COMP2521 19T0
Week 7, Thursday: Tropical Paradise

Jashank Jeremy

jashank.jeremy@unsw.edu.au

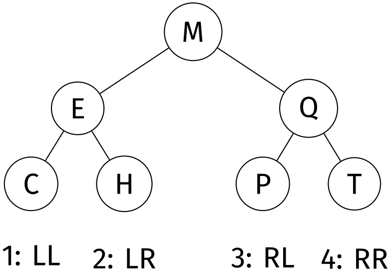
exotic trees

Balanced Trees

Splay Trees

Double-Rotation Cases

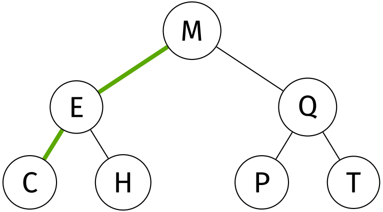
Four choices to consider for a double-rotation:



Splay Rotations

Double-Rotation: Left, Left

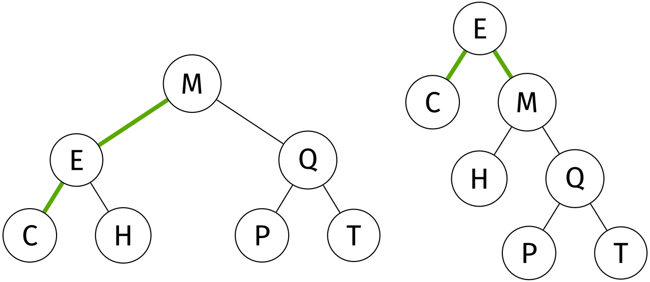
$\textsc{rotateR}\ \tau_{\mathsf{M}}$ $\textsc{rotateR}\ \tau_{\mathsf{E}}$



Splay Rotations

Double-Rotation: Left, Left

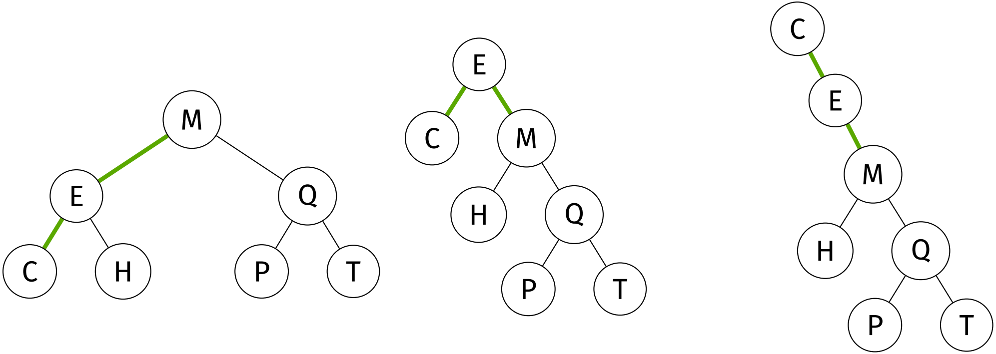
$\textsc{rotateR}\ \tau_{\mathsf{M}}$ $\quad$ $\textsc{rotateR}\ \tau_{\mathsf{E}}$



Splay Rotations

Double-Rotation: Left, Left

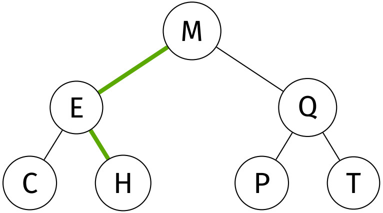
$\textsc{rotateR}\ \tau_{\mathsf{M}}$ $\textsc{rotateR}\ \tau_{\mathsf{E}}$



Splay Rotations

Double-Rotation: Left, Right

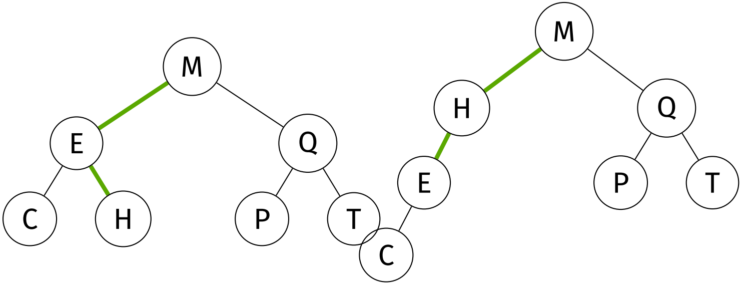
$\textsc{rotateL}\ \tau_{\mathsf{E}} \qquad \textsc{rotateR}\ \tau_{\mathsf{M}}$



Splay Rotations

Double-Rotation: Left, Right

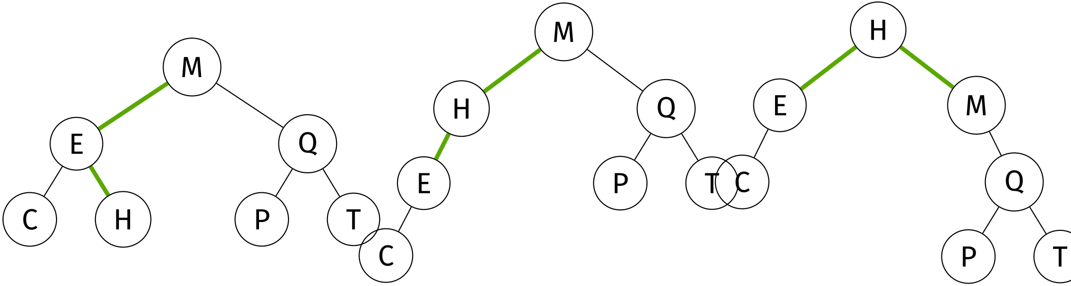
$\textsc{rotateL}\ \tau_{\mathsf{E}} \qquad \textsc{rotateR}\ \tau_{\mathsf{M}}$



Splay Rotations

Double-Rotation: Left, Right

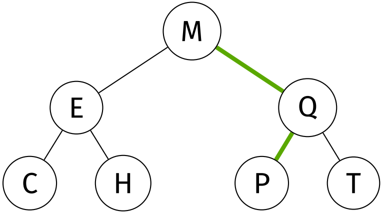
$\textsc{rotateL}\ \tau_E \qquad \textsc{rotateR}\ \tau_M$



Splay Rotations

Double-Rotation: Right, Left

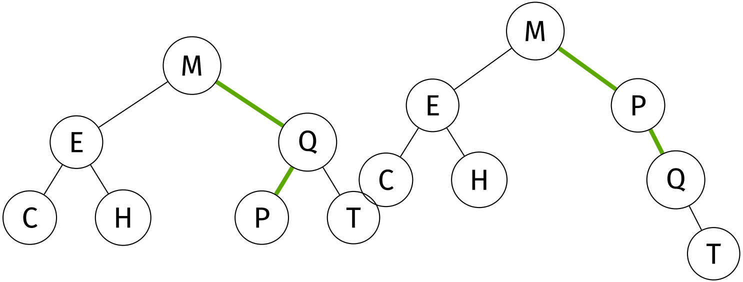
$\textsc{rotateR}\ \tau_{\mathsf{Q}} \qquad \textsc{rotateL}\ \tau_{\mathsf{M}}$



Splay Rotations

Double-Rotation: Right, Left

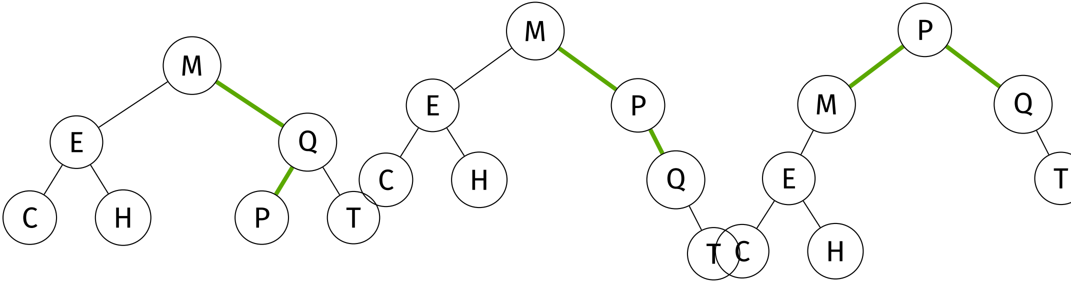
$\textsc{rotateR}\ \tau_{\mathsf{Q}} \qquad \textsc{rotateL}\ \tau_{\mathsf{M}}$



Splay Rotations

Double-Rotation: Right, Left

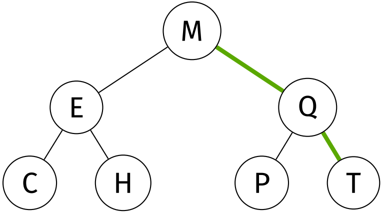
$\textsc{rotateR}\ \tau_{\mathsf{Q}} \qquad \textsc{rotateL}\ \tau_{\mathsf{M}}$



Splay Rotations

Double-Rotation: Right, Right

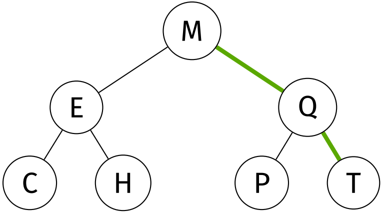
$\textsc{rotate}\mathrm{L}\ \tau_{\mathsf{M}} \qquad \textsc{rotate}\mathrm{L}\ \tau_{\mathsf{Q}}$

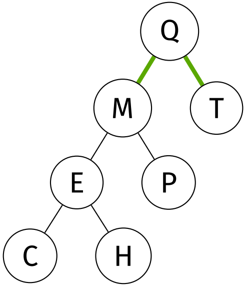


Splay Rotations

Double-Rotation: Right, Right

$\textsc{rotateL}\ \tau_{\mathsf{M}}$ $\qquad$ $\textsc{rotateL}\ \tau_{\mathsf{Q}}$

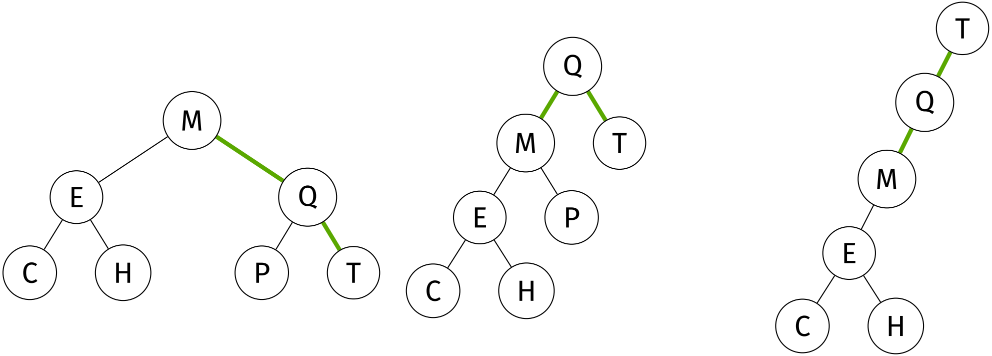




Splay Rotations

Double-Rotation: Right, Right

$\textsc{rotateL}\ \tau_{\mathsf{M}}$ $\textsc{rotateL}\ \tau_{\mathsf{Q}}$



Splay Search

Some implementations do rotation-on-search,
which has a similar effect to periodic rotations;
increases search cost by doing more work,
decreases search cost by moving likely keys closer to root.

Even on a degenerate tree,
splay search massively improves the balance of the tree.

Splay Trees

C Implementation (I)

```
btree_node *btree_insert_splay (btree_node *tree, Item it)
{
    if (tree == NULL) return btree_node_new (it, NULL, NULL);
    int diff = item_cmp (it, tree->value);
```

Splay Trees

C Implementation (I)

```
btree_node *btree_insert_splay (btree_node *tree, Item it)
{
    if (tree == NULL) return btree_node_new (it, NULL, NULL);
    int diff = item_cmp (it, tree->value);
    if (diff <  0) {
        if (tree->left == NULL) {
            tree->left = btree_node_new (it, NULL, NULL);
            return tree;
        }
```

Splay Trees

C Implementation (I)

```
btree_node *btree_insert_splay (btree_node *tree, Item it)
{
    if (tree == NULL) return btree_node_new (it, NULL, NULL);
    int diff = item_cmp (it, tree->value);
    if (diff <  0) {
        if (tree->left == NULL) {
            tree->left = btree_node_new (it, NULL, NULL);
            return tree;
        }
        int ldiff = item_cmp (it, tree->left->value);
        if (ldiff < 0) {
            // Case 1: left-left
            tree->left->left = btree_insert_splay (tree->left->left, it);
            tree = btree_rotate_right (tree);
        } else {
            // Case 2: left-right
            tree->left->right = btree_insert_splay (tree->left->right, it);
            tree->left = btree_rotate_left (tree->left);
        }
        return btree_rotate_right (tree);
```

Splay Trees

C Implementation (II)

```
// ... btree_insert_splay continues ...
    } else if (diff  > 0) {
        int rdiff = item_cmp (it, tree->right->value);
        if (rdiff < 0) {
            // Case 3: right-left
            tree->right->left = btree_insert_splay (tree->right->left, it);
            tree->right = btree_rotate_right (tree->right);
        } else {
            // Case 4: right-right
            tree->right->right = btree_insert_splay (tree->right->right, it);
            tree = btree_rotate_left (tree);
        }
        return btree_rotate_left (tree);
    } else
        tree->value = it;
    return tree;

}
```

without insertion-specific code, we might call this `btree_splay`

Splay Search

C Implementation

```
btree_node *btree_search_splay (btree_node **root, Item it)
{
    assert (root != NULL);
    if (*root == NULL) return NULL;
    *root = btree_splay (*root, it);
    if (item_cmp ((*root)->value, it) == 0)
        return *root;
    else
        return NULL;

}
```

Splay Trees: Why Bother?

Insertion Time Complexity

worst case (for work):
item inserted at the end of a degenerate tree.
$O(n)$ steps necessary here...
but overall tree height now halved

worst case (from resulting tree):
item inserted at root of degenerate tree.
$O(1)$ steps necessary. surprise!

even in the worst case,
not possible to *repeatedly* have
$O(n)$ steps to insert

Splay Trees: Why Bother?

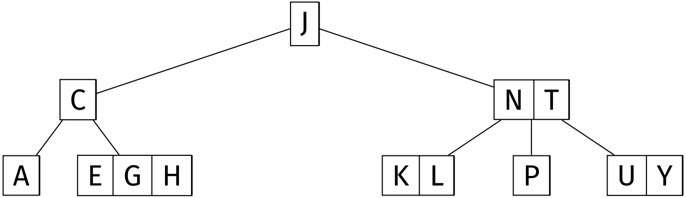
Overall Time Complexity

Assuming we do splay operations on insert and search, assuming we have N nodes and M inserts/searches: average $O\left(\left(N + M\right)\log_2\left(N + M\right)\right)$

A good (amortised) cost overall ...
but no guarantees of improved individual operations:
some may still be $O\left(N\right)$.

2–3–4 Trees

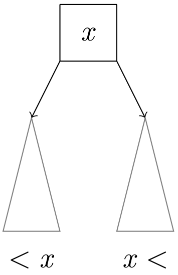
2–3–4 trees have three types of nodes:
2–nodes have one value and two children;
3–nodes have two values and three children;
4–nodes have three values and four children;



2–3–4 trees grow 'upwards' from the leaves,
all of which are equidistant to the root.

2–3–4 Trees
Node Value Ordering

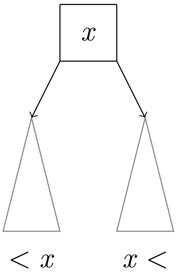
A similar ordering to a conventional BST:

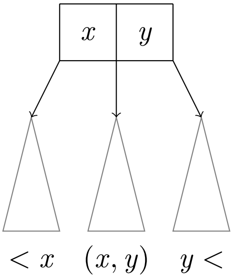


2–3–4 Trees

Node Value Ordering

A similar ordering to a conventional BST:

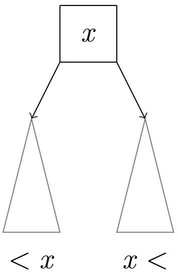


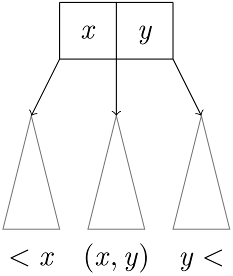


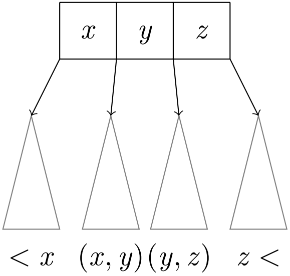
2–3–4 Trees

Node Value Ordering

A similar ordering to a conventional BST:







2–3–4 Tree Search

2–3–4 trees are always balanced; depth is $O(\log n)$

worst case for depth: all nodes are 2-nodes
same case as for balanced BSTs, i.e. $d \simeq \log_2 n$

best case for depth: all nodes are 4-nodes
balanced tree with branching factor 4, i.e. $d \simeq \log_4 n$

2–3–4 Tree Insertion

The Algorithm

1. find leaf node where item belongs (via `search`)

2–3–4 Tree Insertion

The Algorithm

1. find leaf node where item belongs (via `search`)
2. if node is not full (i.e., order < 4),
   insert item in this node, order++.

2–3–4 Tree Insertion

The Algorithm

1. find leaf node where item belongs (via `search`)
2. if node is not full (i.e., order < 4),
   insert item in this node, order++.
3. if node is full (i.e., contains 3 Items):

2–3–4 Tree Insertion

The Algorithm

1. find leaf node where item belongs (via `search`)
2. if node is not full (i.e., order < 4),
   insert item in this node, order++.
3. if node is full (i.e., contains 3 Items):
   1. split into two 2-nodes as leaves

2–3–4 Tree Insertion

The Algorithm

1. find leaf node where item belongs (via `search`)
2. if node is not full (i.e., order < 4),
   insert item in this node, order++.
3. if node is full (i.e., contains 3 Items):
   1. split into two 2-nodes as leaves
   2. promote middle element to parent

2–3–4 Tree Insertion

The Algorithm

1. find leaf node where item belongs (via `search`)
2. if node is not full (i.e., order < 4),
   insert item in this node, order++.
3. if node is full (i.e., contains 3 Items):
   1. split into two 2-nodes as leaves
   2. promote middle element to parent
   3. insert item into appropriate leaf 2-node

2–3–4 Tree Insertion

The Algorithm

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

2–3–4 Tree Insertion

The Algorithm

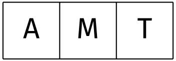
1. find leaf node where item belongs (via `search`)
2. if node is not full (i.e., order < 4),
   insert item in this node, order++.
3. if node is full (i.e., contains 3 Items):
   1. split into two 2-nodes as leaves
   2. promote middle element to parent
   3. insert item into appropriate leaf 2-node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

2–3–4 Tree Insertion (I)

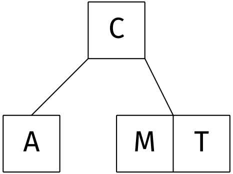
A	M	T

INSERT C

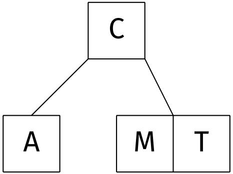
2–3–4 Tree Insertion
(I)



INSERT C

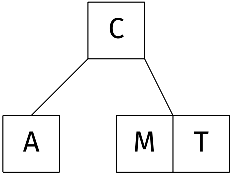


2–3–4 Tree Insertion (II)

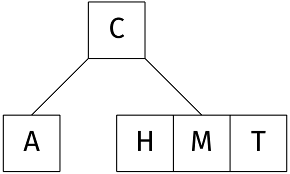


INSERT H

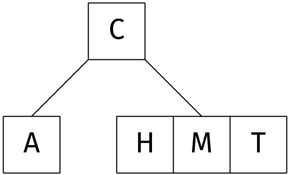
2–3–4 Tree Insertion (II)



INSERT H

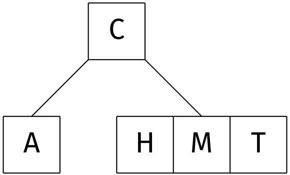


2–3–4 Tree Insertion (III)

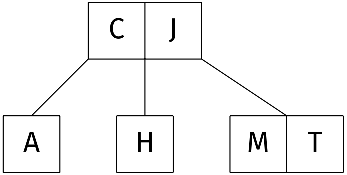


INSERT J

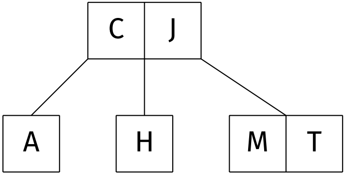
2–3–4 Tree Insertion (III)



INSERT J

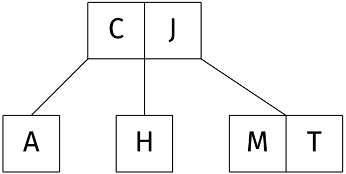


2–3–4 Tree Insertion (IV)

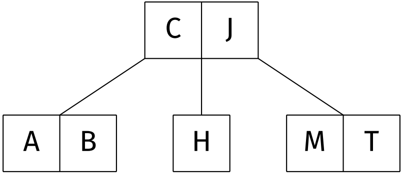


INSERT B

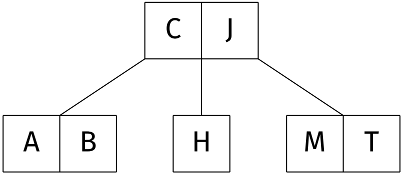
2–3–4 Tree Insertion (IV)



INSERT B

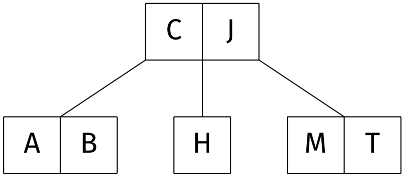


2–3–4 Tree Insertion (V)

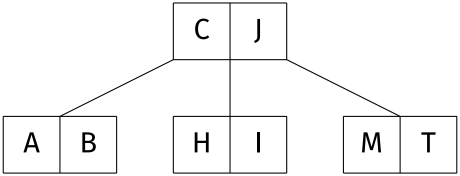


INSERT I

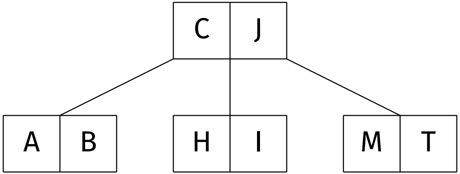
2–3–4 Tree Insertion (V)



INSERT I

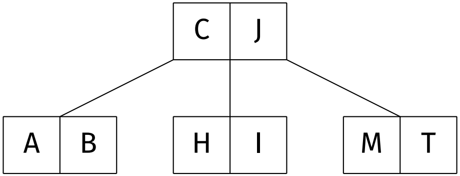


2–3–4 Tree Insertion (VI)

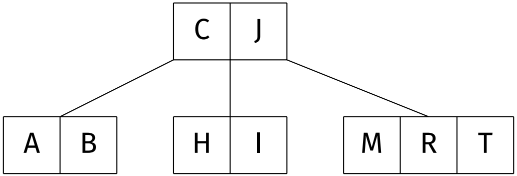


INSERT R

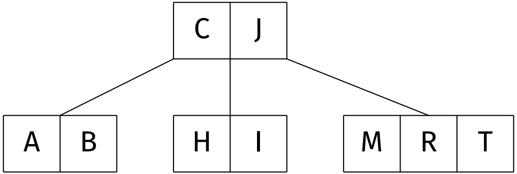
2–3–4 Tree Insertion (VI)



INSERT R

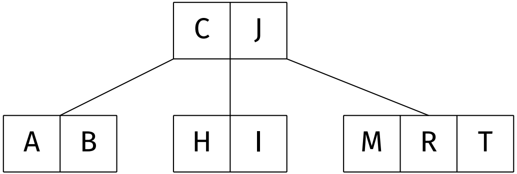


2–3–4 Tree Insertion (VII)

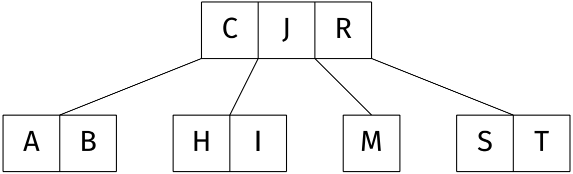


INSERT S

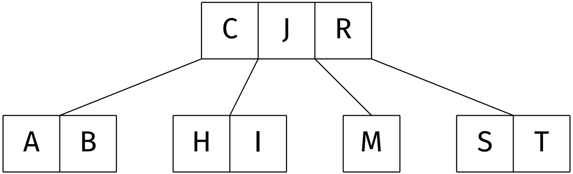
2–3–4 Tree Insertion (VII)



INSERT S

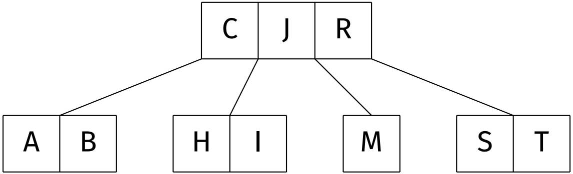


2–3–4 Tree Insertion (VIII)

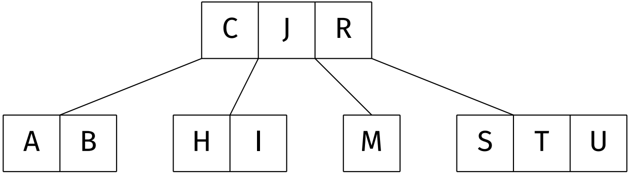


INSERT U

2–3–4 Tree Insertion
(VIII)

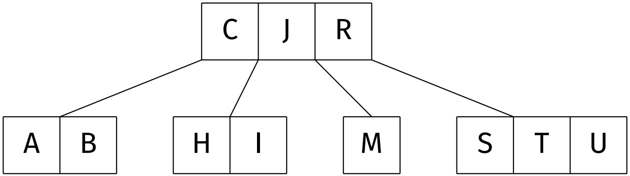


INSERT U



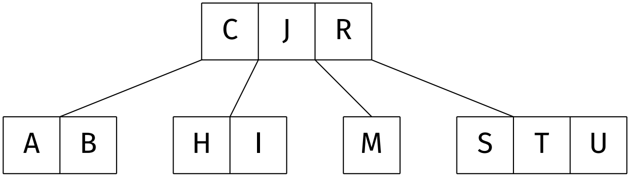
2–3–4 Tree Insertion

(IX)

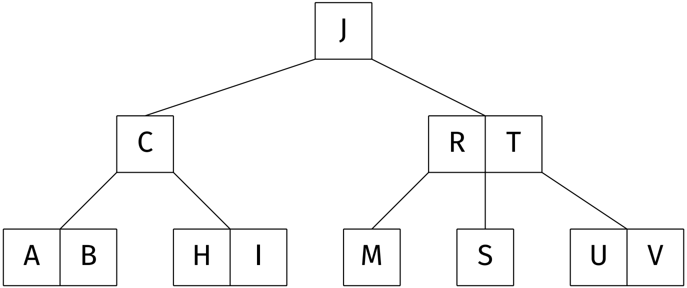


INSERT V

2–3–4 Tree Insertion
(IX)



INSERT V



2–3–4 Trees

C Implementation: Node Structure

```
typedef struct t234_node t234_node;
struct t234_node {
    int order;        // 2, 3, 4
    Item data[3];     // items in node
    t234_node *child[4]; // links to subtrees
};
```

2–3–4 Tree Search

C Implementation

```
Item *t234_search (t234_node *tree, Item it)
{
    if (tree == NULL) return NULL;
    int i, diff = 0;
    for (i = 0; i < tree->order - 1; i++) {
        diff = item_cmp (it, tree->data[i]);
        if (diff <= 0) break;
    }
    if (diff == 0) return &(t->data[i]);
    else return t234_search (t->child[i], k);
}
```

2–3–4–M

Why stop with just 2-, 3-, and 4-nodes?
If we allow nodes to hold $M/2$ to M items,
we have a B-tree.

commonly used in DBMS, FS, ...
where a node represents a disk page.

Red-Black Trees

2–3–4, unplugged (I)

red-black trees are a representation of 2–3–4 trees
using only plain old BST nodes;
each node needs one extra value to encode link type,
but we no longer have to deal with different kinds of nodes.

plain old binary search tree search works, unmodified
get benefits of 2–3–4 tree self-balancing on insert, delete
... with great complexity in insertion/deletion.

Red-Black Trees

2–3–4, unplugged (II)

red links combine nodes to represent 3- and 4-nodes;
effectively, child along red link is a 2–3–4 neighbour.
black links are analogous to 'ordinary' child links.
some texts call these 'red nodes' and 'black nodes'

THE RULES:
each link is either red or black
no two red links appear consecutively on any path
all paths from root to leaf have same number of black links