Data Structures and Algorithms: Homework #6

Due on June 9, 2015 at 16:20

Instructors Hsuan-Tien Lin, Roger Jang

Tim Liou (b03902028)

6.1 Skip List, Binary Search Tree

(1) Do Exercise C-9.15 of the textbook.

For every node, also store the width of the link. The width is defined as the number of bottom layer links being traversed by each of the higher layer "express lane" links. Following is Example from `http://en.wikipedia.org/wiki/Skip_list#Indexable_skiplist`

```
  1                               10
 o---> o---------------------------------------------------------> o   Top
   1           3              2                    5
 o---> o---------------> o---------> o---------------------------> o
   1        2         1       2                    5
 o---> o---------> o---> o---------> o---------------------------> o
   1     1     1     1     1     1     1     1     1     1     1
 o---> o---> o---> o---> o---> o---> o---> o---> o---> o---> o---> o   Bottom

Head   1st   2nd   3rd   4th   5th   6th   7th   8th   9th   10th  NIL
       Node  Node  Node  Node  Node  Node  Node  Node  Node  Node
```

function Median
  node $\leftarrow$ head
  remain $\leftarrow \left\lfloor \frac{n}{2} \right\rfloor$
  while Below(node) $\neq$ null **do**
    node $\leftarrow$ Below(node)
    while remain $\geq$ Width(node) **do**
      remain $\leftarrow$ remain $-$ Width(node)
      node $\leftarrow$ After(node)
    end while
  end while
  return node
end function

Algorithm 1: Find the median element

We can use **Width** function to get the width in $O(1)$ since we store the value. This **Median** function involes two nested **while** loops. One performs a scan forward, and the other drops down to the next level. Since the height h of S is $O(\log n)$ with high probability, the number og drop-down steps is $O(\log n)$ with high probability. Let n_i be the number of keys examined while scanning forward at level i. We can easily observe that, after the key at the starting postion, each additional key examined in a scan-forward at level i cannot also belong to level $i+1$. If any of these keys were on the previous level, we would have encountered them in the previous level. Thus, the expected value of n_i is exactly equal to the expected number of times we must flip a fair coin before it comes up heads, That is 2. Hence, the expected

Problem 6.1 continued on next page...

amount of time spent scanning forward at any level i is $O(1)$. Since S has $O(\log n)$ levels with high probability, this **Median** function in S takes expected time $O(\log n)$.

(2) Do Exercise R-10.5 of the textbook.

Consider these two input orders, $\{1, 3, 2, 4, 5\}$ and $\{4, 2, 1, 5, 3\}$. Note that these two sets contain same entries $\{1, 2, 3, 4, 5\}$. We can easily find that they generate different trees.

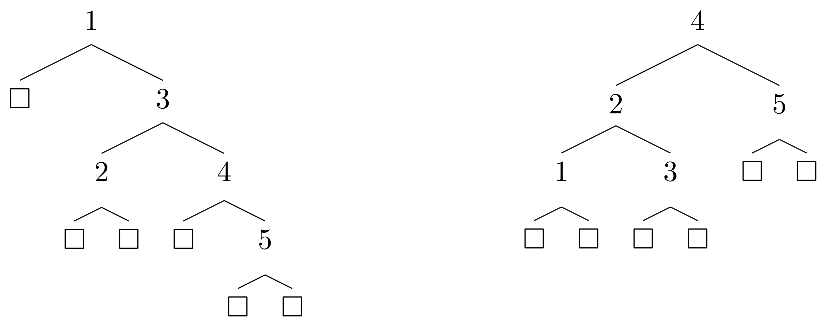
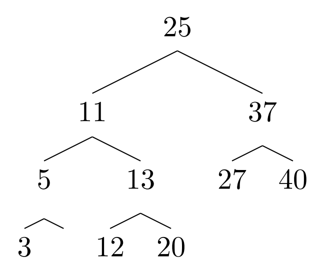


Figure 1: Left tree generates from $\{1, 3, 2, 4, 5\}$, and right tree generates from $\{4, 2, 1, 5, 3\}$.

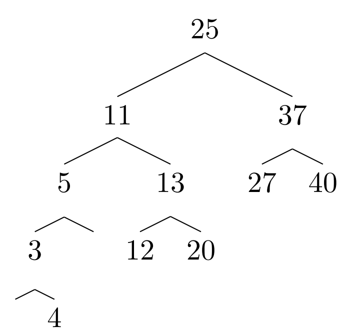
We conclude that the order of insertion does matter.

(3) Do Exercise C-10.12 of the textbook.

Consider this AVL tree.

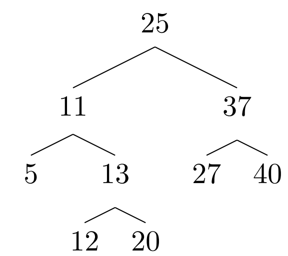


After insert 4, it becomes unbalanced.

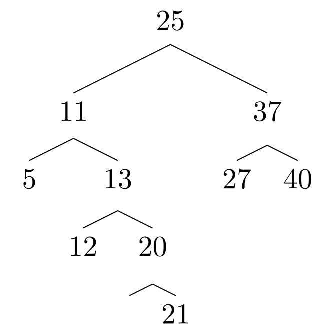


We can easily find that the unbalanced nodes are 5 and 25, which are nonconsecutive.

Then, consider this case.



After insert 21, it becomes unbalanced, and the unbalanced nodes are 13, 11 and 25, which are consecutive.



We conclude that the nodes that become unbalanced in an AVL tree during an `insert` operation may be nonconsecutive on the path from the newly inserted node to the root.

6.2 Balanced Binary Search Trees

(1) Write a program `hw6_2` that reads 32 strings (of length at most 128 that can be compared lexicographically) line by line (each line containing one string) from `stdin` and inserts them to the AVL tree (`avl.c`), height-bounded binary search tree (`bst.c`), and Red-Black tree (`rb.c`). Please output the resulting trees (pre-order) to `stdout` with specific format.

Input:

```
auto
break
case
char
const
continue
default
do
double
else
enum
extern
float
```

Problem 6.2 continued on next page. . .

```
for
goto
if
int
long
register
return
short
signed
sizeof
static
struct
switch
typedef
union
unsigned
void
volatile
while
```

Output:

```
if (do (char (break (auto , case ), continue (const , default )), extern ( else (double , enum ), for (float , goto ))), static (return (long (int , register ), signed (short , sizeof )), union (switch (struct , typedef ) , void (unsigned , volatile (, while )))))
auto (, break (, case (, char (, const (, continue (, default (, do (, double (, else (, enum (, extern (, float (, for (, goto (, if (, int (, long (, register (, return (, short (, signed (, sizeof (, static (, struct (, switch (, typedef (, union (, unsigned (, void (, volatile (, while ))))))))))))))))))))))))))))))))
do (char (break (auto , case ), continue (const , default )), if (extern ( else (double , enum ), for (float , goto )), return (long (int , register ), static (signed (short , sizeof ), union (switch (struct , typedef ), void (unsigned , volatile (, while )))))))
```

6.3 Disjoint Set

(1) Prove that the disjoint-set forest with this heuristic yields a worst-case running time for find and union within $O(\log n)$.

We can provide a variable stored the height in every node. By doing so, `union` only take $O(1)$ since we just change the pointer of root in shorter tree to the root of taller tree.

The worse-case running time of `find` depends on the depth of the tree.

We will prove that each `union` operation keeps trees within depth $O(\log n)$ by showing that within the forest, any tree of height h always contains at least 2^h nodes.

- Base case: It is trivial that $h = 0$ is right.
- Inductive hypothesis: assume true for $h - 1$.
- A tree of height h is created only by `union` two trees of height $h - 1$
- By inductive hypothesis, each subtree has $\geq 2^{h-1}$ nodes $\Rightarrow$ resulting tree has $\geq 2^h$

We proved that any tree of height h always contains at least 2^h nodes. That is, For any tree containing n nodes, the height is $O(\log n)$. We can conclude that the worst-case running time for `find` is also within $O(\log n)$.

(2) Suppose that you only need to output u rather than u and k for this problem. Write down the pseudo-code of an efficient algorithm based on the disjoint forest.

```
function Owner(id, OwnerTable)
    t ← Find(id)
    return OwnerTable[t]
end function
```

Algorithm 2: An efficient algorithm to output u based on the disjoint forest.

An array **OwnerTable** store the real owner of every sets in the index of the top node of the tree. Everytime we **Union** two sets, we will adjust **OwnerTable** properly.

(3) Suppose that the prices of your friend u's games are stored in a balanced BST as keys, and you have access to the size and the sum of all keys of any subtree of the BST in an $O(1)$ time, write down the pseudo-code of an efficient algorithm for outputting k for the particular u.

```
function GameNumCanBuy(m, root)
    num ← 0
    if the sum of all keys in root's left subtree > m then
        return GameNumCanBuy(m, root's left node)
    end if
    m ← m - the sum of all keys in root's left subtree
    num ← num + the size of root's left subtree
    while duplicate key count > 0 do
        if m < the key of the root then
            break the while loop
        end if
        m ← m - the key of root
        num ← num + 1
        duplicate key count ← duplicate key count − 1
    end while
    if the sum of all keys in root's right subtree > m then
        num ← num + GameNumCanBuy(m, root's right node)
    end if
    m ← m - the sum of all keys in root's right subtree
    num ← num + the size of root's right subtree
    return num
end function
```

Algorithm 3: Find the maximum number of games can buy with m dollars

(4) If we take the same heuristic as (1) and always insert the elements of the smaller BST into the bigger one, prove that processing all incidents of the first kind takes $O(n(\log n)^2)$ time.

The worst case is that we always **Union** two BSTs of same size. Suppose $n = 2^{k+1}$, **Union** all BSTs in this case,

$$
\begin{aligned}
2^k \times 1 \log_2 2 + 2^{k-1} \times 2 \log_2 4 + \cdots + 2 \times 2^{k-1} \log_2 2^k + 1 \times 2^k \log_2 2^{k+1} &\leq (k+1) \times 2^k \log_2 2^{k+1} \\
&= (k+1)^2 \times 2^k \\
&\leq (k+1)^2 \times 2^{k+1} \\
&= \frac{1}{(\log 2)^2} \times n(\log n)^2
\end{aligned}
$$

It implies that processing all incidents of the first kind takes $O(n(\log n)^2)$ time.