Discussion 2C Notes (Week 10, March 11)

TA: Brian Choi

Solutions to Final Practice Problems

[Big-O]

1. Suppose there is an array of `N` (~ 10,000) elements in a random order. You want to run a search and look for a certain item. Using what you have learned in this course, what is the best you can do if:

(a) you run a search once? ("Is there 5 in the array?")
O(N), using linear search.
(b) you run a search 10,000 times? ("Is there 5? 16? 73? ...")
O(N). You can pre-process your array by traversing it and adding each element into a hash table. This process takes O(N). But after doing this once, you can simply perform your searches on this hash table, which takes O(1) per search. (Of course, the hash table must be pretty big!)

2. Consider two `vector<int>`'s `x` and `y`, each having `N` distinct integers. We want to merge `x` and `y` to create a third vector `z`, such that `z` has all integers that `x` and `y` have. Like `x` and `y`, `z` should not have any duplicate numbers. We are not concerned about keeping the elements in `x`, `y`, or `z` in any certain order.

Here is one algorithm:

```
void merge(const vector<int>& x, const vector<int>& y, vector<int>& z)
{
    z.clear();
    z.reserve(x.size() + y.size());

    for (int i = 0; i < x.size(); ++i)
        z.push_back(x[i]);

    for (int j = 0; j < y.size(); ++j)
    {
        bool duplicate = false;
        for (int i = 0; i < x.size(); ++i)
        {
            if (y[j] == x[i])
            {
                duplicate = true;
                break;
            }
        }
        if (!duplicate)
            z.push_back(y[j]);
    }
}
```

(a) What is the complexity of this algorithm? **$O(N^2)$**

(b) Here is a different implementation of `merge`. What is its complexity? **O(N log N)**

```
void merge(const vector<int>& x, const vector<int>& y, vector<int>& z)
{
    z.clear();
    z.reserve(x.size() + y.size());

    for (int i = 0; i < x.size(); i++)  // O(N)
        z.push_back(x[i]);

    for (int j = 0; j < y.size(); j++)  // O(N)
        z.push_back(y[j]);

    sort(z.begin(), z.end());   // O(N log N)

    int last = 0;
    for (int k = 1; k < z.size(); k++)  // O(N)
    {
        if (z[last] != z[k])
        {
            last++;
            z[last] = z[k];
        }
    }

    z.resize(last + 1);
}
```

(c) Which one performs better, (a) or (b)? **(b), because O(N log N) < O(N^2)**

(d) (Open-ended question) Is there any algorithm for `merge` that performs better than the version in (b)?

One possible idea is to create a hash table, and store all elements in x and y into this hash table, while eliminating duplicates (if the same value is in the hash table already, ignore the value). Then you can iterate through elements in this hash table and store them into z. This takes O(N).

[Trees]

3. Write `nodeCount`, a recursive function that counts the number of nodes in a tree rooted at `node`. This is a general tree, where a node can have a variable number of children. Use the following `Node` structure.

```
struct Node
{
   int val;
   vector<Node *> children;
};
```

```
int nodeCount(Node *node)
{
    int count = 1;

    for (int i = 0; i < node->children.size(); i++)
        count += nodeCount(node->children[i]);

    return count;
}
```

4. Write a one-line function that returns the number of edges in a tree, using the function you defined above.

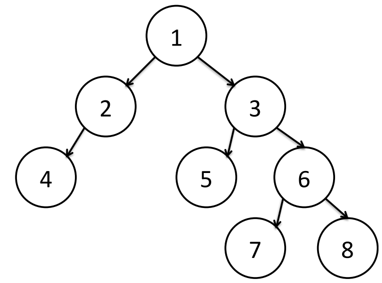
```
int edgeCount(Node *node)
{
    return nodeCount(node) - 1;
}
```

5. Write `leafCount`, a function that returns the number of leaves in the tree rooted at the node (a `Node` is a leaf if all of its children are `NULL`).

```
int leafCount(Node *node)
{
    if (node->children.size() == 0)
        return 1;

    int count = 0;
    for (int i = 0; i < node->children.size(); i++)
        count += leafCount(node->children[i]);

    return count;
}
```



6. The **closest common ancestor** of two nodes `n1` and `n2` is the closest ancestor node of both `n1` and `n2` (and it's the furthest from the root). Consider the graph on the right side. The closest common ancestor of 5 and 6 is 3, and the closest common ancestor of 4 and 7 is 1, etc. If `n1` and `n2` are the same, then `n1` (or `n2`) itself is the closest common ancestor of `n1` and `n2`.

Write the recursive function `cca` that takes in three parameters -- `current`, `n1`, and `n2`, that will return the pointer to the closest common ancestor node in the tree rooted at `current`. Assume both `n1` and `n2` are pointing valid nodes (which can be the same) in the tree, thus there should be a non-null return value. Also assume that the values in the tree are unique.

```
struct Node
{
   int val;
   Node *left;      // left child, NULL if none
   Node *right;     // right child, NULL if none
};

Node *cca(Node *current, const Node *n1, const Node *n2)
{
    if (current == NULL)
        return NULL;

    if (n1 == current || n2 == current)
        return current;

    Node *left = cca(current->left, n1, n2);
    Node *right = cca(current->right, n1, n2);

    if (left != NULL && right != NULL)
        return current;

    if (left != NULL)
        return left;

    return right;
}
```

7. Draw the height-2 binary tree whose postorder traversal is `U C N L A G E` (where height is the number of edges in the longest path between the root and a node).

```
     E
    / \
   N   G
  / \ / \
  U C L A
```

8. Draw the height-2 binary tree whose preorder traversal is `U C N L A G E`.

```
     U
    / \
   C   A
  / \ / \
  N L G E
```

[Stacks / Queues]

9. (a) Write `countFront`, which is a function that, given a `queue<char>`, returns the number of times the front value (the value returned when `front()` is called) appears in the queue. You may not create any auxiliary stack or queue. When the function returns, the queue must look the same way as it did when the function was called. If the queue is empty, return 0.

```
int countFront(queue<char>& q)
{
    if (q.empty())
        return 0;

    int temp = q.size(), count = 0;
    char frontChar = q.front();

    for (int i = 0; i < temp; i++)
    {
        char ch = q.front();
        q.pop();

        if (ch == frontChar)
           count++;

        q.push(ch);
     }

     return count;
}
```

(b) Write `countTop`, similar to `countFront` but works with a `stack<char>`. This time, you are allowed to use an auxiliary stack or queue (but not both).

```
int countTop(stack<char>& s)
{
    if (s.empty())
        return 0;

    char topChar = s.top();
    int count = 0;
    stack<char> aux;

    while (!s.empty())
    {
        char ch = s.top();
        s.pop();
        if (ch == topChar)
            count++;
        aux.push(ch);
    }

    // Recover originals.
    while (!aux.empty())
    {
        s.push(aux.top());
        aux.pop();
    }

    return count;
}
```

[Hash Table / Binary Heap]

Consider the following hash function.

```
int hashFunc(int x)
{
   return (x * 2) % HASH_SIZE;
}
```

Assume `HASH_SIZE = 10`. Here is the hash table's insert function:

```
void insert(int key)
{
    int index = hashFunc(key);
    hash_array[index].push_back(key);
}
```

where `hash_array` is an array of `list<int>`'s.

10. Draw the state of `hash_array` on the right side after the following calls. Assume this is a chaining hash table (each element in the array is a linked list of records). The first two elements are drawn in there for you.

0			
1			
2	1		
3			
4	7	37	
5			
6	23	53	83
7			
8	14	19	
9			

```
insert(7);
insert(1);
insert(23);
insert(14);
insert(19);
insert(53);
insert(37);
insert(83);
```

Is `hashFunc()` a good hash function? Why or why not?

It is NOT a good hash function. Odd-numbered buckets 1, 3, 5, 7, 9 are never used.

11. Consider the following array-based binary <u>maxheap</u>, which supports two operations, `removeMax()` and `insert(num)`.

15	10	14	7	9	8	11	4	3	5	6

How does it look after `removeMax()`?

14	10	11	7	9	8	6	4	3	5

How does it look after `insert(12)`?

14	12	11	7	10	8	6	4	3	5	9

[Data Structures & Big-O]

12. You are hired to design a website called `brutionary.com`, a `dictionary.com` variant. You are given a list of dictionary words (and their definitions) in a text file, and would like to preprocess it, such that you can readily provide information to users who visit your website and look up words. Assume that your dictionary is not going to be updated once it is preprocessed. We still want your system to "scale" -- that is, it should be able to efficiently take care of a lot of queries that may come in.

(a) First of all, the users should be able to look up words. Which option would you take, and why?

[Option A] Store `Word`s in a binary search tree.

[Option B] Store `Word`s in a hash table.

Option B lets you retrieve elements in O(1), while Option A takes O(log n) for each search operation. We are not too concerned about preprocessing time, but Option B outperforms Option A (i.e., O(n) < O(n log n)) nonetheless.

(b) You want to add a functionality that prints all words within a specified range (e.g., all words between abstain and abstract). Which option would you take, and why?

[Option A] Add a sorted linked list with pointers to `Word`s in the structure used in (a). Each time a range [`x`:`y`] is specified, we search `x` in the list, and then traverse the list to print each word, until we hit `y`.

[Option B] Add a sorted vector with pointers to `Word`s in the structure used in (a). Each time a range [`x`:`y`] is specified, we search `x` in the vector, and then traverse the vector to print each word, until we hit `y`.

Option B lets you use binary search (O(log n)) to find `x`, and then it is a linear scan from that point on. Going with Option A, you use a linear search to locate `x`, and then scan elements linearly. If the specified range is close to `n`, the scanning process dominates, so there isn't much difference. Yet if the range is considerably shorter than `n`, Option B is more attractive as it takes less time to find `x`.

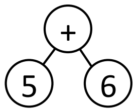
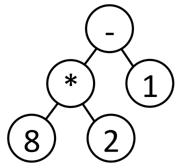
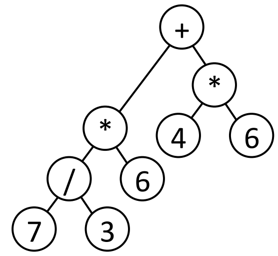
(c) In the right corner of the website, you want to display "`k` most popular words", where `k` is some integer, which are determined by queries that were received in the past hour, and is updated every hour. Assume queries are made on `n` distinct words, where `n` >> `k`. Which option would you take, and why?

[Option A] In the beginning of every hour, create a hash table (initially empty) that stores (`word`, `count`)-pairs. Each time a query for a word `x` comes in, look up `x` in the hash table (or add a new one if one does not exist), and increase the `count` for `x`. At the end of the hour, iterate through all (`word`, `count`)-pairs in the hash table and store them in a maxheap, using their `count`s as the keys. Then extract `k` words from the heap.

[Option B] In the beginning of every hour, create a vector (initially empty) that stores (`word`, `count`)-pairs. Each time a query for a word `x` comes in, look up `x` in the vector (or add a new one if one does not exist), and increase the `count` for `x`. At the end of the hour, sort the pairs in the vector in the decreasing order of their `count`s, and print the first `k` words.

In Option A, it takes O(1) to record each query, and at the end of the hour, all records in the hash table are and stored in a heap, which takes O(n log n). Retrieving k words from the heap takes O(k log n), or O(log n) if k is small. In Option B, it takes O(n) to record each query (one may argue that you can keep the vector sorted and use O(log n)-binary search to look up x, but in this case, inserting a new element into the vector takes O(n)). Sorting that takes place every hour is a O(n log n)-operation, and retrieving k words takes O(1), if k is considerably small. Since in both options the bulk processing that occurs at the end of each hour takes O(n log n), it is better to go with Option A, which takes less time to record each query. Because we want the system to scale, being able to process queries that may come in frequently (like, every 50 milliseconds) is crucial.

13. Define a function that generates an **arithmetic expression tree**, given an infix arithmetic expression. Here are a few examples:

		
5+6	8*2-1	7/3*6+4*6

Assume there are only four binary operators involved (+, -, *, /), and there is no parenthesis, and every number is a single digit. * and / have higher precedence than + and -, and if two consecutive operators have the same precedence, they must be evaluated from left to right.

We will write `createArithmeticExpTree`, a function that takes in a valid arithmetic expression as a string and generates an expression tree. Assume the following structure.

```
struct Node
{
    Node(char inVal, Node *inLeft, Node *inRight)
    : val(inVal), left(inLeft), right(inRight) {}
    char val;
    Node *left;
    Node *right;
};
```

Here is an implementation of `createArithmeticExpTree`, which uses a helper function called `addOp`.

```
Node *createArithmeticExpTree(string exp)
{
    if (exp.empty())
        return NULL;

    Node *root = new Node(exp[0], NULL, NULL);

    for (int i = 1; i < exp.size(); i += 2)
        root = addOp(root, exp[i], exp[i + 1]);

    return root;
}
```

(a) On the next page, provide the implementation for `addOp`.


```
Node *addOp(Node *root, char op, char digit)
{
    if (op == '+' || op == '-')
    {
        Node *digitNode = new Node(digit, NULL, NULL);
        Node *newRoot = new Node(op, root, digitNode);
        return newRoot;
    }
    else  // operator is either '*' or '/'.
    {
        if (isdigit(root->val) || root->val == '*' || root->val == '/')
        {
            Node *digitNode = new Node(digit, NULL, NULL);
            Node *newRoot = new Node(op, root, digitNode);
            return newRoot;
        }
        else
        {
            root->right = addOp(root->right, op, digit);
            return root;
        }
    }
}

// You can combine the two if bodies (they look identical), but to make the concept
// more understandable, I have split it into two different cases.
```

(b) Write `evaluate`, which takes in an arithmetic expression tree and returns the evaluated result of the expression. Assume the division (/) is integer division (i.e., decimal points are cut off).

```
int evaluate(Node *root)
{
    if (isdigit(root->val))
        return root->val - '0';

    switch (root->val)
    {
        case '+':
            return evaluate(root->left) + evaluate(root->right);
        case '-':
            return evaluate(root->left) - evaluate(root->right);
        case '*':
            return evaluate(root->left) * evaluate(root->right);
        case '/':
            return evaluate(root->left) / evaluate(root->right);
    }
}
```