Divide and Conquer

Lecture 2

Timothy Kim
GWU CSCI 6212

Quiz

- Prove or disprove $2^{3n} = O(2^n)$

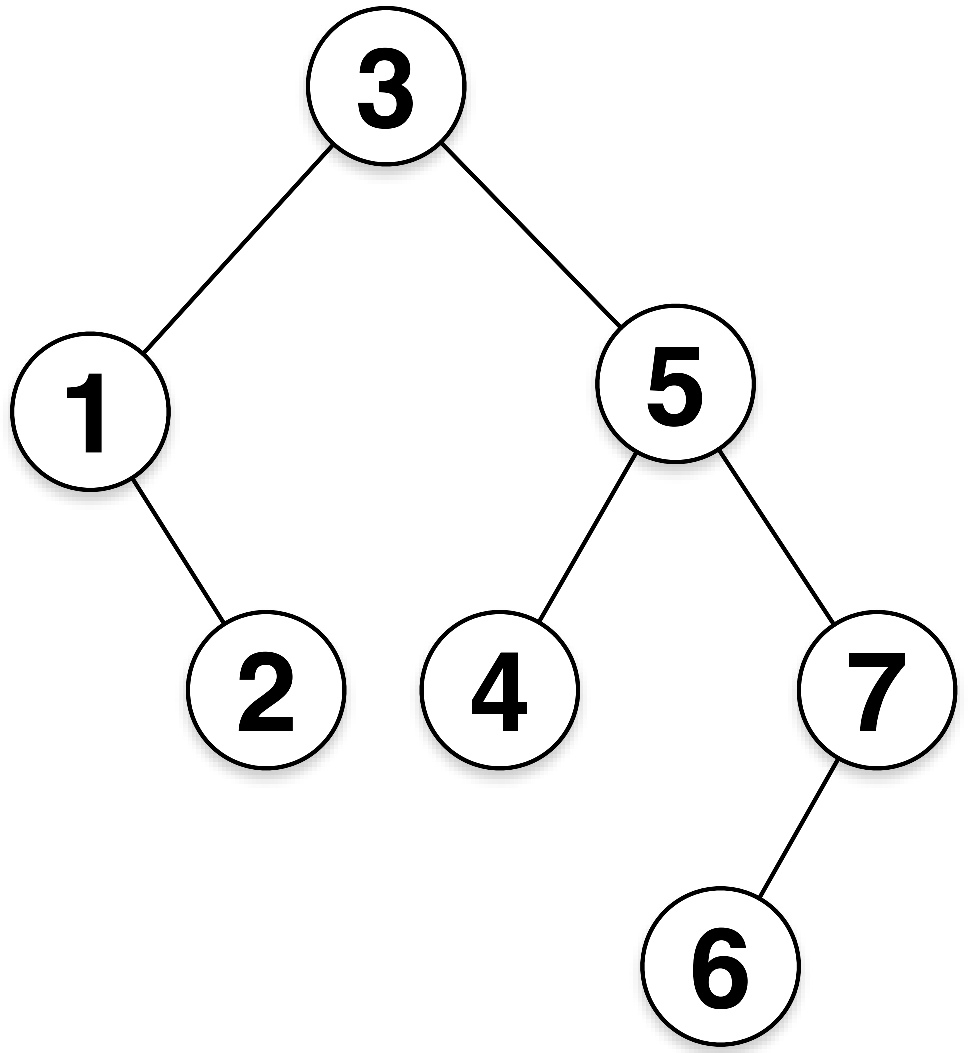
Answer

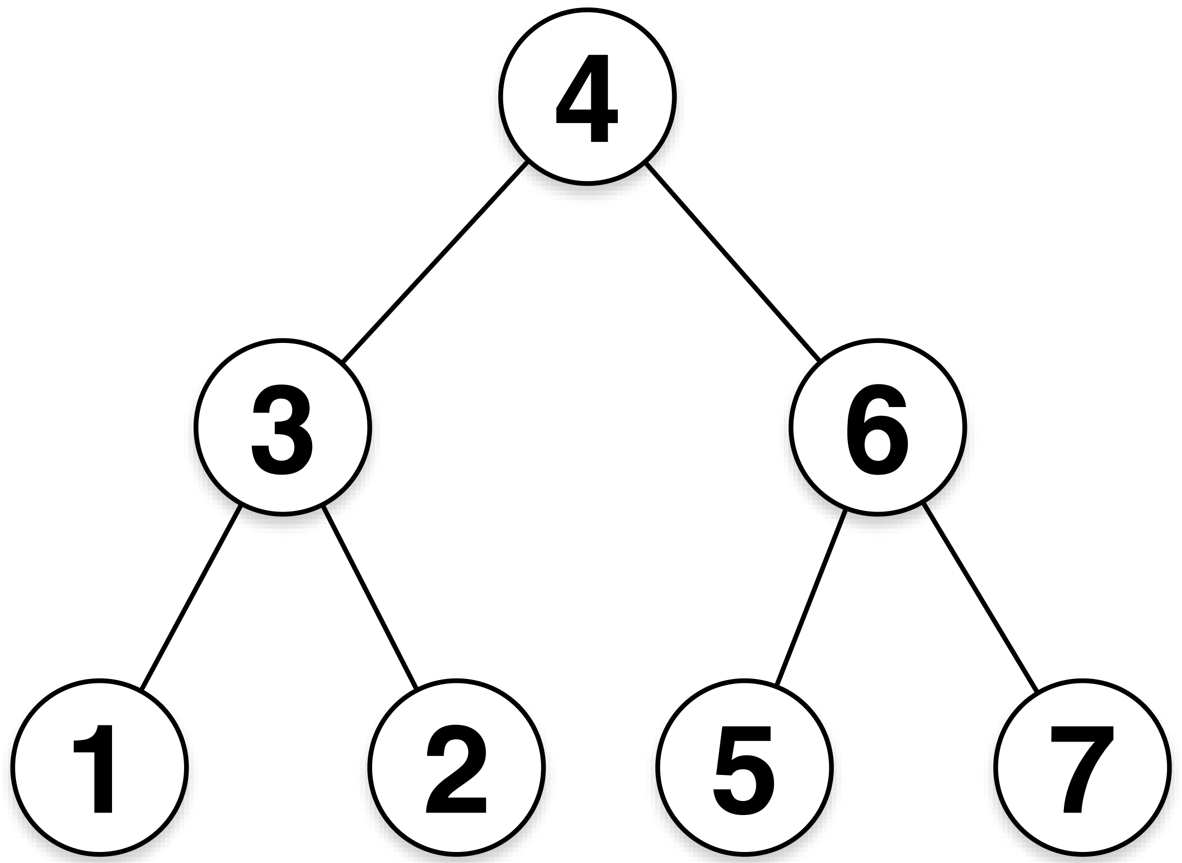
- Prove or disprove $2^{3n} = O(2^n)$
  - Assume above statement is true, then we need to find *c* and n_o such that $2^{3n} \leq c \cdot 2^n \text{ for all } n \geq n_o$
  - Note: $$\frac{2^{3n}}{2^n} \leq \frac{c \cdot 2^n}{2^n}$$ $$2^{2n} \leq c$$
  - *n* cannot be bounded by *c*, which is a contradiction
  - Thus $2^{3n} \neq O(2^n)$

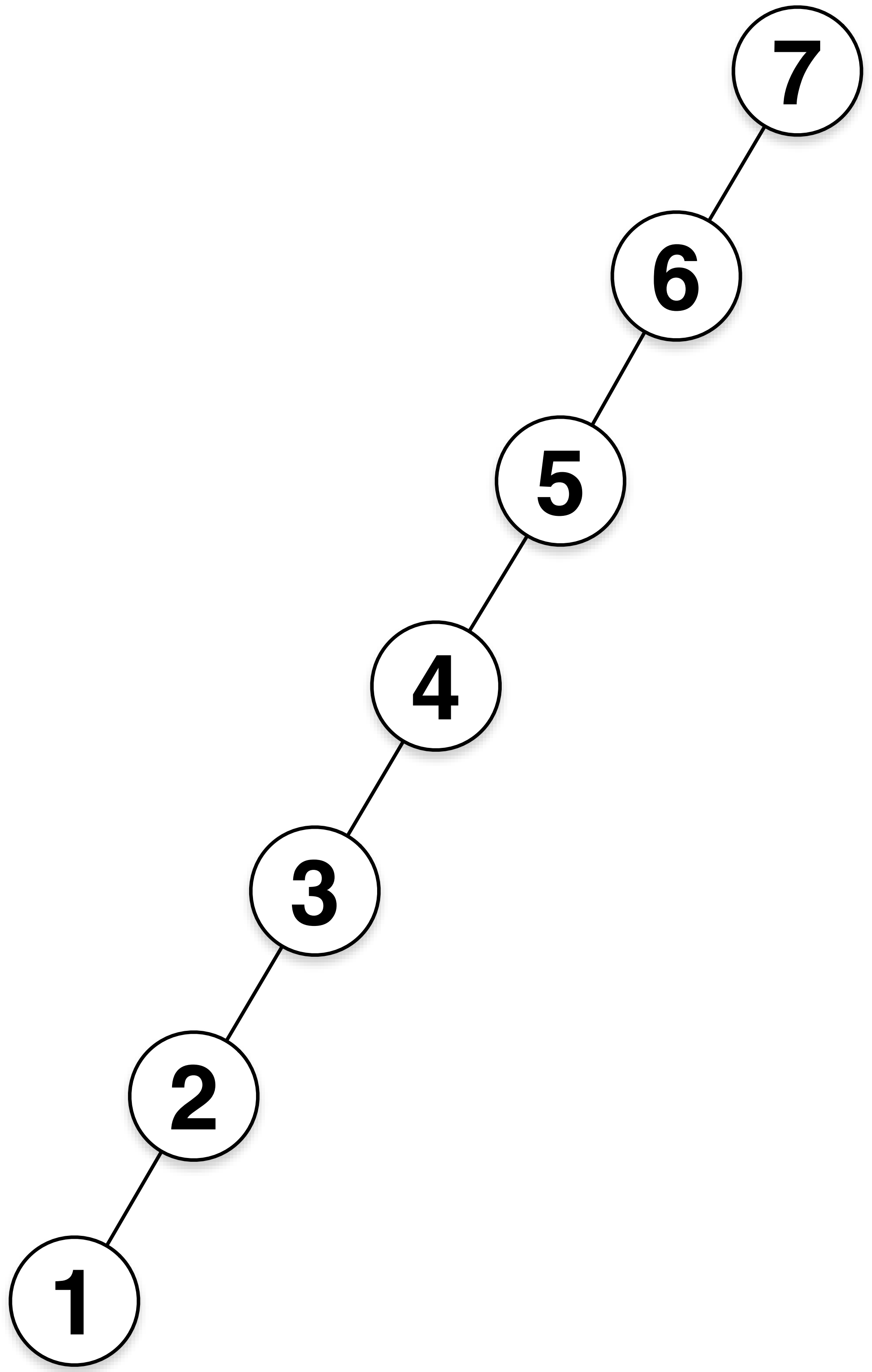
Binary Search Trees

- A binary search tree or BST is a binary tree satisfying the following
    1. Every node has a key
    2. The keys (if any) in the left subtree are smaller than the node
    3. The keys (if any) in the right subtree are greater than the node
    4. All subtrees are also binary search trees

Examples







One of the above is not a BST!

Height of a BST

- What is the maximum and the minimum height of a BST with n number of keys?
  - Max: n
  - Min: $\log_2 n$

Operations

- In-Order-Walk
- Searching
- Min and Max
- Successor and Predecessor
- Insertion and deletion

In-Order-Walk

```
InOrderWalk(node):
  InOrderWalk(node.left)
  print node.key
  InOrderWalk(node.right)
```

Running time?

A. $O(1)$
B. $O(n)$
C. $O(h)$ where h is the height of the tree
D. $O(\log n)$

Search

```
Search(node,k):
  if node == null or node.key == k:
    return k
  if k < node.key:
    return Search(node.left, k)
  else:
    return Search(node.right, k)
```

Worst case running time?

Min and Max

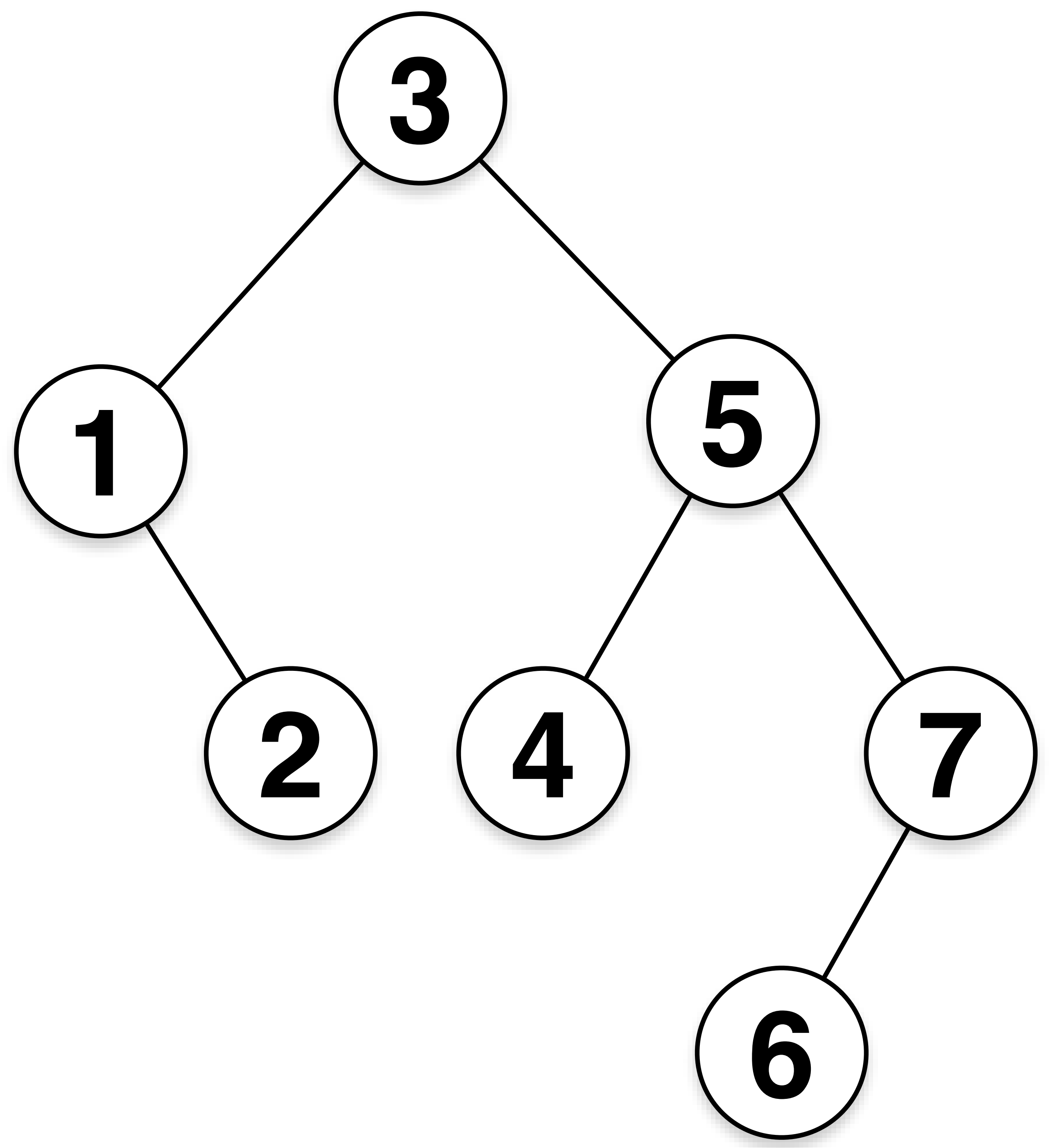
```
Min(root):
  curr = root
  while curr.left != null:
     curr = curr.left
  return cure

Max(root):
  curr = root
  while curr.right != null:
    curr = curr.right
  return curr
```

Successor and Predecessor

```
Successor(node):
  if node.right != null:
    return Min(node.right)
  succ = node.parent
  curr = node
  while succ != null and succ.left == curr:
    succ = curr.parent
    curr = succ
  return succ
```

Successor and Predecessor



Successor and Predecessor

```
Predecossor(node):
  if node.left != null:
    return Max(node.right)
  succ = node.parent
  curr = node
  while succ != null and succ.left == curr:
    succ = curr.parent
    curr = succ
  return succ
```

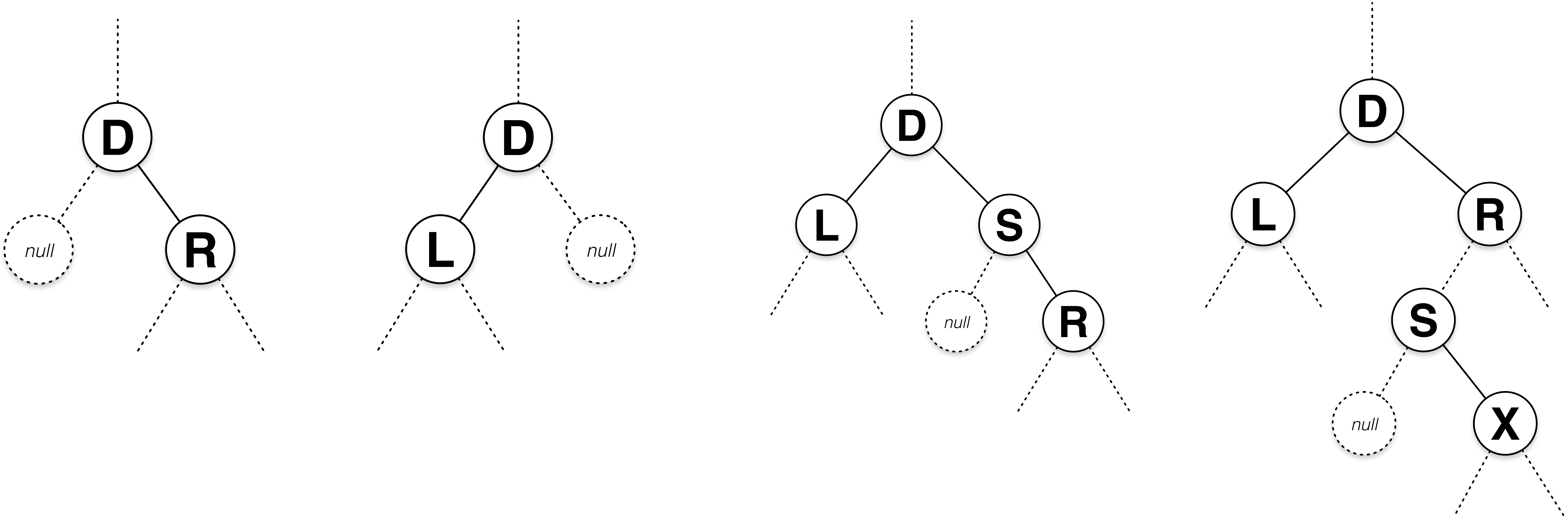
Insertion

```
Insert(Tree, key):
  // Search for key until you hit NULL
  //   and place the node in its place
  // Homework
```

Deletion

- **Case 1**: left subtree is null
- **Case 2**: right subtree is null, left subtree is not null
- **Case 3**: both subtree exists

Deletion



Divide and Conquer

- General Method:

  1. Partition the problem into smaller parts of the same type of the original problem.

  2. Solve the smaller problem

  3. Combine the solutions of the smaller problem into a solution for the whole

Basic Structure

```
DivideAndConquer(S):
    if input size is small enough:
        return Solve(S) //brute force
    else:
        S1, S2, ... = Divide(S)
        r1 = DivideAndConquer(S1)
        r2 = DivideAndConquer(S2)
        ...
        return Conquer(r1, r2, ... )
```

Example

- Recursive MinMax
- Merge Sort
- Quick Sort
- Matrix Multiplication

Recursive MinMax

```
MaxMin(S_n):
  if |S| = 2:
    return Max(S[1], S[2]), Min(S[1], S[2])
  else:
    S_1 = S[1] ... S[n/2]
    S_2 = S[n/2 + 1] ... S[n]
    max_1, min_1 = MaxMin(S_1)
    max_2, min_2 = MaxMin(S_2)
    return Max(max_1, max_2), Min(min_1, min_2)
```

Recursive MinMax Analysis

- Running Time:

$$T(n) = 2T(\frac{n}{2}) + 5 = \frac{3}{2}n - 2 = O(n)$$

Merge Sort

```
MergeSort(A):                                   T(n)
  n = |A|                                       1
  if (n < 2):
    return A
  S1 = MergeSort(A[1 ... n/2])                  T(n/2)
  S2 = MergeSort(A[n/2+1 ... n])                T(n/2)
  return Merge(S1, S2)                          5n + 3

Merge(A,B):                                     5n + 3
  C = New Empty Array of size |A| + |B|         1
  i = 1, j = 1, k = 1                           2
  while (k ≤ |C|):                              5n
    if A[i] < B[j]:
      C[k] = A[i]; i++
    else:
      C[k] = B[j]; j++
    k++
  return C
```

Merge Sort Analysis

- Recurrence Relation:

$$T(n) = 2T(\frac{n}{2}) + 5n + 4$$

- Since:

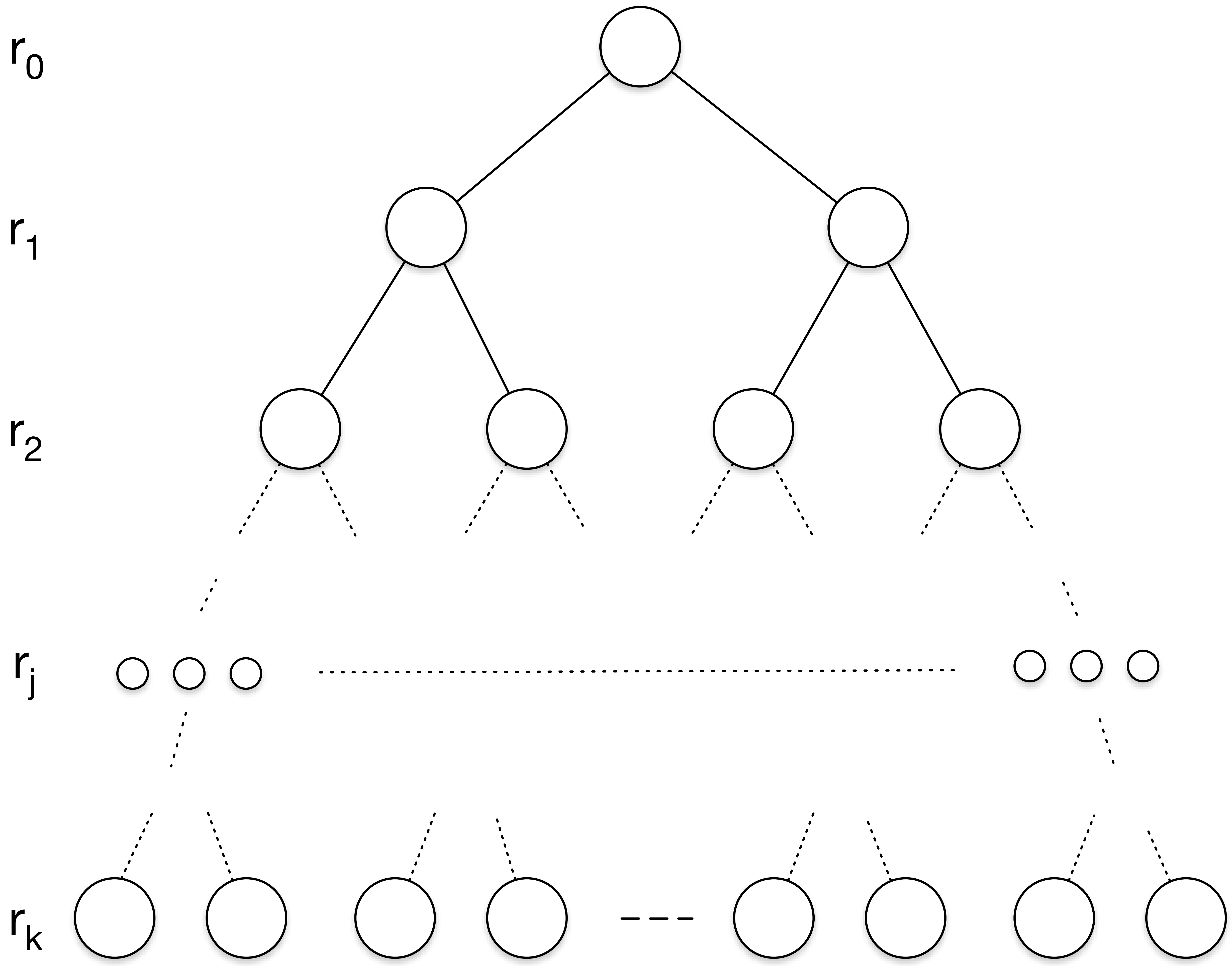
$$5n + 4 \leq 6n$$

- Therefore:

$$T(n) \leq 2T(\frac{n}{2}) + 6n$$

Recurrence Tree

- Each circle represents call to the **`Merge()`** subroutine
- How high is the tree? k = ?
- How many calls to **`Merge()`** are made at level r_k?
- How big is the input size at r_1?
- How big is the input size at r_k?
- At level r_j, how many calls to **`Merge()`** is made?
- At level r_j, how big is the input size of **`Merge()`**?



Merge Sort Analysis

- Given a level *j*
  - # of subproblems = 2^j
  - size of input = $\dfrac{n}{2^j}$
- Running time each node = $6\dfrac{n}{2^j}$

- Running time at r_j

$$\leq 2^j \cdot 6\frac{n}{2^j} = \cancel{2^j} \cdot 6\frac{n}{\cancel{2^j}} = 6n$$

- Running time at all levels

$$= \sum_{j=0}^{\log_2 n} 6n$$
$$= 6n \cdot (\log_2 n + 1)$$
$$= O(n \cdot \log n)$$

Master Method

- If

$$T(n) \leq a \cdot T(\frac{n}{b}) + O(n^d)$$

- Then

$$\begin{aligned}
T(n) &= O(n^d \log n) && \text{if } a = b^d \\
T(n) &= O(n^d) && \text{if } a < b^d \\
T(n) &= O(n^{\log_b a}) && \text{if } a > b^d
\end{aligned}$$

Merge Sort Application

- Merge Sort Running Time
$$T(n) \leq 2T(\tfrac{n}{2}) + 6n$$
$$\leq 2T(\tfrac{n}{2}) + O(n)$$

- Master Method
$$T(n) \leq a \cdot T(\frac{n}{b}) + O(n^d)$$

$$a = 2$$
$$b = 2$$
$$d = 1$$

$$a = b^d \therefore T(n) = O(n \log n)$$

Example

- Given the following recurrence form find the big O equivalent:
  - $T(n) = 4\,T(n/3) + n^2 = O(???)$

- If

$$T(n) \leq a \cdot T(\frac{n}{b}) + O(n^d)$$

- Then

$$\begin{array}{ll} T(n) = O(n^d \log n) & \text{if } a = b^d \\ T(n) = O(n^d) & \text{if } a < b^d \\ T(n) = O(n^{\log_b a}) & \text{if } a > b^d \end{array}$$

Master Method Proof

- If

$$T(n) \leq a \cdot T(\frac{n}{b}) + O(n^d)$$

- Then

$$\begin{aligned} T(n) &= O(n^d \log n) && \text{if } a = b^d \\ T(n) &= O(n^d) && \text{if } a < b^d \\ T(n) &= O(n^{\log_b a}) && \text{if } a > b^d \end{aligned}$$

Assumption to make our proof simpler

- Assume

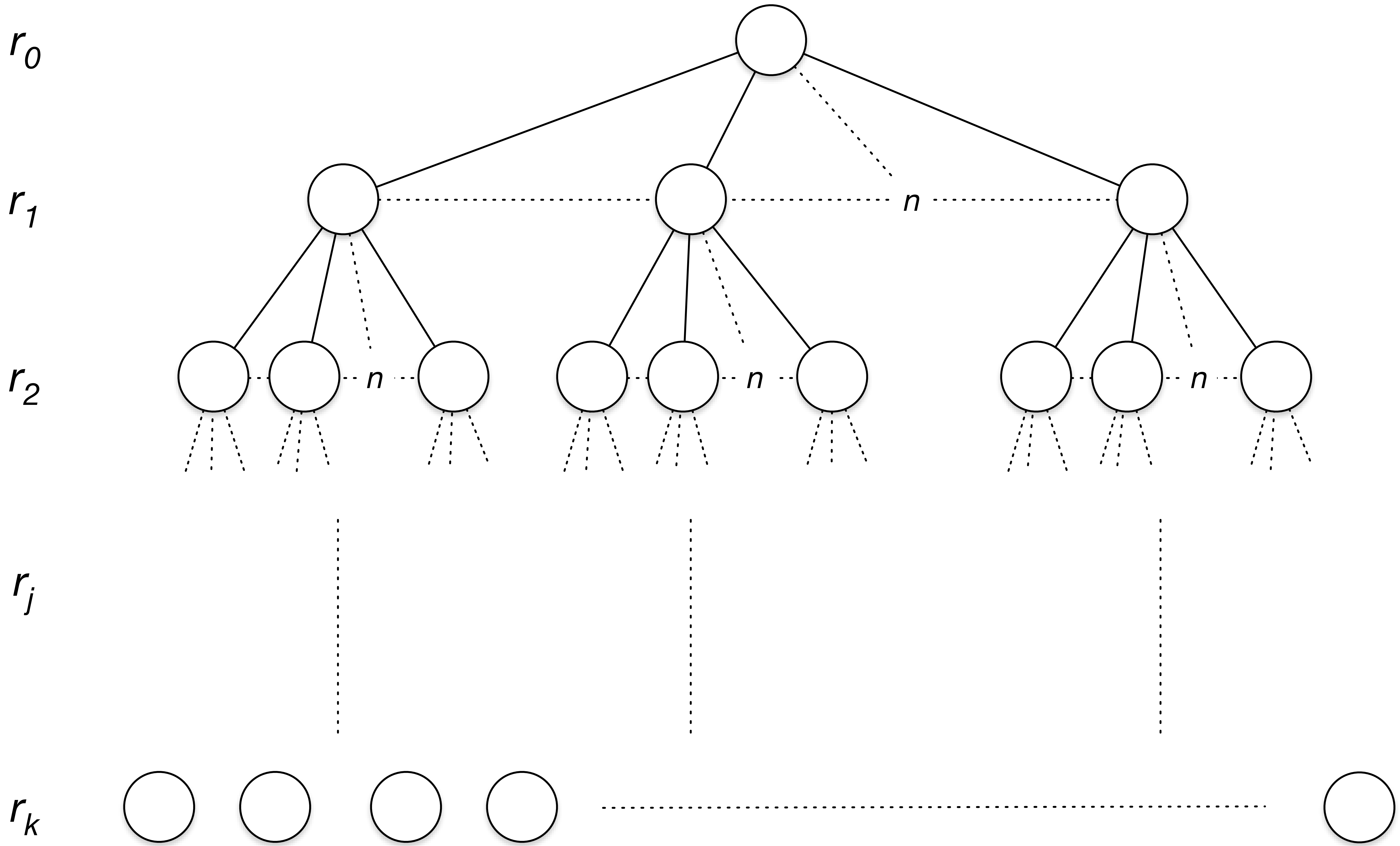
$$T(1) \leq c$$

$$T(n) \leq aT(\tfrac{n}{b}) + cn^d$$

$$n = b^k$$

High Level Idea

- How deep is the tree?
- At any level how many sub problems are there?
- How big is the input size for each sub problem?



Running time at a single level

- Given a level *j*
  - # of subproblems = a^j
  - size of input = $\frac{n}{b^j}$
- Running time each node:
  - Recall $T(n) \leq aT(\frac{n}{b}) + cn^d$
  - Thus running time $\leq c(\frac{n}{b^j})^d$
- Running time at r_j

$$\leq a^j \cdot c\left(\frac{n}{b^j}\right)^d = cn^d \cdot \left(\frac{a}{b^d}\right)^j$$

- Running time at all levels

$j = 1, 2, ..., \log_b n$

$$\leq cn^d \cdot \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j$$

What does that mean?

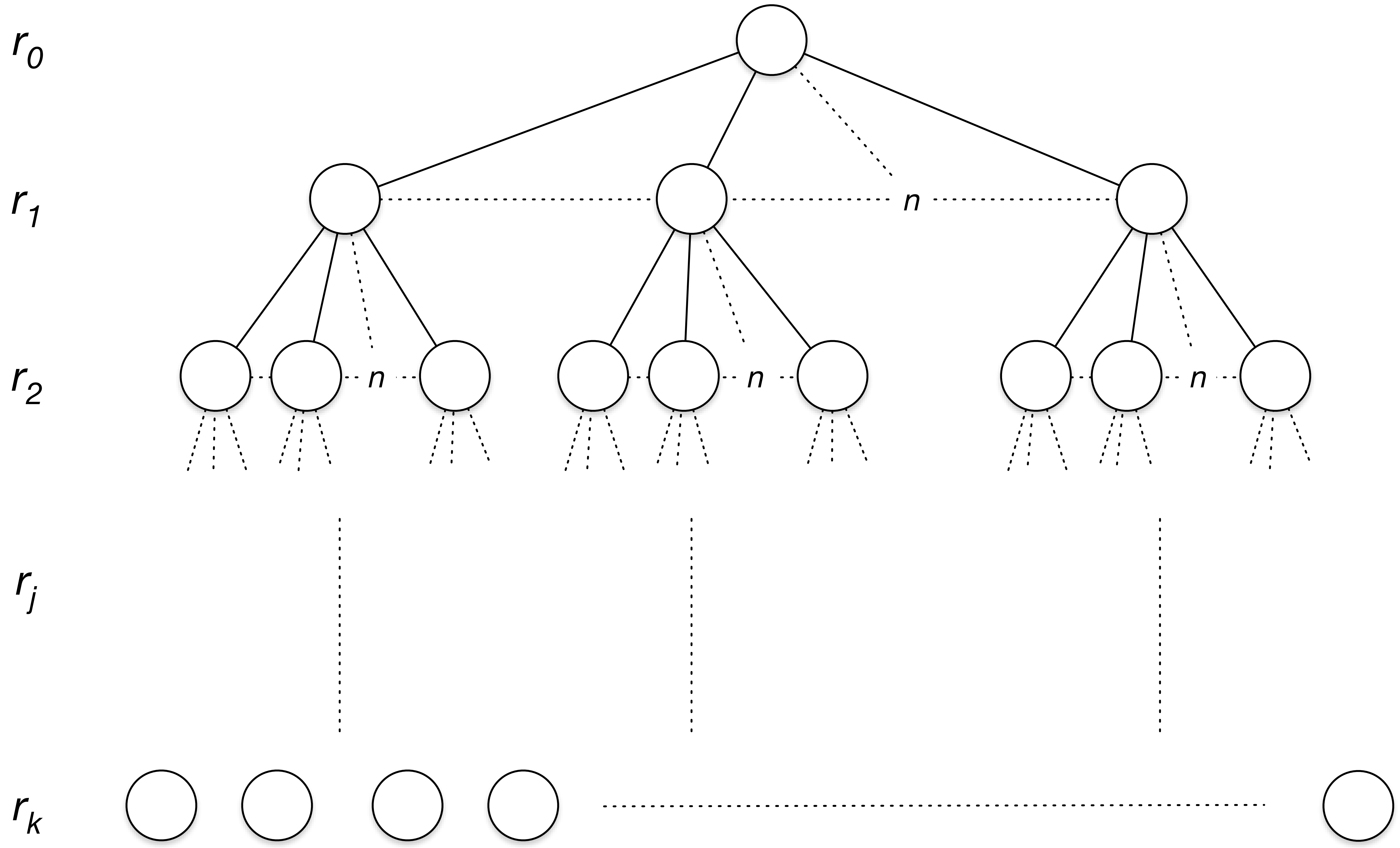
- What does *a* represent?
- What does *b* represent?
- What does b^d represent?

$$\leq cn^d \cdot \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j$$

What does THAT mean?

- Three cases:

$$a < b^d$$
$$a = b^d$$
$$a > b^d$$



Intuition

- If a = b^d then amount of the work each level is the same
- If a < b^d then less work as you go down the tree, most of the work is done at the root
- If a > b^d then more work as you go down the tree, most of the work is done the leaf

$$
\begin{aligned}
T(n) &= O(n^d \log n) && \text{if } a = b^d \\
T(n) &= O(n^d) && \text{if } a < b^d \\
T(n) &= O(n^{\log_b a}) && \text{if } a > b^d
\end{aligned}
$$

Master Method Proof

- If

$$T(n) \leq a \cdot T(\frac{n}{b}) + O(n^d)$$

- Then

$$\begin{aligned} T(n) &= O(n^d \log n) && \text{if } a = b^d \\ T(n) &= O(n^d) && \text{if } a < b^d \\ T(n) &= O(n^{\log_b a}) && \text{if } a > b^d \end{aligned}$$

Proof Case 1

- Case 1 $a = b^d$

$$T(n) = O(n^d \cdot \log n) = cn^d \cdot \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j$$

- Note: $\frac{a}{b^d} = 1$

$$cn^d \cdot \sum_{j=0}^{\log_b n} 1^j = cn^d \cdot \sum_{j=0}^{\log_b n} 1$$

$$= cn^d \cdot (\log_b n + 1)$$

$$= O(n^d \cdot \log n)$$

Master Method Proof

- If

$$T(n) \le a \cdot T(\frac{n}{b}) + O(n^d)$$

- Then

$$\begin{aligned} T(n) &= O(n^d \log n) && \text{if } a = b^d \\ T(n) &= O(n^d) && \text{if } a < b^d \\ T(n) &= O(n^{\log_b a}) && \text{if } a > b^d \end{aligned}$$

Lemma

- Given the following geometric sum:

$r \neq 1$

$$1 + r + r^2 + ... + r^k = \frac{r^{k+1} - 1}{r - 1}$$

- Note:
  - If $r < 1 \rightarrow \frac{r^{k+1} - 1}{r - 1} \leq \frac{1}{1 - r}$ (bounded by some constant)

    (dominated by the first term)
  - If $r > 1 \rightarrow \frac{r^{k+1} - 1}{r - 1} \leq r^k \cdot (1 + \frac{1}{1 - r})$

    (dominated by the last term)

Proof Case 2

- Case 2 $a < b^d$ $T(n) = O(n^d) = cn^d \cdot \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j$
- Note: $\frac{a}{b^d} < 1$

$$cn^d \cdot \sum_{j=0}^{\log_b n} r^j | r < 1$$

$$= cn^d \cdot c' = O(n^d)$$

Proof Case 3

- Case 3 $a > b^d$ $\qquad T(n) = O(n^{\log_b a}) = cn^d \cdot \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j$
- Note: $r = \dfrac{a}{b^d} > 1$

$$cn^d \cdot \sum_{j=0}^{\log_b n} r^j | r > 1$$

$$\leq cn^d \cdot \left(\frac{a}{b^d}\right)^{\log_b n} \cdot c' \qquad = O(n^d \cdot \left(\tfrac{a}{b^d}\right)^{\log_b n})$$

Proof Case 3

$$= O\left(n^d \cdot \left(\frac{a}{b^d}\right)^{\log_b n}\right)$$
$$= O\left(n^d \cdot a^{\log_b n} \cdot (b^d)^{-\log_b n}\right)$$
$$= O\left(n^d \cdot a^{\log_b n} \cdot (b^{\log_b n})^{-d}\right)$$
$$= O\left(n^d \cdot a^{\log_b n} \cdot n^{-d}\right)$$
$$= O\left(n^d \cdot \frac{a^{\log_b n}}{n^d}\right)$$
$$= O\left(a^{\log_b n}\right)$$
$$= O\left(n^{\log_b a}\right)$$

$$= O(a^{\log_b n}) = O(\#\text{ of leaves})$$

Master Case Implication

- If

$$T(n) \leq a \cdot T(\frac{n}{b}) + O(n^d)$$

- Then

$$\begin{aligned}
T(n) &= O(n^d \log n) && \text{if } a = b^d \\
T(n) &= O(n^d) && \text{if } a < b^d \\
T(n) &= O(n^{\log_b a}) && \text{if } a > b^d
\end{aligned}$$

Example Question

- Given $T(n) = 2T(n/4) + n^3$ find the asymptotic running time of the recurrence.