Runtime

WC: upper bound $O()$
let n be arbitrary. For any input x of size n (perform at most ...) for a total of at most
WC: lower bound $\Omega()$
Find a input x of size n. $T(x) \geq ... \geq \Omega()$ at least

极值问题，不sorted.

HEAP: for any node at index i

- Parent(i) = $\lfloor \frac{i}{2} \rfloor$; left[i] = 2i; right(i) = 2i+1
- { Parent priority > children priority; no order btw siblings; almost complete }

leaf = $\lceil \frac{n}{2} \rceil$, internal = $\frac{n}{2}$

- MAX $\Theta(1)$
- INSERT $\Theta(\log n)$ { size+1; insert at heapsize; bubble up } (leaf 里有 max or min)
- EXTRACT_MAX $\Theta(\log n)$

```
A[1], A[A.heapsize] = A[1], A[heapsize]
A.heapsize = A.heapsize - 1
HEAPIFY(A, 1)
return A[A.heapsize + 1]
```

从1开始. 一层一层

- HEAPIFY(A, i) logn 次 depth↓ Bubble up↑ height↑

```
largest = i
l = left(i)
r = right(i)
if l ≤ A.heapsize and A[l] > A[i]:
    largest = l
if r ≤ A.heapsize and A[r] > A[largest]:
    largest = r
if largest ≠ i:
    exchange A[i] with A[largest]
    MAX_HEAPIFY(A, largest)
```

BUILD_HEAP(A): 从[n/2]开始 $\Theta(n)$

```
A.heapsize = A.length
for i = ⌊A.length/2⌋ down to 1
    MAX_HEAPIFY(A, i)
```

HEAP_SORT(A): $\Theta(n\log n)$

```
Build_MAX_HEAP(A)          Θ(n)
for i = A.length down to 2:     -> n-1次   or heap is not empty
    exchange A[i] with A[1]
    A.heapsize = A.heapsize - 1
    MAX-HEAPIFY(A, 1)      -> logn 次
```

Priority Queue:

- INSERT(Q, x)   "no duplicate"
- MAX(Q)   "no search"
- EXTRACT_MAX(Q)   没有重复

Dictionary (ADT):

- SEARCH(S, k)   x has key x.key
- INSERT(S, x)   key is unique
- DELETE(S, x)

BST:

- INSERT(S, x): S.root = BST_INSERT(S.root, x)
- BST_INSERT(root, x)   同 AVL
- SEARCH(S, k):

```
node = BST_SEARCH(S.root, k)
if node == NIL: return NIL
return node.item
```

- BST_SEARCH(root, k)

```
if root == NIL: pass
elif k < root.key:
    root = BST_SEARCH(root.left, k)
elif k > root.key:
    root = BST_SEARCH(root.right, k)
else: # k == root.key
    pass
return root
```

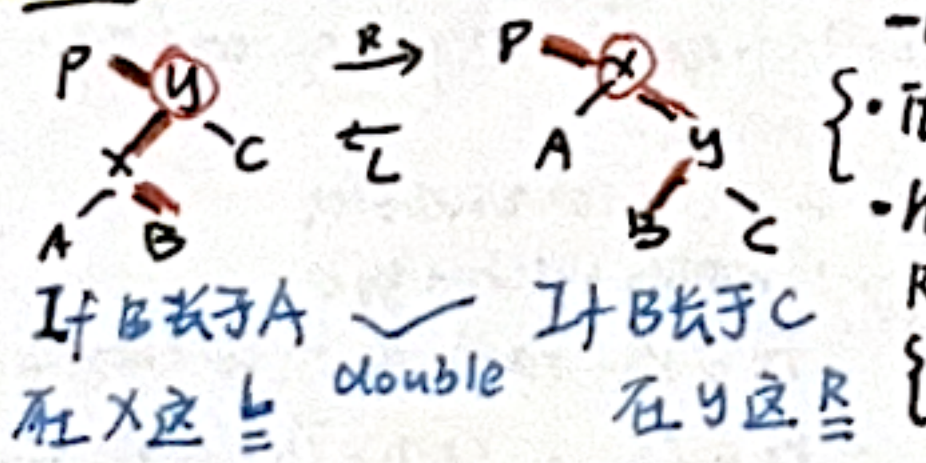
- DELETE(S, x): root = BST_DELETE(S.root, x)
- BST_DELETE(root, x):

```
同 AVL-DELETE
else:
    root.item, root.right = BST_DEL_MIN(root.right)
```

- BST_DEL_MIN(root):

```
if root.left == NIL:
    return root.item, root.right
else:
    item, root.left = BST_DEL_MIN(root.left)
    return item, root.
```

AVL (Balanced Search Tree) order



- Node: { item, left, right, height, parent, size }
- Ref: { root, NIL }

max min

BS = Height(left_sub) - height(right_sub)
AVL Invariant: bs ∈ [-1, 1]

AVL_INSERT(root, x): $O(\log n)$

```
# insert x into subtree at root; return new root
if root == NIL:
    root = AVLNode(x)        · item = x  · left = NIL  · right = NIL  · height = 0
elif x.key < root.item.key
    root.left = AVL_INSERT(root.left, x)
    root = AVL_REBALANCE_RIGHT(root)
elif x.key > root.item.key:
    root.right = AVL_INSERT(root.right, x)
    root = AVL_REBALANCE_LEFT(root)
else: x.key == root.item.key
    return root.
```

AVL_REBALANCE_LEFT(root):

```
# pre: root != NIL
# first recalculate height
root.height = 1 + max(root.left.height, root.right.height)
# rebalance to left, if necessary:
if root.right.height > 1 + root.left.height:
    # check for double rotation:
    if root.right.left.height > root.right.right.height:
        root.right = ROTATE_Right(root.right)
    root = ROTATE_LEFT(root)
return root.
```

AVL_ROTATE_LEFT(parent)

```
# pre parent != NIL   parent.right != NIL
child = parent.right
parent.right = child.left
child.left = parent
parent.height = 1 + max(parent.right.height, parent.left.height)
child.height = 1 + max(child.left(right).height)
return child
```

- AVL_DELETE(root, x): $O(\log n)$

```
# Delete x from tree at root; return new root
if root = NIL: # not in tree:   pass
elif: x.key < root.item.key:
    root.left = AVL_DELETE(root.left, x)
    root = AVL_REBALANCE_LEFT(root)
elif x.key > root.item.key:
    root.right = AVL_DELETE(root.right, x)
    root = AVL_REBALANCE_RIGHT(root)
else: # x.key == root.item.key
    if root.left == NIL: root = root.right
    elif root.right == NIL: root = root.left
    else: # replace from taller subtree
        if root.left.height > root.right.height:
            root.item, root.left = AVL_DEL_MAX(root.left)
        else:
            root.item, root.right = AVL_DEL_MIN(root.right)
        root.height = 1 + max(root.left(right).height)
        # no need reblance
return root
```

AVL_DEL_MAX(root): # pre root != NIL

```
if root.right == NIL: return root.item
else: item, root.right = AVL_DEL_MAX(root.right)
    root = AVL_REBALANCE_RIGHT(root)
    return item, root
```

Augmentation

Ordered set: Search, Insert, delete, rank, select

1. sum: sum of the keys of the nodes in the tree rooted at node
2. Search for the node u that has...
   if search fails....
   set u's ... to ...
   For each node on the path from u to root set...
   recalculate (base on children's attr)
3. The running time of AVL is $O(\lg n)$ bc it performs a constant amount of work at each node, and recurses down a single path from the root to a leaf.
   The running time of insert and delete is unchanged bc the new info can be maintain in constant time for each node, based on the values on nodes's children.
4. For insert/delete: after each recursive call recalculate new attribute as ... 子to children

Hash average, expected × order.

1. create a hash table of size m using chaining and hash function mapping .. onto table
2. For each index i, it search A[i] in T[h(A[i])]
3. we assume the size m of hash table is within a constant factor of the size of the data n. ($m = kn$ for some constant $k>0$) We will also assume SUHA (each distinct value is equally likely to be hashed into any of T's slot), the expected length of a linked list in T is α, where $\alpha = \frac{n}{m}$ then $\alpha \in \Theta(1)$, and so the expected length of a linked list in T is $\Theta(1)$. Therefore, the expected time for an INSERT operation is $\Theta(1)$. The Query operation runs in constant time since accessing every element in A is a constant time.

Quick Sort:

```
if length(A) ≤ 1: return A
select pivot ele p ins   -> at random!   p = S[0]
L <- elements in A < P
E <- elements in A = P
G <- elements in A > P
return (QUICKSORT(L) + E + QUICK(G))
```

$WC \in \Theta(n^2)$ $AC \in \Theta(n\log n)$ $BC(n\log n)$

Average case

1. Sample space $A_n = [n, n-1, ..., 1]$, and $S_n = \{(A_n, k): 0 \le k \le n\}$
2. $Pr[(A_n, k)] = \begin{cases} p, & \text{if } k=0 \\ \frac{1-p}{n}, & \text{if } k=1,2,...n \end{cases}$
3. let c be # of comparisons of times the... is performed on (A_n, k). The total runtime is $\Theta(c)$

bc the algo performs a constant # of ops for each such comparison. when $1 \le k \le n$, $C(k) = k$ when $k=0$, $C(k)=n$.

$E[C] = \sum_{i=1}^{n} Pr[C[k]=i] \cdot i$ -

Aggregate method → no constant

for all n, n operation has a worst case running time T(n).

~~T(n)~~ Amortized = $\frac{T(n)}{n}$

Accounting method

charge ~~digits~~ $\frac{16}{15}$ ← digit.

C1: after each operation, ...

Initially, ...

Assume.. holds, next operation.

- grow
- not grow.

∴ ~~Total~~ amortized = ~~m~~ Total cost for m ops

(total credit never < 0) ≤ $\frac{\text{total charge}}{m}$

Graph

	Adjacency list	matrix
space	|V|+|E|	|V|²
Access a vertex	O(1)	O(1)
all vertex	O(n)	O(n)
a/ all edge	O(|V|)	O(1)
Loop neighbors	O(|V|) 可小 ← 找到neig O(1)	O(|V|) 只能
Loop all edges	O(|E|)	O(|V|²)
BFS	O(m+n)	O(n²)
DFS	O(m+n)	O(n²)

BFS(G,s)

```
# Initialize tracking info for all vertices
for v ∈ G.V:
    color[v] = white
    π[v] = NIL          Predecessor
    d[v] = ∞
# initialize empty queue and source vertex
Q = Make-Queue()
color[s] = grey
π[s] = NIL
d[s] = 0
Enqueue(Q, s)
# main loop. LI: Q contains all (and only) grey vertices
while not isEmpty(Q):
    u = Dequeue(Q)
    for v ∈ G.adj[u]
        if color[v] = white:
            color[v] = grey
            π[v] = u
            d[v] = d[u]+1
            Enqueue(Q, v)
    color[u] = black
```

undirect (BFS) { Tree / cross

思路: BFS 得是 shortest 经过一波 发现不行

? u<v ⇒ d[u] ≤ d[v] 先访问的. Why?: 后面的要么是同层/下层

① BFS - explorer current component.
② 找 safe to delete node.

runtime ∈ θ(m+n)

3 shortest path, 有起点,
4 d[v] = shortest distance from root to v
5. d[u] = +∞ ⇐ disconnected

DFS(G)

```
# Initialization
for v ∈ G.V:
    d[v] = f[v] = ∞
    π[v] = NIL
time = 0
for v ∈ G.V:
    if d[v] == ∞:
        DFS_VISIT(G, v)
DFS_VISIT(G, v):
    d[v] = time = time+1
    # do some thing with v
    # explorer v's edge
    for u ∈ G.adj(v):
        if d[u] == ∞:
            π[u] = v
            DFS_VISIT(G, u)
    f[v] = time = time+1
```

θ(n) ; 碰到grey → cycle ;) n ; 总共 m ; 无起点.

(Example: graph A, B, C, D, E; Q: B C D E; Tree {} 3,C 2,C 2,C; {(C,D)} 3,C 1,D; {(C,D),(D,E)} 3,C; {(C,D),(D,E),(C,B)})

DFS on undirect can only ~~create~~ produce tree and back edge (× cross)

Runtime = θ(m+n)

back ward → create circle; forward; cross ward; { Tree. (DFS) / back.

Application:

1. cycle deduction
2. Topological Sorting: order v in directed graph ⟺ G contains no cycle ⟺ DFS finds no back edge
3. Find component.
4. order
{ 1. run DFS θ(m+n) / 2. List vertices by decreasing finish time θ(n)

Parenthesis Theorem:

after DFS. (nested / disjoint intervals of u, v) 不能 (overlapping)

Strongly connected component 有来回 path u↔v

1. run DFS on the transpose of G θ(m+n)
2. Reorder vertices in decreasing order of finish time (in G.V and with each adjacency list). Run DFS with new order and original edges θ(m+n) |scc| > 1 ⇒ cyclic
3. Each ~~tree~~ DFS tree = one strongly connected component θ(m+n)

Minimum Spanning Tree

contradiction 多个 G

- input: connected, undirect, weighted graph
- output: a connect, acyclic subset of edges with min weight.

(n-1 edges) add one edge → a cycle

delete one edge → two tree

Prim's Algorithm

```
Prim(G, w: E→ℝ, r ∈ G.V):
    T = ∅   # current tree edges
    # Initialization
    Q = Make_Queue()
    for v ∈ G.V:
        priority[v] = ∞
        π[v] = NIL
        Q.insert(v)
    # set r as root
    Pr[r] = 0
    Q.Decrease_Key(r)
    # main loop
    while not Q.isEmpty():
        # connect a vertex with minimum priority
        u = Q.EXTRACTMIN()
        if π[u] ≠ NIL: T = T ∪ {(π[u], u)}
        # update priorities for neighbors of u
        for v ∈ G.adj[u]:
            if v ∈ Q and w(u,v) < Pri[v]:
                π[v] = u
                Pri[v] = W[u,v]
                Q.Decrease-key(v)
    return T
```

place all vertex in Q } linear time

Ancestor successor

n iteration

-lgn, lgn+... | ~nlogn

m 次 } mlgn ; lgn

runtime ∈ (m+n) lgn → (m log n)

Note: Decrease_key: need to track index of v — position of v in min-heap and keep this updated during swaps.

KRUSKAL(G, w):

```
sort edges: w(e1) ≤ w(e2) ≤ ... ≤ w(en)
T = ∅
for v ∈ G.V:
    Make_Set(v)
for ei = e1, e2, ..., em:
    let (ui, vi) = ei
    if FIND_SET(ui) ≠ FINDSET(vi):
        UNION(ui, vi)
        T = T ∪ {ei}
return T
```

∈ mlogm ; component ; θ(n) ; m 次

amortized cost α

runtime ∈ θ(m log* m)

Disjoint sets

(sequence of #) ~ (m log n)

- make_set(x) O(1)
- FIND_set(x) return the representive O(α(n)) — amortized
- UNION(x, y) by rank O(α(n)) (by height/weight (nlogn))

path com

no path comparison was done

Implementation: · rank = height of tree if

- Make set(x) · rank = 0
- FND-SET(x): — path comparission — ranks unchange
- UNION(x,y): $r_1 > r_2$ rank of A unchanged; $r_1 = r_2$ A.rank increment.

A ← B

```
union_by_rank
r1 ← find_set(w1)
r2 ← find_st(w2)
if r1 ≠ r2:
    if r1.rank 大:
        r1.children.append(r2)
    elif: 反
    else:
        r1.chid.append(r2)
        r1.rank++
```

linear time

eg: Distance(G, e1, e2, ... em):

```
for i = 1, 2, ..., n
M = 1
for ei = em, em-1, ... e1:
    let (u, v) = ei
    u' = FIND_SET(u)
    v' = FIND SET(v)
    if u' ≠ v':
        x = UNION(u', v')
        M = MAX(M, FINDSIZE(X))
if M > ⌊n/3⌋:
    return (u, v)
```

To support this procedure, we store the size of each tree in an additional variable associated with the root of each tree.

Since we use the forest of rooted tree disjoint ~~dat~~ sets, with weighted unions and path compression rules, the time is O(m log* n)

O(n) to initialize; O(m) to form the list of edges; (n-1) unions and ~~2m~~ at most 2m FINDSET (at most 2 find set for each edge)

lower bound

- \# of outcomes = n!
- \# of leaves ≥ n!
- WC ~~of~~ # of comparision of any comparison based algo ≥ min height of any com-tree ≥ log(# of leaves) ≥ log(# of outcomes) ≥ log(# of outputs)

Adversal Argument

1. let A be an arbitrary algo
2. Construct an input that makes the Arbitrary algo runs in lower bound. show: less than lower bound cannot satisfy the problem
3. Since A is arbitrary, we know we need at least lower bound to solve.

- eg: let D be an arbitrary algo that..

In WC, it compares n-2 times

let D' be an algo that run n-3 times comp. Then at least one pair A[i], A[i+1] is not compared. 自己写一个input

Suppose D' returns 0, make A[i] ≠ A[i+1] ↺ D failed

suppose D' returns 1, make A[i] = A[i+1] ↺

So D' is not suffice. so need examine at least. 最好情况 - Lower Bound.

---

find max:

- input: unsorted A output: i of max
- Info theory L.B: Ω(logn)
- AA:
  - all comparision "A[i] >? A[j]"
  - upper bound n-1 comparison
  - WTS: every ~~com~~ algo that comp. ≤ n-2 must give wrong output (≥ 2 undetermined)

When comp. tree reaches a leaf, it must out put the index of some undetermin ele. and the algo return i, there is always an input where the max is j

a_i 和 a_j 比较 $\frac{2}{j-i+1}$

$nP_k = \binom{n}{k} k!$

$nC_k = \binom{n}{k} = \frac{n!}{(n-k)!k!}$