```
/*****************************************************************************
 * AUTHOR           : Nick Reardon
 * Assignment #4    : Deque To Queue
 * CLASS            : CS1D
 * SECTION          : MW - 2:30p
 * DUE DATE         : 02 / 10 / 20
 *****************************************************************************/
#ifndef _LINKEDDEQUE_H_
#define _LINKEDDEQUE_H_
#include <exception>
#include <sstream>



class Except : virtual public std::runtime_error {

protected:

    int error_number;               ///< Error number
    int error_offset;               ///< Error offset

public:

    /** Constructor (C++ STL string, int, int).
     *  @param  msg      The error message
     *  @param  err_num  Error number
     *  @param  err_off  Error offset
     */
    explicit
        Except(const std::string& msg, int err_num, int err_off) :
        std::runtime_error(msg)
    {
        error_number = err_num;
        error_offset = err_off;

    }

    /** Destructor.
     *  Virtual to allow for subclassing.
     */
    virtual ~Except() throw () {}

    /** Returns error number.
     *  @return #error_number
     */
    virtual int getErrorNumber() const throw() {
        return error_number;
    }

    /**Returns error offset.
     * @return #error_offset
     */
```

```
53      virtual int getErrorOffset() const throw() {
54          return error_offset;
55      }
56
57  };
58
59  enum ERROR_TYPE
60  {
61      DEFUALT,
62      FULL,
63      EMPTY,
64      OUT_OF_RANGE
65  };
66
67  template <class Type>
68  struct Node
69  {
70      Type value;
71
72      Node<Type>* prev;
73      Node<Type>* next;
74
75      Node<Type>(const Type& newValue, Node<Type>* prevNode, Node<Type>* nextNode)
76      {
77          value = newValue;
78
79          prev = prevNode;
80          next = nextNode;
81      }
82
83  };
84
85  template <class Type>
86  class LinkedDeque
87  {
88  private:
89      Node<Type>* head;
90      Node<Type>* tail;
91
92      int capacity;
93      int currentSize;
94
95
96  protected:
97
98  public:
99
100     LinkedDeque<Type>(const int newCapacity = 32);
101
102     LinkedDeque<Type>(const LinkedDeque<Type>& otherDeque);
103
104     ~LinkedDeque();
```

```
105
106     void destroy();
107
108     bool empty() const;
109     bool full() const;
110     int size() const;
111
112     void insertBefore(const Type& newItem, const int index);
113     void insertAfter(const Type& newItem, const int index);
114
115     void insertFront(const Type& newItem);
116     void insertBack(const Type& newItem);
117
118     void eraseFront();
119     void eraseBack();
120
121
122     Type front() const;
123     Type back() const;
124
125     void printAll(std::ostream& output) const;
126
127 };
128
129
130 template<class Type>
131 inline LinkedDeque<Type>::LinkedDeque(const int newCapacity)
132 {
133     head = nullptr;
134     tail = nullptr;
135
136     capacity = newCapacity;
137     currentSize = 0;
138 }
139
140 template<class Type>
141 inline LinkedDeque<Type>::LinkedDeque(const LinkedDeque<Type>& otherDeque)
142 {
143     capacity = otherDeque.capacity;
144     currentSize = 0;
145
146     head = nullptr;
147     tail = nullptr;
148
149     for (Node<Type>* temp = otherDeque.head; temp != nullptr; temp = temp->next)
150     {
151         insertBack(temp->value);
152     }
153 }
154
155
156 template<class Type>
```

```
157 inline LinkedDeque<Type>::~LinkedDeque()
158 {
159     destroy();
160 }
161
162 template<class Type>
163 inline bool LinkedDeque<Type>::empty() const
164 {
165     bool value = (currentSize == 0 && head == nullptr && tail == nullptr);
166
167     return value;
168 }
169
170 template<class Type>
171 inline bool LinkedDeque<Type>::full() const
172 {
173     bool value = (currentSize == capacity);
174
175     return value;
176 }
177
178 template<class Type>
179 inline void LinkedDeque<Type>::destroy()
180 {
181     for (Node<Type>* temp = head; temp != nullptr; )
182     {
183         Node<Type>* hold = temp;
184         temp = temp->next;
185         delete hold;
186     }
187
188     head = nullptr;
189     tail = nullptr;
190
191 }
192
193 template<class Type>
194 inline void LinkedDeque<Type>::insertBefore(const Type& newItem, const int index)
195 {
196     if (empty())
197     {
198         throw Except("container is empty", EMPTY, 5);
199     }
200     else if (full())
201     {
202         throw Except("container is full", FULL, 5);
203     }
204     else
205     {
206         if (index <= (size() - 1 / 2))
207         {
208             Node<Type>* ptr = head;
```

```
209             for (int i = 0; i < index; i++)
210             {
211                 ptr = ptr->next;
212             }
213             Node<Type>* temp = new Node<Type>(newItem, nullptr, nullptr);
214 
215             temp->prev = ptr->prev;
216             temp->next = ptr;
217 
218             ptr->prev = temp;
219             temp->prev->next = temp;
220 
221             temp = nullptr;
222             ptr = nullptr;
223 
224             currentSize++;
225         }
226         else
227         {
228             Node<Type>* ptr = head;
229             for (int i = size() - 1; i >= size() - 1 - index; i--)
230             {
231                 ptr = ptr->next;
232             }
233             Node<Type>* temp = new Node<Type>(newItem, nullptr, nullptr);
234 
235             temp->prev = ptr->prev;
236             temp->next = ptr;
237 
238             ptr->prev = temp;
239             temp->prev->next = temp;
240 
241             temp = nullptr;
242             ptr = nullptr;
243 
244             currentSize++;
245         }
246     }
247 
248 }
249 
250 template<class Type>
251 inline void LinkedDeque<Type>::insertAfter(const Type& newItem, const int index)
252 {
253     if (empty())
254     {
255         throw Except("container is empty", EMPTY, 5);
256     }
257     else if (full())
258     {
259         throw Except("container is full", FULL, 5);
260     }
```

```
261     else
262     {
263         if (index <= (size() - 1 / 2))
264         {
265             Node<Type>* ptr = head;
266             for (int i = 1; i <= index; i++)
267             {
268                 ptr = ptr->next;
269             }
270             Node<Type>* temp = new Node<Type>(newItem, nullptr, nullptr);
271 
272             temp->next = ptr->next;
273             ptr->next = temp;
274 
275             temp->prev = ptr;
276             temp->next->prev = temp;
277 
278             temp = nullptr;
279             ptr = nullptr;
280 
281             currentSize++;
282         }
283         else
284         {
285             Node<Type>* ptr = head;
286             for (int i = size() - 1; i >= size() - 1 - index; i--)
287             {
288                 ptr = ptr->next;
289             }
290             Node<Type>* temp = new Node<Type>(newItem, nullptr, nullptr);
291 
292             temp->next = ptr->next;
293             ptr->next = temp;
294 
295             temp->prev = ptr;
296             temp->next->prev = temp;
297 
298             temp = nullptr;
299             ptr = nullptr;
300 
301             currentSize++;
302         }
303     }
304 
305 }
306 
307 template<class Type>
308 inline void LinkedDeque<Type>::insertFront(const Type& newItem)
309 {
310 
311     if (!full())
312     {
```

```
313         if (head == 0)
314         {
315             head = new Node<Type>(newItem, nullptr, nullptr);
316             tail = head;
317         }
318         else
319         {
320             Node<Type>* temp = new Node<Type>(newItem, nullptr, nullptr);
321 
322             head->prev = temp;
323 
324             temp->next = head;
325             temp->prev = nullptr;
326 
327             head = temp;
328 
329             temp = nullptr;
330         }
331         currentSize++;
332 
333     }
334     else
335     {
336         throw Except("container is full", FULL, 5);
337     }
338 }
339 
340 template<class Type>
341 inline void LinkedDeque<Type>::insertBack(const Type& newItem)
342 {
343 
344     if (!full())
345     {
346         if (head == 0)
347         {
348             head = new Node<Type>(newItem, nullptr, nullptr);
349             tail = head;
350         }
351         else
352         {
353             Node<Type>* temp = new Node<Type>(newItem, nullptr, nullptr);
354 
355             tail->next = temp;
356 
357             temp->prev = tail;
358             temp->next = nullptr;
359 
360             tail = temp;
361 
362             temp = nullptr;
363         }
364         currentSize++;
```

```
365
366     }
367     else
368     {
369         throw Except("container is full", FULL, 5);
370     }
371
372 }
373
374 template<class Type>
375 inline void LinkedDeque<Type>::eraseFront()
376 {
377
378     if (!empty())
379     {
380         if (head == tail)
381         {
382             delete head;
383             head = nullptr;
384             tail = nullptr;
385         }
386         else
387         {
388             Node<Type>* temp = head;
389             head = head->next;
390             head->prev = nullptr;
391             delete temp;
392         }
393         currentSize--;
394
395     }
396     else
397     {
398         throw Except("container is empty", EMPTY, 5);
399     }
400
401 }
402
403 template<class Type>
404 inline void LinkedDeque<Type>::eraseBack()
405 {
406     if (!empty())
407     {
408         if (head == tail)
409         {
410             delete head;
411             head = nullptr;
412             tail = nullptr;
413         }
414         else
415         {
416             Node<Type>* temp = tail;
```

```
417             tail = tail->prev;
418             tail->next = nullptr;
419             delete temp;
420         }
421         currentSize--;
422 
423     }
424     else
425     {
426         throw Except("container is empty", EMPTY, 5);
427     }
428 
429 }
430 
431 template<class Type>
432 inline int LinkedDeque<Type>::size() const
433 {
434     return currentSize;
435 }
436 
437 
438 template<class Type>
439 inline Type LinkedDeque<Type>::front() const
440 {
441     return head->value;
442 }
443 
444 template<class Type>
445 inline Type LinkedDeque<Type>::back() const
446 {
447     return tail->value;
448 }
449 
450 template<class Type>
451 inline void LinkedDeque<Type>::printAll(std::ostream& output) const
452 {
453     for (Node<Type>* temp = head; temp != nullptr; temp = temp->next)
454     {
455         output << temp->value << '\n';
456     }
457 }
458 
459 
460 
461 template<class Type>
462 class LinkedQueue : protected LinkedDeque<Type>
463 {
464 private:
465     LinkedDeque<Type> deque;
466 
467 protected:
468 
```

```
469 public:
470
471     LinkedQueue<Type>(const int newCapacity = 32);
472
473     LinkedQueue<Type>(LinkedQueue<Type>& otherQueue);
474     LinkedQueue<Type>(LinkedDeque<Type>& otherDeque);
475
476     ~LinkedQueue();
477
478     void destroy();
479
480     bool empty() const;
481     bool full() const;
482     int size() const;
483
484     void enqueue(const Type& newItem);
485     void dequeue();
486
487     Type front() const;
488     Type back() const;
489
490     void printAll(std::ostream& output) const;
491
492 };
493
494
495
496 #endif // !_LINKEDDEQUE_H_
497
498 template<class Type>
499 inline LinkedQueue<Type>::LinkedQueue(const int newCapacity) :
500     LinkedDeque<Type>(newCapacity)
501 {
502
503 }
504
505 template<class Type>
506 inline LinkedQueue<Type>::LinkedQueue(LinkedQueue<Type>& otherQueue) :
507     LinkedDeque<Type>(otherQueue)
508 {
509
510 }
511
512 template<class Type>
513 inline LinkedQueue<Type>::LinkedQueue(LinkedDeque<Type>& otherDeque) :
514     LinkedDeque<Type>(otherDeque)
515 {
516
517 }
518 template<class Type>
519 inline void LinkedQueue<Type>::destroy()
520 {
```

```
521     deque.destroy();
522 }
523
524 template<class Type>
525 inline LinkedQueue<Type>::~LinkedQueue()
526 {
527     deque.destroy();
528 }
529
530 template<class Type>
531 inline bool LinkedQueue<Type>::empty() const
532 {
533     return deque.empty();
534 }
535
536 template<class Type>
537 inline bool LinkedQueue<Type>::full() const
538 {
539     return deque.full();
540 }
541
542 template<class Type>
543 inline int LinkedQueue<Type>::size() const
544 {
545     return deque.size();
546 }
547
548 template<class Type>
549 inline void LinkedQueue<Type>::enqueue(const Type& newItem)
550 {
551     deque.insertBack(newItem);
552 }
553
554 template<class Type>
555 inline void LinkedQueue<Type>::dequeue()
556 {
557     deque.eraseFront();
558 }
559
560 template<class Type>
561 inline Type LinkedQueue<Type>::front() const
562 {
563     return deque.front();
564 }
565
566 template<class Type>
567 inline Type LinkedQueue<Type>::back() const
568 {
569     return deque.back();
570 }
571
572 template<class Type>
```

```
573 inline void LinkedQueue<Type>::printAll(std::ostream& output) const
574 {
575     deque.printAll(output);
576 }
577
578
579
580 template<class Type>
581 void PrintWithLabel(const std::string& label, const LinkedDeque<Type> &container, ↵
       std::ostream& output)
582 {
583     output << '\n' << label << '\n';
584
585     if (container.empty())
586     {
587         output << "Container is empty \n\n";
588
589     }
590     else
591     {
592         container.printAll(output);
593         output << '\n';
594     }
595 }
596
597 template<class Type>
598 void PrintWithLabel(const std::string& label, const LinkedQueue<Type>& container, ↵
       std::ostream& output)
599 {
600     output << '\n' << label << '\n';
601
602     if (container.empty())
603     {
604         output << "Container is empty \n\n";
605
606     }
607     else
608     {
609         container.printAll(output);
610         output << '\n';
611     }
612 }
```