

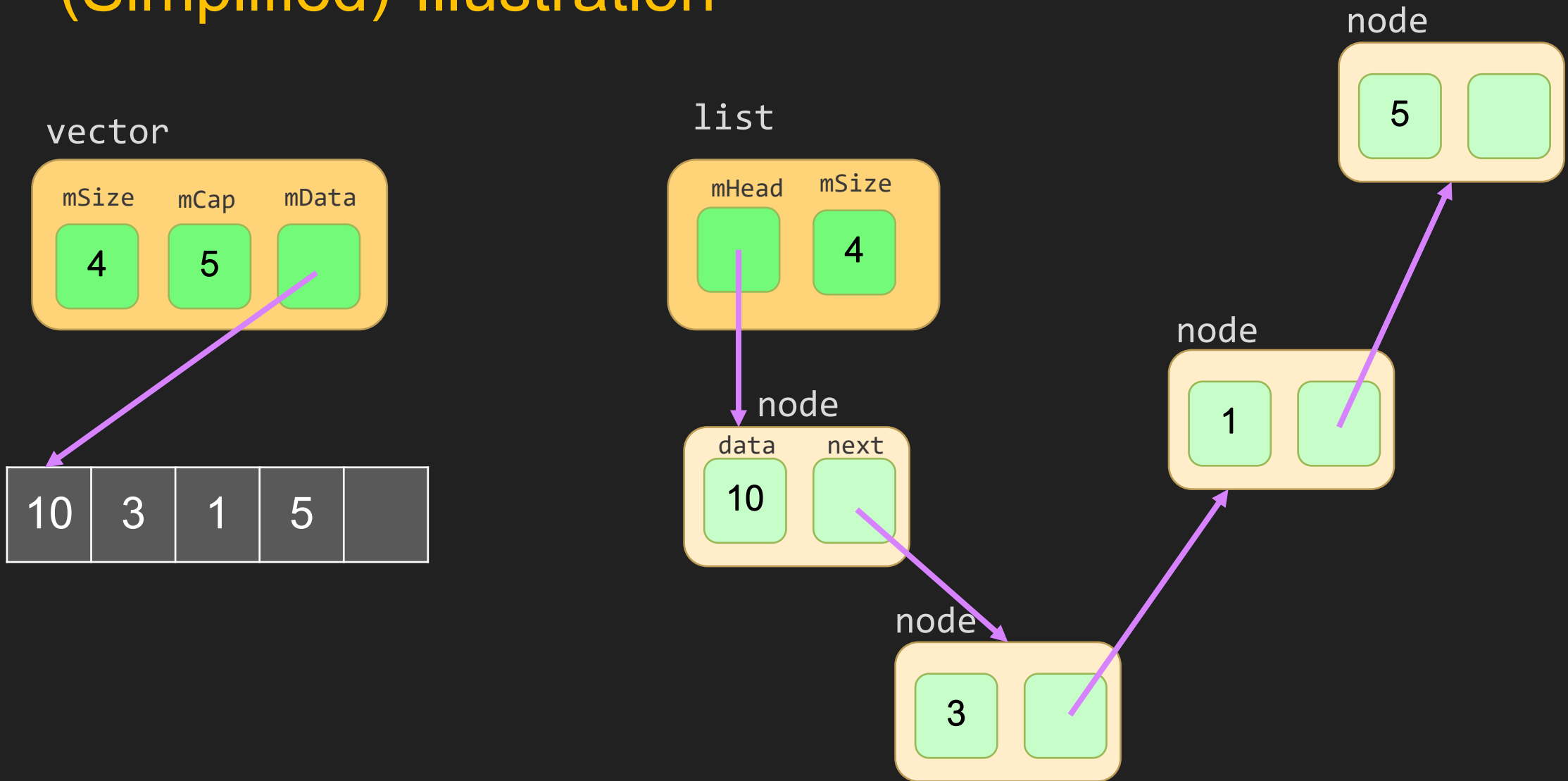
Linked List

Faster Insert & Erase but slower access

Overview

- Vector takes $O(n)$ in insert/erase at position specified by an iterator
 - But it is very fast to access any member, $O(1)$
- List gives better performance on insert/erase as $O(1)$, providing that we have iterator to the position of insertion/erase
- Achieved by storing data into a **node** and each **node** use a **pointer** to identify the **next element**
 - Access to any elements is $O(n)$, if we don't have an iterator to that element
- Use more memory than vector

(Simplified) Illustration



List vs Vector

List

- Allocate each data separately
 - Each data points where is the next data
 - Very fast insert/erase (just change some pointer)
 - Very slow access because we don't know where k-th element is

Vector

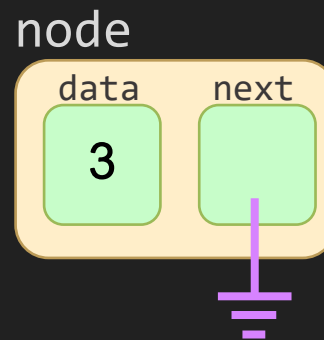
- Allocate data as a consecutive block
 - Very fast to access any element
 - Very slow Insert/erase requires every element after point of insertion

v0.1 node

- Simple object that stores a **data** and a **link to another node**
- **NULL** is a special value for any pointer that points to nowhere
 - Draw as a ground

```
template <typename T>
class node {
public:
    T data;
    node *next;
    node() :
        data( T() ), next( NULL ) { }

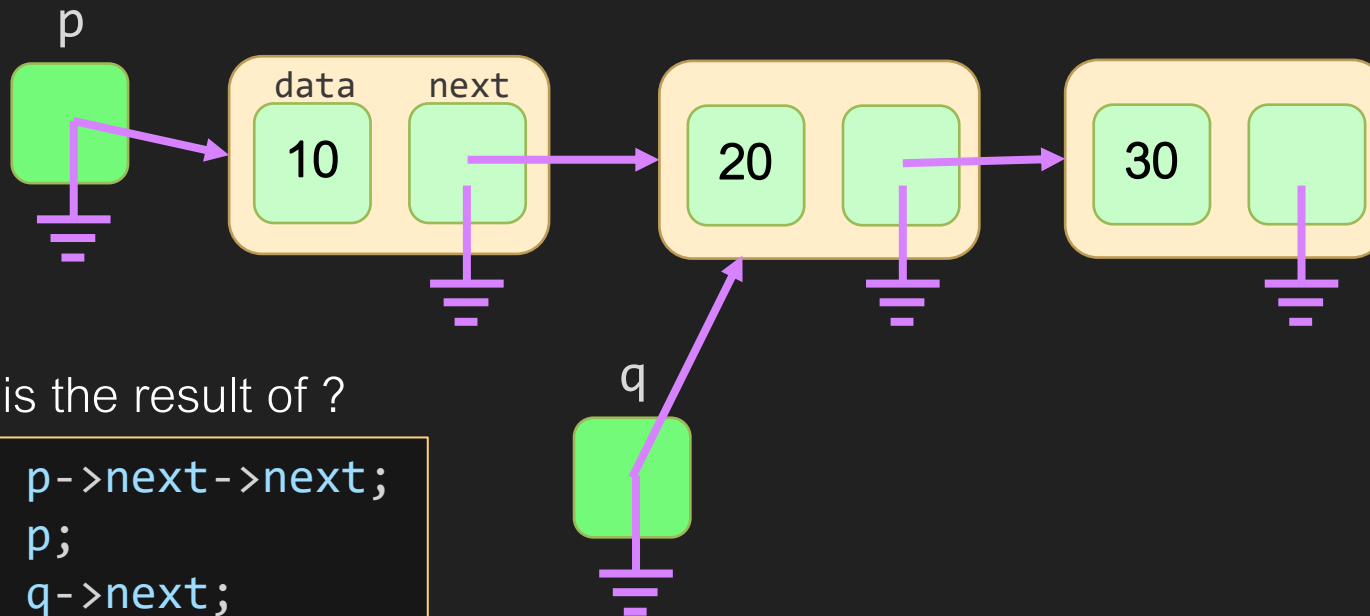
    node(const T& data, node* next) :
        data ( T(data) ), next( next ) { }
};
```



Pointer to Node

- Working with linked list needs a pointer to a node

```
int main() {  
    CP::node<int> *p = NULL;  
    p = new CP::node<int>(10,NULL);  
    CP::node<int> *q;  
    q = new CP::node<int>(20,NULL);  
    p->next = q;  
    q->next = new CP::node<int>(30,NULL);  
}
```



What is the result of ?

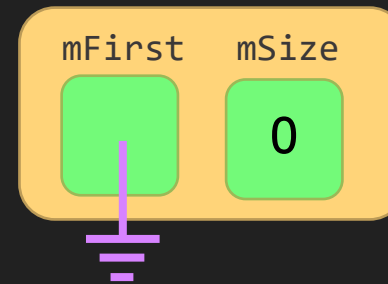
```
p = p->next->next;  
q = p;  
q = q->next;
```

v0.1 Simple (Singly) Linked List

```
template <typename T>
class list {
protected:
    class node {
        friend class list;
    public:
        T data;
        node *next;
        node() :
            data( T() ), next( NULL ) { }

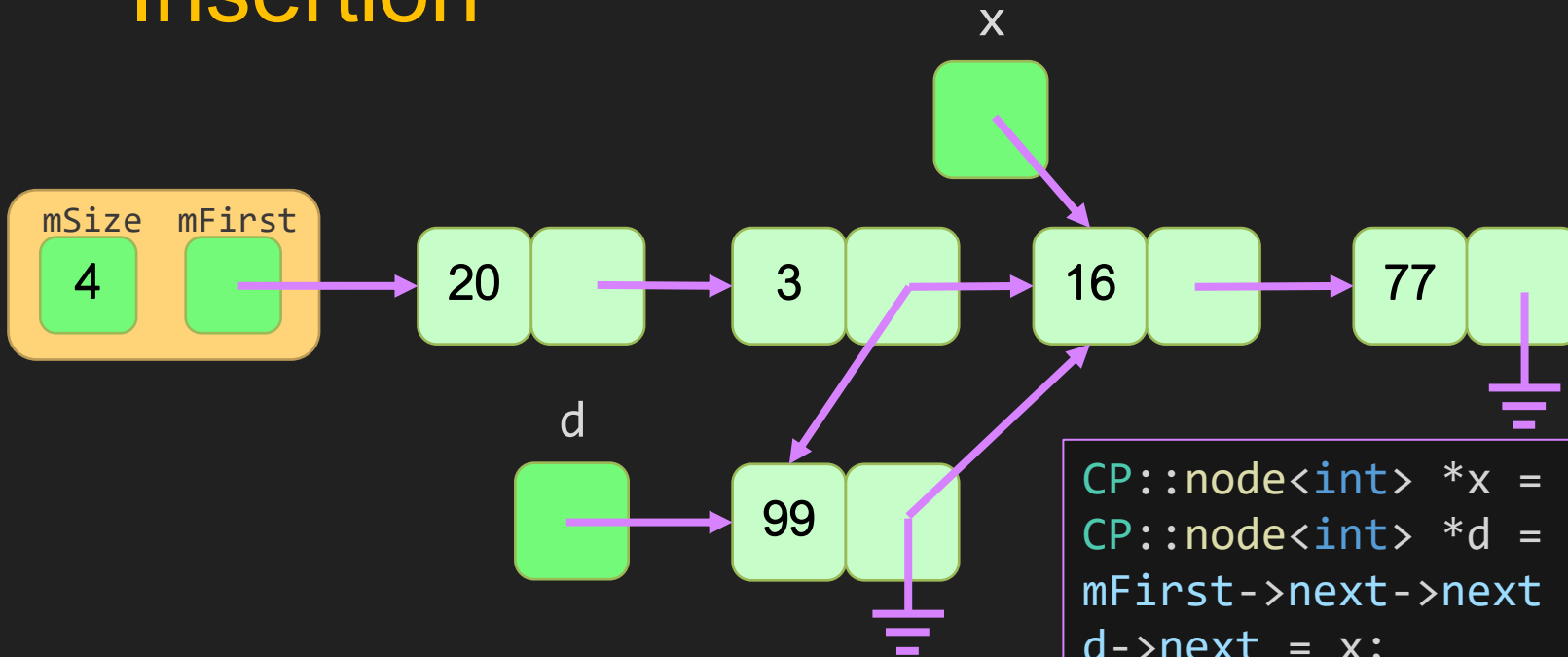
        node(const T& data, node* next) :
            data ( T(data) ), next( next ) { }
    };
protected:
    node *mFirst;
    size_t mSize;
public:
    list() : mFirst( NULL ), mSize( 0 ) { }
    ~list() { clear(); }
    //... more function
};
```

- Node is an inner class



For clarity, node will be drawn as light green box

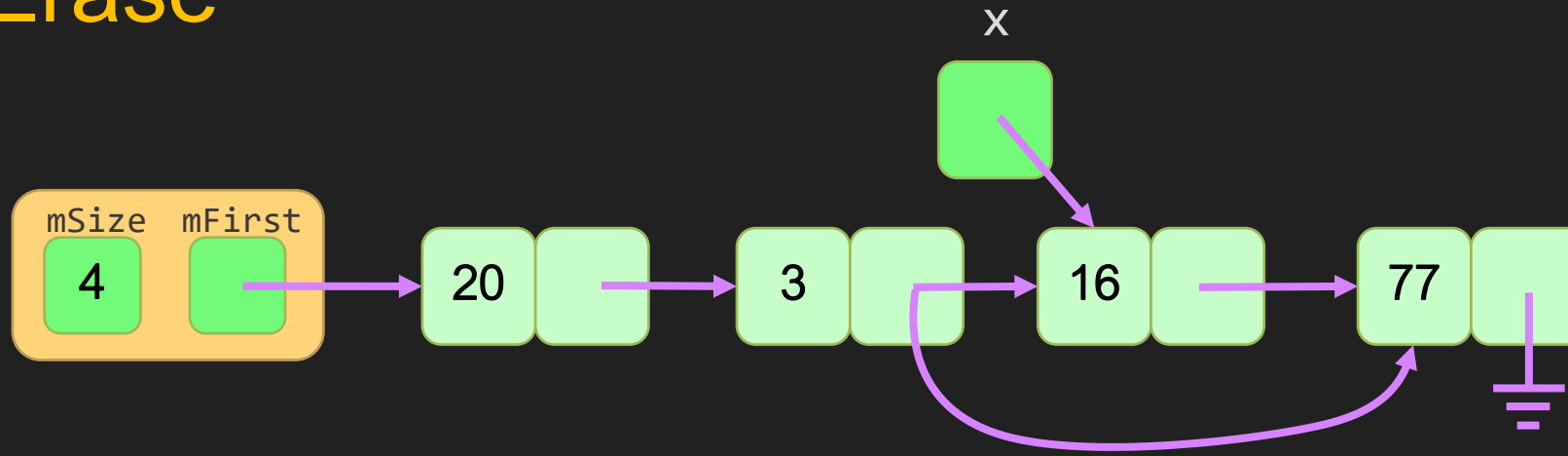
Insertion



```
CP::node<int> *x = mFirst->next->next;
CP::node<int> *d = new CP::node<int>(99,NULL);
mFirst->next->next = d;
d->next = x;
```

- To insert a value 99 before value 16, Let **x** be a pointer that point to the node of 16
 1. Create a **new node d** containing a data to be inserted
 2. Change pointer of **a node before x** (which point to x) to point to **d** instead
 3. Make pointer of **d** points to **x**

Erase



```
CP::node<int> *x = mFirst->next->next;  
mFirst->next->next = x->next;  
delete x;
```

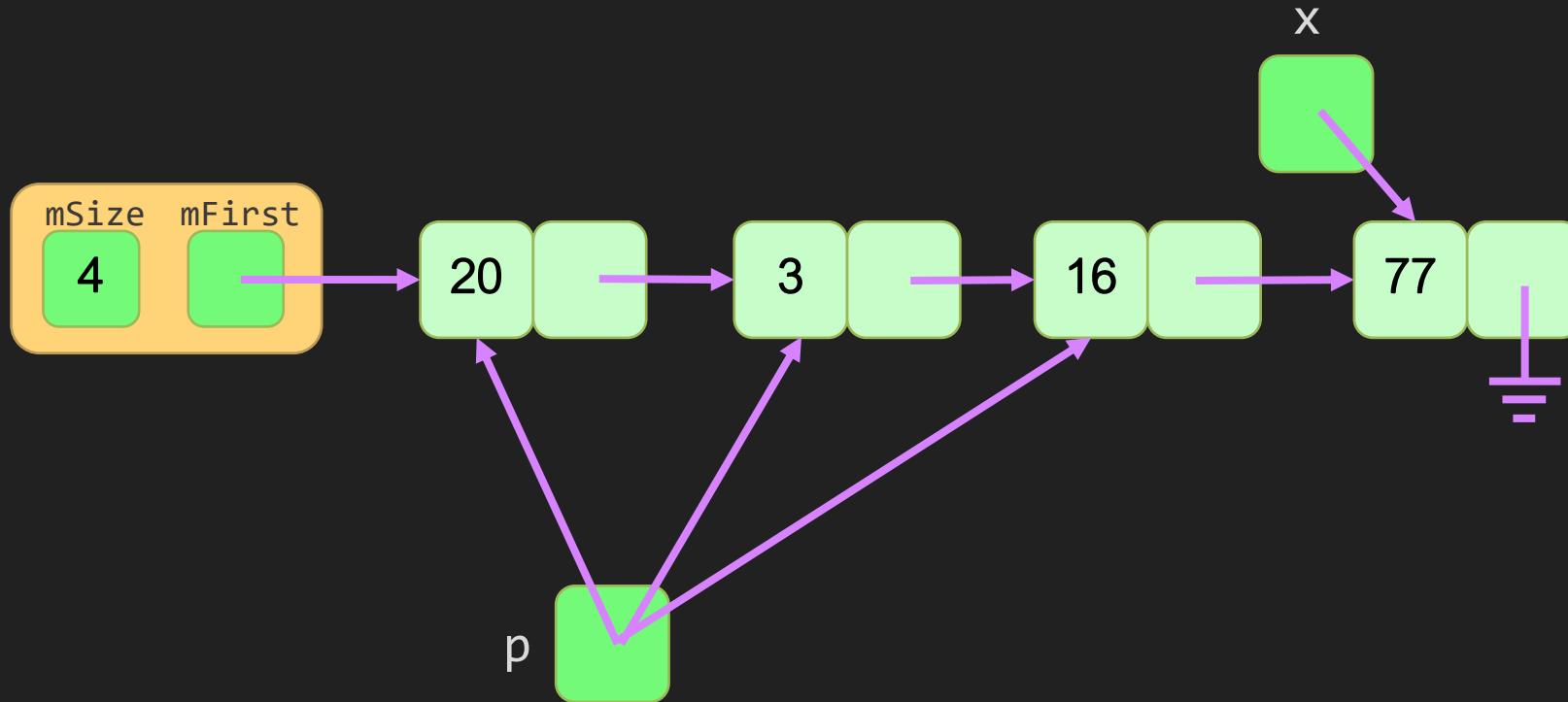
- To delete a value 16, in the node pointed by `x`
 1. Change pointer of a node before `x` (which point to `x`) to point to the node that `x` points to instead
 2. Don't forget to delete `x`

Problem with SLL

- Erase/insert at iterator X is hard
 - If we have an iterator point to X , we cannot easily go to the node before X
 - Cannot go backward
 - Need to start from $mFirst$ and move on
- Adding data to the end takes long time
 - We have to get iterator that points to the last element (which is $O(n)$)

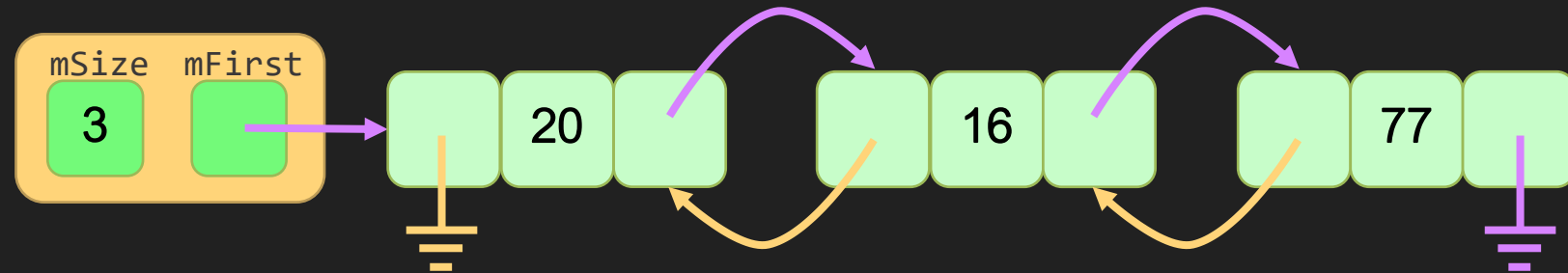
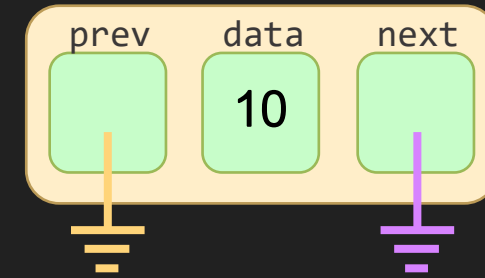
Finding node before X

```
node *p = mFirst;  
while (p != NULL && p->next != x) p = p->next;
```

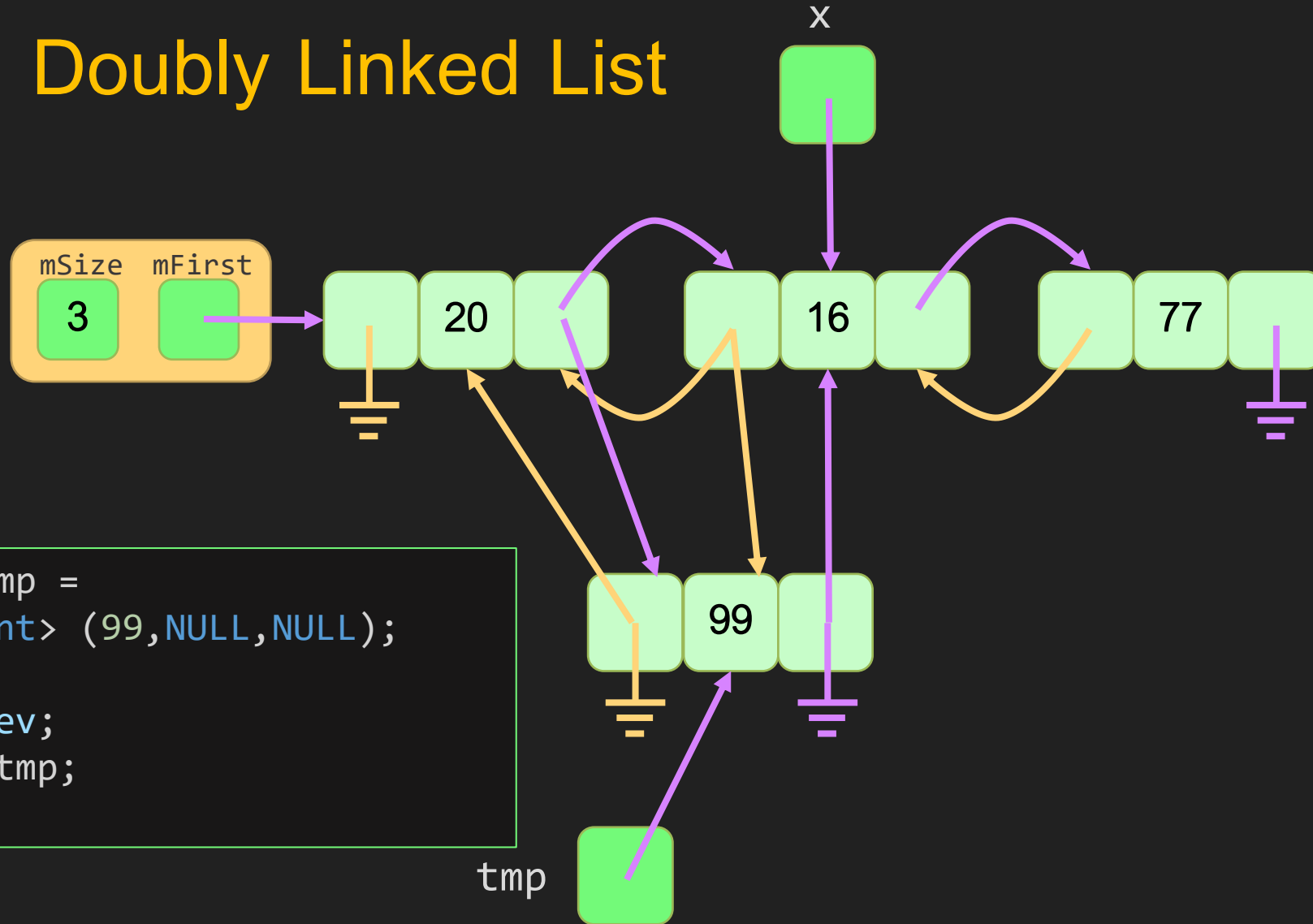


v0.2 Doubly Linked List

- Each node has 2 pointers
 - *next* and *prev*
- Can now move forward and backward
- Now, if *X* is a pointer to a node, we can easily go to *node before X* and then erase or insert

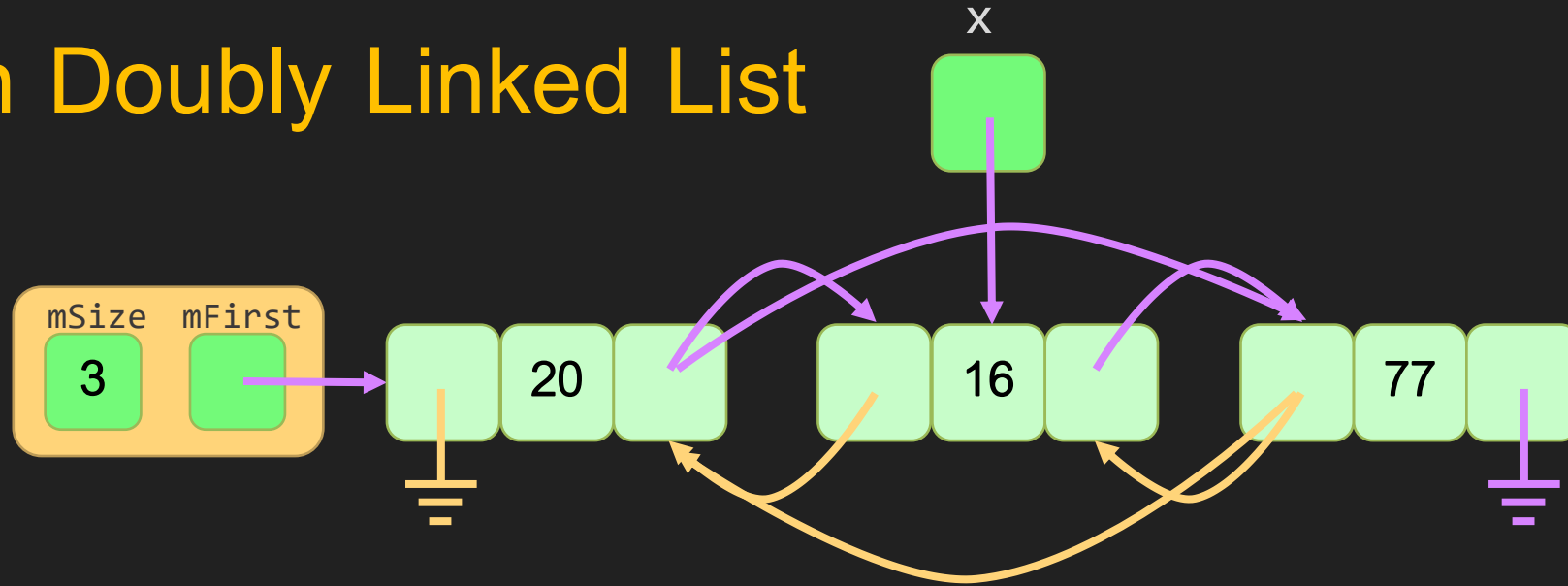


Insert in Doubly Linked List



```
CP::node<int> *tmp =  
    new CP::node<int> (99, NULL, NULL);  
tmp->next = x;  
tmp->prev = x->prev;  
x->prev->next = tmp;  
x->prev = tmp;
```

Erase in Doubly Linked List



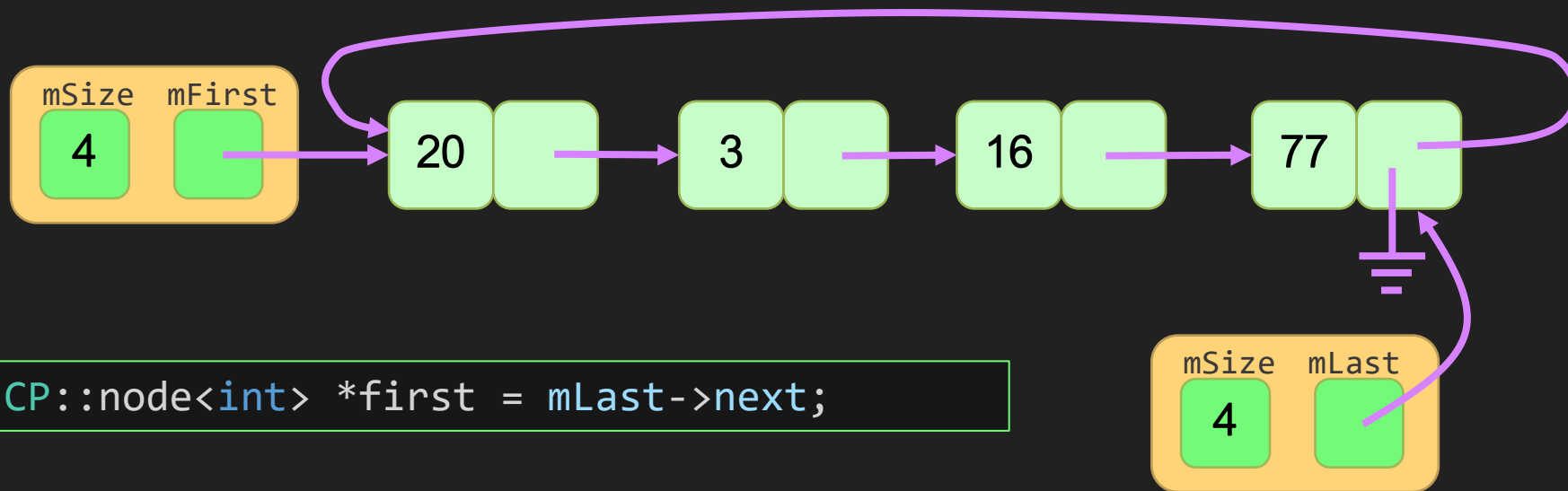
```
x->prev->next = x->next;  
x->next->prev = x->prev;  
delete x;
```

Problem solved

- Erase/insert at iterator `X` is now `easy`
- But, adding data at the end (`push_back`) is still hard
 - Need to get `X` to point to the last element
 - `push_back` is popular in real world
 - Right now we have only `push_front` (fast addition to the first)
- Also some minor issue about `code cleanliness`
 - Insert/erase the first/last node

v0.3 Circular Singly Linked List

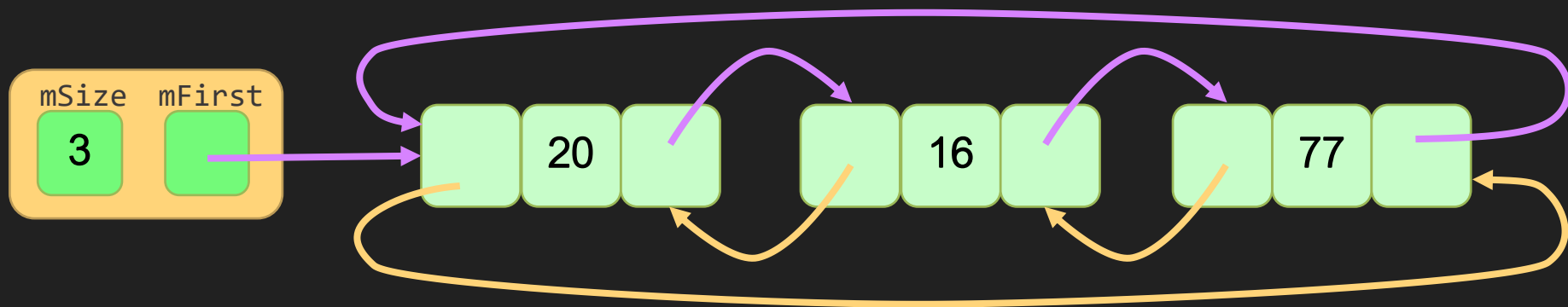
- Use `mLast` instead of `mFirst`
- Fast access to both first and last element



v0.4 Circular Doubly Linked List

- Special linking to last element
 - Fast access to both first and last element
 - Can now easily insert at the end

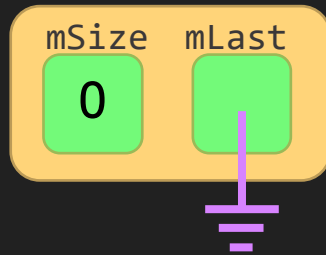
```
CP::node<int> *last = mFirst->next->prev;
```



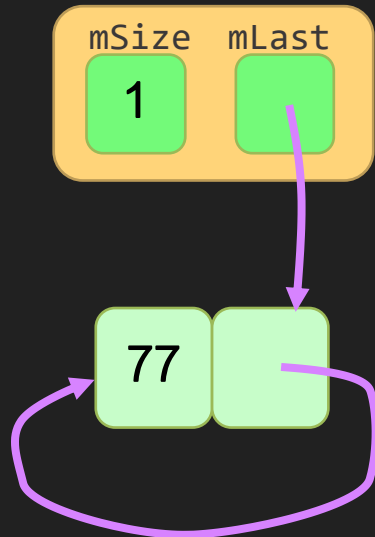
More Circular Example

Singly

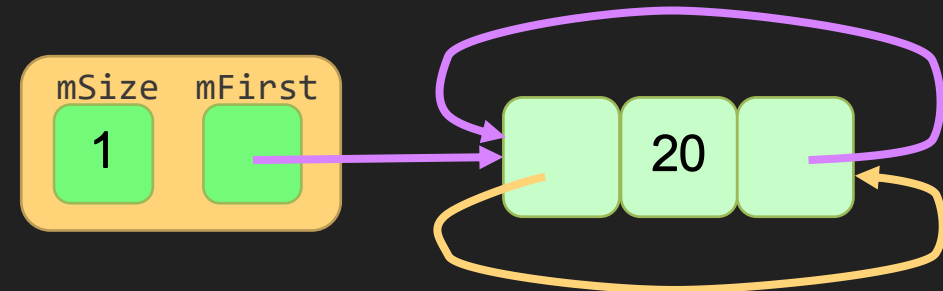
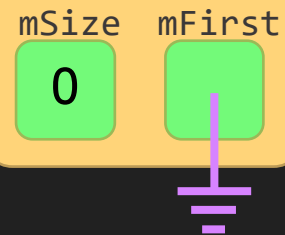
0 node



1 node

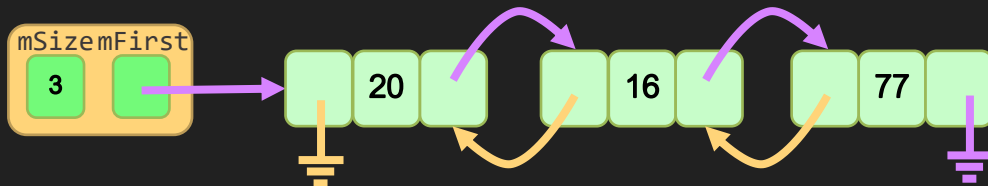


Doubly



Minor problem

- Code is **not really clean** because of the first element (or the last element)
- Consider insert at the first node and insert at the second node of Doubly Linked List
 - Assume we have a pointer to that node



```
// assume that s points to the second node
CP::node<int> *tmp =
    new CP::node<int>(99,NULL,NULL);
tmp->next = s;
tmp->prev = s->prev;
s->prev->next = tmp;
s->prev = tmp;
```

```
// assume that f points to the first node
CP::node<int> *tmp =
    new CP::node<int>(99,NULL,NULL);
tmp->next = f;
tmp->prev = f->prev;
mFirst = tmp;
f->prev = tmp;
```

Because first node is
different

Special First Node Problem

- First node (and last node) is different from other nodes in both singly or doubly linked list
- For circular singly and circular doubly, each node look the same but we also have to adjust the mFirst (or mLast)
- This affects both insert and erase
- Also need to deal when mFirst is NULL (when mSize == 0)

```
//circular doubly linked list
void push_front(const T& e) {
    if (mSize == 0) {
        mFirst = new node(e, NULL, NULL);
        mFirst->next = mFirst;
        mFirst->prev = mFirst;
    } else {
        node* tmp =
            new node(e, mFirst->prev, mFirst);
        mFirst->prev->next = tmp;
        mFirst->prev = tmp;
        mFirst = tmp;
    }
}
```

Another Example

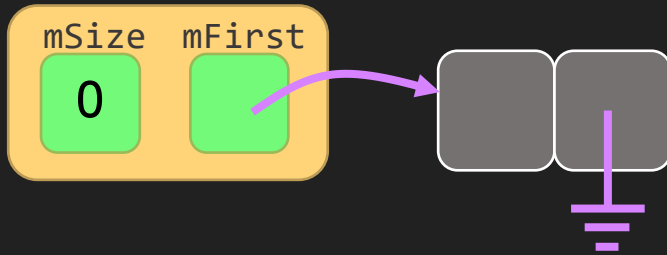
- Remove for circular doubly linked list
- Remove is find and then erase

```
//circular doubly linked list
void remove(const &T e) {
    node *p = mFirst;
    for (size_t i = 0; i < mSize; i++, p=p->next) {
        if (p->data == e) {
            p->next->prev = p->prev;
            p->prev->next = p->next;
            if (p == mFirst) {
                mFirst = p->next;
            }
            delete p;
            mSize--;
            break;
        }
    }
    if (mSize == 0) mFirst = NULL;
}
```

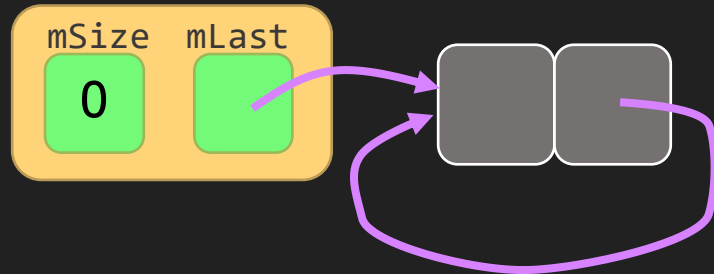
Linked List with Header

Singly

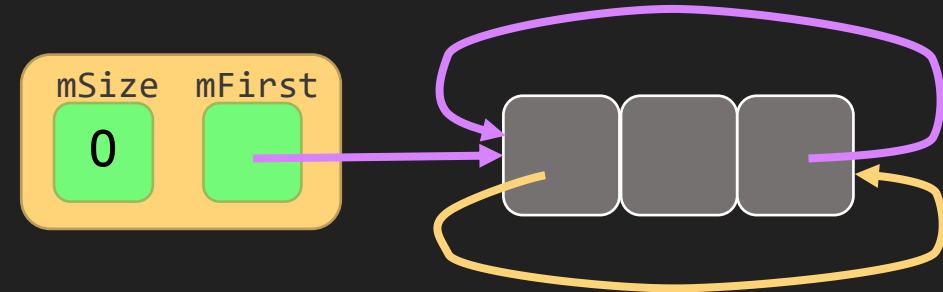
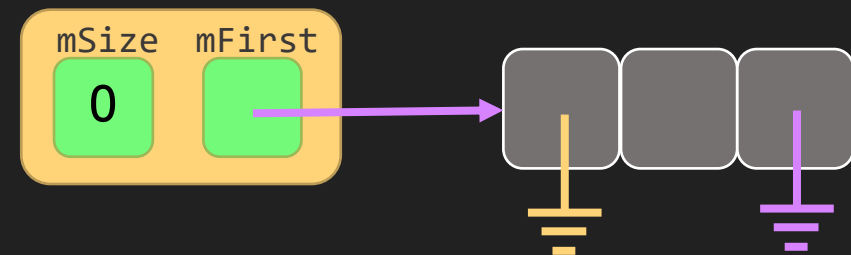
normal



circular



Doubly



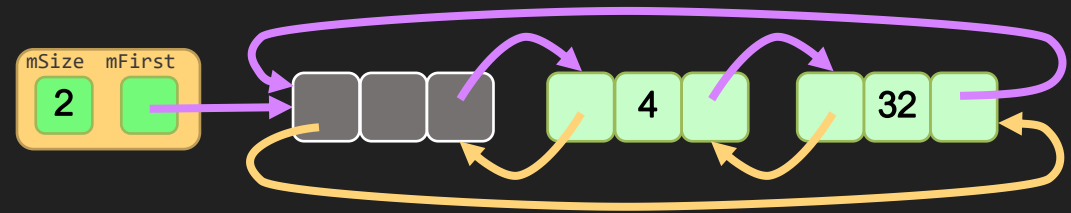
- Add a special node that will not be used to stored data

Simpler Code with header

```
void push_front(const T& e) {  
    node* f = mFirst->next;  
    node* tmp = new node(e,f,mFirst);  
    mFirst->next = tmp;  
    f->prev = tmp;  
}
```

```
void remove(const T& e) {  
    node *p = mFirst->next;  
    while (p != mHeader && p->data != e)  
        p = p->next;  
    if (p != mHeader) {  
        p->next->prev = p->prev;  
        p->prev->next = p->next;  
        mSize--;  
    }  
}
```

- Header simplify code
 - Because mFirst always points to the header (mFirst never is NULL)



Variant Summary

- Circular makes accessing first and last element fast
- Doubly makes accessing previous element fast
 - Also making erase at node p easy if we have pointer to p
 - Need more space for prev pointer
- Header makes code simpler
 - Need more space for header node

Final Version

- CP::list is “circular doubly linked list with header”
 - Simple code for insert/erase
 - Use most space (two pointers per nodes, need header nodes)
 - Can push_back, pop_back, push_front, pop_front
- Also need custom iterator class
 - Iterator just store a pointer to a node
 - We cannot directly use a pointer to a node (node*) because we need to override some operator (--, ++ and something else)

Layout

- Inner class is a class inside another class
 - Inner class can access any members of outer class
 - Outer class cannot access protected or private of the inner class
- Friend class allow other class to access

```
template <typename T>
class list {
protected:
    class node {
        friend class list;
    public:
        T data;
        node *prev, *next;
        //some functions
    };

    class list_iterator {
        friend class list;
    protected:
        node* ptr;
    public:
        //some functions && operators
    };

public:
    typedef list_iterator iterator;
protected:
    node *mHeader; // pointer to a header node
    size_t mSize;
public:
    //functions
};
```

Doubly Linked List Node

```
class node {  
    friend class list;  
public:  
    T data;  
    node *prev;  
    node *next;  
  
    node() :  
        data( T() ), prev( this ), next( this ) { }  
    node(const T& data,node* prev, node* next) :  
        data ( T(data) ), prev( prev ), next( next ) { }  
};
```

- Inner class can use template T of the outer class

Constructor

```
// default constructor
list() : mHeader( new node() ), mSize( 0 ) { }

// copy constructor
list(const list<T>& a) : mHeader( new node() ), mSize( 0 ) {
    for (auto &x : *this) {
        push_back(x);
    }
}

list<T>& operator=(list<T> other) {
    using std::swap;
    swap(this->mHeader, other.mHeader);
    swap(this->mSize, other.mSize);
    return *this;
}

~list() {
    clear();
    delete mHeader;
}
```

Small Functions

```
//----- capacity function -----  
bool empty() const { return mSize == 0; }  
size_t size() const { return mSize; }  
  
//----- access -----  
T& front() { return mHeader->next->data; }  
T& back() { return mHeader->prev->data; }  
  
//----- modifier -----  
void push_back(const T& element) {  
    insert(end(),element);  
}  
void push_front(const T& element) {  
    insert(begin(),element);  
}  
void pop_back() {  
    erase(iterator(mHeader->prev));  
}  
void pop_front() {  
    erase(begin());  
}
```

- Task is delegated to insert and erase
- Need iterator

Iterator

```
class list_iterator {  
    friend class list;  
protected:  
    node* ptr;  
public:  
  
    list_iterator() : ptr( NULL ) { }  
    list_iterator(node *a) : ptr(a) { }  
  
    list_iterator& operator++() {  
        ptr = ptr->next;  
        return (*this);  
    }  
  
    list_iterator& operator--() {  
        ptr = ptr->prev;  
        return (*this);  
    }  
}
```

- Has custom constructor that takes node pointer

Iterator

```
class list_iterator {  
    friend class list;  
protected:  
    node* ptr;  
public:  
  
    list_iterator operator++(int) {  
        list_iterator tmp(*this);  
        operator++();  
        return tmp;  
    }  
  
    list_iterator operator--(int) {  
        list_iterator tmp(*this);  
        operator--();  
        return tmp;  
    }  
}
```

- operator++() is an operator for ++it
- operator++(int) is a syntax for it++
- operator++(int) delegates to operator++()
- Same for operator--

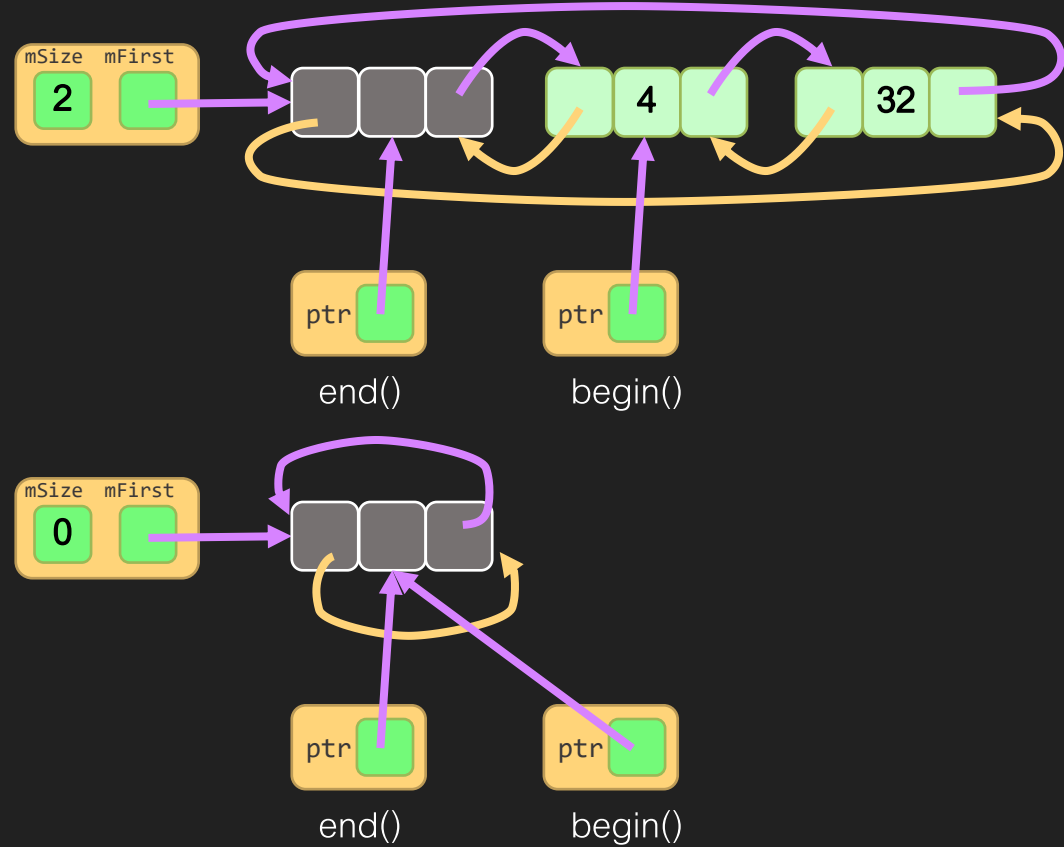
Iterator

```
class list_iterator {  
    friend class list;  
protected:  
    node* ptr;  
public:  
  
    T& operator*() { return ptr->data; }  
    T* operator->() { return &(ptr->data); }  
  
    bool operator==(const list_iterator& other) {  
        return other.ptr == ptr;  
    }  
  
    bool operator!=(const list_iterator& other) {  
        return other.ptr != ptr;  
    }  
}
```

- Nothing special here
 - Just syntax

Other small function

```
iterator begin() {  
    return iterator(mHeader->next);  
}  
  
iterator end() {  
    return iterator(mHeader);  
}  
  
void clear() {  
    while (mSize > 0) erase(begin());  
}
```



Insert & Erase

```
iterator insert(iterator it, const T& element) {  
    node *n = new node(element, it.ptr->prev, it.ptr);  
    it.ptr->prev->next = n;  
    it.ptr->prev = n;  
    mSize++;  
    return iterator(n);  
}
```

```
iterator erase(iterator it) {  
    iterator tmp(it.ptr->next);  
    it.ptr->prev->next = it.ptr->next;  
    it.ptr->next->prev = it.ptr->prev;  
    delete it.ptr;  
    mSize--;  
    return tmp;  
}
```

- Header make insert/erase very simple

