Trees

Hierarchical Abstract Data Type (ADT)

- Root node at the top
  - All other nodes descend from the root
- May be either **position oriented** or **value oriented**
  - position oriented: ADT primarily cares about an element’s position. e.g. arrays, lists, stacks...
  - value oriented: ADT primarily cares about an element’s value. e.g. anything sorted, binary search trees...
- Terminology:
  - node: basic element of the tree
  - parent of node n: node directly above n in the tree
  - child of node n: node directly below n in the tree
  - root: start of tree, only node with no parent
  - leaf: node with no children
  - edge: connection between two nodes
  - Prichard pp. 569 has a nice list of terms
- Traversals, orders for moving through all the nodes
  - in-order:
    * “left, root, right”
    * if done to a BST or similar, result will be in sorted search-key order
  - pre-order:
    * “root, left, right”
    * commonly used to parse arithmetic / algebra expressions
      ( 2+2 -> + 2 2 )
  - post-order:
    * “left, right, root”
    * also commonly used to parse arithmetic / algebra
- Two ways to represent when programming:
  - reference based: each node has references to other node objects which are it’s children. Default representation for this course (see previous labs)
  - array based: nodes are stored in an array, each node has indices into that array which are it’s children
Binary Search Tree

Is a Binary Tree (every node has at most 2 children), such that:

- any node’s left subtree only contains nodes with keys less than the current node
- any node’s right subtree only contains nodes with keys greater than current node

Basic properties:

- Full: tree of height \( h \) with no missing nodes, all leaves are at level \( h \) and all other nodes have two children
- Complete: tree of height \( h \) which is full to level \( h - 1 \), level \( h \) filled left to right
- Balanced: left and right subtrees of any node have height which only differs by 1.

Operations:

- Search:
  - move through tree looking for a search key. At any node, compare search key with that node’s data. If less than, go left. If greater than, go right. If equal, search key found.

- Insert:
  - move through tree in same manner as search, looking for an empty child node to put the data in

- Delete, depends on type of node
  - if it’s a leaf node, just delete it
  - if it has only one child, delete it and replace with child
  - if it has two children (is in middle of tree): find the in-order successor node, swap this node’s data with the successor, delete the successor.

Tree sort: put all data into a BST, do an in-order traversal. The tree’s nodes will be visited in sorted search key order.
Big O complexity: Search, insert, delete all have same performance or \( O(\log n) \) average and \( O(n) \) worst case. Traversals is always \( O(n) \).

2-3 trees

Improvement upon binary trees, designed to keep tree balanced when inserting and deleting data. Has two types of node:

- 2-node: one data element, 2 child nodes
- 3-node: two data elements, 3 child nodes
2-3 trees retain the left < root < right ordering of binary search trees, thus algorithms such as search and traversal are very similar to those in BSTs. Only new thing is 3-nodes, which have two data elements, recorded in sorted order. When visiting a 3-node in traversal / search, both its elements are visited or searched in order before moving on to the next node.

- Insert is a bit different:
  - Locate the leaf at which a search for the data to be inserted would terminate.
  - Add the data to that leaf, one of two things happens:
    * A) leaf contains two or less data items: done!
    * B) leaf contains three data items: must split leaf. if leaf was (low, mid, high):
      · promote mid to the parent node (move it up there)
      · low and high become 2-nodes under the parent node
      · if promotion of mid over filled the parent, recurse to split the parent in the same manner
      · if splitting the root, mid becomes the new root as a 2-node

- Delete is similar:
  - Locate data to delete:
    * if it is a leaf node, just delete it.
    * if it is not a leaf, swap data with in-order successor. Delete the successor (which will be a leaf)
      · this may cause an empty node somewhere down the tree: if a sibling has two items: rotate into the empty node if no sibling has two items: combine child and parent into a 3-node

An example of inserting (on next page). **Note that these are not necessarily correct 2-3 trees, they are purely an example of insertion.** Also, the 4-node is purely for explanation, since several of these steps probably happen within the same method call.
Figure 3: Example of inserting stuff into a 2-3 tree. Note, these 2-3 tree are not necessarily correct, purely an example of insert

**2-3-4 tree**

Improvement upon the 2-3 tree. many algorithms are very similar (search, traverse, etc...). Big difference is:

- has 4-nodes. Three data elements, four children.
- Insert: splits up any 4-nodes it encounters while looking for insertion point
  - then inserts into whichever node make sense, this may generate a 4-node which will be split up late when another insert finds it
– to split a 4-node: move its middle data element into the parent node. If the parent is now a 4-node, it will be split by the next insert.

• Delete:
  – find the node
  – swap with in-order successor (always delete in a leaf)
  – if leaf is a 3 or 4-node, remove item
  – if you never delete from a 2-node, deletion can be done with only one pass through tree (transform each 2-node into a 3 or 4-node while searching for delete item)

**Grading**

Attendance only, come sign sheet