**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.

**Chapter 18:** B-Trees

* B-trees are balanced search trees designed to work well on disks or other direct access secondary storage devices.
* They are similar to red-black trees, but better at minimizing disk I/O operations.
* Many database systems use b-trees, or variants of.
* B-trees differ from rb-trees because b-tree nodes may have many children, from a few thousands.
* They are similar to rb-trees because every n-node B-tree has height .

A diagram of a tree

Description automatically generated

* This is an example of a b-tree.
* Definition of b-trees
  + A b-tree is a rooted tree (whose root is ) having the following properties:
    - Every node has the following properties:
      * , the number of keys currently stored in node
      * The keys themselves, , stored in nondecreasing order, so that .
      * , a Boolean value that is TRUE if is a leaf and FALSE if is an internal node.
    - Each internal node also contains pointers to its children. Leaf nodes have no children, and so their attributes are undefined.
    - The keys separate the ranges of key stored in each subtree: if is any key stored in the subtree with root , then .
    - All leaves have the same depth, which is the tree’s height .
    - Nodes have lower and upper bounds on the number of keys they can contain. We express these bounds in terms of a fixed integer called the minimum degree of the b-tree:
      * Every node other than the root must have at least keys. Every internal node other than the root thus has at least children. If the tree is nonempty, the root must have at least one key.
      * Every node may contain at most keys. Therefore, an internal node may have at most children. We say that a node is full if it contains exactly keys.
  + The simplest b-tree occurs when . Every internal node then has either 2, 3, or 4 children, and we have a 2-3-4 tree. In practice, however, much larger values of t yields b-trees with smaller height.
* Basic operations on b-trees:
  + B-tree search.
  + B-tree create.
  + B-tree insert.
    - The root of a b-tree is always in main memory, so that we never need to perform a disk-read on the root; we do have to perform a disk-write of the root, however, whenever the root node is changed.
    - Any nodes that are passed as parameters must already have had a disk-read operation performed on them.
  + B-tree search
    - Like a binary search tree, except that instead of making a binary, “two-way”, branching decision at each node, we make a multiway branching decision according to the number of the node’s children. More precisely, at each internal node , we make an -way branching decision.