BinaryTrees

August 7, 2020

1 Tree Data Structures & Binary Trees

- https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/BinaryTreeIntro.html
- http://cslibrary.stanford.edu/110/BinaryTrees.pdf

1.1 Table of Contents

1.2 Tree Data Structure (DS)

- **Tree DS** structures in general enable efficient access and efficient update to large collections of data
- look like upside-down real-world trees

1.2.1 Some serious advantages of Tree DS

- reflect structural relationship in the data
- represent hierarchies
- provide an efficient insertion and searching
- very flexible data, allowing to move subtrees around with minimum effort (cost)

1.3 Binary Trees

- **Binary Trees** in particular are widely used for many things besides searching
  - prioritizing jobs, describing mathematical expressions, examining syntactic elements of computer programs, organizing information needed to drive data compression algorithms
- Binary Trees are made of a finite set of elements called **nodes**
  - nodes are represented as a box or a circle as shown in the following figures
  - each node typically contains data and two pointers pointing to left and right children

- Binary Tree can be either empty or consists of a special node called the **root** node with at most two binary subtrees, called the **left subtree** and **right subtree**
- subtrees are disjoint (no nodes in common)
- there's an edge (path) from a node (**parent**) to each of its **children**
- **Path**: the sequence of nodes from a node to the destination node, e.g., 5 -> 3 -> 1 is the path from node 5 to node 1 in the following figure 2.
- **length of the path** is the no. of edges in the path; if there are n nodes in the path, length is $n - 1$
  - e.g., length of path: 5 -> 3 -> 1 is 2
- if there's a path from A to B, A is the **ancestor** of B and B is a **descendant** of A
  - all nodes in the tree are descendants of the root of the tree
  - root is the ancestor of all the nodes
- **depth** of a node M in the tree is the length of the path (# of edges) from the root of the tree to M
- **leaf node** is the node that doesn't have any children
- **height** of a tree is the depth of the deepest node in the tree
  - longest path from root to one of the **leaf nodes**
- the root is at **level** 0
  - all nodes of depth d are at **level** d in the tree
- **internal node** is any node that has at least one child

1.3.1 Exercise

Describe the properties of the following binary tree: - root node? - internal nodes? - leaf nodes? - ancestors of G? - level 2 nodes? - what is the level of node I? - height of the tree? - path from A to H? - length of the path from A to H?

1.4 Special Binary Trees

1.4.1 Full binary tree

- each node is either:
  1. an internal node with exactly two children or
  2. a leaf
- Huffman coding tree is a full binary tree
- Figure (a) is full binary tree

1.4.2 Complete binary tree

- has a restricted shape obtained by starting at the root and filling the tree by levels from left to right
- in a complete binary tree of height d, all levels except possibly level d are completely full
- heap data structure is an example
- Figure (b) is complete binary tree; is it full?
- is Figure (a) complete?

1.4.3 Remember the difference:

- "Complete" is a wider word than "full", and complete binary trees tend to be wider than full binary trees because each level of a complete binary tree is as wide as possible

1.4.4 Exercise

Which statement is correct? 1. The tree is complete but not full - The tree is full but not complete - The tree is neither full nor complete - The tree is full and complete

1.5 Binary Tree as a Recursive Data Structure

- recursive data structure is a data structure that is partially composed of smaller or simpler instances of the same data structure
- e.g., **linked lists** and **binary trees**
- a linked list is a recursive data structure because a list can be defined as either:
  1. an empty list or
  - a node followed by a list
- a binary tree is typically defined as:
  1. an empty tree or
  - a node pointing to at most two binary trees
- nice visualization and animation of recursive DS: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/RecursiveDS.html

1.6 Binary Tree Theorem

- the number of empty subtrees in a non-empty binary tree is more than the number of nodes in the tree
  - empty subtrees are non-existing left/right subtree of a node

1.7 Full Binary Tree Theorem

- the number of leaves in a non-empty full binary tree is one more than the number of internal nodes with two children
- proof by mathematical induction: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/BinaryTreeFul
- see full binary tree figure above!
- the number of nodes at any level (if full) is 1 more than total number of internal nodes

1.8 Binary Tree Traversals

- process of "visiting" all the nodes in some order
  - each time performing a specific action such as printing (enumerating) the contents of the node
- three types of traversals

1.8.1 Preorder Traversal

- recursive algorithm:
  1. visit the node
  - preorder traverse left subtree
  - preorder traverse right subtree

1.8.2 Inorder Traversal

- recursive algorithm:

1. inorder traverse left subtree
  - visit the node
  - inorder traverse right subtree

1.8.3 Postorder Traversal

- recursive algorithm:
  1. postorder traverse left subtree
  2. postorder traverse right subtree
  - visit the node

Preorder enumeration of the above tree: A B D C E G F H I ### Inorder enumeration of the above tree: B D A G E C H F I ### Postorder enumeration of the above tree: D B G E H I F C A

1.9 Implementing Complete Binary Tree

- https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/CompleteTree.html
- note that complete binary tree is a BT where each level L except the last has 2^L nodes
  - the last level nodes are all left aligned
- maximum number of nodes a full and complete binary tree with height H is $2^{H+1} - 1$
- if a complete binary tree has n nodes, its height is $\lfloor \log_2(n) \rfloor$
- the nodes in the complete binary tree are inserted from left to right from level 0 to until all nodes are inserted
- usually represented using arrays (vectors)
  - indexing of nodes can start either from 0 to $(n-1)$ or 1 to n; prefer first
  - given parent index i:
    * left child is at: $2 \times i + 1$
    * right child is at: $2 \times i + 2$
  - given a child index i:
    * its parent index is at: $\lfloor \frac{i-1}{2} \rfloor$

[1]:
```cpp
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;
```

[2]:
```cpp
// Complete Binary Tree - ADT
// Array/Vector-based implementation
class CompleteBinaryTree {
    // vector to store data for binary tree of char types
    private: vector<char> bt;
    // meta data
```

```
    // root is the root index which is always as index 0 for non-empty tree
    private: int root, size, max_size;

    // does the actual inorder traversal
    private: void inorder(int root) {
        if (root >= this->bt.size() || this->bt[root] == '\0') // base case
            return;
        // inorder left-subtree
        inorder(2*root+1);
        // visit the node
        cout << this->bt[root] << " ";
        // inorder right-subtree
        inorder(2*root+2);
    }

    private:
        // converts tree to its mirror
        void mirror(int node) {
            if (this->bt[node] == '\0')
                return;

            int left = 2 * node + 1;
            int right = 2 * node + 2;
            mirror(left); // mirror left substree
            mirror(right); // mirror right subtree
            // swap the left/right nodes
            swap(this->bt[left], this->bt[right]);
        }

        // check if Tree is Full
        bool isFull() {
            return this->size == this->max_size;
        }

    // constructor
    public: CompleteBinaryTree(int max_size) {
        this->root = 0;
        this->size = 0;
        this->bt.resize(max_size);
        this->max_size = max_size;
        // initialize bt with \0 null character
        fill(this->bt.begin(), this->bt.end(), '\0');
    }

    public: // methods
        // get the actual size of Binary Tree
        int getSize() { return this->size; }
```

```
    // get the max size of Binary Tree
    int getMaxSize() { return this->max_size; }

    //updates or adds root node
    void updateRoot(char data) {
        if (bt[this->root] == '\0')
            this->size++;
        this->bt[this->root] = data;
    }

    // insert a node; left to right level by level
    void insertNode(char data) {
        if (isFull()) {
            cerr << "Debug: Binary Tree is Full!" << endl;
            return;
        }
        this->bt[size++] = data;
    }

    // insert or update left child of given parent with data
    void updateLeftChild(int parent, char data) {
        int leftChild = 2 * parent + 1;
        if (leftChild >= this->max_size)
            cerr << "Debug: Binary Tree out of bounds!" << endl;
        else if (this->bt[parent] == '\0')
            cerr << "Debug: parent at index " << parent << " does NOT exist!
↪";

        else {
            if (bt[leftChild] == '\0')
                size++; // add a new child
            this->bt[leftChild] = data;
        }
    }

    // insert or update right child of given parent with data
    void updateRightChild(int parent, char data) {
        int rightChild = 2 * parent + 2;
        if (rightChild >= this->max_size)
            cerr << "Debug: Binary Tree out of bounds!" << endl;
        else if (this->bt[parent] == '\0')
            cerr << "Debug: Parent at index " << parent << " does NOT exist!
↪";

        else {
            if (bt[rightChild] == '\0')
                size++;
            this->bt[rightChild] = data;
        }
```

```
        }

        // print all nodes level by level
        void print() const {
            for(auto ch: this->bt)
                if (ch == '\0') cout << "- ";
                else cout << ch << " ";
            cout << endl;
        }

        // public inorder method traversal
        void inorder() {
            this->inorder(0); // calls private inorder method
        }

       // FIXME: Write public preorder traversal method
       // FIXME: Write public postorder traversal method

       /* mirror tree:
         Changes the tree into its mirror image.
         So the tree...
             4
            / \
           2   5
          / \
         1   3
         is changed to...
             4
            / \
           5   2
          / \
         3   1
         Uses a recursive helper that recurs over the tree,
         swapping the left/right pointers.
        */
        void mirror() {
             this->mirror(this->root); // call private mirror
        }
};
```

1.9.1 e.g., build this binary tree

[ ]:
```
// three levels: max # of nodes = (2^4)-1 = 15
CompleteBinaryTree cbt(15);
```

[4]:
```
cbt.updateRoot('A'); // level 0; add the root
cbt.print();
cout << "size = " << cbt.getSize() << endl;
cout << "maxSize = " << cbt.getMaxSize() << endl;
```

```
A - - - - - - - - - - - - - -
size = 1
maxSize = 15
```

[5]:
```
cbt.updateLeftChild(0, 'B'); // level 1
cbt.updateRightChild(0, 'C');
cbt.updateRightChild(1, 'D'); // level 2
cbt.updateLeftChild(2, 'E');
cbt.updateRightChild(2, 'F');
cbt.updateLeftChild(5, 'G'); // level 3
cbt.updateLeftChild(6, 'H');
cbt.updateRightChild(6, 'I');
cbt.print();
cout << "size = " << cbt.getSize() << endl;
```

```
A B C - D E F - - - - G - H I
size = 9
```

[6]:
```
cbt.inorder();
```

```
B D A G E C H F I
```

[7]:
```
cbt.mirror();
cbt.print();
```

```
A C B D - F E - - - - - G I H
```

[3]:
```
char data[] = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I'};
```

[4]:
```
CompleteBinaryTree cbt1(15);
```

[5]:
```
for (int i=0; i<9; i++) {
    cbt1.insertNode(data[i]);
}
```

[6]:
```
cbt1.print();
```

```
A B C D E F G H I - - - - - -
```

1.10 Searching a Binary Tree

- How many comparisons will it need to search for some value in a Binary tree?
  - e.g., A, E, I?

- what if there are duplicate nodes (keys)?

1.11 Binary Search Tree (BST)

- a binary tree with the following properties:
  1. the key value of each node is greater than or equal to the left child and
  2. the key value of each node is less than the right child
- **inorder traversal** will enumerate the sorted order from lowest to highest key values
- BST depends on the order of the values inserted, e.g.
- two BSTs for a collection of same values inserted in two different order
  - Figure (a) will be produced if values are inserted in the order $37, 24, 42, 7, 2, 42, 40, 32, 120$
  - Figure (b) will be produced if the values are inserted in the order $120, 42, 42, 7, 2, 32, 37, 24, 40$

1.11.1 BST Search Algorithm

- if the key, K is found at the current node, return the node
- if the key K is less than the key stored in the node, recursively search in the left subtree
- if the key K is greater than the key stored in the node, recursively search in the right subtree
- if key is not found, return NULL
- see visualization of BST search here: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/BST.html#bst-search

1.11.2 BST Insert Algorithm

- handle duplicate: depending on application, either ignore or insert to the left substree
- find where the new node with given K will go
  - insert at the location maintaining BST
- see visualization of BST insert here: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/BST.html#bst-insert

1.11.3 BST Remove

- remove a node with given key K
- a bit tricky!
- four cases:
  1. if the node is a leaf node, simply delete it
  2. if the node has one child (right or left), make the child new child of it's parent
  3. if the node has two children:
     1. find and copy the data of the min node on its right subtree to the node you're deleting
     - remove the node with the duplicate value in the right subtree
- Images used from: Section **??**

1.12 BST Implementation as ADT

- implemented as container ADT using links (pointers)

[1]:
```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;
```

[2]:
```
// Binary Tree Node
template<class T>
struct Node {
    T data; //store data as key
    //int count;
    Node<T>* lTree;
    Node<T>* rTree;
};
```

[3]:
```
//  Binary Search Tree (BST) as Abstract Data Type (ADT)
// Pointer-based implementation of BST
#define DEBUG 1 // change it to 0 if you do not want debug statement to print

template <class T>
class BST {
  private:
    Node<T> *root;
    int nodeCount; // keep track of no. of nodes in BST

    //inorder traversal
    void inorder(Node<T> *p) const {
        if (p != nullptr) { // General case
            // 1. recursively call inorder on p's left subtree; traverse left␣
 ↪tree
            // 2. visit node: print the data of root/current node
            // 3. recursively call inorder on p's right subtree; traverse right␣
 ↪tree
            inorder(p->lTree);
            cout << p->data << " ";
            inorder(p->rTree);
        }
        // base case, do nothing; stop
    }

    //preorder traversal
    void preorder(Node<T> *p) const {
        // Base case: if p equals nullptr, do nothing
        // General case: otherwise do the following:
        //      1. visit node
        //      2. traverse left tree
```

```
        //      3. traverse right tree
        // LEFT as an exercise
        cout << "FIXME: Implement preorder method..." << endl;
    }

    // postorder traversal
    void postorder(Node<T> *p) const {
        // FIXME
        // LEFT as an exercise
        cout << "FIXME: Implement postorder method..." << endl;
    }

    // counts nodes in the longest path instead of edges
    // return 1 more than the actual definition of height according
    // to the opendsa text definition of height:
    // https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/BinaryTree.
↪html#definitions-and-properties
    int height(Node<T> *p) const {
        if (p == nullptr)
            return 0;
        else
            return  1 + max(height(p->lTree), height(p->rTree));
    }

    int max(T x, T y) const {
        return (x >= y) ? x : y;
    }

    int leavesCount(Node<T> *p) const {
        // FIXME - Left as an exercise
        cout << "FIXME: Implement leavesCount method..." << endl;
        // 1. Base case: if the tree is empty, return 0
        // 2. Base case: else if the left and right subtree are empty, return 1
        // 3. Otherwise, general case: return sum of leavesCount of left␣
↪subtree and leavesCount of right subtree
        return 0;
    }

    // find and return the node with key value K, nullptr otherwise
    Node<T>* find(Node<T> *p, const T& K) const {
        if (p == nullptr) return nullptr;
        if (K == p->data)
            return p;
        else if (K < p->data)
            return find(p->lTree, K);
        else
            return find(p->rTree, K);
```

```
    }

    // insert a given node into the tree
    void insert(Node<T>* &p, Node<T> *newNode) {
        /*
         Given a binary search tree pointed to by p and a newNode,
         the function inserts the newNode in the correct place in the tree.
         Since the tree could be changed, it is passed by reference.
         */
        // 1. If the tree is empty, insert at that location
        // increment nodeCount
        if (p == nullptr) {
            p = newNode;
            this->nodeCount++;
        }
        else {
            // 2. Otherwise, recurse down the tree and insert at the correct␣
↪branch
            // can handle the duplicates differently depending on the␣
↪application
            if (newNode->data <= p->data)
                insert(p->lTree, newNode);
            // 2.c. Otherwise, recursively insert newNode into the right subtree
            else
                insert(p->rTree, newNode);
        }
    }

    Node<T>* findMin(Node<T>* p) {
        if (p->lTree == nullptr) return p;
        return findMin(p->lTree);
    }

    // remove a node from the tree
    // key: the key value of the record
    void remove(Node<T>* &p, const T& key) {
        if (p != nullptr) // general case
        {
            if (p->data == key) {//found node
                if (p->lTree != nullptr && p->rTree != nullptr){//case 4: two␣
↪children
                    if (DEBUG)
                        cerr << "Debug: Deleting node with two children..." <<␣
↪endl;
                    // find and copy the data of the min node on its right␣
↪subtree to the node you're deleting
                    Node<T>* temp = findMin(p->rTree);
```

```
                p->data = temp->data;
                // remove the node with the duplicate value in the right␣
↪subtree
                remove(p->rTree, temp->data);
            }
            else if (p->rTree != nullptr){//case 2: has right child
                if (DEBUG)
                    cerr << "Debug: Deleting node with right child..." <<␣
↪endl;
                Node<T>* temp = p;
                // make the right child
                p = p->rTree;
                delete temp;
            }
            else if (p->lTree != nullptr){//case 2: has left child
                if (DEBUG)
                    cerr << "Debug: Deleting node with left child..." <<␣
↪endl;
                Node<T>* temp = p;
                p = p->lTree;
                delete temp;
            }
            else{//case 1: no child/leaf node
                if (DEBUG)
                    cerr << "Debug: Deleting leaf node..." << endl;
                delete p;
                p = nullptr;
            }
        }
        else if (p->data > key) {//search left subtree
            if(DEBUG)
                cerr << "Debug: Searching left subtree..." << endl;
            remove(p->lTree, key);
        }
        else {//search into right subtree
            if (DEBUG)
                cerr << "Debug: Searching right subtree..." << endl;
            remove(p->rTree, key);
        }
    }
}

// Reinitialize tree
void clear(Node<T>* &p) {
    if (p != nullptr) {
        clear(p->lTree); // first clear the left subtree
        clear(p->rTree); // then clear the right subtree
```

```
            delete p; // delete the node itself
            p = nullptr;
        }
    }

public:
    //Default constructor
    BST() {
        this->root = nullptr;
        this->nodeCount = 0;
    }

    // check if the bst is empty
    bool isEmpty() const {
        return this->root == nullptr;
    }

    // enumerate BST using inorder traversal
    void inorder() const {
        inorder(this->root);
    }

    // enumerate BST using preorder traversal
    void preorder() const {
        preorder(this->root);
    }

    // enumerate BST using postorder traversal
    void postorder() const {
        postorder(this->root);
    }

    //find a node with the given key and return the node if found
    Node<T>* find(const T& key) {
        return find(this->root, key);
    }

    // find an return height of BST
    int height() const {
        return height(this->root);
    }

    // find and return number of leaves in BST
    int leavesCount() const {
        return leavesCount(this->root);
    }
```

```
    // reset tree
    void clear() {
        clear(this->root);
    }

    // insert given item with key into the tree
    void insert(const T& key) {
        Node<T> *node = new Node<T>;
        node->data = key;
        node->lTree = nullptr;
        node->rTree = nullptr;
        insert(this->root, node);
    }

    // remove the node with the given key
    void remove(const T& key) {
        remove(this->root, key);
    }

    // get the value of the root node
    T getRoot(){
        return this->root->data;
    }

    //Destructor
    ~BST() {
        clear(this->root);
    }
};
```

1.13 BST Application

- quick demo to test BST operations
- create this BST

[4]:
```
// Generate BST of figure (a)
// store the numbers first in an array
int nums[] = {37, 24, 42, 7, 2, 42, 40, 32, 120};
```

[5]:
```
// Test BST
BST<int> bst;
```

[6]:
```
for (int i=0; i<sizeof(nums)/sizeof(int); i++)
    bst.insert(nums[i]);
```

[7]:
```
bst.inorder();
```

```
2 7 24 32 37 40 42 42 120
```

[9]:
```
cout << "height = " << bst.height()-1 << endl;
// Note: height (counts nodes) is 1 more than the definition of height (counts␣
↪edges)
// NOT the edges in the longest path from root to a leaf
```

```
height = 3
```

[9]:
```
@0x10d0c6ec0
```

[10]:
```
cout << "# of leaves = " << bst.leavesCount() << endl;
```

```
# of leaves = FIXME: Implement leavesCount method…
0
```

[11]:
```
Node<int> *n;
```

[21]:
```
n = bst.find(120);
```

[23]:
```
if (n != nullptr) {
    cout << "found node with data = " << n->data << endl;
}
else
    cout << "not found!" << endl;
```

```
found node with data = 120
```

[25]:
```
// print the address of the node with value 120
cout << "node n is @" << n << " and its data = " << n->data << endl;
cout << n->data << endl;
```

```
node n is @0x7f9e64444ea0 and its data = 120
120
```

[27]:
```
bst.remove(2);
bst.inorder();
```

```
Debug: Searching left subtree…
Debug: Searching left subtree…
Debug: Searching left subtree…
Debug: Deleting leaf node…
```

```
7 24 32 37 40 42 42 120
```

[28]:
```
// delete root, 37
bst.remove(37);
bst.inorder();
```

```
Debug: Deleting node with two children…
Debug: Searching left subtree…
```

```
Debug: Searching left subtree…
Debug: Deleting leaf node…
```

```
7 24 32 40 42 42 120
```

[29]:
```
// 40 should be the new root
cout << bst.getRoot() << endl;
```

```
40
```

[31]:
```
bst.remove(500);
bst.inorder();
```

```
Debug: Searching right subtree…
Debug: Searching right subtree…
```

```
7 24 32 40 42 120
```

[32]:
```
bst.clear();
bst.inorder(); // tree is empty!
```

1.14 Kahoot.it

- https://play.kahoot.it/v2/intro?quizId=2cf8fe51-dd0f-4181-a962-8cd4384dffbb
- participants: go to https://kahoot.it and enter the game code displayed on the screen

1.15 Exercise

- Kattis problem - Binary search tree - https://open.kattis.com/problems/bst
- Given two binary trees, return true if and only if they are mirror images of each other. Note that two empty trees are considered mirror images.

```
bool isMirror(BST &root1, BST &root2);
```

- Given two binary trees, return true if they are identical (they have nodes with the same values, arranged in the same way).

```
bool isSame(BST &root1, BST &root2);
```

- Given two binary trees, return true if and only if they are structurally identical (they have the same shape, but their nodes can have different values).

```
bool isIdentitical(BST & root1, BST & root2);
```

[ ]:
```
```