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**

ExtractMax(Q): 最后一个 elm 和 root 交换，再 **bubble up（跟较大的孩子）**

IncreasePriority (Q, x, k): 改变了值以后 **bubble up/down**

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

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

Store ternary heap and index starts at 1:

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

Sort without using extra space (heap-ordered array only): 记录 heap size，array 首末元素交换，末尾的就是最大元素了，heap size 减一，前面的部分 restore heap property （Θ(log n)），再次首位交换，repeat。

BuildMaxHeap(A)：takes O(n)

for i ← floor(n/2) downto 1: BubbleDown(A, i) 从下往上 fix heap property，leaves (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)。

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

HeapSort(A) ：

for i ← A.size downto 2:

  swap A[1] and A[i]

  A.size ← A.size – 1;

  A.BubbleDown(A, 1)

Q: How many leaf nodes are there in a binary heap with 263 nodes? A: ceiling(163/2)

$\sum_{k=1} k^2 = \frac{n(n+1)(2n+1)}{6}$

$\sum_{k=1} k = \frac{n(n+1)}{2}$((前项+末项)✖末项)/2

$\sum_{k=1}^{n} r^k = \frac{r(1-r^n)}{1-r}$

$\sum_{i=0}^{k} n^i = \frac{n^{k+1}-1}{n-1}$

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): 如果要删除的 node 有两个孩子，用 successor

Because of BST property, we can say that the keys in a BST are **sorted**, just do an **in-order** traversal.

找 successor：

if no right child：找到左系祖先（是他父亲的左孩子），if 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

Lecture 4 Dictionary and AVL Tree (BST)

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 log n 个 ancestors 都要一起 update，所以 update balance factors 是 O(log n)

Ordered set:

Rank(k): return the rank of key k (第几小)

Select(r): return the key with rank r

Implementation: 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.

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.

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.

Number of nodes in an AVL tree: let N(h) be the minimum number of nodes in an AVL tree of height h, N(h) >= N(h-1) + N(h-2) +1 N(0) = 1, N(1) = 2, N(2) = 4, N(3) = 7

Rank(S, x):

r = x.left.size + 1//在他 subtree 里的 rank

y = x

while y.p != NIL://一直找到 root

(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

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

Hash function h(k): a functions maps universe of keys to {0, …, m-1}

load factor: 现有 element 数量/slot 数量

search: 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 (m better be a prime number)

2.h(k) = (ak + b) mod m

The multiplication method

h(k) = floor(m * (k*A mod 1))

“Magic” constant: A = (sqrt(5)-1)/2 = 0.618

Hash Table Operations (linear probing):

Delete:把删掉的元素设为”deleted”

Search:从 h(k) 生成的 key 开始找，要么找到了，要么 reach 到了空 slot 就不存在

Open addressing (No chains allowed)

linear probing: (h(k) + i) mod m

Keys tend to cluster, which causes long runs of probing.

quadratic probing: (h(k) + i^2) mod m

有些 slot 永远不会被用到，并且同样会 cause cluster.

Note: if the hash table contains less than ⌊m/2⌋ keys, then the insertion of a new key is guaranteed to be successful.

double hashing: (h1(k) + i * h2(k)) mod m

用两个 hash functions, two sources of “randomness”

Lecture 6 Amortized Analysis and Quicksort

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

Expansion factor = 1/(cost-2) + 1

Quicksort

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 cn^2 comparisons, make very unbalanced partitions, sorted array

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。

Lecture 7 Graphs and BFS

Graphs: used to model relationships between objects.

BFS 好处：

1. Useful for getting single-source shortest paths on unweighted graphs （每个 node 的 d[v]都是 shortest path）

2. testing reachability

3. Fast! Linear-time graph operation (O(|V|+|E|) with adjacency)

Data structures for the graph ADT

1. Adjacency matrix |V|x|V

takes space & runtime O(V^2) for BFS&DFS

directed graph:

根据 edge 的方向，统一从 V1 连线去 V2，两个 vertex 能连成一个 edge 的话 matrix 里相应位置就是 1，else 0.

Undirected graph:

如果两个 vertex 能连成一个 edge 的话，两个方向 matrix 里相应位置就是 1. matrix of an undirected graph is **symmetric**

2. Adjacency list of element |V|

takes space & runtime O(V+E) for BFS&DFS

每个 element 都存了从当前 element 能连接到的 node

directed graph: 单向的

Undirected graph: 双向的，会重复， matrix 更好

It takes space |V|+2|E|

Adjacency list is more space-efficient if |E| ≪ |V|² (graph is not very **dense**)

Matrix is more efficient than list: Check whether edge (vi, vj) is in E

BFS in a tree: use a queue

先建立一个 queue，把 tree root 放进 queue 里，while queue 不为空，先 dequeue 并 print 出当前 element，再依次把当前 element 的 child 放进 queue，循环

BFS in a graph:

White: Initial status，Gray: the first encounter

Black: all its neighbors have been encountered

pi[v]: I was introduced as whose neighbour?

找 shortest path 就一个一个套回去

d[v]: the distance from v to the source vertex (also the shortest path distance)

Pseudocode for the real BFS

1.initialize: for loop traverse 一遍 G，把每个 node 的颜色设为 white，d[v] = infinity，pi[v]=NIL

2. 创建一个 queue，把 source node 设为 grey，d[v] =0, enqueuer(source node)

3.while queue is not empty：dequeue

for each neighbor v of u:

if colour[v] = white:{

colour[v] = gray ; d[v] = d[u] + 1; pi[v] ← u

Enqueue(Q, v)}

 colour[u]= black

Lecture 8 DFS (Depth First Search)

The pseudo-code for DFS

Strongly Connected Components

DFS in a tree （preorder traversal）: use a stack
建立一个 stack，把 tree root push 进 stack 里，while stack 不为空，pop 出当前 element 并 print，再依次把当前 element 的 child 放进 stack，循环
也可以用 recursion 写：NOT_YET_DFS(root):
print root for each child c of x: NOT_YET_DFS(c)

BFS in a graph:
White: "unvisited" , Gray: "encountered" , Black: "explored"
1. time：incremented whenever someone's color is changed（当前一共走了几步）
2.pi[v]: I was introduced as whose neighbor?
3.**d[v]:** "discovery time", when the vertex is first encountered and becomes grey （NOT DISTANCE）
4.**f[v]:** "finishing time", when all the vertex's neighbors have been visited and becomes black

DFS(G, u): initialize: for loop traverse 一遍 G，把每个 node 设为 white，d[v],f[v]= infinity，pi[v]=NIL，time=0
for each vertex in graph：（而不是 for each child）
  if color[v] == white: DFS_visit(G, vertex)

DFS_visit(G, u):
Color[u] = grey; Time += 1; d[u]= time
for each neighbor v of u{
  if color[v] == white:
    pi[v] = u; DFS（G，v）}
color[u] = black; time+=1
f[u] = time # finishing time after exploring all neighbors

DFS 好处：
1. Detect cycle.
2. tell us whether a graph is connected
3. Topological Sort
Place the vertices in such an order that all edges are pointing to the right side.
1. Do a DFS 2. Order vertices according to their **finishing times f[v]**

- any pair of vertices can reach each other
- maximal set of vertices so that any vertex is reachable from any other
-use DFS to solve this for any directed graph!

How do we detect cycle? if a (directed) graph is cyclic if and only if a DFS yields a back edge.(相当于又指回了自己，不指回自己的不算 cycle) 怎么确定是 back edge：指向的东西 d 有东西，f 没有东西，就说明他在我的 path 上，我还没走完。如果 d 和 f 都不为空那就说明是旁支。
Note: after a DFS on a undirected graph, every edge is a tree edge or a back edge (not forward edge or cross edge)
the parenthesis structure (check edge types)
1. one pair contains the other pair
2.or one pair is disjoint of the other
Interval of u contains interval of v, if and only if u is an ancestor of v in the DFS forest.
如果 u 和 v 的 interval 互不相交，那 neither one is the ancestor

Lecture 9 Minimum Spanning Tree
Tree 的性质：
1. A tree with n vertices has exactly n-1 edges
2. removing one edge from T will disconnect the tree
3.adding one edge to T will create a cycle
a MST of a connected graph has |V| vertices， |V| - 1 edges.

keep deleting edges until a MST remains, in worst case, we need to delete n(n-1)/2 – (n-1) edges.

生长的方法(prim)
：keep one tree plus isolated vertices, use priority queue (min heap) to store all candidate vertices whose keys are the weight of the crossing edge. in worst case, we need to add O(V) edges.
Runtime analysis: each extractMin() takes O(log V) time, call extractMin() V times. check at most O(E) neighbours, each check neighbor could decreaseKey() which takes log V. so in total O((E+V)log V)= O(E logV)

构建的方法(kruskal)：uses disjoint set, sort all edges according to weight, add to MST from the lightest one,只要加上一个 edge 会 form 一个 cycle 就不加（the two endpoint must belong to two different component）。
Runtime analysis:
Sort the edges takes O(E log E), 要用到 disjoint set 的 operation 去 check "如果新加的 edge 的两个 end point 在不同的 component 里的话就可以 union 他们"

Lecture 10 Disjoint Set
MakeSet(x): set of one element, assign it as representative.
FindSet(x): return the representative of the set
Union(x,y): create a new set that unions the two sets that contain x and y, pick a new element as new representative.
Application:
1.KRUSKAL-MST()
2.finding connected components
Implementation
Linked lists
1.Normal circularly-linked list O(m^2)
Union: 交换（交叉）两组 head 的 next pointer O(1)
2.Linked list with extra pointer to head O(m^2)
union: append one list to the other, then update the pointers to head, takes O(length)
3.linked list with pointer to head and union-by-weight O(nlog n)
append the shorter one to the longer one
(need to keep track of the size (weight) of each list)

Trees (disjoint set forest)
each set is an inverted tree and the root(representative) points to itself
Trees with Union-by-rank O(m log m)
note: rank is upper-bound on its height (path compression does not maintain height info)
before union，rank 就是 height，path compression 之后每个 node 的 rank 也不改变（尽管 height 变了）
1. need to keep track of the tree's height, each node keeps a rank, which is its height
2. let the taller tree be the root, let the root with lower rank point to the tree with higher rank. (if two trees have the same rank, choose either root as the new root and increment its rank)
3. has height at most floor(n/2)

3、Trees with Path compression O(m log m)
你第一次 FindSet()经过一个 path 的时候就把你要找的那个 node 之前的所有 node compress 到都指向 root（子孙都成了 direct children）
4、Tress with path compression and union-by-rank O(m)
1.path compression happens in the FindSet operation
2. Union-by-rank happens in the Union operation

findSet(x):
if x != x.p:
  x.p = FindSet(x.p)
return x.p

According to the theorem that we learned in the lecture, the **safe edge** is the minimum weight edge crossing the two disjoint components of T - (u, v).

Lecture 11 Randomization and Lower Bound
1. Las Vegas algorithm: same answer, random runtime
2.Monte Carlo algorithm: same runtime, random answer(牺牲一点正确率来确保 runtime 的提升)
equality testing: choose a prime number p <= len(x)^2
retuen (x mod p) == (y mod p)
Probability of getting a wrong answer is upper bounded by the probability of getting a bad prime number: (2ln n)/n

Randomized algorithms:
1.guarantees expected performance
2.make algorithm less vulnerable to malicious inputs

Assume a insert sequence is a random permutation of {0,…n}, total possible insert sequence is n!.

Lower bounds
1.The lower bound n log n only applies to **comparison based** sorting algorithms with no assumptions on the values of the input. if we know the values of input, we can do better than n log n.
2.every comparison-based sorting algorithm has a corresponding **decision tree**. each leaf node corresponds to a possible sorted order of inputs. A decision tree has to contain n! possible orders for n elements.

General method of formal proofs of lower bounds
Adversarial argument: to prove a lower bound L(n) on the complexity of problem P, 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. (no matter what algorithm you use to solve, the worst-case is at least L(n))

reduction: proving one problem's lower bound using another problem's known lower bound.
If we know problem **B** can be solved by solving an instance of problem **A**, i.e., **A** is "harder" than **B,** and we know that **B** has lower bound **L(n)**, then **A** must also be lower-bounded by **L(n)**

Q1: augment AVL trees to implement 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. **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.

Master Theorem: (找的是 asymptotic bound) Let T(n) be defined by the reccurence T(n) = aT(n/b) + f(n), for some constants a ⩾1，b>1 and k ⩾1，then we can conclude the following about the asymptotic complexity of T(n): 注 意一定要满足形式 (1)Ifk=$\log_b$ a,thenT(n)=O(n^k logn). (2)Ifk<$\log_b$ a,thenT(n)=O($n^{\log_b a}$). (3)Ifk>$\log_b$ a,thenT(n)=O(n^k). When master theorem does not directly apply,把小的合到大的那个里在用

Q2: How many different insert sequences are there that would result in exactly the same tree as T? **Left subtree** 同形数量***right subtree** 同形数量***(左支节点数+右支节点数choose左支节点数)** Q3**.how many insert sequences would lead to a BST with the maximum height? 2^(n-1)因为每次都要选极值，每次都选择都有两个option（最大或最小），除了最后一位（因为没其他选择了）**Q4：**after DFS， select a vertex whose removal does not disconnect the graph? 只要是leaves就可以，如果v所指向的结点可以不通过v连接，就可以删除。**Q5：**MST，e1是最轻edge，prove e1一定是MST的一部分。假设一个MST不包括e1，现在吧e1放进去那一定形成一个cycle，最大的那个edge拿掉就会出现一个新的MST，说明原来的不是一个MST。第二轻的edge一定在MST里，第三轻的不一定在MST.**Q6.**a graph如果有双数node就一定有cycle。**

Q7.一帮孩子互相仇恨那道题，给的关系是谁恨谁，但我们不能用仇恨作为 edge，要先把所有 edge 全都连起来再删除仇恨的 edge，这样如果最后要知道还可能 arrange 这些熊孩子让他们不打架的话就 check 这个 graph is connected 就好。check 用 DFS 生成一个 tree，看他的结点等不等于 total vertex。
Q8.还有一种 graph 记得是 decision tree！比如一个 puzzle，他的每个结点就是每走一步后新的 puzzle。用 BFS 就可以知道解这个 puzzle 的最快解。
Q9.Prove that an undirected graph is bipartite iff it contains no cycle whose length is odd (called simply an "odd cycle")
Q10.Minimum cycler question: build a maximum-spanning tree,然后用原来的 set of edges 减去这个 MST 里的 edge。（剩下的 edge 一定是一个 cycle 里的）