1. Analysis of algorithms

- Count number of primitive operations (basic operations provided by a programming language)
  - Constant (coefficient) does not matter when n is large, interested in growth rate instead
  - Only care about dominating terms: Drop "low-order" terms, ignore leading constants
- Best case, Worst case (upper bound), Average case (expected value, need probability distribution)

Mathematical definition

$f(n) = \Theta(g(n)) \Leftrightarrow \exists\, c_1, c_2, n_0\ s.t.\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n)\ \forall n \ge n_0$	$\approx (g(n) = f(n))$
$f(n) = O(g(n)) \Leftrightarrow \exists\, c > 0, n_0 > 0\ s.t.\ 0 \le f(n) \le cg(n)\ \forall n \ge n_0$	$\approx (g(n) \ge f(n))$
$f(n) = \Omega(g(n)) \Leftrightarrow \exists\, c > 0, n_0 > 0\ s.t.\ 0 \le cg(n) \le f(n)\ \forall n \ge n_0$	$\approx (g(n) \le f(n))$
$f(n) = \Theta(g(n))\ iff\ f(n) = \Omega(g(n))\ and\ f(n) = O(g(n))$	
$f(n) = o(g(n)) \Leftrightarrow$ *for any* $c > 0$, $\exists n_0 > 0\ s.t.\ 0 \le f(n) < cg(n)\ \forall n \ge n_0$	
$f(n) = \omega(g(n)) \Leftrightarrow$ *for any* $c > 0$, $\exists n_0 > 0\ s.t.\ 0 \le cg(n) < f(n)\ \forall n \ge n_0$	

Limit (always use this, unless specified)

$$f(n) = \Omega(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} \in (0,\infty] \in \{c,\infty\}$$
$$f(n) = \Theta(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} \in (0,\infty) \in \{c\}$$
$$f(n) = O(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} \in [0,\infty) \in \{0,c\}$$

Harmonic series

$$\sum_{i=1}^{k} \left[\frac{1}{i}\right] \in [\ln(k+1), \ln(k)+1]$$

Sum of geometric series

$$S_n = \frac{a_1(1-r^n)}{1-r}$$

Master theorem

For an algorithm that runs in time complexity $T(n)$,

If $T(n)$ can be written as a recurrence formula in form of $T(n) \le aT\left(\frac{n}{b}\right) + O(n^d)$

If we divide original problem into a subproblems, parameter reduced from n to $\frac{n}{b}$

We have $T(n) = \begin{cases} O(n^d) & if\ d > \log_b a \\ O(n^d \log n) & if\ d = \log_b a \\ O(n^{\log_b a}) & if\ d < \log_b a \end{cases}$

Special proof examples

Strategy: Decide intuitively whether it is correct or not, then use limit, let constant or counterexample

Q: $100n^2 + 20n = \Theta(1000n)$? (False)

$$Assume\ it\ is\ true, then\ \exists\ c_1, c_2, n_0 > 0\ s.t. \forall n > n_0, c_1(1000n) \le 100n^2 + 20n \le c_2(1000n)$$
$$\Rightarrow c_1 \le n + {}^1/_{50} \le c_2$$
$$Take\ n = \max\{c_2, n_0\}, then\ n + {}^1/_{50} > c_2 \Rightarrow Contradiction$$

Main point: $f(n)$ *should be upper bounded, but somehow when n is large, the bound fails.*

Q: $\min\{f(n), g(n)\} = O(f(n) + g(n))$? (True)

$$\min\{f(n), g(n)\} \le f(n) + g(n)\ for\ all\ n \ge 0$$

Q: $If\ f(n) = O(n), then\ 2^{f(n)} = \Theta(2^n)$? (False)

$$Let\ f(n) = 2n. Then\ 2^{f(n)} = 2^{2n}$$
$$2^{2n} \le c_2 2^n \Rightarrow 2^n \le c_2$$
$$Cannot\ find\ c_2, n_0\ s.t.\ for\ all\ n \ge n_0, 2^n \le c_2\ holds.$$

**Q: $f(n) = \Omega(f(n) + 10)$? (False)

$$Let\ f(n) = \frac{1}{n}, Assume\ it\ is\ true. Then\ \exists n_0, c > 0, \forall n > n_0, \frac{1}{n} > c\left(\frac{1}{n} + 10\right)$$
$$\Rightarrow 1 > c(1 + 10n) \Rightarrow n < \frac{1-c}{10c}, contradictory\ to\ \forall n > n_0$$

Q: $\omega(n) + \omega(n^2) = \Omega(n^2)$? (True)

$$f_1(n) = \omega(n), f_2(n) = \omega(n^2)$$
$$f_1(n) > c_1(n)\ \forall n > n_1, f_2(n) > c_2(n^2)\ \forall n > n_2$$
$$f_1(n) + f_2(n) > c_1(n) + c_2(n^2) \ge c_1(n)\ \forall n > \max\{n_1, n_2\}$$

Q: $If\ f(n) = \omega(g(n)), then\ \log(f(n)) = \omega(\log(g(n)))$? (False)

$$f(n) = n^2, g(n) = n$$
$$\log(f(n)) = 2\ logn, \log(g(n)) = logn$$
$$But\ let\ c > 2, for\ any\ n > 1, 2\ logn < c\ logn$$

**Q: $If\ f(n) = \Theta(g(n)), then\ \log(f(n)) = \Theta(\log(g(n)))$? (False)

$$f(n) = 2, g(n) = 1 + \frac{1}{n}$$
$$\exists c_1 = 1, c_2 = 2, n_0 = 1, \forall n \ge n_0, c_1 g(n) \le f(n) \le c_2 g(n)$$
$$Assume\ true, \exists c_3, c_4, n_1\ s.t. \forall n \ge n_0, c_3 \log\left(1 + \frac{1}{n}\right) \le \log(2) \le c_4 \log\left(1 + \frac{1}{n}\right)$$
$$2 < \left(\frac{n+1}{n}\right)^{c_4} \Rightarrow n \le \frac{1}{2^{\frac{1}{c_4}} - 1}, contradicts\ n \ge n_1$$

2. Recursion, Divide and Conquer

- Solving linear recurrence: Write down a few iterations by hand, then use summation formula
- Divide and conquer: Assume that the sub-problem is solved ("ask god for help"), think how to combine sub-problems into the main problem
  - Remember the base case
- Representation of running time (cases): $T(n) = \begin{cases} c_1 & if\ n = 1 \\ f(n-1) + c_2 n & if\ n > 1 \end{cases}$

Tower of Hanhoi

```
TofH (A, B, C, n) { // source: A, destination: C
    if (n == 1)
        move disk from A to C
    else {
        TofH(A, C, B, n - 1);
        move disk from A to C
        ToFH(B, A, C, n - 1);
    }
}
```

$$f(n) = \begin{cases} 1 & if\ n = 1 \\ 2f(n-1) + 1 & if\ n > 1 \end{cases}$$

Fibonacci

```
Fib1(n) {
    if (n <= 2) return 1;
    return Fib1(n - 1) + Fib1(n - 2);
```

$$\begin{gathered} Proof: T(n) > 2^{(n-2)/2} \\ T(n) > T(n-1) + T(n-2) \\ > T(n-2) + T(n-3) + T(n-2) \\ = 2T(n-2) + T(n-3) \\ > 2T(n-2) \\ > 2\big(2T(n-4)\big) \\ > 2^k\, T(n-2k) \end{gathered}$$

```
Power(A, p) {
    if (p == 0) return I;
    if (p % 2 == 1) return A * Power(A, p - 1);
    T = Power(A, p / 2);
    return T * T;
}
```

$$T(n) = \begin{cases} c_1 & if\ n = 1 \\ T(n/2) + c_2 & if\ n > 1 \end{cases}$$

Matrix Linear Recurrence

Suppose that we have a linear recursive equation

$$f_i = \sum_{j=1}^{k} b_j f_{i-j}$$

where $b_1, b_2, \dots, b_k, f_1, f_2, \dots, f_k$ are known constatns. Then the following equation matrices holds for some $i \geq k$

$$\begin{bmatrix} b_1 & b_2 & b_3 & \cdots & b_{k-1} & b_k \\ 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} f_i \\ f_{i-1} \\ f_{i-2} \\ f_{i-3} \\ \vdots \\ f_{i-k+1} \end{bmatrix} = \begin{bmatrix} f_{i+1} \\ f_i \\ f_{i-1} \\ f_{i-2} \\ \vdots \\ f_{i-k+2} \end{bmatrix}$$

$$\begin{gathered} Av_i = v_{i+1} \\ A^{(n-k)} v_k = f_n \end{gathered}$$

Maximum subarray

```
// idea: find maximum subarray in first and second half,
//       find maximum subarray across midpoint, sweep forward and backward in O(n)

Max_subarray(A, l, r) {
    (l1, r1, max1) = Max_subarray(A, l, (l + r) / 2);
    (l2, r2, max2) = Max_subarray(A, (l + r) / 2, r);
    (l3, r3, max3) = Max_subarray_middle(A, l, r); // O(n)

    if (max1 > max2 and max1 > max3) return (i1, j1, max1);
    if (max2 > max1 and max2 > max3) return (i2, j2, max2);
    if (max3 > max1 and max3 > max2) return (i3, j3, max3);
}
```

$$\begin{gathered} T(n) = 2T(n/2) + cn \quad for\ n > 1 \\ \Rightarrow T(n) = O(n\ logn) \end{gathered}$$

Find min(A[j] – A[i]) s.t. i<j

```
MinValue(A, l, r) {
    If (l + 1 >= r) return INF;
    Mi = (l + r) / 2;
    (i1, j1, min1) = MinValue(A, l, Mi)
    (i2, j2, min2) = MinValue(A, Mi, r)
    (i3, j3, min3) = Min(A, Mi, r) – Max(A, l, Mi)
    Return Min(min1, min2, min3) and corresponding i, j
```

Generate size <= k combination

```
Subset(array A, array mark, start, k) [
    If (start == n or k == 0) {
        For (I = 1 to n) {
            If (mark[i] == 1) output a[i]
    } }
    Mark[start] = 1;
    Subset(A, mark, start + 1, k - 1);
    Mark[start] = 0;
    Subset(A, mark, start + 1, k);
```

Generate permutation

```
Perm (A, k, n) {
    // characters stored in A[1..n].
    // output all permutations of A[1..k] with A[k+1..n] appended.
    // chars in array A should be in the same order as it was before perm is called

    If (k == 1) ouput A[1..n]
    Else
        For (I = 1 to k) {
            Swap(A[i], A[k]);
            Perm(A, k – 1, n);
            Swap(A[i], A[k]);
        }
}

// call: Perm(A, n, n);
```

Find fixed point (A[i] == i) in strictly decreasing array

```
Binary search:
    Check middle
    If (value large than index), check right part
    Otherwise check left part
```

3. Introduction to data structures

- Abstract data types (ADT): Define what data; What operations required; (Based on applications)

List

```
Create(L) // create an empty list L
Search(L, x) // return an index to the element x or return -1 if X is not in L
Insert(L, x, i) // insert item x at position i + 1, need to check overflow
Delete(L, i) // delete item in L at position i
```

```
Struct List {
    int length;
    element entry[Max_Len];
};
List L;
```

Stack: Last in first out

```
Empty(S) // return if S is empty or not
Top(S)   // return the element at the top of S
Push(S, x) // insert x to the top of S
Pop(S)   // return and delete the elemtnat the top of S
```

```
Stuct stack{
    Int index = -1;         // points to top of stack
    Element entry[max];
}
Stack S;

// Emtpy(S): return (S.index == -1);
```

Queue: First in first out

```
// Implementation: two pointers, circular array

// head points to first element, tail points to next available slot

bool Empty(Q): return Q.head == Q.tail
bool Full(Q) : return Q.head == (Q.tail + 1) mod max
void Enque(Q, x): Q.entry[Q.tail] = x, Q.tail = (Q.tail + 1) mod max
void Deque(Q)   : x = Q.entry[Q.head]; Q.head = (Q.head + 1) mod max; return x;
```

Dictionary

```
Insert(T, x) // insert an element x into a set T
Search(T, k) // search a record with key = k in a set T
Delete(T, x) // delete an element x from a set
```

Graph

- Simple: No self loops, No double edges
- Undirected: A-C equal to C-A
- Multigraph: No self loops, Allow double edges
- Pseudograph: Allow self loops, Allow double edges
- If |E| is large (dense graph), then adjacency matrix is better (space efficiency)
- If |E| is small (sparse graph), then adjacency list is better (space efficiency)

Topological Sort: O(V+E): Detect Cycles   Lexi. smallest: priority_queue All orders: backtracking

```
for (all vertices v)
    if (indegree[v]==0) push v into queue q      // priority_queue also ok
while (q is not empty){
    x = q.deque();                   // order.push_back(x)
    for (all nodes connected to x)
        if (--in[v]==0) q.push(v)
}
```

DFS: O(V+E) Example: Flood Fill (Static DSU), DFS Tree (Re: Connectivity)

```
dfs (v){
    visited[v] = 1;  // color[v] = gray
    st[v] = time++;
    for (all vertex w adjacent to v)
        if (visited[w] == 0)    dfs(w)
    ft[v] = time     // color[v] = black, rev_ts.pb(v);
}
```

BFS: O(V+E) Application: Shortest path, State searching (Water jug problem), BFS Tree

"BFS queue only contains elements from at max two successive levels of the BFS tree."

Multisource BFS: O(V+E): Push all the sources into the queue first

```
bfs (x){
visited[x]=1, distance[x]=0;
q.push(x);
while (q.size()!=0){
    t = q.front(); q.pop();
    for (all vertex w adjacent to t){   // int w:v[t]
        if (visited[w]==0){
            visited[w]=1;
            q.push(w);      distance[q] = distance[t] + 1
} } }
```

Path extraction (with par[])

```
for (auto t = end; t != -1; t = par[t])  path.pb(t);
reverse(path.begin(), path.end());
```

4. Hashing

- Direct addressing: Elements with key i is stored in $Table[i]$
  - Storage: $O(|U|)$, $|U|$ can be large, while $|K|$ can be relatively small
  - Keys in the universe U can be mapped to I (integer domain), one to one
  - Given a key k, the mapped integer i can be computed in $O(1)$
- Solution: Allow more than one key map to the same array entry: Collision: $h(k_i) = h(k_j)$
  - Try to find a good hash function s.t. collision does not occur that often
  - Design collision resolution strategy

Chaining (open hashing)

- Keep a linked list of elements with same hash value
- Load factor: $\alpha = n/m, where\ n = \#keys, m = size$
- Average time complexity for searching: $\Theta(1+\alpha)$
  - Ideal case (simple uniform hashing): Any key is equally likely to hash into any slot

Hash function

- Distribute keys evenly into m slots, but not easy to find; Fast to compute; Use all information
- Division method: $h(k) = k\ mod\ m$
  - m should not be close to a power of 2 (not use all bits), or a power of 10. Good values are primes
  - E.g. n = 2000, α = 3. $m \approx 2000/3 = 667, pick\ m = 701$
- Multiplication method: $h(k) = int(frac(k \times A) \times m),\ where\ 0 < A < 1$
  - Choice of m is not that important, usually take $m = 2^p$ $[p\ most\ sig. bits\ from\ fractional\ part]$
  - Choice of A is important: Close to an irrational number (e.g. golden ratio)

Open addressing

- All elements are stored in the hash table (T[i] = element x, "NIL" or "Deleted")
- No. of records stored in T (n) <= No. of slots in T(m) (load factor $\alpha \leq 1$)
  - Advantage: Avoid pointers and linked list, save space
- Collision handling: Probe sequence
  - Given k, compute a sequence <s0, s1, s2, …>, pick first unoccupied slot
  - Linear probing: $h(k,i) = (h'(k) + i)\ mod\ m$
    - Problem: Primary clustering, long runs of occupied slots build up
  - Quadratic probing: $h(k,i) = (h'(k) + c_1 i + c_2 i^2)\ mod\ m$
    - Problem: Secondary clustering: IF two keys have same initial probe position, their probe sequences are the same
  - Double hashing: $h(k,i) = \left(h_1(k) + ih_2(k)\right)\ mod\ m$
    - $h_2(k)$ must be relatively prime to m for entire table to be searched (permutation of 0 … m)
- May need to reorganize hash table if many deletions have occurred
- When the table is getting full, we should build a larger table and rehash the elements there

Prove probe sequence is permutation (prove by contraction)

Use gcd condition if necessary

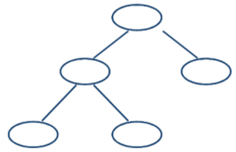
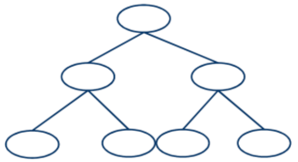
$$h(k,i) = \left(h'(k) + {}^1\!/_2\,(i+i^2)\right) mod\ m, where\ m = 2^p$$
$$Assume\ h(k,i) = h(k,j)\ where\ 0 \leq i,j \leq m-1, i \neq j$$
$$k + \frac{1}{2}(i+i^2) = k + \frac{1}{2}(j+j^2)\ \ mod\ m$$
$$\frac{1}{2}(i+i^2-j-j^2) = 0\ \ mod\ m$$
$$(i-j)(i+j+1) = 0\ \ mod\ 2m$$
$$Parity: Either\ (i-j)\ mod\ 2m = 0\ \ or\ \ (i+j+1)\ mod\ 2m\ = 0$$
$$1 \leq i-j \leq m-1, \qquad 1 \leq i+j+1 \leq 2m-1, \qquad both\ indivisible\ by\ 2m$$
$$Contradiction$$

5. Tree

- Connected undirected graph with no cycles
- Siblings: nodes with same parent
- Degree: number of children a node has
- Depth: Length of a path from root to v (root has depth 0)
- Height: max{depth of a node in a tree}

- Ordered tree: rooted tree where the children of each node are ordered (e.g. left to right)
- M-ary tree: Every internal node has no more than m children
  - Number of nodes for complete m-ary tree with height h: $1 + m + m^2 + \cdots + m^h = \frac{m^{h+1}-1}{m-1}$
- Binary tree: Left child of T[i] = T[2i + 1], right child of T[i] = T[2i + 2] (rooted at 0)

Full vs Complete

Full tree: Every internal node has exactly m children 	Complete tree: All leaves are of the same depth, every internal node has exactly m children  m^h leaves

For any non-empty binary tree, $n_0 = n_2 + 1$

$$n = n_0 + n_1 + n_2$$
$$n - 1 = 1 * n_1 + 2 * n_2 \qquad (number\ of\ edges)$$
$$Substituting, n_0 + n_1 + n_2 - 1 = n_1 + 2n_2, n_0 = n_2 + 1$$

Traversing order

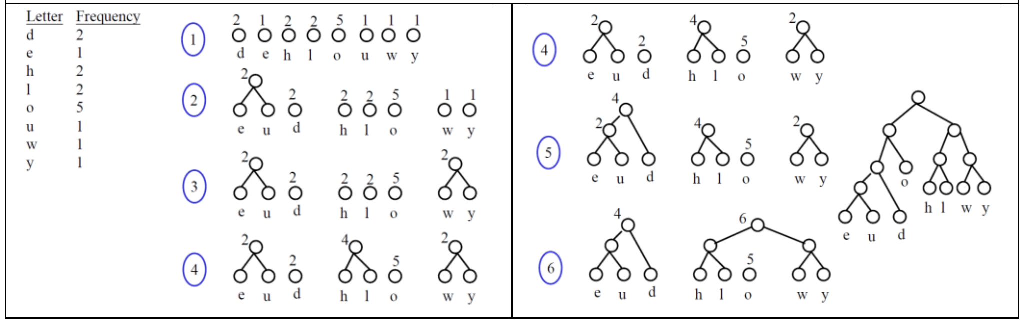
Given the pre-order/post-order and in-order traversal, we can construct the binary tree uniquely

Pre-order: st[], store node val	In-order: Only in binary tree	Post-order: re children ans, ft[]
Process node	Traverse left, Process node,	Traverse left, traverse right
Traverse left, Traverse right	Traverse right	Process node

Huffman code

Prefix code s.t. fewer bits are used for more frequent letters, optimal encoding

```
1. Create n trees, each having one node representing one letter
   Each tree has a weight, storing total frequencies of all symbols in leaves
2. Repeat until we got only one tree:
       Pick 2 trees T1, T2 with the smallest weights
       Create a new tree T, with T1 and T2 as left and right subtrees resp.
       Weight of T = Weight of T1 + Weight of T2
```



6. Binary search tree

- BST property: All node in left subtree less than current, All node in right subtree greater than current
- Inorder traversal gives a sorted sequence of nodes
- Can build unique structure (topology) given the preodrer or postorder traversal
- Randomly built BST has O(logn) expected height

Search

```
Tree-search(x, k) {
    If (x == null) or (k = key(x))
        Return x;
    Else
        If (k < key(x))   Tree-search(x.left, k);
        Else              Tree-search(x.right, k);
}
```

Maximum and Minimum: O(h); O(n) in worst cast for unbalanced tree

```
Tree-minimum(x) {
    while (x.left != null)
        x = x.left;
    return x;
}
```

```
Tree-maximum(x) {
    while (x.right != null)
        x = x.right;
    return x;
}
```

Successor and Predecessor: O(h)

```
Tree-Successor(x) { // just g.t. x
    if (x.right != null)
         return Tree-minimum(x.right);
    y = x.p;

    // climb up until x not right child
    while (y != null and x == y.right) {
        x = y;
        y = y.p;
    }
    return y;
}
```

```
Tree-Predecessor(x) { // just g.t. x
    if (x.left != null)
         return Tree-maximum(x.left);
    y = x.p;

    // climb up until x not left child
    while (y != null and x == y.left) {
        x = y;
        y = y.p;
    }
    return y;
}
```

Insertion

```
Tree-insert(T, x) { // T is root
    y = T;
    z = null;
    while (y != null) {
        z = y;
        if (x.key < y.key) y = y.left;
        else y = y.right;
    }
    x.p = z;
    if (x.key < z.key) z.left = x;
    else z.right = x;
}
```

Deletion

```
Delete(T, x) {
    z = Tree-Successor(x);
    x.element = z.element;   // copy successor over to x
    if z has no child        // remove successor
        delete z, remove x.p linkage
    else   // z has 1 child
        delete z, link z.p to child of z
```

K-th smallest

```
int kthSmallest (TreeNode* root, int k) {
     TreeNode *p = root;
     Stack stack;

     // goto far left of the tree
     while (p) {
          stack.push(p);
          p = p -> left;
     }

     // index of the node to be visited
     int cnt = 1;

     while (!stack.empty() && cnt <= k) {
          // assign p with top of stack
          p = stack.top();
          stack.pop();
          ++cnt;
          TreeNode * q = p -> right;
          while (q) {
               stack.push(q);
               q = q -> left;
} } }
```

Average time complexity of binary search

Number of cases requiring exactly i comparison $= 2^{i-1}$

$$T(n) = \sum_{i=1}^{k} \frac{i}{n} * 2^{i-1} \Rightarrow T(n) = \frac{1}{n}\sum_{i=1}^{k} i * 2^{i-1}$$

$$= \frac{1}{n}\left((1+2+4+\cdots+2^{k-1}) + (2+4+\cdots+2^{k-1}) + (4+\cdots+2^{k-1}) \ldots + (2^{k-1})\right)$$

$$= \frac{1}{n}\left((2^k-1)+(2^k-2)+(2^k-4)+\cdots+(2^k-2^{k-1})\right)$$

$$= \frac{1}{2^k}(k2^k - 2^k + 1) = k - 1 + \frac{1}{2^k} = O(k) = O(logn)$$

Prove DFS complexity

Let $D(n)$ be the time complexity of travelling T with n nodes.

$$D(1) = c$$

$$D(n) = D(n_L) + D(n_M) + D(n_R) + c \quad for\ n \geq 2$$

Induction: Assume when $n-1$, $D(n-1) = (n-1) * c$

$$D(n) = D(n_L) + D(n_M) + D(n_R) + c$$

$$D(n) = c(n_L) + c(n_M) + c(n_R) + c$$

$$D(n) = (n-1) * c + c = n * c$$

$$D(n) = O(n)$$

BST True-False

- If node u has two children, its predecessor has no left child (False)
- If node u has two children, its successor has no right child (False)
- If node u has two children, its predecessor has no right child (True) (rightmost in left subtree)
- If node u has one child, its successor has no left child (False)
- If node u has two children, successor of u's successor must be in u's right subtree (False)
- The operation of deletion in BST is "commutative" (False) (e.g. get successor vs del single child)

7. AVL tree

- BST s.t. for every node, the difference between heights of left and right subtrees is at most 1
  - Height of a null tree is defined as -1 (?)
- Balance factor: Height of left tree – Height of right tree

Insertion

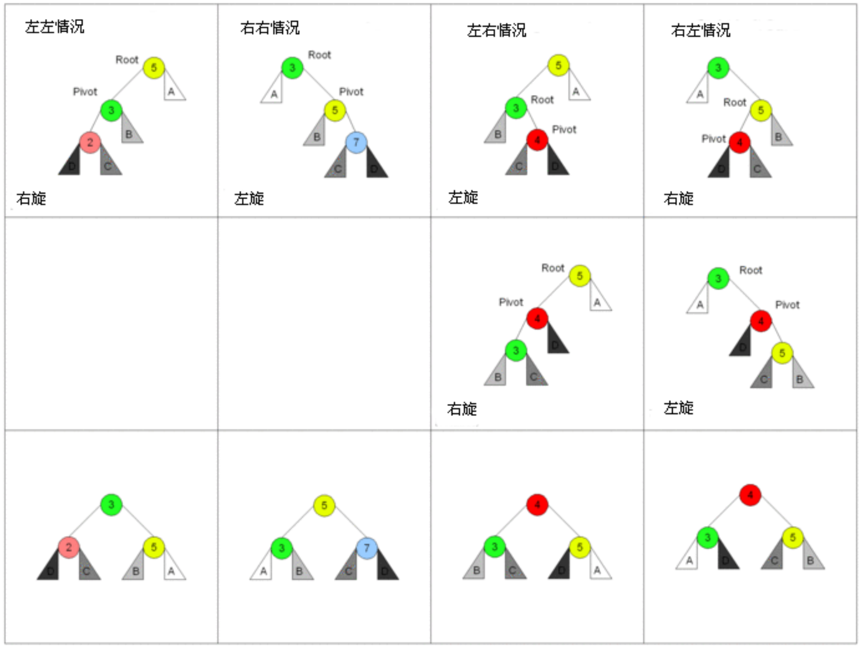
```
Insert as in BST
Go up the root along the path from inserted node
  For each node, update the value of b, rotate if node violates AVL property:

1. After insertion, the left subtree of the unbalanced node is too tall
  a. the new node is added to the left subtree of left child -> right rotate
  b. the new node is added to the right subtree of left child -> left-right rotate

2. After insertion, the right subtree of the unbalanced node is too tall
  a. the new node is added to the right subtree of right child -> left rotate
  b. the new node is added to the left subtree of right child -> right-left rotate
```

Deletion

```
Delete as in BST
Go up the root along the path from parent of deleted node
  For each node, update the value of b, rotate if node violates AVL property
```

Rotation cases

Root 是失去平衡的樹的根節點，Pivot 是旋轉后重新平衡的樹的根節點。



Prove M-AVL height

Modified-AVL tree: for every node, the height of its left subtree and right subtree differ by at most 2. Prove/disprove that the height of a modified-AVL tree of n nodes is bounded by $O(logn)$.

Proof:
Lemma 1: If T is a modified-AVL tree, then T_L and T_R are both modified-AVL trees.
Lemma 2: Since T is a modified-AVL tree, if the height of T is h, then

a. The heights of T_L and T_R are both equal to $h-1$
b. One of them is $h-1$ and the other is $h-2$
c. One of them is $h-1$ and the other is $h-3$

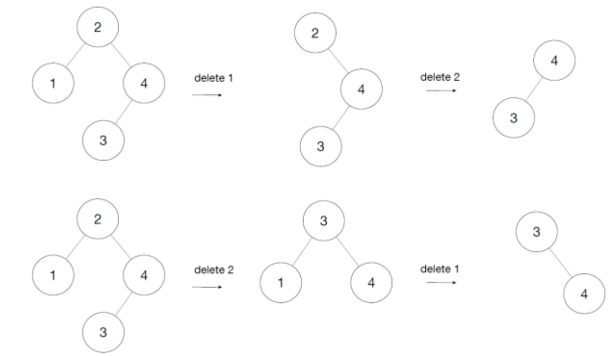
Induction: Let h be the height of T.
By (2), WLOG, let the height of T_L be $h-1$ and the height of T_R be <u>at least</u> $h-3$
By (1), T_L and T_R are both modified-AVL trees.
By the induction hypothesis, the number of nodes in T_L is $F(h-1)$ while the number of nodes in T_R is at least $F(h-3)$. So the number of node in T is at least $F(h) = F(h-1) + F(h-3)$

$$n \geq F(h) = F(h-1) + F(h-3)$$
$$> 2 * F(h-3) > 2^2 F(h-6) > \cdots > 2^{h/3} F(0) = 2^{h/3}$$
$$n > 2^{h/3} \Rightarrow logn > \frac{log2}{3} h \Rightarrow h < 3\ logn \Rightarrow h = O(logn)$$

AVL True-False

- It is not possible to add a new node to an AVL tree s.t. the parent and grandparent of the new node are balanced, but the parent of the grandparent is not balanced (False)
- Assume the balance factor of a node x in an AVL tree is 1 (left subtree taller). Let v be the right child of x. After deleting a node in the subtree rooted at x, the balance factor of v changes from -1 to 0, x should still follow the AVL tree property and the balance factor of x is changed from 1 to 2 (False)
- A constant number of rotations is sufficient to re-balance an AVL tree in the worst case when we add a node to it (True) (at most 2 rotations))
- A constant number of rotations is sufficient to re-balance an AVL tree in the worst case when we delete a node from it (False) (worst case: delete leaf node, need logn rotations to rotate all parents)

Counterexample of commutative BST deletion



8. Sorting by comparisons

Sorting Source code

```
void bubble_sort() { // O(n^2)
     for (int i = n - 1; i >= 0; i--) {
          for (int j = 0; j < i; j++) {
               if (a[j] > a[j + 1]) swap(a[j], a[j + 1]);
} } }

void insertion_sort() { // online, O(n^2)
     for (int i = 1; i < n; i++) {
          for (int j = i; j >= 1; j--) {
               if (a[j - 1] > a[j]) swap(a[j - 1], a[j]);
} } }

void selection_sort() { // O(n^2)
     for (int i = 0; i < n - 1; i++) {
          int pos = i; // position of minimum element
          for (int j = i + 1; j < n; j++) {
               if (a[j] < a[pos]) pos = j;
          }
          swap(a[i], a[pos]);
} }

void merge_arrays(int l1, int r1, int l2, int r2) { // O(n)
     vector<int> ret;
     int p1 = l1, p2 = l2;
     while (p1 < r1 || p2 < r2) {
          if (p1 == r1) ret.push_back(a[p2]), p2++;
          else if (p2 == r2) ret.push_back(a[p1]), p1++;
          else if (a[p1] <= a[p2]) ret.push_back(a[p1]), p1++;
          else ret.push_back(a[p2]), p2++;
     }

     for (int i = l1; i < r2; i++) a[i] = ret[i - l1];
}

void merge_sort(int l = 0, int r = n) { // O(nlogn)
     if (l + 1 >= r) return; // single element
     int mi = (l + r) / 2;
     merge_sort(l, mi);
     merge_sort(mi, r);
     merge_arrays(l, mi, mi, r);
}

void quick_sort(int l = 0, int r = n) { // O(nlogn)
     if (l + 1 >= r) return; // single element
     int pos = partition(l, r); // T(n) = 2/n (sum(T(k) from 0…n-1)) + cn
     quick_sort(l, pos);
     quick_sort(pos + 1, r);
}
```

Sorting time complexity lower bound (decision tree)

Longest path represents the worst case for a particular algorithm (height is # of comparisons)
# of leaves in corresponding decision tree
>= # of possible permutations of n numbers = $n!$

Let h be the height of the tree. Then $2^h >= n! \Rightarrow h >= log(n!) \Rightarrow h = \Omega(nlogn)$

In-place partitioning

```
int partition(int l, int r) {
     int v = a[r - 1], pivot = l; // last element as pivot.
     for (int i = l; i < r; i++) {
          if (a[i] <= v) {
               swap(a[i], a[pivot]);
               pivot++;
          }
     }
     return pivot - 1; // a[l, pivot) stores elements fulfilling condition
}
```

Heap and Heapsort

- Max-heap property: Value of a node >= value of any of its children
- There are 2^i nodes with depth i ($i < h$) and the nodes at depth h are packed from the left

Basic operations

```
void insert(A, x) { // O(logn)
    A[size + 1] = x;
    size++;
    i = size;
    while (i > 1 && A[i] > A[i / 2]) {
        swap(A[i], A[i / 2]);
        i = i / 2;
    }
}

void pop_heap(A) {  // O(logn)
    A[1] = A[size];
    size--;
    heapify(A, 1);
}
```

```
int maximum(A, x) { // O(1)
    return A[1];
}


heapsort(A){ // O(nlogn), in-place
    build_max_heap(A);
    for (int i = 1; i <= n - 1; i++) {
        temp = maximum(A);
        pop_heap(A);
        A[size] = temp;
        size--;
    }
}
```

Bottom-up approach for building heap (offline)

```
void heapify(A, cur) { // push node A[cur] down, make A[cur]'s subtree be a heap
    nxt = cur; // swapping target
    if (cur > n / 2) return; // A[pos] has no child
    left = 2 * cur, right = 2 * cur + 1;
    if (A[left] < A[cur]) nxt = left; // left child should be at root
    if (right <= n and A[right] < A[nxt]) nxt = right; // right should be root
    if (nxt != cur) {
        swap(nxt, cur);
        heapify(A, nxt);
    }
}

void build_heap(A) { // n is number of elements, A is 1 based
    for (int i = n / 2; i >= 1; i--) { // start from lower non-leaf elements
        heapify(A, i);
    }
}
```

Time complexity for build_heap:

$$O\left(\sum_{i=1}^{h} i\left(2^{h-i}\right)\right) = O\left(2^h \sum_{i=1}^{h} \frac{i}{2^i}\right) = O(2^h) = O(n)$$

8. Sorting in linear time

Counting sort: O(n+k)

From the cumulative frequency table, find where each element should be in the output

```
for (int i = 1; i <= n; i++)
    C[A[i]]++;

for (int i = 1; i <= k; i++)
    C[i] = C[i] + C[i - 1];

for (int i = n; i >= 1; i--) {
    B[C[A[i]]] = A[i];
    C[A[i]]--;
}
```

Radix sort: O(d(n+k)), if d and k are constants, O(n)

Time complexity: Assume that each digit have k different values, perform d counting sorts

```
radix_sort(A, d) {
    for (int i = 1; i <= d; i++) // start from least significant (rightmost) digit
        counting_sort(A on digit i) // can be replaced by any stable sort
}
```

Bucket sort: O(n) when m=#buckets=O(n)

Idea: Divide [0, 1) into m equal-sized subintervals (buckets), distribute the numbers into their respective buckets, sort numbers in each bucket separately (e.g. in $O(n^2)$)

```
bucket_sort(A) {
    for (int i = 1; i <= n; i++)
        insert A[i] into bucket B[floor(m * A[i])]

    for (int i = 0; i >= m - 1; i--)
        insertion_sort(B[i])

    concatenate(B[0..m-1])
}
```

Time complexity: $T(n) = O(n) + \sum_{j=0}^{m-1} O(n_j^2)$

- If m = O(n), then n_j is probably a constant
- Assumption: numbers are uniformly distributed over the interval [0, 1)

Stability of sorts

The original order is preserved among elements with the same sorting key

```
Stable:
  Insertion
  Bubble
  Counting
  Radix
  Merge

Unstable:
  Selection
  Shell
  Quick
```

Tree sort

Counting sort on map, then inorder traversal

Q: Given n numbers s.t. the #distinct = 2logn, devise a comparison-based algorithm to sort these numbers in O(nloglogn) time?

A:

- Use an AVL tree to store the array. For each node x, store T_x to record how many numbers in this array that has a value equal to x.
- Insert numbers into the AVL tree one by one
  If the number already exists in the AVL tree, then set T_x =T_x + 1
- After building the AVL tree, use in-order traversal to output the result.
  For each node x, we output it T_x times when we visit it

Q: Why does the $\Omega(nlogn)$ lower bound not apply?

A: There are less than $n!$ possible permutations when the values are not distinct