

Discussion 1B Note (Week 10)

TA: Zhou Ren

Acknowledgement: Brian Choi's material

Final Practice Problems

Note: The work space provided for each problem may not be sufficient. Use scratch papers if needed.

[Big-O]

1. Suppose there is an array of N ($\sim 10,000$) 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?”)
- (b) you run a search 10,000 times? (“Is there 5? 16? 73? ...”)

You may place this data in a secondary data like and/or re-arrange the data to solve the problem. Support your argument using Big-O.

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?

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

```
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++)
        z.push_back(y[j]);

    sort(z.begin(), z.end());

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

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

(c) Which one performs better, (a) or (b)?

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

[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)
{
```

```
}
```

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)
{
```

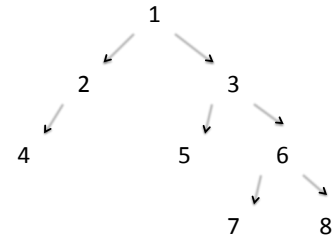
```
}
```

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)
{
```

```
}
```

6. The **closest common ancestor** of two nodes n_1 and n_2 is the closest ancestor node of both n_1 and n_2 (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 n_1 and n_2 are the same, then n_1 (or n_2) itself is the closest common ancestor of n_1 and n_2 .



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)
{
}

}
  
```

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).

8. Draw the height-2 binary tree whose preorder traversal is **U C N L A 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)
{
```

```
}
```

(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)
{
```

```
}
```

To refresh your memory, here are a few things you can do with stacks and queues:

<code>stack<char> s;</code>	<code>queue<char> q;</code>
<code>s.push('A');</code>	<code>q.push('A');</code>
<code>char c = s.top();</code>	<code>char c = q.front();</code>
<code>s.pop();</code>	<code>q.pop();</code>
<code>if (s.empty())</code>	<code>if (q.empty())</code>
<code>cout << "it's empty" << endl;</code>	<code>cout << "it's empty" << endl;</code>
<code>cout << s.size() << endl;</code>	<code>cout << q.size() << endl;</code>

[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
5	
6	
7	
8	
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?

11. Consider the following array-based binary maxheap, 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()`?

--	--	--	--	--	--	--	--	--	--	--

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

--	--	--	--	--	--	--	--	--	--	--

[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.

Assume a **Word** structure like the following is used to store each word and definition.

```
struct Word
{
    string word;
    string definition;
};
```

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

[Option A] Store **Words** in a binary search tree.

[Option B] Store **Words** in a hash table.

(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 **Words** 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 **Words** 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 .

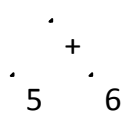
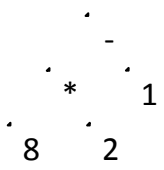
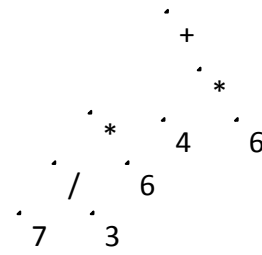
(c) In the right corner of the website, you want to display “ k most popular words”, where k is some integer, which are determined by queries that were received in the past hour, and is updated every hour. Assume queries are made on n distinct words, where $n \gg k$. Which option would you take, and 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 **counts** 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 **counts**, and print the first k words.

Note: There may, of course, be a better design choice to each problem than ones presented above. Feel free to discuss other possibilities with your peers.

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`.


```
// op: an operator to be added
// digit: a right-hand side operand that goes with op
```

```
Node *addOp(Node *root, char op, char digit)
{
```

```
}
```

(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)
{
```

```
}
```

Good luck on the final!