Tries

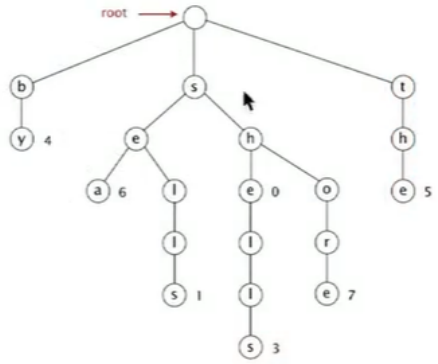
> Faster than hashing, more flexible than BSTs

R-way Tries

> from retreieval, but pronounced "try"

Structure

- 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



Operations

- search
  - 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
  - encounter a null link---create new node
  - encounter the last character of the key---set value in that node
- deletion
  - find the node corresponding to key and set value to null
  - If node has null value and all null links, remove that node and recur

Performance

- search
  - search hit
    - need to examine all L characters for equality
  - search miss
    - could have mismatch on first character

        - typical cases: examine only a few characters (sublinear)
- space
    - R null links at each leaf
    - sublinear space possible if many short srings share common prefixes
- Bottom line
    - fast search hit and even faster search miss, but waste space
- challenge
    - use less memory

Implementation

```
public class TrieST<Value> {
    private static final int R = 256;        // extended ASCII


    private Node root;      // root of trie
    private int n;          // number of keys in trie

    // R-way trie node
    private static class Node {
        private Object val;
        private Node[] next = new Node[R];
    }

    public TrieST() {
    }

    public Value get(String key) {
        if (key == null) throw new IllegalArgumentException("argument to get() is null");
        Node x = get(root, key, 0);
        if (x == null) return null;
        return (Value) x.val;
    }

    public boolean contains(String key) {
        if (key == null) throw new IllegalArgumentException("argument to contains() is null");
        return get(key) != null;
    }

    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);
    }

```

```
    public void put(String key, Value val) {
        if (key == null) throw new IllegalArgumentException("first argument to put() is null");
        if (val == null) delete(key);
        else 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()) {
            if (x.val == null) n++;
            x.val = val;
            return x;
        }
        char c = key.charAt(d);
        x.next[c] = put(x.next[c], key, val, d+1);
        return x;
    }

    public int size() { return n; }

    public boolean isEmpty() { return size() == 0; }

    public Iterable<String> keys() { return keysWithPrefix(""); }

    private void collect(Node x, StringBuilder prefix, Queue<String> results) {
        if (x == null) return;
        if (x.val != null) results.enqueue(prefix.toString());
        for (char c = 0; c < R; c++) {
            prefix.append(c);
            collect(x.next[c], prefix, results);
            prefix.deleteCharAt(prefix.length() - 1);
        }
    }

    private void collect(Node x, StringBuilder prefix, String pattern, Queue<String> results) {
        if (x == null) return;
        int d = prefix.length();
        if (d == pattern.length() && x.val != null)
            results.enqueue(prefix.toString());
        if (d == pattern.length())
            return;
        char c = pattern.charAt(d);
        if (c == '.') {
            for (char ch = 0; ch < R; ch++) {
                prefix.append(ch);
                collect(x.next[ch], prefix, pattern, results);
                prefix.deleteCharAt(prefix.length() - 1);
            }
        }
```

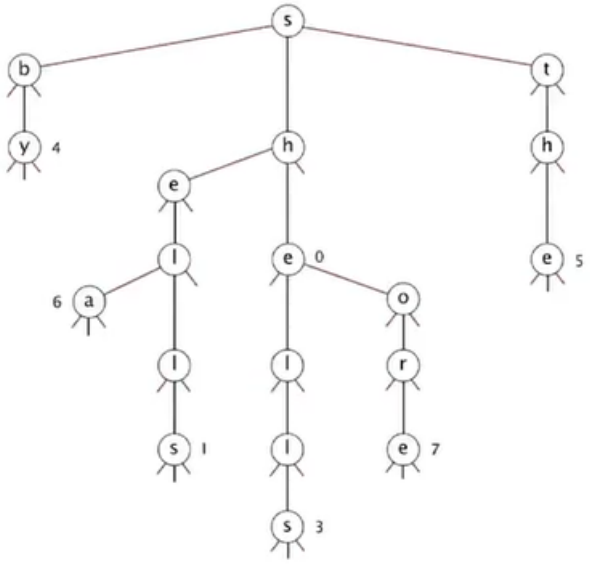
```
        else {
            prefix.append(c);
            collect(x.next[c], prefix, pattern, results);
            prefix.deleteCharAt(prefix.length() - 1);
        }
    }

    public void delete(String key) {
        if (key == null) throw new IllegalArgumentException("argument to
delete() is null");
        root = delete(root, key, 0);
    }

    private Node delete(Node x, String key, int d) {
        if (x == null) return null;
        if (d == key.length()) {
            if (x.val != null) n--;
            x.val = null;
        }
        else {
            char c = key.charAt(d);
            x.next[c] = delete(x.next[c], key, d+1);
        }

        // remove subtrie rooted at x if it is completely empty
        if (x.val != null) return x;
        for (int c = 0; c < R; c++)
            if (x.next[c] != null)
                return x;
        return null;
    }

    public static void main(String[] args) {

        // build symbol table from standard input
        TrieST<Integer> st = new TrieST<Integer>();
        for (int i = 0; !StdIn.isEmpty(); i++) {
            String key = StdIn.readString();
            st.put(key, i);
        }

        // print results
        if (st.size() < 100) {
            StdOut.println("keys(\"\"):");
            for (String key : st.keys()) {
                StdOut.println(key + " " + st.get(key));
            }
            StdOut.println();
        }

        StdOut.println("longestPrefixOf(\"shellsort\"):");
        StdOut.println(st.longestPrefixOf("shellsort"));
```

```
        StdOut.println();

        StdOut.println("longestPrefixOf(\"quicksort\"):");
        StdOut.println(st.longestPrefixOf("quicksort"));
        StdOut.println();

        StdOut.println("keysWithPrefix(\"shor\"):");
        for (String s : st.keysWithPrefix("shor"))
            StdOut.println(s);
        StdOut.println();

        StdOut.println("keysThatMatch(\".he.l.\"):");
        for (String s : st.keysThatMatch(".he.l."))
            StdOut.println(s);
    }
}
```

Ternary Search Tries

Structure

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



Implementation

```
public class TST<Value> {
    private int n;              // size
    private Node<Value> root;   // root of TST

    private static class Node<Value> {
        private char c;                        // character
        private Node<Value> left, mid, right;  // left, middle, and right
subtries
        private Value val;                     // value associated with string
```

```
    }

    public TST() {}

    public int size() { return n; }

    public boolean contains(String key) {}

    public Value get(String key) {}

    // return subtrie corresponding to given key
    private Node<Value> get(Node<Value> x, String key, int d) {
        if (x == null) return null;
        if (key.length() == 0) throw new IllegalArgumentException("key must
have length >= 1");
        char c = key.charAt(d);
        if      (c < x.c)              return get(x.left,  key, d);
        else if (c > x.c)              return get(x.right, key, d);
        else if (d < key.length() - 1) return get(x.mid,   key, d+1);
        else                           return x;
    }

    public void put(String key, Value val) {}

    private Node<Value> put(Node<Value> x, String key, Value val, int d) {
        char c = key.charAt(d);
        if (x == null) {
            x = new Node<Value>();
            x.c = c;
        }
        if      (c < x.c)               x.left  = put(x.left,  key, val, d);
        else if (c > x.c)               x.right = put(x.right, key, val, d);
        else if (d < key.length() - 1)  x.mid   = put(x.mid,   key, val, d+1);
        else                            x.val   = val;
        return x;
    }

    public Iterable<String> keys() {
        Queue<String> queue = new Queue<String>();
        collect(root, new StringBuilder(), queue);
        return queue;
    }

    // all keys in subtrie rooted at x with given prefix
    private void collect(Node<Value> x, StringBuilder prefix, Queue<String>
queue) {
        if (x == null) return;
        collect(x.left,  prefix, queue);
        if (x.val != null) queue.enqueue(prefix.toString() + x.c);
        collect(x.mid,   prefix.append(x.c), queue);
        prefix.deleteCharAt(prefix.length() - 1);
        collect(x.right, prefix, queue);
```

```
    }

    private void collect(Node<Value> x, StringBuilder prefix, int i, String
pattern, Queue<String> queue) {
        if (x == null) return;
        char c = pattern.charAt(i);
        if (c == '.' || c < x.c) collect(x.left, prefix, i, pattern, queue);
        if (c == '.' || c == x.c) {
            if (i == pattern.length() - 1 && x.val != null)
queue.enqueue(prefix.toString() + x.c);
            if (i < pattern.length() - 1) {
                collect(x.mid, prefix.append(x.c), i+1, pattern, queue);
                prefix.deleteCharAt(prefix.length() - 1);
            }
        }
        if (c == '.' || c > x.c) collect(x.right, prefix, i, pattern, queue);
    }

    public static void main(String[] args) {}
}
```

Patricia Tries

> practical algorithm to retrieve information coded in alphanumeric

Structure

- remove one-way branching
- each node represents a sequence of characters

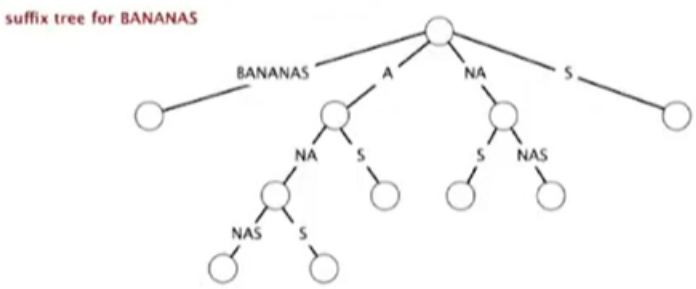
Application

- database search
- P2P network search
- IP routing tables: find longest prefix match
- compressed quad-tree for N-body simulation
- efficiently storing and querying XML documents

Suffix Tree

Structure

- Patricia trie of suffixes of a string
- Linear-time construction: beyond this course



Comparison

TST vs. R-way T

	character accesses (typical case)				dedup	
implementation	search hit	search miss	insert	space (references)	moby.txt	actors.txt
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	4 N	1.40	97.4
hashing (linear probing)	L	L	L	4 N to 16 N	0.76	40.6
R-way trie	L	$\log_R N$	L	(R + 1) N	1.12	out of memory
TST	$L + \ln N$	$\ln N$	$L + \ln N$	4 N	0.72	38.7

TST vs. hashing

- hashing
  - need to examine entire key
  - search hits and misses cost about the same
  - performance relies on hash function
  - doesn't support ordered symbol table operations
- TST
  - work only for strings (or digital keys)
  - only examines just enough key characters
  - search miss may involve only a few characters
  - supports ordered symbol tabe operations (plus others)
- Bottom line
  - TST is faster than hashing (especially for search misses)
  - TST is more flexible than red-black BSTs

Character-based Operations

prefix match

R-way trie

```
    public Iterable<String> keysWithPrefix(String prefix) {
        Queue<String> results = new Queue<String>();
        Node x = get(root, prefix, 0);
        collect(x, new StringBuilder(prefix), results);
        return results;
    }
```

Ternary Search Tries

```
     public Iterable<String> keysWithPrefix(String prefix) {
         if (prefix == null) {
             throw new IllegalArgumentException("calls keysWithPrefix() with
null argument");
         }
         Queue<String> queue = new Queue<String>();
         Node<Value> x = get(root, prefix, 0);
         if (x == null) return queue;
         if (x.val != null) queue.enqueue(prefix);
         collect(x.mid, new StringBuilder(prefix), queue);
         return queue;
     }
```

Wildcard match

R-way trie

```
    public Iterable<String> keysThatMatch(String pattern) {
        Queue<String> results = new Queue<String>();
        collect(root, new StringBuilder(), pattern, results);
        return results;
    }
```

Ternary Search Tries

```
    public Iterable<String> keysThatMatch(String pattern) {
        Queue<String> queue = new Queue<String>();
        collect(root, new StringBuilder(), 0, pattern, queue);
        return queue;
    }
```

Longest prefix

R-way trie

```
    private int longestPrefixOf(Node x, String query, int d, int length) {
        if (x == null) return length;
        if (x.val != null) length = d;
        if (d == query.length()) return length;
        char c = query.charAt(d);
        return longestPrefixOf(x.next[c], query, d+1, length);
    }
```

Ternary Search Tries

```
    public String longestPrefixOf(String query) {
        if (query == null) {
            throw new IllegalArgumentException("calls longestPrefixOf() with
null argument");
        }
        if (query.length() == 0) return null;
        int length = 0;
        Node<Value> x = root;
        int i = 0;
        while (x != null && i < query.length()) {
            char c = query.charAt(i);
            if      (c < x.c) x = x.left;
            else if (c > x.c) x = x.right;
            else {
                i++;
                if (x.val != null) length = i;
                x = x.mid;
            }
        }
        return query.substring(0, length);
    }
```