

JUTGE PRO2 FIB

L12. Classe List

GitHub: <https://github.com/MUX-enjoyer/PRO2-FIB-2025>

Índex de Fitxers

X25312 Mètode de la classe llista per a moure el segon element al final.cc (pàgina 2)

X41197 Mètode de llistes per a moure l'element apuntat per un iterador al final de la llista.cc (pàgina 11)

X48340 Modificar operadors ++ i -- dels iteradors de la classe List per a que siguin circulars.cc (pàgina 20)

X75139 Mètode de llistes per a moure l'element apuntat per un iterador una posició cap al final.cc (pàgina 28)

X96416 Mètode de llistes per intercanviar (swappejar) el primer i l'últim element.cc (pàgina 37)

Exercici: X25312 Mètode de la classe llista per a moure el segon element al final.cc

```
1 #include <iostream>
2
3 using namespace std;
4
5 template <typename T> class List {
6 private:
7
8 // Items:
9 class Item {
10 public:
11 T value;
12 Item *next;
13 Item *prev;
14 };
15
16 // Data:
17 int _size;
18 Item iteminf, itemsup;
19
20 // Mètodes auxiliars
21 void insertItem(Item *pitemprev, Item *pitem)
22 /* Pre: pitemprev apunta a un element del p.i. pitemprev != pitem,
   pitemprev.next = P, pitem != nullptr, pitem apunta a un element singular */
23 /* Post: inserta al p.i. el node apuntat per pitem entre el node apuntat per
   pitemprev i P, augmenta _size en 1 */
24 {
25 pitem->next = pitemprev->next;
26 pitem->next->prev = pitem;
27 pitem->prev = pitemprev;
28 pitemprev->next = pitem;
29 _size++;
30 }
31
32 void insertItem(Item *pitemprev, const T &value)
33 /* Pre: pitemprev apunta un element del p.i. pitemprev.next = P */
34 /* Post: inserta al p.i. un node amb valor value entre el node apuntat per
   pitemprev i P, augmenta _size en 1 */
35 {
36 Item *pitem = new Item;
37 pitem->value = value;
38 insertItem(pitemprev, pitem);
39 }
40
41 void extractItem(Item *pitem)
42 /* Pre: pitem != & iteminf, pitem != & itemsup pitem apunta a un element del
   p.i. */
43 /* Post: enllaça doblement el node anterior a pitem amb el posterior a pitem,
   disminueix en 1 _size */
44 {
45 pitem->next->prev = pitem->prev;
46 pitem->prev->next = pitem->next;
47 _size--;
48 }
49
50 void removeItem(Item *pitem)
```

```

51 /* Pre: pitem != &iteminf, pitem != &itemsup */
52 /* Post: enllaça doblement el node anterior a pitem amb el posterior a pitem,
s'allibera la memòria del node apuntat per pitem */
53 {
54 extractItem(pitem);
55 delete pitem;
56 }
57
58 void removeItems() {
59 /* Pre: _size = S */
60 /* Post: s'ha alliberat la memòria dels S nodes entre iteminf i itemsup,
iteminf.next = &itemsup, itemsup.prev = &iteminf, _size = 0*/
61 while (_size > 0)
62 removeItem(iteminf.next);
63 }
64
65 void copyItems(List & l) {
66 /* Pre: cert*/
67 /* Post: copia els elements de la llista l al p.i. */
68 for (Item *pitem = l.itemsup.prev; pitem != &l.iteminf; pitem = pitem->prev)
69 insertItem(&iteminf, pitem->value);
70 }
71
72
73 public:
74
75 // Constructors/Destructors:
76 List()
77 /* Pre: cert*/
78 /* Post: el resultat és una llista sense elements */
79 {
80 _size = 0;
81 iteminf.next = &itemsup;
82 itemsup.prev = &iteminf;
83 }
84
85 List(List &l)
86 /* Pre: cert */
87 /* Post: El resultat és una còpia d'l */
88 {
89 _size = 0;
90 iteminf.next = &itemsup;
91 itemsup.prev = &iteminf;
92 copyItems(l);
93 }
94
95 ~List()
96 // Destructora: Esborra automàticament els objectes locals en // sortir d'un
àmbit de visibilitat {
97 removeItems();
98 }
99
100 // Assignment:
101 List &operator=(const List &l)
102 /* Pre: cert */
103 /* Post: El p.i. passa a ser una còpia d'l i qualsevol contingut anterior del
p.i. ha estat esborrat (excepte si el p.i. i l ja eren el mateix objecte) */
104 {
105 if (this != &l) {
106 removeItems();
107 copyItems(l);
108 }
109 return *this;

```

```

110 }
111
112 // Standard operations:
113 int size() const
114 /* Pre: cert */
115 /* Post: el resultat és el nombre de nodes del p.i (els nodes iteminf i
itemsup no es contenen) */
116 {
117 return _size;
118 }
119
120 bool empty() const
121 /* Pre: cert */
122 /* Post: el resultat és si el p.i té nodes o no (els nodes iteminf i itemsup
no contenen) */
123 {
124 return _size == 0;
125 }
126
127
128 void push_back(const T &value)
129 /* Pre: cert */
130 /* Post: s'inserta un node amb valor value al final del p.i. */
131 {
132 insertItem(itemsup.prev, value);
133 }
134
135 void push_front(const T &value)
136 /* Pre: cert */
137 /* Post: s'inserta un node amb valor value al principi del p.i. */
138 {
139 insertItem(&iteminf, value);
140 }
141
142 void pop_back()
143 /* Pre: el p.i. no és buit */
144 /* Post: s'esborra el primer node del p.i. */
145 {
146 if (_size == 0) {
147 cerr << "Error: pop_back on empty list" << endl;
148 exit(1);
149 }
150 removeItem(itemsup.prev);
151 }
152
153 void pop_front()
154 /* Pre: el p.i. no és buit */
155 /* Post: s'esborra el darrer node del p.i. */
156 {
157 if (_size == 0) {
158 cerr << "Error: pop_front on empty list" << endl;
159 exit(1);
160 }
161 removeItem(iteminf.next);
162 }
163
164
165 // Read and write:
166 template <typename U> friend istream &operator>>(istream &is, List<U> &l);
167
168 template <typename U> friend ostream &operator<<(ostream &os, List<U> &l);
169
170

```

```

171 // Iterators mutables
172 class iterator {
173 friend class List;
174 private:
175 List *plist;
176 Item *pitem;
177 public:
178
179
180 // Preincrement iterator operator++()
181 /* Pre: el p.i apunta a un element E de la llista, que no és el end() */
182 /* Post: el p.i apunta a l'element següent a E el resultat és el p.i. */
183 {
184 if (pitem == &(amp;plist->itemsup)) {
185 cerr << "Error: ++iterator at the end of list" << endl;
186 exit(1);
187 }
188 pitem = pitem->next;
189 return *this;
190 }
191
192 // Postincrement iterator operator++(int)
193 /* Pre: p.i. = I, el p.i apunta a un element E de la llista, que no és el
end() */
194 /* Post: el p.i apunta a l'element següent a E, el resultat és I */
195 {
196 if (pitem == &(amp;plist->itemsup)) {
197 cerr << "Error: iterator++ at the end of list" << endl;
198 exit(1);
199 }
200 iterator aux = *this;
201 pitem = pitem->next;
202 return aux;
203 }
204
205 // Predecrement iterator operator--()
206 /* Pre: el p.i apunta a un element E de la llista que no és el begin() */
207 /* Post: el p.i apunta a l'element anterior a E, el resultat és el p.i. */
208 {
209 if (pitem == plist->iteminf.next) {
210 cerr << "Error: --iterator at the beginning of list" << endl;
211 exit(1);
212 }
213 pitem = pitem->prev;
214 return *this;
215 }
216
217 // Postdecrement iterator operator--(int)
218 /* Pre: p.i. = I, el p.i apunta a un element E de la llista que no és el
begin() */
219 /* Post: el p.i apunta a l'element anterior a E, el resultat és I */
220 {
221 if (pitem == plist->iteminf.next) {
222 cerr << "Error: iterator-- at the beginning of list" << endl;
223 exit(1);
224 }
225 iterator aux;
226 aux = *this;
227 pitem = pitem->prev;
228 return aux;
229 }
230
231 T& operator*( )

```

```

232 /* Pre: el p.i apunta a un element E de la llista, que no és el end() */
233 /* Post: el resultat és el valor de l'element E */
234 {
235     if (pitem == &(plist->itemsup)) {
236         cerr << "Error: ++iterator at the end of list" << endl;
237         exit(1);
238     }
239     return pitem->value;
240 }
241
242 bool operator==(const iterator &it) const
243 /* Pre: cert */
244 /* Post: el resultat ens diu si els dos iteradors són iguals */
245 {
246     return plist == it.plist and pitem == it.pitem;
247 }
248
249 bool operator!=(const iterator &it) const
250 /* Pre: cert */
251 /* Post: el resultat ens diu si els dos iteradors són diferents */
252 {
253     return not (plist == it.plist and pitem == it.pitem);
254 }
255
256 };
257
258 // Operations with iterators:
259 iterator begin ()
260 /* Pre: cert */
261 /* Post: el resultat apunta al primer element del p.i o a l'element fictici
si el p.i. no té cap element no fictici */
262 {
263     iterator it;
264     it.plist = this;
265     it.pitem = iteminf.next;
266     return it;
267 }
268
269 iterator end()
270 /* Pre: cert */
271 /* Post: el resultat apunta a l'element fictici */
272 {
273     iterator it;
274     it.plist = this;
275     it.pitem = &itemsup;
276     return it;
277 }
278
279 iterator insert(iterator it, const T & value)
280 /* Pre: it ha d'apuntar a un element del p.i o a l'element fictici end() del
p.i */
281 /* Post: s'inserta un element amb valor value abans de l'element al que
apunta it i el resultat apunta al nou element inserit */
282 {
283     if (it.plist != this) {
284         cout << "Error: insert with an iterator not on this list" << endl;
285         exit(1);
286     }
287     iterator res = it;
288     insertItem(res.pitem->prev, value);
289     res.pitem = res.pitem->prev;
290     return res;
291 }

```

```

292
293 iterator erase(iterator it)
294 /* Pre: it ha d'apuntar a un element del p.i que no sigui l'end() */
295 /* Post: s'elimina l'element apuntat per it, el resultat apunta a l'element
posterior a l'eliminat */
296 {
297     if (it.plist != this) {
298         cout << "Error: erase with an iterator not on this list" << endl;
299         exit(1);
300     }
301     if (it.pitem == &itemsup) {
302         cout << "Error: erase with an iterator pointing to the end of the list" <<
endl;
303         exit(1);
304     }
305     iterator res = it;
306     res.pitem = res.pitem->next;
307     removeItem(res.pitem->prev);
308     return res;
309 }
310
311
312 // Iteradors constants
313 class const_iterator {
314     friend class List;
315 private:
316     List const *plist;
317     Item const *pitem;
318
319 public:
320
321
322 // Preincrement const_iterator operator++()
323 /* Pre: el p.i apunta a un element E de la llista, que no és el end() */
324 /* Post: el p.i apunta a l'element següent a E, el resultat és el p.i. */
325 {
326     if (pitem == &(plist->itemsup)) {
327         cerr << "Error: ++iterator at the end of list" << endl;
328         exit(1);
329     }
330     pitem = pitem->next;
331     return *this;
332 }
333
334 // Postincrement const_iterator operator++(int)
335 /* Pre: p.i. = I, el p.i apunta a un element E de la llista, que no és el
end() */
336 /* Post: el p.i apunta a l'element següent a E, el resultat és I */
337 {
338     if (pitem == &(plist->itemsup)) {
339         cerr << "Error: iterator++ at the end of list" << endl;
340         exit(1);
341     }
342     const_iterator aux = *this;
343     pitem = pitem->next;
344     return aux;
345 }
346
347 // Predecrement const_iterator operator--()
348 /* Pre: el p.i apunta a un element E de la llista que no és el begin() */
349 /* Post: el p.i apunta a l'element anterior a E, el resultat és el p.i. */
350 {
351     if (pitem == plist->iteminf.next) {

```

```

352 cerr << "Error: --iterator at the beginning of list" << endl;
353 exit(1);
354 }
355 pitem = pitem->prev;
356 return *this;
357 }
358
359 // Postdecrement const_iterator operator--(int)
360 /* Pre: p.i. = I, el p.i apunta a un element E de la llista que no és el
begin() */
361 /* Post: el p.i apunta a l'element anterior a E, el resultat és I */
362 {
363 if (pitem == plist->iteminf.next) {
364 cerr << "Error: iterator-- at the beginning of list" << endl;
365 exit(1);
366 }
367 const_iterator aux;
368 aux = *this;
369 pitem = pitem->prev;
370 return aux;
371 }
372
373 const T& operator*()
374 /* Pre: el p.i apunta a un element E de la llista, que no és el end() */
375 /* Post: el resultat és el valor de l'element E */
376 {
377 return pitem->value;
378 }
379
380 bool operator==(const const_iterator &it) const
381 /* Pre: cert */
382 /* Post: el resultat ens diu si els dos iterators són iguals */
383 {
384 return plist == it.plist and pitem == it.pitem;
385 }
386
387 bool operator!=(const const_iterator &it) const
388 /* Pre: cert */
389 /* Post: el resultat ens diu si els dos iterators són diferents */
390 {
391 return not (plist == it.plist and pitem == it.pitem);
392 }
393 };
394
395
396 const_iterator cbegin() const
397 /* Pre: cert */
398 /* Post: el resultat apunta al primer element del p.i o a l'element fictici
si el p.i. no té cap element no fictici */
399 {
400 const_iterator it;
401 it.plist = this;
402 it.pitem = iteminf.next;
403 return it;
404 }
405
406 const_iterator cend() const
407 /* Pre: cert */
408 /* Post: el resultat apunta a l'element fictici */
409 {
410 const_iterator it;
411 it.plist = this;
412 it.pitem = &itemsup;

```



```

413 return it;
414 }
415
416 const_iterator insert(const_iterator it, const T & value)
417 /* Pre: it ha d'apuntar a un element del p.i o a l'element fictici end() del
p.i */
418 /* Post: s'inserta un element amb valor value abans de l'element al que
apunta it i el resultat apunta al nou element inserit */
419 {
420 if (it.plist != this) {
421 cout << "Error: insert with an iterator not on this list" << endl;
422 exit(1);
423 }
424 const_iterator res = it;
425 insertItem(res.pitem->prev, value);
426 res.pitem = res.pitem->prev;
427 return res;
428 }
429
430 const_iterator erase(const_iterator it)
431 /* Pre: it ha d'apuntar a un element del p.i que no sigui l'end() */
432 /* Post: s'elimina l'element apuntat per it, el resultat apunta a l'element
posterior a l'eliminat */
433 {
434 if (it.plist != this) {
435 cout << "Error: erase with an iterator not on this list" << endl;
436 exit(1);
437 }
438 if (it.pitem == &itemsup) {
439 cout << "Error: erase with an iterator pointing to the end of the list" <<
endl;
440 exit(1);
441 }
442 const_iterator res = it;
443 res.pitem = res.pitem->next;
444 removeItem(res.pitem->prev);
445 return res;
446 }
447
448 // Pre: // Post: El segon element de la llista s'ha mogut a l'última posició.
// Si hi havia menys de tres elements a la llista, llavors res ha canviat. //
Descomenteu les següents dues línies i implementeu la funció: void
moveSecondToLast() {
449 if (_size < 3) return;
450 Item *second = iteminf.next->next;
451 second->prev->next = second->next;
452 second->next->prev = second->prev;
453
454 second->prev = itemsup.prev;
455 second->next = &itemsup;
456
457 itemsup.prev->next = second;
458 itemsup.prev = second;
459 }
460
461 };
462 };
463
464 // Implementation of read and write lists.
465 template <typename T> istream &operator>>(istream &is, List<T> &l)
466 {
467 l.removeItem();
468 int size;

```

```
469 cin >> size;
470 for (int i = 0; i < size; ++i) {
471     T value;
472     cin >> value;
473     l.insertItem(l.itemsup.prev, value);
474 }
475 return is;
476 }
477
478 template<typename T> ostream &operator<<(ostream &os, List<T> &l)
479 {
480     os << l._size;
481     for (typename List<T>::Item* pitem = l.iteminf.next; pitem != &l.itemsup;
482          pitem = pitem->next)
483         cout << " " << pitem->value;
484     return os;
485 }
```

Exercici: X41197 Mètode de llistes per a moure l'element apuntat per un iterador al final de la llista.cc

```
1 #include <iostream>
2
3 using namespace std;
4
5 template <typename T> class List {
6 private:
7
8 // Items:
9 class Item {
10 public:
11 T value;
12 Item *next;
13 Item *prev;
14 };
15
16 // Data:
17 int _size;
18 Item iteminf, itemsup;
19
20 // Mètodes auxiliars
21 void insertItem(Item *pitemprev, Item *pitem)
22 /* Pre: pitemprev apunta a un element del p.i. pitemprev != pitem,
   pitemprev.next = P, pitem != nullptr, pitem apunta a un element singular */
23 /* Post: inserta al p.i. el node apuntat per pitem entre el node apuntat per
   pitemprev i P, augmenta _size en 1 */
24 {
25 pitem->next = pitemprev->next;
26 pitem->next->prev = pitem;
27 pitem->prev = pitemprev;
28 pitemprev->next = pitem;
29 _size++;
30 }
31
32 void insertItem(Item *pitemprev, const T &value)
33 /* Pre: pitemprev apunta un element del p.i. pitemprev.next = P */
34 /* Post: inserta al p.i. un node amb valor value entre el node apuntat per
   pitemprev i P, augmenta _size en 1 */
35 {
36 Item *pitem = new Item;
37 pitem->value = value;
38 insertItem(pitemprev, pitem);
39 }
40
41 void extractItem(Item *pitem)
42 /* Pre: pitem != & iteminf, pitem != & itemsup pitem apunta a un element del
   p.i. */
43 /* Post: enllaça doblement el node anterior a pitem amb el posterior a pitem,
   disminueix en 1 _size */
44 {
45 pitem->next->prev = pitem->prev;
46 pitem->prev->next = pitem->next;
47 _size--;
48 }
49
50 void removeItem(Item *pitem)
```

```

51 /* Pre: pitem != &iteminf, pitem != &itemsup */
52 /* Post: enllaça doblement el node anterior a pitem amb el posterior a pitem,
s'allibera la memòria del node apuntat per pitem */
53 {
54 extractItem(pitem);
55 delete pitem;
56 }
57
58 void removeItems() {
59 /* Pre: _size = S */
60 /* Post: s'ha alliberat la memòria dels S nodes entre iteminf i itemsup,
iteminf.next = &itemsup, itemsup.prev = &iteminf, _size = 0 */
61 while (_size > 0)
62 removeItem(iteminf.next);
63 }
64
65 void copyItems(List & l) {
66 /* Pre: cert */
67 /* Post: copia els elements de la llista l al p.i. */
68 for (Item *pitem = l.itemsup.prev; pitem != &l.iteminf; pitem = pitem->prev)
69 insertItem(&iteminf, pitem->value);
70 }
71
72
73 public:
74
75 // Constructors/Destructors:
76 List()
77 /* Pre: cert */
78 /* Post: el resultat és una llista sense elements */
79 {
80 _size = 0;
81 iteminf.next = &itemsup;
82 itemsup.prev = &iteminf;
83 }
84
85 List(List &l)
86 /* Pre: cert */
87 /* Post: El resultat és una còpia d'l */
88 {
89 _size = 0;
90 iteminf.next = &itemsup;
91 itemsup.prev = &iteminf;
92 copyItems(l);
93 }
94
95 ~List()
96 // Destructora: Esborra automàticament els objectes locals en // sortir d'un
àmbit de visibilitat {
97 removeItems();
98 }
99
100 // Assignment:
101 List &operator=(const List &l)
102 /* Pre: cert */
103 /* Post: El p.i. passa a ser una còpia d'l i qualsevol contingut anterior del
p.i. ha estat esborrat (excepte si el p.i. i l ja eren el mateix objecte) */
104 {
105 if (this != &l) {
106 removeItems();
107 copyItems(l);
108 }
109 return *this;

```

```

110 }
111
112 // Standard operations:
113 int size() const
114 /* Pre: cert */
115 /* Post: el resultat és el nombre de nodes del p.i (els nodes iteminf i
itemsup no es contenen) */
116 {
117 return _size;
118 }
119
120 bool empty() const
121 /* Pre: cert */
122 /* Post: el resultat és si el p.i té nodes o no (els nodes iteminf i itemsup
no contenen) */
123 {
124 return _size == 0;
125 }
126
127
128 void push_back(const T &value)
129 /* Pre: cert */
130 /* Post: s'inserta un node amb valor value al final del p.i. */
131 {
132 insertItem(itemsup.prev, value);
133 }
134
135 void push_front(const T &value)
136 /* Pre: cert */
137 /* Post: s'inserta un node amb valor value al principi del p.i. */
138 {
139 insertItem(&iteminf, value);
140 }
141
142 void pop_back()
143 /* Pre: el p.i. no és buit */
144 /* Post: s'esborra el primer node del p.i. */
145 {
146 if (_size == 0) {
147 cerr << "Error: pop_back on empty list" << endl;
148 exit(1);
149 }
150 removeItem(itemsup.prev);
151 }
152
153 void pop_front()
154 /* Pre: el p.i. no és buit */
155 /* Post: s'esborra el darrer node del p.i. */
156 {
157 if (_size == 0) {
158 cerr << "Error: pop_front on empty list" << endl;
159 exit(1);
160 }
161 removeItem(iteminf.next);
162 }
163
164
165 // Read and write:
166 template <typename U> friend istream &operator>>(istream &is, List<U> &l);
167
168 template <typename U> friend ostream &operator<<(ostream &os, List<U> &l);
169
170

```

```

171 // Iterators mutables
172 class iterator {
173 friend class List;
174 private:
175 List *plist;
176 Item *pitem;
177 public:
178
179
180 // Preincrement iterator operator++()
181 /* Pre: el p.i apunta a un element E de la llista, que no és el end() */
182 /* Post: el p.i apunta a l'element següent a E el resultat és el p.i. */
183 {
184 if (pitem == &(amp;plist->itemsup)) {
185 cerr << "Error: ++iterator at the end of list" << endl;
186 exit(1);
187 }
188 pitem = pitem->next;
189 return *this;
190 }
191
192 // Postincrement iterator operator++(int)
193 /* Pre: p.i. = I, el p.i apunta a un element E de la llista, que no és el
end() */
194 /* Post: el p.i apunta a l'element següent a E, el resultat és I */
195 {
196 if (pitem == &(amp;plist->itemsup)) {
197 cerr << "Error: iterator++ at the end of list" << endl;
198 exit(1);
199 }
200 iterator aux = *this;
201 pitem = pitem->next;
202 return aux;
203 }
204
205 // Predecrement iterator operator--()
206 /* Pre: el p.i apunta a un element E de la llista que no és el begin() */
207 /* Post: el p.i apunta a l'element anterior a E, el resultat és el p.i. */
208 {
209 if (pitem == plist->iteminf.next) {
210 cerr << "Error: --iterator at the beginning of list" << endl;
211 exit(1);
212 }
213 pitem = pitem->prev;
214 return *this;
215 }
216
217 // Postdecrement iterator operator--(int)
218 /* Pre: p.i. = I, el p.i apunta a un element E de la llista que no és el
begin() */
219 /* Post: el p.i apunta a l'element anterior a E, el resultat és I */
220 {
221 if (pitem == plist->iteminf.next) {
222 cerr << "Error: iterator-- at the beginning of list" << endl;
223 exit(1);
224 }
225 iterator aux;
226 aux = *this;
227 pitem = pitem->prev;
228 return aux;
229 }
230
231 T& operator*( )

```

```

232 /* Pre: el p.i apunta a un element E de la llista, que no és el end() */
233 /* Post: el resultat és el valor de l'element E */
234 {
235     if (pitem == &(plist->itemsup)) {
236         cerr << "Error: ++iterator at the end of list" << endl;
237         exit(1);
238     }
239     return pitem->value;
240 }
241
242 bool operator==(const iterator &it) const
243 /* Pre: cert */
244 /* Post: el resultat ens diu si els dos iteradors són iguals */
245 {
246     return plist == it.plist and pitem == it.pitem;
247 }
248
249 bool operator!=(const iterator &it) const
250 /* Pre: cert */
251 /* Post: el resultat ens diu si els dos iteradors són diferents */
252 {
253     return not (plist == it.plist and pitem == it.pitem);
254 }
255
256 };
257
258 // Operations with iterators:
259 iterator begin ()
260 /* Pre: cert */
261 /* Post: el resultat apunta al primer element del p.i o a l'element fictici
si el p.i. no té cap element no fictici */
262 {
263     iterator it;
264     it.plist = this;
265     it.pitem = iteminf.next;
266     return it;
267 }
268
269 iterator end()
270 /* Pre: cert */
271 /* Post: el resultat apunta a l'element fictici */
272 {
273     iterator it;
274     it.plist = this;
275     it.pitem = &itemsup;
276     return it;
277 }
278
279 iterator insert(iterator it, const T & value)
280 /* Pre: it ha d'apuntar a un element del p.i o a l'element fictici end() del
p.i */
281 /* Post: s'inserta un element amb valor value abans de l'element al que
apunta it i el resultat apunta al nou element inserit */
282 {
283     if (it.plist != this) {
284         cout << "Error: insert with an iterator not on this list" << endl;
285         exit(1);
286     }
287     iterator res = it;
288     insertItem(res.pitem->prev, value);
289     res.pitem = res.pitem->prev;
290     return res;
291 }

```

```

292
293 iterator erase(iterator it)
294 /* Pre: it ha d'apuntar a un element del p.i que no sigui l'end() */
295 /* Post: s'elimina l'element apuntat per it, el resultat apunta a l'element
posterior a l'eliminat */
296 {
297     if (it.plist != this) {
298         cout << "Error: erase with an iterator not on this list" << endl;
299         exit(1);
300     }
301     if (it.pitem == &itemsup) {
302         cout << "Error: erase with an iterator pointing to the end of the list" <<
endl;
303         exit(1);
304     }
305     iterator res = it;
306     res.pitem = res.pitem->next;
307     removeItem(res.pitem->prev);
308     return res;
309 }
310
311
312 // Iteradors constants
313 class const_iterator {
314     friend class List;
315 private:
316     List const *plist;
317     Item const *pitem;
318
319 public:
320
321
322 // Preincrement const_iterator operator++()
323 /* Pre: el p.i apunta a un element E de la llista, que no és el end() */
324 /* Post: el p.i apunta a l'element següent a E, el resultat és el p.i. */
325 {
326     if (pitem == &(plist->itemsup)) {
327         cerr << "Error: ++iterator at the end of list" << endl;
328         exit(1);
329     }
330     pitem = pitem->next;
331     return *this;
332 }
333
334 // Postincrement const_iterator operator++(int)
335 /* Pre: p.i. = I, el p.i apunta a un element E de la llista, que no és el
end() */
336 /* Post: el p.i apunta a l'element següent a E, el resultat és I */
337 {
338     if (pitem == &(plist->itemsup)) {
339         cerr << "Error: iterator++ at the end of list" << endl;
340         exit(1);
341     }
342     const_iterator aux = *this;
343     pitem = pitem->next;
344     return aux;
345 }
346
347 // Predecrement const_iterator operator--()
348 /* Pre: el p.i apunta a un element E de la llista que no és el begin() */
349 /* Post: el p.i apunta a l'element anterior a E, el resultat és el p.i. */
350 {
351     if (pitem == plist->iteminf.next) {

```



```

352 cerr << "Error: --iterator at the beginning of list" << endl;
353 exit(1);
354 }
355 pitem = pitem->prev;
356 return *this;
357 }
358
359 // Postdecrement const_iterator operator--(int)
360 /* Pre: p.i. = I, el p.i apunta a un element E de la llista que no és el
begin() */
361 /* Post: el p.i apunta a l'element anterior a E, el resultat és I */
362 {
363 if (pitem == plist->iteminf.next) {
364 cerr << "Error: iterator-- at the beginning of list" << endl;
365 exit(1);
366 }
367 const_iterator aux;
368 aux = *this;
369 pitem = pitem->prev;
370 return aux;
371 }
372
373 const T& operator*()
374 /* Pre: el p.i apunta a un element E de la llista, que no és el end() */
375 /* Post: el resultat és el valor de l'element E */
376 {
377 return pitem->value;
378 }
379
380 bool operator==(const const_iterator &it) const
381 /* Pre: cert */
382 /* Post: el resultat ens diu si els dos iteradors són iguals */
383 {
384 return plist == it.plist and pitem == it.pitem;
385 }
386
387 bool operator!=(const const_iterator &it) const
388 /* Pre: cert */
389 /* Post: el resultat ens diu si els dos iteradors són diferents */
390 {
391 return not (plist == it.plist and pitem == it.pitem);
392 }
393
394 };
395
396 const_iterator cbegin() const
397 /* Pre: cert */
398 /* Post: el resultat apunta al primer element del p.i o a l'element fictici
si el p.i. no té cap element no fictici */
399 {
400 const_iterator it;
401 it.plist = this;
402 it.pitem = iteminf.next;
403 return it;
404 }
405
406 const_iterator cend() const
407 /* Pre: cert */
408 /* Post: el resultat apunta a l'element fictici */
409 {
410 const_iterator it;
411 it.plist = this;
412 it.pitem = &itemsup;

```

```

413 return it;
414 }
415
416 const_iterator insert(const_iterator it, const T & value)
417 /* Pre: it ha d'apuntar a un element del p.i o a l'element fictici end() del
p.i */
418 /* Post: s'inserta un element amb valor value abans de l'element al que
apunta it i el resultat apunta al nou element inserit */
419 {
420 if (it.plist != this) {
421 cout << "Error: insert with an iterator not on this list" << endl;
422 exit(1);
423 }
424 const_iterator res = it;
425 insertItem(res.pitem->prev, value);
426 res.pitem = res.pitem->prev;
427 return res;
428 }
429
430 const_iterator erase(const_iterator it)
431 /* Pre: it ha d'apuntar a un element del p.i que no sigui l'end() */
432 /* Post: s'elimina l'element apuntat per it, el resultat apunta a l'element
posterior a l'eliminat */
433 {
434 if (it.plist != this) {
435 cout << "Error: erase with an iterator not on this list" << endl;
436 exit(1);
437 }
438 if (it.pitem == &itemsup) {
439 cout << "Error: erase with an iterator pointing to the end of the list" <<
endl;
440 exit(1);
441 }
442 const_iterator res = it;
443 res.pitem = res.pitem->next;
444 removeItem(res.pitem->prev);
445 return res;
446 }
447
448 // Pre: it apunta a algun element de la llista implícita. // Post: it
continua apuntant al mateix element, el qual ha estat mogut al final // de la
llista. En el cas en que l'element apuntat per it ja era l'últim, // res ha
canviat. // Descomenteu les següents dues línies i implementeu el mètode: void
moveToEnd(iterator &it) {
449 if (it.pitem == itemsup.prev) return;
450 Item *last = it.pitem;
451 last->prev->next = last->next;
452 last->next->prev = last->prev;
453
454 itemsup.prev->next = last;
455 last->prev = itemsup.prev;
456 itemsup.prev = last;
457 last->next = &itemsup;
458 }
459
460 };
461
462 // Implementation of read and write lists.
463 template <typename T> istream &operator>>(istream &is, List<T> &l)
464 {
465 l.removeItem();
466 int size;
467 cin >> size;

```

```
468 for (int i = 0; i < size; ++i) {
469     T value;
470     cin >> value;
471     l.insertItem(l.itemsup.prev, value);
472 }
473 return is;
474 }
475
476 template<typename T> ostream &operator<<(ostream &os, List<T> &l)
477 {
478     os << l._size;
479     for (typename List<T>::Item* pitem = l.iteminf.next; pitem != &l.itemsup;
480          pitem = pitem->next)
481         cout << " " << pitem->value;
482     return os;
483 }
```

Exercici: X48340 Modificar operadors ++ i -- dels iteradors de la classe List per a que siguin circulars.cc

```
1 #include <iostream>
2
3 using namespace std;
4
5 template <typename T> class List {
6 private:
7
8 // Items:
9 class Item {
10 public:
11 T value;
12 Item *next;
13 Item *prev;
14 };
15
16 // Data:
17 int _size;
18 Item iteminf, itemsup;
19
20 // Mètodes auxiliars
21 void insertItem(Item *pitemprev, Item *pitem)
22 /* Pre: pitemprev apunta a un element del p.i. pitemprev != pitem,
   pitemprev.next = P, pitem != nullptr, pitem apunta a un element singular */
23 /* Post: inserta al p.i. el node apuntat per pitem entre el node apuntat per
   pitemprev i P, augmenta _size en 1 */
24 {
25 pitem->next = pitemprev->next;
26 pitem->next->prev = pitem;
27 pitem->prev = pitemprev;
28 pitemprev->next = pitem;
29 _size++;
30 }
31
32 void insertItem(Item *pitemprev, const T &value)
33 /* Pre: pitemprev apunta un element del p.i. pitemprev.next = P */
34 /* Post: inserta al p.i. un node amb valor value entre el node apuntat per
   pitemprev i P, augmenta _size en 1 */
35 {
36 Item *pitem = new Item;
37 pitem->value = value;
38 insertItem(pitemprev, pitem);
39 }
40
41 void extractItem(Item *pitem)
42 /* Pre: pitem != & iteminf, pitem != & itemsup pitem apunta a un element del
   p.i. */
43 /* Post: enllaça doblement el node anterior a pitem amb el posterior a pitem,
   disminueix en 1 _size */
44 {
45 pitem->next->prev = pitem->prev;
46 pitem->prev->next = pitem->next;
47 _size--;
48 }
49
50 void removeItem(Item *pitem)
```

```

51 /* Pre: pitem != &iteminf, pitem != &itemsup */
52 /* Post: enllaça doblement el node anterior a pitem amb el posterior a pitem,
s'allibera la memòria del node apuntat per pitem */
53 {
54 extractItem(pitem);
55 delete pitem;
56 }
57
58 void removeItems() {
59 /* Pre: _size = S */
60 /* Post: s'ha alliberat la memòria dels S nodes entre iteminf i itemsup,
iteminf.next = &itemsup, itemsup.prev = &iteminf, _size = 0 */
61 while (_size > 0)
62 removeItem(iteminf.next);
63 }
64
65 void copyItems(List & l) {
66 /* Pre: cert */
67 /* Post: copia els elements de la llista l al p.i. */
68 for (Item *pitem = l.itemsup.prev; pitem != &l.iteminf; pitem = pitem->prev)
69 insertItem(&iteminf, pitem->value);
70 }
71
72
73 public:
74
75 // Constructors/Destructors:
76 List()
77 /* Pre: cert */
78 /* Post: el resultat és una llista sense elements */
79 {
80 _size = 0;
81 iteminf.next = &itemsup;
82 itemsup.prev = &iteminf;
83 }
84
85 List(List &l)
86 /* Pre: cert */
87 /* Post: El resultat és una còpia d'l */
88 {
89 _size = 0;
90 iteminf.next = &itemsup;
91 itemsup.prev = &iteminf;
92 copyItems(l);
93 }
94
95 ~List()
96 // Destructora: Esborra automàticament els objectes locals en // sortir d'un
àmbit de visibilitat {
97 removeItems();
98 }
99
100 // Assignment:
101 List &operator=(const List &l)
102 /* Pre: cert */
103 /* Post: El p.i. passa a ser una còpia d'l i qualsevol contingut anterior del
p.i. ha estat esborrat (excepte si el p.i. i l ja eren el mateix objecte) */
104 {
105 if (this != &l) {
106 removeItems();
107 copyItems(l);
108 }
109 return *this;

```

```

110 }
111
112 // Standard operations:
113 int size() const
114 /* Pre: cert */
115 /* Post: el resultat és el nombre de nodes del p.i (els nodes iteminf i
itemsup no es contenen) */
116 {
117 return _size;
118 }
119
120 bool empty() const
121 /* Pre: cert */
122 /* Post: el resultat és si el p.i té nodes o no (els nodes iteminf i itemsup
no contenen) */
123 {
124 return _size == 0;
125 }
126
127
128 void push_back(const T &value)
129 /* Pre: cert */
130 /* Post: s'inserta un node amb valor value al final del p.i. */
131 {
132 insertItem(itemsup.prev, value);
133 }
134
135 void push_front(const T &value)
136 /* Pre: cert */
137 /* Post: s'inserta un node amb valor value al principi del p.i. */
138 {
139 insertItem(&iteminf, value);
140 }
141
142 void pop_back()
143 /* Pre: el p.i. no és buit */
144 /* Post: s'esborra el primer node del p.i. */
145 {
146 if (_size == 0) {
147 cerr << "Error: pop_back on empty list" << endl;
148 exit(1);
149 }
150 removeItem(itemsup.prev);
151 }
152
153 void pop_front()
154 /* Pre: el p.i. no és buit */
155 /* Post: s'esborra el darrer node del p.i. */
156 {
157 if (_size == 0) {
158 cerr << "Error: pop_front on empty list" << endl;
159 exit(1);
160 }
161 removeItem(iteminf.next);
162 }
163
164
165 // Read and write:
166 template <typename U> friend istream &operator>>(istream &is, List<U> &l);
167
168 template <typename U> friend ostream &operator<<(ostream &os, List<U> &l);
169
170

```

```

171 // Iterators mutables
172 class iterator {
173 friend class List;
174 private:
175 List *plist;
176 Item *pitem;
177 public:
178
179
180 // Preincrement iterator operator++()
181 /* Pre: el p.i apunta a un element E de la llista, que no és el end() */
182 /* Post: el p.i apunta a l'element següent a E el resultat és el p.i. */
183 {
184 if (pitem == &(plist->itemsup)) pitem = plist->iteminf.next;
185 else pitem = pitem->next;
186 return *this;
187 }
188
189 // Postincrement iterator operator++(int)
190 /* Pre: p.i. = I, el p.i apunta a un element E de la llista, que no és el
end() */
191 /* Post: el p.i apunta a l'element següent a E, el resultat és I */
192 {
193 iterator aux = *this;
194 operator++();
195 return aux;
196 }
197
198 // Predecrement iterator operator--()
199 /* Pre: el p.i apunta a un element E de la llista que no és el begin() */
200 /* Post: el p.i apunta a l'element anterior a E, el resultat és el p.i. */
201 {
202 if (pitem == plist->iteminf.next) pitem = &(plist->itemsup);
203 else pitem = pitem->prev;
204 return *this;
205 }
206
207 // Postdecrement iterator operator--(int)
208 /* Pre: p.i. = I, el p.i apunta a un element E de la llista que no és el
begin() */
209 /* Post: el p.i apunta a l'element anterior a E, el resultat és I */
210 {
211 iterator aux = *this;
212 operator--();
213 return aux;
214 }
215
216 T& operator*()
217 /* Pre: el p.i apunta a un element E de la llista, que no és el end() */
218 /* Post: el resultat és el valor de l'element E */
219 {
220 if (pitem == &(plist->itemsup)) {
221 cerr << "Error: ++iterator at the end of list" << endl;
222 exit(1);
223 }
224 return pitem->value;
225 }
226
227 bool operator==(const iterator &it) const
228 /* Pre: cert */
229 /* Post: el resultat ens diu si els dos iterators són iguals */
230 {
231 return plist == it.plist and pitem == it.pitem;

```

```

232 }
233
234 bool operator!=(const iterator &it) const
235 /* Pre: cert */
236 /* Post: el resultat ens diu si els dos iteradors són diferents */
237 {
238     return not (plist == it.plist and pitem == it.pitem);
239 }
240
241 };
242
243 // Operations with iterators:
244 iterator begin ()
245 /* Pre: cert */
246 /* Post: el resultat apunta al primer element del p.i o a l'element fictici
si el p.i. no té cap element no fictici */
247 {
248     iterator it;
249     it.plist = this;
250     it.pitem = iteminf.next;
251     return it;
252 }
253
254 iterator end()
255 /* Pre: cert */
256 /* Post: el resultat apunta a l'element fictici */
257 {
258     iterator it;
259     it.plist = this;
260     it.pitem = &itemsup;
261     return it;
262 }
263
264 iterator insert(iterator it, const T & value)
265 /* Pre: it ha d'apuntar a un element del p.i o a l'element fictici end() del
p.i */
266 /* Post: s'inserta un element amb valor value abans de l'element al que
apunta it i el resultat apunta al nou element inserit */
267 {
268     if (it.plist != this) {
269         cout << "Error: insert with an iterator not on this list" << endl;
270         exit(1);
271     }
272     iterator res = it;
273     insertItem(res.pitem->prev, value);
274     res.pitem = res.pitem->prev;
275     return res;
276 }
277
278 iterator erase(iterator it)
279 /* Pre: it ha d'apuntar a un element del p.i que no sigui l'end() */
280 /* Post: s'elimina l'element apuntat per it, el resultat apunta a l'element
posterior a l'eliminat */
281 {
282     if (it.plist != this) {
283         cout << "Error: erase with an iterator not on this list" << endl;
284         exit(1);
285     }
286     if (it.pitem == &itemsup) {
287         cout << "Error: erase with an iterator pointing to the end of the list" <<
endl;
288         exit(1);
289     }

```



```

290 iterator res = it;
291 res.pitem = res.pitem->next;
292 removeItem(res.pitem->prev);
293 return res;
294 }
295
296
297 // Iteradors constants
298 class const_iterator {
299 friend class List;
300 private:
301 List const *plist;
302 Item const *pitem;
303
304
305 public:
306
307 // Preincrement const_iterator operator++()
308 /* Pre: el p.i apunta a un element E de la llista, que no és el end() */
309 /* Post: el p.i apunta a l'element següent a E, el resultat és el p.i. */
310 {
311 if (pitem == &(plist->itemsup)) {
312 cerr << "Error: ++iterator at the end of list" << endl;
313 exit(1);
314 }
315 pitem = pitem->next;
316 return *this;
317 }
318
319 // Postincrement const_iterator operator++(int)
320 /* Pre: p.i. = I, el p.i apunta a un element E de la llista, que no és el
end() */
321 /* Post: el p.i apunta a l'element següent a E, el resultat és I */
322 {
323 if (pitem == &(plist->itemsup)) {
324 cerr << "Error: iterator++ at the end of list" << endl;
325 exit(1);
326 }
327 const_iterator aux = *this;
328 pitem = pitem->next;
329 return aux;
330 }
331
332 // Predecrement const_iterator operator--()
333 /* Pre: el p.i apunta a un element E de la llista que no és el begin() */
334 /* Post: el p.i apunta a l'element anterior a E, el resultat és el p.i. */
335 {
336 if (pitem == plist->iteminf.next) {
337 cerr << "Error: --iterator at the beginning of list" << endl;
338 exit(1);
339 }
340 pitem = pitem->prev;
341 return *this;
342 }
343
344 // Postdecrement const_iterator operator--(int)
345 /* Pre: p.i. = I, el p.i apunta a un element E de la llista que no és el
begin() */
346 /* Post: el p.i apunta a l'element anterior a E, el resultat és I */
347 {
348 if (pitem == plist->iteminf.next) {
349 cerr << "Error: iterator-- at the beginning of list" << endl;
350 exit(1);

```

```

351 }
352 const_iterator aux;
353 aux = *this;
354 pitem = pitem->prev;
355 return aux;
356 }
357
358 const T& operator*()
359 /* Pre: el p.i apunta a un element E de la llista, que no és el end() */
360 /* Post: el resultat és el valor de l'element E */
361 {
362 return pitem->value;
363 }
364
365 bool operator==(const const_iterator &it) const
366 /* Pre: cert */
367 /* Post: el resultat ens diu si els dos iteradors són iguals */
368 {
369 return plist == it.plist and pitem == it.pitem;
370 }
371
372 bool operator!=(const const_iterator &it) const
373 /* Pre: cert */
374 /* Post: el resultat ens diu si els dos iteradors són diferents */
375 {
376 return not (plist == it.plist and pitem == it.pitem);
377 }
378
379 };
380
381 const_iterator cbegin() const
382 /* Pre: cert */
383 /* Post: el resultat apunta al primer element del p.i o a l'element fictici
384 si el p.i. no té cap element no fictici */
385 {
386 const_iterator it;
387 it.plist = this;
388 it.pitem = iteminf.next;
389 return it;
390 }
391
392 const_iterator cend() const
393 /* Pre: cert */
394 /* Post: el resultat apunta a l'element fictici */
395 {
396 const_iterator it;
397 it.plist = this;
398 it.pitem = &itemsup;
399 return it;
400 }
401
402 const_iterator insert(const_iterator it, const T & value)
403 /* Pre: it ha d'apuntar a un element del p.i o a l'element fictici end() del
404 p.i */
405 /* Post: s'inserta un element amb valor value abans de l'element al que
406 apunta it i el resultat apunta al nou element inserit */
407 {
408 if (it.plist != this) {
409 cout << "Error: insert with an iterator not on this list" << endl;
410 exit(1);
411 }
412 const_iterator res = it;
413 insertItem(res.pitem->prev, value);

```

```

411 res.pitem = res.pitem->prev;
412 return res;
413 }
414
415 const_iterator erase(const_iterator it)
416 /* Pre: it ha d'apuntar a un element del p.i que no sigui l'end() */
417 /* Post: s'elimina l'element apuntat per it, el resultat apunta a l'element
posterior a l'eliminat */
418 {
419 if (it.plist != this) {
420 cout << "Error: erase with an iterator not on this list" << endl;
421 exit(1);
422 }
423 if (it.pitem == &itemsup) {
424 cout << "Error: erase with an iterator pointing to the end of the list" <<
endl;
425 exit(1);
426 }
427 const_iterator res = it;
428 res.pitem = res.pitem->next;
429 removeItem(res.pitem->prev);
430 return res;
431 }
432
433
434
435 };
436
437 // Implementation of read and write lists.
438 template <typename T> istream &operator>>(istream &is, List<T> &l)
439 {
440 l.removeItem();
441 int size;
442 cin >> size;
443 for (int i = 0; i < size; ++i) {
444 T value;
445 cin >> value;
446 l.insertItem(l.itemsup.prev, value);
447 }
448 return is;
449 }
450
451 template<typename T> ostream &operator<<(ostream &os, List<T> &l)
452 {
453 os << l._size;
454 for (typename List<T>::Item* pitem = l.iteminf.next; pitem != &l.itemsup;
pitem = pitem->next)
455 cout << " " << pitem->value;
456 return os;
457 }

```

Exercici: X75139 Mètode de llistes per a moure l'element apuntat per un iterador una posició cap al final.cc

```
1 #include <iostream>
2
3 using namespace std;
4
5 template <typename T> class List {
6 private:
7
8 // Items:
9 class Item {
10 public:
11 T value;
12 Item *next;
13 Item *prev;
14 };
15
16 // Data:
17 int _size;
18 Item iteminf, itemsup;
19
20 // Mètodes auxiliars
21 void insertItem(Item *pitemprev, Item *pitem)
22 /* Pre: pitemprev apunta a un element del p.i. pitemprev != pitem,
23    pitemprev.next = P, pitem != nullptr, pitem apunta a un element singular */
23 /* Post: inserta al p.i. el node apuntat per pitem entre el node apuntat per
24    pitemprev i P, augmenta _size en 1 */
24 {
25 pitem->next = pitemprev->next;
26 pitem->next->prev = pitem;
27 pitem->prev = pitemprev;
28 pitemprev->next = pitem;
29 _size++;
30 }
31
32 void insertItem(Item *pitemprev, const T &value)
33 /* Pre: pitemprev apunta un element del p.i. pitemprev.next = P */
34 /* Post: inserta al p.i. un node amb valor value entre el node apuntat per
35    pitemprev i P, augmenta _size en 1 */
35 {
36 Item *pitem = new Item;
37 pitem->value = value;
38 insertItem(pitemprev, pitem);
39 }
40
41 void extractItem(Item *pitem)
42 /* Pre: pitem != & iteminf, pitem != & itemsup pitem apunta a un element del
43    p.i. */
43 /* Post: enllaça doblement el node anterior a pitem amb el posterior a pitem,
44    disminueix en 1 _size */
44 {
45 pitem->next->prev = pitem->prev;
46 pitem->prev->next = pitem->next;
47 _size--;
48 }
49
50 void removeItem(Item *pitem)
```

```

51 /* Pre: pitem != &iteminf, pitem != &itemsup */
52 /* Post: enllaça doblement el node anterior a pitem amb el posterior a pitem,
s'allibera la memòria del node apuntat per pitem */
53 {
54 extractItem(pitem);
55 delete pitem;
56 }
57
58 void removeItems() {
59 /* Pre: _size = S */
60 /* Post: s'ha alliberat la memòria dels S nodes entre iteminf i itemsup,
iteminf.next = &itemsup, itemsup.prev = &iteminf, _size = 0 */
61 while (_size > 0)
62 removeItem(iteminf.next);
63 }
64
65 void copyItems(List & l) {
66 /* Pre: cert */
67 /* Post: copia els elements de la llista l al p.i. */
68 for (Item *pitem = l.itemsup.prev; pitem != &l.iteminf; pitem = pitem->prev)
69 insertItem(&iteminf, pitem->value);
70 }
71
72
73 public:
74
75 // Constructors/Destructors:
76 List()
77 /* Pre: cert */
78 /* Post: el resultat és una llista sense elements */
79 {
80 _size = 0;
81 iteminf.next = &itemsup;
82 itemsup.prev = &iteminf;
83 }
84
85 List(List &l)
86 /* Pre: cert */
87 /* Post: El resultat és una còpia d'l */
88 {
89 _size = 0;
90 iteminf.next = &itemsup;
91 itemsup.prev = &iteminf;
92 copyItems(l);
93 }
94
95 ~List()
96 // Destructora: Esborra automàticament els objectes locals en // sortir d'un
àmbit de visibilitat {
97 removeItems();
98 }
99
100 // Assignment:
101 List &operator=(const List &l)
102 /* Pre: cert */
103 /* Post: El p.i. passa a ser una còpia d'l i qualsevol contingut anterior del
p.i. ha estat esborrat (excepte si el p.i. i l ja eren el mateix objecte) */
104 {
105 if (this != &l) {
106 removeItems();
107 copyItems(l);
108 }
109 return *this;

```

```

110 }
111
112 // Standard operations:
113 int size() const
114 /* Pre: cert */
115 /* Post: el resultat és el nombre de nodes del p.i (els nodes iteminf i
itemsup no es conten) */
116 {
117 return _size;
118 }
119
120 bool empty() const
121 /* Pre: cert */
122 /* Post: el resultat és si el p.i té nodes o no (els nodes iteminf i itemsup
no conten) */
123 {
124 return _size == 0;
125 }
126
127
128 void push_back(const T &value)
129 /* Pre: cert */
130 /* Post: s'inserta un node amb valor value al final del p.i. */
131 {
132 insertItem(itemsup.prev, value);
133 }
134
135 void push_front(const T &value)
136 /* Pre: cert */
137 /* Post: s'inserta un node amb valor value al principi del p.i. */
138 {
139 insertItem(&iteminf, value);
140 }
141
142 void pop_back()
143 /* Pre: el p.i. no és buit */
144 /* Post: s'esborra el primer node del p.i. */
145 {
146 if (_size == 0) {
147 cerr << "Error: pop_back on empty list" << endl;
148 exit(1);
149 }
150 removeItem(itemsup.prev);
151 }
152
153 void pop_front()
154 /* Pre: el p.i. no és buit */
155 /* Post: s'esborra el darrer node del p.i. */
156 {
157 if (_size == 0) {
158 cerr << "Error: pop_front on empty list" << endl;
159 exit(1);
160 }
161 removeItem(iteminf.next);
162 }
163
164
165 // Read and write:
166 template <typename U> friend istream &operator>>(istream &is, List<U> &l);
167
168 template <typename U> friend ostream &operator<<(ostream &os, List<U> &l);
169
170

```

```

171 // Iterators mutables
172 class iterator {
173 friend class List;
174 private:
175 List *plist;
176 Item *pitem;
177 public:
178
179
180 // Preincrement iterator operator++()
181 /* Pre: el p.i apunta a un element E de la llista, que no és el end() */
182 /* Post: el p.i apunta a l'element següent a E el resultat és el p.i. */
183 {
184 if (pitem == &(amp;plist->itemsup)) {
185 cerr << "Error: ++iterator at the end of list" << endl;
186 exit(1);
187 }
188 pitem = pitem->next;
189 return *this;
190 }
191
192 // Postincrement iterator operator++(int)
193 /* Pre: p.i. = I, el p.i apunta a un element E de la llista, que no és el
end() */
194 /* Post: el p.i apunta a l'element següent a E, el resultat és I */
195 {
196 if (pitem == &(amp;plist->itemsup)) {
197 cerr << "Error: iterator++ at the end of list" << endl;
198 exit(1);
199 }
200 iterator aux = *this;
201 pitem = pitem->next;
202 return aux;
203 }
204
205 // Predecrement iterator operator--()
206 /* Pre: el p.i apunta a un element E de la llista que no és el begin() */
207 /* Post: el p.i apunta a l'element anterior a E, el resultat és el p.i. */
208 {
209 if (pitem == plist->iteminf.next) {
210 cerr << "Error: --iterator at the beginning of list" << endl;
211 exit(1);
212 }
213 pitem = pitem->prev;
214 return *this;
215 }
216
217 // Postdecrement iterator operator--(int)
218 /* Pre: p.i. = I, el p.i apunta a un element E de la llista que no és el
begin() */
219 /* Post: el p.i apunta a l'element anterior a E, el resultat és I */
220 {
221 if (pitem == plist->iteminf.next) {
222 cerr << "Error: iterator-- at the beginning of list" << endl;
223 exit(1);
224 }
225 iterator aux;
226 aux = *this;
227 pitem = pitem->prev;
228 return aux;
229 }
230
231 T& operator*( )

```

```

232 /* Pre: el p.i apunta a un element E de la llista, que no és el end() */
233 /* Post: el resultat és el valor de l'element E */
234 {
235     if (pitem == &(plist->itemsup)) {
236         cerr << "Error: ++iterator at the end of list" << endl;
237         exit(1);
238     }
239     return pitem->value;
240 }
241
242 bool operator==(const iterator &it) const
243 /* Pre: cert */
244 /* Post: el resultat ens diu si els dos iterators són iguals */
245 {
246     return plist == it.plist and pitem == it.pitem;
247 }
248
249 bool operator!=(const iterator &it) const
250 /* Pre: cert */
251 /* Post: el resultat ens diu si els dos iterators són diferents */
252 {
253     return not (plist == it.plist and pitem == it.pitem);
254 }
255
256 };
257
258 // Operations with iterators:
259 iterator begin ()
260 /* Pre: cert */
261 /* Post: el resultat apunta al primer element del p.i o a l'element fictici
si el p.i. no té cap element no fictici */
262 {
263     iterator it;
264     it.plist = this;
265     it.pitem = iteminf.next;
266     return it;
267 }
268
269 iterator end()
270 /* Pre: cert */
271 /* Post: el resultat apunta a l'element fictici */
272 {
273     iterator it;
274     it.plist = this;
275     it.pitem = &itemsup;
276     return it;
277 }
278
279 iterator insert(iterator it, const T & value)
280 /* Pre: it ha d'apuntar a un element del p.i o a l'element fictici end() del
p.i */
281 /* Post: s'inserta un element amb valor value abans de l'element al que
apunta it i el resultat apunta al nou element inserit */
282 {
283     if (it.plist != this) {
284         cout << "Error: insert with an iterator not on this list" << endl;
285         exit(1);
286     }
287     iterator res = it;
288     insertItem(res.pitem->prev, value);
289     res.pitem = res.pitem->prev;
290     return res;
291 }

```



```

292
293 iterator erase(iterator it)
294 /* Pre: it ha d'apuntar a un element del p.i que no sigui l'end() */
295 /* Post: s'elimina l'element apuntat per it, el resultat apunta a l'element
posterior a l'eliminat */
296 {
297     if (it.plist != this) {
298         cout << "Error: erase with an iterator not on this list" << endl;
299         exit(1);
300     }
301     if (it.pitem == &itemsup) {
302         cout << "Error: erase with an iterator pointing to the end of the list" <<
endl;
303         exit(1);
304     }
305     iterator res = it;
306     res.pitem = res.pitem->next;
307     removeItem(res.pitem->prev);
308     return res;
309 }
310
311
312 // Iteradors constants
313 class const_iterator {
314     friend class List;
315 private:
316     List const *plist;
317     Item const *pitem;
318
319 public:
320
321
322 // Preincrement const_iterator operator++()
323 /* Pre: el p.i apunta a un element E de la llista, que no és el end() */
324 /* Post: el p.i apunta a l'element següent a E, el resultat és el p.i. */
325 {
326     if (pitem == &(plist->itemsup)) {
327         cerr << "Error: ++iterator at the end of list" << endl;
328         exit(1);
329     }
330     pitem = pitem->next;
331     return *this;
332 }
333
334 // Postincrement const_iterator operator++(int)
335 /* Pre: p.i. = I, el p.i apunta a un element E de la llista, que no és el
end() */
336 /* Post: el p.i apunta a l'element següent a E, el resultat és I */
337 {
338     if (pitem == &(plist->itemsup)) {
339         cerr << "Error: iterator++ at the end of list" << endl;
340         exit(1);
341     }
342     const_iterator aux = *this;
343     pitem = pitem->next;
344     return aux;
345 }
346
347 // Predecrement const_iterator operator--()
348 /* Pre: el p.i apunta a un element E de la llista que no és el begin() */
349 /* Post: el p.i apunta a l'element anterior a E, el resultat és el p.i. */
350 {
351     if (pitem == plist->iteminf.next) {

```

```

352 cerr << "Error: --iterator at the beginning of list" << endl;
353 exit(1);
354 }
355 pitem = pitem->prev;
356 return *this;
357 }
358
359 // Postdecrement const_iterator operator--(int)
360 /* Pre: p.i. = I, el p.i apunta a un element E de la llista que no és el
begin() */
361 /* Post: el p.i apunta a l'element anterior a E, el resultat és I */
362 {
363 if (pitem == plist->iteminf.next) {
364 cerr << "Error: iterator-- at the beginning of list" << endl;
365 exit(1);
366 }
367 const_iterator aux;
368 aux = *this;
369 pitem = pitem->prev;
370 return aux;
371 }
372
373 const T& operator*()
374 /* Pre: el p.i apunta a un element E de la llista, que no és el end() */
375 /* Post: el resultat és el valor de l'element E */
376 {
377 return pitem->value;
378 }
379
380 bool operator==(const const_iterator &it) const
381 /* Pre: cert */
382 /* Post: el resultat ens diu si els dos iterators són iguals */
383 {
384 return plist == it.plist and pitem == it.pitem;
385 }
386
387 bool operator!=(const const_iterator &it) const
388 /* Pre: cert */
389 /* Post: el resultat ens diu si els dos iterators són diferents */
390 {
391 return not (plist == it.plist and pitem == it.pitem);
392 }
393
394 };
395
396 const_iterator cbegin() const
397 /* Pre: cert */
398 /* Post: el resultat apunta al primer element del p.i o a l'element fictici
si el p.i. no té cap element no fictici */
399 {
400 const_iterator it;
401 it.plist = this;
402 it.pitem = iteminf.next;
403 return it;
404 }
405
406 const_iterator cend() const
407 /* Pre: cert */
408 /* Post: el resultat apunta a l'element fictici */
409 {
410 const_iterator it;
411 it.plist = this;
412 it.pitem = &itemsup;

```

```

413 return it;
414 }
415
416 const_iterator insert(const_iterator it, const T & value)
417 /* Pre: it ha d'apuntar a un element del p.i o a l'element fictici end() del
p.i */
418 /* Post: s'inserta un element amb valor value abans de l'element al que
apunta it i el resultat apunta al nou element inserit */
419 {
420 if (it.plist != this) {
421 cout << "Error: insert with an iterator not on this list" << endl;
422 exit(1);
423 }
424 const_iterator res = it;
425 insertItem(res.pitem->prev, value);
426 res.pitem = res.pitem->prev;
427 return res;
428 }
429
430 const_iterator erase(const_iterator it)
431 /* Pre: it ha d'apuntar a un element del p.i que no sigui l'end() */
432 /* Post: s'elimina l'element apuntat per it, el resultat apunta a l'element
posterior a l'eliminat */
433 {
434 if (it.plist != this) {
435 cout << "Error: erase with an iterator not on this list" << endl;
436 exit(1);
437 }
438 if (it.pitem == &itemsup) {
439 cout << "Error: erase with an iterator pointing to the end of the list" <<
endl;
440 exit(1);
441 }
442 const_iterator res = it;
443 res.pitem = res.pitem->next;
444 removeItem(res.pitem->prev);
445 return res;
446 }
447
448 // Pre: it apunta a algun element de la llista implícita. // Post: it
continua apuntant al mateix element, el qual ha estat mogut una posició // cap al
final. En el cas en que l'element apuntat per it ja era l'últim, // res ha
canviat. // Descomenteu les següents dues línies i implementeu el mètode: void
moveTowardsEnd(iterator &it) {
449 Item *mogut = it.pitem;
450 Item *no_mogut = mogut->next;
451 if (mogut == itemsup.prev) return;
452 //desconnecto mogut mogut->prev->next = mogut->next;
453 mogut->next->prev = mogut->prev;
454
455 //modifico nodes mogut mogut->next = no_mogut->next;
456 mogut->prev = no_mogut;
457
458 //reconnecto mogut no_mogut->next->prev = mogut;
459 no_mogut->next = mogut;
460 }
461 };
462
463 // Implementation of read and write lists.
464 template <typename T> istream &operator>>(istream &is, List<T> &l)
465 {
466 l.removeItem();
467 int size;

```

```
468 cin >> size;
469 for (int i = 0; i < size; ++i) {
470     T value;
471     cin >> value;
472     l.insertItem(l.itemsup.prev, value);
473 }
474 return is;
475 }
476
477 template<typename T> ostream &operator<<(ostream &os, List<T> &l)
478 {
479     os << l._size;
480     for (typename List<T>::Item* pitem = l.iteminf.next; pitem != &l.itemsup;
481          pitem = pitem->next)
482         cout << " " << pitem->value;
483     return os;
484 }
```

Exercici: X96416 Mètode de llistes per intercanviar (swappejar) el primer i l'últim element.cc

```
1 #include <iostream>
2
3 using namespace std;
4
5 template <typename T> class List {
6 private:
7
8 // Items:
9 class Item {
10 public:
11 T value;
12 Item *next;
13 Item *prev;
14 };
15
16 // Data:
17 int _size;
18 Item iteminf, itemsup;
19
20 // Mètodes auxiliars
21 void insertItem(Item *pitemprev, Item *pitem)
22 /* Pre: pitemprev apunta a un element del p.i. pitemprev != pitem,
   pitemprev.next = P, pitem != nullptr, pitem apunta a un element singular */
23 /* Post: inserta al p.i. el node apuntat per pitem entre el node apuntat per
   pitemprev i P, augmenta _size en 1 */
24 {
25 pitem->next = pitemprev->next;
26 pitem->next->prev = pitem;
27 pitem->prev = pitemprev;
28 pitemprev->next = pitem;
29 _size++;
30 }
31
32 void insertItem(Item *pitemprev, const T &value)
33 /* Pre: pitemprev apunta un element del p.i. pitemprev.next = P */
34 /* Post: inserta al p.i. un node amb valor value entre el node apuntat per
   pitemprev i P, augmenta _size en 1 */
35 {
36 Item *pitem = new Item;
37 pitem->value = value;
38 insertItem(pitemprev, pitem);
39 }
40
41 void extractItem(Item *pitem)
42 /* Pre: pitem != & iteminf, pitem != & itemsup pitem apunta a un element del
   p.i. */
43 /* Post: enllaça doblement el node anterior a pitem amb el posterior a pitem,
   disminueix en 1 _size */
44 {
45 pitem->next->prev = pitem->prev;
46 pitem->prev->next = pitem->next;
47 _size--;
48 }
49
50 void removeItem(Item *pitem)
```

```

51 /* Pre: pitem != &iteminf, pitem != &itemsup */
52 /* Post: enllaça doblement el node anterior a pitem amb el posterior a pitem,
s'allibera la memòria del node apuntat per pitem */
53 {
54 extractItem(pitem);
55 delete pitem;
56 }
57
58 void removeItems() {
59 /* Pre: _size = S */
60 /* Post: s'ha alliberat la memòria dels S nodes entre iteminf i itemsup,
iteminf.next = &itemsup, itemsup.prev = &iteminf, _size = 0*/
61 while (_size > 0)
62 removeItem(iteminf.next);
63 }
64
65 void copyItems(List & l) {
66 /* Pre: cert*/
67 /* Post: copia els elements de la llista l al p.i. */
68 for (Item *pitem = l.itemsup.prev; pitem != &l.iteminf; pitem = pitem->prev)
69 insertItem(&iteminf, pitem->value);
70 }
71
72
73 public:
74
75 // Constructors/Destructors:
76 List()
77 /* Pre: cert*/
78 /* Post: el resultat és una llista sense elements */
79 {
80 _size = 0;
81 iteminf.next = &itemsup;
82 itemsup.prev = &iteminf;
83 }
84
85 List(List &l)
86 /* Pre: cert */
87 /* Post: El resultat és una còpia d'l */
88 {
89 _size = 0;
90 iteminf.next = &itemsup;
91 itemsup.prev = &iteminf;
92 copyItems(l);
93 }
94
95 ~List()
96 // Destructora: Esborra automàticament els objectes locals en // sortir d'un
àmbit de visibilitat {
97 removeItems();
98 }
99
100 // Assignment:
101 List &operator=(const List &l)
102 /* Pre: cert */
103 /* Post: El p.i. passa a ser una còpia d'l i qualsevol contingut anterior del
p.i. ha estat esborrat (excepte si el p.i. i l ja eren el mateix objecte) */
104 {
105 if (this != &l) {
106 removeItems();
107 copyItems(l);
108 }
109 return *this;

```

```

110 }
111
112 // Standard operations:
113 int size() const
114 /* Pre: cert */
115 /* Post: el resultat és el nombre de nodes del p.i (els nodes iteminf i
itemsup no es contenen) */
116 {
117 return _size;
118 }
119
120 bool empty() const
121 /* Pre: cert */
122 /* Post: el resultat és si el p.i té nodes o no (els nodes iteminf i itemsup
no contenen) */
123 {
124 return _size == 0;
125 }
126
127
128 void push_back(const T &value)
129 /* Pre: cert */
130 /* Post: s'inserta un node amb valor value al final del p.i. */
131 {
132 insertItem(itemsup.prev, value);
133 }
134
135 void push_front(const T &value)
136 /* Pre: cert */
137 /* Post: s'inserta un node amb valor value al principi del p.i. */
138 {
139 insertItem(&iteminf, value);
140 }
141
142 void pop_back()
143 /* Pre: el p.i. no és buit */
144 /* Post: s'esborra el primer node del p.i. */
145 {
146 if (_size == 0) {
147 cerr << "Error: pop_back on empty list" << endl;
148 exit(1);
149 }
150 removeItem(itemsup.prev);
151 }
152
153 void pop_front()
154 /* Pre: el p.i. no és buit */
155 /* Post: s'esborra el darrer node del p.i. */
156 {
157 if (_size == 0) {
158 cerr << "Error: pop_front on empty list" << endl;
159 exit(1);
160 }
161 removeItem(iteminf.next);
162 }
163
164
165 // Read and write:
166 template <typename U> friend istream &operator>>(istream &is, List<U> &l);
167
168 template <typename U> friend ostream &operator<<(ostream &os, List<U> &l);
169
170

```

```

171 // Iterators mutables
172 class iterator {
173 friend class List;
174 private:
175 List *plist;
176 Item *pitem;
177 public:
178
179
180 // Preincrement iterator operator++()
181 /* Pre: el p.i apunta a un element E de la llista, que no és el end() */
182 /* Post: el p.i apunta a l'element següent a E el resultat és el p.i. */
183 {
184 if (pitem == &(plist->itemsup)) {
185 cerr << "Error: ++iterator at the end of list" << endl;
186 exit(1);
187 }
188 pitem = pitem->next;
189 return *this;
190 }
191
192 // Postincrement iterator operator++(int)
193 /* Pre: p.i. = I, el p.i apunta a un element E de la llista, que no és el
end() */
194 /* Post: el p.i apunta a l'element següent a E, el resultat és I */
195 {
196 if (pitem == &(plist->itemsup)) {
197 cerr << "Error: iterator++ at the end of list" << endl;
198 exit(1);
199 }
200 iterator aux = *this;
201 pitem = pitem->next;
202 return aux;
203 }
204
205 // Predecrement iterator operator--()
206 /* Pre: el p.i apunta a un element E de la llista que no és el begin() */
207 /* Post: el p.i apunta a l'element anterior a E, el resultat és el p.i. */
208 {
209 if (pitem == plist->iteminf.next) {
210 cerr << "Error: --iterator at the beginning of list" << endl;
211 exit(1);
212 }
213 pitem = pitem->prev;
214 return *this;
215 }
216
217 // Postdecrement iterator operator--(int)
218 /* Pre: p.i. = I, el p.i apunta a un element E de la llista que no és el
begin() */
219 /* Post: el p.i apunta a l'element anterior a E, el resultat és I */
220 {
221 if (pitem == plist->iteminf.next) {
222 cerr << "Error: iterator-- at the beginning of list" << endl;
223 exit(1);
224 }
225 iterator aux;
226 aux = *this;
227 pitem = pitem->prev;
228 return aux;
229 }
230
231 T& operator*( )

```



```

232 /* Pre: el p.i apunta a un element E de la llista, que no és el end() */
233 /* Post: el resultat és el valor de l'element E */
234 {
235     if (pitem == &(plist->itemsup)) {
236         cerr << "Error: ++iterator at the end of list" << endl;
237         exit(1);
238     }
239     return pitem->value;
240 }
241
242 bool operator==(const iterator &it) const
243 /* Pre: cert */
244 /* Post: el resultat ens diu si els dos iteradors són iguals */
245 {
246     return plist == it.plist and pitem == it.pitem;
247 }
248
249 bool operator!=(const iterator &it) const
250 /* Pre: cert */
251 /* Post: el resultat ens diu si els dos iteradors són diferents */
252 {
253     return not (plist == it.plist and pitem == it.pitem);
254 }
255
256 };
257
258 // Operations with iterators:
259 iterator begin ()
260 /* Pre: cert */
261 /* Post: el resultat apunta al primer element del p.i o a l'element fictici
si el p.i. no té cap element no fictici */
262 {
263     iterator it;
264     it.plist = this;
265     it.pitem = iteminf.next;
266     return it;
267 }
268
269 iterator end()
270 /* Pre: cert */
271 /* Post: el resultat apunta a l'element fictici */
272 {
273     iterator it;
274     it.plist = this;
275     it.pitem = &itemsup;
276     return it;
277 }
278
279 iterator insert(iterator it, const T & value)
280 /* Pre: it ha d'apuntar a un element del p.i o a l'element fictici end() del
p.i */
281 /* Post: s'inserta un element amb valor value abans de l'element al que
apunta it i el resultat apunta al nou element inserit */
282 {
283     if (it.plist != this) {
284         cout << "Error: insert with an iterator not on this list" << endl;
285         exit(1);
286     }
287     iterator res = it;
288     insertItem(res.pitem->prev, value);
289     res.pitem = res.pitem->prev;
290     return res;
291 }

```

```

292
293 iterator erase(iterator it)
294 /* Pre: it ha d'apuntar a un element del p.i que no sigui l'end() */
295 /* Post: s'elimina l'element apuntat per it, el resultat apunta a l'element
posterior a l'eliminat */
296 {
297     if (it.plist != this) {
298         cout << "Error: erase with an iterator not on this list" << endl;
299         exit(1);
300     }
301     if (it.pitem == &itemsup) {
302         cout << "Error: erase with an iterator pointing to the end of the list" <<
endl;
303         exit(1);
304     }
305     iterator res = it;
306     res.pitem = res.pitem->next;
307     removeItem(res.pitem->prev);
308     return res;
309 }
310
311
312 // Iteradors constants
313 class const_iterator {
314     friend class List;
315 private:
316     List const *plist;
317     Item const *pitem;
318
319 public:
320
321
322 // Preincrement const_iterator operator++()
323 /* Pre: el p.i apunta a un element E de la llista, que no és el end() */
324 /* Post: el p.i apunta a l'element següent a E, el resultat és el p.i. */
325 {
326     if (pitem == &(plist->itemsup)) {
327         cerr << "Error: ++iterator at the end of list" << endl;
328         exit(1);
329     }
330     pitem = pitem->next;
331     return *this;
332 }
333
334 // Postincrement const_iterator operator++(int)
335 /* Pre: p.i. = I, el p.i apunta a un element E de la llista, que no és el
end() */
336 /* Post: el p.i apunta a l'element següent a E, el resultat és I */
337 {
338     if (pitem == &(plist->itemsup)) {
339         cerr << "Error: iterator++ at the end of list" << endl;
340         exit(1);
341     }
342     const_iterator aux = *this;
343     pitem = pitem->next;
344     return aux;
345 }
346
347 // Predecrement const_iterator operator--()
348 /* Pre: el p.i apunta a un element E de la llista que no és el begin() */
349 /* Post: el p.i apunta a l'element anterior a E, el resultat és el p.i. */
350 {
351     if (pitem == plist->iteminf.next) {

```

```

352 cerr << "Error: --iterator at the beginning of list" << endl;
353 exit(1);
354 }
355 pitem = pitem->prev;
356 return *this;
357 }
358
359 // Postdecrement const_iterator operator--(int)
360 /* Pre: p.i. = I, el p.i apunta a un element E de la llista que no és el
begin() */
361 /* Post: el p.i apunta a l'element anterior a E, el resultat és I */
362 {
363 if (pitem == plist->iteminf.next) {
364 cerr << "Error: iterator-- at the beginning of list" << endl;
365 exit(1);
366 }
367 const_iterator aux;
368 aux = *this;
369 pitem = pitem->prev;
370 return aux;
371 }
372
373 const T& operator*()
374 /* Pre: el p.i apunta a un element E de la llista, que no és el end() */
375 /* Post: el resultat és el valor de l'element E */
376 {
377 return pitem->value;
378 }
379
380 bool operator==(const const_iterator &it) const
381 /* Pre: cert */
382 /* Post: el resultat ens diu si els dos iteradors són iguals */
383 {
384 return plist == it.plist and pitem == it.pitem;
385 }
386
387 bool operator!=(const const_iterator &it) const
388 /* Pre: cert */
389 /* Post: el resultat ens diu si els dos iteradors són diferents */
390 {
391 return not (plist == it.plist and pitem == it.pitem);
392 }
393
394 };
395
396 const_iterator cbegin() const
397 /* Pre: cert */
398 /* Post: el resultat apunta al primer element del p.i o a l'element fictici
si el p.i. no té cap element no fictici */
399 {
400 const_iterator it;
401 it.plist = this;
402 it.pitem = iteminf.next;
403 return it;
404 }
405
406 const_iterator cend() const
407 /* Pre: cert */
408 /* Post: el resultat apunta a l'element fictici */
409 {
410 const_iterator it;
411 it.plist = this;
412 it.pitem = &itemsup;

```

```

413 return it;
414 }
415
416 const_iterator insert(const_iterator it, const T & value)
417 /* Pre: it ha d'apuntar a un element del p.i o a l'element fictici end() del
p.i */
418 /* Post: s'inserta un element amb valor value abans de l'element al que
apunta it i el resultat apunta al nou element inserit */
419 {
420 if (it.plist != this) {
421 cout << "Error: insert with an iterator not on this list" << endl;
422 exit(1);
423 }
424 const_iterator res = it;
425 insertItem(res.pitem->prev, value);
426 res.pitem = res.pitem->prev;
427 return res;
428 }
429
430 const_iterator erase(const_iterator it)
431 /* Pre: it ha d'apuntar a un element del p.i que no sigui l'end() */
432 /* Post: s'elimina l'element apuntat per it, el resultat apunta a l'element
posterior a l'eliminat */
433 {
434 if (it.plist != this) {
435 cout << "Error: erase with an iterator not on this list" << endl;
436 exit(1);
437 }
438 if (it.pitem == &itemsup) {
439 cout << "Error: erase with an iterator pointing to the end of the list" <<
endl;
440 exit(1);
441 }
442 const_iterator res = it;
443 res.pitem = res.pitem->next;
444 removeItem(res.pitem->prev);
445 return res;
446 }
447
448 // Pre: // Post: L'element que era el primer de la llista ha passat a ser
l'últim de la llista. // L'element que era l'últim de la llista ha passat a ser
el primer de la llista. // A part d'això, res més ha canviat. // En els casos
particulars en que hi havien 0 o 1 elements a la llista, res ha canviat. //
Descomenteu les següents dues línies i implementeu el mètode: void
swapFirstLast() {
449 if (_size < 2) return;
450 else {
451 Item *first = iteminf.next;
452 Item *last = itemsup.prev;
453
454 first->next->prev = last;
455 iteminf.next = last;
456 last->next = first->next;
457
458 last->prev->next = first;
459 first->prev = last->prev;
460
461 last->prev = &iteminf;
462 itemsup.prev = first;
463 first->next = &itemsup;
464 }
465 }
466 };

```

```
467
468 // Implementation of read and write lists.
469 template <typename T> istream &operator>>(istream &is, List<T> &l)
470 {
471     l.removeItem();
472     int size;
473     cin >> size;
474     for (int i = 0; i < size; ++i) {
475         T value;
476         cin >> value;
477         l.insertItem(l.itemsup.prev, value);
478     }
479     return is;
480 }
481
482 template<typename T> ostream &operator<<(ostream &os, List<T> &l)
483 {
484     os << l._size;
485     for (typename List<T>::Item* pitem = l.iteminf.next; pitem != &l.itemsup;
486          pitem = pitem->next)
487         cout << " " << pitem->value;
487     return os;
488 }
```