# Tries

- Red-Black BST
- Hash Table

* can we do better? — Yes, if we can avoid examining the entire key

## String Symbol Table

### class StringST

- void put (String key, Value val)
- Value get (String key)
- void delete (String key)

## R-Way Tries

**Tries** (from retrieval)

- for now store characters in nodes, not keys
- each node has R children, one for each possible character

◦ store values in nodes corresponding to last characters in keys

* search hit: node where search ends has a non-null value

* search miss: reach null link or node where search ends has null value

* insertion: follow links corresponding to each character in the key

    - encounter a null link: create new node

    - encounter last character of the key: set value in that node

---

```
private static class Node
{   private Object value;
    private Node[] next = new Node[R];
}
```

* characters are implicitly defined by link index

Performance

- search hit: need to examine all L characters
- search miss: examine only a few characters (sublinear)

Space : R null links at each leaf

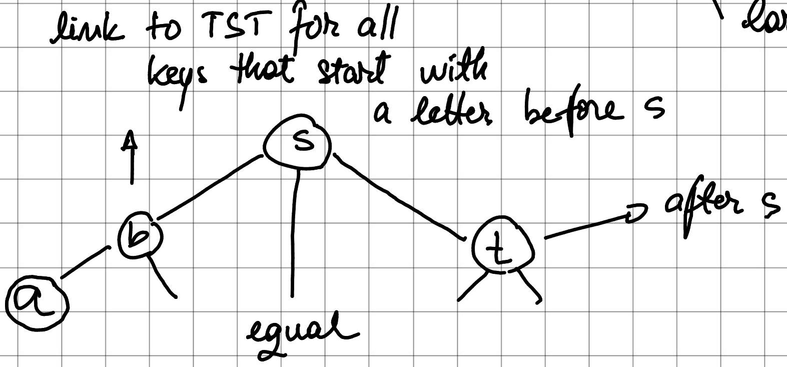
* data structure to perform spell checking

Delete : find node corresponding to key and set value to null

- if node has all null links, remove that node (and recur)

# Ternary Search Tries

- Store characters and values in nodes (not keys)
- each node has 3 children:
  - smaller (left)
  - equal (middle)
  - larger (right)



## Search in TST

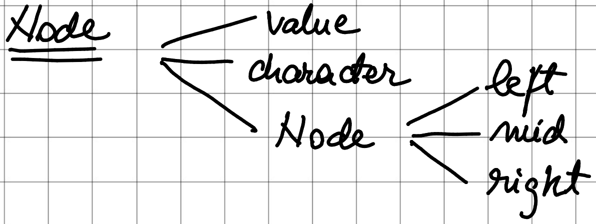
- follow links corresponding to each character in key
  - if less take left link, if greater take right link
  - if equal, take the middle link and move to the next key character

* search hit = node where search end has a non-null value

* search miss = reach a null link or node where search ends has a null value.

Key = a sequence of characters from root to value using middle links

Java



* can build balanced TSTs via rotations to achieve L + logN worst-case guarantees

Practice → hybrid of R-way trie and TST

- do $R^2$-way branching at root
- each of $R^2$ root nodes points to a TST

* HASHING : does not support ordered symbol table operations

## Character-based operations

- * prefix match
- * wildcard match
- ○ longest prefix

## String symbol table API

```
Iterable < String >  keys ()
Iterable < string >  keysWithPrefix (String s)
Iterable < String >  keysThatMatch (String s)
String  longestPrefixOf (String s)
```

- ○ to iterate through all keys in sorted order: do inorder traversal of trie (left, middle, right) => add keys encountered to a queue
- ○ maintain sequence of characters on path from root to node

* Prefix matches = autocomplete on the phone (eg)

- prefix matches in a R-way trie
  - → find subtrie for all keys beginning with prefix
  - → collect keys in that subtrie

Longest prefix

- find longest key in symbol table that is a prefix of a query string
  - → search for query string
  - → keep track of longest key encountered