Tries & Ternary Trees

© Sanjeev Qazi 2021

Tries

- Data structure for very fast look up.
- Also called *prefix* tree (we will see why)

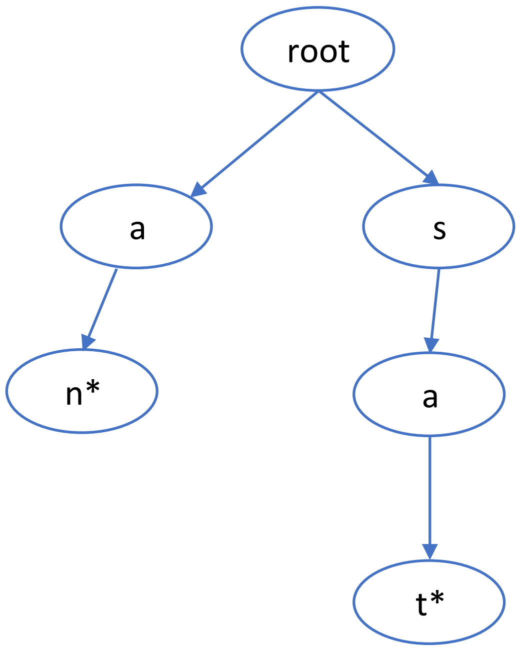
- Usage:
  - auto correct
  - get all words with a prefix
  - type ahead suggestions, etc.

© Sanjeev Qazi 2021

Tries

- Words in this trie:

  1. an
     - prefixes: a
  2. sat
     - prefixes: s, sa



Note that I am not drawing the null nodes.

You can imagine that a node, say, ‘a’ has space already allocated to store a pointer to other character nodes… e.g.: if the word ‘app’ was in this trie, then node ‘a’ would also have a path to node ‘p’ (in addition to the existing path to node ‘n’)

© Sanjeev Qazi 2021

Tries

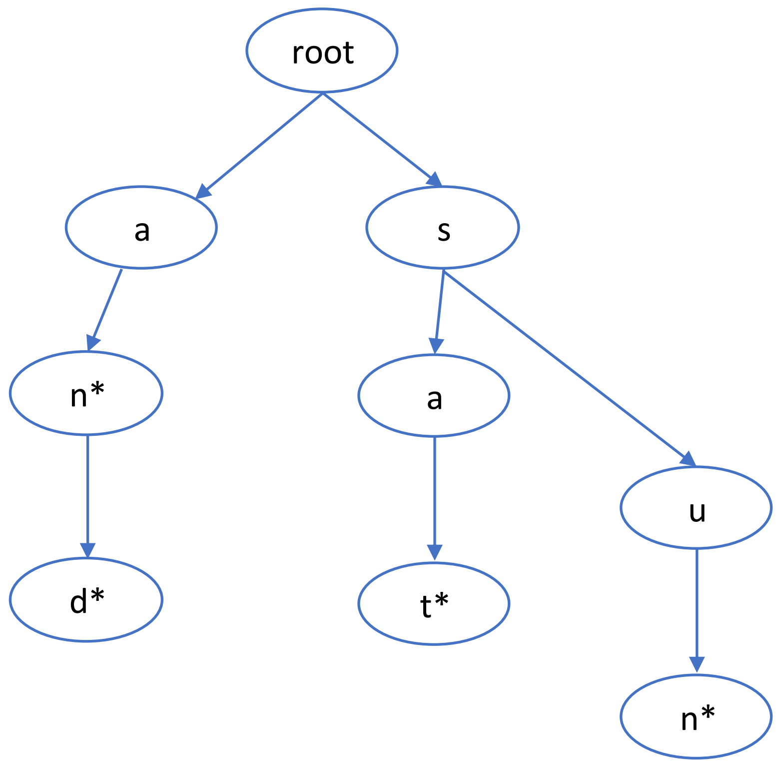
- Words in this trie:

1. an
   - prefixes: a

2. and
   - prefixes: a, an

3. sat
   - prefixes: s, sa

4. sun
   - prefixes: s, su

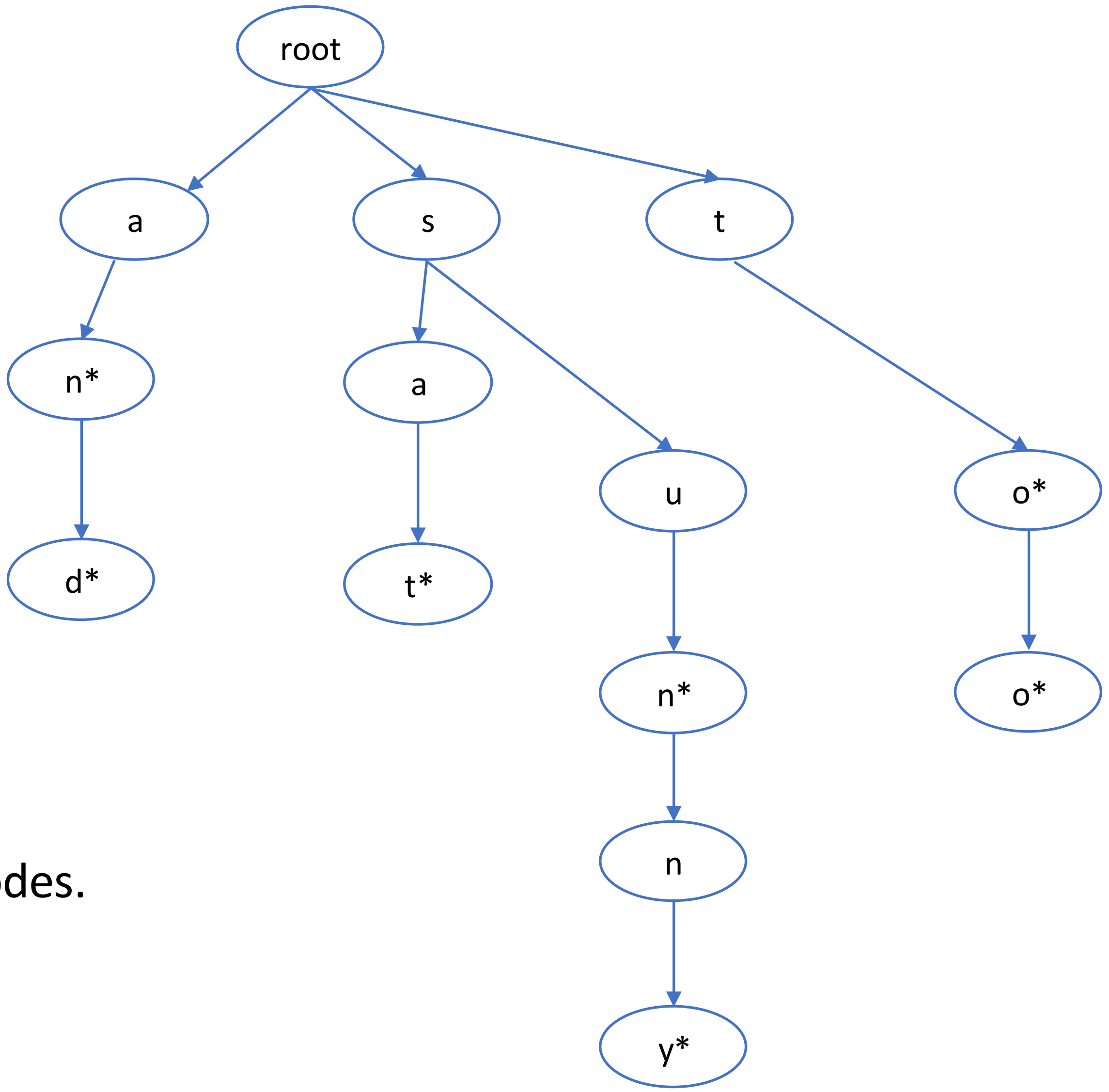


© Sanjeev Qazi 2021

Tries

- Words in this trie:

1. an
2. and
3. sat
4. sun
5. to
6. too
7. Sunny

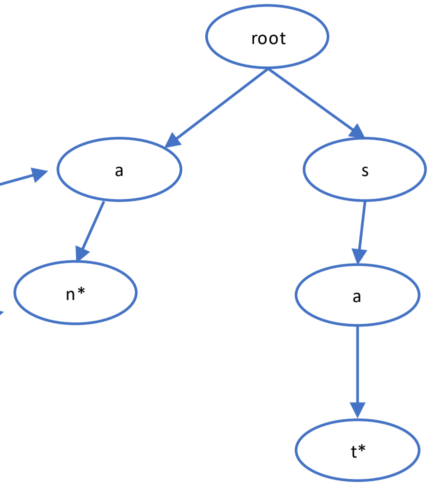


Remember: I am not drawing the null nodes.

© Sanjeev Qazi 2021

Tries

- Each node represents either the end of a word, or a prefix.
- Root can be thought of representing the empty string.



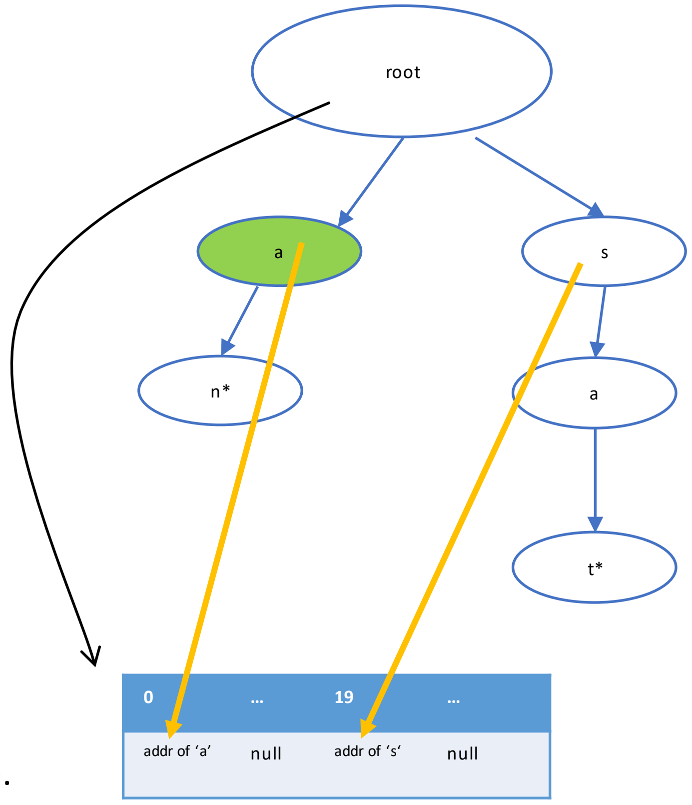
- Nodes at level 1 represent prefixes of length 1 and/or words of length 1
- Nodes at level 2 represent prefixes of length 2 and/or words of length 2

Tries

- Another minor thing ☺
  - We are not really storing the characters in the nodes ☺ even tho it looks that way.

  - Imagine I have an array of size 26, and I tell u that the characters map to it from ‘a’ to ‘z’
  - Index 0 represents ‘a’
  - Index 1 represents ‘b’
  - Index 25 represents ‘z’

  - Now, if I have a node pointer at index 0 (for e.g.: rootNode.nodeArray[0] in the diagram):
    - it would look something like
      - rootNode.nodeArray[0] has the address of (or reference to) another node.
    - And the presence of this node signifies that there is a key that was inserted along that path (“an”).




```
Node {
        Node        nodeArray  [ ALPHABET_SIZE ] ;        // array of node references (or pointers, depending on ur language)
        bool        WordEnd;
}
```

© Sanjeev Qazi 2021

Pseudocode for recursive

```
bool  Contains(  rootNode,    key   )
            if ( key is empty )
                        return true;
            return  ContainsHelper(  rootNode,  key,  0 )

bool ContainsHelper(  node,    key,    index   )
            char  ch = key [  index  ]
            bool  isLastCharInKey =  index ==  (key.length-1);
            if   isLastCharInKey
                return   true if   node.nodesArray  [ ch ]  is not null    AND   node.nodesArray  [ ch ].EndWord  is true
            else
                return ContainsHelper ( node.nodesArray  [ ch ],  key,   index + 1  )
```

Iterative version coming up.

© Sanjeev Qazi 2021

Tries

Pseudocode for recursive

```
void  Add(  rootNode,    key   )
            if ( key is not empty )
                  AddHelper(  rootNode,  key,  0 )

void  AddHelper(  node,    key,    index   )
            char  ch = key [  index  ]
            if   node.nodesArray  [ ch ]  is null
                  node.nodesArray  [ ch ]  =  new  Node          // this character path is new, so, create a new node ( note that we don't store ch )

            if   index ==  (key.length-1)                         // last character in key
                  Set the endWord  flag on  node.nodesArray  [ ch ]
            else
                  AddHelper  (  node.nodesArray  [ ch ] ,  key,  index +  1  )
```

I will leave Delete as an exercise for u all

Iterative version coming up.

© Sanjeev Qazi 2021

Tries

Pseudocode for iterative:

```
bool  Contains(  node,    key   )
            for each  char  ch  in  key
                  node = node.nodeArray  [  ch ]
                  if ( node is null )
                        return   false;

            return  node.EndWord                    // if we are here, we will return true if the end word flag is set.

void  Add(  node,    key   )                        // Add is called with root node, so we assume node is not null.
            for each  char  ch  in  key
                  if  node.nodeArray  [  ch ]   is null
                        node.nodeArray  [  ch ]  = new Node;  // this character path is new, so, create a new node ( note that we don't store ch )
                  node = node.nodeArray  [  ch ]

            Set endWord  flag on node
```

I will leave Delete as an exercise for u all

© Sanjeev Qazi 2021

Tries - LAB 1 of 2

- Write a Trie class that has the following functions:

  - bool Contains (string str)
  - void Add (string str)

- Add the following words to ur Trie:
  1. “washington”
  2. “washing”
  3. “washingmachine”
  4. “university”
  5. “washer”
  6. “web”
  7. “sanitation”
  8. “sanctuary”
  9. “water”

© Sanjeev Qazi 2021

Tries

- Some observations:

  - Structure of a trie is independent of the order in which keys are inserted.
    - This is not true of BST.

  - Trie does not store the keys, in fact, does not even store the key characters (since the character is implied from position of the corresponding node in array).
    - BST and Hash table store the keys.

- **Time complexity**:
  - Search miss:
    - will require <= L comparisons, where L is the length of the string being looked up.
    - There is a mathematical proof this is $\log_A N$, but u could just remember it is <= L comparisons.
  - Search hit:
    - L comparisons ➔ O (L)
  - Insertion: O (L)

© Sanjeev Qazi 2021

Tries

- **Space complexity:**

  - Worst case: O (L * A * N).
    - A is size of alphabet.
    - N is number of keys in the trie.
    - L here can be thought of as average key length.
    - First char of each key is different (this is worst case, probably unlikely), causing space to be L*A*N

    - Note: some texts will say A * N
      - where N is the number of nodes in the trie, but I think this is meaningful only for someone implementing the trie functionality. For others, N should represent number of keys.

  - Now, trie storage requirement is quite high, especially when u have **long keys**.
  - For long keys:
    - u can imagine last few characters in long keys will have nodes with mostly nulls in their nodesArray.
    - The nodes towards the end could have only 1 pointer to next character and the rest as nulls (i.e. A – 1 nulls).

- Next slide: What are possible values of A?

© Sanjeev Qazi 2021

Tries

- **Space complexity:**

  - Worst case: O (L * A * N).

  - What are possible values of A:

    - Just the English alphabet: 26 (case insensitive)
    - Any ASCII character: 256
    - Unicode: 65536 or more?

  - Tries are great if u have :
    - Lot of short words.
    - Proper distribution of words across the alphabet space.
  - That would reduce the nulls.
    - i.e., the node pointer array would **<u>not</u>** be sparse…leading to better space utilization

  - Alternative is ternary search tries (TST). These are space efficient.

© Sanjeev Qazi 2021

Tries

- What are ur thoughts on storage taken up by tries?

  - Space consumption pros and cons compared to:
    - BST
    - Hash table

    - Storing all words (BST, Hash table) vs only chars
      - Think of it like all keys are distributed across the trie (a node in the trie represents only one character of a key)
    - Null entries in array

  - Other pros and cons:
    - Collisions
    - Hash function
    - Look up all strings starting with a prefix.
    - All keys with the same prefix share storage.
    - Cache locality
    - Regeneration of hashes on growth (expensive)

© Sanjeev Qazi 2021

Tries, Trees and Hash tables

- Tries:
  - Lots of short keys
  - Faster when key search unsuccessful (as u can determine early without processing whole key)
  - U care about prefixes
  - Longest prefix functionality needed.
  - Spell check scenario
  - Key order matters (for listing them, not for insertion)
  - Alphabet size is not big.
  - Compares are character based.
  - Balancing is not a thing here.

- Hash table:
  - U care only about lookup.
  - Key order does not matter.
  - Compares are key based.
  - U have a good hash function.
  - Needs to be rebuilt (rehashing) when growing.

- BST:
  - Key order matters.
  - Key sizes may be long.
  - Compares are key based.
  - Does not need to be a great distribution (based on key length)

Tries - LAB 2 of 2

- In ur Trie class, add the following functions:

  - Collection of string   GetWordsWithPrefix (string prefix)
  - Collection of string   GetStringsWithPrefix (string prefix)

- Call the functions like this and have them return a collection of strings:

  - GetWordsWithPrefix(“wash”)
  - GetWordsWithPrefix(“washing”)
  - GetWordsWithPrefix(“u”)
  - GetWordsWithPrefix(“”)
  - GetStringsWithPrefix (“”)

© Sanjeev Qazi 2021

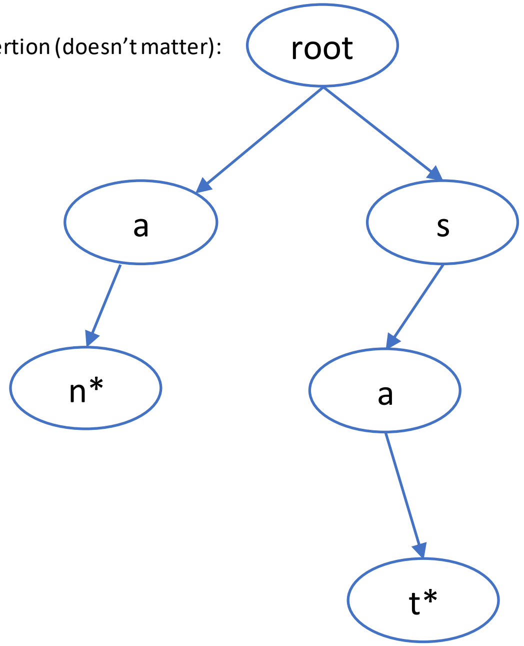
Ternary Search Trie aka TST

- In a trie, the number of possible links emanating from a node is the size of alphabet (A).
- However, in a TST, there are 3 links (and every node does have a character stored).
  - Left link
  - Middle link
  - Right link

Trie:

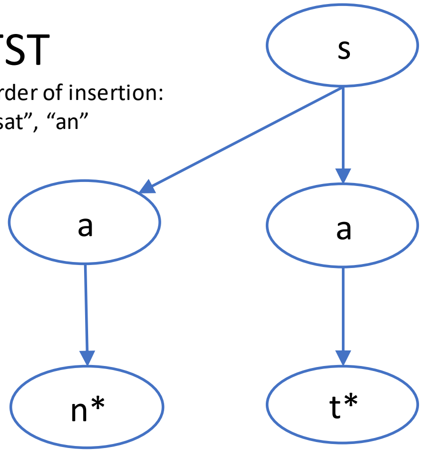
Order of insertion (doesn't matter):

"sat", "an"
OR
"an", "sat"



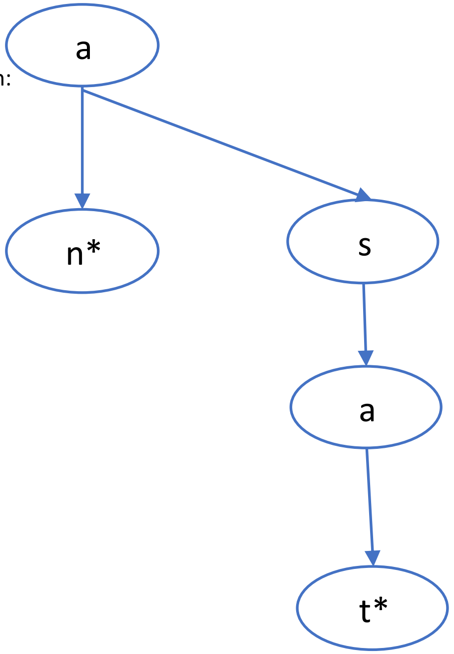
TST

Order of insertion:
"sat", "an"



TST

Order of insertion:
"an", "sat"



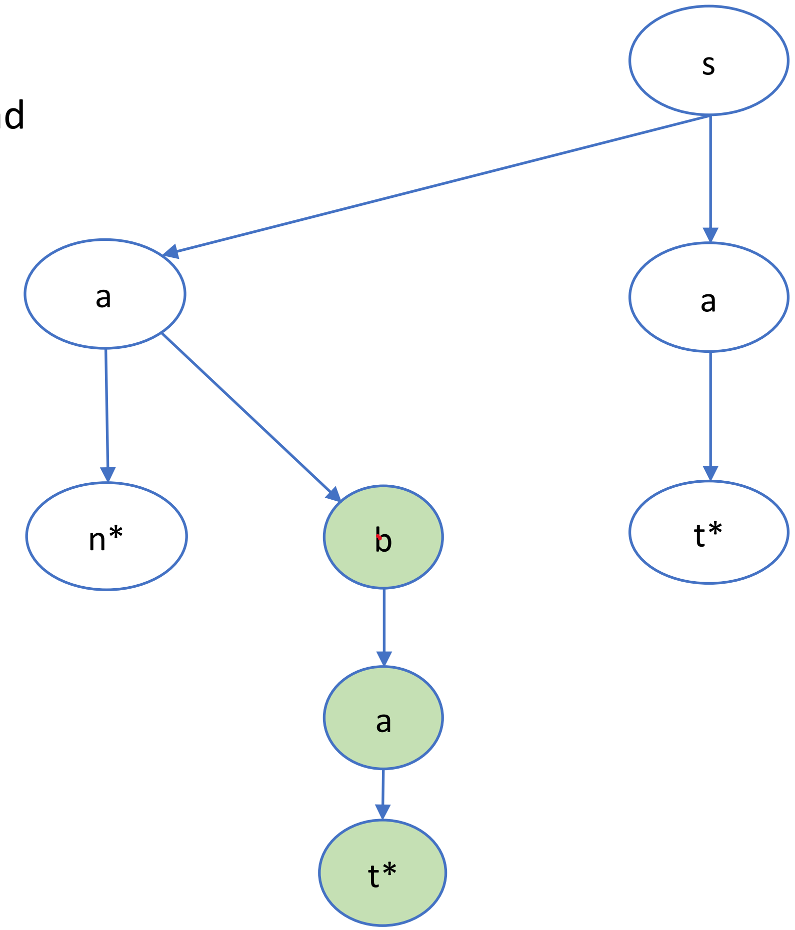
© Sanjeev Qazi 2021

Ternary Search Trie aka TST

- In a trie, the number of possible links emanating from a node is the size of alphabet (A).
- However, in a TST, there are 3 links (and every node does have a character stored).
  - Left link
  - Middle link
  - Right link

Add “bat” to TST we had

Order of insertion:
“sat”, “an” “bat”

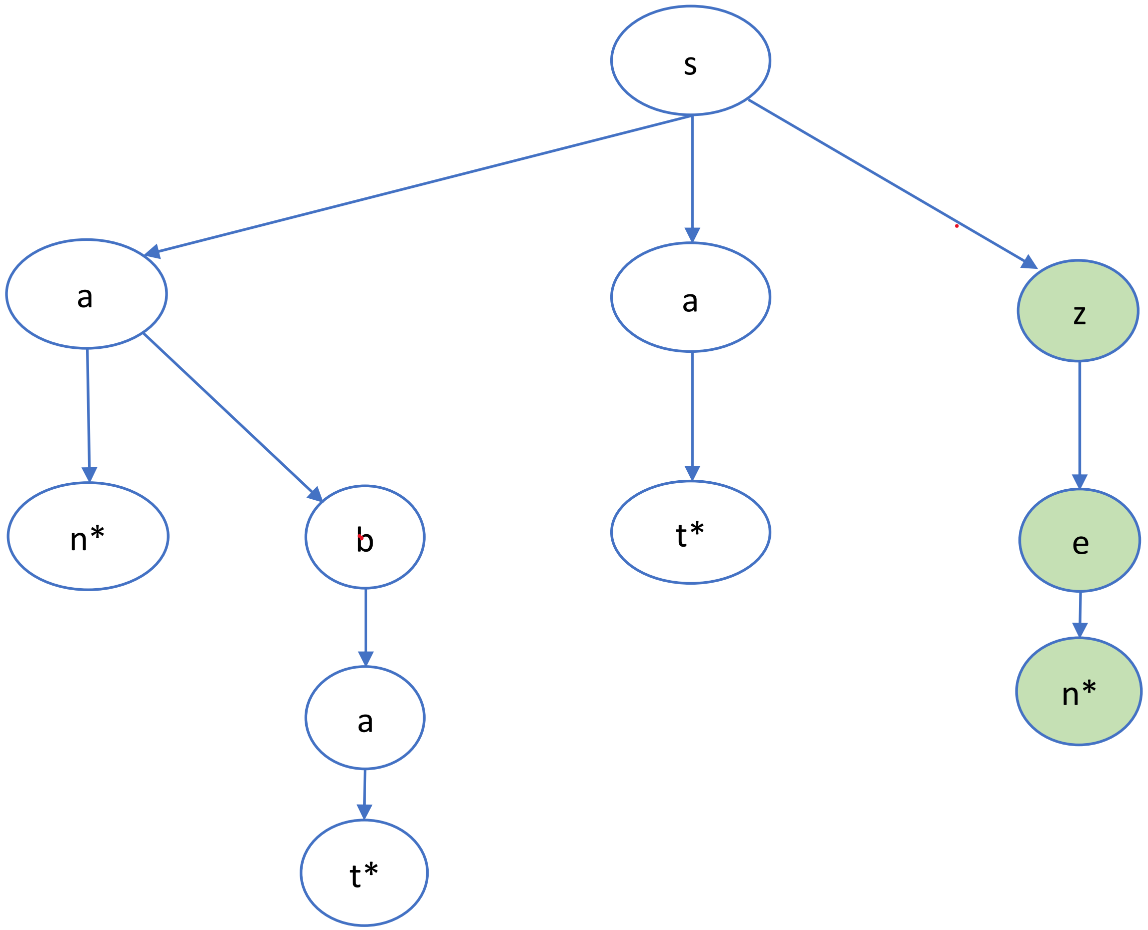


Ternary Search Trie aka TST

- In a trie, the number of possible links emanating from a node is the size of alphabet (A).
- However, in a TST, there are 3 links (and every node does have a character stored).
  - Left link
  - Middle link
  - Right link

Add “zen” to TST we had

Order of insertion:
“sat”, “an”, “bat”, “zen”



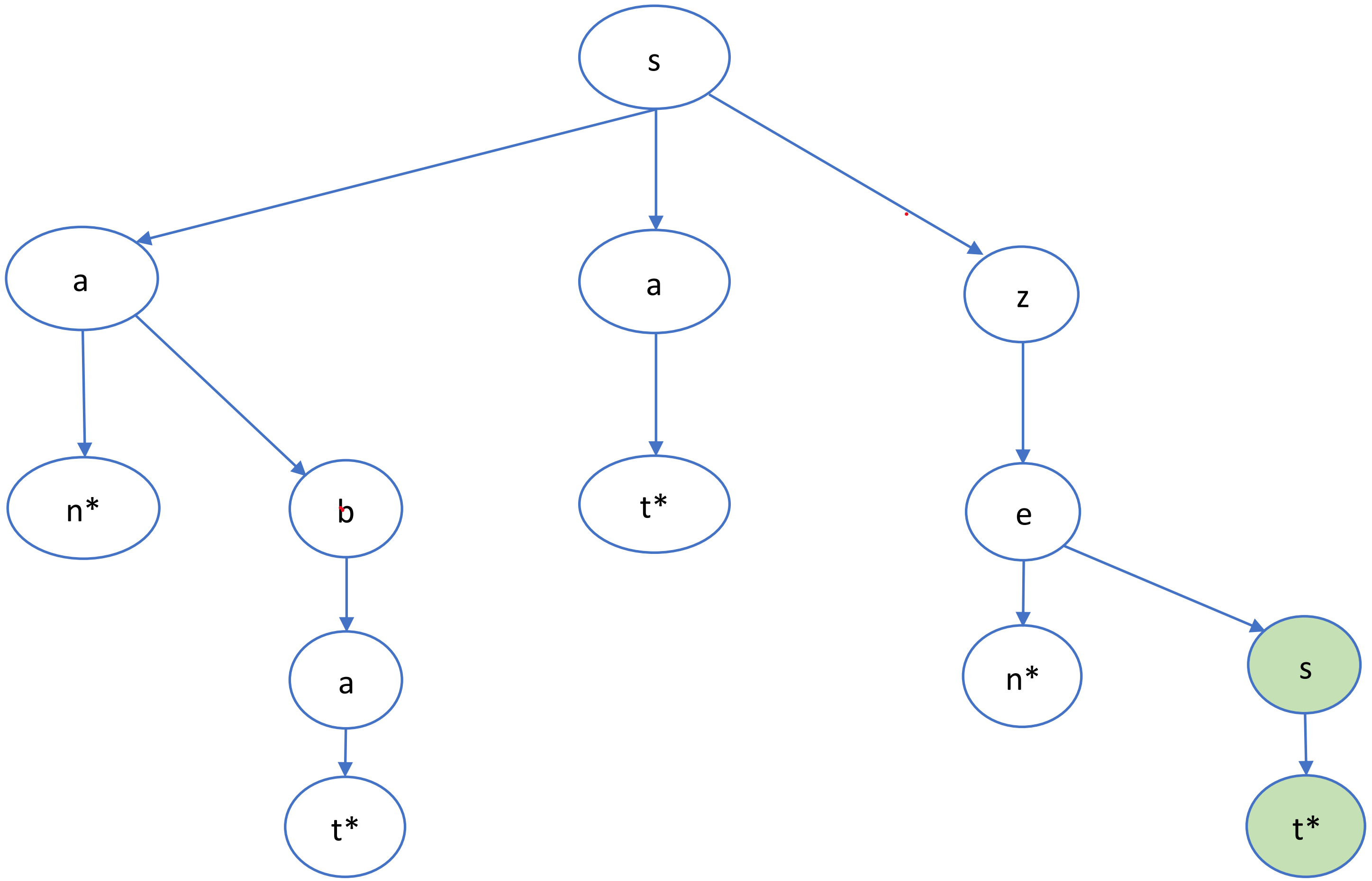
© Sanjeev Qazi 2021

Ternary Search Trie aka TST

- In a trie, the number of possible links emanating from a node is the size of alphabet (A).
- However, in a TST, there are 3 links (and every node does have a character stored).
  - Left link
  - Middle link
  - Right link

Add “zest” to TST we had

Order of insertion:
“sat”, “an”, “bat”, “zen”, “zest”



TST

- As u can see, TSTs are more space efficient than tries.

- When searching for a key:
  - Compare the current node's character with character of the key (key[i]):
    - if the key character is < node's character
      - Go down left link
    - else if the key character is > node's character
      - Go down right link
    - else
      - Go down middle link
      - **Advance** to next character of the key.
        - **Note**: we do this only when following the middle link

TST

Adding a key to a TST.   Call Add on root. Note that root can be null.

```
Add ( root,  key,  keyCharIndex )
      if  root == null
          root = new TSTNode( key [ keyCharIndex ] )

    for each  char  ch  in  key
          if  ch <  root.character
                Add ( root.left, key, keyCharIndex )
          else if  ch  >  root.character
                Add ( root.right, key, keyCharIndex )
          else   if ( keyCharIndex   is not   last character )
                Add ( root.middle, keyCharIndex + 1 )
```

See code for Search function

TST

- Ternary Search Trees:

  - Solve the space waste problem of Tries, space efficient like BST.
  - Searches are still character based, so efficiency of tries.
  - Keys in sorted order.
  - Partial match and near neighbor searches.
  - Search speed is similar to hash tables.