WAVL TREE project:

Noam wolf 318556206 noamwolf.

Ohad gazit 308274570 ohadgazit.

Wavlnode tree documentation:

wavltree class:

fields:

root = the tree root

static extleaf = 1 exernal leaf for a tree

static in_order_ind = 0 (for the keystoarray recursion method)

methods:

constructor:

creating a new external leaf

complexity O(1)

public empty():

return true if the root is null, if not, return false

complexity O(1)

private node_search(int key):

standart search in a binary search tree, going to the right son if k>current node key

and left otherwise. if not found return null

if found return the wanted node.

complexity O(logn)

public search (int key):

calling the node_search method and return the value of the wanted node.

complexity O(logn)

public insert (int key, string value):

if the root is empty creating new root with 2 external leaf childrens.

otherwise finding place to insert (similiar to searching) and adding the new leaf as a

right/left children by calling addleaf function (more info in the addleaf method documentation)

after inserting calling the insertRebalance method.

return the rebalance operations counted by inserRebalance

complexity O(logn)

private insertRebalance (wavlnode current):

climbing the tree from the parent of the added node up to the root.

updating the subtreesize of all the parents in the route to the root (not part of the rebalancing operations counting).

while the rebalance problem didnt solve ,in every level the method checks the rank differences of the node and the rank differences of one of his childrens with the rankDifference() method and calling the rotate() or doublerotate() methods if needed ,if not - update the rank . after rotating keep climbing to the root for subtreesize updates.

return the number of rebalance operation made.

complexity O(logn).

private reg_delete_switched (wavlnode to_del, boolean isfollow):

called after the to_delete item and his succesor got switched.

getting a node to delete and a boolean variable that have true value

if the to_delete node and his succesor were parent-children.

deleting the to_del node and calling the delReb() method.

complexity O(1) + delReb complexity O(logn)

private change (wavlnode to_del , wavlNode replacing):

getting 2 nodes and change between them, update their subtreesize and rank.

return true if they are children-parent

complexity O(1).

private delReb (wavlnode current, string status):

climbing the tree from the deleted item to the root.

update the subtreesize of the nodes (not part of the rebalance operations counting) , update ranks if needed .

calling the casefirstfind() method to check which rebalance step is needed

and calling the rotate()/doublerotate() methods if needed.

return the number of rebalance operations.

complexity O(logn).

public delete(int key):

calling node_search method to find the node to delete.

checking if to_del node is a leaf or unary node and call

delreb() method.

complexity O(1).

if its a binary node, finding its succesor with successor() method

change between them, and calling reg_delete_switched() method.

compexity O(1)+ delReb comlexity O(logn).

public min():

return the value leftmost node in the tree

complexity O(logn).

public max():

return the value of the rightmose node in the tree

complexity O(logn).

public keystoarray():

creating a int[] array and calling the toArrayHelper() method which update the array with the keys of the tree nodes in in-oreder traversal (see mor information in toarrayhelper documentation)

returns the int[] array.

complexity O(n).

public infotoarray():

creating string[] array and calling the toarrayhelper() method which update the array with the values of the tree nodes in in-order traversal.

creating a

returns the string[] array.

complexity O(n)

public size():

return subtreesize of the root

complexity O(1).

public getroot():

return the root of the tree.

complexity O(1).

public select(int i):

on every level checking the size of the node (start from the tree-root) and deciding if going to the left children, going to the right children (and update i) or return the value of the node itself.

return the value of the node with the i-smallest key in the tree.

complexity O(logn).

wavlnode class:

fields :

value

key

right = rightchlidren

left = leftchildren

parent = parent of node

subtreesize

rank

constructors:

Wavlnode(int rank):

creating new wavlnode with the specific rank (used to create extleaf)

complexity O(1).

wavlnode(int k, string value , int rank, int subtreesize):

creaing new wavlnode with the specifics values.

complexity O(1).

methods:

private addlead (int k , string value , string side):

creating a new wavlnode leaf as a right/left children of the node by calling a constructor method.

complexity O(1).

private rankdifference():

return the rank differences between the node and his childrens , ((current - left) - (current - right)).

complexity O(1).

private rotate (string side):

getting a side argument that says which side to rotate calling the method rebalancesizeupdate() to update the sizes as they will be after the rotating.

return the new parent of the subtree.

complexity O(1).

private doublerotate(string side):

getting a side argument that says which side to doublerotate and calls the rotate method twice.

return the new parent of the subtree.

complexity O(1).

private rebalancesizeupdate (string side):

update the subtreesizes of the nodes as seen in the class.

complexity O(1).

private toarrayhelper(int [] int_arr, string[] str_arr, bollean is_int):

passing through the tree in in-order traversal, adding the key or value (depand in the is_int value) to the array (ints array or strings array) in the in_order_ind index and adding 1 to it.

complexity O(n).

private succesor():

finding the leftmost chlidren of the node right children (if have right chlidren). if the node do not have right children and he is a left children of his parent- return his parent

if he is right parent of his children and do not have right son, going up-left until turning up-right and reutrn that node.

complexity O(logn)

private casefirstfind (string status):

find the rebalance after deleation case by checking (exactly as seen in class) the ranks and return how to solve the problem. if no problem has been created return "finished", if demote is needed return "demote"

and if its other case it calls the casefind() method to check how to solve it and return it.

complexity O(1).

private casefind()

find the rebalance after deleation case by checking the ranks (exatcly as seen in class) and return how to solve the problem ("rotate+side", "doublerotate+side").

complexity O(1).

public getkey()

return the key of the node.

complexity O(1).

public getvalue ():

return the value of the node

complexity O(1).

public getleft():

return the left children of the node.

complexity O(1).

public getright():

return the right children of the node.

complexity O(1).

public isinnernode():

return true if inner node by checking the rank.

complexity O(1).

public getsubtreesize():

return the subtreesize of the node.

מדידות

צפי:

נצפה עקב חסם ה-$amortized\ O(1)$ למספר פעולות האיזון בסדרת פעולות הכנסה ומחיקה בעץ שמספר פעולות האיזון הממוצע בפעולת $insert$ ומספר פעולות האיזון הממוצע בפעולת $delete$ יהיה קבוע (פרט לשינויים זניחים, זאת משום שמספר הפעולות שמבוצעות על העץ גדול ומכאן נגיע לקבוע בחסם ה-$amortized\ O(1)$. הקבועים בפעולות המחיקה אינם בכרח יהיו זהים).

נצפה שהמספר המקסימלי של פעולות האיזון בהכנסה ופעולות איזון במחיקה יהיה קרוב למספר במקרה הגרוע כלומר יהיה $\Theta(\log n)$.

תוצאות:

מספר סידורי	מספר פעולות	מספר פעולות האיזון הממוצע לפעולת insert	מספר פעולות האיזון הממוצע לפעולת delete	מספר פעולות האיזון המקסימלי לפעולת insert	מספר פעולות האיזון המקסימלי לפעולת delete
1	10,000	3.4047	2.5147	*15*	9
2	20,000	3.4184	2.5249	16	10
3	30,000	3.4267666666666665	2.538	17	10
4	40,000	3.401575	2.53495	17	11
5	50,000	3.40306	2.52548	18	11
6	60,000	3.4272	2.5290333333333335	18	12
7	70,000	3.4127	2.5268571428571427	19	11
8	80,000	3.4148625	2.5292375	19	11
9	90,000	3.409777777777778	2.5313666666666665	19	12
10	10,0000	3.3973	2.53178	19	12

מספר פעולות האיזון הממוצע לפעולת $insert$ ומספר פעולות האיזון הממוצע לפעולת $delete$ עולים בקנה אחד עם הצפי התאוריתי, אכן הם קבועים עד כדי הפרש זניח (ההפרש המקסימלי בין שני עצים במקרה של $insert$ הוא 0.0299 ובמקרה של $delete$ הוא 0.0233).

מכאן התוצאות אכן מעידות על חסם ה-$amortized\ O(1)$ למספר פעולות האיזון בסדרת פעולות הכנסה ומחיקה.

על מנת להתייחס לתוצאות עבור מספר פעולות המקסימלי בהכנסה\מחיקה נסתכל על הטבלה הבאה:

מספר פעולות	מספר פעולות האיזון המקסימלי לפעולת insert	מספר פעולות האיזון המקסימלי לפעולת delete	$\log$(גודל העץ)
10,000	*15*	9	13.287712379549449
20,000	16	10	14.287712379549449
30,000	17	10	14.872674880270605
40,000	17	11	15.287712379549449
50,000	18	11	15.609640474436812
60,000	18	12	15.872674880270605
70,000	19	11	16.095067301607052
80,000	19	11	16.28771237954945
90,000	19	12	16.457637380991763
10,0000	19	12	16.609640474436812

מטבלה זו ניתן לראות מספר פעולות האיזון המקסימלי לפעולת $insert$ ומספר פעולות האיזון המקסימלי לפעולת $delete$ שתיהן מסדר גודל של לוגריתם של גודל על עץ וקצב גדילתן גם הוא דומה לזה של לוגריתם (זאת היה גם ניתן לראות לפני ההשוואה ללוגריתם משום שהכפלת גודל העץ פי 2 גורת לעלייה בקבוע של המספר המקסימלי של פעולות איזון בהכנסה\מחיקה) ומכאן מספר פעולות האיזון המקסימלי אכן משקף את המקרה הגרוע.

תוצאות אלו אכן מראות שבסרה של הכנסות ומחיקות עדיין יכול להתקבל המקרה הגרוע למרות חסם ה-$amortized\ O(1)$ לפעולות האיזון (כלומר שלעיתים מתקבל המקרה הגרוע אך בממוצע על הסדרה כולה יש מספר פעולות איזון קבוע) *ושמקרה הגרוע אכן*.$\Theta(\log n)$