Build A Binary Search Tree

林家丰 郭泽均 余瑞璟

Date: 2018-11-2

Catalogue

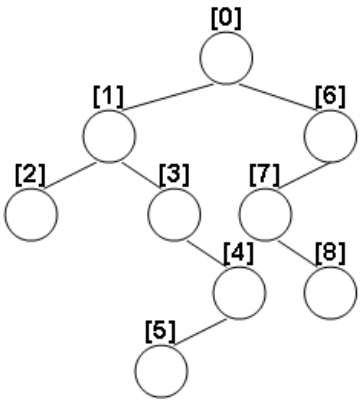
1. Chapter 1: Introduction

1.1. Problem Description

Binary search tree is a data structure, it is either an empty tree or a binary tree which has the following properties: if its left subtree is not empty, the value of all nodes in left subtree are less than the value of root node, and if its right subtree is not empty, the value of all nodes in right subtree are greater than the value of root node, both the left subtree and right subtree are binary search tree.

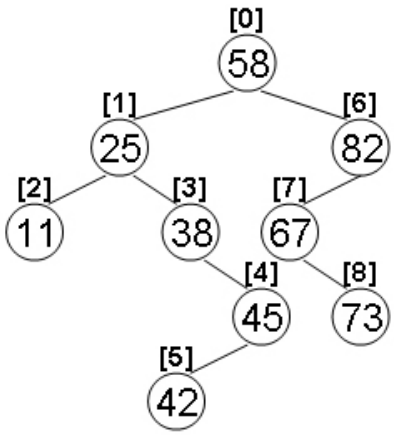
By the definition, the position of a node in a specified binary search tree determines the order of node's value in value sequence of all nodes. Consequently, a binary tree with specified structure and a sequence of distinct integer keys can determine the only binary search tree by putting these keys into the nodes according to its properties.

Example: using the given binary tree structure



and the specified integer sequence

73 45 11 58 82 25 67 38 42 , we can determine the only



binary search tree :

Based on the above, the problem requires to output the level order traversal sequence of the binary search tree according to the given binary tree structure and integer key sequence.

1.2. Detailed Requirements of Input and Output

Input: Each input file contains one test case. For each case, the first line gives a positive integer N (≤100) which is the total number of nodes in the tree. The next N lines each contains the left and the right children of a node in the format left_index right_index, provided that the nodes are numbered from 0 to N−1, and 0 is always the root. If one child is missing, then −1 will represent the NULL child pointer. Finally, N distinct integer keys are given in the last line.

Output: For each test case, print in one line the level order traversal sequence of that tree. All the numbers must be

separated by a space, with no extra space at the end of the line.

Sample Input:

```
9
1 6
2 3
-1 -1
-1 4
5 -1
-1 -1
7 -1
-1 8
-1 -1
73 45 11 58 82 25 67 38 42
```

Sample Output:

```
58 25 82 11 38 67 45 73 42
```

1.3. Algorithm Background

In-order traversal is a fundamental operation in binary tree, it can recursively define as: if the tree is empty, then do nothing, otherwise, first do an in-order traversal in left subtree, then visit the root, finally do an in-order traversal in right subtree. In binary search tree, because of its properties, the value sequence formed by in-order traversal is already sorted in ascending order. In our project, when we fill the nodes with integers in ascending sequence, we correspond every element in both integer sequence and node sequence formed by in-

order traversal, and finally put all the integers in their corresponding node to complete the construction of tree.

Level order traversal is also a fundamental operation in binary tree, it visits all the nodes once in the order that is defined as: visit the node from top level to bottom level, and visit them from left to right when in same level. In general, it is implemented based on queue structure. First enqueue the root, then repeat the following process until the queue is empty: dequeue and get the node, then enqueue its left child and right child if exists. In our project, we output the result sequence based on level order traversal operation.

2. Chapter 2: Algorithm Specification

2.1. Sketch of Main Program

We solve the problem in four steps. First, read input data and store the structure of binary search tree and integer sequence. Then, sort the integer sequence in ascending order. Next, correspond the nodes to the integers using inorder traversal and fill all the tree nodes with corresponding integers to complete the construction of tree. Finally, use level order traversal to visit all tree nodes and output the result.

2.2. Algorithm of reading data and constructing

2.2.1. Pseudo-Code

```
1.  procedure CONSTRUCT_TREE (InputData)
2.      number ← read from InputData
3.      construct a TreeArray with specified length according to the above
4.      for i ← 0 to number -1 do
5.          left_index ← read from input data
6.          right_index ← read from input data
7.          TreeArray[i].identifier ← i
8.          if left_index = -1 then
9.              TreeArray[i].left ← empty node
10.         else
11.             TreeArray[i].left ← TreeArray[left_index]
12.         end if
13.         if right_index = -1 then
14.             TreeArray[i].left ← empty node
15          else
16.             TreeArray[i].right ← TreeArray[right_index]
17.         end if
18.     end for
19. end procedure
```

2.2.2. Description

This algorithm reads the input data and stores the tree's structure and node identifiers using an array. Each element in the array represents a node in the tree, and the identifier of each node is corresponding to the index of array element. Because the input gives information in the order of identifier of the node, the procedure simply uses a loop to traverse the array and simultaneously reads data of each node, finally assigns the left child and right child for it.

2.2.3. Data Structure

The Data Structure used in this algorithm is tree array, it's an array with structural elements representing the nodes in the tree. Each structural element in array has four fundamental element: two integers respectively representing key of node and the identifier of node, two pointers respectively pointing to its left child element and right child element. This data structure stores the nodes in continuous space and uses pointer to link the nodes. It combines the advantages of array and pointer, not only can the tree be constructed quickly by simply traverse the array, but also the nodes can access quickly by skipping the space interval via pointers.

2.3. Algorithm of sorting

2.3.1. Pseudo-Code

```
1.  procedure QUICKSORT_SUB(Array, left, right)
2.      if left < right then
3.          pivot ← a chosen value in these subarray
4.          i ← left
5.          j ← right
6.          while i ≤ j do
7.              while Array[i] < pivot do
8.                  i ← i+1
9.              end while
10.             while Array[j] > pivot do
11.                 j← j-1
12.             end while
13.             if i < j then
14.                 swap Array[i] and Array[j]
15.             end if
16.         end while
17.         QUICKSORT_SUB(Array, left, j )
```

```
18.             QUICKSORT_SUB(Array, i, right)
19.         end if
20.   end procedure
21.
22.   procedure QUICKSORT(Array, number)
23.         QUICKSORT_SUB(Array, 0 , number-1)
24.   end procedure
```

2.3.2. Description

The algorithm sort the integer array and make it in ascending order. Its main idea is: divide the array in two parts, make each element in left part is less than each elements in right part, recursively process in both left and right part, and finally get a ordered array. To divide the array in two parts, the algorithm first choose a pivot from the elements in operated array, then traverse the array from left side until the element is not less than pivot, then similarly traverse the array from right side until the element is not larger than pivot. Now the algorithm stops at two elements which are both in the inverse side of the array, we simply swap them and make them both in right place, repeatedly continue the above process until two sides are met. Finally, the array will be divided into two parts, and we finish it by recursively processing the algorithm in two parts.

2.3.3. Data structure

In this algorithm, an integer array is being used to store the integer sequence.

2.4. Algorithm of filling the node with key

2.4.1. Pseudo-Code

```
1.  procedure INORDER_TRAVERSAL (TreeNode, OrderArray)
2.      index ← 0 /* excuted only for the first time running this procedure */
3.      if TreeNode is not empty then
4.          INORDER_TRAVERSAL (TreeNode.left,OrderArray)
5.          OrderArray[index] ← TreeNode.identifier
6.          index ← index +1
7.          INORDER_TRAVERSAL (TreeNode.right,OrderArray)
8.      end if
9.  end procedure
10.
11. procedure FILL_TREE(TreeArray OrderArray, IntegerArray, number)
12.     INORDER_TRAVERSAL(Tree root, OrderArray)
13.     for i ← 0 to number-1 do
14.         TreeArray[OrderArray[i]].key←IntegerArray[i]
15.     end for
16. end procedure
```

2.4.2. Description

This algorithm is divided into two sub procedures. First, use in-order traversal to store the identifiers of nodes in an array in the ascending order of node's key. Then, fill the node with the integer according to the array generated by first step. The following text will describe the details.

In first step, the algorithm first recursively traverse node's left subtree, then assign its identifier to the array element at

current index, finally recursively traverse its right subtree. "index" is defined as a global variable and is assigned to initial value 0, and it is increased by 1 whenever it visits a node in tree. Because of the property of in-order traversal, the OrderArray will finally store the identifiers of nodes in the ascending order of node's key.

In second step, the algorithm uses a loop to assign elements in ascending integer array to the nodes in the tree. In the loop, because "IntegerArray[i]" is the (i+1)th least element in integer sequence and "OrderArray[i]" is the identifier of tree node with (i+1)th least key in key sequence, we assign "IntegerArray[i] " to "TreeArray[OrderArray[i] ", while the latter is the node with (i+1)th least key in key sequence. When the loop is over, it means filling operation ends and algorithm will get a well- constructed tree.

2.4.3. Data Structure

The algorithm mainly uses three arrays, a tree array already described above and two integer arrays, which respectively store the identifier sequence and key sequence.

2.5. Algorithm of outputting level order sequence

2.5.1. Pseudo-Code

```
1.  procedure LEVELORDER_TRAVERSAL( TreeRoot, NodeNumber )
2.      if TreeRoot is empty then
3.          exit procedure
4.      end if
5.      CurrentNode ← TreeRoot
6.      construct a node queue with enough space
7.      put CurrentNode into the queue
8.      while the queue is not empty do
9.          remove the node at the head of the queue
10.         CurrentNode ← the removed node
11.         output the key value of CurrentNode
12.         if CurrentNode.left is not empty then
13.             put CurrentNode.left into the tail of queue
14.         end if
15.         if CurrentNode.right is not empty then
16.             put CurrentNode.right into the tail of queue
17.         end if
18.     end while
19. end procedure
```

2.5.2. Description

This algorithm uses level order traverse to output the level order key sequence based on a well-constructed binary tree. We use a node queue to finish this procedure. First, put the root into the queue, then do a series of operations: remove the node at the head of the queue, print the key of removed node, add its left child and right child into tail of queue if exists. Finally, repeat above process until the queue is empty. According to algorithm, the removed sequence is exactly the level order sequence of the node. Because it prints the key

every time the node is removed, it finally outputs the key of node in level order sequence.

2.5.3. Data Structure

The data structure mainly using in this algorithm is queue. It is a special linear list which has "first in first out" property. It means that, the first inserted element will be removed first in the delete operation. In our program, we implement it using node array as the container and two integer as the identifier of the head and tail of queue.

3. Chapter 3: Testing Results

3.1. Environment

Processor: intel i7-7700HQ

System: Windows 10.0.17134

Memory: 32 Gigabyte

3.2. Declaration

In the test case we selected some unique examples to test the robustness of this algorithm.

In the 1st test case, we input a tree only has left child, and see if the algorithm can work effectively, if the program didn’ t consider the recall situation then it will crash.

In the 2nd case, input a tree only has right child. If the

algorithm uses wrong table output or it only consider the left side then the algorithm will crash and print a wrong answer.

In the 3rd test case, we input a normal test case and the case is same with the PTA test example, if the case passes the program is able to switch from left child to right child and traverse the tree in level-traverse properly.

In the 4th case, we input a tree start at only has left child and then it only has right child. If the in-order traverse isn't designed properly, the tree will be filled in with a wrong array.

In the 5th case, we input only 1 node to see if the program can work if there is only head node.

In the 6th case, we test the maximum test case with 100 node to test the program in a comprehensive way.

3.3. Test Case

3.3.1. Test Case 1

Input:

```
5
1 -1
2 -1
3 -1
4 -1
-1 -1
12 23 34 56 31
```

Output:

```
56 34 31 23 12
```

3.3.2. Test Case 2

Input:

```
5
-1 1
-1 2
-1 3
-1 4
-1 -1
12 23 34 56 31
```

Output:

```
12 23 31 34 56
```

3.3.3. Test Case 3

Input:

```
9
1 6
2 3
-1 -1
-1 4
5 -1
-1 -1
7 -1
-1 8
-1 -1
73 45 11 58 82 25 67 38 42
```

Output:

```
58 25 82 11 38 67 45 73 42
```

3.3.4. Test Case 4

Input:

```
9
1 -1
2 -1
3 -1
```

```
4 -1
-1 5
-1 6
-1 7
-1 8
-1 -1
213 2313 12 45 454 5551 515 15 789165
```

Output:

```
789165 5551 2313 515 12 15 45 213 454
```

3.3.5. Test case 5

Input:

```
1
-1 -1
2333
```

Output:

```
2333
```

3.3.6. Test case 6

Input:

```
100
1 97
2 71
3 20
4 9
5 7
6 -1
-1 -1
8 -1
-1 -1
10 18
11 16
12 -1
13 15
14 -1
-1 -1
-1 -1
```

```
17 -1
-1 -1
19 -1
-1 -1
21 54
22 27
23 26
24 -1
25 -1
-1 -1
-1 -1
28 45
29 -1
30 39
31 32
-1 -1
33 36
34 35
-1 -1
-1 -1
37 38
-1 -1
-1 -1
40 43
41 -1
42 -1
-1 -1
44 -1
-1 -1
46 53
47 50
48 49
-1 -1
-1 -1
51 -1
52 -1
-1 -1
-1 -1
55 65
56 59
57 58
-1 -1
-1 -1
60 64
```

```
61 62
-1 -1
63 -1
-1 -1
-1 -1
66 68
67 -1
-1 -1
69 -1
70 -1
-1 -1
72 79
73 77
74 -1
75 76
-1 -1
-1 -1
78 -1
-1 -1
80 88
81 87
82 84
83 -1
-1 -1
85 86
-1 -1
-1 -1
-1 -1
89 93
90 91
-1 -1
92 -1
-1 -1
94 -1
95 -1
96 -1
-1 -1
98 99
-1 -1
-1 -1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83
```

```
84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
```

Output:

```
96 69 98 17 77 97 99 5 51 74 86 2 14 23 62 73 76 84 91 1 4 11
16 21 41 55 65 71 75 80 85 88 95 0 3 10 13 15 20 22 40 49 53
60 64 68 70 72 79 82 87 90 94 8 12 19 33 45 50 52 54 57 61 63
67 78 81 83 89 93 7 9 18 25 37 43 48 56 59 66 92 6 24 29 36
39 42 44 47 58 27 31 35 38 46 26 28 30 32 34
```

4. Chapter 4: Analysis and Comments

4.1. Analysis

4.1.1. Analysis for Time Complexity

Firstly, we use a recursive function to traverse the tree in in-order traversal. Because the function traverse each node only once, and the complexity of each time traverse is a constant number O(7) so the time complexity is O(7N).

Secondly, we use a cycle to fill the Tree in in-order traverse with input numbers and the time complexity is O(N).

Lastly, we use a function to output the Tree in level-order traversal. The statements in cycle is been used for N times. And the statements in while statement, which have a time complexity of O(2 + 2 + 2), the total time complexity is O(6N).

Additionally, the qsort time complexity is O(NlogN) which is easy to consider and there is no need to prove it.

In all, The total complexity is O(NlogN + 6N + 7N + N), so the time complexity of the program is O(NlogN).

4.1.2. Analysis for Space Complexity

We use a 1-dimensional array to put the data so the space complexity is O(N).

4.2. Comment

If we use a more efficient way to sort the array we can complete the program in a more efficient way.

A global data is imported in the program and this technique is not always safe.

5. Appendix: Source Code (in C)

5.1. main.c

```c
#include "BinarySearchTree.h"

/*
 * Global data: int p = 0
 * ----------------------
 * Description: record the position of array In_order
 */
int p = 0;

int main()
{
    int N; /*N is the number of nodes*/
    int i;
    int *inorder;
    int *input;
    char *filename;
    FILE *fpointer;
    Tree *Array; /*This struct array is used to record the Input*/

    filename = (char *)malloc(100 * sizeof(char));
    do{
        printf("Please Input The filename: \n");
```

```
        scanf("%s", filename);
        fpointer = fopen(filename, "r");
        if(fpointer == NULL)
            printf("Please input valid filename!\n\n");
    }while(fpointer == NULL);

    fscanf(fpointer,"%d", &N);
    fscanf(fpointer, "\n");
    /*Exit the program with error message if N was assigned 0*/
    if(N == 0){
        exit(0);
    }
    inorder = (int *)malloc(N * sizeof(int));
    input = (int *)malloc(N * sizeof(int));
    Array = (Tree *)malloc(N * sizeof(struct Node));
    for(i = 0; i < N; i++){
        Array[i] = (Tree)malloc(sizeof(struct Node)); /*Set space for
Tree Array*/
    }
    ReadFile(N, input, Array, fpointer);

    qsort(input, N, sizeof(int) ,comp);/*Use the qsort to sort input
from min to max which is the same order of Inorder Travrese*/
    In_order(Array[0],inorder);
    for(i = 0; i < N; i++){
        Array[inorder[i]]->key = input[i];
    }
    /*Print the tree by level order*/
    Level_order(Array[0],N);

    return 0;
}

/*The pointer points to function to determine sorting order (for
qsort)*/
int comp(const void *a, const void *b)
{
    return *(int *)a - *(int *)b; /*The pointer points to function to
determine sorting order for qsort*/
}

/*The Inorder Traverse*/
void In_order(Tree ptr, int *inorder)
{
```

```
    if(ptr == NULL){
        exit(0); /*Avoid pointer error*/
    }
    if(ptr->left != NULL){
        In_order(ptr->left, inorder);
    }
    inorder[p++] = ptr->rank; /*Add the rank in the array into inorder
so that each node can be visited by array[inoder[i]]*/
    if(ptr->right != NULL){
        In_order(ptr->right, inorder);
    }
}

/*Use Level_order to output to satisfy the requirement*/
void Level_order(Tree ptr, int number)
{
    Tree * Queue;
    Queue = (Tree *)malloc(sizeof(Tree *));
    int head = 0, next = 0;

    if(ptr == NULL){
        exit(0); /*Avoid visiting unintialized pointer and exit with
error message*/
    }
    Queue[0] = ptr;
    while(head <= next){
        if(Queue[head]->left != NULL){
            Queue[++next] = Queue[head]->left;
        }
        if(Queue[head]->right != NULL){
            Queue[++next] = Queue[head]->right;
        }
        if(head == 0){
            printf("%d", Queue[head]->key); /*First output with no
space*/
        }else{
            printf(" %d", Queue[head]->key); /*The output behind first
output will output a space before it*/
        }
        head++;
    }
}
```

5.2. read_file.c

```c
#include "BinarySearchTree.h"

void ReadFile(const int N, int *input, Tree *Array, FILE *fp)
{
    int i;
    int left_child, right_child;

    /*Progress the input message which contain the position information
of tree nodes*/
    for(i = 0; i < N; i++)
    {
        fscanf(fp, "%d %d", &left_child, &right_child);
        Array[i]->rank = i; /*Record the Depth-First-Search order of
the Tree*/

        /*To initialize the pointer in each nodes*/
        if(left_child > 0){ /*If the left_child is not empty*/
            Array[i]->left = Array[left_child];
        }else{
            Array[i]->left = NULL;
        }
        if(right_child > 0){ /*If the right_child is not empty*/
            Array[i]->right = Array[right_child];
        }else{
            Array[i]->right = NULL;
        }
    }
    /*Get key nodes from keyboard and sort it from small to big*/
    for(i = 0; i < N; i++){
        fscanf(fp, "%d ",&input[i]);
    }
}
```

5.3. BinarySearchTree.h

```c
#ifndef BINARYSEARCHTREE_H_INCLUDED
#define BINARYSEARCHTREE_H_INCLUDED

/*
 *Project Requirement:
 *
 *  Input:
 *  1.A positive integer N.
```

```
 *  2.N lines each contains the left and the right children of a node in
the format left_index right_index.
 *  3.N distinct integer keys.
 *  Attention:
 *  1.The nodes are numbered from 0 to N-1, and 0 is always the root.
 *  2.If one child is missing, then -1 will represent the NULL child
pointer.
 *
 *  Output:
 *  1.Print the level order traversal sequence of the tree in one line.
 *  Attention:
 *  1.All the numbers must be separated by a space, with no extra space
at the end of the line.
 *
 */
#include <stdio.h>
#include <stdlib.h>

/*
 * Type definition: typedef struct Node * Tree
 * -------------------------------------
 * Description: Define the type struct Node * to Tree so we can use the
pointer easily
 * and if we want to modify the data type it will be much easier
 */
typedef struct Node * Tree;

/*
 * Data Type: struct Node
 * ----------------------
 * Description: standard ADT with attribute rank to record the rank
message of node
 */
struct Node{
    int key;
    int rank; /*Rank refers to the sequence of nodes in the array*/
    Tree left;
    Tree right;
};
/*
 * Function: int comp(const void *a, const void *b)
 * ------------------------------------------------
 * Call: void qsort(void *base, size_t nitems, size_t size, int
(*compar)(const void *, const void*));
```

```
 * Description: determine the sort order is from small to big
 */
int comp(const void *a, const void *b);

/*
 * Function: void In_order(Tree ptr, int *inorder)
 * ------------------------------------------------
 * Description: Traverse the tree with in-order traverse recursively
 */
void In_order(Tree ptr,int * inorder);

/*
 * Function: void Level_order(Tree ptr,int number)
 * ------------------------------------------------
 * Description: Use Level_order to output to satisfy the requirement
 */
void Level_order(Tree ptr,int number);

/*

 */
void ReadFile(const int N, int *input, Tree *Array, FILE *fp);
#endif // BINARYSEARCHTREE_H_INCLUDED
```

6. Declaration

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

7. Duty Assignments

Programmer: 林家丰

Tester: 郭泽均

Report Writer: 余瑞璟