1. **抽象数据类型的定义（ADT）**

作用：抽象数据类型可以使我们更容易描述现实世界。例：用线性表描述学生成绩表，用树或图描述遗传关系。

定义：一个数学模型以及定义在该模型上的一组操作。

关键：使用它的人可以只关系它的逻辑特征，不需要了解它的存储方式。定义它的人同样不必要关心它如何存储。

例：线性表这样的抽象数据类型，其数学模型是：数据元素的集合，该集合内的元素有这样的关系：除第一个和最后一个外，每个元素有唯一的前驱和唯一的后继。可以有这样一些操作：插入一个元素、删除一个元素等。

|  |  |
| --- | --- |
| 抽象数据类型分类 | |
| 原子类型 | 值不可分解，如int |
| 固定聚合类型 | 值由确定数目的成分按某种结构组成，如复数 |
| 可变聚合类型 | 值的成分数目不确定，如学生基本情况 |

抽象数据类型表示法：

1. 三元组表示：（D，S，P）

其中D是数据对象，S是D上的关系集，P是对D的基本操作集。

1. 书中的定义格式：

ADT抽象数据类型名 {

数据对象：<数据对象的定义>

数据关系：<数据关系的定义>

基本操作：<基本操作的定义>

}ADT抽象数据类型名

例：线性表的表示

|  |  |  |
| --- | --- | --- |
| 名称 | 线性表 |  |
| 数据对象 | D={ai| ai(-ElemSet,i=1,2,...,n,n>=0} | 任意数据元素的集合 |
| 数据关系 | R1={<ai-1,ai>| ai-1,ai(- D,i=2,...,n} | 除第一个和最后一个外，每个元素有唯一的直接前驱和唯一的直接后继 |
| 基本操作 | ListInsert(&L,i,e) | L为线性表，i为位置，e为数据元素。 |
| ListDelete(&L,i,e) |
| ... |

我们要使用单链表这个数据结构来解决问题的前提是首先得创建一个单链表数据结构。创建单链表数据结构，就得自定义个单链表的抽象数据类型，抽象数据类型只是对数据结构定义一组逻辑操作，没有具体的实现。在实际应用中，必须实现这个抽象数据类型，才能使用它们，而实现抽象数据类型依赖于数据存储结构。

**1. 为单链表的数据元素自定义**结点类**Node.java**

1. /\*\*
2. \*为单链表自定义的结点类
3. \*单链表结点类，泛型参数T指定结点的元素类型
4. \*/
5. **public** **class** Node<T>
6. {
7. **public** T data;   //数据域，保存数据元素
8. **public** Node<T> next; //地址域，引用后继结点
10. **public** Node(T data, Node<T> next)//构造结点，data指定数据元素，next指定后继结点
11. {
12. **this**.data = data;
13. **this**.next = next;
14. }
15. **public** Node()//无参构造函数
16. {
17. **this**(**null**, **null**);
18. }
20. //4、Node类可声明以下方法：
21. **public** String toString()                     //返回结点元素值对应的字符串
22. {
23. **return** **this**.data.toString();
24. }
25. **public** **boolean** equals(Object obj)            //比较两个结点值是否相等，覆盖Object类的equals(obj)方法
26. {
27. **return** obj==**this** || obj **instanceof** Node && **this**.data.equals(((Node<T>)obj).data);
28. }
29. }

**2.使用Java接口为线性表自定义的**抽象数据类型**：LList.java**

1. /\*\*
2. \* 为数据结构线性表自定义的抽象数据类型
3. \* 在Java中，抽象数据类可以使用接口来描述
4. \* 线性表接口LList，描泛型参数T表示数据元素的数据类型
5. \*/
6. **public** **interface** LList<T>     //线性表接口，泛型参数T表示线性表中的数据元素的数据类型
7. {
8. **boolean** isEmpty();        //判断线性表是否空
9. **int** length();             //返回线性表长度
10. T get(**int** i);             //返回第i（i≥0）个元素
11. **void** set(**int** i, T x);     //设置第i个元素值为x
12. **void** insert(**int** i, T x);  //插入x作为第i个元素
13. **void** append(T x);         //在线性表最后插入x元素
14. T remove(**int** i);          //删除第i个元素并返回被删除对象
15. **void** removeAll();         //删除线性表所有元素
16. T search(T key);          //查找，返回首次出现的关键字为key元素
17. String toString();        //返回显示线性表所有元素值对应的字符串
18. }
19. **实现线性表接口---抽象数据类型的实现类-LinkedList.java（链表类，提供Llist接口中抽象方法的具体实现）**
20. /\*\*
21. \*线性表的链式表示和实现
22. \*带头结点的单链表类
23. \*实现线性表接口
24. \*/
25. **public** **class** LinkedList<T>**implements** LList<T>//带头结点的单链表类，实现线性表接口
26. {
27. **protected** Node<T> head;//头指针，指向单链表的头结点
28. //默认构造方法，构造空单链表。创建头结点，data和next值均为null
29. **public** LinkedList()
30. {
31. **this**.head=**new** Node<T>();
32. }
33. 由指定数组中的多个对象构造单链表，采用尾插入构造单链表
34. **public** LinkedList(T[] element){
36. **this**(); //创建空单链表，只有头结点
37. Node<T> rear = **this**.head;//rear指向单链表的最后一个结点
38. /\*
39. \*若element==null，抛出空对象异常
40. \*element.length==0时，构造空链表
41. \*/
42. **for**(**int** i=0;i<element.length;i++){
43. rear.next=**new** Node<T>(element[i],**null**);//尾插入，创建结点链入rear结点之后
44. rear=rear.next;//rear指向新的链尾结点
45. }
46. }
47. //判断单链表是否为空，O(1)
48. **public** **boolean** isEmpty(){
49. **return** **this**.head.next==**null**;
50. }
51. //求链表的长度
52. **public** **int** length(){
53. **int** i=0;
54. Node<T> p = **this**.head.next;//p从单链表第一个结点开始
55. **while**(p!=**null**){ //若单链表未结束
56. i++;
57. p=p.next;//p到达后继结点
58. }
59. **return** i;
60. }
61. //返回单链表所有元素的描述字符串，形式为“（，）”，覆盖Object类的toString()方法，O(n)
62. **public** String toString(){
63. String str="(";
64. Node<T> p =**this**.head.next;
65. **while**(p!=**null**){
66. str+=p.data.toString();
67. **if**(p.next!=**null**){
68. str +="，";  //不是最后一个结点时后加分隔符
69. }
70. p=p.next;
71. }
72. **return** str+=")";
73. }
74. //返回第i(i>=0)个元素，若i无效，则返回null
75. **public** T get(**int** i){
76. **if**(i>=0){
77. Node<T> p=**this**.head.next;
78. **for**(**int** j=0; p!=**null**&&j<i;j++)
79. p=p.next;
80. **if**(p!=**null**)
81. **return** p.data;//p指向第i个结点
82. }
83. **return** **null**;
84. }
85. //设置第i(i>=0)个元素值为x，若i指定序号无效则抛出序号越界异常
86. **public** **void** set(**int** i,T x){
87. **if**(x==**null**)
88. **return**;//不能设置空对象
89. **if**(i>=0){
90. Node<T> p =**this**.head.next;
91. **for**(**int** j =0;p!=**null**&&j<i;j++){
92. p=p.next;
93. }
94. **if**(p!=**null**)
95. p.data=x;
96. }**else** **throw** **new** IndexOutOfBoundsException(i+"");//抛出序号越界异常
97. }
98. //将x对象出插入在序号为i结点前，O(n)
99. **public** **void** insert(**int** i,T x){
100. **if**(x==**null**)
101. **return**;
102. Node<T> p =**this**.head;
103. **for**(**int** j =0;p.next!=**null**&&j<i;j++){
104. p=p.next;
105. }
106. p.next=**new** Node<T>(x,p.next);
107. }
108. //在单链表的最后添加对象，O(n)
109. **public** **void** append(T x){
110. insert(Integer.MAX\_VALUE,x);
111. }
112. //删除序号为i的结点
113. **public** T remove(**int** i){
114. **if**(i>0){
115. Node<T> p=**this**.head;
116. **for**(**int** j =0;p.next!=**null**&&j<i;j++)
117. p=p.next;
118. **if**(p.next!=**null**){
119. T old=p.next.data;
120. p.next=p.next.next;
121. **return** old;
122. }
123. }
124. **return** **null**;
125. }
126. //删除链表所有元素，Java自动收回各结点所占用的内存空间
127. **public** **void** removeAll(){
128. **this**.head.next=**null**;
129. }
130. /\*
131. public T search(T key){
132. if(key==null)
133. return null;
134. Node<T> p=this.head.next;
135. while(p!=null&&p.data.compareTo(key)<=0){
136. if(p.data.compareTo(key)==0)
137. return p.data;
138. p=p.next;
139. }
140. return null;
141. }
142. \*/
143. //查找，返回首次出现的关键字为key的元素
144. **public** T search(T key)
145. {
146. **if** (key==**null**)
147. **return** **null**;
148. **for** (Node<T> p=**this**.head.next;  p!=**null**;  p=p.next)
149. **if** (p.data.equals(key))
150. **return** p.data;
151. **return** **null**;
152. }
153. }

**4.利用前面新创建的单链表数据类型，实现单链表逆转SinglyLinkedList\_reverse.java**

注：LinkedList<T>声明为泛型类，类型形式参数T表示链表元素的数据类型。当声明LinkedList类的对象并创建实例时，再指定泛型参数T的实际类型参数为一个确定的类，例如：LinkedList<String>list=new LinkedList<String>();LinkedList<Integer>list=new LinkedList<Integer>();这样可保证一个链表中的所有数据元素是相同类及其子类的对象。如果向链表添加指定泛型以外的对象，则会出现编译错误。T的实际类型参数必须是类，不能使int、char等基本数据类型。如果需要表示基本数据类型，则必须使用对应数据类型的包装类，如Integer、Character等。

1. /\*\*
2. \*利用前面新创建的单链表数据类型，实现单链表逆转
3. \*/
4. **public** **class** SinglyLinkedList\_reverse
5. {
6. //将单链表逆转，泛型方法，返回值类型前声明类型参数T
7. //public static <T> void reverse(SinglyLinkedList<T> list)
8. **public** **static** <T> **void** reverse(LinkedList<T> list)
9. {
10. Node<T> p=list.head.next, succ=**null**, front=**null**;   //head必须声明为public
11. **while** (p!=**null**)
12. {
13. succ = p.next;  //设置succ是p结点的后继结点
14. p.next = front; //使p.next指向p结点的前驱结点
15. front = p;
16. p = succ;       //p向后走一步
17. }
18. list.head.next = front;
19. }
21. **public** **static** **void** main(String args[])
22. {
23. String value[]={"A","B","C","D","E","F"};
24. //SinglyLinkedList<String> list = new SinglyLinkedList<String>(value);
25. LinkedList<String> list = **new** LinkedList<String>(value);
26. System.out.println("list: "+list.toString());
27. reverse(list);
28. System.out.println("逆转后 "+list.toString());
29. }
30. }
31. /\*
32. 程序运行结果如下：
33. list: (A, B, C, D, E, F)
34. 逆转后 (F, E, D, C, B, A)
35. \*/