CS 106B, Lecture 22
Advanced Binary Trees

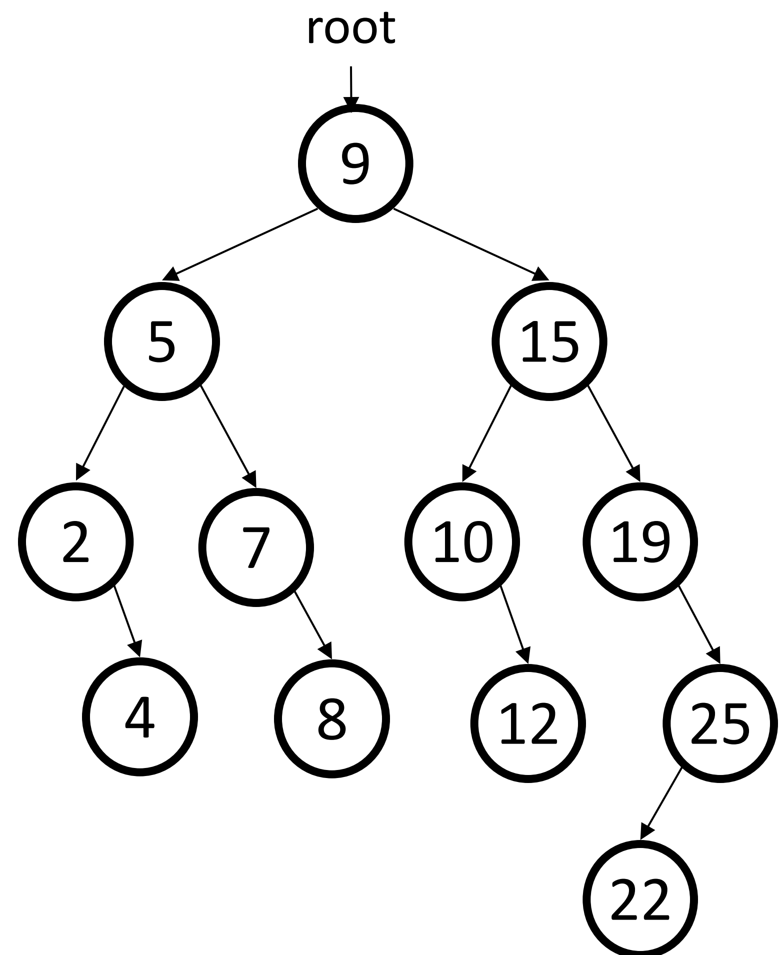
This document is copyright (C) Stanford Computer Science and Ashley Taylor, licensed under Creative Commons Attribution 2.5 License. All rights reserved.
Based on slides created by Marty Stepp, Chris Gregg, Keith Schwarz, Julie Zelenski, Jerry Cain, Eric Roberts, Mehran Sahami, Stuart Reges, Cynthia Lee, and others

Plan for Today

- Finish up BSTs
  - Removal
- Advanced (Balanced) BSTs
- Non-BST binary trees
  - Heaps
  - Cartesian Trees
- Non-Binary Trees
  - Tries (how to implement a Lexicon)

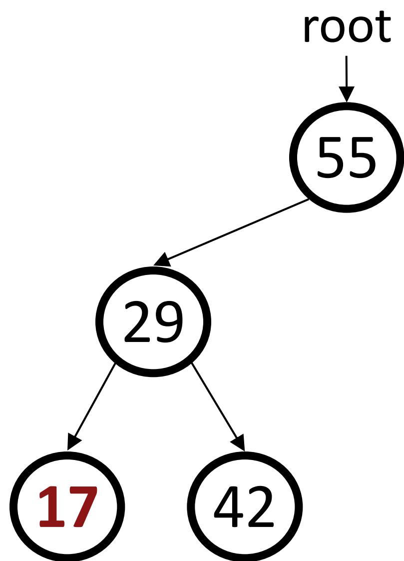
Removing from a BST

- Suppose we want to **remove** values from the BST below.
  - Removing a leaf like 4 or 22 is easy.
  - What about removing 2? 19?
  - How can you remove a node with two large subtrees under it, such as 15 or 9?

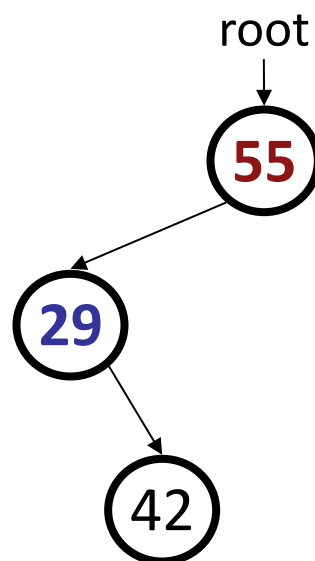


Cases for removal

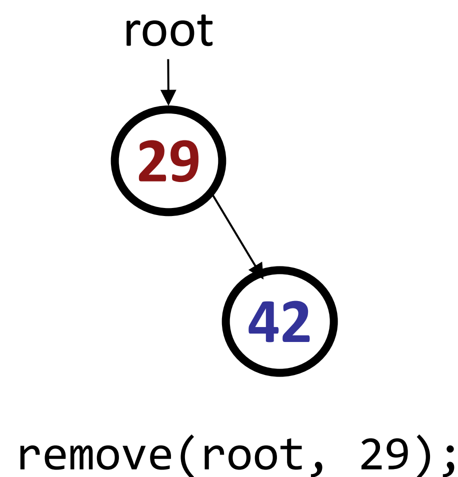
1. a **leaf**: Replace with `nullptr`
2. a node with a **left child only**: Replace with left child
3. a node with a **right child only**: Replace with right child

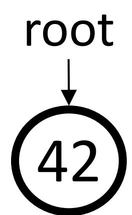


`remove(root, 17);`



`remove(root, 55);`



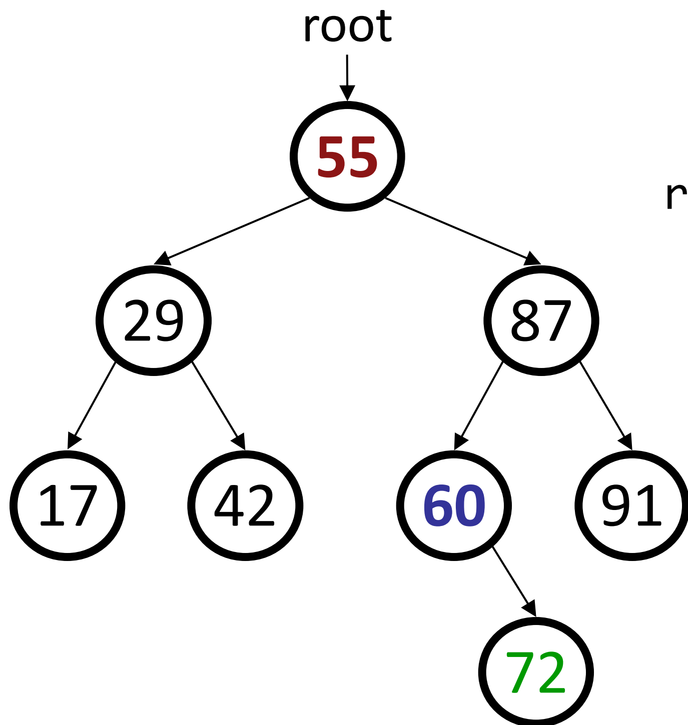


`remove(root, 29);`

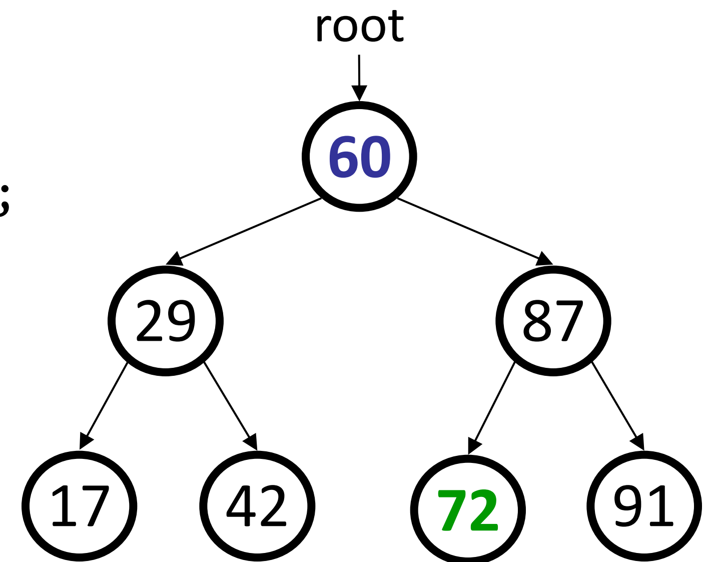
Cases for removal

4. a node with **both** children:

replace with **min from right**
(replacing with **max from left** would also work)

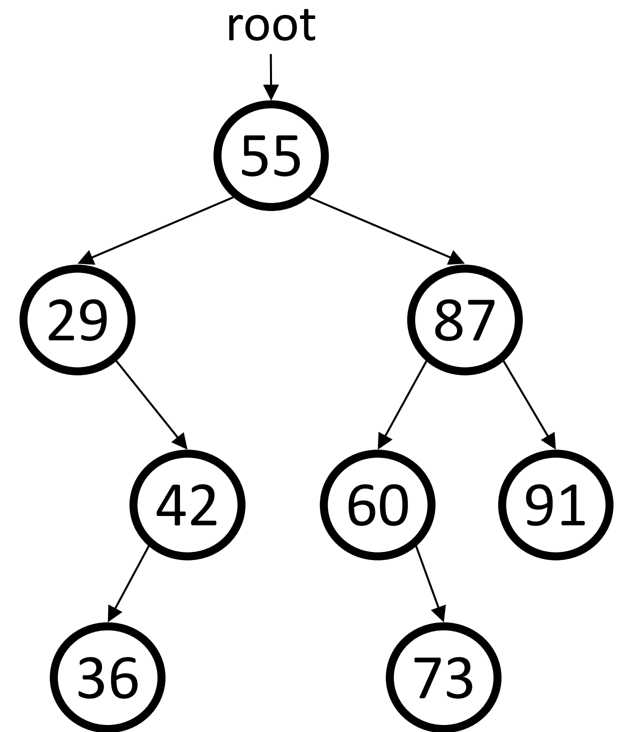



```
remove(root, 55);
```



Exercise: remove

- Add a function **remove** that accepts a root pointer and removes a given integer value from the tree, if present. Remove the value in such a way as to maintain BST ordering.
  - `remove(root, 73);`
  - `remove(root, 29);`
  - `remove(root, 87);`
  - `remove(root, 55);`



remove solution

```
// Removes the given value from this BST, if it exists.
// Assumes that the given tree is in valid BST order.
void remove(TreeNode*& node, int value) {
    if (node == nullptr) {
        return;
    } else if (value < node->data) {
        remove(node->left, value);    // too small; go left
    } else if (value > node->data) {
        remove(node->right, value);   // too big; go right
    } else {
        // value == node->data; remove this node!
        // (continued on next slide)
        ...
```

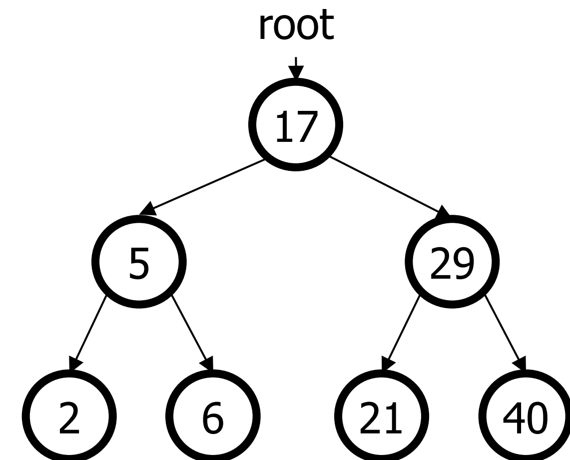
remove solution

```
        // value == node->data; remove this node!
        if (node->right == nullptr) {
            // case 1 or 2: no R child; replace w/ left
            TreeNode* trash = node;
            node = node->left;
            delete trash;
        } else if (node->left == nullptr) {
            // case 3: no L child; replace w/ right
            TreeNode* trash = node;
            node = node->right;
            delete trash;
        } else {
            // case 4: L+R both; replace w/ min from right
            int min = getMin(node->right);
            remove(node->right, min);
            node->data = min;
        }
    }
}
```

Implementing TreeSet and TreeMap

A BST set class

```
// TreeSet.h
// A set of integers represented as a binary search tree.
class TreeSet {
    members;
    ...
private:
    TreeNode* root;
};
```



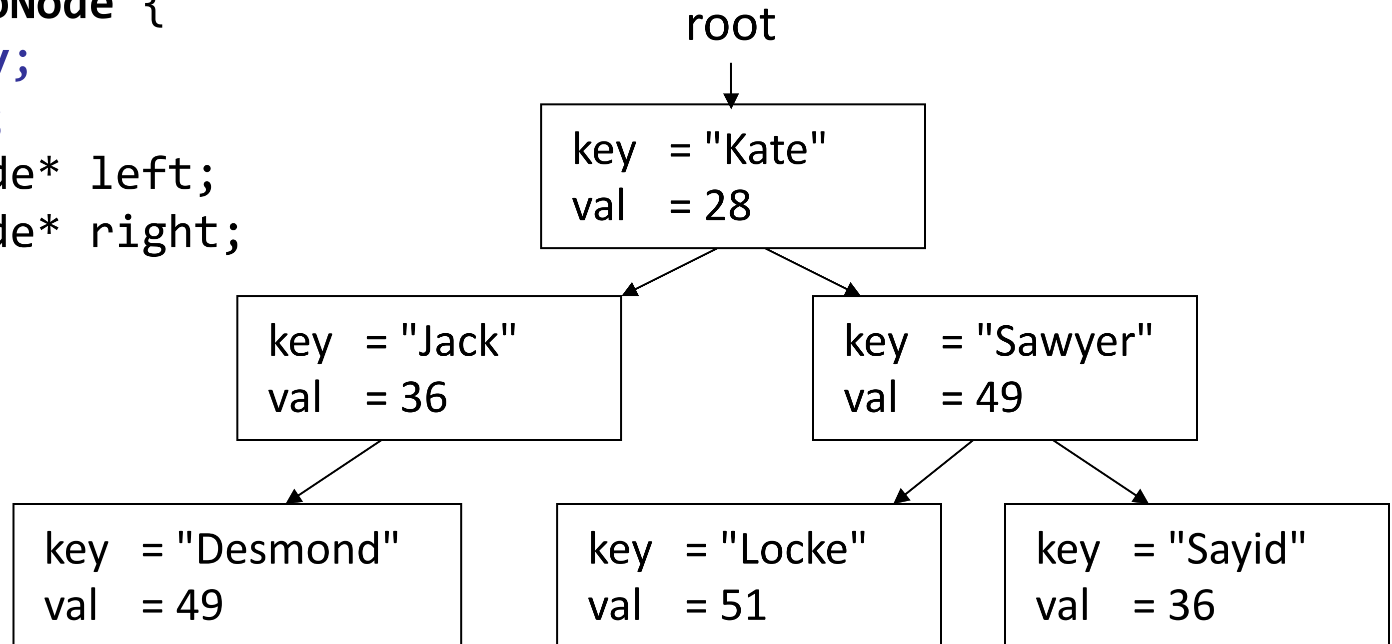
- This is basically how Stanford library's `Set` class is implemented.

- Provides abstraction. You have been using a set the whole class without knowing how a BST works.

Tree maps

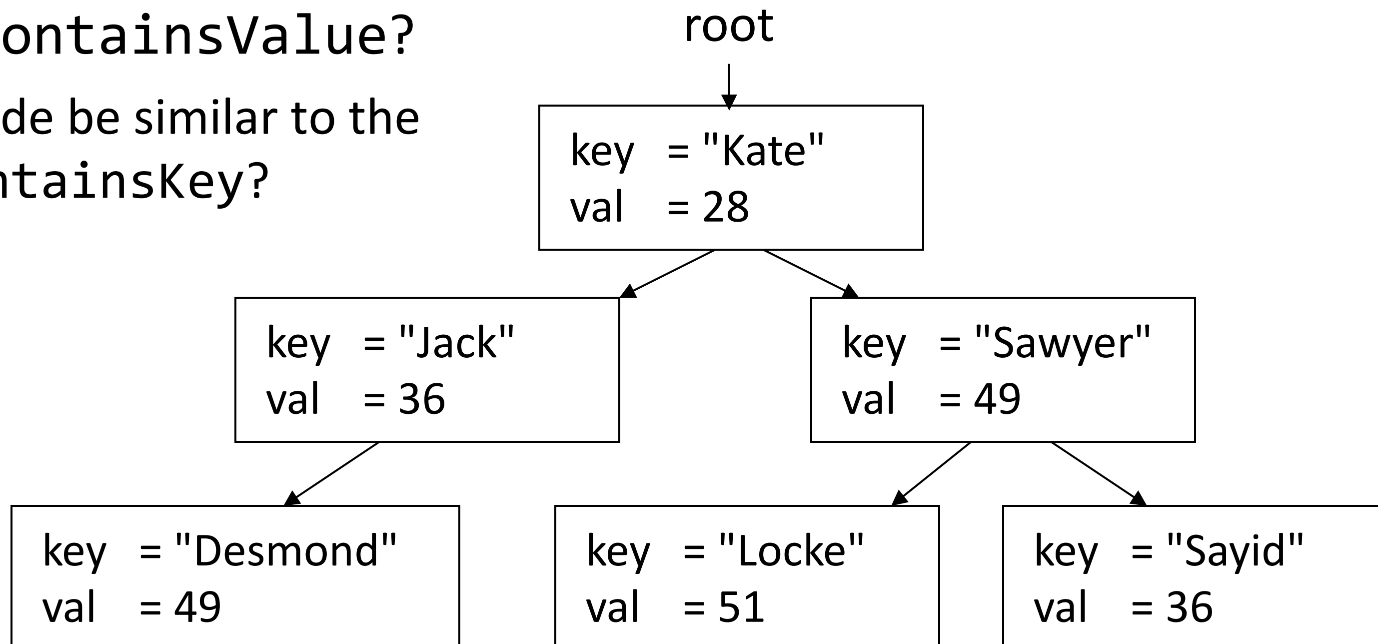
- Converting a tree set into a **tree map**:
  - Each tree node will store both a *key* and a *value*
  - tree is BST-ordered by its keys
  - **keys must be comparable (have a < operator) for ordering**

```
struct TreeMapNode {
    string key;
    int value;
    TreeMapNode* left;
    TreeMapNode* right;
};
```



Tree map details

- Each tree set operation corresponds to one in the tree map:
  - `add(`***`value`***`)` → `put(`***`key`***`, `***`value`***`)`
  - `contains(`***`value`***`)` → `containsKey(`***`key`***`)`
  - `remove(`***`value`***`)` → `remove(`***`key`***`)`
  - must add an operation: `get(`***`key`***`)`
  - What about `containsValue`?
    - Would its code be similar to the code for `containsKey`?



Announcements

- Assn. 6 Due Thursday
- Midterm regrade requests close tonight at 5PM
- Kate’s OH moved this week to tonight
- My OH on Tuesday 8/6 (tomorrow) are cancelled

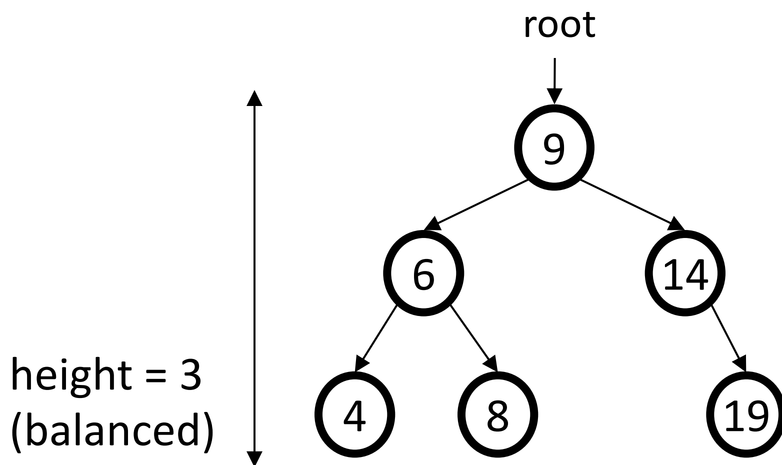
Overflow

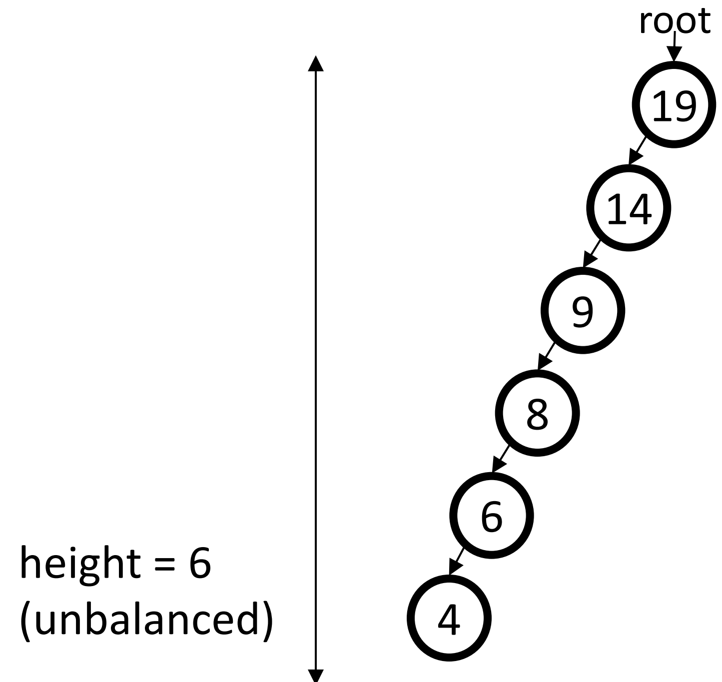
- We saw how to add to a binary search tree. Does it matter what order we add in?
  - Try adding: 9, 6, 14, 4, 8, 19
  - Now add the same numbers but in sorted order: 4, 6, 8, 9, 14, 19

Balanced Trees

Trees and balance

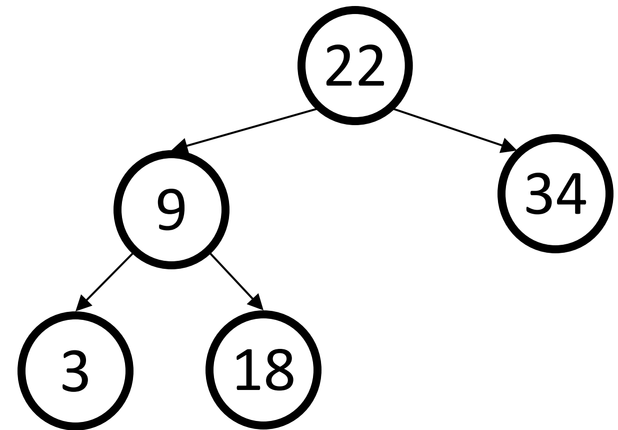
- **balanced tree**: One where for every node R, the height of R's subtrees differ by at most 1, and R's subtrees are also balanced.
  - Runtime of add / remove / contains are closely related to height.
  - Balanced tree's height is roughly $\mathbf{log_2}\ \boldsymbol{N}$. Unbalanced is closer to $\boldsymbol{N}$.





BST balance question

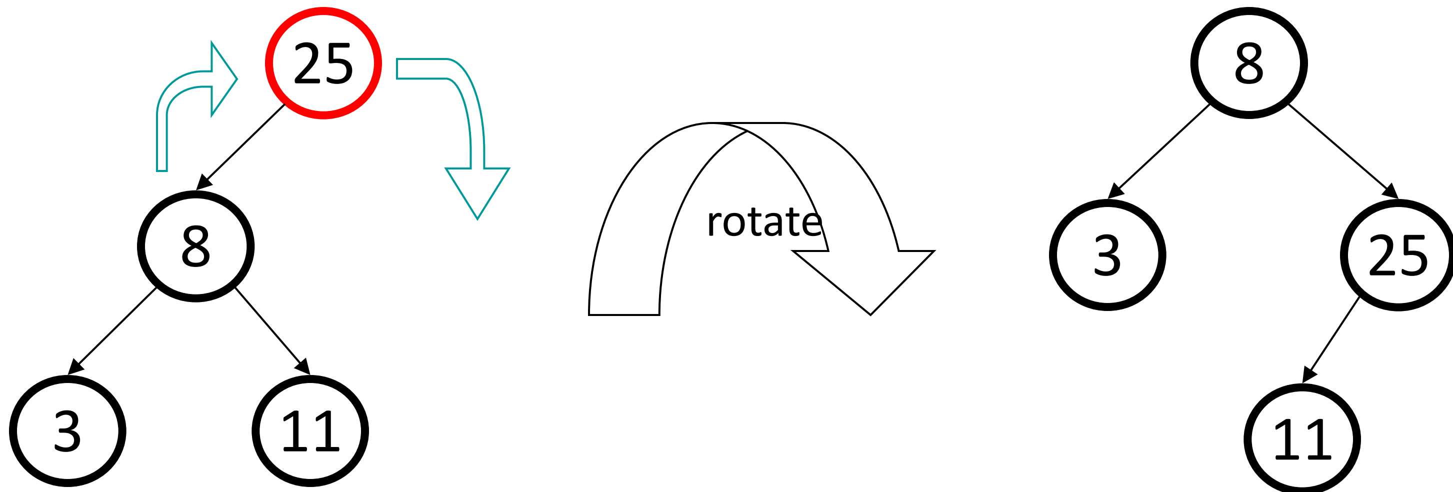
- Adding the following nodes to an empty BST in the following order produces the tree at right: 22, 9, 34, 18, 3.



- **Q:** What is an order in which we could have added the nodes to produce an unbalanced tree?
  - **A.** 18, 9, 34, 3, 22
  - **B.** 9, 18, 3, 34, 22
  - **C.** 9, 22, 3, 18, 34
  - **D.** none of the above

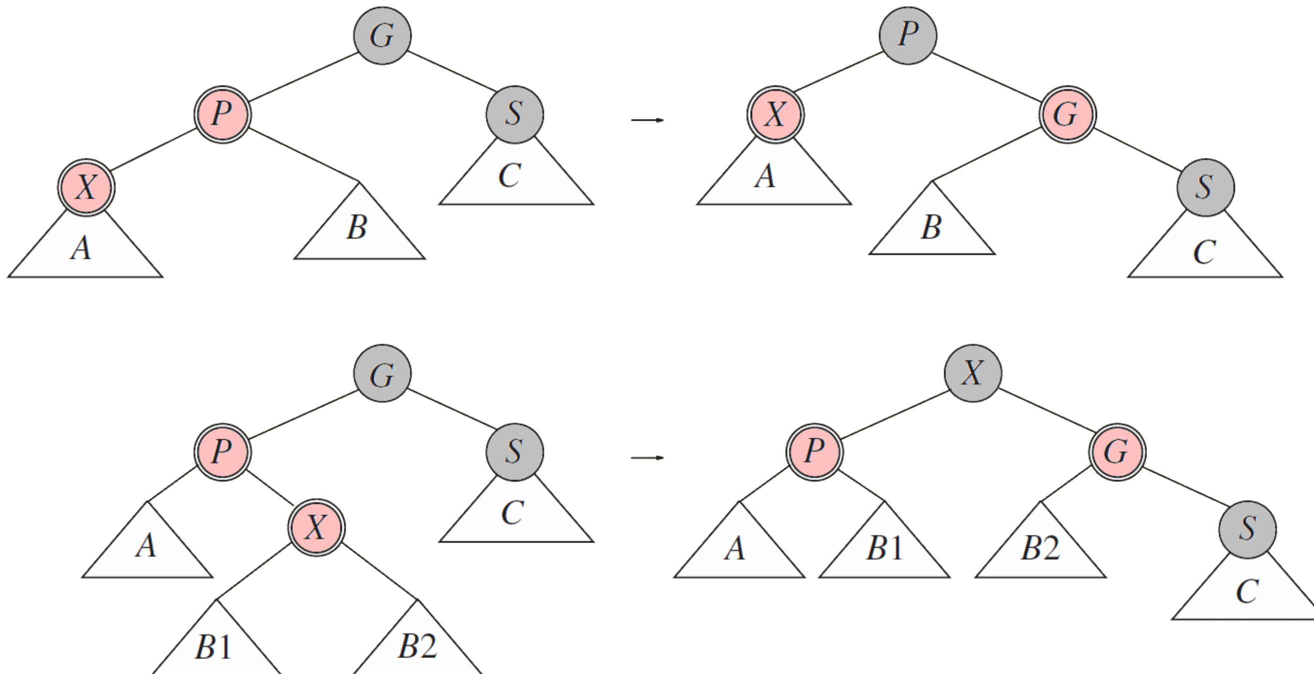
AVL trees

- **AVL tree**: A binary search tree that uses modified add and remove operations to stay balanced as its elements change.
  - *basic idea:* When nodes are added/removed, repair tree shape until balance is restored.
    - rebalancing is O(1); overall tree maintains an O(log *N*) height



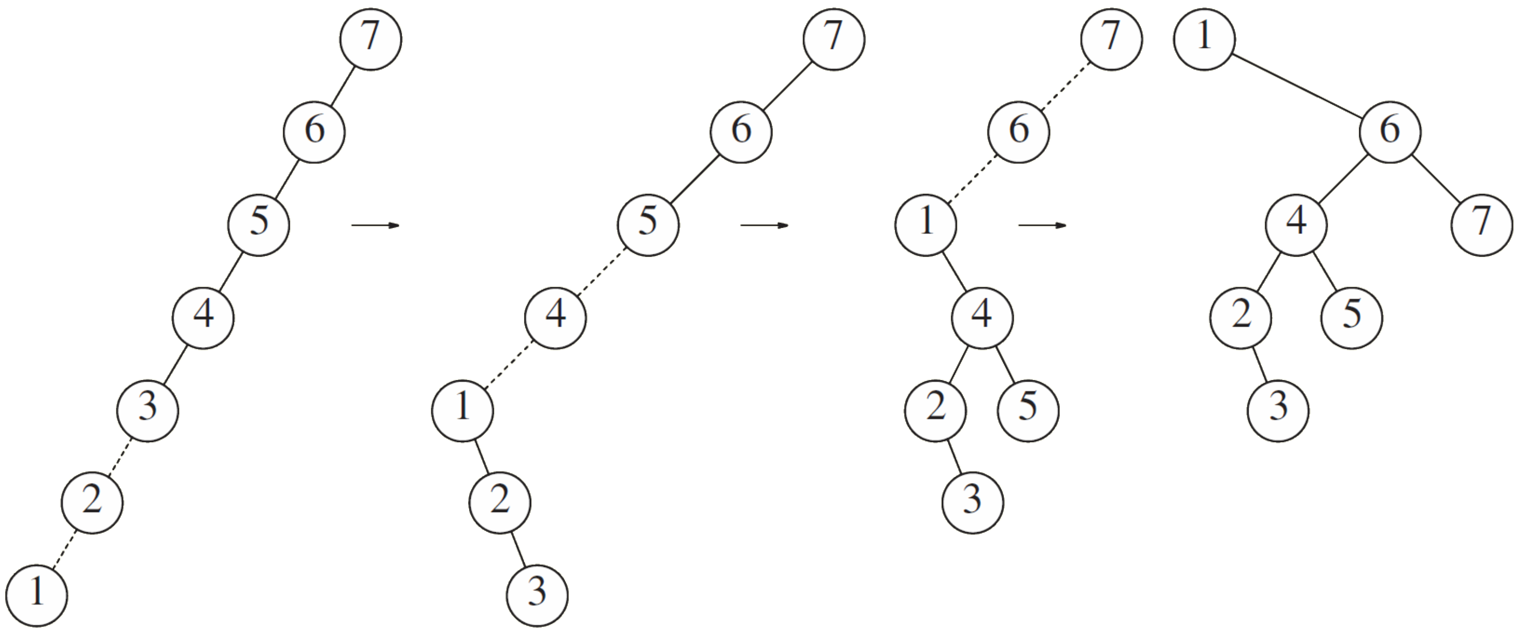
Red-Black trees

- **red-black tree:** Gives each node a "color" of red or black.
  - Root is black. Root's direct children are red. All leaves are black.
  - If a node is red, its children must all be black.
  - Every path downward from a node to the bottom must contain the same number of "black" nodes.



Splay trees

- **splay tree:** Rotates each element you access to the top/root
  - very efficient when that element is accessed again (happens a lot)
  - easy to implement and does not need height field in each node



Non-BST Binary Trees

Heaps

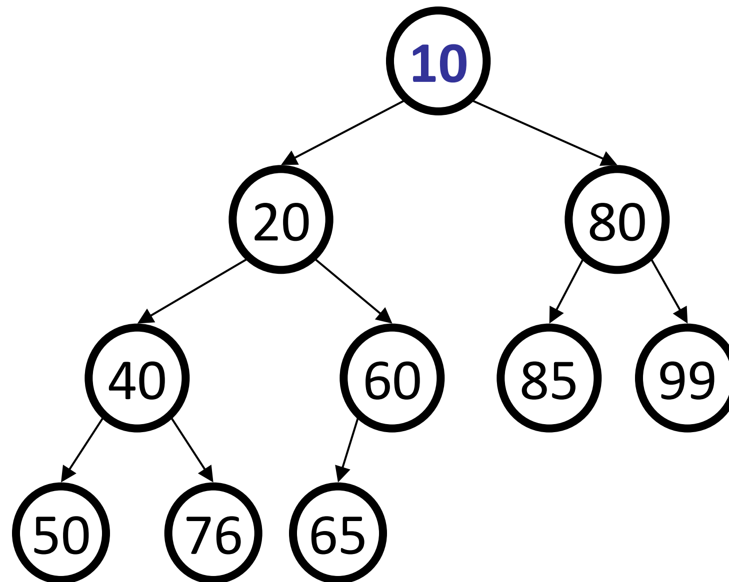
- What if you want to find the k-smallest elements in an unsorted Vector?
  - Find the top 10 students in a class?
- What if you wanted to constantly insert and remove in sorted order?
  - Model a hospital emergency room where individuals are seen in order of their urgency
  - **Priority Queue**
- What's a good choice?

Heaps

- Idea: if we use a Vector, it takes a long time to insert or remove in sorted order (or search the Vector for the smallest element)
- If we use a binary search tree, it's fast to insert and remove (O(log N)) but it's slow to find the minimum/maximum element (O(log N))
- Idea: use a tree, but store the minimum/maximum element as the root
  - Trees have log(N) insertion/deletion
  - Looking at the root is O(1)

Heaps

- **heap**: A <u>complete</u> binary tree with vertical ordering:
  - **min-heap**: all children must be ≥ parent's value
  - **max-heap**: all children must be ≤ parent's value

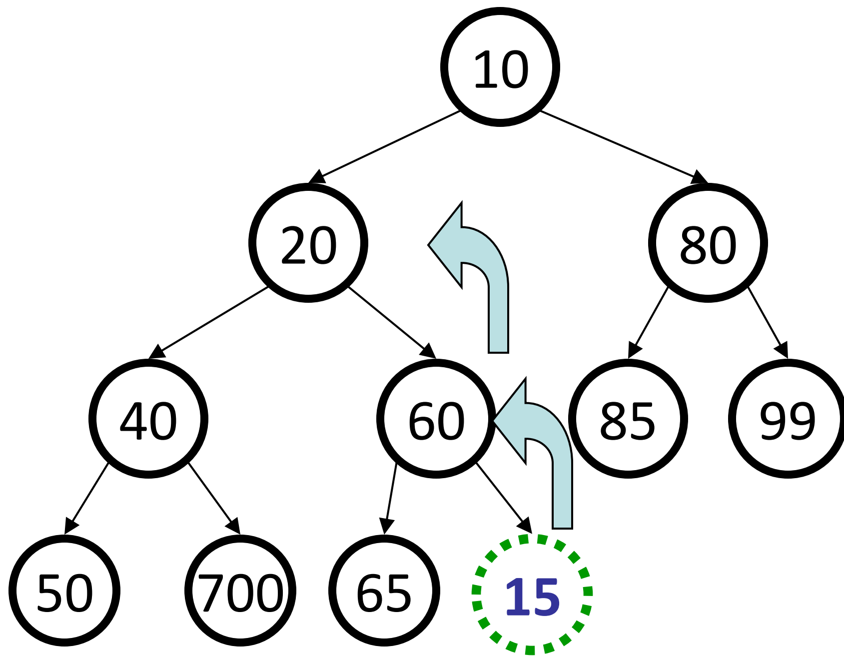


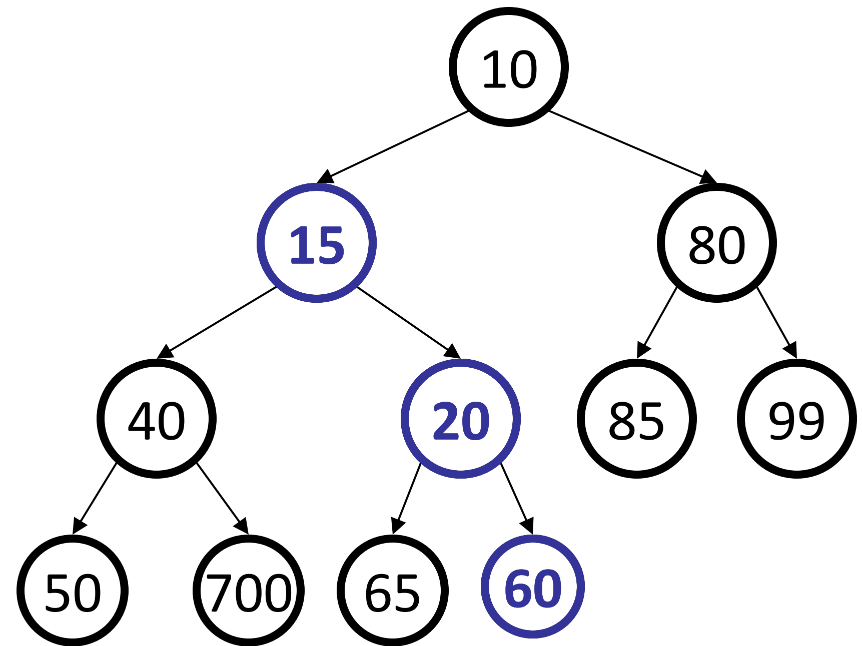
a min-heap

  - **complete tree**: all levels are full of children except perhaps the bottom level, in which all existing nodes are maximally to the left.
    - Nice corollary: heaps are *always* balanced

Heap enqueue

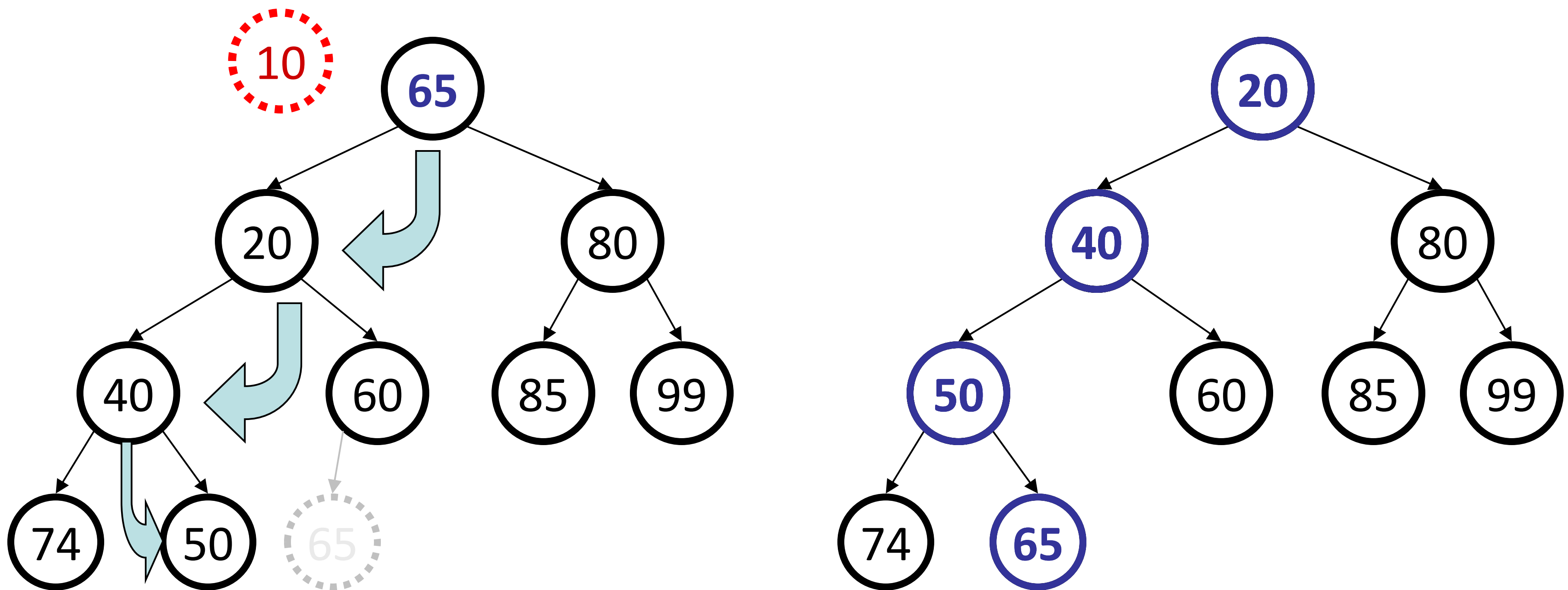
- When adding to a heap, the value is first placed at bottom-right.
  - To restore heap ordering, the newly added element is shifted ("**bubbled**") up the tree until it reaches its proper place (we reach the root, or the element is smaller than its parent [min-heap]).
  - Enqueue 15 at bottom-right; bubble up until in order.





Heap dequeue

- Remove the root, and replace it with the furthest-right ancestor
- To restore heap order, the improper root is shifted ("bubbled") down the tree by swapping with its smaller child.
  - dequeue min of 10; swap up bottom-right leaf of 65; bubble down.



Cartesian Trees

- How would you quickly find the minimum/maximum element in an range?
  - Maximum elevation on a hike?
  - Best time to buy/sell a stock within a certain range of times?

Cartesian Trees

- The root stores the minimum (or maximum) element in the entire array
- The left subtree is then the minimum (or maximum) element in the range to the left of the root; the right subtree is the minimum (or maximum) element in the range to the right of the root
  - Follows the min- (or max)-heap property: every parent is smaller (or bigger) than its child

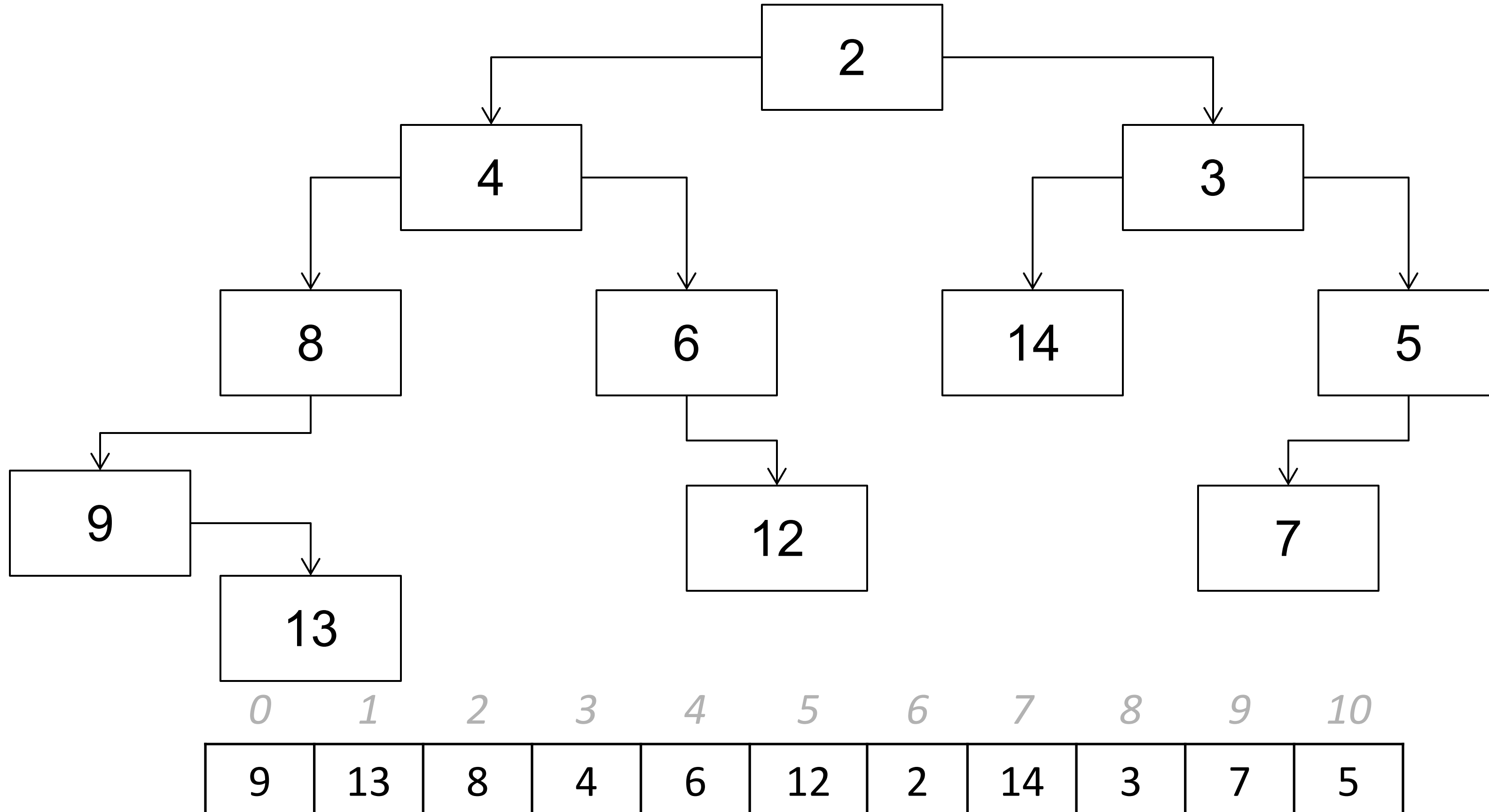
Cartesian Trees

- What would the Cartesian tree look like for this array if we're trying to find the minimum value in a range?

0	1	2	3	4	5	6	7	8	9	10
9	13	8	4	6	12	2	14	3	7	5

Cartesian Trees

- What would the Cartesian tree look like for this array if we're trying to find the minimum value in a range?



Cartesian Trees

- How would we write the following function:

```
findMinElemInRange(CartesianNode *node, int start, int end)

struct CartesianNode {
    int index;
    CartesianNode *left;
    CartesianNode *right;
};
```

Tries

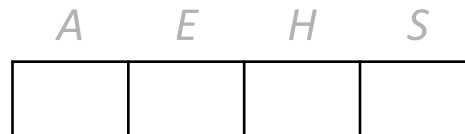
The Lexicon

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

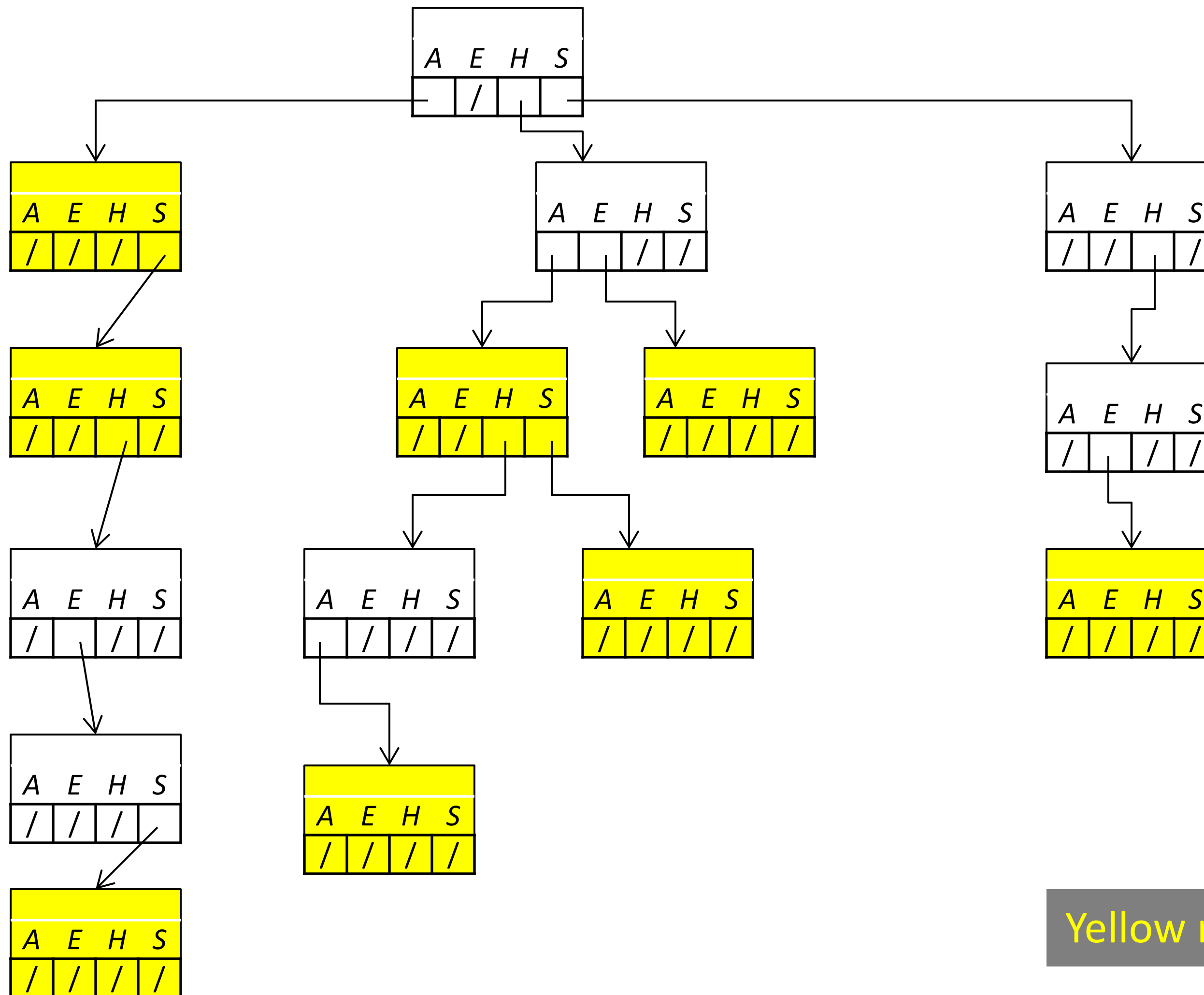
Trie

- **trie** ("try"): A tree structure optimized for "prefix" searches
  - 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 our example: {A, E, H, S}

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

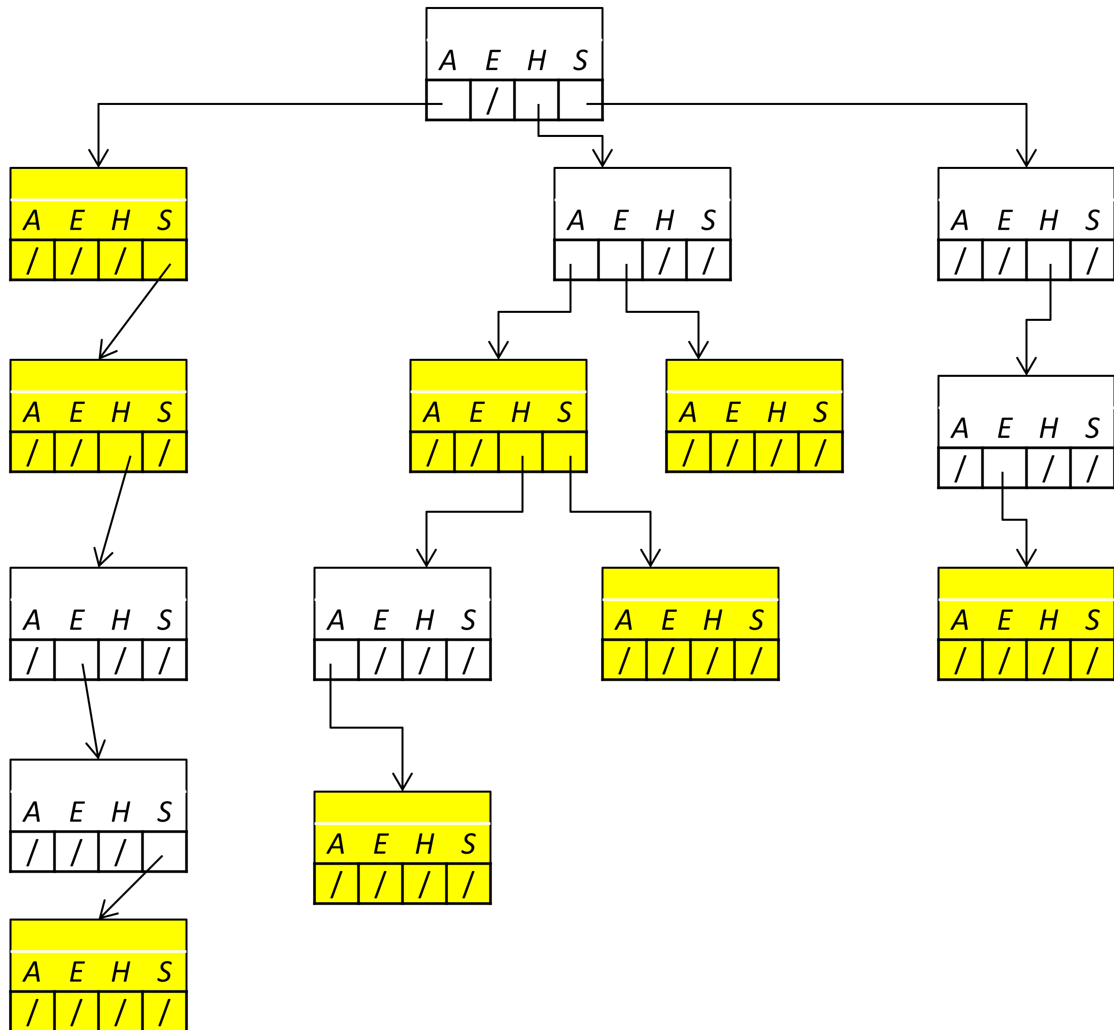


Let's "Trie" an Example



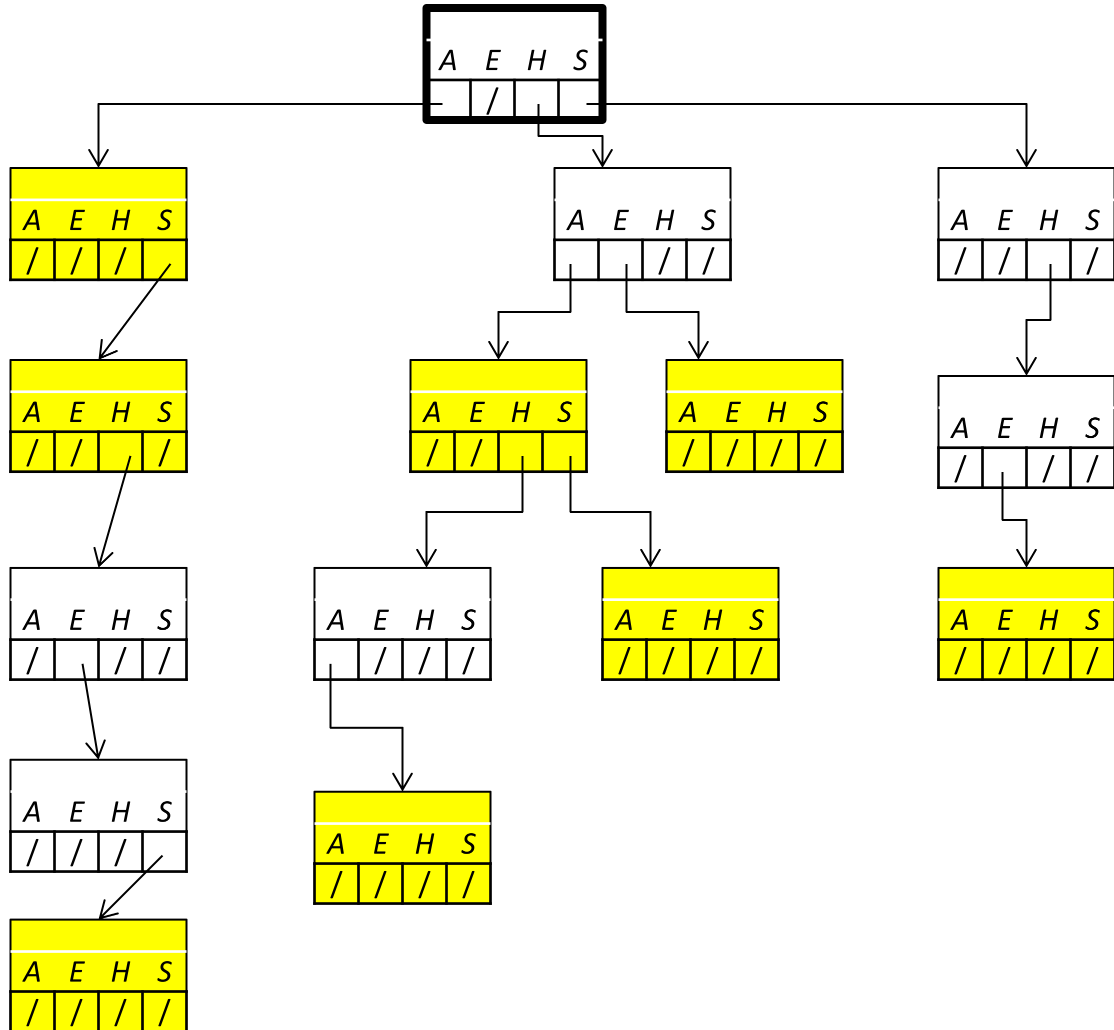
Yellow nodes are words!

Reading Words



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

Reading Words



- Example:

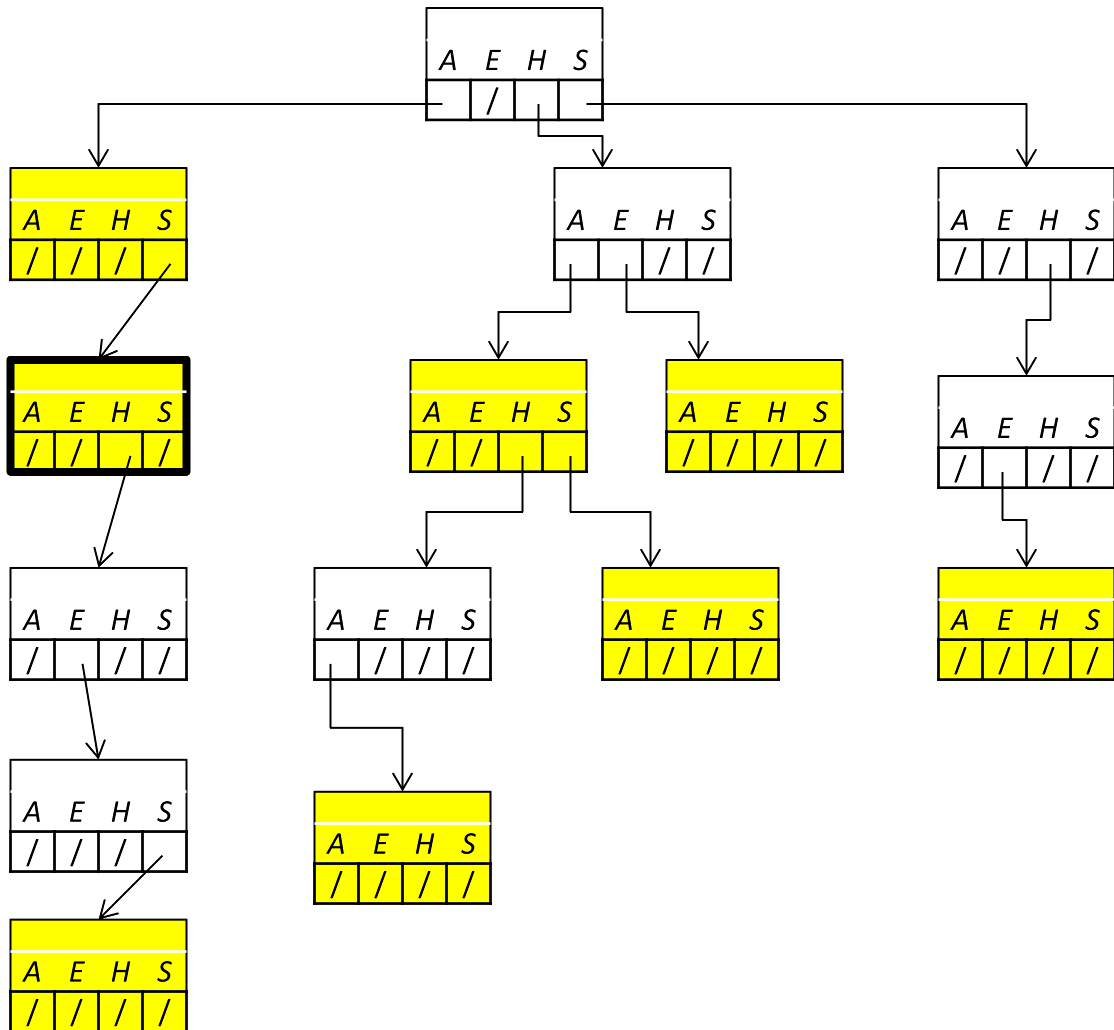
""

Reading Words

- Example:

"a"

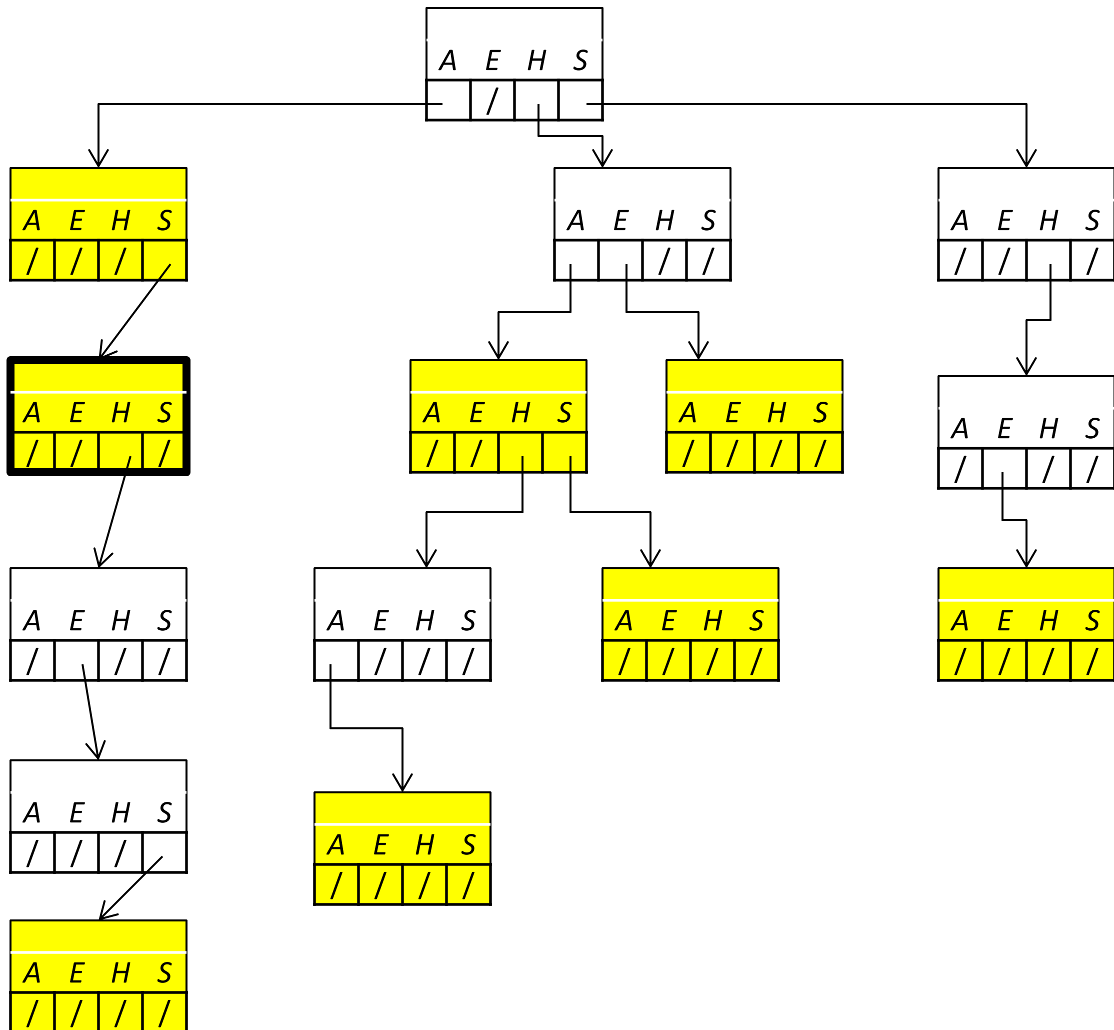
Reading Words



- Example:

"as"

Reading Words

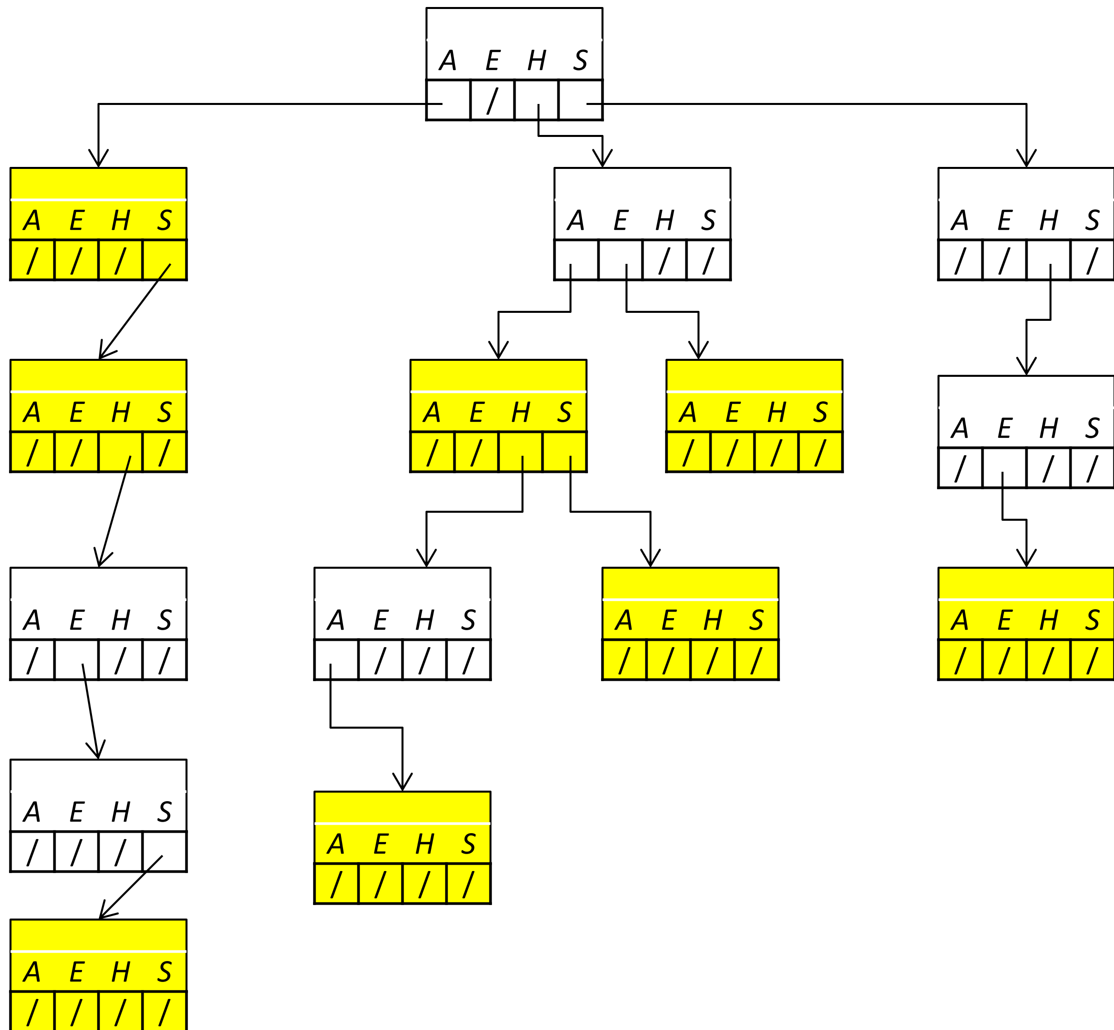


- Example:

"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

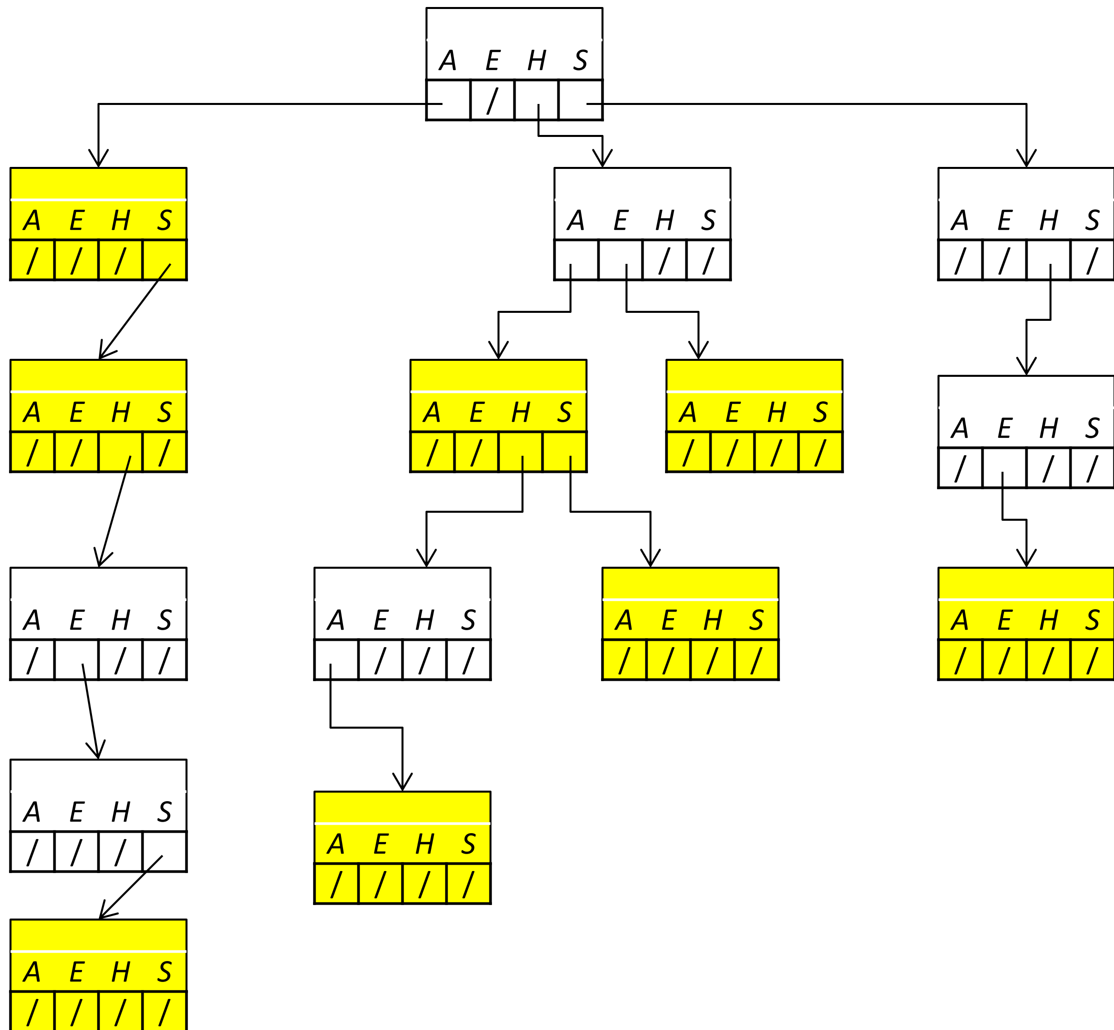
- How could we write a function that prints all words in a trie?

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));
}
```