CS 240: Programming in C

Lecture 17: Trees

PURDUE UNIVERSITY

Announcements

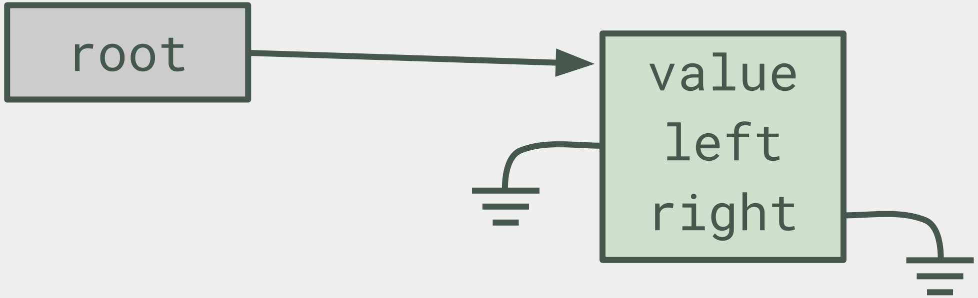
- Homework 8 extended to Friday 11/1
  - 10 extra credit points if submitted by original deadline (10/30)
  - Extra credit is applied to homeworks, not exams
- Midterm review on Monday
  - Come prepared with questions!
  - We'll cover some of the practice questions as well

Trees

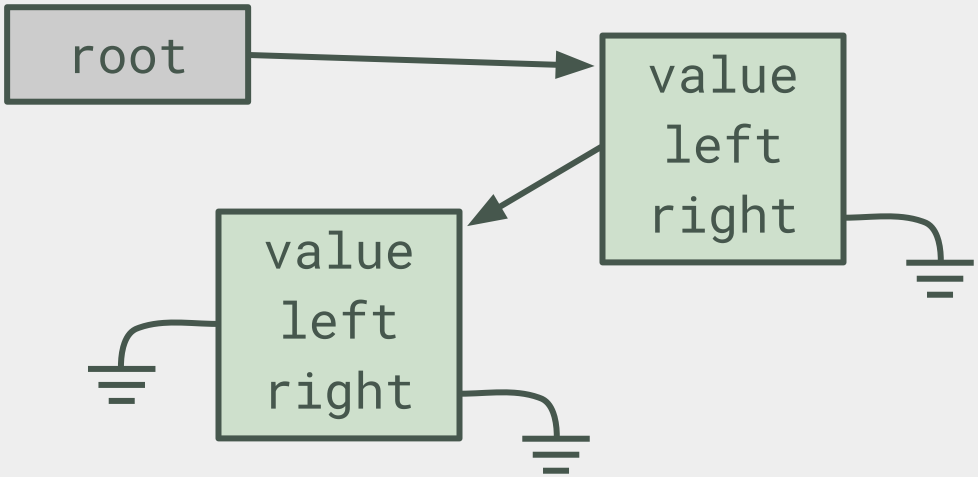
- Until now, we’ve only looked at lists that have one “dimension”
  - Forward/backward or next/previous
- Consider a structure that acts as a “parent” and has at most two “children” - a **binary** tree

```
struct node {
  int value;
  struct node *left;
  struct node *right;
};
```

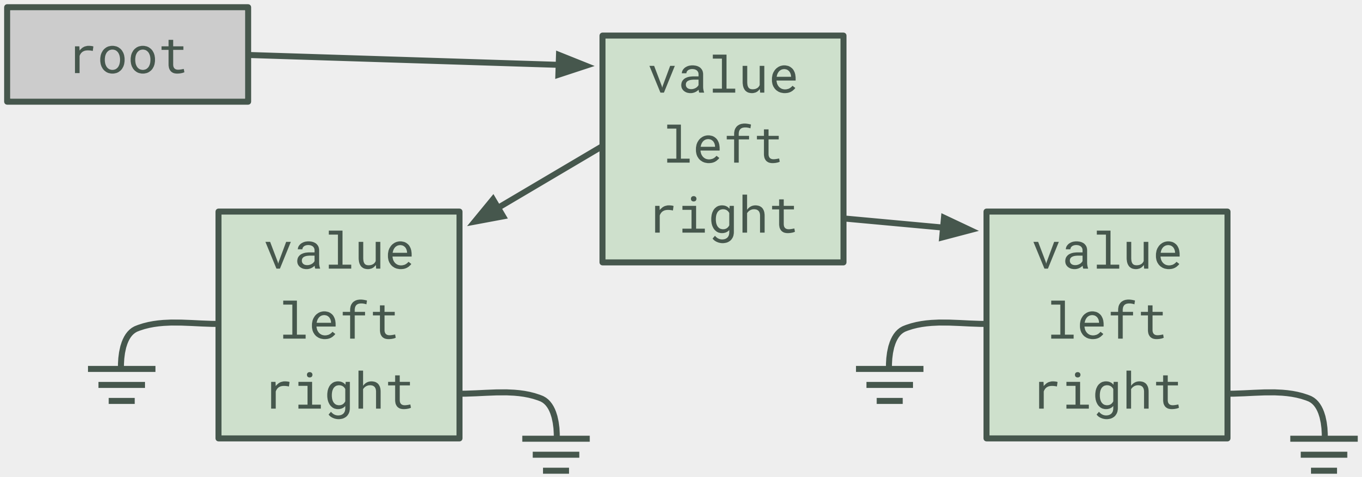
Single node



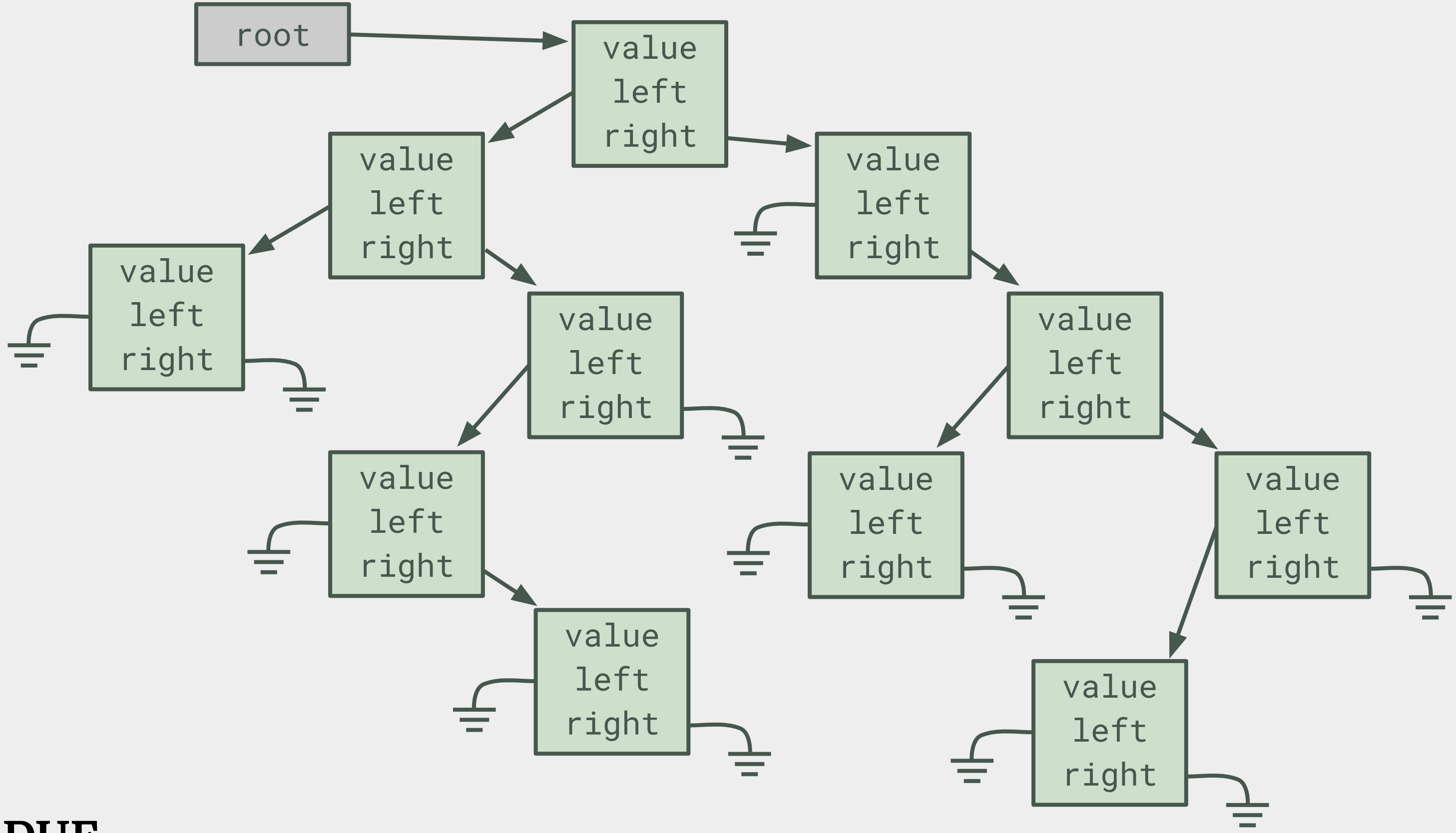
Parent & left child



Parent & two children



Many children

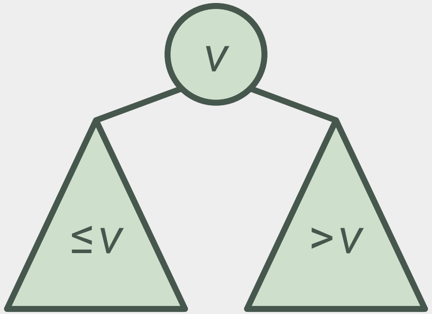


Interesting properties of trees

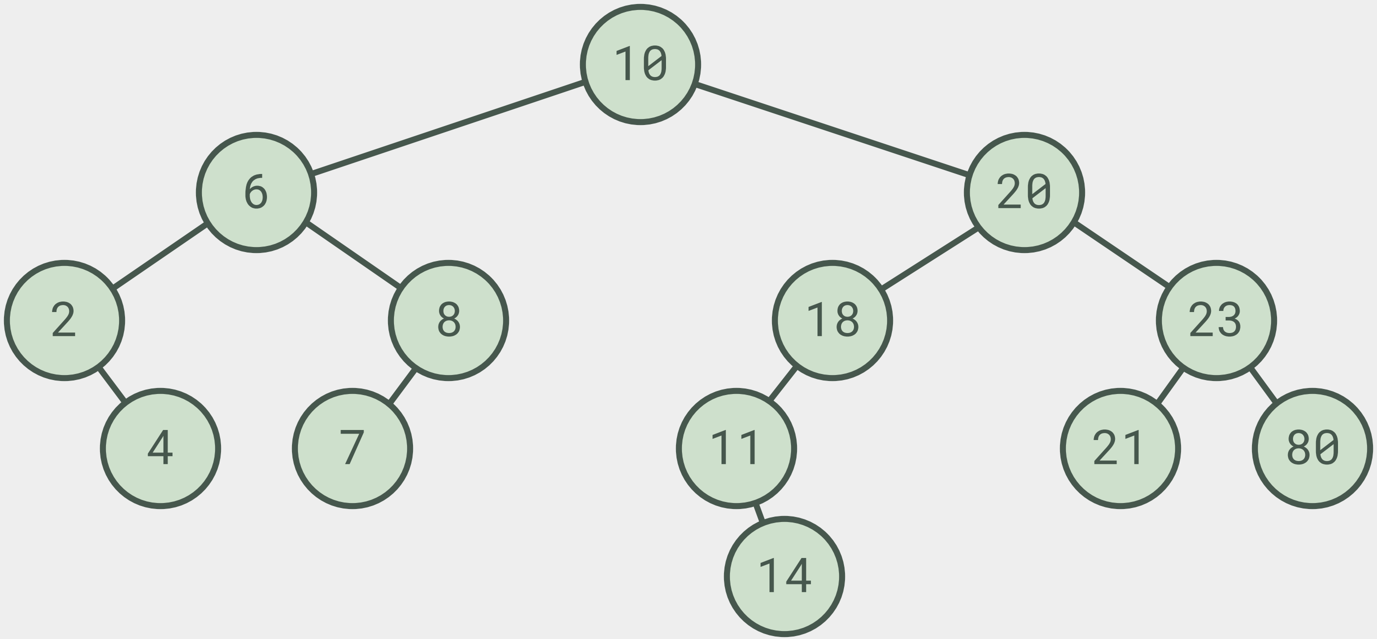
- We hold one pointer to the "root" of the tree
- Nodes without any children are called "leaf" nodes
- Nodes with one or two children are called "internal" nodes
- The "height" of a tree is the number of nodes on the *longest* path from the root to a leaf
- Each node stores a value
- Every internal node is also the root of a "subtree"

Binary Search Trees

- It's often convenient to enforce an order on the values in the nodes
- For **any** node with value *v*:
  - All nodes in the left subtree always have values less than or equal to *v*
  - All nodes in the right subtree always have values greater than *v*
- We call such an ordered binary tree a **B**inary **S**earch **T**ree (BST)



Binary Search Tree example



Binary Search Trees

- BST $\subset$ Binary Tree
- A general binary tree does not require an order of the nodes
- But we'll *mostly* discuss BSTs in this class
- A BST is always fully sorted
- It is easily searchable

BST functions (create)

```c
struct node *create_node(int value) {
  struct node *ptr = NULL;

  ptr = malloc(sizeof(struct node));
  assert(ptr != NULL);

  ptr->left = NULL;
  ptr->right = NULL;
  ptr->value = value;

  return ptr;
}
```

BST functions (insert, iterative)

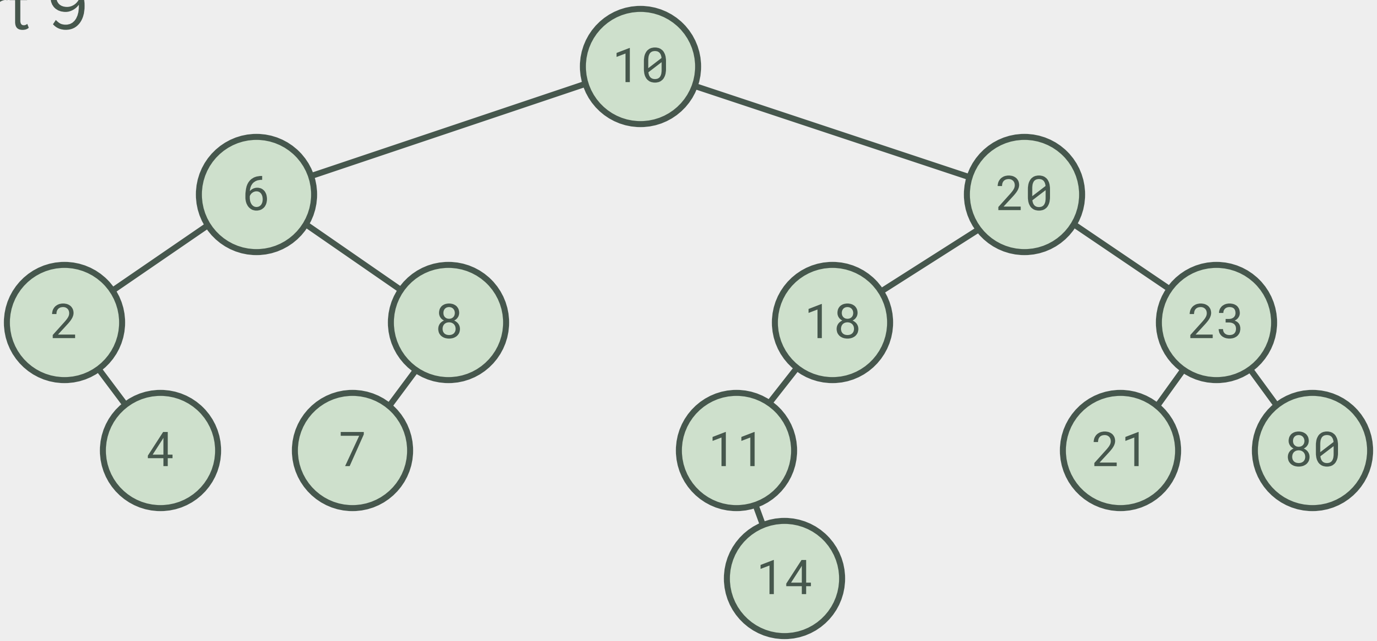
```
void insert_node(struct node *root, struct node *new) {
  while (1) {
    if (new->value <= root->value) {
      if (root->left == NULL) {
        root->left = new;
        return;
      } else {
        root = root->left;
      }
    } else {
      if (root->right == NULL) {
        root->right = new;
        return;
      } else {
        root = root->right;
} } } }
```

BST functions (insert, recursive)

```
void insert_node(struct node *root, struct node *new) {
  if (new->value <= root->value) {
    if (root->left == NULL) {
      root->left = new;
      return;
    } else {
      insert_node(root->left, new);
    }
  } else {
    if (root->right == NULL) {
      root->right = new;
      return;
    } else {
      insert_node(root->right, new);
    }
  }
}
```

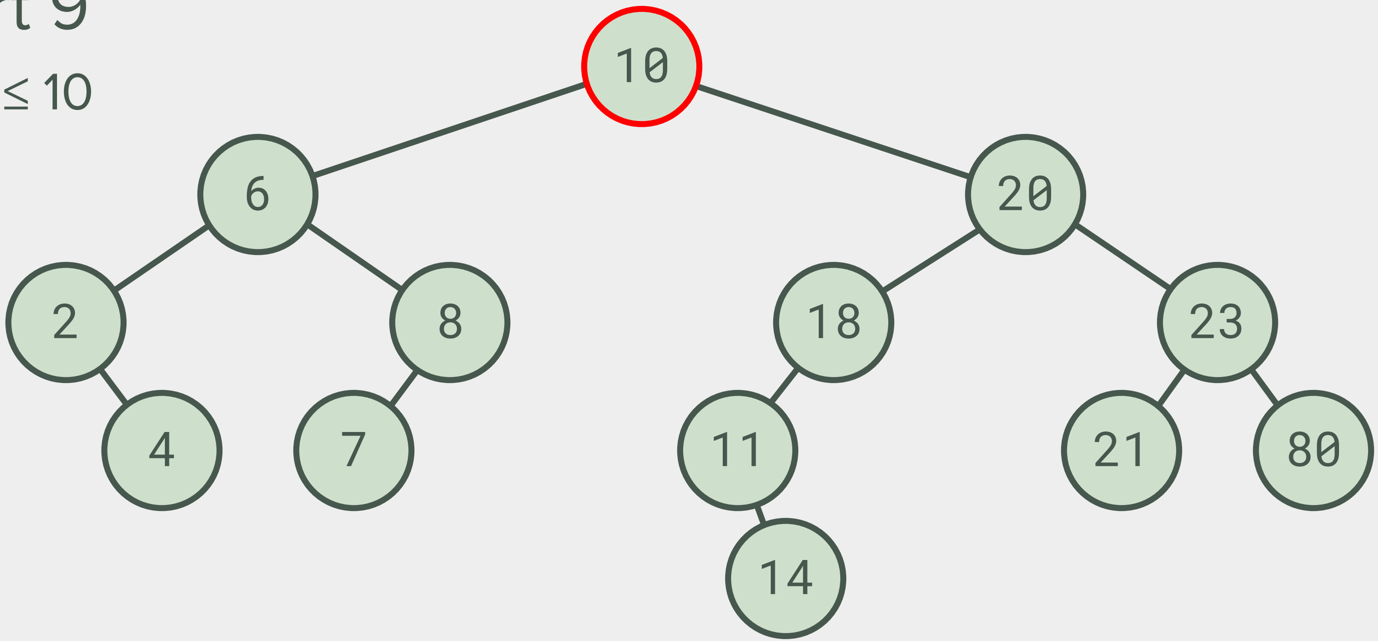
BST insert example

- Insert 9



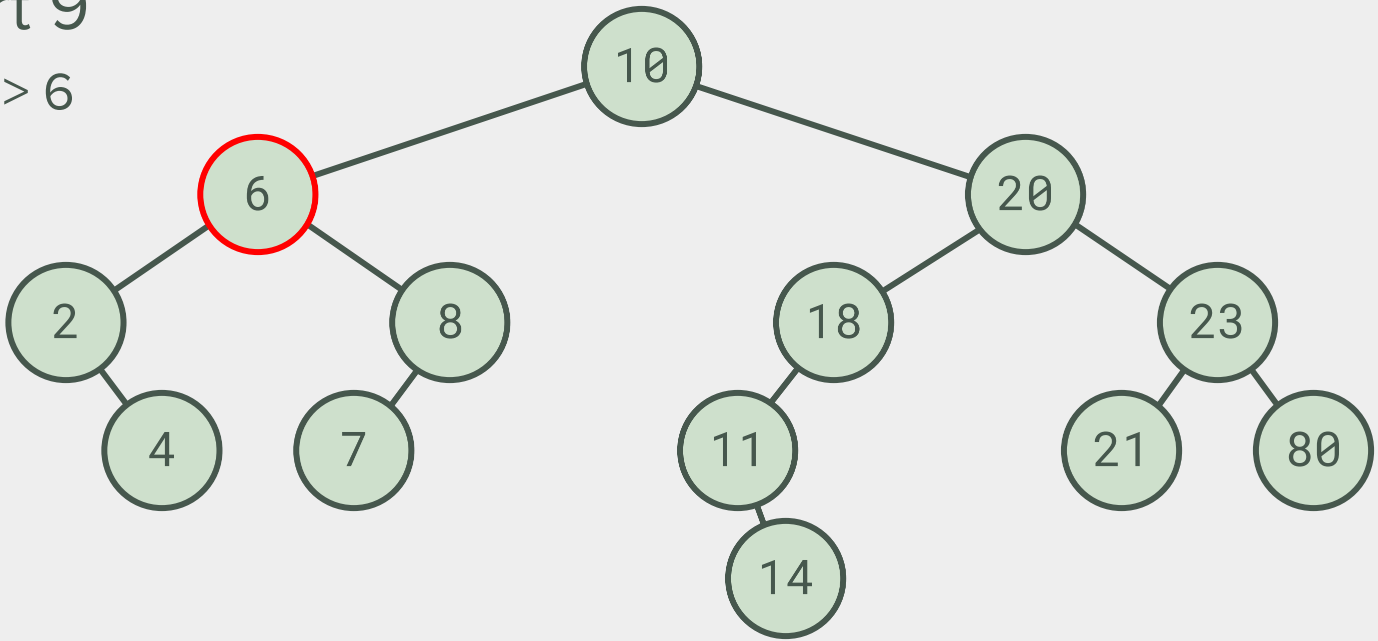
BST insert example

- Insert 9
  - 9 ≤ 10



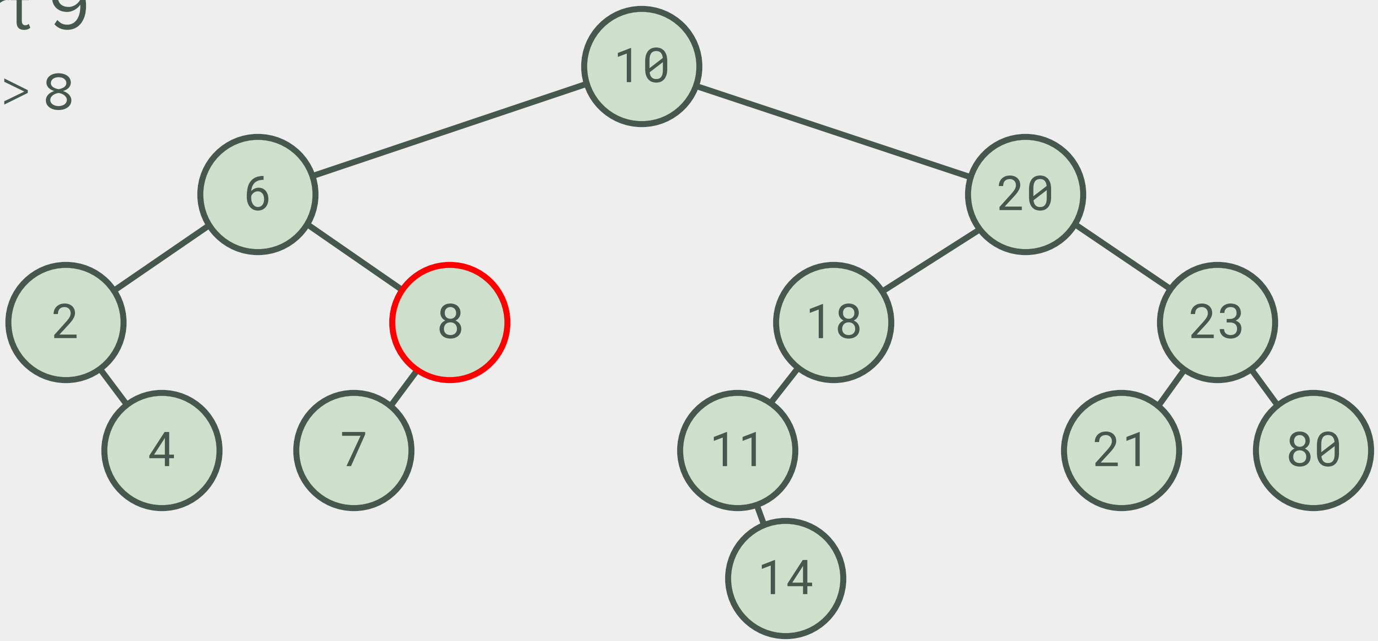
BST insert example

- Insert 9
  - 9 > 6



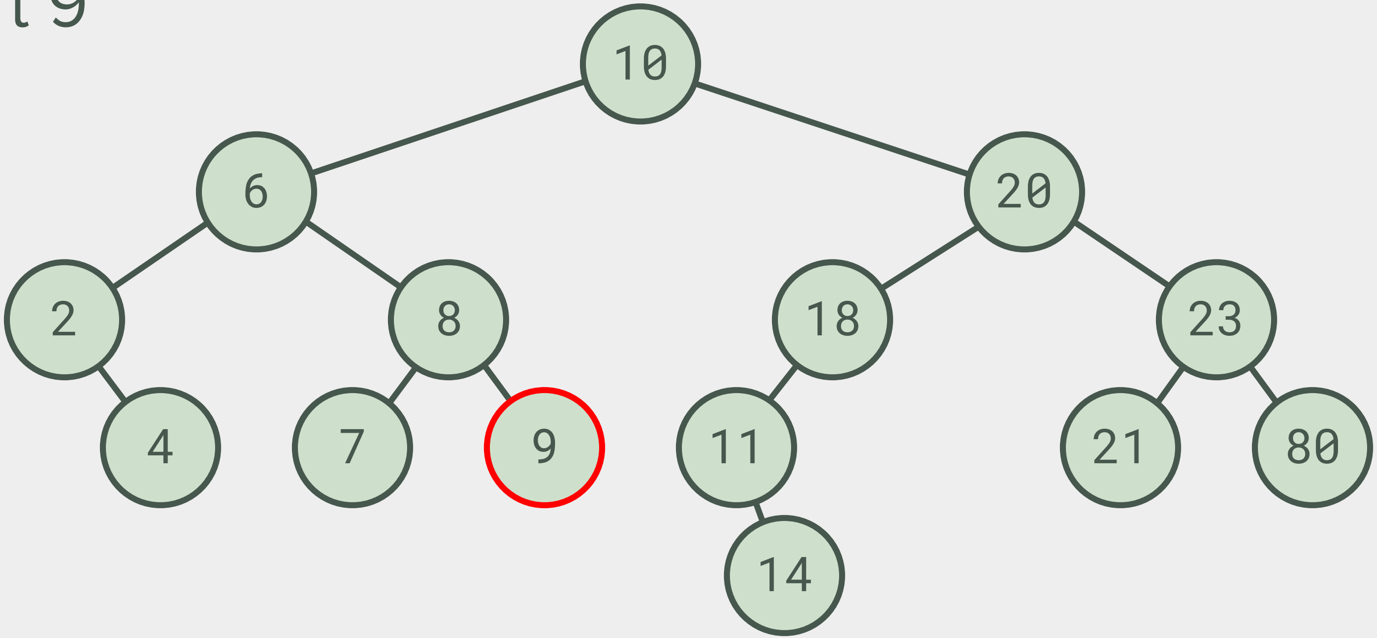
BST insert example

- Insert 9
  - 9 > 8



BST insert example

- Insert 9



Searching a BST

- Searching a BST is just as easy as inserting
  1. Set the current node pointer to point to the root
  2. If the value we're looking for == the current node's value
     - Return the current node
  3. If the search value < the current node's value, go left
     - i.e., pointer = pointer->left;
  4. Otherwise, go right
  5. Repeat from step 2
  6. If the pointer is ever NULL, return NULL to indicate the value was not found

Recursive bst_find()

```
struct node *bst_find(struct node *root, int value) {
  if (root == NULL)
    return NULL;  /* Not found */

  if (value == root->value)
    return root;  /* Found it */

  if (value < root->value)  /* Go left */
    return bst_find(root->left, value);

  /* Go right */
  return bst_find(root->right, value);
}
```

How do we access the sorted values?

- We know that a BST is fully sorted
  - The “least” element in the tree is at the far left
  - The “greatest” element in the tree is at the far right
- Our tree nodes do not point back to their parents
  - How can we start at the far left and go through each node in order?

Tree traversal

- Accessing each of the nodes of a tree is order is called a **tree traversal**. We can do this in several ways:
    - Least to greatest: for each node, access the left node recursively, then the node itself, then the right node recursively
      L-N-R
    - Greatest to least: same way, but reversed: **R-N-L**
    - Prefix: **N-L-R**
    - Postfix: **L-R-N**

Example of ordered printing

```
void print_tree(struct node *ptr) {
  if (ptr == NULL)
    return;

  print_tree(ptr->left);    /* Go left */

  printf("%d\n", ptr->value);  /* Node */

  print_tree(ptr->right);  /* Go right */
}
```

Recall recursive examples

```
void countdown(int n) {
  if (n >= 0) {
    printf("%d...\n", n);
    countdown(n-1);
  }
  return;
}
```

```
void countup(int n) {
  if (n >= 0) {
    countup(n-1);
    printf("%d...\n", n);
  }
  return;
}
```

```
5...
4...
3...
2...
1...
0...
```

```
0...
1...
2...
3...
4...
5...
```

Tree traversals

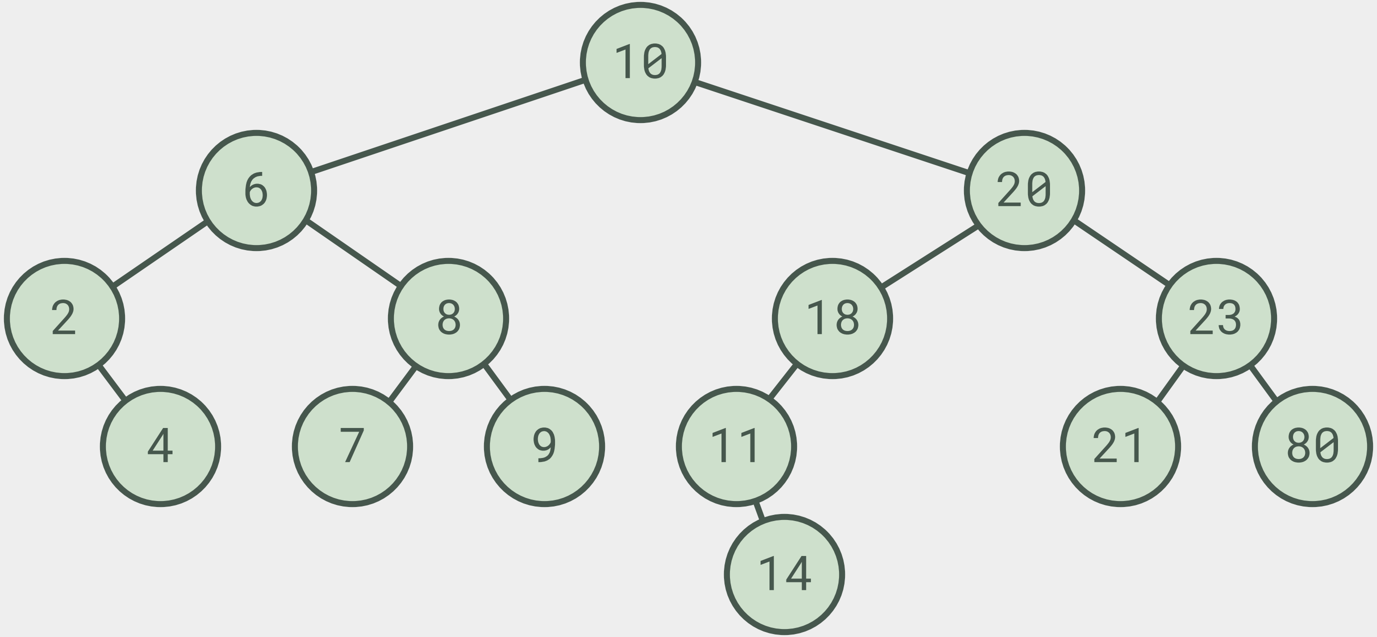
- Changing the order of the recursive calls gives us a different printout order

Example of ordered printing

```
void print_tree(struct node *ptr) {
  if (ptr == NULL)
    return;

  printf("%d\n", ptr->value);  /* Node */

  print_tree(ptr->left);    /* Go left */

  print_tree(ptr->right);  /* Go right */
}
```

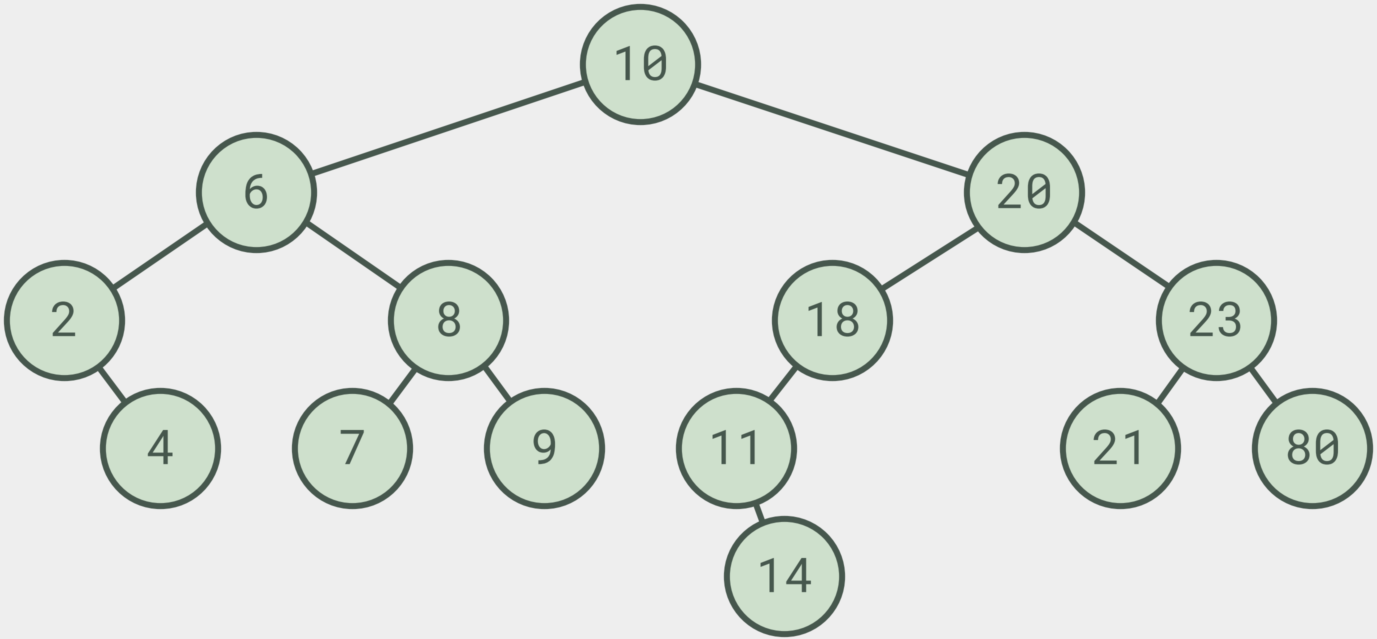
- What order will be printed?

“Prefix” traversal (N-L-R)



PURDUE UNIVERSITY

“Prefix” traversal (N-L-R)



10, 6, 2, 4, 8, 7, 9, 20, 18, 11, 14, 23, 21, 80

Takehome Quiz #4

- Write a function to perform reverse order traversal
  - WITHOUT recursion (i.e., iteratively)
- Hint: you will need to keep track of previously visited nodes in an array (i.e., a stack)
  - Assume the longest branch can have at most `MAX_HEIGHT` nodes

```
struct node *stack[MAX_HEIGHT] = { NULL };
int cur_height = -1;  /* index into stack */
```

  - See next slide for a primer on stacks

- Handwritten responses only, one page double-sided
  - No need to use the template

Simple stack *(for the quiz, not on the midterm)*

- A stack is just an array with an index to the “top” of the stack

```
int stack[CAPACITY] = { 0 };
int top = -1;  /* index to top; -1 means empty stack */
```

- To “push” an `item` onto the stack:

```
stack[++top] = item;
```

- To “pop” an `item` from the stack:

```
item = stack[top--];
```

- Check if stack is full or empty before pushing or popping
- Keep it simple
  - No need for a `struct stack` or `push()` or `pop()` functions

For next lecture

- Study for the Midterm!
  - Come with questions for the review
- Work on Homework 7 & 8!
- Study the examples in this lecture at home
- Practice the examples
- Modify the examples

Slides

- Slides are heavily based on Prof. Turkstra’s material from previous semesters.