本页PDF

树

树的节点代表集合，树的边代表关系

广度优先遍历

- 也叫做层序遍历，一层一层来遍历树
- 使用队列来进行遍历，遍历完当前节点的子节点就弹出该节点

```
void bfs(Node *root) //广度优先遍历
{
    head = tail = 0;
    Queue[tail++] = root;
    while(head<tail)
    {
        Node *node = Queue[head];
        cout << node->key << endl;
        if(node->lchild) Queue[tail++] = node->lchild;
        if(node->rchild) Queue[tail++] = node->rchild;
        head++;
    }
    return;
}
```

深度优先遍历

- 使用栈来实现遍历
- 先左后右
- 判断栈顶元素是否有子节点，有子节点则入栈，无子节点则出栈，直到变为空栈

```
void dfs(Node *root) //用栈模拟
{
    if(root == NULL) return;
    int start,end;
    tot += 1;
    start = tot;
    if(root->lchild) dfs(root->lchild); //递归，调用系统栈
    if(root->rchild) dfs(root->rchild); //调用一次函数实质就是入一次栈
    tot += 1;                           //函数返回实质就是出栈
```

```
        end = tot;
        cout << root->key << endl;
        return;
    }
```

二叉树性质

- 度为0的节点比度为2的节点多一个
- 种类
  - 完全二叉树：只有最后一层缺少右侧节点
    a. 对于编号为i的节点
      - 该节点左节点：2*i
      - 该节点右节点：2*i+1

    b. 其编号连续，可以用连续的数组存储
  - 满二叉树：没有度为1的节点
  - 完美二叉树：每一层都是满的
- 遍历

前序遍历	中序遍历	后序遍历
根左右	左根右	左右根

- 实现树的序列化
- 前加后不能还原，因为不能判断有多少节点

二叉树线索化

有利于让二叉树的遍历方式由递归变为非递归 本质上是利用冗余的指针空间

- 左指针指向前驱
- 右指针指向后继
  - 前驱指的是在相应遍历方式下某节点的前一个节点
  - 后继指的是在相应遍历方式下某节点的后一个节点

```
#include<bits/stdc++.h>
using namespace std;
void build_inorder_thread(Node *root)
{
    if(root == NULL) return ;
    if(root->ltag == 0) build_inorder_thread(root->lchild);
```

```
        if(inorder_root == NULL) inorder_root = root; //root
        是中序遍历的第一个节点，赋值给inorder_root
        if(root->lchild == NULL) //前驱
        {
            root->lchild = prenode;
            root->ltag = 1;
        }
        if(prenode && prenode->rchild == NULL) //后继
        {
            prenode->rchild = root;
            prenode->rtag = 1;
        }
        prenode = root;  //更新prenode，让其指向当前已经处理完
        毕的节点
        if(root->rtag == 0) build_inorder_thread(root->rchild);
        return;
    }

    void __build_inorder_thread(Node *root)
    {
        build_inorder_thread(root);
        prenode->rchild = NULL; //处理完前面所有节点后，prenode指向最后一
        prenode->rtag = 1;
        return;
    }

    Node *getnext(Node *root)
    {
    if(root->rtag == 1) return root->rchild; //要注意的是，线
    索化的实现是按照中序遍历的顺序来的
        root = root->rchild;          //代码到这一行说明root的
        后继是一条实实在在的边，而在中序遍历中，当前节点的后继是这
        个节点右子树的最左边的节点，因此使用循环遍历
        while(root->ltag == 0 && root->lchild) //当root->tag
        为1时，就说明其没有左子树，因此其就是当前节点右子树的最左
        边的节点
        {
            root = root->lchild;
        }
        return root;
    }
    int main()
    {
        Node *node = inorder_root; //要指向中序遍历的第一个节点
        while(node)
        {
        cout << node->key << " ";
        node = getnext(node);
        }
```

```
            clear(root);
            return 0;
        }
```

使用的方法是站在每个节点的下一个节点去处理当前节点的后继 对于最后一个节点，由于其没有下一个节点，因此无法处理其后继

二叉树与广义表

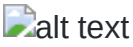
二叉树转广义表

```
#include<bits/stdc++.h>
using namespace std;
#define KEY(n) (n ? n->key : -1) //空地址返回负一

typedef struct Node
{
    int key;
    struct Node *lchild,*rchild;
}Node;

Node *getnewnode(int key)
{
    Node *p = new Node;
    p->key = key;
    p->lchild = p->rchild = NULL;
    return p;
}

void clear(Node *root)
{
    if(root == NULL)  return ;
    clear(root->lchild);
    clear(root->rchild);
    delete root;
    return ;
}

Node *insert(Node *root,int key)
{
    if(root == NULL) return getnewnode(key);
    if(rand()%2) root->lchild = insert(root->lchild,key);
    else root->rchild = insert(root->rchild,key);
    return root;
}

Node *getranderbinarytree(int n)
```

```
{
    Node *root =NULL;
    for(int i = 0;i < n;i++)
    {
        root = insert(root,rand() % 100);
    }
    return root;
}

char buff[1000];
int len = 0; //广义表信息长度

void __serialize(Node *root) //使用前序遍历序列化
{
    if(root == NULL) return;
    len += sprintf(buff + len ,"%d",root->key);//sprintf的返回值是输出的字符数
    if(root->lchild == NULL && root->rchild == NULL) return;
    len += sprintf(buff + len ,"(");
    __serialize(root->lchild);
    if(root->rchild)
    {
        len += sprintf(buff + len ,",");
        __serialize(root->rchild);
    }
    len += sprintf(buff + len ,")");
    return ;
}

void serialize(Node *root)
{
    memset(buff,0,sizeof(buff));
    len = 0;
    __serialize(root);
    return;
}


void print(Node *node)
{
    printf("%d(%d,%d)\n",KEY(node),KEY(node->lchild),KEY(node->rchild));
    return ;
}

void output(Node *root)
{
    if(root == NULL) return;
    print(root);
    output(root->lchild);
    output(root->rchild);
```

```
    return;
}
int main()
{
    srand((unsigned)time(NULL));
    #define n 10
    Node *root = getranderbinarytree(n);
    serialize(root);
    output(root);
    cout << buff << " " <<"广义表";
    clear(root);
    return 0;
}
```

广义表转二叉树

 alt text

- 运用栈的思想
    - 遇到关键字，生成节点
    - 碰到左括号入栈
    - 碰到逗号标记flag为1（flag为0代表左子树，为1代表右子树）
    - 碰到右括号弹栈
- 设立左右子树时是为栈顶元素设立的 **这道题使用状态机的算法思想，使用对应数字来分配任务 但分配完任务i须减1，同样地，完成任务后scode要设置为0**

```
#include<bits/stdc++.h>
using namespace std;
#define KEY(n) (n ? n->key : -1) //空地址返回负一

typedef struct Node
{
    int key;
    struct Node *lchild,*rchild;
}Node;

Node *getnewnode(int key)
{
    Node *p = new Node;
    p->key = key;
    p->lchild = p->rchild = NULL;
    return p;
}
```

```
void clear(Node *root)
{
    if(root == NULL)  return ;
    clear(root->lchild);
    clear(root->rchild);
    delete root;
    return ;
}

Node *insert(Node *root,int key)
{
    if(root == NULL) return getnewnode(key);
    if(rand()%2) root->lchild = insert(root->lchild,key);
    else root->rchild = insert(root->rchild,key);
    return root;
}

Node *getranderbinarytree(int n)
{
    Node *root =NULL;
    for(int i = 0;i < n;i++)
    {
        root = insert(root,rand() % 100);
    }
    return root;
}
char buff[1000];
int len = 0; //广义表信息长度

void __serialize(Node *root) //使用前序遍历序列化
{
    if(root == NULL) return;
    len += sprintf(buff + len ,"%d",root->key);//sprintf的返回值是输出的字
    if(root->lchild == NULL && root->rchild == NULL) return;
    len += sprintf(buff + len ,"(");
    __serialize(root->lchild);
    if(root->rchild)
    {
        len += sprintf(buff + len ,",");
        __serialize(root->rchild);
    }
    len += sprintf(buff + len ,")");
    return ;
}



void serialize(Node *root)
{
```

```
    memset(buff,0,sizeof(buff));
    len = 0;
    __serialize(root);
    return;
}


Node *deserialize(char *buff,int n)
{
    Node **s = (Node **)malloc(sizeof(Node *) * 100);
    int top = -1,flag = 0,scode = 0; //这里使用了状态机的算法思维
    Node *p = NULL , *root =NULL;
    for(int i = 0 ; buff[i];i++)
    {
        switch(scode)
        {
        case 0:
            {
            if(buff[i] >= '0' && buff[i] <= '9') scode = 1;
            else if(buff[i] == '(') scode = 2;
            else if(buff[i] == ',') scode = 3;
            else scode = 4;
            i -= 1;
            }
            break;
        case 1:
            {
            int num = 0;
            while(buff[i] <= '9' && buff[i] >= '0')
            {
                num = num * 10 +(buff[i] - '0');
                i += 1;
            }
            p = getnewnode(num);
            if(top >= 0 && flag == 0) s[top]->lchild = p;
            if(top >= 0 && flag == 1) s[top]->rchild = p;
            i -= 1; //得到num后，i会指向下一个位置，但外层循环有i+1，为抵消影响，
            scode = 0;
            }
            break;
        case 2:
            {
            s[++top] = p;
            flag = 0;
            scode = 0;
            }
            break;
        case 3:
            {
```

```
                flag = 1;
                scode = 0;
                }
                break;
            case 4:
                {
                root = s[top--];
                scode = 0;
                }
                break;
            }
        }
        return root;
    }

    void print(Node *node)
    {
        printf("%d(%d,%d)\n",KEY(node),KEY(node->lchild),KEY(node->rchild))
        return ;
    }

    void output(Node *root)
    {
        if(root == NULL) return;
        print(root);
        output(root->lchild);
        output(root->rchild);
        return;
    }
    int main()
    {
        srand((unsigned)time(NULL));
        #define n 10
        Node *root = getranderbinarytree(n);
        serialize(root);
        output(root);
        cout << buff << " " <<"广义表";
        Node *new_root = deserialize(buff,len);
        output(new_root);
        clear(root);
        return 0;
    }
```

哈夫曼编码

两个字符编码不能形成前缀关系

```
#include<bits/stdc++.h>
using namespace std;
typedef struct Node
{
    int freq;
    char ch;
    struct Node *lchild,*rchild;
}Node;

Node *getnewnode(int freq,char ch)
{
    Node *p = new Node;
    p->ch = ch;
    p->freq = freq;
    p->lchild = p->rchild = NULL;
    return p;
}


void swap_node(Node **node_arr,int i,int j)
{
    Node *temp = node_arr[i];
    node_arr[i] = node_arr[j];
    node_arr[j] = temp;
    return;
}

int find_min_node(Node **node_arr,int n)
{
    int ind = 0;
        for(int j = 1;j <= n;j++)
        {
            if(node_arr[ind]->freq > node_arr[j]->freq) ind = j;
        }
    return ind;
}
//重难点：哈夫曼树建立过程
Node *buildhaffmantree(Node **node_arr,int n)
{
    for(int i = 1;i < n;i++)
    {
        int ind1 = find_min_node(node_arr,n - i);
        swap_node(node_arr,ind1,n-i); //将最小值与当前最后一个节点交换位置
        int ind2 = find_min_node(node_arr,n - i - 1);
        swap_node(node_arr ,ind2,n - i - 1);
        int freq = node_arr[n - i]->freq + node_arr[n - i - 1]->freq;
        Node *node = getnewnode(freq , 0);
        node->lchild = node_arr[n - i];
```

```
        node->rchild = node_arr[n- i - 1];
        node_arr[n - i - 1] = node;
    }
    return node_arr[0];
}

void extracthaffmancode(Node *root ,char buff[],int k)
{
    buff[k] = 0;
    if(root->lchild == NULL && root->rchild == NULL)
    {
        cout << root->ch << buff << endl;
        return ;
    }
    buff[k] = '0';
    extracthaffmancode(root->lchild,buff,k+1);
    buff[k] = '1';
    extracthaffmancode(root->rchild,buff,k+1);
    return ;
}
void clear(Node *root)
{
    if(root = NULL) return ;
    clear(root->lchild);
    clear(root->rchild);
    delete root;
    return ;
}


int main()
{
    int n,freq;
    char s[10];
    cin >> n;
    Node **node_arr = new Node *[n];
    for(int i = 0;i < n;i++)
    {
        cin >> s >> freq;
        node_arr[i] = getnewnode(freq,s[0]);
    }
    Node *root = buildhaffmantree(node_arr,n);
    char buff[1000];
    extracthaffmancode(root,buff,0);
    clear(root);
    return 0;
}
```

n叉树前序遍历

 alt text /* // Definition for a Node. class Node { public: int val; vector<Node*> children;

```
    Node() {}

    Node(int _val) {
        val = _val;
    }

    Node(int _val, vector<Node*> _children) {
        val = _val;
        children = _children;
    }
};
*/

class Solution {
public:
    vector<int> preorder(Node* root) {
    if(root == NULL) return vector<int>();
    vector<int> ans;
    ans.push_back(root->val);
    for(auto x: root->children)
    {
        vector<int> temp = preorder(x);
        for(auto y : temp) ans.push_back(y);
    }
    return ans;
    }
};
```

从前序与中序遍历序列构造二叉树

 alt text

- 使用递归思想
- 对大树使用前序与中序遍历结果恢复二叉树
    - 对左右子树也分别用前中序遍历结果恢复
    - 返回根节点

```
class Solution {
public:
```

```
        TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder)
        if(preorder.size() == 0) return NULL;
        int pos = 0;
        while(inorder[pos] != preorder[0]) pos += 1;
        TreeNode *root = new TreeNode(preorder[0]);
        vector<int> pre , in;
        for(int i = 1;i <= pos;i++) pre.push_back(preorder[i]);
        for(int i = 0;i <= pos -1;i++) in.push_back(inorder[i]);
        root->left = buildTree(pre,in);
        pre.clear();
        in.clear();
        for(int i = pos + 1;i < preorder.size();i++)
        {
        pre.push_back(preorder[i]);
        in.push_back(inorder[i]);
        }
        root->right = buildTree(pre,in);
        return root;
        }
    };
```

二叉树的层序遍历

 alt text

广搜

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
    if(root == NULL) return vector<vector<int>>();
    TreeNode *node;
    queue<TreeNode *> q;
    q.push(root);
    vector<vector<int>> ans;
    while(!q.empty())
    {
        int cnt = q.size();
        vector<int> temp;
        for(int i = 0;i < cnt;i++)
        {
            node = q.front();
            temp.push_back(node->val);
            if(node->left) q.push(node->left);
            if(node->right) q.push(node->right);
            q.pop();
        }
```

```
            ans.push_back(temp);
        }
        return ans;
    }
    };
```

深搜

- 深搜的精髓是要找到当前节点的层数，并在相应二维数组的一维数组的位置插入数值
- 要记得在相应层数扩充一维数组

```
    class Solution {
        void dfs(TreeNode *root,int k,vector<vector<int>> &ans)
        {
            if(root == NULL) return ;
            if(k == ans.size()) ans.push_back(vector<int>());
            ans[k].push_back(root->val);
            dfs(root->left,k+1,ans);
            dfs(root->right,k+1,ans);
            return ;
        }
    public:
        vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> ans;
        dfs(root,0,ans);
        return ans;
    }
    };
```

翻转二叉树

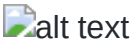
 alt text

- 利用递归思想 解决完当前根节点的子节点 就解决子节点的子节点
- 可以利用c++里面的swap函数，不需要手写交换函数

```
    class Solution {
    public:
        TreeNode* invertTree(TreeNode* root) {
            if(root == NULL) return NULL;
            swap(root->right,root->left);
            invertTree(root->left);
            invertTree(root->right);
            return root;
```

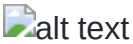
```
        }
    };
```

二叉树层序遍历



```
class Solution {
public:
    void dfs(TreeNode *root,int k,vector<vector<int>> &ans)
    {
        if(root == NULL) return;
        if(k == ans.size()) ans.push_back(vector<int>());
        ans[k].push_back(root->val);
        dfs(root->left,k+1,ans);
        dfs(root->right,k+1,ans);
        return ;
    }
    vector<vector<int>> levelOrderBottom(TreeNode* root){
        vector<vector<int>> ans;
        dfs(root,0,ans);
        for(int i = 0,j = ans.size()-1;i < j;i++,j--)
        {
            swap(ans[i],ans[j]);
        }
    return ans;
    }
};
```

二叉树的锯齿形层序遍历



```
class Solution {
public:
    void dfs(TreeNode *root,int k,vector<vector<int>> &ans)
        {
        if(root == NULL) return;
        if(k == ans.size()) ans.push_back(vector<int>());
        ans[k].push_back(root->val);
        dfs(root->left,k+1,ans);
        dfs(root->right,k+1,ans);
        return ;
        }
    vector<vector<int>> zigzagLevelOrder(TreeNode* root){
        vector<vector<int>> ans;
```

```
            dfs(root,0,ans);
            for(int k = 1;k < ans.size();k+=2)
            {
                for(int i = 0,j = ans[k].size()-1;i < j;i++,j--)
                swap(ans[k][i],ans[k][j]);
            }
            return ans;
        }
    };
```

合并果子

 alt text **哈夫曼编码的应用**  alt text

- 最优体力是堆的数量乘以路径长度（个人理解是层数）
- 与哈夫曼算法本质是一样的
- 这里用到了set类模板 要使用pair才可以存储键值对
- 先找出最小的，将其删除，再找出次小的，再删除，然后将这两堆合并，合并次数为n-1次

```
#include<bits/stdc++.h> //合并果子 海贼oj287
using namespace std;
typedef pair<int,int> PII;
int main()
{
    int n;
    set<PII> s;
    cin >> n;
    for(int i = 0,a;i < n;i++)
    {
        cin >> a;
        s.insert(PII(a,i));
    }
    int ans = 0;
    for(int i = 1;i < n;i++)
    {
        int a = s.begin()->first;
        s.erase(s.begin());
        int b = s.begin()->first;
        s.erase(s.begin());
        ans += a+b;
        s.insert(PII(a + b,n + i));
    }
    cout << ans;
```

```
        return 0;
    }
```

货仓选址

 alt text **分析**

> 设货仓建在x坐标，x左侧商店有P家，x右侧商店有Q家。若P < Q,则每把货仓的选址向右移动1单位距离，距离之和就会变小Q - P(左侧的P家店到x的距离增加P，因为每家店到x的距离都增加了1单位距离，右侧的Q家店到x的距离减少Q，而P < Q,因此距离减少了Q - P)，若P > Q ,同理可证明距离增加了P - Q 因此当货仓在所有位置的中位数时，距离之和最小

```
#include<bits/stdc++.h> //海贼oj245 货仓选址
using namespace std;
int main()
{
    int n;
    vector<int> arr;
    cin >> n;
    for(int i = 0,a;i < n;i++)
    {
        cin >> a;
        arr.push_back(a);
    }
    sort(arr.begin(),arr.end());
    int p = arr[n/2],ans = 0;//n不论奇偶n/2都是中位数
    for(int i = 0;i < n;i++)
    {
        ans += abs(arr[i] - p);
    }
    cout << ans;
    return 0;
}
```