VE281

Data Structures and Algorithms

Linked List and Generic Programming

Announcement

- Homework Two will be posted on Sakai by this Saturday.
  - Please check Sakai announcement.

- Due after the national holiday break.
  - Detailed due date will be announced later.

Review

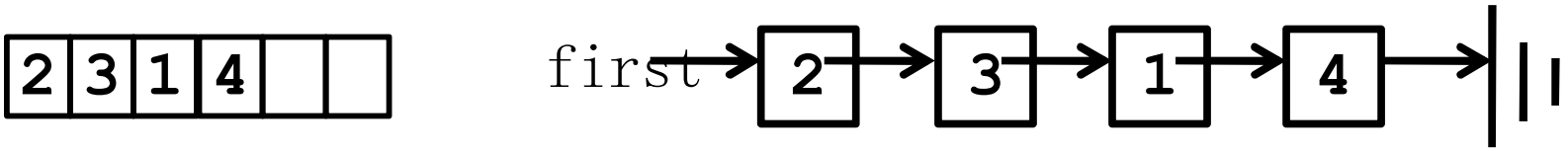
- **Operator<<** for Linear List
- Basics of Linked List
  - **isEmpty()**
  - **insertFirst()**
  - **removeFirst()**
- Linked List Traversal
  - **getSize()**
  - **appendNode()**
  - **removeNode()**

Outline

- Linked List Optimization

Arrays versus Linked Lists

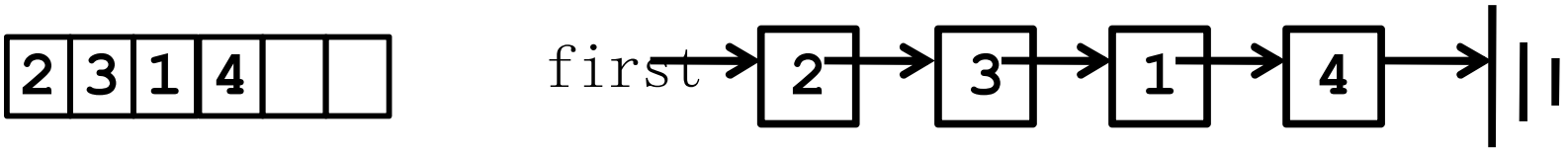
Worst Case Time Complexity



	Array	Linked List
Random access	$O(1)$ time	$O(n)$ time
insertFirst	$O(n)$ time	$O(1)$ time
removeFirst	$O(n)$ time	$O(1)$ time
appendNode	$O(1)$ time	$O(n)$ time
removeNode	$O(n)$ time	$O(n)$ time

Arrays versus Linked Lists

Memory Requirement



	Array	Linked List
Bookkeeping	Pointer to the beginning, Size or pointer to the end	Pointer to the first no “next” pointer in eac
Memory	Free in $O(1)$ time Wastes memory if size is too large. Requires reallocation if too small.	Free in $O(n)$ time Allocates memory as needed. Allocation and de-allocation costly.

Linked List Optimization

getSize()

- How to reduce the complexity of **getSize()**?
- Hint: remember the space-time tradeoff?

```
class LinkedList {
   node *first;
   int size;
public:
   ...
};
```

```
int LinkedList::getSize()
{
    return size;
}
```

- Question: do we need to change any other parts of the code?
  - We need to increment/decrement **size** when nodes are inserted/removed.

Complexity Discussion

- When **`getSize()`** is called frequently, does this implementation reduce the overall time complexity?

- When **`getSize()`** is not called very often, does this implementation reduce the overall time complexity?

  Hint: In analyzing the overall complexity, remember the cost overhead of insert and remove.

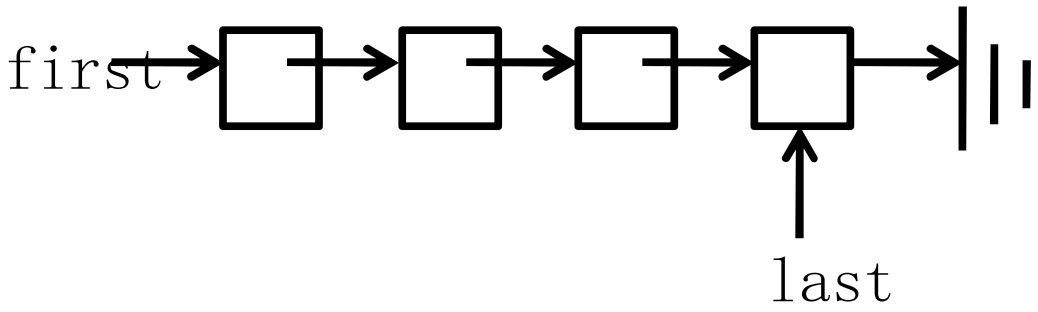
Linked List Optimization

appendNode()

- How to reduce the complexity of **appendNode(node *n)**?

- Use double-ended list

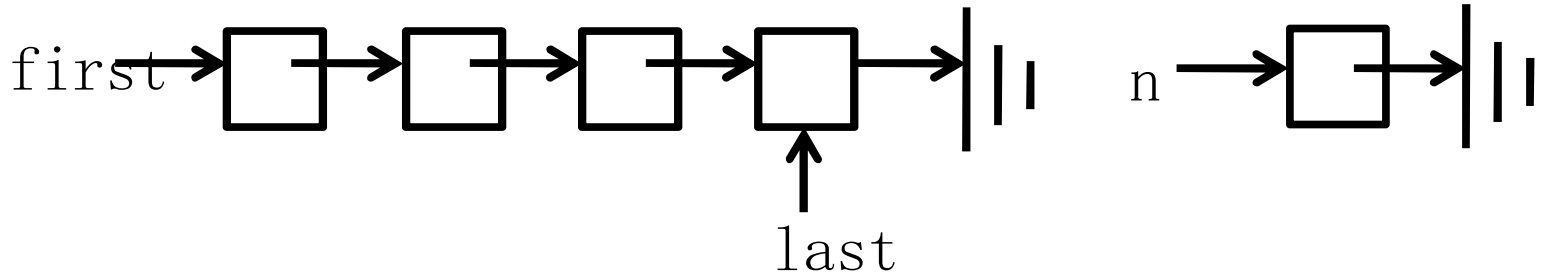
```
class LinkedList {
   node *first;
   node *last;
public:
   ...
};
```



Linked List Optimization

appendNode()

- Append a node with a double-ended list




```
void LinkedList::appendNode(node* n){
  if(!first) first = last = n;
  else {
    last->next = n;
    last = n;
  }
}
```

Time complexity becomes $O(1)$.

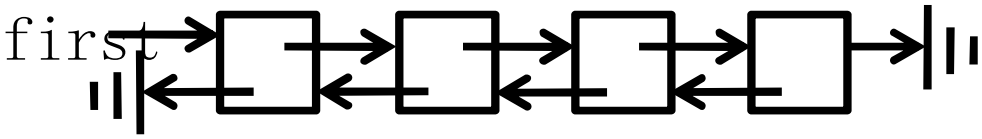
Linked List Optimization

removeNode()

- How to reduce the complexity of **removeNode(node *n)**?

- Use doubly-linked list

```
struct node {
    node *next;
    node *prev;
    int value;
};
```

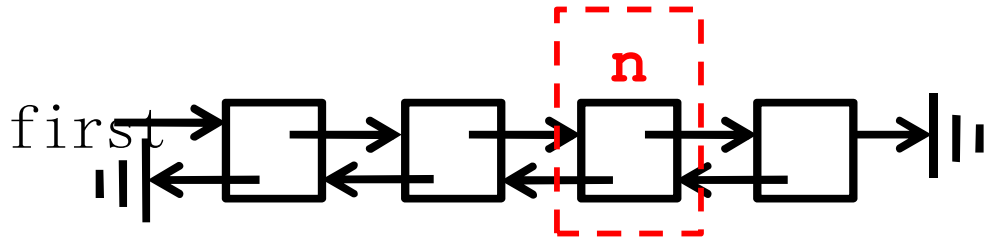


First node's **prev** and last node's **next** point to NULL.

Linked List Optimization

removeNode()

- Remove a node with a doubly-linked list




```
void LinkedList::removeNode(node* n){
  // A simplified version without
  // considering boundary situations.
  n->prev->next = n->next;
  n->next->prev = n->prev;
  delete n;
}
```

Time complexity becomes $O(1)$.

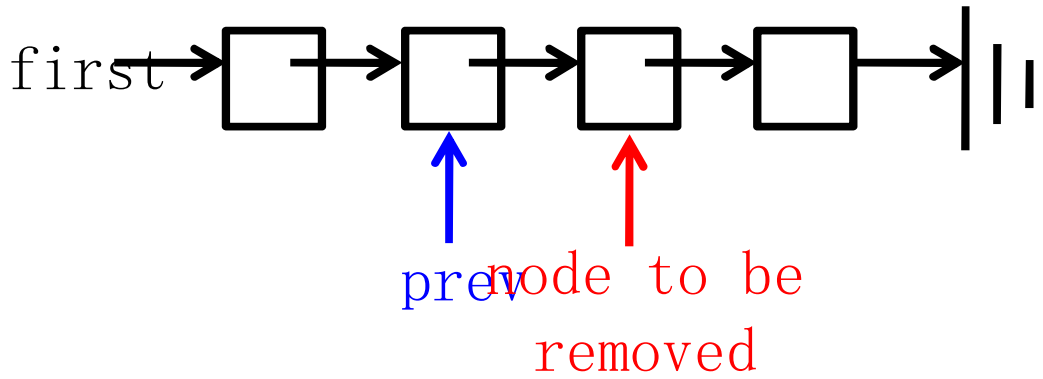
Linked List Optimization

removeNode()

- The worst case time complexity of removing a node is $O(n)$ for a singly-linked list.
- Can we do better for a **singly-linked list**?

- Point to the node "previous" to the node to be removed.

```
removeNode(node *prev)
```

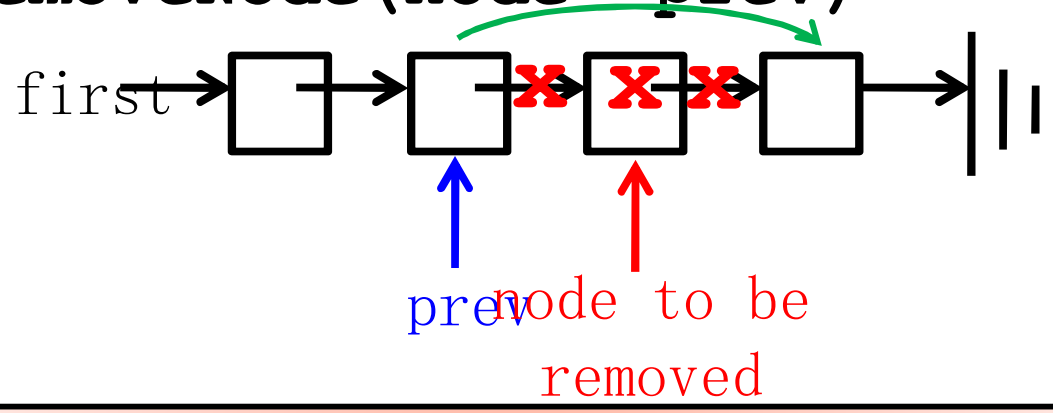


Linked List Optimization

removeNode()

- Point to the node "previous" to the node to be removed.

removeNode(node *prev)

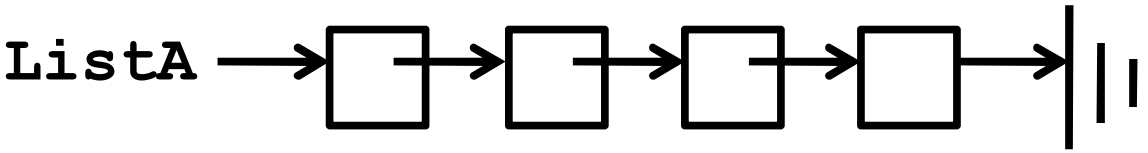


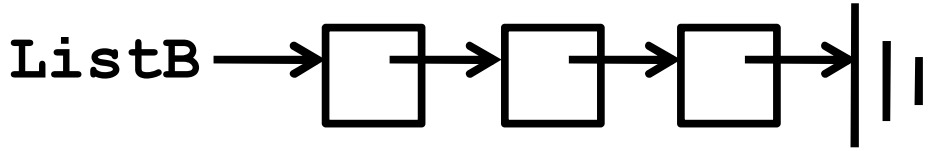

```
node *del_node = prev->next;
prev->next = prev->next->next;
delete del_node;
```

- Use a **dummy** "previous" node when removing the first node.

Exercise: Merge Linked List

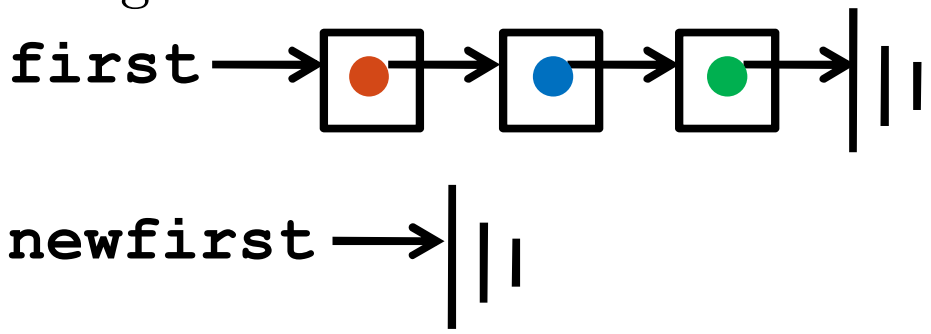
- How long does it take to merge **ListA** and **ListB** into one list?
- What if both lists are double-ended?

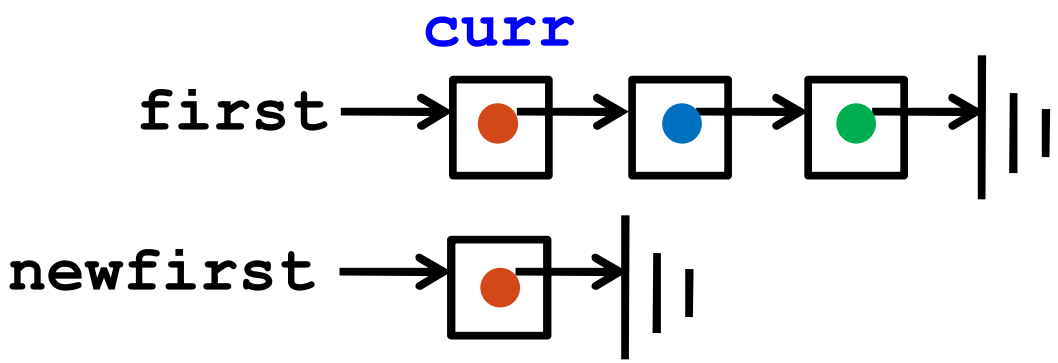




Reversing a Linked List

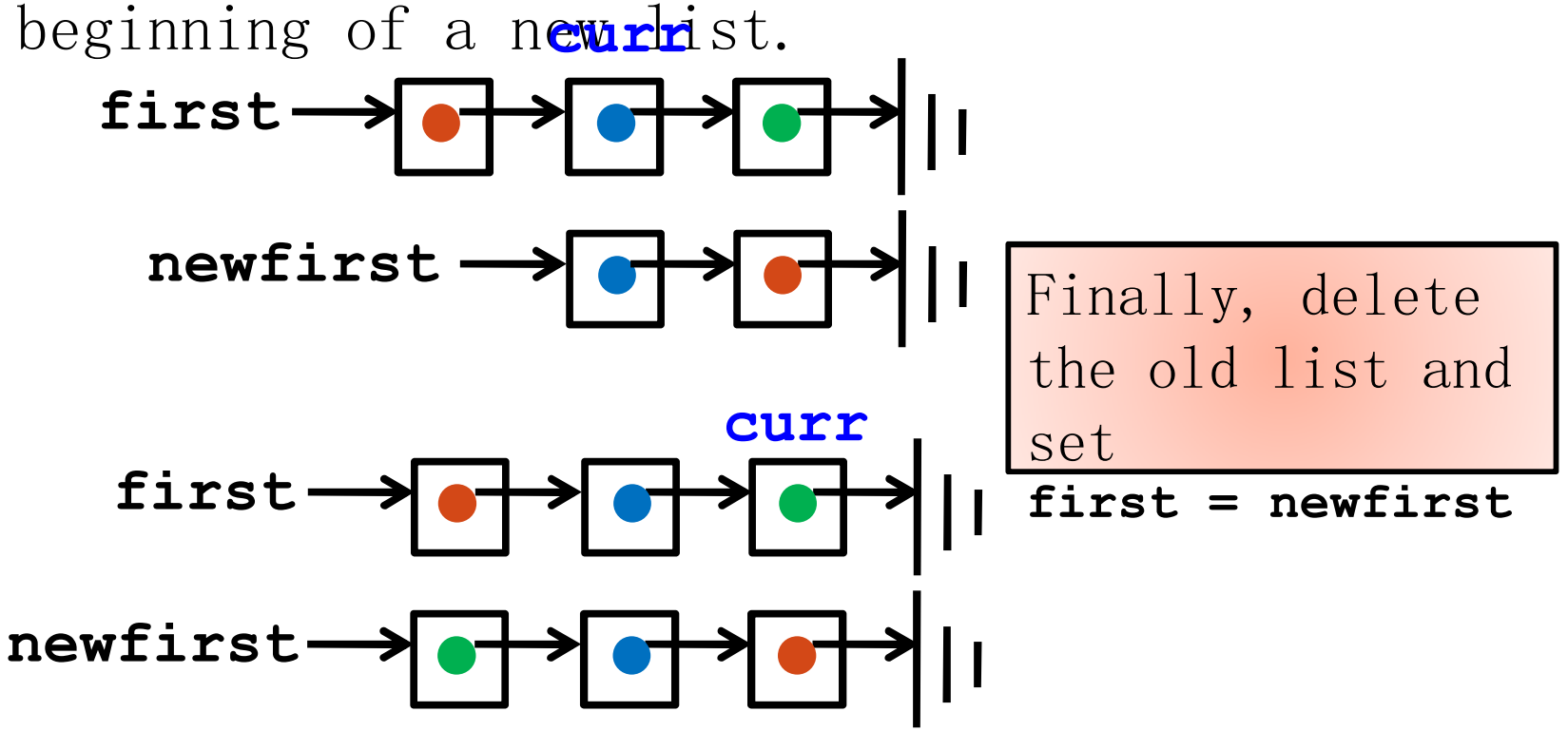
- Traverse the old list. Each time visit a node, insert a copy of that node at the beginning of a new list.



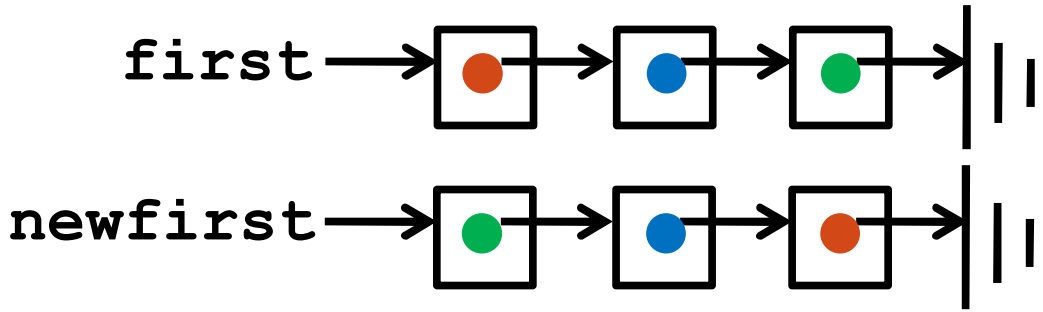


Reversing a Linked List

- Traverse the old list. Each time visit a node, insert a copy of that node at the beginning of a new list.



Complexity of Reversing a Linked List

- 

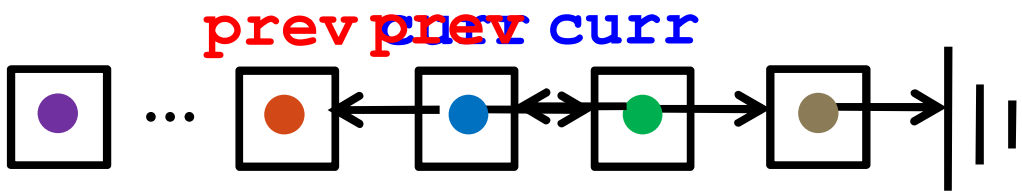
- How long does reversal take?
- How much memory is needed?

- Can reversal be made more space efficient?
  - I.e., can we reverse with only $O(1)$ additional memory?
- Can reversal be made more time efficient?

Reversing a Linked List

Algorithm with O(1) Space Complexity

- Keep a **prev** pointer
- Set **curr->next** to **prev**
- Advance both **prev** and **curr**.



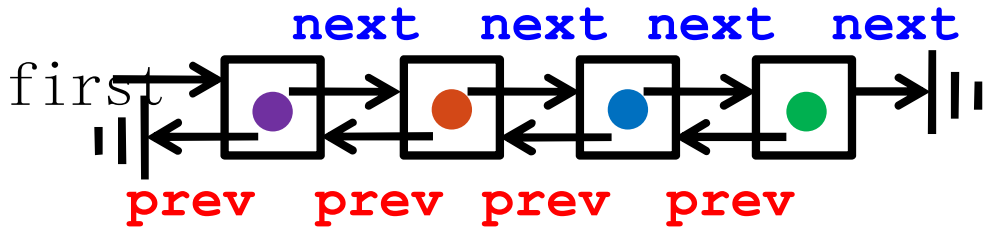
Reversing a Linked List

Algorithm with O(1) Space Complexity

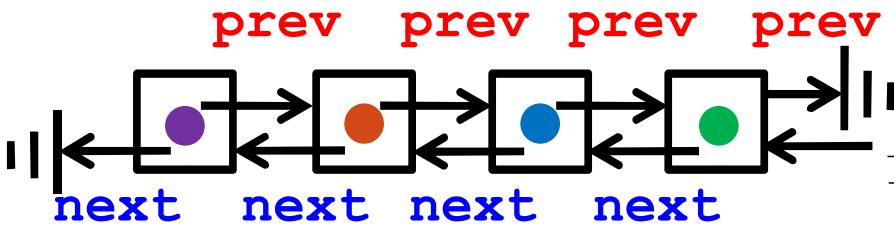
```
void LinkedList::reverse() {
  node *curr = first;
  node *prev = NULL;
  node *next = NULL;
  while(curr != NULL){
    next = curr->next; // Record next pointer
    curr->next = prev; // Reverse
    prev = curr; // Advance prev
    curr = next; // Advance curr
  }
  first = prev; // Set new first as the one
                // "previous to NULL"
}
```

Additional memory:
three pointers -- $O(1)$

Reversing a Doubly-Linked List

- 

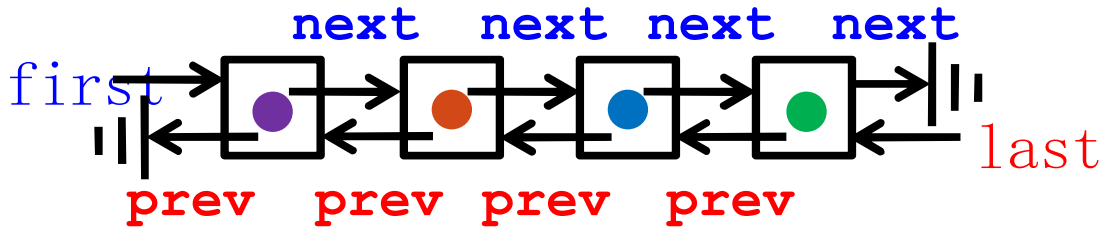
- What is the reversal of a doubly-linked list?



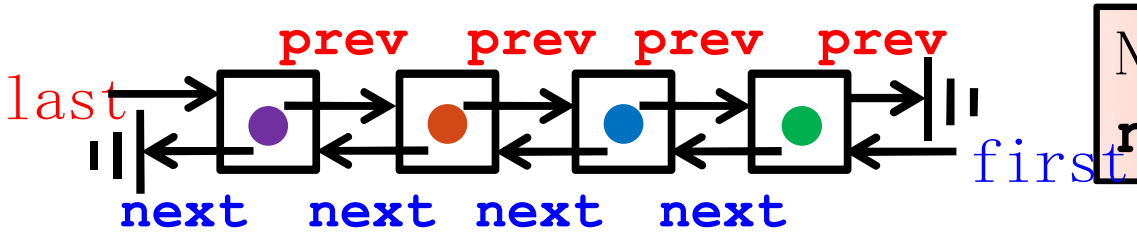
Note that **prev** and **next** are exchanged!

- Can we reverse the list in $O(1)$ memory?
- Can we reverse the list in $O(1)$ time?

Reversing a Double-Ended, Doubly-Linked List

- 

- The reversal of the list is



Note that **prev** and **next** are exchanged

- Can we reverse the list in $O(1)$ memory?
- Can we reverse the list in $O(1)$ time?

Speeding-up Allocation/De-allocation

Free List

- The allocation (**new**) and de-allocation (**delete**) operation in the system are slow.

- Prepend deleted node to free list instead of de-allocating them.