Module 6: Dictionaries for special keys

T. Biedl  M. Petrick  O. Veksler
Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Winter 2019

References: Sedgewick 12.4, 15.2-15.4
Goodrich & Tamassia 23.5.1-23.5.2
Outline

1. Lower bound

2. Interpolation Search

3. Tries
   - Standard Tries
   - Variations of Tries
   - Compressed Tries
Lower bound for search

The fastest implementations of the dictionary ADT require $\Theta(\log n)$ time to search a dictionary containing $n$ items. Is this the best possible?

Theorem

In the comparison model (on the keys), $\Omega(\log n)$ comparisons are required to search a size-$n$ dictionary.

Proof:

But can we beat the lower bound for special keys?
Lower bound for search

The fastest implementations of the dictionary ADT require $\Theta(\log n)$ time to search a dictionary containing $n$ items. Is this the best possible?

**Theorem**: In the comparison model (on the keys), $\Omega(\log n)$ comparisons are required to search a size-$n$ dictionary.

**Proof**: But can we beat the lower bound for special keys?
Binary Search

Ordered array
- insert, delete: $\Theta(n)$
- search: $\Theta(\log n)$

Binary-search($A$, $n$, $k$)
$A$: Array of size $n$, $k$: key
1. $\ell \leftarrow 0$
2. $r \leftarrow n - 1$
3. while ($\ell < r$)
4. $m \leftarrow \lfloor \frac{\ell + r}{2} \rfloor$
5. if ($A[m] < k$) $\ell = m + 1$
6. elseif ($k < A[m]$) $r = m - 1$
7. else return $m$
8. if ($k = A[\ell]$) return $\ell$
9. else return “not found, but would be between $\ell - 1$ and $\ell$”
Interpolation Search: Motivation

binary search\((A[\ell, r], k)\): Compare at index \([\frac{\ell + r}{2}] = \ell + \lfloor \frac{1}{2}(r - \ell)\rfloor\)
Interpolation Search: Motivation

binary search \((A[\ell, r], k)\): Compare at index \(\left\lfloor \frac{\ell + r}{2} \right\rfloor = \ell + \left\lfloor \frac{1}{2}(r - \ell) \right\rfloor\)

\[\begin{array}{c|c|c}
\ell & \downarrow & r \\
40 & & 120 \\
\end{array}\]

**Question**: If keys are numbers, where would you expect key \(k = 100\)?
Interpolation Search: Motivation

**binary search** ($A[\ell, r], k$): Compare at index $\lfloor \frac{\ell + r}{2} \rfloor = \ell + \lfloor \frac{1}{2} (r - \ell) \rfloor$

<table>
<thead>
<tr>
<th>$\ell$</th>
<th>$\downarrow$</th>
<th>$r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td></td>
<td>120</td>
</tr>
</tbody>
</table>

**Question**: If keys are numbers, where would you expect key $k = 100$?

**Interpolation Search** ($A[\ell, r], k$): Compare at index $\ell + \left\lfloor \frac{k - A[\ell]}{A[r] - A[\ell]} (r - \ell) \right\rfloor$
Interpolation Search Example

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>449</td>
<td>450</td>
<td>600</td>
<td>800</td>
<td>1000</td>
<td>1200</td>
<td>1500</td>
</tr>
</tbody>
</table>

Search(449):

Works well if keys are uniformly distributed:
Can show: the array in which we recurse into has expected size $\sqrt{n}$.
Recurrence relation is $T(\text{avg})(n) = T(\text{avg})(\sqrt{n}) + \Theta(1)$.
This resolves to $T(\text{avg})(n) \in \Theta(\log\log n)$.

But: Worst case performance $\Theta(n)$. 
Interpolation Search Example

Search(449):

- Initially \( \ell = 0, \ r = n - 1 = 10, \ m = \ell + \left\lfloor \frac{449 - 0}{1500 - 0} (10 - 0) \right\rfloor = \ell + 2 = 2 \)
Interpolation Search Example

Search(449):

- Initially $\ell = 0$, $r = n - 1 = 10$, $m = \ell + \left\lfloor \frac{449 - 0}{1500 - 0} (10 - 0) \right\rfloor = \ell + 2 = 2$
- $\ell = 3$, $r = 10$, $m = \ell + \left\lfloor \frac{449 - 3}{1500 - 3} (10 - 3) \right\rfloor = \ell + 2 = 5$
## Interpolation Search Example

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>449</td>
<td>450</td>
<td>600</td>
<td>800</td>
<td>1000</td>
<td>1200</td>
<td>1500</td>
<td></td>
</tr>
</tbody>
</table>

\[ \ell \quad \uparrow, r \]

**Search**(449):

- Initially \( \ell = 0, r = n - 1 = 10 \), \( m = \ell + \left\lfloor \frac{449 - 0}{1500 - 0} (10 - 0) \right\rfloor = \ell + 2 = 2 \)
- \( \ell = 3, r = 10 \), \( m = \ell + \left\lfloor \frac{449 - 3}{1500 - 3} (10 - 3) \right\rfloor = \ell + 2 = 5 \)
- \( \ell = 3, r = 4 \), \( m = \ell + \left\lfloor \frac{449 - 3}{449 - 3} (4 - 3) \right\rfloor = \ell + 1 = 4 \), found at \( A[4] \)
Interpolation Search Example

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>449</td>
<td>450</td>
<td>600</td>
<td>800</td>
<td>1000</td>
<td>1200</td>
<td>1500</td>
</tr>
</tbody>
</table>

**Search**(449):
- Initially $\ell = 0$, $r = n - 1 = 10$, $m = \ell + \lfloor \frac{449 - 0}{1500 - 0} (10 - 0) \rfloor = \ell + 2 = 2$
- $\ell = 3$, $r = 10$, $m = \ell + \lfloor \frac{449 - 3}{1500 - 3} (10 - 3) \rfloor = \ell + 2 = 5$
- $\ell = 3$, $r = 4$, $m = \ell + \lfloor \frac{449 - 3}{449 - 3} (4 - 3) \rfloor = \ell + 1 = 4$, found at $A[4]$

Works well if keys are uniformly distributed:
- Can show: the array in which we recurse into has expected size $\sqrt{n}$.
- Recurrence relation is $T^{(avg)}(n) = T^{(avg)}(\sqrt{n}) + \Theta(1)$.
- This resolves to $T^{(avg)}(n) \in \Theta(\log \log n)$.

But: Worst case performance $\Theta(n)$
Interpolation Search

- Code very similar to binary search, but compare at interpolated index
- Need a few extra tests to avoid crash due to $A[\ell] = A[r]

**Interpolation-search**($A, n, k$)

$A$: Array of size $n$, $k$: key

1. $\ell \leftarrow 0$
2. $r \leftarrow n - 1$
3. while $(\ell < r) \land (A[r] \neq A[\ell]) \land (k \geq A[\ell]) \land (k \leq A[r])$

4. $m \leftarrow \ell + \left\lfloor \frac{k - A[\ell]}{A[r] - A[\ell]} \cdot (r - \ell) \right\rfloor$

5. if $(A[m] < k)$  $\ell \leftarrow m + 1$
6. elsif $(A[m] = k)$ return $m$
7. else $r \leftarrow m - 1$
8. if $(k = A[\ell])$ return $\ell$
9. else return “not found, but would be between $\ell - 1$ and $\ell$”
Outline

1. Lower bound

2. Interpolation Search

3. Tries
   - Standard Tries
   - Variations of Tries
   - Compressed Tries
Tries: Introduction

- **Trie (Radix Tree):** A dictionary for binary strings
  - Comes from retrieval, but pronounced “try”
  - A tree based on **bitwise comparisons**
  - Similar to **radix sort:** use individual bits, not the whole key

- Keys can have different number of bits
Tries: Introduction

- **Trie (Radix Tree)**: A dictionary for binary strings
  - Comes from retrieval, but pronounced “try”
  - A tree based on **bitwise comparisons**
  - Similar to **radix sort**: use individual bits, not the whole key
- Keys can have different number of bits

**Prefix** of a string $S[0..n − 1]$: a substring $S[0..i]$ of $S$ for some $0 ≤ i ≤ n − 1$.

**Prefix-free**: there is no pair of binary strings in the dictionary where one is the prefix of the other.
Tries: Introduction

- Trie (Radix Tree): A dictionary for binary strings
  - Comes from retrieval, but pronounced “try”
  - A tree based on bitwise comparisons
  - Similar to radix sort: use individual bits, not the whole key

- Keys can have different number of bits

Prefix of a string $S[0..n-1]$: a substring $S[0..i]$ of $S$ for some $0 \leq i \leq n-1$.

Prefix-free: there is no pair of binary strings in the dictionary where one is the prefix of the other.

Assumption: Dictionary is prefix-free:

- This is always satisfied if all strings have the same length.
- This is always satisfied if all strings end with a special ‘end-of-word’ character $\$. 
Tries: structure

Structure of trie:
- Items (keys) are stored **only** in the leaf nodes
- Edge to child is labelled with corresponding bit or $\$

**Example**: A trie for 

$S = \{00\$, $0001\$, $01001\$, $011\$, $01101\$, $110\$, $1101\$, $111\$\}$

![Trie Diagram](image-url)
Tries: Search

- start from the root and the most significant bit of $x$
- follow the link that corresponds to the current bit in $x$; return failure if the link is missing
- return success if we reach a leaf (it must store $x$)
- else recurse on the new node and the next bit of $x$

\[
\text{Trie-search}(v \leftarrow \text{root}, d \leftarrow 0, x) \\
v: \text{node of trie}; d: \text{level of } v, x: \text{word} \\
1. \text{if } v \text{ is a leaf} \\
2. \quad \text{return } v \\
3. \quad \text{else} \\
4. \quad \text{let } c \text{ be child of } v \text{ labelled with } x[d] \\
5. \quad \text{if } \text{there is no such child} \\
6. \quad \quad \text{return } \text{“not found”} \\
7. \quad \quad \text{else } \text{Trie-search}(c, d + 1, x)
\]
Tries: Search Example

Example: Search(011$)
Tries: Search Example

Example: Search(011$)
Tries: Search Example

Example: Search(011$)
Tries: Search Example

Example: Search(011$)
Tries: Search Example

Example: Search(011$) successful
Tries: Search Example

Example: Search(0111$)

Diagram of a trie with labels and edges to illustrate the search process for the example string 0111$.

- The root node is labeled with an empty string $.
- The path 00$ follows the left branch to the leftmost leaf.
- The path 011$ follows the right branch to another node labeled 011$.
- The path 110$ follows the right branch again to the rightmost leaf.

Each node represents a partial prefix of the search string, and the edges with labels indicate the characters in the string.

The diagram illustrates how to navigate the trie to find the longest prefix match for the given search string.
Example: Search(0111$) unsuccessful

Tries: Search Example
Tries: Insert & Delete

- **Insert(\(x\))**
  - Search for \(x\), this should be unsuccessful
  - Suppose we finish at a node \(v\) that is missing a suitable child. Note: \(x\) has extra bits left.
  - Expand the trie from the node \(v\) by adding necessary nodes that correspond to extra bits of \(x\).

- **Delete(\(x\))**
  - Search for \(x\)
  - let \(v\) be the leaf where \(x\) is found
  - delete \(v\) and all ancestors of \(v\) until we reach an ancestor that has two children.

- **Time Complexity of all operations:** \(\Theta(|x|)\)
  - \(|x|\): length of binary string \(x\), i.e., the number of bits in \(x\)
Tries: Insert Example

Example: Insert(0111$)

```
Tries: Insert Example

Example: Insert(0111$)

```

```
  root
     /\    /
    /  \  /  \
   0    1 0    1
  / \  / \ / \ / \ 
/  \ 0 1 0 1 1 0 0
/     \ /     \ /     \ 
0  $  1 0 0 1 $ 1 0 1 1 0
  $  $  $  $  $  $  $  $  $  $  $
  $  $  $  $  $  $  $  $  $  $  $
 00$ 01001$ 01101$ 1101$ 111$
```

```
Biedl, Petrick, Veksler (SCS, UW)  
CS240 – Module 6  
Winter 2019  
```
Tries: Insert Example

Example: Insert(0111$)
Tries: Delete Example

Example: Delete(01001$)
Tries: Delete Example

Example: Delete(01001$)
Tries: Delete Example

Example: Delete(01001$)

```
root

0
0

1
1

0
1

00$
0001$
011$
01101$

1
1
1

1
1

1
1

1
1

110$
1101$
111$
111$
```
Outline

1. Lower bound
2. Interpolation Search
3. Tries
   - Standard Tries
   - Variations of Tries
   - Compressed Tries

Biedl, Petrick, Veksler (SCS, UW)
CS240 – Module 6
Winter 2019
Variation 1 of Tries: No leaf labels

Do not store actual keys at the leaves.

- The key is stored implicitly through the characters along the path to the leaf. It therefore need not be stored again.
- This halves the amount of space needed.
Variations 2 of Tries: Remove Chains to Labels

Stop adding nodes to trie as soon as the key is unique.

- A node has a child only if it has at least two descendants.
- Saves space if there are only few bitstrings that are long.
- Note that this variation *cannot* be combined with the previous one (why not?)

This variation is the one presented in Sedgewick.
Variation 3 of Tries: Allow Proper Prefixes

Allow prefixes to be in dictionary.

- Internal nodes may now also represent keys. Use a *flag* to indicate such nodes.
- Can remove $-$children, replace by flags.
- Now trie is a binary tree. Can express 0-child and 1-child implicitly via left and right child.
- More space-efficient.
Outline

1. Lower bound

2. Interpolation Search

3. Tries
   - Standard Tries
   - Variations of Tries
   - Compressed Tries
Compressed Tries (Patricia Tries)

- Morrison (1968): Patricia-Tries: *Practical Algorithm to Retrieve Information Coded in Alphanumeric*

- **Idea:** compress paths of unflagged nodes with only one child

- Each node stores an *index*: next bit to be tested during a search (index = 0 for the first bit, index = 1 for the second bit, etc.)

- A compressed trie storing $n$ keys always has at most $n - 1$ internal (non-leaf) nodes
Compressed Tries: Search

- start from the root and the bit indicated at that node
- follow the link that corresponds to the current bit in \( x \);
  return failure if the link is missing
- if we reach a leaf, explicitly check whether word stored at leaf is \( x \)
- else recurse on the new node and the next bit of \( x \)

\[
\text{Patricia-Trie-search}(v \leftarrow \text{root}, x)
\]

\( v \): node of trie; \( x \): word

1. \( \text{if } v \text{ is a leaf} \)
2. \( \quad \text{return } \text{strcmp}(x, \text{key}(v)) \)
3. \( \quad \text{else} \)
4. \( \quad \quad d \leftarrow \text{index stored at } v \)
5. \( \quad \quad c \leftarrow \text{child of } v \text{ labelled with } x[d] \)
6. \( \quad \quad \text{if } \text{there is no such child} \)
7. \( \quad \quad \quad \text{return } \text{"not found"} \)
8. \( \quad \quad \text{else } \text{Patricia-Trie-search}(c, x) \)
Compressed Tries: Search Example

Example: Search(10$)
Example: Search(10$) unsuccessful
Compressed Tries: Search Example

Example: Search(101$)
Compressed Tries: Search Example

Example: Search(101$) unsuccessful
Compressed Tries: Insert & Delete

- **Delete(x):**
  1. Perform Search(x)
  2. Remove the node \( v \) that stored \( x \)
  3. Compress along path to \( v \) whenever possible.

- **Insert(x):**
  1. Perform Search(x)
  2. Let \( v \) be the node where the search ended.
  3. Conceptually simplest approach:
     - Uncompress path from root to \( v \).
     - Insert \( x \) as in an uncompressed trie.
     - Compress paths from root to \( v \) and from root to \( x \).

   But it can also be done by only adding those nodes that are needed, see the textbook for details.

- All operations take \( O(|x|) \) time.
Multiway Tries: Larger Alphabet

- To represent **Strings** over any fixed alphabet $\Sigma$
- Any node will have at most $|\Sigma| + 1$ children (one child for the end-of-word character $\$$)
- Example: A trie holding strings $\{\text{bear$\$, ben$\$, be$, soul$\$, soup$\}\}$
Compressed Multiway Tries

- **Compressed** multi-way tries
- **Example:** A compressed trie holding strings \{bear$, ben$, be$, soul$, soup$\}
Multiway Tries: Summary

- Operations Search($x$), Insert($x$) and Delete($x$) are exactly as for tries for bitstrings.
- Run-time $O(|x| \cdot \text{time to find the appropriate child})$
- Each node now has up to $|\Sigma| + 1$ references to children. How should they be stored?

**Solution 1:** Array of size $|\Sigma| + 1$ for each node.  
**Complexity:** $O(1)$ time to find child, $O(|\Sigma|n)$ space.

**Solution 2:** List of children for each node.  
**Complexity:** $O(|\Sigma|)$ time to find child, $O(\#\text{children})$ space.

**Solution 3:** Dictionary (AVL-tree?) of children for each node.  
**Complexity:** $O(\log(\#\text{children}))$ time, $O(\#\text{children})$ space.  
Best in theory, but not worth it in practice unless $|\Sigma|$ is huge.

In practice, use *hashing* (keys are in (typically small) range $\Sigma$).