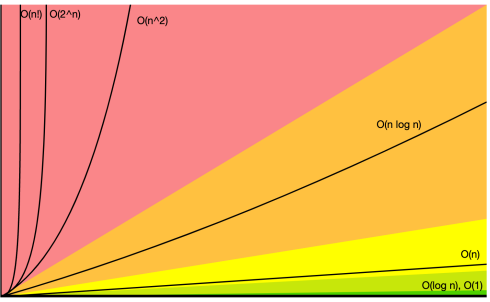
	Best	Average	Worst	Space	Stable
Selection	$O(n^2)$			$O(1)$	
Insertion	$O(n)$	$O(n^2)$		$O(1)$	✓
Bubble	$O(n^2)$			$O(1)$	✓
Merge	$O(n\log(n))$			$O(n)$	✓
Quick	$O(n\log(n))$		$O(n^2)$	$O(\log(n))$	
Bucket	$O(n)$	$O(n+k)$ (k: numbers of buckets)	$O(n^2)$	$O(n+k)$	✓
Counting	$O(n+k)$ (k: range of the input values, max-min)			$O(n+k)$	✓
Heap	$O(n\log(n))$			$O(1)$	

	Worst			Average			Space
	Insert	Delete	Search	Insert	Delete	Search	
Vector Ordered	$O(n)$		$O(\log(n))$	$O(n)$		$O(\log(n))$	$O(n)$
Vector Unordered	$O(1)$*	$O(n)$		$O(1)$*	$O(n)$		
Linked List	$O(1)$			$O(1)$			
Binary Search Tree	$O(n)$			$O(\log(n))$			
Balanced BST (RBT)	$O(\log(n))$						
Priority Queues	**Insert**	**RMH****	**Peek**	**Insert**	**RMH****	**Peek**	
Linked List	$O(n)$	$O(1)$		$O(n)$	$O(1)$		
Heap	$O(\log(n))$		$O(1)$	$O(1)$	$O(\log(n))$	$O(1)$	

*: Amortised - ie over a sequence of this operation. Resizing vector $O(n)$
**: Remove highest priority



I Linked List & Friend Classes

I.1 T/F

- Linked lists require at least two methods for insertion, one to cover addition at the front and another for addition at an arbitrary position.
- Linked lists have O(1) access to any node.
- Linked lists are not guaranteed to have their nodes located in adjacent blocks of memory.
- Linked lists have an aggregation relationship with nodes.

I.2 T/F

- A singly linked list has all the nodes linking in one direction.
- A singly linked list can have a tail as well as head. When it does, insertion and deletion at both either end becomes O(1)
- Insertion/deletion from the front of a singly linked list is O(1).
- A singly linked list cannot have a tail member variable that records where the last node is in memory, as it violates the definition of the singly linked list.

In a singly linked list all nodes point to the next node in the list. It is not a node's responsibility in this type of linked list to know who its predecessor is.

Insertion and deletion from the front of a singly linked list is O(1) because we do not need to use traversal. While we can give a tail member variable to a singly linked list, deletion from the tail is not O(1) because the last node does not know where its predecessor is located, and thus to update the second to last node, we need to traverse the entire list.

I.3 Code

```
class Point {
    int x, y;
    friend class Line;
public:
    Point(int x, int y) : x(x), y(y) {}
    void display() {
        cout << "Point (" << x << ", " << y << ")" << endl;
    }
};

class Line {
    Point p1;
    Point p2;
public:
    Line(Point p1, Point p2) : p1(p1), p2(p2) {}
    double length() {
        int dx = p1.x - p2.x;
        int dy = p1.y - p2.y;
        return sqrt(dx * dx + dy * dy);
    }
};

int main() {
    Point p1(1, 2);
    Point p2(4, 6);
    p1.display();
    p2.display();
    Line line(p1, p2);
    cout << "The length between the two points is: " << line.length() <<
endl;
    return 0;
}
```

- As Line is a friend of Point, Point can access any of the Line's private and protected member variables.
- Point should be listed as a friend class of Line because we may want to extend functionality to let Line invoke the display method.
- The Point's display method cannot be called from inside the Line class, because we did not use friend functions.
- As Line is a friend of Point, Line can access any of the Point's private and protected member variables.

Friend classes are specified in the class who is giving access of its private and protected member variables to its friend.

Mutual friendship can exist, but is not given automatically. As such Point cannot access Line's private/protected members. We do not need mutual friendship in this code, as Point's display method is public.