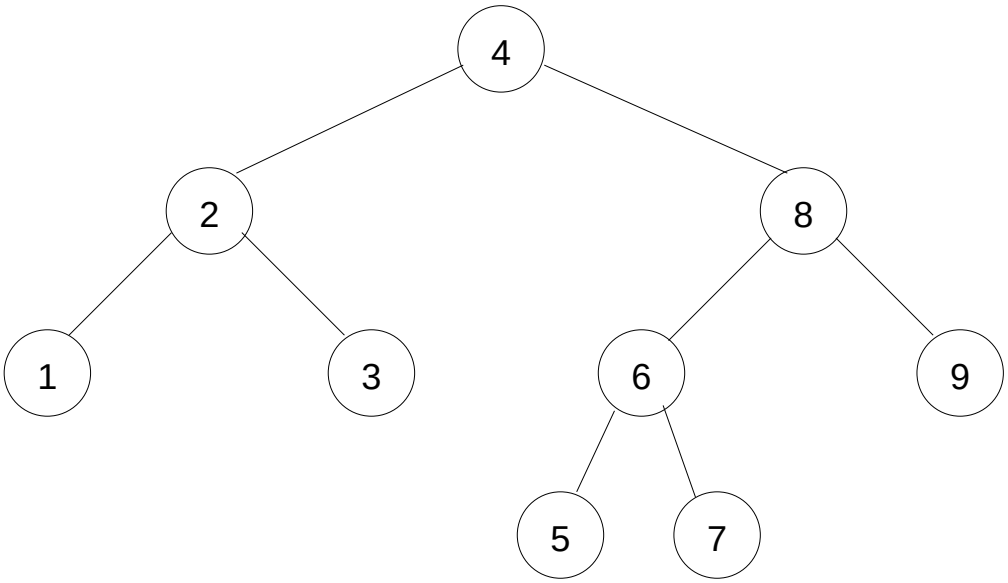
Binary Trees

(Version of April 26, 2010)
(Based on original slides by Yves Deville)

Concepts

Binary trees of objects of type α: α bTree



Terminology:

- *Root* node, *leaf* node (plural: *leaves*), *internal* node
- Left and right *subtree*
- *Parent* node, left and right *child* node, *sibling* node

Graphical Representation Convention:
Empty trees are not drawn (but they consume memory!)

Operations

value emptyBtree
TYPE: α bTree
VALUE: the empty binary tree

function isEmptyBtree T
TYPE: α bTree $\rightarrow$ bool
PRE: (none)
POST: **true** if T is empty
false otherwise

function consBtree v L R
TYPE: $\alpha \rightarrow \alpha$ bTree $\rightarrow \alpha$ bTree $\rightarrow \alpha$ bTree
PRE: (none)
POST: the binary tree with root v, left sub-tree L, and right sub-tree R

function left T
TYPE: α bTree $\rightarrow \alpha$ bTree
PRE: T is non-empty
POST: the left sub-tree of T

function right T
TYPE: α bTree $\rightarrow \alpha$ bTree
PRE: T is non-empty
POST: the right sub-tree of T

function root T
TYPE: α bTree $\rightarrow \alpha$
PRE: T is non-empty
POST: the root of T

Realisation

Representation

```
datatype 'a bTree = Void
                  | Bt of 'a * 'a bTree * 'a bTree
REPRESENTATION CONVENTION: a binary tree with root x,
left subtree L, and right subtree R is represented by Bt(x,L,R)
EXAMPLE: Bt(4, Bt(2, Bt(1,Void,Void), Bt(3,Void,Void)),
               Bt(8, Bt(6, Bt(5,Void,Void), Bt(7,Void,Void)), Bt(9,Void,Void)))
```

Operations

```
abstype 'a bTree = Void
                 | Bt of 'a * 'a bTree * 'a bTree
with
    val emptyBtree = Void
    fun isEmptyBtree Void = true
      | isEmptyBtre (Bt(v,L,R)) = false
    fun consBtree v L R = Bt(v,L,R)
    fun left Void = error "left: empty bTree"
      | left (Bt(v,L,R)) = L
    fun right Void = error "right: empty bTree"
      | right (Bt(v,L,R)) = R
    fun root Void = error "root: empty bTree"
      | root (Bt(v,L,R)) = v
end
```

Walk Operations

function inorder T
TYPE: α bTree $\rightarrow$ α list
PRE: (none)
POST: the nodes of T upon an inorder walk

```
fun  inorder  Void  =  [ ]
  |  inorder  (Bt(v,L,R))  =  (inorder  L)  @  (v  ::  inorder  R)
```

No tail recursion!
It takes $\Theta(n \log n)$ time for a binary tree of n nodes...

function inorderGen T acc
TYPE: α bTree $\rightarrow$ α list $\rightarrow$ α list
PRE: (none)
POST: (the nodes of T upon an inorder walk) @ acc

```
fun  inorderGen  Void  acc  =  acc
  |  inorderGen  (Bt(v,L,R))  acc  =
       let  val  rAcc  =  inorderGen  R  acc
       in  inorderGen  L  (v::rAcc)  end

fun  inorder  t  =  inorderGen  t  [ ]
```

One tail recursion! No call to @ (concatenation)!
It takes $\Theta(n)$ time for a binary tree of n nodes

Exercises

- Efficiently realise the preorder and postorder walks of a binary tree, and analyse the underlying algorithms
- How to test the equality of two binary trees?

Other Operations

function exists k T
TYPE: $\alpha^{=} \rightarrow \alpha^{=}$ bTree $\rightarrow$ bool
PRE: (none)
POST: **true** if T contains node k
false otherwise
function insert k T
TYPE: $\alpha^{=} \rightarrow \alpha^{=}$ bTree $\rightarrow \alpha^{=}$ bTree
PRE: (none)
POST: T with node k
function delete k T
TYPE: $\alpha^{=} \rightarrow \alpha^{=}$ bTree $\rightarrow \alpha^{=}$ bTree
PRE: (none)
POST: if exists k T, then T without one occurrence of node k, otherwise T
function nbNodes T
TYPE: α bTree $\rightarrow$ int
PRE: (none)
POST: the number of nodes of T
function nbLeaves T
TYPE: α bTree $\rightarrow$ int
PRE: (none)
POST: the number of leaves of T

Exercises

- Efficiently realise these five functions
- Show that their algorithms at worst take $\Theta(n)$ time, if not $\Theta(1)$ time, on a binary tree with initially n nodes

Height of a Binary Tree

- The *height of a node* is the length of the longest path (measured in its number of nodes) from that node to a leaf
- The *height of a tree* is the height of its root

```
function height T
TYPE: α bTree → int
PRE:  (none) ; POST: the height of T

fun height Void = 0
  | height (Bt(v,L,R)) = 1 + Int.max (height L, height R)
```

No tail recursion!
It takes $\Theta(n)$ time for a binary tree of n nodes.
Note that `heightGen T acc = acc + height T` does *not* suffice to get a tail recursion: why?!

```
function heightGen T acc hMax
TYPE: α bTree → int → int → int
PRE: (none) ; POST: max (acc + height T, hMax)

fun heightGen Void acc hMax = Int.max (acc, hMax)
  | heightGen (Bt(v,L,R)) acc hMax =
      let val h1 = heightGen R (acc+1) hMax
      in heightGen L (acc+1) h1 end

fun height2 bt = heightGen bt 0 0
```

One tail recursion!
It also takes $\Theta(n)$ *time* for a binary tree of n nodes, but it takes less *space*!

Binary Search Trees

Concepts and terminology (see the tree on page 1)

Binary search trees of nodes of type $(\alpha^{=}, \beta)$: $(\alpha^{=}, \beta)$ bsTree where:

- $\alpha^{=}$ is the type of the *keys* (need for an equality test)
- β is the type of the *satellite data* for each key

are binary trees with:

REPRESENTATION INVARIANT: for a binary search tree
with (k,s) in the root, left subtree L, and right subtree R:
- every element of L has a key smaller than k
- every element of R has a key larger than k

and recursively so on, for L and R

Note that we (arbitrarily) ruled out duplicate keys!

Benefits

- The inorder walk of a binary search tree lists its nodes by increasing order of their keys!
- The basic operations at worst take $\Theta(n)$ time on a binary search tree with (initially) n nodes, but they take $O(\lg n)$ time on *randomly* built binary search trees

Let us restrict our realisation to *integer* keys: β bsTree

Some Operations

value emptyBsTree
TYPE: β bsTree
VALUE: the empty binary search tree

function isEmptyBsTree T
TYPE: β bsTree $\rightarrow$ bool
PRE: (none)
POST: **true** if T is empty
false otherwise

function exists k T
TYPE: int $\rightarrow$ β bsTree $\rightarrow$ bool
PRE: (none)
POST: **true** if T contains a node with key k
false otherwise

function insert (k,s) T
TYPE: (int $*\beta$) $\rightarrow$ β bsTree $\rightarrow$ β bsTree
PRE: (none)
POST: if exists k T, then T with s as satellite data for key k
otherwise T with node (k,s)

function retrieve k T
TYPE: int $\rightarrow$ β bsTree $\rightarrow$ β
PRE: exists k T
POST: the satellite data associated to key k in T

function delete k T
TYPE: int $\rightarrow$ β bsTree $\rightarrow$ β bsTree
PRE: (none)
POST: if exists k T, then T without the node with key k, otherwise T

Realisation

Representation

```
datatype 'b bsTree = Void
                   | Bst of (int * 'b) * 'b bsTree * 'b bsTree
```

REPRESENTATION CONVENTION: a BST with (k,s) in the root, left subtree L, and right subtree R is represented by Bst((k,s),L,R)
REPRESENTATION INVARIANT: (see page 7)

Operations

```
val emptyBsTree = Void

fun isEmptyBsTree Void = true
  | isEmptyBsTree (Bst((key,sat),L,R)) = false

fun exists k Void = false
  | exists k (Bst((key,sat),L,R)) =
      if k = key then true
      else if k < key then exists k L
      else (* k > key *) exists k R

fun insert (k,s) Void = Bst((k,s),Void,Void)
  | insert (k,s) (Bst((key,sat),L,R)) =
      if k = key then Bst((k,s),L,R)
      else if k < key then Bst((key,sat), (insert (k,s) L), R)
      else (* k > key *) Bst((key,sat), L, (insert (k,s) R))

fun retrieve k Void = error "retrieve: non-existing node"
  | retrieve k (Bst((key,sat),L,R)) =
      if k = key then sat
      else if k < key then retrieve k L
      else (* k > key *) retrieve k R
```

When deleting a node (key,sat) whose subtrees L and R are *both* non-empty, we must not violate the repr. invariant!

1. Replace (key,sat) by the node with the *maximal* key of L, whose key is smaller than the key of *any* node of R (one could also replace by the node with the *minimal* key of R)
2. Remove this node with the maximal key from L

So we need a deleteMax function:

function deleteMax T
TYPE: β bsTree $\rightarrow$ (int $*$ β) $*$ β bsTree
PRE: T is non-empty
POST: (max, NT), where max is the node of T with the maximal key, and NT is T without max

```
fun deleteMax Void = error "deleteMax: empty bsTree"
  | deleteMax (Bst(r,L,Void): 'b bsTree) = (r, L)
  | deleteMax (Bst(r,L,R)) =
      let val (max, newR) = deleteMax R
      in (max, Bst(r,L,newR)) end

fun delete k Void = Void
  | delete k (Bst((key,sat),L,R)) =
      if k < key then Bst((key,sat), (delete k L), R)
      else if k > key then Bst((key,sat), L, (delete k R))
      else (* k = key *)
          case (L,R) of
              (Void, _ ) => R
            | ( _ ,Void) => L
            | ( _ , _ ) => let val (max, newL) = deleteMax L
                           in Bst(max,newL,R) end
```