FEIT 飞天教育 EDU

— 选择飞天，重仓未来 —

期末总复习课程 – COMP2123

by

梨老师

FEIT EDUCATION

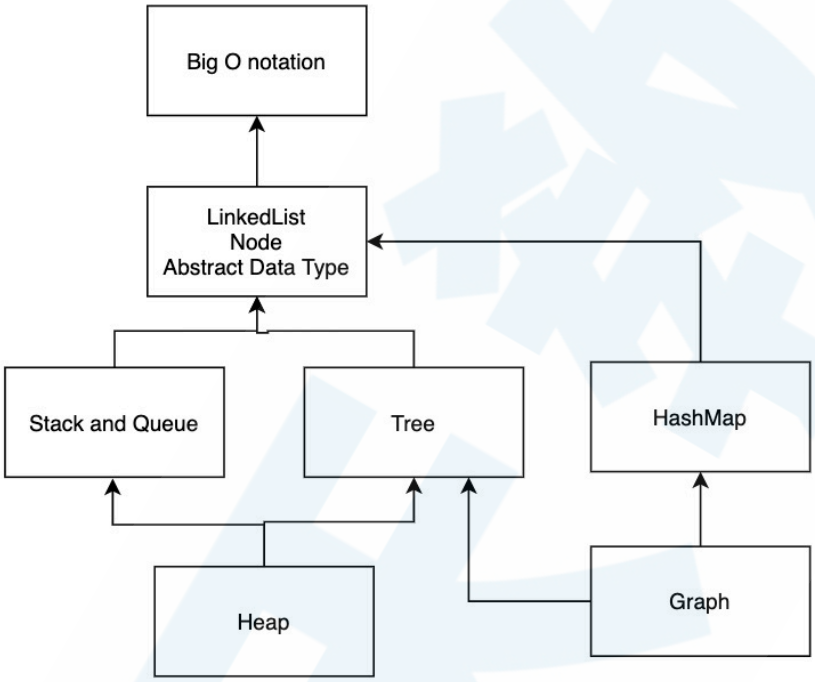
飞天教育

FEIT小助手【USYD】

澳大利亚 悉尼

Table of Contents

知识图谱与考试题型



题型:
前 2 题 (20 分):
简单的描述算法过程(很可能是 Graph 相关)
prove 复杂度 (week 1),
概念区分 (例如 Heap vs BST) ,
比较优劣 (例如 Graph 的存储选择 Matrix 还是 Adjacency List)
prove 一个算法是否是错的 (例如贪心)

后 2 题 (40 分):
Design an algorithm that ….. , (for full mark- time complexity: O(…))
A describe your algorithm
B argue/prove correctnes
C analyse complexity.

最可能会被问的类型: Divide and conquer, Tree (证明格式较有规律), graph.

Cheat sheet

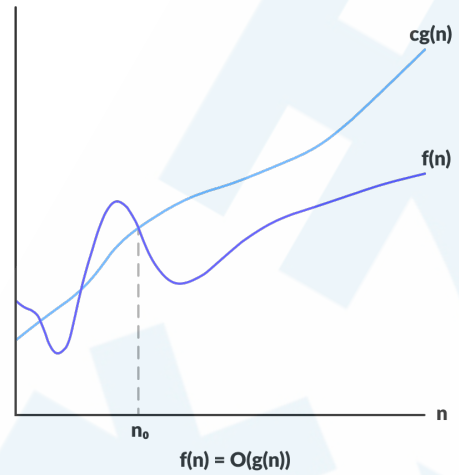
不同数据结构基于不同具体实现对应的复杂度. 通常是表格.
复杂算法的过程(比如 graph 里的最短路算法, 找 MST, 等等).
注意算法基于不同具体实现也有不同的复杂度.

我们课程里在设计一个数据结构的时候, 会先定义:
需要拿它来做什么 – 执行什么任务, 完成什么动作.
在确定动作之后, 确认实现细节, 以及最好需要怎样的时间复杂度.
所以, 在涉及到具体数据结构的操作的时候, 我们会倾向于先记住一些常用的操作, 其次是其复杂度.
对于同一种数据结构, 其实现细节不同, 复杂度也会不同.
比如 List 结构, 无论 ArrayList 的实现或者 LinkedList 的实现, 可以实现相同的"动作": 比如 append, insert, insertHead, removeHead, removeTail, get(index)….etc.
但是, Array based list 的 insertHead 复杂度是 O(N), LinkedList based list 的 insertHead 复杂度是 O(1)
这是由于数据结构的实现方式不同而造成的.

Complexity definition 与证明

可能单独拿出来考的部分: 第一问/第二问作为小题来提问.

定义



例题

Problem 5. Given an array A consisting of n integers, we want to compute the upper triangle matrix C where

$$C[i][j] = \frac{A[i] + A[i+1] + \cdots + A[j]}{j - i + 1}$$

for $0 \leq i \leq j < n$. Consider the following algorithm for computing C:

```
1  def summing_up(A)
2    C ← new matrix of len(A) by len(A)
3    for i in [0:n-1]
4      for j in [i:n-1]
5        compute average of entries A[i:j]
6        store result in C[i, j]
7    return C
```

a) Using the O-notation, upperbound the running time of SUMMING-UP.

b) Using the Ω-notation, lowerbound the running time of SUMMING-UP.

技巧:

大 O 的计算是 round up 的, Omega 的计算是由部分到整体的.

a) 计算 O-notation:

外层 i, j 循环的总次数:

$$(n-1) + (n-2) + \ldots + 1 = \sum_{i=1}^{n-1} i$$

By arithmetic progression,

$$\sum_{i=1}^{n-1} i = \frac{(n-1)+1}{2}(n-1) = \frac{1}{2}n(n-1) = \frac{1}{2}(n^2 - n)$$

内部 compute C[i][j]的次数: 次数等于 i – j + 1 , O(n)

b) 计算 Omega-Notation

我们通过展示函数的一部分已占用 f(n)的 complexity 来证明.

在这个例子里, 观察 i < n/4, j > 3n/4 部分. 这样的这样的组合一共有$3n^2/16$组. 每组内部 compute C[i][j] 的复杂度至少是 n/2. 所以这一部分的复杂度至少是$3n^3/32$.

因为运算的一部分复杂度至少是 $3n^3/32$了, 所以整个函数 Omega(n^3)

为什么选 n/4? 这个是随机的.

通用证明技巧

证明正确的方法

1. Prove by contradiction

2. Prove by math induction – 适用于有递归的算法
    a. 例题: assignment 2, divide and conquer
3. Prove loop invariant – 适用于非递归, 有 loop 的算法, 比如 assignment 1
    a. Assignment 3 的第二问, Merge 步骤的正确性证明
4. 2, 3 结合: 适用于 Divide and conquer.
    a. 例题: merge sort 正确性证明.
5. Exchange argument - 仅适用于 Greedy.
    a. 例题: assignment 5 第一题 b 问.

证明错误的方法

1. 这类题目会对某个算法问题假定一个解法. 要证明此解法不正确, 我们只需要举反例即可.

例如 (Assignmnet):
(0-1 背包问题)
我们有一些物品, 每个物品占的空间是S_i, 价值是F_i
我们试图在容量为 S 的背包里放尽可能最大化总F值的物品, 请问是否可以直接按照 F 值倒序来选取 (PickLargest strategy)

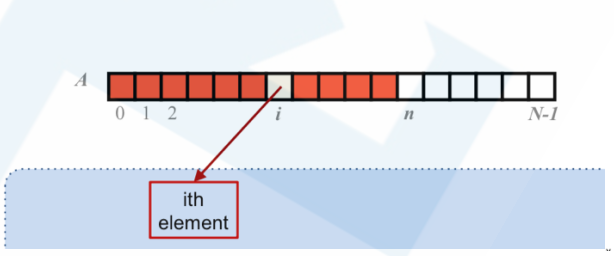
在举反例的时候要注意描述完整的 input 以及最优解是什么, 以证明题目给出的算法并非最优.
S = 10

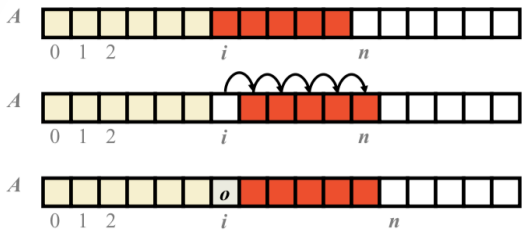
Item	1	2	3	4	5
Space	1	2	3	4	5
Fun	3	4	5	6	7

按照 Fun 最大值来选, {5, 4, 1} 的 FUN 值加起来是 7+6+3 = 16
但是显然, 我们可以找到一组更好的解:
{1,2,3,4} 的 FUN 值加起来是 3+4+5+6 = 18.
所以 $PickLargest$ 并不能给我们正确的最优解. 这样就证明完毕了.

Node, Linkedlist & Array List

Array Based List





例题

如何实现一个支持 O(1) enqueue, O(1) dequeue, O(1) getAverage 的 array based queue? (asm1)

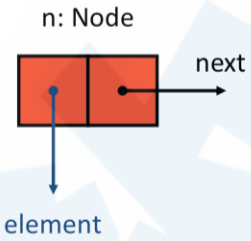
在 class 定义里面加一个 sum 和 length.
每当 enqueue 一个元素, sum += e, length += 1
每当 dequeue 一个元素, sum -= e, length -= 1

LinkedList Based List

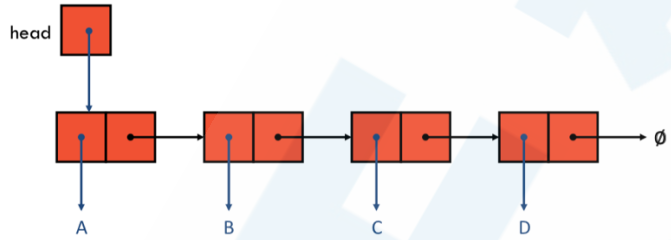
由更小的单元: Node 组成.
Node 的数据结构长这样:

```
class Node:
  def __init__(self, e):
    self.val = e
    self.next = None

  def setNext(self, node):
    # node: is another Node instance
    self.next = node
```



Singly/Doubly LinkedList



在头部加一个新 node
Traverse list
删除某个 index 的 list.

```
class Node:
  def __init__(self, v):
    self.val = v
    self.next = None

class MyLinkedList:
  def __init__(self):
    self.head = Node(-1)
    self.length = 0

  def addAtHead(self, val: int) -> None:
    newFirst = Node(val)
    oldFirst = self.head.next
    self.head.next = newFirst
```

```
        newFirst.next = oldFirst
        self.length+=1

    def deleteAtIndex(self, index: int) -> None:
        if index >= self.length or index < 0:
            return -1
        curr = self.head
        while(curr.next):
            if index==0:
                break
            curr = curr.next
            index-=1
        curr.next = curr.next.next
        self.length-=1
```

时间复杂度总结

ArrayList 和 LinkedList 的空间复杂度都是 O(n)
ArrayList 适合快速访问某个 index, 删除或增加在任何位置都是 O(n)
LinkedList 适合 list 只修改头 or 尾的情况
可以基于这两者来设计适合自己的数据结构. 常用的技巧是:
给基础的 List 类加 pointer
给类加"记忆"
我们一般用的 Stack 和 Queue 是基于 Linkedlist 的复杂度

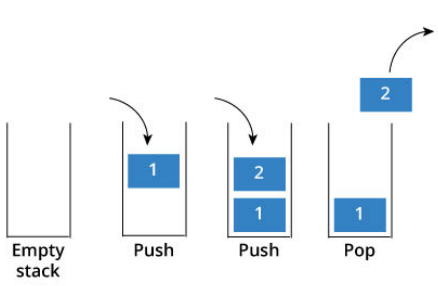
	ArrayList	LinkedList
Size()	O(1)	O(1)
isEmpty()	O(1)	O(1)
Get(index)	O(1)	O(n)
Set(index, e)	O(1)	O(n)
Add(index, e)	O(n)	O(n)
Remove(index)	O(n)	O(n)
Remove(Ref)	?	O(1)
AddFront(e)	O(n)	O(1)
InsertBefore(Ref, e)	?	O(1)
InsertAfter(Ref, e)	?	O(1)

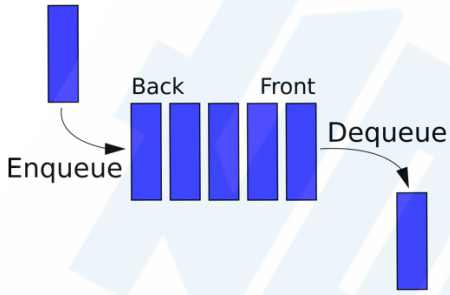
例题

1. 用 Singly Linked List 实现的 Queue, 其 push 和 pop 的复杂度是多少?
   O(1)
   因为 Singly LinkedList 不擅长删除尾部元素 (O(n)), 但如果在尾部保持一个 pointer, 就可以实现 O(1) append.
   头部 enqueue, 尾部 dequeue, 即可两个操作都是 O(1).

Stack & Queue

记忆点: Stack 先进后出 (FILO, first in last out) , Queue 先进先出 (FIFO, first in first out)





Stack 和 Queue 在 Python 里都可以由 list 的操作简单实现.

在 Java 里是两种不同的数据结构, 但也可以通过 List 来简单实现.

时间复杂度总结

记忆点: Stack 和 Queue 的所有操作都是 O(1) – 基于 LinkedList 的实现, 或者基于不限长度的 array + pointer.

	Stack	Queue
Size()	O(1)	O(1)
isEmpty()	O(1)	O(1)
Push()/Enqueue()	O(1)	O(1)
Pop()/Dequeue()	O(1)	O(1)

例题

1. 用 stack 实现 queue

Using only two stacks, provide an implementation of a queue.

Analyze the time complexity of enqueue and dequeue operations.

用两个 stack(只允许 push/pop)来实现一个 queue 的操作(enqueue/dequeue)

我们使用一个 instack 和一个 outstack 来实现.

当 enqueue 的时候, 我们往 instack 里 push 一个数字; 当 dequeue 的时候, 我们首先检查 outstack 里是否有值. 如果没有, 我们则把 instack 里的所有东西 pop 进 outstack 里. 然后我们让 outstack pop 出去一个数.

举个例子:

operation	Instack	outstack	Return
Enqueue 1	[1]		
Enqueue 2	[1,2]		
Queue.pop()	[]	[2,1]	[2,1].pop() = 1.

复杂度:

Enqueue 是 O(1), Dequeue 是 O(n).

2. 用 queue 实现 stack

用最多两个 queue(只允许 enqueue/dequeue)来实现一个 stack 的操作(push/pop)

解答: 只需要一个 queue.
Push:

q.enqueue(x)
假如现在 q 的大小是 N, dequeue N 次, 并 enqueue 给自己.

pop:

q.dequeue(x)

operation	Q	Return
Push 1	[1]	
Push 2	[2, 1]	
Push 3	[3, 2, 1]	
Pop	[2, 1]	3

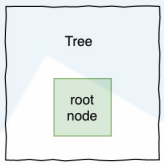
(Binary) Tree

Binary Tree 与 LinkedList 的联系:
LinkedList 是一个 Node 只有一个 reference 去另一个 Node, 只有一个方向. Doubly Linkedlist 只不过可以让你从那个方向返回.
Tree 是一个 Node 可以有 1~N 个(如果最多 2 个, 则为二叉树) reference 去其他 Node, 有很多方向.
一般而言, 关于 Tree 的 worst case, 都可以参考 LinkedList.

数据结构

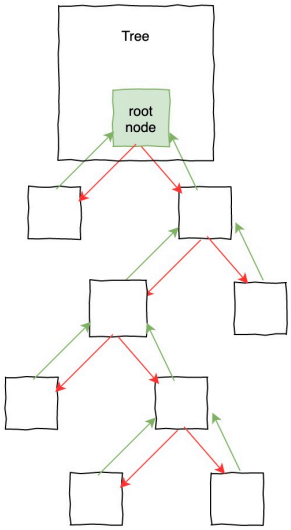
```
class Node:
  def __init__(self, v):
    self.val = v
    self.left = None
        self.right = None

class Tree:
  def __init__(self, root):
    self.root = root
```



Binary Tree 一般只有左右两个 reference, 没有 parent.
算法严格意义上的 Tree 没有 parent.

如果没有 parent, 每次搜索必须来自 root 向下.
如果有 parent, 则表示可以从 leaf node 往上走.
Tree 的搜索, 增删改查, 一般全部是从 root 开始. 先找到想要操作的 node, 然后进行修改.



Tree Traversal

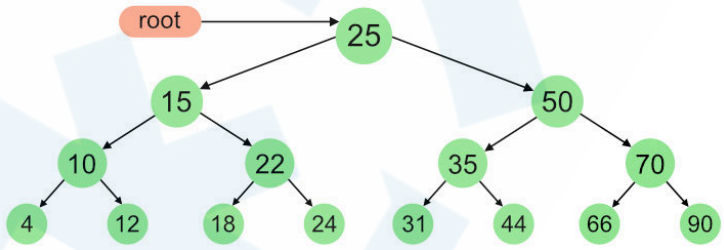
树的题目基本都涉及递归, 最基本的递归是几种 traversal 方式.
最常用的递归从 traversal 的角度上来看是 Postorder.
一般对于某个 Node 的修改即可能涉及到它 subtree 的改动.
断开一个 Node 和 parent 的链接即代表断开它所有 child node 和 parent 的链接.

三种非常重要的树递归遍历算法 – preorder, inorder, postorder.

InOrder(root) visits nodes in the following order:
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:
25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



三种遍历复杂度都是 O(N). 每个 node 只访问一次.

Preorder

先自己, 再左子树递归 再右子树递归
典型用途: 文档查阅.

```
def preorder(node):
  if node == None:
    return
  print(node.val)
  preorder(node.left)
  preorder(node.right)
```

例题

Invert Binary Tree 翻转二叉树

Example:

Input:

```
     4
   /   \
  2     7
 / \   / \
1   3 6   9
```

Output:

```
     4
   /   \
  7     2
 / \   / \
9   6 3   1
```

```
def invertTree(self, root: TreeNode) -> TreeNode:
    def preorder(root):
      if root==None:
        return
      root.left, root.right = root.right, root.left
      preorder(root.left)
      preorder(root.right)
      return
    preorder(root)
    return root
```

正确性: 在每一次 recursion call 之前, parent 都已经被正确翻转, 直到翻转到 leaf 为止.

复杂度: O(n).

Inorder

先左子树递归, 再自己,再右子树递归.

注意: 这个算法只存在于二叉树里.

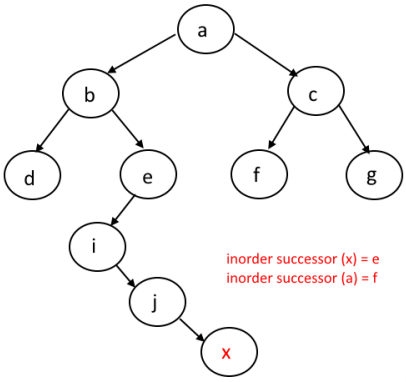
典型用途: 依次访问 binary search tree, 得到排好序的 list

```
def inorder(node):
  if node == None:
    return
```

```
  inorder(node.left)
  print(node.val)
  inorder(node.right)
```

例题

2. 在一个 tree 里, 如何找一个 node 的 inorder successor



如果这个 node 有右子树, successor 是右子树最左的点.
如果这个 node 没有右子树, 循环检查其 parent. 如果可以找到某个点是它 parent 的左子树, 这个 parent 就是我们要找的.

3. 如何按次序输出一棵树的第 N 层?

算法:

```
def inorder(node, level):
  if node == None:
    return
  inorder(node.left, level +1)
  if  level == N:
        print(node.val)
  inorder(node.right, level + 1)
```

正确性: Inorder 的输出次序在每一层的顺序上是从左到右.
复杂度: O(n)

PostOrder

先左递归, 再右递归, 最后自己
通常用于很多我们会写到的算法, 包括这次 assignment

```
def postorder(node):
  if node == None:
    return
  postorder(node.left)
  postorder(node.right)
  print(node.val)
```

PreOrder, Inorder, PostOrder 复杂度都是 O(N).
对于一个算法, 如果树里所有的 node 都要看一遍: O(N)
如果递归的时候, 每次只选择一个 child node: O(h), where h = height of the tree.
在这门课程里, 如果没有说明 binary tree 是 balanced, 那 O(h) = O(n).
如果 balanced 则为 O(logN)

例题

1. Design a linear time algorithm that given a (binary) tree *T* computes for every node *u* in *T* the size of the subtree rooted at *u*

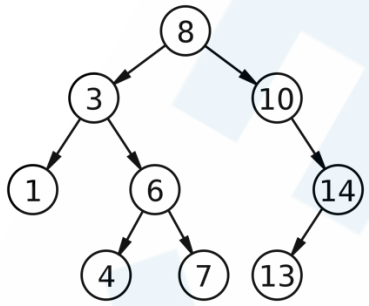
算法: post-order.
Recursively find out the left/right child subtree size.
Return Func(r.left)+Func(r.right)+1.

正确性: for any internal node, the total subtree size equals to sum of subtree rooted at its left and right child.

Complexity: O(n)

Binary Search Tree

(一般假设 BST 里没有重复)
Binary search tree 定义: 在每个 node 上,左子树的值会自己的值小, 而自己的值又比右子树小.



BST search

```
def searchBST(node, val):
    if node == None:
        return None
    if val < node.val:
        return searchBST(node.left, val)
    elif val > root.val:
        return searchBST(node.right, val)
    else:
```

```
    return node
```

AVL 树

- O(h) = O(logN)

一种均衡的 BST. 每一个 node 它的 subtree 的深度差 (从任何一条路到达 leaf 的深度)不会超过 1.

Search: logN
insertion: logN
Removal: logN.

如果插入 30-20-10, 有几种可能性? 但只有 20-30-10 的顺序或者 20-10-30 的顺序才是最好的.

AVL tree 可以通过 rotation 的方式来减少高度

时间复杂度总结

在 binary search tree 里搜索一个值的时间是 O(h) h = height of tree.

如果 tree 是 balanced (AVL), 则为 O(logN).

例题

1. Given a tree node root, validate if it is a binary search tree.

以下这个方法对吗? 如果不对, 是否可以证明?

```
def test-bst(T)
  for u in T do
    if u.left ≠nil and u.key < u.left.key then
      return False
    if u.right ≠nil and u.key > u.right.key then
      return False
  return True
```

解答:

```
def Isvalid(root, min, max):
  if not root: return True
  if root.val > max or root.val < min: return False
  return isValid(root.left, min, root.val) and isValid(root.right, root.val, max)

Isvalid(root, -inf, +inf)
```

2. Consider the following operation on a binary search tree: median() that returns the median element of the tree (one whose ranking is $\lfloor n/2 \rfloor$ in sorted order).

Give an implementation that runs in $O(h)$, where h is the height of the tree. You are allowed to augment the insertion and delete routines to keep additional data at each node.

假设我们可以让 node 储存其子树的 size, 是否可以在 O(h)时间内找到一个 BST 里的 median?

只要可以存储子树的 size, 我们可以很方便的找到一个 BST 中的第 k 个大的 element.
而 median 只是一个特殊的 k 值.

在检查某个 node x 的时候:

1. 如果 x.left.size == k – 1, 说明 x 就是第 k 个 element
2. 如果 x.left.size >= k, node = node.left
   因为 x 至少比 k 个数大了.
3. 如果 x.left.size < k – 1, 说明第 k 个数一定在右子树中.
   此时我们要更新 k = k – u.left.size – 1, 因为缩小范围后之前排除的值要相加.

按照这个过程, 可以在 O(h)时间内找到一个 BST 里的 median.

Priority Queue & Heap

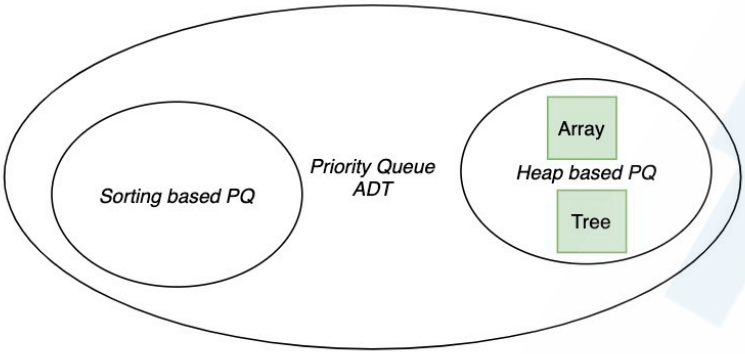
1. Heap 与 priority Queue 的关系
2. Heap 的特性: Complete Binary Tree, min/max heap, removeMin(max), insert 的顺序操作
3. Heap/PQ 的使用

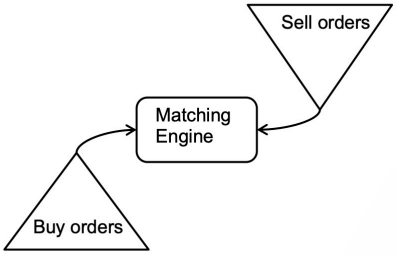
Priority Queue

一种数据结构, 支持 insert, removeMin(max), min()等等.
在考试的时候如何写 PQ 的 deskcheck:

Method	Return value	Priority queue
insert(5,A)		{(5,A)}
insert(9,C)		{(5,A),(9,C)}
insert(3,B)		{(3,B),(5,A),(9,C)}
min()	(3,B)	{(3,B),(5,A),(9,C)}
remove_min()	(3,B)	{(5,A),(9,C)}
insert(7,D)		{(5,A),(7,D),(9,C)}
remove_min()	(5,A)	{(7,D),(9,C)}
remove_min()	(7,D)	{(9,C)}
remove_min()	(9,C)	{}
is_empty()	true	{}

注意: PQ 是抽象的, 不是某种具体的实现. PQ 的实现方法有排序(可选 Insertion/selection sort), heap. Heap 又分 array based heap 和 tree based heap.





PQ 的常用场景:

1. 在数据流中找中位数 – stock matching engine 的例子

基于 sorting 与 list 的 PQ

如何构造 PQ 我们有两种选择:

1. 一个是直接 insert, removeMin 时候排序.
2. 在 insert 的时候排序, 就可以直接 removeMin.

这两者的时间都差不多: insert O(1), removeMin O(n) | insert O(n), removeMin O(1)

Insertion sort 与 Selection sort

为什么刚才两种方法可以帮助 sorting? 因为排序做的事情就是重复 removeMin 和 insert 的步骤. 由刚刚两种方法延伸出两种排序算法:

1. Selection sort 的原理是直接 insert, removeMin 时候排序. 为什么叫 selection sort? 因为它只有在 select(remove)的时候才排序.
2. Insertion sort 的原理是 在 insert 的时候排序, 就可以直接 removeMin. 为什么叫 Insertion sort? 因为它只有在 insert 的时候才排序.

两种 sort 的复杂度都是 $O(n^2)$ – 因为一共需要排 n 个数字, 每次会用到一个 O(n)的方法, 所以总的复杂度是 $O(n^2)$.

基于 Heap 的 PQ

Tree based heap

1. 完全二叉树

除了最底下一排之外, 其余位置全满. 每次插入位置是最底下一排的最右边.

完全二叉树的高度是 log(N) . N 是 node 的个数.

(证明方法: 每一层 i 都有 2^i 个 node, 从 i = 0 加到 i = x, 一共有 N 个 node, 求解 x)

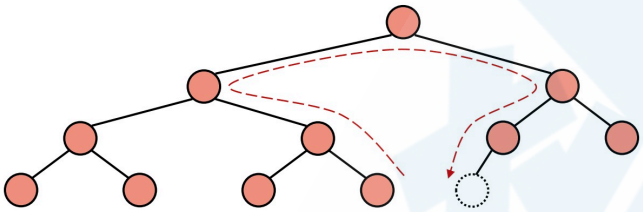
注意区分这里的复杂度和一般的 tree 里的算法复杂度

2. Parent 比左右孩子小. 左右孩子之间无排序.
3. 用 heap 进行 insert()和 removeMin()都是 **LogN.** 一般来说, 如果需要使用 PQ, 用 heap 做是最好的.
4. 基于 heap 的 sorting 叫 heapsort, 时间是 NlogN.

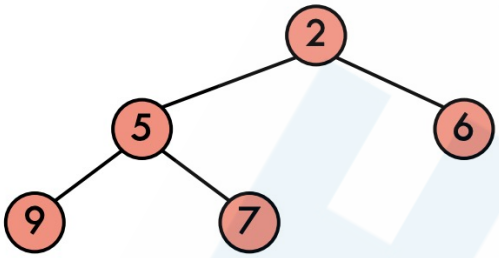
Heap 中的 Insert/Remove

Insert 分为两步:

1. 先放找到最后一位 (最底层的最右)的位置



2. 在这个位置创建新 node, 然后与 parent 交换上浮.



时间复杂度总结

时间: O(logN)

1. 找最后一位的时间是 O(logN).
2. 因为只会与 parent path 上的 node 交换, parent path 最长只有 log(N)这么长.

Remove:

1. 把最后一位(最底层的最右) 的值给 root
2. 然后让这个假的 root 与左右孩子里面比较小的那一个交换下沉.

时间: O(logN)

1. 同样只会与 parent path 上的 node 交换, parent path 最长只有 log(N)这么长.

Array based heap

时间&空间复杂度同上.

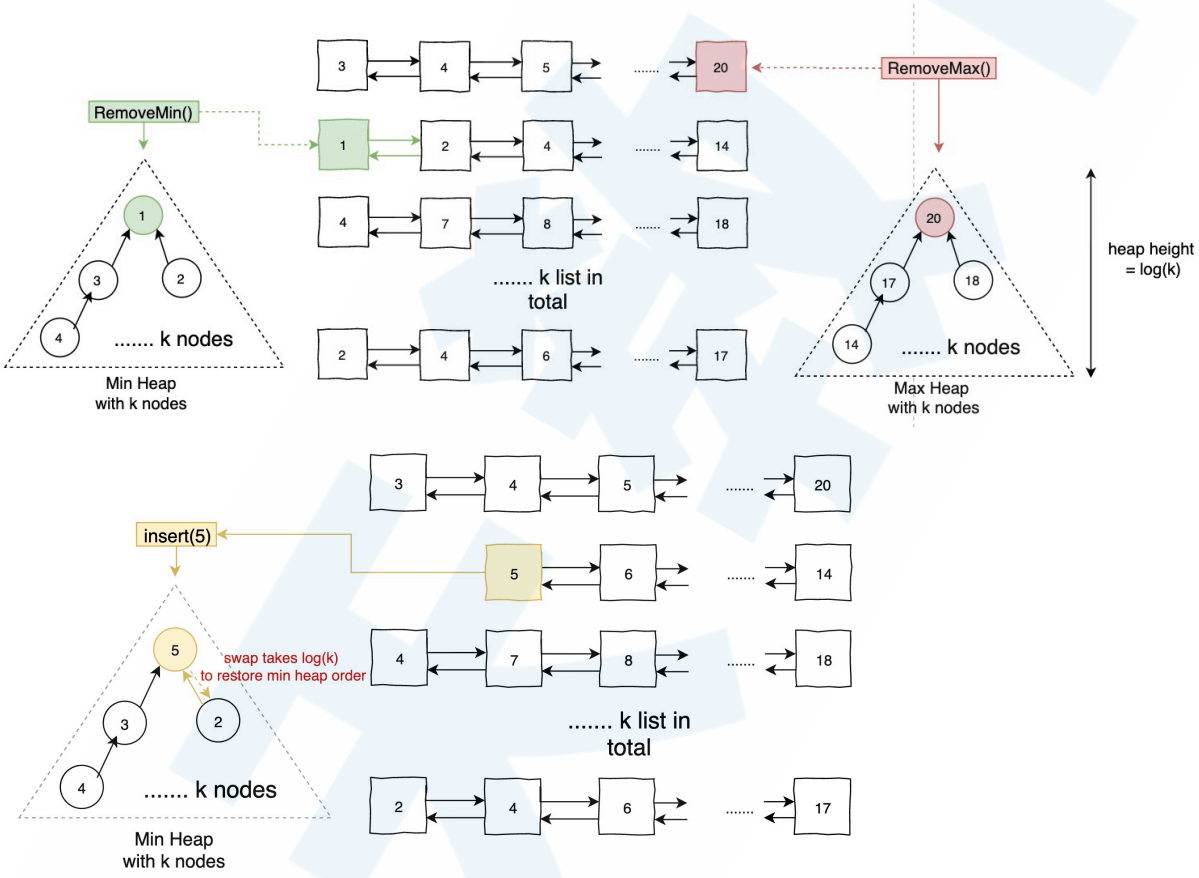
Root 的位置是 array[0]

Array[i]的左孩子在 2i+1, 右孩子在 2i+2 位置. Parent 在(i-1)//2 的位置.

例题

1. Given k sorted lists of length m, design an algorithm that merges the list into a single sorted lists in $O(mk \log k)$ time

一共有 m*k 个元素.
[1,2,4……]
[3,6,7,….]
[2,5,9,….]
目的: 把这些 list merge 成一个 sorted list.



a) 我们分别创造一个 max heap 和一个 min heap.
Minheap 里存的是每一个 list 的 head 的 reference, 也就是 list 里最小的那个数.
Maxheap 里存的是每个 list 里 tail 的 reference, 也就是 list 里最大的那个数.
在 removeMin()的时候, 我们从 min heap 里 removeMin() 获取这些 head 里最小的那个. 并且把 head 之后的那个数字放进 minHeap 里.
在 removeMax()的时候, 我们从 max heap 里 removeMax() 获取这些 tail 里最大的那个. 并且把 tail 之前的那个数字放进 maxHeap 里.

b) 我们的 input list 是从小到大排好序的.

所以, 在任何一个时间点, 一个 list 的 head 都是这个 list 里最小的数字, 而 tail 是最大的数字.
我们想要做的是从 k 个这样的 list 里的找到最小的元素, 意味着只需要从头部里取最小, 从尾部里取最大即可.
以取最小为例, Min heap 存着所有 head 的 reference, 而 Min Heap 的结构可以保证它在$logN$ 时间里取走是最小的元素 (N是 heap 里总元素的数量, 在我们的题里, N=k, 因为有 k 个 list, 就有 k 个头.).
在取走最小 head 之后, 我们要更新这个 list 的头部 reference, 也就是需要 insert 一个新的值进 Min Heap. Insert(e) 在 Heap 里也是 logN.
在 max heap 找最大尾部的情况下也是同理.

c) removeMin/Max() 和 insert() 在 heap based Priority Queue 里的时间都是 $O(log(N))$, N is 是 queue 里的总元素数量.
Min/Max heap 结构只需要保存k 个 LinkedList 里的头和尾 reference. 所以分别每个 heap 里的 $N = k$.
在 doubly linkedlist 里去除头尾的时间是 O(1), 所以并不占大头.
Therefore, overall complexity of removeMin/Max is $O(log(k))$

2. Given an array A with n distinct integers, design an O(n log k) time algorithm for finding the kth value in sorted order.

算法:

1. 首先将 array 的前 k 个数字加入 max heap
2. 从第 k+1 个数字开始, 与 max heap 的 root (最大值)比较.
3. 如果 array[i] < root, remove max from heap, 将 array[i]加入 max heap. 反之不要加入.
4. 最后留在 heap 里的就是整个数组的前 k 个值.

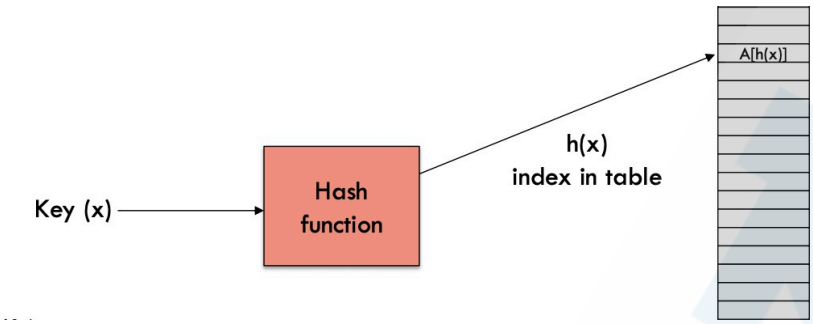
正确性:

Invariant: 在 iterate i 的任何时刻, heap 里仅包含 array 的前 i 个值里最小的 k 个值

1. 当 i = k 时, 符合 invariant.
2. 假设当 i=n 时符合 invariant, 也就是说 root = array[0~i] 最小 k 个值里最大值.
3. 当 i=n+1 时, 如果 array[i] < root, 可见 root 至少比 array[0~i+1]里 k 个值要大了, 不符合" heap 里仅包含 array 的前 i 个值里最小的 k 个值"的要求, 所以被移除. 反之, 可见 array[i] 至少比 array[0~i+1]里 k 个值要大, 所以不加入 heap
4. 所以 I = n+1 更新完之后, heap 里的内容也符合 invariant.

Hashing

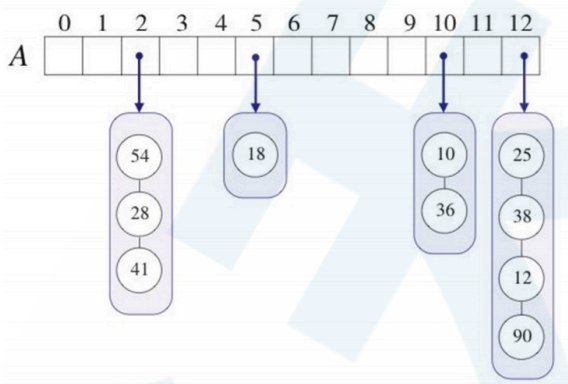
数据结构



有了 hashing, key 可以是任何东西, 比如 string, 比如 object, 都可以.
实际上电脑只能接受数字的 index. 所以需要通过一步: 把 key 转写成一个数字, 才能用 key 来当 index. 这就是 hash function.
Hash function 的处理过程是 O(1), 因为它在有限的步数内完成, 通常是数学计算

Separate Chaining

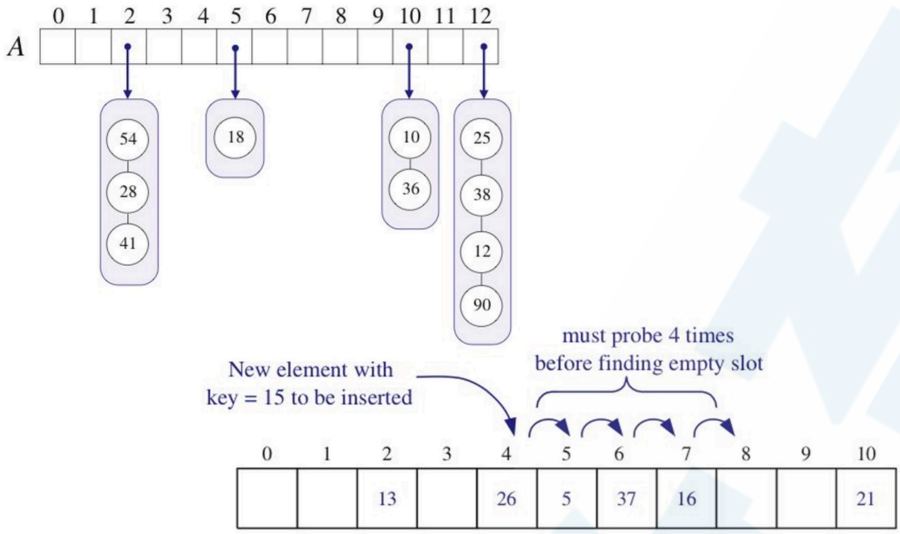
但是实际使用的时候, hash function 可能会产生碰撞.
比如, 为了在一个[0~12]的表里 fit 一些 0~100 之间的数字, 我们可以定义 hash function 为 $n\ MOD\ 12$.
显然, 不同的数字 MOD 12 会得到相同的 index. 这个时候, 我们只能在一个 index 位置用一个 linkedlist 来存放多个值.
这样, 一次搜索就不再是 O(1), 而是 O(n)了, 因为最差情况可能是所有的 input 都得到了同一个 index, 形同于在 linkedlist 里搜索了.



显然我们想要让 hash function 尽可能无序且分散的去产生 index, 减少碰撞.

Linear Probing/Open addressing

如果我们的数据结构有 memory 要求 , 不允许在每一个 index 位置挂一个 linkedlist, 那么遇到 collision 的时候我们还可以顺序往后找位置.



- put 遇到 collision 的话, 往后看有没有空或者 DEFUNCT 的位置.
- Delete 遇到 collision 的时候, 往后寻找要删除的数, 直到遇到空位置. 删除后标记为 DEFUNCT.
- Get 遇到 collision 的话, 往后搜索, 一直搜到空为止. (看到 DEFUNCT 不停下来)

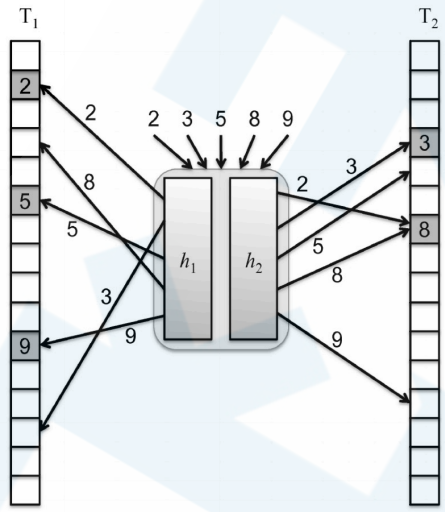
Load Factor

如果我们要把 n 个 key 放进[0~N-1]个位置里去, load factor α= n/N, 代表了通常情况下两个或多个 key 在同一个位置相撞的概率.

Get/insert 的时间通常认为是 O(1+ α), 最坏情况是 O(n).

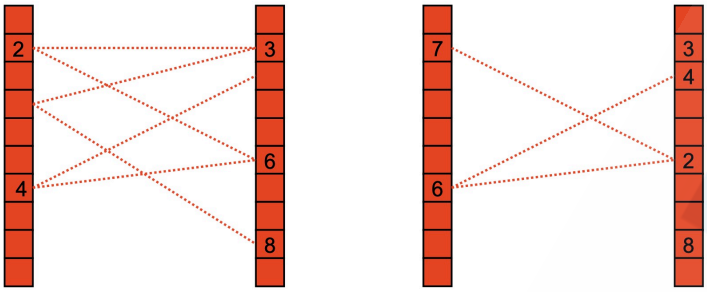
Cuckoo hashing

原理是给配两组 hash function. 每一个位置都会对应两个 hash table 上的不同位置.



Get 和 remove 的时候, 分别用 h1 和 h2 搜索两个 table. 如果 element 存在, 则必然两个 table 里会命中一个.

但是 Put 依旧是 O(n). 如果某个 element 在 put 的时候很不幸两边都 hash collide 了, 它就会直接放进 T1(或 T2)里, 让原先那个位置的 element 去用对面的位置. 这个尝试的过程最坏可以是 O(n)



时间复杂度总结

虽然一般情况下 insert(key, value) 和 get(key, value)是 O(1), 但这门课程里最差情况一般认为是 O(N).

	Put	**Get**	**Remove**
Chaining	O(n)	O(n)	O(n)
Linear Probing	O(n)	O(n)	O(n)
Cuckoo hashing	O(n)	O(1)	O(1)

例题

在关于 HashMap 的题型里, 需要注意必须注明什么是你的 hash function 以及你 hashmap 的实现方式 – 通常可以直接说 linear probing.

1. Suppose that you have a group of *n* people and you would like to know if there are two people that share a birthday. Design an *O*(1) algorithm that given the information about the *n* people's birthday, finds a pair that share a birthday, or reports that no such pair exists.

方法:

使用基于 linear probing 的 hashmap, hashmap 的 memory 大小是 2*365. (这样最多往后移一位)

Hash function: 生日 - > [1, 365]中的一个数.

依次将生日通过 hash function 放入 hashmap, 如果移位 = 1, 直接返回.这个位置

如果全部放入且未返回, 说明没有两人有同一个生日.

正确性:

1 尽管 linear probing, 只需要 2*365 大小的 table, 因为一旦移位等于 1 就说明已经找到了一组 pair, 此时直接返回即可.

复杂度: O(1) 为什么是 O(1)? 因为 365 这个数是固定的. 整个算法运行不会超过 365*2 次.