

Red-Black Tree Algorithms

Rules

- Nodes either **red** or **black**
 - If red, represented by "0", if black, "1"
- Root and leaf (nil) nodes are always **black**
- If a node is **red**, both of its children must be **black**
- The black height of all leaf (nil) nodes must be the same
 - No nodes can have one **black** child and one **nil** child

Properties

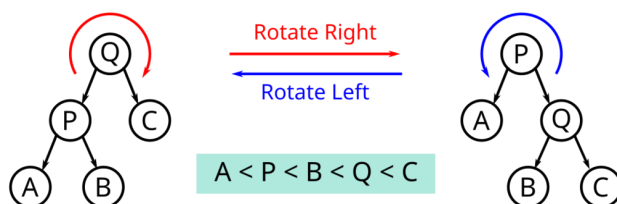
- Black height: the number of **black** nodes on any path from a given node (not including it) to a leaf
- The longest path from the root to nil is never more than double the shortest path
 - Shortest path: will have all **black** nodes
 - Longest path: will have alternating **red/black** nodes
 - **Balanced**: The left and right subtrees of every node differ in height by no more than 1.
 - **Full**: All nodes except leaf nodes have either 0 or 2 children
 - **Complete**:
 - All nodes except for the level before the last must have 2 children.
 - All nodes in the last level are as far left as possible.

Space complexity

- Size: $O(n)$
- Height: $O(\log(n))$

Rotate

To determine which way to rotate, think about it as the tree shifting more to the side (left/right) specified



Time complexity: $O(1)$

Insert

Strategy:

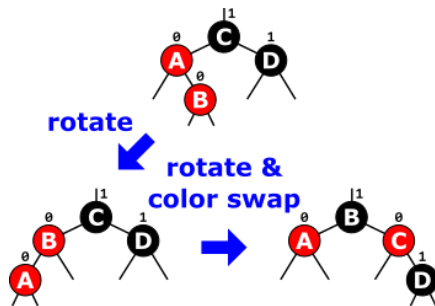
1. Insert key using BST insert algorithm
2. Color red
3. Check RBT properties and repair (if necessary)

Properties to check:

- If inserted the tree's root node, the node must be black
- No **red** children of **red** nodes

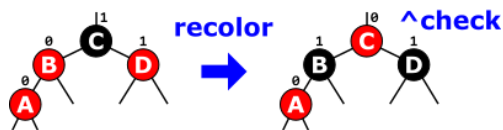
If property violated, to repair:

- If uncle **black** (or null):



The first step should be skipped if the tree already looks like the second step

- If uncle **red**:



After recoloring, **check** that the new tree doesn't violate the RBT properties

Time complexity: $O(\log(n))$ or $O(h)$

Remove

Strategy:

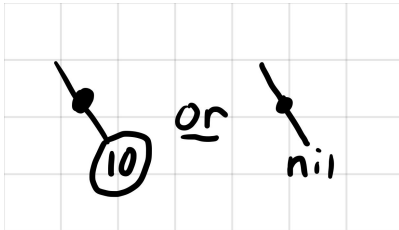
1. Delete key using BST delete algorithm:
 - a. Swap node to be deleted and its child
 - i. If the node has two children, use the rightmost child of the left subtree (in order predecessor)
 - b. Remove the node that was to be deleted from the tree
2. Check RBT properties and repair (if necessary)

Properties to check:

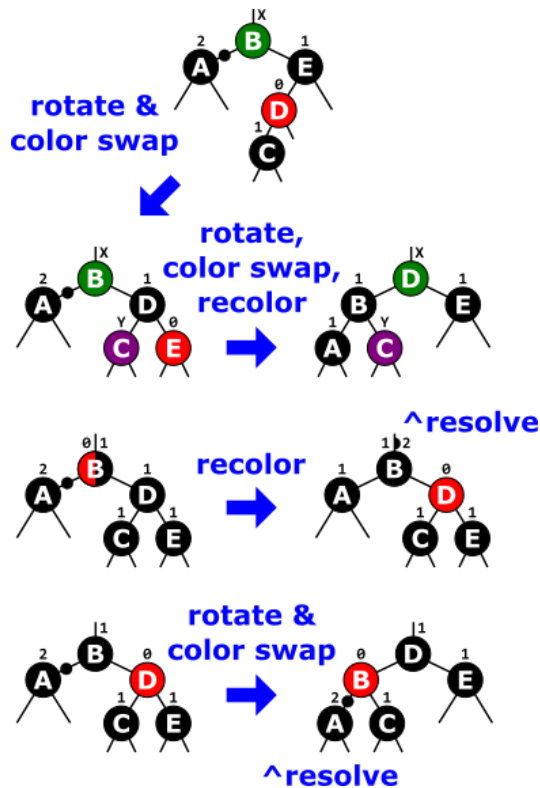
- No **red** children of **red** nodes

If property violated, to repair:

- If removed **black** and replacement **red**:
Turn node **black**
- If removed **black** and replacement NOT **red**:
Add “double black” on the path missing a **black** node



Follow the cases below to resolve double black:



- Multiple may have to be used in a recursive fashion
- Deletion not done until all “double blacks” removed and there are no violations of RBT properties
 - EXCEPTION: If “double black” ends up on root node, can just remove it
- When recoloring, minimize the number of nodes whose colors must be changed

Time complexity: $O(\log(n))$ or $O(h)$

Search

Same as BST search

Time complexity: $O(\log(n))$ or $O(h)$

x

RBT Visualization Tool

<https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>

Counting and Radix Sort

Stable vs Unstable

A stable sort is one which preserves the original order of the input set when keys equal

Counting Sort

Strategy:

1. Calculate the range of the values in the list to be sorted
2. Declare the array "counts" with the length equaling the number of integers in the range
 - a. Count how many of each value are in the list to be sorted
 - b. Store this count in the "counts" index matching the value
3. Declare array "endpos"
 - a. In the "endpos" index matching each value to be sorted, store the last index in where the items with the value will be stored in the final array
 - b. If no values match the item index, set it to the value of the pre. index
4. Declare array "index output"
 - a. Starting at end of the list to be sorted, get value
 - i. Lookup in "endpos" index at which insert value in "output"
 - ii. Insert value in "index output" at that index
 - iii. Decrement "endpos"
 - iv. Continue backward through the list to be sorted

What if the range is too big? (double digits): Radix sort!!!

Time complexity: $O(n)$ (hence, "linear sort")

Counting Sort Example

input: 8, 8, 9, 0, 1, 3, 9, 0, 3, 5, 3

counts									
0	1	2	3	4	5	6	7	8	9

2	1	0	3	0	1	0	0	2	2
---	---	---	---	---	---	---	---	---	---

endpos									
0	1	2	3	4	5	6	7	8	9
1	2	2	5	5	6	6	6	8	10

output										
0	1	2	3	4	5	6	7	8	9	10
0	0	1	3	3	3	5	8	8	9	9

Radix Sort

For when the range is too big for counting sort

Strategy:

1. Iteratively run counting sort on each digit, starting with the least significant
 - a. Only run counting sort on one digit at a time, but organize the full multiple-digit value in "output"

Time complexity: ALSO $O(n)$

Merge sort and quick sort?

Hash Tables

Why Use Hash Tables?

Their speed to perform insertion, deletion, and search operations. Hash tables can do them all in (on average) constant time ($O(1)$)

Hash Table Properties

- Hash table: an array containing key, value pairs
- Table size: current capacity (array length)
- Load factor (LF):

$$LF = \frac{(\text{num of stored nodes})}{(\text{array cap})}$$

Hash Functions

- Hash functions turn the key of a key/value pair into a hash index to insert the pair at in the array
- Properties of a good hash function:
 - Deterministic (same key = same hash index)
 - Should achieve uniform key distribution
 - Should minimize collisions
 - Should be fast & easy to compute
- In java, can use `Math.abs(obj.hashCode()) % arrayLen` for easy hash function
- A perfect hash function maps every key to a unique hash index

Hash Collisions: Open Addressing

If a node is already at the hash index of the new node, look for the next “open address”

Strategy for Linear Probing insertion:

1. Try to insert the node at the index returned by the hash function
2. If the index is occupied, look at the next index in array
 - a. Can insert node if the index is occupied by a “dummy node”
3. If a key/value pair is also at the next index, repeat

Strategy for Quadratic Probing insertion:

1. Declare the variable “I”, inc. for every iteration of trying to insert the node
2. Try to insert the node at the index returned by the hash function
3. If the index is occupied, look at the index given by

$$IND. = (START\ IND.) + I^2$$
 Sometimes may also have a linear term in the eq.
4. If a key/value pair is also at the next index, repeat

Strategy for Double Hashing insertion:

1. Declare the variable “I”, inc. for every iteration of trying to insert the node
2. Create a second, independent hash function “hash2(key)”
3. Try to insert the node at the index returned by the hash function
4. If the index is occupied, look at the index given by

$$IND. = (START\ IND.) + I * hash2(key)$$
5. If a key/value pair is also at the next index, repeat

Strategy for removing:

1. Follow the same strategy as the insertion strategy to find the node to be removed
 - a. Keep searching for the node if encounter a “dummy node”
2. When removing a node, insert a dummy node so that any nodes inserted after the removed node b/c of a hash collision can still be found

Hash Collisions: Chaining

Store a linked list at each hash index so if hash collisions occur, no further action is necessary

Insert and delete time complexity: $O(n)$

Rehashing

When the load factor exceeds a certain threshold (ex. 70%), need to expand the hash table

Strategy:

1. Create a new empty hash table with double the size
2. Iterate over nodes in the old hash table, and add each one to the new hash table
 - a. Skip any "dummy nodes" when re-hashing

Time Complexity

	Worst Case	Average Case	Best Case
Insert	$O(n)$	$O(1)$	$O(1)$
Lookup	$O(n)$	$O(1)$	$O(1)$
Remove	$O(n)$	$O(1)$	$O(1)$
Re-hash	$O(n)$, amortized over several inserts $O(1)$		

Other Topics (pre. midterm)

Basic Commands

- `tar -xzf <source>` - extracts contents of source to current directory
- `cd /path` - changes the working directory to the given path
- `pwd` - prints the working directory
- `ls -a` - Shows hidden files with prefix (.) like .gitignore
- `~` - refers to user's home directory

Git Commands

- `git add *.java` - adds all of the changes in java files to git staging
- `git commit -m "the message"` - commits all changes in staging
- `git push` - pushes all commits to remote directory
- `git log` - shows change history
- `git merge <source>` - takes the source and combines with checked out repository

SSH Command

- `ssh user@1.1.1.1` - connects to the remote computer

SCP Command

- `scp user@1.1.1.1:~/message.txt ./` - copies the file from the remote to the local computer
- `scp ./message.txt user@1.1.1.1:~/` - copies the file from the local to the remote computer

Vim/Emacs Commands

Movement	<code>h</code> (left) <code>j</code> (down) <code>k</code> (up) <code>l</code> (right)
Quit	<code>:q!</code>
Delete at cursor	<code>x</code>
Insert Mode	<code>i</code>
Copy/Paste	in visual mode (<code>v</code>) use <code>y</code> to copy <code>p</code> to paste
Save	<code>:w</code>
Save and quit	<code>:wq</code>

Emacs

Movement	Arrow keys
Quit	<code>Ctrl-x Ctrl-c</code>
Save	<code>Ctrl-x Ctrl-s</code>

Makefile

Dependency rules have the following structure:

`<target> : <dependencies>`

`<tab><command>`

The rule will be evaluated if any of the files or targets are newer than the result of the rule evaluation.

Example: assume only Dependency1 and Dependency2 exists, and "Dependencies" are files

Target 1: Dependency1 Dependency2

Dependency1: Dependency3

Dependency2:

Dependency3:

How to “walk” a Makefile:

1. Chase the dependencies to its (root) i.e. until your target has no dependencies
 - a. If Target has multiple dependencies, start from left to right
 - b. **E.g.** target1 -> Dependencies1 -> Dependencies3 (root)
2. Evaluate rules from “root” backwards
 - a. Two evaluation “ideas” -> rule executes if either is “no”
 - i. Does the target exist?
 - ii. Is Target (file) newer than dependencies (if no dependencies, then it is newer)
3. Repeated step 1 and 2 as you walk backwards from dependency tree
 - a. **Walkthrough of example above**
 - i. **(step1)** chase dependency: target1 -> Dependency1 -> Dependencies3 (root)
 - ii. **(step2)** evaluate:
 1. Dependency3 doesn't exist → rule runs
 2. Dependency1 exists, but is not newer than Dependency3 (we just ran) → rule runs
 3. Target1 doesn't exist (which should run), but we haven't evaluated all its dependencies
 - iii. **(step1)** chase dependency: target1 -> Dependency2 (root)
 - iv. **(step2)** evaluate
 1. Dependency2 exists and has no dependencies (i.e. assume newer) → doesn't run
 2. Target1 is now fully evaluated and doesn't exist → rule runs

Test Writing

Black/opaque box testing:

- Focuses on software's external attributes and behavior

White/clear box testing:

- Tests the software by using the knowledge of internal data structures, physical logic flow, and architecture
- Testing from the developer's point of view

Unit Testing is a kind of white box testing, whereas Integration Testing is a kind of black-box testing

JUnit Tests

When does a JUnit test pass vs. fail?

- If the method returns successfully, the test passes
- If an exception occurs, the test fails

Commands to know

- `@BeforeAll` - executes once before all test methods
- `@BeforeEach` - executes before each test method
- `assertEquals` - tests if first parameter is equal to the second
- `assertAll` - is just a culmination of assertions

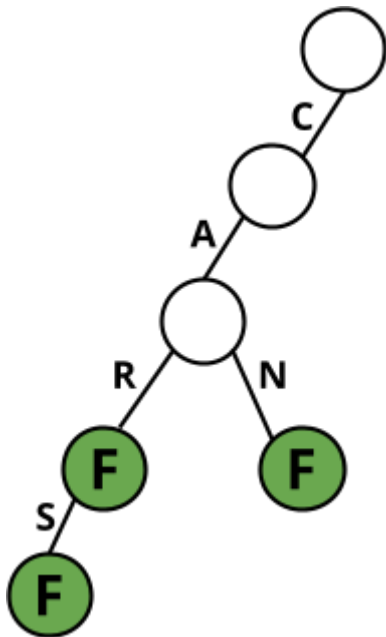
In our activity we compiled and ran our test file by specifying the classpath of the JUnit.jar file with `javac -cp .:junit5.jar TestMyList.java`

Tries

What is a trie?

A tree-like data structure to store Strings.

In this trie there are stored the words: Car, Cars, and Can



Insert

Strategy:

1. Start at the root of the Trie

2. For each character in the new word:
 - a. Check if the current node has a child pointer matching the character
 - i. If it does, follow pointer to child node and continue to the next character
 - ii. If it doesn't, create a child pointer matching the character and pointing to a new node
 1. Set "final" flag of the new node to False
 2. Move to the new node and continue to the next character
 - b. Repeat until all characters in the new word have been inserted
3. Once all characters inserted, set the "final" flag of the final node to True to mark the end of the word

Remove/Lookup

Strategy:

1. Start at the root of the Trie
2. Follow the child pointers matching the characters to the end of the word
 - a. If a child pointer doesn't exist, the word is not in the Trie
 - b. If final flag is False, the word is not in the Trie

If removing:

3. Set "final" flag of the final node as False
 - a. If final flag is already False, the word is not in the Trie
4. If the final node has no child nodes, recursively delete the node and its parent nodes until a node with other child pointers or a node with final as true is encountered, or the root node is reached

Time Complexity

Always $O(n)$, where n is the length of the string stored (for all operations)

Trie Visualization Tool

<https://www.cs.usfca.edu/~galles/visualization/Trie.html>

Lambda Expressions, etc.

Lambda expressions

- Defined using the syntax (parameters) -> expression
- More concise and functional than anonymous inner classes

```
public static MathOperation mul() {
```

```
        return (double first, double second) -> first * second;
    }
```

Anonymous inner classes:

- Used to define a class on-the-fly without giving it a name
- Defined using the syntax `new <class or interface> () {...}`
- Useful for creating a subclass of an existing class or interface that is only used once

```
public static MathOperation add() {
    return new MathOperation() {
        public double compute(double first, double second) {
            return first + second;
        }
    };
}
```

Named inner classes

- Also known as nested classes
- Defined as a class within another class, with its own name and scope
- Can be static or non-static
- Useful for encapsulating logic that is only used within a single class

```
public static MathOperation sub() {
    class SubtractionOperation implements MathOperation {
        public double compute(double first, double second) {
            return first - second;
        }
    }
    return new SubtractionOperation();
}
```

Sets

An unordered collection of unique items

Operations

- Union: All elements in either set A or set B
- Intersection: All elements in both set A and set B

Terminology / Symbols

\emptyset : Null/Empty set

\in : is a member of ($1 \in \{0-9\}$)

\notin : is not a member of ($a \notin \{0-9\}$)

$|A|$ - Cardinality - Number of elements in set

$A \subset B$ - Proper Subset: A is a subset of B if all elements of A are elements of B but $A \neq B$

$A \subseteq B$ - Subset: A is a subset of B if all elements of A are elements of B

$A \supset B$ - Proper Superset: A is a superset of B if all elements of B are elements of A but $A \neq B$

$A \supseteq B$ - Superset: A is a superset of B if all elements of B are elements of A

$A \cup B$ - Union: Elements of either A or B

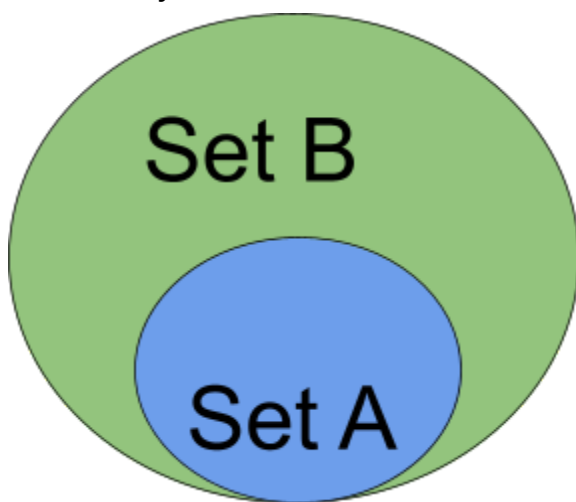
$A \cap B$ - Intersection: Elements of both A and B

Special Methods

- `Set.containsAll(Set s)`: returns true if the set contains all of the elements in s
- `Set.addAll(Set s)`: adds all elements of s not already in the set to the set (set instance \rightarrow union of set instance and s)
- `Set.retainAll(Set s)`: removes all elements of the set not in s (set instance \rightarrow intersection of set instance and s)

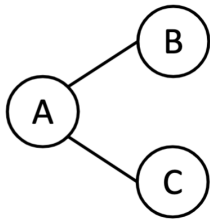
Implementations:

- HashSet (hashtable)
- Trie
- Search Tree / BST
- Array/LinkedList



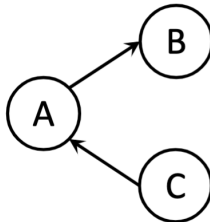
Graphs

Undirected Graphs



- Edges have no direction
- Degree of a node: number of edges connected to the node

Directed Graphs



- Edges have a direction
- In-degree of a node: number of edges pointing towards node
- Out-degree of a node: number of edges pointing away from node

Paths in a Graph

- Cycle: a path that returns to a previously visited node
- Cyclic: contains at least one cycle
- Acyclic: contains no cycles

Weighted Graphs

- Assign a cost or weight to each edge

Subgraphs

- G' is a subgraph of G if:
 - Nodes of G' subset of nodes of G
 - Edges of G' subset of edges of G

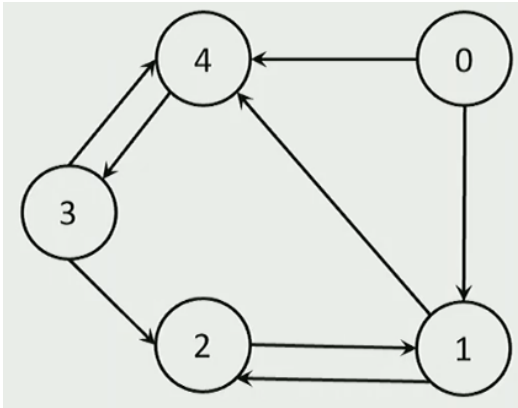
Connected Graphs

- A path exists between every pair of nodes

- For directed graphs:
 - Strongly connected: a path exists between every pair of nodes with edge direction respected
 - Weakly connected: a path exists between every pair of nodes if edge direction is not respected

$O(n) \leq \#edges \leq O(n^2)$
 sparse dense

Adjacency Matrix



	0	1	2	3	4
0	F	F	F	F	F
1	T	F	T	F	F
2	F	T	F	T	F
3	F	F	F	F	T
4	T	T	F	T	F

In example: columns source; rows target

- Matrix representation of a graph
- Shows connections between nodes
- Directed vs. undirected graphs
 - If the graph is an undirected graph, the matrix is symmetric (meaning that the entry (i, j) is equal to the entry (j, i))
- Weighted vs. unweighted graphs
 - In a weighted graph, the entries represent the weight of the edge between the nodes
 - In an unweighted graph, the entries equal 1 if there is an edge between the nodes, or 0 if not

Depth-First Traversal

Pseudocode:

```

DFT(v):
  mark v as visited
  for each unvisited neighbor u of v:
    DFT(u)
  
```

Breadth-First Traversal

Pseudocode:

```
BFT(v):
    q = new Queue()
    mark v as visited
    q.enqueue(v)
    while(!q.isEmpty()):
        c = q.dequeue();
        for each unvisited neighbor u of c:
            mark u as visited
            q.enqueue(u)
```

JavaFX

Application Structure

```
public class JavaFXActivity extends Application {

    @Override
    public void start(Stage stage) {
        // JavaFX code here
    }







    public static void main(String[] args) {
        launch(args);
    }
}
```

Useful Classes

Name	Constructor	Notes
Scene	<code>new Scene(root, width, height);</code> <code>stage.setScene(scene);</code>	Represents the visual content of a JavaFX application
Group	<code>new Group(node1, node2, node3);</code>	Group nodes together

Layout Managers

<code>new BorderPane();</code>	<code>new HBox();</code>	<code>new VBox();</code>
--------------------------------	--------------------------	--------------------------

		
<code>new FlowPane();</code>	<code>new GridPane();</code>	<code>new AnchorPane();</code>
		

Event Handlers

Events:

- Java object that represents user interaction with the application
- A subtype of `javafx.event.Event`

How event handling work:

- Event is created at the GUI element the user interacted with
- “Bubbles up” GUI tree until it is stopped by calling `event.consume()` in an event handler
- An event handler is created by calling `.addEventHandler(EventType eventType, EventHandler eventHandler)` on a GUI node

EventHandler is given by:

```
public interface EventHandler<T extends Event> {
    public void handle(T event);
}
```

(EventHandler is a great application of Lambda expressions)

Event Filters

How event filters work:

- Before an event is processed by an event handler, it first can be handled by event filters.
- When an Event is dispatched, it gets sent down the GUI tree from the top:
 - As it goes down the tree, it goes through every event filter in the chain

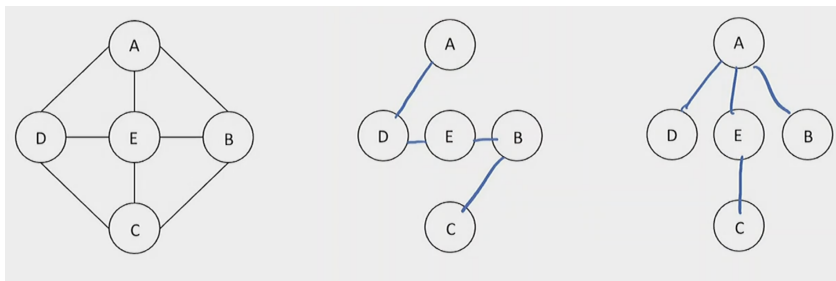
- If any filter calls `event.consume()`, it stops and bubbles back up tree to be handled by any event handlers
 - If no filter calls `consume`, it stops at the element interacted with before bubbling back up
- An event filter is created by calling `.addEventFilter(EventType eventType, EventHandler eventFilter)` on a GUI node

Spanning Trees

Definition

A tree is a spanning tree of an undirected graph, if:

- Contains all nodes of the graph
- All nodes are connected
- Number of edges is: $(\text{num. of nodes}) - 1$



General Spanning Trees

Use Depth-First and Breadth-First Traversal to insert edges

```
DFT(v):
    mark v as visited
    for each unvisited neighbor u of v:
        add edge v-u to tree T
        DFT(u)
```

```
BFT(v):
    q = new Queue()
    mark v as visited
    q.enqueue(v)
    while(!q.isEmpty()):
        c = q.dequeue();
        for each unvisited neighbor u of c:
            add edge c-u to tree T
            mark u as visited
```

```
q.enqueue(u)
```

Minimum spanning trees

- Only apply to weighted, undirected graphs
- The minimum spanning tree is the one with the lowest sum of edge weights

Prim's Algorithm

1. Choose a node to start the tree
 - a. Mark node as visited
2. Create a priority queue to store the edges that connect to the tree and their weights
 - a. Add all edges that connect to the starting node to the priority queue
3. While the priority queue is not empty:
 - a. Remove the edge with the lowest weight from the priority queue
 - b. Add edge to tree
 - c. For the node the edge connects to:
 - i. Mark node as visited
 - ii. Add all of the node's outgoing edges to nodes not yet visited to the priority queue

Kruskal's Algorithm

1. Sort the list of edges from lowest to highest weights
2. Create a list of sets, putting each of the nodes of the tree in its own set
3. While the edge list is not empty:
 - a. Remove the edge with the lowest weight from list
 - b. If the start node and the end node of the edge are in different sets:
 - i. Add edge to tree
 - ii. Combine the sets containing the start and end nodes

Dijkstra's Algorithm

1. Initialize the starting node with a cost of 0, every other node with a cost of ∞
2. Add all nodes to a priority queue of unvisited nodes
3. While the queue is not empty:
 - a. Remove the node with the lowest cost from the queue
 - b. For each neighbor of the node:
 - i. Compute: $C = \text{the cost of the current node} + \text{cost of the edge to the neighbor}$
 - ii. If $C < \text{current cost of neighbor}$:

1. Update the cost of neighbor to C
2. CSet the previous node of the neighbor to the current node

Time Complexity

V: nodes in graph

E: edges in graph

Algorithm	Average Case	Worst Case
Prim's	$O(E \cdot \log E)$	$O(E \cdot \log V)$
Kruskal's	$O(E \cdot \log V)$	
Dijkstra's	$O(E \cdot \log E)$	$O(E \cdot \log V)$

.DOT Files

Undirected graphs

- Keyword: graph
- Edges represented as from -- to [properties]
- Nodes represented as name [properties]

Example:

```
graph Network {
  dejope -- ogg [label="University" distance=3]
  dejope -- waters [label="Observatory" distance=3]

  dejope [label="Dejope" pop=1000 dining="Flakes"]
  ogg [label="Ogg" pop=2000 dining="None"]
  waters [label="Waters" pop=800 dining="Liz Market"]
}
```

Directed graphs

- Keyword: digraph
- Edges represented as from -> to [properties]
- Nodes represented as name [properties]

Example:

```
digraph Network {
  dejope -> ogg [label="University" distance=3]
  dejope -> waters [label="Observatory" distance=3]
```

```
dejope [label="Dejope" pop=1000 dining="Flakes"]
ogg [label="Ogg" pop=2000 dining="None"]
waters [label="Waters" pop=800 dining="Liz Market"]
}
```

Properties

- Contained in the square brackets after edge or node definition
- Contain information about edge or node
- In examples above, contain information about campus roads (the edges) and residence halls (the nodes)

Using .DOT files

To generate an image of the graph/digraph, run: `dot -Tsvg example.dot -O`

HTML, CSS, JS, CGI

CGI

- Stored in the cgi-bin located at `/usr/lib/cgi-bin`
- Make a call to Java programs by passing arguments through to the `String[] args` in the main method

Not really going to take notes on this.

B-Trees/234-Trees

Properties

- Branching factor: the number of children at each node
 - In 2-3-4 trees, the branching factor is 4
- Number of keys in each node is: $\text{branching factor} - 1$
 - Keys and child nodes are sorted in ascending order
- $\text{Height} = \log_m N$, where N is the number of nodes, M is branching factor
- Self-balancing
 - All leaves are on the same level
 - Insertions only happen in leaves
 - B-Trees grow (add a new level) at the root

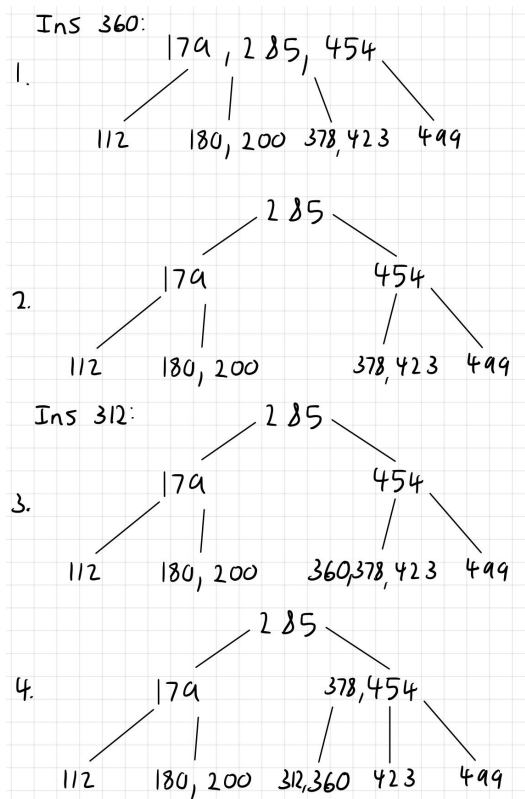
Insertion (234 Tree)

1. Start at the root of the tree
2. Do insertion normally like any other search tree (ex. BST)

3. Every time a 3-node is encountered split it and put the middle key in the parent node
 - a. Recursively doing this if this causes the parent to now be a 3-node

Really simple!

Example (from 234 quiz):



Time Complexity

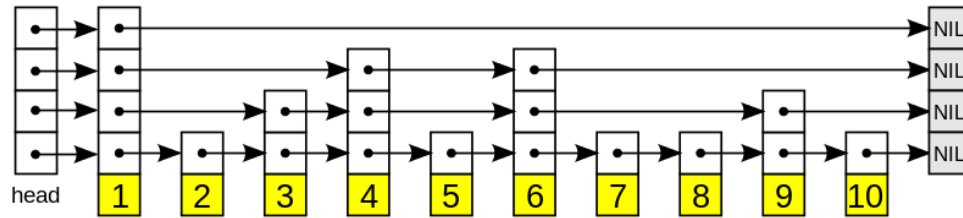
All operations, always (average, worst case): $O(\log n)$

2-3-4 Tree Visualization Tool

<https://www.cs.usfca.edu/~galles/visualization/BTree.html>

(make sure to select "Max. Degree = 4")

Skip Lists



Search

1. Start at highest level
2. Look ahead: is the next node at level bigger than the node we're looking for?
 - a. If so, go down a level
 - b. Otherwise, move forward to that node, then go down a level
 - c. Repeat until reached target node

Insertion

1. Follow search algorithm to determine where to insert node
2. Insert node in lowest level
3. Flip coin:
 - a. If heads, insert in next level & repeat
 - b. If tails, don't insert in any more levels

Deletion

Follow search algorithm to determine where to delete from, then follow normal linked list deletion algorithm.

- Singly-linked list will compare the successor node
- Doubly-linked list will compare the current node
- In the image above, the arrows are one directional, meaning it's a singly-linked list.

Time Complexity

	Average Case	Worst Case
All Operations	$O(\log N)$	$O(N)$

Space Complexity

Average Case	Worst Case
$O(N)$	$O(N \log N)$

Skip List Visualization Tool

<https://cmps-people.ok.ubc.ca/ylucet/DS/SkipList.html>

Regular Expressions

Play around with regex: <https://regex101.com/>

(abc) == find "abc" exactly

[abc] == find either "a" or "b" or "c"

[^abc] == find any character but "a" or "b" or "c"

the following **are not** in []

^a == "a" must be the start of line

a\$ == "a" must be the end of line

the following **are** in []

[^a] == find anything but "a"

[a\$] == find "a" or "\$"

a* == find "a" between 0 and infinite times (prefer as many as possible)

- "a" : has 1 a
- "aaaa" : has 4 a
- "" : has 0 a

a+ == find "a" between 1 and infinite times (prefer as many as possible)

- "a" : has 1 a
- "aaaa" : has 4 a

a? == find "a" between 0 and 1 times (prefer as many as possible)

- "a" : has 1 a
- "" : has 0 a

a{5, 9} == find "a" between 5 and 9 times, inclusive (prefer as many as possible)

- "aaaaa" has 5 a
- "aaaaaaa" has 7 a
- "aaaaaaaaa" has 9 a

Streams

What is required for a stream?

- Data source -> i.e Stream<T> (type tells us what kind of data we are processing with that stream)

- Chain of operations (Stream -> operation -> Stream) will do some operation on the stream can link them together
- One terminal operation
 - Intermediate operations: are lazy and output stream for pipelining other operations
 - Terminal operations: are eager and compute results of the whole pipeline

Data Source: Examples

- `java.util.stream.Stream.generate(Supplier<T>)`
 - This is a way to generate a stream, the generate method expects an object as a parameter of type supplier which is an interface type (supplier has a get method only method of that object and so we can use lambda expression to implement if needed)
- `java.util.stream.Stream.of(T...items)`
 - Can pass in a list or an array of all the data items that we want to put on the stream, and will put those on to the stream one by one
- We can create a new stream from existing streams by combining them together (`Stream.concat(Stream, Stream2)`) they have to be of the same type
- `Stream.empty`
 - No parameters
 - Creates an empty stream that does not have any data items on it
- `java.nio.file.Files.lines(Filepath)`
 - Creates a stream for us with data items of type string
 - On that stream puts from top to bottom puts lines of the text file that we give it the path to

Intermediate operations: Examples

- `.filter(Predicate<T>)`
 - Predicate has single method of return type boolean
 - Only puts it on outgoing stream if method evaluates to true
 - Ex. (1, 2, 3, 4) -> `.filter((item) -> item%2 == 0)`
 - Outgoing stream will only have even integers
- `.map(Function<T, R>)`
 - Say we have an incoming stream "CS400", "JAVA" -> `.map((item) -> item.toLowerCase())`
 - Maps uppercase to lowercase in this case
- `.limit(n)`

- Takes an integer
 - Stop data items from passing through after n data items have passed through
- `.skip(n)`
 - Integer as parameter
 - Ignores first n elements from the incoming stream

Terminal Operations: Examples

- `.findFirst()`
 - No parameters are taken
 - What this operation does is it will wait until the first element comes out of the incoming stream stops the stream and returns that first element as the stream
- `.forEach(Consumer<T>)`
 - Consumer is an interface type that has a single method of type void so can be implemented with a lambda expression
- `.count()`
 - Takes no parameters
 - Counts number of data items that come out of the stream, after counts all returns number of data items coming out of the stream
- `.min(Comparator<T>)`
 - T is the data item of the stream
- `.max(Comparator<T>)`
 - Maximum or minimum data item that are coming out of our stream
- `.reduce(T identity, BinaryOperator<T> Accumulator)`
 - identity: The identity element is both the initial value of the reduction and the default result if there are no elements in the stream. In this example, the identity element is 0; this is the initial value of the sum of ages and the default value if no members exist in the collection roster.
 - accumulator: The accumulator function takes two parameters: a partial result of the reduction (in this example, the sum of all processed integers so far) and the next element of the stream (in this example, an integer). It returns a new partial result. In this example, the accumulator function is a lambda expression that adds two Integer values and returns an Integer value:
(a, b) -> a + b

What streams look like in code

```
public class Streams {  
    Run | Debug  
    public static void main(String[] args) throws Exception {  
        Stream.of(...values:3,6,4,7,6,5)  
            .filter( (number) -> number < 6 )  
            .map( (number) -> number * 2 )  
            .forEach( (number) -> System.out.println(number) );  
    }  
}
```

2 OUTPUT TERMINAL

- Filter is using a lambda to remove each value of the stream that is greater than 6
- The map multiplies each number by 2 so mapping the inputted number to the value double it
- And the for each prints out the value (thus is a terminal operation)

AVL Trees

Adelson-Velsky and Landis Trees

- Self Balancing binary tree (alternative to RBTs)
- Uses rotations to maintain balance
- Goal:
 - Keep H (height of tree (with N keys) growing in $O(\log N)$)

AVL Tree Insertion

- Insert using BST insertion algorithm
- Check if tree is out of balance and rebalance if it is

AVL Tree Deletion

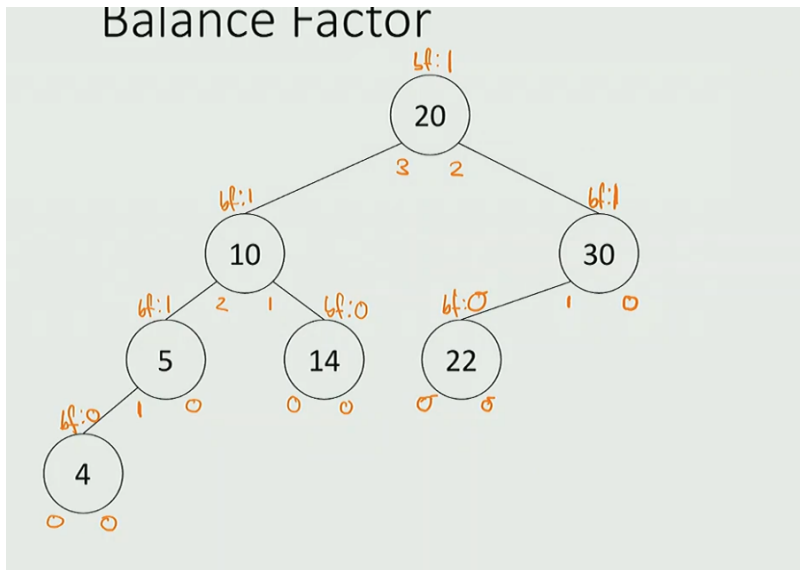
- Won't be on the exam! ([Source](#))

Balance factor

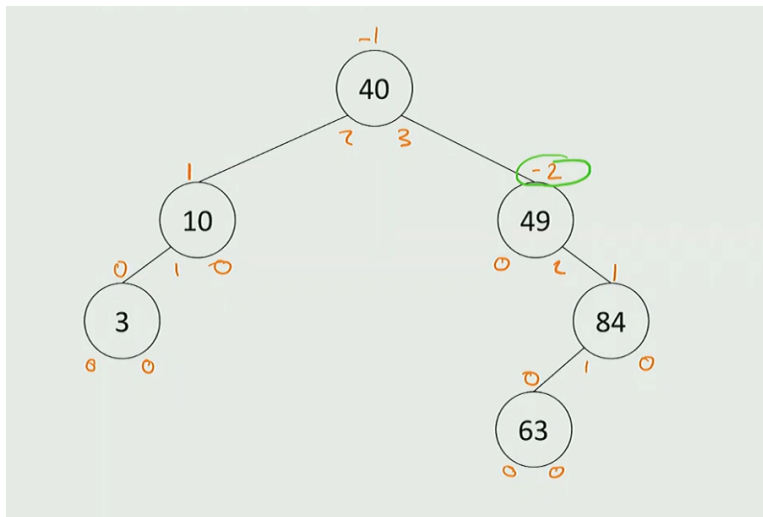
- $Bf = \text{height}(\text{left subtree}) - \text{height}(\text{right subtree})$
- A node is balanced if $|bf| < 2$
- Sign tells us direction of imbalance

- A tree is balanced if every node in the tree is balanced

Examples:



- This is an example of a balanced tree as no absolute values of the balance factors are 2 or larger



- For this tree node 49 has a balance factor of -2 and now we must look at the parent child relationships of the nodes below it to figure out what kind of re-balancing operation we need to do
- Since the parent child relationships are different we will need to do a combination rotation
- Since the value is negative we know the right subtree is the larger height so we must do a right left rotation
- A right rotation at 84 and a left rotation at 49 to balance it

AVL Tree Time Complexity

- Worst case (insertion) = $O(\log N)$
- Spacetime complexity average and worst is $O(n)$

AVL Tree Visualization Tool

<https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>