14 Binary Search Trees

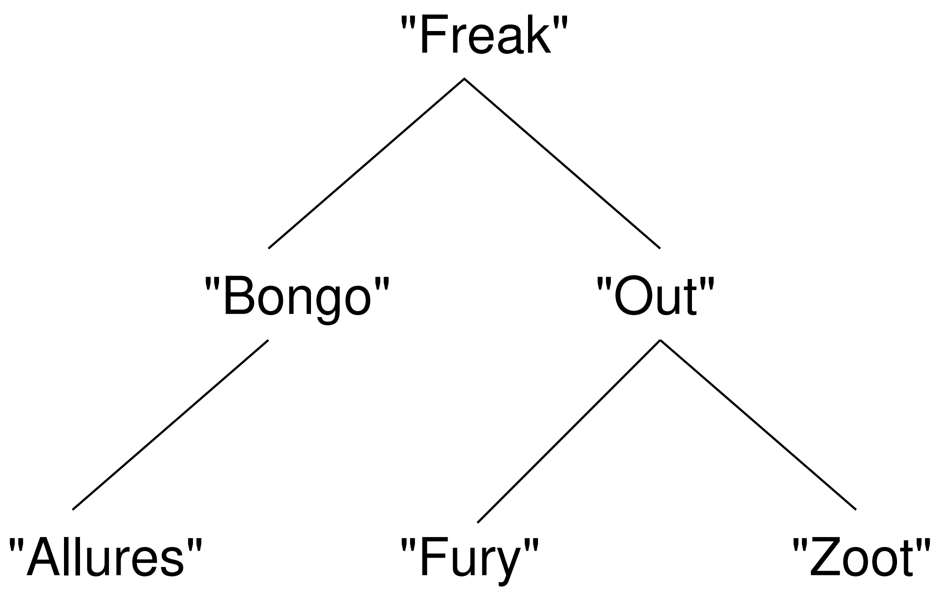
Binary search trees are *data structures* that correspond to the binary search *algorithm* discussed in the previous section.

14.1 Trees

A *TREE* is a recursive data structure that is either:

- *empty*
- an object with a *root* and an ordered set of branches (or subtrees), each of which is a *TREE*.

Trees are usually drawn upside-down:



Examples:

- Family trees.
- Tournament draws:

$$\left.\begin{array}{l}\left.\begin{array}{l}\text{player 1}\\ \text{player 2}\end{array}\right\}\text{winner 1, 2}\\ \left.\begin{array}{l}\text{player 3}\\ \text{player 4}\end{array}\right\}\text{winner 3, 4}\end{array}\right\}\text{winner 1, 2, 3, 4}$$

  note the regularity of this tree — it is a *binary* tree.

- Arithmetic expressions, programs, any “nested” object.

Terminology

The terminology of trees is mainly derived from the language of family trees, e.g.:

parent, **child**, **sibling**, **ancestor**, **descendant**, . . .

A leaf is a node with no children, e.g. “Allures”, “Fury”, “Zoot”

The height of a tree is the length of the longest path from the root down to a leaf.

The size of a tree is the total number of nodes in the tree.

Binary Trees

Binary trees are those where every node has at most two children.

Hence a binary tree is either:

- *empty*
- an object with a *root* and a *left subtree* (which is a binary tree), and a *right subtree* (which is a binary tree).

Representing Binary Trees in Haskell

Haskell's algebraic types allow us to directly represent binary trees:

```
> data BTree = Empty | Tree BTree Int BTree
```

The node values in this case are of type `Int`.

`BTree` is a new type built by constructor functions `Empty` and `Tree`:

```
? :i Empty
Empty :: BTree
  -- data constructor
? :i Tree
Tree :: BTree -> Int -> BTree -> BTree
  -- data constructor
```

Hence, a `BTree` value is either `Empty` or of the form `(Tree left n right)` where `left` and `right` are `BTree`s and `n` is an `Int`.

Notice how the *definition* of `BTree` is recursive.

Algorithms on binary trees are generally recursive, following the structure of the tree.

It is possible, and more general, to give a definition of *polymorphic* binary trees:

```
> data BinTree a = Empty
>                | Tree (BinTree a) a (BinTree a)
```

In that case we have:

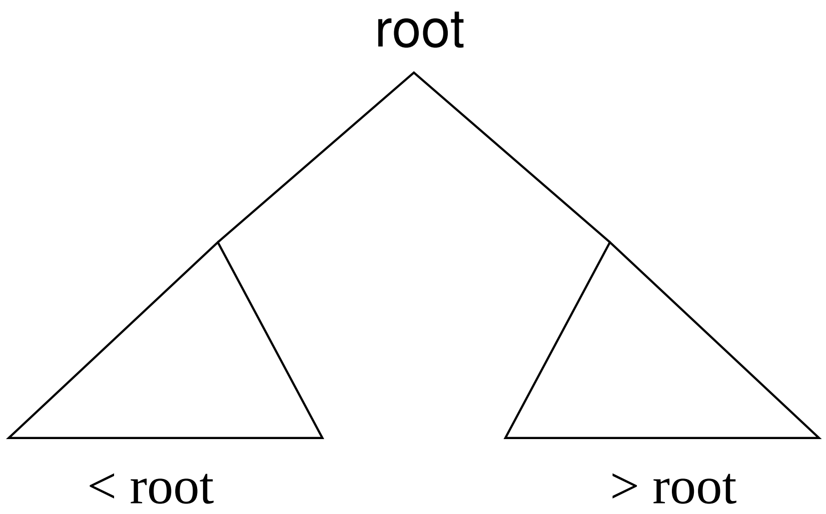
```
> type BTree = BinTree Int
```

14.2 Binary *Search* Trees

Binary search trees are just binary trees where the data values are arranged according to an ordering relation, $<$, $==$, $>$.

The rule is simple:

- *All* values in the *left* subtree are *less* than the root value.
- *All* values in the *right* subtree are *greater* than the root value.
- *This rule also applies recursively to all subtrees.*



The tree of words on the first slide is a binary search tree.

Connection with Binary Search Algorithm

Recall that binary search of a list proceeds by comparing the value being sought with the *middle element* of the list, thus restricting the range of values to *half* the list.

Searching a binary search tree proceds by comparing the sought value with the *root* of the tree, thus restricting the range of values to either the left or right subtree — approximately *half* the values.

Why should we prefer binary search trees over binary search of an ordered list?

Generally, the main operations on search tables are **lookup** an item, **insert** a new item (in the right place), **delete** an item.

Without going into details, binary search trees perform well on *all* these operations while linear representations do not.

Suppose the table contains N items (some large number N).

Binary search tree:
lookup, insertion and deletion all require about $\log_2 N$ `operations' (e.g. key comparisons).

Binary search:
A linear, direct-access table using binary search requires about $\log_2 N$ operations for lookup, but insertion and deletion both require about $N/2$ operations.

For large N (say 1000000) this difference is very significant.

In general, binary search trees are more efficient than any linear table organisation.

Algorithm to Search a BST

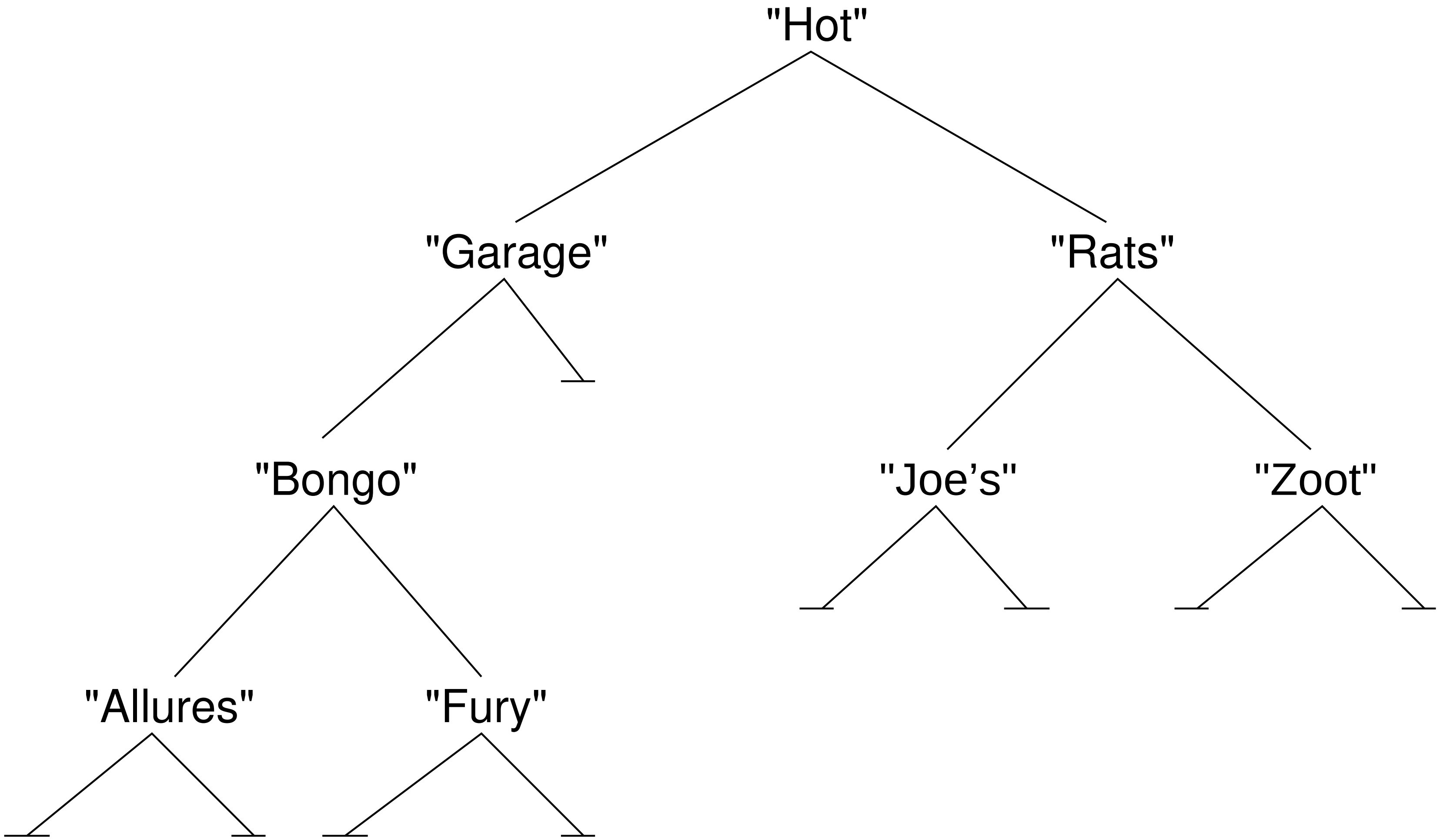
- Compare the search key with the root.
  - If they are equal, the item is found.
  - If key $>$ root, search the right subtree (recursively).
  - If key $<$ root, search the left subtree (recursively).
- If the tree is empty then item not found.

Examples:

Search for “Fury”, “Rats” and “Motels” in the BST on the next slide.

The binary search tree was built by inserting, in the following order:

“Hot” – “Rats” – “Joe's” – “Garage” – “Bongo” – “Fury” – “Zoot” – “Allures”


"Hot"
"Garage"
"Rats"
"Bongo"
"Joe's"
"Zoot"
"Allures"
"Fury"

Haskell implementation of BST search algorithm

For this example, the function simply returns `True` or `False` indicating whether the search key is in the BSTree.

```
> data BSTree = Empty
>             | Tree BSTree Word BSTree

> type Word = String

> bstFind :: Word -> BSTree -> Bool

> bstFind val Empty = False
> bstFind val (Tree lesser root greater)
>    | val == root  = True
>    | val < root   = bstFind val lesser
>    | val > root   = bstFind val greater
```

More on BST search

Finding the *minimum* value in a BSTree:

Idea:
If a BSTree has a *left subtree*, the minimum is in that subtree.
If that left subtree has a left subtree, the minimum is in that subtree.
And so on . . .

If a BSTree has *no* left subtree, the minimum value must be the *root*.

This describes an algorithm: recursively take the left (i.e. lesser) subtree until it is `Empty`. Then return the root.

Note that the algorithm must stop when the *left subtree* is empty, not when the tree is empty.

Finding the *maximum* is analogous, taking the right (i.e. greater) subtree.

```
> least, greatest :: BSTree -> Word

> least Empty
>     = error "Empty tree -- no minimum"
> least (Tree Empty root greater)
>     = root
> least (Tree lesser root greater)
>     = least lesser

> greatest Empty
>     = error "Empty tree -- no maximum"
> greatest (Tree lesser root Empty)
>     = root
> greatest (Tree lesser root greater)
>     = greatest greater
```

BSTree Insertion

The second basic operation on search tables is *insertion of a new item*.

This action must maintain the rule of organisation of the table. That is, a sorted list must still be sorted; a binary search tree must still be a binary search tree.

Insertion is thus a two-step process:

1. Find a correct position for the new item;
2. Insert the item

Finding the position is similar to lookup *except* we do not expect to find the item.

So when do we stop the search?

Observation 1: It is easy to add a new node as a *leaf* but not so easy to add a new internal node.

Observation 2: It is *ALWAYS* possible to add a new node as a leaf and still maintain the relative order of the binary search tree organisation.

BSTree Insertion Algorithm

Corresponding to the two-step process above, based on observations 1 & 2:

1. “Search” for the item being inserted until the search fails. This will occur when a *vacant leaf position* is found.
2. Attach the new item as a leaf at that position.

This algorithm can be expressed to correspond to the recursive data structure:

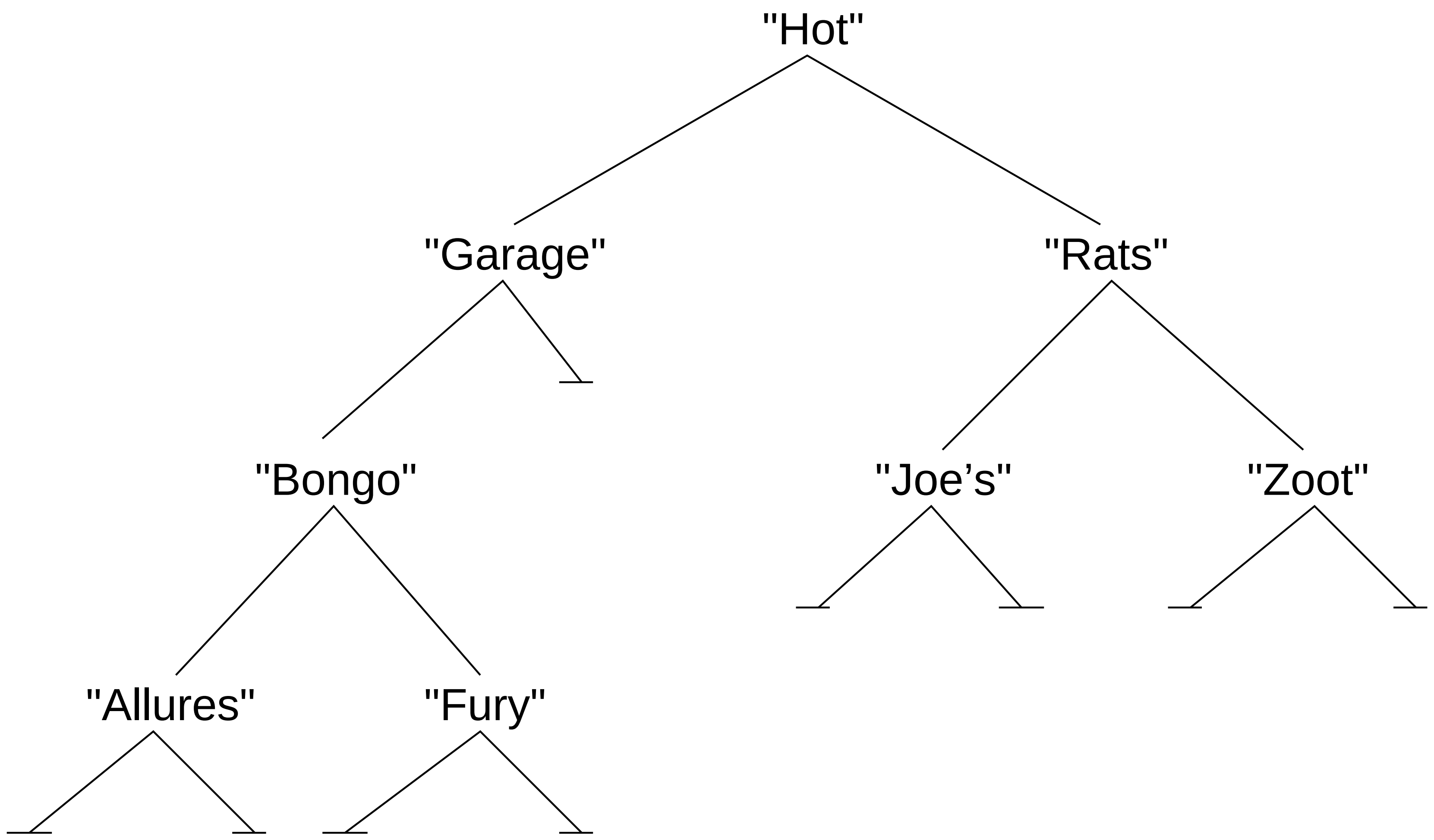
- If we have an *empty* BSTree, place the new item at the root.
- Otherwise, insert (recursive) the new item into the *left* subtree if it is *less* than the root; or, insert the new item into the *right* subtree if it is *greater* than the root.

Haskell implementation of BSTree insertion algorithm

```
> bstInsert :: Word -> BSTree -> BSTree

> bstInsert val Empty
>     = Tree Empty val Empty
> bstInsert val (Tree lesser root greater)
>   | val < root
>     = Tree (bstInsert val lesser) root greater
>   | val > root
>     = Tree lesser root (bstInsert val greater)
>   | val == root
>     = error " value already in search tree"
```

Exercise:
Insert each of the following items into the BSTree in the following slide: “Yellow”, “Shark”, “Overnite”, “Sensation”.


"Hot"
"Garage"
"Rats"
"Bongo"
"Joe's"
"Zoot"
"Allures"
"Fury"

An obvious thing we might want to do is build a binary search tree from a list of items:

```
> buildBST :: [Word] -> BSTree
> buildBST []
>     = Empty
> buildBST (x:xs)
>     = bstInsert x (buildBST xs)
```

Alternatively, we could use one of those nifty fold functions:

```
> buildBST' = foldr bstInsert Empty
```

The above two version insert the items in reverse order of the list. The following version inserts them first to last.

```
> buildBST'' = foldl (flip bstInsert) Empty
```

Exercise:

Evaluate the following expressions:

```
buildBST ["c","f","e","h","a","g","b","d"]

buildBST ["d","f","g","e","a","h","b","c"]

buildBST ["a","b","c","d","e","f","g","h"]

buildBST ["e","f","g","h","a","b","c","d"]
```

Observation:

Different binary search trees can be constructed from the *same data values*.

Some arrangements lead to more efficient searching than others.

For example, a `degenerate' tree as produced by the third exercise above will take on average $N/2$ comparisons to find an item in a table of N entries.

The best possible shape is a *balanced* tree where as many nodes as possible have 2 children. In that case the height of the tree is $\lceil log_2 N \rceil$ so that is the maximum number of comparisons that will be required.

There are schemes where insertion and deletion are more complicated than the algorithms that we have seen but guarantee that trees will always be nearly balanced. (AVL trees.)

Very pretty but too advanced for this unit.

14.3 Some Simple Algorithms on Binary Trees.

The following common and straightforward algorithms on binary trees (not just *search* trees) are all variants on the notion of ***traversal*** — `visiting' every node in a tree.

The hidden agenda is just to give some more examples. Note that they can be useful for testing your own tree-processing scripts.

1. Height of a Binary Tree

The height of a binary tree is 1 more than the maximum of the heights of the left and right subtrees.

Note the naturally recursive definition.

Note that `height` `visits' each node in the tree exactly once.

```
> height :: BinTree -> Int
> height Empty
>   = 0
> height (Tree left root right)
>   = 1 + height left `max` height right
```

2. Size of a Binary Tree

The size of a tree is the number of nodes in the tree.

For a binary tree, that is one more than the sum of the sizes of the left and right subtrees.

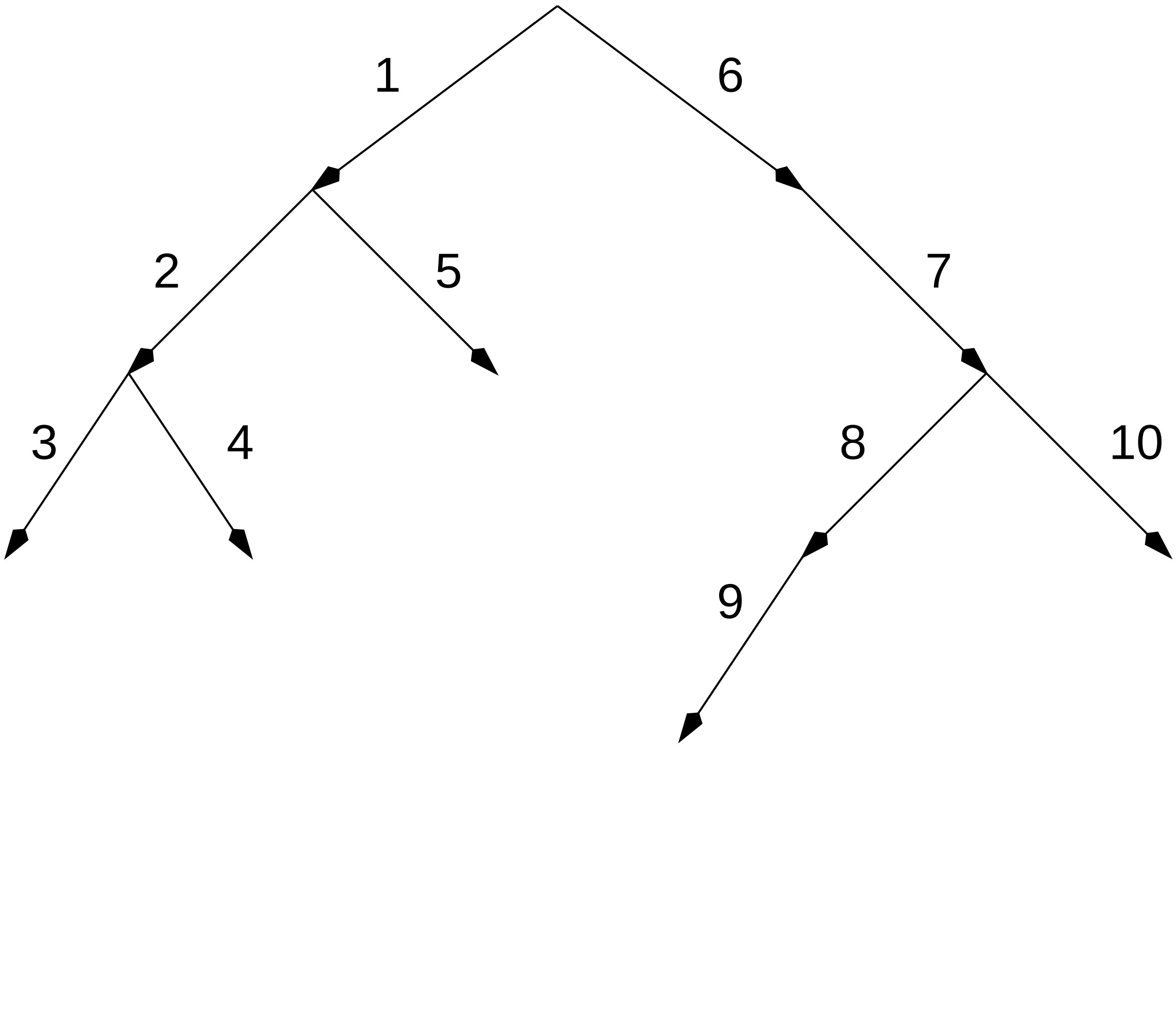
Note the naturally recursive definition.

Note that `size` `visits' each node in the tree exactly once.

```
> size :: BinTree a -> Int
> size Empty
>   = 0
> size (Tree left root right)
>   = 1 + size left + size right
```

Notice that `size` and `height` both traverse the tree *depth-first*.

That is, the algorithm passes all the way down to a leaf before moving on to the next subtree.



3. Inorder, Preorder, Postorder

Within depth-first traversal we have 3 basic choices:

Pre-order: process each node *before* its subtrees.

Post-order: process each node *after* its subtrees.

In-order: traverse the left subtree, then process the node, then traverse the right subtree.

It so happens that in-order traversal of a binary *search* tree visits the nodes in *ascending order*.

The following algorithm produces a list of the values in a BSTree in ascending order.

```
> inOrder :: BSTree -> [Word]
> inOrder Empty = []
> inOrder (Tree lesser root greater)
>   = inOrder lesser ++
>     [root] ++
>     inOrder greater
```

Exercise:
Combine the functions `buildBST` and `inOrder` to construct a *sort* function. How efficient do you think this technique is compared to the insertion sort approach?

Exercise:
Write a binary tree version of the `map` function:

```
> mapBTree :: (a -> b) -> BinTree a -> BinTree b
```

14.4 Binary Search Tree Deletion

This section is not examinable. It is included for general interest and completeness.

There are two phases to delete an item from a BSTree:

1. find the item
2. delete it

Unlike insertion, the first step is *exactly* like the search algorithm.

Unlike insertion (unfortunately), we are not always dealing with *leaves*. The node to be deleted may be *internal*.

there are 3 cases to consider:

(i) The node has *0* subtrees (i.e. it is a leaf);

(ii) The node has *1* subtree;

(iii) The node has *2* subtrees.

For example:

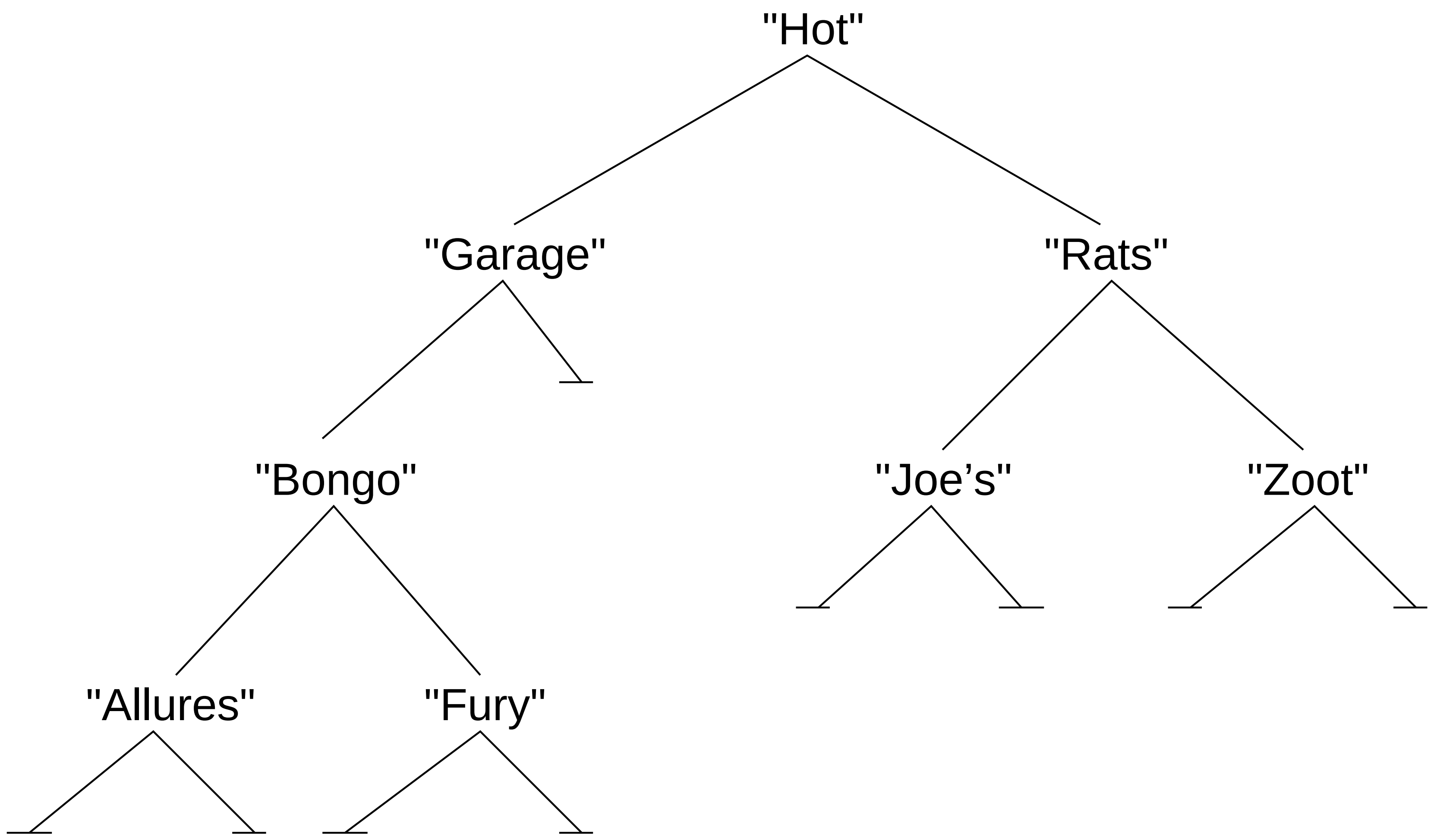
(i) Delete `"Joe's"` from the BSTree.
Easy — set the left subtree of `"Rats"` to `Empty`.

(ii) Delete `"Garage"` from the BSTree.
Easy again — replace the tree rooted at `"Garage"` by its only subtree. The ordering arrangement is maintained.

(iii) Delete `"Hot"` from the BSTree.
Not so easy — how to reconstruct a BSTree from the 3 resulting subtrees?

Solution: *Replace* the deleted item with the *maximum of the lesser subtree* (or the minimum of the greater subtree). This restores the required order.

That means we need to delete the maximum value. However, that node must have only 0 or 1 subtree so it will be easy to delete as in (i) or (ii).


"Hot"
"Garage"
"Rats"
"Bongo"
"Joe's"
"Zoot"
"Allures"
"Fury"

Here is the Haskell implementation of the deletion algorithm:

```
> bstDelete :: Word -> BSTree -> BSTree

> bstDelete val Empty
>     = Empty
> bstDelete val (Tree lesser root greater)
>   | val < root
>     = Tree (bstDelete val lesser) root greater
>   | val > root
>     = Tree lesser root (bstDelete val greater)
>   | val == root
>     = rebuild lesser greater
>   where
>   rebuild Empty greater
>       = greater
>   rebuild lesser Empty
>       = lesser
>   rebuild lesser greater
>       = Tree lesser' nextBiggest greater
>     where
>     nextBiggest = greatest lesser
>     lesser'     = bstDelete nextBiggest lesser
```