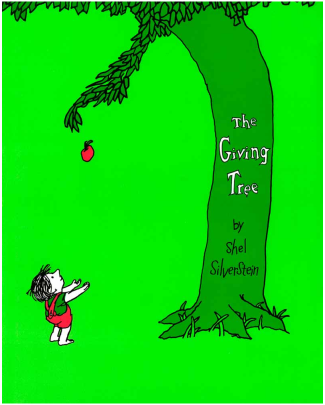
Agenda:

- Heaps:
  - Max-Heapify
  - Build-Max-Heap
- Heapsort
- Priority Queues
- Binary Search Trees
  - Find()
  - Insert(), Delete()
  - Predecessor(), Successor() (on your own)
- Balancing BST
  - AVL
  - Red-Black Tree



Reading:

- CLRS Ch.6
- CLRS Ch.12 & 13

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
- Much like in a doubly linked list, it is also useful to keep a pointer $parent$ back to the tree which this tree is one if its children.
- We say 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, which we denote as *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.

Heaps data structure:

- An array $A[1..n]$ of n pairwise *comparable* keys (either '$\geq$' or '$\leq$')
- An implicit *binary tree*, where
  - $A[2j]$ is the left child of $A[j]$
  - $A[2j+1]$ is the right child of $A[j]$
  - So: $A[\lfloor \frac{j}{2} \rfloor]$ is the parent of $A[j]$
- There are max-heap and min-heap. We use *max-heap*.
- Keys satisfy the *max-heap property*: for every node j we have
  $A[\lfloor \frac{j}{2} \rfloor] \geq A[j]$ (key of parent $\geq$ key of node)
- So the root ($A[1]$) is the maximum among the n keys.
- This gives the alternative definition of a heap:
  In any *sub-heap*, the root is the largest key
- Viewing heap as a binary tree, height of the tree is $h = \lfloor \lg n \rfloor$.
  h is called the *height* of the heap.
  [— the number of edges on the longest root-to-leaf path]
- All layers i from 0 to $h-1$ are full.
- A heap of height h can hold $[2^h, ..., 2^{h+1} - 1]$ keys.

Heaps - examples:

- Examples of Heaps:
  - $A = [31]$, or any array with a single element
  - $A = [2, 1]$
  - $A = [6, 3, 5]$
  - $A = [6, 3, 5, 1, 2, 4]$
  - $A = [100, 42, 78, 13, 41, 77, 12]$
- Examples of Non-Heaps:
  - $A = [1, 2]$
  - $A = [4, 3, 5]$
  - $A = [100, 42, 78, 13, 41, 77, 12, 14]$
- Remember: The heap is stored in an array. The tree is *implicit*.
- Thus, all layers except for maybe the last are full.

Max-Heapify:

- It makes an almost-heap into a heap.
  - Almost-heap: only the root of the heap might violates the heap-property
- Pseudocode:

```
procedure Max-Heapify(A, i)
      **turns almost-heap into a heap
      **pre-condition:  tree rooted at A[i] is an almost-heap
      **post-condition:  tree rooted at A[i] is a heap
   lc ← leftchild(i)
   rc ← rightchild(i)
   largest ← i
   if (lc ≤ heapsize(A) and A[lc] > A[largest]) then
      largest ← lc
   if (rc ≤ heapsize(A) and A[rc] > A[largest]) then
      largest ← rc        **largest = index of max{A[i], A[rc], A[lc]}
   if (largest ≠ i) then
      exchange A[i] ↔ A[largest]
      Max-Heapify(A, largest)
```

- WC running time: $O(h) = O(\lg n)$.

Building a heap from an array:

- Given an array of n keys $A[1], A[2], \ldots, A[n]$, permute the keys in A so that A is a heap
- Outline:
  1. Look at the implicit binary tree that A induces
  2. Consider the leafs (the bottom-level nodes in the binary tree):
     Each of them has a single-key $\Rightarrow$ each of them is a heap
  3. Consider the nodes on the second-to-last level:
     The subtrees rooted at these nodes are almost-heaps:
     `Max-Heapify` them into heaps!
  4. Now, consider the nodes on the third-to-last level:
     The subtrees rooted at those nodes are almost-heaps:
     `Max-Heapify` them into heaps!

     $\vdots$
  5. The whole tree becomes an almost heap:
     Max-Heapify tree root into a heap!

     DONE!

Building a heap from an array (cont'd):

- Pseudocode:

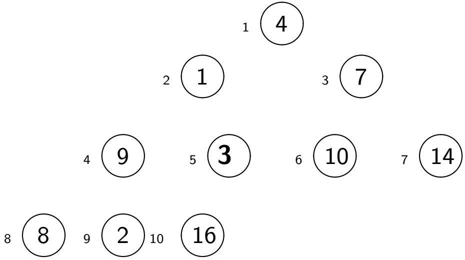
procedure Build-Max-Heap(A) **turn an array into a heap

  $heapsize(A) \leftarrow length[A]$

  for ($i \leftarrow \left\lfloor \frac{length[A]}{2} \right\rfloor$ downto 1) do

    Max-Heapify(A, i)

$A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$:   Max-Heapify($A, 5$):

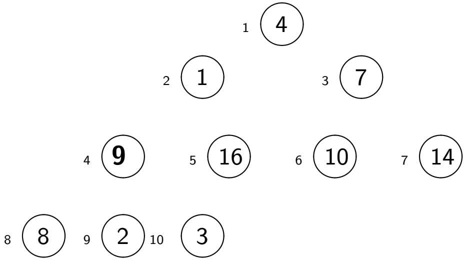


Building a heap from an array (cont'd):

- Pseudocode:

procedure Build-Max-Heap(A)
  **turn an array into a heap
 $heapsize(A) \leftarrow length[A]$
 for $i \leftarrow \left\lfloor \frac{length[A]}{2} \right\rfloor$ downto 1
  do Max-Heapify(A, i)

$A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$ Max-Heapify($A, 4$):

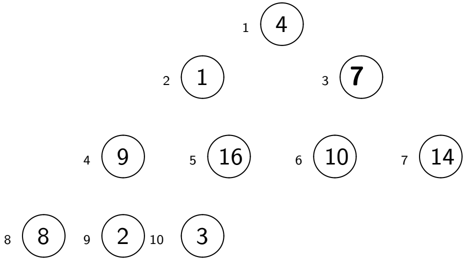


Building a heap from an array (cont'd):

- Pseudocode:

```
procedure Build-Max-Heap(A)
      **turn an array into a heap
   heapsize(A) ← length[A]
   for i ← ⌊length[A]/2⌋ downto 1
      do Max-Heapify(A, i)
```

$A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$ Max-Heapify$(A, 3)$:



Building a heap from an array (cont'd):

- Pseudocode:

`procedure Build-Max-Heap`(A)

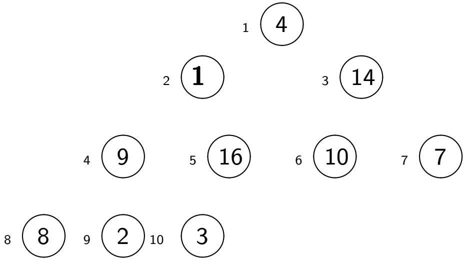
  `**turn an array into a heap`

 $heapsize(A) \leftarrow length[A]$

 `for` $i \leftarrow \left\lfloor \frac{length[A]}{2} \right\rfloor$ `downto` 1

  `do Max-Heapify`(A, i)

$A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$ `Max-Heapify`$(A, 2)$:

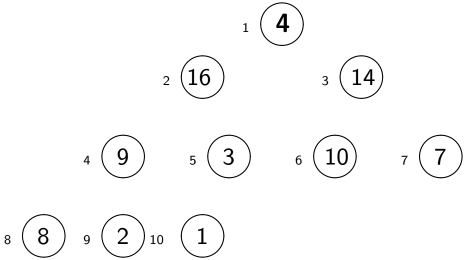


Building a heap from an array (cont'd):

- Pseudocode:

procedure Build-Max-Heap(A)
      **turn an array into a heap
   $heapsize(A) \leftarrow length[A]$
   for $i \leftarrow \left\lfloor \frac{length[A]}{2} \right\rfloor$ downto 1
      do Max-Heapify(A, i)

$A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$ Max-Heapify$(A, 1)$:



Building a heap from an array (cont'd):

- Pseudocode:

procedure Build-Max-Heap(A)

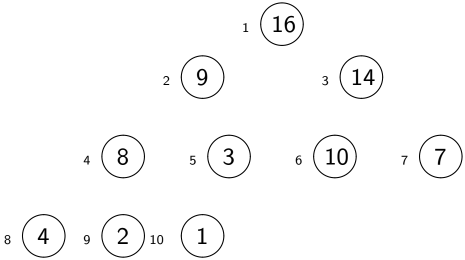
  **turn an array into a heap

 $heapsize(A) \leftarrow length[A]$

 for $i \leftarrow \left\lfloor \frac{length[A]}{2} \right\rfloor$ downto 1

  do Max-Heapify(A, i)

$A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$ result:



Building a heap from an array (cont'd):

- Pseudocode:

procedure `Build-Max-Heap`(A)
    `**turn an array into a heap`
  $heapsize(A) \leftarrow length[A]$
  `for` $i \leftarrow \left\lfloor \frac{length[A]}{2} \right\rfloor$ `downto` 1
    `do Max-Heapify`(A, i)

- Worst case running time: because we make at most $\frac{n}{2}$ calls to Max-Heapfiy, each takes $O(\lg(n))$ we have $O(n \log(n))$.

Building a heap from an array (cont'd):

- Correct bound is $O(n)$:
- `Max-Heapify`'s runtime is $O(k)$ for a node at height k.
  - At height 1 we have at most $n/2$ nodes.
  - At height 2 we have at most $n/4$ nodes.
  - At height 3 we have at most $n/8$ nodes.
  - ...
  - At height $\lg(n)$ we have at most 1 node.
- So runtime is upper bounded by:

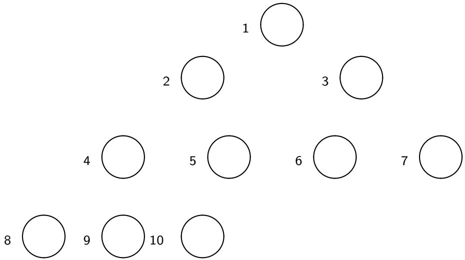
$$\begin{aligned}
\sum_{k=1}^{\lg(n)} k \cdot \frac{n}{2^k} &= n \sum_{k=1}^{\lg(n)} \frac{k}{2^k} = n \left(\sum_{k=1}^{3} \frac{k}{2^k} + \sum_{k=4}^{\lg(n)} \frac{k}{2^k} \right) \leq n \left(\sum_{k=1}^{3} \frac{k}{2^k} + \sum_{k=4}^{\lg(n)} \frac{2^{k/2}}{2^k} \right) \\
&= n \left(\sum_{k=1}^{3} \frac{k}{2^k} + \sum_{k=4}^{\lg(n)} \frac{1}{2^{k/2}} \right) \leq n \left(\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \sum_{k=4}^{\infty} \frac{1}{2^{k/2}} \right) \\
&\leq n \left(2 + \sum_{k=0}^{\infty} \left(\sqrt{\frac{1}{2}} \right)^k \right) = n \left(2 + \frac{1}{1-\sqrt{\frac{1}{2}}} \right) \leq n \left(2 + \frac{1}{1-0.75} \right) \leq 6n
\end{aligned}$$

- Tighter analysis will yield running time is actually $2n - \lg n - 2$.

Heapsort algorithm:

- We can use heaps to design another sorting algorithm.
- `Heapsort` is a sorting algorithm using heaps.
- The ideas:
  - Build the array into a heap (WC cost $\Theta(n)$)
  - The first key $A[1]$ is the maximum and thus should be in the last position when sorted
  - Exchange $A[1]$ with $A[n]$, and decrease heap size by 1
  - Max-Heapify the array $A[1..(n-1)]$, which is an almost-heap, into a heap.
  - Repeat for positions $n-1, n-2, \ldots, 2$.

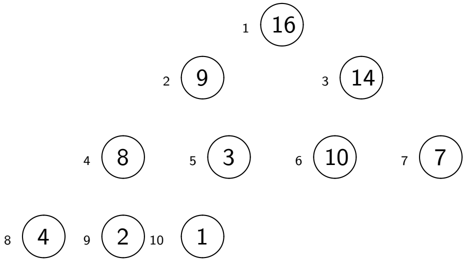
- An example: $A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$
  Build into a heap:



Heapsort algorithm (cont'd):

- We can use heaps to design another sorting algorithm.
- `Heapsort` is a sorting algorithm using heaps.
- The ideas:
  - Build the array into a heap (WC cost $\Theta(n)$)
  - The first key $A[1]$ is the maximum and thus should be in the last position when sorted
  - Exchange $A[1]$ with $A[n]$, and decrease heap size by 1
  - Max-Heapify the array $A[1..(n-1)]$, which is an almost-heap, into a heap.
  - Repeat for positions $n-1, n-2, \ldots, 2$.

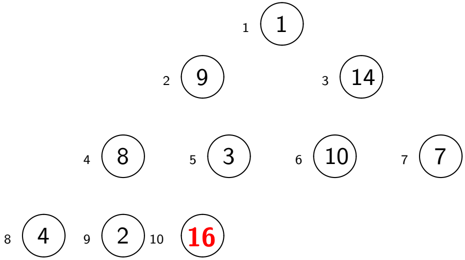
- An example: $A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$
  Heapsize $= 10$:



Heapsort algorithm (cont'd):

- We can use heaps to design another sorting algorithm.
- `Heapsort` is a sorting algorithm using heaps.
- The ideas:
  - Build the array into a heap (WC cost $\Theta(n)$)
  - The first key $A[1]$ is the maximum and thus should be in the last position when sorted
  - Exchange $A[1]$ with $A[n]$, and decrease heap size by 1
  - Max-Heapify the array $A[1..(n-1)]$, which is an almost-heap, into a heap.
  - Repeat for positions $n-1, n-2, \ldots, 2$.

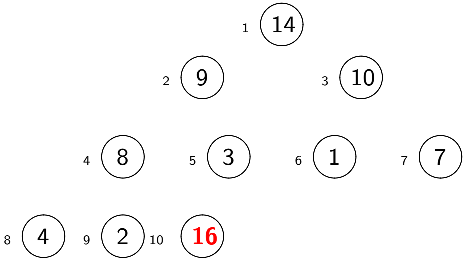
- An example: $A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$
  Exchange $A[1]$ and $A[10]$, decrement Heapsize to 9, and Max-Heapify it (restore the heap property):



Heapsort algorithm (cont'd):

- We can use heaps to design another sorting algorithm.
- `Heapsort` is a sorting algorithm using heaps.
- The ideas:
  - Build the array into a heap (WC cost $\Theta(n)$)
  - The first key $A[1]$ is the maximum and thus should be in the last position when sorted
  - Exchange $A[1]$ with $A[n]$, and decrease heap size by 1
  - Max-Heapify the array $A[1..(n-1)]$, which is an almost-heap, into a heap.
  - Repeat for positions $n-1, n-2, \ldots, 2$.

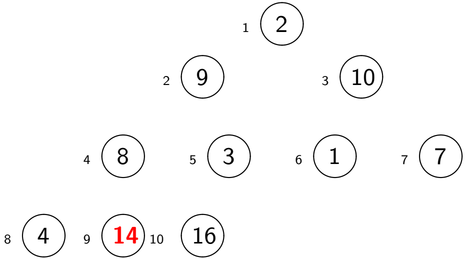
- An example: $A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$
  Resultant tree: Heapsize $= 9$:



Heapsort algorithm (cont'd):

- We can use heaps to design another sorting algorithm.
- `Heapsort` is a sorting algorithm using heaps.
- The ideas:
  - Build the array into a heap (WC cost $\Theta(n)$)
  - The first key $A[1]$ is the maximum and thus should be in the last position when sorted
  - Exchange $A[1]$ with $A[n]$, and decrease heap size by 1
  - Max-Heapify the array $A[1..(n-1)]$, which is an almost-heap, into a heap.
  - Repeat for positions $n-1, n-2, \ldots, 2$.

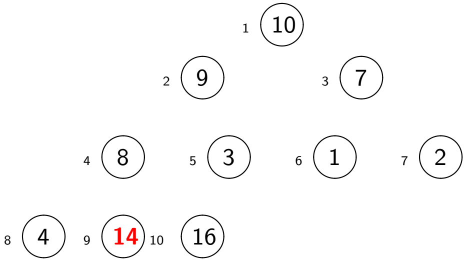
- An example: $A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$
  Exchange $A[1]$ and $A[9]$, decrement Heapsize to 9, and Max-Heapify it (restore the heap property):



Heapsort algorithm (cont'd):

- We can use heaps to design another sorting algorithm.
- `Heapsort` is a sorting algorithm using heaps.
- The ideas:
  - Build the array into a heap (WC cost $\Theta(n)$)
  - The first key $A[1]$ is the maximum and thus should be in the last position when sorted
  - Exchange $A[1]$ with $A[n]$, and decrease heap size by 1
  - Max-Heapify the array $A[1..(n-1)]$, which is an almost-heap, into a heap.
  - Repeat for positions $n-1, n-2, \ldots, 2$.

- An example: $A[1..10] = \{4, 1, 7, 9, 3, 10, 14, 8, 2, 16\}$
  Resultant tree: Heapsize $= 8$:



Heapsort algorithm (cont'd):

- Pseudocode:

```
procedure Heapsort(A)
      **post-condition:  sorted array
   Build-Max-Heap(A)
   for (i ← heapsize(A) downto 2) do
      exchange A[1] ↔ A[i]
      heapsize(A) ← heapsize(A) − 1
      Max-Heapify(A, 1)
```

- WC running time analysis:
  - Build-Max-Heap in $O(n)$
  - For each position $i = n, n-1, ..., 2$, Max-Heapify takes $O(\lg i)$, so in total this is $\Theta(n \log(n))$.

$$\sum_{i=2}^{n} \log(i) \quad \leq \quad \sum_{i=1}^{n} \log(n) = n \log(n)$$

$$\sum_{i=2}^{n} \log(i) \quad \geq \quad \sum_{i=\lceil n/2 \rceil}^{n} \log(n/2) = \lfloor \tfrac{n}{2} \rfloor \cdot (\log(n) - 1) \geq \tfrac{1}{3} n \log(n)$$

  - So, in total $\Theta(n \log n)$

Heapsort algorithm — Best Case (optional):

- BC running time analysis:
  - $\Omega(n \log n)$ when all keys are distinct. Proof outline:
    - Let's look at S, the set of the largest $n/2$ keys. Denote $t = \log_2(n) - 2$.
    - Case 1: If all t layers are populated by keys from S, then any leaf not in S when put as a root (exchanged with the largest element) has to travel down t times. As $n/4 - 1$ key from S populate the top t layers, only $n/2 - n/4 + 1 = n/4 + 1$ elements from S can appear on the last 2 layers. Since there are $n/2$ leafs we have that $n/4 - 1$ leafs must be not from S, incurring a runtime of $(\frac{n}{4} - 1) \cdot t = \Omega(n \log(n))$.
    - Case 2: Otherwise, at least $|S|/2 = n/4$ elements appear in the bottom two layers of the heap. How do we remove them from the heap? They have to percolate all the way to the root of the heap. So for any swap done by `Max-Heapify`, instead of charging 1 (the cost of the swap) to the node that's dropping down, charge it to the node that is moved up. The overall cost of getting each of those S nodes in the bottom two layers to "climb up" to the root is $(S/2) \cdot t = \Omega(n \log(n))$.
  - So WC and BC are both $\Theta(n \log n)$.

Heapsort algorithm — Conclusion:

- WC running time:
  - Build-Max-Heap takes $O(n)$.
  - $n-1$ calls to Max-Heapify: $O(n \log n)$.
  - Overall $O(n \log n)$.
  - In the worst-case, It is not hard to show that Max-heapify can take $\Omega(\log n)$.
  - Thus the WC running time is also $\Omega(n \log n)$.
  - Total: $\Theta(n \log n)$.
- Correctness – prove on your own:
  - Correctness for Max-Heapify?
    (a recursion, use induction on height of i)
  - LI for Build-Max-Heap?
    For any $j \geq i$, the subtree rooted at j is heap (CLRS p.157)
  - LI for heapsort
    $A[1,...,i]$ is a heap & $A[i+1,...n]$ contains the $n-i$ largest keys, sorted.

Priority Queue:

- An abstract data structure for maintaining a set S of *elements* each associated with a *key*
- Key — represents the priority of the element
- Example: a set of jobs to be scheduled on a shared computer.
  - The jobs arrive and should be placed in the queue.
  - Each has a priority. Queue should be with respect to this.
  - To perform a job, we "extract" the one in the queue with highest priority.
- In general, a PQ supports these operations:
  - `initialize` — insert all keys at once
  - `insert` — a new element
  - `maximum` — return the element with the maximum key
  - `extract maximum` — return the maximum and remove the element from the queue
  - `increase key` — increase the priority for an element
- Implementation? Heap !!!

Priority Queue:

- `Initialize`(A) — Build-Max-Heap. So this takes $\Theta(n)$ time.
- `Maximum`(A) — Return $A[1]$. Takes $\Theta(1)$ time.
- `Extract-Maximum`(A) —
  Like deleting from an array: put $A[n]$ as the new first element before returning the max.
  The difference: we `Max-Heapify`$(A, 1)$ to make this array into a heap.
  $\Theta(\lg n)$ time.

  ```
  procedure Heap-Extract-Max(A)        ** precondition: A isn't empty
      max ← A[1]
      A[1] ← A[heapsize[A]]
      heapsize[A] ← heapsize[A] − 1
      if (heapsize[A] > 0) then
          Max-Heapify(A, 1)
      return max
  ```

- `Increase-Key`(A, i, new_key)
- `Insert`(A, new_key)

Priority Queue:

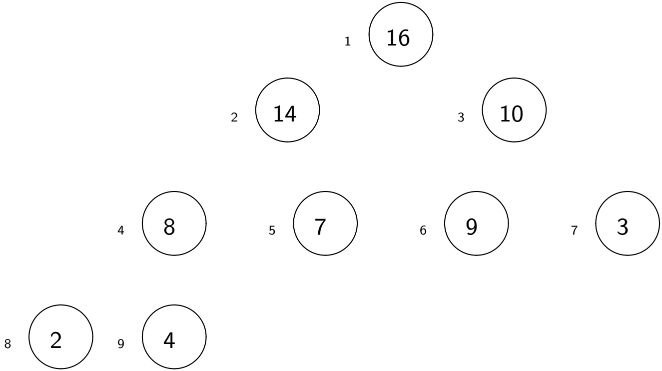
- `Initialize`(A)
- `Maximum`(A)
- `Extract-Maximum`(A)
- `Increase-Key`(A, i, new_key) — The inverse of `Max-Heapify`: Increase the priority value for $A[i]$ and bubble up to till max-heap property is restored. $\Theta(\lg n)$ time.

  procedure Heap-Increase-Key(A, i, key) ** Precondition: $key \geq A[i]$
  $A[i] \leftarrow key$
  while $(i > 1$ **and** $A[Parent(i)] < A[i])$ **do**
  exchange $A[i] \leftrightarrow A[Parent(i)]$
  $i \leftarrow Parent(i)$

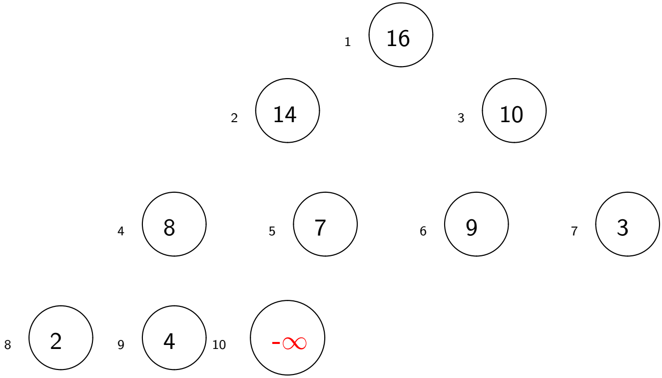
- `Insert`(A, new_key) — Add a new key with lowest priority, increase its priority to new_key. $\Theta(\lg n)$ time.

  procedure Heap-Insert(A, key)
  $heapsize[A] \leftarrow heapsize[A] + 1$
  $A[heapsize[A]] \leftarrow -\infty$ ** or any value smaller than all keys in A
  Heap-Increase-key $(A, heapsize[A], key)$

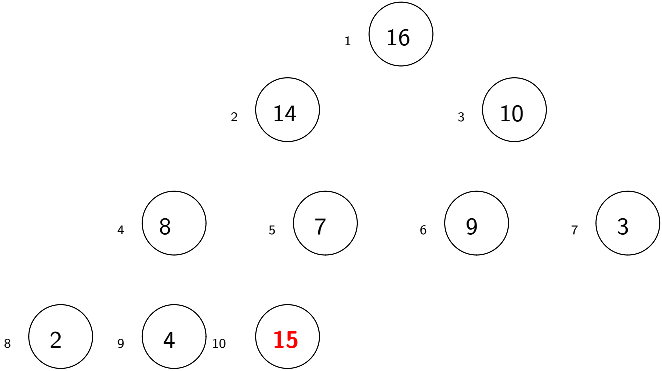
Starting with a heap:



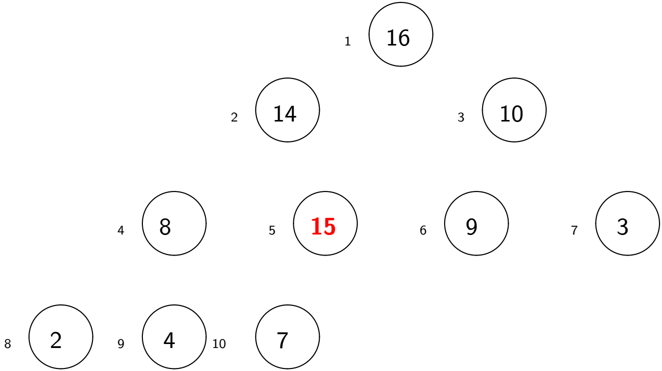
Max-heap-Insert $(A, 15)$:



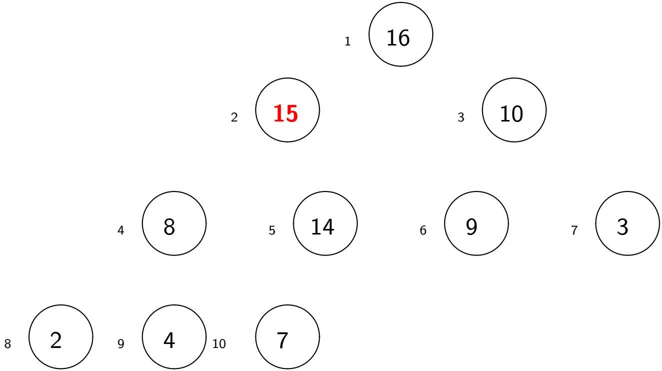
Heap-Increase-Key $(A, heapsize[A], 15)$ is called:



Bubbling up key 15:



Bubbling up key 15:



Priority Queue:

- `Initialize`(A) — Build-Max-Heap. Takes $\Theta(n)$ time.
- `Maximum`(A) — Return $A[1]$. Takes $\Theta(1)$ time.
- `Extract-Maximum`(A) — Like deleting from an array: put $A[n]$ as the new first element before returning the max. The difference: we `Max-Heapify(`$A, 1$`)` to make this array into a heap. $\Theta(\lg n)$ time.
- `Increase-Key`(A, i, new_key) — The inverse of `Max-Heapify`: Increase the priority value for $A[i]$ and bubble up to till max-heap property is restored. $\Theta(\lg n)$ time.
- `Insert`(A, new_key) — Add a new key with lowest priority, increase its priority to new_key. $\Theta(\lg n)$ time.

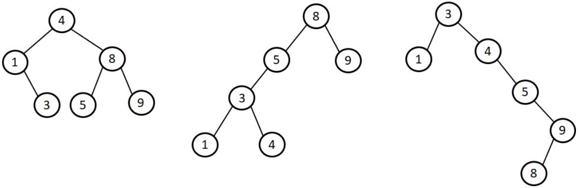
- Note that we didn't mention `Decrease-key(`A, i, new_key`)`. Why?
- Because we already know how to deal with decreasing keys!
  Once i's key is set to a new *smaller* value, then the subheap rooted at i becomes an almost-heap.
  Run `Max-Heapify(`A, i`)`.

Sorting on the Fly

- So far — we have considered the notion of a static problem: someone gives you an array of n items and we have to sort them.
- However, the problem can be studied also in the dynamic setting: the set of items changes, keys are added and removed (inserted and deleted), and every now and then, we wish to sort them (or find a value x among them).
- Option 1: use a data-structure that does insertion and deletion fast (array, list, hash-table), and upon a sorting request - run a sorting algorithm.
- Option 2: use a data-structure that keeps the elements sorted. (a binary search tree)
- Which is the better option: depends on the sequence of calls.
  If a `find()/sort()` request comes once in a long while, after many insertion/deletions, then use option 1.
  If there are many `find()/sort()` requests, or the they appear after the insertion/deletion of only a few elements — use option 2.

Binary Search Trees

- Explicit data structure: we have Trees, roots, left- and right-children.
- Definition: A binary rooted-tree is called 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 $> v.key$
- Note: 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

- Explicit data structure: we have Trees, roots, left- and right-children.
- Definition: A binary rooted-tree is called 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 $> v.key$
- 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 (T.root.key ≤ x) 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 $O(h)$.

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 $h_L \stackrel{\text{def}}{=}$ 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. $\square$
- 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 ← a new tree
Tnew.root ← x
Tnew.root.left ← nil
Tnew.root.right ← 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 ≤ T.root.key) then
   if (T.root.left = nil) then
      T.root.left ← Tnew
      Tnew.root.parent = T
   else
      InsertTree(T.root.left, Tnew)
else      ** I.e., Tnew.root.key > T.root.key
   if (T.root.right = nil) then
      T.root.right ← Tnew
      Tnew.root.parent = T
   else
      InsertTree(T.root.right, Tnew)
```

- HW: prove correctness and that `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.

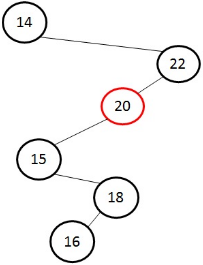
Binary Search Tree: Delete

- 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?
  - Suggestion: Find a different node y that can replace x — delete y (recursively) and put y instead of x.
  - Which node y shall we look for?
- procedure DeleteRoot(T)

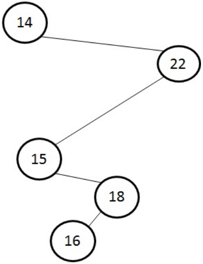
```
** precondition: we're deleting the root of T (i.e. already called Find())
if (T.root.left =nil)
   if (T.parent ≠nil) then
      ReplaceChild(T.parent, T, T.root.right)
   if (T.root.right ≠nil) then
      SetParent(T.root.right, T.parent)
   delete T
else if (T.root.right =nil)
   if (T.parent ≠nil) then
      ReplaceChild(T.parent, T, T.root.left)
   if (T.root.left ≠nil) then
      SetParent(T.root.left, T.parent)
   delete T
else
   T_y ←FindMax(T.root.left)
   y ← T_y.root.key
   DeleteRoot(T_y)
   T.root.key ← y
```

Binary Search Tree: Delete

- `ReplaceChild()` and `SetParent()` do precisely what you think.
  - `ReplaceChild`(T, old, new) checks which of T's children is old and sets it as new instead.
  - `SetParent`(T, p) sets $T.root.parent \leftarrow p$.
- An example of the simple deletion case:

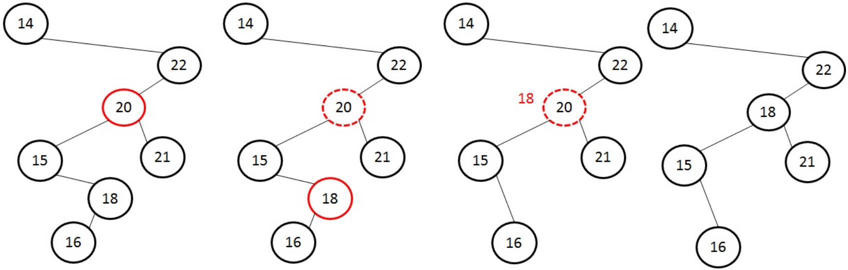


Delete(20)



Delete(20)

An example of the more complex deletion case:



Delete(20)

Delete(20)

Delete(18)

Delete(20)

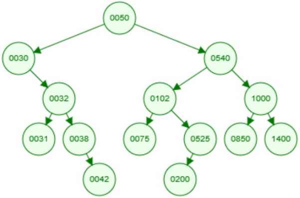
Binary Search Tree: Delete

- Runtime Analysis?
- Naïve first attempt: write the runtime as a recursive relation, $T(h) = O(1) + T(l)$ 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: Successor()

- `Successor`(T): We have a rooted tree T with a certain key k in its root. We want to find in the overall tree that holds T the key immediately following k.
  - Namely, if we were to order the key in the tree holding T, `Successor`(T) will come immediately after $T.root.key$.
  - Alternatively, `Successor`(T) is the min-key of all keys $\geq T.root.key$.



Examples:
`Successor`$(32) = 38$
`Successor`$(102) = 200$
`Successor`$(540) = 850$
`Successor`$(42) = 50$
`Successor`$(200) = 525$
`Successor`$(1400) = \perp$ (no such element)

- The examples show:
  If T has a right child, then `Successor`(T) is the min-key in T's right subtree.
  If T's right child $=$ `nil` then `Successor`$(T) =$ the first ancestor for which T is in its left subtree.

Binary Search Tree: Successor()

- If T has a right child, then `Successor(`T`)` is the min-key in T's right subtree.
  If T's right subtree = `nil` then `Successor(`T`)`= the first ancestor for which T is in its left subtree.
- procedure `Successor(`T`)`

```
if (T.root.right ≠nil) then
   return FindMin(T.root.right)
else
   T1 ← T                  ** we maintain the invariant that T1 is an ancestor of T
   T2 ← T.parent           ** T2 is the candidate solution
   while (T2 ≠nil) do
      if (T2.root.left = T1) then
         return T2         ** T2 is the lowest ancestor for which T1
                           ** (and therefore T) lies in its left subtree
      else
         T1 ← T2
         T2 ← T2.parent
   return nil              ** we've reached the root of the overall tree
```

- Runtime?
- `Predecessor(`T`)`: We have a rooted tree T with a certain key k in its root. We want to find in the overall tree that holds T the key immediately before k. How does the code of `Predecessor(`T`)` looks like?

Binary Search Tree: Successor()

- If T has a right child, then `Successor(`T`)` is the min-key in T's right subtree.
  If T's right subtree $=$ `nil` then `Successor(`T`)`$=$ the first ancestor for which T is in its left subtree.
- There's a subtle point worth mentioning here:
  In the right-child$=$`nil` case, instead of the original code that keeps two pointers (T_1 and T_2) we could potentially just use a single pointer. We would just iterate over the ancestors of T and compare each $T_2.root.key$ to $T.root.key$, and halt upon the first ancestor with a key larger then $T.root.key$. Namely: replace the suitable lines with—

  $T_2 \leftarrow T.parent$
  while ($T_2 \neq$**nil**) **do**
      if ($T_2.root.key > T.root.key$) **then**
          return T_2
      else
          $T_2 \leftarrow T_2.parent$
  return nil
- This is another $O(h)$-time code, and thus equivalent to our original code.
- But this is a code that (in the WC) makes $O(h)$ Key-Comparisons, whereas the original code only checks pointers and assignments.
  Therefore, the former is better in the case where Key-Comparisons are "heavy" operations that take considerably more time.

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 ≠nil) then
    In-Order(T.root.left)
    Print(T.root.key)
    In-Order(T.root.right)
```

- As opposed to the pre-order / post-order traversals:

```
procedure Pre-Order(T)
if (T ≠nil) then
    Print(T.root.key)
    Pre-Order(T.root.left)
    Pre-Order(T.root.right)
```

```
procedure Post-Order(T)
if (T ≠nil) then
    Post-Order(T.root.left)
    Post-Order(T.root.right)
    Print(T.root.key)
```

- 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 $T(n) = O(n)$.

Binary Search Tree: Summary

Operation	Description	Time
`Find(`x`)`	Recurse on left/right subtree based on KC between $root.key$ and x	$O(h)$
`FindMax(`x`)`	Recurse on right subtree until no right child exists	$O(h)$
`Insert(`x`)`	Like `Find()` until reach a `nil` and replace it with a new tree that has $root.key = x$	$O(h)$
`Delete(`x`)`	If x has ≤ 1 child — connect $x.parent$ with $x.child$ (could be `nil`). If x has two children — replace x with y, largest key in the left subtree, and delete y.	$O(h)$
`Predecessor(`x`)`	If left subtree isn't empty: `FindMax` of left subtree. But if left-subtree is empty — go up the $parents$ until you find a parent that x is in its right subtree. (If reached the root, x is smallest in the whole tree)	$O(h)$
`Successor(`x`)`	If right subtree isn't empty: `FindMin` of right subtree. But if right-subtree is empty — go up the $parents$ until you find a parent that x is in its left subtree. (If reached the root, x is largest in the whole tree)	$O(h)$
`In-Order`	Print all keys in left subtree in order; print $root$; print all keys in right subtree in order	$O(n)$

Binary Search Tree: Tree Height

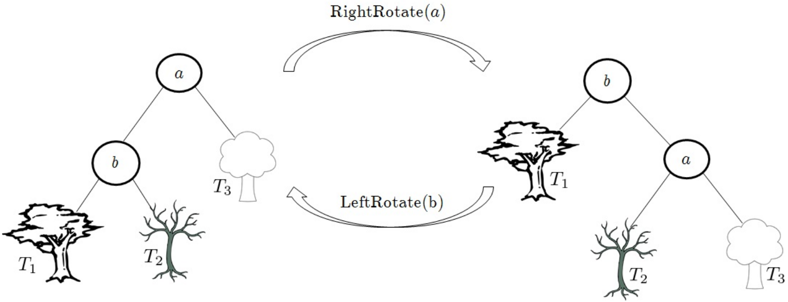
- Previous discussion shows that runtime of `Insert()`/`Delete()` is $O(h)$.
- However, in the worst case, the height of the tree is linear $\Theta(n)$.
- Hence, n `Insert()` and then `In-Order()` (which means we output the keys in a sorted order) take $n \cdot O(n) + O(n) = O(n^2)$.
- Whereas we can sort already in time $O(n \log(n))$.
- Observation: if we maintain the tree of height $O(\log(n))$ then sorting would takes us $O(n \log(n))$
- For that end, we need balanced BST
  - AVL-trees
  - Red-Black trees
- These maintain the tree "shallow" — i.e., $height = O(\log(\#nodes))$

Balanced Binary Search Trees

- We discuss two types of balanced BST for which $height = O(\log(\#nodes))$
  - AVL-trees (Adelson-Velsky & Landis)
  - Red-Black trees
- Plan of attack:
  - Both AVL/RB-trees work by enforcing some special properties on the BST.
  - Only `Insert()/Delete()` alter the tree (all other operation don't change the tree's structure).
  - So when we introduce a new leaf / remove a node with at least one child missing — the tree, which before the call had this property, now might violate this property.
  - Therefore, upon introducing a new leaf / removing a node with at least one child missing — we will start from the node and traverse upwards, check for each node that its subtree either satisfies the required property or fix the subtree to re-instate this property.
  - This involves case-analysis, breaking the violations that may occur into sub-categories, and fixing each sub-category using <u>rotations</u>.

Balanced Binary Search Tree: Rotations

- Two types: Left-rotation and Right-rotation



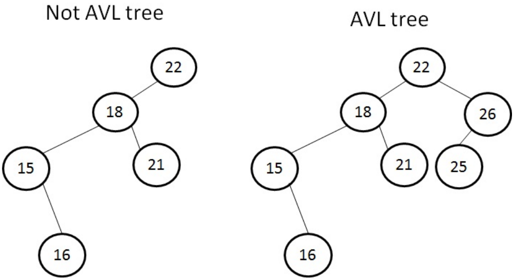
- Right-Rotate: the old-root becomes the right child of the new root.
- Left-Rotate: the old-root becomes the left child of the new root.

```
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 ≠ 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'd satisfy $h_L = h_R$
  - Impossible. 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 $|h_L - h_R| \leq 1$.
  - Not fully balanced, but almost



AVL Trees

- Def: An AVL tree is a BST where for any node we have $|h_L - h_R| \le 1$.
  - Not fully balanced, but almost
- Claim: If T is an AVL tree of height h then the number of nodes in T is at least $F(h+1)$ (= 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 $h_L \ge h_R$.
  - Hence, $h_L = h - 1$.
  - Because of AVL-property, $h_R \ge h_L - 1 = h - 2$.
  - $N(h) = 1 + N(h_L) + N(h_R) \ge 1 + N(h-1) + N(h-2)$, with $N(0) = 1$ and $N(1) = 2$.
  - (HW2) This solves to $N(h) \ge F(h+1) = \Theta((1.618...)^h)$.
- Corollary: $h \le O(\log(n))$
  - We know that $n \ge F(h+1) \ge 1.5^{h+1}$ (in fact, $F(h) \approx 1.618^h$), so $h \le \log_{1.5}(n) - 1 = O(\log(n))$
- 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

- Upon `Insert()`/`Delete()` we run the risk of incrementing / decrementing a subtree's height, causing some ancestor of the node we added/removed to be imbalanced.
- So, starting with the node we added/removed, an we climb up through the parent pointers to the root and see if any node along the way needs fixing.
- First step — re-evaluate height. Each node will maintain its height and the heights of the left/right subtrees.

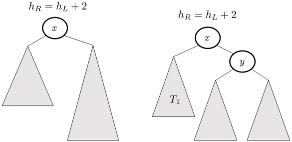
  procedure FindHeights(T)

  $T.h_L \leftarrow -1$
  if $(T.root.left \neq$ **nil**$)$ **then**
      $T.h_L \leftarrow T.root.left.height$
  $T.h_R \leftarrow -1$
  if $(T.root.right \neq$ **nil**$)$ **then**
      $T.h_R \leftarrow T.root.right.height$
  $T.height \leftarrow \max\{T.h_L, T.h_R\} + 1$

  Clearly, runs in $O(1)$-time.
- This makes detecting violation easy: after updating heights, we check if $|T.h_L - T.h_R| = 2$.
  - Note: before we added/removed a node, we had $|T.h_L - T.h_R| \leq 1$

How to Maintain the AVL Property

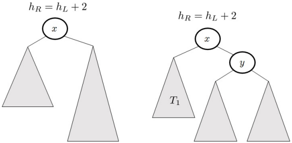
- We take an AVL tree, using standard BST for `Insert()`/`Delete()` we add/remove a node, and now we have caused a violation at node x — where all descendants of x have no violation.
- Assume that the right subtree is the higher one (other case is symmetric).



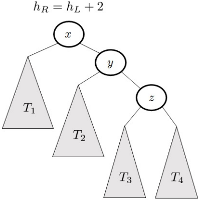
- $h_L \geq -1 \Rightarrow h_R \geq 1$. So not only does x has a right child, y, but also y's right or left subtrees aren't empty. Let z denote the root of y's highest subtree.
- How did we create this violation? (Recall, before we had $|h_L - h_R| \leq 1$)
- Insertion can only *increase* the height, so we must have increased h_R. Namely, we have inserted a node into the subtree rooted at z and thus incremented its height $+1$.
  In particular, in this case, out of the two subtrees of y, the one rooted in z must be higher than the subtree not rooted in z. (Why?)
- Deletion can only *decrease* the height, so must have removed a node from x's left subtree, decrementing the height of x's left subtree.
  In this case, both subtrees of y can be of the same height.

How to Maintain the AVL Property

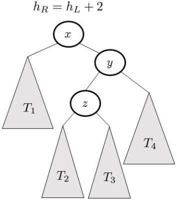
- We take an AVL tree, using standard BST for `Insert()`/`Delete()` we add/remove a node, and now we have caused a violation at some node — along the path from the added/removed node to the root.
- Assume that the right subtree is the higher one (other case is symmetric).



- $h_L \geq -1 \Rightarrow h_R \geq 1$. So not only does x has a right child, y, but also y's right or left subtrees aren't empty.
- Let z denote the root of y's highest subtree — breaking ties in favor of the outer subtree. So we split into cases:



Right-Right case

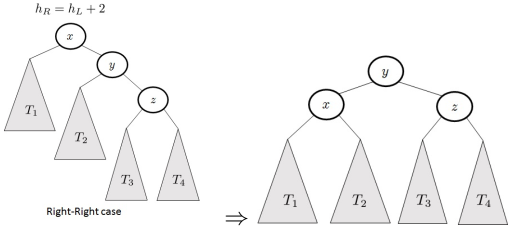


Right-Left case

All in all we have 4 potential cases: RR, RL, LR, LL

How to Maintain the AVL Property — Right-Right Case

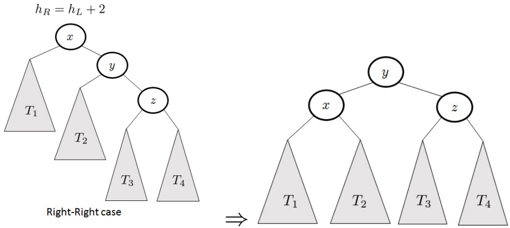
- We argue that the Right-Right case is easy: we can solve it with a single rotation. (On which node?)
- We do `LeftRotate`(x)



- Is this an AVL-tree?
- z was balanced and remains balanced — its height is $h_R - 1 = h_L + 1$.
- x is balanced: on the left side $h(T_1) = h_L$; on the right side: $h(T_2) \leq h_R - 1 = h_L + 1$ (as a descendant of y) but as y was balanced, $h(T_2) \geq h_R - 2 = h_L$.
- So now y is balanced — left subtree has height $\in \{h_L + 1, h_L + 2\}$, right subtree has height $h_R - 1 = h_L + 1$.
- AVL property has been restored!

How to Maintain the AVL Property — Right-Right Case

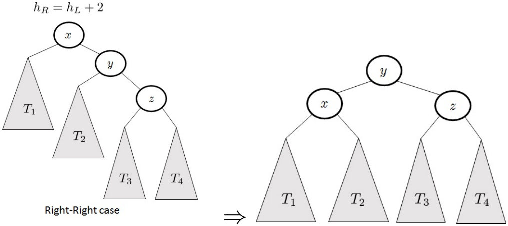
- We argue that the Right-Right case is easy: we can solve it with a single rotation. (On which nodes?)
- We do `LeftRotate(`x`)`



- Effects — `Insert()` case: we've increased the height of z's subtree by 1. I.e., h_L remained unchanged, it is h_R that has increased.
- Before insertion, height of x was $\max\{h_L, h_L + 1\} + 1 = h_L + 2$.
- As we commented, in the insertion case $height(T_2) < height(z) = h_R - 1 = h_L + 1$. So $height(T_2) = h_L$.
- Now height of $y = \max\{(\max\{h_L, h_L\} + 1), h_L + 1\} + 1 = h_L + 2$.
- I.e., this subtree has the same height as before the insertion. AVL property is therefore maintained in all of the tree's ancestors. No need to check any further up!

How to Maintain the AVL Property — Right-Right Case

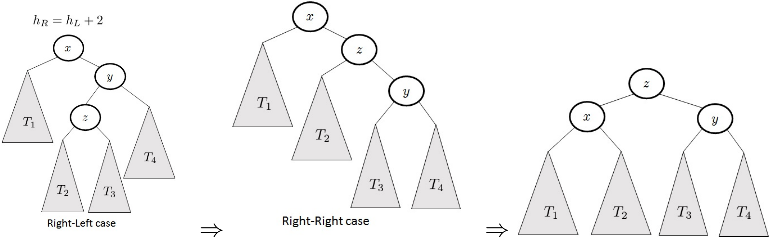
- We argue that the Right-Right case is easy: we can solve it with a single rotation. (On which nodes?)
- We do `LeftRotate(`x`)`



- Effects — `Delete()` case: we've decreased the height of x's left subtree by 1. I.e., h_R remains the same, it is h_L that has decreased.
- Before insertion: Height of y was h_R, height of z was $h_R - 1$. Height of x was $h_R + 1$.
- In the deletion case, $height(T_2) \in \{height(z), height(z) - 1\} = \{h_R - 1, h_R - 2\}$.
- If $height(T_2) = h_R - 2$ we get that height of y is $\max\{(\max\{h_R - 2, h_R - 2\} + 1), h_R - 1\} + 1 = h_R$.
- I.e., this subtree has smaller height then before the deletion. AVL property is therefore uncertain for the tree's ancestors. We must continue climb up to y's new parent and check further violations!

How to Maintain the AVL Property — Right-Left Case

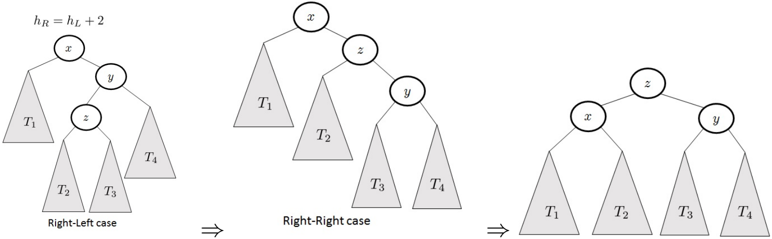
- To deal with the Right-Left case, we first convert it to a Right-Right case, then solve the Right-Right case.
- Namely, do (1) `RightRotate(`y`)`, and then (2) `LeftRotate(`x`)`



- Is this an AVL-tree? Is balance restored?
- z had height $h_R - 1$ and was balanced, so $h(T_2)$ is either $h_R - 2 = h_L$ or $h_R - 3 = h_L - 1 \Rightarrow x$ is balanced.
- y satisfied the AVL property, and z's subtree is strictly higher than T_4. So $h(T_4) = h_R - 2 = h_L$; $h_R - 3 \leq h(T_3) \leq h_R - 2$. Thus y is balanced.
- Moreover, z is balanced — height of left subtree: $h_L + 1$; height of right subtree: $h_L + 1$.
- And the overall height of this tree: $h_L + 2 = h_R$.

How to Maintain the AVL Property — Right-Left Case

- To deal with the Right-Left case, we first convert it to a Right-Right case, then solve the Right-Right case.
- Namely, do (1) `RightRotate`(y), and then (2) `LeftRotate`(x)



- Effects — `Insert()` case: we've increased the height of z's subtree by 1. I.e., h_L remained unchanged, it is h_R that has incremented.
  - Before insertion: Height of x was $(h_L + 1) + 1 = h_L + 2$.
  - I.e., this tree has the same height as before the insertion. AVL property is therefore maintained in all of the tree's ancestors. No need to check any further up!
- Effects — `Delete()` case: we've decremented height of x's left subtree. I.e., h_R remained unchanged, it is h_L that was decremented.
  - Before insertion: Height of x was $h_R + 1$.
  - I.e., this tree has smaller height then before the deletion. AVL property is therefore uncertain for the tree's ancestors. We must continue climb up to z's new parent and check further violations!

How to Maintain the AVL Property — Rebalancing

- So the code looks like:

```
Rebalance(T, toRecurse)
** toRecurse – a boolean flag: whether to recurse on ancestors after balancing.
if (T =nil) then
   return
FindHeights(T)
if (T.h_L − T.h_R = 2) then
   y ← T.root.left
   if (y.h_R > y.h_L) then
      LeftRotate(y)      ** case Left-Right
   RightRotate(T)      ** case Left-Left or continuing case Left-Right
   if (toRecurse =TRUE) then
      Rebalance(y.parent, toRecurse)
else if (T.h_R − T.h_L = 2) then
   y ← T.root.right
   if (y.h_L > y.h_R) then
      RightRotate(y)      ** case Right-Left
   LeftRotate(T)      ** case Right-Right or continuing case Right-Left
   if (toRecurse =TRUE) then
      Rebalance(y.parent, toRecurse)
else      ** There isn't a violation on T, but might be on its parent
   Rebalance(T.parent, toRecurse)
```

- Runtime $= O(h) = O(\log(n))$.

How to Maintain the AVL Property — Rebalancing

- So the code looks like:

```
Rebalance(T, toRecurse)
** toRecurse – a boolean flag: whether to recurse on ancestors after balancing.
if (T =nil) then
   return
FindHeights(T)
if (T.h_L − T.h_R = 2) then
   y ← T.root.left
   if (y.h_R > y.h_L) then
      LeftRotate(y)     ** case Left-Right
   RightRotate(T)      ** case Left-Left or continuing case Left-Right
   if (toRecurse =TRUE) then
      Rebalance(y.parent, toRecurse)
else if (T.h_R − T.h_L = 2) then
   y ← T.root.right
   if (y.h_L > y.h_R) then
      RightRotate(y)     ** case Left-Right
   LeftRotate(T)      ** case Left-Left or continuing case Left-Right
   if (toRecurse =TRUE) then
      Rebalance(y.parent, toRecurse)
else      ** There isn't a violation on T, but might be on its parent
   Rebalance(T.parent, toRecurse)
```

- `Insert()`: call `Rebalance(`T`,FALSE)` when inserting a leaf, on its parent.
- `Delete()`: call `Rebalance(`T`,TRUE)` on the deleted node's parent (deleted node = the node with at most one child).

Red-Black Trees

- Another BST that achieves balancing — this time using colors.
- The RB-property actually breaks down to 4 properties:
  1. Every node has a color: either red or black.
  2. Root and `nil` nodes are colored black.[1]
  3. No red node is a parent of another red node. (2Reds2Many)
  4. Each root→leaf path in the tree has the same number of black nodes. (Black heights matter!)
- <u>Claim:</u> A RB-tree with n nodes height at most $2\log_2(n+1)$.
- <u>Proof:</u> Fix any RB-tree, let h denote its height and n its size.
  - On the root→leaf path of length h, there are $h+1 \geq h$ nodes.
  - Clearly, the path has no two consecutive red nodes (property 3). So the number of black nodes is $\geq \frac{h}{2}$.
  - So property 4 assures that all root→leaf paths have at least $\frac{h}{2}$ black nodes.
  - In particular, all root→leaf paths are of length $\geq \frac{h}{2}$.I.e., the first $\frac{h}{2}$ layers in the tree are full.
  - Thus

$$n \geq 1+2+4+8+...+2^{\frac{h}{2}-1} = 2^{\frac{h}{2}} - 1 \qquad \Rightarrow \qquad \log_2(n+1) \geq \frac{h}{2} \qquad \square$$

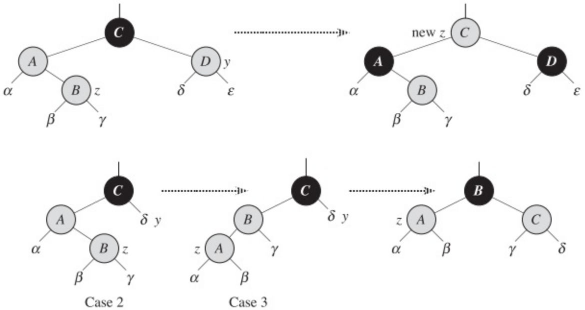
- How to maintain the BR-tree properties upon insertion / deletion?

[1] Merely a convention. Easier for us to assume `nil` nodes exist and are black, with black children, rather than break the already quite involved case-analysis into further subcases based on which nodes are `nil` and which aren't.

Red-Black Trees — Insertion

- Typically, a new leaf is given the color red.
- If it's the very first node (T's root) — just color it black. Done.
- Only property 3 can be violated: so if the new leaf is z, then its parent y is red and we need to fix it up.
- Due to property 2, the red parent y has to have a parent x.
  Due to property 3, x must be black.
- So now it comes to z's uncle u (the non-y child of x) —
  - If u is red — we just flip the colors: color y and u black and color x red, and recurse up the fix-up.
  - If u is black and z is in the same side of y as y is to x (I.e, Left-Left / Right-Right case) — we rotate(x) to make y the parent, then flip colors of x and y.
  - If u is black and z is in the opposite side of y as y is to x (I.e, Left-Right / Right-Left case) — we rotate(y) first to make it like the Left-Left / Right-Right case, then we rotate(x) to make z the parent, then flip colors of x and z.

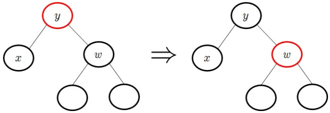
Red-Black Trees — Insertion

- 

    - If u is red — we just flip the colors: color y and u black and color x red, and recurse up the fix-up.
    - If u is black and z is in the same side of y as y is to x (I.e, Left-Left / Right-Right case) — we rotate(x) to make y the parent, then flip colors of x and y.
    - If u is black and z is in the opposite side of y as y is to x (I.e, Left-Right / Right-Left case) — we rotate(y) first to make it like the Left-Left / Right-Right case, then we rotate(x) to make z the parent, then flip colors of x and z.
- Very similar to AVL insert. Main exception: we spare a rotation if the color of z's uncle u is red.

Red-Black Trees — Deletion

- BST's delete eventually deletes a node with at least one child missing.
- If this node is red — no violation occurs.
- If this node is black, we start the fix-up with the deleted node's child (could be a black `nil`)
- Let's deal with one easy case: x is red (and isn't `nil`) — just color it black and halt. We've compensated for the one black node we've lost.
- Another easy case: if x is the root of the tree — just color it black.
- Here's one more easy case: x is black, x's parent y is red and x's sibling w (**Why must w exist?**) is (i) black and (ii) with only black children: This is easy because all we need to do is to replace the parent and w's color.



  - y's subtree on x side lost a black node; y's subtree on w's side also lost a black node, and by coloring y black we compensate for both losses. (No need even to recurse from y.)
- Here's a slight complication: y is black already, but w satisfies the two properties.
  Well, we will still color w red, and then we do need to recurse upwards from y.
- True complications: either (i) w isn't black, or (ii) w has a red child.

Red-Black Trees — Deletion

- BST deletion deletes a node with at least one child missing.
- If this node is red — no violation occurs.
- If this node is black, we start the fix-up with the deleted node's child (could be a black `nil`)
- If it is red — color it black.
- If we're fixing the root of the tree — just color it black.
- O/w, we're looking into x's sibling w.

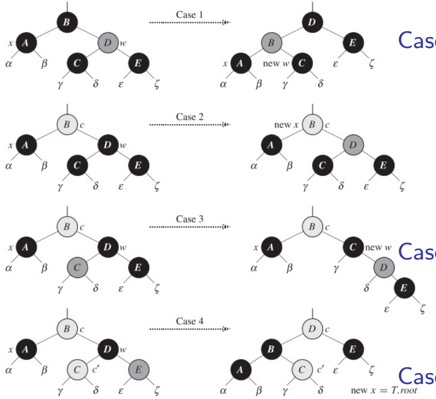
  Case 1. <u>w is red:</u> Color w black, color its parent red, and `Rotate` the parent to so now x's sibling is black (check cases $2, 3$ or 4)

  Case 2. <u>black w has only black children:</u> Color w red. If w's (and x's) parent is red – color w's parent black and this compensates for the black node we removed; If w's (and x's) parent is black — recurse the fix-up on it.

  Case 3. <u>w is black and its child in the reverse direction is red:</u> Rotate on w and flip w and its (new) parent color to make it case 4.

  Case 4. <u>w is black and its child in the same direction is red:</u> Give w the color of w's parent; rotate w's parent (so w is the new root of this subtree); color both children of w black.

Red-Black Trees — Deletion



Case 1. w is red: Color w black, color its parent red, and `Rotate` the parent to so now x's sibling is black (check cases $2, 3$ or 4)

Case 2. black w has only black children: Color w red.
If w's (and x's) parent is red – color w's parent black and this compensates for the black node we removed; If w's (and x's) parent is black — recurse the fix-up on it.

Case 3. w is black and its child in the reverse direction is red: Rotate on w and flip w and its (new) parent color to make it case 4.

Case 4. w is black and its child in the same direction is red: Give w the color of w's parent; rotate w's parent (so w is the new root); color both children of w black.

AVL and Red-Black Trees

- Both used in practice. RB-tree tends to be used more often.
- AVL tends to create shallower trees, but tends to make more rotations.
- So `Find()` operations are less costly for AVL-trees
  - Less costly – in terms of empirical experiments, not asymptotic notation
- RB-trees avoid some rotations by flipping colors. In fact, it makes at most $O(1)$ rotations per `Insert()` / `Delete()` (but could make more color-alterations).
- For our needs — both guarantee $h = O(\log(n))$ so both are fine.

- For those who care solely about the final exam — not going to ask you to remember the different cases and the remedy in each case.
  Might ask about the high-level details (e.g., when to recurse on parent and when not-to, why must there be a sibling, etc.)

Summary:

- Heaps / BST — two data-structures that are based on trees.
- Min- / Max-Heaps: root is the smallest / largest element in its subtree.
  - We can build a heap in $O(n)$ time.
  - We can extract the smallest/largest element in $O(\log(n))$ time.
  - Thus we get the $O(n + n\log(n)) = O(n\log(n))$-time HeapSort algorithm.
- Priority-Queues: updating heaps via inserting/changing a key.
- BST — keys in left subtree $\leq$ root $<$ keys in right subtree.
  - Finding — compare to key, move to left/right subtree accordingly.
  - Inserting — a new leaf
  - Deleting — delete a node with (at least) one child missing is easy; to delete a node with two children replace it with the maximum of its left subtree.
  - Predecessor() / Successor()
  - All take $O(h)$-time.
  - In-Order traversal / printing — takes $O(n)$ time.
- AVL tree and RB-trees are self-balancing trees with $h = O(\log(n))$.
  - AVL-tree — left-height and right-height differ by at most one.
  - RB-tree — (i) nodes are either black or red, (ii) root and `nils` are black, (iii) no red-red edge, (iv) all root$\rightarrow$leaf paths have same black-height.
- Inserting/Deleting now involve case-dependent `Rotation()` operations.