Murdoch
UNIVERSITY

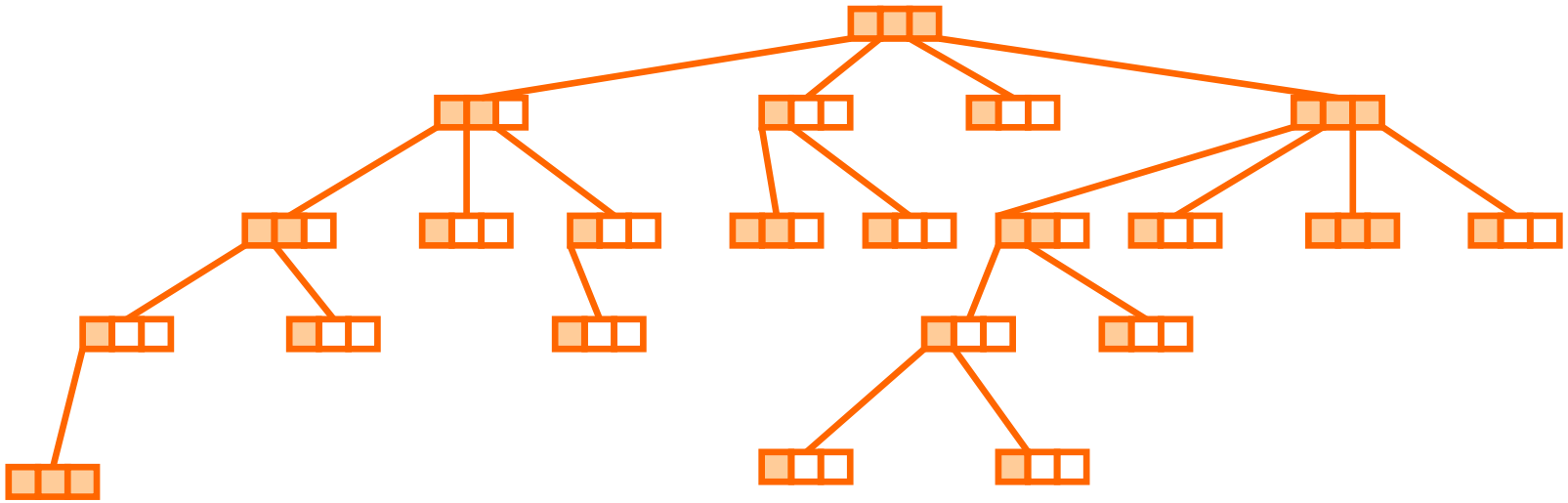
Data Structures and Abstractions

# Trees and Tree Searching & Binary Tree & Other Tree Types

Lecture 9

# Trees

- Trees are ADS where every Node has directional links to one or more nodes underneath it. [1]
- An m-tree is a node with 0:*m* links and 1:*m*-1 pieces of data in each node. For example a 4-tree (quadtree or 4-way tree) might look something like this:



Murdoch UNIVERSITY

# Tree Definitions

- The top node is called the root. [1]
- Any node that is not the root node is a child of some parent.
- Any node that has one or more children is a parent.
- Nodes connected to same parent are called siblings. [2]
- Any node that has no children is called a leaf.
- Any part of the tree smaller than the whole is called a subtree.

# Tree Use [1]

- The back-ends of databases.
- Data stores that are not databases.
- Problem solving.
- Game playing.
- Graphics and virtual reality: for tracking line of sight as well as storing screen objects.
- Graph theory (e.g. path finding).

The algorithm used to insert data into the tree will vary from application to application.

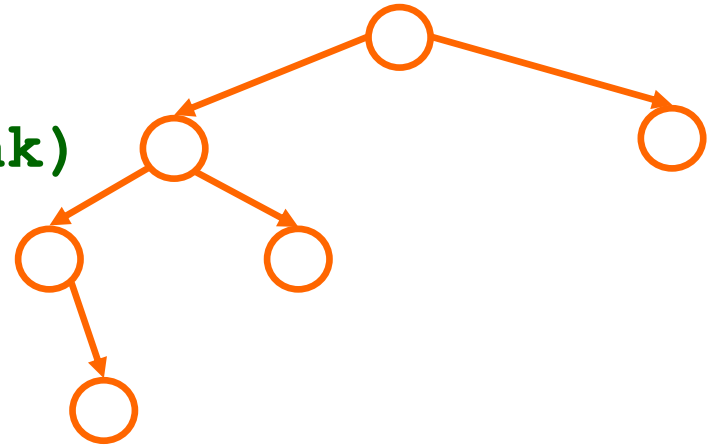
# Traversal

- Traversing a tree involves going to every node.
- This needs to be done for processes such as printing, gathering statistics, end-of-month calculations, searching etc.
- It can be done either in-order, pre-order or post-order.
- The method chosen depends on the application.
- In the examples, we look at a 2-way or *binary* tree, as this is the simplest to understand.

# In-Order Traversal

Examples don't have the terminating condition for recursion – see note [1]

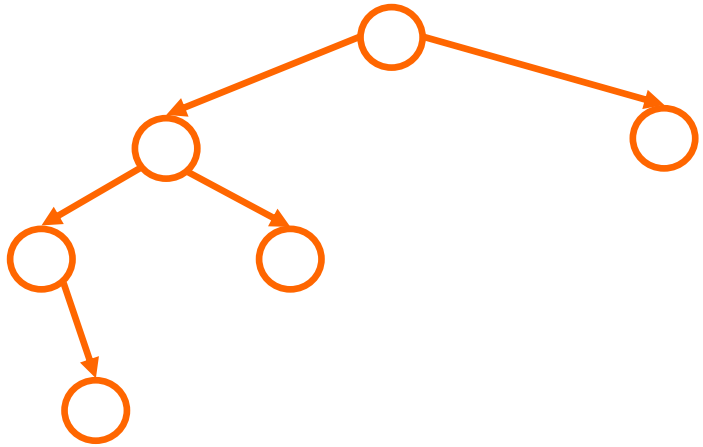
```
ProcessNode (node) [1]
    ProcessNode (leftLink)
    Process this node
    ProcessNode (rightLink)
End ProcessNode
```



Murdoch UNIVERSITY

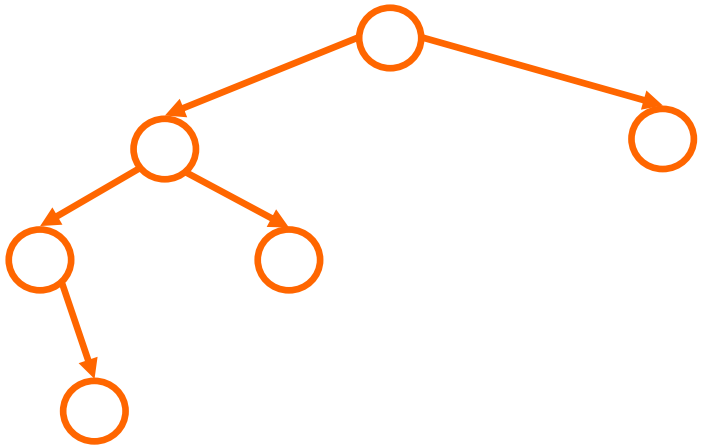
# In-Order Traversal Animation

```
ProcessNode (node)
    ProcessNode (leftLink)
    Process this node
    ProcessNode (rightLink)
End ProcessNode
```



# Pre-Order Traversal

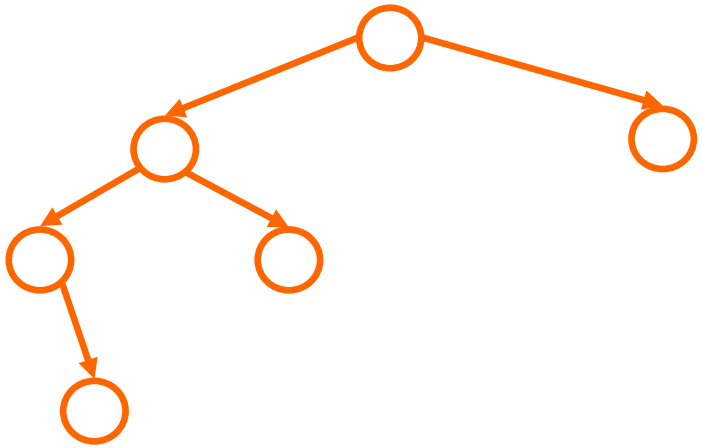
```
ProcessNode (node)
   Process this node
   ProcessNode (leftLink)
   ProcessNode (rightLink)
End ProcessNode
```



Murdoch UNIVERSITY

# Pre-Order Traversal Animation

```
ProcessNode (node)
   Process this node
   ProcessNode (leftLink)
   ProcessNode (rightLink)
End ProcessNode
```

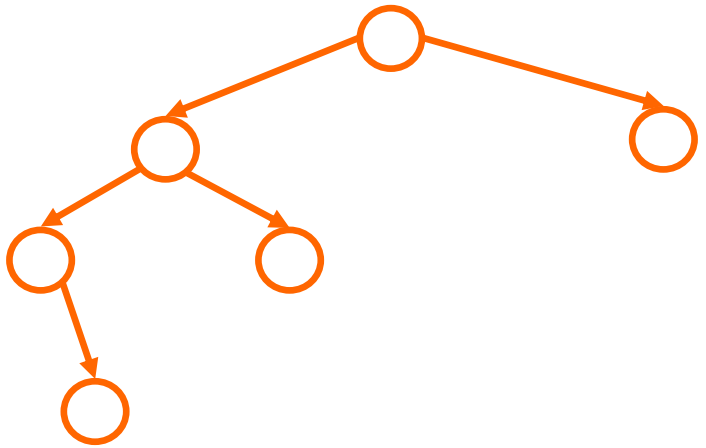


Murdoch UNIVERSITY

END

# Post-Order Traversal

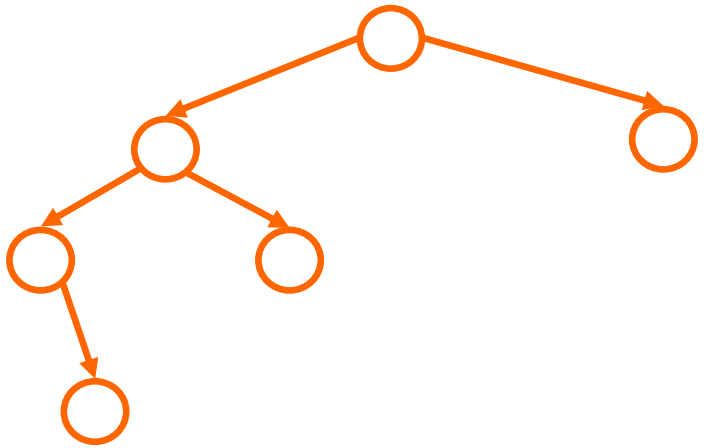
```
ProcessNode (node)
    ProcessNode (leftLink)
    ProcessNode (rightLink)
    Process this node
End ProcessNode
```



Murdoch UNIVERSITY

# Post-Order Traversal Animation

```
ProcessNode (node)
    ProcessNode (leftLink)
    ProcessNode (rightLink)
    Process this node
End ProcessNode
```

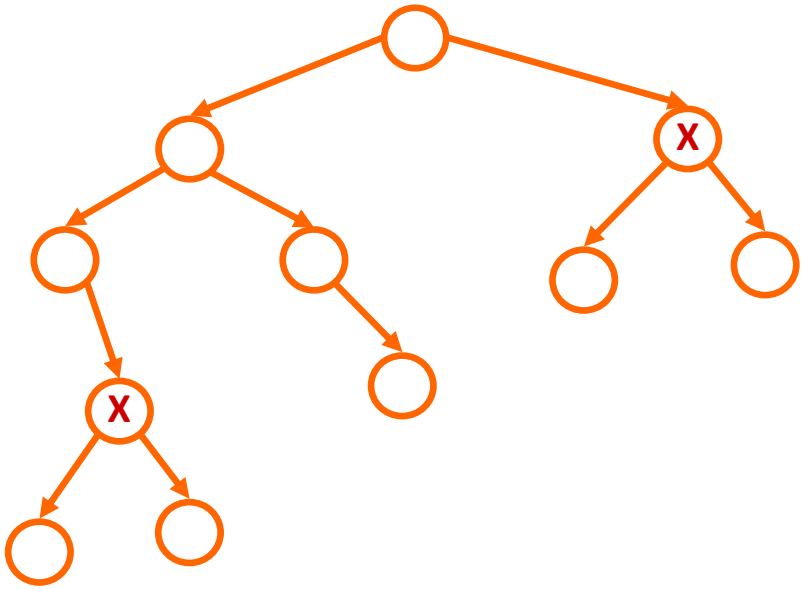


# Tree Searching

- Trees will also need to be searched, and there are many different search algorithms available.
- When not using heuristics, there are two main ways in which to do a logical search:
  - Depth first
    - which is simply a search done in pre-order stopping when the target data is found;
    - the aim is to find ***any*** match to the target;
    - this is commonly used when trying to find the one unique match.
  - Breadth first
    - where the nodes are searched in layers, down from the top;
    - this search aims to find the match that is ***closest*** to the start;
    - a common use for this is in game playing: you want to win as soon as possible.

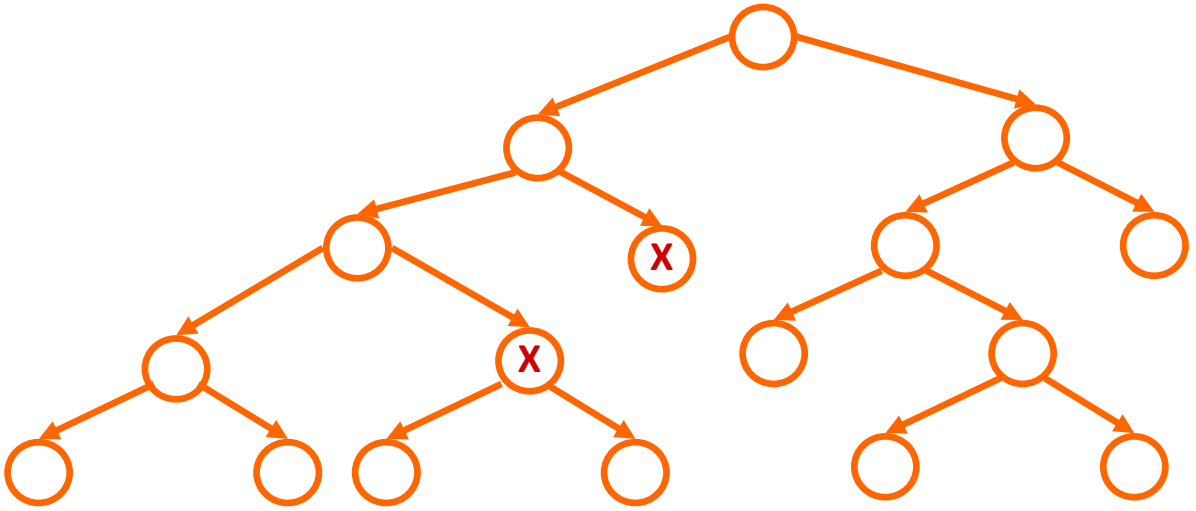
# Depth First Search Animation

```
Search (node) : boolean
    boolean found
    found = target at this node
    IF not found
        found = Search (leftLink)
        IF not found
            found = Search (rightLink)
        ENDIF
    ENDIF
    return found
End Search
```



# Breadth First Animation [1]

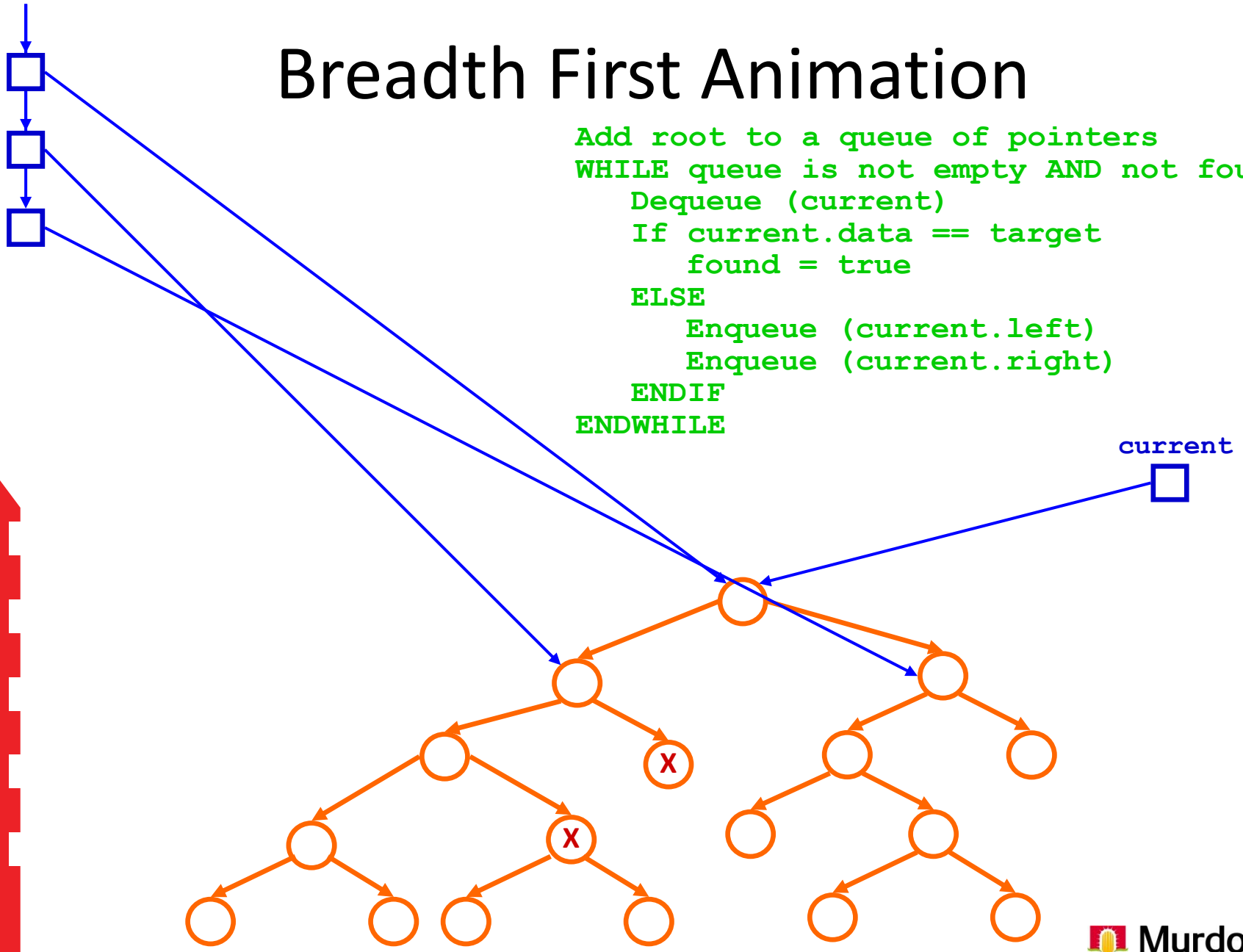
```
Add root to a queue of pointers
WHILE queue is not empty AND not found
   Dequeue (current)
   If current.data == target
      found = true
   ELSE
      Enqueue (current.left)
      Enqueue (current.right)
   ENDIF
ENDWHILE
```




Murdoch UNIVERSITY

# Breadth First Animation

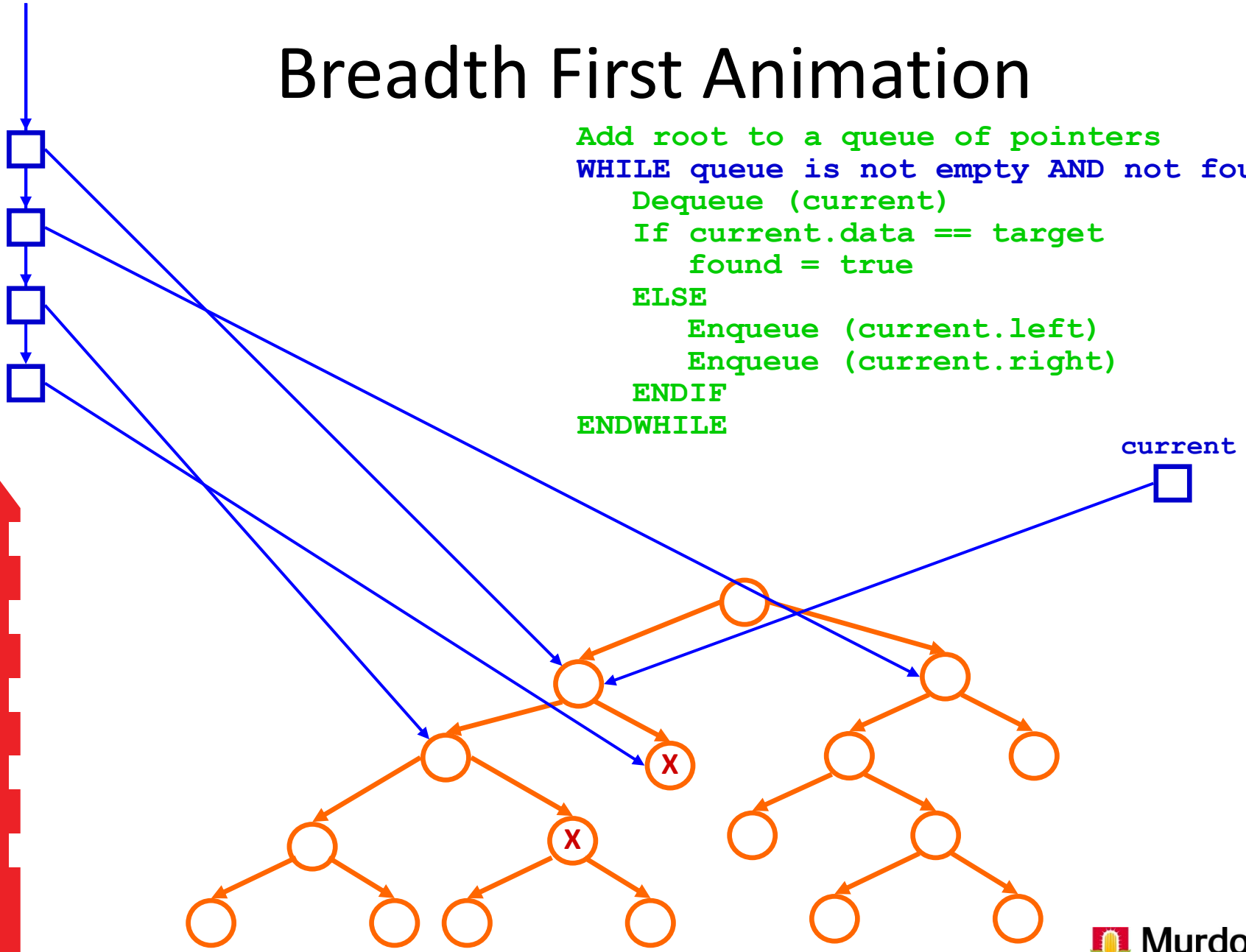
```
Add root to a queue of pointers
WHILE queue is not empty AND not found
   Dequeue (current)
   If current.data == target
      found = true
   ELSE
      Enqueue (current.left)
      Enqueue (current.right)
   ENDIF
ENDWHILE
```




Murdoch UNIVERSITY

# Breadth First Animation

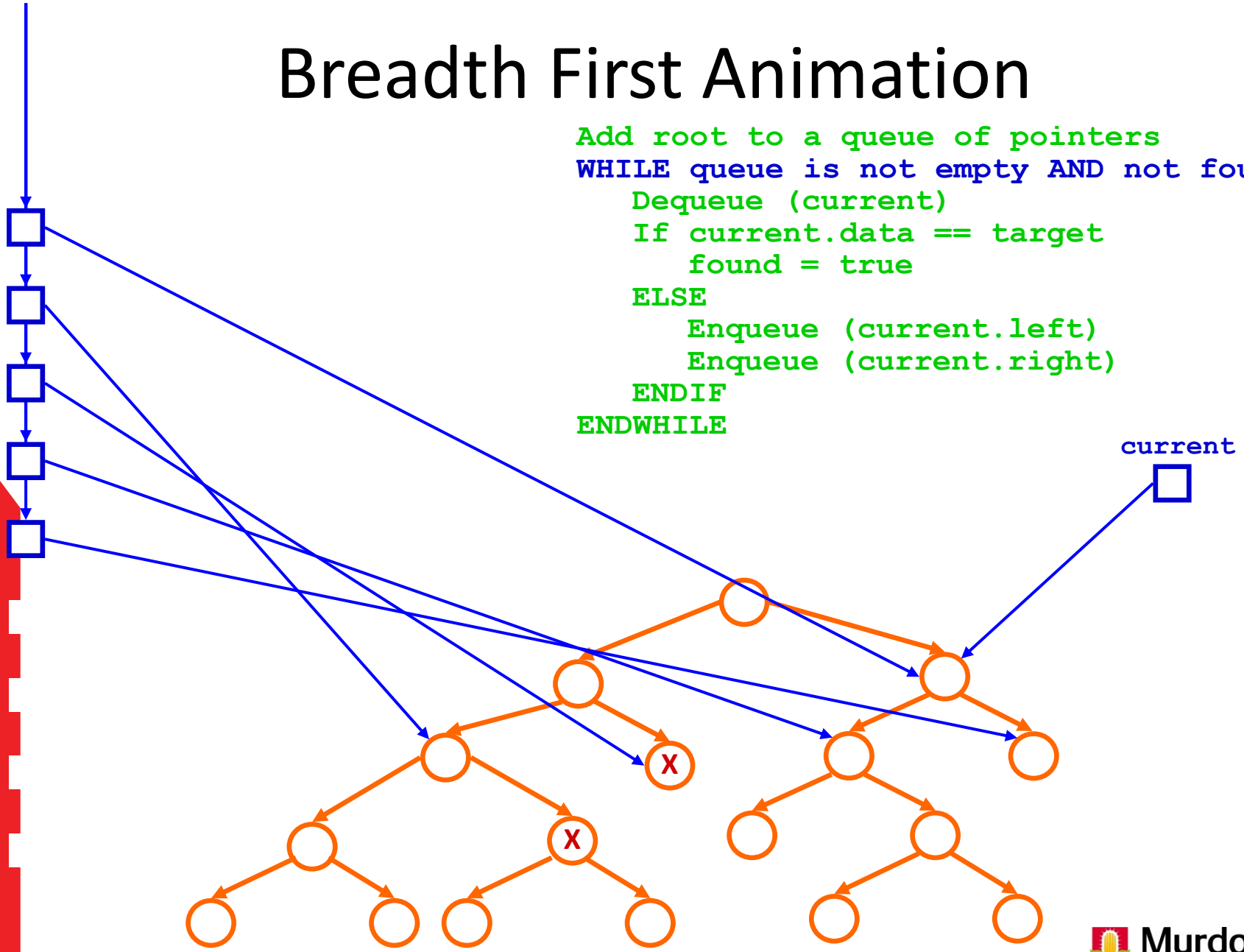
```
Add root to a queue of pointers
WHILE queue is not empty AND not found
   Dequeue (current)
   If current.data == target
      found = true
   ELSE
      Enqueue (current.left)
      Enqueue (current.right)
   ENDIF
ENDWHILE
```




Murdoch UNIVERSITY

# Breadth First Animation

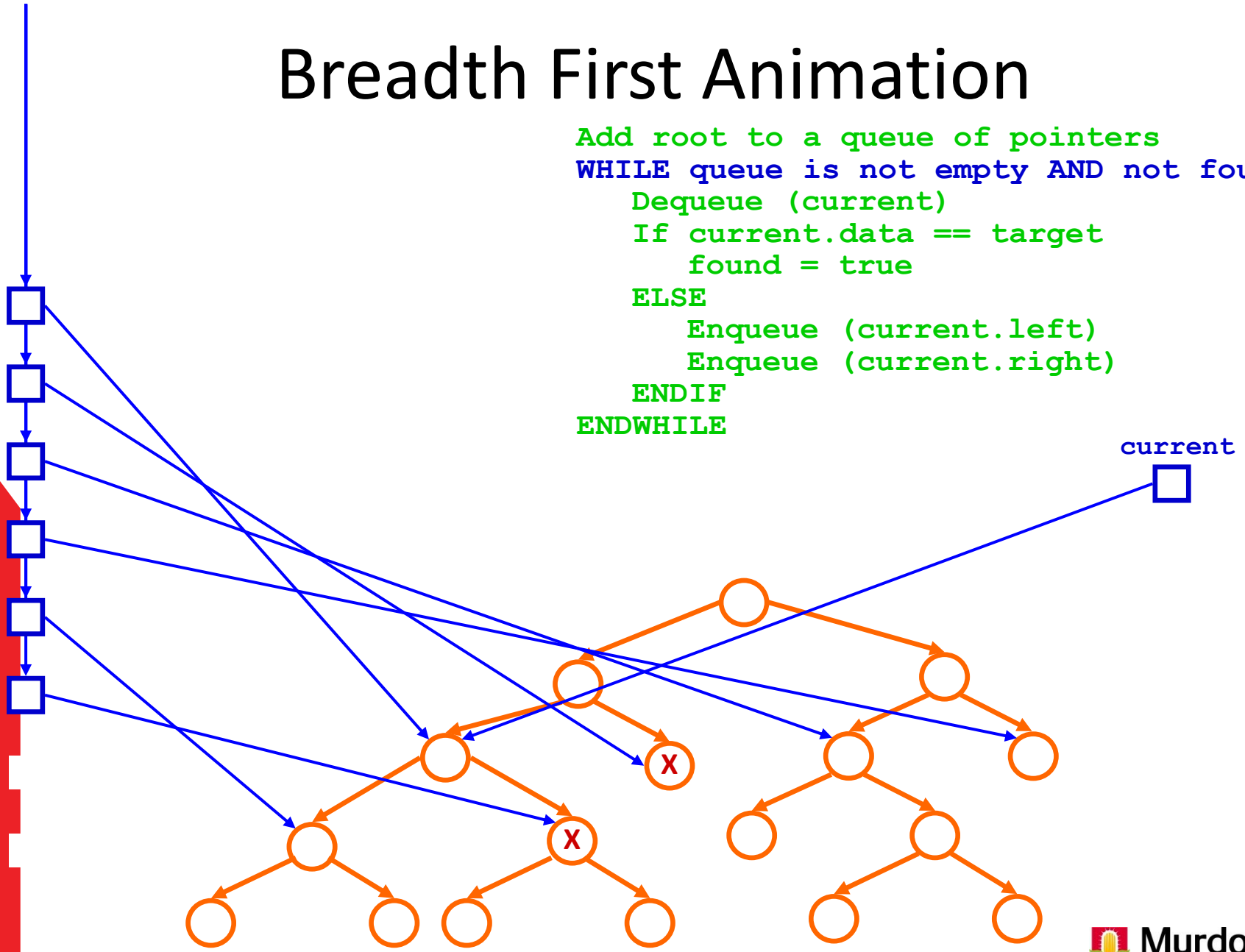
```
Add root to a queue of pointers
WHILE queue is not empty AND not found
   Dequeue (current)
   If current.data == target
      found = true
   ELSE
      Enqueue (current.left)
      Enqueue (current.right)
   ENDIF
ENDWHILE
```




Murdoch UNIVERSITY

# Breadth First Animation

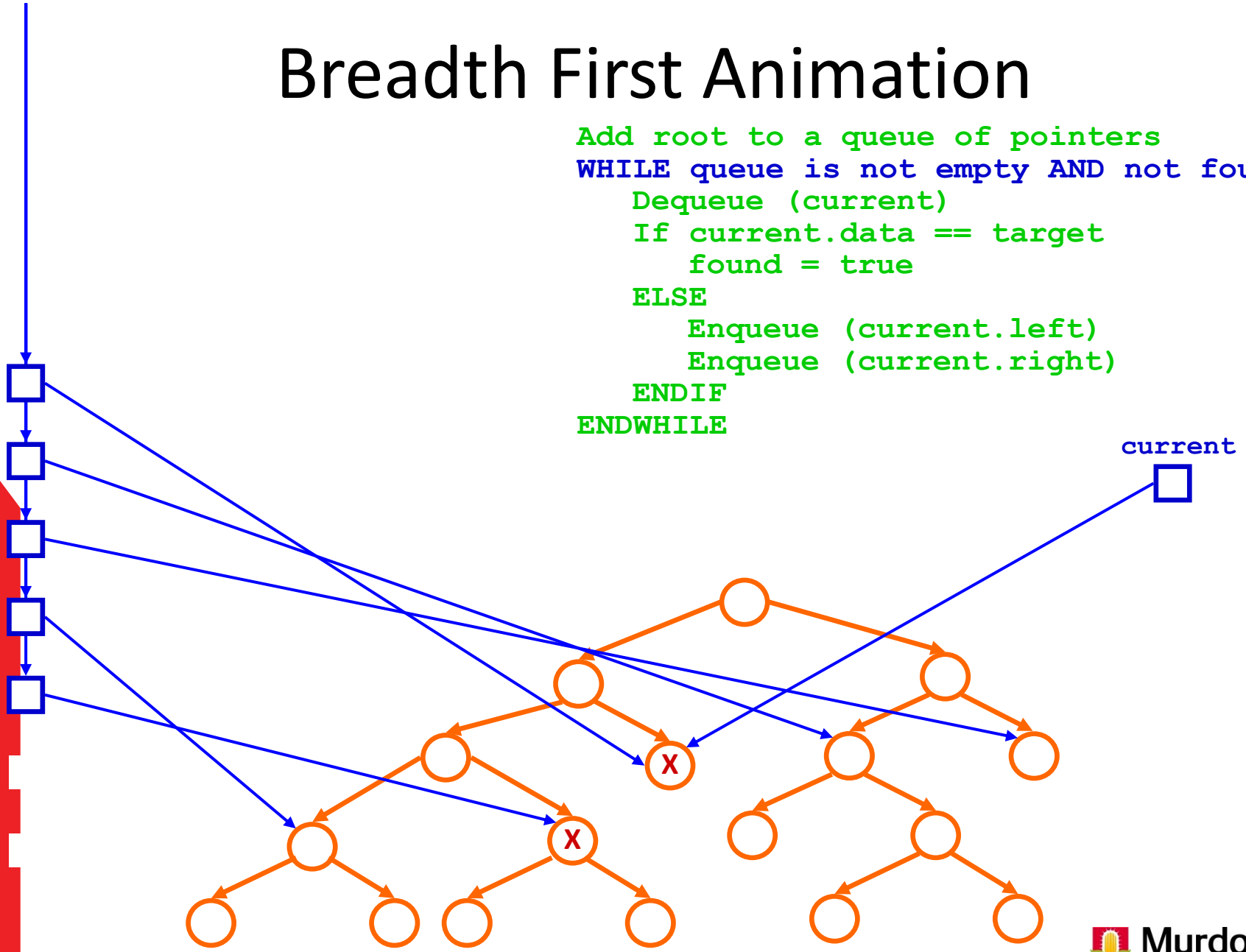
```
Add root to a queue of pointers
WHILE queue is not empty AND not found
   Dequeue (current)
   If current.data == target
      found = true
   ELSE
      Enqueue (current.left)
      Enqueue (current.right)
   ENDIF
ENDWHILE
```




Murdoch UNIVERSITY

# Breadth First Animation

```
Add root to a queue of pointers
WHILE queue is not empty AND not found
   Dequeue (current)
   If current.data == target
      found = true
   ELSE
      Enqueue (current.left)
      Enqueue (current.right)
   ENDIF
ENDWHILE
```




END

# Readings

- Textbook: Chapter on Binary Trees
  - Should go through the programming example at the end of the chapter.
- Textbook: Chapter on Recursion
  - Revise the concept covered in earlier units and be able to implement recursive routines.
  - Recursion vs Iteration
- Further exploration:
  - Reference book, Introduction to Algorithms. For further study, there are many tree and tree algorithms described in the reference book. For this unit, the lecture notes, practical work and the textbook is sufficient.

# Binary Search Trees

# Introduction to ADS Sorted Data Stores

- As pointed out in the earlier lecture, trees are used for problem solving, game playing, virtual reality and data storage, amongst other things.
- When used for data storage they are always built so that the data is sorted as it is inserted.
- We will be looking at several different sorted trees including Binary Search Trees, AVL Trees, Multiway Trees, B-Trees and B+ trees. [1]
- In later lectures we will also consider non-sorted trees used to store information during graph processing.

Murdoch UNIVERSITY

# The Data to be Stored

- The data stored in the tree can either be the actual data or a pointer/index to the actual data.
- The actual **data** stored will almost always contain a ***key*** plus other data.
- The key is used to place (order) the data in the container.
- Examples of keys are account numbers, membership numbers, names, or keys calculated from some part of the data.
- The key should be unique to enable the BST to be more efficient.
- It also possible to have secondary keys, where a list, array or tree is 'overlayed' on the first structure giving a different sorted order.

Murdoch UNIVERSITY

# Binary Search Trees

- Binary trees are trees where
  - every node has 1 piece of data and two pointers (left, right)
  - every node, except root, can have a parent pointer [1]
  - therefore every node has 0:2 children
- Binary search trees are binary trees where
  - every node has data that is greater than the data in all nodes to the left of it.
  - every node has data that is less than the data in all nodes to the right of it.
- Note that this contrasts with the heap (see later), where a node's data was always guaranteed to be less than (for a min-heap) or greater than (for a max-heap) all data in its subtree. [2]
- Since the data sorting is based on a unique key, there is normally no two identical sets of data. [3]

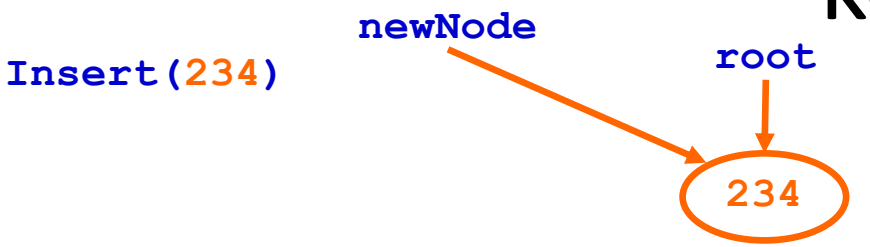
Murdoch UNIVERSITY

# BST Algorithms

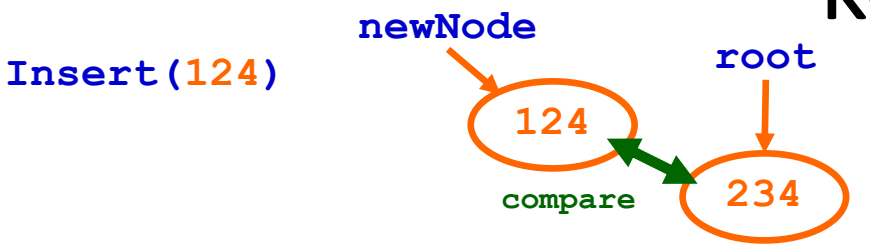
- Almost all BST algorithms are recursive as this makes them very simple.
- However, the root node might be treated differently because it has no parent but should it? [1]
- All the methods require that root has been set to NULL in the constructor.
- Traversal of a BST is almost always done either in-order or pre-order. [2]

Murdoch UNIVERSITY

# BST Insert Animation (for integer key)

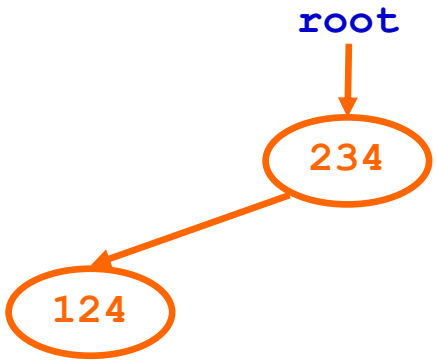


# BST Insert Animation (for integer key)



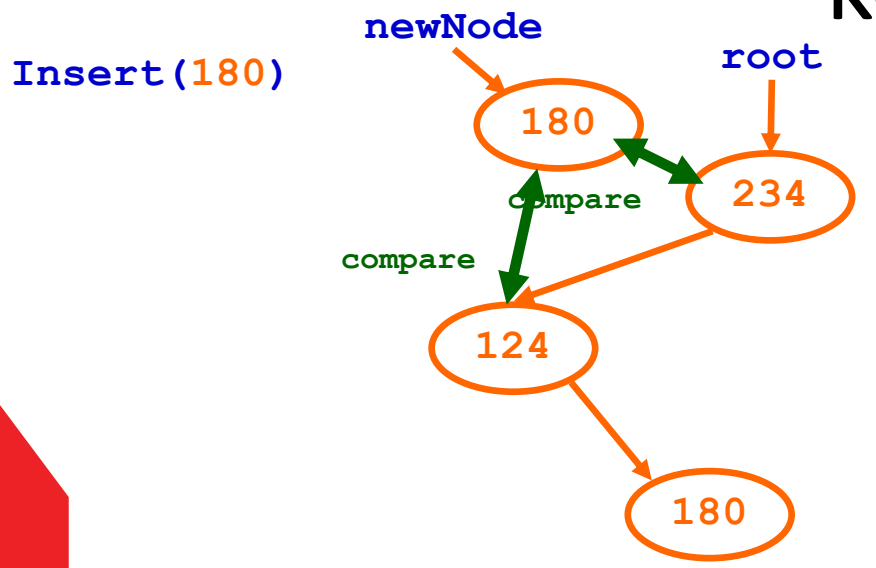
# BST Insert Animation (for integer key)

```
Insert(124)
```



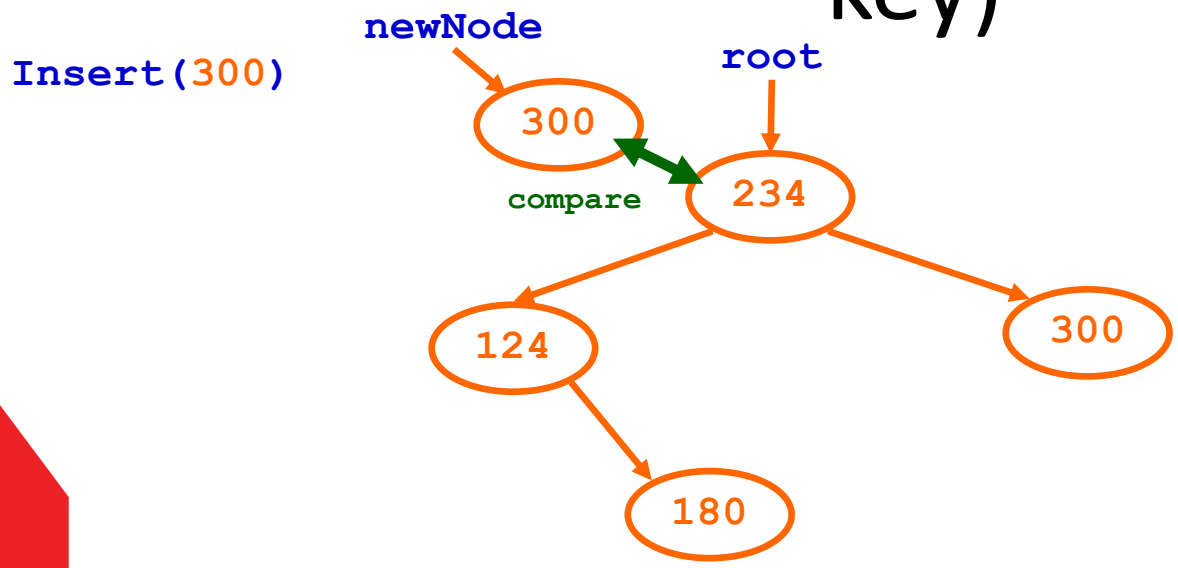

Murdoch UNIVERSITY

# BST Insert Animation (for integer key)

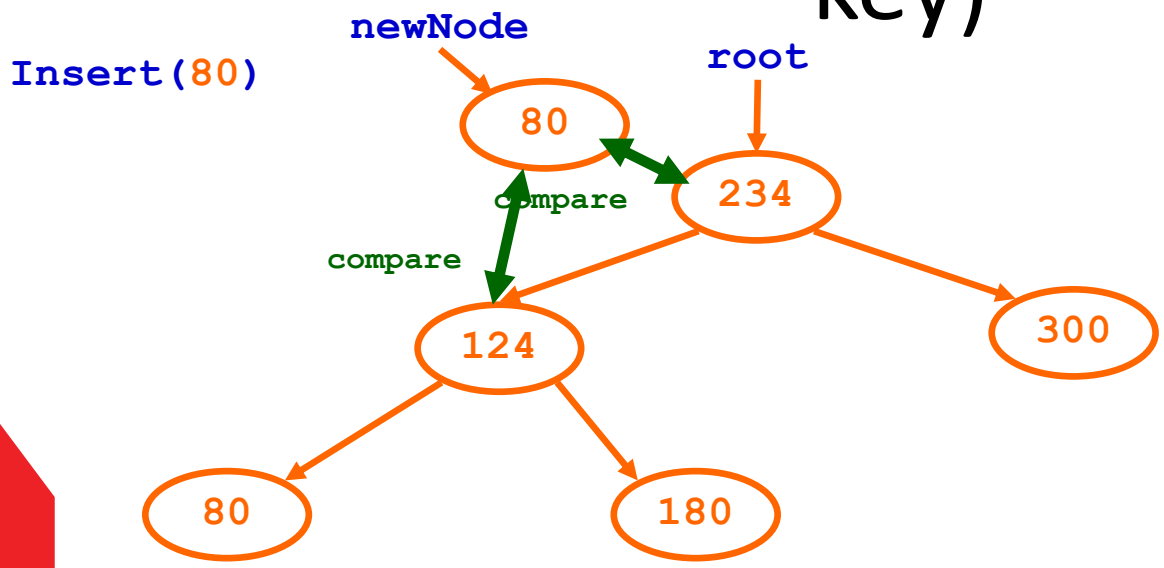



Murdoch UNIVERSITY

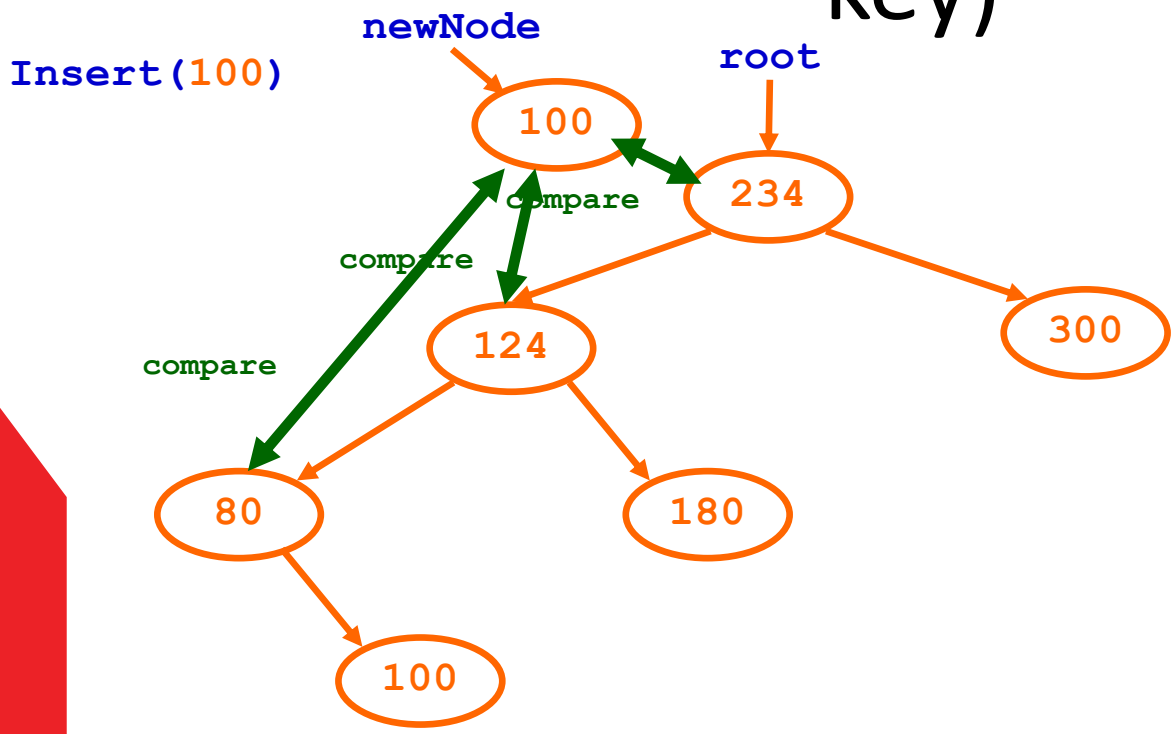
# BST Insert Animation (for integer key)



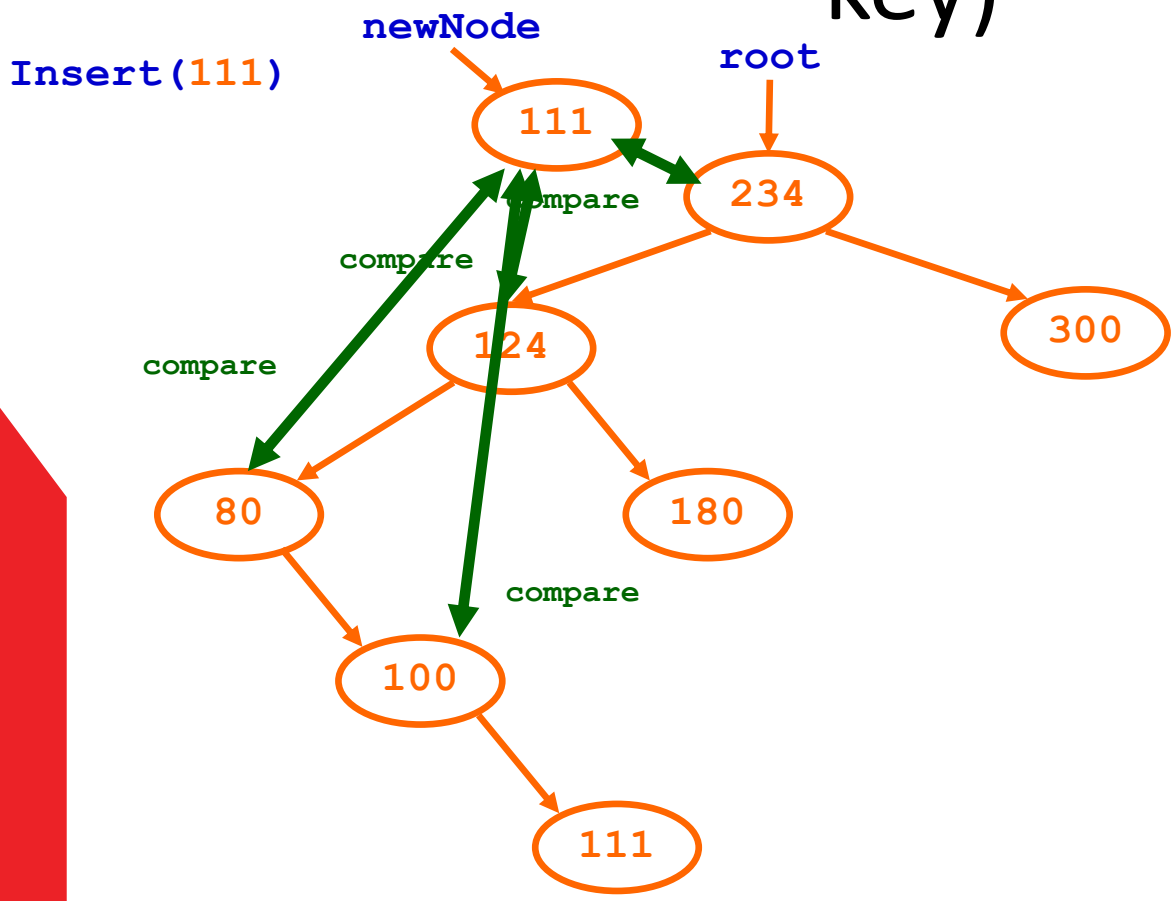
# BST Insert Animation (for integer key)



# BST Insert Animation (for integer key)

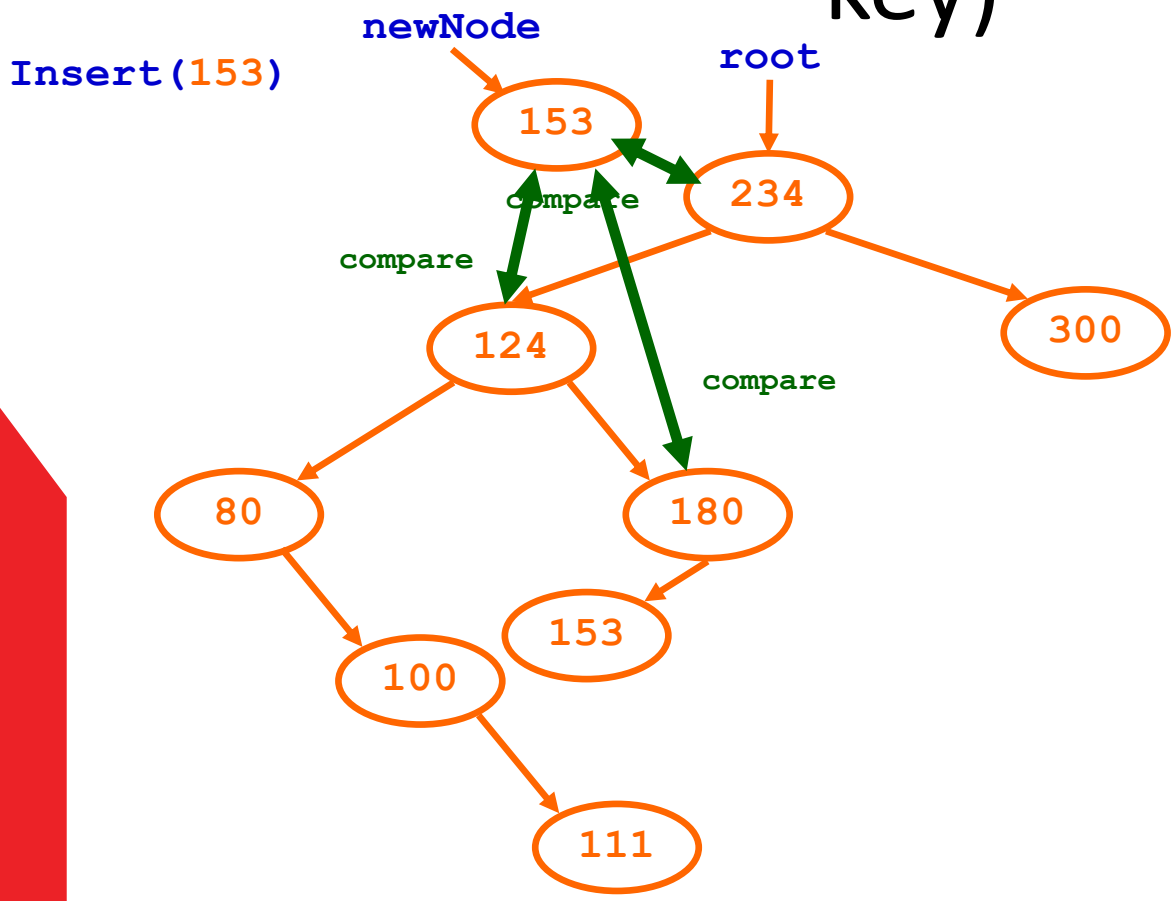


# BST Insert Animation (for integer key)



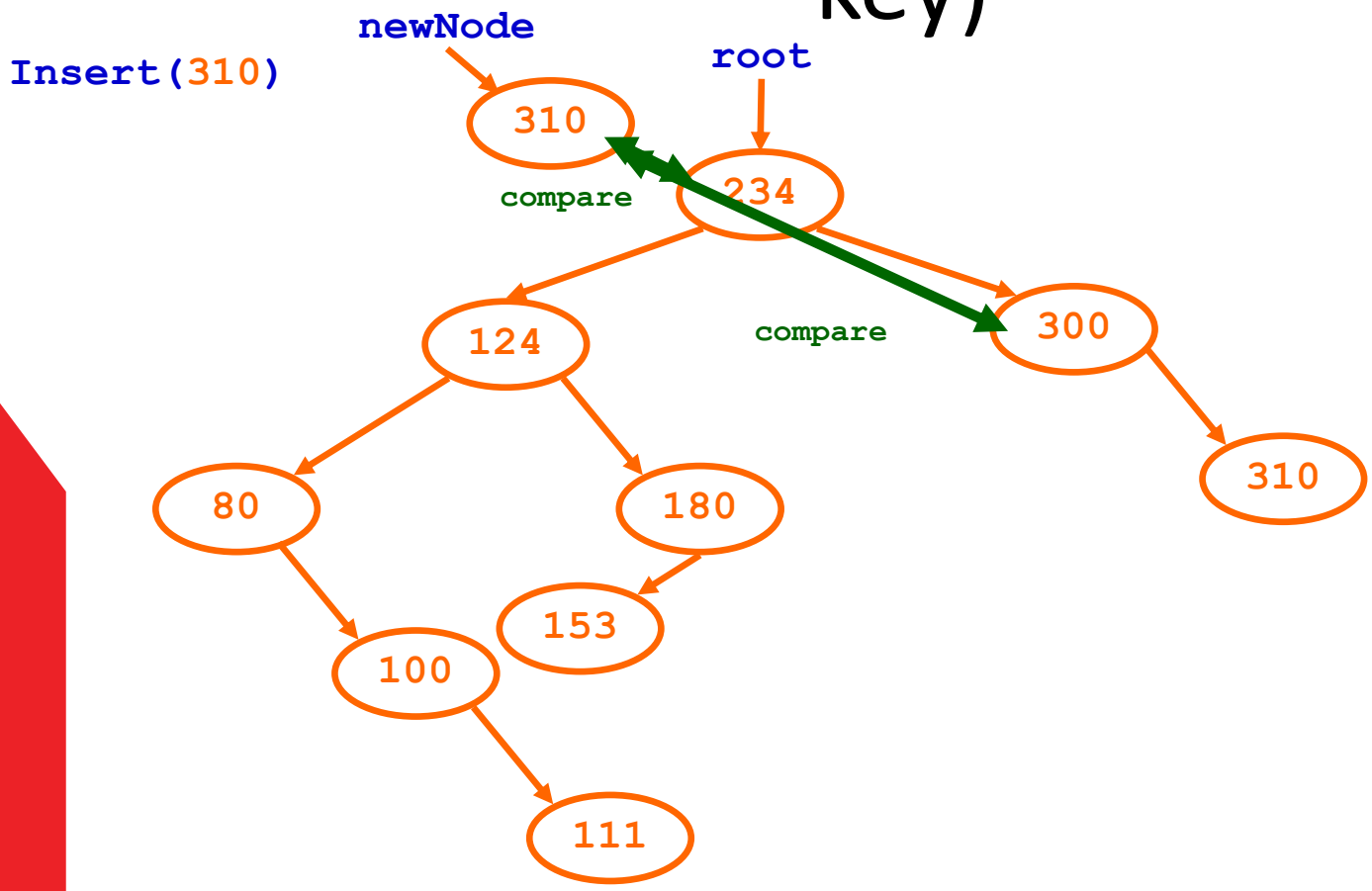

Murdoch UNIVERSITY

# BST Insert Animation (for integer key)




Murdoch UNIVERSITY

# BST Insert Animation (for integer key)




END

# BST Insert

- Insert (newdata) [1]
- Get memory for a newNode
- Place new data in the newNode
- IF root is NULL
- root = newNode [2]
- ELSE
- Insert (newNode, root) [3]
- ENDIF
- END Insert

Murdoch UNIVERSITY

```
Insert (newNode, parent) [1]
    IF newNode's data < parent.data
        IF parent has no left child
            parent.leftLink = newNode
        ELSE
            Insert (newNode, parent.leftLink)
        ENDIF
    ELSE // what happens if newNode data == parent.data?
        IF parent has no right child
            parent.rightLink = newNode
        ELSE
            Insert (newNode, parent.rightLink)
        ENDIF
    ENDIF
END Insert
```

Murdoch UNIVERSITY

# BST Problem

- The trouble with the ordinary BST, is that if ordered data is inserted, you end up with a linked list.
- This means that searching a BST is O(log n) on average at best, but has a worst case complexity of O(n). ( O(h) )
- This problem is solved by using a *balanced* BST instead of the simple BST.
- However, the solution comes at the cost of more difficult algorithms.
- Which in turn means that programming, testing, debugging and maintaining becomes more time consuming.

Murdoch UNIVERSITY

# AVL Trees

- Invented by **A**delson-**V**elski and **L**andis (so AVL) [1]
- It is a height balanced tree. [2] – see separate diagram.
- In other words the height of the left and right subtrees is never allowed to differ by more than 1.
- This ensures that the complexity of a search remains at O(log n).
- The height of a subtree is defined recursively as:

```
IF the tree is empty
   height = -1
ELSE
   height = 1 + max(height(leftLink), height(rightLink)
ENDIF
```

Murdoch UNIVERSITY

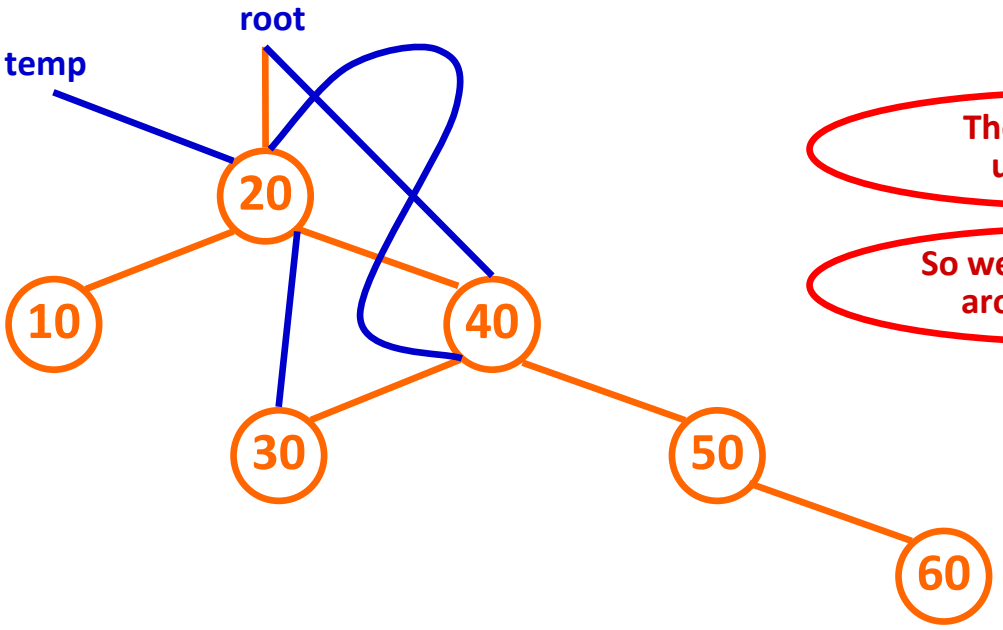
# Insertion into an AVL Tree

- Insertion is done the same as for an ordinary BST.
- But if the height is unbalanced, the insertion is followed with a rebalance:

```
Insert (newNode, parent) [1]
     IF newNode's data < parent.data
          IF parent has no left child
               parent.leftLink = newNode
          ELSE
               Insert (newNode, parent.leftLink)
               RebalanceBelowLeftOf (parent)
          ENDIF
     ELSE
          IF parent has no right child
               parent.rightLink = newNode
          ELSE
               Insert (newNode, parent.rightLink)
               RebalanceBelowRightOf (parent)
          ENDIF
     ENDIF
END Insert
```

Murdoch UNIVERSITY

# AVL Tree Insert Animation

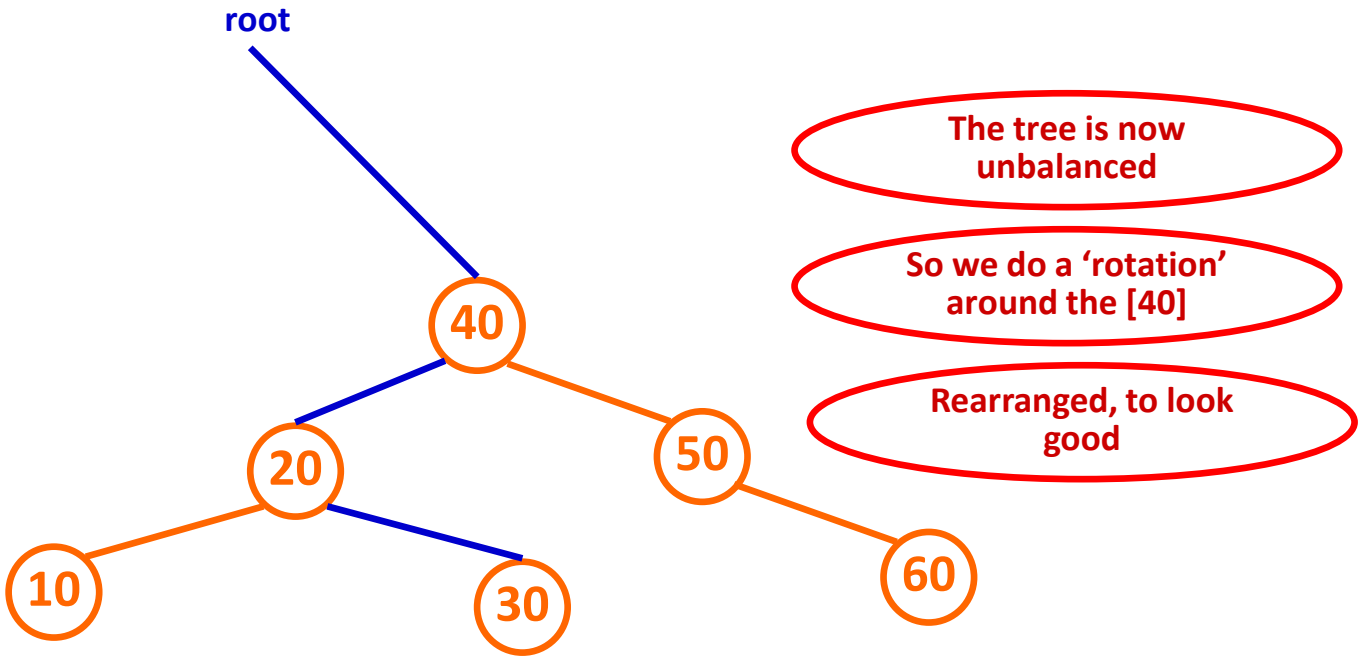



The tree is now unbalanced

So we do a 'rotation' around the [40]

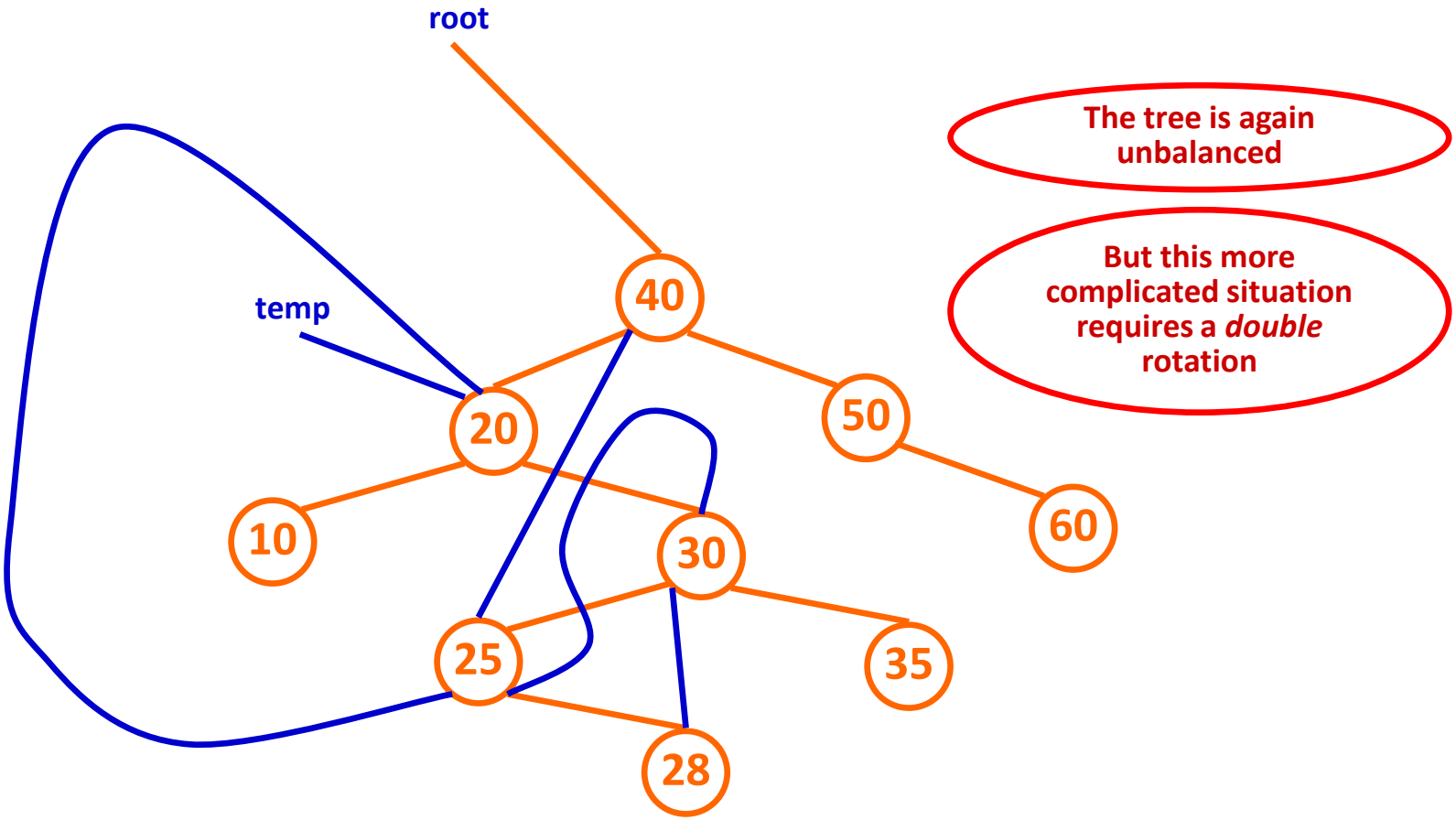
END

# AVL Tree Insert Animation



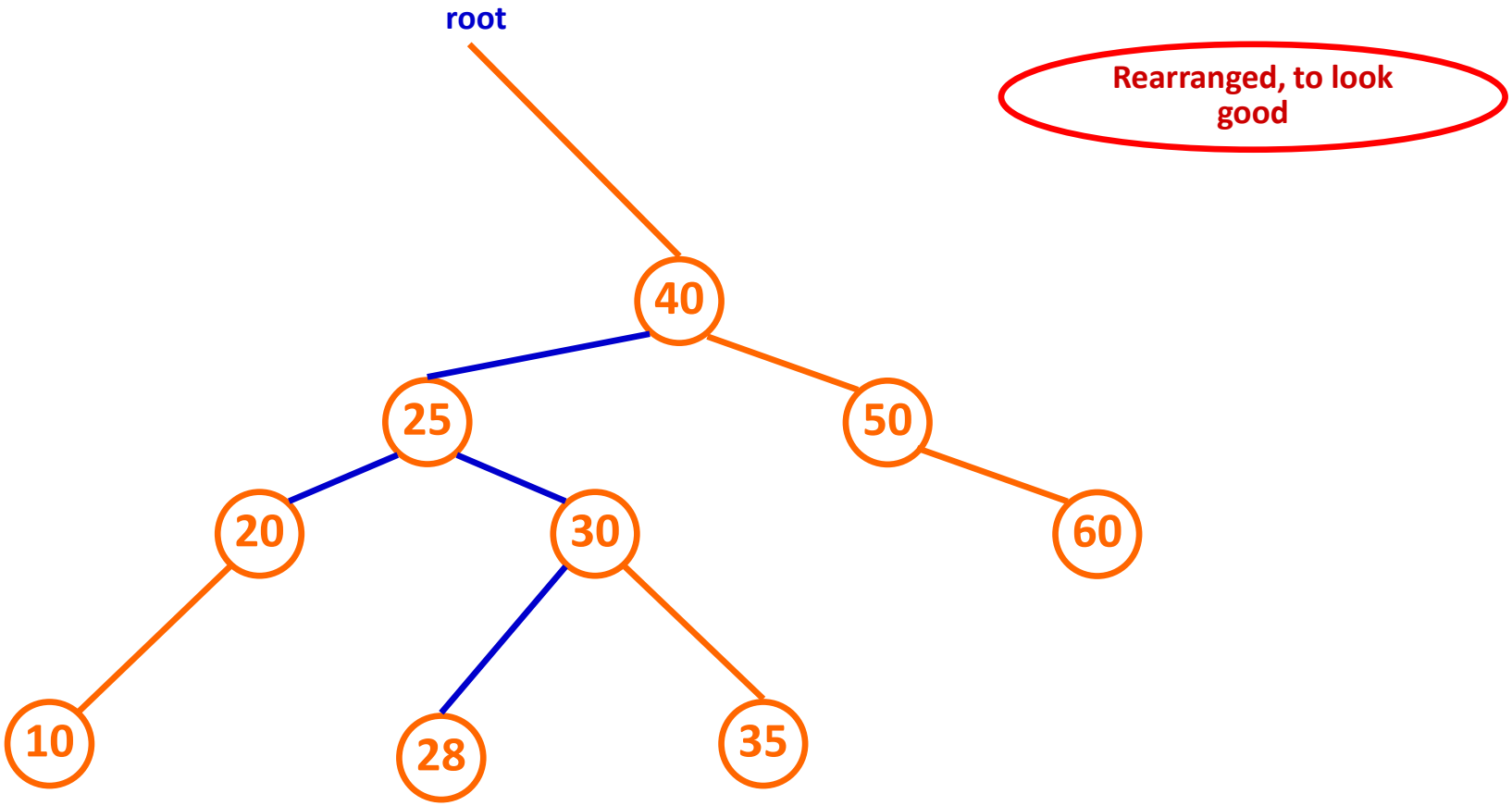

Murdoch UNIVERSITY

# AVL Tree Insert Animation




Murdoch UNIVERSITY

# AVL Tree Insert Animation




Murdoch UNIVERSITY

# Rebalancing and Rotations

- Inserting into an AVL tree can result in the tree becoming unbalanced.
- The part of the tree that is unbalanced is going to be somewhere on the tree from the insertion point to the root of the tree as only these subtrees are affected by the insertion.
- Rebalancing needs to be carried out to maintain the AVL property.
- The rebalancing is done by rotation operations.
- For example a single rotation swaps the role of the parent and child maintaining the search order. For a number of cases, single rotation doesn't work so double rotations are used. You are encouraged to find out how these operations work on your own. It is not examinable this semester. [1]
- What *is* examinable is the ability to draw the tree *after* the rebalancing, so that is what you need to be able to do.
- You are also encouraged to find out more about Red-Black trees. These are good alternatives to AVL trees.

Murdoch UNIVERSITY

# The Programs [1]

- **`BTreeSolver`** shows the resulting BST, AVL Tree, Max Heap and/or Min Heap after insertions (which can be randomly generated or chosen).
- **`HeapSort`** shows the steps involved in a heap sort.
- **`MTreeSolver`** shows the resulting Multiway tree, BTree and/or B Plus Tree after insertions. These are covered in the next lecture.
- **`Graphs`** allows you to build graphs and then view information about the graphs (future lectures).

Murdoch UNIVERSITY

# Readings

- Textbook: Chapter on Binary Trees, particularly the section on Binary Search Trees.
  - Should go through the programming example at the end of the chapter.

  Textbook: Chapter on Recursion

# Further exploration

- In the lab/assignment, you would normally be asked to provide a rational for your data structures. In this video  link below  (from an MIT unit on Introduction to Algorithms) for BST justification one particular example is used.
- MIT Lecture:
  - https://www.youtube.com/watch?v=9Jry5-82I68&index=5&list=PLUl4u3cNGP61Oq3tWYp6V_F-5jb5L2iHb. In the video, the tree algorithm is modified to cater for a new requirement. This approach shouldn't be used– see Open Closed Principle. Think of a better solution. Other than that, the video explains the BST and its use very well.

Murdoch UNIVERSITY

# Further exploration

- Reference book, Introduction to Algorithms. For further study, there are many tree and tree algorithms described in the reference book. For this unit, the lecture notes, practical work and the textbook is sufficient.
- Optional – recurrence trees https://www.youtube.com/watch?v=8F2OvQIlGiU
- An earlier textbook used in this unit (some years ago) is a better reference to some of the more interesting Tree (and graph) data structures like AVL trees, Red Black trees and AA trees. The book is available in the library. It is "Algorithms, Data Structures, and Problem solving using C++" by Mark Weiss.

- AVL trees .. from an MIT unit Introduction to Algorithms
- MIT Lecture:
  - https://www.youtube.com/watch?v=FNeL18KsWPc&index=6&list=PLUl4u3cNGP61Oq3tWYp6V_F-5jb5L2iHb
  - MIT Tutorial (different to a lab, no computers) https://www.youtube.com/watch?v=IWzYoXKaRIc&list=PLUl4u3cNGP61Oq3tWYp6V_F-5jb5L2iHb&index=29
  - https://www.youtube.com/watch?v=r5pXu1PAUkI&list=PLUl4u3cNGP61Oq3tWYp6V_F-5jb5L2iHb&index=28

Murdoch UNIVERSITY

# Multiway Trees

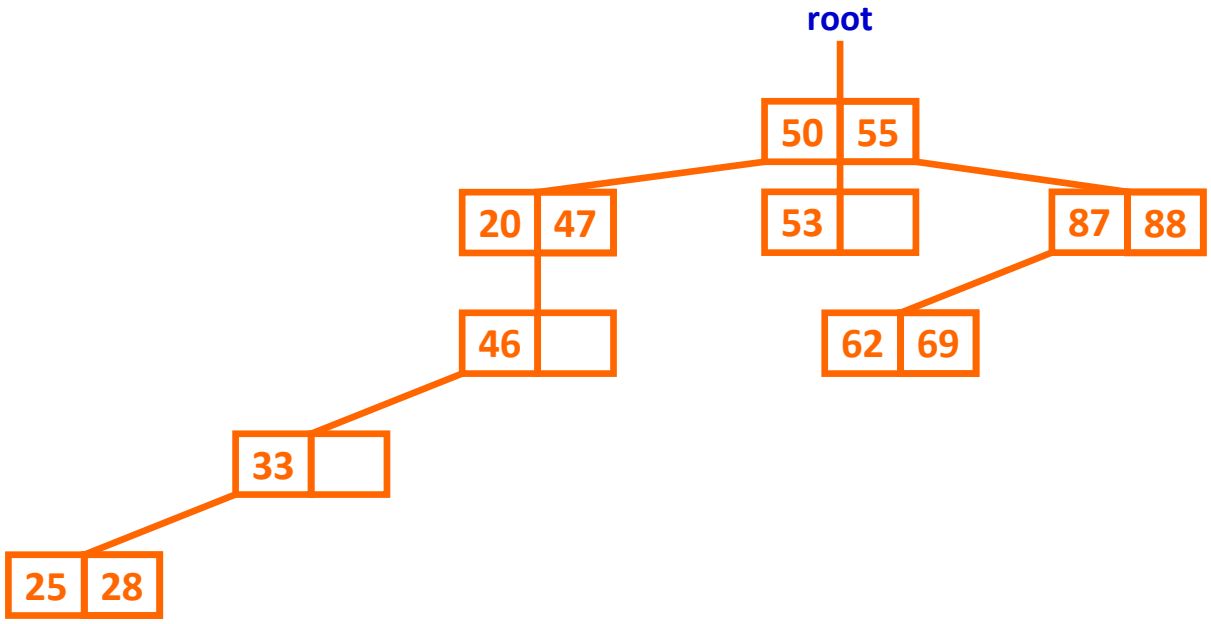
# Multiway Trees

- Multiway trees are trees that store more than one piece of data in a node and more than two links.
- A 3-way tree stores up to 2 items of data per node.
  A 4-way tree stores up to 3 items of data per node,
  etc.
- Insertion is done in the same way as with a simple BST.
- Of course this means that, like a simple BST, it is possible to end up with a linked list.
- Reminder: in the animations we just show storage of a single integer, but in reality trees are used to store larger amounts of information using a key. [1]

Murdoch UNIVERSITY

Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)
Insert(25)
Insert(87)
Insert(88)
Insert(53)
Insert(62)
Insert(69)
Insert(28)

# 3-way Tree Insert Animation




END

# B Trees

- B Trees are balanced multi-way trees in which a node can have up to k subtrees.
- Suitable for data storage on disks when collections are too large for internal memory.
- As for most data stores, the elements are usually records, which have a key and a value.
- The key is used to locate the node where the record is to be stored.

Murdoch UNIVERSITY

# B Tree Definition

- Formally, a B Tree of order m is a multi-way tree in which:
  - the root is either a leaf or has at least two subtrees;
  - each leaf node holds at least m/2 keys;
  - each non-leaf node holds k-1 keys and k pointers to subtrees where m/2 <= k <= m;
  - all leaves are on the same level.
- m is normally large (50-500) so that all the information stored in one block on disk can fit into one node.

Murdoch UNIVERSITY

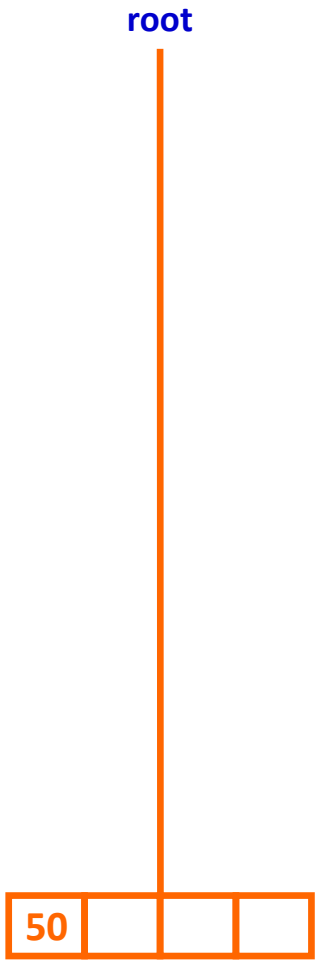
# Insertion into a B Tree

- Insertion of a key (and its record) is always done at a leaf node.
- This may cause changes higher up the tree.
- The method is:
  1. Locate: Do a search to locate the leaf in which the new record should be inserted.
  2. Insert:
     a) If the leaf has room, insert the record, in order of key.
     b) If the node if full, 'split' it and move the record with the median key upwards.
     c) Repeat (b) until either a non-full node is found, or root is reached.
     d) If the root is full, split it and create a new root node containing one key.

Murdoch UNIVERSITY

Insert(50)
Insert(20)

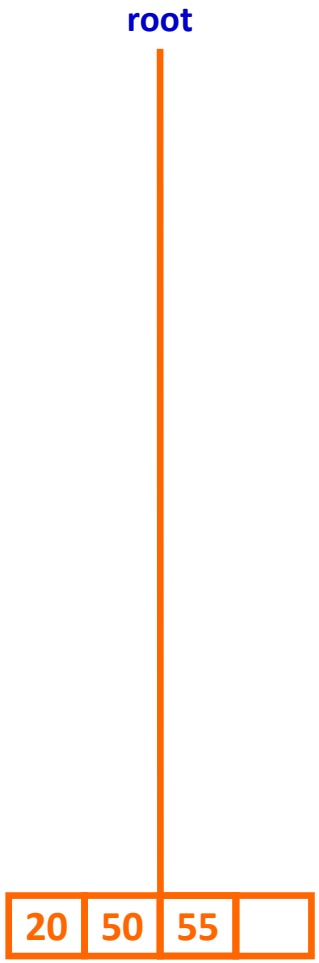
# 5-way B Tree Insert Animation




Murdoch UNIVERSITY

Insert(50)
Insert(20)
Insert(55)
Insert(47)

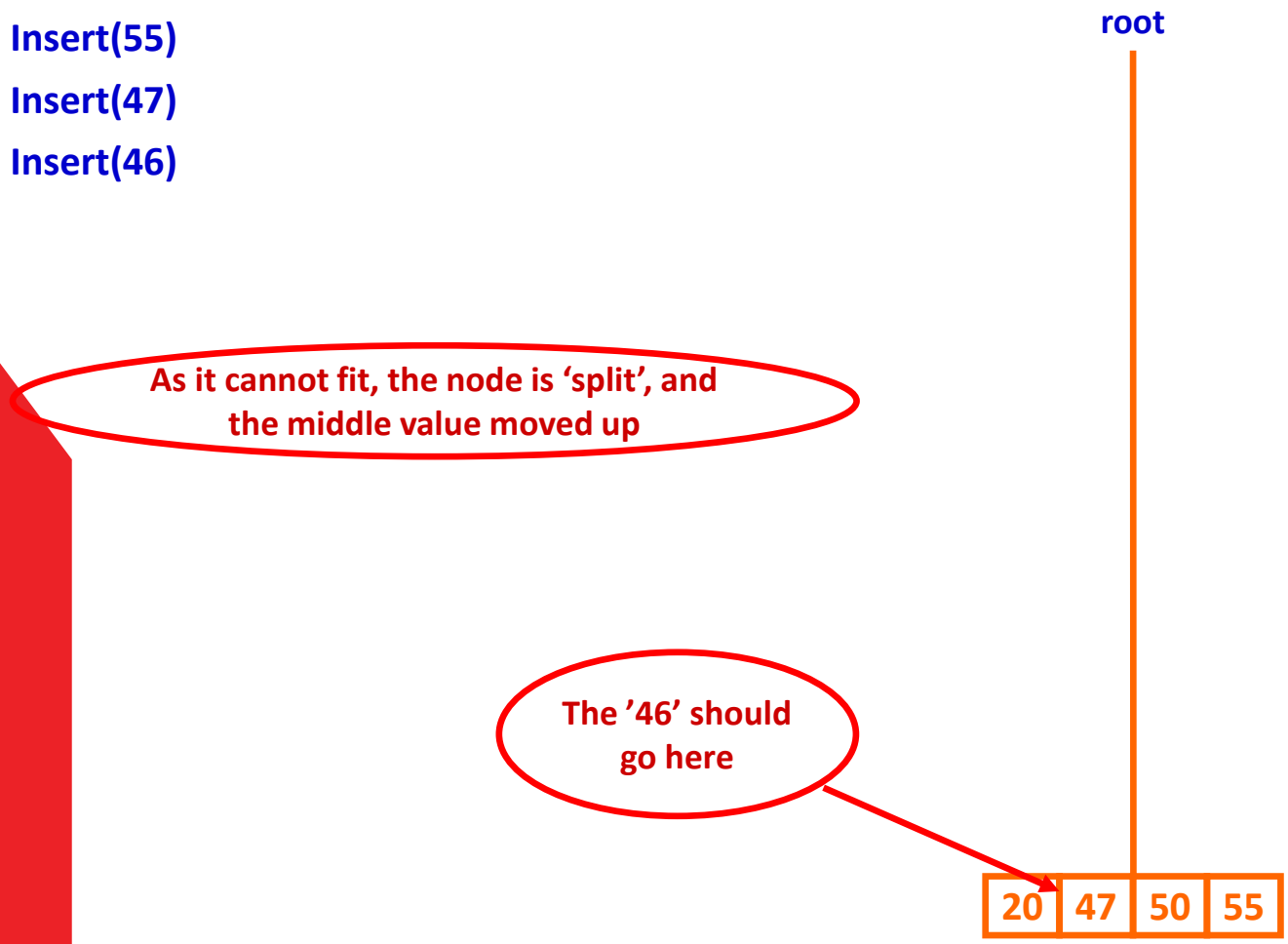
# 5-way B Tree Insert Animation




Murdoch UNIVERSITY

Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)

# 5-way B Tree Insert Animation

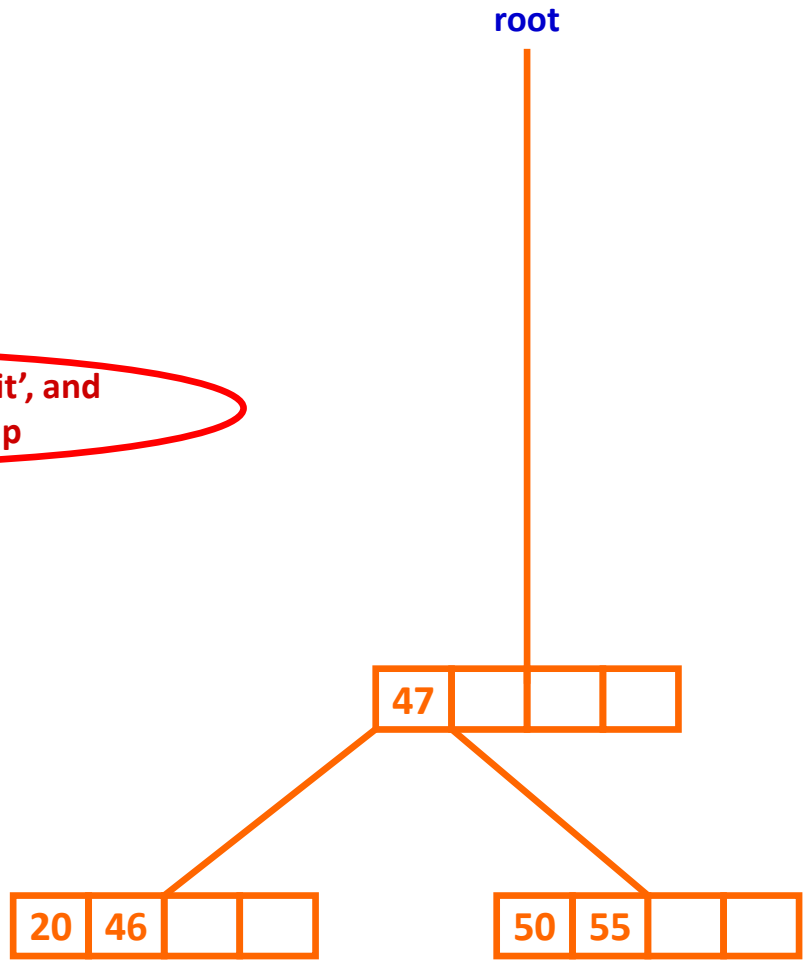



Murdoch UNIVERSITY

Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)

# 5-way B Tree Insert Animation

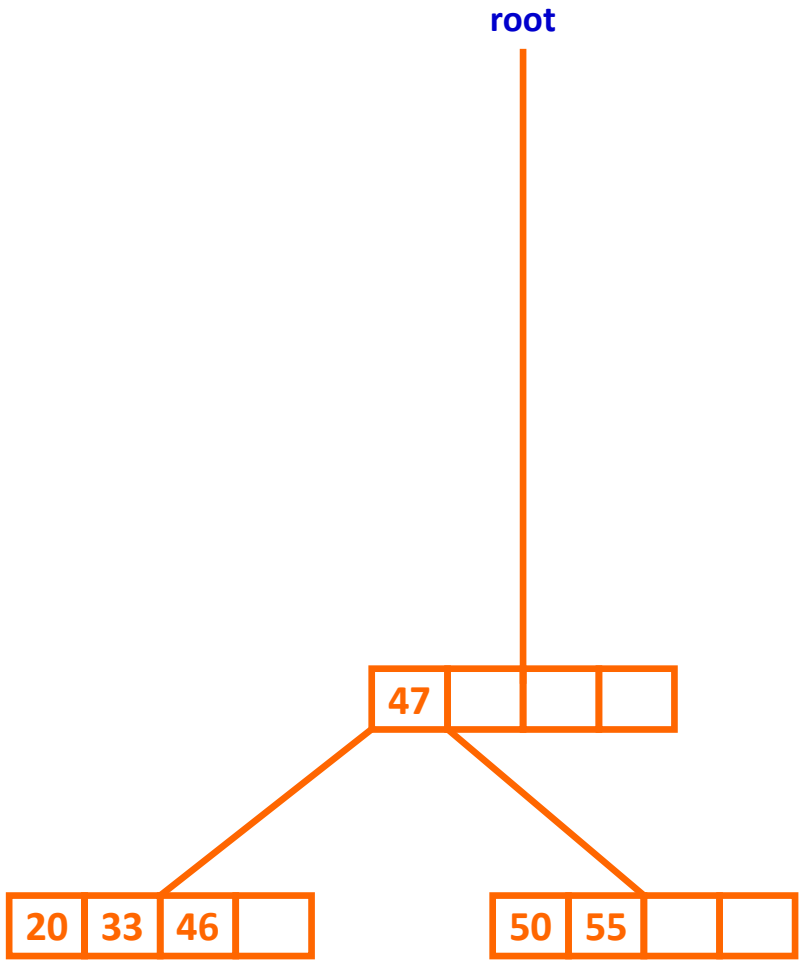
As it cannot fit, the node is 'split', and the middle value moved up




Murdoch UNIVERSITY

Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)
Insert(25)

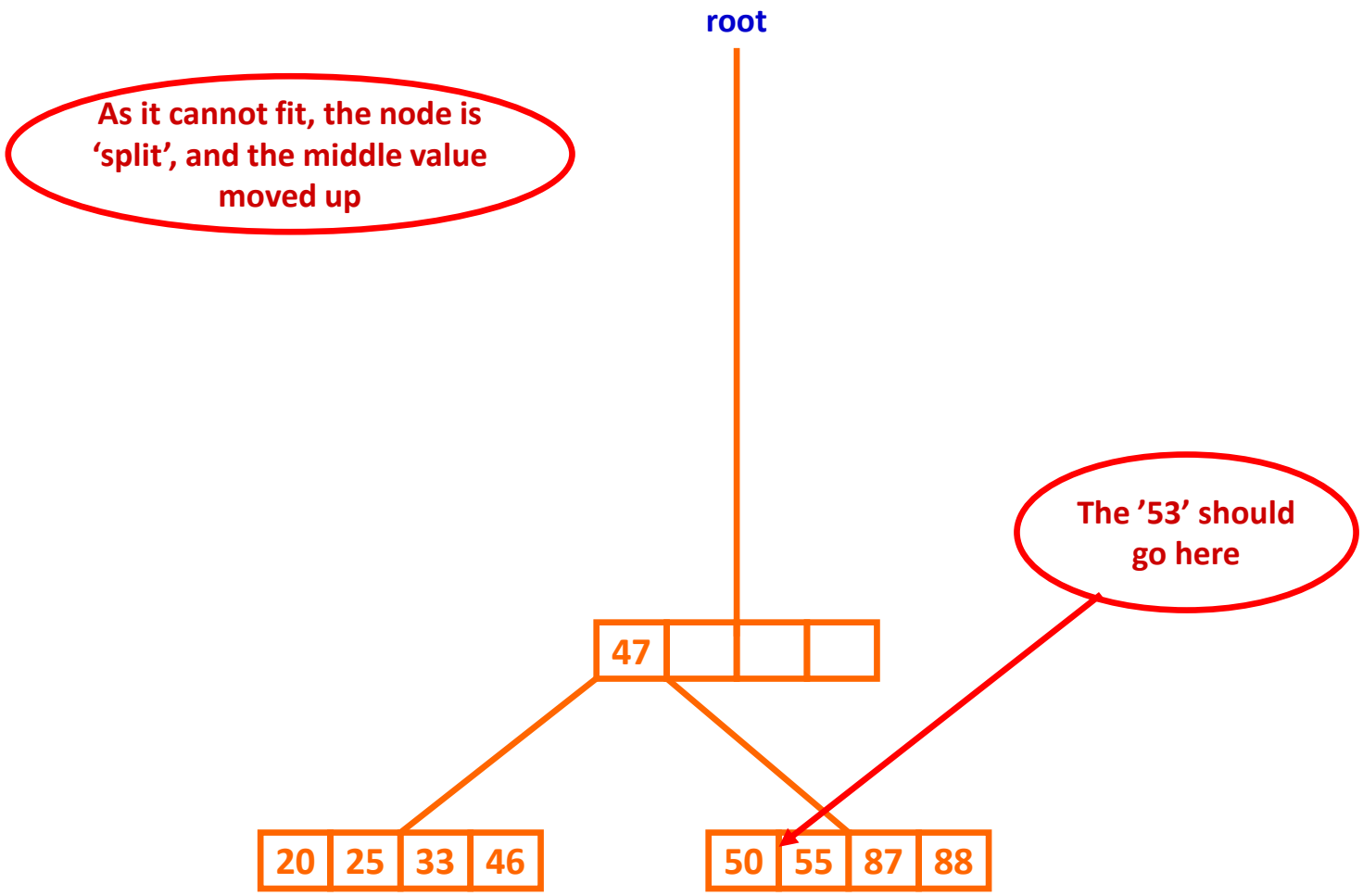
# 5-way B Tree Insert Animation




Murdoch UNIVERSITY

Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)
Insert(25)
Insert(87)
Insert(88)
Insert(53)

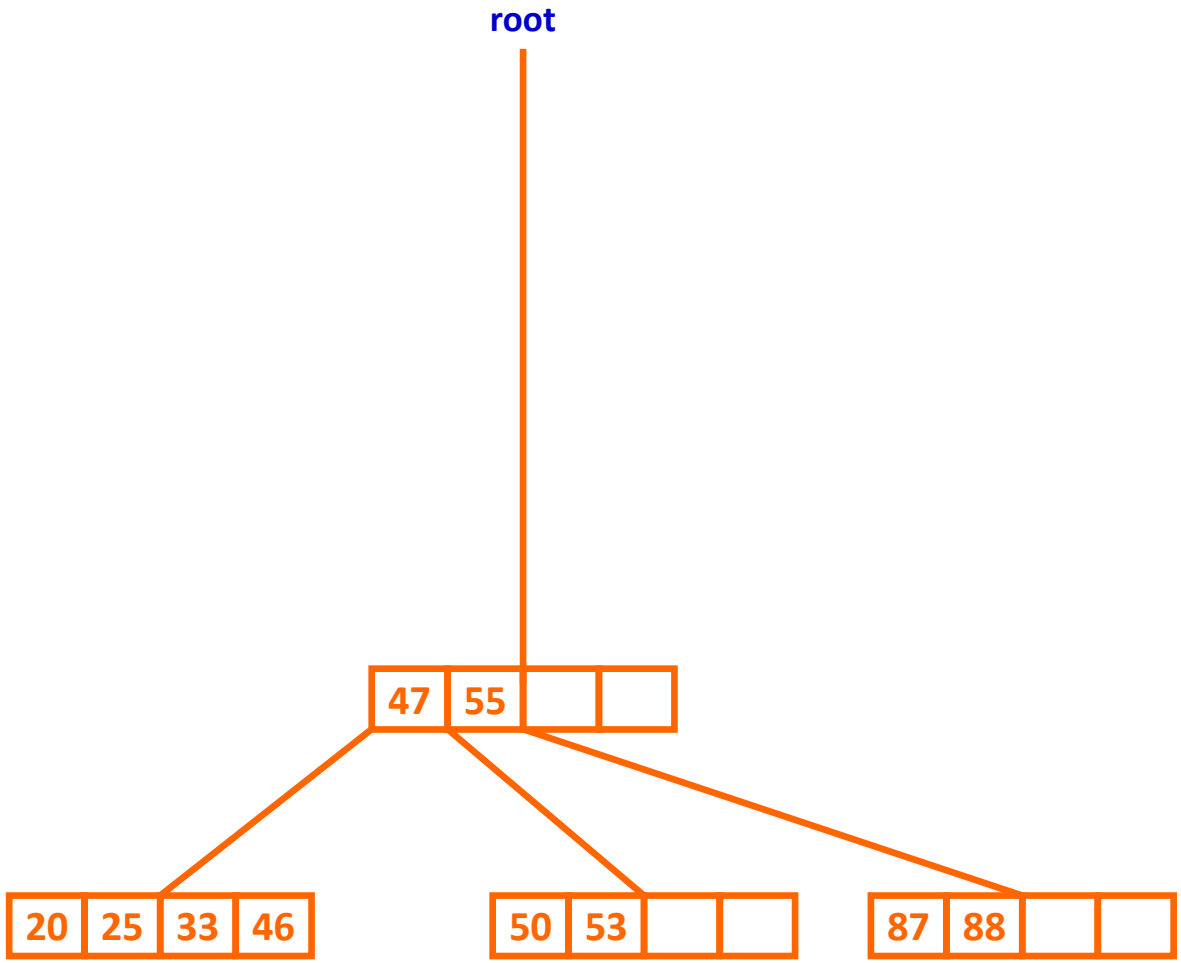
# 5-way B Tree Insert Animation




Murdoch UNIVERSITY

Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)
Insert(25)
Insert(87)
Insert(88)
Insert(53)
Insert(62)

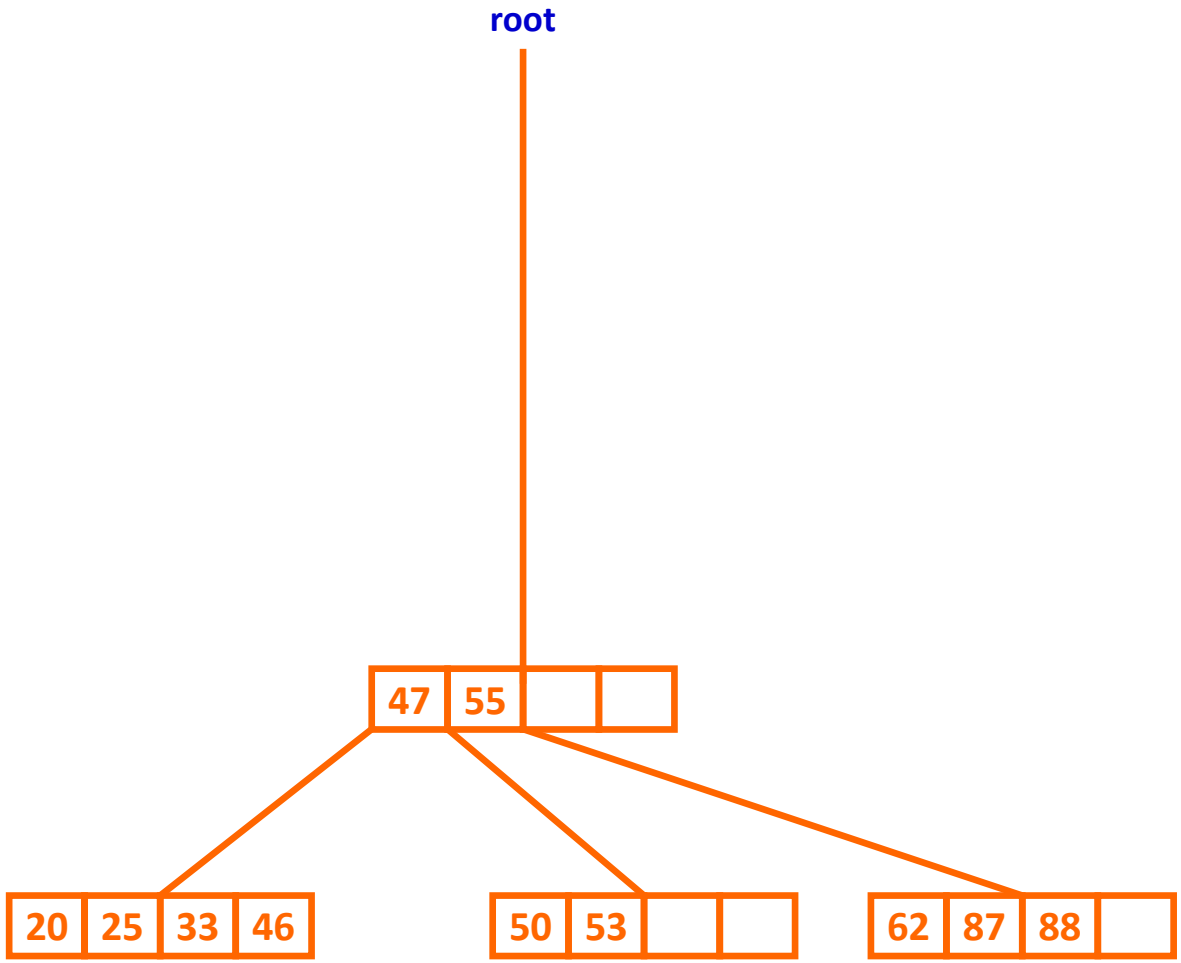
# 5-way B Tree Insert Animation




Murdoch UNIVERSITY

Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)
Insert(25)
Insert(87)
Insert(88)
Insert(53)
Insert(62)
Insert(69)

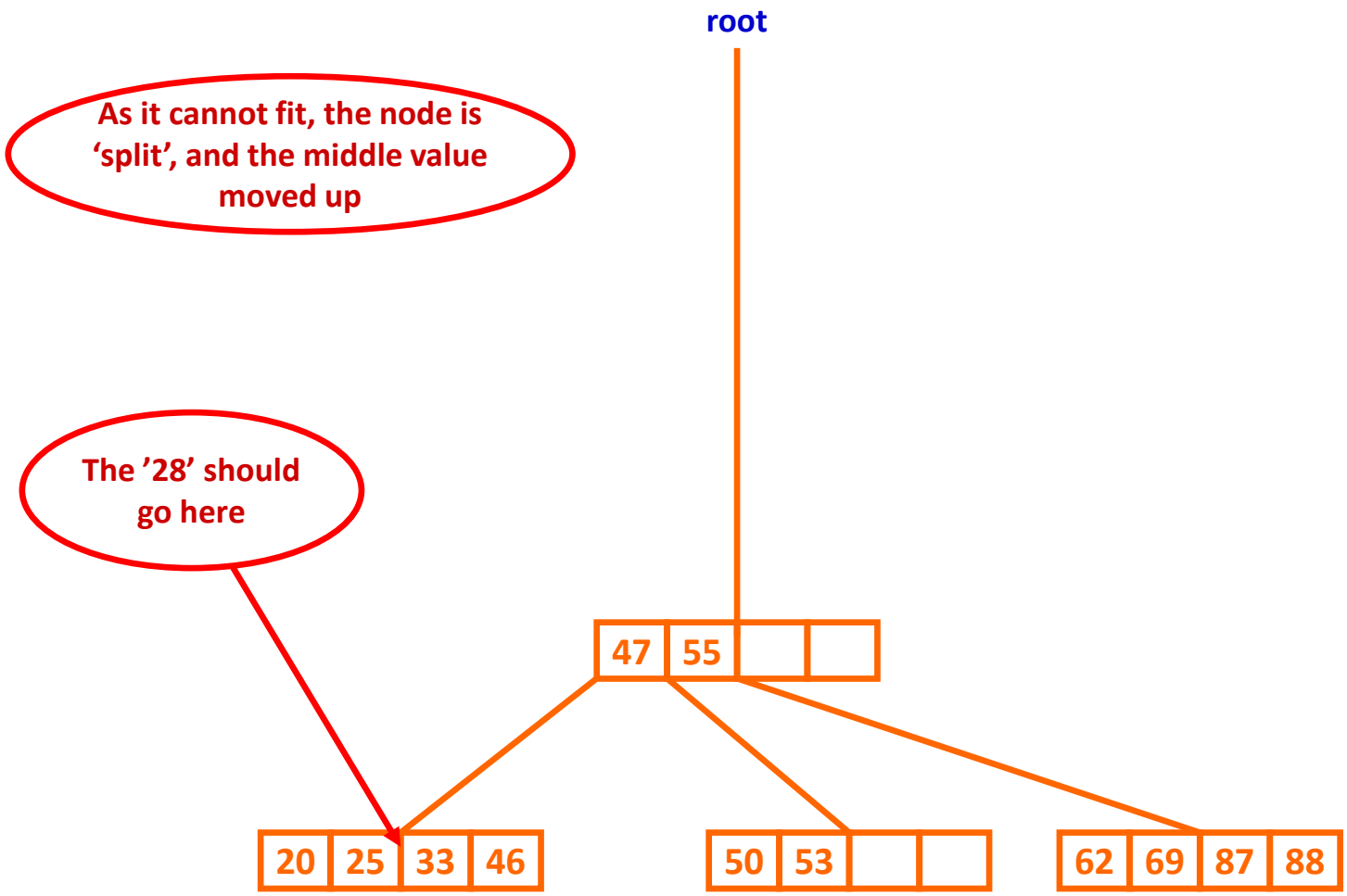
# 5-way B Tree Insert Animation




Murdoch UNIVERSITY

Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)
Insert(25)
Insert(87)
Insert(88)
Insert(53)
Insert(62)
Insert(69)
Insert(28)

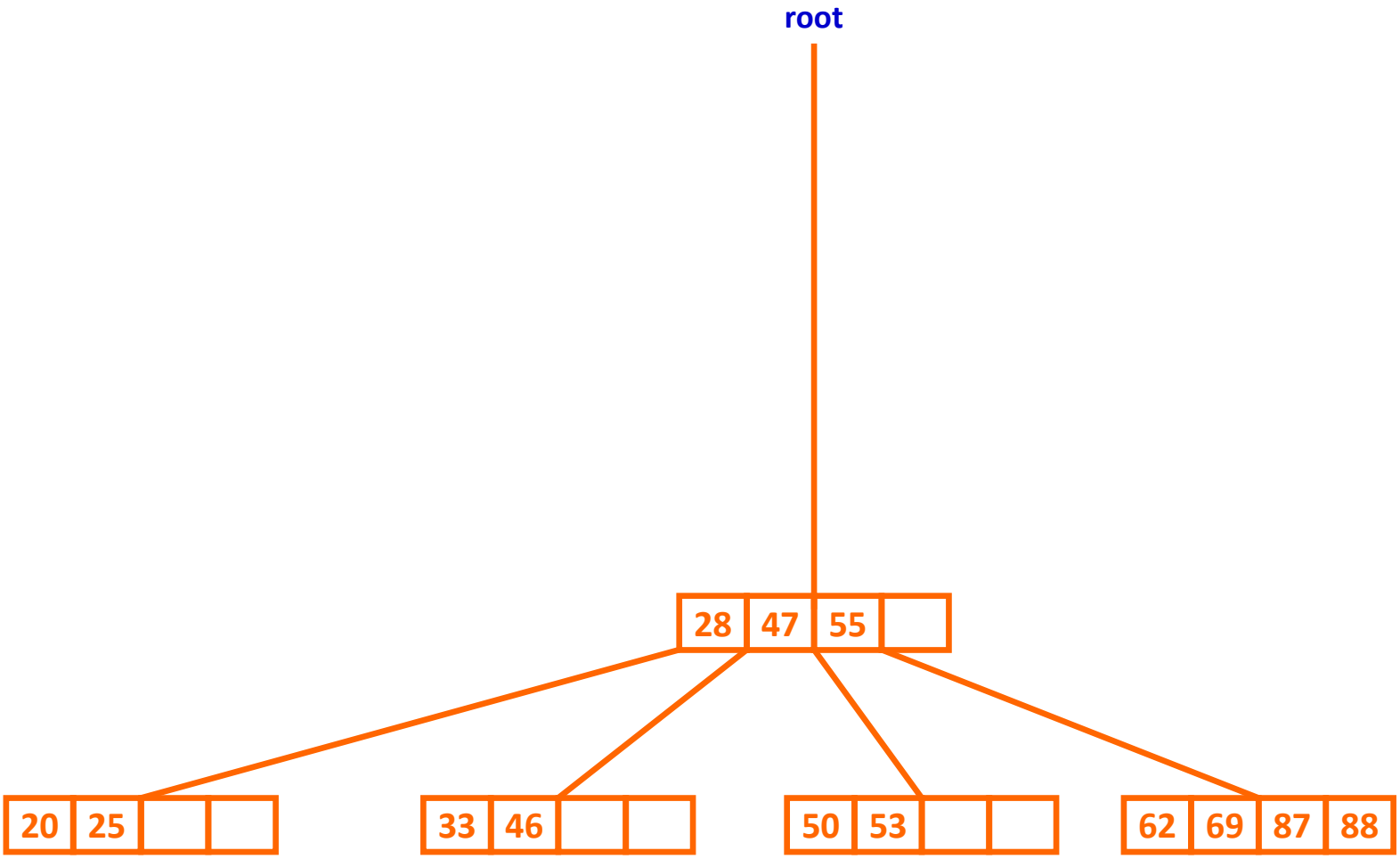
# 5-way B Tree Insert Animation




Murdoch UNIVERSITY

Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)
Insert(25)
Insert(87)
Insert(88)
Insert(53)
Insert(62)
Insert(69)
Insert(28)

# 5-way B Tree Insert Animation




Murdoch UNIVERSITY

END

# Multiway Tree vs B Tree

- The B Tree is clearly more efficient in terms of space.
- The B Tree is balanced and therefore has a lower search time.
- The B Tree, however, is more complicated to code.
- If search time is important (as with a database or list of objects in the scene of a game) the use of B Trees is essential.

Murdoch UNIVERSITY

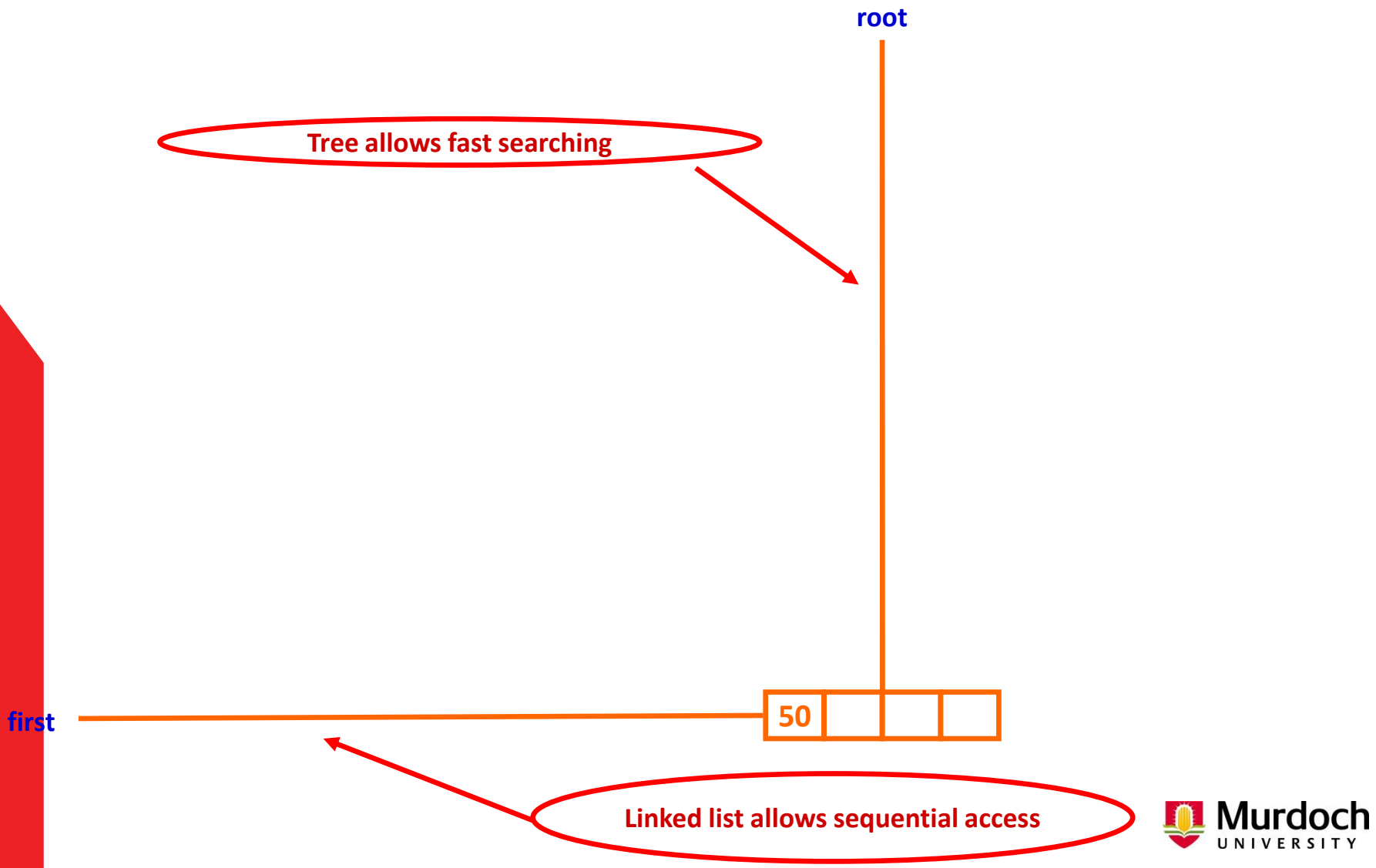
# B+ Trees

- Trees are good for searching, but have poor sequential access.
- Some databases require both types of processing, for these one uses a B+ tree.
- A B+ tree is a B Tree where only the keys are stored in the tree, all the data actually resides in the leaves.
- And the leaves are all connected with a list.
- This kind of tree is particularly useful for databases that reside entirely on disk.

Murdoch UNIVERSITY

Insert(50)
Insert(20)

# 5-way B+ Tree Insert Animation




Murdoch UNIVERSITY

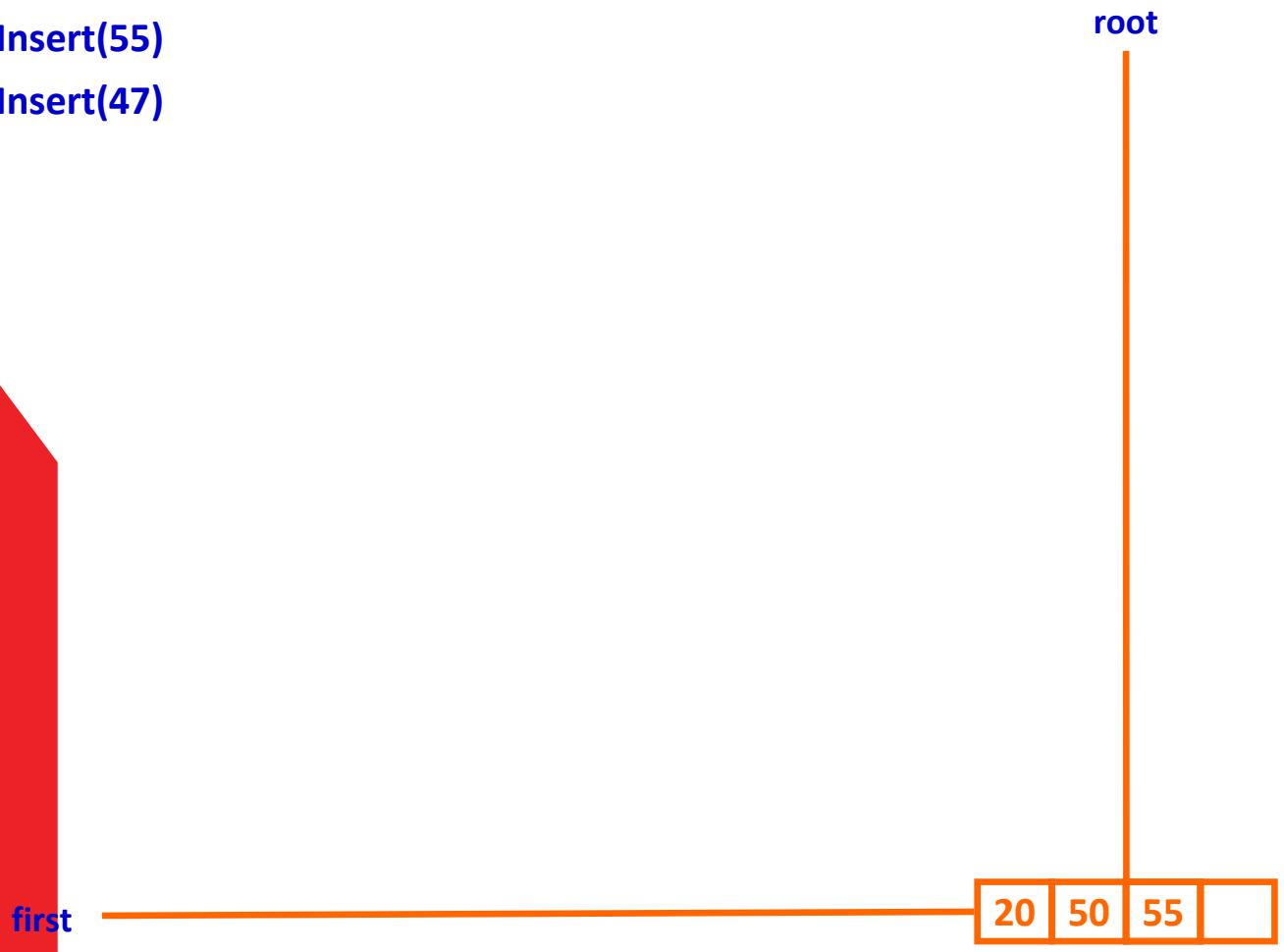
# 5-way B+ Tree Insert Animation

Insert(50)

Insert(20)

Insert(55)

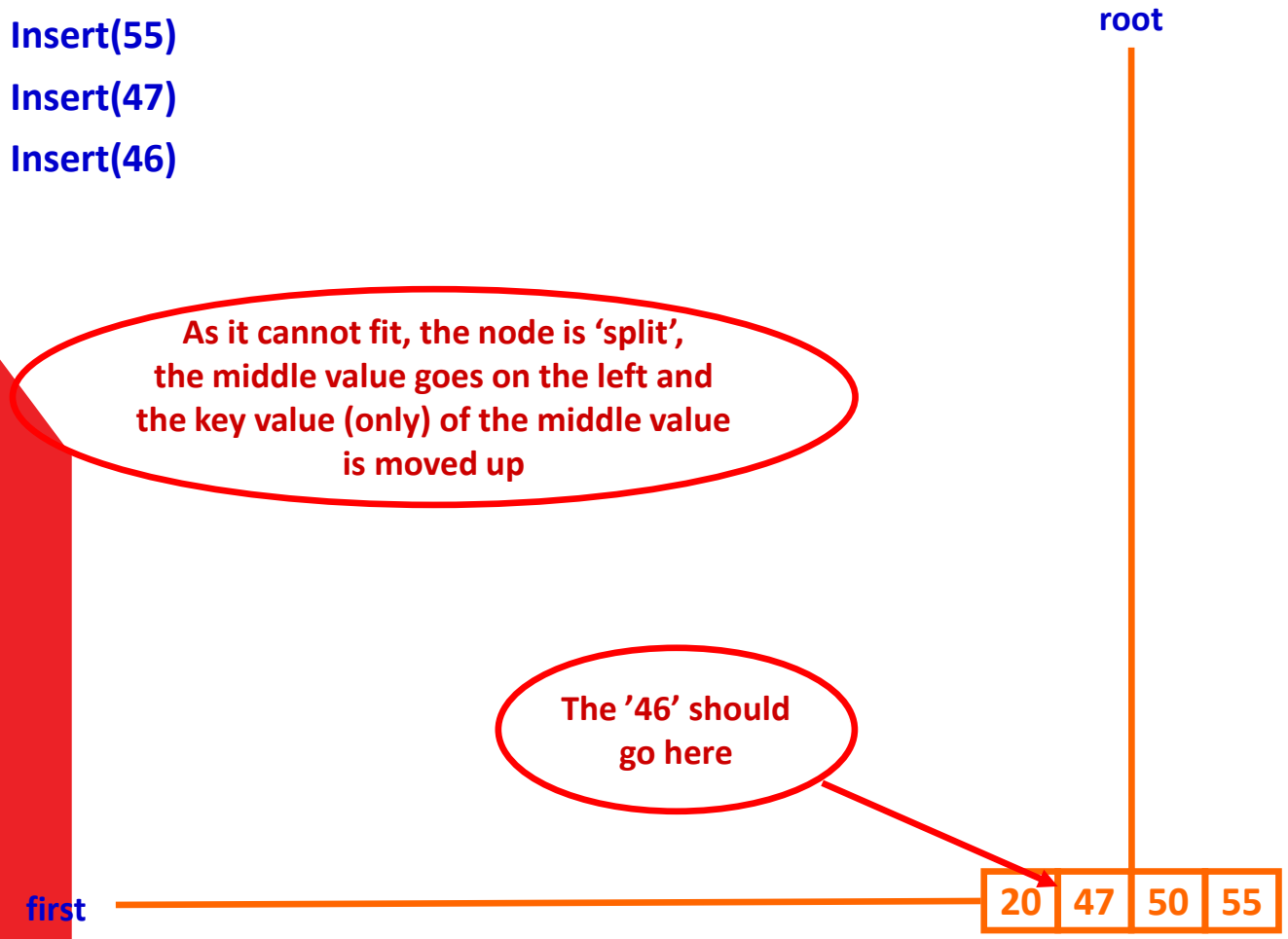
Insert(47)




Murdoch UNIVERSITY

# 5-way B+ Tree Insert Animation

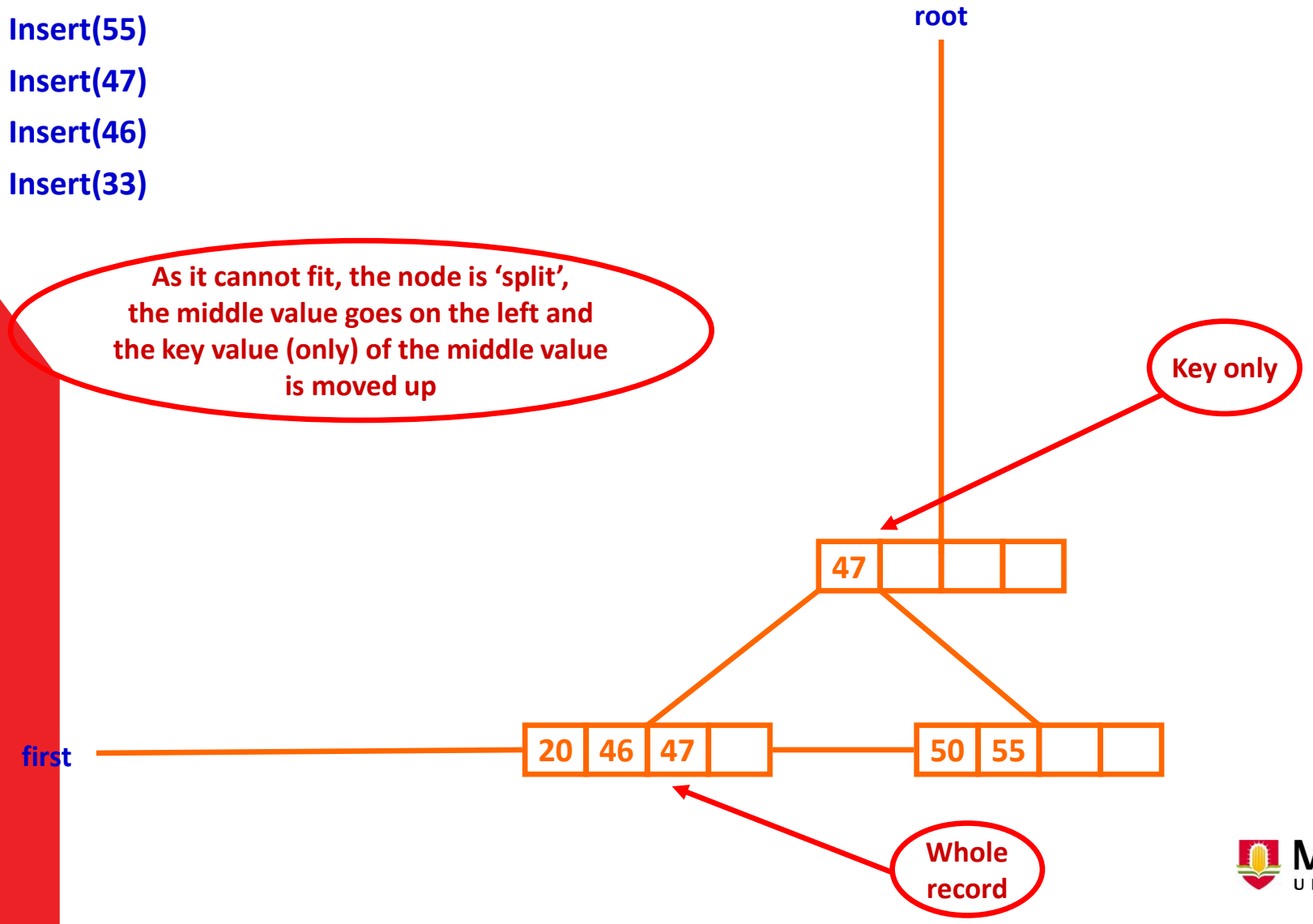
Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)




Murdoch UNIVERSITY

# 5-way B+ Tree Insert Animation

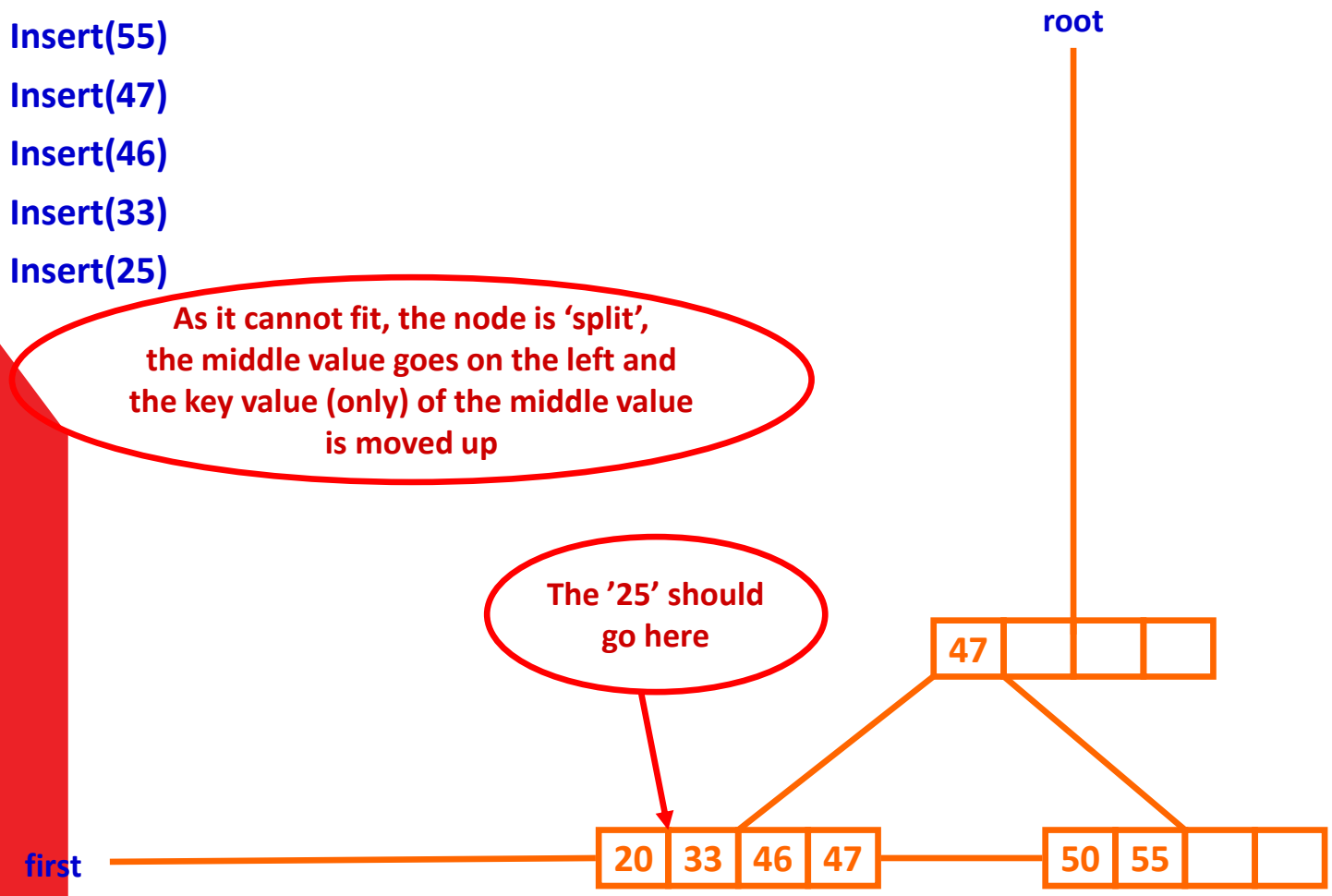
Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)




Murdoch UNIVERSITY

# 5-way B+ Tree Insert Animation

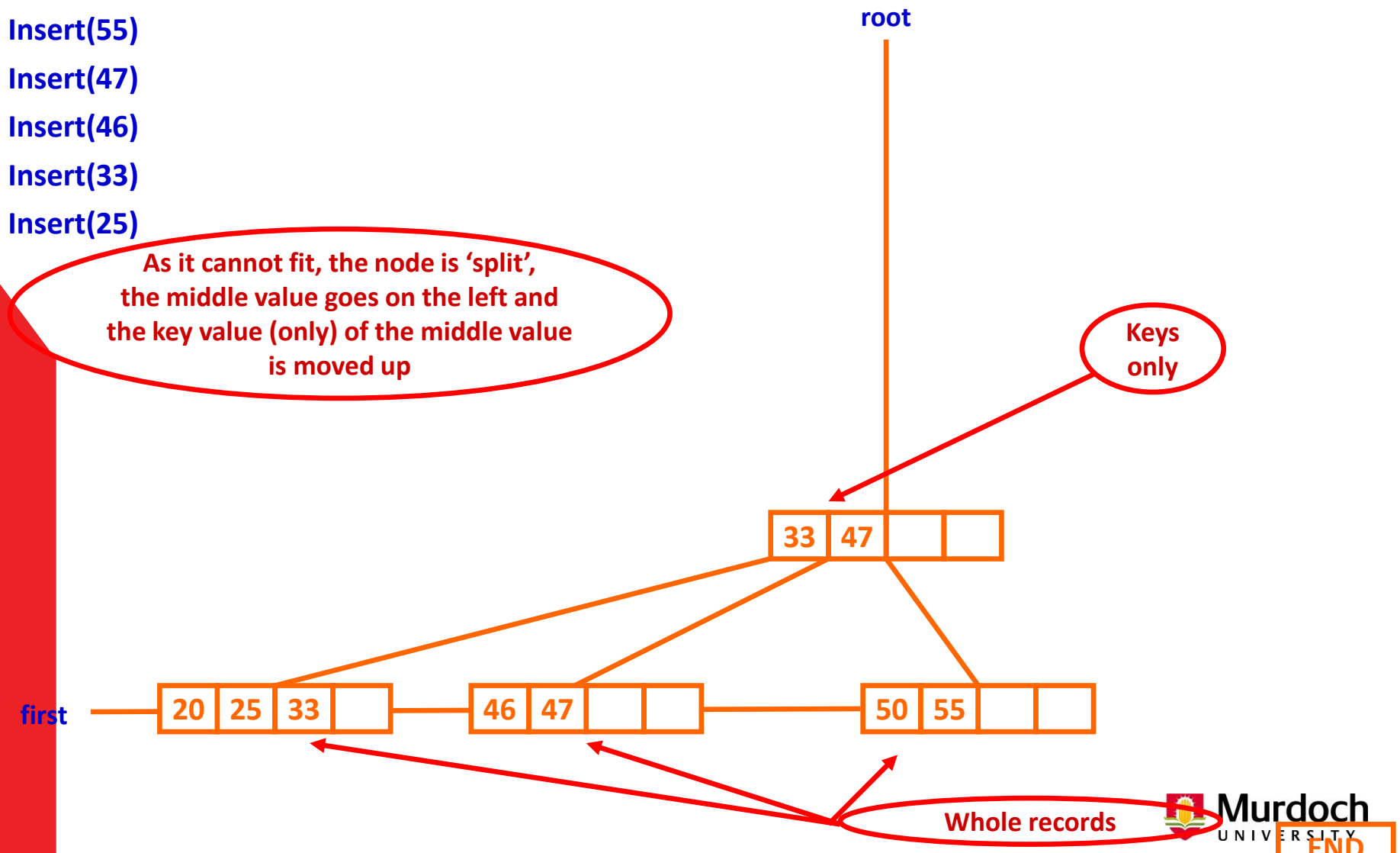
Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)
Insert(25)




Murdoch UNIVERSITY

# 5-way B+ Tree Insert Animation

Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)
Insert(25)




END

# Comparison of B and B+ Trees

- They are both balanced so that operations such as Insert and Delete can be done in O(h) time where h is the height of the tree.
- B+ trees also allow for fast sequential processing.
- B+ trees store the key only in RAM, not the whole record, therefore they use less RAM.
- Both can be tuned to have node sizes that allow fast disk reads.
- As B+ trees use less RAM, they can have larger nodes which improves the speed of operations based on the height.
- Both B Trees and B+ Trees have one major disadvantage in common: since any node can be up to half empty, they waste space.

Murdoch UNIVERSITY

# Handling the Wasted Space Problem

- A way to get around this is to really treat them as ADSs, i.e. they are conceptually B Trees but are actually stored in some other way.
- For example a vector, linked list, dynamic array etc.
- The operations (Insert, Delete, Search etc) in the interface do not change, but the internal representation and code do change.
- However, it is worth noting that although there are many different ways of solving the space problem, there will *always* be a space/time/simplicity trade off.

Murdoch UNIVERSITY

# Further exploration

- Reference book, Introduction to Algorithms. For further study, there are many tree and tree algorithms described in the reference book. For this unit, the lecture material is sufficient.

- An earlier textbook used in this unit (some years ago) is a better reference to some of the more interesting Tree (and graph) data structures. The book is available in the library. It is “Algorithms, Data Structures, and Problem solving using C++” by Mark Weiss.