Pseudo code

I'm putting all pseudo code here, to reduce clogging up the main readme.

Red-black order statistic tree

```
root = null
size = 0

# Puts a new key/value mapping into the tree.
# Keep in mind each node + key may have multiple mappings.
put(key, value)
    size++
    new_node = {key: key, values: [value], color: red, size: 1}

    if root == null:
        root = new_node
        return

    parent = root

    while parent:
        parent.size++

        if key == parent.key:
            parent.values.append(value)

        if key < parent.key:
            if parent.left:
                parent = parent.left
            else:
                parent.left = new_node
        else:
            if parent.right:
                parent = parent.right
            else:
                parent.right = node

    node.parent = parent
    put_repair(node)

# Repairs the red-black tree balance after insertion.
put_repair(node)
    while node.parent.color == red:
        towards = left if node == node.parent.left else right
        against = opposite towards
        uncle = node.grandparent.against

        if uncle.color == red:
            node.parent.color = black
            uncle.color = black
            node.grandparent = red
            node = node.grandparent
            continue
```

```
        if node == node.parent.against:
            node = node.parent
            rotate_towards(node)

        node.parent.color = black
        node.grandparent.color = red
        rotate_against(node.grandparent)

# Finds the node associated with a key.
find(key)
    node = root
    while node:
        if key < node.key:
            node = node.left
        elif key > node.key:
            node = node.right
        else:
            return node
    return null

# Selects the list of values at a particular index in the tree.
select(index)
    node = root
    while node:
        size = node.left.size
        if index == size:
            return node.values

        if index < size:
            node = node.left
        else:
            index -= size + len(node.values)
            node = node.right
    return null

# Finds the rank of the key in this tree.
rank(key)
    node = find(key)

    if node == null:
        return null

    index = node.left.size

    while node.parent:
        if node.parent.left != node:
            if node.parent.left:
                index += node.parent.left.size
            index += len(node.parent.values)
        node = node.parent
    return index

# Deletes a key/value mapping from the tree.
delete(key, value)
    node = find(key)
    if node == null:
        return
```

```
    size--
    parent = node.parent
    while parent:
        parent.size--
        parent = parent.parent

    node.values.delete(value)

    if node.values.size > 0:
        node.size--
        return

    if node.left and node.right:
        node.size--
        successor = node.right
        while successor.left:
            successor = successor.left

        parent = successor.parent
        while parent != node:
            parent.size -= successor.size
            parent = parent.parent

        node.key = successor.key
        node.values = successor.values
        node = successor

    replacement = node.left == null ? node.left : node.right

    if replacement:
        replacement.parent = node.parent

        if node.parent == null:
            root = replacement
        elif node == node.parent.left:
            node.parent.left = replacement
        else:
            node.parent.right = replacement

        if node.color == black:
            delete_repair(replacement)
        return

    if node.parent == null:
        root = null
        return

    if node.color == black:
        delete_repair(node)

    if node.parent:
        if node == node.parent.left:
            node.parent.left = null
        elif node == node.parent.right:
            node.parent.right = null

# Repairs the red-black tree balance after deletion.
```

```
delete_repair(node)
    while node != root and node == black:
        towards = left if node == node.parent.left else right
        against = opposite towards
        sibling = node.parent.against

        if sibling.color == red:
            sibling.color = black
            node.parent.color = red
            rotate_towards(node.parent)
            sibling = node.parent.against

        if both siblings children color == black:
            sibling.color = red
            node = node.parent
            continue

        if node.against.color == black:
            sibling.color = red
            sibling.towards.color = black
            rotate_against(sibling)
            sibling = node.parent.against

        sibling.color = node.parent.color
        node.parent.color = black
        sibling.against = black
        rotate_towards(node.parent)
        node = root
    node.color = black

# Perform a left tree rotatation.
rotate_left(node)
    pivot = node.right
    node.right = pivot.left
    pivot.left.parent = node
    pivot_left_size = pivot.left.size
    pivot.parent = node.parent

    if node.parent == null:
        root = pivot
    elif node == node.parent.right:
        node.parent.left = pivot
    else:
        node.parent.right = pivot

    pivot.left = node
    node.size -= pivot.size
    pivot.size += node.size
    node.size += pivot_left_size
    root.parent = pivot

# Perform a right tree rotation.
rotate_right(node)
    # Same as rotate_left(node), but with left/right swapped.
```

Hash table

```
table = [] # Array of linked lists
size = 0

# Puts (replaces if already exists) the key/value mapping.
put(key, value)
    ensure_capacity(size + 1)
    hash_code = hash(key)
    bucket = table[hash_code]

    for element in bucket:
        if element.key == key:
            element.value = value
            return

    bucket.append({key: key, value: value})
    size++

# Gets the value associated with the given key.
get(key)
    hash_code = hash(key)
    bucket = table[hash_code]

    for element in bucket:
        if element.key == key:
            return element.value

    return null

# Deletes the value associated with the given key.
delete(key)
    hash_code = hash(key)
    bucket = table[hash_code]
    removed = bucket.remove(key)

    if removed:
        size--

# Increases the hash table size if too small to produce O(1) searches.
ensure_capacity(min_size)
    if table.size >= min_size:
        return

    new_size = (size * 3) / 2 + 1

    if new_size < min_size:
        new_size = min_size

    old_table = table
    table = [new_size]
    size = 0

    for key, value in old_table:
        put(key, value)
```

Doubly-linked list

```
left = null
right = null
size = 0

# Appends an item to the back of the list.
append_back(item)
    size++
    node = {item: item}
    if last != null:
        node.left = last
        last.right = node
    else:
        first = node
    last = node

# Appends an item to the front of the list.
append_front(item)
    size++
    node = {item: item}
    if first != null:
        node.right = first
        first.left = node
    else:
        last = node
    first = node

# Finds the node of an item.
find(item)
    node = left
    while node:
        if node.item == item:
            return node
        node = node.left
    return null

# Deletes an item from the list.
delete(item)
    node = find(item)

    if node == null:
        return

    size--
    node.left.right = node.right
    node.right.left = node.left

    if left == node:
        left = node.right

    if right == node:
        right = node.left
```

```
# Selects an item at the index of the list.
select(index)
    if index <= size / 2:
        node = first
        while index > 0:
            node = node.right
            index--
    else:
        node = last
        while index < size:
            node = node.left
            index++
    return node.item

# Convers the list to an array.
to_array()
    array = [size]
    node = left
    i = 0
    while node:
        array[i++] = node.item
        node = node.right
    return array
```

Pipe sort

```
runs = Tree() # Any multi-value order statistic tree would work here.
run = List() # Linked lists ensure value insertion is O(1).
previous = null consume_function= init  # The current state of the value consuming
pipeline.

# Initial consuming state.
init(value)
run.append(value)
previous = value
consume_function = single

# Consume state when only 1 element exists in the current run.
single(value)
if value >= previous:
    run.append(value)
    previous = value
    consume_function = front
else:
    run.append(value)
    previous = value
    consume_function = back

# Consume state when current run is ascending.
front(value)
if value >= previous:
    run.append(value)
    previous = value
else:
    runs.insert(run.size, run)
```

```
    run = List()
    run.append(value)
    previous = value
    consume_function = single

# Consume state when current run is descending.
back(value)
if value <= previous:
    run.append_front(value)
    previous = value
else:
    runs.insert(run.size, run)
    run = List()
    run.append(value)
    previous = value
    consume_function = single

# Merges all runs and returns as array.
sort()
runs.insert(run.size, run)
run = List()
consume_function = init

outer:
while runs.size > 1:
    new_runs = Tree()

    inner:
    for run_a, run_b in runs:
        if run_b == null:
            new_runs.insert(run_a.size, run_a)
            continue
            outer
        merged = merge(run_a, run_b)
        new_runs.insert(merged.size, merged)

    runs = new_runs

sorted_run = runs[0]

if sorted_run instance of linked list:
    sorted_run = sorted_run.to_array()

return sorted_run

# Merges two runs.
merge(run_a, run_b)
merged = [run_a.size + run_b.size]
index = 0
ia = 0
ib = 0

while ia < run_a.size and ib < run_b.size:
    if run_a[ia] > run_b[ib]:
        merged[index] = run_b[ib]
        ib++
    else:
```

```
        merged[index] = run_a[ia]
        ia++
    index++

while ia < run_a.size:
    merged[index] = run_a[ia]
    ia++

while ib < run_b.size:
    merged[index] = run_b[ib]
    ib++

return merged
```

Overview

```
load
- saved season stats
- saved tournament stats
- tournament types
- players by name hash tables
- ranking points hash table

if previous session saved
    load circuit and player progress

loop forever:
    request user command
    execute command

on tournament start command:
    if current running season is complete or non-existant:
        ask user for new season name
        update current running season
    loop each round until user requests to stop:
        ask user for the next track (gender) playing
        ask user for input method
        if input method is file:
            ask user for round files
            load player scores for round
        else input method is manual:
            calculate number of matches for this round
            update the track with the expected number of matches
        loop through each match:
            run match:
                loop until players are filled and valid:
                    request user input for players
                    validate players can play eachother based off seedings
                loop until scores are filled and valid:
                    request user input for scores
                    validate scores (no draws, 1 winner at max track score)
                adjust losers position in the scoreboard
            if match is complete:
                adjust winners position in the scoreboard
                print tournament scoreboard
```

```
            else:
                print players who won this round
        ask user if they would like to stop entering scores
    if season complete:
        print season scoreboard

on help command:
    print help message

on scoreboard command:
    if requested specific track:
        print track scoreboard
    if requested tournament:
        print tournament scoreboard
    if requested season:
        print season scoreboard

on circuit stats command:
    sort and find players with most winners
    print players with most wins and their win counts
    sort and find players with most losses
    print players with most losses and their loss counts

on score stats command:
    if no specific tournament or season defined:
        print table with all scores player has and their tallys
        return
    parse scores to find
    parse stats type to use (either tournament or seasonal)
    print number of times player recieved the given score in the selected stats

on wins or losses command:
    parse player and their stats (tournament or season) to use
    calculate percentage of wins for the stats
    print the players wins, losses and success percentage

on program exit command:
    save circuit and player progress
```