Contents

- Balanced Search Trees
- 2-3 Trees
- 2-3-4 Trees
Balanced Search Trees

- Height of a binary search tree sensitive to order of insertions and removals
  - Minimum = $\log_2 (n + 1)$
  - Maximum = $n$
- Various search trees can retain balance despite insertions and removals

**FIGURE 19-1** (a) A binary search tree of maximum height; (b) a binary search tree of minimum height
2-3 Trees

- A 2-3 tree not a binary tree
- A 2-3 tree never taller than a minimum-height binary tree

A 2-node (has two children) must contain a single data item greater than left child’s item(s) and less than right child’s item(s).

A 3-node (has three children) must contain two data items, S and L, such that:
- S is greater than left child’s item(s) and less than middle child’s item(s);
- L is greater than middle child’s item(s) and less than right child’s item(s).

Leaf may contain either one or two data items.
2-3 Trees

- FIGURE 19-3 Nodes in a 2-3 tree: (a) a 2-node;

(a) Data items < S
(b) Data items > S

Data items > S
Data items > L

A 2-3 tree
Traversing a 2-3 Tree

- Traverse 2-3 tree in sorted order by performing analogue of inorder traversal on binary tree:

```c
// Traverses a nonempty 2-3 tree in sorted order.
inorder(TwThreeTree): void
    if (root node r is a leaf)
        visit the data item(s)
    else if (r has two data items)
        inorder(left subtree of 2-3 tree's root)
        visit the first data item
        inorder(middle subtree of 2-3 tree's root)
        visit the second data item
        inorder(right subtree of 2-3 tree's root)
    else // r has one data item
        inorder(left subtree of 2-3 tree's root)
        visit the data item
        inorder(right subtree of 2-3 tree's root)
```

Searching a 2-3 Tree

- Retrieval operation for 2-3 tree similar to retrieval operation for binary search tree:

```c
// Locates the value target in a nonempty 2-3 tree. Returns either the located // entry or throws an exception if such a node is not found.
findItem(TwThreeTree, target: Item): Item
    if (target is in 2-3 tree's root node r)
        // The item has been found
        treeItem = the data portion of r
        return treeItem // Success
    
```

Data Structures and Problem Solving with C++: Walls and Mirrors, Carrano and Henry, © 2013
Searching a 2-3 Tree

- Possible to search 2-3 tree and shortest binary search tree with approximately same efficiency, because:
  - Binary search tree with $n$ nodes cannot be shorter than $\log_2 (n + 1)$
  - 2-3 tree with $n$ nodes cannot be taller than $\log_2 (n + 1)$
  - Node in a 2-3 tree has at most two items
A balanced binary search tree

A 2-3 tree with the same entries
After inserting 39 into the tree:

The steps for inserting 38 into the tree:
(a) The located node has no room;
(b) the node splits; (c) the resulting tree
After inserting 37 into the tree

(a), (b), (c) The steps for inserting 36 into the tree
Inserting Data into a 2-3 Tree

(d) the resulting tree

The tree after the insertion of 35, 34, and 33
Inserting Data into a 2-3 Tree

(a) Splitting a leaf in a 2-3 tree when the leaf is a (a) left child; (b) right child

(b) Splitting an internal node in a 2-3 tree when the node is a (a) left child; (b) right child
Inserting Data into a 2-3 Tree

Summary of insertion strategy

```cpp
// Inserts a new item into a 2-3 tree whose items are distinct and differ from the
// new item.
void insertItem(23TreeNode* T, int newKeyType)
{
    23TreeNode* n = findLeafNode(T); // Locate the leaf, leafNode, in which newItem belongs
    Node* newNode = new Node(keyType); // Add new item to leafNode
    if (leafNode has three items) // split(leafNode)
        split(leafNode);
    // Splits node n, which contains two items. Note: If n is
    // not a leaf, it has four children.
    if (n is the root)
    {
        create a new node p
        let p be the parent of n
    }
    else
    {
        replace node n with two nodes, n1 and n2, so that p is their parent
        give n1 the item in n with the smallest value
    }
}
```

Splitting the root of a 2-3 tree
Inserting Data into a 2-3 Tree

(a) A 2-3 tree; 
(b), (c), (d), (e) the steps for removing 70;

Removing Data from a 2-3 Tree

(a) A 2-3 tree; 
(b), (c), (d), (e) the steps for removing 70;
Removing Data from a 2-3 Tree

(f) the resulting tree;

(a), (b), (c) The steps for removing 100 from the tree in Figure 19-15f; (d) the resulting tree
Removing Data from a 2-3 Tree

- FIGURE 19-17 The steps for removing 80 from the tree in Figure 19-16d

...
Removing Data from a 2-3 Tree

FIGURE 19-18 Results of removing 70, 100, and 80 from (a) the 2-3 tree of Figure 19-15a and (b) the binary search tree of Figure 19-5a.

Algorithm for removing data from a 2-3 tree

```cpp
// Removes the given data item from a 2-3 tree. Returns true if successful // or false if no such item exists.
removeItem(TwoThreeTree tree, dataItem item): boolean

Attempt to locate dataItem
if (dataItem is found)
{
    if (dataItem is not in a leaf)
        Swap dataItem with its inorder successor, which will be in a leaf leafNode
    // The removal always begins at a leaf
    Remove dataItem from leaf leafNode
    if (leafNode now has no items)
        fixTree(leafNode)
    return true
}
else
    return false
```
Removing Data from a 2-3 Tree

- Algorithm for removing data from a 2-3 tree

```cpp
    return false
    // Completes the removal when node n is empty by either deleting the root,
    // redistributing values, or merging nodes. Note: If n is internal, it has one child.
    fixTree(n: TwoThreeNode)
    if (n is the root)
        Delete the root
    else
    { Let p be the parent of n
        if (some sibling of n has two items)
        { Distribute items appropriately among n, the sibling, and p
        }
        else // Merge the node
        { Choose an adjacent sibling s of n
            Bring the appropriate item down from p into s
            if (n is internal)
                Move n's child to s
            Remove node n
            if (p is now empty)
                fixTree(p)
        }
    }
```
Removing Data from a 2-3 Tree

- FIGURE 19-19 (a) Redistributing values; (b) merging a leaf;

- FIGURE 19-19 (c) redistributing values and children; (d) merging internal nodes
Removing Data from a 2-3 Tree

- FIGURE 19-19 (e) deleting the root

2-3-4 Trees

- FIGURE 19-20 A 2-3-4 tree with the same data items as the 2-3 tree in Figure 19-6 b
2-3-4 Trees

- Rules for placing data items in the nodes of a 2-3-4 tree
  - 2-node (two children), must contain a single data item that satisfies relationships pictured in Figure 19-3 a.
  - 3-node (three children), must contain a single data item that satisfies relationships pictured in Figure 19-3 b.
  - ...

- 4-node (four children) must contain three data items S, M, and L that satisfy:
  - S is greater than left child’s item(s) and less than middle-left child’s item(s)
  - M is greater than middle-left child’s item(s) and less than middle-right child’s item(s)
  - L is greater than middle-right child’s item(s) and less than right child’s item(s).
  - A leaf may contain either one, two, or three data items
2-3-4 Trees

- FIGURE 19-21 A 4-node in a 2-3-4 tree

- Has more efficient insertion and removal operations than a 2-3 tree
- Has greater storage requirements due to the additional data members in its 4-nodes

```
template<class ItemType>
class QuadNode
{
private:
    ItemType smallItem, middleItem, largeItem; // Data portion
    QuadNode<ItemType>* leftChildPtr; // Left-child pointer
    QuadNode<ItemType>* leftMidChildPtr; // Middle-left-child pointer
    QuadNode<ItemType>* rightMidChildPtr; // Middle-right-child pointer
    QuadNode<ItemType>* rightChildPtr; // Right-child pointer
    // Constructors, accessor methods, and mutator methods are here.
    ...
}; // end QuadNode
```
2-3-4 Trees

- Searching and Traversing a 2-3-4 Tree
  - Simple extensions of the corresponding algorithms for a 2-3 tree
- Inserting Data into a 2-3-4 Tree
  - Insertion algorithm splits a node by moving one of its items up to its parent node
  - Splits 4-nodes as soon as it encounters them on the way down the tree from the root to a leaf

FIGURE 19-22 Inserting 20 into a one-node 2-3-4 tree (a) the original tree; (b) after splitting the node; (c) after inserting 20
2-3-4 Trees

FIGURE 19-23 After inserting 50 and 40 into the tree in Figure 19-22c

FIGURE 19-24 The steps for inserting 70 into the tree in Figure 19-23: (a) after splitting the 4-node; (b) after inserting 70
2-3-4 Trees

Figure 19-25: After inserting 80 and 15 into the tree in Figure 19-24b.

2-3-4 Trees

Figure 19-26: The steps for inserting 90 into the tree in Figure 19-25.
2-3-4 Trees

FIGURE 19-27 The steps for inserting 100 into the tree in Figure 19-26b

2-3-4 Trees

FIGURE 19-28 Splitting a 4-node root during insertion into a 2-3-4 tree
2-3-4 Trees

FIGURE 19-29 Splitting a 4-node whose parent is a 2-node during insertion into a 2-3-4 tree, when the 4-node is a (a) left child; (b) right child

FIGURE 19-30 Splitting a 4-node whose parent is a 3-node during insertion into a 2-3-4 tree, when the 4-node is a (a) left child
2-3-4 Trees

(b) FIGURE 19-30 Splitting a 4-node whose parent is a 3-node during insertion into a 2-3-4 tree, when the 4-node is a (b) middle child

(c) FIGURE 19-30 Splitting a 4-node whose parent is a 3-node during insertion into a 2-3-4 tree, when the 4-node is a (c) right child
2-3-4 Trees

- Removing Data from a 2-3-4 Tree
  - Removal algorithm has same beginning as removal algorithm for a 2-3 tree
  - Locate the node $n$ that contains the item $I$ you want to remove
  - Find $I$’s inorder successor and swap it with $I$ so that the removal will always be at a leaf
  - If leaf is either a 3-node or a 4-node, remove $I$. 