>>

Red-black Trees

^ >>

❖ Red-Black Trees

Red-black trees are a representation of 2-3-4 trees using BST nodes.

- each node needs one extra value to encode link type
- but we no longer have to deal with different kinds of nodes

Link types:

- red links ... combine nodes to represent 3- and 4-nodes
- black links ... analogous to "ordinary" BST links (child links)

Advantages:

- standard BST search procedure works unmodified
- get benefits of 2-3-4 tree self-balancing (although deeper)

<< ∧ >>

❖ ... Red-Black Trees

Definition of a red-black tree

- a BST in which each node is marked **red** or **black**
- no two red nodes appear consecutively on any path
- a red node corresponds to a 2-3-4 sibling of its parent
- a black node corresponds to a 2-3-4 child of its parent

Balanced red-black tree

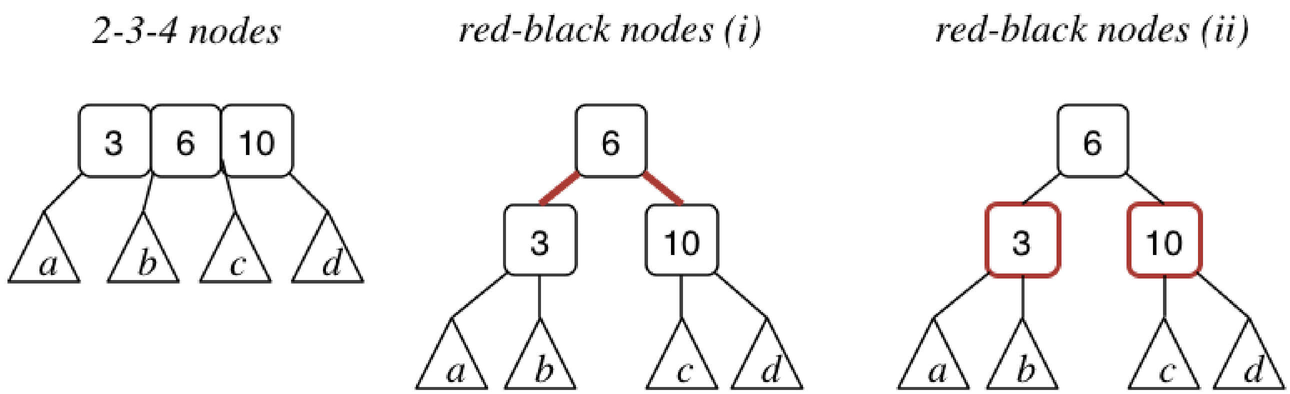
- all paths from root to leaf have same number of black nodes

Insertion algorithm: avoids worst case *O(n)* behaviour

Search algorithm: standard BST search

❖ ... Red-Black Trees

Representing 4-nodes in red-black trees:

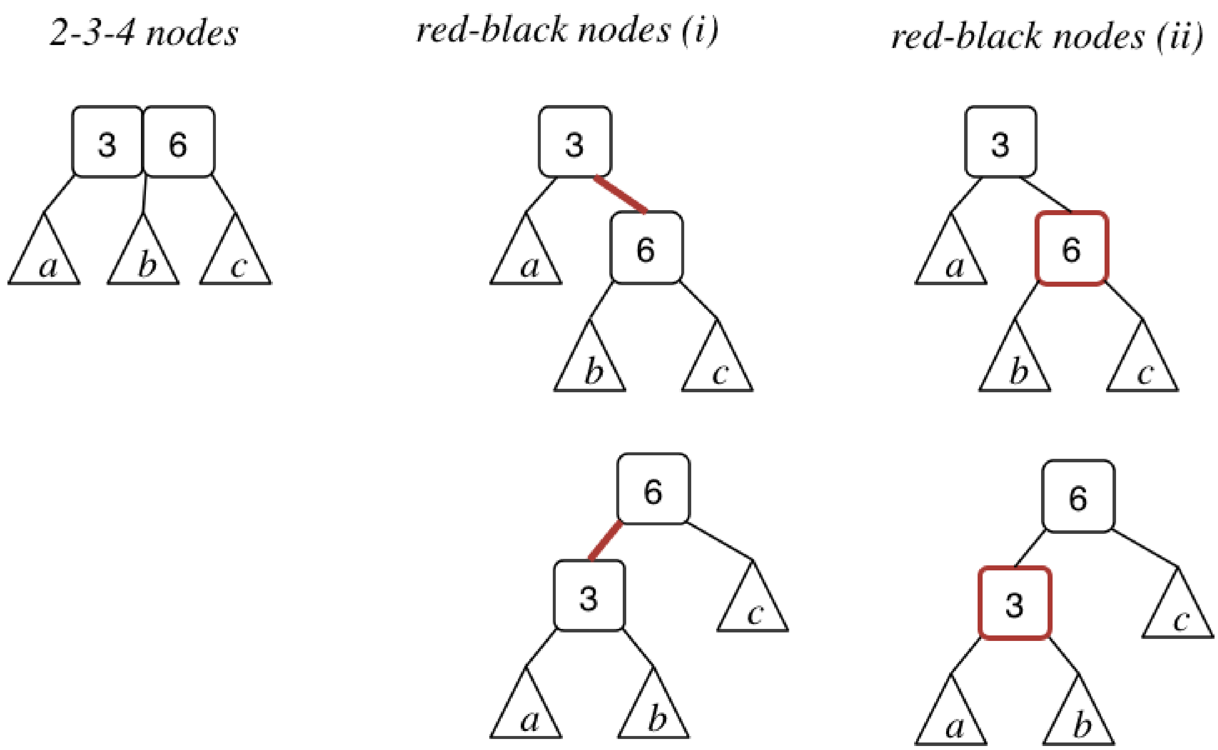


Some texts colour the links rather than the nodes.

<< ∧ >>

❖ ... Red-Black Trees

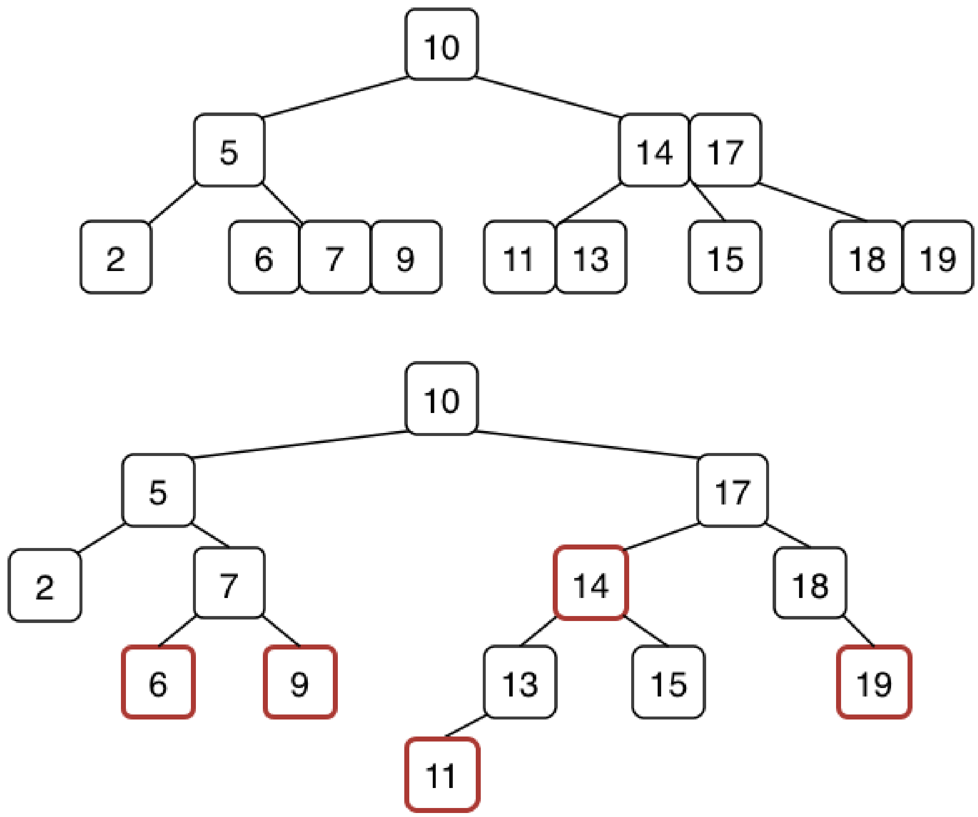
Representing 3-nodes in red-black trees (two possibilities):



COMP2521 20T2 ◊ Red-black Trees [4/18]

❖ ... Red-Black Trees

Equivalent trees (one 2-3-4, one red-black):



<< ^ >>

❖ ... Red-Black Trees

Red-black tree implementation:

```
typedef enum {RED,BLACK} Colour;
typedef struct node *RBTree;
typedef struct node {
   int    data;    // actual data
   Colour colour;  // relationship to parent
   RBTree left;    // left subtree
   RBTree right;   // right subtree
} node;

#define colour(tree) ((tree) != NULL && (tree)->colour)
#define isRed(tree)  ((tree) != NULL && (tree)->colour == RED)
```

RED = node is part of the same 2-3-4 node as its parent (sibling)

BLACK = node is a child of the 2-3-4 node containing the parent

<< ^ >>

❖ ... Red-Black Trees

New nodes are always red ...

```
RBTree newNode(Item it) {
   RBTree new = malloc(sizeof(Node));
   assert(new != NULL);
   data(new) = it;
   color(new) = RED;
   left(new) = right(new) = NULL;
   return new;
}
```

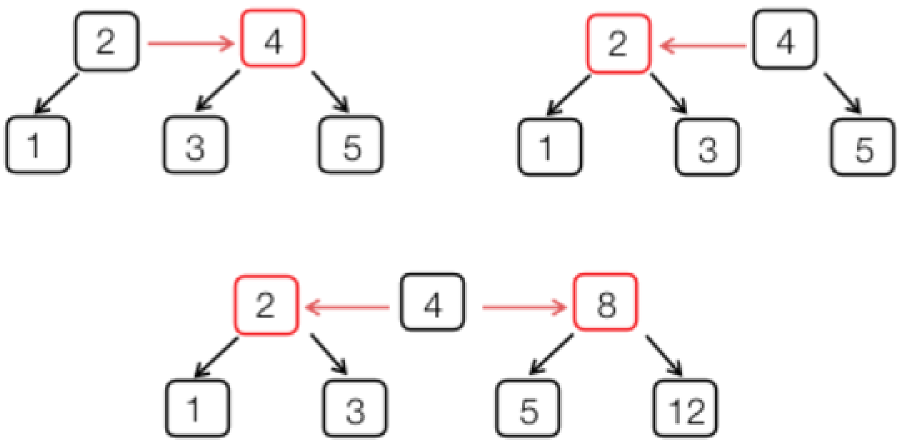
.. because they're always inserted into a leaf node

<< ^ >>

❖ ... Red-Black Trees

`Node.red` allows us to distinguish links

- black = parent node is a "real" parent
- red = parent node is a 2-3-4 neighbour



❖ Searching in Red-black Trees

Search method is standard BST search:

```
searchRedBlack(tree,item):
|  Input  tree, item
|  Output true if item found in tree,
|         false otherwise
|
|  if tree is empty then
|     return false
|  else if item < data(tree) then
|     return SearchRedBlack(left(tree),item)
|  else if item > data(tree) then
|     return SearchRedBlack(right(tree),item)
|  else   // found
|    return true
|  end if
```

❖ Insertion in Red-Black Trees

Insertion is more complex than for standard BSTs

- need to recall direction of last branch (L or R)
- need to recall whether parent link is red or black
- splitting/promoting implemented by rotateLeft/rotateRight

Several cases to consider depending on colour/direction combinations

<< ^ >>

❖ ... Insertion in Red-Black Trees

High-level description of insertion algorithm:

```
insertRedBlack(tree,item):
|  Input  red-black tree, item
|  Output tree with item inserted
|
|  tree = insertRB(tree,item,false)
|  colour(tree) = BLACK
|  return tree
```

This function acts as a "wrapper" around the recursive function.

Having restructured the tree, it then makes the root node `BLACK`

<< ∧ >>

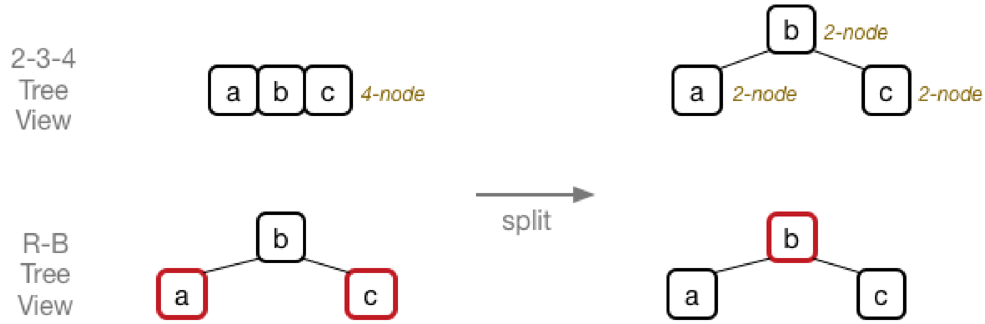
❖ ... Insertion in Red-Black Trees

Overview of the recursive function ...

```
insertRB(tree,item,inRight):
|  Input  tree, item, inRight
|         // inRight = direction of last branch
|  Output tree with item inserted
|
|  if tree is empty then return newNode(item)
|  if data(tree) = item then return tree
|  if isRed(left(tree)) ∧ isRed(right(tree)) then
|     split 4-node
|  end if
|  recursive insert cases (cf. regular BST)
|  re-arrange links/colours after insert
|  return modified tree
```

❖ ... Insertion in Red-Black Trees

Splitting a 4-node, in a red-black tree:

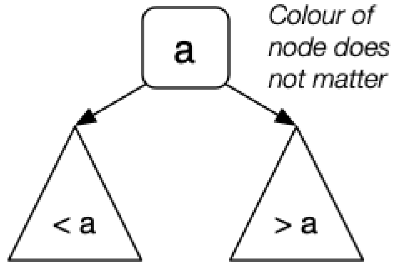


Algorithm:

```
if isRed(left(tree)) ∧ isRed(right(tree)) then
   colour(tree) = RED
   colour(left(tree)) = BLACK
   colour(right(tree)) = BLACK
end if
```

❖ ... Insertion in Red-Black Trees

Simple recursive insert (a la BST):



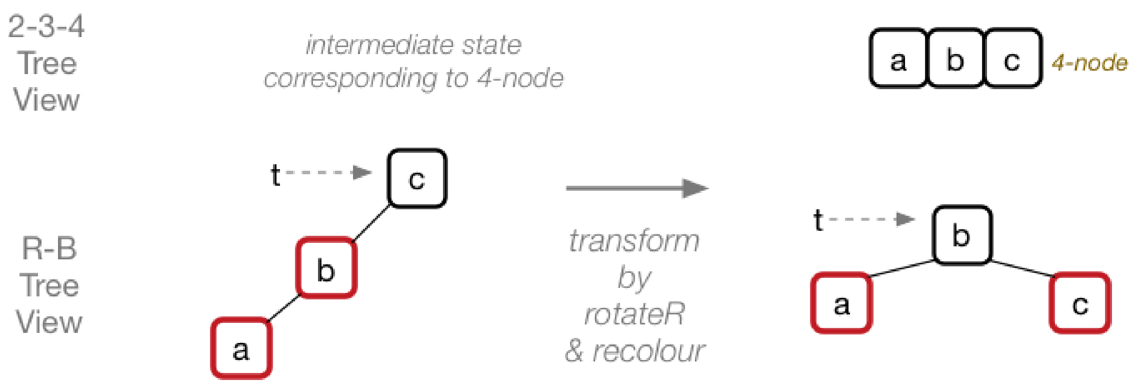
Algorithm:

```
if item < data(tree)) then
   left(tree) = insertRB(left(tree),item,false)
   re-arrange links/colours after insert
else        // item larger than data in root
   right(tree) = insertRB(right(tree),item,true)
   re-arrange links/colours after insert
end if
```

<< ^ >>

❖ ... Insertion in Red-Black Trees

Re-arrangement #1: two successive red links = newly-created 4-node



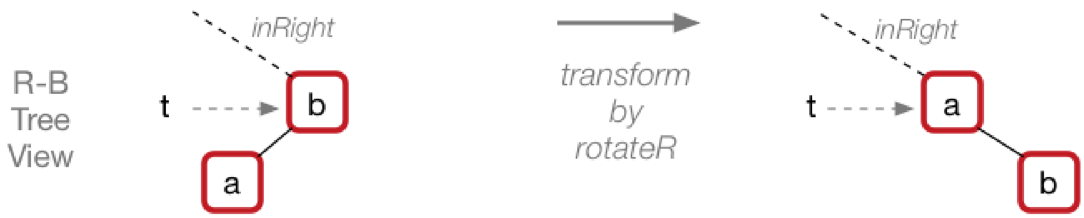
Algorithm:

```
if isRed(left(tree)) ∧ isRed(left(left(tree))) then
   tree=rotateRight(tree)
   colour(tree)=BLACK
   colour(right(tree))=RED
end if
```

<< ∧ >>

❖ ... Insertion in Red-Black Trees

Re-arrangement #2: "normalise" direction of successive red links



Algorithm:

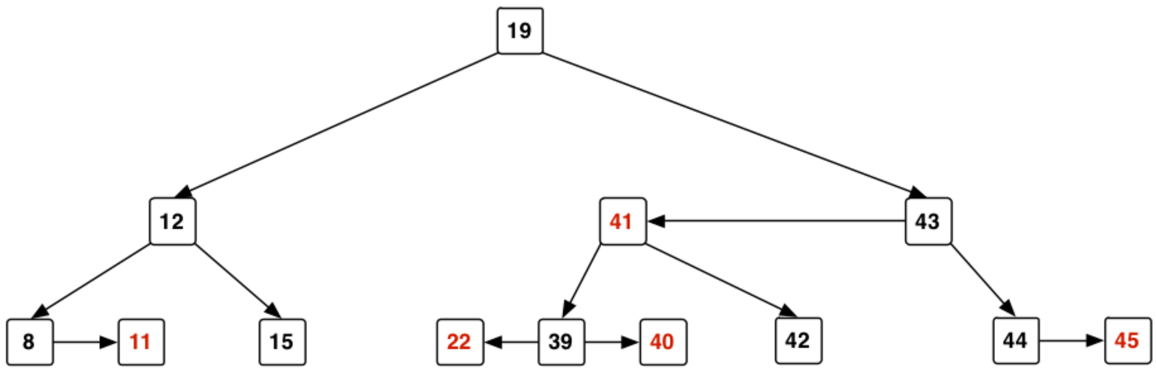
```
if inRight ∧ isRed(tree) ∧ isRed(left(tree)) then
   tree=rotateRight(tree)
end if
```

<< ^ >>

❖ ... Insertion in Red-Black Trees

Example of insertion, starting from empty tree:

```
22, 12, 8, 15, 11, 19, 43, 44, 45, 42, 41, 40, 39
```



To see how built:
www.cs.usfca.edu/~galles/visualization/RedBlack.html

<< ∧

❖ Red-black Tree Performance

Cost analysis for red-black trees:

- tree is well-balanced; worst case search is $O(\log_2 n)$
- insertion affects nodes down one path; max #rotations is $2 \cdot h$
  (where h is the height of the tree)

Only disadvantage is complexity of insertion/deletion code.

Note: red-black trees were popularised by Sedgewick.

Produced: 18 Jun 2020