Disclaimer

This complementary study package is provided by Easy 4.0 Education Inc. and its affiliated mentors. This study package seeks to support your study process and should be used as a complement, **NOT** substitute to course material, lecture notes, problem sets, past tests and other available resources.

We acknowledge that this package contains some materials provided by professors and staff of the University of Toronto, and the sources of these materials are cited in details wherever they appear.

This package is distributed for free to students participating in Easy 4.0's review seminars, and are not for sale or other commercial uses whatsoever. We kindly ask you to refrain from copying or selling in part or in whole any information provided in this package.

Thank you for choosing Easy 4.0. We sincerely wish you the best of luck in all of your exams.

Easy 4.0 Education Inc.

Part A. Time Complexity Analysis

1. Different Cases

Suppose t(x) is the runtime of an algorithm with input x.

- **Worst case**: T(n) = max{ t(x) : |x| = n } 当 input 大小为 n 时最坏情况下的运行时间
- **Best case**: T(n) = max{ t(x) : |x| = n } 当 input 大小为 n 时最好情况下的运行时间
- **Average case**: T(n) = E[t(x) | |x| = n] 当 input 大小为 n 时的平均运行时间 [Discussed in Part B]

  这里要注意 worst case 和 best case 描述的是一个算法在相同size下, 最坏和最好的的情况。 一般来讲， 如果算法当中存在不同的branch (path) , 会导致算法提前结束, 或者省略某些步骤, 一般会出现best case和worst case的情况.

2. Asymptotic Notation

- Big-Oh:
  f = O(g) means the function f(x) grows slower or at the same rate of g(x).
- Big-Omega
  f = Ω(g) means the function f(x) grows faster or at the same rate of g(x).
- Big-Theta
  f = Θ(g) means the function f(x) grows at the same rate of g(x). Also, Big-Theta is Big-Oh AND Big-Omega.

  同样的, 这里也要注意, Big-Oh代表的是Upper bound of running time, 和worst case一毛钱关系都没有. 相同的道理, Big-Omega也和Best case没有丝毫关系.

3. 常见证明

- 相关Worst Case的证明
  Worst Case是我们最常见的证明, 证明的格式和我们之前学习过的相同, 我们有很多种常见方法, 在165中对iterations的直接分析, 在236中利用induction来计算runtime等.
  - Show Big-Oh of Worst Case.
    要说明 **for every** input x of size n, t(x) is **no larger** than c * something, where c is a positive constant.

  - Show Big-Omega of Worst Case.
    要说明 **there exists at least one** input x of size n, t(x) is **no smaller** c * something, where c is a positive constant.

- 相关Best Case的证明
  快速记忆就是直接和worst-case反过来!

  - Show Big-Oh of Best Case.
    要说明 **there exists at least one** input x of size n, t(x) is **no larger** than c * something, where c is a positive constant.
  - Show Big-Omega of Best Case.
    要说明 **for every** input x of size n, t(x) is **no smaller** c * something, where c is a positive constant.

- 注意事项

  - 对于there exists 一定要确保说明存在, 必要的时候给出一个实际例子, 因为在有些情况下, 我们脑补的lower bound的情况可能根本就不存在.
  - 证明Big-Theta的时候, 要注意需要证明Big-Oh和Big-Omega
  - 引用CLRS的时候, 一定要注意说明所有augment和在原有algorithm上更改的内容不会影响runtime.

Part B. Average Case Analysis

本部分和 statistics 关系非常大, 对于function的analysis基本都很简单, 主要是要定义好 simple space, 找好distribution, 最后求出 Expected Value, 也就是我们的 Average Case Running Time Complexity.

1. 找 Simple Space.
   一般来说 Simple Space 就是所有可能的inputs. For example, 对于sorting问题来说, simple space是size n的list的所有permutation.

2. 确定 Distribution
   最常见的情况是, 我们每个input都有 equal probability to happen, 这样就是uniform distribution. 当然还有例如 binomial distribution等等.

3. 计算Expected Value
   根据Distribution, 确认每种input的Probability和cost, 并计算Expected value.

Part C. Amortized Analysis

Amortized Analysis关心的是在一个sequence of operation 被执行的时候的平均时间, 在这里要注意amortized analysis和average case analysis的区别. average case是说平均来讲一个 operation 要花的时间, 而amortized analysis是说连续很多次operation后, 平均每一次 operation要花的时间.

我们在CSC263的学习中, 学习了两种Amortized Analysis的方法, 分别是Aggregate Method 和 Accounting Method.

- Aggregate Method
  Aggregate method是十分简单的两步法, 首先我们算出 for all n, n operations has a worst case running time T(n), 这一步一般都是简单的计算或求和公式. 第二步, 我们直接通过 T(n) / n 来得到 Aggregate Method的结果.

- Accounting Method
  Accounting method 与 Aggregate method 不同的是, 我们不在从overall入手考虑问题了, 而是考虑在每一步上都要为下一步做准备. 首先我们把每个constant的operation都assignment一个token, 或者说一块钱. 这时我们主要需要证明的是: 我们可以找到一个固定的存款surplus, 保证我们不破产， 也就是在找我们的 Credit Invariant.

  1. Analysis
     直接从Token入手, 说明每一个operation结算后, token数量(Credit Invariant 不变). 比如我们在分析Dynamic Array时, 我们可以通过说明每个insert assignment 3个token来保证每次copy to another list的时候有足够的token. 在这个过程中, 我们也可以通过induction来证明我们有不变的Credit Invariant.

  2. Proof the Invariant
     首先我们可以设立Ind. Hyp. 说明在任意operation执行后, 我们都有足够的token. Basecase一般为还没有任何operation perform开始, WTS 对于任何operation执行之后, token的数量还是大于等于我们得ind. Hyp.的. 这样我们就说明了我们得credit invariant是可以maintain的.

  3. using invariant to just best constant
     紧接着, 我们要说明, 由于我们一直都可以有富裕的tokens, 说明对于任何sequence of n operations, total cost = total charge. 并且我们可以根据我们对每个operation assign 的token花销来说明 total charge是一个固定值 for example c * n. 根据amortized cost 的定义, 我们可以说明 amortized cost是perfectly fit我们的operations的, 一分钱不多, 一分钱不少.

Part D. Binary Heap [MAX Heap]

1. Basic of Binary Heap

Binary MAX Heap 首先要满足的是一个 complete binary tree. 然后要满足的是每一个node的value都要小于等于它的parent (对于 MIN Heap是大于等于 parent). 因此, 我们首先可以得到以下几条最基本的性质:

- Root 包含是整个data structure中最大的value.
- Left child和right child没有任何关联, 在heap里面只有parent-child之间有关系.
- Heap的height是 $\lfloor \log n \rfloor$, 所以要记住, 从root到leaf的操作都是 O(logn).
- 一个有n个elements 的 Binary Heap中有 $\lceil (n/2) \rceil$ 个 leaves.
- 最小的element一定在leaf上.

2. Implementation

在implement一个Binary Heap的时候我们有两种不同的implement方式, 首先我们可以像implement一个binary tree一样来implement binary heap, 在这里就不多赘述了.

重要的一点是, binary heap可以用array来implement. 在这里要注意, 我们是从index 1开始的. 我们相当于对一个tree representation的binary heap进行了一次level order traversal. 假如一个node是index k. left children就是2k, right children就是2k + 1. 这个node的parent就在 $\lfloor n/2 \rfloor$.

3. Operations

Bubble Up & Bubble Down - O(logn): Keep comparing with Parent or Children, and swap two nodes if necessary until the heap property is recovered. This will at most travel the height of the binary heap.

MAX - O(1): Just returned the root.

INSERT - O(logn): Put the node to the end of the heap, and bubble up.

EXTRACT_MAX - O(logn): swap the root with the last element, pop, and bubble down the root.

BUILD_HEAP - O(n): Reorder the unsorted array into a heap. Note: it has to be an array.

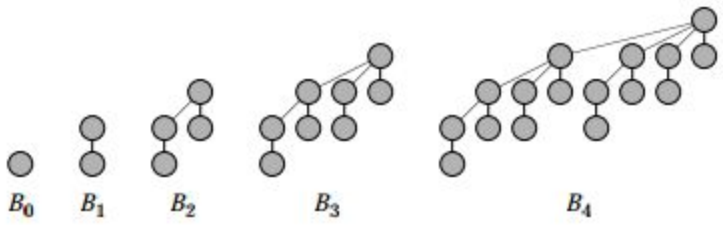
4. 常见题型

1. 画Heap 找index.
2. 最大/最小的K个elements.
3. 漏斗问题
4. K个array merge sort.

Part E. Binomial Heap

1. Binomial Tree

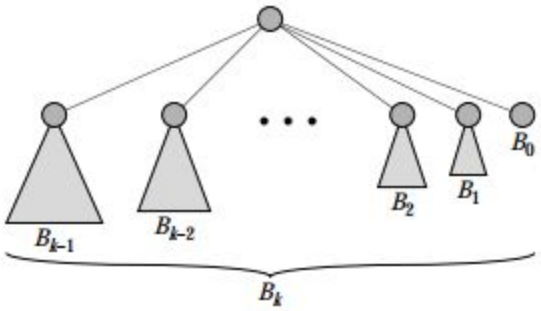
定义： The Binomial tree is B_k is an ordered tree defined recursively. the binomial tree B_0 consists of a single node. The binomial tree B_k consists of two binomial trees B_k–1 that are linked together.



Properties:

For the binomial tree B_k,

1. there are 2^k nodes,
2. the height of the tree is k,
3. there are exactly $\binom{k}{i}$ nodes at depth i for $i = 0, 1, \ldots, k$, and
4. the root has degree k, which is greater than that of any other node; moreover if the children of the root are numbered from left to right by $k-1, k-2, \ldots, 0$, child i is the root of a subtree B_i.



2. Binomial Heap

定义： Binomial heap是一个满足binomial heap property的 binomial tree.

1. Each binomial tree in H obeys the ***min-heap property***: the key of a node is greater than or equal to the key of its parent. We say that each such tree is ***min-heap-ordered***.
2. For any nonnegative integer k, there is at most one binomial tree in H whose root has degree k.

Property:

- 一个binomial heap最多只能有$\lfloor \log n \rfloor + 1$ 个binomial trees.
- 最小的element一定是root of one of the binomial trees.

3. Operations

- **UNION – O(log n)**

  根据 Binomial Heap的定义, 是不可以有两个order相同的binomial tree存在的, 因此如果遇到两个相同order的binomial tree, 我们可以把他们union成一个order + 1的binomial tree, 然后我们把两个tree中root较小的一个作为新的root 以满足 binomial heap的property. 由于我们可能在Union之后, 又发生了和其他order 的 tree重复的情况, 因此在最坏情况下, 我们要连续union掉所有binomial trees.

- **INSERTION – O(log n)**

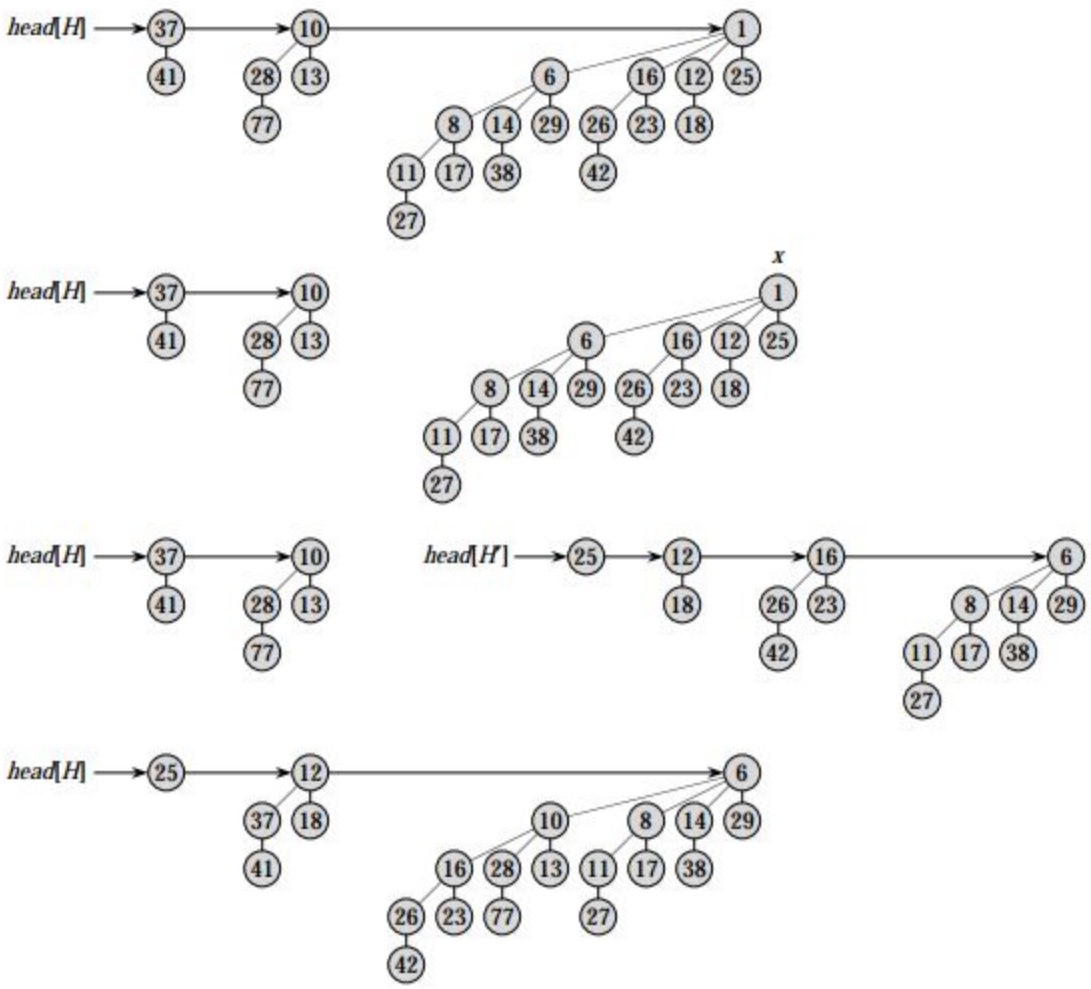
  新建一个order 0的binomial tree, 然后执行 Union.

- **MINIMUM – O(log n)**

  根据我们之前分析过的, 我们要最多遍历 log n 个binomial tree来找到最小的element.

- **EXTRACT_MIN – O(log n)**

  找到最小element后, 我们直接删除掉这个node, 这样我们会把这个tree变成 order较小的很多个tree, 然后我们执行union.



- **EXTRACT_MIN**

Part F. AVL Tree

1. 定义

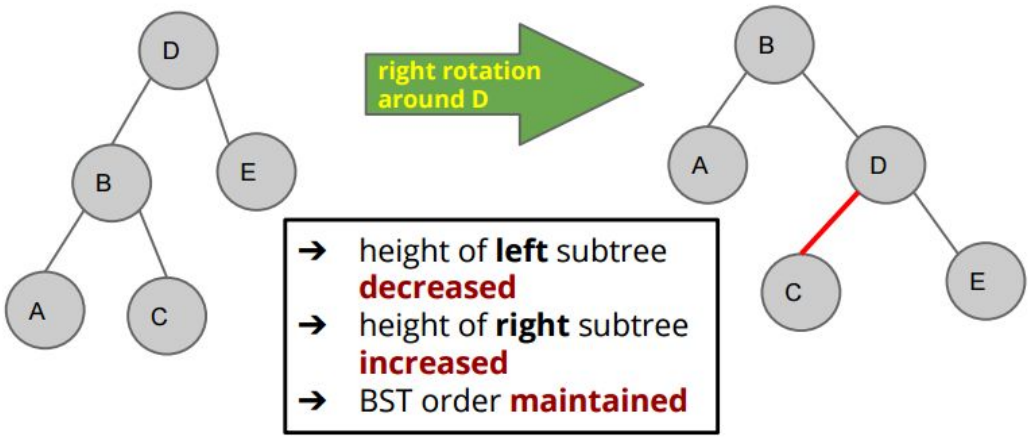
一个 height 近似于 balanced 的 Binary Search Tree.

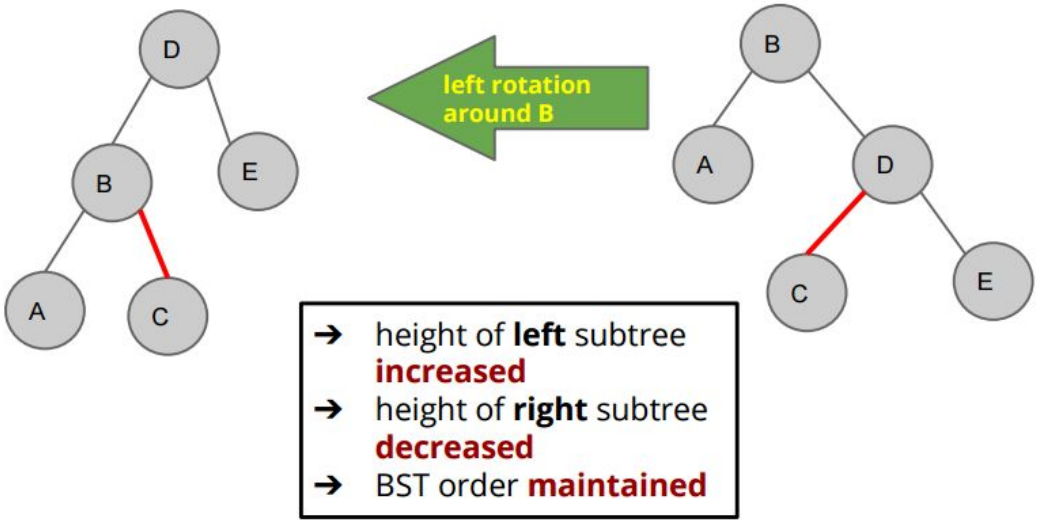
2. Operation

在 AVL Tree的学习中, Operation的记忆是非常重要的, AVL Tree是通过各种operation来保证自身能够在各种modification之后保持height balanced. 在AVL Tree中， 任意left和right subtree of a node, 高度的差距最多只能是1.

- **Rotation**
  Rotation 是AVL Tree中一个非常重要的概念, 我们要通过 rotation 来保证AVL Tree在更改后还能保持balanaced. 既然如此， 和rotation关联十分紧密的就有一个概念：**Balance factor**. Balance factor 指的是左边和右边 subtree height的差, 根据AVL Tree的定义, balance factor最多是正负一，如果遇到了BF不大于1或者小于-1的情况, 说明我们要进行rotation来恢复AVL Tree的property.

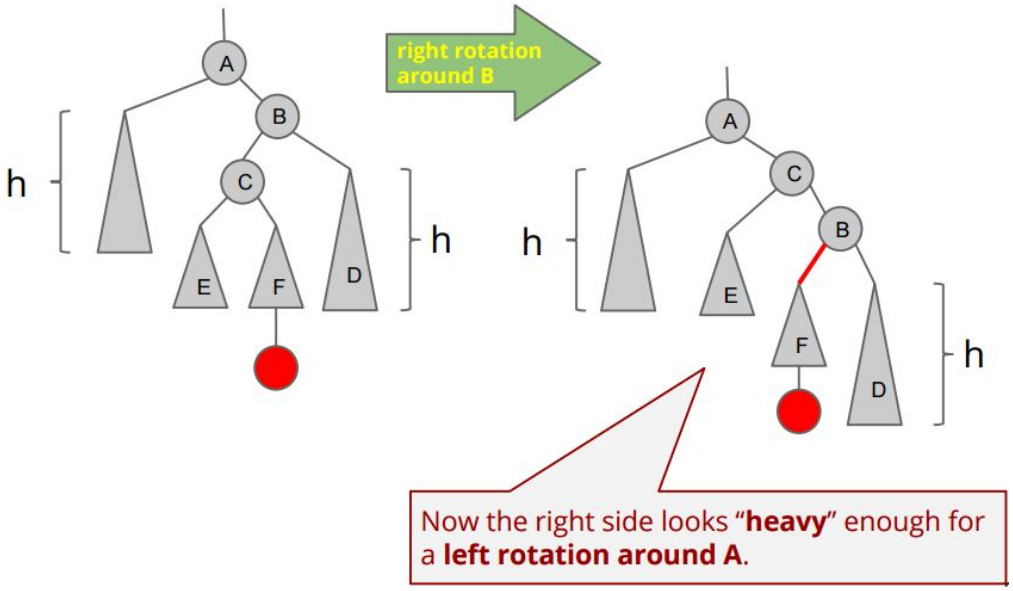
  首先是最基本的Left 和 Right Rotation:

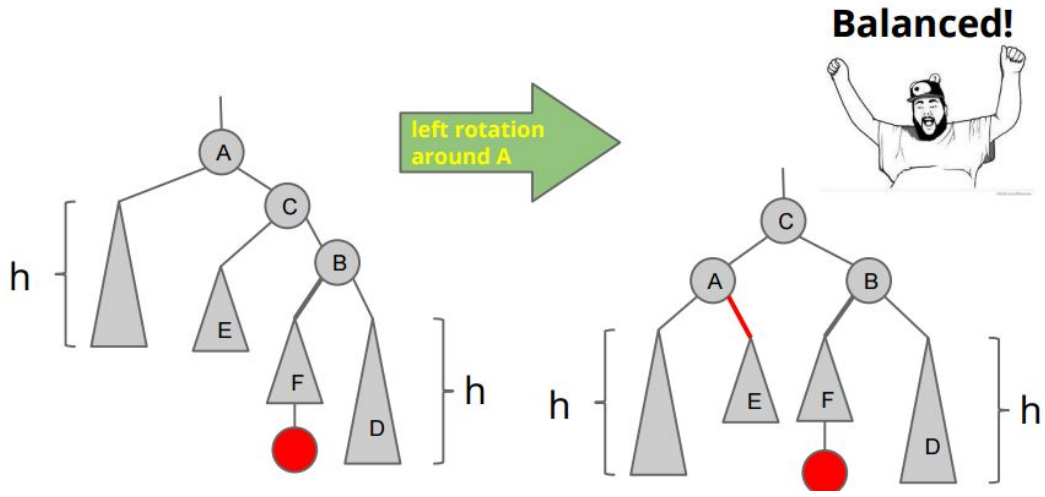




对于最左和最右重的问题, 我们一般只需要一次left 或 right rotation就可以解决问题.

但是... 如果我们是中间的node重, 那么我就需要做**double rotation**, 首先先通过一次left or right rotation把这个heavy的branch移动到最左或者最右侧, 然后再进行一次rotation把整个tree balance好.





- **Search – O(log n)**

  Binary Search的runtime 是和height相关的, 由于AVL Tree是balanced, 我们可以确定为O(log n).

- **Insert – O(log n)**

  首先在balanced binary search tree上insert一个node要take O(log n)的时间, 然后在这里我们需要再加上一个constant time来进行rotation 和 BF updating.

- **Delete – O(log n)**

  和insert 类似, delete 一个node take了 O(log n)的时间 + constant time来进行rotation 和 BF updating.

3. Augmentation

Augmentation 是我们在design algorithm时候最经常用到的技巧, 我们以一些现有的algorithms为蓝本, 增加一些fields, 然后通过这些额外的信息, 我们可以将一些原本复杂的计算过程简单化.

- **什么样的attribute可以augment进入 AVL Tree ?**

  通常, 我们会选择一些可以 reduce 的 factor 来augment 到每个node 上, 从而减少重复的计算. 换言之, 这个factor一定要满足可以在从root向下search时, 或者操作完成后propagate回root时, 完成对这个factor的update, 来保证这个factor的准确性. 所有满足这一点的factor都可以作为我们augment的fields.

- **常见的augmentation有哪些?**

  一般来讲, 我们经常augment的有 size, sum, 还有对一些特定内容的count.

- **注意事项**

  1. 描述Augmenting的field的作用和内容.
  2. 说明在所有operation时, 如何propagate (update) 我们augmenting的field.
  3. 说明我们得propagation并不会对runtime造成影响. 一般来说propagation会发生在search上, 或者回溯到root的时候, 也就是只在我们经过的path, 和rotation触及的node上增加一些constant cost. 总体而言不会改变我们的runtime.

Part G. Hashing

Hash简单的说就是key对应value, 然后可以通过计算key来快速找到对应的value.

1. SUHA

Simple Uniform Hash Assumption, CSC263 中所讨论的hashing都要遵循这一assumption. 根据这个assumption, 我们可以说明每个bucket中, 放入东西的概率是相同的.

2. Collision & Load factor

如果, 我们在hashing的时候不存在collision, 那么我们可以保证在constant的时间内访问到key对应的value. 但是在 Hash的时候, 经常我们item的数量是要远大于我们buckets的数量的, 也就是说会出现collision的情况, collision会导致我们hash中每一个slot上有多个item. 而我们在多个item上寻找我们想要的target就会花更多的时间.

首先, 在SUHA的条件下, 我们可以assume我们每一个bucket都是平均地被load的. 因此如果我们有m个bucket, n个items的话, 我们可以说我们的expected number of items the same for each bucket $\alpha = n / m$.

3. Hashing Function的选择.

在选择Hash Function的时候我们有以下两点要注意:

- Should depend on every part/bit of input.
- Should spread out.
- Should be computed in a easy time.

4. Bloom Filter

Bloom Filter 是一个常见的hashing的应用, 比如chrome的反钓鱼网站域名检测功能, 就是用bloom filter 来完成的.

Bloom Filter 在implement的时候首先是将全部m个buckets全部设置为0, 然后在bloom filter上会有k个independent的hash functions.

- Insert

  在Insert的时候, 我们分别将k个 hash functions apply到item上, 能够得到 k 个不同的key. 我们分别将这k个不同的key对应的bucket修改成1.

- Search

  在search的时候, 类似于insert的时候, 我们通过k个hash functions, 找到k个不同的buckets, 如果这些buckets上都是1, 那么说明这个item是存在于我们的bloom filters里面.

- False Positive

  在Search的时候, 如果k个buckets全部是1的时候, 我们就认为,这个item是被bloom filter记录过的. 但是这时可能会出现这个item并没有真的被记录过, 这些buckets是由于其他items的insertion而变成了1. 这种情况, 我们称之为 false positive.

  False Positive的概率有一个直接的记忆公式. 假设我们有n个items, m个hash functions. 我们可以得到 false positive 的概率是:

$$(0.6185)^{m/n}.$$

Part H. Randomized QuickSort

Quicksort是我们在之前学习中就学习过的内容, 在当时我们发现, QuickSort的runtime很大程度上取决于对pivot的选择, pivot选得好, quicksort完成得就快. 我们发现如果我们随机的选取pivot的话, 我们的expected runtime是 Θ(nlog n)的.

在randomized quicksort的问题上, 我们经常考虑的是两个不同的item会互相比较的概率. 我们在考虑这个问题的时候, 有一个非常简单的计算方法. 比如我们有一个长度是6的list, 我们首先把这个list 的sorted form写出来, 举个栗子, 是 [a1, a2, a3, a4, a5, a6]. 我们要计算a1和a6的概率. 那么根据quicksort的算法逻辑. 任何在a1和a6之间的items被选择成pivot的话, a1和a6都不会再相见了. 也就是我们这里的概率是 2 / (6 – 1 + 1) = 1 / 3.

In general, 我们可以说, 在sorted form下. 从a_i 和 a_j (assume j >= i)直接比较的概率是 2 / (j – i + 1).

Part I. Graph

Graph 是263最最核心的内容, 我们之前所学的tree, linkedlist,其实都是graph的一种. Graph是由 vertex 和 edge 两部分组成的. Vertex 相对比较简单, 每个vertex代表一个节点, 存储着各种fields. Edges是我们在研究graph的时候, 非常重要的环节, graph上很多的属性,都是和edges有牵连的. 在余下的内容中, 除非特殊声明, 我们以n指代|V|, 以m指代|E|

- **Connected Component**
  Connected Component is a **subgraph** in which any two vertices are **connected** to each other by paths, and which is **connected** to no additional vertices in the **supergraph**

- **Directed / Undirected**
  Edges是否有方向, 对于任意edge (u, v), 如果是undirected的, 那么(u, v) 和 (v, u) 是一回事儿, 否则的话, 是两条不同的edges.

- **Weighted / Unweighted**
  Weighted的意思就是edge上是否有权重, 比如距离, cost等.

- **Acyclic / Cyclic**
  Acyclic 就是无环图, cyclic里面存在最少一个cycle.

- **Adjacency Matrix**
  一个n x n的array.

```
           { 1   if (v_i,v_j) in E
A[i,j] = {
           { 0   if (v_i,v_j) not in E
```

  优势: Check edge existence.
  Space Complexity: O(n^2)

- **Adjacency List**
  每个vertex一个list, list里面存储着所有这个vertex能reach到的neighbours.
  优势: space efficient.
  Space Complexity: O(m + n)

Part J. BFS

Breadth First Search, 广度优先搜索. 在搜索的时候, 我们是通过一个Queue来排序所有即将被访问的candidates的. 在搜索的时候要注意, 我们使用了一种coloring strategy, 我们把

vertex根据访问情况区分成三种不同的颜色.

White 代表还没被访问 unvisited
最开始, 所有vertex都是白色的.
Grey 代表已经被发现的 encountered / discovered
第一次发现一个vertex的时候, 把他变成灰色.
Black 代表已经被探索过的 explored
当一个vertex所有的neighbor都被discovered之后, 把他变成黑色.

BFS 最常见的应用是 **Shortest Path**. 在BFS中, 我们是按照level的顺序进行访问的, 我们因此可以得到从source vertex到每个vertex最短的path数量.

- **常见题型**
  - **Shortest Path**
  - **Bipartite**

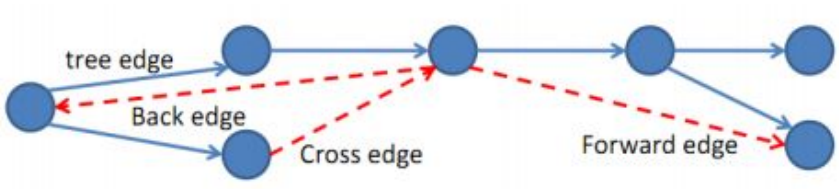
Part K. DFS & Topological Sorting

- **DFS 算法**

DFS和BFS相似的, 也会把node color成三种颜色. 但是不同的, 我们现在使用stack来管理candidates. 因此, 我们会一条路走到黑. 走到尽头之后, 才回过头来, 从后向前的一个branch, 一个branch的梳理. 要注意的是, DFS 一般是随机选取一个开始的vertex. 当整个component都被标记后, 如果还有白色的vertex, 我们再选取另外一个白色的 vertex 继续进行DFS.

- **Four Types of edges**

Tree edge 通过edge找到了一个新的node
Forward edge 通过edge找到了 descendant
Backward edge 通过edge找到了ancestor
cross edge 通过edge找到的node既不是ancestor也不是descendant.



- **DFS 常见应用**
  - Find component
    根据DFS的算法，我们可以在每从一个vertex开始的时候, 都把这个component里所有的vertex进行标记. 以此类推.
  - Find cycle
    类比贪吃蛇, 如果我们在DFS的途中, 遇到了一个grey的vertex, 说明图中有cycle.
  - Topological Sorting
- **Topological Sorting**
  Topological Sorting指的是对dependency的整理，我们在DFS的问题上, 经常讨论的是利用edges来表示不同objects之间的dependency, 而这时, 我们就可以通过dfs 来确定dependency关系, 和是否存在cycle. 如果存在cycle则无法找到validated的dependency关系. 否则, 我们DFS的结果就是我们的topological sorting的结果.
- **常见题型**
  - **找Cycle**
  - **Topological Sorting**

Part L. MST

1. Minimum Cost Spanning Tree 定义

首先, 我们要搞清楚什么是spanning tree. Spanning Tree是一个每一个vertex都是至少一个edge的一个endpoint, 也就是说, 在spanning tree里面, 一定不会出现cycle. 再加上minimum cost, 字面意思, 也就是总weight / cost最小的spanning tree. 在MST的证明过程中, 我们经常要使用到MST的基本定义来用contradiction的方式来证明.

注意注意, 在讨论MST的时候, 一定要注意, 对于非 distinct weight 的Graph来讲, 不是只有一个MST的.

2. Prim's Algorithm

算法概述: Prim's Algorithm是我们接触到的第一个来构建MST的算法. Prim的做法很简单,

从一个vertex开始延展一个component. 每次都把距离这个component最近的vertex添加进入component, 并且记录这个edge. 到最后我们会得到的一个MST.

时间复杂度分析: 我们在Prim's里面使用 min-heap structure 来管理记录我们的component 到每个vertex的距离. 那么我们可以确定heap里面有n - 1 个item. Extract Min还有update priority的runtime都是 O(log n), 那么对于每个vertex, 我们最多要执行一次Extract Min, 对于每个edge我们最多都要update一次 priority. 所以, 我们的total cost应该是 O((n + m) log n) 也就是 O(m log n).

3. Disjoint Set

在Kruskal's Algorithm之中, 最重要的环节也就是Disjoint Set. Disjoint Set是专门用来计算components的. 在这里有两个非常重要的概念, union by rank 和 path compression. 通过这两个概念, 我们的disjoint set的runtime可以优化到O(α(n)). 这里要注意, α(n) 是一个在CLRS中定义的function, 我们可以简单地记为是constant的.

- Make Set - O(1)
  Make set就是很简单的把一个vertex, 建立成一个Disjoint Set里的一个新的set, 它自己就是自己的representative.

- Find set & Path Compression - O(α(n))
  Find set就是去找这个vertex的representative, 在找的过程中, 如果发现这个vertex不是representative的直接children, 那就compress path, 把它变成直接children.

- Union by Rank - O(α(n))
  首先我们对要union的两个candidate做find set, 找到他们的representative. 如果representative不同, 那么我们根据representative的rank来进行Union, rank较大的representative作为新的representative, 另外一个作为他的直接children.

Disjoint Set是专门用来处理components的问题的, 一旦遇到要区分, 寻找 components一类的问题, 我们都可以尝试使用disjoint set来进行解决.

4. Kruskal's Algorithm

算法概述: Kruskal's Algorithm在构建MST的时候使用到了Disjoint Set来管理K算法进行中的components. Kruskal不再是从单一components grow成完整的MST. 我们首先将所有的edges sort一下, 把所有vertices 都make set一下. 然后从最小的edge开始一直union by rank. 所有真的发生了union的 edge 是我们要记录的edge. 这样到最后, 我们也可以找到一个MST..

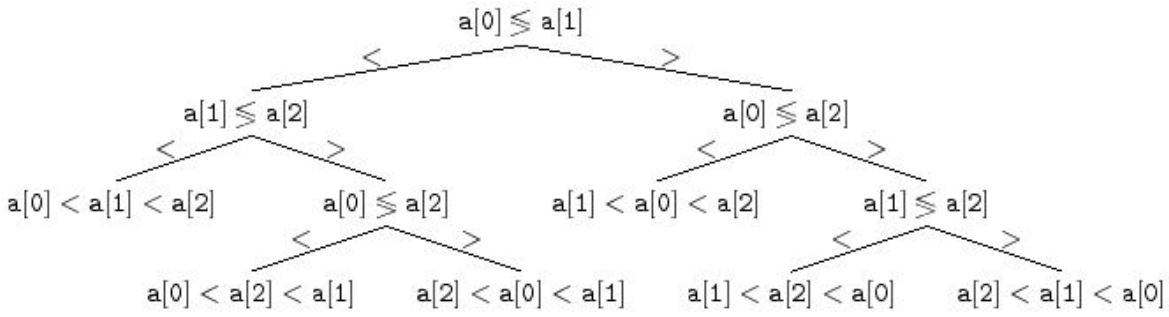
时间复杂度分析: 首先我们需要 O(m log m) 的时间来把所有的edge sort好, make set一共花了 O(n) 的时间. 然后我们对所有的edges从小到大, 对edge的两个端点u, v进行union by

rank. 一共是O(m * α(n)) 时间. 所以总共我们的时间花费是O(m log m + n + m * α(n)) 也就是O(m log n)

Part M. Decision Tree

Decision Tree 是 CSC263 中学习的最后一个Topic, 他的作用非常单一, 通过把问题变成一个一个 decision making 的branch, 来构造一个树状的 Decision Graph, 也就是Decision Tree. 通过Decision Tree 我们可以了解到, 最少（**Lower Bound**） 我们需要多少步才可以完成我们的算法.

- **Comparison Based Sorting Algorithm**



对于comparison based sorting algorithm, 我们可以得到类似于上图的一个comparison tree. 我们一定要对每个item的位置都make decision才行, 根据图上的二分法,我们可以确定我们一定要从root (a random list) 一直走到leaf (one possible output) 才能得到我们需要的结论, 一个sorted list. 因此, 我们可以得到一个conclusion, decision tree的height是我们在worst case下要经过的decision数量. 在这里我们可以简单记忆, 如果有 O(n!) 个leaf存在, height为 O(nlog n). 换言之, Comparison Based Sorting Algorithm的worst case lower bound 是 O(nlog n).

- **Adversary Argument**

我们在试图证明lower bound的时候, 经常使用的就是adversary argument. Adversary 的意思就是对手, 简单说就是给我们的算法找一个对手. 我们的算法小G会尝试以最快的速度, 完成一个问题. 每走一步小G都想完成这个任务. 但是小G的死对头, 小P就会一直质疑小G当前的solution. 每次质疑, 小P都会找到一个可以击破小G完美solution的方法. 小G就会被小P push, 因此多走一步. 我们一直循环这个过程, 终于有一天, 小G 到达了一个小P 无法反驳的state, 我们说目前走过的总步数, 也就是能满足solution的最低步数.

Formally..
To prove a lower bound L(n) on the complexity of problem P, we show that for every algorithm A and arbitrary input size n, there exists some input of size n (picked by an imaginary adversary) for which A takes at least L(n) steps.

Proof steps

1. Let A be an arbitrary algorithm.
2. Construct an input that makes the arbitrary algorithm runs in <lower bound> number of operations. Must show that less than <lower bound> number of operations cannot satisfy the problem.
3. Since A is arbitrary, we know we at least need <lower bound> number of operations to solve the problem.