University of Southern California

Viterbi School of Engineering

EE599 (to be EE595)
Software Design and Optimization

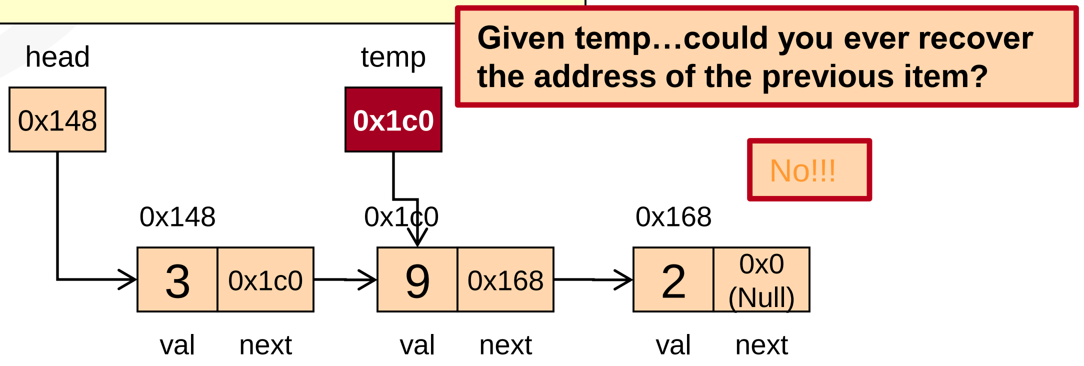
Data Structures – CLLs, DLLs

Singly Linked List Review

- Used 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 holding the address of the following Item object
- Traversal and iteration is only easily achieved in one direction

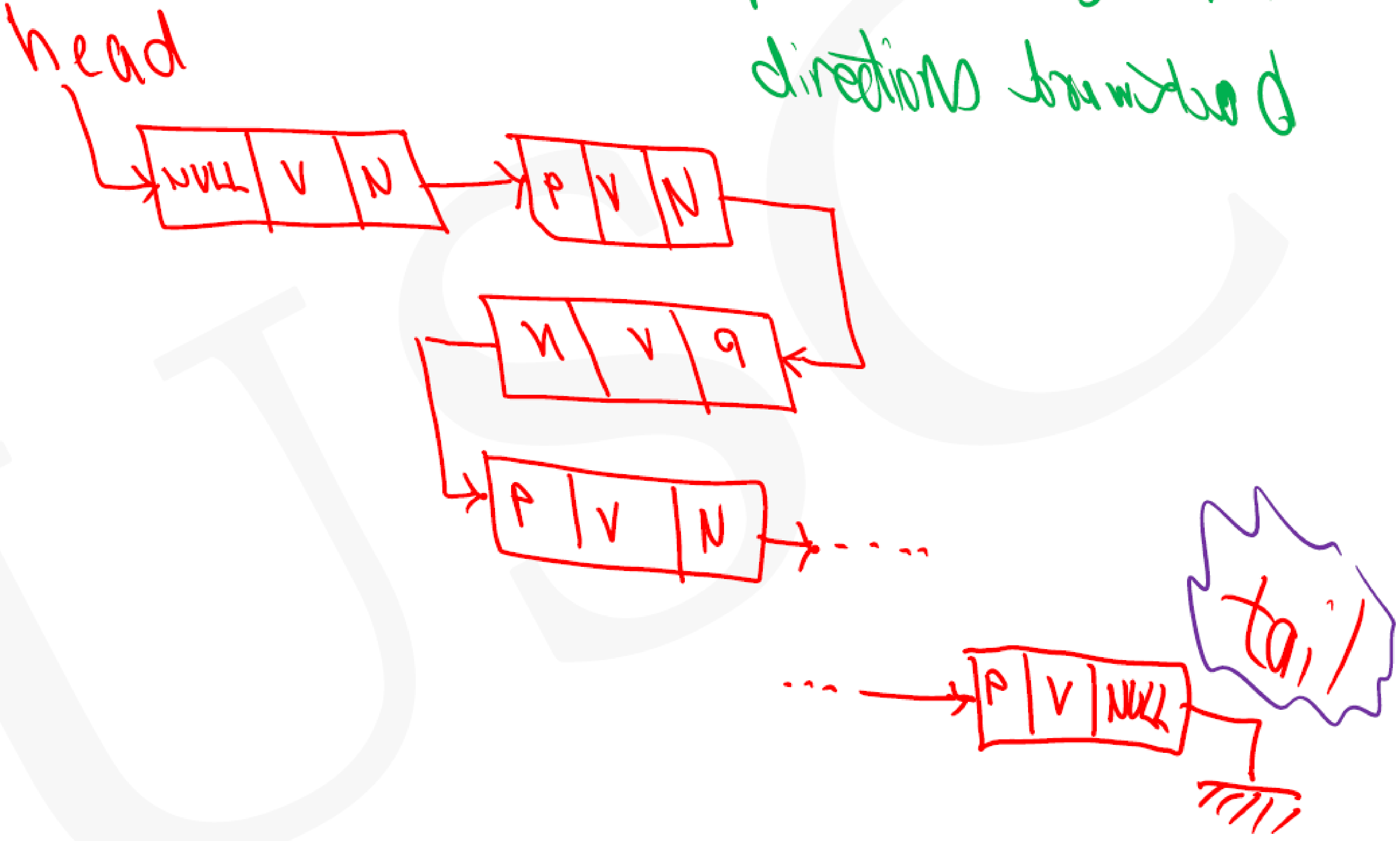
```
#include<iostream>

using namespace std;

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

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

struct Item blueprint:



Motivation

Traversal possible in forward &
directions backward



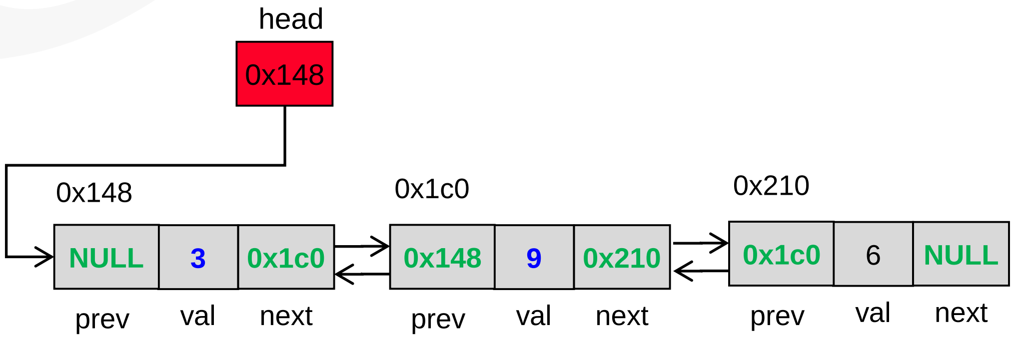
DLL

- 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;
};
```

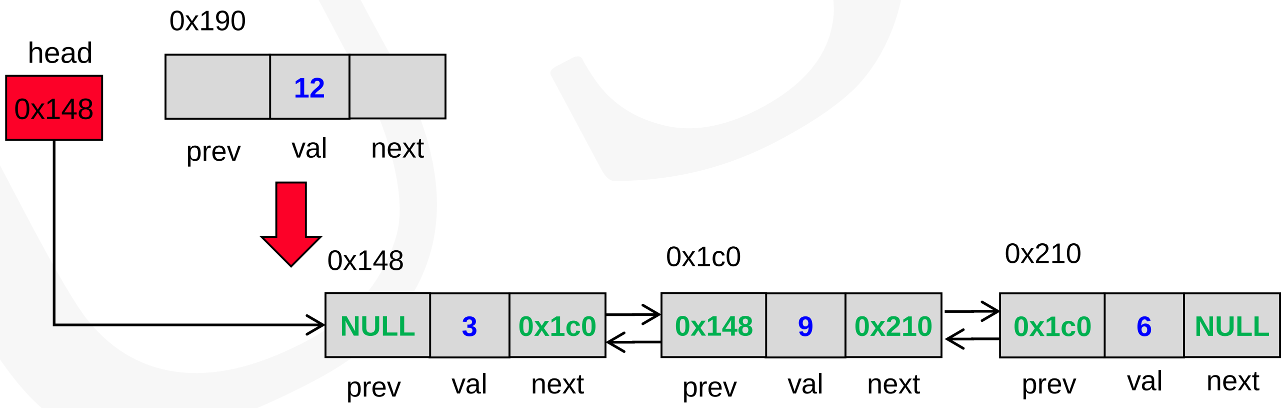
struct Item blueprint:

DLItem *	int	DLItem *
prev	val	next

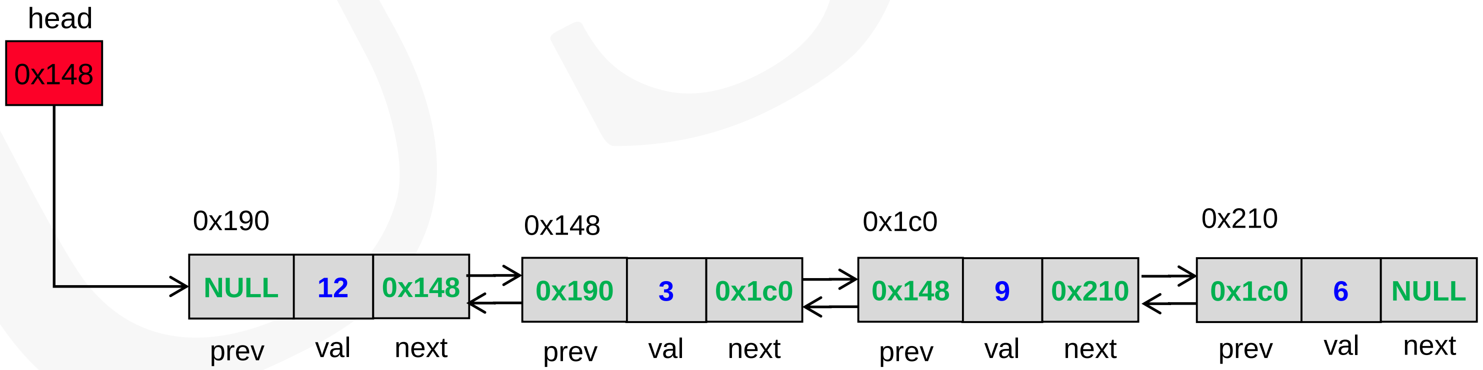


DLL – Add Front

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

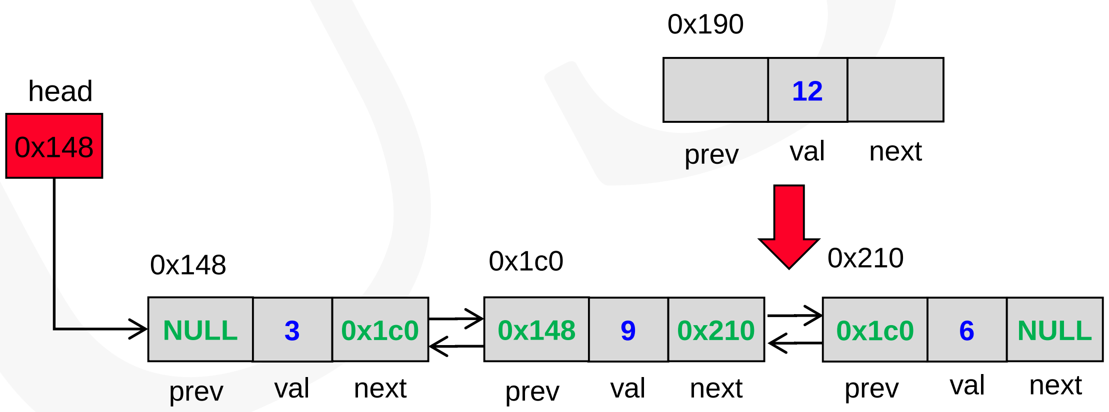


DDL – Add Front (cont.)

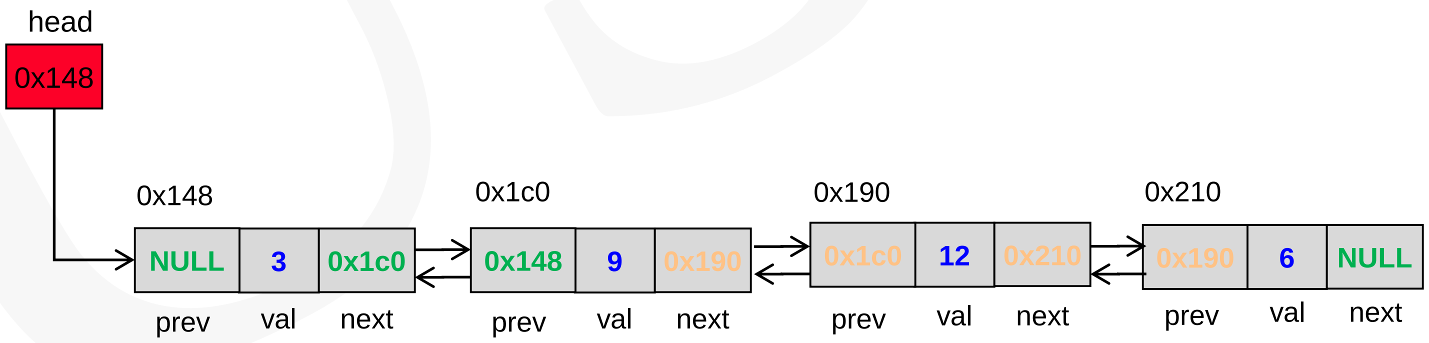


DLL – Add Middle

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

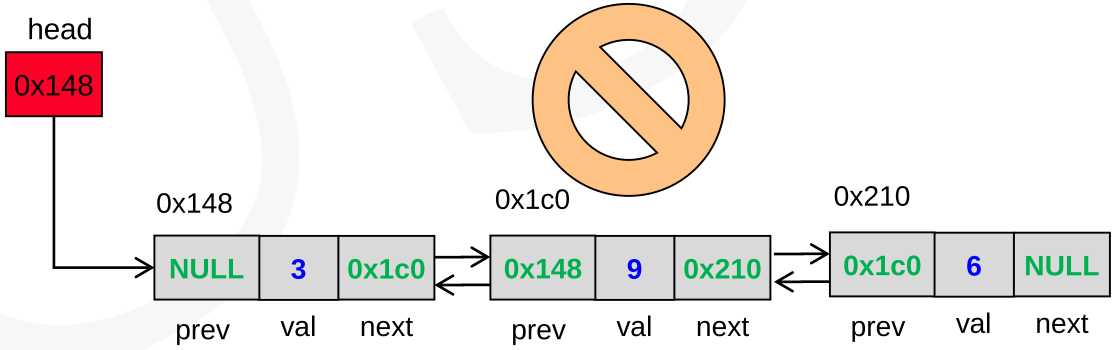


DLL – Add Middle (cont.)

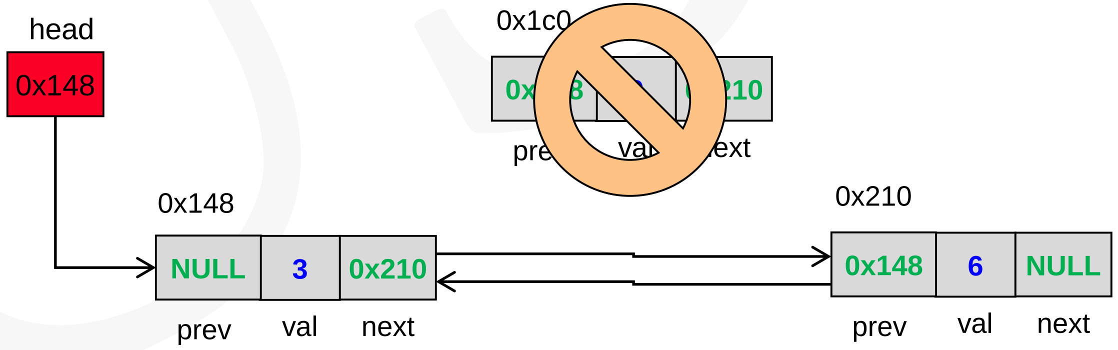


DLL – Remove Middle

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



DLL – Remove Middle (cont.)



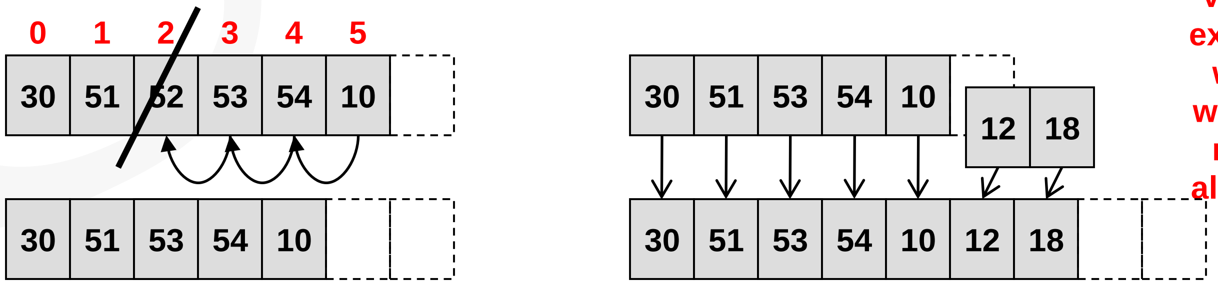
Using a Doubly-Linked List to Implement a Deque

DEQUES AND THEIR IMPLEMENTATION

Understanding Performance

- Recall vectors are good at some things and worse at others in terms of performance
- The Good:
  - Fast access for random access (i.e. indexed access such as myvec[6])
  - Allows for ‘fast’ addition or removal of items at the back of the vector
- The Bad:
  - Erasing / removing item at the front or in the middle (it will have to copy all items behind the removed item to the previous slot)
  - Adding too many items (vector allocates more memory that needed to be used for additional push_back()’s…but when you exceed that size it will be forced to allocate a whole new block of memory and copy over every item

After deleting we have to move everyone up



Vector may have 1 extra slot, but when we add 2 items a whole new block of memory must be allocated and items copied over

Deque Class

- Double-ended queues (like their name sounds) allow for efficient (fast) additions and removals from either 'end' (*front or back*) of the list/queue
- 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

Deque Class

- Similar to vector but allows for push_front() and pop_front() options
- Useful when we want to put things in one end of the list and take them out of the other

1 my_deq

0	1	2	3	4
50	51	52	53	54

2 my_deq

0	1	2	3	4
51	52	53	54	60

after 1st iteration

3 my_deq

0	1	2	3	4
60	61	62	63	64

after all iterations

4 my_deq

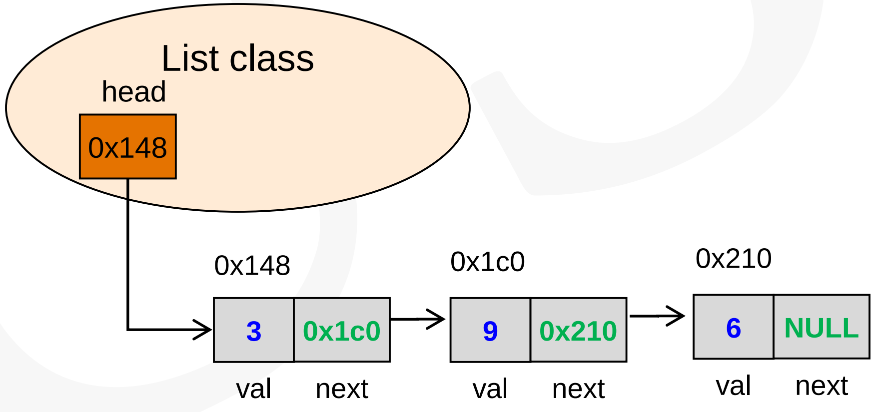
```
#include <iostream>
#include <deque>

using namespace std;

int main()
{
  deque<int> my_deq;
  for(int i=0; i < 5; i++){
    my_deq.push_back(i+50);
  }
1
  cout << “At index 2 is: ” << my_deq[2] ;
  cout << endl;

  for(int i=0; i < 5; i++){
2   int x = my_deq.front();
    my_deq.push_back(x+10);
3   my_deq.pop_front();
  }
  while( ! my_deq.empty()){
    cout << my_deq.front() << “ ”;
    my_deq.pop_front();
4 }
  cout << endl;

}
```

Deque Implementation

- Let's consider how we can implement a deque
- Could we use a singly-linked list and still get fast (i.e., O(1)) insertion/removal from both front and back?

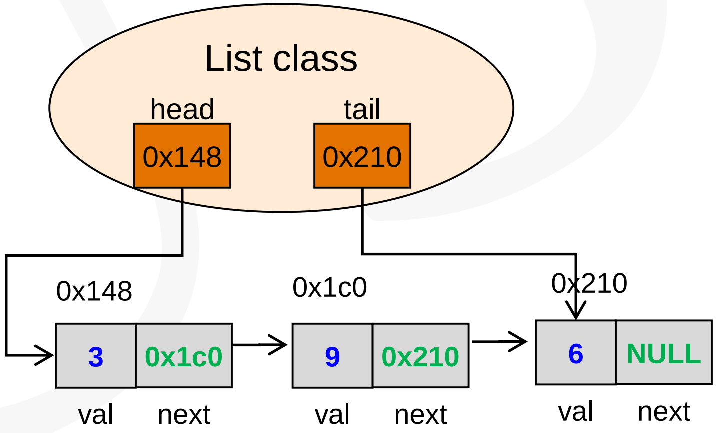
SLL Deque

- Recall a deque should allow for fast (i.e., O(1)) addition and removal from front or back
- In our current singly-linked list we only know where the front is and would have to traverse the list to find the end (tail)



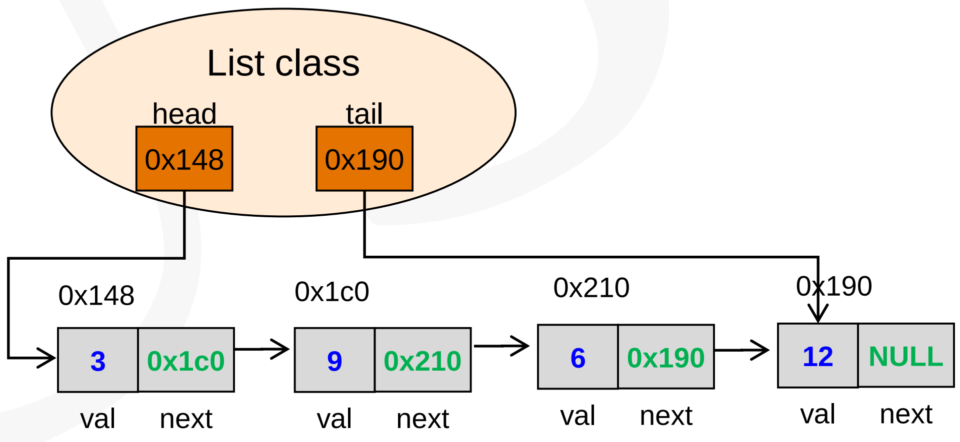
Option 1: SLL + Tail Pointer

- We might think of adding a tail pointer data member to our list class
  - How fast could we add an item to the end?



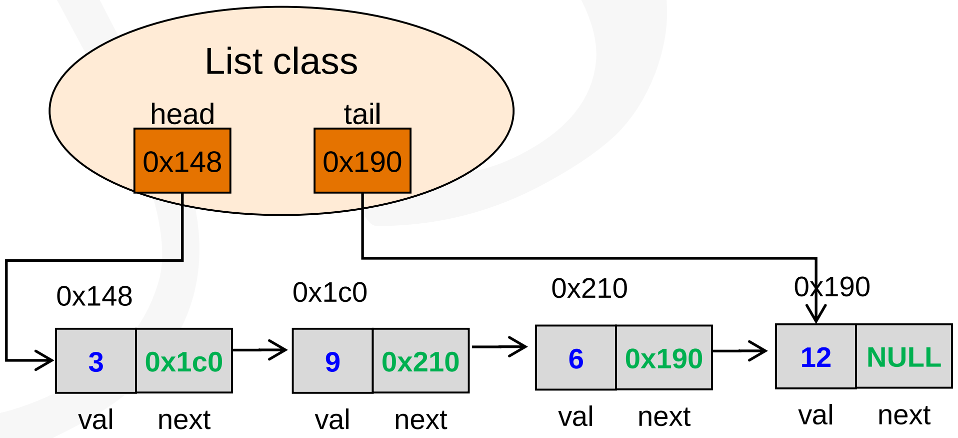
Option 1: (cont.)

- How fast could we add an item to the end? O(1)
- How fast could we remove the tail item?



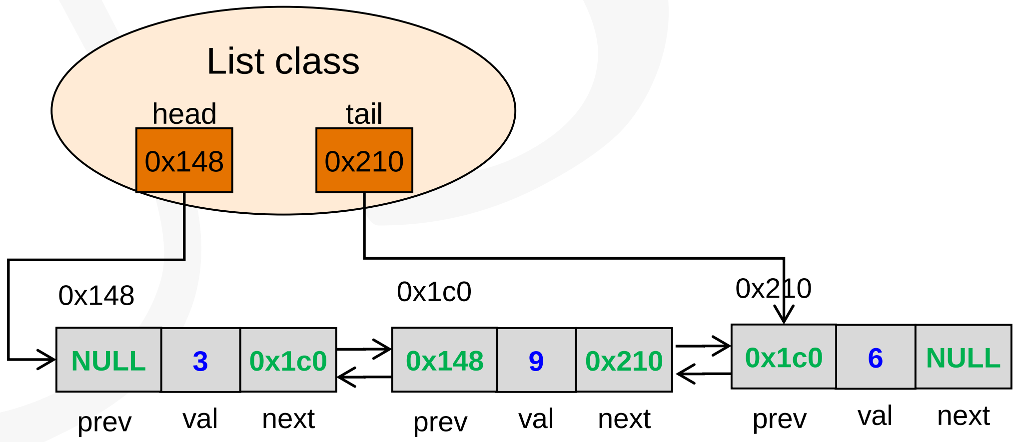
Option 1 (cont.)

- How fast could we add an item to the end? O(1)
- How fast could we remove the tail item? O(n)
  - **Would have to walk to the 2nd to last item**



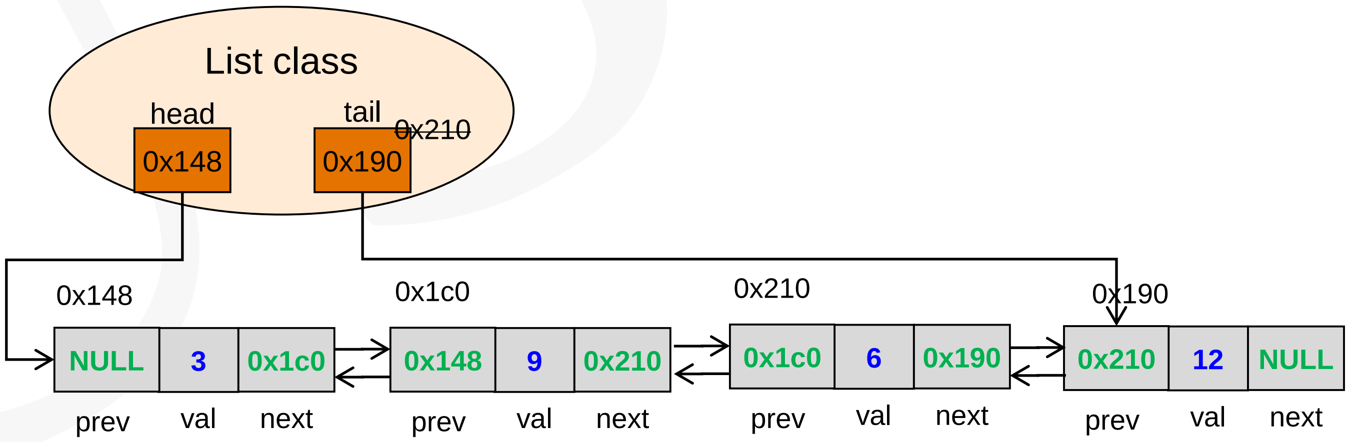
Option 2: Tail Pointer + DLL

- We might think of adding a tail pointer data member to our list class
  - How fast could we add an item to the end?



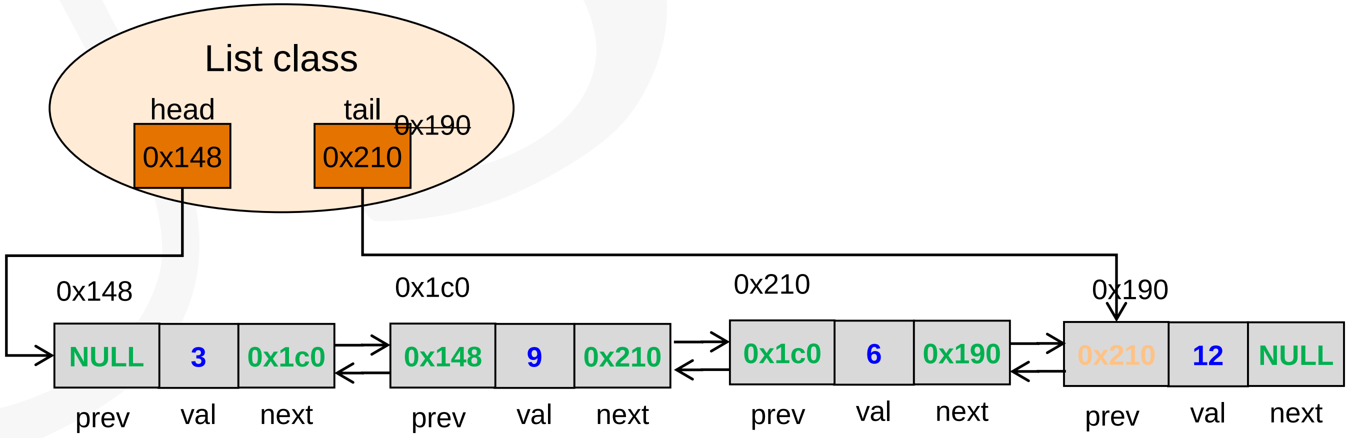
Option 2 (cont.)

- How fast could we add an item to the end? O(1)
- How fast could we remove the tail item?

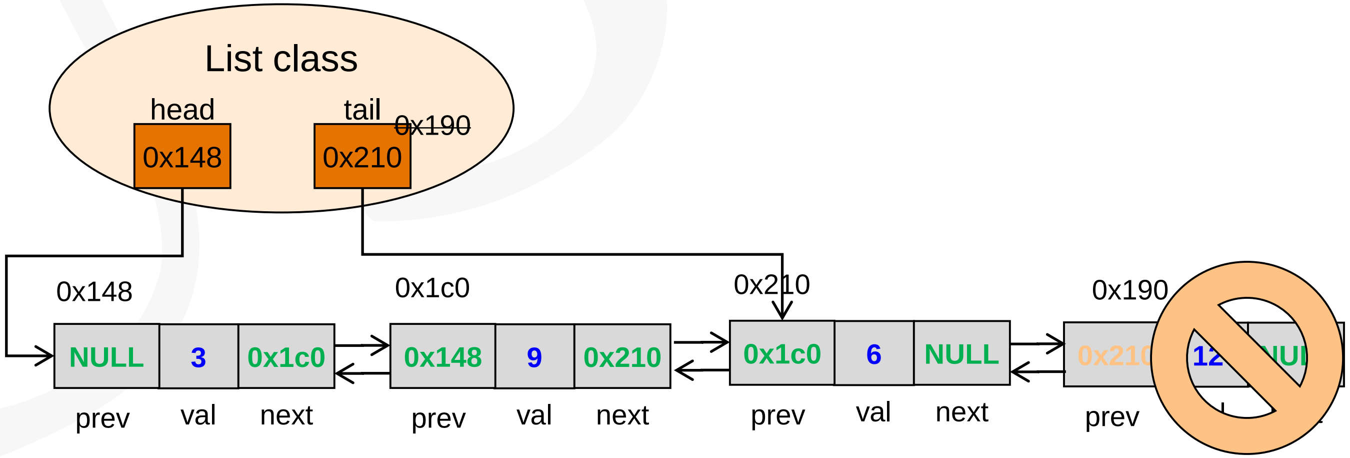


Option 2 (cont.)

- How fast could we remove the tail item? O(1)
  - We use the PREVIOUS pointer to update tail

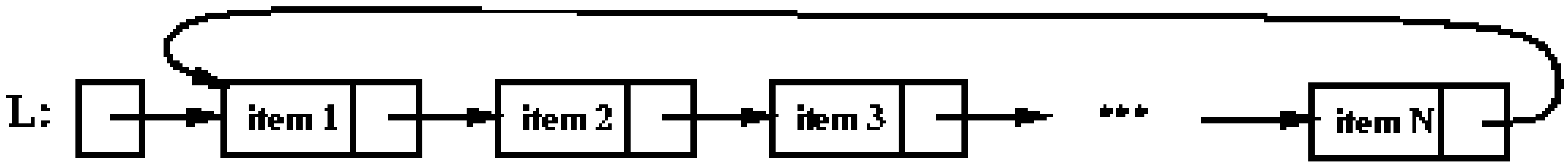


Option 2 (cont.)

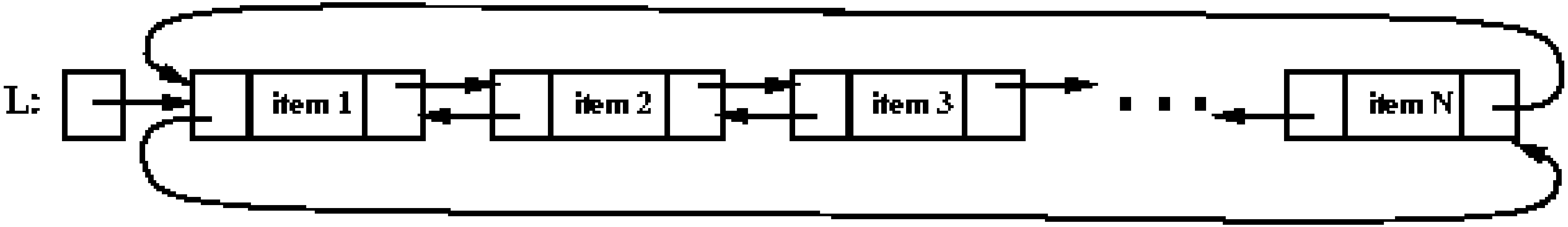


Circular Lists

Circular, singly linked list:

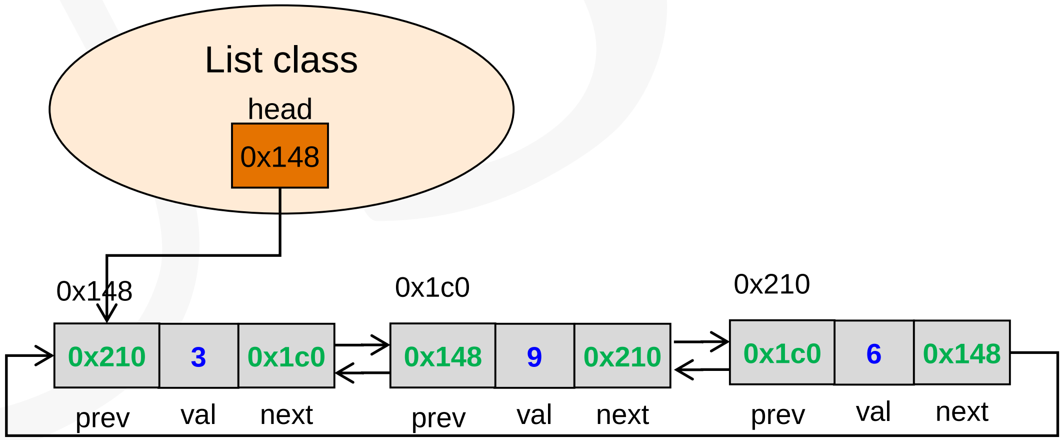


Circular, doubly linked list:



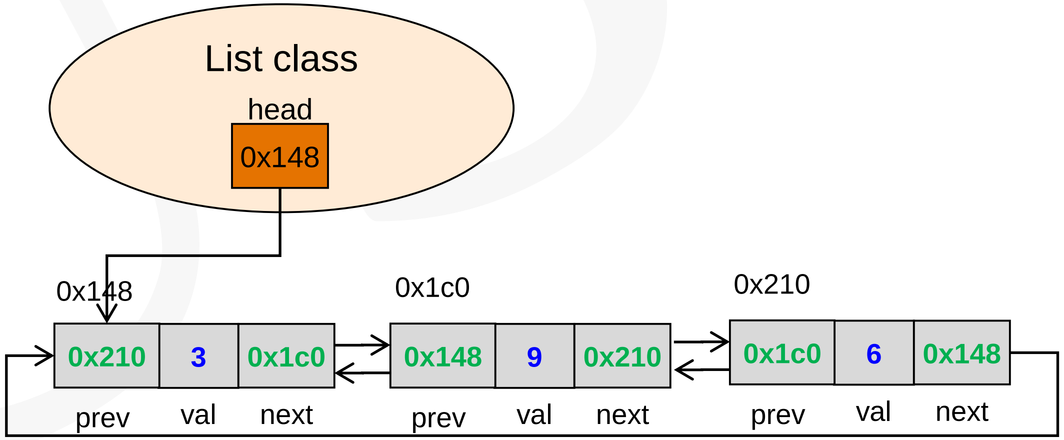
Option 3: Circular DLL

- Make first and last item point at each other to form a circular list
  - We know which one is first via the 'head' pointer



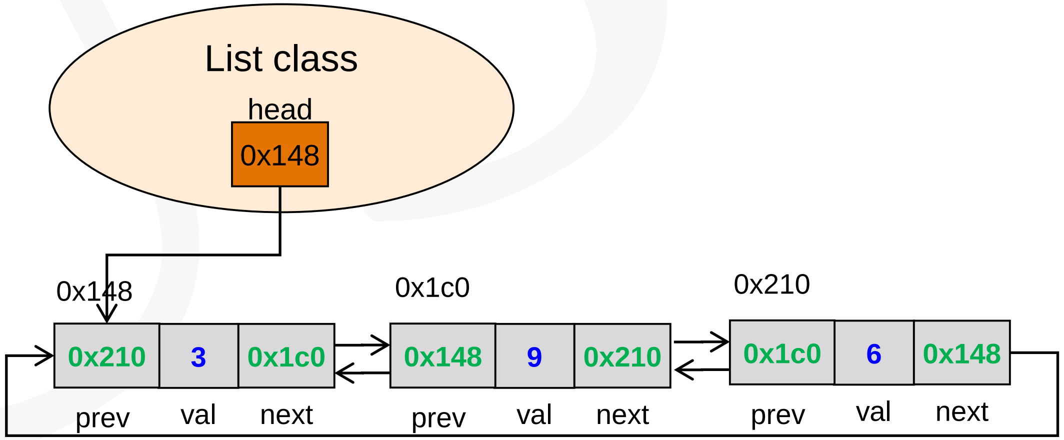
Option 3 (cont.)

- **What expression would yield the tail item?**



Option 3 (cont.)

- What expression would yield the tail item?
  - head->prev

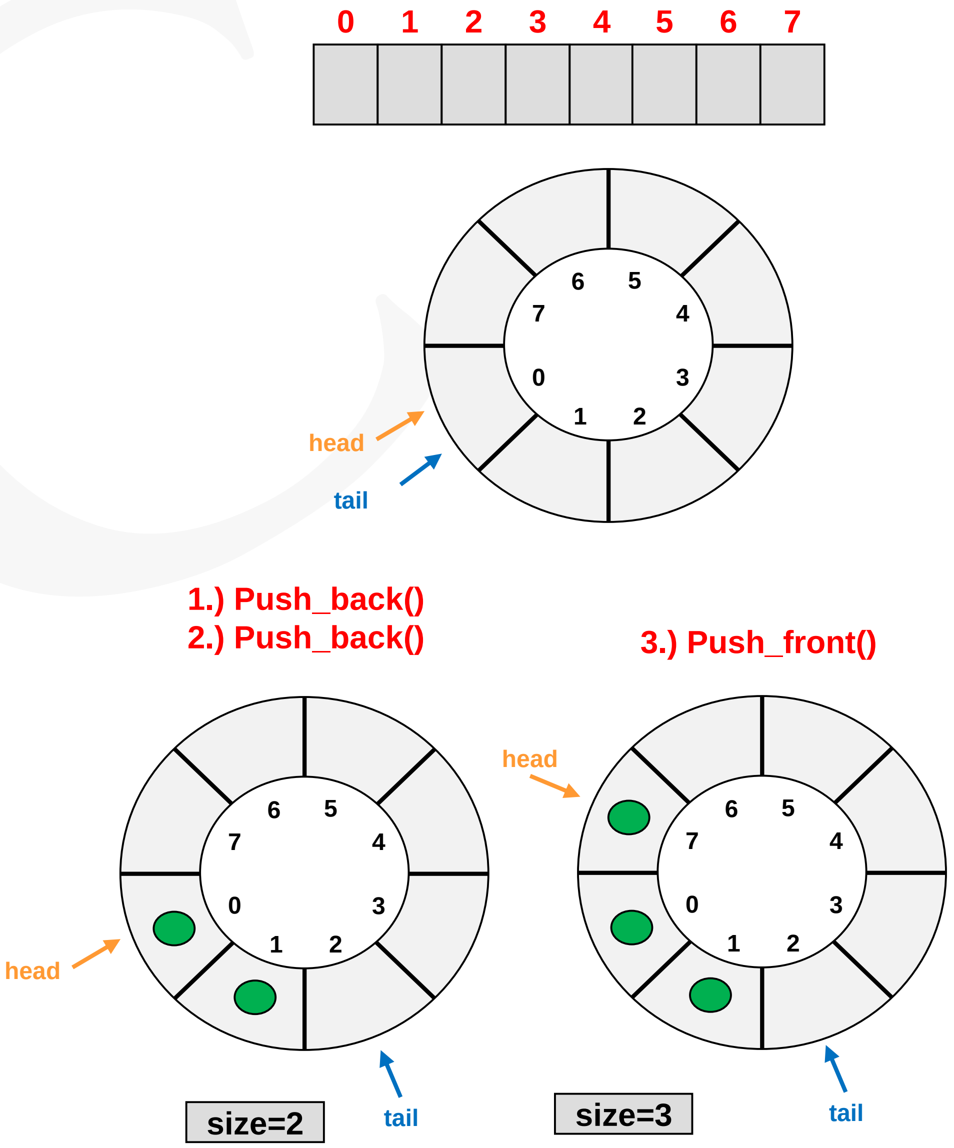


One Last Point

- Can this kind of deque implementation support O(1) access to element i?, i.e., can you access list[i] quickly for any i?
- No!!! Still need to traverse the list
- You can use a "circular" array based deque implementation to get fast random access
  - This is what the actual C++ deque<T> class does
  - Don't worry about this though...

Exercise

- Implement a circular buffer
- You should think about empty and full cases
- What is the best way to handle those cases?



Circular Buffers

- 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
  - Index = 7; Index++ should cause index = 0
  - index = (index + 1)%MAX
  - Index = 0; Index-- should cause index = 7
  - if(--index < 0) index = MAX-1;
- Get item at index i
  - It's relative to the head pointer
  - Return item at (head + i)%MAX