Trees

- mathematical abstraction:
  - often occurs implicitly in algorithms
  - used to describe dynamic properties of algorithms
  - build concrete realizations of trees for problem solution
- the union find family of solutions to the connectivity problem generates a tree structure
- types of trees:
  - in decreasing generality:
    - trees
    - rooted trees
    - ordered trees
    - m-ary trees and binary trees
- definitions:
  - vertex: (node) has a name (ID) and can carry other information
  - edge: a connection between two vertices (nodes) (also carries information in graph problems)
  - path: a list of distinct vertices with connections
  - tree: a nonempty collection of vertices and edges that satisfy the following properties
    - exactly one path connecting any two nodes
    - NOTE: if there is more than one path between a pair of nodes or no path then it is a graph
  - forest: a disjoint set of trees
  - rooted tree: one node is designated as the root (generally implied by the term tree)
  - free tree: more general structure (no designated root)
  - subtree: a tree delineated by taking a non-root node in a rooted tree as the root
  - parent, children, grandparent, sibling
  - leaves (terminal nodes): nodes with no children
  - nonterminal nodes: nodes with children
  - ordered tree: a rooted tree with the child at each node specified with a position
  - m-ary tree: a tree where each nonterminal node must have a specific number of children
  - general tree:
  - external node: node with no children
  - internal node: node with n children
  - binary tree:
    - an external node or an internal node connected to a pair of binary trees (the left and right subtree)

- m-ary tree:
    - an external node or an internal node connected to a sequence of M trees that are also m-ary trees
  - tree/ordered tree: a node (the root) connected to a sequence of disjoint trees (i.e. a forest)
  - rooted tree/unordered tree: root connected to a multiset fo rooted trees (unordered forest)
  - graph: set of nodes with a set of edges that connect distinct pairs of nodes (with at most one edge connecting any pair of nodes (NOTE: edges have directions so 0,1 is distinct from 1,0))
  - simple path: a sequence of edges defining a path between nodes with no node appearing tqice
  - connected graph: a simple path connecting any pair of nodes
  - cycle: a connected graph with an equivalence between the first and final nodes
- properties:
  - for a tree, there is a exactly one path between the root and each of the other nodes in the tree (this implies no direction for edges, edges point away from the root)
  - there is a one-to-one correspondence between binary trees and ordered forests
  - ordered trees can represent unordered trees (tree isomorphism problem)
- NOTE: there are multiple representations of trees
  - just child node links
  - child node links and a parent node link
  - there is an equivalence between representing the children of a node as a linked list and a corresponding binary tree (review this and consider it)

Mathematical Properties of Binary Trees

- properties:
  - a binary tree with N internal nodes has N + 1 external nodes
  - a binary tree with N internal nodes has 2 * N link; N - 1 links to internal nodes and N + 1 links to external nodes
  - the external path length of a binary tree with N internal nodes is 2 * N greater than the internal path length
  - the height of a binary tree with N internal nodes is at least lg(N) and at most N - 1
  - the internal path length of a binary tree with N internal nodes is at least N * lg(N / 4) and at most N * (N - 1) / 2
- definition:
  - level: a node's level is 1 higher than the level of the parent (with the root at 0)
  - height: max of the levels of a tree's nodes
  - path length: the sum of the levels of the tree's nodes
  - internal path length: sum of the levels of the all of the tree's internal nodes
  - external path length: sum of the levels of all of the tree's external nodes

Tree Traversal

- traversal ordering:
  - preorder:
    - current node
    - left subtree
    - right subtree
  - inorder:
    - left subtree
    - current node
    - right subtree
  - postorder:
    - left subtree
    - right subtree
    - current node
  - preorder - operate on current node (i.e. visit) then visit the left subtree then visit the right subtree
- recursive tree traversal (preorder):

```
void traverse(link h, void visit(link)) {
    if (h == nullptr) {
        return;
    }
    visit(h);
    traverse(h->l, visit);
    traverse(h->r, visit);
}
```

- pay attention to the state of the call stack for these
- tree traversal (non-recursive, preorder):

```
void traverse(link h, void visit(link)) {
    Stack<link> s(max);
    s.push(h);
    while (!s.empty()) {
        visit(h = s.pop());
        if (h->r != 0) {
            s.push(h->r);
        }
        if (h->l != 0) {
            s.push(h->l);
        }
```

```
    }
}
```

- for ordering in the non-recursive version:
  - preorder:
    - push right, left, node
    - push right, node, left
    - push node, right, left
- level order traversal:

```
void traverse(link h, void visit(link)) {
    Queue<link> q(max);
    q.put(h);
    while (!q.empty()) {
        visit(h = q.get());
        if (h->l != 0) {
            q.put(h->l);
        }
        if (h->r != 0) {
            q.put(h->r);
        }
    }
}
```

- these approaches are important because they correspond to different orderings for processing work to be done
- level order does not inherently correspond to a recursive implementation that corresponds to the recursive structure of a tree
- each of these traversal approaches correspond to forests as well (if you think of a forest as a tree with an imaginary root)
  - preorder: visit the root then each of the subtrees
  - postorder: vist the subtrees then the root
  - level order: same as for binary trees

Recursive Binary Tree Algorithms

- recursive nature of the tree as a data structure leads to recursive algorithms for tree operations
- tree operations:

```
int count(link h) {
    if (h == nullptr) return 0;
    return count(h->l) + count(h->r) + 1;
}

int height(link h) {
    if (h == nullptr) return -1;
```

```
    int u = height(h->l), v = height(h->r);
    if (u > v) return u+1; else return v+1;
}

void print_node(Item x, int h) {
    for (int i = 0; i < h; i++) cout << "  ";
    cout << x << endl;
}
void show(link t, int h) {
    if (t == nullptr) { printnode('*', h); return; }
    show(t->r, h+1);
    printnode(t->item, h);
    show(t->l, h+1);
}
```

- for print, the effect of preorder vs. postorder is important to consider
- tournament (tree): is a form of binary heap/min (max) tree
- joining tournaments:
  - create new node
  - make the new node's left link point to one tournament
  - make the new node's right link point to the other
- constructing a tournament from an array:

```
struct node
  { Item item; node *l, *r;
    node(Item x)
      { item = x; l = 0; r = 0; }
  };
typedef node* link;
link max(Item a[], int l, int r) {
    int m = (l+r)/2;
    link x = new node(a[m]);
    if (l == r) return x;
    x->l = max(a, l, m);
    x->r = max(a, m+1, r);
    Item u = x->l->item, v = x->r->item;
    if (u > v) {
        x->item = u;
    } else {
        x->item = v;
    }
    return x;
}
```

- prefix expression parse tree:

```
char *a; int i;
struct node {
    Item item; node *l, *r;
    node(Item x) {
        item = x;
```

```
        l = 0;
        r = 0;
    }
};
typedef node* link;
link parse() {
    char t = a[i++]; link x = new node(t);
    if ((t == '+') || (t == '*')) {
        x->l = parse();
        x->r = parse();
    }
    return x;
}
```

Graph Traversal

- graph depth-first search is a generalization of tree traversal methods
- depth-first search:

```
void traverse(int k, void visit(int)) {
    visit(k);
    visited[k] = 1;
    for (link t = adj[k]; t != 0; t = t->next) {
        if (!visited[t->v]) {
            traverse(t->v, visit);
        }
    }
}
```

- can also define a DFS method that uses an explicit stack
- gives a linear time solution to the connectivity problem
- for large graphs it may still be preferable to use the union find technique because the whole graph takes space proportional to E whereas union find takes space proportional to V
- breadth-first search:

```
void traverse(int k, void visit(int)) {
    Queue<int> q(V*V);
    q.put(k);
    while (!q.empty()) {
        if (visited[k = q.get()] == 0) {
            visit(k); visited[k] = 1;
            for (link t = adj[k]; t != 0; t = t->next) {
                if (visited[t->v] == 0) q.put(t->v);
            }
        }
```

```
    }
}
```

- properties:
  - DFS requires time proportional to num vertices + num edges when using an adjacency list representation

2-3 search trees

- definition:

  - 2-3 search tree:
    - empty (null link)
    - a 2-node (with one key and associated value) and two links, a left link to a 2-3 search tree with smaller keys, and a right link to a 2-3 search tree with larger keys
    - a 3-node with two keys and associated values and three links, a left link to a 2-3 search tree with smaller keys, a middle link to a 2-3 search tree with keys between the node's keys, and a right link to a 2-3 search tree with larger keys
  - perfectly balanced (2-3 search tree): all null links are the same distance from the root

- search: compare at current node, otherwise recursively following the links that point to the interval corresponding to the key

- insertion:

  - insert into a 2-node - replace 2-node with 3-node
  - insert into a 3-node:
    - convert 3-node into a temporary 4-node
    - if 4-node is the root:
      - convert temporary 4-node into 3 2-nodes
    - if parent is a 2-node:
      - move the middle key to the parent, converting it two a 3-node
    - if parent is a 3-node:
      - split 4-node and move middle key to the parent
  - the process for modifying a 3-node moves upwards to the last parent that is a 3-node, thus modifying its parent

- 2-3 tress maintain perfect balance by inserting/growing upwards -> a full tree will convert from all 3-nodes along the path of insertion to the root to 2-nodes and the level will be added at the root

- properties:
    - search and insert operations in a 2-3 tree with N keys are guaranteed to visit at most lg (N) nodes

Red-black BSTs

- definition:
    - red-black BST:
        - a form of binary search tree (i.e. has same requirements as BST)
        - has red and black links (i.e. node pointers)
        - red links (pointers to nodes with a red coloring) lean left
        - no node has two red links connected to it
        - the tre has perfect black balance: every path from the root to a null link has the same number of black links
- there is a 1-1 correspondence between red-black BSTs and 2-3 trees
- encode red/black as boolean (or enum/enum class) in nodes
- by convention, null links are black
- when referring to the color of a link it means the color of the node being pointed to by the link
- rotation:
    - rotate left: starts with a child that is greater than the parent and moves the trees so that the child (previously parent) is less than the parent
    - rotate right: starts with a child tht is less than the parent and moves the trees so that the child (previously parent) is greater than the parent
    - the end result of a rotation always needs to reset the link to the parent (in the grandparent, or parent of the parent)
    - NOTE: an alternative approach is to either pass the grandparent also or pass the pointer as a reference (or pointer to pointer) and modify it upon completion
    - always preserve color in parent by setting new parent's color equal to new child's color (and setting new child's color equal to red)
- insertion:
    - insert into a single 2-node (i.e. a node with two black links or null links)
        - if the new key is less than, just add a left link to a red node
        - if greater, insert the red node to the right and set the root equal to the result of a left rotation about the root
        - results in red-black representation of a 3-node
    - insert into a 2-node at the bottom:
        - always attach a new red node and follow the instructions above

- insert into a 3-node:
  - covers 3 cases where a 3-node (i.e. a parent black node and child red node (left link)) have 3 potentially null links:
    a. parent's right child (greater than both)
    b. child's left child (less than both)
    c. child's right child (between parent and child keys)
  a. simple:
     - add right black node to parent
  b. results in two red links in a row:
     - attach red node to left of child
     - rotate right about the parent
     - then flip the colors of the new parent's (old child's) right and left child nodes to black
  c. again results in two red links in a row, one from parent to child to the left, and one from child to the new node to the right:
     - rotate new node left about the child (results in case 2)
     - rotate right about the parent
     - flip the color's of the new parent's children
- insertion to 3-node at bottom:
  - follow same process as above which will result in the need to pass red links up the tree, just as a 2-3 tree grows upwards
  - will result in recursively moving up the tree and handling doubled red links as described above
- general process:
  - right = red, left = black:
    - rotate left about parent
  - left child = red, left child of left child = red:
    - rotate right about parent
  - left child = red, right child = red:
    - flip colors

- flipping colors:
  - set passed node to red
  - set left and right children to black
  - NOTE: to preserve the requirement that root->color == black, always color the root black at the end of each insertion

- deletion:
  - requires transformations on the way down and on the way up
  - insertion in top-down 2-3-4 trees:
    - on way down transform any 4-node to ensure there is room at the bottom (same as splitting 4-nodes in 2-3 trees)
    - on the way up transform any 4-node to restore balance

    - for red-black BSTs, implement the top-down 2-3-4 corollary by:
        - representing 4-nodes as a balanced subtree of three 2-nodes with both left and right children as red nodes
        - split 4-nodes on the way down with color flips
        - balance 4-nodes on the way up with rotations (as for insertion)
    - delete the minimum:
        - can easily delete from a 3-node at the bottom of a tree
        - cannot easily delete from a 2-node at the bottom of a tree
            - leaves a node with no keys, replacing with null violates balance
        - if the root is a 2-node and both children are 2-nodes, convert to temporary 4-node, otherwise, borrow from right sibling to ensure the left child of the root is not a 2-node
        - to avoid ending on a 2-node, transform on the way down by maintaining that the current node is not a 2-node
            - if left child is not a 2-node, do nothing
            - if the left child is a 2-node and sibling is not, move a key from the sibling to the left-child
            - if the both are 2-nodes, combine them with the parent's smallest key to make a 4-node, changing parent from 3-node to 2-node or 4-node to 3-node
        - on the way up, split any temporary 4-nodes
    - deletion:
        - TODO: more here and review deletion in normal BSTs
        - use same transformations as delete the minimum

- properties:

    - all symbol table operations in red-black BST's are guaranteed to be logarithmic in the size of the tree (except range search)
    - the height of a red-black BST with N nodes is no more than 2*lg(N)
    - the average length of a path from the root to a node in a red-black BST N nodes is about 1.00*lg(N)
    - in a red-black BST, the following operations take logarithmic time in the worst case: search, insertion, finding the minimum, finding the maximum, floor, ceiling, rank, select, delte the minimum, delete the maximum, delete, and range count

```
                 worst-case (n inserts)  avg cost (n random inserts)
algorithm        search       insert       search hit      insert       efficient orde

sequential       N            N            N/2             N            no
(unordered ll)

binary search
(ordered array)  lg(N)        N            lg(N)           N/2          yes

BST              N            N            1.39*lg(N)      1.39*lg(N)   yes
```

```
2–3/red–black   2*lg(N)     2*lg(N)     ~1.00*lg(N)     ~1.00*lg(N) yes
```

- NOTE: see implementation doc for more traversal information

Balancing Binary Trees

- definition:
    - height balanced (balanced): (binary tree) if the difference in height of both subtrees of any node in the tree is either zero or one
    - perfectly balanced: balanced and all leaves are in (the last) one or two levels
- some algorithms keep balanced on insertion
- some rebalance by restructuring/reordering the tree and then constructing a balanced tree
- balancing from ordered array
- DSW algorithm
- AVL tree
- Red-black trees

Additional

- B-trees & Multiway Trees
    - B-trees
    - B*-trees
    - B+-trees
    - prefix B+-trees
    - K-d B-trees
    - bit-trees
    - R-trees
    - 2-4 trees
- van Emde Boas Trees