Agenda:

- Quicksort and Analysis (CLRS ch 7.1-2, 7.4)
- Sorting Lower Bounds (CLRS ch 8.1)
- Binary Search Trees (BST): a review (CLRS ch 12.1-3)
- Balanced Binary Search Trees: AVL Trees (See the note below)
- Hash Tables (CLRS ch 11.1-3) (may be postponed to the following Monday)

- **Note:** AVL trees is one of the approaches to balanced BSTs. CLRS leaves it as a problem on p333. AVL trees use the same *tree rotation* method given in CLRS ch 13.2. Good exaplanations of AVL trees can be found on the Internet too, e.g., `https://www.geeksforgeeks.org/avl-tree-set-1-insertion/`

QuickSort: Another sorting meets divide-and-conquer

- The ideas:
  - Pick one key ($pivot$), compare it to all others.
  - Rearrange A to be: [elements $\leq pivot$, $pivot$, elements $> pivot$]
  - Recursively sort subarrays before and after the $pivot$.
- Pseudocode:

```
procedure QuickSort$(A, p, r)$
    $**$ sorts the subarray $A[p, ..., r]$
    if $(p < r)$ then
        $q \leftarrow$ Partition$(A, p, r)$
            $**$ Partition returns $q$ such that
            $**$ (i) $A[q] = pivot$,
            $**$ (ii) All elements $\leq pivot$ appear in $A[p, ..., q-1]$
            $**$ (iii) All elements $> pivot$ appear in $A[q+1, ..., r]$
        QuickSort$(A, p, q-1)$
        QuickSort$(A, q+1, r)$
```

3 2 1 5 6 4

↑

3 2 1 4 6 5

- How will you prove QuickSort's correctness?
- By induction on $n = \#$ elements in $A = r - p + 1$
  - Your base case needs to be both $n = 1$ and $n = 0$. (Why?)
  - Induction step is easy if we know Partition() is correct

6 2 4 7 1 3 $\boxed{5}$

2 6 4 7 1 3 5

2 4 6 7 1 3 5

2 4 1 7 6 3 5

2 4 1 3 6 7 5

2 4 1 3 5 7 6

Partition(A, p, r):

- procedure Partition(A, p, r)

```
    ** last element, $A[r]$, is the $pivot$ key picked of the partition
    $pivot \leftarrow A[r]$
    $i \leftarrow p - 1$   ** $i$ is the location of the last element known to be $\leq pivot$
    for ($j$ from $p$ to $r - 1$) do
        if ($A[j] \leq pivot$) then
            $i \leftarrow i + 1$
            exchange $A[i] \leftrightarrow A[j]$
    exchange $A[i + 1] \leftrightarrow A[r]$
    return $i + 1$
```

- Example: $A[1..8] = \{3, 1, 7, 6, 4, 8, 2, 5\}$, $p = 1$, $r = 8$, $pivot = A[8] = 5$

3	1	7	6	4	8	2	5	$i = 0, j = 1$
3	1	7	6	4	8	2	5	$i = 1, j = 2$
3	1	7	6	4	8	2	5	$i = 2, j = 3$
3	1	7	6	4	8	2	5	$i = 2, j = 4$
3	1	7	6	4	8	2	5	$i = 2, j = 5$
3	1	4	6	7	8	2	5	$i = 3, j = 5$
3	1	4	6	7	8	2	5	$i = 3, j = 6$
3	1	4	6	7	8	2	5	$i = 3, j = 7$
3	1	4	2	7	8	6	5	$i = 4, j = 7$
3	1	4	2	5	8	6	7	$i = 4, j = 8$ (for-loop ended)

8 2 6 7 3 4 5 i=0

$\boxed{2}$ 8 6 7 3 4 $\overset{=}{5}$ j=1

$\boxed{2 \quad 3}$ 6 7 8 4 5 i=2

2 3 4 7 8 6 5 i=3

`Partition`(A, p, r):

- The $pivot$ happens to be $A[r]$.
- Works by traversing the keys of the array from p to $r - 1$.
- j indicates the current element we are considering.
- i is the index of the last known element which is $\leq pivot$.

- The invariant:
  - $i < j$
  - $A[p..i]$ contains keys $\leq pivot$
  - $A[(i+1)..(j-1)]$ contains keys $> pivot$
  - $A[j..(r-1)]$ contains keys yet to be compared to $pivot$
  - $A[r]$ is the $pivot$

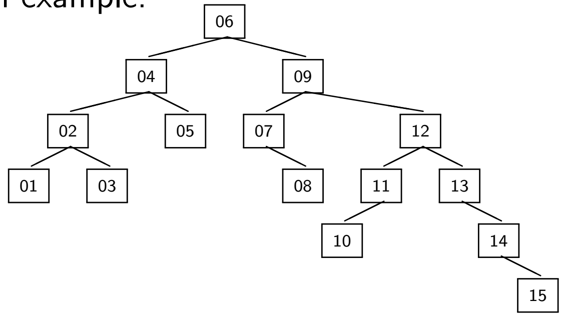
- Ideas:
  - $A[j]$ is the current key
  - If $A[j] > pivot$ — no need to change i or exchange keys as the invariant is maintained
  - If $A[j] \leq pivot$, exchange $A[j]$ with the first larger-then-pivot element ($A[i+1]$) and increment i to maintain the fact that $A[p, ..., j]$ is built from two consecutive subarrays of elements $\leq pivot$ and elements $> pivot$
  - At the end, exchange $A[r] \leftrightarrow A[i+1]$ such that:
    - $A[p..i]$ contains keys $\leq pivot$
    - $A[i+1]$ contains $pivot$
    - $A[(i+2)..r]$ contains keys $> pivot$

QuickSort Analysis

- Why we study QuickSort and its analysis:
  - very efficient, in use
  - divide-and-conquer
  - huge literature
  - a model for analysis of algorithms
  - a terrific example of the usefulness of *randomization*
- Observations:
  - (Again) key comparison is the dominant operation
  - Counting KC — *only* need to know (at each call) the rank of the split key

QuickSort recursion tree:

- root = pivot, left and right children = trees for the first and second recursive calls
- An example:



- More observations:
  - In the resulting recursion tree, at each node (all keys in left subtree) $\leq$ (key in this node) $<$ (all keys in right subtree)
  - QuickSort recursion tree $\longrightarrow$ binary search tree

QuickSort Running Time

- Like before - dominated by #KC.
- The pivot is compared with every other key: $(n-1)$ KC
- Recurrence:

$$T(n) = \begin{cases} 0, & n \leq 1 \\ T(n_1) + T(n-1-n_1) + (n-1), & n \geq 2 \end{cases}$$

  where $0 \leq n_1 \leq n-1$
- This raises the question: how do we estimate what n_1 is going to be?
- There is no single answer.

QuickSort WC running time:

- Recurrence:

$$T(n) = \begin{cases} 0, & n \leq 1 \\ T(n_1) + T(n-1-n_1) + (n-1), & n \geq 2 \end{cases}$$

- Notice that when both subarrays are non-empty, then #KC in next level is

$$(n_1 - 1) + (n - 1 - n_1 - 1) = (n-3)$$

  But when one subarray is empty then #KC in next level is $(n-2)$.

- WC recurrence:

$$T(n) = T(0) + T(n-1) + (n-1) = T(n-1) + (n-1),$$

- Solving the recurrence — Master Theorem doesn't apply

$$\begin{aligned} T(n) &= T(n-1) + (n-1) = T(n-2) + (n-2) + (n-1) \\ &= \ldots \\ &= T(1) + 1 + 2 + \ldots + (n-1) \\ &= \tfrac{(n-1)n}{2} \end{aligned}$$

  So, $T(n) \in \Theta(n^2)$

- What is a worst-case instance for QuickSort?

QuickSort Almost-WC running time:

- Recurrence:

$$T(n) = \begin{cases} 0, & n \leq 1 \\ T(n_1) + T(n-1-n_1) + (n-1), & n \geq 2 \end{cases}$$

- Let's try an almost-WC situation: At every step, we find a pivot for which $n_1 = n-2$.
- WC recurrence:

$$T(n) = T(1) + T(n-2) + (n-1) = T(n-2) + (n-1),$$

- Solving the recurrence —

$$\begin{aligned} T(n) &= T(n-2) + (n-1) = T(n-4) + (n-3) + (n-1) \\ &= \ldots \\ &= T(1) + 1 + 3 + \ldots + (n-3) + (n-1) \end{aligned}$$

- Clearly $T(n) \leq \frac{n}{2}(n-1) \in O(n^2)$, and also $T(n) \geq \frac{n}{4} \cdot \frac{n}{2} \in \Omega(n^2)$. So, $T(n) \in \Theta(n^2)$.

QuickSort BC running time:

- Recurrence:

$$T(n) = \begin{cases} 0, & n \leq 1 \\ T(n_1) + T(n-1-n_1) + (n-1), & n \geq 2 \end{cases}$$

- When both subarrays are non-empty, we are saving 1 KC ...
- Best case: each partition is a bipartition !!!
  Saving as many KC as possible every level ...
  The recursion tree is as short as possible ...
- Recurrence:

$$T(n) = 2 \times T(\frac{n-1}{2}) + (n-1)$$

- Solving the recurrence — Master Theorem $T(n) = 2T(\frac{n}{2}) + (n-1)$ solves to $\Theta(n \log n)$.
- Question: What is the best case array for the case of $\{1, 2, ..., 7\}$?

QuickSort Almost-BC running time:

- Let's assume the at each round we get an approximated bipartition. Namely, each split is $\frac{3}{4}n$ and $\frac{1}{4}n$, resulting in recurrence

$$T(n) = T(\frac{3n}{4}) + T(\frac{n}{4}) + cn$$

  where, for simplicity, we just write cn to mean linear time in n
- We may even consider a more extreme case where split is $\frac{9}{10}n$ and $\frac{1}{10}n$, resulting in recurrence

$$T(n) = T(\frac{9n}{10}) + T(\frac{n}{10}) + cn$$

- In both cases, the running time is $O(n \log n)$, and in fact $\Theta(n \log n)$ considering time $\Omega(n \log n)$ for BC.
- In fact, for any split of *constant* proportionality, the running time remains to be $\Theta(n \log n)$ (See CLPR p.175-176).

QuickSort Average Case running time:

- The recurrence for running time is:
  $$T(n) = \begin{cases} 0, & \text{if } n = 0, 1 \\ T(n_1) + T(n-1-n_1) + (n-1), & \text{if } n \geq 2 \end{cases}$$
- Average case: always ask “average over what input distribution?”
  - Here, we assume each possible input equiprobable, i.e. *uniform distribution*.
  - Key observation: equiprobable inputs imply for each key, rank among keys so far is equiprobable.
  - So, n_1 can be $0, 1, 2, \ldots, n-2, n-1$, with the same probability $\frac{1}{n}$

Solving $T(n)$:

- As $\Pr[n_1 = i] = \frac{1}{n}$ for every i, we get

$$
\begin{aligned}
T(0) &= 0,\\
T(1) &= 0,\\
T(2) &= (2-1) + \tfrac{1}{2}\left(T(0) + T(2-1-0)\right) + \tfrac{1}{2}\left(T(1) + T(2-1-1)\right),\\
T(n) &= (n-1) + \tfrac{1}{n}\left(T(0) + T(n-1)\right)\\
&\qquad\qquad + \tfrac{1}{n}\left(T(1) + T(n-2)\right)\\
&\qquad\qquad + \ldots\\
&\qquad\qquad + \tfrac{1}{n}\left(T(n-2) + T(1)\right)\\
&\qquad\qquad + \tfrac{1}{n}\left(T(n-1) + T(0)\right)\\
&= (n-1) + \tfrac{2}{n}\sum_{i=0}^{n-1} T(i).
\end{aligned}
$$

- Master Theorem does NOT apply here.
- But one can verify that $T(n) \le 2(n+1)[H(n+1) - 1]$

  (The Harmonic number $H(n) = \sum_{i=1}^{n} \frac{1}{i} = \ln n + \gamma$, where $\gamma \approx 0.577\cdots$)

Sorting Algorithms: Running Time Comparison

Alg.	BC	WC	AC
InsertionSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
MergeSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$*$
HeapSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$*$
QuickSort	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n \log n)$
Random QuickSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$*$

What is "$*$"?

QuickSort Improvement and space requirement:

- QuickSort is considered an in-place sorting algorithm:
  - extra space required at each recursive call is only constant.
  - whereas in MergeSort, at each recursive call up to $\Theta(n)$ extra space is required.
- To improve the algorithm we use a **random** pivot

Difference between a Randomized Algorithm and AC Analysis

- AC analysis means we make an assumption on the input
  - No guarantee that the assumption holds
  - Input is chosen once: on avg we might have a good running time, but once input is given our running time is determined.
- A randomized algorithm works for *any* input (WC analysis)
  - Randomness in the coins we toss (not in the input) so we control the distribution of the coin toss
  - We can always start the algorithm anew if it takes too long; or run it multiple times in parallel

Randomized QuickSort

- We invoke `RandomPartition` rather than `Partition`
- Pseudocode

```
procedure RandomPartition($A, p, r$)
    $i \leftarrow$ uniformly chosen random integer in $\{p, ..., r\}$
    exchange $A[i] \leftrightarrow A[r]$
    return Partition($A, p, r$)
```

- Q: How do we analyze the #KC of the Random QuickSort?
  Depends on which pivot we pick, we can compare any two elements. And of course, there is a chance we pick the worst-pivot (last element) in every iteration...
- A: We analyze the *expected* WC #KC (As always, proportional to Expected WC running time)
- Conclusion: the expected WC running time for random quicksort is $\Theta(n \log n)$. (We ignore the details in this course.)

Sorting lower bound:

- So far: we looked at BC runtime — for lower bounds purposes.
  - They serve as lower bounds, for specific algorithms.
  - E.g., “Even in the best case, my algorithm makes _ KC.”
  - So this is a lower bound of the form

$$\exists \text{ algoritm } A \text{ s.t. } \forall \text{ input } I, \quad \text{runtime}(A(I)) \geq ....$$

- We now give a lower bound for the problem of sorting.
  - A lower bound on *any* algorithm for sorting - even those not invented yet.
  - This is a lower bound of the form

$$\forall \text{ algoritm } A \; \exists \text{ input } I, \quad \text{runtime}(A(I)) \geq ....$$

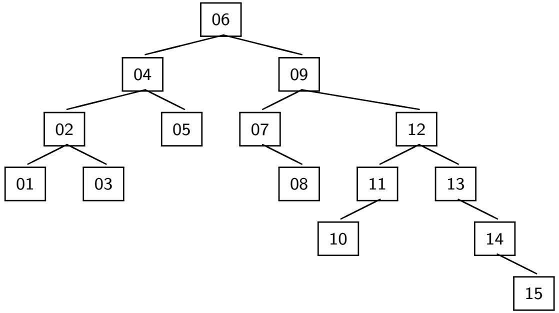
- Q: Can we derive a lower bound of the form

$$\forall \text{ algoritm } A \text{ and } \forall \text{ input } I, \quad \text{runtime}(A(I)) \geq ....?$$

- A: Not a very informative bound, since for every input I_0 we can always “massage” any algorithm into an algorithm that first checks for I_0.
  `If` (input= I_0) `return` solution(I_0) `else ...` (run original algorithm)

Two useful trees in algorithm analysis:

1. Recursion tree
   - node ⟷ recursive call
   - describes algorithm execution for one particular input by showing all calls made
   - one algorithm execution ⟷ all nodes (a tree)
   - useful in analysis: sum the numbers of operations over all nodes



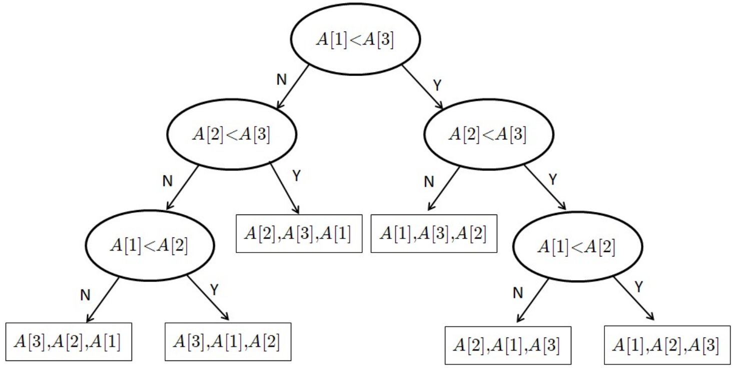
Two useful trees in algorithm analysis:

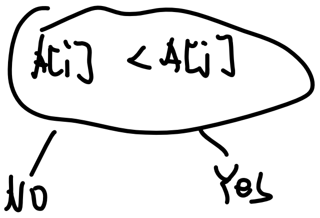
2. Decision tree

- node ⟷ algorithm decision
- describes algorithm execution for all possible inputs by showing all possible algorithm decisions
- one algorithm execution ⟷ one root-to-leaf path
- useful in analysis: sum the numbers of operations over nodes on one path

decision tree is binary & must have ≥n leaves, one for each answer

height ≥ lgn





$A[5] \le A[7] \le A[1] \le A[0] \le$

decision tree is binary and # of leaves ≥ # of possible answers = n!

height ≥ $\lg(n!)$

$\ge \lg n + \lg(n-1) + \lg(n-2) + \dots + \log(1)$

$\sum_{i=1}^{n} \lg i$

$\sum_{i=\frac{n}{2}}^{n} \lg \frac{n}{2} \ge \sum_{i=\frac{n}{2}}^{n} (\log n - 1)$

Sorting lower bound:

- Consider comparison-based sorting algorithms. These algorithms map to decision trees (nodes have exactly 2 children).
- Binary tree facts:
  - Suppose there are t leaves and k levels. Then,
  - $t \le 2^{k-1}$
  - So, $\log t \le (k-1)$
  - Equivalently, $k \ge 1 + \log t$

    — binary tree with t leaves has *at least* $(1 + \log t)$ levels
- Comparison-based sorting algorithm facts:
  - Look at its *Decision Tree*. It's a binary tree.
  - It should contain every possible output: every permutation of the positions $\{1, 2, \dots, n\}$.
  - So, it contains at least $n!$ leaves ...
  - Equivalently, it has at least $1 + \log(n!)$ levels.
  - A longest root-to-leaf path of length at least $\log(n!)$.
  - So in the worst case, the algorithm makes at least $\log(n!)$ KC, and $\log(n!) \in \Theta(n \log n)$

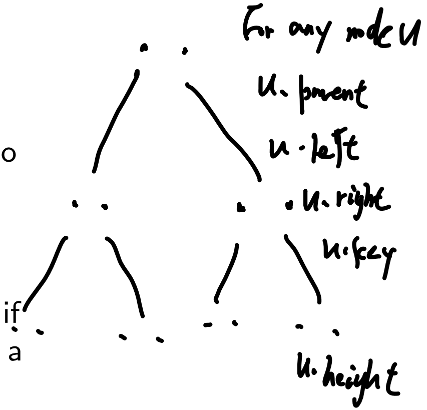
$= \frac{n}{2} \lg n - \frac{n}{2}$

Now let us turn to some data structures and study their role in the design of algorithms:

- Binary Search Trees (BST): a review (CLRS ch 12.1-3)
- Balanced Binary Search Trees: AVL Trees
- Hash Tables (CLRS ch 11.1-3)

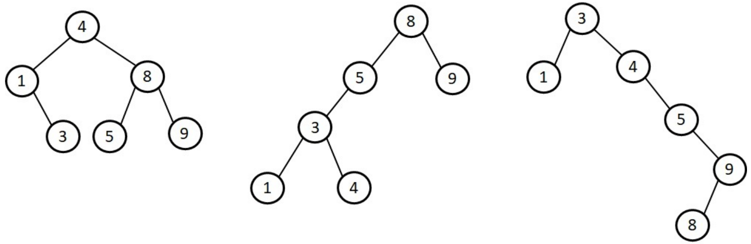
Rooted Trees

- Definition: A rooted tree is a data-structure defined recursively as:
  - The empty rooted tree, `nil`
  - A special node, the root, which has children — pointers to *distinct* and *unique* rooted trees.
    Unique: A non-`nil` tree cannot be pointed more than once
- A tree is implemented by a doubly linked list. A node contains pointers to its child nodes and parent (`nil` if non-existent) and a node holds a key (possibly other satellite data); if needed there is also an attribute $T.height$. (review CLRS ch 10.2 for linked lists).
- A node u is a descendant of node v (or that node v is an ancestor of u) if there's a path from $v.root$ to u that uses only child-pointers (and there's a path from $u.root$ to v that uses only $parent$ pointers).
- A leaf is a rooted tree with no children (all children are `nil`)
- A binary rooted tree is a rooted tree in which all nodes have at most two children, *left* and *right*.
- The length (= # edges) of the longest path from the root of the tree to a leaf is called the height of the tree.
- The i-th layer in a tree is the set of all nodes whose distance (= # edges) to the root is precisely i.

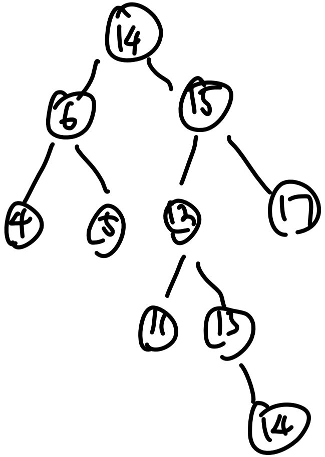


Binary Search Trees

- Definition: A binary rooted-tree is called a binary search tree if for every node v we have the following two properties
  1. All keys stored in the v's left subtree are $\leq v.key$
  2. All keys stored in the v's right subtree are $> v.key$
- On the same set of nodes/keys, there could be many binary trees of different heights.



- Like in the case of heaps: $n \leq 2^{\text{height}+1} - 1$...
- ...but (as opposed to heaps) it is wrong to assume $2^{\text{height}} \approx n$
- In fact, in the worst case $\text{height} \approx n$



Binary Search Trees

- In a BST, finding a key x is rather simple:

```
procedure Find($T, x$)
if ($T$ =nil) then
    return nil
if ($T.root.key = x$) then
    return $T$        ** alternatively, return $T.root$
else if ($x < T.root.key$) then
    return Find($T.root.left, x$)
else
    return Find($T.root.right, x$)
```

- Runtime for tree of height h is $T(h) = \begin{cases} O(1), & \text{if } h = 0 \\ O(1) + T(h-1), & \text{o/w} \end{cases}$
- I.e., runtime is $O(h)$.
- Note: While we use T to denote a BST and $T.root$ to refer to the root node of T, CLRS uses a root node to represent a tree, e.g., on p.290, the procedure TREE-SEARCH(x, k) is to find key k in the BST whose root node is x.

Binary Search Trees: Find Min

- Of particular importance is finding the min/max in a tree.

```
procedure FindMin($T$)
** precondition: $T$ isn't nil
if ($T.root.left$ =nil) then
    return $T$        ** alternatively, return $T.root$
else
    return FindMin($T.root.left$)
```

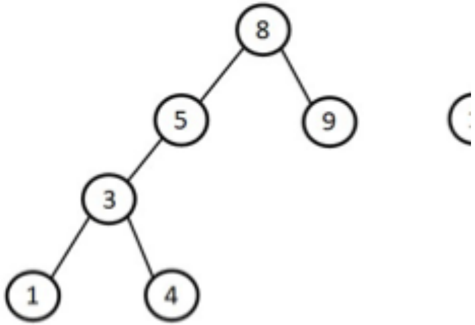
- How do we prove correctness of this code?
- Clearly, by induction. But on what?
- Answer: induction on $h_L \overset{\text{def}}{=}$ the height of the left subtree.
  - Base case: $h_L = 0$. This means that there are no left descendants. So, by definition, all keys in the right subtree are greater than $T.root.key$ so by returning $T.root$ we return the elements with the smallest key.
  - Induction step: Fix any $h_L \geq 0$. Assuming that for any tree with left-height h_L FindMin() returns the min key of this tree, we show FindMin() returns the minimum of any tree whose left-height is $h_L + 1$.
    Let T be any tree whose left-subtree's height is $h_L + 1 > 0$. So FindMin(T) returns FindMin($T.root.left$), and by IH we return the smallest of all the keys that are $\leq T.root.key$. All other keys in the tree are either $T.root.key$ or the keys stored in the right subtree which are greater than $T.root.key$. Hence, the minimum of the left subtree is indeed the minimum of the whole tree. □
- What's the code for FindMax(T)?

Binary Search Tree: Insert

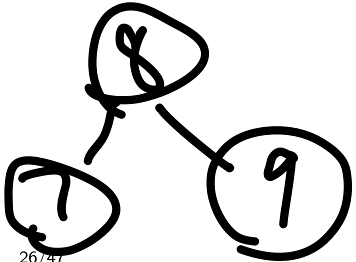
- In a BST, inserting a key x is done at the leaf:

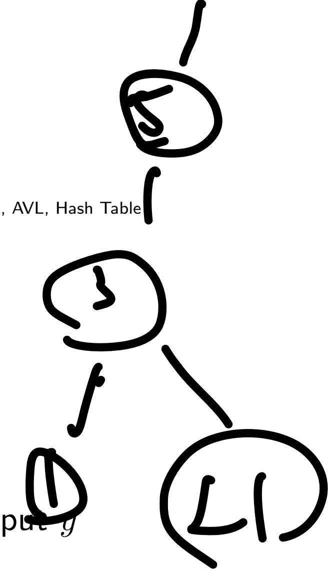
```
procedure Insert($T, x$)
$Tnew \leftarrow$ a new tree
$Tnew.root \leftarrow x$
$Tnew.root.left \leftarrow$nil
$Tnew.root.right \leftarrow$nil
if ($T =$nil) then
    ** if this is the first item in the tree.
    replace $T$ with $Tnew$
else
    InsertTree($T, Tnew$)
```

```
procedure InsertTree($T, Tnew$)
** precondition: $T$ isn't nil
if ($Tnew.root.key \leq T.root.key$) t
    if ($T.root.left =$nil) then
        $T.root.left \leftarrow Tnew$
        $Tnew.root.parent = T$
    else
        InsertTree($T.root.left, Tne$
else        ** I.e., $Tnew.root.key > T.root.key$
    if ($T.root.right =$nil) then
        $T.root.right \leftarrow Tnew$
        $Tnew.root.parent = T$
    else
        InsertTree($T.root.right, Tnew$)
```



- Insert() takes $O(\text{height of } T)$.
- The correctness statement: After invoking Insert(T, x) is resulting tree is a BST that contains all previous keys and x.



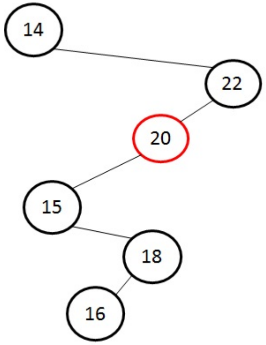


Binary Search Tree: Delete

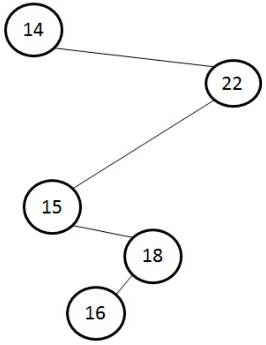
- How to do deletion of a node x?
- Easy case: x is a leaf – remove it. Done.
- How to delete a node with only a single child?
- What about deleting a node with two non-`nil` children?
- Find a different node y that can replace x — delete y and put y instead of x.
- Which node y shall we look for? The largest in the left subtree (or the smallest in the rightsubtree, both works), and delete y.

Binary Search Tree: Delete

- An example of the simple deletion case:

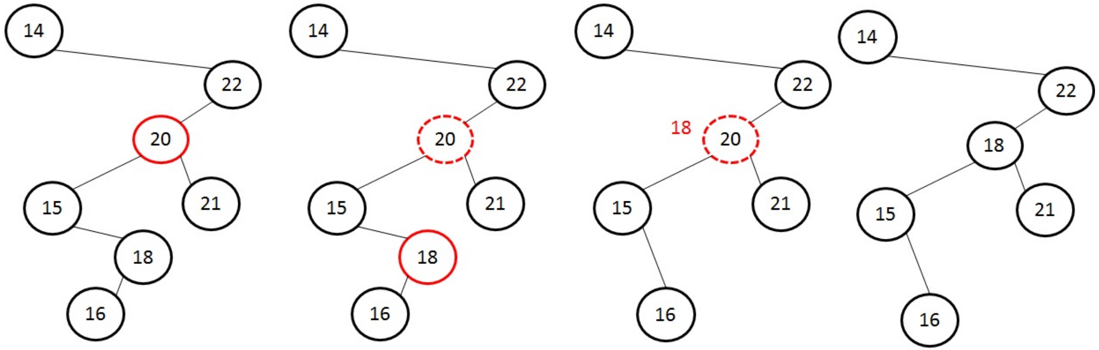


Delete(20)



Delete(20)

An example of the more complex deletion case:



Delete(20)

Delete(20)

Delete(18)

Delete(20)

Binary Search Tree: Delete

- (WC) Runtime Analysis?
- Naïve first attempt: write the runtime as a recursive relation, $T(h) = O(1) + O(h) + T(l)$, where $l = \text{depth of max node}$ (with $T(0) = O(1)$).
- But note: invoking the recursive call `DeleteRoot()` on the largest element in the left-subtree means that this node must not have a right child!
- Hence, the recursive call is invoked at most once.
- Thus, $T(h) = O(1) + O(h) + O(1) = O(h)$.

- Exercise: Depict the tree after each of instruction:
  (`Delete(`x`)` refers to `DeleteRoot(Find(`x`))`)
  `Insert(1), Insert(2), Insert(3), Insert(5), Insert(4),`
  `Delete(1), Delete(5), Delete(3), Insert(1), Insert(5),`
  `Delete(2)`

Binary Search Tree: Outputting the Sorted Sequence

- How to output all keys held in the tree from smallest to largest?
- In-order traversal:

```
procedure In-Order($T$)
if ($T \neq$nil) then
    In-Order($T.root.left$)
    Print($T.root.key$)
    In-Order($T.root.right$)
```

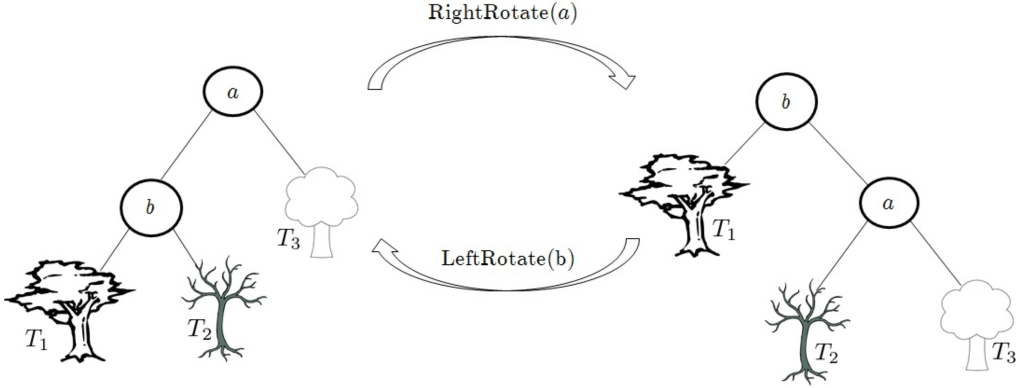
- Runtime: $T(n) = O(1) + T(n_L) + T(n_R)$ when n_L, n_R denote the number of nodes on the left/right subtree respectively. (Of course, $T(0) = O(1)$).
- Solves to $T(n) = O(n)$.

Blancing Binary Search Trees

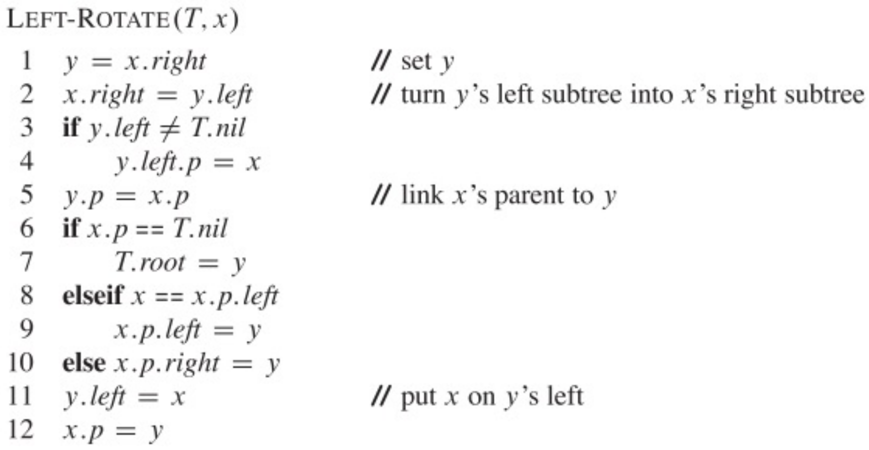
- Previous discussion shows that runtime of `Insert()`/`Delete()` is $O(h)$.
- However, in the worst case, the height of the tree is linear $\Theta(n)$.
- But a balanced or near balanced BST has height $O(\log n)$.
- We need balancing BST
  - AVL-trees (we focus on this method in this coruse)
  - Red-Black trees
  - both rely on *tree rotations* with cases how a tree rotation can achieve the desired effect in each
- These maintain the tree "shallow" — i.e., $height = O(\log(\#nodes))$

Balanced Binary Search Tree: Rotations

- Two types: Left-rotation and Right-rotation



- Right-Rotate: the old-root becomes the right child of the new root.
- Left-Rotate: the old-root becomes the left child of the new root.

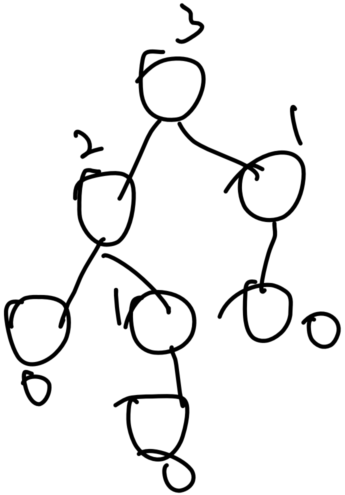


```
LEFT-ROTATE($T, x$)
$y = x.right$                          // set $y$
$x.right = y.left$                     // turn $y$'s left subtree into $x$'s right subtree
if $y.left \neq T.nil$
    $y.left.p = x$
$y.p = x.p$                            // link $x$'s parent to $y$
if $x.p == T.nil$
    $T.root = y$
elseif $x == x.p.left$
    $x.p.left = y$
else $x.p.right = y$
$y.left = x$                           // put $x$ on $y$'s left
$x.p = y$
```

Right rotate pseudo-code is symmetric.

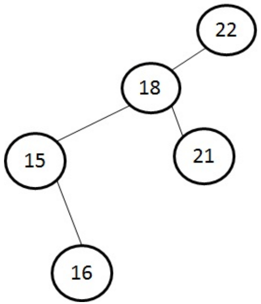
AVL Trees

- (Recall) Definition: Given a rooted tree T, its height is the max path-length from the root to a leaf.
- Notation: Given a rooted tree T, we denote h_L and h_R as the heights of the left subtree and the right-subtree (respectively) of T's root. If a subtree is `nil` we say its height is -1.
- So $T.height = \max\{T.h_L, T.h_R\} + 1$.
- Ideally: at every node we have $h_L = h_R$
  - Too restrictive. The only trees that satisfy this are complete trees (all layers are full). Such trees can only hold $2^k - 1$ nodes.
- Definition: An AVL tree is a BST where for any node we have $|h_L - h_R| \leq 1$.
  - May not be fully balanced, but almost

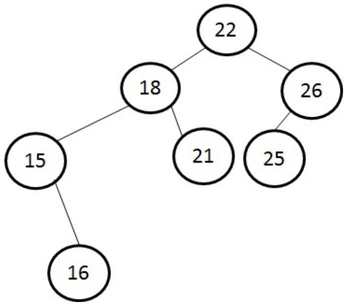


$$\text{height} = \max\left\{ \begin{array}{l} \text{height(left child)}, \\ \text{height(right child)} \end{array} \right\} + 1$$

Not AVL tree



AVL tree



AVL Trees

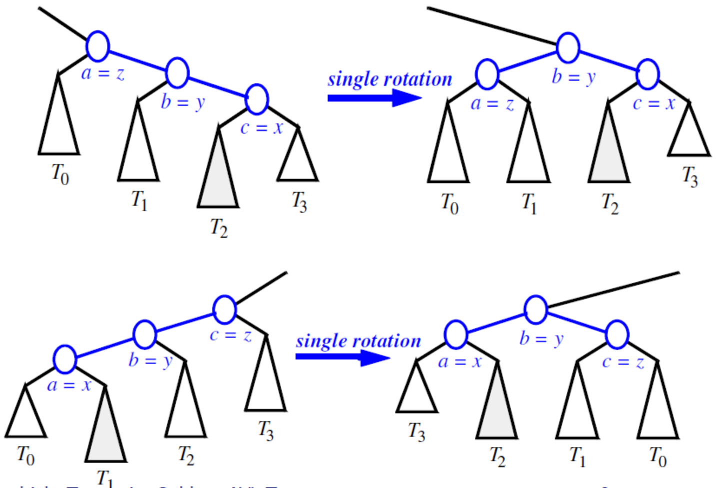
- Claim: If T is an AVL tree of height h then the number of nodes in T is at least $F(h)$ (= Fibonacci number
    - Proof: Let $N(h)$ be the minimal number of nodes in a AVL-tree of height h.
    - Fix h. Fix a tree of size h. It has left- and right- subtrees of heights h_L and h_R, and WLOG (without loss of generality) $h_L \geq h_R$.
    - Hence, $h_L = h - 1$.
    - Because of AVL-property, $h_R \geq h_L - 1 = h - 2$.
    - $N(h) = 1 + N(h_L) + N(h_R) \geq 1 + N(h-1) + N(h-2)$, with $N(0) = 1$ and $N(1) = 2$.
    - This solves to $N(h) \geq F(h) = \Theta((1.618...)^h)$.
- Corollary: $h \leq O(\log(n))$
    - We know that $n \geq F(h+1) \geq 1.5^h$ (in fact, $F(h) \approx 1.618^h$), so $h \leq \log_{1.5}(n) = O(\log(n))$
- So we want to keep our tree with the AVL property.
- Note: any tree with 1 or 2 nodes is an AVL-tree (or of height 0 or 1). But there are trees with 3 nodes that aren't AVL-trees...

How to Maintain the AVL Property

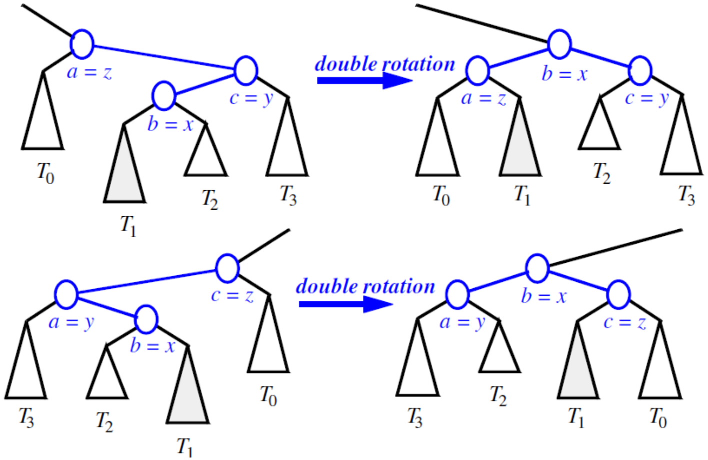
- We can use the two tree rotation operations to balance any subtree that has lost the “height-balance” property after an insertion (or deletion). There are four cases that need to be handled and are depicted on the next two slides
- We can write a procedure that will balance a node as required (using Rotation); it can be then shown that the tree is kept an AVL after each Insert and Delete operation and the cost of each of these operations remains $\Theta(h)$.
- Since in an AVL tree the height is always $O(\log n)$ then all the update operations for AVL tree will take at most $O(\log n)$.
- Note: In quizzes, tests and exams, you are not required to remember the cases on the next two slides. But you are expected to answer high level questions regarding the role of tree rotations.

How to Maintain the AVL Property

- Notice how a tree's height is reduced.



How to Maintain the AVL Property



Search :

worst case: $\Theta(n)$

average case: $O(1)$

insert: $O(1)$ delete $O(1)$

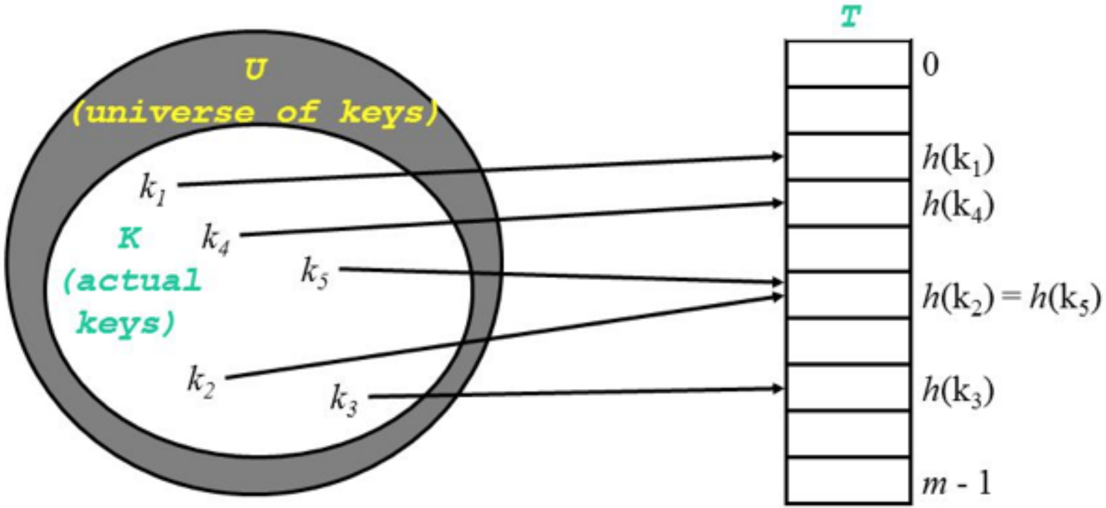
Hash-Table:

- A data structure Abstract data type
- Supports `insert`(x) (insert an element pointed to by x), `find`(key) and `remove`(x) in time $O(1)$ under reasonable assumptions.
- Doesn't keep the keys sorted. (Good or bad?)

- When is it useful?
- When there's a large universe U of potential keys, but we use only n keys.
  - E.g., think of the warehouse of "Ramazon," a new online shopping website. You type in a product's serial number and it checks whether the product exists in the Ramazon's warehouse.
  - E.g., student IDs for UofA — I just want to access the record of student #4413928
  - E.g., think of your C compiler: there are many variables, the universe of potential names for variables and functions is huge — but your code only has a moderate amount of variables and function names.

Hash-Table Illustration

The main idea: use a easy-to-compute Hash Function that maps $h : U \to \{1, 2..., m\}$. (Ideally $m = n$, but we may have $m = O(n)$.)



Hash-Table:

- Since multiple keys may be mapped to the same value $h(key_1) = h(key_2) = v$, $T[v]$ stores a list / an array of elements.
- So `insert`(x), `find`(key) and `remove`(x) all work by
  1. Compute $v = h(key)$ (or $v = h(x.key)$)
  2. Goto $T[v]$ (in $O(1)$)
  3. Traverse the list in $T[v]$ to add/search/remove
- Runtime: $O(\text{computing } h(k)) + O(\text{length of list in } T[v])$
- We use a simple h, so assume computing $h(k)$ takes $O(1)$ The dominating factor is the length of the longest list in T
- Since we assume $|U| \gg m$ we can't have an injective h.
- In fact, worst-case — all n elements have the same hash-value and the table is reduced to a linked-list, ...

Collision Resolution by Chaining:

- Place all elements hashed to the same slot into the same linked list
- `insert`(T, x): insert x at the head of list $T[h(x.key)]$;

  –(worst-case) runtime $O(1)$ if assuming x is not in T.
- `search`(T, k): search for an element with key k in list $T[h(k)]$;

  –runtime proportional to the length of the list
- `delete`(T, x): delete x (a pointer to an object) from the list $T[h(x.key)]$;

  – runtime $O(1)$ if using doubly linked lists

Analysis of Hashing by Chaining

- **Load factor:** $\alpha = n/m$, with n elements and m slots - average #elements per slot.
- **Simple uniform hashing:** Assume any element is equally likely to hash into any of the m slots, independently of where any other element has hashed to.
- Assume $O(1)$ time to compute $h(k)$.
- **Theorem:**(CLRS p.259) In a hash table in which collisions are resolved by chaining,
  - an unsuccessful search takes average-case time $\Theta(1 + \alpha)$
  - a successful search takes average-case time $\Theta(1 + \alpha)$, under the assumption of simple uniform hashing.
- Its proof (p.260 of CLRS) will be discussed in class.
- **Implication:** Suppose $n = O(m)$. Then,

$$\alpha = n/m = O(m)/n = O(1)$$

Searching takes constant time on average!

Hash Functions:

- Simple uniform hashing: we typically have no way to check this condition, as we in general rarely know the probability distribution from which the keys are drawn
- h that works reasonably well in practice:
    - **The division method:** map $h(k) = (k \mod m)$
    - If distribution of keys is uniform over $\{1, 2, ..., |U|\}$, then distribution of $h(k)$s is $\sim$uniform on $\{0, 1, 2, ..., m - 1\}$.
- Fixed hash functions are subject to adversarial attacks.

Universal Hashing (Optional, p.265-268 of CLRS):

- At each execution, select the hash function randomly from a carefully designed family of hash functions.
- Different behavior even for the same input; the probability of worst-case scenario is small.
- Def: A collection of hash functions $H = \{h : U \to [m]\}$ is said to be **universal** if $\forall k, l \in U$, $k \neq l$: $\Pr_{h\in H}[h(k) = h(l)] \leq 1/m$.
  I.e., the probability of k and l to be hashed into the same value is no larger than $1/m$.
- Under the chaining method to resolve collisions, we have
  Theorem: Let H be a universal collection of hash functions. Suppose h is chosen randomly from H and has been used to hash n keys into a table T of size m. If key k is not in T, then the expected length $\mathtt{E}[n_{h(k)}]$ of the list that k hashed to is $\leq$ the load factor $\alpha = n/m$. If key k is in T, then the expected length $\mathtt{E}[n_{h(k)}]$of the list containing k is $\leq 1 + \alpha$.

A Hash Function with Provable Guarantees:

- Pick some prime $p > |U|$
- Pick a uniformly at random from $\{1, 2, ..., p-1\}$ and b uniformly at random from $\{0, 1, 2, .., p-1\}$
- Define $h_{ab}(k) = \Big((a \cdot k + b) \mod p\Big) \mod m$
- Let H_{pm} be the collection of such functions h_{ab}.
- **Theorem:** The family of hash functions H_{pm} is universal.
- Proof (p. 267-268 of CLRS)