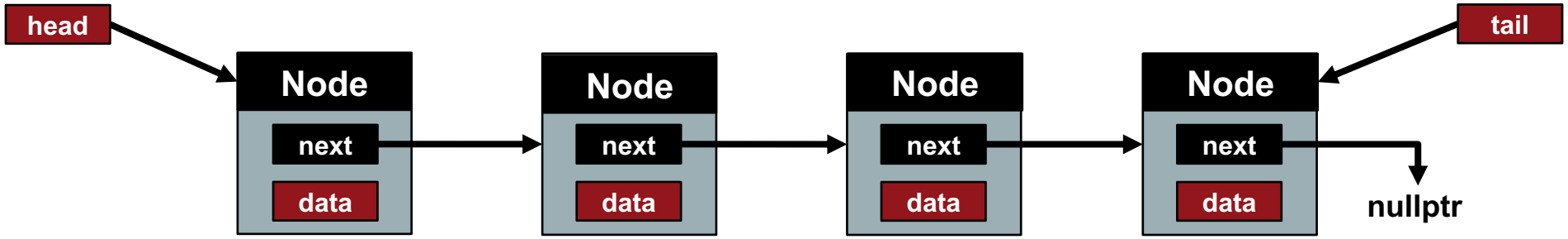
LINKED DATA STRUCTURES PART II

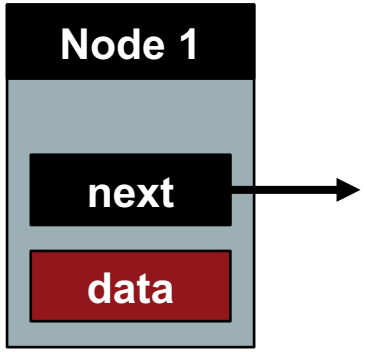
Singly Linked List

Concept	a list of nodes where each node has one link to the next node the last node points to nullptr
Head	a pointer to the first node in the linked list
Tail	a pointer to the last node in the linked list
Note	a singly linked list can only be traversed forwards (from head to tail)



Singly Linked List Node

Concept	singly linked list nodes contain two variables, data and next
Data	the information to be stored
Next	a pointer to the next node in the linked list



Singly Linked Node Class

Concept node classes are custom designed for the container they will be used with

Example a node class to store integers in a singly linked list

```
class Node {
public:
    int data;                                   // integer data
    Node *next;                                 // pointer to next node

    Node(const int &d): data(d), next(nullptr) { }   // node constructor
                                                // next points to nullptr
};
```

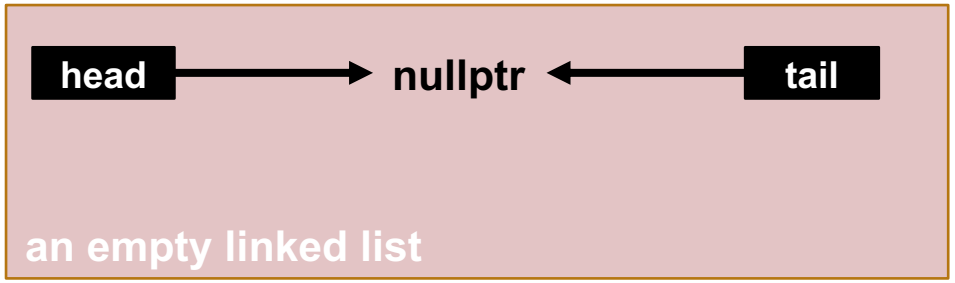
Note this is a simple implementation where the entire class is publicly accessible

Singly Linked List

Concept construct an empty list which contains no node objects

Example

```
class LList {
private:
    Node *head;                                  // pointer to the first node
    Node *tail;                                  // pointer to the last node
    int size;                                    // track # of nodes
public:
    Llist(): head(nullptr), tail(nullptr), size(0) { }   // head and tail set to nullptr
    additional functions
};
```



Common Singly Linked List Functions

empty	return true if the list is empty (list contains no nodes)
push_back	add a node to the end of the list (append)
push_front	add a node to the start of the list (prepend)
pop_front	remove the first node in the list
pop_back	remove the last node in the list
erase	erase a node from the list (node specified by data value)
insert	insert a node before a specified node in the list

Singly Linked List: Push_Front

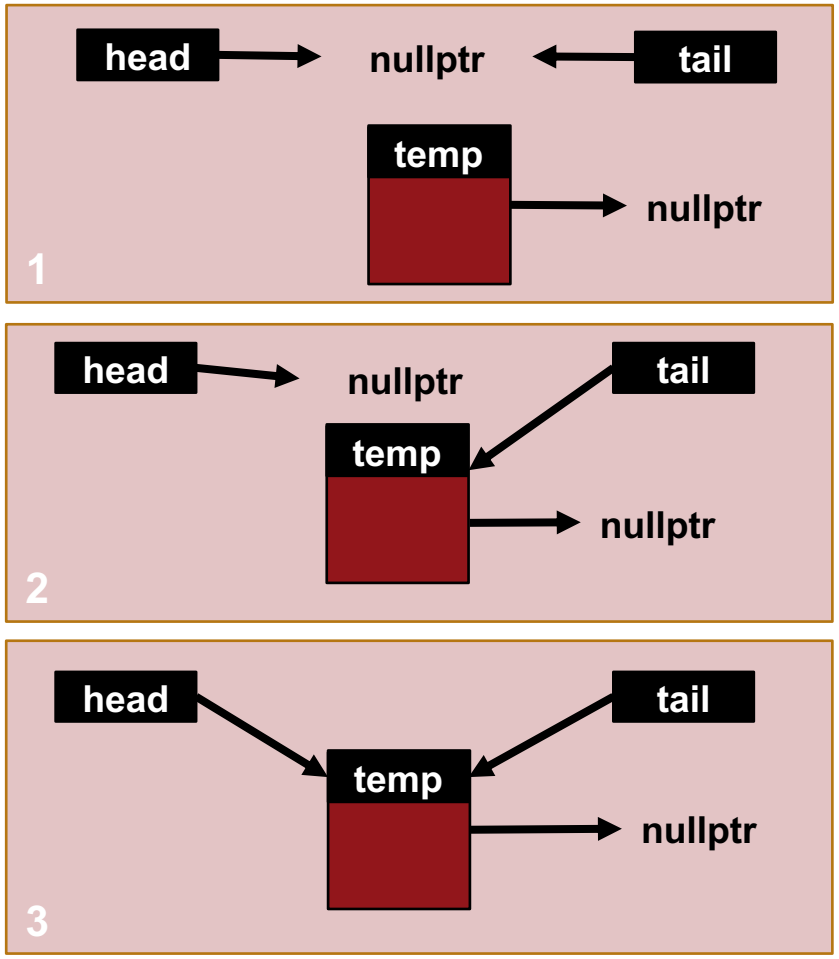
Concept add a node to the start of a list: O(1)

Example

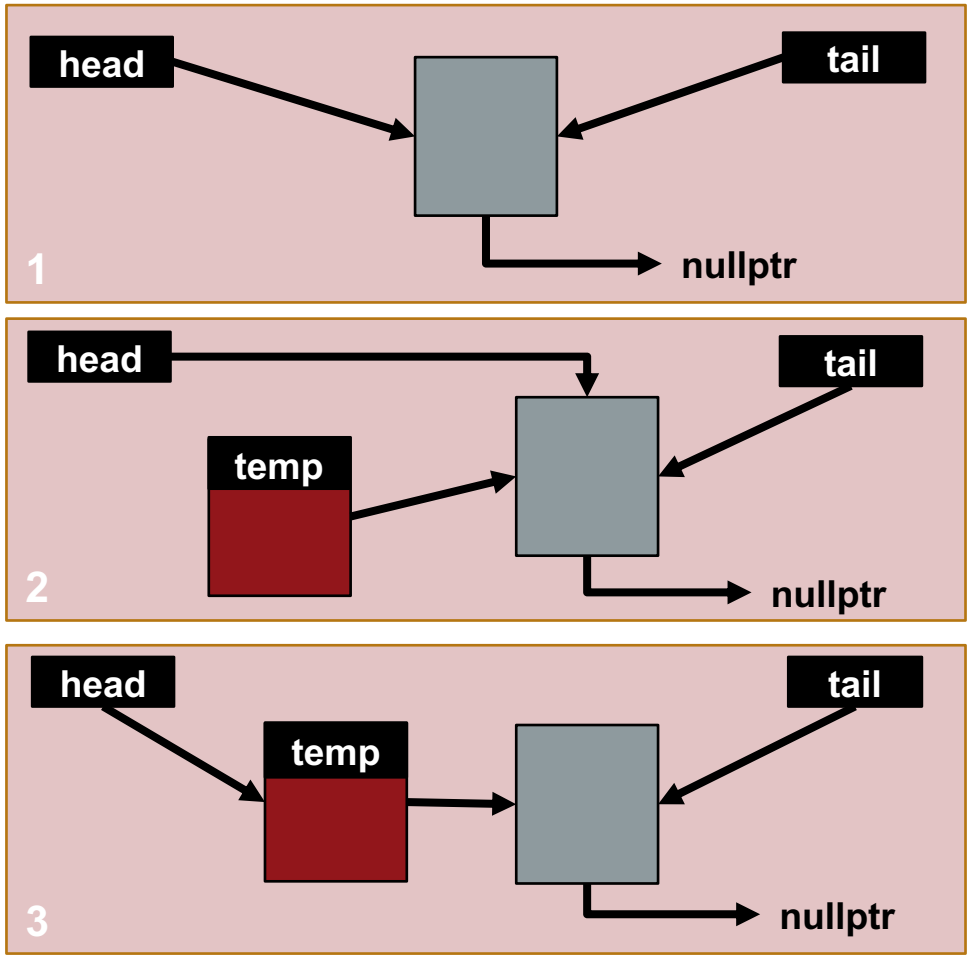
```
void push_front(int n) {                // function accepts an integer
    Node *temp = new Node(n);           // create a new node with the integer data
    if(head == nullptr) {               // list empty:
        tail = temp;                    //    point tail to new node
    } else {                            // list not empty:
        temp->next = head;              //    new node points to current head
    }
    head = temp;                        // point head to new node
    ++size;                             // increment size
}
```

Singly Linked List: Push_Front

Case 1: empty list



Case 2: non-empty list



Singly Linked List: Push_Back

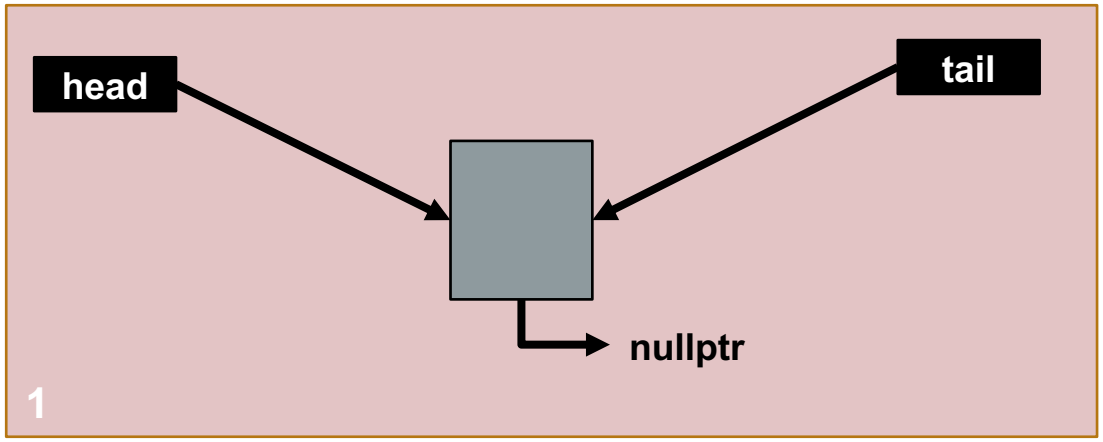
Concept add a node to the end of a list: O(1)

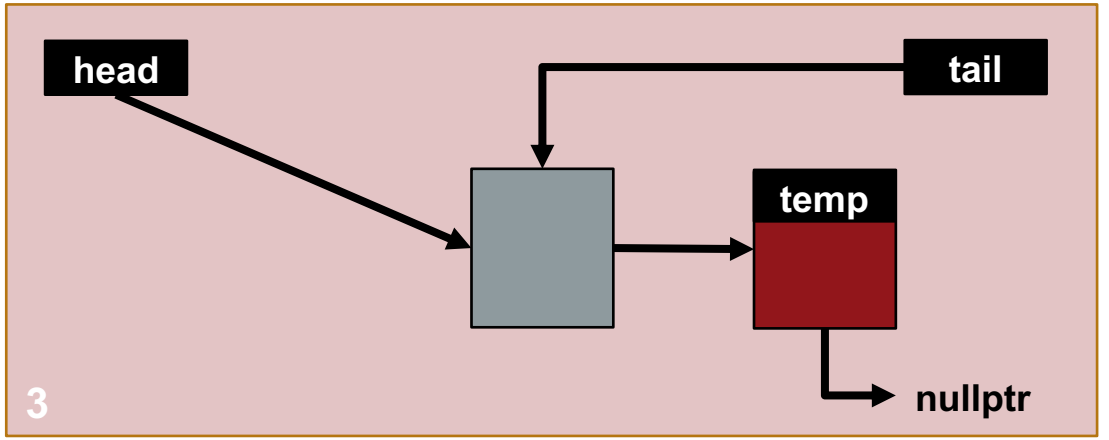
Example

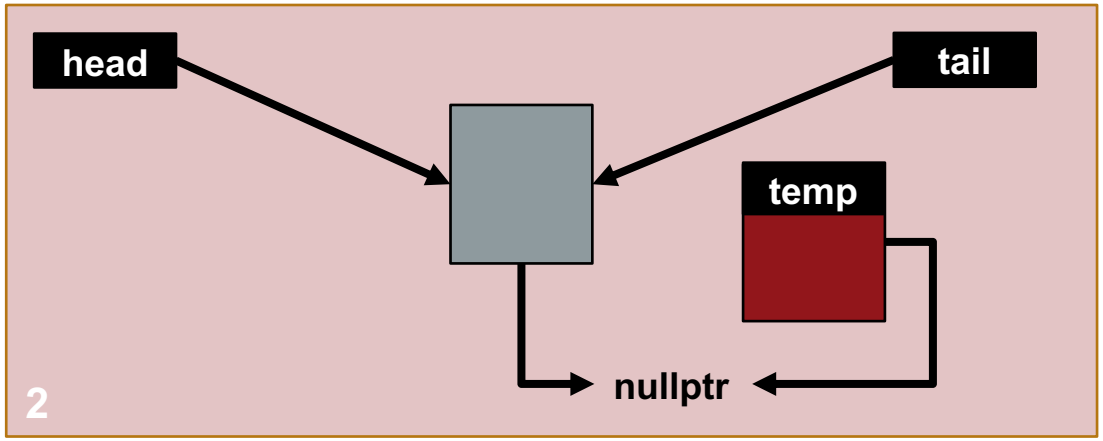
```
void push_back(int n) {                 // function accepts an integer
    if(head == nullptr) {               // list empty: call push_front and return
        push_front(n);
        return;
    }
    Node *temp = new Node(n);           // create a new node
    tail->next = temp;                  // point current tail node to new node
    tail = temp;                        // point tail to new node
    ++size;                             // increment size
}
```

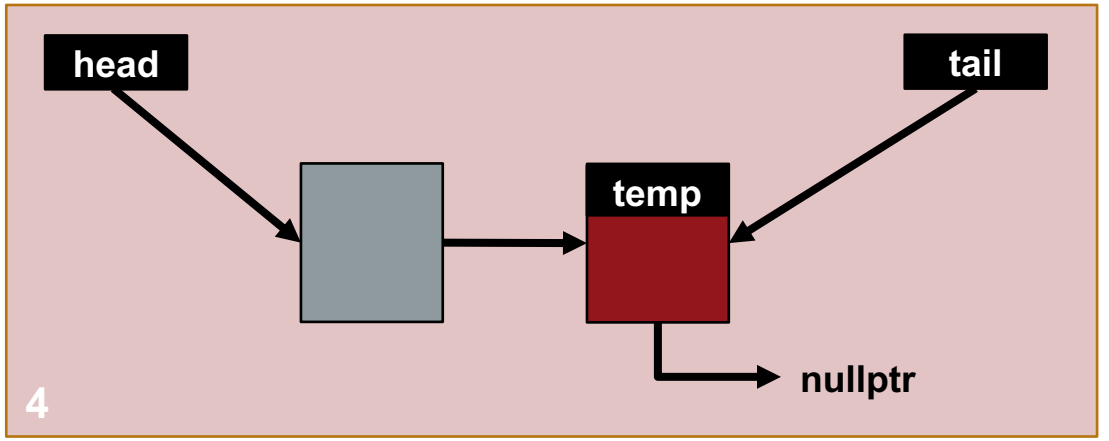
Singly Linked List: Push_Back

Case 2: non-empty list









Singly Linked List: Pop_Front

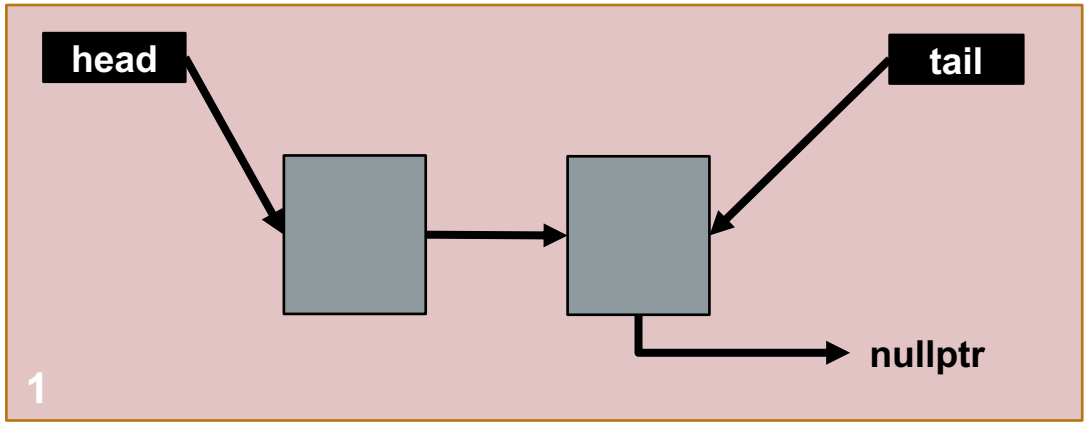
Concept remove the first node in the list: O(1)

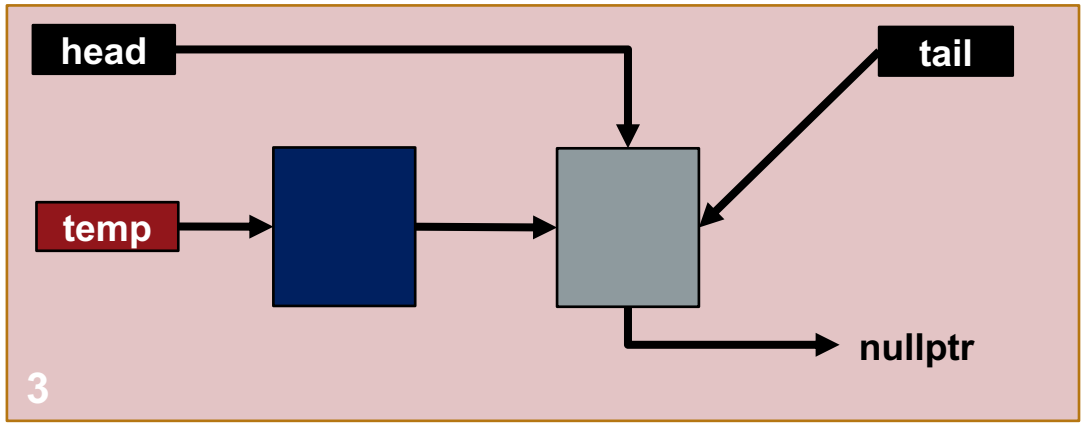
Example

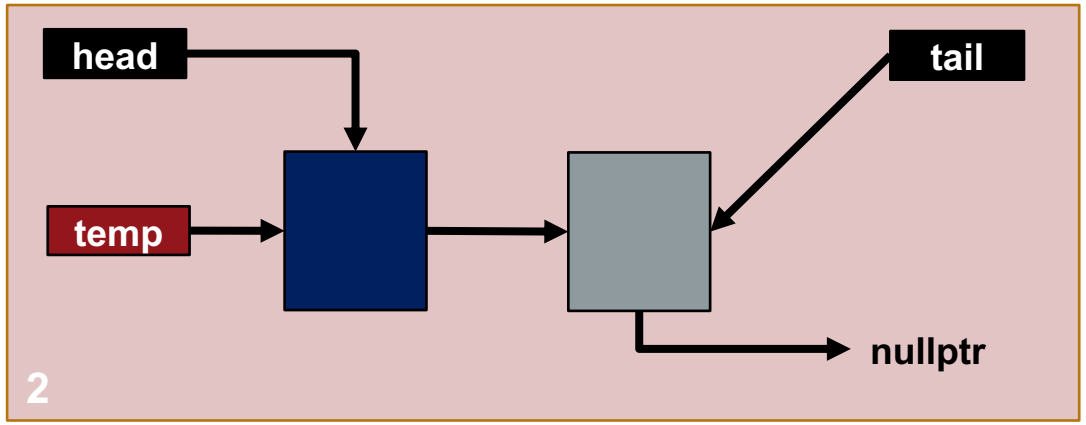
```
void pop_front() {
    if(head == nullptr) { return; }    // list empty: nothing to remove
    Node *temp = head;                 // set temp pointer to head
    head = head->next;                 // point head to second node in list
    delete temp;                       // delete the original head
    --size;                            // decrement size
}
```

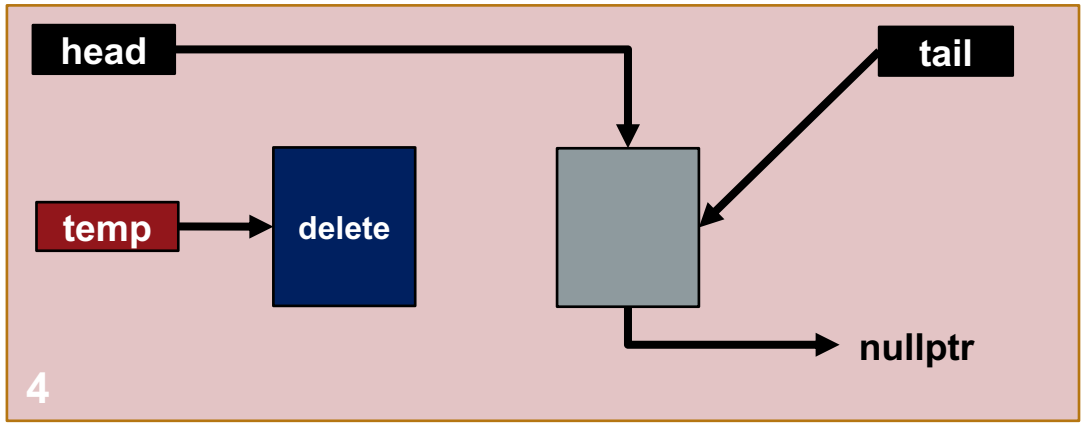
Singly Linked List: Pop_Front

Case 2: non-empty list









Singly Linked List: Pop_Back

Concept remove the last node in a list: O(n)

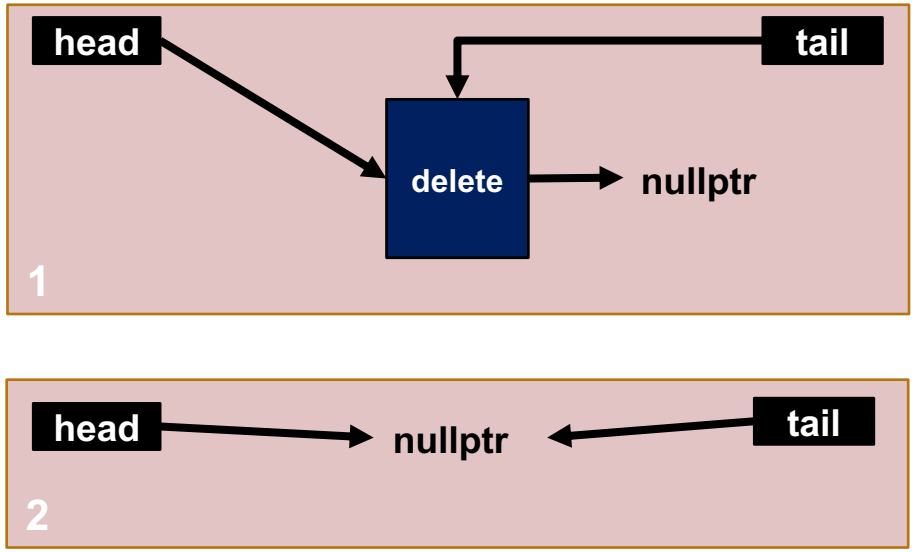
Example

```
void pop_back() {
    if(head == nullptr) { return; }          // list empty: end function
    if(head == tail) {                       // list has one node:
        delete head;                         //     delete the node
        head = tail = nullptr;               //     point head and tail to nullptr
        return;                              //     exit function
    }
    Node *prev = head;                       // set prev to point to the first node
    Node *current = head->next;              // set current to point to second node
    while(current >next != nullptr) {        // traverse until current is last, prev is second to last node
        prev = prev->next;                   //     advance prev (repetitively)
        current = current->next;             //     advance current (repetitively)

    }
    prev ->next = nullptr;                   // point second to last node to nullptr
    tail = prev;                             // point tail to new last node
    delete current;                          // delete the original last node
    --size;                                  // decrement size
}
```

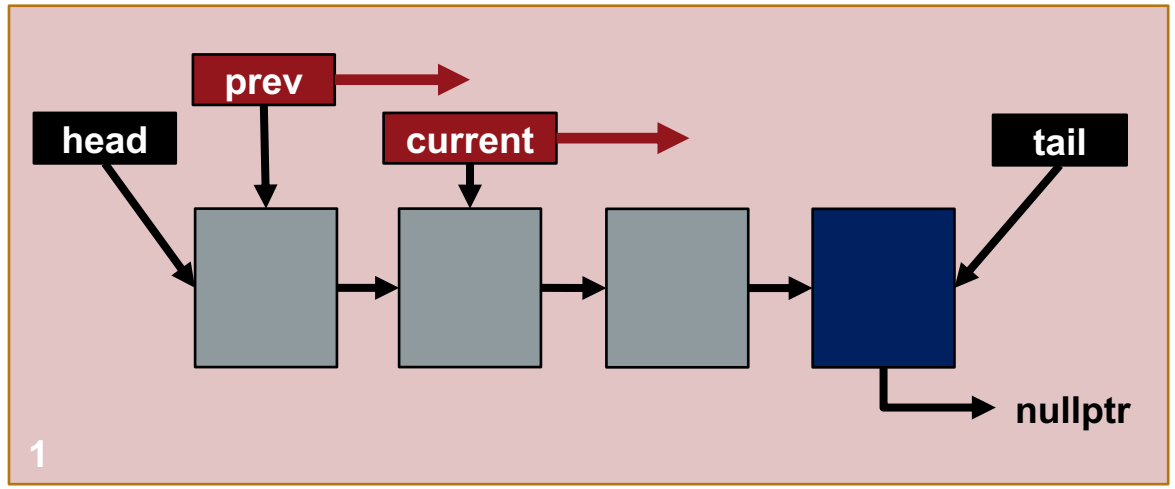
Singly Linked List: Pop_Back

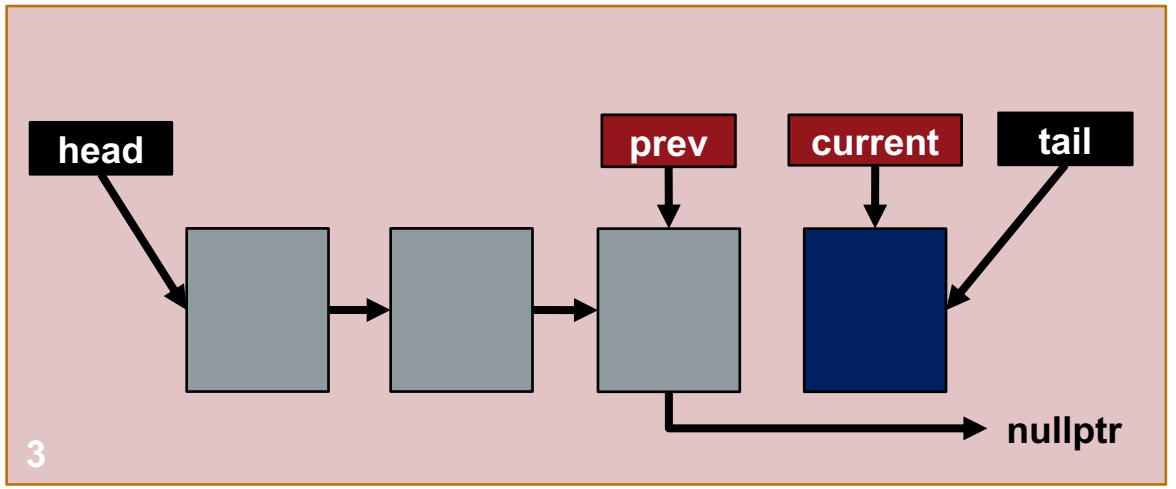
Case 2: remove the last node in a one node list

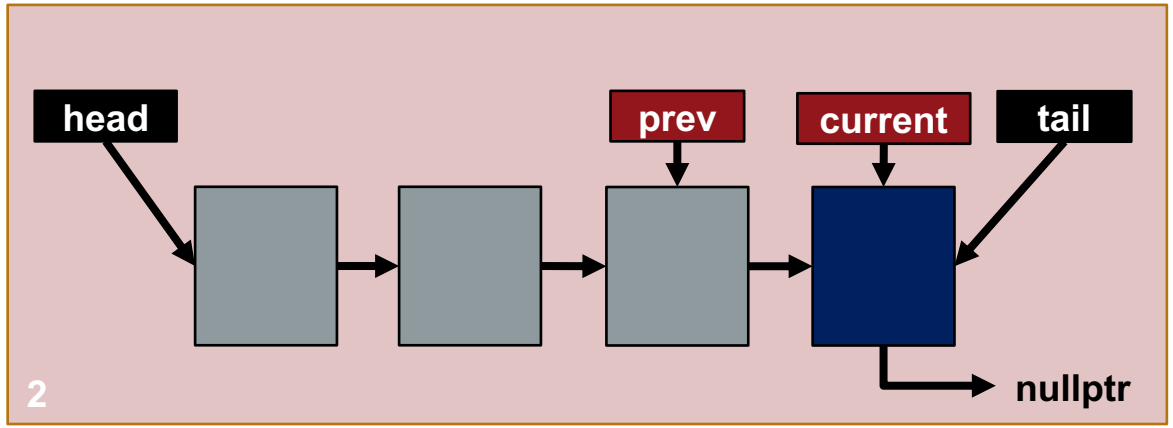


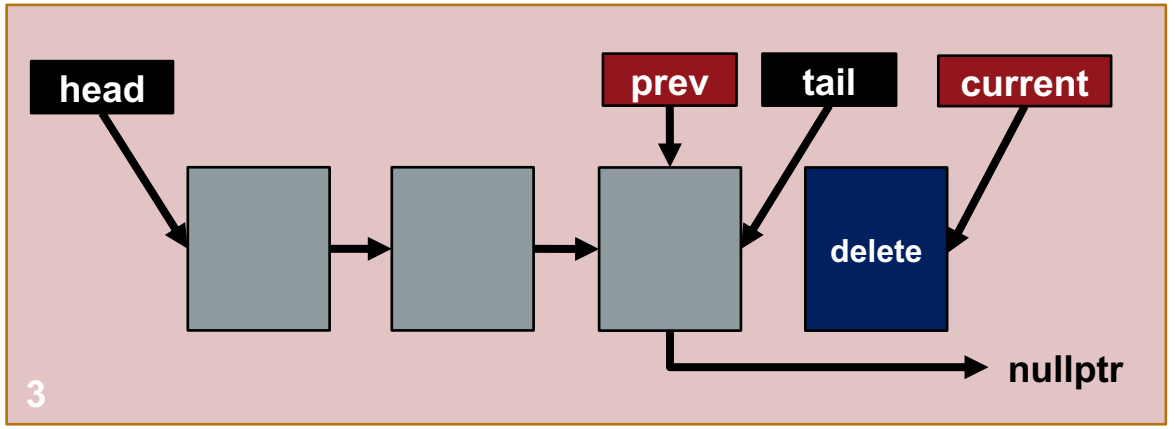
Singly Linked List: Pop_Back

Case 3: remove the last node from a list of multiple nodes









Singly Linked List: Insert

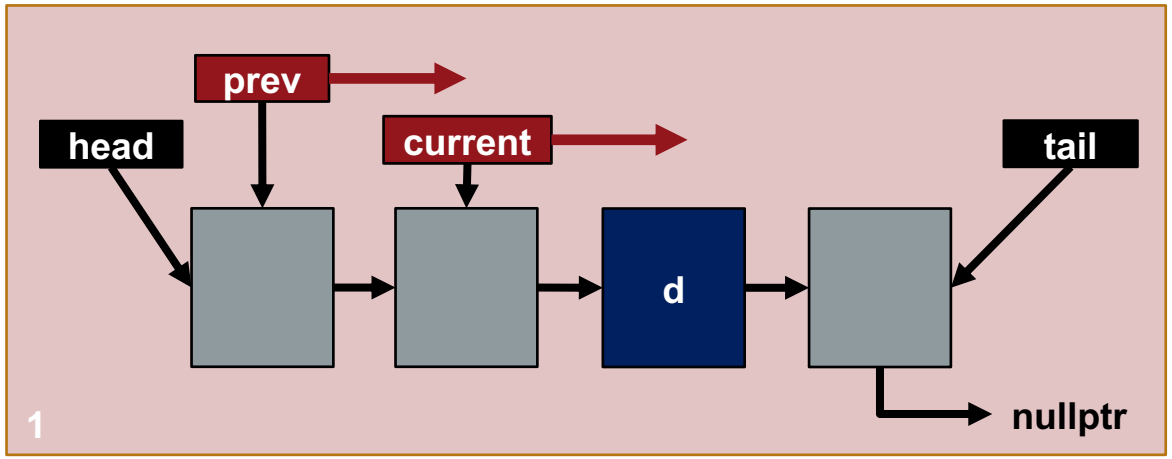
Concept insert new node before node containing data d: O(n)

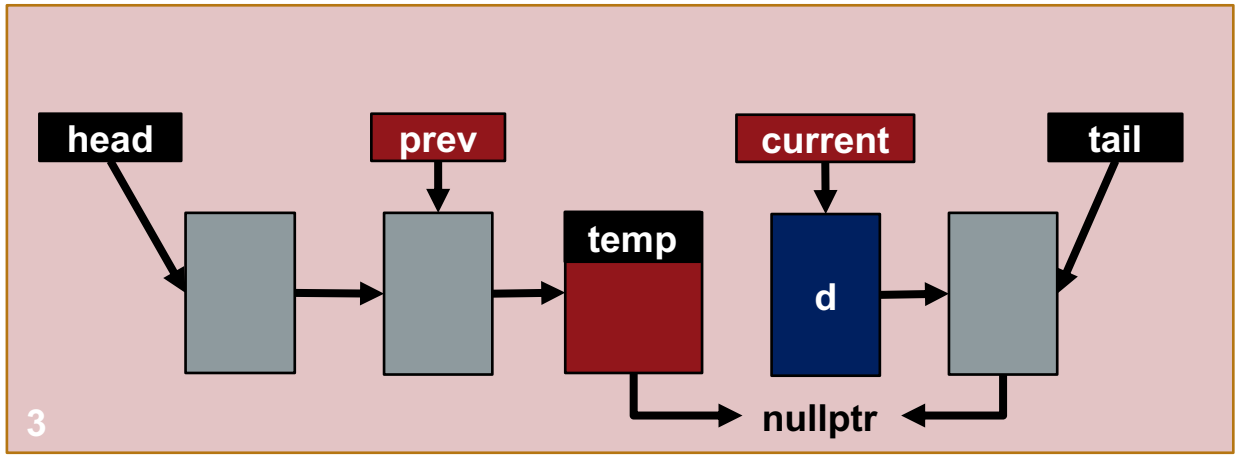
Example

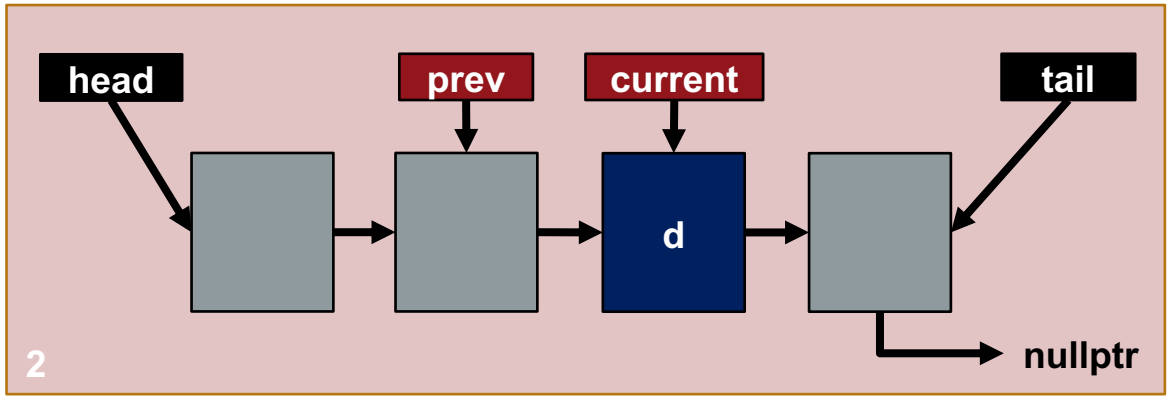
```
void insert(int d, int n) {
        if(head == nullptr) { return; }                  // insert node before node with data d
        if(head->data == d) {                            // list empty: end function
                push_front(n);                           // if the first node contains d
                return;                                  //      insert element at front of list
        }                                                //      exit function
        Node *prev = head;                               // set previous pointer to first node
        Node *current = head->next;                      // set current pointer to second node
        while(current != nullptr && current->data != d) {// traverse until end or d is found
                prev = prev->next;                       //      assign prev to node before d
                current = current >next;                 //      assign current to d
        }
        if( current != nullptr) {                        // if current is d
                Node *temp = new Node(n);                // create new node
                prev->next = temp;                       // point prev->next to temp
                temp->next = current;                    // point temp->next to d
                ++size;                                  // increment size
        }
}
```

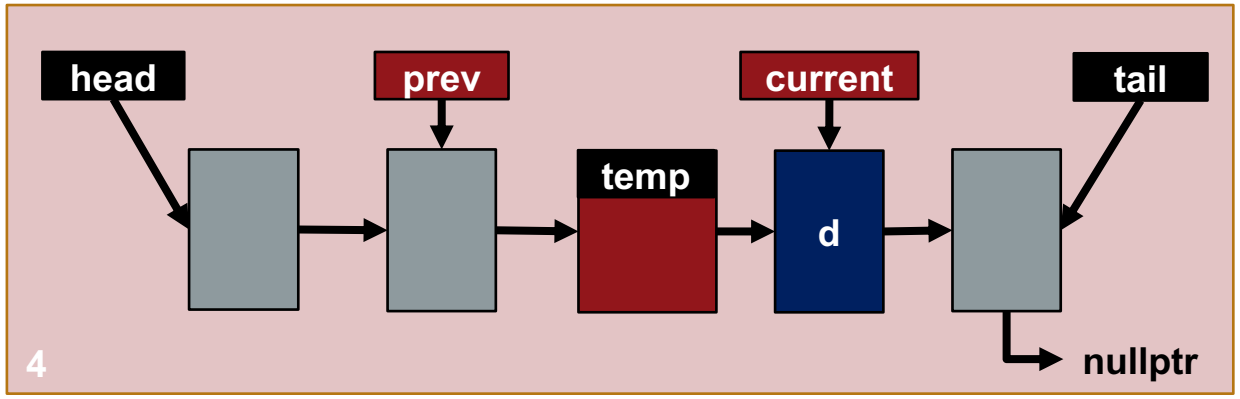
Singly Linked List: Insert

Case 3: insert a node before the node containing d in the middle of a list of multiple nodes









Singly Linked List: Erase

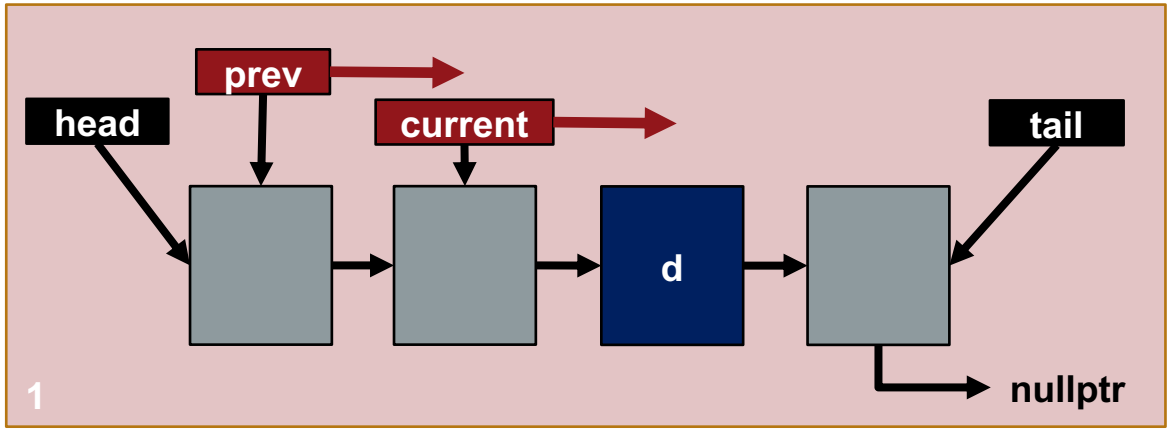
Concept erase the node containing data d: O(n)

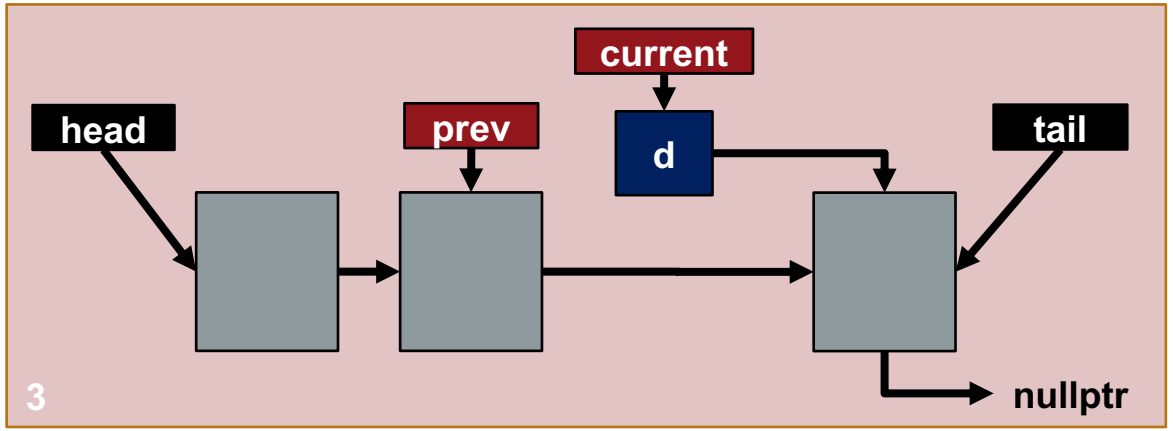
Example

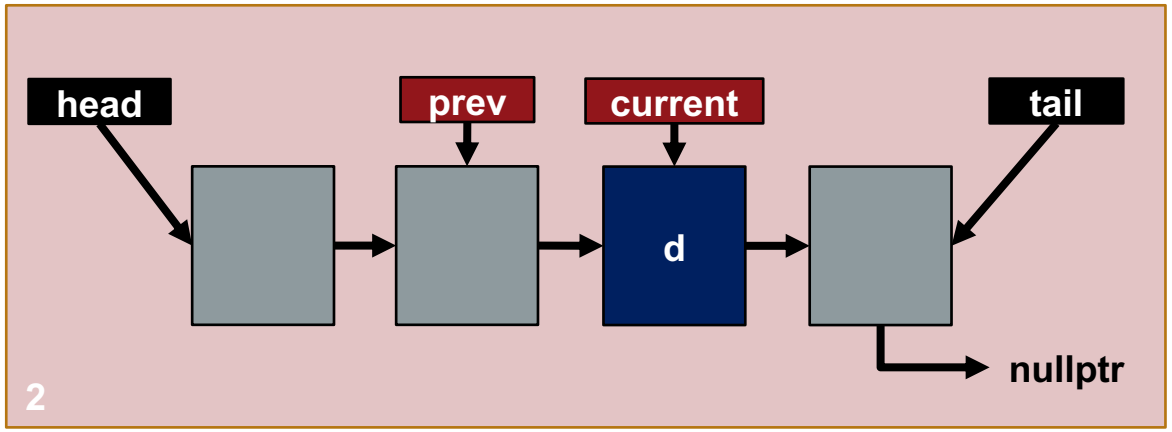
```
void erase(int d) {
        if(head == nullptr) { return; }                 // erase node which contains data d
        if(head->data == d) {                           // list empty: end function
                pop_front();                            // first node contains d
                return;                                 //      remove first node
        }                                               //      exit function
        if(tail->data == d) {                           // last node contains d
                pop_back();                             //      remove last node
                return;                                 //      exit function
        }
        Node *prev = head;                              // set previous pointer to first node
        Node *current = head->next;                     // set current pointer to second node
        while(current != nullptr && current->data != d) {   // traverse until end or d is found
                prev = prev->next;                      //      assign prev to node before d
                current = current >next;                //      assign current to d
        }
        if( current != nullptr) {                       // if d was found
                prev->next = current->next;             // point prev->next to current->next (skip node d)
                delete current;                         // delete d (no memory leak)
                --size;                                 // decrement size
        }
}
```

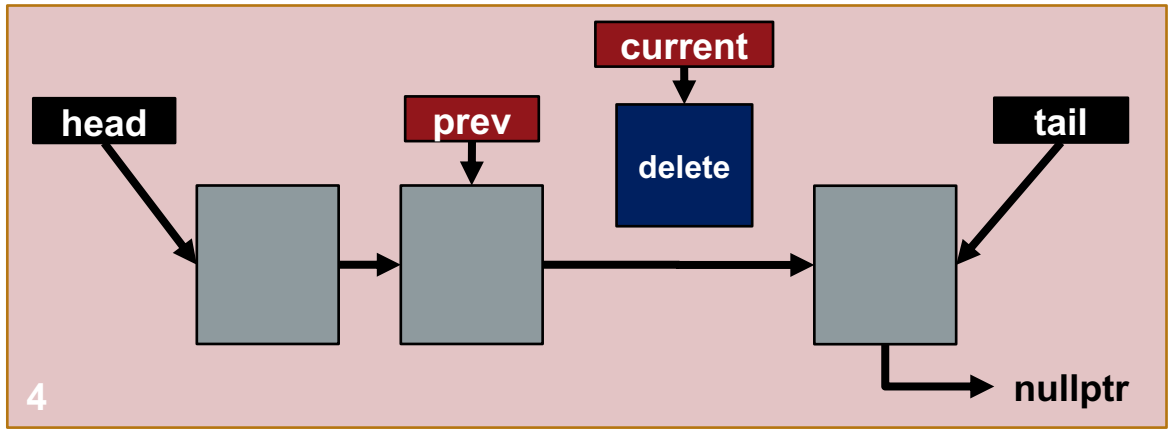
Singly Linked List: Erase

Case 3: multiple nodes where d is not the first or last node



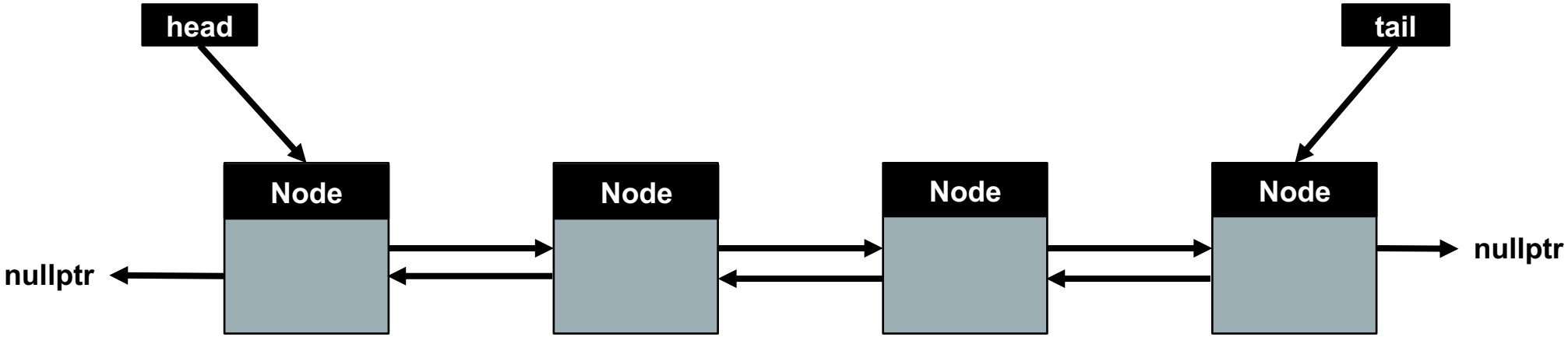






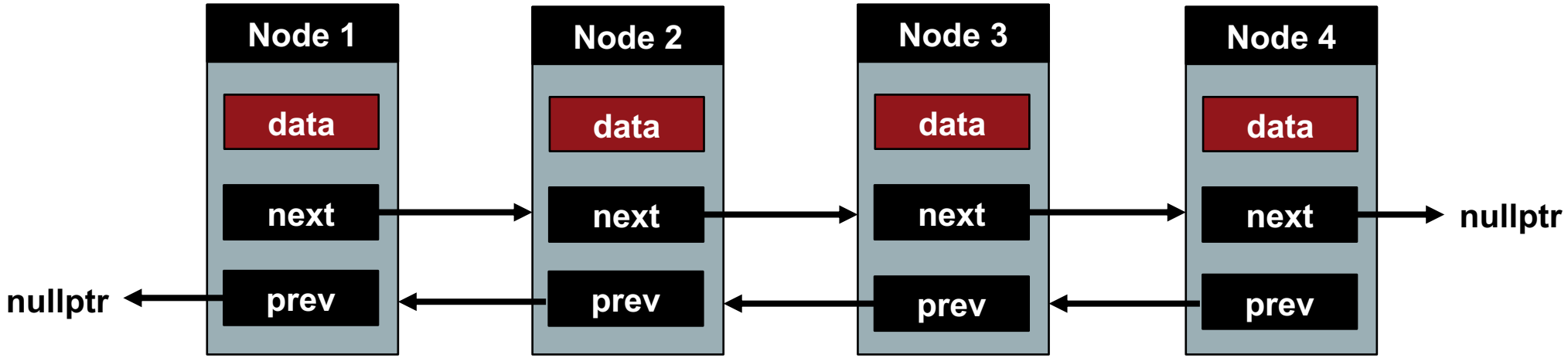
Doubly Linked List

Concept	a list of nodes where each node has two links, one to previous, one to next the last node points to nullptr
Head	a pointer to the first node in the linked list
Tail	a pointer to the last node in the linked list
Note	a doubly linked list can be traversed forwards or backwards



Doubly Linked List Node

Concept	doubly linked list nodes contain three variables, data, next and prev
Data	the information to be stored
Next	a pointer to the next node in the linked list
Prev	a pointer to the previous node in the linked list



Doubly Linked Node Class

Concept node classes are custom designed for the container they will be used with

Example a node class to store integers in a doubly linked list

```
class Node {
public:
    int data;                                   // integer data
    Node *next;                                 // pointer to next node
    Node *prev;                                 // pointer to previous node

    Node(const int &d)                          // node constructor
        : data(d), next(nullptr), prev(nullptr) { }   // next points to nullptr
                                                // prev points to nullptr
};
```

Doubly Linked List: Push_Front

Concept add a node to the front of a list: O(1)

Example

```
void push_front(int n) {
    Node *temp = new Node(n);
    if(head == nullptr) {
        tail = temp;
    } else {
        temp->next = head;
        head-prev = temp;                    // point head to new node
    }
    head = temp;
    ++size;
}
```

Doubly Linked List: Push_Back

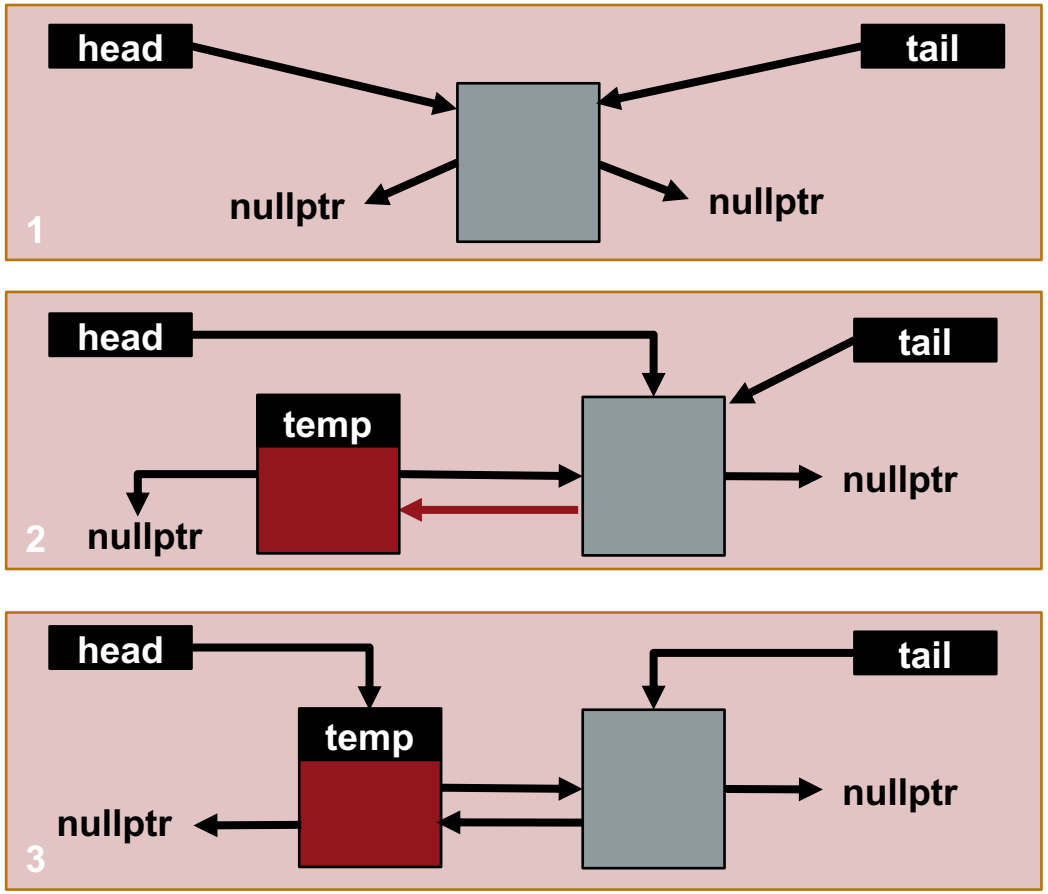
Concept add a node to the back of a list: O(1)

Example

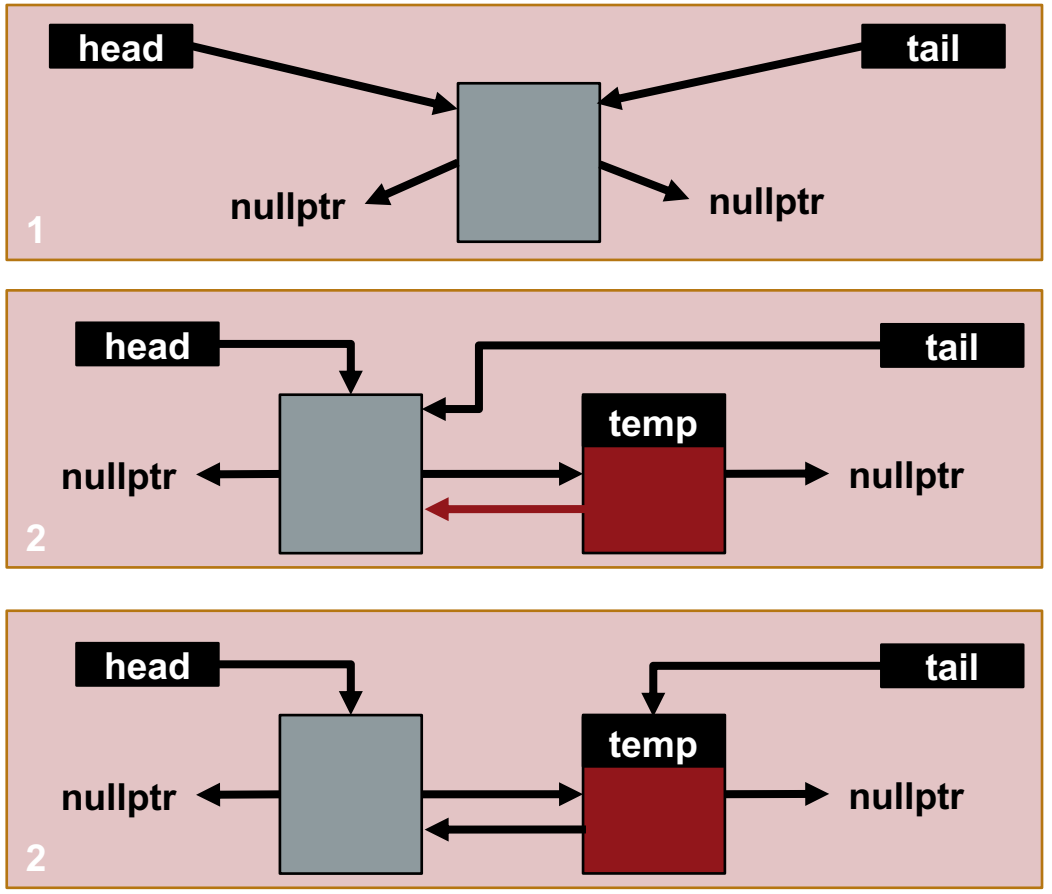
```
void push_back(int n) {
    if(head == nullptr) {
        push_front(n);
        return;
    }
    Node *temp = new Node(n);
    tail->next = temp;
    temp->prev = tail;                  // point temp->prev to last node
    tail = temp;
    ++size;
}
```

Doubly Linked List: Push_Front/Back

Case 2: add a node to the front



Case 2: add a node to the back



Doubly Linked List: Pop_Front

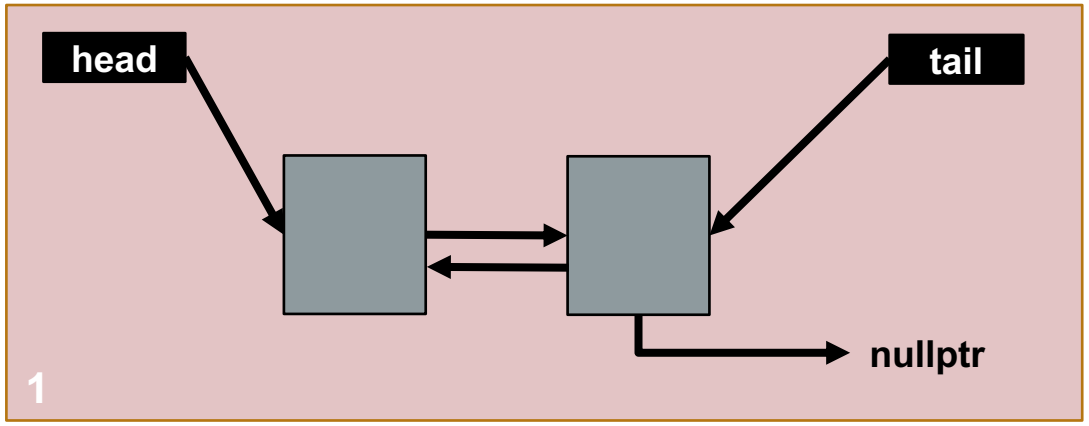
Concept remove the first node in the list: O(1)

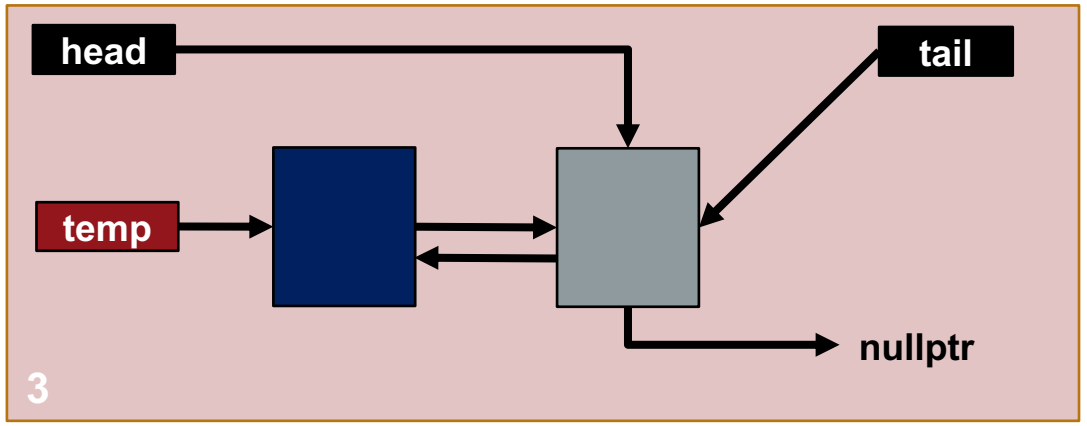
Example

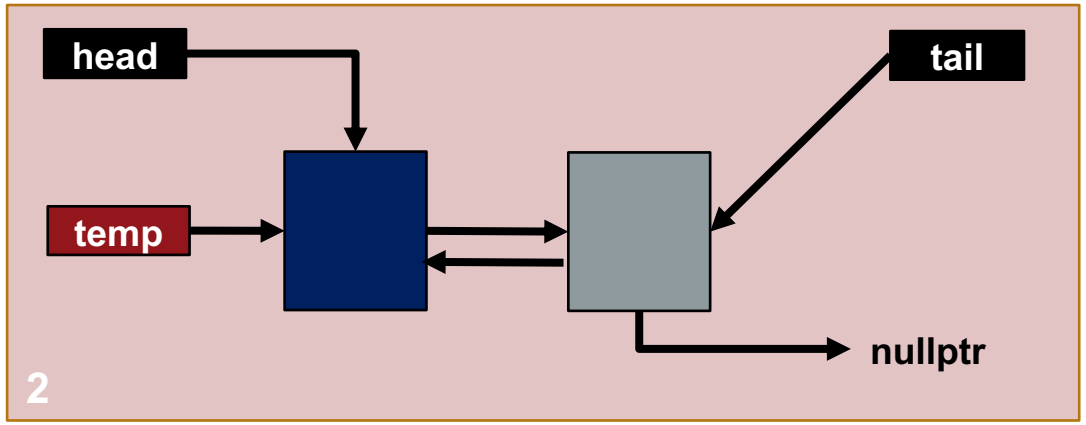
```
void pop_front() {
    if(head == nullptr) { return; }
    Node *temp = head;
    head = head->next;
    delete temp;
    if(head != nullptr) {          // if at least one node still exists:
        head->prev = nullptr       //     point head->prev to nullptr
    }
    --size;
}
```

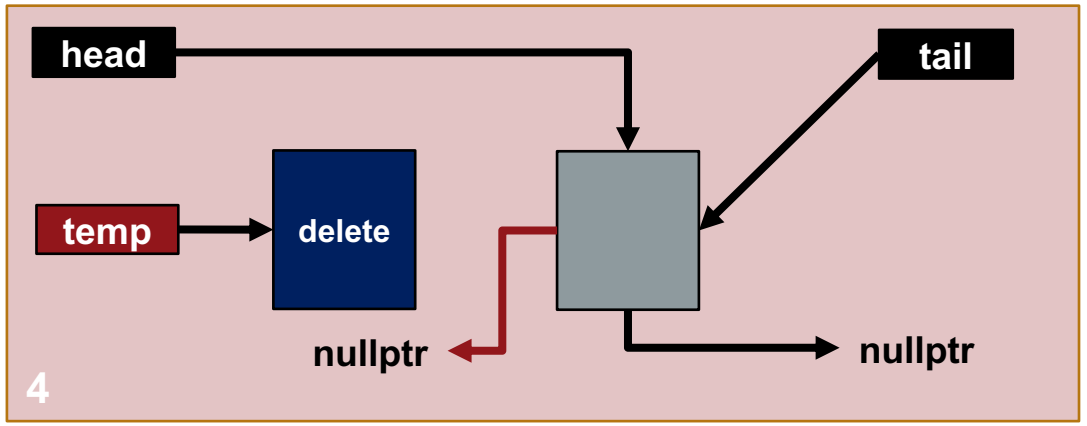
Doubly Linked List: Pop_Front

Case 2: remove the front node of a non-empty list









Doubly Linked List: Pop_Back

Concept remove the last node in the list

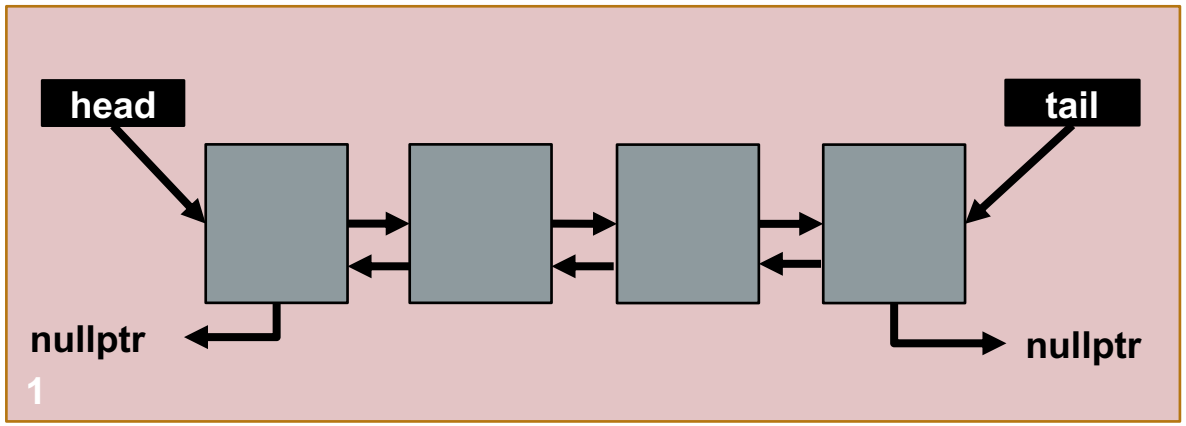
significantly faster algorithm then singly linked list: O(1) versus O(n)

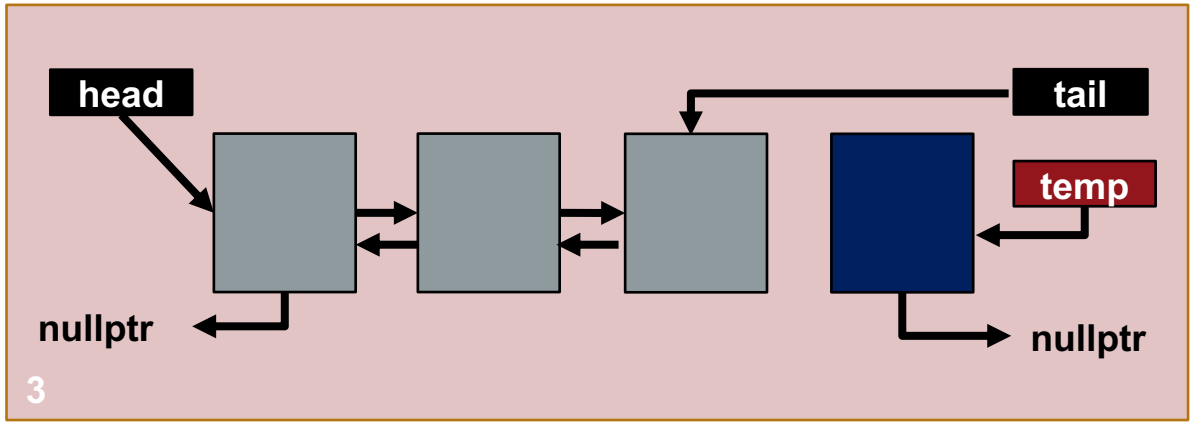
Example

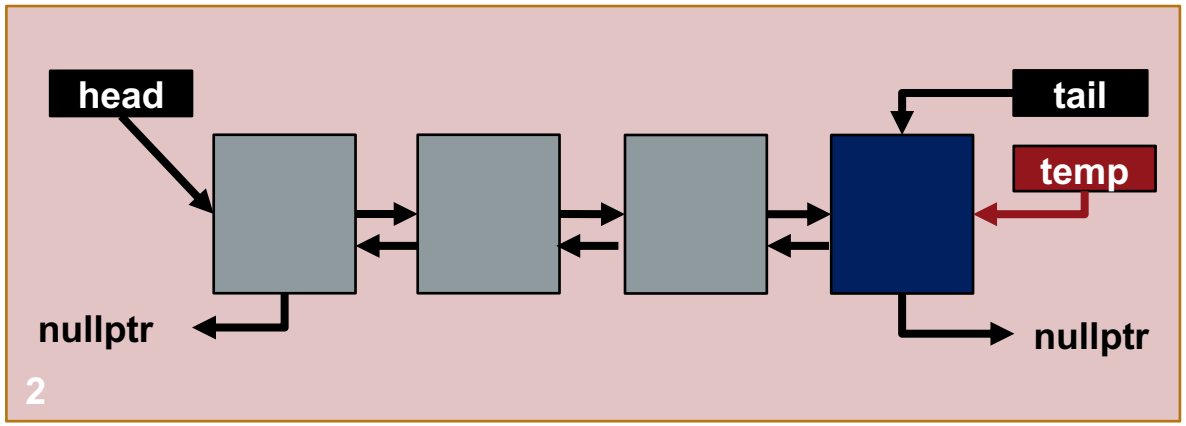
```
void pop_back() {
    if(head == nullptr) { return; }
    if(head == tail) {
        delete head;
        head = tail = nullptr;
        return;
    }
    Node *temp = tail;              // point temp to tail
    tail = tail->prev;              // point tail to second to last node
    tail->next = nullptr;           // point new tail.next to nullptr
    delete temp;                    // delete original tail
    --size;                         // decrement size
}
```

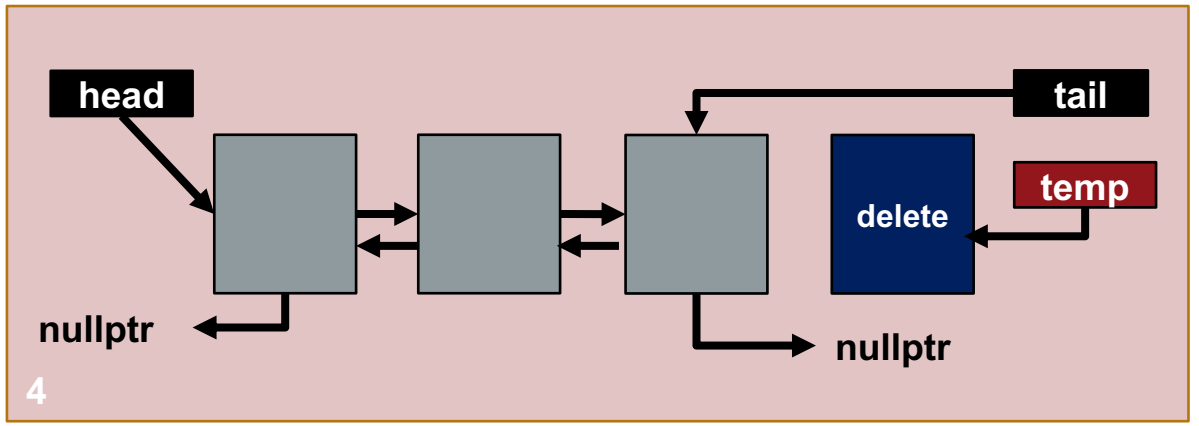
Doubly Linked List: Pop_Back

Case 2: remove the last node from a list of multiple nodes









Doubly Linked List: Insert

Concept

insert new node before node containing data d given a node pointer

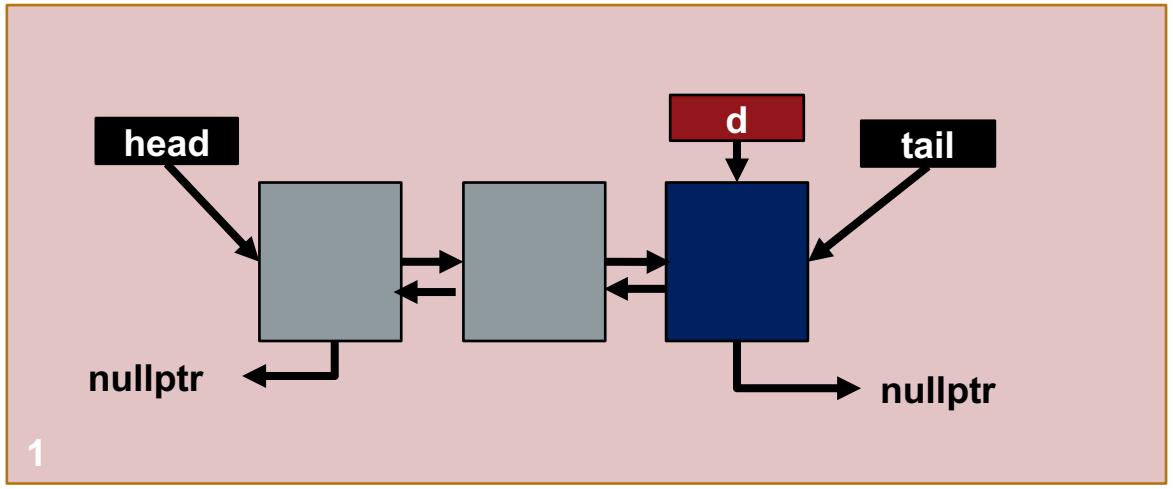
alternative O(1) insert function for doubly linked lists
this assumes that code provides access to node pointers

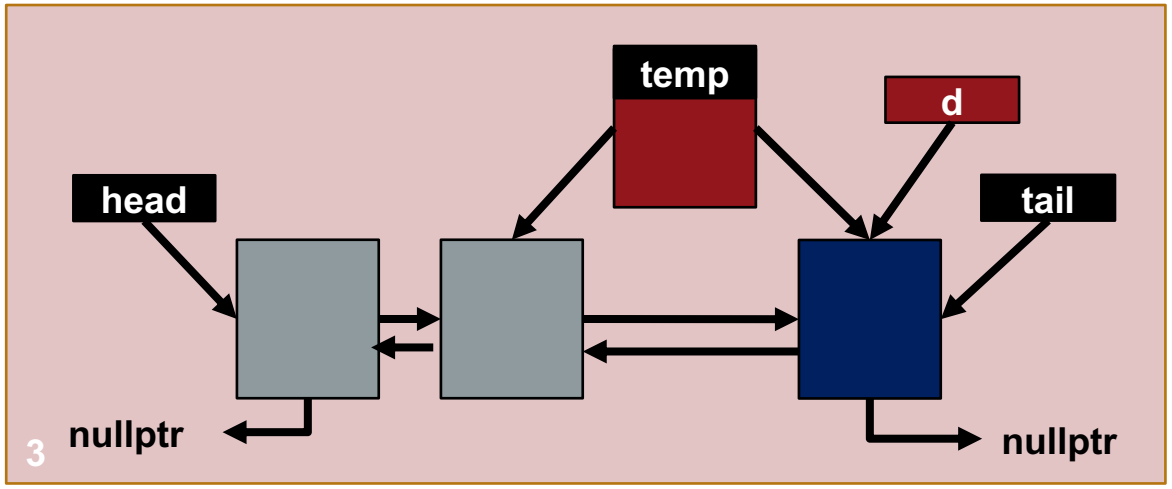
Example

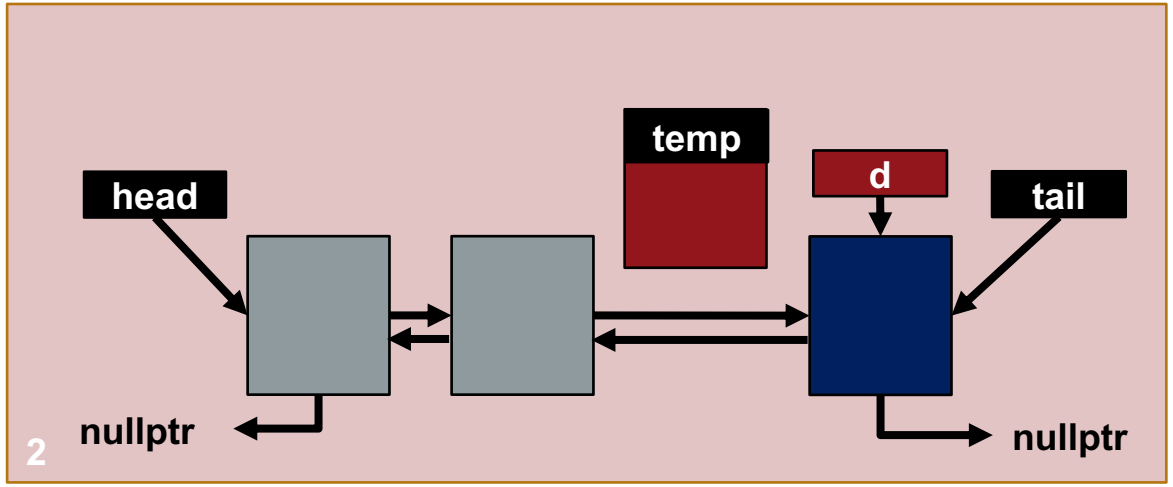
```
void insert(Node *d, int n) {          // insert node before d, given pointer to d
    if(head == nullptr) { return; }    // d does not exist, exit function
    if(head->data == d->data) {        // if d is the head node:
        push_front(n);                 //     push front
        return;                        //     exit function
    }
    Node *temp = new Node(n);          // create node to insert
    temp->next = d;                    // point temp->next to node d
    temp->prev = d->prev;              // point temp->prev to previous node
    d->prev->next = temp;              // point previous->next to temp
    d->prev = temp;                    // point d->prev to temp
}
```

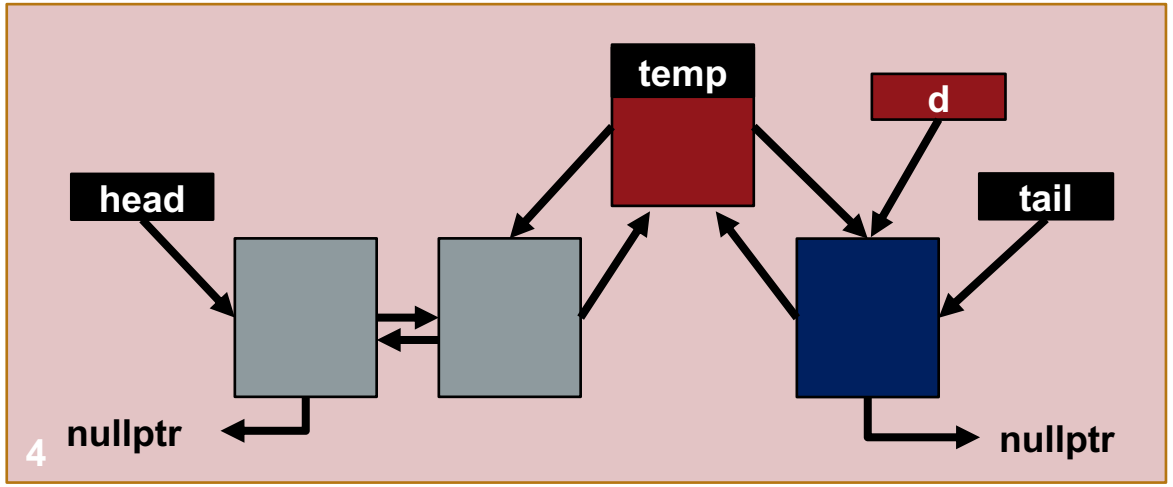
Doubly Linked List: Insert

Case 2: insert before the d node









Doubly Linked List: Erase

Concept

erase node before node containing data d given a node pointer

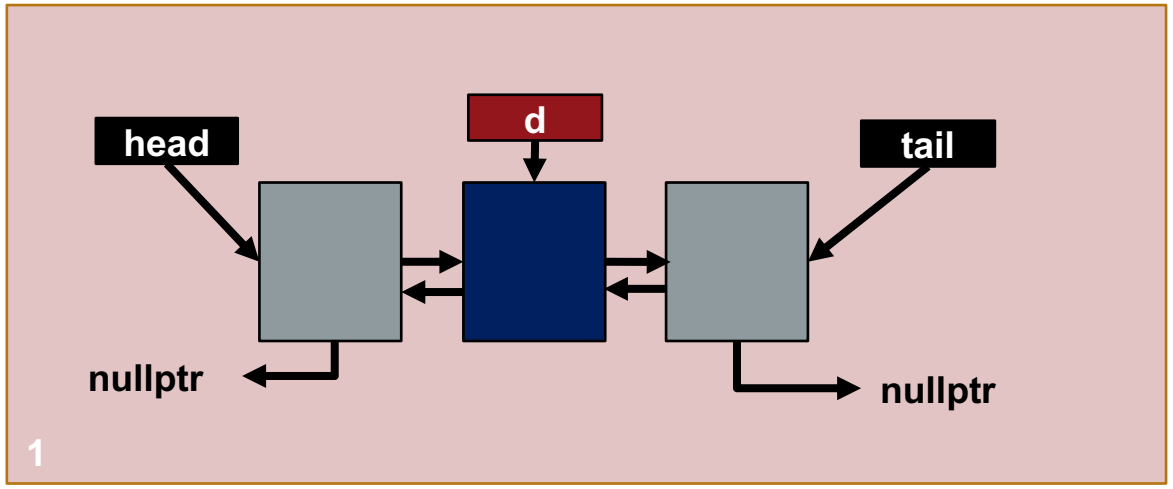
alternative O(1) erase function for doubly linked lists
this assumes that code provides access to node pointers

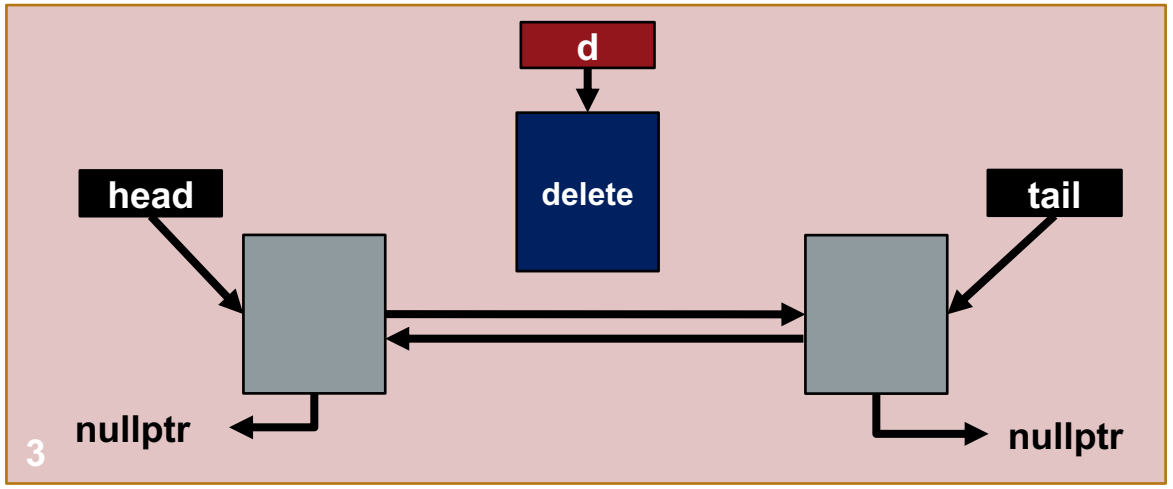
Example

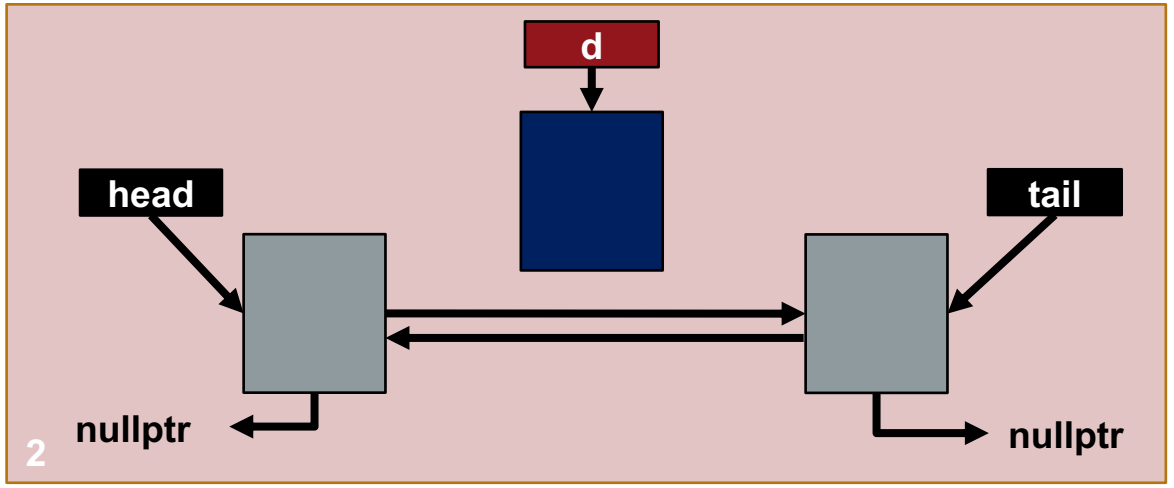
```
void erase(Node *d) {                    // erase node d
    if(head->data == d->data) {          // d is the head node:
        pop_front(n);                    //    push_front
        return;                          //    exit function
    } else if(tail->data = d->data) {    // d is tail node
        pop_back();                      //    pop_back
        return;                          //    exit function
    }
    d->prev->next = d->next;             // point previous to next
    d->next->prev = d->prev;             // point next to previous
    delete d;                            // delete d
}
```

Doubly Linked List: Erase

Case 2: erase node d in the middle of the list







Singly vs Doubly Linked List

Singly

less memory per node
only forward traversal is possible

pop_back function slower since it is O(n)
insert and delete function slower for known positions since time complexity is O(n)

Doubly

more memory per node
forward and reverse traversal is possible

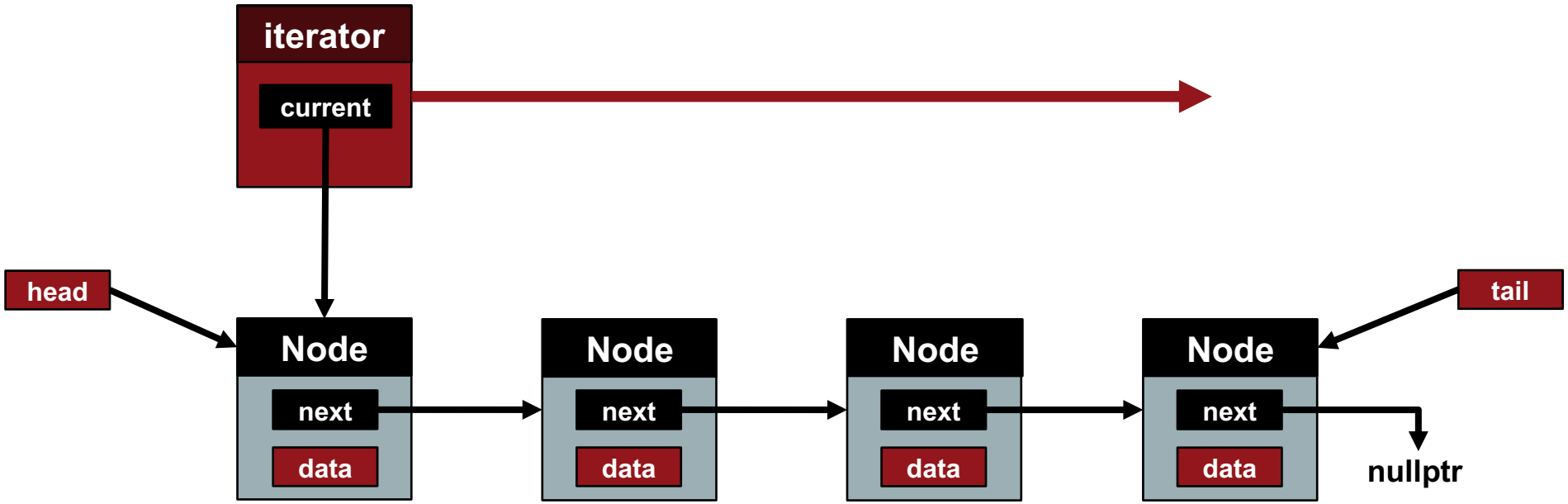
pop_back function faster since it is O(1)
insert and delete function faster for known positions since time complexity is O(1)

Introduction to Iterator Classes

Concept a class that is used to traverse through a container class
iterator objects are used to access data from within a container class

iterators replace print and overloaded << functions for containers

iterators can be coded to traverse forwards or reverse and be const or not const



Iterator List Class Functions

Example

```
class List {
    ListIterator& begin();          // return a ListIterator that points to the first node
}

class ListIterator {
    Node *current;                  // pointer to the current node in the list
    ListIterator(list *list);       // construct an iterator for a linked list object
    bool hasNext();                 // return true if more nodes exist in the list
    ListIterator operator++();      // advance the iterator to the next object (prefix)
    ListIterator operator++(int);   // advance the iterator to the next object (postfix)
    int& operator*() const;         // overloaded * operator to access list objects
}
```

Iterator List Class Functions

Example

```
class List {
    Node *head;                                  // first node in the list
    ListIterator& begin( ) {                     // return a ListIterator
                                                 // pointing to the first node
        return *( new ListIterator(this) );      // call ListIterator constructor with this list
    }
};

class ListIterator {
    Node *current;
    ListIterator(list *list)                     // construct iterator for a linked list
        : current( list->head) {                 // set current to the first list node
    }
};
```

Iterator List Class Functions

Example

```
class ListIterator {
    bool hasNext() {
        return current != nullptr;          // return true if not at the end of a list
    }
    ListIterator& operator++() {            // prefix move iterator forward in list
        current = current->next;            // advance iterator to next object in list
        return *this;                       // return updated iterator
    }
    ListIterator& operator++(int) {         // postfix move iterator forward in list
        ListIterator temp = *this;          // copy the calling object
        current = current->next;            // advance iterator to next object in list
        return temp;                        // return the non-advanced object copy
    }
    int& operator*() {                      // * overload to access data
        return current->data;               // return data at iterator object location
    }
};
```

ex5_iterators.cpp

Template Containers

Problem

to dereference or not to dereference

should functions provide return nodes or data

should insert/erase functions deference nodes