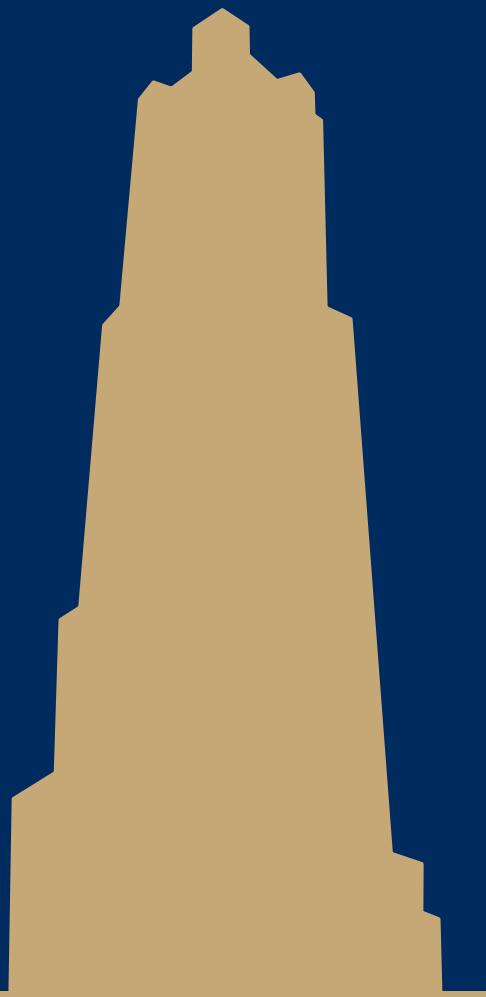
CS 1501

Tries



The Searching Problem

(yes, still)

Given a collection of keys *C*, determine whether or not *C* contains a specific key *k*

A closer look

- BSTs and Red/Black trees gave us solutions to the searching problem with O(lg n) runtimes (average/worst case, resp.)
- Can we do better than these?
- Both methods depend on comparisons against other keys
  - I.e., *k* is compared against other keys in the data structure
- 4 options at each node in a BST, searching for a key *k*:
  - Node ref is null, *k* not found
  - *k* is equal to the current node's key, *k* is found
  - *k* is less than current key, continue to left child
  - *k* is greater than the current key, continue to right child

Digital Search Trees (DSTs)

- Instead of looking at less than/greater than, lets go left right based on the bits of the key, so we again have 4 options:
  - Node ref is null, *k* not found
  - *k* is equal to the current node's key, *k* is found
  - current bit of *k* is 0, continue to left child
  - current bit of *k* is 1, continue to right child
- Is this going to asymptotically improve our runtime?

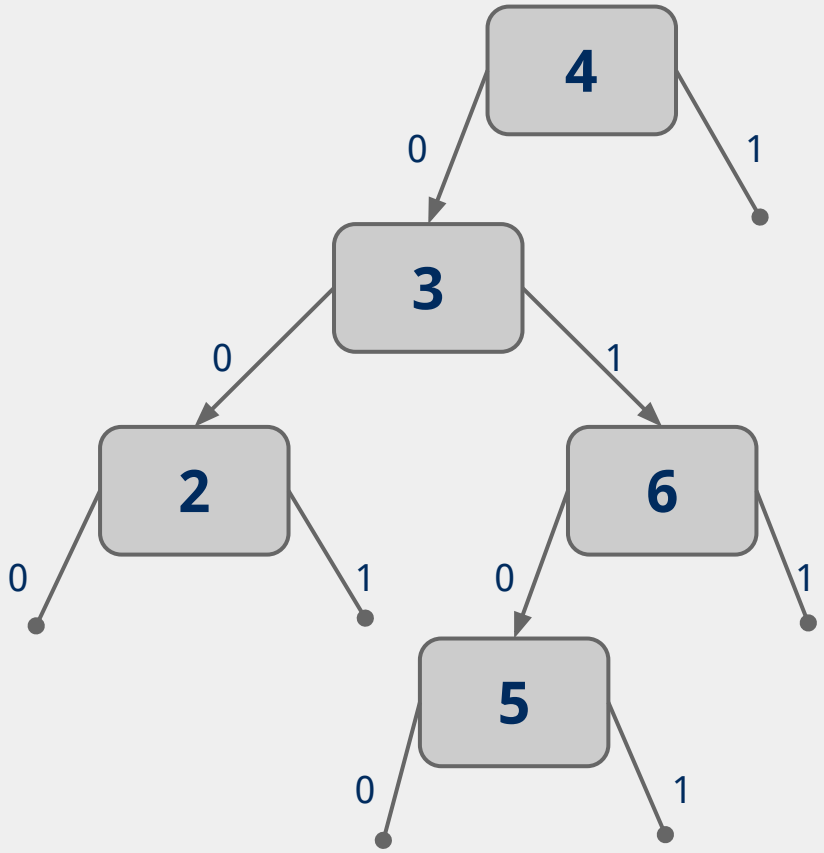
DST example

Insert:

4	0100
3	0011
2	0010
6	0110
5	0101

Search:

3	0011
7	0111



Analysis of digital search trees

- Runtime?

- We end up doing many comparisons against the full key, can we improve on this?

Radix search tries (RSTs)

- Trie as in retrieve, pronounced the same as "try"
- Instead of storing keys as nodes in the tree, we store them implicitly as paths down the tree
  - Interior nodes of the tree only serve to direct us according to the bitstring of the key
  - Values can then be stored at the end of key's bit string path

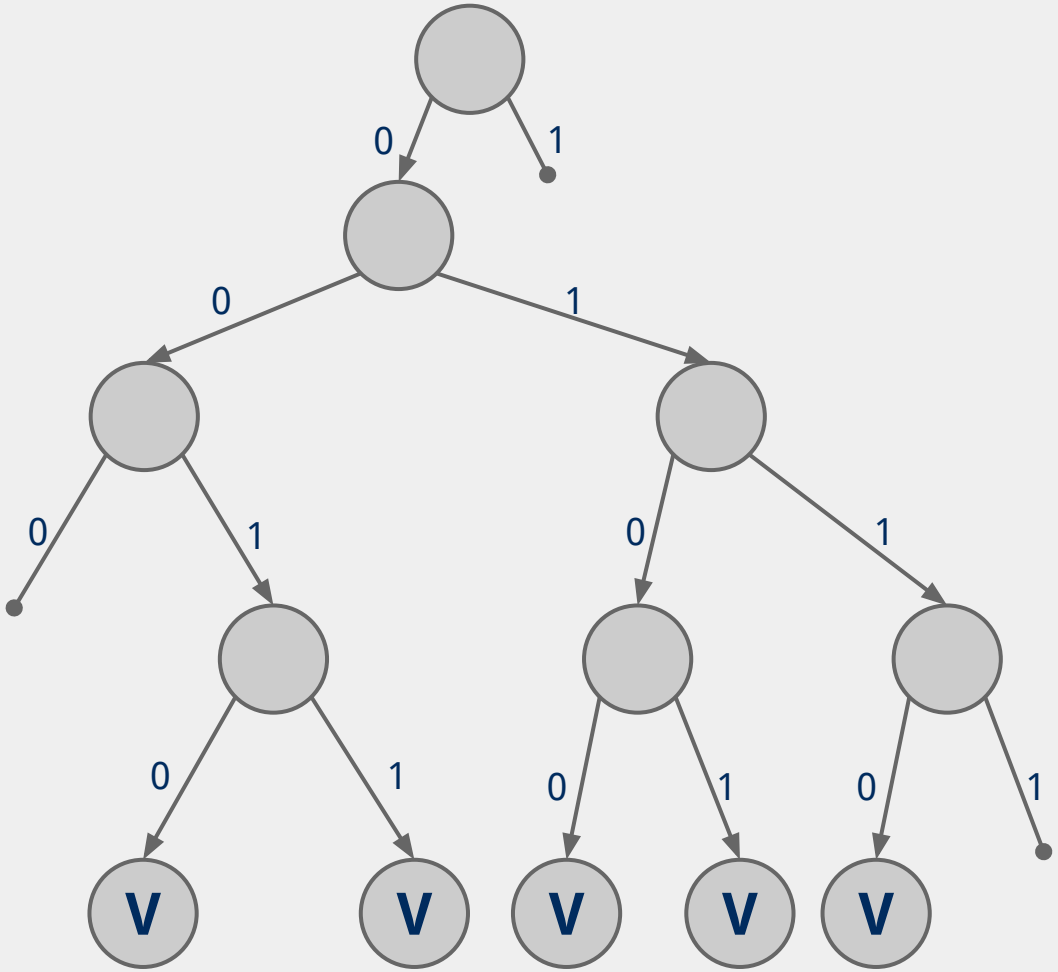
RST example

Insert:

4	0100
3	0011
2	0010
6	0110
5	0101

Search:

3	0011
7	0111



RST analysis

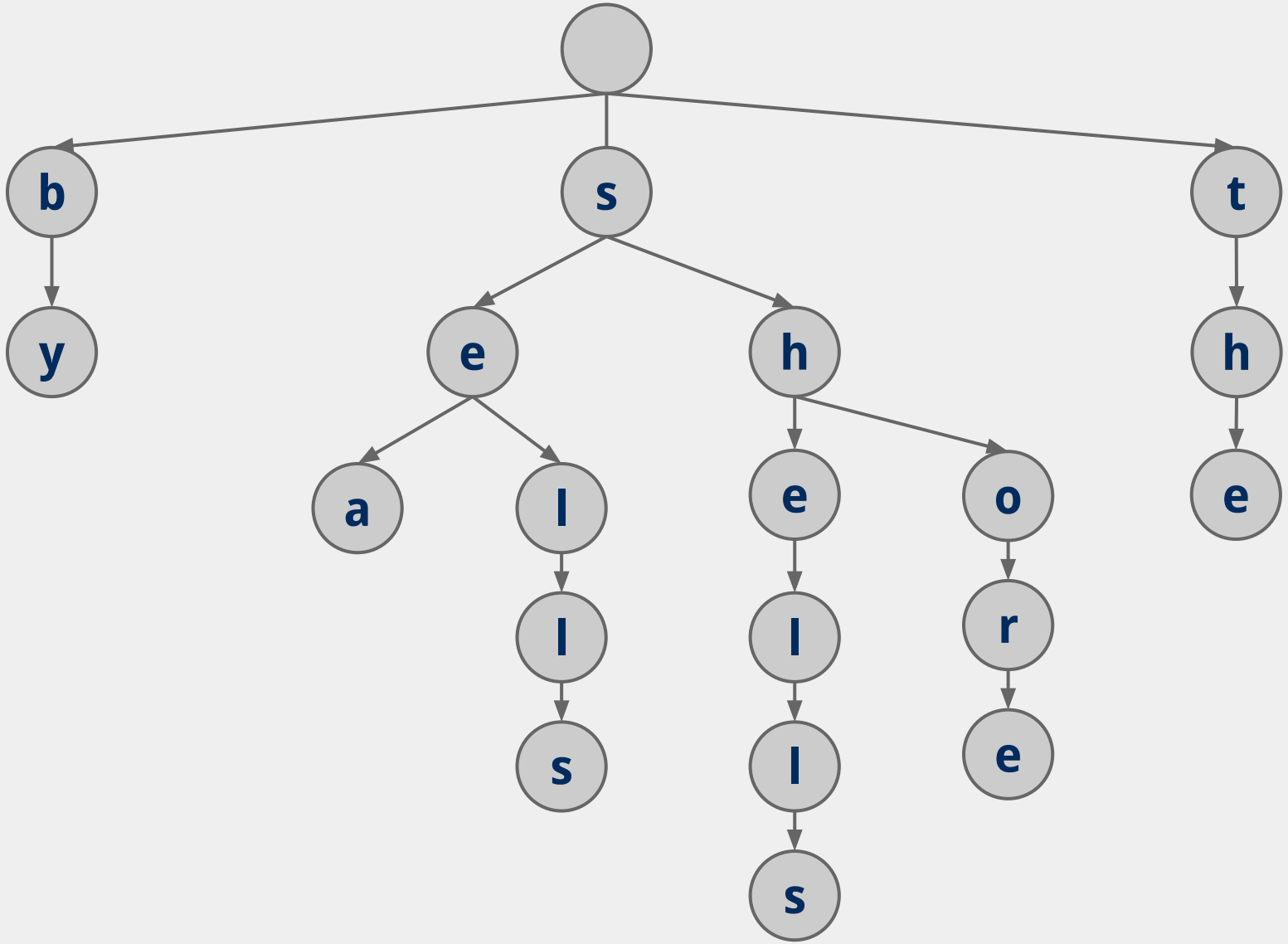
- Runtime?

- Would this structure work as well for other key data types?
  - Characters?
  - Strings?

Larger branching factor tries

- In our binary-based Radix search trie, we considered one bit at a time
- What if we applied the same method to characters in a string?
  - What would like this new structure look like?
- Let’s try inserting the following strings into an trie:
  - she, sells, sea, shells, by, the, sea, shore

Another trie example



Implementation Concerns

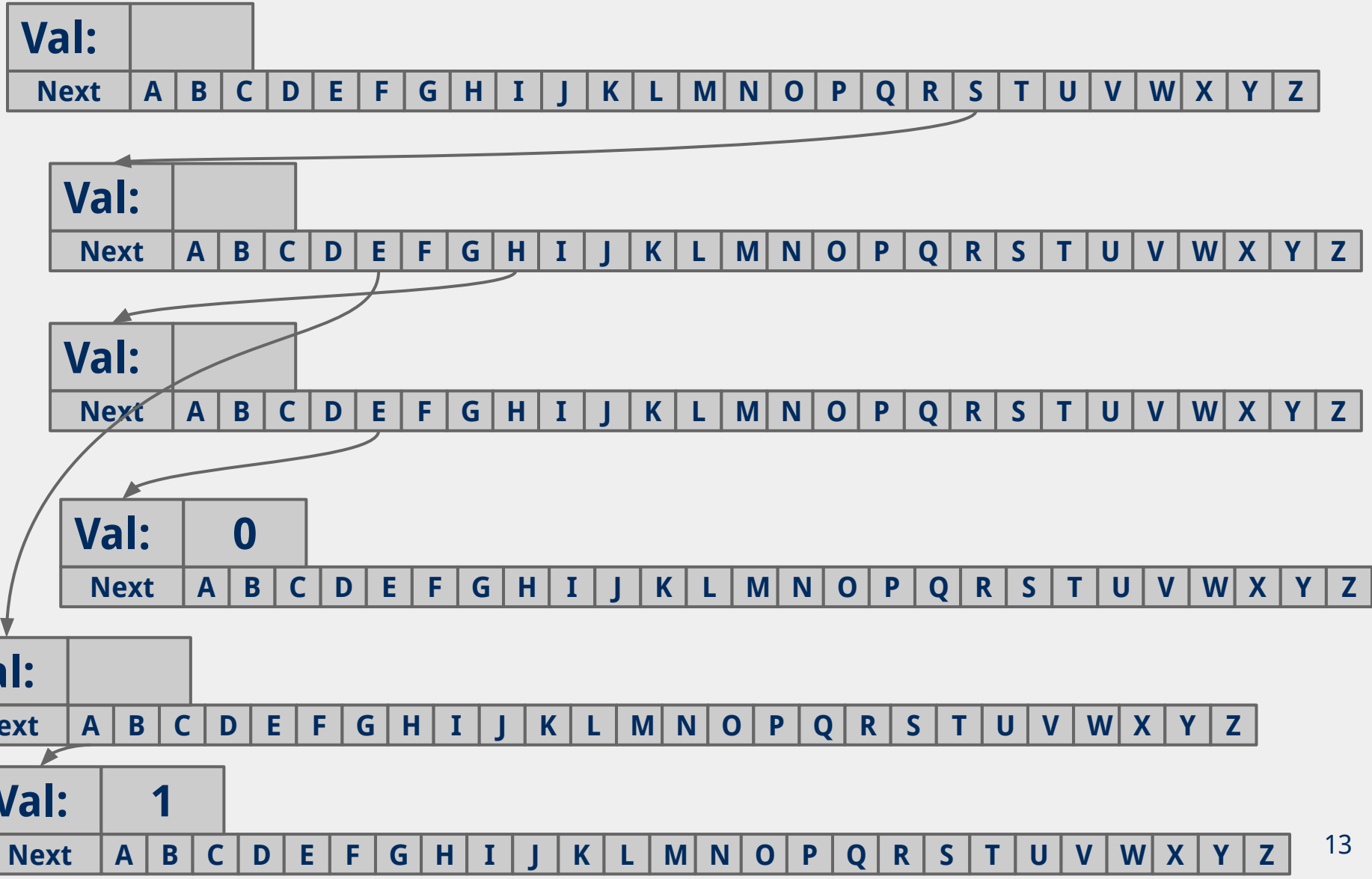
- See TrieSt.java
  - Implements an R-way trie
- Basic node object:

Where R is the branching factor

```
class Node:
    def __init__(self):
        self.val = None
        self.next = [None for i in range(R)]
```

- Non-null val means we have traversed to a valid key
- Again, note that keys are not directly stored in the trie at all

R-way trie example



Analysis

- Runtime?

Further analysis

- Miss times
  - Require an average of $\log_R(n)$ nodes to be examined
    - Where R is the size of the alphabet being considered
    - Proof in Proposition H of Section 5.2 of the text

  - Average # of checks with 2^{20} keys in an RST?
  - With 2^{20} keys in a large branching factor trie, assuming 8-bits at a time?

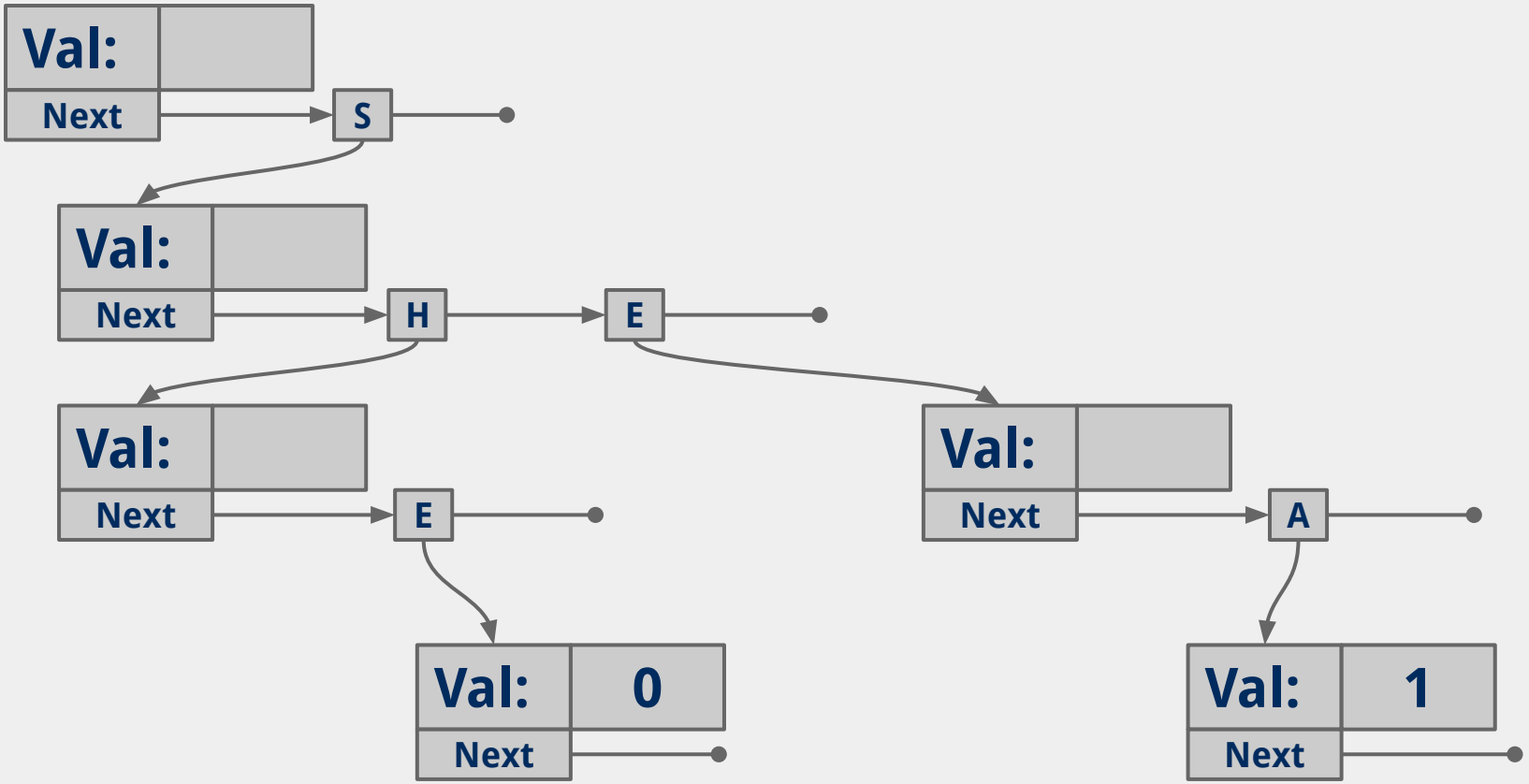
So what's the catch?

- Space!
  - Considering 8-bit ASCII, each node contains 2^8 references!
  - This is especially problematic as in many cases, alot of this space is wasted
    - Common paths or prefixes for example, e.g., if all keys begin with "key", thats 255*3 wasted references!
    - At the lower levels of the trie, most keys have probably been separated out and reference lists will be sparse

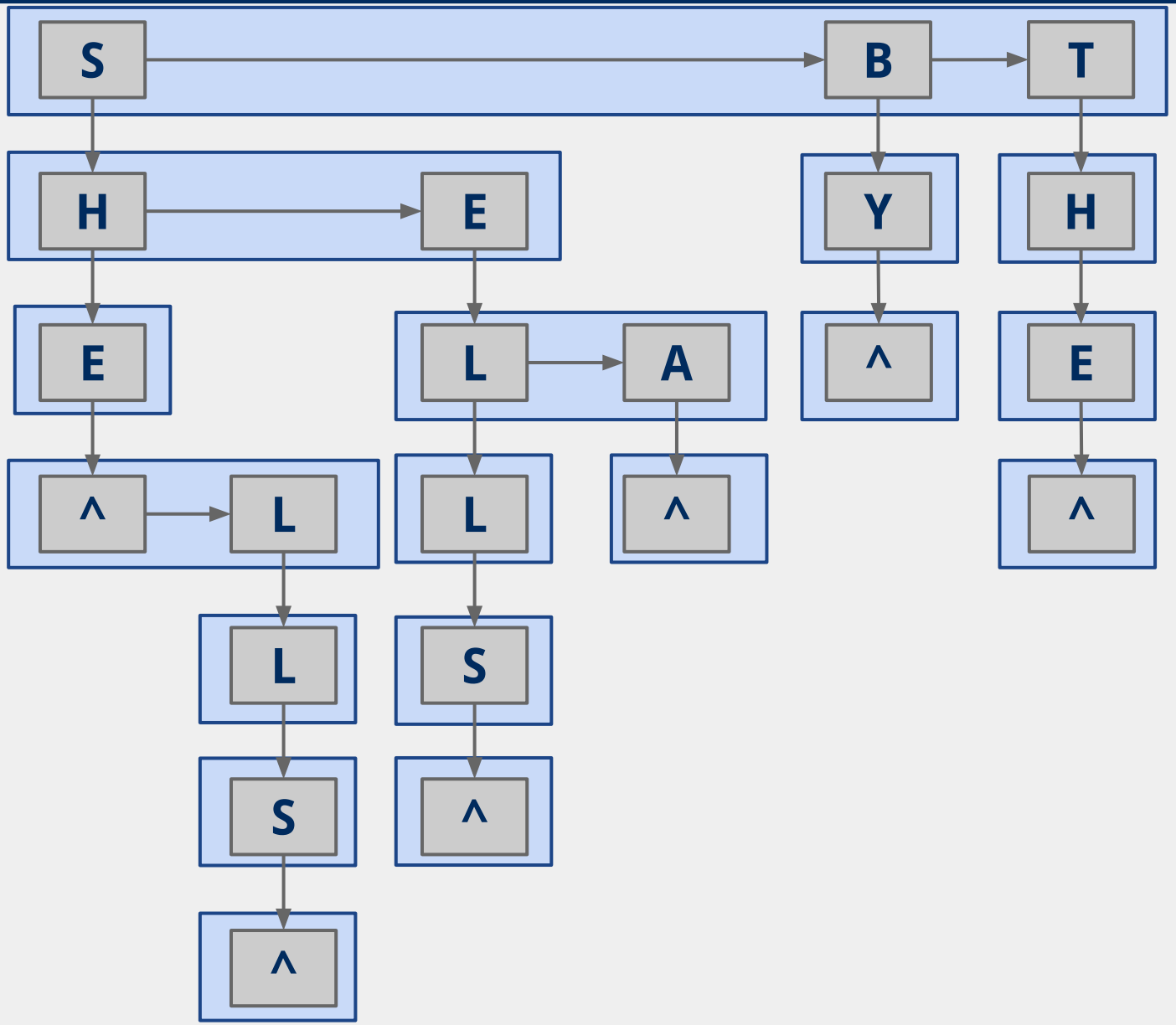
De La Briandais tries (DLBs)

- Replace the `.next` array of the R-way trie with a linked-list

DLB trie example



Another DLB Example



DLB analysis

- How does DLB performance differ from R-way tries?
- Which should you use?

Modifying the searching problem

- So far we've continually assumed each search would only look for the presence of a whole key
- What about if we wanted to know if our search term was a prefix to a valid key?