>>

2-3-4 Trees

∧ >>

❖ Search Cost

Critical factor determining search cost in BSTs

- worst case: length of longest path
- average case: < average path length (not all searches end at leaves)

Either way, path length (tree depth) is a critical factor

In a perfectly balanced tree, max path length = $log_2 n$

The 2 in the path length is the branching factor (*binary* search tree)

What if branching factor > 2?

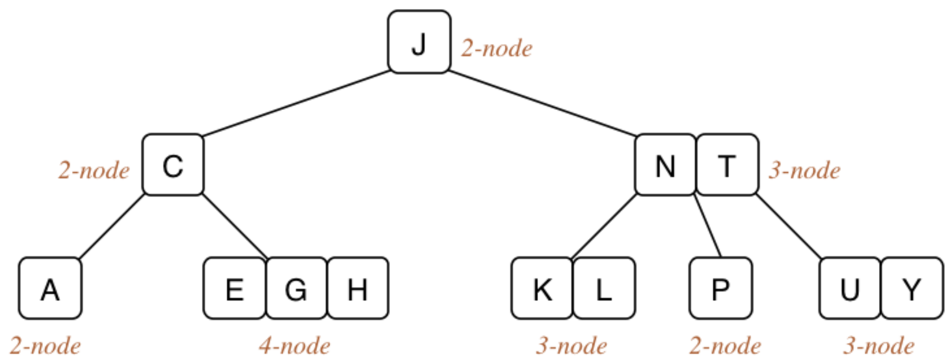
- $\log_2 4096 = 12$, $\log_4 4096 = 6$, $\log_8 4096 = 4$

<< ^ >>

❖ 2-3-4 Trees

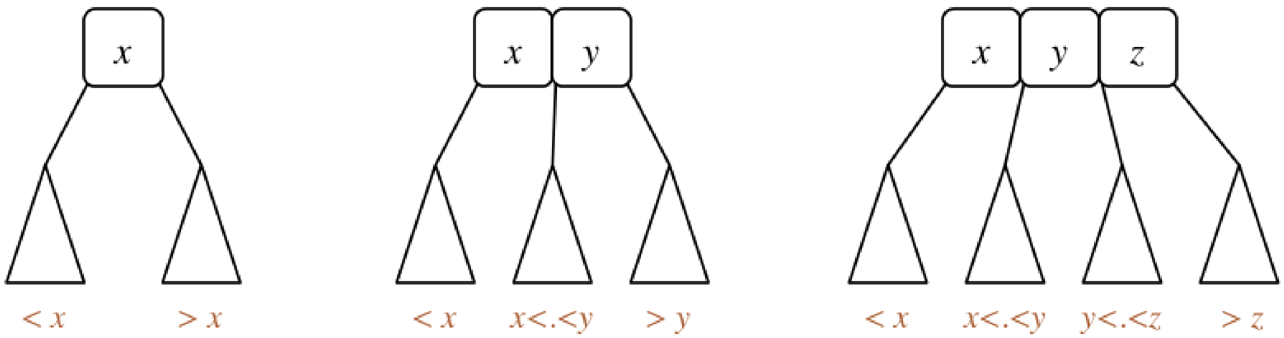
2-3-4 trees have three kinds of nodes

- 2-nodes, with two children (same as normal BSTs)
- 3-nodes, two values and three children
- 4-nodes, three values and four children



❖ ... 2-3-4 Trees

2-3-4 trees are ordered similarly to BSTs



In a balanced 2-3-4 tree:

- all leaves are at same distance from the root

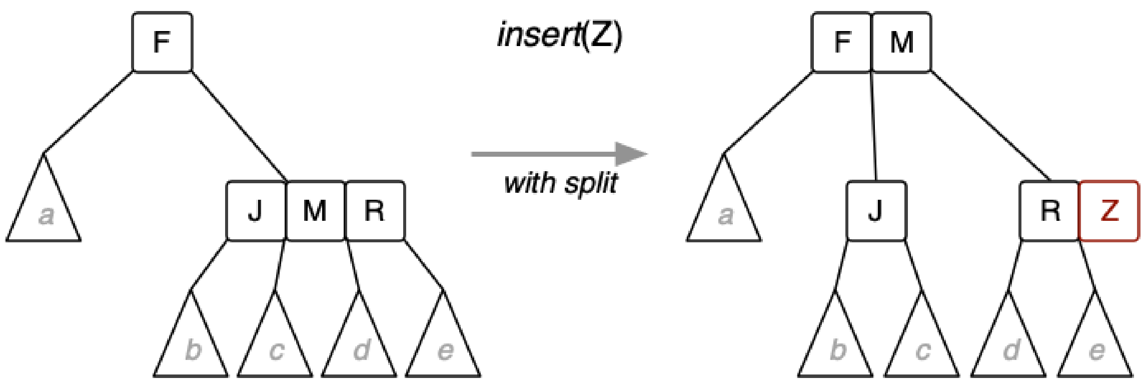
2-3-4 trees grow "upwards" from the leaves, via node splitting.

<< ∧ >>

❖ Node splitting

Insertion into a full node causes a split

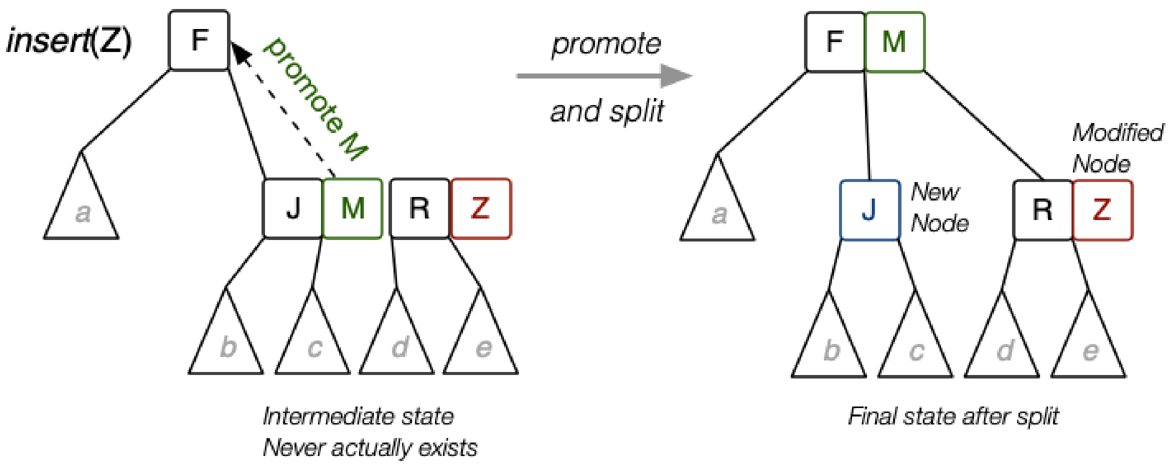
- middle value propagated to parent node
- values in original node split across original node and new node



<< ∧ >>

❖ ... Node splitting

Intermediate stage of insert-split:



<< ^ >>

❖ ... Node splitting

Searching in 2-3-4 trees:

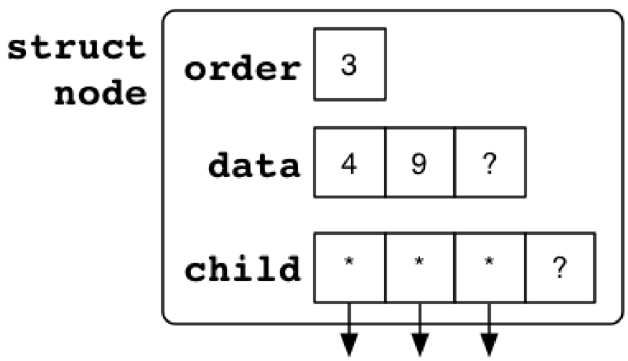
```
Search(tree,item):
|  Input  tree, item
|  Output address of item if found in 2-3-4 tree
|         NULL otherwise
|
|  if tree is empty then
|     return NULL
|  else
|  |  scan tree.data to find i such that
|  |     tree.data[i-1] < item ≤ tree.data[i]
|  |  if item=tree.data[i] then   // item found
|  |     return address of tree.data[i]
|  |  else        // keep looking in relevant subtree
|  |     return Search(tree.child[i],item)
|  |  end if
|  end if
```

<< ^ >>

❖ Data Structure

Possible concrete 2-3-4 tree data structure:

```
typedef struct node {
   int         order;        // 2, 3 or 4
   int         data[3];      // items in node
   struct node *child[4];    // links to subtrees
} node;
```

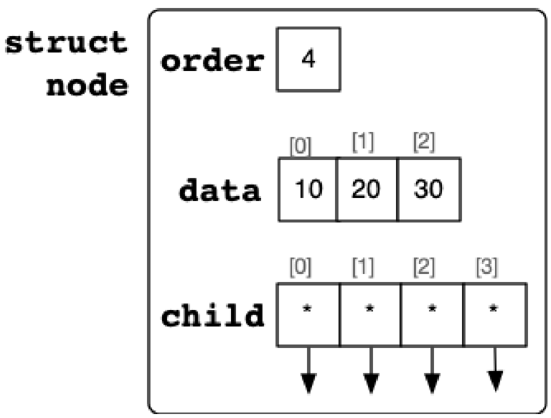


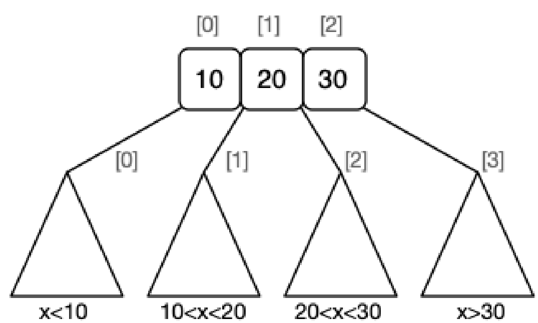
<< ^ >>

❖ ... Data Structure

Finding which branch to follow

```
// n is a pointer to a (struct node)
int i;
for (i = 0; i < n->order-1; i++) {
   if (item <= n->data[i]) break;
}
// go to the ith subtree, unless item == n->data[i]
```





<< ∧ >>

❖ Search Cost Analysis

2-3-4 tree searching cost analysis:

- as for other trees, worst case determined by height h
- 2-3-4 trees are always balanced ⇒ height is $O(\log n)$
- worst case for height: all nodes are 2-nodes
  (same case as for balanced BSTs, i.e. $h \cong \log_2 n$)
- best case for height: all nodes are 4-nodes
  (balanced tree with branching factor 4, i.e. $h \cong \log_4 n$)

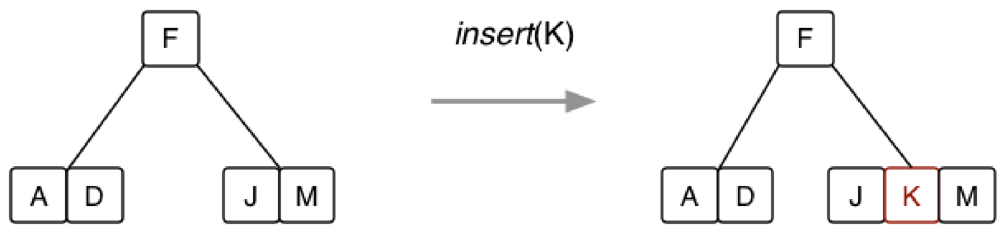
❖ Insertion into 2-3-4 Trees

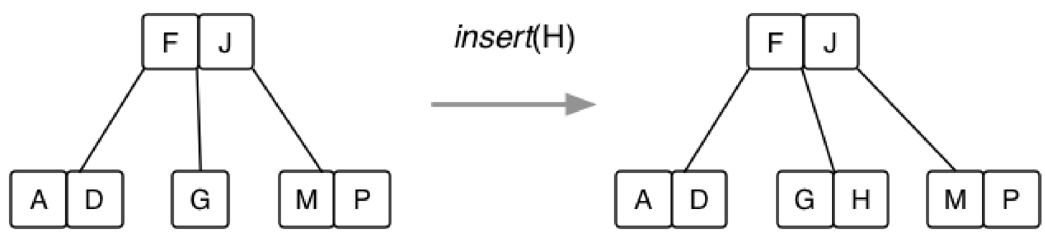
Insertion algorithm:

- find leaf node where Item belongs (via search)
- if not full (i.e. order < 4)
  - insert Item in this node, order++
- if node is full (i.e. contains 3 items)
  - split into two 2-nodes as leaves
  - promote middle element to parent
  - insert item into appropriate leaf 2-node
  - if parent is a 4-node
    - continue split/promote upwards
  - if promote to root, and root is a 4-node
    - split root node and add new root

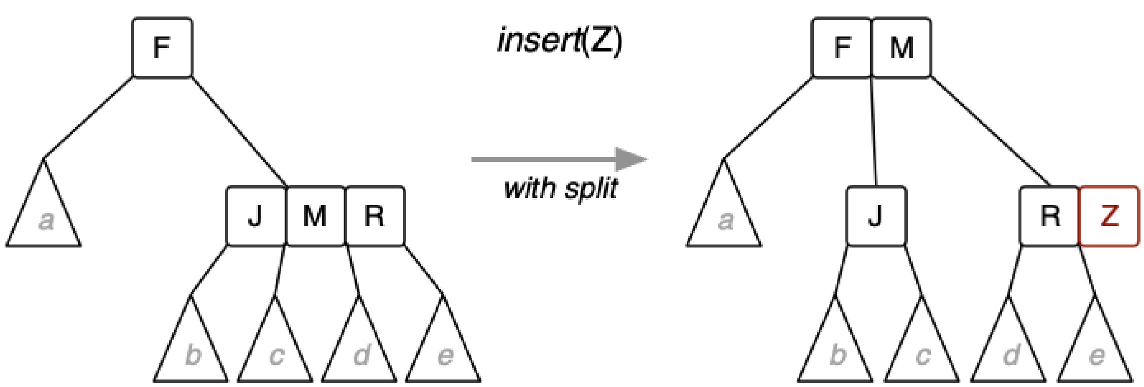
❖ ... Insertion into 2-3-4 Trees

Insertion into a 2-node or 3-node:



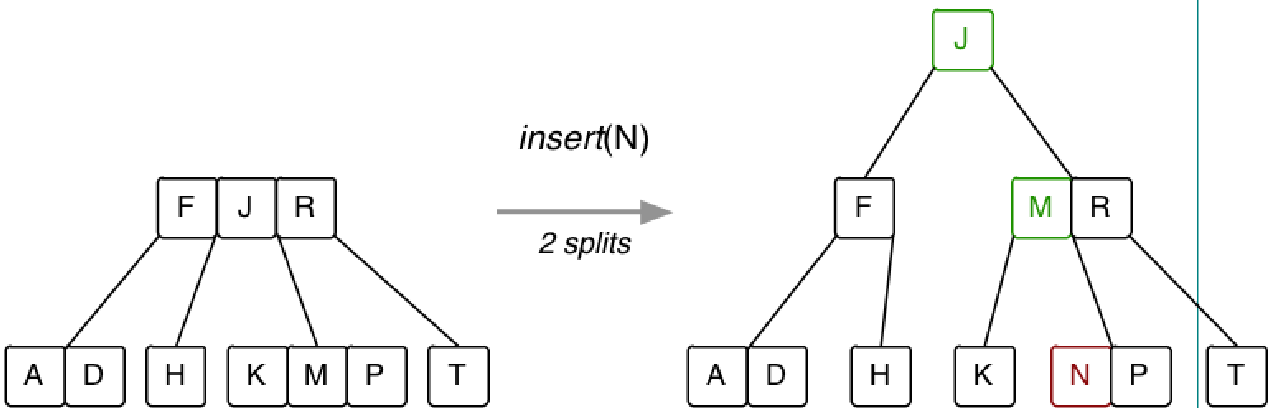


Insertion into a 4-node (requires a split):



❖ ... Insertion into 2-3-4 Trees

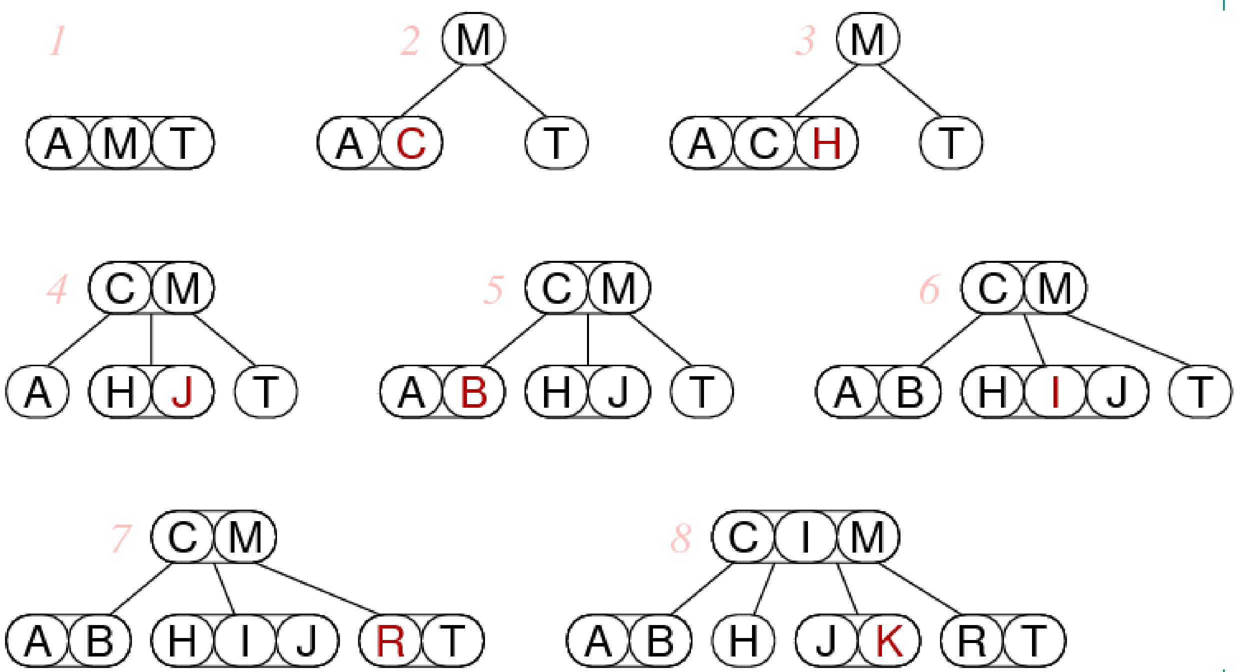
Splitting the root:



<< ∧ >>

❖ ... Insertion into 2-3-4 Trees

Building a 2-3-4 tree ... 7 insertions:



❖ ... Insertion into 2-3-4 Trees

Insertion algorithm:

```
insert(tree,item):
|  Input  2-3-4 tree, item
|  Output tree with item inserted
|
|  if tree is empty then
|     return new node containing item
|  end if
|  node=Search(tree,item)
|  parent=parent of node
|  if node.order < 4 then
|     insert item into node
|     increment node.order
|  else
|  |  promote = node.data[1]     // middle value
|  |  nodeL   = new node containing data[0]
|  |  nodeR   = new node containing data[2]
|  |  delete node
|  |  if item < promote then
|  |     insert(nodeL,item)
|  |  else
|  |     insert(nodeR,item)
|  |  end if
|  |  insert(parent,promote)
|  |  while parent.order=4 do
|  |     continue promote/split upwards
|  |  end while
|  |  if parent is root ∧ parent.order=4 then
|  |     split root, making new root
|  |  end if
|  end if
```

❖ ... Insertion into 2-3-4 Trees

Insertion cost (remembering that 2-3-4 trees are balanced ⇒ $h = \log_4 n$)

- search for leaf node in which to insert = *O(log n)*
- if node not full, insert item into node = *O(1)*
- if node full, promote middle, create two new nodes = *O(1)*
- if promotion propagates ...
  - best case: update parent = *O(1)*
  - worst case: propagate to root = *O(log n)*

Overall insertion cost = *O(log n)*

❖ 2-3-4 Variations

Variations on 2-3-4 trees ...

Variation #1: why stop at 4? why not 2-3-4-5 trees? or *M*-way trees?

- allow nodes to hold between *M/2* and *M-1* items
- if each node is a disk-page, then we have a B-tree (databases)
- for B-trees, depending on **`Item`** size, *M > 100/200/400*

Variation #2: don't have "variable-sized" nodes

- use standard BST nodes, augmented with one extra piece of data
- implement similar strategy as 2-3-4 trees → red-black trees.

Produced: 17 Jun 2020