CS 1332 Final Exam

60% free response, diagram/coding of new stuff not on other exams
40% multiple choice of old stuff

Multiple choice (20, 2 points each)
- on most topics in list (everything before pattern matching)

Diagramming (3, 10 points each)
- graph algorithms
- text processing

Coding (3, 10 points each)
full code or fill-in-the-blank
- Pattern Matching
- Graphs
- Trees (not difficult, but interesting)

Topic List

- Big O
- Generics, interfaces, ADT
- Linked List
- java.lang.Iterable / java.util.Iterator
  - for-each loop, next( ), hasNext( )
  - Iterators vs using get( ) methods
- Stacks, Queues, Deques
  - Uses for a stack
  - Uses for a queue
  - Options to implement them
  - LIFO/FIFO meaning
- List ADTs
  - Array List
  - Linked List
  - Efficiencies of various operations
- Arrays
- Trees
  - Terminology (height, depth, complete, etc)
  - Traversals
  - Operations (add, remove, calculate height)
  - Order property
  - Binary trees
  - BST
  - AVL trees
    - Rotations
    - Adding/removing
    - Using, storing, updating, calculating heights and balance factors
    - Working with the typical node definition used to support an AVL tree.
  - 2-4 trees
- Heaps and Priority Queues
  - Min-heap, max-heap
  - Adding/removing min or max, build heap, respectively
  - Structure property
  - Order property
  - Complete tree terminology
  - Implementing a Priority Queue using a heap
- Maps

- Hash tables
  - Styles of hash tables based on collision resolution
  - External Chaining
  - Linear Probing
  - Quadratic Probing
  - Collision
  - Using a DEFUNCT marker/removed flag
  - Load factor
  - Hash code
  - Compression Function
- Skip Lists
- Sorting
  - Stable vs Non-Stable
  - In-place vs Out-of-place
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Cocktail Shaker Sort
  - Merge Sort
  - Quick Sort
  - Radix Sort
  - Number of comparisons done by each algorithm given an input array
- Text Processing
  - Brute Force
  - Boyer-Moore
  - KMP
  - Longest common subsequence
  - Number of comparisons done by each algorithm given a text and pattern.
- Graphs
  - Terminology
    - Directed vs Undirected
    - Weighted vs unweighted
  - Representations
    - Adjacency List
    - Adjacency matrix
  - Searching
    - Breadth-first Search
    - Depth-first Search
    - General Graph Search and how to use it for DFS/BFS
  - Single Source Shortest Path
    - Dijkstra's
    - Limitations of Dijkstra's
  - Minimum spanning trees
    - Prim's algorithm
    - Kruskal's algorithm

Not included:
- Splay trees
- Linear Probing with a cellar area
- Rabin-Karp
- Floyd-Warshall
- Regular Expressions (RegEx)

Data Structure	Average				Worst			
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)
Hash Table	—	O(1)	O(1)	O(1)	—	O(n)	O(n)	O(n)
BST	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)
AVL	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))
2-4 Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))
Heap	O(1)	O(n)	O(log(n))	O(log(n))	O(1)	O(n)	O(log(n))	O(log(n))

Sorting Algorithm	Average	Worst	In-place	Stable	Adaptive
Bubble/Cocktail Sort	$O(n^2)$	$O(n^2)$	✓	✓	✓
Insertion Sort	$O(n^2)$	$O(n^2)$	✓	✓	✓
Selection Sort	$O(n^2)$	$O(n^2)$	✓		
Quicksort	O(n log(n))	$O(n^2)$	✓		
LSD Radix Sort	O(kn)	O(kn)		✓	
MSD Radix Sort	O(kn)	O(kn)			
Mergesort	O(n log(n))	O(n log(n))		✓	
Quick Select	O(n)	$O(n^2)$		✓	

Algorithm	Average	Worst
kmp	O(m + n)	O(m + n)
boyermoore	O(m + n)	O(mn)
Prim's	O(ElogV)	
Dijkstra's	O(ElogV)	
Kruskal's	O(ElogV)	
BFS	O(V + E)	
DFS	O(V + E)	

Method	Average
heapify	O(log(n)
buildHeap	O(n)
Tree traversals	O(n)

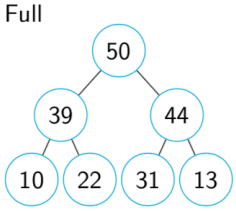
Stacks
- LIFO - last in first out
- Uses:
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in many programming languages
  - Auxiliary data structure for algorithms
  - Component of other data structures
  - Check to make sure a parenthesis has a matching parentheses ( ) { } [ ]
- Can be backed with a LinkedList or an Array

Queues
- FIFO - first in first out
- Uses:
  - Waiting lists
  - Access to shared resources (e.g. printer)
  - Multithreading
  - Auxiliary data structure for algorithms
  - Component of other data structures
  - Round Robin scheduler for a CPU to determine what process to do if multiple are running with three steps:
    1. process = processes.dequeue()
    2. Do work on process
    3. processes.enqueue(process)
  - Can be backed with a LinkedList or an Array

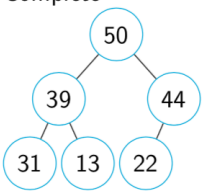
List ADTs
- LinkedList
- ArrayList

Trees
- Height — maximum number of nodes needed to be traversed to reach a leaf node. For a binary tree: $h \leq i$
  - The height of a leaf node is 0
  - height(node) = max(height(node.Left), height(node.Right)) + 1
- Depth — number of nodes needed to be traversed to reach the root node.
  - The depth of the root node is 0
- Root — node without a parent, where you enter the tree
- Internal node — node with at least one child
- External (leaf) node — node with no children. For a binary tree: $e \leq 2^h$
- Full — every node, except the leaves, has two children.
- Complete — all levels, except the last, are full, and the leaves are filled left to right.
- Balanced — the heights of sibling nodes can differ by at most by 1.

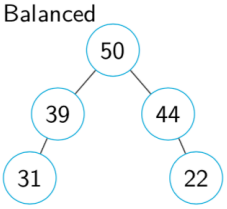
Full



Complete

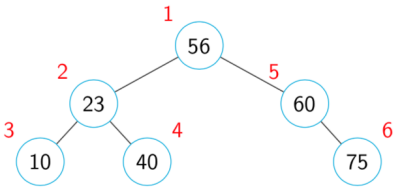


Balanced

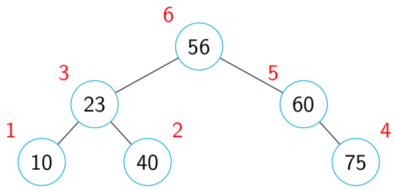


Tree Traversals

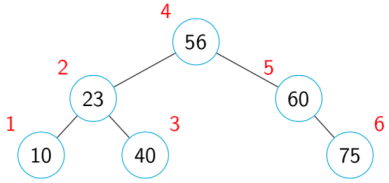
- Preorder — a node is visited first, then the child nodes are visited (from left to right).
  - Pattern: Parent-Left-Right
- Postorder — the child nodes are visited first (from left to right), then the node itself is visited.
  - Pattern: Left-Right-Parent
- Inorder — the left subtree of the node is visited first, then the node itself is visited, then the right subtree is visited.
  - This traversal applies only to BSTs and visits the data items in the tree in ascending order.
  - Pattern: Left-Parent-Right



Preorder



Postorder



Inorder

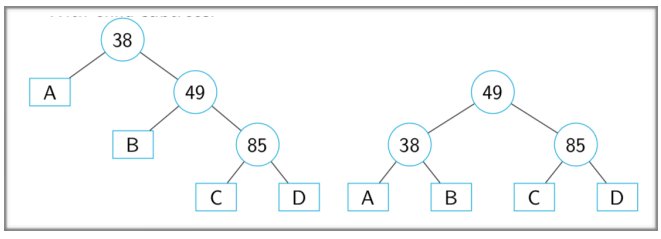
AVLs

- A BST that self balances to ensure searching is O(log(n)). Does this using "rotations"
- Balance Factor (bf) — describes how balanced the subtree of that node is and, if it is not perfectly balanced, which side of the subtree (left side or right side) is “heavier”.
  - Calculated by subtracting the height of the left node by the height of the right node.
    - node.getRight( ).height.( ) — node.getLeft( ).height.( )
    - Remember the height of a leaf node is 0, even if it is on a higher "level" than another node
  - A subtree is considered "balanced" if the bf is between -1 and 1.

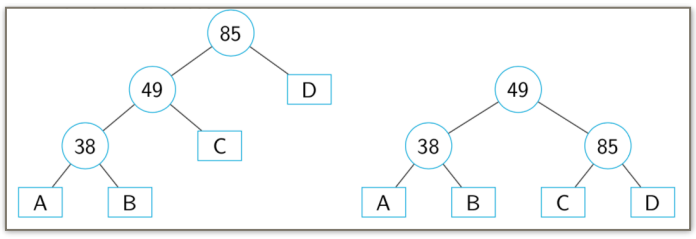
AVL Rotations

Single:

- Left Rotation — occurs when you have a node whose balance factor is -2, and its right child’s has a balance factor of 0 or -1 (bf = -2 & $-1 \leqq bf_R \leqq 0$).
  - When a left rotation happens, the top node becomes the left child of the middle node.
- Right Rotation — occurs when you have a node whose balance factor is 2, and its left child’s has a balance factor of 0 or 1 (bf = 2 & $0 \leqq bf_L \leqq 1$).
  - When a right rotation happens, the top node becomes the right child of the middle node.



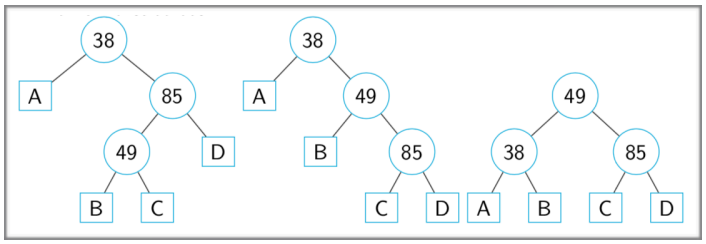
Left Rotation



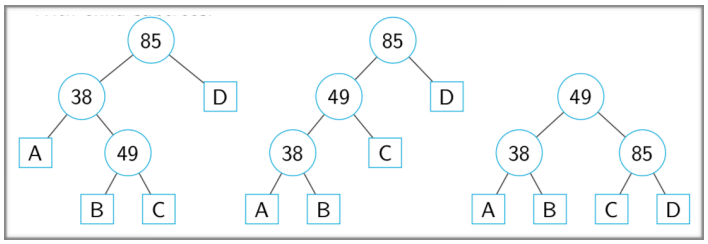
Right Rotation

Double:

- Left-Right Rotation — occurs when you have a node whose balance factor is 2, and its left child has a balance factor of -1 (bf = 2 & bf_L = -1).
  - When a left-right rotation happens, the top node becomes the right child of the bottom node, and the middle node becomes the left child of the bottom node.
- Right-Left Rotation — occurs when you have a node whose balance factor is -2, and its right child has a balance factor of 1. (bf = -2 & bf_R = 1)
  - When a right-left rotation happens, the top node becomes the left child of the bottom node, and the middle node becomes the right child of the bottom node.

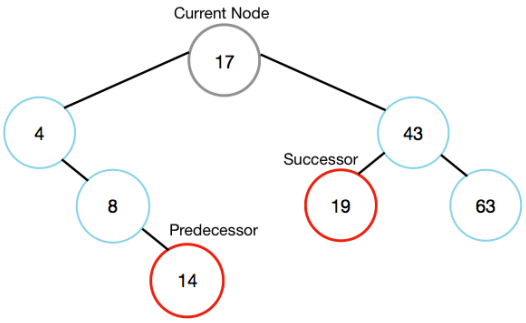


Left-Right Rotation



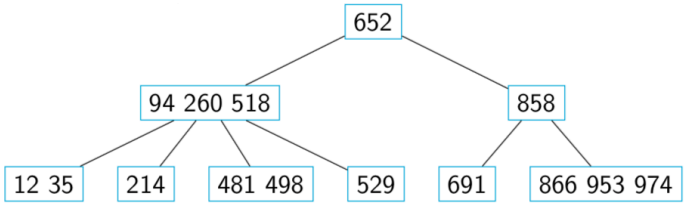
Right-Left Rotation

- Adding/Removing — same as BST, but you to update the bf and height of the node and all ancestors and do any necessary rotations.
  - When removing you might have to promote a successor or predecessor node to where the removed node was
    - Successor — node that is greater than your current node, but less than all other nodes
      - Obtained by getting the left child, than the deepest right child
    - Predecessor — node that is less than your current node, but greater than all other nodes
      - Obtained by getting the right child, than the deepest left child



2-4 Trees

- Similar to BSTs and AVLs, but each node can hold multiple data items
- A node has n data items and must of either 0 or n + 1 children
  - ex. if a node has 2 data items, it has 0 or 3 children.
- The depth of all leaf nodes must be the same
  - This means 2-4 trees are always balanced
- Data in nodes are arranged from least to greatest
- Nodes are positioned in the same as AVLs and BSTs, with smaller nodes are the left and greater nodes on the right.



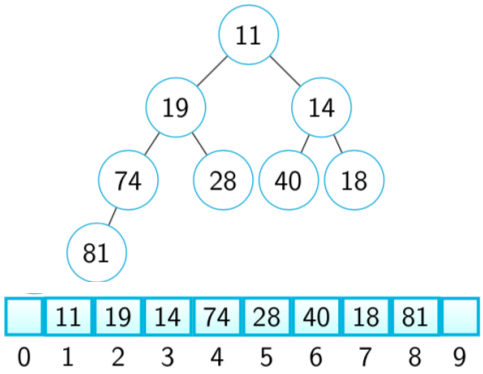
2-4 Tree pseudocode

```
procedure Search(data, node)
   if node is not valid then
      return FALSE
   else
      for i ← 1, n do
         if data = node.data[i] then
            return TRUE
         else if data < node.data[i] then
            child ← ith child node
            return Search(data, child) end if
      end for
      child ← last child node
      return Search(data, child)
   end if
end procedure
```

```
procedure Add(data, node)
   for i ← 1, n do
      if data < node.data[i] then
         if node has any children then
            child ← ith child node
            Add(data, child)
            if child has too many items then
               Break child up into two nodes,
               promoting the middle item
            end if
            return
         else
            Add data into this node in the ith
            position
            return
         end if
      end if
   end for
   if node has any children then
      child ← last child node
      Add(data, child )
      if child has too many items then
         Break child up into two nodes,
         promoting the middle
   end if
   else
      Add data into the last slot of this node
   end if
end procedure
```

Heaps

- Can be backed easily with a List, but is very inefficient. Instead we use a binary tree, or an Array.
- MinHeap — root is smallest data
- MaxHeap — root is largest data
- Order:
  - The parent node is always larger (in the case of a maxheap) or smaller (in the case of a minheap) than its children.
  - There is no relationship between the children.
- Shape:
  - A heap is always complete.
- Using an Array to back a heap:
  - The "root" of the heap would be in index 1 of the array. (Index 0 is skipped to make the math easier.)
  - The left child of an item in index *i* would then be at index $2i$, and the right child would be at index $2i + 1$.
  - The parent of an item in index *i* would be at $i/2$ (assuming you are doing integer division).
  - Priority Queue
    - A priority queue can be easily implemented using a heap
    - Uses:
      - Printer jobs
      - CPU schedulers
      - Flight boarding
      - Bandwidth management in routers
      - Auxiliary data structure in other algorithms.



Maps

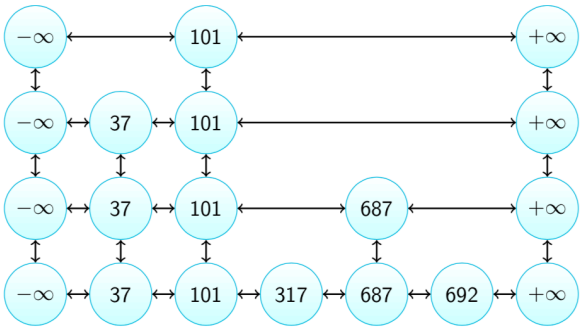
- Map — data structure where, for each key, there is a single value associated with it. In other words, it is a mapping from key to value.
- Can be easily implemented using a LinkedList or an Array, but is not very efficient (results in O(n) for searching). To make this O(1) use a HashMap
- HashMaps are backed by Arrays
- To calculate the index for the data in the Array use a % b, where a is the hashCode and b is the length of the Array
- Load factor — # of key-value pairs divided by the length of the array
  - this is used to determine when to grow the array (it is usually .75)

Collision Resolution

- Linear Probing — if a key-value pair is being added into index i, but there is already another key-value pair at index i, and the keys are different, then index i + 1 is checked, then i + 2, then i + 3, etc.
  - Searching
    - If the key is at that index, then you've found the key, and you can get the value.
    - If the key is not at that index, then go on to the next index (loop back around if at the end of the array).
    - Continue until you find the key, you reach an index with *null* at that index, or you go through the entire array.
  - Removing
    - When we remove we cannot just set the index to *null* as it breaks searching
    - Instead, mark the index with "DELETED" or some other marker to show something was here and to continue searching
  - Adding
    - Because of how we remove we cannot just add a key-value pair to the first available index always
    - If the ideal index is *null* add it there
    - If when probing the array to add, you find a DELETED marker, save that index, then continue searching. If you find that key in the array, update that key with the new value. If you reach the end of the array or find *null* then that key is not in the index and you can add the key-value pair to the DELETED marker you saved earlier
- Quadratic Probing — same as linear, but check i + 1^2, then i + 2^2, then i + 3^2, etc.
  - Since it is possible to probe the array w/o checking all the indices, quadratic probing requires an implementation of forced resizing
    - In general, if the backing Array is length *n*, you resize after probing *n* times
- External Chaining — there is essentially a linked list in each index of the array.
  - If there is a collision, just add the new key-value pair to the LinkedList at that index
  - Searching
    - First determine the ideal key
    - Then search the LinkedList
  - Removing
    - Determine the key, then remove that key-value pair from the LinkedList

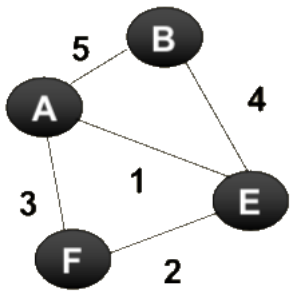
Skip List

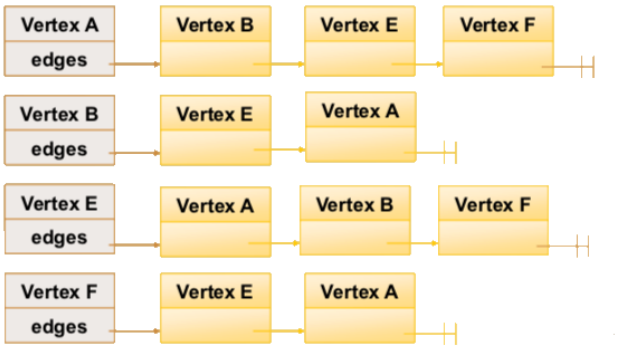
- Like a LinkedList, but with "levels" to making searching quicker
- When adding a node, there is a 50% chance the node is promoted to a top level, so ideally every level has half the nodes the level below it has.
- Every node is on the first level
- Each level has -∞ (or Integer.MIN_VALUE) node. These are called phantom nodes, because they contain no real data



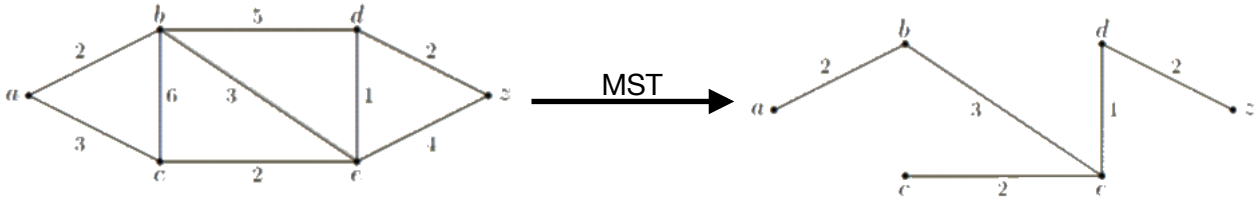
Graphs

- Graph — a set of vertices and a collection of edges that each connect a pair of vertices.
- Order — number of vertices
- Size — number of edges
- Directed — edged have direction, meaning you can only cross a vertex one direction
- Undirected — edges flow in both direction
- Weighted — edges have a cost associated with them
- Adjacency List — A list of maps, where each map's key is a vertex and the value is a list of edges that connect to that vertex



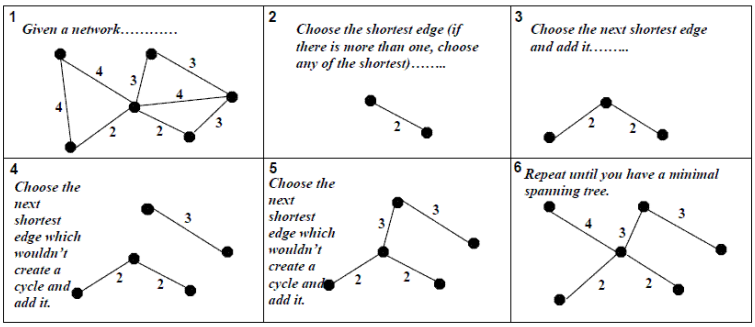


- MST (minimum spanning tree) — a subtree that connects all vertices in a graph and has a minimum total weight

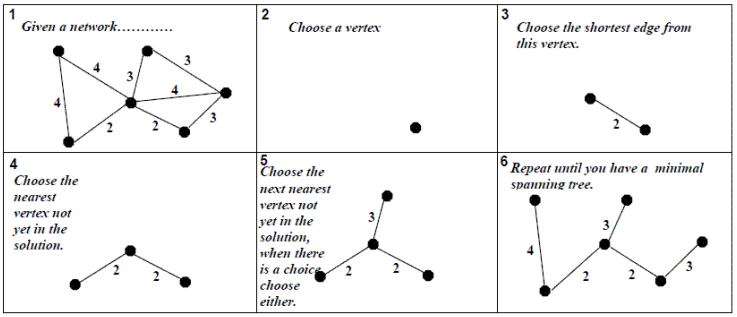


- Dijkstra's — algorithm to find shortest path from one vertex to all other vertices
  - Limitations: cannot handle negative edges
- Prim's — algorithm to find an MST of a graph, given a Graph and a starting Node
  - O(ElogV), but can be improved with Fibonacci heaps to O(E + logV)
  - Performs better than Kruskal's with dense graphs (more edges than vertices)
- Kruskal's — algorithm to find an MST of a graph, given a Graph and a starting Node
  - O(ElogV)
  - Performs better than Prim's with sparse graphs (more vertices than edges)

Kruskal's Algorithm



Prim's Algorithm



Sorting Algorithms

- In-place — algorithm doesn't copy the data into another data structure to sort it
- Stable — original order of duplicates is preserved
- Bubble Sort
  - Compare the first two items. If they are in the wrong order, then swap them; if not, then keep them as-is. Then increment index
  - Repeat the above step until you get to the end of the array. This is one iteration
  - Continue to repeat, but stop early by one more each time, since the last spot is now sorted
  - If you do not make any swaps during an iteration, then this means that the array is sorted, and you can terminate early.
- Insertion Sort
  - Assume the first item is sorted. Then, take the second item, and "slide" it to the left so that it is correctly placed in the sorted portion of the array. The first two items are now considered sorted. This is one iteration
  - Repeat with the third item and so on until the entire array is sorted.
  - Unlike bubble sort, a fixed number of iterations are done for insertion sort.
- Selection Sort
  - Search the entire array for the smallest item. Swap that item with the first item.
  - Then, search the entire array (excluding the first item) for the next smallest item. Swap that item with the second item.
  - Repeat until the entire array is sorted.
- Merge Sort
  - Divide the array into 2 equal parts.
    - If there is an odd number of elements, the middle element will go to either the first part or the second part.
  - Then, perform merge sort on each half of the array. After this is done, each part should be sorted.
  - Finally, merge the two parts together.
    - To do this, have a marker on the first item in each part. Take the smaller of the two items and add that into the larger (merged) array, and move that marker forward. Repeat until all of the items have been added into the merged array.
- Quick Sort
  - First, choose an item at random to be the pivot, and swap with the first item.
  - Have a left "marker" that starts with the second item (the item after the pivot), and a right "marker" that starts at the last item.
  - If the item pointed to by the left marker is smaller than the pivot, move the marker one item to the right. Repeat this step until the marker points to an item that is larger than the pivot or goes beyond the right marker (they cross over).
    - If the item and the pivot are equal, then either can be done.
  - After the markers cross over, swap the pivot with the right marker (note that the right marker is now to the left of the left marker).
  - The pivot is now in the right place within the final sorted array. All items to the left of the pivot (if there are any) are smaller than the pivot, and all items to the right of the pivot (if there are any) are larger than the pivot. Perform Quick Sort on the smaller items and on the larger items.

- Radix Sort
  - Radix Sort is different from previous sorts as no actual comparisons are done
  - It can only be done on numbers (with some base)
  - Two variants of radix sort:
    - LSD — starts by looking at the least significant digit and works upwards.
    - MSD —starts by looking at the most significant and works downwards.
- LSD Radix Sort
  - Create 10 “buckets” (if being done in base 10), and label them from 0 to 9.
    - Treat each bucket as a queue.
  - For each number, take the first digit (least significant digit), and add the number into the appropriate bucket.
    - For example, if the number is 27, then the first digit is 7, and add the number into bucket 7
  - After all of the numbers have been added, remove all of the numbers one at a time, starting from bucket 0.
  - Repeat this process for each digit in the longest number
    - For example, if the longest number has 7 digits, this process is done 7 times.

Iterable/Iterator

- Iterators are a class/object that allows you to traverse through the data structure
  - the *Iterator* interface contains three methods: hasNext( ), next( ), and remove( )
    - remove( ) is not necessary to implement
- *Iterable* is an interface with one method:
  - iterator( )
    - returns an instance of the *Iterator* interface the data structure has implemented
- Two importanta methods:
  - hasNext( )
    - returns true if calling next( ) returns something
  - next( )
    - returns the next item in the data structure
- Iterators are used in for-each loops (also known as enhanced for loops)
  - Given the data structure, Java gets the iterator object and gives back to you each item.
  - This requires that the Object in the for-each loop implements *Iterable*.

Directly using Iterators:

```
Iterator<Integer> example = linkedList.iterator();
while (example.hasNext()) {
    Integer data = example.next();
    System.out.println("Data: " + data);
}
```

for-each loop:

```
for (Integer data : linkedList) {
    System.out.println("Data: " + data);
}
```

Pattern Matching Coding

KMP

```
public static int[ ] buildFailureTable(CharSequence pattern) {
    int[ ] table = new int[pattern.length( )];
    if (pattern.length( ) == 0) {
        return table;
    }
    int i = 0;
    int j = 1;
    table[0] = 0;
    while (j < pattern.length( )) {
        if (pattern.charAt(i).equals(pattern.charAt(j))) {
            i++;
            table[j] = i;
            j++;
        } else if (i == 0) {
            table[j] = 0;
            j++;
        } else {
            i = table[i - 1];
        }
    }
    return table;
}

public static List<Integer> kmp(CharSequence pattern, CharSequence text) {
    List<Integer> matches = new ArrayList<>( );
    if (pattern.length( ) > text.length( )) {
        return matches;
    }
    int[ ] failureTable = buildFailureTable(pattern);
    int i = 0;
    int j = 0;
    while (i <= text.length( ) - pattern.length( )) {
        while (j < pattern.length( ) && text.charAt(i + j).equals(pattern.charAt(j))) {
            j++;
        }
        if (j == 0) {
            i++;
        } else {
            if (j == pattern.length( )) {
                matches.add(i);
            }
            int nextAlignment = failureTable[j - 1];
            i = i + j - nextAlignment;
            j = nextAlignment;
        }
    }
    return matches;
}
```

Boyer-Moore

```
public static Map<Character, Integer> buildLastTable(CharSequence pattern) {
    HashMap<Character, Integer> table = new HashMap<>( );
    for (int i = 0; i < pattern.length( ); i++) {
        table.put(pattern.charAt(i), i);
    }
    return table;
}

public static List<Integer> boyerMoore(CharSequence pattern, CharSequence text) {
    List<Integer> list = new ArrayList<>( );
    if (pattern.length( ) > text.length( )) {
        return list;
    }
    Map<Character, Integer> lasttable = buildLastTable(pattern);
    int i = 0;
    int j;
    int shiftindex;
    while (i <= text.length( ) - pattern.length( )) {
        j = pattern.length( ) - 1;
        Character temp = text.charAt(i + j);
        while (j >= 0 && temp.equals(pattern.charAt(j))) {
            j--;
            if (j >= 0) {
                temp = text.charAt(i + j);
            }
        }
        if (j == -1) {
            list.add(i);
            i++;
        } else {
            if (lasttable.containsKey(temp)) {
                shiftindex = lasttable.get(temp);
            } else {
                shiftindex = -1;
            }
            if (shiftindex < j) {
                i = i + (j - shiftindex);
            } else {
                i++;
            }
        }
    }
    return list;
}
```

Graph Coding

Dijkstra's

```
public static <T> Map<Vertex<T>, Integer> dijkstras(Vertex<T> start, Graph<T>
graph) {
    Set<Vertex<T>> vertices = graph.getVertices( );
    Map<Vertex<T>, List<Edge<T>>> adjList = graph.getAdjList( );
    Map<Vertex<T>, Integer> result = new HashMap<>( );

    for (Vertex<T> vertex : vertices) {
        if (vertex.equals(start)) {
            result.put(vertex, 0);
        } else {
            result.put(vertex, Integer.MAX_VALUE);
        }
    }

    Queue<Edge<T>> distQ = new PriorityQueue<>( );
    distQ.add(new Edge<>(start, start, 0));

    while (!distQ.isEmpty( )) {
        Edge<T> vertexWithDist = distQ.remove( );
        Vertex<T> currU = vertexWithDist.getV( );

        for (Edge<T> currEdge : adjList.get(currU)) {
            Vertex<T> currV = currEdge.getV( );
            if (result.get(currV)
                    > currEdge.getWeight( ) + vertexWithDist.getWeight( )) {
                result.put(currV,
                        currEdge.getWeight( ) + vertexWithDist.getWeight( ));
                distQ.add(new Edge<T>(start, currV, currEdge.getWeight( )
                            + vertexWithDist.getWeight( )));
            }
        }
    }
    return result;
}
```

Prims

```
    public static <T> Set<Edge<T>> prims(Vertex<T> start, Graph<T> graph) {
        Set<Vertex<T>> vertices = graph.getVertices( );
        Set<Edge<T>> mst = new HashSet<>( );
        Map<Vertex<T>, List<Edge<T>>> adjList = graph.getAdjList( );
        Queue<Edge<T>> edgePrioQ = new PriorityQueue<>( );
        Set<Vertex<T>> resultVertices = new HashSet<>( );

        edgePrioQ.addAll(adjList.get(start));
        while (!edgePrioQ.isEmpty( )) {
            Edge<T> minEdge = edgePrioQ.remove( );
            if (!mst.contains(minEdge)
                    && !resultVertices.contains(minEdge.getV( ))) {
                mst.add(new Edge<>(minEdge.getU( ), minEdge.getV( ),
                        minEdge.getWeight( )));
                mst.add(new Edge<>(minEdge.getV( ), minEdge.getU( ),
                        minEdge.getWeight( )));
                edgePrioQ.addAll(adjList.get(minEdge.getV( )));
                resultVertices.add(minEdge.getU( ));
                resultVertices.add(minEdge.getV( ));
            }
        }

        return mst.size( ) < 2 * (graph.getVertices( ).size( ) - 1) ? null : mst;
    }
}
```

Breadth First

```
public static <T> List<Vertex<T>> breadthFirstSearch(Vertex<T> start,
                                                     Graph<T> graph) {

    List<Vertex<T>> result = new ArrayList<>( );
    Queue<Vertex<T>> queue = new LinkedList<>( );
    Set<Vertex<T>> visited = new HashSet<>( );
    Map<Vertex<T>, List<Edge<T>>> adjList = graph.getAdjList( );

    queue.add(start);
    visited.add(start);
    while (!queue.isEmpty( )) {
        Vertex<T> currVertex = queue.remove( );
        result.add(currVertex);
        for (Edge<T> edge : adjList.get(currVertex)) {
            if (!visited.contains(edge.getV( ))) {
                queue.add(edge.getV( ));
                visited.add(edge.getV( ));
            }
        }
    }
    return result;
}
```

Depth First

```
public static <T> List<Vertex<T>> depthFirstSearch(Vertex<T> start,
                                        Graph<T> graph) {

    List<Vertex<T>> result = new ArrayList<>( );
    Set<Vertex<T>> visited = new HashSet<>( );
    Map<Vertex<T>, List<Edge<T>>> adjList = graph.getAdjList( );

    realDepthFirst(start, visited, adjList, result);

    return result;
}

private static <T> void realDepthFirst(Vertex<T> currVertex, Set<Vertex<T>> visited,
                    Map<Vertex<T>, List<Edge<T>>> adjList, List<Vertex<T>> result) {
    result.add(currVertex);
    visited.add(currVertex);

    for (Edge<T> edge : adjList.get(currVertex)) {
        if (!visited.contains(edge.getV( ))) {
            realDepthFirst(edge.getV( ), visited, adjList, result);
        }
    }
}
```