CSC263 Cheat Sheet

Lecture 1 Complexity Review

Complexity: Amount of resource required by an algorithm, measured as a function of the input size.
Average running time: expectation of the running time which is distributed between best and worst case

Let tn be a random variable, **E(tn) = Σt·P(tn = t),** to know Pr(tn = t), we need to be **given the probability distribution of the inputs**.

***(slowest) 1＜log n＜√n＜n＜nlogn＜n^2＜n^3＜2^n＜n^n (fastest)**
注意：让你 analysis best/worst case 的时候一定要看清 loop 的次数！best case 是运行到什么时候 program 会 terminate！

Example: Search42(), for each key in the linked list, we pick an integer between 1 and 100 (inclusive), uniformly at random.
P(tn = 1) = 0.01 (when head is 42)
P(tn = 2) = 0.99*0.01 (head is not 42, the second one is)
P(tn = 3) = 0.99*0.99*0.01
P(tn = n) = (0.99)^(n-1)*0.01
P(tn = n+1) = (0.99)^n(when none of n keys is 42)
Σt·P(tn = t) = Σt*(0.99)^(t-1)*0.01 + (n+1)(0.99)^n
Probability of worst case 一定要算上前 n 次都没找到的概率和最后一次找到了的概率（如果 sample space 里一定找得到 target 的话）
Example：input of FIND_LARRY() is uniformly distributed at random. The average case analysis of the runtime is
E[X] = \sum_{i=1}^(n-1) i*1/(n-1) = 1/(n-1) * \sum_{i=1}^(n-1) i = n/2

$\sum_{k=1} k^2 = \frac{n(n+1)(2n+1)}{6}$
$\sum_{k=1} k^3 = \frac{n\text{^}2(n+1)\text{^}2}{4}$
$\sum_{k=1} k = \frac{n(n+1)}{2}$((前项+末项)✖末项)/2
$\sum^{n}_{k=1} r^k = \frac{r(1-r\text{^}n)}{1-r}$
$\sum^{k}_{i=0} n^i = \frac{n^{k+1}-1}{n-1}$
$\log_b(mn) = \log_b(m) + \log_b(n)$
$\log_b(^m/_n) = \log_b(m) - \log_b(n)$
$\log_b(m^n) = n \cdot \log_b(m)$
$a^{\log b(c)} = c^{\log b(a)}$ $\log_b(1) = 0$

Lecture 2 Heap and Priority Queue

Heap: max-heap is a **nearly-complete** binary tree that every node has a key (priority) **greater than or equal to keys of its immediate children.** (every subtree of a heap is also a heap)

Insert(Q,x) Append 到 array 末尾，再 **bubble up/down**	Max(Q) return array[1]	ExtractMax(Q) 用最后一个 elm 去替换 root，再 **bubble up/down（跟较大的孩子交换）**	IncreasePriority (Q, x, k) 改变了值以后 **bubble up/down**
Θ(log n)	Θ(1)	Θ(log n)	Θ(log n)

Store a heap in an array (**index starts from 1**):

Left(i) = 2i	Right(i) = 2i + 1	Parent(i) = floor(i/2)

如果是 ternary heap and index starts at 1:

Left(i) = 3i	Middle = 3i	Right(i) = 3i + 1	Parent(i) = floor((i+1)/3)

Heap Sort：each ExtractMax is O(log n), we do it n times, so overall it's O(n logn)

Sort without using extra space (used for heap-ordered array): 先记录 heap 的 size，array 首元素和末元素交换，现在在末尾的就是最大元素了，heap size 减一(相当于从 heap 里删除了)，前面的部分 restore heap property （Θ(log n)），再次首位交换，repeat。
BuildMaxHeap(A)：Converts an array into a max-heap ordered array, in O(n) time. 一个 heap，从下往上 fix heap property，最下面一层（占一半元素）都不用考虑，从 **floor(n/2)**那个元素开始(就算是最高的一层 root 也只需要 logn 次 bubble down)。

```
HeapSort(A) ：
for i ← A.size downto 2:
    swap A[1] and A[i]
      A.    size ← A.size – 1;
    A.BubbleDown(A, 1)
```

```
BuildMaxHeap(A):
for i ← floor(n/2) downto 1: BubbleDown(A, i)
```

So, total number of swaps:
T(n) = 1(n/4)+2(n/8)+3(n/16)…

Q: How many leaf nodes are there in a binary heap with 263 nodes? Write down your answer in the space below
A: 163/2 向上取整

Lecture 3 Dictionary and BST

Dict: A set S where each node x has a field x.key
Search(S, k): return x in S, such that, x.key = k
Insert(S, x): insert node x into S (if already exists same key , replace)
Delete(S, x): 如果要删除有两个孩子的 node，用左支最大值或右支最小值替换

	unsorted list	sorted array	BST	Balanced BST （AVL）	Hash table
Search(S,k)	O(n)	O(logn)	O(n)	O(log n)	O(1)
Insert(S, x)	O(n)	O(n)	O(n)	O(log n)	O(1)
Delete(S, x)	O(1)	O(n)	O(n)	O(log n)	O(1)

BST 的 worst case 都是 O(n)，因为他不要求要 balance，所以 tree 的高度 worst case 是 linear 的，要 search 的话就是 O(n)
BST property：For every node x in the tree，左支值比 root 小，右支值比 root 大
Because of BST property, we can say that the keys in a BST are **sorted**， just do an inorder traversal.

找 successor：
if no right child：找到左系祖先（是他父亲的左孩子），如果找到 root 还没有就 return NIL
If right child exists： return TreeMinimum(x.right)

找 pre-successor:
如果左支不存在，找到最近的右系祖先
如果左支存在，找到左支最大值

```
Successor(x):
if x.right ≠ NIL:
return TreeMinimum(x.right)
y ← x.p
while y ≠ NIL and x = y.right: #x is right child x = y
y = y.p # keep going up
return y
```

TreeInsert(root, x)：Insert node x into the BST rooted at root return the new root of the modified tree if exists y, s.t. y.key = x.key, **replace y with x**
TreeDelete(root, x)：如果要删除的结点有两个孩子，找左支最大值或右支最小值去替换

Lecture 4 Dictionary and AVL Tree

The height of an AVL Tree is guaranteed to be O (log n)
Rotation: find the lowest ancestor of the new node who became imbalanced.
Unbalance 的就是 root:
RR case (最外面一支)：single left rotation (1.1)
LL case(最外面一支)：single right rotation (2.1)
Unbalance 的 是 sub tree（double rotation）:
1.(RL) sub tree 如果已经 left heavy: 先 right rotation sub tree 到 right heavy，再从 root 做 left rotation (1.2)
2.(LR) 如果已经 right heavy: 先 left rotation sub tree 到 left heavy，再从 root 做 right rotation (2.2)
Insert O(log n)**:** insert 过后 tree 的 height 是不会变的。所以只用 update 我们 rotate 过的 nodes 的 balance factors O(1)，tree 的其他部分都不用动.
Delete O(log n)：delete 过后 tree 的 height 是会变的（single rotation 的不一定会变，double rotation 的一定会减 1），这时就不止要 update rotate 过的 subtree 的 balance factors，rotate 过的 subtree 的所有 ancestors 都要一起 update，共有 log n 个，所以 update balance factors 是 O(log n)

Augmentation：maintain this additional information efficiently in modifying operations (within O(log n) time.
Ordered set:
Rank(k): return the rank of key k
Select(r): return the key with rank r
1.加个 node.rank 的话，insert new node 以后要把所有 node 的 rank 都 update 一遍，O(n)
2. 用原本的 AVL tree，select 和 rank 都是 O(n)
3.（正确）AVL tree with additional attribute node.size (size of subtree and itself) for each node.
这样 insert 和 delete 都只用 update O(log n) time.

Example:
The bank account, 需要存的 attribute 有 1.自己的 bank amt 2.自己这个 amt 的 dup 个数 3.孩子的所有 dup 个数

Q: why can it be maintained efficiently upon modifications to the AVL-tree？
A: The value of sum only depends on the node's two children and the node itself, therefore it can be maintained efficiently.

```
Rank(S, x):
(如果我们要找的 node 在左支)
 r <- x.left.size + 1
y <- x
while y.p != NIL:
(如果我们要找的 node 在右支，
要加上左边所有 node 的个数)
 if y == y.p.right:
r <- r + y.p.left.size + 1
y <- y.p
return r
```

Strategy for Select(S, r):
Calculate rank p of root
If p = r, stop
If p > r, proceed to the left subtree and search for rank r
If p < r, proceed to the right subtree and search for rank r – p

Theorem 14.1 of CLRS: If the additional information of a node depends only on the information stored in its children and itself then this information can be maintained efficiently during Insert() and Delete() without affecting their O(log n) worst-case runtime.

NOTE:把一样 value 的 node 也添加到 tree 里不可行，因为 rotation 会把这些 node 弄得到处都是，就不好找了

Lecture 5 Dictionary and Hash Table

Direct address table(array): directly using the key as the index of the table(缺点:key 只能是 integer)

Hash function h(k): a functions maps universe of keys to {0, 1, …, m-1}
Chaining: Store a linked list at each bucket, and insert new ones at the head

Chaining O(n) worst case: Search(k), Insert(k), Delete(k)的 runtime 都是 O(n), length of the chain

Chaining O(1) average case search analysis (assume keys and hashing are uniformly at random): So, the probability that key k gets hashed to bucket j is 1/m, T = 1+ the length of the linked list stored at h(k), Let L be a random variable representing the number of keys hashed to h(k) , So, E[T] = 1 + E[L], Each key has a 1/m chance of being hashed to bucket h(k), and there are n keys, E[T] = 1 + E[L] = 1 + (1/m) ∗ n = 1 + n/m where n/m is the load factor.

load factor: average number of keys per bucket
search is Θ(n/m), key 数量小于 slots 数量或 load factor 是个 constant 时， runtime 都是 Θ(1), 但如果 load factor 是√ n runtime 就不是 Θ(1)了，所以 slot 数量一定要够用。若 slots 够用， keys 和 hashing 都是 simple random 的话 runtime 就是 Θ(1)

The division method
1.h(k) = k mod m where h(k) is between 0 and m-1
注意: m better be a prime number 这个方法才好用
2.h(k) = (ak + b) mod m
achieve simple uniform hashing

The multiplication method
h(k) = floor(m * (k*A mod 1))
"Magic" constant: A = (sqrt(5)-1)/2 = 0.618
the magic constant guarantees the result is between 0 and m-1, not sensitive to the value of m

Hash Table Operations (linear probing):
Insert: 用 h(k) 生成他自己的 key，如果被占用了就一直往下加 1 直到找到空位为止然后 insert
Delete:把删掉的元素设为"deleted"，这样 search 的时候就不会因为有个空位就停止寻找然后 return 找不到，然后 insert 时可以 insert 到"deleted"的位置
Search:从 h(k) 生成的元素的 key 开始找，只要没找到就一直往后找，结果要么找到，要么 reach 到了空 slot，就不存在

Open addressing (No chains allowed)
linear probing: (h(k) + i) mod m
缺点: Keys tend to cluster, which causes long runs of probing.
quadratic probing: $(h(k) + c_1 i + c_2 i^2) \bmod m$
缺点：有些 slot 永远不会被用到，并且同样会 cause cluster，因为遇到同样的 key 以后永远都会去同一个地方
double hashing: (h1(k) + i * h2(k)) mod m
用两个 hash functions, two sources of "randomness", each key has its own starting point h1, and its own jump sequence h2

Assuming simple uniform hashing, the average-case number of probes in an unsuccessful search is 1/(1-α).

For a successful search that uses chaining is Θ(1 + α).
For a successful search it is 1/αln(1/(1- α))
In open addressing, we must have α < 1

Provable worst-case O(1) search and O(1) delete, Expected, amortized O(1) insert

Lecture 6 Amortised Analysis and Quicksort

compute the total runtime T(m) of m operations, the amortized cost of these m operations is then computed as T(m)/m.

Dynamic Arrays
Aggregate method: 每个 operation 还是一样的 cost，所以只有在 elm 是第 power of 2 个时， cost 才是 n+1， 其他时候都是 1

Accouting method:
If the array become full, double the size
If the array become ¼ full, shrink the size to half.
Earning $3 is enough for append and delete operation.

Example1:
Consider the following BST T created by inserting a sequence of keys into an initially empty BST. How many different insert sequences are there that would result in exactly the same tree as T?
ANS:
Left subtree 同形数量*right subtree 同形数量*combination
例：一个左支有 2 个 node，右支有 3 个 node 的 BST，共有 6 个 node，root 必须排在前面，所以 5C2*1*1(左支的 5C2 或右支的 2C2 都是一样的，乘左子树同形数量和右子树同形数量)

Aggregate method: total cost divided by total number of operations, gives each type of operation the same amortized cost

Accounting method: give each type of operation a different amortized cost

By dynamically resizing, the hash table can maintain a constant load factor and average-case O(1) operation cost.

Example:
Expansion factor of doubling the size: 2
Expansion factor = 1/(cost-2) + 1
Now change the expansion factor so that the amortized cost per operation becomes tightly upper-bounded by 7. What should the new expansion factor be?
ANS: 1.2, each item has 7-2=5 recharge dollars, so each old item needs 0.2 new items to be recharged.

Quicksort
choosing a pivot element, and partitioning the array so that elements smaller than the pivot are to its left and elements bigger than the pivot are to its right
def partition(array, pivot)
def quicksort(array)

Worst-Case Upper-Bound
1.each element in A can be chosen as pivot at most once
2. every comparison involves an element chosen as a pivot (for a and b to be compared, one of them must be the pivot)
3. any pair (a, b) of elements in A are compared at most once so total number of comparisons is no more than the total number of pairs in A, O (n^2), happened on sorted array (两个相邻的数被比较到的概率是 1)

Worst-Case Lower-Bound
show a type of input that takes cn2 comparisons, make very unbalanced partitions, sorted array, Ω(n^2)
任何 sort 的 worst case 都是 n^2

quicksort 的 average case 是 nlogn（前提就是 array 是完全无序的，randomization）

Consider elements {i, i + 1, . . . , j}
If i or j is the first of these to be chosen as pivot, then i and j get compared, 如果 j 和 i 之前的任何一个数作为 pivot 了，那 i 和 j 永远不会被 compare 到。

Pr[i and j are compared] = 2/(j − i + 1)

如果 user 一直提供 bad arrays 怎么办？我们就在 sort 里再 shuffle 一遍所有 element，就能保证 O(n lg n) expected runtime.

Q: In this question, you will augment AVL trees to implement the following new operation: NumGreater(k): return the number of elements with a key strictly greater than k.
1. additional information will you store at each node of the AVL tree: x.size, the size of the subtree rooted at the node.
2. Explain how to maintain your new information during Insert operations: The insertion can only change the size attribute of the ancestors of the newly inserted node. So the information can be maintained at the same time as the insertion is carried out (including any necessary rotations), using only a constant amount of additional time on each level. Since in an AVL tree the number of ancestors of a node is in O(logn), the maintenance takes O(logn) time, without affecting the running time of Insert.
3.Give a detailed implementation for operation NumGreater:

```
NumGreater(k) :
return TreeSum(root, k)
TreeSum(root, k) : #helper function
if root = nil :
return 0 else if k > root.key : # Values in the left subtree can be ignored: they all have key less than k.
return TreeSum(root.right, k)
else if k < root.key : # Values in the right subtree must be counted: they all have key larger than k.
return TreeSum(root.left, k) + root.right.size + 1
else: return root.right.size
```

Binomial distribution: probability of getting exactly *k* successes in *n* trials (total number of trials is fixed)

Negative Binomial distribution: random variable is the number *X* of repeated trials to produce *r* successes (number of success is fixed)