

Binary Trees

M269 Tutorial

Contents

Commentary 1	3
1 Agenda	3
2 Adobe Connect	5
2.1 Interface	5
2.2 Settings	6
2.3 Sharing Screen & Applications	7
2.4 Ending a Meeting	8
2.5 Invite Attendees	8
2.6 Layouts	9
2.7 Chat Pods	10
2.8 Web Graphics	10
2.9 Recordings	11
Commentary 2	11
3 Binary Trees	11
3.1 Terminology	12
3.2 Examples	12
Activity 1 Binary Tree Types	14
3.3 Representation	16
3.3.1 Python from 21J	16
3.3.2 Alternate Representation	17
3.3.3 Operations	18
Activity 2 Python Representation	20
3.4 Tree Traversals	22
3.5 Depth First	22
Activity 3 Depth First Traversals	23
3.6 Breadth First	24
Activity 4 Breadth First Traversals	24
3.6.1 Breadth First V01	25
3.6.2 Breadth First V02	28
3.6.3 Breadth First V03	28
3.6.4 Breadth First V04	29
Activity 5 Breadth First Error	29
3.6.5 Breadth First V05	31
4 Iterative Traversals	31
4.1 Iterative InOrder Traversal	32
4.2 Iterative PreOrder Traversal	33
4.3 Iterative PostOrder Traversal	34
Commentary 3	35

5	Binary Search Trees	35
5.1	Definition	35
	Activity 6 Nodes of Perfect Tree	36
5.2	Inserting a Node	36
	Activity 7 Inserting an Item	37
5.3	BST Operations	39
	Activity 8 Membership	39
	5.3.1 Testing a BST	39
	5.3.2 List to BST	40
	Activity 9 Insert List	40
5.4	BST Deletion	43
	Activity 10 joinBST Version 1	45
	Activity 11 joinBST Version 2	46
	Activity 12 Split Trace	49
	Activity 13 Join Trace	49
	Activity 14 Delete Trace	50
	Commentary 4	51
6	AVL Trees	51
6.1	AVL Trees and Functions	52
6.2	Example AVL Trees	53
	Activity 15 Height and Balance Factors	53
	Activity 16 Add Item LL	54
6.3	AVL Transformations	55
	Activity 17 Add Item RR	56
	Activity 18 Rebalance RR	57
	Activity 19 egBSTree05 Add Item LR 1	59
	Activity 20 Add Item LR 2	60
	Activity 21 Case RL Heavy	63
6.4	makeAVLTree Function	64
	6.4.1 Comparison with Storing Balance Factors	66
6.5	Insertion and Deletion	68
	Activity 22 Insert Lists and Delete Items	69
	Activity 23 Deleting Inserted List	75
	Activity 24 Delete with Rebalance	77
6.6	AVL Tree Performance	82
	6.6.1 Proof of Euler-Binet Formula	84
7	AVL Trees: Sets	85
7.1	Set Representation	86
	7.1.1 Split, Join	86
	Activity 25 joinLeftAVLS Diagrams	89
	Activity 26 joinLeftAVLS Bug	89
7.2	Set Operations	90
	Activity 27 Set to Ascending List	92
	Activity 28 Set Equality	92
	Activity 29 Subset	93
7.3	Sets — Implementation Points	93
	Commentary 5	94

8 Binary Tree Exercises	95
8.1 Binary Tree Shapes	95
Activity 30 Shape Exercises	95
8.1.1 Isomorphic Binary Trees	96
8.2 Generating Binary Trees	96
8.3 Catalan Numbers	98
8.3.1 Cauchy Product	99
8.3.2 Catalan Recurrence	99
8.3.3 Sample Catalan Numbers	102
Commentary 6	102
9 Future Work	102
10 References	102
10.1 Python Web Links & References	102
10.2 Haskell Web Links & References	103
References	103
Python Code Index	107
Diagrams Index	109

Commentary 1

1 Agenda, Aims and Topics

- Overview of aims of tutorial
- Note selection of topics
- Recursion is used throughout the topics
- Points about my own background and preferences
- Adobe Connect slides for reference

[ToC](#)

1 M269 Tutorial Agenda — Binary Trees, Recursion, Searching

1. Welcome and introductions
2. Material on Binary Trees and Searching
3. Implementation in Python
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)

8. *Source*: of slides, notes, programs:

pmolyneux.co.uk/OU/M269FolderSync/M269TutorialSamples2023/

M269TutorialBinaryTreesCmnty2023/

- There is a lot more material in these slides/notes than we can cover in the available time, so I will cover:

(1) Binary Tree terminology and representation — some choices

(2) Tree traversal — depth first recursive

(3) Tree traversal — breadth first — recursive first, transformed to the usual iterative version

- These notes are as much about recursion as Binary trees — the notes give several examples of evaluations and what to do when you make a mistake

(4) Binary search trees — deleting a node — choices

(5) AVL or height balanced trees — brief introduction

Health Warning These notes contain some material that is not part of M269 but is present for interest

Tutorial Materials

- From the Web link to the folder containing the tutorial materials you should find:
- File with name ending [.beamer.pdf](#) — the slides
- File with name ending [.article.pdf](#) — the notes version
- Table of contents — in the slides this is a clickable sidebar; in the notes it is an expanded list of sections with links from the end of sections
- Indices — the notes version has an index of the Python code and the diagrams
- References — the notes version has references which have back references to the pages where the reference is cited

Introductions — Phil

- *Name* Phil Molyneux
- *Background*
 - Undergraduate: Physics and Maths (Sussex)
 - Postgraduate: Physics (Sussex), Operational Research (Brunel), Computer Science (University College, London)

- Worked in Operational Research, Business IT, Web technologies, Functional Programming
- *First programming languages* [Fortran](#), [BASIC](#), [Pascal](#)
- *Favourite Software*
 - [Haskell](#) — pure functional programming language
 - Text editors [TextMate](#), [Sublime Text](#) — previously [Emacs](#)
 - Word processing in [L^AT_EX](#) — all these slides and notes
 - [Mac OS X](#)
- *Learning style* — I read the manual before using the software

Introductions — You

- *Name ?*
- *Favourite software/Programming language ?*
- *Favourite [text editor](#) or [integrated development environment \(IDE\)](#)*
- [List of text editors](#), [Comparison of text editors](#) and [Comparison of integrated development environments](#)
- *Other OU courses ?*
- *Anything else ?*

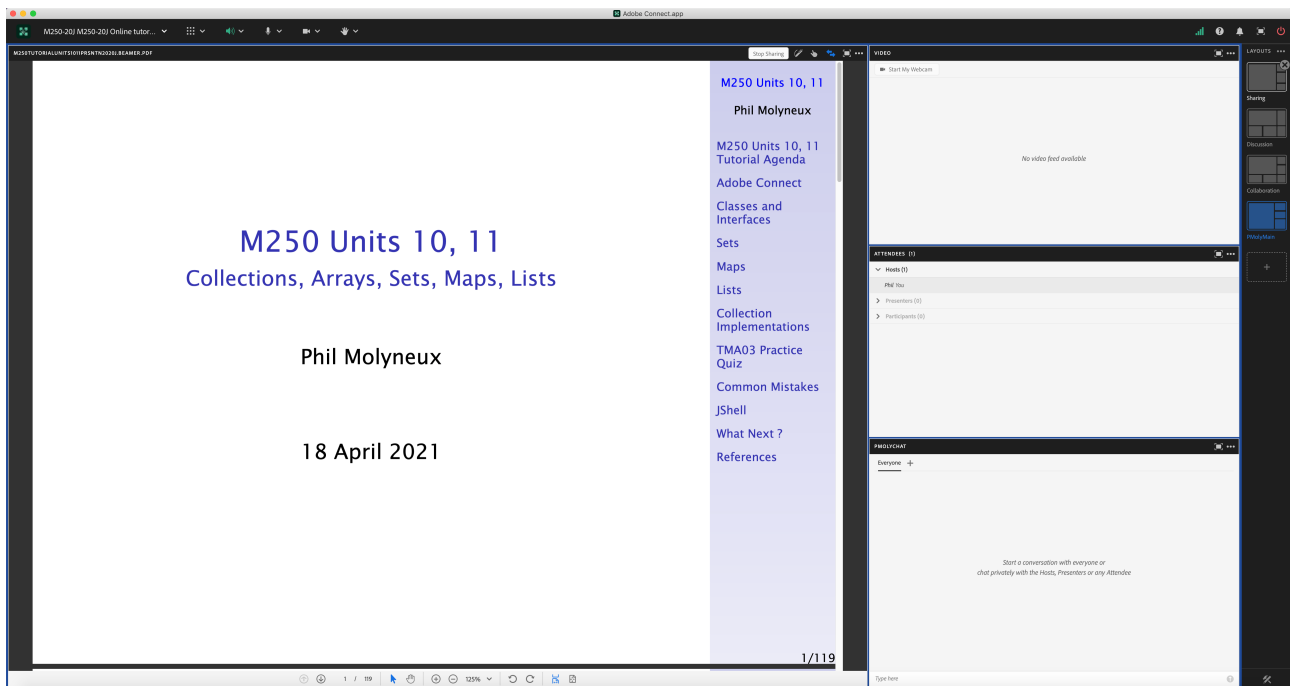
[Go to Table of Contents](#)

[ToC](#)

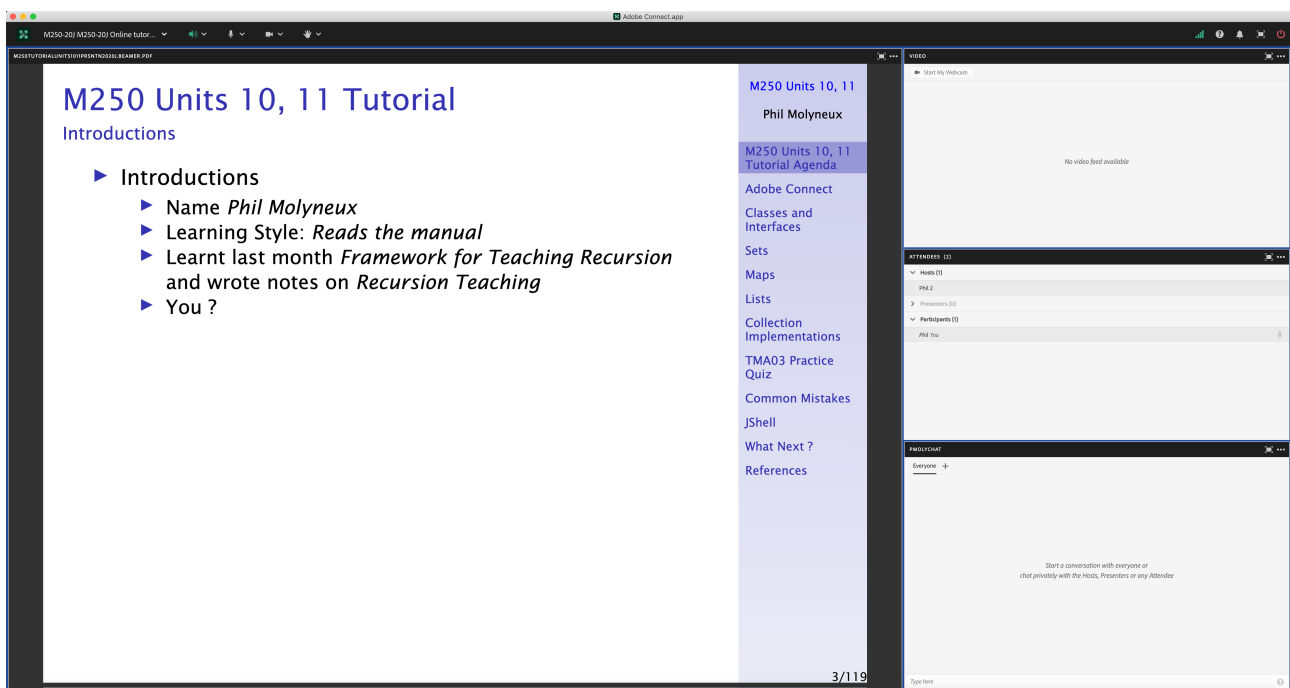
2 Adobe Connect Interface and Settings

2.1 Adobe Connect Interface

Adobe Connect Interface — Host View



Adobe Connect Interface — Participant View



2.2 Adobe Connect Settings

Adobe Connect — Settings

- **Everybody** **Menu bar** **Meeting** **Speaker & Microphone Setup**
- **Menu bar** **Microphone** **Allow Participants to Use Microphone** ✓
- Check Participants see the entire slide including slide numbers bottom right **Workaround**
 - **Disable Draw** **Share pod** **Menu bar** **Draw icon**
 - **Fit Width** **Share pod** **Bottom bar** **Fit Width icon** ✓

- **Meeting** > Preferences > General > **Host Cursor** > Show to all attendees
- **Menu bar** > Video > Enable **Webcam** for Participants ✓
- Do not *Enable single speaker mode*
- Cancel hand tool
- Do not enable green pointer
- **Recording** > Meeting > Record Session ✓
- **Documents** Upload PDF with drag and drop to share pod
- Delete > Meeting > Manage Meeting Information > Uploaded Content and > check filename > click on delete

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 — 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 blue 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**

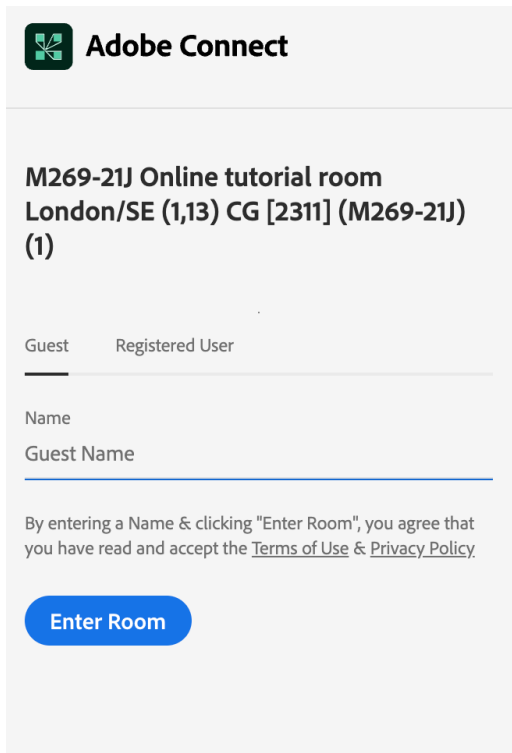
2.4 Adobe Connect — Ending a Meeting

- *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.
- **Delete File Upload** **Meeting** > **Manage Meeting Information** > **Uploaded Content tab** select file(s) and click **Delete**
- **Attendance Report** see course Web site for joining room

2.5 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](#)
- Click on the link sent in email from the Host
- Get the following on a Web page

- As *Guest* enter your name and click on **Enter Room**



Adobe Connect

M269-21J Online tutorial room
London/SE (1,13) CG [2311] (M269-21J)
(1)

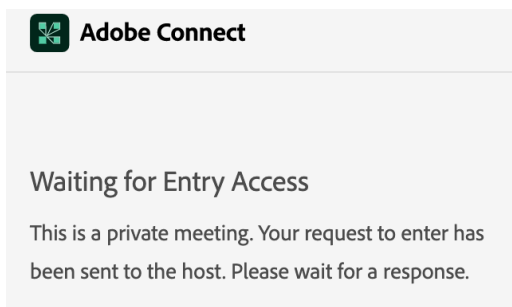
Guest Registered User

Name
 Guest Name

By entering a Name & clicking "Enter Room", you agree that you have read and accept the [Terms of Use](#) & [Privacy Policy](#).

Enter Room

- See the *Waiting for Entry Access for Host* to give permission

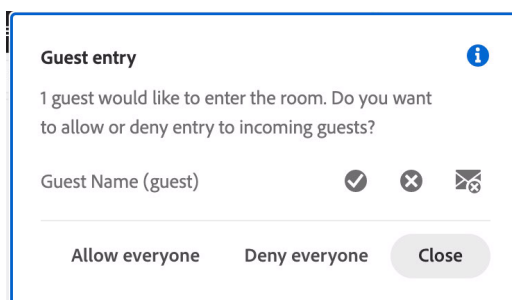


Adobe Connect

Waiting for Entry Access

This is a private meeting. Your request to enter has been sent to the host. Please wait for a response.

- *Host* sees the following dialog in *Adobe Connect* and grants access



Guest entry

1 guest would like to enter the room. Do you want to allow or deny entry to incoming guests?

Guest Name (guest) ✓ ✕ 🔒

Allow everyone Deny everyone Close

2.6 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 *PMolyShare*
- **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 — rename 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)
- **Auxiliary Layouts** name *PMolyAuxOn*
 - Create new Share pod
 - Use existing Chat pod
 - Use same Video and Attendance pods

2.7 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**

2.8 Graphics Conversion for Web

- Conversion of the screen snaps for the installation of Anaconda on 1 May 2020
- Using GraphicConverter 11
- **File** > **Convert & Modify** > **Conversion** > **Convert**
- Select files to convert and destination folder
- Click on **Start selected Function** or **⌘ + ⌘**

2.9 Adobe Connect Recordings

- [Menu bar](#) > [Meeting](#) > [Preferences](#) > [Video](#)
- [Aspect ratio](#) > [Standard \(4:3\)](#) (not Wide screen (16:9) default)
- [Video quality](#) > [Full HD](#) (1080p not High default 480p)
- **Recording** [Menu bar](#) > [Meeting](#) > [Record Session](#) ✓
- **Export Recording**
- [Menu bar](#) > [Meeting](#) > [Manage Meeting Information](#)
- [New window](#) > [Recordings](#) > [check Tutorial](#) > [Access Type button](#)
- [check Public](#) > [check Allow viewers to download](#)
- **Download Recording**
- [New window](#) > [Recordings](#) > [check Tutorial](#) > [Actions](#) > [Download File](#)

[ToC](#)

Commentary 2

2 Binary Trees

- Usage, terminology, example trees
- Representation, Abstract Data Types and notation
- Tree traversals, Depth First and Breadth First
- Recursive versions first
- Iterative versions derived from recursive versions
- Iterative depth first traversals for interest only
- Points on performance

[ToC](#)

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.

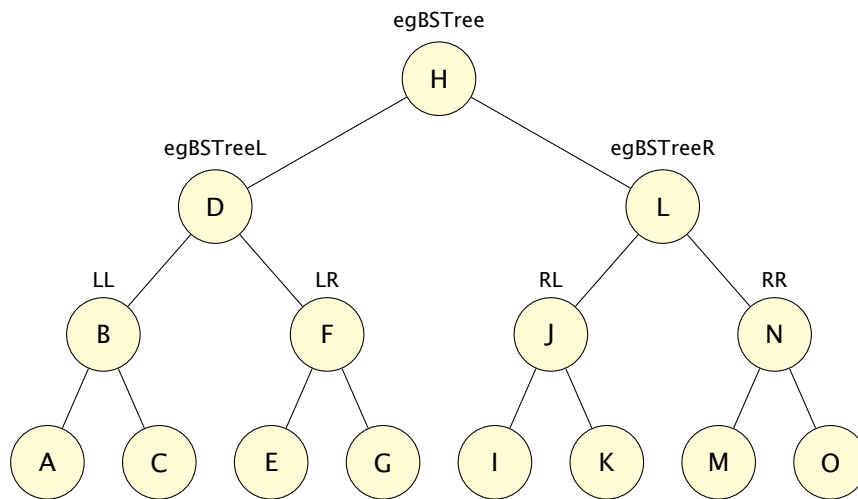
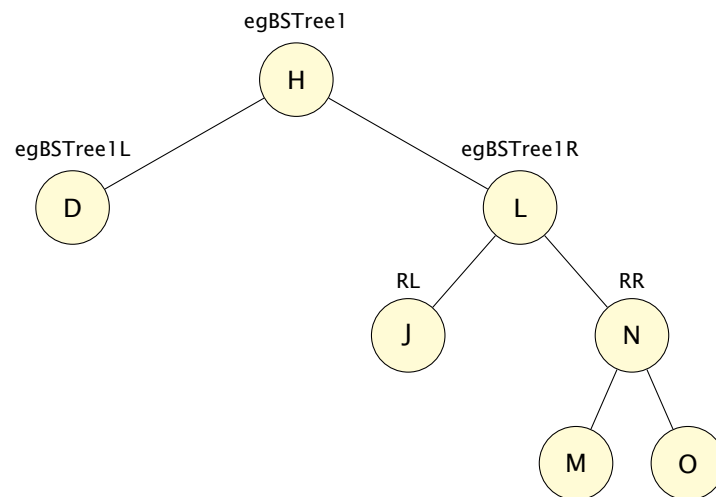
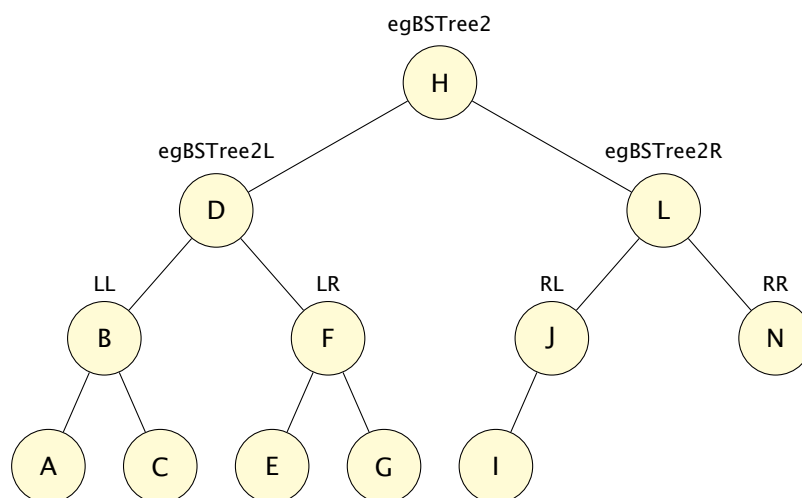
Binary Trees — Terminology

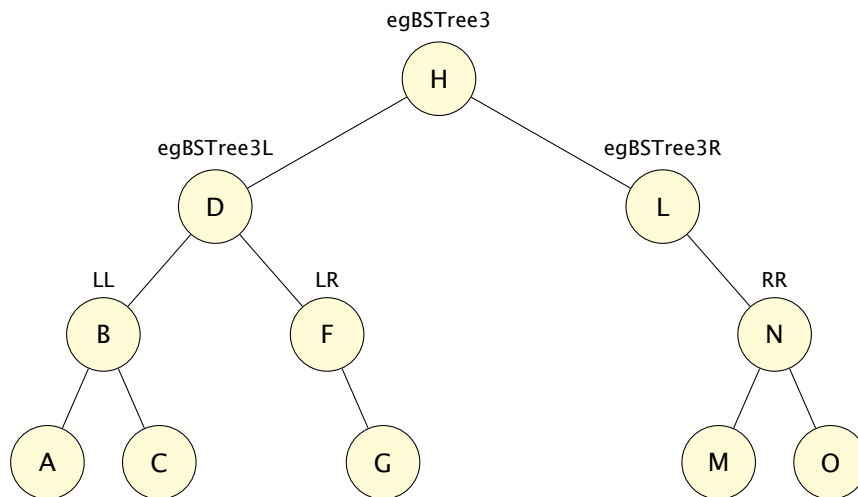
- [Children](#) — subtrees of a node that are not empty
- [Leaves](#) — nodes with two empty subtrees
- [Full Binary Tree](#) — every node other than the leaves has two non empty subtrees
- [Perfect Binary Tree](#) — all leaves are at the same level (or depth) children
- [Complete Binary Tree](#) — every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

[ToC](#)

3.2 Binary Tree Examples

Example [egBSTree](#)

**Example [egBSTree1](#)****Example [egBSTree2](#)****Example [egBSTree3](#)**



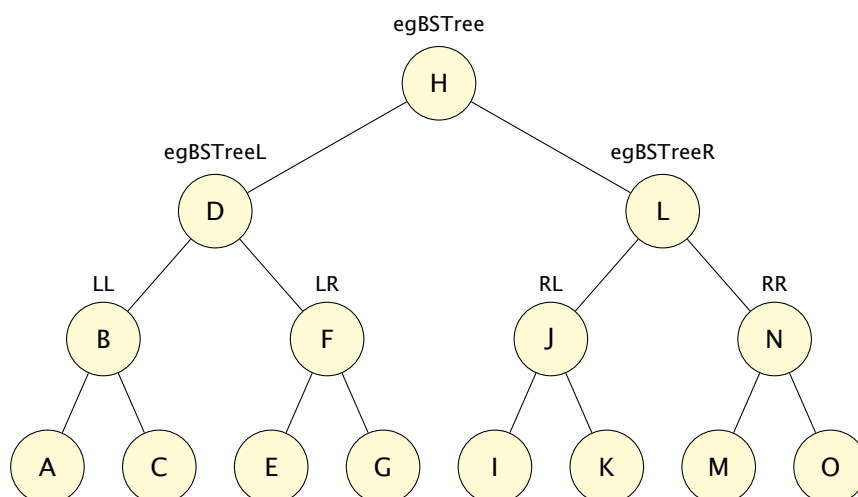
Activity 1 Binary Tree Types

- What types of trees are the above example trees ?
- [egBSTree](#)
- [egBSTree1](#)
- [egBSTree2](#)
- [egBSTree3](#)

[Go to Answer](#)

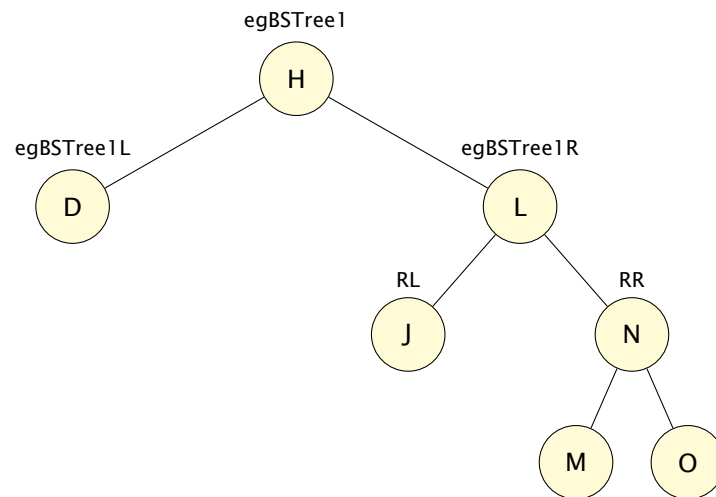
Answer 1 Binary Tree Types

- [egBSTree](#) — perfect



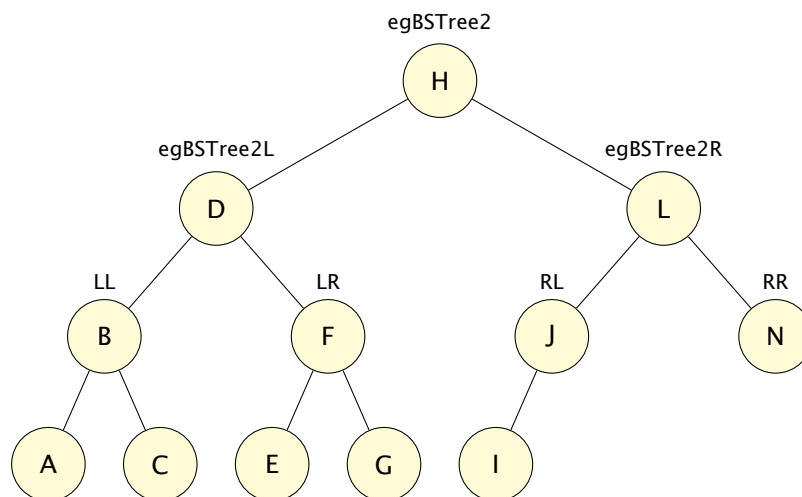
Answer 1 Binary Tree Types

- [egBSTree1](#) — full



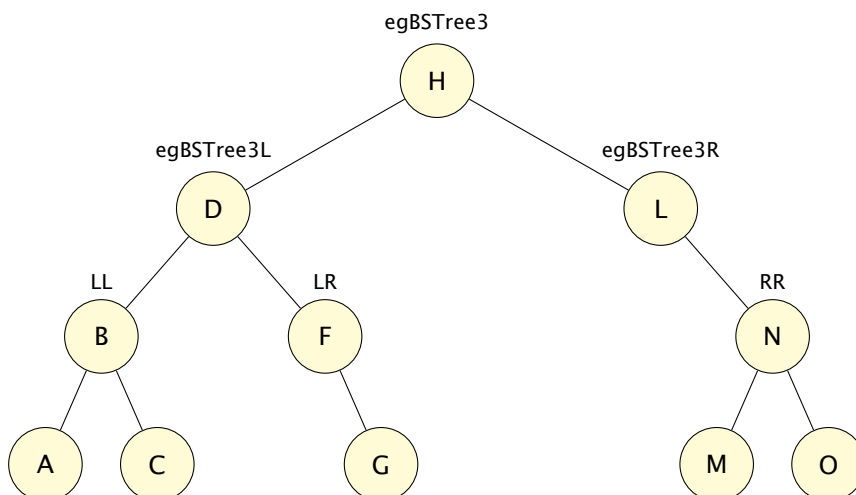
Answer 1 Binary Tree Types

- [egBSTree2](#) — complete



Answer 1 Binary Tree Types

- [egBSTree3](#) — just a binary tree



[Go to Activity](#)

3.3 Representation of Binary Trees

3.3.1 Python Representation from 2021J

- In 2021J M269 revision the Binary Tree Abstract Data Type (ADT) is represented by the following Python *Class*
- The code is in the M269 Jupyter Notebooks and the provided file [m269_trees.py](#)
- The code is reproduced in the file [M269BinaryTrees2021J.py](#) but, for brevity, without the [docstrings](#)

```

10 class Tree :
11
12     def __init__(self) :
13         self.root = None
14         self.left = None
15         self.right = None
16
17     def is_empty(tree: Tree) -> bool :
18         return (tree.root == tree.left == tree.right
19                 == None)
20
21     def join(item: object, left: Tree, right: Tree) -> Tree :
22         tree = Tree()
23         tree.root = item
24         tree.left = left
25         tree.right = right
26         return tree

```

- The functions [is_leaf](#), [size](#), [height](#)

```

28 def is_leaf(tree: Tree) -> bool :
29     return (not is_empty(tree)
30             and is_empty(tree.left) and is_empty(tree.right))
31
32 def size(tree: Tree) -> int :
33     if is_empty(tree) :
34         return 0
35     else :
36         return (size(tree.left) + size(tree.right) + 1)
37
38 def height(tree: Tree) -> int :
39     if is_empty(tree) :
40         return 0
41     else :
42         return (max(height(tree.left), height(tree.right)) + 1)

```

- This representation works (see the M269 book) but has the slight disadvantage in that it has no default print representation that is useful
- For example, here is what happens when we attempt to print a very small tree — [threeBT](#) is the same as [THREE](#) in the chapter

```

Python3>>> threeBT = join(3,Tree(),Tree())
Python3>>> threeBT
<M269BinaryTrees2021J.Tree object at 0x10095a0a0>
Python3>>>

```

- We *could* write a method to implement a print representation of an instance of the class [Tree\(\)](#) but that might be a lot of code
- The main point of having an Abstract Data Type (ADT) is we can swap out the underlying implementation for another one
- This might be done for efficiency reasons but here we do it to get an underlying type with a default print representation

- All we have to do is keep operations which provide access to the underlying representation
- *This is called a learning opportunity!*

[ToC](#)

3.3.2 Python Alternate Representation

- We first list the operations which will (or might) need to have direct access to the underlying representation
- Make an empty tree
- Construct a new tree from an item and two trees
- Query if a given tree is an empty tree
- Given a tree, return the left sub tree
- Given a tree, return the right sub tree
- Find the height of a tree
- Find the size of a tree
- The last two do not *need* access to the underlying representation (we can calculate the size and height with just the other operations) but as we will see later, we might give access for efficiency reasons
- To fully hide the ADT implementation we give common function names to the operations

Tree Class	Common Name	Category
<code>Tree()</code>	<code>mkEmptyBT()</code>	Constructor
<code>join()</code>	<code>mkNodeBT()</code>	Constructor
<code>is_empty()</code>	<code>isEmptyBT()</code>	Inspector
<code>tree.root</code>	<code>getDataBT()</code>	Destructor
<code>tree.left</code>	<code>getLeftBT()</code>	Destructor
<code>tree.right</code>	<code>getRightBT()</code>	Destructor
<code>height()</code>	<code>heightBT()</code>	Operation
<code>size()</code>	<code>sizeBT()</code>	Operation

- The functions labelled Constructor, Inspector, Destructor are operations that have direct access to the underlying representation
- `sizeBT()` and `heightBT()` are just ordinary operations in this version of the Tree ADT but for efficiency reasons they may become Inspectors in a later version
- 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.

- It has a default print representation

```

7  from collections import namedtuple
9  EmptyBT = namedtuple('EmptyBT', [])
11 NodeBT = namedtuple('NodeBT'
12                    , ['dataBT', 'leftBT', 'rightBT'])

```

- The Python code above is in the file [Python/M269TutorialBinaryTrees2022.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**
- Python built-in functions in **this color**
- User defined data constructors and functions such as **NodeBT** and **EmptyBT** are shown in **that color**
- **Health Warning:** these notes may not be totally consistent with syntax colouring.
- We declare the Python type for a *Union* type since a Tree is either an empty tree or a non-empty tree
- This is venturing into some of the areas of Python Type Annotations that feel rather awkward but we shall use them in a simple way
- Remember that the Python interpreter only checks the type annotations for validity but not for correctness — they just have to look like proper types but the processor does *not* check them

```

14  # Tree type
16  from typing import TypeVar, Union, NewType
18  T = TypeVar('T')
19  Tree = NewType('Tree', Union[EmptyBT, NodeBT]).

```

- Note that using **namedtuple** means that all items are assumed to have **Any** types (see [mypy: Named tuples](#))
- You *could* use **NamedTuple** which is a typed version of namedtuple but this would be getting a lot more complicated than types as used in M269
- in particular you would get involved in specifying **user-defined generic types** and **forward references** (since the Tree data type is recursive)
- We could have avoided **Union** by just having **NodeBT** and representing an empty tree by the Python **None**
- This would be isomorphic to the Class version with default printing

[ToC](#)

3.3.3 Operations

- We now provide functions to create, inspect and take apart binary trees
- The code with the line numbers is the code for the implementation using namedtuples

```

23 def mkEmptyBT() -> Tree :
24     return EmptyBT()

26 def mkNodeBT(x : T, t1 : Tree, t2 : Tree) -> Tree :
27     return NodeBT(x, t1, t2)

29 def isEmptyBT(t : Tree) -> bool:
30     return t == EmptyBT()

```

- `mkEmptyBT`, `mkNodeBT` 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: should I use is or ==?](#)
- The code with no line numbers illustrates how the previous implementation using Class `Tree` can be given the same operations interface

```

def mkEmptyBT() -> Tree :
    return Tree()

def mkNodeBT(x : T, t1 : Tree, t2 : Tree) -> Tree :
    return join(x, t1, t2)

def isEmptyBT(t : Tree) -> bool:
    return is_empty(t)

```

- Here are the operations that access the parts of the tree

```

32 def getDataBT(t : Tree) -> T:
33     if isEmptyBT(t):
34         raise RuntimeError("getDataBT_applied_to_EmptyBT()")
35     else:
36         return t.dataBT

38 def getLeftBT(t : Tree) -> Tree :
39     if isEmptyBT(t):
40         raise RuntimeError("getLeftBT_applied_to_EmptyBT()")
41     else:
42         return t.leftBT

44 def getRightBT(t : Tree) -> Tree :
45     if isEmptyBT(t):
46         raise RuntimeError("getRightBT_applied_to_EmptyBT()")
47     else:
48         return t.rightBT

```

- The Class `Tree` implementation of the above operations

```

def getDataBT(t : Tree) -> T:
    if isEmptyBT(t):
        raise RuntimeError("getDataBT_applied_to_empty_tree")
    else:
        return t.root

def getLeftBT(t : Tree) -> Tree :
    if isEmptyBT(t):
        raise RuntimeError("getLeftBT_applied_to_empty_tree")
    else:
        return t.left

def getRightBT(t : Tree) -> Tree :
    if isEmptyBT(t):

```

```

    raise RuntimeError("getRightBT_applied_to_empty_tree")
else:
    return t.right

```

- Here are the operations `heightBT()` and `sizeBT()`
- Note that `height` of an empty tree is 0

```

59 def heightBT(t : Tree) -> int :
60     if isEmptyBT(t):
61         return 0
62     else:
63         return (1 + max(heightBT(getLeftBT(t)),
64                         heightBT(getRightBT(t))))
65
66 def sizeBT(t : Tree) -> int :
67     if isEmptyBT(t) :
68         return 0
69     else :
70         return (1 + sizeBT(getLeftBT(t))
71                 + sizeBT(getRightBT(t)))

```

- The Class Tree implementation of the above operations is exactly the same
- If we make `height` or `size` directly part of the data structure this may change

[ToC](#)

Activity 2 Python Representation

- Write Python implementations of the following trees (from the diagrams above) using the named tuple `NodeBT` and `EmptyBT`
- egBSTree
- egBSTree1
- egBSTree2
- egBSTree3

[Go to Answer](#)

Answer 2 Python Representation — egBSTree

```

70 egBSTree = mkNodeBT('H',
71                     mkNodeBT('D',
72                             mkNodeBT('B',
73                                     mkNodeBT('A',mkEmptyBT(),mkEmptyBT()),
74                                     mkNodeBT('C',mkEmptyBT(),mkEmptyBT())
75                                     ),
76                             mkNodeBT('F',
77                                     mkNodeBT('E',mkEmptyBT(),mkEmptyBT()),
78                                     mkNodeBT('G',mkEmptyBT(),mkEmptyBT())
79                                     ),
80                             ),
81                     mkNodeBT('L',
82                             mkNodeBT('J',
83                                     mkNodeBT('I',mkEmptyBT(),mkEmptyBT()),
84                                     mkNodeBT('K',mkEmptyBT(),mkEmptyBT())
85                                     ),
86                             mkNodeBT('N',
87                                     mkNodeBT('M',mkEmptyBT(),mkEmptyBT()),
88                                     mkNodeBT('O',mkEmptyBT(),mkEmptyBT())
89                                     ),
90                             ),
91                     )

```

Answer 2 Python Representation — egBSTree1

```

195 egBSTree1 = mkNodeBT('H',
196                     mkNodeBT('D',mkEmptyBT(),mkEmptyBT()),
197                     mkNodeBT('L',
198                             mkNodeBT('J',mkEmptyBT(),mkEmptyBT()),
199                             mkNodeBT('N',
200                                     mkNodeBT('M',mkEmptyBT(),mkEmptyBT()),
201                                     mkNodeBT('O',mkEmptyBT(),mkEmptyBT())
202                             )
203                     )
204 )

```

Answer 2 Python Representation — egBSTree2

```

221 egBSTree2 = mkNodeBT('H',
222                     mkNodeBT('D',
223                             mkNodeBT('B',
224                                     mkNodeBT('A',mkEmptyBT(),mkEmptyBT()),
225                                     mkNodeBT('C',mkEmptyBT(),mkEmptyBT())
226                             ),
227                             mkNodeBT('F',
228                                     mkNodeBT('E',mkEmptyBT(),mkEmptyBT()),
229                                     mkNodeBT('G',mkEmptyBT(),mkEmptyBT())
230                             )
231                     ),
232                     mkNodeBT('L',
233                             mkNodeBT('J',
234                                     mkNodeBT('I',mkEmptyBT(),mkEmptyBT()),
235                                     mkEmptyBT()
236                             ),
237                             mkNodeBT('N',mkEmptyBT(),mkEmptyBT())
238                     )
239 )

```

Answer 2 Python Representation — egBSTree3

```

265 egBSTree3 = mkNodeBT('H',
266                     mkNodeBT('D',
267                             mkNodeBT('B',
268                                     mkNodeBT('A',mkEmptyBT(),mkEmptyBT()),
269                                     mkNodeBT('C',mkEmptyBT(),mkEmptyBT())
270                             ),
271                             mkNodeBT('F',
272                                     mkEmptyBT(),
273                                     mkNodeBT('G',mkEmptyBT(),mkEmptyBT())
274                             )
275                     ),
276                     mkNodeBT('L',
277                             mkEmptyBT(),
278                             mkNodeBT('N',
279                                     mkNodeBT('M',mkEmptyBT(),mkEmptyBT()),
280                                     mkNodeBT('O',mkEmptyBT(),mkEmptyBT())
281                             )
282                     )
283 )

```

[Go to Activity](#)

Answer 2 Python Representation — egBSTreeL

```

122 egBSTreeL = mkNodeBT('D',
123                     mkNodeBT('B',
124                             mkNodeBT('A',mkEmptyBT(),mkEmptyBT()),
125                             mkNodeBT('C',mkEmptyBT(),mkEmptyBT())
126                     ),
127                     mkNodeBT('F',
128                             mkNodeBT('E',mkEmptyBT(),mkEmptyBT()),
129                             mkNodeBT('G',mkEmptyBT(),mkEmptyBT())
130                     )
131 )

```

[Go to Activity](#)

Answer 2 Python Representation — `egBSTreeLL`

```

146 egBSTreeLL = mkNodeBT('B',
147                       mkNodeBT('A',mkEmptyBT(),mkEmptyBT()),
148                       mkNodeBT('C',mkEmptyBT(),mkEmptyBT()))
149 
```

[Go to Activity](#)

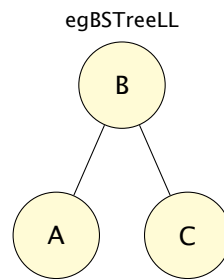
3.4 Binary Tree Traversals

- 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

[ToC](#)

3.5 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*
- We first provide the usual recursive implementations — in a later section we translate the recursive versions into iterative versions
- Tree `egBSTreeLL` Python code at line 146 on page 22



- 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']
  
```

```

311 def inOrderBT(t) :
312     if isEmptyBT(t) :
313         return []
314     else :
315         return (inOrderBT(getLeftBT(t)) + [getDataBT(t)]
316               + inOrderBT(getRightBT(t)))
317
318 def preOrderBT(t) :
319     if isEmptyBT(t) :
320         return []
321     else :
322         return ([getDataBT(t)] + preOrderBT(getLeftBT(t))
323               + preOrderBT(getRightBT(t)))
324
325 def postOrderBT(t) :
326     if isEmptyBT(t) :
327         return []
328     else :
329         return (postOrderBT(getLeftBT(t))
330               + postOrderBT(getRightBT(t)) + [getDataBT(t)])
  
```

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 311 on page 23 (Python)
- Binary tree code is from line 70 on page 20 (Python)

Ans 3

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']
```

[Act 3](#)
[ToC](#)

3.6 Tree Traversals — Breadth First

- The M269 book section 16.3.5 gives an iterative version of a breadth first traversal but only mentions a recursive version briefly
- We shall start with a recursive version and transform that by stages into the iterative version in the book
- I find it easier to think of the recursive version first — you should observe how people think they think about programming
- First we do some exercises

Activity 4 Breadth First Traversals

- A *level order* traversal of a binary tree takes a tree and returns the list of levels
- Each level is the list of items at that level
- Give the list of levels for:
 - `egBSTree`
 - `egBSTreeL`
 - `egBSTree1`
 - `egBSTree2`

- [egBSTree3](#)

[Go to Answer](#)

Answer 4 Breadth First Traversals

- Answer 4 Breadth First Traversals

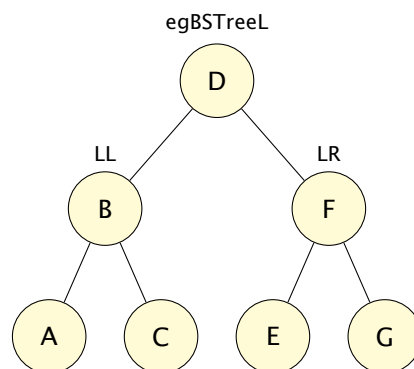
```
Python3>>> levelOrderBT(egBSTree)
[['H'], ['D', 'L'], ['B', 'F', 'J', 'N'],
 ['A', 'C', 'E', 'G', 'I', 'K', 'M', 'O']]
Python3>>> levelOrderBT(egBSTreeL)
[['D'], ['B', 'F'], ['A', 'C', 'E', 'G']]
Python3>>> levelOrderBT(egBSTree1)
[['H'], ['D', 'L'], ['J', 'N'], ['M', 'O']]
Python3>>> levelOrderBT(egBSTree2)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

[Go to Activity](#)

Answer 4 Breadth First Traversals

- Answer 4 Breadth First Traversals

```
Python3>>> levelOrderBT(egBSTreeL)
[['D'], ['B', 'F'], ['A', 'C', 'E', 'G']]
Python3>>>
```



3.6.1 Breadth First V01

- The first version will be recursive and driven by the structure of trees and will start by writing `levelOrder()` which takes a binary tree and returns a list of levels — a `level` is a list of items at the level
- (1) An empty tree has an empty list of levels (level zero)
 - (2) A non-empty tree has the list of the root item followed by combining the two lists of the levels for the two sub-trees
- We will call the function that combines the two lists of levels `longZipMerge()` since it is similar to the Python library `zip()` function

```
428 def levelOrderBT(t : Tree) -> [[T]] :
429     if isEmptyBT(t) :
430         return []
431     else :
432         x = getDataBT(t)
433         left = getLeftBT(t)
434         right = getRightBT(t)
435         return ([x] +
```

```
436 longZipMerge(levelOrderBT(left), levelOrderBT(right)))
```

- `longZipMerge()` is a variant on the Python library function `zip()`
- `zip()` iterates over several iterables in parallel, producing tuples with an item from each one.
- `longZipMerge()` takes two lists and returns a new list with merged pairs of items from each list which is a level order traverse of the subtrees
- The two lists do not need to be of the same length — any excess is just appended to the merged result so far

```
438 def longZipMerge(xss : [[T]], yss : [[T]]) -> [[T]] :
439     if xss == [] :
440         return yss
441     elif yss == [] :
442         return xss
443     else :
444         return ([xss[0] + yss[0]] + longZipMerge(xss[1:], yss[1:]))
```

- Evaluation of `levelOrderBT(egBSTreeL)`

```
levelOrderBT(egBSTreeLL)
= [['B']] # by line 431
+ longZipMerge(levelOrderBT(egBSTreeLLL),
               levelOrderBT(egBSTreeLLR))
= [['B']] # by lines 431, 439
+ longZipMerge(['A'], ['C'])
= [['B']] + [['A', 'C']] # by line 441
= [['B'], ['A', 'C']]
```

```
levelOrderBT(egBSTreeLR)
= [['F'], ['E', 'G']] # as above
```

```
levelOrderBT(egBSTreeL)
= [['D']] # by line 431
+ longZipMerge(levelOrderBT(egBSTreeLL),
               levelOrderBT(egBSTreeLR))
= [['D']] # as above
+ longZipMerge(['B'], ['A', 'C'],
               ['F'], ['E', 'G'])
= [['D']] # by line 441
+ [['B', 'F'], ['A', 'C', 'E', 'G']]
= [['D'], ['B', 'F'], ['A', 'C', 'E', 'G']] # correct - check the steps
```

- Testing can only show the presence of bugs but not the absence of bugs ([Edsger W Dijkstra Quotes](#))
- We shall now investigate a similar program with a subtle error

```
465 def levelOrderBT01(t : Tree) -> [[T]] :
466     if isEmptyBT(t) :
467         return []
468     else :
469         x = getDataBT(t)
470         left = getLeftBT(t)
471         right = getRightBT(t)
472         return ([x] +
473               longZipMerge01(levelOrderBT01(left), levelOrderBT01(right)))
475
476 def longZipMerge01(xss : [[T]], yss : [[T]]) -> [[T]] :
477     if xss == [] :
478         return yss
479     elif yss == [] :
480         return xss
481     else :
482         return ([xss[0], yss[0]] + longZipMerge01(xss[1:], yss[1:]))
```

- We first do a few tests

```
Python3>>> levelOrderBT(egBSTreeLL)
[['B'], ['A', 'C']]
Python3>>> levelOrderBT01(egBSTreeLL)
['B', 'A', 'C']
Python3>>> levelOrderBT(egBSTree1)
[['H'], ['D', 'L'], ['J', 'N'], ['M', 'O']]
Python3>>> levelOrderBT01(egBSTree1)
['H', 'D', 'L', 'J', 'N', 'M', 'O']
```

- Correct order but a list of items not a list of levels

```
Python3>>> levelOrderBT(egBSTreeL)
[['D'], ['B', 'F'], ['A', 'C', 'E', 'G']]
Python3>>> levelOrderBT01(egBSTreeL)
['D', 'B', 'F', 'A', 'E', 'C', 'G']
```

- Wrong order — we now do an evaluation to see where the error is
- Evaluation of `levelOrderBT01(egBSTreeL)`

```
levelOrderBT01(egBSTreeLL)
= ['B'] # by line 468
+ longZipMerge01(levelOrderBT01(egBSTreeLL),
                  levelOrderBT01(egBSTreeLL))
= ['B'] # by lines 468,476
+ longZipMerge01(['A'], ['C'])
= ['B'] + ['A', 'C'] # by line 478
= ['B', 'A', 'C']
```

```
levelOrderBT01(egBSTreeLR)
= ['F', 'E', 'G'] # as above
```

```
levelOrderBT01(egBSTreeL)
= ['D'] # by line 468
+ longZipMerge01(levelOrderBT01(egBSTreeLL),
                  levelOrderBT01(egBSTreeLR)))
= ['D'] # as above
+ longZipMerge01(['B', 'A', 'C'],
                  ['F', 'E', 'G'])
= ['D'] # by line 478
+ ['B', 'F', 'A', 'E', 'C', 'G']
= ['D', 'B', 'F', 'A', 'E', 'C', 'G'] # notice the error ?
```

- `levelOrderBT01()` is not only of the wrong type but produces the wrong order except for a limited number of trees
- The Python type annotations are only checked for syntax but not for correctness
- We get the final `breadthBT01()` by flattening the list of levels
- This uses a *list comprehension* as a shorthand for nested loops — see explanation below

```
454 def flattenLevels(levels : [[T]]) -> [T] :
455     return [elem for level in levels for elem in level]
457 def breadthBT01(t : Tree) -> [T] :
458     return flattenLevels(levelOrderBT(t))
```

- Python [List comprehensions](#) (tutorial), [List comprehensions](#) (reference) — a neat way of expressing iterations over a list
- Example (a) Square the even numbers between 0 and 9
- Example (b) Generate a list of pairs which satisfy some condition

```
Python3>>> [x ** 2 for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]
Python3>>> [(x,y) for x in range(4)
...           for y in range(4)
...           if x % 2 == 0
...           and y % 3 == 0]
[(0, 0), (0, 3), (2, 0), (2, 3)]
Python3>>>
```

- In general

```
[expr for target1 in iterable1 if cond1
     for target2 in iterable2 if cond2 ...
     for targetN in iterableN if condN ]
```

- Instead of the list comprehension, `flattenLevels()` could be defined with an accumulating list and nested loops.
- M269 does not mention list comprehensions so you would have to decide whether they are worth mentioning

```
490 def flattenLevelsA(levels : [[T]]) -> [T] :
491     accumList = []
492     for level in levels :
493         for elem in level :
494             accumList = accumList + [elem]
495     return accumList
```

[ToC](#)

3.6.2 Breadth First V02

- We now have a correct program `breadthBT01()` but this does lots of (how many?) traversals of the data — we may want a more efficient version and hence we transform our program
- Version 02 uses a helper function `bfTraverse(vs,ts)` which takes a list of item seen, `vs`, and a list (or queue) of trees to be visited, `ts`
- As we visit a node, we add its subtree to the queue, `ts`

```
501 def breadthBT02(t : Tree) -> [T] :
502     return bfTraverse([], [t])

504 def bfTraverse(vs : [T], ts : [Tree]) -> [T] :
505     if ts == []:
506         return vs
507     elif isEmptyBT(ts[0]):
508         return bfTraverse(vs, ts[1:])
509     else:
510         return (bfTraverse(vs + [getDataBT(ts[0])],
511                             ts[1:] + [getLeftBT(ts[0]), getRightBT(ts[0])]))
```

[ToC](#)

3.6.3 Breadth First V03

- Version 02 removed some of the recursion by enqueueing trees to be visited
- This version has the disadvantage that no output until all the nodes are visited, which could mean a long wait or never if the tree is infinite

- Version 03 enables a lazier approach — Python could use a [Generator expression](#) or augment the code with [Yield expressions](#) (both not used in M269) but other languages, such as [Haskell](#) use lazy evaluation by default

```

513 def breadthBT03(t : Tree) -> [T] :
514     return lbfBT([t])

516 def lbfBT(ts : [Tree]) -> [T] :
517     if ts == []:
518         return []
519     elif isEmptyBT(ts[0]):
520         return lbfBT(ts[1:])
521     else:
522         return ([getDataBT(ts[0])]
523             + lbfBT(ts[1:] + [getLeftBT(ts[0]),getRightBT(ts[0])]))

```

3.6.4 Breadth First V04

- Version 03 has the only recursive call as (almost) the last thing
- So we can implement this with a [while](#) loop

```

527 def breadthBT04(t : Tree) -> [T] :
528     ts = [t] # Trees to visit
529     vs = [] # Values seen
530     while (ts != []):
531         if not (isEmptyBT(ts[0])) :
532             vs = vs + [getDataBT(ts[0])]
533             ts = ts[1:] + [getLeftBT(ts[0]),getRightBT(ts[0])]
534         else :
535             ts = ts[1:]
536     return vs

```

Activity 5 Breadth First Error

- There is an error in the following version — what is the error ?
- Why would the print statements not help ?
- Why don't the Python type annotations help ?

```

543 def breadthBT04A(t : Tree) -> [T] :
544     ts = [t] # Trees to visit
545     vs = [] # Values seen
546     while not (isEmptyBT(ts)) :
547         print('ts_=_',ts)
548         if not (isEmptyBT(ts[0])) :
549             print('len(ts)_=_',len(ts))
550             print('getDataBT(ts[0])_=_',getDataBT(ts[0]))
551             vs = vs + [getDataBT(ts[0])]
552             ts = ts[1:] + [getLeftBT(ts[0]),getRightBT(ts[0])]
553         else :
554             print('ts[0]_is_empty_',len(ts)_=_',len(ts))
555             ts = ts[1:]
556     return vs

```

[Go to Answer](#)

Answer 5 Breadth First Error

- The error is the [while](#) condition at line 546
- `isEmptyBT(ts)` will never return `True` since `ts` is a list of trees
- The article version of these notes contains an output of the print statements and the error report
- The error is reported at line 548 as `IndexError: list index out of range`

- So the print statements do not show the real error
- The Python interpreter does not check the type annotations for correctness, just the syntax
- Remember that Python is a weakly typed language

[Go to Activity](#)

- Here is a sample error listing for `breadthBT04A()`

```

1 Python3>>> breadthBT04(egBSTreeL)
2 ['D', 'B', 'F', 'A', 'C', 'E', 'G']
3 Python3>>> breadthBT04A(egBSTreeL)
4 ts = [NodeBT(dataBT='D', leftBT=NodeBT(dataBT='B', leftBT=NodeBT(dataBT='A', leftBT=EmptyBT(), ✓
      ✓ rightBT=EmptyBT()), rightBT=NodeBT(dataBT='C', leftBT=EmptyBT(), rightBT=EmptyBT()), ✓
      ✓ rightBT=NodeBT(dataBT='F', leftBT=NodeBT(dataBT='E', leftBT=EmptyBT(), rightBT=EmptyBT()), ✓
      ✓ rightBT=NodeBT(dataBT='G', leftBT=EmptyBT(), rightBT=EmptyBT())))]
5 len(ts) = 1
6 getDataBT(ts[0]) = D
7 ts = [NodeBT(dataBT='B', leftBT=NodeBT(dataBT='A', leftBT=EmptyBT(), rightBT=EmptyBT()), ✓
      ✓ rightBT=NodeBT(dataBT='C', leftBT=EmptyBT(), rightBT=EmptyBT()), NodeBT(dataBT='F', ✓
      ✓ leftBT=NodeBT(dataBT='E', leftBT=EmptyBT(), rightBT=EmptyBT()), rightBT=NodeBT(dataBT='G', ✓
      ✓ leftBT=EmptyBT(), rightBT=EmptyBT()))]
8 len(ts) = 2
9 getDataBT(ts[0]) = B
10 ts = [NodeBT(dataBT='F', leftBT=NodeBT(dataBT='E', leftBT=EmptyBT(), rightBT=EmptyBT()), ✓
      ✓ rightBT=NodeBT(dataBT='G', leftBT=EmptyBT(), rightBT=EmptyBT()), NodeBT(dataBT='A', ✓
      ✓ leftBT=EmptyBT(), rightBT=EmptyBT()), NodeBT(dataBT='C', leftBT=EmptyBT(), rightBT=EmptyBT())]
11 len(ts) = 3
12 getDataBT(ts[0]) = F
13 ts = [NodeBT(dataBT='A', leftBT=EmptyBT(), rightBT=EmptyBT()), NodeBT(dataBT='C', leftBT=EmptyBT(), ✓
      ✓ rightBT=EmptyBT()), NodeBT(dataBT='E', leftBT=EmptyBT(), rightBT=EmptyBT()), ✓
      ✓ NodeBT(dataBT='G', leftBT=EmptyBT(), rightBT=EmptyBT())]
14 len(ts) = 4
15 getDataBT(ts[0]) = A
16 ts = [NodeBT(dataBT='C', leftBT=EmptyBT(), rightBT=EmptyBT()), NodeBT(dataBT='E', leftBT=EmptyBT(), ✓
      ✓ rightBT=EmptyBT()), NodeBT(dataBT='G', leftBT=EmptyBT(), rightBT=EmptyBT()), EmptyBT(), ✓
      ✓ EmptyBT()]
17 len(ts) = 5
18 getDataBT(ts[0]) = C
19 ts = [NodeBT(dataBT='E', leftBT=EmptyBT(), rightBT=EmptyBT()), NodeBT(dataBT='G', leftBT=EmptyBT(), ✓
      ✓ rightBT=EmptyBT()), EmptyBT(), EmptyBT(), EmptyBT(), EmptyBT()]
20 len(ts) = 6
21 getDataBT(ts[0]) = E
22 ts = [NodeBT(dataBT='G', leftBT=EmptyBT(), rightBT=EmptyBT()), EmptyBT(), EmptyBT(), EmptyBT(), ✓
      ✓ EmptyBT(), EmptyBT(), EmptyBT()]
23 len(ts) = 7
24 getDataBT(ts[0]) = G
25 ts = [EmptyBT(), EmptyBT(), EmptyBT(), EmptyBT(), EmptyBT(), EmptyBT(), EmptyBT(), EmptyBT()]
26 ts[0] is empty len(ts) = 8
27 ts = [EmptyBT(), EmptyBT(), EmptyBT(), EmptyBT(), EmptyBT(), EmptyBT(), EmptyBT(), EmptyBT()]
28 ts[0] is empty len(ts) = 7
29 ts = [EmptyBT(), EmptyBT(), EmptyBT(), EmptyBT(), EmptyBT(), EmptyBT()]
30 ts[0] is empty len(ts) = 6
31 ts = [EmptyBT(), EmptyBT(), EmptyBT(), EmptyBT(), EmptyBT()]
32 ts[0] is empty len(ts) = 5
33 ts = [EmptyBT(), EmptyBT(), EmptyBT(), EmptyBT()]
34 ts[0] is empty len(ts) = 4
35 ts = [EmptyBT(), EmptyBT(), EmptyBT()]
36 ts[0] is empty len(ts) = 3
37 ts = [EmptyBT(), EmptyBT()]
38 ts[0] is empty len(ts) = 2
39 ts = [EmptyBT()]
40 ts[0] is empty len(ts) = 1
41 ts = []
42 Traceback (most recent call last):
43   File "<stdin>", line 1, in <module>
44   File ✓
      ✓ "/Users/molyneux/MyData/Documents/OU/Courses/Computing/M269/M269TutorialResources/M269TutorialResources2022/
      ✓ line 548, in breadthBT04A
45     if not (isEmptyBT(ts[0])) :
46 IndexError: list index out of range
47 Python3>>>

```

3.6.5 Breadth First V05

- There is one disadvantage of version 01
- The program traverses the entire left subtree before traversing the right subtree
- Bad news for large trees and very bad for infinite trees
- This version produces the traversal level by level

```

563 def labelsAtDepth(d : int, t : Tree) -> [T] :
564     if isEmptyBT(t) :
565         return []
566     else :
567         x = getDataBT(t)
568         left = getLeftBT(t)
569         right = getRightBT(t)
570         if d == 0 :
571             return [x]
572         else :
573             return (labelsAtDepth((d-1),left) + labelsAtDepth((d-1),right))

```

- Breadth first traversal with `labelsAtDepth`
- Version based on Sannella et al. (2022, page 261) *Introduction to Computation: Haskell, Logic and Automata*

```

577 def bftTraverseByLevels(t : Tree) -> [T] :
578     return bftByL(0,t)

580 def bftByL(d : int, t : Tree) -> [T] :
581     xs = labelsAtDepth(d,t)
582     if xs == [] :
583         return []
584     else :
585         return (xs + bftByL((d+1),t))

```

[ToC](#)

4 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.

4.1 Iterative InOrder Traversal

- Here is the original recursive version (from line 311 on page 23)

```

311 def inOrderBT(t) :
312     if isEmptyBT(t) :
313         return []
314     else :
315         return (inOrderBT(getLeftBT(t)) + [getDataBT(t)]
316                 + inOrderBT(getRightBT(t)))

```

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

```

335 def inOrderBT0(t) :
336     result = []
337     if not isEmptyBT(t) :
338         result = result + (inOrderBT0(getLeftBT(t)))
339         result.append(getDataBT(t))
340         result = result + (inOrderBT0(getRightBT(t)))
341     return result

```

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

```

343 def inOrderIterBT1(t) :
344     result = []
345     while not isEmptyBT(t) :
346         result = result + (inOrderIterBT1(getLeftBT(t)))
347         result.append(getDataBT(t))
348         t = getRightBT(t)
349     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.

```

351 def inOrderIterBT2(t) :
352     result = []
353     stack = []
354     while (not (stack == []) or not isEmptyBT(t)) :
355         if not isEmptyBT(t) :
356             stack.append(t)
357             t = getLeftBT(t)
358         else:
359             t = stack.pop()
360             result.append(getDataBT(t))
361             t = getRightBT(t)
362     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`

[ToC](#)

4.2 Iterative PreOrder Traversal

- Here is the original recursive version (from line 318 on page 23)

```

318 def preOrderBT(t) :
319     if isEmptyBT(t) :
320         return []
321     else :
322         return ([getDataBT(t)] + preOrderBT(getLeftBT(t))
323               + preOrderBT(getRightBT(t)))

```

- Start with recursive version with accumulating result.

```

364 def preOrderBT0(t) :
365     result = []
366     if not isEmptyBT(t) :
367         result.append(getDataBT(t))
368         result = result + (preOrderBT0(getLeftBT(t)))
369         result = result + (preOrderBT0(getRightBT(t)))
370     return result

```

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

```

372 def preOrderIterBT1(t) :
373     result = []
374     while not isEmptyBT(t) :
375         result.append(getDataBT(t))
376         result = result + (preOrderIterBT1(getLeftBT(t)))
377         t = getRightBT(t)
378     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

```

380 def preOrderIterBT2(t) :
381     result = []
382     stack = []
383     while (not (stack == [])) or not isEmptyBT(t) :
384         if not isEmptyBT(t) :
385             result.append(getDataBT(t))
386             stack.append(t)
387             t = getLeftBT(t)
388         else :
389             t = stack.pop()
390             t = getRightBT(t)
391     return result

```

[ToC](#)

4.3 Iterative PostOrder Traversal

- Here is the original recursive version (from line 325 on page 23)

```

325 def postOrderBT(t):
326     if isEmptyBT(t):
327         return []
328     else:
329         return (postOrderBT(getLeftBT(t))
330               + postOrderBT(getRightBT(t)) + [getDataBT(t)])

```

- Start with recursive version with accumulating result.

```

393 def postOrderBT0(t) :
394     result = []
395     if not isEmptyBT(t) :
396         result = result + (postOrderIterBT1(getLeftBT(t)))
397         result = result + (postOrderIterBT1(getRightBT(t)))
398         result.append(getDataBT(t))
399     return result

```

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

```

401 def postOrderIterBT1(t) :
402     result = []
403     if isEmptyBT(t) :
404         return result
405     stack = []
406     while not (stack == []) or (not isEmptyBT(t)) :
407         while not isEmptyBT(t) :
408             if not isEmptyBT(getRightBT(t)) :
409                 stack.append(getRightBT(t))
410                 stack.append(t)
411                 t = getLeftBT(t)
412             t = stack.pop()
413             if ((not isEmptyBT(getRightBT(t)))
414                 and (stack != [] and stack[-1] is getRightBT(t))) :
415                 tr = stack.pop()
416                 stack.append(t)
417                 t = getRightBT(t)
418             else :
419                 result.append(getDataBT(t))
420                 t = mkEmptyBT() # To avoid infinite loop - it is t.rightBT
421     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.

[ToC](#)

Commentary 3

3 Binary Search Trees

- Binary trees with the *binary search tree* property
- Inserting a node
- Other BST operations
- Deletion — investigating choices

[ToC](#)

5 Binary Search Trees

5.1 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:

`insertBST()`, `inBST()`, `isBSTree()`, `insertListBST()`, `buildBST()`, `deleteBST()`

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 6 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 6 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 $L_0 = 0$, $L_1 = 1$, $L_2 = 2$, $L_3 = 4, \dots$ and in general $L_h = 2^{h-1}$, $h \geq 1$
- Now $N_h = L_1 + L_2 + \dots + L_h = 2^0 + 2^1 + \dots + 2^{h-1}$
- $2N_h = 2^1 + 2^2 + \dots + 2^{h-1} + 2^h$
- 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](#)

[ToC](#)

5.2 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.

Binary Search Tree — Inserting a Node

```
564 def insertBST(x,t):
565     if isEmptyBT(t):
566         return mkNodeBT(x,mkEmptyBT(),mkEmptyBT())
567     else:
568         y = getDataBT(t)
569         if x < y:
570             return mkNodeBT(y,insertBST(x,getLeftBT(t)),getRightBT(t))
571         elif x > y:
572             return mkNodeBT(y,getLeftBT(t),insertBST(x,getRightBT(t)))
573         else:
574             return t
```

Activity 7 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](#)

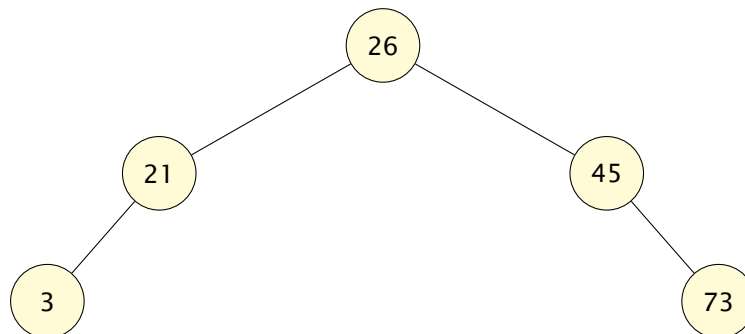
insBSTreeA

insBSTreeA

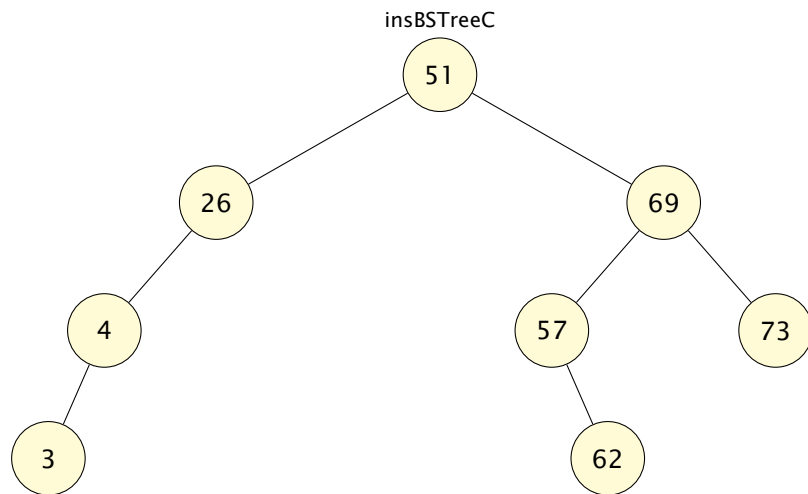


insBSTreeB

insBSTreeB

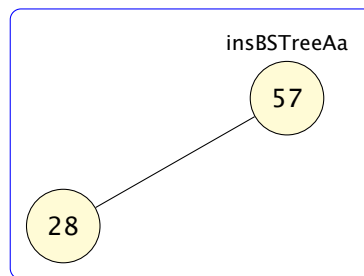


insBSTreeC

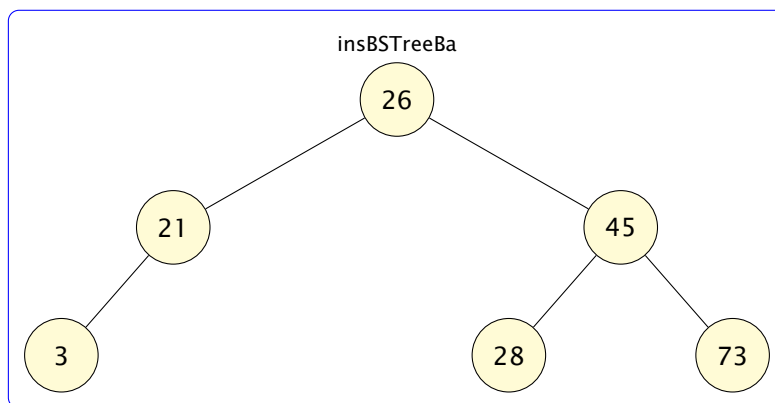
[Go to Answer](#)

Answer 7 Inserting an Item

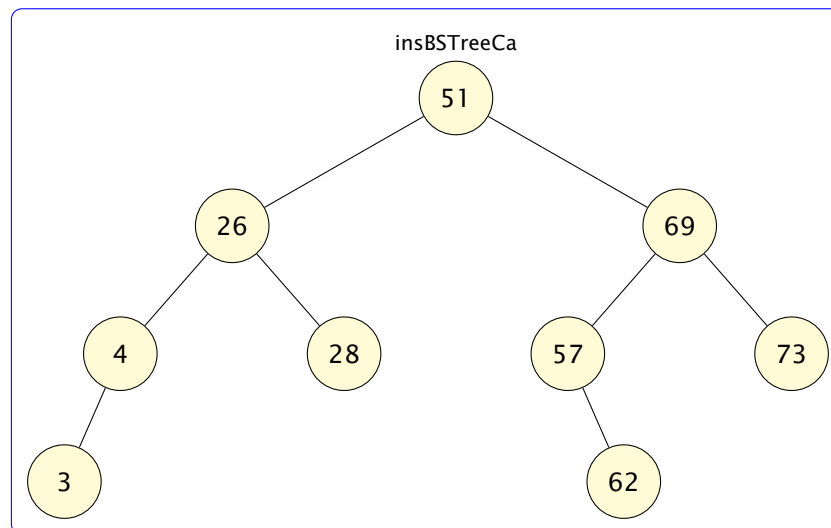
insBSTreeAa



insBSTreeBa



insBSTreeCa



[Go to Activity](#)

5.3 BST Operations

Activity 8 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 8 Membership

```

576 def inBST(k,t):
577     if isEmptyBT():
578         return False
579     else:
580         p = getDataBT(t)
581         if k < p:
582             return inBST(k, getLeftBT(t))
583         elif k > p:
584             return inBST(k, getRightBT(t))
585         else:
586             return True
  
```

[Go to Activity](#)

5.3.1 Testing a BST

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

```

588 def isBSTree(t):
589     return orderedList(inOrderBT(t))

591 def orderedList(xs):
592     return (len(xs) <= 1
593             or (xs[0] < xs[1] and orderedList(xs[1:]))
  
```

5.3.2 List to BST

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:

```

595 def insertListBST(xs,t) :
596     if xs == [] :
597         return t
598     else :
599         return insertListBST (xs[1:], insertBST(xs[0],t))

```

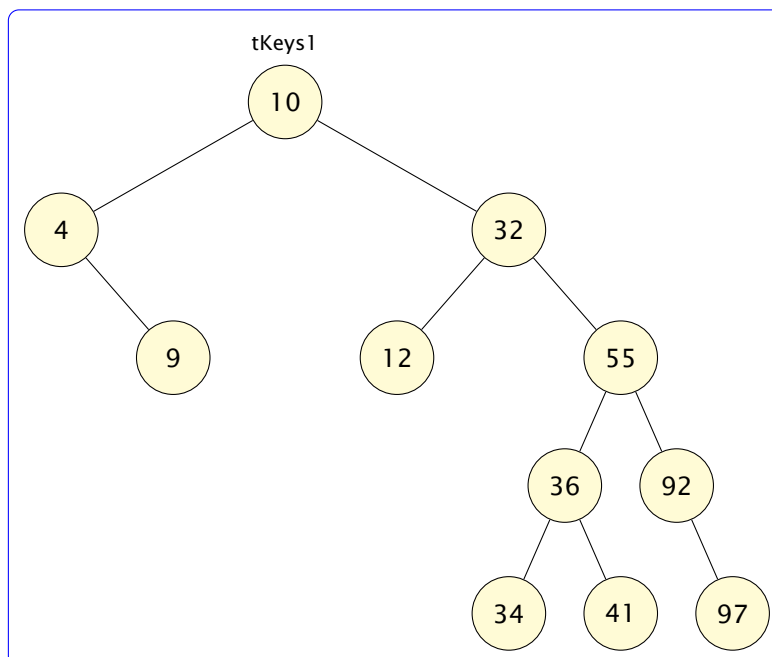
- However, see what happens when we insert various lists — how *compact* is the resulting tree ?

Activity 9 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 9 Insert List `tKeys1 = insertListBST(keys1, mkEmptyBT())`



```

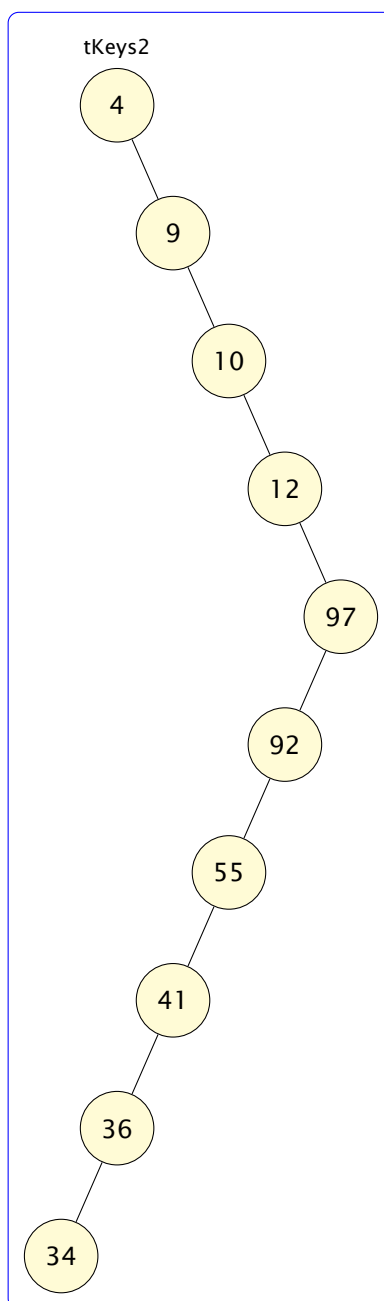
tKeys1 = mkNodeBT(10,
    mkNodeBT(4,
        mkEmptyBT(),
        mkNodeBT(9, mkEmptyBT(), mkEmptyBT()))),
    mkNodeBT(32,
        mkNodeBT(12, mkEmptyBT(), mkEmptyBT()),
        mkNodeBT(55,
            mkNodeBT(36,
                mkNodeBT(34, mkEmptyBT(), mkEmptyBT()),
                mkNodeBT(41, mkEmptyBT(), mkEmptyBT()))),
            mkNodeBT(92,
                mkEmptyBT(),
                mkNodeBT(97, mkEmptyBT(), mkEmptyBT())))))

```

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

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

Answer 9 Insert List `tKeys2 = insertListBST(keys2, mkEmptyBT())`

[illegible]

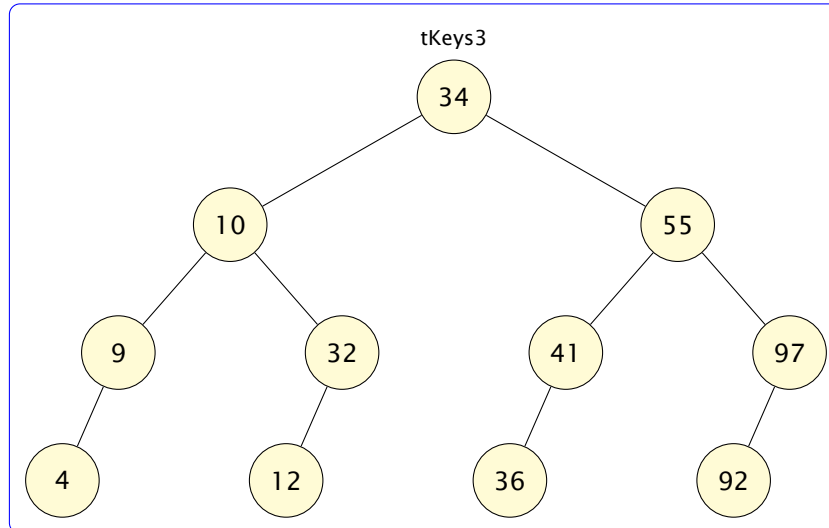
```

        mkEmptyBT(),
        mkEmptyBT(),
        mkEmptyBT()))))

tKeys2Test = (tKeys2
  == insertListBST(keys2, mkEmptyBT()))

```

Answer 9 Insert List `tKeys3 = insertListBST(keys3, mkEmptyBT())`



```

tKeys3 = mkNodeBT(34,
  mkNodeBT(10,
    mkNodeBT(9,
      mkNodeBT(4, mkEmptyBT(), mkEmptyBT()),
      mkEmptyBT()),
    mkNodeBT(32,
      mkNodeBT(12, mkEmptyBT(), mkEmptyBT()),
      mkEmptyBT()),
    mkNodeBT(55,
      mkNodeBT(41,
        mkNodeBT(36, mkEmptyBT(), mkEmptyBT()),
        mkEmptyBT()),
      mkNodeBT(97,
        mkNodeBT(92, mkEmptyBT(), mkEmptyBT()),
        mkEmptyBT()))))

tKeys3Test = (tKeys3
  == insertListBST(keys3, mkEmptyBT()))

```

- 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 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 $O(\log n)$
- The proof of this requires knowledge of statistics outside the remit of this course — if interested, a proof is in [Cormen et al. \(2009, page 300\) Theorem 12.4](#) and [Cormen et al. \(2022, page 328\) Problem 12-3](#)

[Go to Activity](#)

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

```

604 def buildBST(xs) :
605     return bBST(mkEmptyBT(), sorted(xs))

608 def bBST(t,xs) :
609     if xs == [] :
610         return t
611     else :
612         half = len(xs) // 2
613         x = xs[half]
614         frontxs = xs[:half]
615         backxs = xs[half+1:]
616         if isEmptyBT(t) :
617             return (mkNodeBT(x,
618                             bBST(mkEmptyBT(), frontxs),
619                             bBST(mkEmptyBT(), backxs)))
620         else :
621             errMsg = ("bBST:_Trying_to_insert" + str(xs)
622                     + "_into_nonempty_tree" + str(t))
623             raise RuntimeError(errMsg)

```

```

Python3>>> xs = [1, 9, 2, 8, 3, 7, 4, 6, 5]
Python3>>> buildBST(xs)
NodeBT(dataBT=5,
       leftBT=NodeBT(dataBT=3,
                     leftBT=NodeBT(dataBT=2,
                                   leftBT=NodeBT(dataBT=1,
                                                  leftBT=EmptyBT(),
                                                  rightBT=EmptyBT()),
                                   rightBT=EmptyBT()),
                     rightBT=NodeBT(dataBT=4,
                                   leftBT=EmptyBT(),
                                   rightBT=EmptyBT())),
       rightBT=NodeBT(dataBT=8,
                     leftBT=NodeBT(dataBT=7,
                                   leftBT=NodeBT(dataBT=6,
                                                  leftBT=EmptyBT(),
                                                  rightBT=EmptyBT()),
                                   rightBT=EmptyBT()),
                     rightBT=NodeBT(dataBT=9,
                                   leftBT=EmptyBT(),
                                   rightBT=EmptyBT()))))
Python3>>>

```

- Note that when the Python interpreter prints a namedtuple it includes field names and values and has no line breaks

[ToC](#)

5.4 BST 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 `mkEmptyBT()`
- 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:

Deleting an Item from a BST — Python

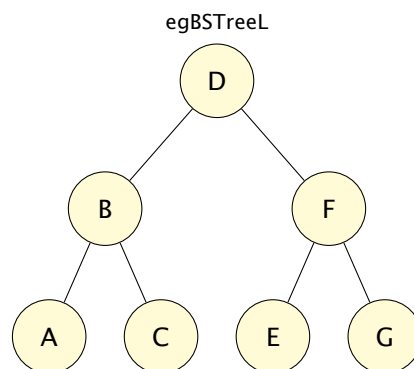
```

627 def deleteBST(x, t):
628     if isEmptyBT(t):
629         return mkEmptyBT()
630     else:
631         y = getDataBT(t)
632         leftT = getLeftBT(t)
633         rightT = getRightBT(t)
634         if x < y:
635             return mkNodeBT(y, (deleteBST(x, leftT)), rightT)
636         elif x > y:
637             return mkNodeBT(y, leftT, (deleteBST(x, rightT)))
638         else:
639             return 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`

```

122 egBSTreeL = mkNodeBT('D',
123                     mkNodeBT('B',
124                             mkNodeBT('A', mkEmptyBT(), mkEmptyBT()),
125                             mkNodeBT('C', mkEmptyBT(), mkEmptyBT()))
126                     ),
127                     mkNodeBT('F',
128                             mkNodeBT('E', mkEmptyBT(), mkEmptyBT()),
129                             mkNodeBT('G', mkEmptyBT(), mkEmptyBT()))
130                     )

```

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.

```

643 def joinBST1(leftT, rightT):
644     if isEmptyBT(leftT):
645         return rightT
646     else:
647         return (insertListBST(inOrderBT(leftT), rightT))

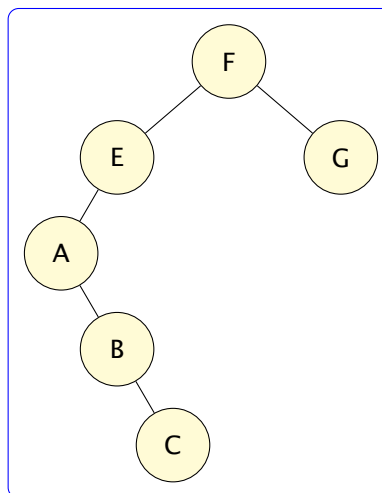
```

Activity 10 joinBST Version 1

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

[Go to Answer](#)

Answer 10 joinBST Version 1



```

651 delBSTreeJoin1 = joinBST1(egBSTreeLL, egBSTreeLR)

653 delBSTreeJoin1ans = \
654     NodeBT(dataBT='F',
655         leftBT=NodeBT(dataBT='E',
656             leftBT=NodeBT(dataBT='A',
657                 leftBT=EmptyBT(),
658                 rightBT=NodeBT(dataBT='B',
659                     leftBT=EmptyBT(),
660                     rightBT=NodeBT(dataBT='C',
661                         leftBT=EmptyBT(),
662                         rightBT=EmptyBT()))),
663             rightBT=EmptyBT()),
664         rightBT=NodeBT(dataBT='G',
665             leftBT=EmptyBT(),
666             rightBT=EmptyBT()))

668 delBSTreeJoin1test = delBSTreeJoin1 == delBSTreeJoin1ans

```

[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.

```

672 def joinBST2(leftT, rightT):
673     if isEmptyBT(rightT):
674         return leftT
675     else:
676         return (mkNodeBT(getDataBT(rightT),
677                          joinBST2(leftT, getLeftBT(rightT)),
678                          getRightBT(rightT)))

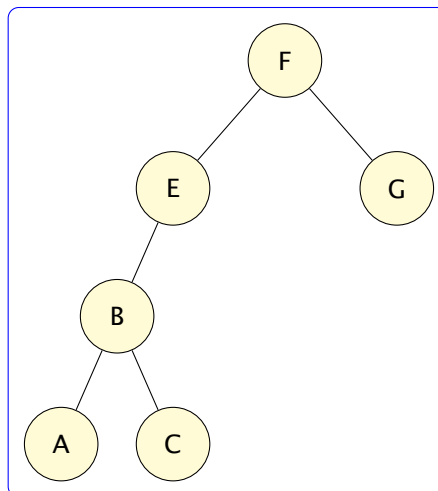
```

Activity 11 joinBST Version 2

- Draw the diagram of the tree resulting from deleting D with [joinBST2](#)
- Can you see how we can improve on [joinBST2](#) ?

[Go to Answer](#)

Answer 11 joinBST Version 2



```

681 delBSTreeJoin2 = joinBST2(egBSTreeLL, egBSTreeLR)
682
683 delBSTreeJoin2ans = NodeBT('F',
684                             NodeBT('E',
685                                     NodeBT('B',
686                                             NodeBT('A', EmptyBT(), EmptyBT()),
687                                             NodeBT('C', EmptyBT(), EmptyBT()),
688                                             EmptyBT()),
689                                     NodeBT('G', EmptyBT(), EmptyBT()))
690
691 delBSTreeJoin2test = (delBSTreeJoin2 == delBSTreeJoin2ans)

```

- To see how this works we shall do a step by step evaluation
- Follow the function code for [joinBST2](#) from line 672 on page 46
- Note from the definitions of [delBSTreeJoin1test](#) (from line 668 on page 45) and [delBSTreeJoin2test](#) (from line 691 on page 46) we can use the field names or leave them out
- **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 673 tests if the second argument to `joinBST2` is an empty tree
- Since it is not empty, `joinBST2` evaluates to the value at line 676
- Hence we have

```
mkNodeBT('F',
  joinBST2(leftT, getLeftBT(rightT)),
  getRightBT(rightT))
```

- **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 673 tests if the first argument to `joinBST2` is an empty tree
- Since it is not empty, `joinBST2` evaluates to the value at line 676
- Hence we have

```
mkNodeBT('F',
  mkNodeBT('E',
    joinBST2(leftT, getLeftBT(getLeftBT(rightT))),
    getRightBT(getLeftBT(rightT))),
  getRightBT(rightT))
```

- **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

```
mkNodeBT('F',
  mkNodeBT('E',
    leftT,
    mkEmptyBT()),
  getRightBT(rightT))
```

- 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.

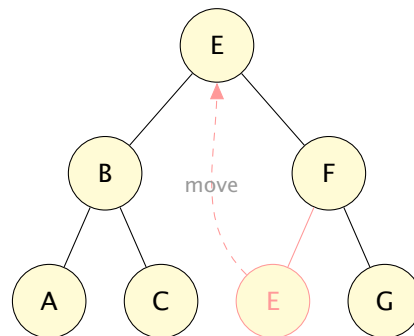
[Go to Activity](#)

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 code for the above diagram:

```

696 def joinBST(leftT, rightT):
697     if isEmptyBT(rightT):
698         return leftT
699     else:
700         (y,t) = splitBST(rightT)
701         return mkNodeBT(y, leftT, t)

```

- `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.

```

706 def splitBST1(t):
707     if isEmptyBT(t):
708         raise RuntimeError("splitBST1_applied_to_EmptyBT()")
709     elif isEmptyBT(getLeftBT(t)):
710         return (getDataBT(t), getRightBT(t))
711     else:
712         y = minItemBST(getLeftBT(t))
713         return (y, mkNodeBT(getDataBT(t),
714                             deleteBST(y, getLeftBT(t)),
715                             getRightBT(t)))
716
717 def minItemBST(t):
718     if isEmptyBT(t):
719         raise RuntimeError("minItemBST_applied_to_EmptyBT()")
720     elif isEmptyBT(getLeftBT(t)):
721         return (getDataBT(t))
722     else:
723         return (minItemBST(getLeftBT(t)))

```

`splitBST` Final Version

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

```

727 def splitBST(t):
728     if isEmptyBT(t):
729         raise RuntimeError("splitBST_applied_to_EmptyBT()")
730     else:

```

```

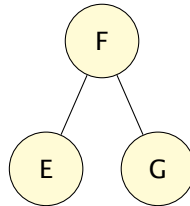
731 x = getDataBT(t)
732 t1 = getLeftBT(t)
733 t2 = getRightBT(t)
734 if isEmptyBT(t1):
735     return (x,t2)
736 else:
737     (y,t3) = splitBST(t1)
738     return (y, mkNodeBT(x, t3, t2))

```

Activity 12 Split Trace

- Trace an evaluation of `splitBST(egBSTreeLR)`

`splitBST(egBSTreeLR)`



```

160 egBSTreeLR = mkNodeBT('F',
161                       mkNodeBT('E',mkEmptyBT(),mkEmptyBT()),
162                       mkNodeBT('G',mkEmptyBT(),mkEmptyBT()))
163

```

- `egBSTreeLR` is defined at line 160 on page 49, `splitBST` is defined at line 727 on page 48

[Go to Answer](#)

Answer 12 Split Trace

- Evaluation of `splitBST(egBSTreeLR)`

Step 1

`getLeftBT(egBSTreeLR)` is not empty so the `else` clause at line 736 is executed

Step 2

A recursive call is made to `splitBST` with argument `getLeftBT(egBSTreeLR)`

`getLeftBT(getLeftBT(egBSTreeLR))` is empty so the `if` at line 734 returns `('E',getRightBT(getLeftBT(egBSTreeLR)))` which is `('E',EmptyBT())`

Answer 12 Split Trace

Step 3

The calling function then returns

`('E', makeBT('F', EmptyBT(), getRightBT(egBSTreeLR)))`

```

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

```

[Go to Activity](#)

Activity 13 Join Trace

- Trace an evaluation of

```
joinBST(egBSTreeLL, egBSTreeLR)
```

- `egBSTreeLL` is defined at line 146 on page 22, `joinBST` is defined at line 696 on page 48

[Go to Answer](#)

Answer 13 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 699 — this invokes `splitBST(egBSTreeLR)`

Step 2

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

```
('E', makeBT('F', EmptyBT(), getRightBT(egBSTreeLR)))
```

Answer 13 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 14 Delete Trace

- Trace an evaluation of

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

- `egBSTreeL` is defined at line 122 on page 21, `deleteBST` is defined at line 627 on page 44

[Go to Answer](#)

Answer 14 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 630 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 638 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.

[ToC](#)

Commentary 4

4 Height Balanced Trees

- Binary search trees with the *height balanced* property
- Also called AVL trees
- Inserting a node
- AVL transformations
- Local changes preserve global AVL property
- Deletion
- AVL trees application: representing sets (advanced topic)
- Note: Haskell uses the same ideas but with *size* balanced trees
- Python uses something like dictionaries to implement sets using hashtables

[ToC](#)

6 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.

6.1 AVL Trees and Functions

AVL Tree Data Type

- As with the Binary Search Tree, we shall use a union of named tuples to represent the data type for an AVL Tree.
- 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

```

743 # AVL Tree Data Type
744
745 # from collections import namedtuple
746
747 EmptyABT = namedtuple('EmptyABT', [])
748
749 NodeABT = (namedtuple('NodeABT',
750     ['heightABT', 'dataABT', 'leftABT', 'rightABT']))
751
752 # Tree type --- Augmented Binary Tree
753
754 # from typing import TypeVar, Union, NewType
755
756 # T = TypeVar('T')
757 ABTree = NewType('ABTree', Union[EmptyABT, NodeABT]).

```

AVL Tree Operations

```

759 # AVL Tree Operations
760
761 def mkEmptyABT() -> ABTree :
762     return EmptyABT()
763
764 def mkNodeABT(x: T, t1: ABTree, t2: ABTree) -> ABTree :
765     h = 1 + max(getHeightABT(t1), getHeightABT(t2))
766     return NodeABT(h, x, t1, t2)
767
768 def isEmptyABT(t: ABTree) -> bool :
769     return t == EmptyABT()
770
771 def getHeightABT(t: ABTree) -> int :
772     if isEmptyABT(t) :
773         return 0
774     else:
775         return t.heightABT
776
777 def getDataABT(t: ABTree) -> T :
778     if isEmptyABT(t) :
779         raise RuntimeError("getDataABT_applied_to_EmptyABT()")
780     else:
781         return t.dataABT
782
783 def getLeftABT(t: ABTree) -> ABTree :
784     if isEmptyABT(t) :
785         raise RuntimeError("getLeftABT_applied_to_EmptyABT()")
786     else:
787         return t.leftABT
788
789 def getRightABT(t: ABTree) -> ABTree :
790     if isEmptyABT(t) :
791         raise RuntimeError("getRightABT_applied_to_EmptyABT()")
792     else:

```

```
793 return t.rightABT
```

AVL Trees Property Functions

```
797 def isBSABTree(t):
798     return orderedList(inOrderABT(t))

800 def inOrderABT(t):
801     if isEmptyABT(t):
802         return []
803     else:
804         return (inOrderABT(getLeftABT(t)) + [getDataABT(t)]
805               + inOrderABT(getRightABT(t)))
```

```
832 def convertBTtoABT(t):
833     if isEmptyBT(t):
834         return mkEmptyABT()
835     else:
836         leftABT = convertBTtoABT(getLeftBT(t))
837         rightABT = convertBTtoABT(getRightBT(t))
838         return mkNodeABT(getDataBT(t), leftABT, rightABT)
```

```
809 def balFactorABT(t):
810     if isEmptyABT(t):
811         return 0
812     else:
813         return (getHeightABT(getLeftABT(t))
814               - getHeightABT(getRightABT(t)))
```

```
818 def hasAVLpropABT(t):
819     if isEmptyABT(t):
820         return True
821     else:
822         return (abs(balFactorABT(t)) <= 1
823               and hasAVLpropABT(getLeftABT(t))
824               and hasAVLpropABT(getRightABT(t)))
```

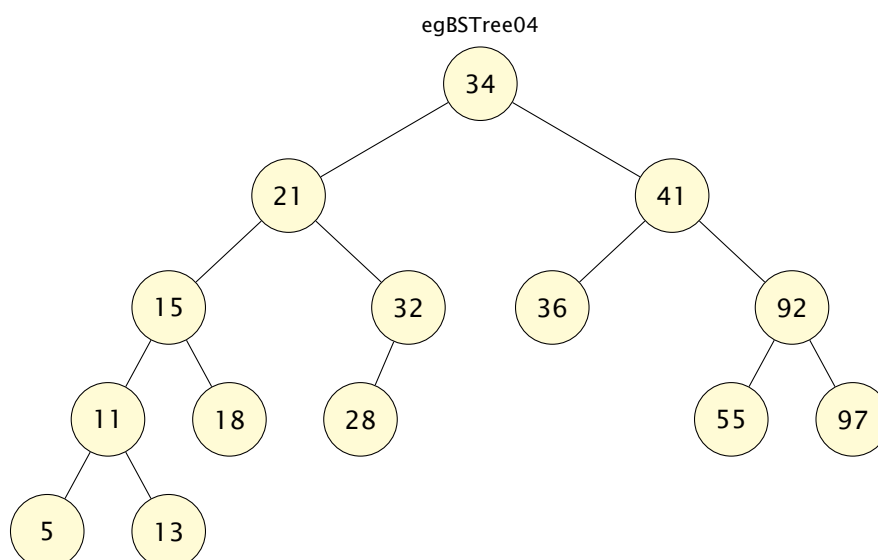
```
827 def isAVLABTree(t):
828     return (isBSABTree(t) and hasAVLpropABT(t))
```

- Some texts 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.

6.2 Example AVL Trees

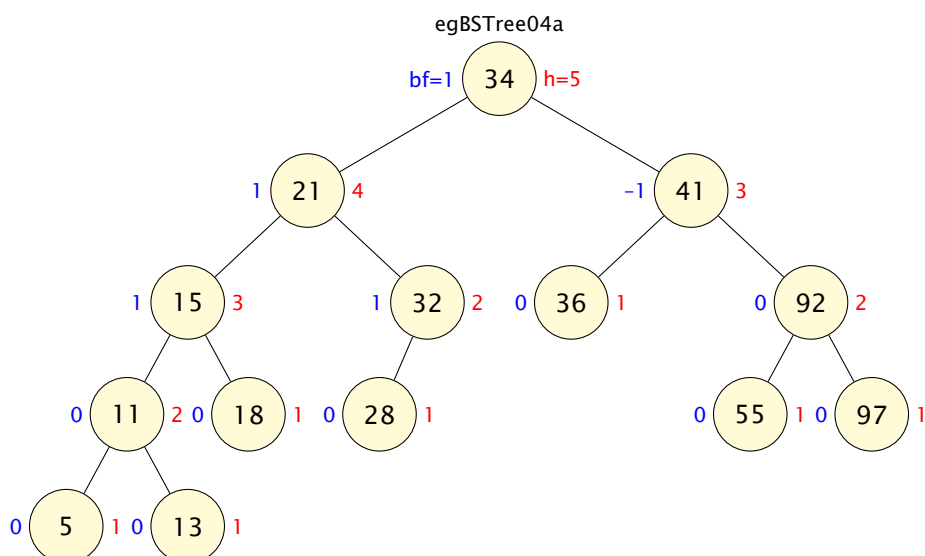
Activity 15 Height and Balance Factor

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



[Go to Answer](#)

Answer 15 Height and Balance Factor



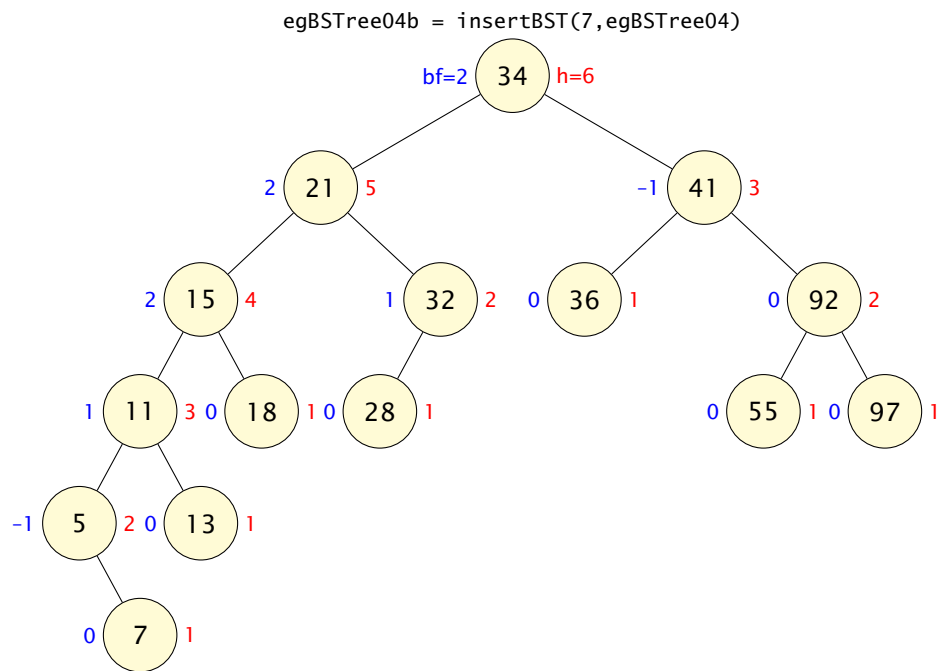
[Go to Activity](#)

Activity 16 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 16 Add Item LL



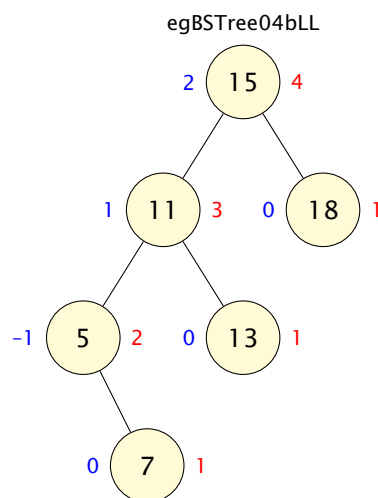
- Lowest node which is out of balance is node with key 15

[Go to Activity](#)

6.3 AVL Tree Transformations

- Since the subtree of `egBSTree04b = insertBST(7, egBSTree04)` at node with key 15 is the part of the tree out of balance we shall focus on that

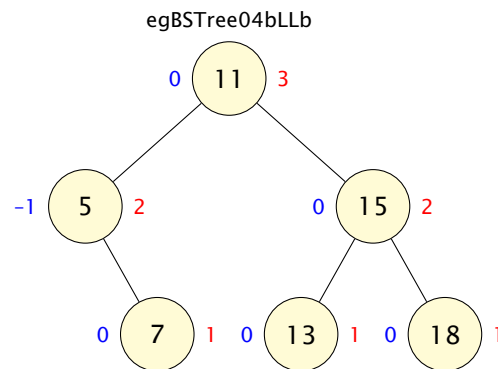
```
NodeABT(4, 15,
  NodeABT(3, 11,
    NodeABT(2, 5,
      EmptyABT(),
      NodeABT(1, 7, EmptyABT(), EmptyABT()),
      NodeABT(1, 13, EmptyABT(), EmptyABT()),
      NodeABT(1, 18, EmptyABT(), EmptyABT())
```



- 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.

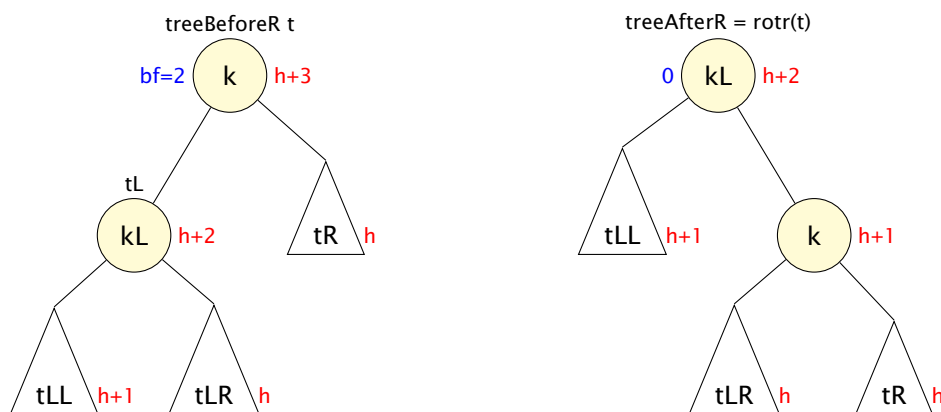
```
NodeABT(3, 11,
NodeABT(2, 5,
    EmptyABT(),
    NodeABT(1, 7, EmptyABT(), EmptyABT()))),
NodeABT(2, 15,
    NodeABT(1, 13, EmptyABT(), EmptyABT()),
    NodeABT(1, 18, EmptyABT(), EmptyABT()))
```



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

```
843 def rotr(t):
844     k = getDataABT(t)
845     kL = getDataABT(getLeftABT(t))
846     leftLeftT = getLeftABT(getLeftABT(t))
847     leftRightT = getRightABT(getLeftABT(t))
848     rightT = getRightABT(t)
849     return (mkNodeABT(kL,
850                     leftLeftT,
851                     mkNodeABT(k, leftRightT, rightT)))
```

Case LL Heavy — Right Rotation

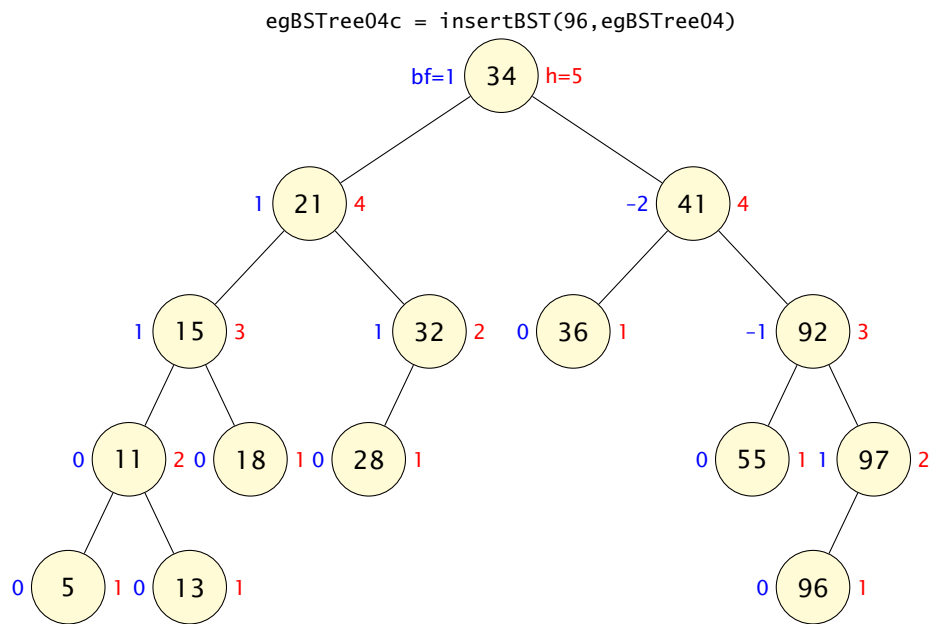


Activity 17 Add Item RR

- Consider [egBSTree04](#) again (defined in Activity 15 on page 53) — now add node with key 96 and recalculate the heights and balance factors

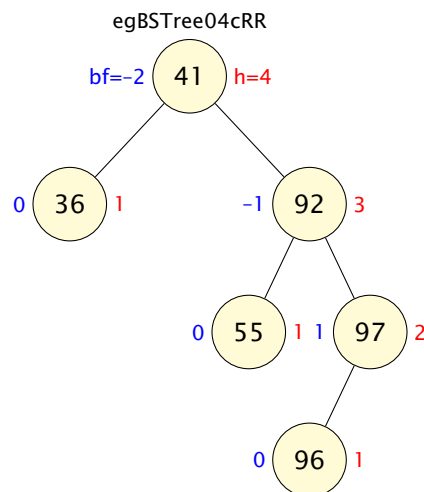
[Go to Answer](#)

Answer 17 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 18 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](#)

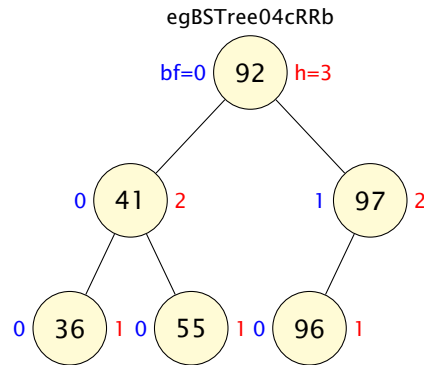
Answer 18 Rebalance RR

- 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.

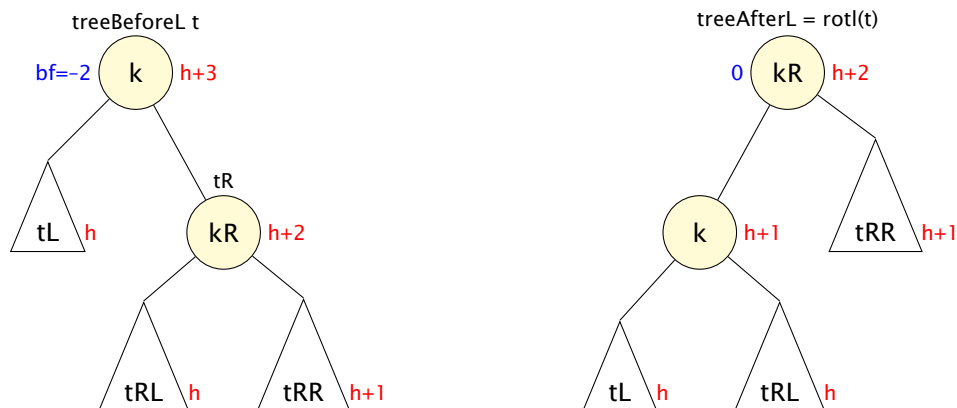
```

853 def rotl(t):
854     k = getDataABT(t)
855     kR = getDataABT(getRightABT(t))
856     rightLeftT = getLeftABT(getRightABT(t))
857     rightRightT = getRightABT(getRightABT(t))
858     leftT = t.leftABT
859     return (mkNodeABT(kR,
860                     mkNodeABT(k, leftT, rightLeftT),
861                     rightRightT))

```

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

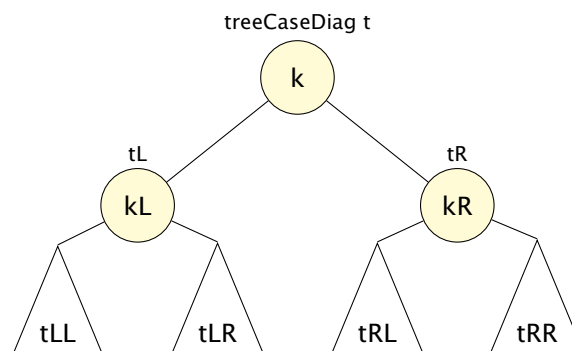
Case RR Heavy — Left Rotation



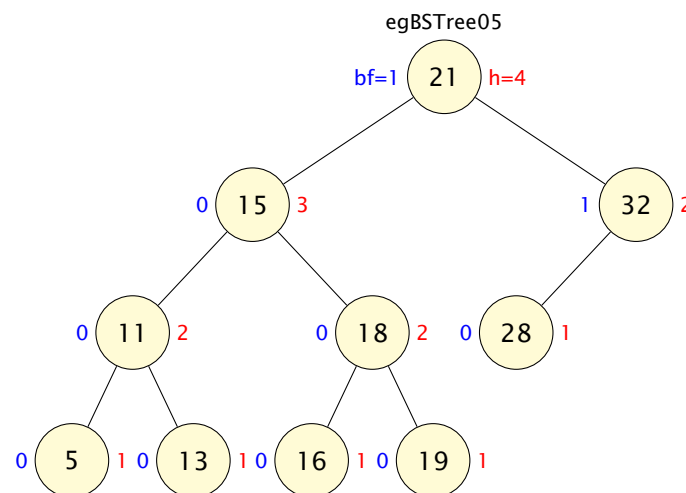
AVL Tree Transformations — Insight

- The functions for insertion and deletion of an item in an AVL tree will be the same as a Binary Search tree except

- When we construct a new tree we must maintain the AVL property via a function `makeAVLTree` (line 865 on page 65) not just `makeABTree` (line 764 on page 52). (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`

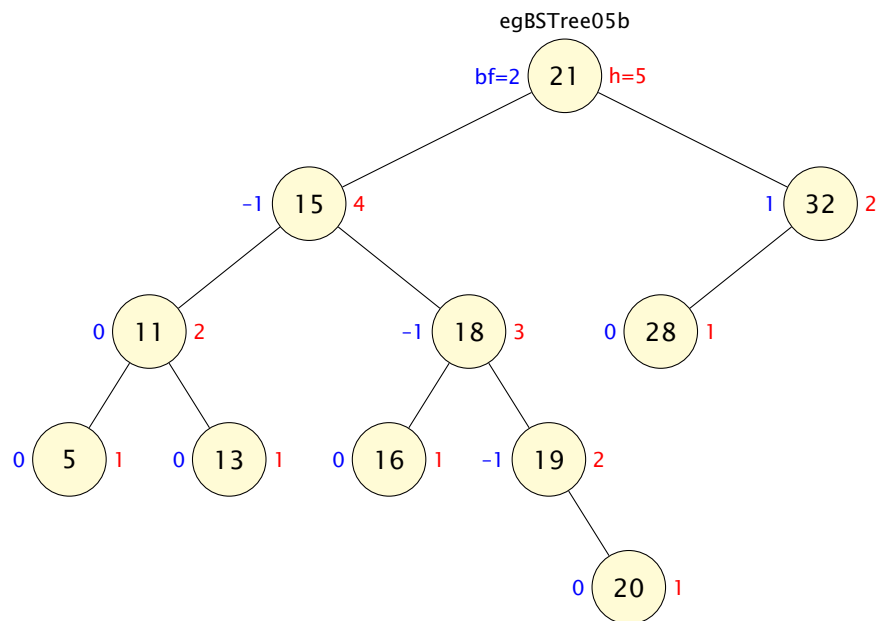


Activity 19 egBSTree05 Add Item LR 1

- 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 19 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](#)

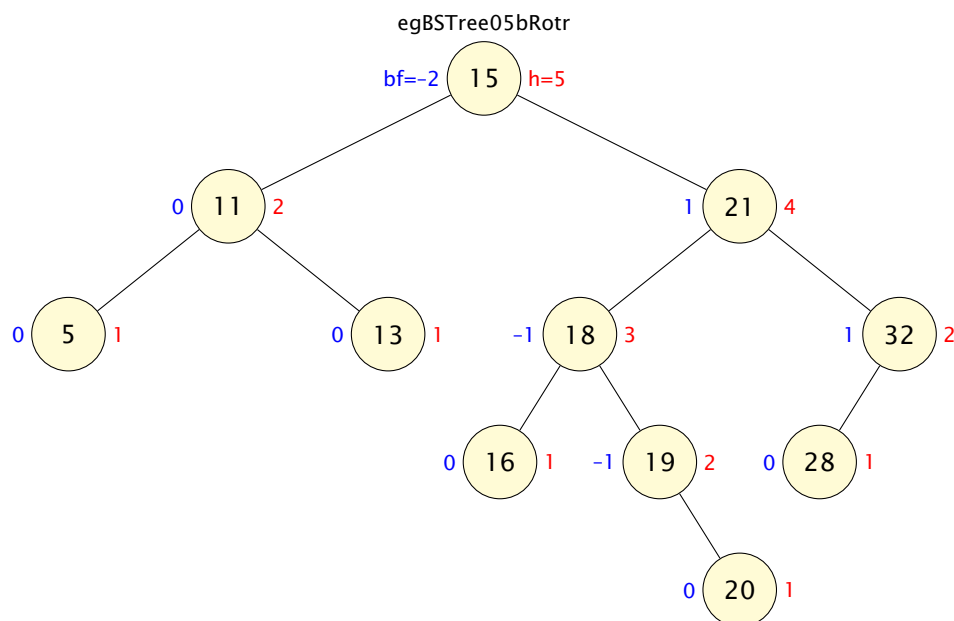
Activity 20 Add Item LR 2

- Given the resulting tree from Self-assessment activity 19, does a right rotation around the lowest node which is out of balance bring it back to balance ?

[Go to Answer](#)

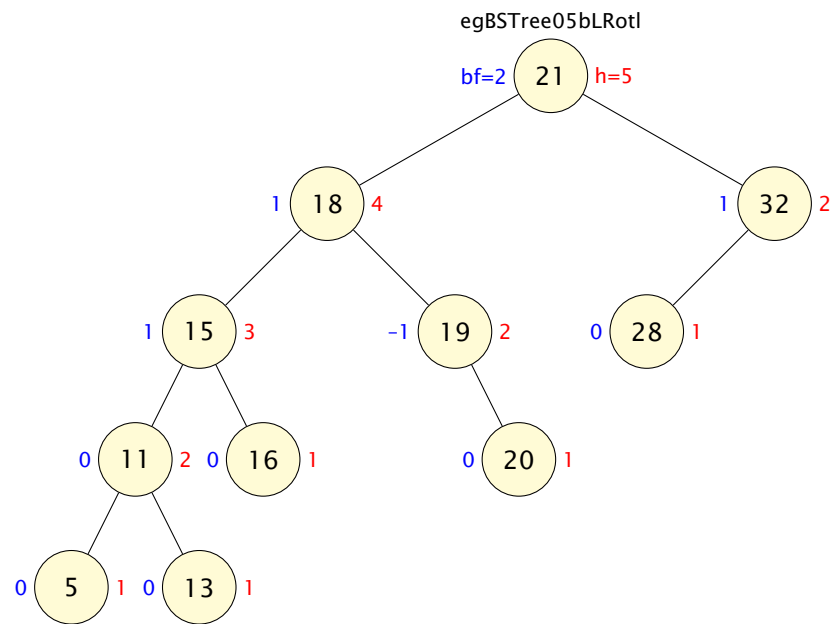
Answer 20 Add Item LR 2

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

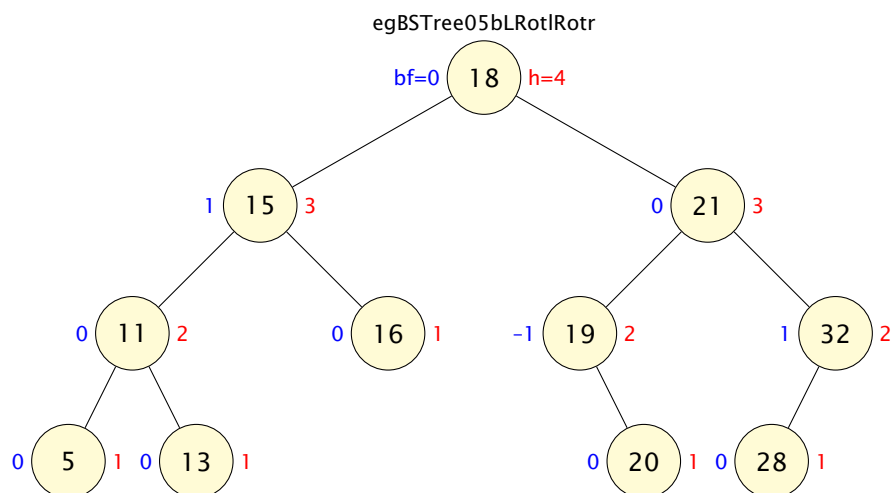


[Go to Activity](#)

- 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.

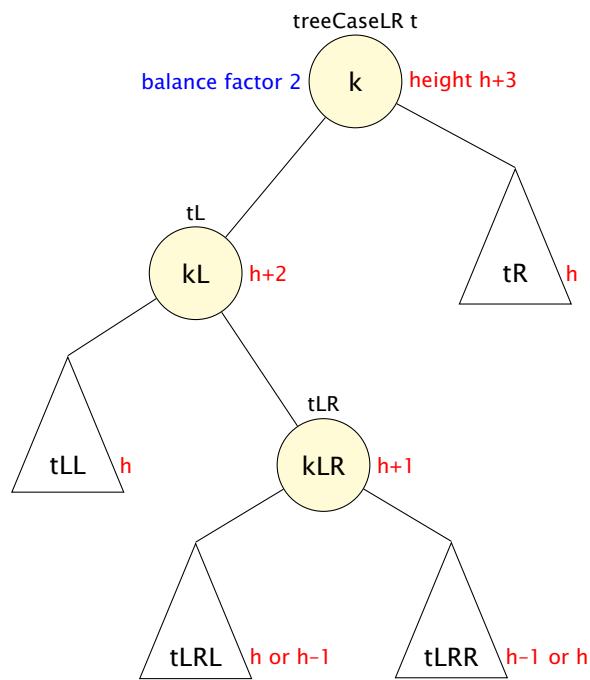


- Notice that this has converted a tree which was LR heavy to one where it is LL heavy — so we can now use a right rotation on the tree rooted at 21 to get the following:

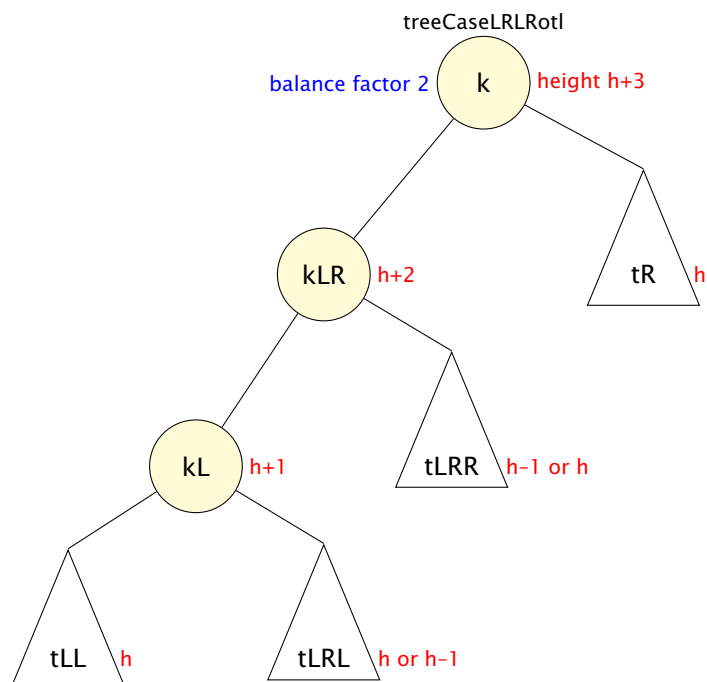


- We now have a balanced tree — but were we just lucky or have we found a general rule? Here are diagrams of the double rotation to show it works in general:

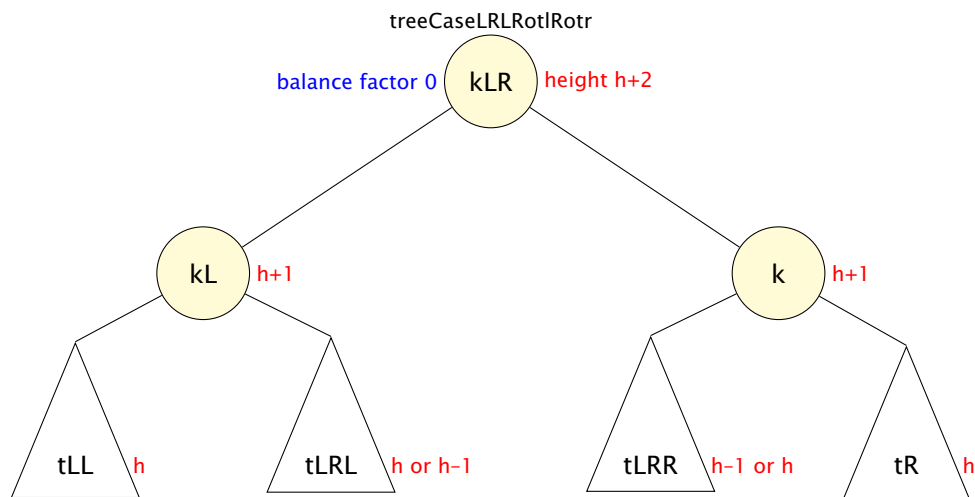
Case LR subtree heavy



Case LR — Step (A) Rotate Left about **kL**



Case LR — Step (B) Rotate Right about **k**

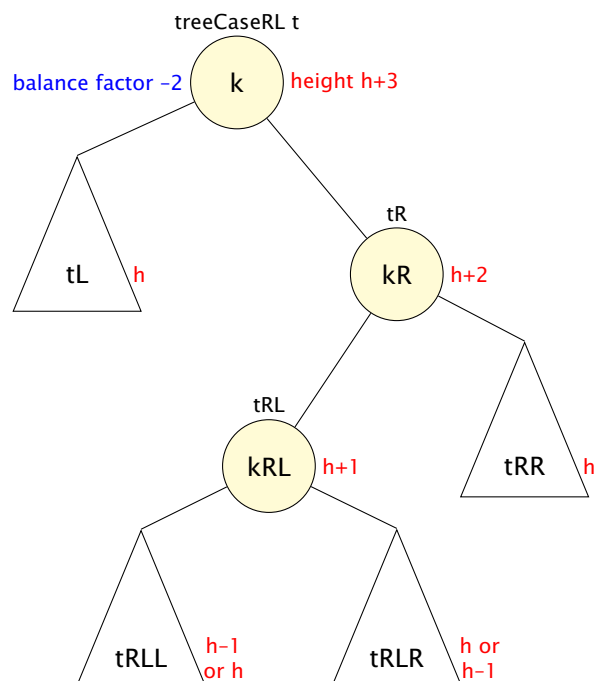


Activity 21 Case RL Heavy

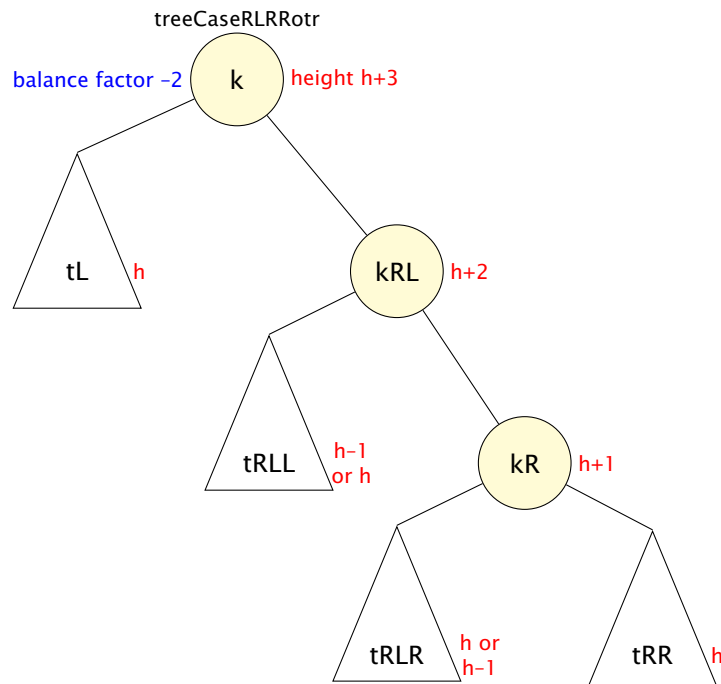
- Draw the equivalent diagram for the final case where subtree **tRL** is heavy
- Note that this must be the mirror image of the **tLR** case

[Go to Answer](#)

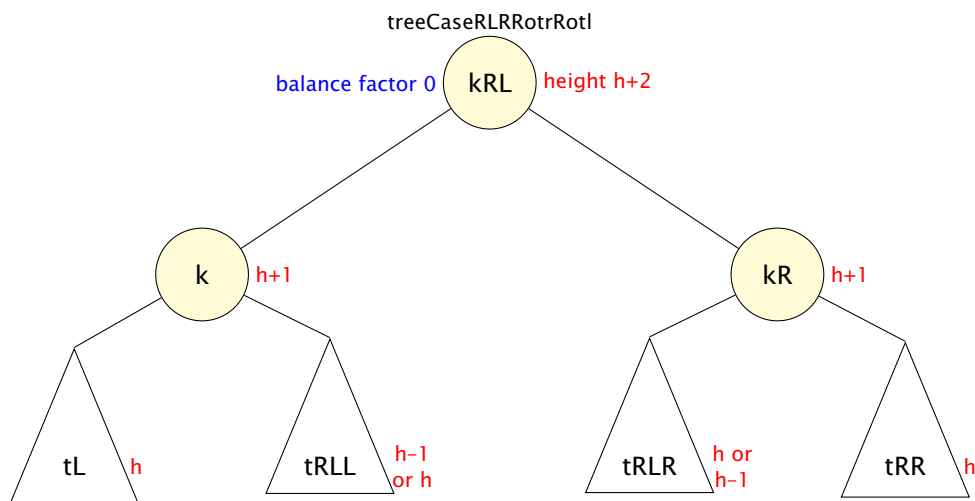
Answer 21 Case RL Heavy



Answer 21 Case RL Heavy — Step (A) Rotate Right about **kR**



Answer 21 Case RL Heavy — Step (B) Rotate Left about k



[Go to Activity](#)

6.4 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

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

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

Case 2 LR Heavy

```
(getHeightABT(leftT) - getHeightABT(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

```
(getHeightABT(leftT) - getHeightABT(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

```
(getHeightABT(leftT) - getHeightABT(rightT) = -2
and balFactorABT(rightT) <= 0)
```

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

Case 5 Otherwise

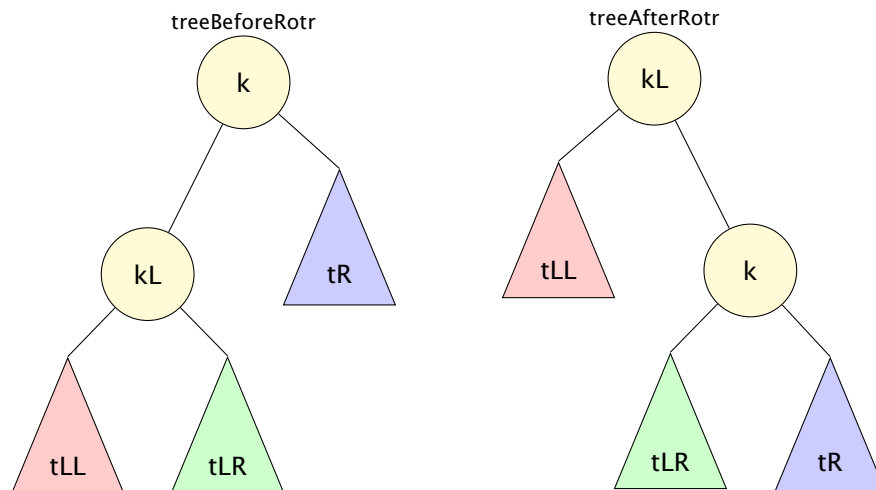
- Just use `makeABTree(x, leftT, rightT)`
- No transformations required

makeAVLTree Function — Python

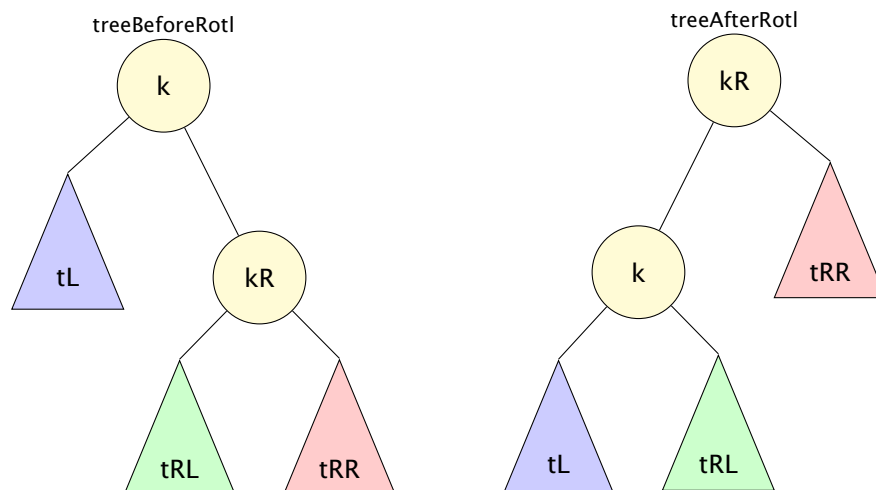
```
865 def makeAVLTree(x, leftT, rightT):
866     hL = getHeightABT(leftT)
867     hR = getHeightABT(rightT)
868     if (hR + 1 < hL) and (balFactorABT(leftT) >= 0):
869         return rotr(mkNodeABT(x, leftT, rightT))
870     elif (hR + 1 < hL):
871         return rotr(mkNodeABT(x, (rotl(leftT)), rightT))
872     elif (hL + 1 < hR) and (balFactorABT(rightT) > 0):
873         return rotl(mkNodeABT(x, leftT, rotr(rightT)))
874     elif (hL + 1 < hR):
875         return rotl(mkNodeABT(x, leftT, rightT))
876     else:
877         return mkNodeABT(x, leftT, 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



6.4.1 Comparison with Storing Balance Factors

- Some texts implement 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 66)
- 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 56,

```

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 58,

```

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

```

[ToC](#)

6.5 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.

```

881 def insertAVLT(x,t):
882     if isEmptyABT(t):
883         return mkNodeABT(x, mkEmptyABT(), mkEmptyABT())
884     else:
885         y = getDataABT(t)
886         leftT = getLeftABT(t)
887         rightT = getRightABT(t)
888         if x < y:
889             return makeAVLTree(y, insertAVLT(x, leftT), rightT)
890         elif x > y:
891             return makeAVLTree(y, leftT, insertAVLT(x, rightT))
892         else:
893             return t

```

```

895 def insertListAVLT(xs,t):
896     if xs == []:
897         return t
898     else:
899         return insertListAVLT(xs[1:], (insertAVLT(xs[0],t)))

```

```

901 def deleteAVLT(x,t):
902     if isEmptyABT(t):
903         return mkEmptyABT()
904     else:
905         y = getDataABT(t)
906         leftT = getLeftABT(t)
907         rightT = getRightABT(t)
908         if x < y:

```

```

909     return makeAVLTree(y, deleteAVLT(x, leftT), rightT)
910 elif x > y:
911     return makeAVLTree(y, leftT, deleteAVLT(x, rightT))
912 else:
913     return joinAVLT(leftT, rightT)

```

```

917 def joinAVLT(leftT, rightT):
918     if isEmptyABT(rightT):
919         return leftT
920     else:
921         (y,t) = splitAVLT(rightT)
922         return makeAVLTree(y, leftT, t)

```

```

926 def splitAVLT(t):
927     if isEmptyABT(t):
928         raise RuntimeError("splitAVLT_applied_to_EmptyABT()")
929     else:
930         x = getDataABT(t)
931         t1 = getLeftABT(t)
932         t2 = getRightABT(t)
933         if isEmptyABT(t1):
934             return (x,t2)
935         else:
936             (y,t3) = splitAVLT(t1)
937             return (y, makeAVLTree(x, t3, t2))

```

Activity 22 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 881, page 68 (Python), the `deleteAVLT` function is defined at line 901, page 68 (Python),
- `insertListAVLT` is defined at line 895, page 68 (Python),

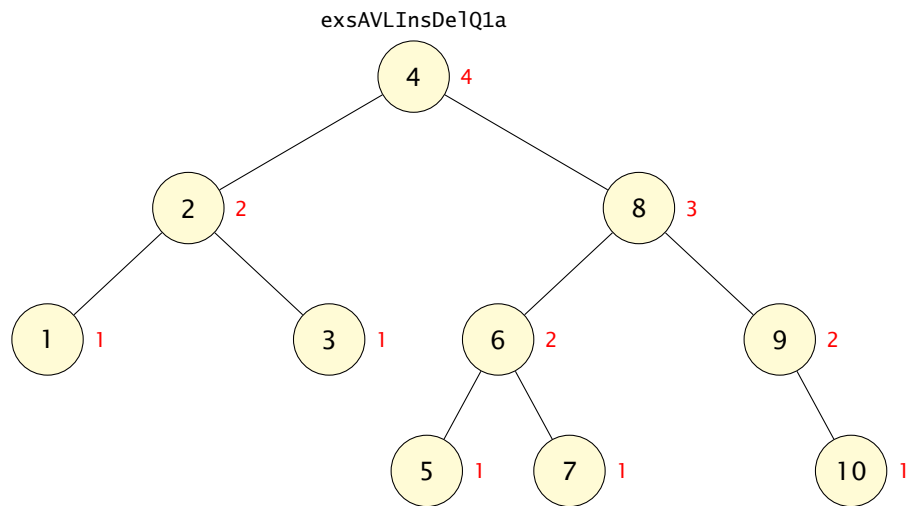
[Go to Answer](#)

Answer 22 Insert Lists and Delete Items

```

listQ1a = [1,2,3,4,5,6,7,8,9,10]
exsAVLInsDe1Q1a = insertListAVLT(listQ1a, EmptyABT())

```



Answer 22 Insert Lists and Delete Items

- Here is the Python representation of the resulting AVL tree

```

exsAVLInsDe1Q1aAns \
= (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 22 Insert Lists and Delete Items

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

```

exsAVLInsDe1Q1a01 \
= insertListAVLT(listQ1a[:1], EmptyABT())

```

exsAVLInsDe1Q1a01



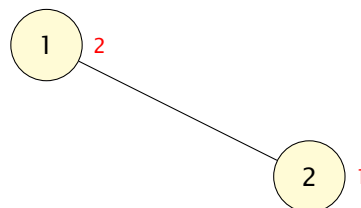
Answer 22 Insert Lists and Delete Items

```

exsAVLInsDe1Q1a02 \
= insertListAVLT(listQ1a[:2], EmptyABT())

```

exsAVLInsDe1Q1a02



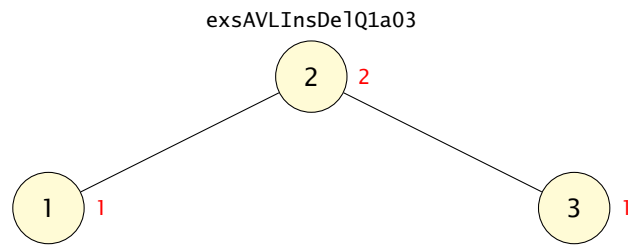
Answer 22 Insert Lists and Delete Items

```

exsAVLInsDe1Q1a03 \
= insertListAVLT(listQ1a[:3], EmptyABT())

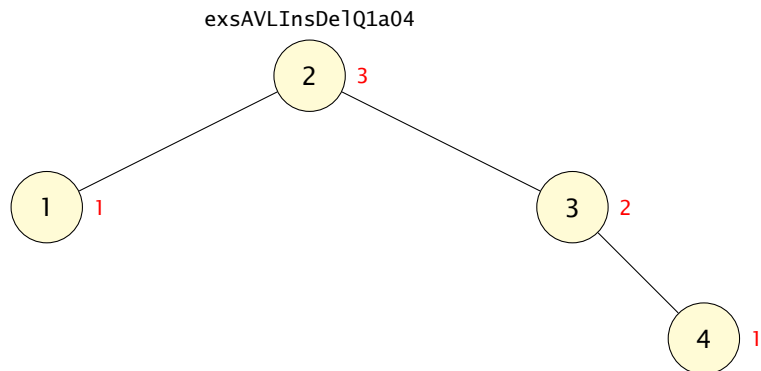
```

Left rotation about 1



Answer 22 Insert Lists and Delete Items

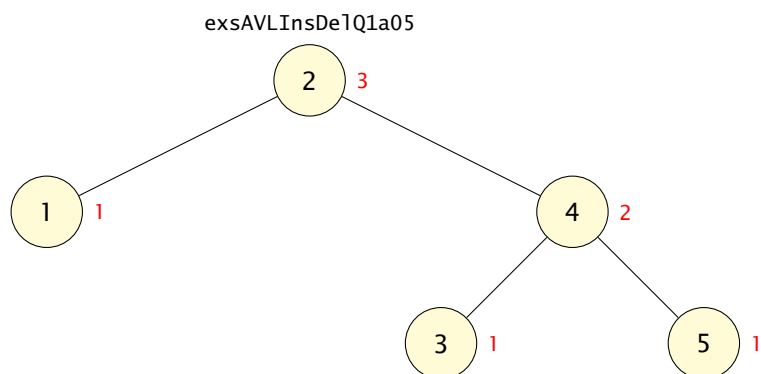
```
exsAVLInsDe1Q1a04 \
= insertListAVLT(listQ1a[:4], EmptyABT())
```



Answer 22 Insert Lists and Delete Items

```
exsAVLInsDe1Q1a05 \
= insertListAVLT(listQ1a[:5], EmptyABT())
```

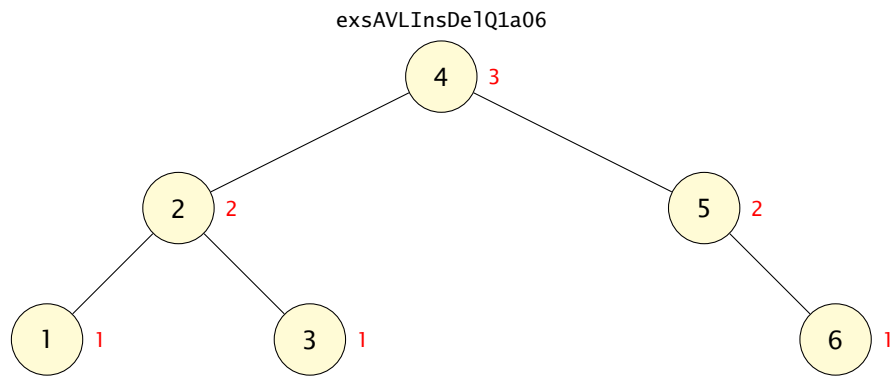
Left rotation about 3



Answer 22 Insert Lists and Delete Items

```
exsAVLInsDe1Q1a06 \
= insertListAVLT(listQ1a[:6], EmptyABT())
```

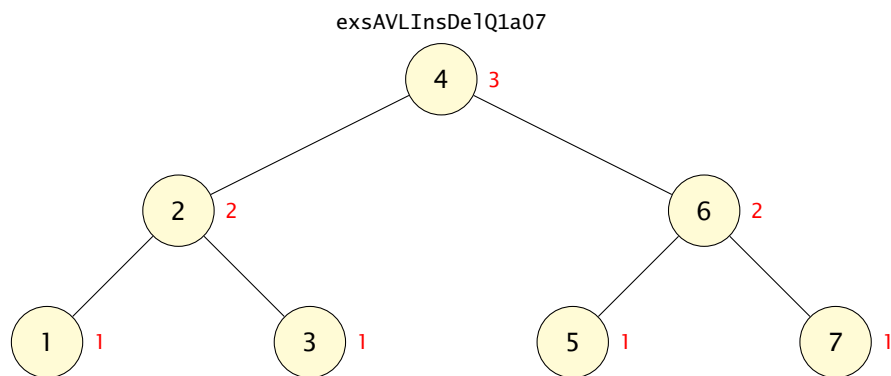
Left rotation about 2



Answer 22 Insert Lists and Delete Items

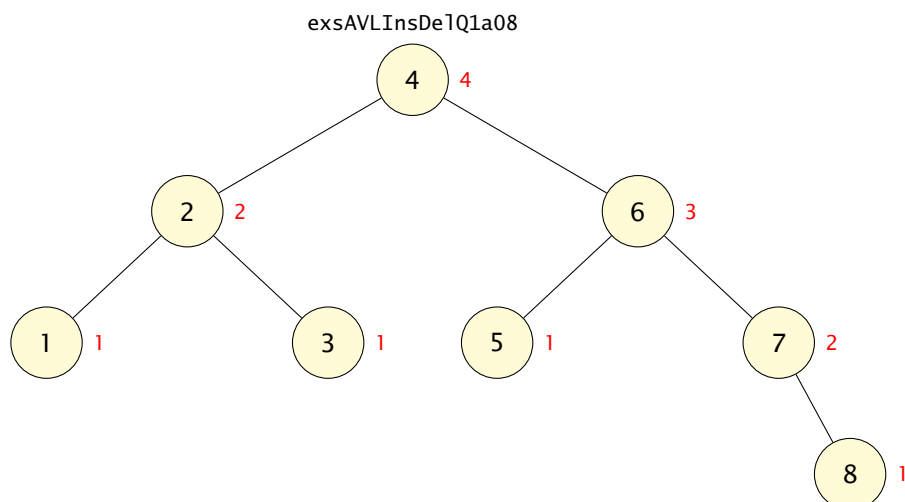
```
exsAVLInsDe1Q1a07 \
= insertListAVLT(listQ1a[:7], EmptyABT())
```

Left rotation about 5



Answer 22 Insert Lists and Delete Items

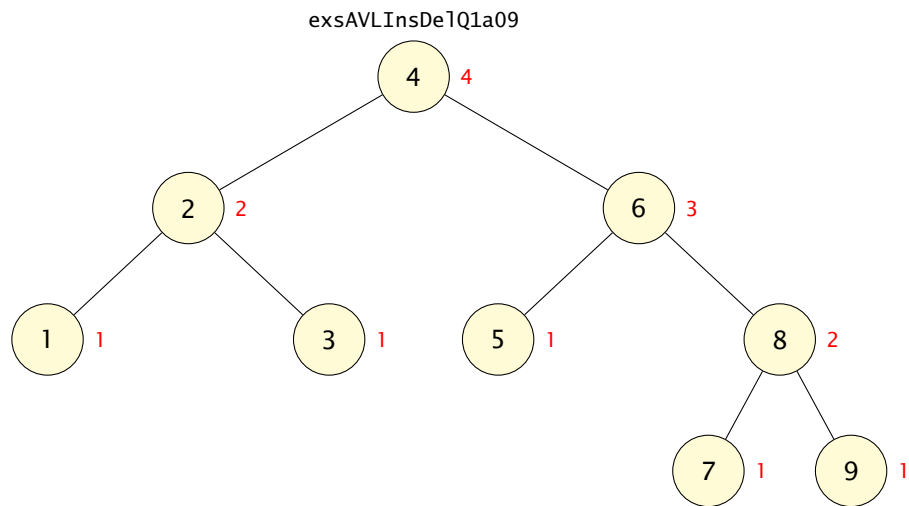
```
exsAVLInsDe1Q1a08 \
= insertListAVLT(listQ1a[:8], EmptyABT())
```



Answer 22 Insert Lists and Delete Items

```
exsAVLInsDe1Q1a09 \
= insertListAVLT(listQ1a[:9], EmptyABT())
```

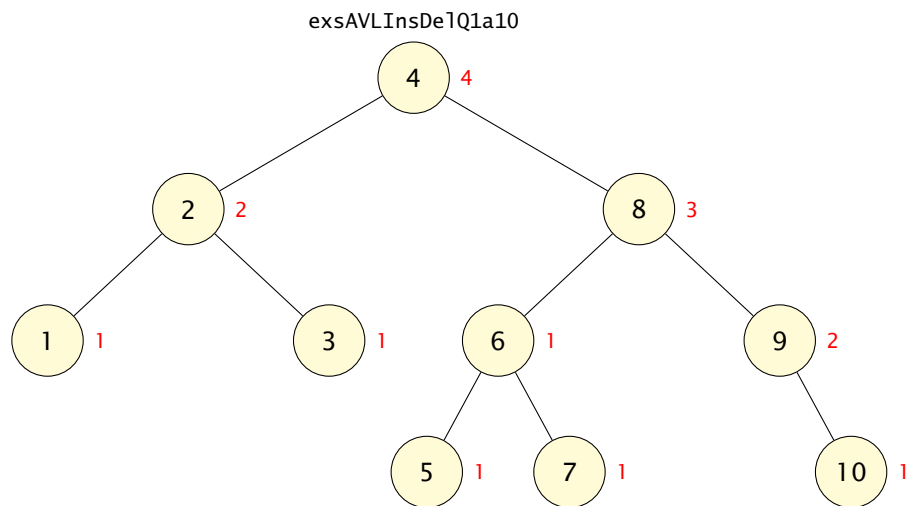
Left rotation about 7



Answer 22 Insert Lists and Delete Items

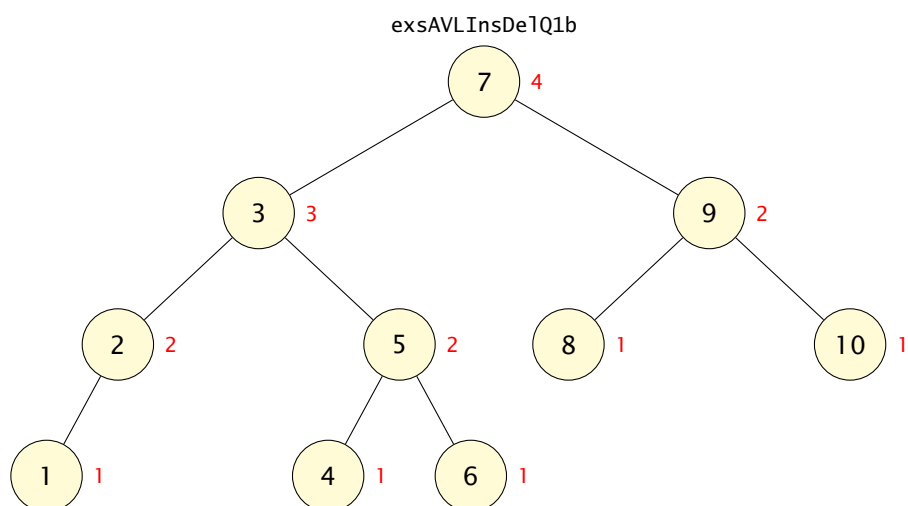
```
exsAVLInsDe1Q1a10 \
= insertListAVLT(listQ1a[:10], EmptyABT())
```

Left rotation around 6



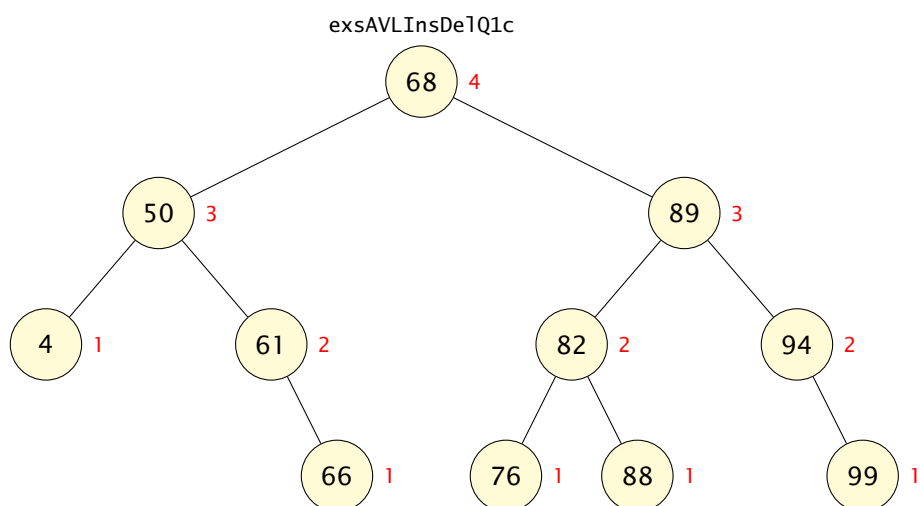
Answer 22 Insert Lists and Delete Items

```
listQ1b = [10,9,8,7,6,5,4,3,2,1]
exsAVLInsDe1Q1b = insertListAVLT(listQ1b, EmptyABT())
```



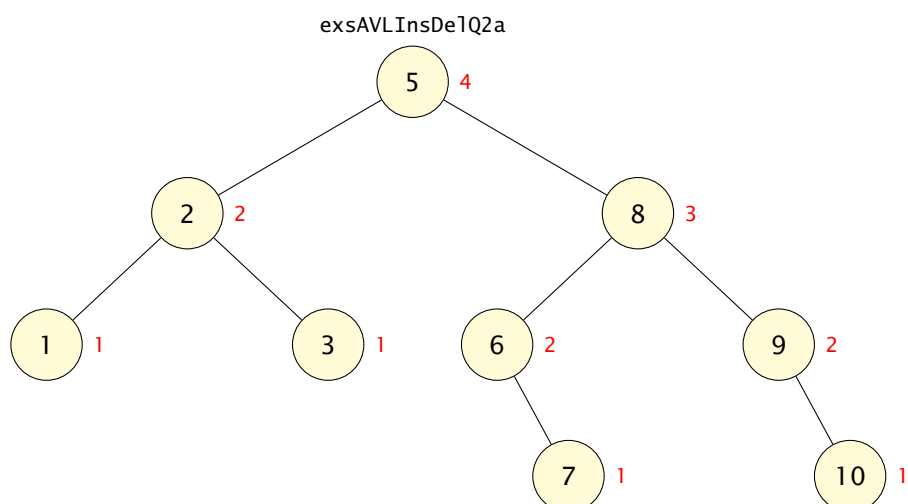
Answer 22 Insert Lists and Delete Items

```
listQ1c = [68,88,61,89,94,50,4,76,66,82,99]
exsAVLInsDe1Q1b = insertListAVLT(listQ1c, EmptyABT())
```



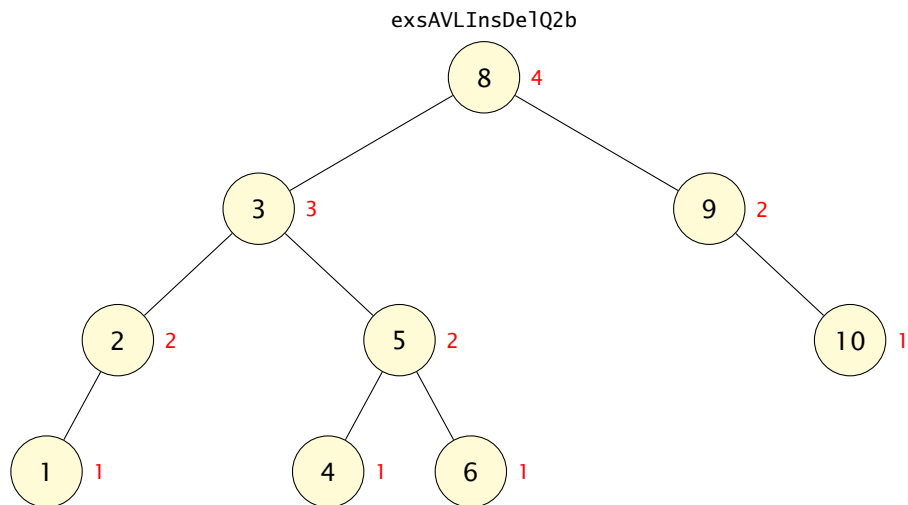
Answer 22 Q2(a) Delete 4th Item

```
listQ1a = [1,2,3,4,5,6,7,8,9,10]
exsAVLInsDe1Q2a \
= deleteAVLT(listQ1a[3], exsAVLInsDe1Q1a)
```



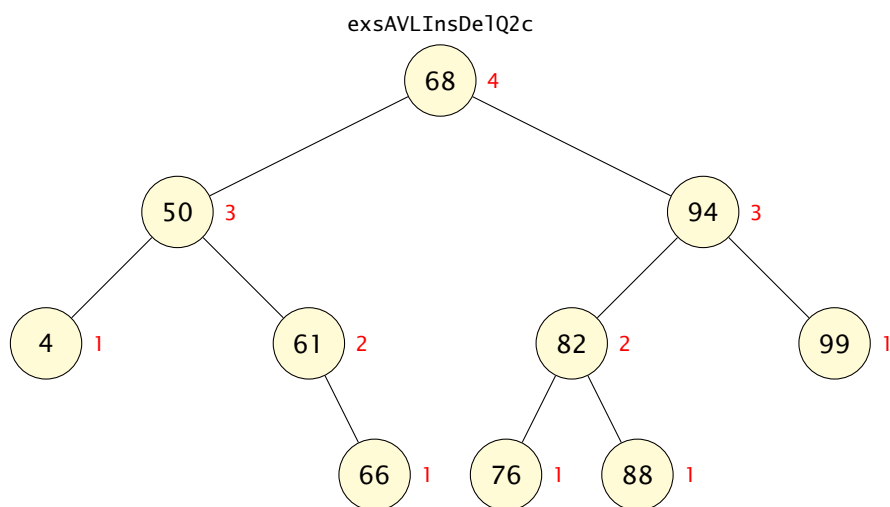
Answer 22 Q2(b) Delete 4th Item

```
listQ1b = [10,9,8,7,6,5,4,3,2,1]
exsAVLInsDe1Q2b \
  = deleteAVLT(listQ1b[3], exsAVLInsDe1Q1b)
```



Answer 22 Insert Lists and Delete Items

```
listQ1c = [68,88,61,89,94,50,4,76,66,82,99]
exsAVLInsDe1Q2c \
  = deleteAVLT(listQ1c[3], exsAVLInsDe1Q1c)
```



[Go to Activity](#)

Activity 23 Deleting Inserted List

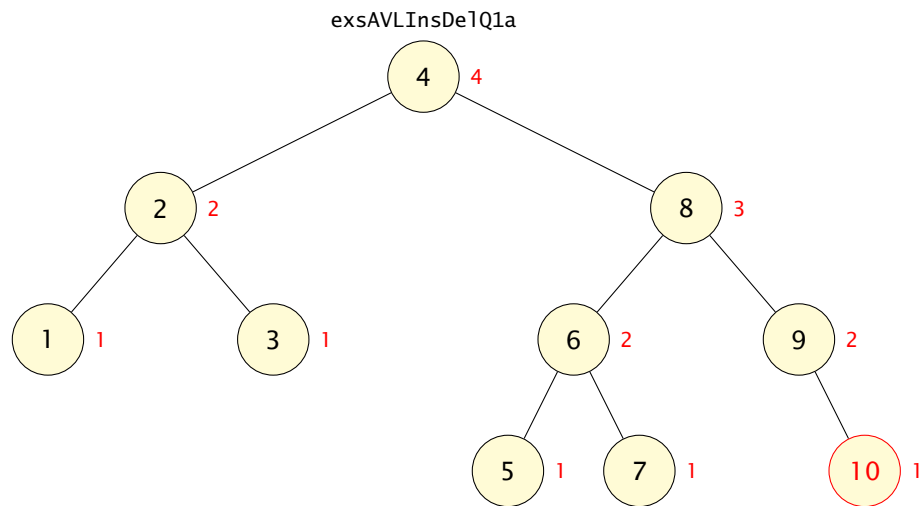
- Using `listQ1a` show that deleting the elements of the list from the tree one by one in reverse order does not result in the reverse sequence of AVL trees

```
listQ1a = [1,2,3,4,5,6,7,8,9,10]
exsAVLInsDe1Q1a = insertListAVLT(listQ1a, EmptyABT())
```

[Go to Answer](#)

Answer 23 Deleting Inserted List

```
listQ1a = [1,2,3,4,5,6,7,8,9,10]
exsAVLInsDe1Q1a = insertListAVLT(listQ1a, EmptyABT())
# delete listQ1a[-1]
```

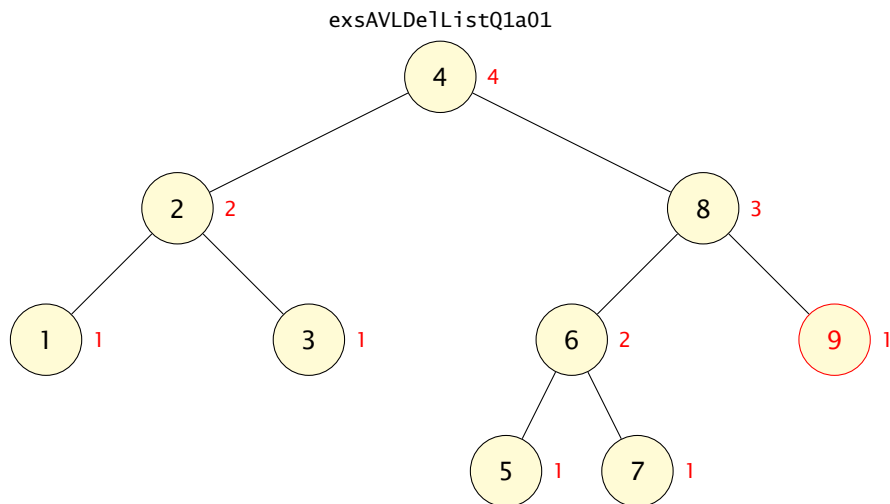


Answer 23 Deleting Inserted List

```

exsAVLDe1ListQ1a01 \
= deleteAVLT(listQ1a[-1], exsAVLInsDe1Q1a)
# delete listQ1a[-2]

```

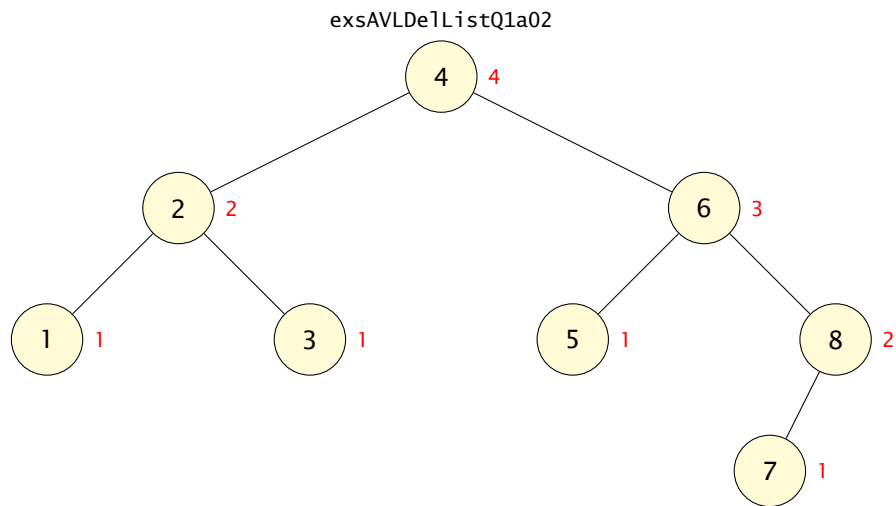


Answer 23 Deleting Inserted List

```

exsAVLDe1ListQ1a02 \
= deleteAVLT(listQ1a[-2], exsAVLDe1ListQ1a01)
# Right rotation about node 8

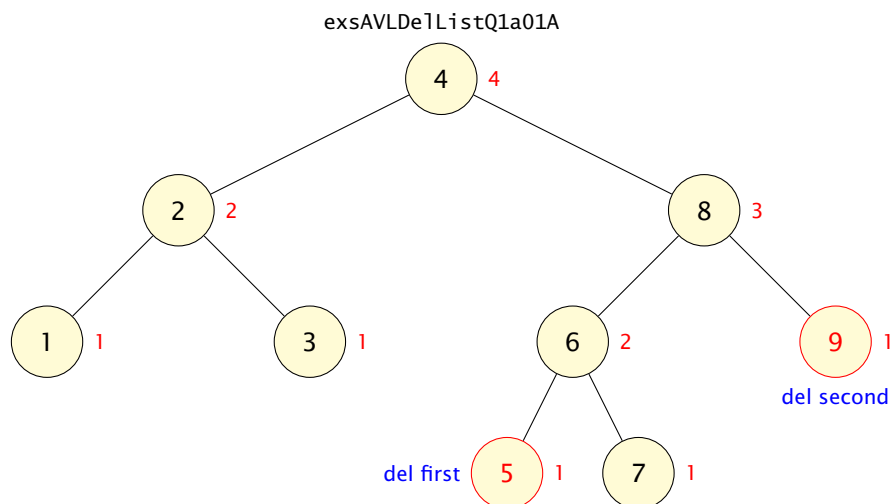
```



Answer 23 Deleting Inserted List

```

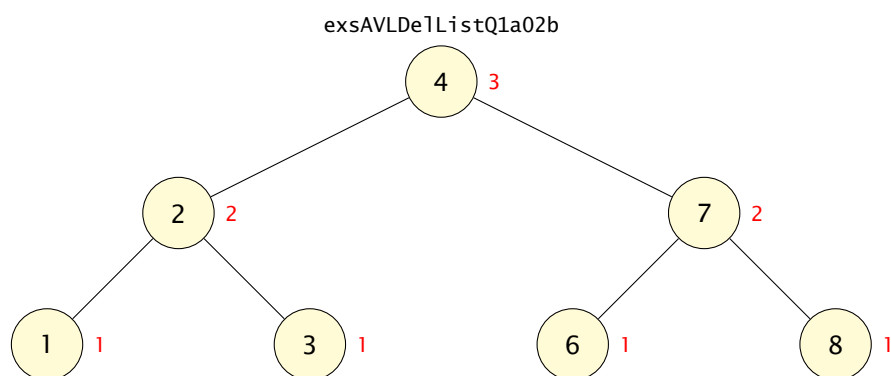
exsAVLDe1ListQ1a01 \
= deleteAVLT(listQ1a[-1], exsAVLInsDe1Q1a)
# delete 5 first followed by 9
  
```



Answer 23 Deleting Inserted List

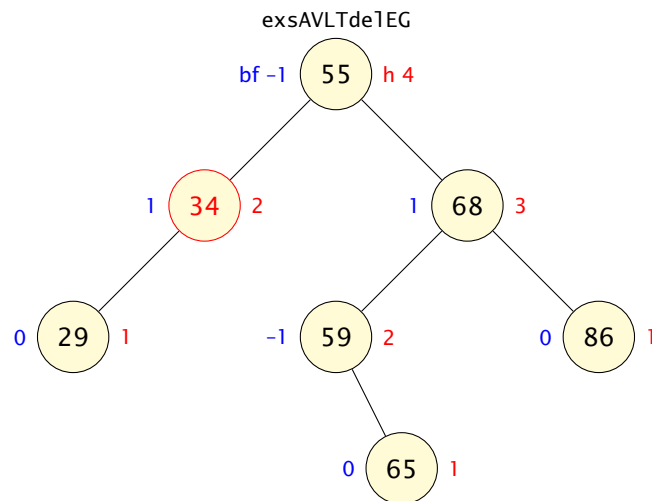
```

exsAVLDe1ListQ1a02a \
= deleteAVLT(listQ1a[-6], exsAVLDe1ListQ1a01)
exsAVLDe1ListQ1a02b \
= deleteAVLT(listQ1a[-2], exsAVLDe1ListQ1a02a)
# Double rotation: left about 6, right about 8
  
```



Activity 24 Delete with Rebalance

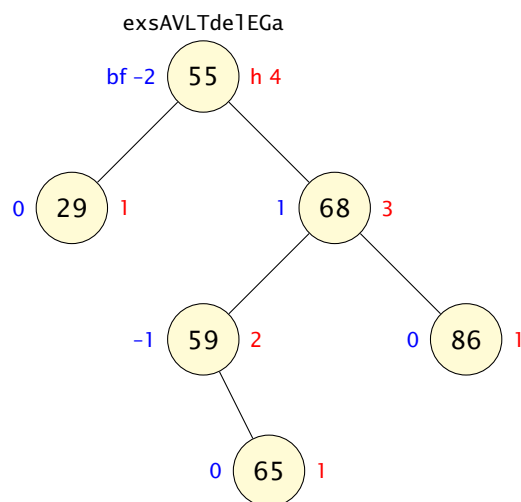
- Example from Specimen Exam (2016) Q 8
- Redraw the tree with **node 34** deleted and tree rebalanced. Note here we have height of empty tree as 0 and singleton node as 1



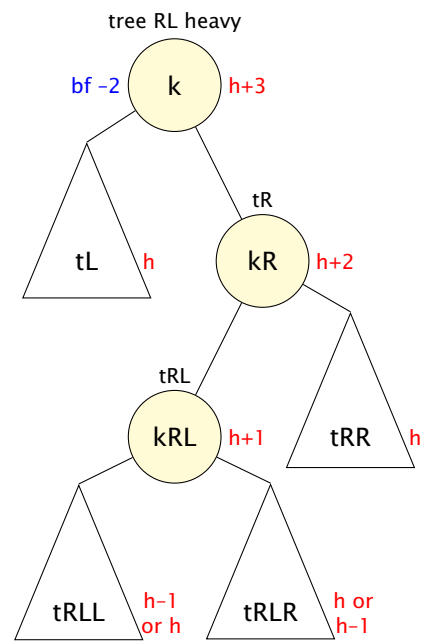
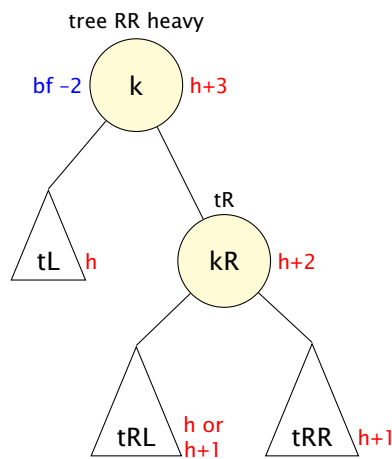
[Go to Answer](#)

Answer 24 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](#)



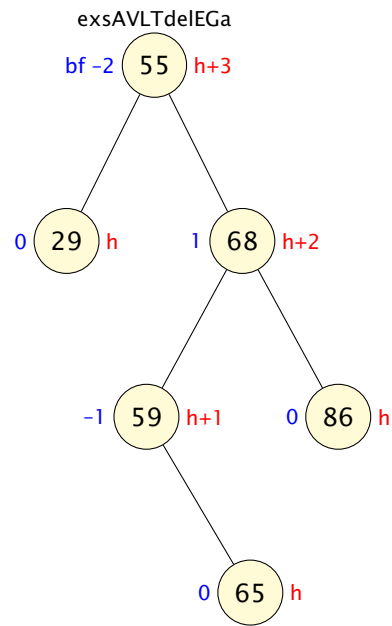
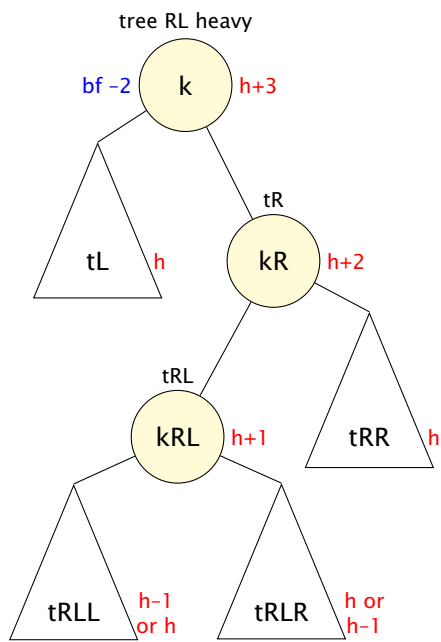
[Go to Activity](#)

- 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 ?

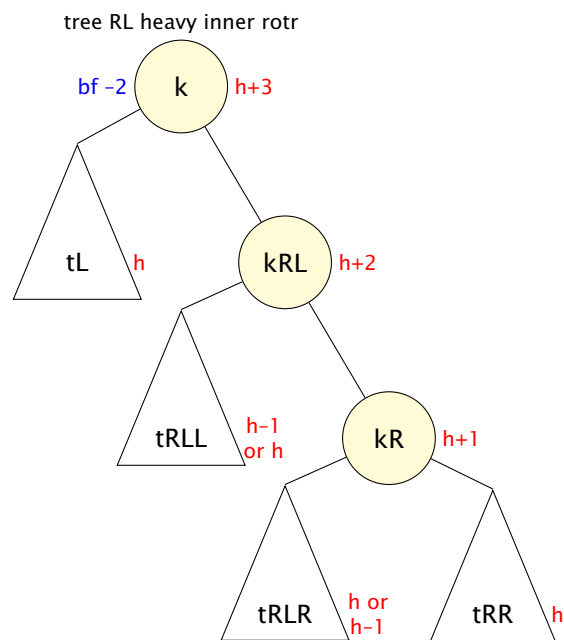
[Go to Activity](#)

- 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

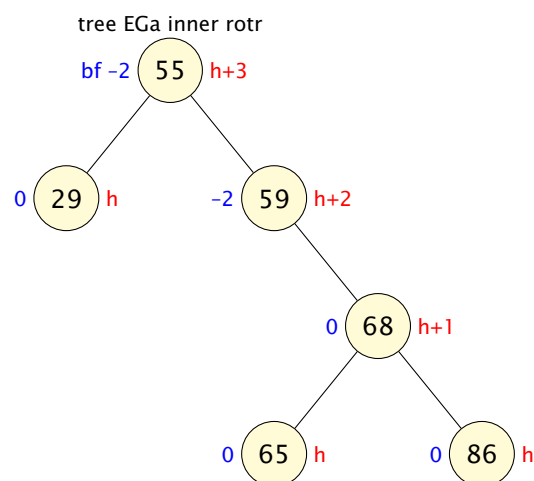
[Go to Activity](#)



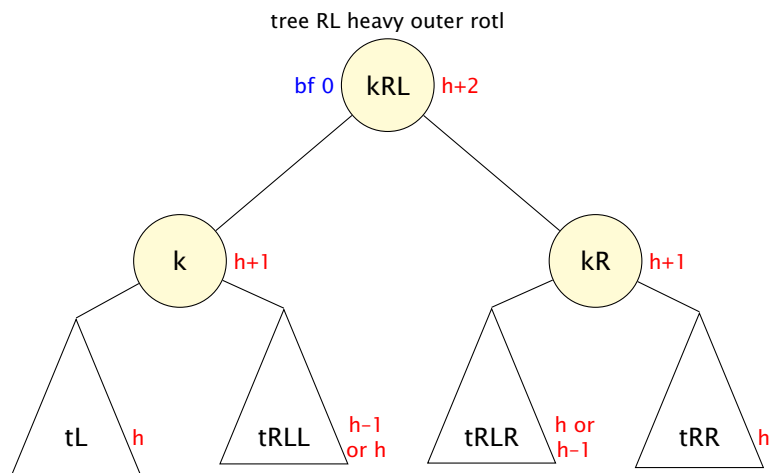
[Go to Activity](#)



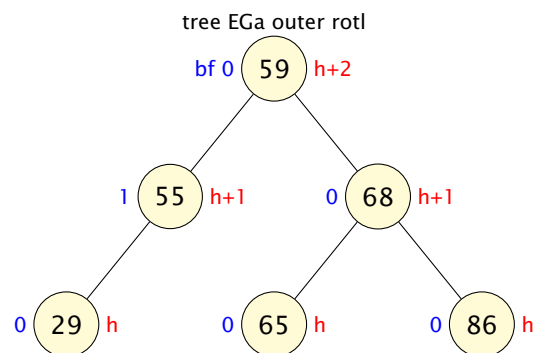
[Go to Activity](#)



[Go to Activity](#)



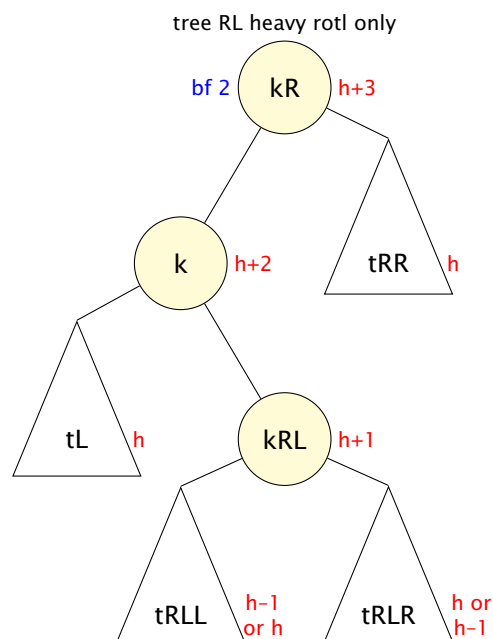
[Go to Activity](#)



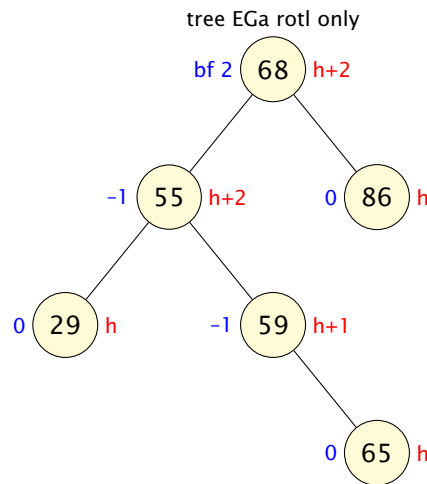
[Go to Activity](#)

- 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](#)

ToC

6.6 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 $h \geq 2$ 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](#)
- $F_0 = 0, F_1 = 1$
- $F_k = F_{k-1} + F_{k-2}, k \geq 2$

- 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 $R_h = F_h, \forall h \geq 2$
- Hence $T_h = F_{h+2} - 1, \forall h \geq 0$
- 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
 $N_1 = 2$
- $N_h = 1 + N_{h-1} + N_{h-2}, h \geq 2$
- 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 $S_h = F_{h+1}, \forall h \geq 2$
- Hence $N_h = F_{h+3} - 1, \forall h \geq 0$
- $P(h) : T_h = F_{h+2} - 1$ 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 $T_h = F_{h+2} - 1$, $\forall h \geq 0$
- There is a closed-form solution for the Fibonacci sequence known as the [Euler-Binet Formula](#) (see also [A formula for Fib\(n\)](#))
- $F_k = \frac{\phi^k - (1 - \phi)^k}{\sqrt{5}}$
- ϕ is the [Golden mean](#)
- $\phi = \frac{1}{\phi - 1} = \frac{1 + \sqrt{5}}{2} \approx 1.61803 \dots$
- Hence $T_h = \frac{\phi^{h+2} - (1 - \phi)^{h+2}}{\sqrt{5}} - 1$
- Since $(1 - \phi) < 1$ then for large h we have
- $T_h = n \approx \frac{\phi^{h+2}}{\sqrt{5}} - 1 \rightarrow \log(\sqrt{5}(n+1)) \approx (h+2) \log \phi$
- Hence in the worst case, the height of a AVL tree is $O(\log n)$

6.6.1 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 $P(n) = F_n = \frac{\phi^n - (1 - \phi)^n}{\sqrt{5}}$

- **Basis for Induction**

- $P(0)$ is true since

$$\frac{\phi^0 - (1 - \phi)^0}{\sqrt{5}} = \frac{1 - 1}{\sqrt{5}} = 0 = F_0$$

- $P(1)$ is true since

$$\frac{\phi^1 - (1 - \phi)^1}{\sqrt{5}} = \frac{\left(\frac{1+\sqrt{5}}{2}\right) - \left(1 - \left(\frac{1+\sqrt{5}}{2}\right)\right)}{\sqrt{5}}$$

- $= 1 = F_1$

- **Induction Hypothesis Step**

- Show $P(j) : 0 \leq j \leq k+1 \Rightarrow P(k+2)$

- $\phi^{k+2} - (1 - \phi)^{k+2} = \phi^2 \phi^k - (1 - \phi)^2 (1 - \phi)^k$ and

- $\phi^2 = \left(\frac{1+\sqrt{5}}{2}\right)^2 = \frac{1}{4}(1 + 2\sqrt{5} + 5) = 1 + \phi$

- $(1 - \phi)^2 = \left(\frac{1-\sqrt{5}}{2}\right)^2 = 1 + (1 - \phi)$ 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 $\forall n \in \mathbb{N} : F_n = \frac{\phi^n - (1 - \phi)^n}{\sqrt{5}}$
- 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 $F_n = \frac{\phi^n - (1 - \phi)^n}{\sqrt{5}}$

[ToC](#)

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 \(1993\)](#)
- Note that this example also shows the use of recursive thinking in practice
- 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 representation of sets uses the [ABTree](#) data type but with generalised versions of the [split](#) and [join](#) functions
- While implementing sets in AVL trees is not directly part of M269, it gives good examples of recursive thinking in an important application
- Note that the [data](#) item is used as the key for a node in the tree — in practice there would be separate key and data
- [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 element [k](#) respectively and the rest of the tree — [splitFirstAVLS](#) is similar to [splitAVLT](#) at line 926 on page 69
- [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 917 on page 69
- [exposeABT](#) takes apart an augmented tree, [t](#) to give ([tL](#), [k](#), [tR](#))
- Here is a reminder of some of the [ABTree](#) constructors and inspectors from file [M269TutorialBinaryTrees2022.py](#)

```

761 def mkEmptyABT() -> ABTree :
762     return EmptyABT()

764 def mkNodeABT(x: T, t1: ABTree, t2: ABTree) -> ABTree :
765     h = 1 + max(getHeightABT(t1), getHeightABT(t2))
766     return NodeABT(h, x, t1, t2)

768 def isEmptyABT(t: ABTree) -> bool :
769     return t == EmptyABT()

```

- And here is the additional inspector `expose` in `M269TutorialBinaryTrees2022AVLSets.py`

```

11 def exposeABT(t: ABTree) -> (ABTree, T, ABTree) :
12     if isEmptyABT(t) :
13         raise RuntimeError("exposeABT_applied_to_EmptyABT()")
14     else :
15         tL = getLeftABT(t)
16         k = getDataABT(t)
17         tR = getRightABT(t)
18         return (tL, k, tR)

```

- `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

```

20 def joinAVLS(k: T, tL: ABTree, tR: ABTree) -> ABTree :
21     if getHeightABT(tL) > getHeightABT(tR) + 1 :
22         return joinRightAVLS(k, tL, tR)
23     elif getHeightABT(tR) > getHeightABT(tL) + 1 :
24         return joinLeftAVLS(k, tL, tR)
25     else :
26         return mkNodeABT(k, tL, tR)

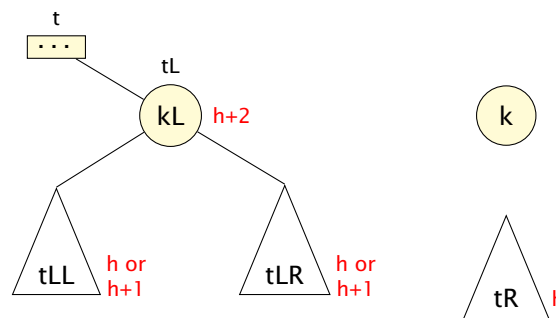
```

- `joinRightAVLS` description is in the following diagrams

```

28 def joinRightAVLS(k: T, tL: ABTree, tR: ABTree) -> ABTree :
29     (tLL, kL, tLR) = exposeABT(tL)
30     if getHeightABT(tLR) <= getHeightABT(tR) + 1 :
31         t1 = mkNodeABT(k, tLR, tR)
32         if getHeightABT(t1) <= getHeightABT(tLL) + 1 :
33             return mkNodeABT(kL, tLL, t1)
34         else :
35             return rotl(mkNodeABT(kL, tLL, (rotr(t1))))
36     else :
37         t2 = joinRightAVLS(k, tLR, tR)
38         t3 = mkNodeABT(kL, tLL, t2)
39         if getHeightABT(t2) <= getHeightABT(tLL) + 1 :
40             return t3
41         else :
42             return rotl(t3)

```



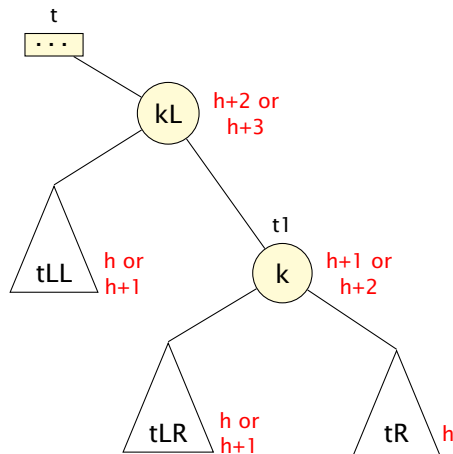
- The base case (line 30 on page 87) of `joinRightAVLS` follows the right spine of `t` to a node `kL` for which

```

getHeightABT(tL) > getHeightABT(tR) + 1
getHeightABT(tLR) <= getHeightABT(tR) + 1

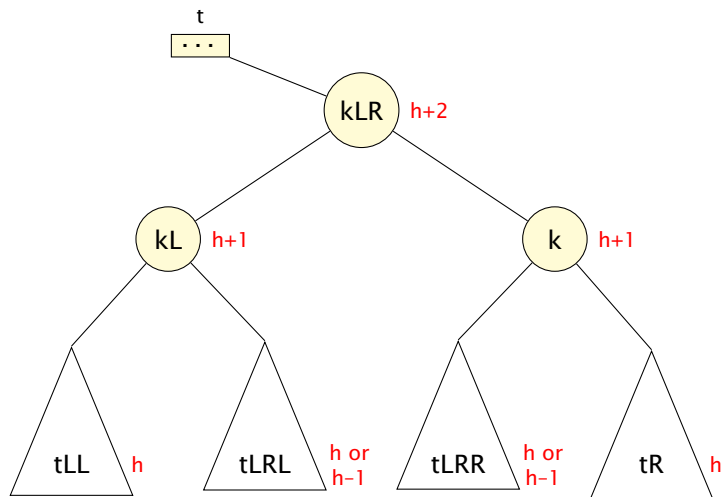
```

- We then connect `tL`, `k` and `tR`



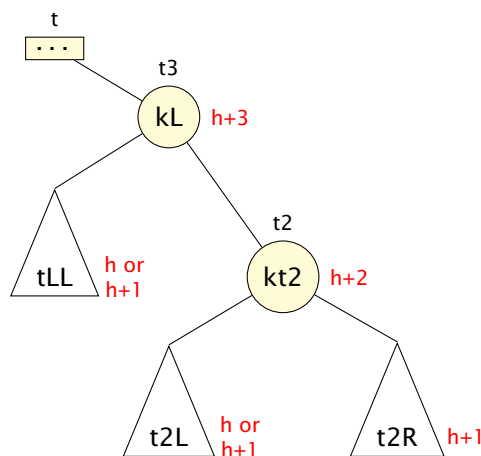
- Needs double rotation if

$$\text{getHeightABT}(t1) > \text{getHeightABT}(tLL) + 1$$



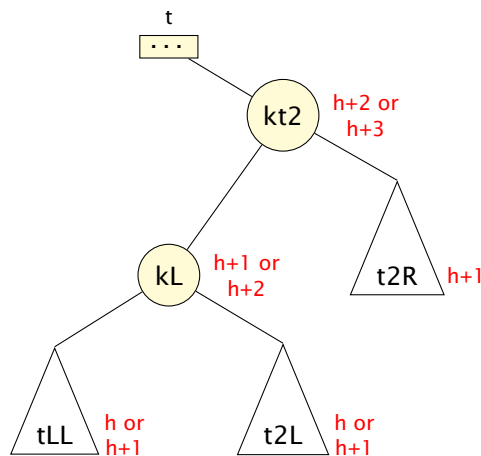
- The recursive case (line 36 on page 87) of `joinRightAVLS` follows the right spine further

$$\text{getHeightABT}(tLR) > \text{getHeightABT}(tR) + 1$$



- A single left rotation is needed if

$$\text{getHeightABT}(t2) > \text{getHeightABT}(tLL) + 1$$



```

44 def joinLeftAVLS(k: T, tL: ABTree, tR: ABTree) -> ABTree :
45     (tRL, kR, tRR) = exposeABT(tR)
46     if getHeightABT(tRL) <= getHeightABT(tL) + 1 :
47         t1 = mkNodeABT(k, tL, tRL)
48         if getHeightABT(t1) <= getHeightABT(tRR) + 1 :
49             return mkNodeABT(kR, t1, tRR)
50         else :
51             return rotr(mkNodeABT(kR, (rotl(t1)), tRR))
52     else :
53         t2 = joinLeftAVLS(k, tL, tRL)
54         t3 = mkNodeABT(kR, t2, tRR)
55         if getHeightABT(t2) <= getHeightABT(tRR) + 1 :
56             return t3
57         else :
58             return rotr(t3)

```

Activity 25 joinLeftAVLS Diagrams

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

[Go to Answer](#)

Answer 25 joinLeftAVLS Diagrams

- TODO: Answer 25 joinLeftAVLS Diagrams

[Go to Activity](#)

Activity 26 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 ?

```

def joinLeftAVLS(k: T, tL: ABTree, tR: ABTree) -> ABTree :
    (tRL, kR, tRR) = exposeABT(tR)
    if getHeightABT(tRL) <= getHeightABT(tL) + 1 :
        t1 = mkNodeABT(k, tL, tRL)
        if getHeightABT(t1) <= getHeightABT(tRR) + 1 :
            return mkNodeABT(kR, t1, tRR)
        else :
            return rotr(mkNodeABT(kR, (rotl(t1)), tRR))
    else :
        t2 = joinRightAVLS(k, tL, tRL)
        t3 = mkNodeABT(kR, t2, tRR)
        if getHeightABT(t2) <= getHeightABT(tRR) + 1 :
            return t3
        else :
            return rotr(t3)

```

[Go to Answer](#)

Answer 26 joinLeftAVLS Bug

- TODO: Answer 26 joinLeftAVLS Bug

[Go to Activity](#)

```

60 def splitLastAVLS(t: ABTree) -> (ABTree,T) :
61   (tL,k,tR) = exposeABT(t)
62   if isEmptyABT(tR) :
63     return (tL,k)
64   else :
65     (tR1,k1) = splitLastAVLS(tR)
66     return (joinAVLS(k,tL,tR1),k1)

68 def splitFirstAVLS(t: ABTree) -> (ABTree,T) :
69   (tL, k, tR) = exposeABT(t)
70   if isEmptyABT(tL) :
71     return (tR,k)
72   else :
73     (tL1,k1) = splitFirstAVLS(tL)
74     return (joinAVLS(k,tL1,tR),k1)

```

```

76 def splitAVLS(k: T,t: ABTree) -> (ABTree,bool,ABTree) :
77   if isEmptyABT(t) :
78     return (mkEmptyABT(),False,mkEmptyABT())
79   else :
80     (tL, k1, tR) = exposeABT(t)
81     if k == k1 :
82       return (tL,True,tR)
83     elif k < k1 :
84       (tLL, b, tLR) = splitAVLS(k,tL)
85       return (tLL, b, (joinAVLS(k1,tLR,tR)))
86     else :
87       (tRL, b, tRR) = splitAVLS(k,tR)
88       return ((joinAVLS(k1,tL,tRL)), b, tRR)

```

```

90 def join2AVLS(tL: ABTree,tR: ABTree) -> ABTree :
91   if isEmptyABT(tL) :
92     return tR
93   else :
94     (tL1, k) = splitLastAVLS(tL)
95     return joinAVLS(k,tL1,tR)

```

ToC

7.2 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
- `disjoint(t1,t2)` takes two AVL trees and returns `True` if and only if they have no members in common
- `differenceAVLS t1 t2` takes two AVL trees and returns the elements that are in `t1` but not `t2`
- `subsetAVLS(t1,t2)` takes two AVL trees and returns `True` if and only if every member of `t1` is a member of `t2`

```

97 def insertAVLS(t: ABTree, k: T) -> ABTree :
98     (tL, found, tR) = splitAVLS(k, t)
99     return joinAVLS(k, tL, tR)

101 def deleteAVLS(t: ABTree, k: T) -> ABTree :
102     (tL, found, tR) = splitAVLS(k, t)
103     return join2AVLS(tL, tR)

105 def insertListAVLS(t: ABTree, xs: [T]) -> ABTree :
106     if xs == [] :
107         return t
108     else :
109         return insertListAVLS(insertAVLS(t, xs[0]), xs[1:])

111 def setFromListAVLS(xs: [T]) -> ABTree :
112     return insertListAVLS(mkEmptyABT(), xs)

```

```

114 def unionAVLS(t1: ABTree, t2: ABTree) -> ABTree :
115     if isEmptyABT(t1) :
116         return t2
117     elif isEmptyABT(t2) :
118         return t1
119     else :
120         (t2L, k2, t2R) = exposeABT(t2)
121         (t1L, found, t1R) = splitAVLS(k2, t1)
122         tL = unionAVLS(t1L, t2L)
123         tR = unionAVLS(t1R, t2R)
124         return joinAVLS(k2, tL, tR)

```

- `unionAVLS(t1, t2)` returns the set of all members of `t1` or `t2` (or both)

```

126 def intersectAVLS(t1: ABTree, t2: ABTree) -> ABTree :
127     if isEmptyABT(t1) :
128         return mkEmptyABT()
129     elif isEmptyABT(t2) :
130         return mkEmptyABT()
131     else :
132         (t2L, k2, t2R) = exposeABT(t2)
133         (t1L, found, t1R) = splitAVLS(k2, t1)
134         tL = intersectAVLS(t1L, t2L)
135         tR = intersectAVLS(t1R, t2R)
136         if found :
137             return joinAVLS(k2, tL, tR)
138         else :
139             return join2AVLS(tL, tR)

```

- `intersectAVLS(t1, t2)` returns the set of all members of both `t1` and `t2`
- Notice it needs the `if` statement to check that a member of `t2` is a member of `t1`

```

141 def disjointAVLS(t1: ABTree, t2: ABTree) -> ABTree :
142     if isEmptyABT(t1) :
143         return True
144     elif isEmptyABT(t2) :
145         return True
146     else :
147         (t2L, k2, t2R) = exposeABT(t2)
148         (t1L, found, t1R) = splitAVLS(k2, t1)
149         return (not found
150                 and disjointAVLS(t1L, t2L)
151                 and disjointAVLS(t1R, t2R))

```

- `disjoint(t1, t2)` returns `True` if there are no elements in common
- If an element in common is found then `False` is returned
- Note that the behaviour of `splitAVLS()` ensures the search space is reduced at each recursive call

```

153 def differenceAVLS(t1: ABTree, t2: ABTree) -> ABTree :
154     if isEmptyABT(t1) :

```

```

155     return mkEmptyABT()
156 elif isEmptyABT(t2) :
157     return t1
158 else :
159     (t2L, k2, t2R) = exposeABT(t2)
160     (t1L, found, t1R) = splitAVLS(k2,t1)
161     tL = differenceAVLS(t1L,t2L)
162     tR = differenceAVLS(t1R,t2R)
163     return join2AVLS(tL,tR)

```

- `differenceAVLS(t1,t2)` returns the set of members of `t1` that are not in `t2`
- On first reading it may be surprising there is no `if` statement
- Remember the behaviour of `splitAVLS()`

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

- Probably the simplest solution at this stage is to use `inOrderABT()`

```

181 def setToAscList(t) :
182     return inOrderABT(t)

```

```

Python3>>> list3 = [2,1,4,3,6,5,8,7,10,9]
Python3>>> t10 = setFromListAVLS(list3)
Python3>>> type(t10)
<class 'M269TutorialBinaryTrees2022.NodeABT'>
Python3>>> list4 = setToAscList(t10)
Python3>>> list4
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Python3>>>

```

Answer 27 Set to Ascending List

- Of course, someone from the pure functional programming world would define a higher order function to capture the recursion pattern with `setFoldr()` (this is not part of M269)

```

184 def setFoldr(f,z,t) :
185     def go(y,t) :
186         if isEmptyABT(t) :
187             return y
188         else :
189             (tL,x,tR) = exposeABT(t)
190             return (go(f(x,(go(y,tR))),tL))
191     return go(z,t)

193 def setToAscListA(t) :
194     def cons(x,xs) :
195         return ([x] + xs)
196     return setFoldr(cons,[],t)

```

[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

```

198 def setEquality(t1,t2) :
199     list1 = setToAscList(t1)
200     list2 = setToAscList(t2)
201     return (len(list1) == len(list2)
202             and list1 == list2)

```

```

Python3>>> list1
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Python3>>> t1 = setFromListAVLS(list1)
Python3>>> list10
[2, 1, 4, 3, 6, 5, 8, 7, 10, 9]
Python3>>> t10 = setFromListAVLS(list10)
Python3>>> t1 == t10
False
Python3>>> setEquality(t1,t10)
True
Python3>>>

```

[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

```

165 def subsetAVLS(t1: ABTree,t2: ABTree) -> bool :
166     if isEmptyABT(t1) :
167         return True
168     elif isEmptyABT(t2) :
169         return False
170     else :
171         (t1L, k1, t1R) = exposeABT(t1)
172         (t2L, found, t2R) = splitAVLS(k1,t2)
173         return (found
174                 and subsetAVLS(t1L,t2L)
175                 and subsetAVLS(t1R,t2R))

```

- How does this work ?
- The recursive case at line 170 checks that the key at the root of `t1` is in `t2` and recursively checks the sub-trees
- `splitAVLS()` ensures that the subtrees are the correct ones to be checked

[Go to Activity](#)[ToC](#)

7.3 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](#)

Operation	Python		Haskell
	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

[ToC](#)

Commentary 5

5 Binary Tree Exercises

- Binary Tree shapes
- Generating Binary Trees
- Catalan Numbers (advanced)

[ToC](#)

8 Binary Tree Common Exercises

- This section contains some common exercises used in *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 using recursion
- Further questions may be added to this section

8.1 Binary Tree Shapes

Activity 30 Shape Exercises

- `isSameShape(t1,t2)` takes two binary trees and returns `True` if they have the same shape
- `isMirrorShape(t1,t2)` takes two binary trees and returns `True` if they are a mirror of each other
- `isSymmetric(t)` takes a binary tree and returns `True` if it is symmetric
- `genMirrorShape(t)` takes a binary tree and returns the mirror of the tree

[Go to Answer](#)

Answer 30 Shape Exercises — `isSameShape(t1,t2)`

- `isSameShape(t1,t2)` takes two binary trees and returns `True` if they have the same shape

```

206 def isSameShape(t1: ABTree, t2: ABTree) -> bool :
207     if isEmptyABT(t1) and isEmptyABT(t2) :
208         return True
209     elif isEmptyABT(t1) or isEmptyABT(t2) :
210         return False
211     else :
212         (t1L, k1, t1R) = exposeABT(t1)
213         (t2L, k2, t2R) = exposeABT(t2)
214         return (isSameShape(t1L, t2L)
215                 and isSameShape(t1R, t2R))

```

Answer 30 Shape Exercises — `isMirrorShape(t1,t2)`

- `isMirrorShape(t1,t2)` takes two binary trees and returns `True` if they are a mirror of each other

```

217 def isMirrorShape(t1: ABTree, t2: ABTree) -> bool :
218     if isEmptyABT(t1) and isEmptyABT(t2) :
219         return True
220     elif isEmptyABT(t1) or isEmptyABT(t2) :
221         return False
222     else :
223         (t1L, k1, t1R) = exposeABT(t1)
224         (t2L, k2, t2R) = exposeABT(t2)
225         return (isMirrorShape(t1L, t2R)
226                 and isMirrorShape(t1R, t2L))

```

Answer 30 Shape Exercises — `isSymmetric(t)`

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

```

228 def isSymmetric(t: ABTree) -> bool :
229     return isMirrorShape(t,t)

```

Answer 30 Shape Exercises — `genMirrorShape(t)`

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

```

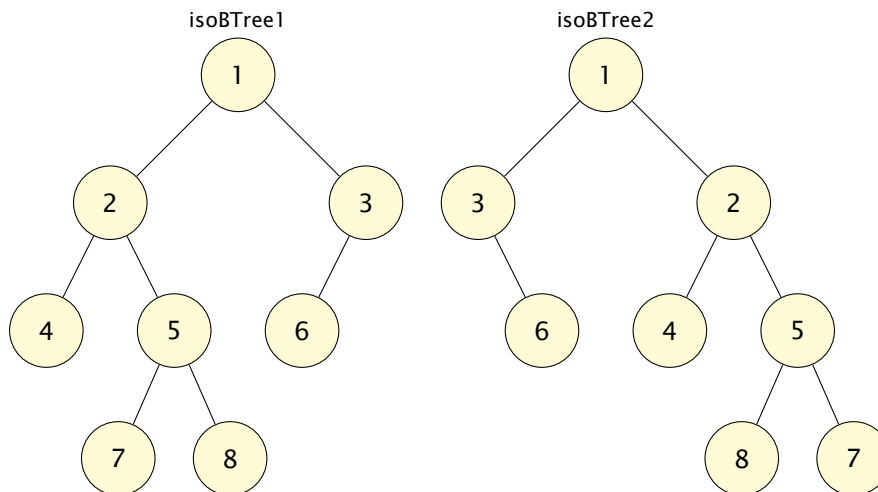
231 def genMirrorShape(t: ABTree) -> ABTree :
232   if isEmptyABT(t) :
233     return mkEmptyABT()
234   else :
235     (tL,x,tR) = exposeABT(t)
236     return (mkNodeABT(x, genMirrorShape(tR),
237                       genMirrorShape(tL)))

```

[Go to Activity](#)

8.1.1 Isomorphic Binary Trees

- Two binary trees are isomorphic if one can be obtained from the other by flipping the left and right subtrees. Two empty trees are isomorphic
- See [Tree isomorphism problem](#)
- See [Is the recursive approach to binary tree isomorphism actually linear?](#)



- `isoBTree1`, `isoBTree2` are isomorphic with the following flips:
(2,3), (`EmptyBTree`, 6), (7,8)

```

def isIsomorphic(t1 : ABTree, t2 : ABTree) -> bool :
  if isEmptyABT(t1) and isEmptyABT(t2) :
    return True
  elif isEmptyABT(t1) or isEmptyABT(t2) :
    return False
  else :
    (t1L,k1,t1R) = exposeABT(t1)
    (t2L,k2,t2R) = exposeABT(t2)
    if (k1 != k2) :
      return False
    else :
      return ((isIsomorphic(t1L, t2L) and isIsomorphic(t1R,t2R))
              or
              (isIsomorphic(t1L, t2R) and isIsomorphic(t1R,t2L))
              )

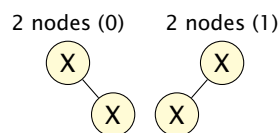
```

[ToC](#)

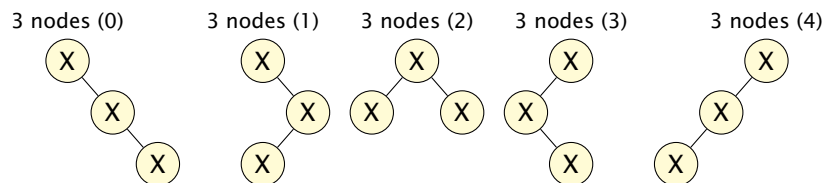
8.2 Generating Binary Trees

- The aim is to generate the shapes of all possible trees given a number of nodes

- First sketch a few trees to spot any pattern
- Write down a recurrence relation for the number of binary tree shapes with n nodes based on the number of tree shapes for less than n nodes
- Write a function `genBTs(x,n)` given a value x and an integer n generates the Python representation of all shapes of binary trees with n nodes with x at each node
- 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



- Let C_n be the number of binary tree shapes with n nodes then from the above diagrams we have:
- $C_0 = 1$
- $C_1 = 1$
- $C_2 = 2$
- $C_3 = 5$
- **Eureka insight** for a tree with n nodes if the left subtree has i nodes then the right subtree must have $n - i - 1$ nodes and i can range over 0 to $n - 1$
- The left and right subtrees must have C_i and C_{n-i-1} different possible shapes
- and there are n possible values for i from 0 to $n - 1$

- Hence
$$C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}$$

- $$C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1} = \sum_{i=1}^n C_{i-1} C_{n-i}$$

- Alternatively
$$C_{n+1} = \sum_{i=0}^n C_i C_{n-i}$$

- Check $C_1 = C_0 C_0 = 1 \times 1 = 1$
- $C_2 = C_0 C_1 + C_1 C_0 = 1 \times 1 + 1 \times 1 = 2$

- $C_3 = C_0C_2 + C_1C_1 + C_2C_0 = 1 \times 2 + 1 \times 1 + 2 \times 1 = 5$
- $C_4 = C_0C_3 + C_1C_2 + C_2C_1 + C_3C_0$
 $= 1 \times 5 + 1 \times 2 + 2 \times 1 + 5 \times 1 = 14$
- The C_n are known as the [Catalan numbers](#)
- The simple recursive definition of [genBTs](#) follows from the recurrence relation directly
- Uses [Python List Comprehensions](#) see below
- 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 \(1997, section 2.3.4.4\)](#)

```

239 def genABTs(x: T, n: int) -> [ABTree] :
240     if n == 0 :
241         return [mkEmptyABT()]
242     elif n == 1 :
243         return [mkNodeABT(x, mkEmptyABT(), mkEmptyABT())]
244     else :
245         ts = ([mkNodeABT(x, leftT, rightT)
246               for (nu, nv) in splitsInt(n)
247               for leftT in genABTs(x, nu)
248               for rightT in genABTs(x, nv)])
249         return ts
251 def splitsInt(n: int) -> [(int, int)] :
252     prns = [(i, n - i - 1) for i in range(n)]
253     return prns

```

- [List comprehensions](#) (tutorial), [List comprehensions](#) (reference) — a neat way of expressing iterations over a list, came from [Miranda](#) (see [Wikipedia: List comprehension](#))
- Example: Square the even numbers between 0 and 9

```

Python3>>> [x ** 2 for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]
Python3>>> [(x, y) for x in range(4)
...           for y in range(4)
...           if x % 2 == 0
...           and y % 3 == 0]
[(0, 0), (0, 3), (2, 0), (2, 3)]
Python3>>>

```

- In general

```

[expr for target1 in iterable1 if cond1
 for target2 in iterable2 if cond2 ...
 for targetN in iterableN if condN ]

```

8.3 Catalan Numbers

- As with many other problems, it may be easier to find a recursive relation or recurrence for a problem and harder to find an efficient calculation.
- For the [Catalan numbers](#) it is possible to find a closed (non-recursive) expression for the Catalan numbers
- Below is a derivation of a closed expression — this is not part of M269 and is included for interest — the derivation uses a bit more Maths than the rest of these notes but it is explained as we progress

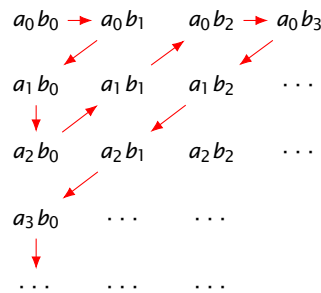
- This derivation is from [Spivey \(2019, page 208\)](#) and [Wilf \(1994, page 44\)](#)

8.3.1 Cauchy Product

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

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

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



$$\begin{aligned} & \left(\sum_{i=0}^{\infty} a_i x^i \right) \left(\sum_{j=0}^{\infty} b_j x^j \right) \\ &= (a_0 + a_1 x + a_2 x^2 + \dots)(b_0 + b_1 x + b_2 x^2 + \dots) \\ &= a_0 b_0 + (a_0 b_1 + a_1 b_0)x + (a_0 b_2 + a_1 b_1 + a_2 b_0)x^2 + \dots \end{aligned}$$

- See [Wikipedia: Cauchy product](#)

$$\bullet \text{ If we have } c(x) = \sum_{i=0}^{\infty} c_i x^i$$

$$\begin{aligned} \bullet (c(x))^2 &= \left(\sum_{i=0}^{\infty} c_i x^i \right) \left(\sum_{j=0}^{\infty} c_j x^j \right) \\ &= \sum_{n=0}^{\infty} \left(\sum_{k=0}^n c_k c_{n-k} \right) x^n \end{aligned}$$

- This result is used in finding a closed form for the [Catalan numbers](#)
- Based on [Mike Spivey 2013](#)

[ToC](#)

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

- $$\sum_{n=0}^{\infty} C_{n+1} x^n = \sum_{n=0}^{\infty} \left(\sum_{k=0}^n C_k C_{n-k} \right) x^n$$

- $$\sum_{n=0}^{\infty} C_{n+1} x^n = \sum_{n=0}^{\infty} \left(\sum_{k=0}^n C_k C_{n-k} \right) x^n$$

- $$\frac{1}{x} \sum_{n=0}^{\infty} C_{n+1} x^{n+1} = (c(x))^2 \text{ by Cauchy product}$$

- $$\frac{1}{x} \left(\sum_{n=0}^{\infty} C_n x^n - C_0 \right) = (c(x))^2$$

- $$\frac{1}{x} (c(x) - 1) = (c(x))^2$$

- $$x(c(x))^2 - c(x) + 1 = 0$$

- $$c(x) = \frac{1 \pm \sqrt{1 - 4x}}{2x}$$

- We know $c(0) = C_0 = 1$

- $$c(x) = \frac{1 \pm \sqrt{1 - 4x}}{2x}$$

- We know $c(0) = C_0 = 1$

- Applying **L'Hôpital's rule**

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

- $$\lim_{x \rightarrow 0^+} \frac{1 - \sqrt{1 - 4x}}{2x} = \lim_{x \rightarrow 0^+} \frac{2(1 - 4x)^{-\frac{1}{2}}}{2} = 1$$

- Hence
$$c(x) = \frac{1 - \sqrt{1 - 4x}}{2x}$$

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

- $c(x) = \frac{1}{2x}(1 - \sqrt{1 - 4x}) = \frac{1}{2x} \left(1 - \sum_{n=0}^{\infty} \binom{1/2}{n} (-4x)^n \right)$

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

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

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

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

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

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

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

[ToC](#)

Commentary 6

6 Tutorial End, References and Colophon

- Future work and dates
- References to other Python texts or documentation
- References to other computing material
- Article version has the full references and bibliography with back references
- **Colophon**
- LaTeX with Beamer, Listings and other packages
- Index of Python code and diagrams
- PGF/TikZ for the diagrams
- External copies of the diagrams as PDF with tight bounding boxes are available

[ToC](#)

9 Future Work

- Hashing and hash tables
- Binary search trees, height balanced binary search trees, AVL trees
- Graph algorithms
- Greedy algorithms
- Logic, Computability
- Future dates for tutorials and TMAs

[ToC](#)

10 Web Sites & References

10.1 Python Web Links & References

- [Lutz \(2013\)](#) — one of the best introductory books

- **Lutz (2011)** — a more advanced book
- **Martelli et al. (2022)**
- **Ramalho (2022)** a more advanced book
- **Python 3 Documentation** <https://docs.python.org/3/>
- **Python Style Guide PEP 8** <https://www.python.org/dev/peps/pep-0008/> (Python Enhancement Proposals)

[ToC](#)

10.2 Haskell Web Links & References

- **Haskell Language** <https://www.haskell.org>
- **HaskellWiki** <https://wiki.haskell.org/Haskell>
- **Learn You a Haskell for Great Good!** <http://learnyouahaskell.com> — very readable introduction to Haskell
- **Real World Haskell** <http://book.realworldhaskell.org> — more advanced
- **Thompson (2011)** — a good text for functional programming for beginners
- **Bird and Wadler (1988); Bird (1998, 2014)** — one of the best introductions but tough in parts, requires some mathematical maturity — the three books are in effect different editions
- **Bird and Gibbons (2020)** — developing the algorithms in a purely functional way
- **Functors, Applicatives, and Monads in Pictures** http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html — a very good outline with cartoons
- **Typeclassopedia** <https://wiki.haskell.org/Typeclassopedia> — a more formal introduction to Functors, Applicatives and Monads
- **Haskell Wikibook** <https://en.wikibooks.org/wiki/Haskell>

[ToC](#)

References

Abelson, Harold and Gerald Jay Sussman (1984). *Structure and Interpretation of Computer Programs*. MIT Press, first edition. URL <http://mitpress.mit.edu/sicp/>.

Abelson, Harold and Gerald Jay Sussman (1996). *Structure and Interpretation of Computer Programs*. MIT Press, second edition. ISBN 0262510871. URL <http://mitpress.mit.edu/sicp/>.

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/>. 85

Adelson-Velskii, G M and E M Landis (1962). An algorithm for the organization of information. In *Doklady Akademii Nauk SSSR*, volume 146, pages 263–266. Translated from *Soviet Mathematics — Doklady*; 3(5), 1259–1263. 51

- Azmoodeh, Manoochehr (1990). *Abstract Data Types and Algorithms*. Palgrave Macmillan, second edition. ISBN 0333512103.
- Bird, Richard (1998). *Introduction to Functional Programming using Haskell*. Prentice Hall, second edition. ISBN 0134843460. 103
- Bird, Richard (2014). *Thinking Functionally with Haskell*. Cambridge University Press. ISBN 1107452643. URL <https://www.cs.ox.ac.uk/publications/books/functional/>. 103
- Bird, Richard and Jeremy Gibbons (2020). *Algorithm Design with Haskell*. Cambridge University Press. ISBN 9781108869041. URL <https://www.cs.ox.ac.uk/publications/books/adwh/>. 103
- Bird, Richard and Phil Wadler (1988). *Introduction to Functional Programming*. Prentice Hall, first edition. ISBN 0134841972. 103
- 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. 85
- Böhm, Corrado and Giuseppe Jacopini (1966). Flow diagrams, Turing Machines and Languages with Only Two Formation Rules. *Communications of the ACM*, 9(5):366–371.
- 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>. 42
- Cormen, Thomas H.; Charles E. Leiserson; Ronald L. Rivest; and Clifford Stein (2022). *Introduction to Algorithms*. MIT Press, fourth edition. ISBN 9780262046305. URL <https://mitpress.mit.edu/books/introduction-algorithms-fourth-edition>. 42
- Dromey, R.Geoff (1982). *How to Solve it by Computer*. Prentice-Hall. ISBN 0134340019.
- Dromey, R.Geoff (1989). *Program Derivation: The Development of Programs from Specifications*. Addison Wesley. ISBN 0201416247.
- 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. 86
- Knuth, D.E. (1998). *The Art of Computer Programming Vol. 3: Sorting and Searching*. The Art of Computer Programming: Sorting and Searching. Addison Wesley, second edition. ISBN 0201896850. URL <https://www-cs-faculty.stanford.edu/~knuth/taocp.html>.
- Knuth, Donald E. (1997). *The Art of Computer Programming Vol. 1: Fundamental Algorithms*. Addison Wesley, third edition. ISBN 0201896834. 98
- Knuth, Donald E. (2011). *The Art of Computer Programming Vol. 4A: Combinatorial Algorithms, Part 1*. Addison Wesley, first edition. ISBN 0201038048. 98
- 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 (2009). *Learning Python*. O'Reilly, fourth edition. ISBN 0596158068.
- Lutz, Mark (2011). *Programming Python*. O'Reilly, fourth edition. ISBN 0596158106. URL <http://learning-python.com/books/about-pp4e.html>. 103
- Lutz, Mark (2013). *Learning Python*. O'Reilly, fifth edition. ISBN 1449355730. URL <http://learning-python.com/books/about-lp5e.html>. 102
- Marlow, Simon and Simon Peyton Jones (2010). Haskell Language and Library Specification. Web. URL http://www.haskell.org/haskellwiki/Language_and_library_specification.
- Martelli, Alex; Anna Ravenscroft; and Steve Holden (2017). *Python in a Nutshell: A Desktop Quick Reference*. O'Reilly, third edition. ISBN 144939292X.
- Martelli, Alex; Anna Martelli Ravenscroft; Steve Holden; and Paul McGuire (2022). *Python in a Nutshell: A Desktop Quick Reference*. O'Reilly, fourth edition. ISBN 1098113551. 103
- Merritt, SM and KK Lau (1997). A logical inverted taxonomy of sorting algorithms. In *Proceedings of the Twelfth International Symposium on Computer and Information Sciences*, pages 576–583. Citeseer.
- Merritt, Susan M (1985). An inverted taxonomy of sorting algorithms. *Communications of the ACM*, 28(1):96–99.
- 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 <https://runestone.academy/ns/books/published/pythonds/index.html>. 66
- O'Sullivan, Bryan; John Goerzen; and Donald Stewart (2008). *Real World Haskell*. O'Reilly, first edition. ISBN 0596514980. URL <http://book.realworldhaskell.org/>. 94
- Ramalho, Luciano (2022). *Fluent Python: Clear, Concise, and Effective Programming*. O'Reilly. ISBN 1492056359. 103
- Sannella, Donald; Michael Fourman; Haoran Peng; and Philip Wadler (2022). *Introduction to Computation: Haskell, Logic and Automata*. Springer. ISBN 3030769070. 31
- Spivak, Michael (2008). *Calculus*. Publish or Perish, fourth edition. ISBN 0914098918. URL <http://www.mathpop.com/>. 99
- Spivey, Michael Z (2019). *The Art of Proving Binomial Identities*. CRC Press. ISBN 0815379420. 99
- Sussman, Julie (1985a). *Instructor's Manual to Accompany Structure and Interpretation of Computer Programs*. MIT Press. ISBN 0262 691019. URL <http://mitpress.mit.edu/sites/default/files/sicp/index.html>.
- Sussman, Julie (1985b). *Instructor's Manual to Accompany Structure and Interpretation of Computer Programs*. MIT Press, second edition. ISBN 0262 692201. URL <http://mitpress.mit.edu/sites/default/files/sicp/index.html>.
- Thompson, Simon (2011). *Haskell the Craft of Functional Programming*. Addison Wesley, third edition. ISBN 0201882957. URL <http://www.haskellcraft.com/craft3e/Home.html>. 103

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.

Wilf, Herbert S (1994). *Generatingfunctionology*. Academic Press. ISBN 0127519564. [99](#)

[ToC](#)

Python Code Index

Index for some (but not all) of the python code. Note that the index commands are placed after any Beamer frame containing the code to be indexed.

ABTree, [52](#)

`balFactorABT`, [53](#)

`bBST`, [43](#)

`bftbyL`, [31](#)

`bftTraverse`, [28](#)

`bftTraverseByLevels`, [31](#)

`breadthBT01`, [27](#)

`breadthBT02`, [28](#)

`breadthBT03`, [29](#)

`breadthBT04`, [29](#)

`breadthBT04A`, [29](#)

`buildBST`, [43](#)

`convertBTtoABT`, [53](#)

`delBSTTreeJoin1`, [45](#)

`delBSTTreeJoin1ans`, [45](#)

`delBSTTreeJoin1test`, [45](#)

`delBSTTreeJoin2`, [46](#)

`delBSTTreeJoin2ans`, [46](#)

`delBSTTreeJoin2test`, [46](#)

`deleteAVLS`, [91](#)

`deleteAVLT`, [69](#)

`deleteBST`, [44](#)

`differenceAVLS`, [92](#)

`disjointAVLS`, [91](#)

`egBSTTree`, [20](#)

`egBSTTree1`, [21](#)

`egBSTTree2`, [21](#)

`egBSTTree3`, [21](#)

`egBSTTreeL`, [21](#), [44](#)

`egBSTTreeLL`, [22](#)

`egBSTTreeLR`, [49](#)

EmptyABT, [52](#)

EmptyBT, [18](#)

`exposeABT`, [87](#)

`flattenLevels`, [27](#)

`flattenLevelsA`, [28](#)

`genABTs`, [98](#)

`genMirrorShape`, [96](#)

`getDataABT`, [53](#)

`getDataBT`, [19](#)

`getHeightABT`, [53](#)

`getLeftABT`, [53](#)

`getLeftBT`, [19](#)

`getRightABT`, [53](#)

`getRightBT`, [19](#)

`hasAVLpropABT`, [53](#)

`heightBT`, [20](#)

`inBSTree`, [39](#)

`inOrderABT`, [53](#)

`inOrderBT`, [23](#), [32](#)

`inOrderBT0`, [32](#)

`inOrderIterBT1`, [32](#)

`inOrderIterBT2`, [32](#)

`insertAVLS`, [91](#)

`insertAVLT`, [68](#)

`insertBST`, [37](#)

`insertListAVLS`, [91](#)

`insertListAVLT`, [68](#)

`insertListBST`, [40](#)

`intersectAVLS`, [91](#)

`isAVLABTree`, [53](#)

`isBSABTree`, [53](#)

`isBSTree`, [39](#)

`isEmptyABT`, [52](#), [87](#)

`isEmptyBT`, [19](#)

`isIsomorphic`, [96](#)

`isMirrorShape`, [95](#)

`isSameShape`, [95](#)

`isSymmetric`, [96](#)

`join2AVLS`, [90](#)

`joinAVLS`, [87](#)

`joinAVLT`, [69](#)

`joinBST`, [48](#)

`joinBST1`, [45](#)

`joinBST2`, [46](#)

`joinLeftAVLS`, [89](#)

`joinRightAVLS`, [87](#)

`labelsAtDepth`, [31](#)

`lbfBT`, [29](#)

`levelOrderBT`, [26](#)

`levelOrderBT01`, [26](#)

`longZipMerge`, [26](#)

`longZipMerge01`, [26](#)

makeAVLTree, [65](#)
minItemBST1, [48](#)
mkEmptyABT, [52](#), [87](#)
mkEmptyBT, [19](#)
mkNodeABT, [52](#), [87](#)
mkNodeBT, [19](#)
[NodeABT](#), [52](#)
[NodeBT](#), [18](#)
orderedList, [39](#)
postOrderBT, [23](#), [34](#)
postOrderBT0, [34](#)
postOrderIterBT1, [34](#)
preOrderBT, [23](#), [33](#)
preOrderBT0, [33](#)
preOrderIterBT1, [33](#)
preOrderIterBT2, [33](#)
rotl, [58](#)
rotr, [56](#)
setEquality, [93](#)
setFoldr, [92](#)
setFromListAVLS, [91](#)
setToAscList, [92](#)
setToAscListA, [92](#)
sizeBT, [20](#)
splitAVLS, [90](#)
splitAVLT, [69](#)
splitBST, [49](#)
splitBST1, [48](#)
splitFirstAVLS, [90](#)
splitLastAVLS, [90](#)
splitsInt, [98](#)
subsetAVLS, [93](#)
[Tree](#), [18](#)
unionAVLS, [91](#)

Diagrams Index

Index for some of the PGF/TikZ diagrams. Note that the indexing commands are placed after the diagram code

actBSTreeJoin, [48](#)
actBSTreeJoin1, [45](#)
actBSTreeJoin2, [46](#)

egBSTree, [12](#), [14](#)
egBSTree04, [54](#)
egBSTree04a, [54](#)
egBSTree04b, [55](#)
egBSTree04bLL, [55](#)
egBSTree04bLLb, [56](#)
egBSTree04c, [57](#)
egBSTree04cRR, [57](#)
egBSTree04cRRb, [58](#)
egBSTree05, [59](#)
egBSTree05b, [60](#)
egBSTree05bLRotl, [61](#)
egBSTree05bLRotlRotr, [61](#)
egBSTree05bRotr, [60](#)
egBSTree1, [13](#), [14](#)
egBSTree2, [13](#), [15](#)
egBSTree3, [13](#), [15](#)
egBSTreeL, [25](#), [44](#)
egBSTreeLL, [23](#)
egBSTreeLR, [49](#)
exsAVLDeListQ1a01, [76](#)
exsAVLDeListQ1a01A, [77](#)
exsAVLDeListQ1a02, [77](#)
exsAVLDeListQ1a02b, [77](#)
exsAVLInsDelQ1a, [70](#)
exsAVLInsDelQ1a01, [70](#)
exsAVLInsDelQ1a02, [70](#)
exsAVLInsDelQ1a03, [71](#)
exsAVLInsDelQ1a04, [71](#)
exsAVLInsDelQ1a05, [71](#)
exsAVLInsDelQ1a06, [72](#)
exsAVLInsDelQ1a07, [72](#)
exsAVLInsDelQ1a08, [72](#)
exsAVLInsDelQ1a09, [73](#)
exsAVLInsDelQ1a10, [73](#)
exsAVLInsDelQ1aA, [76](#)
exsAVLInsDelQ1b, [74](#)
exsAVLInsDelQ1c, [74](#)
exsAVLInsDelQ2a, [74](#)
exsAVLInsDelQ2b, [75](#)

exsAVLInsDelQ2c, [75](#)
exsAVLTdeIEG, [78](#)
exsAVLTdeIEGa, [78](#)
exsAVLTdeIEGaA, [80](#)

insBSTreeA, [37](#)
insBSTreeAa, [38](#)
insBSTreeB, [37](#)
insBSTreeBa, [38](#)
insBSTreeC, [38](#)
insBSTreeCa, [39](#)

joinRightAVLSa, [87](#)
joinRightAVLSb, [88](#)
joinRightAVLSc, [88](#)
joinRightAVLSd, [88](#)
joinRightAVLSe, [89](#)

tkeys1, [40](#)
tkeys2, [41](#)
tkeys3, [42](#)
treeAfterL, [58](#)
treeAfterR, [56](#)
treeAfterRotl, [66](#)
treeAfterRotr, [66](#)
treeBeforeL, [58](#)
treeBeforeR, [56](#)
treeBeforeRotl, [66](#)
treeBeforeRotr, [66](#)
treeCaseDiag, [59](#)
treeCaseLR, [62](#)
treeCaseLRLRotl, [62](#)
treeCaseLRLRotlRotr, [63](#)
treeCaseRL, [63](#)
treeCaseRLRRotr, [64](#)
treeCaseRLRRotrRotl, [64](#)
treeEGaInnerRotr, [80](#)
treeEGaOuterRotl, [81](#)
treeEGaRotlOnly, [82](#)
treeRLheavy, [79](#)
treeRLheavyA, [80](#)
treeRLheavyInnerRotr, [80](#)
treeRLheavyOuterRotl, [81](#)
treeRLheavyRotlOnly, [81](#)
treeRRheavy, [79](#)

