COSC160: Data Structures
Dynamic and Circular Arrays

Jeremy Bolton

GEORGETOWN UNIVERSITY

Outline

I. Dynamic Arrays
II. Circular Arrays

Arrays

- Array limitations
  - Arrays are stored contiguously in a computer, and thus the size must remain fixed. However, one can implement a dynamic array which provides for an extra level of abstraction (in this case extra memory management behind the scenes)

- “Resize” it!
  - If the array is full, copy items from full array to a new array of larger size

Simple ReSizing Scheme

- When initialized, allocate enough memory for a “reasonable” storage amount.
- If the array is full and an attempt is made to add an item to the array, simply create a new, larger array and copy over the items from the previous array and then the new item to add.
- Since allocating a new array and copying the current list is computationally expensive, it is best to try to reduce the number of times the array reaches its max capacity.
  - One simple scheme is to assure the new array created (when max capacity is reached) is “large” (double the original is common).
- Exercise: Assume the maximum number of entries to be placed into a list is n and is unknown. If a dynamic array is used to implement this list, what is the number of times we would need to re-allocated and copy in the worst case?

Allocation and Copying are slow

- Copying a Dynamic Array to increase capacity when full
  - Minimize the number of copies
  - Difficult to do without knowing the max capacity needed

- Exponential Growth Scheme
  - Assume the max capacity is N (which is unknown *apriori*)
  - Assume array is initialized to size i.
  - How many copies are needed if we follow the following re-sizing rule? **When attempting to add element and capacity is full, allocate a new array with double capacity and copy.**
  - Total number of allocation-copy events: $i\ log_2(N - i)$

Dynamic Array Investigation: Add elements

- To the beginning, will result in $\Theta(n)$ operations
- To the middle, will result in $\Theta(n)$ operations (average case)
- To the end of the list will result in $\Theta(1)$ operations
  - UNLESS the array is full, then the result would be $\Theta(n)$ operations
    - Good news: we were able to add items to a seemingly full array!
    - Bad news: the time complexity is poor when the array is copied.
- Permits dynamic size change, but improvement can be made upon a standard array.

Dynamic Array Investigation: Remove elements

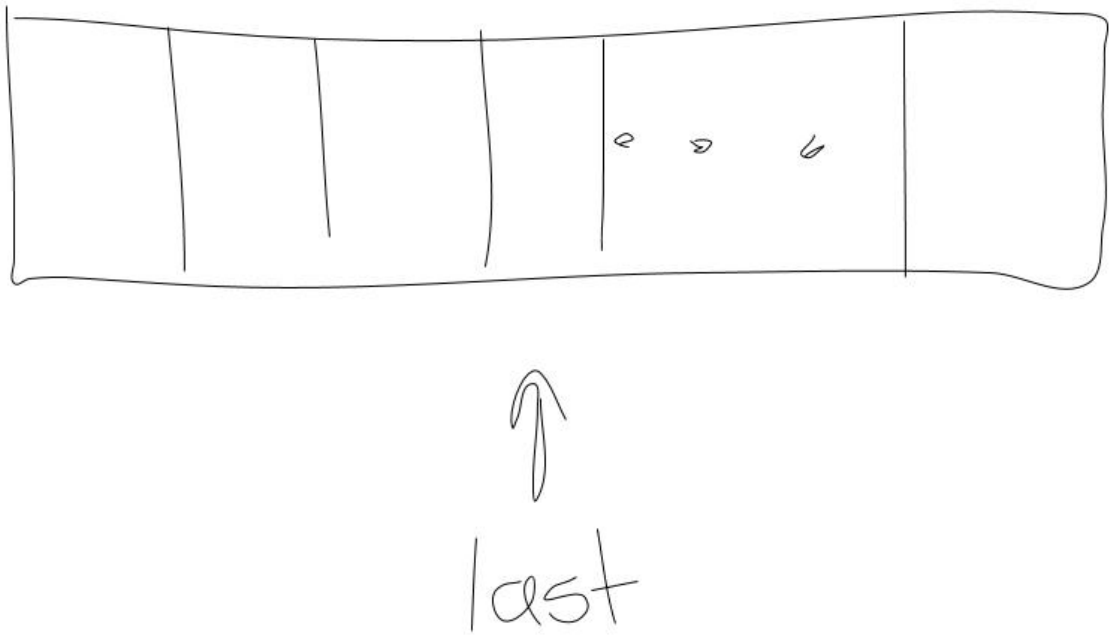
- From the beginning of the list will result in $\Theta(n)$ operations
- From the middle of the list will result in $\Theta(n)$ operations (average case)
- From the end of the list will result in $\Theta(1)$ operations

Can we improve upon this?

- The dynamic array appears to help with some memory issues encountered by the standard array, but some operations are still "slow"
  - Add to back is fast. Why? Index is generally know.
  - Add to front. Must copy and make room. Slow.
  - Reason: beginning of array is assume index 0 (or 1), but what if this was not required?

Dynamic Array: Access to Last Item

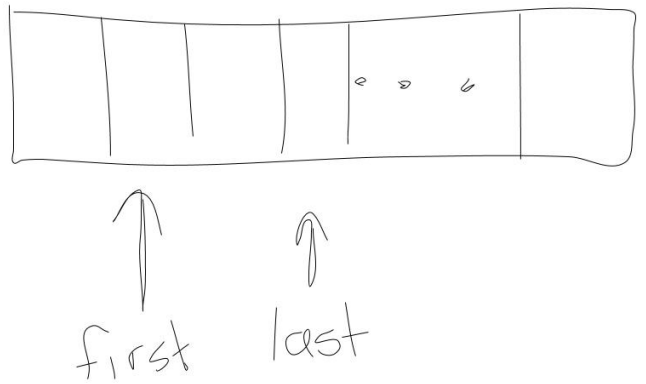
- As with standard arrays we will need to keep track of the number of items in the array – but more specifically we need to have the index of the current last item in the array
- Observation: Linked Lists permit efficient additions to beginning and end of lists by maintaining a pointer to the head and tail.
- We can apply this same concept to the first element of the array which can improve flexibility and complexity



Design Improvement: Circular Arrays

- We might also allow access to the first item in the array.
  - Design decision: if we require the "first" item in the array to be the item located at index zero, we will incur a $\Theta(n)$ copies.
  - Keep the index to the first item in the array (the first item does not need to be at location 0).
    - Reduces insertions to front and removals from front to $\Theta(1)$ steps. (with some minimal indexing overhead)

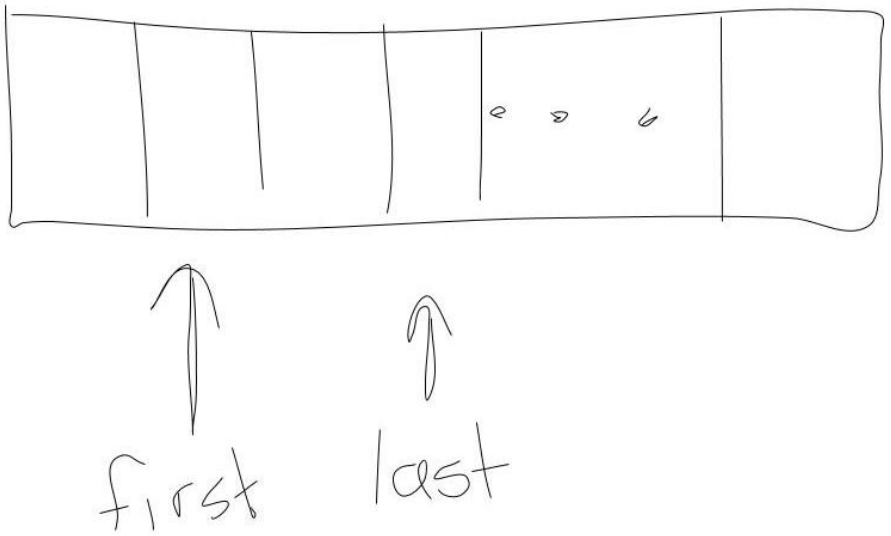
- **<u>Circular Arrays</u>**
  - Allow for both front and last indexes to change
  - Can use modular arithmetic to find relative indexes

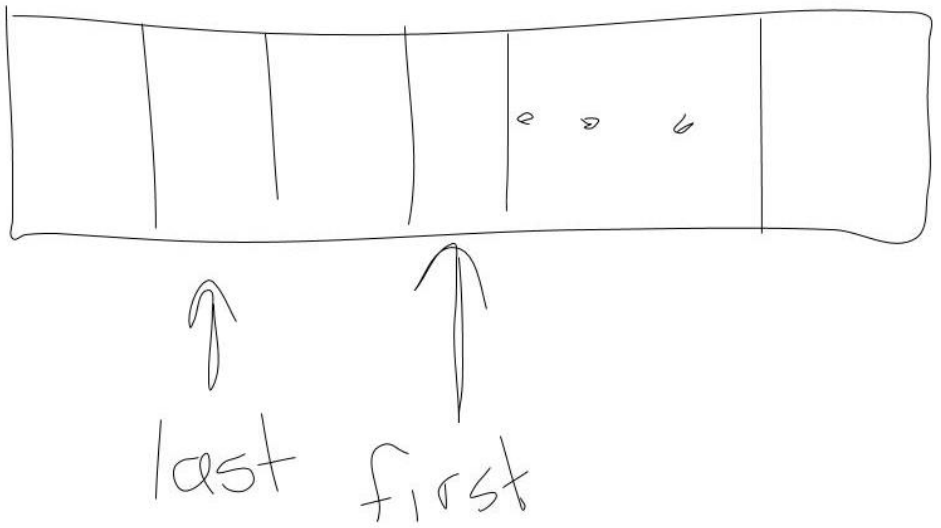


Circular Array: Indexes to both first and last item

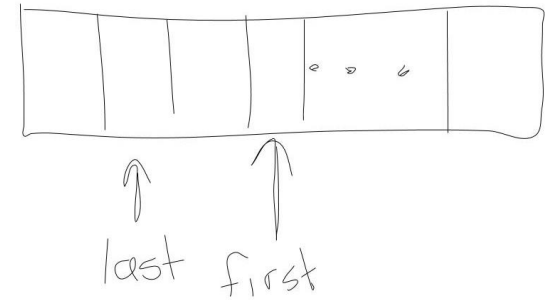
- Allows easy access to first and last item (with some simple overhead)

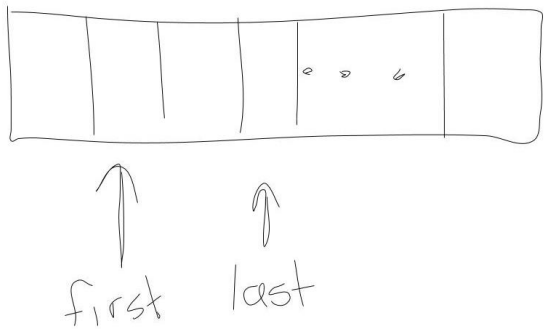
- There are two keys “states” of the list that should be recognized (for a traversal). ALLOWS “WRAP-AROUND”.





Efficient Arrays: Add To Front





Dynamic Circular Arrays

- At home Exercise. Write Pseudocode for the following:
  - addToBack(item i)
  - removeFromFront()
  - removeFromBack()

Circular Dynamic Arrays: Time Complexity

- With the addition of some simple design elements, we can make arrays efficient (in most cases).
- If the array is full, we will incur an overhead of $\Theta(n)$ for the allocation (if doubling size) and $\Theta(n)$ for the copy.
- Space complexity:
  - Chains use only the memory necessary to store the current list
  - Unless the size of the list is known at array allocation time and remains static, an array may use an unnecessary amount of memory.

IMPLEMENTATION	CHAINING***	ARRAYS*
INSERT FRONT	$\Theta(1)$	$\Theta(1)$**
INSERT MIDDLE	$\Theta(n)$	$\Theta(n)$
INSERT BACK	$\Theta(1)$	$\Theta(1)$**
RETRIEVE FRONT	$\Theta(1)$	$\Theta(1)$
RETRIEVE MIDDLE	$\Theta(n)$	$\Theta(1)$
RETRIEVE BACK	$\Theta(1)$	$\Theta(1)$
REMOVE FRONT	$\Theta(1)$	$\Theta(1)$
REMOVE MIDDLE	$\Theta(n)$	$\Theta(n)$
REMOVE BACK	$\Theta(1)$	$\Theta(1)$
* Assumes first and last index		
** Assumes alloc	*** Assumes Tail	

GEORGETOWN UNIVERSITY

Lists with restricted access

- Note: we have identified that we can make efficient use of dynamic arrays if access is restricted to first or last item.
- Are lists that restrict access to just the first or last item useful?
- Queues…