Outline

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

The fastest realizations of *ADT Dictionary* require $\Theta(\log n)$ time to search among $n$ items. Is this the best possible?
Lower bound for search

The fastest realizations of *ADT Dictionary* require $\Theta(\log n)$ time to search among $n$ 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**: via decision tree

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

Recall the run-times in a *sorted array*:

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

\[
\text{binary-search}(A, n, k)
\]
\[
A: \text{Sorted array of size } n, \ k: \text{key}
\]
\[
1. \quad \ell \leftarrow 0, \ r \leftarrow n - 1
\]
\[
2. \quad \text{while } (\ell \leq r)
\]
\[
3. \quad m \leftarrow \left\lfloor \frac{\ell + r}{2} \right\rfloor
\]
\[
4. \quad \text{if } (A[m] < k) \ \text{then } \ell = m + 1
\]
\[
5. \quad \text{else if } (k < A[m]) \ \text{then } r = m - 1
\]
\[
6. \quad \text{else return "found at } A[m]"
\]
\[
7. \quad \text{return "not found, but would be between } A[\ell - 1] \ \text{and} \ A[\ell]"
\]
Interpolation Search: Motivation

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

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

| \ell | 40 | \downarrow | r | 120 |

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

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

<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>

\textbf{Question}: If keys are \textit{numbers}, where would you expect key \( k = 100 \)?

\textit{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

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
0 & 1 & 2 & 3 & 449 & 450 & 600 & 800 & 1000 & 1200 & 1500
\end{array}
\]

\textit{interpolation-search}(A[0..10],449):

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

Works well if keys are uniformly distributed:

Can show: 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

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
0 & 1 & 2 & 3 & 449 & 450 & 600 & 800 & 1000 & 1200 & 1500 \\
\end{array}
\]

\textit{interpolation-search}(A[0..10],449):

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

Works well if keys are uniformly distributed:

Can show: Recurrence relation is \( T(\text{avg})(n) = T(\text{avg})(\sqrt{n}) + \Theta(1) \).

This resolves to \( T(\text{avg})(n) \in \Theta(\log\log n) \).
Interpolation Search Example

interpolation-search(A[0..10],449):

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

\[\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 449 & 450 & 600 & 800 & 1000 & 1200 & 1500 \\
\end{array}\]

...\left\lceil \frac{449-0}{1500-0} (10 - 0) \right\rceil = \ell + 2 = 2

\ell = 3, r = 10, m = \ell + \left\lceil \frac{449-3}{1500-3} (10 - 3) \right\rceil = \ell + 2 = 5

\ell = 3, r = 4, m = \ell + \left\lceil \frac{449-3}{449-3} (4 - 3) \right\rceil = \ell + 1 = 4, \text{ found at } A[4]
Interpolation Search Example

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
0 & 1 & 2 & 3 & 449 & 450 & 600 & 800 & 1000 & 1200 & 1500 \\
\end{array}
\]

**interpolation-search**(A[0..10],449):

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

Works well if keys are *uniformly* distributed:

- Can show: 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

- Code very similar to binary search, but compare at interpolated index
- Need a few extra tests to avoid crash during computation of \( m \).

\[
\text{interpolation-search}(A, n, k)
\]

\( A \): Sorted array of size \( n \), \( k \): key

1. \( \ell \leftarrow 0, \ r \leftarrow n - 1 \)
2. \textbf{while} \( (\ell \leq r) \)
3. \quad \textbf{if} \ (k < A[\ell] \text{ or } k > A[r]) \ \textbf{return} \text{ “not found”}
4. \quad \textbf{if} \ (A[\ell] = A[r]) \ \textbf{then return} \text{ “found at } A[\ell]”
5. \quad m \leftarrow \ell + \lfloor \frac{k - A[\ell]}{A[r] - A[\ell]} \cdot (r - \ell) \rfloor
6. \quad \textbf{if} \ (A[m] < k) \ \textbf{then} \ \ell = m + 1
7. \quad \textbf{else if} \ (k < A[m]) \ \textbf{then} \ r = m - 1
8. \quad \textbf{else} \ \textbf{return} \text{ “found at } A[m]”
9. \quad \textbf{// We always return from somewhere within while-loop}
Outline

1. Lower bound

2. Interpolation Search

3. Tries
   - Standard Tries
   - Variations of Tries
   - Compressed Tries
Trie (also known as **radix tree**): A dictionary for bitstrings.

(Should know: string, word, $|w|$, alphabet, prefix, suffix, comparing words, ....)

- Comes from retrieval, but pronounced “try”
- A tree based on **bitwise comparisons**: Edge labelled with corresponding bit
- Similar to **radix sort**: use individual bits, not the whole key
More on tries

**Assumption:** Dictionary is **prefix-free:** no string is a prefix of another
- Assumption satisfied if all strings have the same length.
- Assumption satisfied if all strings end with ‘end-of-word’ character $\$.

**Example:** A trie for \{00$, 0001$, 0100$, 011$, 0110$, 110$, 1101$, 111$\}

```
           0
          / \  
         0    1
        / \  / \  
       0 1 0 1 $ 0 1 $ 1 $ 1 1 $ 110 $ 1 1 1 $ 1101 $ 0100 $ 001 $ 00 $ 011 $
```

Then items (keys) are stored only in the leaf nodes.
More on tries

**Assumption:** Dictionary is *prefix-free*: no string is a prefix of another

- Assumption satisfied if all strings have the same length.
- Assumption satisfied if all strings end with ‘end-of-word’ character $.

**Example:** A trie for \{00$, 0001$, 0100$, 011$, 0110$, 110$, 1101$, 111$\}

![Trie Diagram]

Then items (keys) are stored *only* in the leaf nodes.
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 $x$

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

Example: Trie::search(011$)
Tries: Search Example

Example: Trie::search(011$)
Example: Trie::search(011$)
Tries: Search Example

Example: Trie::search(011$)
Example: Trie::search(011$) successful
Example: Trie::search(0111$)

```
Example: Trie::search(0111$)
```

![Trie diagram]

- **Tries: Search Example**
- **Example:** Trie::search(0111$)

```
Biedl, Schost, Veksler (SCS, UW)  CS240 – Module 6  Winter 2021  10 / 23
```
Example: Trie::search(0111$) unsuccessful
Tries: Insert & Delete

- **Trie::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 $x$.

- **Trie::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 $x$
Example: `Trie::insert(0111$)`

```
  0  1
  0  1
     no 1-child
```

```
  00$  011$  110$  111$
```

```
  0001$
```

```
  01001$  01101$
```

```
Example: \textit{Trie::insert}(0111\$)
Example: \textit{Trie::delete}(01001$)$
Example: \texttt{Trie::delete}(01001$)$
Outline

1. Lower bound

2. Interpolation Search

3. Tries
   - Standard Tries
   - Variations of Tries
   - Compressed Tries
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.
Variation 2 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.
- No need for end-of-word character $\$$
- Now a trie of bitstrings is a binary tree. Can express 0-child and 1-child implicitly via left and right child.
- More space-efficient.
Variations 3 of Tries

Pruned Trie: 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.
- Note that now we must store the full keys (why?)
- Saves space if there are only few bitstrings that are long.
- Could even store infinite bitstrings (e.g. real numbers)

This is in practice the most efficient version of tries, but the operations get a bit more complicated.
Outline

1. Lower bound

2. Interpolation Search

3. Tries
   - Standard Tries
   - Variations of Tries
   - Compressed Tries
Variation 4 of Tries

**Compressed Trie:** compress paths of nodes with only one child
- Each node stores an *index*, corresponding to the depth in the uncompressed trie.
  - This gives the next bit to be tested during a search
- A compressed trie with $n$ keys has at most $n - 1$ internal nodes

Also known as **Patricia-Tries**:
*Practical Algorithm to Retrieve Information Coded in Alphanumeric*
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 \( x \)

```cpp
CompressedTrie::search(v ← root, x)
```

\( v \): node of trie; \( x \): word
1. \( \text{if } v \text{ is a leaf} \)
2. \( \text{return } \text{strcmp}(x, v.key) \)
3. \( d ← \text{index stored at } v \)
4. \( \text{if } x \text{ has at most } d \text{ bits} \)
5. \( \text{return } \text{“not found”} \)
6. \( v' ← \text{child of } v \text{ labelled with } x[d] \)
7. \( \text{if there is no such child} \)
8. \( \text{return } \text{“not found”} \)
9. \( \text{CompressedTrie::search}(v', x) \)
Example: CompressedTrie::search(10$)
Example: `CompressedTrie::search(10\$)` unsuccessful
Compressed Tries: Search Example

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

Example: `CompressedTrie::search(101$)` unsuccessful
Compressed Tries: Search Example

Example: CompressedTrie::search(1$)
Compressed Tries: Search Example

Example: `CompressedTrie::search(1$)` unsuccessful

```
0
1
2
00$
$
0001$
01001$
0
1
3
011$
$
110$
1101$
1
111$

011$
$
01101$
```

"x too short"
Compressed Tries: Insert & Delete

- **CompressedTrie::delete(x):**
  - Perform `search(x)`
  - Remove the node $v$ that stored $x$
  - Compress along path to $v$ whenever possible.

- **CompressedTrie::insert(x):**
  - Perform `search(x)`
  - Let $v$ be the node where the search ended.
  - 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 \{bear$, ben$, be$, soul$, soup$\}
Compressed Multiway Tries

**Variation**: Compressed multi-way tries: compress paths as before

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

- Operations $\text{search}(x)$, $\text{insert}(x)$ and $\text{delete}(x)$ 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 $|\Sigma| + 1$ children. How should they be stored?

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

- **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 $\Sigma$).
Multiway Tries: Summary

- Operations \textit{search}(x), \textit{insert}(x) and \textit{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 \(|\Sigma| + 1\) children. How should they be stored?
Multiway Tries: Summary

- Operations $\text{search}(x)$, $\text{insert}(x)$ and $\text{delete}(x)$ 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 $|\Sigma| + 1$ children. How should they be stored?

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

**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 $\Sigma$).