COT 6405: Analysis of Algorithms

Giri NARASIMHAN

Recap of Sorting Algorithms

- SelectionSort
- InsertionSort
- BubbleSort
- QuickSort
- MergeSort
- HeapSort
- Bucket & Radix Sort
- Counting Sort

Worst Case: \(O(N^2)\)

Avg Case: \(O(N \log N)\)

Lower Bound for Comparison-based Sorting

Worst Case: \(O(N)\); Not comparison-based
Tree Sorting

- BST is a search structure that helps efficient search
  - Search can be done in $O(h)$ time, where $h =$ height of BST
  - Also inserts and deletes can be done in $O(h)$ time
  - Unfortunately, Height $h = O(N)$
- Balanced BST improves BST with $h = O(\log N)$
  - Thus search can be done in $O(\log N)$
  - And, inserts and deletes too can be done in $O(\log N)$ time
- We can use BBSTs in the following way:
  - Repeatedly insert $N$ items into a BBST
  - Repeatedly delete the smallest item from the BBST until it is empty
- $N$ inserts and $N$ deletes can be done in $O(N \log N)$ time
**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
  - \( T(N) = T(N-1) + O(N) \)
  - \( T(N) = O(N^2) \)

```
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
```
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 $O(N^2)$
k-Selection & Median: Improved Algorithm

Start with initial array
Use median of medians as pivot

\[ T(n) < O(n) + T(n/5) + T(3n/4) \]
ImprovedSelect

**ImprovedSelect**\((array \ A, \ int \ k, \ int \ p, \ int \ r)\)

\[
\begin{align*}
\text{\> Select } k\text{-th largest in subarray } A[p..r] \\
1 \quad \text{if } (p = r) \\
2 \quad \text{then return } A[p] \\
3 \quad \text{else } N \leftarrow r - p + 1 \\
4 \quad\text{Partition } A[p..r] \text{ into subsets of 5 elements and} \\
\text{collect all medians of subsets in } B[1..\lceil N/5 \rceil]. \\
5 \quad Pivot \leftarrow \text{ImprovedSelect}(B, 1, \lceil N/5 \rceil, \lceil N/10 \rceil) \\
6 \quad q \leftarrow \text{PivotPartition}(A, p, r, Pivot) \\
7 \quad i \leftarrow q - p + 1 \quad \text{\> Compute rank of pivot} \\
8 \quad \text{if } (i = k) \\
9 \quad \text{then return } A[q] \\
10 \quad \text{if } (i > k) \\
11 \quad \text{then return } \text{ImprovedSelect}(A, k, p, q - 1) \\
12 \quad \text{else return } \text{ImprovedSelect}(A, k - i, q + 1, r)
\end{align*}
\]
PivotPartition

\[\text{PivotPartition}(\text{array } A, \text{int } p, \text{int } r, \text{item } Pivot)\]

\[\triangleright \text{ Partition using provided } \text{Pivot}\]

1. \(i \leftarrow p - 1\)
2. \(\text{for } j \leftarrow p \text{ to } r\)
3. \(\text{do if } (A[j] \leq Pivot)\)
   
4. \(\text{then } i \leftarrow i + 1\)
5. \(\text{exchange } A[i] \leftrightarrow A[j]\)
6. \(\text{return } i + 1\)
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

\[ \text{Time Complexity: } O(h) \]
\[ h = \text{height of binary search tree} \]

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

\begin{algorithm}
\textbf{TREE INSERT}(tree } T, \text{ node } z) \\
\text{▷ Insert node } z \text{ in tree } T \\
1. \quad y \leftarrow NIL \\
2. \quad x \leftarrow root[T] \\
3. \quad \text{while } (x \neq NIL) \\
4. \quad \quad \text{do } y \leftarrow x \\
5. \quad \quad \quad \text{if } (key[z] < key[x]) \\
6. \quad \quad \quad \quad \text{then } x \leftarrow left[x] \\
7. \quad \quad \quad \quad \text{else } x \leftarrow right[x] \\
8. \quad \quad p[z] \leftarrow y \\
9. \quad \text{if } (y = NIL) \\
10. \quad \quad \text{then } root[T] \leftarrow z \\
11. \quad \quad \text{else if } (key[z] < key[y]) \\
12. \quad \quad \quad \text{then } left[y] \leftarrow z \\
13. \quad \quad \quad \text{else } right[y] \leftarrow z
\end{algorithm}

Time Complexity: $O(h)$

$h = \text{height of binary search tree}$
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>Data Structure</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

RB-Insert \((T, z)\)  
// pg 315

// Insert node \(z\) in tree \(T\)
\(y = \text{NIL}[T]\)
\(x = \text{root}[T]\)

while \((x \neq \text{NIL}[T])\) do
    \(y = x\)
    if \((\text{key}[z] < \text{key}[x])\)
        \(x = \text{left}[x]\)
        \(x = \text{right}[x]\)
    \(p[z] = y\)
    if \((y = \text{NIL}[T])\)
        \(\text{root}[T] = z\)
    else if \((\text{key}[z] < \text{key}[y])\)
        \(\text{left}[y] = z\)
    else \(\text{right}[y] = z\)
    // new stuff
\(\text{left}[z] = \text{NIL}[T]\)
\(\text{right}[z] = \text{NIL}[T]\)
\(\text{color}[z] = \text{RED}\)
RB-Insert-Fixup \((T, z)\)

RB-Insert-Fixup \((T, z)\)

while \((\text{color}[p[z]] = \text{RED})\) do
    if \((p[z] = \text{left}[p[p[z]]])\) then
        \(y = \text{right}[p[p[z]]]\)
    if \((\text{color}[y] = \text{RED})\) then  // C-1
        \(\text{color}[p[z]] = \text{BLACK}\)
        \(\text{color}[y] = \text{BLACK}\)
        \(z = p[p[z]]\)
        \(\text{color}[z] = \text{RED}\)
    else  // \((z = \text{right}[p[p[z]]])\) then  // C-2
        \(z = p[z]\)
        \(\text{LeftRotate}(T, z)\)
        \(\text{color}[p[z]] = \text{BLACK}\)  // C-3
        \(\text{color}[p[p[z]]] = \text{RED}\)
        \(\text{RightRotate}(T, p[p[z]])\)
    else
        // Symmetric code: “right” ↔ “left”
        .
        .
        .
        \(\text{color}[\text{root}[T]] = \text{BLACK}\)
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] == NULL[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
## More Dynamic Operations

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

<table>
<thead>
<tr>
<th></th>
<th>Se/In/De</th>
<th>Rank</th>
<th>Select</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Balanced BSTs</strong></td>
<td>O(log N)</td>
<td>O(N)</td>
<td>O(N)</td>
<td></td>
</tr>
<tr>
<td><strong>Augmented BBSTs</strong></td>
<td>O(log N)</td>
<td>O(log N)</td>
<td>O(log N)</td>
<td></td>
</tr>
</tbody>
</table>
Operations on Dynamic RB Trees

- **K-Selection**
  - Select an item with a specified rank
  - "Efficient" solution not possible without preprocessing
- Preprocessing - store additional information at nodes
- Inverse of K-Selection
  - Find rank of an item in the tree
- What information should be stored?
  - Rank
  - ??
OS-Rank

OS-RANK(x, y)

// Different from text (recursive version)
// Find the rank of x in the subtree rooted at y

1. r = size[left[y]] + 1
2. if x = y then return r
3. else if (key[x] < key[y]) then
4. return OS-RANK(x, left[y])
5. else return r + OS-RANK(x, right[y])

Time Complexity $O(\log n)$
OS-Select

OS-SELECT(x,i) //page 304
// Select the node with rank i
// in the subtree rooted at x
1. \( r = \text{size}[\text{left}[x]]+1 \)
2. if \( i = r \) then
3. \hspace{1em} \text{return x} \\
4. elseif \( i < r \) then
5. \hspace{1em} \text{return OS-SELECT (left[x], i)} \\
6. else \hspace{1em} \text{return OS-SELECT (right[x], i-r)} \\

Time Complexity O(log n)
RB-Tree Augmentation

Augment x with $\text{Size}(x)$, where

- $\text{Size}(x)$ = size of subtree rooted at x
- $\text{Size}(\text{NIL}) = 0$
Augmented Data Structures

- Why is it needed?
  - Because basic data structures not enough for all operations
  - storing extra information helps execute special operations more efficiently.

- Can any data structure be augmented?
  - Yes. Any data structure can be augmented.

- Can a data structure be augmented with any additional information?
  - Theoretically, yes.

- How to choose which additional information to store.
  - Only if we can maintain the additional information efficiently under all operations. That means, with additional information, we need to perform old and new operations efficiently maintain the additional information efficiently.
How to augment data structures

1. choose an underlying data structure
2. determine additional information to be maintained in the underlying data structure,
3. develop new operations,
4. verify that the additional information can be maintained for the modifying operations on the underlying data structure.
Augmenting RB-Trees

Theorem 14.1, page 309

Let $f$ be a field that augments a red-black tree $T$ with $n$ nodes, and $f(x)$ can be computed using only the information in nodes $x$, $\text{left}[x]$, and $\text{right}[x]$, including $f[\text{left}[x]]$ and $f[\text{right}[x]]$. Then, we can maintain $f(x)$ during insertion and deletion without asymptotically affecting the $O(\log n)$ performance of these operations.

For example,

\[
\text{size}[x] = \text{size}[\text{left}[x]] + \text{size}[\text{right}[x]] + 1
\]

\[
\text{rank}[x] = ?
\]
Augmenting information for RB-Trees

- Parent
- Height
- Any associative function on all previous values or all succeeding values.
- Next
- Previous
Reading for next class

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