Week 4 - Trees & Heaps

AD 325 - Brenden West

Contents

Reading & Videos

- Carrano & Henry: Chapters 8.34, 20, 21, 24, 25
- https://www.coursera.org/learn/algorithms-part1/home/week/4

Reference

- https://algs4.cs.princeton.edu/31elementary/
- https://algs4.cs.princeton.edu/24pq/
- https://www.geeksforgeeks.org/heap-data-structure/
- https://www.geeksforgeeks.org/binary-tree-data-structure/ (Sets 1-3)

Learning Outcomes

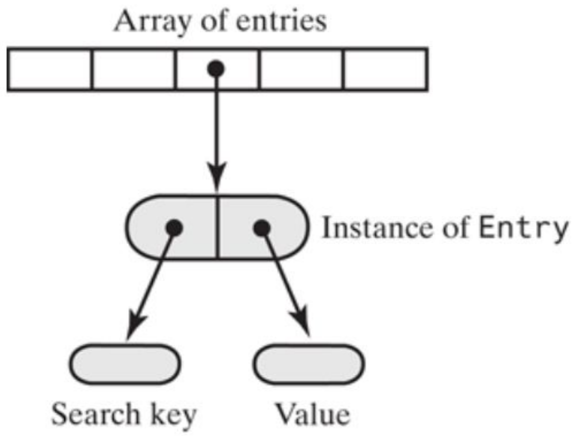
- Symbol Tables
- Tree structures
- Binary trees
- Heaps
- Priority queues

Symbol Tables

A ***symbol table*** associates a ***value*** with a ***key***, allowing clients to search for the value of a given key. A symbol table is much like an array, where keys are indices and values are array entries.

Common applications of a symbol table are - dictionary, web search, book index.

Symbol tables can be implemented using a single array, two parallel arrays, or a linked list to store data.



Symbol Table Constraints

Some constraints on symbol tables:

- ***No Duplicate keys*** - Putting a key-value pair into a table already containing that key replaces the existing value
- ***No Null values*** - No key can be associated with the value null. Because a get() should return null if the requested key is not in the table. And keys are deleted by setting their value to null.
- ***Key equality*** - Keys must be comparable

Symbol Table Operations

Symbol tables can be implemented using arrays or linked lists.

Array implementation may use a single array, where each entry is an object containing key-value pair, or use two synchronized arrays - one for keys and one for values.

Worst-case performance is generally O(n) for operations in both sorted and unsorted dictionaries, whether using an array or linked list, except retrieval from a sorted array-based dictionary is O(log n) at worst.

Ordered symbol tables allow key comparison for efficient insert & get operations and keep the table entries in order. This ordering enables a larger set of operations, such as:

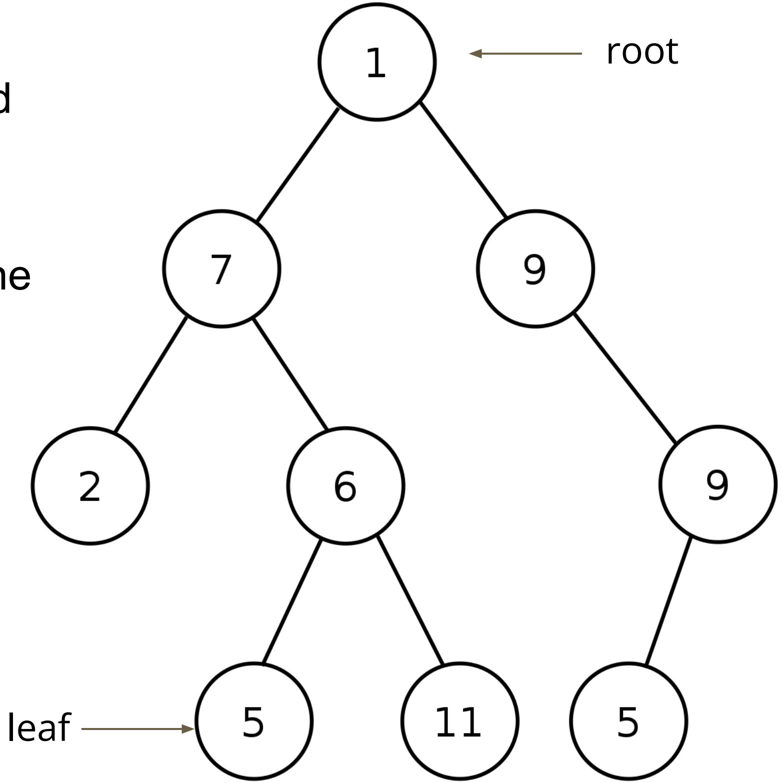
- min & max - smallest or largest key
- floor & ceiling - smallest or largest key relative to a given key
- rank - number of keys less than a given key
- range - number or collection of keys between high and low values

Trees

A **tree** is a symbol table arranged as **nodes** connected by **edges** that indicate the relationships between the nodes. The nodes are arranged in **levels** that indicate the hierarchy. The top level has a single node called the **root**.

Nodes at each successive level are **children** of a **parent** node. Nodes that are children of the same parent are **siblings**.

A **leaf** node has no children.



Trees

Trees whose nodes are constrained to some number (n) of children are called **n-ary trees**. In a **binary tree**, nodes may have at most two children.

Tree **height** is the number of levels in the tree. An empty tree has height = 0.

Nodes in a tree are accessed by a **path** starting at the root and following the connected nodes. The number of edges in a path are its **length**.

Binary Tree

In a binary tree each node has at most two children, called the **left child** and **right child**.

Properties of a binary tree

- The maximum number of nodes at level 'l' of a binary tree is 2^l
- The maximum number of nodes in a binary tree of height ‘h’ is $2^h – 1$
- In a binary tree with N nodes, minimum height is - $\log_2 (N+1)$

Types of Binary Trees

- **Full** - A binary tree is full if every node has 0 or 2 children.
- **Complete** - A binary tree is complete if all the levels are completely filled except possibly the last level, and the last level has all keys to the left as much as possible,
- **Perfect** - A binary tree is perfect if all internal nodes have two children and all leaf nodes are at the same level. A perfect binary tree of height h has $2^{h+1} – 1$ nodes.
- **Balanced** - A binary tree is balanced if the height of the tree is $O(log_n)$, where n is the number of nodes.

Binary Tree Traversal

Binary trees can be traversed (each node visited), usually through recursive methods that take one of these approaches:

- ***pre-order - visit all nodes in tree order (starting from root):***
  - visit the root,
  - traverse the left sub-tree
  - traverse the right sub-tree

- ***in-order - visit all the nodes ascending order, based on their key values:***
  - traverse the left sub-tree,
  - visit the root,
  - traverse the right sub-tree.

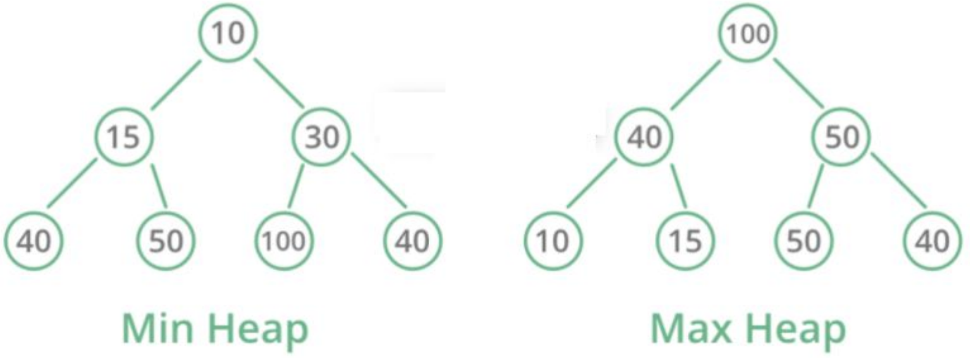
- ***post-order - useful for deleting a tree or getting 'postfix' expression:***
  - traverse the left sub-tree
  - traverse the right sub-tree
  - visit the root

Binary Heap

A Binary Heap is a **complete binary tree** structure that can efficiently support **priority-queue** operations.

A Binary Heap is either a Min Heap or a Max Heap. In a Min Binary Heap, the root node must have the minimum value among all nodes in the tree. The tree is **heap-ordered**, meaning all nodes in the tree are less than or equal to their children.

A Max Heap is similar, but with the maximum value at the root and each node larger than or equal to its children.



Heap Operations

Heap operations involve making a simple change that could violate the heap condition, then modifying the heap (**reheapifying**)as needed to restore that heap order.

- A **sink** operation is performed when a node becomes larger than its parent node. The node is exchanged with it's parent, until heap order is restored
- A **swim** operation is performed when a node becomes smaller than one or both of it's child nodes. The node is exchanged with the larger child until heap order is restored.
- Binary heaps stored in arrays can be traversed through simple arithmetic on array indices,