BINARY SEARCH FOREST ALGORITHM

By

SHANGSHAN ZHANG

Bachelor of Science
Beijing Forestry University
Beijing, China
1982

Master of Art
Indiana University
Bloomington, Indiana
1988

Doctor of Philosophy
North Carolina State University
Raleigh, North Carolina
1993

Master of Science
Oklahoma State University
Stillwater, Oklahoma
1998

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment
of the requirements of
the Degree of
MASTER OF SCIENCE
December, 1998

BINARY SEARCH FOREST ALGORITHM

Thesis approved:

[illegible]

Thesis Adviser

J Chandler

[illegible]

Wayne B. Powell

Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my sincere appreciation to Dr. George E. Hedrick, Chairman of the Advisory Committee, for his guidance, assistance and patience throughout my study at Oklahoma State University. I would also like to thank my committee members, Dr. John Chandler and Dr. Blayne E. Mayfield, for their helpful contributions and advice.

A deep felt thanks goes to my parents, Junzhow and Huilian, for their unending encouragement, and emotional support throughout the years.

Finally, to my wife, Jian, I wish to express my deepest appreciation for her love, extra patience and understanding. Completion of this thesis would not have been possible without her unfailing and indispensable support.

TABLE OF CONTENT

LIST OF TABLES AND FIGURES

1. INTRODUCTION

A binary search tree (BST) is a data structure with two important structural properties: 1) a node in the tree can have at most two children, 2) all keys in the tree are arranged in a total order manner; the values of all keys in the left subtree of every node X are smaller than the key value in X, and the values of all the keys in the right subtree are larger than the key value in X [16]. An important application of binary trees is their use in searching because the structure of binary search tree is well-suited for storing ordered set of elements. A binary search tree allows searching for an element in (log n) average-case time because the average depth of a binary search tree is O(log n). However, each operation could have linear time in the worst case because the depth can be as large as n-1.

To improve the worst-case behavior of binary search trees, different types of balanced search trees have been developed. Well-known balanced trees included AVL trees, red-black trees, and splay trees [5, 13, 15]. AVL trees and red-black trees enforce balance using balance-maintenance algorithms which allow insertion, deletion and searching to be performed in logarithmic worst-case time. However, the application of a balance-maintenance algorithm increases overhead cost because the balance must be checked and updated (if necessary) for every insert and delete operation. Splay trees do not require the maintenance of height or balance information, but achieves balance by self-adjusting. In a splay tree, a node is pushed to the root, after it is accessed, by a series of rotations. The side effect of the push-to-root operation is that the depth of most nodes on the access path is reduced to roughly half. Eventually, the depth of a splay tree becomes roughly log n after a series of the push-to-root operations. As a result, splay trees guarantee that any m consecutive operations take at most O(m log n) time. Unfortunately, this guarantee does not exclude the possibility that an single operation could take O(n) time.

Another way to obtain a balanced binary search tree is using binary representations of B-trees, such as symmetric binary B-tree [2], because B-trees are totally balanced trees with all leaves at the same depth [3]. A symmetric binary B-tree has several properties that makes it a good alternative to other binary search trees. It can be represented with only one extra bit per stored element for balance information, it has relatively simple updating algorithms and it can be maintained with a constant number of restructuring operations per update. However, the maximal height of an symmetric binary B-tree is 2*log n. Another drawback of symmetric binary B-trees is that elements are stored only in leaves, and all internal nodes store only indexes. If a tree is large, then there will be many internal nodes. It is expensive to maintain and store these internal nodes in terms of space utilization and time complexity.

In addition to these well-known balanced binary search trees, there are a number of other algorithms developed for balancing binary search trees; some are for general purposes while some are for special purposes such as secondary storage access [1, 4, 6, 7, 8, 9, 10, 11, 12, 14]. This study presents a new data structure called binary search forest which inherits all proprieties of binary search trees but has a better worst-case performance than binary search trees. In addition, this data structure can also perform sorting operations in O(n) time for sorting n elements. The implementation is also relatively simple. It does not require the maintenance of tree balance after each operation.

2. DATA STRUCTURE AND ALGORITHM

2.1 A Simple Idea (which does not always work):

For a given binary search tree of size n, the worst-case time is O(n). However, if the tree is broken into m subtrees and linked together using a root-list, then the worst-case time for each subtree in the forest is reduced to O(n/m) because each subtree has the size of only n/m. The reduction in subtree size, however, is associated with an extra cost of maintaining and searching the root-list. In order to minimize the cost of searching over the root-list, the arrangement of subtrees in the forest becomes the primary concern of the algorithm.

Because each subtree in the forest itself is a binary search tree, a given key could reside in any subtree in the forest. This makes the implementation difficult because if we do not know to which subtree a given key belongs, it is possible that we must search every subtree to access the key. Consequently the binary search forest is no better than a single unified binary search tree. One way to solve this problem is to impose an order among subtrees. That is, for every tree in the forest, the value of its maximum key is smaller than that of the minimum key of the tree on its right, and the value of its minimum key is larger than that of the maximum key of the tree on its left. For example (Figure 1), a given binary search tree with 14 keys with values 1 to 14 can be divided into 3 binary search trees. Thus, each subtree has at most 5 nodes. A root-list is used to link the three binary trees together to form a binary search forest. A total order is imposed among the three trees. Subtree 1 has nodes with key values less than 5, subtree 2 has nodes with key values less than 10, and subtree 3 has key values less than 15. A node with key value of 10 will be inserted into subtree 2. If a key to be inserted is out of the range of any existed tree, a new tree is created.

(a)

```
          4
         / \
        1   5
         \   \
          2   10
           \  / \
            3 7  14
             / \  /
            6  8 11
                \  \
                 9  13
                    /
                   12
```

(b)

```
 ------------------------------------------
 |   5    |    10     |    15    |
 ------------------------------------------
     /          |          /
    4          10         14
   / \         /          /
  1   5       7          11
   \         / \           \
    2       6   8          13
     \           \         /
      3           9       12
```

Figure 1: A regular binary search tree (a) and the corresponding forest implementation (b). The range of each tree is indicated by the value in each cell of the root-list. Both structures are implemented by consecutively inserting key 4, 5, 10, 7, 1, 2, 8, 14. 11, 6, 13, 3, 9, and 12.

Two steps are involved in accessing a key; first scan the root-list to find the appropriate subtree, then traverse the binary search tree to find the exact location of the key in the tree. Since the root-list can be implemented as a sorted array of size m (number of trees), a binary search algorithm can be used to locate a tree. Thus, the running time to find a tree is O(log m). Since the total n nodes are divided into m trees, each tree has at most n/m nodes. Therefore, the worst-case time of any individual subtree is O(n/m). Together, the worst-case running time to access a node in the binary search forest is

total time = (time for tree search) + (time for key search)

= O(log m) + O(n/m)

= O(n/m)

The forest implementation obviously has a better worst-case performance than a single regular binary search tree of the same size.

The average time for this data structure is O(log n) if each tree in the forest is full. Since each tree in the forest itself is a binary search tree, the average time to access a key in a tree will be log (n/m). Assume the average time for tree-search in the root-list is also log m, the same as the worst-case time, then the average time to search all the trees necessary is the sum of the average time of the tree-search step and the tree-traversal step.

average time = (log m) + (log n/m)

= log(m * n/m)

= log n

Thus, the average time of the binary search forest of size n is approximately same as the average time of a single binary search tree of the same size. Both algorithms have an average running time of O(log n). However, the average time for this implementation is determined under the assumption that every tree in the forest is full so that the size of each tree is exact n/m. This situation is unlikely to occur in practice. If a tree is not full, then the number of nodes in the tree will be less than n/m, and accessing a key in the tree will cost less than in a full tree.

The structure of the binary search forest is quite simple. Each subtree in the forest is a regular binary search but smaller in its size. Same to the node structure in a binary search tree, each unit of the root-list has two pointers; one points to the root of a subtree, one points to next unit in the list. Each unit also contains a key indicating the upper bound of its tree.

As mentioned in the beginning of this section, this simple implementation may not always work. For example, when a tree grows larger than n/m due to insert operation, the largest element in that tree must be reinserted into the tree on the right. This could propagate if that tree is also full. Consider the situation where key 3.5 is inserted into the forest present in Figure 1b. Since tree 1 is already full, the node with key 5 must be reinserted into tree 2. Since tree 2 is also full, then key 10 must be moved to tree 3, and so on until a non-full tree is found. To avoid this propagation, the following improved implementation was developed.

2.2 The improved implementation:

In this improved implementation, a tree-splitting method is used to avoid the propagation problem. This implementation requires pre-determination of the size of the trees in the forest. When a key is inserted into a tree which is already full, instead of propagation, we simple split the tree into two trees. In order to effect this implementation, the structure of the root-list needs to be modified slightly. First, instead of using an index to indicate the upper bound of each tree, the actual maximum-element of each tree is kept in the root-list as shown in Figure 2. If a tree has only one node, the node will be kept in root-list. Second, the root-list will be implemented as a linked list so that the split and merge operation can be performed in constant time.

```
 -------------------------------------------
 |    5     |     10     |     14     |
 -------------------------------------------
      /           /             /
     4           7             11
    /           / \              \
   1           6   8              13
    \               \            /
     2               9          12
      \
       3
```

Figure 2: The modified representation of Figure 1b, with the largest element of each tree actually residing in the root-list. Notice that the number of 15 in the root-list is changed to 14 which is the largest element in tree 3.

2.2.1 Insert and build-tree

If the forest has only one tree, we insert each key, one by one, into the tree until it is full. The key to be inserted is first compared with the root. If the key is smaller than the key value of the root, then directly insert it into the tree, same as the standard insert operation of a binary search tree. If the key is larger than the root, swap it with the root and insert the old root into the tree so that tree root always contains the maximum key of the tree. Once the tree is full, insertion of next key requires tree splitting. After a tree is split, more nodes can be inserted into the new trees until one of the trees is full again. However, if there are more than one trees in the forest, to insert a key, we must first search the tree to which the key belongs. To find the tree, the key to be inserted is compared with the keys of roots until the first root with key value larger than the insert key is found. Then the key is inserted into the tree with that root. If the key to be inserted is larger than any key in the forest, it will be inserted into the last tree in the forest.

Trees built by this method are slightly different from the standard binary search trees (Figure 3). The root of each tree in the forest has only one child. On other words, each root has only a left subtree because the root always contains the maximum key of that tree. The actual binary search tree structure starts from the first descendant of the tree. For convenience, we call the root of the first subtree of the root of a tree the "second root". In Figure 3, the nodes with key values 4, 8, and 11 are the second roots of tree 1, 2, and 3, respectively.

```
------------------------------------------

|    5    |    10    |    14    |
------------------------------------------
     /          /           /
    4          8           11
   /          / \            \
  1          7   9           13
   \        /               /
    2     6               12
     \
      3
```

Figure 3: A forest representation of binary search tree built by the improved implementation using the same keys used in the previous example. Maximum number of nodes each tree can keep is assigned to be 5 (k=5) in this example.

2.2.2 Split

Tree splitting takes place at the second root of the tree. The second root and its left subtree are split from the old tree to form a new tree. The second root then becomes the root of the new tree and is inserted into the root-list on the left next to the old tree. By implementing the root-list as a linked list, it is a simple job to insert a root into the linked list. The right subtree of the old second root of the old tree is linked to the root of the old tree (Figure 4). In this way, the old tree is divided into two trees at about the mid point of the tree if it is balanced. This is the best way to generate two trees with about equal number of nodes. However, when the tree is highly unbalanced, such as when it has an almost linear structure in the extreme, it is possible to generate two trees with one tree having only one node and the other having all rest of the nodes.

Figure 4 exhibits an example of this process. In this example, 8.5 is the key to be inserted into the forest in Figure 3. By searching the root-list we find that 8.5 should be inserted into tree 2 since 8.5 is smaller than the key value (key = 10) of the root of tree 2. After 8.5 is inserted into tree 2, the number of nodes in the tree is 6 which is larger than tree size (k = 5). Then tree 2 is split into two trees; the one rooted at key 8 and the one rooted at key 10. All keys with value smaller than 8 are moved to the tree rooted at 8 and the rest are kept in the tree rooted at 10. Both new trees maintain their order property. The process operates as: first insert the key, then check whether the tree is overfull, if it is, then split the tree.

The split operation occurs in constant time; we assume, without loss of generality, that it takes 1 unit of time to create a new unit in the root-list, 1 unit of time to insert the new tree, and 1 unit of time to relink the old tree. Therefore, the order of time complexity is O(1).

```
        ---------------------------      -----------------------------
        |  5  |  10  |  14  |            |  5  |  8  | 10 | 14 |
        ---------------------------      -----------------------------
+8.5    /       /       /       split    /      /      /      /
---->   4       8       11      ---->    4      7      9      11
       /       / \       \              /      /      /        \
      1       7   9       13           1      6      8.5       13
       \     /   /        /             \                      /
        2   6   8.5     12               2                   12
         \                                \
          3                                3
```

Figure 4: The insertion of 8.5 into the binary search forest in Figure 3. The original tree rooted at key 10 is overfull (number of nodes > k=5) after the insertion, and then it is split into two trees; the one rooted at key 8 and the other rooted at key 10.

2.2.3 Delete

Similar to the insert operation, to delete a key, the first thing is to find the tree to which the key belongs. Once the tree is found, the rest of the work is same to the standard delete operation of a binary search tree except the deletion of the root. Since the root of a tree in the forest contains the maximum key, after the root is deleted, we need to search the new maximum key of the tree and insert it into the root-list as the new root of the tree.

2.2.4 Merge

Tree merge is needed if the total number of nodes of two adjacent trees is less than or equal to the fixed tree size ($k \geq n_i + n_{i+1}$). This situation may occur after split or delete operations. After tree splitting, new trees are smaller. If the adjacent neighbor trees are also small enough, two adjacent trees can be merged into a larger tree as long as the resulting size is not larger than the maximum tree size k.

When two trees are merged, the root of the tree on the right (the tree with larger key values) is served as the root of the merged tree. The root of the tree on the left will become the second root of the merged tree, and the second root of the old tree on the right will become the right subtree of the new second root of the merged tree (Figure 5). For example, after key 8.5 is deleted from the third tree on Figure 3, tree 2 and tree 3 are needed to be merged together because the total number of the two trees are 5 which is equal to the tree size. The root (key = 10) of tree 3 becomes the root of the merged tree,

```
 ----------------------      ----------------------      --------------------
 | 5 |  8 | 10 | 14 |        | 5 | 8 | 10 | 14 |         | 5  | 10 | 14 |
 ----------------------      ----------------------      --------------------
   /   /   /   /     -8.5   /   /   /   /     merge  /     /     /
  4    7    9  11    ----> 4    7   9  11     -----> 4     8     11
 /    /   /     \           /  /         \           /    / \     \
 1    6  8.5     13        1   6         13         1     7  9   13
 \              /          \             /          \  /         /
  2            12           2           12           2 6        12
   \                         \                        \
    3                         3                        3
```

Figure 5: Delete 8.5 from the forest. After the delete operation, tree 2 and tree 3 need to be merged into one tree because the total number of nodes of the two trees are 5 which equals the required tree size.

the root (key = 8) of tree 2 becomes the second root of the merged tree, and the second root (key = 9) of tree 3 becomes the right subtree of the new second root of the merged tree. After merging, we then delete the node from the root-list, which is used to hold tree 2. This process is exactly the reverse of the split process. Therefore, the time complexity for a tree merger operation is also O(1).

2.3 Time complexity

Based on the analysis in previous section, the worst-case time for the binary search forest should be O(n/m) if the root-list is implemented as an array. However, for the improved implementation, the root-list is implemented as a linked list. The worst-case time to search over the linked listed be O(m). Thus, the worst-case time to access a key in the forest is:

worst-case time = maximum time for tree search + maximum time for key search

= O(m) + O(n/m)

= O(O(m), O(n/m))

In practice, it is more likely that the number of trees in the forest will be smaller than the size of the tree. Based on above equal, the worst-case time of this implementation will take O(n/m) in most cases. However, because each tree in the forest is designed to have a maximum size, conducting an operation in a tree should take only a constant time no matter how larger the tree size is. Then the worst-case time of this data structure is O(m) +O(1) = O(m). Obviously, this worst-case time complexity is better than O(n/m) in most cases, and is much better than the worst-case time O(n) of regular binary search trees. It should mentioned, however, that if the tree size is vary large, then there will be a very large constant in the time complexity expression. Thus, the actual running time of the implementation may not have as good practical performance as it seems to have theoretically. For this reason, we will take O(n/m), instead of O(m), as the worst-case time for this data structure in general.

2.4 Amortized analysis

Amortized time is the worst-case running time for any sufficiently long sequence of z operations. It contrasts with the worst-cast analysis which is given for any single operation [15]. In amortized analysis, the state of the data structure at any time is given by a function known as the potential. When operations take less time than the time allocated to them, the unused time is saved in the form of a higher potential. When operations occur that take more time than the allocated, then the excess cost is covered by the saved time. Once a potential function is chosen, the amortized time is determined as follow:

amortized time (a) = actual time (t) + potential (Φ)

If the final potential, after a sequence of operations, is at least as large as the initial potential which is commonly chosen to be 0, then the amortized time is the upper bound on the actual time used during the execution of the sequence.

Amortized analysis normally is applied for data structures whose time complexities are difficult to be determined by traditional methods. As discussed above, the worst-case time for this data structure could be as good as O(m), considering the maximum tree size. In this case, the time complexity of the forest implementation can be determined easily. It is simply the time for searching for a tree over the root-list. Thus, no amortized analysis is necessary. However, assume the time complexity of O(m) for this data structure is not realistic in practice because the value of the tree size could be a very large constant. In this case, we must account for the actual cost of all the processes involved in an operation to determine its upper bound. Amortized analysis may be appropriate to determine the time bound for this data structure.

The extra cost of the alternative implementation is the split process and the merge process after an insert operation and a delete operation, respectively. As stated above, the split process requires 3 units of time; 1 for creating a new unit in the root-list, 1 for linking the new tree to the new root, and 1 for relinking the old tree. The merge process is the reversal of the split process, so it also takes 3 time units. The result of a split operation or a merge operation is a net change in the number of trees in the forest. This allows the use of the number of trees as the basis for the potential function. The potential function is defined as follow:

potential = 3 time units * number of trees = 3 * m

The potential before operation is: $\Phi_b = 3 * m_b$

The potential after operation is: $\Phi_a = 3 * m_a$

The change in potential after a operation is: $\Delta\Phi = \Phi_b - \Phi_a = 3(m_a - m_b)$

The initial potential is 0 since there is no tree present at the beginning. The net change in potential is 0 after an insert or delete operation without a split or merge, because the number of trees is not changed. When splitting occurs, one more tree is added into the forest, resulting in a net increase in potential by 3 time units. When merging occurs, one tree is removed from the forest, resulting in a net decrease in potential by 3 time units.

The decrease in potential by merging is covered by the energy saved during splitting. Because the number of trees can never be negative, the amortized time bound holds. Because each operation takes O(log n) time on average, then the amortized time is

amortized time (a) = actual time (t) + net potential change ($\Delta\ \Phi$)

$$= (\log n) + 3(m_a - m_b)$$

After z operations, the total potential will be $3(m_z - m_0) = 3m$, and the actual time for z operations will be (z log n). Together, the amortized time for z operations will be

$$\sum_{i=0}^{z} a = (z \log n) + 3m$$

Thus, this data structure takes O(z log n) time for z successive operations. By comparing this result with the worst-case analysis which takes O(n/m) time for each operation and O(z n/m) time for z successive operations, the amortized time bound is tighter than the worst-case bound. However, the amortized bound is not as strong as the worst-case bounds, because there is no guarantee for any single operation. The amortized bound is stronger than the corresponding average-case bound, because even the average time is O(log n) per operation, it is still possible for a sequence of z operations to take z times of the worst-case time.

3. DISCUSSION

The selection of subtree size (k) or the number of subtrees (m = n/k) is an important factor in the implementation. If m is too large, then the search of the root-list becomes expensive. If m is too small, then each subtree has a relatively large size, implying that a search within a subtree can be quite costly. In one extreme, if m = n, then every single node becomes a subtree and what we get is a sorted linked list. In other extreme, if m = 1, then the forest is a one-tree forest. Thus, the forest implementation of the binary search tree is just a redundant work. There is no unique requirement for the selection of tree size, which depends on the actual situations. One way is to make m a constant so that the tree search process in the root-array takes only a constant time O(1). The problem associated with this method is that trees in the forest could be too small or too large, depending on the total number of keys. Another way is to make m a variable, depending on the predetermined maximum tree size k. If a tree is full, then a new tree is created. This is the method we used in this study. The problem with the method is the number of trees may become too large as the number of keys increased. Theoretically, this forest implementation should outperform the regular binary search trees, since we can implement either the number of the trees (m) or the size of the trees (k) as a constant number and take advantage of that fact.

The extra overhead cost of this implementation occurs in the split and merge operations. However, both the split and merge operations are rather simple and fast, since they take only a constant time. Neither rebalancing nor sorting is needed for these two operations. The only work needed for the split and merge process is to switch the pointers of the roots and the second roots of the two trees involved, but leave rest of the nodes of the two trees untouched. In practice, the size of each tree is expected to be more than just a few nodes. If the designated tree size is not very small, the number of tree-split operations should small.

This algorithm requires no an extra space to implement the root-list because each unit of the root-list also holds a actual element of a tree. Therefore, it will not affect the space complexity relative to the regular binary search trees.

The structure of this algorithm not only has the binary search tree order but also has, in part, the heap order because each tree has its maximum key located in the root. Thus, we can perform the delete_max operation on this data structure for sorting purpose. Since the keys with the largest values in the forest are all located in the most right tree (last tree) and the maximum key of the tree is located in the root-list, so it takes only a constant time to find and delete the maximum key of the forest. However, after deleting the root of the rightmost tree, we have to traverse the tree to find next largest key of the tree to replace the deleted root. This process takes worst-case time of O(k), which is O(1) because k is a constant. As a result, the entire delete_max operation takes O(1) time. If we consider tree size k to be (n/m) instead of a constant, then the delete_max operation takes O(n/m) worst-case time and O(log(n/m)) average-case time. Because this data structure has both search tree propriety and heap propriety, it may have a great potential for applications which require both efficient binary-search and heap-sort.

Another advantage of this data structure is its structural flexibility by which all trees in the forest can be easily merged into one single binary search tree or be extended to an arbitrary number of trees. Therefore, the size of trees and the number of trees in the forest can be easily adjusted if needed. Because this data structure employs the split and merge operations, we can perform the sorting operation by continuously splitting trees until only one node left for each tree. This sorting process is very efficient and it takes only O(n) time for sorting n nodes and O(1) average time for each individual operation. In the same way, a sorted list of size n can be merged into a single tree in O(n) time. This is specially important for building a forest when the input data are presorted. For a regular binary search tree, to build a tree using presorted data will take a worst-case time for each step and result in a linear structure. But for the binary search forest, to build a forest using the same presorted data takes almost optimal time and results in a optimal structure. The detail of the SplitSort and MergeBuild processes will be discussed in next section.

4. IMPLEMENTATION AND TESTING

This section will first demonstrate the basic operations of the binary search forest, and then compare its performance with that of the regular binary search tree in both worst case and average case. The running time is determined by the number of loops and recursions instead of the actual running time. In data structure theory, time complex of a search tree is usually determined by the depth of the tree, and the constant factor of the time complexity is ignored. Following are the results of the actual operations of the implementations. The number in the parenthesis associated with each node indicates the number of nodes in that tree. At the end of each operation, the time count is displayed. For the convenience of presentation, we limit the total number of nodes to 15 in most cases.

4.1 Basic operations of binary search forest

The basic operations include Create, Find, Insert, Delete, Merge, Split, ChangeTreeSize, SplitSort, DelMaxSort, MergeBuild, FindMin, FindMax, CountNodes, PrintRoots, and PrintForest. This section only demonstrates those key operations. The other operations will be demonstrated in next section.

4.1.1 Create (a forest of size 10 with tree size of 5 by Insert operations)

```
Enter tree size: 5

Enter the element to be inserted: 15
T1: 15(1)
Time count: 1
```

```
Enter the element to be inserted: 6
T1: 15(2)
         6(1)
Time count: 2

Enter the element to be inserted: 4
T1: 15(3)
         6(2)
              4(1)
Time count: 3

Enter the element to be inserted: 10
T1: 15(4)
              10(1)
         6(3)
              4(1)
Time count: 3

Enter the element to be inserted: 2
T1: 15(5)
              10(1)
         6(4)
              4(2)
                   2(1)
Time count: 4

Enter the element to be inserted: 12
T2: 15(3)
              12(1)
         10(2)
T1: 6(3)
         4(2)
              2(1)
Time count: 5

Enter the element to be inserted: 13
T2: 15(4)
                   13(1)
              12(2)
         10(3)

T1: 6(3)
         4(2)
              2(1)
Time count: 5
```

```
Enter the element to be inserted: 11
T2: 15(5)
                   13(1)
              12(3)
                   11(1)
         10(4)

T1: 6(3)
         4(2)
              2(1)
Time count: 5

Enter the element to be inserted: 3
T2: 15(5)
                   13(1)
              12(3)
                   11(1)
         10(4)

T1: 6(4)
         4(3)
                   3(1)
              2(2)
Time count: 4

Enter the element to be inserted: 5
T2: 15(5)
                   13(1)
              12(3)
                   11(1)
         10(4)

T1: 6(5)
              5(1)
         4(4)
                   3(1)
              2(2)
Time count: 3
```

The maximum size of each tree is 5. Tree 1 (T1) is full after inserting 2 so that the insertion of next key (12) resulted in the split of tree 1 to two trees. After the splitting, each tree is about half full, more nodes can be inserted into the trees until they are full again. Then tree splitting will take place again. These processes can go on and go on. Next few operations will further demonstrate the splitting and merging actions.

4.1.2 Insert, Delete, Split and Merge

```
Enter the element to be inserted: 9
T3: 15(4)
                   13(1)
         12(3)
                   11(1)

T2: 10(2)
         9(1)

T1: 6(5)
                   5(1)
         4(4)
                             3(1)
                   2(2)
Time count: 5

Enter the element to be deleted: 9
T2: 15(5)
                             13(1)
                   12(3)
                             11(1)
         10(4)

T1: 6(5)
                   5(1)
         4(4)
                             3(1)
                   2(2)
Time count: 4
```

After inserting 9, tree 2 was split into two trees. After deleting 9, the total number of nodes in tree 2 and tree 3 are less than the maximum tree size and the two trees are merged into one tree.

If the insertion of 9 is followed by the insertion of 1, the key 5 and 6 that are split from tree 1 will be merged into tree 2 which already has key 9 and 10. The number of trees in the forest is still 3 as showed by the following figures.

```
Enter the element to be inserted: 9

T3: 15(4)
                   13(1)
         12(3)
                   11(1)
```

```
T2: 10(2)
         9(1)

T1: 6(5)
              5(1)
         4(4)
                   3(1)
              2(2)
Time count: 5

Enter the element to be inserted: 1

T3: 15(4)
              13(1)
         12(3)
              11(1)

T2: 10(4)
              9(1)
         6(3)
              5(1)

T1: 4(4)
              3(1)
         2(3)
              1(1)
Time count: 6
```

4.1.3 ChangeTreeSize, SplitSort, MergeBuild, and DelMaxSort

```
Change tree size (by Merge operations for above forest)
Enter tree size: 12
T1: 15(12)
                   13(1)
              12(3)
                   11(1)
         10(11)
                        8(1)
                   6(3)
                        5(1)
              4(7)
                        3(1)
                   2(3)
                        1(1)
Time count: 2

SplitSort (by Split operations for above tree)
Enter tree size: 1
T12: 15(1)
T11: 13(1)
T10: 12(1)
```

```
T9: 11(1)
T8: 10(1)
T7: 8(1)
T6: 6(1)
T5: 5(1)
T4: 4(1)
T3: 3(1)
T2: 2(1)
T1: 1(1)
Time count: 11

MergeBuild  (by Merge operations for above sorted data set)
Enter tree size: 15
T1: 15(12)
                    13(1)
               12(3)
                    11(1)
          10(11)
                         8(1)
                    6(3)
                         5(1)
               4(7)
                         3(1)
                    2(3)
                         1(1)
Time count: 11

DelMaxSort (by DelMax operations for above tree)
15 13 12 11 10 8 6 5 4 3 2 1
Time count: 33
```

One of the advantage of this data structure is its structure flexibility by which the number of trees or the size of each tree can be easily modified with little cost. The modifications are achieved by split and merge operations. This approach provides a very efficient way to perform the sorting operation and the build-forest operation from the presorted or ordered data set (To build a BST from a presorted data set by the regular approach is the most costly operation. We will discuss this in detail later). From above operations we can find that for the same set of data in the same format the DelMaxSort (sorting by DelMax operations) takes 33 time units while the SplitSort(sorting by Split operations) takes only 11 time units. If the forest has only one tree, the SplitSort process takes only n-1 time for sorting n nodes, suggesting that the time complexity for each individual operation takes only O(1) time. If the forest has more than one trees, the SplitSort process takes even less time because some split operations have already done. The advantage of the SplitSort is that it does not require tree traverse to find the max-key as the DelMaxSort does.

The MergeBuild process using the sorted data set is the exact reverse of the SplitSort operation. It takes also only n-1 units of time. Next few figures demonstrate the step by step MergeBuild process by successively applying the Merge operation. The basic approach is to consider each individual node as a single-node tree and then merge them pair by pair, following by merging two small trees to a bigger tree. This process go on and go on until only one tree left. In each step the size of tree doubles and the number of trees in the forest is reduced to half. The time complexity for this whole MergeBuild process is O(n) which is much faster than the insert approach.

The step by step process of MergeBuild using above sorted data:

```
Enter tree size: 2
T6: 15(2)
          13(1)
T5: 12(2)
          11(1)
T4: 10(2)
          8(1)
T3: 6(2)
          5(1)
T2: 4(2)
          3(1)
T1: 2(2)
          1(1)
Time count: 6

Enter tree size: 4
T3: 15(4)
               13(1)
          12(3)
               11(1)

T2: 10(4)
               8(1)
          6(3)
               5(1)

T1: 4(4)
               3(1)
          2(3)
               1(1)
Time count: 3

Enter tree size: 8
T2: 15(4)
               13(1)
          12(3)
               11(1)

T1: 10(8)
                    8(1)
               6(3)
                    5(1)
          4(7)
                    3(1)
               2(3)
                    1(1)
Time count: 1

Enter tree size: 12
T1: 15(12)
                    13(1)
               12(3)
                    11(1)
          10(11)
                         8(1)
                    6(3)
                         5(1)
               4(7)
                         3(1)
                    2(3)
                         1(1)
Time count: 1

Total time = 6 + 3 + 1 + 1 = 11
```

As above operations show that the MergeBuild process can stop at any step to build a binary forest instead of a single binary search tree, depending on the tree size one wants. The earlier the process stops, the less the time used. Besides, the forest built by the MergeBuild method is perfectly balanced, as demonstrated by the following forest built by the MergeBuild method from 32 presorted data.

```
Enter tree size: 8

T4: 32(8)
                        31(1)
                30(3)
                        29(1)
        28(7)
                        27(1)
                26(3)
                        25(1)


T3: 24(8)
                        23(1)
                22(3)
                        21(1)
        20(7)
                        19(1)
                18(3)
                        17(1)


T2: 16(8)
                        15(1)
                14(3)
                        13(1)
        12(7)
                        11(1)
                10(3)
                        9(1)


T1: 8(8)
                        7(1)
                6(3)
                        5(1)
        4(7)
                        3(1)
                2(3)
                        1(1)

Time count: 28
```

4.2 Comparison between regular BST and binary search forest

4.2.1 Worst-case performance

4.2.1.1. Binary search forest

Create (by successively inserting 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

```
Enter tree size: 5
Enter the total number of nodes: 10

T2: 10(5)
         9(4)
              8(3)
                   7(2)
                        6(1)
T1: 5(5)
         4(4)
              3(3)
                   2(2)
                        1(1)
Time count: 11

Find (find deepest node)
Enter the element to be found: 6
10 9 8 7 6 (found)
Time count: 6

FindMax
The max-key is: 10
Time count: 1


FindMin
the min-key is: 1
Time count: 5

DelMaxSort
10 9 8 7 6 5 4 3 2 1
Time count: 20

SplitSort
Enter tree size: 1
T10: 10(1)
T9: 9(1)
T8: 8(1)
T7: 7(1)
T6: 6(1)
T5: 5(1)
T4: 4(1)
T3: 3(1)
T2: 2(1)
T1: 1(1)
Time count: 8

MergeBuild (tree size = 5)

T3: 10(2)
        9(1)

T2: 8(4)
```

```
                   7(1)
         6(3)
                   5(1)

T1: 4(4)
                   3(1)
         2(3)
                   1(1)
Time count: 7
```

In the MergeBulid operation, the process stopped when the forest still has 3 trees because further merging of any two trees in the forest will cause the number of nodes in the resulted tree excesses the pre-set tree size 5. In this example, the total number of nodes is 10 and maximum size of each tree is set to be 5. This does not necessarily mean that there will be only two full trees. In practice, some trees will be completely full and some only half full. Thus, further insertion of new node into the forest will not necessarily result in a tree splitting. Tree merging will take place only when the total number of nodes of two adjacent trees are smaller than the maximum tree size.

4.2.1.2 Binary search tree

```
Create (by successively inserting 1, 2, 3, 4, 5, 6, 7, 8, 9,
10)
Enter tree size: 10
                                                      10(1)
                                                 9(2)
                                            8(3)
                                       7(4)
                                  6(5)
                             5(6)
                        4(7)
                   3(8)
              2(9)
         1(10)
Time count: 55

Find (find the deepest node)
Enter the element to be found: 10
1 2 3 4 5 6 7 8 9 10 (found)
Time count: 10
```

```
FindMax
The max-key is 10
Time count: 10

FindMin
The min-key is 1
Time count: 1

DelMaxSort
10 9 8 7 6 5 4 3 2 1
Time count: 55
```

Table 1: The comparison of the worst-case performance between regular BST and binary search forest

	Time	
Operation	BST	Forest
Create	55	11
Find deepest node	10	6
FindMax	10	1
FindMin	1	5
DelMaxSort	55	20
SplitSort	--	8
MergeBuild	--	7

As Table 1 shows that the forest implementation has significant advantages than the regular BST implementation in worst-case performance. The worst-case of a binary search tree is that its structure moves towards to a linear structure because binary search trees has no control on its balance. This occurs when presorted data are inserted successively into an empty binary tree. In this example, the Create operation of binary search tree takes 55 units of time but the Create operation of the forest implementation takes only 11 units of time. The forest implementation keeps max-keys in the roots. If the persorted data set is in ascending order, each new key will be inserted into the root so that there will be no need for tree travel to find the location. Therefore, the time complexity is O(n) for the whole create process and O(1) for each individual insert operation. This time complexity will not happen if the input data set is sorted in descending order. However, because the forest implementation employs the split and merge operations, a forest can be built using MergeBuild method for any presorted data set. In this example, the MergeBuild process takes only 7 time units, which is about 8-times faster than the create process of a regular binary search tree. Furthermore, as mentioned above, a forest built by the MergeBuild method has a completely balanced structure. Thus, for a same set of sorted data, a BST built by the regular insert approach will have the worst-case structure while a BST or a forest built by the merge approach will have the optimal structure.

Because each tree in the forest is smaller the a regular BST of the same size, to find the deepest node will be faster in the forest. For the same reason, the DelMaxSort operation which performs sorting by continuously deleting max-key is less expansive for the forest (20 time units) than for the single BST (50 time units). In addition, the forest implementation can use split operation to perform sorting, the whole sorting process for the forest by the SplitSort approach takes only 7 time units.

4.2.2 Average-case performance

4.2.2.1 Binary search forest

```
Create (by successively inserting randomly generated data)
Enter tree size: 5
Enter the total number of nodes: 10
T3: 10(1)

T2: 9(5)
              8(2)
                   7(1)
         6(4)
```

```
                5(1)

T1: 4(4)
                3(1)
           2(3)
                1(1)
Time_count: 34

Find (find the deepest node)
Enter the element to be found: 7
9 6 8 7 (found)
Time_count: 5

FindMax
The max-key is: 10
Time_count: 1

FindMin
the min-key is: 1
Time_count: 3

SplitSort
Enter tree size: 1
T10: 10(1)
T9: 9(1)
T8: 8(1)
T7: 7(1)
T6: 6(1)
T5: 5(1)
T4: 4(1)
T3: 3(1)
T2: 2(1)
T1: 1(1)
Time count: 7

DelMaxSort
10 9 8 7 6 5 4 3 2 1
Time count: 23
```

4.2.2.2 Binary search tree

```
Create (by randomly generated data)
Enter tree size: 10
          10(1)
     9(10)
                    8(2)
                         7(1)
               6(4)
                    5(1)
          4(8)
                    3(1)
               2(3)
```

```
                              1(1)
Time count: 32

Find (find the deepest node)
Enter the element to be found: 7
9 4 6 8 7 (found)
Time count: 5

FindMax
The max-key is 10
Time count: 2

FindMin
The min-key is 1
Time count: 4

DelMaxSort
10 9 8 7 6 5 4 3 2 1
Time count: 18
```

Table 2: The comparison of the average-case performance between regular BST and binary search forest

Operation	Time	
	BST	Forest
Create	32	34
Find deepest node	5	5
FindMax	2	1
FindMin	4	3
DelMaxSort	18	23
SplitSort	--	7

In this section both BST and forest are created by randomly generated data to avoid the worst-case situation. Table 2 shows that the time to create a BST of size 10 and the time to create a forest of the same size is 32 and 34, respectively. Because the data is randomly generated, the structure of a BST or a forest may differ from time to time. But in a long run, the average time to create a regular BST and the average time to create a forest are very close, suggesting that the average performance of the regular BST and the forest is in the same time complexity order.

The forest implementation still has some advantages over the regular BST in some special cases. For example, it takes only 1 time unit to find the max-key in any forest because the max-key is always located in the root of the last tree in a forest. It also takes less time to find the min-key in a forest because the min-key is always located in the first tree in a forest. To find the min-key all it takes is to search the first tree instead of the whole forest. The forest implementation also can use the SplitSort, in case that a sorting is required in practice, to change a forest or trees to a sorted linked list at very low cost. This option is not available for a regular BST.

5. SUGGESTED FURTHER STUDIES

The implementation of this algorithm could be simplified by arranging the merge process to be the last operation of a sequence of operations. After merging two trees, the merged tree is normally close to full. Thus, there is a good chance that the tree will be split again during next few insert operations. It is common practice to follow a delete operation with an insert operation. To reduce the chances of the repeated split and merge operations, we can leave the trees as they are after split or delete operations, even if they are qualified to be merged. Thus, trees are less full and have more space for insertion without the necessity of repeated splits within a short time. The merge operations are performed only at the end of a sequence of operations. This merge procedure can be done easily by scanning the root-list to find candidate trees and then merge them pair by pair. Actually, there are several alternatives to deal with the merging process, such as merging trees during the tree search process of each individual operation, or to merge trees at a given operation intervals, or to merge trees only when the number of nodes in both trees has dropped to a certain low level, such as 1/3 full.

The idea of forest implementation of binary search tree could be applied to other types of tree structures. Since each tree in the forest is m times smaller than a single tree of the same size, operations on a tree in the forest should be m times faster. This is especially important for trees which need path updating or rebalancing after an operation, because these processes could propagate to the root. Propagation costs less in a smaller tree than in a larger tree.

This data structure can also be applied to applications which need both binary search tree operations and heap operations because this data structure also has partial heap properties. For example, it can be used to sort data in a quite efficient way. Further studies are suggested in order to fully extend its applications in various areas.

6. SUMMARY

A binary search forest algorithm with worst-case time of max(O(m), O(n/m)), where m is the number of trees in the forest and n is the total number of keys, is presented in this study. The idea underlying this implementation is to break a large binary search tree into m subtrees, then to link all the subtrees together using a root-list. We impose a total order among subtrees in the forest so that the value of the minimum key of the tree on the right is larger than that of the maximum key of the tree on the left. Accessing a key in the forest takes two steps: tree searching in the root-list, and key searching in a tree.

For a given binary search tree of size n, the worst-case time is O(n). However, if the tree is broken into m subtrees then the worst-case time for each subtree is reduced to O(n/m). Although searching over the root-list is an extra operation, the process takes only O(log m) time when the root-list is arranged as an ordered array since a binary search of the root-list is possible. The worst-case time of this data structure is O(log m) + O(n/m) = O(n/m). If the root-list is implemented as a linked list, the worst-case time to search a root in the list is O(m). Thus, the worst-case time of this data structure is max (O(m) + O(n/m)). The amortized time bound for this data structure is O(z log n) for z successive operations.

Another advantage of this data structure is that it also has a partial heap order because each tree has its maximum key located in the root. Thus, we can perform the delete_max operation on this data structure for sorting purposes. The delete_max operation takes O(log (n/m)) average time. The sorting procedure can also be done by performing split operations, which takes O(n) time for sorting n nodes and O(1) average time for individual operation.

BIBLIOGRAPHY

1. Andersson, A., Christian, I., Klein, R., Ottmann, T., 1990, Binary search trees of almost optimal height. Acta Informatica. 28: 165-178.

2. Bayer, R., 1972a, Symmetric binary B-trees: Data structure and maintenance algorithms. Acta Informatica. 1 (4): 290-360.

3. Bayer, R., 1972b, Binary B-trees for virtual memory, in Proc. 1971 ACM SIGFIDET Workshop, ACM, New York, 219-235.

4. Day, A.C., 1976, Balancing a binary tree. Computer Journal. 19 (4): 360-361.

5. Huddleston, S., Mehlhorn, K., 1982, A new data structure for representing sorted lists. Acta Informatica. 17: 157-184.

6. Martin, W.A., Ness, D.N., 1972, Optimizing binary trees grown with a sorting algorithm. Communications of the ACM. 15 (1): 88-93.

7. Mauer, H.A., Ottmann, T., Six, H.W., 1976, Implementing dictionaries using binary trees of very small height. Information Processing Letters. 5 (1): 11-14.

8. McCreight, E.M., 1985, Priority search trees. SIAM Journal on Computing. 14 (2): 257-276.

9. Nievergelt, J., Reingold, E.M., 1973, Binary trees of bounded balance. SIAM Journal on Computing. 2 (1): 33-43.

10. Nurmi, O., Soisalon-Soininen, E., 1996, Chromatic binary search trees: a structure for concurrent rebalancing. Acta Informatica. 33: 547-557.

11. Olivie, H.J., 1982, A new class of balanced search trees: Half-balanced binary search trees. RAIRO Theoretical Informatics and applications. 16: 51-71.

12. Sherk, M., 1995, Self-adjusting k-way search trees. Journal of Algorithm. 19: 25-44.

13. Sleator, D.D., Tarjan, R.E., 1985, Self-adjusting binary trees. Journal of ACM 32: 652-686.

14. Stout, Q.F., Warren, B.L., 1986, Tree rebalancing in optimal time and space. Communications of the ACM. 29 (9): 902-908.

15. Tarjan, R.E. 1985, Amortized computational complexity. SIAM Journal on Algebraic

and Discrete Methods 6 (2): 306-318.

16. Weiss, M.A., 1993, Data Structures and Algorithm Analysis in C. Benjamin-Cummings, New York.

APPENDIX: SOURCD CODE

```
/*************************************************************************************
                    FOREST IMPLEMENTATION OF BINARY SEARCH TREES

                                  Shangshan Zhang

INTRODUCTION:
   This program implements a binary search tree as a binary search forest by dividing a single binary search
tree of size n into m subtrees of size n/m, then linking all subtrees together using a rootlist. Thus, the worst-
case time complexity of this implementation will be O(n/m) instead of O(n). The data structure has not only
the search-tree properties but also the binomial heap properties. It can sort n node in total O(n) time. The
penalty for this implementation is the time required for tree-splitting and tree-merging processes.

APPROACH:
   This implementation includes two major components: 1) the root-list object, and 2) the BST objects. The
root-list links all trees together to form a forest. The root-list is implemented as an ordered linked list. The
max-key of each tree is stored in the root-list as the root of that tree. The value of the root key (max-key) of
the tree in the left is smaller than the min-key of the tree in the right. Each tree in the forest is a regular
BST.
   To access a key in the forest, it needs to first search the root-list to find the tree to which the key belongs
to, then travel the tree to find the exact location of the key. A split operation is performed after an insert
operation if the number of nodes in the tree excesses the pre-set limit. Similarly, a merge operation is
required after a delete operation if the total number of nodes of two adjacent trees is less than the limit.
   Tree size can be changed dynamically by applying post-merge or post-split operations. In one extreme,
if every tree in the forest has only one node, then the forest will be a linked list. On the other hand, if the
tree size is n, then the forest is actually a single regular BST. Sorting can be achieved by applying
continuously either the delete-max operation or the post-split operation.

MAJOR OPERATIONS:
Forest Level:
1 Create, 2 Search, 3 Insert, 4 Delete, 5 Split, 6 Merge,
7 Heap sort, 8 Split sort, 9 Change tree size, 10 Print forest,
11 List roots, 12 Change forest to BST, 13 Count total nodes
Tree Level:
1 Find node, 2 Insert, 3 Delete, 4 Find min-key, 5 Delete max-key
6 Inorder print, 7 Preorder print, 8 Postorder print

TIME COMPLEXITY:
In order to make a comparison with a standard BST, time complexity is determined for each operation.
The time complexity is determined based on the number of loops and recursive calls that a given
operation performed. A global variable is used to sum the time used by various operations.
*************************************************************************************/

#include<iostream.h>
#include<fstream.h>
#include<stdlib.h>

#define Stat                    //statistics
#define MAX_NODE 200            //maximum number of nodes
```

```
class TreeNode;
class Tree;
class List;
void ShowMenu()                          //utility function
void RandomData()                        //utility function

typedef int bool;
int TreeSize = 10;                       //default tree size
int TotalNum = 20;                       //default total number of nodes
int TimeCount = 0;                       //used for statistics
int Array[MAX_NODE];                     //set up the maximum array

class TreeNode{                          //tree node class
public:
     TreeNode(int theData):myLeft(0), myRight(0), myData(theData), nodeCount(0){ }
     friend class Tree;                  //make it accessi ble by class tree
     friend class List;                  //make it accessible by class list (forest)
private:
     TreeNode *myLeft;
     TreeNode *myRight;
     int nodeCount;                      //number of nodes in a tree or subtree
     int myData;
};

class Tree{                              //binary search tree class
public:
     Tree(int theData): myRoot(0),myNext(0),myPrev(0),myData(theData),nodeCount(1){ }
     ~Tree(){ }
     friend List;
     bool InsertNode(int theData){return InsertHelp(&myRoot, theData);}
     TreeNode *DeleteNode(int theData){return myRoo t= DeleteHelp(myRoot, theData);}
     int FindMin(){TreeNode *theNode=FindMinHelp(myRoot); return theNode->myData;}
     int DelMax(){return DelMaxHelp(myRoot);}
     int FindNode(int);
     void PreOrder(){PreOrderHelp(myRoot);}
     void InOrder(){int sp = 9; InOrderHelp(myRoot, sp);}
     void PostOrder(){PostOrderHelp(myRoot);}
     void DestroyTree(){DestroyTreeHelp(myRoot);}
private:
     bool InsertHelp(TreeNode **, int);
     void PreOrderHelp(TreeNode *);
     void InOrderHelp(TreeNode *, int);
     void PostOrderHelp(TreeNode *);
     TreeNode *DeleteHelp(TreeNode *, int);
     TreeNode *FindNodeHelp(TreeNode *, int);
     TreeNode *FindMinHelp(TreeNode *);
     int DelMaxHelp(TreeNode *);
     void ResetCountUp(TreeNode *, int);
     void ResetCountDown(TreeNode *, int);
     void DestroyTreeHelp(TreeNode *);
     Tree *myNext;
     Tree *myPrev;
     TreeNode *myRoot;
     int myData;
     int nodeCount;
```

```
};

class List{                          // root-list class: link individual BSTs together
public:
      List():myHead(0), myTail(0), treeCount(1){ }
      ~List(){ }
      void Create();
      void RandCreate();
      void ShowRoot();
      void Search(int);
      Tree* FindTree(int);
      void ShowForest();
      void Merge(Tree*, Tree*);
      void Insert(int);
      void Delete(int);
      void Split(Tree*);
      void PostMerge(int);
      void PostSplit(int);
      void ChangeTreeSize();
      void ForestToTree(int);
      int TotalNodes();
      void DelMaxSort();
      void ShowMaxKey();
      void ShowMinKey();
      void DestroyForest();
private:
      Tree *myHead;
      Tree *myTail;
      int treeCount;
};


                              /*********************************
                              *   following are forest operations      *
                              *********************************/

/************************************ Create ************************************
Create the binary search forest by continuously performing insert operations
*********************************************************************************/
void List::Create()
{
      int theData, i=0;

      ifstream InputFile("Data.dat", ios::in);    //open file
      if(!InputFile){
            cout<<"File could not be opened\n";
            exit(1);
      }

      while ((InputFile>>theData)&&(i<TotalNum)){          //read in data
            Insert(theData);               //create forest by insert operation
            i++;
      }
      return;
```

```
}


/*********************************** RandCreate ***********************************
Create the binary search forest using randomly generated data
*************************************************************************************/
void List::RandCreate()
{
     int i, theData;
     for(i=0; i<TotalNum; i++){
          theData=Array[i];     //call random function
          Insert(theData);
     }
     return;
}


/*************************************** Insert ***************************************
Insert operation
First find the tree, then call the insert method of the BST object to perform the insert operation
*************************************************************************************/
void List::Insert(int theData)
{
     Tree *newTree, *currentTree, *prev, *next;
     int newData;
     bool boolean;

     if(myHead==0){                    //if empty list
          TimeCount++;                 //statistics
          newTree = new Tree(theData);
          myHead = myTail = newTree;
          return;
     }
     else if(myTail->myData<=theData){     //if theData is largest
          if(myTail->myData==theData){
               cout<<"can not insert identical data "<<currentTree->myData<<endl;
               return;
          }
          TimeCount++;                 //statistics
          currentTree=myTail;
          newData=currentTree->myData;
          currentTree->myData=theData;
          TreeNode *newNode=new TreeNode(newData);
          newNode->nodeCount=currentTree->nodeCount;
          newNode->myLeft=currentTree->myRoot;
          currentTree->myRoot=newNode;
          currentTree->nodeCount++;
     }
     else{
          currentTree = FindTree(theData); //find the internal tree
          if(theData == currentTree->myData){
               cout<<"can not insert identical data "<<currentTree->myData<<endl;
               return;
          }
```

```
        boolean=currentTree->InsertNode(theData);  //insert by BST method
        if(boolean == true)
            currentTree->nodeCount++;            //update nodeCount
    }

    if(currentTree->nodeCount>TreeSize)          //Split if necessary
        Split(currentTree);

    prev = currentTree->myPrev;                  //Merging after splitting
    next = currentTree->myNext;
    if((prev && prev->myPrev)&&((prev->nodeCount+prev->myPrev->nodeCount)<=TreeSize))
        Merge(prev->myPrev, prev);               //merge left and left->left trees
    if((next)&&((currentTree->nodeCount+next->nodeCount)<=TreeSize))
        Merge(currentTree, next);                //merge right and right->right trees
    return;
}


/*************************************** Delete ************************************
Delete an element from the forest.
First find the tree, then call the delete method of the BST object to perform the delete operation
***********************************************************************************/
void List::Delete(int theData)
{
    Tree *currentTree, *prev, *next;
    int maxData, theCount;

    currentTree = FindTree(theData);
    if((!currentTree)||((currentTree->myData!=theData)&&(!currentTree->myRoot))){
        cout<<"Can not find "<<theData<<endl;
        return;
    }
    else if((currentTree->myData == theData)&&(currentTree->myRoot==0)){
        prev=currentTree->myPrev;                    //no tree, root only
        next=currentTree->myNext;
        if((prev==0)&&(next==0))                     //only one tree
            myHead = myTail =0;
        else if(prev==0){                            //head tree
            myHead = next;
            next->myPrev = 0;
        }
        else if(next==0){                            //tail tree
            myTail = prev;
            prev->myNext = 0;
        }
        else{                                        //mid tree
            prev->myNext = next;
            next->myPrev = prev;
        }
        delete currentTree;
        currentTree = prev;
        treeCount--;                                 //update treeCount
    }
    else if(currentTree->myData == theData){         //delete root of a tree
        maxData=currentTree->DelMax();
```

```
          currentTree->myData = maxData;
          currentTree->nodeCount--;
          if(currentTree->nodeCount==1)
               currentTree->myRoot=0;
     }
     else{
          theCount = currentTree->myRoot->nodeCount;
          currentTree->DeleteNode(theData);                //delete internal node--let BST handle it
          if(currentTree->myRoot==0)                       //update nodeCount if delete success
               currentTree->nodeCount--;                   //if deleted last node except tree root
          else if(theCount==currentTree->myRoot->nodeCount + 1)
               currentTree->nodeCount--;
     }

     if(currentTree){                                      //merging if necessary
          if((currentTree->myPrev)&&(currentTree->nodeCount+currentTree->myPrev-
>nodeCount)<=TreeSize)
               Merge(currentTree->myPrev, currentTree); //merge left
          if((currentTree->myNext)&&(currentTree->nodeCount+currentTree->myNext-
>nodeCount)<=TreeSize)
               Merge(currentTree, currentTree->myNext); //merge right
     }
     return;
}


/************************************** Split **************************************
Split tree after insert operation if the size of the tree is larger than the maximum size
***********************************************************************************/
void List::Split(Tree *currentTree)
{
     TimeCount++;                                          //statistics
     Tree *prev;
     TreeNode *leftRoot, *rightRoot, *oldRoot;

     prev = currentTree->myPrev;                           //some of them could be NULL
     oldRoot=currentTree->myRoot;
     if(oldRoot==0)
          return;
     leftRoot = oldRoot->myLeft;
     rightRoot = oldRoot->myRight;

     Tree *newTree = new Tree(oldRoot->myData);            //create and insert new tree
     newTree->myRoot = leftRoot;
     newTree->myNext = currentTree;
     treeCount++;                                          //update treeCount
     if(prev==0)
          myHead=newTree;                                  //if insert in front
     else{
          newTree->myPrev = prev;
          prev->myNext = newTree;                          //if insert in mid
     }
     if(leftRoot!=0)                                       //if leftRoot exists
          newTree->nodeCount = leftRoot->nodeCount + 1;
```

```
    currentTree->myPrev = newTree;                    //adjust old tree
    currentTree->myRoot = rightRoot;
    currentTree->nodeCount -= newTree->nodeCount;
    delete oldRoot;

    return;
}


/*************************************** Merge ***************************************
Merge operation after a delete or a split operation.
Merging takes place if the number of nodes of two adjacent trees is less than the maximum size
*************************************************************************************/
void List::Merge(Tree *leftTree, Tree *rightTree)
{
    TimeCount++;                                      //statistics
    TreeNode *newNode;
    Tree *prev;

    newNode = new TreeNode(leftTree->myData);         //create and set-up new node
    newNode->myLeft = leftTree->myRoot;
    newNode->myRight = rightTree->myRoot;
    newNode->nodeCount = leftTree->nodeCount + rightTree->nodeCount - 1;
    rightTree->myRoot = newNode;                      //set-up right tree

    prev = leftTree->myPrev;                          //adjust list pointers
    if(prev==0){
        myHead=rightTree;
        rightTree->myPrev = 0;
    }
    if(prev!=0){
        prev->myNext = rightTree;
        rightTree->myPrev = prev;
    }
    rightTree->nodeCount += leftTree->nodeCount;      //update nodeCount

    delete leftTree;
    treeCount--;                                      //update  treeCount
    return;
}


/*************************************** Search ***************************************
Search the location of a given element
*************************************************************************************/
void List::Search(int theData)
{
    Tree *currentTree;

    currentTree=FindTree(theData);
    if(currentTree==0){
        cout<<"Element not found\n\n";
        return;
    }
    else if(currentTree->myData == theData){
        cout<<"found "<<currentTree->myData<<" (root)\n";
    }
    else{
        cout<<currentTree->myData<<" ";
        currentTree->FindNode(theData);
    }
    return;
}


/**************************************FindTree **************************************
Find the tree to which a given element belongs.
Search the root-list to find the tree whose root-key value is larger than the given data
*************************************************************************************/
Tree* List::FindTree(int theData)
{
    Tree *currentTree;
    currentTree = myHead;
    while(currentTree != 0){
        TimeCount++;
        if(theData<=currentTree->myData)                //found the tree
            return currentTree;
        else{
            currentTree = currentTree->myNext;
        }
    }
    return currentTree;
}


/**********************************PostMerge **************************************
This independent tree-merge process is not associated with any individual delete operation. It can be
performed after a series of operations
*************************************************************************************/
void List::PostMerge(int theSize)
{
    Tree *leftTree, *rightTree;
    if(myHead==0)
        return;
    leftTree = myHead;
    rightTree = myHead->myNext;
    while ((leftTree)&&(rightTree)){
        if((leftTree->nodeCount + rightTree->nodeCount)<=theSize)
            Merge(leftTree, rightTree);
        leftTree = rightTree;
        rightTree = rightTree->myNext;
    }
    return;
}
```

```
/**********************************PostSplit *************************************
This independent tree-split process is not associated with any individual insert operation.  It can be
performed after a series of operations
**************************************************************************************/
void List::PostSplit(int theSize)
{
     Tree *currentTree=myHead;

     while(currentTree!=0){
          while(currentTree->nodeCount>theSize){
               Split(currentTree);
               if((currentTree)&&(currentTree->myPrev->nodeCount>theSize))
                    currentTree = currentTree->myPrev; //split again if still large
          }
          currentTree=currentTree->myNext;               //split next tree
     }
     return;
}


/******************************** ChangeTreeSize  *********************************
Change the size of binary search trees in the forest
**************************************************************************************/
void List::ChangeTreeSize()
{
     PostSplit(TreeSize);                         //split if tree size becomes smaller
     PostMerge(TreeSize);                         //merge if tree size becomes larger
     return;
}


/***********************************ForestToTree **********************************
Change the binary search forest to a single binary search tree
**************************************************************************************/
void List::ForestToTree(int theSum)
{
     int i;

     for(i=2*TreeSize; i<=theSum; i=i*2){
          PostMerge(i);                           // heapify merge
     }
     if((i/2)!=theSum)                            //if odd  num or 2*TreeSize > theSum
          PostMerge(theSum);
     return;
}

/*********************************DelMaxSort  *************************************
Sorting by continuously performing delete-max operation
**************************************************************************************/
void List::DelMaxSort()
{
     Tree *currentTree, *tempTree;
     int i=0;
```

```
    TotalNum = TotalNodes();
    currentTree = myTail;
    myTail=0;
    while(currentTree!=0){                          //from one tree to another
         tempTree=currentTree;
         while (currentTree->nodeCount>1){          //perform DelMax in BST
              Array[i++]=currentTree->myData;
              currentTree->myData=currentTree->DelMax();
              currentTree->nodeCount--;
         }
         Array[i]=currentTree->myData;              //last element stored in the root
         currentTree = currentTree->myPrev;
         delete tempTree;
         treeCount--;                               //update treeCount
         i++;
    }
    myHead=0;
    myTail=0;
    treeCount=1;
    for(i=0; i<TotalNum; i++)                       //print result
         cout<<Array[i]<<" ";
    cout<<endl;
    return;
}


/********************************** TotalNodes **********************************
Obtain the total number of nodes in the forest
*********************************************************************************/
int List::TotalNodes()
{
    int theSum=0;
    Tree *tempTree;

    tempTree = myHead;
    while(tempTree!=0){                             //sum nodes of each tree
         TimeCount++;                               //statistics
         theSum = theSum + tempTree->nodeCount;
         tempTree=tempTree->myNext;
    }

    return theSum;
}


/********************************** ShowForest **********************************
Show forest -- print each individual tree
*********************************************************************************/
void List::ShowForest()
{
    int treeNum=treeCount;
    Tree *tempTree;
    tempTree = myTail;
    while(tempTree!=0){
         TimeCount++;                               //statistics
```

```
        cout<<"T"<<treeNum<<": ";
        cout<<tempTree->myData<<"("<<tempTree->nodeCount<<")\n"; //root
        tempTree->InOrder();                          //print each BST with tree appearance
        tempTree=tempTree->myPrev;
        cout<<"\n\n";
        treeNum--;
    }
    return;
}


/*********************************** ShowRoot *************************************
Show the root of each tree
***********************************************************************************/
void List::ShowRoot()
{
    Tree *tempTree;
    tempTree = myHead;
    while(tempTree!=0){
        TimeCount++;                                  //for statistics
        cout<<tempTree->myData<<"("<<tempTree->nodeCount<<") ";
        tempTree=tempTree->myNext;
    }
    cout<<endl;
    return;
}


/************************************ShowMaxKey ********************************
Find the max-key of the forest
***********************************************************************************/
void List::ShowMaxKey()
{
    cout<<"The max-key is: "<<myTail->myData<<endl;
    TimeCount++;
    return;
}


/************************************ShowMinKey ********************************
Find the min-key of the forest
***********************************************************************************/
void List::ShowMinKey()
{
    if(myHead!=0){
        TimeCount++;
        if(myHead->myRoot==0)
            cout<<"the min-key is: "<<myHead->myData<<"\n";
        else
            cout<<"the min-key is: "<<myHead->FindMin()<<"\n";
    }
    return;
}
```

```
/****************************** DestroyForest  *****************************
Destroy the forest and release occupied memory
***************************************************************************/
void List::DestroyForest()
{
     Tree *currentTree, *tempTree;

     currentTree = myHead;
     while(currentTree != 0){
          currentTree->DestroyTree();
          tempTree=currentTree;
          currentTree=currentTree->myNext;
          delete tempTree;
     }
     myHead=0;
     myTail=0;
     treeCount=1;
     return;
}


                         /*******************************
                         * following are BST operations       *
                         *******************************/

/******************************** FindNode  ********************************
Find a specific node
***************************************************************************/
int Tree::FindNode(int theData)
{
     TreeNode *tempNode;

     tempNode=FindNodeHelp(myRoot, theData);
     if(tempNode==0)
          return -1;
     else
          return tempNode->myData;
}


/******************************** FindNodeHelp  *****************************
Actual find-node operation
***************************************************************************/
TreeNode *Tree::FindNodeHelp(TreeNode *theNode, int theData)
{
     TimeCount++;                                    //used for statistics
     if(theNode==0){
          cout<<"Element not found\n";
          return 0;
     }
     else {
          cout<<theNode->myData<<" ";
          if(theNode->myData==theData){
               cout<<"(found)"<<"\n";
```

```
            return theNode;
        }
        else if(theNode->myData<theData)
            FindNodeHelp(theNode->myRight, theData);
        else
            FindNodeHelp(theNode->myLeft, theData);
    }
    return 0;
}


/******************************** InsertHelp ********************************
Actual Insert operation
****************************************************************************/
bool Tree::InsertHelp(TreeNode **theNode, int theData)
{
    TimeCount++;                                        //statistics
    if(*theNode==0){
        *theNode=new TreeNode(theData);
        (*theNode)->nodeCount++;
    }
    else{
        if(theData > (*theNode)->myData){
            (*theNode)->nodeCount++;
            InsertHelp(&((*theNode)->myRight), theData);
        }
        else{
            if(theData<(*theNode)->myData){
                (*theNode)->nodeCount++;
                InsertHelp(&((*theNode)->myLeft), theData);
            }
            else{
                ResetCountDown(myRoot, theData);
                cout<<"Can not insert identical data: "<<theData<<endl;
                return false;
            }
        }
    }
    return true;
}


/********************************* DeleteHelp ******************************
Actual delete operation
****************************************************************************/
TreeNode *Tree::DeleteHelp(TreeNode *theNode, int theData)
{
    TimeCount++;                                        //used for statistics
    TreeNode *temp, *child;
    if(theNode==0){
        cout<<"Can not find "<<theData<<"\n";
        ResetCountUp(myRoot, theData);
    }
    else if(theData<theNode->myData){                   //go left
        theNode->nodeCount--;
```

```
            theNode->myLeft=DeleteHelp(theNode->myLeft, theData);
      }
      else if(theData>theNode->myData){                          //go right
            theNode->nodeCount--;
            theNode->myRight=DeleteHelp(theNode->myRight, theData);
      }
      else if((theNode->myRight)&&(theNode->myLeft)){      //found node with two children
            temp=FindMinHelp(theNode->myRight);              //replace with smallest in right subtree
            theNode->myData=temp->myData;
            theNode->myRight=DeleteHelp(theNode->myRight, theNode->myData);
      }
      else{                                                      //one child and no child
            temp=theNode;
            if(theNode->myRight==0)                              //no right child
                  child=theNode->myLeft;
            if(theNode->myLeft==0)                               //no left child
                  child=theNode->myRight;
            delete(temp);
            return child;                           //return child as myRight or myLeft of parent node
      }
      return theNode;                                            //if not found
}


/************************************DelMaxHelp ********************************
Actual delete-maximum-key operation
*********************************************************************************/
int Tree::DelMaxHelp(TreeNode *theNode)
{
      TreeNode *tempNode, *deletedNode;
      int theData;

      if((theNode->myRight==0)&&(theNode->myLeft==0)){           //if root only
            TimeCount++;                                     //statistics
            theData=theNode->myData;
            delete theNode;
            theNode = 0;
            return theData;
      }

      if(theNode->myRight==0){                               //if root has only left child
            TimeCount++;                                     //statistics
            theData=theNode->myData;
            myRoot = theNode->myLeft;
            delete theNode;
            return theData;
      }

      else{
            TimeCount++;                                      //statistics
            tempNode = theNode;
            tempNode->nodeCount--;
            while(tempNode->myRight->myRight!=0){
                  TimeCount++;                                //statistics
                  tempNode=tempNode->myRight;
            tempNode->nodeCount--;
        }
        theData = tempNode->myRight->myData;
        deletedNode = tempNode->myRight;
        tempNode->myRight=tempNode->myRight->myLeft;
        delete deletedNode;
        return theData;
    }
}


/********************************** FindMinHelp **********************************
Actual operation of find minimum-key
***********************************************************************************/
TreeNode *Tree::FindMinHelp(TreeNode *theNode)
{
    TimeCount++;                                        //used for statistics
    if(theNode==0)
        return 0;
    else if(theNode->myLeft==0)
        return theNode;
    else
        return FindMinHelp(theNode->myLeft);
}


/********************************** PreOrderHelp **********************************
Actual operation of preorder print of BST
***********************************************************************************/
void Tree::PreOrderHelp(TreeNode *theNode)
{
    TimeCount++;                                        //used for statistics
    if(theNode!=0){
        cout<<theNode->myData<<"("<<theNode->nodeCount<<") ";
        PreOrderHelp(theNode->myLeft);
        PreOrderHelp(theNode->myRight);
    }
    return;
}


/********************************** InOrderHelp **********************************
Actual operation of inorder print of BST.
This operation will print-out the BST with tree structure (not just data). This function is a modified
inorder operation
***********************************************************************************/
void Tree::InOrderHelp(TreeNode *theNode, int sp)
{
    TimeCount++;
    int i;
    if(theNode!=0){
        if(theNode->myRight)
            InOrderHelp(theNode->myRight, sp+5);
        for(i=0; i<sp; i++)                             //print space between nodes
            cout<<" ";
```

```
          cout<<theNode->myData<<"("<<theNode->nodeCount<<")\n";
          if(theNode->myLeft)
               InOrderHelp(theNode->myLeft, sp+5);
     }
     return;
}


/********************************* PostOrderHelp ********************************
Actual operation of postorder print of BST.
***************************************************************************************/
void Tree::PostOrderHelp(TreeNode *theNode)
{
     TimeCount++;                                    //used for statistics
     if(theNode!=0){
          PostOrderHelp(theNode->myLeft);
          PostOrderHelp(theNode->myRight);
          cout<<theNode->myData<<" ";
     }
     return;
}



/*********************************** ResetCountUP *****************************
Update the count of nodes up.
***************************************************************************************/
void Tree::ResetCountUp(TreeNode *theNode, int theData)
{
     if(theNode==0)
          return;
     else{
          theNode->nodeCount++;
          if(theNode->myData<theData)
               ResetCountUp(theNode->myRight, theData);
          else
               ResetCountUp(theNode->myLeft, theData);
          return;
     }
}


/*********************************** ResetCountDown ****************************
Update the count of nodes down
***************************************************************************************/
void Tree::ResetCountDown(TreeNode *theNode, int theData)
{
          if(theNode==0)
               return;
          else{
               theNode->nodeCount--;
               if(theNode->myData<theData)
                    ResetCountDown(theNode->myRight, theData);
               else
                    ResetCountUp(theNode->myLeft, theData);
```

```
                return;
            }
}


/************************************ DestoryTree *********************************
Destroy the tree and release occupied memory
************************************************************************************/
void Tree::DestroyTreeHelp(TreeNode *theNode)
{
    if(theNode != 0){
        DestroyTreeHelp(theNode->myLeft);
        DestroyTreeHelp(theNode->myRight);
        delete theNode;
    }
    return;
}


/*************************************RandomData *********************************
Create a set of random data--utility function
************************************************************************************/
RandomData()
{
    int flag, theData, i=0, j=0;
    while(i<TotalNum){
        theData=rand()%TotalNum+1;
        flag=0;
        for(j=0; j<=i; j++){
            if(theData==Array[j]){
                flag=1;
                break;
            }
        }
        if(flag==0){
            Array[i++]=theData;
        }
    }
    return 0;
}


/*************************************ShowMenu *********************************
Print the detail of each command (independent function)
This is an utility function
************************************************************************************/
void ShowMenu()
{
    cout<<"*****************************************************************\n";
    cout<<"0 Stop: Exit the program.          1 Create: Create a forest using file data\n";
    cout<<"2 Insert: Insert a node.        3 Delete: Delete a node\n";
    cout<<"4 ShowForest: Print forest.        5 ChangeSize: Change tree size\n";
    cout<<"6 ShowRoot: Print tree roots.      7 ChangeToBST: make forest to a BST\n";
    cout<<"8 TotalNodes: Count total nodes.   9 DelMaxSort: Sort by delMax operation\n";
    cout<<"10 Search: Find a node             11 RandomCreate: Create a forest using\n";
    cout<<"12 MaxKey: Show max-key                   randomly generated data\n";
    cout<<"13 MinKey: Show min-key          14 Help: Show detail command menu\n";
    cout<<"15 Clear: Destroy the forest\n\n";
    cout<<"note: number in parenthesis indicates the number of nodes in that tree\n";
    cout<<"*******************************************************************\n\n";
    return;
}


int main()
{
    List myList;
    int theData, command, theSum, timeCount;

    do{
        #ifdef Stat
            TimeCount=0;
        #endif
        cout<<"\n----------------------------------------------------------------------\n";
        cout<<"0 Stop, 1 Create, 2 Insert, 3 Delete, 4 ShowForest, 5 ChangeSize\n";
        cout<<"6 ShowRoots, 7 ChangeToBST, 8 TotalNodes, 9 DelMaxSort, 10 Search\n";
        cout<<"11 RandomCreate, 12 MaxKey, 13 MinKey, 14 Help, 15 Clear—ENTER
                COMMAND: ";
        cin>>command;
        cout<<"----------------------------------------------------------------------\n\n";
        switch(command){
        case 0:
            return 0;
            break;
        case 1:
            cout<<"Enter tree size: ";
            cin>>TreeSize;
            cout<<endl;
            cout<<"Enter the total number of nodes: ";
            cin>>TotalNum;
            cout<<endl;
            myList.Create();
            #ifdef Stat
                timeCount = TimeCount;
            #endif
            myList.ShowForest();
            cout<<"\nTime_count: "<<timeCount<<"\n\n";
            break;
        case 2:
            cout<<"Enter the element to be inserted: ";
            cin>>theData;
            cout<<endl;
            myList.Insert(theData);
            #ifdef Stat
                timeCount = TimeCount;
            #endif
            myList.ShowForest();
            cout<<"\nTime_count: "<<timeCount<<"\n\n";
            break;
        case 3:
```

```
        cout<<"Enter the element to be deleted: ";
        cin>>theData;
        cout<<endl;
        myList.Delete(theData);
        #ifdef Stat
            timeCount = TimeCount;
        #endif
        myList.ShowForest();
        cout<<"\nTime_count: "<<timeCount<<"\n\n";
        break;
    case 4:
        myList.ShowForest();
        #ifdef Stat
            cout<<"\nTime_count: "<<TimeCount<<"\n\n";
        #endif
        break;
    case 5:
        cout<<"Enter tree size: ";
        cin>>TreeSize;
        cout<<endl;
        myList.ChangeTreeSize();
        #ifdef Stat
            timeCount = TimeCount;
        #endif
        myList.ShowForest();
        cout<<"\nTime_count: "<<timeCount<<"\n\n";
        break;
    case 6:
        myList.ShowRoot();
        #ifdef Stat
            cout<<"\nTime_count: "<<TimeCount<<"\n\n";
        #endif
        break;
    case 7:
        theSum=myList.TotalNodes();
        TimeCount=0;
        myList.ForestToTree(theSum);
        #ifdef Stat
            timeCount = TimeCount;
        #endif
        myList.ShowForest();
        cout<<"\nTime_count: "<<timeCount<<"\n\n";
        break;
    case 8:
        cout<<"the total number of nodes of the forest is "<<myList.TotalNodes()
                    <<"\n";
        #ifdef Stat
            cout<<"\nTime_count: "<<TimeCount<<"\n\n";
        #endif
        break;
    case 9:
        myList.DelMaxSort();
        #ifdef Stat
            cout<<"\nTime_count: "<<TimeCount<<"\n\n";
        #endif
```

```
                break;
            case 10:
                cout<<"Enter the element to be found: ";
                cin>>theData;
                cout<<endl;
                myList.Search(theData);
                #ifdef Stat
                    cout<<"\nTime_count: "<<TimeCount<<"\n\n";
                #endif
                break;
            case 11:
                cout<<"Enter tree size: ";
                cin>>TreeSize;
                cout<<endl;
                cout<<"Enter the total number of nodes: ";
                cin>>TotalNum;
                cout<<endl;
                RandomData();
                myList.RandCreate();
                #ifdef Stat
                    timeCount = TimeCount;
                #endif
                myList.ShowForest();
                cout<<"\nTime_count: "<<timeCount<<"\n\n";
                break;
            case 12:
                myList.ShowMaxKey();
                #ifdef Stat
                    cout<<"\nTime_count: "<<TimeCount<<"\n\n";
                #endif
                break;
            case 13:
                myList.ShowMinKey();
                #ifdef Stat
                    cout<<"\nTime_count: "<<TimeCount<<"\n\n";
                #endif
                break;
            case 14:
                ShowMenu();
                break;
            case 15:
                myList.DestroyForest();
                break;
            default:
                cout<<"wrong command\n";
            }
        }while(command!=0);
        return 0;
}
```

VITA

Shangshan Zhang

Candidate for the Degree of

Master of Science

Thesis: BINARY SEARCH ALGORITHM

Major Field: Computer Science

Biographical:

Personal data: Born on February 16, 1958 in Chongqing, China. Married to the former Jian Su of Beijing, China in 1985. They have one son, Charles, age eleven.

Education: Received Bachelor of Science degree in Forestry from Beijing Forestry University, Beijing, China in February 1982; received Master of Arts degree in Biology from Indiana University, Bloomington, Indiana in August, 1988; received Doctor of Philosophy degree from North Carolina State University, Raleigh, North Carolina in August 1993. Completed the Requirements for the Master of Science degree at Oklahoma State University in December 1998.

Experience: Worked as a Research Assistant for Forestry Department, North Carolina State University from 1990 to 1993; employed as a Research Associate by Oklahoma State University, Department of Forestry, 1994 to 1996.