[Closed book]
There are 5 questions leading up to 50 points.
Max allotted time: 1 hour

[You are welcome to write a pseudo-code or a valid C++ code for questions where you are asked to write a function. There will be no points taken off for missing out on the syntax of C++. The suggested function signature can be taken merely as a hint in case you would prefer to write a pseudo-code. State all assumptions around the pseudo-code you write.]

Problem 1 (1x10=10 points). Fill in the blanks in terms of the big-theta (Θ) notation to show the asymptotic running time complexity of each operation. Suppose the dynamic array is unsorted, and the singly linked list is unsorted and holds only the head pointer; n is the number of contained entries, and the stored data (of integer type) can be used as a key. State any assumptions you made.

<table>
<thead>
<tr>
<th>Operation</th>
<th>dynamic array (unsorted)</th>
<th>singly linked list (unsorted, only head pointer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A) find the i-th entry</td>
<td>Θ( 1 )</td>
<td>Θ( n )</td>
</tr>
<tr>
<td>(B) find a key</td>
<td>Θ( n )</td>
<td>Θ( n )</td>
</tr>
<tr>
<td>(C) insert/delete one entry in the beginning</td>
<td>Θ( n )</td>
<td>Θ( 1 )</td>
</tr>
<tr>
<td>(D) insert/delete one entry in the middle</td>
<td>Θ( n )</td>
<td>Θ( n )</td>
</tr>
<tr>
<td>(E) insert/delete one entry in the end</td>
<td>Θ( n )</td>
<td>Θ( n )</td>
</tr>
</tbody>
</table>

Assumptions:
Problem 2 (2+4+6=12 points). (a) What is the main advantage and the main disadvantage of a doubly-linked list compared to a singly-linked list? (b) Show a valid representation for a doubly-linked list. (c) Write down an `insertInTheEnd` function to insert an item at the end of the doubly-linked list (using the chosen representation in (b)).

Suggested function signature:

```cpp
DoublyLinkedList::insertInTheEnd(Item aKey)
```

(a) Advantage: Deleting an element in the end becomes \( \Theta(1) \) for a doubly-linked list; for singly-linked list, it would take \( \Theta(n) \) to first locate the previous entry (in the worst case), which is needed for deletion.

Disadvantage: Need extra space for every node to store the previous-node pointer.

(b) Class `DLLNode`:
```
struct DLLNode {
    Item mKey;
    DLLNode *mPrev;
    DLLNode *mNext;
};
```

Class `DLL`:
```
struct DLL {
    DLLNode *mHead;
    DLLNode *mTail;
    int mSize;
};
```

(c) ```cpp
void DLL::insertInTheEnd(T aKey) {
    // Calling constructor DLLNode (int, DLLNode*, DLLNode*)
    DLLNode *newNode = new DLLNode(aKey, nullptr, nullptr);
    if (getSize() == 0) {
        setHead(newNode);
        setTail(newNode);
    } else {
        setSize(1);
    }
}
```
// Linked list has at least 1 entry
DLLNode* tail = getTail();  // Get the tail pointer
    tail = setNext(newNode);  // Set the next of tail
newNode = setPrev(tail);  // Set the prev of newNode to tail
    setSize(++getSize());  // Increment size and update
    setTail(newNode);  // Set tail.
Problem 3 (3+3+4=10 points). Consider the following two algorithms, A1 and A2.

A1:

```java
for (int i = 0; i < n; i++) {
    for (int j = 0; j < 100; j++) {
        A[i] += A[j];
        ...
    }
}
```

Considering \( A[i] += A[j] \) as the main operation, write down the running time function \( T(n) \) for the above snippet of code. [Here, you are only expected to write down the raw function, not express in terms of the order notation.]

\[ T(n) = \ldots \]
A2:

for (int i = 0; i < n; i++) {
    // Perform binary search of key A[i] in array B.
    // Both A and B are of length n.
    // Performing the binary search should take O(lg n).
    int index = binarySearch(A[i], B, 0, n-1);
    ...
}

Considering binarySearch() as the main operation, write down the running time function $T(n)$ for the above for loop. [Here, you are only expected to write down the raw function, not express in terms of the order notation.]

$T(n) = \frac{n \ lg \ n}{\ \ }$

Which algorithm (A1 or A2) is going to run faster in practice? Why?

A2 is faster.

$\ lg \ n \ > \ 100 \ \ iff \ \ n \ > \ 2^{100}, \ which \ is \ an \ astronomical \ large \ number.$

In practice, $\ lg \ n \ should \ be \ \ \leq 20.$

$\therefore \ A2 \ will \ run \ faster \ in \ practice.$
Problem 4 (8+2=10 points). (a) Write a recursive function that determines whether a string of characters is a palindrome. (b) Also, clearly show the inputs with which you are going to invoke this function from main(). Note that your algorithm should not have an unnecessary space overhead.

A palindrome is a word or phrase that reads the same when you reverse it. Examples (shown along with the end-of-string character ‘\0’):
‘radar\0’, ‘naan\0’, etc. An empty string ‘\0’ can be also considered to be a palindrome.

Suggested function signature:

```c
bool isPalindrome(char* aStr, int aBegin, int aEnd)
```

```c
if (aBegin == aEnd)
    return true;
if ((aBegin + 1 == aEnd)
    && (aStr[aBegin] == aStr[aEnd]))
    return true;
if ((aStr[aBegin] == aStr[aEnd]))
    && isPalindrome(aStr, aBegin + 1, aEnd - 1))
    return true;
return false;
```
Calling the function

3

// input_str captures the input string
if (input_str[0] == '0') return true;
return isPalindrome(input_str, 0, strlen(input_str) - 2);

3
Problem 5 (4+(2+2)+2=10 points). (a) Draw a binary search tree that can result from the insertion of the following input sequence: 5,3,2,9,8,6,1,7. (b) Write down the result of performing a postorder and an inorder traversal on the resultant tree. (c) What is the worst-case time complexity of a “find” operation in a Binary search tree?

(a)

Post order: < left, right, root >

1 2 3 7 6 8 9 5

Inorder: 1 2 3 5 6 7 8 9

(c) $O(n)$ where n is the number of nodes in the BST.