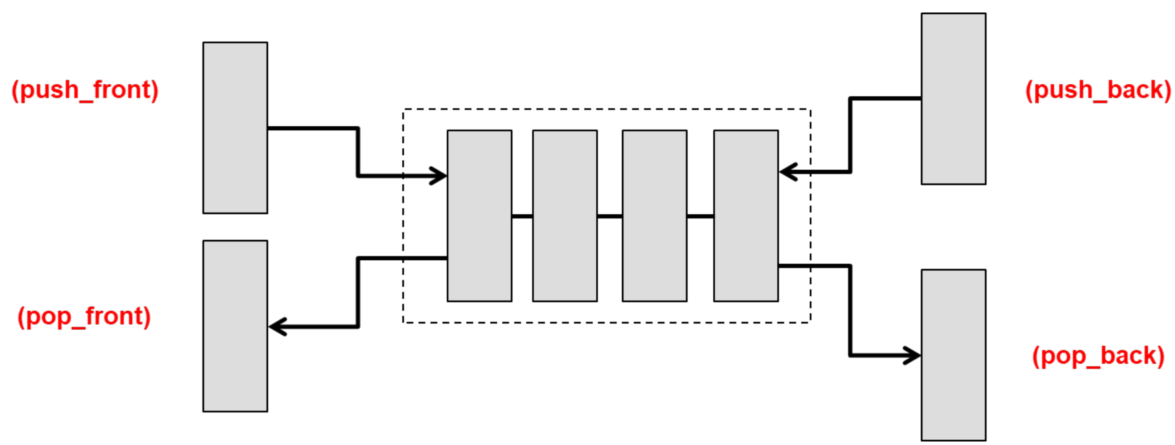
CS 103 Unit 4c – Deque Implementation using Doubly-Linked Lists

CSCI 103L Teaching Team

Deques

- **D**ouble-**e**nded **que**ues (like their name sounds) are the name given to any data structure that allows fast (i.e. O(1)) insertion and removal from **BOTH SIDES (front and back)** of the list
  - `push_front(): O( 1 )` `push_back(): O( 1 )`
  - `pop_front(): O( 1 )` `pop_back(): O( 1 )`
- There are several possible implementations of such a data structure
  - We will first explore a linked-list implementation for a deque
  - C++ also provides an array-based deque implementation



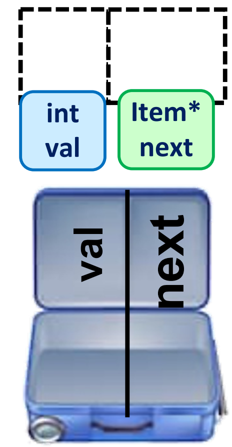
DOUBLY-LINKED LISTS

Singly-Linked List Review

- Use structures/classes and pointers to make **linked** data structures
- Singly-linked Lists **dynamically allocates** each item when the user decides to add it
- Each item includes a **next** pointer to the following item
- **Traversal and iteration is only easily achieved in one direction (i.e. the forward direction)**

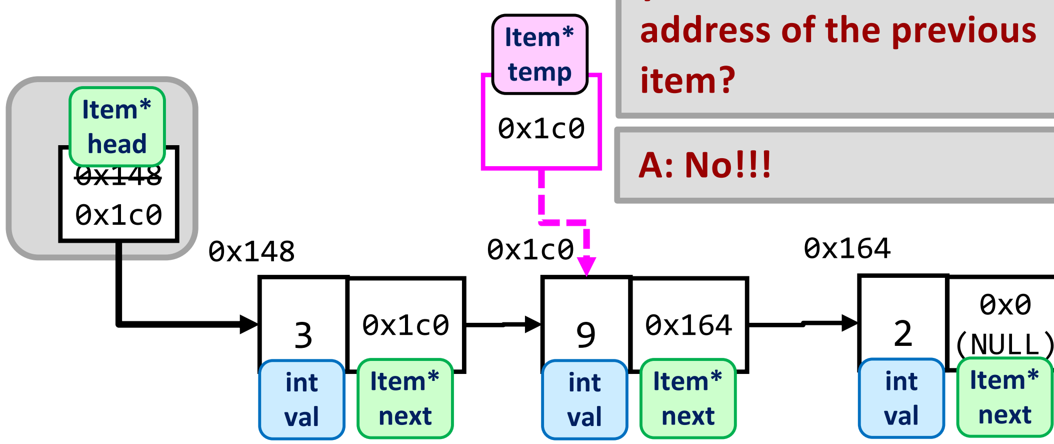
```
struct Item {
  int val;
  Item* next;
};

class List {
public:
  List();
  ~List();
  void push_back(int v); ...
private:
  Item* head;
};
```

struct Item blueprint:



Q: Given temp…could you ever recover the address of the previous item?

A: No!!!

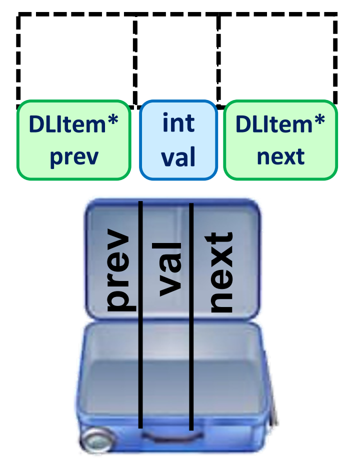


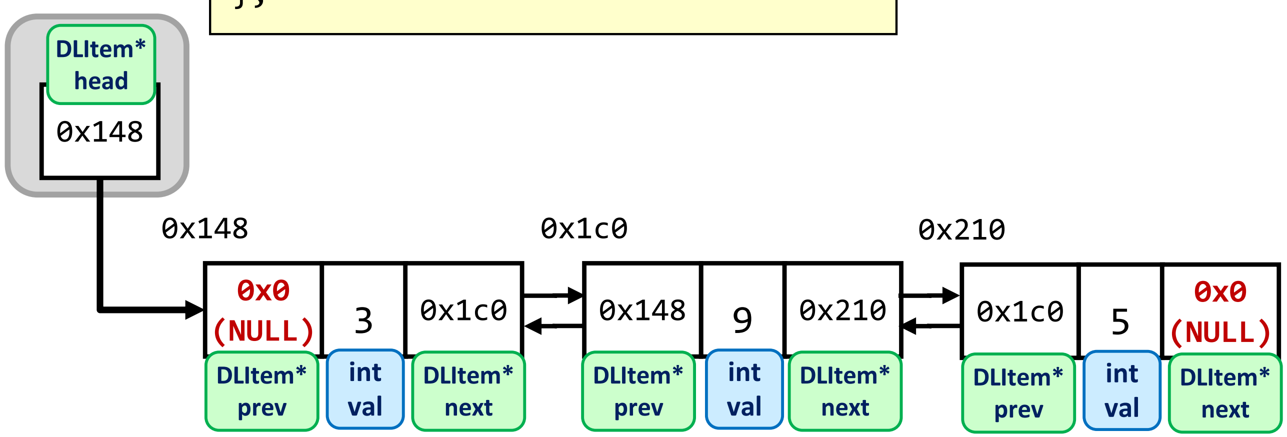
Doubly-Linked Lists

- Includes a previous pointer in each item so that we can traverse/iterate backwards or forward
- First item's previous field should be NULL
- Last item's next field should be NULL

```
#include<iostream>

using namespace std;
struct DLItem {
  int val;
  DLItem* prev;
  DLItem* next;
};

class DLList
{
  public:
   DLList();
   ~DLList();
   void push_back(int v); ...
  private:
   DLItem* head;
};
```

struct DLItem blueprint:

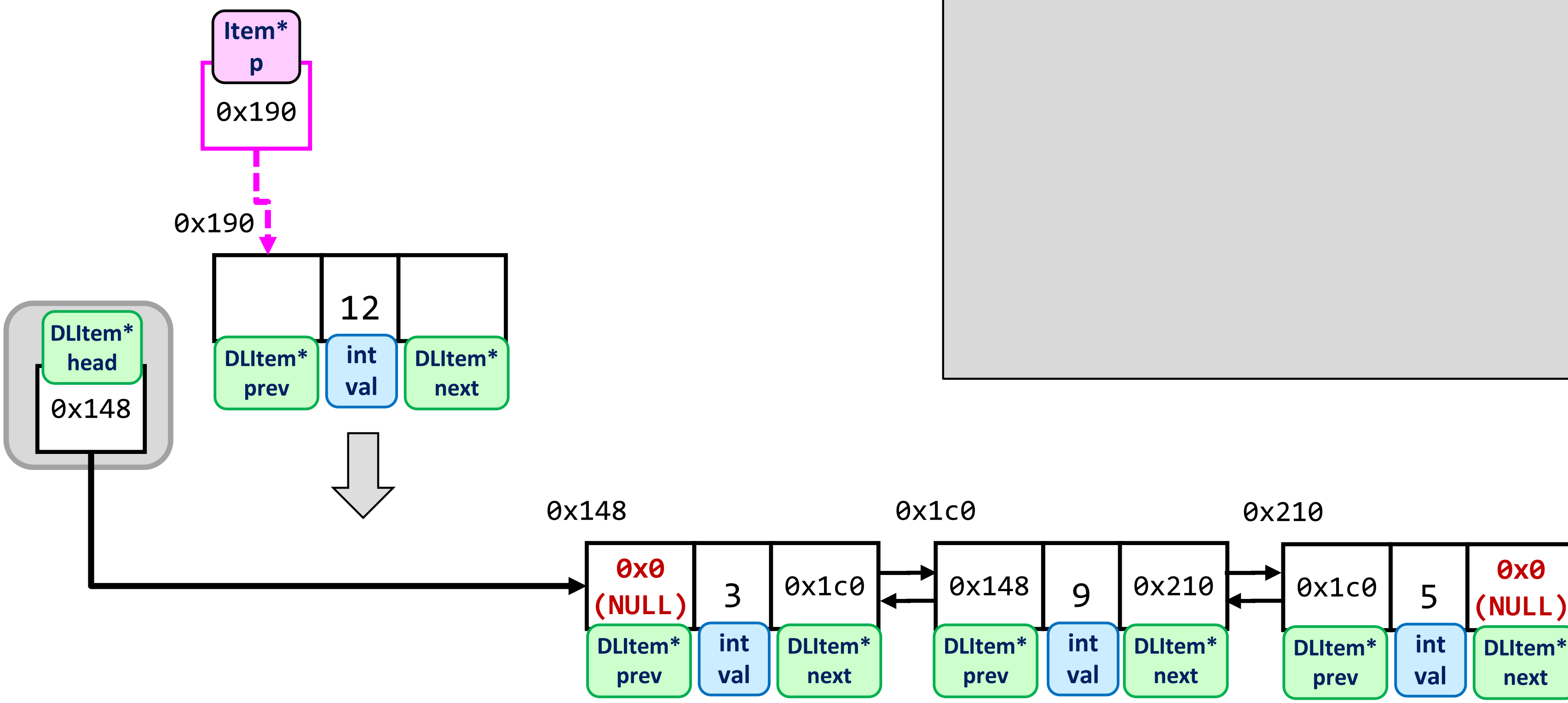




Doubly-Linked List Add Front

- Adding to the front requires you to update which pointers?
  - Take care of the ORDER in which you update the pointers not to lose information you need
  - Tip: Fill in the blank fields/pointers first before changing existing ones.

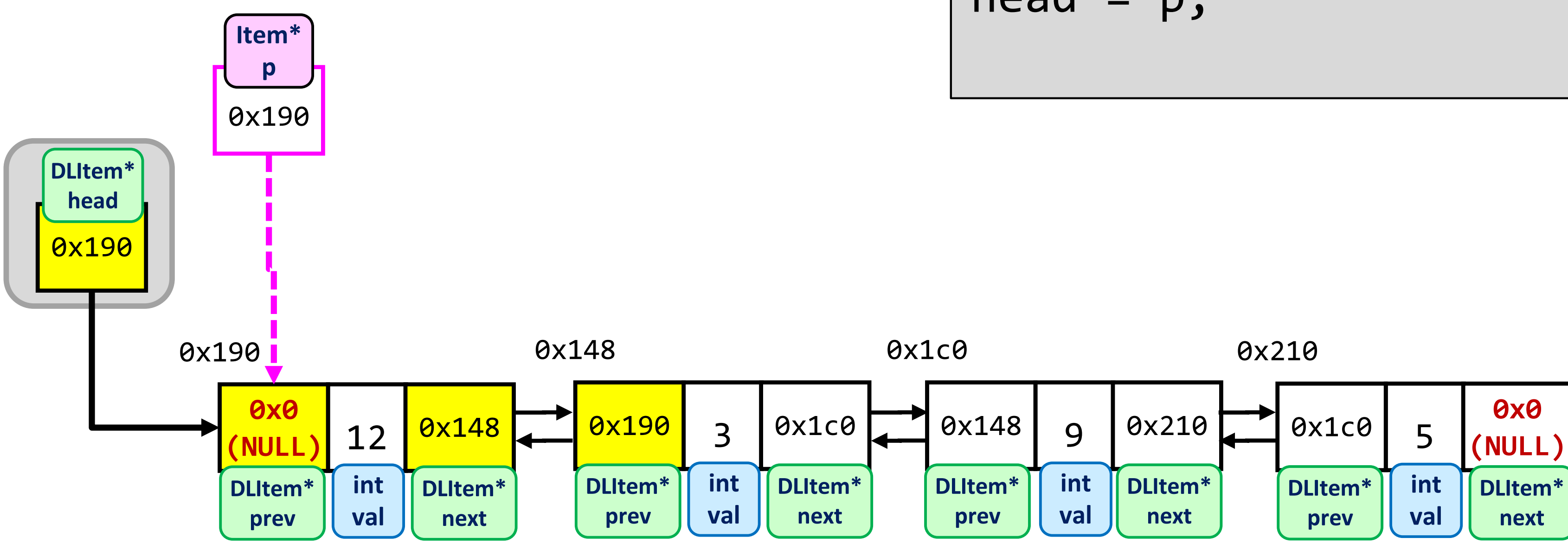
```
head = p; // ??
```



Doubly-Linked List Add Front

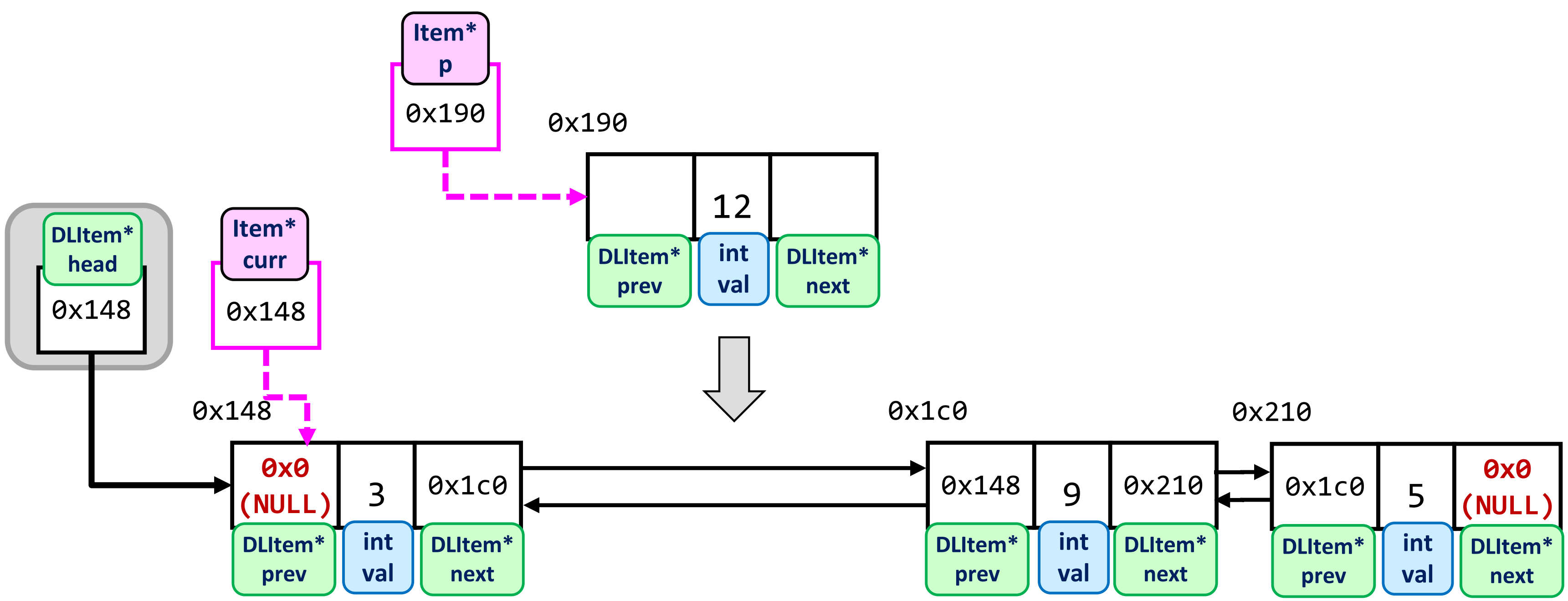
- Adding to the front requires you to update which pointers?
  - Head
  - New front's next & previous
  - Old front's previous

```
p->prev = NULL;
p->next = head;
head->prev = p;
  // or p->next->prev = p;
head = p;
```



Doubly-Linked List Add Middle

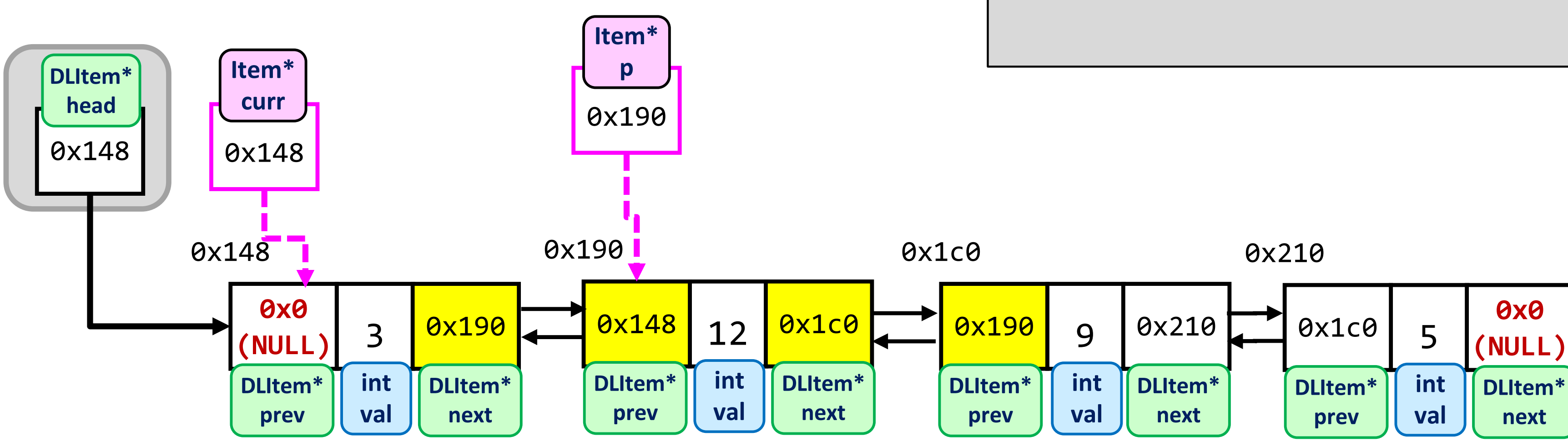
- Adding to the middle requires you to update which pointers?



Doubly-Linked List Add Middle

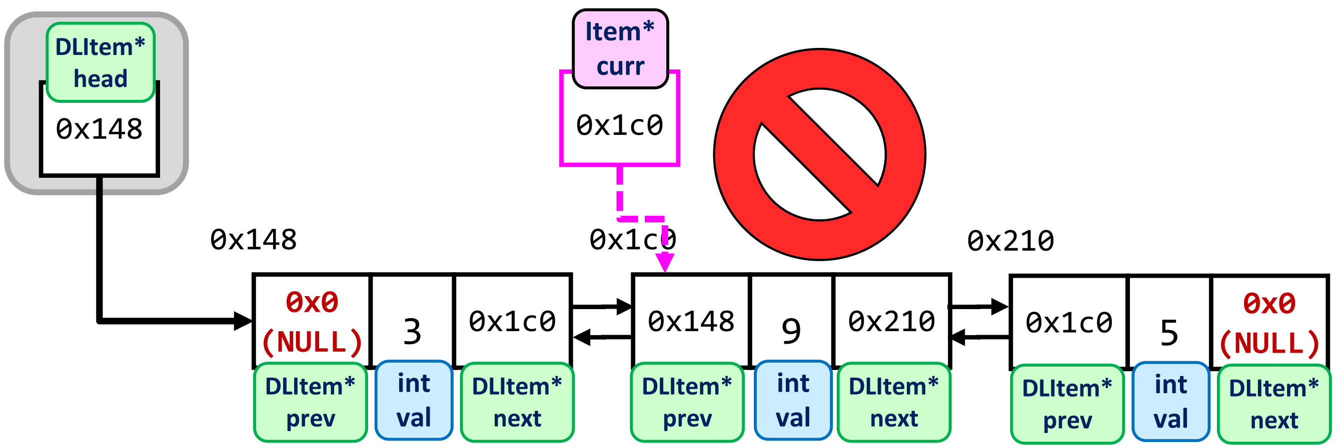
- Adding to the middle requires you to update which pointers?
  - Previous item's next field
  - Next item's previous field
  - New item's next field
  - New item's previous field

```
p->prev = curr;
p->next = curr->next;
curr->next->prev = p;
  // or p->next->prev = p;
curr->next = p;
  // or p->prev->next = p;
```



Doubly-Linked List Remove Middle

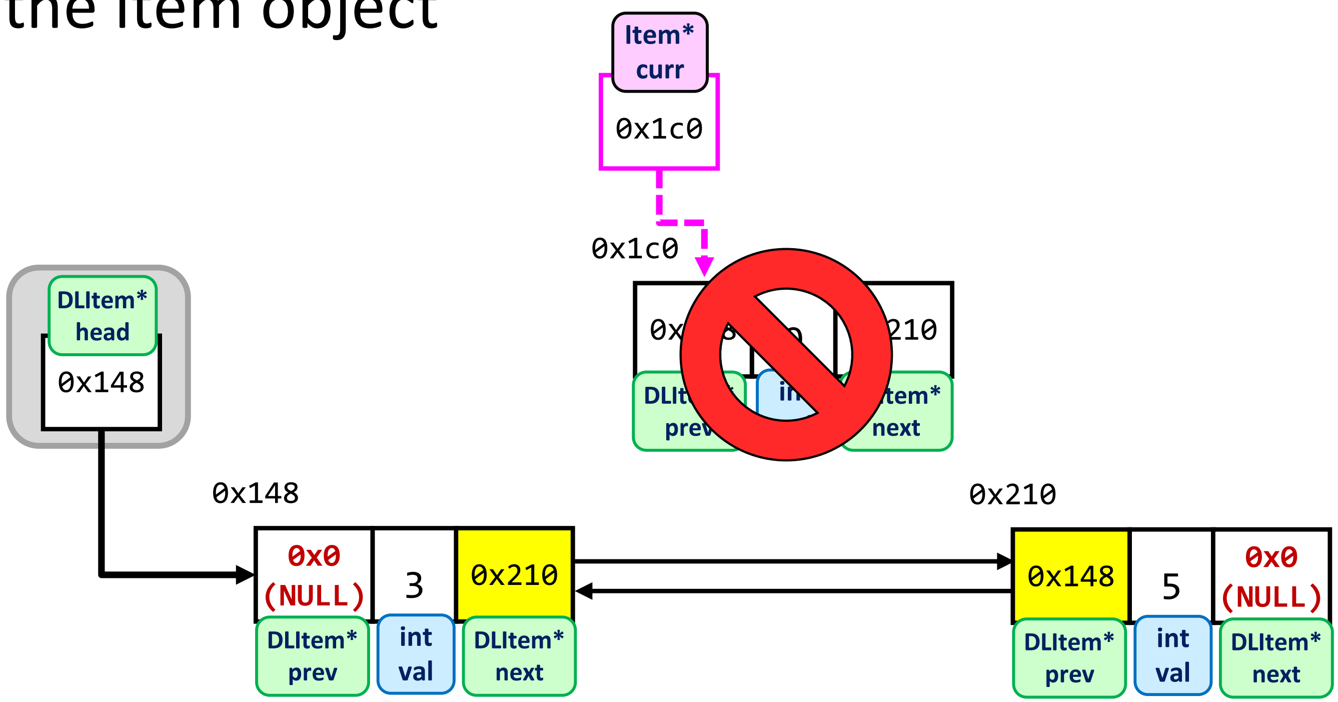
- Removing from the middle requires you to update which pointers?



Doubly-Linked List Remove Middle

- Removing from the middle requires you to update which pointers?
  - Previous item's next field
  - Next item's previous field
  - Delete the item object

```
curr->next->prev = curr->prev;
curr->prev->next = curr->next;
```



Using a Doubly-Linked List to Implement a Deque

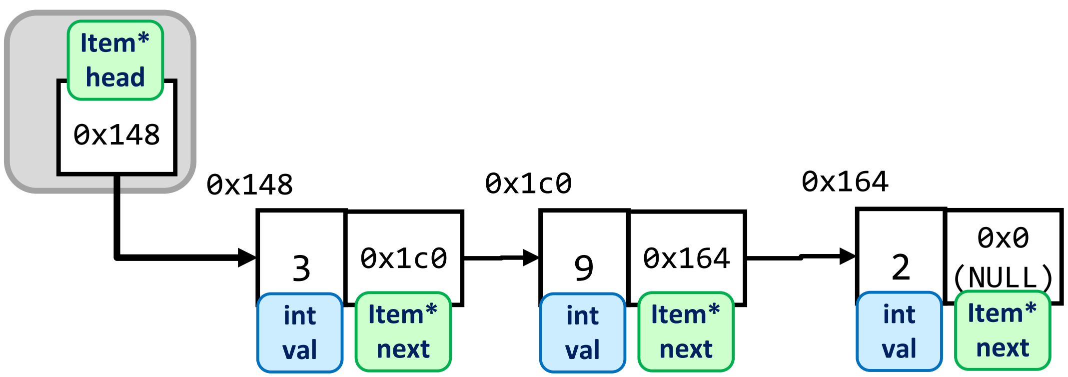
DEQUES AND THEIR IMPLEMENTATION

Deque Implementation

- Let's consider how we can implement a deque
- **Question for exploration**:
  – Could we use a singly-linked list and still get fast [i.e. O(1)] insertion/removal from both front and back?
  – For various implementations, analyze the runtime for each operation:
    - `push_front(), pop_front(), push_back(), pop_back()`

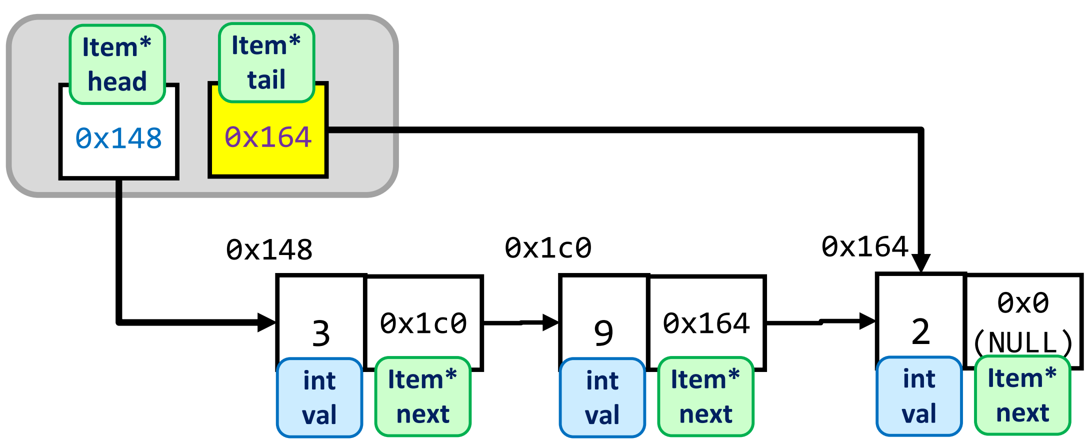
Singly-Linked List Deque?

- With our singly-linked list, which operation are O(1) vs. O(n)
  - push_front(): O( )
  - pop_front(): O( )
  - push_back(): O( )
  - pop_back(): O( )



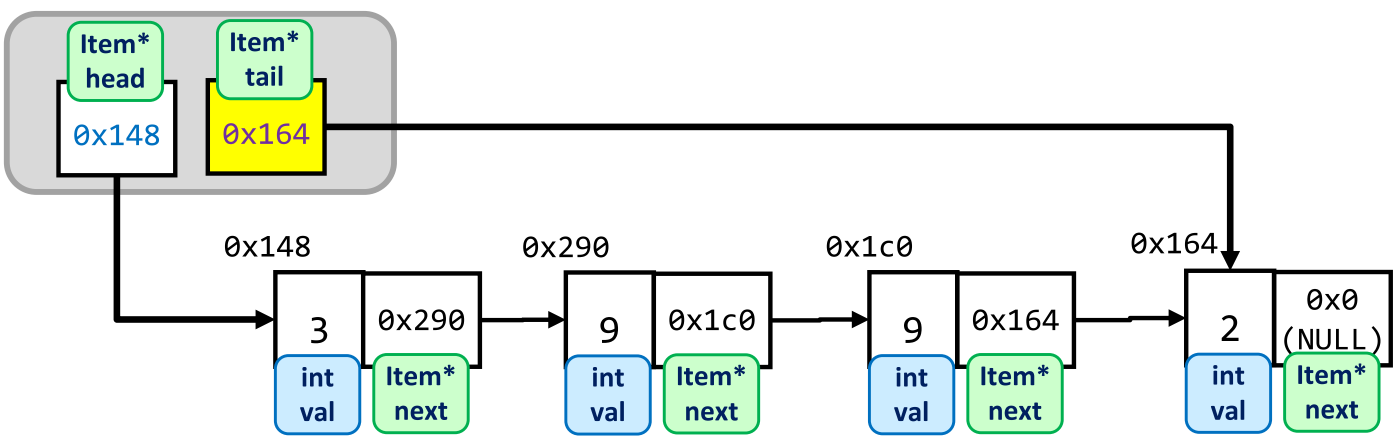
Singly-Linked List With Tail Pointer

- Suppose we keep a second pointer (aka a tail pointer) to always point at the LAST element.
  - head points at the FIRST element; `tail` points at the LAST element
  - Which operations are O(1) vs. O(n)?
  - `push_front(): O( 1 )    pop_front(): O( 1 )`
  - `push_back(): O(   )`



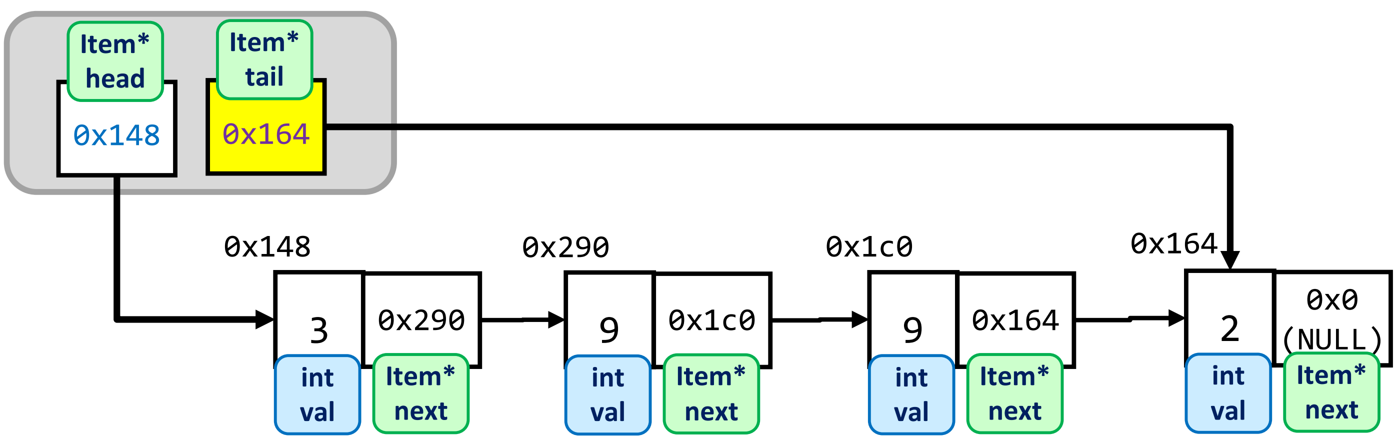
Singly-Linked List With Tail Pointer

- Suppose we keep a second pointer (aka a tail pointer) to always point at the LAST element.
  - head points at the FIRST element; `tail` points at the LAST element
  - Which operations are O(1) vs. O(n)?
  - `push_front(): O( 1 )` `pop_front(): O( 1 )`
  - `push_back(): O( 1 )` `pop_back(): O( )`



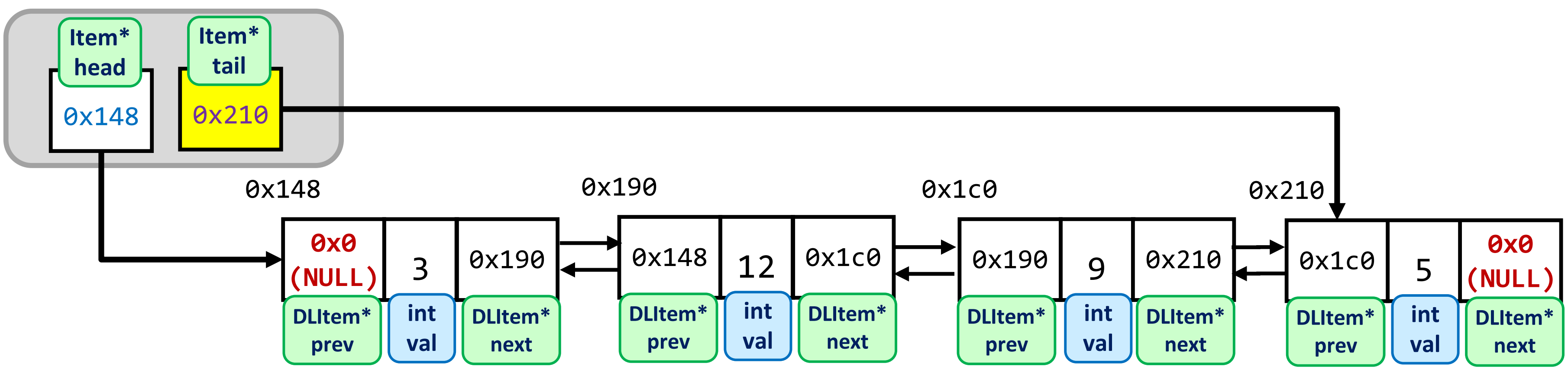
Singly-Linked List With Tail Pointer

- Singly Linked Lists w/ Tail Pointer:
  - `push_front(): O( 1 )`   `pop_front(): O( 1 )`
  - `push_back(): O( 1 )`   `pop_back(): O( n )`
- So even with a tail pointer a singly-linked list **CANNOT remove from the back in O(1)**!
  - We have to update the 2nd to last item, and we can't find that quickly (i.e. without walking the list).



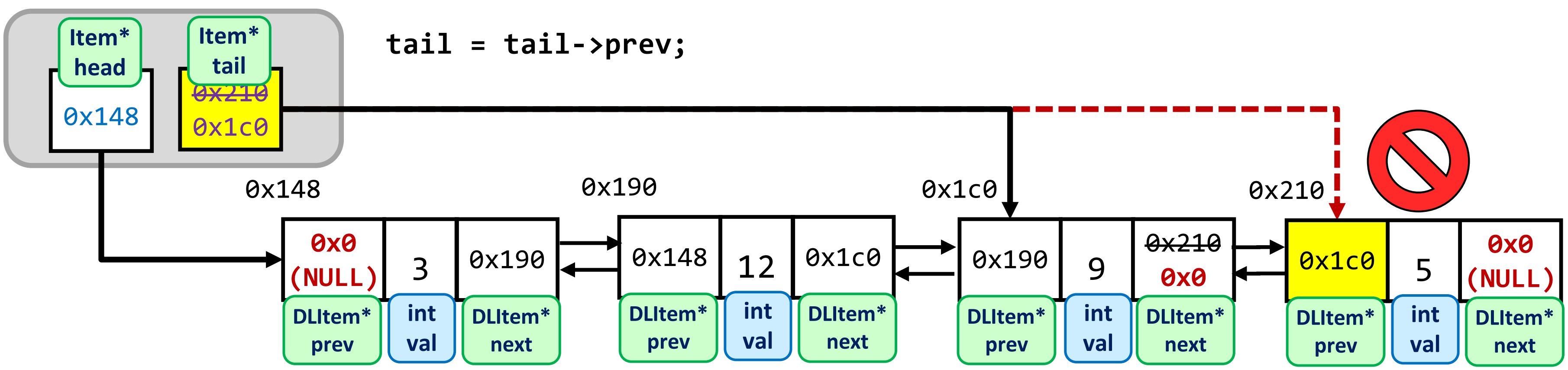
Doubly-Linked List With Tail Pointer

- Since removal from the back needs to move the tail BACK 1 step to the previous item, let's use a doubly-linked list!
  - `push_front(): O( 1 )`   `pop_front(): O( 1 )`
  - `push_back():  O( 1 )`   `pop_back():  O(   )`



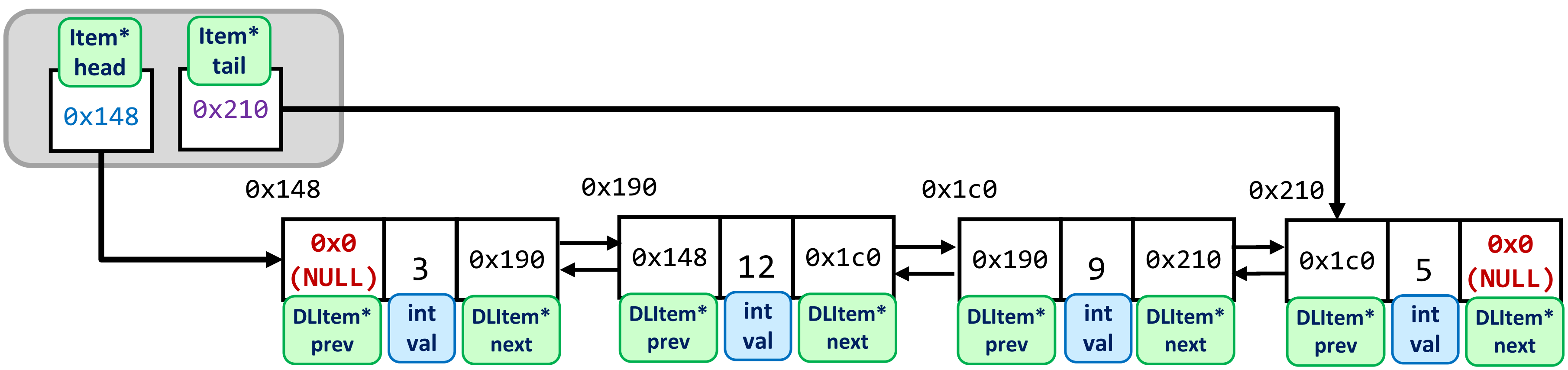
Doubly-Linked List With Tail Pointer

- Since removal from the back needs to move the tail BACK 1 step to the previous item, let's use a doubly-linked list!
  - push_front(): O(1)   pop_front(): O(1)
  - push_back(): O(1)   pop_back(): O(1)
- We can update the tail pointer with the tail item's previous pointer before we delete the tail item



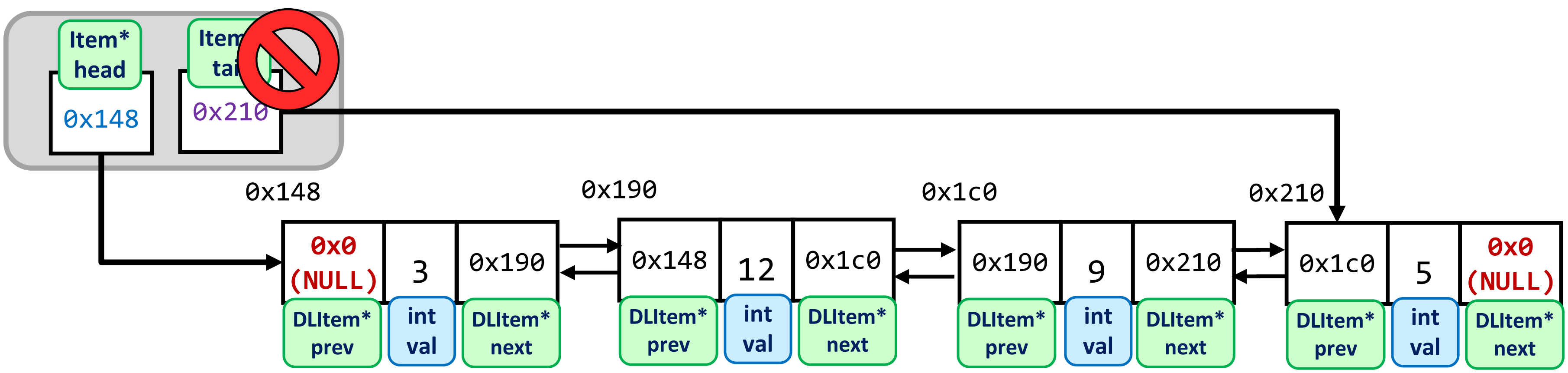
Success

- We've done it! We can insert and remove from the front or back quickly (in O(1))!
  - `push_front(): O( 1 )    pop_front():  O( 1 )`
  - `push_back():  O( 1 )    pop_back():   O( 1 )`
- So a singly-linked list CANNOT serve as a deque!
- But a double-linked list with tail pointer CAN!



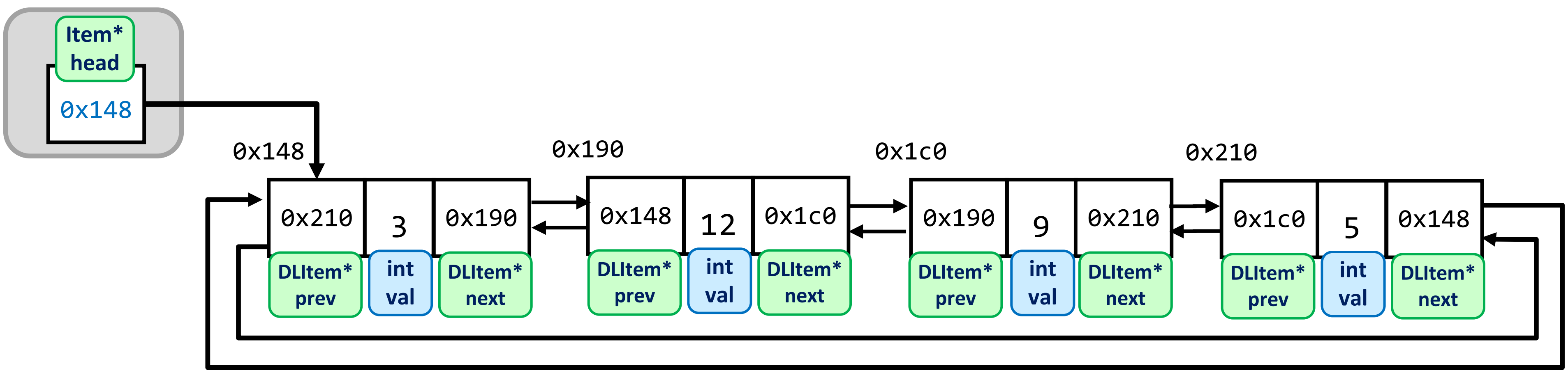
A Thought Experiment

- Do we really need the tail pointer to achieve O(1) insert and removal from the end?
  - Seemingly yes. How else would we know where the end is?
- Is there a way we could structure our linked list to make finding the last item easy and fast but by only using a head pointer?



Circular Linked List

- Though there's no great benefit over just using a tail pointer, we can implement a deque as a doubly-linked list with only a head pointer by making the first and last item point at each other
  - This effectively creates a **circular linked list**.
- What expression would yield a pointer to the tail item?
  - ________________
- It saves a pointer, but is harder to code/implement
  - Let's just stick with the tail pointer implementation ☺



C++ DEQUE CLASS

C++ Deque Class

- **Pros**: Similar to vector but allows for efficient (O(1)) `push_front()` and `pop_front()` options
- **Cons**: Slightly slower random access (i.e. accessing the i-th element / `data[i]`)
- Useful when we want to put things in one end of the list and take them out of the other

1 my_deq — after 1st iteration

0	1
0	50

2 my_deq — after all iterations

0	1	2	3	4	5
2	1	0	50	51	52

3 my_deq

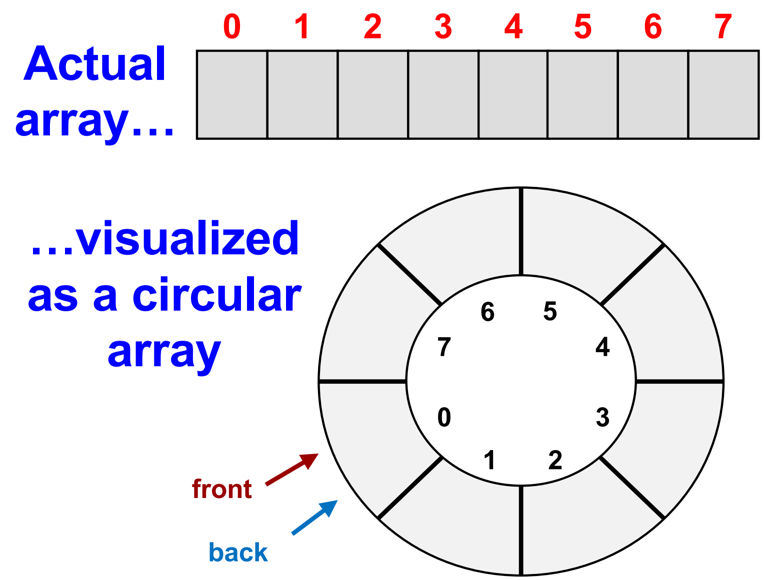
0	1	2	3	4	5
12	11	10	60	61	62

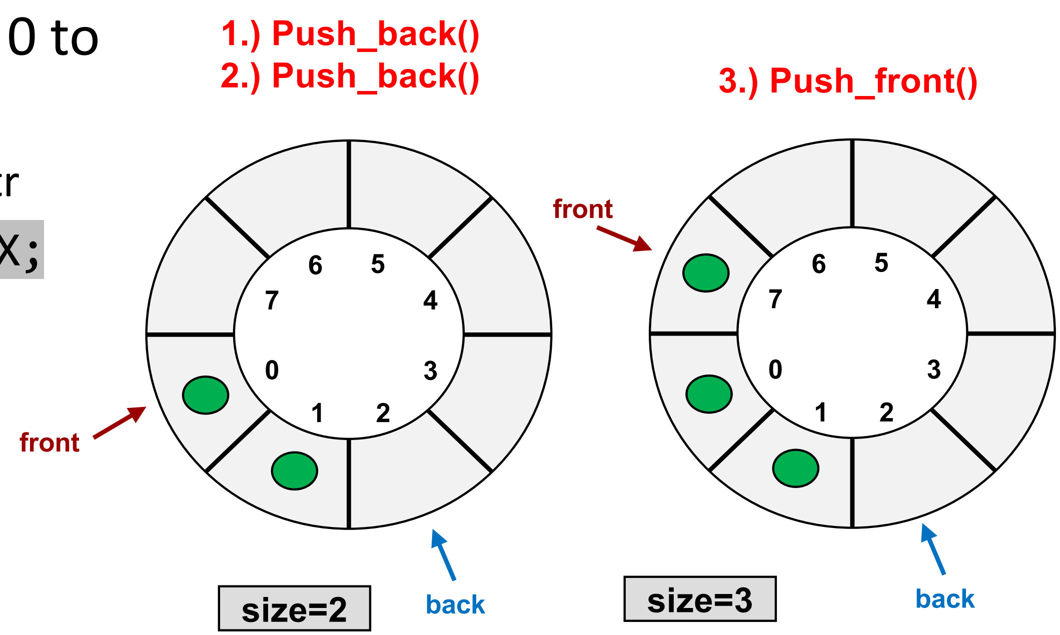
4 my_deq

```
  #include <iostream>
  #include <deque>
  using namespace std;
  int main()
  {
    deque<int> mydeq;
    for(int i=0; i < 3; i++){
1     mydeq.push_back(i+50);
      mydeq.push_front(i);
2   }
    cout << "At loc. 2 is: "
         << my_deq[2] << endl;

    for(int i=0; i < mydeq.size(); i++){
      int x = my_deq.front();
      my_deq.push_back(x+10);
      my_deq.pop_front();
3   }
    while( ! my_deq.empty() ){
4     cout << my_deq.front() << " ";
      my_deq.pop_front();
    }
    cout << endl;
  }
```

Array-Based Deque Implementation

- Take an array but imagine it wrapping into a circle to implement a deque
- Setup a head and tail pointer
  - Head points at first occupied item, tail at first free location
  - Push_front() and pop_front() update the head pointer
  - Push_back() and pop_back() update the tail pointer
- To overcome discontinuity from index 0 to MAX-1, use modulo operation
  - Cannot just use `back++;` to move back ptr
  - Instead, use `back = (back + 1) % MAX;`
- Get item at index i
  - Must be relative to the `front` pointer





C++ Deque Class

- Performance:
  - Slightly slower at random access (i.e. array style indexing access such as: data[3]) than vector
  - Fast at adding or removing items at front or back

Summary

- Multiple possible implementations of Deque's exist
  - Array based
  - Doubly-linked list (recall a singly-linked list can't pop_back() in O(1))
- The implementation will determine the runtime of various operations:
  - Can a doubly-linked list support O(1) access to a random element (i.e. Can you access list[i] quickly for any i)?
  - No!!! Still need to traverse the list
- The only thing a deque guarantees is O(1) access to add/remove from either the front or back