Project2
Build
Binary Search
Tree

Author Names:Group2

Date: 2018-11-02

Contents

1. Introduction

1.1 Background description

A binary search tree is a binary tree which may be empty. The tree satisfies the binary search property, which states that the key in each node must be greater than or equal to any key stored in the left sub-tree, and less than or equal to any key stored in the right sub-tree. It support three main operations: insertion of elements, deletion of elements, and lookup (checking whether a key is present). The shape of the binary search tree depends entirely on the order of insertions and deletions, and can become degenerate.

Binary search trees are a fundamental data structure used to construct more abstract data structures such as sets, multisets, and associative arrays.

1.2 Our expected goals

Given the structure of a binary tree and a sequence of distinct integer keys, we are supposed to find the only way to fill these keys into the tree to make the resulting tree a binary search tree. To present the results, we are asked to output the level order travelsal sequence of the tree. Besides, we shall provide a set of test cases to confirm the accuracy of the program.

2. Algorithm Specification

2.1 Data Structure used to represent BST

Since the number of nodes in the binary search tree and the left and right children of each node have been given, it is very convenient to represent this binary tree with an array of structures containing the indexes of left and right child and key value of each node.

We can easily represent the binary search tree as an array of structures as shown below (note that -1 represents the NULL child pointer)

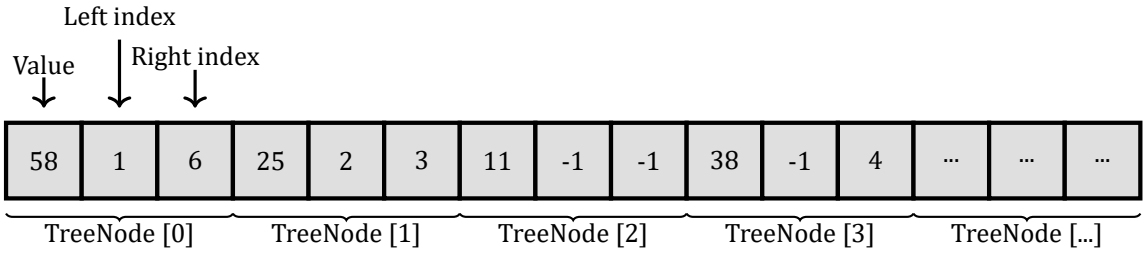


Figure 2.1: Represention of A Binary Search Tree

2.2 Algorithm to build BST

We need to build a binary search tree based on the given N distinct integer keys.

The property of the binary search tree states that the key in each node must be greater than or equal to any key stored in the left sub-tree, and less than or equal to any key stored in the right sub-tree.So when we do the in-order traversal of the binary search tree, we can get the sequence of key values in ascending order.

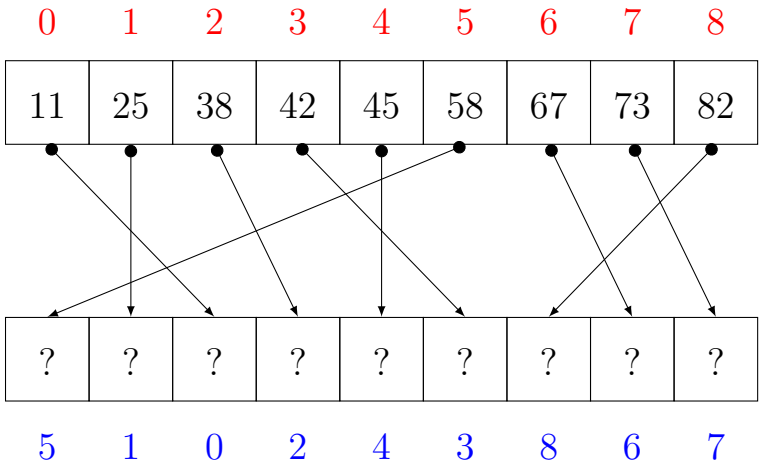
Conversely, when we sort out the out-of-order key values in ascending order and use the in-order traversal to set them to the value of the binary search tree in turn, we successfully build a binary search tree.

So we only need **three steps** to build a binary search tree.

step 1: Building a binary tree

step 2: Sorting Keys

step 3: Setting Value of Node by Inorder-traversing



We use **Quick sort** and **Inorder-Traversal** with recursion.

2.2.1 The pseudo-code of Quick Sort

Algorithm 1 **Quick sort**

Input:

A is an array of integer

QUICKSORT(A,p,r)
if p<r **then**
  $q \leftarrow$PARTITION(A,p,r)
  QUICKSORT(A,p,$q-1$)
  QUICKSORT(A,$q+1$,r)
end if

PARTITION(A,p,r)
$pivot \leftarrow A[r]$
$i \leftarrow p$
for $j := ptor - 1$ **do**
  if $A[j \leq pivot$ **then**
    swap $A[i]$ with $A[j]$
    $i := i + 1$
  end if
end for
swap $A[i]$ with $A[r]$
return i

2.2.2 The pseudo-code of Inorder-Traversal and Setting Value

Algorithm 2 **InorderSetValue**

Input:

T is a binary search tree
$Root$ is the root node
Key is the array of keys of ascending order

1: **if** $Root$ has a left child **then**
2: **InorderSetValue**(T, the left child of $Root$)
3: **end if**
4: The value of $Root$:= Key[$index$]
5: $index$:=$index$+1
6: **if** $Root$ has a right child **then**
7: **InorderSetValue**(T, the right child of $Root$)
8: **end if**

2.3 Level Order Tree Traversal

Level order traversal of a tree is breadth first traversal for the tree.We can implement it bu using a queue.

For each node, first the node is visited and then it' s child nodes are put in a FIFO queue.

Queue is implemented using an array with maximum size of 100. We can implement queue as linked list also.

2.3.1 The pseudo-code of Level Order Tree Traversal

Algorithm 3 **Levelorder Traversal**

Input: T is a tree to be traversaled $TempNode$ is the root of T

1: Create an empty queue Q
2: **while** $TempNode$ is not NULL **do**
3: **print** Data of $TempNode$
4: Enqueue $TempNode$'s children(first left then right)
5: $TempNode \leftarrow$ Dueque(Q)
6: **end while**

3. Testing Results

In this chapter we will show the correctness of our algorithm and the robustness of our program by 6 representative and identifiable test case.

We think that these 6 conditions should be representative of most conditions. So we are convinced that our program works correctly. Details and results of these test cases are as follows.

3.1 Details of test case

3.1.1 Test case 1

Test ID: 1
Test File: TestCase1.dat
Test Purpose: To demonstrate correct execution in case that the tree is a single root

Fig. 3.1. Test Case 1 (A single root)


```
Input:
1
-1 -1
X (X is an arbitrary integer)
```

```
Expected Output:
X
```

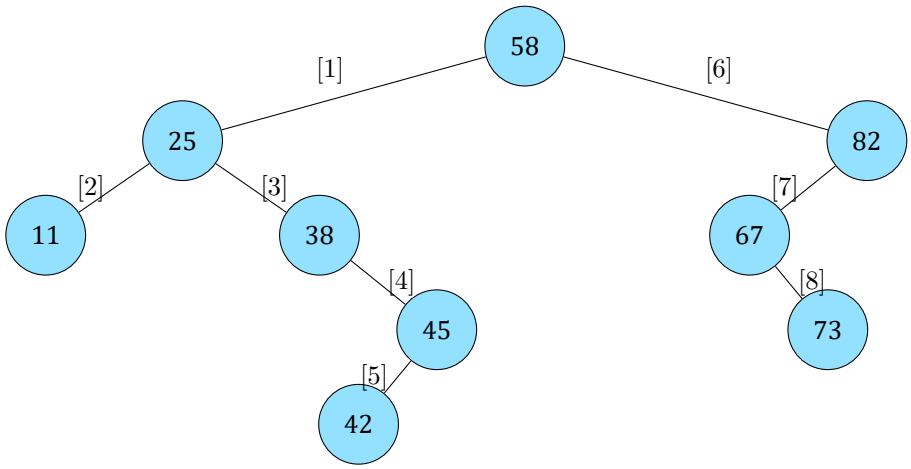
```
Actual Output:
X
```

```
Testing Result:
PASS
```

3.1.2 Test case 2

Test ID: 2
Test File: TestCase2.dat
Test Purpose: To demonstrate correct execution in a normal case(Sample)

Fig. 3.2. Test Case 2 (Sample)




```
Input:
Get from TestCase2.dat
```

```
Expected Output:
58 25 82 11 38 67 45 73 42
```

```
Actual Output:
58 25 82 11 38 67 45 73 42
```

```
Testing Result:
PASS
```

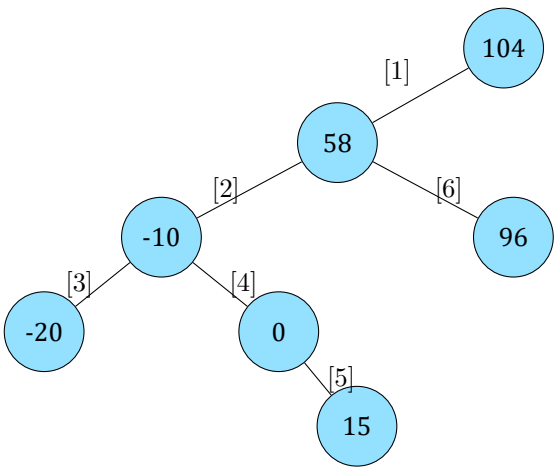
3.1.3 Test case 3

Test ID: 3

Test File: TestCase3.dat

Test Purpose: To demonstrate correct execution in case that the root has no right child and keys contains negative numbers

Fig. 3.3. Test Case 3



```
Input:
Get from TestCase3.dat
```

```
Expected Output:
104 58 -10 96 -20 0 15
```

```
Actual Output:
104 58 -10 96 -20 0 15
```

```
Testing Result:
PASS
```

3.1.4 Test case 4

Test ID: 4
Test File: TestCase4.dat
Test Purpose: To demonstrate correct execution in case of largest size of node (N = 100)

```
Input:
Get from TestCase4.dat
```

```
Expected Output:
```

```
Actual Output:
```

```
Testing Result:
PASS
```

3.1.5 Test case 5

Test ID: 5
Test File: TestCase5.dat
Test Purpose: To demonstrate correct execution in case that the tree is oblique and every non-leaf node has only a left child and no right child.

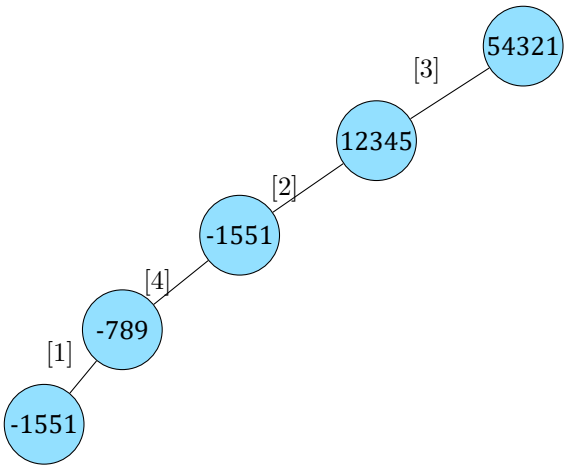
```
Input:
Get from TestCase5.dat
```

```
Expected Output:
54321 12345 1551 789 -1551
```

```
Actual Output:
54321 12345 1551 789 -1551
```

```
Testing Result:
PASS
```

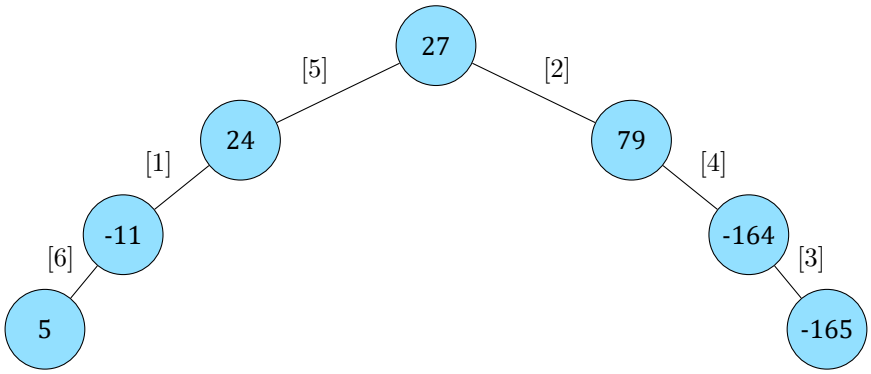
Fig. 3.4. Test Case 5



3.1.6 Test case 6

Test ID: 6
Test File: TestCase6.dat
Test Purpose: To demonstrate correct execution in case that the tree seems like a trumpet.

Fig. 3.5. Test Case 6




```
Input:
Get from TestCase6.dat
```

```
Expected Output:
27 24 79 11 164 5 165
```

```
Actual Output:
27 24 79 11 164 5 165
```

```
Testing Result:
PASS
```

4. Analysis and Comments

4.1 Time and Space complexity analysis

4.1.1 Step 1: Build a binary tree

Since we use a structure array to represrnt the binary tree, an on-line method can be used to build a binary tree from input data.

```
Tree BuildTree(FILE* fp, int N){
    int index; /* the index of current node */
    /* initialize the tree */
    Tree T = (Tree)calloc(sizeof(struct TreeNode),(N));
    if (T==NULL){
        printf("Error! Can't allocate memory!\n");
        return 0;
    }
    /* read the parent-child relationship
    and build the tree */
    for (index=0; index<N; index++){
        /* get the index of left and right child */
        int left_index, right_index;
        fscanf(fp, "%d%d", &left_index,&right_index);
        T[index].left = left_index;
        T[index].right = right_index;
        T[index].value = INF;
    }
    return T;
}
```

It can be easily seen that we only traversed the input data once. So the time complexity of this step is $O(N)$. We created a structure array of size N. So the space complexity of this step is $O(N)$.

We also created an array of N size for the storage of N keys, so we get the keys from input data with $O(N)$ space time complexity.

4.1.2 Step 2: Sort the keys

We used quick sort to make these key values sorted in ascending order.

```
void SortKey(int* A, int left, int right){
    int i,j,temp,key;
    key = A[left]; /* set a key to compare */
    i = left; j = right;
    while (i<j){
        while (i<j && A[j]>=key)
            j--;
        A[i] = A[j];
```

```
        while (i<j && A[i]<=key)
            i++;
        A[j] = A[i];
    }
    A[j] = key;
    if (left < j-1)
        SortKey(A,left,j-1);
    if (j+1 < right)
        SortKey(A,j+1,right);
}
```

Since the QuickSort procedure is a recursive algorithm, the space complexity depends on the recursion depth. For average, the recursion will repeat $logN$ times, so the space complexity is $O(log_2 N)$. For the worst case, QuickSort will degenerate into BubbleSort. The recursion will repeat N times, so the worst space complexity is $O(N)$.

4.1.3 Inorder Traversal and Set value

We implement the inorder traversal using recursion.

```
void InOrderSetValue(Tree T, int root){
    if (T[root].left != Empty){
        InOrderInsert(T, T[root].left);
    }
    T[root].value = *K; /* set the value */
    K++; /* update the pointer */
    if (T[root].right != Empty){
        InOrderInsert(T, T[root].right);
    }
}
```

The maximum numbers of frame on the function stack at the same time is h, where h is the height of the tree. That said, the worst case space complexity can be $O(N)$.

We travrese each node only once. And for each node 1 increment, 2 comparison and 1 assignment are performed.That means we will get a time complexity of $O(N)$, where N is the number of nodes in binary search tree.

4.1.4 Level order Traversal

We implement the level order traversal using a queue.

```
void LevelOrder(Tree T){

    struct TreeNode queue[MaxTree], p;

    int head = 0, tail = 1;
    queue[0] = T[0];
    while (head != tail){
    /* the queue is not empty */
```

```
            p = queue[head];
            /* get the head of queue */
            head = (head+1) % MaxTree; /* update the head */
            fprintf(fp, "%d ", p.value); /* visit the node */
            if (p.left != Empty){
            /* if the current node has left children,
            let it enqueue */
                queue[tail] = T[p.left];
                tail = (tail+1) % MaxTree;
            }
            if (p.right != Empty){
            /* if the current node has right children,
            let it enqueue */
                queue[tail] = T[p.right];
                tail = (tail+1) % MaxTree;
            }
        }
}
```

We created a queue of size MAXTREE, and the queue size would be proportional to the number of nodes. So the space complexity is $O(N)$

Since each node is visited twice, once during enqueue operation and once during dequeue operation, the time complexity of level order using queue is $O(N)$

4.1.5 Complexity of total procedure

According to the above analysis, the space complexity of the total procedure is $O(N)$, the time complexity of the total procedure is $O(NlogN)$

4.2 Comments

4.2.1 Accuracy

The algorithms can deal the 6 test cases and give correct answers.

4.2.2 Readility

The source codes have a lot of comments to help reading. Besides, the use of Macro substitution makes it easier to understand the whole program. Furthermore, the test code, algorithm code and the header file are all separate, which facilitates quick browsing and finding.

4.2.3 Robustness

The test program can deal with invalid inputs and some special cases. No bugs found yet.

5. Appendix

5.1 Source Code

5.1.1 main.c

```c
#include "BuildBST.c"

int main(void)
{
  char test = 'y'; /*initialize the varaible to judge whether to continue testing*/
  char filename[30] = "TestCaseX.dat"; /*char array to store file name*/
  char choice;
  Tree T;
  int* K;
  printf("----------------------------------------");

  while(test == 'y' || test == 'Y')
  {
    int N; /* the number of the nodes */
    printf("\nPlease choose the TestCase:\n>>");
    /*choose the test case*/
    scanf("%c",&choice);
    /*get the enter char*/
    while(getchar() != '\n');

    if(choice >= '1' && choice <= '4')
    {
      FILE *fp;
      filename[8] = choice;       /*set the file name*/
      int index = 0;
        /* open the test data file */
        if( (fp = fopen( filename,"r")) == NULL )
        {
            printf( "Open File Failed\n");
            exit(1);
        }

        /*Input the number of node*/
        fscanf(fp,"%d",&N);

        /*Build a binary tree*/
        T = BuildTree(fp,N);

        /*Input the keys*/
        K = GetKey(fp,N);

        /*Sort the keys*/
        SortKey(K,0,N-1);

        /*Set the value of each node of binary tree to build a binary search tree*/
        InOrderInsert(T,0,K,&index);

        /*print the data of the node in level order*/
        LevelOrder(T);
```

```
50
51          /*free the memory allocated*/
52          free(T);
53          free(K);
54      }
55
56      else
57        printf("Invalid Input\n");
58    }
59
60    return 0;
61  }
```

5.1.2 BuildBST.h

```
1  #ifndef PROJECT2_BUILDBST_H
2  #define PROJECT2_BUILDBST_H
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #define MaxTree (100) /* the max number of tree nodes */
7  #define Empty (-1) /* the value of empty node */
8  #define INF (0xFFFFFFF) /* the initial value of every tree node */
9
10 typedef struct TreeNode *Tree;
11 struct TreeNode{
12     int value; /* the element value of every node */
13     int left,right; /* the index of left node and right node */
14 };
15
16 Tree BuildTree(FILE* fp, int N);
17 int* GetKey(FILE* fp, int N);
18 void SortKey(int* A, int left, int right);
19 void InOrderInsert(Tree T, int root, int* K, int* index);
20 void LevelOrder(Tree T);
21
22 #endif
```

5.1.3 BuildBST.c

```
1  #include "BuildBST.h"
2
3  /*
4   * this function will read the relationship between nodes
5   * then build a tree with initial value INF
6   */
7  Tree BuildTree(FILE* fp, int N){
8      int index; /* the index of current node */
9      Tree T;
10     T = (Tree)malloc(sizeof(struct TreeNode)*N); /* initialize the tree */
11     if (T==NULL){
12         printf("Error! Can't allocate memory!\n");
13         return 0;
14     }
15     /* read the parent-child relationship and build the tree */
16     for (index=0; index<N; index++){
17         /* get the index of left child and right child */
18         int left_index, right_index;
19         fscanf(fp, "%d%d", &left_index,&right_index);
```

```
        T[index].left = left_index;
        T[index].right = right_index;
        T[index].value = INF; /* set the initial value */
    }
    return T;
}

/*
 * the function GetKey will read the given keys
 */
int* GetKey(FILE* fp, int N){
    int* K;
    K = (int*)malloc(sizeof(int)*N); /* initialize the key array */
    int i;
    for (i=0; i<N; i++){
        fscanf(fp,"%d",&K[i]);
    }
    return K;
}


/*
 * the function SortKey use Quick Sort algorithm to sort the key
 * the keys will be sorted in ascending order
 */
void SortKey(int* A, int left, int right){
    int i,j,temp,key;
    key = A[left]; /* set a key to compare */
    i = left; j = right;
    while (i<j){
        while (i<j && A[j]>=key)
            j--;
        A[i] = A[j];
        while (i<j && A[i]<=key)
            i++;
        A[j] = A[i];
    }
    A[j] = key;
    if (left < j-1)
        SortKey(A,left,j-1);
    if (j+1 < right)
        SortKey(A,j+1,right);
}


/*
 * according to the characteristic of the Binary Search Tree
 * its in-order traversal is an ascending sequence
 * thus, we can insert the keys into the tree while do in-order traversal
 * then a Binary Search Tree is build
 */
void InOrderInsert(Tree T, int root, int* K, int* index){
  if (T[root].left != Empty){
        InOrderInsert(T, T[root].left, K, index);
    }
    T[root].value = K[(*index)++]; /* insert the value */
    if (T[root].right != Empty){
        InOrderInsert(T, T[root].right, K, index);
    }
}

```

```c
/*
 * level-order traversal by using queue
 * first enqueue the root node
 * while queue is not empty, dequeue the node and then enqueue its children
 */
void LevelOrder(Tree T){

    struct TreeNode queue[MaxTree], p;
    /* the head points to the front of the queue
     * the tail points to the rear of the queue
     */
    int head = 0, tail = 1;
    queue[0] = T[0];
    while (head != tail){ /* the queue is not empty */
        p = queue[head]; /* get the head of queue */
        head = (head+1) % MaxTree; /* update the head */
        printf("%d ", p.value); /* visit the node */
        if (p.left != Empty){
        /* if the current node has left children, let it enqueue */
            queue[tail] = T[p.left];
            tail = (tail+1) % MaxTree;
            /* prevent array subscripts from crossing the boundary  */
        }
        if (p.right != Empty){
        /* if the current node has right children, let it enqueue */
            queue[tail] = T[p.right];
            tail = (tail+1) % MaxTree;
            /* prevent array subscripts from crossing the boundary  */
        }
    }
}
```

6. Declartion

We hereby declare that all the work done in this project titled ”Build Binary Search Tree” is of our independent effort as a group.

7. Duty Assignments:

Programmer: 章含挺
Tester: 柴子炜
Report Writer: 柴子炜/章含挺