CPSC 223
Algorithms & Data Abstract Structures

Lecture 22:
B Trees: 2-3 & 2-3-4 trees

Today …

• 2-3 Trees [Ch 12: 651-670]
• 2-3-4 Trees (insertion)
2-3 Trees

B Trees (… starting with 2-3 trees)

• A “2-3 Tree” is a (search) tree in which:
  – Every internal (non-leaf) node has either 2 or 3 children
  – All leaves are at the same level

• A node with 2 children is called a “2-node”
• A node with 3 children is called a “3-node”
Note on Notation

The book draws a 2-node as a regular *binary tree* node:

5

And a 3-node as a “wide” rounded node:

5 10

These are equivalent to the nodes:

<table>
<thead>
<tr>
<th>5</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

The slots denote left, middle, and right pointers

2-3 Tree Examples

2-3 tree with a single item “5” (a leaf node)

| 5 |   |   |

2-3 tree with items “5” and “10” (also a leaf node)

| 5 | 10 |

2-3 tree with 5 items (4 nodes)

| 5 | 10 |

2-3 tree with the same 5 items (but with 3 nodes)

| 6 |

| 4 | 5 | 10 | 11 |
Placing Data in a 2-3 Tree

1. A 2-node (2 children) must contain a single data item. The item's search key $S$ is:
   - greater than its left child's search keys
   - smaller than (or equal to) its middle child's search keys

2. A 3-node (3 children) must contain two data items. For $S$ the left and $L$ the right search key:
   - $S$ is greater than the left child's search keys
   - $S$ is smaller than (or equal to) the middle child's search keys
   - $L$ is greater than the middle child's search keys
   - $L$ is smaller than (or equal to) the right child's search keys

3. A leaf node may contain one or two data items.

Placing Data in a 2-3 Tree

- Is this a well-formed 2-3 tree?
  - NO!
Traversing a 2-3 Tree

• Traversal very similar to a BST!

```cpp
void inorder(const Node* root) const {
    if(root != NULL) {
        inorder(root->leftChild);
        visit(root->item1); // visit 1st data item
        inorder(root->middleChild);
        if(has2Keys(root)) {
            visit(root->item2); // visit 2nd data item
        }
        inorder(root->rightChild);
    }
}
```

Traversing a 2-3 Tree (inorder)
Traversing a 2-3 Tree

Start here (root)
Traverse left

New root
Traverse left

Start here (root)
Traverse left

New root
Traverse left
Traversing a 2-3 Tree

Start here (root)

New root

Traverse left

New root

Traverse left

New root
Traversing a 2-3 Tree

Output: 10

Start here (root)

New root

Traverse left

30

New root

50

Visit first item (after traversing to NULL node)

10 20

Traverse left

30

Traverse left

30

Visit second item (after traversing to NULL node)

10 20

50

Output: 10 20

65

80

40

60

70

90
Traversing a 2-3 Tree

Output: 10 20 30

Start here (root)
New root
Visit first item (finished traversing left)

Traversing a 2-3 Tree

Output: 10 20 30

Start here (root)
New root
Traverse middle
Traversing a 2-3 Tree

Output: 10 20 30

Start here (root)

Traverse left

New root

Traverse middle

Visit first item (after traversing to NULL node)

Output: 10 20 30 40

Start here (root)

Traverse left

New root

Traverse middle

New root

Visit first item (after traversing to NULL node)
Traversing a 2-3 Tree

Output: 10 20 30 40

Start here (root)

Traverse left

New root

Finished visiting nodes

---

Traversing a 2-3 Tree

Output: 10 20 30 40 50

Start here (root)

Visit first item (after traversing to NULL node)
Traversing a 2-3 Tree

Output: 10 20 30 40 50

Start here (root)

Traverse middle

New root
Traversing a 2-3 Tree

Output: 10 20 30 40 50

Start here (root)

Traverse middle

New root

Traverse left

New root

New root
Traversing a 2-3 Tree

Output: 10 20 30 40 50 60

Start here (root)

Traverse middle

New root

Traverse left

Visit first item (after traversing to NULL node)
Traversing a 2-3 Tree

Output: 10 20 30 40 50 60 65

Start here (root)

Traverse middle

New root

... and so on

Retrieving from a 2-3 Tree

• Retrieval very similar to a BST!

```cpp
void retrieve(const Node* root, const Key& key, Entry& entry) const {
  if(root != NULL) {
    if(key == root->item1)
      entry = root->item1;
    else if(has2Keys(root) && key == root->item2)
      entry = root->item2;
    else if(key < root->item1)
      retrieve(root->leftChild, key, entry);
    else if(has1Key(root) || (has2Keys(root) && key < root->item2))
      retrieve(root->middleChild, key, entry);
    else
      retrieve(root->rightChild, key, entry);
  }
}
```
Searching a 2-3 Tree

Start here (root)

Lookup "70"

Since 2-node and 50 < 70
Search middle child
Searching a 2-3 Tree

Lookup "70"

Since 3-node and 65 < 70 < 80
Search middle child
Searching a 2-3 Tree

Lookup "70"

Since 70 == 70
We found a match!
2-3 Trees

Some Observations

- A 2-3 tree is (obviously) not a binary tree
  - But a 2-3 tree is very similar to a full binary tree
  - All leaves are on the same level in a 2-3 tree

- So a 2-3 tree is always (perfectly) balanced
  - … and has height at most $\lceil \log_2(n+1) \rceil$
  - which is the case for a full BST
  - Best case height is $\lceil \log_3(n+1) \rceil$ (each node has 3 children)

2-3 Trees

Advantages/Disadvantages

- 2-3 trees are (fully) balanced … $O(\log n)$ height
  - Insertion and deletion not (terribly) difficult to implement

- Shorter height (because of “wider” nodes) does not necessarily mean more efficient retrieval time
  - Requires additional comparisons (up to 2 per node)
  - Height helps in some cases (e.g., File Indexes/Databases)
Inserting Nodes

So how can we maintain a 2-3 tree on insertion?

– Locate the leaf node where the item “should” be
– If the leaf contains only one item, then we are done!
– Otherwise, we have to “split” the node
– Splitting a node may cause the parent to split
– And so on up the tree

Insertion into a 2-3 Tree

Insert “75”

1. Search for item
Insertion into a 2-3 Tree

1. Search for item
2. Try to insert item into the last search node

The leaf node where the search stopped

Insert "75"

In this case, we're OK since the leaf only had 1 item

1. Search for item
2. Try to insert item into the last search node
Insertion into a 2-3 Tree

1. Search for item
2. Try to insert item into the last search node
3. If the leaf node has 2 items, “split” the node

Splitting Nodes in a 2-3 Tree

If after the insertion, we would have three items $S < M < L$
- Place smallest item $S$ into $n_1$
- Place largest item $L$ into $n_2$
- Move middle item $M$ into the leaf’s parent
Splitting Nodes in a 2-3 Tree – Symmetric Case

If after the insertion, we would have three items \( S < M < L \)

- Place smallest item \( S \) into \( n_1 \)
- Place largest item \( L \) into \( n_2 \)
- Move middle item \( M \) into the leaf’s parent

Splitting Nodes in a 2-3 Tree (General Case)

What happens if the parent node is already full?

- In this case, we have to maintain the child pointers …
Insertion into a 2-3 Tree

1. Search for item
2. Try to insert item into the last search node
3. If the leaf node has 2 items, “split” the node

Now Insert “72”

We have to split the leaf node!

1. Search for item
2. Try to insert item into the last search node
3. If the leaf node has 2 items, “split” the node

Are we done?
Insertion into a 2-3 Tree

1. Search for item
2. Try to insert item into the last search node
3. If the leaf node has 2 items, “split” the node
4. And recursively split nodes as needed up the tree

Now Insert “72”
We have to split the parent!

Now we are done!

1. Search for item
2. Try to insert item into the last search node
3. If the leaf node has 2 items, “split” the node
4. And recursively split nodes as needed up the tree
Removing Nodes

What about for removing nodes?
– The inverse of insertion …
– Locate the node that contains the item to remove
– If the node is not a leaf, then swap with inorder successor
– If the leaf contains two items, then we are done!
– Otherwise, we either “redistribute” or “merge” nodes
– Again, this may repeat up the tree

We want to delete “X”
• If one of X’s adjacent siblings has 2 items, then redistribute (and we are done!)
• Otherwise, merge with an adjacent sibling (cascade)
Removing Nodes (General Case)

Both of these cases will only occur after a merge.

Does redistribute look familiar?

What about merge?

2-3-4 Trees
2-3-4 Trees

- A 2-3-4 Tree is a 2-3 Tree that also allows 4-nodes
  - That is, nodes with 3 items and 4 children

  More efficient insertion
  - Split 4 Nodes \textit{as you search} for the leaf
  - As opposed to bottom up in a 2-3 tree
  - Guarantees we insert into a leaf node with an empty slot
  - Works because a 4-node has 3 items (consider root case)

\begin{figure}
\centering
\includegraphics[width=\textwidth]{image}
\caption{Insert 25}
\end{figure}