Trees

William Hendrix

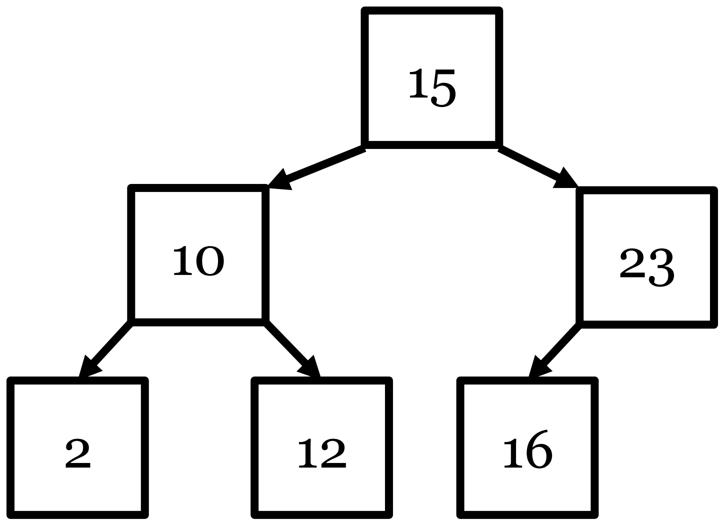
Lecture 28

Today

- Trees
- Terminology
- Traversals
- Binary search trees
- Minilab

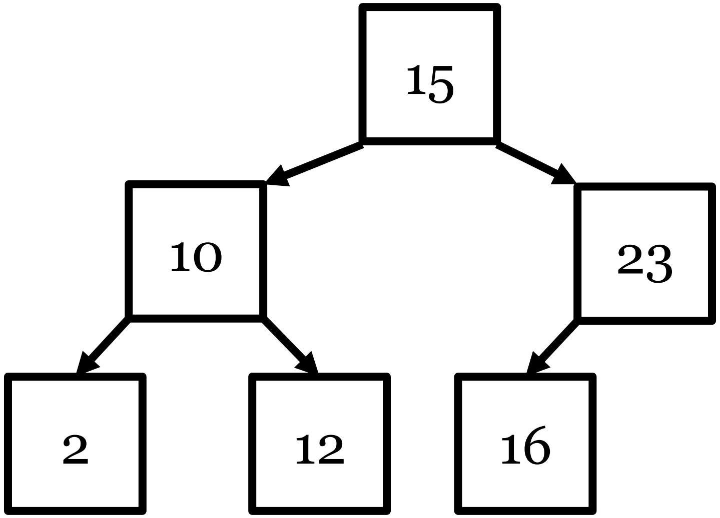
Trees

- Networked structure (nonlinear)
- Technically an abstract data structure, but nearly always implemented as link-based
- Starts at *root* node
- *Binary tree*: nodes have left and right *children*
  - *m*-ary tree: nodes have up to *m* children
- Nodes with no children are called *leaves*
- **Example**



Tree terminology

- *Level*: the set of nodes that need the same number of links to reach
- *Height*: the "lowest" level in the tree (root is level 0)
- *Siblings*: nodes that share the same parent
- *Descendant*: node that can be reached by following child links
- *Subtree rooted at a node*: the tree of that node and its descendants
- *Ancestor*: node that can be reached by following parent links
- *Least common ancestor*: lowest-level ancestor of given nodes
- *Balanced tree*: left and right subtrees have same number of nodes
- *Complete binary tree*: all nodes exist up to tree height



Tree implementation

- `Tree` class stores pointer to `root` node
  - May optionally store `size` and `height`

- `TreeNode` class stores `left` and `right` node pointers and `data`
  - Usually stores `parent` pointer
  - May store `sibling` pointer, `isLeftChild`, `level`

- Accessing nodes generally relies on some sort of *tree traversal*
  - Iterating through all nodes in a regular order

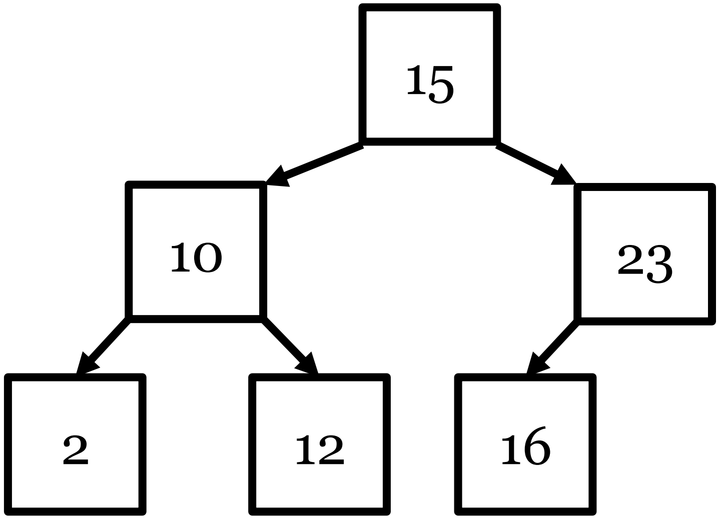
- Other functions, like `insert`, `delete`, and `find`, depend on specialized tree type

Tree traversal

- Often uses a stack or queue
- **Pseudocode**

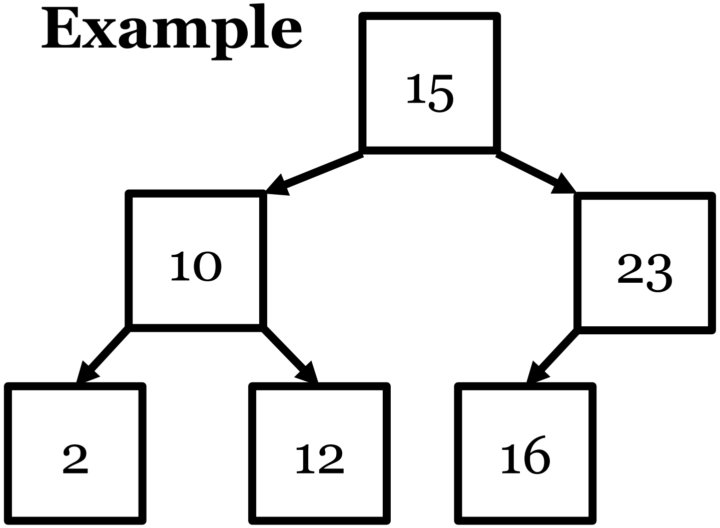
```
Push/enqueue root
While not empty
  Pop node
  Deal with node (e.g., print)
  Push/enqueue node children
```

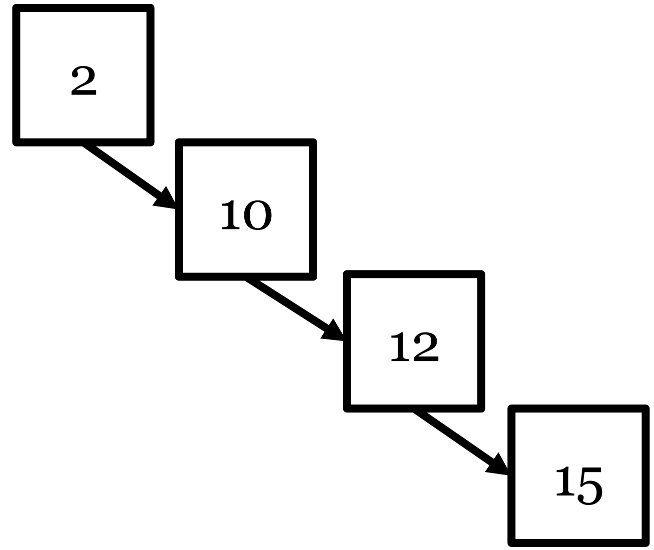
  – Can also implement recursively (like stack)
- *Stack traversa*l: depth-first search (DFS)
  – Travels down to leaf before backing up (backtracking)
- *Queue traversal*: breadth-first search (BFS)
  – Level-wise traversal
  – Travels to all nodes on a level, then moves to next level
- Since levels may contain exponentially more nodes, DFS generally uses a lot less memory



Binary search trees

- Used to organize data for faster searching
- Left child is always < parent, right child always >
- **Example**





- To search: start at root, move to right/left child if target is larger/smaller, repeat until you find target or a `NULL` child
- If tree is *balanced*, search is equivalent to binary search
- If tree is *unbalanced*, more like searching a sorted LinkedList
- Balancing a BST is generally hard
  - Specialized trees like heaps and red-black trees solve this

Binary search tree operations

- Insertion
  - Search for a node, then insert where it would go
  - Usually log time (unless balance is bad)
- Deletion
  - For a leaf: just delete and change parent's child pointer
  - If one child: point parent to child and delete node
  - If two children: replace data with left or right child and delete that child
- Comparison to sorted arrays
  - Search is comparable/slightly worse
    - Could be much worse if balance is bad
  - Insertion and deletion are usually faster
- Comparison to sorted lists
  - Nearly always faster
  - Deletion is somewhat more complicated

Tonight

- **Recommended reading:** Sections 13.2 and 13.3