Group 20

Edward Francis
Danielle Goodman
Justin Kruse
Charlotte Murphy
Garret Sweetwood
Woohyuk Yang

Worksheet 20: Dynamic Array Deque and Queue

In Preparation: Read Chapter 7 to learn more about the Queue and Deque data types. If you have not done it already, complete worksheets 14 and 16 to learn more about the basic features of the dynamic array data type.

A question in the worksheet on the dynamic array stack asked why you cannot use a *Dynamic Array* as the basis for an efficient queue. This is because adding to or removing from the first location (that is, index position 0) is very slow O(n).

Removing a value requires elements to slide left. Sliding left means every element must be moved, and hence is an O(n) operation.

Adding a value requires elements to slide right. Sliding right opens up the location where a new value can be inserted. Once more, however, every element must be moved, and hence this is an O(n) operation.

This does not mean that a Dynamic Array-*like* data structure cannot be used to implement a queue. The key insight is that we can allow the starting location of the block of elements to "float", rather than being fixed at location zero. An internal integer data field records the current starting location.

Notice that the "logical" index no longer corresponds to the physical index. The value with logical index zero is found in this diagram at array location 2. The value with logical index 1 is found at location 3, and so on.

With this change, it is now easy to implement additions or removal from either front or back. To add to the front, simply decrement the starting location, and place the new element in the new starting place. To

add to the back, simply increase the size, and place the new value in the location determined by the addition of the starting location and size. But there is one subtle complexity. The block of values stored in the collection can wrap around from the end back to the beginning:

Here the block of elements begins at index position 7. The next three elements are found in index positions 8, 9 and 10. But the element after that is found at index position zero. As with the Dynamic Array, the internal array must be reallocated and doubled if the count of elements becomes equal to the array capacity. The internal function _dequeSetCapacity will set the size of the internal buffer to the passed in capacity. The code for this function is written for you below. You should study this to see how the variable named "j" can wrap around the end of the data array as the values are copied into the new array. The new array always begins with the starting position set to zero.

Using this idea, complete the implementation of the Deque. Implement the methods that will add, retrieve, or remove an element from either the front or back. Explain why each operation will have constant (or amortized constant) execution time.

```
void _dequeSetCapacity (struct deque *d, int newCap) {
        int i;

        /* Create a new underlying array*/
        TYPE *newData = (TYPE*)malloc(sizeof(TYPE)*newCap);
        assert(newData != 0);

        /* copy elements to it */
        int j = d->beg;
        for(i = 0; i <  d->size; i++)
        {
        newData[i] = d->data[j];
        j = j + 1;
        if(j >= d->capacity)
                j = 0;
        }

        /* Delete the oldunderlying array*/
        free(d->data);
        /* update capacity and size and data*/
        d->data = newData;
        d->capacity = newCap;
        d->beg = 0;
}

void dequeFree (struct deque *d) {
  free(d->data); d->size = 0; d->capacity = 0;
}
```

```
struct deque {
  TYPE * data;
  int capacity;
  int size;
  int beg;
};

void dequeInit (struct deque *d, int initCapacity) {
  d->size = d->beg = 0;
  d->capacity = initCapacity; assert(initCapacity > 0);
  d->data = (TYPE *) malloc(initCapacity * sizeof(TYPE));
  assert(d->data != 0);
}

int dequeSize (struct deque *d) { return d->size; }
```

/*each of the functions for add front, add back, get front, and get back, have near constant execution time, because they are not dependent on the size n of the dataset, in general. These operations are O(1)+ according to the terminology adopted for amortized performance analysis. Execution is O(n) only in those circumstances in which the array must be resized and all of the n values must be copied over to a new dynamically-allocated array. This occurs for the add front and add back functions, when the array is full at execution time.*/

```
void dequeAddFront (struct deque *d, TYPE newValue) {
        if (d->size >= d->capacity)
                _dequeSetCapacity(d, 2*d->capacity);
        d->beg--;
        if(d->beg < 0)
                d->beg = d->capacity -1;
        d->data[d->beg] = newValue;
        d->size++;
}

void dequeAddBack (struct deque *d, TYPE newValue) {
        if (d->size >= d->capacity)
                _dequeSetcapacity(d, 2* d->capacity);
        int index = d->beg + d->size;
        if(index >= d->capacity)
                index = index – d->capacity;
        d->data[index] = newValue;
        d->size++;
}


TYPE dequeFront (struct deque *d) {
        assert(d ! = 0);
        assert( d->size != 0);
        return d->data[d->beg];
```

```
}


TYPE dequeBack (struct deque *d) {
        assert (d ! = 0);
        assert (d->size != 0);
        int index = d->beg + d->size -1;
        if(index >= d->capacity)
                index = index – d->capacity;
        return d->data[index];
}


void dequeRemoveFront (struct deque *d) {
        assert(d ! = 0);
        assert (d->size != 0);
        d->beg++;
        if(d->beg == d->capacity)
                d->beg = 0;
        d->size--;
}

void dequeRemoveBack (struct deque *d) {
        assert(d ! = 0);
        assert(d->size != 0);
        d->size--;
}
```