COMP2521 24T1
Tries

Kevin Luxa
cs2521@cse.unsw.edu.au
Many applications require searching through a set of strings with a *pattern*

**Examples:**
- Autocomplete
- Predictive text
- Approximate string matching
- Spell checking
Autocomplete
For example, pressing “4663” can be interpreted as the word good, home, hood or hoof.
How can we implement a set of strings using data structures covered so far?

**AVL tree**

Performance: $O(\log n)$ worst case

**Hash table**

Performance: $O(1)$ average case
AVL trees and hash tables are efficient, but...

...they are not efficient when searching for a *pattern*

Possible solution: *tries*
A trie...

- is a tree data structure
- used to represent a set of strings
  - e.g., all the distinct words in a document, a dictionary, etc.
  - we will call these strings *keys* or *words*
- supports string matching queries in $O(m)$ time
  - where $m$ is the length of the string being searched for

Note: the word *trie* comes from *retrieval*, but pronounced as “try” not “tree”
Example:

Keys in the trie:
- ace
- aces
- ape
- apes
- app
- apply
- early
- earth
- east
Important features of tries:

- Each link represents an individual character
- A key is represented by a path in the trie
- Each node can be tagged as a “finishing” node
  - A “finishing” node marks the end of a key
- Each node may contain data associated with key
- Unlike a search tree, the nodes in a trie do not store their associated key
  - Instead, keys are implicitly defined by their position in the trie
Assuming alphabetic strings:

```c
#define ALPHABET_SIZE 26

struct node {
  struct node *children[ALPHABET_SIZE];
  bool finish; // marks the end of a key
  Data data;   // data associated with key
};
```
Consider this trie:

```
s
  e
  |
  l
  |
  a
  |
  s
  h
  |
  e
  l
  |
  e
```
Concrete representation:
(f = finishing node)
Trie Insertion

Process for insertion:

- Start at the root
- For each character $c$ in the key (from left to right):
  - If there is no child node corresponding to $c$, create one
  - Descend into the child node corresponding to $c$
- Mark the resulting node as a finishing node and insert data (if any)
Insert the following words into an initially empty trie:

sea  shell  sell  shore  she
Insert the following words into an initially empty trie:

sea  shell  sell  shore  she
Recursive method:

trieInsert(t, key, data):

**Input:** trie \( t \) 
- key of length \( m \) and associated data

**Output:** \( t \) with key and data inserted

**Algorithm:**

if \( t \) is empty:
   \( t = \) new node

if \( m = 0 \):
   \( t\rightarrow finish = \) true
   \( t\rightarrow data = \) data

else:
   first = key[0]
   rest = key[1..m-1] // i.e., slice off first character from key
   \( t\rightarrow children[first] = \) trieInsert(\( t\rightarrow children[first] \), rest, data)

return \( t \)

**EXERCISE** Try writing an iterative version.
Trie Search

Search is similar to insertion:

- Start at the root
- For each character $c$ in the key (from left to right):
  - If there is no child node corresponding to $c$, return false
  - Descend into the child node corresponding to $c$
- If the resulting node is a finishing node, then return true, otherwise return false
Search for “early”

- Trie Search
- Example
- Search for “early”

```
Found!
```
Trie Search

Example

Search for “early”

```
  a
 / \                     /  \
 c   p                   1   e
 / \    \               /   /\
 e   e   p             r   a
 /  \    /           \  /  \
 s   s   l            l  t
    \  /       \  /    \  /  \
     y   y       l   t   h
```

Found!
Trie Search Example

Search for “apple”

Not found - node for “appl” has no child node for ‘e’
Search for “apple”

Not found - node for “appl” has no child node for ‘e’
Search for “ear”

Not found - node for “ear” is not a finishing node
Search for “ear”

Not found - node for “ear” is not a finishing node
Recursive method:

```python
trieSearch(t, key):
    Input: trie t
           key of length m
    Output: true if key is in t
            false otherwise

    if t is empty:
        return false
    else if m = 0:
        return t->finish = true
    else:
        first = key[0]
        rest = key[1..m - 1]
        return trieSearch(t->children[first], rest)
```

**EXERCISE** Try writing an iterative version.
Deletion is trickier...

- Can simply find node corresponding to given key and mark it as a non-finishing node
- ...but this can leave behind dead branches
  - i.e., branches that don’t contain any finishing nodes
  - dead branches waste memory
Example of dead branch:

Delete "shore"
Process for deletion:

- Find node corresponding to given key
  - If node doesn’t exist, do nothing
- Mark the node as a non-finishing node
- While current node is not a finishing node and has no child nodes:
  - Delete current node and move up to parent
    - Handled recursively
Delete “ace”

Deleted - node for “ace” is no longer marked as a finishing node.
Delete “ace”

Deleted - node for “ace” is no longer marked as a finishing node
Delete "apply"
Delete “apply”

Deleted - deleted nodes corresponding to “apply” and “appl”
Delete “earth”

Deleted - deleted nodes corresponding to “earth” and “eart”
Delete “earth”

Deleted - deleted nodes corresponding to “earth” and “eart”
Trie Deletion

Recursive method:

trieDelete(t, key):
    Input:    trie t
              key of length m
    Output:   t with key deleted

    if t is empty:
        return t
    else if m = 0:
        t->finish = false
    else:
        first = key[0]
        rest = key[1..m - 1]
        t->children[first] = trieDelete(t->children[first], rest)

    if t->finish = false and t has no child nodes:
        return NULL
    else:
        return t
Analysis of standard trie:
- $O(m)$ insertion, search and deletion
  - where $m$ is the length of the given key
  - each of these needs to examine at most $m$ nodes
- $O(nR)$ space
  - where $n$ is the total number of characters in all keys
  - where $R$ is the size of the underlying alphabet (e.g., 26)
Simple trie representation consumes an enormous amount of memory

- Each node contains ALPHABET_SIZE pointers
  - If keys are alphabetic, then this is 26 pointers...
    - ...which is $8 \times 26 = 208$ bytes on a 64-bit machine!
  - If keys can contain any ASCII character, then this is 128 pointers!
- Even if trie contains many keys, most child pointers will be unused
Different representations exist to reduce memory usage at the cost of increased running time:

- Use a singly linked list to store child nodes
- Alphabet reduction - break each character into smaller chunks, and treat these chunks as the characters
One technique to reduce memory usage:

Have each node store a linked list of its children instead of an array of ALPHABET_SIZE pointers.
Instead of:

We have:

```c
struct node {
    struct child *children;
    bool finish;
    Data data;
};

struct child {
    char c;
    struct node *node;
    struct child *next;
};
```
Consider the following trie:
Its concrete representation:
We can simplify this representation by merging each linked list node with its corresponding trie node. This produces the left-child right-sibling **binary tree** representation:

```c
struct node {
    char c;
    struct node *children;
    struct node *sibling;
    bool finish;
    Data data;
};
```
Concrete representation of above trie:

A trie is a binary tree where each node represents a single character and the tree is structured such that a path from the root to a leaf node spells out a word. In this case, the trie represents the word "texted". The left-child right-sibling binary tree (LCRSB) is a variant where children are linked linearly, allowing for efficient traversal and search operations.

The concrete representation shows:
- The root node is 'a', which has a child 'n' and another child 't'.
- The child 'n' has a child 'e'.
- The child 't' has another child 'e'.
- The root node 'a' also has a child 'b', with children 'i' and 'o'.
- The child 'i' has a child 'o', and its sibling 'd' has children 'd' and 'w'.
- The child 'b' is connected to the child 'd'.
Analysis:

- This representation uses much less space
  - Each node just stores one extra pointer to its sibling instead of ALPHABET_SIZE pointers
- But this is at the expense of running time
  - Need to traverse up to ALPHABET_SIZE nodes before reaching desired child
Another technique to reduce memory usage: alphabet reduction

Break each 8-bit character into two 4-bit nybbles

This reduces the branching factor, i.e., the number of pointers in each node
For example, the word “sea” consists of the following bytes:

\[
\begin{array}{ccc}
 s & e & a \\
01110011 & 01100101 & 01100001 \\
\end{array}
\]

We break it into 4-bit nybbles like so:

\[
\begin{array}{ccc}
 s & e & a \\
01110011 & 01100101 & 01100001 \\
0110 & 0011 & 0110 & 0101 & 0110 & 0001 \\
\end{array}
\]

Instead of storing the word “sea”, we now insert the following word:

\[
0111 \ 0011 \ 0110 \ 0101 \ 0110 \ 0001
\]
Analysis:

- This representation uses much less space
  - Much fewer pointers per node
- But this is at the expense of running time
  - Path to each key is twice as long - lookups need to visit twice as many nodes
Another technique to reduce memory usage: use a compressed trie

In a compressed trie, each node contains $\geq 1$ character

Obtained by merging non-branching chains of nodes
Specifically, non-finishing nodes with only one child are merged with their child
Variants

Compressed tries

Motivation

Tries

Insertion

Search

Deletion

Analysis

Variants
Linked list of children
Binary tree
Alphabet reduction

Compressed tries

Applications

Appendix
Idea:

Given a document, preprocess it by storing all words in a trie, and for each word, store the location of all its occurrences.

When user searches for a word, can query the trie instead of scanning entire document.
Applications
Word finding

Motivation
Tries
Insertion
Search
Deletion
Analysis
Variants
Applications
Word finding
Autocomplete
Predictive text
Appendix
Autocomplete
Given a series of letters, find all words that start with it
**Predictive text**

Given a series of button presses (e.g., on a keypad), where each button can represent multiple letters, find all possible matching words.
Appendix
Trie Insertion Example

Insert the following words into an initially empty trie:

sea  shell  sell  shore  she
Insert the following words into an initially empty trie:

sea    shell    sell    shore    she
Trie Insertion Example

Insert the following words into an initially empty trie:

sea  shell  sell  shore  she
Insert the following words into an initially empty trie:

sea    shell    sell    shore    she

```plaintext
  s
  e
  a
```

Trie Insertion Example
Insert the following words into an initially empty trie:

sea shell sell shore she
Insert the following words into an initially empty trie:

sea  shell  sell  shore  she
Insert the following words into an initially empty trie:

sea  shell  sell  shore  she
Insert the following words into an initially empty trie:

sea shell sell shore she
Insert the following words into an initially empty trie:

sea shell sell shore she
Insert the following words into an initially empty trie:

sea shell sell shore she

```
s
  e
  |
  s

  l
  |
  a

  h
  |
  e

  o
  |
  r

  l
  |
  e
```
Insert the following words into an initially empty trie:

sea  shell  sell  shore  she
Insert the following words into an initially empty trie:

sea shell sell shore she