CS 103 Unit 15

Doubly-Linked Lists and Deques

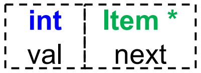
Mark Redekopp

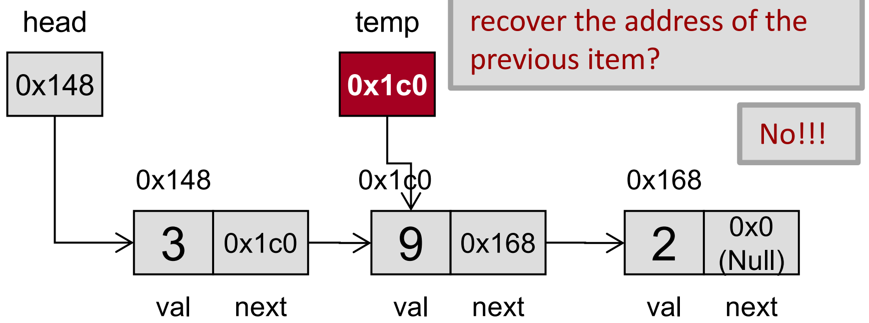
Singly-Linked List Review

- 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:





Given temp...could you ever recover the address of the previous item?

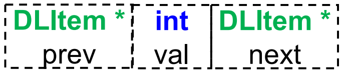
No!!!

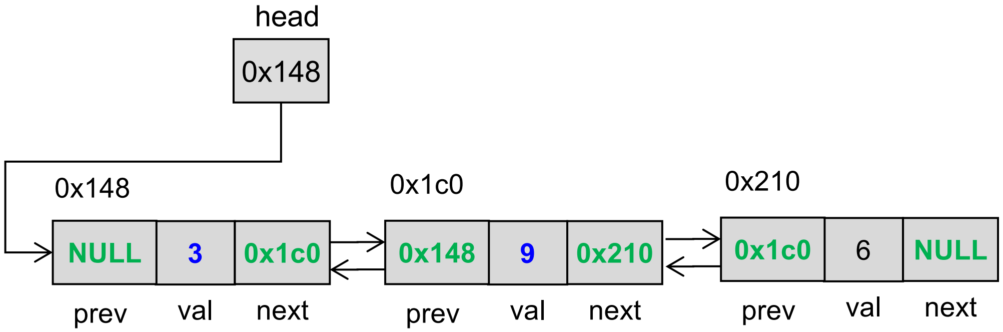
Doubly-Linked Lists

- 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;
};
```

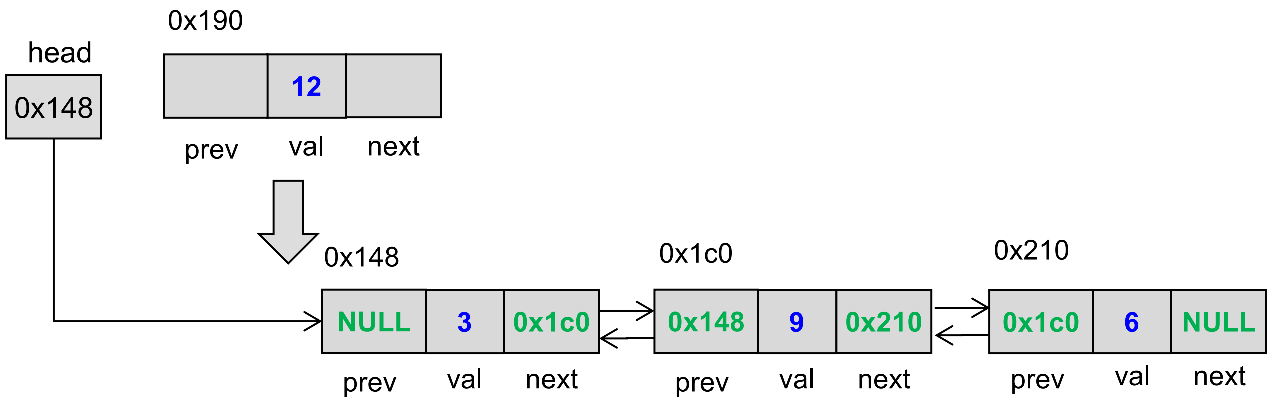
struct Item blueprint:





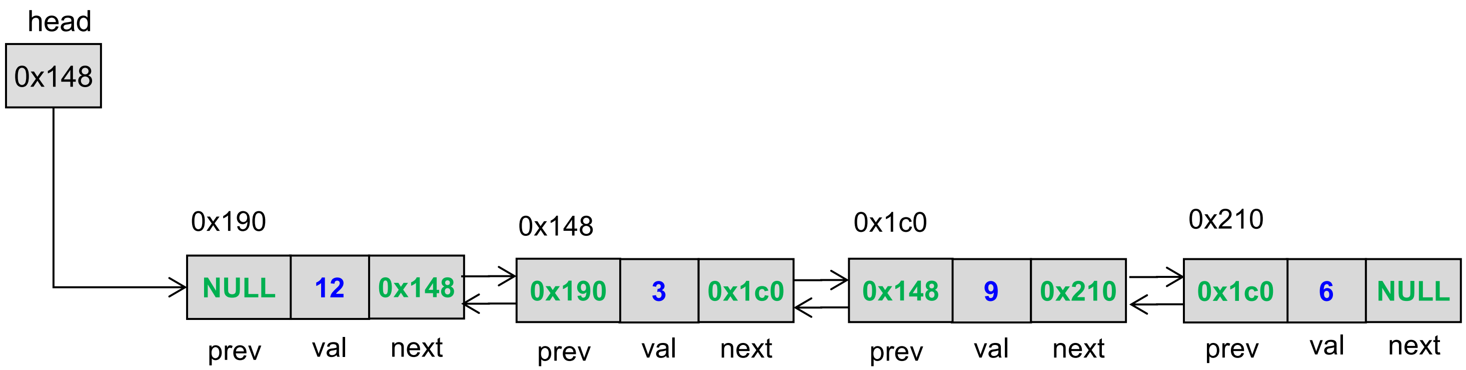
Doubly-Linked List Add Front

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



Doubly-Linked List Add Front

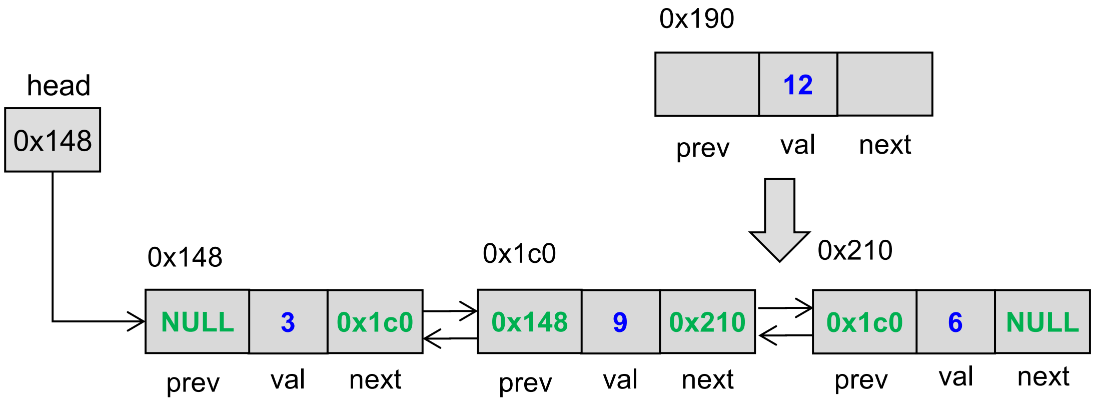
- Adding to the front requires you to update…
  - Head
  - New front's next & previous
  - Old front's previous



Doubly-Linked List 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

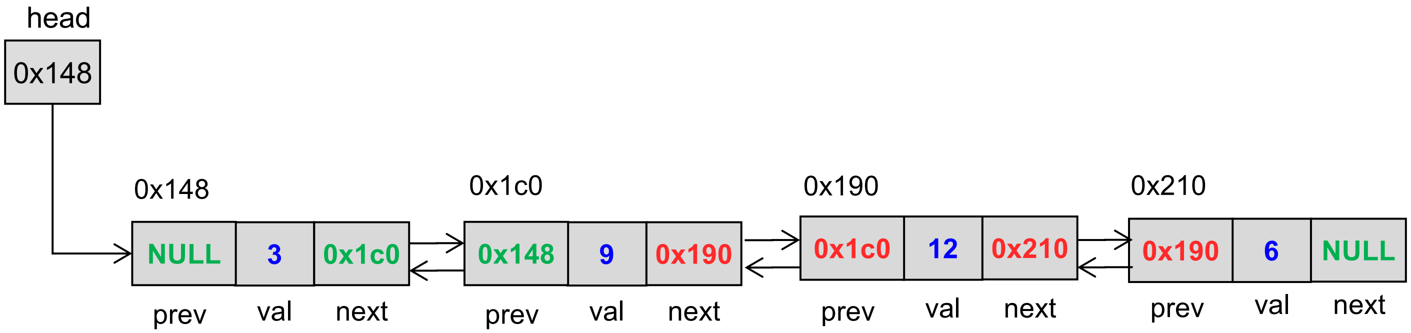
```
curr->prev = temp;
curr->next = temp->next;
if (temp ) {
      temp->next = curr;
}
if (curr->next) {
      curr->next->prev =
curr;
}
```



Doubly-Linked List Add Middle

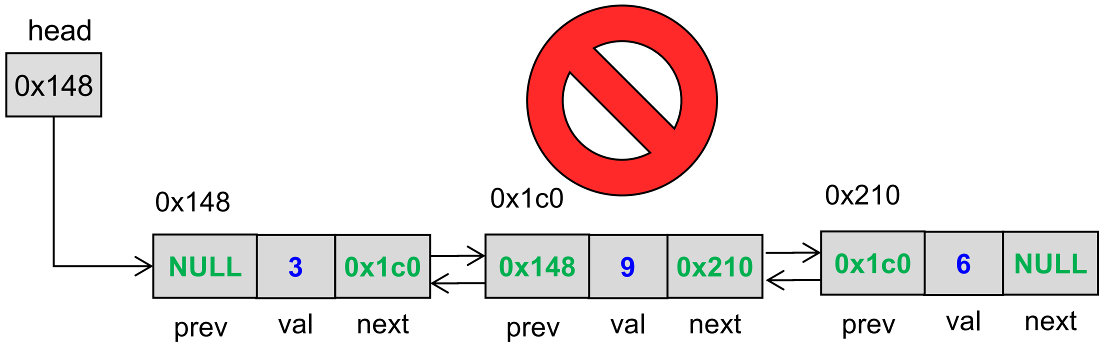
- Adding to the middle requires you to update…
  - Previous item's next field
  - Next item's previous field
  - New item's next field
  - New item's previous field

```
if (curr->prev) {
      curr->prev->next = curr->next;
}
if (curr->next) {
      curr->next->prev = curr->prev;
}
delete curr;
```



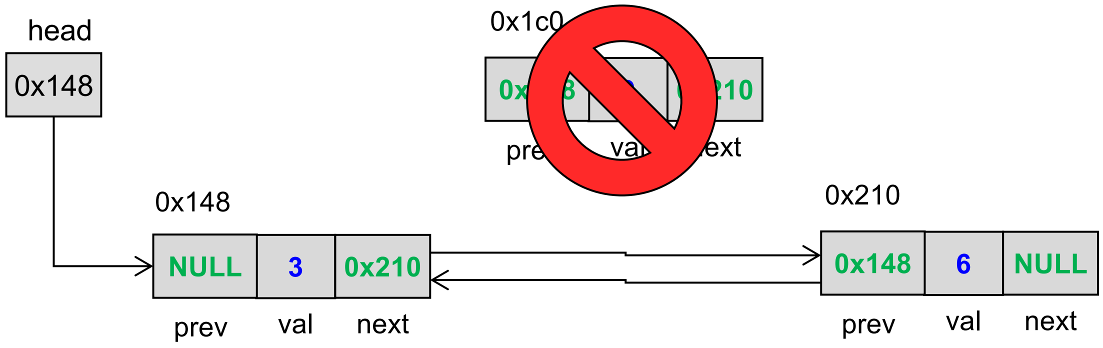
Doubly-Linked List Remove Middle

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



Doubly-Linked List Remove Middle

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



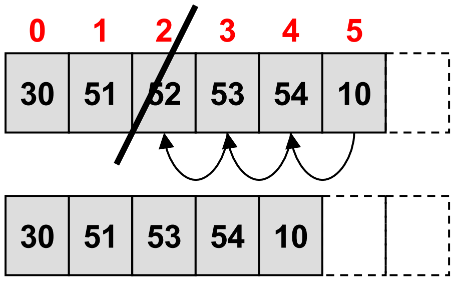
Using a Doubly-Linked List to Implement a Deque

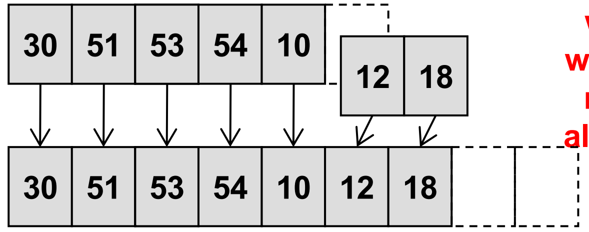
DEQUES AND THEIR IMPLEMENTATION

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

```
  #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;

  }
```

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

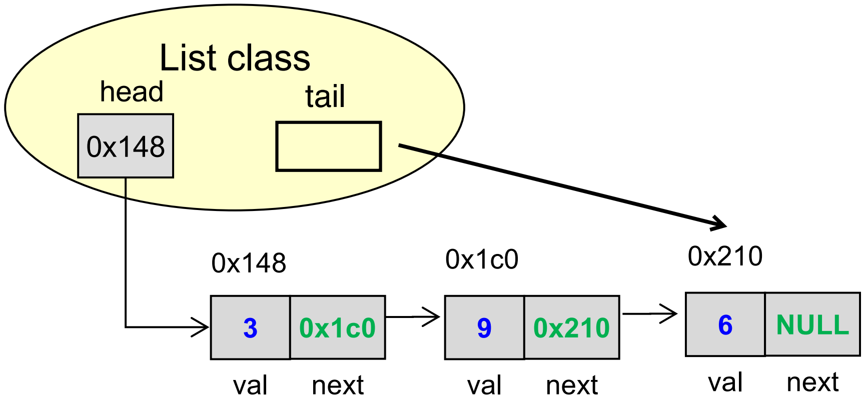
4 my_deq

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?

Singly-Linked List Deque

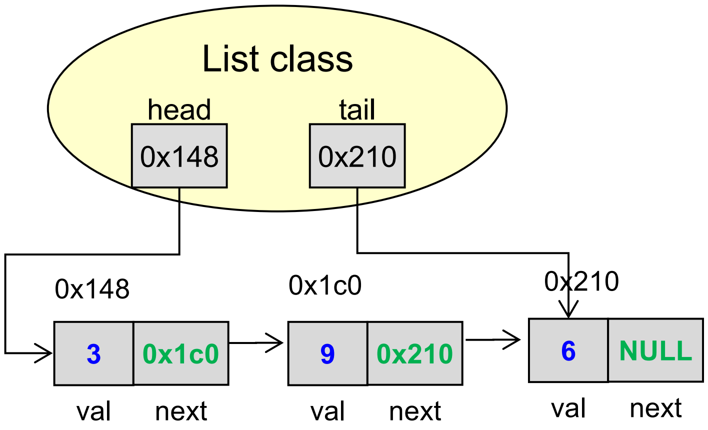
- Recall a deque should allow for fast [i.e. O(1)] addition and removal from front or back
- In our current singly-linked list we only know where the front is and would have to traverse the list to find the end (tail)



easy to push front and pop front; but hard to push back and pop back; therefore to create a new tail item to which last item points; so that four actions only require O(n).

Option 1: Singly-Linked List + Tail Pointer

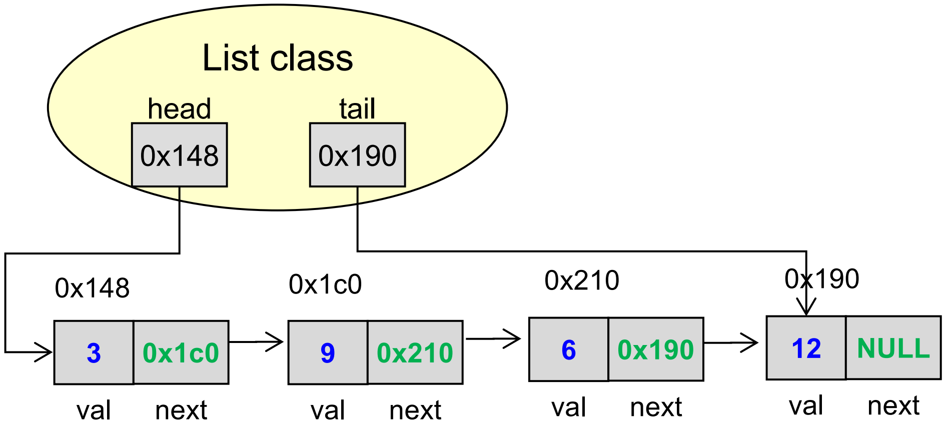
- We might think of adding a tail pointer data member to our list class
  - How fast could we add an item to the end?



*remove from back cannot be done quickly in the singly-linked list but can be done quickly in doubly-linked list.

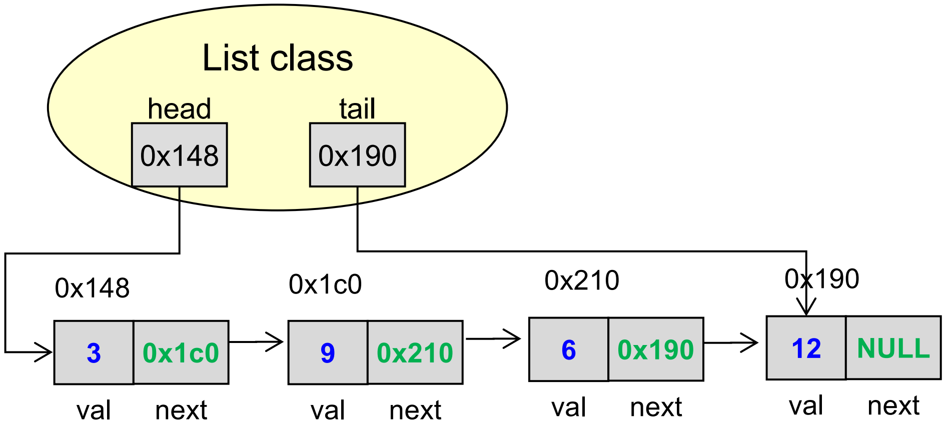
Option 1: Singly-Linked List + Tail Pointer

- We might think of adding a tail pointer data member to our list class
  - How fast could we add an item to the end? O(1)
  - How fast could we remove the tail item?



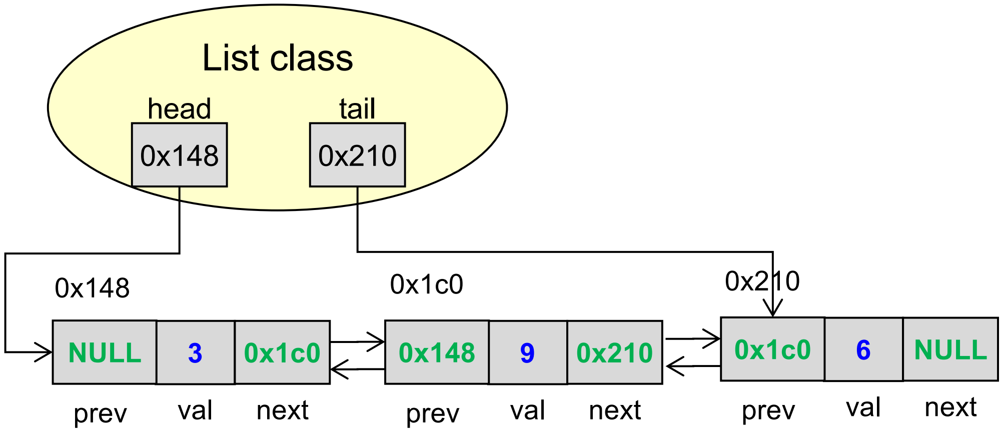
Option 1: Singly-Linked List + Tail Pointer

- We might think of adding a tail pointer data member to our list class
  - How fast could we add an item to the end? O(1)
  - How fast could we remove the tail item? O(n)
    - Would have to walk to the 2nd to last item



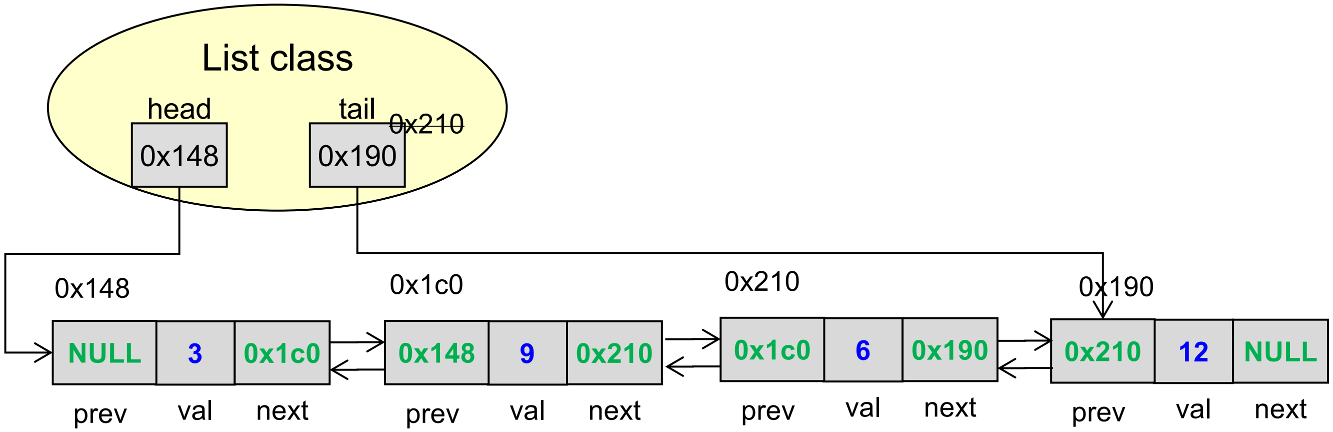
Option 2: Tail Pointer + Double-Linked List

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



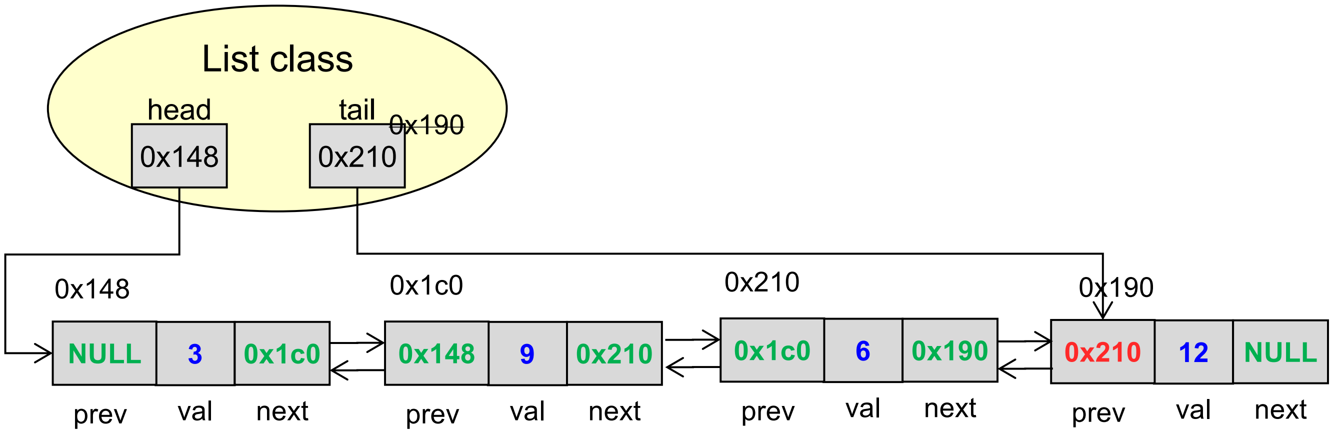
Option 2: Tail Pointer + Double-Linked List

- We might think of adding a tail pointer data member to our list class
  - How fast could we add an item to the end? O(1)
  - How fast could we remove the tail item?



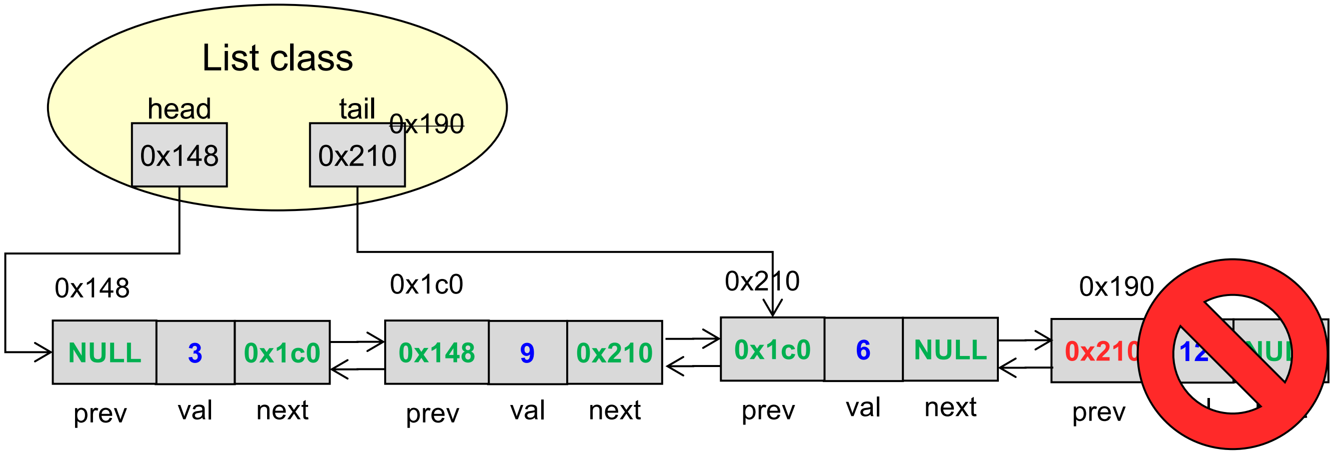
Option 2: Tail Pointer + Double-Linked List

- We might think of adding a tail pointer data member to our list class
  - – How fast could we add an item to the end? O(1)
  - – How fast could we remove the tail item? O(1)
    - We use the PREVIOUS pointer to update tail



Option 2: Tail Pointer + Double-Linked List

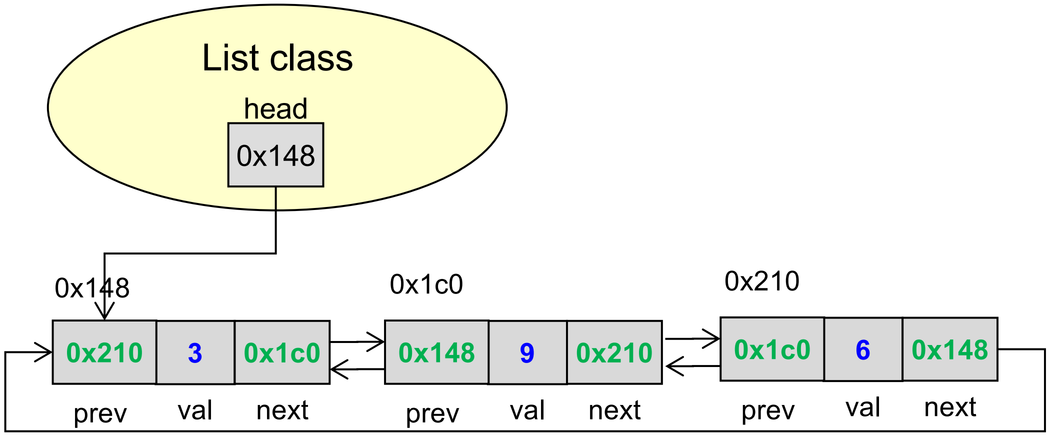
- We might think of adding a tail pointer data member to our list class
  - How fast could we add an item to the end? O(1)
  - How fast could we remove the tail item? O(1)
    - We use the PREVIOUS pointer to update tail



Option 3: Circular Double-Linked List

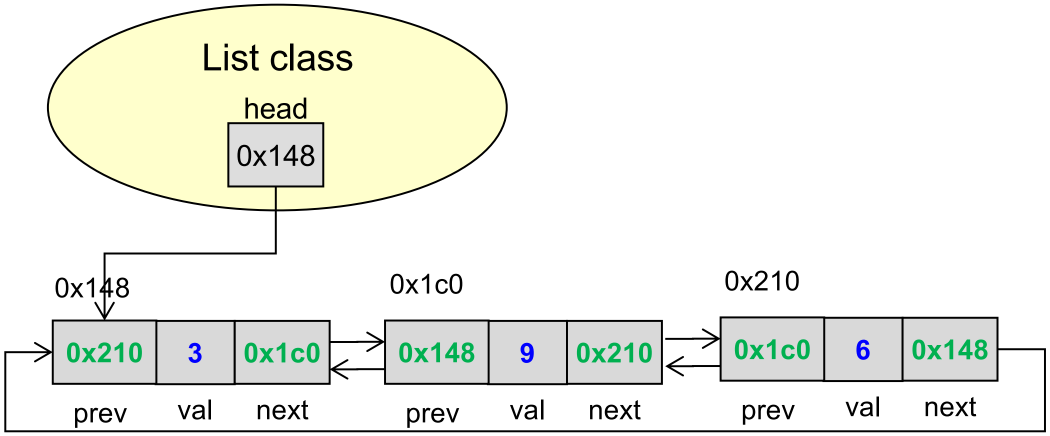
- Make first and last item point at each other to form a circular list
  - We know which one is first via the 'head' pointer

access both sides equally fast,
push back and push front equally fast



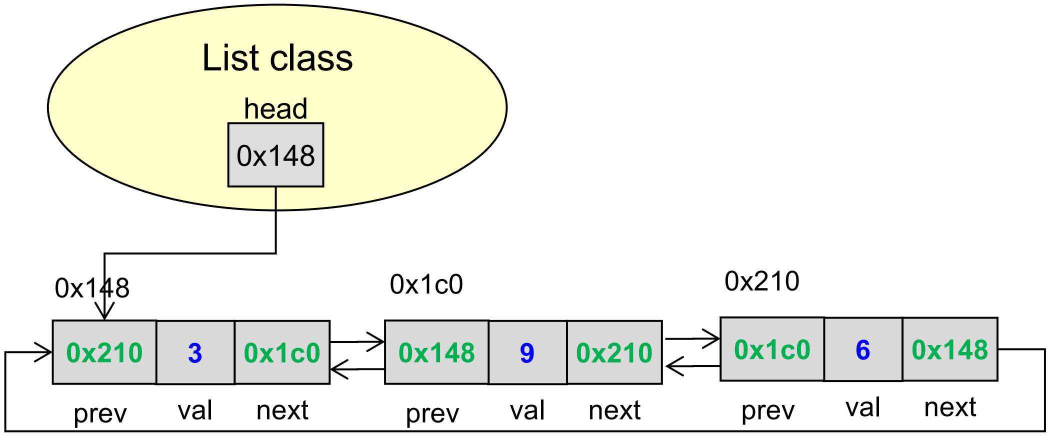
Option 3: Circular Double-Linked List

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



Option 3: Circular Double-Linked List

- Make first and last item point at each other to form a circular list
  - We know which one is first via the 'head' pointer
  - 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 similar to what the actual C++ deque<T> class does
  - More to come in CS 104!

Activity: Write a 'delist' class

- Write a 'double-ended list' class to store integers that mimics a deque
- Support the following methods
  - size()
  - empty()
  - push_back() and pop_back()
  - push_front() and pop_front()
  - back() and front() [returns back or front integer]