CS 261-020
Data Structures

Lecture 16

Dijkstra's

Final Exam Review

3/14/24, Thursday

Oregon State University

Odds and Ends

- Due Reminder:
  - Quiz 5 due Sunday midnight via Canvas – open today after the lecture
  - Assignment 5 due Sunday midnight via TEACH

- Tomorrow (Friday 3/15) is the last day to demo any assignments
  - No late demo penalty for assignment 4
  - 30% late demo penalty for assignment 1-3

Lecture Topics:

- Dijkstra's
- Final Exam Review

Dijkstra's algorithm: single source lowest-cost paths

- Dijkstra's algorithm: finds the shortest/lowest-cost path from a specified vertex in a graph to all other reachable vertices in the graph.

- In Dijkstra's algorithm, we will use a priority queue to order our search.
  - The priority values used in the queue correspond to the cumulative distance to each vertex added to the PQ.
  - Thus, we are always exploring the remaining node with the minimum cumulative cost.

Dijkstra's algorithm: single source lowest-cost paths

Algorithm, which begins with some source vertex v_s:

- Initialize an empty map/hash table representing visited vertices.
  - Key is the vertex v.
  - Value is the min distance d to vertex v.
- Initialize an empty priority queue, and insert v_s into it with distance (priority) 0.
- While the priority queue is not empty:
  - Remove the first element (a vertex) from the priority queue and assign it to v. Let d be v's distance (priority).
  - If v is not in the map of visited vertices:
    - Add v to the visited map with distance/cost d.
    - For each direct successor v_i of v:
      - Let d_i equal the cost/distance associated with edge (v, v_i).
      - Insert v_i to the priority queue with distance (priority) $d + d_i$.

Dijkstra's algorithm: single source lowest-cost paths

- This version of the algorithm only keeps track of the minimum distance to each vertex, but it can be easily modified to keep track of the min-distance path, too.
  - Augment the visited vertex map and the priority queue to keep track of the vertex previous to each one added.

- The complexity of this version of the algorithm is $O(|E| \log |E|)$.
  - The innermost loop is executed at most $|E|$ times, and the cost of the instructions inside the loop is $O(\log |E|)$.
    - Inner cost comes from inserting into the PQ.

Lecture Topics:

- Final Exam Review

Final Exam

- 3/20 Wednesday from 2:00 – 3:20 pm
- Same classroom
- Close book, close notes
- No calculator allowed
- Question types: multiple choices, T/F, short answer
  - Similar to the Midterm Exam
- Bring pencil/pen, and **your photo ID** (student ID/driver license/passport)
- Scratch paper will be provided upon request

Final

- Topics: Week 6-10 (lecture 9-16):
  - Binary Search Trees
    - Tree vs. Binary Tree
    - BST Operations and their complexity:
      - Finding an element
      - Inserting an element
      - Removing an element
    - Traversal
      - DFS: Pre-order vs. in-order vs. post order
      - BFS: level order

Final

- Topics: Week 6-10 (lecture 9-16):
  - AVL Tree
    - Balance factor of a node
    - Single rotation vs. double rotation
    - Runtime complexity of AVL tree operations
  - Priority Queues
    - Array-based heap (min/max heap)
    - Operations:
      - Insert, remove
      - Percolations
    - Build a heap from an arbitrary array
    - Heapsort
  - Map and Hash table
  - Graph

		`balanceFactor(N)`	
		-2 (left-heavy)	2 (right-heavy)
`balanceFactor(C)`	-1 (left-heavy)	**Left-left imbalance** Single rotation: right around *N*	**Right-left imbalance** Double rotation: 1. right around *C* 2. left around *N*
	0		
	1 (right-heavy)	**Left-right imbalance** Double rotation: 1. left around *C* 2. right around *N*	**Right-right imbalance** Single rotation: left around *N*

Final

- Topics: Week 6-10 (lecture 9-16):
  - Map and Hash table
    - Hash functions
    - HT operations and their runtime complexity:
      - lookup
      - Insert
      - Remove
    - Resolve Hash collisions
      - Chaining
      - Open Addressing: Tombstone
    - Load factor

Final

- Topics: Week 6-10 (lecture 9-16):
  - Graph
    - Representation: adjacency list vs. adjacency matrix
    - Single source reachability
    - DFS vs. BFS in graph
    - Single source lowest-cost paths
      - Dijkstra’s Algorithm

Study Guide

- Review quiz questions
- Review slides
- Take practice final (and time yourself)
- Study recitation and assignments

Assignment 5 Q&A

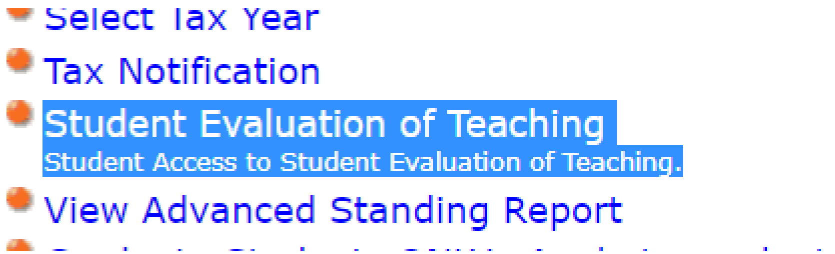
Be Confident…

Now you are able to…

- Describe the properties, interfaces, and behaviors of basic abstract data types
- Read an algorithm or program code segment and analyze the time complexity.
- State the time complexity of the fundamental operations associated with a variety of data structures.
- Recall the space utilization of common data structures in terms of the long-term storage needed to maintain the structure, as well as the short-term memory requirements of fundamental operations, such as sorting.
- Design and implement general-purpose, reusable data structures that implement one or more abstractions.
- Compare and contrast the operation of common data structures in terms of time complexity, space utilization, and the abstract data types they implement.

Final Remarks…

- Thank you so much for your commitment to this course
- Future improvements?
  - MyOSU → Student Records →



- ULA position
  - Contact me! And apply through: https://jobs.oregonstate.edu/postings/140560

Final Remarks…

- Submit all your work by the deadline
  - Assignment 5, quiz 5
- Final exam on Wednesday, 3/20 2:00 pm @ WNGR 151
  - Bring your photo ID
- Grade disputation:
  - By 3/23 6pm

*Additional Topics

- Sets ADT and its implementation
- Git and GitHub

- *Will not be on the final

Set

- Set – An ADT that can store unique values, without any particular order.
- Unique → no duplicates
- Unordered → cannot access items using index values

- Array: [1,1,2,2,3,4,1,5,8,7]
- Set: {1,2,3,4,5,8,7} ← Note: no duplicates

- Why using set?
  - Check if a specific element is contained in the set

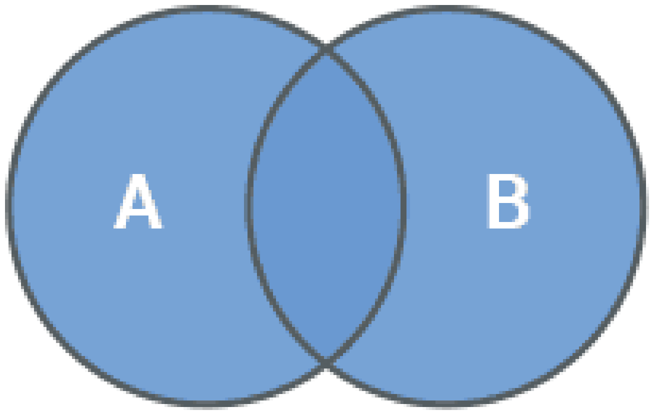
Set Operations

- The idea of a Set has been translated directly from mathematics into programming languages.
  - Such as in Python

- Basic operations:
  - *contains()* – search for a specific element and see if it is contained in the set
  - *add()* – add an element into the set
  - *remove()* – remove an element from the set

Set Operations

- More operations:
  - *union()* – return the union of two sets
  - Example:
    - A = {2, 5, 7}
    - B = {1, 2, 5, 8}
    - Then A Union B (A U B) = {1, 2, 5, 8}



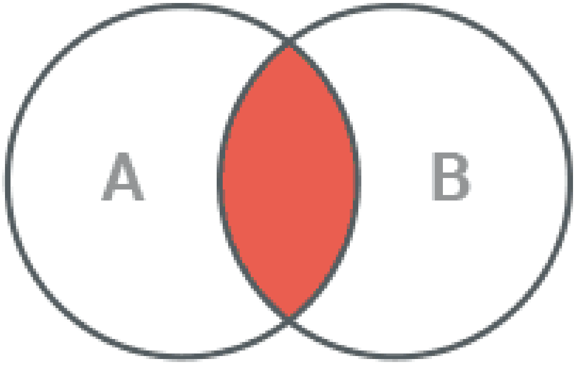
Union

- In Python:

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}
# by operator
print(A | B)
# Prints {'blue', 'green', 'yellow', 'orange', 'red'}
# by method
print(A.union(B))
# Prints {'blue', 'green', 'yellow', 'orange', 'red'}
```

Set Operations

- More operations:
  - *intersection()* – return the intersection of two sets
  - Example:
    - A = {2, 5, 7}
    - B = {1, 2, 5, 8}
    - Then A intersects B (A ∩ B) = {2, 5}



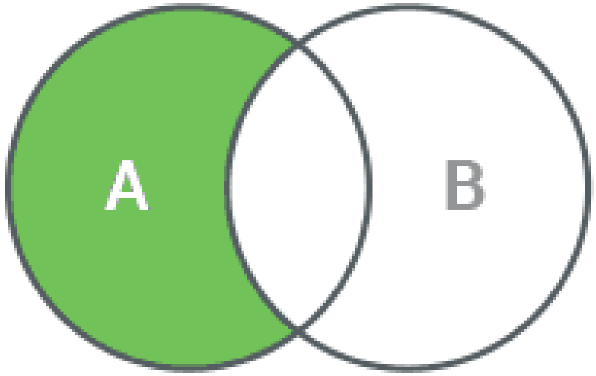
Intersection

- In Python:

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}
# by operator
print(A & B)
# Prints {'red'}
# by method
print(A.intersection(B))
# Prints {'red'}
```

Set Operations

- More operations:
  - *difference()* – return the difference of two sets
  - Example:
    - A = {2, 5, 7}
    - B = {1, 2, 5, 8}
    - Then Set difference of A and B (A - B) = {7}



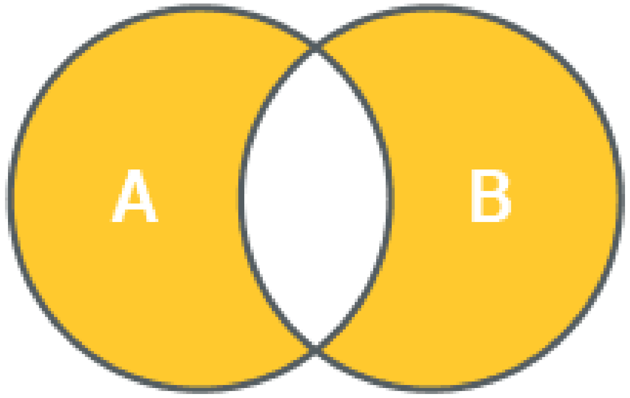
Difference

- In Python:

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}
# by operator
print(A - B)
# Prints {'blue', 'green'}
# by method
print(A.difference(B))
# Prints {'blue', 'green'}
```

Set Operations

- More operations:
  - *symmetric_difference()* – return the set of all elements in either A or B, but not both
  - Example:
    - A = {2, 5, 7}
    - B = {1, 2, 5, 8}
    - Then Set difference of A and B (A ^ B) = {7, 1, 8}



Symmetric Difference

- In Python:

```
>>> first_set = {1, 2, 3, 4, 5, 6}
>>> second_set = {4, 5, 6, 7, 8, 9}
>>> first_set.symmetric_difference(second_set)
{1, 2, 3, 7, 8, 9}
>>>
>>> first_set ^ second_set     # using the `^` operator
{1, 2, 3, 7, 8, 9}
```

Set Implementation

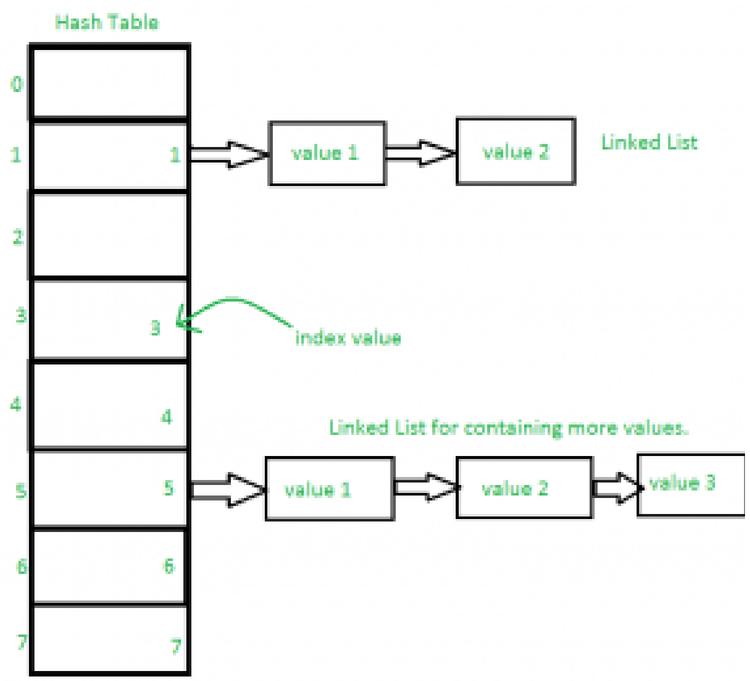
- Multiple ways of implementing a set ADT
  - Hash-based approach
  - Tree-based approach

Set Implementation: Using a Hash Table

- The underlying data structure is a hash table

Key (element) → Hash Function → Index

- Use either chaining or open addressing to resolve collisions



Set Implementation: Using a Hash Table

- *contains()* – search for an element and see if it is contained in the set

- Similar to the lookup() in the hash table:
  - Take the element (key)
  - Apply the hash function, and get the index
  - Access

- Complexity: O(1)

Set Implementation: Using a Hash Table

- *add()* – add an element into the set

- Similar to the insert() in the hash table:
  - Take the element (key)
  - Apply the hash function, and get the index
  - Insert
    - Resize and rehash if needed
    - Resolve collision if needed

- Complexity: avg. O(1)

Set Implementation: Using a Hash Table

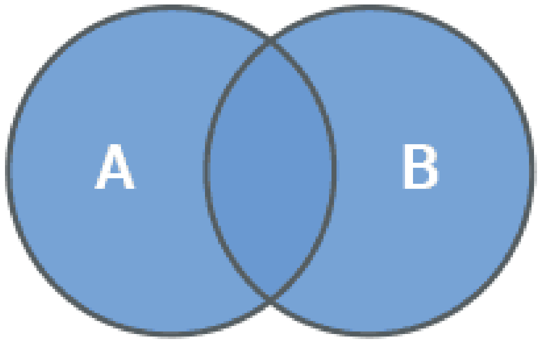
- *remove()* – remove an element from the set

- Similar to the remove() in the hash table:
  - Take the element (key)
  - Apply the hash function, and get the index
  - Remove
    - Add dummy node (tombstone) if needed

- Complexity: O(1)

Set Implementation: Using a Hash Table

- *union(set A, set B)* – return the union of two sets



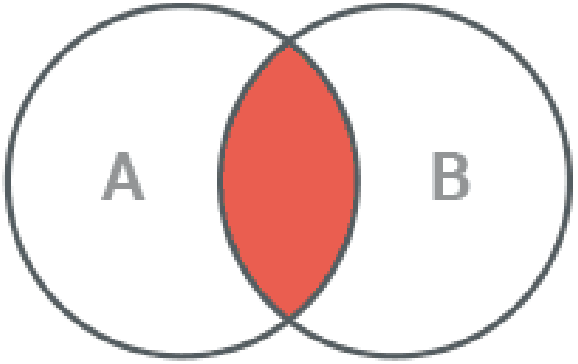
Union

- Procedure:
  - Create an empty set, say S
  - Add all elements of A into S
  - Add all elements of B into S
  - Return S
  - *Note: since hash table cannot have duplicate keys, it handles "no duplicates" rule in Sets
- Complexity: O(size(A) + size(B))

Set Implementation: Using a Hash Table

- *intersection(set A, set B)* – return the intersection of two sets

- Procedure:
  - Create an empty set, say S
  - Loop through each element A_i in set A
    - If A_i is in B (by calling contains())
      - Add A_i into S
  - Return S

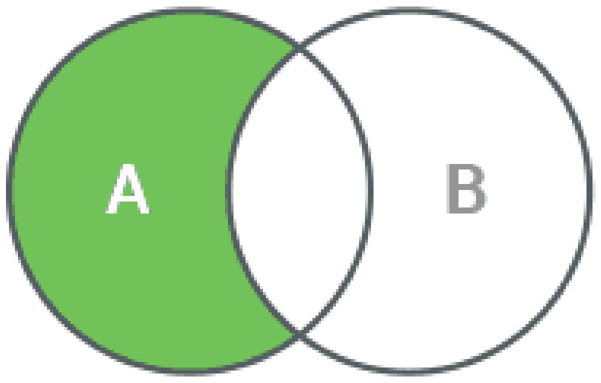


Intersection

- Complexity: O(min(size(A), size(B)))

Set Implementation: Using a Hash Table

- *difference(set A, set B)* – return the difference of two sets
  - in this case: A - B



Difference

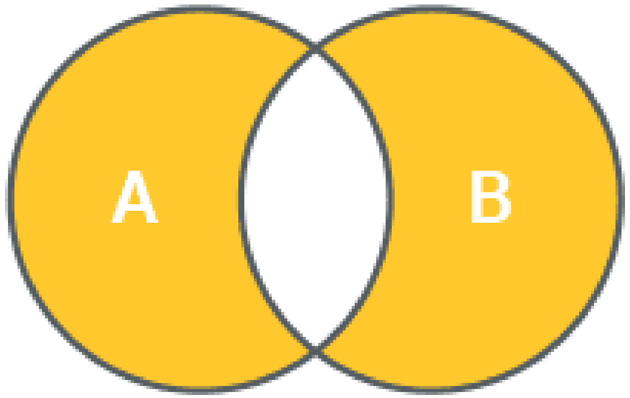
- Procedure:
  - Create an empty set, say S
  - Loop through each element A_i in set A
    - If A_i is NOT in B (by calling contains())
    - Add A_i into S
  - Return S

- Complexity: O(size(A))

Set Implementation: Using a Hash Table

- *symmetric_difference(set A, set B)* – return the symmetric difference of two sets

- Procedure:
  - Create an empty set, say S
  - Loop through each element A_i in set A
    - If A_i is NOT in B (by calling contains())
    - Add A_i into S
  - Loop through each element B_i in set B
    - If B_i is NOT in A (by calling contains())
    - Add B_i into S
  - Return S



Symmetric Difference

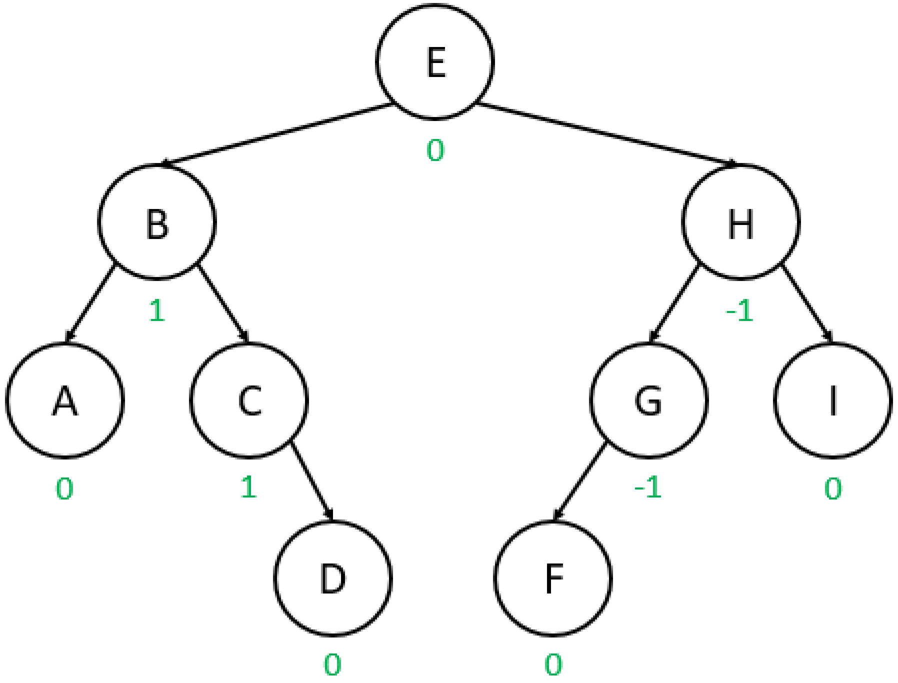
- Complexity: O(size(A)+size(B))

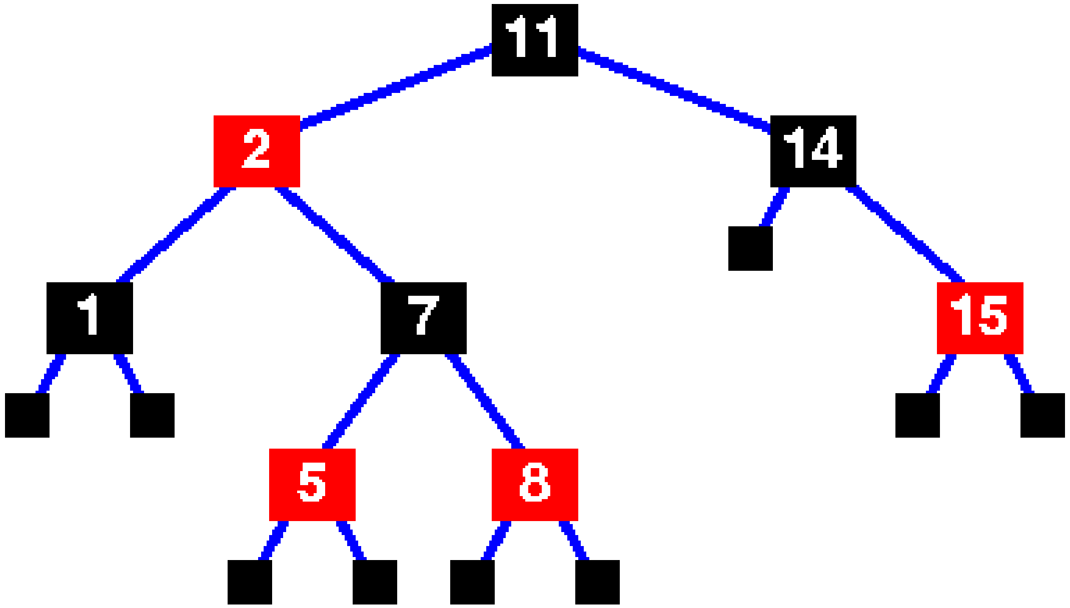
Set Implementation: Using a Hash Table

- Example Set Implementation in C using hash table:
- https://github.com/barrust/set

Set Implementation: Using a Tree

- The underlying data structure is a self-balancing tree:
  - AVL Tree
  - Red-black tree





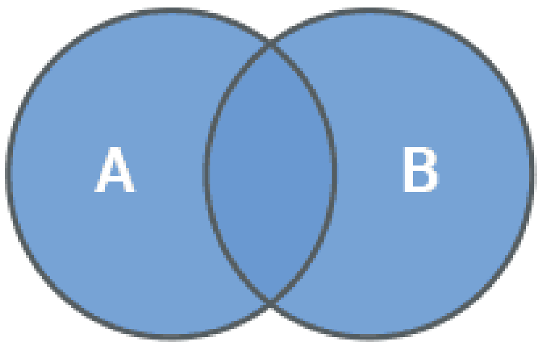
Set Implementation: Using a Tree

- *contains()* – search for an element and see if it is contained in the set
- *add()* – add an element into the set
- *remove()* – remove an element from the set

- Similar to AVL tree's *loopup()*, *insert()*, and *remove()*

- Complexity: O(log n) where n is the number of elements in the set

Set Implementation: Using a Tree



Union

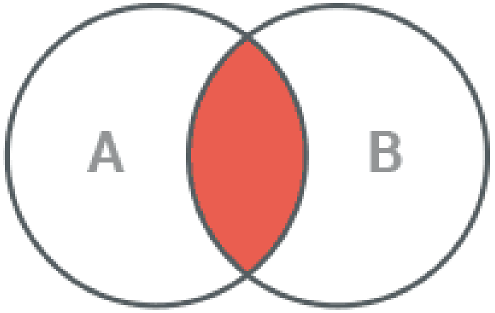
- *union(set A, set B)* – return the union of two sets

- Procedure:
  - Create an empty set S
  - Insert all elements of A into S → n elements, each takes O(log n), so O(nlogn)
  - For each element B_i in B:
    - If S contains B_i, skip
    - Else, insert B_i into S
  - Return S

Set Implementation: Using a Tree

- *intersection(set A, set B)* – return the intersection of two sets

- Procedure:
  - Create an empty set, say S
  - Loop through each element A_i in set A
    - If B contains A_i
      - Insert A_i into S
  - Return S

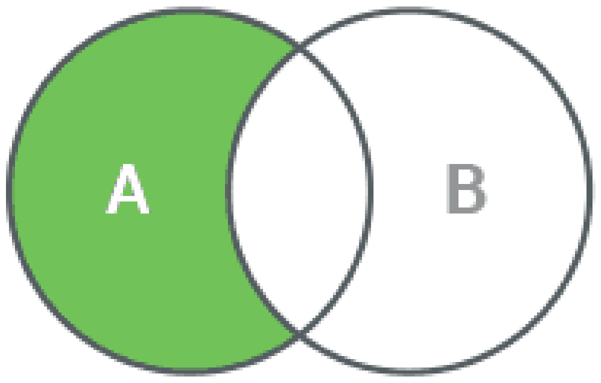


Intersection

Set Implementation: Using a Tree

- *difference(set A, set B)* – return the difference of two sets
  - in this case: A - B

- Procedure:
  - Create an empty set, say S
  - Loop through each element A_i in set A
    - If A_i is NOT in B (by calling contains())
    - Insert A_i into S
  - Return S

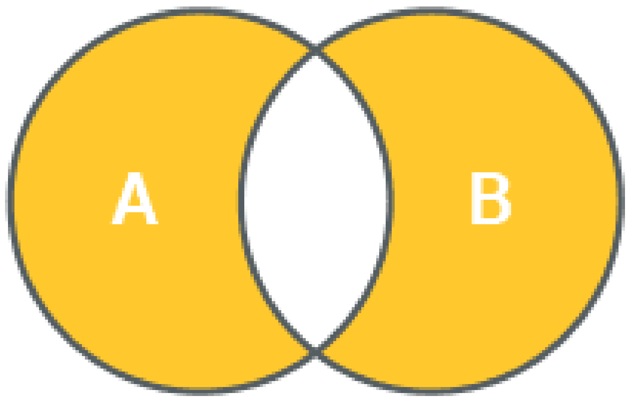


Difference

Set Implementation: Using a Tree

- *symmetric_difference(set A, set B)* – return the symmetric difference of two sets

- Procedure:
  - Create an empty set, say S
  - Loop through each element A_i in set A
    - If A_i is NOT in B (by calling contains())
    - Insert A_i into S
  - Loop through each element B_i in set B
    - If B_i is NOT in A (by calling contains())
    - Insert B_i into S
  - Return S



Symmetric Difference

Red-Black Tree

- Another type of self-balancing tree:

- Explore 6 YouTube videos here:

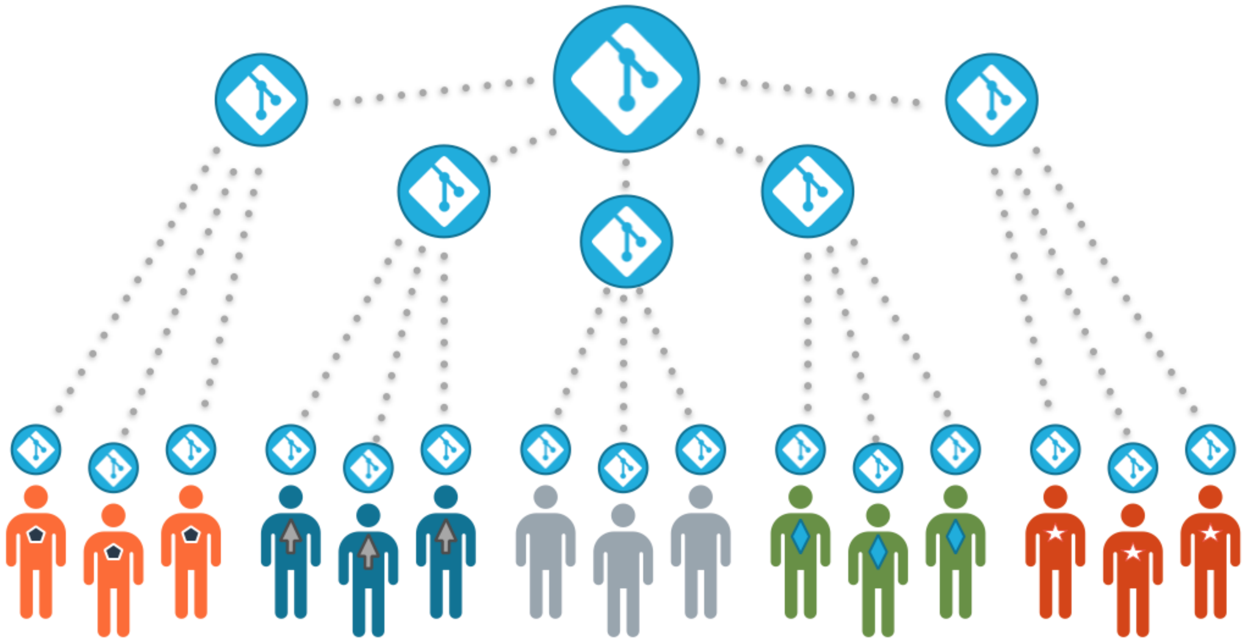
*Additional Topics

- Sets ADT and its implementation
- Git and GitHub

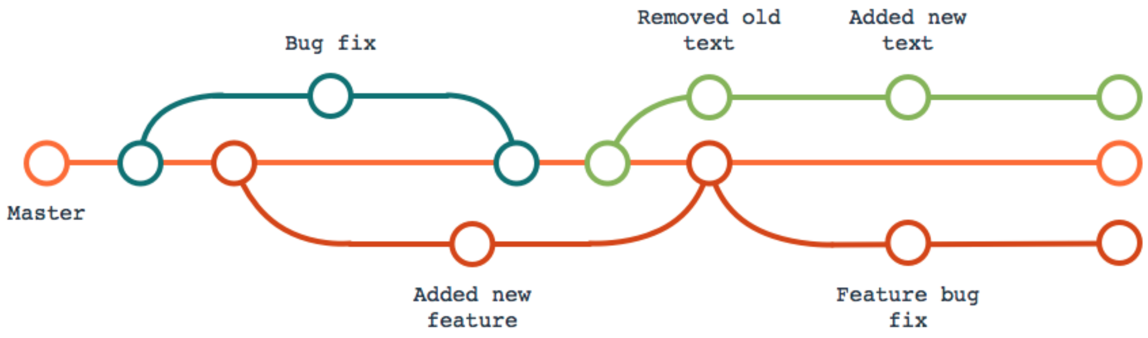
- *Will not be on the final

Git Overview

- Git is one of the most popular **version control systems** (VCS)
  - A VCS is a tool (a program) for managing changes to your code and for making it easier to work with many people on the same code.

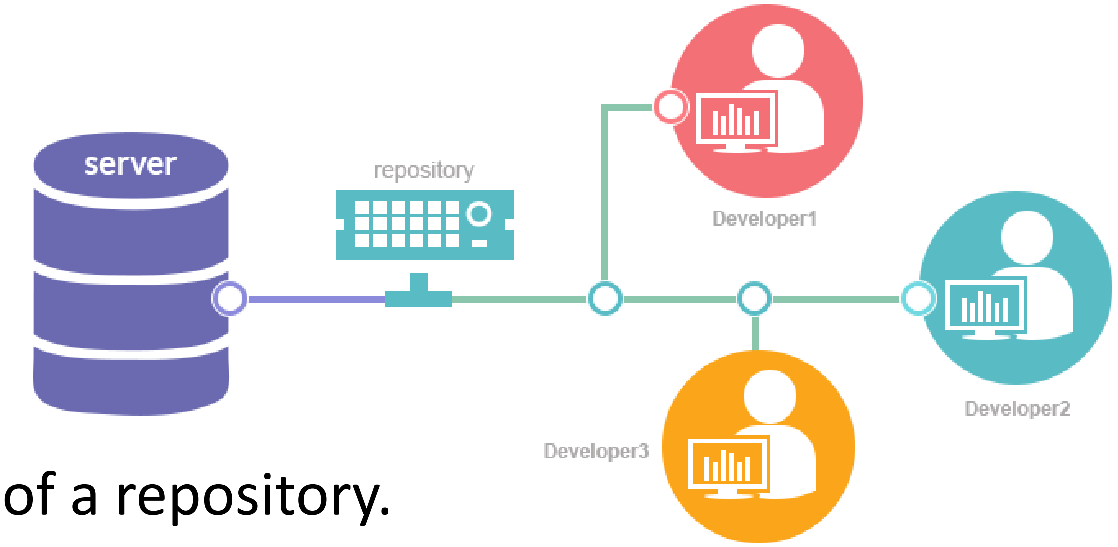


Git Overview



- Git manages changes in code by taking a snapshot of the entire codebase every time you tell it to
  - These snapshots are stored permanently in a repository.
  - Storing a snapshot in the repository is called committing your code.
  - Every new commit records a new version of the code.
  - Git maintains a history of all of the versions of a project ever recorded.
    - You can look at (and even revert to) your code at different points in its history, and compare the differences between different points in the history.

Git Overview



- Git is a **distributed VCS**.
  - Many computers can hold a copy of a repository.
    - Any non-local Git repository is called a remote repository.
  - Git has commands to synchronize copies of a repository between two machines.
  - This allows many people to work on the same piece of code easily.
    - Each person makes changes and commits them to their local repository.
    - Then they use Git’s synchronization commands to make sure their repositories are in sync.
      - Changes can be pushed from the local repository to a remote repository.
      - Changes can also be pulled from a remote repository to the local repository

GitHub Overview

- GitHub is a web application that does several things:
  - Hosts Git repositories on the cloud.
    - These typically serve as a central (master) remote repository for one or more developers.
  - Provides a nice web interface for browsing code in a Git repository.
  - Provides nice web-based tools to collaborate on code (centered around Git repos).
  - Provides tools to link code to external services (e.g. for building, testing, or publishing code).

- Signup here: https://github.com/join

Git & GitHub

1. Create a Git repository hosted on GitHub
2. Use Git to make a copy of this repository on your development machine using the command: **`git clone [url]`**
3. Start working in that directory as you wish
   - At any point, to print a summary of the current state of your work:
     `git status`

Git & GitHub

4. To commit a snapshot of your code:
   - In Git, committing is a two-step process:

   1. stage (i.e. mark as ready for commit) the files you want to commit.

   ```
   git add some_code.cpp
   ```

   2. commit the staged files.

   ```
   git commit -m "A short message describing this commit"
   ```

   - The `-m` option allows you to provide a short message to describe your commit, so you can get a quick sense for the commit when you look back on it later.
   - If you omit the `-m` option, Git will open a text editor for you so you can write a message to describe your commit.

Git & GitHub

5. Lastly save your work onto GitHub:

```
git push
```

- This synchronizes the remote repository on GitHub with your local repository, pushing any new commits you've made into the remote repo.

Useful Git Commands

- **`clone`** – copies an entire remote repo to the local machine
- **`log`** – prints the history of all commits made to the local repo
- **`status`** – prints a brief message describing the working state of the local repo
- **`diff`** – prints the actual differences between different versions of the local repo
  - By default, diff prints the difference between the working (i.e. current) code and the last commit.
- **`add`** – stages a file for commit
- **`commit`** – commits all the staged files
- **`push`** – synchronizes all commits from your local repo to a remote repo (e.g. your GitHub remote repo)