CSCI - 241 - Binary Trees

Matt Warner

Contents

Binary Trees

When every vertex in a rooted tree has at most two children and each child is designated either the (unique) left child or the (unique) right child, the result is a binary tree.

A **binary tree** is a rooted tree in which every parent has at most two children.

Each child in a binary tree is designated either a **left child** or a **right child** (but not both), and every parent has at most one left child and one right child.

A **full binary tree** is a binary tree in which each parent has exactly two children.

Given any parent v in a binary tree T, if v has a left child, then the **left subtree** of v is the binary tree whose root is the left child of v, whose vertices consist of the left child of v and all its descendants, and whose edges consist of all those edges of T that connect the vertices of the left subtree. The **right subtree** of v is defined analogously.

`~/Documents/figures/binarytree.png`

Figure 1: A Binary Tree

1 Why trees?

The number one reason why tree are used, and why they are an important data structure is because you can insert items in trees, remove items, or find items in $O(\log_n)$ time. In other words, the main benefit of implementing a binary tree is its efficency with the insertion and deletion of data, which is not the case in a linked list structure.

The way data is stored in a binary tree can be very organized. And we will see that with **Binary Search Trees**.

Another reason to use binary trees is because they are cost efficient due to their dyamic nature.

Unlike arrays that typically require a block of contiguous memory, binary trees utilize pointers to non-contiguous memory blocks for their nodes. This can be more memory-efficient, especially in scenarios where the data structure needs to be frequently resized. Trees only allocate memory for the nodes that are actually used, without needing to reserve extra capacity upfront.

Since trees don't require a predefined fixed size, they can grow as new nodes are added and shrink as nodes are removed, without the overhead of resizing an array. There is, however, a case that ruins a binary trees efficency.

2 Unbalanced Binary Trees

An unbalanced binary tree is a type of binary tree in which the heights of the two child subtrees of any node differ significantly. This height difference can lead to suboptimal performance for basic operations such as search, insertion, and deletion.

An unbalanced binary trees built from sorted data is effectively the same as a linked list.

With a balanced tree, acesss[1] is $O(\log_n)$

With an unbalanced tree, access[1] is $O(n)$ (worst case)

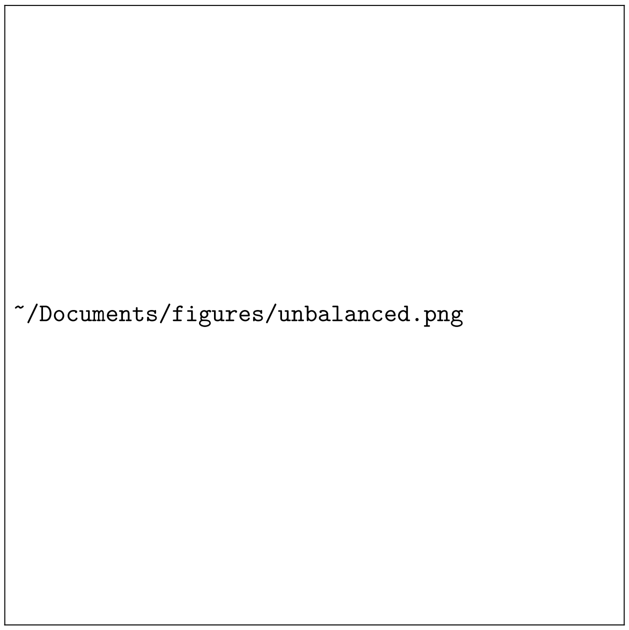


Figure 2: Balanced vs Unbalanced Binary Tree

To avoid this problem, there is a concept called ***self balancing trees***, which will be talked about in detail later on.

3 Where are Binary Trees used?

Here are some common areas where binary trees are used:

- file systems
- Databases
- Networking
- Math
- Decision trees / machine learning
- compression of files

4 Terminology of Binary Trees

- **Root Node**: The topmost node of a tree, which serves as the origin or starting point of the tree structure. It has no parent and is the only node at level 0.
- **Node (Vertex)**: A fundamental element of the tree structure.
- **Internal Nodes**: Nodes that have at least one child, i.e., non-leaf nodes.
- **Leaf Nodes (Terminal Nodes)**: Nodes without children, marking the extremities of the tree.
- **Size**: The total number of nodes within the tree.
- **Child & Parent Relationships**: Describes the hierarchical link between a node and its direct descendants or ancestor.
- **Siblings**: Nodes that share the same parent.
- **Edge**: A line connecting two nodes, indicating a parent-child relationship.
- **Height**: For a node t, the height is defined as the maximum number of edges on the longest downward path between node t and a leaf.
- **Level**: The level of a node t is defined as the number of edges along the unique path from the root node to t. The root node is at level 0.
- **Depth:** Given a node t, the depth of t is the number of edges from the root node to t.
- **Degree**: the degree of a node is the number of children it has, any node can either have degree: 0, 1 or 2.

Note:-

Level and ***Depth*** are essentially the same in that they both measure the distance from the root node to a specific node in a binary tree. However, the term ***Level*** is more commonly used to refer to all nodes that are at the same distance from the root, essentially categorizing nodes into horizontal groups within the tree.

5 Types of Binary tree

5.0.1 Complete binary tree

A complete binary tree is a special type of binary tree where all the levels of the tree are filled completely except the lowest level nodes, which are filled from as left as possible.

`~/Documents/figures/complete-binary-tree-2719456170.jpg`

5.0.2 Full / Strict binary tree

A binary tree in which every node either has exactly ***two*** or ***zero*** children.

~/Documents/figures/Binary-Tree-in-Data-Structure-2-354246584.png

5.0.3 Segment tree

In a segment tree, all internal nodes have exactly ***two*** children, and all leaf nodes are on the same ***level***.

~/Documents/figures/segment_tree.png

Figure 3: Segment tree

5.0.4 Height balanced Binary Tree

A **height-balanced binary tree** is defined as a binary tree in which the ***height*** of the left and the right subtree of any node differ by not more than ***one***.

AVl tree, red-black tree are examples of ***height-balanced trees***.

~/Documents/figures/heightbalanced.png

Figure 4: height-balanced tree

5.0.5 Skewed Binary Tree

A binary tree in which every ***node*** has precicely ***one*** child.

`~/Documents/figures/skewed-trees-1024x421.png`

5.0.6 Perfect Tree

A Perfect binary tree simply refers to a binary tree in which all ***levels*** are ***filled***.

`~/Documents/figures/img2.png`

Figure 5: A perfect binary tree

Some interesting theorems about binary trees

Theorem 8.3. If k is a positive integer and T is a full binary tree with k internal vertices, then:

1. T has a total of $2k + 1$ vertices.
2. T has $k + 1$ leaves.

Example 8.3.1. Is there a full binary tree that has 10 internal vertices and 13 terminal vertices?

Proof.

No. By Theorem 8.3, a full binary tree with 10 internal vertices should have $10 + 1 = 11$ leaves, not 13.

Theorem 8.4. For every integer $h \geqslant 0$, if T is any binary tree with height h and t leaves, then

$$t \leqslant 2^h$$

Equivalently,

$$\log_2(t) \leqslant h$$

Example 8.4.1. Is there a binary tree that has a height of 5 and 38 leaves?

Proof.

No. By Theorem 8.4, a binary tree with a height of 5 can have at most $2^5 = 32$ leaves, therefore 38 leaves is impossible.

Theorem 8.5. For a perfect binary tree with height h, the total number of nodes N is given by:

$$N = 2^{h+1} - 1$$

Example 8.5.1. How many nodes are there in a perfect binary tree with a height of 3?

Proof.

By Theorem 8.5, the number of nodes N in a perfect binary tree with a height of 3 is calculated as:

$$N = 2^{3+1} - 1 = 2^4 - 1 = 15$$

Therefore, there are 15 nodes in a perfect binary tree of height 3.

6 Binary Tree properties

For a Perfect binary tree, we have:

1. *leaf nodes:* 2^h
2. *Total number of nodes:* $2^{h+1} - 1$
3. *Total number of interal nodes*: $2^{h+1} - 1 - 2^h$

Where $h =$ the height of the tree

For a Full binary tree, we have:

1. *Leaf nodes:* $k + 1$
2. *maximum amount of nodes:* $2k + 1$

Where $k =$ the number of internal nodes

Alternatively, If $k =$ No. of leaf nodes, then then total number of *internal nodes* $= k - 1$

For calculating the minium height of a binary tree we have:

1. if n = *No. of leaf nodes*, then, min height $= \log n + 1$
2. if n = *No. of nodes*, then, min height $= \log(n + 1)$

7 Implementation

In terms of implementing a binary tree, there are two approaches:

1. Linked representation
2. Sequential representation (using an array)

8 Linked representation

8.1 Node Structure

Structurally, the following typifies the definition of a binary tree node:

```
struct node {
  int value;
  node_ptr left;
  node_ptr right;
};
```

8.2 Computing the Size of a Tree

Recursive Approach

If the tree has no nodes, then its size is zero. If the tree has at least one node it is the root node. We can “ask” its children for the sizes of trees rooted at them, and then add the two numbers together and add one for the root. This gives us the total number of nodes in the tree.

Here is the pseudocode for the recursive algorithm

```
int recSize( node<T> root) {
  if root == null
    return 0
  else
    return recSize(root->left) + recSize(root->right) + 1
}
```

Iterative Approach

The iterative method can be much more complicated, because at each node we have multiple branches, so we need to keep track of unexplored branches as we explore the others.

Here is the code:

```
size_t size () {
  // size counter
  size_t size = 0;
  // Create a pointer to a tree node
  node<T> *p;
  // Create an empty queue
  queue<node<T>*> q;

  // push the root node onto the queue
  q.push(root)
  // loops untill all nodes are visited
  while (!q.empty()) {
    // Set our tree node pointer to the front of the queue
    p = q.front();
    // Get rid of it
    q.pop()
    // Increment the counter
    size++;

    // Add children of current tree node
    if (p->left != nullptr)
      q.push(p->left)
    if (p->right != nullptr)
      q.push(p->right)
  }
  return size;
}
```

The queue allows us to put the unexplored nodes on hold while we explore other nodes.

An empty queue indicates that all the nodes in the tree have been counted.

Note:-

This Implementation is called breadth first search (BFS).

.

8.3 Binary Tree Traversal

8.3.1 Preorder Traversal

In a *preorder traversal* of a binary tree, we "visit" a node and then traverse both of its subtrees.

Usually, we traverse the node's left subtree first and then traverse the node's right subtree

~/Documents/figures/preorder.png

Printing the value of each node as we “visit” it, we get the following output:

A B X E M S W T P N C H

Recursive Approach

```
void preorder(node *p) {
  if (p != nullptr) {
    // Visit the node pointed to by p (usually a print statement)
    preorder(p->left)
    preorder(p->right)
  }
}
```

On the initial call to the **preorder()** procedure, we pass it the root of the BT.

Note:-

To convert the pseudocode above to a right-to-left traversal, just swap **left** and **right** so that the right subtree is traversed before the left subtree.

Iterative Approach

In order to backtrack up the tree from a node to its parent, we use a stack.

```
iterative_preorder() {
  node<T> *p // Pointer to a tree node
  stack<node<T>*> s; // Stack of tree nodes

  p = root; // Start at the root node

  // While p is not nullptr or the stack is not empty...
  while (p != nullptr || !s.empty()) {

    // Go all the way to the left
    while (p != nullptr) {
      // Visit the node pointed to by p
      // std::cout << p->value << std::endl;

      // Place a pointer to the node on the stack
      // before traversing the node's left subtree
      s.push(p)
      p = p->left;
    }

   // p must be nullptr at this point, so backtrack one level
   p = s.top();
   s.pop();

   // We have visited the node and its left subtree, so
   // now we traverse the node's right subtree
   p = r->right;
  }
}
```

8.3.2 Inorder Traversal

In an *inorder traversal* of a binary tree, we traverse one subtree of a node, then “visit” the node, and then traverse its other subtree.

Usually, we traverse the node’s left subtree first and then traverse the node’s right subtree.

~/Documents/figures/inorder.png

Printing the value of each node as we “visit” it, we get the following output:

E X M B S A P T N W H C

Recursive Approach

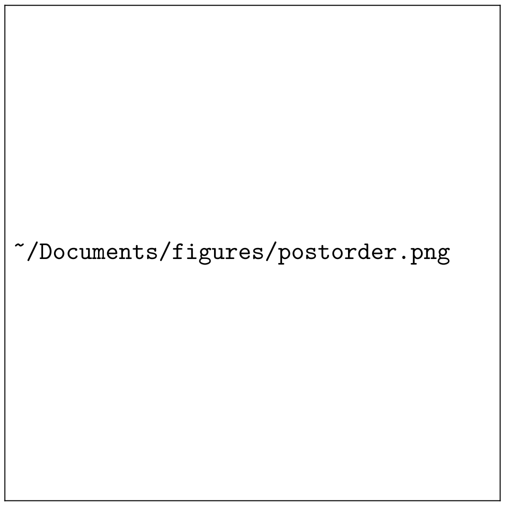
```
  void inorder(node<t> *p) {
    if (p != nullptr) {
      inorder(p->left)
      // Visit the node
      // std::cout << p << endl;
      inorder(p->right)
    }
}
```

Iterative Approach

```

void iterative_inorder() {
  node<T> *p; // Pointer to a tree node
  stack<node<T>*> s;

  p = root // Start at the root node

  while (p != nullptr || !s.empty()) {

    // Go all the way to the left
    while (p != nullptr) {
      // Place a pointer to the node on the stack before
      // traversing the node's left subtree
      s.push(p);
      p = p->left;
    }
    // p must be nullptr at this point, so backtrack one level
    p = s.top();
    s.pop();

    // Visit the node pointed to by p

    // We have visited the node and its left subtree, so now we traverse the node's right subtree.
    p = p->right
  }
}
```

8.3.3 Postorder Traversal

In a *postorder traversal* of a binary tree, we traverse both subtrees of a node, then "visit" the node. Usually we traverse the node's left subtree first and then traverse the node's right subtree.



Printing the value of each node as we "visit" it, we get the following output:

E M X S B P N T H C W A

Once again, pseudocode for a recursive inorder traversal is just a minor variation of the pseudocode for the recursive preorder and inorder traversals:

Recursive Approach

```
void Rec_postorder(node<T>* p) {
  if (p != nullptr) {
    Rec_postorder(p->left)
    Rec_postorder(p->right)
    // Visit the node
  }
}
```

Iterative Approach

Postorder traversal can be performed using a non-recursive or iterative algorithm. This is a trickier algorithm to write than the iterative preorder or inorder traversals, since we will ned to backtrack from a node to its parent *twice*. Some sources solve this problem by using two different stacks. Donald Knuth's *The art of Computer Programming* has a more efficient version of the algorithm that maintains an extra pointer to the node that was last visited and uses it to distinguish between backtracking to a node after traversing its left subtree versus backtracking to a node after traversing its right subtree.

```
  iterative_postorder() {

    node<T> * p = root; // Pointer to a tree node: start at root
    node<T> *last_visited; // Pointer to the left tree node visited
    stack<node<T>*> s; // a stack of pointer to tree nodes

    while (p != nulltpr && last_visited != root {

      // Go all the way left
      while (p != nullptr || p != last_visited) {
        // Place the current node on the stack before traversing the left subtree
        s.push(p);
        p = p->left
      }
    // p must be nullptr at this point, so we backtrack one level
    p = s.top(); s.pop();

    if (p->right == nullptr || p->right == last_visited)
      // Visit the node pointed to by p

      // Mark this node as the last visited
      last_visited = p;
    else { // Push and go right
      s.push(p)
      p = p->right
    }
  }
}
```

8.3.4 Level Order Traversal (BFS)

In a *level order traversal* of a binary tree, traverse all of the nodes on level 0, then all of the nodes on level 1, etc.

Here's an example of a left-to-right level order traversal of a binary tree.

`~/Documents/figures/bfs.png`

Printing the value of each node as we "visit" it, we get the following output:

A B W X S T C E M P N H

Iterative Level Order Traversal (BFS)

```
bfs() {
  node<T> *p; // pointer to a tree node
  queue<node<T>*> q;

  q.push(root) // start queue with the root node

  while (!q.empty()) {

    // set the tree pointer to the node at the front of the queue
    p = q.front();
    // Get rid of it
    q.pop();

    // Visit the node

    // Add its children
    if (p->left) {
      q.push(p->left);
    }
    if (p->right)
      q.push(p->right);
  }
}
```

Recursive Approach

Level order traversal can also be written as a recursive algorithm

Heres one example of that:

```
void level_order() {
  size_t h; // computed height of the tree (i.e number of levels)
  size_t i; // loop counter

  h = height(root);

  i = 1;
  while (i <= h) {
    print_level(root, i);
    i++;
  }
}
void print_level(node<T> *p, size_t level) {
  if (p == nullptr) {
    return;
  }
  if (level == 1) {
    // Visit the node pointed to by p
  }
  else if (level > 1) {
    print_level(p->left, level-1);
    print_level(p->right, level-1);
  }
}
size_t height(node<T> *p) {
  size_t l_height; // computed height of node's left subtree
  size_t r_right; // computed height of node's right subtree

  if (p == nullptr) {
    return 0;
  }
  l_height = height(p->left);
  r_height = height(p->right);

  if (l_height > r_height)
    return l_height + 1;
  else
    return r_height + 1;
}
```

9 Binary Search Trees

Definition. A binary search tree is a kind of binary tree where data records can be stored and searched efficiently. Records are arranged in a total order. If they do not have a natural total order, a key from a totally ordered set can be used. The keys guide the placement and retrieval of records.

In a binary search tree, for every internal vertex v:

- All keys in the left subtree of v are less than the key in v.
- All keys in the right subtree of v are greater than the key in v.

Example 8.5.1. Building a binary search tree for the keys $15, 10, 19, 25, 12, 4$.

Solution.

- **Insert 15**: as the root.
- **Insert 10**: to the left of 15 because $10 < 15$.
- **Insert 19**: to the right of 15 because $19 > 15$.
- **Insert 25**: to the right of 19 because $25 > 19$.
- **Insert 12**: to the right of 10 because $12 > 10$.
- **Insert 4**: to the left of 10 because $4 < 10$.

`~/Documents/figures/tree3.png`

Figure 6: Binary Search Tree for the data listed above

Building a Binary Search Tree

To build a binary search tree, start by inserting the root key. To add a new key, compare it to the key at the root and decide whether to move left or right, inserting the key into the correct position to maintain the binary search tree property.

~/Documents/figures/tree6.png

Figure 7: Steps for building the example tree