311 Home Work 1

Ben purdy

March 3, 2020

1

An alternative method of performing an tree walk of an inorder n-node binary search tree finds the minimum element in the tree by calling Minimum and then making $n-1$ calls to Successor. Analyze this algorithm, and show that it runs in$\Theta(n)$ time.
tree walk is O(n)

Proof. By calling successor n-1 time after calling minimum once running in O(h). This will start at the leftmost node of the tree propagate up to its parent by the first successor call. At the parent, this will go right and then down the leftmost subtree. repeating this action will visit all nodes in the tree. At worst all nodes will be visited on the way down and on the way up. Since all nodes will be visited and at most twice. This means that the running time is twice the number of edges. $2(nodes-1) \Rightarrow O(n^2)$, and $\Omega(n)$ because each node is visited once.

hence $\Theta(n)$. □

2

We can sort a given set of n numbers by first building a binary search tree containing these numbers (using Insert repeatedly to insert the numbers one by one) and then printing the numbers by an inorder tree walk. (a) What are the worst-case and best-case running times for this sorting algorithm if we use an ordinary binary search tree? Justify your answer. (b) Suppose that, instead of an ordinary binary search tree, we use a red-back tree. What are the worst-case and best-case running times for this sorting algorithm? Justify your answer.

Trivially inorder tree traversal is O(n).

Proof. - Justification

(a)

The worst-case for insertion in an ordinary binary tree in (ascending/descending) order creating a list extending to the right/left of the head node. When building the binary tree, since it is a list each iteration takes the hight time. So with n nodes ($\sum_{k=0}^{n} = \frac{n(n+1}{2}$). Thus the run time is $O(n^2)$

The best case. If the binary tree is balanced then insertion takes O(h). For any size balanced binary tree insertion still takes O(h). The hight of the tree cant exceed log(n) depth, thus for adding n nodes insertion takes O(nlog(n))

(b).

Worst case. red-black trees are balanced binary search trees. This means that insertion takes O(h) = O(log(n)). Since a red-black tree can only be balanced the worst case for insertion is $O(nlog(n))$. "also see RBinsert run time proof from RB Trees-1.pdf"

Best case. Insertion takes O(logn) because its balanced. to insert n nodes into a full binary tree takes O(nlogn). see reasoning for above in (a) bestcase and (b) worst case

red-black trees are balanced binary search trees that guarantee tha tall query (Search,Maximum, etc.) and update (Insert and Delete)operations takeO(logn)time in the worst case.

□

3

Suppose that instead of each node x keeping the attribute x.parent, pointing to x's parent, it keeps x.successor, pointing to x's successor. Give pseudocode for Search, Insert, and Delete on a binary search tree T using this representation. These procedures should operate in time O(h), where h is the height of the tree T, and should update successor fields as necessary. Consider only ordinary binary search trees, not red-black trees. (Hint: You may wish to implement a subroutine that returns the parent of a node.)

assume for the deletion procedure that all the keys are distinct, as that has been a frequent assumption throughout this chapter. This will however depend on it. Our deletion procedure first calls search until we are one step away from the node we are looking for, that is, it calls Predecessor(Root T,key)

```
Successor(tree T,node x)
if right[x] != null then
    return minimum(right[x])
else
    return TreeSuccessor(root[T], x, NIL)
TreeSuccessor(y, x, c)
if y = x then
    return c
if key[x] < key[y] then
    return TreeSuccessor(left[y], x, y)
else
    return TreeSuccessor(right[y], x, c)
```

```
INSERT(T, z)
y = NIL
x = T.root
pred = NIL
while  $x$ != $null$ do
    y = x
if $z.key < x.key$ then
    x = x.left
else
    pred = x
    x = x.right
if $y$ == $null$ then
    T.root = z
    z.succ = NIL
else if  $z.key < y.key$ then
    y.left = z
    z.succ = y
    if $pred$ != $null$ then
        pred.succ = z
else
    y.right = z
    z.succ = y.succ
    y.succ =
```

```
comment == how is this wrong. I got scared and left it.
 Insert(treeRoot T,node x) if  $T$ == $null$ then
    T = x
    return
if $x < T$ then
    return Insert( T.left,x);
else if $x > T$ then
    return Insert( T.right,x);
return T;
```

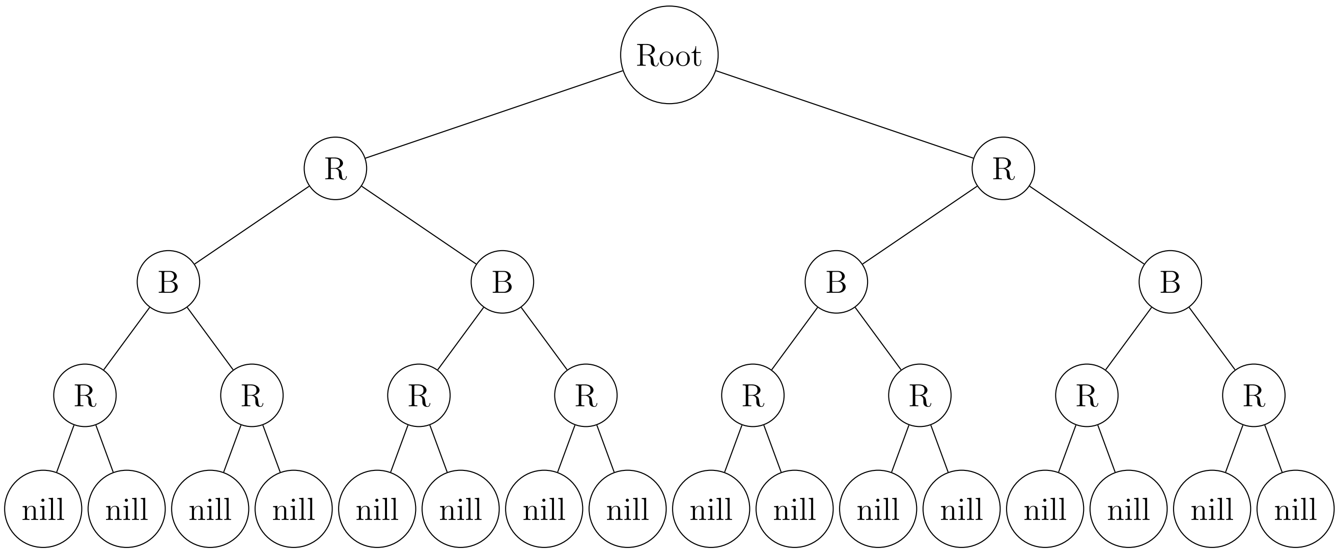
```
Predecessor(value x, key k) if  $k < x.key$ then
    y = x.left
else
    y = x.right
if $y == NIL$ then
    throw error
else if  $y.key = k$ then
    return x
else

return Predecessor(y,k)
```

4

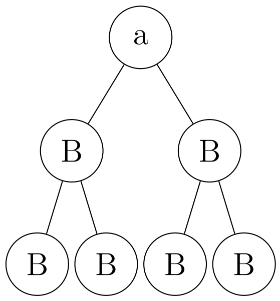
What is the largest possible number of internal nodes in a red-black tree with black-height k? What is the smallest possible number? Express your answer as a function of k.

black hight for 2 example



$$a_1 = \sum_{n=0}^{2k-1} 2^n = 4^k - 1 = 2^{2k} - 1 \tag{1}$$

the root is counted as a internal node and the largest number of internal nodes behaves on the summation a_1 becuase by starting with red nodes after the root you have to double the hight adding k more rows. The by adding k more rows and removing the external nill nodes you get 2k-1 for your internal hight.



$$a_2 = \sum_{n=0}^{k-1} 2^n = 2^k - 1 \tag{2}$$

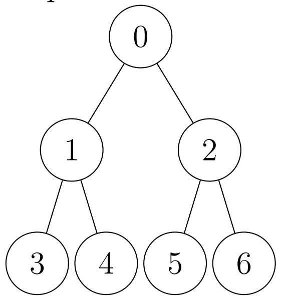
the smallest case is a_2 becuase if the tree is compitly black the number of nodes is a_2. After you remove the nill nodes you get k-1 hight of external nodes

5

A d-ary heap is like a binary heap, but (with one possible exception) non-leaf nodes have d children instead of 2 children.
(a) How would you represent a d-ary heap in an array?

A d-array heap will behave the same way as a normal heap but you add d children behind the parent node.

parent(index i) = (i-1)/d – this returns the parent position
child(index i , j jth child) = j+di+1 – this returns any child position for any nodes parent.



(b) What is the height of a d-ary heap of n elements in terms of n and d? Justify your answer. A-ary heap would have a height of S $\Theta(log_d n)$ We know that

$$n = \sum_{i=0}^{h} d^i = \frac{d^{h+1} - 1}{d - 1} \tag{3}$$

$$\begin{aligned} n &= \frac{d^{h+1} - 1}{d - 1} \\ n(d-1) &= d^{h+1} - 1 \\ n(d-1) + 1 &= d^{h+1} \\ log_d(n(d-1)+1) &= h + 1 \end{aligned}$$

$$n = \sum_{i=0}^{h-1} d^i = \frac{d^h - 1}{d - 1} \tag{4}$$

$$\begin{aligned} n &= \frac{d^h - 1}{d - 1} \\ n(d-1) &= d^h - 1 \\ n(d-1) + 1 &= d^h \\ log_d(n(d-1)+1) &= h \end{aligned}$$

hence

$$\begin{aligned} \sum_{i=0}^{h-1} d^i &< n \leq \sum_{i=0}^{h} d^i \\ \frac{d^h - 1}{d - 1} &< n \leq \frac{d^{h+1} - 1}{d - 1} \\ h < log_d(n(d{-}1)+1) &\leq h + 1 \\ log_d(n(d{-}1)+1) - 1 &= h \end{aligned}$$

becuase the $\sum_{i=0}^{h-1} d^i$ is a strict lower bound for h and that $\sum_{i=0}^{h} d^i$ is the max upper bound for h this shows that hight h is $\Theta(log_d n)$.

(c) Give an efficient implementation of ExtractMax in a d-ary max-heap. Analyze its running time in terms of d and n.

```
Heapify(Array A, parent)
Array children = parent.Getchildren;
largest = children[0]
for k= 1 ⇒ d children do
    if largest ≤ children[k] then
        largest = children[k];
if parent ≤ largest then
    swap largest and child
    Heapify(A, child)
```

```
ExtractMax(Heap A)
if $A.size < 1$ then
    "issue"
max = A[0]
A[0] = A[A.heap-size-1]
A.size--
Heapify(A, 0)
return max
```

The running time of Heaify is O($dlog_dn$) because at each ideration it touches d nodes and can extend O(h) hight. Since we have d children the hight of the tree is log_dn thus d opperatins log_dn times $\Rightarrow dlog_dn$.

the runnging time for ExtractMax is constant work plus the time of Heaify. Thus O($dlog_dn$) + c $\Rightarrow dlog_dn$.

(d) Give an efficient implementation of Insert in a d-ary max-heap. Analyze its running time in terms of d and n.

```
Insert(Array A, n, size)
size + 1
A[size-1] = n
heapify(A, A[n-1])
```

the runnging time for Insert is constant work plus the time of Heaify. Thus O($dlog_dn$) + c $\Rightarrow dlog_dn$.

(e) Give an efficient implementation of IncreaseKey(A, i, k), which flags an error if k < A[i], but otherwise sets A[i] = k and then updates the d-ary max-heap structure appropriately. Analyze its running time in terms of d and n

```
IncreaseKey(A, i, k)
if $k < A[i]$ then
    error
A[i] = k
while $i > 1$ || A[i] ¿ A[parent of i] do
    swap A[i] and A[parent of i]
    i = parent.index
```

as the node properdates up it updates the key of max h nodes on the d-heap. This insures that the running time is O($log_d n$)

6

(For practice only — will not be graded) Show the red-black trees that result after successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty red-black tree.