Final Exam Review

Trees, Heaps, Sorting, and Sets / Maps / Hash Tables

About This Review Session

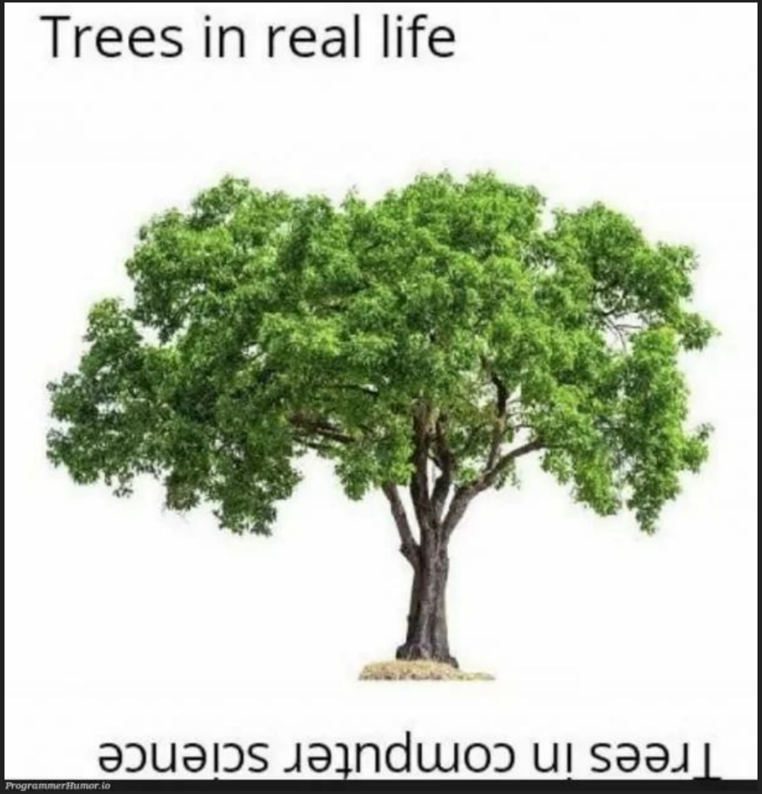
- Aimed at providing a high-level review for reference
- Not intended to be a comprehensive review of all the data structures and algorithms from this class
- Continuation of content covered by Aman

EXAM TOPICS AND EXPECTATIONS

Content Review

- Trees
- Heaps
- Sorting
- Sets/Maps/Hash Tables

Trees


Trees in real life
Trees in computer science
ProgrammerHumor.io

Tree Algorithms

Traversals are used to go through elements of a tree.

There's several types of traversals we've learned in this course:

- Preorder, inorder, and postorder traversals

We also learned about searches to find an element in a tree:

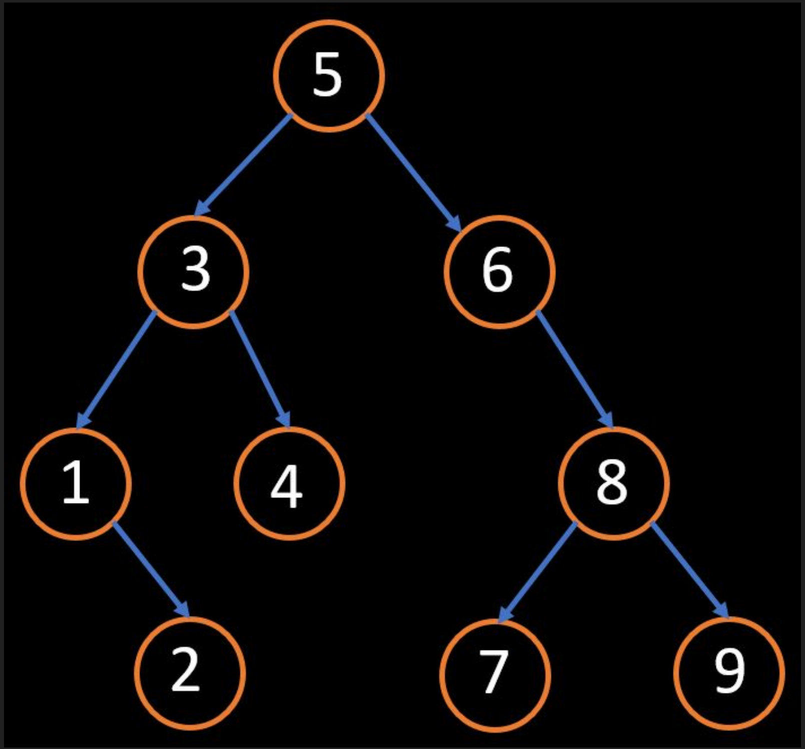
- Breadth-first (level-order) search and depth-first search

The time complexity for all of these are going to be O(n) because we expect to visit every node in the tree in the worst case.

Poll Question

Given the following binary tree, which of the following are valid permutations of the orders in which the nodes were accessed using the breadth first search algorithm? (Select all that apply) Note: Accessed means when any of a node's properties are used (value, left node, or right node) in the breadth first search.

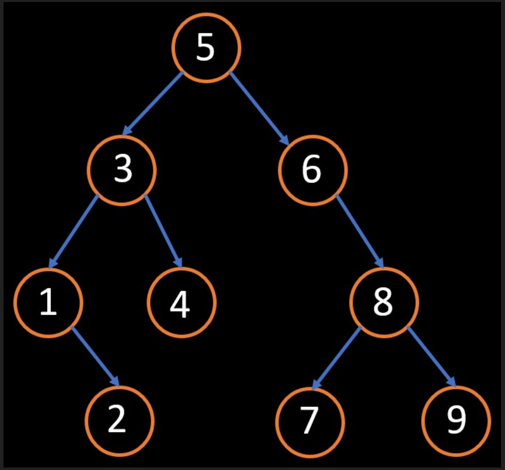
A. 5 3 6 1 4 8 2 7 9
B. 5 3 1 2 4 6 8 7 9
C. 5 6 3 8 4 1 9 7 2
D. 5 6 8 7 9 3 1 2 4



Hint: In this binary tree, there is no inherent left or right direction for the search algorithm. Use the properties of the breadth first search algorithm to determine which are valid permutations.

Poll Question

Given the following binary tree, which of the following are valid permutations of the orders in which the nodes were accessed using the breadth first search algorithm? (Select all that apply)



A. 5 3 6 1 4 8 2 7 9
B. 5 3 1 2 4 6 8 7 9
C. 5 6 3 8 4 1 9 7 2
D. 5 6 8 7 9 3 1 2 4

Breadth First Search (BFS)

- Adds the non-null children of the node that is accessed to a queue.
- Searches level by level.
- Never searches the node of a greater depth while there are nodes of lesser depth in the queue.

Poll Question

Why is Depth-First Search (DFS) less efficient than Breadth-First Search (BFS) for finding the minimum depth of a binary tree?

A. DFS always examines all nodes in the tree before finding the minimum depth.

B. DFS might explore more nodes even after encountering the shallowest node.

C. DFS is more complex to implement than BFS for tree traversal.

D. DFS requires more memory than BFS, making it less efficient.

Poll Question

Why is Depth-First Search (DFS) less efficient than Breadth-First Search (BFS) for finding the minimum depth of a binary tree?

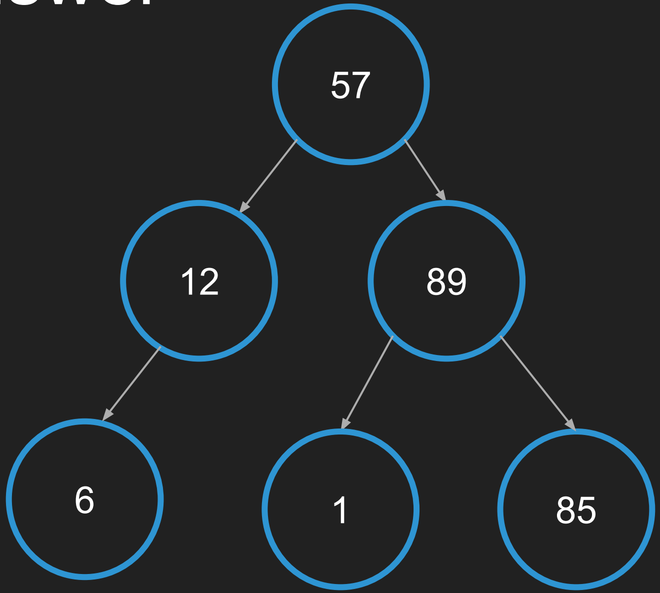
A. DFS always examines all nodes in the tree before finding the minimum depth.

B. DFS might explore more nodes even after encountering the shallowest node.

C. DFS is more complex to implement than BFS for tree traversal.

D. DFS requires more memory than BFS, making it less efficient.

Poll Question Answer

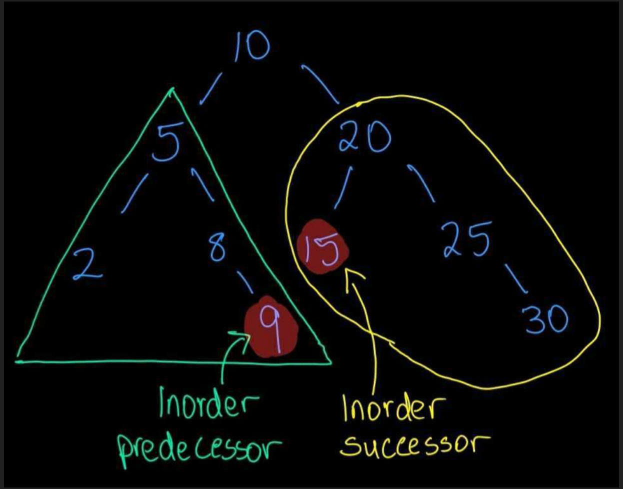


BFS: 57, 12, 89, 6, RETURN

DFS (preorder): 57, 12, 6, 89, 1, 85, RETURN

Inorder successors and predecessors

When you do an inorder traversal of a binary tree, the inorder predecessor is the node that lies behind the given node, and the inorder successor is the node that lies ahead of a given node.

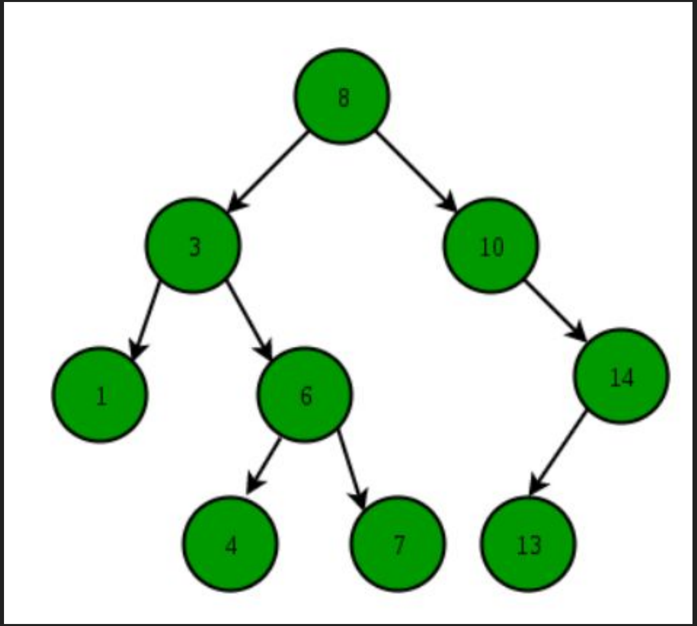


Inorder traversal:
2, 5, 8, 9, 10, 15, 20, 25, 30

The inorder **predecessor** of the 10 is **9**
The inorder **successor** of the 10 is **15**

What is a Binary Search Tree (BST)?

- The left subtree contains nodes with smaller keys
- The right subtree contains nodes with greater keys.
- All subtrees *must also be BSTs*.
- O(log(n)) when balanced for insertion, deletion, and removal
- O(n) for three operations if unbalanced.



https://www.geeksforgeeks.org/binary-search-tree-data-structure/

Poll Question

When deleting a node with two children in a BST, what can we do to find a replacement for the deleted node?

A) Select the smallest value from the right subtree.

B) Select the largest value from the right subtree.

C) Randomly choose a node from either subtree.

D) Replace it with the root of the tree.

Poll Question

When deleting a node with two children in a BST, what can we do to find a replacement for the deleted node?

A) Select the smallest value from the right subtree.

B) Select the largest value from the right subtree.

C) Randomly choose a node from either subtree.

D) Replace it with the root of the tree.

We know this as the **inorder successor**!

BST Search

- **Start at Root:** Begin the search from the root node.
- **Comparing Keys:** If the key is less than the root, search the left subtree; if more, search the right.
- **Stop Cases:**
  - Key is found in tree
  - Root is null so key is not in tree

```
Node* search(Node* root, int key) {
    // Base Cases: root is null or key is
present at root
    if (root == nullptr || root->data == key) {
        return root;
    }

    // Key is greater than root's key
    if (root->data < key) {
        return search(root->right, key);
    }

    // Key is smaller than root's key
    return search(root->left, key);
}
```

BST Insertion

- **Insertion Rule**: Left child is less than the parent, right child is greater.
- **Traversal for Insertion**: Start from the root, traverse to find the correct position.
- **Compare and Place**: Compare the new value with each node and place it accordingly.

```
Node* insert(Node* root, int value) {
    // If the tree is empty, return a new node
    if (root == nullptr) {
        return new Node(value);
    }

    // Otherwise, recur down the tree
    if (value < root->data) {
        root->left = insert(root->left, value);
    } else if (value > root->data) {
        root->right = insert(root->right,
value);
    }

    // Return the unchanged node pointer
    return root;
}
```

BST Deletion

- First, traverse down the tree to the node to delete (search operation)
- **Three Scenarios**
  - Deleting a leaf node
  - a node with one child
  - or a node with two children
- **Leaf Node Deletion:** Simply remove the node from the tree.
- **Single Child Node:** Replace the node with its child.
- **Two Children Node:** Find the inorder successor or predecessor and copy over the data. Then remove the inorder successor/predecessor.

```
Node* deleteNode(Node* root, int value) {
    if (root == nullptr) return root;

    if (value < root->data) {
        root->left = deleteNode(root->left, value);
    } else if (value > root->data) {
        root->right = deleteNode(root->right, value);
    } else {
        // Node with only one child or no child
        if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            return temp;
        } else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }

        // Node with two children: Get the inorder
successor
        Node* temp = findInorderSuccessor(root);
        root->data = temp->data;

        // Delete the inorder successor
        root->right = deleteNode(root->right,
temp->data);
    }
    return root;
}
```

Problem-solving: Minimum Depth of Binary Tree

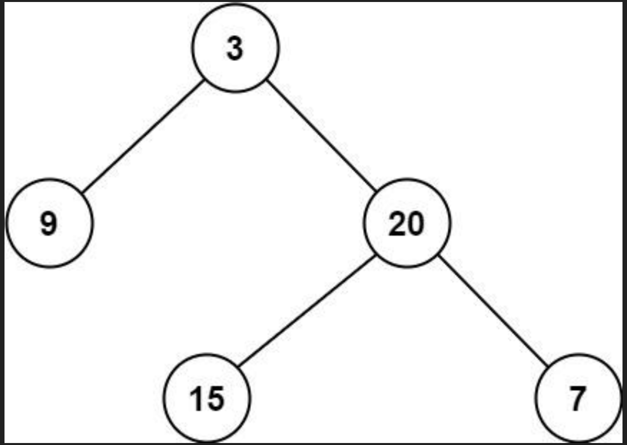
Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

Note: A leaf is a node with no children.

Example:

Input: root = [3,9,20,null,null,15,7]
Output: 2



https://leetcode.com/problems/minimum-depth-of-binary-tree/

- Use BFS and stop at the first leaf node
    - **Level-by-Level Traversal**: BFS traverses the tree level by level, starting from the root
    - **First Leaf Node Encountered:** First leaf encountered is guaranteed to be at the minimum depth because BFS does not go deeper until all nodes at the current depth are explored!

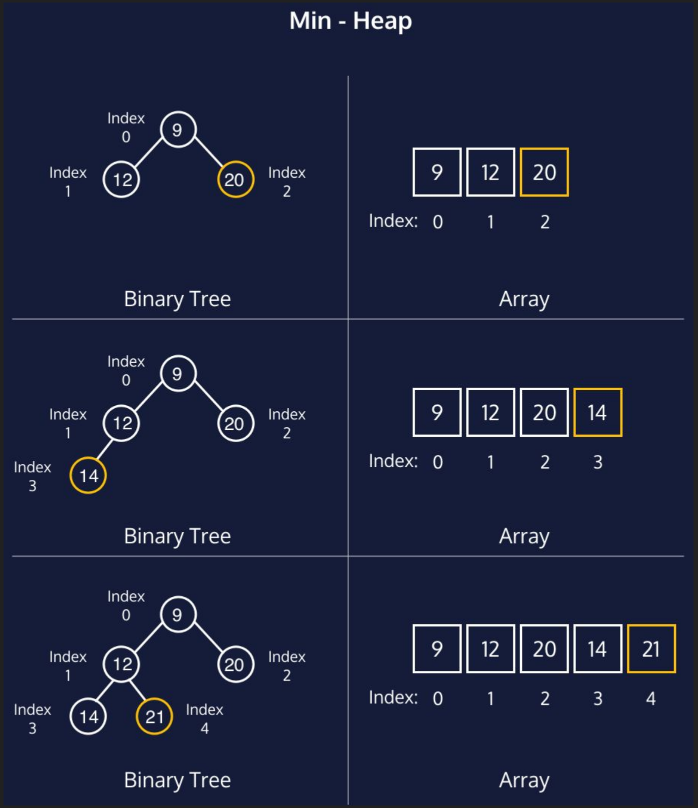
```cpp
int findMinDepth(Node* root) {
    if (root == nullptr) {
        return 0;
    }

    std::queue<std::pair<Node*, int>> q;
    q.push({root, 1});

    while (!q.empty()) {
        Node* node = q.front().first;
        int depth = q.front().second;
        q.pop();

        // Check if it is a leaf node
        if (node->left == nullptr && node->right ==
nullptr) {
            return depth;
        }

        // Add children to queue with increme
        if (node->left != nullptr) {
            q.push({node->left, depth + 1});
        }
        if (node->right != nullptr) {
            q.push({node->right, depth + 1});
        }
    }

    return 0; // This should never be reached
}
```

Heaps

HEAPS GETTING REPRESENTED AS TREE
INTERNALLY HEAPS GETTING STORED AS ARRAY
imgflip.com

Heaps

- **Binary Tree:** Heaps are complete binary trees
- **Heap Order:** Each node is greater (max-heap) or smaller (min-heap) than its children
- **Root Access:** The root is the largest (max-heap) or smallest (min-heap) element
- Implemented using an **array**
- Insertion, remove time complexity: **O(logn)**

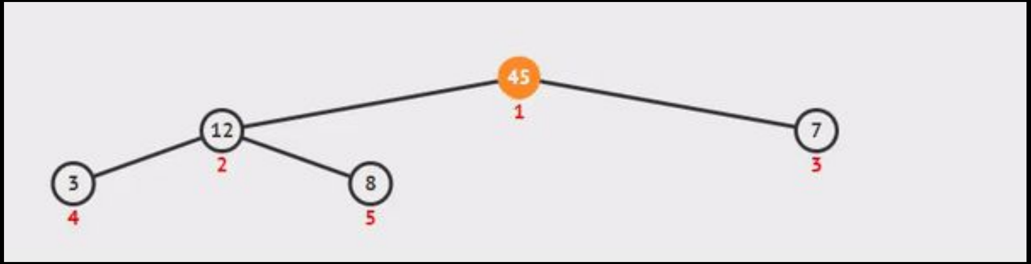


Insertion

```
function insertHeap(heap, element):
    // Step 1: Add the new element to the end of the heap
    heap.append(element)

    // Step 2: Heapify-up - move the new element to its correct position
    index = heap.size() - 1
    while index > 0:
        parentIndex = (index - 1) / 2

        // For max-heap, if the parent is smaller, swap. For min-heap, if the parent is larger,
swap.
        if (maxHeap and heap[index] > heap[parentIndex]) or (minHeap and heap[index] <
heap[parentIndex]):
            swap(heap[index], heap[parentIndex])
            index = parentIndex
        else:
            break
```

Deletion

```
function removeHeap(heap):
    // Step 1: Replace the root of the heap with the last element
    removedElement = heap[0]
    heap[0] = heap.pop()

    // Step 2: Heapify-down - move the new root to its correct position
    index = 0
    while index < heap.size():
        leftChildIndex = 2 * index + 1
        rightChildIndex = 2 * index + 2
        swapIndex = index

        // For max-heap, find the larger child. For min-heap, find the smaller child.
        if leftChildIndex < heap.size() and ((maxHeap and heap[leftChildIndex] > heap[swapIndex]) or (minHeap
and heap[leftChildIndex] < heap[swapIndex])):
            swapIndex = leftChildIndex

        if rightChildIndex < heap.size() and ((maxHeap and heap[rightChildIndex] > heap[swapIndex]) or (minHeap
and heap[rightChildIndex] < heap[swapIndex])):
            swapIndex = rightChildIndex

        // If no swap needed, break
        if swapIndex == index:
            break

        // Swap with the larger/smaller child
        swap(heap[index], heap[swapIndex])
        index = swapIndex

    return removedElement
```



Poll Question

In a binary max heap, after the root is deleted, which element typically replaces it before reheapifying?

a) The leftmost node of the deepest level in the heap

b) The rightmost node of the deepest level in the heap.

c) The child of the root with the highest root value.

d) The child of the root with the smallest root value.

Poll Question

In a binary max heap, after the root is deleted, which element typically replaces it before reheapifying?

a) The leftmost node of the deepest level in the heap

b) The rightmost node of the deepest level in the heap.

c) The child of the root with the highest root value.

d) The child of the root with the smallest root value.

This maintains the completeness property! This is also the node with index `len(heap) - 1`.

Priority Queue in C++

- STL implementation of heap

```
std::priority_queue<int> maxHeap;  // Default max-heap of integers
std::priority_queue<int, std::vector<int>, std::greater<int>> minHeap;  // Min-heap of integers
```

- Custom comparator!

```
auto minAgeComparator = [](const Person& person1, const Person& person2) {
      return person1.age > person2.age;  // Compare ages in ascending order
};
std::priority_queue<Person, std::vector<Person>,decltype(minAgeComparator)> minAgeHeap(minAgeComparator);
```

- Important functions
  - push() // insertion
  - top() // access top element
  - pop() // remove top element
  - empty() // check if empty
  - size() // return size

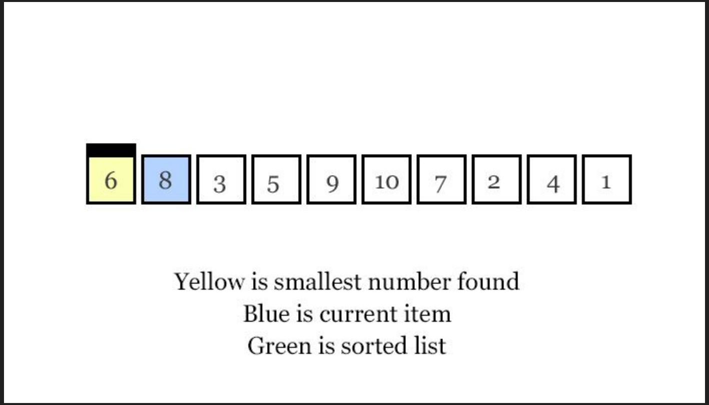
Sorting

CS Student: *uses Bubble Sort*
CS Prof:
This little maneuver is gonna cost us n^2

Selection Sort

- Repeatedly selects the minimum element from the unsorted portion and swaps it with the item at the beginning of the unsorted portion.
- Worst case time complexity: O(n^2)
- Worst case space complexity: O(1)

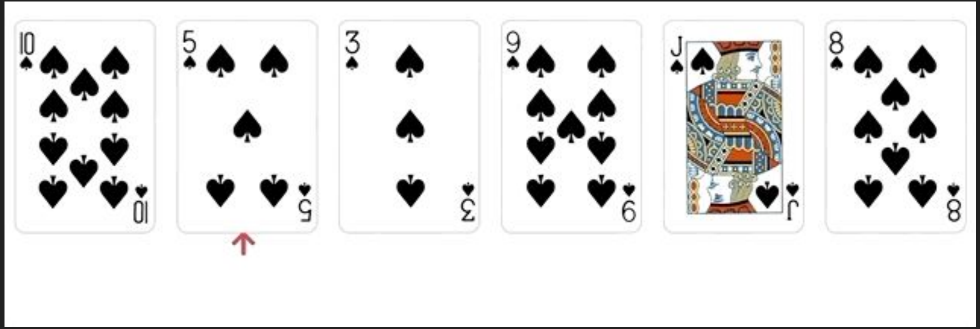
```
function selectionSort(arr):
    n = length(arr)
    for i from 0 to n - 1:
        minIndex = i
        for j from i + 1 to n:
            if arr[j] < arr[minIndex]:
                minIndex = j
        swap(arr[i], arr[minIndex])
```



Insertion Sort

- Iterate through the unsorted portion, comparing each element to the elements in the sorted portion and placing it in its correct position
- Worst case time complexity: O(n^2)
- Worst case space complexity: O(1)

```
function insertionSort(arr):
    n = length(arr)
    for i from 1 to n - 1:
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j = j - 1
        arr[j + 1] = key
```

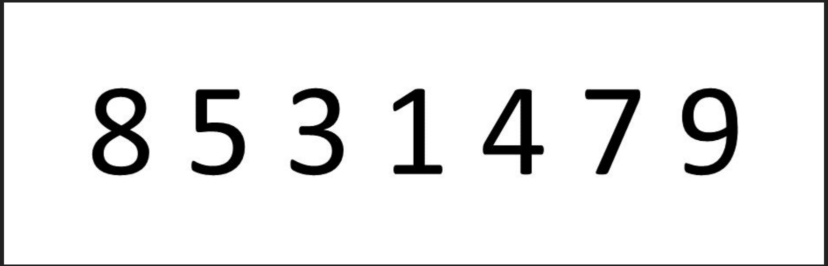


^ descending order sort!

Bubble Sort

- **R**epeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order
- Worst case time complexity: O(n^2)
- Worst case space complexity: O(1)

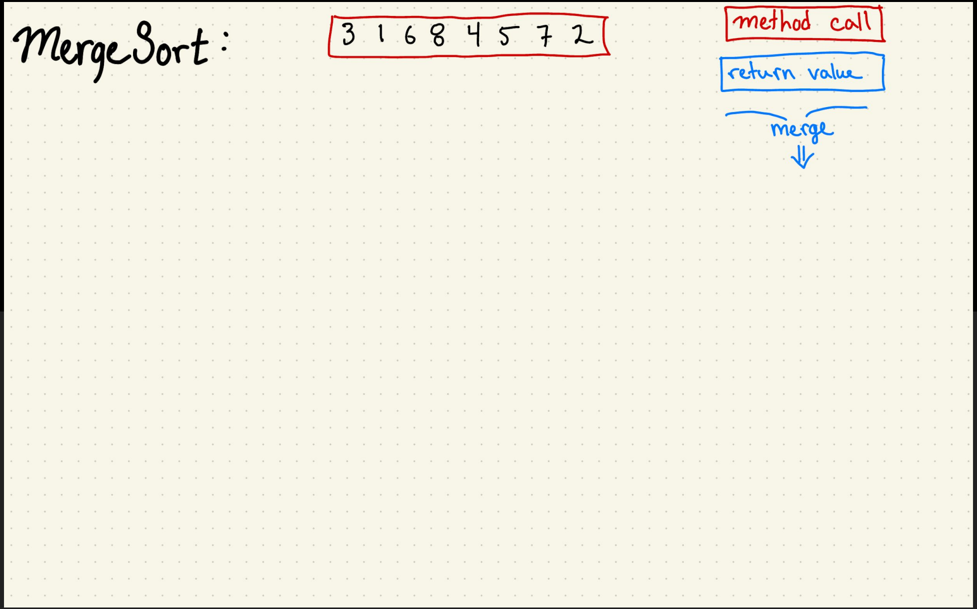
```
function bubbleSort(arr):
    n = length(arr)
    for i from 0 to n - 1:
        for j from 0 to n - 1 - i:
            if arr[j] > arr[j + 1]:
                swap(arr[j], arr[j + 1])
```



Merge Sort

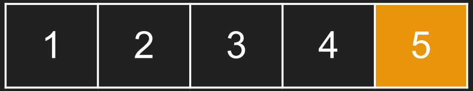
- **Divide-and-conquer:** Divides the unsorted list into ‘n’ sublists, each containing one element, and then repeatedly merges sublists to produce new sorted sublists until there is only one sublist remaining.
- Worst case time complexity: O(nlogn)
- Worst case space complexity: O(n)
    - Requires additional memory for temporary storage

```
function mergeSort(arr):
    if length(arr) <= 1:
        return arr  // Base case: return if the list has zero or one element
    mid = length(arr) / 2
    left = mergeSort(arr[0 to mid-1])  // Recursively sort the left half
    right = mergeSort(arr[mid to end])  // Recursively sort the right half
    return merge(left, right)  // Merge the sorted left and right halves

function merge(left, right):
    result = []  // Create an empty list to store the merged result
    while left is not empty and right is not empty:
        if left[0] <= right[0]:
            result.append(left[0])
            left = left[1:]
        else:
            result.append(right[0])
            right = right[1:]
    result.extend(left)  // Append remaining
    result.extend(right)
    return result
```

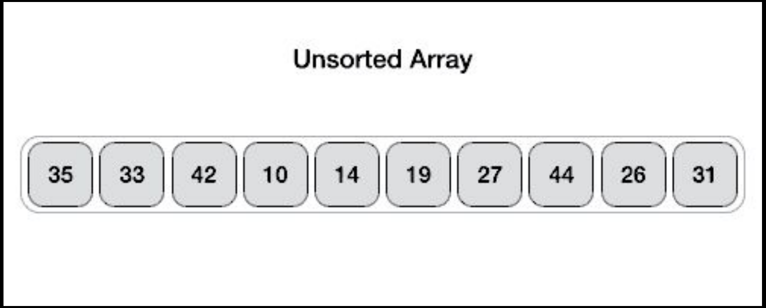


Quick Sort

- **Divide-and-conquer:** Select a "pivot" element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot
    - Repeat recursively
- Worst case time complexity: O(n^2)
    - Bad pivot will cause an uneven partition!
- **Average case time complexity: O(nlogn)**
- Worst case space complexity: O(logn)
    - Recursive stack



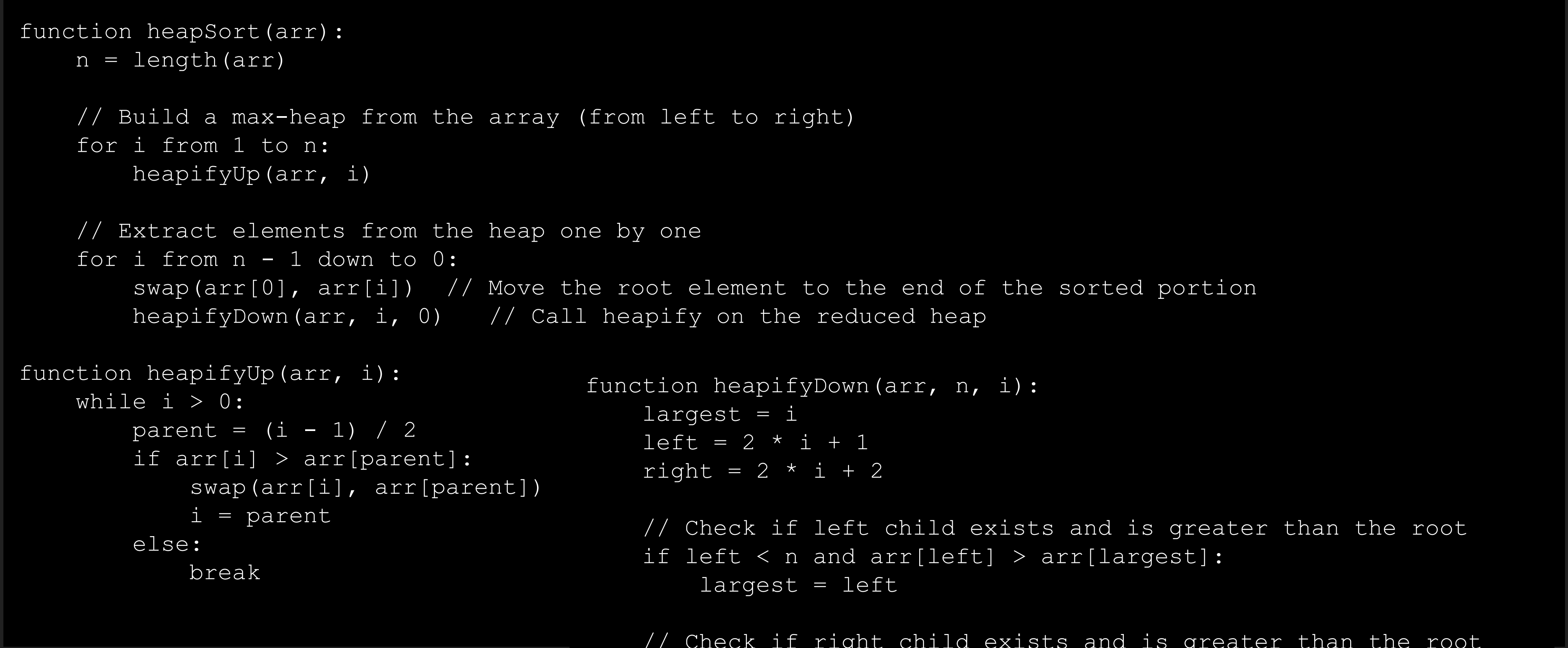
```
function quicksort(arr, low, high)
    if low < high
        pivotIndex = partition(arr, low, high)
        quicksort(arr, low, pivotIndex - 1)  // Recursively sort the left part
        quicksort(arr, pivotIndex + 1, high) // Recursively sort the right part

function partition(arr, low, high)
    pivot = arr[high]  // Choose the rightmost element as pivot
    left = low
    right = high - 1

    while true
        while left <= right and arr[left] < pivot
            left = left + 1
        while left <= right and arr[right] > pivot
            right = right - 1
        if left >= right
            break
        swap arr[left] and arr[right]

    swap arr[left] and arr[high]  // Move pivot to middle
    return left  // Return the partitioning index
```



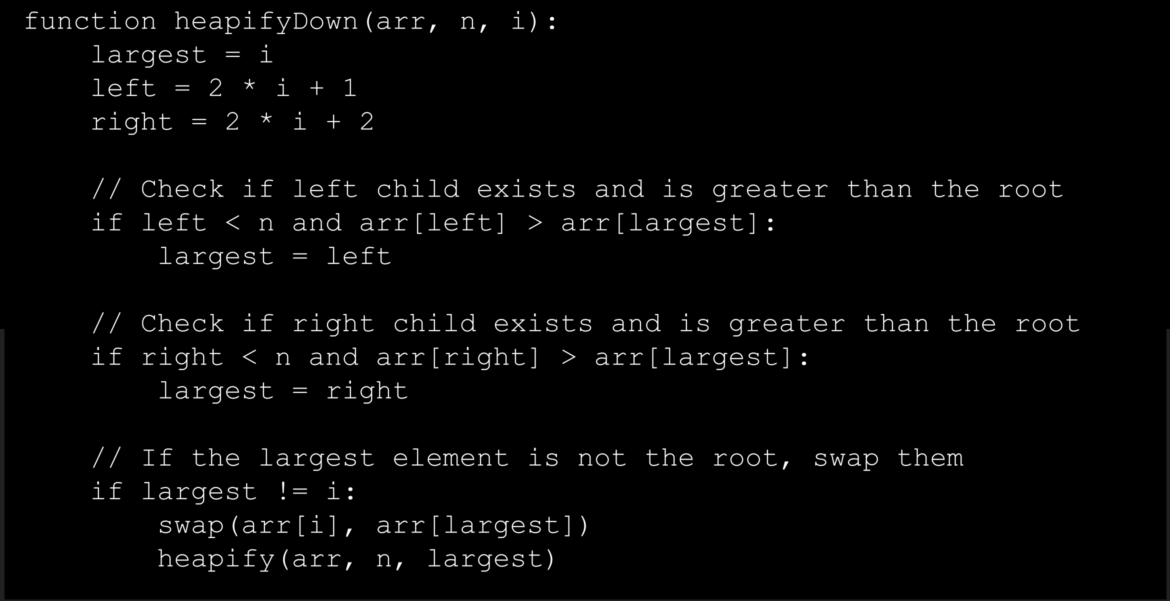
Heap Sort

- Builds a max-heap from the input array and then repeatedly removes the root element and add it to the sorted section
- Worst case time complexity: O(nlogn)
- Worst case space complexity: O(1)
  - This may be surprising! We build the heap in-place so we do not need extra storage.

10	4	8	5	12	2	6	11	3	9	7	1



```
function heapSort(arr):
    n = length(arr)

    // Build a max-heap from the array (from left to right)
    for i from 1 to n:
        heapifyUp(arr, i)

    // Extract elements from the heap one by one
    for i from n - 1 down to 0:
        swap(arr[0], arr[i])  // Move the root element to the end of the sorted portion
        heapifyDown(arr, i, 0)   // Call heapify on the reduced heap

function heapifyUp(arr, i):
    while i > 0:
        parent = (i - 1) / 2
        if arr[i] > arr[parent]:
            swap(arr[i], arr[parent])
            i = parent
        else:
            break
```



```
function heapifyDown(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    // Check if left child exists and is greater than the root
    if left < n and arr[left] > arr[largest]:
        largest = left

    // Check if right child exists and is greater than the root
    if right < n and arr[right] > arr[largest]:
        largest = right

    // If the largest element is not the root, swap them
    if largest != i:
        swap(arr[i], arr[largest])
        heapify(arr, n, largest)
```

Algorithm	TLDR	Complexity
Selection Sort	Extract smallest one by one	$O(n^2)$
Insertion Sort	Place next one where it belongs	$O(n^2)$
Bubble Sort	Swap adjacent until sorted	$O(n^2)$
Merge Sort	Split in half until you can’t, then put pieces together	O(nlogn)
Quicksort	Choose pivot, organize around pivot; repeat for each half	O(nlogn)*
Heap Sort	Add all to heap, then remove from heap	O(nlogn)

Poll Question

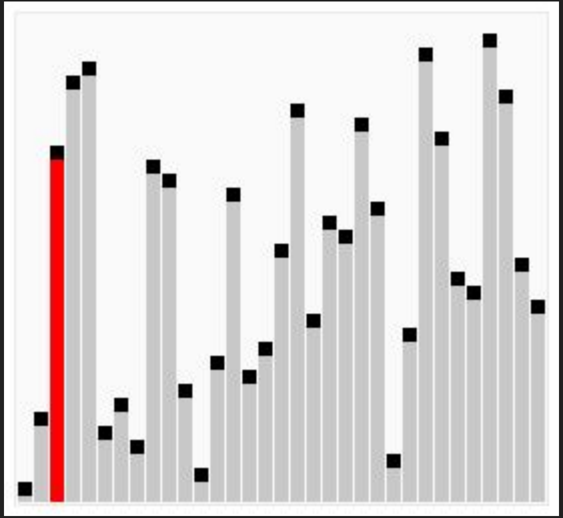
Which sorting algorithm is this?

A. Heap sort

B. Quick sort

C. Merge sort

D. Bubble sort



Notice how the biggest element bubbles to the top the first loop! Then the second biggest element in the second loop!

Poll Question

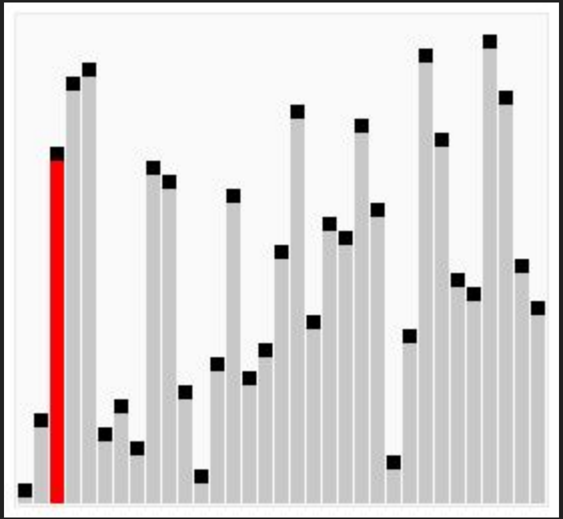
Which sorting algorithm is this?

A. Heap sort

B. Quick sort

C. Merge sort

D. Bubble sort



Notice how the biggest element bubbles to the top the first loop! Then the second biggest element in the second loop!

Poll Question

Which sorting algorithm is this?

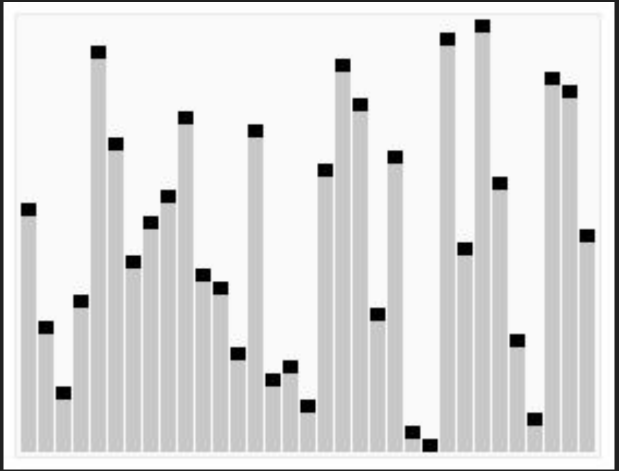
A. Heap sort

B. Quick sort

C. Merge sort

D. Bubble sort

Notice how it chooses a pivot and does divide and conquer!



Poll Question

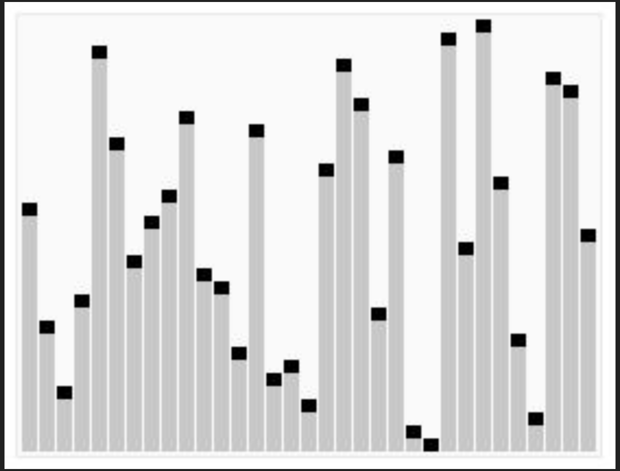
Which sorting algorithm is this?

A. Heap sort

B. Quick sort

C. Merge sort

D. Bubble sort



Notice how it chooses a pivot and does divide and conquer!

Poll Question

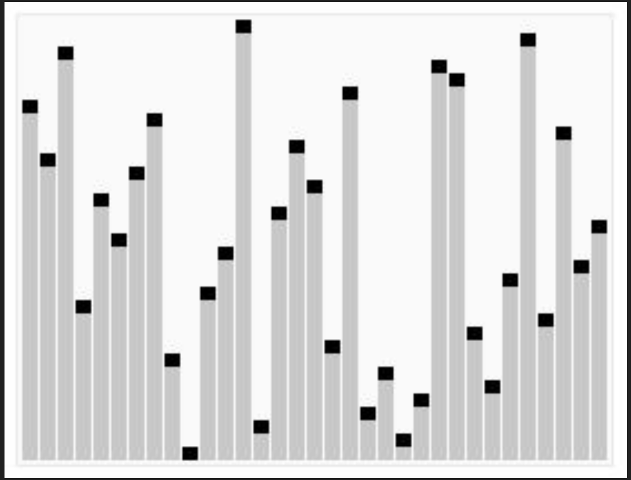
Which sorting algorithm is this?

A. Heap sort

B. Quick sort

C. Merge sort

D. Bubble sort



It’s hard to see with this visualization that a heap is created, but notice how it repeatedly takes the first element from the heap and then heapifies!

Poll Question

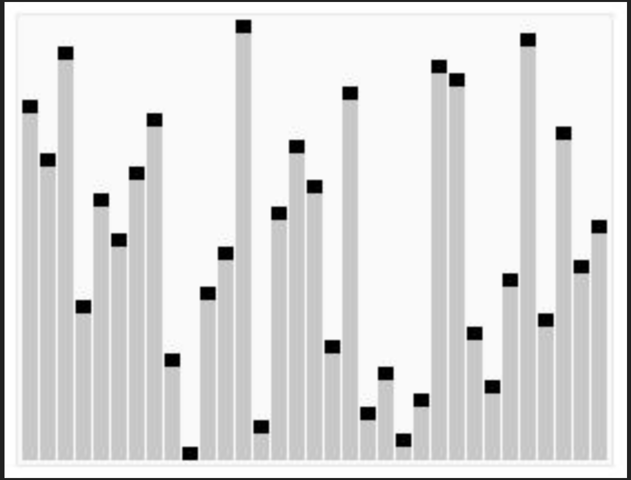
Which sorting algorithm is this?

A. Heap sort

B. Quick sort

C. Merge sort

D. Bubble sort



It’s hard to see with this visualization that a heap is created, but notice how it repeatedly takes the first element from the heap and then heapifies!

Sets / Maps / Hash Tables

Linear Probing
Separate Chaining

Sets

- Sets are a collection that do not contain duplicate elements
- Properties:
    - Not indexed
    - No inherent ordering
    - Allow membership testing (find, insert, remove)
- In C++, std::set internally uses a Red-Black Tree and std::unordered_set uses a hash table
- Use cases:
    - Maintaining a unique element collection
    - Performing set operations (union, intersection, difference)

Poll Question

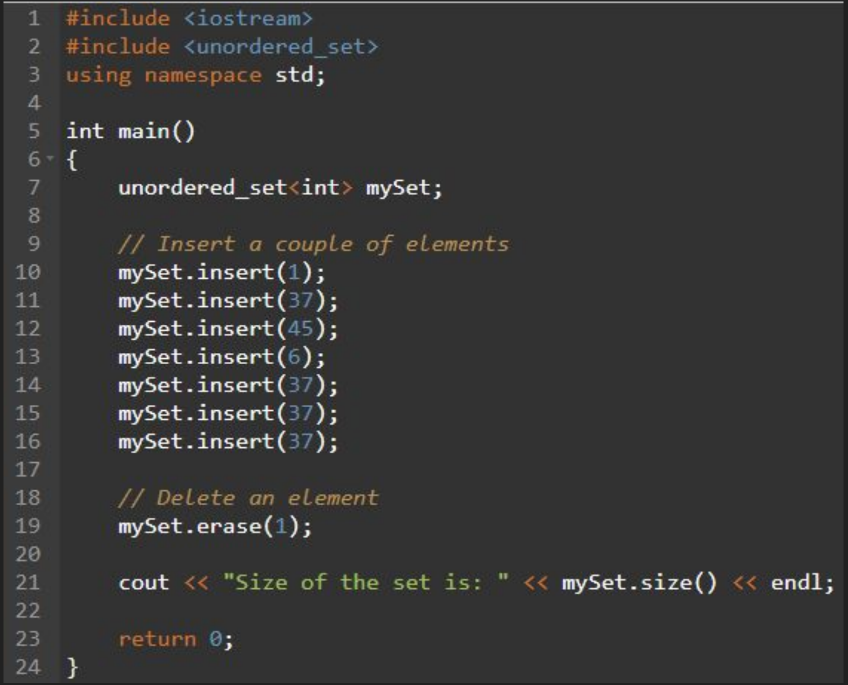
What is the output of the following code?

A. Size of the set is: 4

B. Size of the set is: 5

C. Size of the set is: 3

D. This wouldn't compile.



```cpp
#include <iostream>
#include <unordered_set>
using namespace std;

int main()
{
    unordered_set<int> mySet;

    // Insert a couple of elements
    mySet.insert(1);
    mySet.insert(37);
    mySet.insert(45);
    mySet.insert(6);
    mySet.insert(37);
    mySet.insert(37);
    mySet.insert(37);

    // Delete an element
    mySet.erase(1);

    cout << "Size of the set is: " << mySet.size() << endl;

    return 0;
}
```

https://onlinegdb.com/rJR1blruw

Poll Question

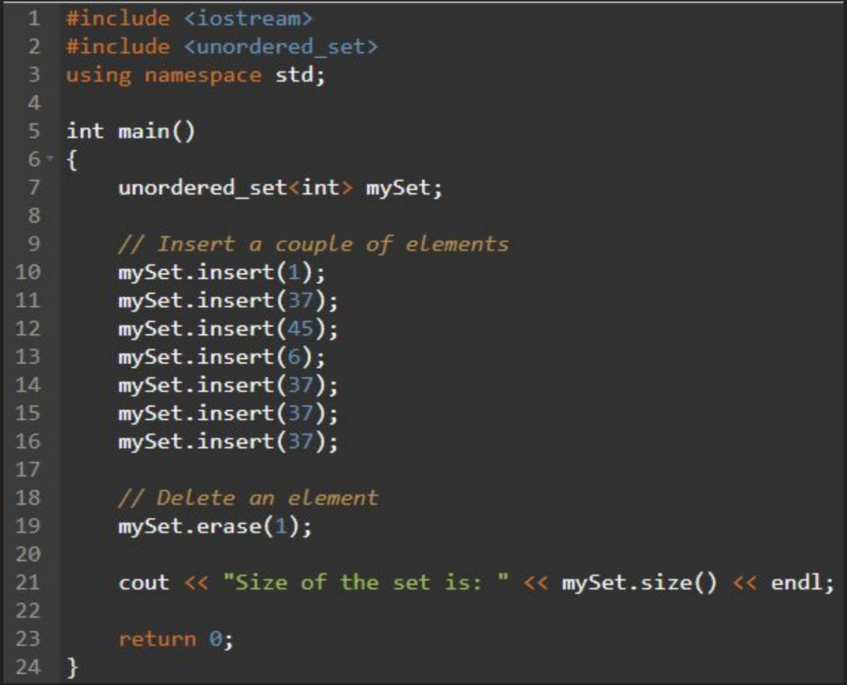
What is the output of the following code?

A. Size of the set is: 4

B. Size of the set is: 5

C. Size of the set is: 3

D. This wouldn't compile.



```cpp
#include <iostream>
#include <unordered_set>
using namespace std;

int main()
{
    unordered_set<int> mySet;

    // Insert a couple of elements
    mySet.insert(1);
    mySet.insert(37);
    mySet.insert(45);
    mySet.insert(6);
    mySet.insert(37);
    mySet.insert(37);
    mySet.insert(37);

    // Delete an element
    mySet.erase(1);

    cout << "Size of the set is: " << mySet.size() << endl;

    return 0;
}
```

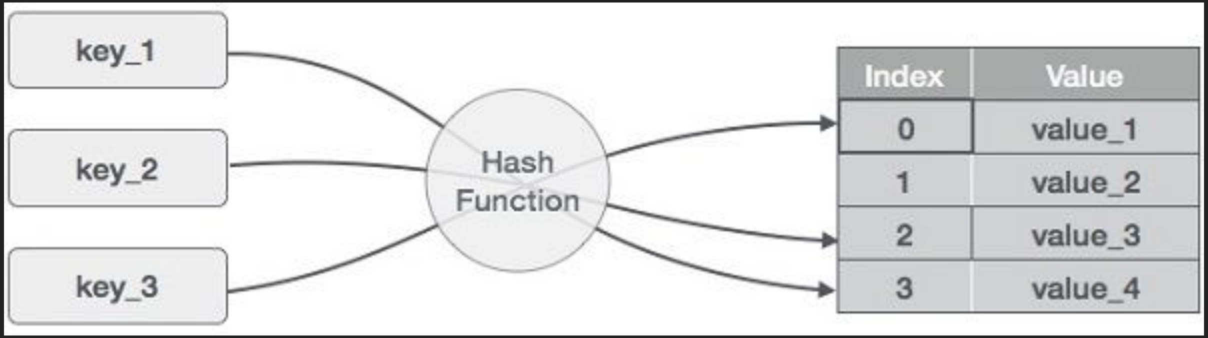
https://onlinegdb.com/rJR1blruw

Hash Tables

Hash tables are a non-ordered data structure that store key-value pairs

A **hash function** is used to compute the index for data

- Should be able to evenly distribute the data
- Should be easy to compute (constant time)
- Should not be random



Load Factor

Load factor = number of total elements / size of the array

- Higher load factor => higher collision rate => slower hash table
- We dynamically change the array size to maintain a load factor

Poll Question

If we want to maintain a load factor of 0.5 and have an array of current capacity 20, when do we increase the capacity of the array?

a. When the array has 5 elements

b. When the array has 10 elements

c. When the array has 11 elements

d. When the array has 15 elements

Poll Question

If we want to maintain a load factor of 0.5 and have an array of current capacity 20, when do we increase the capacity of the array?

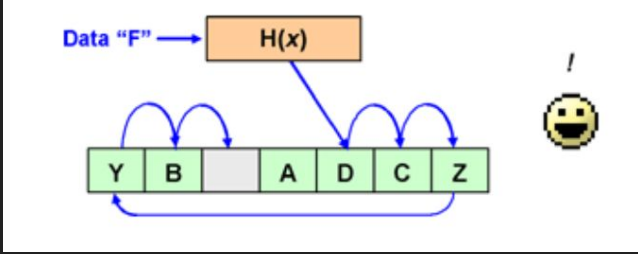
a. When the array has 5 elements

b. When the array has 10 elements

c. When the array has 11 elements

d. When the array has 15 elements

Collision Resolution



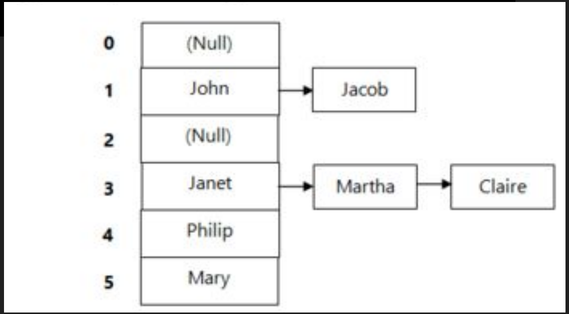
- Open addressing
  - When there's a conflict, "probe" alternative locations
  - Linear probing: probe subsequent slots one after another
    - Suffers from clustering
  - Quadratic probing: distance between probes increases quadratically
- Chaining
  - Each index in the array stores a list
  - If a conflicts arises, add the element to the end of the list
  - Does not suffer from clustering

Separate Chaining Pseudocode

```
Class HashMap
    Define map as a vector of vectors
    Define capacity as an integer
    Define size as an integer
```

```
Function hash(key)
    Return len(key) modulo capacity
```

```
Function search(key)
    bucketIndex = hash(key)
    bucket = map[bucketIndex]

    For each element in bucket
        If element equals key
            Return True

    Return False
```

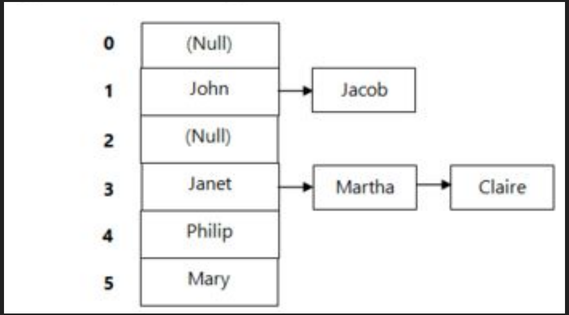


- **Locate the Bucket:** Determine the bucket index for the key to be removed via the hash function
  Bucket Traversal: Linearly traverse the elements in the identified bucket until a match is found
  - Separate chaining!
- Return **False** at the end when match is not found

Separate Chaining Pseudocode

```
   Function insert(key)
       If size >= capacity * 0.75 // max load factor
            resizeAndRehash(capacity * 2)

      bucketIndex = hash(key)
       bucket = map[bucketIndex]

       For each element in bucket
           If element equals key
               Return // Key already exists, do not
insert

       Add key to the end of the bucket
       Increment size

   Function resizeAndRehash(newCapacity)
        Define newMap as a vector of vectors with size
newCapacity
        For each bucket in map
            For each key in bucket
                newBucketIndex = hash(key)
                Add element to end of
newMap[newBucketIndex]

        map = newMap
        capacity = newCapacity
```

- **Bucket Index:** Calculate index using hash function
- **Load Factor Check:** Resize to twice the size and rehash if load factor > 0.75
- **Avoid Duplicates:** Check if key already exists in bucket
- **Insert Key:** Add key to bucket if unique and increment size



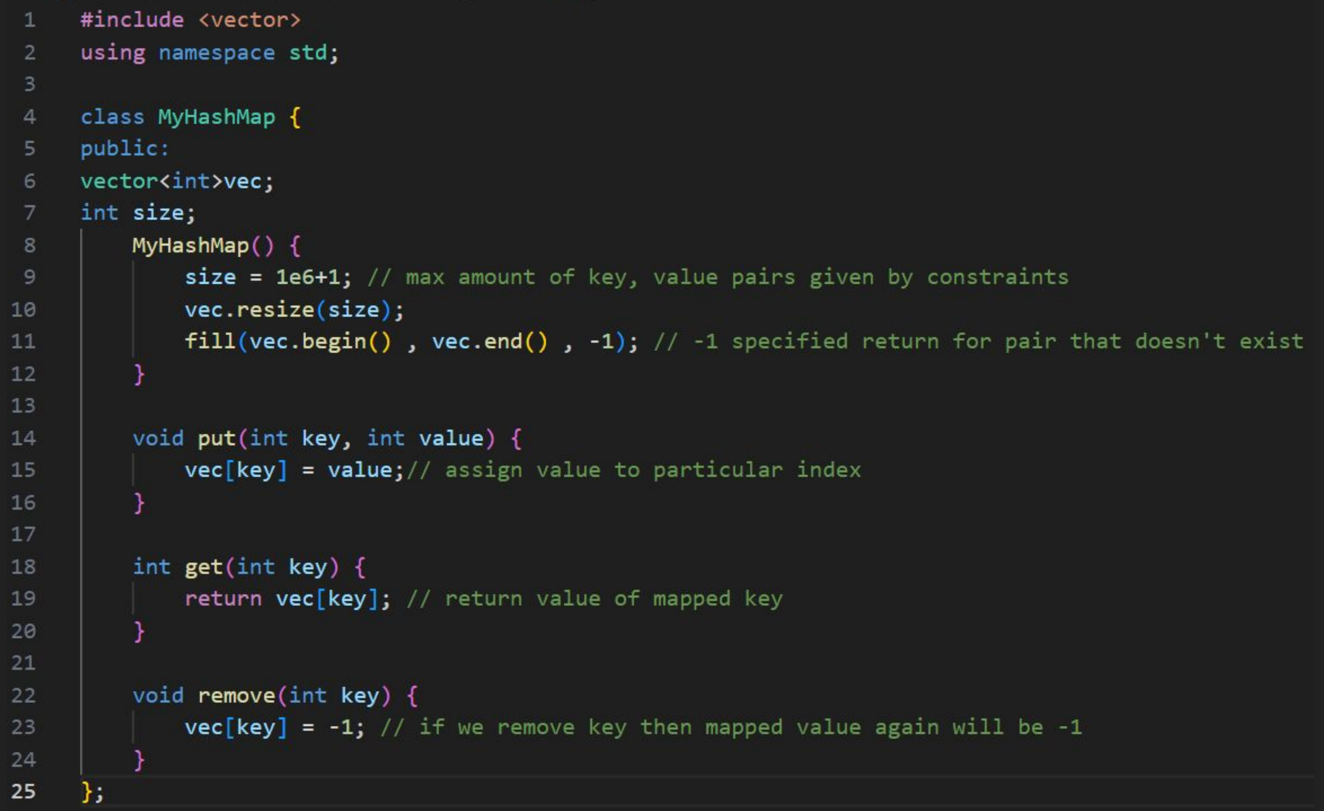
Participation Activity: Live Coding!

LeetCode 706. Design HashMap

Participation Activity: Solution

Operations are:

- O(1) time
- O(n) space where n is the number of key, value pairs



```cpp
#include <vector>
using namespace std;

class MyHashMap {
public:
vector<int>vec;
int size;
    MyHashMap() {
        size = 1e6+1; // max amount of key, value pairs given by constraints
        vec.resize(size);
        fill(vec.begin() , vec.end() , -1); // -1 specified return for pair that doesn't exist
    }

    void put(int key, int value) {
        vec[key] = value;// assign value to particular index
    }

    int get(int key) {
        return vec[key]; // return value of mapped key
    }

    void remove(int key) {
        vec[key] = -1; // if we remove key then mapped value again will be -1
    }
};
```

Problem-solving: Frequency Sort

Given a string s, sort it based on the frequency of characters in descending order. If multiple characters have the same frequency, they can be returned in any order.

Example 1:

- Input: s = "tree"
- Output: eert or eetr
- Explanation: 'e' appears twice while 'r' and 't' both appear once. So 'e' must appear before 'r' and 't'.

- **Step 1:** Count character frequencies using a hash map
- **Step 2:** Sort characters by frequency in decreasing order
- **Step 3:** Construct the result strong

```
string frequencySort(string s) {
    // Create a hashmap to store frequency of each character
    unordered_map<char, int> frequencyMap;
    for (char c : s) {
        frequencyMap[c]++;
    }

    // Convert the hashmap into a vector of pairs (frequency,
character)
    vector<pair<int, char>> frequencyVector;
    for (auto& pair : frequencyMap) {
        frequencyVector.emplace_back(pair.second, pair.first);
    }

    // Sort the vector based on frequency in descending order
    sort(frequencyVector.begin(), frequencyVector.end(),
        [](const pair<int, char>& a, const pair<int, char>& b) {
            return a.first > b.first;
        });

    // Build the result string based on sorted frequencies
    string result;
    for (auto& pair : frequencyVector) {
        result += string(pair.first, pair.second);
    }

    return result;
}
```

Any Exam Questions? :)

You after taking this exam 😎
YouG0Tthis

If you have 3 minutes, please please fill out the TA Feedback Form!

- If you think a TA did a great job, let Aman know so they can be awarded!
- We would also love to know how to improve our course and our teaching
- Make sure you're logged into your ufl account