数据结构的Java表示

[1．链表 2](#_Toc429161095)

[1.1线性链表 2](#_Toc429161096)

[1.2链式链表 4](#_Toc429161097)

[1.3双端链表 8](#_Toc429161098)

[2.栈 10](#_Toc429161099)

[2.1线性栈 10](#_Toc429161100)

[2.2链式栈 12](#_Toc429161101)

[3.队列 14](#_Toc429161102)

[3.1线性单端队列 14](#_Toc429161103)

[3.2链式单端队列 16](#_Toc429161104)

[3.3线性循环队列 19](#_Toc429161105)

[3.4线性双端队列 22](#_Toc429161106)

[4.二叉树（二叉链表的实现） 26](#_Toc429161107)

[5.图(邻接链表表示法)和最短路径Dijkstra算法 30](#_Toc429161108)

[6.排序算法 36](#_Toc429161109)

[7.常见考题（更新中…） 41](#_Toc429161110)

[7.1单链表反转 41](#_Toc429161111)

*详情请访问个人github主页:* [*https://github.com/HuiT2015/MyDocuments*](https://github.com/HuiT2015/MyDocuments)

# 1．链表

## 1.1线性链表

1. package com.ex.list;
2. import java.io.BufferedReader;
3. import java.io.FileNotFoundException;
4. import java.io.FileReader;
5. import java.io.IOException;
6. public class MySeqList<M> {
7. // 设置线性链表的最大容量
8. private static final int MAX\_SIZE = 100;
9. // 创建线性链表的数据结构
10. private static class SeqList<T> {
11. T[] list = (T[]) new Object[MAX\_SIZE];
12. int length;
13. }
14. // 定义一个线性链表
15. private SeqList<M> seqList;
16. // 初始化线性链表
17. private void initSeqList() {
18. if (seqList == null) {
19. seqList = new SeqList<M>();
20. }
21. }
22. // 获取位置为i的链表节点
23. private M getElem(int i) {
24. if (i < 1 || i > seqList.length)
25. return null;
26. return seqList.list[i - 1];
27. }
28. // 在地i个位置插入节点e
29. private void insert(int i, M e) {
30. if (i < 1 || i > seqList.length + 1) {
31. System.out.println("插入位置非法！");
32. return;
33. } else if (seqList.length >= MAX\_SIZE) {
34. System.out.println("链表已满员，无法进行插入操作！");
35. seqList.list = (M[]) new Object[2 \* MAX\_SIZE];
36. return;
37. } else {
38. for (int j = seqList.length; j >= i; j++) {
39. seqList.list[j] = seqList.list[j - 1];
40. }
41. seqList.list[i - 1] = e;
42. seqList.length += 1;
43. }
44. }
45. // 删除第i个节点的值
46. private M deleteAt(int i) {
47. M result = null;
48. if (i < 1 || i > seqList.length) {
49. System.out.println("删除位置非法！");
50. return result;
51. }
52. result = seqList.list[i - 1];
53. for (int j = i - 1; j < seqList.length - 1; j++) {
54. seqList.list[j] = seqList.list[j + 1];
55. }
56. return result;
57. }
58. // 从文件中创建链表
59. private void createSeqList() {
60. try {
61. BufferedReader bufferedReader = new BufferedReader(new FileReader("list.txt"));
62. try {
63. String[] strArr = bufferedReader.readLine().split(" ");
64. int nodeNum = Integer.parseInt(strArr[0]);
65. for (int i = 0; i < nodeNum; i++) {
66. Character elem = Character.valueOf(strArr[i + 1].charAt(0));
67. insert(i + 1, (M) elem);
68. }
69. bufferedReader.close();
70. } catch (IOException e) {
71. // TODO Auto-generated catch block
72. e.printStackTrace();
73. }
74. } catch (FileNotFoundException e) {
75. // TODO Auto-generated catch block
76. e.printStackTrace();
77. }
78. }
79. // 打印线性链表
80. private void printSeqList() {
81. for (int i = 0; i < seqList.length; i++) {
82. System.out.print(seqList.list[i] + " ");
83. }
84. }
85. /\*\*
86. \* @param args
87. \*/
88. public static void main(String[] args) {
89. // TODO Auto-generated method stub
90. MySeqList<Character> mySeqList = new MySeqList<Character>();
91. mySeqList.initSeqList();
92. System.out.println("创建线性链表... ");
93. mySeqList.createSeqList();
94. System.out.println("打印线性链表...");
95. mySeqList.printSeqList();
96. }
97. }

## 1.2链式链表

1. package com.ex.list;
2. import java.io.BufferedReader;
3. import java.io.FileNotFoundException;
4. import java.io.FileReader;
5. import java.io.IOException;
6. import java.util.Scanner;
7. public class MyLinkedList<M> {
8. // 定义单链表的数据结构
9. private static class LinkedNode<T> {
10. T data;
11. LinkedNode<T> next;
12. public LinkedNode() {
13. this(null, null);
14. }
15. public LinkedNode(T data) {
16. this(data, null);
17. }
18. public LinkedNode(T data, LinkedNode<T> next) {
19. this.data = data;
20. this.next = next;
21. }
22. }
23. // 定义一个单链表的头指针
24. private LinkedNode<M> head;
25. /\*\*
26. \* 初始化链表
27. \*/
28. private void initList() {
29. head = null;
30. }
31. /\*\*
32. \* 创建链表
33. \*/
34. private void createLinkedList() {
35. if (head == null) {
36. head = new LinkedNode<M>();
37. }
38. BufferedReader bufferedReader;
39. String[] strArr = null;
40. try {
41. bufferedReader = new BufferedReader(new FileReader("list.txt"));
42. try {
43. strArr = bufferedReader.readLine().split(" ");
44. bufferedReader.close();
45. } catch (IOException e) {
46. // TODO Auto-generated catch block
47. e.printStackTrace();
48. }
49. } catch (FileNotFoundException e) {
50. // TODO Auto-generated catch block
51. e.printStackTrace();
52. }
53. int nodeNum = Integer.parseInt(strArr[0]);
54. LinkedNode<M> newNode = null;
55. for (int i = 0; i < nodeNum; i++) {
56. Character inputCh = strArr[i + 1].charAt(0);
57. newNode = new LinkedNode<M>((M) inputCh);
58. newNode.next = head.next;
59. head.next = newNode;
60. }
61. }
62. /\*\*
63. \* 打印链表
64. \*/
65. private void printLinkedList() {
66. LinkedNode<M> p = head.next;
67. while (p != null) {
68. System.out.print(p.data + "\t");
69. p = p.next;
70. }
71. }
72. /\*\*
73. \* n指第几个节点（1,2,3...),nFlag={-1,0,1},分别返回前、中、后节点
74. \*
75. \* @param n
76. \* @param nFlag
77. \* @return
78. \*/
79. private LinkedNode<M> locateNode(int n, int nFlag) {
80. LinkedNode<M> p = head;
81. int nCnt = 0;
82. while (p.next != null && nCnt < n + nFlag) {
83. p = p.next;
84. nCnt++;
85. }
86. return p;
87. }
88. /\*\*
89. \* 插入节点操作
90. \*
91. \* @param e
92. \* @param i
93. \*/
94. private void insertNode(M e, int i) {
95. LinkedNode<M> pre = locateNode(i, -1);
96. LinkedNode<M> newNode = new LinkedNode<M>(e);
97. newNode.next = pre.next;
98. pre.next = newNode;
99. }
100. /\*\*
101. \* 删除节点操作
102. \*
103. \* @param i
104. \*/
105. private void eraseNode(int i) {
106. LinkedNode<M> pre = locateNode(i, -1);
107. LinkedNode<M> cur = locateNode(i, 0);
108. pre.next = cur.next;
109. cur.next = null;
110. cur.data = null;
111. cur = null;
112. }
113. /\*\*
114. \* 主函数
115. \*
116. \* @param args
117. \*/
118. public static void main(String[] args) {
119. MyLinkedList<Character> myLinkedList = new MyLinkedList<Character>();
120. myLinkedList.initList();
121. System.out.println("建立单链表！");
122. myLinkedList.createLinkedList();
123. System.out.println("打印单链表！");
124. myLinkedList.printLinkedList();
125. }
126. }

## 1.3双端链表

1. package com.ex.list;
2. public class DBLinkedList<T> {
3. /\*\*
4. \* 定义双端链表的数据结构
5. \*
6. \* @author lenovo
7. \* @param <M>
8. \*/
9. static final class DBLinkNode<M> {
10. M data;
11. DBLinkNode<M> prior;
12. DBLinkNode<M> next;
13. public DBLinkNode(M data, DBLinkNode<M> prior, DBLinkNode<M> next) {
14. this.data = data;
15. this.prior = prior;
16. this.next = next;
17. }
18. }
19. // 定义一个双端链表头指针
20. private DBLinkNode<T> head;
21. /\*\*
22. \* 双端链表初始化函数
23. \*/
24. private void init() {
25. head = new DBLinkNode<T>(null, null, null);
26. head.prior = head;
27. head.next = head;
28. }
29. /\*\*
30. \* 在指定位置插入元素
31. \*
32. \* @param e
33. \* @param nPos
34. \*/
35. private void insertElement(T e, int nPos) {
36. DBLinkNode<T> cur = getLocation(nPos);
37. DBLinkNode<T> newNode = new DBLinkNode<T>(e, cur.prior, cur);
38. cur.prior.next = newNode;
39. cur.prior = newNode;
40. }
41. /\*\*
42. \* 获取指定位置的元素
43. \*
44. \* @param nPos
45. \* @return
46. \*/
47. private DBLinkNode<T> getLocation(int nPos) {
48. int i = 0;
49. DBLinkNode<T> cur = head.next;
50. while (i < nPos && cur != head) {
51. cur = cur.next;
52. i++;
53. }
54. return cur;
55. }
56. /\*\*
57. \* 删除指定位置的元素，并返回它
58. \*
59. \* @param nPos
60. \* @return
61. \*/
62. private T deleteElement(int nPos) {
63. DBLinkNode<T> cur = getLocation(nPos);
64. cur.next.prior = cur.prior;
65. cur.prior.next = cur.next;
66. return cur.data;
67. }
68. /\*\*
69. \* 打印双端链表元素
70. \*/
71. private void printDBLinkedList() {
72. DBLinkNode<T> cur = head.next;
73. int i = 0;
74. while (cur != head) {
75. System.out.println("第" + (++i) + "个元素为:" + cur.data);
76. cur = cur.next;
77. }
78. }
79. /\*\*
80. \* 主函数
81. \*
82. \* @param args
83. \*/
84. public static void main(String[] args) {
85. // TODO Auto-generated method stub
86. DBLinkedList<Integer> dLinkedList = new DBLinkedList<Integer>();
87. dLinkedList.init();
88. for (int i = 0; i < 10; i++) {
89. dLinkedList.insertElement(2 \* i + 1, i);
90. }
91. System.out.println("初始化时双端链表中的元素为:");
92. dLinkedList.printDBLinkedList();
93. System.out.println("删除第4个元素" + dLinkedList.deleteElement(3) + "后的双端链表为:");
94. dLinkedList.printDBLinkedList();
95. }
96. }

# 2.栈

## 2.1线性栈

1. package com.ex.stack;
2. import java.io.BufferedReader;
3. import java.io.FileNotFoundException;
4. import java.io.FileReader;
5. import java.io.IOException;
6. public class MySeqStack<M> {
7. // 定义一个线性栈的容量常值
8. private static int STACK\_SIZE = 100;
9. // 定义线性栈的数据结构
10. private static class SeqStack<T> {
11. T[] stack = (T[]) new Object[STACK\_SIZE];
12. int top;
13. }
14. // 定义一个线性栈
15. private SeqStack<M> seqStack;
16. // 初始化线性栈
17. private void initSeqStack() {
18. if (seqStack == null) {
19. seqStack = new SeqStack<M>();
20. }
21. }
22. /\*\*
23. \* 进栈操作
24. \*
25. \* @param e
26. \*/
27. private void push(M e) {
28. if (seqStack.top == STACK\_SIZE) {
29. System.out.println("线性栈已满员，需要扩容!");
30. STACK\_SIZE \*= 2;
31. seqStack.stack = (M[]) new Object[STACK\_SIZE];
32. }
33. seqStack.stack[seqStack.top++] = e;
34. }
35. /\*\*
36. \* 栈顶元素出栈
37. \*
38. \* @return
39. \*/
40. private M pop() {
41. return seqStack.stack[--seqStack.top];
42. }
43. /\*\*
44. \* 主函数
45. \*
46. \* @param args
47. \*/
48. public static void main(String[] args) {
49. MySeqStack<Integer> mySeqStack = new MySeqStack<Integer>();
50. System.out.println("建立线性栈...");
51. mySeqStack.initSeqStack();
52. int stackCapacity = -1;
53. try {
54. BufferedReader bufferedReader = new BufferedReader(new FileReader("stack.txt"));
55. try {
56. String[] strArr = bufferedReader.readLine().split(" ");
57. stackCapacity = Integer.parseInt(strArr[0]);
58. for (int i = 0; i < stackCapacity; i++) {
59. mySeqStack.push(Integer.parseInt(strArr[i + 1]));
60. }
61. bufferedReader.close();
62. } catch (IOException e) {
63. // TODO Auto-generated catch block
64. e.printStackTrace();
65. }
66. } catch (FileNotFoundException e) {
67. // TODO Auto-generated catch block
68. e.printStackTrace();
69. }
70. System.out.println("打印线性栈...");
71. for (int i = 0; i < stackCapacity; i++) {
72. System.out.print(mySeqStack.pop() + " ");
73. }
74. }
75. }

## 2.2链式栈

* 1. package com.ex.stack;
  2. public class LinkedStack<T> {
  4. /\*\*
  5. \* 定义一个链式栈的数据结构
  6. \* @author lenovo
  7. \*
  8. \* @param <T>
  9. \*/
  10. static class LinkedNode<T>
  11. {
  12. T data;
  13. LinkedNode<T> next;
  14. public LinkedNode(T data,LinkedNode<T> next) {
  15. this.data = data;
  16. this.next = next;
  17. }
  19. }
  21. //顶一个栈顶指针
  22. public LinkedNode<T> top=null;
  24. /\*\*
  25. \* 进栈
  26. \* @param e
  27. \*/
  28. public void push(T e) {
  29. if (top==null) {
  30. top=new LinkedNode<T>(null, null);
  31. }
  32. LinkedNode<T> newNode=new LinkedNode<T>(e, top.next);
  33. top.next=newNode;
  34. }
  36. /\*\*
  37. \* 出栈
  38. \* @return
  39. \*/
  40. public T pop() {
  41. T result=null;
  42. LinkedNode<T> p=top.next;
  43. if (p!=null) {
  44. top.next=p.next;
  45. result=p.data;
  46. }
  47. return result;
  48. }
  50. /\*\*
  51. \* 返回栈顶元素而不删除
  52. \* @return
  53. \*/
  54. public T peekTop() {
  55. return top.next.data;
  56. }
  58. /\*\*
  59. \* 清空栈中的元素
  60. \*/
  61. public void clear() {
  62. LinkedNode<T> p=top;
  63. LinkedNode<T> q=null;
  64. while (p!=null) {
  65. q=p;
  66. p.data=null;
  67. p.next=null;
  68. p=null;
  69. p=q.next;
  70. }
  71. }
  73. /\*\*
  74. \* 判断栈是否为空
  75. \* @return
  76. \*/
  77. public boolean isEmpty() {
  78. return top.next==null;
  79. }
  80. }

# 3.队列

## 3.1线性单端队列

* 1. package com.ex.queue;
  2. import java.io.BufferedReader;
  3. import java.io.FileNotFoundException;
  4. import java.io.FileReader;
  5. import java.io.IOException;
  6. public class MySeqQueue<M> {
  7. // 定义队列的容量大小
  8. private static final int QUEUE\_SIZE = 100;
  9. // 定义单端队列的数据结构
  10. private static class SeqQueue<T> {
  11. T[] queue = (T[]) new Object[QUEUE\_SIZE];
  12. int front, rear;
  13. }
  14. // 定义单端队列
  15. private SeqQueue<M> seqQueue;
  16. /\*\*
  17. \* 初始化单端队列
  18. \*/
  19. private void initSeqQueue() {
  20. if (seqQueue == null) {
  21. seqQueue = new SeqQueue<M>();
  22. }
  23. seqQueue.front = seqQueue.rear = 0;
  24. }
  25. /\*\*
  26. \* 向队尾添加元素
  27. \*
  28. \* @param e
  29. \*/
  30. private void enterQueue(M e) {
  31. if (seqQueue.rear == QUEUE\_SIZE) {
  32. System.out.println("队列满员！");
  33. return;
  34. }
  35. seqQueue.queue[seqQueue.rear++] = e;
  36. }
  37. /\*\*
  38. \* 从对头删除元素
  39. \*
  40. \* @return
  41. \*/
  42. private M deleteQueue() {
  43. if (seqQueue.front == seqQueue.rear) {
  44. System.out.println("队列已经为空！");
  45. return null;
  46. }
  47. return seqQueue.queue[seqQueue.front++];
  48. }
  49. /\*\*
  50. \* 从文件中创建队列
  51. \*/
  52. private void createSeqQueue() {
  53. BufferedReader bufferedReader;
  54. String[] strArr = null;
  55. try {
  56. bufferedReader = new BufferedReader(new FileReader("queue.txt"));
  57. try {
  58. strArr = bufferedReader.readLine().split(" ");
  59. bufferedReader.close();
  60. } catch (IOException e) {
  61. // TODO Auto-generated catch block
  62. e.printStackTrace();
  63. }
  64. } catch (FileNotFoundException e) {
  65. // TODO Auto-generated catch block
  66. e.printStackTrace();
  67. }
  68. int queueSize = Integer.parseInt(strArr[0]);
  69. for (int i = 0; i < queueSize; i++) {
  70. Object elem = (Object) Integer.parseInt(strArr[i + 1]);
  71. enterQueue((M) elem);
  72. }
  73. }
  74. /\*\*
  75. \* 主函数
  76. \*
  77. \* @param args
  78. \*/
  79. public static void main(String[] args) {
  80. MySeqQueue<Integer> mySeqQueue = new MySeqQueue<Integer>();
  81. mySeqQueue.initSeqQueue();
  82. System.out.println("建立线性队列！");
  83. mySeqQueue.createSeqQueue();
  84. System.out.println("打印线性队列！");
  85. for (int i = 0; i < mySeqQueue.seqQueue.rear; i++) {
  86. System.out.print(mySeqQueue.deleteQueue() + " ");
  87. }
  88. }
  89. }

## 3.2链式单端队列

1. package com.ex.queue;
2. import java.io.BufferedReader;
3. import java.io.FileNotFoundException;
4. import java.io.FileReader;
5. import java.io.IOException;
6. public class MyLinkedQueue<M> {
7. /\*\*
8. \* 定义链式队列结点数据结构
9. \*
10. \* @author lenovo
11. \*
12. \* @param <T>
13. \*/
14. private final static class QueueNode<T> {
15. T data;
16. QueueNode<T> next;
17. public QueueNode() {
18. this(null, null);
19. }
20. public QueueNode(T data) {
21. this(data, null);
22. }
23. public QueueNode(T data, QueueNode<T> next) {
24. this.data = data;
25. this.next = next;
26. }
27. }
28. /\*\*
29. \* 定义链式队列数据结构
30. \*
31. \* @author lenovo
32. \*
33. \* @param <M>
34. \*/
35. private final static class LinkQueue<M> {
36. QueueNode<M> front;
37. QueueNode<M> rear;
38. }
39. /\*\*
40. \* 定义链式队列
41. \*/
42. private LinkQueue<Integer> LQ;
43. /\*\*
44. \* 初始化队列操作
45. \*/
46. private void initLinkedQueue() {
47. if (LQ == null) {
48. LQ = new LinkQueue<Integer>();
49. }
50. LQ.front = LQ.rear = new QueueNode<Integer>();
51. }
52. /\*\*
53. \* 入队列操作
54. \*
55. \* @param elem
56. \*/
57. private void enterQueue(int elem) {
58. QueueNode<Integer> newNode = new QueueNode<Integer>(elem);
59. LQ.rear.next = newNode;
60. LQ.rear = newNode;
61. }
62. /\*\*
63. \* 出队列操作
64. \*
65. \* @return
66. \*/
67. private int deleteQueue() {
68. if (LQ.front == LQ.rear) {
69. System.out.println("队列为空！");
70. return -1;
71. }
72. QueueNode<Integer> newNode = LQ.front.next;
73. int result = newNode.data;
74. LQ.front.next = newNode.next;
75. if (LQ.rear == newNode) {
76. LQ.rear = LQ.front;
77. }
78. return result;
79. }
80. /\*\*
81. \* 测试
82. \*
83. \* @param args
84. \*/
85. public static void main(String[] args) {
86. MyLinkedQueue<Integer> myLinkedQueuey = new MyLinkedQueue<Integer>();
87. System.out.println("建立链式队列...");
88. myLinkedQueuey.initLinkedQueue();
89. int queueSize = -1;
90. try {
91. BufferedReader bufferedReader = new BufferedReader(new FileReader("queue.txt"));
92. try {
93. String[] strArr = bufferedReader.readLine().split(" ");
94. queueSize = Integer.parseInt(strArr[0]);
95. for (int i = 0; i < queueSize; i++) {
96. myLinkedQueuey.enterQueue(Integer.parseInt(strArr[i + 1]));
97. }
98. bufferedReader.close();
99. } catch (IOException e) {
100. // TODO Auto-generated catch block
101. e.printStackTrace();
102. }
103. } catch (FileNotFoundException e) {
104. // TODO Auto-generated catch block
105. e.printStackTrace();
106. }
107. System.out.println("打印链式队列...");
108. while (myLinkedQueuey.LQ.front != myLinkedQueuey.LQ.rear) {
109. System.out.print(myLinkedQueuey.deleteQueue() + " ");
110. }
111. }
112. }

## 3.3线性循环队列

* + 1. package com.ex.queue;
  1. public class MySCQueue<T> {
  2. // 顺序循环队列大小，可以自动扩容
  3. private static int nQueueSize = 10;
  4. /\*\*
  5. \* 定义顺序循环队列的数据结构
  6. \*
  7. \* @author lenovo
  8. \*
  9. \* @param <M>
  10. \*/
  11. private static final class SCQueue<M> {
  12. int front;
  13. int rear;
  14. M[] queue;
  15. public SCQueue() {
  16. // TODO Auto-generated constructor stub
  17. front = rear = 0;
  18. queue = (M[]) new Object[nQueueSize];
  19. }
  20. }
  21. /\*\*
  22. \* 定义一个顺序循环队列的对象
  23. \*/
  24. private SCQueue<T> SQ;
  25. /\*\*
  26. \* 初始化顺序循环队列SQ
  27. \*/
  28. private void initQueue() {
  29. SQ = new SCQueue<T>();
  30. SQ.rear = SQ.front = 0;
  31. }
  32. /\*\*
  33. \* 向顺序循环队列中插入元素，如果队列已满，队列将会自动扩容两倍
  34. \*
  35. \* @param e
  36. \*/
  37. private void EnterQueue(T e) {
  38. if (SQ.front == (SQ.rear + 1) % nQueueSize) {
  39. System.out.println("当前队列已满，队列将扩容为原来的两倍");
  40. enlargeQueue();
  41. }
  42. SQ.queue[SQ.rear] = e;
  43. SQ.rear = (SQ.rear + 1) % nQueueSize;
  44. }
  45. /\*\*
  46. \* 扩大队列的容量为原来的两倍
  47. \*/
  48. private void enlargeQueue() {
  49. T[] tmpQueue = (T[]) new Object[(SQ.rear + nQueueSize - SQ.front) % nQueueSize];
  50. for (int i = 0; i < tmpQueue.length; i++) {
  51. tmpQueue[i] = DeleteQueue();
  52. }
  53. SQ.front = SQ.rear = 0;
  54. nQueueSize \*= 2;
  55. SQ.queue = (T[]) new Object[nQueueSize];
  56. for (int i = 0; i < tmpQueue.length; i++) {
  57. EnterQueue(tmpQueue[i]);
  58. }
  59. }
  60. /\*\*
  61. \* 删除队列的对头元素并返回对头元素
  62. \*
  63. \* @return
  64. \*/
  65. private T DeleteQueue() {
  66. if (SQ.front == SQ.rear) {
  67. System.out.println("队列中已经为空:");
  68. return null;
  69. }
  70. T e = SQ.queue[SQ.front];
  71. SQ.front = (SQ.front + 1) % nQueueSize;
  72. return e;
  73. }
  74. /\*\*
  75. \* 打印队列元素
  76. \*/
  77. private void printSCQueue() {
  78. int i = SQ.front, j = 0;
  79. while (i != SQ.rear) {
  80. System.out.println("队列的第" + (++j) + "个的元素为:" + SQ.queue[i]);
  81. i = (i + 1) % nQueueSize;
  82. }
  83. }
  84. /\*\*
  85. \* 主函数
  86. \*
  87. \* @param args
  88. \*/
  89. public static void main(String[] args) {
  90. // TODO Auto-generated method stub
  91. MySCQueue<Character> queue = new MySCQueue<Character>();
  92. queue.initQueue();
  93. for (int i = 0; i < 26; i++) {
  94. queue.EnterQueue((char) (i + 65));
  95. }
  96. System.out.println("出队列前队列中的元素为:");
  97. queue.printSCQueue();
  98. queue.DeleteQueue();
  99. System.out.println("对头元素出队列后队列中的元素为:");
  100. queue.printSCQueue();
  101. }
  102. }

## 3.4线性双端队列

1. **package** com.ex.queue;
2. **import** java.util.Random;
3. **public** **class** MyDQueue<T> {
4. // 定义双端队列的大小
5. **private** **static** **int** *nQueueSize* = 100;
6. **private** **static** **final** **class** DQueue<M> {
7. **int** end1;
8. **int** end2;
9. M[] queue;
10. }
11. /\*\*
12. \* 定义一个枚举结构来表示左右队列
13. \*
14. \* **@author** lenovo
15. \*
16. \*/
17. **static** **enum** QType {
18. ***Left*** {
19. **void** tellDirection(**boolean** nFlag) {
20. **if** (nFlag == **true**) {
21. System.***out***.println("元素从左边入队列:");
22. } **else** {
23. System.***out***.println("元素从左边出队列：");
24. }
25. }
26. },
27. ***Right*** {
28. **void** tellDirection(**boolean** nFlag) {
29. **if** (nFlag == **true**) {
30. System.***out***.println("元素从右边入队列:");
31. } **else** {
32. System.***out***.println("元素从右边出队列：");
33. }
34. }
35. };
36. **abstract** **void** tellDirection(**boolean** nFlag);
37. }
38. // 定义一个双端队列对象
39. **private** DQueue<T> DQ;
40. **private** **void** initDQ() {
41. DQ = **new** DQueue<T>();
42. DQ.end2 = *nQueueSize* / 2;
43. DQ.end1 = DQ.end2 - 1;
44. DQ.queue = (T[]) **new** Object[*nQueueSize*];
45. }
46. /\*\*
47. \* 元素入队
48. \*
49. \* **@param** e
50. \* **@param** nFlag
51. \*/
52. **private** **void** enterQueue(T e, QType eFlag) {
53. eFlag.tellDirection(**true**);
54. **switch** (eFlag) {
55. **case** ***Left***:
56. **if** (DQ.end1 != DQ.end2) {
57. DQ.queue[DQ.end1] = e;
58. DQ.end1 = (DQ.end1 - 1) % *nQueueSize*;
59. }
60. **break**;
61. **case** ***Right***:
62. **if** (DQ.end1 != DQ.end2) {
63. DQ.queue[DQ.end2] = e;
64. DQ.end2 = (DQ.end2 + 1) % *nQueueSize*;
65. }
66. **break**;
67. **default**:
68. System.***out***.println("非法进入队列！");
69. **break**;
70. }
71. }
72. /\*\*
73. \* 出队列
74. \*
75. \* **@param** nFlag
76. \* **@return**
77. \*/
78. **private** T deleteQueue(QType eFlag) {
79. T ret = **null**;
80. eFlag.tellDirection(**false**);
81. **switch** (eFlag) {
82. **case** ***Left***:
83. **if** (DQ.end1 + 1 != DQ.end2) {
84. DQ.end1 = (DQ.end1 + 1) % *nQueueSize*;
85. ret = DQ.queue[DQ.end1];
86. }
87. **break**;
88. **case** ***Right***:
89. **if** (DQ.end1 + 1 != DQ.end2) {
90. DQ.end2 = (DQ.end2 - 1) % *nQueueSize*;
91. ret = DQ.queue[DQ.end2];
92. }
93. **break**;
94. **default**:
95. System.***out***.println("出队列非法！");
96. **break**;
97. }
98. **return** ret;
99. }
100. /\*\*
101. \* 打印双端队列
102. \*/
103. **private** **void** printDQ() {
104. **int** i = 0;
105. **int** nEnd1 = DQ.end1, nEnd2 = DQ.end2;
106. **while** (DQ.end1 + 1 != DQ.end2) {
107. DQ.end1 = (DQ.end1 + 1) % *nQueueSize*;
108. System.***out***.println("左边对列中的第" + (++i) + "个元素为:" + DQ.queue[DQ.end1]);
109. DQ.end2 = (DQ.end2 - 1) % *nQueueSize*;
110. System.***out***.println("右边对列中的第" + (++i) + "个元素为:" + DQ.queue[DQ.end2]);
111. }
112. DQ.end1 = nEnd1;
113. DQ.end2 = nEnd2;
114. }
115. /\*\*
116. \* 主函数
117. \*
118. \* **@param** args
119. \*/
120. **public** **static** **void** main(String[] args) {
121. // **TODO** Auto-generated method stub
122. MyDQueue<Double> dQueue = **new** MyDQueue<Double>();
123. dQueue.initDQ();
124. Random random = **new** Random();
125. **for** (**int** i = 0; i < 12; i++) {
126. dQueue.enterQueue(i \* 1.0, QType.*values*()[random.nextInt(QType.*values*().length)]);
127. }
128. System.***out***.println("出队列前队列中的元素为:");
129. dQueue.printDQ();
130. System.***out***.println("左边队列对头元素出队列，右边队列队尾元素出队列后的队列为:");
131. dQueue.deleteQueue(QType.***Left***);
132. dQueue.deleteQueue(QType.***Right***);
133. dQueue.printDQ();
134. }
135. }

# 4.二叉树（二叉链表的实现）

1. package com.ex.tree;
2. import java.io.BufferedReader;
3. import java.io.FileNotFoundException;
4. import java.io.FileReader;
5. import java.io.IOException;
6. import java.util.Scanner;
7. import com.ex.graph.LinkedStack;
8. public class BinaryTree {
9. // 定义根节点
10. private BinaryNode<Character> root;
11. // 定义一个链式栈用来存储访问过的节点
12. private LinkedStack<BinaryNode<Character>> linkedStack = null;
13. /\*\*
14. \* 初始化
15. \*/
16. private void initBinaryTree() {
17. this.root = null;
18. linkedStack = new LinkedStack<BinaryNode<Character>>();
19. }
20. /\*\*
21. \* 创建二叉树
22. \*
23. \* @param treeNode
24. \* @return
25. \*/
26. private BinaryNode<Character> createBinaryTree(BinaryNode<Character> treeNode, BufferedReader bufReader)
27. throws IOException {
28. Character inputCh = bufReader.readLine().charAt(0);
29. if (inputCh.equals('#'))
30. return null;
31. else {
32. if (treeNode == null)
33. treeNode = new BinaryNode<Character>(inputCh);
34. treeNode.left = createBinaryTree(treeNode.left, bufReader);
35. treeNode.right = createBinaryTree(treeNode.right, bufReader);
36. return treeNode;
37. }
38. }
39. /\*\*
40. \* 插入左子树
41. \*
42. \* @param p
43. \* @param newEle
44. \*/
45. private void insertLeftChild(BinaryNode<Character> p, Character newEle) {
46. if (p != null) {
47. BinaryNode<Character> newNode = new BinaryNode<Character>(newEle);
48. newNode.right = p.left;
49. p.left = newNode;
50. }
51. }
52. /\*\*
53. \* 插入右子树
54. \*
55. \* @param p
56. \* @param newEle
57. \*/
58. private void insertRightChild(BinaryNode<Character> p, Character newEle) {
59. if (p != null) {
60. BinaryNode<Character> newNode = new BinaryNode<Character>(newEle);
61. newNode.right = p.right;
62. p.right = newNode;
63. }
64. }
65. /\*\*
66. \* 前序遍历
67. \*/
68. private void preOrderTraverse() {
69. linkedStack.clear();
70. BinaryNode<Character> p = root;
71. while (p != null || linkedStack.isEmpty() == false) {
72. while (p != null) {
73. System.out.print(p.element + " ");
74. linkedStack.push(p);
75. p = p.left;
76. }
77. if (linkedStack.isEmpty() != true) {
78. p = linkedStack.pop();
79. p = p.right;
80. }
81. }
82. System.out.print("\n");
83. }
84. /\*\*
85. \* 中序遍历
86. \*/
87. private void inOrderTraverse() {
88. linkedStack.clear();
89. BinaryNode<Character> p = root;
90. while (p != null || linkedStack.isEmpty() == false) {
91. while (p != null) {
92. linkedStack.push(p);
93. p = p.left;
94. }
95. if (linkedStack.isEmpty() != true) {
96. p = linkedStack.pop();
97. System.out.print(p.element + " ");
98. p = p.right;
99. }
100. }
101. System.out.print("\n");
102. }
103. /\*\*
104. \* 后序遍历
105. \*/
106. private void postOrderTraverse() {
107. linkedStack.clear();
108. BinaryNode<Character> p = root;
109. BinaryNode<Character> q = null;
110. while (p != null || linkedStack.isEmpty() == false) {
111. while (p != null) {
112. linkedStack.push(p);
113. p = p.left;
114. }
115. if (linkedStack.isEmpty() != true) {
116. p = linkedStack.peekTop();
117. if (p.right == null || p.right == q) {
118. System.out.print(p.element + " ");
119. q = p;
120. p = null;
121. linkedStack.pop();
122. } else {
123. p = p.right;
124. }
125. }
126. }
127. System.out.print("\n");
128. }
129. /\*\*
130. \* @param args
131. \*/
132. public static void main(String[] args) {
133. // TODO Auto-generated method stub
134. BinaryTree binaryTree = new BinaryTree();
135. binaryTree.initBinaryTree();
136. System.out.println("创建二叉树...");
137. try {
138. BufferedReader bufferedReader = new BufferedReader(new FileReader("tree.txt"));
139. try {
140. binaryTree.root = binaryTree.createBinaryTree(binaryTree.root, bufferedReader);
141. bufferedReader.close();
142. } catch (IOException e) {
143. // TODO Auto-generated catch block
144. e.printStackTrace();
145. }
146. } catch (FileNotFoundException e) {
147. // TODO Auto-generated catch block
148. e.printStackTrace();
149. }
150. System.out.println("二叉树前序遍历为...");
151. binaryTree.preOrderTraverse();
152. System.out.println("二叉树中序遍历为...");
153. binaryTree.inOrderTraverse();
154. System.out.println("二叉树后序遍历为...");
155. binaryTree.postOrderTraverse();
156. System.out.println("二叉树示例完毕!");
157. }
158. }

# 5.图(邻接链表表示法)和最短路径Dijkstra算法

1. package com.ex.graph;
2. /\*\*
3. \*
4. \* @author lenovo 定义图的邻接链表数据结构实现
5. \* @param <T>
6. \*/
7. public class AdjGraph<T> {
8. private static final int MAX\_SIZE = 1000;
9. VNode<T>[] vertexNodes;
10. int vexNum, arcNum;
11. GraphKind graphKind;
12. @SuppressWarnings("unchecked")
13. public AdjGraph() {
14. vertexNodes = (VNode<T>[]) new VNode[MAX\_SIZE];
15. vexNum = arcNum = 0;
16. graphKind = GraphKind.DG;
17. }
18. }
19. /\*\*
20. \* 定义弧节点
21. \*/
22. class ArcNode {
23. int adjvex;
24. ArcNode nextArcNode;
25. int weight;
26. public ArcNode(int adjvex, ArcNode nextArcNode, int weight) {
27. this.adjvex = adjvex;
28. this.nextArcNode = nextArcNode;
29. this.weight = weight;
30. }
31. }
32. /\*\*
33. \* 定义顶点结点
34. \*/
35. class VNode<T> {
36. T data;
37. ArcNode firstArcNode;
38. boolean know;
39. int dist;
40. VNode<T> nearNode;
41. }
42. /\*\*
43. \* 定义图 类型
44. \*/
45. enum GraphKind {
46. DG, DN, UG, UN
47. }
48. package com.ex.graph;
49. import java.io.BufferedReader;
50. import java.io.File;
51. import java.io.FileInputStream;
52. import java.io.FileNotFoundException;
53. import java.io.IOException;
54. import java.io.InputStream;
55. import java.io.InputStreamReader;
56. import java.util.Scanner;
57. public class DGraph {
58. // 定义无穷大值作为路径不可达的标志
59. private static final int INFINITY = (int) 1e9;
60. // 定义一个图的对象
61. private AdjGraph<Character> graph;
62. /\*\*
63. \* 创建图
64. \*/
65. private void createGraph() {
66. if (graph == null)
67. graph = new AdjGraph<Character>();
68. graph.graphKind = GraphKind.DG;
69. BufferedReader bufferedReader = null;
70. try {
71. bufferedReader = new BufferedReader(
72. new InputStreamReader(new FileInputStream(new File("data.txt").getAbsoluteFile())));
73. try {
74. if (bufferedReader.ready() == true) {
75. String[] graphInfoArr = bufferedReader.readLine().split(" ");
76. graph.vexNum = Integer.parseInt(graphInfoArr[0]);
77. graph.arcNum = Integer.parseInt(graphInfoArr[1]);
78. String[] vertexArr = bufferedReader.readLine().split(" ");
79. for (int i = 0; i < graph.vexNum; i++) {
80. graph.vertexNodes[i] = new VNode<Character>();
81. graph.vertexNodes[i].data = Character.valueOf(vertexArr[i].charAt(0));
82. graph.vertexNodes[i].firstArcNode = null;
83. graph.vertexNodes[i].know = false;
84. graph.vertexNodes[i].dist = INFINITY;
85. graph.vertexNodes[i].nearNode = null;
86. }
87. Character ch1, ch2;
88. ArcNode arcNode = null;
89. for (int i = 0; i < graph.arcNum; i++) {
90. String[] arcInfoArr = bufferedReader.readLine().split(" ");
91. ch1 = Character.valueOf(arcInfoArr[0].charAt(0));
92. ch2 = Character.valueOf(arcInfoArr[1].charAt(0));
93. int u = locateVertex(ch1);
94. int v = locateVertex(ch2);
95. arcNode = new ArcNode(v, graph.vertexNodes[u].firstArcNode, Integer.parseInt(arcInfoArr[2]));
96. graph.vertexNodes[u].firstArcNode = arcNode;
97. }
98. }
99. bufferedReader.close();
100. } catch (IOException e) {
101. // TODO: handle exception
102. e.printStackTrace();
103. }
104. } catch (FileNotFoundException e) {
105. // TODO Auto-generated catch block
106. e.printStackTrace();
107. }
108. }
109. /\*\*
110. \* 获取某个节点在图中的位置
111. \*
112. \* @param v
113. \* @return
114. \*/
115. private int locateVertex(Character v) {
116. for (int i = 0; i < graph.vexNum; i++) {
117. if (graph.vertexNodes[i].data.equals(v)) {
118. return i;
119. }
120. }
121. return 0;
122. }
123. /\*\*
124. \* 打印图
125. \*/
126. private void displayGraph() {
127. ArcNode p;
128. System.out.printf("总共有%d个顶点!\n", graph.vexNum);
129. for (int i = 0; i < graph.vexNum; i++) {
130. System.out.println(graph.vertexNodes[i].data);
131. }
132. System.out.printf("总共有%d条边!\n", graph.arcNum);
133. for (int i = 0; i < graph.vexNum; i++) {
134. p = graph.vertexNodes[i].firstArcNode;
135. while (p != null) {
136. System.out.printf("%s->%s\t", graph.vertexNodes[i].data, graph.vertexNodes[p.adjvex].data);
137. p = p.nextArcNode;
138. }
139. System.out.println();
140. }
141. }
142. /\*\*
143. \* 主函数
144. \*
145. \* @param args
146. \*/
147. public static void main(String[] args) {
148. DGraph dGraph = new DGraph();
149. dGraph.createGraph();
150. dGraph.displayGraph();
151. System.out.println("请输入起点节点...");
152. Scanner scanner = new Scanner(System.in);
153. Character s = Character.valueOf(scanner.next().charAt(0));
154. int s0 = dGraph.locateVertex(s);
155. dGraph.dijkstra(s0);
156. for (int i = 0; i < dGraph.graph.vexNum; i++) {
157. if (i != s0) {
158. System.out.printf("%s->%s的最短路径为:%d\t", dGraph.graph.vertexNodes[s0].data,
159. dGraph.graph.vertexNodes[i].data, dGraph.graph.vertexNodes[i].dist);
160. dGraph.showPath(s0, i);
161. }
162. }
163. }
164. /\*\*
165. \* 单源最短路径
166. \*
167. \* @param s
168. \*/
169. private void dijkstra(int s0) {
170. ArcNode arcNode = null;
171. // 初始化各个节点距离初始源节点的路径长度
172. for (int i = 0; i < graph.vexNum; i++) {
173. arcNode = graph.vertexNodes[s0].firstArcNode;
174. while (arcNode != null) {
175. if (arcNode.adjvex == i) {
176. graph.vertexNodes[i].dist = arcNode.weight;
177. graph.vertexNodes[i].nearNode = graph.vertexNodes[s0];
178. break;
179. }
180. arcNode = arcNode.nextArcNode;
181. }
182. if (arcNode == null) {
183. graph.vertexNodes[i].dist = INFINITY;
184. graph.vertexNodes[i].nearNode = null;
185. }
186. }
187. graph.vertexNodes[s0].dist = 0;
188. for (int i = 0; i < graph.vexNum; i++) {
189. int m = -1;
190. int min\_dist = INFINITY;
191. // 寻找出还未访问过的最短路径点
192. for (int j = 0; j < graph.vexNum; j++) {
193. if (graph.vertexNodes[j].know == false && graph.vertexNodes[j].dist < min\_dist) {
194. min\_dist = graph.vertexNodes[j].dist;
195. m = j;
196. }
197. }
198. if (m == -1)
199. continue;
200. graph.vertexNodes[m].know = true;
201. // 根据已找出最短路径的节点修正
202. for (int j = 0; j < graph.vexNum; j++) {
203. arcNode = graph.vertexNodes[m].firstArcNode;
204. if (graph.vertexNodes[j].know == false) {
205. while (arcNode != null) {
206. if (arcNode.adjvex == j) {
207. if (arcNode.weight > 0
208. && graph.vertexNodes[j].dist > graph.vertexNodes[m].dist + arcNode.weight) {
209. graph.vertexNodes[j].dist = graph.vertexNodes[m].dist + arcNode.weight;
210. graph.vertexNodes[j].nearNode = graph.vertexNodes[m];
211. }
212. }
213. arcNode = arcNode.nextArcNode;
214. }
215. }
216. }
217. }
218. }
219. /\*\*
220. \* 显示最短路径
221. \*
222. \* @param v1
223. \* @param v2
224. \*/
225. private void showPath(int v0, int v) {
226. LinkedStack<Character> linkedStack = new LinkedStack<Character>();
227. boolean bCanReach = true;
228. while (v != v0) {
229. linkedStack.push(graph.vertexNodes[v].data);
230. if (graph.vertexNodes[v].dist == INFINITY) {
231. bCanReach = false;
232. break;
233. }
234. if (graph.vertexNodes[v].nearNode != null)
235. v = locateVertex(graph.vertexNodes[v].nearNode.data);
236. }
237. linkedStack.push(graph.vertexNodes[v0].data);
238. Character e = null;
239. while ((e = linkedStack.pop()) != null) {
240. System.out.print(e + " ");
241. }
242. if (bCanReach == false)
243. System.out.print("终点不可达！");
244. System.out.println();
245. }
246. }

# 6.排序算法

* 1. package com.ex.sort;
  2. import java.util.Arrays;
  3. import java.util.Scanner;
  4. /\*\*
  5. \* 排序算法汇总，默认按照升序进行排序 待排序数组为：{2,16,9,8,11,33,5,4} 具体排序算法见注释部分
  6. \*
  7. \* @author lenovo write:Sep 3,2015 21:05 in School of Remote Sensing and
  8. \* Information Technology in Wuhan University
  9. \*/
  10. public class SortAlgorithms {
  11. static enum ESort {
  12. DI {
  13. void print(int[] nArray) {
  14. System.out.println("直接插入排序的结果为:" + Arrays.toString(nArray));
  15. }
  16. },
  17. HF {
  18. void print(int[] nArray) {
  19. System.out.println("折半查找插入排序的结果为:" + Arrays.toString(nArray));
  20. }
  21. },
  22. SH {
  23. void print(int[] nArray) {
  24. System.out.println("希尔排序的结果为:" + Arrays.toString(nArray));
  25. }
  26. },
  27. SS {
  28. void print(int[] nArray) {
  29. System.out.println("简单选择排序的结果为:" + Arrays.toString(nArray));
  30. }
  31. },
  32. BS {
  33. void print(int[] nArray) {
  34. System.out.println("冒泡排序的结果为:" + Arrays.toString(nArray));
  35. }
  36. };
  37. abstract void print(int[] nArray);
  38. }
  39. /\*\*
  40. \* 待排序数组
  41. \*/
  42. private static int[] nArr = { 2, 16, 9, 8, 11, 33, 5, 4 };
  43. /\*\*
  44. \* 入口函数
  45. \*
  46. \* @param args
  47. \*/
  48. public static void main(String[] args) {
  49. // TODO Auto-generated method stub
  50. System.out.println("请输入排序方式:DI表示直接插入排序，HF表示折半插入排序，SH表示希尔排序，SS表示简单选择排序，BS表示冒泡排序.");
  51. ESort sortType;
  52. Scanner scanner = new Scanner(System.in);
  53. sortType = ESort.valueOf(scanner.next());
  54. switch (sortType) {
  55. case DI:
  56. insertSort();
  57. break;
  58. case HF:
  59. halfFindSort();
  60. break;
  61. case SH:
  62. shellInsertSort();
  63. break;
  64. case SS:
  65. simpleSelectSort();
  66. break;
  67. case BS:
  68. bubbleSort();
  69. break;
  70. default:
  71. break;
  72. }
  73. sortType.print(nArr);
  74. }
  75. /\*\*
  76. \* 直接插入排序
  77. \*/
  78. private static void insertSort() {
  79. int nTemp = 0;
  80. for (int i = 0; i < nArr.length - 1; i++) {
  81. nTemp = nArr[i + 1];
  82. int j = i;
  83. while (j > -1 && nTemp < nArr[j]) {
  84. nArr[j + 1] = nArr[j];
  85. j--;
  86. }
  87. nArr[j + 1] = nTemp;
  88. }
  89. }
  90. /\*\*
  91. \* 折半插入排序
  92. \*/
  93. private static void halfFindSort() {
  94. int temp, low, high, mid;
  95. int i, j;
  96. for (i = 0; i < nArr.length - 1; i++) {
  97. temp = nArr[i + 1];
  98. low = 0;
  99. high = i;
  100. while (low <= high) {
  101. mid = (low + high) / 2;
  102. if (nArr[mid] > temp) {
  103. high = mid - 1;
  104. } else {
  105. low = mid + 1;
  106. }
  107. }
  108. for (j = i; j >= low; j--) {
  109. nArr[j + 1] = nArr[j];
  110. }
  111. nArr[low] = temp;
  112. }
  113. }
  114. /\*\*
  115. \* 希尔排序
  116. \*/
  117. private static void shellInsertSort() {
  118. for (int i = nArr.length / 2 - 1; i >= 0; i--) {
  119. shellInsert(2 \* i + 1);
  120. }
  121. }
  122. private static void shellInsert(int delta) {
  123. int nTemp, j;
  124. for (int i = delta; i < nArr.length; i++) {
  125. if (i == 1) {
  126. System.out.println("最后一趟排序！");
  127. }
  128. if (nArr[i] < nArr[i - delta]) {
  129. nTemp = nArr[i];
  130. for (j = i - delta; j > 0 && nArr[j] > nTemp; j -= delta) {
  131. nArr[j + delta] = nArr[j];
  132. }
  133. nArr[j + delta] = nTemp;
  134. }
  135. }
  136. }
  137. /\*\*
  138. \* 简单选择排序
  139. \*/
  140. private static void simpleSelectSort() {
  141. int nTemp = 0;
  142. for (int i = 0; i < nArr.length - 1; i++) {
  143. for (int j = i + 1; j < nArr.length; j++) {
  144. if (nArr[i] > nArr[j]) {
  145. nTemp = nArr[i];
  146. nArr[i] = nArr[j];
  147. nArr[j] = nTemp;
  148. }
  149. }
  150. }
  151. }
  152. /\*\*
  153. \* 冒泡排序
  154. \*/
  155. private static void bubbleSort() {
  156. int nTemp = 0;
  157. for (int i = 0; i < nArr.length - 1; i++) {
  158. for (int j = 0; j < nArr.length - i - 1; j++) {
  159. if (nArr[j] > nArr[j + 1]) {
  160. nTemp = nArr[j];
  161. nArr[j] = nArr[j + 1];
  162. nArr[j + 1] = nTemp;
  163. }
  164. }
  165. }
  166. }
  167. }

# 7.常见考题（更新中…）

## 7.1单链表反转

1. package com.ex.problems;
2. public class LinkedListReverse {
3. /\*\*
4. \* 单链表的数据结构
5. \*
6. \* @author lenovo
7. \*
8. \* @param <T>
9. \*/
10. static final class LNode<T> {
11. T data;
12. LNode<T> next;
13. /\*\*
14. \* 注明，构造函数不需要带额外的类型参数，不管是C++还是Java，注意啦，带额外参数的是类类型
15. \*
16. \* @param data
17. \* @param next
18. \*/
19. public LNode(T data, LNode<T> next) {
20. this.data = data;
21. this.next = next;
22. }
23. }
24. // 定义单链表的头指针
25. private static LNode<Integer> head;
26. // 初始化单链表
27. private static void initLinkedList() {
28. head = new LNode<Integer>(null, null);
29. }
30. /\*\*
31. \* 创建单链表
32. \*
33. \* @param nArr
34. \*/
35. private static void createLinkedList(int... nArr) {
36. LNode<Integer> prev = head, cur;
37. for (int i = 0; i < nArr.length; i++) {
38. cur = new LNode<Integer>(nArr[i], null);
39. prev.next = cur;
40. prev = cur;
41. }
42. }
43. /\*\*
44. \* 打印单链表
45. \*/
46. public static void printLinkedList() {
47. LNode<Integer> cur = head;
48. while (cur.next != null) {
49. System.out.println(cur.next.data);
50. cur = cur.next;
51. }
52. }
53. /\*\*
54. \* 反转单链表
55. \*/
56. private static void reverseLinkedList() {
57. if (head == null) {
58. return;
59. }
60. LNode<Integer> prev, cur, next;
61. prev = head;
62. cur = prev.next;
63. while (cur != null) {
64. next = cur.next;
65. if (prev != head) // 当时收个节点时候直接将next域置为空
66. cur.next = prev;
67. else
68. cur.next = null;
69. prev = cur;
70. cur = next;
71. }
72. head.next = prev;
73. }
74. /\*\*
75. \* 主函数
76. \*
77. \* @param args
78. \*/
79. public static void main(String[] args) {
80. // TODO Auto-generated method stub
81. initLinkedList();
82. createLinkedList(1, 3, 5, 6, 7);
83. System.out.println("反正前链表为:");
84. printLinkedList();
85. System.out.println("反转后链表为:");
86. reverseLinkedList();
87. printLinkedList();
88. }
89. }