MATH1510 - Discrete Mathematics
Trees

University of Newcastle

UoN

Binary search trees

In computing, it is essential to keep track of the location of data, whether this is

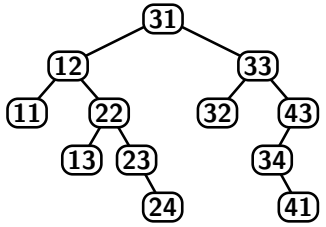
- in the working memory (RAM) of a running program,
- within a file on a disk,
- across several files in a filesystem, or
- in a database.

All of these require a flexible way of accessing and changing the data, and often being able to retrieve it based on some attribute to be looked up. (e.g. a file name, a record number, etc)
Trees provide several ways of organising data in a flexible, efficient ways. We will look at a particular example: Binary Search Trees.

Definition and Properties

If the data to be stored can be arranged in some natural order (such as numerical order or alphabetical order), then it can be stored in the nodes of a Binary Search Tree (BST) for easy access.
To be a binary search tree, a binary tree must satisfy this condition:

For every internal vertex v, all the items in the left subtree of v must be less than the item stored in v, and all the items in the right subtree of v must be greater than the item stored in v.



Some questions

- Where is the smallest item stored in a BST?
- Where is the largest?
- How do you list the items in a BST?
- What does the preorder traversal of a BST look like?
- Inorder & Postorder?

Searching a BST

How do we locate an item in a BST? An algorithm in pseudocode:

```
function FindInTree(tree T, data X) {
  Set here to the root of T
  Repeat {
    If X = here, then stop, data X is found
    If X < here, then
      if leftchild of here exists, then
        set here to leftchild of here
      else stop, data X not found
    If X > here, then
      if rightchild of here exists, then
        set here to rightchild of here
      else stop, data X not found
    }
  }
```


Inserting into a BST

The above algorithm can be modified to give

```
function InsertInTree(tree T, data X)
```

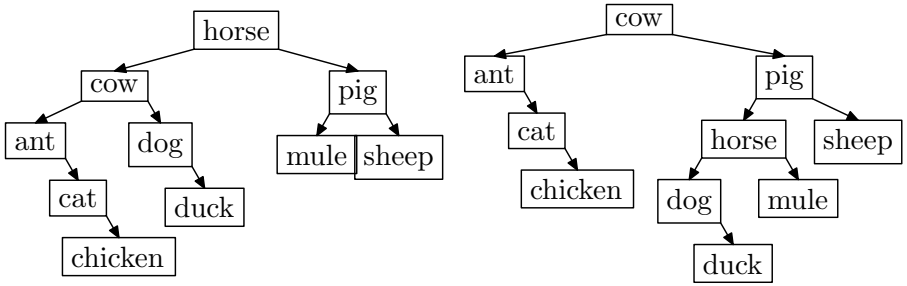
Example

Create a BST holding the following items of data, ordered alphabetically: horse, cow, dog, pig, ant, cat, mule, chicken, duck, sheep

Example

Create a new BST for the same data, inserting them in the following order: cow, pig, horse, ant, dog, cat, mule, chicken, sheep, duck

Examples



Keys

Sometimes the data doesn't have a natural order arrangement, so we need to impose one.

An item of data usually is a record, which is made up of a number of fields, each containing some information. We can simply add another field to the record, called the Key, which contains a number. We then identify the record by its key-value and use the key to order the records. (This is common in databases.)

Efficiency

Question: Is a binary search tree always an efficient way to store data?
Not necessarily - it depends on the shape of the tree.
For a tree containing n vertices (records)

- it may take as many as n steps to locate a record.
- it may take as few as 1
- we may try to minimise the average access time
- we may try to minimise the maximum access time.

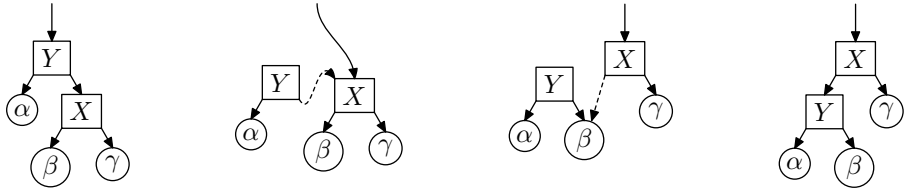
Balancing

A BST is less efficient when it has some very long branches, so it is useful to be able to modify the tree to shorten its longest branches, and generally balance things up. The standard way to do this is called rotation.

The following implements Right-to-left rotation at a vertex Y that has a right child.

function RotateRightToLeft(tree T, vertex Y)

if Y has no right child **then Fail**
else
Set X to the right child of Y
Remove Y from T and put X in its place
Set the right child of Y to the left-child of X
Set the left-child of X to vertex Y



Right-to-left rotation can be done at any node in the tree with a right-child, including the root.

Left-to-right rotation is the ǝɔuɐɥɔ ɹoɹɹıW.

Rotation preserves the BST conditions

before after

- InOrder traversal: $\alpha Y \beta X \gamma$

What is a balanced tree?

Now the question is how to choose when and where to rotate, and in which direction.

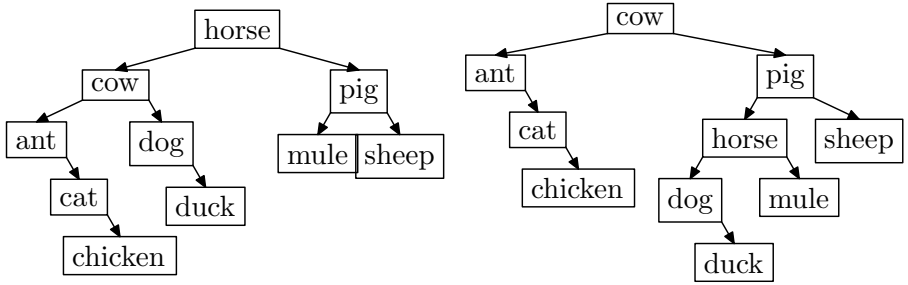
Clearly, some benefit may be obtained by rotating in the direction from longer branch to shorter branch.

The main thing we need is a criterion for deciding when a tree is balanced.

Definition

A BST is called balanced if for every vertex v, the height of the left and the right subtree differ by at most 1.

Example: Balance the animal trees



Balancing on-the-fly

A useful strategy is to do the balancing along the way, as the trees are built. This is a good idea, as the building of the tree involves searching the tree each time we want to insert a node. This will be more efficient in a balanced tree.

Example

Construct the animal trees again, ensuring the trees are balanced after each insertion.

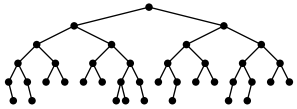
Maximal efficiency

It is worth considering other criteria for a tree to be "balanced". One guide is to ask for the most efficient searching: What is the best possible shape for a BST?

Having some long branches and some short branches makes the *average* time to locate a record longer. We want a tree with all terminal vertices at the same level or as near as possible to the same level.

Complete binary trees

For any n, a complete binary tree with n vertices, has all terminal vertices are on a single level or on two adjacent levels. Then search is as efficient as possible: finding a location in the tree requires as most $\lfloor \log_2(n) \rfloor$ steps, and on average about $\lfloor \log_2(n) \rfloor - 1$ steps.



BUT, balancing a tree to a complete binary tree requires extra work. Insertion of a single vertex can require a large number of vertices to be shifted around the tree (in the worst case, every vertex).

Complete binary trees

The decision of which criterion to use for "balanced" for a given tree is a trade-off between

- the work required to detect when a tree is unbalanced and to then rebalance it (worse if tree changes a lot or requires complicated rebalancing)
- the overhead in working with an unbalanced tree (worse if there are long branches to search along)

Other types of trees

Binary Search Trees are not the only way of using an ordered tree to store or access data. Other types of ordered trees are used, and each has its own characteristics that make it suitable for a given application.

Each type of tree has its own set of algorithms for insertion, deletion and balancing.

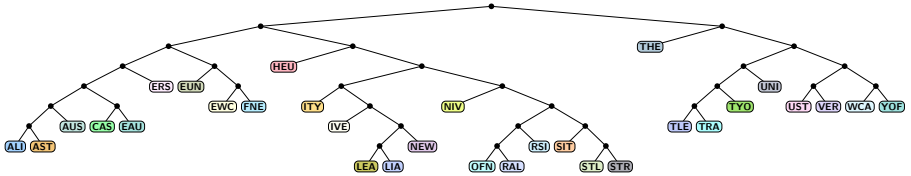
The examples given here are two that can occur.

Binary trees

Binary trees can have data stored only at terminal vertices (like Huffman codes). In comparison with BST:

- The test at each vertex is a 2-way test (less/not less) rather than the BST 3-way test (less/equal/more). Faster per vertex.
- A deeper tree, so more vertices must be retrieved and examined to find the one you want. More vertices per search.

This will be faster when it is quick to retrieve and examine many small pieces of data (e.g. low-latency storage like RAM)



B-trees

B-trees are ordered trees where there can be many children at each node, and with data stored at terminal vertices. The number of children per vertex has a fixed maximum, giving *ternary*, *quaternary*, *n-ary* B-trees. In comparison with a BST:

- A more complex *n*-way test at each vertex. Slower per vertex.
- A broader and shallower tree. Fewer vertices per search.

This will be faster when it is faster to retrieve fewer, larger chunks of data (e.g. high-latency storage like HDD, CD-ROM & network)