CS 106B, Lecture 21
Other Kinds of Trees

Plan for Today

- Non-Binary Trees
  - Tries (how to implement a Lexicon)
  - B-Trees (how to implement a database)
  - Idea: each node can store more than two pointers and have more than two children
    - Generally store pointers in a data structure

The Lexicon

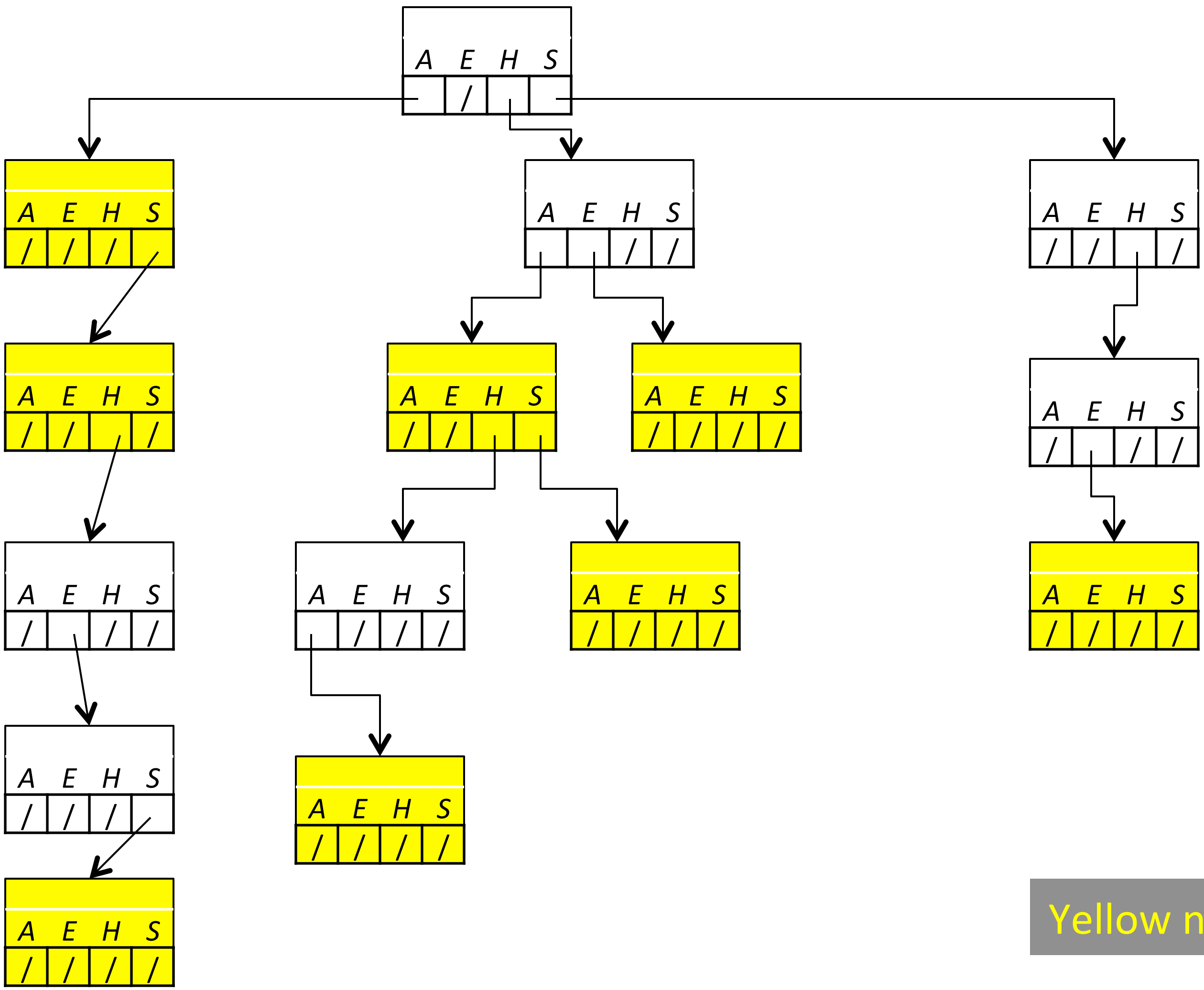
- Lexicons are good for storing words
  - `contains`
  - `containsPrefix`
  - `add`
- Implemented with a **trie**

Trie (prefix tree)

- **trie** ("try"): A tree structure optimized for "prefix" searches
  - e.g. Do any words in the set begin with the prefix `"ash"`?
    - `containsPrefix`
  - The idea: instead of a binary tree, store a pointer for each character in the alphabet
  - For English: each node has 26 children for A-Z
    - We're going to use a simpler alphabet for the tries in class: {A, E, H, S}

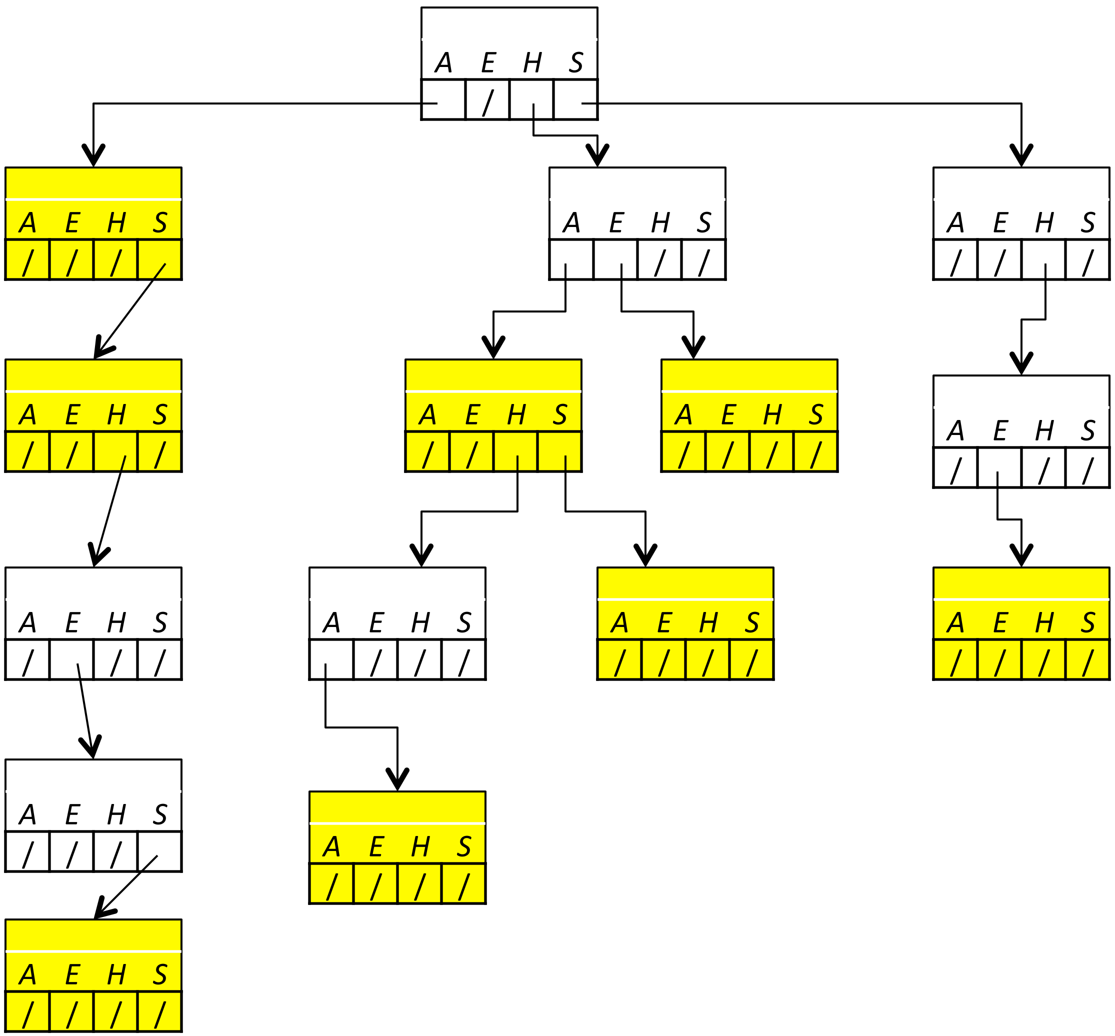
```
struct TrieNode {
    bool isWord;
    TrieNode * children[26];
    // storing children depends on the alphabet
};
```

Let's "Trie" an Example



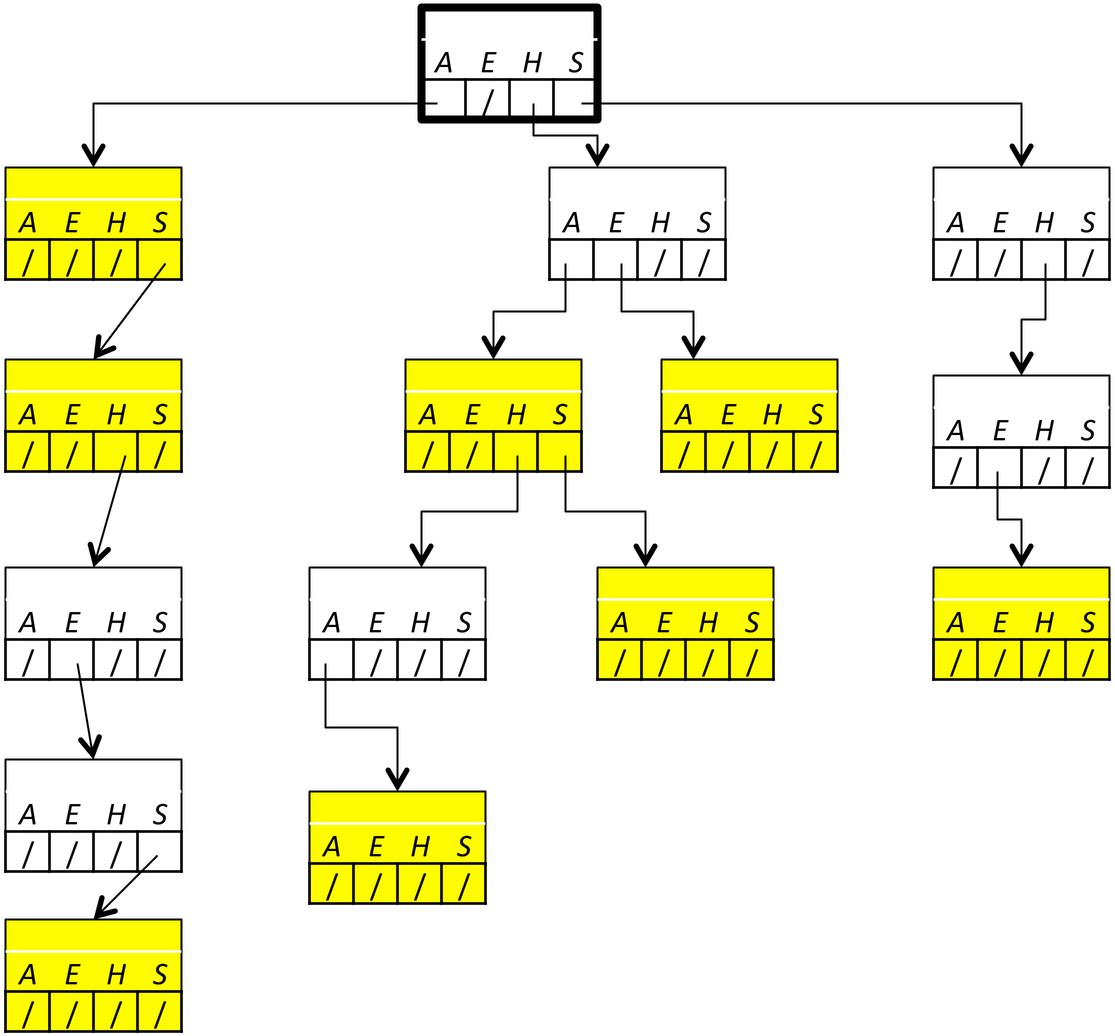
Yellow nodes are words!

Reading Words



- Start at root – corresponds to empty string
- Every pointer we travel contributes one character to our final string

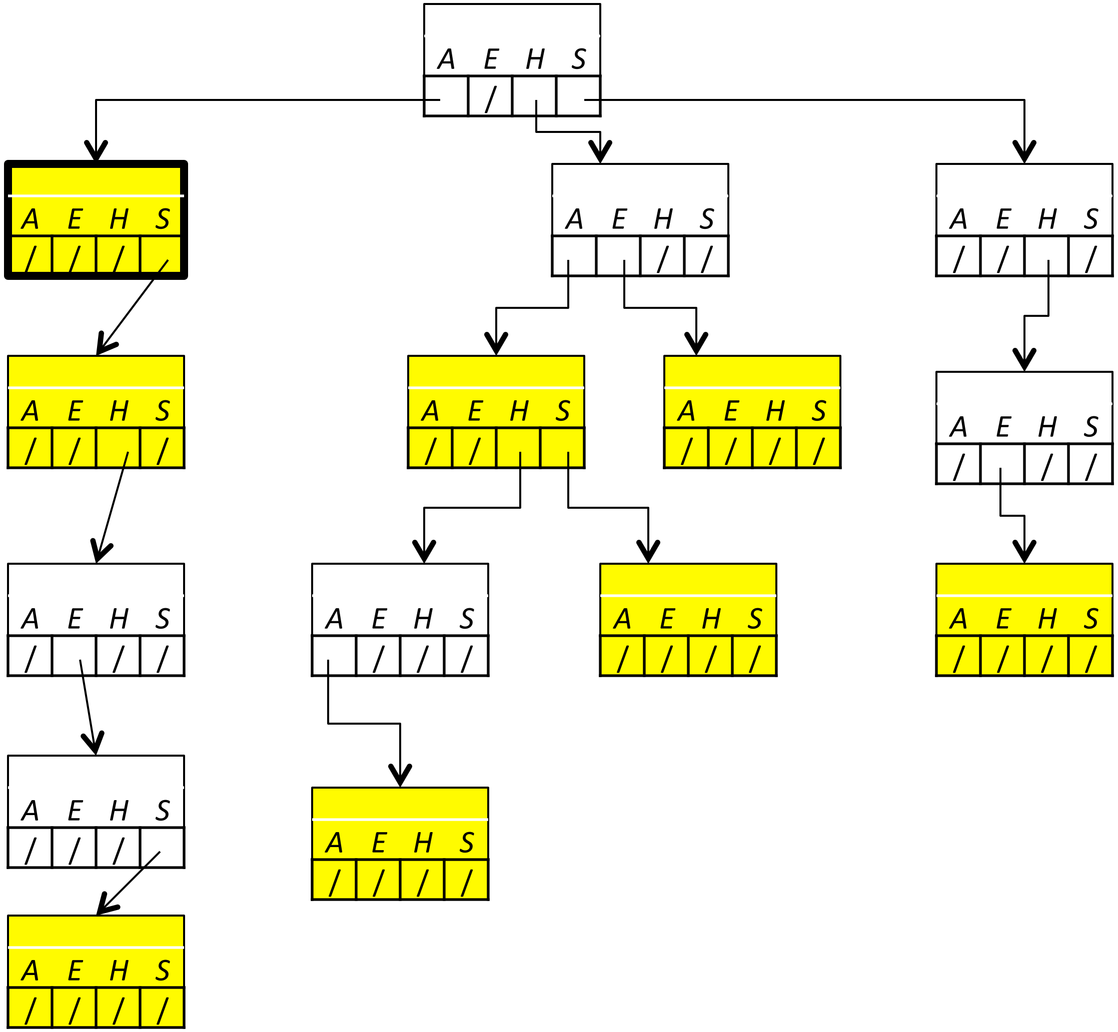
Reading Words



- Example:

""

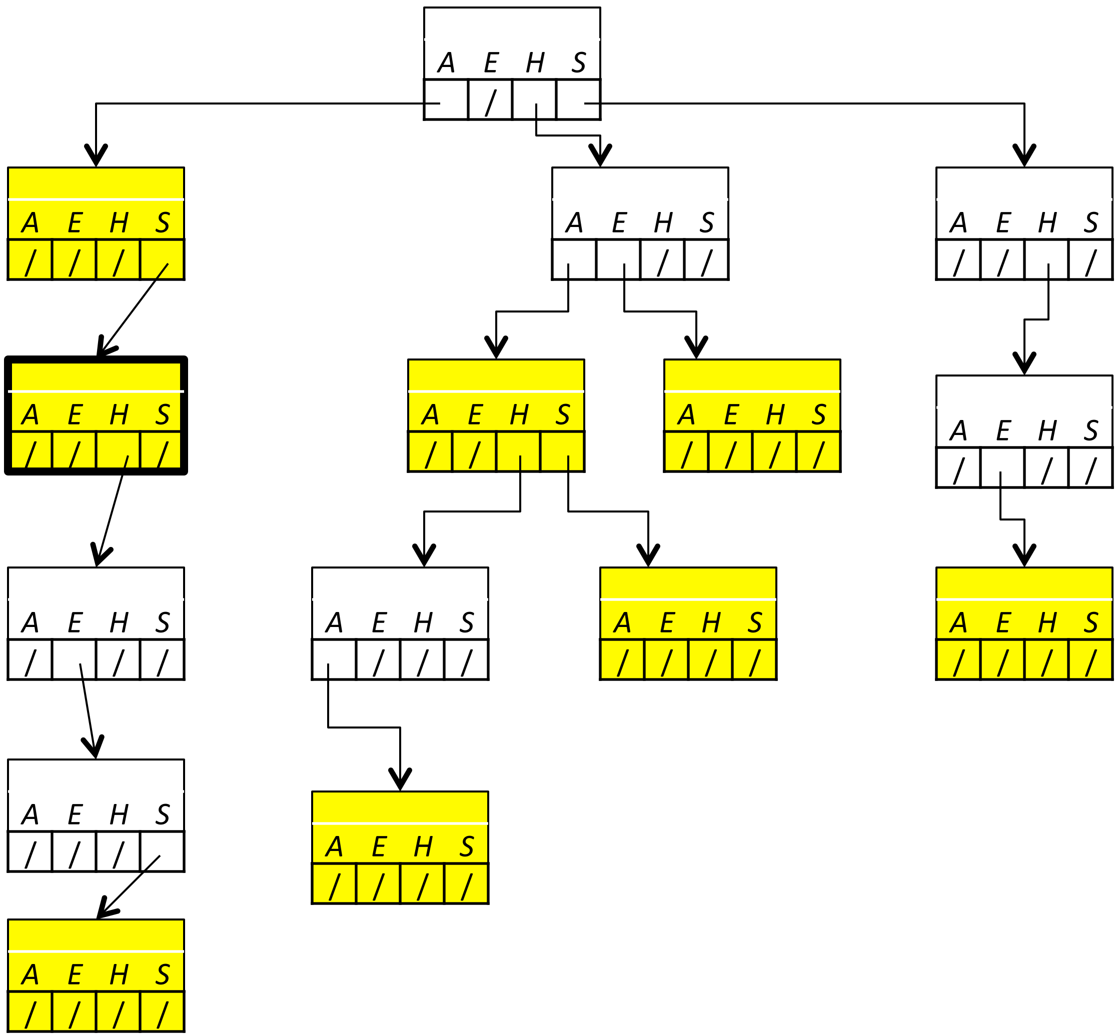
Reading Words



- Example:

"a"

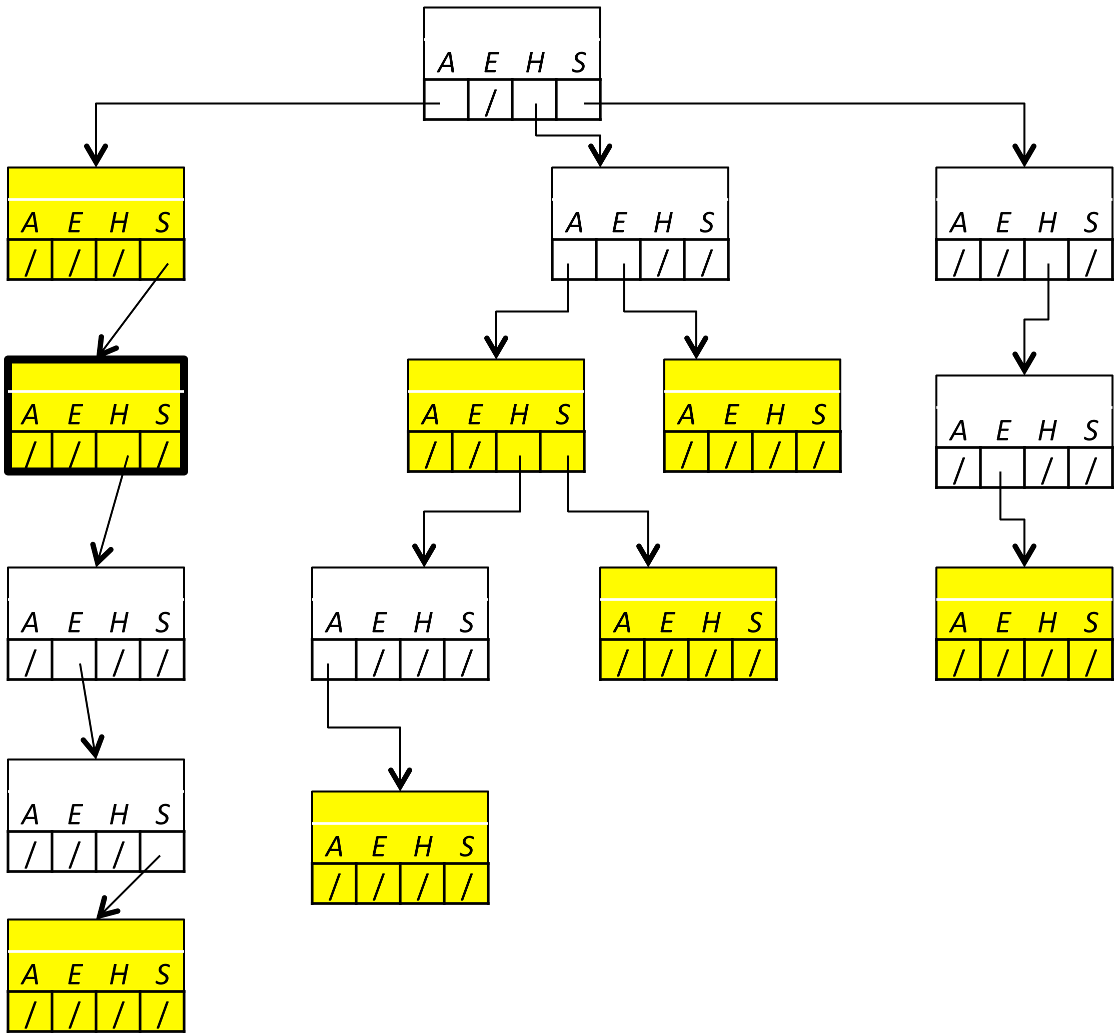
Reading Words



- Example:

"as"

Reading Words

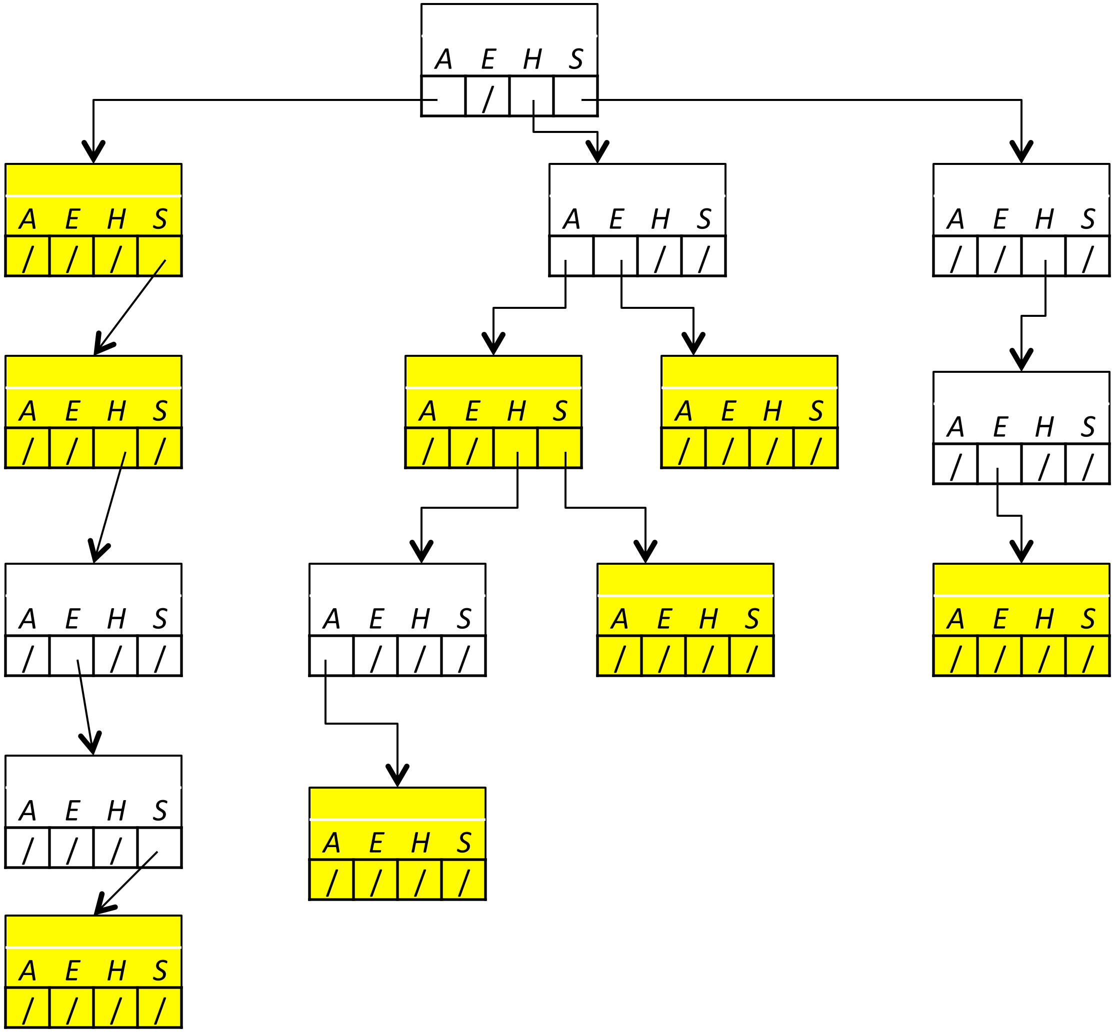


- Example:

  "as"

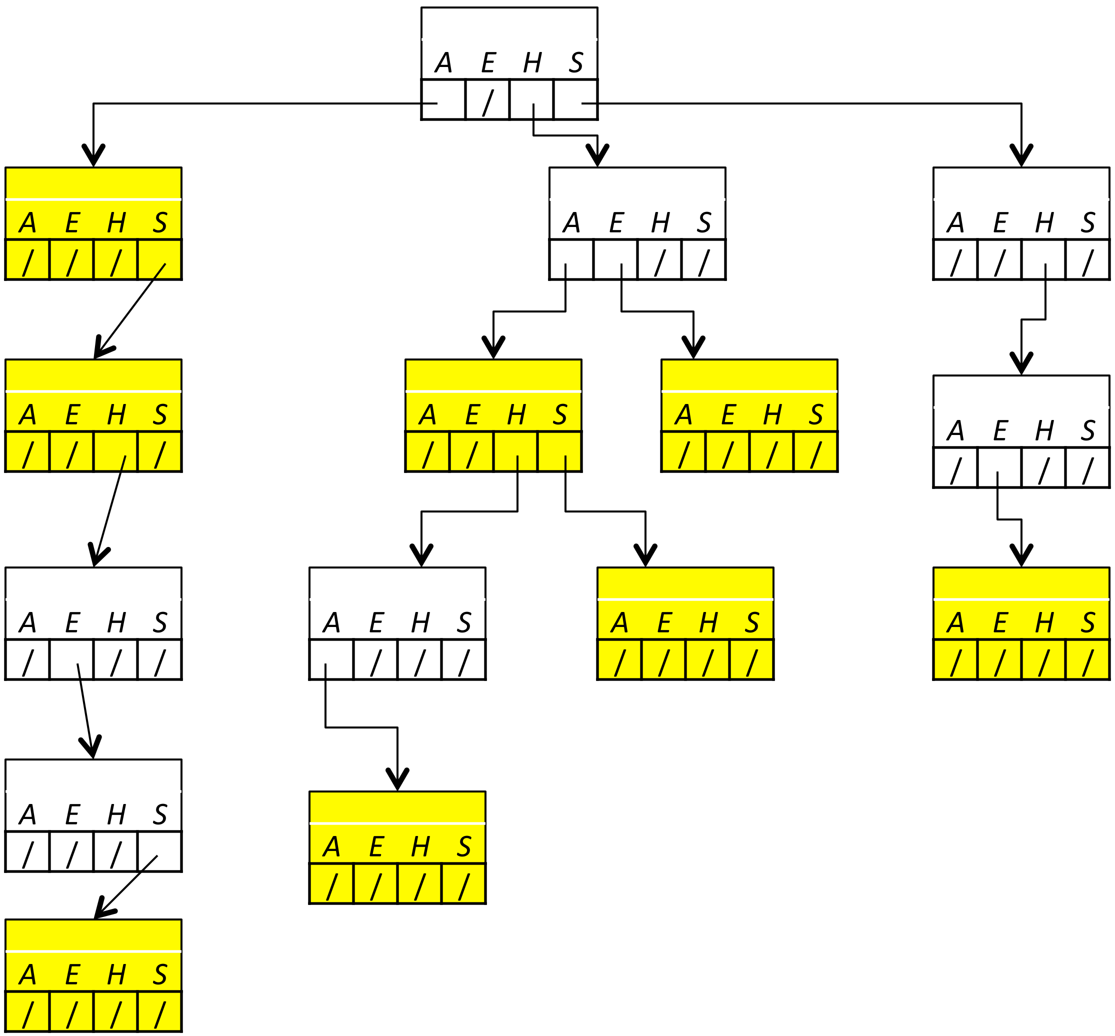
  "as" is a word because its corresponding node is yellow (meaning `isWord` is true)

Reading Words



- What are all the words in this trie?

Reading Words



- What are all the words in this trie?

a

as

ashes

ha

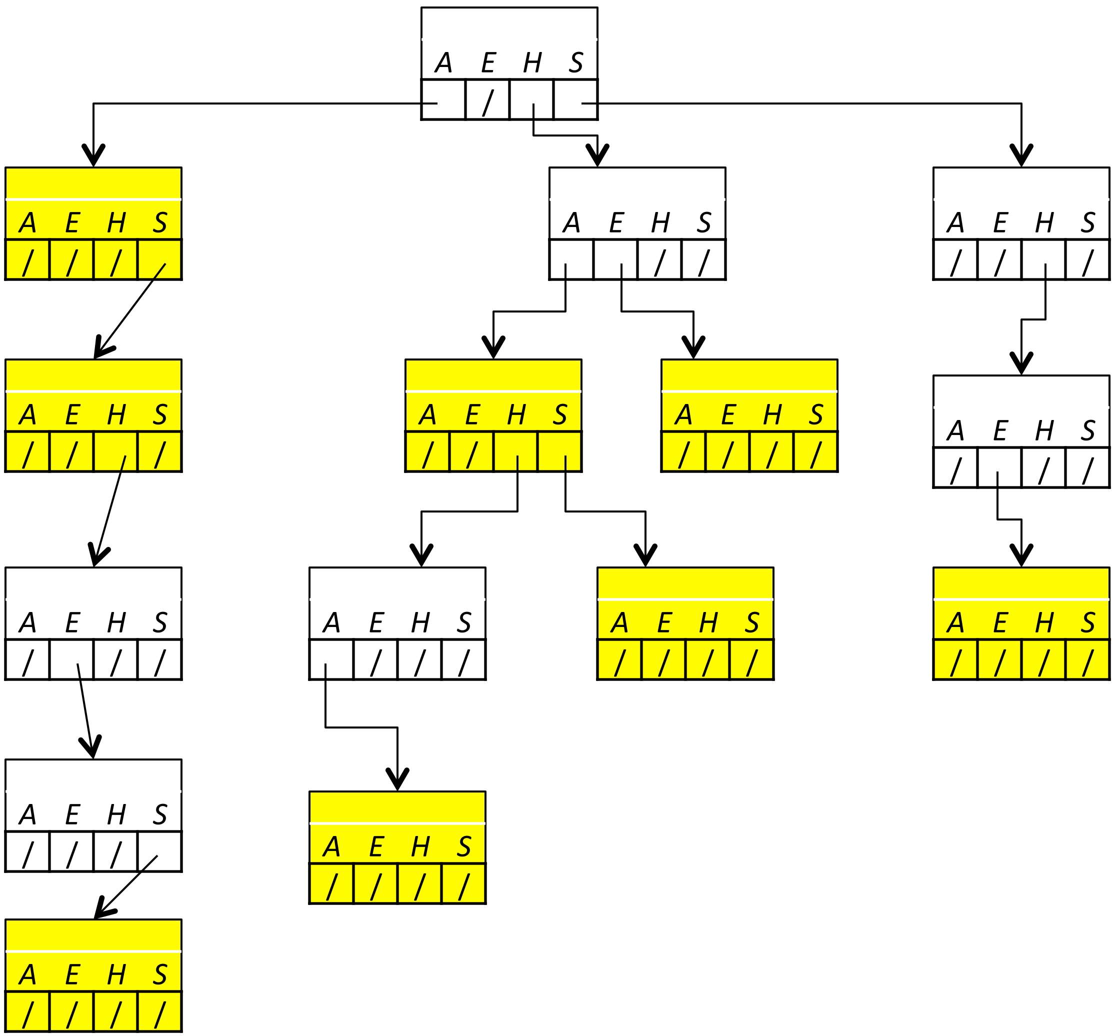
haha

has

he

she

PrintAllWords



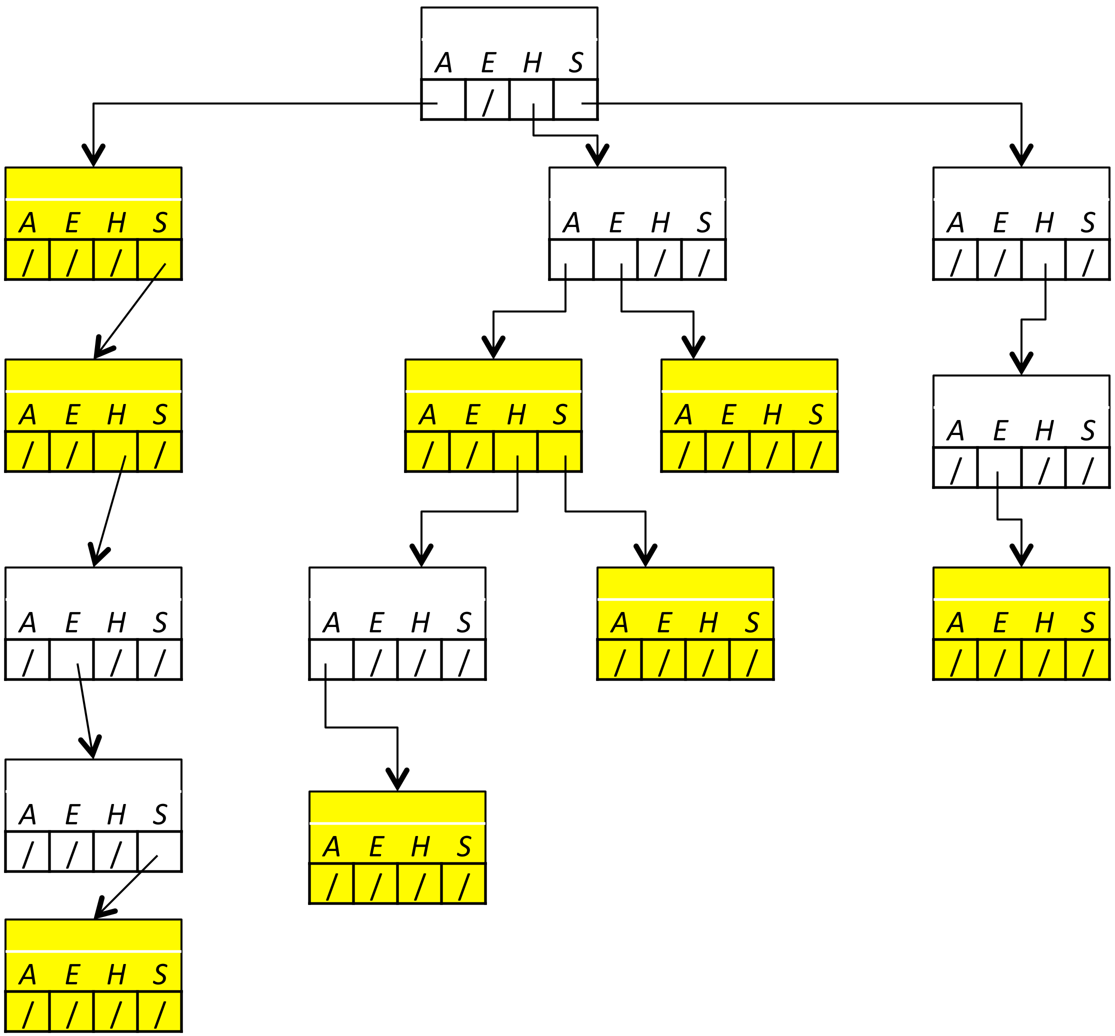
- How could we write a function that prints all words in a trie?

PrintAllWords

```
void printAllWords(TrieNode *root) {
    printAllWordsHelper(root, "");
}

void printAllWordsHelper(TrieNode *root, string str) {
    if (root == nullptr) {
        return;
    }
    if (root->isWord) {
        cout << str << endl;
    }
    for (int i = 0; i < 26; i++) {
        printAllWordsHelper(root->children[i], str + char('a' + i));
    }
}
```

ContainsPrefix



- How could we write containsPrefix?
  - containsPrefix("a") = true
  - containsPrefix("hahas") = false
  - What are some prefixes that don't exist in this trie?

containsPrefix

```
bool containsPrefix(TrieNode* node, string prefix) {
    if (node == nullptr) {
        return false;
    }
    if (prefix.length() == 0) {
        return true;
    }
    return containsPrefix(node->children[prefix[0] - 'a'],
                          prefix.substr(1));
}
```

Announcements

- You should be finishing MiniBrowser's Cache today. LineManager is hard. The last part is a trie, which you can get started with now ☺
- Please give us feedback! cs198.stanford.edu
- Feel free to use seepluspl.us to help you understand trees or pointers. It's still in development, so be patient with quirks
- I read your feedback, and several people wanted more real-world examples of concepts in class. Let's talk about databases

Databases

- Computers are famous for storing lots of information for fast retrieval
- Common solution: databases
  - Store keys and values (like a fancy map) but can have millions or billions of "records" (key-value pairs)
  - Common example: return all students who are at least 21
  - Another example: give me the record associated with "Ashley Taylor"
- Basically, just a BST

Database Problems

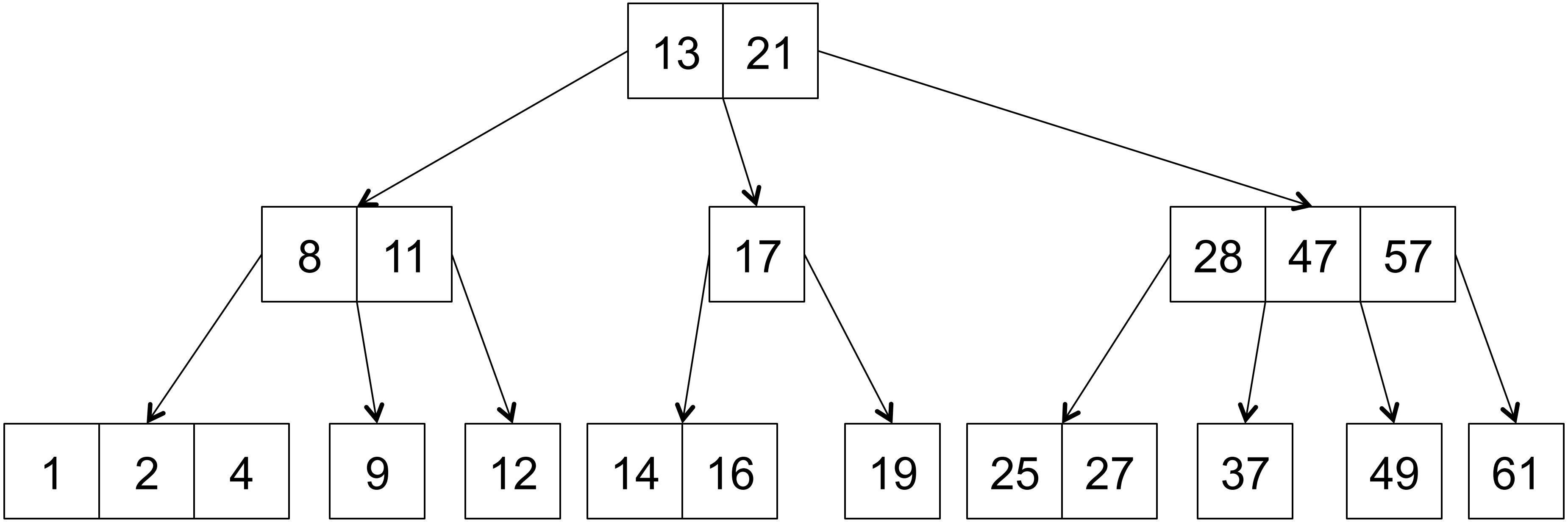
- Databases can't store all the information in main memory
  - Have to read from "disk", which is VERY slow
  - For the purposes of this class, reading a small chunk of memory from disk takes the same amount of time as reading a large chunk of memory
- Problem: each binary search tree node is pretty small, and we have to read a lot (O(log N)) of them

Database Problems

- Idea: what if we stored more elements per node in a BST?
  - If we store 3 elements per node, we cut out ¾ of the tree at each level, so we'll reach the leaf nodes in half the number of disk reads

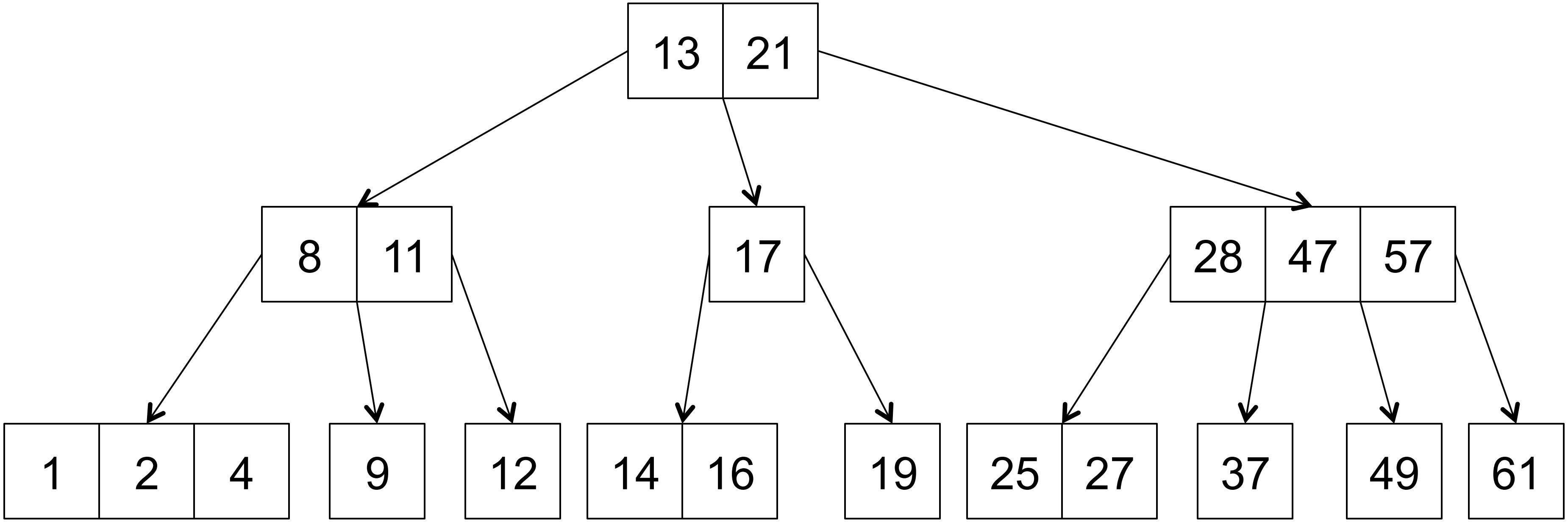
B-Tree

- Idea: besides the root, every node has between *k* and *2k* children (and between *k - 1* and *2k - 1* elements)
- Below is a B-Tree with *k = 2*
  - Nodes have between 2 and 4 children
- All leaf nodes are at the same height (balanced)



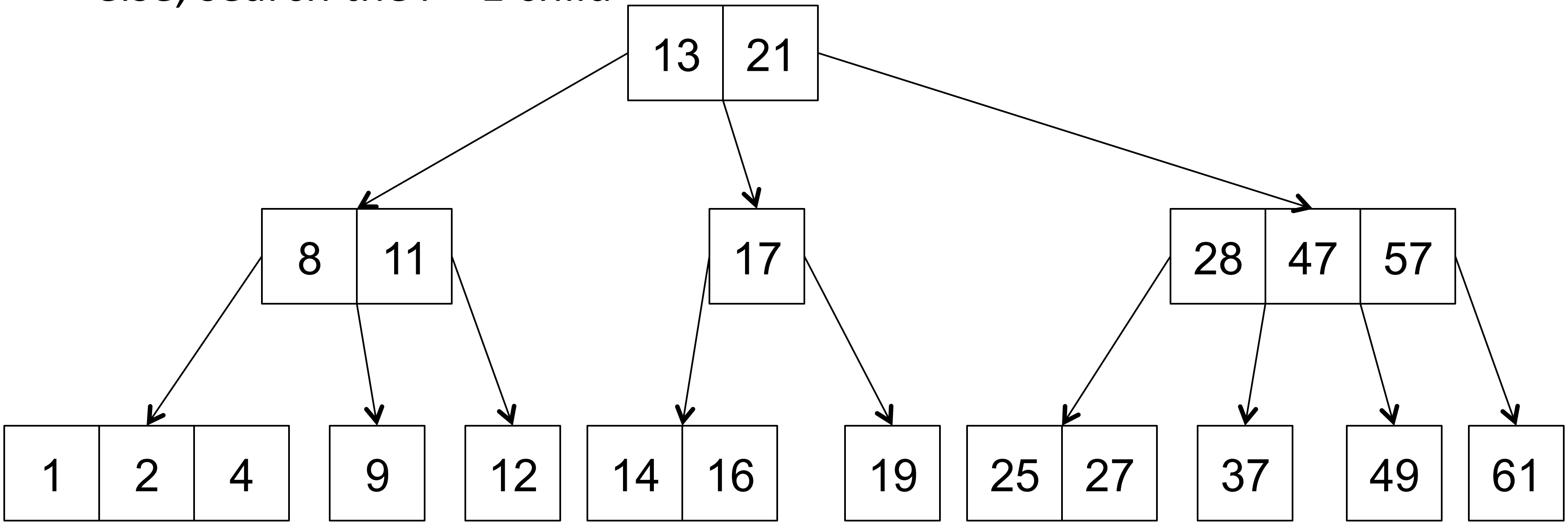
B-Tree and Contains

- How would we write `contains` for a B-tree?



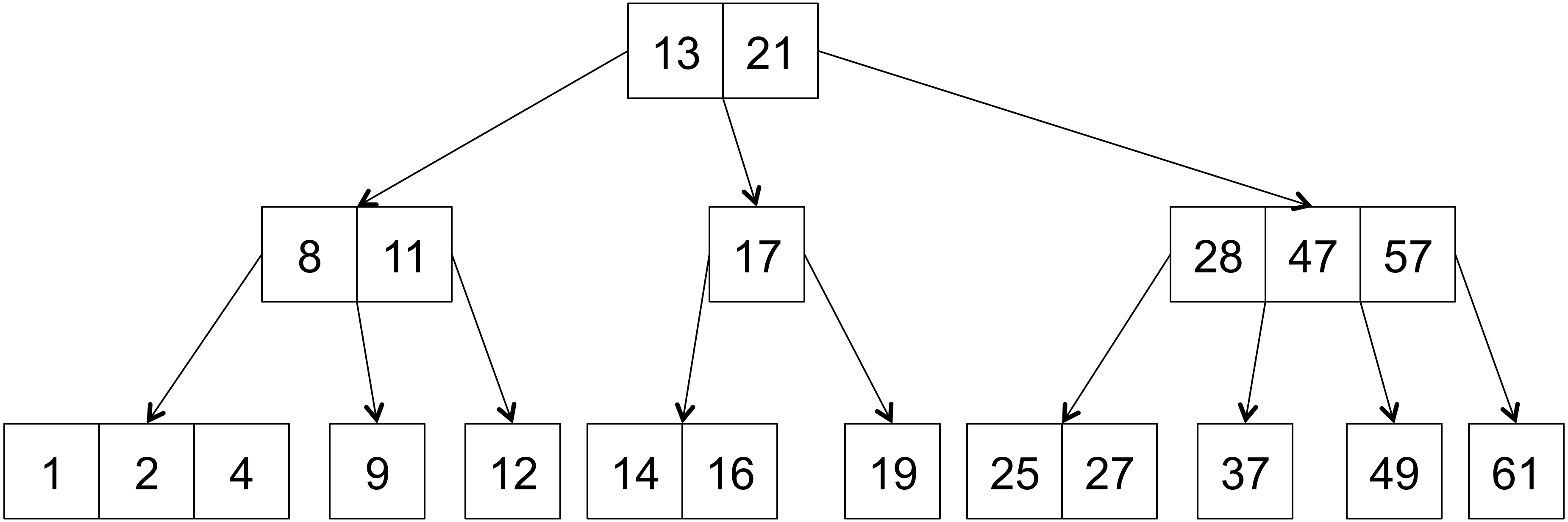
B-Tree and Contains

- How would we write `contains` for a B-tree?
- Start at root:
  - closest element (at index *i*) is the smallest element in the root <= to the target [we can binary search!]
  - if closest element is equal to target, we've found it
  - else, search the *i* + 1 child



Printing B-Trees

- How would we print the tree in-order?



Printing B-Trees

- How would we print the tree in-order?
- Print the 0^{th} subtree, then the 0^{th} element, then the 1^{st} subtree, then the 1^{st} element…