# Strings: Tries

Neerja Mhaskar

Dept. of Computing and Software, McMaster University, Canada

**Acknowledgments:** Material mainly based on the textbook Algorithms by Robert Sedgewick and Kevin Wayne (Chapters 5.2)

# Summary of the performance of symbol-table implementations

Order of growth of the frequency of operations.

| implementation | typical case | | | ordered operations | operations on keys |
|---|---|---|---|---|---|
| | search | insert | delete | | |
| **red-black BST** | $\log N$ | $\log N$ | $\log N$ | ✔ | `compareTo()` |
| **hash table** | $1$ † | $1$ † | $1$ † | | `equals()` `hashCode()` |

† under uniform hashing assumption

Q. Can we do better?

A. Yes, if we can avoid examining the entire key, as with string sorting.

# String symbol table basic API

String symbol table. Symbol table specialized to string keys.

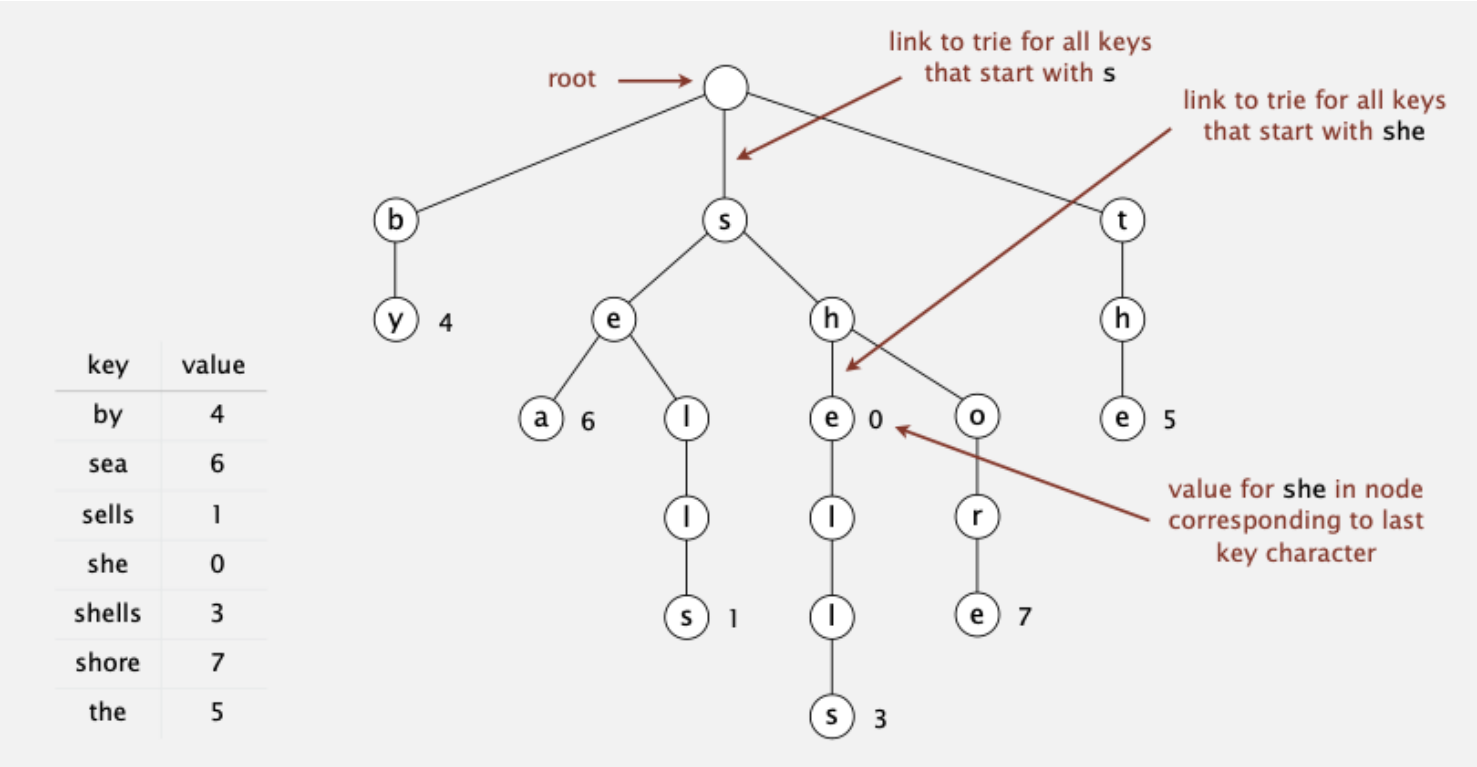
| public class StringST<Value> | |
|---|---|
| StringST() | *create an empty symbol table* |
| void put(String key, Value val) | *put key-value pair into the symbol table* |
| Value get(String key) | *return value paired with given key* |
| void delete(String key) | *delete key and corresponding value* |
| ⋮ | |

Goal. Faster than hashing, more flexible than BSTs.

# Trie

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

- Store characters in nodes (not keys).
- Each node has R children, one for each possible character. (for now, we do not draw null links)

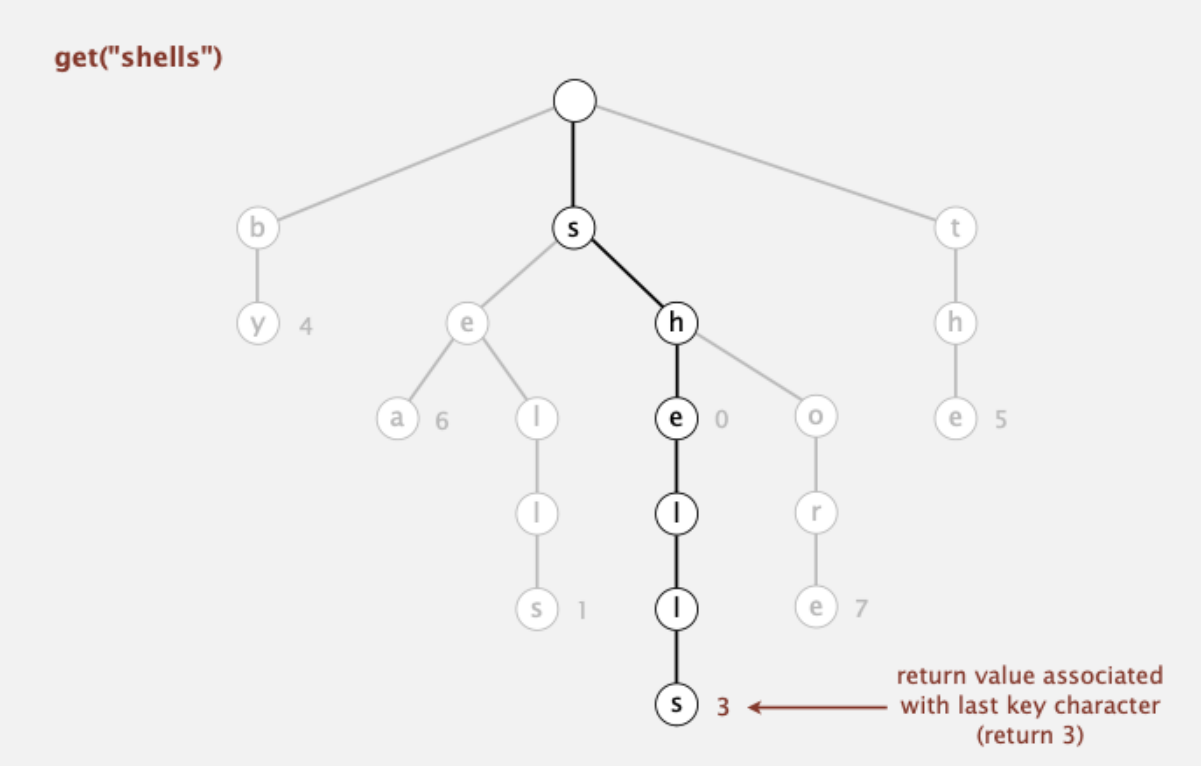



| key | value |
|---|---|
| by | 4 |
| sea | 6 |
| sells | 1 |
| she | 0 |
| shells | 3 |
| shore | 7 |
| the | 5 |

# Search in a Trie I

Follow links corresponding to each character in the key.

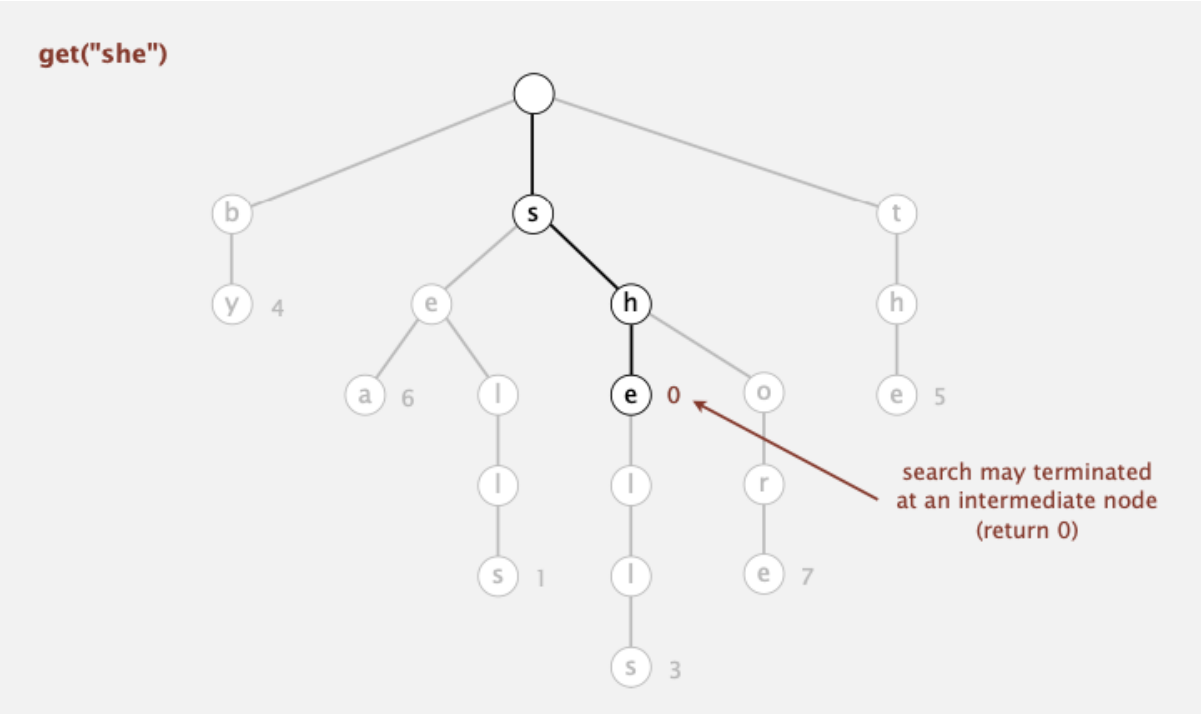
- Search hit: node where search ends has a non-null value.
- Search miss: reach null link or node where search ends has null value.



# Search in a Trie II

Follow links corresponding to each character in the key.

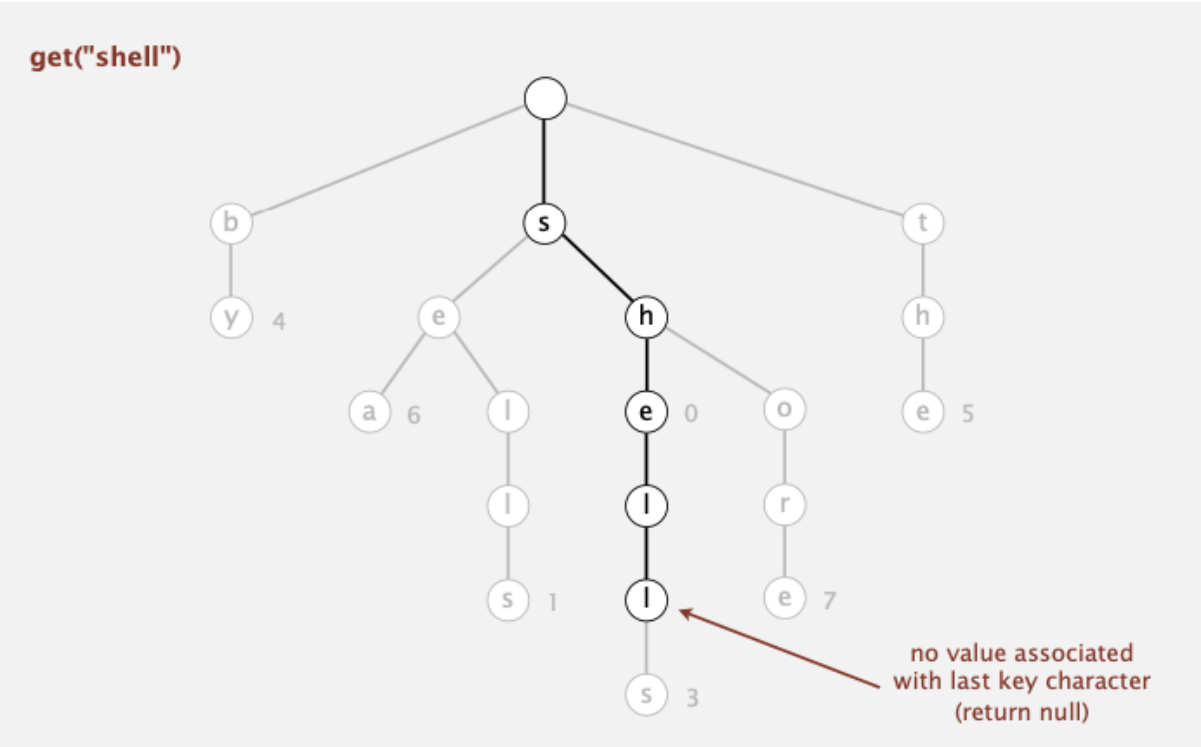
- Search hit: node where search ends has a non-null value.
- Search miss: reach null link or node where search ends has null value.



# Search in a Trie III

Follow links corresponding to each character in the key.

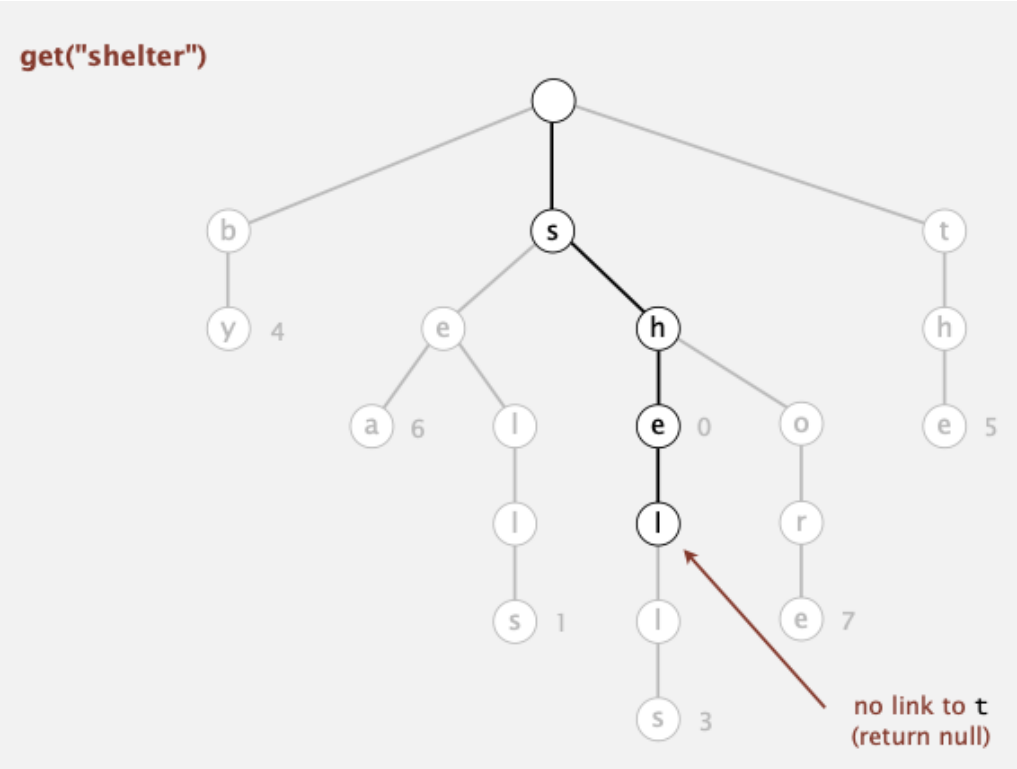
- Search hit: node where search ends has a non-null value.
- Search miss: reach null link or node where search ends has null value.



# Search in a Trie IV

Follow links corresponding to each character in the key.

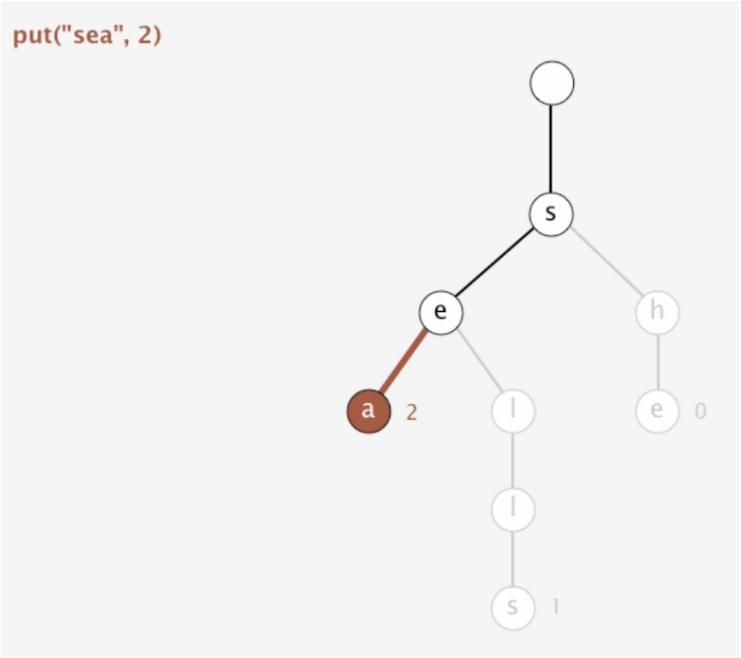
- Search hit: node where search ends has a non-null value.
- Search miss: reach null link or node where search ends has null value.



# Insertion into a trie - I

Follow links corresponding to each character in the key.

- Encounter a null link: create new node.
- Encounter the last character of the key: reset value in that node with new value.

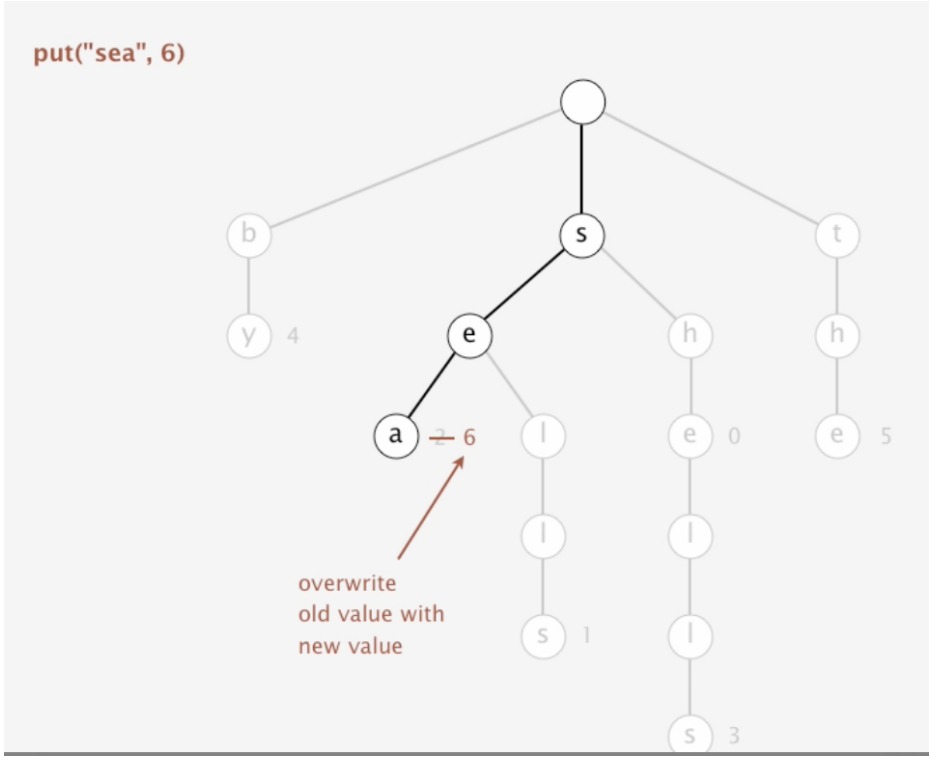



See Demo: `https://algs4.cs.princeton.edu/lectures/`

# Insertion into a trie - II

Follow links corresponding to each character in the key.

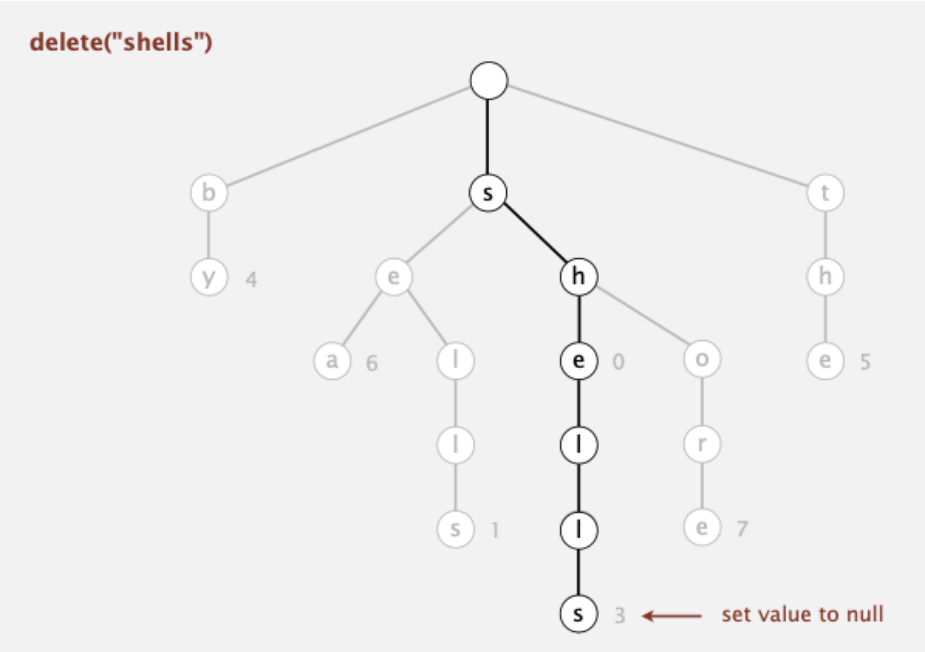
- Encounter a null link: create new node.
- Encounter the last character of the key: reset value in that node with new value.



# Deletion in an R-way trie I

To delete a key-value pair:

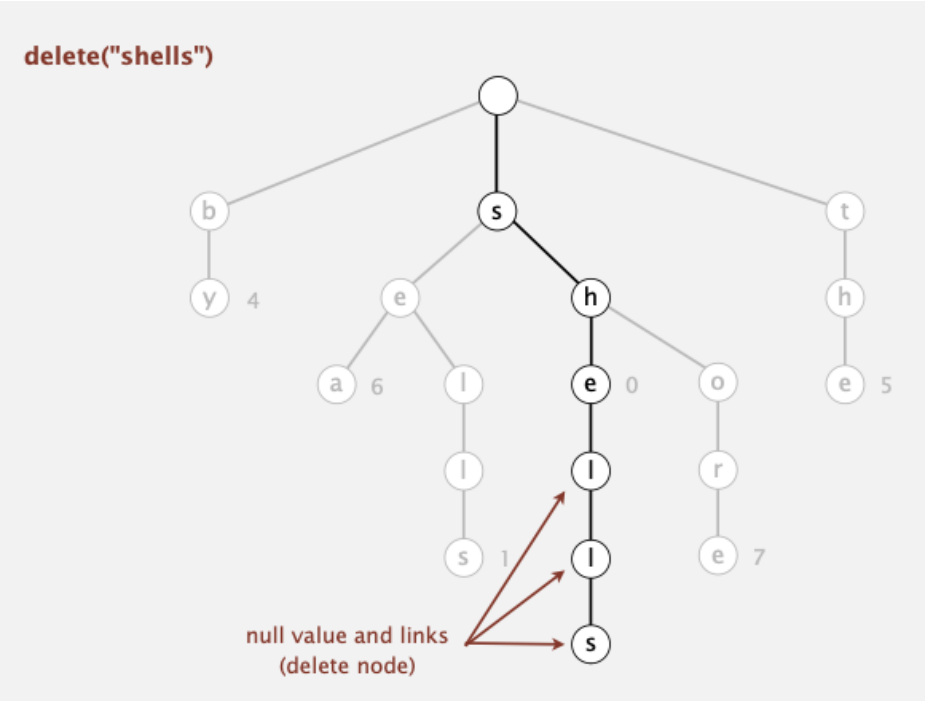
- Find the node corresponding to the last character in the key and set value to null.
- If node has null value and all null links, remove that node (and recur).



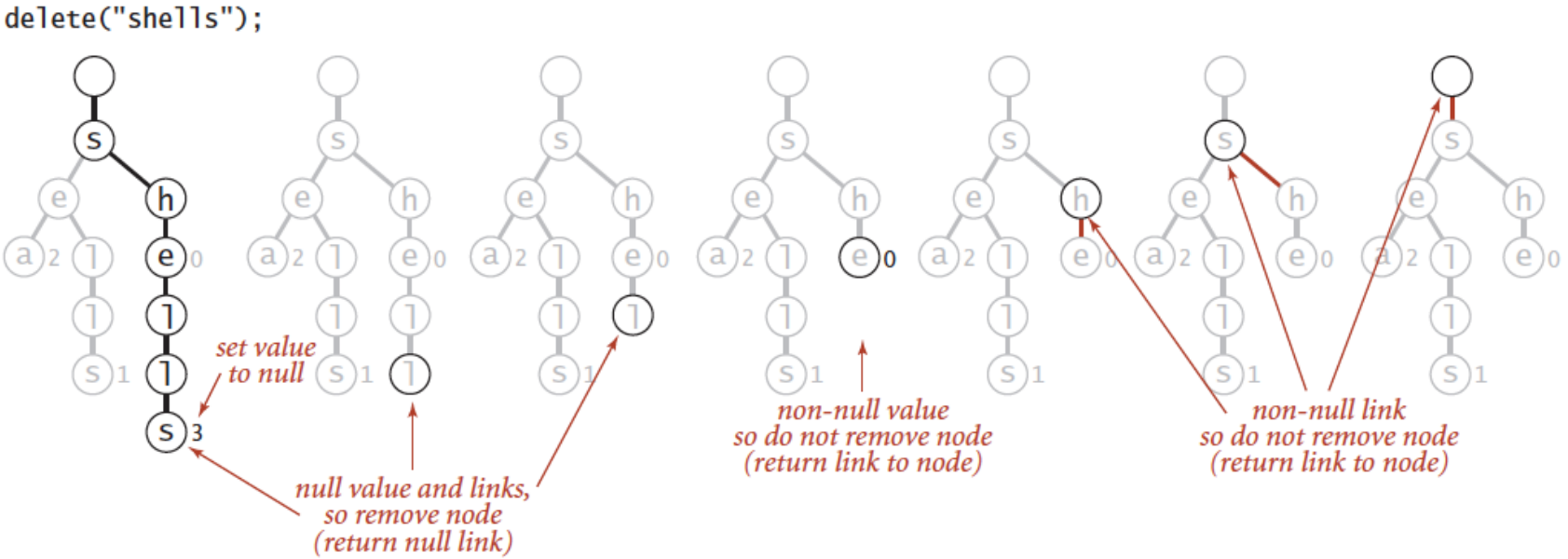
# Deletion in an R-way trie II

To delete a key-value pair:

- Find the node corresponding to the last character in the key and set value to null.
- If node has null value and all null links, remove that node (and recur).



# Deletion in an R-way trie complete example III




Deleting a key (and its associated value) from a trie

# Trie performance I

Search hit. Need to examine all characters in the key to test for equality.

Search miss.

- Could have mismatch on first character.
- Typical case: examine only a few characters (sublinear).

If we assume that the keys are drawn from the random string model (each character is equally likely to have any one of the R different character values) we can prove this fact:

Preposition. The average number of nodes examined for search miss in a trie built from N random keys over an alphabet of size R is $\sim \log_R N$.

From a practical standpoint, the most important implication of this proposition is that search miss does not depend on the key length. For example, it says that unsuccessful search in a trie built with 1 million random keys will require examining only three or four nodes, whether the keys are 7-digit license plates or 20-digit account numbers.

# Trie performance II

The R-way trie contains R links at each node.

Space. The number of links in a trie is between RN and RNw, where w is the average key length. Therefore the space requirement is ranges from O(RN) to O(RNw).

Bottom line. Fast search hit and even faster search miss, but wastes space.

# R-way trie summary

- If space is available, R-way tries provide the fastest search, essentially completing the job with a constant number of character compares.
- For large alphabets, where space may not be available for R-way tries (due to excessive memory usage), Ternary symbol table (not covered in the course) are preferable, since they use a logarithmic number of character compares, while BSTs use a logarithmic number of key compares.
- Hashing can be competitive, but, as usual, cannot support ordered symbol-table operations or extended character-based API operations such as prefix or wildcard match.