List of some common linear data structures along with their average time complexities for insert, remove, and search operations:

1. **Arrays:**
   - Insert: O(1)
   - Remove: O(1)
   - Search: O(n)

2. **Hash Tables:**
   - Insert: O(1) [On average, assuming a good hash function and proper handling of collisions]
   - Remove: O(1) [On average]
   - Search: O(1) [On average]

3. **Bit Array (Bitset):**
   - Insert: N/A [using masks]
   - Remove: N/A [using masks]
   - Search: O(1) [using masks]

4. **Stacks:**
   - Using Array:
     - Insert (Push): O(1)
     - Remove (Pop): O(1)
     - Search: O(n) [If you want to find an element other than the top]

   - Using Linked List:
     - Insert (Push): O(1)
     - Remove (Pop): O(1)
     - Search: O(n) [If you want to find an element other than the top]

5. **Vectors (Dynamic Arrays):**
   - Insert at End: O(1) [Amortized constant time]
   - Insert at Middle/Beginning: O(n)
   - Remove at End: O(1) [Amortized constant time]
   - Remove at Middle/Beginning: O(n)
   - Search: O(n)

6. **Circular Buffer (Circular Queue):**
   - Insert (Enqueue): O(1)
   - Remove (Dequeue): O(1)
   - Search: O(n) [In worst case]

7. **Linked Lists:**
   - 7.a Singly Linked List:
     - Insert / Remove at End: O(1) [if dummy node is available]
     - Insert / Remove at Beginning: O(n) [If no dummy, and no amortized update of head during inser & remove operations]
     - Search: O(n)

   - 7.b Doubly Linked List: [2 constant dummy nodes are available as strucs fields]
     - Insert at Beginning: O(1)
     - Insert at End: O(1)
     - Remove at Beginning: O(1)
     - Remove at End: O(1)
     - Search: O(n)

8. **Sorted List:**
   - Insert: O(log n) [On average, for self-balancing trees like AVL or Red-Black

Trees]
   - Remove: O(log n) [On average]
   - Search: O(log n) [On average]

9. **Queues:**
   - Using Array:
     - Insert (Enqueue): O(1)
     - Remove (Dequeue): O(1)
     - Search: O(n)

   - Using Linked List:
     - Insert (Enqueue): O(1)
     - Remove (Dequeue): O(1)
     - Search: O(n)

The best time complexity for a search operation is O(1), which means constant time complexity. This indicates that the time taken for the search operation does not depend on the size of the input data.

Data structures that achieve O(1) search time complexity are those that provide direct access to the desired element without requiring any significant computation or traversal. Some examples include:

1. **Hash Tables:** When using a well-designed hash function, hash tables provide constant-time average case lookup, insertion, and deletion.

2. **Direct Address Tables:** When the range of possible keys is limited, direct address tables can achieve O(1) search by directly indexing into an array.

3. **Bit Arrays (Bitsets):** Bit arrays allow for constant-time access to individual bits, which is equivalent to looking up the presence of an element.

It's important to note that while these data structures can provide constant-time search in the average case, real-world performance might vary based on factors such as hash collisions, memory layout, and the quality of the hash function. Additionally, these structures might have higher time complexities in worst-case scenarios or when certain conditions are not met.