

# Introduction to Algorithms and Data Structures

## Lecture 9: Balanced trees

John Longley

School of Informatics  
University of Edinburgh

October 2021

## Tackling that 'worst case'

- ▶ We've considered hash table implementations of sets/dictionaries in which **lookup/insert/delete** are usually fast – but worst case time for all operations is  $\Theta(n)$ .
- ▶ For lists (a.k.a. **vectors**): some operations have worst-case time  $\Theta(1)$ , but **insert/delete** are  $\Theta(n)$  even in average case.

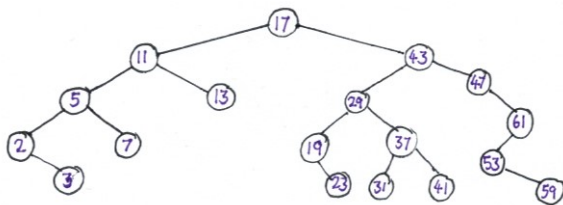
??? Can we find implementations of sets/dictionaries/lists for which *all* operations have acceptable worst-case times ???

**This lecture:** We'll see that 'balanced trees' (e.g. **red-black** trees) achieve this: all ops have worst-case and average time  $\Theta(\lg n)$ .

Will do sets/dictionaries here; ideas can also be applied to lists.



## Representing sets by trees



Consider **binary trees**: each node  $x$  has a **left** and a **right** branch, each of which may be **null** or a pointer to a **child** node.

(Implementation detail: should use **doubly linked** tree structures.)

Write  $L(x)$ ,  $R(x)$  for **left** and **right subtrees** at  $x$  (may be empty).

Label nodes with **keys** (e.g. integers or strings) in such a way that for every node  $x$  we have

$$\forall y \in L(x). y.\text{key} < x.\text{key}, \quad \forall z \in R(x). x.\text{key} < z.\text{key}$$

Can use such trees to represent **sets of keys**.

(For **dictionaries**, just add value component to each node.)

## Implementing contains/lookup

This is easy. Let a node  $x$  stand for the tree rooted at  $x$ .

**contains'**( $x, k$ ):

if  $x = \text{null}$  then return False

else if  $x.\text{key} = k$  then return True

else if  $k < x.\text{key}$  then return **contains'**( $x.\text{left}, k$ )

else return **contains'**( $x.\text{right}, k$ )

**contains**( $k$ ):

return **contains'**( $\text{root}, k$ )

Suppose the tree has  $n$  nodes and is **perfectly balanced**, i.e. all non-leaf nodes have 2 children, and all leaf nodes are at the same depth  $d$ . (Possible only if  $n = 2^{d+1} - 1$ .)

Then  $d = \lfloor \lg n \rfloor$ , so **contains** will take time  $O(\lg n)$ .

More generally, for trees that are 'not too unbalanced' (say max depth  $\leq 2\lceil \lg n \rceil$ ), can say **contains** take  $O(\lg n)$  time.

However, **worst case is still  $\Theta(n)$ !**

## Insert on binary trees

This too is easy: walk down tree to find where  $k$  wants to go, and create a new leaf node for it.

**insert'**( $x, k$ ):

if  $x.\text{key} = k$  then return KeyAlreadyPresent

else if  $k < x.\text{key}$  then

if  $x.\text{left} = \text{null}$  then  $x.\text{left} = \text{new Node}(k)$

else **insert'**( $x.\text{left}, k$ )

else

if  $x.\text{right} = \text{null}$  then  $x.\text{right} = \text{new Node}(k)$

else **insert'**( $x.\text{right}, k$ )

**insert**( $k$ ):

if  $\text{root} = \text{null}$  then  $\text{root} = \text{new Node}(k)$

else return **insert'**( $\text{root}, k$ )

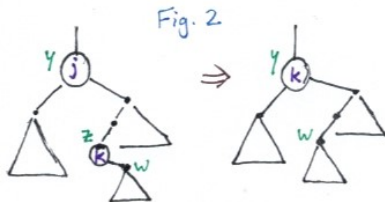
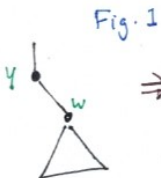
Again,  $O(\lg n)$  time if tree not too unbalanced,  $\Theta(n)$  in worst case.

NB. Nothing here to guard against tree *becoming* unbalanced!

# Delete on binary trees

A bit more subtle. To perform **delete(j)**:

- ▶ Locate the node  $y$  bearing  $j$  (assume there is one).
- ▶ If  $y$  has **no children**, can just delete it.
- ▶ If  $y$  has **one child**, easy to elide the node  $y$  (Fig. 1).
- ▶ If  $y$  has **two children**:
  - ▶ Locate **leftmost** node in  $R(y)$ , i.e. starting at  $y$ , turn right, then left as often as possible. This finds the node  $z$  bearing the smallest key in  $R(y)$  (call it  $k$ ).
  - ▶ Copy  $z$ .key to  $y$ .key.
  - ▶ If  $z$  has a right child, elide  $z$ , otherwise just delete  $z$ . (Fig. 2).



Same runtime characteristics.

# Balanced tree representations

## General strategy:

- ▶ Work with some special class of trees (red-black trees) that are guaranteed to be ‘not too unbalanced’, so that all operations will take time  $O(\lg n)$ .
- ▶ Whenever an **insert/delete** threatens to take us outside this class, do some ‘re-balancing’ work to restore it.  
**Clever bit:** Can arrange that this re-balancing work also takes just  $O(\lg n)$  time!

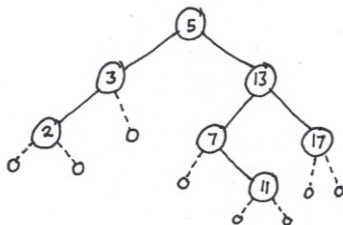
This leads to worst-case  $O(\lg n)$  time for all operations.

This broad strategy works for several classes of trees:  
red-black trees, AVL trees, 2-3 trees, . . .

We choose **red-black** trees as they’re covered in detail in CLRS.

## Small preliminary: adding trivial nodes

For mathematical convenience, extend our trees so that original *null* branches now point to **trivial nodes**, with no children and bearing no key. Original nodes are **proper nodes**.



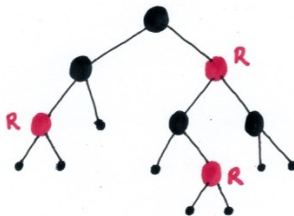
Call this an **extended tree**.

Just makes rules easier to state.

Wouldn't need these trivial nodes in an implementation.



## Red-black trees



Work with extended trees as above.

In a red-black tree, every node is coloured **red** or **black**.

### Tree rules:

- ▶ Root and all (trivial) leaves are black.
- ▶ All paths root  $\rightarrow$  leaf contain same number  $b$  of blacks.
- ▶ On a path root  $\rightarrow$  leaf, never have two reds in a row.

So min possible path length is  $b$ , and max is  $2b - 1$ .

Red-black trees are **not too unbalanced**:

Can show  $b \leq \lg(n+1) + 1$ , so all path lengths  $\leq 2\lg(n+1) + 1$ .

So **contains** works as usual with worst-case time  $\Theta(\lg n)$ .

## Insert for red-black trees

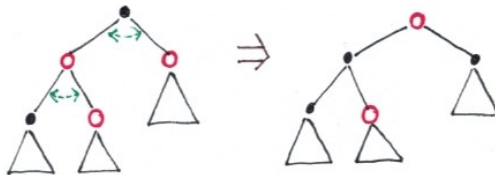
Can **insert** a key-bearing node as usual (adding two trivial leaves).  
**Colour it red.** This all takes  $O(\lg n)$  time.

**Problem:** Resulting tree might no longer be a legal red-black tree:

- ▶ New red node might have red parent (2 reds in succession), or
- ▶ (Trivial case) New red node might be root (should be black).

So need to apply a **fix-up** operation to restore red-black-ness.

Main ingredient is the **red-uncle rule**:



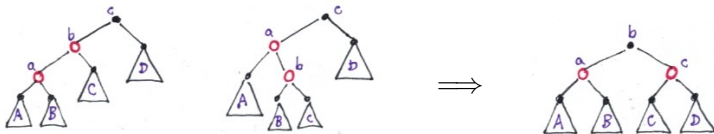
(Just colour-flipping: fast. No rewiring involved!)

## Insert fix-up, continued

Applying the red-uncle rule pushes a red upward, so may result in another double-red higher up.

So we **apply the red-uncle rule as often as possible** (will be at most  $O(\lg n)$  times). We'll then be in one of three **endgame scenarios**:

1. Problem cured: tree now legal.
2. Red pushed to root: **turn it black**.  
Adds 1 to all black-lengths.
3. Have some configuration involving a black with 4 'nearest black descendants'. Replace by obvious 'balanced' version:



$O(1)$  amount of rewiring.

Note order of constituents is preserved: AaBbCcD.

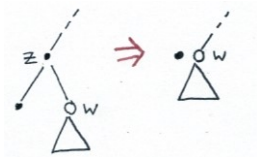
(Subtrees A,B,C,D may be empty.)

## Delete for red-black trees

Just the main ideas: won't give full details.

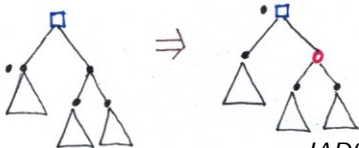
Do delete as usual: this involves removing some proper node  $z$ .

**Problem:** All paths must have same black-length. So if  $z$  was black, want to remove  $z$  but keep the 'blackness'.



**Easy case:** Node it haunts is now red: can just turn it black.

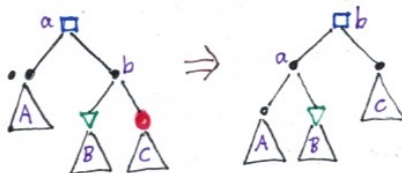
**Wandering black rule:** apply this as often as possible (will be  $O(\lg n)$  times).



## Delete for red-black trees: the endgame

Finitely many endgame scenarios, each fixable in  $O(1)$  time. E.g.

- ▶ Floating black haunts a red node: **turns it black**.
- ▶ Floating black reaches root: just remove it.
- ▶ We're in some other fixable scenario, e.g.



Blue square and green triangle are colour variables.

- ▶ 4 other scenarios like this: see CLRS 13 for full details.

## Balanced trees: conclusion

- ▶ Balanced trees offer a way of implementing sets/dictionaries so that all operations have worst-case time  $O(\lg n)$ . (Idea can be applied to lists too.)
- ▶ Not much to choose between red-black and AVL trees. AVL are 'more balanced' (better for lookup); red-blacks possibly have faster insert/delete.
- ▶ Red-black trees used in practice:
  - ▶ Linux [completely fair scheduler](#)
  - ▶ Java 8 [HashMap](#) class: dictionary via bucket-style hash table, but each bucket is a red-black tree rather than a linked list. Retains excellent typical-case performance of hash tables, but kills off the nasty 'worst cases'.

**Reading:** CLRS 12.1-12.3, 13.1-13.3

**Today's music:** John Williams, *Hedwig's Theme*  
(from the Harry Potter films).