**人工智能导论-数据结构C++和Python实现**

**1、队列**

1. **template**<**class** ElemType>
2. **struct** QNode
3. {
4. ElemType data;      // 数据
5. QNode \*next;        // 下一个节点
6. };
8. **template**<**class** ElemType>
9. **class** LinkQueue {
10. QNode<ElemType>\* front;       // 队头指针，始终指向队列头元素
11. QNode<ElemType>\* rear;        // 队尾指针，始终指向队列尾元素
12. **public**:
13. // 构造函数
14. LinkQueue()
15. {
16. front = **new** QNode<ElemType>;
17. **if** (!front)
18. exit(ERROR);
19. rear = front;
20. front->next = NULL;
21. }
22. // 析构函数
23. ~LinkQueue()
24. {
25. **while** (front)
26. {

**Python对应方法(在队列末尾添加一个元素)：**

**①Queue.put(e)**

**②deque.append(e)**

1. rear = front->next;
2. **delete** front;
3. front = rear;
4. }
5. }
6. // 在队列末尾插入一个元素
7. **int** push(ElemType e)
8. {
9. **if** (!front)
10. **return** ERROR;
11. QNode<ElemType> \*p;
12. p = **new** QNode<ElemType>;
13. **if** (!p)
14. **return** ERROR;
15. p->data = e;

**Python对应方法(访问队列头部第一个元素)：**

**①Queue.get()**

**②deque.popleft()**

1. p->next = NULL;
2. rear->next = p;
3. rear = p;
4. **return** OK;
5. }
6. // 访问队列的第一个元素
7. **int** getFront(ElemType &e)
8. {
9. **if** (front == rear)
10. **return** ERROR;
11. e = front->next->data;
12. **return** OK;

**Python对应方法(删除队列头部第一个元素)：**

**①Queue.pop()**

**②deque.popleft()**

1. }
2. // 删除队列的第一个元素
3. **int** pop()
4. {
5. **if** (front == rear)
6. **return** ERROR;
7. QNode<ElemType> \*p;
8. p = front->next;
9. front->next = p->next;
10. **if** (rear == p)
11. rear = front;
12. **else**
13. **delete** p;

**Python对应方法(判断队列是否为空)：**

**①Queue.empty()**

1. **return** OK;
2. }
3. // 判断是否为空
4. **bool** empty()
5. {
6. **if** (front == rear)
7. **return** **true**;
8. **else**
9. **return** **false**;
10. }
11. };

**2、栈**

1. **template**<**class** ElemType>
2. **struct**  Stack\_Node
3. {
4. ElemType   data;        // 数据
5. Stack\_Node  \*next;      // 下一个节点
6. };
8. **template**<**class** ElemType>
9. **class** LinkStack
10. {
11. Stack\_Node<ElemType>\* top;    // 头节点
12. **public**:
13. // 构造函数
14. LinkStack()
15. {
16. top = **new** Stack\_Node<ElemType>;
17. top->next = NULL;
18. }
19. // 析构函数

**Python对应方法(在栈顶添加一个元素)：**

**①list.append(e)**

1. ~LinkStack()
2. {
3. **delete** top;
4. }
5. // 将元素压入栈顶
6. **int** push(ElemType  e)
7. {
8. Stack\_Node<ElemType>  \*p;
9. p = **new** Stack\_Node<ElemType>;
10. **if** (!p)
11. **return** ERROR;
12. p->data = e;

**Python对应方法(弹出栈顶第一个元素)：**

**①list.pop()**

1. p->next = top->next;
2. top->next = p;
3. **return** OK;
4. }
5. // 弹出栈顶元素
6. **int** pop()
7. {
8. Stack\_Node<ElemType>  \*p;
9. **if** (top->next == NULL)
10. **return** ERROR;
11. p = top->next;
12. top->next = p->next;

**Python对应方法(访问栈顶第一个元素)：**

**①list.pop()**

**②list[len(list)-1]**

1. **delete** p;
2. **return** OK;
3. }
4. // 访问栈顶元素
5. **int** getTop(ElemType &e)
6. {
7. Stack\_Node<ElemType>  \*p;
8. **if** (top->next == NULL)
9. **return** ERROR;
10. p = top->next;

**Python对应方法(判断栈是否为空)：**

**①len(list)**

1. e = p->data;
2. **return** OK;
3. }
4. // 判断是否为空
5. **bool** empty()
6. {
7. **if** (top->next == NULL)
8. **return** **true**;
9. **else**
10. **return** **false**;
11. }
12. };

**3、树**

1. #define Max\_Node\_Num 10             //输入的数组大小
2. **struct** BiNode
3. {
4. **char** data;                             //可根据需要存储的数据类型进行修改
5. BiNode \*lchild,\*rchild;
6. };
7. **class** BiTree
8. {
9. **private**:
10. BiNode \*root;                       //root存储整棵树的根节点
11. BiNode \*Create(**char** \*c,**int** i)       //据顺序结构存储建立树,构造函数可根据树的表达形式进行修改

**C++中的数组在Python中可以用list代替，list的功能会比C++的数组更加强大。（树的结构在Python中也是可以通过构建类class来实现）**

1. {
2. **if**(i>=Max\_Node\_Num)
3. **return** NULL;
4. **if** (c[i]=='0')
5. **return** NULL;
6. BiNode\* T=**new** BiNode;
7. T->data=c[i];
8. T->lchild=Create(c,2\*i+1);
9. T->rchild=Create(c,2\*i+2);
10. **return** T;
11. }
12. **void** PreOrderTraverse(BiNode\* T)    //先序遍历
13. {
14. **if** (T)
15. {
16. cout<<T->data<<" ";
17. PreOrderTraverse(T->lchild);
18. PreOrderTraverse(T->rchild);
19. }
20. }
21. }

**4、图**

**①邻接矩阵建图**

1. /\*
2. \*  邻接矩阵建图
3. \*  适合稠密图（点少边多）
4. \*/
5. **const** **int** maxn = 1e5+5; // 图中点的最大数量
6. **int** g[maxn][maxn]; // 邻接矩阵 g[u][v]表示u->v的边的数量
7. **int** main(){
8. **int** n,m;  // 点数n 边数m
9. memset(g,0,**sizeof**(g));  // 清空矩阵
10. cout<<"请输入图的点数和边数"<<endl;
11. cin>>n>>m;
12. cout<<"请输入每一条边（示例：1 2表示1到2有一条边）"<<endl;
13. **for**(**int** i=0,v,u;i<m;++i){
14. cin>>v>>u;
15. ++g[v][u];
16. // g[u][v]=1;  // 若建无向图则要建立双向边
17. }

**C++中的数组在Python中可以用list代替，list的功能会比C++的数组更加强大，对于C++的多维数组，Python可以通过list相互嵌套代替**

1. }

**②邻接表建图**

1. /\*
2. \*  使用vector实现邻接表
3. \*  邻接表建图适合稀疏图（点多边少）
4. \*/
5. **class** Edge{
6. **public**:
7. **int** from;   // 始点
8. **int** to;     // 终点
9. Edge(){}
10. Edge(**int** \_from,**int** \_to):from(\_from),to(\_to) {}
11. };
12. **class** Graph{
13. **int** point\_nums;  // 点的数量
14. **int** edge\_nums;   // 边的数量
15. vector<vector<Edge> > G;
16. **public**:
17. Graph(**int** n,**int** m):point\_nums(n), edge\_nums(m),G(point\_nums) {}
18. **int** add\_edge(**int** from,**int** to){   // 增加一条从from点到to点的单向边
19. G[from].push\_back(Edge(from,to));   // 尾插法在点from的出边表插入边
20. }
21. };
22. **int** main(){
23. **int** n,m;  // 点数n 边数m
24. cout<<"请输入图的点数和边数"<<endl;
25. cin>>n>>m;
26. Graph g(n,m);
27. cout<<"请输入每一条边（示例：1 2表示1到2有一条边）"<<endl;
28. **for**(**int** i=0,v,u;i<m;++i){
29. cin>>v>>u;
30. g.add\_edge(v,u);
31. // g.add\_edge(u,v);  // 若建无向图则要建立双向边
32. }

**C++中的数组或者动态数组vector在Python中都可以用list代替，list的功能会比C++的数组更加强大，对于C++的多维数组，Python可以通过list相互嵌套代替（图的结构在Python中也是可以通过构建类class来实现）**

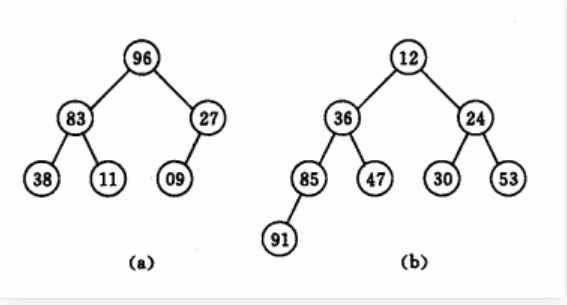
1. }

**5、优先级队列**

优先级队列，也叫二叉堆、堆

堆的本质是一种完全二叉树，分为：

* 最大堆（大根堆）：树中每个非叶子节点都不小于其左右孩子节点的值，也就是根节点值最大的堆，图（a）。
* 最小堆（小根堆）：树中每个非叶子节点都不大于其左右孩子节点的值，也就是根节点值最小的堆，图（b）。



1. **import** queue as Q
3. **class** Skill(object):
4. **def** \_\_init\_\_(self,priority,description):
5. self.priority = priority

**为优先级队列定义元素大小关系**

1. self.description = description
3. **def** \_\_lt\_\_(self,other):
4. **return** self.priority < other.priority
6. **def** \_\_str\_\_(self):
7. **return** '(' + str(self.priority)+',\'' + self.description + '\')'

**向优先级队列添加元素**

1. **def** PriorityQueue\_class():
2. que = Q.PriorityQueue()
3. que.put(Skill(7,'proficient7'))
4. que.put(Skill(5,'proficient5'))
5. que.put(Skill(6,'proficient6'))
6. que.put(Skill(10,'expert'))
7. que.put(Skill(1,'novice'))
8. **print** ('end')

**访问优先级队列的根节点（最小值或最大值）**

1. **while** **not** que.empty():
2. **print** (que.get())
4. PriorityQueue\_class()

**人工智能导论-数据结构习题**

**1、队列**

（1）在操作序列push(1)、 push(2) 、pop、 push(5)、 push(7)、pop、push(6)之后，队头元素和队尾元素分别是什么？(push(k)表示整数k入队，pop表示队头元素出队)

（2）循环队列的队首和队尾下标分别为front, rear，最大长度为n（base[n]），写出判断队满和队空的条件

1. 若n=8，front=3, rear=2，请判断队列是否满或空
2. 若n=8，front=3, rear=6，队列从0开始编号，指出哪几个位置为空

（3）**（编程题）**输入多个正整数（0表示输入结束），将输入数据的偶数作为一列，奇数作为一列，分别在屏幕上输出

**输入格式**

输入多个正整数，最后以0表示输入结束

**输出格式**

按输入的顺序输出

第一行输出为奇数

第二行输出为偶数

**输入输出样例**

输入：1 2 4 5 7 8 10 3 6 9 **0**

输出：1 5 7 3 9

2 4 8 10 6

（4）**(编程题)**四月一日快到了，Vayko想了个愚人的好办法——送礼物。嘿嘿，不要想的太好，这礼物可没那么简单，Vayko为了愚人，准备了一堆盒子，其中有一个盒子里面装了礼物。盒子里面可以再放零个或者多个盒子。假设放礼物的盒子里不再放其他盒子。

用()表示一个盒子，B表示礼物，Vayko想让你帮她算出愚人指数，即最少需要拆多少个盒子才能拿到礼物。

**输入格式**

输入一行字符串代表Vayko的礼物透视图

**输出格式**

输出愚人指数，即最少需要拆多少个盒子才能拿到礼物。

**输入输出样例**

输入：(((()(B)()))())

输出：4

**2、栈**

（1）一个栈的入栈序列是a,b,c,d,e,允许任何时候出栈，则栈的不可能的输出序列是（）。

（A）edcba

（B）decba

（C）dceab

（D）abcde

（2）入栈顺序为1、2、3、4，允许任何时候出栈，写出所有可能的出栈序列。

（3）（编程题）

将十进制转换为八进制，其原理为N=(N / 8)\*8 + N mod 8，例如(1348)10=(2504)8

**输入格式**

输入只含一个 十进制整数n

**输出格式**

输出为转化后的八进制数

**输入输出样例**

输入：1348 输出：2504

（4）（编程题）

一个算术表达式中包括**圆括号、方括号和花括号**三种形式的括号，判别表达式中括号是否正确匹对

括号匹配结果有四种：

* **0：左右括号匹配正确 {[(1+2)\*3]-1}**
* **-1：左右括号配对次序不正确 {[(1+2]\*3)-1}**
* **-2：右括号多于左括号 (1+2)\*3)-1}**
* **-3：左括号多于右括号 {[(1+2)\*3-1]**

**输入格式**

输入一个算术表达式

**输出格式**

输出对应的符号匹配结果

**输入输出样例**

输入：**{[(1+2)\*3]-1}**  输出：0

输入：**{[(1+2]\*3)-1}** 输出：-1

输入：**(1+2)\*3)-1}** 输出：-2

输入：**{[(1+2)\*3-1]**  输出：-3

**3、树**

（1）已知一颗完全二叉树最后一层为第7层有20个结点

①求整棵树的结点数

②求这棵树的叶子节点个数

（2）以下不能唯一确定一棵二叉树的是？

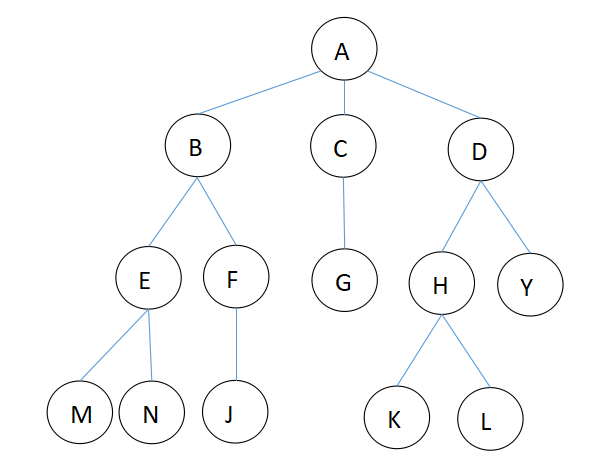
（A）中序遍历+先序遍历

（B）中序遍历+后序遍历

（C）先序遍历+后序遍历

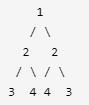
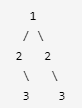
（3）假设一棵二叉树的先序遍历序列为ABDEHCFGI，中序遍历序列为DBEHAFCIG，请画出这棵二叉树，并写出这棵树的顺序存储（空节点用0表示）。

（4）请写出下面这棵树的深度遍历结果与广度遍历结果：



（5）**（编程题）**在已经实现的链式存储二叉树的类中，新增一个bool类型的判断方法isSymmetric，判断这棵树是否为对称二叉树，即镜像对称的。

例：

对应的顺序存储（0表示空节点）： 1223443 1220303

是否为对称二叉树： 是 否

**输入格式**：树的顺序存储

**输出输出**：是否为对称二叉树

**测试数据（0表示空节点）： 对应结果：**

1223443 是

1220303 否

1. 是
2. 是
3. 是

1201 是

（6）**（编程题）**编程实现多叉树类的基本实现，要求至少包含以下方法：

①构造函数建立根节点

②根据给出的节点关系建立父亲节点与孩子的关系

③查询某个节点的父亲

④查询某个节点的孩子

⑤树的深度遍历

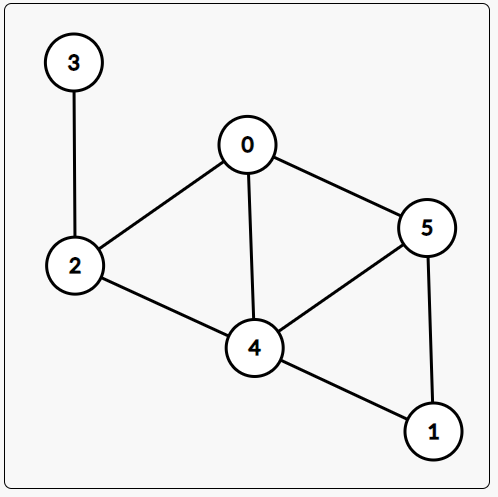
⑥树的广度遍历

注：建立树的方式：建立根节点后，自己通过②中的插入方法进行构建，测试多叉树的基本功能，可以借助上题中的树进行测试。

提示：可以利用python中的字典来储存父亲节点与孩子的信息。

**4、图**

（1）求下图从点0出发的深度优先遍历依次经过的点为 0 , , , , , , （优先访问编号小的点），从点0出发的广度优先遍历依次经过的点为 0 , , , , , , 。（优先访问编号小的点）



（2）**（编程题）**给出N个点，M条边的有向图，对于每个点v，求A(v)表示从点v出发，能到达的编号最大的点

**输入格式**

第1行，2个整数N，M。

接下来M行，每行2个整数Ui，Vi，表示边（Ui，Vi）。点用1，2，…，N编号。

**输出格式**

N个整数A(1)，A(2)，…，A(N)。

**输入输出样例**

输入： 输出：

4 3 4 4 3 4

1 2

2 4

4 3