Part 1 (50%) – written answers

Question 1:

a. p=&i; no error, p point to address of i
b. p=*&i; error, *&I is content at address of i, equivlent to i, and pointer p cannot be signed to an integer
c. p=&*i; error , &*i=&(*i), i is an integer, *i will produce an error
d. i=*&*p; no error, *&*p=*(&*p), which is *p (see (e) for def. of *&p), *p is content of p, an integer
e. i=*&p; error, *&p is content at (address of pointer p), which is p. integer i cannot equal to a pointer.
f. i=&*p; error, &*p is address of (content of pointer p), equivlent to p. integer i cannot equal to a pointer.
g. p=&*&i; no error, &*&I=&(*(&i)), address of content of pointer point at address of i, which is &i
h. q=*&*p; error, *&*p = *(&*p), which is *p (see (f) for def. of &*p), so q = *p, an error
i. q=**&p; error, **&p = *(*&p), which is *p (see (e) for def. of *&p), so q = *p, an error
j. q=*&p; no error, *&p is content at (address of pointer p), which is p. so q = p
k. q=&*p; no error, &*p is address of (content of pointer p), equivlent to p. so q = p

Question 2:

big O notation set asymptotic upper bound for the growth while big theta notion set upper and lower bounds. The faster it grows as input n grows, the more run time it requires to run the program, the less efficient the algorithm.

f(n) is O(1) means there is a constant c, that for all n, $f(n) \leq c$, that is f(n) grows at most as fast as a constant function (the program's run time is at most a constant c).

f(n) is theta(1) means there are constants c and d, such that for all n, $d \leq f(n) \leq c$, that is f(n) grow at least as fast and at most as fast as a constant function (the program's run time is bounded in between constant d and c).

Question 3:

Suppose $f1(n)= n^2$, $f2(n)= n$, $g(n) = n^2$, both f1(n) and f2(n) are $O(n^2)$ since both functions grow no faster than n^2 and $O(n^2)$ is the upper bound of both. But f1(n) is not O(f2) = O(n) since $f1(n)= n^2$ grow faster than n and O(n) cannot be the upper bound of it.

Question 4:

First loop is being executed k times, and second loop will be executed n times. Since k is always smaller or equal than n, we can set the time complexity to $O(n^2)$, or O(kn) for a tighter bound

Question 5:

```
void insertMiddle(const int newElem){
   Node* v = new Node;
   v->elem = newElem;
  //if linked list is empty
   if (head == nullptr)
```

```
        head = v;
      Node* ptr = head;
      int size = 0;
      int count = 0;
    //find size of the linked list
      while (ptr!=nullptr){
        ptr = ptr->next;
        size++;
      }
     //find middle
      if((size % 2) == 0)
        count = size / 2;
      else
        count = (size+1)/2;
      ptr = head;
    //traverse to middle
      while (count>1){
        ptr = ptr->next;
        count--;
      }
    //if linked list has only 1 node
      if (ptr->next == nullptr){
        ptr->next = v;
        v->prev = ptr;
        v = tail
      }
      v->next = ptr->next;
      v->prev = ptr;
      ptr->next->prev = v;
      ptr->next = v;
 }
```

Question 6:

```
    void sort(stack<int> &S)
    {
       stack<int> temp;
       //swap contents of two stack, S is now a empty stack
       temp.swap(S);
       //repeat until all elements move back to S and in ascending order
       while (!temp.empty())
       {
          int num = temp.top();
          temp.pop();
          //ipop all numbers in S that's smaller than this number back to temp.
          while (!S.empty() && S.top() < num )
          {
             temp.push(S.top());
             S.pop();
```

```
            }
            //push this element (current max) to S
            S.push(num);
          }
      }
```

Question 7a:

```
    void transfer(stack<int> &stack1, stack<int> &stack2){
        stack<int> temp;
       //move elements from stack1 to temp stack then from temp to stack2
        while (!stack1.empty()){
            temp.push(stack1.top());
            stack1.pop();
        }
        while (!temp.empty()){
            stack2.push(temp.top());
            temp.pop();
        }
}
```

Question 7b:

```
    void transfer(stack<int> &stack1, stack<int> &stack2){
        int bottom;
        int num1=0;
        int num2=0;
        while (!stack1.empty()){
            stack2.push(stack1.top());
            stack1.pop();
           //number of available elements
            num1++;
        }
       //repeat until no available elements, which stack2 will be in correct order
        while (num1!=0){
           //pop top of stack 2, store it in a variable
            bottom = stack2.top();
            stack2.pop();
           //decrease available elements
            num1--;
            num2 = num1;
           //pop all available elements back to stack1
            while (num2!=0){
                stack1.push(stack2.top());
                stack2.pop();
                num2--;
        }
           //push stored value to stack2, this element will become not available
            stack2.push(bottom);
```

```
        //pop elements from stack1 back to stack2
        while (!stack1.empty()){
            stack2.push(stack1.top());
            stack1.pop();
        }
    }
}
```

Question 8:

```
    int findLength(){
        //pass head of linked list to recursive function
        return recursiveLength(head);
    }
    int recursiveLength(Node *current){
        //base condition, when node is null, stop recursion
        if (current == nullptr)
            return 0;
        //count is sum of current and all its children nodes
        int count = 1+ recursiveLength(current->next);
        return count;
}
```

Question 9:

```
//negative input values will be considered
    int multiply(int x, int y){
        //reduce recursion call improve efficiency
        if (abs(x)<abs(y))
            return multiply(y, x);
        if ((x<0 && y<0) || (x>0 && y<0)){
            x = -x;
            y = -y;
        }
        if (x == 0 || y == 0)
            return 0;
        x= x + multiply(x, y - 1);
        return x;
    }
```

Question 10a:

```
    int countNodes(Node* root)
    {
        int count = 1;
        //return 0 when node is null
        if (root == nullptr) {
            return 0;
```

```
    }
    //base condition, reach leave nodes, return 1
    else if (root->left == nullptr && root->right == nullptr)
        return 1;
    //if not base condition, go down the tree, return sum of current and all children nodes
    else {
        return count + countNodes(root->left) + countNodes(root->right);
    }
}
```

Question 10b:

```
int countLeaves(Node* root)
{
    if (root == nullptr) {
        return 0;
    }
    //base condition, when a node has no child, then it's a leaf, stop recursion
    else if (root->left == nullptr && root->right == nullptr)
        return 1;
    else {
    //if not base condition, continue go down the tree
        return countLeaves(root->left) + countLeaves(root->right);
    }
}
```

Question 10c:

```
int height(Node* root)
{
    int count = 1;
    if (root == nullptr) {
        return 0;
    }
    //base condition, if a node has no child, then it's a leaf(or it's a 1 node tree), height is 0
    else if (root->left == nullptr && root->right == nullptr)
        return 0;
    else {
    //if not base condition, go down the tree, return max depth up to current node
        return count + max(height(root->left),height(root->right));
    }
}
```

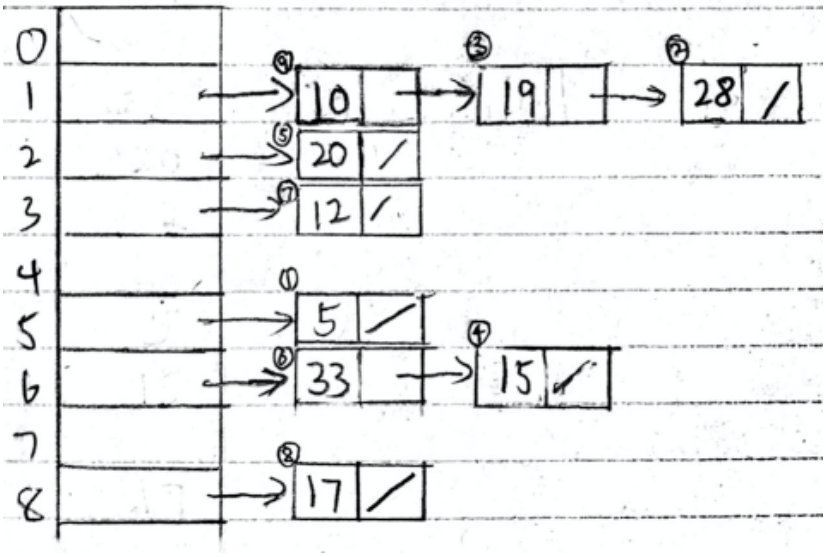
Question 11:

In binary trees, the sequence of preorder traversal is parent node -> left child -> right child, and the sequence of inorder traversal is left child -> parent node -> right child. Hence, for binary trees that don't have any left child for any node of the trees will have the same sequence for preorder and inorder traversals.

Question 12:
The stable sorting algorithms we encountered in this semester are bubble sort, insertion sort, merge sort, count Sort, radix sort. Stable sorting algorithms will maintain relative order of objects that have same key values after the sorting. One of the pratical example mentioned in class is that when we want to sort a list of students that have two key values, name and grade(A,B,C,D,E,F). First we sort it based on alphabetical order of names, second we sort the list based on the grades. However, for the second sorting, we still want the names to be maintained in alphabetical order. Stable sorting algorithms are useful and necessary in this case since multiple students will have same grades, and we don't want the sorting to change alphabetical order of names. After the sorting, students will be sorted into differect grade categories while maintaining the alphabetical order from previous sorting.

Question 13:
Table size is 9, index is 0 to 8. Each key will map to a index value that is calculated by hash function h(k)=k%9 : 5->5, 28->1, 19->1, 15->6, 20->2, 33->6, 12->3, 17->8, 10->1; collision happen at h(28) = h(19) = h(10) = 1, and h(15) = h(33) = 6. Each index will contain a linked list that has all elements with the same index value. Since hash table doesn't allow duplicated value, at each insertion, need to check if current value matches the previous inserted values.

Sequece: insert 5->5, linked list at index 5 is empty, insert 5 to head of linked list. Insert 28->1 , linked list at index 1 is empty, insert 28 to head of linked list. Insert 19->1 , linked list at index 1 is not empty, traverse the linked list, does not find key value 19, insert 19 to head of the linked list. Insert 15->6, linked list at index 6 is empty, insert 15 to head of linked list. Insert 20->2, linked list at index 2 is empty, insert 20 to head of linked list. Insert 33->6, linked list at index 6 is not empty, traverse the linked list, does not find key value 33, insert 33 to head of the linked list. Insert 12->3, linked list at index 3 is empty, insert 12 to head of linked list. Insert 17->8, linked list at index 8 is empty, insert 17 to head of linked list. Insert 10->1, linked list at index 1 is not empty, traverse the linked list, does not find key value 10, insert 10 to head of the linked list.
Hash table(sequence is marked):



Question 14:
In undirected graph, edges don't have direction, and the degree of of each vertex is the number of edges connected to it. In directed graph, edges have direction, and each vertex has in degree and out degree. In degree is the number of edges that coming to the vertex, and out degree is the number of edges that leaving the vertex.

Undirected graph: each edge is associated with two vertices, and the same edge is counted as degree by two vertices twice. Therefor the sum of the degrees of all vertices is twice number of the total edges of graph G=(V,E).

Directed graph: each edge is associated with two vertices, each edge can be in degree or out degree of a vertex but cannot be both, so edges will be counted as in degree once and out degree once. Therefore, the sum of in degrees of all vertices is equal to sum of out degrees of all vertices, and is equal to number of edges of graph G=(V,E). Hence, the sum of the degrees of all vertices (include in degree and out degree) is twice number of the total edges of graph G=(V,E).