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

- Tries
- R-way tries
TRIES

- R-way 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.

<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>
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")**

![Trie Diagram](image-url)
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")
```
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.

```c
get("shell")
```

![Trie diagram](image-url)
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("shelter")

![Diagram of a trie tree with nodes labeled with characters and weights, indicating the process of searching for "shelter" in the trie. The search path is marked with black lines, and the node where the search ends is highlighted. If there is no link to 't', the search returns null.](image-url)
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

put("she", 0)

key is sequence of characters from root to value
value is in node corresponding to last character
Trie construction demo

she
trie

```
s
h
e
0
```
Trie construction demo

```
she
put("sells", 1)
```
she
sells
trie

Trie construction demo
Trie construction demo

she
sells
trie
she
sells

put("sea", 2)
Trie construction demo

she
sells
sea
trie
Trie construction demo

she
sells
sea

put("shells", 3)
Trie construction demo

she
sells
sea
trie
Trie construction demo

she
sells
sea
put("by", 4)
she
sells
sea
by
trie
she
sells
sea
by

`put("the", 5)`
Trie construction demo

she
sells
sea
by
the
trie
Trie construction demo

```
put("sea", 6)
```

![Trie diagram]

- `bs`: Parent node
- `e`: Child node with value 6
- `a`: Child node with value 6
- `y`: Child node with value 4

- `t`: Parent node
- `h`: Child node with value 0
- `h`: Child node with value 5

- `l`: Child node with value 0
- `s`: Child node with value 3

The diagram shows the Trie construction process for the word "sea". Each node represents a letter, and the value associated with each node is highlighted.

**Overwrite Old Value with New Value**

- The insertion of "sea" overwrites the existing value (if any) at the corresponding path in the Trie.
Trie construction demo
Trie construction demo

 trie

trie

trie
Trie construction demo

she
sells
sea
by
the

put("shore", 7)
she
sells
sea
by
the
shore
trie
**Trie representation:** implementation

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

```
struct Node {
    int value;
    Node * next[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: implementation

```c
#define R 256

Node * root;

put(&root, key, val, 0);

void put(Node ** x, char *key, int val, int d)
{
    if (*x == null) *x = getNode();
    if (d == strlen(key)) { *x->value = val; return;}
    char c = key[d];
    put(&(x->next[c]), key, val, d+1);
}
```

...
Node * getNode(){
    Node * pNode = NULL;
    pNode = (Node *)malloc(sizeof(Node));
    if (pNode){
        for (int i = 0; i < R; i++)
            pNode->next[i] = NULL;
    }
    return pNode;
}
int get(Node * x, char * key, int d)
{
    if (x == null) return -1; // -1 refers no match
    if (d == strlen(key)) return x->value;
    char c = key[d];
    return get(x->next[c], key, d+1);
}

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.
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>
</tr>
</thead>
<tbody>
<tr>
<td>hashing (separate chaining)</td>
<td>N</td>
<td>N</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>R-way trie</td>
<td>L</td>
<td>log&lt;sub&gt;R&lt;/sub&gt;N</td>
<td>L</td>
<td>RNw</td>
</tr>
</tbody>
</table>

N = number of entries, L= key length, R= alphabet size, w= average key length

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!