Tree Structures in Computer Science

SEDDIK Mohamed Taki Eddine

Batna 2 UNIVERSITY
Faculty of Mathematics and Computer Science
Department of Computer Science

December 13, 2023

SORTING ALGORITHMS OUTLINE I

1. Trees: A Brief Review
   - Introduction to Tree Structures in Computer Science
   - Trees in Computer science
   - Trees in Data Structures and Algorithms
   - Types of Trees and Their Usage
2. Binary Trees
   - Binary Tree Primitives
   - Construction and Modification Primitives
   - Categories of Binary Trees
   - Specialized Binary Trees
   - Representing an Arbitrary Tree as a Binary Tree

SORTING ALGORITHMS OUTLINE II

- Binary Search Trees
- Self Balanced BSTs

3 Heap Data Structure

- What is Heap
- Heap Concept
- Heap in C
- Heap Usage
- conclusion

Introduction to Tree Structures in Computer Science

- Exploring the fundamental role of tree structures in computing.
- Understanding trees as a cornerstone in data organization and algorithm design.
- Preparing to delve into types, implementations, and applications of trees.
- Emphasizing the ubiquitous presence of trees in various computer science domains.

The Family Tree: A Real-Life Analogy for Tree Structures

- The family tree as a natural example of hierarchical structure.
- Each family member represented as a node.
- Generational layers illustrating tree levels.
- Parent-child relationships mirroring tree connections.

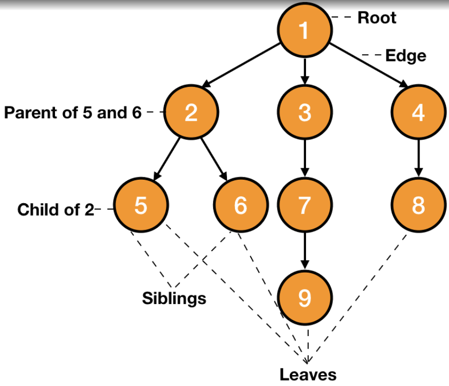
Concept of Tree Structures in Computer Science I

- **Hierarchical Data Structures**: Composed of nodes and edges forming a tree-like structure.
- **Root Node**: The topmost node, serving as the origin or starting point of the tree.
- **Child Nodes**: Nodes directly connected to another node moving away from the root.
- **Parent Node**: A node that has one or more nodes connected below it, known as children.
- **Siblings**: Nodes with the same parent are called siblings.

Concept of Tree Structures in Computer Science II

- **Ancestors**: Nodes which are higher in the hierarchy are ancestors of a given node.
- **Descendents**:Nodes which are lower in the hierarchy are descendants of a given node.

Concept of Tree Structures in Computer Science III



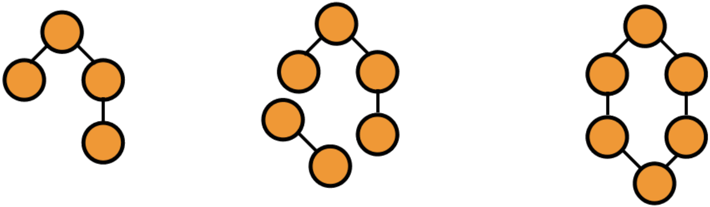
Tree

Properties of a Tree I

- **Finite and Nonempty Nodes**:
    - A tree consists of a finite number of nodes, and it cannot be empty.
- **Path to Every Node**:
    - Each node in a tree is accessible through a unique path from the root node, ensuring complete connectivity within the tree.
- **No Cycles**:
    - A tree is an acyclic structure, meaning it does not contain any cycles.

Properties of a Tree II

- The number of edges in a tree is always one less than the number of nodes.



A tree

Not a tree
All nodes are not connected

Not a tree
Cycle exists

Properties of a Tree III

Figure: Tree Vs not a Tree

Trees in Data Structures and Algorithms

- Essential for efficient data storage and retrieval.
- Fundamental in designing algorithms for sorting and searching.
- Various tree types for specific computational needs.
- Applications in databases, networking, and AI.

Application in Hierarchical Data Organization

- Ideal for representing and managing hierarchical data.
- Commonly used in file systems for organizing files and directories.
- Underpins web document structures like DOM trees.
- Facilitates decision-making processes in machine learning.

Types of Trees and Their Usage

- Binary Trees: Each node has up to two children.
- Balanced Trees (AVL, Red-Black Trees): Automatically balances itself.
- B-Trees and B+ Trees: Optimized for systems that read and write large blocks of data.
- Trie (Prefix Tree): Specialized tree used in searching, particularly with text.

Binary Tree Primitives

- Understanding Nodes: The basic unit of a binary tree.
- Properties: Value, left child, right child.
- Tree Initialization: Creating a root node.

```
// General Tree Node Structure
typedef struct TreeNode {
    int data;
    struct TreeNode *firstChild;
    struct TreeNode *nextSibling;
} TreeNode;

// Function to create a new Tree Node
TreeNode* createTreeNode(int data) {
    TreeNode *newNode = (TreeNode *)malloc(sizeof(TreeNode));
    newNode->data = data;
    newNode->firstChild = NULL;
    newNode->nextSibling = NULL;
    return newNode;
}

// Example Usage
```

Construction and Modification Primitives

- Adding Nodes: Insertion operations in a binary tree.
- Deleting Nodes: Removing nodes and restructuring the tree.
- Updating Nodes: Changing values within the tree.

Adding Node

```
typedef struct Node {
    int data;
    struct Node *left, *right;
} Node;

// Function to create a new Node
Node* newNode(int data) {
    Node* node = (Node*)malloc(sizeof(Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// Function to insert a new node in the binary tree
Node* insert(Node* node, int data) {
    if (node == NULL) return newNode(data);

    if (data < node->data)
        node->left  = insert(node->left, data);
    else if (data > node->data)
        node->right = insert(node->right, data);

    return node;
```

Deleting Nodes I

```
// Function to find the minimum value node in the tree
Node* minValueNode(Node* node) {
    Node* current = node;
    while (current && current->left != NULL)
        current = current->left;
    return current;}
// Function to delete a node from the binary tree
Node* deleteNode(Node* root, int data) {
    if (root == NULL) return root;

    if (data < root->data)
        root->left = deleteNode(root->left, data);
    else if (data > root->data)
        root->right = deleteNode(root->right, data);
    else {
        if (root->left == NULL) {
            Node* temp = root->right;
            free(root);
            return temp;}
        else if (root->right == NULL) {
```

Deleting Nodes II

```
            Node* temp = root->left;
            free(root);
            return temp;}
        Node* temp = minValueNode(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);}
    return root;}
```

Updating Nodes

```
// Function to update a node value in the binary tree
void updateNode(Node* node, int oldData, int newData) {
    Node* current = node;
    while (current != NULL) {
        if (current->data == oldData) {
            current->data = newData;
            return;
        }
        else if (newData < current->data)
            current = current->left;
        else
            current = current->right;
    }
}
```

Understanding Tree Traversal

- Tree traversal is the process of visiting each node in a tree data structure, exactly once, in a systematic way.
- Types of Traversals:
  1. Inorder Traversal
  2. Preorder Traversal
  3. Postorder Traversal
  4. Level-order Traversal
- Importance: Fundamental for operations like searching, modification, and displaying the tree.

Inorder Traversal

- Process: Left subtree → Root node → Right subtree.
- Application: Sorting, retrieving sorted data from BSTs.

```
void inorder(TreeNode *root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d␣", root->val);
        inorder(root->right);
    }
}
```

Preorder Traversal

- Process: Root node → Left subtree → Right subtree.
- Application: Creating a copy of the tree, prefix notation expressions.

```
void preorder(TreeNode *root) {
    if (root != NULL) {
        printf("%d␣", root->val);
        preorder(root->left);
        preorder(root->right);
    }
}
```

Postorder Traversal

- Process: Left subtree → Right subtree → Root node.
- Application: Deleting the tree, postfix notation expressions.

```
void postorder(TreeNode *root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d␣", root->val);
    }
}
```

Level-order Traversal

- Process: Nodes are visited level by level from top to bottom.
- Application: Breadth-first search algorithms.

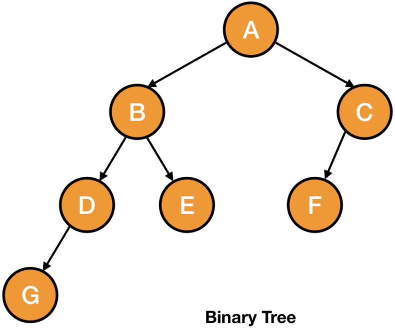
```
void levelOrder(TreeNode *root) {
    if (root == NULL) return;
    Queue q;
    enqueue(&q, root);

    while (!isEmpty(q)) {
        TreeNode *node = front(&q);
        dequeue(&q);
        printf("%d␣", node->val);

        if (node->left != NULL)
            enqueue(&q, node->left);
        if (node->right != NULL)
            enqueue(&q, node->right);
    }
}
```

Categories of Binary Trees

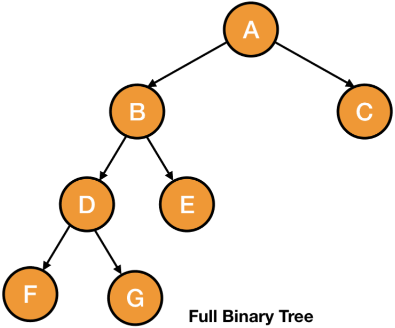
- Full Binary Tree
- Complete Binary Tree
- Perfect Binary Tree
- Balanced Binary Tree
- Degenerate (Pathological) Tree



Binary Tree

Full Binary Tree

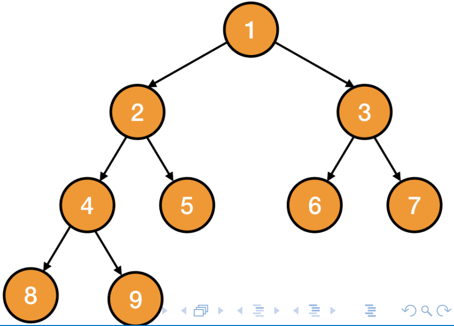
- Definition: A binary tree where every node has either 0 or 2 children.
- Properties:
    - No nodes with only one child.
    - The number of leaf nodes is one more than nodes with two children.
- Applications: Used in scenarios requiring complete binary structure without any



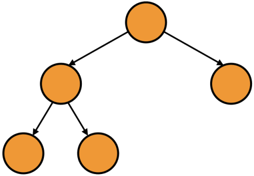
Full Binary Tree

Complete Binary Tree

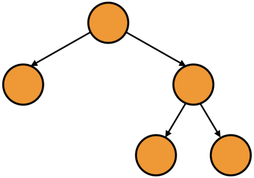
- Definition: A binary tree where all levels are completely filled except possibly the last level, which is filled from left to right.
- Properties:
  - Utilizes array-based representation efficiently.
- Applications: Ideal for priority queues and



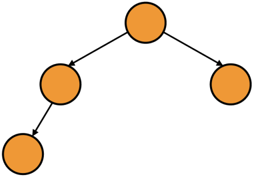
complete & Full Binary Tree



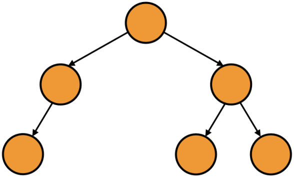
Complete and Full



Full but NOT Complete



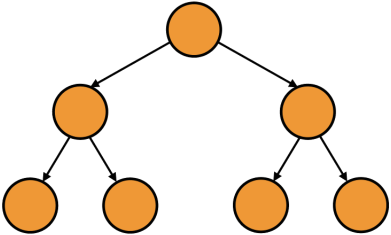
Complete but NOT Full



Neither Complete nor Full

Perfect Binary Tree

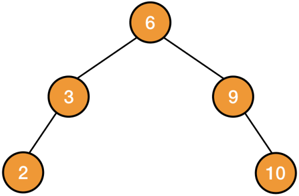
- Definition: A binary tree in which all interior nodes have two children and all leaves have the same depth or level.
- Properties:
  - Every level of the tree is fully populated.
- Applications: Used in scenarios where complete and symmetrical structure is required.



Perfect Binary Tree

Balanced Binary Tree

- Definition: A binary tree where the height of the two subtrees of any node differ by no more than one.
- Properties:
  - Ensures O(log n) complexity for insertion, deletion, and search operations.
- Examples: AVL trees, Red-Black trees.
- Applications: Widely used in real-world scenarios where frequent insertions and deletions occur.



Not Perfectly Balanced, Height = O(lgn)

Specialized Binary Trees

- Binary Search Trees (BST): Maintains a specific order for efficient searching.
- AVL Trees: Self-balancing BST for optimized searching and insertion.
- Red-Black Trees: Another form of self-balancing BST, widely used in practical applications.

Concept of Representing an Arbitrary Tree as a Binary Tree

- Transforming any tree structure into a binary tree format.
- Facilitates the application of binary tree algorithms to general trees.
- Enhances understanding and manipulation of complex tree structures.

Methods: Left-child, Right-sibling Representation

- Left Child: Represents the first child of a node in the general tree.
- Right Sibling: Represents the next sibling of a node in the general tree.
- Preserves the hierarchical and sibling relationships of the original tree.

Advantages of Binary Tree Representation

- Simplifies handling of trees with varying numbers of children.
- Makes general trees compatible with binary tree operations.
- Enhances efficiency in algorithms and data structures.

What is Binary Search Tree

Binary Search Trees

- BST Properties: Left child < Parent < Right child.
- Operations: Insertion, search, and deletion.
- Efficiency: Time complexity analysis.

Binary Search Tree in C

BST Node Structure

```c
typedef struct BSTNode {
    int data;
    struct BSTNode *left, *right;
} BSTNode;

// Helper function to create a new BST Node
BSTNode* createBSTNode(int data) {
    BSTNode *newNode = (BSTNode*)malloc(sizeof(BSTNode));
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}
```

Insertion

```
// Function to insert a new value into the BST
BSTNode* insertBST(BSTNode *node, int value) {
    if (node == NULL) return createBSTNode(value);

    if (value < node->data)
        node->left = insertBST(node->left, value);
    else if (value > node->data)
        node->right = insertBST(node->right, value);

    return node;
}
```

Search

```
// Function to search for a value in the BST
BSTNode* searchBST(BSTNode* node, int value) {
    if (node == NULL || node->data == value)
        return node;

    if (value < node->data)
        return searchBST(node->left, value);
    else
        return searchBST(node->right, value);
}
```

Deletion I

```
// Function to find the minimum value node in a BST subtree
BSTNode* minValueNode(BSTNode* node) {
    BSTNode* current = node;
    while (current && current->left != NULL)
        current = current->left;
    return current;
}

// Function to delete a node from the BST
BSTNode* deleteBSTNode(BSTNode* root, int value) {
    if (root == NULL) return root;

    if (value < root->data)
        root->left = deleteBSTNode(root->left, value);
    else if (value > root->data)
        root->right = deleteBSTNode(root->right, value);
    else {
        if (root->left == NULL) {
            BSTNode* temp = root->right;
            free(root);
```

Deletion II

```
            return temp;
        } else if (root->right == NULL) {
            BSTNode* temp = root->left;
            free(root);
            return temp;
        }

        BSTNode* temp = minValueNode(root->right);
        root->data = temp->data;
        root->right = deleteBSTNode(root->right, temp->data);
    }
    return root;
}
```

Balanced Binary Search Trees

- Importance of Balancing: Ensuring optimal operation time.
- Types: AVL Trees and Red-Black Trees.
- Balancing Mechanisms: Rotations and color changes.

AVL Trees

- An AVL tree is a self-balancing Binary Search Tree (BST).
- Balancing Factor: The height difference between left and right subtrees is no more than 1.
- Rotations are used to rebalance the tree: single and double rotations.

```c
typedef struct AVLNode {
    int data;
    struct AVLNode *left;
    struct AVLNode *right;
    int height;
} AVLNode;

// Function prototypes for AVL tree operations
AVLNode* insertAVL(AVLNode* node, int data);
AVLNode* rotateRight(AVLNode* y);
AVLNode* rotateLeft(AVLNode* x);
int getBalance(AVLNode* N);
//... Other necessary functions
```

Introduction to Heap Structures

- A heap is a specialized tree-based data structure.
- Max Heap: Parent $\geq$ Children; Min Heap: Parent $\leq$ Children.
- Applications: Priority queues, heap sort, graph algorithms.

Characteristics of Heaps

- A complete binary tree structure (or Quasi-perfect binary tree).
- Efficiently represented in an array.
- Key in various algorithm implementations.

Heap Representation in Memory

- Array-based representation.
- Index relationships: Parent at i, left child at $2i + 1$, right child at $2i + 2$.

Basic Operations in Heaps

- Insertion and its upward adjustment.
- Deletion (typically the root) and its downward adjustment.
- Heapify process to maintain heap properties.

Insertion in a Heap - C Code Example

```
void insert(int arr[], int* n, int Key) {
    *n = *n + 1;
    int i = *n - 1;
    arr[i] = Key;
    while (i != 0 && arr[(i - 1) / 2] < arr[i]) {
        swap(&arr[i], &arr[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
}
```

Deletion in a Heap - C Code Example

```
void deleteRoot(int arr[], int* n) {
    int lastElement = arr[*n - 1];
    arr[0] = lastElement;
    *n = *n - 1;
    heapify(arr, *n, 0);
}

void heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < n && arr[left] > arr[largest])
        largest = left;
    if (right < n && arr[right] > arr[largest])
        largest = right;
    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        heapify(arr, n, largest);
    }
}
```

Heap Sort - C Code Example

```
void heapSort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
    for (int i=n-1; i>=0; i--) {
        swap(&arr[0], &arr[i]);
        heapify(arr, i, 0);
    }
}
```

Advanced Concepts and Applications

- Implementing quasi-perfect binary trees in heaps.
- Heap's role in complex algorithms like Dijkstra's and Prim's.
- Heap peeling in ordered data processing.

Conclusion and Best Practices

- Heaps are essential for efficient algorithm implementation.
- Deep understanding of heap operations enhances problem-solving skills.
- Theoretical knowledge combined with practical implementation is key.