COT 6405: Analysis of Algorithms
Giri NARASIMHAN
Definition of big-Oh

- We say that
  - \( F(n) = O(G(n)) \)

If there exists two positive constants, \( c \) and \( n_0 \), such that
- For all \( n \geq n_0 \), we have \( F(n) \leq c \cdot G(n) \)

- Thus, to show that \( F(n) = O(G(n)) \), you need to find two positive constants that satisfy the condition mentioned above.
- Also, to show that \( F(n) \neq O(G(n)) \), you need to show that for any value of \( c \), there does not exist a positive constant \( n_0 \) that satisfies the condition mentioned above.
Algorithm Analysis

- **Worst-case time complexity**
  - Worst possible time of all input instances of length N

- **(Worst-case) space complexity**
  - Worst possible space of all input instances of length N

- **Average-case time complexity**
  - Average time of all input instances of length N
Computation Tree for A on n inputs

- Assume A is a comparison-based sorting alg
- Every node represents a comparison between two items in A
- Branching based on result of comparison
- Leaf corresponds to algorithm halting with output
- Every input follows a path in tree
- Different inputs follow different paths
- Time complexity on input x = depth of leaf where it ends on input x
Upper and Lower Bounds

- Time Complexity of a Problem
  - **Difficulty:** Since there can be many algorithms that solve a problem, what time complexity should we pick?
  - **Solution:** Define upper bounds and lower bounds within which the time complexity lies.

- What is the upper bound on time complexity of sorting?
  - **Answer:** Since SelectionSort runs in worst-case $O(N^2)$ and MergeSort runs in $O(N \log N)$, either one works as an upper bound.
  - **Critical Point:** Among all upper bounds, the best is the lowest possible upper bound, i.e., time complexity of the best algorithm.

- What is the lower bound on time complexity of sorting?
  - **Difficulty:** If we claim that lower bound is $O(f(N))$, then we have to prove that no algorithm that sorts $N$ items can run in worst-case time $o(f(N))$. 
Lower Bounds

- It's possible to prove lower bounds for many comparison-based problems.
- For comparison-based problems, for inputs of length $N$, if there are $P(N)$ possible solutions, then any algorithm needs $\log_2(P(N))$ to solve the problem.
- Binary Search on a list of $N$ items has at least $N + 1$ possible solutions. Hence lower bound is $\log_2(N+1)$.
- Sorting a list of $N$ items has at least $N!$ possible solutions. Hence lower bound is $\log_2(N!) = O(N \log N)$
- Thus, MergeSort is an optimal algorithm. Because its worst-case time complexity equals lower bound!
Beating the Lower Bound

- **Bucket Sort**
  - Runs in time $O(N+K)$ given $N$ integers in range $[a+1, a+K]$
  - If $K = O(N)$, we are able to sort in $O(N)$
  - How is it possible to beat the lower bound?
    - Only because we know more about the data.
    - If nothing is known about the data, the lower bound holds.

- **Radix Sort**
  - Runs in time $O(d(N+K))$ given $N$ items with $d$ digits each in range $[1,K]$

- **Counting Sort**
  - Runs in time $O(N+K)$ given $N$ items in range $[a+1, a+K]$
Stable Sort

- A sort is **stable** if equal elements appear in the same order in both the input and the output.
- Which sorts are stable? Homework!
Order Statistics

- **Maximum, Minimum**
  - **Upper Bound**
    - $O(n)$ because ??
    - We have an algorithm with a single for-loop: $n-1$ comparisons
  - **Lower Bound**
    - $n-1$ comparisons

- **MinMax**
  - **Upper Bound:** $2(n-1)$ comparisons
  - **Lower Bound:** $3n/2$ comparisons

- **Max and 2ndMax**
  - **Upper Bound:** $(n-1) + (n-2)$ comparisons
  - **Lower Bound:** Harder to prove

**Rank**$_{A}(x) = \text{position of } x \text{ in sorted order of } A$
k-Selection; Median

- Select the $k$-th smallest item in list
- Naïve Solution
  - Sort;
  - pick the $k$-th smallest item in sorted list.
    \[ O(n \log n) \] time complexity
- Idea: Modify Partition from QuickSort
  - How?
- Randomized solution: Average case $O(n)$
- Improved Solution: worst case $O(n)$
Using Partition for k-Selection

- Perform Partition from QuickSort (assume all unique items)
- \( \text{Rank}(\text{pivot}) = 1 + \# \text{ of items that are smaller than pivot} \)
- If \( \text{Rank}(\text{pivot}) = k \), we are done
- Else, recursively perform k-Selection in one of the two partitions

```
PARTITION(array A, int p, int r)
1  x ← A[r]  ▶ Choose pivot
2  i ← p − 1
3  for j ← p to r − 1
4     do if (A[j] ≤ x)
5         then i ← i + 1
7  exchange A[i + 1] ← A[r]
8  return i + 1
```
QuickSelect: a variant of QuickSort

```
QUICKSELECT(array A, int k, int p, int r)
    ▷ Select k-th largest in subarray $A[p..r]$
1   if ($p = r$)
2     then return $A[p]$
3   q ← PARTITION($A, p, r$)
4   i ← $q - p + 1$ ▷ Compute rank of pivot
5   if ($i = k$)
6     then return $A[q]$
7   if ($i > k$)
8     then return QUICKSELECT($A, k, p, q$)
9   else return QUICKSELECT($A, k - i, q + 1, r$)
```
k-Selection Time Complexity

- Perform Partition from QuickSort (assume all unique items)
- \( \text{Rank}(\text{pivot}) = 1 + \# \text{ of items that are smaller than pivot} \)
- If \( \text{Rank}(\text{pivot}) = k \), we are done
- Else, recursively perform k-Selection in one of the two partitions

- On the average:
  - \( \text{Rank}(\text{pivot}) = n / 2 \)
- Average-case time
  - \( T(N) = T(N/2) + O(N) \)
  - \( T(N) = O(N) \)
- Worst-case time
  - \( T(N) = T(N-1) + O(N) \)
  - \( T(N) = O(N^2) \)
Randomized Solution for k-Selection

- Uses RandomizedPartition instead of Partition
  - RandomizedPartition picks the pivot uniformly at random from among the elements in the list to be partitioned.
- Randomized k-Selection runs in O(N) time on the average
- Worst-case behavior is very poor O(N^2)
Readings for next class

- Trees,
- Binary Trees,
- Binary Search Trees,
- Balanced Binary Search Trees
Data Structure Evolution

- Standard operations on data structures
  - Search
  - Insert
  - Delete
- Linear Lists
  - Implementation: Arrays (Unsorted and Sorted)
- Dynamic Linear Lists
  - Implementation: Linked Lists
- Dynamic Trees
  - Implementation: Binary Search Trees
BST: Search

\[
\text{TreeSearch}(\text{node } x, \text{key } k)
\]

- Search for key \( k \) in subtree rooted at node \( x \)

1. if \( ((x = \text{NIL}) \text{ or } (k = \text{key}[x])) \) then return \( x \)
2. if \( (k < \text{key}[x]) \) then return \( \text{TreeSearch}(\text{left}[x], k) \)
3. else return \( \text{TreeSearch}(\text{right}[x], k) \)

Time Complexity: \( O(h) \)

\( h = \text{height of binary search tree} \)

Not \( O(\log n) \) — Why?
BST: Insert

TREEINSERT\((tree \ T, node \ z)\)
\>
Insert node \(z\) in tree \(T\)

1. \(y \leftarrow NIL\)
2. \(x \leftarrow root[T]\)
3. while \((x \neq NIL)\)
   - do \(y \leftarrow x\)
   - if \((key[z] < key[x])\)
     - then \(x \leftarrow left[x]\)
     - else \(x \leftarrow right[x]\)
4. \(p[z] \leftarrow y\)
5. if \((y = NIL)\)
   - then \(root[T] \leftarrow z\)
   - else if \((key[z] < key[y])\)
     - then \(left[y] \leftarrow z\)
     - else \(right[y] \leftarrow z\)

Time Complexity: \(O(h)\)
\(h = \) height of binary search tree

Search for \(x\) in \(T\)
Insert \(x\) as leaf in \(T\)
BST: Delete

Time Complexity: $O(h)$

$h = \text{height of binary search tree}$

Set $y$ as the node to be deleted. It has at most one child, and let that child be node $x$.

If $y$ has one child, then $y$ is deleted and the parent pointer of $x$ is fixed.

The child pointers of the parent of $x$ is fixed.

The contents of node $z$ are fixed.
## Common Data Structures

<table>
<thead>
<tr>
<th></th>
<th>Search</th>
<th>Insert</th>
<th>Delete</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Arrays</td>
<td>O(N)</td>
<td>O(1)</td>
<td>O(N)</td>
<td></td>
</tr>
<tr>
<td>Sorted Arrays</td>
<td>O(log N)</td>
<td>O(N)</td>
<td>O(N)</td>
<td></td>
</tr>
<tr>
<td>Unsorted Linked Lists</td>
<td>O(N)</td>
<td>O(1)</td>
<td>O(N)</td>
<td></td>
</tr>
<tr>
<td>Sorted Linked Lists</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(N)</td>
<td></td>
</tr>
<tr>
<td>Binary Search Trees</td>
<td>O(H)</td>
<td>O(H)</td>
<td>O(H)</td>
<td>H = O(N)</td>
</tr>
<tr>
<td>Balanced BSTs</td>
<td>O(log N)</td>
<td>O(log N)</td>
<td>O(log N)</td>
<td>As H = O(log N)</td>
</tr>
</tbody>
</table>
Animations

- https://visualgo.net/
- http://www.cs.jhu.edu/~goodrich/dsa/trees/
- https://www.youtube.com/watch?v=Y-5ZodPvhmM
- http://www.algoanim.ide.sk/
Red-Black (RB) Trees

- Every node in a red-black tree is colored either red or black.
  - The root is always black.
  - Every path on the tree, from the root down to the leaf, has the same number of black nodes.
  - No red node has a red child.
  - Every NIL pointer points to a special node called NIL[T] and is colored black.
- Every RB-Tree with n nodes has black height at most $\log n$.
- Every RB-Tree with n nodes has height at most $2\log n$. 
**Red-Black Tree Insert**

```plaintext
RB-Insert (T,z)  // pg 315
   // Insert node z in tree T
   y = NIL[T]
   x = root[T]
   while (x ≠ NIL[T]) do
      y = x
      if (key[z] < key[x])
         x = left[x]
      else
         x = right[x]
      p[z] = y
      if (y == NIL[T])
         root[T] = z
      else if (key[z] < key[y])
         left[y] = z
      else
         right[y] = z
      // new stuff
      left[z] = NIL[T]
      right[z] = NIL[T]
      color[z] = RED
      RB-Insert-Fixup (T,z)
```

```plaintext
RB-Insert-Fixup (T,z)
   while (color[p[z]] == RED) do
      if (p[z] = left[p[p[z]]]) then
         y = right[p[p[z]]]
         if (color[y] == RED) then         // C-1
            color[p[z]] = BLACK
            color[y] = BLACK
            z = p[p[z]]
            color[z] = RED
         else if (z = right[p[z]]) then // C-2
            z = p[z]
            LeftRotate(T,z)
            color[p[z]] = BLACK         // C-3
            color[p[p[z]]] = RED
            RightRotate(T,p[p[z]])
         else
           // Symmetric code: “right” ↔ “left”
           . . .
      color[root[T]] = BLACK
```
Case 1: Non-elbow; sibling of parent (y) red
Case 2: Elbow case
Case 3: Non-elbow; sibling of parent black
Rotations

```
LeftRotate(T, x)  // pg 278  
    // right child of x becomes x's parent.
    // Subtrees need to be readjusted.
    y = right[x]
    right[x] = left[y]  // y's left subtree becomes x's right
    p[left[y]] = x
    p[y] = p[x]
    if (p[x] == Nil[T]) then  
        root[T] = y  
    else if (x == left[p[x]]) then  
        left[p[x]] = y  
    else right[p[x]] = y  
    left[y] = x  
    p[x] = y
```
Reading for next class

- Red Black Trees
  - Properties
  - Invariants
  - Insert and Delete
- Mathematical Induction