Worksheet 20 - Group 1

Worksheet Group 1 Members

Marc Clinedinst: clinedim@onid.oregonstate.edu
Kelby Faessler: faesslek@onid.oregonstate.edu
James Fitzwater: fitzwatj@onid.oregonstate.edu
Tom Gariepy: gariepyt@onid.oregonstate.edu
Sean Reilly: reillys@onid.oregonstate.edu
Joseph Struth: struthj@onid.oregonstate.edu

Worksheet 20: Dynamic Array Deque and Queue

In this worksheet, we implement the functions that define the behavior of the deque and queue data structures when they are implemented on top of a dynamic array. More specifically, we implement the `dequeAddFront`, `dequeAddBack`, `dequeFront`, `dequeBack`, `dequeRemoveFront`, and `dequeRemoveBack` functions. The code for these functions is provided below, along with comments which explain the inner workings of each function.

```
/*
     This function adds a value to the front of the deque.  Before performing the addition, the
     function checks to see whether the deque needs to be resized; if this is the case, then
     the function performs the resize using the provided _dequeSetCapacity function. The
     function then moves onto adding the value to the front of the deque.  If the deque is
     empty, then the function adds the value at the index that currently marks the start of
     the deque.  Otherwise, the function computes the index where the value should be inserted.
     This computation looks complex on first sight, but it simply leverages the "wrapping"
     behavior of the modulus operator described in the following Wikipedia article:
     https://en.wikipedia.org/wiki/Modular_arithmetic
     In a language like Python, the computation would simply be (start - 1) % capacity, but C
     has a . . . weird implementation of the modulus operator.
*/
```

```
void dequeAddFront(struct deque *d, TYPE newValue) {
	int index;

	if (dequeSize(d) >= d->capacity) {
		_dequeSetCapacity(d, 2 * d->capacity);
	}

	if (dequeSize(d) == 0) {
		d->data[d->start] = newValue;
	} else {
		index = (((d->start - 1) % d->capacity) + d->capacity) % d->capacity;
		d->data[index] = newValue;
		d->start = index;
	}
	d->size++;
}

/*
	This function returns the value at the front of the deque.  Given that this operation
	should not be performed on an empty deque, it first checks that the size of the deque
	is greater than 0; the program will terminate with an assertion error if this condition is
	not met.  If the assertion passes, then the value at the index marking the start of the
	deque is returned.
*/
TYPE dequeFront (struct deque *d) {
	assert(dequeSize(d) > 0);
	return d->data[d->start];
}


/*
```

```
     This function removes the value at the front of the deque.  Given that this operation
     should not be performed on an empty deque, it first checks that the size of the deque is
     greater than 0; the program will terminate with an assertion error if this condition is
     not met.  If the assertion passes, then the function checks to see if the deque's size is
     not 1. If this is true, then the starting position is shifted to the next value's
     position.  In either case, the size of the deque is decremented.
*/
void dequeRemoveFront(struct deque *d) {
     assert(dequeSize(d) > 0);
     if (dequeSize(d) != 1) {
          d->start = (d->start + 1) % d->capacity;
     }

     d->size--;
}

/*
     This function adds a value at the back of the deque.  Before performing this addition, the
     function first checks to see whether the deque needs to be resized.  If a resize does need
     to occur, the function performs this resize.  The function then performs the addition at
     the appropriate index, which is again calculated using the modulus operator, and increases
     the size of the deque.
*/
void dequeAddBack(struct deque *d, TYPE value) {
     if (dequeSize(d) >= d->capacity) {
          _dequeSetCapacity(d, 2 * d->capacity);
     }

     d->data[(d->start + dequeSize(d)) % d->capacity] = value;
     d->size++;
}
```

```
/*
     This function returns the value at the back of the deque.  Given that this operation
     should not occur on an empty deque, the function first checks that there are values in
     the deque.  If this is not the case, then the function will throw an assertion error,
     and the program will terminate.  Otherwise, the value at the back of the deque will be
     returned; the index of this value is once again calculated using modular arithmetic.
*/
TYPE dequeBack(struct deque *d) {
     assert(dequeSize(d) > 0);
     return d->data[(d->start + dequeSize(d) - 1) % d->capacity];
}


/*
     This function removes the value from the back of the deque.  Given that this operation
     should not occur on an empty deque, the function first checks that there are values in
     the deque.  If this is not hte case, then the function will throw an assertion error, and
     the program will terminate.  Otherwise, the value at the back of the deque will be
     removed; this removal is accomplished by simply decreasing the size of the deque.
*/
void dequeRemoveBack(struct deque *d) {
     assert(dequeSize(d) > 0);
     d->size--;
}
```

Piazza Discussion

https://piazza.com/class/ib2kus4hsie528?cid=118