CSE 250
Lecture 38
Final Review
Day 2
Edge List Summary

- **addEdge, addVertex**: $O(1)$
- **removeEdge**: $O(1)$
- **removeVertex**: $O(1) + O(\text{vertex.inEdges})$
- **vertex.outEdges, vertex.inEdges, vertex.incidentEdges**: $O(m)$
  - (total cost to visit all out/in/incident edges)
- **vertex.edgeTo**: $O(m)$
- **Space Used**: $O(n+m)$
Add an Adjacency List

class DirectedGraphV3[LV, LE] {
    def addEdge(orig: Vertex, dest: Vertex, label: LE): Edge = {
        val edge = new Edge(label)
        edge._listNode = edges.append(edge)
        orig._outEdges.append(edge)
        dest._inEdges.append(edge)
        return edge
    }
    class Vertex(_label: LV){
        val _outEdges: LinkedList[Edge]
        val _inEdges: LinkedList[Edge]
        // ...
    }
}
Adjacency List Summary

- **addEdge, addVertex**: $O(1)$
- **removeEdge**: $O(1)$
- **removeVertex**: $O(\text{deg}(\text{vertex}))$
- **vertex.outEdges**: $O(|\text{outEdges}|)$ to visit all outEdges
  - Same for vertex.inEdges, vertex.incidentEdges
- **vertex.edgeTo**: $O(|\text{outEdges}|)$
- **Space Used**: $O(n+m)$
Binary Search Trees
Tree Terminology

- Rooted directed tree
  - **root** is the topmost vertex
  - EmptyTree contains 0 vertices, null for mutable tree.
- **Parent** references one or more children
  - **leaf** vertex: Vertex with zero children
- **Depth** of a vertex
  - Number of edges in the path from the root to the vertex
- **Level** of a vertex
  - Depth + 1
Tree Terminology

- The **size** of a tree
  - the number of vertices
  - Typically represented as $n$
- The **depth** of a tree - the maximum depth of any node
  - Typically represented as $d$
- The **height** of a vertex
  - The maximum number of edges from vertex to any leaf
Tree Terminology

- A **binary tree** is a tree where
  - every vertex has $\leq 2$ children
- A **full binary tree** is a tree where
  - all leaf vertices are at the lowest depth of the tree
  - Every vertex has either 0 or 2 children
- Depth of a full tree: $d$
- Size of a full tree: $n = \sum_{i=0}^{d} 2^i = 2^{d+1} - 1 = O(2^d)$
Tree Traversals

- **Pre-order (top-down)**
  - visit **root**, visit **left** subtree, visit **right** subtree
- **In-order**
  - visit **left** subtree, visit **root**, visit **right** subtree
- **Post-order (bottom-up)**
  - visit **left** subtree, visit **right** subtree, visit **root**
Computing the height of a tree

- Height (depth) of a tree = height of the root

\[
d(\text{root}) = \begin{cases} 
-1 & \text{if the tree is empty} \\
1 + \max(d(\text{root.left}), d(\text{root.right})) & \text{otherwise}
\end{cases}
\]
Priority Queues / Heaps
Priority Queue

- PriorityQueue[A: Ordering]
  - enqueue(v: A): Unit
    - Insert value v into the priority queue
  - head: A
    - Retrieve the highest-priority value in the priority queue
  - dequeue: A
    - Remove the highest-priority value from the priority queue
(Binary) Heap

- **Idea**: Keep the priority queue “kinda” sorted
  - Keep larger items closer to the front of the list
  - Trade off between...
    - Moving larger elements forward
    - Leaving some elements out-of-order
- **Challenge**: How track which elements are already sorted?
- **Inspiration**: Trees
(Binary) Heaps

- A (binary) heap is a tree-like structure with the properties:
  - A complete (binary) tree
  - Each vertex is “non-increasing” relative to its children
    - Strictly decreasing if no duplicates present
- A complete (binary) tree is a tree where
  - Each node has at most 2 children
  - Every level except for the last is full
    - Nodes in the last level are as far left as possible
Heaps

• What is the max depth of a binary heap?
  - Level 1: 1 value
  - Level 2: up to 2 values
  - Level 3: up to 4 values
  - Level 4: up to 8 values
  - Level i: up to $2^i$ values

$$n = \sum_{i=1}^{\ell_{max}} 2^i = 2^{\ell_{max}+1} - 1$$

$$\ell_{max} = \log(n + 1) - 1$$
Heap Methods

- isEmpty: Boolean
- length: Int
- head: A
- pushHeap(elem: A)
- popHeap: A
Heap Methods: pushHeap

- **Idea**: Insert into the next available location and then fix up
  - Insert at next available location (call it `current`)
  - While `current` isn’t root and `parent < current`
    - Swap `current` and `parent`
    - Repeat with `current = parent`
Heap Methods: popHeap

- **Idea**: Fill root with value in last filled location and then fix down
  - Start with the root (call it `current`)
  - While `current` isn’t a leaf and there’s a `child < current`
    - Swap `current` and the larger `child`
    - Repeat with `current = child`
Storing Heaps in Memory

- **Observations:**
  - Each layer has a maximum size
  - Each layer grows left-to-right
  - Only the last layer grows

- **Idea:** Use an array to store the heap
Analysis

- pushHeap
  - Append to end of ArrayBuffer
    - Amortized $O(1)$
  - fixUp
    - $\log(n)$ steps, each $O(1) = O(\log(n))$
- popHeap
  - Remove end of ArrayBuffer
    - $O(1)$
  - fixDown
    - $\log(n)$ steps, each $O(1) = O(\log(n))$

$O(\log(n))$ amortized
$O(n)$ worst-case
Binary Search Trees
Binary Search Tree

- Store key/value pairs \( T = (K, V) \)
  - Require an Ordering\([K]\)
- Enforce constraints:
  - No duplicate keys
  - For every vertex \( v_L \) in the left subtree of \( v_1 \),
    - \( v_L.\text{key} < v_1.\text{key} \)
  - For every vertex \( v_R \) in the right subtree of \( v_1 \),
    - \( v_R.\text{key} > v_1.\text{key} \)
BST Mutations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>insert</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(d)$</td>
</tr>
</tbody>
</table>
Tree Depth vs Size

height(left) $\approx$ height(right)

d = $O(\log(n))$

height(left) $\ll$ height(right)

d = $O(n)$
“Balanced” Trees

• Faster search: Want height(left) ≈ height(right)
  – Make it more precise: |height(left) - height(right)| ≤ 1
  – (left, right height differ by at most 1)

• **Question**: How do we keep the tree balanced?
  – Option 1: Keep left/right subtrees within +/- 1 of each other
    • Add a field to track the “imbalance factor”
  – Option 2: Ensure leaves are at a minimum depth of \( d / 2 \)
    • Add a designation marking each node as red or black
Rebalancing Trees
Rebalancing Trees

Rotate(A, B)
AVL Trees

• An AVL tree (Adelson-Velsky and Landis) is a BST where every node is “depth-balanced”
  - \(|\text{depth(left subtree)} - \text{depth(right subtree)}| < 1\)
• define \(\text{balance}(v) = \text{height}(v.\text{right}) - \text{height}(v.\text{left})\)
  - Maintain balance(v) \(\in \{-1, 0, 1\}\)
    • balance(v) = 0 \(\rightarrow\) “v is balanced”
    • balance(v) = -1 \(\rightarrow\) “v is left-heavy”
    • balance(v) = 1 \(\rightarrow\) “v is right-heavy”

If the balance constraint is obeyed, the tree must have \(\Omega(2^d)\) nodes (\(d = \log(n)\))
Maintaining Balance

- Enforcing height-balance is too strict
  - May require “unnecessary” rotations
- Weaker restriction:
  - Balance the depth of EmptyTree nodes
  - If a, b are EmptyTree nodes:
    - $\text{depth}(a) \geq \left( \frac{\text{depth}(b)}{2} \right)$
    - or
    - $\text{depth}(b) \geq \left( \frac{\text{depth}(a)}{2} \right)$
Balancing Empty Node Depth

\[
d/2 = \log(n) \\
d = 2\log(n) = O(\log(n))
\]

Must be full \((2^{d/2} \text{ nodes})\)
Red-Black Trees

• Color each node red or black
  1) # of black nodes from each empty to root must be identical
  2) Parent of a red node must be black

• On Insertion (or deletion)
  – Inserted node is red (won’t change # of black nodes)
  – “Repair” violations of rule 2 by rotating or recoloring
    • Each repair guarantees rule 1 is preserved
    • Each repair creates at most 1 new violation of rule 2 at the parent.
TreeSet[A: Ordering]

- **add(a: A): Unit**
  
  $O(\log(n))$ – Insert a into the balanced binary search tree

- **apply(a: A): Boolean**
  
  $O(\log(n))$ – Find a in the binary search tree, return true if found

- **remove(a: A): Unit**
  
  $O(\log(n))$ – Remove a from the binary search tree
TreeMap[K: Ordering, V]

- **put(k: K, v: V): Unit**
  \(O(\log(n))\) – Insert the pair \((k,v)\) into the balanced binary search tree according to the ordering on \(k\).

- **apply(k: K): V**
  \(O(\log(n))\) – Find \(k\) in the binary search tree, return the matching \(v\).

- **remove(k: K): Unit**
  \(O(\log(n))\) – Remove \(k\) from the binary search tree.

- **range(from: K, until: K): TreeMap[K, V]**
  \(O(\log(n)+|\text{range}|)\) – Return a sub-map containing only keys in the range \([\text{from}, \text{until})\).
Hash Tables
Hash Table with Chaining

- Create an array of size N
- Pick an O(1) function $h(k)$ to assign each record to $[0, N)$
  - A record with key $k$ can only be stored in bucket $h(k)$
  - Use linked lists if the bin is occupied
# Hash Table with Chaining

<table>
<thead>
<tr>
<th>Athos</th>
<th>B</th>
<th>C</th>
<th>D'Artagnan</th>
<th>...</th>
<th>Porthos</th>
<th>...</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>...</td>
<td>Ø</td>
<td>...</td>
<td>Ø</td>
<td>Ø</td>
</tr>
</tbody>
</table>

- Aramis
Picking a Lookup Function

- Desirable Features for $h(x)$
  - Fast
    - needs to be $O(1)$
  - “Unique”
    - As few duplicate bins as possible
Hash Functions

- **Examples**
  - SHA256 ← used by GIT
  - MD5, BCRYPT ← used by unix login, apt
  - MurmurHash3 ← used by Scala
- hash(x) is pseudorandom
  1) hash(x) ~ uniform random value in [0, INT_MAX)
  2) hash(x) always returns the same value
  3) hash(x) uncorrelated with hash(y) for x ≠ y
Lookup Table

- We want fewer than INT_MAX buckets
- Store a record with key k in bucket h(k) % N
Modulus

$h(k) =$

$$
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
7 & 8 & 9 & 10 & 11 & 12 & 13 \\
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
14 & 15 & 16 & 17 & 18 & 19 & 20 \\
\end{array}
$$
Iterating over a hash table

• Runtime
  – Visit every hash bucket
    • $O(N)$
  – Visit every element in every bucket
    • $O(n)$
    $= O(N + n)$
Hash Functions + Buckets

Everything is: \( O\left(\frac{n}{N}\right) \)

Let’s call \( \alpha = \frac{n}{N} \) the load factor.

**Idea:** Make \( \alpha \) a constant

Fix an \( \alpha_{max} \) and start requiring that \( \alpha \leq \alpha_{max} \)

What happens when the user inserts \( n = N \times \alpha_{max} + 1 \) records?
Rehashing

- Resize the array from $N_{\text{old}}$ to $N_{\text{new}}$.
  - Element $x$ moves from $\text{hash}(x) \mod N_{\text{old}}$ to $\text{hash}(x) \mod N_{\text{new}}$
- Runtime?
  - Allocate new array: $O(1)$
  - Visit every hash bucket: $O(N_{\text{old}})$
  - Hash and copy each element to the new array: $O(n)$
  - Free the old array: $O(1)$
  - $O(1) + O(N_{\text{old}}) + O(n) + O(1) = O(N_{\text{old}} + n)$
Rehashing

- Whenever $\alpha > \alpha_{\text{max}}$, rehash to double size
  - Contrast with ArrayBuffer
- Starting with $N$ buckets, after $n$ insertions...
  - Rehash at $n_1 = \alpha_{\text{max}} \times N$: From $N$ to $2N$ Buckets
  - Rehash at $n_2 = \alpha_{\text{max}} \times 2N$: From $2N$ to $4N$ Buckets
  - Rehash at $n_3 = \alpha_{\text{max}} \times 4N$: From $4N$ to $8N$ Buckets
  - ...
  - Rehash at $n_j = \alpha_{\text{max}} \times 2^jN$: From $2^{j-1}N$ to $2^jN$ Buckets
Number of Rehashes

With $n$ insertions...

$$n = 2^j \alpha_{max}$$

$$2^j = \frac{n}{\alpha_{max}}$$

$$j = \log \left( \frac{n}{\alpha_{max}} \right)$$

$$j = \log(n) - \log(\alpha_{max})$$

$$j = O(\log(n))$$
Total Work

Rehashes required: \( O(\log(n)) \)

The i-th rehashing: \( O(2^i N) \)

Total work after n insertions...

\[
O(\log(n)) \sum_{i=0} O(2^i N) = O \left( \sum_{i=0} O(\log(n)) 2^i + \sum_{i=0} O(\log(n)) N \right) \\
= O \left( 2^{O(\log(n)+1)} - 1 + O(\log(n)N) \right) \\
= O(n + N \log(n))
\]

Work per insertion: (ammortized cost)

\[
O \left( \frac{n + N \log(n)}{n} \right) = O \left( \frac{n}{n} + \frac{N \log(n)}{n} \right) = O(1)
\]
HashSet[A]

- **add(a: A): Unit**
  - Compare all elements in bucket \( h(a) \mod N \) to \( a \). If a match is not present, insert \( a \) at the head.
  
- **apply(a: A): Boolean**
  - Compare all elements in bucket \( h(a) \mod N \) to \( a \). If a match is found, return true.

- **remove(a: A): Unit**
  - Compare all elements in bucket \( h(a) \mod N \) to \( a \). If a match is found, remove the matched element.

- **expected: O(1)**
  - **worst-case: O(N)**
HashMap[K, V]

- **put(k: K, v: V): Unit**
  - Compare the key of all elements in bucket \( h(k) \% N \) to \( k \). If a match is present, remove it. Insert \((k, v)\) at the head.

- **apply(k: K): V**
  - Compare the key of all elements in bucket \( h(k) \% N \) to \( k \). If a match is found, return the corresponding value.

- **remove(a: A): Unit**
  - Compare the key of all elements in bucket \( h(k) \% N \) to \( k \). If a match is found, remove the matching element.

- **NO range operation**

  - expected: \( O(1) \)
  - worst-case: \( O(N) \)
Variations

- **Hash Table with Chaining**
  - ... but re-use empty hash buckets instead of chaining
  - **Hash Table with Open Addressing**
  - **Cuckoo Hashing** (Double Hashing)
    - ... but avoid bursty rehashing costs
  - **Dynamic Hashing**
    - ... but avoid O(N) iteration cost
  - **Linked Hash Table**
Open Addressing

- **insert(X)**
  - While bucket hash(X)+i %N is occupied, i = i + 1
  - Insert at bucket hash(X)+i %N

- **apply(X)**
  - While bucket hash(X)+i %N is occupied
    - If the element at bucket hash(X)+i %N is X, return it
    - Otherwise i = i + 1
  - Element not found
Open Addressing

- **Linear Probing**: Offset to $\text{hash}(X) + c_i$ for some constant $c$
- **Quadratic Probing**: Offset to $\text{hash}(X) + c_i^2$ for some constant $c$
- Follow Probing Strategy to find the next bucket

- Runtime Costs
  - Chaining: Dominated by following chain
  - Open Addressing: Dominated by probing
- With a low enough $\alpha_{\text{max}}$, operations still $O(1)$
Cuckoo Hashing

• Use two hash functions: hash₁, hash₂
  - Each record is stored at one of the two
• insert(x)
  - If both buckets are available: pick at random
  - If one bucket is available: insert record there
  - If neither bucket is available, pick one at random
    • “Displace” the record there, move it to the other bucket
    • Repeat displacement until an empty bucket is found

apply(x) and remove(x) is guaranteed O(1)
Linked Hash Table

- Iteration over Hash Table is $O(N + n)$
  - Can be much slower than $O(n)$
- **Idea:** Connect entries together in a Doubly Linked List
Linked Hash Table

- Athos
- B
- C
- D
- ... (head)
- Porthos
- ... (tail)
- Y
- Z

- Aramis
-∅ ∅
Linked Hash Table

- O(n) Iteration
- apply(x)
  - O(1) increase in cost
- insert(x)
  - O(1) increase in cost
- remove(x)
  - O(1) increase in cost
Lossy Sets / Bloom Filters
“Lossy Sets”

- Set[A]
  - **add(a: A)**: Insert a into the set
  - **apply(a: A)**: Return true if a is in the set

- What if we didn’t need apply to be perfect?
Lossy Sets

- LossySet[A]
  - add(a: A): Insert a into the set.
  - apply(a: A):
    - If a is in the set, always return true
    - If a is not in the set, usually return false
      - Is allowed to return true, even if a is not in the set
Bloom Filters

class BloomFilter[A](_size: Int, _k: Int) extends LossySet[A]
{
  val bits = new Array[Boolean](_size)

  def add(a: A): Unit = {
    for(i <- 0 until _k) { bits( ithHash(a, i) % _size ) = true }
  }

  def apply(a: A): Boolean = {
    for(i <- 0 until _k) {
      if( !bits( ithHash(a, i) % _size ) { return false; }
    }
    return true
  }
}
Bloom Filter Parameters

- \_size
  - Intuitively: More space, fewer collisions
- \_k
  - Intuitively: more hash functions means...
    - ...more chances for one of b’s bits to be unset.
    - ...more bits set = higher chance of collisions.

To preserve a constant false-positive rate:
Grow \_size as O(n)
Value of \_k is fixed for a given size.
Bloom Filters: Analysis

- $N/n = 5 \rightarrow \sim 10\%$ collision chance
- $N/n = 10 \rightarrow \sim 1\%$ collision chance

- 10 bits vs
  - 32 bits for one Int (3 to 1 savings)
  - 64 bits for a Double/Long (6 to 1 savings)
  - $\sim 8000$ bits for a full record (800 to 1 savings)