*Федеральное государственное бюджетное образовательное учреждение* *высшего образования*

|  |  |
| --- | --- |
|  | ***«Московский государственный технический университет  имени Н.Э. Баумана»***  ***(национальный исследовательский университет)***  ***(МГТУ им. Н.Э. Баумана)*** |

ФАКУЛЬТЕТ \_\_\_\_ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ\_\_\_\_\_\_\_\_

КАФЕДРА \_\_\_\_\_\_Компьютерные Системы и сети (ИУ6)\_\_\_\_\_\_\_\_\_\_\_

**Отчет**

**по лабораторной работе № 8**

**Название лабораторной работы: Программирование с использованием классов в С++. Создание контейнеров.**

**Дисциплина: Объектно-ориентированное программирование**

Студент гр. ИУ6-22Б  **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ А. П. Плютто**



(Подпись, дата) (И.О. Фамилия)



Преподаватель  **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ О. А. Веселовская**

(Подпись, дата) (И.О. Фамилия)

Москва, 2023

Задание

Моделировать множество, в качестве элементов которого могут использоваться числа заданного диапазона и символы (стандартный тип "множество" не использовать). Операции: добавление элемента, удаление элемента, печать элементов и проверка вхождения. Создать класс-потомок, который содержит функцию определения мощности множества. Тестировать полученную модель. Примечание: сначала реализовать и отладить структуру данных как класс, после чего преобразовать класс в шаблон.

В отчете представить диаграмму классов и обосновать выбранную структуру представления данных.

Решение

Для начала реализуем все требования без использования шаблонов

Создадим класс-родитель и добавим в него поле для организации односвязного списка и метод печати.

1. class Elem

2. {

3. public:

4. Elem \*next;

5. Elem(Elem \*next = nullptr) : next(next){};

6. virtual void print();

7. };

8.

9. void Elem::print()

10. {

11. cout << "parent elem class";

12. };

13.

Создадим 2 класса-потомка для целочисленных и символьных элементов. Переопределим метод вывода в каждом из классов. Добавим поле соответствующего типа в protected и функцию, которая возвращает значение элемента.

1. class ElemInt : public Elem

2. {

3. protected:

4. int i;

5.

6. public:

7. ElemInt(int i, Elem \*next = nullptr) : Elem(next), i(i){};

8. void print();

9. int getElem();

10. };

11.

12. class ElemChar : public Elem

13. {

14. protected:

15. char c;

16.

17. public:

18. ElemChar(char c, Elem \*next = nullptr) : Elem(next), c(c){};

19. void print();

20. char getElem();

21. };

22. void ElemInt::print()

23. {

24. cout << "(int)" << i << "\n";

25. };

26.

27. int ElemInt::getElem()

28. {

29. return i;

30. };

31.

32. void ElemChar::print()

33. {

34. cout << "(char)" << c << "\n";

35. };

36.

37. char ElemChar::getElem()

38. {

39. return c;

40. };

41.

Добавим функцию, которая проверяет тип элемента.

1. bool fl(Elem \*c)

2. {

3. // return (static\_cast<void \*>(dynamic\_cast<ElemInt \*>(c)) >

4. // static\_cast<void \*>(dynamic\_cast<ElemChar \*>(c)));

5. return (typeid(\*c) == typeid(ElemInt));

6. }

7.

Создадим класс для множества и внесем в него все необходимые функции.

1. class set

2. {

3. protected:

4. Elem \*st = nullptr;

5.

6. public:

7. set();

8. ~set();

9. bool contains(Elem \*cn);

10. void add(Elem \*nw);

11. void del(Elem \*dt);

12. void print();

13. };

14. set::set()

15. {

16. cout << "Set init\n";

17. };

18.

19. set::~set()

20. {

21. cout << "Set delete\n";

22. };

23.

24. bool set::contains(Elem \*cn)

25. {

26. Elem \*tmp = st;

27. bool isInt = fl(cn);

28.

29. while (tmp)

30. {

31. bool isTmpInt = fl(tmp);

32. if (isInt == isTmpInt)

33. {

34. if (isInt)

35. {

36. if (dynamic\_cast<ElemInt \*>(cn)->getElem() ==

37. dynamic\_cast<ElemInt \*>(tmp)->getElem())

38. return true;

39. }

40. else

41. {

42. if (dynamic\_cast<ElemChar \*>(cn)->getElem() ==

43. dynamic\_cast<ElemChar \*>(tmp)->getElem())

44. return true;

45. }

46. }

47. tmp = tmp->next;

48. };

49. return false;

50. };

51. void set::add(Elem \*nw)

52. {

53. if (!contains(nw))

54. {

55. nw->next = st;

56. st = nw;

57. }

58. };

59. void set::del(Elem \*dt)

60. {

61. Elem \*tmp = st;

62. Elem \*tm = st;

63. bool isInt = fl(dt);

64.

65. while (tmp)

66. {

67. bool isTmpInt = fl(tmp);

68. if (isInt == isTmpInt)

69. {

70. if (isInt)

71. {

72. if (dynamic\_cast<ElemInt \*>(dt)->getElem() ==

73. dynamic\_cast<ElemInt \*>(tmp)->getElem())

74. {

75. tm->next = tmp->next;

76. tmp = tm;

77. return;

78. }

79. }

80. else

81. {

82. if (dynamic\_cast<ElemChar \*>(dt)->getElem() ==

83. dynamic\_cast<ElemChar \*>(tmp)->getElem())

84. {

85. tm->next = tmp->next;

86. tmp = tm;

87. return;

88. }

89. }

90. }

91. tm = tmp;

92. tmp = tmp->next;

93. }

94. };

95. void set::print()

96. {

97. Elem \*tmp = st;

98. while (tmp)

99. {

100. tmp->print();

101. tmp = tmp->next;

102. };

103. };

104.

Создадим класс-наследник и наследуем класс множества.

1. class setP : public set

2. {

3. public:

4. int power();

5. };

6. int setP::power()

7. {

8. Elem \*tmp = st;

9. int count{};

10. while (tmp)

11. {

12. count++;

13. tmp = tmp->next;

14. };

15. return count;

16. }

17.

Реализуем тестирующую программу.

1. int main()

2. {

3. setP gh;

4.

5. ElemInt int1(10);

6. ElemInt int2(11);

7. ElemInt int3(12);

8. ElemInt int4(13);

9. ElemInt int5(14);

10. ElemChar char1('a');

11. ElemChar char2('b');

12. ElemChar char3('c');

13. ElemChar char4('d');

14. ElemChar char5('e');

15.

16. gh.add(&int1);

17. gh.add(&int2);

18. gh.add(&char1);

19. gh.add(&char2);

20. gh.add(&int3);

21. gh.add(&int4);

22. gh.add(&int5);

23. gh.add(&char3);

24. gh.add(&char4);

25. gh.add(&char5);

26.

27. gh.print();

28. cout << "int3 (12) in set: " << gh.contains(&int3) << "\n";

29. gh.del(&int3);

30. cout << "int3 (12) in set: " << gh.contains(&int3) << "\n";

31. gh.print();

32. cout << gh.power() << "\n";

33. return 0;

34. }

35.

Приведем полный код программы.

1. #include "iostream"

2. using std::cout;

3.

4. class Elem

5. {

6. public:

7. Elem \*next;

8. Elem(Elem \*next = nullptr) : next(next){};

9. virtual void print();

10. };

11.

12. class ElemInt : public Elem

13. {

14. protected:

15. int i;

16.

17. public:

18. ElemInt(int i, Elem \*next = nullptr) : Elem(next), i(i){};

19. void print();

20. int getElem();

21. };

22.

23. class ElemChar : public Elem

24. {

25. protected:

26. char c;

27.

28. public:

29. ElemChar(char c, Elem \*next = nullptr) : Elem(next), c(c){};

30. void print();

31. char getElem();

32. };

33.

34. bool fl(Elem \*c)

35. {

36. // return (static\_cast<void \*>(dynamic\_cast<ElemInt \*>(c)) >

37. // static\_cast<void \*>(dynamic\_cast<ElemChar \*>(c)));

38. return (typeid(\*c) == typeid(ElemInt));

39. }

40.

41. class set

42. {

43. protected:

44. Elem \*st = nullptr;

45.

46. public:

47. set();

48. ~set();

49. bool contains(Elem \*cn);

50. void add(Elem \*nw);

51. void del(Elem \*dt);

52. void print();

53. };

54.

55. class setP : public set

56. {

57. public:

58. int power();

59. };

60.

61. void Elem::print()

62. {

63. cout << "parent elem class";

64. };

65.

66. void ElemInt::print()

67. {

68. cout << "(int)" << i << "\n";

69. };

70.

71. int ElemInt::getElem()

72. {

73. return i;

74. };

75.

76. void ElemChar::print()

77. {

78. cout << "(char)" << c << "\n";

79. };

80.

81. char ElemChar::getElem()

82. {

83. return c;

84. };

85.

86. set::set()

87. {

88. cout << "Set init\n";

89. };

90.

91. set::~set()

92. {

93. cout << "Set delete\n";

94. };

95.

96. bool set::contains(Elem \*cn)

97. {

98. Elem \*tmp = st;

99. bool isInt = fl(cn);

100.

101. while (tmp)

102. {

103. bool isTmpInt = fl(tmp);

104. if (isInt == isTmpInt)

105. {

106. if (isInt)

107. {

108. if (dynamic\_cast<ElemInt \*>(cn)->getElem() ==

109. dynamic\_cast<ElemInt \*>(tmp)->getElem())

110. return true;

111. }

112. else

113. {

114. if (dynamic\_cast<ElemChar \*>(cn)->getElem() ==

115. dynamic\_cast<ElemChar \*>(tmp)->getElem())

116. return true;

117. }

118. }

119. tmp = tmp->next;

120. };

121. return false;

122. };

123. void set::add(Elem \*nw)

124. {

125. if (!contains(nw))

126. {

127. nw->next = st;

128. st = nw;

129. }

130. };

131. void set::del(Elem \*dt)

132. {

133. Elem \*tmp = st;

134. Elem \*tm = st;

135. bool isInt = fl(dt);

136.

137. while (tmp)

138. {

139. bool isTmpInt = fl(tmp);

140. if (isInt == isTmpInt)

141. {

142. if (isInt)

143. {

144. if (dynamic\_cast<ElemInt \*>(dt)->getElem() ==

145. dynamic\_cast<ElemInt \*>(tmp)->getElem())

146. {

147. tm->next = tmp->next;

148. tmp = tm;

149. return;

150. }

151. }

152. else

153. {

154. if (dynamic\_cast<ElemChar \*>(dt)->getElem() ==

155. dynamic\_cast<ElemChar \*>(tmp)->getElem())

156. {

157. tm->next = tmp->next;

158. tmp = tm;

159. return;

160. }

161. }

162. }

163. tm = tmp;

164. tmp = tmp->next;

165. }

166. };

167. void set::print()

168. {

169. Elem \*tmp = st;

170. while (tmp)

171. {

172. tmp->print();

173. tmp = tmp->next;

174. };

175. };

176.

177. int setP::power()

178. {

179. Elem \*tmp = st;

180. int count{};

181. while (tmp)

182. {

183. count++;

184. tmp = tmp->next;

185. };

186. return count;

187. }

188.

189. int main()

190. {

191. setP gh;

192.

193. ElemInt int1(10);

194. ElemInt int2(11);

195. ElemInt int3(12);

196. ElemInt int4(13);

197. ElemInt int5(14);

198. ElemChar char1('a');

199. ElemChar char2('b');

200. ElemChar char3('c');

201. ElemChar char4('d');

202. ElemChar char5('e');

203.

204. gh.add(&int1);

205. gh.add(&int2);

206. gh.add(&char1);

207. gh.add(&char2);

208. gh.add(&int3);

209. gh.add(&int4);

210. gh.add(&int5);

211. gh.add(&char3);

212. gh.add(&char4);

213. gh.add(&char5);

214.

215. gh.print();

216. cout << "int3 (12) in set: " << gh.contains(&int3) << "\n";

217. gh.del(&int3);

218. cout << "int3 (12) in set: " << gh.contains(&int3) << "\n";

219. gh.print();

220. cout << gh.power() << "\n";

221. return 0;

222. }

223.

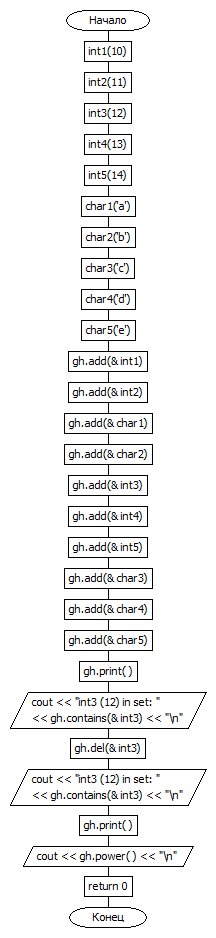


Рисунок : Схема алгоритма тестирующей функции

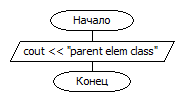


Рисунок : Схема алгоритма elem print()

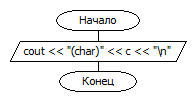


Рисунок : Схема алгоритма elemchar print()

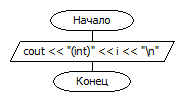


Рисунок : : Схема алгоритма elemint print()

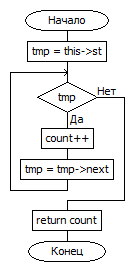


Рисунок : Схема алгоритма setpp power()

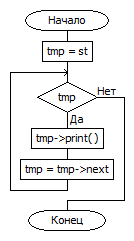


Рисунок : Схема алгоритмa set print()

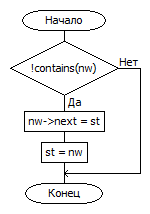


Рисунок : Схема алгоритма set add()

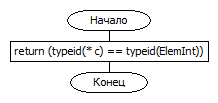


Рисунок : Схема алгоритма fl()

Преобразуем класс множество и его класс-наследник в шаблоны.

1. #include "iostream"

2. #include "string"

3. #include <typeinfo>

4. using std::cout, std::string;

5.

6. class Elem

7. {

8. public:

9. Elem \*next;

10. Elem(Elem \*next = nullptr) : next(next){};

11. virtual void print();

12. };

13.

14. class ElemInt : public Elem

15. {

16. protected:

17. int i;

18.

19. public:

20. ElemInt(int i, Elem \*next = nullptr) : Elem(next), i(i){};

21. void print();

22. int getElem();

23. };

24.

25. class ElemChar : public Elem

26. {

27. protected:

28. char c;

29.

30. public:

31. ElemChar(char c, Elem \*next = nullptr) : Elem(next), c(c){};

32. void print();

33. char getElem();

34. };

35.

36. template <class element>

37. class set

38. {

39. protected:

40. element \*st = nullptr;

41.

42. public:

43. set();

44. ~set();

45. bool contains(element \*cn);

46. void add(element \*nw);

47. void del(element \*dt);

48. void print();

49. };

50.

51. template <class element>

52. class setP : public set<element>

53. {

54. public:

55. int power();

56. };

57.

58. void Elem::print()

59. {

60. cout << "parent elem class";

61. };

62.

63. void ElemInt::print()

64. {

65. cout << "(int)" << i << "\n";

66. };

67.

68. int ElemInt::getElem()

69. {

70. return i;

71. };

72.

73. void ElemChar::print()

74. {

75. cout << "(char)" << c << "\n";

76. };

77.

78. char ElemChar::getElem()

79. {

80. return c;

81. };

82.

83. template <class element>

84. set<element>::set()

85. {

86. cout << "Set init\n";

87. };

88.

89. template <class element>

90. set<element>::~set()

91. {

92. cout << "Set delete\n";

93. };

94.

95. template <class element>

96. bool fl(element c)

97. {

98. // return (static\_cast<void \*>(dynamic\_cast<ElemInt \*>(c)) >

99. // static\_cast<void \*>(dynamic\_cast<ElemChar \*>(c)));

100. return (typeid(\*c) == typeid(ElemInt));

101. }

102.

103. template <class element>

104. bool set<element>::contains(element \*cn)

105. {

106. element \*tmp = st;

107. bool isInt = fl(cn);

108.

109. while (tmp)

110. {

111. bool isTmpInt = fl(tmp);

112. if (isInt == isTmpInt)

113. {

114. if (isInt)

115. {

116. if (dynamic\_cast<ElemInt \*>(cn)->getElem() ==

117. dynamic\_cast<ElemInt \*>(tmp)->getElem())

118. return true;

119. }

120. else

121. {

122. if (dynamic\_cast<ElemChar \*>(cn)->getElem() ==

123. dynamic\_cast<ElemChar \*>(tmp)->getElem())

124. return true;

125. }

126. }

127. tmp = tmp->next;

128. };

129. return false;

130. };

131.

132. template <class element>

133. void set<element>::add(element \*nw)

134. {

135. if (!contains(nw))

136. {

137. nw->next = st;

138. st = nw;

139. }

140. };

141.

142. template <class element>

143. void set<element>::del(element \*dt)

144. {

145. element \*tmp = st;

146. element \*tm = st;

147. bool isInt = fl(dt);

148.

149. while (tmp)

150. {

151. bool isTmpInt = fl(tmp);

152. if (isInt == isTmpInt)

153. {

154. if (isInt)

155. {

156. if (dynamic\_cast<ElemInt \*>(dt)->getElem() ==

157. dynamic\_cast<ElemInt \*>(tmp)->getElem())

158. {

159. tm->next = tmp->next;

160. tmp = tm;

161. return;

162. }

163. }

164. else

165. {

166. if (dynamic\_cast<ElemChar \*>(dt)->getElem() ==

167. dynamic\_cast<ElemChar \*>(tmp)->getElem())

168. {

169. tm->next = tmp->next;

170. tmp = tm;

171. return;

172. }

173. }

174. }

175. tm = tmp;

176. tmp = tmp->next;

177. }

178. };

179.

180. template <class element>

181. void set<element>::print()

182. {

183. element \*tmp = st;

184. while (tmp)

185. {

186. tmp->print();

187. tmp = tmp->next;

188. };

189. };

190.

191. template <class element>

192. int setP<element>::power()

193. {

194. element \*tmp = this->st;

195. int count{};

196. while (tmp)

197. {

198. count++;

199. tmp = tmp->next;

200. };

201. return count;

202. }

203.

204. int main()

205. {

206. setP<Elem> gh;

207.

208. ElemInt int1(10);

209. ElemInt int2(11);

210. ElemInt int3(12);

211. ElemInt int4(13);

212. ElemInt int5(14);

213. ElemChar char1('a');

214. ElemChar char2('b');

215. ElemChar char3('c');

216. ElemChar char4('d');

217. ElemChar char5('e');

218.

219. gh.add(&int1);

220. gh.add(&int2);

221. gh.add(&char1);

222. gh.add(&char2);

223. gh.add(&int3);

224. gh.add(&int4);

225. gh.add(&int5);

226. gh.add(&char3);

227. gh.add(&char4);

228. gh.add(&char5);

229.

230. gh.print();

231. cout << "int3 (12) in set: " << gh.contains(&int3) << "\n";

232. gh.del(&int3);

233. cout << "int3 (12) in set: " << gh.contains(&int3) << "\n";

234. gh.print();

235. cout << gh.power() << "\n";

236. return 0;

237. }

238.

Здесь возникает проблема, что функция, проверяющая класс не может быть нормально реализована для 3 и более классов, т. е. добавить еще один класс-наследник Elem не выйдет без серьезной переработки всей программы.

После этого было придумано не использовать проверку равенства через тип и значения, а проверять лишь совпадение адресов

1. #include "iostream"

2. #include <typeinfo>

3.

4. using std::cout;

5.

6. class Elem

7. {

8. public:

9. Elem \*next;

10. Elem(Elem \*next = nullptr) : next(next){};

11. virtual void print();

12. };

13.

14. class ElemInt : public Elem

15. {

16. protected:

17. int i;

18.

19. public:

20. ElemInt(int i, Elem \*next = nullptr) : Elem(next), i(i){};

21. void print();

22. int getElem();

23. };

24.

25. class ElemChar : public Elem

26. {

27. protected:

28. char c;

29.

30. public:

31. ElemChar(char c, Elem \*next = nullptr) : Elem(next), c(c){};

32. void print();

33. char getElem();

34. };

35.

36. template <class element>

37. class set

38. {

39. protected:

40. element \*st = nullptr;

41.

42. public:

43. set();

44. ~set();

45. bool contains(element \*cn);

46. void add(element \*nw);

47. void del(element \*dt);

48. void print();

49. };

50.

51. template <class element>

52. class setP : public set<element>

53. {

54. public:

55. int power();

56. };

57.

58. void Elem::print()

59. {

60. cout << "parent elem class";

61. };

62.

63. void ElemInt::print()

64. {

65. cout << "(int)" << i << "\n";

66. };

67.

68. int ElemInt::getElem()

69. {

70. return i;

71. };

72.

73. void ElemChar::print()

74. {

75. cout << "(char)" << c << "\n";

76. };

77.

78. char ElemChar::getElem()

79. {

80. return c;

81. };

82.

83. // class ElemFloat : public Elem

84. // {

85. // public:

86. // float f;

87. // ElemFloat(float f, Elem \*next = nullptr) : Elem(next), f(f){};

88. // void print();

89. // float getElem();

90. // };

91.

92. // void ElemFloat::print()

93. // {

94. // cout << "(float)" << f << "\n";

95. // };

96.

97. // float ElemFloat::getElem()

98. // {

99. // return f;

100. // };

101.

102. template <class element>

103. set<element>::set()

104. {

105. cout << "Set init\n";

106. };

107.

108. template <class element>

109. set<element>::~set()

110. {

111. cout << "Set delete\n";

112. };

113.

114. template <class element>

115. bool set<element>::contains(element \*cn)

116. {

117. element \*tmp = st;

118.

119. while (tmp)

120. {

121. if (cn == tmp)

122. return true;

123. tmp = tmp->next;

124. };

125. return false;

126. };

127.

128. template <class element>

129. void set<element>::add(element \*nw)

130. {

131. if (!contains(nw))

132. {

133. nw->next = st;

134. st = nw;

135. }

136. };

137.

138. template <class element>

139. void set<element>::del(element \*dt)

140. {

141. element \*tmp = st;

142. element \*tm = st;

143.

144. while (tmp)

145. {

146. if (dt == tmp)

147. {

148. tm->next = tmp->next;

149. tmp = tm;

150. return;

151. }

152. tm = tmp;

153. tmp = tmp->next;

154. }

155. };

156.

157. template <class element>

158. void set<element>::print()

159. {

160. element \*tmp = st;

161. while (tmp)

162. {

163. tmp->print();

164. tmp = tmp->next;

165. };

166. };

167.

168. template <class element>

169. int setP<element>::power()

170. {

171. element \*tmp = this->st;

172. int count{};

173. while (tmp)

174. {

175. count++;

176. tmp = tmp->next;

177. };

178. return count;

179. }

180.

181. int main()

182. {

183. setP<Elem> gh;

184.

185. ElemInt int1(10);

186. ElemInt int2(11);

187. ElemInt int3(12);

188. ElemInt int4(13);

189. ElemInt int5(14);

190. ElemChar char1('a');

191. ElemChar char2('b');

192. ElemChar char3('c');

193. ElemChar char4('d');

194. ElemChar char5('e');

195. // ElemFloat float1(10.1);

196. // ElemFloat float2(10.2);

197. // ElemFloat float3(10.3);

198. // ElemFloat float4(10.4);

199. // ElemFloat float5(10.5);

200.

201. gh.add(&int1);

202. gh.add(&int2);

203. gh.add(&char1);

204. gh.add(&char2);

205. gh.add(&int3);

206. gh.add(&int4);

207. gh.add(&int5);

208. gh.add(&char3);

209. gh.add(&char4);

210. gh.add(&char5);

211. // gh.add(&float1);

212. // gh.add(&float2);

213. // gh.add(&float3);

214. // gh.add(&float4);

215. // gh.add(&float5);

216.

217. gh.print();

218. cout << "int3 (12) in set: " << gh.contains(&int3) << "\n";

219. gh.del(&int3);

220. cout << "int3 (12) in set: " << gh.contains(&int3) << "\n";

221. // cout << "float4 (10.4) in set: " << gh.contains(&float4) << "\n";

222. // gh.del(&float4);

223. // cout << "float4 (10.4) in set: " << gh.contains(&float4) << "\n";

224. gh.print();

225. cout << gh.power() << "\n";

226. return 0;

227. }

228.

229.

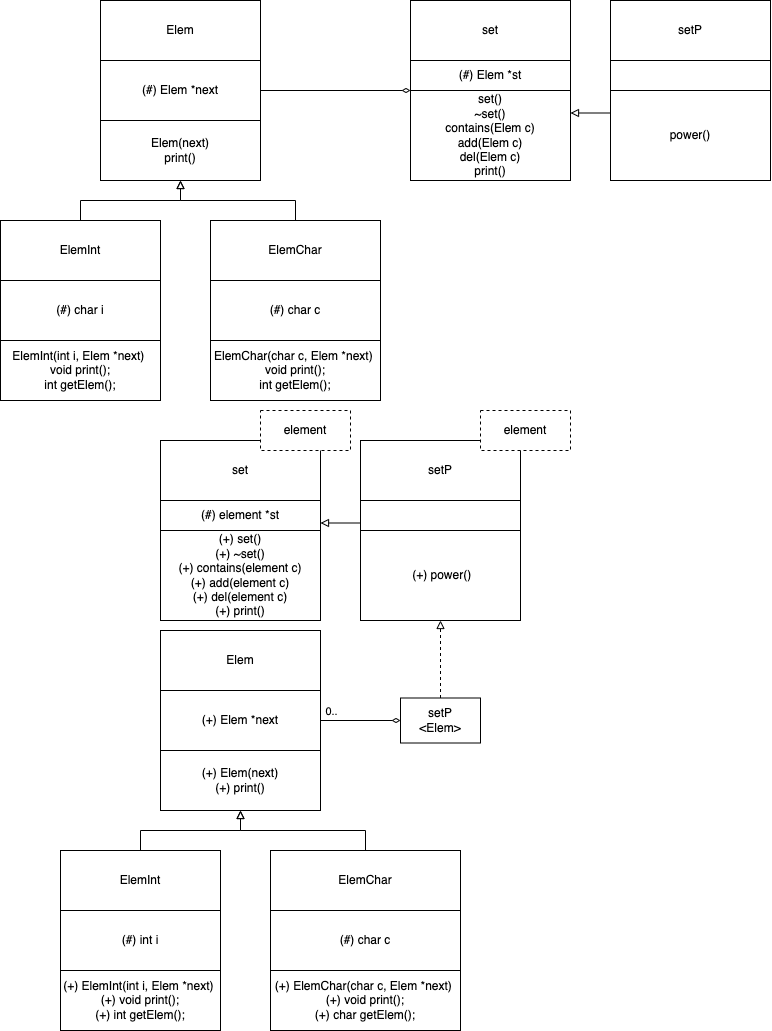


Рисунок : Диаграммы классов для 2 реализаций

Рисунок : Результаты тестирования

*Вывод:* были изучены и применены на практике классы и шаблоны. Были разработаны два множества, одно – как класс, второе – как шаблон. Тестирование полученных структур данных показало корректность работы программ.