4 Trees

4.1 Basics

Reading: MAW 4.1 and 4.2

- Trees: Defined recursively.
  - Terminology: node, edge, parent, child, root, leaf, ancestor, descendent, level, depth, height, path.
  - Implementation: Linked list, first child/next sibling.
  - Traversal: Preorder (with root first) and postorder (with root last).
- Binary trees: At most two children, left and right.
  - Height: $O(\sqrt{n})$ on average and $O(n)$ in the worst case if there are n nodes.
  - Implementation: Linked list in most cases and array for special trees.
  - Traversal: Preorder, postorder, and inorder.

4.2 Binary search tree (BST)

Reading: MAW 4.3

- Motivation: Given a set of items (numbers). Can we store the items in a tree instead of a list/stack/queue to achieve better time complexity for operations such as insertion and deletion?
- Binary search trees:
  - What is a binary search tree?
  - Where are the maximum and the minimum located?
  - How is searching done in a binary search tree?
  - How is a new number (node) inserted?
  - How is a number (node) removed?
  - How is a binary search tree initialized given a sequence of numbers?
  - What are the worst- and average-cases for the above operations?
- Dictionary: Implementation of insertion, deletion, and search among a set of numbers.

Worst-case	I	D	S
array-based (unsorted)	$O(1)$	$O(n)$	$O(n)$
array-based (sorted)	$O(n)$	$O(n)$	$O(\log n)$
pointer-based	$O(1)$	$O(n)$ $O(1)$	$O(n)$
BST	$O(n)$ $O(h)$	$O(n)$ $O(h)$	$O(n)$ $O(h)$

Question: $O(\log n)$ for insertion, deletion, and search?

4.3 AVL trees

Reading: MAW 4.4

- Motivation: Can we use the BST idea for storing numbers, but try to keep the tree somewhat balanced all the time?
- An AVL tree is a height-balanced binary search tree. i.e., it is first a BST, and for any node, the height difference of its left and right subtrees is at most 1.
- An example of an AVL tree: Figure 4.32 (page 143).
- Height: The height of an AVL tree of n nodes is $O(\log n)$.

  Proof: What is N_h, the minimum number of nodes in an AVL tree of height h?

h	N_h	F_h
0	1	0
1	2	1
2	4	1
3	7	2
4	12	3
5	20	5

$$N_h = \begin{cases} 1 & \text{if } n = 0 \\ 2 & \text{if } n = 1 \\ N_{h-1} + N_{h-2} + 1 & \text{if } n \geq 2 \end{cases}$$

$$F_h = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{h-1} + F_{h-2} & \text{if } n \geq 2 \end{cases}$$

  We observe that $n \geq N_h = F_{h+3} - 1 = O(c^h)$. So $h = O(\log n)$.

- Maintaining an AVL tree:
  - Single rotation: Figure 4.43 on page 153. Note the rotation is done at two levels.
  - Double rotation: Figure 4.45 on page 154. Note the rotation is done at three levels.

  Time complexity for each rotation: $O(h) = O(\log n)$.
- Insertion/deletion/search: Every time the height difference of sibling subtrees becomes more than 1, single or double rotations are applied to maintain the balance of the tree.

  Time complexity for each operation: $O(h) = O(\log n)$.
- Example: Insert $13, 14, 16, 2, 1, 10, 5, 11, 6$ into an initially empty tree. Maintain the AVL tree property.

4.4 Splay trees

Reading: MAW 4.5

- Motivation: Maybe AVL (with single and double rotations) is doing too much by keeping the BST balanced all the time. After accessing a node in a BST, we can try to move the node up to the root level by some rotations so that the next time the same node needs to be accessed, the time complexity will be much smaller. One thing to remember, we don't want to push other nodes deeper as a result of the rotations.
- The first attempt uses the idea of single rotations (pp. 156-157). Although the accessed node is moved up to the root level, some other nodes are pushed deeper in the tree. So the method does not work.

- The second attempt borrows the idea of double rotations (pp. 158-159). In particular, the rotations are zig-zag (Figure 4.47 on page 158) and zig-zig (Figure 4.48 on page 158). This is called splaying.
- Splaying not only moves the accessed node to the root, but also has the effect of roughly halving the depth of most nodes on the access path with some shallow nodes being pushed down at most two levels.
- Time complexity: A single operation may take $O(n)$ in the worst case, but it may be compensated by an operation which does not take too much time due to the work done by splaying. It has been proved that a sequence of m tree operations takes $O(m \log n)$.

4.5 B trees

Reading: MAW 4.7

- A B tree of order m is such that
  - all data is sorted and stored at leaf nodes, which are at the same level;
  - the number of keys in each leaf node is between $\lceil \frac{m}{2} \rceil$ and m;
  - interior nodes of the tree contain pointers and indexing information;
  - the root is either a leaf or has between 2 and m children; and
  - all interior nodes other than the root have between $\lceil \frac{m}{2} \rceil$ and m children.
- In general, an interior node in a B tree contains $a_1 : a_2 : a_3 : \cdots : a_k$ if the nodes has $k+1$ subtrees, $T_0, T_1, \ldots, T_k$, where a_i is the smallest key in T_i.
- Example: A B tree of order $m = 4$ (which will be given in class).
  - Root: 2, 3, or 4 children.
  - Other interior nodes: 2, 3, or 4 children.
  - Leaves: 2, 3, or 4 keys.

  Such a tree ($m = 4$) is also called a 2-3-4 tree.
- Example: A B tree of order $m = 3$ (which will be given in class).
  - Root: 2 or 3 children.
  - Other interior nodes: 2 or 3 children.
  - Leaves: 2 or 3 keys.

  Such a tree ($m = 3$) is also called a 2-3 tree.
- Search: Let x be the search target. Assume that the search reaches a node with $a_1 : a_2 : \cdots : a_k$. If $a_i \leq x < a_{i+1}$, go to subtree T_i.

  Time complexity:

$$\begin{aligned} T(n) &= O(h \log m) \\ &= O(\log n \log m) \\ &= O(\log n) \qquad (\text{for fixed } m) \end{aligned}$$

- Insertion:

  Use search to locate the leaf where the new key is to be inserted. A series of splitting operations may be necessary to maintain the B tree properties.

  For example, for the 2-3 tree mentioned above,

Step 1: Insert 18 (no structure change)

Step 2: Insert 1 (a leaf of 4 keys will split into two leaves, each with 2 keys)

Step 3: Insert 19 (another split)

Step 4: Insert 28 (split, split, and split)

Time complexity:

$$\begin{aligned} T(n) &= O(mh) \\ &= O(m\log n) \\ &= O(\log n) \qquad (\text{for fixed } m) \end{aligned}$$

- Deletion:

  Use search to locate the leaf containing the key to be deleted. Use a procedure (opposite to split) to merge nodes such that each interior node has at least $\lceil \frac{m}{2} \rceil$ children.

  Time complexity:

$$\begin{aligned} T(n) &= O(mh) \\ &= O(m\log n) \\ &= O(\log n) \qquad (\text{for fixed } m) \end{aligned}$$

- Initialization:

  Start from an empty tree. Insert keys one by one. Since n insertions are performed, the time complexity is $O(n\log n)$ for fixed m.