List of Slides

35 `OBTInt find()`
36 Get the size of an OBT
37 `OBTInt getSize()`
38 Get the depth of an OBT
39 `OBTInt getDepth()`
40 Get the elements of an OBT
41 `OBTInt elementsAscending()`
42 Six orders of elements of an OBT
43 `OBTInt elements()`
44 `OBTInt addElements()`
45 `OBTInt addElements()`
46 `OBTInt addElements()`
47 **Section 3:** ContinuedImplementationoftrees
48 More advanced algorithms
49 Delete an item from an OBT
50 Deleting the item when it is found
51 Deleting the item when it has two empty children
52 Deleting the item when it has one empty child

90 More general trees
91 Simplicity / efficiency trade off
92 `OBTInt balance()`
93 `OBTInt balance()`
94 Features of an improved algorithm
95 `OBTInt balanceB()`
96 `OBTInt balanceB()`
97 `OBTInt shallowCopyTo()`
98 `OBTInt pushLeft()`
99 `OBTInt pushRight()`
100 `OBTInt prepend()`
101 `OBTInt append()`
102 `OBTInt pruneLargestOnto()`
103 `OBTInt pruneLargestOnto()`
104 `OBTInt pruneLargestOnto()`
105 `OBTInt pruneSmallestOnto()`
106 `OBTInt pruneSmallestOnto()`
107 `OBTInt pruneSmallestOnto()`

Topic 08

Recursive Data Structures: Ordered Binary Trees

Section 1

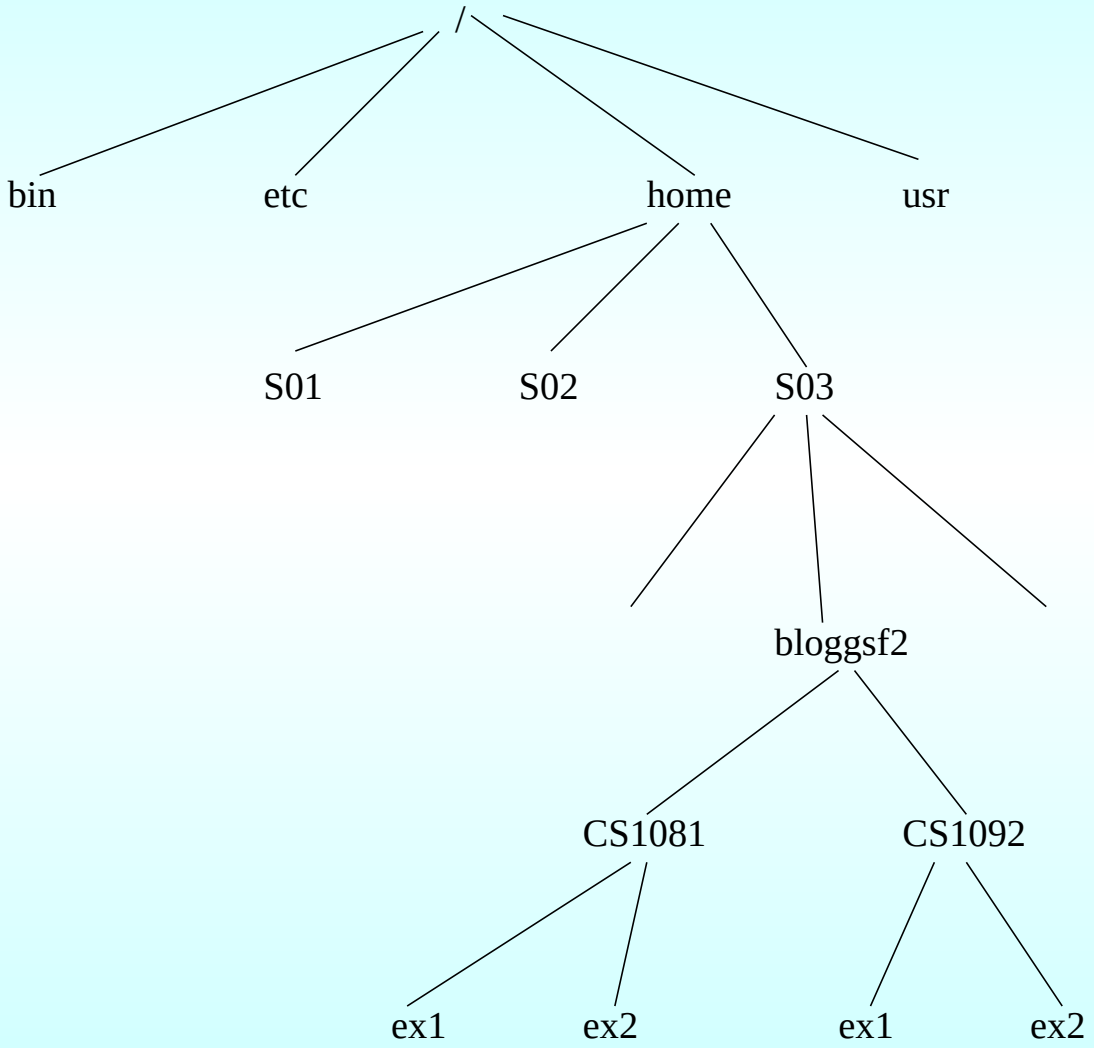
An overview of trees

Trees

- Trees consist of **tree node**s.
- Nodes might contain data.
- Nodes might have children which are (sub) trees.
- Sub-trees have exactly one parent.
- The top of the tree is called the **root**.
- Trees are recursive: children of a node are trees.

Example: Unix file system

E.g. Unix file system (ignoring links and nfs mounts).

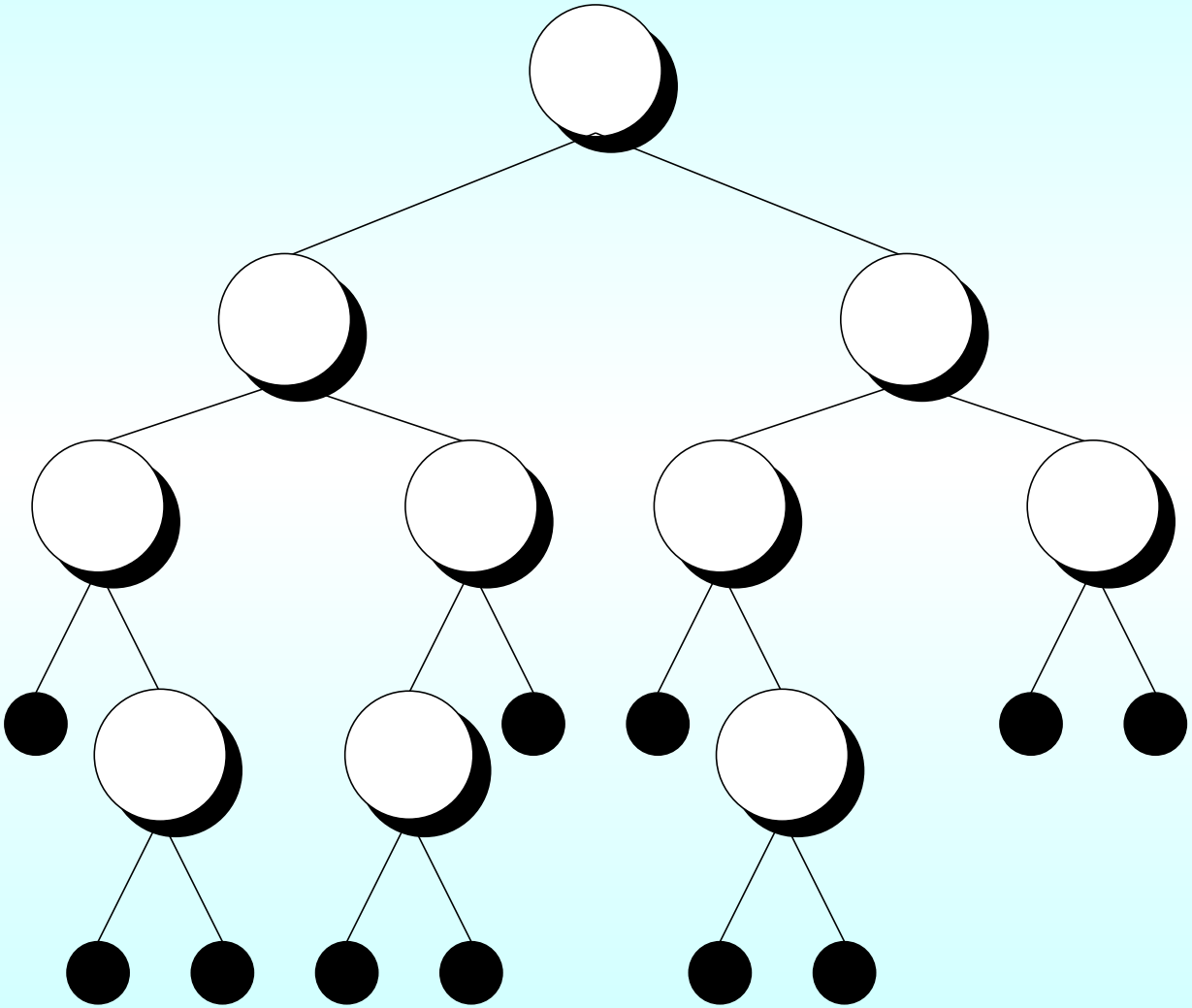


Binary trees

- Trees for which each node can have at most two children are called **binary tree**s.
- Here is one definition of binary trees (the one we shall use).
  - Each node is either an **empty node** or a **non-empty node**.
  - Empty nodes have no children.
  - All non-empty nodes have exactly two children: **left child** and **right child**.
- This definition is useful here because it is close to the way we will implement binary trees. You should compare it to the way binary trees were defined in CS1021.

Example binary tree

E.g. a **binary tree**, showing **non-empty node**s as white, and **empty node**s as black.

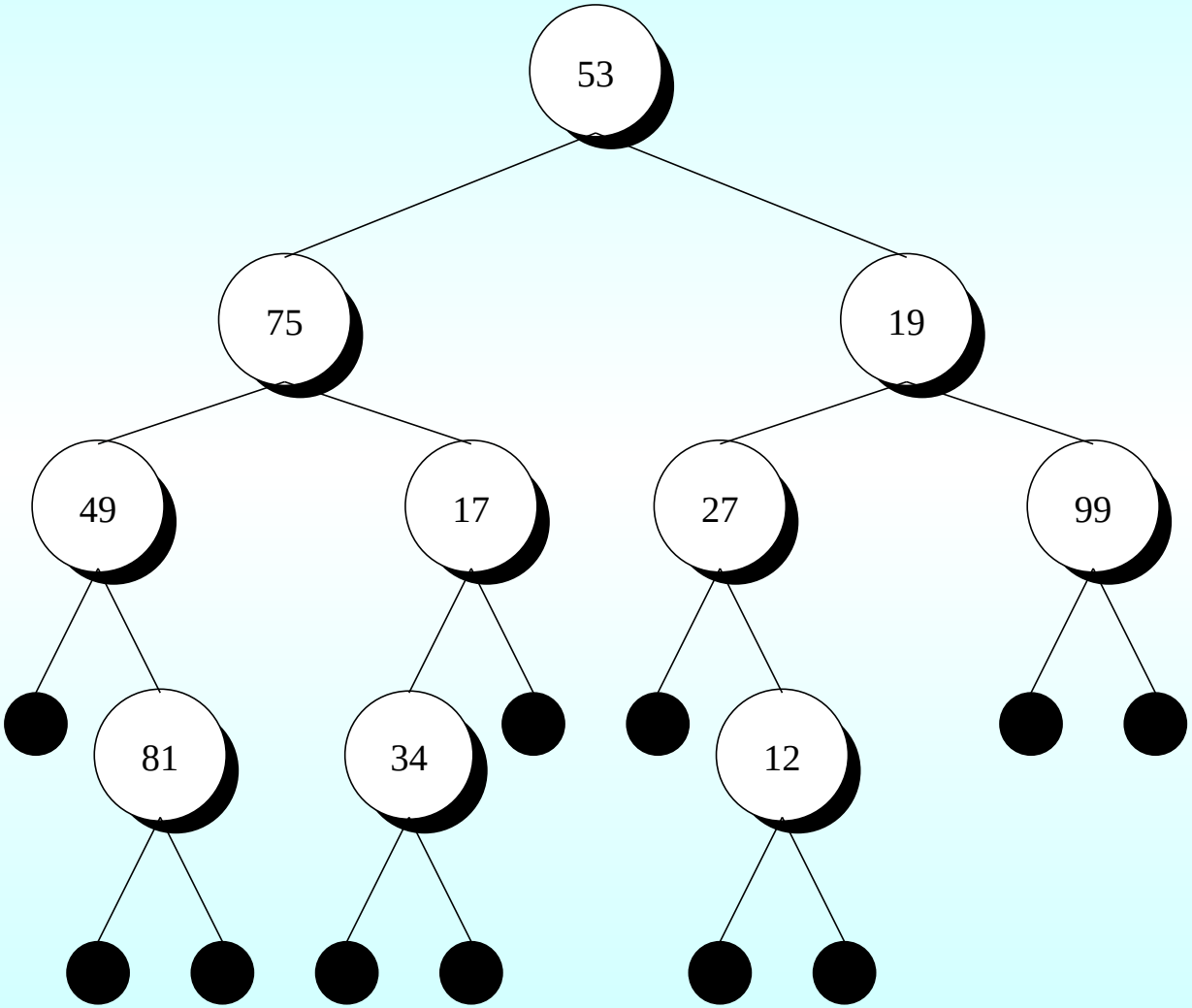


Binary trees containing data

- Trees have more than just **tree structure** – they also usually contain **tree data**.
- Here is how this is typically done (and how we shall do it):
  - All **non-empty node**s have *one* piece of data.
  - All **empty node**s do not have data: they are empty *trees*.
- In all our examples, the data will just be non-negative integers for simplicity.

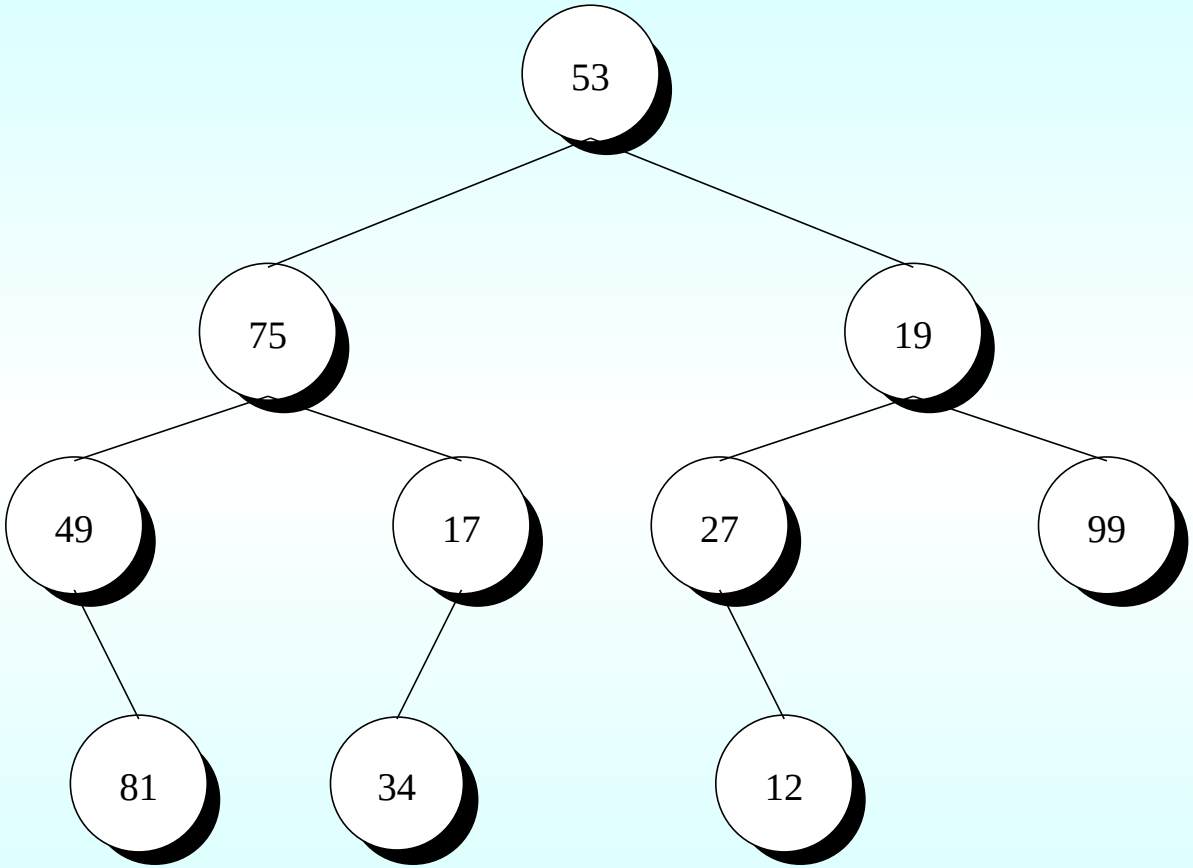
Example binary tree with tree data

E.g. a **binary tree** of numbers, showing **empty node**s and **non-empty node**s.



Example binary tree, hiding empty nodes

All **non-empty node**s have two children, so we do not need to show the **empty node**s.



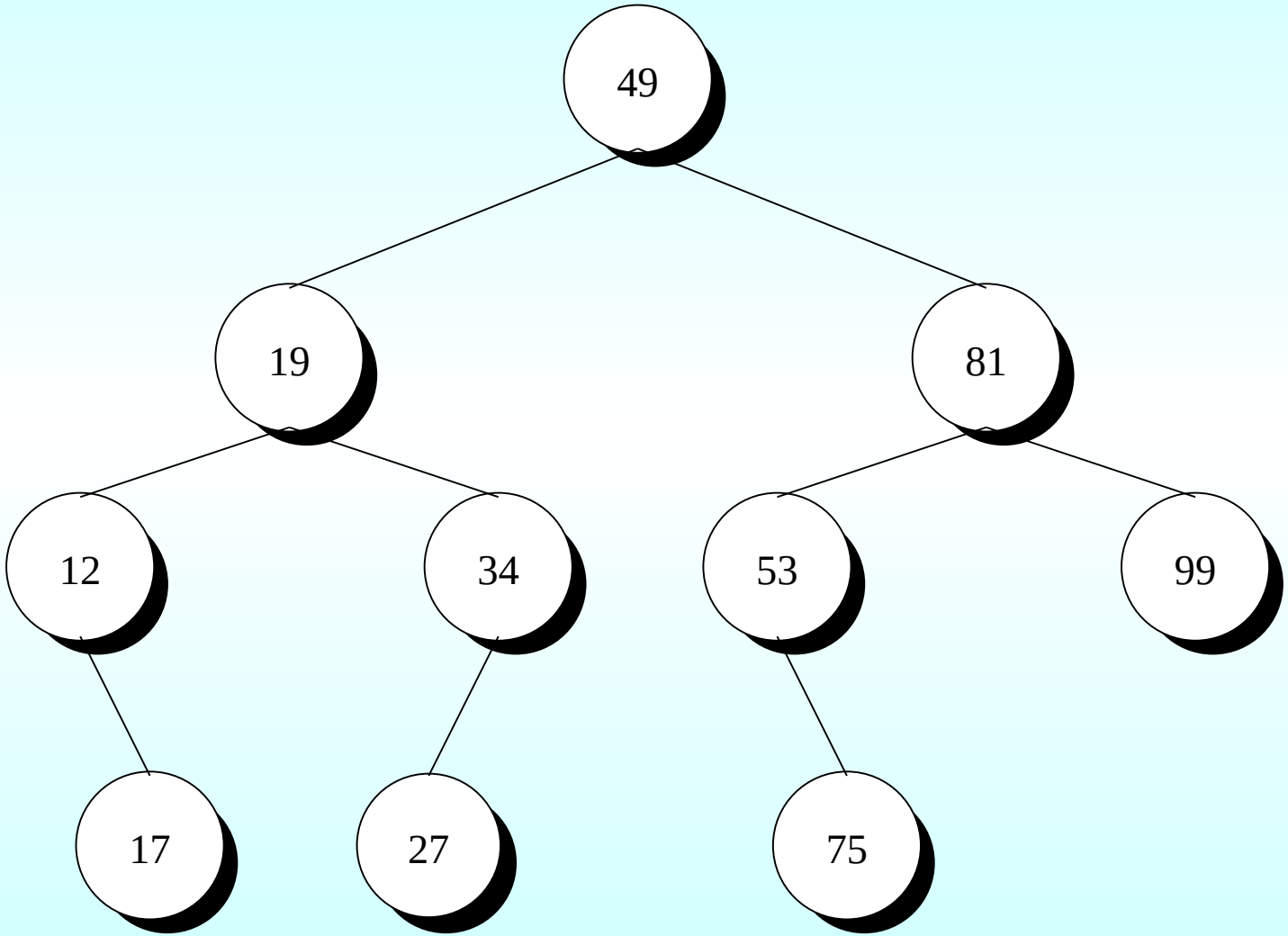
We can tell where the empty nodes are without having to see them.

Ordered binary trees (OBTs)

- An **ordered binary tree** or **OBT**, is a **binary tree** which has an a ordering of the **tree data** in it.
- There is a **total order** on the data (i.e. the data can be sorted).
- E.g. **less than or equal** on numbers is a total order.
- We shall use less than with natural numbers in our examples:
  - Each non-empty node has a natural number, *n*.
  - All numbers in the **left child** sub-tree are less than *n*.
  - All numbers in the **right child** sub-tree are greater than or equal to *n*.

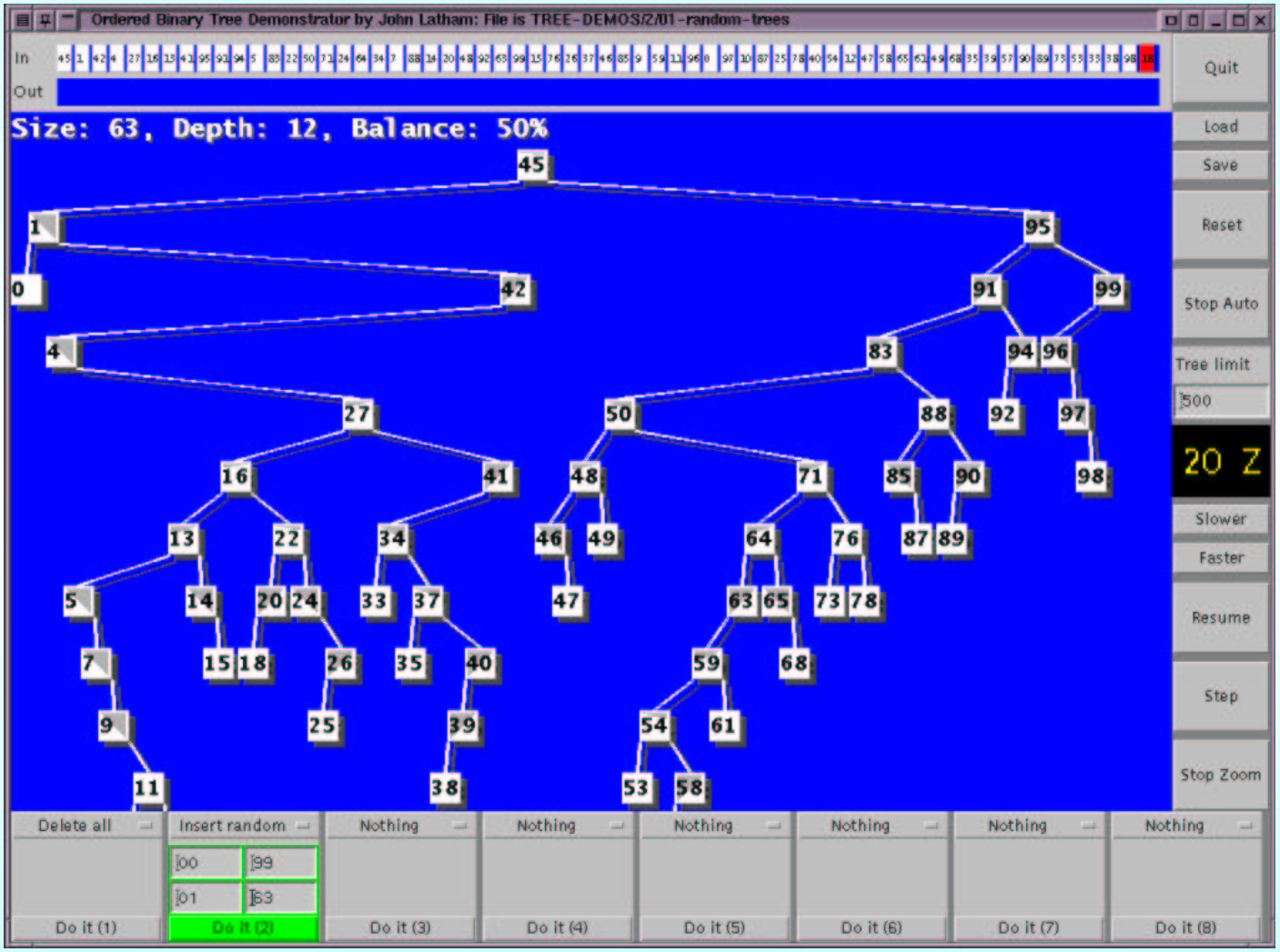
Example OBT of numbers

E.g. an **ordered binary tree** of numbers.

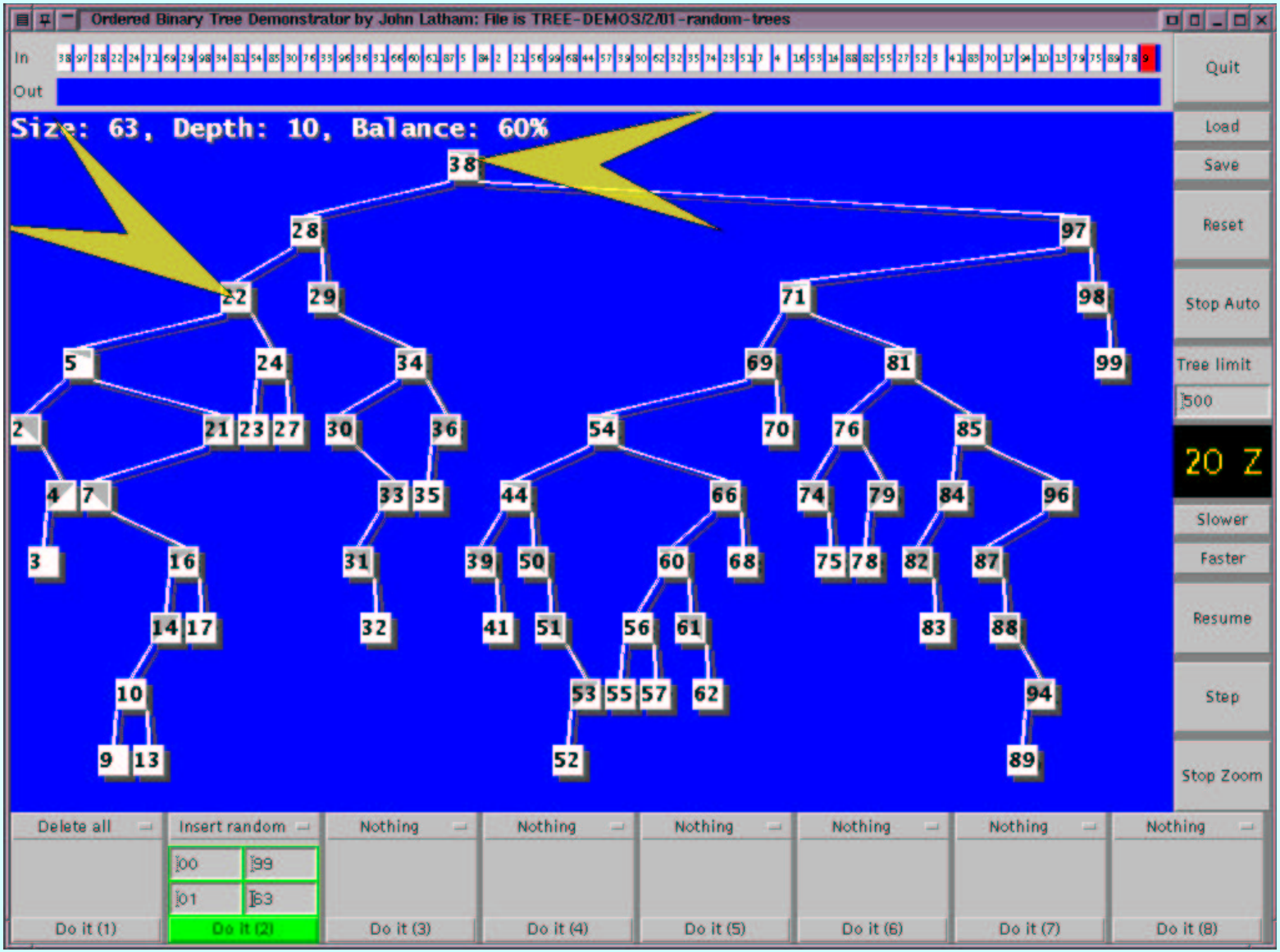


OBT demonstrator

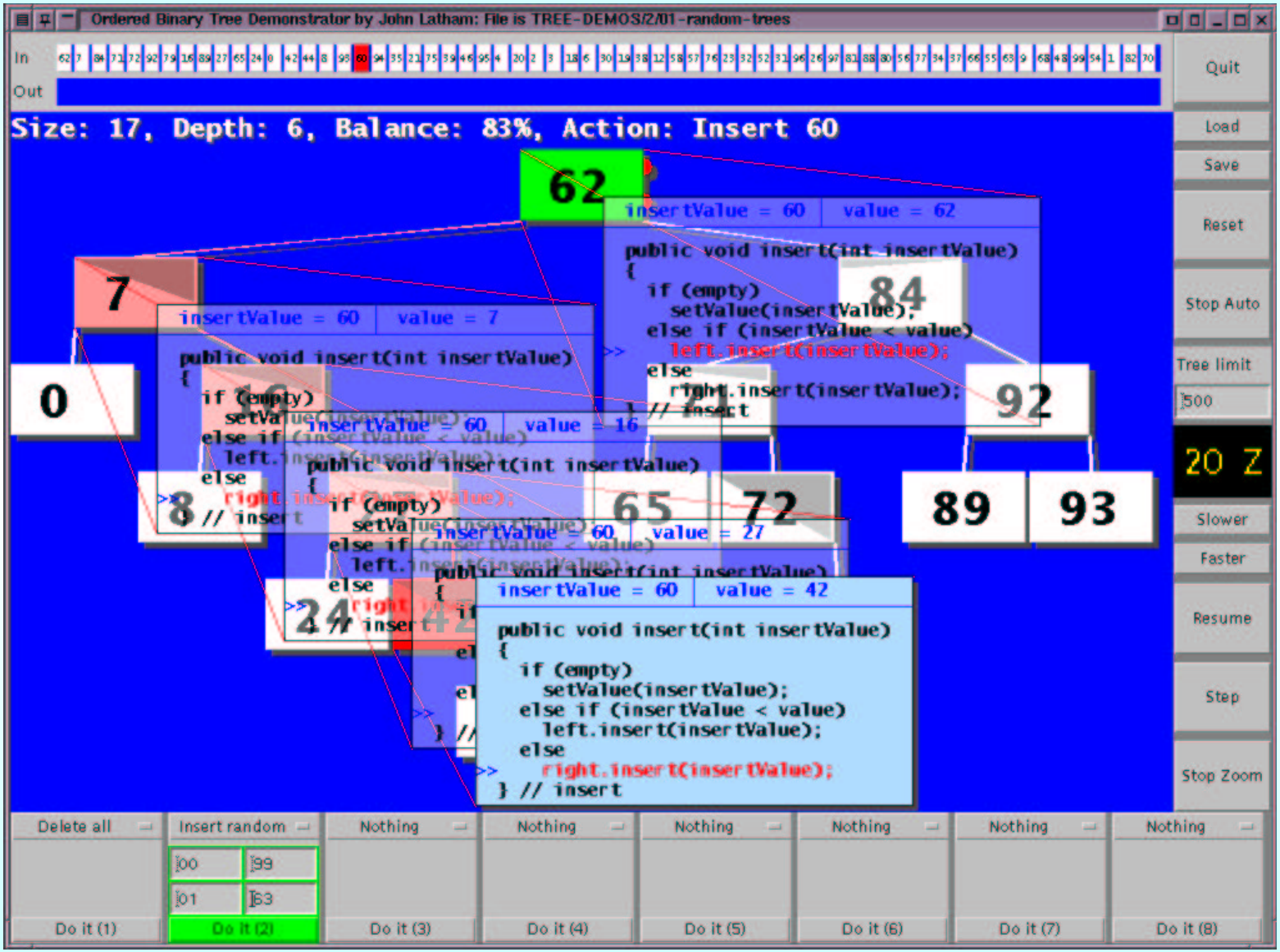
I have built an **OBT demonstrator** which we will use to explore **OBT**s.



OBT demonstrator



OBT demonstrator



Randomly created OBTs

- Let us look at some randomly created **OBT**s.
- Tree demo:  random-trees
  - Notice how the shapes vary a lot, even though these all contain 63 numbers between 0 and 99, and are ordered.
  - Notice the varying **balance** of the trees.

Even randomly created OBTs are ordered

- **OBT** shapes can look very chaotic.
- But it is only the balance which is chaotic.
- OBTs are ordered by value from left to right.
- Tree demo:  random-trees-showing-order
  - When we visit each number in ascending order, notice how the vertical line just moves from left to right, even though the horizontal line jumps up and down chaotically.

Why do we want OBTs?

- Why are **OBT**s so important to your studies?
- OBTs are a fundamental data structure, and crop up all over Computer Science.
- Searching an OBT is efficient, compared with searching a list.
  - At each node of the tree one of three things occurs:
    - We have found the item we are looking for.
    - The item is less than the value at this node, so go left.
    - The item is greater than the value at this node, so go right.
- E.g. OBTs are often used for database indices.
- Searching a list: worst case takes length of list number of steps.
- Searching an OBT: worst case takes depth of tree number of steps.
- Tree demo:  random-trees-with-random-searches

Another OBT application: tree sort

- **OBT**s are also used in **tree sort**.
- We start with an unsorted list of numbers.
- We insert these into a tree.
- Then we scan the tree in ascending order.
- Thus we obtain the original numbers in ascending order.
- Tree demo:  random-tree-sorts
  - Notice how long it takes to build the tree compared with how long it takes to scan it to get the result.

Depth of a tree

- The **depth** of a tree is its number of rows.
- The length of the longest path from the **root** to any **empty node**.
- E.g. an empty tree (i.e. just an **empty node**) has depth 0, a singleton tree (just one **non-empty node**) has depth 1.
- The depth of a tree depends on how many items are in it and how balanced it is.
- Tree demo:  random-trees-with-depth
  - Look at the depth of the trees, and compare with overall balance.

Fully balanced trees

- A **fully balanced tree** is one where every **non-empty node** has an equal number of items on its left and right.
- This can only happen if the number of items is $2^x - 1$, for some x.
- A fully balanced tree of size N has depth $log_2(N+1)$.
- Each row is twice as big as the previous one.
- Tree demo:  fully-balanced-trees
  - All these trees have a size which is $2^x - 1$ for some x.

Balanced trees

- A **balanced tree** is one where every **non-empty node** has a difference of at most one in the number of items on its left and right.
- A tree of any size can be balanced.
- Tree demo:  balanced-trees
  - Here the tree grows by one each time, and then we balance it.

Why is balance significant

- The **balance** of a tree can be important.
- Balancing a tree minimises its depth, so it is faster for searching. Obviously the deeper the tree, the longer any searches take.
- The shape of a tree depends on the order in which the items were inserted into it.
- Tree demo:  random-searching-and-balance
  - What happens if we balance the trees before doing the searches? (Replace the “Nothing” after the “Insert Random” by “Balance”.)
- Tree demo:  bad-case-searching-and-balance
  - Here some items are inserted in ascending order first, resulting in a deep tree.
  - The rest are then inserted in descending order to give a shape rather like a ‘backbone with ribs’.

Warning: building and balancing is expensive

- In the demos so far we have (mostly) hidden the work involved with building and balancing trees.
- These are not instantaneous operations.
- Tree demo:  random-tree-builds
  - The speed of inserting depends on the balance of the tree.
  - Which depends on the order of the inserts.
- Tree demo:  bad-case-tree-builds
- Balancing can be very expensive, especially if we use an ‘obvious’ simple algorithm.
  - Tree demo:  random-balances
  - Tree demo:  bad-case-balances

Section 2

Implementation of trees

Implementation of trees

- We shall study a way of implementing OBTs.
- We shall study various tree manipulating algorithms, including:
  - insert
  - find
  - getSize
  - getDepth
  - elements
- And (in the next section) more advanced algorithms, like:
  - delete
  - balance
  - balanced insert
  - balanced build from ordered list

Recursive algorithms

- We shall implement these algorithms recursively because
  - it is a natural way to process recursive structures,
  - it is (by far) the easiest way for many of the operations,
  - and you will practise recursion.

Trees are recursive

- The key to tree implementation is to recognise that trees are recursive.
- Every tree is:
  - either an **empty node**
  - or a **non-empty node**, containing:
    - a **tree data value** (a non-negative int for our examples)
    - a **left child** tree, containing only numbers $<$ data
    - a **right child** tree, containing only numbers $>=$ data
- In this way we do not distinguish between a tree and the nodes of a tree. So, a node is genuinely a sub-tree.

OBTInt code

```
public class OBTInt
{
  // A tree is either empty or not.
  private boolean empty;

  // If the tree is not empty then it has
  //          a value, a left and a right.
  // These are not used if empty == true.
  private int value;
  private OBTInt left;
  private OBTInt right;
```

OBTInt constructor

```
// Create an empty tree.
public OBTInt()
{
  setEmpty();
} // OBTInt

// Make this tree into an empty tree.
private void setEmpty()
{
  empty = true;
  value = 0; // arbitrary
  left = null;
  right = null;
} // setEmpty
```

OBTInt accessor methods

```
// See if this is an empty (sub)tree.
public boolean isEmpty()
{ return empty; }

// Get the value which is here.
public int getValue()
{ return value; }

// Get the left sub-tree.
public OBTInt getLeft()
{ return left;  }

// Get the right sub-tree.
public OBTInt getRight()
{ return right; }
```

Insert an item into an OBT

- Inserting a number, x, into an OBT raises 3 possibilities:
  - The tree is empty, and so x goes at that position.
  - Or the tree is non-empty, and contains a value v, so:
    * Either $x < v$ and x needs to be inserted on the left.
    * Or $x >= v$ and x needs to be inserted on the right.
- Tree demo:  insert
  - Note: empty trees are only shown when we are running at that node.
  - (Left click on any node to set it as a break point.)

Turning an empty tree into a singleton

```
// Store a value at this position in the tree.
private void setValue(int requiredValue)
{
  if (empty)
  {
    empty = false;
    left  = new OBTInt(); // Makes a new empty tree.
    right = new OBTInt(); // Makes a new empty tree.
  } // if
  value = requiredValue;
} // setValue
```

OBTInt insert()

```
// Insert a value, allowing multiple instances.
public void insert(int insertValue)
{
  if (empty)
    setValue(insertValue);
  else if (insertValue < value)
    left.insert(insertValue);
  else
    right.insert(insertValue);
} // insert
```

Tree demo:  insert-code

(This demonstration starts paused – click RESUME to start, or STEP it.)

Find an item in an OBT

- Finding a number, x, in an OBT raises 4 possibilities:
  - The tree is empty, and so x is not in the tree.
  - Or the tree is non-empty, and contains a value v, so:
    - $x == v$, and the number has been found.
    - Or $x < v$ and we need to search for x on the left.
    - Or $x > v$ and we need to search for x on the right.
- Assume we just want to return true or false depending on whether the item is in the tree.
- Tree demo:  find
  - Note: empty trees are only shown when we are running at that node.

OBTInt find()

```
public boolean find(int findValue)
{
  if (empty)
    return false;
  else if (findValue == value)
    return true;
  else if (findValue < value)
    return left.find(findValue);
  else
    return right.find(findValue);
} // find
```

Tree demo:  find-code

Get the size of an OBT

- Counting the items in an OBT raises 2 possibilities:
  - The tree is empty, and so the size is zero.
  - Or the tree is non-empty, and so the size is
    - the size from the left tree,
    - plus one for the data here,
    - plus the the size from the right tree.
- Tree demo:  getSize
  - Note: empty trees are only shown when we are running at that node.

OBTInt getSize()

```
public int getSize()
{
  if (empty)
    return 0;
  else
    return (left.getSize() + 1 + right.getSize());
} // getSize
```

Tree demo:  getSize-code

Get the depth of an OBT

- Finding the depth of an OBT raises 3 possibilities.
  - The tree is empty, and so the depth is zero,
  - or the tree is non-empty, and so the depth is the biggest of:
    - the depth of the left tree plus one,
    - or the depth of the right tree plus one.
- Tree demo:  getDepth
  - Note: empty trees are only shown when we are running at that node.

OBTInt getDepth()

```
public int getDepth()
{
  if (empty)
    return 0;
  else
  {
    int leftDepth = left.getDepth();
    int rightDepth = right.getDepth();
    if (leftDepth > rightDepth)
      return leftDepth + 1;
    else
      return rightDepth + 1;
  } // else
} // getDepth
```

Tree demo:  getDepth-code

Get the elements of an OBT

- We often want to produce a list of all the elements in an OBT.
- For example, get the elements from left to right in ascending order, to make a tree sort.
- Tree demo:  elements-ascending-sort
- To obtain the elements from left to right gives 2 possibilities:
  - either the tree is empty, in which case there are no elements,
  - or, it is not empty, in which case:
    - we obtain the elements from the left,
    - then the element at this position,
    - then the elements from the right.
- Tree demo:  elements

OBTInt elementsAscending()

```
public Iterator elementsAscending()
{ ArrayList elementsList = new ArrayList();
  addElementsAscending(elementsList);
  return elementsList.iterator();
} // elementsAscending

private void addElementsAscending(List elementsList)
{ if (! empty)
  {
    left.addElementsAscending(elementsList);
    elementsList.add(new Integer(value));
    right.addElementsAscending(elementsList);
  } // if
} // addElementsAscending
```

Tree demo:  elements-code

Six orders of elements of an OBT

- There are actually 6 obvious element scanning orders we can choose:
  - **ascending in-order** – left then parent then right (123).
  - **descending in-order** – right then parent then left (321).
  - **ascending pre-order** – parent then left then right (213).
  - **descending pre-order** – parent then right then left (231).
  - **ascending post-order** – left then right then parent (132).
  - **descending post-order** – right then left then parent (312).
- Tree demo:  elements-tree-copy
  - Pre-order can be used to make a structural copy of a tree.

OBTInt elements()

- A more general elements method takes an `int` order parameter and produces one of the 6 orders if the parameter is one of the values:
  123, 321, 213, 231, 132 or 312.

```
public Iterator elements(int order)
{
  ArrayList elementsList = new ArrayList();
  addElements(order, elementsList);
  return elementsList.iterator();
} // elements
```

OBTInt addElements()

```
private void addElements(int order, List elementsList)
{
  if (! empty)
  {
    switch (order)
    {
      case 123:
      case 132: left.addElements(order, elementsList);
                break;
      case 213:
      case 231: elementsList.add(new Integer(value));
                break;
      case 312:
      case 321: right.addElements(order, elementsList);
                break;
    } // switch
```

OBTInt addElements()

```
switch (order)
{
  case 213:
  case 312: left.addElements(order, elementsList);
            break;
  case 123:
  case 321: elementsList.add(new Integer(value));
            break;
  case 132:
  case 231: right.addElements(order, elementsList);
            break;
} // switch
```

OBTInt addElements()

```
    switch (order)
    {
      case 231:
      case 321: left.addElements(order, elementsList);
                break;
      case 132:
      case 312: elementsList.add(new Integer(value));
                break;
      case 123:
      case 213: right.addElements(order, elementsList);
                break;
    } // switch

  } // if
} // addElements
```

Section 3

Continued Implementation of trees

More advanced algorithms

- These more advanced algorithms are:
  - delete
  - balance
  - balanced insert
  - balanced build from ordered list

Delete an item from an OBT

- We wish to delete an occurrence of a given number from a tree.
  - If more than one occurrence, just delete the first *one* occurrence we find.
- Deleting a number, x, from an OBT raises 4 possibilities:
  - the tree is empty, and so x is not in the tree,
  - or the tree is non-empty, and contains a value v, so:
    - $x == v$, and the number has been found,
    - or $x < v$ and we need to delete x from the left,
    - or $x > v$ and we need to delete x from the right.

Deleting the item when it is found

- So far the algorithm has the same structure as most of the tree algorithms: empty, here, left or right.
- The complication in delete exists once we have found the element that needs deleting.
- The element we wish to delete is at the root of the (sub) tree, but there are 4 possibilities:
  - the tree has an empty tree on both the left and the right,
  - or the tree has an empty tree on its left only,
  - or the tree has an empty tree on its right only,
  - or the tree has a non-empty tree on both the left and the right.

Deleting the item when it has two empty children

- When both children are empty is the easiest case: all we need to do is make the tree into an empty tree.
- Tree demo:  delete-chosen-one
  - We create a random tree of 31 numbers.
  - Then we delete one of the numbers at the “bottom” of the tree.
  - (Use break point to select item: left click on the item.)
  - Remember that empty trees are not being shown.

Deleting the item when it has one empty child

- Deleting when one child is empty is the next easiest case: all we need to do is copy the value, left and right from whichever child is non-empty, so that this node is replaced by it.
- Tree demo:  delete-chosen-one
  - We create a random tree of 31 numbers.
  - Then we delete one of the numbers that has an empty left sub tree.
  - (Use break point to select item)
  - Also we delete one of the numbers that has an empty right sub tree.
  - Remember that empty trees are not being shown.

Deleting the item when it has no empty children

- Deleting when there are no empty children is the trickiest case.
- We find the largest element on the left of this one.
- We replace the value here with this largest left element.
- Then we delete this largest left element from the left tree.
- Actually, we combine finding the largest left element with deleting it.
- (We could find the smallest element on the right instead.)
- Tree demo:  delete-chosen-one
  - We create a random tree of 31 numbers.
  - Then we delete one of the numbers that has no empty children.
  - (Use break point to select item)
  - Remember that empty trees are not being shown.

OBTInt largest()

Tree demo:  largest

```
public int largest()
{
  if (empty)
    return -1;
  else if (right.empty)
    return value;
  else
    return right.largest();
} // largest
```

Tree demo:  largest-code

OBTInt smallest()

Tree demo:  smallest

```
public int smallest()
{
  if (empty)
    return -1;
  else if (left.empty)
    return value;
  else
    return left.smallest();
} // smallest
```

Tree demo:  smallest-code

OBTInt deleteLargest()

```
public int deleteLargest()
{
  if (empty)
    return -1;
  else if (right.empty)
  {
    int result = value;
    delete(value);
    return result;
  } // else if
  else
    return right.deleteLargest();
} // deleteLargest
```

OBTInt deleteSmallest()

```
public int deleteSmallest()
{
  if (empty)
    return -1;
  else if (left.empty)
  {
    int result = value;
    delete(value);
    return result;
  } // else if
  else
    return left.deleteSmallest();
} // deleteSmallest
```

OBTInt delete()

```
// Delete a value, only the first instance found.
// Returns true iff the value was found (and so deleted).
public boolean delete(int deleteValue)
{
  if (empty)
    return false;
  else if (deleteValue < value)
    return left.delete(deleteValue);
  else if (deleteValue > value)
    return right.delete(deleteValue);
```

OBTInt delete()

```
else // if (deleteValue == value)
{
  // If both children are empty then can prune it off.
  if (left.empty && right.empty)
    setEmpty();

  // Does not have two empty children,
  // so see if left is empty.
  else if (left.empty)
  {
    value = right.value;
    left = right.left;
    right = right.right;
  } // else if
```

OBTInt delete()

```
// Okay, so see if just right is empty.
else if (right.empty)
{
  value = left.value;
  right = left.right;
  left = left.left;
} // else if
```

OBTInt delete()

```
      // Okay, so replace this one with largest from left
      // and delete that.
      else
        value = left.deleteLargest();

      return true;
    } // if
  } // delete
```

Tree demo:  delete-chosen-one-code

Tree demo:  delete-random

Balance an OBT

- There are various ways to balance a tree.
- The ‘most obvious’ algorithm is as follows.
  - If (sub) tree is empty, then nothing to do.
  - Else, first balance here: move items from left to right or vice versa.
    - E.g. delete largest from left, put it here, and insert what was here on the right.
  - Then recursively balance on the left.
  - Then recursively balance on the right.
- This is not the most efficient algorithm, just the ‘most obvious’.
- We shall see some others later.
- Tree demo:  random-balances-with-ghost

OBTInt balance()

```
public void balance()
{
  if (! empty)
  {
    while (left.getSize() > right.getSize() + 1)
    {
      right.insert(value);
      value = left.deleteLargest();
    } // while
```

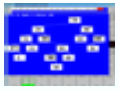
OBTInt balance()

```
      while (left.getSize() + 1 < right.getSize())
      {
        left.insert(value);
        value = right.deleteSmallest();
      } // while

      left.balance();
      right.balance();
    } // if
  } // balance
```

Tree demo:  random-balances-code

Balanced inserts

- One way to always have balanced trees is to use a balanced insert.
- This means the tree is balanced before and after the insert.
- There are two 'obvious' approaches.
  - Decide whether to insert a new item on the left or right based on the order of values in the existing tree, but first shift items left to/from right so that we balance the tree – `balancedInsertA`.
    - Tree demo:  random-balancedInsertA
  - Decide whether to insert a new item on the left or right based on the balance of the existing tree, but manipulate the values so that we maintain the order – `balancedInsertB`.
    - Tree demo:  random-balancedInsertB
- Neither of the following algorithms are *optimal* – they could be improved.

OBTInt balancedInsertA()

```
// Existing value says which way to go,
// but need to balance the sizes.
public void balancedInsertA(int insertValue)
{
  if (empty)
    setValue(insertValue);
  else if (insertValue < value) // Insert on left.
  {
    if (left.getSize() <= right.getSize())
      left.balancedInsertA(insertValue);
    else
    {
      right.balancedInsertA(value);
      int largestOnLeft = left.largest();
```

OBTInt balancedInsertA()

```
    if (largestOnLeft <= insertValue)
      value = insertValue;
    else
    {
      value = largestOnLeft;
      left.delete(largestOnLeft);
      left.balancedInsertA(insertValue);
    } // else
  } // else
} // else if
```

OBTInt balancedInsertA()

```
else // if (insertValue >= value) -- Insert on right.
{
  if (left.getSize() >= right.getSize())
    right.balancedInsertA(insertValue);
  else
  {
    left.balancedInsertA(value);
    int smallestOnRight = right.smallest();
```

OBTInt balancedInsertA()

```
        if (smallestOnRight >= insertValue)
          value = insertValue;
        else
        {
          value = smallestOnRight;
          right.delete(smallestOnRight);
          right.balancedInsertA(insertValue);
        } // else
      } // else
    } // else
  } // balancedInsertA
```

Tree demo:  random-balancedInsertA

OBTInt balancedInsertB()

```
// Existing size says where to insert,
// but first order the values for insertion.
public void balancedInsertB(int insertValue)
{
  if (empty)
    setValue(insertValue);
```

OBTInt balancedInsertB()

```
else if (left.getSize() < right.getSize())
// Insert on the left.
{
  if (insertValue > value)
  // We need to change insertValue to value.
  {
    int smallestOnRight = right.smallest();
    if (insertValue > smallestOnRight)
    // We need to put insertValue on the right.
    {
      right.delete(smallestOnRight);
      right.balancedInsertB(insertValue);
      insertValue = value; // This will go on the left.
      value = smallestOnRight;
    } // if
```

OBTInt balancedInsertB()

```
    else // if (insertValue <= smallestOnRight)
    {
      int oldInsertValue = insertValue;
      insertValue = value;
      value = oldInsertValue;
    } // else
  } // if
  left.balancedInsertB(insertValue);
} // else if
```

OBTInt balancedInsertB()

```
else if (left.getSize() > right.getSize())
// Insert on the right.
{
  if (insertValue < value)
  {
    int largestOnLeft = left.largest();
    if (insertValue < largestOnLeft)
    // We need to put insertValue on the left.
    {
      left.delete(largestOnLeft);
      left.balancedInsertB(insertValue);
      insertValue = value; // This will go on the right.
      value = largestOnLeft;
    } // if
```

OBTInt balancedInsertB()

```
    else // if (insertValue >= largestOnLeft)
    {
      int oldInsertValue = insertValue;
      insertValue = value;
      value = oldInsertValue;
    } // else
  } // if
  right.balancedInsertB(insertValue);
} // else if
```

OBTInt balancedInsertB()

```
    else // if (left.getSize() == right.getSize())
      if (insertValue < value)
        left.balancedInsertB(insertValue);
      else
        right.balancedInsertB(insertValue);
  } // balancedInsertB
```

Tree demo:  random-balancedInsertB

Building balanced tree from sorted list

- Sometimes we know the data is already ordered.
- E.g. dictionary words for a spelling checker program.
- We can efficiently build a balanced tree to store the data as we scan through it, as long as we know how many items there are before we start.
- E.g. half way though scanning the items we should have finished building the left tree, etc..
- Tree demo:  build-balanced-from-sorted-list
  - The sorted items here are obtained by scanning a random tree – this shows another way of balancing a tree.
    - Try copying to ghost before build from list.

OBTInt buildFromList()

```
private List buildElementsList;
private int buildElementsIndex;
private int buildElementsSortedLength;


// Build tree from list,
// hoping that the list is already ordered.
public void buildFromList(List elements)
{
  setEmpty();
  buildElementsList = elements;
```

OBTInt buildFromList()

```
// Find out the length of the leading sorted portion.
if (elements.size() == 0)
  buildElementsSortedLength = 0;
else
{
  buildElementsSortedLength = 1;
  while (buildElementsSortedLength < elements.size()
         && ((Integer) elements.get
                          (buildElementsSortedLength - 1))
              .compareTo
                ((Integer) elements.get
                             (buildElementsSortedLength))
             <= 0)
    buildElementsSortedLength++;
} // else
```

OBTInt buildFromList()

```
    buildElementsIndex = 0;
    buildFromList(this, buildElementsSortedLength);

    // Insert any left over that were not in order.
    while (buildElementsIndex < buildElementsList.size())
    {
      insert(((Integer) buildElementsList
                          .get(buildElementsIndex))
                .intValue());
      buildElementsIndex++;
    } // while
  } // buildFromList
```

OBTInt buildFromList()

```
private void buildFromList(OBTInt tree, int size)
{
  if (size > 0)
  {
    tree.setValue(-1); // First set to an arbitrary value.
    int leftSize = (size - 1) / 2;
    buildFromList(tree.left, leftSize);
    tree.value
      = ((Integer) buildElementsList
                     .get(buildElementsIndex))
            .intValue();
    buildElementsIndex++;
    int rightSize = size - 1 - leftSize;
    buildFromList(tree.right, rightSize);
  } // if
} // buildFromList
```

OBTInt buildFromList()

Tree demo:  build-balanced-from-sorted-list-code

End of `OBTInt`

```
} // class OBTInt
```

- We have now seen a characterisation of the implementation of ordered binary trees.
- Our coverage has not been too concerned with efficiency of the implementation, although some efficiency issues have been exposed.

Section 4

Concluding Implementation of trees

More efficient empty trees

- The implementation we have seen is more suitable for study than for real applications.
- The main inefficiency, in the name of clarity, is the way we model empty trees.
- For a tree containing N numbers, we need N instances of `OBTInt` each storing a number, and also $N + 1$ instances storing empty trees.
- For real applications, we would use a `null` reference to represent empty trees.
- This requires us to alter the most of the tree operation methods, making them a little more obscure.
- However, the basic algorithms we have studied would be the same.

insert() as we had it

```
// Insert a value, allowing multiple instances.
public void insert(int insertValue)
{
  if (empty)
    setValue(insertValue);
  else if (insertValue < value)
    left.insert(insertValue);
  else
    right.insert(insertValue);
} // insert
```

Turn `null` into empty before recursion

```
public void insert(int insertValue)
{
  if (empty)
    setValue(insertValue);
  else if (insertValue < value)
  {
    if (left == null) left = new OBTInt();
    left.insert(insertValue);
  }
  else
  {
    if (right == null) right = new OBTInt();
    right.insert(insertValue);
  }
} // insert
```

Or, return the new tree

```
public static OBTInt insert(OBTInt tree, int insertValue)
{
  if (tree == null)
    return new OBTInt(insertValue);
  else
  {
    if (insertValue < value)
      left = insert(left, insertValue);
    else
      right = insert(right, insertValue);
    return tree;
  }
} // insert
```

```
myTree = OBTInt.insert(myTree, 10);
```

Storing depth and size

- We have seen algorithms to compute the size and depth of trees.
- The size and to a lesser extent the depth of trees is also used a lot in other algorithms.
- Computing the size or depth of a tree takes time proportional to the size of that tree.
- For real applications we would store the size and depth of the tree at each node, each in an extra instance variable.
- The various tree manipulating algorithms would need to be altered so that they maintain these size and depth values.

Separating Tree and Node

- In our implementation, we did not distinguish between a tree and the nodes it is made up of: we just had one class.
- For real applications we would probably have two classes, one for trees and one for tree nodes.
- This ties in with using `null` references for empty trees.
- For example, an instance of `Tree` would contain a `root` variable, of type `TreeNode`. If this variable is `null` then the tree is empty.
- Hence we could still, for example, add items into an existing tree, even if that tree is empty: we would not use `null` references of type `Tree` to denote an empty tree.

More general trees

- We have seen trees of natural numbers.
- This is enough to characterise the tree properties we wanted to study.
- For real applications we would want something more general – we can store any objects in an OBT as long as they have a total order.
- In the laboratory you will implement an OBT of `Comparable` objects.

Simplicity / efficiency trade off

- A recuring principle in programming is the typical trade off between simplicity and efficiency, and we have seen that occur in our study of OBTs.
- A good example is our algorithm to balance trees. We have noted that it is slow, but it's simplicity is attractive.
- As a finale for trees, we briefly look at a more complex, more efficient **in place algorithm** for balancing trees.
- However, don't forget you have also seen an algorithm to build a balanced tree from a sorted list, and seen that used to balance an existing tree. That did require the numbers to be stored in a list before the tree was rebuilt: it was not an in place algorithm.

OBTInt balance()

Our simplest balance algorithm:

```
public void balance()
{
  if (! empty)
  {
    while (left.getSize() > right.getSize() + 1)
    {
      right.insert(value);
      value = left.deleteLargest();
    } // while
```

OBTInt balance()

```
      while (left.getSize() + 1 < right.getSize())
      {
        left.insert(value);
        value = right.deleteSmallest();
      } // while

      left.balance();
      right.balance();
    } // if
  } // balance
```

Tree demo:  random-balanceAs

Features of an improved algorithm

- Our improved algorithm has the following features.
  - It shifts chunks of tree from left to right or right to left, rather than just one node at a time.
  - When bits of tree are shifted over, they are placed at the top of the other side, rather than at the bottom, so they are more quickly accessible for chopping up and shifting about again.
- Tree demo:  random-balanceBs
  - Try it with orderded tree inserts for a worst case scenario.

    Tree demo:  ascending-balanceBs

OBTInt balanceB()

The more complex balance algorithm:

```
public void balanceB()
{
  if (! empty)
  {
    if (left.getSize() > right.getSize() + 1)
    {
      int pruneSize
        = (left.getSize() - right.getSize()) / 2;
      right.pushLeft(value);
      left.pruneLargestOnto(pruneSize, right);
      value = right.deleteSmallest();
    } // if
```

OBTInt balanceB()

```
    else if (left.getSize() + 1 < right.getSize())
    {
      int pruneSize
        = (right.getSize() - left.getSize()) / 2;
      left.pushRight(value);
      right.pruneSmallestOnto(pruneSize, left);
      value = left.deleteLargest();
    } // else if

    left.balanceB();
    right.balanceB();
  } // if
} // balanceB
```

OBTInt shallowCopyTo()

```
private void shallowCopyTo(OBTInt copyTo)
{
  copyTo.value = value;
  copyTo.empty = empty;
  copyTo.left = left;
  copyTo.right = right;
} // shallowCopyTo
```

OBTInt pushLeft()

```
// Prepend new value on left of tree, at the top.
private void pushLeft(int newValue)
{
  OBTInt newRight = new OBTInt();
  shallowCopyTo(newRight);
  setEmpty();
  setValue(newValue);
  right = newRight;
} // pushLeft
```

OBTInt pushRight()

```
// Append new value on right of tree, at the top.
private void pushRight(int newValue)
{
  OBTInt newLeft = new OBTInt();
  shallowCopyTo(newLeft);
  setEmpty();
  setValue(newValue);
  left = newLeft;
} // pushRight
```

OBTInt prepend()

```
// Tree data must all be smaller than this one.
private void prepend(OBTInt tree)
{
  if (empty)
  {
    value = tree.value;
    empty = tree.empty;
    left = tree.left;
    right = tree.right;
  } // if
  else
    left.prepend(tree);
} // prepend
```

OBTInt append()

```
// Tree data must all be larger than this one.
private void append(OBTInt tree)
{
  if (empty)
  {
    value = tree.value;
    empty = tree.empty;
    left = tree.left;
    right = tree.right;
  } // if
  else
    right.append(tree);
} // append
```

OBTInt pruneLargestOnto()

```
private void pruneLargestOnto(int pruneSize, OBTInt result)
{
  while (pruneSize > 0 && ! empty)
  {
    if (right.empty) // Then delete this value.
    {
      result.pushLeft(value);
      empty = left.empty;
      value = left.value;
      right = left.right;
      left = left.left;
      pruneSize--;
    } // if
```

OBTInt pruneLargestOnto()

```
else if (right.getSize() <= pruneSize)
// Delete all right.
{
  OBTInt prune = right;
  pruneSize -= prune.getSize();
  right = new OBTInt();
  OBTInt copyResult = new OBTInt();
  result.shallowCopyTo(copyResult);
  prune.shallowCopyTo(result);
  result.append(copyResult);
} // else if
```

OBTInt pruneLargestOnto()

```
    else // (right.getSize() > pruneSize)
    {
      right.pruneLargestOnto(pruneSize, result);
      pruneSize = 0;
    } // else
  } // while
} // pruneLargestOnto
```

OBTInt pruneSmallestOnto()

```
private void pruneSmallestOnto(int pruneSize, OBTInt result)
{
  while (pruneSize > 0 && ! empty)
  {
    if (left.empty) // Then delete this value.
    {
      result.pushRight(value);
      empty = right.empty;
      value = right.value;
      left = right.left;
      right = right.right;
      pruneSize--;
    } // if
```

OBTInt pruneSmallestOnto()

```
else if (left.getSize() <= pruneSize)
// Delete all left.
{
  OBTInt prune = left;
  pruneSize -= prune.getSize();
  left = new OBTInt();
  OBTInt copyResult = new OBTInt();
  result.shallowCopyTo(copyResult);
  prune.shallowCopyTo(result);
  result.prepend(copyResult);
} // else if
```

OBTInt pruneSmallestOnto()

```
      else // (left.getSize() > pruneSize)
      {
        left.pruneSmallestOnto(pruneSize, result);
        pruneSize = 0;
      } // else
    } // while
  } // pruneSmallestOnto
```

Tree demo:  random-balanceBs

Tree demo:  balance-by-elements-and-rebuild