Q2:

First delete the old node, then increment the value, and lastly add back the node.

IncKey(k,d){

old <- Delete(k);

Insert(d + old);

}

Deletion costs , increment costs , and insertion costs . Intotal the cost is in .

Since deletion doesn’t break the AVL property, the tree is an AVL tree after “Delete(k)”; Since insertion also keeps the AVL property, the tree after increment is an AVL tree.

Q3:

I:

a)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| -inf |  |  |  |  |  |  |  |  |  | +inf |
| -inf |  |  |  |  |  |  |  |  | 53 | +inf |
| -inf |  |  |  |  |  |  |  |  | 53 | +inf |
| -inf |  |  |  |  |  |  |  |  | 53 | +inf |
| -inf | 9 | 12 |  |  |  | 50 | 51 |  | 53 | +inf |
| -inf | 9 | 12 | 26 | 37 | 44 | 50 | 51 | 52 | 53 | +inf |

b)

P contains: -inf, -inf, 23, 50, 50, 66, 66, 79

II:

a)

The algorithm decide to go down or go right by checking the number of keys skipped: the number of keys skipped > k means it is in range, go down; the number of keys skipped < k means it is in next few blocks, go right, and update k. If they are the same, the algorithm finds the key.

Select(L, k)

L: A skip list, k: index

p <- topmost left position of L

h <- L.depth , the height of the list

while (true){

if (dist(p, h) > k){

--h;

p <- below(p);

} else if (dist(p, h) < k){

k -= dist(p, h);

p <- after(p);

} else {

while (h > 0){

--h;

p <- below(p);

}

return p;

}

}

b)

After insertion, if the coin toss gives a height which is higher than current height, we need to increase the height of the current list(make the tower for two special keys higher), so that the inserted key is not higher than the two ends.

After deletion, we need to lower the tower for the two special keys, so that it matches the height of the highest key(1 level higher).

Q4:

a)

ABDCEFGH to EHGDABCF

Requires search sequence : D,G,H,E

ABDCEFGH

DABCEFGH

GDABCEFH

HGDABCEF

EHGDABCF

b)

ABDCEFGH to ADBCEGHF

Requires search sequence : D,G,H

ABDCEFGH

ADBCEFGH

ADBCEGFH

ADBCEGHF

c)

for search sequence DHHGHEGH,

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Compare + swap | D | H | H | G | H | E | G | H | Total |
| Before search | ABDCEFGH | DABCEFGH | HDABCEFG | HDABCEFG | GHDABCEF | HGDABCEF | EHGDABCF | GEHDABCF | HGEDABCF |
| MTF | 3+2(1) | 8+7(1) | 1+0(0) | 8+7(1) | 2+1(1) | 7+6(1) | 3+2(1) | 3+2(1) | 35+34(7) |
| Before search | ABDCEFGH | ADBCEFGH | ADBCEFHG | ADBCEHFG | ADBCEHGF | ADBCHEGF | ADBCEHGF | ADBCEGHF | ADBCEHGF |
| Transpose heuristics | 3+1 | 8+1 | 7+1 | 8+1 | 6+1 | 6+1 | 7+1 | 7+1 | 52+8 |

Note: If moving an element to the front and push back other elements requires only 1 swap, the number of swaps required is written in the bracket.

Q5: