Trees, Binary Search Tree

Bryn Mawr College
CS246 Programming Paradigm

Tree

• A tree consists of a set of nodes and a set of edges that connect pairs of nodes.
• Property: there is exactly one path (no more, no less) between any two nodes of the tree.
• A path is a connected sequence of zero or more edges.
• In a rooted tree, one distinguished node is called the root. Every node c, except the root, has exactly one parent node c's parents, which is the first node traversed on the path from c to the root. c is p's child.
• The root has no parent.
• A node can have any number of children.

Rooted Tree Terminology

• A leaf is a node with no children.
• Siblings are nodes with the same parent.
• The ancestors of a node d are the nodes on the path from d to the root. These include d's parent, d's parent's parent, d's parent's parent's parent, and so forth up to the root. Note that d's ancestors include d itself. The root is an ancestor of every node in the tree.
• If a is an ancestor of d, then d is a descendant of a.
• The length of a path is the number of edges in the path.
• The depth of a node n is the length of the path from n to the root. (The depth of the root is zero.)

Rooted Tree Terminology (cont.)

• The height of a node n is the length of the path from n to its deepest descendant. (The height of a leaf is zero.)
• The height of a tree is the depth of its deepest node = height of the root.
• The subtree rooted at node n is the tree formed by n and its descendants.
• A binary tree is a tree in which no node has more than two children, and every child is either a left child or a right child, even if it is the only child its parent has.

Binary Trees

Rooted trees can also be defined recursively. Here is the definition of a binary tree:

• A binary tree T is a structure defined on a finite set of nodes that either
  o Contains no nodes, or
  o Is composed of three disjoint sets of nodes:
    • a root node,
    • a binary tree called the left subtree of T, and
    • a binary tree called the right subtree of T.

Representing Rooted Trees

• A direct way to represent a tree is to use a data structure where every node has three references:
  o one reference to the object stored at that node,
  o one reference to the node's parent, and
  o one reference to the node's children.
• The child-sibling (CS) representation is another popular tree representation. It spurns separately encapsulated linked lists so that siblings are directly linked.
  o It retains the item and parent references, but instead of referencing a list of children, each node references just its leftmost child.
  o Each node also references its next sibling to the right.
  o These nextSibling references are used to join the children of a node in a singly-linked list, whose head is the node's first child.
Binary Search Trees

• The binary-search-tree property
  o If node y in left subtree of node x, then key[y] ≤ key[x].
  o If node y in right subtree of node x, then key[y] ≥ key[x].

• Binary search trees are an important data structure that supports dynamic set operations:
  o Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete.
  o Basic operations take time proportional to the height of the tree – $O(h)$.

• Q: Where is the minimum/maximum key?

Invalid BSTs

Binary Tree Traversals

(a) Preorder traversal
(b) Inorder traversal
(c) Postorder traversal

Inorder Traversal of BST

Prints out keys in sorted order: 10, 20, 25, 30, 31, 35, 37, 50, 53, 55, 60, 62.

Inorder-Tree-Walk(x)
1. if x = NIL then return NIL
2. if k ≥ key[x] then Inorder-Tree-Walk(left[x])
3. print key[x]
4. if k ≤ key[x] then Inorder-Tree-Walk(right[x])

Querying a Binary Search Tree

• All dynamic-set search operations can be supported in $O(h)$ time.
  • $h = \Theta(\sqrt{n})$ for a balanced binary tree (and for an average tree built by adding nodes in random order.)
  • $h = \Theta(n)$ for an unbalanced tree that resembles a linear chain of $n$ nodes in the worst case.

Tree Search

Running time $O(h)$ where $h$ is tree height

Tree-Search(x, k)
1. if $x$ = NIL or $k$ = key[x] then return $x$
2. if $k$ < key[x] then return Tree-Search(left[x], k)
3. if $k$ ≥ key[x] then return Tree-Search(right[x], k)
4. else return Tree-Search(right[x], k)
**Iterative Tree Search**

- Search for 37: Running time $O(h)$ where $h$ is tree height.

**Tree Min & Max**

- The binary-search-tree property guarantees that:
  - The minimum is located at the leftmost node.
  - The maximum is located at the rightmost node.

**Pseudo-code for Successor**

- Code for successor is symmetric.
- Running time: $O(h)$.

**BST Insertion : Pseudo-code**

- Beginning at root of the tree, trace a downward path, maintaining two pointers.
  - Parent: traces the downward path.
  - Pointer y: “walking pointer” to keep track of parent.
- Traverse the tree downward by comparing the value of node at $x$ with key[$y$], and move to the left or right child accordingly.
- When $x$ is NIL, it is at the correct position for node $z$.
- Compare $z$’s value with $y$’s value, and insert $z$ at either $y$’s left or right, appropriately.
- Complexity: $O(h)$.
  - Initialization: $O(1)$
  - While loop (3-7): $O(h)$ time
  - Insert the value (8-13): $O(1)$