ECE281 Final RC

By Qizhen Sun

Table of Contents:

BST Takeaways

Definition: A binary search tree is a binary tree where for each node,

- All the nodes in the left subtree are $<, \leq$ it
- All the nodes in the right subtree are $\geq, >$ it

Time complexity for basic operations on a BST.

Operation	Average	Worst
Search	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Insert	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Delete	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$

Other Operations

- Output as sorted list: inorder traversal
- Predecessor/Successor: get the largest of left subtree/smallest of right subtree, $\mathcal{O}(\log n)$
- Rank search: Access the nodes in the tree as if they are in an array. $\mathcal{O}(\log n)$

Those BST operations are also available in the self-balancing BSTs, like AVL tree, RB tree, B tree...
We can build lots of sorted containers (`sortedcontainers.*` in Python) based on them, which are sometimes useful.

Sorted containers in C++: `std::set` , `std::map` , `std::multiset` . (usually implemented in RB tree)

AVL Tree

A nice website for visualizing common data structures: click here or here

AVL Tree:

- An empty tree is AVL balanced
- A non-empty binary tree is AVL balanced if (1) both left and right subtrees are AVL balanced, (2) the height of left and right subtrees differ by at most 1.

It is better to learn these data structures with some kinds of visualization. You are recommended to either follow some online tutorials or try on the visualization website yourself. For CN student, this series of videos are recommended.

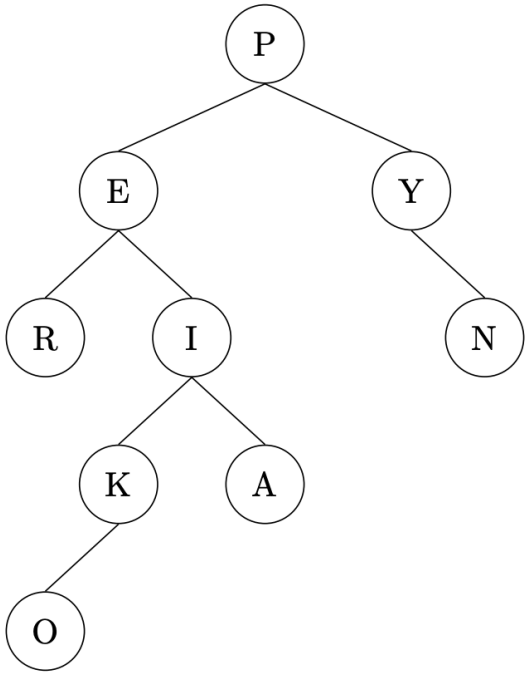
AVL Tree Checklist

For AVL tree, you should at least know the following:

- How to do a left/right rotation on a node?
- What is a balance factor?
- How to do an insertion? What kind of rotations will you need?
- How to do a deletion? How many rotations will you need at most?

Exercise

Consider the following BST with unique integer elements.



- How many elements must be greater than the element at node K?
- It is possible for a rooted tree with more than 1 node to be simultaneously a valid max-heap and a valid BST?

Exercise

Apply the following operations to an empty AVL tree.

- Insert 1, 2, 3, 4, 6, 5
- Remove 6, 5

Provide a sequence of operations inserting 1,2,3,4 on an empty AVL tree that gives a different tree.

Exercise

A rooted binary tree of height at most 10 has the same in-order and post-order traversal.

What is the maximum number of nodes that could be in this tree? Explain.

$10?$

BST Applications

Quick question: Which one is more balanced, AVL or RB tree?

Although RB tree will not be covered in the exam, you should know that in practice RB trees are very commonly used. Being less balanced means lower cost of insertion/deletion.

- Common sorted containers: RB tree
- Database: B/B+ tree

In practice, no one will ask you to implement such data structures yourself. But you should know these basics to understand how to efficiently use them.

Trie

Trie (prefix tree) is a very commonly used data structure. It is easy to understand and to implement, while it is quite efficient and has pretty many applications.

Key points:

- Two strings share the longest common prefix.
- Mark the end of a string in the trie.
- Some kind of lazy propagation can be applied. If we can already identify a string based on the prefix down to the current node, we can store the rest of the string in the node. When we insert new strings in the future, we then do propagation.

KD Tree

At first glance, KD tree does not look useful. The implementation is simple, and it is not balanced so no guarantee on the worst case time complexity. Why do we need it?

> They are used in raytacing to break a scene down into a couple of boxes. It speeds up collisions because if a ray doesn't collide with the bounding box, it surely doesn't collide with its contents.

So it is indeed useful in some situations. The main idea of KD tree is to cut the space into boxes. You should know the following

- How does KD tree cut the spaces with regard to its k dimensions?
- How to insert/delete on a KD tree?
- How to find the minimum on one dimension?
- How to do a multi-dimensional range search?
- How to do a nearest neighbor search?

Graph

There will be no questions directly ask you to write down the concepts, but you should be familiar with them.

A graph $G = (V, E)$, where V is the vertices/nodes set and E is the edges/arcs set.

- In a *directed graph*, edges are directional. $(v_i, v_j) \in E$ represents an edge from v_i to v_j.
- In an *undirected graph*, $(v_i, v_j) \in E$ represents an edge between v_i and v_j.
- A *simple graph* is a graph with no self-loops or parallel edges.
- A *weighted graph* is a graph where each edge has a weight/cost.
- In a *sparse* graph, $|E| = \Theta(|V|)$
- In a *dense* graph, $|E| = \Theta(|V|^2)$

Undirected Graph

Concepts checklist (undirected graph):

- v_i and v_j are *adjacent* if $(v_i, v_j) \in E$.
- v_j is the *neighbor* of v_i if they are adjacent.
- $e \in E$ is *incident* on v_i and v_j if $e = (v_i, v_j)$.
- The *degree* of a node is the number of edges incident to it.
- A *path* is a sequence of vertices $v_1, \cdots v_n$ where $(v_i), v_{i+1} \in E$.
- A *simple path* is a path where vertices are distinct.
- A *connected graph* is a graph where there exists a (simple) path between any two of the nodes.
- A *cycle* is a path whose start and end are the same.
- A *simple cycle* is a cycle with no repeated vertices except start and end.

Directed Graph

Concepts checklist (directed graph):

- A *strongly connected graph* is a directed graph where there is a directed path between any two of the nodes.
- A *weakly connected graph* is a directed graph where there is an undirected path between any two of the nodes.
- The *in-degree* of a node is the number of incoming edges.
- The *out-degree* of a node is the number of outgoing edges.
- A *source* is a node with in-degree 0.
- A *sink* is a node with out-degree 0.
- A *directed cycle* is a directed path whose start and end are the same.
- A *directed acyclic graph*(DAG) is a directed graph with no directed cycles.

Graph Representation

Adjacent matrix: $\mathcal{O}(|V|^2)$ space

- A $|V| \times |V|$ matrix A
- $A[i][j]$ is 1 if
  - $(i, j) \in E$ or $(j, i) \in E$ in undirected graph
  - $(i, j) \in E$ in directed graph

Time complexity:

- Check adjacent: $\mathcal{O}(1)$
- Visit all neighbors: $\mathcal{O}(|V|)$

Question: What's the mathematical meaning of $A \cdot A$ (matrix multiplication)? What about A^n?

Graph Representation

Adjacent list: $\mathcal{O}(|V| + |E|)$ space

- An array of lists
  - Each node is mapped to an index of the array
  - Each list store the neighbors of the nodes

Time complexity:

- Check adjacent: $\mathcal{O}(d(v))$, $\mathcal{O}(1)$ if a hash set is used instead of linked list.
- Visit all neighbors: $\mathcal{O}(d(v))$. $d(v)$ is the degree of v.

Question: What is $\sum d(v)$? (handshaking lemma)

BFS

```
void bfs(int start, std::vector<std::vector<int>>& neighbors) {
    auto n = neighbors.size();
    std::queue<int> q;
    std::vector<bool> visited(n, false)
    while (!q.empty) {
        int v = q.front();
        do_something(v);
        q.pop();
        for (int u : neighbors[v]) {
            if (!visited[u]) {
                q.push(u);
                visited[u] = false;
            }
        }
    }
}
```

DFS: replace `queue` with `stack`

Topological Sort

```
std::vector<int> toposort_bfs(std::vector<std::vector<int>>& neighbors) {
    auto n = neighbors.size();
    std::vector<int> in_degree(n, 0)
    // initialize in_degree, omitted
    std::queue<int> q;
    for (int v = 0; v < n; ++v) {
        if (in_degree[v] == 0) {
            q.push(v)
        }
    }
    std::vector<int> order;
    while (!q.empty()) {
        int v = q.front();
        q.pop();
        order.push_back(v);
        for (int u : neighbors[v]) {
            if ((--in_degree[v]) == 0) {
                q.push(u);
            }
        }
    }
    return order;
}
```

Exercise

Consider a weakly connected, unweighted DAG with N nodes.

- What is the best case and worst case asymptotic growth of the longest path in the graph? Provide Θ bound.

Remark: For a strongly connected graph, the problem is $\mathcal{NP}$-hard.

Exercise

Reminder: You might(will?) be asked to do some mathematical proofs in your final...

We can define pre-order/post-order/level-order traversals on DAGs just like we do for trees.

- A DAG may not be even weakly connected. If there any possible ways to do the traversal as if the DAG is strongly connected? (*Hint*: use a dummy node)
- Which (may be more than 1) of pre-order/post-order/level-order traversal can be used to get a topological sort of a DAG? Explain.
- Which traversal does the code on last slide correspond to?

Reference

- UCB CS61B course materials. https://sp24.datastructur.es/

Thank you!