Ternary Tree

Bentley and Sedgewick (1998)

Introduction

- A ternary tree is a second version of a trie which can also be used to implement a hash table, among other applications.
- As a hash table, items are inserted, removed, and retrieved from the ternary tree based on <key,value> pairs.
- The ternary tree has a branching factor of three and therefore, can save some space over an r-way trie.

- Like an r-way trie, keys are examined one character at a time. But there is not a one-to-one correspondence between a character and a branch.

- Then how does one progress through the ternary tree?

General strategy to traverse the ternary tree *

Compare the current character k of the key with the character c at the current node.

Then

- Move left if $k < c$
- Move right if $k > c$
- Move to the middle and onto the next character if $k = c$

* Similar to traversing a binary search tree

Data structure

```
class Trie<T> : ITrie<T>
   {
       private Node root;                  // Root node of the Trie
       private int size;                   // Number of values in the Trie

       class Node
       {
           public char ch;                 // Character of the key
           public T value;                 // Value at Node; otherwise, default
           public Node low, middle, high; // Left, middle, and right subtrees
           ...
       }
   ...
   }
```

Key methods

bool Insert (string key, T value)

Insert a <key,value> pair and return true if successful; false otherwise

Duplicate keys are not permitted

bool Remove (string key)

Remove the <key,value> and return true if successful; false otherwise

Omitted

T Value (string key)

Return the value associated with the given key

Constructor

- The initial ternary tree is set to empty, i.e. the root is set to null

Insert

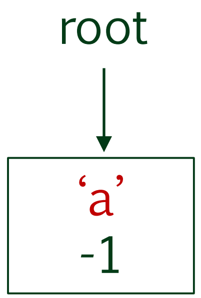
- Basic strategy (similar to the R-Way Trie)
  - Follow the path laid out by the key (described earlier), “breaking new ground and building a spine” if needed until the full key has been explored.
  - The value is then inserted at the final destination (node) unless the key has already been used (i.e. a value already exists for that key)

Insert <abba, 10>

root

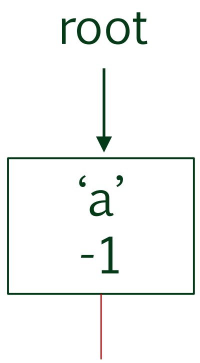
Because the root is null, create a node to store 'a'

Insert <abba, 10>



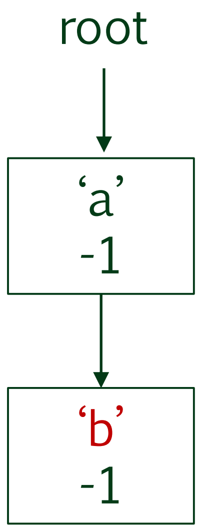
Because the key character and the character in the node
are the same, proceed down the middle path
with the next character in the key.
Note: Once a node is created for a character,
all subsequent nodes will be added along the middle branch,
creating a spine.

Insert <abba, 10>



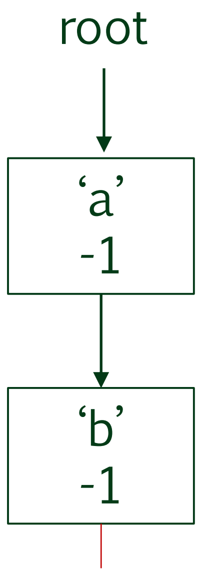
Because the middle reference is null, create a node to store ‘b’

Insert <abba, 10>

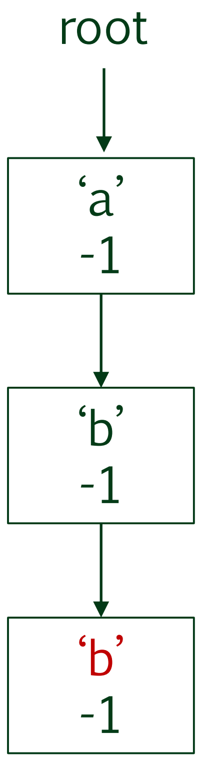


Because the key character and the character in the node
are the same, proceed down the middle path
with the next character in the key.

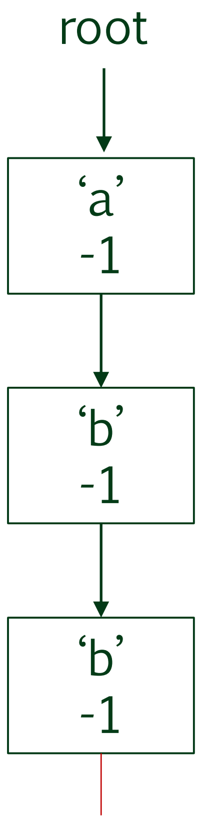
Insert <abba, 10>



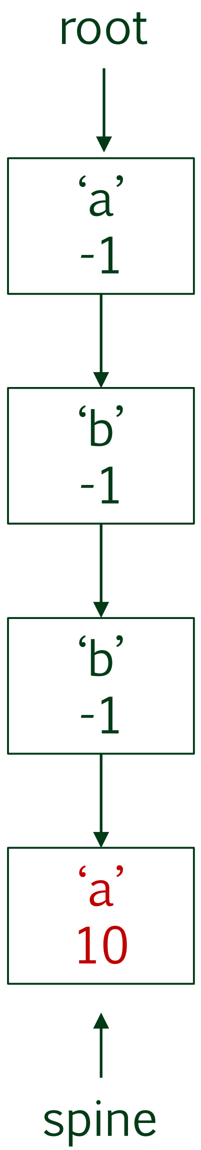
Insert <abba, 10>



Insert ⟨abba, 10⟩

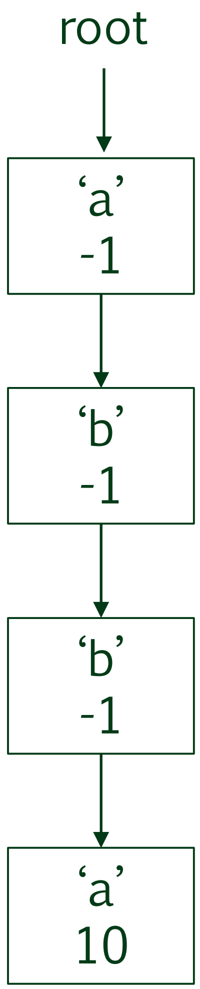


Insert <abba, 10>

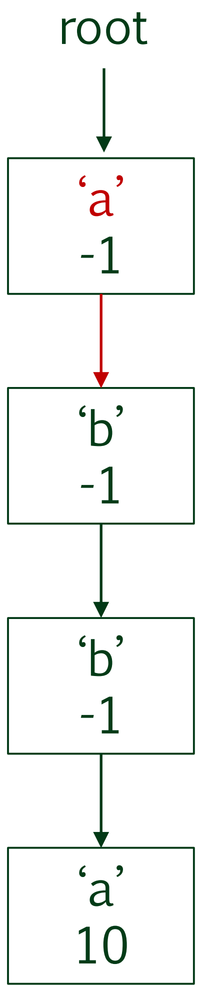


Now that all characters of the key have examined, the value 10 is added to the last node and true is returned

Insert ⟨ab, 20⟩

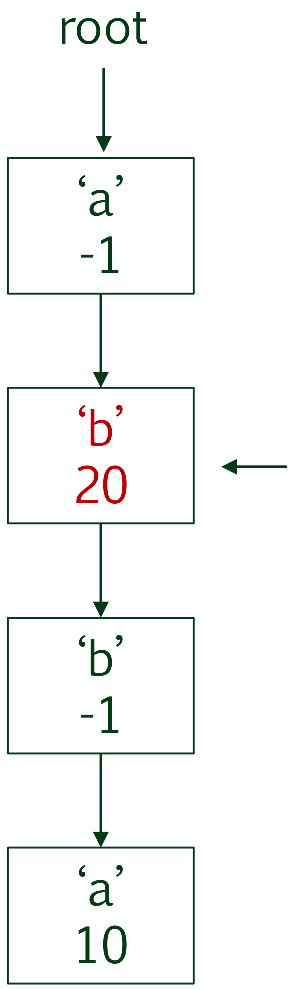


Insert <ab, 20>



Since the first character of the key is equal to ‘a’, proceed down the middle branch

Insert <ab, 20>



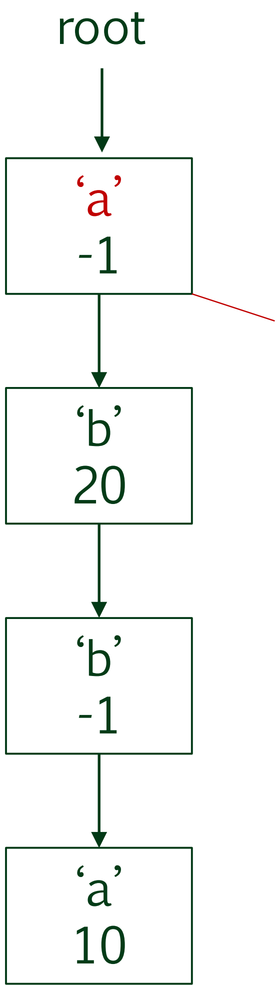
Since:
1) the second character of the key is equal to 'b'
2) all characters of the key have been examined, and
3) the node contains the default value

then the value 20 is added here and true is returned

Note: If a value was already found then false would have been returned

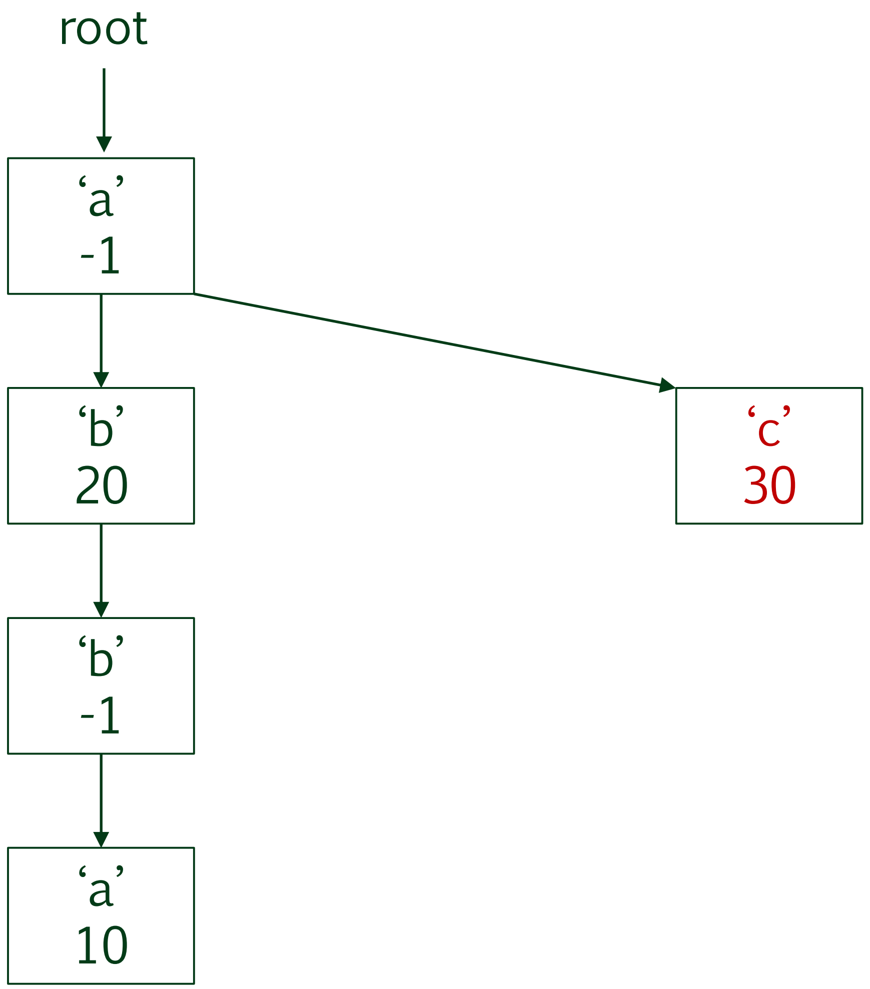
Insert <c, 30>

Since ‘c’ is greater than ‘a’
proceed to the right



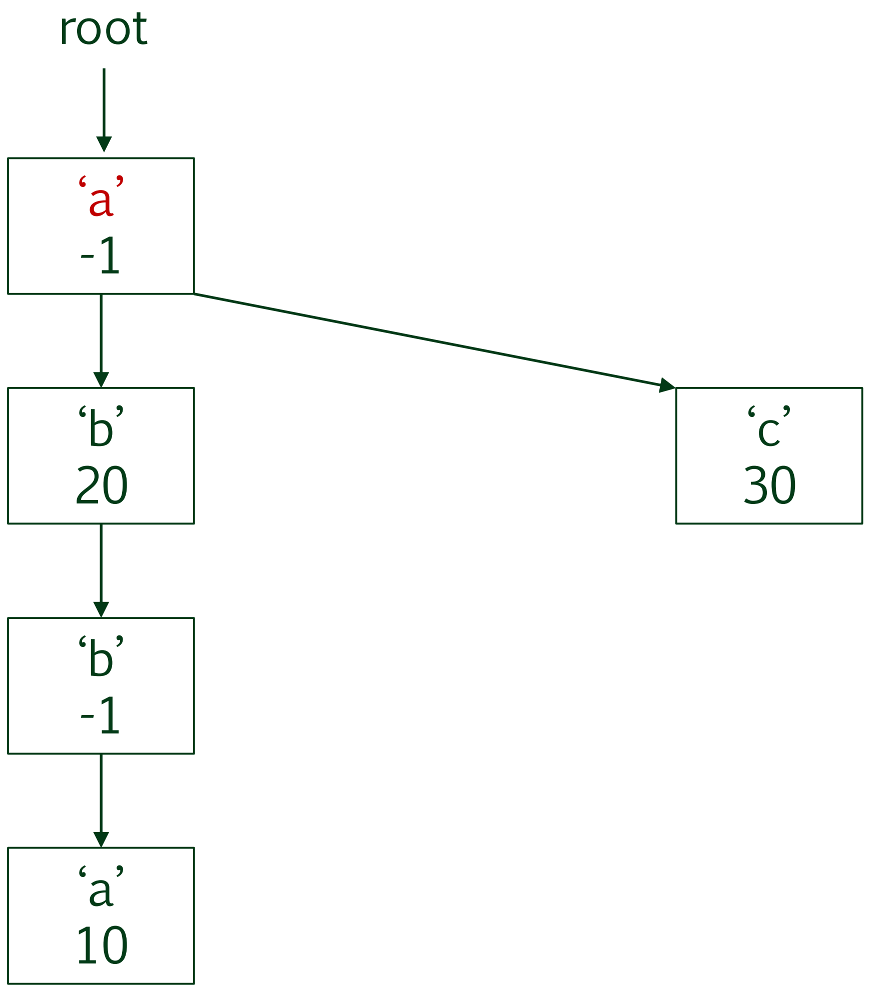
Because the right reference is null,
create a node to store ‘c’

Insert <c, 30>

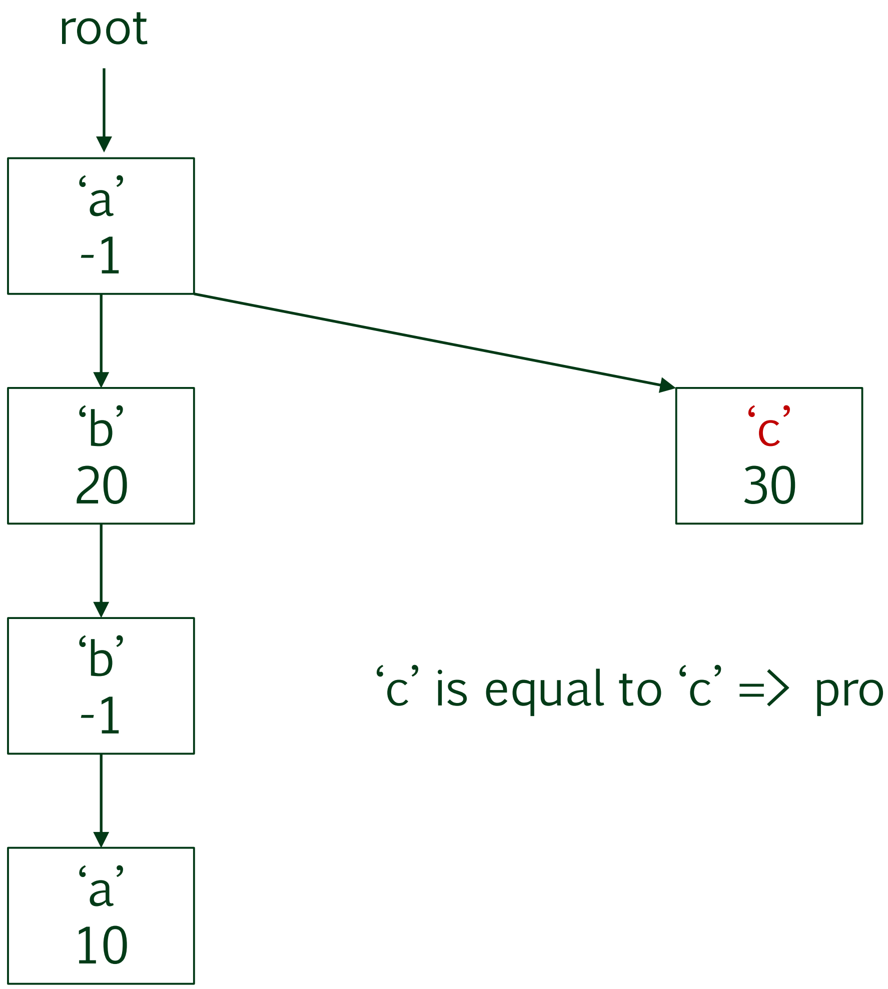


Insert <cbc, 30>

‘c’ is greater than ‘a’ => proceed right

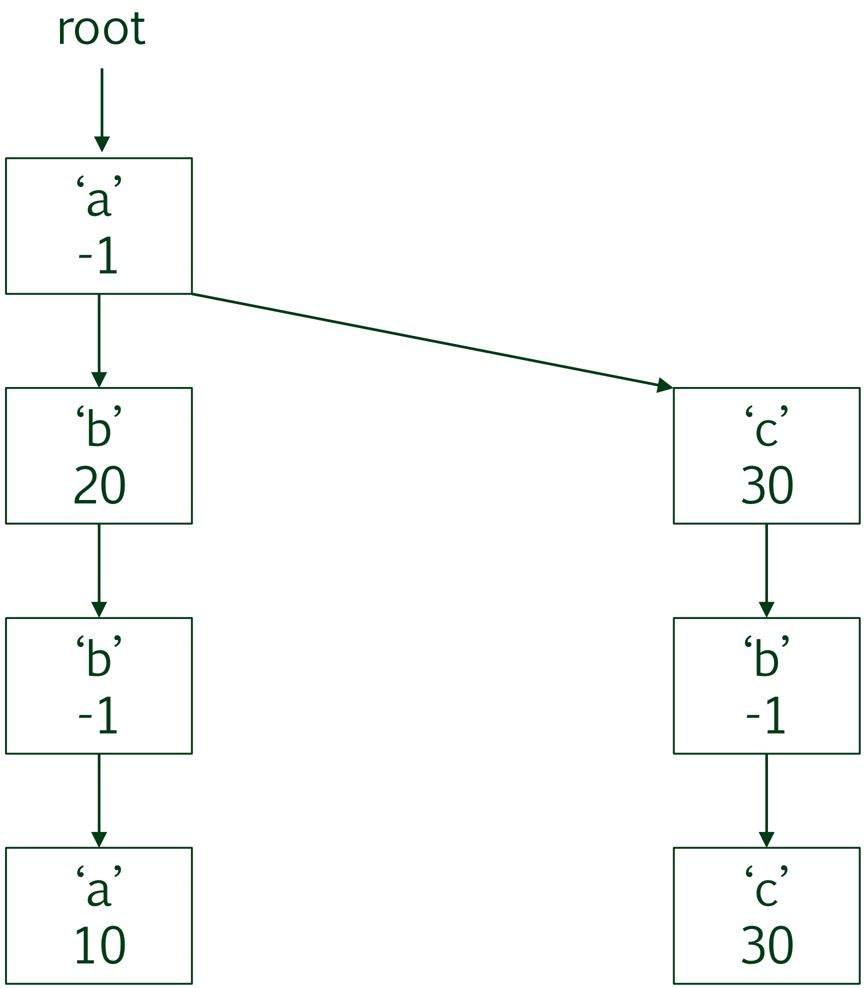


Insert <cbc, 30>

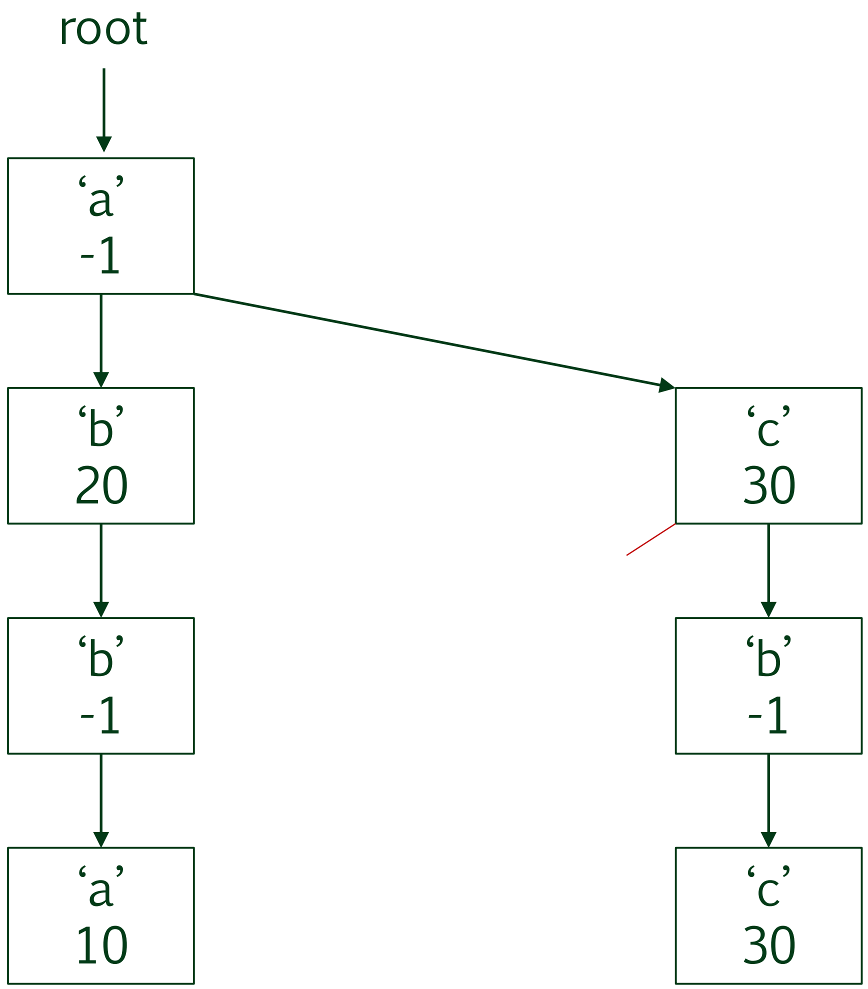


'c' is equal to 'c' => proceed middle

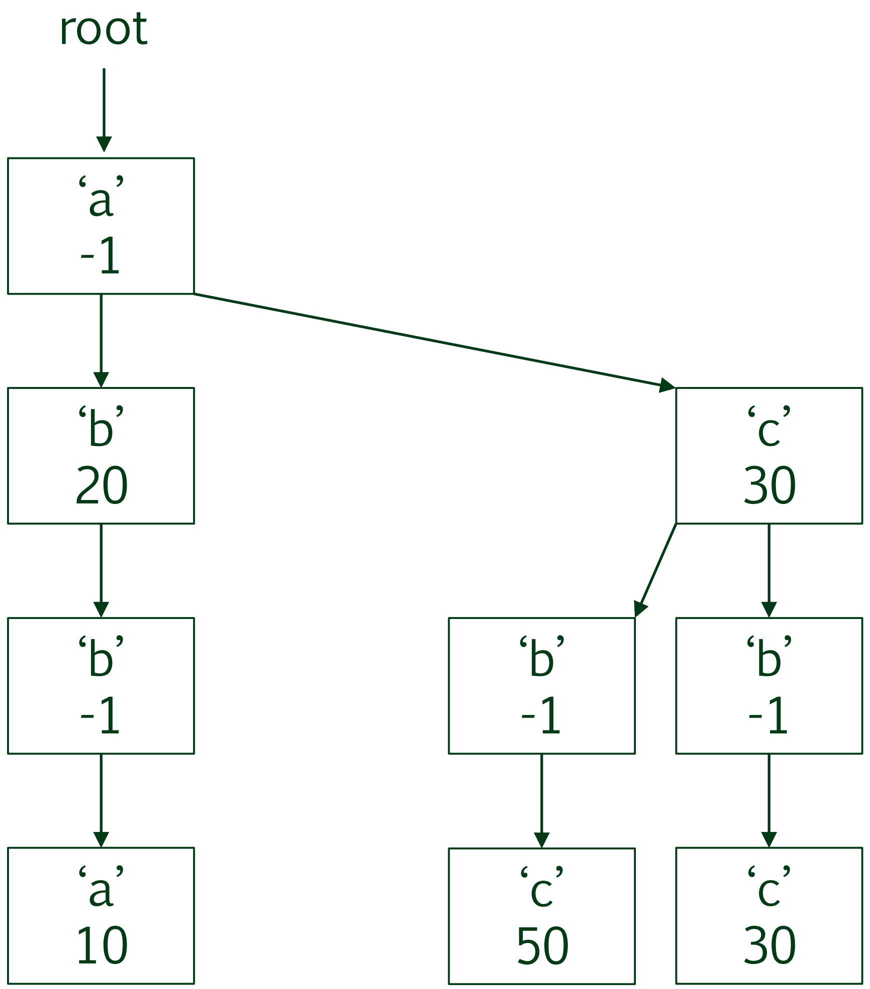
Insert <cbc, 30>



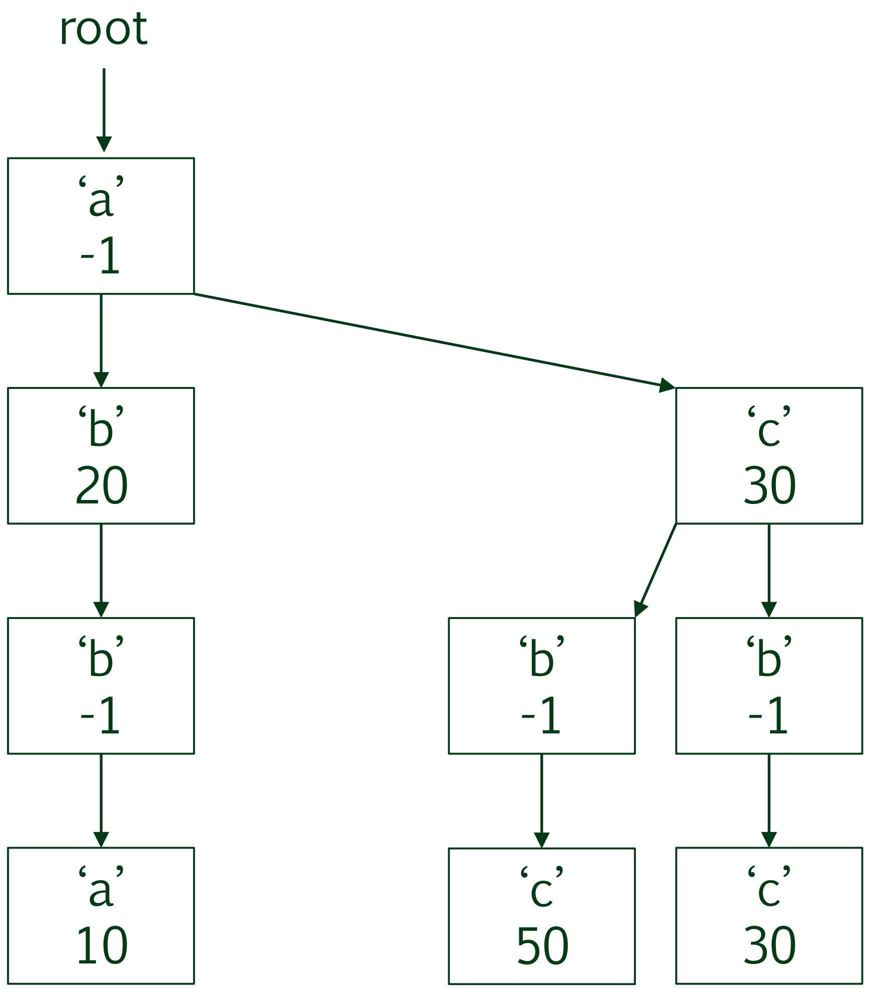
Insert <bc, 50>



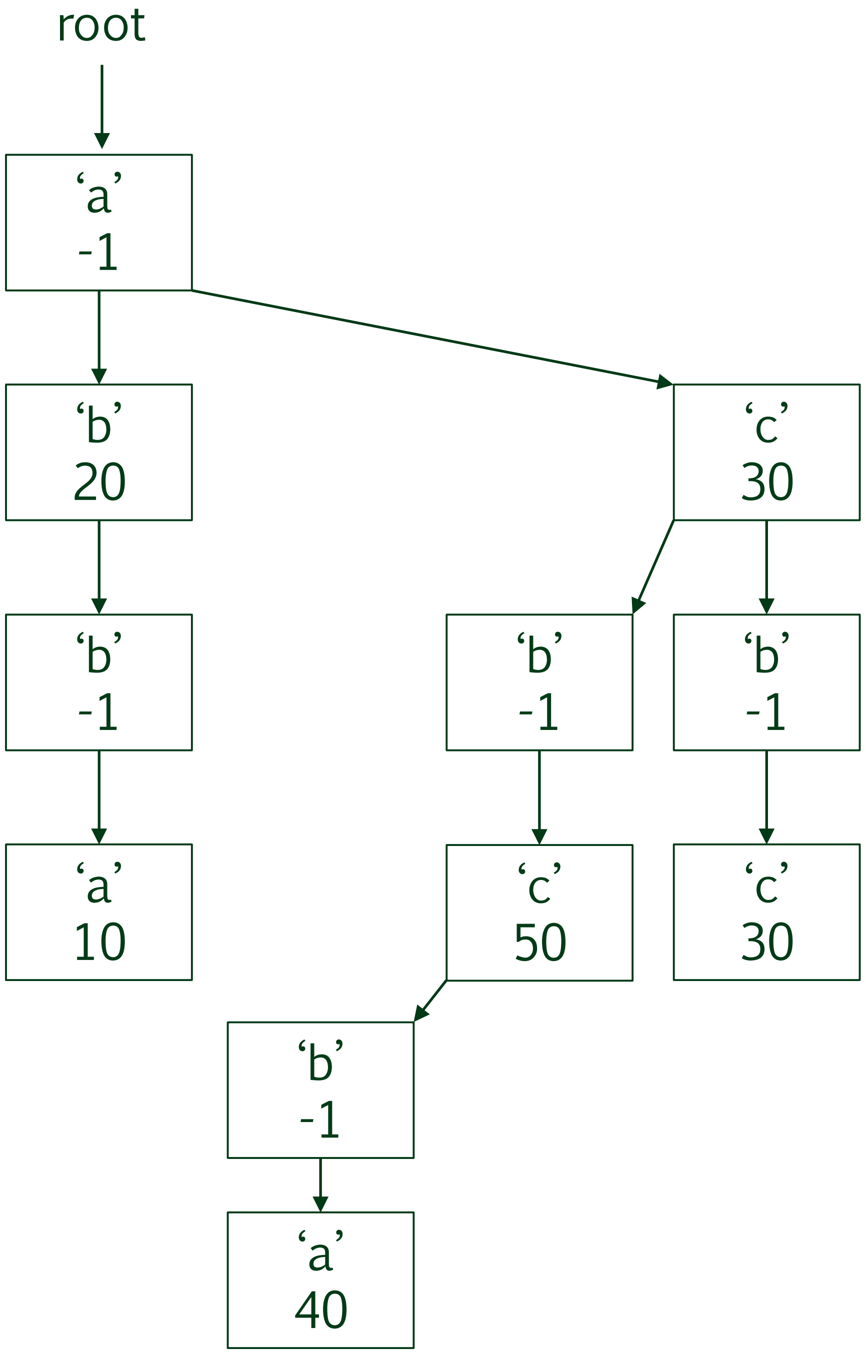
Insert <bc, 50>



Exercise
Insert <bba, 40>



Insert <bba, 40>



Key observation

Only once you proceed down a middle path
do you move on to the next character in the key

Exercises

- Show how the worst-case time complexity of Insert is O(n+L) where n is the number of nodes in the ternary tree and L is the length of the given key.

- Choose six random four-letter words as keys and insert six <key,value> pairs into an initially empty ternary tree. What could be the maximum possible depth of the resultant tree?

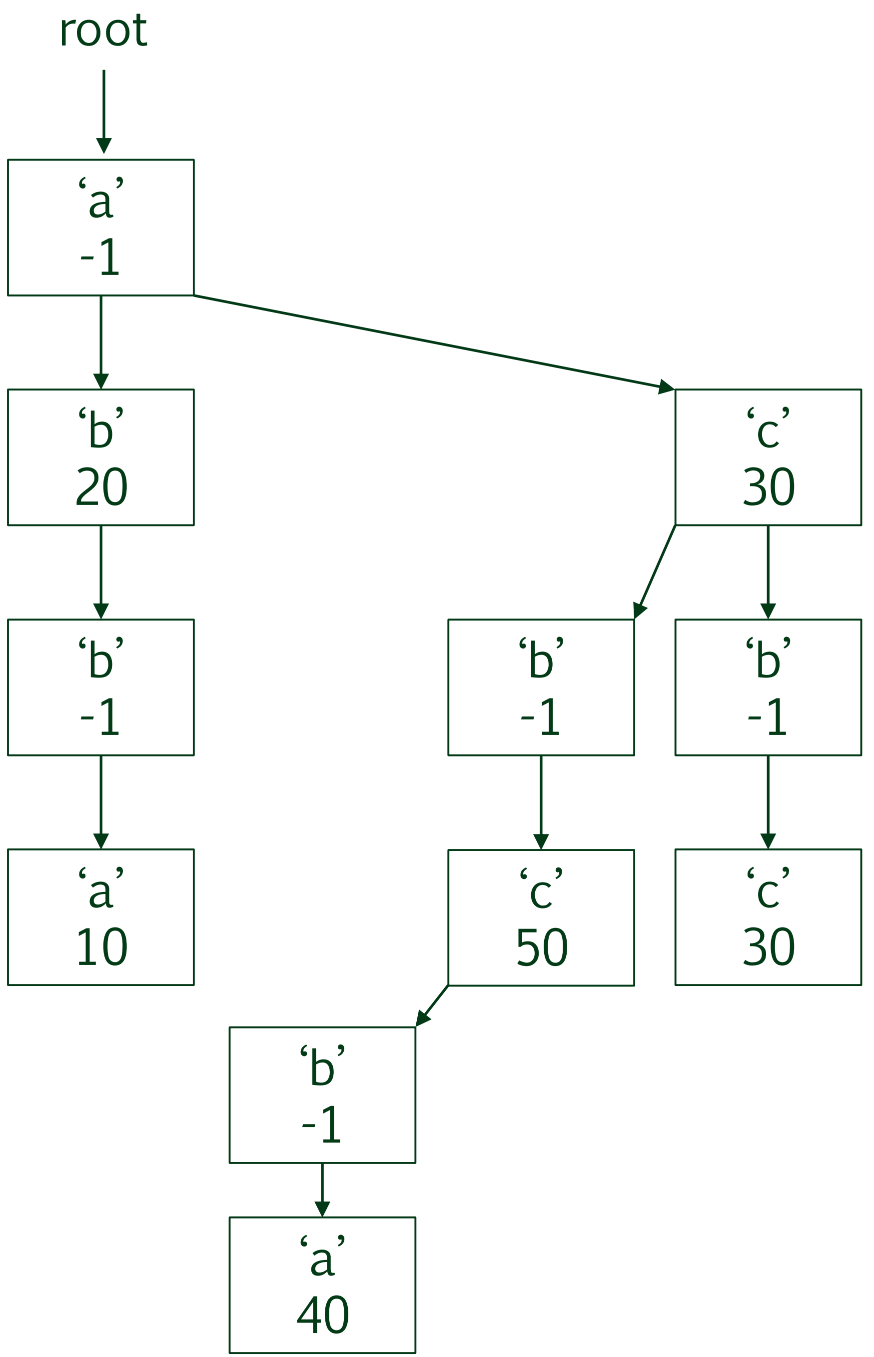
Value

› Basic strategy

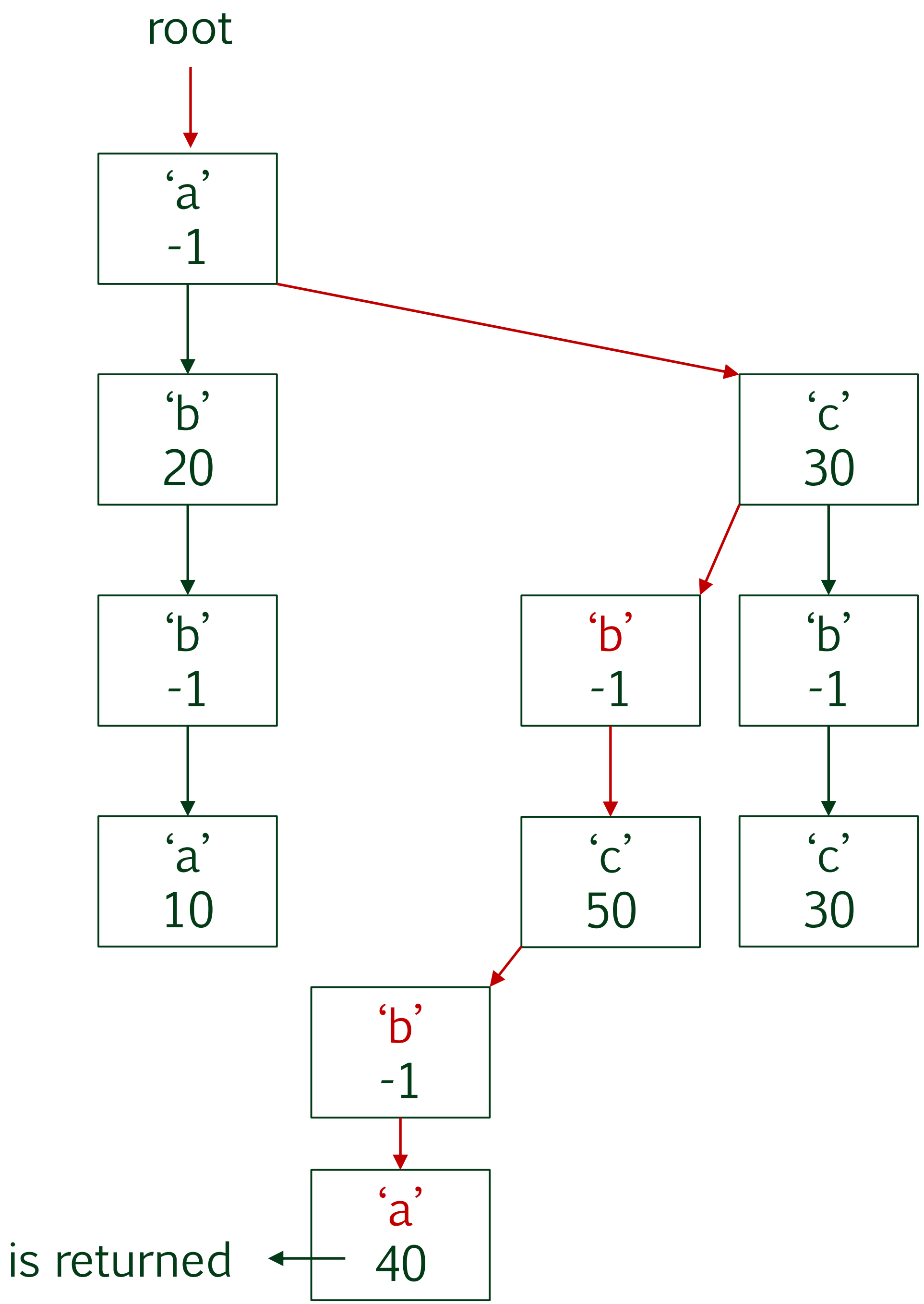
– Beginning at the root and the first character in the key, traverse the ternary tree (as described earlier) until all characters of the key have been examined or a null pointer is reached

– Return true if a value is found; false otherwise

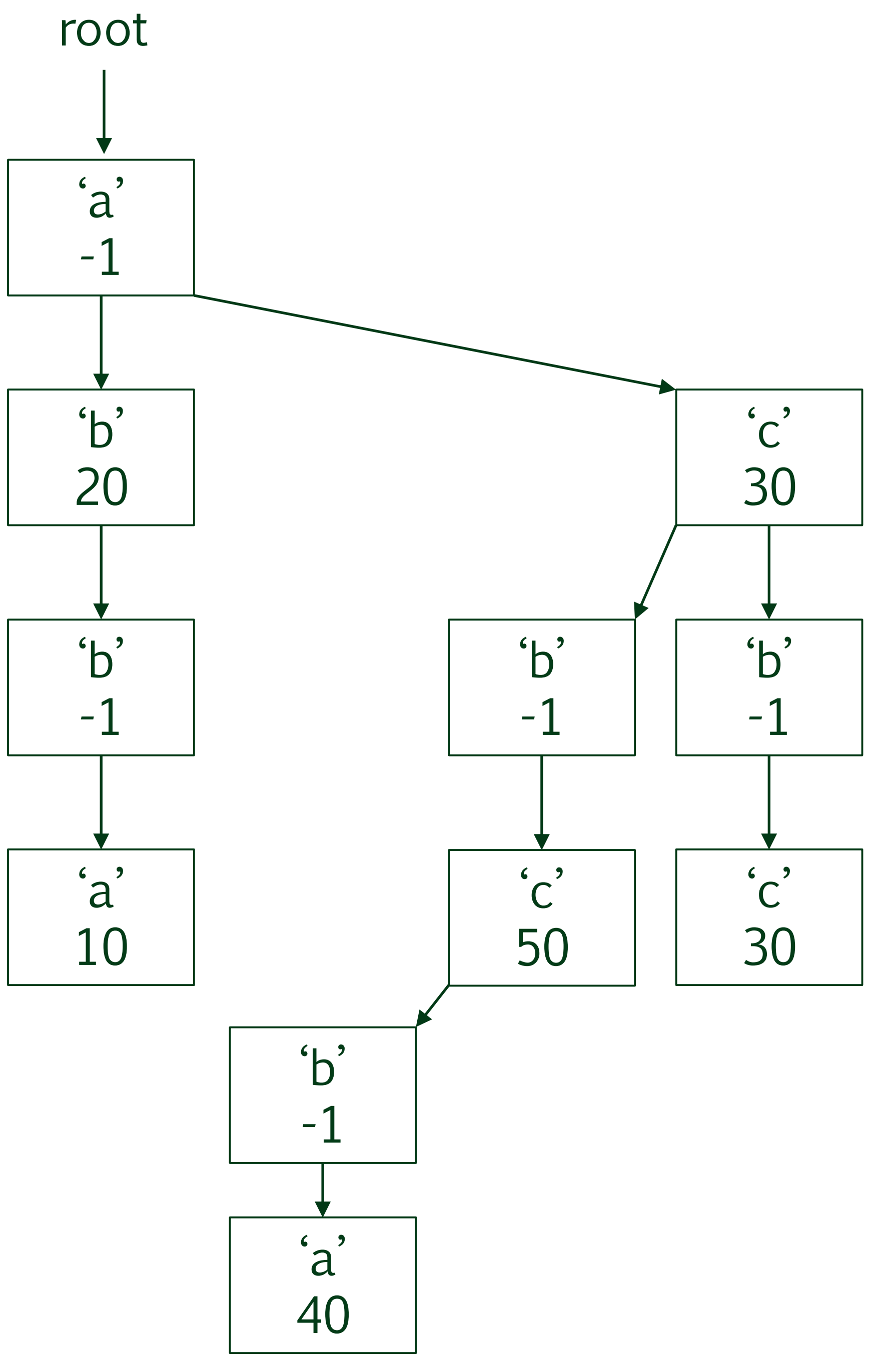
Value <bba, ?>



Value <bba, ?>

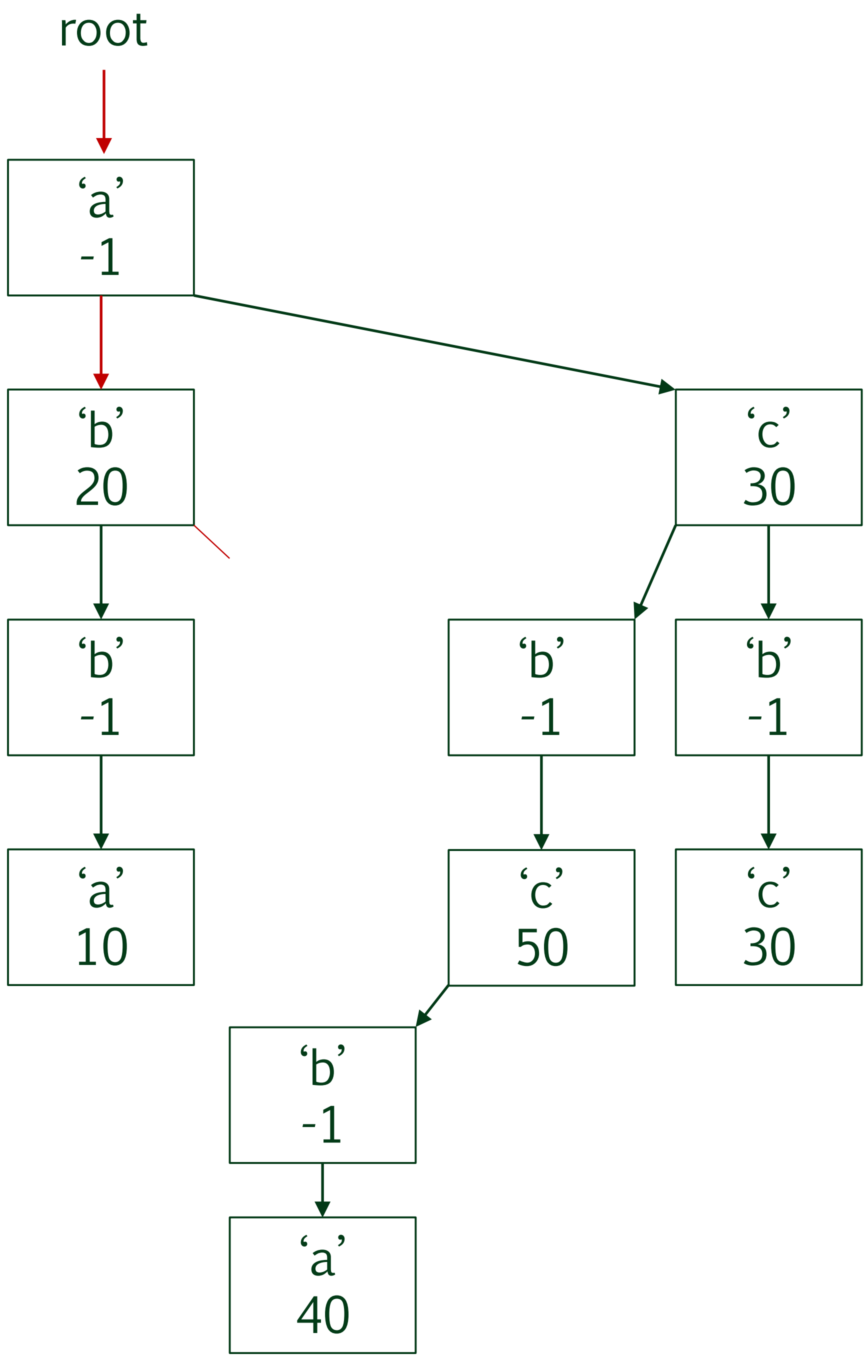


Value <ac, ?>

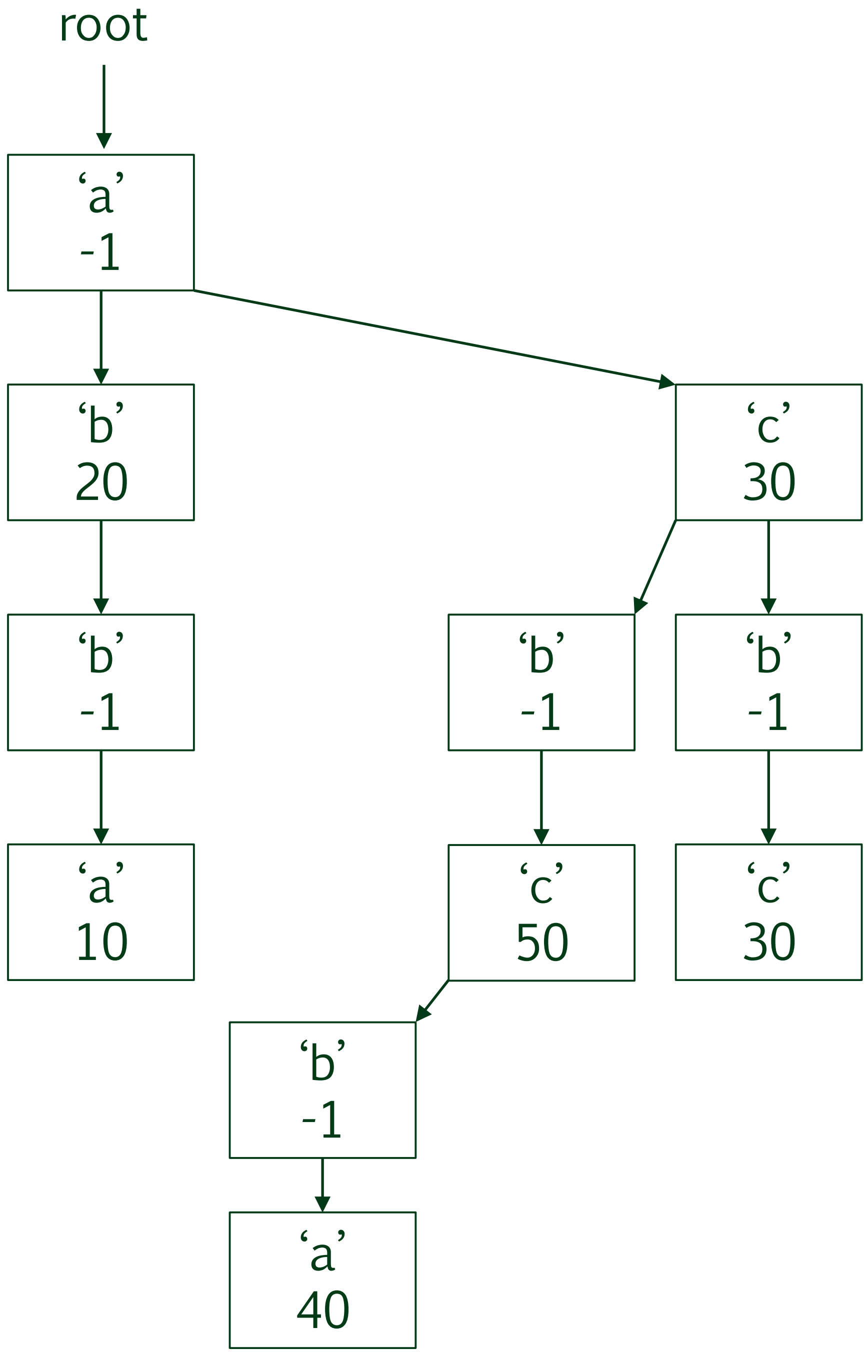


Value <ac, ?>

Null pointer is reached
Default value is returned

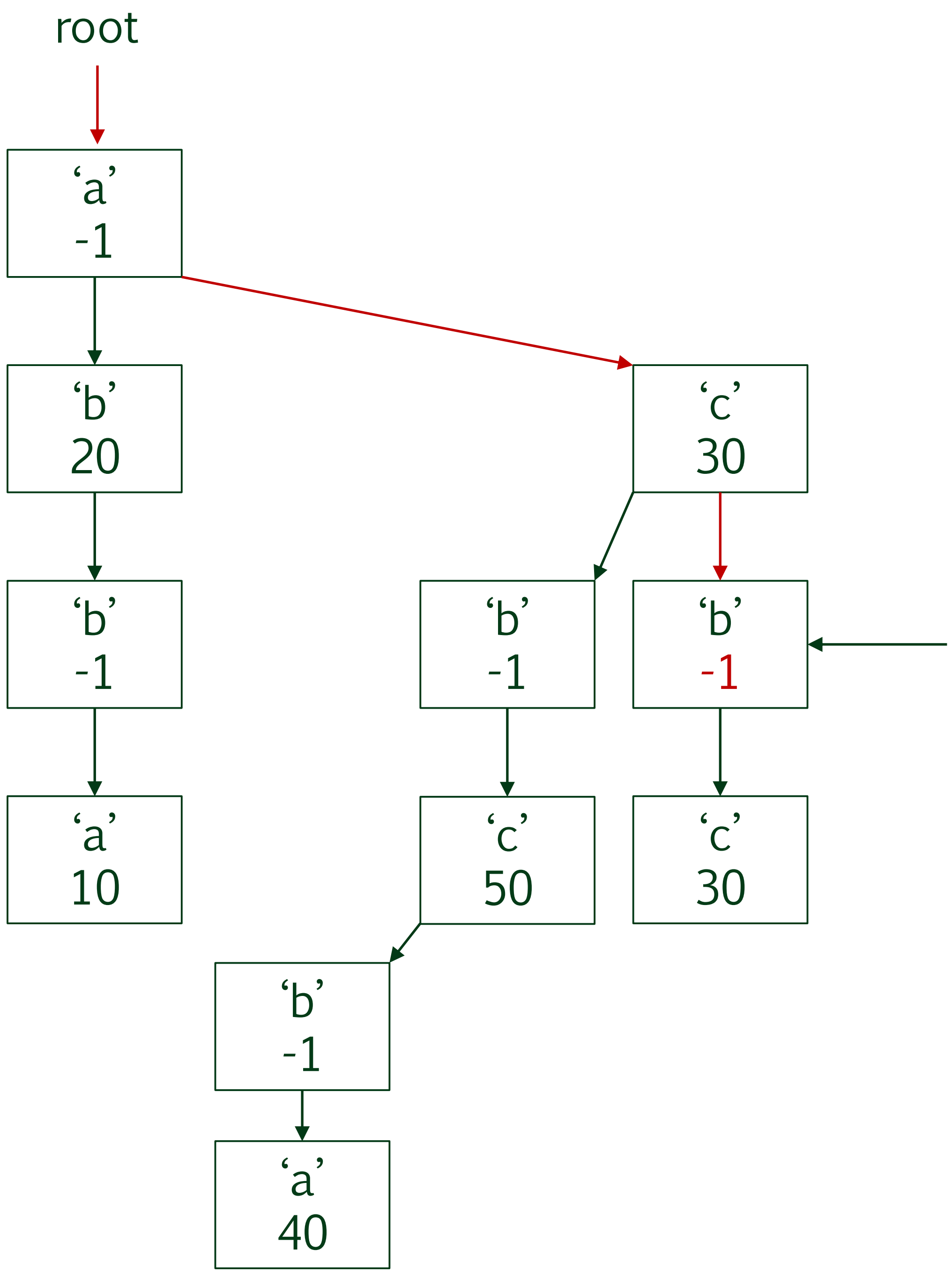


Value <cb, ?>



Value <cb, ?>

Default value is returned



Time complexity

- The time complexity of Value is O(d) where d is the depth of the ternary tree
- If the tree is balanced then d is O(log n); otherwise, the depth is O(n) in the worst case where n is the number of nodes

Exercise

- Implement a method that returns all keys that match a given pattern. A pattern is defined as a string with letters and asterisks where each asterisk represents a wildcard character. Therefore, the pattern *a*d would return all four-letter keys whose second and fourth characters are ‘a’ and ‘d’ such as “card” and “band”.

Print

- Like the r-way trie, the keys can output in order
- Also, like the r-way trie, the keys are constructed as the inorder traversal proceeds through the ternary tree

Algorithm

```
public void Print()
{
    Print(root,"");
}

private void Print(Node p, string key)
{
    if (p != null)
    {
        Print(p.low, key);
        if (!p.value.Equals(default(T)))
            Console.WriteLine(key + p.ch + " " + p.value);
        Print(p.middle, key+p.ch);
        Print(p.high, key);
    }
}
```

Key is being constructed

Exercise

- Write a method that returns all keys that begin with a given prefix. For example, if the prefix is bag, then the method returns all keys that begin with bag. Do you see the similarity with the last exercise?

Very useful reference from Robert Sedgewick and Kevin Wayne of Princeton University

https://www.cs.princeton.edu/courses/archive/spring21/cos226/lectures/52Tries.pdf