Data Structure and Algorithm

Homework #2 Solution

TA email: dsa1@csie.ntu.edu.tw

Problem 1. Unrolled Linked List (30%)

Sample code is shown on the course website.

Problem 2. Linked List (15%)

2.1. (5%)

```
void insert(Node **pNode, Data* data, int i){
    if(i<=1 || *pNode==NULL){
        Node *newNode = (Node*)malloc(sizeof(Node));
        newNode->data = data;
        newNode->next = *pNode;
        *pNode = newNode;
        return;
    }
    insert(&((*pNode)->next), data, i-1);
}
```

2.2. (5%)

```
void insert(Node** pNode, Data* data, int i){
    while(i>1 && *pNode!=NULL){
        pNode = &((*pNode)->next);
        --i;
    }
    Node *newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = *pNode;
    *pNode = newNode;
}
```

2.3. (3%)

2.1: $O(n)$

2.2: $O(n)$

2.4. (2%)

2.1: $O(n)$

2.2: $O(1)$

Problem 3. Tree, Tree and Tree!! (20%)

3.1. (10%)

a. (5%)

$$preorder : HDBACEGFMPROQNKLJ$$
$$inorder : ACBEGDFHRPQONMLKJ$$
$$postorder : CAGEBFDRQNOPLJKMH$$

b. (5%)

Because the first node in the *preorder* sequence will be the root of the tree, we can find this node in the *inorder* sequence easily. Then the *inorder* sequence will be splitted into two sub-sequences representing the left and right sub-trees. With two sub-sequences, we can get the node sets of two sub-trees and make the use of them to split the remaining sequence *preorder* into two parts of left and right sub-trees. Finally, we can construct the tree shown in Fig. 1 recursively by following the above instructions.

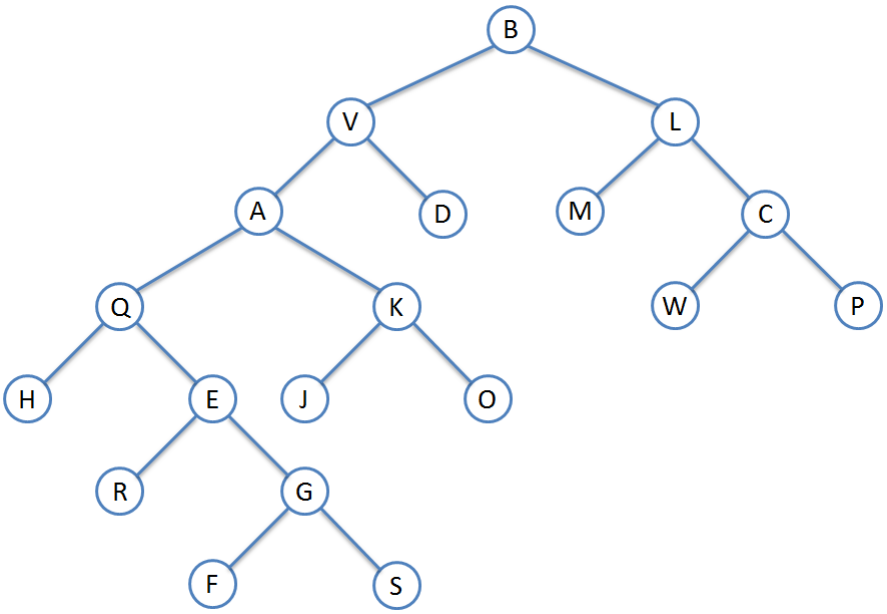

Figure 1: The tree of the first part of problem 3.1 (b)

Because both *preorder* and *postorder* sequences put the left sub-tree and right sub-tree together. If one of left and right side has no sub-tree, we cannot recognize the correct construction having sub-tree with *preorder* and *postorder* sequences. So we can construct the conflict cases easily as shown in Fig. 2.

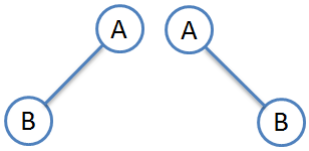

Figure 2: The tree of the second part of problem 3.1 (b)

3.2. (10%)

a. (5%)

Consider two cases as follows:

1. If $x < y$, then x the left child of y obviously. Because x is a leaf and the sole node of y's left sub-tree, x will be putted before y snugly in the *inorder* sequence. By the definition of BST, the *inorder* sequence is the sorted sequence, so y is the smallest in the values larger than x.
2. If $x > y$, then x the right child of y obviously. Because x is a leaf and the sole node of y's right sub-tree, x will be putted after y snugly in the *inorder* sequence. By the definition of BST, the *inorder* sequence is the sorted sequence, so y is the largest in the values smaller than x.

So y is either the smallest in the values larger than x or the largest in the values smaller than x.

b. (5%)

By default algorithm, the identical values will be inserted together and become a chain such that the height of the BST becomes n finally. So the complexity of insert operation will be $\mathrm{O}(n)$ and the complexity of querying a node which represents the identical value will still be $\mathrm{O}(\log n)$.

To improve the time complexity, we can make two counters for each node to record the number of nodes in left sub-tree and right sub-tree. Then the nodes will be distributed to left and right sub-trees equally, the height of BST also becomes $O(\log n)$. So the complexity of insert operation will be $\mathrm{O}(\log n)$ improved.

Problem 4. Online Dictionary (15%)

4.1. (2%) It takes $O(L)$ to compare two words' dictionary order by compare their characters one-by-one. So the time complexity to build is $O(N^2L + NL) = O(N^2L)$ for comparing two words $O(N^2)$ times and assign at most NL characters into the array. Of course, if we use any $O(NlgN)$ sorting algorithm instead, the time complexity becomes $O(NLlgN)$.

4.2. (3%) Since these words are already sorted, we can just print them out from $D[0]$ to $D[N-1]$. The time complexity is $O(NL)$ for printing at most NL characters.

For searching a query word, we can look through every word from $D[0]$ to $D[N-1]$ and compare the query word with it. So the complexity is also $O(NL)$. Also, since these word are sorted, we can use binary search instead. Hence the time complexity is improved to $O(LlgN)$.

4.3. (4%) The data structure is known as TRIE. It consists of a tree, initially contains a root only. Nodes in the tree have 26 pointers $c[]$, which will point to its 26 children representing 'a' to 'z', and a boolean tag which will record if there's a word end at here. Also, one may want another node pointer to remember its father.

When building the dictionary, we will add a word into the tree one-by-one. To add a word into the tree, we set a node pointer p at root. Each time we look the next character in the word and go down to the p's child which represents the character. For example, if the word is 'cat', we set p at root, then set $p = p->c['c']$, $p = p->c['a']$, $p = p->c['t']$. Note that the path from root to where p is at the end is just the word we'd like to record. If in the process the child we're interested in doesn't exist(the pointer points to NULL), just create a new node for it, and keep processing normally. At the end of the word, set the last node's tag to True.

Since we only do $O(1)$ thing for each character in the dictionary, the total time complexity is $O(NL)$. Also, since we for each character we will create at most one new node, there are at most $O(NL)$ nodes. The time complexity and space complexity are both $O(NL)$.

4.4. (3%) To search a word, we can copy the method how we add a word into the tree. Again, set a node pointer p at root. Each time we look the next character in the query word and go down to the p's child which represents the character. If the query word is really in the dictionary, this path we go through will be exactly same as the path when we add the word into the dictionary. So, if we find that there is a child we're interested in doesn't exist, or at the end the last node's tag is not True, the word does not belongs to the dictionary. And vice versa. The time complexity is same as add a word, i.e. $O(L)$.

4.5. (3%) To list-all, we go through the tree by pre-order DFS. In a node, we'll firstly check if its tag is True. If so, print the word out. Then we'll travel its children in the order from 'a' to

'z'. To print a word, we can trace back from the node to root(by recursively find the node's father), hence we go through the path that represents the word in the reverse order, so we can reversely print out the path.

The time complexity consists of two parts. One is the time for doing DFS, whose time complexity is same as the number of nodes, i.e. $O(NL)$. The other is the time for print out every word. Notice that for a node whose tag is True, the path for it to root and the length of word it represents are both $O(L)$, hence it takes $O(L)$ to print the word. Thus, the overall time complexity is $O(NL + N * L) = O(NL)$

Problem 5. Interesting Puzzle (20%)

If we don't emphasize the struct of the islands, please use the following structure as the defaunt.

```
struct Island
{
    struct Island *next;
};
struct Island *lakeside;
```

5.1. (3%)

```
int findLoopLength()
{
    struct Island *ptr1, *ptr2;
    int dis1, dis2;

    // will break when dis1 == n, O(n)
    for(ptr1 = lakeside, dis1=0;
        true;
        ptr1=ptr1->next, dis1++)
    {
        // Test the real distance between ptr1 and ptr2, O(
           dis1)
        for(ptr2 = lakeside, dis2=0;
            ptr1 != ptr2;
            ptr2=ptr2->next, dis2++);


        // If ptr1 visited all islands and back to the
           beginning of the loop.
        // dis1 will be n+1, but dis2 will be n - m + 1.
        // m is the length of the loop.
        if(dis1 != dis2)
            return dis1 - dis2;
    }
}
```

The total running time is $T(n) = O(1) + O(2) + \ldots + O(n) = O(n^2)$. Then, we only use 4 additional variables, so $O(1)$ additional space is used.

1 point for correctness. 1 point for time complexity analysis. 1 point for additional space complexity analysis.

5.2. (3%)

Editing the initial data structure isn't forbidden. However, we can copy it to the new data structure in $O(n)$ time. So, without loss of generality, we can edit the initail data structure as we want.

```
struct Island
{
    struct Island *next;
    int index = -1;
};
struct Island *lakeside;

int findLoopLength()
{
    struct Island *begin = lakeside;
    int counter = 0;

    // will stop in "n" iterations, O(n)
    while(begin->index == -1)
    {
        begin->index = counter++;
        begin = begin->next;
    }
    return counter - begin->index;
}
```

Time complexity is $O(n)$.

There're n "index" variables, $O(n)$ additional space.

1 point for correctness. 1 point for time complexity analysis. 1 point for additional space complexity analysis.

If using hashing, will be considered not correct, it may occur errors when n is larger.

If using a data structure (etc. sets, hashing tables) but not giving the implementation or the total size of structure, will be considered not correct. Because each step of algorithm should be definiteness and effectiveness.

5.3. (8%)

(a) (4%)

```
int check(int k)
{
    // walk k-1 steps, O(k)
    struct Island *begin = lakeside;
    for(int i = 0; i < k-1; i++)
        begin = begin->next;

    // test if there's a cycle in k-1 steps, O(k)
    struct Island *now = begin->next;
    int counter;
    for(counter = 1; counter < k-1 && begin != now;
       counter++)
        now = now->next;

    if(begin == now)
        return counter;
    else
        return -1; // not found
}
```

Suppose that the distance between the loop and lakeside is smaller than k, the pointer will point the island inside the loop after walking $k-1$ steps forward.

Then, it will reach this point again after walking *loopLength* steps. If $loopLength \leq k-1$, we can find out the loop length when trying to walk $k-1$ steps forward.

Otherwise, either the first or the second condition is not satisfied.

Time complexity is $T(n) = O(k+k) = O(k)$. We only use constant amount of variable, so the space complexity is $O(1)$.

1 point for correctness of algorithm. 2 point for descriptions for each part in algorithm. 1 point for time complexity analysis. If no additional space analysis, 1 point for penalty.

(b) (4%)

```
int findLoopLength()
{
    for(int k = 1; true; k *= 2)
    {
        int result = check(k);

        if(result != -1)
            return result;
    }
}
```

This algorithm will eventually stop. (1%)

Proof. First, we know k will greater than n in some time. Suppose that $k > n$, the condition "*the length of the loop is less than k if the distance between the loop and lakeside is less than k*" will be satisfied, because the length of the loop and the distance between the loop and lakeside are at most $n - 1$, so they're both less than n. So this algorithm will stop and return answer when $k > n$. □

Then, we will get $T(n) = 1 + 2 + 4 + \cdots + k$, such that $k = 2^j > n$ and $2^{j-1} \leq n$.

If mutiply 2 at the last ineuqality will get $2^j \leq 2n$.

$$\begin{aligned} T(n) &= (1 + 2 + 4 + \cdots + 2^{j-1}) + 2^j \\ &= (2^j - 1) + 2^j \\ &< 2 \times 2^j \\ &\leq 2 \times 2n \\ &= 4n \end{aligned}$$

The total running time of this algorithm is $O(n)$.

The check function uses $O(1)$ additional space and constant amount of variables are used at the other part, so the total additional space is $O(1)$.

1 point for correctness of algorithm. 2 points for time complexity. If no additional space analysis, 1 point for penalty.

5.4. (6%)

(a) (3%)

When `ptr1 == ptr2` only if they are both inside the loop. (1%)

Proof. If they both doesn't fall into the loop, the distance will increase 1 each iteration.

If one pointer doesn't fall into the loop but the other does, means that `ptr1 != ptr2`.

So, `ptr1 == ptr2` only if the are both inside the loop. □

`ptr1 == ptr2` must happen in some time. (2%)

Proof. Suppose that when `ptr1` falls into the loop, `ptr1` points a-th island and `ptr2` points b-th island, and the loop length is m.

For each iteration, `ptr2` chases `ptr1` 1 step, and the distance decreases 1 only, so that they won't miss each other.

If $a < b$, `ptr1 == ptr2` after $m - (b - a)$ iterations.

If $a > b$, `ptr1 == ptr2` after $a - b$ iterations.

Finally, two pointers will fall into the loop in $n - m$ iterations and then `ptr2` will meet `ptr1` in m iterations. In other words, it will happen in $(n - m) + m$ iterations. Now, we know that `ptr1 == ptr2` will happen in n iterations. □

*Students said that "becouse of different speed, **`ptr1 == ptr2`** will happen," will get only 1 point for this proof.*

(b) (3%)

```
int findLoopLength()
{
    //will stop in "n" iterations
    ptr1 = lakeside; ptr2 = lakeside->next;
    while(ptr1 != ptr2)
    {
        ptr1 = ptr1->next;
        ptr2 = ptr2->next->next;
    }

    // find out the cycle length in "n" iterations, O(n)
    struct Island *now = ptr1->next;
    int counter;
    for(counter = 1; ptr1 != now; counter++)
        now = now->next;
    return counter;
}
```

We already know the first part will stop in n iterations. It's proved at 5.4(a). So the time complexity is $T(n) = O(n + n) = O(n)$.

The algorithm only use 3 additional pointers and 1 variable, so the total additional space is $O(1)$.

1 point for correctness of algorithm. 2 points for time complexity analysis.

If no additional space complexity analysis, 1 point for penalty.

5.5. (Bonus 5%)

```
int findBeginningOfLoop(int m)
{
    struct Island *ptr1, *ptr2;
    ptr1 = ptr2 = lakeside;
    for(int i=0; i<m; i++)
        ptr2 = ptr2->next;

    while(ptr1 != ptr2)
        ptr1 = ptr1->next, ptr2 = ptr2->next;

    return ptr1;
}
```

Suppose that we already know the length of the loop m.

We know that after moving m steps from some island will return only if this island is in loop. so, we can test each island ordered to find out the first island in the loop, the beginning of the loop.

However, naïve testing will cost $O(m)$ for each island, the worse case will be $O(n^2)$. We can use other pointer to keep the island after moving m steps, and move them 1 step together for each iterations. Now, this algorithm cost $O(1)$ for each test, and almost n tests. The time complexity is $O(n)$.

The algorithm only use 2 additional pointers, so the total additional space is $O(1)$. *3 points for algorithm correctness. Some penaties for slight errors.*

1 points for time complexity analysis. 1 points for space complexity analysis.