How the **symbol map** changes in case of **static** scoping

Outer declaration
int value is shadowed by
inner declaration **string value**

Map becomes bigger as
we enter more scopes,
later becomes smaller again
Imperatively: need to make
maps bigger, later smaller again.
Functionally: immutable maps,
keep old versions.

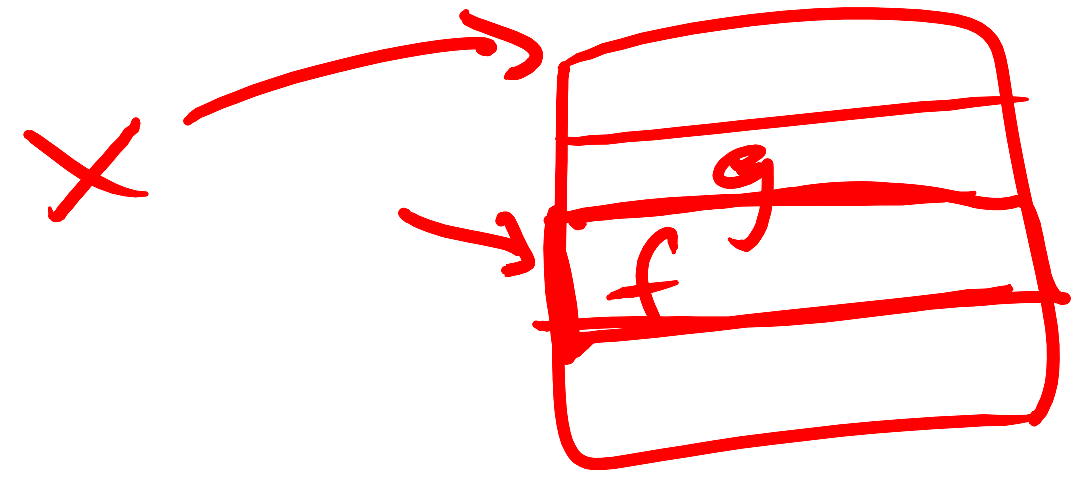
```
class World {
  int sum;  int value;
  // value → int, sum → int
  void add(int foo) {
     // foo → int, value → int, sum → int
     string z;
     // z → string, foo → int, value → int, sum → int
     sum = sum + value; value = 0;
  }
  // value → int, sum → int
  void main(string bar) {
     // bar → string, value → int, sum → int
     int y;
     // y → int, bar → string, value → int, sum → int
     sum = 0;
     value = 10;
     add();
     // y → int, bar → string, value → int, sum → int
     if (sum % 3 == 1) {
       string value;
       // value → string, y → int, bar → string, sum → int
       value = 1;
       add();
       print("inner value = ", value);
       print("sum = ", sum); }
     // y → int, bar → string, value → int, sum → int
     print("outer value = ", value);
} }
```

Representing Data

- In Java, the standard model is a mutable graph of objects
- It seems natural to represent references to symbols using mutable fields (initially null, **resolved** during name analysis)
- Alternative way is functional
  - store the **backbone** of the graph as a algebraic data type (immutable)
  - pass around a map linking from identifiers to their declarations
- Note that a field **class** A { **var** f:T } is like   f: Map[A,T]

Symbol Table (Γ) Contents

- Map identifiers to the symbol with relevant information about the identifier
- All information is derived from syntax tree - symbol table is for efficiency
  - in old one-pass compilers there was only symbol table, no syntax tree
  - in modern compiler: we could always go through entire tree, but symbol table can give faster and easier access to the part of syntax tree, or some additional information
- Goal: efficiently supporting phases of compiler



- In the name analysis phase:
  - finding which identifier refers to which definition
  - we store *definitions*
- What kinds of things can we define? What do we need to know for each ID?

variables (globals, fields, parameters, locals):

- need to know types, positions - for error messages
- later: memory layout. To compile x.f = y into memcopy(addr_y, addr_x+6, 4)
  - e.g. 3rd field in an object should be stored at offset e.g. +6 from the address of the object
  - the size of data stored in x.f is 4 bytes
- sometimes more information explicit: whether variable local or global

methods, functions, classes: recursively have with their own symbol tables

Functional: Different Points, Different Γ

```
class World {
  int sum;
  void add(int foo) {
     sum = sum + foo;
  }
  void sub(int bar) {
     sum = sum - bar;
  }
  int count;
}
```

$\Gamma_0 = \{(sum, int), (count, int)\}$

$\Gamma_1 = \Gamma_0 [foo := int]$

Γ_0

$\Gamma_1 = \Gamma_0 [bar := int]$

Imperative Way: Push and Pop

```
class World {
  int sum;
  void add(int foo) {
    sum = sum + foo;
  }
  void sub(int bar) {
    sum = sum - bar;
  }
  int count;
}
```

$\Gamma_0 = \{(sum, int), (count, int)\}$

$\Gamma_1 = \Gamma_0\,[foo := int]$

change table, record change

Γ_0

revert changes from table

$\Gamma_1 = \Gamma_0\,[bar := int]$

change table, record change

revert changes from table

Imperative Symbol Table

- Hash table, mutable Map[ID,Symbol]
- Example:
  - hash function into array
  - array has linked list storing (ID,Symbol) pairs
- Undo stack: to enable entering and leaving scope
- Entering new scope (function,block):
  - add beginning-of-scope marker to undo stack
- Adding nested declaration (ID,sym)
  - lookup old value symOld, push old value to undo stack
  - insert (ID,sym) into table
- Leaving the scope
  - go through undo stack until the marker, restore old values

Functional: Keep Old Version

```
class World {
  int sum;
  void add(int foo) {
     sum = sum + foo;
  }
  void sub(int bar) {
     sum = sum - bar;
  }
  int count;
}
```

$\Gamma_0 = \{(sum, int), (count, int)\}$

$\Gamma_1 = \Gamma_0 [foo := int]$

create new Γ_1, keep old Γ_0

Γ_0

$\Gamma_2 = \Gamma_0 [bar := int]$

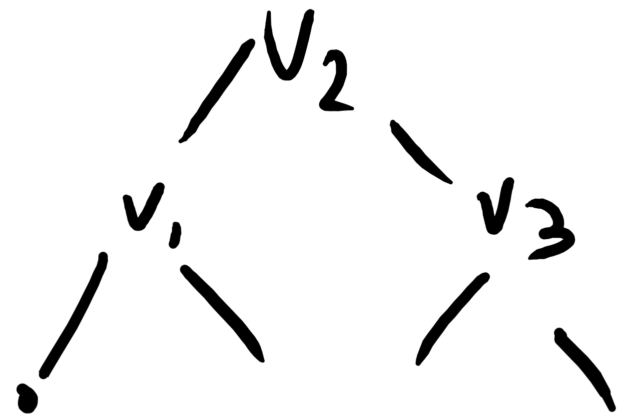
create new Γ_2, keep old Γ_0

Functional Symbol Table Implemented

- Typical: Immutable Balanced Search Trees

```
sealed abstract class BST
case class Empty() extends BST
case class Node(left: BST, value: Int, right: BST) extends BST
```

Simplified. In practice, BST[A], store Int key and value A

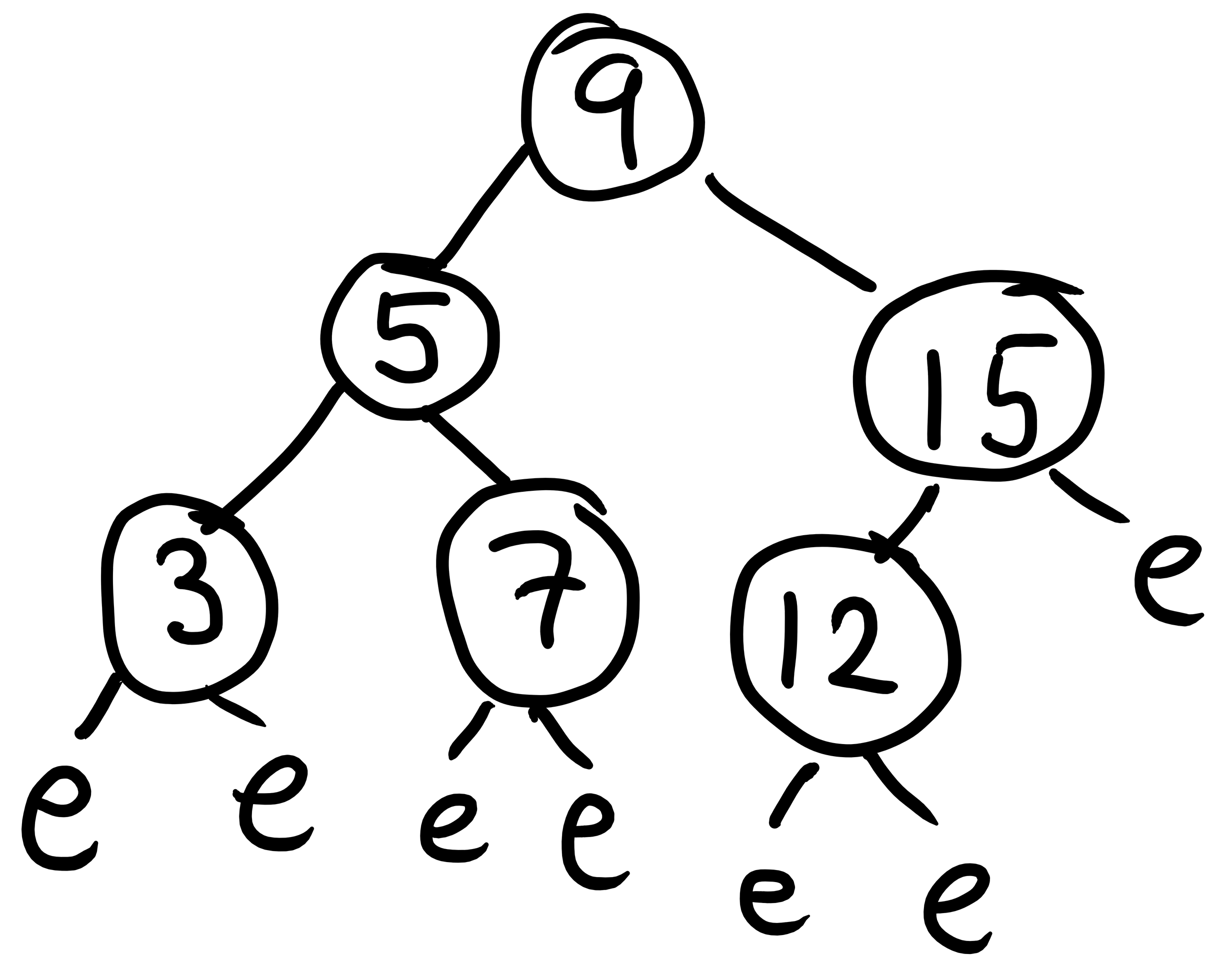


- Updating returns new map, keeping old one
  - lookup and update both *log(n)*
  - update creates new path (copy *log(n)* nodes, share rest!)
  - memory usage acceptable

Lookup

```
def contains(key: Int, t : BST): Boolean = t match {
  case Empty() => false
  case Node(left,v,right) => {
    if (key == v) true
    else if (key < v) contains(key, left)
    else contains(key, right)
  }
}
```

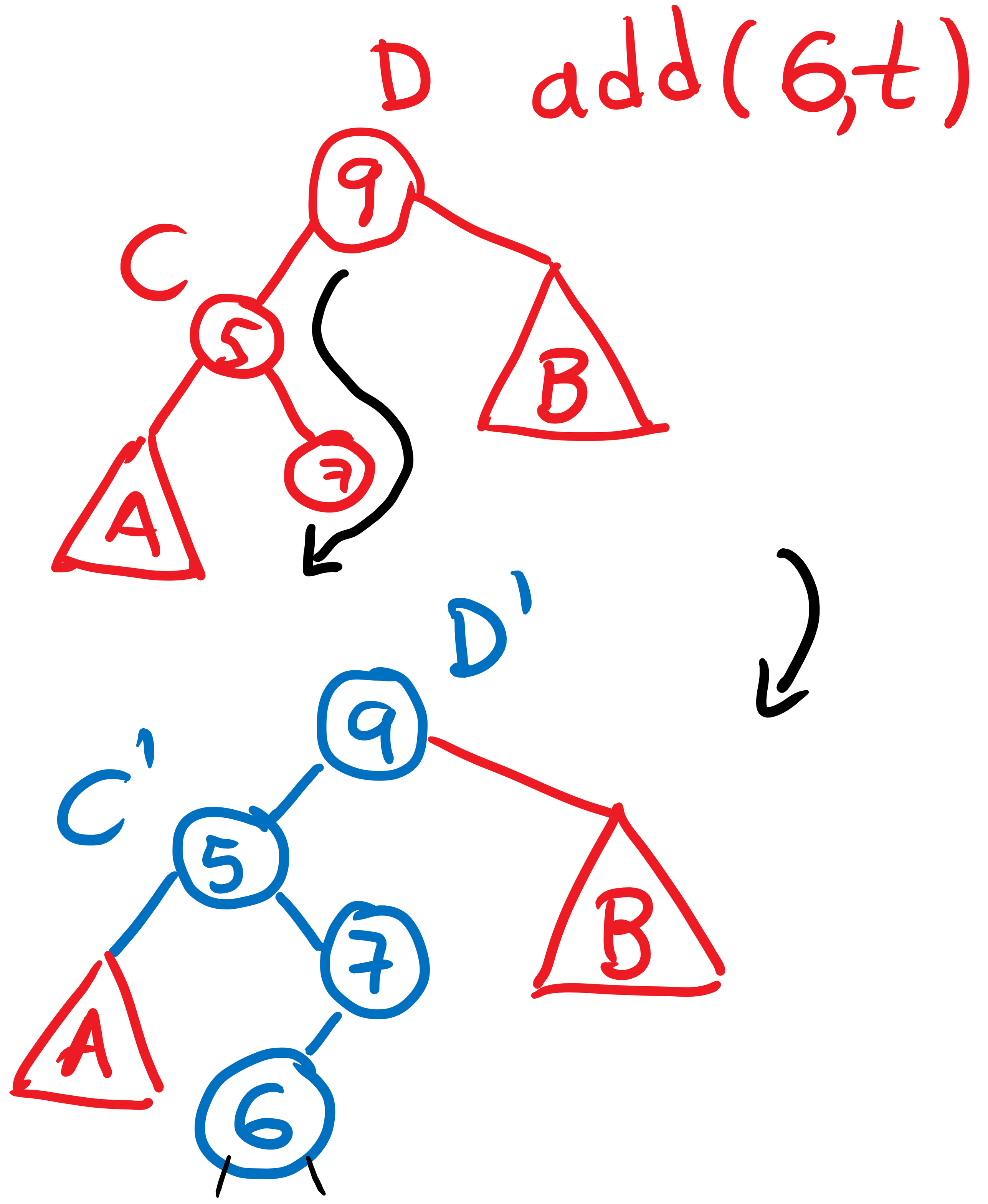
Running time bounded by tree height.



contains(6,t) ?

Insertion

```
def add(x : Int, t : BST) : Node = t match {
   case Empty() => Node(Empty(),x,Empty())
   case t @ Node(left,v,right) => {
     if (x < v) Node(add(x, left), v, right)
     else if (x==v) t
     else Node(left, v, add(x, right))
   }
}
```

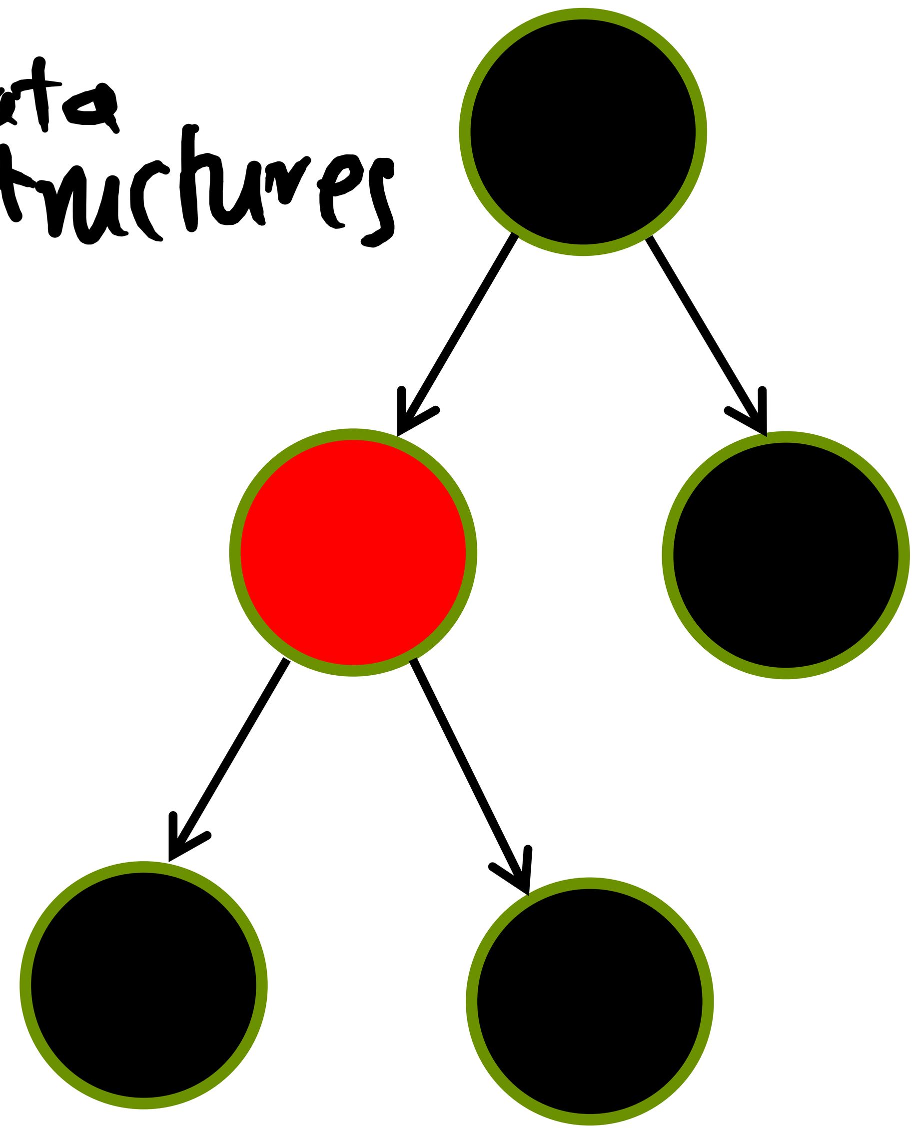


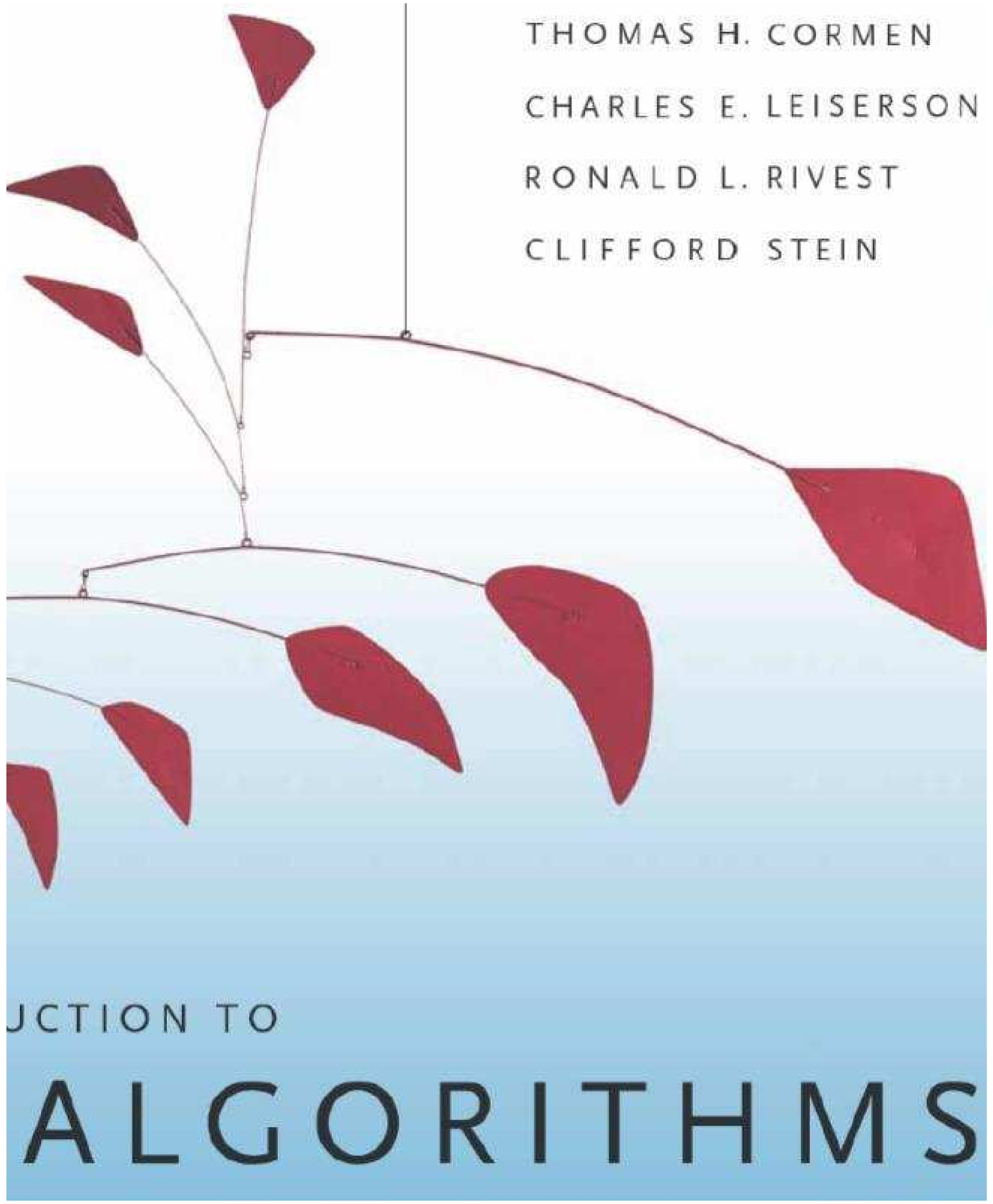
Both add(x,t) and t remain accessible.

Running time and newly allocated nodes bounded by tree height.

Chris Okasaki : Purely Functional Data Structures

Balanced Trees: Red-Black Trees





Balanced Tree: Red Black Tree

Goals:

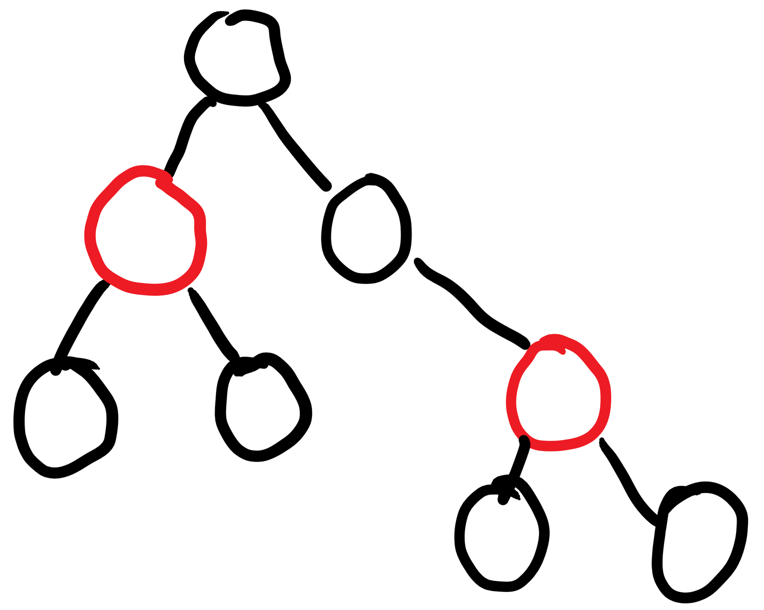
- ensure that tree height remains at most log(size)

  add(1,add(2,add(3,...add(n,Empty())...))) ~ linked list

- preserve efficiency of individual operations:

  rebalancing arbitrary tree: could cost O(n) work

Solution: maintain mostly balanced trees: height still O(log size)



```
sealed abstract class Color
case class Red() extends Color
case class Black() extends Color

sealed abstract class Tree
case class Empty() extends Tree
case class Node(c: Color,left: Tree,value: Int, right: Tree)
                    extends Tree
```

Properties of red-black trees

A ***red-black tree*** is a binary search tree with one extra bit of storage per node: its *color*, which can be either RED or BLACK. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately ***balanced***.

Each node of the tree now contains the attributes *color*, *key*, *left*, *right*, and *p*. If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL. We shall regard these NILs as being pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.

A red-black tree is a binary tree that satisfies the following ***red-black properties***:

balanced tree constraints

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

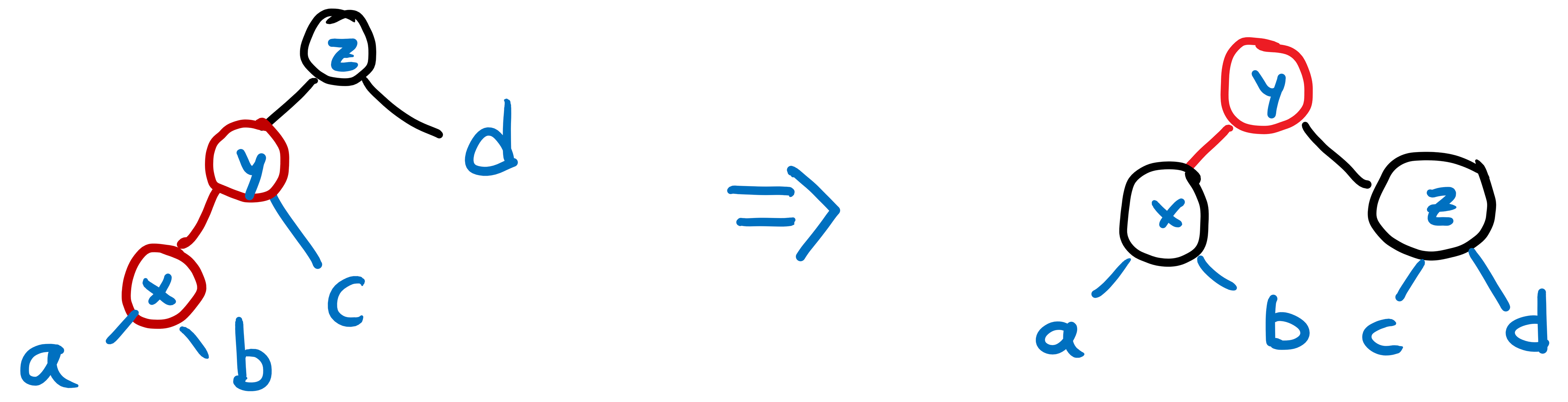
From 4. and 5.: tree height is O(log size).

Analysis is similar for mutable and immutable trees.

for immutable trees: see book by Chris Okasaki

Balancing

```
def balance(c: Color, a: Tree, x: Int, b: Tree): Tree = (c,a,x,b) match {
  case (Black(),Node(Red(),Node(Red(),a,xV,b),yV,c),zV,d) =>
  Node(Red(),Node(Black(),a,xV,b),yV,Node(Black(),c,zV,d))
```




```
  case (Black(),Node(Red(),a,xV,Node(Red(),b,yV,c)),zV,d) =>
  Node(Red(),Node(Black(),a,xV,b),yV,Node(Black(),c,zV,d))
  case (Black(),a,xV,Node(Red(),Node(Red(),b,yV,c),zV,d)) =>
  Node(Red(),Node(Black(),a,xV,b),yV,Node(Black(),c,zV,d))
  case (Black(),a,xV,Node(Red(),b,yV,Node(Red(),c,zV,d))) =>
  Node(Red(),Node(Black(),a,xV,b),yV,Node(Black(),c,zV,d))
  case (c,a,xV,b) => Node(c,a,xV,b)
}
```

Insertion

```
def add(x: Int, t: Tree): Tree = {
 def ins(t: Tree): Tree = t match {
  case Empty() => Node(Red(),Empty(),x,Empty())
  case Node(c,a,y,b) =>
   if (x < y) balance(c, ins(a), y, b)
   else if (x == y) Node(c,a,y,b)
   else balance(c,a,y,ins(b))
 }
 makeBlack(ins(t))
}
def makeBlack(n: Tree): Tree = n match {
  case Node(Red(),l,v,r) => Node(Black(),l,v,r)
  case _ => n
}
```

Modern object-oriented languages (e.g. Scala) support abstraction and functional data structures. Just use Map from Scala.

Exercise

Determine the output of the following program assuming static and dynamic scoping. Explain the difference, if there is any.

```
object MyClass {
  val x = 5
  def foo(z: Int): Int = { x + z }
  def bar(y: Int): Int = {
     val x = 1; val z = 2
     foo(y)
  }
  def main() {
    val x = 7
    println(foo(bar(3)))
  }
}
```