Problem 1 (2x10=20 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 (6 points). Revisiting the operations shown in Problem 1, if the dynamic array and the singly linked list are already sorted, would you change any of your answers in Problem 1? Why or why not?

(B) find a key → $\Theta(lg n)$ for the dynamic array (using binary search).
Problem 3 (4+6+10=20 points). (a) What is the main advantage and the main disadvantage of a doubly-linked list compared to a singly-linked list? (b) Write a C++ class declaration for a doubly-linked list that supports the operations in question 1. (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
void DoublyLinkedList::insertInTheEnd(int aKey)
```

(a) **Advantage**: deleting an element becomes $\Theta(1)$ for a doubly-linked list; for a 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)

```cpp
class DLLNode {
    int mKey;
    DLLNode* mNext;
    DLLNode* mPrev;
public:
    DLLNode(int aKey) : mKey(aKey), mNext(NULL), mPrev(NULL) {}
    void setNext(DLLNode* aNext) { mNext = aNext; }
    void setPrev(DLLNode* aPrev) { mPrev = aPrev; }
};

class DLL {
    DLLNode* mHead;
    DLLNode* mTail;
};
```

(c)

```cpp
void DLL::insertInTheEnd(int aKey)
{
    // Calling the non-default constructor
    DLLNode* newNode = new DLLNode(aKey, NULL, NULL);
    if (mHead == NULL) {
        mHead = newNode; mTail = newNode; return;
    }
    mTail->setNext(newNode);// mTail->mNext = newNode
    newNode->setPrev(mTail); // newNode->mPrev = mTail
```
mTail = newNode; // update mTail
}

**Problem 4 (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.]

\[T(n) = 100n\]
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 \(O(\log n)\).
    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.]

\[ T(n) = n \log(n) \]

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

A2 will run faster in practice.

\(\log n > 100\) iff \(n > 2^{100}\), which is an astronomically large number.

In practice, \(\log n\) should be \(\leq 20\).
Problem 5 (10+2=12 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:

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

(a)  bool isPalindrome(char* aStr, int aBegin, int aEnd)
{
   // Base
   if (aBegin > aEnd)
      return true;
   // Recursion
   if (aStr[aBegin] == aStr[aEnd])
      isPalindrome(aStr, aBegin+1, aEnd-1)
      return true;
   return false;
}

(b)

// inputStr contains the input string
// strlen(inputStr) ignores the ‘\0’ in the end of the inputStr
if (isPalindrome(inputStr, 0, strlen(inputStr)-1))
   cout << “Palindrome” << endl;
else
   cout << “Not a palindrome” << endl;

Problem 6 (6+(5+5)+4=20 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)

(b) postorder: 1, 2, 3, 7, 6, 8, 9, 5
Inorder: 1, 2, 3, 5, 6, 7, 8, 9

(c) O(n)
Might end up with a BST that has the structure of a linked list
e.g. by inserting the items in order
Problem 7 (12 points). Write a function to compute the sum of all data in a binary tree. A binary tree has the pointer to the root node. A node holds an integer value as data and pointers to the left and right sub-trees.

```cpp
int computeSum(Node* aRoot)
{
    // Base
    if (aRoot == NULL) return 0;
    // Recursion
    return aRoot->key() + computeSum(aRoot->left()) + computeSum(aRoot->right());
}
```