CMPS 101 Homework 2

Dominik Schmidt

November 9, 2018

Question 1

Heaps naturally lead to a sorting algorithm, Heapsort. Starting from array A, build it into a min-heap. Repeatly called Extract-Min to get the elements of A in sorted order. Give pseudocode for Heapsort and show that it is not stable.

Pseudo Code

The Pseudo Code below makes use of an **extractMin**(*Array*) method that
1) Returns and swaps the root element with the last element.
2) Decrements the size of the Heap.
3) Restores the Minimum Heap property.

Algorithm 1 Pseudo Code for Heap Sort

```
buildMinHeap(Array)
for all i in Array do
    Array[i] ← extractMin(Array)
end for
return Array
```

Heap Sort stability

Heap Sort is not a stable sorting algorithm, although it can be implemented to be stable. For the purpose of the question we will show that the Heap Sort Implementation in *Introduction to Algorithms*, *3rd Edition* which uses a Max Heap and builds the array from the largest index to the smallest by inserting the largest element in the heap in the largest unsorted index in the array is not stable. With the following example we can show that Heap Sort is not stable: Consider the Array $\{56, 33, 31, 29, 13_1, 13_2, 7, 2, 0\}$. Note that 13_1 and 13_2 both have value 13, the subscript merely keeps track of their order. Heap Sort will insert the largest value in the Heap in the largest unsorted the index so our sorted Array will look like this: $\{0, 2, 7, 13_2, 13_1, 29, 31, 33, 56\}$. We can clearly see that 13_1 and 13_2 switched order, therefore Heap Sort is not stable.

Question 2

Given an input array A of length n and a positive integer k > 0, design an algorithm that outputs the largest k elements in sorted order. Provide pseudocode and give a time-complexity analysis. You will get full credit if the time-complexity is O(n+k log k).

Pseudo Code

Heap Sort k times

Algorithm 2 Pseudo Code to find the k largest elements in sorted order

kLargest($Array, k$)
$SortedK \leftarrow new\ Array[k]$
buildMaxHeap($Array$**)**
for $i \leftarrow k - 1$, $i >= 0$, i - - **do**
  $SortedK[i] \leftarrow$ **extractMax(**$Array$**)**
end for
return $SortedK$

Time Complexity Analysis

The time complexity of building a Max Heap out of an array with n elements is $O(n)$. The time complexity of extractMax which also restores the heap property is $O(\log n)$ and is executed k times so overall Time Copmplexity is $O(n+k\log n)$.

Question 3

Prove that the number of keys stored in a 2-3 tree of height h is $\Omega(2^h)$ and $O(3^h)$.

Proof that the number of keys is $\Omega(2^h)$

Proof. To prove this statement we will use Induction.

> **Basis Step:** Let the height of the 2-3 tree $h = 0$, then number of keys stored $k = 2^0 = 1$. $1 \geq 1$ so $1 = \Omega(2^0)$ meaning that our Base Case holds.
>
> **Inductive Hypothesis:** $k = \Omega(2^h)$, so our number of keys $k \geq 2^h$.
>
> **Inductive Step:** Assume that our Hypothesis holds up to h. Prove for $h + 1$. Let l be the number of keys in a 2-3 tree oh height h

+ 1. A 2-3 tree of height h + 1 is a 2-3 tree of height h where at least one of the leaf nodes has an additional subtree of height 1 as a child which by definition of a 2-3 tree stores at least 2 keys. By the Inductive Hypothesis $l \geq 2 \cdot k \geq 2 \cdot 2^h \geq 2^{h+1}$.

□

Proof that the number of keys is $O(3^h)$

Proof. To prove this statement we will use Induction.

Basis Step: Let the height of the 2-3 tree $h = 0$, then number of keys stored $k = 3^0 = 1$. $1 \leq 1$ so $1 = O(3^0)$ meaning that our Base Case holds.

Inductive Hypothesis: $k = O(3^h)$, so our number of keys $k \leq 3^h$.

Inductive Step: Assume that our Hypothesis holds up to h. Prove for h + 1. Let l be the number of keys in a 2-3 tree oh height h + 1. A 2-3 tree of height h + 1 is a 2-3 tree of height h where at most all of the leaf nodes have an additional subtree of height 1 as a child which by definition of a 2-3 tree stores at most 3 keys. By the Inductive Hypothesis $l \leq 3 \cdot k \leq 3 \cdot 3^h \leq 3^{h+1}$.

□

Question 4

Suppose you are given two 2-3 trees T1, T2 and a value x such that all keys in T1 are less than x, and all keys in T2 are greater than x. Give an algorithm that constructs a single new T 0 that has the union of keys in T1, T2, and x. Give a running time analysis.

Pseudo Code

Algorithm 3 Pseudo Code for merging 2-3 Trees

$T0 \leftarrow$ *new* 2-3 Tree
if $T1.height == T2.height$ **then**
  $T0.root \leftarrow x$
  $T0.left \leftarrow T1$
  $T0.right \leftarrow T2$
else if $T1.height > T2.height$ **then**
  $T2$.**insert**(x)
  $currentNode \leftarrow T1.root$
  for $i \leftarrow 0$, $i < T1.height$ - $T2.height$, $i{+}{+}$ **do**
    $currentNode \leftarrow currentNode.right$
  end for
  $currentNode.right \leftarrow T2$
  if $currenNode.right.numOfKeys > 3$ **then**
    fixOverall($currentNode$)
  end if
  $T0.root \leftarrow T1.root$
else if $T1.height < T2.height$ **then**
  $T1$.**insert**(x)
  $currentNode \leftarrow T2.root$
  for $i \leftarrow 0$, $i < T2.height$ - $T2.height$, $i{+}{+}$ **do**
    $currentNode \leftarrow currentNode.right$
  end for
  $currentNode.right \leftarrow T1$
  if $currenNode.right.numOfKeys > 3$ **then**
    fixOverall($currentNode$)
  end if
  $T0.root \leftarrow T2.root$
end if

Runtime Analysis

This alorithm constructs a single new 2-3 Tree $T0$ that has the union of keys in $T1$, $T2$, and x in time $O(\log n)$. The algorithm needs to consider three cases: First, $T1$ and $T2$ are of equal height. Here, the algorithm will construct $T0$ in constant time $O(1)$ by making x the root Node and storing the subtree $T1$ which contains all keys smaller than x as the left child of the root, and storing the subtree $T2$ which contains all keys greaterthan x as the right child of the root. The next 2 cases are that either $T1$ or $T2$ are bigger. If this is the case, the algorithm will find the $h1 - h2$th largest node and insert $T2$ as a subtree of $T1$ if the height of $T1$ is bigger or find the $h2 - h1$th smallest node and insert $T1$ as a subtree of $T2$. If necessary, the algorithm will then restore the 2-3 Tree property, Giving us a total runtime of $O(\log n)$.

Question 5

Consider a binary search tree where keys are positive integers. Augment the tree to answer Range queries of the form: "how many elements have key in the range [a, b]"? Thus, such a query is called by the function Range(a, b). Provide pseudocode for Insert, Delete, and Range queries. Provide a running time analysis for all these queries, in terms on n (the number of nodes in the tree) and D (the maximum depth). (Hint: you might want to maintain subtree sizes at the nodes.)

BST Insertion

Algorithm 4 Pseudo Code for inserting a key k in a BST *node*

```
if node == null then
    node.data ← k
    node.left ← null
    node.right ← null
    node.treeSize ← 0
    return
else if node != null then
    if k < node.data then
        node.treeSize++
        node.left ← insert(node.left, k)
    else if k > node.data then
        node.treeSize++
        node.right ← insert(node.right, k)
    end if
end if
return
```

Time Complexity Analysis:

The insert algorthim runs in worst case $O(n)$ and $O(D)$. If the tree becomes very unbalanced it effectively functions as a linked list and the time compelxity of inserting at the last node in a Linked List is $O(n)$.

BST Deletion

Algorithm 5 Pseudo Code for deleting a node with key k in a BST $node$

```
if node != null then
  if k < node.data then
    node.treeSize − −
    node.left ← delete(node.left, k)
  else if k > node.data then
    node.treeSize − −
    node.right ← delete(node.right, k)
  else
    if node.right == null and node.left == null then
      node ← null
    else if node.right == null and node.left != null then
      node ← node.left
    else if node.right != null and node.left == null then
      node ← node.right
    else
      smallestRightChild ← node.right.data
      tempNode ← node.right
      while tempNode.left != null do
        smallestRightChild ← tempNode.left.data
        tempNode ← tempNode.left
      end while
      node.data ← smallestRightChild
      node.right ← delete(node.right, node.data)
    end if
  end if
end if
```

Time Analysis:
The delete algorthim runs in worst case $O(n)$ and $O(D)$. If the tree becomes very unbalanced it effectively functions as a linked list and the time compelxity of deleting the last node in a Linked List is $O(n)$.

BST Range Query

Time Complexity Analysis:
The Range Query Algorithm runs in $O(n)$ and $O(D)$. Such a runtime occurs in the case that the tree is a linked list where the key at the root is equal to a and the key at the rightmost child of the tree is equal to b.

Algorithm 6 Pseudo Code for num of keys in Range [a,b] of a BST *node*

```
if node == null then
  return 0
else if node.data < a then
  Range(node.right, a, b)
else if node.data > b then
  Range(node.left, a, b)
else
  leftNode ← node.left
  leftCount ← 0
  while leftNode.left ! = null and leftNode.data > a do
    if leftNode.right ! = null then
      leftCount += leftNode.right.treeSize
    end if
    leftCount + +
    leftNode ← leftNode.left
  end while
  if leftNode.left ! = null and leftNode.left.right.data ! = null and
  leftNode.left.right.data > a then
    leftCount += Range(leftNode.left.right, a, b)
  end if
  rightNode ← node.right
  rightCount ← 0
  while rightNode.right. ! = null and rightNode.data < b do
    if rightNode.left ! = null then
      rightCount += rightNode.left.treeSize
    end if
    rightCount + +
    rightNode ← rightNode.right
  end while
  if rightNode.right ! = null and rightNode.righ.left.data ! = null and
  righttNode.right.left.data < b then
    rightCount += Range(rightNode.right.left, a, b)
  end if
  return 1 + rightCount + leftCount
end if
```