DS–Autumn 2023 — Homework 3Solutions

王子隆, SID 2221411126

2023 年 11 月 16 日

摘要

二叉树实现及应用面向过程面向对象
定义
二叉树遍历（九种方法递归非递归 * 2 * 3
preorder inorder postorder
复杂度时间空间
最坏情况最好情况最废空间最省空间
二叉树的主要性质
语法制导编辑器
修改数据结构（不借助外力 stack queue
三叉链表利用空指针（增加标识
哈夫曼树定义性质应用
二叉树分类
BST
ASL 平均查找长度

1. 二叉树实现

(a) 功能实现

(b) 应用

1.1 面向过程

Solutions:

1. 结构体 ArrayDeque.java

```
/** Array based list.
*  @author zilong
*/
```

1.2 面向对象

Solutions:

1. interface BinaryTree.java

```
#include <iostream>
#include "../hw2/SeqStack.cpp"
//#define NULL nullptr
#define cin std::cin
#define cout std::cout
#define endl std::endl


template <class T>
class BinaryTreeNode{
private:
    T data;
    BinaryTreeNode<T>*  left;
    BinaryTreeNode<T>*  right;


public:
    BinaryTreeNode();
    BinaryTreeNode(const T& elem);
    BinaryTreeNode(const T& elem, BinaryTreeNode<T>* l,
        BinaryTreeNode<T>* r);
    ~BinaryTreeNode(){};
    T value() const;
    BinaryTreeNode<T>*  leftchild() const;
    BinaryTreeNode<T>*  rightchild() const;
    void  setLeftchild(BinaryTreeNode<T>*);
    void  setRightchild(BinaryTreeNode<T>*);
    void  setValue(const T& val);
    bool  isLeaf() const;
};



template<class T>
BinaryTreeNode<T>::BinaryTreeNode(){
    data = T();
    left = nullptr;
    right = nullptr;
}

template<class T>
```

```
41      BinaryTreeNode<T>::BinaryTreeNode(const T& elem){
42          data = elem;
43          left = nullptr;
44          right = nullptr;
45      }
46
47      template<class T>
48      BinaryTreeNode<T>::BinaryTreeNode(const T& elem, BinaryTreeNode<T>* l,
            BinaryTreeNode<T>* r){
49          data = elem;
50          left = l;
51          right = r;
52      }
53
54
55      template<class T>
56      T BinaryTreeNode<T>::value() const{
57          return this->data;
58      }
59
60      template<class T>
61      BinaryTreeNode<T>*  BinaryTreeNode<T>::leftchild() const{
62          return this->left;
63      }
64
65      template<class T>
66      BinaryTreeNode<T>*  BinaryTreeNode<T>::rightchild() const{
67          return this->right;
68      }
69
70      template<class T>
71      void BinaryTreeNode<T>::setLeftchild(BinaryTreeNode<T>* Lchild){
72          this->left = Lchild;
73      }
74
75      template<class T>
76      void BinaryTreeNode<T>::setRightchild(BinaryTreeNode<T>* Rchild){
77          this->right = Rchild;
78      }
79
80      template<class T>
81      void BinaryTreeNode<T>::setValue(const T& Value){
82          this->data = Value;
83      }
84
85      template<class T>
```

```
86      bool BinaryTreeNode<T>::isLeaf() const{
87          return false;
88      }
89
90
91      enum Tag{L, R, M};
92      template <class T>
93      class StackNode{
94      public:
95          BinaryTreeNode<T>* pointer;
96          Tag tag;
97          StackNode() {pointer = nullptr; tag = L;}
98          StackNode(BinaryTreeNode<T>* ptr,Tag tg){pointer = ptr; tag =tg;}
99      };
100
101
102
103
104     template <class T>
105     class BinaryTree{
106     protected:
107         BinaryTreeNode<T>* root;
108     public:
109         BinaryTree() {root = nullptr;}
110         BinaryTree(BinaryTreeNode<T>* r) {root = r;}
111         ~BinaryTree() { DeleteBinaryTree(root); };
112         bool isEmpty() { return root==nullptr; };
113         void visit(const BinaryTree<T>& curr){cout << curr.root->value() << "␣
                ";}
114         BinaryTreeNode<T>*& Root() {return root;};
115         void CreateTree(const T& data, BinaryTreeNode<T>* lefttree,
                BinaryTreeNode<T>* righttree);
116         void CreateTree(BinaryTreeNode<T> *&r);
117         void DeleteBinaryTree(BinaryTreeNode<T>* root);
118
119         void PreOrder(BinaryTreeNode<T>* root);
120         void InOrder(BinaryTreeNode<T>* root);
121         void PostOrder(BinaryTreeNode<T>* root);
122         void PreOrderLikeRecusion(BinaryTreeNode<T>* root);
123         void InOrderLikeRecusion(BinaryTreeNode<T>* root);
124         void PostOrderLikeRecusion(BinaryTreeNode<T>* root);
125         void PreOrderWithoutRecusion(BinaryTreeNode<T>* root);
126         void InOrderWithoutRecusion(BinaryTreeNode<T>* root);
127         void PostOrderWithoutRecusion(BinaryTreeNode<T>* root);
128         void LevelOrder(BinaryTreeNode<T>* root);
129     };
```

```
130
131
132     template<class T>
133     void BinaryTree<T>::CreateTree(const T& data, BinaryTreeNode<T>*
            leftTree, BinaryTreeNode<T>* rightTree) {
134         root = new BinaryTreeNode<T>(data, leftTree, rightTree);
135         BinaryTree(root);
136     }
137
138
139
140
141     template<class T>
142     void BinaryTree<T>::DeleteBinaryTree(BinaryTreeNode<T>* Root){
143         if (Root != NULL) {
144             DeleteBinaryTree(Root->leftchild());
145             DeleteBinaryTree(Root->rightchild());
146             Root->~BinaryTreeNode();
147         }
148     }
```

2.二叉树遍历

Traversals. When we iterate over a tree, we call this a “tree traversal”.

Depth First Traversals.

We have three depth first traversals: Pre-order, in-order and post-order. In a pre-order traversal, we visit a node, then traverse its children. In an in-order traversal, we traverse the left child, visit a node, then traverse the right child. In a post-order traversal, we traverse both children before visiting. These are very natural to implement recursively. Pre-order and post-order generalize naturally to trees with arbtirary numbers of children. In-order only makes sense for binary trees.

(a) 二叉树遍历（九种方法递归非递归 * 2 * 3

preorder inorder postorder

复杂度时间空间

最坏情况最好情况最废空间最省空间

preorder

recursive

Solutions: 遍历指走一遍只走一遍递归算法的关键————写好两个条件 1.递归条件 2. 结束条件

```
template<class T>
void BinaryTree<T>::PreOrder(BinaryTreeNode<T>* root){
    if (root == NULL)  return;
    visit(root);
    PreOrder(root->leftchild());
    PreOrder(root->rightchild());
}
```

nonrecursive

1. recursivelike

   Solutions:

```
template<class T>
void BinaryTree<T>::PreOrderLikeRecusion(BinaryTreeNode <T> * root){
    SeqStack<BinaryTreeNode<T>*> tStack(10);
    BinaryTreeNode<T>* pointer = root;
    while(!tStack.IsEmpty() || pointer){
        if (pointer){
            visit(pointer);
            tStack.Push(pointer);
            pointer = pointer->leftchild();
```

```
        } else{
            pointer = tStack.Pop();
            //tStack.Pop();
            pointer = pointer->rightchild(); }
    }
}
```

2. other way

Solutions:

```
template<class T>
void BinaryTree<T>::PreOrderWithoutRecusion(BinaryTreeNode <T> * root){
    SeqStack<BinaryTreeNode<T>*> tStack(10);
    BinaryTreeNode<T>* pointer = root;
    if (!pointer) {return;};
    tStack.Push(pointer);
    while(!tStack.IsEmpty()){  //use the feature of stack
        pointer = tStack.Pop();
        visit(pointer);
        if(pointer->rightchild()){
            tStack.Push(pointer->rightchild());// first in then out
        }
        if (pointer->leftchild()){
            tStack.Push(pointer->leftchild());
        }
    }
    //does queue do the same work?
}
```

inorder

recursive

Solutions:

```
template<class T>
void BinaryTree<T>::InOrder(BinaryTreeNode<T>* root){
    if (root == NULL) return;
    InOrder(root->leftchild());
    visit(root);
    InOrder(root->rightchild());
}
```

nonrecursive

1. recursivelike

与前序指针行进类似，访问次序不同

Solutions:

```
template<class T>
void BinaryTree<T>::InOrderLikeRecusion(BinaryTreeNode <T> * root){
    SeqStack<BinaryTreeNode<T>*> tStack(10);
    BinaryTreeNode<T>* pointer = root;
    while(!tStack.IsEmpty() || pointer){
        if (pointer){
            tStack.Push(pointer);
            pointer = pointer->leftchild();
        } else{
            pointer = tStack.Pop();
            visit(pointer);
            //tStack.Push(pointer);
            //tStack.Pop();
            pointer = pointer->rightchild(); }
    }
}
```

2. other way

Solutions:

```
template<class T>
void BinaryTree<T>::InOrderWithoutRecusion(BinaryTreeNode <T> * root){
    SeqStack<StackNode<T>*> tStack(10);
    //BinaryTreeNode<T>* pointer = root;
    StackNode<T>* Tagptr = new StackNode<T>();
    Tagptr->pointer = root;

    if (!Tagptr->pointer) {return;};
    tStack.Push(Tagptr);
    while(!tStack.IsEmpty()){
        Tagptr = tStack.Pop();
        if (Tagptr->tag == L) {
            if(Tagptr->pointer->rightchild()){
                StackNode<T>* tem = new
                    StackNode<T>(Tagptr->pointer->rightchild(), L);
                tStack.Push(tem);
            }
            StackNode<T>* tem = new StackNode<T>(Tagptr->pointer, R);
            tStack.Push(tem);
            if (Tagptr->pointer->leftchild()){
                StackNode<T>* tem = new
                    StackNode<T>(Tagptr->pointer->leftchild(), L);
                tStack.Push(tem);
```

```
            }
        } else visit(Tagptr->pointer);
    }
}
```

postorder

recursive

Solutions:

```
        template<class T>
void BinaryTree<T>::PostOrder(BinaryTreeNode<T>* root){
    if (root == NULL) return;
    PostOrder(root->leftchild());
    PostOrder(root->rightchild());
    visit(root);
}
```

nonrecursive

1. recursivelike

Solutions:

```
template<class T>
void BinaryTree<T>::PostOrderLikeRecusion(BinaryTreeNode <T> * root){
    SeqStack<StackNode<T>* > tStack(10);
    StackNode<T>* Tagptr = new StackNode<T>(root, L);

    while (!tStack.IsEmpty() || Tagptr->pointer != nullptr ){
        while (Tagptr->pointer != nullptr) {
            StackNode<T>* tem = new StackNode<T>(Tagptr->pointer, L);
            tStack.Push(tem);
            Tagptr->pointer = Tagptr->pointer->leftchild();
        }
        if (!tStack.IsEmpty()) {
            Tagptr = tStack.Pop();
            if (Tagptr->tag == L ) {
                StackNode<T>* tem = new StackNode<T>(Tagptr->pointer, R);
                tStack.Push(tem);
                Tagptr->pointer = Tagptr->pointer->rightchild();
            } else {
                visit(Tagptr->pointer);
                Tagptr->pointer = nullptr;
            }
        }
```

```
    }
}
```

2. other way

Solutions:

```
template<class T>
void BinaryTree<T>::PostOrderWithoutRecusion(BinaryTreeNode <T> * root){
    SeqStack<StackNode<T>*> tStack(10);
    //BinaryTreeNode<T>* pointer = root;
    StackNode<T>* Tagptr = new StackNode<T>();
    Tagptr->pointer = root;

    if (!Tagptr->pointer) {return;};
    tStack.Push(Tagptr);
    while(!tStack.IsEmpty()){
        Tagptr = tStack.Pop();
        if (Tagptr->tag == L) {
            StackNode<T>* tem = new StackNode<T>(Tagptr->pointer, R);
            tStack.Push(tem);
            if (Tagptr->pointer->rightchild()){
                StackNode<T>* tem = new
                    StackNode<T>(Tagptr->pointer->rightchild(), L);
                tStack.Push(tem);
            }
            if (Tagptr->pointer->leftchild()){
                StackNode<T>* tem = new StackNode<T>
                    (Tagptr->pointer->leftchild(), L);
                tStack.Push(tem);
            }
        } else if (Tagptr->tag == R) {
            StackNode<T>* tem = new StackNode<T>(Tagptr->pointer, M);
            tStack.Push(tem);
        } else {
            visit(Tagptr->pointer);
        }
    }
}
```

复杂度分析

1. 复杂度时间空间

   n 为节点数每个节点都在栈里走了两遍故为2n

   空间为栈的长度，树的高度

traversalMethod	Time	Space
PreOrder	$2N$	*height*
InOrder	$2N$	*height*
PostOrder	$3N$	$\log_2 N$

2. 最坏情况最好情况

traversalMethod	Bestcase	worstcase
PreOrder	only rightchild	only leftchild
InOrder	only rightchild	only leftchild
PostOrder	FBT with least height	only leftchild

层次遍历

Level Order Traversal.

A level-order traversal visits every item at level 0, then level 1, then level 2, and so forth.

1. 借助外力 stack queue 逻辑

```
1
2  template<class T>
3  void BinaryTree<T>::LevelOrder(BinaryTreeNode <T> * root){
4      SeqQueue<BinaryTreeNode<T>*> Queue(16);
5      Queue.InQueue(root);
6      while(!Queue.IsEmpty()) {
7          //cout << "Starting Level Order Traversal..." << endl;
8          BinaryTreeNode<T>* pointer = Queue.OutQueue();
9          if (pointer->leftchild()){
10             Queue.InQueue(pointer->leftchild());
11         }
12         if (pointer->rightchild()){
13             Queue.InQueue(pointer->rightchild());
14         }
15         visit(pointer);
16     }
17 }
```

3.二叉树的定义、主要性质、定理

(a) 递归定义及基本术语

(b) 分类顺序存储链式存储

(c) 性质*5

definition

Solutions:

1. a recursive definition

   Trees.

   A tree consists of a set of nodes and a set of edges connecting the nodes, where there is only one path between any two nodes. A tree is thus a graph with no cycles and all vertices connected.

   Definition.

   we can define a binary tree as a either a null link or a node with a left link and a right link, each references to (disjoint) subtrees that are themselves binary trees.

category

Solutions:

1. sequential structure
2. list structure

properties quality character

Solutions:

1. i & 2^{i-1} floor
2. k & $2^k - 1$ all the tree
3. $n_0 = n_2 + 1$
4. height $= [\log_2 n] + 1$
5. structure of root, Lchild, Rchild

4.证明

YOUR ANSWER GOES HERE

something to prove

Solutions:

1. 给定tree 遍历序列唯一给定位置唯一存在（顺序唯一

   idea：给定位置唯一存在

2. 中序遍历猜想 + 先序猜想

3. 遍历时的性质*5

4. n node 2*n pointer N-1 in use

5.修改数据结构

我们利用stack记录路径以便遍历：右边值or访问结点值（后序）、（中反一下），利用Queue记录下一层访问节点。有没有方法可以不借助外力？修改原来的数据结构!

idea:

1. 利用现有Data Structure没有用的空间
2. 使用标识/取值范围，准确定义
3. 利用已有知识，线性表后继

(a) 利用空指针（增加标识

(b) 三叉链表

1.1 利用空指针

发现：
n个节点，$2n$个pointer，$n-1$ 有用， $n+1$ 没用。
可以通过增加标识，利用好这些pointer。

Solutions:

1. ThreadBT.java

   as known as TBT.

1.2 三叉链表

Solutions:

1. interface List.java

```
1 public interface List<Item>
```

6.Huffman tree

(a) 递归定义及基本术语

(b) 分类顺序存储链式存储

(c) 应用*4

definition

Solutions:

1. a recursive definition

category

Solutions:

1. sequential structure
2. list structure

properties quality character

Solutions:

1. i & 2^{i-1} floor
2. k & $2^k - 1$ all the tree
3. $n_0 = n_2 + 1$
4. height $= [\log_2 n] + 1$
5. structure of root, Lchild, Rchild

application

1. 编码 code
2. 压缩非等长
3. 判定

   一般的数据大多呈正态分布，Huffmantree可加快速度
4. 加密解密

   MD5 加密解密，2次异或

7.Binary searching/sorting Tree

search – 不可重复 sort – 可重复

BST A binary search tree (BST) is a binary tree where each node has a Comparable key (and an associated value) and satisfies the restriction that the key in any node is larger than the keys in all nodes in that node's left subtree and smaller than the keys in all nodes in that node's right subtree.

delete Deletion is quite a bit more complex, since when one removes an internal node, one can't just let its children fall off, but must re-attach them somewhere in the tree. Obviously, deletion of an external node is easy; just replace it with the null tree (see Figure 6.3(a)). It's also easy to remove an internal node that is missing one child—just have the other child commit patricide and move up (Figure 6.3(b)).

When neither child is empty, we can find the successor of the node we want to remove—the first node in the right tree, when it is traversed in inorder. Now that node will contain the smallest key in the right subtree. Furthermore, because it is the first node in inorder, its left child will be null [why?]. Therefore, we can replace that node with its right child and move its key to the node we are removing, as shown in Figure 6.3(c).

Solutions:

1. remove

```
    /** Delete the instance of key from Node that is closest
    *  to the root and return the modified tree. The nodes of
    *  the original tree may be modified. */
    public Node removehelper(Node node, K key) {
        if (node == null) {
            return null;
        }
        if (key.compareTo(node.key) < 0) {
            node.left = removehelper(node.left, key);
        } else if (key.compareTo(node.key) > 0) {
            node.right = removehelper(node.right, key);
        } else if (node.left == null) {
            // Otherwise, we've found key
            return node.right;
        } else if (node.right == null) {
            return node.left;
        } else {
            node.right = swapSmallest(node.right, node);
        }
        return node;
    }
    /** Move the label from the first node in node (in an inorder
    *  traversal) to node R (over-writing the current label of R),
```

```
24      *  remove the first node of node from node, and return the resulting
            tree. */
25      private Node swapSmallest(Node node, Node R) {
26          if (node.left == null) {
27              R.key = node.key;
28              return node.right;
29          } else {
30              node.left = swapSmallest(node.left, R);
31              return node;
32          }
33      }
```

Shape of Tree	Height	add	search
“Bushy” (Best Case)	$\log N$	$(logN)$	$(logN)$
“Spindly” (Worst Case)	N	(N)	(N)

最大值最小值中序遍历有序

平均查找长度ASL insert create

8.BalancedBinarySortingTree categories

2-3 Tree BST where a node can have 2,3 children Ensures tree is “bushy” -¿ height is log(n)

2-3 and 2-3-4 trees. Understand how to search and insert. Understand why insertion technique leads to perfect balance.

Terminology. Know that a 2-3 tree is also called a B-Tree of order 3 and that a 2-3-4 tree is a B-Tree of order 4. Know that an M-node is a node with M children, e.g. a 4-node is a node containing 3 items and thus with 4 children.

Performance of trees. Tree height is between $log_M N$ and $log_2 N$. All paths are of the same height.

1-1 correspondence between LLRBs and 2-3 trees. Understand how to map a 2-3 tree to an LLRB and vice versa. You do not need to know how to perform tree operations ‘natively’ on an LLRB.

LLRBs. BST such that no node has two red links touching it; perfect black balance; red links lean left.

Left Leaning Red Black Tree

This is simply an implementation of a 2-3 Tree with the same ideas. Be able to convert between a 2-3 Tree and a LLRB tree. We use red links to indicate two nodes that would be in the same 2-3 Node. In a left leaning RB tree, we arbitrarily enforce that edges are always to the left (for convenience).

Properties of LLRB’s Here are the properties of LLRB’s:

1-1 correspondence with 2-3 trees.

No node has 2 red links.

There are no red right-links.

Every path from root to leaf has same number of black links (because 2-3 trees have same number of links to every leaf).

Height is no more than 2x height of corresponding 2-3 tree.

LLRB get Exactly the same as regular BST get.

LLRB performance. Perfect black balance ensures worst case performance for get and insert is $2\ log_2 N$.

LLRB insert. You are not responsible for knowing how to insert into an LLRB using rotations and color flipping. However, you should know how to convert back and forth between 2-3 trees and LLRBs, and you should also know how to insert into 2-3 trees. Thus, you should know effectively how one can insert into an LLRB.

LLRB Tree Properties of a 2-3 tree, but implemented like a BST

Tree rotaions We rotateLeft or rotateRight on a node, creating a different but valid BST with the same elements. Notice when we rotateLeft(G) we move the node G to be the left child of the new root.

There are two important propertires for LLRBs:

No node ever has 2 red links (It wouldn't be a valid node in a 2-3 Tree if it did) Every path from the root to a leaf has the same number of black links. This is because every leaf in a 2-3 tree has same numbers of links from root. Therefore, the tree is balanced. LLRB operations Always insert with a red link at the correct location. Then use the following three operations to "fix" or LLRB tree. See slides for visual.

If there is a right leaning red link, rotate that node left. If there are two consecutive left leaning links, rotate right on the top node. If there is a node with two red links to children, flip all links with that node.

9.BinaryTree categories

10.heap

Priority Queue. A Max Priority Queue (or PQ for short) is an ADT that supports at least the insert and delete-max operations. A MinPQ supposert insert and delete-min.

Tree Representations. Know that there are many ways to represent a tree, and that we use Approach 3b (see lecture slides) for representing heaps, since we know they are complete.

Running times of various PQ implementations. Know the running time of the three primary PQ operations for an unordered array, ordered array, and heap implementation.

Heap definitions The binary heap is a data structure that can efficiently support the basic priority-queue operations. In a binary heap, the keys are stored in an array such that each key is guaranteed to be larger than (or equal to) the keys at two other specific positions. In turn, each of those keys must be larger than (or equal to) two ad- ditional keys, and so forth. This ordering is easy to see if we view the keys as being in a binary tree structure with edges from each key to the two keys known to be smaller.

Heaps. A max (min) heap is an array representation of a binary tree such that every node is larger (smaller) than all of its children. This definition naturally applies recursively, i.e. a heap of height 5 is composed of two heaps of height 4 plus a parent.

11.Tree and forest

parent

child

child sibling

.BinaryTree in java