Q2.

Asymptotic Analysis for List-based Dictionary			
	Best case time cost	Worst case time cost	Average case time cost
clear	Ω(1)	O(1)	Θ(1)
insert	Ω(1)	O(1)	Θ(1)
remove	Ω(1)	O(1)	Θ(1)
removeAny	Ω(1)	O(1)	Θ(1)
find	Ω(n)	O(n)	Θ(n)
size	Ω(n)	O(n)	Θ(n)

Q4.

Asymptotic Analysis for Double List-based Dictionary			
	Best case time cost	Worst case time cost	Average case time cost
clear	Ω(1)	O(1)	Θ(1)
insert	Ω(1)	O(1)	Θ(1)
remove	Ω(1)	O(1)	Θ(1)
removeAny	Ω(1)	O(1)	Θ(1)
find	Ω(n)	O(n)	Θ(n)
size	Ω(n)	O(n)	Θ(n)

Asymptotic analysis List-based: clear()

```
public void clear()          {
                    pair.clear();
          }

//method from LList called
public void clear() {
                    head.setNext(null);
                    curr = tail = head = new Link<E>(null);   // O(cons.)
                    cnt = 0;
          }

/method from Link called
Link<E> setNext(Link<E> nextval) {                  // O(cons.)
                    return next = nextval;                  // O(cons.)
          }
```

Therefore:
T(n) = O(1)

Asymptotic analysis List-based: insert()

```
public void insert(Key k, E e) {
		// ensure the key is unique
		KVpair<Key, E> temp = new KVpair<Key, E>(k, e);     //O(cons.)
		if ( find( k ) == null ) {
			pair.append(temp);
		}
	}

//method from LLDictionary called
public E find(Key k) {
		pair.moveToStart();                          //O(cons.) : moveToStart sets current at list head
		for(int i = 0; i < pair.length(); i++) {          //O(n)     : sequential search, until key match
			if(pair.getValue().key() == k) {                  //O(cons.)
				return pair.getValue().value();              //O(cons.)
			}
			pair.next();
		}
		return null;
	}


//method from LList called
public void append(E it) {
		tail = tail.setNext(new Link<E>(it, null));        //O(cons.)
		cnt++;
	}
```

O(cons.) terms ignored, since O(n) present (higher order)

Therefore,
T(n) = O(n)

Asymptotic analysis List-based: remove()

```
public E remove(Key k) {
                    E temp = find(k);
      if (temp != null)
         pair.remove();
      return temp;
          }


//method from LLDictionary called
public E find(Key k) {
                    pair.moveToStart();                         //O(cons.) : moveToStart sets current at list head
                    for(int i = 0; i < pair.length(); i++) {          //O(n)     : sequential search, until key match
                              if(pair.getValue().key() == k) {              //O(cons.)
                                        return pair.getValue().value();           //O(cons.)
                              }
                              pair.next();
                    }
                    return null;
          }


//method from LLDictionary called
public E remove() {
                    if (curr.next() == null) return null; // Nothing to remove    //O(cons.)
                    E it = curr.next().element(); // Remember value            //O(cons.)
                    if (tail == curr.next()) tail = curr; // Removed last        //O(cons.)
                    curr.setNext(curr.next().next()); // Remove from list        //O(cons.)
                    cnt--; // Decrement count
                    return it; // Return value
          }
```

Therefore,
T(n) = O(n)

Asymptotic analysis List-based: removeAny()

```
public E removeAny() {
        if ( size() != 0 ) {
                pair.moveToEnd();          //O(cons.) sets curr element to end
                pair.prev();
                KVpair<Key, E> e = pair.remove();
                return e.value();
        }
        else
                return null;
```

Asymptotic analysis List-based: find()

```
public E find(Key k) {
		pair.moveToStart();				//O(cons.) : moveToStart sets current at list head
		for(int i = 0; i < pair.length(); i++) {		//O(n)	: 'for' loop, sequential search, until key match found
			if(pair.getValue().key() == k) {		//O(cons.) : checks value, return key, compares, constant time cost
				return pair.getValue().value();		//O(cons.)
			}
			pair.next();
		}
		return null;
	}
```

Asymptotic analysis List-based: size()

```
public int size() {
                return pair.length();      //O(cons.) : returns a stored value; therefore no sequential
counting required
        }


//method from LLDictionary called

public int length() {
                return cnt; }
```