

**Actividad:** Actividad de Aprendizaje 11. La Pila y la Cola, implementación dinámica

**Nombre:** Madrigal Escoto Miguel Arturo

**Carrera:** Ingeniería en computación

**Escuela:** Centro Universitario de Ciencias Exactas e Ingenierías

**Materia:** Estructuras de datos

**Profesor:** Gutiérrez Hernández, Alfredo

**Sección:** D12

**Resumen personal del trabajo realizado, y forma en que fue abordado el problema**

En la actividad de aprendizaje de esta semana se vio el manejo adecuado de apuntadores, en la implementación dinámica de la pila y cola.

La implementación de la pila y cola de manera dinámica nos ayuda a sólo reservar el espacio de memoria que se vaya necesitando, e irla liberando adecuadamente. No como la implementación estática, que dejábamos muchos espacios sin usar al trabajar con arreglos, además de que se puede llenar si se alcanza el tamaño del arreglo - 1 en el caso de la cola, o el tamaño del arreglo en el caso de la pila. En la implementación dinámica esto no sucede, solo se podría llegar a llenar si ya no hay memoria disponible para crear nodos.

Ambas implementaciones se basaron en el modelo de la lista, pero adaptando las operaciones de la pila y cola.

La implementación de la pila es simplemente ligada lineal sin encabezado. Muchos métodos quedan igual que en la misma implementación de la lista, como el copia, anula, los constructores y la de vacía. Las operaciones que le dan su personalidad son la de apilar, que consiste en colocar un elemento en el tope de la pila, el desapilar que quita el elemento que esta en el tope de la lista además de retornarlo, y el tope; que esta operación que solamente retorna el dato que está en el tope de la pila.

Cabe recalcar que para todas las operaciones en que se quiere quitar un elemento tanto en pila como en cola se debe de asegurar el nodo, porque si no hacemos esto se pierde la referencia y ya no podremos hacer nada con ese nodo. El asegurarlo quiere decir guardarlo en un apuntador temporal para no perder la referencia, y después de hacer los movimientos correspondientes con los apuntadores, para posteriormente liberar la memoria que ocupa ese nodo.

Para la implementación de la cola hay dos maneras de hacerlo, la cuál es simplemente ligada lineal con encabezado de atributos y doblemente circular con encabezado dummy. El encabezado de atributos se le denomina así porque tenemos dos encabezados, el frente y el final; los cuales son atributos de la cola. El encabezado dummy es porque es un nodo que tiene un anterior y un siguiente, el cual el anterior va a tener la funcionalidad de apuntar al último y el siguiente al primero.

Sin importar cuál sea la implementación que se haga en código, el modelo es el mismo. Una operación es encolar, la cual pone elementos en el final de la cola. El desencolar, que quita elementos por la parte de el frente, además de que lo retorna. Otra operación es el frente, que retorna el elemento que esta en el frente de la cola y la de vacía, que verifica si la dirección siguiente del propio encabezado es el mismo, es decir, no hay elementos. También tiene operaciones para copiar el contenido de una cola a otra, y otra para liberar el espacio de memoria cuando finalice el programa, por medio de la de anula y el destructor.

Esto se adaptará a los requerimientos de la actividad 4, la cual pide transformar de una expresión infija a una posfija.

**Código fuente**

**main.cpp**

1 #include <iostream>

2 #include "notation.h"

3

4 **using namespace std**;

5

6 **int** main()

7 {

8 Notation myNotation;

9 **string** myString, myPosfix;

10

11 **cout** << " - - CONVERTIR DE INFIJA A POSFIJA - - " << **endl** << **endl**;

12 **cout** << "Infija: ";

13 getline(**cin**, myString);

14

15 myNotation.setMyInputString(myString);

16

17 myPosfix = myNotation.toString();

18

19 **cout** << "Posfija: " << myPosfix << **endl**;

20

21 **return** 0;

22 }

23

**stack.h**

1 #ifndef STACK\_H\_INCLUDED

2 #define STACK\_H\_INCLUDED

3

4 **template** <**class** T>

5 **class** Stack{

6 **private**:

7 **class** Node{

8 **private**:

9 T data;

10 Node\* next;

11

12 **public**:

13 Node();

14 Node(**const** T&);

15

16 T getData() **const**;

17 Node\* getNext() **const**;

18

19 **void** setData(**const** T&);

20 **void** setNext(Node\*);

21 };

22

23 Node\* anchor;

24

25 **void** copyAll(**const** Stack&);

26

27 **void** deleteAll();

28

29 **public**:

30 **class** Exception : **public std**::**exception**{

31 **private**:

32 **std**::**string** msg;

33

34 **public**:

35 **explicit** Exception(**const char**\* message) : msg(message){}

36

37 **explicit** Exception(**const std**::**string**& message) : msg(message){}

38

39 **virtual** ~Exception() **throw**(){}

40

41 **virtual const char**\* what() **const throw**() {

42 **return** msg.c\_str();

43 }

44 };

45

46 Stack();

47 Stack(**const** Stack&);

48

49 ~Stack();

50

51 **bool** isEmpty() **const**;

52

53 **void** push(**const** T&);

54

55 T pop();

56

57 T getTop() **const**;

58

59 Stack& **operator** = (**const** Stack&);

60 };

61

62 **///Node**

63 **template** <**class** T>

64 Stack<T>::Node::Node() : next(**nullptr**) {}

65

66 **template** <**class** T>

67 Stack<T>::Node::Node(**const** T& e) : data(e), next(**nullptr**){}

68

69 **template** <**class** T>

70 T Stack<T>::Node::getData() **const** {

71 **return** data;

72 }

73

74 **template** <**class** T>

75 **typename** Stack<T>::Node\* Stack<T>::Node::getNext() **const** {

76 **return** next;

77 }

78

79 **template** <**class** T>

80 **void** Stack<T>::Node::setData(**const** T& e) {

81 data = e;

82 }

83

84 **template** <**class** T>

85 **void** Stack<T>::Node::setNext(Node\* p){

86 next = p;

87 }

88

89 **/// Stack**

90 **template** <**class** T>

91 **void** Stack<T>::copyAll(**const** Stack& s){

92 Node\* aux(s.anchor);

93 Node\* last(**nullptr**);

94 Node\* newNode;

95

96 **while**(aux != **nullptr**) {

97 newNode = **new** Node(aux->getData());

98

99 **if**(newNode == **nullptr**){

100 **throw** Exception("Memoria no disponible, copyAll");

101 }

102

103 **if**(last == **nullptr**){

104 anchor = newNode;

105 }**else**{

106 last->setNext(newNode);

107 }

108

109 last = newNode;

110 aux = aux->getNext();

111 }

112 }

113

114 **template** <**class** T>

115 **void** Stack<T>::deleteAll(){

116 Node\* aux;

117

118 **while**(anchor != **nullptr**) {

119 aux = anchor;

120

121 anchor = anchor->getNext();

122

123 **delete** aux;

124 }

125 }

126

127 **template** <**class** T>

128 Stack<T>::Stack() : anchor(**nullptr**) {}

129

130 **template** <**class** T>

131 Stack<T>::Stack(**const** Stack& s) : anchor(**nullptr**) {

132 copyAll(s);

133 }

134

135 **template** <**class** T>

136 Stack<T>::~Stack(){

137 deleteAll();

138 }

139

140 **template** <**class** T>

141 **bool** Stack<T>::isEmpty() **const**{

142 **return** anchor == **nullptr**;

143 }

144

145 **template** <**class** T>

146 **void** Stack<T>::push(**const** T& e){

147 Node\* aux(**new** Node(e));

148

149 **if**(aux == **nullptr**){

150 **throw** Exception("Memoria no disponible, push");

151 }

152

153 aux->setNext(anchor);

154

155 anchor = aux;

156 }

157

158 **template** <**class** T>

159 T Stack<T>::pop(){

160 **if**(anchor == **nullptr**){

161 **throw** Exception("Insuficiencia de datos, pop");

162 }

163

164 T result(anchor->getData());

165

166 Node\* aux(anchor);

167

168 anchor = anchor->getNext();

169

170 **delete** aux;

171

172 **return** result;

173 }

174

175 **template** <**class** T>

176 T Stack<T>::getTop() **const** {

177 **if**(anchor == **nullptr**){

178 **throw** Exception("Insuficiencia de datos, getTop");

179 }

180

181 **return** anchor->getData();

182 }

183

184 **template** <**class** T>

185 Stack<T>& Stack<T>::**operator** = (**const** Stack& s){

186 deleteAll();

187

188 copyAll(s);

189

190 **return** \***this**;

191 }

192

193 #endif // STACK\_H\_INCLUDED

**queue.h**

1 #ifndef QUEUE\_H\_INCLUDED

2 #define QUEUE\_H\_INCLUDED

3

4 **template** <**class** T>

5 **class** Queue{

6 **private**:

7 **class** Node{

8 **private**:

9 T\* dataPtr;

10 Node\* prev;

11 Node\* next;

12

13 **public**:

14 **class** Exception : **public std**::**exception**{

15 **private**:

16 **std**::**string** msg;

17

18 **public**:

19 **explicit** Exception(**const char**\* message) : msg(message){}

20

21 **explicit** Exception(**const std**::**string**& message) : msg(message){}

22

23 **virtual** ~Exception() **throw**(){}

24

25 **virtual const char**\* what() **const throw**() {

26 **return** msg.c\_str();

27 }

28 };

29

30 Node();

31 Node(**const** T&);

32

33 ~Node();

34

35 T\* getDataPtr() **const**;

36 T getData() **const**;

37 Node\* getPrev() **const**;

38 Node\* getNext() **const**;

39

40 **void** setDataPtr(T\*);

41 **void** setData(**const** T&);

42 **void** setPrev(Node\*);

43 **void** setNext(Node\*);

44 };

45

46 Node\* header;

47

48 **void** copyAll(**const** Queue<T>&);

49

50 **void** deleteAll();

51

52 **public**:

53 **class** Exception : **public std**::**exception**{

54 **private**:

55 **std**::**string** msg;

56

57 **public**:

58 **explicit** Exception(**const char**\* message) : msg(message){}

59

60 **explicit** Exception(**const std**::**string**& message) : msg(message){}

61

62 **virtual** ~Exception() **throw**(){}

63

64 **virtual const char**\* what() **const throw**() {

65 **return** msg.c\_str();

66 }

67 };

68

69 Queue();

70 Queue(**const** Queue&);

71

72 ~Queue();

73

74 **bool** isEmpty() **const**;

75

76 **void** enqueue(**const** T&);

77

78 T dequeue();

79

80 T getFront() **const**;

81

82 Queue& **operator** = (**const** Queue&);

83 };

84

85 **/// Node**

86 **template** <**class** T>

87 Queue<T>::Node::Node() : dataPtr(**nullptr**), prev(**nullptr**), next(**nullptr**){}

88

89 **template** <**class** T>

90 Queue<T>::Node::Node(**const** T& e) : dataPtr(**new** T(e)), prev(**nullptr**), next(**nullptr**){

91 **if**(dataPtr == **nullptr**){

92 **throw** Exception("Memoria insuficiente, creando nodo");

93 }

94 }

95

96 **template** <**class** T>

97 Queue<T>::Node::~Node(){

98 **delete** dataPtr;

99 }

100

101 **template** <**class** T>

102 T\* Queue<T>::Node::getDataPtr() **const**{

103 **return** dataPtr;

104 }

105

106 **template** <**class** T>

107 T Queue<T>::Node::getData() **const**{

108 **if**(dataPtr == **nullptr**){

109 **throw** Exception("Dato inexistente, getData");

110 }

111

112 **return** \*dataPtr;

113 }

114

115 **template** <**class** T>

116 **typename** Queue<T>::Node\* Queue<T>::Node::getPrev() **const**{

117 **return** prev;

118 }

119

120 **template** <**class** T>

121 **typename** Queue<T>::Node\* Queue<T>::Node::getNext() **const**{

122 **return** next;

123 }

124

125 **template** <**class** T>

126 **void** Queue<T>::Node::setDataPtr(T\* p){

127 dataPtr = p;

128 }

129

130 **template** <**class** T>

131 **void** Queue<T>::Node::setData(**const** T& e){

132 **if**(dataPtr == **nullptr**){

133 **if**((dataPtr = **new** T(e)) == **nullptr**){

134 **throw** Exception("Memoria no disponible, setData");

135 }

136 }

137 **else**{

138 \*dataPtr = e;

139 }

140 }

141

142 **template** <**class** T>

143 **void** Queue<T>::Node::setPrev(Node\* p){

144 prev = p;

145 }

146

147 **template** <**class** T>

148 **void** Queue<T>::Node::setNext(Node\* p){

149 next = p;

150 }

151

152 **/// Queue**

153 **template** <**class** T>

154 **void** Queue<T>::copyAll(**const** Queue<T>& l){

155 Node\* aux(l.header->getNext());

156 Node\* newNode;

157

158 **while**(aux != l.header){

159 **try**{

160 **if**((newNode = **new** Node(aux->getData())) == **nullptr**){

161 **throw** Exception("Memoria no disponible, copyAll");

162 }

163 }

164 **catch**(**typename** Node::Exception ex){

165 **throw** Exception(ex.what());

166 }

167

168 newNode->setPrev(header->getPrev());

169 newNode->setNext(header);

170

171 header->getPrev()->setNext(newNode);

172 header->setPrev(newNode);

173

174 aux = aux->getNext();

175 }

176 }

177

178 **template** <**class** T>

179 **void** Queue<T>::deleteAll(){

180 Node\* aux;

181

182 **while**(header->getNext() != header){

183 aux = header->getNext();

184

185 header->setNext(aux->getNext());

186

187 **delete** aux;

188 }

189

190 header->setPrev(header);

191 }

192

193 **template** <**class** T>

194 Queue<T>::Queue() : header(**new** Node){

195 **if**(header == **nullptr**){

196 **throw** Exception("Memoria no disponible, inicializando Queue");

197 }

198

199 header->setPrev(header);

200 header->setNext(header);

201 }

202

203 **template** <**class** T>

204 Queue<T>::Queue(**const** Queue& q) : Queue(){

205 copyAll(q);

206 }

207

208 **template** <**class** T>

209 Queue<T>::~Queue(){

210 deleteAll();

211

212 **delete** header;

213 }

214

215 **template** <**class** T>

216 **bool** Queue<T>::isEmpty() **const**{

217 **return** header->getNext() == header;

218 }

219

220 **template** <**class** T>

221 **void** Queue<T>::enqueue(**const** T& e){

222 Node\* aux;

223

224 **try**{

225 **if**((aux = **new** Node(e)) == **nullptr**){

226 **throw** Exception("Memoria no suficiente, enqueue");

227 }

228 }**catch**(**typename** Node::Exception ex){

229 **throw** Exception(ex.what());

230 }

231

232 aux->setPrev(header->getPrev());

233 aux->setNext(header);

234

235 header->getPrev()->setNext(aux);

236 header->setPrev(aux);

237 }

238

239 **template** <**class** T>

240 T Queue<T>::dequeue(){

241 **if**(isEmpty()){

242 **throw** Exception("Insuficiencia de datos, dequeue");

243 }

244

245 T result(header->getNext()->getData());

246

247 Node\* aux(header->getNext());

248

249 aux->getPrev()->setNext(aux->getNext());

250 aux->getNext()->setPrev(aux->getPrev());

251

252 **delete** aux;

253

254 **return** result;

255 }

256

257 **template** <**class** T>

258 T Queue<T>::getFront() **const**{

259 **if**(isEmpty()){

260 **throw** Exception("Insuficiencia de datos, getFront");

261 }

262

263 **return** header->getNext()->getData();

264 }

265

266 **template** <**class** T>

267 Queue<T>& Queue<T>::**operator** = (**const** Queue& q){

268 deleteAll();

269

270 copyAll(q);

271

272 **return** \***this**;

273 }

274

275 #endif // QUEUE\_H\_INCLUDED

**notation.h**

1 #ifndef NOTATION\_H\_INCLUDED

2 #define NOTATION\_H\_INCLUDED

3

4 #include <iostream>

5 #include <string>

6 #include "stack.h"

7 #include "queue.h"

8

9 **class** Notation{

10 **private**:

11 Queue <**char**> myInfix;

12 Queue <**char**> myPosfix;

13 Stack <**char**> myStack;

14 **std**::**string** myInputString;

15

16 **public**:

17 Notation();

18 Notation(**const** Notation&);

19

20 **void** setMyInputString(**const std**::**string**&);

21

22 **void** enqueueMyInfix();

23 **void** enqueueMyPosfix();

24 **void** pushMyOperators();

25 **void** toPosfix();

26

27 **std**::**string** toString();

28 **int** getPriority(**const char**&);

29

30 Notation& **operator** = (**const** Notation&);

31

32 };

33

34

35 #endif // NOTATION\_H\_INCLUDED

**notation.cpp**

1 #include "notation.h"

2

3 **using namespace std**;

4

5 Notation::Notation(){};

6

7 Notation::Notation(**const** Notation& n): myInfix(n.myInfix), myPosfix(n.myPosfix),

8 myStack(n.myStack), myInputString(n.myInputString){}

9

10 Notation& Notation::**operator** = (**const** Notation& n){

11 myInfix = n.myInfix;

12 myPosfix = n.myPosfix;

13 myStack = n.myStack;

14 myInputString = n.myInputString;

15

16 **return** \***this**;

17 }

18 **void** Notation::setMyInputString(**const string**& myInputString){

19 **this**->myInputString = myInputString;

20 }

21 **void** Notation::enqueueMyInfix(){

22 **for**(**size\_t** i(0); i <= **this**->myInputString.length(); i++)

23 **this**->myInfix.enqueue(myInputString[i]);

24 }

25 **void** Notation::enqueueMyPosfix(){

26 **while**(!myStack.isEmpty())

27 **this**->myPosfix.enqueue(myStack.pop());

28 }

29 **void** Notation::toPosfix(){

30 **while**(!myInfix.isEmpty()){

31 **if**(isalnum(myInfix.getFront()))

32 **this**->myPosfix.enqueue(myInfix.getFront());

33 **else if**(myInfix.getFront() == '(')

34 **this**->myStack.push(myInfix.getFront());

35 **else if**(myInfix.getFront() == ')'){

36 **try**{

37 **while**(myStack.getTop() != '(' || myStack.isEmpty())

38 **this**->myPosfix.enqueue(myStack.pop());

39 **if**(myStack.getTop() == '(')

40 myStack.pop();

41 }**catch**(**typename** Stack<**char**>::Exception ex){

42 ex.what();

43 }

44 }

45 **else if**( myInfix.getFront() == '-' || myInfix.getFront() == '+' || myInfix.getFront() == '\*' || myInfix.getFront() == '/' || myInfix.getFront() == '^' ){

46 **try**{

47 **while**(getPriority(myStack.getTop()) >= getPriority(myInfix.getFront()) && (myStack.getTop()!='(' || myStack.isEmpty()))

48 **this**->myPosfix.enqueue(myStack.pop());

49

50 }**catch**(**typename** Stack<**char**>::Exception ex){

51 ex.what();

52 }

53 **this**->myStack.push(myInfix.getFront());

54 }

55 myInfix.dequeue();

56 }

57 }

58

59 **string** Notation::toString(){

60 **string** myResult;

61

62 enqueueMyInfix();

63 toPosfix();

64 enqueueMyPosfix();

65

66 Queue <**char**> myCopyQueue(myPosfix);

67

68 **while**(!myCopyQueue.isEmpty())

69 myResult += myCopyQueue.dequeue();

70

71 **return** myResult;

72 }

73

74 **int** Notation::getPriority(**const char**& c){

75

76 **switch**(c){

77 **case** '+': **return** 1; **break**;

78 **case** '-': **return** 1; **break**;

79 **case** '\*': **return** 2; **break**;

80 **case** '/': **return** 2; **break**;

81 **case** '^': **return** 3; **break**;

82 }

83

84 **return** 0;

85 }

**Capturas de pantalla**







