CS 112 – Introduction to Computing II

Wayne Snyder
Computer Science Department
Boston University

Today:

Efficiency of binary trees;
Balanced Trees
2-3 Trees

Next Time:

2-3 Trees continued
B-Trees and External Search
So far, we have seen that the best case for a BST is a perfect triangle, and the worst case is a linked list:

Of course it may not be possible to get a perfect triangle, but we can always create a tree in which the leaves are always within two levels of each other:

Best case: $\Theta(\log N)$

Worst case: $\Theta(N)$

What happens on average?
Efficiency of Binary Search Trees

What happens on average? The scenario would be modeled on our experiments with average case for sorting:

- Create 1000 random BSTs for each size \( N = 1, 2, 3, 4, \ldots, 100 \) (or similar parameters) by creating a random array of size \( N \) and then inserting each key into an initially-empty tree;
- Find the average cost of lookups in each tree (sum of cost of each node / \( N \));
- This simulates a situation where a random BST is created, then we repeatedly lookup keys (we could alternately do a random series of inserts, lookups, and deletes on a single tree and see what happens – results are similar).

Cost of paths:
- \( S: 1 \), \( E,X: 2 \), \( A,R: 3 \), \( C,R: 4 \)

Sum: 19

Average Cost: \( 19/7 = 2.71 \)
Efficiency of Binary Search Trees

What happens on average? The scenario would be modeled on our experiments with average case for sorting:

- Create 1000 random BSTs for each size $N = 1, 2, 3, 4, \ldots, 100$ (or similar parameters) by creating a random array of size $N$ and then inserting each key into an initially-empty tree;
- Find the average cost of lookups in each tree (sum of cost of each node / $N$);
- This simulates a situation where a random BST is created, then we repeatedly lookup keys (we could alternately do a random series of inserts, lookups, and deletes on a single tree and see what happens – results are similar).

Result: Average case behavior of a BST is

$$\Theta(\log N)$$

You determined the constant $C$ for the estimate $\sim(CN)$ in lab! Note that $C$ was very small! This is an excellent result!
Balanced BSTs

The next question is always: Can we do better?

Specifically, can we find a way to eliminate the worst case trees, and get $O(\log N)$ for all operations?

This amounts to the following problem: Can we restructure the tree during inserts and deletes to prevent imbalanced trees?

The answer, of course, is YES, and one solution to creating balanced trees is called 2-3 Trees....
2-3 Trees generalize binary search trees by allowing “wider” nodes that can contain 1 or 2 keys, and 2 or 3 pointers:

```java
class Node {
    int K1, K2;
    Node left;
    Node mid;
    Node right;
}
```
Generalizing the basic idea of binary search trees, we have “trinary search trees” where the two keys divide up the descendant nodes into three instead of two subtrees:
But we may consider normal BST nodes (1 key, 2 pointers) to be a special case, where the second key does not exist:
2-3 Trees

But we may consider normal BST nodes (1 key, 2 pointers) to be a special case, where the second key does not exist, and we will draw these as we would with normal BSTs:
2-3 Trees

Searching such a tree is a simple generalization of search in BSTs: at each node you scan from the left through the two keys and figure out where the search key k might be:

```java
boolean member(int k, Node p) {
    if(p == null)
        return false;
    else if(k < p.K1)
        return find(k, p.left);
    else if(k == p.K1)
        return true;
    else if(p.K2 does not exist || k < p.K2)
        return find(k, p.mid);
    else if(k == K2)
        return true;
    else
        return find(k, p.right);
}
```
Insertion into a 2-3 tree is a little bit complicated, because we will want to maintain the trees in balanced form (perfect triangles):

A 2-3 tree is **balanced** if every path from the root to a leaf node has the same length; note that nodes may contain 2 keys and 3 pointers, or 1 key and 2 pointers:
Rules for inserting a new key into a 2-3 tree:

1. As with BSTs, you search for the key; if you find it, do nothing (don’t insert duplicates); if you don’t find it, then insert into the leaf node that you last looked in. If there is room, you are done.

Example: Let’s insert a 12 into an empty tree; when you insert into an empty tree, you create a new node and insert into the $K_1$ slot:

```
  12
    --
```
2-3 Trees

Rules for inserting a new key into a 2-3 tree:

1. As with BSTs, you search for the key; if you find it, do nothing (don’t insert duplicates); if you don’t find it, then insert into the leaf node that you last looked in. If there is room, you are done.

Example: Let’s insert a 12 into an empty tree; when you insert into an empty tree, you create a new node and insert into the $K_1$ slot:

Now let’s insert an 8, which can fit into the node if we move the 12 over:
2-3 Trees

Rules for inserting a new key into a 2-3 tree:

1. As with BSTs, you search for the key; if you find it, do nothing (don’t insert duplicates); if you don’t find it, then insert into the leaf node that you last looked in. If there is room, you are done.

2. But if there are already 2 keys, then insert into the node anyway, creating an “error node” containing 3 keys (too many!).

Example: Let’s insert a 12 into an empty tree; when you insert into an empty tree, you create a new node and insert into the $K_1$ slot:

Now let’s insert an 8, which can fit into the node if we move the 12 over:

Next let’s insert a 15, which expands the node into an error node containing too many keys:
Rules for inserting a new key into a 2-3 tree:

1. As with BSTs, you search for the key; if you find it, do nothing (don’t insert duplicates); if you don’t find it, then insert into the leaf node that you last looked in. If there is room, stop.

2. But if there are already 2 keys, then insert into the node anyway, creating an “error node” containing 3 keys (too many!). Then apply the \( \alpha \)-transformation to change this into a legal configuration of three nodes.

Next let’s insert a 15, which expands the node into an error node containing too many keys:

\[
\begin{array}{c}
8 \\
12 \\
15
\end{array}
\]

Immediately fix this error by transforming this node into a balanced three-node tree:

\[
\begin{array}{c}
8 \\
12 \\
15
\end{array}
\]

\[\alpha\text{-transformation}\]
2-3 Trees

α-transformation:

The subtrees A – D may be null!
2-3 Trees

Rules for inserting a new key into a 2-3 tree:

1. As with BSTs, you search for the key; if you find it, do nothing (don’t insert duplicates); if you don’t find it, then insert into the leaf node that you last looked in. If there is room, stop.

2. But if there are already 2 keys, then insert into the node anyway, creating an “error node” containing 3 keys (too many!). Then apply the $\alpha$-transformation to change this into a legal configuration of three nodes.

Immediately fix this error by transforming this node into a balanced three-node tree:

Next let’s insert a 20, which expands the right-most leaf node:
**2-3 Trees**

**Rules for inserting a new key into a 2-3 tree:**

1. As with BSTs, you search for the key; if you find it, do nothing (don’t insert duplicates); if you don’t find it, then insert into the leaf node that you last looked in. If there is room, stop.

2. But if there are already 2 keys, then insert into the node anyway, creating an “error node” containing 3 keys (too many!). Then apply the \( \alpha \)-transformation to change this into a legal configuration of three nodes.

Next let’s insert a 20, which expands the right-most leaf node:

```
12
8   15 20
```

Then let’s insert a 30, which creates another error node:

```
12
8   15 20 30
```
2-3 Trees

Rules for inserting a new key into a 2-3 tree:

1. As with BSTs, you search for the key; if you find it, do nothing (don’t insert duplicates); if you don’t find it, then insert into the leaf node that you last looked in. If there is room, stop.

2. But if there are already 2 keys, then insert into the node anyway, creating an “error node” containing 3 keys (too many!). Then apply the \( \alpha \)-transformation to change this into a legal configuration of three nodes.

Then let’s insert a 30, which creates another error node:

But we immediately fix the error by using the \( \alpha \)-transformation:
2-3 Trees

**Rules for inserting a new key into a 2-3 tree:**

1. As with BSTs, you search for the key; if you find it, do nothing (don’t insert duplicates); if you don’t find it, then insert into the leaf node that you last looked in. If there is room, stop.

2. But if there are already 2 keys, then insert into the node anyway, creating an “error node” containing 3 keys (too many!). Then apply the $\alpha$-transformation to change this into a legal configuration of three nodes.

3. After applying the $\alpha$-transformation, if there is a parent node, then we must apply the $\beta$-transformation to fix the imbalance created by the $\alpha$-transformation.

But we immediately fix the error by using the $\alpha$-transformation:

But this is imbalanced, so we will combine the root of the new subtree with the parent node:
2-3 Trees

Rules for inserting a new key into a 2-3 tree:

1. As with BSTs, you search for the key; if you find it, do nothing (don’t insert duplicates); if you don’t find it, then insert into the leaf node that you last looked in. If there is room, stop.

2. But if there are already 2 keys, then insert into the node anyway, creating an “error node” containing 3 keys (too many!). Then apply the $\alpha$-transformation to change this into a legal configuration of three nodes.

3. After applying the $\alpha$-transformation, if there is a parent node, then we must apply the $\beta$-transformation to fix the imbalance created by the $\alpha$-transformation.
2-3 Trees

\( \beta\)-transformation(s): If the parent has only 1 key, then insert the root into the parent node and distribute the subtrees accordingly:
2-3 Trees

**β-transformation(s):** If the parent has only 1 key, then insert the root into the parent node and distribute the subtrees accordingly:
2-3 Trees

**β-transformation(s):** If the parent has 2 keys, then create an error node and repeat the α-transformation (you may have to continue apply α- and β-transformations up the tree):
2-3 Trees

**β-transformation(s):** If the parent has 2 keys, then create an error node and go back to the α-transformation (you may have to continue apply α- and β-transformations up the tree):
2-3 Trees

β-transformation(s): If the parent has 2 keys, then create an error node and go back to the α-transformation (you may have to continue apply α- and β-transformations up the tree):
2-3 Trees

Rules for inserting a new key into a 2-3 tree:

1. As with BSTs, you search for the key; if you find it, do nothing (don’t insert duplicates); if you don’t find it, then insert into the leaf node that you last looked in. If there is room, stop.

2. But if there are already 2 keys, then insert into the node anyway, creating an “error node” containing 3 keys (too many!). Then apply the α-transformation to change this into a legal configuration of three nodes.

3. After applying the α-transformation, if there is a parent node, then we must apply the β-transformation to fix the imbalance created by the α-transformation.

4. You may have to continue a series of α- and β-transformations moving up the path to the root, until a balanced tree with no error nodes is obtained.

Let’s continue with our example....

Insert a 16 into the tree:
2-3 Trees

Insert 16:

Insert 18:

α

β
Summary of rules for inserting a new key into a 2-3 tree:

1. Insert new key into appropriate leaf node, potentially creating an error node;
2. If there is an error node, apply $\alpha$- and $\beta$-transformations moving up the path to the root, until a balanced tree with no error nodes is obtained.
2-3 Trees

Worst-Case Time Complexity of 2-3 Trees (counting the number of comparisons): Member(....)

Consider the following tree:
- What is the cost (# of comparisons) for finding 2?
- How about 27?
- Which keys represent the worst case for this tree?
2-3 Trees

Worst-Case Time Complexity of 2-3 Trees (counting the number of comparisons): Member(….)

Consider the following tree:

- What is the cost (# of comparisons) for finding 2? 3
- How about 27? 5
- Which keys represent the worst case for this tree? 46 or 66, with 6 comparisons
Worst-Case Time Complexity of 2-3 Trees (counting the number of comparisons): Member(…)

The worst-case for member(…) is to go all the way to a leaf node, and do 2 comparisons at each node; in a balanced tree with $N$ keys, the height is $\Theta(\log N)$, i.e., $C \log N + \ldots$ for some constant $C$, but if we have to do 2 comparisons at each node, this becomes $2 \cdot C \cdot \log N + \ldots$ which is still $\Theta(\log N)$ comparisons.
2-3 Trees

Worst-Case Time Complexity of 2-3 Trees (counting the number of comparisons): Insert(….)

For insert(...), the worst thing that can happen is that you insert the new key at the bottom of the tree, and it causes α- and β-transformations all the way back up the tree. Each transformation takes a constant $C$ amount of work, so the cost is $\Theta(\log N)$ to find the location (as in member(...)), and $C \cdot \Theta(\log N)$ transform the tree back up to the root. $(1 + C) \cdot \Theta(\log N)$ is still $\Theta(\log N)$. 
Worst-Case Time Complexity of 2-3 Trees (counting the number of comparisons):

Member(….): $\Theta(\log N)$

Insert(….): $\Theta(\log N)$

Delete(….): $\Theta(\log N)$ (not described)
**Code Complexity:** 2-3 Trees are generally encoded as normal BSTs with two different colored links ("Red-Black Trees"), and the code for insert is not as complicated as you would imagine:

```java
private static Node insert(int key, Node t) {
    if (t == null)
        return new Node(key);
    else if (key < t.key)
        t.left = insert(key, t.left);
    return applyTransformations(t);
} else if (key > t.key)
    t.right = insert(key, t.right);
return applyTransformations(t);
} else
    return t;
}

private static Node leanRight( Node t ) {
    Node newRoot = t.left;
    t.left = newRoot.right;
    newRoot.right = t;
    newRoot.red = t.red;
    t.red = true;
    return newRoot;
}

private static Node rotateLeft( Node t ) {
    Node newRoot = t.right;
    t.right = t.right.left;
    newRoot.left = t;
    newRoot.red = true;
    newRoot.left.red = false;
    newRoot.right.red = false;
    return newRoot;
}

private static Node applyTransformations( Node t ) {
    if(t == null)
        return null;
    if(t.left != null && t.left.red)
        t = leanRight( t );
    if( t.right != null && t.right.red
        && t.right.right != null && t.right.right.red)
        t = rotateLeft( t );
    return t;
}
```