CS/COE 1501

www.cs.pitt.edu/~lipschultz/cs1501/

Searching

Symbol tables

- Abstract structures that link *keys* to *values*
  - Key is used to search the data structure for a value
  - Described as a class in the text, but probably more accurate to think of the concept of a symbol table in general as an interface
    - Key functions:
      - put()
      - contains()

Review: Searching through a collection

- Given a collection of keys C, how do we search for the value associated with a given key k?
  - Store collection in an array
    - Unsorted
    - Sorted
  - Linked list
    - Unsorted
    - Sorted
  - Binary search tree

- Differences?
- Runtimes?

A closer look

- BinarySearchST.java and BST.java present symbol tables based on sorted arrays and binary search trees, respectively
- 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:
    - Node ref is null, K not found
    - K is equal to the current 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 key, K is found
    - current bit of K is 0, continue to left child
    - current bit of K is 1, continue to right child

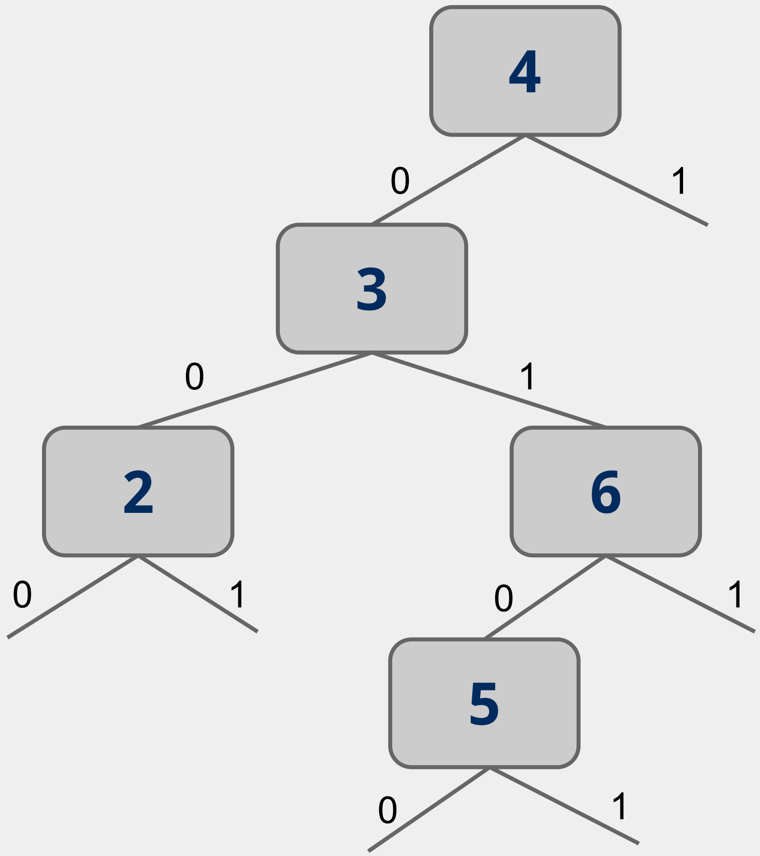
DST example

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

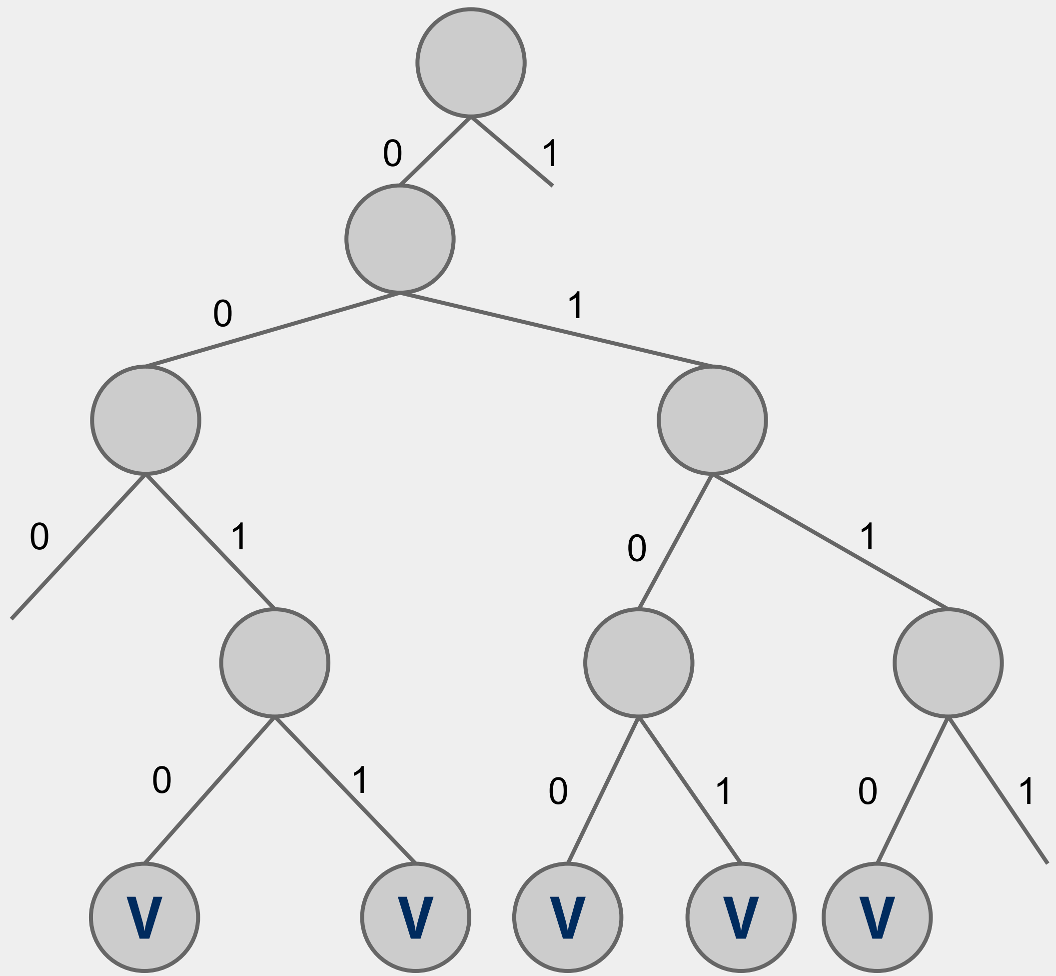
RST example

Insert:

4	0100
3	0011
2	0010
6	0110
5	0101

Search:

3	0011
7	0111



RST analysis

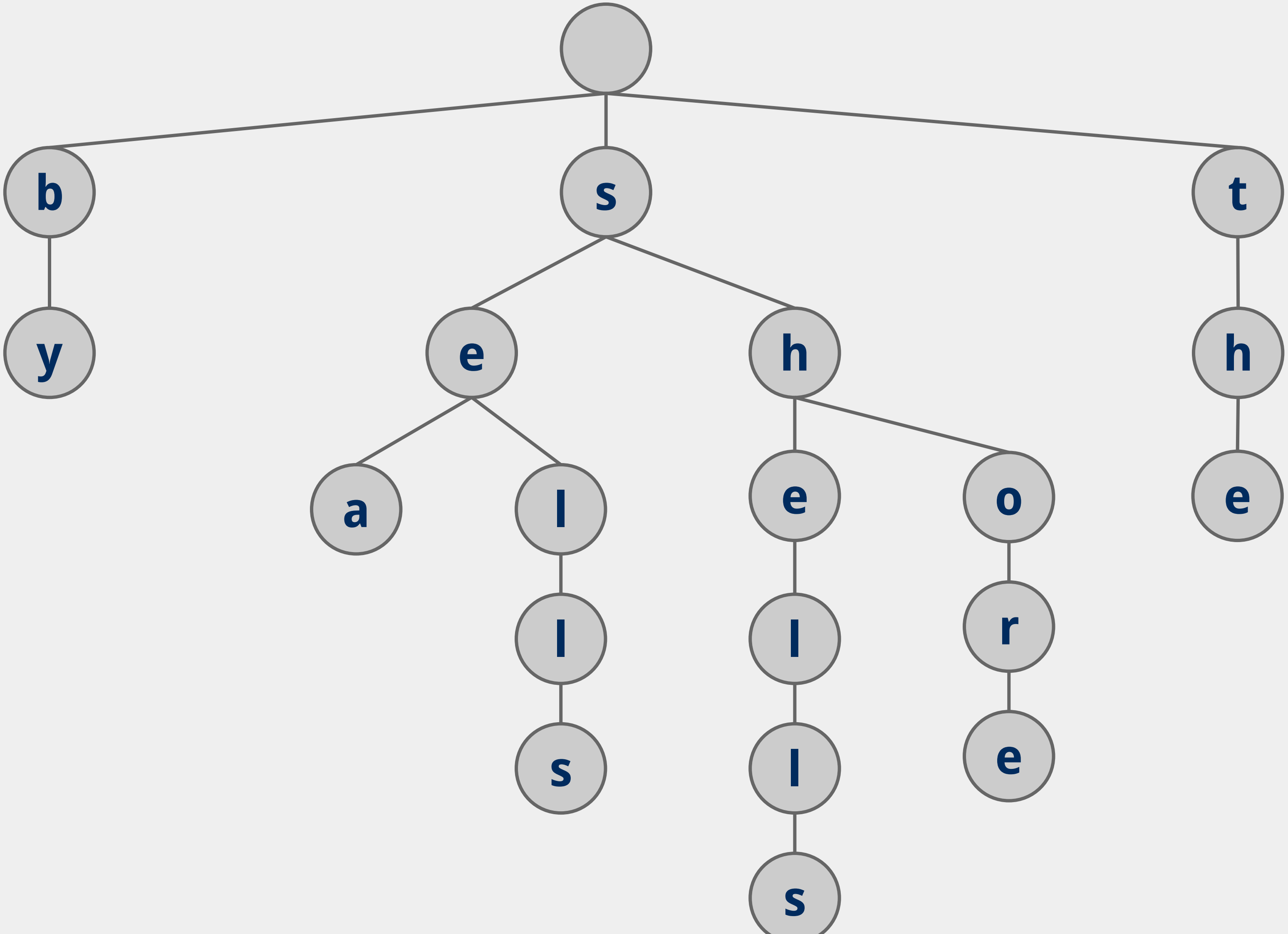
- Runtime?

- 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 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



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 an R-way trie, assuming 8-bit ASCII?

Implementation Concerns

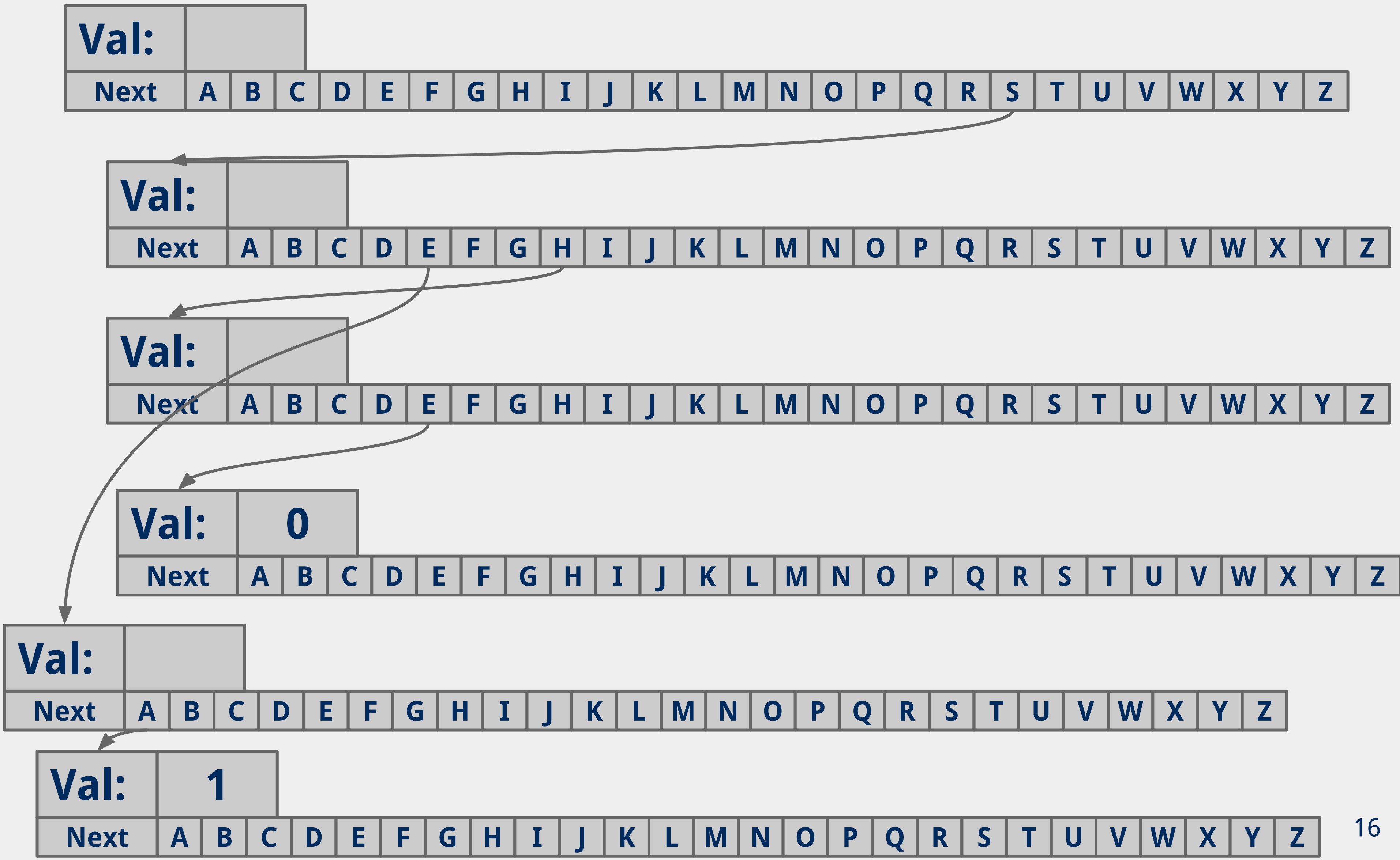
- See TrieSt.java
    - Implements an R-way trie
- Basic node object:

Where R is the branching factor

```
private static class Node {
    private Object val;
    private Node[] next = new Node[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



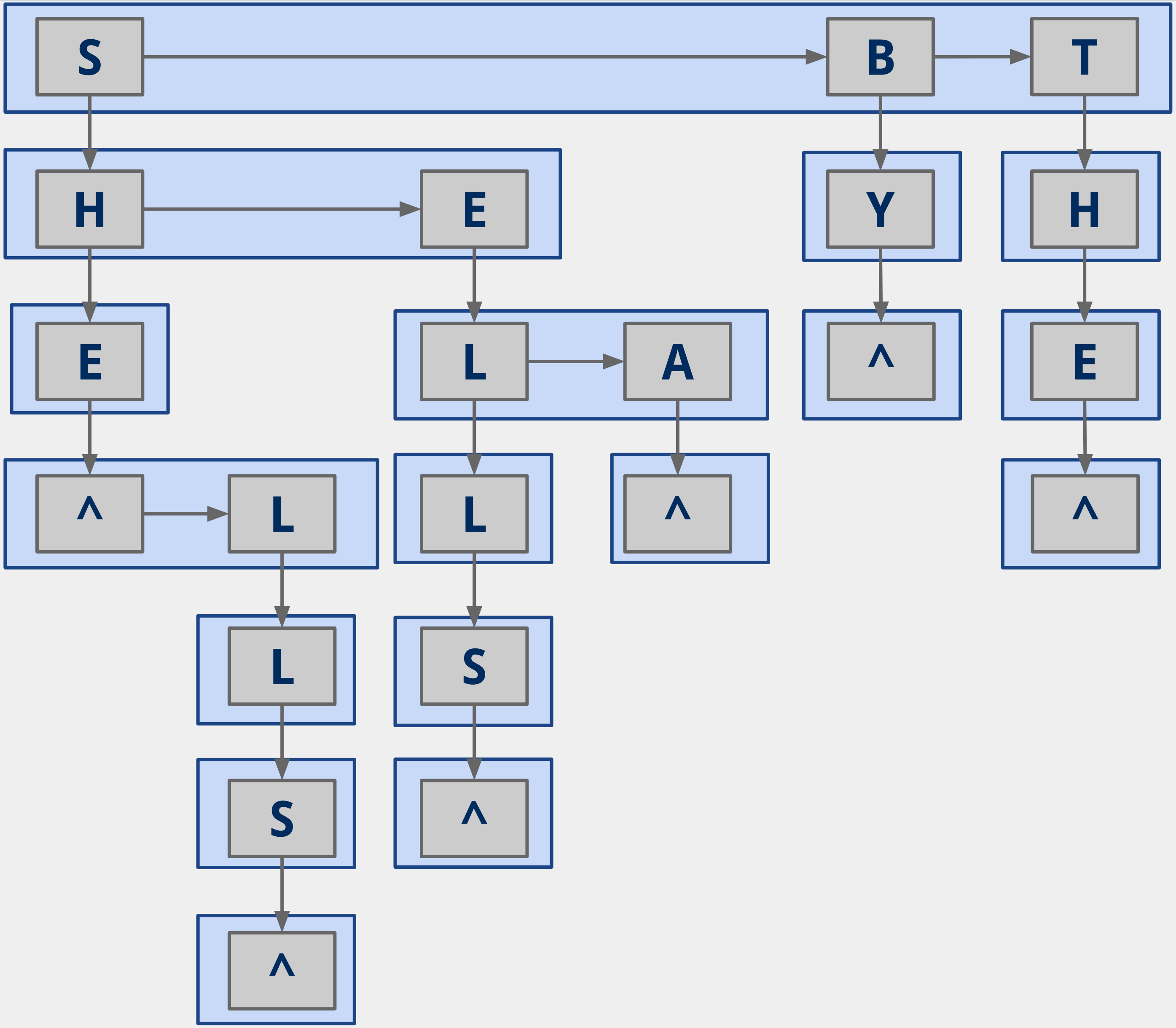
So what's the catch?

- Space!
  - Considering 8-bit ASCII, each node contains 2^8 references!
  - This is especially problematic as in many cases, a lot 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
- How does this affect trie performance?
    - No wasted space!
    - But search/insert are now Θ(kR)
        - In the worst case, we have to iterate through all R characters in the alphabet for each node
        - For implementations with a lot of sparse nodes are expected, use a DLB
            - Runtime will still be close to Θ(k) for sparse nodes
        - For dense nodes, stick with R-way tries
            - If most of the alphabet is a valid reference for most nodes, you won't get a whole lot of space savings with DLBs

DLB Example



DLB analysis

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

Searching

- So far we’ve continually assumed each search would only look for the presence of a whole key, what about prefix search as was needed for Boggle?

Final notes

- This lecture does not present an exhaustive look at search trees/tries, just the sampling that we're going to focus on
- Many variations on these techniques exist and perform quite well in different circumstances
  - Red/black BSTs
  - Ternary search Tries
  - R-way tries without 1-way branching
- See the table at the end of Section 5.2 of the text