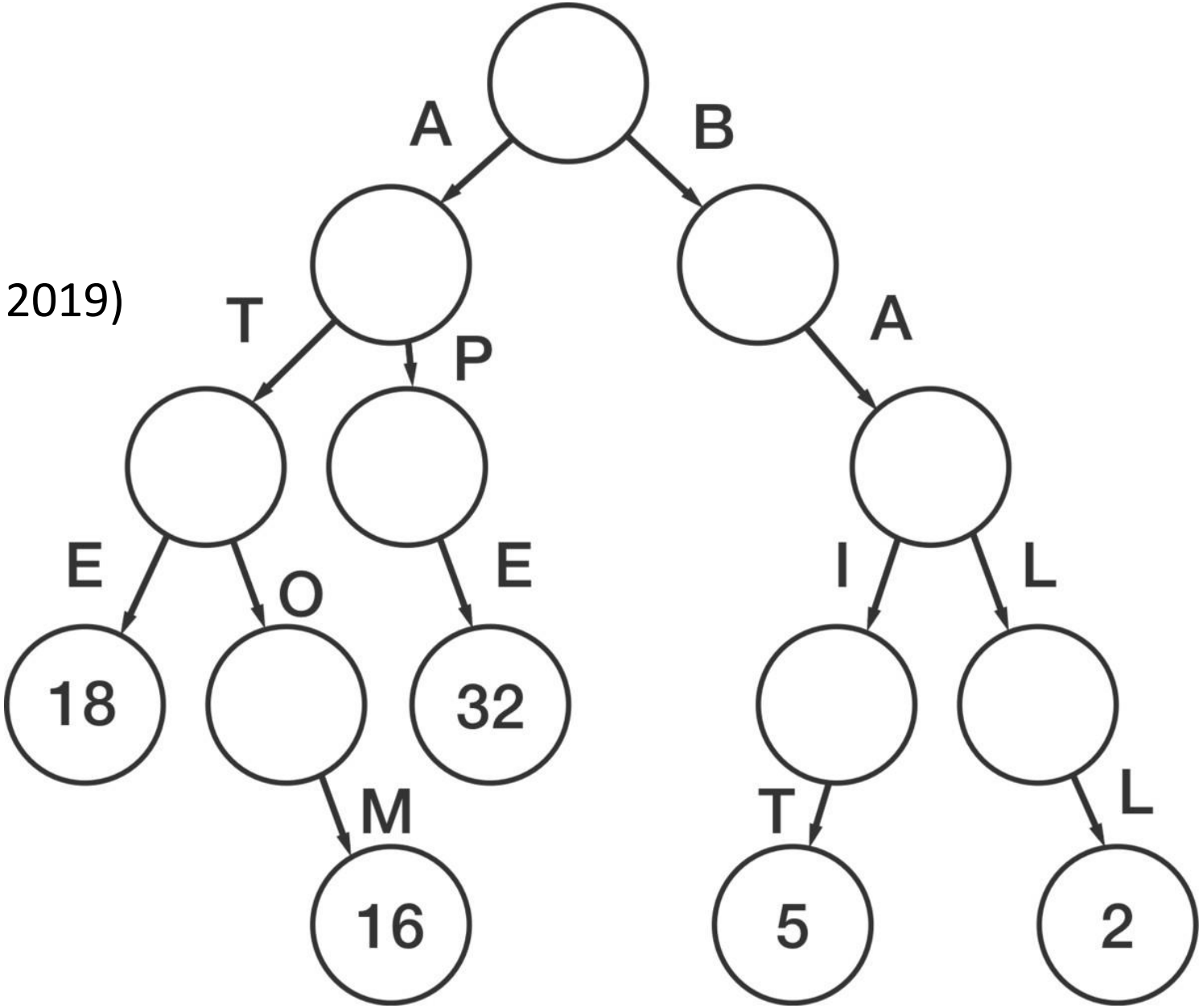
Tries

By Adri Wessels

IOI Training Camp 2 (8-9 February 2019)

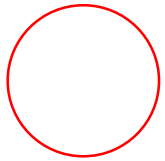


Tries

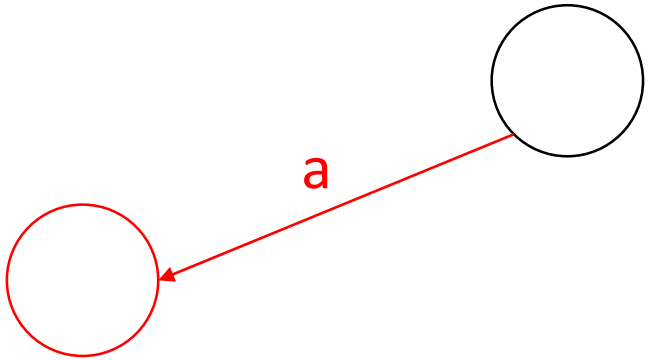
- A “tree”-type structure used to store key-value pairs or optionally just existence of keys.
- Also known as re**trie**val trees, radix trees or prefix trees.
- Made to be a compromise between time and memory.
- Each node stores only a single character of the key, with the final node of the key also storing the value inserted with the key.
- Creates an implicit ordering of the keys through it’s implementation.
- Pronounced “try”.

Tries – Example:

We start with the empty root node



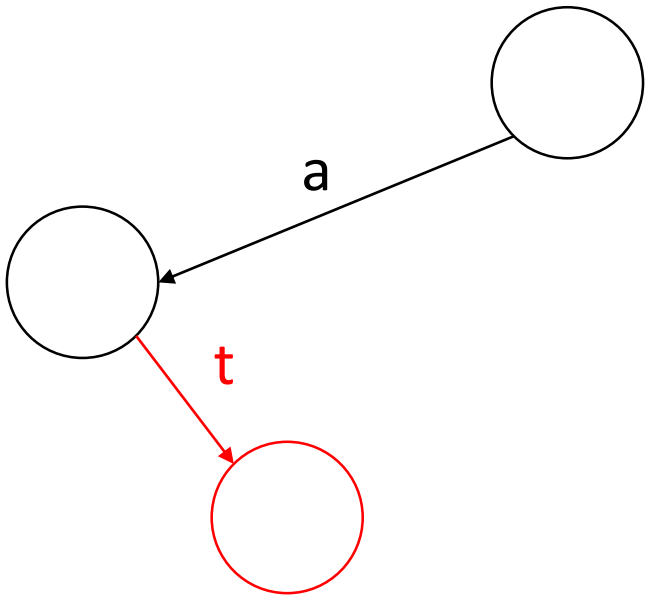
Tries – Example:



We start with the empty root node

We then begin adding the nodes for the first key: “atom”

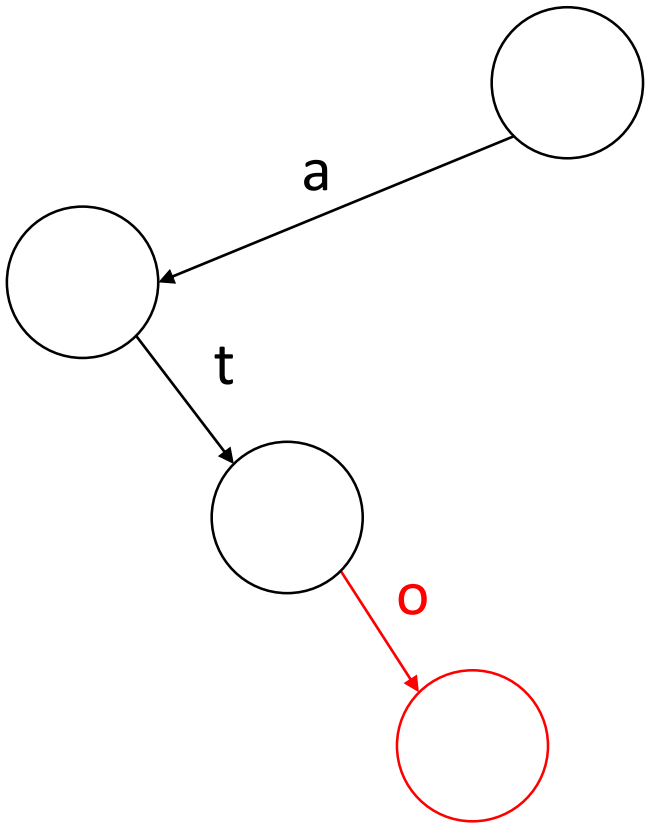
Tries – Example:



We start with the empty root node

We then begin adding the nodes for the first key: “atom”

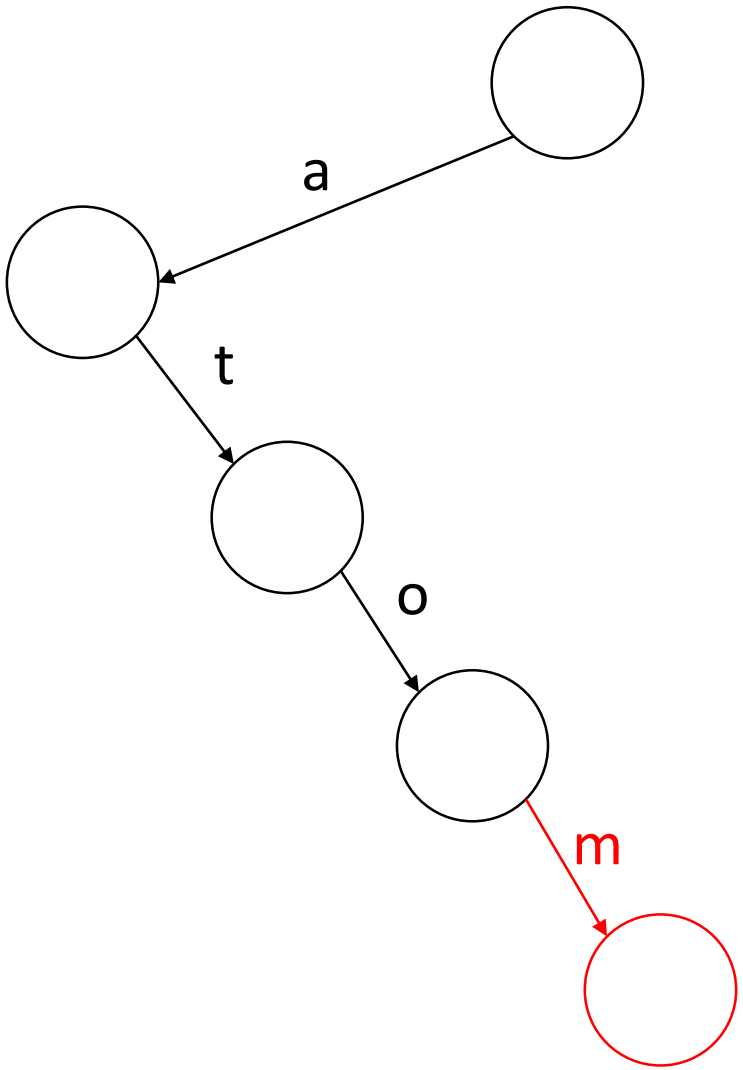
Tries – Example:



We start with the empty root node

We then begin adding the nodes for the first key: “atom”

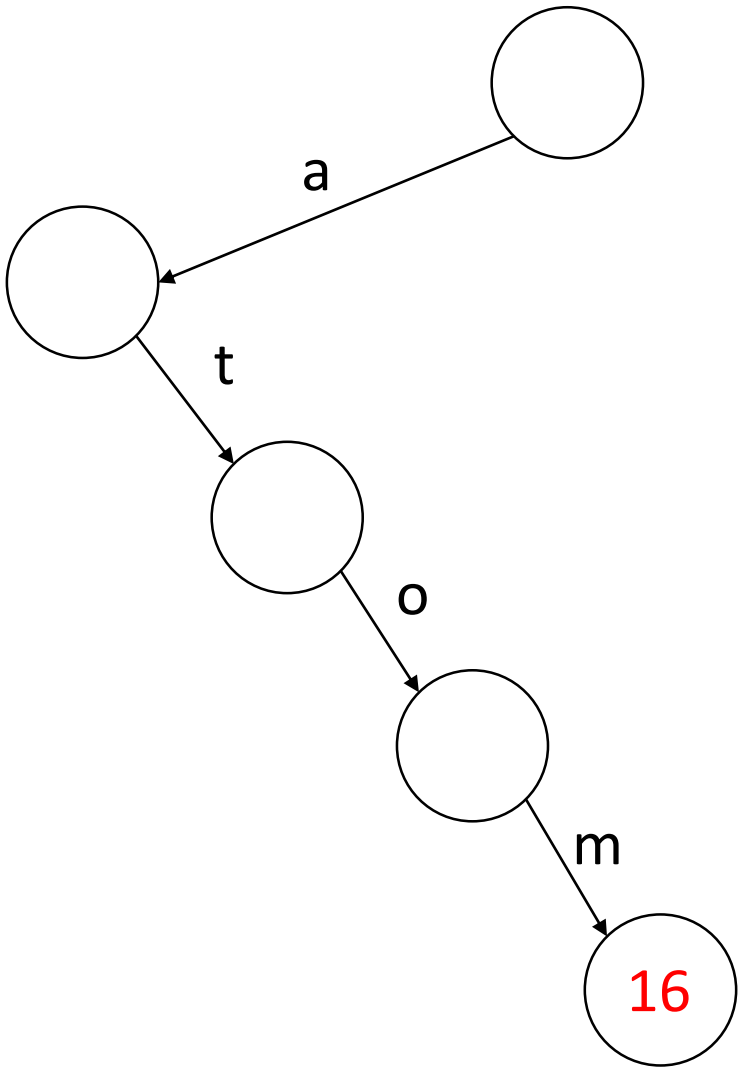
Tries – Example:



We start with the empty root node

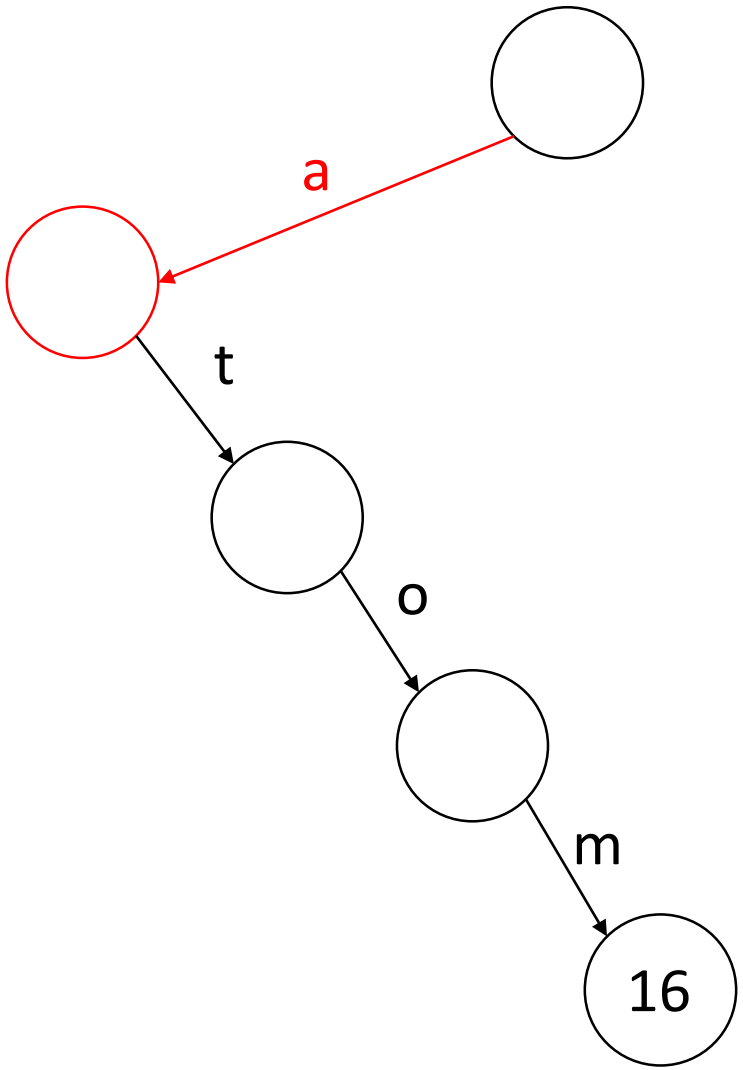
We then begin adding the nodes for the first key: “atom”

Tries – Example:



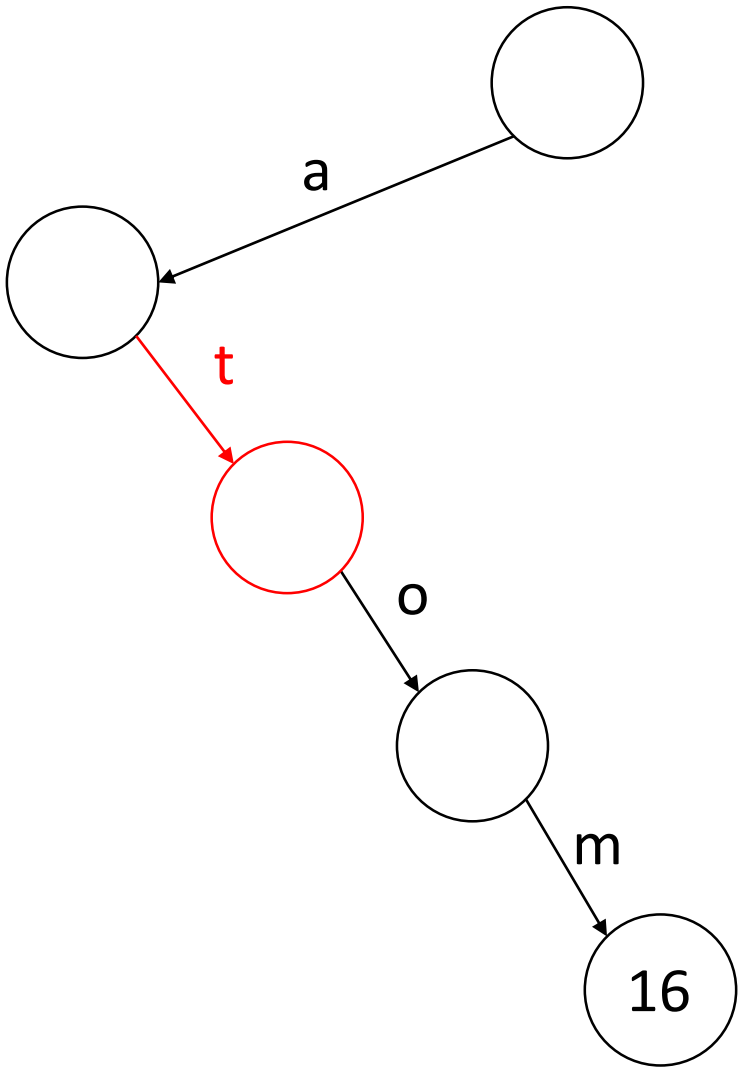
We start with the empty root node
We then begin adding the nodes for the first key: “atom”
Then the associated value: 16

Tries – Example:



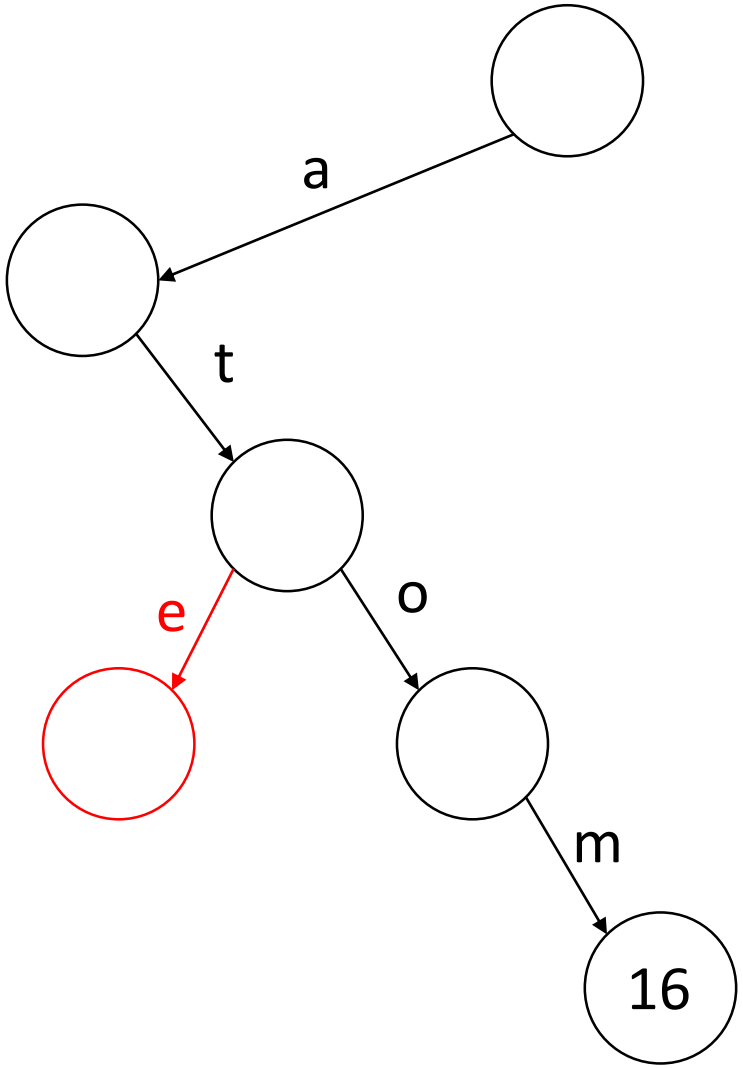
We start with the empty root node
We then begin adding the nodes for the first key: “atom”
Then the associated value: 16
We can then add another key: “ate”

Tries – Example:



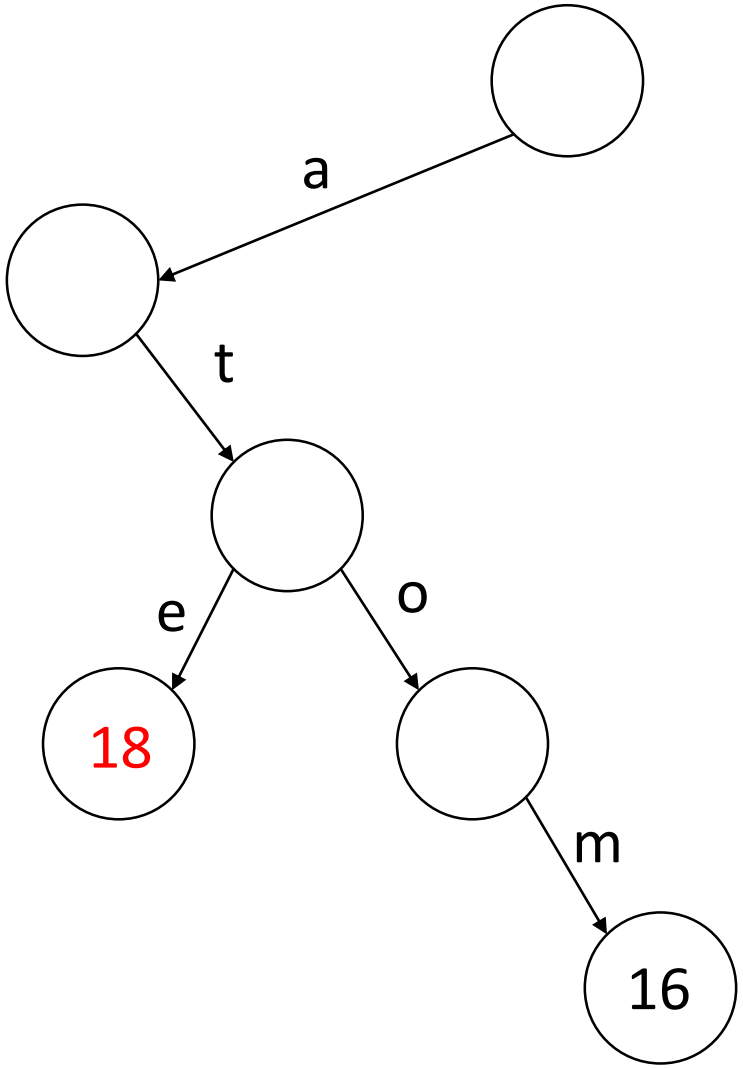
We start with the empty root node
We then begin adding the nodes for the first key: “atom”
Then the associated value: 16
We can then add another key: “ate”

Tries – Example:



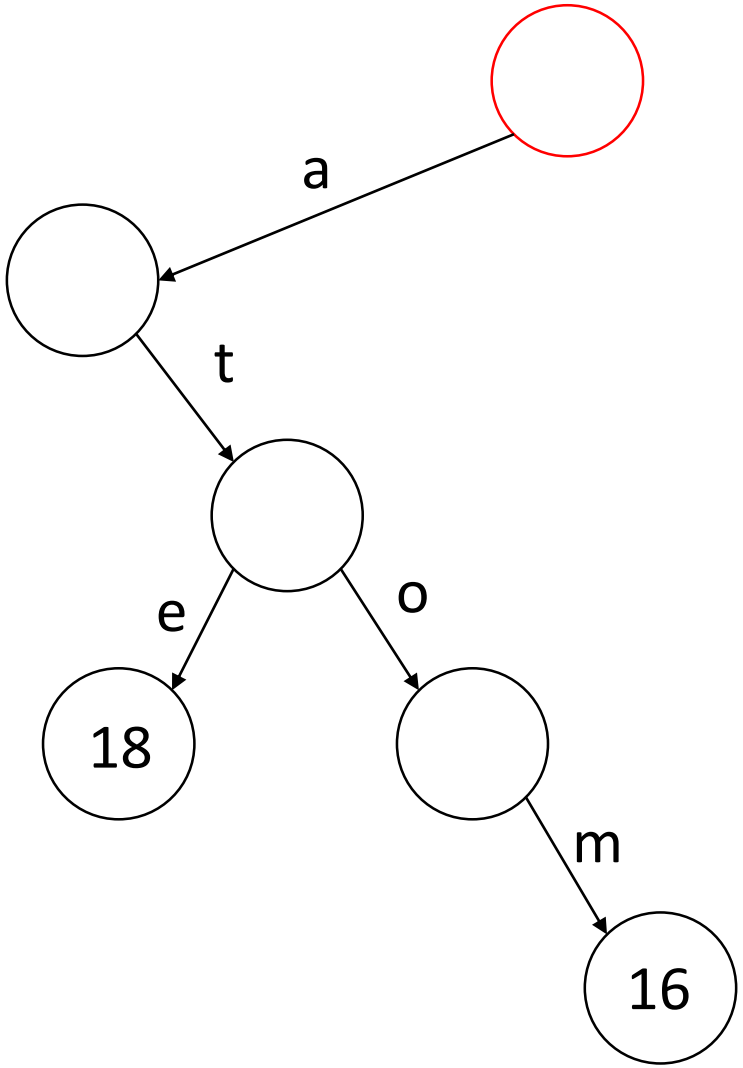
We start with the empty root node
We then begin adding the nodes for the first key: “atom”
Then the associated value: 16
We can then add another key: “ate”

Tries – Example:



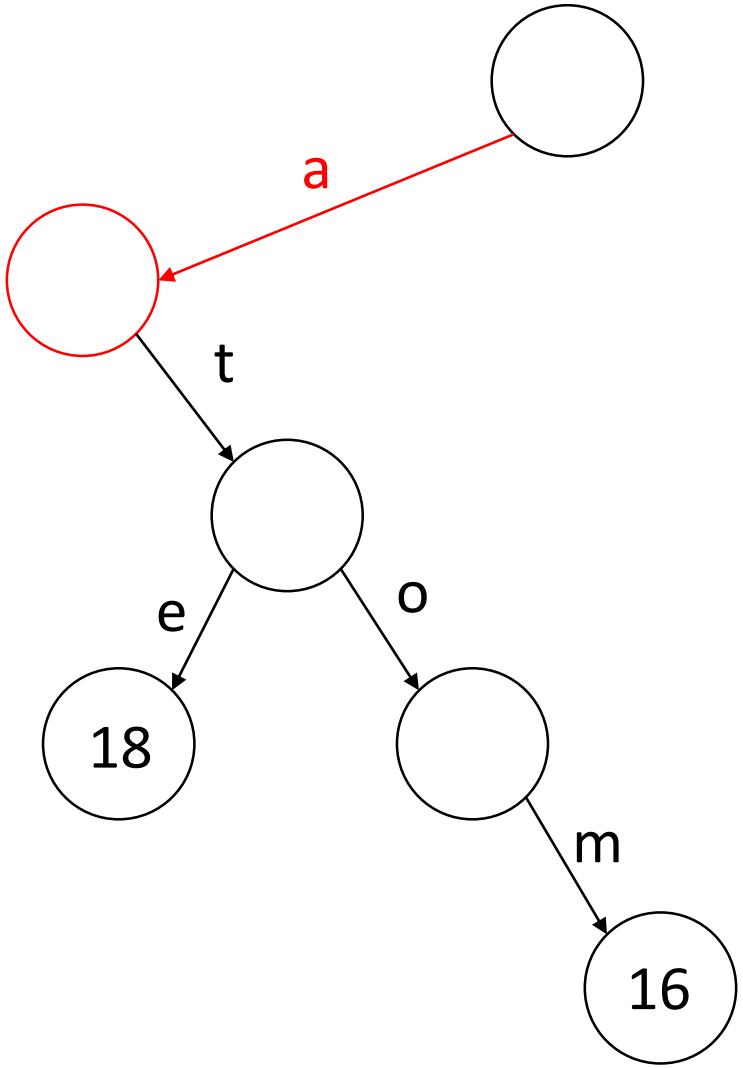
We start with the empty root node
We then begin adding the nodes for the first key: “atom”
Then the associated value: 16
We can then add another key: “ate”
Then the value again: 18

Tries – Example:



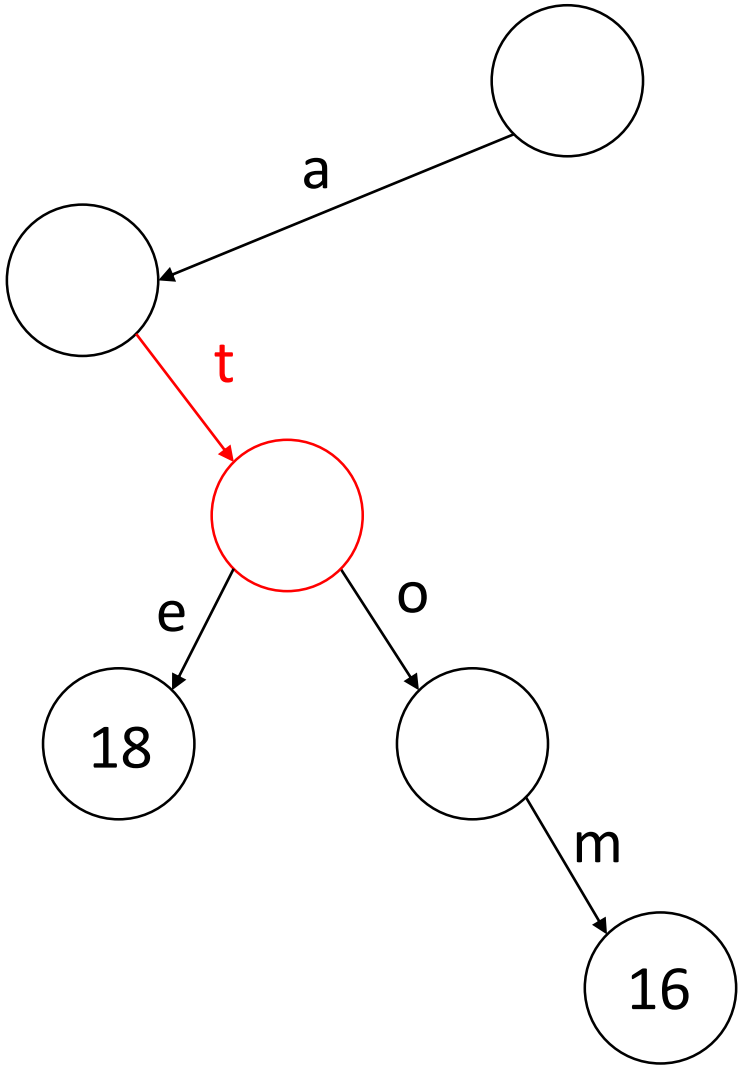
We start with the empty root node
We then begin adding the nodes for the first key: “atom”
Then the associated value: 16
We can then add another key: “ate”
Then the value again: 18
Maybe we want to add the key: “at”

Tries – Example:



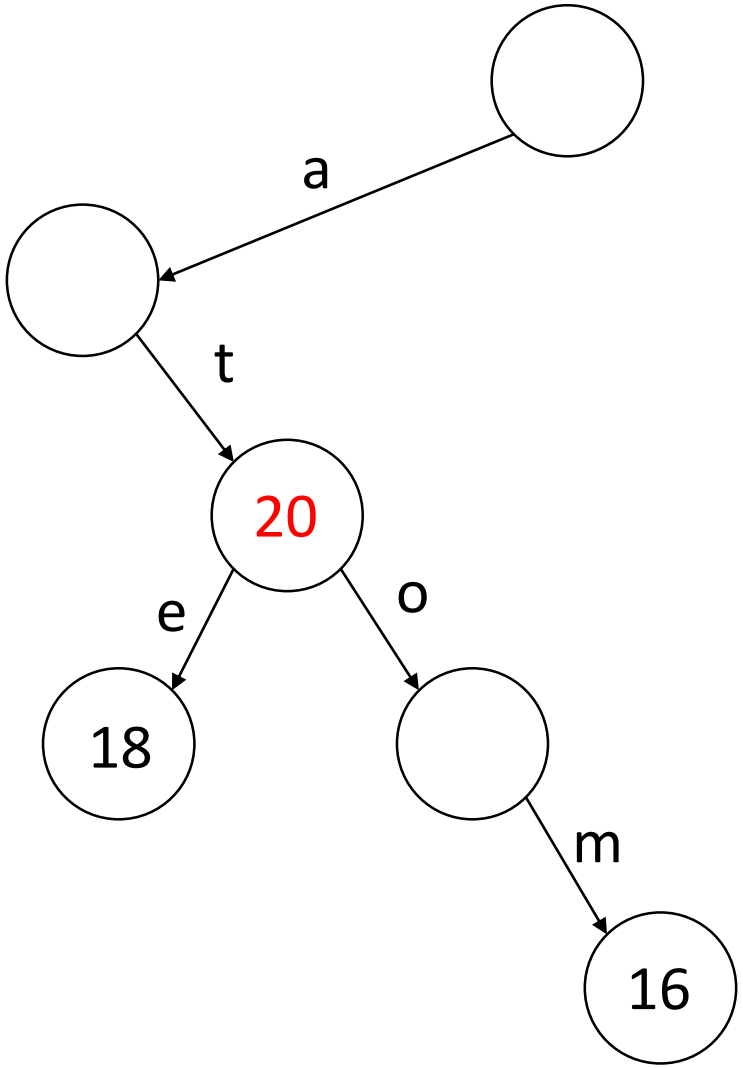
We start with the empty root node
We then begin adding the nodes for the first key: “atom”
Then the associated value: 16
We can then add another key: “ate”
Then the value again: 18
Maybe we want to add the key: “at”

Tries – Example:



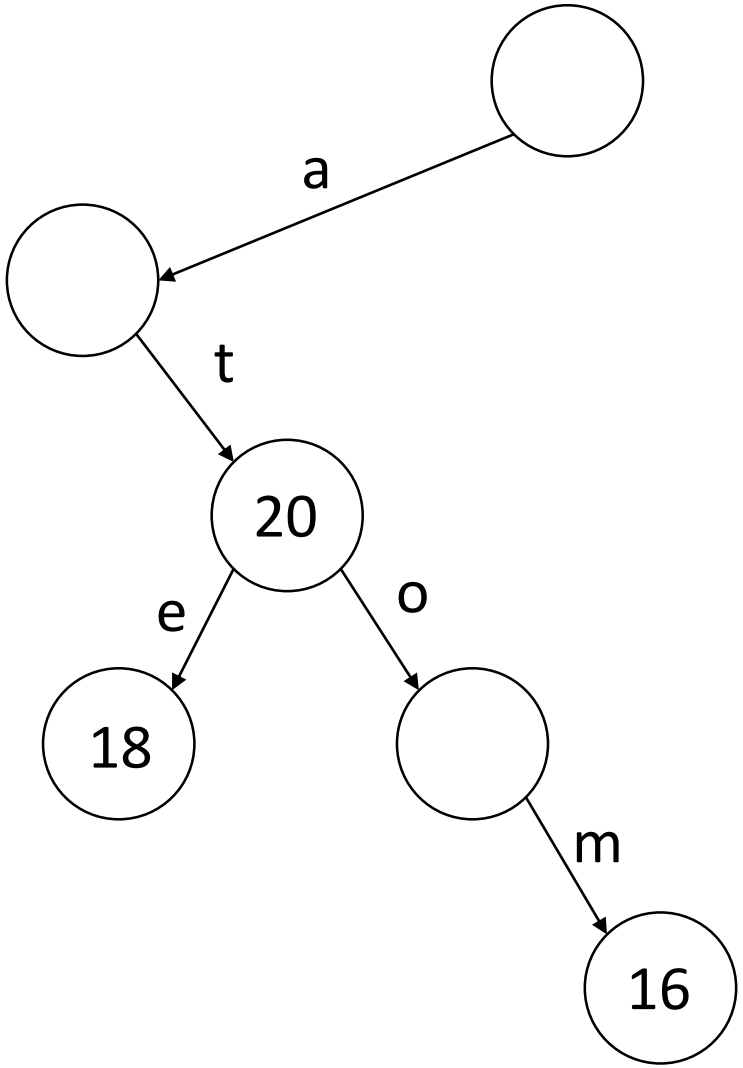
We start with the empty root node
We then begin adding the nodes for the first key: “atom”
Then the associated value: 16
We can then add another key: “ate”
Then the value again: 18
Maybe we want to add the key: “at”

Tries – Example:



We start with the empty root node
We then begin adding the nodes for the first key: “atom”
Then the associated value: 16
We can then add another key: “ate”
Then the value again: 18
Maybe we want to add the key: “at”
Then the value: 20

Tries – Example:



We start with the empty root node
We then begin adding the nodes for the first key: “atom”
Then the associated value: 16
We can then add another key: “ate”
Then the value again: 18
Maybe we want to add the key: “at”
Then the value: 20
Finally we have our completed trie with our 3 keys ordered: at, ate, atom.

Tries – Example:

- From our example we can see that insertion is O(L) where L is the length of the key since we just traverse the tree down and go through L nodes along the way.
- Similarly checking if a key exists or getting the value of a key is also O(L) since we essentially do the same thing, but then just retrieve the value at the end.
- The example also shows that a prefix of another key can also be its own key (Note: This is dependant on the implementation).
- It should also be noted that the keys do not have to be strings and can be any type that can be represented as an array of a smaller type where the smaller type is the equivalent “character” type.

Tries – Implementation:

- First we need some defines

```
#define ALPHABET_SIZE 26
#define SMALLEST_CHAR 'a'
#define DEFAULT_VAL 0
```

- ALPHABET_SIZE is the amount of different values each character can take.
- SMALLEST_CHAR is the smallest value that each character can take so that the characters can be offset to start at 0.
- DEFAULT_VAL is the default value that a node stores when it doesn’t store an inserted value. Another implementation is to have nodes store pointers to values and then use nullptr as the default value.

Tries – Implementation:

- The implementation of the tree itself amounts to a single Node pointer.

```
Node* root;
```

- The implementation shown uses strings as keys and ints as values, but you can change this if you need to without too much trouble.

Tries – Implementation:

- Next we have the Nodes which store all the data and connect to each other by storing pointers to their children.

```
struct Node
{
    array<Node*, ALPHABET_SIZE> children;
    int value;
    char childNum; //Only needed for deletion of nodes

    Node::Node() : children(), value(DEFAULT_VAL), childNum(0)
    {
        children.fill(nullptr);
    }
};
```

- Note: std::array is used here, but it can be replaced with a normal array. This does require you to include <array>.

Tries – Implementation:

- To insert we traverse down the tree until we reach the end of the key and then add the value in.
- At each step you check if the node already exists.
- If the node exists, traverse to it.
- If it doesn’t, create it and traverse to it.
- When you traversed down to the last node of the key, give that node the value.
- Note: The new keyword is used to create new Nodes because it’s easier (and probably faster) than having a global vector to store them all in. You can also create the root note with new if you want to.

Tries – Implementation:

```
void insert(const string& key, const int& value)
{
    Node* search = root;

    for (int i = 0; i < key.size(); i++)
    {
        if (search->children[key[i] - SMALLEST_CHAR] == nullptr)
        {
            search->children[key[i] - SMALLEST_CHAR] = new Node();
            search->childNum++; //Only needed for deletion
        }

        search = search->children[key[i] - SMALLEST_CHAR];
    }

    search->value = value;
}
```

Tries – Implementation:

- The method to get the value for the key returns a pointer to the value.
- We also traverse the tree, but if a node doesn't exist we just return nullptr.
- If we get to the end we return the address of the value.

```
int* getVal(const string& key)
{
    Node* search = root;

    for (int i = 0; i < key.size(); i++)
    {
        search = search->children[key[i] - SMALLEST_CHAR];
        if (search == nullptr) return nullptr;
    }

    if (search->value == DEFAULT_VAL) return nullptr;
    else return &search->value;
}
```

Tries – Implementation:

- To delete we need to traverse down the tree and then traverse back up while deleting nodes that don’t correspond to any other keys along the way.
- The easiest way to do this is with recursion.
- We traverse down to the bottom and then remove the value from the last node of the key.
- On the way back up the key we check if the node has any children, and if it is storing a value. If neither of these are true we can delete it.
- We then return whether we deleted the node so that the next node up knows to remove the deleted node from it’s children.
- Not as easy as the rest of the implementation, but also not always necessary.

Tries – Implementation:

```
bool deleteKey(const string& key, Node* cur = nullptr, int depth = 0)
{
    if (cur == nullptr) cur = root;
    if (depth == key.size())
    {
        cur->value = DEFAULT_VAL;
    }
    else
    {
        bool deleted = deleteKey(key, cur->children[key[depth] - SMALLEST_CHAR], depth + 1);
        if (deleted)
        {
            cur->childNum--;
            cur->children[key[depth] - SMALLEST_CHAR] = nullptr;
        }
    }

    bool out = false;
    if (cur->childNum == 0 && cur->value == DEFAULT_VAL)
    {
        out = true;
        delete cur;
    }

    return out;
}
```

Tries – Analysis:

- Inserting, retrieving and deleting values is O(L) because the dominating term is the traversal of the trie along the key.
- Space complexity is O(NL) where N is the number of keys and L is the length of the longest key.