CS 261
Data Structures

Lecture 10

Stacks, Queues, and Deques (cont.)

Encapsulation and Iterators

7/7/22, Thursday

Recap: Stacks

- A linear ADT that imposes a Last In, First Out (LIFO) order on elements
  - The last element inserted must be the first one to remove
  - Real life examples: a stack of books, a stack of dishes, web browser's "back" history, "undo" operation in a text editor
- A stack ADT has two ends: top and bottom
  - New elements can only be inserted at top
  - Only the element at the top may be removed
- Two main operations:
  - *Push* – inserts an element on the top
  - *Pop* – removes the top element

Recap: Queues

- A linear ADT that imposes a First In, First Out (FIFO) order on elements
    - The first element to be removed is the first one that was placed into it
    - Real life examples: a line of people waiting for check out
- A Queue ADT has two ends: front and back
    - Inserting elements to the back
    - Removing elements from the front
- Two main operations:
    - *Enqueue* – insert an element at the back
   - *Dequeue* – remove an element at the front

Implement Queue using Dynamic Array

- An array that allows data to wrap around from the back to the front is known as a circular buffer

- Q: How do we know which index corresponds to the back of the queue?
  - By computing a mapping between the array's *logical indices* and its *physical indices*
- Logical indices – the indices relative to the start of the data
- Physical indices – the indices relative to the start of the physical array

Implement Queue using Dynamic Array

- Mapping formula: `physical = start + logical;`

- Since it is circular, add the following to check:

```
if (physical >= capacity) {
    physical -= capacity;
}
```


start = 2   size = 3
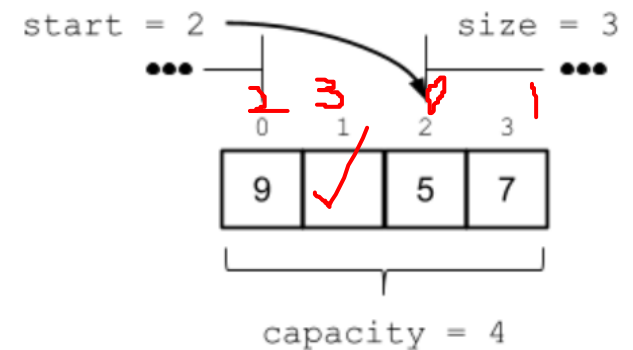
capacity = 4

- OR: **physical = (start + logical) % capacity;**

phy. of 9 = (2 + 2) % 4 = 0

- Index at which the next element will be inserted:
  - Previously: array[size] – when the data starts at physical index 0
 - Now: array[physical] – where physical = (start + size) % capacity

(2 + 3) % 4 = 1

5

Implement Queue using Dynamic Array

- Dynamic Array resizing for the queue implementation
- When do we need to resize?
  - size >= capacity
- When resize, reindex!
  - Logical index 0 ←→ Physical index 0

- How?
  - Loop through the logical indices from 0 to size – 1
  - Copy elements at each logical index in the old array to the equivalent physical index in the new array

Implement Queue using Dynamic Array

- Visually, look like this:
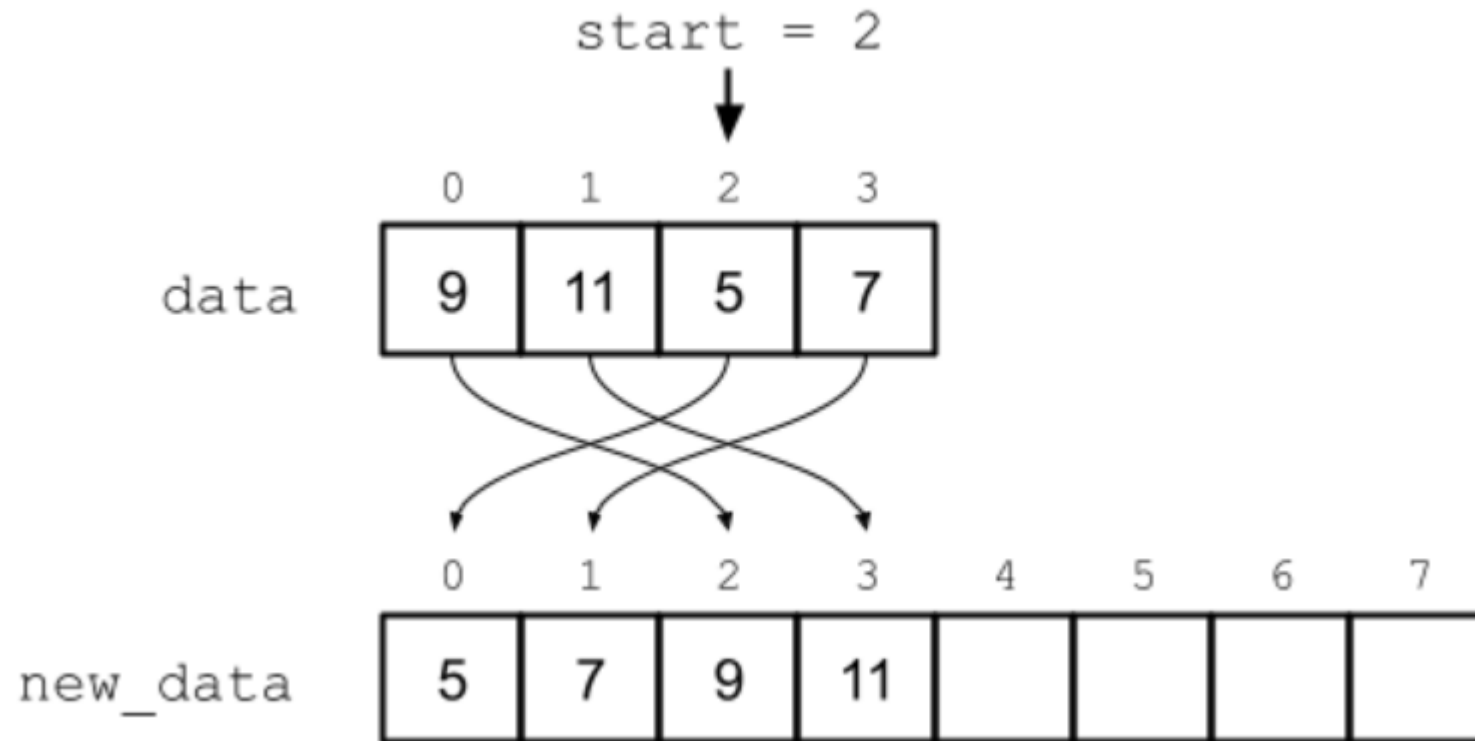
Implement Queue using Dynamic Array

- Complexity:
  - Dequeue – O(1) for all best-case, worst-case, and average case

  - Enqueue
    - O(1) for best-case and average case
   - O(n) for worst-case, when resize is needed

Deques

- A deque (double-ended queue) is a linear ADT that supports insertion and removal at both ends

- Examples: multi-processor job scheduling

- Four primary operations:
    - Add to front
    - Add to back
    - Remove from front
    - Remove from back

*Implement Deque using Dynamic Array

- Very similar to dynamic array-based queue implementation
  - Using circular buffer

- Not covered in this class

- FYI: https://www.geeksforgeeks.org/implementation-deque-using-circular-array/
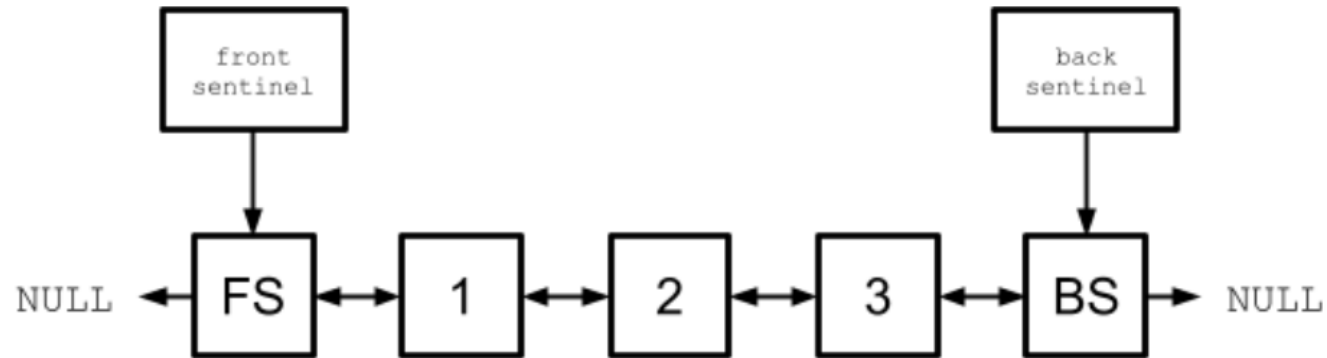
Implement Deque using Linked List

- Since a deque supports removal from both front and back, we need to use a doubly linked list
  - Allows to remove from the back and find the new back

- Use front and back sentinel in the list
  - Sentinel: a special node that is never removed from the list (doesn't store a value)

front
sentinel

back
sentinel

NULL ← FS ↔ BS → NULL
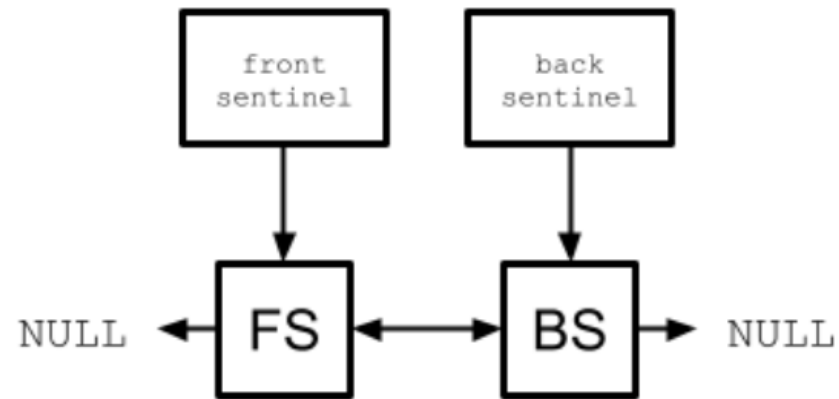
nodes

node* prev
void* val
node* next

Implement Deque using Linked List

- Values are inserted into the list in nodes that live between the sentinels. For example:

- Add front: insert a new node after the front sentinel
- Add back: insert a new node before the back sentinel
- Remove front: remove the node after the front sentinel
- Remove back: remove the node before the back sentinel

Implement Deque using Linked List

- Why do we use sentinels?

  - w/o sentinels, each operation would have to implemented differently, i.e.:
    - Add to the front w/o sentinels → update the head pointer upon each insertion
    - Add to the back w/o sentinels → update the tail pointer upon each insertion

  - w/ sentinels, both insertions (add to front and add to back) can use the exact same mechanics
    - So can both of the removal operations

Implement Deque using Linked List

- add_before() – insert a new node with a given value before a specified node already in the list, i.e.:

```
void add_before(void* value, struct node* next) {

    struct node* new_node = malloc(sizeof(struct node));

    new_node->value = value;

    new_node->prev = next->prev;

    next->prev->next = new_node;

    new_node->next = next;

    next->prev = new_node;

}
```
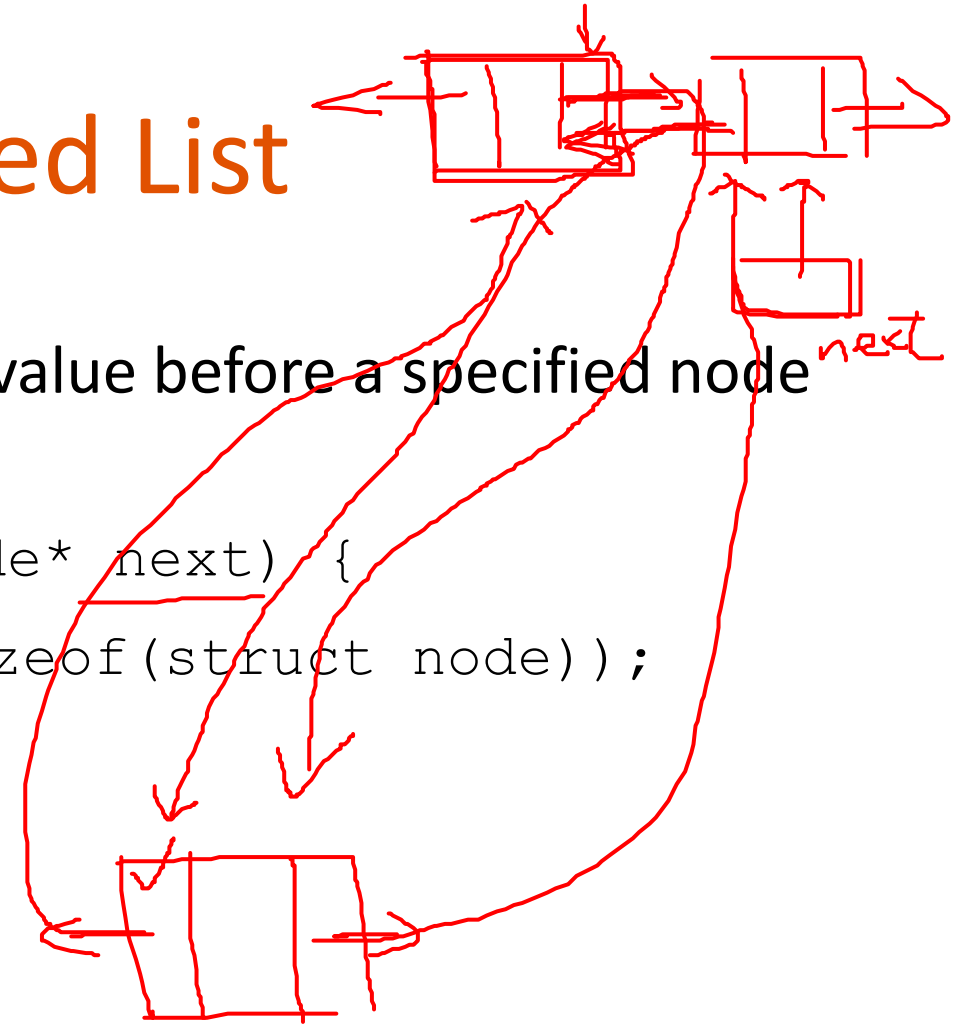
Implement Deque using Linked List

- Since our list uses sentinels, then our add_to_front() becomes:

```
void add_to_front(void* value) {
    add_before(value, front_sentinel->next);
}
```

- Our add_to_back() becomes:

```
void add_to_back(void* value) {
    add_before(value, back_sentinel);
}
```

Implement Deque using Linked List

- Similarly, assuming our list has a `remove_node()` function, then our remove_front() becomes:

```
void remove_front() {
    remove_node(front_sentinel->next);
}
```

- Our remove_back() becomes:

```
void remove_back() {
    remove_node(back_sentinel->prev);
}
```

- To check if the list is empty:

```
if (front_sentinel->next == back_sentinel)
```

Implement Deque using Linked List

- Complexity:
  - Add to front – O(1)
  - Add to back – O(1)
  - Remove front – O(1)
  - Remove back – O(1)

*For all best case, worst case, and average case

Lecture Topics:

- Stack, Queue, and Deque
- Encapsulation and Iterators

Encapsulation

- Encapsulation – hide the internal details of a data type from the user of that data type, instead exposing only a simplified interface through which the user interacts with the data type
  - User – another developer who will be using the code we've written

- For example, linked list implementation has hidden the details of the list implementation behind a simplified interface.
  - Only the name of linked list data type was exposed to the user (i.e., `struct list`)
  - If the user tried to access internal fields (`list->head`) → error
    - "dereferencing a pointer of incomplete type"

Why Encapsulation?

- Reduces the cognitive overhead to understand
- Cannot misuse (and possibly break) the data type
  - Cannot set `list->head` to `NULL` (could cause a memory leak)
- Easier to implement the data type
  - Avoid tedious error checking

- Potential challenges:
  - What if our user wants to *iterate* through each element in the collection within a loop?
    - Problem: cannot access the internals, i.e., for linked list, cannot access the `head`

Iterator

- Iterator – a data type acts as a companion to a collection and provides a mechanism to iterate through that collection
  - Implemented to have access to the internals of the collection

- Each specific kind of collection will have its own iterator data type

- Two common functions:
  - next () – returns the current value, and moves the iterator to the next element
  - has_next () – returns true or false to indicate whether or not there is another element afterwards

To use an Iterator

- Assuming we have an iterator `iter` over a collection:

```
while (has_next(iter)) {
   value = next(iter);
   ... /* Do something with value. */
}
```

Linked list Iterator

- Implement an iterator for a linked list:
  - In C: defined within the same file
  - In C++: using nested classes or friend

- Our linked list iterator must have access to the internals of the linked list:

```
struct node {
  void* value;
  struct node* next;
};

struct list {
  struct node* head;
};
```

Linked list Iterator

- 1. define a structure to represent the list iterator
  - How to iterate? Using a pointer (i.e., `curr`) to represent the current node
  - Initially points to the head, and moves to the next (i.e., `curr = curr -> next;`)

```
struct list_iterator {
    struct node* curr;
};
```

- 2. implement a function to create a new iterator and associate it with a list to iterate:

```
struct list_iterator* list_iterator_create(struct list* list) {
    struct list_iterator* iter = malloc(sizeof(struct list_iterator));
    iter->curr = list->head;
    return iter;
}
```

Linked list Iterator

- 3. Implement has_next()

```
int has_next(struct list_iterator* iter) {
  return iter->curr != NULL;
}
```

- 4. Implement next()

```
void* next(struct list_iterator* iter) {
  void* value = iter->curr->value;
  iter->curr = iter->curr->next;
  return value;
}
```

- *5. Polish (i.e., add error checking)