Binary Trees

S. M. Niaz Arifin, PhD

Computer Science and Engineering
University of Notre Dame
Indiana, USA
Overview

• Binary tree
• definition, example
• types

• Binary search tree
• definition, example
• operations
  – search
  – maximum, minimum
  – insert
  – traversal
  – delete
Binary Tree

• a tree data structure
• each node has at most two children
  – the left child and the right child
  – may also contain a parent node
• leaf nodes: lowest-level nodes
  – no children
• always rooted
• root: only node whose parent is NIL
Binary Tree: Example

- number of nodes, $n = 9$
- height (depth), $h = 3$
Binary Tree: Types

- full binary tree
- a.k.a. a *proper* or *plane* binary tree
- every node in the tree has either 0 or 2 children
Binary Tree: Types

• complete binary tree
• every level, except possibly the last, is completely filled
• all nodes in the last level are as far left as possible
Binary Tree: Types

• balanced binary tree
• has the minimum possible height (a.k.a. depth) for the leaf nodes
• self-balancing
  – automatically keeps its height small
Binary Tree: Types

unbalanced binary tree

balanced binary tree

image source: wikipedia
Binary Search Tree (BST)

• ordered (sorted) binary tree
• node keys are always stored in such a way as to satisfy the BST property:
  – let $x$ be a node in a binary search tree
  – if $y$ is a node in the left subtree of $x$, then $y:\text{key} \leq x:\text{key}$
  – if $y$ is a node in the right subtree of $x$, then $y:\text{key} \geq x:\text{key}$
Binary Search Tree (BST)

if $y$ is a node in the left subtree of $x$, then $y: \text{key} \leq x: \text{key}$

if $y$ is a node in the right subtree of $x$, then $y: \text{key} \geq x: \text{key}$
Binary Search Tree: Properties

• allows fast search (lookup), insertion, and deletion of nodes

• supports many dynamic-set operations
  – create, add, remove, etc.

• let, $n =$ number of nodes in the tree

• **time complexity** for search, insertion, or deletion
  – proportional to the logarithm of $n$: $O(\log n)$
    • average case [ worst case: $O(n)$ ]
  – much better than linear time for unsorted arrays: $O(n)$
Binary Search Tree: Operations

• search
• maximum, minimum
• insert
• traversal
• delete
Search

• **search** the tree for a specific key
• can be *recursive* or *iterative*
• example: search for **45**
1. start at the root, $45 > 25$
   - search in right subtree
1. start at the root, 45 > 25
   – search in right subtree
2. 45 < 50, search in 50’s left subtree
Search

1. start at the root, 45 > 25
   - search in right subtree
2. 45 < 50, search in 50’s left subtree
3. 45 > 35, search in 35’s right subtree
Search

1. start at the root, 45 > 25
   – search in right subtree
2. 45 < 50, search in 50’s left subtree
3. 45 > 35, search in 35’s right subtree
4. 45 > 44, but 44 has no right subtree so 45 is not in the BST

S. M. Niaz Arifin, PhD
Search

- algorithm
- \( n = \) number of nodes in tree
- *time complexity*: \( O(\log n) \)

\[
\text{Tree-Search}(x, k)
\]

1. \( \text{if } x == \text{NIL} \) or \( k == x.key \)
2. \( \text{return } x \)
3. \( \text{if } k < x.key \)
4. \( \text{return } \text{Tree-Search}(x.left, k) \)
5. \( \text{else return } \text{Tree-Search}(x.right, k) \)

\[
\text{Iterative-Tree-Search}(x, k)
\]

1. \( \text{while } x \neq \text{NIL} \) and \( k \neq x.key \)
2. \( \text{if } k < x.key \)
3. \( x = x.left \)
4. \( \text{else } x = x.right \)
5. \( \text{return } x \)
Maximum and Minimum

- **maximum**

  \[
  \text{Tree-Maximum}(x) \\
  1 \quad \text{while } x.\text{right} \neq \text{NIL} \\
  2 \quad x = x.\text{right} \\
  3 \quad \text{return } x
  \]

- **minimum**

  \[
  \text{Tree-Minimum}(x) \\
  1 \quad \text{while } x.\text{left} \neq \text{NIL} \\
  2 \quad x = x.\text{left} \\
  3 \quad \text{return } x
  \]

- **time complexity:** \(O(\log n)\)
Insert

• insert new node $x$: works much like search
• time complexity: $O(lg n)$
• set root as current node ($current$)
• recursively examine $current$
• if $x: key < current: key$
  – if $current$ has a left child, search left
  – else add $x$ as $current$’s left child
• if $x: key \geq current: key$
  – if current node has a right child, search right
  – else add $x$ as $current$’s right child
Insert

- example: insert node 60
Insert

1. start at the root, $60 > 25$
   - search in right subtree
1. start at the root, \textbf{60} > 25
   - search in right subtree
2. \textbf{60} > 50, search in 50’s right subtree
1. start at the root, 60 > 25
   – search in right subtree
2. 60 > 50, search in 50’s right subtree
3. 60 < 70, search in 70’s left subtree
Insert

1. start at the root, 60 > 25
   – search in right subtree
2. 60 > 50, search in 50’s right subtree
3. 60 < 70, search in 70’s left subtree
4. 60 < 66, add 60 as 66’s left child
Traversals

- visit every node in the tree
- three steps to a traversal:
  1. visit the *current* node
  2. traverse its *left* subtree
  3. traverse its *right* subtree
- *order* defines different traversal methods:
  - pre-order traversal: *(1) (2) (3)*
  - in-order traversal: *(2) (1) (3)*
  - post-order traversal: *(2) (3) (1)*
- *time complexity*: \( O(\log n) \)
Traversals

- **example: in-order traversal:** (2) (1) (3)
Traversals

- example: **in-order** traversal: (2) (1) (3)
Traversals

• example: in-order traversal: (2) (1) (3)

![Tree Diagram]

- **Root**: 25
- **Left Child**: 15
  - **Left Child**: 10
    - **Left Child**: 4
    - **Right Child**: 12
  - **Right Child**: 22
    - **Left Child**: 18
    - **Right Child**: 24
- **Right Child**: 50
  - **Left Child**: 35
  - **Right Child**: 70
    - **Left Child**: 66
    - **Right Child**: 90
Traversal

- example: **in-order** traversal: (2) (1) (3)

- 4,
Traversals

- example: **in-order** traversal: (2) (1) (3)

- 4, 10,
Traversal

• example: **in-order** traversal: (2) (1) (3)

• 4, 10, 12,
Traversal

• example: in-order traversal: (2) (1) (3)

• 4, 10, 12, 15,
Traversals

- example: **in-order** traversal: (2) (1) (3)

- 4, 10, 12, 15,
Traversal

• example: **in-order** traversal: (2) (1) (3)

• 4, 10, 12, 15, 18,
Traversals

- **example:** *in-order* traversal: (2) *(1) (3)*

- 4, 10, 12, 15, 18, **22**,
Traversal

• example: **in-order** traversal: (2) (1) (3)

• 4, 10, 12, 15, 18, 22, **24**, 

• and so on...
Traversal

- example: in-order traversal: (2) (1) (3)

- 4, 10, 12, 15, 18, 22, 24, ... sorted order
Delete

• three possible cases:
  1. deleting a node with *no* children
  2. deleting a node with *one* child
  3. deleting a node with *two* children

• *time complexity*: $O(\log n)$
Delete

1. deleting a node with *no* children
   - simply remove the node from the tree

```
     8
    / \  
   5   11
  / \     
 1   6  9   13
 /     \   /     
3       7 10      3
```

delete node 7
Delete

2. deleting a node with *one* child
   – remove the node and replace it with its child

```
   delete node 1
```
3. deleting a node with *two* children
   • harder case
   • let, node $z$ (having two children) to be deleted
   • identify $z$’s successor, $y$
   • **successor** of a node $z$
     – the node $y$ with the **smallest** key in tree
       such that $y: \text{key} > z: \text{key}$
Delete

3. deleting a node with *two* children
   - z’s successor, y
   - y either is a leaf or has only the right child
     1. promote y to z’s place
     2. treat the loss of y using one of the above two solutions
3. deleting a node with two children
   • to delete: \( z = \text{node 8}; \)
   • successor, \( y = \text{node 9} \)
     – **smallest** key in tree with \( y: \text{key} > z: \text{key} \)
Delete

3. deleting a node with two children
   • to delete: $z = \text{node } 8$;
   • successor, $y = \text{node } 9$
     1. promote node 9 to node 8’s place
     2. treat the loss of node 9 using:
        • deleting a node with one child
Summary

• Binary tree

• Binary search tree
  – definition, example
  – operations
    • search
    • maximum, minimum
    • insert
    • traversal
    • delete
Advanced Binary Trees

• **Red-black tree**

• a special type of self-balancing BST

• each node has an extra color bit
  – red or black

• color bits are used to ensure the tree remains approximately balanced
  – during insertions and deletions
• Red-black tree

1. the root and all leaves (NIL) are black
2. if a node is red, then both its children are black
3. for each node, all simple paths from the node to descendant leaves contain the same number of black nodes
Advanced Binary Trees

- **AVL tree**
- another special type of self-balancing BST
- heights of the two child subtrees of any node differ by *at most one*
- if at any time they differ by more than one, *rebalancing* is done to restore this property
Questions?
Thank You!