CCF 编程能力等级认证
Grade Examination of Software Programming

C++ 六级

2024 年 12 月

1 单选题（每题 2 分，共 30 分）

题号	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
答案	D	C	C	B	B	B	A	C	A	B	B	D	C	B	C

第 1 题 面向对象编程(OOP)是一种特殊的程序设计方法。下面( )不是重要的OOP特性。

- ☐ **A.** 抽象
- ☐ **B.** 封装
- ☐ **C.** 继承
- ☐ **D.** 模块化

第 2 题 以下关于C++中类的说法，哪一项是正确的?

- ☐ **A.** 类中定义的所有成员变量和成员函数默认是 `public` 访问权限。
- ☐ **B.** 类的构造函数必须显式声明返回类型为 `void` 。
- ☐ **C.** 在C++中，类的数据一般设置为私有，其公有成员函数提供访问私有数据的唯一途径。
- ☐ **D.** 同一个类的实例有各自的成员数据和成员函数。

第 3 题 以下C++代码段中存在语法错误或逻辑错误，（ ）是正确的。

```cpp
#include <iostream>
using namespace std;
class MyClass {
public:
    MyClass() {
        cout << "Constructor called!" << endl;
    }
    void display() {
        cout << "Display function called!" << endl;
    }
};
int main() {
    MyClass* obj = NULL;
    obj->display();
    return 0;
}
```

- ☐ **A.** `NULL` 在C++中无法用于指针初始化，应使用 `nullptr` 。

- [ ] **B.** `obj` 的定义应该是 `MyClass obj;` 而不是指针类型。
- [ ] **C.** `obj->display()` 语句存在空指针访问错误，`obj` 应该初始化为一个有效的对象。
- [ ] **D.** `obj->display()` 语句会调用 `display()` 函数，但它没有输出任何内容。

第 4 题 阅读以下代码，下面哪一项是正确的？

```
void processData() {
    stack<int> s;
    queue<int> q;
    for (int i = 1; i <= 5; ++i) {
        s.push(i);
        q.push(i);
    }
    while (!s.empty()) {
        cout << "Stack pop: " << s.top() << endl;
        s.pop();
    }
    while (!q.empty()) {
        cout << "Queue pop: " << q.front() << endl;
        q.pop();
    }
}
```

- [ ] **A.** 栈 `s` 的输出顺序是 `1 2 3 4 5`，队列 `q` 的输出顺序是 `5 4 3 2 1`。
- [ ] **B.** 栈 `s` 的输出顺序是 `5 4 3 2 1`，队列 `q` 的输出顺序是 `1 2 3 4 5`。
- [ ] **C.** 栈 `s` 的输出顺序是 `1 2 3 4 5`，队列 `q` 的输出顺序是 `1 2 3 4 5`。
- [ ] **D.** 栈 `s` 的输出顺序是 `1 2 3 4 5`，队列 `q` 的输出顺序是 `1 2 3 4 5`，程序不会正常执行。

第 5 题 N个节点的双向循环链，在其中查找某个节点的平均时间复杂度是( )。

- [ ] **A.** $O(1)$
- [ ] **B.** $O(N)$
- [ ] **C.** $O(\log N)$
- [ ] **D.** $O\left(N^3\right)$

第 6 题 以下关于树的说法，（ ）是正确的。

- [ ] **A.** A. 在一棵二叉树中，叶子结点的度一定是2。
- [ ] **B.** B. 满二叉树中每一层的结点数等于$O\left(2^{(层数-1)}\right)$。
- [ ] **C.** C. 在一棵树中，所有结点的度之和等于所有叶子结点的度之和。
- [ ] **D.** D. 一棵二叉树的先序遍历结果和中序遍历结果一定相同。

第 7 题 已知字符集 `{A, B, C, D}` 的出现频率如下表所示：

字符	频率
A	8
B	3
C	1
D	6

根据哈夫曼编码法，下面（ ）是正确的哈夫曼树。

☐ **A.**

```
      ABCD
      / \
     A   BCD
        /  \
       D   BC
           / \
          B   C
```

☐ **B.**

```
      ABCD
      / \
     A   BCD
        /  \
       B   CD
           / \
          C   D
```

☐ **C.**

```
      ABCD
      / \
     D   ABC
        /  \
       A   BC
           / \
          B   C
```

☐ **D.**

```
      ABCD
      / \
     C   ABC
        /  \
       B   AD
           / \
          A   D
```

第 8 题 上一题中各字符的哈夫曼编码是（ ）。

☐ **A.** `A: 0, B: 10, C: 110, D: 111`

☐ **B.** `A: 0, B: 10, C: 11, D: 10`

☐ **C.** `A: 0, B: 101, C: 100, D: 11`

- [ ] **D.** `A: 11, B: 10, C: 01, D: 00`

第 9 题 （)是3位格雷编码。

- [ ] **A.** 000 001 011 010 110 111 101 100
- [ ] **B.** 000 001 010 011 100 101 110 111
- [ ] **C.** 000 001 100 101 011 010 111 110
- [ ] **D.** 000 010 001 011 100 110 101 111

第 10 题 根据下面二叉树和给定的代码，

```
#include <iostream>
using namespace std;

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

TreeNode* search(TreeNode* root, int val) {
    cout << root->val << " ";
    if (root == NULL || root->val == val) return root;

    if (val < root->val)
        return search(root->left, val);
    else
        return search(root->right, val);
}
```

给定以下二叉搜索树，调用函数 `search(root,7)` 时，输出的结果是（ ）。

```
      5
     / \
    3   7
   / \ / \
  2  4 6  8
```

- [ ] **A.** 5 3 7
- [ ] **B.** 5 7
- [ ] **C.** 2 3 4 5 6 7
- [ ] **D.** 8 7

第 11 题 阅读以下二叉树的深度优先搜索算法，横线上应填写（ ）。

```
void dfs(TreeNode* root) {
    if (root == nullptr)
        return;

    stack<TreeNode*> s;
```

```
    s.push(root);
    while (!s.empty()) {
        ————————————————————— // 在此处填入代码
        cout << node->value << " ";

        if (node->right) s.push(node->right);
        if (node->left) s.push(node->left);
    }
}
```

- A. `TreeNode* node = s.top();`
- B. `TreeNode* node = s.top();  s.pop();`
- C. `TreeNode* node = s.front();`
- D. `TreeNode* node = s.front();  s.pop();`

第 12 题 阅读以下二叉树的广度优先搜索的代码，横线上应填写（ ）。

```
#include <queue>
void bfs(TreeNode* root) {
    if (root == NULL) return;

    queue<TreeNode*> q;
    q.push(root);
    while (!q.empty()) {
        ————————————————————— // 在此处填入代码
        cout << node->val << " ";
        if (node->left) {
            q.push(node->left);
        }
        if (node->right) {
            q.push(node->right);
        }
    }
}
```

- A. `TreeNode* node = q.top();`
- B. `TreeNode* node = q.top();  q.pop();`
- C. `TreeNode* node = q.front();`
- D. `TreeNode* node = q.front();  q.pop();`

第 13 题 使用上题中的宽度优先搜索算法遍历以下这棵树，可能的输出是( )。

```
        1
       / \
      2   3
     / \   \
    8  9    6
      / \    \
      4  5    7
```

- [ ] **A.** 1 2 8 9 4 5 3 6 7
- [ ] **B.** 1 2 3 4 5 6 6 8 9
- [ ] **C.** 1 2 3 8 9 6 4 5 7
- [ ] **D.** 8 4 5 9 2 1 3 6 7

第 14 题 以下关于动态规划的描述，（ ）是正确的。

- [ ] **A.** 动态规划适用于没有重叠子问题的优化问题。
- [ ] **B.** 动态规划要求问题具有最优子结构和无后效性。
- [ ] **C.** 动态规划通常通过递归来实现。
- [ ] **D.** 动态规划与贪心算法不同，贪心算法不适用于有重叠子问题的问题。

第 15 题 假设背包的最大容量$W = 8kg$，共有有4个物品可供选择，4个物品的重量分别为$weights = [2, 3, 5, 7]$，对应的价值分别为 $values = [30, 40, 60, 80]$，则该0/1背包问题中，背包的最大价值为（ ）。

- [ ] **A.** 70
- [ ] **B.** 90
- [ ] **C.** 100
- [ ] **D.** 120

2 判断题（每题 2 分，共 20 分）

题号	1	2	3	4	5	6	7	8	9	10
答案	√	×	√	√	×	×	√	√	√	×

第 1 题 构造函数是一种特殊的类成员函数，构造函数的名称和类名相同。但通过函数重载，可以创建多个同名的构造函数，条件是每个构造函数的参数列表不同。

第 2 题 类的静态成员函数既能访问类的静态数据成员，也能访问非静态数据成员。

第 3 题 栈中元素的插入和删除操作都在栈的顶端进行，所以方便用单向链表实现。

第 4 题 下面代码构建的树一定是完全二叉树：

```
struct TreeNode {
    int value;
    TreeNode* left;
    TreeNode* right;
};

TreeNode* buildCompleteBinaryTree() {
    TreeNode* root = new TreeNode{1};
    root->left = new TreeNode{2};
    root->right = new TreeNode{3};
    root->left->left = new TreeNode{4};
    root->left->right = new TreeNode{5};
    root->right->left = new TreeNode{6};
    return root;
```

```
15  }
```

第 5 题 在二叉排序树中，左子树所有节点的值都大于根节点的值，右子树所有节点的值都小于根节点的值。

第 6 题 在生成一个派生类的对象时，只调用派生类的构造函数。

第 7 题 下面的代码实现了二叉树的前序遍历，它通过递归方法访问每个节点并打印节点值。

```
1  void preorder(TreeNode* root) {
2      if (root == NULL) return;
3      cout << root->val << " ";
4      preorder(root->left);
5      preorder(root->right);
6  }
```

第 8 题 宽度优先搜索算法（BFS）保证了每个节点在最短路径的情况下被访问。

第 9 题 在解决简单背包问题时，动态规划的状态转移方程如下：

```
1  dp[i][w] = max(dp[i-1][w], dp[i-1][w - weights[i-1]] + values[i-1]);
```

该方程表示：在考虑第 `i` 个物品时，当前背包容量为 `w`，如果不放物品 `i`，则最大价值是 `dp[i-1][w]`；如果放入物品 `i`，则最大价值是 `dp[i-1][w - weights[i-1]] + values[i-1]`，其中数组 `weights` 和 `values` 分别表示所有物品的重量和价值，数组下标从 `0` 开始。

第 10 题 栈中元素的插入和删除操作都在栈的顶端进行，所以方便用双向链表比单向链表更合适表实现。

3　编程题（每题 25 分，共 50 分）

3.1　编程题 1

- **试题名称**：树上游走
- **时间限制**：1.0 s
- **内存限制**：512.0 MB

3.1.1　题面描述

小杨有一棵包含无穷节点的二叉树（即每个节点都有左儿子节点和右儿子节点；除根节点外，每个节点都有父节点），其中根节点的编号为 1，对于节点 i，其左儿子的编号为 $2 \times i$，右儿子的编号为 $2 \times i + 1$。

小杨会从节点 s 开始在二叉树上移动，每次移动为以下三种移动方式的任意一种：

- **第1种移动方式**：如果当前节点存在父亲节点，向上移动到当前节点的父亲节点，否则不移动；
- **第2种移动方式**：移动到当前节点的左儿子；
- **第3种移动方式**：移动到当前节点的右儿子。

小杨想知道移动 n 次后自己所处的节点编号。**数据保证最后的所处的节点编号不超过** 10^{12}。

3.1.2 输入格式

第一行包含一个正整数 n, s，代表移动次数和初始节点编号。

第二行包含一个长度为 n 且仅包含大写字母 U, L, R 的字符串，代表每次移动的方式，其中 U 代表第1种移动方式，L 代表第2种移动方式，R 代表第3种移动方式。

3.1.3 输出格式

输出一个正整数，代表最后所处的节点编号。

3.1.4 样例

```
3 2
URR
```

```
7
```

3.1.5 样例解释

小杨的移动路线为 2-1-3-7。

子任务编号	数据点占比	n	s
1	20%	≤ 10	≤ 2
2	20%	≤ 50	≤ 10
3	60%	$\leq 10^6$	$\leq 10^{12}$

对于全部数据，保证有 $1 \leq n \leq 10^6, 1 \leq s \leq 10^{12}$。

3.1.6 参考程序

```cpp
#include <bits/stdc++.h>
#define ll long long
using namespace std;
const ll INF = 1e12;
int n;
stack<char> st;
string c;
ll s;
int main(){
    cin >> n >> s >> c;
    for(int i = 0; i < n; i ++){
        if(c[i] == 'U'){
            if(s == 1) continue;
            if(st.size()){
                st.pop();
                continue;
            }
            s >>= 1;
        }else if(c[i] == 'L'){
            if((s << 1) > INF){
                st.push('L');
                continue;
            }
            s = s << 1;
        }else{
```

```
            if((s << 1 | 1) > INF){
                st.push('R');
                continue;
            }
            s = s << 1 | 1;
        }
    }
    cout << s << "\n";
    return 0;
}
```

3.2 编程题 2

- **试题名称**：运送物资
- **时间限制**：1.0 s
- **内存限制**：512.0 MB

3.2.1 题面描述

小杨管理着 m 辆货车，每辆货车每天需要向 A 市和 B 市运送若干次物资。小杨同时拥有 n 个运输站点，这些站点位于 A 市和 B 市之间。

每次运送物资时，货车从初始运输站点出发，前往 A 市或 B 市，之后返回初始运输站点。A 市、B 市和运输站点的位置可以视作数轴上的三个点，其中 A 市的坐标为 0，B 市的坐标为 x，运输站点的坐标为 p 且有 $0 < p < x$，货车每次去 A 市运送物资的总行驶路程为 $2p$，去 B 市运送物资的总行驶路程为 $2(x - p)$。

对于第 i 个运输站点，其位置为 p_i 且至多作为 c_i 辆车的初始运输站点。小杨想知道，在最优分配每辆货车的初始运输站点的情况下，所有货车每天的最短总行驶路程是多少。

3.2.2 输入格式

第一行包含三个正整数 n, m, x，代表运输站点数量，货车数量和两市距离。

之后 n 行，每行包含两个正整数 p_i, c_i，代表第 i 个运输站点的位置和最多容纳车辆数。

之后 m 行，每行包含两个正整数 a_i, b_i，代表第 i 辆货车每天需要向 A 市运送 a_i 次物资，向 B 市运送 b_i 次物资。

3.2.3 输出格式

输出一个正整数，代表所有货车每天的最短总行驶路程。

3.2.4 样例

```
3 4 10
1 1
2 1
8 3
5 3
7 2
9 0
1 10000
```

```
40186
```

3.2.5 样例解释

第 1 辆车的初始运输站点为站点 3，第 2 辆车的初始运输站点为站点 2。第 3 辆车的初始运输站点为站点 1，第 4 辆车的初始运输站点为站点 3。此时总行驶路程最短，为 40186。

子任务编号	数据点占比	n	m	c_i
1	20%	2	2	1
2	20%	$\leq 10^5$	$\leq 10^5$	1
3	60%	$\leq 10^5$	$\leq 10^5$	$\leq 10^5$

对于全部数据，保证有 $1 \leq n, m \leq 10^5, 2 \leq x \leq 10^8, 0 < p_i < x, 1 \leq c_i \leq 10^5, 0 \leq a_i, b_i \leq 10^5$。数据保证 $\sum c_i \geq m$。

3.2.6 参考程序

```cpp
#include<bits/stdc++.h>
using namespace std;
#define ll long long
const int N = 1e5+10;
int n,m;
ll x;
vector<pair<int,int> > st;
int a[N],b[N];
int main(){
    cin>>n>>m>>x;
    for(int i=1;i<=n;i++){
        int p,c;
        cin>>p>>c;
        st.push_back(make_pair(p,c));
    }
    sort(st.begin(),st.end());
    for(int i=1;i<=m;i++){
        cin>>a[i]>>b[i];
    }
    vector<pair<int,int> > neg,pos;
    ll res=0;
    for(int i=1;i<=m;i++){
        if(a[i]>=b[i])
            pos.push_back(make_pair(a[i]-b[i],i));
        else{
            neg.push_back(make_pair(a[i]-b[i],i));
        }
        res+=1ll*b[i]*x;
    }
    sort(neg.begin(),neg.end());
    sort(pos.begin(),pos.end());
    reverse(pos.begin(),pos.end());
    int l=0,r=n-1;
    for(auto i:neg){
        while(r>=1&&st[r].second==0)r--;
        res+=1ll*(a[i.second]-b[i.second])*st[r].first;
        st[r].second-=1;
    }
    for(auto i:pos){
        while(l<=n&&st[l].second==0)l++;
```

```
41          res+=1ll*(a[i.second]-b[i.second])*st[l].first;
42          st[l].second-=1;
43      }
44      cout<<res*2<<"\n";
45  }
```