Motivation

Problems with linked lists

- Linear access time of linked list is prohibitive
  - Does there exist any simple data structure for which the running time of most operations (insert/add, delete (remove), search) is $O(\lg N)$?

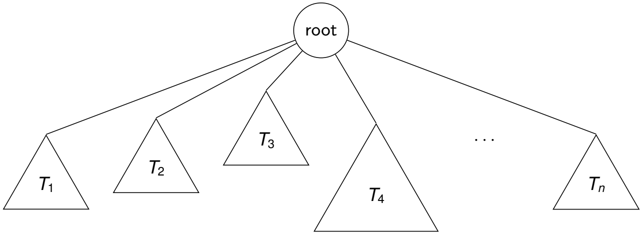
CPT108 Data Structures and Algorithms

Lecture 16

Trees

Trees

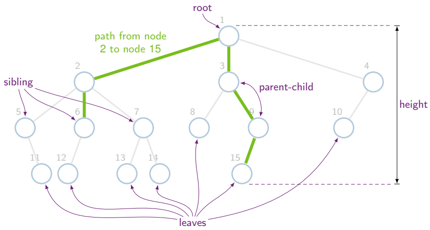
- Similar to linked list, a tree, T, is a collection of nodes
- Recursively speaking, if not empty, a tree T consists of
  - a (distinguished) node r, i.e., the root, and
  - zero or more non-empty subtrees $T_1, \ldots, T_n$ (with different sizes)



Trees (cont.)

Some terminologies

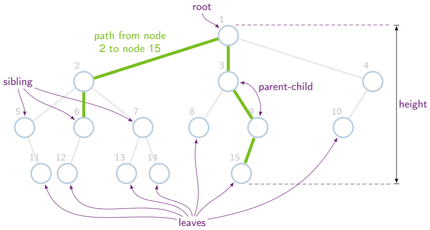
- Parent and child
  - Every node except the root has *one* parent
  - A node, including the root, can have an zero or more children
- Leaves (or leaf nodes)
  - nodes with no child
- Sibling
  - nodes with same parent



- Degree of a *node* is the number of children it has. Degree of a *tree* is the maximum of its element degree
  i.e., in the tree above, `degree(2)` $= 3$, `degree(3)` $= 2$, `degree(4)` $= 1$ and the degree of the tree is 3

Some terminologies

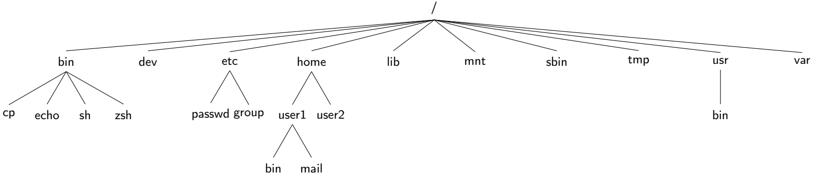
- Path
  - A sequence of edges from one node to another, e.g., 6 – 2 – 1 – 3 – 9 – 15
- Depth of a node
  - number of edges from the root to that node,
    e.g., `depth(10)` $= 2$
- Height of a tree
  - the longest path from the root to a leaf node,
    e.g., `height(tree)` $= 3$



- Ancestor and descendant
  - An ancestor of a node is any node on the path from the node to the root,
    e.g., `ancestor(9)` $= \{1, 3\}$
  - A descendant is the inverse relation of ancestor, i.e., a node *p* is a descendant of a node *q* if and only if *q* is an ancestor of *p*,
    e.g., `descendant(3)` $= \{8, 9, 15\}$,
    `descendant(4)` $= \{10\}$

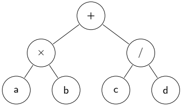
Trees (cont.)

Example: (Simplified) Unix Directory Structure

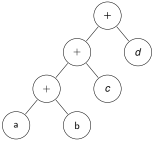


Trees (cont.)

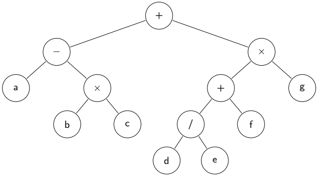
Example: Expression trees



(a) $a \times b + c/d$



(b) $a + b + c + d$

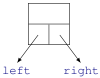


(c) $(a - (b \times c) + ((d/e) + f) \times g$

- Leaf nodes are operands, i.e., constants or variables
- Internal nodes are operators
- Will not be a binary tree if some operators are not binary, such as max, min, etc.

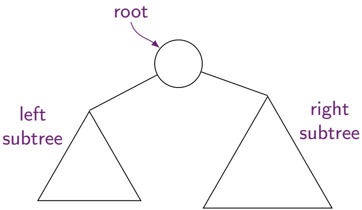
Binary tree

- In a *Binary Tree*
  - The tree degree is two
  - Each node has a maximum of two children (links)
    - ★ One to the left child of the node;
    - ★ One to the right child of the node
    - ★ If *no* child node exists for a node, the link is set to `null`



- A binary tree is either *empty*

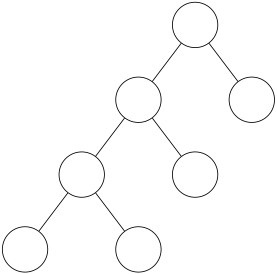
  or
- Consists of a node called *root*
  - Root points to two *disjoint* binary subtrees: the *left* and *right* subtrees



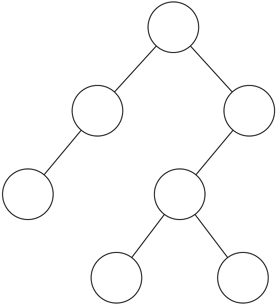
Binary tree (cont.)

Full Binary Tree

- Every internal node has exactly two children



Full binary tree

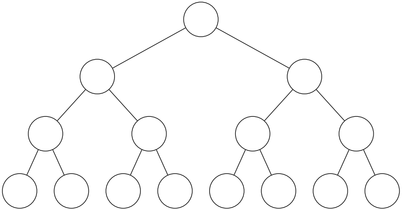


NOT a full binary tree

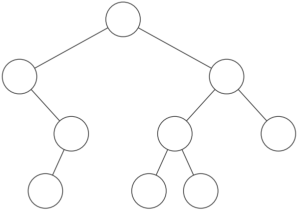
Binary tree (cont.)

Perfect Binary Tree

- Every level is full



Perfect binary tree

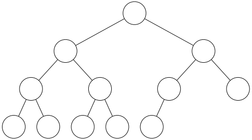


NOT a perfect binary tree

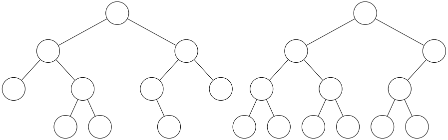
Binary tree (cont.)

Complete Binary Tree

- Every level is full except possibly the bottommost level
- If the bottommost level is not full, then the nodes must be packed to the left



Complete binary tree



NOT a complete binary tree

Binary tree (cont.)

Height of a Binary Tree

- The number of edges on the longest path from the root to a leaf
- A binary tree of heigh k has
  - At least $k + 1$ elements (linear chain)
  - At most $2^{k+1} - 1$ elements (perfect tree)

Tree traversal

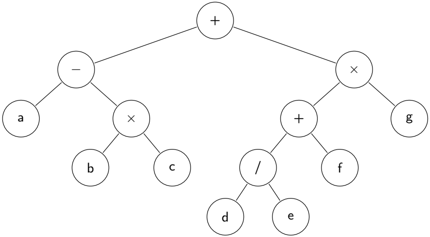
- Used to print out or search the data in a tree in a certain order
- Three types:
  - Preorder traversal
  - Postorder traversal
  - Inorder traversal

Tree traversal (cont.)

Preorder traversal

- Prints the subtree recursively in the order of:
  1. root of the subtree,
  2. values in the left subtree, and
  3. values in the right subtree

```
PREORDER-TREE-WALK(x)
   // x: root of a subtree
if x ≠ NIL
    print x.key
    PREORDER-TREE-WALK(x.left)
    PREORDER-TREE-WALK(x.right)
```



Prefix expression:

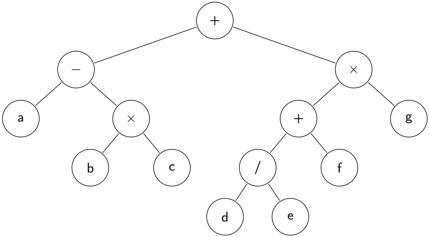
+ − a × b c × + / d e f g

Tree traversal (cont.)

Postorder traversal

- Prints the subtree recursively in the order of:
  1. values in the left subtree,
  2. values in the right subtree, and
  3. root of the subtree

```
POSTORDER-TREE-WALK(x)
    // x: root of a subtree
if x ≠ NIL
        POSTORDER-TREE-WALK(x.left)
        POSTORDER-TREE-WALK(x.right)
        print x.key
```



Postfix expression:

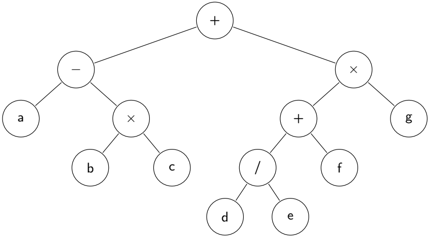
a b c × − d e / f + g × +

Tree traversal (cont.)

Inorder traversal

- Prints the subtree recursively in the order of:
  1. values in the left subtree,
  2. root of the subtree, and
  3. values in the right subtree

```
INORDER-TREE-WALK(x)
    // x: root of a subtree
if x ≠ NIL
    INORDER-TREE-WALK(x.left)
    print x.key
    INORDER-TREE-WALK(x.right)
```



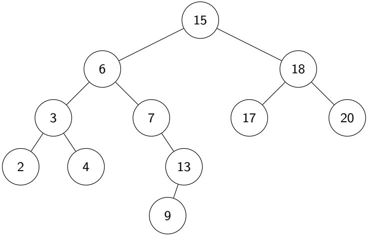
Infix expression:

$a - b \times c + d / e + f \times g$

Tree traversal (cont.)

Exercise

Write down the preorder, postorder, and inorder traversals of the following (search) tree.



- Preorder (root, left, right): 15, 6, 3, 2, 4 , 7, 13, 9, 18, 17, 20
- Postorder (left, right, root): 2, 4, 3, 9, 13, 7, 6, 17, 20, 18, 15
- Inorder (left, root, right): 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

Binary Search Tree (BST)

Disadvantage of binary search

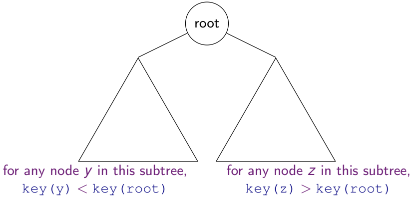
- Elements need to be sorted first
- Requires a sequential storage
- Not appropriate for linked lists (Why?)

Question

Is it possible to use a linked structure which can be searched in a binary-like manner?

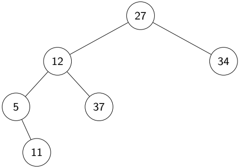
What is Binary Search Tree (BST)?

- A collection of elements in a binary tree structure
- Stores keys in the nodes of the binary tree in a way so that searching, insertion and deletion can be done efficiently
- All keys are unique! I.e., *no* two elements have the same key
- The keys (if any) in the left subtree of the root are smaller than the key in the root
- The keys (if any) in the right subtree of the root are larger than the key in the root

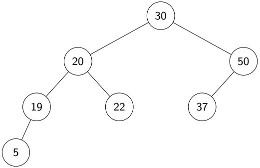


- The left and right subtrees of the root are also binary search trees (BSTs)

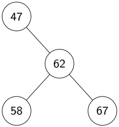
- For each node x,
  values in *left* subtree $\leq$ value in $x <$ values in *right* subtree



NOT a BST

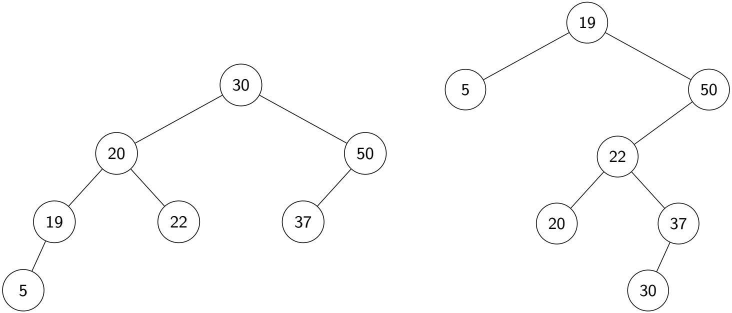


BST



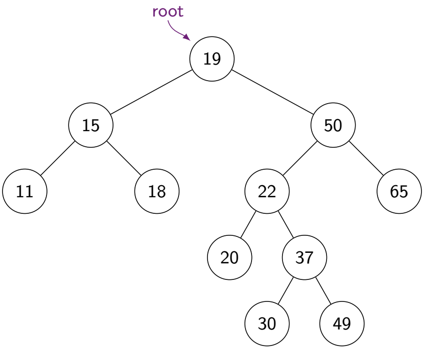
BST

- Two BSTs representing the same set of elements



Tree Search

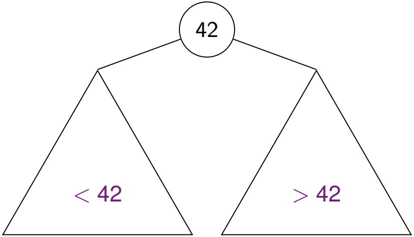
Inorder traversal of BST prints out all the keys in sorted order.



Inorder expression: 11, 15, 18, 19, 20, 22, 30, 37, 49, 50, 65

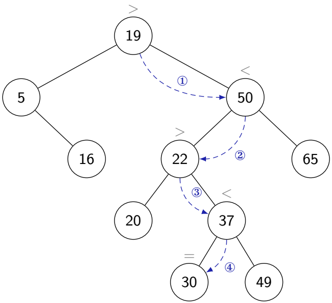
Tree Search (cont.)

To search for an element in a BST



- If we are searching for 42, then we are done!
- If we are searching for a key < 42, then we should search for it in the *left* subtree
- If we are searching for a key > 42, then we should search for it in the *right* subtree

Tree Search (cont.)
Example

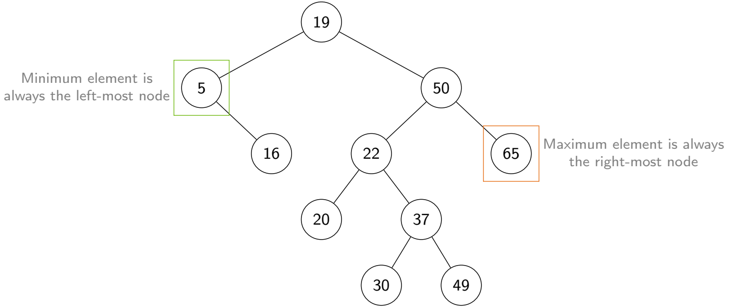


Search for 30:

1. Compare 30:19 (the root), go to right subtree;
2. Compare 30:50, go to left subtree;
3. Compare 30:22, go to right subtree;
4. Compare 30:37, go to left subtree;
5. Compare 30:30 found it!

$\Rightarrow$ *Time complexity: O(Height of the tree)*

Find Min and Max



FIND-MIN(x)

// x: root of a subtree
while LEFT(x) $\neq$ NIL
  $x = x.left$
return x

FIND-MAX(x)

// x: root of a subtree
while RIGHT(x) $\neq$ NIL
  $x = x.right$
return x

Reading

- Chapter 12.1-12.2, Cormen (2022)

References