Search Trees

Search Trees

Contents

1 Tutorial Agenda & Aims

1. Welcome and introductions
2. Material on Binary Trees and Searching (Unit 4)
3. Implementation in Python and (optional) Haskell
4. Learning themes:
   - Evaluation of expressions
   - Synthesising an algorithm from an initial idea
5. To cover some of
   - Binary Trees
   - Binary Search Trees
   - Height Balanced (AVL) Trees
6. Questions & discussion (at any point)
7. *Adobe Connect* — if you or I get cut off, wait till we reconnect (or send you an email)

Introductions — Me

- *Name* Phil Molyneux
- *Background* Physics & Maths, Operational Research, Computer Science
- *First programming languages* Fortran, BASIC, Pascal
- *Favourite Software*
  - Haskell — pure functional programming language
  - Text editors TextMate, Sublime Text — previously Emacs
  - Word processing in LaTeX — all these slides and notes

  - Mac OS X

- *Learning style* — I read the manual before using the software

Introductions — You

- *Name* ?
- *New topics last month* ?
- *Binary Tree or AVL Tree topics you want covered* ?
- *Other OU courses* ?
- *Learning style* ?
- *Anything else* ?

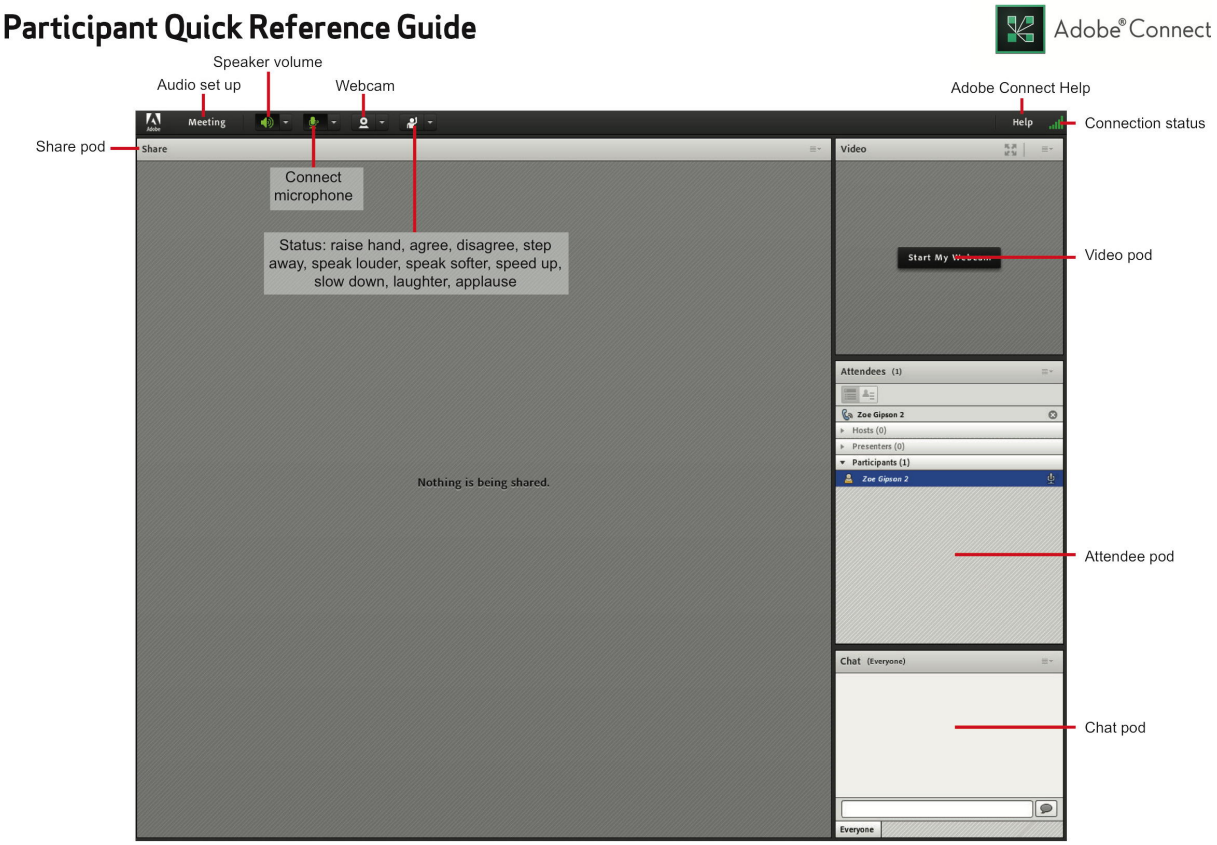
1.1 M269 Binary Trees — Resources

1. *OU Live* — if you or I get cut off, wait till we reconnect (or send you an email)
2. Source of files: pmolyneux.co.uk/OU/M269FolderSync/M269TutorialNotes/M269TutorialBinaryTrees
3. Slides PDF M269TutorialBinaryTrees2019J.beamer.pdf
4. Notes version of slides PDF M269TutorialBinaryTrees2019J.article.pdf
5. Python file, parts included in 3 m269TutorialBinaryTrees2019J.py
6. Python test harness for 5 m269TutorialBinaryTrees2019JTest.py
7. Haskell test harness, imports 8 M269TutorialBinaryTrees2019JTest.hs
8. Haskell literate script, main source for 3 and 4 M269TutorialBinaryTrees2019J.lhs — only look at this if you have some LaTeX knowledge
9. LaTeX slides harness M269TutorialBinaryTrees2019J.beamer.tex
10. LaTeX notes harness M269TutorialBinaryTrees2019J.article.tex

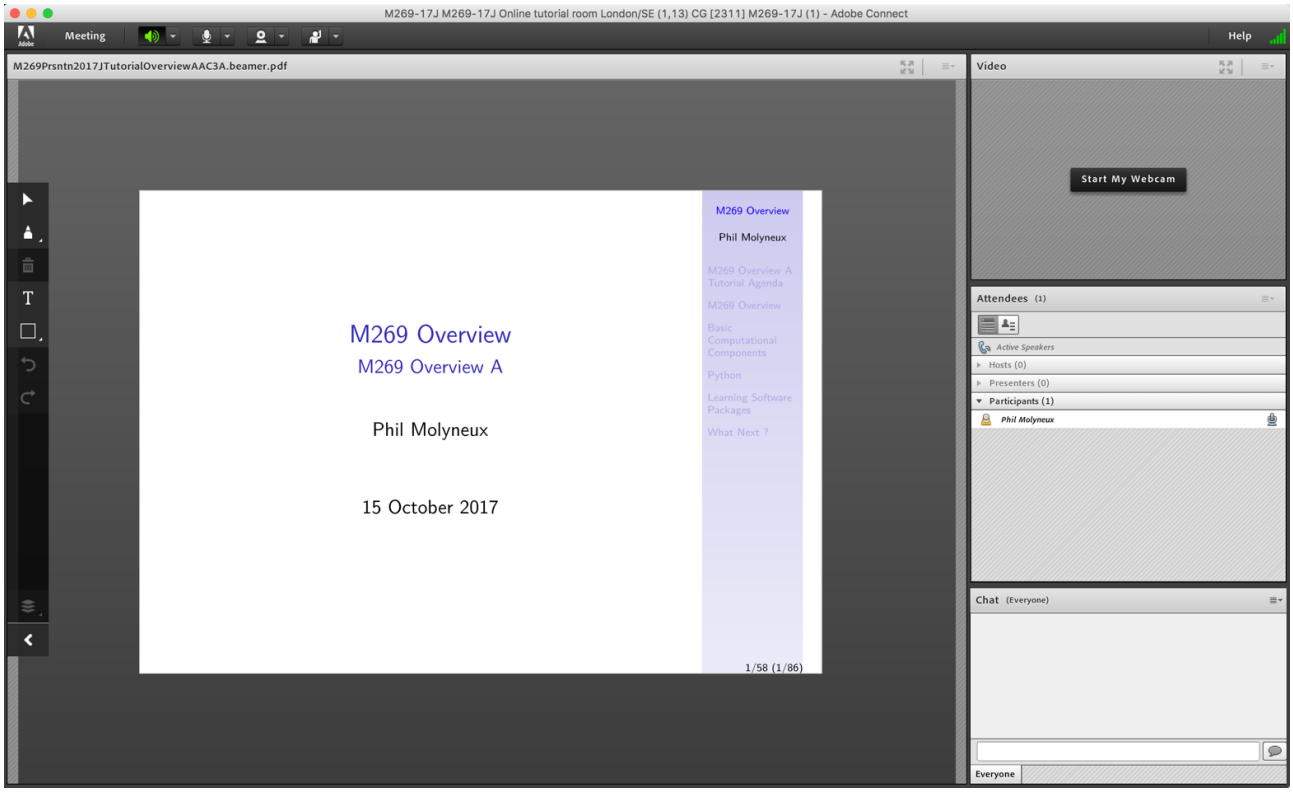
2 Adobe Connect Interface and Settings

2.1 Adobe Connect Interface — Student View

Adobe Connect Interface — Student Quick Reference



Adobe Connect Interface — Student View



2.2 Adobe Connect Settings

Adobe Connect Settings

- **Everybody: Audio Settings** Meeting › Audio Setup Wizard. . .
- **Audio** Menu bar › Audio › Microphone rights for Participants ✔
- Do not *Enable single speaker mode*
- **Drawing Tools** Share pod menu bar › Draw (1 slide/screen)
- Share pod menu bar › Menu icon › Enable Participants to draw ✔ gray
- Meeting › Preferences › Whiteboard › Enable Participants to draw ✔
- Cancel hand tool . . . Do not enable green pointer. . .
- Meeting › Preferences › Attendees Pod ✘ *Raise Hand notification*
- Meeting › Preferences › Display Name Display First & Last Name
- **Cursor** Meeting › Preferences › General tab › Host Cursors › Show to all attendees ✔ (default *Off*)
- Meeting › Preferences › Screen Share › Cursor › Show Application Cursor
- **Webcam** Menu bar › Webcam › Enable Webcam for Participants ✔
- **Recording** Meeting › Record Meeting. . . ✔

Adobe Connect — Access

- **Tutor Access**

  TutorHome › M269 Website › Tutorials

  Cluster Tutorials › M269 Online tutorial room

  Tutor Groups › M269 Online tutor group room

  Module-wide Tutorials › M269 Online module-wide room

- **Attendance**

  TutorHome › Students › View your tutorial timetables

- **Beamer Slide Scaling** 440% (422 x 563 mm)
- **Clear Everyone's Status**

  Attendee Pod › Menu › Clear Everyone's Status

- **Grant Access** and send link via email

  Meeting › Manage Access & Entry › Invite Participants. . .

- **Presenter Only Area**

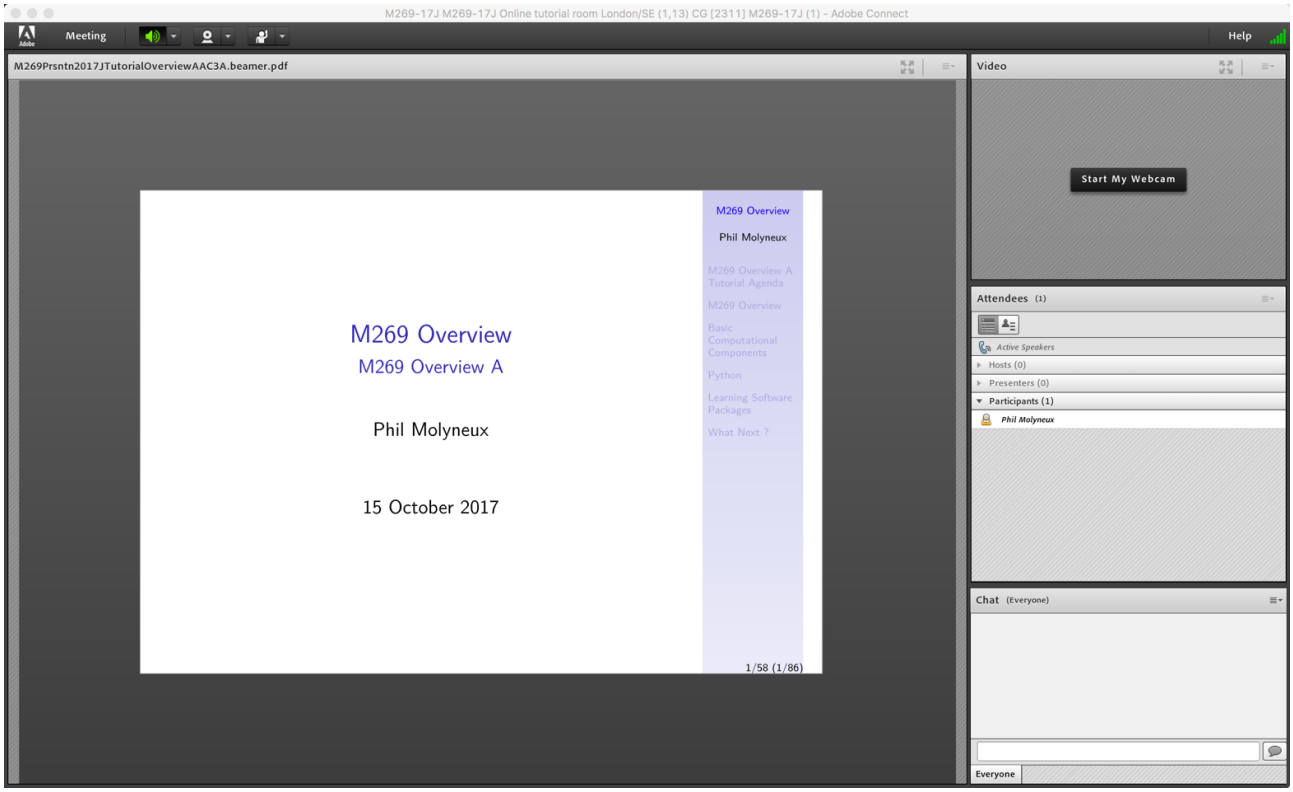
  Meeting › Enable/Disable Presenter Only Area

Adobe Connect — Keystroke Shortcuts

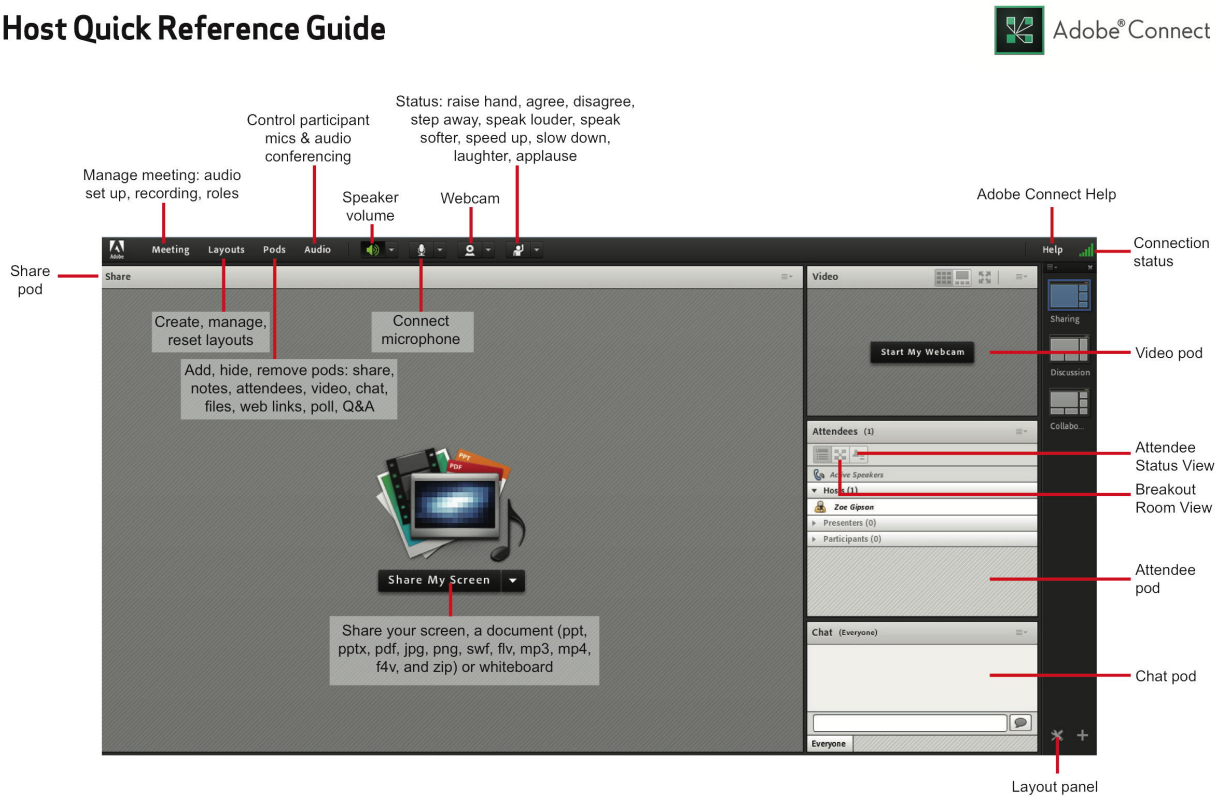
- Keyboard shortcuts in Adobe Connect
- **Toggle Mic** ⌘ + M (Mac), Ctrl + M (Win) (On/Disconnect)
- **Toggle Raise-Hand status** ⌘ + E
- **Close dialog box** ⎋ (Mac), Esc (Win)
- **End meeting** ⌘ + \

2.3 Adobe Connect Interface — Student & Tutor Views

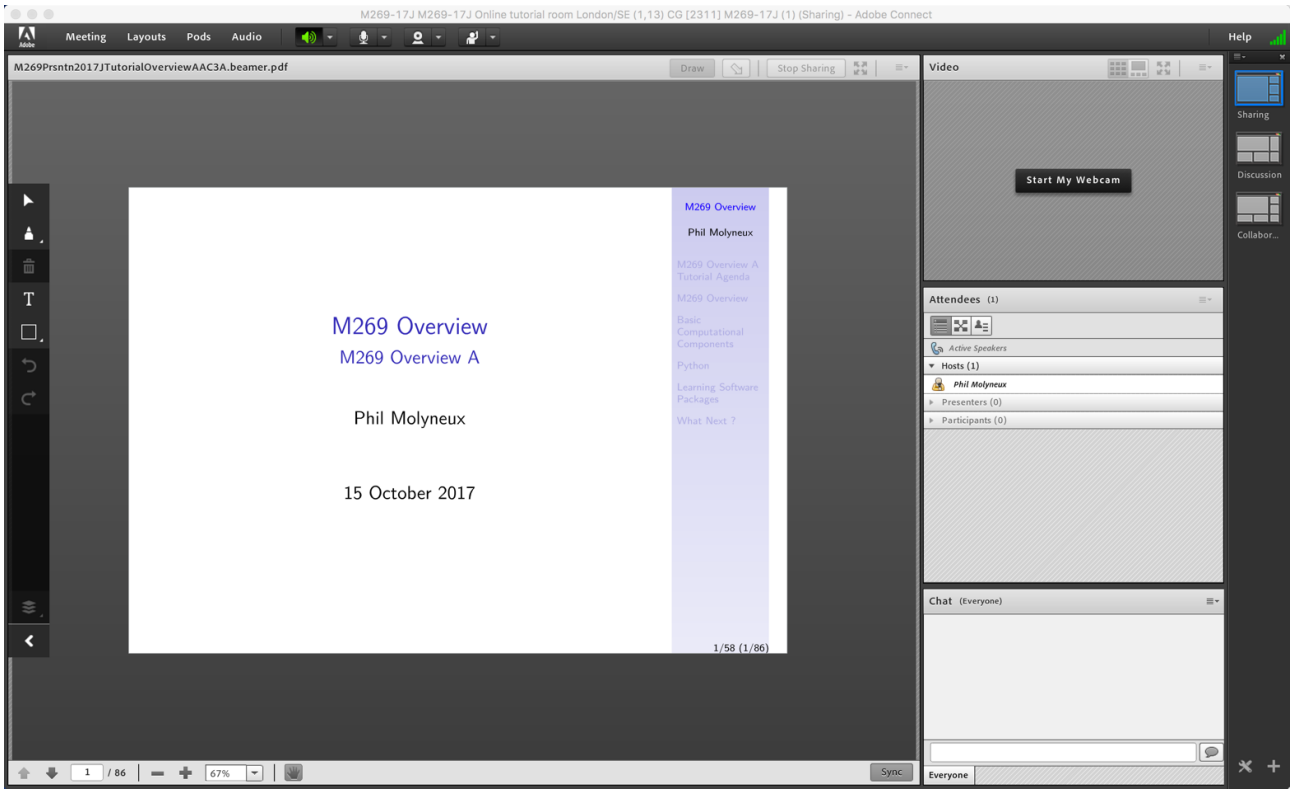
Adobe Connect Interface — Student View (default)



Adobe Connect Interface — Tutor Quick Reference



Adobe Connect Interface — Tutor View



2.4 Adobe Connect — Sharing Screen & Applications

- Share My Screen › Application tab › Terminal for Terminal
- Share menu › Change View › Zoom in for mismatch of screen size/resolution (Participants)
- (Presenter) Change to 75% and back to 100% (solves participants with smaller screen image overlap)
- Leave the application on the original display
- Beware blued hatched rectangles — from other (hidden) windows or contextual menus
- Presenter screen pointer affects viewer display — beware of moving the pointer away from the application
- First time: System Preferences › Security & Privacy › Privacy › Accessibility

2.5 Adobe Connect — Ending a Meeting

- *Notes for the tutor only*
- **Student:** Meeting › Exit Adobe Connect
- **Tutor:**
- **Recording** Meeting › Stop Recording ✔
- **Remove Participants** Meeting › End Meeting... ✔
  - Dialog box allows for message with default message:
  - *The host has ended this meeting. Thank you for attending.*
- **Recording availability** *In course Web site for joining the room, click on the eye icon in the list of recordings under your recording* — edit description and name
- **Meeting Information** Meeting › Manage Meeting Information — can access a range of information in Web page.
- **Attendance Report** see course Web site for joining room

2.6 Adobe Connect — Invite Attendees

- **Provide Meeting URL** Menu › Meeting › Manage Access & Entry › Invite Participants...
- **Allow Access without Dialog** Menu › Meeting › Manage Meeting Information provides new browser window with *Meeting Information* Tab bar › Edit Information
- Check *Anyone who has the URL for the meeting can enter the room*
- Default *Only registered users and accepted guests may enter the room*
- **Reverts to default next session but URL is fixed**
- Guests have blue icon top, registered participants have yellow icon top — same icon if URL is open

- See Start, attend, and manage Adobe Connect meetings and sessions

2.7 Layouts

- **Creating new layouts** example *Sharing* layout
- Menu > Layouts > Create New Layout... > Create a New Layout dialog > Create a new blank layout and name it *PMolyMain*
- New layout has no Pods but does have Layouts Bar open (see Layouts menu)
- **Pods**
- Menu > Pods > Share > Add New Share and resize/position — initial name is *Share n*
- **Rename Pod** Menu > Pods > Manage Pods... > Manage Pods > Select > Rename or Double-click & rename
- Add Video pod and resize/reposition
- Add Attendance pod and resize/reposition
- Add Chat pod — name it *PMolyChat* — and resize/reposition
- Dimensions of **Sharing** layout (on 27-inch iMac)
  - Width of Video, Attendees, Chat column 14 cm
  - Height of Video pod 9 cm
  - Height of Attendees pod 12 cm
  - Height of Chat pod 8 cm
- **Duplicating Layouts** does *not* give new instances of the Pods and is probably not a good idea (apart from local use to avoid delay in reloading Pods)

2.8 Chat Pods

- **Format Chat text**
- Chat Pod > menu icon > My Chat Color
- Choices: Red, Orange, Green, Brown, Purple, Pink, Blue, Black
- Note: Color reverts to Black if you switch layouts
- Chat Pod > menu icon > Show Timestamps

Go to Table of Contents

3 Binary Trees — Introduction

- The *tree data structure* is the most widely used non-linear structure in many algorithms.

- Almost all algorithms that take logarithmic time, $O(\log n)$, do so because of an underlying tree structure.
- Common examples
- Binary search tree — this is used in many search applications
- Huffman coding tree — used in compression algorithms in, for example, JPEG and MP3 files
- Heaps — used to implement priority queues
- B-trees — generalisation of Binary search trees used in databases.

3.1 Binary Trees — Terminology

- **Binary Tree definition** — a Binary tree is either
  - an `Empty Tree` or
  - a `Node with an item and two subtrees`
  - One subtree is designated a left subtree and the other a right subtree
- Note that this is a recursive or inductive definition — this is very common in programming.
- Can also define trees as *graphs without cycles* — see *graph notes*

Other Recursive Data Structures

- Other examples of recursive or inductively defined data structures we have seen include:
- A `List` is either
  - an `Empty List` or
  - an `Item followed by the rest of the list`
- A `Stack` is either
  - an `Empty Stack` or
  - the `Top item followed by the rest of the stack`
- In each case the recursive nature of the data structure definition frequently gives a clue about how to write a recursive program for a computational problem.

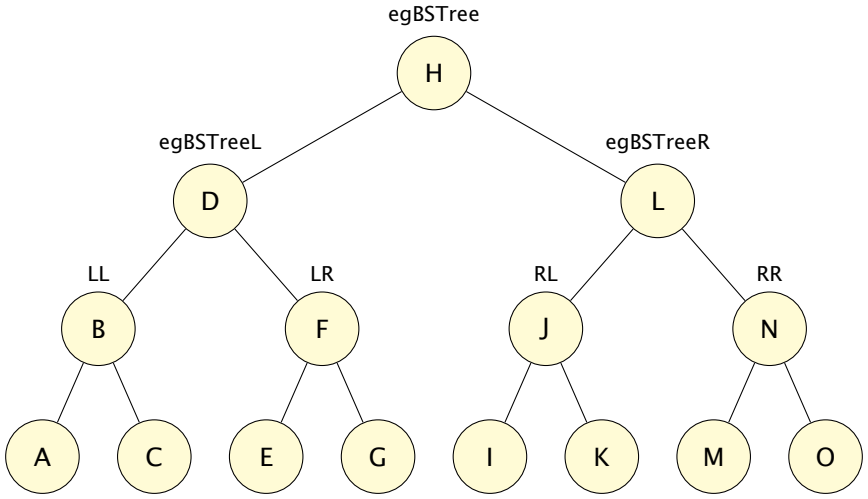
Binary Trees — Terminology

- `Children` — subtrees of a node that are not empty
- `Leaves` — nodes with two empty subtrees
- `Full Binary Tree` — every node other than the leaves has two non empty subtrees
- `Perfect Binary Tree` — all leaves are at the same level (or depth) children

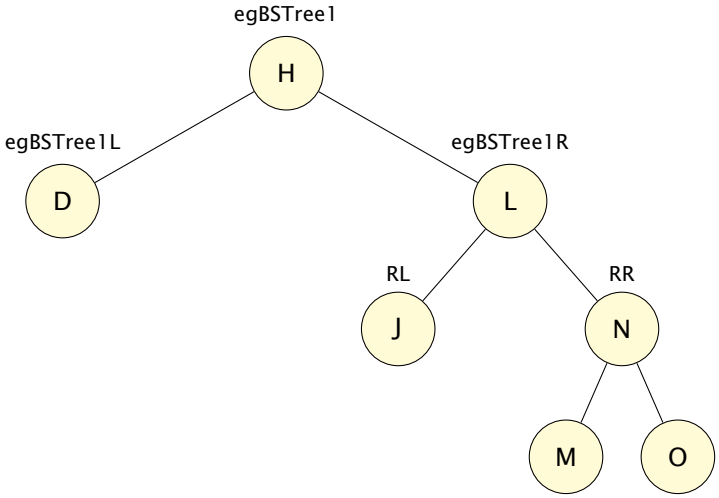
- Complete Binary Tree — every level, except possibly the last, is completely filled, and all nodes are as far left as possible — used for *Binary Heap*
- **Health Warning:** the terminology varies from text to text and between graph theory in mathematics and computing.

3.2 Binary Tree Examples

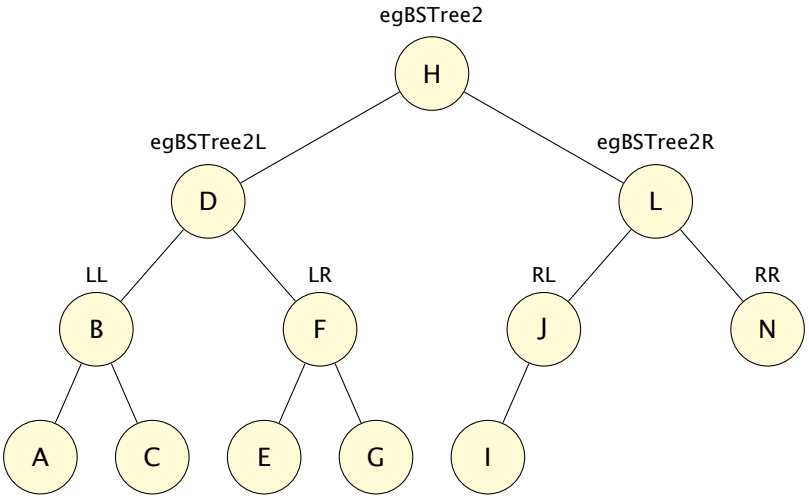
Example egBSTree



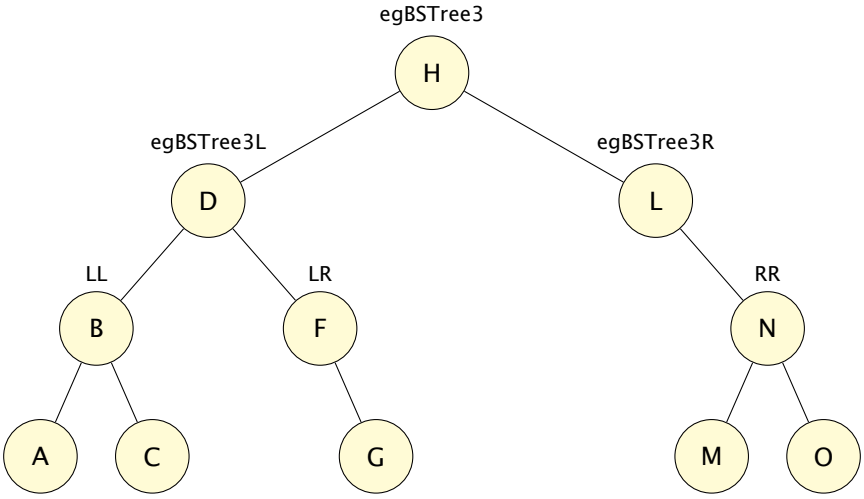
Example egBSTree1



Example egBSTree2



Example egBSTree3



Activity 1 Binary Tree Types

- What types of trees are the above example trees ?
- egBSTree
- egBSTree1
- egBSTree2
- egBSTree3

Go to Answer

Answer 1 Binary Tree Types

- egBSTree — perfect
- egBSTree1 — full
- egBSTree2 — complete
- egBSTree3 — just a binary tree

Go to Activity

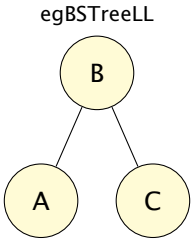
3.3 Representation of Binary Trees

3.3.1 Python Representation

- We shall represent nodes by a named tuple — a *quick and dirty* object recommended by Guido van Rossum (author of the Python programming language).
- `namedtuple()` is a factory function for creating tuple subclasses with named fields
- It is imported from the `collections` module.

```
9  from collections import namedtuple
11 EmptyBT = namedtuple('EmptyBT',[])
13 NodeBT = namedtuple('NodeBT'
14                     ,['dataBT','leftBT','rightBT'])
```

- The Python code above is in the file Python/M269TutorialBinaryTrees2019J.py
- The line numbers in the margin correspond to the line numbers in the file.
- Notational convention:
- Python reserved identifiers are shown in `this color`
- User defined data constructors such as NodeBT and EmptyBT are shown in `that color`
- **Health Warning:** these notes may not be totally consistent with syntax colouring.
- Here is a diagram of a small binary tree `egBSTreeLL` and its implementation in Python using our named tuple




```
52 egBSTreeLL = NodeBT('B',
53                NodeBT('A',EmptyBT(),EmptyBT()),
54                NodeBT('C',EmptyBT(),EmptyBT())
55              )
```

- If we ask Python to print the value of a named tuple then it lists the tuple contents in a `name=value` format.

```
Python3>>> egBSTreeLL
NodeBT(dataBT='B',
  leftBT=NodeBT(dataBT='A'
                , leftBT=EmptyBT(), rightBT=EmptyBT()),
  rightBT=NodeBT(dataBT='C'
                 , leftBT=EmptyBT(), rightBT=EmptyBT())
)
```

- (Some line breaks and spaces have been added to make it easier to read)
- We display the Python prompt as `Python3>>>`
- We can access the fields in the named tuple using the `.fieldName` operator (as with objects)

```
Python3>>> egBSTreeLL.dataBT == 'B'
True
Python3>>> egBSTreeLL.leftBT.dataBT == 'A'
True
Python3>>> egBSTreeLL.leftBT == NodeBT(dataBT='A', leftBT=EmptyBT(), rightBT=EmptyBT())
True
```

- The long input expression can be spread over several lines with implicit line joining

```
Python3>>> (egBSTreeLL.leftBT
... == NodeBT(dataBT='A'
...           , leftBT=EmptyBT()
...           , rightBT=EmptyBT()))
True
```

- Here is the implementation of egBSTree3 in Python using our named tuple.

```
637 egBSTree3 = NodeBT('H',
638               NodeBT('D',
639                 NodeBT('B',
640                   NodeBT('A',EmptyBT(),EmptyBT()),
641                   NodeBT('C',EmptyBT(),EmptyBT())
642                 ),
643                 NodeBT('F',
644                   EmptyBT(),
645                   NodeBT('G',EmptyBT(),EmptyBT())
646                 )
647               ),
648               NodeBT('L',
649                 EmptyBT(),
650                 NodeBT('N',
651                   NodeBT('M',EmptyBT(),EmptyBT()),
652                   NodeBT('O',EmptyBT(),EmptyBT())
653                 )
654               )
655             )
```

Activity 2 Python Representation

- Write Python implementations of the following trees (from the diagrams above) using the named tuple NodeBT and EmptyBT
- egBSTree1
- egBSTree2
- egBSTree

Go to Answer

Answer 2 Python Representation — egBSTree1

```
602 egBSTree1 = NodeBT('H',
603               NodeBT('D',EmptyBT(),EmptyBT()),
604               NodeBT('L',
605                 NodeBT('J',EmptyBT(),EmptyBT()),
606                 NodeBT('N',
607                   NodeBT('M',EmptyBT(),EmptyBT()),
608                   NodeBT('O',EmptyBT(),EmptyBT())
609                 )
610               )
611             )
```

Answer 2 Python Representation — egBSTree2

```
615 egBSTree2 = NodeBT('H',
616               NodeBT('D',
617                 NodeBT('B',
618                   NodeBT('A',EmptyBT(),EmptyBT()),
619                   NodeBT('C',EmptyBT(),EmptyBT())
```

```
                ),
                NodeBT('F',
                  NodeBT('E',EmptyBT(),EmptyBT()),
                  NodeBT('G',EmptyBT(),EmptyBT())
                )
              ),
              NodeBT('L',
                NodeBT('J',
                  NodeBT('I',EmptyBT(),EmptyBT()),
                  EmptyBT()
                ),
                NodeBT('N',EmptyBT(),EmptyBT())
              )
            )
```

Answer 2 Python Representation — egBSTree

```
egBSTree = NodeBT('H',
             NodeBT('D',
               NodeBT('B',
                 NodeBT('A',EmptyBT(),EmptyBT()),
                 NodeBT('C',EmptyBT(),EmptyBT())
               ),
               NodeBT('F',
                 NodeBT('E',EmptyBT(),EmptyBT()),
                 NodeBT('G',EmptyBT(),EmptyBT())
               )
             ),
             NodeBT('L',
               NodeBT('J',
                 NodeBT('I',EmptyBT(),EmptyBT()),
                 NodeBT('K',EmptyBT(),EmptyBT())
               ),
               NodeBT('N',
                 NodeBT('M',EmptyBT(),EmptyBT()),
                 NodeBT('O',EmptyBT(),EmptyBT())
               )
             )
           )
```

Go to Activity

3.3.2 Haskell Representation

```
module M269TutorialBinaryTrees2020J where
  import Data.List
  import Data.Maybe
```

1. A Haskell script starts with a module header which starts with the reserved identifier, `module` followed by the module name, `M269TutorialBinaryTrees2017J`
2. The module name must start with an upper case letter and is the same as the file name (without its extension of *.lhs*)
3. Haskell uses *layout* (or the *off-side rule*) to determine scope of definitions, similar to Python
4. The body of the module follows the reserved identifier `where` and starts with two `import` declarations
5. These import the built-in libraries `Data.List` and `Data.Maybe`
6. We use the `sort` function from `Data.List`.
7. The `Maybe` datatype from `Data.Maybe` will be used at the end of this script to implement the trees with data.

Haskell Binary Tree Datatype Declaration

```haskell
data BinTree a
  = EmptyBT | NodeBT a (BinTree a) (BinTree a)
    deriving (Eq, Show, Read)
```

1. The reserved identifier `data` introduces the name `BinTree`. The `a` following the type name is a *type variable*, that is, a variable that ranges over types. So a could be `Int` for a tree with numbers or `String` for a tree with text.
2. Following the `=` are the names or constructors of the elements of the type — in this case we have `EmptyBT` and `NodeBT` — these names are invented by the programmer — the only Haskell requirement is that they start with an upper case letter.
3. `NodeBT` takes 3 arguments which are an item of type `a` and two Binary Trees
4. The `deriving` part of the declaration automates the production of instances of the *Type Classes* `Eq` for equality and `Show` and `Read` for writing and reading.

Haskell egBSTree Implementation

```haskell
data Letters
  = A|B|C|D|E|F|G|H|I|J|K|L|M|N|O
    deriving (Eq, Ord, Enum, Show, Read)

egBSTree :: BinTree Letters
egBSTree
  = NodeBT H
      (NodeBT D
        (NodeBT B
           (NodeBT A EmptyBT EmptyBT)
           (NodeBT C EmptyBT EmptyBT))
        (NodeBT F
           (NodeBT E EmptyBT EmptyBT)
           (NodeBT G EmptyBT EmptyBT))
      )
      (NodeBT L
        (NodeBT J
           (NodeBT I EmptyBT EmptyBT)
           (NodeBT K EmptyBT EmptyBT))
        (NodeBT N
           (NodeBT M EmptyBT EmptyBT)
           (NodeBT O EmptyBT EmptyBT))
      )
```

Haskell Code Description

- Reserved identifiers, built in type class names, functions and constants are shown in `this color`
- User defined data constructors such as `NodeBT` and `EmptyBT` are shown in `that color`
- The type classes `Ord` is for those types that provide ordering relations `(<)`, `(<=)`, `(>=)`, `(>)`, `max`, `min` and `compare`
- The type class `Enum` is for those types which provide enumerations (such as `[1..4]` for `[1,2,3,4]`) and the functions `succ` and `pred` for successor and predecessor.
- **Health Warning:** these notes may not be totally consistent with syntax colouring.

Haskell Example Binary Trees 2

```
33  egBSTreeL :: BinTree Letters
34  egBSTreeL
35    =  NodeBT D
36         (NodeBT B
37            (NodeBT A EmptyBT EmptyBT)
38            (NodeBT C EmptyBT EmptyBT))
39         (NodeBT F
40            (NodeBT E EmptyBT EmptyBT)
41            (NodeBT G EmptyBT EmptyBT))

43  egBSTreeLL :: BinTree Letters
44  egBSTreeLL
45    =  NodeBT B
46         (NodeBT A EmptyBT EmptyBT)
47         (NodeBT C EmptyBT EmptyBT)

49  egBSTreeLR :: BinTree Letters
50  egBSTreeLR
51    =  NodeBT F
52         (NodeBT E EmptyBT EmptyBT)
53         (NodeBT G EmptyBT EmptyBT)
```

Haskell Example Binary Trees 3

```
55  egBSTreeR :: BinTree Letters
56  egBSTreeR
57    =  NodeBT L
58         (NodeBT J
59            (NodeBT I EmptyBT EmptyBT)
60            (NodeBT K EmptyBT EmptyBT))
61         (NodeBT N
62            (NodeBT M EmptyBT EmptyBT)
63            (NodeBT O EmptyBT EmptyBT))

65  egBSTreeRL :: BinTree Letters

67  egBSTreeRL
68    =  NodeBT J
69         (NodeBT I EmptyBT EmptyBT)
70         (NodeBT K EmptyBT EmptyBT)

72  egBSTreeRR :: BinTree Letters
73  egBSTreeRR
74    =  NodeBT N
75         (NodeBT M EmptyBT EmptyBT)
76         (NodeBT O EmptyBT EmptyBT)
```

3.4 Binary Tree Operations

3.4.1 Binary Tree Operations — Python

- We now provide functions to create, inspect and take apart binary trees

```
88  def makeEmptyBT():
89    return EmptyBT()

91  def makeBT(x,t1,t2):
92    return NodeBT(x,t1,t2)

94  def isEmptyBT(t):
95    return t == EmptyBT()
```

- makeEmptyBT, makeBT are *constructor* functions — we could have used the raw named tuples but the discipline is good for you and it makes it easier to refactor in future
- isEmptyBT uses the == operator for identity check (not identity (is))
- this is in line with *PEP 8 — Style Guide for Python Code*
- to see an example of why using (==) for comparison with None may produce odd results see When is the == operator not equivalent to the is operator? or Python None comparison: shouldI use is or ==?

```
99  def getDataBT(t):
100   if isEmptyBT(t):
101     raise RuntimeError("getLeftBT applied to EmptyBT()")
102   else:
103     return t.dataBT

105 def getLeftBT(t):
106   if isEmptyBT(t):
107     raise RuntimeError("getLeftBT applied to EmptyBT()")
108   else:
109     return t.leftBT

111 def getRightBT(t):
112   if isEmptyBT(t):
113     raise RuntimeError("getRightBT applied to EmptyBT()")
114   else:
115     return t.rightBT
```

```
117 def heightBT(t):
118   if isEmptyBT(t):
119     return 0
120   else:
121     return (1 + max(heightBT(t.leftBT)
122                     ,heightBT(t.rightBT)))
```

- We define the height of a binary tree recursively:
- The height of an *empty tree* is 0
- The height of a *non-empty tree* is 1 plus the maximum of the heights of its two sub-trees

We will mainly use the *height* of a tree in section 5 *Height Balanced Trees*.

Note that some texts (including Miller & Ranum) define the height of a singleton node to be zero — just subtract one from the height as defined here. Since we will only be interested in differences between heights in height balanced trees, it will have no overall effect on our computations.

Some texts do not use empty trees — so where these notes might say a singleton node has an element and two empty subtrees, some texts might say a singleton node has no subtrees. This leads to problems with nodes that have only one non-empty sub-tree — without an explicit empty tree we would have to attach information to indicate whether it was a left or right sub-tree.

3.4.2 Binary Tree Operations — Haskell

```
78  makeEmptyBT :: BinTree a
79  makeEmptyBT  =  EmptyBT

81  makeBT :: a -> BinTree a -> BinTree a -> BinTree a
```

```
82  makeBT x t1 t2  =  NodeBT x t1 t2

84  isEmptyBT :: BinTree a -> Bool
85  isEmptyBT  EmptyBT  =  True
86  isEmptyBT  t        =  False
```

Note that `isEmptyBT` would require `a` to be an instance of the type class `Eq` if we define it as follows

```
isEmptyBT  ::  (Eq a) => BinTree a -> Bool
isEmptyBT  t  =  (t == EmptyBT)
```

Haskell Code Description 1

- `f :: t` is a type signature for variable `f` that reads *`f` is of type `t`*
- `f :: t1 -> t2` means that `f` has the type of a function that takes elements of type `t1` and returns elements of type `t2`
- We read the type of `makeBT` as *`makeBT` takes an item (of some type, `a`), two binary trees (with items of the same type, `a`)* and returns a binary tree (of the same type)
- `f x` — function application is denoted by juxtaposition and is more binding than (almost) any other operation.
- `f x y` means `(f x) y` — function application is left associative

Binary Tree Operations 2 — Haskell

```
88  getLeftBT :: BinTree a -> BinTree a
89  getLeftBT  (NodeBT x leftT rightT) = leftT
90  getLeftBT  EmptyBT
91    =  error "getLeftBT applied to EmptyBT"

93  getRightBT :: BinTree a -> BinTree a
94  getRightBT  (NodeBT x leftT rightT) = rightT
95  getRightBT  EmptyBT
96    =  error "getRightBT applied to EmptyBT"

98  height EmptyBT = 0
99  height (NodeBT x leftT rightT)
100   = 1 + max (height leftT) (height rightT)
```

- Notice we have forgotten to give the type signature for `height`
- Haskell can *deduce* the most general type
- Does it then matter if we leave out the type signature ?
- Yes — see below

Haskell Code Description 2

- `error` is a built-in function that takes a string and stops execution, displaying an error message
- `max` is a built-in function that takes two items and returns the maximum

Health Warning 1

- Some texts (including Miller & Ranum) define the height of a singleton node to be zero — just subtract one from the height as defined here.
- Some texts do not use empty trees — so where these notes might say a singleton nodes has an element and two empty subtrees, some texts might say a singleton node has no subtrees

Type Signature for `height`

- We can ask Haskell/GHC for the type of `height`

```
GHCi> :t height
height :: (Num a, Ord a) => BinTree t -> a
```

- This states that whatever type `Height` returns, `a`, it must be in the number class, `Num`, since we use `(+)` and it must be in the order class, `Ord`, since we use `max`
- This is more general than we require — does that matter? Yes:
- Error messages will be more helpful with more precise type signatures
- Frequently, defining the type of a function is half of the programming effort
- More precise types can save computation

```
height :: BinTree a -> Int
```

Haskell Numbers, `Int` and `Integer`

- Haskell provides several kinds of numbers
- Numeric function names and operators are usually overloaded and Haskell controls this via the `Num` and other classes
- The `Prelude` provides fixed sized integers (`Int`), arbitrary precision integers (`Integer`), single and double precision floating, (`Float`), (`Double`)

```
GHCi> maxBound :: Int
9223372036854775807
GHCi> 2^63 - 1
9223372036854775807
GHCi> minBound :: Int
-9223372036854775808
```

- To resolve ambiguities in the class `Num` Haskell provides a `default` declaration
- If no `default` declaration is given it is assumed to be:

```
default (Integer, Double)
```

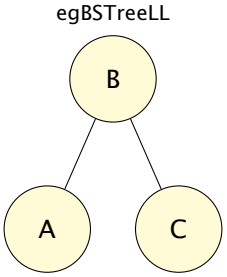
3.5 Tree Traversals

- Many applications require visiting each node in a binary tree and doing some processing.

- This could be adding quantities to find a total, identifying the number of nodes with a particular property and so on.
- There are several common patterns of visiting each node or traversing a tree
  - Depth first where the search tree is deepened as much as possible on each child before visiting the next sibling
  - Breadth first where we visit every node on a level before visiting the next level
- Each traversal takes a tree and returns a list of items at the nodes of the tree

3.5.1 Tree Traversals — Depth First

- **In-Order traversal of tree `t`**
  1. If `t` is an empty tree then return the empty list
  2. Otherwise do an In Order traversal of the left subtree of `t` then append a list just containing the data item at the root of `t` followed by an In Order traversal of the right subtree of `t`
- **Pre-Order traversal of tree `t`**
  - As In-Order but output a list with the item at the root of `t` before traversing the two subtrees
- **Post-Order traversal of tree `t`**
  - As Pre-Order but output a list with the item at the root of `t` after traversing the two subtrees
- *In-Order*, *Pre-Order* and *Post-Order* traversals are collectively termed *Depth First Traversals*
- Tree egBSTreeLL Python code at line 52 on page 15



- The *Depth first* traversals are implemented in Python by inOrderBT(), preOrderBT() and postOrderBT()

```
Python3>>> inOrderBT(egBSTreeLL)
['A', 'B', 'C']
Python3>>> preOrderBT(egBSTreeLL)
['B', 'A', 'C']
Python3>>> postOrderBT(egBSTreeLL)
['A', 'C', 'B']
```

3.5.2 Depth First Traversals — Python

```
126 def inOrderBT(t):
127   if isEmptyBT(t):
128     return []
129   else:
130     return (inOrderBT(t.leftBT) + [t.dataBT]
131             + inOrderBT(t.rightBT))

133 def preOrderBT(t):
134   if isEmptyBT(t):
135     return []
136   else:
137     return ([t.dataBT] + preOrderBT(t.leftBT)
138             + preOrderBT(t.rightBT))

140 def postOrderBT(t):
141   if isEmptyBT(t):
142     return []
143   else:
144     return (postOrderBT(t.leftBT)
145             + postOrderBT(t.rightBT) + [t.dataBT])
```

3.5.3 Depth First Traversals — Haskell

```
102   inOrderBT :: BinTree a -> [a]
103   inOrderBT EmptyBT = []
104   inOrderBT (NodeBT x leftT rightT)
105     = (inOrderBT leftT) ++ [x]
106       ++ (inOrderBT rightT)

108   preOrderBT :: BinTree a -> [a]
109   preOrderBT EmptyBT = []
110   preOrderBT (NodeBT x leftT rightT)
111     =  [x] ++ (preOrderBT leftT)
112        ++ (preOrderBT rightT)

114   postOrderBT :: BinTree a -> [a]
115   postOrderBT EmptyBT = []
116   postOrderBT (NodeBT x leftT rightT)
117     = (postOrderBT leftT) ++ (postOrderBT rightT)
118       ++ [x]
```

Haskell Code Description 3

- `[x1,x2,...,xn]` denotes a list of items (all of the same type)
- `(:)` is a list constructor

  `1 : [2,3] == [1,2,3]`
- `(++)` is the list append operator

  `[1,2] ++ [3,4] == [1,2,3,4]`
- `[]` denotes the empty list

  `[1,2] ++ [] == [1,2]`

  `1 : [] == [1]`

  `[1,2] : [] == [[1,2]]`

Activity 3 Depth First Traversals

- Give the lists of items in an in-order traversal of `egBSTree`, `egBSTree1`, `egBSTree2`, `egBSTree3`

- Give the lists of items in a pre-order traversal of egBSTree, egBSTree1, egBSTree2, egBSTree3
- Give the lists of items in a post-order traversal of egBSTree, egBSTree1, egBSTree2, egBSTree3
- Depth first traversal code is from line 126 on page 24 (Python) or line 102 on page 24 (Haskell)
- Binary tree code is from line 16 on page 17 (Python) or line 13 on page 18 (Haskell)

Go to Answer

Answer 3 Depth First Traversals — In-Order

```
Python3>>> inOrderBT(egBSTree)
['A', 'B', 'C', 'D', 'E', 'F', 'G',
 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O']
Python3>>> inOrderBT(egBSTree1)
['D', 'H', 'J', 'L', 'M', 'N', 'O']
Python3>>> inOrderBT(egBSTree2)
['A', 'B', 'C', 'D', 'E', 'F', 'G',
 'H', 'I', 'J', 'L', 'N']
Python3>>> inOrderBT(egBSTree3)
['A', 'B', 'C', 'D', 'F', 'G', 'H',
 'L', 'M', 'N', 'O']
```

- (Line breaks introduced for layout)

Answer 3 Depth First Traversals — Pre-Order

```
Python3>>> preOrderBT(egBSTree)
['H', 'D', 'B', 'A', 'C', 'F', 'E',
 'G', 'L', 'J', 'I', 'K', 'N', 'M', 'O']
Python3>>> preOrderBT(egBSTree1)
['H', 'D', 'L', 'J', 'N', 'M', 'O']
Python3>>> preOrderBT(egBSTree2)
['H', 'D', 'B', 'A', 'C', 'F', 'E',
 'G', 'L', 'J', 'I', 'N']
Python3>>> preOrderBT(egBSTree3)
['H', 'D', 'B', 'A', 'C', 'F', 'G',
 'L', 'N', 'M', 'O']
```

Answer 3 Depth First Traversals — Post-Order

```
Python3>>> postOrderBT(egBSTree)
['A', 'C', 'B', 'E', 'G', 'F', 'D',
 'I', 'K', 'J', 'M', 'O', 'N', 'L', 'H']
Python3>>> postOrderBT(egBSTree1)
['D', 'J', 'M', 'O', 'N', 'L', 'H']
Python3>>> postOrderBT(egBSTree2)
['A', 'C', 'B', 'E', 'G', 'F', 'D',
 'I', 'J', 'N', 'L', 'H']
Python3>>> postOrderBT(egBSTree3)
['A', 'C', 'B', 'G', 'F', 'D', 'M',
 'O', 'N', 'L', 'H']
```

Go to Activity

3.5.4 Tree Traversals — Breadth First

- A *breadth first* (or *level order*) tree traversal visits each node in the tree level by level from the top (or root).
- We shall write several versions to illustrate some of the choices.

- Version 1 uses a helper function which enqueues the trees yet to be visited
- Version 2 is similar to version 1 but enables output as we traverse the tree
- Version 3 takes a different approach — it generates a list of the levels and then concatenates them together
- `breadthBT1` takes a tree `t` and returns a list of items at the nodes, level by level from the top — it calls the helper function `bfTraverse` with an empty list and a list just with `t`
- `bfTraverse` takes two arguments, `vs`, a list of values at visited nodes, and `ts`, a list of trees yet to visit
  1. If `ts` is empty then return `vs`
  2. Else if the first element of `ts` is empty then recursively apply `bfTraverse` to `vs` and the tail of `ts`
  3. Otherwise, recursively apply `bfTraverse` to `vs` appended with the item at the root of the first tree in `ts` and the tail of `ts` appended with the children of the first tree in `ts`

3.5.5 Breadth First Traversals — Python

```
150 def breadthBT1(t):
151   return bfTraverse([],[t])

153 def bfTraverse(vs,ts):
154   if ts == []:
155     return vs
156   elif isEmptyBT(ts[0]):
157     return bfTraverse(vs,ts[1:])
158   else:
159     return (bfTraverse(vs + [ts[0].dataBT],
160                 ts[1:] + [ts[0].leftBT,ts[0].rightBT]))
```

- Naive approach
- Accumulates all nodes visited (to be output at the end)
- Enqueues the children of nodes visited (whose roots will be visited in turn)
- Why is version 1 naive ?
- If the tree is very big — we get no output till all the nodes have been visited.
- If the tree is infinite — we get no output ever.

Breadth First Traversal

Algorithm 2 Description

- `breadthBT2` takes a tree `t` and returns a list of items at the nodes, level by level from the top — it calls the helper function `lbfBT` with a list just with `t`
- `lbfBT` takes one argument, `ts`, a list of trees yet to visit
  1. If `ts` is empty then return the empty list

2. Else if the first element of `ts` is empty then recursively apply `lbfBT` to `vs` and the tail of `ts`
3. Otherwise, return a list with the item at the root of the first tree in `ts` followed by the result of a recursive call of `lbfBT` to the tail of `ts` appended with the children of the first tree in `ts`

Breadth First Traversal 2

Python

```
162 def breadthBT2(t):
163   return lbfBT([t])

165 def lbfBT(ts):
166   if ts == []:
167     return []
168   elif isEmptyBT(ts[0]):
169     return lbfBT(ts[1:])
170   else:
171     return ([ts[0].dataBT]
172         + lbfBT(ts[1:] + [ts[0].leftBT,ts[0].rightBT]))
```

- This arranges the return value so that it can be given up item by item
- Python still requires a further construct (`yield expression`) to actually return the values as they are generated
- Haskell allows us to do that without special notation (see below)

3.5.6 Breadth First Traversals — Haskell

```
120 breadthBT1  ::  BinTree a -> [a]

122 breadthBT1 t = bfTraverse  []  [t]

124 bfTraverse :: [a] -> [BinTree a] -> [a]

126 bfTraverse visited [] = visited

128 bfTraverse visited (EmptyBT : ts)
129   = bfTraverse visited ts

131 bfTraverse visited ((NodeBT x leftT rightT) : ts)
132   = bfTraverse (visited ++ [x])
133                (ts ++ [leftT,rightT])
```

- Why is version 1 naive ?
- If the tree is very big — we get no output till all the nodes have been visited.
- If the tree is infinite — we get no output ever.

Breadth First Traversal 2

Haskell

```
135 breadthBT2 :: BinTree a -> [a]

137 breadthBT2 t = lbfBT [t]

139 lbfBT :: [BinTree a] -> [a]
```

```
141  lbfBT []                = []
142  lbfBT (EmptyBT : ts) = lbfBT  ts
143  lbfBT ((NodeBT x leftT rightT) : ts)
144    = x : lbfBT (ts ++ [leftT, rightT])
```

- Lazy evaluation allows Haskell to output the resulting list as required

Breadth First Traversal 3

Haskell

```
146  breadthBT :: BinTree a -> [a]

148  breadthBT = concat . levelOrderBT
```

- `concat` is the built-in list concatenation function (see below)
- `levelOrderBT` is defined below — it takes a binary tree and outputs a list of lists of items at each level
- `(.)` is the *function composition* operator

```
      (f . g) x = f (g x)
```

- Where has the argument gone ? Point free style — can mislead beginners — argument not needed since not used in the definition. (see references)

`levelOrderBT`

```
150  levelOrderBT :: BinTree a -> [[a]]

152  levelOrderBT EmptyBT = []

154  levelOrderBT (NodeBT x leftT rightT)
155    =  [x] : longZipWith (++)
156             (levelOrderBT leftT)
157             (levelOrderBT rightT)

159  longZipWith :: (a -> a -> a) -> [a] -> [a] -> [a]

161  longZipWith f [] ys      = ys

163  longZipWith f (a:xs) []  = (a:xs)

165  longZipWith  f  (a:xs)  (b:ys)
166    =  (f a b) : (longZipWith f xs ys)
```

- `longZipWith` is like *zipping* up two lists to produce an output list combining the elements with a function.

Haskell Code Style

- Uses *higher order functions* — function composition `(.)`
  - Function composition `(.)`
  - `longZipWith`
- Higher order functions can take or return functions
- Uses *lazy evaluation* to decouple two tasks: concatenation and generating the level lists.

- *Lazy evaluation*: don't do today what you can put off till tomorrow because you may never have to do it — turns out to be a good evaluation strategy.

Haskell Code Description

concat

- concat is built-in — we could have defined it as follows

```
concat :: [[a]] -> [a]

concat [] = []
concat (xs : xss) = xs ++ concat xss
```

- The built-in definition uses a common higher order function to *capture the pattern of the definition*

```
concat xss = foldr (++) [] xss

foldr :: (a -> b -> b) -> b -> [a] -> b

foldr f z [] = z
foldr f z (x : xs)
  = f x (foldr f z xs)
```

- Python has a function reduce() which is similar to the Haskell foldl (fold left)
- Other functions such as sum, product are defined with folds
- Foldable Traversable in the Haskell Prelude from 2015 and GHC 7.10 generalised definitions of functions such as concat further

Activity 4 Breadth First Tree Traversals

1. Give the lists of items in a breadth first or level-order traversal of egBSTree, egBSTree1, egBSTree2, egBSTree3
2. Give the result of applying levelOrderBT to egBSTree, egBSTree1, egBSTree2, egBSTree3
3. What is the type of the expression:

```
longZipWith (++)
```

- Breadth first traversal code is from line 162 on page 27 (Python) or line 135 on page 27 (Haskell)
- Binary tree code is from line 16 on page 17 (Python) or line 13 on page 18 (Haskell)
- levelOrderBT code is from line 150 on page 28 (Haskell)

Go to Answer

Answer 4 Breadth First Tree Traversals — breadthBT2()

```
Python3>>> breadthBT1(egBSTree)
['H', 'D', 'L', 'B', 'F', 'J', 'N',
 'A', 'C', 'E', 'G', 'I', 'K', 'M', 'O']
Python3>>> breadthBT1(egBSTree1)
['H', 'D', 'L', 'J', 'N', 'M', 'O']
Python3>>> breadthBT1(egBSTree2)
['H', 'D', 'L', 'B', 'F', 'J', 'N',
 'A', 'C', 'E', 'G', 'I']
Python3>>> breadthBT1(egBSTree3)
['H', 'D', 'L', 'B', 'F', 'N', 'A',
 'C', 'G', 'M', 'O']
```

- (Line breaks added for layout)

Answer 4 Breadth First Tree Traversals — breadthBT2()

```
Python3>>> breadthBT2(egBSTree)
['H', 'D', 'L', 'B', 'F', 'J', 'N',
 'A', 'C', 'E', 'G', 'I', 'K', 'M', 'O']
Python3>>> breadthBT2(egBSTree1)
['H', 'D', 'L', 'J', 'N', 'M', 'O']
Python3>>> breadthBT2(egBSTree2)
['H', 'D', 'L', 'B', 'F', 'J', 'N',
 'A', 'C', 'E', 'G', 'I']
Python3>>> breadthBT2(egBSTree3)
['H', 'D', 'L', 'B', 'F', 'N', 'A',
 'C', 'G', 'M', 'O']
```

- (Line breaks added for layout)

Answer 4 Breadth First Tree Traversals — levelOrderBT egBSTree

```
egBSTree :: BinTree Letters
egBSTree
  = NodeBT H
      (NodeBT D
        (NodeBT B
           (NodeBT A EmptyBT EmptyBT)
           (NodeBT C EmptyBT EmptyBT))
        (NodeBT F
           (NodeBT E EmptyBT EmptyBT)
           (NodeBT G EmptyBT EmptyBT)))
      (NodeBT L
        (NodeBT J
           (NodeBT I EmptyBT EmptyBT)
           (NodeBT K EmptyBT EmptyBT))
        (NodeBT N
           (NodeBT M EmptyBT EmptyBT)
           (NodeBT O EmptyBT EmptyBT)))
```

```
GHCi> levelOrderBT egBSTree
[[H],[D,L],[B,F,J,N],[A,C,E,G,I,K,M,O]]
```

Answer 4 Breadth First Tree Traversals — levelOrderBT egBSTree1

```
egBSTree1 :: BinTree Letters
egBSTree1
  = NodeBT H
      (NodeBT D EmptyBT EmptyBT)
      (NodeBT L
         (NodeBT J EmptyBT EmptyBT)
         (NodeBT N
            (NodeBT M EmptyBT EmptyBT)
            (NodeBT O EmptyBT EmptyBT)))
```

```
GHCi> levelOrderBT egBSTree1
[[H],[D,L],[J,N],[M,O]]
```

Answer 4 Breadth First Tree Traversals — levelOrderBT egBSTree2

```
egBSTree2 :: BinTree Letters
egBSTree2
  = NodeBT H
      (NodeBT D
         (NodeBT B
            (NodeBT A EmptyBT EmptyBT)
            (NodeBT C EmptyBT EmptyBT))
         (NodeBT F
            (NodeBT E EmptyBT EmptyBT)
            (NodeBT G EmptyBT EmptyBT)))
      (NodeBT L
         (NodeBT J
            (NodeBT I EmptyBT EmptyBT)
            EmptyBT)
         (NodeBT N EmptyBT EmptyBT))
```

```
GHCi> levelOrderBT egBSTree2
[[H],[D,L],[B,F,J,N],[A,C,E,G,I]]
```

Answer 4 Breadth First Tree Traversals — `levelOrderBT egBSTree3`

```
egBSTree3 :: BinTree Letters
egBSTree3
  = NodeBT H
      (NodeBT D
        (NodeBT B
           (NodeBT A EmptyBT EmptyBT)
           (NodeBT C EmptyBT EmptyBT))
        (NodeBT F
           EmptyBT
           (NodeBT G EmptyBT EmptyBT)))
      (NodeBT L
         EmptyBT
         (NodeBT N
            (NodeBT M EmptyBT EmptyBT)
            (NodeBT O EmptyBT EmptyBT)))
```

```
GHCi> levelOrderBT egBSTree3
[[H],[D,L],[B,F,N],[A,C,G,M,O]]
```

Answer 4 Breadth First Tree Traversals — `longZipWith (++)`

```
GHCi> :type (longZipWith (++))
(longZipWith (++)) :: [[a]] -> [[a]] -> [[a]]
```

- How do we reconcile the above type with:

```
GHCi> :type longZipWith
longZipWith :: (a -> a -> a) -> [a] -> [a] -> [a]
```

Go to Activity

- Python does not do an optimisation for recursion called *tail recursion elimination*
- This means that you may need to convert your recursive program into one that just uses iteration.
- It is always possible to convert a recursive program to a non-recursive one using while or for loops
- We give some examples of doing this for the depth first tree traversals in section 6
- Note that Haskell does do *tail recursion elimination*
- Also, Haskell can do further optimisations with particular patterns of recursive programs.

4 Binary Search Trees

4.1 Binary Search Trees — Definition

- A binary search tree (BST) is a binary tree with the *binary search tree* property:
  1. The left sub tree contains nodes with keys less than the root node
  2. The right sub tree contains nodes with keys greater than the root node

3. The left and right sub trees must also be binary search trees
4. No nodes with duplicate keys
5. An empty tree is a binary search tree
6. Nothing else is a binary search tree

- The data at each node is to contain a key and any values. The operations required for a BST will include:

Binary Search Trees — Motivation

- A *perfect* binary tree of height h will have $2^h - 1$ nodes
- This means that there will be at most $\log_2(n+1)$ steps from the root of the tree to any node in the tree.
- This provides the basis for efficient searching if we give an appropriate structure to the tree — a *Binary Search Tree (BST)*
- However we have to keep the BST as near to a perfect tree otherwise we lose the advantage

Activity 5 Nodes of Perfect Tree

- Justify the statement that a *perfect* binary tree of height h will have $2^h - 1$ nodes

Go to Answer

Answer 5 Nodes of Perfect Tree

- There are many ways of showing that a *perfect* binary tree of height h will have $2^h - 1$ nodes — here is one way
- Let N_h be the number of nodes in a perfect tree and L_h be the number of leaves in the same tree.
- Then we have $L_0 = 0$, $L_1 = 1$, $L_2 = 2$, $L_3 = 4, \ldots$ and in general $L_h = 2^{h-1}$, $h \geqslant 1$
- Now $N_h = L_1 + L_2 + \cdots + L_h = 2^0 + 2^1 + \cdots + 2^{h-1}$
- $2N_h = 2^1 + 2^2 + \cdots + 2^{h-1} + 2^h$
- Subtract the N_h from $2N_h$ and we get $N_h = 2^h - 1$
- Notice that $N_h = 1 + 2 \times N_{h-1}$ — when we consider the performance we will use similar recurrence relations

Go to Activity

4.2 Inserting a Node

Inserting a Node — Description

- The function that takes a item (key and payload) and an existing BST has to return a new binary tree with the item inserted and the new tree must be a BST. We deal with each possible binary tree: an empty tree and a non-empty tree:
- To insert an item into an empty tree, we return a new tree which is a singleton node with the item and two empty sub-trees.

- To insert an item, with key k, into a tree which is a node with an item with key p and two sub-trees then we have two possibilities
  - If k is less than p then insert the item in the left sub-tree
  - If k is greater than p then insert the item in the right sub-tree
- We are going to assume duplicate keys are not allowed — in practice, we might have an error message or update the payload of an item with the same key but we are keeping things simple here.

4.2.1 Inserting a Node — Python

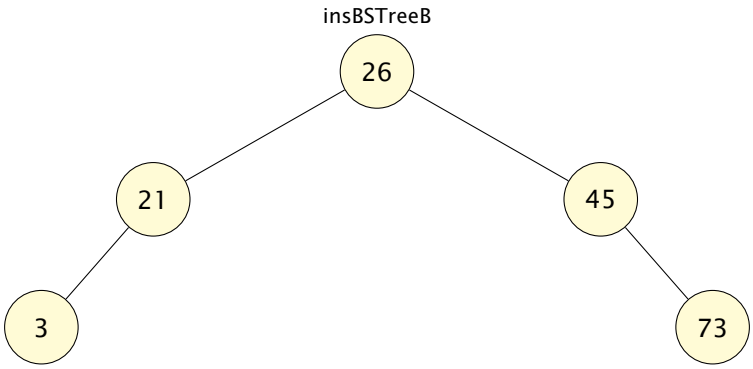
```
178 def insertBST(x,t):
179   if isEmptyBT(t):
180     return makeBT(x,makeEmptyBT(),makeEmptyBT())
181   else:
182     y = t.dataBT
183     if x < y:
184       return makeBT(y,insertBST(x,t.leftBT),t.rightBT)
185     elif x > y:
186       return makeBT(y,t.leftBT,insertBST(x,t.rightBT))
187     else:
188       return t
```

Activity 6 Inserting an Item

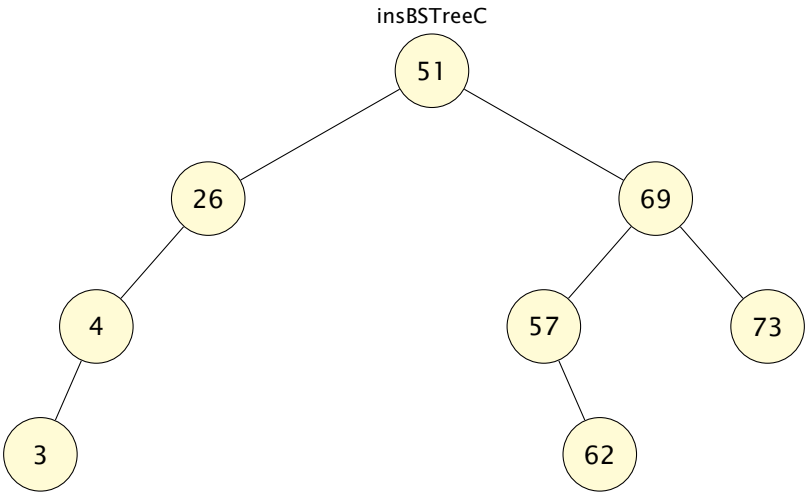
- Draw diagrams of the binary search trees that result from inserting an item with key 28 into each of the three BSTs in the diagrams below of `insBSTreeA`, `insBSTreeB`, `insBSTreeC`

insBSTreeA

insBSTreeB



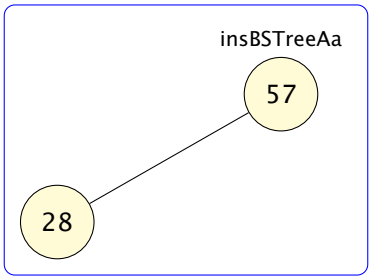
insBSTreeC



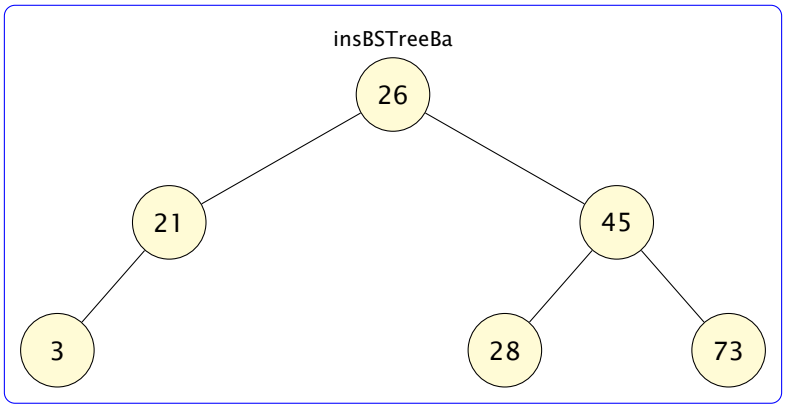
Go to Answer

Answer 6 Inserting an Item

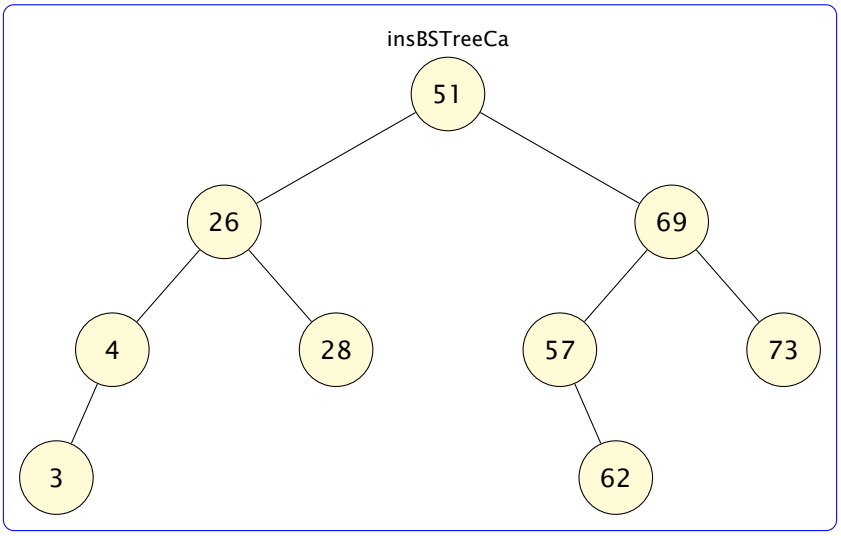
insBSTreeAa



insBSTreeBa



insBSTreeCa



Go to Activity

Activity 7 Membership

- Write Python code for a function, `inBST(k,t)` which take a key, k, and a BST, t and returns `True` if an item with key k is in the tree and `False` otherwise

Go to Answer

Answer 7 Membership

```
193 def inBST(k,t):
194   if isEmptyBT():
195     return False
196   else:
197     p = t.dataBT
198     if k < p:
199       return inBST(k,t.leftBT)
200     elif k > p:
201       return inBST(k,t.rightBT)
202     else:
203       return True
```

Go to Activity

Testing if a Binary Tree is a BST

- One strategy for this might be to do an in-order traversal of the tree and check that the list returned was an ordered list.

```
205 def isBSTree(t):
206   return orderedList(inOrderBT(t))

208 def orderedList(xs):
209   return (len(xs) <= 1
210           or (xs[0] < xs[1] and orderedList(xs[1:])))
```

- The ordering relation is `(<)` for strict ordering and no duplicates

Building a Binary Search Tree from a list of items

- We *could* insert a list of items one by one from the list in turn — here is the Python code:

```
212 def insertListBST(xs,t):
213   if xs == []:
214     return t
215   else:
216     return insertListBST (xs[1:], insertBST(xs[0],t))
```

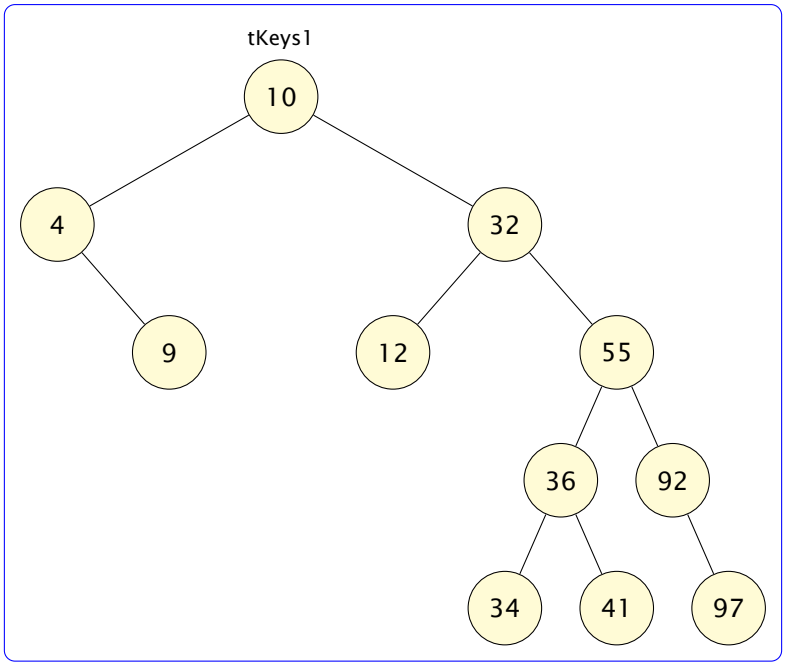
- However, see what happens when we insert various lists — how *compact* is the resulting tree ?

Activity 8 Insert List

- For the following lists of keys, draw the diagrams of the trees produced when `insertListBST()` is used to produce the trees from the lists inserting the keys into an initially empty tree
- `keys1 = [10, 4, 32, 12, 9, 55, 92, 97, 36, 41, 34]`
- `keys2 = [4, 9, 10, 12, 32, 97, 92, 55, 41, 34, 32]`
- `keys3 = [34, 10, 9, 4, 32, 12, 55, 41, 36, 97, 92]`

Go to Answer

Answer 8 Insert List `tKeys1 = insertListBST(keys1, makeEmptyBT())`

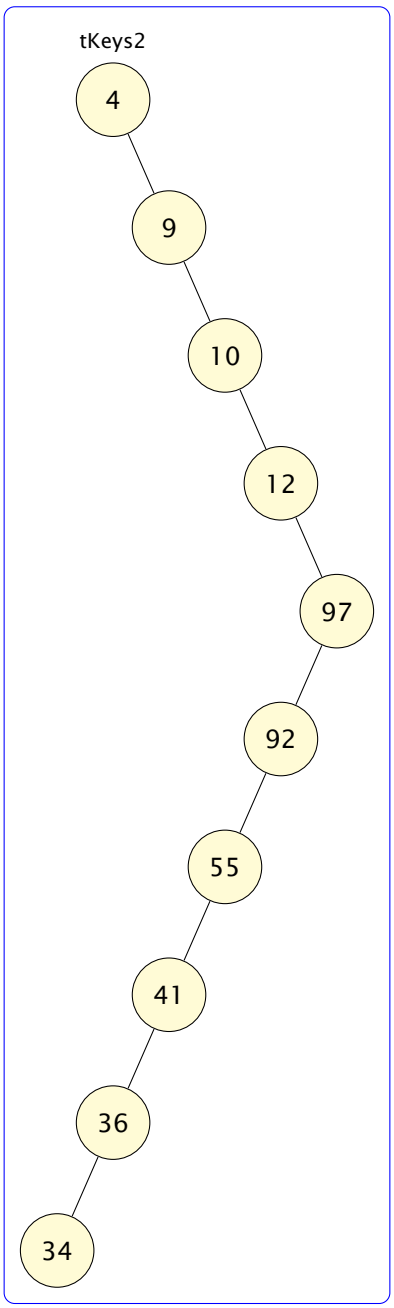



```
tKeys1 = NodeBT(10,
           NodeBT(4,
             EmptyBT,
             NodeBT(9, EmptyBT, EmptyBT)),
           NodeBT(32,
             NodeBT(12, EmptyBT, EmptyBT),
             NodeBT(55,
               NodeBT(36,
                 NodeBT(34, EmptyBT, EmptyBT),
                 NodeBT(41, EmptyBT, EmptyBT)),
               NodeBT(92,
                 EmptyBT,
                 NodeBT(97, EmptyBT, EmptyBT)))))

tKeys1Test = (tKeys1
              == insertListBST(keys1, makeEmptyBT()))
```

- Note that when the Python interpreter prints `tKeys1` it includes field names and values and has no line breaks.

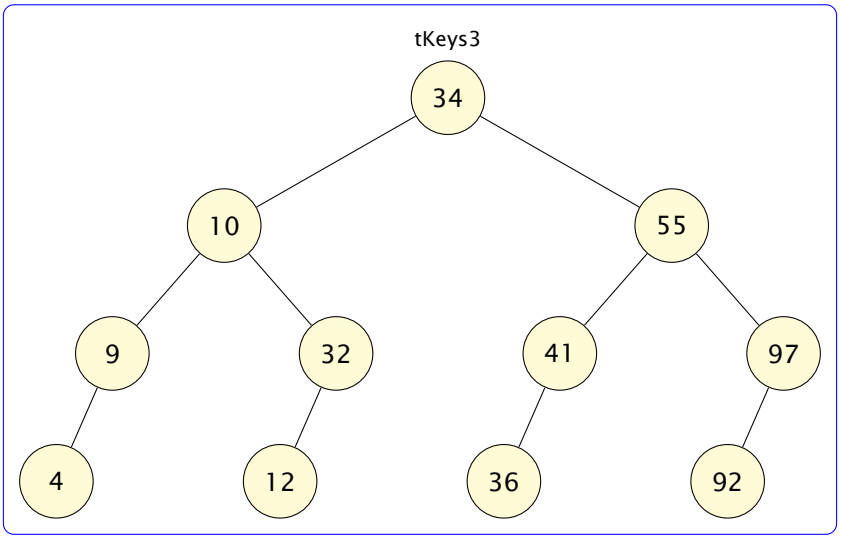
Answer 8 Insert List `tKeys2 = insertListBST(keys2, makeEmptyBT())`




```
tKeys2 = NodeBT(4, EmptyBT,
           NodeBT(9, EmptyBT,
             NodeBT(10, EmptyBT,
               NodeBT(12, EmptyBT,
                 NodeBT(32, EmptyBT,
                   NodeBT(97,
                     NodeBT(92,
                       NodeBT(55,
                         NodeBT(41,
                           NodeBT(36,
                             NodeBT(34, EmptyBT,
                                        EmptyBT),
                             EmptyBT),
                           EmptyBT),
                         EmptyBT),
                       EmptyBT),
                     EmptyBT),
                   EmptyBT))))))

tKeys2Test = (tKeys2
              == insertListBST(keys2, makeEmptyBT()))
```

Answer 8 Insert List `tKeys3 = insertListBST(keys3, makeEmptyBT())`




```
tKeys3 = NodeBT(34,
           NodeBT(10,
             NodeBT(9,
               NodeBT(4, EmptyBT, EmptyBT),
               EmptyBT),
             NodeBT(32,
               NodeBT(12, EmptyBT, EmptyBT),
               EmptyBT)),
           NodeBT(55,
             NodeBT(41,
               NodeBT(36, EmptyBT, EmptyBT),
               EmptyBT),
             NodeBT(97,
               NodeBT(92, EmptyBT, EmptyBT),
               EmptyBT)))

tKeys3Test = (tKeys3
              == insertListBST(keys3, makeEmptyBT()))
```

- Notice that tree tKeys2 has height equal to the number of items 11
- The structure might as well be a list
- In this case the tree structure would not be more efficient than a list for searching.
- The height of tKeys3 is 4 which is as compact a tree as we could get with the number of items between 8 and 15.
- tKeys2 shows inserting a list in even partly sorted order results in the worst case for efficiency.
- If a binary search tree is built from insertion of a list of random data then it can be shown that the expected height of the tree is $O(\log n)$

Go to Activity

- The proof of this requires knowledge of statistics outside the remit of this course — if interested, a proof is in Cormen et al. (2009, page 300)

Building a Compact BST

- To produce as compact a tree as possible, we could we could do the following:
- Sort the list
- Find the middle item in the sorted list
- Construct a binary tree node with the middle item as the data

- The left and right sub-trees should be formed by recursively building binary trees from the front and back parts of the sorted list
- Below is an implementation in Python

```python
def buildBST(xs):
  return bBST(makeEmptyBT(), sorted(xs))


def bBST(t,xs):
  if xs == []:
    return t
  else:
    half = len(xs) // 2
    x = xs[half]
    frontxs = xs[:half]
    backxs = xs[half+1:]
    if isEmptyBT(t):
      return (makeBT(x,
                bBST(EmptyBT(),frontxs),
                bBST(EmptyBT(),backxs)))
    else:
      errMsg = ("bBST:␣Trying␣to␣insert" + str(xs)
               + "␣into␣nonempty␣tree" + str(t))
      raise RuntimeError(errMsg)
```

4.2.2 Inserting a Node — Haskell

```haskell
  type BSTree a = BinTree a

  insertBST :: (Ord a) => a -> BSTree a -> BSTree a

  insertBST x EmptyBT
    = NodeBT x EmptyBT EmptyBT

  insertBST x (NodeBT y leftT rightT)
    | x < y   =  NodeBT y (insertBST x leftT) rightT
    | x > y   =  NodeBT y leftT (insertBST x rightT)
    | x == y  =  NodeBT y leftT rightT
```

Haskell Code Description 4

- The `type` declaration introduces a synonym for an existing type, for convenience (we still have to ensure our Binary Search trees have the Binary Search tree properties)
- The lines starting with `(|)` are *guarded* definitions — if the boolean expression to the right is `True` then the following expression is used
- The `(Ord a) =>` is the *context* for the type of `insertBST` which is asserting that type `a` must be a member of the `Ord` class (for ordering)

Binary Search Tree Operations — Haskell

```haskell
  isBSTree :: (Ord a) => BSTree a -> Bool

  isBSTree t = ((orderedList (<)) . inOrderBT) t

  orderedList comp []   =  True
  orderedList comp [x]  =  True
  orderedList comp (x:y:xs)
    | x `comp` y  =  orderedList comp (y:xs)
    | otherwise   =  False

  insertListBST :: (Ord a) =>
```

```
232                   [a] -> BSTree a -> BSTree a

234   insertListBST [] t = t
235   insertListBST (x:xs) t
236     =  insertListBST xs (insertBST x t)
```

- The ordering relation is `(<)` for strict ordering and no duplicates

Haskell Code Description 5

- In `` `comp` `` the grave accents `(`)` make a function into an infix operator (OK, that is syntactic sugar I need not have introduced — and my formatting program has coerced the grave accent to a left single quotation mark Unicode U+2018, not the grave accent U+0060)
- `otherwise` is a synonym for `True`

Building a BST — Haskell

```
238   buildBST :: (Ord a, Show a) => [a] -> BSTree a
239   buildBST xs = bBST EmptyBT  (sort xs)

241   bBST :: (Show a) => BSTree a -> [a] -> BSTree a
242   bBST t []  =  t
243   bBST EmptyBT  xs
244     = NodeBT x (bBST EmptyBT frontxs)
245                (bBST EmptyBT backxs)
246       where
247       x        =  xs !! half
248       frontxs  =  take half xs
249       backxs   =  drop (half + 1) xs
250       half     =  length xs `div` 2
251   bBST  t  xs
252     =  error ("bBST:␣Trying␣to␣insert␣"
253               ++ show xs ++ "␣into␣nonempty␣tree␣"
254               ++ show t)
```

Haskell Code Description 6

- `sort` is built-in from the `Data.List` library
- `!!` is the list indexing operator — first index is 0
- `div` is the built-in integer division function

4.3 Deleting a Node

- Deleting an item from a binary search tree involves more choices than insertion
- *Initial insight*
  - Find the node with the item (key) by following left or right sub trees
  - Delete the item by joining the two sub trees of the node
  - If the item is not in the tree, just return `EmptyBT`

- The tricky bit is deciding how to join the two sub trees while keeping the binary search tree property and keeping the tree compact (otherwise we lose the advantage of a binary search tree).
- The following presents three alternatives which each use some information about binary search trees — each version is correct but the later versions produce a more compact tree.
- Below is the initial insight implemented in Python and Haskell:

Deleting an Item from a BST — Python

```
259 def deleteBST(x, t):
260   if isEmptyBT(t):
261     return makeEmptyBT()
262   else:
263     y = t.dataBT
264     leftT = t.leftBT
265     rightT = t.rightBT
266     if x < y:
267       return makeBT(y, (deleteBST(x,leftT)), rightT)
268     elif x > y:
269       return makeBT(y, leftT, (deleteBST(x,rightT)))
270     else:
271       return joinBST(leftT, rightT)
```

Deleting an Item from a BST — Haskell

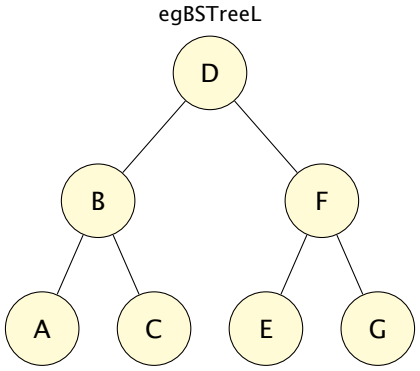
```
256   deleteBST :: (Ord a) => a -> BSTree a -> BSTree a

258   deleteBST x EmptyBT = EmptyBT

260   deleteBST x (NodeBT y leftT rightT)
261     | x < y    =  NodeBT y (deleteBST x leftT) rightT
262     | x > y    =  NodeBT y leftT (deleteBST x rightT)
263     | x == y   =  joinBST leftT rightT
```

Example Binary Search Tree Deletion

- We now investigate different ways of joining two subtrees with the function `joinBST(leftT, rightT)`
- We shall use the small tree `egBSTreeL` to illustrate the choices deleting the node with key `D`



- Here is a Python implementation of the tree `egBSTreeL`

```
42 egBSTreeL = NodeBT('D',
43               NodeBT('B',
44                 NodeBT('A',EmptyBT(),EmptyBT()),
45                 NodeBT('C',EmptyBT(),EmptyBT())
46               ),
47               NodeBT('F',
48                 NodeBT('E',EmptyBT(),EmptyBT()),
```

```
49                NodeBT('G',EmptyBT(),EmptyBT())
50              ))
```

- The Haskell implementation of the tree egBSTreeL is at line 33 on page 19

joinBST Version 1

- joinBST1 lists out all the elements of the left sub tree and inserts them in the right subtree.
- It does an in-order traversal of the left sub tree and then inserts the resulting list of items in the right subtree.

Python

```
275 def joinBST1(leftT, rightT):
276   if isEmptyBT(leftT):
277     return rightT
278   else:
279     return (insertListBST(inOrderBT(leftT), rightT))
```

Haskell

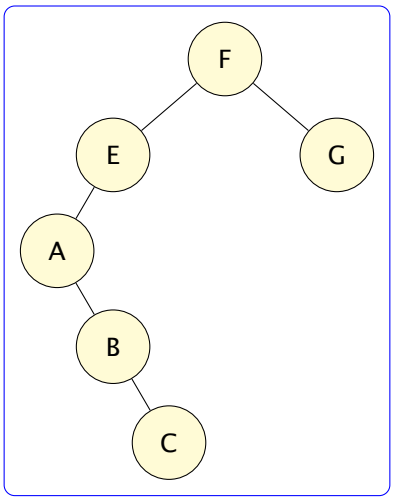
```
265   joinBST1 :: (Ord a) => BSTree a -> BSTree a
266                          -> BSTree a
267   joinBST1 EmptyBT rightT = rightT
268   joinBST1 (NodeBT x t1 t2) rightT
269     = insertListBST (inOrderBT (NodeBT x t1 t2)) rightT
```

Activity 9 joinBST Version 1

- Draw the diagram of the tree resulting fron deleting D with joinBST1
- Why is joinBST1 not a good strategy ?

Go to Answer

Answer 9 joinBST Version 1




```
  joinBST1 egBSTreeLL egBSTreeLR
    ==  NodeBT F (NodeBT E
          (NodeBT A EmptyBT
             (NodeBT B EmptyBT
                (NodeBT C EmptyBT EmptyBT))) EmptyBT)
        (NodeBT G EmptyBT EmptyBT)
```

Go to Activity

joinBST Version 2

- joinBST1 results in a near linear structure and is not as compact as it could be.
- The first definition made no use of our knowledge of binary search trees.
- We know that:

  ```
  maxKey leftT < minKey rightT
  ```

  since they were subtrees of the original Binary Search tree, egBSTreeL
- In particular we therefore know that the root of the left subtree is less than any item in the right subtree.
- So we attach the left subtree under the smallest element in the right sub tree.

Python

```
283 def joinBST2(leftT, rightT):
284   if isEmptyBT(rightT):
285     return leftT
286   else:
287     return (makeBT(rightT.dataBT,
288                    joinBST2(leftT, rightT.leftBT),
289                    rightT.rightBT))
```

Haskell

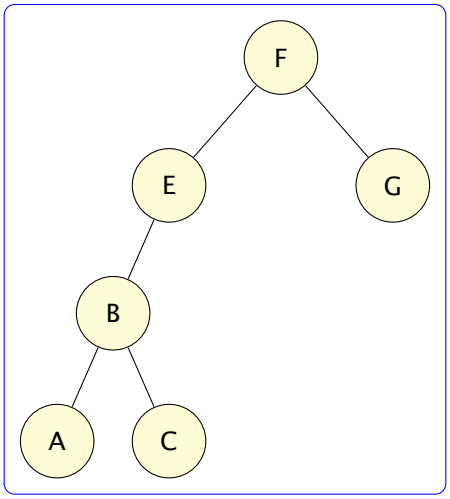
```
271   joinBST2 :: BinTree a -> BinTree a -> BinTree a
272   joinBST2 leftT EmptyBT = leftT
273   joinBST2 leftT (NodeBT x t1 t2)
274     =  makeBT x (joinBST2 leftT t1) t2
```

Activity 10 joinBST Version 2

- Draw the diagram of the tree resulting fron deleting D with joinBST2
- Can you see how we can improve on joinBST2 ?

Go to Answer

Answer 10 joinBST Version 2




```
delBSTreeL2 = NodeBT('F',
                NodeBT('E',
                  NodeBT('B',
                    NodeBT('A', EmptyBT(), EmptyBT()),
                    NodeBT('C', EmptyBT(), EmptyBT())),
                  EmptyBT()),
                NodeBT('G', EmptyBT(), EmptyBT))
```

```
delBSTreeL2Test = (delBSTreeL2
                   == deleteBST2('D', egBSTreeL))
```

- To see how this works we shall do a step by step evaluation
- Follow the function code for joinBST2 from line 283 on page 43
- **Step 1** In the first call to joinBST2 the leftT is the tree rooted at B and the rightT is the tree rooted at F
- Line 284 tests if the second argument to joinBST2 is an empty tree
- Since it is not empty, joinBST2 evaluates to the value at line 287
- Hence we have

```
makeBT('F',
       joinBST2(leftT, rightT.leftBT),
       rightT.rightBT)
```

- **Step 2** Since the return value has a recursive call to joinBST2 we need to evaluate that.
- Its second argument is rightT.leftBT which is the tree rooted at E
- Line 284 tests if the first argument to joinBST2 is an empty tree
- Since it is not empty, joinBST2 evaluates to the value at line 287
- Hence we have

```
makeBT('F',
       makeBT('E',
              joinBST2(leftT, rightT.leftBT.leftBT),
              rightT.leftBT.rightBT),
       rightT.rightBT)
```

- **Step 3** We have to evaluate a further recursive call
- The second argument to the recursive call to joinBST2 is rightT.leftBT.leftBT which is EmptyBT(), so the recursive call to joinBST2 evaluates to leftT
- rightT.leftBT.rightBT evaluates to EmptyBT()
- Hence we have

```
makeBT('F',
       makeBT('E',
              leftT,
              EmptyBT()),
       rightT.rightBT)
```

- Hence the final value is

```
NodeBT('F',
  NodeBT('E',
    NodeBT('B',
      NodeBT('A', EmptyBT(), EmptyBT()),
      NodeBT('C', EmptyBT(), EmptyBT())),
    EmptyBT()),
  NodeBT('G', EmptyBT(), EmptyBT()))
```

- Doing a step by step evaluation of recursive function calls should help you get a better feel for recursive thinking.
- We can do better than this — see the following.

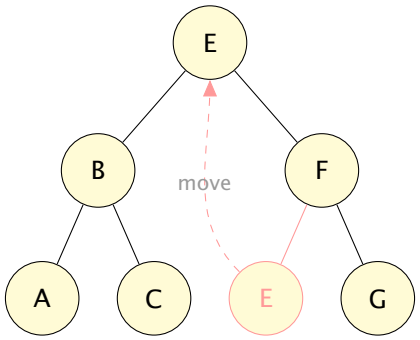
Go to Activity

joinBST Final Version

- We can make better use of our knowledge of Binary Search trees
- We know that:

```
maxKey leftT < root key < minKey rightT
```

- Hence we can promote the minimum item in the right subtree to be the new root and delete it from its original position.
- **Note** we could equally well promote the maximum item in the left subtree to be the new root (and delete it from its original position).



- Here is Python and Haskell code for the above diagram:

```python
def joinBST(leftT, rightT):
  if isEmptyBT(rightT):
    return leftT
  else:
    (y,t) = splitBST(rightT)
    return makeBT(y, leftT, t)
```

```haskell
joinBST :: BinTree a -> BinTree a -> BinTree a
joinBST leftT EmptyBT = leftT
joinBST leftT (NodeBT x t1 t2)
  =  NodeBT y leftT t
     where
     (y, t) = splitBST (NodeBT x t1 t2)
```

- splitBST will take the right subtree and return a pair of minimum item and the subtree with that item removed.
- This preserves much of the compact nature of the binary search tree.

splitBST Version 1

- We still have choices in defining splitBST
- We could define splitBST by finding the minimum item and then deleting that from the subtree.

```python
def splitBST1(t):
  if isEmptyBT(t):
    raise RuntimeError("splitBST1 applied to EmptyBT()")
  elif isEmptyBT(t.leftBT):
    return (t.dataBT, t.rightBT)
  else:
    y = minItemBST(t.leftBT)
    return (y, makeBT(t.dataBT,
                      deleteBST(y, t.leftBT),
                      t.rightBT))
```

```
314 def minItemBST(t):
315   if isEmptyBT(t):
316     raise RuntimeError("minItemBST␣applied␣to␣EmptyBT()")
317   elif isEmptyBT(t.leftBT):
318     return (t.datatBT)
319   else:
320     return (minItemBST(t.leftBT))
```

splitBST Version 1 — Haskell

- We still have choices in defining splitBST
- We could define splitBST by finding the minimum item and then deleting that from the subtree.

```
283 splitBST1 :: Ord a => BinTree a -> (a, BinTree a)
284 splitBST1 (NodeBT x EmptyBT t2) = (x, t2)
285 splitBST1 (NodeBT x t1 t2)
286   = (y, NodeBT x (deleteBST y t1) t2)
287     where
288     y = minItemBST t1

290 minItemBST :: BinTree a -> a
291 minItemBST (NodeBT x EmptyBT t2) = x
292 minItemBST (NodeBT x t1 t2)
293   =  minItemBST t1
```

splitBST Final Version

- We can do better than splitBST1
- It is possible to define splitBST using only one traversal of the tree.

```
324 def splitBST(t):
325   if isEmptyBT(t):
326     raise RuntimeError("splitBST␣applied␣to␣EmptyBT()")
327   else:
328     x = t.dataBT
329     t1 = t.leftBT
330     t2 = t.rightBT
331     if isEmptyBT(t1):
332       return (x,t2)
333     else:
334       (y,t3) = splitBST(t1)
335       return (y, makeBT(x, t3, t2))
```

splitBST Final Version — Haskell

- We can do better than splitBST1
- It is possible to define splitBST using only one traversal of the tree.

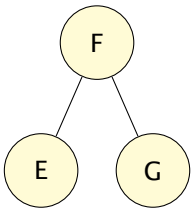
```
295 splitBST :: BinTree a -> (a, BinTree a)
296 splitBST (NodeBT x EmptyBT t2) = (x, t2)
297 splitBST (NodeBT x t1 t2)
298   = (y, NodeBT x t3 t2)
299     where
300     (y, t3) = splitBST t1

302 splitBST EmptyBT
303   = error "Attempt␣to␣apply␣splitBST␣to␣EmptyTree"
```

Activity 11 Split Trace

- Trace an evaluation of `splitBST(egBSTreeLR)`

splitBST(egBSTreeLR)




```
57 egBSTreeLR = NodeBT('F',
58                NodeBT('E',EmptyBT(),EmptyBT()),
59                NodeBT('G',EmptyBT(),EmptyBT())
60              )
```

- egBSTreeLR is defined at line 57 on page 47, splitBST is defined at line 324 on page 46

Go to Answer

Answer 11 Split Trace

- Evaluation of `splitBST(egBSTreeLR)`

Step 1

egBSTreeLR.leftBT is not empty so the else clause at line 333 is executed

Step 2

A recursive call is made to splitBST with argument egBSTreeLR.leftBT

egBSTreeLR.leftBT.leftBT is empty so the if at line 331 returns

('E',egBSTreeLR.leftBT.rightBT) which is ('E',EmptyBT())

Answer 11 Split Trace

Step 3

The calling function then returns

('E', makeBT('F', EmptyBT(), egBSTreeLR.rightBT))

```
('E',NodeBT('F',
       EmptyBT(),
       NodeBT('G',EmptyBT(),EmptyBT())))
```

Go to Activity

Activity 12 Join Trace

- Trace an evaluation of

`joinBST(egBSTreeLL,egBSTreeLR)`

- egBSTreeLL is defined at line 52 on page 15, joinBST is defined at line 293 on page 45

Go to Answer

Answer 12 Join Trace

- Evaluation of `joinBST(egBSTreeLL,egBSTreeLR)`

Step 1

The second argument to `joinBST` is not the empty tree so the `else` clause at line 296 — this invokes `splitBST(egBSTreeLR)`

Step 2

From the previous activity, `splitBST(egBSTreeLR)` returns

```
('E', makeBT('F', EmptyBT(), egBSTreeLR.rightBT))
```

Answer 12 Join Trace

Step 3

Finally, the `return` statement returns

```
NodeBT('E',
  NodeBT('B',
    NodeBT('A', EmptyBT(), EmptyBT()),
    NodeBT('C', EmptyBT(), EmptyBT())),
  NodeBT('F',
    EmptyBT(),
    NodeBT('G', EmptyBT(), EmptyBT())))
```

Go to Activity

Activity 13 Delete Trace

- Trace an evaluation of

  ```
  deleteBST('D',egBSTreeL)
  ```

- `egBSTreeL` is defined at line 42 on page 41, `deleteBST` is defined at line 259 on page 41

Go to Answer

Answer 13 Delete Trace

- Evaluation of `deleteBST('D',egBSTreeL)`

Step 1

The second argument of `deleteBST` is not the empty tree so the `else` clause at line 262 is executed.

Step 2

The first argument of `deleteBST` is `'D'` which is equal to the item at the root of the tree which is the second argument, so the `else` clause at line 270 is executed

Step 3

This evaluates to `joinBST(egBSTreeLL,egBSTreeLR)` — see previous activity

Go to Activity

- As we noted earlier, on average the height of a binary search tree is $O(\log n)$ where n is the number of nodes in the tree.
- However in the worst case the height is $O(n)$ and this will affect the performance of searches.

- However it is possible to construct variants of binary search trees which have $O(\log n)$ performance in both average and worst cases
- In the next section we will consider one approach.

5 Height Balanced Trees

- Binary search trees have the problem that in the worst case the complexity of a search could be $O(n)$ and maintaining a complete tree during insertions and deletions involves too much restructuring.
- A solution is to keep the tree *balanced* so that access time is still $O(\log n)$ in both the average and worst cases.
- The essential approach is to have some *local* transformations involving only a few nodes to keep the tree *height balanced*.
- We shall consider one approach called *AVL Trees*, named after the Russian inventors G M Adelson-Velskii and E M Landis (Adelson-Velskii and Landis, 1962).
- *AVL Trees* are Binary Search Trees with the property that for every subtree the heights of the trees differs by at most 1 (the *balance factor*).
- AVL trees require an extra couple of functions to maintain the AVL property on each insertion or deletion.

5.1 AVL Trees and Functions

As with the Binary Search Tree, we shall use a named tuple to represent the data type for an AVL Tree. We could have taken an Object Oriented approach and defined AVL trees as a sub class of a Binary Search Tree class but that would not have saved much effort and would provide extra code to understand.

Here is the Python code:

```
344 EmptyABT = namedtuple('EmptyABT',[])

346 NodeABT = (namedtuple('NodeABT',
347             ['heightABT','dataABT','leftABT','rightABT']))

349 def isEmptyABT(t):
350   return t == EmptyABT()

352 def heightABT(t):
353   if isEmptyABT(t):
354     return 0
355   else:
356     return t.heightABT

358 def makeABTree(x,t1,t2):
359   h = 1 + max(heightABT(t1),heightABT(t2))
360   return NodeABT(h,x,t1,t2)
```

- Note that we store the height of a tree in the node
- This is essential to avoid lots of tree traversals to re-calculate balance factors

AVL Tree Data Type — Haskell

```haskell
data ABinTree a
  = EmptyABT
    | NodeABT Height a (ABinTree a) (ABinTree a)
    deriving (Eq, Show, Read)
type Height = Int

heightABT :: ABinTree a -> Height
heightABT EmptyABT  =  0
heightABT (NodeABT h x leftABT rightABT)  =  h

makeABTree :: a -> ABinTree a -> ABinTree a
               -> ABinTree a
makeABTree x leftABT rightABT
  = NodeABT h x leftABT rightABT
    where
    h = 1 + max (heightABT leftABT)
                (heightABT rightABT)
```

- Note that we store the height of a tree in the node
- This is essential to avoid lots of tree traversals to re-calculate balance factors

AVL Property Functions 1 — Python

```python
def isBSABTree(t):
  return orderedList(inOrderABT(t))

def inOrderABT(t):
  if isEmptyABT(t):
    return []
  else:
    return (inOrderABT(t.leftABT) + [t.dataABT]
            + inOrderABT(t.rightABT))
```

AVL Property Functions 1 — Haskell

```haskell
isBSABTree :: (Ord a) => ABinTree a -> Bool
isBSABTree t = ((orderedList (<)) . inOrderABT) t

inOrderABT :: ABinTree a -> [a]
inOrderABT EmptyABT  =  []
inOrderABT (NodeABT h x leftT rightT)
  = (inOrderABT leftT) ++ [x] ++ (inOrderABT rightT)
```

- The ordering relation is (<) for strict ordering and no duplicates

AVL Property Functions 2 — Python

```python
def balFactorABT(t):
  if isEmptyABT(t):
    return 0
  else:
    return heightABT(t.leftABT) - heightABT(t.rightABT)
```

```python
def hasAVLpropABT(t):
  if isEmptyABT(t):
    return True
  else:
    return (abs(balFactorABT(t)) <= 1
            and hasAVLpropABT(t.leftABT)
            and hasAVLpropABT(t.rightABT))
```

- The abs function is the Python built-in function which returns the absolute value of a number.

```python
def isAVLABTree(t):
  return (isBSABTree(t) and hasAVLpropABT(t))
```

AVL Property Functions 2 — Haskell

```haskell
balFactorABT :: ABinTree a -> Int
balFactorABT EmptyABT  =  0
balFactorABT (NodeABT h x leftT rightT)
  = (heightABT leftT) - (heightABT rightT)

hasAVLpropABT :: ABinTree a -> Bool
hasAVLpropABT EmptyABT  =  True
hasAVLpropABT (NodeABT h x leftT rightT)
  = abs (balFactorABT (NodeABT h x leftT rightT))
        <= 1
    && hasAVLpropABT leftT  &&  hasAVLpropABT rightT

isAVLABTree :: (Ord a) => ABinTree a -> Bool
isAVLABTree  t
  = isBSABTree t && hasAVLpropABT t
```

- && is the logical *and* operator
- || is the logical *or* operator

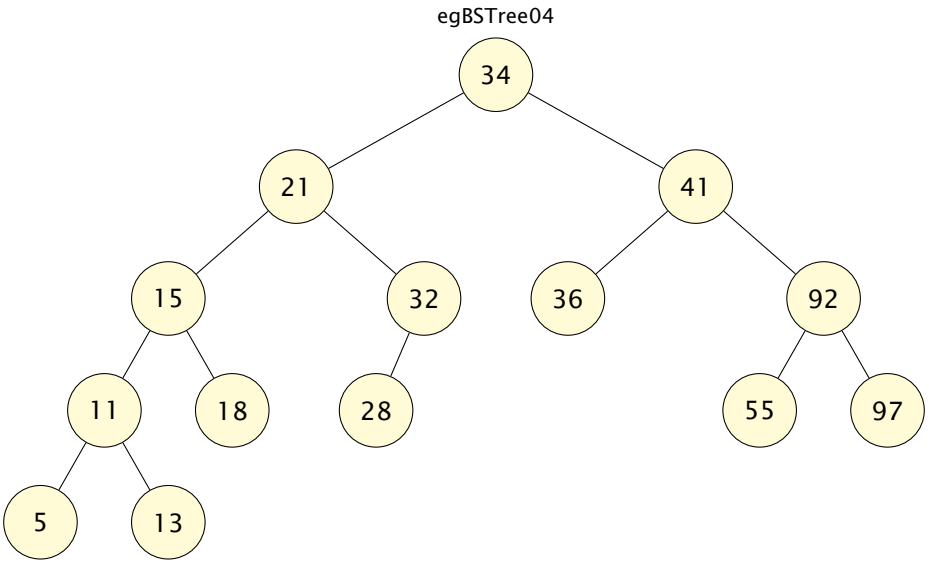
Health Warning 2

- Some texts (including Miller & Ranum) define the height of a singleton node to be zero — just subtract one from the height as defined here.
- Some texts do not use empty trees — so where these notes might say a singleton nodes has an element and two empty subtrees, some texts might say a singleton node has no subtrees
- Some texts define the height of a subtree differently to the height of a tree or define a subtree differently to here.
- Some texts define the balance factor as the absolute value or the height of the right sub tree minus the height of the left sub tree
- In all cases be aware that you have choices in the exact definition of some terms but the ideas will be the same.

5.2 Example AVL Trees

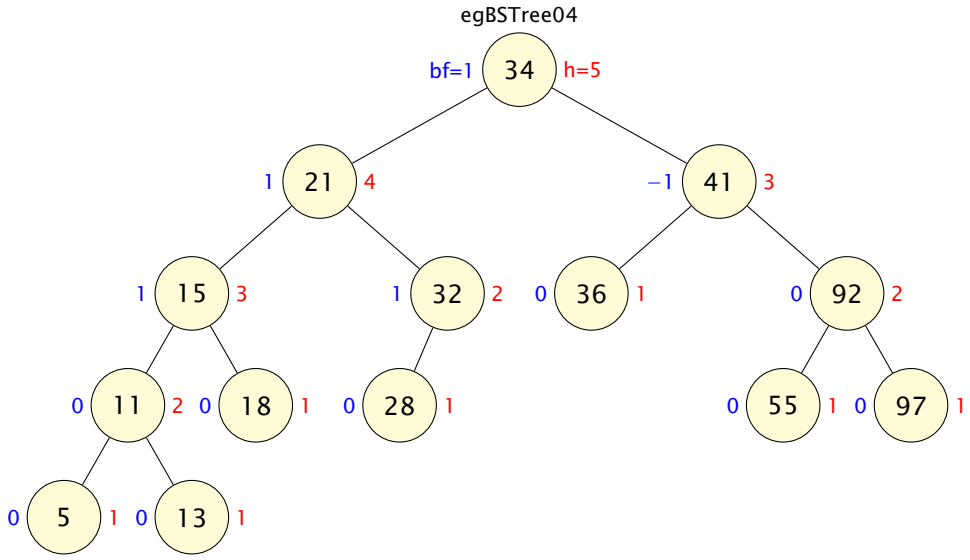
Activity 14 Height and Balance Factor

- For the following diagram of a binary search tree, egBSTree04, add the height and balance factor for each node.



Go to Answer

Answer 14 Height and Balance Factor



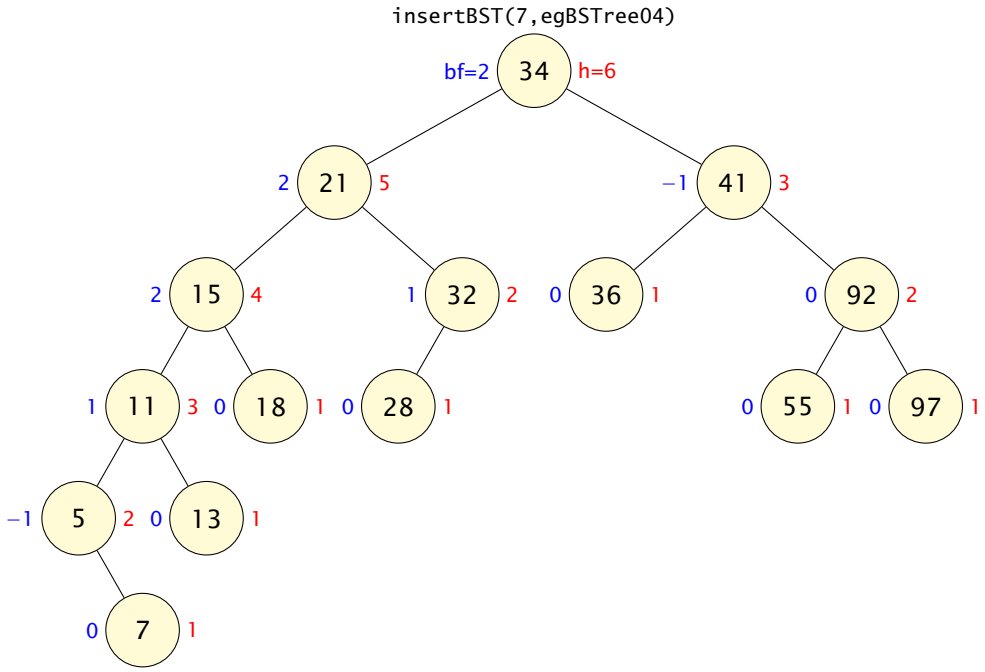
Go to Activity

Activity 15 Add Item LL

- Add the item with key 7 to the tree, egNSTree04, and recalculate the heights and balance factors
- Identify the lowest node in the tree which is out of balance.

Go to Answer

Answer 15 Add Item LL



- The lowest node which is out of balance is the the node with key 15

Go to Activity

AVL Trees — Service Functions — Python

```
398 def convertBTtoABT(t):
399   if isEmptyBT(t):
400     return EmptyABT()
401   else:
402     leftABT = convertBTtoABT(t.leftBT)
403     rightABT = convertBTtoABT(t.rightBT)
404     return makeABTree(t.dataBT, leftABT, rightABT)
```

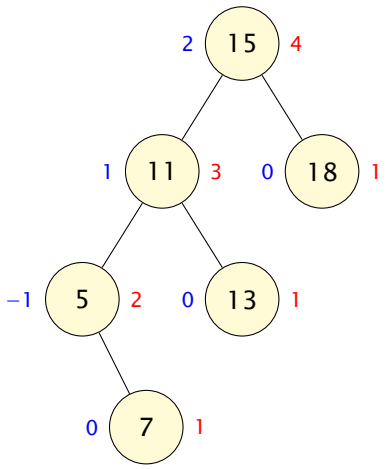
AVL Trees — Service Functions — Haskell

```
347 convertBTtoABT :: BinTree a -> ABinTree a
348 convertBTtoABT EmptyBT = EmptyABT
349 convertBTtoABT (NodeBT x leftT rightT)
350   = makeABTree x leftABT rightABT
351     where
352     leftABT  = convertBTtoABT leftT
353     rightABT = convertBTtoABT rightT
```

5.3 AVL Tree Transformations

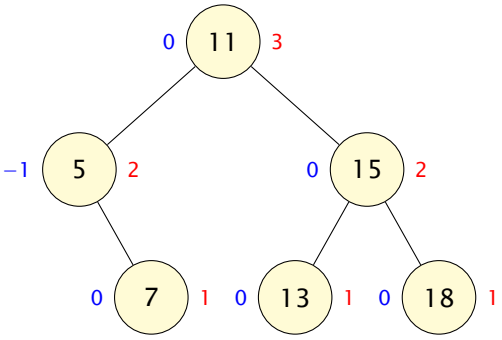
- Since the subtree of `insertBST(7,egBSTree04)` at node with key 15 is the part of the tree out of balance we shall focus on that

```
NodeABT(4, 15,
  NodeABT(3, 11,
    NodeABT(2, 5,
      EmptyABT(),
      NodeABT(1, 7, EmptyABT(), EmptyABT())),
    NodeABT(1, 13, EmptyABT(), EmptyABT())),
  NodeABT(1, 18, EmptyABT(), EmptyABT()))
```



- We can make this tree balanced by
- Make the subtree with root 15 the right child of 11
- Make the subtree with root 13 the left child of 15
- Leave the subtree with root 5 as the left child of 11
- Make the new subtree with root 11 the child of wherever the original subtree with root 15 was (the left child of 21)
- This results in the following tree.

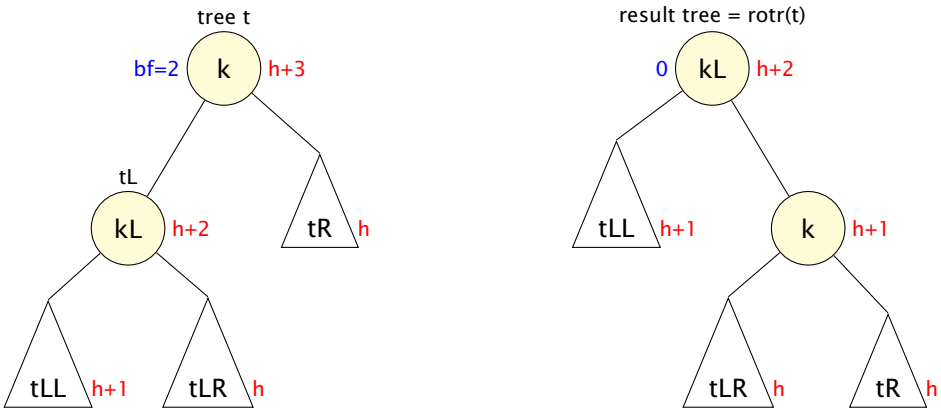
```
NodeABT(3, 11,
  NodeABT(2, 5,
    EmptyABT(),
    NodeABT(1, 7, EmptyABT(), EmptyABT())),
  NodeABT(2, 15,
    NodeABT(1, 13, EmptyABT(), EmptyABT()),
    NodeABT(1, 18, EmptyABT(), EmptyABT())))
```



- This transformation is an instance of what is called a *right rotation*
- Here is Python code that implements it.

```
412 def rotr(t):
413   k = t.dataABT
414   kL = t.leftABT.dataABT
415   leftLeftT = t.leftABT.leftABT
416   leftRightT = t.leftABT.rightABT
417   rightT = t.rightABT
418   return (makeABTree(kL,
419                      leftLeftT,
420                      makeABTree(k, leftRightT, rightT)))
```

Case LL Heavy — Right Rotation

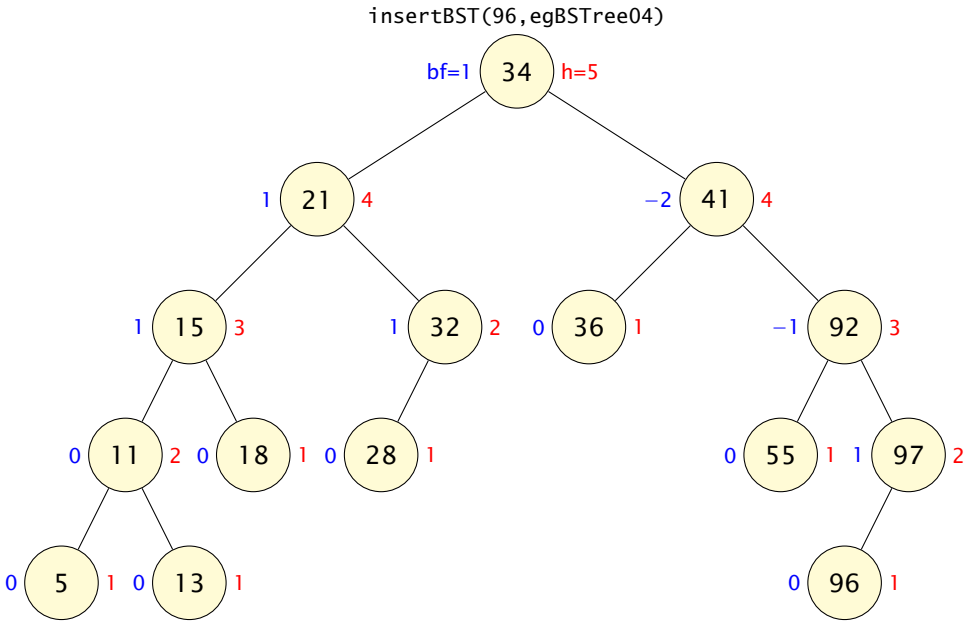


Activity 16 Add Item RR

- Consider egBSTree04 again (defined in Activity 14 on page 51) — now add node with key 96 and recalculate the heights and balance factors

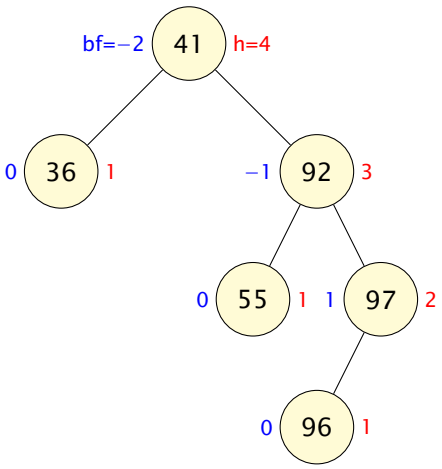
Go to Answer

Answer 16 Add Item RR



Go to Activity

- The subtree at node 41 is now unbalanced with the addition of the node with key 96 to the right subtree of the right subtree.



Activity 17 Rebalance RR

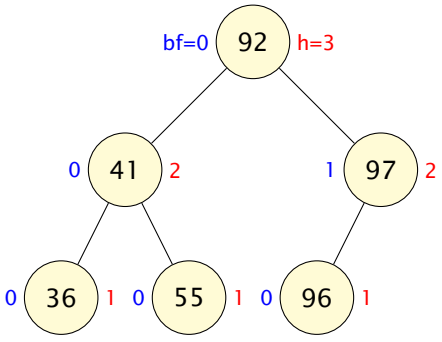
- This is similar to the previous example but on the right side
- Describe how this can be rebalanced using a mirror image local transformation.

Go to Answer

Answer 17 Rebalance RR

- We can make this tree balanced by:
- Make the subtree with root 41 the left child of 92
- Make the subtree with root 55 the right child of 41
- Leave the subtree with root 97 as the right child of 92
- Make the new subtree with root 92 the child of wherever the original subtree with root 41 was (the right child of 34)

```
NodeABT(3, 92,
  NodeABT(2, 41,
    NodeABT(1, 36, EmptyABT(), EmptyABT()),
    NodeABT(1, 55, EmptyABT(), EmptyABT())),
  NodeABT(2, 97,
    NodeABT(1, 96, EmptyABT(), EmptyABT()),
    EmptyABT()))
```



- The transformation is called a *left rotation*

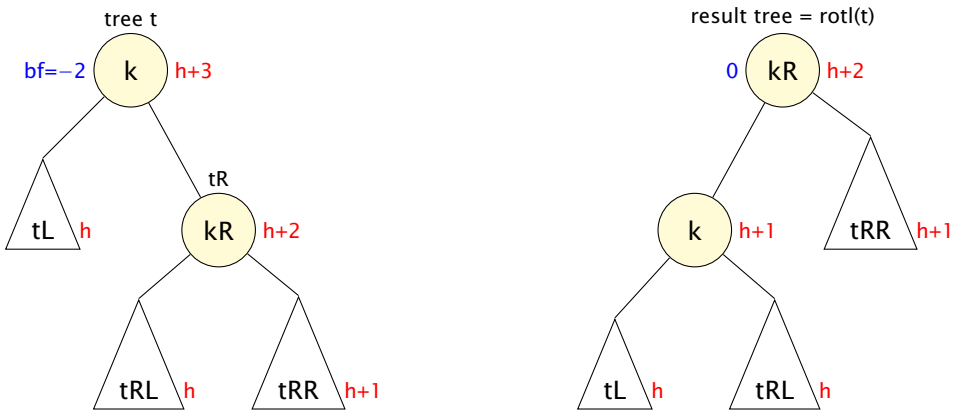
Go to Activity

- The transformation (given in the answer) is an instance of what is called a *left rotation*
- Here is the Python code that implements it.

```
422 def rotl(t):
423   k = t.dataABT
424   kR = t.rightABT.dataABT
425   rightLeftT = t.rightABT.leftABT
426   rightRightT = t.rightABT.rightABT
427   leftT = t.leftABT
428   return (makeABTree(kR,
429                      makeABTree(k, leftT, rightLeftT),
430                      rightRightT))
```

- This is a mirror image of the *right rotation* as you can see from the two diagrams describing it below.

Case RR Heavy — Left Rotation



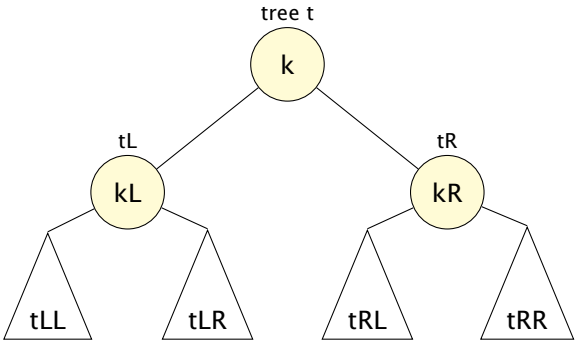

```
355   rotr :: ABinTree a -> ABinTree a
356   rotr (NodeABT h k
357           (NodeABT hL kL leftLeftT leftRightT) rightT)
358     = makeABTree kL
359                  leftLeftT
360                  (makeABTree k leftRightT rightT)

362   rotl :: ABinTree a -> ABinTree a
363   rotl (NodeABT h k
364           leftT (NodeABT hR kR rightLeftT rightRightT))
365     =  makeABTree kR
366                   (makeABTree k leftT rightLeftT)
367                   rightRightT
```

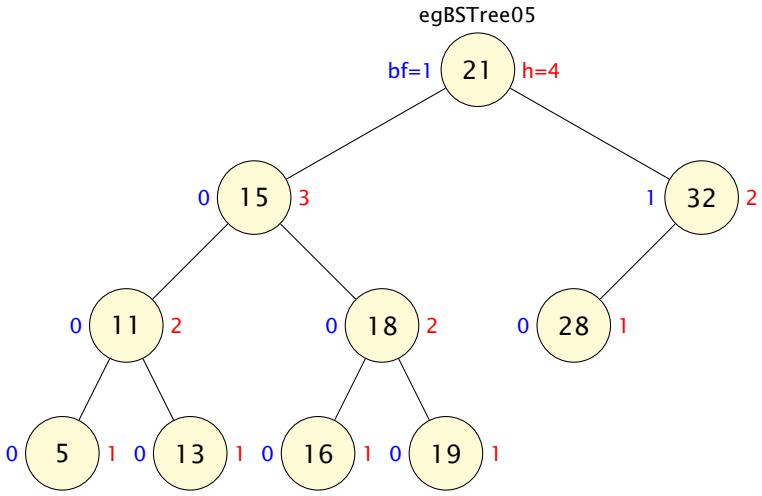
- The definitions assume that `rotr` and `rotl` are only called on valid trees

AVL Tree Transformations — Insight

- The functions for insertion and deletion of an item in an AVL tree will be the same as a Binary Search tree excep
- When we construct a new tree we must maintain the AVL property via a function `makeAVLTree` (line 490 on page 64) not just `makeABTree` (line 358 on page 49). (some texts call this rebalancing or something similar)
- What we know is that the original tree must be a properly formed AVL tree and that the insertion or deletion of one item can alter the height of any subtree by at most 1.
- Hence we can implement `makeAVLTree(x, leftT, rightT)` assuming that `leftT` and `rightT` are both AVL trees whose heights differ by at most 2.
- We proceed by analysing each possible case and provide a manipulation of the tree for each case. Consider the diagram below:



- Our right and left rotation functions, `rotr` and `rotl` have dealt with the cases where the subsubtrees LL and RR had increased by one caused the balance to go outside the permitted range.
- We now have to investigate cases where the subsubtrees LR or RL become heavy.
- Below is an example, egBSTree05

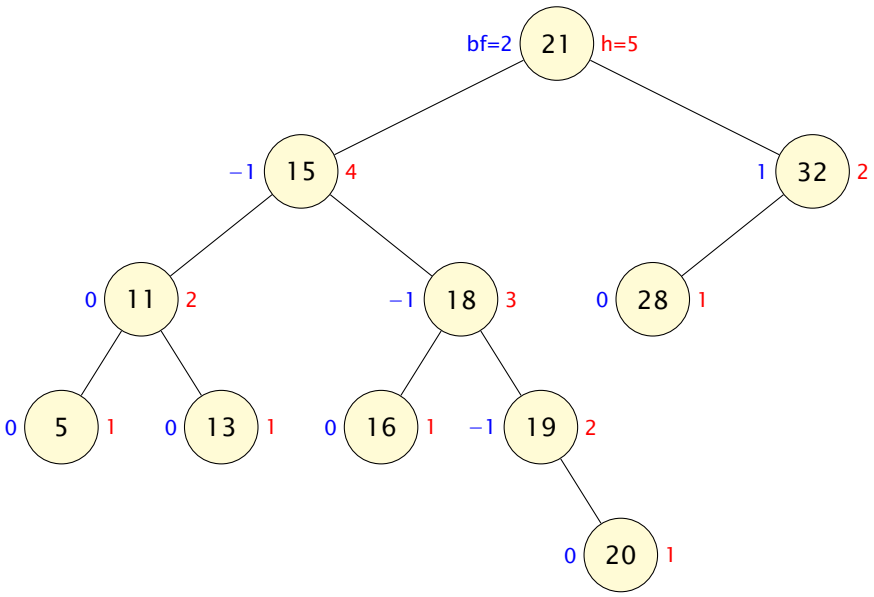


Activity 18 egBSTree05 Add Item LR 1

- Add the item with key 20 to the tree and recalculate the heights and balance factors
- Identify the lowest node in the tree which is out of balance.

Go to Answer

Answer 18 egBSTree05 Add Item LR 1



- The node with key 21 has balance factor 2 and is the lowest node out of balance.

Go to Activity

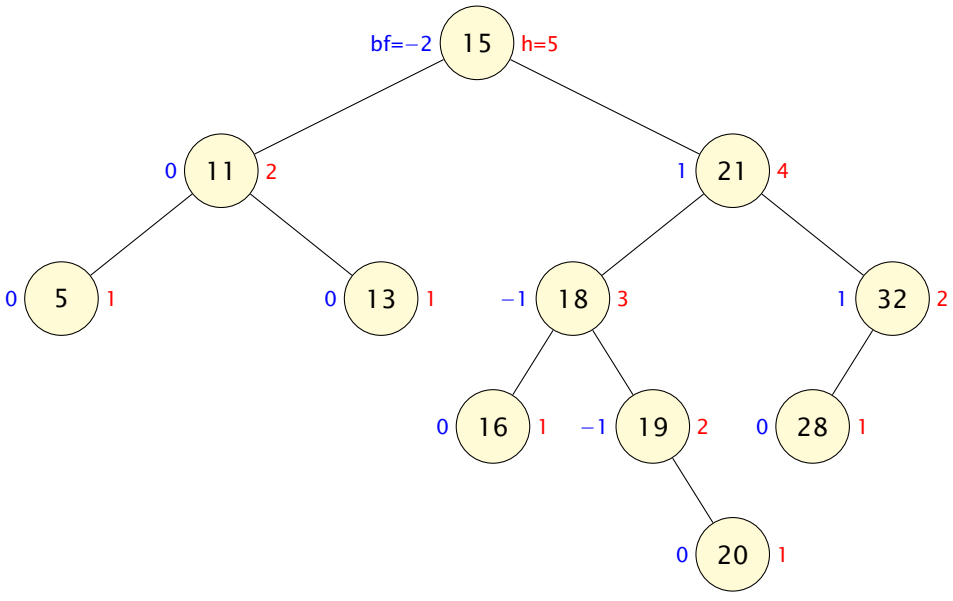
Activity 19 Add Item LR 2

- Given the resulting tree from Self-assessment activity 18, does a right rotation around the lowest node which is out of balance bring it back to balance ?

Go to Answer

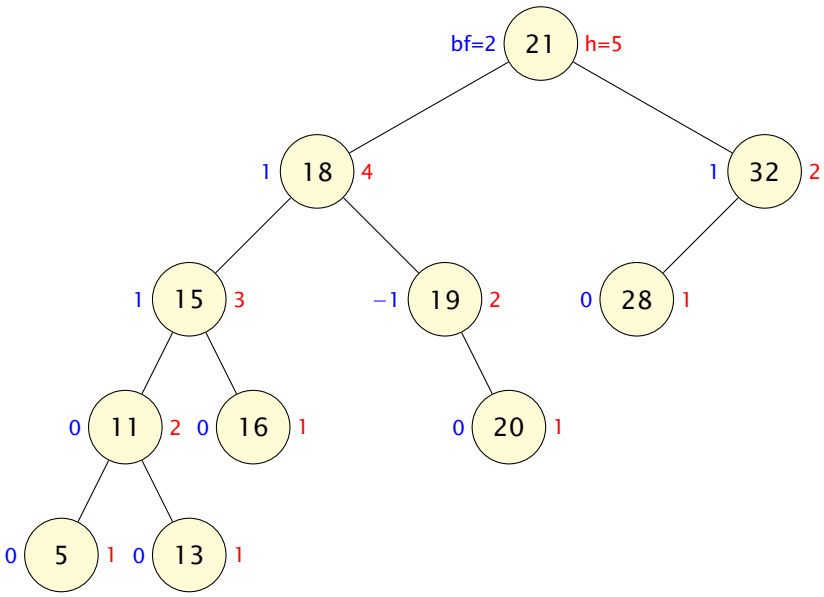
Answer 19 Add Item LR 2

- Here is the result of a right rotation around node with key 21

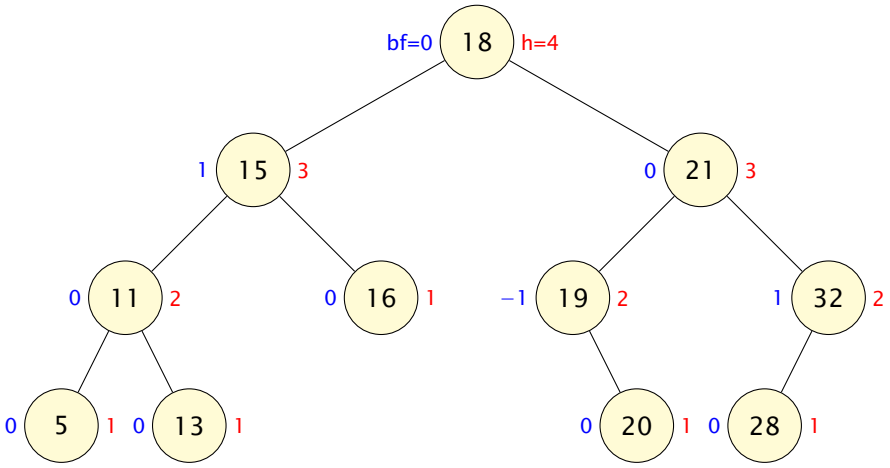


Go to Activity

- This has just switched the balance factor of the root of the tree from 2 to −2 so we have to do something else.
- The *Eureka* step is realising that we can break up the problematic subtree under node 18 by doing a left rotation around node 15 — this produces the following tree.

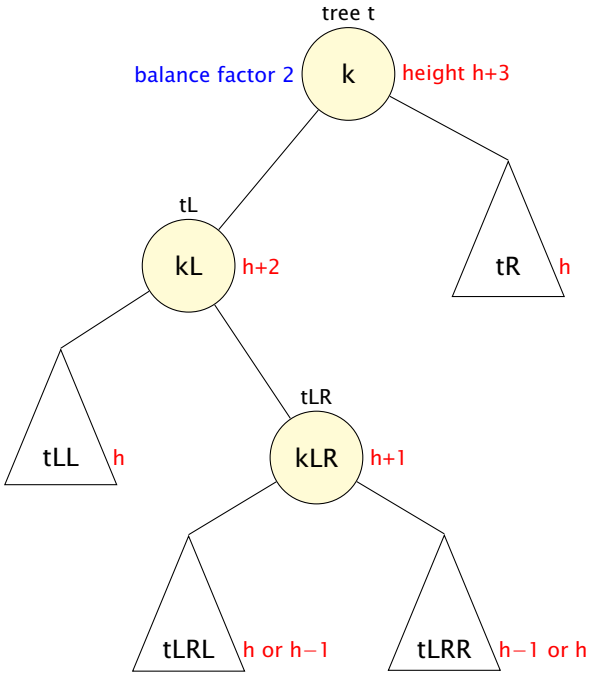


- Notice that this has converted a tree which was LR heavy to one where it is LL heavy — so we can now use a right rotation on the tree rooted at 21 to get the following:

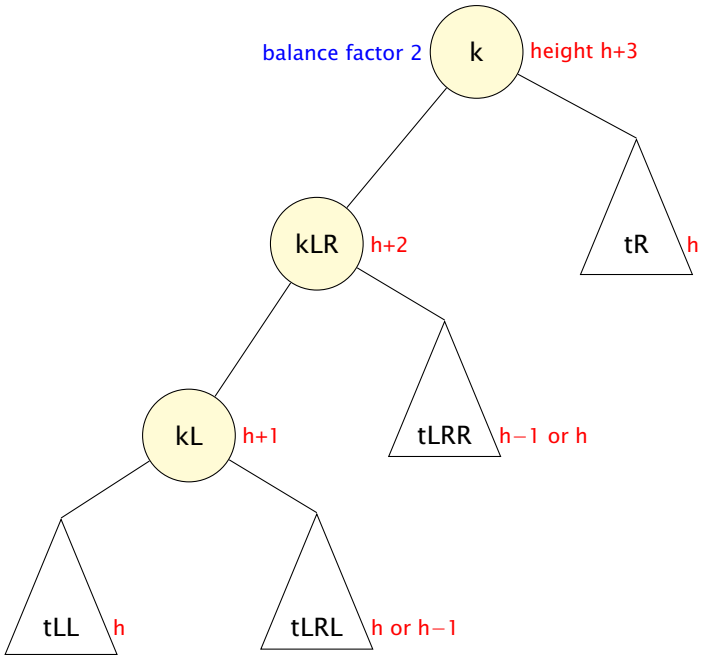


- We now have a balanced tree — but were we just lucky or have we found a general rule ? Here are diagrams of the double rotation to show it works in general:

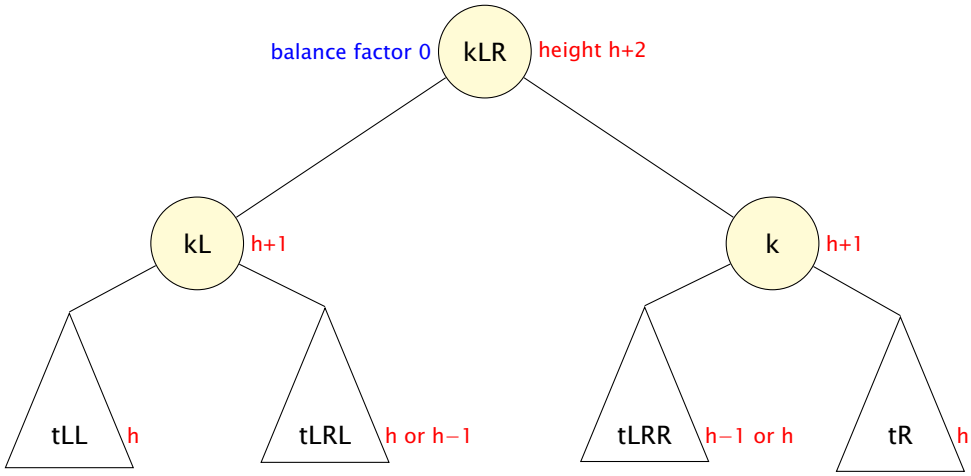
Case LR subsubtree heavy



Case LR — Step (A) Rotate Left about kL



Case LR — Step (B) Rotate Right about k

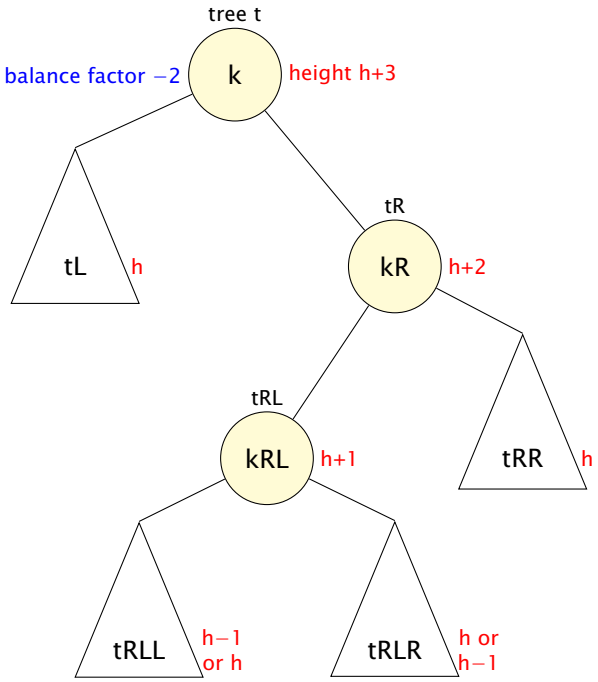


Activity 20 Case RL Heavy

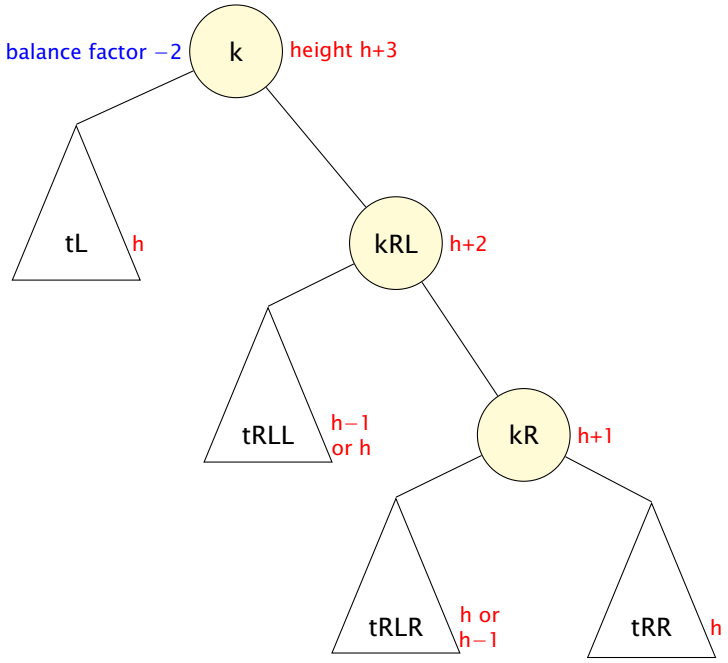
- Draw the equivalent diagram for the final case where subsubtree tRL is heavy
- Note that this must be the mirror image of the tLR case

Go to Answer

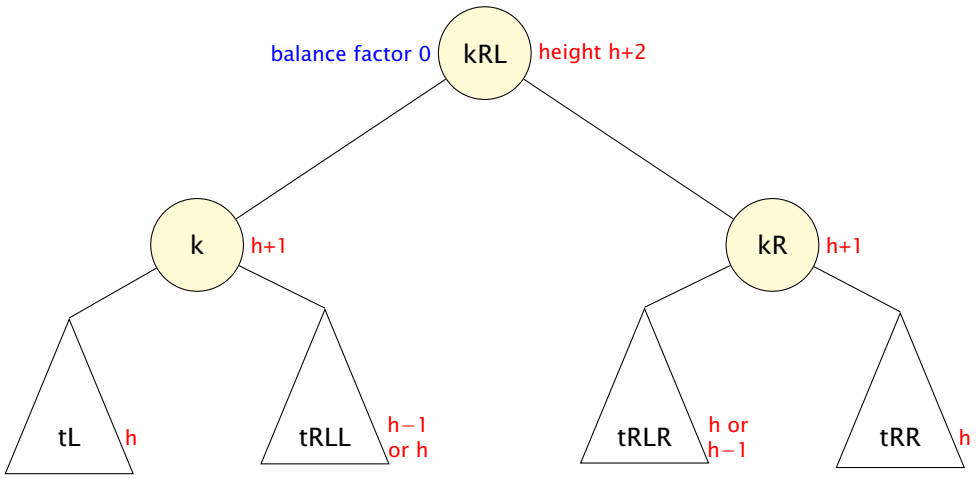
Answer 20 Case RL Heavy



Answer 20 Case RL Heavy — Step (A) Rotate Right about kR



Answer 20 Case RL Heavy — Step (B) Rotate Left about k



Go to Activity

5.4 AVL Trees — The makeAVLTree Function

- The `makeAVLTree` takes an item, `x`, two subtrees, `leftT`, `rightT` and returns a new augmented binary tree
- It would be the same as `makeABTree` except it has to do the appropriate transformation if the new tree would be out of balance.
- We will only ever use `makeAVLTree` when inserting or deleting an item in a valid AVL tree
- So we know from our insight above that the heights of `leftT` and `rightT` can differ by at most 2 after insertion/deletion
- Hence we consider each case in turn using the transformations we have developed above.

Case 1 LL Heavy

```
(heightABT(leftL) - heightABT(rightT) = 2
 and balFactorABT(leftT) >= 0)
```

- Do a right rotation of the tree formed from `makeABTree(x, leftT, rightT)`

Case 2 LR Heavy

```
(heightABT(leftL) - heightABT(rightT) = 2
 and balFactorABT(leftT) == -1)
```

- Do a left rotation of `leftT`
- Do a right rotation of the tree formed from `makeABTree(x, rotl(leftT), rightT)`

Case 3 RL Heavy

```
(heightABT(leftL) - heightABT(rightT) = -2
 and balFactorABT(rightT) == 1)
```

- Do a right rotation of `rightT`
- Do a left rotation of the tree formed from `makeABTree(x, leftT, rotr(rightT))`

Case 4 RR Heavy

```
  (heightABT(leftL) - heightABT(rightT) = -2
   and balFactorABT(rightT) <= 0)
```

- Do a left rotation of the tree formed from `makeABTree(x, leftT, rightT)`

Case 5 Otherwise

- Just use `makeABTree(x, leftT, rightT)`
- No transformations required

`makeAVLTree` Function — Python

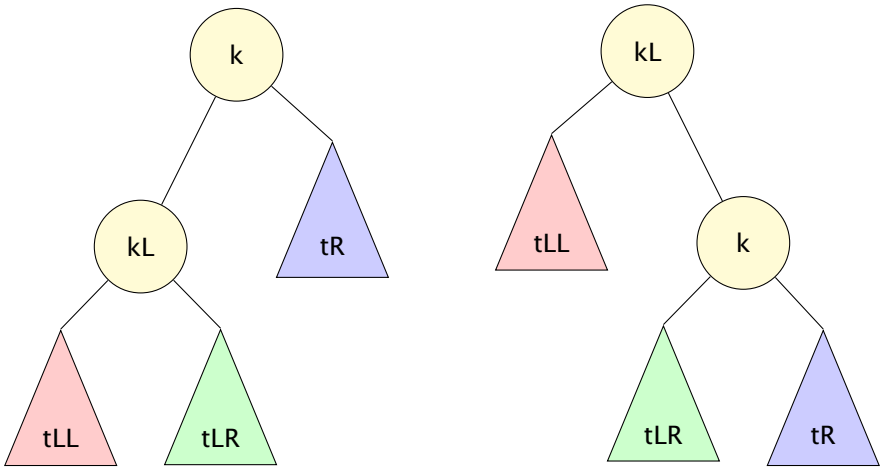
```
490 def makeAVLTree(x, leftT, rightT):
491   hL = heightABT(leftT)
492   hR = heightABT(rightT)
493   if (hR + 1 < hL) and (balFactorABT(leftT) >= 0):
494     return rotr(makeABTree(x, leftT, rightT))
495   elif (hR + 1 < hL):
496     return rotr(makeABTree(x, (rotl(leftT)),rightT))
497   elif (hL + 1 < hR) and (balFactorABT(rightT) > 0):
498     return rotl(makeABTree(x, leftT, rotr(rightT)))
499   elif (hL + 1 < hR):
500     return rotl(makeABTree(x, leftT, rightT))
501   else:
502     return makeABTree(x, leftT, rightT)
```

`makeAVLTree` Function — Haskell

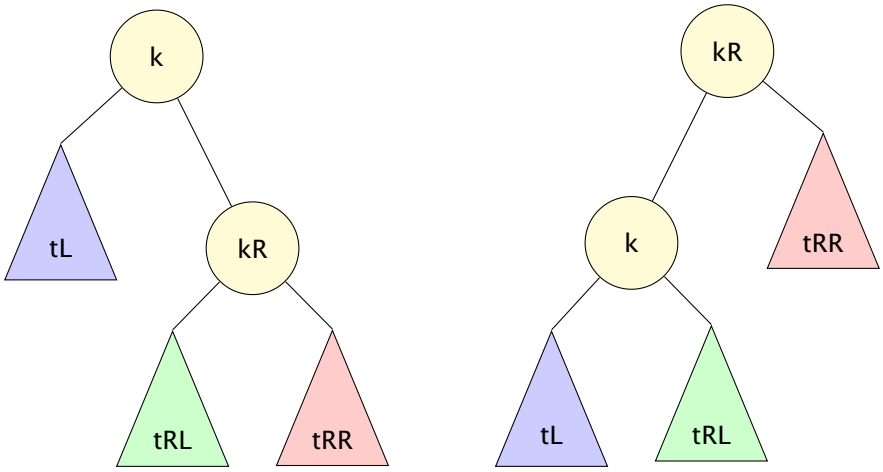
```
369 makeAVLTree :: a -> ABinTree a -> ABinTree a
370                 -> ABinTree a
371 makeAVLTree x leftT rightT
372   | hR + 1 < hL && balFactorABT leftT >= 0
373     =  rotr (makeABTree x leftT rightT)
374
375   | hR + 1 < hL
376     =  rotr (makeABTree x (rotl leftT) rightT)
377
378   | hL + 1 < hR && balFactorABT rightT > 0
379     =  rotl (makeABTree x leftT (rotr rightT))
380
381   | hL + 1 < hR
382     =  rotl (makeABTree x leftT rightT)
383
384   | otherwise
385     = makeABTree x leftT rightT
386     where
387     hL = heightABT leftT
388     hR = heightABT rightT
```

- This section has been quite long but most of the space has been occupied with diagrams
- Some *implementations* can look quite tricky since they may be trying to avoid recursion or manipulate the data structures
- We will discuss efficiency and recursion removal in a later section.
- Here are diagrams of the two rotate functions to emphasise that they are really quite simple.

Rotate Right



Rotate Left



5.4.1 Comparison with Storing Balance Factors

- Miller & Ranum implements AVL Trees by storing balance factors at the nodes rather than the heights (Miller and Ranum, 2011, Section 6.8.2, page 290)
- The Miller and Ranum explanation of updating the balance factors after a right or left rotation refer to diagrams similar to rotate right and rotate left (page 65)
- This note translates the Miller & Ranum notation to the notation used in these diagrams
- Both approaches have performance $O(\log n)$ but have differences in detail

Right rotation

- New and old balance factors of node k
- Using pseudo-code:

```
newBal(k) = height(tLR) - height(tR)
oldBal(k) = oldHeight(kL) - height(tR)
          = (1 + max(height(tLL), height(tLR)))
            - height(tR)

newBal(k) - oldBal(k)
  = (height(tLR) - height(tR))
    - ((1 + max(height(tLL), height(tLR)))
       - height(tR))
```

```
  = height(tLR)
    - 1 - max(height(tLL), height(tLR))
  = height(tLR)
    - 1 + min(-height(tLL), -height(tLR))
  = min(height(tLR) - height(tLL)
      , height(tLR) - height(tLR)) - 1
  = min(-oldBal(kL), 0) - 1
    since -max(a, b)    = min(-a,-b)
          min(a,b) + c = min(a+c, b+c)
```

- New and old balance factors of node kL

```
newBal(kL) = height(tLL) - newHeight(k)
           = height(tLL)
             - (1 + max(height(tLR), height(tR)))
oldBal(kL) = height(tLL) - height(tLR)

newBal(kL) - oldBal(kL)
  = (height(tLL)
    - (1 + max(height(tLR), height(tR))))
      - (height(tLL) - height(tLR))
  = height(tLR)
    - 1 - max(height(tLR), height(tR))
  = height(tLR)
    - 1 + min(-height(tLR), -height(tR))
  = min(height(tLR) - height(tLR)
      , height(tLR) - height(tR)) - 1
  = min(0, newBal(k)) - 1
```

- Right rotation: New and old balance factors of nodes k and kR

```
newBal(k)
  = oldBal(k) + min(-oldBal(kL), 0) - 1

newBal(kL)
  = oldBal(kL) + min(0, newBal(k)) - 1
```

- This fits with the right rotation diagrams annotated with heights and balance factors on page 55,

```
oldBal(k)  = +2
oldBal(kL) = +1
newBal(k)  =  0 = +2 + min(-1, 0) - 1
newBal(kL) =  0 = +1 + min(0, 0) - 1
```

Left rotation

- New and old balance factors of node k
- Using pseudo-code:

```
newBal(k) = height(tL) - height(tRL)
oldBal(k) = height(tL) - oldHeight(kR)
          = height(tL)
            - (1 + max(height(tRL), height(tRR)))

newBal(k) - oldBal(k)
  = (height(tL) - height(tRL))
    - (height(tL)
      - (1 + max(height(tRL), height(tRR))))
  = 1 + max(height(tRL), height(tRR)) - height(tRL)
  = 1 + max(height(tRL) - height(tRL)
          , height(tRR) - height(tRL))
  = 1 + max(0, -oldBal(kR))
  = 1 - min(0, oldBal(kR))
    since max(-a, -b)  = -min(a,b)
          max(a,b) - c = max(a-c, b-c)
```

- New and old balance factors of node kR

```
newBal(kR) = newHeight(k) - height(tRR)
           = (1 + max(height(tL), height(tRL)))
             - height(tRR)
oldBal(kR) = height(tRL) - height(tRR)

newBal(kR) - oldBal(kR)
  = ((1 + max(height(tL), height(tRL)))
    - height(tRR))
       - (height(tRL) - height(tRR))
  = 1 + max(height(tL), height(tRL)) - height(tRL)
  = 1 + max(height(tL) - height(tRL)
          , height(tRL) - height(tRL))
  = 1 + max(newBal(k), 0)
```

- Left rotation: New and old balance factors of nodes k and kR

```
newBal(k)
  = oldBal(k) + 1 - min(0, oldBal(kR))

newBal(kR)
  = oldBal(kR) + 1 + max(newBal(k), 0)
```

- This fits with the left rotation diagrams annotated with heights and balance factors on page 57,

```
oldBal(k)  = -2
oldBal(kR) = -1
newBal(k)  =  0 = -2 + 1 - min(0,-1)
newBal(kR) =  0 = -1 + 1 + max(0, 0)
```

5.5 AVL Tree Insertion and Deletion

- The insertion and deletion functions are the same as for Binary Search Trees except we have to use makeAVLTree to make a tree unless we really know that the AVL property will be preserved.

```
434 def insertAVLT(x,t):
435   if isEmptyABT(t):
436     return makeABTree(x, EmptyABT(), EmptyABT())
437   else:
438     y = t.dataABT
439     leftT = t.leftABT
440     rightT = t.rightABT
441     if x < y:
442       return makeAVLTree(y, insertAVLT(x, leftT), rightT)
443     elif x > y:
444       return makeAVLTree(y, leftT, insertAVLT(x, rightT))
445     else:
446       return t
```

```
448 def deleteAVLT(x,t):
449   if isEmptyABT(t):
450     return EmptyABT()
451   else:
452     y = t.dataABT
453     leftT = t.leftABT
454     rightT = t.rightABT
455     if x < y:
456       return makeAVLTree(y, deleteAVLT(x, leftT), rightT)
457     elif x > y:
458       return makeAVLTree(y, leftT, deleteAVLT(x, rightT))
459     else:
460       return joinAVLT(leftT, rightT)
```

```
466 def joinAVLT(leftT, rightT):
467   if isEmptyABT(rightT):
```

```
    return leftT
  else:
    (y,t) = splitAVLT(rightT)
    return makeAVLTree(y, leftT, t)
```

```
def splitAVLT(t):
  if isEmptyABT(t):
    raise RuntimeError("splitAVLT applied to EmptyABT()")
  else:
    x = t.dataABT
    t1 = t.leftABT
    t2 = t.rightABT
    if isEmptyABT(t1):
      return (x,t2)
    else:
      (y,t3) = splitAVLT(t1)
      return (y, makeAVLTree(x, t3, t2))
```

```
def insertListAVLT(xs,t):
  if xs == []:
    return t
  else:
    return insertListAVLT(xs[1:], (insertAVLT(xs[0],t)))
```

AVL Tree Insertion and Deletion — Haskell

```
insertAVLT :: (Ord a) => a -> ABinTree a
                          -> ABinTree a

insertAVLT x EmptyABT
  = makeABTree x EmptyABT EmptyABT

insertAVLT x (NodeABT h y leftT rightT)
 | x < y
   =  makeAVLTree y (insertAVLT x leftT) rightT
 | x > y
   =  makeAVLTree y leftT (insertAVLT x rightT)
 | x == y
   =  NodeABT h y leftT rightT
```

AVL Tree Insertion and Deletion — Haskell

```
deleteAVLT :: (Ord a) => a -> ABinTree a
                          -> ABinTree a

deleteAVLT x EmptyABT  =  EmptyABT

deleteAVLT  x  (NodeABT h y leftT rightT)
 | x < y
   =  makeAVLTree y (deleteAVLT x leftT) rightT
 | x > y
   =  makeAVLTree y leftT (deleteAVLT x rightT)
 | x == y
   =  joinAVLT leftT rightT
```

```
joinAVLT :: ABinTree a -> ABinTree a -> ABinTree a
joinAVLT leftT EmptyABT = leftT
joinAVLT leftT (NodeABT h x t1 t2)
  = makeAVLTree y leftT t
    where
    (y, t) = splitAVLT (NodeABT h x t1 t2)

splitAVLT :: ABinTree a -> (a, ABinTree a)
splitAVLT (NodeABT h x EmptyABT t2) = (x, t2)
splitAVLT (NodeABT h x t1 t2)
  = (y, makeAVLTree x t3 t2)
    where
    (y, t3) = splitAVLT t1

splitAVLT EmptyABT
  =  error "splitAVLT applied to EmptyABT"
```

```haskell
434 insertListAVLT :: Ord a => [a] -> ABinTree a
435                            -> ABinTree a

437 insertListAVLT [] t = t
438 insertListAVLT (x:xs) t
439   = insertListAVLT xs (insertAVLT x t)

441 buildListAVLT :: Ord a => [a] -> ABinTree a

443 buildListAVLT xs
444   =  insertListAVLT xs EmptyABT
```

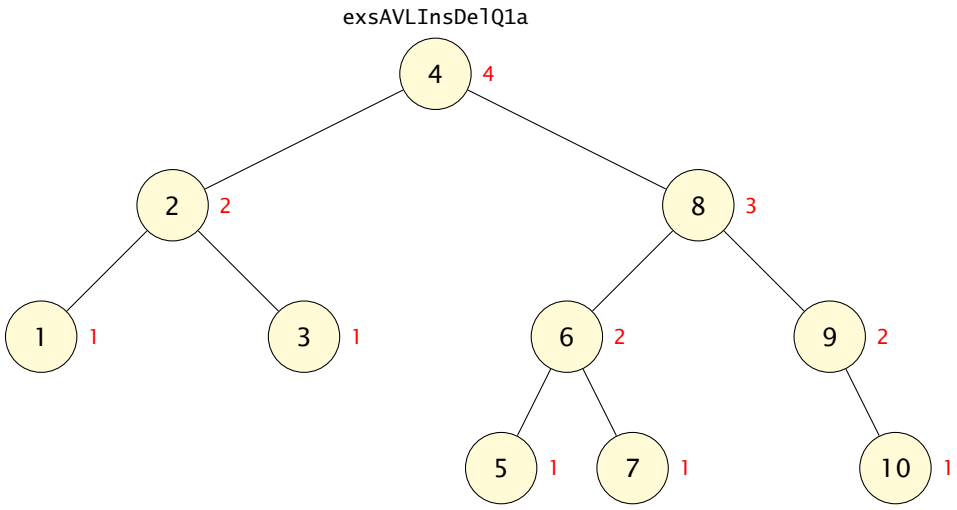
Activity 21 Insert Lists and Delete Items

- Draw the AVL Trees resulting from inserting the following lists of items into an empty tree one by one in order given — do the insertions by hand following the AVL insertion algorithm — you can use the Python code to check your answers
  1. [1,2,3,4,5,6,7,8,9,10]
  2. [10,9,8,7,6,5,4,3,2,1]
  3. [68,88,61,89,94,50,4,76,66,82,99]
- For each of the previous trees, show the result when the fourth item inserted is deleted
- The insertAVLT function is defined at line 434, page 67 (Python), and line 391, page 68 (Haskell), the deleteAVLT function is defined at line 448, page 67 (Python), and line 405, page 68 (Haskell),
- insertListAVLT is defined at line 538, page 68 (Python), and line 435, page 69 (Haskell),

Go to Answer

Answer 21 Insert Lists and Delete Items

```haskell
listQ1a = [1,2,3,4,5,6,7,8,9,10]
exsAVLInsDelQ1a = insertListAVLT(listQ1a, EmptyABT())
```



Answer 21 Insert Lists and Delete Items

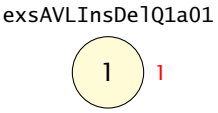
- Here is the Python representation of the resulting AVL tree

```
exsAVLInsDelQ1aAns \
 = (NodeABT(4, 4,
      NodeABT(2, 2,
        NodeABT(1, 1, EmptyABT(), EmptyABT()),
        NodeABT(1, 3, EmptyABT(), EmptyABT())),
      NodeABT(3, 8,
        NodeABT(2, 6,
          NodeABT(1, 5, EmptyABT(), EmptyABT()),
          NodeABT(1, 7, EmptyABT(), EmptyABT())),
        NodeABT(2, 9,
          EmptyABT(),
          NodeABT(1, 10, EmptyABT(), EmptyABT())))))
```

Answer 21 Insert Lists and Delete Items

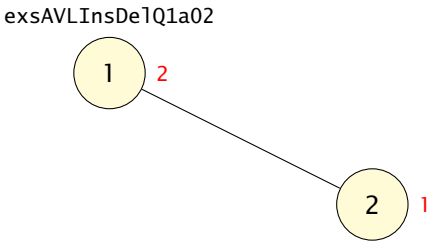
- Here are the insertions done one by one with separate diagrams

```
exsAVLInsDelQ1a01 \
 = insertListAVLT(listQ1a[:1], EmptyABT())
```



Answer 21 Insert Lists and Delete Items

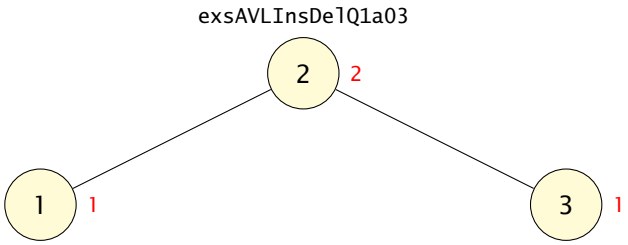
```
exsAVLInsDelQ1a02 \
 = insertListAVLT(listQ1a[:2], EmptyABT())
```



Answer 21 Insert Lists and Delete Items

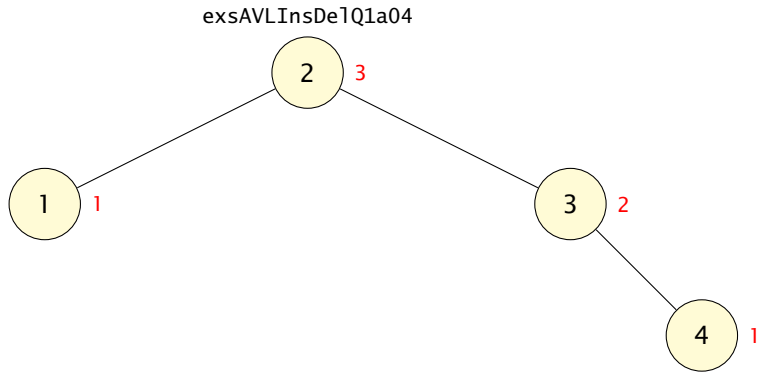
```
exsAVLInsDelQ1a03 \
 = insertListAVLT(listQ1a[:3], EmptyABT())
```

Left rotation about 1



Answer 21 Insert Lists and Delete Items

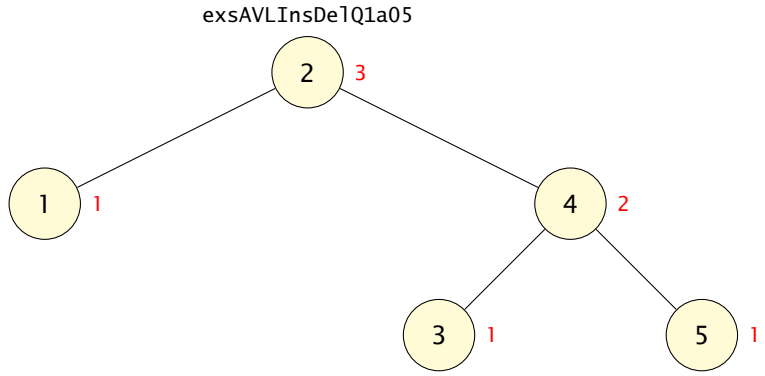
```
exsAVLInsDelQ1a04 \
 = insertListAVLT(listQ1a[:4], EmptyABT())
```



Answer 21 Insert Lists and Delete Items

```
exsAVLInsDelQ1a05 \
 = insertListAVLT(listQ1a[:5], EmptyABT())
```

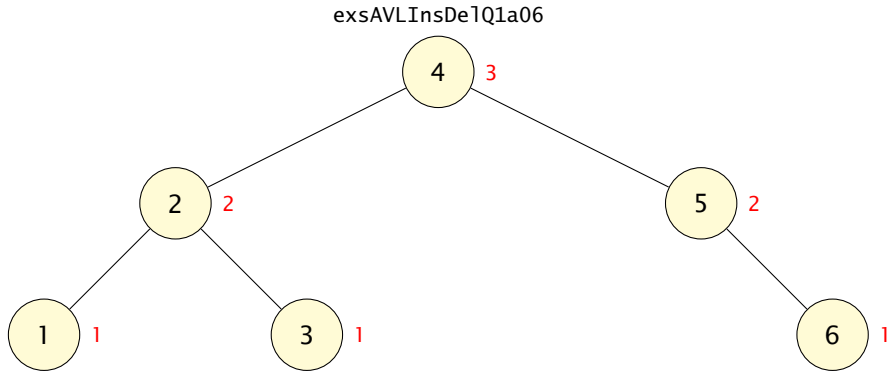
Left rotation about 3



Answer 21 Insert Lists and Delete Items

```
exsAVLInsDelQ1a06 \
 = insertListAVLT(listQ1a[:6], EmptyABT())
```

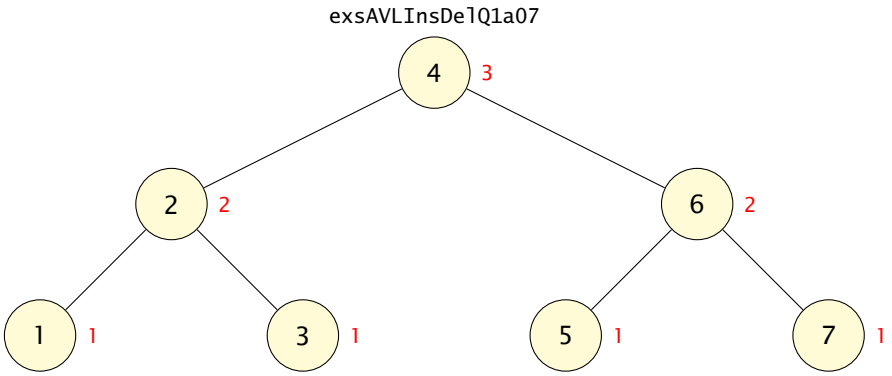
Left rotation about 2



Answer 21 Insert Lists and Delete Items

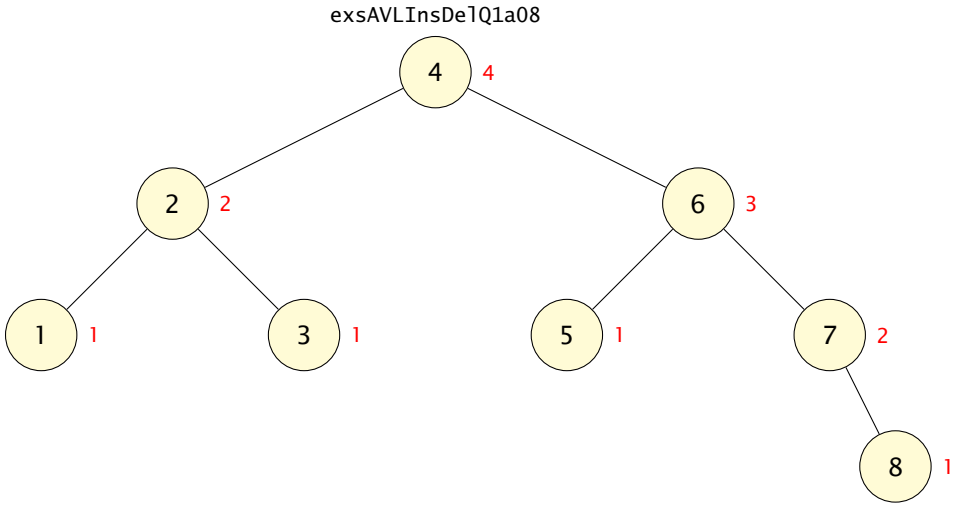
```
exsAVLInsDelQ1a07 \
 = insertListAVLT(listQ1a[:7], EmptyABT())
```

Left rotation about 5



Answer 21 Insert Lists and Delete Items

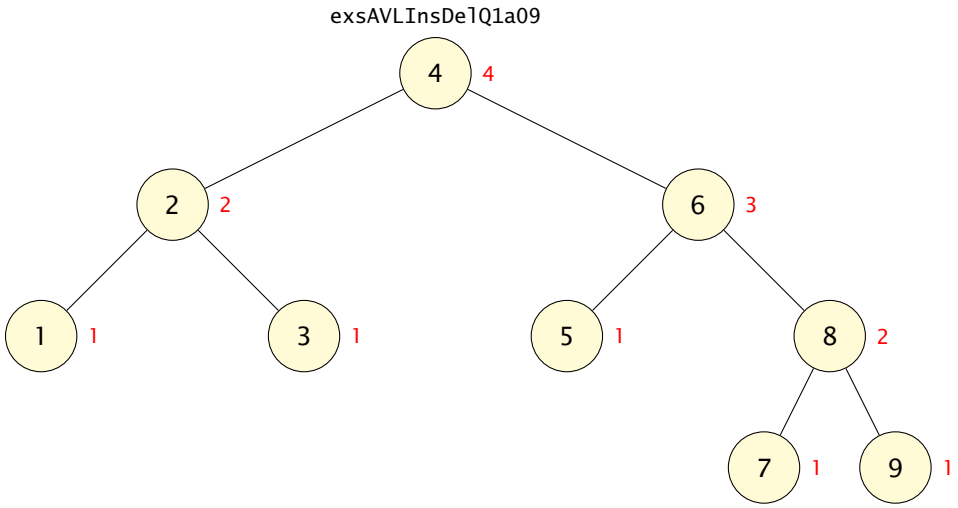
```
exsAVLInsDelQ1a08 \
 = insertListAVLT(listQ1a[:8], EmptyABT())
```



Answer 21 Insert Lists and Delete Items

```
exsAVLInsDelQ1a09 \
 = insertListAVLT(listQ1a[:9], EmptyABT())
```

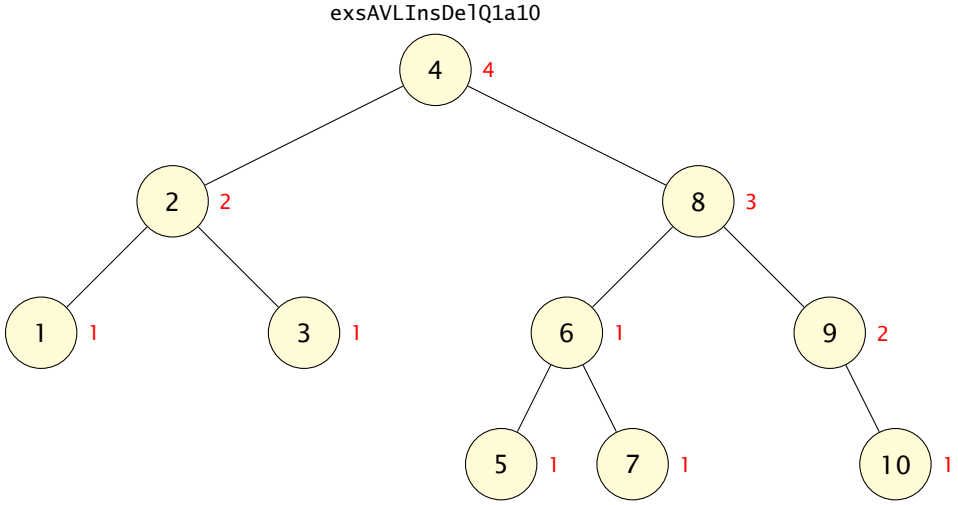
Left rotation about 7



Answer 21 Insert Lists and Delete Items

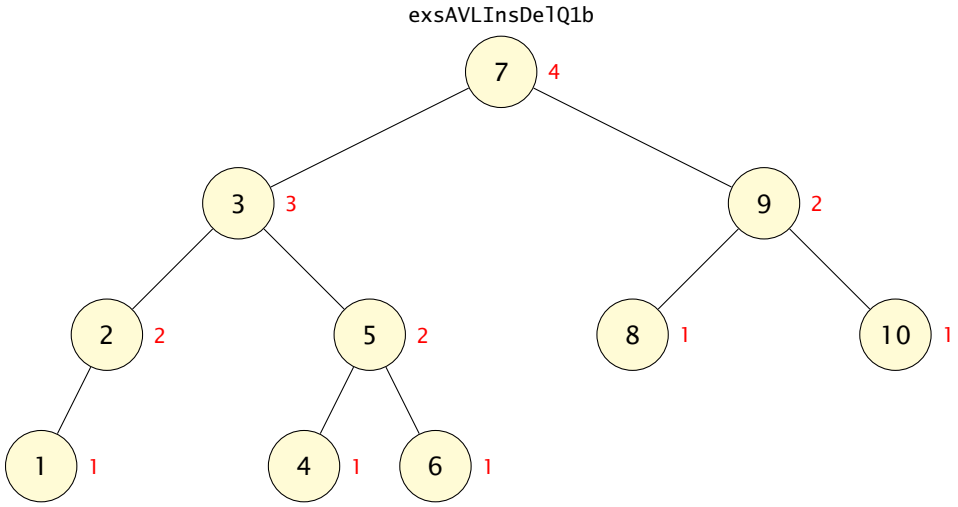
```
exsAVLInsDelQ1a10 \
 = insertListAVLT(listQ1a[:10], EmptyABT())
```

Left rotation around 6



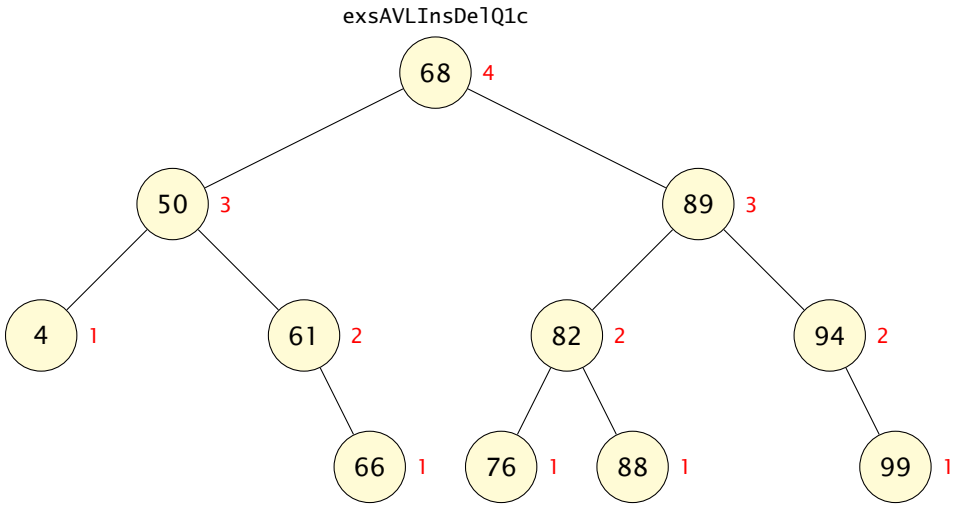
Answer 21 Insert Lists and Delete Items

```
listQ1b = [10,9,8,7,6,5,4,3,2,1]
exsAVLInsDelQ1b = insertListAVLT(listQ1b, EmptyABT())
```



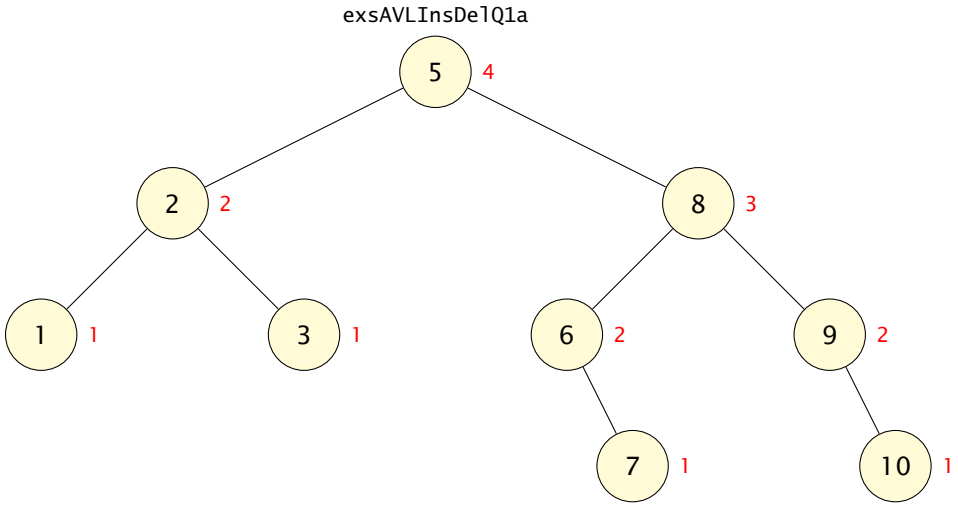
Answer 21 Insert Lists and Delete Items

```
listQ1c = [68,88,61,89,94,50,4,76,66,82,99]
exsAVLInsDelQ1b = insertListAVLT(listQ1c, EmptyABT())
```



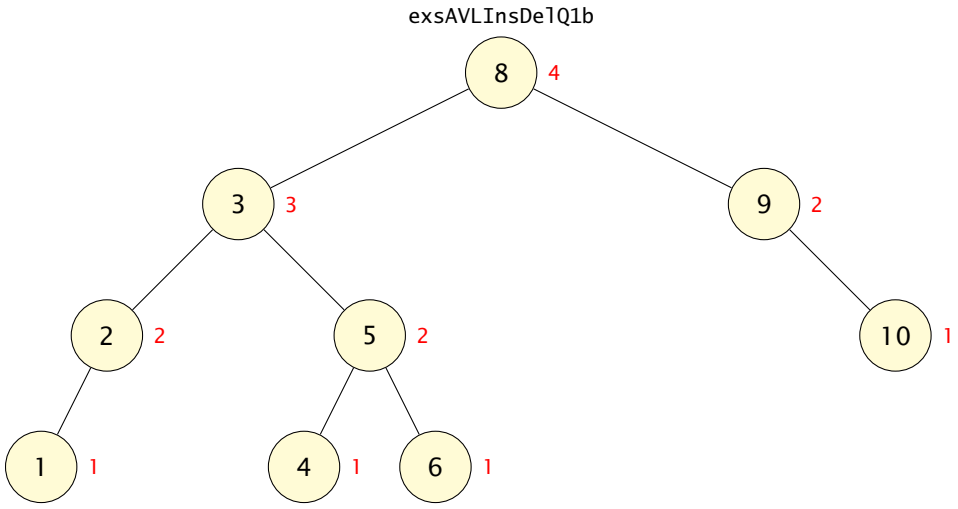
Answer 21 Q2(a) Delete 4th Item

```
listQ1a = [1,2,3,4,5,6,7,8,9,10]
exsAVLInsDelQ2a \
 = deleteAVLT(listQ1a[3], exsAVLInsDelQ1a)
```



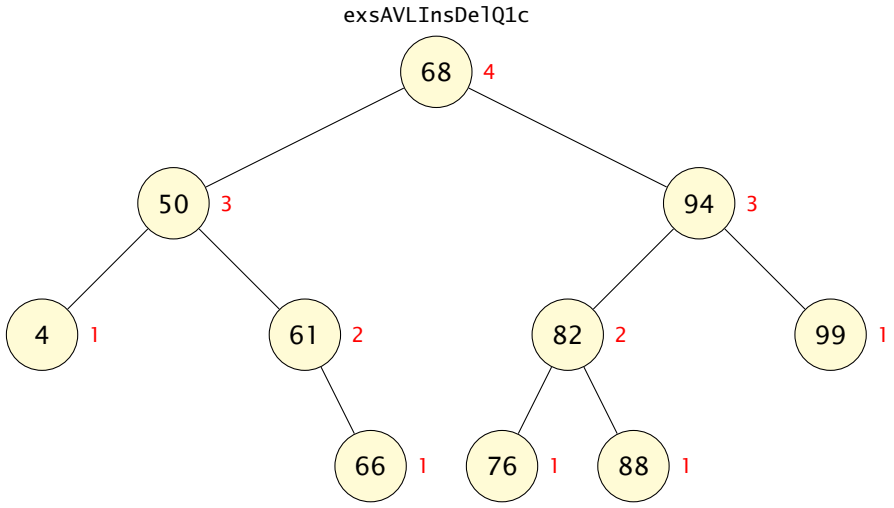
Answer 21 Q2(b) Delete 4th Item

```
listQ1b = [10,9,8,7,6,5,4,3,2,1]
exsAVLInsDelQ1b \
 = deleteAVLT(listQ1b[3], exsAVLInsDelQ1b)
```



Answer 21 Insert Lists and Delete Items

```
listQ1c = [68,88,61,89,94,50,4,76,66,82,99]
exsAVLInsDelQ1c \
 = deleteAVLT(listQ1c[3], exsAVLInsDelQ1c)
```



Go to Activity

Activity 22 Deleting Inserted List

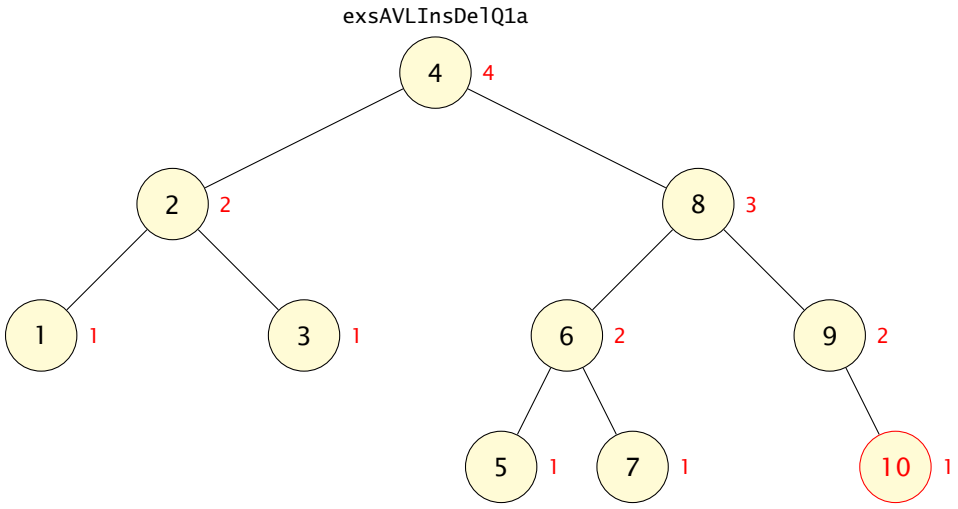
- Using `listQ1a` show that deleting the elements of the list from the tree one by one in reverse order does not result in the reverse sequence of AVL trees

```
listQ1a = [1,2,3,4,5,6,7,8,9,10]
exsAVLInsDelQ1a = insertListAVLT(listQ1a, EmptyABT())
```

Go to Answer

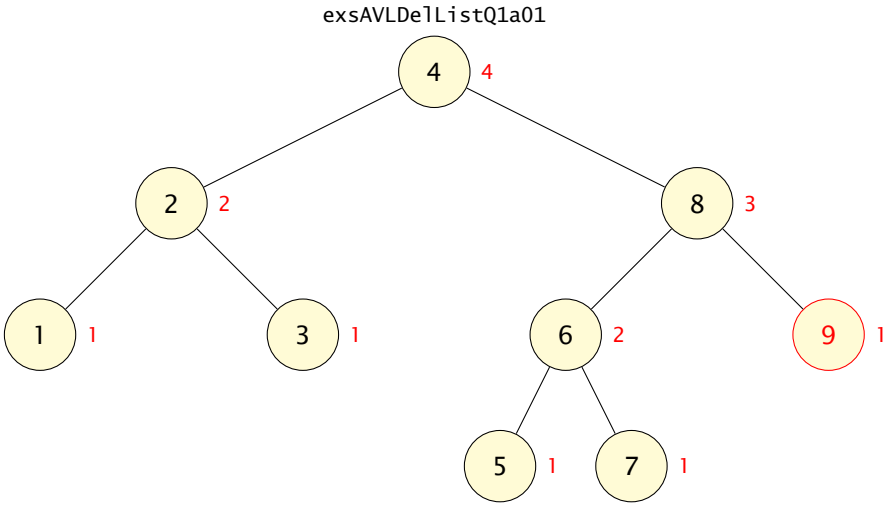
Answer 22 Deleting Inserted List

```
listQ1a = [1,2,3,4,5,6,7,8,9,10]
exsAVLInsDelQ1a = insertListAVLT(listQ1a, EmptyABT())
# delete listQ1a[-1]
```



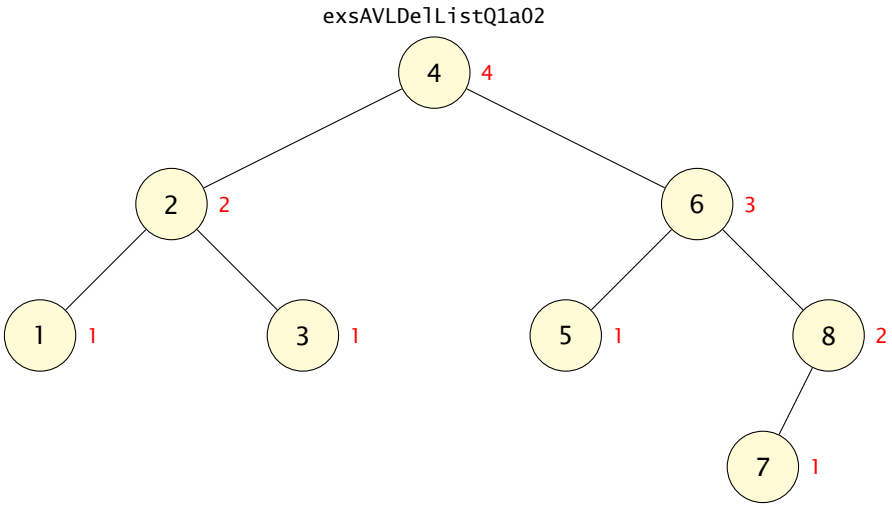
Answer 22 Deleting Inserted List

```
exsAVLDelListQ1a01 \
 = deleteAVLT(listQ1a[-1], exsAVLInsDelQ1a)
# delete listQ1a[-2]
```



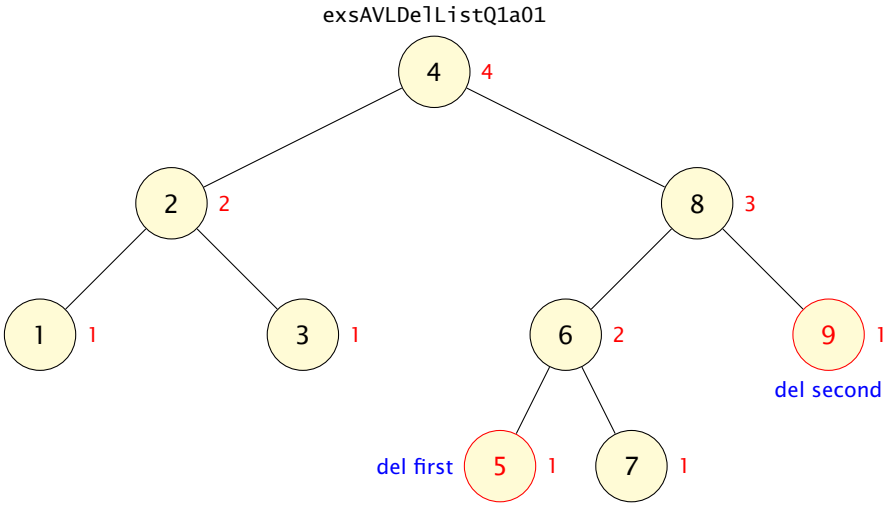
Answer 22 Deleting Inserted List

```
exsAVLDelListQ1a02 \
 = deleteAVLT(listQ1a[-2], exsAVLDelListQ1a01)
# Right rotation about node 8
```



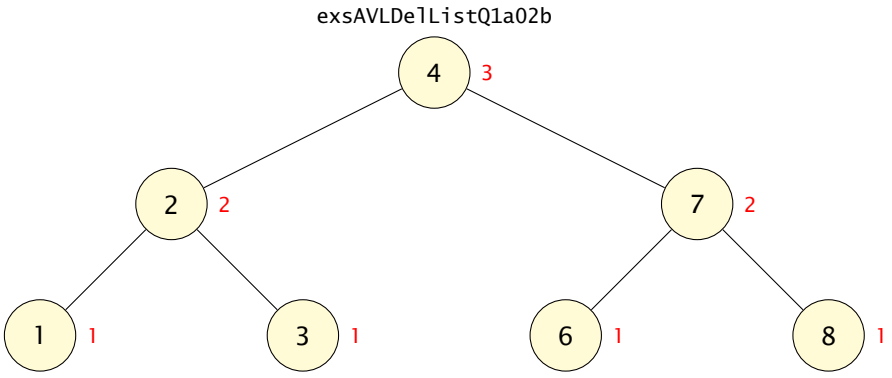
Answer 22 Deleting Inserted List

```
exsAVLDelListQ1a01 \
 = deleteAVLT(listQ1a[-1], exsAVLInsDelQ1a)
# delete 5 first followed by 9
```



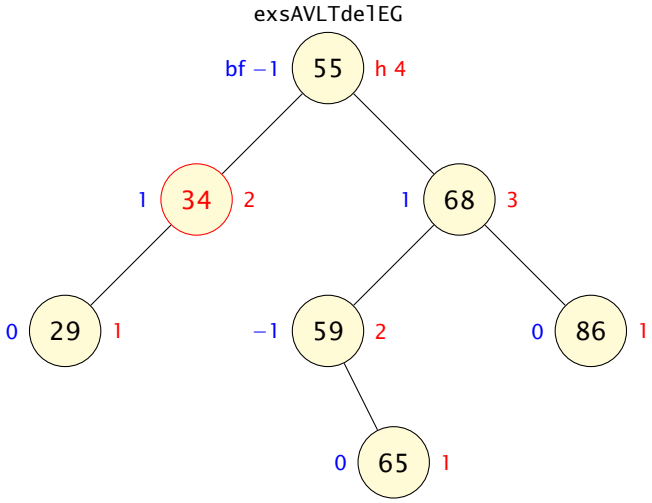
Answer 22 Deleting Inserted List

```
exsAVLDelListQ1a02a \
 = deleteAVLT(listQ1a[-6], exsAVLDelListQ1a01)
exsAVLDelListQ1a02b \
 = deleteAVLT(listQ1a[-2], exsAVLDelListQ1a02a)
# Double rotation: left about 6, right about 8
```



Activity 23 Delete with Rebalance

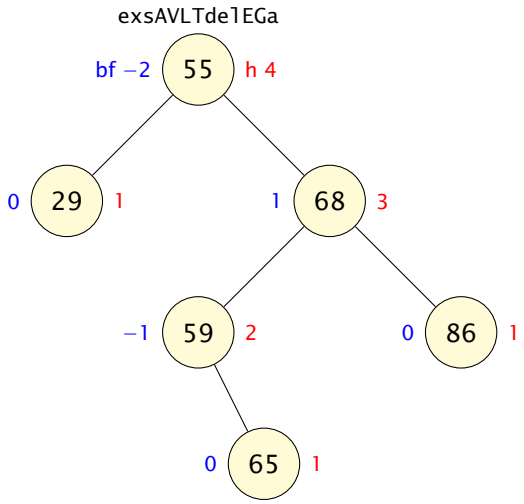
- Example from Specimen Exam (2016) Q 8
- Redraw the tree with node 34 deleted and tree rebalanced. Note here we have height of empty tree as 0 and singleton node as 1



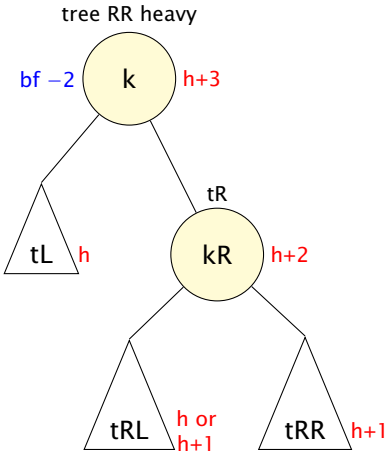
Go to Answer

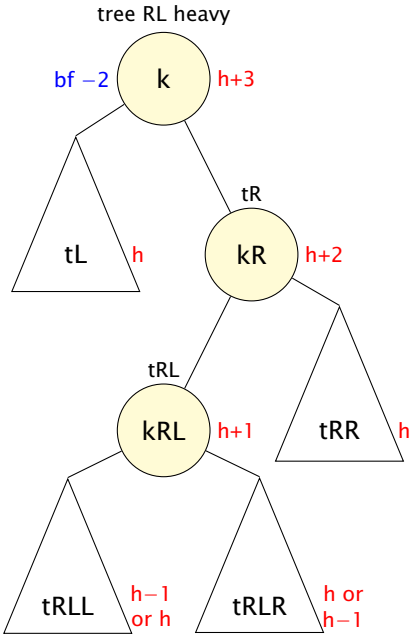
Answer 23 Delete with Rebalance

- Here is the tree with node 34 deleted but not rebalanced
- The new balance factor for the root is -2 so two possible transformations — RR heavy or RL heavy



Go to Activity





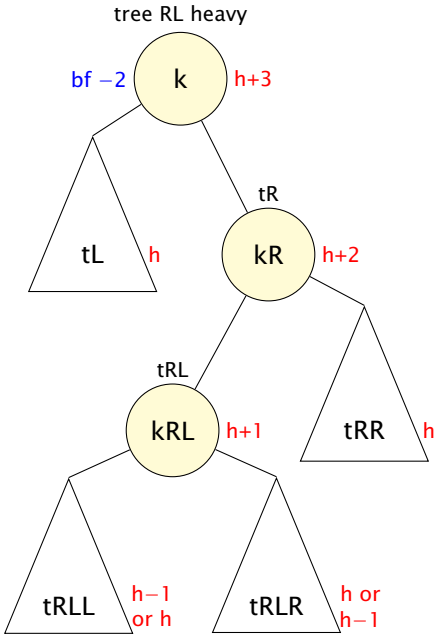
Go to Activity

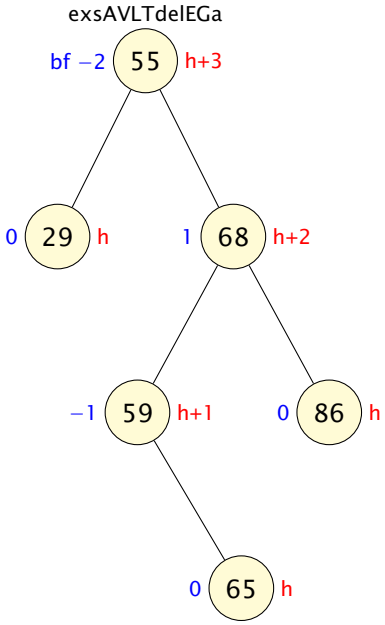
- Exercise: Identify the parts of the tree given in the question with the names given for key nodes and subtrees given in the above diagrams
- Which of the two cases is the given tree an instance of ?

Go to Activity

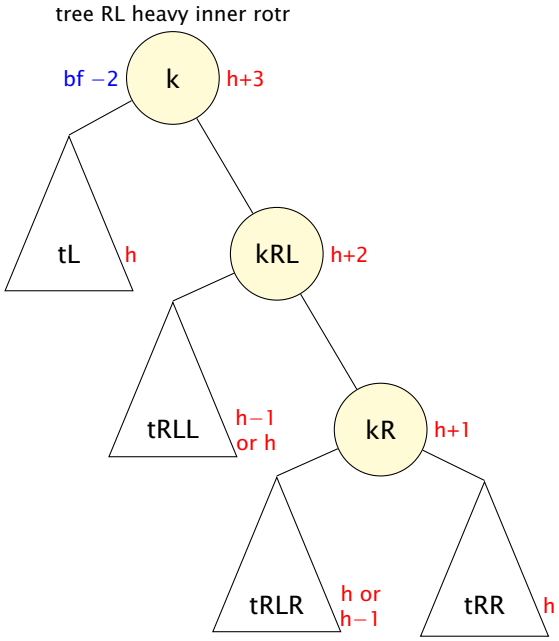
- Key k is 55
- Key kR is 68
- Key kRL is 59
- Subtree tL is rooted at 29
- Subtree tRL is rooted at 59
- Subtree RR is rooted at 86
- Subtree tRLL is an empty tree
- Subtree tRLR is rooted at 65
- The given tree is an instance of RL heavy
- This requires a double rotation to rebalance

Go to Activity

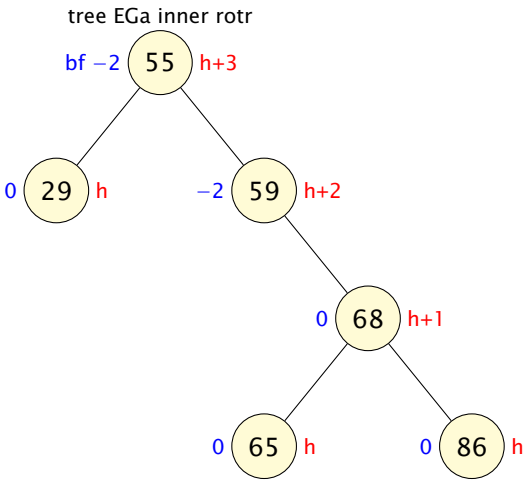




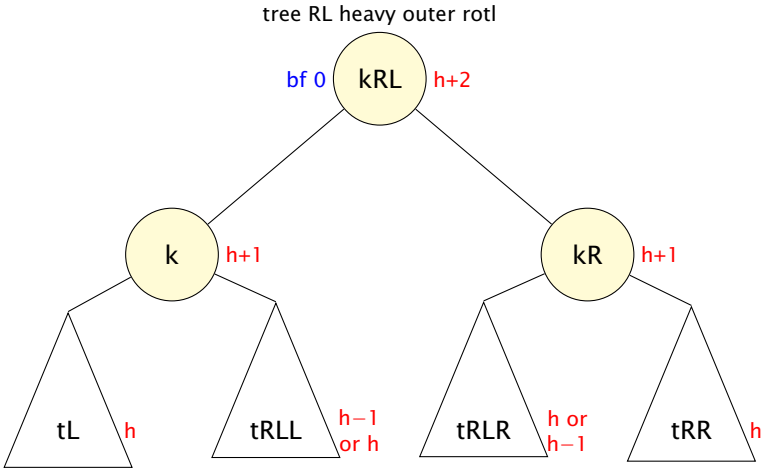
Go to Activity



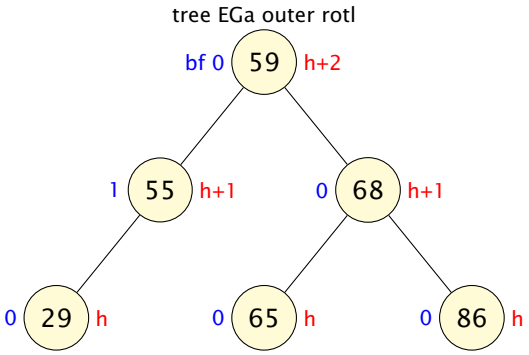
Go to Activity



Go to Activity



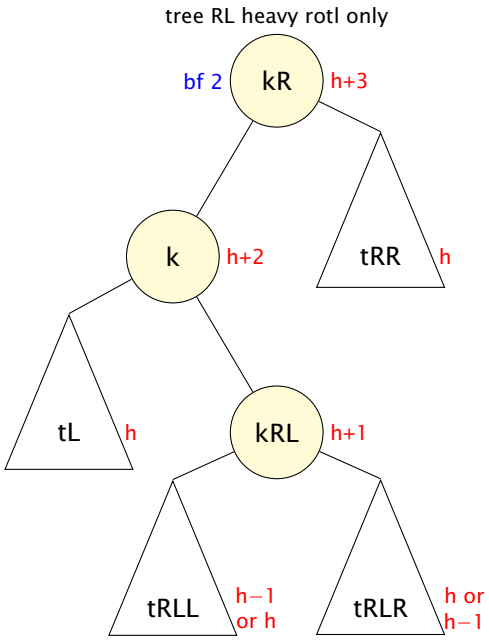
Go to Activity



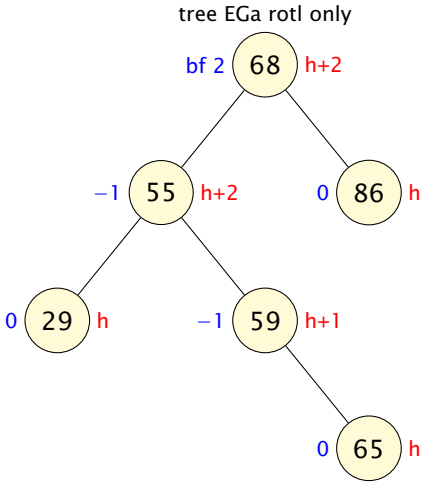
Go to Activity

- Exercise: what would have happened if we had chosen only to do a left rotation around the root ?

Go to Activity



Go to Activity



- This tree is LR heavy and could be rebalanced via a further double rotation but obviously this would be extra work compared to getting the correct double rotation in the first place

Go to Activity

- **Key point** when performing a rebalance, check which case applies
- **LL heavy** right rotation
- **LR heavy** inner left rotation, right outer rotation
- **RL heavy** inner right rotation, left outer rotation
- **RR heavy** left rotation
- See the notes for the details

Go to Activity

5.6 AVL Tree Performance

- While a height balanced tree may not always have the minimum possible height, it has the advantage that it will always be reasonably small
- For a tree with n items we shall show that the maximum number of steps to insert, delete or retrieve an item is $O(\log n)$
- Finding the maximum height of a tree with n items is equivalent to finding the minimum number of items, T_h in a tree of height h
- For $h = 0$ we have an empty tree so $T_0 = 0$
- For T_1 we have a singleton item so $T_1 = 1$
- In general for $h \geqslant 2$ we have $T_h = 1 + T_{h-1} + T_{h-2}$

  since the tree must be balanced and each subtree must have a minimum number of items
- The sequence T_h looks very similar to the Fibonacci sequence
- $F_0 = 0$, $F_1 = 1$

- $F_k = F_{k-1} + F_{k-2}$, $k \geqslant 2$
- The Fibonacci sequence appeared in a work by Leonardo Fibonacci Pisano, who also popularized the Hindu-Arabic numeral system via his 1202 book *Liber Abaci (Book of Calculations)*.
- The sequence also appeared in Indian mathematics much earlier.
- The Fibonacci numbers have lots of interesting properties and turn up in many places in nature
- In our case we have $T_h = F_{h+2} - 1$
- Deriving $T_h = F_{h+2} - 1$
- Let $R_h = T_h - T_{h-1} = 1 + T_{h-2}$
- Then $R_{h+2} = 1 + T_h = 1 + 1 + T_{h-1} + T_{h-2} = R_{d+1} + R_d$
- $R_2 = 1 + T_0 = 1 + 0 = 1 = F_2$ and

  $R_3 = 1 + T_1 = 1 + 1 = 2 = F_3$
- Hence $R_h = F_h,\ \forall h \geqslant 2$
- Hence $T_h = F_{h+2} - 1,\ \forall h \geqslant 0$
- Miller & Ranum approach
- **Level** number of edges from root to node
- **Height** maximum level of any node in the tree — this is one less than my definition
- N_h is the minimum number of nodes in an AVL tree of height h
- $N_0 = 1$ since tree of one node has no edges

  $N_1 = 2$
- $N_h = 1 + N_{h-1} + N_{h-2}$, $h \geqslant 2$
- Let $S_h = N_h - N_{h-1} = 1 + N_{h-2}$
- Then $S_{h+2} = 1 + N_h = 1 + 1 + N_{h-1} + N_{h-2} = S_{d+1} + S_d$
- $S_2 = 1 + N_0 = 1 + 1 = 2 = F_3$ and

  $S_3 = 1 + N_1 = 1 + 2 = 3 = F_4$
- Hence $S_h = F_{h+1},\ \forall h \geqslant 2$
- Hence $N_h = F_{h+3} - 1,\ \forall h \geqslant 0$
- $P(h) : T_h = F_{h+2} - 1$ proof by induction
- **Basis** $P(0), P(1)$
- $T_0 = 1$ and $F_2 - 1 = 1 - 1 = 0$
- $T_1 = 1$ and $F_3 - 1 = 2 - 1 = 1$
- **Inductive step** $\forall k\, P(k) \Rightarrow P(k+1)$
- $T_k = 1 + T_{k-1} + T_{k-2}$

  $= 1 + (F_{k+1} - 1) + (F_k - 1)$

$= F_{k+2} - 1$

- Hence $T_h = F_{h+2} - 1, \ \forall h \geqslant 0$
- There is a closed-form solution for the Fibonacci sequence known as the Euler-Binet Formula (see also A formula for Fib(n))
- $F_k = \dfrac{\phi^k - (1-\phi)^k}{\sqrt{5}}$
- ϕ is the Golden mean
- $\phi = \dfrac{1}{\phi - 1} = \dfrac{1+\sqrt{5}}{2} \approx 1.61803\ldots$
- Hence $T_h = \dfrac{\phi^{h+2} - (1-\phi)^{h+2}}{\sqrt{5}} - 1$
- Since $(1-\phi) < 1$ then for large h we have
- $T_h = n \approx \dfrac{\phi^{h+2}}{\sqrt{5}} - 1 \rightarrow \log(\sqrt{5}(n+1)) \approx (h+2)\log\phi$
- Hence in the worst case, the height of a AVL tree is $O(\log n)$

5.6.1 Proof of Euler-Binet Formula

- Proof of the Euler-Binet Formula is not required for M269 but here is a brief summary
- **Proof by Induction**
- Let $P(n) = F_n = \dfrac{\phi^n - (1-\phi)^n}{\sqrt{5}}$
- **Basis for Induction**
- $P(0)$ is true since

  $\dfrac{\phi^0 - (1-\phi)^0}{\sqrt{5}} = \dfrac{1-1}{\sqrt{5}} = 0 == F_0$
- $P(1)$ is true since

  $\dfrac{\phi^1 - (1-\phi)^1}{\sqrt{5}} = \dfrac{\left(\frac{1+\sqrt{5}}{2}\right) - \left(1 - \left(\frac{1+\sqrt{5}}{2}\right)\right)}{\sqrt{5}}$
- $= 1 == F_1$
- **Induction Hypothesis Step**
- Show $P(j) : 0 \leqslant j \leqslant k+1 \Rightarrow P(k+2)$
- $\phi^{k+2} - (1-\phi)^{k+2} = \phi^2\phi^k - (1-\phi)^2(1-\phi)^k$ and
- $\phi^2 = \left(\frac{1+\sqrt{5}}{2}\right)^2 = \dfrac{1}{4}(1 + 2\sqrt{5} + 5) = 1 + \phi$
- $(1-\phi)^2 = \left(\frac{1-\sqrt{5}}{2}\right)^2 = 1 + (1-\phi)$ hence
- $\phi^{k+2} - (1-\phi)^{k+2} = (1+\phi)\phi^k - (1 + (1-\phi))(1-\phi)^k$
- $= (\phi^k - (1-\phi)^k) + (\phi^{k+1} - (1-\phi)^{k+1})$

- $= \sqrt{5}(F_k + F_{k+1})$ by inductive hypothesis
- $= \sqrt{5}F_{k+2}$ by Fibonacci definition
- Hence $\forall n \in \mathbb{N} : F_n = \dfrac{\phi^n - (1-\phi)^n}{\sqrt{5}}$
- The above proof confirms the formula but here is a derivation
- Define $T(x, y) = (y, x + y)$
- Then $T^n(0, 1) = (F_n, F_{n+1})$ (proof by induction)
- Now find λ_1, λ_2 and $(x_1, y_1), (x_2, y_2)$

  so $T(x_1, y_1) = \lambda_1(x_1, y_1)$ and $T(x_2, y_2) = \lambda_2(x_2, y_2)$

  and $(0, 1) = p_1(x_1, y_1) + p_2(x_2, y_2)$
- $T(x, y) = (y, x + y) = \lambda(x, y)$

  $\rightarrow x + y = \lambda x$ and $x = \lambda y \rightarrow \lambda^2 - \lambda - 1 = 0$

  $\rightarrow \lambda_1 = \phi$ and $\lambda_2 = 1 - \phi$

  and $T(1, \phi) = (\phi, 1 + \phi) = \phi(1, \phi)$

  $T(1, 1 - \phi) = (1 - \phi, 1 + (1 - \phi)) = (1 - \phi)(1, 1 - \phi)$
- $(0, 1) = \frac{1}{\sqrt{5}}(1, \phi) - \frac{1}{\sqrt{5}}(1, 1 - \phi)$ confirm by inspection
- $(F_n, F_{n+1}) = T^n(0, 1)$

  $= \frac{1}{\sqrt{5}}T^n(1, \phi) - \frac{1}{\sqrt{5}}T^n(1, 1 - \phi)$

  $= \frac{1}{\sqrt{5}}\phi^n(1, \phi) - \frac{1}{\sqrt{5}}(1 - \phi)^n(1, 1 - \phi)$

  $= \frac{1}{\sqrt{5}}(\phi^n, \phi^{n+1}) - \frac{1}{\sqrt{5}}((1 - \phi)^n, (1 - \phi)^{n+1})$
- Hence $F_n = \dfrac{\phi^n - (1-\phi)^n}{\sqrt{5}}$

6 Iterative Tree Traversals

- We have used recursion in our implementation of algorithms on binary trees
- This has made it easier to produce correct and fairly simple implementations
- This is mainly because the binary tree data structure is itself defined recursively
- A binary tree is either an empty tree or a node with a data item and two subtrees.
- However the efficiency of this approach will depend on how the chosen programming language is implemented.
- We are using Python and, while Python permits recursion, it does not do some of the optimisations available in other languages, especially pure functional languages (such as Haskell).
- Hence you may find some Python texts down play the use of recursion.

- It is always possible to convert a recursive program into one that just uses iteration with `while` loops or (possibly) `for` loops
- We give below examples of the depth first tree traversals translated from their recursive forms to non-recursive.
- Note that this subsection is for illustration only and you would not be expected to be able to reproduce the code or convert other recursive code.

6.1 Iterative InOrder Traversal

- Here is the original recursive version (from line 126 on page 24)
- The line numbers here are gray to indicate they are a copy of code previously given

```
126 def inOrderBT(t):
127   if isEmptyBT(t):
128     return []
129   else:
130     return (inOrderBT(t.leftBT) + [t.dataBT]
131             + inOrderBT(t.rightBT))
```

- We start with the recursive version but with an accumulating result.

```
660 def inOrderBT0(t):
661   result = []
662   if not isEmptyBT(t):
663     result = result + (inOrderIterBT1(t.leftBT))
664     result.append(t.dataBT)
665     result = result + (inOrderIterBT1(t.rightBT))
666   return result
```

- Turn the final (*almost* tail recursive) call into a `while` loop

```
668 def inOrderIterBT1(t):
669   result = []
670   while not isEmptyBT(t):
671     result = result + (inOrderIterBT1(t.leftBT))
672     result.append(t.dataBT)
673     t = t.rightBT
674   return result
```

- The term *almost* since the last operation is the addition `(+)` but that could be wrapped into the last call
- There is now one recursive call.
- Create a stack to store the function call context.
- In the loop have a conditional to determine whether to store a new context and make the left sub tree the current node or if we are returning, with the appropriate code.

```
676 def inOrderIterBT2(t):
677   result = []
678   stack = []
679   while (not (stack == []) or not isEmptyBT(t)):
680     if not isEmptyBT(t):
681       stack.append(t)
682       t = t.leftBT
683     else:
684       t = stack.pop()
685       result.append(t.dataBT)
686       t = t.rightBT
687   return result
```

Algorithm Description

- `inOrderIterBT2` takes a tree `t` and returns a list of items at the nodes, depth first from left to right
  1. Initialise `result` to an empty list, and `stack`, for the stack of trees to visit, to an empty list
  2. While `stack` is not empty or `t` is not the empty tree
     (a) If `t` is not empty, append `t` to `stack` and assign `t` to be its left sub tree — this is the left recursion
     (b) Otherwise make `t` the top of the `stack` (and remove it), append the item at its node to `result` and make `t` to be its right sub tree
  3. Finally return `result`

6.2 Iterative PreOrder Traversal

- Here is the original recursive version (from line 133 on page 24)

```
133 def preOrderBT(t):
134   if isEmptyBT(t):
135     return []
136   else:
137     return ([t.dataBT] + preOrderBT(t.leftBT)
138             + preOrderBT(t.rightBT))
```

- Start with recursive version with accumulating result.

```
689 def preOrderBT0(t):
690   result = []
691   if not isEmptyBT(t):
692     result.append(t.dataBT)
693     result = result + (preOrderIterBT1(t.leftBT))
694     result = result + (preOrderIterBT1(t.rightBT))
695   return result
```

- Turn the final (*almost* tail recursive) call into a `while` loop.

```
697 def preOrderIterBT1(t):
698   result = []
699   while not isEmptyBT(t):
700     result.append(t.dataBT)
701     result = result + (preOrderIterBT1(t.leftBT))
702     t = t.rightBT
703   return result
```

- There is now one recursive call.
- Create a stack to store the function call context.
- In the loop have a conditional to determine whether to store a new context and make the left sub tree the current node or if we are returning, with the appropriate code

```
705 def preOrderIterBT2(t):
706   result = []
707   stack = []
708   while (not (stack == []) or not isEmptyBT(t)):
709     if not isEmptyBT(t):
710       result.append(t.dataBT)
711       stack.append(t)
712       t = t.leftBT
713     else:
```

```
714       t = stack.pop()
715       t = t.rightBT
716   return result
```

6.3 Iterative PostOrder Traversal

- Here is the original recursive version (from line 140 on page 24)

```
140 def postOrderBT(t):
141   if isEmptyBT(t):
142     return []
143   else:
144     return (postOrderBT(t.leftBT)
145             + postOrderBT(t.rightBT) + [t.dataBT])
```

- Start with recursive version with accumulating result.

```
718 def postOrderBT0(t):
719   result = []
720   if not isEmptyBT(t):
721     result = result + (postOrderIterBT1(t.leftBT))
722     result = result + (postOrderIterBT1(t.rightBT))
723     result.append(t.dataBT)
724   return result
```

- There is now no final (tail recursive) call. We have to mimic the call stack *carefully*

```
726 def postOrderIterBT1(t):
727   result = []
728   if isEmptyBT(t):
729     return result
730   stack = []
731   while not (stack == []) or (not isEmptyBT(t)):
732     while not isEmptyBT(t):
733       if not isEmptyBT(t.rightBT):
734         stack.append(t.rightBT)
735       stack.append(t)
736       t = t.leftBT
737     t = stack.pop()
738     if ((not isEmptyBT(t.rightBT))
739         and (stack != [] and stack[-1] is t.rightBT)):
740       tr = stack.pop()
741       stack.append(t)
742       t = t.rightBT
743     else:
744       result.append(t.dataBT)
745       t = EmptyBT()  # To avoid infinite loop - it is t.rightBT
746   return result
```

Algorithm Description

1. Initialise `result` to an empty list.
2. If `t` is empty then return `result` (not really needed since the loop would take care of this)
3. Initialise `stack`, for the stack of trees to visit, to an empty list
4. While `stack` is not empty or `t` is not the empty tree
   (a) While `t` is not the empty tree
   - If the right sub tree of `t` is not empty, push it on to `stack`
   - Append `t` to `stack`

- Assign `t` its left sub tree

(b) Pop a node from `stack` and set it as `t`

(c) If the popped node has a non empty right child and the right child is at the top of `stack`

- Remove the right child from the `stack`
- Push the current node `t` on to `stack`
- Set `t` to be `t`'s right child

(d) Otherwise

- Append the data at the root of `t` to `result`
- Set `t` to `Empty()` — marking `t` as visited, prevents infinite looping (it is `t.rightBT`)

5. Finally return `result`

Concluding Points

- Recursive versions are easier to get right.
- Iterative versions mimic the stack of recursive function calls.
- Other non-recursive versions use different data structures with pointers to parent nodes. The code is still more complex (and error prone) compared to the recursive versions.

7 AVL Tree Application — Sets

- Ordered sets and ordered maps are important data types in programming
- Some programming languages have them as builtin types (Python) or supply them as standard libraries (C++, C#, Java, Scala, Haskell, ML)
- This section describes an example implementation based on Blelloch et al. (2016) and Adams (1993)
- In Python the documentation for Sets is at Set Types and for Dictionaries (Maps) at Mapping Types — dict
- The Python implementation can be found at the Python Developer's Guide and the source code for Sets is at setobject.c — the implementation is in C using hash tables — see How is set() implemented?
- In Haskell the documentation and implementation of Sets is at Containers: Data.Set and for Maps at Containers: Data.Map.Strict — both of these are from the package containers: Assorted concrete container types
- The Haskell implementation uses size balanced trees — this is similar to AVL balanced trees
- For an introduction see containers - Introduction and Tutorial
- For an overview of Sets see Containers: Sets

- M269 Unit 5 has representation of graphs for various algorithms — here are some references for that topic for future notes
- For Haskell graph libraries see
  - fgl: Martin Erwig's Functional Graph Library
  - graphs: A simple monadic graph library by Edward Kmett
  - Data.Graph based on King and Launchbury (1995)

7.1 Set Representation

7.1.1 Split, Join

- The essence of the representation is to generalise the `split` and `join` functions

```
splitAVLS :: key -> ABinTree key
             -> (ABinTree key, Bool, ABinTree key)

splitLastAVLS :: ABinTree key -> (key, ABinTree key)

splitFirstAVLS :: ABinTree key -> (key, ABinTree key)

join2AVLS :: ABinTree key -> ABinTree key
             -> ABinTree key

joinAVLS :: key -> ABinTree key -> ABinTree key
            -> ABinTree key

exposeAVLS :: ABinTree key
              -> (ABinTree key, key, ABinTree key)
```

- `splitAVLS` takes a key `k` and an AVL tree `t` and returns two trees `tL` and `tR` and a boolean `b` — `tL` and `tR` have elements less than and greater than `k` respectively and `b` indicates if `k` was in `t`
- `splitLastAVLS`, `splitFirstAVLS` take an AVL tree `t` and return the largest, smallest elemeent `k` respectively and the rest of the tree — `splitFirstAVLS` is similar to `splitAVLT` at line 424 on page 68
- `join2AVLS`, `joinAVLS` take two AVL trees, `tL`, `tR` where all elements of `tL` are less than all elements of `tR` and returns a new AVL tree — `joinAVLS` also takes a key `k` with a value in between the elements of the two trees — `join2AVLS` is similar to `joinAVLT` at line 417 on page 68
- `exposeAVLS` takes apart a tree, `t` to give `(tL, k, tR)`

```
446 makeEmptyABT :: ABinTree key
447 makeEmptyABT = EmptyABT

449 isEmptyABT :: ABinTree key -> Bool
450 isEmptyABT EmptyABT  = True
451 isEmptyABT t         = False

453 keyABT :: ABinTree key -> key
454 keyABT   (NodeABT h k tL tR) = k
455 leftABT :: ABinTree key -> ABinTree key
456 leftABT  (NodeABT h k tL tR) = tL
457 rightABT :: ABinTree key -> ABinTree key
458 rightABT (NodeABT h k tL tR) = tR

460 expose :: ABinTree key
461           -> (ABinTree key, key, ABinTree key)
462 expose t
```

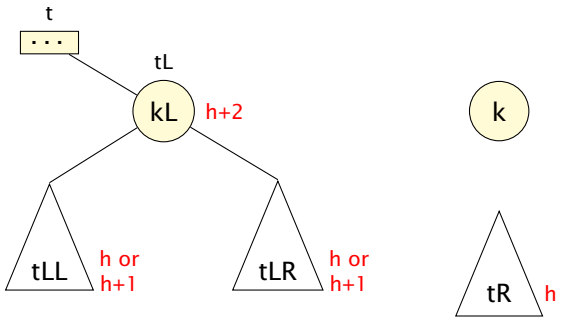
```
463   = (tL, k, tR)
464     where
465       tL = leftABT t
466       k  = keyABT t
467       tR = rightABT t
```

- joinAVLS take a key, k, two AVL trees, tL, tR where all elements of tL are less than k which is less than all elements of tR and returns a new AVL tree

```
469 joinAVLS :: key -> ABinTree key -> ABinTree key
470             -> ABinTree key
471 joinAVLS k tL tR
472  = if heightABT tL > heightABT tR + 1
473    then joinRightAVLS k tL tR
474    else if heightABT tR > heightABT tL + 1
475         then joinLeftAVLS k tL tR
476         else makeABTree k tL tR
```

- joinRight description is in the following diagrams

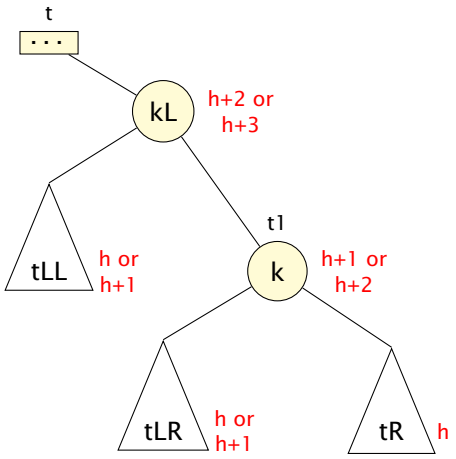
```
478 joinRightAVLS :: key -> ABinTree key -> ABinTree key
479                  -> ABinTree key
480 joinRightAVLS k tL tR
481  = let (tLL, kL, tLR) = expose tL in
482      if heightABT tLR <= heightABT tR + 1
483      then let t1 = makeABTree k tLR tR in
484             if heightABT t1 <= heightABT tLL + 1
485             then makeABTree kL tLL t1
486             else rotl (makeABTree kL tLL (rotr t1))
487      else let t2 = joinRightAVLS k tLR tR
488                t3 = makeABTree kL tLL t2
489           in
490             if heightABT t2 <= heightABT tLL + 1
491             then t3
492             else rotl t3
```



- The base case (line 482 on page 91) of joinRightAVLS follows the right spine of t to a node kL for which

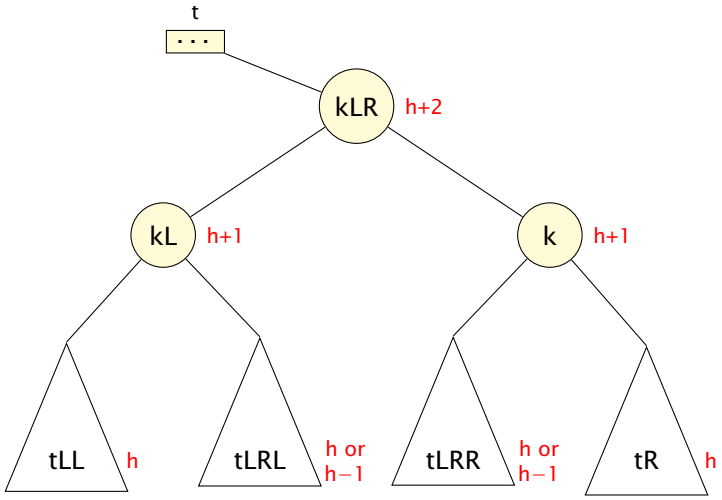
```
heightABT tL > heightABT tR + 1
heightABT tLR <= heightABT tR + 1
```

- We then connect tL, k and tR



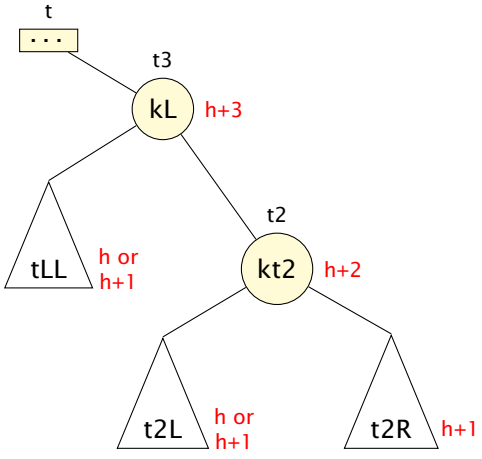
- Needs double rotation if

```
heightABT t1 > heightABT tLL + 1
```



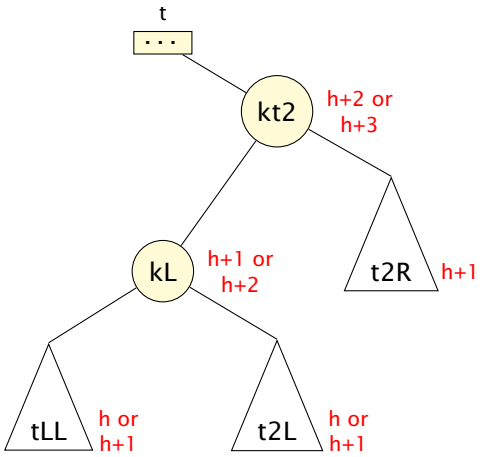
- The recursive case (line 487 on page 91) of `joinRightAVLS` follows the right spine further

```
heightABT tLR > heightABT tR + 1
```



- A single left rotation is needed if

```
heightABT t2 > heightABT tLL + 1
```




```
494  joinLeftAVLS :: key -> ABinTree key -> ABinTree key
495                  -> ABinTree key
496  joinLeftAVLS k tL tR
497   = let (tRL, kR, tRR) = expose tR in
498       if heightABT tRL <= heightABT tL + 1
499       then let t1 = makeABTree k tL tRL in
500              if heightABT t1 <= heightABT tRR + 1
501              then makeABTree kR t1 tRR
502              else rotr (makeABTree kR (rotl t1) tRR)
503       else let t2 = joinLeftAVLS k tL tRL
504                t3 = makeABTree kR t2 tRR
505            in
506              if heightABT t2 <= heightABT tRR + 1
507              then t3
508              else rotr t3
```

Activity 24 joinLeftAVLS Diagrams

- `joinLeftAVLS` is the mirror image of `joinRightAVLS`
- Produce the equivalent diagrams describing the function

Go to Answer

Answer 24 joinLeftAVLS Diagrams

- TODO: Answer 24 joinLeftAVLS Diagrams

Go to Activity

Activity 25 joinLeftAVLS Bug

- A previous version of `joinLeftAVLS` had a bug (beware copy/paste) — see below
- What would happen if the elements of [10,9,8,7,6] were given as input ?

```
joinLeftAVLS :: key -> ABinTree key -> ABinTree key
                -> ABinTree key
joinLeftAVLS k tL tR
 = let (tRL, kR, tRR) = expose tR in
     if heightABT tRL <= heightABT tL + 1
     then let t1 = makeABTree k tL tRL in
            if heightABT t1 <= heightABT tRR + 1
            then makeABTree kR t1 tRR
            else rotr (makeABTree kR (rotl t1) tRR)
     else let t2 = joinRightAVLS k tL tRL
              t3 = makeABTree kR t2 tRR
          in
            if heightABT t2 <= heightABT tRR + 1
            then t3
            else rotr t3
```

Go to Answer

Answer 25 joinLeftAVLS Bug

- TODO: Answer 25 joinLeftAVLS Bug

Go to Activity

```
splitLastAVLS :: ABinTree key -> (ABinTree key, key)
splitLastAVLS t
 = let (tL, k, tR) = expose t in
     if isEmptyABT tR
     then (tL,k)
     else let (tR1,k1) = splitLastAVLS tR in
            (joinAVLS k tL tR1, k1)

splitFirstAVLS :: ABinTree key -> (ABinTree key, key)
splitFirstAVLS t
 = let (tL, k, tR) = expose t in
     if isEmptyABT tL
     then (tR,k)
     else let (tL1,k1) = splitFirstAVLS tL in
            (joinAVLS k tL1 tR, k1)
```

```
splitAVLS :: Ord key => key -> ABinTree key
             -> (ABinTree key, Bool, ABinTree key)
splitAVLS k t
 = if isEmptyABT t
   then (makeEmptyABT, False, makeEmptyABT)
   else
     let (tL, k1, tR) = expose t in
     if k == k1
     then (tL, True, tR)
     else if k < k1
          then
            let (tLL, b, tLR) = splitAVLS k tL in
              (tLL, b, (joinAVLS k1 tLR tR))
          else
            let (tRL, b, tRR) = splitAVLS k tR in
              ((joinAVLS k1 tL tRL), b, tRR)
```

```
join2AVLS :: ABinTree key -> ABinTree key
             -> ABinTree key
join2AVLS tL tR
 = if isEmptyABT tL
   then tR
   else
     let (tL1, k) = splitLastAVLS tL in
     joinAVLS k tL tR
```

7.1.2 Set Operations

- **Set Operations**
- `insertAVLS t k` inserts a key, `k`, into a tree, `t`
- `deleteAVLS t k` deletes key, `k`, from a tree, `t`, if it is in the tree
- `unionAVLS t1 t2` takes two AVL trees whose values may overlap, and returns the union as a tree
- `intersectAVLS t1 t2` takes two AVL trees and returns the intersection as a tree
- `differenceAVLS t1 t2` takes two AVL trees and returns the elements that are in `t1` but not `t2`

```
insertAVLS :: Ord key => ABinTree key -> key
              -> ABinTree key
insertAVLS t k
 = let (tL, found, tR) = splitAVLS k t in
```

```
556       joinAVLS k tL tR

558   deleteAVLS :: Ord key => ABinTree key -> key
559                 -> ABinTree key
560   deleteAVLS t k
561    = let (tL, found, tR) = splitAVLS k t in
562       join2AVLS tL tR

564   insertListAVLS :: Ord key => ABinTree key -> [key]
565                     -> ABinTree key
566   insertListAVLS t [] = t
567   insertListAVLS t (x : xs)
568    = insertListAVLS (insertAVLS t x) xs

570   setFromListAVLS :: Ord key => [key] -> ABinTree key
571   setFromListAVLS xs
572    = insertListAVLS makeEmptyABT xs
```

```
574   unionAVLS :: Ord key => ABinTree key -> ABinTree key
575                -> ABinTree key
576   unionAVLS t1 t2
577    = if      isEmptyABT t1 then t2
578      else if isEmptyABT t2 then t1
579      else let (t2L, k2, t2R) = expose t2
580               (t1L, found, t1R) = splitAVLS k2 t1
581               tL = unionAVLS t1L t2L
582               tR = unionAVLS t1R t2R
583           in
584             joinAVLS k2 tL tR
```

```
586   intersectAVLS :: Ord key =>
587                    ABinTree key -> ABinTree key
588                    -> ABinTree key
589   intersectAVLS t1 t2
590    = if      isEmptyABT t1 then makeEmptyABT
591      else if isEmptyABT t2 then makeEmptyABT
592      else let (t2L, k2, t2R) = expose t2
593               (t1L, found, t1R) = splitAVLS k2 t1
594               tL = intersectAVLS t1L t2L
595               tR = intersectAVLS t1R t2R
596           in
597             if found
598             then joinAVLS k2 tL tR
599             else join2AVLS tL tR
```

```
601   disjointAVLS :: Ord key =>
602                   ABinTree key -> ABinTree key -> Bool
603   disjointAVLS t1 t2
604    = if      isEmptyABT t1 then True
605      else if isEmptyABT t2 then True
606      else let (t2L, k2, t2R) = expose t2
607               (t1L, found, t1R) = splitAVLS k2 t1
608           in
609             not found
610             && disjointAVLS t1L t2L
611             && disjointAVLS t1R t2R
```

- `disjointAVLS` examples — note multiline input

```
GHCi> :{
....> disjointAVLS (setFromListAVLS [2,4,6])
....> (setFromListAVLS [1,3])
....> :}
True
GHCi> :{
....> disjointAVLS (setFromListAVLS [2,4,6,8])
....> (setFromListAVLS [2,3,5,7])
....> :}
False
GHCi> :{
....> disjointAVLS (setFromListAVLS [1,2])
....> (setFromListAVLS [1,2,3,4])
....> :}
False
```

```
GHCi> :{
....> disjointAVLS (setFromListAVLS [])
....> (setFromListAVLS [])
....> :}
True
```

```
differenceAVLS :: Ord key =>
                  ABinTree key -> ABinTree key
                  -> ABinTree key
differenceAVLS t1 t2
 = if      isEmptyABT t1 then makeEmptyABT
   else if isEmptyABT t2 then t1
   else let (t2L, k2, t2R) = expose t2
            (t1L, found, t1R) = splitAVLS k2 t1
            tL = differenceAVLS t1L t2L
            tR = differenceAVLS t1R t2R
        in
          join2AVLS tL tR
```

7.2 Set Operation Examples

- unionAVLS example

```
GHCi> :{
....> unionAVLS (setFromListAVLS [1,3,5,7,9])
....> (setFromListAVLS [2,4,6,8,10])
....> :}
NodeABT 4 4
  (NodeABT 2 2
     (NodeABT 1 1 EmptyABT EmptyABT)
     (NodeABT 1 3 EmptyABT EmptyABT))
  (NodeABT 3 8
     (NodeABT 2 6
        (NodeABT 1 5 EmptyABT EmptyABT)
        (NodeABT 1 7 EmptyABT EmptyABT))
     (NodeABT 2 10
        (NodeABT 1 9 EmptyABT EmptyABT)
        EmptyABT))
```

- List to sets

```
GHCi> setFromListAVLS [1,3,5,7,9]
NodeABT 3 3
  (NodeABT 1 1 EmptyABT EmptyABT)
  (NodeABT 2 7
     (NodeABT 1 5 EmptyABT EmptyABT)
     (NodeABT 1 9 EmptyABT EmptyABT))
GHCi> setFromListAVLS [2,4,6,8,10]
NodeABT 3 4
  (NodeABT 1 2 EmptyABT EmptyABT)
  (NodeABT 2 8
     (NodeABT 1 6 EmptyABT EmptyABT)
     (NodeABT 1 10 EmptyABT EmptyABT))
```

- Lists to sets

```
GHCi> setFromListAVLS [2,3,2,3]
NodeABT 2 3
  (NodeABT 1 2 EmptyABT EmptyABT)
  EmptyABT
```

- sampleSet81to100

```
sampleSet81to100 :: ABinTree Integer
sampleSet81to100
 = setFromListAVLS [81..100]

sampleSet81to100Ans
```

```
630   = NodeABT 5 88
631       sampleSet81to100AnsLeft
632       sampleSet81to100AnsRight

634   sampleSet81to100Test
635    = sampleSet81to100 == sampleSet81to100Ans
```

```
637   sampleSet81to100AnsLeft
638    = NodeABT 3 84
639        (NodeABT 2 82
640           (NodeABT 1 81 EmptyABT EmptyABT)
641           (NodeABT 1 83 EmptyABT EmptyABT))
642        (NodeABT 2 86
643           (NodeABT 1 85 EmptyABT EmptyABT)
644           (NodeABT 1 87 EmptyABT EmptyABT))

646   sampleSet81to100AnsRight
647    = NodeABT 4 96
648        (NodeABT 3 92
649           (NodeABT 2 90
650              (NodeABT 1 89 EmptyABT EmptyABT)
651              (NodeABT 1 91 EmptyABT EmptyABT))
652           (NodeABT 2 94
653              (NodeABT 1 93 EmptyABT EmptyABT)
654              (NodeABT 1 95 EmptyABT EmptyABT)))
655        (NodeABT 3 98
656           (NodeABT 1 97 EmptyABT EmptyABT)
657           (NodeABT 2 100
658              (NodeABT 1 99 EmptyABT EmptyABT)
659              EmptyABT))
```

- sampleSet100to81

```
662   sampleSet100to81 :: ABinTree Integer
663   sampleSet100to81
664    = setFromListAVLS [100,99..81]

666   sampleSet100to81Ans
667    = NodeABT 5 93
668        sampleSet100to81AnsLeft
669        sampleSet100to81AnsRight

671   sampleSet100to81Test
672    = sampleSet100to81 == sampleSet100to81Ans
```

```
674   sampleSet100to81AnsLeft
675    = NodeABT 4 85
676        (NodeABT 3 83
677           (NodeABT 2 81
678              EmptyABT
679              (NodeABT 1 82 EmptyABT EmptyABT))
680           (NodeABT 1 84 EmptyABT EmptyABT))
681        (NodeABT 3 89
682           (NodeABT 2 87
683              (NodeABT 1 86 EmptyABT EmptyABT)
684              (NodeABT 1 88 EmptyABT EmptyABT))
685           (NodeABT 2 91
686              (NodeABT 1 90 EmptyABT EmptyABT)
687              (NodeABT 1 92 EmptyABT EmptyABT)))

689   sampleSet100to81AnsRight
690    = NodeABT 3 97
691        (NodeABT 2 95
692           (NodeABT 1 94 EmptyABT EmptyABT)
693           (NodeABT 1 96 EmptyABT EmptyABT))
694        (NodeABT 2 99
695           (NodeABT 1 98 EmptyABT EmptyABT)
696           (NodeABT 1 100 EmptyABT EmptyABT))
```

7.3 Further Operations

Activity 26 Set Foldr

- Write a function `setFoldr` which implements a foldr on a set

Go to Answer

Answer 26 Set Foldr

- Set Foldr

```
698 setFoldr :: (a -> b -> b) -> b -> ABinTree a -> b
699 setFoldr f z t
700  = go z t
701    where
702      go y EmptyABT = y
703      go y (NodeABT h x tL tR)
704       = go (f x (go y tR)) tL
```

Go to Activity

Activity 27 Set to Ascending List

- Write a function `setToAscList` which takes a set and returns the contents as an ascending list

Go to Answer

Answer 27 Set to Ascending List

- TODO: Answer 27 Set to Ascending List

```
706 setToAscList :: ABinTree a -> [a]
707 setToAscList t
708  = setFoldr (:) [] t

710 sampleSet81to100toAscList
711  = setToAscList sampleSet81to100

713 sampleSet100to81toAscList
714  = setToAscList sampleSet100to81

716 sampleSetToAscListTestA
717  = sampleSet81to100toAscList
718     == sampleSet100to81toAscList
```

Go to Activity

Activity 28 Set Equality

- Write a function `setEquality` which takes two sets, `t1` and `t2` and returns `True` if they are equal and `False` otherwise

Go to Answer

Answer 28 Set Equality

- Answer 28 Set Equality

```
720 setEquality :: Eq a => ABinTree a -> ABinTree a
721                 -> Bool
722 setEquality t1 t2
723  = length l1 == length l2
724     && l1 == l2
725    where
726      l1 = setToAscList t1
727      l2 = setToAscList t2
```

```
729 setEqualityTestA
730  = setEquality sampleSet81to100 sampleSet100to81
```

Go to Activity

Activity 29 Subset

- Write a function subsetAVLS that takes two sets t1, t2 and returns True if t1 is a subset of t2 and False otherwise
- t1 is a subset of t2 if every element of t1 is a member of t2

Go to Answer

Answer 29 Subset

```
732 subsetAVLS :: Ord key => ABinTree key -> ABinTree key
733               -> Bool
734 subsetAVLS t1 t2
735  = if      isEmptyABT t1 then True
736    else if isEmptyABT t2 then False
737    else let (t1L, k1, t1R) = expose t1
738             (t2L, found, t2R) = splitAVLS k1 t2
739         in
740         found
741         && subsetAVLS t1L t2L
742         && subsetAVLS t1R t2R
```

Go to Activity

7.4 Sets — Implementation Points

- The only tree specific functions are joinAVLS, joinRightAVLS and joinLeftAVLS — AVL trees could be changed to size balanced or Red-Black trees with little to be changed
- The various sets operations use splitAVLS and joinAVLS or join2AVLS to avoid more complex algorithms — some implementations may inline the functions for efficiency
- The diagrams for joinRightAVLS are essential for the understanding of the base and recursive cases
- The unionAVLS, intersectAVLS and differenceAVLS functions are very similar in their usage of split and join
- From O'Sullivan et al. (2008, page 301) see Data Structures
- *Maps give us the same capabilities as hash tables do in other languages. Internally, a map is implemented as a balanced binary tree. Compared to a hash table, this is a much more efficient representation in a language with immutable data. This is the most visible example of how deeply pure functional programming affects how we write code: we choose data structures and algorithms that we can express cleanly and that perform efficiently, but our choices for specific tasks are often different [from] their counterparts in imperative languages.*
- See Curious about the HashTable performance issues

	Python		Haskell
Operation	**Average**	**Worst**	**Worst**
Member	$O(1)$	$O(n)$	$O(\log n)$
Union	$O(m+n)$		$O(m \log(\frac{n}{m}+1))$
Intersection	$O(\min(m,n))$	$O(m \times n)$	$O(m \log(\frac{n}{m}+1))$
Difference	$O(m)$		$O(m \log(\frac{n}{m}+1))$
Insert	$O(1)$	$O(n)$	$O(\log n)$
Delete	$O(1)$	$O(n)$	$O(\log n)$

- Python: Time Complexity
- Haskell: Data.Set and Data.Map.Strict
- Remember that actual behaviour will depend on the data and compiler settings

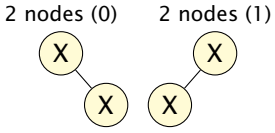
8 Binary Tree Common Exercises

- This section contains some common exercises used *Google Interview Questions*
- See the References section for Web sites with more examples
- Note that this section is not directly part of M269 and is here for interest and practice
- Further questions may be added to this section
- The later parts touch of topics such as lazy evaluation and lazy pattern matching which are not part of M269 but may be of interest

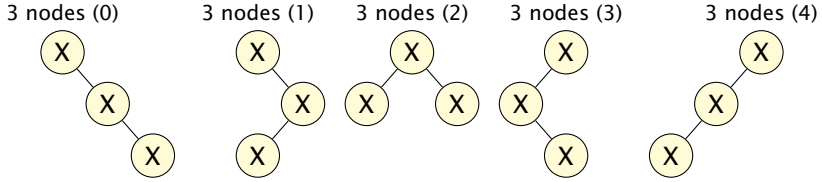
8.1 Generating Binary Trees

8.1.1 Simple Recursive Version

- We first sketch a few trees to spot the pattern
- 0 nodes have 1 tree, `EmptyBT`, 1 node has 1 tree
- 2 nodes have 2 trees



- 3 nodes have 5 trees



- If C_n is the number of binary trees with n nodes then
- $C_0 = 1$
- $C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1} = \sum_{i=1}^{n} C_{i-1} C_{n-i}$
- Alternatively $C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i}$
- $C_1 = C_0 C_0 = 1 \times 1 = 1$
- $C_2 = C_0 C_1 + C_1 C_0 = 1 \times 1 + 1 \times 1 = 2$
- $C_3 = C_0 C_2 + C_1 C_1 + C_2 C_0 = 1 \times 2 + 1 \times 1 + 2 \times 1 = 5$
- $C_4 = C_0 C_3 + C_1 C_2 + C_2 C_1 + C_3 C_0$

  $= 1 \times 5 + 1 \times 2 + 2 \times 1 + 5 \times 1 = 14$
- The recurrence is on all possible distributions of subtrees
- The C_n are known as the Catalan numbers
- The simple recursive definition of genBTs follows from the recurrence relation directly
- This repeats the calculation of subtrees
- This is similar to the definition in Math.Combinat.Trees.Binary which is based on Knuth (2011, section 7.2.1.6),Knuth (1997, section 2.3.4.4)

```
744 genBTs :: a -> Int -> [BinTree a]
745 genBTs x 0 = [EmptyBT]
746 genBTs x 1 = [NodeBT x EmptyBT EmptyBT]
747 genBTs x n
748   = [NodeBT x leftT rightT | (nu,nv) <- splitsInt n
749                            , leftT  <- genBTs x nu
750                            , rightT <- genBTs x nv]

752 splitsInt :: Int -> [(Int,Int)]
753 splitsInt n
754   = [(i, n - i - 1) | i <- [0..(n-1)]]

756 splitsIntA :: Int -> [(Int,Int)]
757 splitsIntA n
758   = [(i - 1, n - i) | i <- [1..n]]
```

8.1.2 Sample Trees

```
760 data DummyData = X
761     deriving (Eq,Ord,Show,Read)

763 dval = X -- dummy value

765 genBTsTest0 = genBTs dval 0

767 genBTsAns0
768   = [EmptyBT]

770 genBTsTest1 = genBTs dval 1

772 genBTsAns1
773   = [NodeBT X EmptyBT EmptyBT]

775 genBTsTest2 = genBTs dval 2
```

```
genBTsAns2
  = [NodeBT X EmptyBT (NodeBT X EmptyBT EmptyBT)
    ,NodeBT X (NodeBT X EmptyBT EmptyBT) EmptyBT]
```

```
genBTsTest3 = genBTs dval 3

genBTsAns3
  = [NodeBT X EmptyBT
              (NodeBT X EmptyBT (NodeBT X EmptyBT EmptyBT))
    ,NodeBT X EmptyBT
              (NodeBT X (NodeBT X EmptyBT EmptyBT) EmptyBT)
    ,NodeBT X (NodeBT X EmptyBT EmptyBT)
              (NodeBT X EmptyBT EmptyBT)
    ,NodeBT X (NodeBT X EmptyBT (NodeBT X EmptyBT EmptyBT))
              EmptyBT
    ,NodeBT X (NodeBT X (NodeBT X EmptyBT EmptyBT) EmptyBT)
              EmptyBT]

genBTsTest4 = genBTs dval 4

genBTsTest5 = genBTs dval 5
```

8.2 Binary Tree Shape Tests

- `isSameShape` takes two binary trees and returns `True` if they have the same shape

```
isSameShape :: BinTree a -> BinTree a -> Bool
isSameShape t1 t2
  | isEmptyBT t1 && isEmptyBT t2  = True
  | isEmptyBT t1 || isEmptyBT t2  = False
  | otherwise
      = isSameShape (getLeftBT t1) (getLeftBT t2)
        && isSameShape (getRightBT t1) (getRightBT t2)
```

- `isMirrorShape` takes two binary trees and returns `True` if they are a mirror of each other

```
isMirrorShape :: BinTree a -> BinTree a -> Bool
isMirrorShape t1 t2
  | isEmptyBT t1 && isEmptyBT t2  = True
  | isEmptyBT t1 || isEmptyBT t2  = False
  | otherwise
      = isMirrorShape (getLeftBT t1) (getRightBT t2)
        && isMirrorShape (getRightBT t1) (getLeftBT t2)
```

- `isSymmetric` takes a binary tree and returns `True` if it is symmetric

```
isSymmetric :: BinTree a -> Bool
isSymmetric t
  = isEmptyBT t
    || isMirrorShape (getLeftBT t) (getRightBT t)

-- Shorter alternative
-- isSymmetric t
--   = isMirrorShape t t
```

- `genMirrorShape` takes a binary tree and returns the mirror of the tree

```
genMirrorShape :: BinTree a -> BinTree a
genMirrorShape EmptyBT = EmptyBT
genMirrorShape (NodeBT x leftT rightT)
  = makeBT x
           (genMirrorShape rightT)
           (genMirrorShape leftT)
```

8.2.1 Sample Tests

```
831 genBTsSym5
832   = [t | t <- genBTsTest5, isSymmetric t]

834 genBTsSym5Ans
835   = [NodeBT X (NodeBT X EmptyBT (NodeBT X EmptyBT EmptyBT))
836               (NodeBT X (NodeBT X EmptyBT EmptyBT) EmptyBT)
837     ,NodeBT X (NodeBT X (NodeBT X EmptyBT EmptyBT) EmptyBT)
838               (NodeBT X EmptyBT (NodeBT X EmptyBT EmptyBT))]
```

```
840 genBTsSym7
841   = [t | t <- genBTs dval 7, isSymmetric t]

843 genBTsSym7Ans
844   = [NodeBT X (NodeBT X EmptyBT
845                         (NodeBT X EmptyBT (NodeBT X EmptyBT EmptyBT)))
846               (NodeBT X (NodeBT X (NodeBT X EmptyBT EmptyBT) EmptyBT)
847                         EmptyBT)
848     ,NodeBT X (NodeBT X EmptyBT
849                         (NodeBT X (NodeBT X EmptyBT EmptyBT) EmptyBT))
850               (NodeBT X (NodeBT X EmptyBT (NodeBT X EmptyBT EmptyBT))
851                         EmptyBT)
852     ,NodeBT X (NodeBT X (NodeBT X EmptyBT EmptyBT)
853                         (NodeBT X EmptyBT EmptyBT))
854               (NodeBT X (NodeBT X EmptyBT EmptyBT)
855                         (NodeBT X EmptyBT EmptyBT))
856     ,NodeBT X (NodeBT X (NodeBT X EmptyBT (NodeBT X EmptyBT EmptyBT))
857                         EmptyBT)
858               (NodeBT X EmptyBT
859                         (NodeBT X (NodeBT X EmptyBT EmptyBT) EmptyBT))
860     ,NodeBT X (NodeBT X (NodeBT X (NodeBT X EmptyBT EmptyBT) EmptyBT)
861                         EmptyBT)
862               (NodeBT X EmptyBT
863                         (NodeBT X EmptyBT (NodeBT X EmptyBT EmptyBT)))]
```

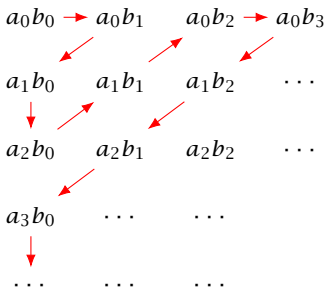
8.3 Catalan Numbers

8.3.1 Cauchy Product

- $$\left(\sum_{i=0}^{\infty} a_i x^i\right)\left(\sum_{j=0}^{\infty} b_j x^j\right) = \sum_{n=0}^{\infty} c_n x^n$$

  where $c_n = \sum_{k=0}^{n} a_k b_{n-k}$

- The product forms a two-dimensional array — however we can arrange a sequence that goes through the array — see below and Spivak (2008, p486, p493, p513)



- $$\left(\sum_{i=0}^{\infty} a_i x^i\right)\left(\sum_{j=0}^{\infty} b_j x^j\right)$$

$$= (a_0 + a_1x + a_2x^2 + \cdots)(b_0 + b_1x + b_2x^2 + \cdots)$$
$$= a_0b_0 + (a_0b_1 + a_1b_0)x + (a_0b_2 + a_1b_1 + a_2b_0)x^2 + \cdots$$

- See Wikipedia: Cauchy product
- If we have $c(x) = \sum_{i=0}^{\infty} c_i x^i$
- $(c(x))^2 = \left(\sum_{i=0}^{\infty} c_i x^i\right)\left(\sum_{j=0}^{\infty} c_j x^j\right)$
$$= \sum_{n=0}^{\infty}\left(\sum_{k=0}^{n} c_k c_{n-k}\right)x^n$$
- This result is used in finding a closed form for the Catalan numbers
- Based on Mike Spivey 2013

8.3.2 Catalan Recurrence

- $C_0 = 1$
- $C_{n+1} = \sum_{k=0}^{n} C_k C_{n-k}$
- Define $c(x)$ to be the generating function of the infinite sequence of the Catalan numbers
- $c(x) = \sum_{n=0}^{\infty} C_n x^n$
- Hence we can multiply both sides of the recurrence by x^n and sum
- $\sum_{n=0}^{\infty} C_{n+1}x^n = \sum_{n=0}^{\infty}\left(\sum_{k=0}^{n} C_k C_{n-k}\right)x^n$
- $\sum_{n=0}^{\infty} C_{n+1}x^n = \sum_{n=0}^{\infty}\left(\sum_{k=0}^{n} C_k C_{n-k}\right)x^n$
- $\frac{1}{x}\sum_{n=0}^{\infty} C_{n+1}x^{n+1} = (c(x))^2$ by Cauchy product
- $\frac{1}{x}\left(\sum_{n=0}^{\infty} C_n x^n - C_0\right) = (c(x))^2$
- $\frac{1}{x}(c(x) - 1) = (c(x))^2$
- $x(c(x))^2 - c(x) + 1 = 0$
- $c(x) = \frac{1 \pm \sqrt{1 - 4x}}{2x}$
- We know $c(0) = C_0 = 1$

- $c(x) = \dfrac{1 \pm \sqrt{1-4x}}{2x}$
- We know $c(0) = C_0 = 1$
- Applying L'Hôpital's rule

  $$\lim_{x \to c} \frac{f(x)}{g(x)} = \lim_{x \to c} \frac{f'(x)}{g'(x)}$$

- $\lim\limits_{x \to 0^+} \dfrac{1-\sqrt{1-4x}}{2x} = \lim\limits_{x \to 0^+} \dfrac{2(1-4x)^{-\frac{1}{2}}}{2} = 1$
- Hence $c(x) = \dfrac{1-\sqrt{1-4x}}{2x}$
- We now use the generalised Binomial theorem to expand this expression
- The generalised Binomial theorem has

  If $|x| > |y|$ and r is any complex number then

  $$(x+y)^r = \sum_{k-0}^{\infty} \binom{r}{k} x^{r-k} y^k$$

  where $\binom{r}{k} = \dfrac{r(r-1)\cdots(r-k+1)}{k!}$

- $c(x) = \dfrac{1}{2x}(1-\sqrt{1-4x}) = \dfrac{1}{2x}\left(1 - \sum\limits_{n=0}^{\infty} \binom{1/2}{n}(-4x)^n\right)$
- The coefficient of x^n expands to

  $$\frac{\frac{1}{2}\left(\frac{1}{2}-1\right)\cdots\left(\frac{1}{2}-n+1\right)}{n!}(-4)^n$$
  $$= \frac{1(1-2)\cdots(1-2n+2)}{n!}(-1)^n 2^n$$

- The coefficient of x^n expands to

  $$\frac{\frac{1}{2}\left(\frac{1}{2}-1\right)\cdots\left(\frac{1}{2}-n+1\right)}{n!}(-4)^n$$
  $$= \frac{1(1-2)\cdots(1-2n+2)}{n!}(-1)^n 2^n$$
  $$= \frac{(1)(3)\cdots(2n-3)(-1)^{n-1}}{(n!)^2}(-1)^n 2^n (n!)$$
  $$= \frac{(1)(3)\cdots(2n-3)(-1)^{n-1}}{(n!)^2}(-1)^n (2n)(2n-2)\cdots(2)$$
  $$= -\frac{(2n)!}{(n!)^2(2n-1)}$$
  $$= -\binom{2n}{n}\frac{1}{2n-1}$$

- Hence $c(x) = \dfrac{1}{2x}\left(1 + \sum\limits_{n=0}^{\infty} \binom{2n}{n}\dfrac{1}{2n-1}x^n\right)$

$$= \frac{1}{2x}\left(1 + (-1) + \sum_{n=1}^{\infty}\binom{2n}{n}\frac{1}{2n-1}x^n\right)$$

$$= \frac{1}{2}\sum_{n=1}^{\infty}\binom{2n}{n}\frac{1}{2n-1}x^{n-1}$$

$$= \frac{1}{2}\sum_{n=0}^{\infty}\binom{2(n+1)}{n+1}\frac{1}{2n+1}x^{n}$$

$$= \frac{1}{2}\sum_{n=0}^{\infty}\frac{(2n+2)(2n+1)}{(n+1)^2}\binom{2n}{n}\frac{1}{2n+1}x^{n}$$

$$= \sum_{n=0}^{\infty}\frac{1}{n+1}\binom{2n}{n}x^{n}$$

- Hence $C_n = \frac{1}{n+1}\binom{2n}{n}$

8.3.3 Sample Catalan Numbers

```
  In[1]:= Series[(1 - Sqrt[1-4x])/(2x),{x,0,12}]
  Out[1]= SeriesData[x, 0, {1, 1, 2, 5, 14, 42, 132, 429, 1430, \
4862, 16796, 58786, 208012}, 0, 13, 1]
```

- Generating function form
- $1 + x + 2x^2 + 5x^3 + 14x^4 + 42x^5 + 132x^6$
  $+ 429x^7 + 1430x^8 + 4862x^9 + 16796x^{10}$
  $+ 58786x^{11} + 208012x^{12} + O\left(x^{13}\right)$

9 Final Points

- The section on search trees used a Python immutable data type, a named tuple, with a more functional style of programming.
- This had the advantage of short, easier to understand programs
- They corresponded more closely to the diagrams describing the ideas of the algorithms.
- Naturally, a correct program is of the first importance but we pointed out that this style of program, using Python, may need transforming to make it more efficient.
- It is important to realise that your chosen language may have as much impact on the efficiency of your code as any programmer optimisation.

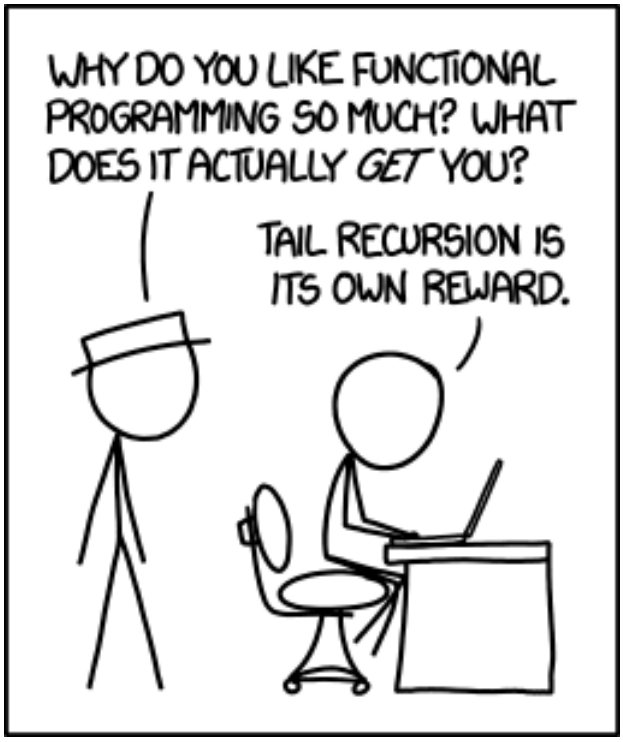
Epigrams on Functional Programming

- We end this section with a couple of epigrams on Functional Programming.
- *Functional programmers know the value of everything, but the cost of nothing.*
- This is a slight misquote of Guy Steele in Steele (2009a,b)

- who in turn was slightly misquoting Alan Perlis in Perlis (1982)
- — both with apologies to Oscar Wilde.
- As Guy Steele says, this is no longer true of modern implementations such as OCaml and Haskell.
- *Recursion is the GOTO of Functional Programming*
- This is from Erik Meijer in Meijer et al. (1991, page 1–2)
- who is referring to Dijkstra's famous 1968 *Go To Statement Considered Harmful* (Dijkstra, 1968),
- which recommends what came to be known as *Structured Programming* (sequence, selection and iteration).
- Meijer is making a similar point about recursion:
- he identifies several common patterns and higher-order functions which capture these common recursion patterns.

Functional Programming Cartoon

- Finally a cartoon on the topic from XKCD (see Functional and Explain 1270: Functional)



- The hover tooltip shown with this cartoon is:
- *Functional programming combines the flexibility and power of abstract mathematics with the intuitive clarity of abstract mathematics.*

10 Future Work

- Unit 5 Optimisation & Graph Algorithms

- Unit 6 Logic
- 10 March iCMA45
- Saturday 13 March 2021 Tutorial Online 10:00–13:00 Units 3,4,5
- 24 March TMA02
- Sunday 11 April 2021 Tutorial Online 10:00–11:00 Unit 6 Logic

11 Web Sites & References

- **Wikipedia: Binary Tree** en.wikipedia.org/wiki/Binary_tree
- **Wikipedia: Binary Search Tree** en.wikipedia.org/wiki/Binary_search_tree
- **Wikipedia: Self Balancing Binary Search Trees** en.wikipedia.org/wiki/Self-balancing_binary_search_-tree
- **Wikipedia: AVL Tree** en.wikipedia.org/wiki/AVL_tree
- **AVL Trees** notes from Eric Alexander (2011) pages.cs.wisc.edu/~ealexand/cs367/NOTES/AVL-Trees/
- **AVL Trees** notes from Patrick Prosser, Glasgow University (uses Java) www.dcs.gla.ac.uk/~pat/52233/slides
- **AVL Trees** notes from Tim Sheard (uses Haskell, 2009) web.cecs.pdx.edu/~sheard/course/Cs163/Doc/AvlTrees.
- **Rosetta Code: AVL Tree** www.rosettacode.org/wiki/AVL_tree
- **Codelike: AVL Tree Visualization** www.codelike.in/animation/avl-tree
- **Data Structure Visualizations** www.cs.usfca.edu/~galles/visualization/AVLtree.html
- **Liang Data Structure Visualizations** www.cs.armstrong.edu/liang/animation/web/AVLTree.html
- **Iterative Tree Traversal**
- Wikipedia Threaded binary tree
- Stack Overflow Iterative inOrder traversal
- Python iterative traversals
- Stack Overflow Iterative pre and post
- Stack Overflow postOrder iterative
- GeeksforGeeks Iterative inOrder traversal
- GeeksforGeeks Iterative postOrder traversal 1
- GeeksforGeeks Iterative postOrder traversal 2
- Wikipedia: Catalan numbers
- Richard P Stanley: Enumerative Combinatorics
- Richard P Stanley: Catalan Addendum
- Mike Spivey Catalan Numbers from Their Generating Function
- Frazer Jarvis: Catalan Numbers

- Generating function
- Finding the generating function for the Catalan number sequence
- Cauchy product
- Spivak (2008, p486,p493,p513)
- Wilf (1994, p44,p53)
- Binary Tree — Interview Questions and Practice Problems
- GeeksforGeeks: Google Interview Questions
- Haskell Wiki: 99 questions: 54A to 60 Binary trees
- Haskell Wiki: 99 questions: 61 to 69 Binary trees (contd)
- Enumeration of Binary Trees
- Hackage: Math.Combinat.Trees.Binary
- Data Structures & Algorithms I Actually Used Working at Tech Companies
- I Don't Want To Hire You If You Can't Reverse a Binary Tree
- Invert Binary Tree: Recursive and Iterative solution
- Wikipedia: Purely functional data structure
- What's new in purely functional data structures since Okasaki?
- 24 Days of GHC Extensions: View Patterns — 24 Days of GHC Extensions
- GHC Extensions: View patterns
- GHC Wiki: View patterns

References

Adams, Stephen (1993). *Functional Pearls* Efficient sets — a balancing act. *Journal of Functional Programming*, 3(04):553–561. URL http://groups.csail.mit.edu/mac/users/adams/BB/.

Adelson-Velskii, G M and E M Landis (1962). An algorithm for the organization of information. In *Doklady Akademia Nauk SSSR*, volume 146, pages 263–266. Translated from *Soviet Mathematics — Doklady*; 3(5), 1259–1263.

Aho, Alfred V; John E Hopcroft; and Jeffrey D Ullman (1983). *Data Structures and Algorithms*. Addison-Wesley. ISBN 0201000237.

Bird, Richard (1998). *Introduction to Functional Programming using Haskell*. Prentice Hall, second edition. ISBN 0134843460.

Bird, Richard (2014). *Thinking Functionally with Haskell*. Cambridge University Press. ISBN 1107452643. URL http://www.cs.ox.ac.uk/publications/books/functional/.

Bird, Richard and Phil Wadler (1988). *Introduction to Functional Programming*. Prentice Hall, first edition. ISBN 0134841972.

Blelloch, Guy E; Daniel Ferizovic; and Yihan Sun (2016). Just Join for Parallel Ordered Sets. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 253–264. ACM.

Burge, William H. (1975). *Recursive Programming Techniques*. Addison-Wesley. ISBN 0201144506.

Cormen, Thomas H.; Charles E. Leiserson; Ronald L. Rivest; and Clifford Stein (2009). *Introduction to Algorithms*. MIT Press, third edition. ISBN 0262533057. URL http://mitpress.mit.edu/books/introduction-algorithms.

Dijkstra, Edsger W (1968). Letters to the editor: Go To Statement Considered Harmful. *Communications of the ACM*, 11(3):147–148.

Graham, Ronald L.; Donald Knuth; and Oren Patashnik (1994). *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, second edition. ISBN 0201558025.

Greene, Daniel H and Donald E Knuth (1982). *Mathematics for the Analysis of Algorithms*. Birkhauser, second edition. ISBN 376433102X.

Hudak, Paul; John Hughes; Simon Peyton Jones; and Phil Wadler (2007). A History of Haskell: Being Lazy with Class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1–12–55. ACM New York, NY, USA.

King, David J and John Launchbury (1995). Structuring depth-first search algorithms in Haskell. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 344–354. ACM.

Knuth, D.E. (1998). *The Art of Computer Programming Vol. 3: Sorting and Searching*. The Art of Computer Programming: Sorting and Searching. Adddison Wesley, second edition. ISBN 0201896850. URL http://books.google.co.uk/books?id=sXa_mwEACAAJ.

Knuth, Donald E. (1997). *The Art of Computer Programming Vol. 1: Fundamental Algorithms*. Addison Wesley, third edition. ISBN 0201896834.

Knuth, Donald E. (2011). *The Art of Computer Programming Vol. 4A: Combinatorial Algorithms, Part 1*. Addison Wesley, first edition. ISBN 0201038048.

Lee, Gias Kay (2013). Functional Programming in 5 Minutes. Web. http://gsklee.im, URL http://slid.es/gsklee/functional-programming-in-5-minutes.

Lutz, Mark (2011). *Programming Python*. O'Reilly, fourth edition. ISBN 0596158106. URL http://learning-python.com/books/about-pp4e.html.

Lutz, Mark (2013). *Learning Python*. O'Reilly, fifth edition. ISBN 1449355730. URL http://learning-python.com/books/about-lp5e.html.

Manber, Udi (1989). *Introduction to Algorithms: A Creative Approach*. Addison-Wesley. ISBN 0201120372.

Marlow, Simon and Simon Peyton Jones (2010). Haskell Language and Library Specification. Web. URL http://www.haskell.org/haskellwiki/Language_and_library_specification.

Meijer, Erik; Maarten Fokkinga; and Ross Paterson (1991). Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, pages 124–144. Springer.

Miller, Bradley W. and David L. Ranum (2011). *Problem Solving with Algorithms and Data Structures Using Python*. Franklin, Beedle Associates Inc, second edition. ISBN 1590282574. URL http://interactivepython.org/courselib/static/pythonds/index.html.

O'Donnell, John; Cordelia Hall; and Rex Page (2006). *Discrete Mathematics Using a Computer*. Springer, second edition. ISBN 1846282411. URL http://www.dcs.gla.ac.uk/~jtod/discrete-mathematics/.

Okasaki, Chris (1998). *Purely Functional Data Structures*. Cambridge University Press. ISBN 0-521-63124-6.

O'Sullivan, Bryan; John Goerzen; and Donald Stewart (2008). *Real World Haskell*. O'Reilly, first edition. ISBN 0596514980. URL http://book.realworldhaskell.org/.

Ottmann, Thomas and Derick Wood (1980). 1-2 brother trees or AVL trees revisited. *The Computer Journal*, 23(3):248–255.

Perlis, Alan J. (1982). Epigrams on Programming. *SIGPLAN Notices*, 17(9):7–13.

Sedgewick, Robert and Kevin Wayne (2011). *Algorithms*. Addison Wesley, fourth edition. ISBN 032157351X. URL http://algs4.cs.princeton.edu/home/.

Spivak, Michael (2008). *Calculus*. Publish or Perish, fourth edition. ISBN 0914098918. URL http://www.mathpop.com/.

Steele, Guy L., Jr. (2009a). Organizing functional code for parallel execution or, foldl and foldr considered slightly harmful. *SIGPLAN Not.*, 44(9):1–2. ISSN 0362-1340. doi: 10.1145/1631687.1596551. URL http://doi.acm.org/10.1145/1631687.1596551.

Steele, Guy L., Jr. (2009b). Organizing functional code for parallel execution or, foldl and foldr considered slightly harmful. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 1–2. ACM, New York, NY, USA. ISBN 978-1-60558-332-7. doi:10.1145/1596550.1596551. URL http://doi.acm.org/10.1145/1596550.1596551.

Thompson, Simon (2011). *Haskell the Craft of Functional Programming*. Addison Wesley, third edition. ISBN 0201882957. URL http://www.haskellcraft.com/craft3e/Home.html.

van Rossum, Guido and Fred Drake (2011a). *An Introduction to Python*. Network Theory Limited, revised edition. ISBN 1906966133.

van Rossum, Guido and Fred Drake (2011b). *The Python Language Reference Manual*. Network Theory Limited, revised edition. ISBN 1906966141.

Weiss, Mark Allen (2012). *Data Structures and Algorithm Analysis in Java*. Pearson Education. ISBN 0273752111. URL http://users.cis.fiu.edu/~weiss/.

Wilf, Herbert S (1994). *Generatingfunctionology*. Academic Press. ISBN 0127519564.

12 Blank Screen