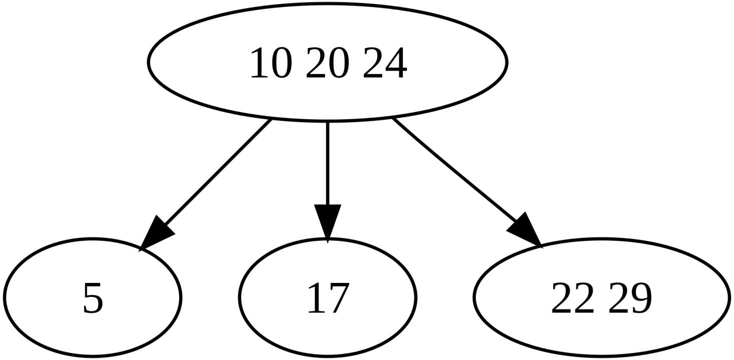
Computer Science 3A - CSC3A10

Lecture 10b: Search Trees

Academy of Computer Science and Software Engineering
University of Johannesburg

- Remedying a Double Red
- Restructuring
- Recolouring
- Red-Black Treee Analysis of Insertion
- Red-Black Tree Deletion
- Remedying a Double Black
- Red-Black Tree Reorganisation

(2,4) Trees



Multi-Way Search Tree

What if we could store two or more Entries together (such as the case in a Binary Search Tree) in hopes of gaining computational performance?

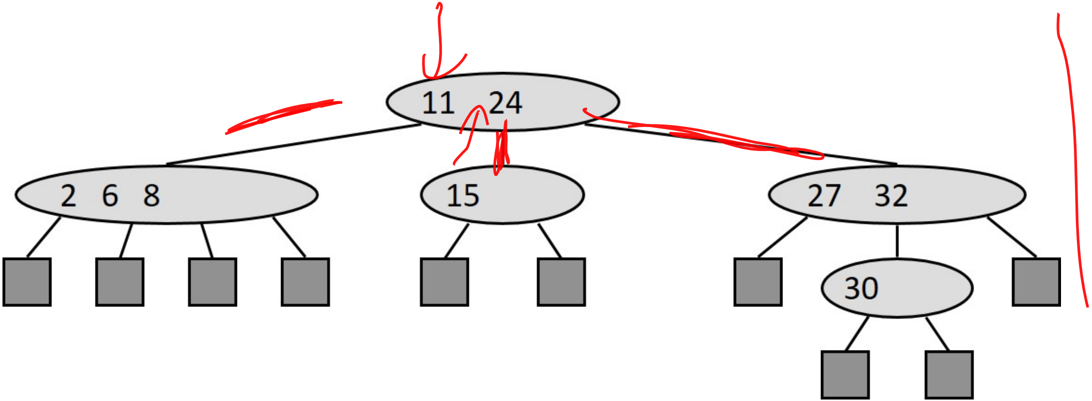
Enter Multi-Way Search Trees

Multi-Way Search Tree II

A multi-way search tree is an ordered tree such that:

- Each internal node has at least two children and stores $d-1$ key-element items (k_i, o_i), where d is the number of children
- For a node with children $v_1 v_2 \ldots v_d$ storing keys $k_1 k_2 \ldots k_{d-1}$
  - keys in the subtree of v_1 are less than k_1
  - keys in the subtree of v_i are between k_{i-1} and $k_i (i = 2, \ldots, d-1)$
  - keys in the subtree of v_d are greater than k_{d-1}
- The leaves store no items and serve as placeholders

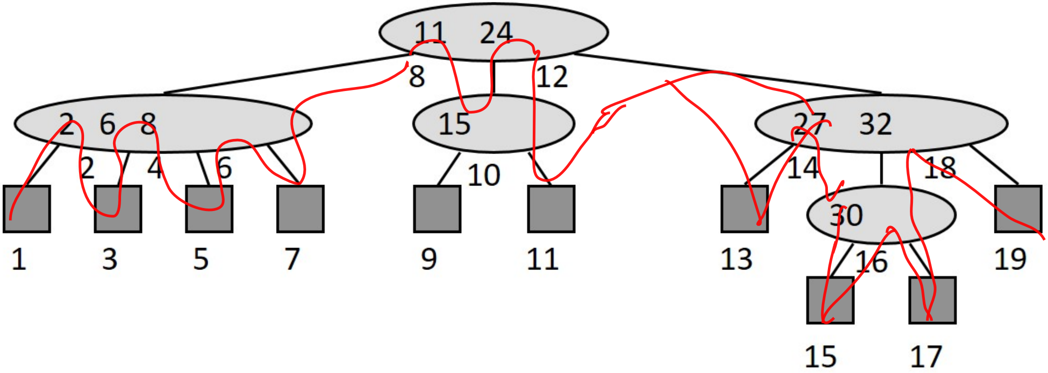
Multi-Way Search Tree II



Multi-Way Inorder Traversal

- We can extend the notion of inorder traversal from binary trees to multi-way search trees
- Namely, we visit item (k_i, o_i) of node v between the recursive traversals of the subtrees of v rooted at children v_i and v_{i+1}
- An inorder traversal of a multi-way search tree visits the keys in increasing order

Multi-Way Inorder Traversal II



Multi-Way Searching

Similar to search in a binary search tree

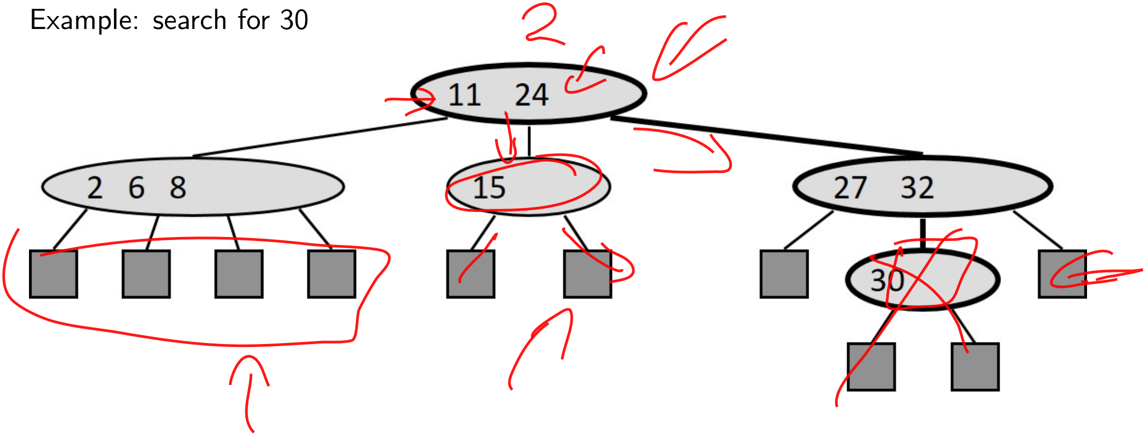
Each internal node with children $v_1 v_2 ... v_d$ and keys $k_1 k_2 ... k_{d-1}$

- $k = k_i (i = 1, ..., d-1)$: the search terminates successfully
- $k < k_1$: we continue the search in child v_1
- $k_{i-1} < k < k_i (i = 2, ..., d-1)$: we continue the search in child v_i
- $k > k_{d-1}$: we continue the search in child v_d

Reaching an external node terminates the search unsuccessfully

Multi-Way Searching

Example: search for 30



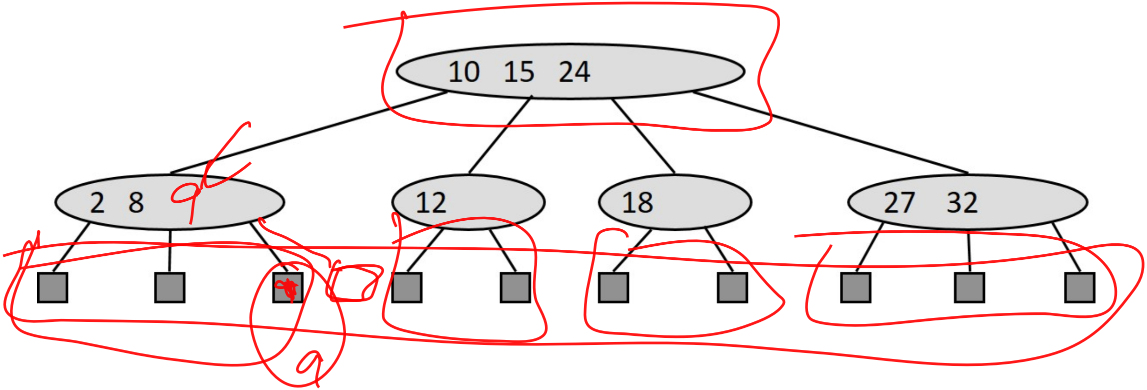
(2,4) Trees

A (2,4) tree (also called 2-4 tree or 2-3-4 tree) is a multi-way search with the following properties

- Node-Size Property: every internal node has at most four children
- Depth Property: all the external nodes have the same depth

Depending on the number of children, an internal node of a (2,4) tree is called a 2-node, 3-node or 4-node

(2,4) Trees II



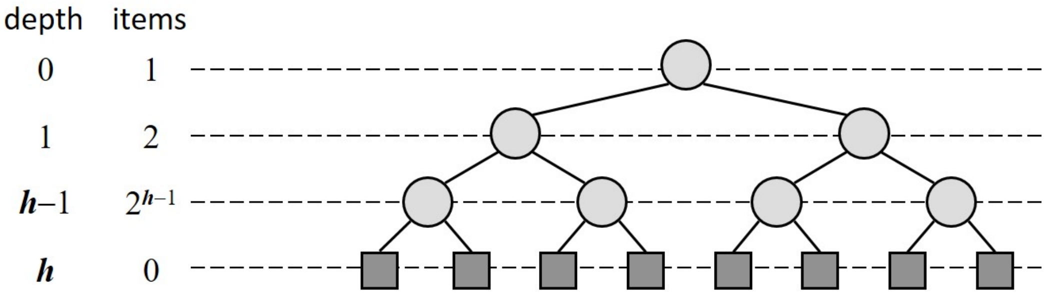
Height of a (2,4) Tree

Theorem: A (2,4) tree storing n items has height $O(logn)$
Proof:

- Let h be the height of a (2,4) tree with n items
- Since there are at least 2^i items at depth $i = 0, ..., h - 1$ and no items at depth h, we have $n \geq 1 + 2 + 4 + ... + 2^{h-1} = 2^h - 1$
- Thus, $h \leq log(n + 1)$

Searching in a (2,4) tree with n items takes $O(logn)$ time

Height of a (2,4) Trees II



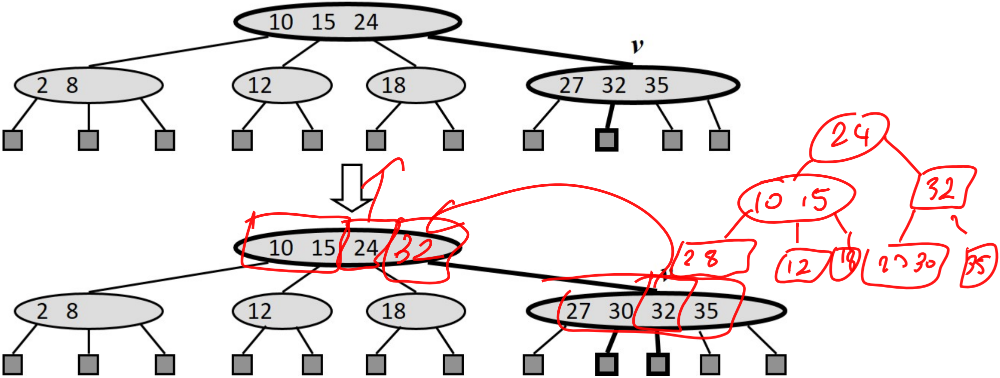
(2,4) Tree Insertion

We insert a new item (k, o) at the parent v of the leaf reached by searching for k

- We preserve the depth property but
- We may cause an overflow (i.e., node v may become a 5-node)

(2,4) Tree Insertion II

Example: inserting key 30 causes an overflow



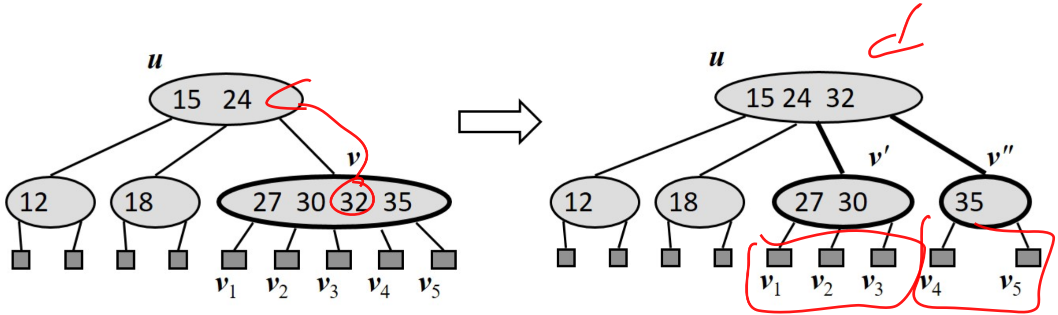
Overflow and Split

We handle an overflow at a 5-node v with a split operation:

- let $v_1 \ldots v_5$ be the children of v and $k_1 \ldots k_4$ be the keys of v
- node v is replaced nodes v' and v"
  - v' is a 3-node with keys $k_1 k_2$ and children $v_1 v_2 v_3$
  - v" is a 2-node with key k_4 and children $v_4 v_5$
- key k_3 is inserted into the parent u of v (a new root may be created)

The overflow may propagate to the parent node u

Overflow and Split II



(2,4) Insert Algorithm

```
Algorithm insert(k, o)
Step 1. We search for key k to locate the insertion node v
Step 2. We add the new entry (k, o) at node v
Step 3. while overflow(v)
    if isRoot(v)
      create a new empty root above v
    v = split(v)
```

insert(k,o)

Analysis of Insertion

Let T be a (2,4) tree with n items

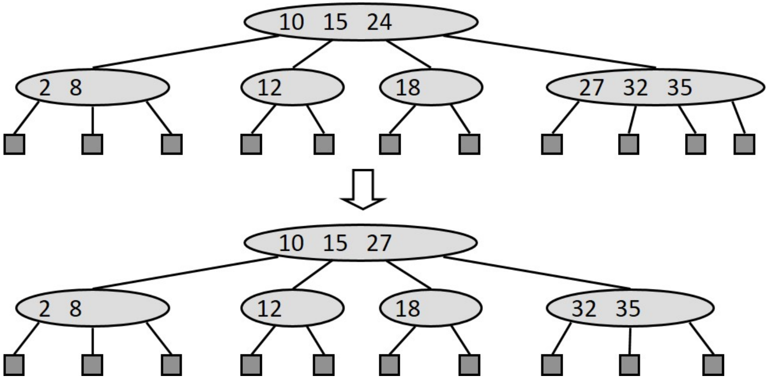
- Tree T has $O(logn)$ height
- Step 1 takes $O(logn)$ time because we visit $O(logn)$ nodes
- Step 2 takes $O(1)$ time
- Step 3 takes $O(logn)$ time because each split takes $O(1)$ time and we perform $O(logn)$ splits

Thus, an insertion in a (2,4) tree takes $O(logn)$ time

(2,4) Tree Deletion

- We reduce deletion of an entry to the case where the item is at the node with leaf children
- Otherwise, we replace the entry with its inorder successor (or, equivalently, with its inorder predecessor) and delete the latter entry

(2,4) Tree Deletion II



Underflow and Fusion

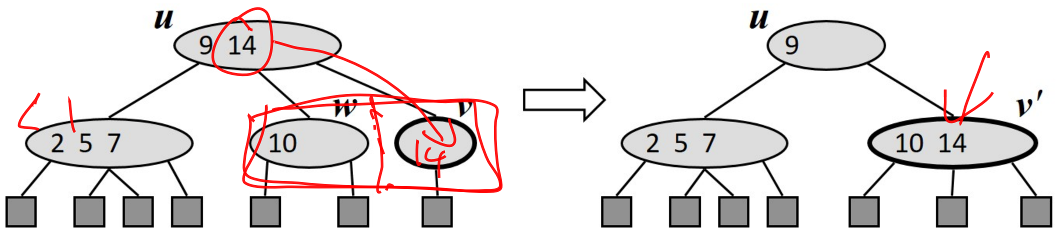
- Deleting an entry from a node v may cause an underflow, where node v becomes a 1-node with one child and no keys
- To handle an underflow at node v with parent u, we consider two cases:

Underflow and Fusion II

Case 1: the adjacent siblings of v are 2-nodes

- Fusion operation: we merge v with an adjacent sibling w and move an entry from u to the merged node v'
- After a fusion, the underflow may propagate to the parent u

Underflow and Fusion III



Underflow and Transfer

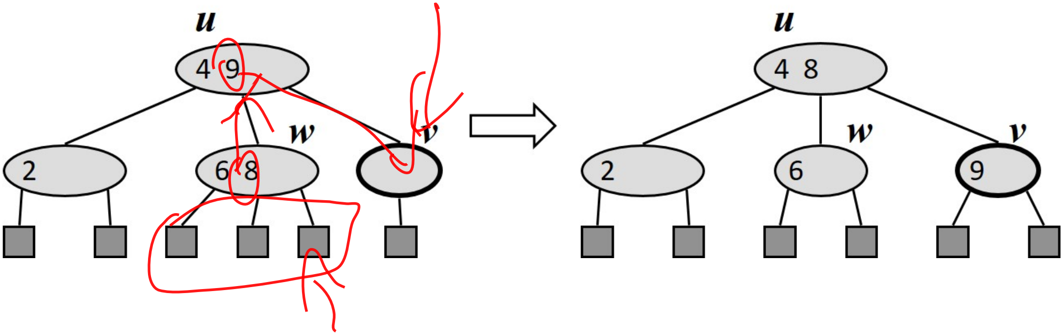
Case 2: an adjacent sibling w of v is a 3-node or a 4-node

Transfer operation:

1. we move a child of w to v
2. we move an item from u to v
3. we move an item from w to u

After a transfer, no underflow occurs

Underflow and Transfer II



Analysis of Deletion

Let T be a (2,4) tree with n items (therefore Tree T has $O(logn)$ height)
In a deletion operation

- We visit $O(logn)$ nodes to locate the node from which to delete the entry
- We handle an underflow with a series of $O(logn)$ fusions, followed by at most one transfer
- Each fusion and transfer takes $O(1)$ time

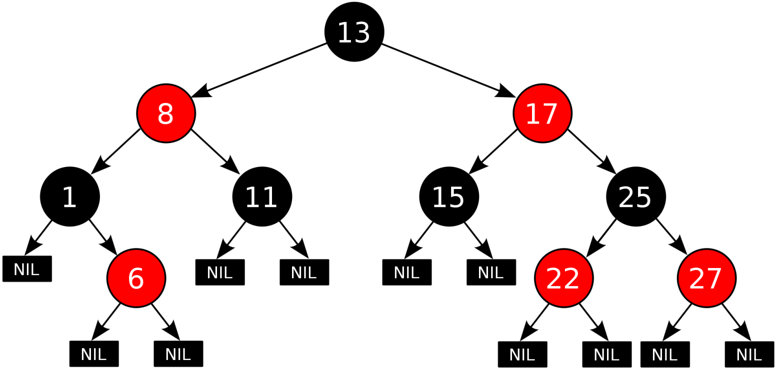
Thus, deleting an item from a (2,4) tree takes $O(logn)$ time

Implementing a Dictionary

Comparison of efficient dictionary implementations

	Search	**Insert**	**Delete**	**Notes**
Hash Table	1 expected	1 expected	1 expected	No ordered dictionary methods Simple to implement
Skip List	$\log n$ high prob.	$\log n$ high prob.	$\log n$ high prob.	Randomized insertion Simple to implement
(2,4) Tree	$\log n$ worst-case	$\log n$ worst-case	$\log n$ worst-case	Complex to implement

Red-Black Trees



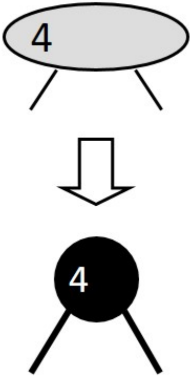
From (2,4) to Red-Black Trees

A red-black tree is a representation of a (2,4) tree by means of a binary tree whose nodes are coloured red or black

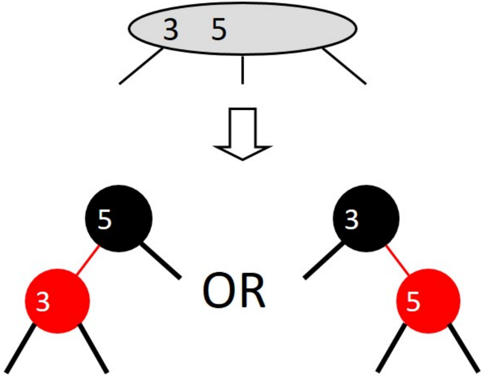
In comparison with its associated (2,4) tree, a red-black tree has

- same logarithmic time performance
- simpler implementation with a single node type

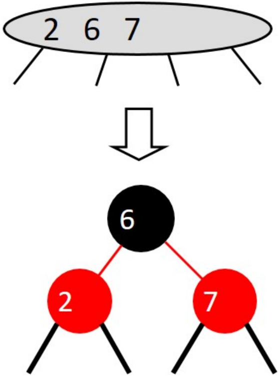
From (2,4) to Red-Black Trees II



From (2,4) to Red-Black Trees III



From (2,4) to Red-Black Trees IV

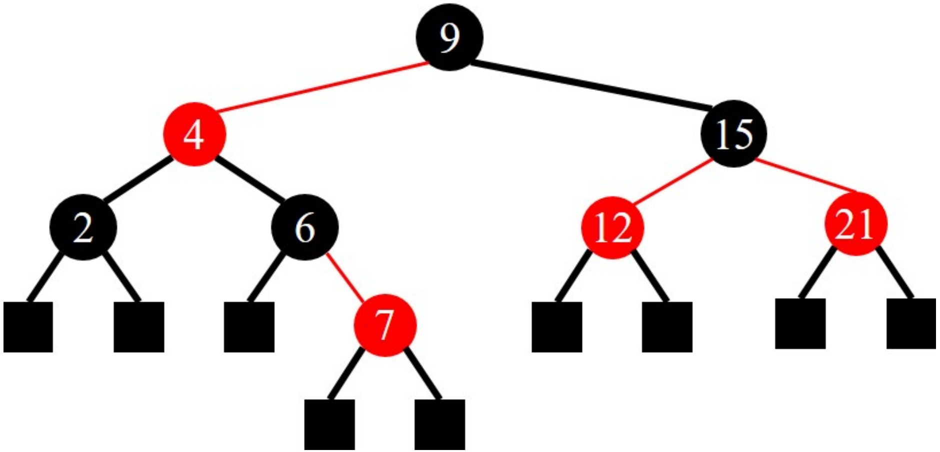


Red-Black Tree Properties

A red-black tree can also be defined as a binary search tree that satisfies the following properties:

- **Root Property**: the root is black
- **External Property**: every leaf is black
- **Internal Property**: the children of a red node are black
- **Depth Property**: all the leaves have the same black depth

Red-Black Tree Properties II



Height of a Red-Black Tree

Theorem: A red-black tree storing n entries has height $O(\log n)$
Proof: The height of a red-black tree is at most twice the height of its associated (2,4) tree, which is $O(\log n)$

- The search algorithm for a binary search tree is the same as that for a binary search tree
- By the above theorem, searching in a red-black tree takes $O(\log n)$ time

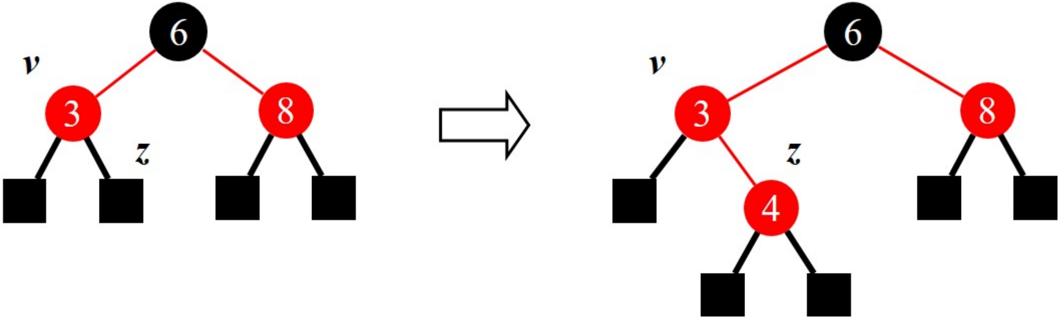
Red-Black Tree Insertion

To perform operation insert(k, o), we execute the insertion algorithm for binary search trees and color red the newly inserted node z unless it is the root

- We preserve the root, external, and depth properties
- If the parent v of z is black, we also preserve the internal property and we are done
- Else (v is red) we have a double red (i.e., a violation of the internal property), which requires a reorganization of the tree

Red-Black Tree Insertion II

Example where the insertion of 4 causes a double red:



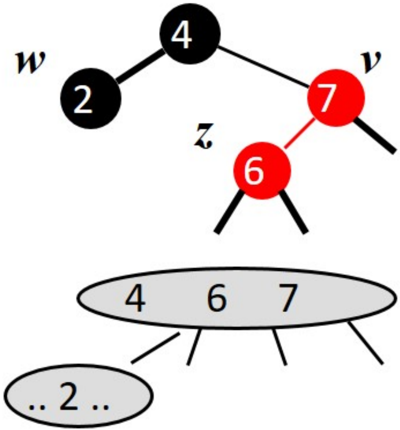
Remedying a Double Red

Consider a double red with child z and parent v, and let w be the sibling of v

Case 1: w is black

- The double red is an incorrect replacement of a 4-node
- Restructuring: we change the 4-node replacement

Remedying a Double Red II

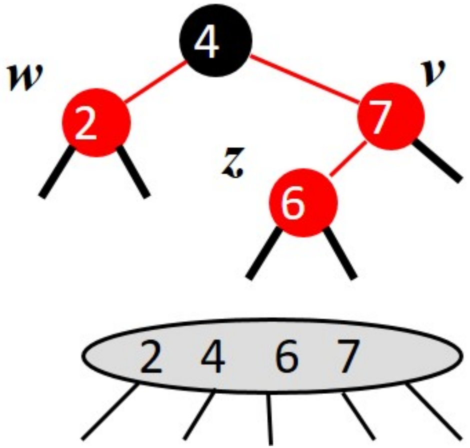


Remedying a Double Red III

Case 2: w is red

- The double red corresponds to an overflow
- Recolouring: we perform the equivalent of a split

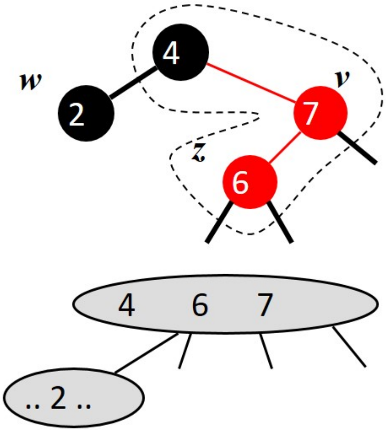
Remedying a Double Red IV

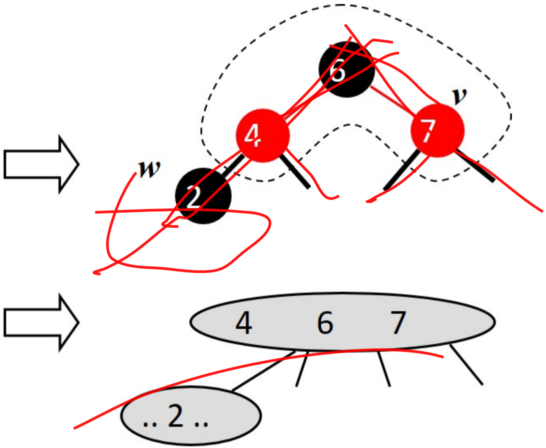


Restructuring

- A restructuring remedies a child-parent double red when the parent red node has a black sibling
- It is equivalent to restoring the correct replacement of a 4-node
- The internal property is restored and the other properties are preserved

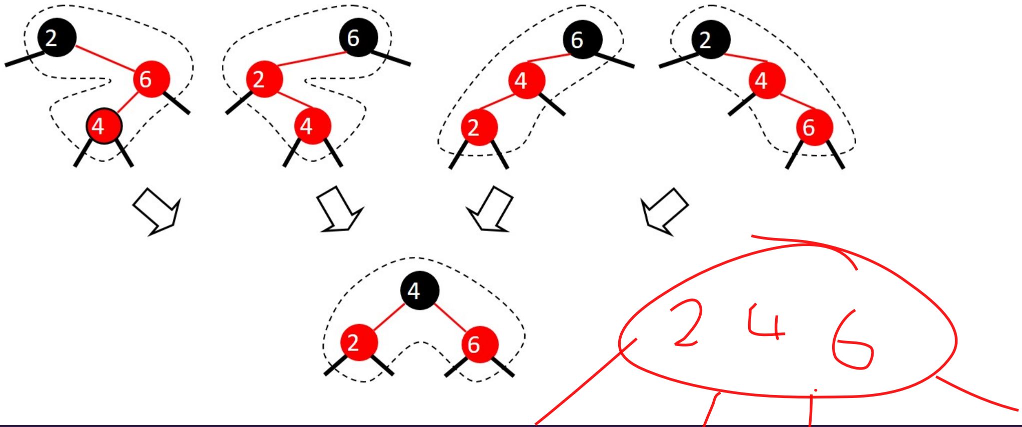
Restructuring II





Restructuring III

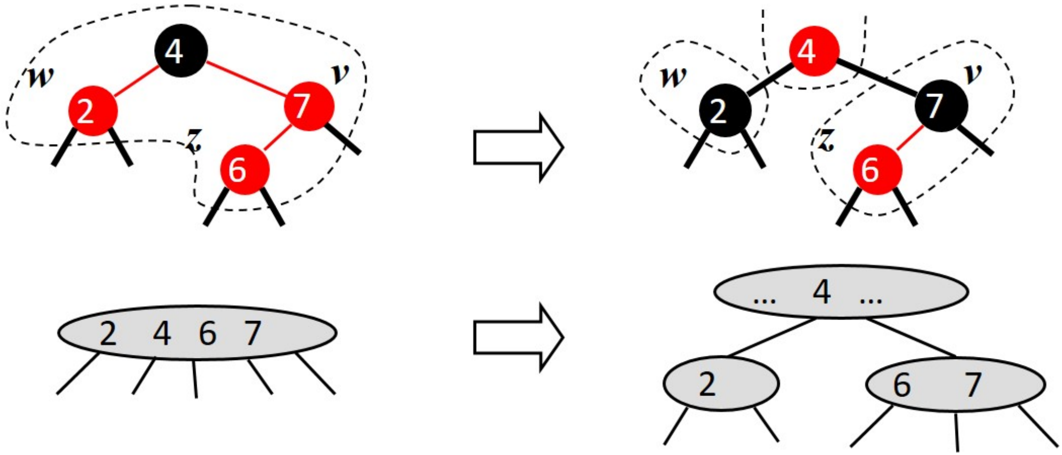
There are four restructuring configurations depending on whether the double red nodes are left or right children



Recolouring

- A recolouring remedies a child-parent double red when the parent red node has a red sibling
- The parent v and its sibling w become black and the grandparent u becomes red, unless it is the root
- It is equivalent to performing a split on a 5-node
- The double red violation may propagate to the grandparent u

Recolouring II



Red-Black Insert Algorithm

```
Algorithm insert(k, o)
Step 1. We search for key k to locate the insertion node z

Step 2. We add the new entry (k, o) at node z and colour z red

Step 3. while doubleRed(z)
  if isBlack(sibling(parent(z)))
    z = restructure(z)
  return
  else { sibling(parent(z) is red }
   z = recolour(z)
```

insert(k,o)

Analysis of Insertion

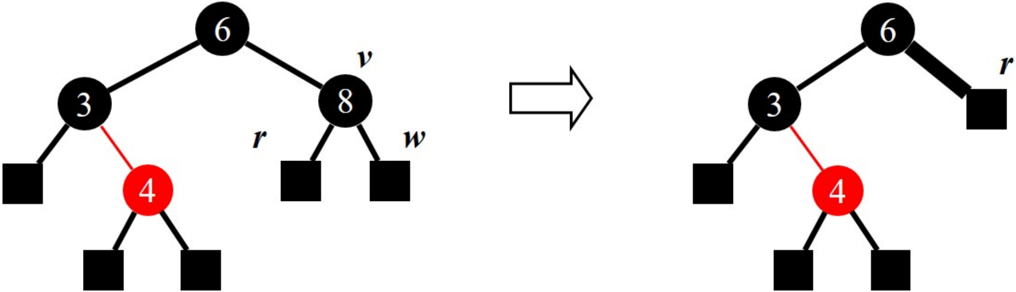
- Recall that a red-black tree has $O(logn)$ height
- Step 1 takes $O(logn)$ time because we visit $O(logn)$ nodes
- Step 2 takes $O(1)$ time
- Step 3 takes $O(logn)$ time because we perform
  - $O(logn)$ recolorings, each taking $O(1)$ time, and
  - at most one restructuring taking $O(1)$ time
- Thus, an insertion in a red-black tree takes $O(logn)$ time

Red-Black Tree Deletion

- To perform operation remove(k), we first execute the deletion algorithm for binary search trees
- Let v be the internal node removed, w the external node removed, and r the sibling of w
  - If either v of r was red, we colour r black and we are done
  - Else (v and r were both black) we colour r double black, which is a violation of the internal property requiring a reorganization of the tree

Red-Black Tree Deletion II

Example where the deletion of 8 causes a double black:



Remedying a Double Black

The algorithm for remedying a double black node w with sibling y considers three cases:

- **Case 1**: y is black and has a red child: We perform a restructuring, equivalent to a transfer and we are done
- **Case 2**: y is black and its children are both black: We perform a recolouring, equivalent to a fusion, which may propagate up the double black violation
- **Case 3**: y is red: We perform an adjustment, equivalent to choosing a different representation of a 3-node, after which either Case 1 or Case 2 applies

Deletion in a red-black tree takes $O(logn)$ time

Red-Black Tree Reorganisation

Insertion	*Remedy double red*	
Red-Black Tree Action	**(2,4) Tree action**	**Result**
Restructuring	Change of 4-node representation	Double red removed
Recolouring	Split	Double red removed or propagated up
Deletion	*Remedy double black*	
Red-Black Tree Action	**(2,4) Tree action**	**Result**
Restructuring	Transfer	Double black removed
Recolouring	Fusion	Double black removed or propagated up
Adjustment	Change of 3-node representation	Restructuring or re-colouring follows