Intuition for Asymptotic Notation

**Big-Oh**
- $f(n)$ is $O(g(n))$ if $f(n)$ is asymptotically less than or equal to $g(n)$

**big-Omega**
- $f(n)$ is $\Omega(g(n))$ if $f(n)$ is asymptotically greater than or equal to $g(n)$

**big-Theta**
- $f(n)$ is $\Theta(g(n))$ if $f(n)$ is asymptotically equal to $g(n)$

**little-oh**
- $f(n)$ is $o(g(n))$ if $f(n)$ is asymptotically strictly less than $g(n)$

**little-omega**
- $f(n)$ is $\omega(g(n))$ if $f(n)$ is asymptotically strictly greater than $g(n)$

Analysis of Merge-Sort

- The height $h$ of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The overall amount of work done at the nodes of depth $i$ is $O(n)$
  - we partition and merge $2^i$ sequences of size $n/2^i$
  - we make $2^i+1$ recursive calls
- Thus, the total running time of merge-sort is $O(n \log n)$

<table>
<thead>
<tr>
<th>depth</th>
<th>#steps</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>$n$</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>$n/2$</td>
</tr>
<tr>
<td>$i$</td>
<td>$2^i$</td>
<td>$n/2^i$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Tools: Recurrence Equation Analysis

- The final step of merge-sort consists of merging two sorted sequences, each with $n/2$ elements. It takes at most $b$ steps, for some constant $b$.
- Likewise, the basis case ($n < 2$) will take at most $b$ steps.
- Therefore, if we let $T(n)$ denote the running time of merge-sort:

$$T(n) = \begin{cases} 
  b & \text{if } n < 2 \\
  2T(n/2) + bn & \text{if } n \geq 2
\end{cases}$$

- We can therefore analyze the running time of merge-sort by finding a closed form solution to the above equation.
  - That is, a solution that has $T(n)$ only on the left-hand side.
The Recursion Tree

- Draw the recursion tree for the recurrence relation and look for a pattern:

\[
T(n) = \begin{cases} 
  b & \text{if } n < 2 \\
  2T(n/2) + bn & \text{if } n \geq 2 
\end{cases}
\]

<table>
<thead>
<tr>
<th>Depth</th>
<th>T's size</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>n</td>
<td>hn</td>
</tr>
<tr>
<td>1</td>
<td>n/2</td>
<td>bn</td>
</tr>
<tr>
<td>i</td>
<td>n/2^i</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Total time = \( bn + bn \log n \)

Iterative Substitution

- In the iterative substitution, or “plug-and-chug,” technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern:

\[
T(n) = 2T(n/2) + bn \\
= 2(2T(n/2) + bn) + bn \\
= 2^2T(n/2^2) + 2bn \\
= 2^2T(n/2^2) + 3bn \\
= 2^3T(n/2^3) + 4bn \\
= ... \\
= 2^iT(n/2^i) + ibn \\
\]

- Note that base, \( T(n) = b \), case occurs when \( 2^i = n \). That is, \( i = \log n \).
- So, \( T(n) = bn + bn \log n \)
- Thus, \( T(n) \) is \( O(n \log n) \).

Guess-and-Test Method

- In the guess-and-test method, we guess a closed form solution and then try to prove it is true by induction:

\[
T(n) = \begin{cases} 
  b & \text{if } n < 2 \\
  2T(n/2) + bn \log n & \text{if } n \geq 2 
\end{cases}
\]

- Guess: \( T(n) < cn \log n \)

\[
T(n) = 2T(n/2) + bn \log n \\
= 2(c(n/2) \log(n/2)) + bn \log n \\
= cn \log n - cn + bn \log n \\
= cn \log n - cn + bn \log n
\]

- Wrong: we cannot make this last line be less than \( cn \log n \)
Guess-and-Test Method, Part 2

- Recall the recurrence equation:
  \[ T(n) = \begin{cases} 
  b & \text{if } n < 2 \\
  2T(n/2) + bn \log n & \text{if } n \geq 2 
\end{cases} \]

- Guess #2: \( T(n) < cn \log^2 n \).
- \( T(n) = 2T(n/2) + bn \log n \)
  \[ = 2c(n/2) \log^2 (n/2) + bn \log n \]
  \[ = cn \log n - \log 2 + bn \log n \]
  \[ = cn \log^2 n - 2cn \log n + cn + bn \log n \]
  \[ \leq cn \log^2 n \]

So, \( T(n) \) is \( O(n \log^2 n) \).
- In general, to use this method, you need to have a good guess and you need to be good at induction proofs.

Master Method (Chapter 5)

- Many recurrence equations have the form:
  \[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases} \]

- The Master Theorem:
  1. if \( f(n) \) is \( \Theta(n^{\log_b a - \epsilon}) \), then \( T(n) = \Theta(n^{\log_b a}) \)
  2. if \( f(n) \) is \( \Theta(n^{\log_b a} \log^k n) \), then \( T(n) = \Theta(n^{\log_b a} \log^k n) \)
  3. if \( f(n) \) is \( \Theta(n^{\log_b a + \epsilon}) \), then \( T(n) = \Theta(n^{\log_b a} f(n)) \)
  provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

Master Method, Example 1

- The form: \( T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases} \)

- The Master Theorem:
  1. if \( f(n) \) is \( \Theta(n^{\log_b a - \epsilon}) \), then \( T(n) = \Theta(n^{\log_b a}) \)
  2. if \( f(n) \) is \( \Theta(n^{\log_b a} \log^k n) \), then \( T(n) = \Theta(n^{\log_b a} \log^k n) \)
  3. if \( f(n) \) is \( \Theta(n^{\log_b a + \epsilon}) \), then \( T(n) = \Theta(n^{\log_b a} f(n)) \)
  provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

- Example:
  \[ T(n) = 4T(n/2) + n \]

Solution: \( \log_b a = 2 \), so Case 1 says \( T(n) \) is \( \Theta(n^2) \).
Master Method, Example 2

- The form: 
  \[ T(n) = \begin{cases} 
  c & \text{if } n < d, \\
  aT(n/b) + f(n) & \text{if } n \geq d 
  \end{cases} \]

- The Master Theorem:
  1. if \( f(n) = O(n^{d-\epsilon}) \), then \( T(n) = \Theta(n^{d}) \)
  2. if \( f(n) = \Theta(n^{d-\epsilon} \log^{k+1} n) \), then \( T(n) = \Theta(n^{d} \log^{k} n) \)
  3. if \( f(n) = \Omega(n^{d+\epsilon}) \), then \( T(n) = \Theta(f(n)) \), provided \( af(n/b) \leq \Theta(f(n)) \) for some \( \delta < 1 \).

- Example: 
  \[ T(n) = 2T(n/2) + n \log n \]
  Solution: \( \log_{a}c = 1 \), so case 2 says \( T(n) = O(n \log^{2} n) \).

Master Method, Example 3

- The form: 
  \[ T(n) = \begin{cases} 
  c & \text{if } n < d, \\
  aT(n/b) + f(n) & \text{if } n \geq d 
  \end{cases} \]

- The Master Theorem:
  1. if \( f(n) = O(n^{d-\epsilon}) \), then \( T(n) = \Theta(n^{d}) \)
  2. if \( f(n) = \Theta(n^{d-\epsilon} \log^{k+1} n) \), then \( T(n) = \Theta(n^{d} \log^{k} n) \)
  3. if \( f(n) = \Omega(n^{d+\epsilon}) \), then \( T(n) = \Theta(f(n)) \), provided \( af(n/b) \leq \Theta(f(n)) \) for some \( \delta < 1 \).

- Example: 
  \[ T(n) = T(n/3) + n \log n \]
  Solution: \( \log_{a}c = 0 \), so case 3 says \( T(n) = O(n \log n) \).

Master Method, Example 4

- The form: 
  \[ T(n) = \begin{cases} 
  c & \text{if } n < d, \\
  aT(n/b) + f(n) & \text{if } n \geq d 
  \end{cases} \]

- The Master Theorem:
  1. if \( f(n) = O(n^{d-\epsilon}) \), then \( T(n) = \Theta(n^{d}) \)
  2. if \( f(n) = \Theta(n^{d-\epsilon} \log^{k+1} n) \), then \( T(n) = \Theta(n^{d} \log^{k} n) \)
  3. if \( f(n) = \Omega(n^{d+\epsilon}) \), then \( T(n) = \Theta(f(n)) \), provided \( af(n/b) \leq \Theta(f(n)) \) for some \( \delta < 1 \).

- Example: 
  \[ T(n) = 8T(n/2) + n^{2} \]
  Solution: \( \log_{a}c = 3 \), so case 1 says \( T(n) = O(n^{3}) \).
Master Method, Example 5

- The form: 
  \[ T(n) = \begin{cases} 
  cn & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
  \end{cases} \]

- The Master Theorem:
  1. If \( f(n) = O(n^{\log_b a}) \), then \( T(n) = \Theta(n^{\log_b a}) \)
  2. If \( f(n) = \Theta(n^{\log_b a} \log^k n) \), then \( T(n) = \Theta(n^{\log_b a} \log^{k+1} n) \)
  3. If \( f(n) = \Omega(n^{\log_b a + \varepsilon}) \), then \( T(n) = \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

- Example:
  \[ T(n) = 9T(n/3) + n^3 \]
  Solution: \( \log_3 n = 2 \), so case 3 says \( T(n) = \Theta(n^3) \).

---

Master Method, Example 6

- The form: 
  \[ T(n) = \begin{cases} 
  cn & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
  \end{cases} \]

- The Master Theorem:
  1. If \( f(n) = O(n^{\log_b a}) \), then \( T(n) = \Theta(n^{\log_b a}) \)
  2. If \( f(n) = \Theta(n^{\log_b a} \log^k n) \), then \( T(n) = \Theta(n^{\log_b a} \log^{k+1} n) \)
  3. If \( f(n) = \Omega(n^{\log_b a + \varepsilon}) \), then \( T(n) = \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

- Example:
  \[ T(n) = T(n/2) + 1 \] (binary search)
  Solution: \( \log_2 n = 0 \), so case 2 says \( T(n) = \Theta(\log n) \).

---

Master Method, Example 7

- The form: 
  \[ T(n) = \begin{cases} 
  cn & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
  \end{cases} \]

- The Master Theorem:
  1. If \( f(n) = O(n^{\log_b a}) \), then \( T(n) = \Theta(n^{\log_b a}) \)
  2. If \( f(n) = \Theta(n^{\log_b a} \log^k n) \), then \( T(n) = \Theta(n^{\log_b a} \log^{k+1} n) \)
  3. If \( f(n) = \Omega(n^{\log_b a + \varepsilon}) \), then \( T(n) = \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

- Example:
  \[ T(n) = 2T(n/2) + \log n \] (heap construction)
  Solution: \( \log_2 n = 1 \), so case 1 says \( T(n) = \Theta(n) \).
Iterative “Proof” of the Master Theorem

- Using iterative substitution, let us see if we can find a pattern:
  \[ T(n) = aT(n/b) + f(n) \]
  \[ = a(aT(n/b^2) + f(n/b)) + bn \]
  \[ = a^2T(n/b^2) + af(n/b) + f(n) \]
  \[ = a^nT(1) + \sum_{i=0}^{n-1} a^i f(n/b^i) \]
  \[ = n^nT(1) + \sum_{i=0}^{n-1} a^i f(n/b^i) \]

- We then distinguish the three cases as:
  - The first term is dominant
  - Each part of the summation is equally dominant
  - The summation is a geometric series

Case Study

Priority Queue Sorting

Priority Queue ADT

- A priority queue stores a collection of items
- An item is a pair (key, element)
- Main methods of the Priority Queue ADT
  - `insert(key, e)`
    - inserts an item with key k and element e
  - `removeMin()`
    - removes the item with smallest key and returns its element
### Sorting with a Priority Queue

We can use a priority queue to sort a set of comparable elements:

1. Insert the elements one by one with a series of insertItem(e, e) operations.
2. Remove the elements in sorted order with a series of removeMin() operations.

*The running time of this sorting method depends on the priority queue implementation.*

```plaintext
Algorithm PQ-Sort(S, C)
Input: sequence S, comparator C for the elements of S
Output: sequence S sorted in increasing order according to C

P ← priority queue with comparator C

while ¬S.isEmpty()

    e ← S.remove(S.first())
    P.insertItem(e, e)

while ¬P.isEmpty()

    e ← P.removeMin()
    S.insertLast(e)
```

### Sequence-based Priority Queue

- Implementation with an unsorted sequence:
  - Store the items of the priority queue in a list-based sequence, in arbitrary order.

- Performance:
  - insertItem takes $O(1)$ time since we can insert the item at the beginning or end of the sequence.
  - removeMin, minKey and minElement take $O(n)$ time since we have to traverse the entire sequence to find the smallest key.

### Selection-Sort

- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence.

- Running time of Selection-sort:
  1. Inserting the elements into the priority queue with $n$ insertItem operations takes $O(n)$ time.
  2. Removing the elements in sorted order from the priority queue with $n$ removeMin operations takes time proportional to $1 + 2 + \ldots + n$.

- Selection-sort runs in $O(n^2)$ time.
Sequence-based Priority Queue

- Implementation with a sorted sequence
  - Store the items of the priority queue in a sequence, sorted by key
- Performance:
  - insertItem takes $O(n)$ time since we have to find the place where to insert the item
  - removeMin, minKey and minElement take $O(1)$ time since the smallest key is at the beginning of the sequence

Insertion-Sort

- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence
- Running time of Insertion-sort:
  1. Inserting the elements into the priority queue with $n$ insertItem operations takes time proportional to $1 + 2 + \ldots + n$
  2. Removing the elements in sorted order from the priority queue with a series of $n$ removeMin operations takes $O(n)$ time
- Insertion-sort runs in $O(n^2)$ time

Heaps and Priority Queues
What is a heap (§2.4.3)

- A heap is a binary tree storing keys at its internal nodes and satisfying the following properties:
  - Heap-Order: for every internal node v other than the root, keys(v) ≥ keys(parent(v))
  - Complete Binary Tree: let h be the height of the heap
    - For i = 0, ..., h − 1, there are 2^i nodes at depth i
    - At depth h − 1, the internal nodes are to the left of the external nodes

- The last node of a heap is the rightmost internal node of depth h − 1

Height of a Heap (§2.4.3)

- Theorem: A heap storing n keys has height O(log n)
  - Proof: (we apply the complete binary tree property)
    - Let h be the height of a heap storing n keys
    - Since there are 2^i keys at depth i = 0, ..., h − 2 and at least one key at depth h − 1, we have n ≥ 2^1 + 2^2 + 2^3 + ... + 2^(h − 1) + 1
    - Thus, n ≤ 2^h − 1, i.e., h ≤ log n + 1

Heaps and Priority Queues

- We can use a heap to implement a priority queue
- We store a (key, element) item at each internal node
- We keep track of the position of the last node
**Insertion into a Heap (§2.4.3)**

- Method `insertItem` of the priority queue ADT corresponds to the insertion of a key \( k \) to the heap.
- The insertion algorithm consists of three steps:
  - Find the insertion node \( z \) (the new last node).
  - Store \( k \) at \( z \) and expand \( z \) into an internal node.
  - Restore the heap-order property (discussed next).

**Upheap**

- After the insertion of a new key \( k \), the heap-order property may be violated.
- Algorithm `upheap` restores the heap-order property by swapping \( k \) along an upward path from the insertion node.
- Upheap terminates when the key \( k \) reaches the root or a node whose parent has a key smaller than or equal to \( k \).
- Since a heap has height \( O(\log n) \), upheap runs in \( O(\log n) \) time.

**Removal from a Heap (§2.4.3)**

- Method `removeMin` of the priority queue ADT corresponds to the removal of the root key from the heap.
- The removal algorithm consists of three steps:
  - Replace the root key with the key of the last node \( w \).
  - Compress \( w \) and its children into a leaf.
  - Restore the heap-order property (discussed next).
Downheap

- After replacing the root key with the key $k$ of the last node, the heap-order property may be violated.
- Algorithm downheap restores the heap-order property by swapping key $k$ along a downward path from the root.
- Upheap terminates when key $k$ reaches a leaf or a node whose children have keys greater than or equal to $k$.
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time.

---

Heap Sort

<table>
<thead>
<tr>
<th>Method</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertItem</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>minKey, minElement</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>removeMin</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

- All heap methods run in logarithmic time or better.
- Thus each phase takes $O(n \log n)$ time, so the algorithm runs in $O(n \log n)$ time also.
- This sort is known as **heap-sort**.
- The $O(n \log n)$ run time of heap-sort is much better than the $O(n^2)$ run time of selection and insertion sort.

---

Heap-Sort (§2.4.4)

- Consider a priority queue with $n$ items implemented by means of a heap.
  - the space used is $O(n)$
  - methods insertItem and removeMin take $O(\log n)$ time
- methods size, isEmpty, minKey, and minElement take time $O(1)$ time
- Using a heap-based priority queue, we can sort a sequence of $n$ elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort.
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort.
Bottom-Up Heap Construction
Algorithm (§2.4.3)

Algorithm BottomUpHeap(S)
Input: A sequence S storing \( n = 2^h - 1 \) keys
Output: A heap \( T \) storing keys in \( S \)
if \( S \) is empty then
return an empty heap
Remove the first key, \( k \), from \( S \)
Split \( S \) into 2 sequences, \( S_1 \) and \( S_2 \), each of size \((n-1)/2\)
\( T_1 \) ← BottomUpHeap\((S_1)\)
\( T_2 \) ← BottomUpHeap\((S_2)\)
Create binary tree \( T \) with root \( r \) storing \( k \), left subtree \( T_1 \) and right subtree \( T_2 \)
Perform a down-heap bubbling from root \( r \) of \( T \)
return \( T \)

Example

- \( S = [10, 7, 25, 16, 15, 5, 4, 12, 8, 11, 6, 7, 27, 23, 24] \)
Binary Search Trees

Binary Search (§3.1.1)

- Binary search performs operation findElement(k) on a dictionary implemented by means of an array-based sequence, sorted by key
  - similar to the high-low game
  - at each step, the number of candidate items is halved
  - terminates after $O(\log n)$ steps
- Example: findElement(7)

Lookup Table (§3.1.1)

- A lookup table is a dictionary implemented by means of a sorted sequence
  - We store the items of the dictionary in an array-based sequence, sorted by key
  - We use an external comparator for the keys
- Performance:
  - findElement takes $O(\log n)$ time using binary search
  - insertItem takes $O(n)$ time since in the worst case we have to shift $n/2$ items to make room for the new item
  - removeElement takes $O(n)$ time since in the worst case we have to shift $n/2$ items to compact the items after the removal
- The lookup table is effective only for dictionaries of small size or for dictionaries on which searches are the most common operations, while insertions and removals are rarely performed (e.g., credit card authorizations)
Binary Search Tree (§3.1.2)

- A binary search tree is a binary tree storing keys (or key-element pairs) at its internal nodes and satisfying the following property:
  - Let $u$, $v$, and $w$ be three nodes such that $u$ is in the left subtree of $v$ and $w$ is in the right subtree of $v$. We have $\text{key}(u) \leq \text{key}(v) \leq \text{key}(w)$
- External nodes do not store items

An inorder traversal of a binary search tree visits the keys in increasing order.

---

recap: Tree Terminology

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (a.k.a. leaf): node without children (E, I, J, G, H, D)
- Ancestors of a node: parent, grandparent, great-grandparent, etc.
- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node (3)
- Descendant of a node: child, grandchild, great-grandchild, etc.

Subtree: tree consisting of a node and its descendants

---

recap: Binary Tree

- A binary tree is a tree with the following properties:
  - Each internal node has two children
  - The children of a node are an ordered pair
- We call the children of an internal node left child and right child
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of children, each of which is a binary tree

Applications:
- arithmetic expressions
- decision processes
- searching
**recap: Properties of Binary Trees**

- **Notation**
  - $n$: number of nodes
  - $e$: number of external nodes
  - $i$: number of internal nodes
  - $h$: height

- **Properties**:
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$
  - $h \geq \log_2 e$
  - $h \geq \log_2 (n + 1) - 1$

---

**recap: Inorder Traversal**

- **In an inorder traversal a node is visited after its left subtree and before its right subtree**
- **Application**: draw a binary tree
  - $v(x)$ = inorder rank of $v$
  - $y(v)$ = depth of $v$

**Algorithm**

```
inOrder(v)
if isInternal(v) then
  inOrder(leftChild(v))
visits(v)
if isInternal(v) then
  inOrder(rightChild(v))
```

---

**recap: Preorder Traversal**

- **In a preorder traversal, a node is visited before its descendants**

**Algorithm**

```
preOrder(v)
visit(v)
for each child $w$ of $v$
  preorder(w)
```
recap: Postorder Traversal

- In a postorder traversal, a node is visited after its descendants

\[
\text{Algorithm postOrder}(v) \\
\text{for each child } w \text{ of } v \\
\text{postOrder}(w) \\
\text{visit}(v)
\]

Search (§3.1.3)

- To search for a key \( k \), we trace a downward path starting at the root.
- The next node visited depends on the outcome of the comparison of \( k \) with the key of the current node.
- If we reach a leaf, the key is not found and we return NO_SUCH_KEY.
- Example: findElement(4)

\[
\text{Algorithm findElement}(k, v) \\
\text{if } T.isExternal(v) \text{ return NO_SUCH_KEY} \\
\text{if } k < \text{key}(v) \text{ return findElement}(k, T.leftChild(v)) \\
\text{else if } k = \text{key}(v) \text{ return element}(v) \\
\text{else } \{ k > \text{key}(v) \} \text{ return findElement}(k, T.rightChild(v))
\]

Insertion (§3.1.4)

- To perform operation insertItem(\( k \)), we search for key \( k \).
- Assume \( k \) is not already in the tree, and let \( w \) be the leaf reached by the search.
- We insert \( k \) at node \( w \) and expand \( w \) into an internal node.
- Example: insert 5
Deletion (§3.1.5)

- To perform operation `removeElement(k)`, we search for key `k`.
- Assume key `k` is in the tree, and let `v` be the node storing `k`.
- If node `v` has a leaf child `w`, we remove `v` and `w` from the tree with operation `removeAboveExternal(w)`.
- Example: remove 4

Deletion (cont.)

- We consider the case where the key `k` to be removed is stored at a node `v` whose children are both internal.
  - We find the internal node `w` that follows `v` in an inorder traversal.
  - We copy key `w` into node `v`.
  - We remove node `w` and its left child `z` (which must be a leaf) by means of operation `removeAboveExternal(z)`.
- Example: remove 3

Time Complexity

- A search, insertion, or removal, visits the nodes along a root-to-leaf path.
- Time O(1) is spent at each node.
- The running time of each operation is O(h), where `h` is the height of the tree.
- Height of a balanced search tree: log(n).
- Unfortunately: The height of binary search tree can be `n` in the worst case.

- To achieve good running time, we need to keep the tree balanced, i.e., with O(log n) height.
AVL Trees

AVL Tree Definition

- AVL trees are balanced.
- An AVL Tree is a binary search tree such that for every internal node \( v \) of \( T \), the heights of the children of \( v \) can differ by at most 1.

An example of an AVL tree where the heights are shown next to the nodes:

Height of an AVL Tree

- Proposition: The height of an AVL tree \( T \) storing \( n \) keys is \( O(\log n) \).
- Justification:
  - Let \( n(h) \) be the minimum number of internal nodes of an AVL tree of height \( h \).
  - We see that \( n(1) = 1 \) and \( n(2) = 2 \).
  - For \( h \geq 3 \):
    - an AVL tree contains the root node
    - one AVL subtree of height \( h-1 \) and
    - the other AVL subtree of height \( h-2 \).
    - i.e., \( n(h) = 1 + n(h-1) + n(h-2) \)
Height of an AVL Tree (cont)

- Knowing $n(h-1) > n(h-2)$, we get $n(h) > 2n(h-2)$
- $n(h) > 2^{(h-2)}$
- $n(h) > 2^{h-2}$

Solving the base case we get: $n(h) \geq 2^{h/2}$
- Taking logarithms: $h < 2\log n(h) +2$
- Thus the height of an AVL tree is $O(\log n)$

Insertion in an AVL Tree

- Insertion is as in a binary search tree
- Always done by expanding an external node
- Example:

```
44
17
78
32
50
88
48
62
54
```

```
before insertion
```
```
after insertion
```

Insertion

- If an insertion causes $T$ to become unbalanced, we travel up the tree from the newly created node until we find the first node $x$ such that its grandparent $z$ is unbalanced node.
- Since $z$ became unbalanced by an insertion in the subtree rooted at its child $y$,
  $\text{height}(y) = \text{height(left}(y)) + 2$
- Now to rebalance...
**Insertion: rebalancing**

- To rebalance the sub-tree rooted at $z$, we must perform a **restructuring**

**Insertion Example, continued**

unbalanced...

...balanced

**Restructuring (as Single Rotations)**

- Single Rotations:
Restructuring (as Double Rotations)
- double rotations:

![Diagram of double rotations]

Restructure Algorithm
1. we rename x, y, and z to a, b, and c based on the order of the nodes in an in-order traversal.
2. z is replaced by b, whose children are now a and c whose children, in turn, consist of the four other sub-trees formerly children of x, y, and z.

Restructure Algorithm
Algorithm restructure(x, T):
Input: A node x of a binary search tree T that has both a parent y and a grandparent z
Output: Tree T restructured by a rotation (either single or double) involving nodes x, y, and z.
Let (a, b, c) be an in-order listing of the nodes x, y, and z
Let (T0, T1, T2, T3) be an in-order listing of the four sub-trees of x, y, and z
Replace the sub-tree rooted at z with a new sub-tree rooted at b
Make a the left child of b and
T0, T1 be the left and right sub-trees of a.
Make c the right child of b and
T2, T3 be the left and right sub-trees of c.
Restructure Algorithm

1. Let x be the first note such that its grandparent z is unbalanced node. Let y be the parent of x.
2. we rename x, y, and z to a, b, and c based on the order of the nodes in an in-order traversal.
3. z is replaced by b, whose children are now a and c whose children, in turn, consist of the four other sub-trees formerly children of x, y, and z.

```plaintext
c = x
b = y
a = z
```

Double Rotation

```plaintext
a = z
b = x
c = y
```
Restructure Algorithm

1. Let x be the first node such that its grandparent z is unbalanced node. Let y be the parent of x.
2. we rename x, y, and z to a, b, and c based on the order of the nodes in an in-order traversal.
3. z is replaced by b, whose children are now a and c whose children, in turn, consist of the four other sub-trees formerly children of x, y, and z.

Restructure Algorithm (continued)

- Any tree that needs to be balanced can be grouped into 7 parts:
  - x, y, z, and
  - the 4 trees anchored at the children of those nodes (T<sub>0-3</sub>)

Restructure Algorithm (continued)

- Make a new tree
  - which is balanced and
  - 7 parts from the old tree appear in the new tree such that the numbering is still correct when we do an in-order-traversal of the new tree.
- This works regardless of how the tree is originally unbalanced.
Restructure Algorithm (continued)

- Number the 7 parts by doing an in-order traversal. (note that x, y, and z are now renamed based upon their order within the traversal)

```
1 2 (a) 3 4 (b) 5 (c) 6 7
```

Restructure Algorithm (continued)

- Now create an Array of 8 elements. At rank 0 place the parent of z.

```
1 2 3 4 5 6 7
```

- Cut the 4 T trees and place them in their in-order rank in the array.

```
T0 T1 T2 T3
```

Restructure Algorithm (continued)

- Now cut x, y, and z in that order (child, parent, grandparent) and place them in their in-order rank in the array.

```
1 2 3 4 5 6 7
```

- Now we can re-link these sub-trees to the main tree.

```
```

Restructure Algorithm (continued)

- Link in rank 4 (b) where the sub-tree’s root formerly

```
```
Restructure Algorithm (continued)

• Link in ranks 2 (a) and 6 (c) as 4’s children.

• Finally, link in ranks 1, 3, 5, and 7 as the children of 2 and 6.

• Now you have a balanced tree!

Restructure Algorithm (continued)

• NOTE:
  – This algorithm for restructuring has the exact same effect as using the four rotation cases discussed earlier.
  – Advantages: no case analysis, more elegant
Trinode Restructuring

- let \((a, b, c)\) be an inorder listing of \(x, y, z\)
- perform the rotations needed to make \(b\) the topmost node of the three

![Diagram of Trinode Restructuring]

Removal

- We can easily see that performing a `removeAboveExternal(w)` can cause \(T\) to become unbalanced.
- Let \(z\) be the first unbalanced node encountered while traveling up the tree from \(w\). Also, let \(y\) be the child of \(z\) with the larger height, and let \(x\) be the child of \(y\) with the larger height.
- We can perform operation `restructure(x)` to restore balance at the sub-tree rooted at \(z\).

Removal in an AVL Tree

- Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, \(w\), may cause an imbalance.
- Example:
Rebalancing after a Removal

- Let $z$ be the first unbalanced node encountered while travelling up the tree from $w$. Also, let $y$ be the child of $z$ with the larger height, and let $x$ be the child of $y$ with the larger height.
- We perform restructure($x$) to restore balance at $z$.
- As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of $T$ is reached.

Running Times for AVL Trees

- a single restructure is $O(1)$
  - using a linked-structure binary tree
- find is $O(\log n)$
  - height of tree is $O(\log n)$, no restructures needed
- insert is $O(\log n)$
  - initial find is $O(\log n)$
  - Restructuring up the tree, maintaining heights is $O(\log n)$
- remove is $O(\log n)$
  - initial find is $O(\log n)$
  - Restructuring up the tree, maintaining heights is $O(\log n)$

Removal (contd.)

- NOTE: restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of $T$ is reached.
Running Times for AVL Trees

- a single restructure is $O(1)$
  - using a linked-structure binary tree
- find is $O(\log n)$
  - height of tree is $O(\log n)$, no restructures needed
- insert is $O(\log n)$
  - initial find is $O(\log n)$
  - Restructuring up the tree, maintaining heights is $O(\log n)$
  - One restructuring is sufficient to restore the global height balance property
- remove is $O(\log n)$
  - initial find is $O(\log n)$
  - Restructuring up the tree, maintaining heights is $O(\log n)$
  - Single re-structuring is not enough to restore height balance globally. Continue walking up the tree for unbalanced nodes.