Part 6. Trees (2)

CS 200 Algorithms and Data Structures

Outline

- 2-3 Trees
- 2-3-4 Trees
- Red-Black Trees
- AVL Trees

Balanced Search Trees

- A search of a binary search tree can be as inefficient as a sequential search of a linked list.
  - Balanced search trees address this problem
- Insert and delete items without deteriorating the tree’s balance while maintaining a minimum-height search tree.
- A type of binary search tree where costs are guaranteed to be logarithmic

2-3 search trees

- A tree in which each internal node (nonleaf) has either two or three children, and all leaves are at the same level.
- A 2-3 search tree is not a binary tree.

Rules for placing items

- A 2-node (with two children) must contain a single data item whose search key is greater than the left child’s search key(s) and less than the right child’s search key(s).
- A 3-node (with three children) must contain two data items whose search keys S and L satisfy the following relationships.
  - S is greater than the left child’s search key(s) and less than the middle child’s search key(s).
  - L is greater than the middle child’s search key(s) and less than the right child’s search key(s).
- A leaf may contain either one or two data items.
Placing items in a 2-node

Placing items in a 3-nodes

Traversing a 2-3 tree

Searching a 2-3 tree (1/2)

Searching a 2-3 tree (2/2)

Efficiency

- A binary search tree with \( n \) nodes cannot be shorter than \( \lceil \log(n+1) \rceil \)
- A 2-3 tree with \( n \) nodes cannot be taller than \( \lceil \log(n+1) \rceil \)
- A node in a 2-3 tree has at most two items.
Is searching a 2-3 tree more efficient than a BST?

- After all, the nodes of a 2-3 tree can have three children
  - Shorter than the shortest possible binary search tree!
- More comparisons for each of the node.
  (twice the number of comparisons)
  - Approximately equal to the number of comparisons in BST that is as balanced as possible.

Then WHY 2-3 trees?

If you add new values to balanced BST, you can lose the balance of the tree.

The insertion algorithm

- Locate the leaf at which the search for \( I \) would terminate.
- Insert the new item into the leaf \( I \)
  - Case 1. If the leaf \( I \) contains two items: you are done
  - Case 2. If the leaf \( I \) contains three items: must split into \( n1 \) and \( n2 \).
    - Split Case A: Split a leaf node
    - Split Case B: Split an internal node
    - Split Case C: Split a root node

Split Case A: Splitting a left leaf node
Split Case A: Splitting a right leaf node

Split Case B: Splitting a left internal node

Split Case B: Splitting a right leaf node
Split Case C: Splitting the root of a 2-3 tree

Growing the heights

- If every node on the path from the root of the tree to the leaf (into which the new item is inserted) contains two items.
- The recursive process of splitting a node and moving an item up to the node’s parent will reach the root $r$.
- Split $r$ into $r_1$ and $r_2$.
- Create a new node $r$ with a middle item.
- The new node becomes a new root of the tree.

Example: Inserting into a 2-3 tree

- You can insert items into the tree while maintaining its shape.
- Insert 39, 38, 37, 36, 35, 34, 33, 32

Insert 39

Case 1. node I contains two items
Finished
### Insert 38

- **Case 2**: Leaf node became overcrowded
  - Split Case A: Move up the middle one
  - Finished

- **Case 2**: Internal node became overcrowded
  - Split case B: Internal node: Move up the middle one

### Insert 37

- **Case 1**: 37 is added to a leaf node and the leaf node has two items
  - Finished

### Insert 36 (1/2)

- **Case 2**: Leaf node became overcrowded
  - Split case A: Move up the middle one
  - Case 2. Internal node became overcrowded
  - Split case B: Internal node: Move up the middle one

### Insert 36 (1/2)

- **Case 2**: Leaf node became overcrowded
  - Split case A: Move up the middle one
  - Case 2. Internal node became overcrowded
  - Split case B: Internal node: Move up the middle one
Case 2. Leaf node became overcrowded
  \( \rightarrow \) Split Case A: Move up the middle one
  Case 2. Internal node became overcrowded
  \( \rightarrow \) Split case B: Internal node: Move up the middle one
Finished
3/23/12

Insert 33

Case 1: 33 is added to a leaf node

Finished

Insert 32

Case 2: Leaf node became overcrowded

→ Split Case A: Move up the middle one

Case 2: Internal node became overcrowded

→ Split Case B: Internal Node: Move up the middle one

Insert 32

Case 2: Leaf node became overcrowded

→ Split Case A: Move up the middle one

Case 2: Internal node became overcrowded

→ Split Case B: Internal Node: Move up the middle one

Insert 32

Case 2: Leaf node became overcrowded

→ Split Case A: Move up the middle one

Case 2: Internal node became overcrowded

→ Split Case B: Internal Node: Move up the middle one

Insert 32

Case 2: Leaf node became overcrowded

→ Split Case A: Move up the middle one

Case 2: Internal node became overcrowded

→ Split Case B: Internal Node: Move up the middle one

Finished

Case 2:

Root node became overcrowded

→ Split Case C: Move up the middle one

Recursive Method

Insert 32

Case 2: Leaf node became overcrowded

→ Split Case A: Move up the middle one

Case 2: Internal node became overcrowded

→ Split Case B: Internal Node: Move up the middle one

Case 2: Root node became overcrowded

→ Split Case C: Root: Move up the middle one

Finished

insertItem(2/2)

if (n is not a leaf){
    n1 becomes the parent of n's two leftmost children
    n2 becomes the parent of n's two right most children
}

Move the item in n that has the middle search-key value up to p

if (p now has three items){
    split(p)
}

Recursive Method
Deletion Algorithm

- **Locate the node** \( n \)
  - Case 1: Is the node a leaf node?
  - Case 2: Is the node an internal node?
    - Find inorder successor and swap it
    - Deletion will be in the leaf now.
  - Fix case A.
    - If an item will be left in the node: done
  - Fix case B.
    - A. If sibling has two items: redistributing values
    - B. If no sibling has two items: merging a leaf

Deletion begins at a leaf.

insertItem(1/2)

```java
insertItem(t, newItem)
  first step: Locate the leaf node \( \mathcal{L} \)
  Let \( x \) be the search key of \( \mathcal{L} \).
  Locate the leaf node \( \mathcal{L} \) in which \( x \) belongs.
  Add \( \mathcal{L} \) to leaf Node.
  if (leaf Node now has three items)
    split(leaf Node)
  }

Case 1: Leaf node \( \mathcal{L} \) has 2 items
split (in out n:TreeNode)
  if (n is the root){
    Create a new node \( p \)
  } else{
    Let \( p \) be the parent of \( n \).
  }
  Replace node \( n \) with two nodes, \( n_1 \) and \( n_2 \), so that \( p \) is their parent.
  Give \( n_1 \) the item in \( n \) with the smallest search key.
  Give \( n_2 \) the item in \( n \) with the largest search key.

Redistributing values and children

Merging internal nodes

A. Redistributing Values:
   If sibling has two items

B. Merging a Leaf:
   If no sibling has two items
Deleting Root

Deletng Root: Moving 80 down

Deletng 70: Delete from the leaf

Deletng 70: swap with inorder successor

Deletng 100: Delete value from leaf

Case 2: 70 is NOT a leaf node

Fix Case B: No sibling has two items

Merge a leaf

Fix Case B: Sibling has two item

Redistribute values
Deleting 100-Does it work?

Deleting 80: swap with inorder successor

Case 2: 80 is NOT a leaf node
Swap with inorder successor

Deleting 80: Delete value from leaf

Fix Case 2: No sibling has two items
Merge a leaf

Deleting 80: Merge by moving 90 down

Recursively called
Fix Case 2: No sibling has two items
Merge an internal node (with children)

Deleting 80: Merge 50 down

After Deleting 80

Finished
High level algorithm (1/2)

```java
deleteItem(n:TreeNode, searchKey in searchKey)
    Attempt to locate item theItem whose search key equals searchKey
    if (theItem is present):{ 
        if (theItem is not in a leaf):
            Swap item theItem with its inorder successor, which will be in a leaf theLeaf
        Delete item theItem from leaf theLeaf
        if (theLeaf now has no items): 
            fix(theadle)
        return true
    } else: 
        return false 
```

### Outline

- 2-3 Trees
- 2-3-4 Trees
- Red-Black Trees
- AVL Trees

### 2-3-4 Trees

- 2-nodes, 3-nodes, and 4-nodes
  - 4-nodes: nodes that have four children
- T is a 2-3-4 tree of height h if
  - T is empty
  - T is of the form
    ```
    /\n   / \
  T_L  T_M  T_R
    ```

### Rules for Placing Data Items in the Nodes of a 2-3-4 Tree

- A 2-node must contain a single data item whose search key satisfies the relationship in a 2-3 Tree
- A 3-node must contain two data items whose search keys satisfy the relationship in a 2-3 Tree
- A 4-node must contain three data items, whose search keys S, M, and L satisfy the following relationship:
  1. left child’s search key(s) < S < middle-left child’s search key(s)
  2. middle-left child’s search key(s) < M < middle-right child’s search key(s)
  3. middle-right child’s search key(s) < L < right child’s search key(s)

### A 4-node in a 2-3-4 tree

```
S < Search keys < M
```

```
S < Search keys < M
M < Search keys < L
```
Searching and traversing a 2-3-4 tree

- Simple extension of the corresponding algorithms for a 2-3 tree
- Adding comparisons for the 4-node

Inserting into a 2-3-4 tree

- Algorithm is similar to the insertion into a 2-3 tree.
  - 2-3 tree: Split a node by moving one of its items up to its parent node.
  - 2-3-4 tree: As soon as the search process encounters 4-nodes, it splits the 4-node.

Insert 20

```
10 30 60
```

While determining the insertion point, you encounter the 4-node
Split by moving the middle value 30 up
Keep searching
Add 20

Insert 20

```
10 20 60
```

While determining the insertion point, you encounter the 4-node
Split by moving the middle value 30 up
Keep searching
Add 20

Insert 50

```
10 20 30 50 60
```

Insert 40

```
10 20 30 40 50 60
```
While determining the insertion point, you encounter the 4-node
Split by moving the middle value 50 up
Keep searching
Add 70

While determining the insertion point, you encounter the 4-node
Split by moving the middle value 70 up
Keep searching
Add 90

Insert 100

Splitting a 4-node root during insertion
Splitting a 4-node whose parent is a 2-node during insertion

Splitting a 4-node whose parent is a 3-node during insertion

Deleting from 2-3-4 tree

- Locate the node
- Swap with inorder successor
  - Deletion should be always in the leaf node
- If the leaf is a 3-node or 4-node, remove item
- If you ensure that the item you delete does not occur in a 2-node, you can delete the item in one pass through the tree from root to leaf

2-3 and 2-3-4 trees

- 2-3 and 2-3-4 trees are easy-to-maintain in balance
  - The reduction in height is offset by the increased number of comparisons that the search algorithm may require at each node.
  - The 2-3-4 tree needs only one pass through the tree for its insertion and deletion.

Trees with MANY children nodes?

- Tree with many child nodes (e.g. 100 children) requires more comparisons at each node to determine which subtree to search.
  - It is appropriate for external storage.
    - Moving from node to node is far more expensive than comparing the data values.

Outline

- 2-3 Trees
- 2-3-4 Trees
- Red-Black Trees
- AVL Trees
Red-Black Trees

- Represent a 2-3-4 tree and retain the advantages of a 2-3-4 tree without the storage overhead.
  - Represent each 3-node and 4-node in a 2-3-4 tree as an equivalent binary tree.
  - Use red and black child references to distinguish between original 2-nodes, and 2-nodes that were generated from 3-nodes and 4-nodes.
  - 2-nodes from original 2-3-4 tree : black
  - 2-nodes those result from splitting 3 and 4-nodes : red

Red-black representation of a 4-node

Red-black representation of a 3-node

2-3-4 tree to a Red-black tree

2-3-4 tree to a Red-black tree
Searching and traversing a red-black tree

- A red-black tree is a binary search tree
  - Search and traverse it with binary search tree algorithms
  - Ignore the color of the references

Inserting into a red-black tree

- Split each 4-node that you encounter
  Case 1: 4-node that is a root
  Case 2: 4-node whose parent is a 2-node
  Case 3: 4-node whose parent is a 3-node
  There is no 4-node whose parent is a 4-node. WHY?

Inserting Case 1: 4-node that is a root

Inserting Case 2: 4-node whose parent is a 2-node

Inserting Case 2: 4-node whose parent is a 2-node
Inserting Case 3: 4-node whose parent is a 3-node

Example 1

Example 2

Inserting Case 3: Example 1

Example 1

Color Changes

4-node whose parent is a 3-node

Inserting Case 3: Example 2

Example 2

Inserting Case 3: Example 2

Rotation and Color Changes

Example 2

4-node whose parent is a 3-node

Inserting Case 3: Example 3

Example 3

Inserting Case 3: Example 3

Rotation and Color Changes

Example 3

4-node whose parent is a 3-node

Inserting Case 3: Example 4

Example 4

Inserting Case 3: Example 4

Rotation and Color Changes

Example 4

4-node whose parent is a 3-node
Inserting Case 3: 4-node whose parent is a 3-node

Deleting from a red-black tree

• This is similar to the 2-3-4 deletion algorithm
  – Frequently requires only color changes
  – More efficient than the corresponding operations on a 2-3-4 tree

Outline

• 2-3 Trees
• 2-3-4 Trees
• Red-Black Trees
• AVL Trees

AVL Tree

• Named after its inventors Adelson-Velskii and Landis
• A balanced binary search tree
• Almost as efficient as a minimum-height binary search tree

AVL Tree

• Maintains a binary search tree with a height close to the minimum
  1. Insert/Delete nodes following the algorithm of the BST.
  2. Monitor the shape.
     • Determine whether any node in the tree has left and right subtrees whose heights differ by more than 1.
  3. If it is not a balanced binary search tree, rotate the tree to rebalance the tree
Before and after a single left rotation that decreases the tree’s height:

Before:

```
    20
   /  \
  40   h
    \
    h + 1
```

After:

```
    20
   /  \
  40   h
   / \
  20  h
 \
 h + 1
```

Rotation and Height? (1/2):

After the rotating, the height of the tree is reduced.

Rotation and Height? (2/2):

After the rotation, the height of the tree is NOT reduced.

Rotations might not affect the tree’s height.
Double Rotation: Before

Double Rotation: During

Double Rotation: After