Lecture 8:
Ordered Dictionaries

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

1

Hash Tables

◆ Hash tables are a highly efficient implementation of the Dictionary ADT
  ■ What are its disadvantages?

2

Wholeness Statement

A dictionary allows users to assign keys to elements then to access or remove those elements by key. An ordered dictionary maintains an order relation among keys allowing access to adjacent keys while supporting efficient implementation of the dictionary ADT. Each of us has access to the source of thought which is a field of perfect order, balance, and efficiency.

3

Review:
Ordered Dictionaries

◆ Keys are assumed to come from a total order.
◆ New operations:
  ■ closestKeyBefore(k)
    Return the key of the item with largest key less than or equal to *k*
  ■ closestElemBefore(k)
    Return the element of the item with largest key less than or equal to *k*
  ■ closestKeyAfter(k)
    Return the key of the item with smallest key greater than or equal to *k*
  ■ closestElemAfter(k)
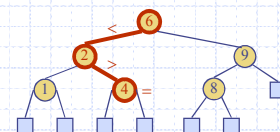   Return the element of the item with smallest key greater than or equal to *k*

4

Ordered Dictionaries

◆ What are the disadvantages of a sorted sequence as the basis of an ordered Dictionary?

5

Binary Search Trees

6

1

Binary Search Tree (§3.1.2)

- 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 $key(u) \leq key(v) \leq key(w)$
- External nodes do not store items

- An in-order traversal of a binary search tree visits the keys in increasing order
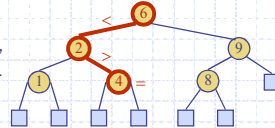
7

Search (§3.1.3)

- 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)
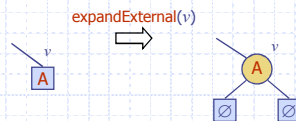
```
Algorithm findElement(k, v)
    if T.isExternal (v)
        return NO_SUCH_KEY
    if k < key(v)
        return findElement(k, T.leftChild(v))
    else if k = key(v)
        return element(v)
    else { k > key(v) }
        return findElement(k, T.rightChild(v))
```
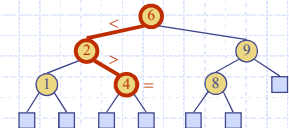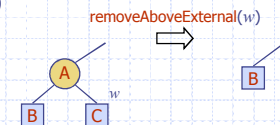
8

Recall
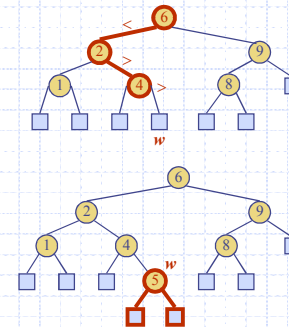## Additional Tree Update Methods

expandExternal(v)

expandExternal(v)

removeAboveExternal(w)

removeAboveExternal(w)
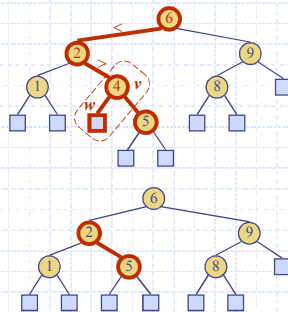
Insertion (§3.1.4)

- 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
- We insert k at node w by expanding w into an internal node using expandExternal(w)
- Example: insert 5

10

Deletion (§3.1.5)

- 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

11

Deletion (cont.)

- 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 follows v in an in-order traversal
  - we copy $key(w)$ into node v
  - we remove node w and its left child z (which must be a leaf) by means of operation removeAboveExternal(z)
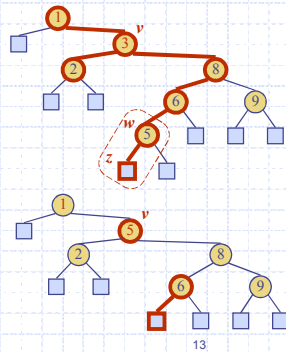- Example: remove 3

12
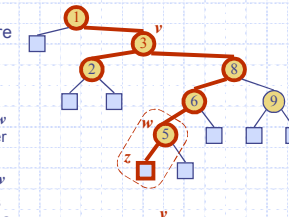
2

Deletion (cont.)

- 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 follows v in an in-order traversal
  - we copy $key(w)$ into node v
  - we remove node w and its left child z (which must be a leaf) by means of operation removeAboveExternal(z)
- Example: remove 3

13

Binary Search Trees

- What are the problems with binary search trees as the basis of a Dictionary?

14

Performance (§3.1.6)

- 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
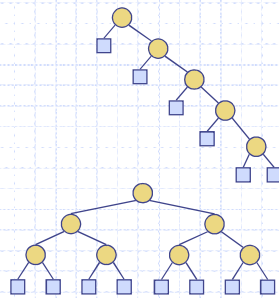
15

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.
   Such data structures reflect the following SCI principles: law of least action, principle of diving, perfect order.
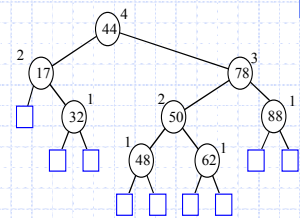
16

AVL Trees
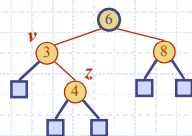
17

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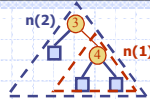
- **AVL trees are balanced.**
- An AVL Tree is a *binary search tree* such that for every internal node v of T, the *heights of the children of v can differ by at most 1*.

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

18

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 $1 \leq h-2i \leq 2$
  - 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)

19

## Recall
Insertion into a BST (§3.1.4)

- 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
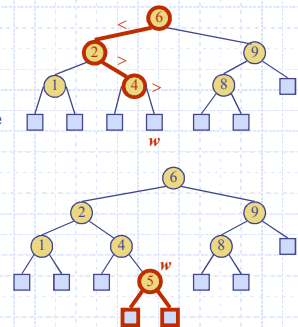
20

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
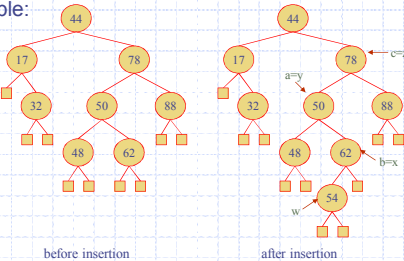
21

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

(other two cases are symmetrical)

case 2: double rotation (a right rotation about *c*, then a left rotation about *a*)

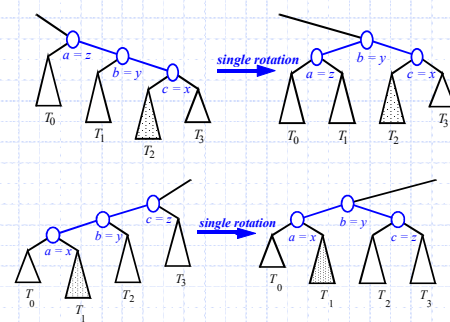case 1: single rotation (a left rotation about *a*)

22

Insertion Example, continued

unbalanced...

...balanced
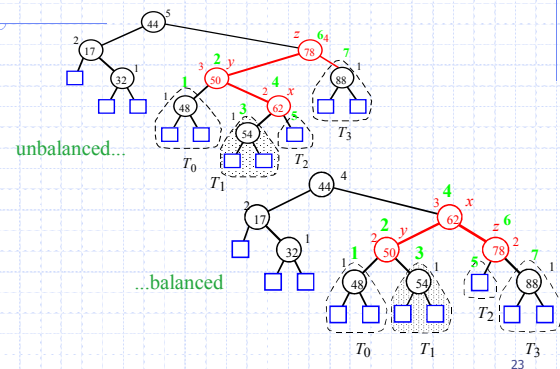
23

Restructuring
(as Single Rotations)

- Single Rotations:

single rotation

single rotation

24

4

## Restructuring (as Double Rotations)

- double rotations:

Right Left

Left Right
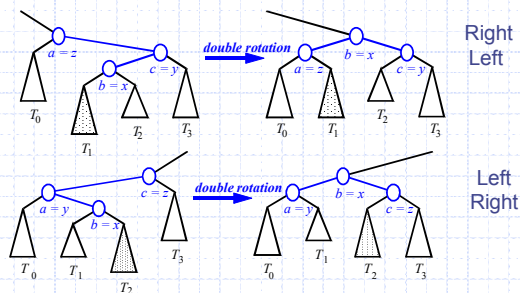
25

Recall — Deletion from a BST (§3.1.5)

- 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
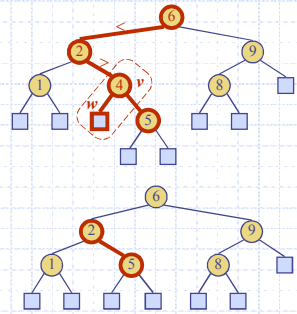

26

BST Deletion (cont.)

- 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 follows v in an in-order traversal
  - we copy $key(w)$ into node v
  - we remove node w and its left child z (which must be a leaf) by means of operation removeAboveExternal(z)
- Example: remove 3
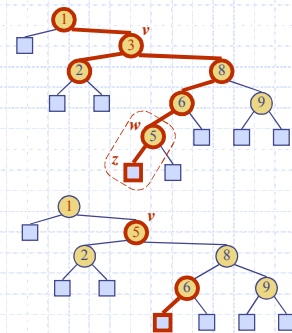
27

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
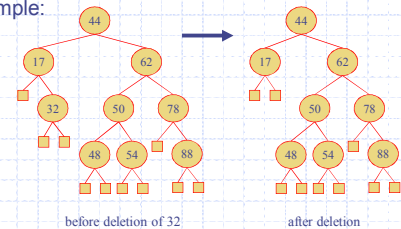
28

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
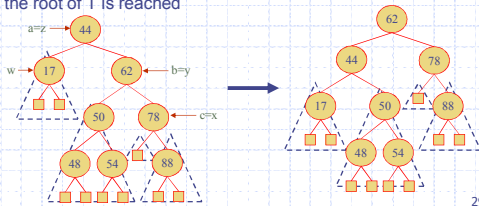
29

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)

30

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)

31

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.

32

(2,4) Trees
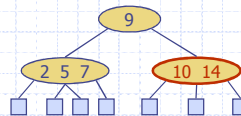
33

Outline and Reading

◆ Multi-way search tree (§3.3.1)
- Definition
- Search

◆ (2,4) tree (§3.3.2)
- Definition
- Search
- Insertion
- Deletion

◆ Comparison of dictionary implementations

34

Multi-Way Search Tree

◆ A multi-way search tree is an ordered tree such that
- Each internal node has at least two children and stores $d-1$ key-element items (k_i, o_i), where d is the number of children
- For a node with children $v_1 v_2 \ldots v_d$ storing keys $k_1 k_2 \ldots k_{d-1}$
  - keys in the subtree of v_1 are less than k_1
  - keys in the subtree of v_i are between k_{i-1} and k_i ($i = 2, \ldots, d-1$)
  - keys in the subtree of v_d are greater than k_{d-1}
- The leaves store no items and serve as placeholders
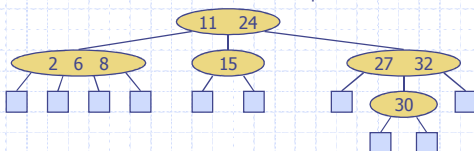
35

## Multi-Way Inorder Traversal

◆ We can extend the notion of inorder traversal from binary trees to multi-way search trees
◆ Namely, we visit item (k_i, o_i) of node v between the recursive traversals of the subtrees of v rooted at children v_i and v_{i+1}
◆ An inorder traversal of a multi-way search tree visits the keys in increasing order

36
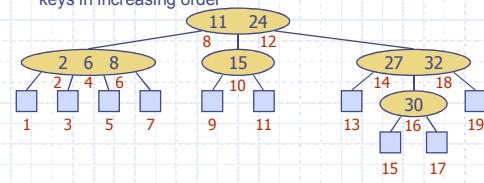
6

Multi-Way Searching

- Similar to search in a binary search tree
- A each internal node with children $v_1 v_2 \ldots v_d$ and keys $k_1 k_2 \ldots k_{d-1}$
  - $k = k_i$ ($i = 1, \ldots, d-1$): the search terminates successfully
  - $k < k_1$: we continue the search in child v_1
  - $k_{i-1} < k < k_i$ ($i = 2, \ldots, d-1$): we continue the search in child v_i
  - $k > k_{d-1}$: we continue the search in child v_d
- Reaching an external node terminates the search unsuccessfully
- Example: search for 30
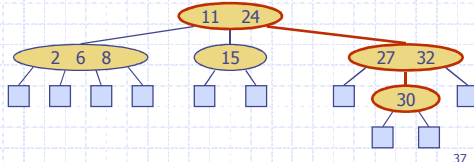
37

(2,4) Tree
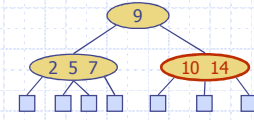
38

(2,4) Tree

- A (2,4) tree (also called 2-4 tree or 2-3-4 tree) is a multi-way search tree with the following properties
  - Node-Size Property: every internal node has at most four children
  - Depth Property: all the external nodes have the same depth
- Depending on the number of children, an internal node of a (2,4) tree is called a 2-node, 3-node or 4-node
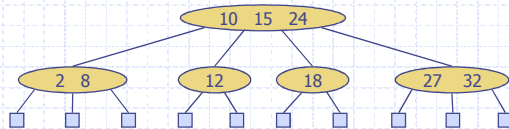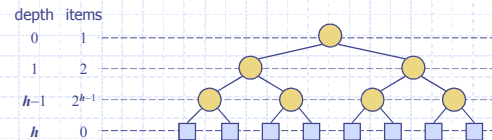

39

Height of a (2,4) Tree

- Theorem: A (2,4) tree storing n items has height $O(\log n)$
  Proof:
  - Let h be the height of a (2,4) tree with n items
  - Since there are at least 2^i items at depth $i = 0, \ldots, h-1$ and no items at depth h, we have
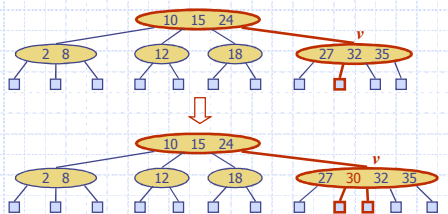 $$n \geq 1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$$
  - Thus, $h \leq \log (n+1)$
- Searching in a (2,4) tree with n items takes $O(\log n)$ time

depth	items
0	1
1	2
$h-1$	2^{h-1}
h	0

40

Insertion

- We insert a new item (k, o) at the parent v of the leaf reached by searching for k
  - We preserve the depth property but
  - We may cause an overflow (i.e., node v may become a 5-node)
- Example: inserting key 30 causes an overflow

41

Overflow and Split

- We handle an overflow at a 5-node v with a split operation:
  - let $v_1 \ldots v_5$ be the children of v and $k_1 \ldots k_4$ be the keys of v
  - node v is replaced nodes v' and v''
    - v' is a 3-node with keys $k_1 k_2$ and children $v_1 v_2 v_3$
    - v'' is a 2-node with key k_4 and children $v_4 v_5$
  - key k_3 is inserted into the parent u of v (a new root may be created)
- The overflow may propagate to the parent node u
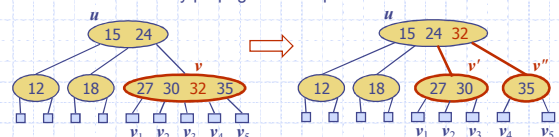
42

Analysis of Insertion

Algorithm *insertItem*(*k*, *o*)

1. We search for key *k* to locate the insertion node *v*

2. We add the new item (*k*, *o*) at node *v*

3. **while** *overflow*(*v*)
   if *isRoot*(*v*)
      create a new empty root above *v*
   v ← *split*(*v*)

- Let *T* be a (2,4) tree with *n* items
  - Tree *T* has $O(\log n)$ height
  - Step 1 takes $O(\log n)$ time because we visit $O(\log n)$ nodes
  - Step 2 takes $O(1)$ time
  - Step 3 takes $O(\log n)$ time because each split takes $O(1)$ time and we perform $O(\log n)$ splits
- Thus, an insertion in a (2,4) tree takes $O(\log n)$ time

43

Example:

- Insert the following into an initially empty 2-4 tree in this order:
 (16, 5, 22, 45, 2, 10, 18, 30, 50, 12, 1)

44

Deletion

- We reduce deletion of an item to the case where the item is at the node with leaf children
- Otherwise, we replace the item with its inorder successor (or, equivalently, with its inorder predecessor) and delete the latter item
- Example: to delete key 24, we replace it with 27 (inorder successor)
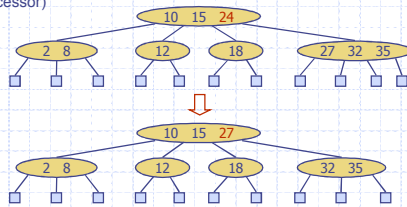

45

Underflow and Fusion

- Deleting an item from a node *v* may cause an underflow, where node *v* becomes a 1-node with one child and no keys
- To handle an underflow at node *v* with parent *u*, we consider two cases
- Case 1: the adjacent siblings of *v* are 2-nodes
  - Fusion operation: we merge *v* with an adjacent sibling *w* and move an item from *u* to the merged node *v'*
  - After a fusion, the underflow may propagate to the parent *u*
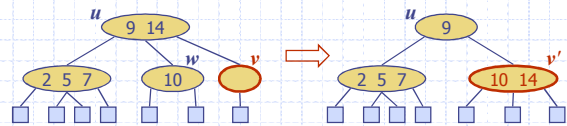
46

Underflow and Transfer

- To handle an underflow at node *v* with parent *u*, we also consider a second case
- Case 2: an adjacent sibling *w* of *v* is a 3-node or a 4-node
  - Transfer operation:
    1. we move an item from *u* to *v*
    2. we move an item from *w* to *u*
  - After a transfer, no underflow occurs
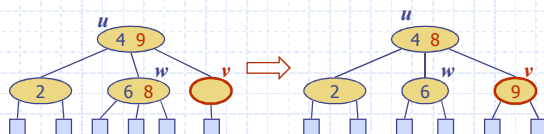
47

Analysis of Deletion

- Let *T* be a (2,4) tree with *n* items
  - Tree *T* has $O(\log n)$ height
- In a deletion operation
  - We visit $O(\log n)$ nodes to locate the node from which to delete the item
  - We handle an underflow with a series of $O(\log n)$ fusions, followed by at most one transfer
  - Each fusion and transfer takes $O(1)$ time
- Thus, deleting an item from a (2,4) tree takes $O(\log n)$ time

48

Main Point

3. By introducing some flexibility in the data content of each node, all leaf nodes of a (2,4) Tree can be kept at the same depth.
Stability and adaptability are fundamentals of progress and evolution in nature.

49

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. In a (2,4) tree, each node has 2, 3, or 4 children and all leaf nodes are at the same depth so search, insertion, and deletion are efficient, O(log n).

2. The insert and delete operations in a (2,4) tree are carefully structured so that the activity at each node promotes balance in the tree as a whole. Each node contributes to the dynamic balance by giving and receiving keys during the splitting and fusion of nodes.

50

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.

51