Search Trees

- Today we are going to look at search trees.
- Seach trees are a way of storing sets of things.
- We will look at an optimisation of binary search trees, red-black trees.

Pattern Matching on trees

```
datatype Tree = Empty | Node of int*Tree*Tree

fun count Empty = 0
 |  count (Node(_,t1,t2)) = 1+count t1 + count t2
```

Notice the _ this is to say we don't care about that value.

Remember to include all the cases.

It is often clearer to use pattern mathcing than lots of nested if statements.

as

```
datatype Tree = Empty | Node of int*Tree*Tree

fun silly Empty = Empty
  | silly (t as Node(i,_,_)) =
     if (i=0) then Empty else t
```

Finding an Element in a Tree

```
datatype T = E | N of int*T*T

fun non_bin_find element E = false
 |  non_bin_find element (N(i,t1,t2)) =
    if element = i then
        true
    else
        non_bin_find element t1
         orelse non_bin_find element t2
```

Worst case?

- If there are N-nodes in the tree, then the worse case search time is $O(n)$.
- How do we make it better?
- Answer: By organising our trees better.

Binary Search Trees Ch-13

- An empty tree is a binary search tree.
- If `s` and `t` are binary search trees then

  ```
  T(i,s,t)
  ```

  is a binary tree if i is greater than than every element in `s` but less than every element in `t`.

To make life easier we will avoid having repeated elements.

Binary Search Trees

If we know that our tree is a binary search tree, we can find element faster.

```
fun member x E = false
  | member x (N(y,s,t)) =
    if x<y then member x s
    else if x>y then member x t
         else true
```

Efficiency of Searching

- Construct a binary search tree with 6-element such that the algorithm will search all 6 nodes for a certain element.
- Construct a search tree of 7 elements, so that the algorithm only every examines at most three nodes.

Building binary search trees

If we want to insert an element in a search tree we have to find the correct branch.

```
fun insert x E = N(x,E,E)
  | insert x (current_tree as (N(y,s,t))) =
     if x<y then N(y,insert x s ,t)
     else if x>y then N(y,s,insert x t)
          else current_tree
```

Inorder Traversal

- Inorder traversal of a tree, vist left sub tree, then the root then the right sub tree.

```
fun in_order E = []
  | in_order (N(n,s,t)) =
    let
        val left = in_order s
        val right = in_order t
    in
       left@(n::right)
    end
```

Inorder on search trees

```
val test_tree =
 insert 4 (insert 7 (insert 2 (insert 3 (insert 6 E))));

- in_order test_tree;
val it = [2,3,4,6,7] : int list
-
```

Do you think this is a coincidence?

Proof by induction on a tree

Same principle as induction on natural number.

- The empty tree is sorted.
- Givem `N(i,s,t)` and the induction hypothesis that the lists `in_order s` and `in_order t` are sorted we have to show that the list:

  ```
  (in_order s)@(i::in_order t)
  ```

  is sorted, but `n` is bigger than every element in `s` and less than every element in `t` so the list is sorted.

Keeping trees balanced

The problem with binary trees is the get unbalanced and the worst case search time can be $O(n)$ for an n-element binary search tree.

Example suppose we insert the elements `1,2,3,4,5,6,7` in sorted order or in the order `4,3,1,2,6,5,7` we get

$$\begin{tikzcd}
1 \arrow[dr] & & & & & & \\
& 2 \arrow[dr] & & & & & \\
& & 3 \arrow[dr] & & & & \\
& & & 4 \arrow[dr] & & & \\
& & & & 5 \arrow[dr] & & \\
& & & & & 6 \arrow[dr] & \\
& & & & & & 7
\end{tikzcd}$$

$$\begin{tikzcd}
& & & 4 \arrow[dll] \arrow[drr] & & & \\
& 3 \arrow[dl] \arrow[dr] & & & & 6 \arrow[dl] \arrow[dr] & \\
1 & & 2 & & 5 & & 7
\end{tikzcd}$$

What are red-black trees?

- Binary search trees can become unbalanced depending on the order that you insert the elements.
- Red-Black trees are binary search trees where they are balanced when elements are inserted or removed from the tree.
- The balancing is achieved by maintaining two invariants (next slide).
- We then have to be careful that insertion and deletion maintain the invariants.

Red-Black Trees

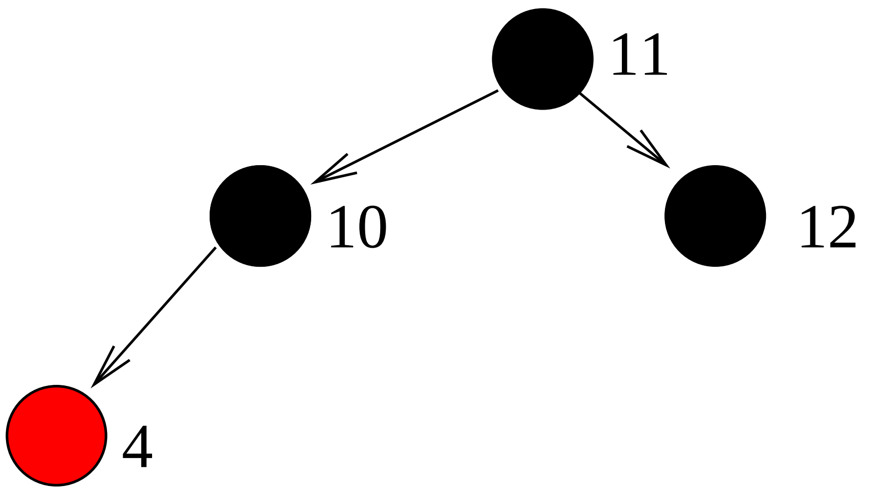
A red-black tree is a binary search tree, where the nodes are coloured red or black. A red-black tree satisfies the following invariants:-

Invariant 1 No red node has a red child

Invariant 2 Every path from the root to an empty node contains the same number of black nodes

Empty nodes are treated as black nodes.

A simple Example of a Red-Black Tree



```
datatype Colour = R | B
datatype Tree = E | T of Colour*int*Tree*Tree
```

What do the invariants give us?

- A red-black tree has depth at most $2\log(n+1)$ where n is the number of nodes.

Proof by induction (possibly later).

But as an intuition the invariants give us that that the longest path (one with alternating black and red nodes) is no more than twice as long as the shortest path.

Question why must the longest path alternate black and red nodes?

Why are red-black trees useful?

- If the depth is a most $2\log(n+1)$ then this puts an upper bound on the search time, since we know that we won't have tall skinny trees only fat wide trees.

Where do we get red-black trees from?

- The problem is that we can't just magic them out of thin air.
- So instead we insert elements into the empty tree and we balance the tree as we go along.

Red-Black Trees :- Inserting

- Treat the tree as an ordinary binary search tree, insert the element in the same place you would in a normal binary search tree but colour it red.
- Colour the top node black
- Balance the tree.

Red-Black Trees :-Inserting

```
fun insert x s =
    let
        fun ins E = T(R,x,E,E)
          | ins(s as T(colour,y,a,b)) =
            if x<y then balance(colour,y,ins a,b)
            else if x>y then balance(colour,y,a,ins b)
                 else s
        val T(_,y,a,b) = ins s
    in T(B,y,a,b)
    end
```

Red-Black Trees :- balancing

Colouring the top node Black satisfies the first invariant, but we have to satisfy the second invariant as well (Every path from the root to an empty node contains the same number of black nodes).

Red-Black Trees :- balancing

```
fun balance (B, z , T(R,y,T(R,x,a,b),c) , d ) =
     T(R,y,T(B,x,a,b),T(B,z,c,d))
  | balance (B, z , T(R,x,a,T(R,y,b,c)) ,  d ) =
     T(R,y,T(B,x,a,b),T(B,z,c,d))
  | balance (B,x,a,T(R,z,T(R,y,b,c),d)) =
     T(R,y,T(B,x,a,b),T(B,z,c,d))
  | balance (B,x,a,T(R,y,b,T(R,z,c,d))) =
     T(R,y,T(B,x,a,b),T(B,z,c,d))
  | balance body = T body
```

When we look for pairs of red nodes, parents and children and reorganise the tree so as to satisfy the second invariant.

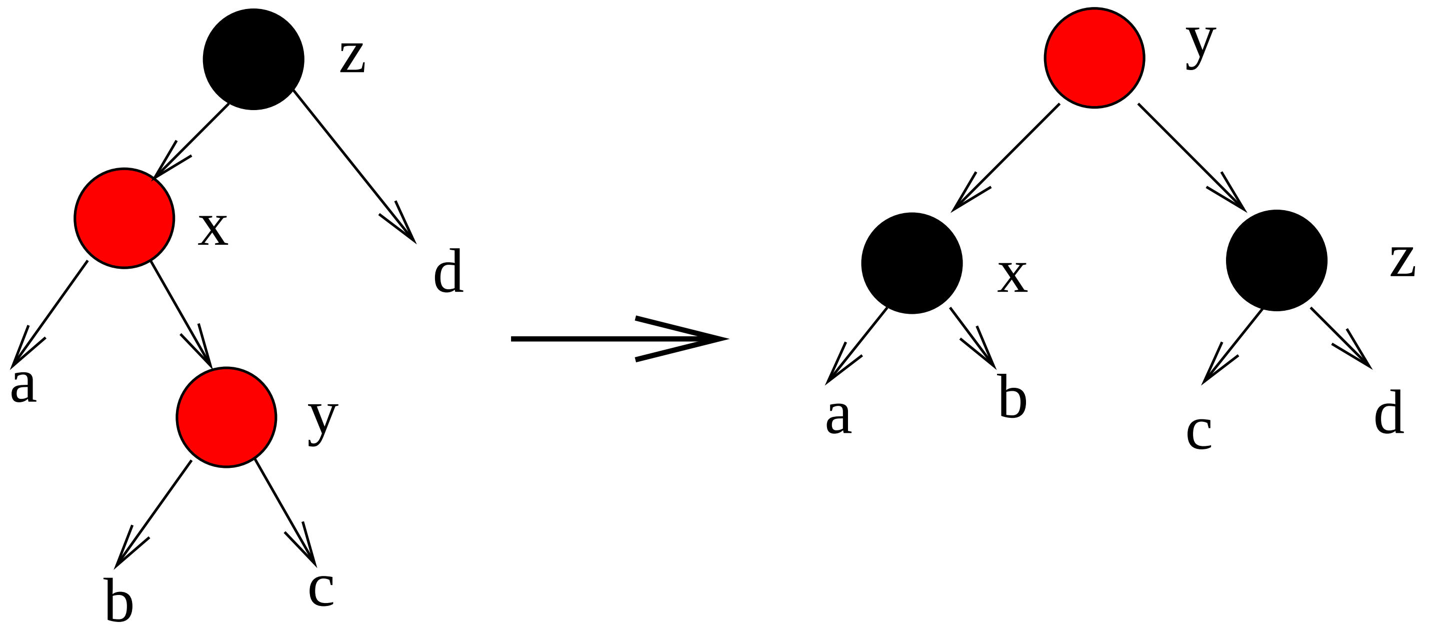
Notice we are looking for red-red violations of the colour invariants.

Understanding balance

To understand the balancing function we have to understand two things

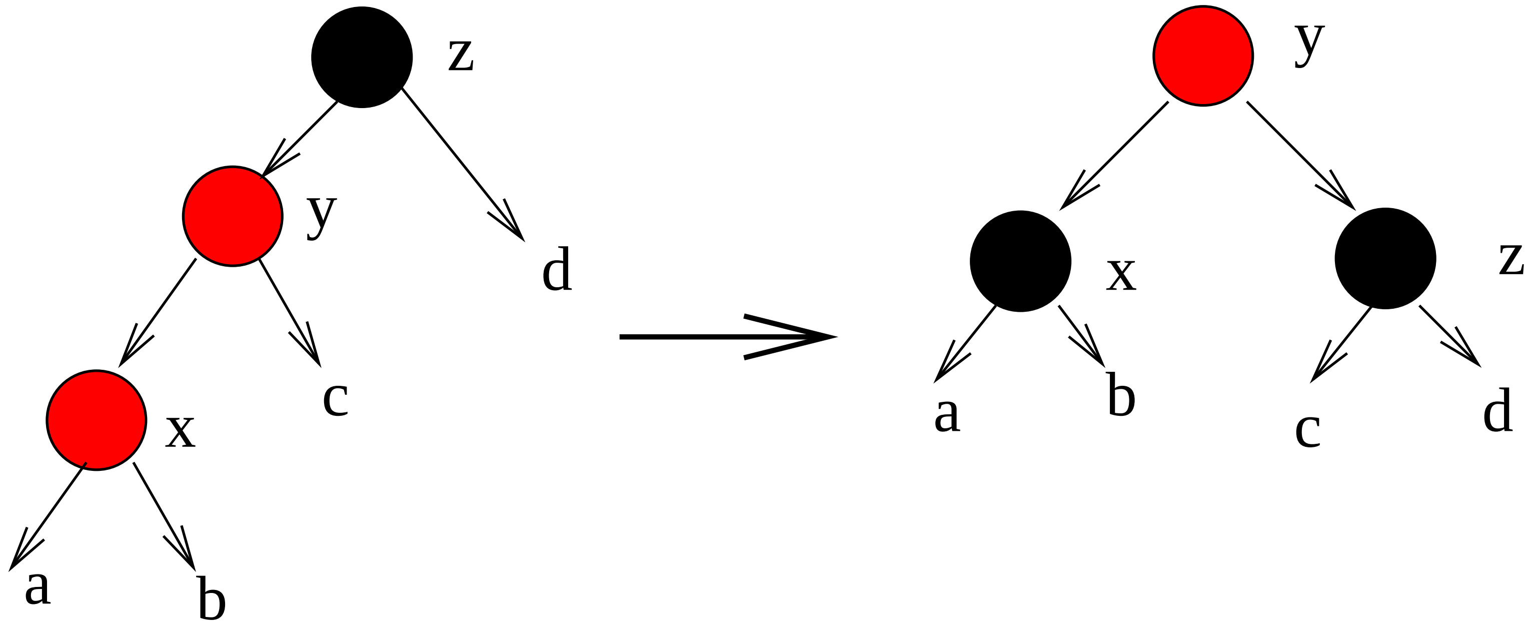
- Why we get a red-black tree which has two parts:
  - Does it satisfy the colour invariants?
  - Is it a binary search tree
- Why it is sufficient just to check for red-red violations.

The first case



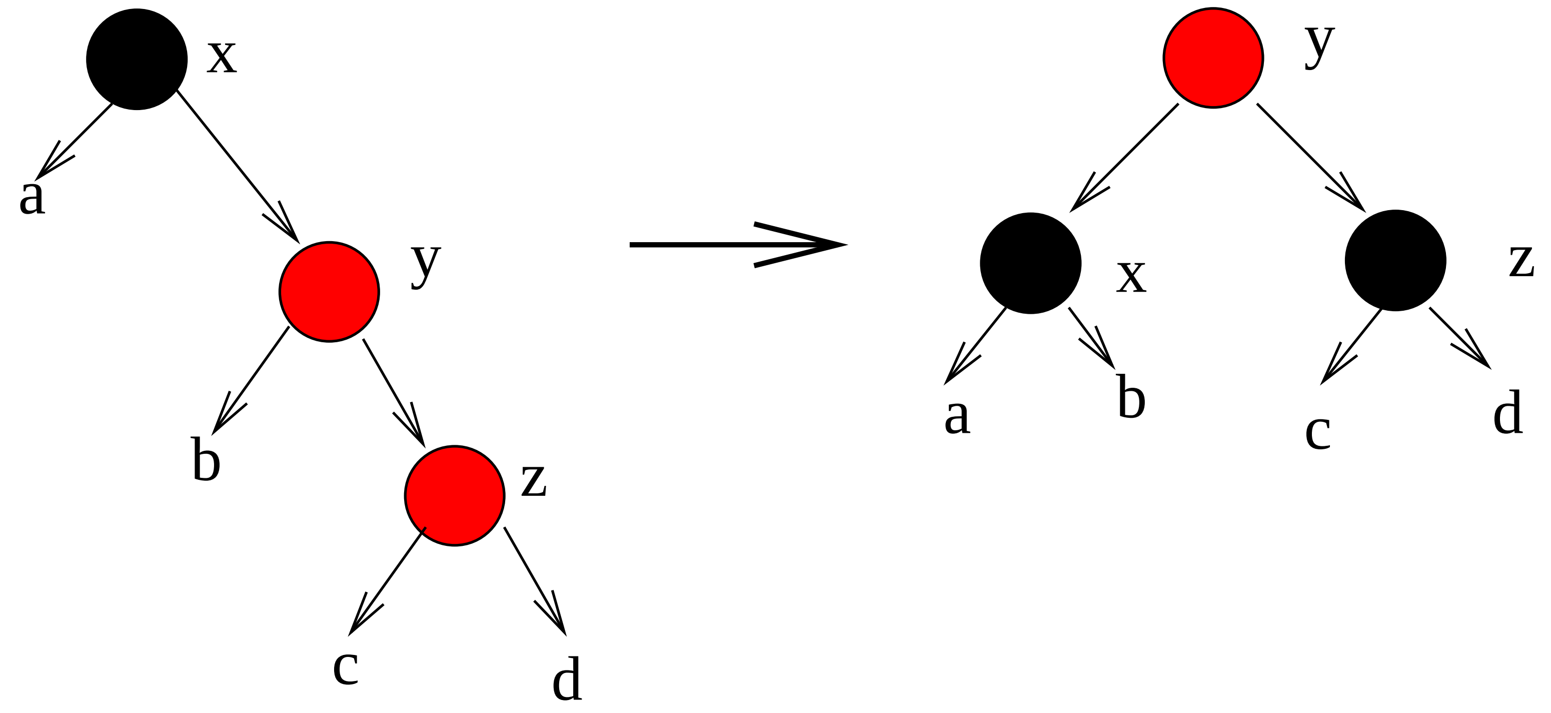
Why is the new tree a red-black tree?

The second case



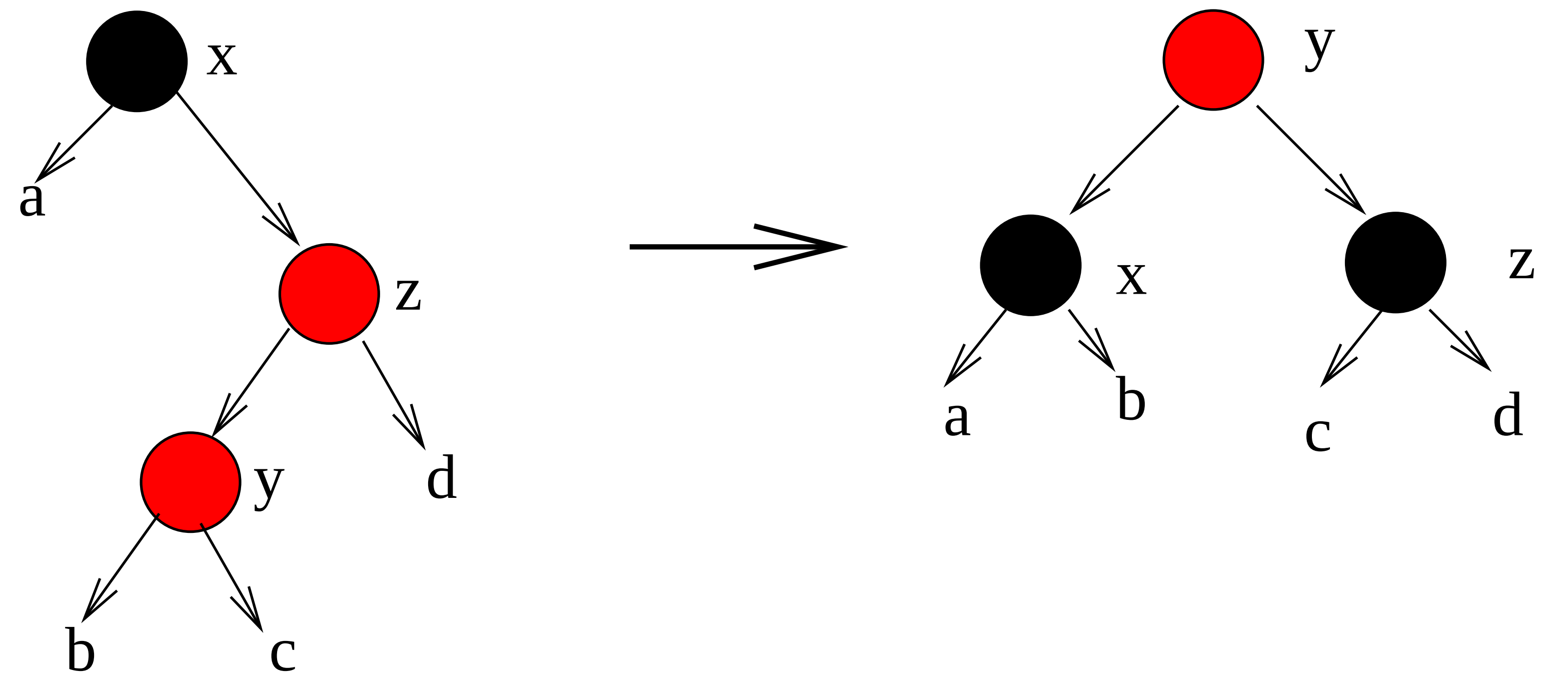
Why is the new tree a red-black tree?

The third case



Why is the new tree a red-black tree?

The last case



Why is the new tree a red-black tree?