CSC263 Notes

Jenci Wei

Winter 2022

Contents

1 Analyzing Runtime Complexity

Data Structure

- **Abstract Data Type (ADT)**: set of objects together with operations
  - The "what" aspect
  - E.g. stack with operations `push(x)`, `pop()`, `isEmpty()`
- **Data Structure**: implementation of an ADT
  - The "how" aspect
  - E.g. stack can be implemented with linked list or array

Runtime Complexity Abstractions

- **Complexity**: amount of resources required by algorithm as a function of input size
- **Resource**: running time, or memory/space
- Input size is problem dependent
  - E.g. length for a list, number of bits for a number

Analyzing Runtime Complexity

- Asymptotic Notation:
  - Big O
  - Big Omega
  - Big Theta
- Runtime Cases:
  - Best case
  - Average case
  - Worst case

Asymptotic Notation

- $\mathcal{O}(g(n)) = \{f(n) : \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}^+ \text{ such that } \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$
  - The function $f(n)$ grows slower or at the same rate as $g(n)$
  - For worst case, the algorithm executes *no more than* $cg(n)$ steps on *any* input of size n
- $\Omega(g(n)) = \{f(n) : \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}^+ \text{ such that } \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\}$
  - The function $f(n)$ grows faster or at the same rate as $g(n)$
  - For worst case, exhibit one (family of) input on which the algorithm executes *at least* $cg(n)$ steps
- $\Theta(g(n)) = \{f(n) : \exists c_1, c_2 \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}^+ \text{ such that } \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$
  - The function $f(n)$ grows at the same rate as $g(n)$

Worst-Case Analysis

- What is the *maximum* possible running time of an algorithm for an input size n?
- $T(n) = \max\{t(x) : x \text{ is an input of size } n\}$

- Give a pessimistic upper bound that could occur for any input of a fixed size n as $T(n) = \mathcal{O}(f)$, must be proved on all possible inputs
- Give a family of inputs (one for each input size) and a lower bound on the number of basic operations that occurs for this particular family of inputs as $T(n) = \Omega(f(n))$
- If the two expressions involve the same function, then $T(n) = \Theta(f(n))$

Average-Case Analysis

- For algorithm A, define S_n to be the sample space of all inputs of size n
- Let $t_n(x)$ be the number of steps executed by A on input x (in S_n)
- Let $T_{avg}(n)$ be the weighted average of the algorithm's running time for all inputs of size n
- $E[T] = \sum_t t \cdot P[T = t]$
  - $\sum_t$: sum over all families of inputs
  - t: time for that family
  - $P[T = t]$: probability of seeing that family

Summation Formulas

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$
$$\sum_{k=0}^{n} a^k = \frac{a^{n+1} - 1}{a - 1}$$
$$\sum_{k=1}^{n} ka^k = \frac{a^{n+1}[n(a-1) - 1] + a}{(1-a)^2}$$

Performing an Average-Case Analysis

1. Define the possible set of inputs and a **probability distribution** over this set
2. Define the "basic operations" being counted
3. Define the **random variable** T over this probability space to represent the running time of the algorithm
4. Compute the expected value of T, using the chosen probability distribution and formula for T
5. Since the average number of steps is counted as an exact expression, it can be written as a Theta expression

2 Priority Queue ADT and Heaps

ADT: Priority Queue

- Data: a collection of items which each have a priority
- Operations:
  - `Insert(PQ, x, priority)`
  - `FindMax(PQ)`
  - `ExtractMax(PQ)`

Possible Data Structures for Priority Queue

Data Structure	Insert	FindMax	ExtractMax
Unsorted Linked List	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Unsorted Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Sorted Array	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Ordered Linked List	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Binary Search Tree	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

Heaps

- Based on a nearly complete binary tree
  - Every row is completely filled except possibly the lowest row
  - The lowest row is filled from the *left*
- Heap property determines relationship between values of parents and children
  - Max heap: the value at every node is *greater than or equal to* the value of its *immediate* children
  - Min heap: the value at every node is *less than or equal to* the value of its *immediate* children
- Stored in an array
  - Use 1-based indexing so root is at element 1
  - For node at position i:
    * Its left child is at $2i$
    * Its right child is at $2i + 1$
    * Its parent is at $\lfloor i/2 \rfloor$

Implementing the PQ Operations: Insert

- Increment heapsize and add element at next position
- Result might violate heap property so *bubble up*, i.e. if element's value is greater than its parent, swap the nodes
- Running time: $\Theta(h) = \Theta(\log n)$

Implementing the PQ Operations: FindMax

- The root element has the maximum value
- Running time: $\Theta(1)$

Implementing the PQ Operations: ExtractMax

- Remove and return root element

- Strategy: restore shape first then fix heap property
  - Replace the root node with the last node, then *bubble down*, i.e. if element's value is less than its greatest child, swap the nodes
- Bubble down also called **max-heapify**
- Running time: $\Theta(h) = \Theta(\log n)$

Heap Sort

- Assuming we start with a valid heap, we want a sorted list
- The root of the heap stores the maximum element
  1. Remove the root and put it in an array at the end
  2. Decrement heap size
  3. Restore heap property
- We can do this *in place* since replacement item for root was in the position of where we want to put the root
- Complexity: $\mathcal{O}(n \log n)$ because we are performing ExtractMax n times

Building a Heap

- Approach 1
  - Start with empty heap, repeatedly call insert
  - Last insert takes $\log n$ time
  - Each insert takes less than $\log n$ time
  - Build time: $\mathcal{O}(n \log n)$
  - Input family: add elements in increasing order, then at least half of the elements are leaves, where each leaf takes at least $\log n - 1$ time
  - We get $\Theta(n \log n)$ time
- Approach 2
  - Start from $i = \lfloor n/2 \rfloor$ (height 1), calling MaxHeapify, and work back to $i = 1$
  - Running time of MaxHeapify is proportional to height of current subtree

Height	Leaves	Swaps Required
0	$n/2$	0
1	$n/4$	1
2	$n/8$	2
$\vdots$	$\vdots$	$\vdots$
$\log n - 1$	2	$\log n - 1$
$\log n$	1	$\log n$

  - A heap contains at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes at height h
  - Total time:

$$\sum_{h=1}^{\lfloor \log n \rfloor} h \left\lceil \frac{n}{2^{h+1}} \right\rceil = \mathcal{O}(n)$$

3 Dictionaries and Binary Search Trees

Dictionary ADT Operations:

- `Insert(S, x)`
  - `x` is both the key and the value
  - Assume that we know where to insert
- `Search(S, k)`
- `Delete(S, x)`
  - Assume that we can directly access the item in the data structure

Possible Data Structures for Dictionary

Data Structure	Search	Insert	Delete
Unsorted Array	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Sorted Array	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
Unsorted Linked List	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Sorted Linked List	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Direct-Access Table	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Hash Table	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Binary Search Tree	$\Theta(h)$	$\Theta(1)$	$\Theta(h)$
Balanced Search Tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

Direct Access Table

- Needs 1 location per possible key
- Suppose keys are 32-bit integers, then we need $\Omega(2^{32})$ space

Hash Table

- Map all possible keys onto a smaller set of actual keys, from which we do direct access
- If two items have keys that map the same new key, then we have a linked list that corresponds to that location
- Worst case for search is $\Theta(n)$

Binary Search Tree

- Worst case height is n
- Best case height is $\log n$

Balanced Binary Search Tree

- If we could keep the BST balanced, then we could have $\Theta(\log n)$ search and $\Theta(\log n)$ delete
- Types of balanced BST
  - Red-black tree
  - AVL tree
  - 2-3-4 tree
  - B tree

Dictionary Using Binary Search Trees

- For every node of the BST:

$$\text{elements in left subtree} \;\leq\; \text{element at node} \;\leq\; \text{elements in right subtree}$$

- Dictionary S stores only S.root (and usually S.size as well)
- TreeNode has members:
  - .item: element stored in node
  - .left and .right: children
- Recursive code for operations

If Duplicate Keys were Allowed

- We want to insert n identical items into an empty tree
- Always go left/right is inefficient
- Alternative methods
  - Strictly alternate using a flag to keep track
  - Randomly pick a side
  - Keep a chain of values inside a node

4 AVL Trees

A binary tree of height h is **ideally height-balanced** if every node of depth $< h-1$ has two children

- A tree of n nodes would be guaranteed to have height $h = \lfloor \log_2 n \rfloor$, so searches would always take time in $\mathcal{O}(\log n)$
- Insertions and deletions may destroy this property

A binary tree is **height-balanced** if the heights of the left and right subtrees of *every* node differ by at most one

- AVL tree (Adelson-Velski Landis)

Time complexity of AVL tree

- The worst case height of an AVL tree with n nodes is $1.44 \log_2(n+2)$
- Search operation is $\mathcal{O}(\log n)$ in the worst case
- Insertions and deletions can be done in $\mathcal{O}(\log n)$ time, while preserving AVL property
- Empirically works well on the average case

Let h_R and h_L be the heights of the right and left subtrees of a node m in a binary tree, respectively. The **balance factor** of m is defined as $BF(m) = h_R - h_L$

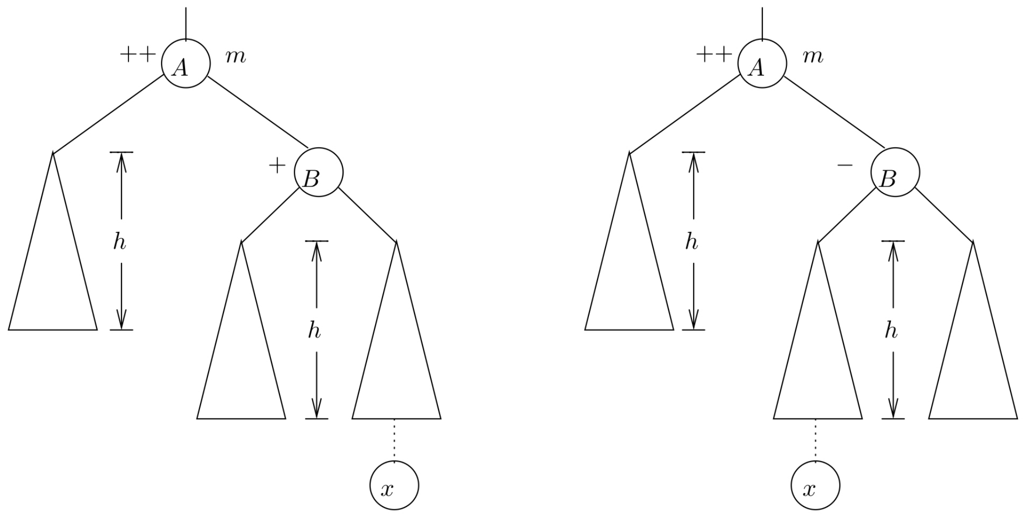
- For an AVL tree, the balance factor of any node is -1, 0, or 1
  - If $BF(m) = 1$, then m is **right heavy**
  - If $BF(m) = -1$, then m is **left heavy**
  - If $BF(m) = 0$, then m is **balanced**
- In AVL trees, we store $BF(m)$ in each node m

Algorithm for Search: binary search

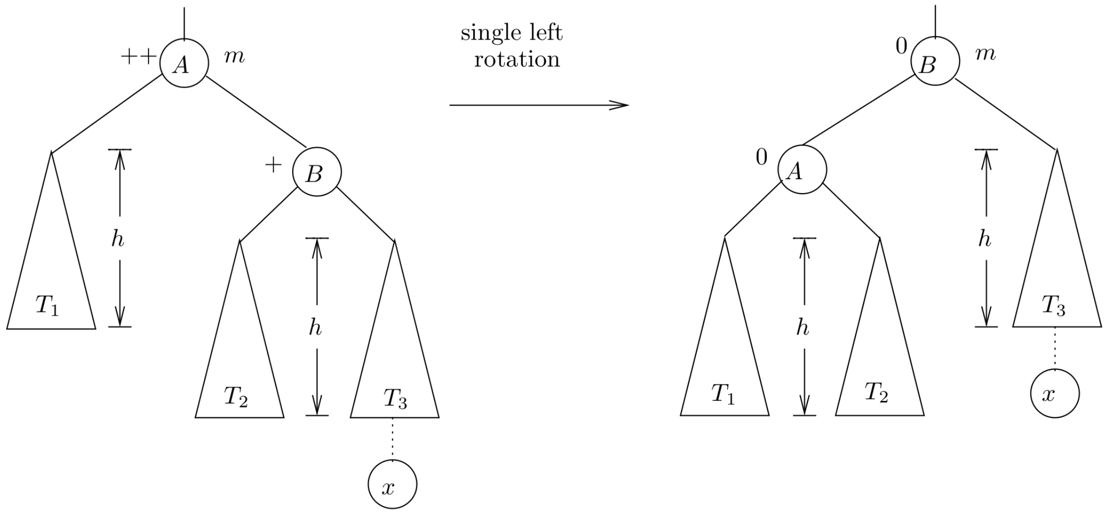
Algorithm for Insert:

- To insert a key x into an AVL tree T, first insert x in T as in ordinary BSTs
- AVL property may be destroyed:
  - The addition of a new leaf may have destroyed the height-balance of some nodes
  - The balance factors of some nodes must be updated to take into account the new leaf
- Rebalancing an AVL tree after insertion:
  - Two potential scenarios:
    1. The new leaf increased the height of the right subtree of a node that was already right heavy
    2. The new leaf increased the height of the left subtree of a node that was already left heavy
  - The insertion of the new leaf can only affect the balance factors of its ancestors

– Two possible scenarios:

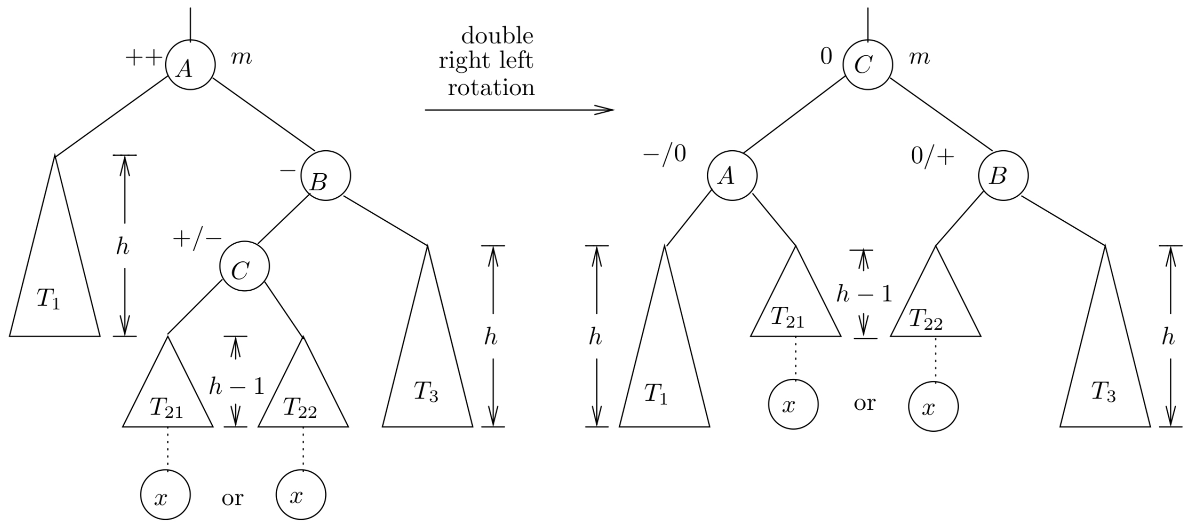


– The first scenario can be rebalanced by performing a *single left rotation* on m



  * Rebalances the subtree rooted at node m
  * Maintains BST property
  * Can be done in constant time (i.e. switching a few pointers around)
  * Keeps the height of m equal to its height *before* the insertion of the new node

– The second scenario can be rebalanced by performing a *double right left rotation*, i.e. rotate B to the right then rotate C to the left



  - ∗ Rebalances the suubtree rooted at m
  - ∗ Maintains BST property
  - ∗ Can be done in constant time (i.e. switching a few pointers around)
  - ∗ Keeps the height of m equal to that node's height *before* the insertion of the new node

- – *Single right rotation* and *double left right rotation* work similarly

• Updating the balance factors after insertion

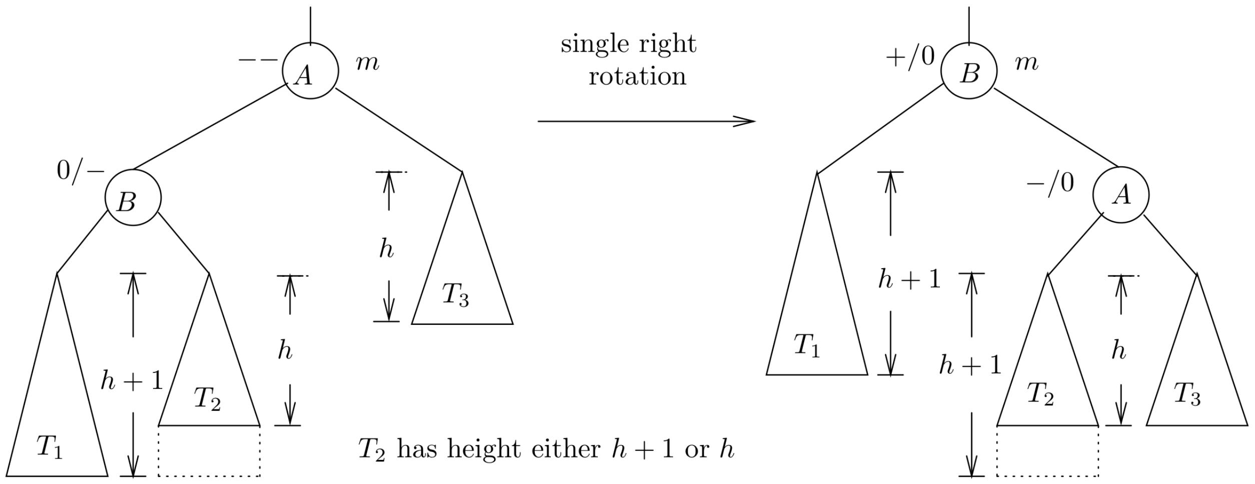
- – Let n be the new node just inserted into the tree and let p be n's parent. Further, let m be the closest ancestor of p that was *not* balanced (i.e. its balance factor is ± 1) before the insertion of n. If no such ancestor of p exists, let m be the root of the tree
- – Only the balance factors of the nodes between p and m (included) need to be changed as a result of the insertion of n

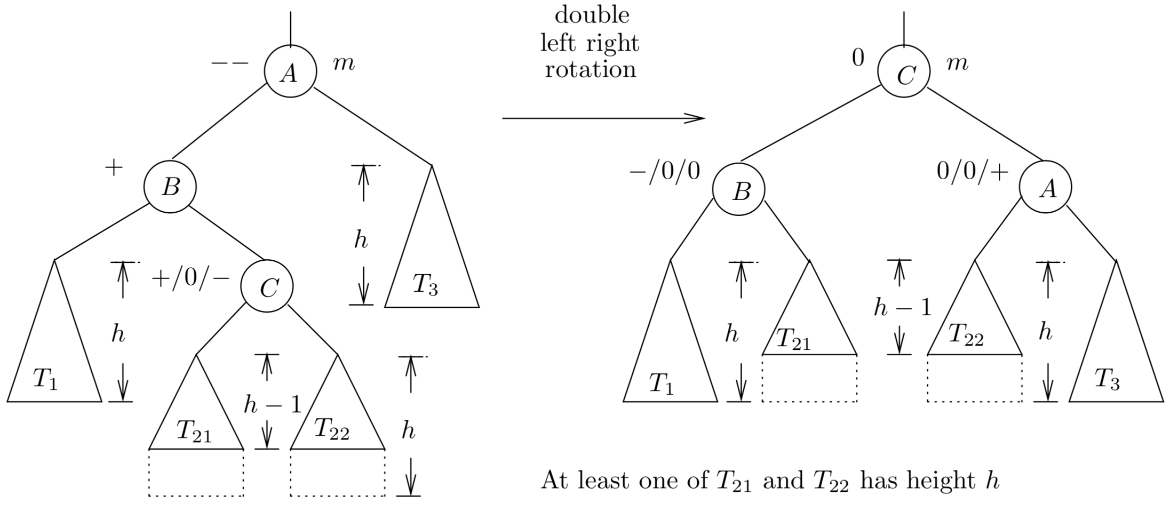
• Algorithm for `insert(x, T)`

1. Trace a path from the root down, as in BSTs, and insert x into a new leaf at the end of that path (in proper position)
2. Set the BF of the new leaf to 0. Retrace the path from the leaf up towards the root and process each node i encountered as follows:
   (a) If the new node was inserted in i's right subtree, then increase $BF(i)$ by 1 (since i's right subtree got taller); otherwise, decrease $BF(i)$ by 1 (since i's left subtree got taller)
   (b) If $BF(i) = 0$ (so the subtree rooted at i became balanced as a result of the insertion, and its height did not change) then stop
   (c) If $BF(i) = 2$ and $BF(rchild(i)) = 1$ then do a single left rotation on i, adjust the balance factors of the rotated nodes, and stop
   (d) If $BF(i) = 2$ and $BF(rchild(i)) = -1$ then do a double right left rotation on i, adjust the balance factors of the rotated nodes, and stop
   (e) If $BF(i) = -2$ and $BF(lchild(i)) = -1$ then do a single right rotation on i, adjust the balance factors of the rotated nodes, and stop
   (f) If $BF(i) = -2$ and $BF(rchild(i)) = 1$ then do a double left right rotation on i, adjust the balance factors of the rotated nodes, and stop
   (g) If $i = root$ then stop

Algorithm for Delete:

- To delete a key x from an AVL tree T, first locate the node n where x is stored
  1. No such node exists. There is nothing to delete and we are done
  2. n is a leaf. We remove it and rebalance the tree
  3. n is a node with only one child. Let n' be n's only child. Note that n' must be a leaf. In this case we copy the key stored at n' into n and remove n' as in the previous case
  4. n has two children. We find the smallest key in n's right subtree (i.e. the smallest key in T larger than the key stored in n). We go to n's right child and follow the longest chain of left child pointers until we get to a node n' that has no left child. We copy the key stored in n' into n and remove n' from the tree, as in the one of the previous cases
- Rebalancing an AVL tree after deleting a leaf
  - The deletion of a leaf n will cause the tree to become unbalanced in one of two cases:
    1. It reduces the height of the right subtree of a left heavy node
    2. It reduces the height of the left subtree of a right heavy node
  - The first case can be rebalanced by a single right rotation or a double left right rotation





    * The tree rooted at m is rebalanced

    * BST property is maintained
    * Can be done in constant time by manipulating a few pointers
    * May decrease the height of the subtree rooted at m

- Updating the balance factors after deleting a leaf
  - Let n be the deleted leaf and let p be its parent
  - Trace the path from p back to the root and process each node i we encounter on the way as follows:
    * If i was balanced before the deletion (i.e. $BF(i) = 0$), then the left and right subtrees of i had the same height. The removal of n shortened one of them, but the height of the subtree rooted at i after the deletion remains the same as before it. And so the deletion of n does not affect the balance factors of i's proper ancestors. So, in this case, we increase $BF(i)$ by 1 if n was in i's left subtree, or decrease $BF(i)$ by 1 if n was in i's right subtree. After this, we can stop the process of updating balance factors
    * If i was right or left heavy before the deletion (i.e. $BF(i) = \pm 1$), we again update $BF(i)$ as above. If this balances node i, the deletion of n shortened one of the two subtrees of i, so we go up the path to consider the next node. Otherwise, the increase or decrease of $BF(i)$ by one causes the subtree rooted at i to become unbalanced (i.e. $BF(i) = \pm 2$). In this case we need to rebalance the subtree by the appropriate rotation. If the rotation causes the height of i to decrease, the process of updating balance factors (and possibly rotating) must continue with i's parent. Otherwise, the rotation leaves the height of i the same as it was before the deletion, and therefore the process stops at i
    * If the process propagated all the way to the root, we can stop

Worst Case Time Complexity for Search, Insert, Delete

- The height of an AVL tree with n nodes is at most $1.44 \log_2(n+2)$
- Thus all the above algorithms take $\mathcal{O}(\log n)$ time in the worst case

5 Hashing

Definitions

- **Universe of keys** U: set of all possible keys
- **Hash table** T: array with m positions
  - Each location is called a **slot** or **bucket**
- **Hash function** $h : U \to \{0, 1, \ldots, m-1\}$ maps key to array position
  - To access key k (or its data), examine $T[h(k)]$
- **Collision**: keys $x \neq y$ with $h(x) = h(y)$
  - Unavoidable since $|U| > m$

Chaining (Close Addressing)

- Each slot of T stores a linked list of items that hash to this bucket
- Worst case: all keys hash to same bucket

Average-Case Runtime With Chaining

- **Simple uniform hashing assumption**: assume that any key is equally likely to hash to any bucket
- Expected number of keys in each bucket is the same, i.e. n items per m slots
- **Load factor**: $\alpha = n/m$
- Distribution over input: we are equally likely to search for any key in U

Runtime of Search with Chaining

- Running time = time to compute hash function + number of keys we need to compare
- Assume k is not in table
  - Compute $h(k)$ then traverse the entire list in $T[h(k)]$
  - $E[\text{length of list in } h(k)] = n/m = \alpha$
  - $E[T] = 1 + \alpha = \Theta(1 + \alpha)$
- Assume k is in the table
  - Values we inserted into table: $X_1, X_2, \ldots, X_n$
  - Search for X_i
  - Probability that we select X_i: $P(X_i) = 1/n$
  - Time = 1 + elements we examine when we search for X_i
  - Items inserted into T after X_i: $n - i$
  - Number of these $n - i$ items that we expect in each bucket: $\frac{n-1}{m}$
  - $E[T] = 1 + \sum_{i=1}^{n} P(X_i) \cdot T(X_i) = 1 + \frac{1}{n} \sum_{i=1}^{n} \left(\frac{n-i}{m} + 1\right)$
    - Leading 1: time for applying the hash function
    - $\frac{n-i}{m}$: comparisons with elements in the chain before the element
    - Trailing 1: comparison with the element

  - $E[T]$ evaluates to $\Theta(1+\alpha)$
- And so search takes $\Theta(1+\alpha)$ in average case

Open Addressing

- Keep all records *in* the table
- When the first bucket is already taken, go to another bucket
- Instead of $h(k)$, we have $h(k,i)$ that maps to $\{0,1,\dots,m-1\}$
- **Probe sequence**, i.e. the sequence of buckets that we try for key k, is $(h(k,0),h(k,1),\dots,h(k,m-1))$
  - We try all m buckets
  - We require the probe sequence to be a permutation of $(0,1,\dots,m-1)$

Linear Probing

- Linear probing: $h(k,i)=(h'(k)+i) \mod m$
- $h'(k)=k \mod m$, the home bucket
- There may be clusters
  - Once we hash to anywhere along the probe sequence in the cluster, we extend the cluster
  - Using a larger step than 1 would *not* help, because we are merely spacing the clusters
  - A random step size would *not* help, because we would not be able to search

Quadratic Probing

- Quadratic probing: $h(k,i)=(h'(k)+c_1 i+c_2 i^2) \mod m$
- $h'(k)=k \mod m$, the home bucket
- Need to pick c_1, c_2, and m carefully, since we want the probe sequence to be a permutation of $(0,1,\dots,m-1)$
- 2 keys with same home bucket have same probe sequence

Double Hashing

- Double hashing: $h(k,i)=(h_1(k)+h_2(k)i) \mod m$
- $h_1(k)=k \mod m$, the home bucket
- Step size is determined by applying another hash function, h_2
- Step size for 2 keys with the same home bucket can be different
- Want the probe sequence to be a permutation of $(0,1,\dots,m-1)$
- Need $h_2(k)$ and m to be relatively prime (coprime)
  - E.g. select m to be a power of 2, and $h_2(k)$ to be odd

Deleting in Open Addressing

- When we delete, we mark the spot with a “tombstone”
- When we are searching and encounter a tombstone, we do not stop and conclude that the element is not in the hash table; instead, we continue searching the next bucket

- When we are inserting and encounter a tombstone, we replace the tombstone with that element

Hash Functions

- A good hash function:
  1. "Spreads out" value (avoid clusters)
  2. Efficient to compute
  3. Depends on *every* part of key, even for complex objects
- From string to natural number
  - Add the ascii code for characters
    - "key" is hashed to $107 + 101 + 121$
    - But "yek" is also hashed to $107 + 101 + 121$
  - Use ascii code as digit in base 128
    - "key" is hashed to $107 * 128^2 + 101 * 128 + 121$
    - "yek" is hashed to $121 * 128^2 + 101 * 128 + 107$
- From natural number k to m
  - Division method: $h(k) = k \mod m$
    - Avoid powers of 2 for m
    - Pick a prime close to the m that we want
  - Multiplication method
    1. Choose an A satisfying $0 < A < 1$
    2. Multiply k by A
    3. Keep the fractional part
    4. Multiply by m
    5. Take the floor
    6. We get $0 \leq \lfloor m\,(kA - \lfloor kA \rfloor)\rfloor < m$

Hashing vs. Balanced Tree

- Hashing is $\Theta(1)$ average case, better than balanced tree
- Use balanced tree for the below circumstances:
  - We can't tolerate the worst case performace
  - We need to find *all* keys or *all* values, or any *range* of values

6 Augmenting Data Structures

Augmented data structure: existing data structure modified to store additional information and/or perform additional operations

1. Choose data structure to augment
2. Determine the additional information
3. Check that additional information can be maintained during each original operation
4. Implement new operations

Ordered Sets

- Rank: position within an ordered list, from lowest to highest
- Operations:
  - `Rank(k)` returns rank of key k
  - `Select(r)` returns the key with rank r
  - `Insert`, `Delete`, `Search`

Approach: AVL Tree

- Queries: carry out in-order traversal of tree, keeping track of the number of nodes visited until we reach the desired key/rank
- New queries take $\Theta(n)$ time in the worst case
- The other operations take the same time

Approach: Augmented AVL Tree, where each node has an additional field that stores its rank in the tree

- Time for new queries: $\Theta(\log(n))$
- Search takes the same time
- Insert/Delete takes $\Theta(n)$ time because the entire tree needs to be updated

Approach: Augmented AVL Tree, subtree size is stored

- Each node n has an additional field `n.size` that stores the number of keys in the subtree rooted at n, including n itself
- The rank of the root of the tree is `root.left.size + 1`
- The *local* (*relative*) rank of a node m in the tree rooted at m is `m.left.size + 1`
- Implementation of Select:

```
Select(T, r):
    if T.left != None:
        local_rank = T.left.size + 1
    else:
        local_rank = 1
    if r == local_rank:
        return T.key
    elif r < local_rank:
        return Select(T.left, r)
    else:
        return Select(T.right, r - local_rank)
```

- Rank
  - Perform Search on k and keep track of the current rank. Each time we go down a level by making a right turn, add the size of the subtrees to the left (that we have skipped), i.e. the local rank of the parent we just left
  - Local rank of a node m in the tree rooted at m: `m.left.size + 1`
  - When we recurse on the left subtree, we add 0
  - When we recurse on the right subtree, we add the parent's local rank
  - When we find x, its true rank is the current rank + the local rank
  - Implementation:

```
Rank(T, k):
    return TreeRank(T.root, k, 0)

TreeRank(root, k, current_rank):
    if root is None: # k not in tree
        return -1
    if root.left is None:
        local_rank = 1
    else:
        local_rank = root.left.size + 1

    if k < root.item.key:
        return TreeRank(root.left, k, current_rank)
    elif k > root.item.key:
        return TreeRank(root.right, k, current_rank + local_rank)
    else:
        return current_rank + local_rank
```

- Time for new queries: $\Theta(\log n)$
- When inserting, add 1 to subtree size as we go down the path
- When deleting, subtract 1 to subtree size as we go down the path
- And so these operations take the same time

Augmenting Strategy

- Use known data structure with extra information
- Extra information must be maintained on original operations

7 Quicksort

Algorithm

```
 1  def quickSort(array):
 2      if len(array) < 2:
 3          return array[:]
 4      else:
 5          pivot = array[0]
 6          smaller, bigger = partition(array[1:], pivot)
 7          smaller = quickSort(smaller)
 8          bigger = quickSort(bigger)
 9          return smaller + [pivot] + bigger
10
11  def partition(array, pivot):
12      smaller = []
13      bigger = []
14      for item in array:
15          if item <= pivot:
16              smaller.append(item)
17          else:
18              bigger.append(item)
19      return smaller, bigger
```

Worst Case – Upper Bound (count item comparisons)

- Each element of S is the pivot at most once
- At most all other elements are compared to the pivot
- So every pair of elements in S are compared at most once
- $|S| = n \implies \binom{n}{2}$ pairs
- $T(n) \le \binom{n}{2} \implies T(n) \in \mathcal{O}(n^2)$

Worst Case – Lower Bound (count item comparisons)

- Let $C(n)$ denote the number of comparisons on input $[n, n-1, \dots, 1]$
- In line 6, pivot n is compared in `partition` to all other $n-1$ values
  - This yields $\texttt{smaller} = [\texttt{n}-\texttt{1}, \texttt{n}-\texttt{2}, \dots, \texttt{1}]$ and $\texttt{bigger} = []$
- All other comparisons happen in the recursive call `quicksort(smaller)` which takes $C(n-1)$ comparisons
- Hence $C(n)$ satisfies the recurrence relation

$$\begin{cases} C(n) = n - 1 + C(n-1), & n > 1 \\ C(1) = 0 \end{cases}$$

- $C(n) = (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2} \in \Omega(n^2)$
- Therefore $T(n) \in \Theta(n^2)$

Average Case

- Inputs: all permutations of $[1, , 2 \cdots, n]$, assuming uniform distribution

- T is the RV counting the number of comparisons
- We want $E[T]$
- Define an indicator RV: for $i, j \in \{1, 2, \cdots, n\}$ satisfying $i < j$,

$$X_{ij} = \begin{cases} 1, & \text{if } i \text{ and } j \text{ are compared} \\ 0, & \text{elsewise} \end{cases}$$

- $E[T] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}]$, summing over the probability that i is compared to j
- Before picking i or j as a pivot, if we pick in the $[i+1, j-1]$ range, then i and j will never be compared, because i will land in `smaller` and j will land in `bigger`
- There are $j - i + 1$ values in the $[i, j]$ range
- The probability that i and j will be compared is $\frac{2}{j-i+1}$

$$\begin{aligned} E[T] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\ &= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \end{aligned}$$

- Observe how many values k can take based on the value of i

$$\begin{aligned} \text{When } i = 1 \quad & k \to 1, 2, 3, \ldots, n-3, n-2, n-1 \\ i = 2 \quad & k \to 1, 2, 3, \ldots, n-3, n-2 \\ i = 3 \quad & k \to 1, 2, 3, \ldots, n-3 \\ \vdots \quad & \\ i = n-2 \quad & k \to 1, 2 \\ i = n-1 \quad & k \to 1 \end{aligned}$$

- For k, we have $n - 1$ of 1s, $n - 2$ of 2s, etc.
- Generalizing, we have $n - k$ copies of $\frac{1}{k+1}$

$$\begin{aligned} E[T] &= 2 \sum_{k=1}^{n-1} \frac{1}{k+1}(n-k) \\ &= 2 \sum_{k=1}^{n-1} \frac{n-k-1+1}{k+1} \\ &= 2 \sum_{k=1}^{n-1} \frac{n+1}{k+1} - 2(n-1) \\ &= 2(n+1) \sum_{k=1}^{n-1} \frac{1}{k+1} - 2(n-1) \end{aligned}$$

- Since the harmonic series is $\Theta(\log n)$, we have $E[T] \in \Theta(n \log n)$

Quicksort Summary

- Performs well on *random* input
- Performs poorly on *nearly sorted* input

Randomized Quicksort

- Average-case analysis depended on each permutation equally likely
- Instead of relying on distribution of inputs, randomize algorithm by picking random element as input
- Random behaviour of algorithm on any fixed input is equivalent to fixed behaviour of algorith on uniformly random input
- *Expected* worst-case time of randomized algorithm on every single input is $\Theta(n \log n)$
- If we ran quicksort on any particular input multiple times and average the time, it would perform $\Theta(n \log n)$
- Randomized algorithms are good when there are lots of good choices, but hard to find a choice that is guaranteed to be good

8 Amortized Analysis

We often perform *sequence* of operations on data structures, and we are interested in the time complexity of the *entire sequence*

Worst-case sequence complexity (WCSC): maximum time over all sequences of m operations

- WCSC $\leq m\times$ (worst-case time complexity of *any one* operation in a sequence of m operations)

Amortized Sequence Complexity

- **Amortized sequence complexity**: $\frac{\text{worst-case sequence complexity over all possible sequences of } m \text{ operations}}{m}$
- Represents the "average" worst-case complexity of each operation
- Different from average-case time (no probability)

Approaches to Calculating Amortized Complexity

- **Aggregate**
  - Add together the costs from the m operations
  - Works best when there is only *one* operation
  - Useful to find upper bound on *all possible* sequences of m operations
- **Accounting**
  - Charge each operation a "fee" and pay the real cost
  - Bank the overcharge to a *specific element* in the data structure
- **Potential**
  - Charge every operation but now store the overcharged amount as "potential energy" in the *data structure overall*

Example: Binary Counter – Aggregate Approach

- Algorithm

```
def increment(A):
    i = 0;
    while i < len(A) and A[i] = 1:
        A[i] = 0;
        i++;
    if i < len(A):
        A[i] = 1
```

- Table of costs

# of Calls	Binary	Decimal Value	Cost
Initial	00000	0	n/a
1	00001	1	1
2	00010	2	2
3	00011	3	1
4	00100	4	3
5	00101	5	1
6	00110	6	2
⋮	⋮	⋮	⋮
29	11101	29	1
30	11110	30	2
31	11111	31	1
32	00000	0	5

- Change based on array element

$k-1$		4	3	2	1	0
	$\cdots$	every 16th	every 8th	every 4nd	every 2nd	increment every time

- Table of changes

Bit #	Changes	Total Number of Changes in n Ops
0	Every time	n
1	Every 2 ops	$\lfloor n/2 \rfloor$
2	Every 4 ops	$\lfloor n/4 \rfloor$
i	Every 2^i ops	$\lfloor n/2^i \rfloor$
$k-1$	Every 2^{k-1} ops	$\lfloor n/2^{k-1} \rfloor$

- Sum over all the bits:

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

- Therefore the total running time for n operations is $\mathcal{O}(n)$
- And so we have an amortized cost per operation of $\frac{\mathcal{O}(n)}{n} = \mathcal{O}(1)$

Example: Multipop Stack – Aggregate Approach

- A stack with `push` and `pop` operations, each take $\mathcal{O}(1)$ time
- There is also a `multipop` operation with the below algorithm

```
def multipop(S, k):
    while not stack_empty(S) and k != 0:
        pop(S)
        k--;
```

- `multipop` takes $\min\{k, |S|\}$ steps
- Worst case operation of `multipop` is $\mathcal{O}(n)$
- If we have a sequence of n operations, all of which are `multipop`, we would have a worst-case of $\mathcal{O}(n^2)$
- However the above scenario is not possible, since if we popped all n elements in the first call, there would be no operations left
- In n operations, there is at most n calls to `push`, which has a cost of n

- Since $|S|$ is at most n, in total there are at most n `pops`
- We spend at most n on `pops` (either solo or inside `multipops`)
- Total cost for n operations is $\mathcal{O}(n)$
- Amortized cost is $\frac{\mathcal{O}(n)}{n} = \mathcal{O}(1)$

Accounting Approach

- Charge each operation an amortized cost
  - Amortized costs can be more or less than the actual cost
  - Can be different for different types of operations
- When amortized cost $\hat{C}_i$ is greater than actual cost C_i, credit is assigned to the associated element in the data structure
- When $\hat{C}_i < C_i$, the difference must be paid for by existing credit
- Can *never* be in debt

$$\sum_{i=1}^{k} \hat{C}_i \geq \sum_{i=1}^{k} C_i \quad \forall k$$

Example: Multipop Stack – Accounting Approach

- Actual costs

`push(S)`	`pop(S)`	`multipop(S, k)`
\$1	\$1	\$ $\min\{k, \lvert S\rvert\}$

- Charge

`push(S)`	`pop(S)`	`multipop(S, k)`
\$2	\$0	\$0

- Credit invariant: every element on the stack has \$1 on it
- Charge for `pop` and `multipop` is always paid for by the \$1 on each of the items that will be popped
- Amortized cost per operation: $\mathcal{O}(1)$

Example: Binary Counter – Accounting Approach

- Charge \$2 for each `increment` operation
- Credit invariant: at any step in the sequence, each bit in the counter that is equal to 1 will have \$1 credit

  Proof. (by induction)
  Initially the counter is 0 and no credit, therefore the credit invariant is trivially true.

  Assume the invariant is true and call `increment`. Cost of flipping 1s to 0s are paid for by the \$1 on each 1. Cost of flipping the only 0 that goes to 1 is paid for by \$1 on the \$2 charge, and the remaining \$1 is stored in that 1 bit.

  Since no other bits change, every 1 has \$1 on it at the end. This shows that the total charge for the sequence of n operations (i.e n calls to `increment`) is upper-bounded by the total cost of $2n$, so amortized cost per operation is $\leq \frac{2n}{n} \in \mathcal{O}(1)$.

  □

Example: Dynamic Array – Aggregate Method

- Algorithm for `insert`:

```
def insert(A, x):
    # Check whether current array is full
    if A.size == A.allocated:
        new_array = new array of length (A.allocated * 2)
        copy all elements of A.array to new_array
        A.array = new_array
        A.allocated *= 2
    # insert the new element into the first empty spot
    A.array[A.size] = x
    A.size++
```

- Time for n operations

$$T(n) = \sum_{k=0}^{n-1} t_k, \qquad \text{where } t_k = \begin{cases} 2k+1, & \text{if } k \text{ is a power of 2} \\ 1, & \text{elsewise} \end{cases}$$

- Let $t'_k = t_k - 1$ so that $t_k = t'_k + 1$ and $t'_k = \begin{cases} 2k, & \text{if } k \text{ is a power of 2} \\ 0, & \text{elsewise} \end{cases}$

$$\begin{aligned} T(n) &= \sum_{k=0}^{n-1} t'_k + n \\ &= \sum_{i=0}^{\lfloor \log(n-1) \rfloor} t'_{2^i} + n \\ &= \sum_{i=0}^{\lfloor \log(n-1) \rfloor} 2 \cdot 2^i + n \\ &= 2 \sum_{i=0}^{\lfloor \log(n-1) \rfloor} 2^i + n \\ &= 2\left(2^{\lfloor \log(n-1) \rfloor + 1} - 1\right) + n \qquad \text{By identity } \sum_{i=0}^{b} 2^i = 2^{b+1} - 1 \\ &\leq 2\left(2 \cdot 2^{\log(n-1)} - 1\right) + n \\ &= 2\left(2 \cdot (n-1) - 1\right) + n \\ &= 5n - 6 \end{aligned}$$

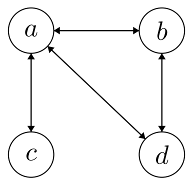
- Therefore $T(n) \in \mathcal{O}(n)$, and so the amortized cost for a single insertion is $\mathcal{O}(1)$

9 Graphs

Representing Graphs

- We care about the *topology* of the graphs, not the *geometry*

Example



Adjacency Lists

- Format:

$$\text{vertex} : \text{sublist of adjacent vertices}$$

- Example:

$$\begin{aligned} a &: b, d, c \\ b &: a, d \\ c &: a \\ d &: a, b \end{aligned}$$

- Usually implemented with linked lists

Adjacency Matrix

- Each entry is an indicator of whether an element is adjacent to another
- Example:

	a	b	c	d
a	0	1	1	1
b	1	0	0	1
c	1	0	0	0
d	1	1	0	0

  - Symmetric since graph is undirected
  - 0s along the diagonal since there are no self loops

Adjacency Lists vs. Adjacency Matrices – Space Complexity

- A-List: $\Theta(|V| + |E|)$
- A-Matrix: $\Theta(|V|^2)$

Operations

- Add/remove vertex
- Add/remove edge
- Edge query, i.e. given vertices u, v, is $(u, v) \in E$?
- Neighbourhood query

  - In-neighbourhood query: given vertex u, return the set of vertices v s.t. $(v, u) \in E$
  - Out-neighbourhood query: given vertex u, return the set of vertices v s.t. $(u, v) \in E$

Breadth First Search

- Starting from *source vertex* $s \in V$, BFS visits every vertex v *reachable* from s
- In the process, we find paths from s to each reachable $v \in V$
- Paths make a BFS tree rooted at s
- Works on directed and undirected graphs
- Keeping track of progress by colouring each vertex
  - **White**: not yet discovered, all vertices start white
  - **Grey**: discovered but not fully explored
  - **Black**: fully explored
- Keeps track of:
  - *Parent* of v in BFS tree
  - *Distance* from s to v
- *Grey* vertices are stored in a queue so they are handled in FIFO order
- Use adjacency list order
- Algorithm

```
def bfs(V, E, s):
    # Initialization of all vertices
    for v in V:
        v.colour = white
        v.dist = inf # Distance
        v.pi = NIL # Parent

    # Initialization of s
    s.colour = grey
    s.dist = 0
    initialize empty queue Q
    enqueue(Q, s) # loop invariant: Q contains exactly the grey vertices

    while Q not empty:
        u = dequeue(Q)
        for (u, v) in E:
            if v.colour == white: # found a new vertex
                v.colour = grey
                v.dist = u.dist + 1
                v.pi = u
                enqueue(Q, v)
        u.colour = black
```

BFS Complexity

- Each vertex is enqueued at most *once* when it is *white*
- And so the while loop iterates at most $\mathcal{O}(|V|)$ times

- The adjacency list of each node is examined at most *once* in total
- Therefore the total running time is $\mathcal{O}(|V| + |E|)$

BFS Claim to Find Shortest Paths

- Define $\delta(s, v)$ = minimum distance from s to v, i.e. smallest number of edges on any path from s to v
- Claim: at the end of BFS, $d[v] = \delta(s, v)$
- Since our graphs are unweighted, all edges in the path count as 1

Lemma 9.1. *$\delta(s, v) \leq \delta(s, u) + 1$ for all $(u, v) \in E$*

Proof. (sketch)
If u is reachable from s, then so is v. $\delta(s, v)$ cannot be longer than $\delta(s, u) + 1$ for going down edge (u, v)
If u is not reachable from s, then $\delta(s, u) = \infty$

□

Lemma 9.2. *$d[v] \geq \delta(s, v)$ for all $v \in V$, at any point during BFS*

Proof. (sketch, by induction on enqueue operations)
When s is enqueued, $d[s] = 0 = \delta(s, s)$ and $d[v] = \infty \geq \delta(s, v)$ for all $v \in V - \{s\}$
When v is discovered from its parent u, then $d[u] \geq \delta(s, u)$ by IH. $d[v] = d[u] + 1 \geq \delta(s, u) + 1 \geq \delta(s, v)$ from Lemma 9.1. Therefore $d[v] \geq \delta(s, v)$, then v is enqueued with this $d[v]$ and painted grey and $d[v]$ does not change.

□

Lemma 9.3. *If $Q = [v_1, \ldots, v_r]$, then $d[v_i] \leq d[v_{i+1}]$ for each i and $d[v_r] \leq d[v_1] + 1$*

Proof. (sketch, by induction on queue operations)
When s is enqueued, the lemma holds.
Assume the Lemma for $Q = [v_1, \ldots, v_r]$.

- When v_1 is dequeued, v_2 becomes the head. By IH, $d[v_1] \leq d[v_2]$, therefore $d[v_r] \leq d[v_1] + 1 \leq d[v_2] + 1$, and so the Lemma holds.
- When v is enqueued, it becomes v_{r+1}. The parent of v is u and it has just been dequeued. v_1 is the new head. So $d[v_1] \geq d[u]$. Therefore $d[v_{r+1}] = d[v] = d[u] + 1 \leq d[v_1] + 1$. Also, $d[v_r] \leq d[u] + 1 = d[v] = d[v_{r+1}]$. And so the Lemma holds.

□

Theorem 9.4. *At the end of BFS, $d[v] = \delta(s, v)$ for all $v \in V$*

Proof. Assume $d[v] > \delta(s, v)$ for some v with minimum $\delta(s, v)$ for a contradiction. This means that for u, $d[u] = \delta(s, u)$. v is the first vertex on some path when $\delta(s, v)$ is shorter than $d[v]$. Let u be the vertex just before v on that path. So when we discovered v and set $d[v]$, we were examining (u, v).

$$d[v] > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1$$

Now consider the point when we dequeue vertex u, then we explore edge (u, v) and v is coloured.

- White: set $d[v] = d[u] + 1$, enqueue $d[v]$, the colour it. Since we never change $d[v]$ afterwards, this is a contradiction
- Black: v was in the queue before u, therefore $d[v] \leq d[u]$ from Lemma 9.3, which is a contradiction
- Grey: v was painted grey by some other vertex w. $d[v] = d[w] + 1 \leq d[u] + 1$. This is a contradiction

□

Depth First Search

- Search through the graph going as *deep* as possible before we *backtrack* to explore the edges of already-discovered vertices
- Each vertex is coloured as we go
  - **White**: not yet discovered
  - **Grey**: discovered but not fully explored
  - **Black**: fully explored
- Store timestamps for each vertex
  - $d[v]$ is the discovery time of v
  - $f[v]$ is the finish time of v
- Algorithm is *recursive*
- No notion of "source vertex", algorithm calls `DFS-VISIT(G, s)` repeatedly on each $s \in V$
  - Usually in adjacency-list order
- Algorithm

```
DFS(G=(V,E)):
    for v in V:
        colour[v] = white
        f[v], d[v] = inf, inf # times
        pi[v] = NIL # parent
    time = 0 # global
    for v in V:
        if colour[v] == white:
            DFS-VISIT(G, v)

DFS-VISIT(G=(V, E), u):
    colour[u] = grey # Change colour
    d[u] = ++time # Set discover time
    for (u, v) in E:
        if colour[v] == white:
            pi[v] = u # Set parent of new node
            DFS-VISIT(G, v) # recurse
    colour[u] = black # Fully discovered
    f[u] = ++time # Set finish time
```

DFS Complexity

- `DFS-VISIT` is called on *white* vertices and then those vertices are immediately painted *grey*
- Therefore `DFS-VISIT` is called *once* on each vertex
- Outside of recursive calls, each execution of `DFS-VISIT` examines the *adjacency list* for one vertex
- And so the total running time is $\mathcal{O}(|V| + |E|)$, i.e. size of adjacency list

DFS Forest

- **Tree edge**: (u, v) is an edge in the DFS-forest
  - v is *white* when (u, v) is examined

- **Back edge**: (u, v) such that v is an ancestor of u in the DFS-forest
  - v is *grey* when (u, v) is examined
- **Forward edge**: (u, v) such that v is a descendant of u in the DFS-forest
  - v is *black* when (u, v) is examined
- **Cross edge**: (u, v) such that v is neither an ancestor nor a descendant of u in the DFS-forest
  - v is *black* when (u, v) is examined

Parenthesis Theorem

- In any DFS of graph G, for any two vertices u and v, exactly one of the following 3 conditions holds:
  1. The intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint
  2. The interval $[d[u], f[u]]$ is entirely contained in interval $[d[v], f[v]]$
  3. The interval $[d[v], f[v]]$ is entirely contained in interval $[d[u], f[u]]$

Topological Sort

- For *directed acyclic* graphs only
- A *linear* ordering of all vertices in graph G such that if u is an ancestor of v, then u appears before v in the ordering
- Algorithm: run DFS, compute finishing times as each vertex finishes, and put vertices in order of decreasing finishing time

Strongly Connected

- In an *undirected* graph, it is connected, if every vertex v is reachable from every other vertex for any pair of vertices u and v
- **Strongly connected** in a *directed* graph means that for every vertex pair (u, v) in G:
  1. v is reachable from u
  2. u is reachable from v
- Algorithm:
  - Pick $s \in V$ and run `DFS-VISIT(G, s)`
  - If $f[s] == 2|V|$, then we have a path $s \to v$ for all $v \in V$
  - Reverse all edges in G to make G' and run `DFS-VISIT(G', s)` to test if we have a path $v \to s$ for all $v \in V$
- In a directed graph $G = (V, E)$, a **strongly-connected component** is the *maximal set* of vertices $C \subseteq V$ such that for every pair of vertices $u, v \in C$, we have a path from u to v and a path from v to u

Minimum Spanning Trees

- Greedy MST algorithm

```
GREEDY-MST(G=(V, E), w: E -> R):
   T = {} # subset of some MST of G
   while T is not a spanning tree: # |T| < |V| - 1
      find e "safe for T" # e is safe iff T union {e} is a subset of some MST of G
      T = T union {e}
```

- If G is a connected, undirected, weighted graph, T is a subset of some MST of G, and e is an edge of *minimum weight* whose endpoints are in different connected components of T, then e is safe for T.

Prim's Algorithm

- Idea: pick a root vertex and "grow" T by connecting an isolated vertex to T
- At each step, connect the vertex not currently in T that has the cheapest edge connecting into T
- Algorithm

```
MST-PRIM(G=(V, E), w:E->R, r):
    for v in V:
        priority[v] = inf # Cost to get this vertex into T
        pi[v] = NIL
    priority[r] = 0
    Q = V
    while Q is not empty:
        u = ExtractMin(Q) # u is safe for T
        T = T union {pi[u], u} # except when u = r
        for v in adj[u]:
            if v in Q and w(u, v) < priority[v]: # update priority (minimum weight)
                decreasePriority(Q, v, w(u, v)) # priority[v] = w(u, v)
                pi[v] = u
```

- For queue, use priority queue, heap implementation
  - `decreasePriority` costs $O(\log(|V|))$
- Running time is $\mathcal{O}(|E| \log |V|)$

Kruskal's Algorithm

- Idea: build a *forest* and *merge* trees in the forest until we have only 1 tree
- At each step, pick the *cheapest* edge of graph G that does not make a cycle
- Algorithm

```
1 MST-KRUSKAL(G=(V, E), w:E->R):
2     T = {}
3     sort edges so w(e_1) <= w(e_2) <= ... <= w(e_m)
4     for i in range(1, m) inclusive:
5         u_i, v_i = e_i
6         if u_1, v_i in different connected components of T:
7             T = T union {e_i}
```

- Sorting takes $\mathcal{O}(|E| \log |E|)$ time
- Finding if 2 vertices are in different components of T is also difficult
  - Using BFS/DFS wold result in $\mathcal{O}(|E||V|)$ time since the loop on line 4 executes $|E|$ times and cost of DFS/BFS is $\mathcal{O}(|V|)$

10 Disjoint Set

Disjoint Set ADT

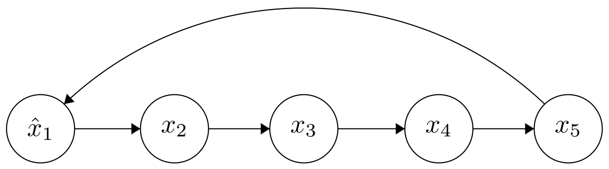
- A set where there is no common elements/intersection
- Each element in at most 1 set
- Operations
  - `MAKE-SET(x)`: given an element x that doesn't already belong to a set, create a new set containing only x and designate x as the rep
  - `FIND-SET(x)`: given an element x, return the *representative* of the set containing x (or NIL if x is not in any set)
  - `UNION(x, y)`: given two elements x and y such that
    - S_x is the set containing x
    - S_y is the set containing y

    make a new set $S_x \cup S_y$, designate a representative, and remove S_x and S_y from the ADT
    - As a precondition, it is required that x and y each be an element of some set

Disjoint Set Implementations

- Disjoint set ADT manages *sets* only, client code manages elements
- Each element x manages a *pointer* to DJS data structure
- Complexity analyzed using *worst case sequence complexity* in keeping with context of Kruskal's algorithm
- Each analysis based on m operations where n is the number of `MAKE-SET`s

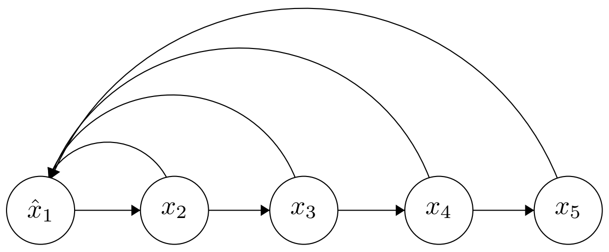
Implementation: Circular Linked List



$$\begin{tikzcd}
\hat{x}_1 \arrow[r] & x_2 \arrow[r] & x_3 \arrow[r] & x_4 \arrow[r] & x_5 \arrow[llll, bend right]
\end{tikzcd}$$

- $n = m/4$ `MAKE-SET`s, $m/4 - 1$ `UNION`s, $m/2$ `FIND-SET`s
  - `MAKE-SET`s: $\mathcal{O}(m)$ for the sequence of `MAKE-SET`s
  - `UNION`s: constant once we perform `FIND-SET`s, so $\mathcal{O}(m)$ for the sequence
  - `FIND-SET`s: $\mathcal{O}(\text{length of the list}) = \mathcal{O}(m)$ each, thus $\mathcal{O}(m^2)$ for the sequence
  - Therefore WCSC is $\mathcal{O}(m^2)$

Implementation: Circular Linked List with Backpointers

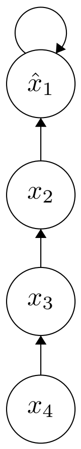


- $n = m/2$ `MAKE-SET`s, $m/2 - 1$ `UNION`s, 1 or 2 `FIND-SET`s
  - We always append the longer list to the shorter list, so `UNION` costs $\mathcal{O}$(length of the list)
  - Cost of union: $\sum_{i=1}^{m/2-1} i = \Theta(m^2)$ for the sequence
  - Therefore WCSC is $\mathcal{O}(m^2)$

Implementation: Circular Linked List with Backpointers that Unions by Weight

- When performing `UNION`s, append the shorter list (less weight) to the longer list (greater weight)
- `MAKE-SET` costs $\mathcal{O}(1)$
- `FIND-SET` costs $\mathcal{O}(1)$
- `UNION` costs $\mathcal{O}(n)$
- We want an upper bound on the number of times we have to update the backpointer of an arbitrary element X
  - x's pointer only changes when x is in the *smaller* set
  - After union, the size of the set x was in has at least doubled
  - This means x's pointer cannot be updated more than $\log n$ times since n is the number of elements
- WCSC is $\mathcal{O}(m + n \log n)$

Implementation: Inverted Tree



- Use an inverted tree where each element points to its parent
- Root is the rep
- Root points to itself
- MAKE-SET costs $\mathcal{O}(1)$
- FIND-SET costs $\mathcal{O}$(depth of x) by following parent pointers
- UNION costs $\mathcal{O}(1)$ + time of FIND-SET
- Bad sequence: $n = m/4$ MAKE-SETs, $m/4 - 1$ UNIONs, $m/2$ FIND-SETs
  - For UNIONs, we append the long chain onto the singleton to have a linked-list-like tree
  - For FIND-SETs, we operate on leaves
  - Therefore WCSC is $\mathcal{O}(m^2)$

Implementation: Tree with Union by Weight

- Make the larger tree the new root upon UNION
- Augment the tree to store the weight, i.e. the number of elements in the tree
- MAKE-SET costs $\mathcal{O}(1)$
- UNION costs $\mathcal{O}(1)$ once we have the reps
- Complexity of FIND-SET:
  - During any sequence of operations, n of which are MAKE-SET, therefore the max height of the tree is $\log n$
  - Therefore FIND-SET takes $\mathcal{O}(\log n)$
  - Total time for m operations is $\mathcal{O}(m \log n)$
- WCSC is $\mathcal{O}(m \log n)$

Implmentation: Tree with Path Compression

- During FIND-SET, keep track of nodes visited on the path from x to the root
- Once root (i.e. rep) is found, update the parent pointers of each node encountered to point directly to the root
- At most doubles the time for FIND-SET but speeds up further operations
- WCSC is $\Theta(m \log m)$

Implementation: Tree with Path Compression and Union by Rank

- Instead of looking at the weight or height, we could maintain an upper bound on the height
- **Rank**: upper bound on the height
- Rank of a leaf is 0
- Rank of an internal node is 1 + max rank of its children
- MAKE-SET: set rank to 0
- FIND-SET: ranks remain unchanged, even if path compression is performed

- `UNION`: node with the higher rank is the new root, ranks are unchanged
  - If the ranks are the same, choose either as the new root and increase its rank by 1
- WCSC is $\mathcal{O}(m \log^* n)$
  - $\log^*$ is the iterated log, the number of times the log function must be iteratively applied before the result ≤ 1

$$\log^*(n) = \begin{cases} 0, & \text{if } n \leq 1 \\ 1 + \log^*(\log n), & \text{elsewise} \end{cases}$$

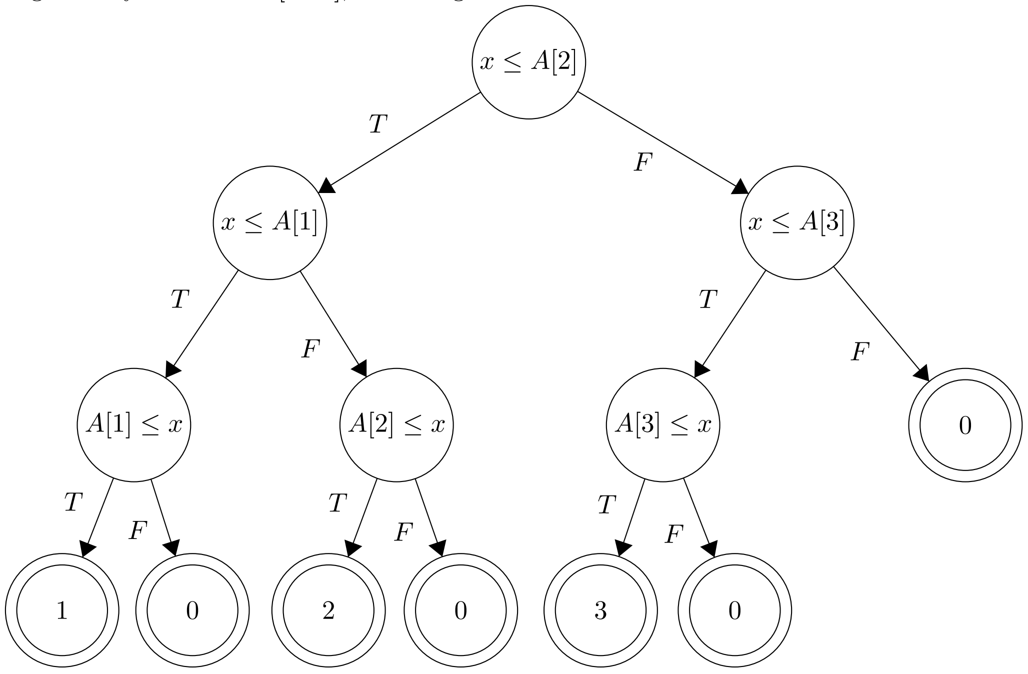
11 Lower Bounds on Sorting

Sorting Performance

- We have algorithms that run in the worst case time of $\mathcal{O}(n \log n)$, but we don't know whether we can do better
- We need to prove the worst-case complexity of *problems*
- For problem P, $C(P)$ is the best worst-case running time of any algorithm that solves P
  - Upper bound on $C(P)$: gives one algorithm and analyze its time
  - Lower bound on $C(P)$: *every* algorithm requires a certain amount of time
- We can prove lower bounds on *classes* of algorithms

Comparison Trees

- Represent algorithms that work by *pairwise comparison* of elements
- E.g. binary search in $A[1:3]$, returning index of x or 0



Information Theory Lower Bounds

- Every binary tree with height h has $\le 2^h$ leaves
- Every binary tree with L leaves has height $\ge \lceil \log_2 L \rceil$
- Every comparison tree that solves the problem P has at least 1 leaf for each possible output
- Every comparison tree for P has height $\ge \lceil \log_2 m \rceil$ where m is the number of possible outputs

Sorting

- Input: $A[1:n]$
- Output: permuation of $[1, \ldots, n]$ indication position of each element

- $n!$ possible outputs
- Every comparison tree has height $\geq \log_2(n!)$
- Every algorithm that uses only comparisons require at least $\log_2(n!)$ comparisons
- $\log_2(n!) = \Theta(n \log n)$

$$\begin{aligned}\log(n!) &= \sum_{i=1}^{n} \log(i) \\ &< \sum_{i=1}^{n} \log(n) \\ &\in \mathcal{O}(n \log n)\end{aligned}$$

$$\begin{aligned}\log(n!) &> \sum_{i=n/2}^{n} \log(i) \\ &> \sum_{i=n/2}^{n} \log(n/2) \\ &\in \Omega(n \log n)\end{aligned}$$

- Therefore, every algorithm that uses only comparisons between pairs of elements to sort requires at least $\Theta(n \log n)$ time

Other Sorting Algorithms

- Some can be done in less than $\mathcal{O}(n \log n)$ comparisons because they work on a *restricted problem*, e.g. radix sort, counting sort
- Lower bounds apply only to algorithms of a *particular type*