# Tries

- Data structure for searching with string keys, similar to BST/Red-black and hash tables.
- From word "retrieval", but read as "try" to be different than "tree.

## Symbol Tables

- Generic ST

```
public class ST<Key extends Comparable<Key>, Value>implements Iterable<Key> {
        private TreeMap<Key, Value> st;
        public void put(Key key, Value val) {…}
        public void delete(Key key) {…}
        public Value get(Key key) {…}
}

//StringST

public class StringST<Value> {
    public void put(String key, Value val) {…}
    public void delete(String key) {…}
    public Value get(String key) {…}
}
```

## Tries examples

- Store characters in the nodes (not keys).
- Each node has R children, one for each possible character.
- Follow links corresponding to each character in the key.
  - **Search hit:** node where search ends has a non-null value.
  - **Search miss:** each null link or node where search ends has null value.

### Nodes

A value, plus references to R nodes.

```java
public class TrieST<Value>{

    private static final int R = 256;
    private Node root = new Node();
    private static class Node{
    private Object values;
    private Node[] next = new Node[R];
    }
    public void put(String key, Value val){
        root = put(root, key, val, 0);
        }
    private Node put(Node x, String key, Value val, int d){
        if (x == null) x = new Node();
        if (d == key.length()) { x.val = val; return x; }
        char c = key.charAt(d);
        x.next[c] = put(x.next[c], key, val, d+1);
        return x;
    }

    public boolean contains(String key){
        return get(key) != null;
        }
    public Value get(String key){
        Node x = get(root, key, 0);
        if (x == null) return null;
        return (Value) x.val;
    }
    private Node get(Node x, String key, int d){
        if (x == null) return null;
        if (d == key.length()) return x;
        char c = key.charAt(d);
        return get(x.next[c], key, d+1);
    }
}
```

- **Search hit** Need to examine all L equality.
- **Search miss**
    - Could have mismatch on first.
    - Typical case: examine only a few characters (sublinear)
- **Space** R null links at each leaf (but sublinear space possible if many short strings share common prefixes)
- **Bottom line** Fast search hit and even faster search miss, but wastes space.

SEEEEE LECTURES FOR ALL THE DIAGRAMS (TRIES FINAL)

## TST

SEEEE LECTURES FOR THE DIAGRAMS

#### Search a TST

Follow links corresponding to each character in the key.

- If less, take left link; if greater, take right link.
- If equal, take the middle like and move to the next key character.

**Search hit.** Node where seach ends has a non-null value.
**Search miss.** Reach a null like or node where search ends has a null value.

## TST vs. Hashing

**Hashing**:

- Need to examine entire key.
- Search hits and misses cost about the same.

- Performance relies on hash function.
- Does not support ordered symbol table operations.

**TSTs**:

- Works only for string (or digital) keys.
- Only examines just enough key characters.
- Search miss may involve only a few characters.
- Supports ordered symbol table operations (plus extras!)

**Bottom line**. TSTs are :

- Faster than hashing
- More flexible then red-black BSTs
    - supports character based operations

- The string symbol table API supports several useful character-based operations.
- Keys with prefix sh: *she, shells* and *shore*.
- Keys that match .he: *she* and *the*.
- Key that is the longest prefix of shellsort: *shells*.

To iterate through all keys in sorted order:

- Do inorder traversal of *trie*; add keys encountered to a queue.
- Maintain sequence of characters on path from root to node.