CS200: Balanced Search Trees

Walls & Mirrors Chapters 13.1
Balanced Search Trees

- The efficiency of binary search trees is related to the tree’s height
  - Height of a binary search tree of n items
    - Maximum possible height: n
    - Minimum possible height: \( \lceil \log_2(n + 1) \rceil \)
    - Locating an item in a BST requires between n and \( \lceil \log_2(n + 1) \rceil \) comparisons
      - BST search can be as inefficient as a sequential search (O(n)) and as efficient as a binary search of an array (O(log n))

- Height of a binary search tree is sensitive to the order of insertions and deletions

- Balanced search trees: maintain search tree height to minimum
2-3 Trees

- In a 2-3 tree each internal node has either 2 or 3 children
  - 2-nodes: one data item and two children
  - 3-nodes: two data items and three children
**2-3 Trees**

**2 node**: a single item whose search key is
- Greater than the left child’s search key(s)
- Less than the right child’s key(s)

**3-node**: two items whose search keys S and L satisfy:
- S > left child’s search key(s), less than the middle child’s search key(s)
- L > middle child’s search key(s), less than the right child’s search key(s)
Properties of 2-3 Trees

- A leaf may contain either one or two data items
- All leaf nodes are at the same level
  - A leaf has 1 or 2 elements in a node
- Never taller than a minimum-height binary tree
  - A 2-3 tree with n nodes has height less than \( \lceil \log_2(n + 1) \rceil \)
2-3 Trees
2-3 Trees vs Binary Trees

- A 2-3 tree with n nodes never has a height greater than \( \lceil \log_2(n + 1) \rceil \)
  - This is the minimal height of a binary tree with n nodes
- Insertion and deletion of nodes does not “unbalance” a 2-3 tree as is the case with BSTs
2-3 Trees

(a) 30
   /   /
  10   50
 /     /
20 40
70
(b) 50
   /   /
  30   70 90
 /     /
10   20
40 60 80 100
2-3 Trees

After a sequence of insertions: insert 39, 38, … 33, 32
A 2-3 node

```java
public class TreeNode<T> {
    private T smallItem;
    private T largeItem;
    private TreeNode leftChild;
    private TreeNode midChild;
    private TreeNode rightChild;
    ...
}
```

If a node has only 2 children then you can set the rightChild to null
Traversing 2-3 Trees

inorder(in ttTree:twoThreeTree)
    if (ttTree’s root node is a leaf) {
        visit the data item(s) }
else if (root has two data items) {
    inorder(left subtree)
    visit first data item
    inorder(middle subtree)
    visit second data item
    inorder(right subtree)
else {  // root has one data item
    inorder(left subtree)
    visit data item
    inorder (right subtree) }
Searching 2-3 Trees

Search for 125
Searching 2-3 Trees

- Analogous to binary search trees
- As efficient as searching the shortest binary search tree
  - Number of comparisons required to search a 2-3 tree:
    - Approximately equal to the number of comparisons required to search a binary search tree that is as balanced as possible
  - Searching a 2-3 tree is $O(\log_2 n)$
Inserting 39: space available in leaf
Inserting 38: no space in leaf, but space in parent
After inserting 37
Inserting 36

No space in leaf or immediate parent…
Exercise

- Insert 35, 34, 33
To insert an item:

- Locate the leaf at which the search would terminate
- Insert the new item into the leaf
- If the leaf now contains only two items, you are done
- If the leaf now contains three items, split the leaf into two nodes, \( n_1 \) and \( n_2 \)
- If the parent contains two items you are done.
- Otherwise - it contains three items and has 4 children.
When an internal contains 3 items:

- Split the node into two nodes
- Accommodate the node’s children
Insert (cont)

- When the root contains three items
  - Split the root into two nodes
  - Create a new root node
Delete

Deleting 70

(a)
Delete

Deleting 70

(b) 80 90
60  -  100
Delete value from leaf

(c) 80 90
60  -  100
Merge nodes by deleting empty leaf and moving 80 down

(d) 90
60  80  100
Delete

Next step:
Delete 100
Delete 100

(a)  
```
90
/   \
|    |
60   80
```
Delete value from leaf

(b)  
```
90
/   \
|    |
60   80
```
Doesn’t work

(c)  
```
80
/   \
|    |
60   90
```
Redistribute

(d)  
```
50
/   \
/    |
30   80
/     |
10  20  60
```

Delete

Deleting 80

(a) Swap with inorder successor
Delete

Deleting 80

Delete value from leaf

Merge by moving 90 down and removing empty leaf

Node becomes empty
Delete

Deleting 80

(d) 
Root becomes empty

Merge: move 50 down, adopt empty leaf’s child, remove empty node

Remove empty root
Delete

- Locate the node that needs to be deleted
- If it’s not a leaf: find its inorder successor and swap
- If the leaf contains two items you can just delete one.
- Otherwise deletion would leave an empty node: need to do some more work
Delete

(a)

\[
\begin{align*}
&\text{P} & \Rightarrow & \text{L} \\
&\text{S} & \text{L} & \Rightarrow & \text{S} & \text{P} \\
&\text{Sibling} & \text{Leaf} & \Rightarrow & \text{Sibling} & \text{Leaf}
\end{align*}
\]

(b)

\[
\begin{align*}
&\text{S} & \Rightarrow & \text{S} & \text{L} \\
&\text{L} & \Rightarrow & \text{S} & \text{L} \\
&\text{Sibling} & \text{Leaf} & \Rightarrow & \text{Sibling} & \text{Leaf}
\end{align*}
\]
Delete

(c) Delete

Redistribute

Empty node n

(d) Delete

Merge

Empty node n

CS200

a b c

a b c

a b c
Delete

- If the root becomes empty simply delete it
Complexity of 2-3 Trees

- Insert?
- Delete?

- Sufficient to consider only the time needed to locate location to insert or item to delete
  - Search has logarithmic efficiency of a binary search - Why?
A 2-3 tree is guaranteed to have a height less than $\lceil \log_2(n + 1) \rceil$.

A 2-3 will usually have reduced height in comparison to a perfect binary search tree, but requires more time to go through each node.
B-Trees

2-3 tree are a special case of B-trees:

The B-tree's creators, Rudolf Bayer and Ed McCreight, have not explained what, if anything, the B stands for...
2-3-4 Trees

- Another special case of B-trees
- Algorithms are similar to 2-3 trees
- Some improvements over 2-3 trees:
  - Insert/delete in a single pass from the root
  - The basis for red-black trees which implement 2-3-4 trees using binary trees
2-3-4 Trees

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

- Advantages
  - balanced
  - insertion and deletion operations use only one pass from root to leaf

- Disadvantage
  - more storage than a binary search tree

- Red-Black trees
  - A representation of 2-3-4 trees using binary trees
  - The advantages of a 2-3-4 tree, without the storage cost
Red-Black Trees

- Represent each 3-node and 4-node in a 2-3-4 tree as an equivalent binary tree.
Red-Black Trees

- Red-Black tree is a binary search tree.
- Search and traversal are the same as a BST.
- Insert and delete – need to translate the 2-3-4 operations
AVL Trees

- Balanced binary search trees
- Named after Adelson-Velsky and Landis
- Can be searched almost as efficiently as a minimum-height binary search tree (same big-O bounds)

Basic strategy:
- After each insertion or deletion
  - Check if the tree is still balanced
  - Restore balance if necessary
AVL Trees

- Rotations restore balance:

(a) 20
   10 40
   30 50
   60

(b) 40
   20 50
   10 30
   60

single rotation
AVL Trees

- Rotations restore balance:

(a) 40
  20
  10
  30
  25
  35
  22

(b) 40
  30
  10
  20
  25
  35
  60

(c) 30
  20
  22
  10
  25
  35
  50
  60

double rotation
AVL Trees

- Advantage
  - Efficient operations

- Disadvantage
  - AVL tree implementation more difficult than other search trees. Fine for coding libraries