**Introduction to Algorithms Notes**

**Chapter 1:** Foundations

* 1. Algorithms
* Algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.
* The **instance** of a problem consists of the input needed to compute a solution to the problem.
* A **data structure** is a way to store and organize data in order to facilitate access and modifications.

**Chapter 2:** Getting Started

2.1 Insertion sort

* The first algorithm we will be studying is insertion sort, which solves the sorting problem.
* The numbers that we would like to sort are known as **keys**.
* The input comes as an array with n elements.
* Insertion sort is efficient for sorting a small number or elements.
* Here is a visual representation of insertion sort:

A picture containing diagram, font

Description automatically generated

* Here is pseudocode for insertions sort:

A picture containing text, font, receipt, white

Description automatically generated

2.2 Analyzing algorithms

* Has come to mean predicting the resources that algorithm requires. Computational time is the main thing we want to measure.
* For this book, we assume a generic one-processer, random-access machine model of computations.
* The time taken by the insertion sort procedure depends on the input. You could have 5 inputs or 1000 inputs.
* Also, the time insertion sort takes depends also on how sorted the list is. Therefore, lists of the same size can have different execution times.
* Here is an example of the analysis of the insertion sort algorithm, where is the number of times the while loop on line 5 executes for that value of .

A picture containing text, font, receipt, screenshot

Description automatically generated

* The best case for insertion sort is when the array is already sorted.
* The worst case is when the array is reverse sorted.
* The **average case** is often roughly as bad as the worst case.

**Chapter 12:** Binary Search Trees

* The search tree data structure supports many dynamic-set operations, including search min, max, predecessor, successor, insert, and delete.
* Basic operations on a binary search tree take time proportional to the height of the tree.
  + For a tree with n nodes, such operations run in worst-case time.
  + If the tree is a linear chain of n nodes, however, the same operations take worst-case time.
* What is a binary search tree?
  + Each node is an object.
  + Each node contains a key, left, right, and p that point to the nodes corresponding to its left child, its right child, and its parent, respectively. If a child or the parent is missing, the appropriate attribute contains the value NIL. The root node is the only node in the tree whose parent is NIL.
  + The keys in a binary search tree are always stored in such a way as to satisfy the **binary-search-tree property:**
    - Let x be a node in a binary search tree. If y is a node in the left subtree of x, then y.key <= x.key. if y is a node in the right subtree of x, then y.key >= x.key.
  + The binary search tree property allows us th print out all the key in a binary search tree in sorted order by a simple recursive algorithm, called an **inorder tree walk**.
    - This algorithm prints the key of the root of a subtree between printing the values in its left subtree and printing those in its right subtree.
  + **Postorder tree walk:**
    - Prints the root after the values in its subtrees.

A picture containing text, font, white

Description automatically generated

* + This prints all elements in a binary search tree in order.
  + This algorithm takes time to walk an n-node binary search tree.
* Searching a binary tree.

A picture containing text, font, white, receipt

Description automatically generated

* The running time of tree search is , where h is the height of the tree.
* We can also utilize the iterative approach.

A picture containing text, font, receipt, white

Description automatically generated

* The minimum of a binary tree is the most left node.
* The max is the rightest node.
  + Both min and max run in .

**Chapter 13:** Red-Black Trees

* In our original binary search tree, we could complete all of our functions in time. This is only good for when out tree is small, with a large tree, this does not differ from a linked list in time.
* Red-Black trees are one of many search-tree schemes that are “balanced” in order to guarantee that basic dynamic-set operations take time in the worst case.
* A Red-Black tree is a binary search tree with one extra bit of storage per node, its color.
  + Red or black.
* This ensures that no such path is more than twice as long as any other, so that the tree is balanced.
* If there is no child or parent, the corresponding value is NIL.
* Here are the red-black tree properties:
  + Every node is red or black.
  + The root is black.
  + Every leaf (NIL) is black.
  + If a node is red, then both its children are black.
  + For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

A diagram of a graph

Description automatically generated

* Now with this principle, we can create search, min, max, successor, and predecessor methods in time.