NOTE: when using auxiliary data structures (e.g. stack and queue) can just publically inherit and add logic for the common interface methods

Implementation

Binary Search Tree

- many nullary public methods have a corresponding private method that takes a node (root to start) as its only argument
- constructors:
  - default -> root = nullptr
  - copy
  - move
- destructor:
  - call private clear() and then set root = nullptr
- traversal:
  - NOTE: each can potentially take a visit function (pointer, lambda, etc)
  - preorder (recursive):
    - nullary calls unary with one Node
    - if node != nullptr
      - visit node
      - preorder(node->left)
      - preorder(node->right)
  - inorder (recursive):
    - nullary calls unary with one Node
    - if node != nullptr
      - inorder(node->left)
      - visit node
      - inorder(node->right)
  - postorder (recursive):
    - nullary calls unary with one Node
    - if node != nullptr
      - postorder(node->left)
      - postorder(node->right)
      - visit node

- breadth first:
    - create queue
    - node = root
    - if node != nullptr
        - enqueue node
        - while !queue.empty
            - node = queue.dequeue
            - visit node
            - enqueue node-> left if node->left != nullptr
            - enqueue node-> right if node->right != nullptr
- iterative preorder:
    - create stack
    - node = root
    - if node != nullptr
        - push node
        - while !stack.empty
            - node = stack.pop
            - visit node
            - push node-> left if node->left != nullptr
            - push node-> right if node->right != nullptr
- iterative inorder:
    - NOTE: iterative inorder takes additional work
    - create stack
    - node = root
    - while node != nullptr
        - while node != nullptr
            - if node->right != nullptr
                - push node->right
            - push node
            - node = node->left
        - node = stack.pop
        - while !stack.empty
            - visit node
            - node = stack.pop
        - visit node
        - is !stack.empty
            - node = stack.pop
        - else node = nullptr

  - iterative postorder:
    - NOTE: this is not just reordering preorder, visit has to follow pop in the while loop for the ordering to be any different
    - NOTE: can alternatively use two stacks to achieve the same results
    - create stack
    - current = root
    - previous = root
    - if node != nullptr
      - for ; current->left != nullptr; current = current->left
        - push current
      - while current->right != nullptr or current->right = previous
        - visit current
        - return if stack.empty
        - current = stack.pop
      - push current
      - current = current->right
  - threaded inorder:
  - morris inorder:
    - NOTE: modifies tree by making nodes with a left descendant the right child of the rightmost node in the node's left descendant
    - create node and tmp
    - while node != nullptr
      - if node->left == nullptr
        - visit node
        - node = node->right
      - else
        - tmp = node->left
        - while tmp->right != nullptr and tmp->right != node
          - tmp = tmp->right
        - if tmp->right == nullptr
          - tmp->right = node
          - node = node->left
        - else
          - visit node
          - tmp->right = nullptr
          - node = node->right

- search:
  - return Value*
  - take Node and Key&

    - while node != nullptr
        - if key == node->key
            - return &node->value
        - else if key < node->key
            - node = node->left
        - else node = node->right
    - return nullptr
- insert/put:
    - iterative:
        - take Key& key, Value& value
        - if root is nullptr
            - create new node with key, value
            - set as root
            - return
        - create current = root and previous = nullptr
        - while current != nullptr
            - previous = current
            - if key < current->key
                - current = current->left
            - else current = current->right
        - if key < previous->key
            - previous->left = new node with key, value
        - else
            - previous->right = new node with key, value
    - recursive:
        - two methods:
            - one takes key, value
            - one takes node, key, value
        - if node == nullptr
            - return new node with key, value
        - if key < node->key
            - node->left = recursively call(node->left, key, val)
        - else if key > node->key
            - node->right = recursively call(node->right, key, val)
        - else node->value = value
        - (set size of node if stored)
        - return

- get:
  - return Value*
  - use search algorithm above and return nullptr for search miss
- delete:
  - delete by merging
  - find and delete by merging
  - delete by copying
- balance:
  - balance factor (either computed or stored):
    - one book says balance factor should be -1, 0, 1
    - return height of node->left - height of node->right
  - with sorted array (recursively):
    - take as arguments: vector data(sorted), first, last
    - if first <= last
      - middle = (first + last) / 2
      - insert/put middle intro tree
      - balance(data, first, middle)
      - balance(data, middle + 1, last)
  - DSW algorithm:
    - TODO: implement this and review
    - also see Day-Stout-Warren on wiki for another approach
    - NOTE: creates a backbone (a linked list from root to right)
    - create backbone:
      - take root as argument
      - grandparent = nullptr
      - parent = root
      - left child
      - while parent != nullptr
        - if parent->left != nullptr
          - grandparent = rotate right(parent)
          - parent = left child
        - else
          - grandparent = parent
          - parent = parent->right

    - make rotations:
      - take an integer n as argument
      - parent = root
      - for n > 0; --n
        - rotate left(parent)
    - dsw balance:
      - if root != nullptr
        - create backbone
        - for tmp = root, n = 0; tmp != nullptr; tmp = tmp->right
          - ++n
        - m = (2 ^ lg(n + 1)) - 1
        - make rotations (n - m)
        - while m > 1
          - m = m / 2
          - make rotations (m)
  - for AVL tree:
    - if balance factor of node < -1
      - if balance factor node->right > 0
        - node->right = rotate right node->right
      - node = rotate left node
    - else if balance factor of node > 1
      - if balance factor node->left < 0
        - node->left = rotate right node->left
      - node = rotate right node
    - return node
- rotate right:
  - option 1:
    - take as arguments grandparent, parent, child
    - grandparent, parent != nullptr
      - parent->left = child-right
      - child->right = parent
      - set grandparent's parent pointer to child
  - option 2:
    - take one node pointer reference as argument (parent)
    - if parent != nullptr
      - child = parent->left
      - parent->left = child->right
      - child->right = parent

            - parent = child
        - (moves E and its left subtree up one level and S and its right subtree down one level)

- general:
    - contains
    - is empty
    - size
    - height:
        - if node == nullptr
            - return 0
        - return 1 + max of height of node->left and height of node->right

- symbol table:
    - keys
    - values
    - min
    - max
    - floor(Key)
    - ceiling(Key)
    - rank(Key)
    - select(int)
    - delete min
    - delete max
    - size(lo, hi)

Threading Trees

- nodes hold pointers to predecessors and successors
- overload existing pointers:
    - left is either:
        - left child
        - predecessor
    - right is either:
        - right child
        - successor
- successor = top of tree after a leaf
- add a data member to indicate if the right points to child or successor

Iterators (in C++)

- need a parent link in the node to just store a pointer
- useful to use a dummy root node to implement end
- with no parent link, can construct a linear structure (stack, queue, vector, etc) to store node pointers in a specific order but this will easily be invalidated
- can also always store parent and current ptr in iterator and set parent to nullptr for root (doesn't work, cannot move back up still)

Handling misses in APIs that take keys as an argument

- C++ -> first option is to choose is throw std::out_of_range or related
- Java implementation allows for null, this requires a pointer type in C++
- the misses return null (Java)
- the C++ std library returns an iterator that references an element or is equivalent to the end() iterator
- Herb Sutter says for DAG/tree that hands out strong references to data work with shared_ptr (since keys can be data/large data also, makes sense to use them here as well)

Value semantics

- in general, do not make as much effort to avoid value semantics
- libraries use references for speed but value semantics simplifies APIs and can be more understandable

Options

- APIs use references
- use pair to store key & value
- use smart pointers to store key & value
- add parent pointers to nodes
- to solve the issue of the usage of null in Java API's, instead of storing as a pointer, can just return the address of the data as a T* (Non_owning_raw_pointer)
- use weak_ptr to return references to shared_ptr