1.树的前序遍历

```
void visitAlongLeftBranch(treeNode *node, stack<treeNode*>&s){//沿着左链访问
    treeNode *leftNode=node;
    while(leftNode != NULL){
        visit(leftNode);
        s.push(leftNode->right);//左链结点依次入栈 ，不必排除空右子树
        leftNode = leftNode->left;
    }
}
void preOder2(treeNode *root){
    stack<treeNode*>s;
    treeNode*childN = root; //整个树
    while(true){
        visitAlongLeftBranch(childN, s); //例程
        if(s.empty()) break;
        childN = s.top();  s.pop(); //子树
    }
}
```

2.树的中序遍历

```
void goAlongLeftBranck(treeNode* node, stack<treeNode*>&s){
    treeNode*curr = node;
    while(curr != NULL){ s.push(curr); curr = curr->left;}
}
void midOder(treeNode*root){
    treeNode*childN = root;
    stack<treeNode*>s;
    while(true){
        goAlongLeftBranck(childN, s);  //从当前结点出发 当前结点作为独立的一颗子树 批次
入栈
        if(s.empty()) break;
        visit(s.top()); childN = (s.top())->right;//访问最左结点，递交访问控制权至右子
树
        s.pop();
    }
}
```

3.树的后序遍历

```
// 判断两个结点是否为兄弟关系 是 返回true 否 返回false
#define rightBrother(a, b) ( ((b->left) == (a) || ((b->right) == (a)))  ? false
:true  )

void gotoLeftLeaf(stack<TN*>&s){  // 最高左侧可见叶结点
    TN *t;
    while(t = s.top()){  //终止条件是 压入NULL 栈顶为空
        /*尽量往左边走*/
        if(t->left != NULL){  // 左子树存在
            if(t->right != NULL)s.push(t->right);  // 先压入右子树
            s.push(t->left);  // 在压入左子树
        }else{
            s.push(t->right);  //如果左子树为空 简明的压入右子树
```

```
        }
    }
    s.pop();  //pop掉栈顶的NULL
}
void traversePost(TN *root){
    stack<TN*>s;  //辅助栈 模拟 后序遍历过程：入栈顺序：尽量往左侧链走， 子树根结点->其右子树->左子树
    s.push(root);
    TN *node = root;  // 栈上次弹出的结点  为了判断当前栈顶结点是父结点还是右子树
    while(!s.empty()){
        if(rightBrother(node, s.top())){  // 栈顶结点和之前弹出结点成右兄关系
            gotoLeftLeaf(s);
        }
        // 如果有右兄子树 已经把他入栈了
        node = s.top(); s.pop(); cout<<node->val<<" ";  // 弹出栈顶，简明的访问
    }
}
```

4.归并排序

```
void merge(vector<int>&v, int lo, int mid, int hi){
    int lenLeft = mid - lo;
    vector<T>B;
    B.assign(v.begin()+lo, v.begin()+lo+mid);
    for(int idx_a = lo,idx_b = 0, idx_c = mid; idx_b < lenLeft;){ // 这里不能判断idx_c是否越界
        //前半段归入的条件是：前半段元素小或者后半段已经遍历完（idx_c越界）
        v[idx_a++] = (idx_c >= hi || B[idx_b] < v[idx_c]) ? B[idx_b++] : v[idx_c++]; // 这里判断idx_c是否越界
    }
    /*A:[lo,              hi)
        B:[lo,    mid)
                C:[mid,  hi)
    如果B已经遍历完了，C还未完成的部分本身就在A中，无需遍历C，归并结束
    如果C已经遍历完了，需要把B中剩余的元素拷贝到A中相应的部分。
    */
}
void mergeSort(vector<int> &v, int lo, int hi){
    if(hi - lo < 2)return; // 单元素自然有序
    int mid = (lo + hi) >> 1;  // 切分
    mergeSort(v, lo, mid); // 使前半部分有序
    mergeSort(v, mid, hi); // 使后半部分有序
    merge(v, lo, mid, hi); // 合并前后两个有序的部分
}
```

5.快排

```
int my_partition_LUG(vector<int> &v, int lo, int hi){
    cout<<"use LUG algorithm "<<endl;
    swap(v[lo], v[rand() % (hi-lo) + lo]); // 随机取得轴点
    int pivot = v[lo]; hi--;
    // 拓展G ----> 拓展L ----> 拓展G ----> 拓展L......
    while(lo < hi){
        while(lo < hi && pivot <= v[hi]) {
            --hi;   //向左拓展G
        }
        v[lo] = v[hi];  // 小于轴点皆归入L
```

```
        while(lo < hi && v[lo] <= pivot){  //向右拓展L
            ++lo;
        }
        v[hi] = v[lo];// 大于轴点者皆归入G
    }
    v[lo] = pivot;
    return lo;
}
void quickSort(vector<int> &v, int lo, int hi){
    if(hi - lo < 2) return ;
    int mi = my_partition_LUG(v, lo, hi); // LUG 培养轴点
    quickSort(v, lo, mi);
    quickSort(v, mi + 1, hi);
}
```

6.希尔排序

```
void shellSort(vector<int> &v, int lo, int hi){
    /*
            ...  i -d ...
            ...  i    ...
    */
    //采用ps序列{1，3，5，7，...,1073741823,...}
    for(int d = 0x3FFFFFFF;d>=1;d >>= 1){
        /*按照矩阵行分别对每列的前i行进行插入排序,实质上就是从序列lo+d处开始到hi-1*/
        for(int i = lo + d; i < hi;i++){
            T x = v[i]; // 备份待插入的元素，插入排序需要依次移动元素
            int k = i - d; // 初始化插入的位置
            while( lo <= k && v[k] > x){
                v[k+d] = v[k];
                k -= d;
            }
            v[k + d] = x;  //插入待排序的元素
        }
    }
}
```

7.冒泡排序

```
void bubbleSort(vector<int> &v, int lo, int hi){
    int lastswap = --hi; //需要扫描的区间[lo,lastswap);初始化
    while(lo < lastswap){
        hi = lastswap;  //得到上次扫描最后的位置
        lastswap = lo;  //初始化，如果本次扫描没有交换，则向量有序，循环退出
        for(int i = lo;i < hi; i ++ ){
            if(v[i] > v[i+1]){
                lastswap = i; // 动态记录最后一次扫描位置
                swap(v[i], v[i+1]);
            }
        }
    }
}
```

8.堆排序

```
/*一些堆的宏*/
```

```
// i处元素的左/右孩子元素 的秩
#define LC(i)           (1+((i)<<1))
#define RC(i)           ((1 + (i))<<1)
// 判断大小为n的堆，某秩x是否合法
#define ISINHEAP(n, x)  (  (   (-1) < (x)   )   &&  (   (x) < (n)  )    )
// #define ISINHEAP(n, i)  ( ( ( -1 ) < ( i ) ) && ( ( i ) < ( n ) ) )
// 判断某节点是否有左/右孩子
#define ISHAVELC(n, i)  (  ISINHEAP(n, LC(i) )  )
#define ISHAVERC(n, i)  (  ISINHEAP(n, RC(i) )  )
#define MAXOFTWOINDEX(v, i, j)  ( ( (v[i]) < (v[j]) ) ? (j) : (i) )
#define MINOFTWOINDEX(v, i, j)  ( ( (v[i]) <= (v[j]) ) ? (i) : (j) )
// 找到父子节点最大的节点在向量中的秩 vector size i  对于堆来说存在右子树存在左子树一定存在
#define ProperParentBigHP(v, n, i)   (  ISHAVERC(n, i) ? MAXOFTWOINDEX( v,
MAXOFTWOINDEX(v, i, LC(i)) , RC(i)) :\
 (  ISHAVELC(n, i) ? MAXOFTWOINDEX( v, i, LC(i)) : (i) )\
                                )
#define ProperParentSmallHP(v, n, i) ( ISHAVERC(n,i) ? MINOFTWOINDEX(v,
MINOFTWOINDEX(v, i, LC(i)), RC(i) ) :\
( ISHAVELC(n, i) ? MINOFTWOINDEX( v, i, LC(i)) : (i) )\
                                                )
void heapify(vector<int>&v, const int len, bool big_heap){ // Floyd建堆算法，O(n)
时间
    for( int index = len / 2 - 1;0 <= index;index--){ //从最后一个内部节点
        percolateDown(v, len, index, big_heap ? true : false);  // 依次下滤
    }
}
/*堆排序*/
void heapSort(vector<int> &v, bool ascend){
    int unsortedLen = v.size();
    heapify(v, unsortedLen, ascend ? true:false); // 建堆 大顶堆和小顶堆
    while(0 < unsortedLen--){
        swap(v[0], v[unsortedLen]); // 选最大的放到有序的右边部分
        percolateDown(v, unsortedLen, 0, ascend?true:false); // 对交换上去堆顶再进
行下滤
    }
}
```

9.快速划分的k-selection

```
int quick_selection(){
    int i , j;
    for(int lo = 0, hi = v.size() - 1;lo < hi;){
        i = lo; j = hi; int pivot = v[lo];
        while( i < j){ //  维护 LUG 。使得轴点就为
            while(i<j&&pivot<=v[j]){j--;}v[i] = v[j];
            while(i<j&&v[i]<=pivot){i++;}v[j] = v[i];
        }
        v[i] = pivot;
        if(k == i){
            return v[k];
        }else if(k < i){
            hi = i-1;
        }else{
            lo = i + 1;
        }
    }
}
```

10.图的dfs（岛屿问题）

```
#include "./Graph.h"
#include <iostream>
#include <vector>
#include <utility>
#include <unordered_map>
#include <list>
using namespace std;
int row,col;
enum state{M_UNVISITED,M_VISITED,M_CHANGE};
struct point{
    int x, y;
    state s;
    point():x(-1), y(-1), s(M_UNVISITED){}
};
using z_chain = vector<pair<int,int>>;
#define index_leg(i, j) ( ( (i > 0) && (i < row) && (j > 0) && (j < col) )   ?
(true):(false))

#define inboundary(i, j)  (  (((i) == (row - 1))||((j) == (col - 1))) ? (true) :
(false))

// 找一个未被访问过的符合条件 的邻接结点
pair<int, int> randAdj(vector<vector<char>>&v,vector<vector<point>>&flag, int i,
int j){
    if(index_leg(i-1,j) && v[i-1][j] == 'z' && flag[i-1][j].s == M_UNVISITED){
        return {i-1,j};
    }
    if(index_leg(i+1,j) && v[i+1][j] == 'z' && flag[i+1][j].s == M_UNVISITED){
        return {i+1,j};
    }
    if(index_leg(i,j-1) && v[i][j-1] == 'z' && flag[i][j-1].s == M_UNVISITED){
        return {i,j-1};
    }
    if(index_leg(i,j+1) && v[i][j+1] == 'z' && flag[i][j+1].s == M_UNVISITED){
        return {i,j+1};
    }
    return {-1,-1};
}

void dfs(vector<vector<char>>&v, int r, int c, vector<vector<point>>&flag,
z_chain &zc, bool &zc_okay){
    if(flag[r][c].s == M_VISITED)return ;
    //if(!index_leg(r,c))return ;
    flag[r][c].s = M_VISITED;  //当前结点标记为访问
    cout<<r<<" "<<c<<endl;
    zc.push_back({r,c});
    if(inboundary(r,c)){  //在边界上
        zc_okay = false;
    }
    for(pair<int, int> adj_index = randAdj(v, flag, r,c);
        index_leg(adj_index.first, adj_index.second);
        adj_index = randAdj(v, flag, adj_index.first, adj_index.second))
    {
        dfs(v, adj_index.first, adj_index.second, flag, zc, zc_okay);
    }
```

```
}
int main(){
    //替换z变为x，与边界连接的z不能被替换。岛屿问题。

    vector<vector<char>>v = {
                              {'x','x','x','x','x'},
                              {'x','z','z','z','x'},
                              {'x','x','z','x','x'},
                              {'x','z','x','z','x'},
                              {'x','x','z','z','x'},
                              {'x','z','z','z','x'},
                              {'x','z','x','z','x'},
                              {'x','z','x','x','x'}
                             };
    row = v.size();
    col = v[0].size();
    list<z_chain>res;
    vector<vector<point>>flags(row, vector<point>(col, point()));
    for(int i = 0; i < row;i++){
        for(int j = 0; j < col;j++){
            if(v[i][j] == 'x' || flags[i][j].s == M_VISITED)continue;
            z_chain t; // 以v[i][j]为起点进行dfs搜索 结点记录在t里面
            bool zc_is_ok = true;
            dfs(v, i, j, flags, t, zc_is_ok);
            if(zc_is_ok){
                res.push_back(t);
            }
        }
    }
    // 遍历res  获得结果
    for(auto e:res){
        for(auto ee: e){
            v[ee.first][ee.second] = 'x';
        }
    }
    for(int i = 0; i < row; i++){
        for(int j = 0; j < col;j++){
            cout<<v[i][j]<<" ";
        }
        cout<<endl;
    }
    return 0;
}
```