Data Structures and Algorithm

Mid Term Examination

Name: Lohitanvita Rompicharla **RUID:** 211009876

Prob1: Given: hash_map<student,set<class> > studentClass

There are S students and an average of C classes per student.
What is the big-oh of printing all students alphabetically and for each student, listing each of their classes alphabetically? Explain your answer.
What is the big-oh of determining who's taking ECE512? Explain your answer.
What is the big-oh of determining if Joe Smith is taking ECE512? Explain your answer.

Solution: Given S -> number of students
C -> average classes per student
Hash_map (unordered_map) with key as student struct and value as set with class struct

i) For listing each of student's classes alphabetically, we need to sort the elements of the set by choosing efficient sorting algorithm which has time complexity as O(n log n) =>
Therefor sorting the classes in set takes O(C log C) run time complexity
Also, for sorting set of classes for each student takes => S * (C log C)
Now for sorting all students alphabetically takes, we are required to sort all the keys of the hash_map-> considering efficient sorting algorithm with O(n log n) time complexity, the sorting of students takes =>O(S log S).
Therefore big-oh of printing all students alphabetically and for each student, listing each of their classes alphabetically is => **O(S*C log C) + O(S log S)**
Therefore, total run time complexity => **O(max((S log S), (S*Clog C)))**

ii) To determine which students are taking ECE512 class, we need to search the SET of classes. Therefore search time complexity of SET is O(log C) and to search for every student, the total run time complexity => **O(S*log C)**

iii) To determine if Joe Smith is taking ECE512 class, the search time complexity of unordered_map for finding Joe Smith is O(1) and search time complexity of set for searching ECE512 is O(log C). Therefor worst time complexity is **O(log C)**

Prob2: Given a randomly ordered array of N integers, what is the big-oh of the most efficient algorithm for determining if any single number in the array makes up more than 50% of the array? Explain your answer.

Solution: Considering an unsorted array of N integers, and length of array as 'len_arr'. The below algorithm helps finding any single number that is makes up(repeats) more than (len_arra/2).

```
Findmaxrepeat(int arr[])

Len_arr -> sizeof(array)/size(arr[0])
Count_map -> empty Unordered_map key: element of array, value: count of each element

for( i = 0;  i< len_arr, i++)
        if (arr[i] in count_map)
            update count_map[i] += 1
        else
            count_map.insert {arr[i] , 1}
for (key, value in count_map)
        if (value > len_arr/2)
                return True
        else
                return False
```

There are two for loops, the first for loop iterates through the whole array => O(n) where n is length of the array and second for loop iterates through the hashmap O(n)
Therefore the worst case run time complexity is O(n) + O(n) => **O(n)**

Prob3: Given a binary search tree, what is the algorithm to print it out in reverse order? What is the big-oh of this algorithm? Explain your answer. What is the big-oh of the algorithm if the tree is not bushy (e.g. if the values were inserted into the tree in alphabetical order) Explain your answer.

Solution: Algorithm for printing BST in reverse order: using BFS
levelorder(root)

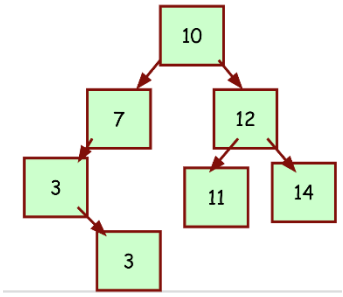
```
q —> empty queue
s —> empty stack
q.enqueue(root)
while (not q.isEmpty())
  node —> q.dequeue()
  s.push(node)
  if (node.right <> null)
    q.enqueue(node.right)
  if (node.left <> null)
    q.enqueue(node.left)

while (not s.isEmpty())
  node —> s.pop()
  print(node)
```

Time Complexity : O(n) and Space Complexity is O(n) where n is the number of nodes in BST

For a non-bushy tree:

Time Complexity : O(n) and Space Complexity is O(n) where n is the number of nodes in BST

Prob4: If you were to perform a pre-order traversal of this binary tree and insert each node, in the order it is processed, into a closed hash table of size 7, show the resulting hash table. Explain your answer.



Solution:

The resulting hash table after doing the preorder traversal of the given tree would be as follows:
[10, 7,3,3,12,11,14]
Since pre order traversal is Root -> Left-> Right, the root nodes will be inserted first in the hashtable followed by their left children and then their right children

Prob5: What is the big-oh of the following algorithm? Explain your answer.

```
void func(int arr[], int n){
   set<int> s;
   for (int i=0;i<n;i++){
     for (int j=0;j<i;j++)
         s.insert(arr[i] * arr[j]);
   }
}
```

Solution: Here the first loop iterates from i = 0 to length of the array and second loop iterates from j = 0 to j < i. Since first condition when i = 0 and j = 0 fails due to j< i condition failing. The first loop iterates a total of n – 1 time and second loop iterates a total of n – 2 times. Considering the sum of all n-2 iterations, i.e
1 + 2+ ……..n-3 + n-2 = ((n-2)/2)(2+n-3) = ((n-2)(n-3))/2 => ignoring the constants, the total runtime complexity = $O(n^2)$
Also, worst case time complexity for inserting the values into a SET is O(log n)
Therefore, the total worst case time complexity is **$O(n^2 \log n)$**

Prob6: Given the following unsorted array of integers, show it after it has been heapified using the efficient heap-sort algorithm that performs heapification in-place. Explain your answer.

6	16	3	19	13	72	13	12	99

Solution: Considering an unsorted array of integers, the below algorithm first heapifies (creates heap out of array) using max_heap and then using in-place (no extra memory) heap sort algorithm, sorts the whole array and prints the step by step output.

6	**16**	**3**	**19**	**13**	**72**	**13**	**12**	**99**
99	19	72	16	13	3	13	12	6
72	19	13	16	13	3	6	12	99
19	16	13	12	13	3	6	72	99
16	13	13	12	6	3	19	72	99
13	12	13	3	6	16	19	72	99
13	12	6	3	13	16	19	72	99
12	3	6	13	13	16	19	72	99
6	3	12	13	13	16	19	72	99
3	**6**	**12**	**13**	**13**	**16**	**19**	**72**	**99**

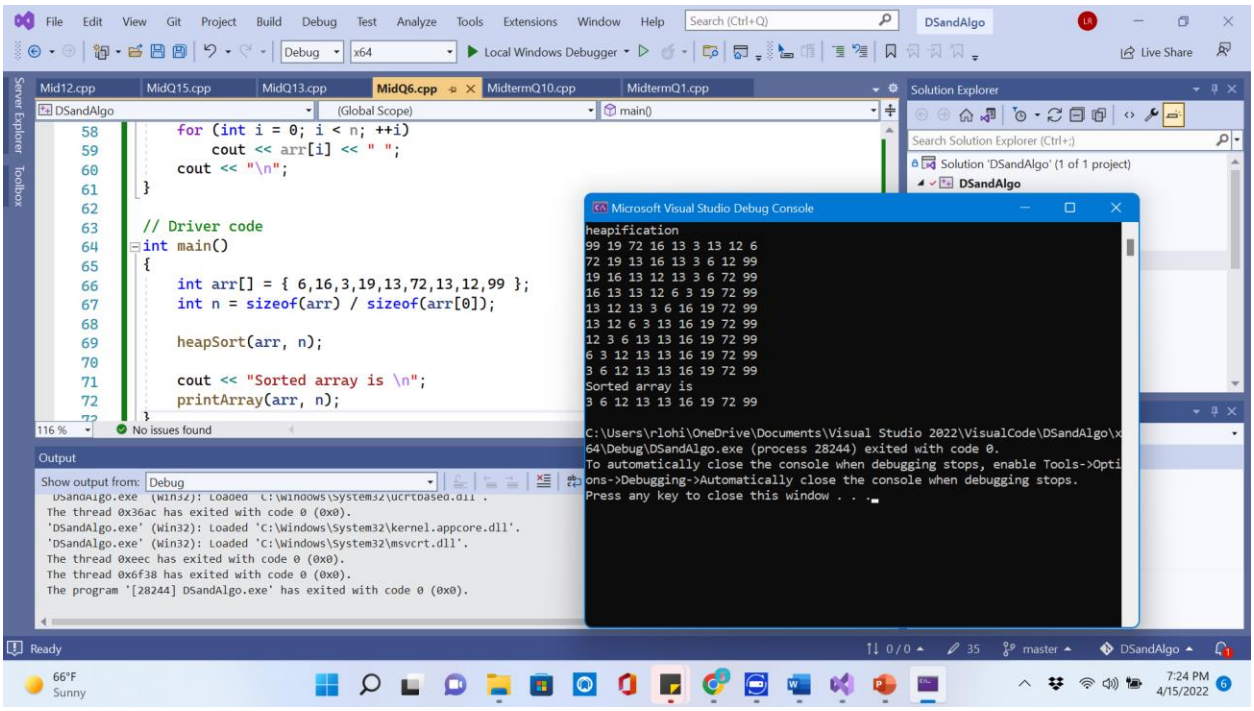
The algorithm followed for heapification is:

```
heapify(array)
  Root = array[0]
  Largest = largest( array[0] , array [2 * 0 + 1]. array[2 * 0 + 2])
  if(Root != Largest)
     Swap(Root, Largest)
```

Therefore time complexity to heapify is O(Log n) and to build heap is O(n), resulting in total worst case time complexity as **O(n Log n)** where is n is the number of nodes.

```
#include <iostream>

using namespace std;

// heapifies a the array with largest node as root node (max_heap)
void heapify(int arr[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int l = 2 * i + 1; // left = 2*i + 1
    int r = 2 * i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root and swaps the current node with largest
    if (largest != i) {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}
```

```cpp
// main function to do heap sort
void heapSort(int arr[], int n)
{
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
    cout << "heapification" << endl;
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << "\n";

    // One by one extract an element from heap
    for (int i = n - 1; i > 0; i--) {
        // Move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);

       // prints every step of the hep sort
        for (int i = 0; i < n; ++i)
            cout << arr[i] << " ";
        cout << "\n";
    }
}

/* A utility function to print array of size n */
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << "\n";
}

int main()
{
    int arr[] = { 6,16,3,19,13,72,13,12,99 };
    int n = sizeof(arr) / sizeof(arr[0]);

    heapSort(arr, n);

    cout << "Sorted array is \n";
    printArray(arr, n);
}
```



Prob7: You need to sort an array of integers which is already almost entirely sorted. Should you use the insertion sort, the merge sort, or the heapsort? Explain your answer.

Solution: To sort an array of integers which is already almost sorted, the most efficient algorithm is

Insertion Sort.

For almost sorted array, insertion sort will only need approximately **O(n)** passes, since it needs no or very few shiftings to perform. Whereas Merge Sort which uses divide and conquer algorithm takes O(n log n) time complexity even if 1 integer is to be sorted. Similarly, heapsort requires O (n log n) in total to heapify and creating the heap.

Prob8: You need to build a table to quickly look up student records by the students' last names; the set of students will grow and shrink each year as new students are enrolled and others graduate. Should you use a sorted array, a closed hash table, or a binary search tree? Explain your answer.

Solution: Assuming n is average number of students present in the table at any given point time.

For a sorted array:
Insertion: time complexity is O(n)
Deletion: time complexity is O(n)
Search: time complexity is O(log n)

For a binary search tree:
Insertion: time complexity is O(log n)
Deletion: time complexity is O(log n)
Search: time complexity is O(log n)

For a closed hash table:
Insertion: time complexity is O(1) when length of array does not change else O(n)
Deletion: time complexity is O(1) when length of array does not change else O(n)
Search: time complexity is O(1)

Since the student set will grow and shrink, a closed hash table and sorted array will not be good options as the time complexity to insert and delete items from them is linear. A binary search tree on the other hand will give intermediate time complexity for insertion, deletion, and search student names. Therefore, I would choose a **Binary Search Tree.**

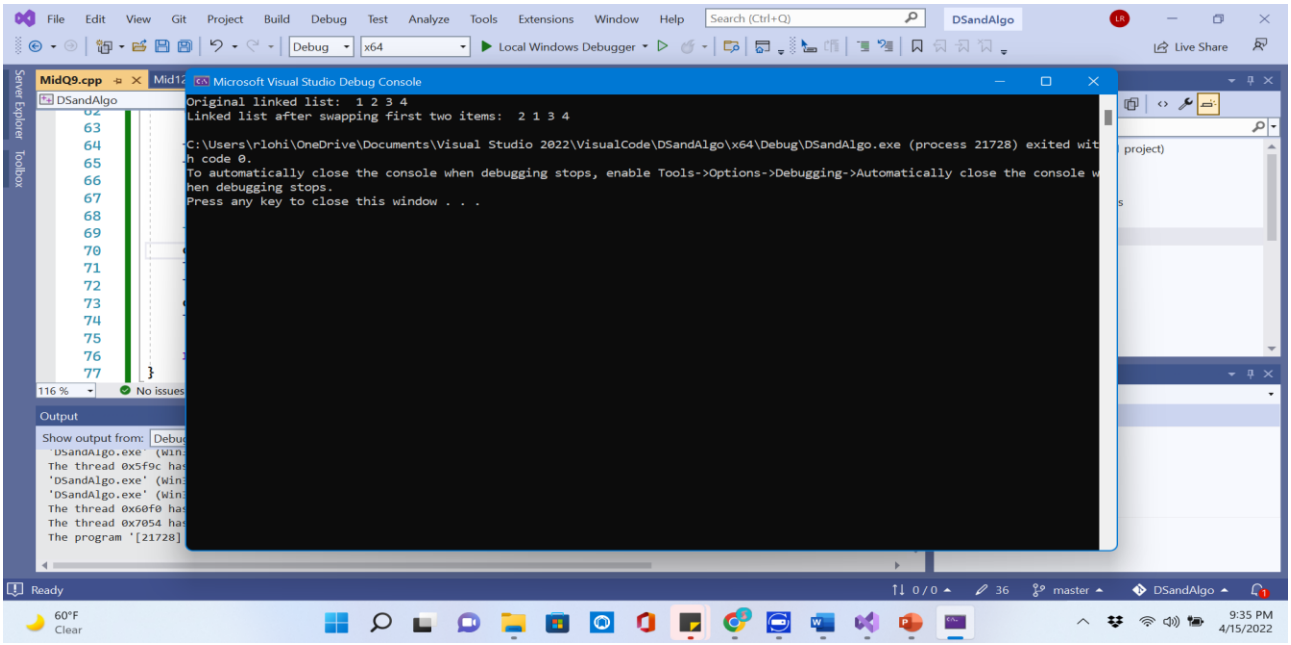
Prob9: Write a function called swapFirstTwo which swaps the first two items in a linked list (if there are at least two items).

```
struct NODE{
        int val;
        NODE *next;
};
```

Solution: The below code swaps the first two nodes of a linked list if there are two or more items in the linked list, otherwise returns nothing.

```
void swapFirstTwo(NODE* head)
{
       if (head == NULL || head->next == NULL)
       {
              return;
       }
       NODE* temp_swap;

       temp_swap = head; //temp pointer pointing head node
       head = head->next; // pointing head pointer to the second node
       temp_swap->next = head->next; // temp pointer next node is head next
       head->next = temp_swap; //pointing head next to temp pointer

}
```

OUTPUT:



Prob10: What does this program print? Explain your answer.

```
class MagicPower{
public:
   MagicPower() { cout << “MP\n”; }
   ~ MagicPower() { cout << “~MP\n”; }
};

class Wand{
public:
   Wand() { cout << “Wand\n”; }
   ~Wand() { cout << “~Wand\n”; }
private:
   MagicPower myPower;
};

class Wizard{
public:
    Wizard()  { cout << “Wizard\n”; }
    ~Wizard()  { cout << “~Wizard\n”; }
private:
    Wand myWand;
    Wand *secondWand;
};

int main(){
    Wizard *tom;
    tom = new Wizard;
   Wizard david[2];
}
```

Solution:
OUTPUT:

```
MP
Wand
Wizard
MP
Wand
Wizard
MP
Wand
Wizard
~Wizard
~Wand
~MP
~Wizard
~Wand
~MP
```

Firstly a Wizard Class object is called name tom, then a it is initialized. When the pointer is initialized, memory allocation in the heap happens. Therefore first member functions of class Wizard are called, then the private member function in class wizard points to Class Wand, then member function of class magicPower are called from the private variables where class object is created. Then they are removed from heap and printed in LIFO order. Later class wizard points to second index from David is stored. Therefore, similarly like above the flow continuous and it is stored in the heap. After printing out or accessing all the member functions, then when the heap is out of memory, the destructors are called.

Prob11: Show the copy constructor for the stomach class.

```
class Stomach{
public:
    Stomach(string items[], int n){
        count = n;

        if (n <= 3){  // if <= 3 items then use array
          for (int i=0;i<n;i++)
              contents[i] = items[i];
        }else{        // otherwise use new to alloc array
             moreThanThree = new string[n];
             for (int i=0;i<n;i++)
                 moreThanThree[i] = items[i];
        }
     }

    ~Stomach(){
        if (count > 3)
            delete [] moreThanThree;
```

```
   }

private:
  string contents[3];
  string *moreThanThree;
        int count;
};
```

Solution: A copy constructor is a member function that initializes an object using another object of the same class. A copy constructor has the following general function prototype:

```
   ClassName (const ClassName &old_obj);
```

Therefore, copy constructor of the above Stomach Class is:

```
class Stomach
{
public:
        Stomach(const Stomach& obj1)
        {
                count = obj1.count;
                if (count <= 3)
                {
                        for (int i = 0; i < count; i++)
                        {
                                contents[i] = obj1.contents[i];
                        }
                }
                else
                {
                        moreThanThree = new string[count];
                        for (int i = 0; i < count; i++)
                                moreThanThree[i] = obj1.moreThanThree[i];

                }
        }
private:
        string contents[3];
        string* moreThanThree;
        int count;

};
```

Prob12: Define a constructor for the NightClub class so it works as specified below:

Sam's joke should be:
"A man walks into a bar…ouch!" and he should tell it count times.

David's joke should be:
"A fish swims into a wall…Damn!" and he should tell it twice as many times as Sam.

```
Class BadComedian{
```

```
public:
   BadComedian(const string &joke, int times){
        myJoke = joke;
        numTimes = times;
   }
   void tellJoke() const {
        for (int i=0;i<numTimes;i++)
           cout << myJoke << endl;
   }
private:
   string myJoke;
   int numTimes;
};

class NightClub{
public:
   NightClub(int count) { }

   void doShow() {
         sam.tellJoke();
         david.tellJoke();
   }
private:
  BadComedian sam;
  BadComedian david;
};
```

Solution: Here we need to initialize variables to the constructor, so that it works as required. There are two ways to initialize the variables to a constructor. The common way is initializing the variables inside the constructor body, other efficient way is by using initializer list.
The initialization list lets us initialize the object to its final value which is noticeably faster than default initializations.

Option 1 : Efficient way using initializer list

```
class NightClub{
public:
   NightClub(int count) : sam("A man walks into a bar…ouch!", count), david("A fish swims into a
wall…Damn!" , 2*count) { }

   void doShow() {
         sam.tellJoke();
         david.tellJoke();
   }
private:
  BadComedian sam;
  BadComedian david;
};
```

Option 2 : initializing inside constructor body

```
class NightClub{
public:
   NightClub(int count)
{
          sam = BadComedian (“A man walks into a bar…ouch!”, count);
          david = BadComedian (“A fish swims into a wall…Damn!”, 2*count);
 }

   void doShow() {
        sam.tellJoke();
        david.tellJoke();
   }
private:
  BadComedian sam;
  BadComedian david;
};
```

Prob13: Write a recursive function called freeList that accepts a pointer to a doubly linked list node. The function must print all of the items in order and also delete all nodes except for the first in the linked list.

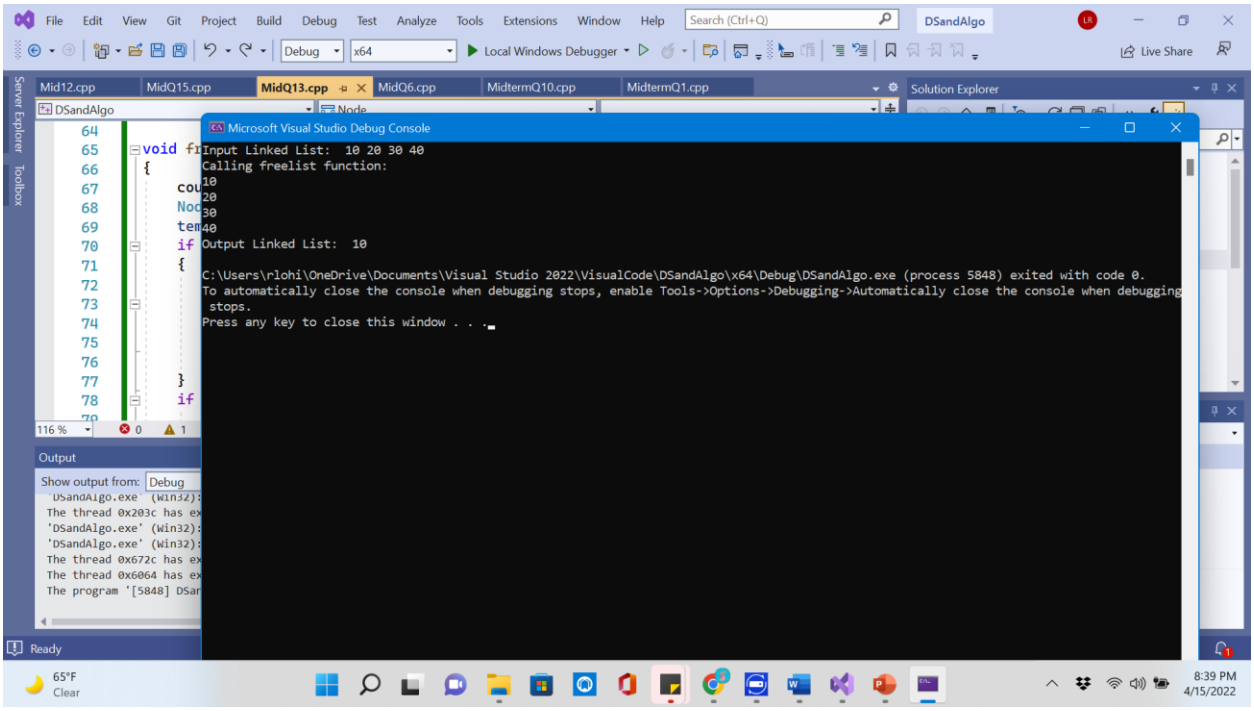
```
struct NODE{
          int val;
          NODE *next, *prev;
};

void main(){
  NODE *head;  // ptr points to the first node (e.g. head) of the linked list
  head = createSomeLinkedList();
  freeList(head);
}
```

Solution:

```
struct NODE{
          int val;
          NODE *next, *prev;
};

void freeList(NODE* node)
{
    cout << node->val << " "; //print the node value
    NODE* temp = new NODE();
    temp = node->next;
    if (node->prev != NULL) //to avoid deleting head node
```

```
    {
        node->prev->next = node->next;
        if (node->next != NULL) { //skip this for tail node
            node->next->prev = node->prev;
        }
        free(node); //delete node
    }
    if (temp != NULL) { //end recursion for tail node
        freeList(temp); //recursive call
    }
    return;
}

void main(){
  NODE *head;  // ptr points to the first node (e.g. head) of the linked list
  head = createSomeLinkedList();
  freeList(head);
}
```

OUTPUT:



Prob14: Create a set of C++ classes using inheritance:

1. All Sea Animals can make noise using a makeNoise function that takes no arguments and returns nothing
2. You must specify a Sea Animal's weight when its born
3. You can get a Sea Animal's weight by calling its weight() function which returns its weight.
4. All Sea Animals can eat(…) another Sea Animal (passed in by pointer) and gain the weight of the consumed animal (the eaten animal should be deleted)
5. Otters have a makeNoise method that prints "Bark".

6. Squid have a makeNoise method that prints “Squeak”.
7. Giant Squid, a species of Squid, burps after it eats another animal by printing “Burp” to the screen.
8. Giant Squid always weight 1000 pounds.

Solution:

```
#include <iostream>

using namespace std;

class SeaAnimal                         // Base class SeaAnimal
{
protected:
	int weight;
public:
	SeaAnimal(int lbs)
	{
		int weight = lbs;
	}
	virtual void makeNoise()            // methos overrided using virtual keyword
	{}

	int weight()
	{
		return weight;
	}
	virtual void eat(SeaAnimal* anotherAnimal)  // method overrided using virtual
	{
		weight += anotherAnimal.weight();
		delete anotherAnimal;
	}

};

class Otter : public SeaAnimal    // Otter class inherits from SeaAnimal base Class
{
public:
	void makeNoise()
	{
		cout << "Bark" << endl;
	}
};

class Squid: public SeaAnimal      // Squid is the child class inherited from
                                        SeaAnimal class
{
public:
	void makeNoise()              // oveveriding the makenoise member function of
                                        base class
	{
		cout << "Squeak" << endl;
	}
};

class GiantSquid:public Squid    // Child class of inherited class Squid
{
```

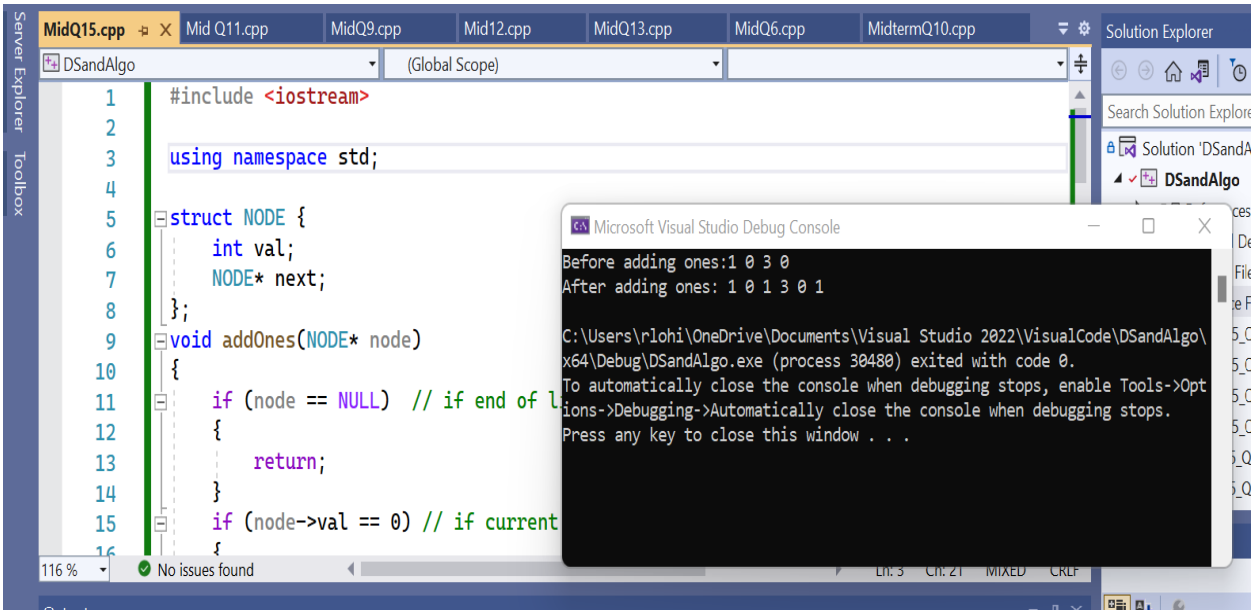
```
public:
	int weight = 1000;

	void eat(SeaAnimal* anotherAnimal)		// overriding the base class
							   member function eat()
	{
		weight += anotherAnimal.weight();
		delete anotherAnimal;
		cout << "Burp" << endl;
	}

};
```

Prob 15: Write a recursive function called addOnes that finds all nodes containing a value of 0 in a linked list and adds a new node after each with a value of 1.

Solutoin: Below is the addOnes function that creates a new node with value 1 after each node having value 0.

```
	struct NODE {
	    int val;
	    NODE* next;
	};
	void addOnes(NODE* node)
	{
		if (node == NULL)  // if end of linked list is reched, then recuresion
				   ends
		{
			return;
		}
		if (node->val == 0) // if current node value is 0 then create a new
				    node withvalue 1
		{
			NODE* curr_node = new NODE();
			curr_node->val = 1;
			curr_node->next = node->next; // adding the new node after the
							      current node
			node->next = curr_node;
		}
		addOnes(node->next);  // recursive call
	}
```

OUTPUT:



Prob16: Write a function called Balanced that determines whether or not a parenthesized expression is properly balanced. Your function should process strings with the following types of delimiters: ( ) { } [and]. Your function should accept two arguments: a string argument and a reference to an integer. The first, string argument contains the input expression that should be evaluated. The second, integer reference should have its value set to the maximum "nesting" depth of the expression. The function should return a bool: true if the expression is properly parenthesized, and false otherwise.

Here are examples of valid, balanced strings:

bletch	MaxDepth = 0
{}	MaxDepth = 1
[( )( )]	MaxDepth = 2
{goober[{face}]}	MaxDepth = 3
(a(b((c)))d)[(ef{g})]	MaxDepth = 4

Here are examples of invalid strings:

Snitch[
[fe[fi[fo}fum]ack)
((start()())

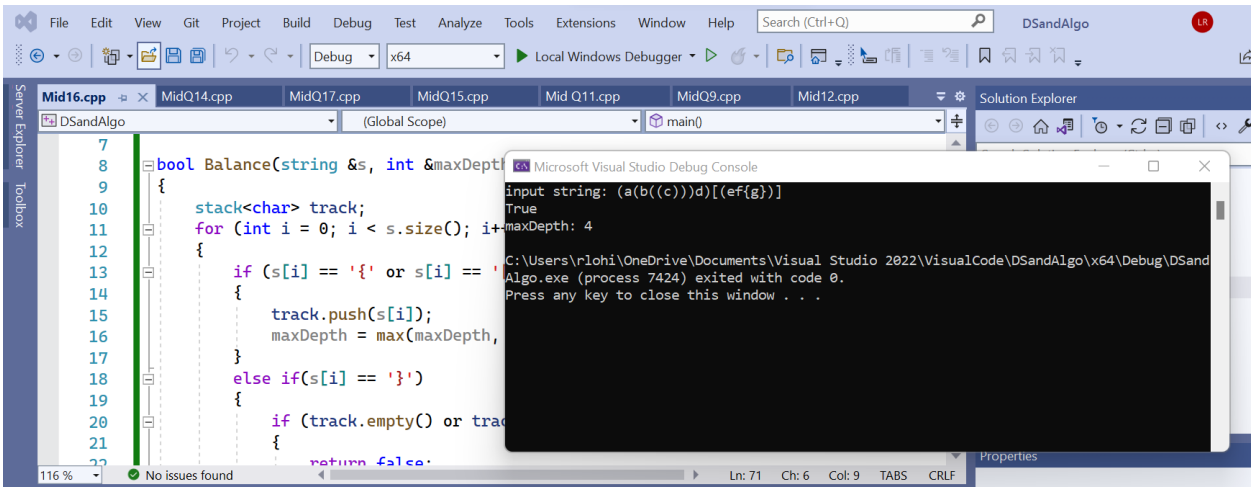
Solution:

```
#include <algorithm>

using namespace std;

bool Balance(string &s, int &maxDepth)
{
	stack<char> track;
```

```
	for (int i = 0; i < s.size(); i++)
	{
		if (s[i] == '{' or s[i] == '[' or s[i] == '(')
		{
			track.push(s[i]);
			maxDepth = max(maxDepth, (int)track.size());
		}
		else if(s[i] == '}')
		{
			if (track.empty() or track.top() != '{')
			{
				return false;
			}
			else
			{
				track.pop();
			}
		}
		else if (s[i] == ']')
		{
			if (track.empty() or track.top() != '[')
			{
				return false;
			}
			else
			{
				track.pop();
			}
		}
		else if (s[i] == ')')
		{
			if (track.empty() or track.top() != '(')
			{
				return false;
			}
			else
			{
				track.pop();
			}
		}
	}
	if (track.empty())
	{
		return true;
	}
	return false;
}

int main()
{
	string s;
	int maxDepth = 0;
	cout << "input string: ";
	cin >>s;
	bool result = Balance(s, maxDepth);
	if (result =1)
	{
		cout << "True" << endl;
```

```
        }
        else
        {
                cout << "False" << endl;
        }

}
```

OUTPUT:

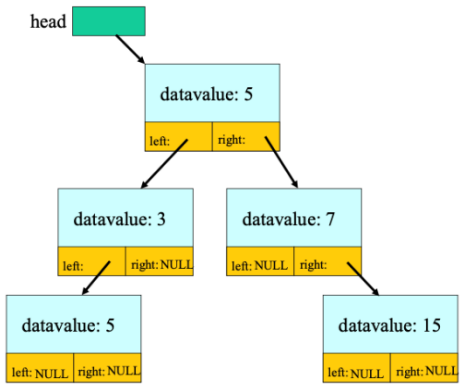


Prob17: A "binary tree" is a data structure that employs a special type of linked list node. In a binary tree, each linked list node has two next pointers. Here is an example binary tree node:

```
struct btnode
{
   int  datavalue;
   btnode *left;
   btnode *right;
};
```

Write a member function called findValue that accepts a pointer to the head of the tree and an int parameter and returns an int. The function must search through each node of the binary tree and count the number of times the value was found in the tree, then return this value.

First implement your function using a queue.
Then implement your function using recursion.



Solution:

```
struct btnode
{
    int  datavalue;
    btnode* left;
    btnode* right;
};

int findValue(struct btnode* head, int n) // approach using queue (level order
traversal)
{
    queue<btnode*> qu;
    int value = 0;
    if (head == NULL)
    {
        return 0;
    }
    qu.push(head);

    while (!qu.empty())
    {
        struct btnode* node = qu.front();
        qu.pop();
        if(node->datavalue == n) // if matching data value found, increament value
        {
            value += 1;
        }

        if (node->left != NULL) // push left child node to queue
        {
            qu.push(node->left);
        }
        if (node->right != NULL) // push right child node to queue
        {
            qu.push(node->right);
        }

    }
    return value;
}
```

```
// approach using recursion (inorder traversal)
int findValue(struct btnode* head, int n)
{
    int value = 0;
    inorderTraversal(head, n, value);
    return value;

}

void inorderTraversal(struct btnode* node, int n, int &value)
{
    if (node == NULL)
    {
        return;
    }
    inorderTraversal(node->left, n, value);
    if (node->datavalue == n)
    {
        value += 1;
    }
    inorderTraversal(node->right, n, value);
}
```