Q1.

a) Time required:

Size (n)	Time required (s)
10,000	0.085
100,000	1.292
1,000,000	9.441

```
-------------- Part 1 --------------
Enter the size of IP Addresses to be read: 100000
The average time (1 trials) for this function is: 1.292 seconds.
The number of items (unique addresses) in setA: 94373
Duplicated items: 5627
The depth of setA: 38
```

Time required for size 100,000 for part 1a.

b)

```
        for (int i = 0; i < size; i++)
        {
            temp = "";
            getline(inFile, temp);
            setA.add(temp);
        }
```

Code for read in the data for the set

BigO for add() = BigO of recursiveAdd()

```
template <class T>
void MySet<T>::add(const T& item)
{
    root = recursiveAdd(root, item);
}
```

```
template <class T>
SetNode<T>* MySet<T>::recursiveAdd(SetNode<T>*curNode, const T& item)
{
    //exclude any duplicate items
    if(!contains(item))
    {
        if(curNode == nullptr)
            return new SetNode<T>(item);
        if (item < curNode->data)
            curNode->left = recursiveAdd(curNode->left, item);
        else
            curNode->right = recursiveAdd(curNode->right, item);
        return curNode;
    }
    else
        //if found duplicated item, just return curNode (the linkage sho
        return curNode;
}
```

Code for add() and recursiveAdd()

```
template <class T>
bool MySet<T>::contains(const T& item) const
{
    SetNode<T>*curNode = root;
    while (curNode != nullptr)
    {
        if (item == curNode->data)
            return true;
        else
            //search for left/right subtree
            if (item < curNode->data)
                curNode = curNode->left;
            else
                curNode = curNode->right;
    }
    return false;
}
```

Code for contains()

BigO for contains(): Average O(logn); (assume balanced BST)

BigO for recursiveAdd(): BigO of contains() + Line 337 – 348

= O(logn) + O(logn) = 2O(logn) = average O(logn); (assume Balanced BST)

BigO for add() = average O(logn); (assume Balanced BST)

Time required for reading 10,000,000 records into set:

= n * O(logn)

= 10,000,000 * log(10,000,000) * 9.441 / 1,000,000 * log(1,000,000)

= 70,000,000 * 9.441 / 6,000,000

= 110.15s

Q2.

a)

```
--------------- Part 1 ---------------
Enter the size of IP Addresses to be read: 1000000
The average time (1 trials) for this function is: 8.908 seconds.
The number of items (unique addresses) in setA: 451337
Duplicated items: 548663
The depth of setA: 45
--------------- Part 2 ---------------
The smallest item in setA: 000.000.055.001
The average time (1000000 trials) for this function is: 0.048 seconds.

Process returned 0 (0x0)   execution time : 12.206 s
```

Code for testing getSmallest for size 1,000,000 for 1,000,000 trials

Time required for getSmallest(): 0.048s.

b)

```
template <class T>
T MySet<T>::getSmallest() const
{
    return smallestValueFrom(root);
}

template <class T>
T MySet<T>::smallestValueFrom(SetNode<T>* curNode) const
{
    /*search for left subtree only, if no left subtree, return root)*/
    if(curNode->left == nullptr)
        return curNode->data;
    else
        return smallestValueFrom(curNode->left);
}
```

Code for getSmallest() and smallestValueFrom()

BigO for smallestValueFrom(): Average log(n); as determined by the depth of the tree

BigO for getSamllest(): Average log(n)

Time required for getSamllest() to read 10,000,000,000 items:

= 10,000,000,000 * 0.048 / 1,000,000

= 480s

Q6.

a) Time required:

Size (n)	Time required (s)
2,000	0.275
4,000	1.068
8,000	3.854

** when counting the time required for the function inserctionWith(), the copy constructor is also included in this exercise to return the intersection between 2 sets.

```
--------------- Part 1 ---------------
Enter the size of IP Addresses to be read: 4000
The average time (1 trials) for this function is: 0.027 seconds.
The number of items (unique addresses) in setA: 3982
Duplicated items: 18
The depth of setA: 24
--------------- Part 6 ---------------
The time (1 trial) for this function is: 1.068 seconds.
The number of items (unique addresses) in setAIB: 73
The smallest item in setAIB: 004.102.111.036

Process returned 0 (0x0)   execution time : 3.932 s
```

Time required for size 4,000 for part 6a.

b)

```
template <class T>
MySet<T> MySet<T>::intersectionWith(const MySet<T>& other) const
{
    // transfer the BST into vector
    vector<T> thisData = getRange(getSmallest(), getLargest());
    vector<T> otherData = other.getRange(other.getSmallest(), other.getLargest());
    vector<T> sortedData;

    MySet<T> setAIB;

    // case if the largest item from 1st vector is smaller than the smallest item in 2nd vector
    // indicating no intersection
    if(this->getLargest() < other.getSmallest() ||
       other.getLargest() < this->getSmallest())
       return setAIB;

    int thisCounter = 0, otherCounter = 0;

    //stop when either vector has visited all the data
    while(thisCounter < thisData.size() && otherCounter < otherData.size())
    {
        //case for matching the data (intersection)
        if(thisData[thisCounter] == otherData[otherCounter])
        {
            sortedData.push_back(thisData[thisCounter]);
            //setAIB.add(thisData[thisCounter]);
            thisCounter++;
            otherCounter++;
        }
        else
        {
            //1st vector data is smaller than 2nd vector data; forward 1st vector
            if(thisData[thisCounter] < otherData[otherCounter])
                thisCounter++;
            else
                otherCounter++;
        }
    }
    setAIB.root = bstFromVector(0, sortedData.size()-1, sortedData);
    return setAIB;
}
```

Code for part 6 (ver.A using sorted data to form BST) .

```
--------------- Part 1 ---------------
Enter the size of IP Addresses to be read: 20000
The average time (1 trials) for this function is: 0.196 seconds.
The number of items (unique addresses) in setA: 19494
Duplicated items: 506
The depth of setA: 31
--------------- Part 6 ---------------
The time (1 trial) for this function is: 23.776 seconds.
The number of items (unique addresses) in setAIB: 1478
The smallest item in setAIB: 000.120.211.040

Process returned 0 (0x0)   execution time : 29.876 s
Press any key to continue.
```

Time required to do 20,000 sample size for intersectionWith() for ver. A

```cpp
template <class T>
MySet<T> MySet<T>::intersectionWith(const MySet<T>& other) const
{
    // transfer the BST into vector
    vector<T> thisData = getRange(getSmallest(), getLargest());
    vector<T> otherData = other.getRange(other.getSmallest(), other.getLargest());
    //vector<T> sortedData;

    MySet<T> setAIB;

    // case if the largest item from 1st vector is smaller than the smallest item in 2nd vector
    // indicating no intersection
    if(this->getLargest() < other.getSmallest() ||
       other.getLargest() < this->getSmallest())
       return setAIB;

    int thisCounter = 0, otherCounter = 0;

    //stop when either vector has visited all the data
    while(thisCounter < thisData.size() && otherCounter < otherData.size())
    {
        //case for matching the data (intersection)
        if(thisData[thisCounter] == otherData[otherCounter])
        {
            //sortedData.push_back(thisData[thisCounter]);
            setAIB.add(thisData[thisCounter]);
            thisCounter++;
            otherCounter++;
        }
        else
        {
            //1st vector data is smaller than 2nd vector data; forward 1st vector
            if(thisData[thisCounter] < otherData[otherCounter])
                thisCounter++;
            else
                otherCounter++;
        }
    }
    //setAIB.root = bstFromVector(0, sortedData.size()-1, sortedData);
    return setAIB;
}
```

Code for part 6 (ver.B BigO: O(nlogn) using add() to form BST) .

```
--------------- Part 1 ---------------
Enter the size of IP Addresses to be read: 20000
The average time (1 trials) for this function is: 0.197 seconds.
The number of items (unique addresses) in setA: 19494
Duplicated items: 506
The depth of setA: 31
--------------- Part 6 ---------------
The time (1 trial) for this function is: 50.033 seconds.
The number of items (unique addresses) in setAIB: 1478
The smallest item in setAIB: 000.120.211.040

Process returned 0 (0x0)   execution time : 59.039 s
Press any key to continue.
```

Time required to do 20,000 sample size for intersectionWith() for ver. B

From the above, I have implemented both ver.A and ver.B for intersectionWith() for comparison. ver.A (using sorted data to form BST) runs faster than ver.B (using add() function to form BST) when the sample size is 20,000.

BigO for other helper functions:

1) **BigO for getSmallest() : O(logn) as determined above**
2) **BigO for getLargest() : O(logn); similar to getSmallest()**
3) **BigO for getRange() : O(n)**
    a. **BigO for recursivePrintRange() = 2 * O(n/2) + c = O(n); the worst case is to travel to the smallest to the largest item (go through all nodes between 2 sides of subtrees)**

```
template <class T>
vector<T> MySet<T>::getRange(const T& startValue, const T& endValue) const
{
    vector<T> IPsHolder;
    return recursivePrintRange(root, startValue, endValue, IPsHolder);
}

template <class T>
vector<T> MySet<T>::recursivePrintRange(SetNode<T>*curNode, const T& startValue, const T& endValue, vector<T> &IPsHolder) const
{

    //if the other set is an empty set, just return the empty vector
    if (curNode == nullptr)
        return IPsHolder;

    //go to the position where startValue < curNode->data
    if(startValue < curNode->data)
        recursivePrintRange(curNode->left, startValue, endValue, IPsHolder);

    // once found the position (should be equal to or larger than startValue)
    // stop when curNode->data > endValue
    // each recursive call will push the curNode->data to vector
    if (startValue <= curNode->data && curNode->data <= endValue)
        IPsHolder.push_back(curNode->data);

    //case for right subtree
    recursivePrintRange(curNode->right, startValue, endValue, IPsHolder);

    return IPsHolder;
}
```

4) **BigO for bstFromVector(): 2 * O(n/2) + c = O(n); as half of the data in vector is in left-subtree, the middle will be the root of the BST, and the half of the data is for right-subtree.**

```
template <class T>
//build bst from sorted data
SetNode<T>* MySet<T>::bstFromVector(int const begin, int const end, vector<T> const sortedData) const
{
    if(begin > end)
        return nullptr;

    int middle = (begin + end)/2;
    //the 1st call will be the root;
    SetNode<T>* newNode = new SetNode<T>(sortedData[middle]);

    //construct the bst through recursive call of bstFromVector
    //1st call will create the left & right child for the root, and continue to expand the tree until (begin > end)
    newNode->left = bstFromVector(begin, middle-1, sortedData);
    newNode->right = bstFromVector(middle+1, end, sortedData);

    //return the newNode(should be the root of the bst)
    return newNode;
}

#endif // MYSET_H
```

Therefore BigO for ver.A intersectionWith():

Line 408-409: 2 * O(n) = O(n)

Line 423: O(m+n); here m refers to no. of data in setA, n refers to no. of data in setB

Line 426-441: constants

Line 442: O(n)

Overall BigO for ver.A intersectionWith(): O(m+n)

(Optional) BigO for ver.B intersectionWith():

Line 408-409: 2 * O(n) = O(n)

Line 423-441: O((m+n)*logX); here m refers to no. of data in setA, n refers to no. of data in setB; X refers to the intersection data (can also be interpreted as the depth of the intersection BST)

**** constants are ignored**

Overall BigO for ver.B intersectionWith(): O((m+n)*logX)

It can be explained that why ver.A is faster than ver.B. when there are large intersection data (e.g. 1478 when the overall input size is 20,000). logX will there be significant comparing to ver.A to explain the difference between the time taken between 2 versions.