- The deadlines for labs 1 - 6 code reviews:
  - Group 1: Friday 15 March
  - Group 2: Monday 18 March

- Friday 8th, group 1
  - Lab 8
  - Lab 7 deadline
  - Lab 6 late submission
- Monday 11th, group 2
  - Lab 8
  - Lab 7 deadline
  - Lab 6 late submission

Software Development 3 (F27SG)

Lecture 15

Tries & Non-Binary Trees

Rob Stewart

Overview

- We’ve covered Binary Trees
  – 2 subtrees
  – Looked in depth at **search**

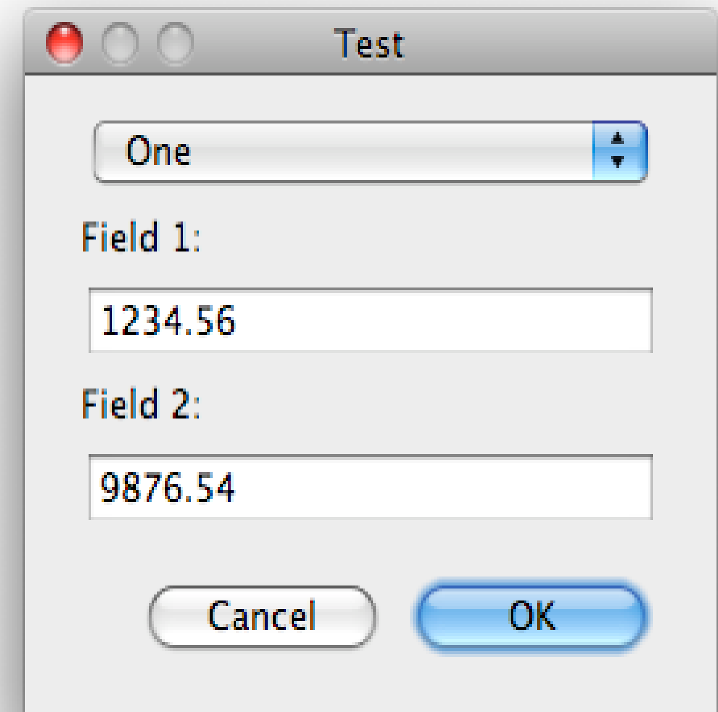
- This lecture
  – trees with **more than 2 subtrees**
  – look in-depth at **Tries**
    - pronounced *trys*
  – a practical application of such trees

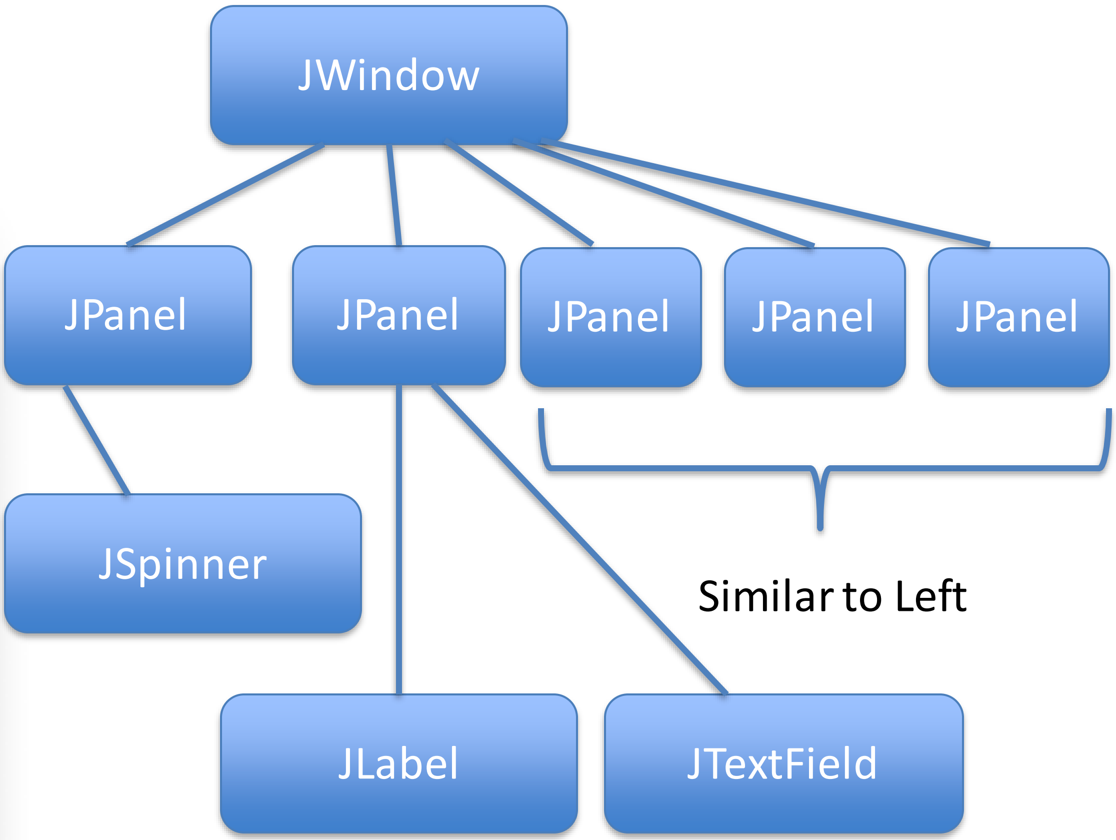
Beyond Binary Trees

- Many variants of trees

- **K-ary Trees**
  - each node can have **K** (more than 2) subtrees
  - e.g. to model window hierarchy of a GUI
- **B-Trees, B+-Trees, B* Trees**
  - efficiently handle disk paging and database indices
- **Tries**
  - Cover these today

K-ary GUI

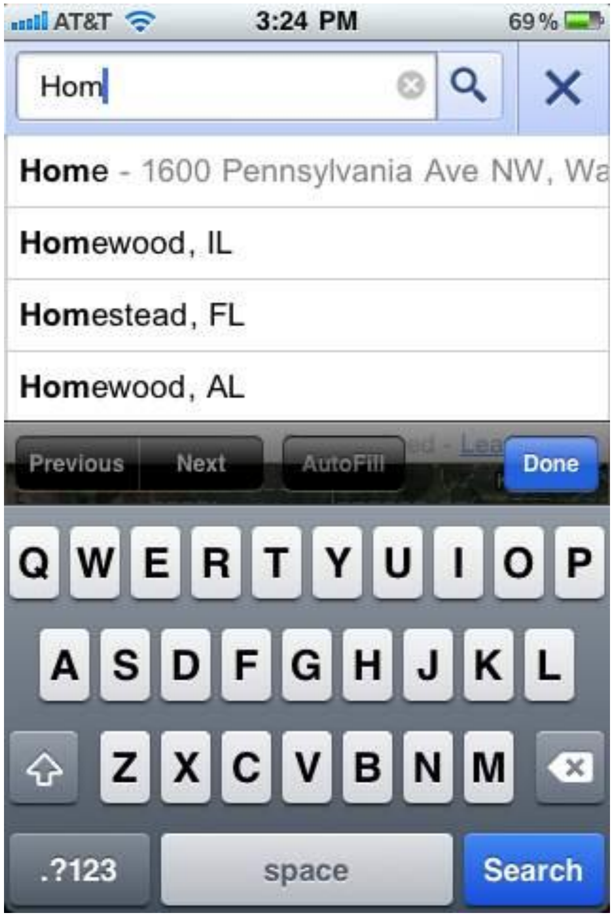

Test
One
Field 1:
1234.56
Field 2:
9876.54
Cancel
OK


JWindow
JPanel
JPanel
JPanel
JPanel
JPanel
JSpinner
Similar to Left
JLabel
JTextField

K-ary Trees for GUIs

- GUIs are hierarchical
- In Java, each component is represented by a class
  - **JContainer**: generic container to hold things
  - **JScrollPanel**: container that shows viewport onto some larger component
  - **JButton**: interactive control that can be pressed
- Components are added to each other
- JVM must know components a component contains (e.g. for redrawing)
- Can use a K-ary Tree

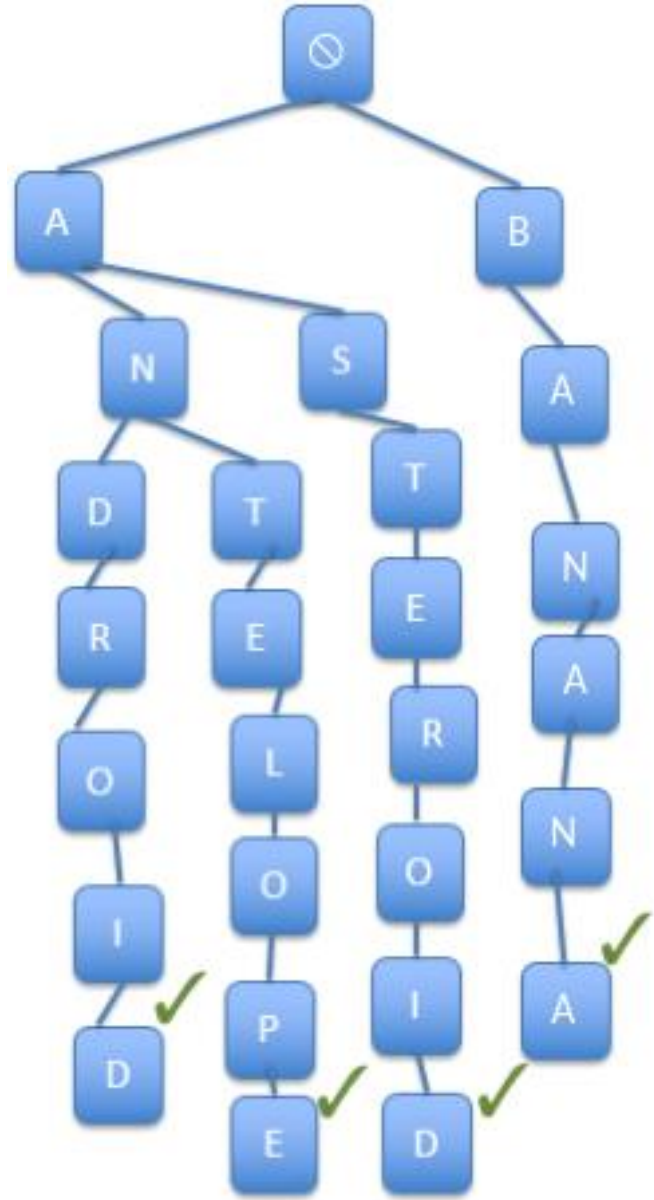
Real World Example



- Mobile devices offer autocomplete/autocorrect
- How do they do this?
  - Efficiently
  - Correctly (mostly)
- Let's look at a basic version

Tries

- A form of n-ary tree (pronounced *try*)
- Efficient way to store a dictionary
- Each level stores a character position
- *Nth* level stores the *nth* character of word
- A word is valid if
  1. Each character in word appears at correct level of tree
  2. Node containing final character is either:
     - a leaf
     - Marked as valid word
  3. That node is **marked** as a **valid word**
- Lookup O(1) worst case
  – where N is number of words



Exercise

- Insert the following words into an empty Trie:
  - Catch
  - Catcher
  - Bedridden
  - Bed
  - Animal

https://www.cs.usfca.edu/~galles/visualization/Trie.html

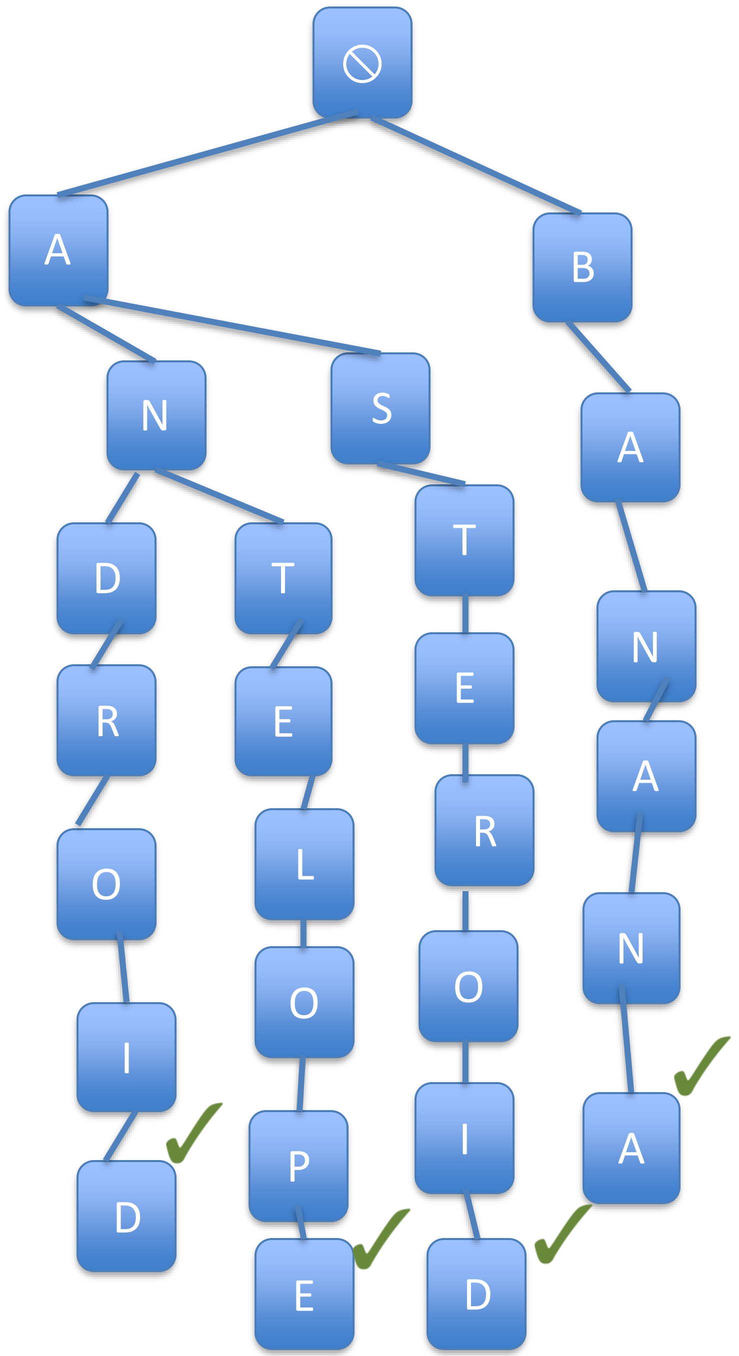
Exercise

- Insert the following words into an empty Trie:
  - Antelope
  - Android
  - Banana
  - Asteroid

https://www.cs.usfca.edu/~galles/visualization/Trie.html

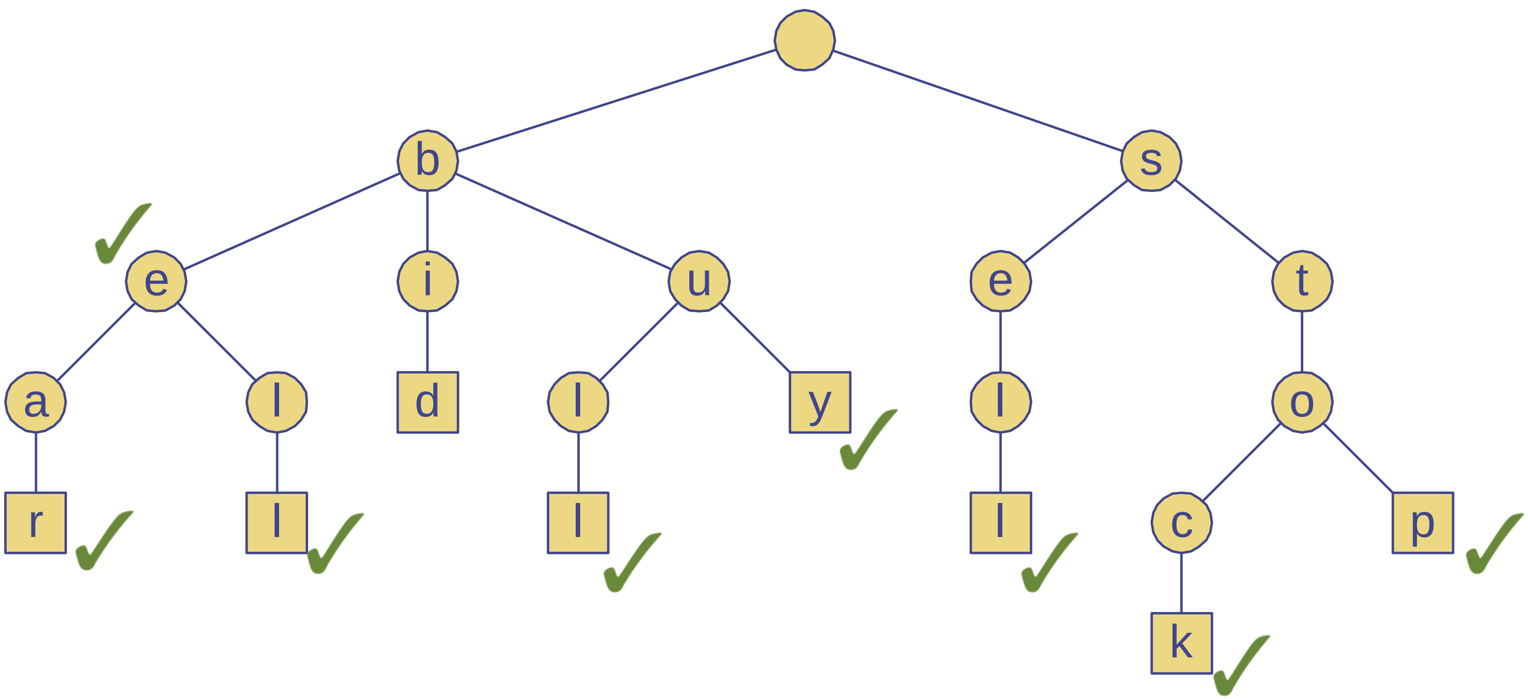
Tries

- A Trie Containing
  - Antelope
  - Android
  - Banana
  - Asteroid
- Note the root node
- End word indicated by ✓



Search for Some Words

- bear
- be



Implementing Tries

- For Linked Lists and Binary Trees
  - **outer** class with trie operations
  - **inner** Node class
- Trie operations
  - Insert word
  - Search for word
  - Remote a word
  - Return list of words with common prefix
    - Auto complete feature
- For tries, ***the nodes are different*** …

Eclipse demo: Trie node class

Trie Node

Nodes different from Linked lists/Binary trees:

```
public class Trie {
 class TrieNode{
   private char value; // the character contained at this node
   private TrieNode[] subnodes; //the subtrees that stem from this node
   private boolean isValidEnd = false; //We need to know if this node is a valid word

   public TrieNode(char c){
        value = c;
        isValidEnd = false;
        subnodes = new TrieNode[26];
        for(int i =0; i < 26; i++){ //initialise each node to null
              subnodes[i] = null;
   }
   ...
 }
 private TrieNode rootNode = null;
 …
```

ASCII character values

a – 97	h – 104	o – 111	v – 118
b - 98	i – 105	p – 112	w – 119
c – 99	j – 106	q – 113	x – 120
d – 100	k – 107	r – 114	y – 121
e – 101	l – 108	s – 115	z - 122
f – 102	m – 109	t – 116	
g – 103	n - 110	u - 117	

```
String s = new String("ab");
int value1 = (int) s.codePointAt(0);
int value2 = (int) s.codePointAt(1);
```

value1 == 97
value2 == 98

ASCII tree subnodes

```
public TrieNode(char c, boolean isEnd){
        value = c;
        isValidEnd = isEnd;
        subnodes = new TrieNode[26];
        for(int i =0; i < 26; i++){ //initialise each node to null
            subnodes[i] = null;
    }
```

subnodes:

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null

“bus” in a trie

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
null	TrieNode	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	TrieNode	null	null	null	null	null

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	TrieNode	null	null	null	null	null	null	null

isValidEnd == true

Search for a Word

- Traverse the tree until we reach the end of the string
  - Return the isValidEnd value
  - Return false if we run out of nodes

Eclipse demonstration

Search for a Word

- Traverse the tree until we reach the end of the string
  - Return the isValidEnd value
  - Return false if we run out of nodes

```
//In Trie
public boolean containsString(String s){
		return rootNode.containsWord(s.toLowerCase());
	}
//In TrieNode
 public boolean containsWord(String s){
	int positionOfNextNode = ((int)s.codePointAt(0)) - 97; //97 is 'a' in ASCII
	if(subnodes[positionOfNextNode] == null){
		return false; //we don't have the word
	}else{ //there are still more characters
		if(s.length()== 1){
			return subnodes[positionOfNextNode].isValidEnd;
		}else{
			return subnodes[positionOfNextNode].containsWord(s.substring(1));
		}
	}
}
```

Insert a Word

- Given a String insert it into the Trie
  - Recursively add each letter in until we get to the last node
  - Mark that node as a valid end point

Eclipse demonstration

Insert a Word

- Given a String insert it into the Trie
  - Recursively add each letter in until we get to the last node
  - Mark that node as a valid end point

```
//in Trie
public void insertString(String s){
        rootNode.addWord(s.toLowerCase());
 }
//in Trie Node
 public void addWord(String s){

         int positionOfNextNode = ((int)s.codePointAt(0)) - 97;  //97 is 'a' in ASCII

         if(subnodes[positionOfNextNode] == null) //add a new node for this value
                 subnodes[positionOfNextNode] = new TrieNode(s.charAt(0));

         if(s.length()== 1){ //if this is the last character
                 subnodes[positionOfNextNode].isValidEnd = true;
         }else{ //add the substring from 1 on to that node
                 subnodes[positionOfNextNode].addWord(s.substring(1));
         }
 }
```

Deleting Words

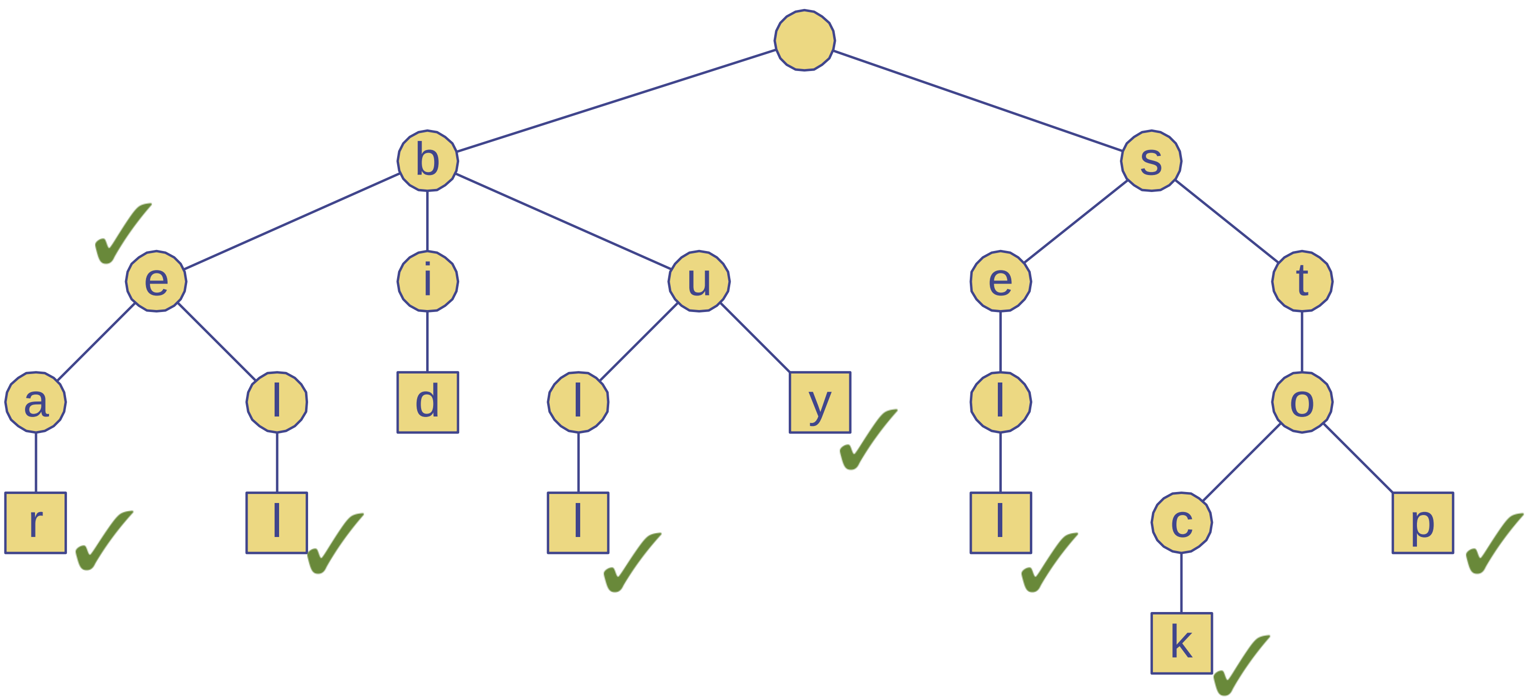
- Simple solution is
  - search then **set isValidEnd to false**
  - But that wastes space!
- Better solution is to delete nodes
  - But you need to be careful. E.g.
    - deleting *bedridden* should not delete *bed*
    - deleting *catch* should not remove *catcher*
- Solution though is actually easy

Deleting Words

- First establish the word is present
  - use the ***search*** method
- Instead of returning true if it is,
  1. set **isValidEnd** to **false**
  2. Each recursive call
     - returns true if the node it is called on can be deleted from the trie
- A node can be deleted if
  1. First case
     - is a **leaf node,**
     - **and isValidEnd == false**
  2. Second case
     - **isValidEnd == false,**
     - **and** it has one **subtree**,
     - **and** subtree returned **true** that it could be deleted
- *In these cases the node should return true*

Delete Some Words

- bear
- be



Deleting a Word

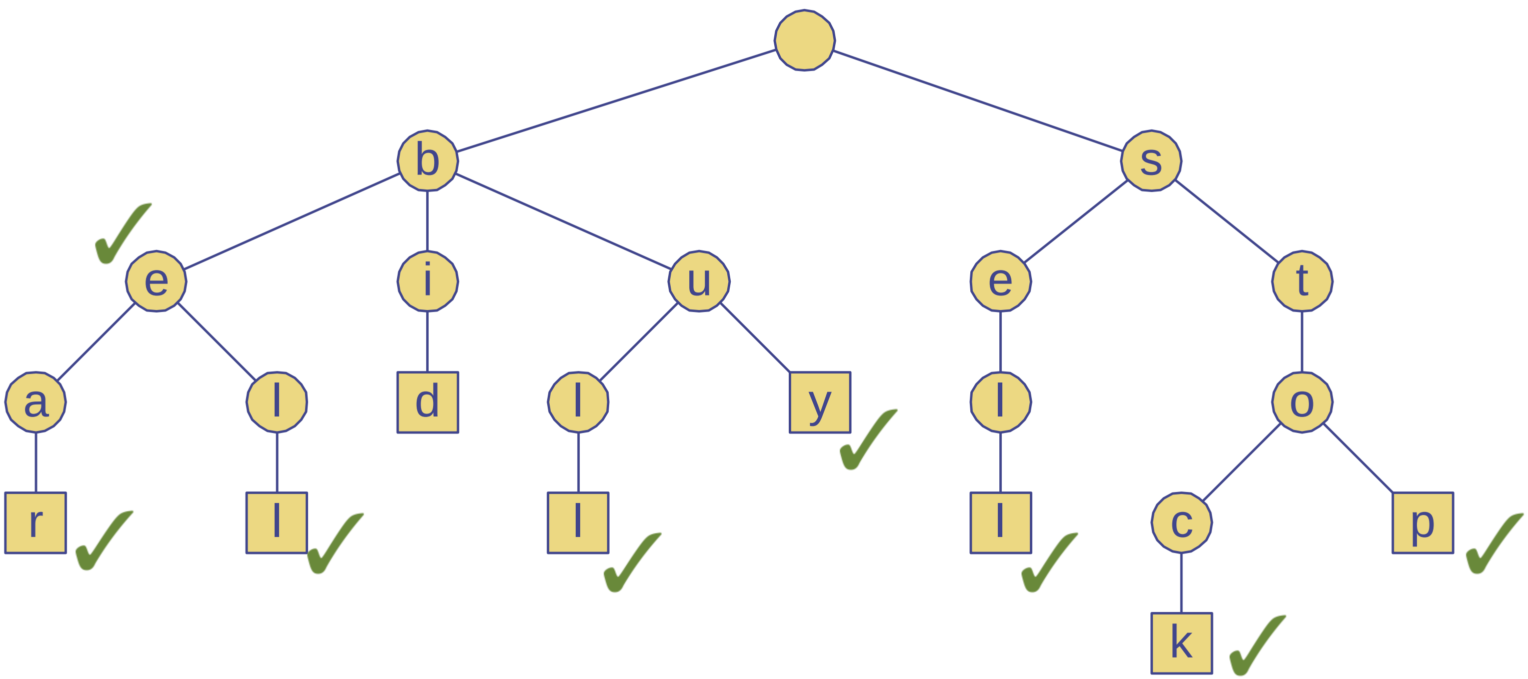
Eclipse demonstration

Return Words With Common Prefix

1. Find node containing last letter of the prefix
2. For each subtree recursively
   – return a list with all possible valid suffixes
   – add current node letter to front of the list
   – merge to one list
3. Add the prefix to each node of the list

Return Words With Common Prefix

- st
- be



Conclusion

- We've introduced **k-ary** trees
  - Trees with more than 2 subtrees
- Useful in many situations e.g.
  - Hierarchical GUIs
  - Predictive text
- **Tries**
  - Useful to store Dictionaries and search efficiently
  - **O(1)** for insert, search and delete
    - Where **N** is number of words in dictionary

- We are now finished with data structures

- Next topic: **sorting algorithms**