- Insertion is as in a binary search tree
- Always done by expanding an external node.
- Example:



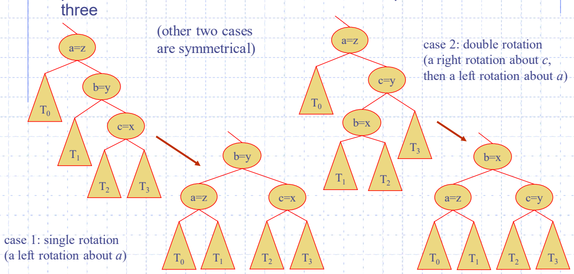
before insertion after insertion

20

20

Trinode Restructuring

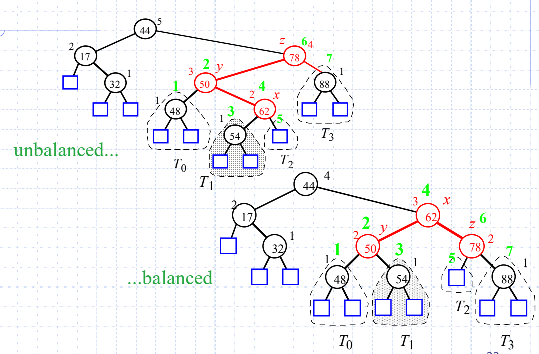
- let (*a*,*b*,*c*) be an inorder listing of *x*, *y*, *z*
- perform the rotations needed to make *b* the topmost node of the three



21

21

Insertion Example, continued

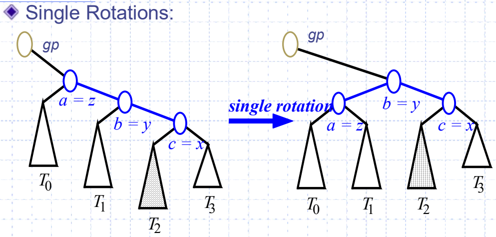


22

22

Restructuring
(as Single Rotations)

- Single Rotations:



23

23

Left Rotation

Algorithm *rotateLeft*(T, *y*)
  Input Binary Tree T and node *y* in T
  Output a left rotation around node *y* is performed

  if *T.isRoot(y)* **then** **throw** InvalidLeftRotation
  p ← *T.parent(y)*
  gp ← *T.parent(p)*

  T.setRightChild(p, T.leftChild(y))
  T.setParent(T.leftChild(y), p)

  T.setLeftChild(y, p)
  T.setParent(p, y)

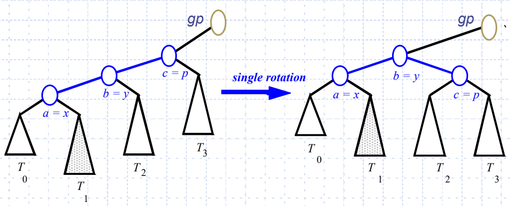
  if *T.isRoot(p)*
    then *T.setRoot(y)*
    else
      if *T.rightChild(gp) = p*
      then *T.setRightChild(gp, y)*
      else *T.setLeftChild(gp, y)*

      T.setParent(y, gp)

24

24

Exercise: Do Right Rotation



25

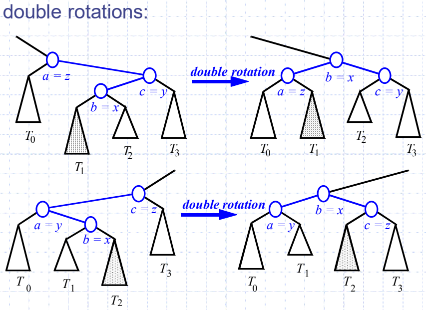
Right Rotation

```
Algorithm rotateRight(T, y)
    Input Binary Tree T and node y in T
    Output a right rotation around node y is performed

     if  T.isRoot(y)  then   throw InvalidRightRotation
     p ← T.parent(y)
     gp ← T.parent(p)

     T.setLeftChild(p, T.rightChild(y))
     T.setParent(T.rightChild(y), p)

     T.setRightChild(y, p)
     T.setParent(p, y)

     if  T.isRoot(p)
        then   T.setRoot(y)
        else
             if   T.rightChild(gp) = p
             then   T.setRightChild(gp, y)
             else   T.setLeftChild(gp, y)

             T.setParent(y, gp)
```


26

Restructuring
(as Double Rotations)

- double rotations:



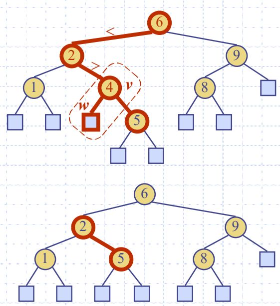
Right Left

Left Right

27

Recall
Deletion from a BST

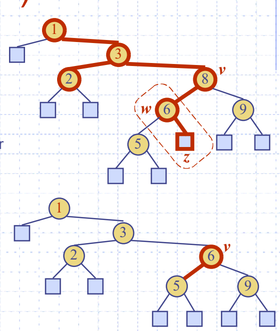
- To perform operation removeElement(*k*), we search for key *k*
- Assume key *k* is in the tree, and let *v* be the node storing *k*
- Two cases:
  - Node *v* has a leaf child *w*
  - Node *v* has no leaf child
- If node *v* has a leaf child *w*, we remove *v* and *w* from the tree with operation removeAboveExternal(*w*)
- Example: remove 4



28

Deletion (Case 2)

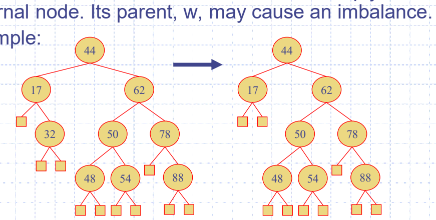
- We consider the case where the key *k* to be removed is stored at a node *v* whose children are both internal
  - we find the internal node *w* that precedes *v* in an in-order traversal
  - we copy *key*(*w*) into node *v*
  - we remove node *w* and its right child *z* (which must be a leaf) by means of operation remove(*w*)
- Example: remove 8



29

Removal in an AVL Tree

- Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, w, may cause an imbalance.
- Example:

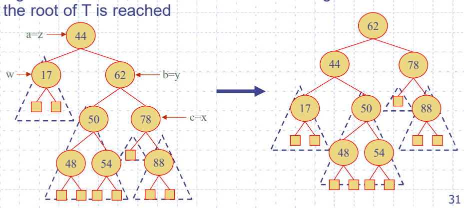


before deletion of 32 after deletion

30

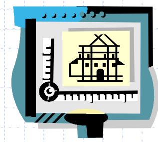
Rebalancing after a Removal

- Let *z* be the first unbalanced node encountered while travelling up the tree from w. Also, let y be the child of z with the larger height, and let x be the child of y with the larger height.
- We perform restructure(x) to restore balance at z.
- As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached



31

Running Times for AVL Trees



- a single restructure is O(1)
  - using a linked-structure binary tree
- find is O(log *n*)
  - height of tree is O(log *n*), no restructures needed
- insert is O(log *n*)
  - initial find is O(log *n*)
  - Restructuring up the tree, maintaining heights is O(log *n*)
- remove is O(log *n*)
  - initial find is O(log *n*)
  - Restructuring up the tree, maintaining heights is O(log *n*)

32

Advantages of Binary Search Trees

- When implemented properly, BST's
  - perform insertions and deletions faster than can be done on Linked Lists
  - perform any find with the same efficiency as a binary search on a sorted array
  - keep all data in sorted order (eliminates the need to sort)

33

Main Point

2. The elimination of the worst case behavior of a binary search tree is accomplished by ensuring that the tree remains balanced, that is, the insert and delete operations do not allow any leaf to become significantly deeper than the other leaves of the tree. Regular experience of pure consciousness during the TM technique reduces stress and restores balance in the physiology. The state of perfect balance, pure consciousness, is the basis for balance in activity.

34

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. In an AVL tree, the heights of the children of each node differ by at most 1; this maintains balance in the tree so search, insert, and delete can be done efficiently in O(log n) time.

2. In order to maintain balance in the tree as a whole, a tri-node restructuring is done whenever the tree gets out of balance, i.e., whenever the heights of the children of a node differ by more than 1.

35

3. **Transcendental Consciousness** is the state of perfect balance, the foundation for wholeness of life, the basis for balance in activity.
4. **Impulses within Transcendental Consciousness**: The dynamic natural laws within this unbounded field create and maintain the order and balance in creation.
5. **Wholeness moving within itself** : In Unity Consciousness, one experiences the dynamics of pure consciousness that gives rise to the laws of nature, the order and balance in creation, as nothing other than one's own Self.

36