Today

- Tries
- R-way tries
- Ternary search tries
- Character-based operations

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

Review: summary of the performance of symbol-table implementations

Order of growth of the frequency of operations.

<table>
<thead>
<tr>
<th>implementation</th>
<th>typical case</th>
<th>ordered operations</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
</tr>
<tr>
<td>red-black BST</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
<tr>
<td>hash table</td>
<td>1 †</td>
<td>1 †</td>
<td>1 †</td>
</tr>
</tbody>
</table>

Q. Can we do better?
A. Yes, if we can avoid examining the entire key, as with string sorting.

Goal. Faster than hashing, more flexible than BSTs.
### String symbol table implementations cost summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>search hit</th>
<th>search miss</th>
<th>insert</th>
<th>space (references)</th>
<th>file</th>
<th>size</th>
<th>words</th>
<th>distinct</th>
</tr>
</thead>
<tbody>
<tr>
<td>red-black RST</td>
<td>L + c \lg^2 N</td>
<td>c \lg^2 N</td>
<td>c \lg^3 N</td>
<td>4N</td>
<td>moby.txt</td>
<td>1.2 MB</td>
<td>210 K</td>
<td>32 K</td>
</tr>
<tr>
<td>hashing (linear probing)</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>4N to 16N</td>
<td>actors.txt</td>
<td>82 MB</td>
<td>11.4 M</td>
<td>900 K</td>
</tr>
</tbody>
</table>

**Parameters**
- N = number of strings
- L = length of string
- R = radix

### Challenge
Efficient performance for string keys.

### Tries

**Tries.** [from retrieval, but pronounced "try"]
- 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.

Follow links corresponding to each character in the key.
- **Search hit**: node where search ends has a non-null value.
- **Search miss**: reach a null link or node where search ends has null value.

**get("shells")**

root

by 4
sea 6
sells 1
she 0
shells 3
shore 7
the 5

for now, we do not draw null links

link to trie for all keys that start with a

value for she in node corresponding to last key character

link to trie for all keys that start with she

return value associated with last key character (return 3)
Search in a trie

Follow links corresponding to each character in the key.
- Search hit: node where search ends has a non-null value.
- Search miss: reach a null link or node where search ends has null value.

get("she")

Insertion into a trie

Follow links corresponding to each character in the key.
- Encounter a null link: create new node.
- Encounter the last character of the key: set value in that node.

put("shore", 7)
Trie construction demo

Trie construction demo

Trie construction demo

Trie construction demo
she
put("sells", 1)

she
sells
trie

she
sells
trie

put("sea", 2)
Trie construction demo

she
sells
sea
trie

Trie construction demo

she
sells
sea
put("she", 3)

Trie construction demo

she
sells
sea
put("shells", 3)

Trie construction demo

she
sells
sea
put("by", 4)
Trie construction demo

she
sells
sea
by
trie

Trie construction demo

she
sells
sea
by
put("the", 5)

Trie construction demo

she
sells
sea
by
put("sea", 6)

Trie construction demo

overwrite
old value with
new value
she
sells
sea
by
the
put("shore", 7)

she
sells
sea
by
the
shore
trie
Trie representation: Java implementation

Node. A value, plus references to $R$ nodes.

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

A child node for each character in Alphabet. No need to search for character, but a pointer reserved for each character in memory.

R-way trie: Java implementation

```java
public class TrieST<Value> {
    private static final int R = 256;
    private Node root;

    private static class Node {
        /* see previous slide */
    }

    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);
    }
}
```

R-way trie: Java implementation (continued)

```java
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);
}
```

Extended ASCII
Trie performance

Search hit. Need to examine all $L$ characters for equality.

Search miss.
- Could have mismatch on first character.
- 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.

Deletion in an R-way trie

To delete a key-value pair:
- Find the node corresponding to key and set value to null.
- If that node has all null links, remove that node (and recur).

String symbol table implementations cost summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>character accesses (typical case)</th>
<th>dedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search hit</td>
<td>search miss</td>
</tr>
<tr>
<td>red-black BST</td>
<td>$L + c \log^2 N$</td>
<td>$c \log^2 N$</td>
</tr>
<tr>
<td>hashing (linear probing)</td>
<td>$L$</td>
<td>$L$</td>
</tr>
<tr>
<td>R-way trie</td>
<td>$L$</td>
<td>$\log R N$</td>
</tr>
</tbody>
</table>

R-way trie.
- Method of choice for small $R$.
- Too much memory for large $R$.

Challenge. Use less memory, e.g., 65,536-way trie for Unicode!
Digression: out of memory?

“640 K ought to be enough for anybody.”
— (mis)attributed to Bill Gates, 1981
(commenting on the amount of RAM in personal computers)

“64 MB of RAM may limit performance of some Windows XP features; therefore, 128 MB or higher is recommended for best performance.”
— Windows XP manual, 2002

“64 bit is coming to desktops, there is no doubt about that. But apart from Photoshop, I can’t think of desktop applications where you would need more than 4GB of physical memory, which is what you have to have in order to benefit from this technology. Right now, it is costly.”
— Bill Gates, 2003

Digression: out of memory?

A short (approximate) history.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Year</th>
<th>Address Bits</th>
<th>Addressable Memory</th>
<th>Typical Actual Memory</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDP-8</td>
<td>1960s</td>
<td>12</td>
<td>6 KB</td>
<td>6 KB</td>
<td>$16K</td>
</tr>
<tr>
<td>PDP-10</td>
<td>1970s</td>
<td>18</td>
<td>256 KB</td>
<td>256 KB</td>
<td>$1M</td>
</tr>
<tr>
<td>IBM S/360</td>
<td>1970s</td>
<td>24</td>
<td>4 MB</td>
<td>512 KB</td>
<td>$1M</td>
</tr>
<tr>
<td>VAX</td>
<td>1980s</td>
<td>32</td>
<td>4 GB</td>
<td>1 MB</td>
<td>$1M</td>
</tr>
<tr>
<td>Pentium</td>
<td>1990s</td>
<td>32</td>
<td>4 GB</td>
<td>1 GB</td>
<td>$1K</td>
</tr>
<tr>
<td>Xeon</td>
<td>2000s</td>
<td>64</td>
<td>enough</td>
<td>4 GB</td>
<td>$100K</td>
</tr>
<tr>
<td>?</td>
<td>future</td>
<td>128+</td>
<td>enough</td>
<td>enough</td>
<td>1 $</td>
</tr>
</tbody>
</table>

“512-bit words ought to be enough for anybody.”
— Kevin Wayne, 1995

Tries

- R-way Tries
- Ternary search tries
- Character-based operations

Ternary search tries

- Store characters and values in nodes (not keys).
- Each node has three children: smaller (left), equal (middle), larger (right).
Ternary search tries

- Store characters and values in nodes (not keys).
- Each node has three children: smaller (left), equal (middle), larger (right).

Search in 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 link and move to the next key character.

Search hit. Node where search ends has a non-null value.
Search miss. Reach a null link or node where search ends has null value.
Ternary search trie insertion demo

Ternary search trie

ternary search trie

Ternary search trie insertion demo

put("she", 0)

Ternary search trie insertion demo

key is sequence of characters from root to value using middle links
value is in node corresponding to last character

Ternary search trie insertion demo

put("she", 0)

Ternary search trie insertion demo

put("sells", 1)

Ternary search trie insertion demo

put("sells", 1)
Ternary search trie insertion demo

Ternary search trie

put("by", 4)

Ternary search trie insertion demo

Ternary search trie

put("the", 5)
Ternary search trie insertion demo

Ternary search trie

The Ternary search trie can be used to store and search for keys in a binary search tree. It is a type of trie that allows for efficient searching of keys.

When inserting a key into the trie, the key is traversed from the root to the leaf, and if the key is not already present, a new leaf is created with the key and the associated value.

For example, consider the insertion of the words "sea" and "shore" into the trie:

1. Insert "sea":
   - Traverse the trie from the root to the leaf, creating a new leaf if necessary.
   - The value associated with "sea" is set to 6.

2. Insert "shore":
   - Traverse the trie from the root to the leaf, creating a new leaf if necessary.
   - The value associated with "shore" is set to 7.

The diagrams illustrate these insertions, showing the path of traversal and the creation of new leaves.

Overwrite old value with new value

The "overwrite" operation allows for updating the value associated with a key in the trie. It involves finding the key in the trie and replacing the value at that node with the new value.

The diagrams demonstrate this operation by highlighting the path of traversal and the update of the value at the leaf node.
Ternary search trie insertion demo

Ternary search trie

26-way trie vs. TST

26-way trie. 26 null links in each leaf.

TST. 3 null links in each leaf.

TST representation in Java

A TST node is five fields:
- A value.
- A character c.
- A reference to a left TST.
- A reference to a middle TST.
- A reference to a right TST.

private class Node {
    private Value val;
    private char c;
    private Node left, mid, right;
}

Trie node representations
TST: Java implementation

```java
public class TST<Value> {
    private Node root;

    private class Node {
        /* see previous slide */
    }

    public void put(String key, Value val) {
        root = put(root, key, val, 0);
    }

    private Node put(Node x, String key, Value val, int d) {
        char c = key.charAt(d);
        if (x == null) { x = new Node(); 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;
    }
}
```

TST: Java implementation (continued)

```java
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 x.val;
}

private Node get(Node x, String key, int d) {
    if (x == null) return null;
    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;
}
```

String symbol table implementation cost summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Character accesses (typical case)</th>
<th>Dedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search hit</td>
<td>search miss</td>
</tr>
<tr>
<td>red-black BST</td>
<td>L + c lg² N</td>
<td>c lg² N</td>
</tr>
<tr>
<td>hashing (linear probing)</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>R-way trie</td>
<td>L</td>
<td>log² N</td>
</tr>
<tr>
<td>TST</td>
<td>L + ln N</td>
<td>ln N</td>
</tr>
</tbody>
</table>

Remark. Can build balanced TSTs via rotations to achieve $L + \log N$ worst-case guarantees.

Bottom line. TST is as fast as hashing (for string keys), space efficient.

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

Bottom line. TSTs are:
- Faster than hashing (especially for search misses).
- More flexible than red-black BSTs. [stay tuned]
Tries.

- Store characters in nodes (not keys).
- Each node has at most one child, corresponding to each possible character.
- Store values in nodes corresponding to last characters in keys.

Ternary Search Trees (TSTs)

- Store characters and values in nodes (not keys).
- Each node has three children: smaller (left), equal (middle), larger (right).

Previous on BBM202...

TST representation of a trie

String symbol table API

Character-based operations. The string symbol table API supports several useful character-based operations.

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>by</td>
<td>4</td>
</tr>
<tr>
<td>sea</td>
<td>6</td>
</tr>
<tr>
<td>sells</td>
<td>1</td>
</tr>
<tr>
<td>she</td>
<td>0</td>
</tr>
<tr>
<td>shells</td>
<td>3</td>
</tr>
<tr>
<td>shore</td>
<td>7</td>
</tr>
<tr>
<td>the</td>
<td>5</td>
</tr>
</tbody>
</table>

Prefix match. Keys with prefix "sh": "she", "shells", and "shore".

Wildcard match. Keys that match ".he": "she" and "the".

Longest prefix. Key that is the longest prefix of "shellsort": "shells".

Tries

- R-way tries
- Ternary search tries
- Character-based operations

String symbol table API

```java
public class StringST<Value>
{
    StringST()
    void put(String key, Value val)
    Value get(String key)
    void delete(String key)
    ...
    Iterable<String> keys()
    Iterable<String> keysWithPrefix(String s)
   Iterable<String> keysThatMatch(String s)
    String longestPrefixOf(String s)
}
```

Remark. Can also add other ordered ST methods, e.g., `floor()` and `rank()`.
Warmup: ordered iteration

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.

Ordered iteration: Java implementation

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.

Prefix matches

Find all keys in symbol table starting with a given prefix.
Ex. Autocomplete in a cell phone, search bar, text editor, or shell.
• User types characters one at a time.
• System reports all matching strings.

Prefix matches

Find all keys in symbol table starting with a given prefix.
**Wildcard matches**

Use wildcard . to match any character in alphabet.

```
co....er  .c....c.
coalizer acresce
coberger acroach
codifier accuracy
cofaster octarch
cofather science
cogmizer scranch
cohelper scrach
coleader scrauch
... scrinch
compiler scritch
... scudick
composer scudock
computer scutock
cowkeeper
```

**Longest prefix**

Find longest key in symbol table that is a prefix of query string.

**Ex.** To send packet toward destination IP address, router chooses IP address in routing table that is longest prefix match.

```
"128"
"128.112"
"128.112.055"
"128.112.055.15"
"128.112.136"
"128.112.155.11"
"128.112.155.13"
"128.222"
"128.222.136"
```

**Note.** Not the same as floor: \( \text{floor}("128.112.100.16") = "128.112.055.15" \)

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

private void collect(Node x, String prefix, String pat, Queue<String> q) {
    if (x == null) return;
    int d = prefix.length();
    if (d == pat.length() && x.val != null) q.enqueue(prefix);
    if (d == pat.length()) return;
    char next = pat.charAt(d);
    for (char c = 0; c < R; c++)
        if (next == '.' || next == c)
            collect(x.next[c], prefix + c, pat, q);
}
```

**Longest prefix**

Find longest key in symbol table that is a prefix of query string.

• Search for query string.
• Keep track of longest key encountered.

```
"she"
"shell"
"shellosort"
```

Possibilities for `longestPrefixOf()`
**Longest prefix: Java implementation**

Find longest key in symbol table that is a prefix of query string.
- Search for query string.
- Keep track of longest key encountered.

```java
public String longestPrefixOf(String query) {
    int length = search(root, query, 0, 0);
    return query.substring(0, length);
}

private int search(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 search(x.next[c], query, d+1, length);
}
```

**T9 texting**

**Goal.** Type text messages on a phone keypad.

**Multi-tap input.** Enter a letter by repeatedly pressing a key until the desired letter appears.

"a much faster and more fun way to enter text"

**T9 text input.**
- Find all words that correspond to given sequence of numbers.
- Press 0 to see all completion options.

**Ex.** hello
- Multi-tap: 4 4 3 3 5 5 5 5 5 5 6 6 6
- T9: 4 3 5 5 6

**Patricia trie**

**Patricia trie.** [Practical Algorithm to Retrieve Information Coded in Alphanumeric]
- Remove one-way branching.
- Each node represents a sequence of characters.
- Implementation: one step beyond this course.

**Applications.**
- 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.

Also known as: crit-bit tree, radix tree.

**Suffix tree**

**Suffix tree.**
- Patricia trie of suffixes of a string.
- Linear-time construction: beyond this course.

**Applications.**
- Linear-time: longest repeated substring, longest common substring, longest palindromic substring, substring search, tandem repeats, ….
- Computational biology databases (BLAST, FASTA).
**String symbol tables summary**

A success story in algorithm design and analysis.

**Red-black BST.**
- Performance guarantee: \( \log N \) key compares.
- Supports ordered symbol table API.

**Hash tables.**
- Performance guarantee: constant number of probes.
- Requires good hash function for key type.

**Tries.** R-way, TST.
- Performance guarantee: \( \log N \) characters accessed.
- Supports character-based operations.

**Bottom line.** You can get at anything by examining 50-100 bits (!!!)