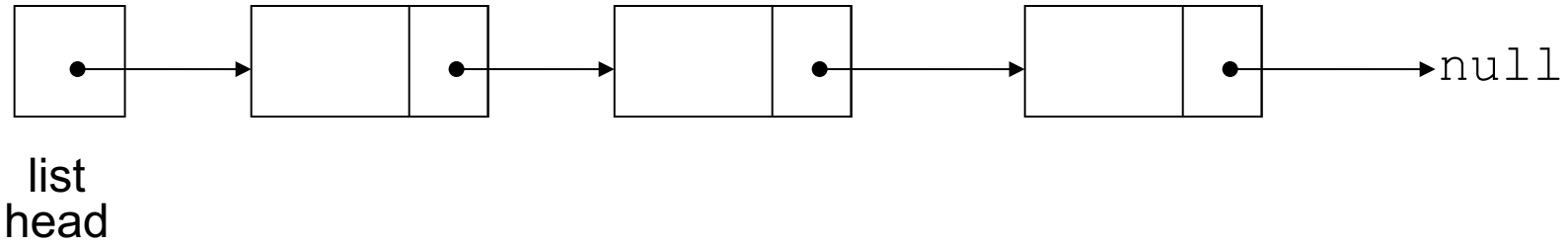
Linked Lists

Part 1&2

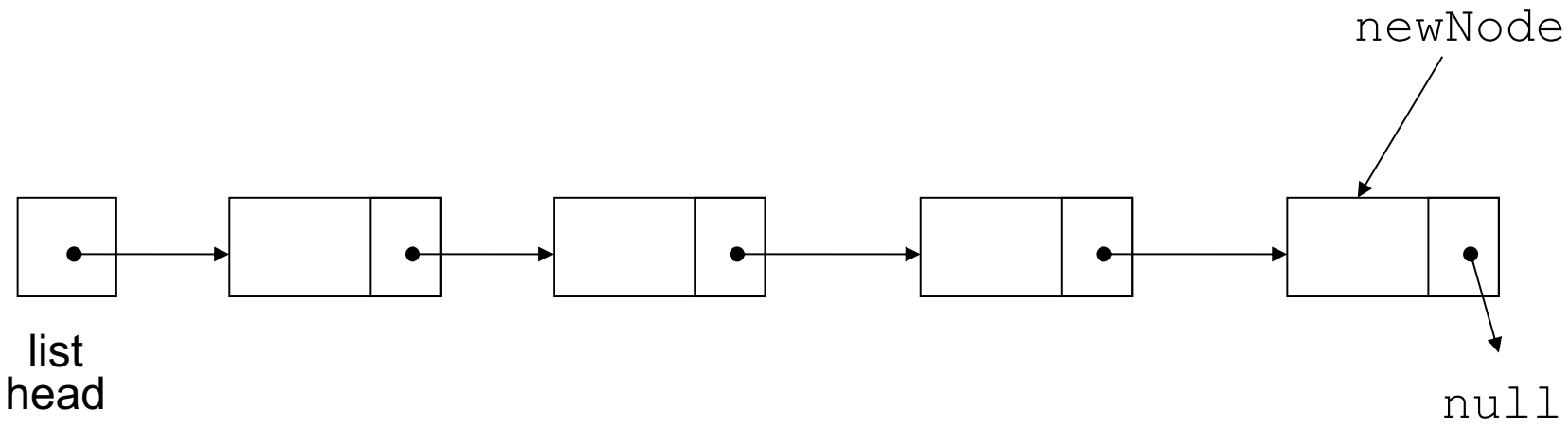
Week 10

Introduction to Linked List ADT

- Linked list: set of data structures (nodes) that contain references to other data structures

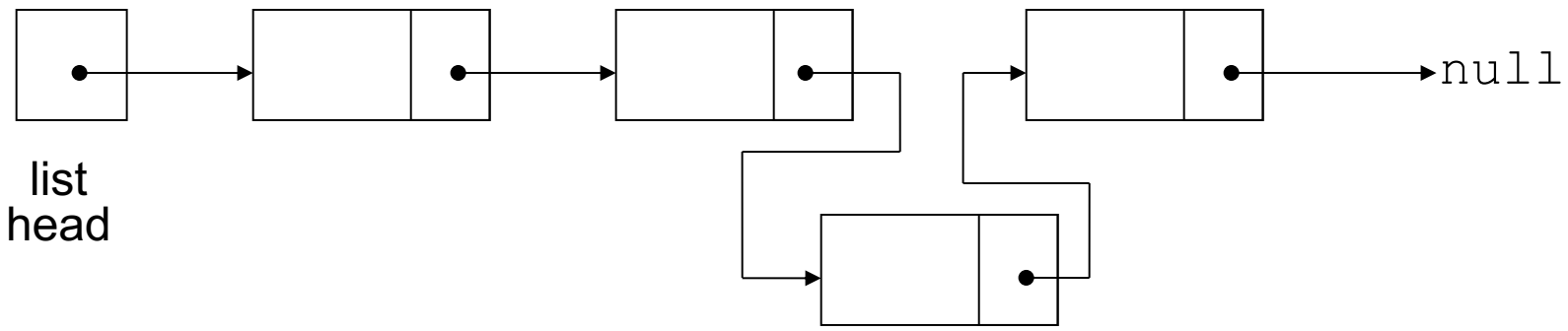


- References may be addresses or array indices
- Data structures can be added to or removed from the linked list during execution



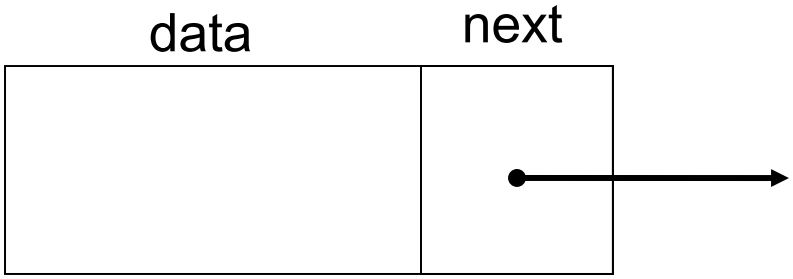
Linked List vs Arrays and vectors

- Linked lists can grow and shrink as needed, unlike arrays, which have a fixed size
- Linked lists can insert a node between other nodes easily

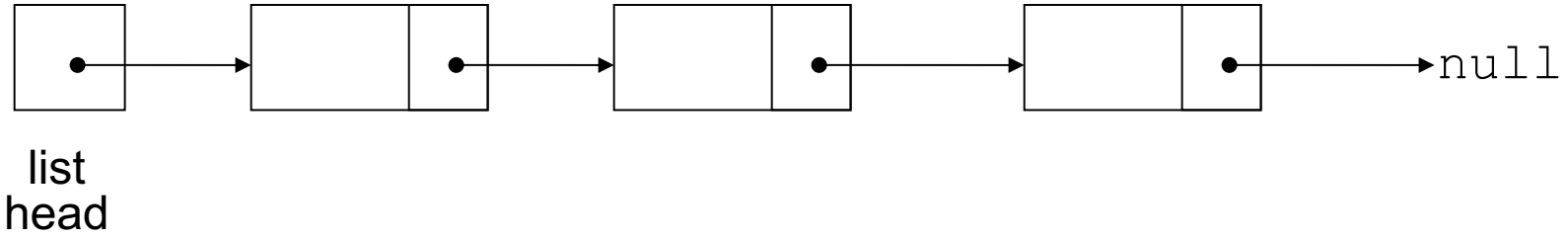


Node organization

- A node contains:
  - data: one or more data fields – may be organized as structure, object, etc.
  - a pointer that can point to another node



- Linked list contains 0 or more nodes:



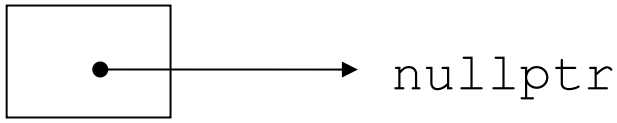
- Has a list head to point to first node
- Last node points to nullptr

Declaring a List

- If a list currently contains 0 nodes, it is the empty list
- In this case the list head points to null
- It is used to indicate the end-of-list
- Should always be tested for before using a pointer:

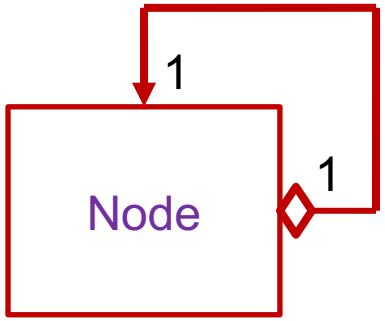
```
ListNode *p;
while (!p)
```

head




```
Node *head = nullptr;
```

```
struct Node
{
    int value;
    Node *next;
};
```




```
struct Node
{
    T value;
    Node *next;
};
```

Linked List Operations

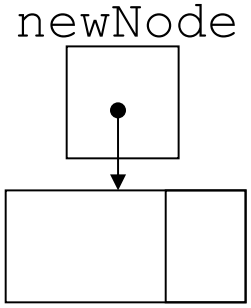
- List basic operations:
  - Create a list
  - append a node to the end of the list
  - Insert a node to the front of the list
  - insert a node within the list
  - traverse the linked list
  - delete a node
  - delete/destroy the list

```
template<class T>
class MyList {
private:
    struct Node{
        T value;
        Node *next;
    };

    Node *head;
    int sz;

    void *getNew(T d);
    void *search(T);
    void *movetoend();

public:
    MyList();
    void push_back(T);
    void push_front(T);
    void insert(T);
    bool insert(int,T);
    bool deleteNode(T);
    void print() const;
    int size();
    ~MyList();
};
```

Create a New Node

- Allocate memory for the new node:

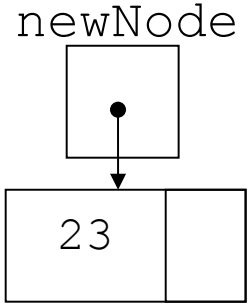
```
newNode = new MyList;
```




```
template<class T>
MyList<T>::MyList() {
    head = nullptr;
    sz = 0;
}
```

- Initialize the contents of the node:

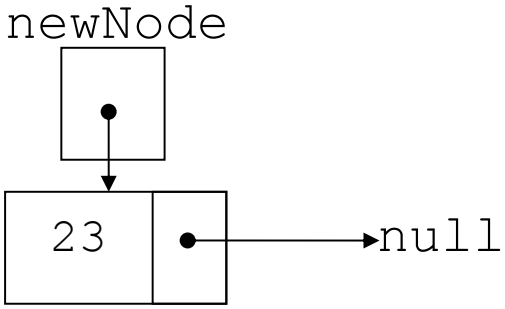
```
newNode->value = num;
```




```
template<class T>
void *MyList<T>::getNew(T d)
{
    Node *newNode = new Node;

    newNode->value = d;

    newNode->next = nullptr;

    return newNode;
}
```

- Set the pointer field to nullptr:

```
newNode->next = nullptr;
```

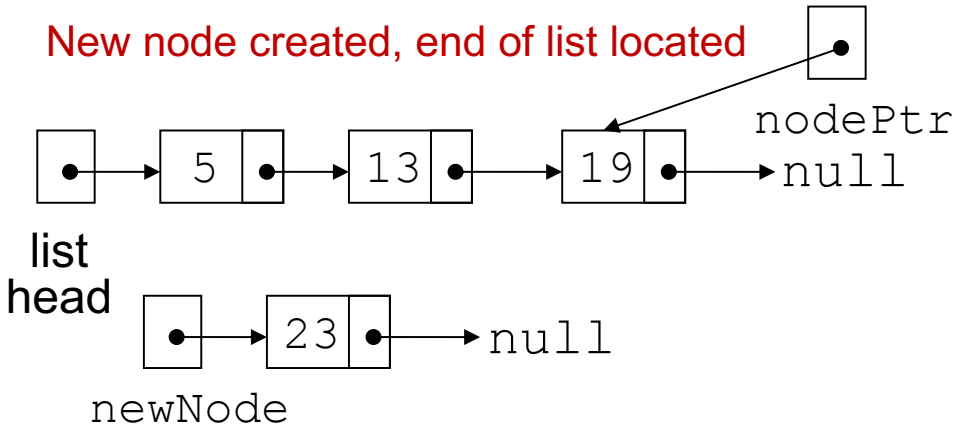


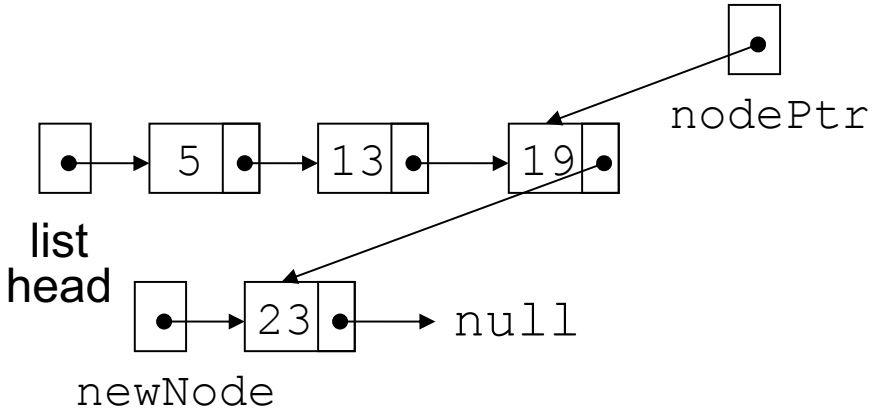
Appending a Node

- Add a node to the end of the list
- Basic process:
  - Create the new node
  - Add node to the end of the list:
    - If list is empty, set head pointer to this node
    - Else,
      - traverse the list to the end
      - set pointer of last node to point to new node

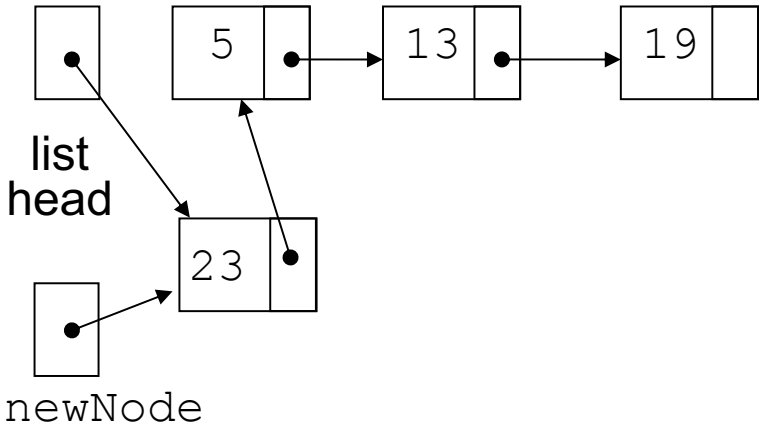
```
template<class T>
void MyList<T>::push_back(T d)
{
   Node *newNode = static_cast<Node*>(getNew(d));

   sz++;     //size of the list

   if (head == nullptr) {
      head = newNode;
   } else {
      Node *ptr = static_cast<Node*> (movetoend());

      ptr->next = newNode;
      newNode->next = nullptr;
   }
}
```

New node created, end of list located





Add to the front of the list



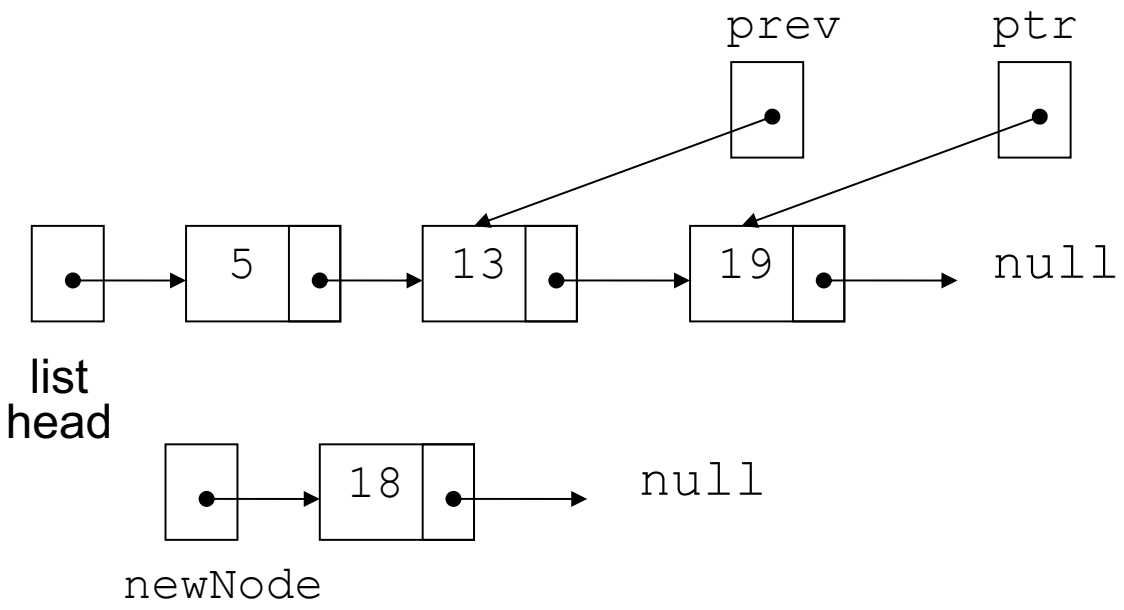

```
template<class T>
void MyList<T>::push_front(T d)
{
    Node *newNode = static_cast<Node*>(getNew(d));
    sz++;

    if(head == nullptr)
    {
        head = newNode;
    }
    else
    {
        newNode->next = head;
        head = newNode;
    }
}
```

```
int main()
{
    MyList<int> lst;
    lst.push_front(10);
    lst.push_back(20);
    cout << lst.size() << endl;
    lst.push_back(40);
    lst.push_front(5);
    cout << lst.size() << endl;
    lst.print();
    return 0;
}
```

```
2
4
5->10->20->40->
```

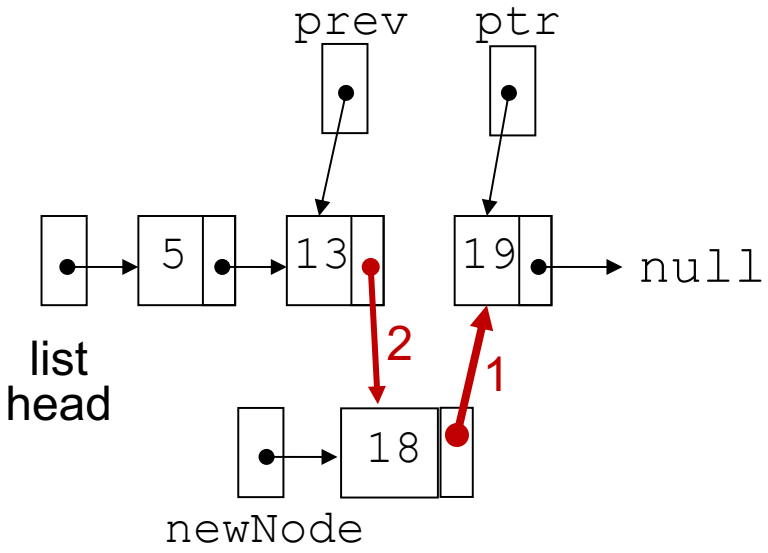
Inserting a Node into a Linked List

- Used to maintain a linked list in order
- Requires two pointers to traverse the list:
  - pointer to locate the node with data value greater than that of node to be inserted
  - pointer to 'trail behind' one node, to point to node before point of insertion
- New node is inserted between the nodes pointed at by these pointers



New node created, correct position located

Inserting a Node into a Linked List



New node inserted in order in the linked list

```
int main() {
    MyList<int> lst;
    lst.push_front(10);
    lst.push_back(20);
    cout << lst.size() << endl;
    lst.insert(15);
    cout << lst.size() << endl;
    lst.print();
    return 0;
}
```

```
2
3
10->15->20->
```

```
template<class T>
void MyList<T>::insert(T d){
    Node *newNode = static_cast<Node*>(getNew(d));
    sz++;

    if (head == nullptr) {
        head = newNode;
    }
    else
    {
        Node *ptr = head;
        Node  *prev=nullptr;
        while (ptr!=nullptr && ptr->value<d)
        {
            prev = ptr;
            ptr = ptr->next;
        }
        if (prev == nullptr){
            head = newNode;
            newNode->next = ptr;
        }
        else {
            prev->next = newNode;        1
            newNode->next = ptr;         2
        }
    }
}
```

Inserting a Node into a Linked List (2)

- Insert a node at a specific position in the linked list

1) Create a new node initialized to a value v.
2) Check if the position is within the linked list
3) Insert at the front of the linked list if pos = 0.
4) Otherwise, traverse the list to the required location.
5) Insert the newNode.

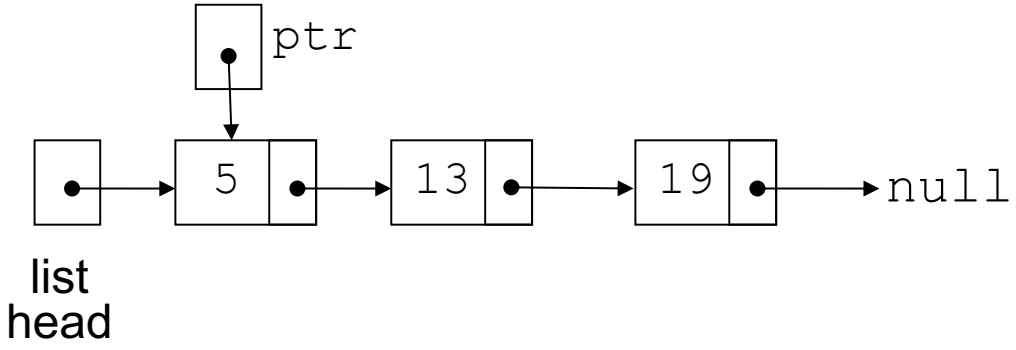
```
int main() {
    MyList<int> lst;
    lst.push_front(10);
    lst.push_back(20);
    lst.push_back(30);
    lst.push_back(40);
    cout << lst.size() << endl;
    lst.insert(3,15);
    cout << lst.size() << endl;
    lst.print();
    return 0;
}
```

```
template<class T>
bool MyList<T>::insert(int pos, T v) {
    if (pos > sz) return false;
    else
    {
        sz++;
1       Node *newNode = static_cast<Node*>(getNew(v));
2       if(pos == 0){
3           newNode->next = head;
            head = newNode;
        } else
        {
4           Node *ptr = head;
            for (int i=0;i<pos-1;i++){
                ptr=ptr->next;
            }

5           newNode->next = ptr->next;
            ptr->next = newNode;
        }
    }
    return true;
}
```

```
4
5
10->20->30->15->40->
```

Traversing a Linked List

- Visit each node in a linked list: display contents, validate data, etc.
- Basic process:
  - set a pointer to the contents of the head pointer
  - while pointer is not a null pointer
    - process data
    - go to the next node by setting the pointer to the pointer field of the current node in the list
  - end while



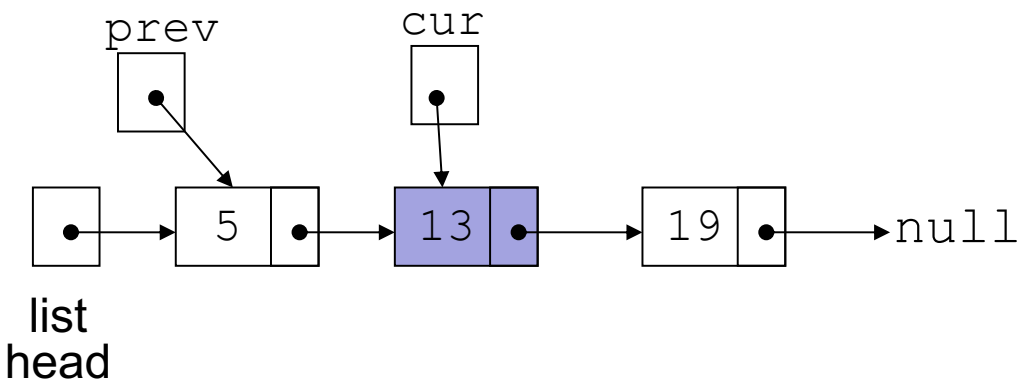

```
template<class T>
void *MyList<T>::movetoend()
{
   Node *ptr = head;

   while(ptr->next != nullptr)
      ptr=ptr->next;

   return ptr;
}
```

```
template<class T>
void MyList<T>::print() const
{
   Node *ptr = head;
   while(ptr!=nullptr)
   {
      cout << ptr->value << "->";
      ptr=ptr->next;
   }
   cout << endl;
}
```

`ptr` points to the node containing `5`, then the node containing `13`, then the node containing `19`, then points to the null pointer, and the list traversal stops

Deleting a Node

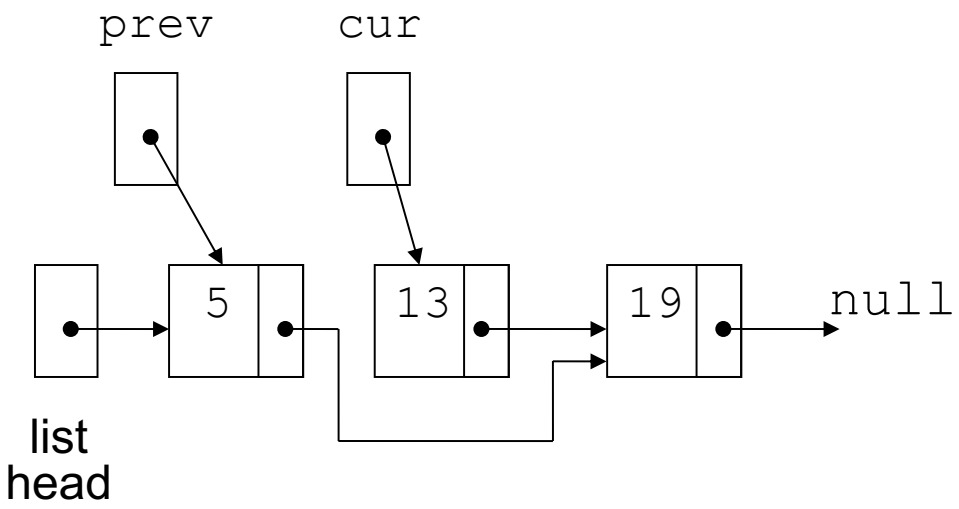
- Used to remove a node from a linked list
- If list uses dynamic memory, then delete node from memory
- Requires two pointers: one to locate the node to be deleted, one to point to the node before the node to be deleted



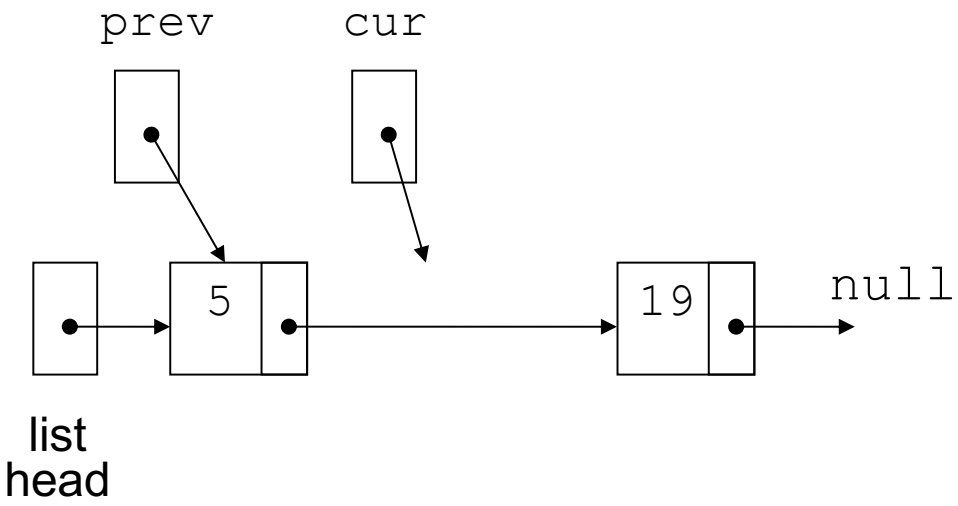
Locating the node containing 13

```
template<class T>
bool MyList<T>::deleteNode (T d)
{
   Node *prev=nullptr;
   Node *cur=head;
   while(cur->next!=nullptr && cur->value!=d)
   {
      prev=cur;
      cur=cur->next;
   }
   if (cur == head)
       head = head->next;          //first node
   else
   if(cur->next == nullptr)
      prev->next = nullptr;        //last node
   else
      prev->next = cur->next;      //middle

   sz--;
   delete cur;
   return true;
}
```

Deleting a Node



Adjusting pointer around the node to be deleted



Linked list after deleting the node containing 13

```
int main() {
    MyList<int> lst;
    lst.push_front(10);
    lst.push_back(20);
    lst.push_back(30);
    lst.push_back(40);
    cout << lst.size() << endl;
    lst.insert(3,15);
    cout << lst.size() << endl;
    lst.print();
    lst.deleteNode(30);
    cout << lst.size() << endl;
    lst.print();
    return 0;
}
```

```
4
5
10->20->30->15->40->
4
10->20->15->40->
```

Destroying a Linked List

- Must remove all nodes used in the list
- To do this, use list traversal to visit each node
- For each node,
  - Unlink the node from the list
  - If the list uses dynamic memory, then free the node's memory
- Set the list head to `nullptr`

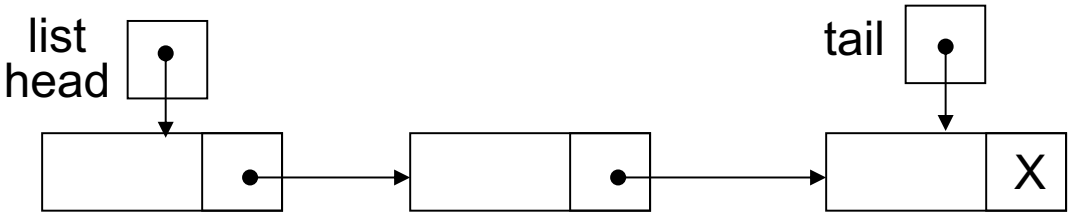
```
template<class T>
MyList<T>::~MyList()
{
   Node *cur;
   Node *nextNode;
   cur = head;
   while(cur != nullptr)
   {
      nextNode = cur->next;
      delete cur;
      cur = nextNode;
   }
   sz =0;
}
```

Destroying a Linked List

- Adding a tail pointer to the end of a linear linked list eliminates the need to traverse the list to reach the last node.

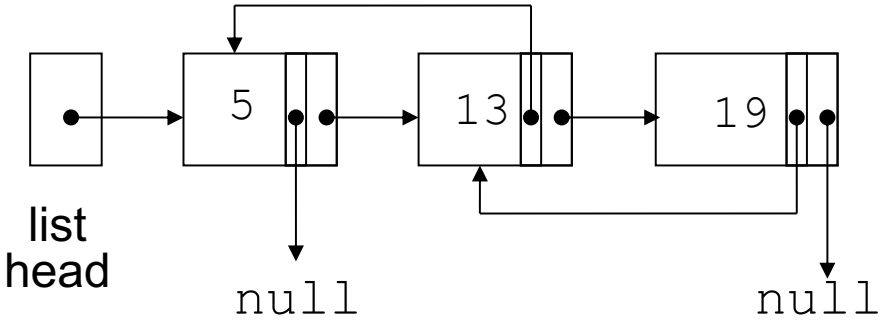
```
template<class T>
void MyList<T>::push_back(T d)
{
    Node *newNode =
static_cast<Node*>(getNew(d));

    sz++;
    if (head == nullptr) {
        head = newNode;
    }
    else
    {
        Node *ptr = static_cast<Node*>(movetoend());
        ptr->next = newNode;
    }
}
```

```
template<class T>
void MyList<T>::push_back_tail(T d)
{
    Node *newNode =
static_cast<Node*>(getNew(d));

    sz++;
    if (head == nullptr) {
        head = newNode;
        tail = newNode;
    }
    else
    {
        tail->next = newNode;
        tail = newNode;
    }
}
```



Doubly Linked List

- Other linked list organizations:
  - doubly-linked list: each node contains two pointers: one to the next node in the list, one to the previous node in the list

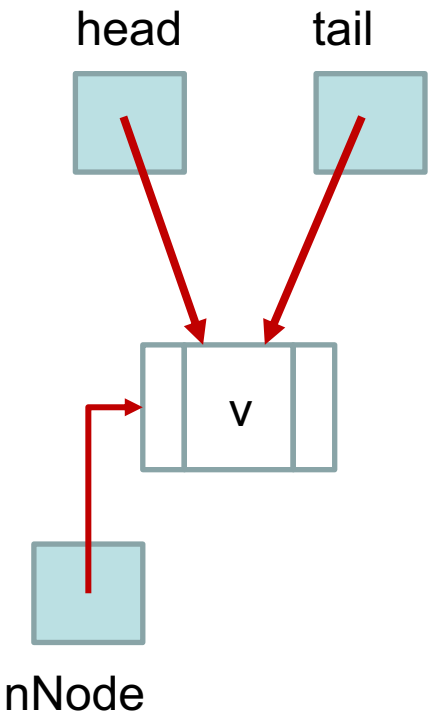



```
template<typename T>
struct DNode{

    T  info;
    DNode<T> *next;
    DNode<T> *prev;

    DNode(T v=T()){
        info = v;
        next = nullptr;
        prev = nullptr;
    }
};
```

```
template<typename T>
class DoubleList {
    DNode<T> *head;
    DNode<T> *tail;
    int sz;
    :
public:
    DoubleList(){
        head = tail = nullptr;
        sz = 0;
    }
    :
};
```

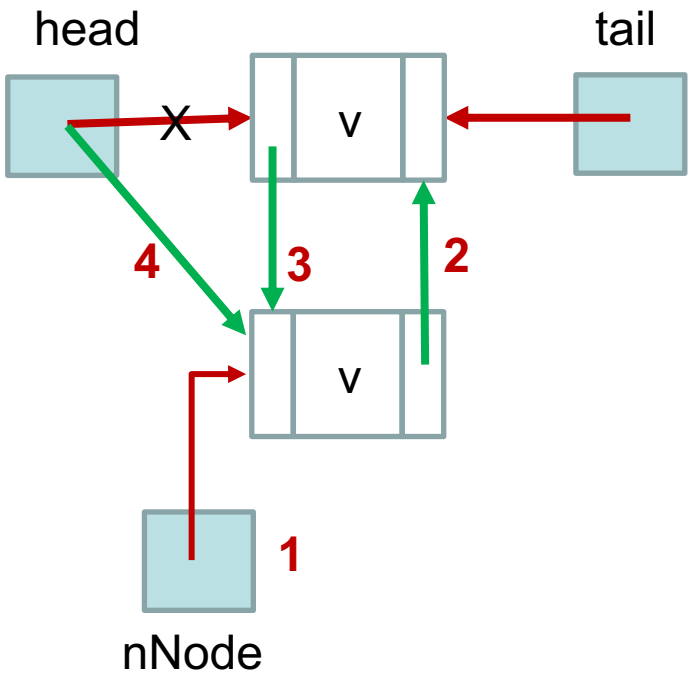
Add the first Node

```
void addFirst (T v)
{
        DNode<T> * nNode = new DNode<T>(v);
        head = tail = nNode;
}
```



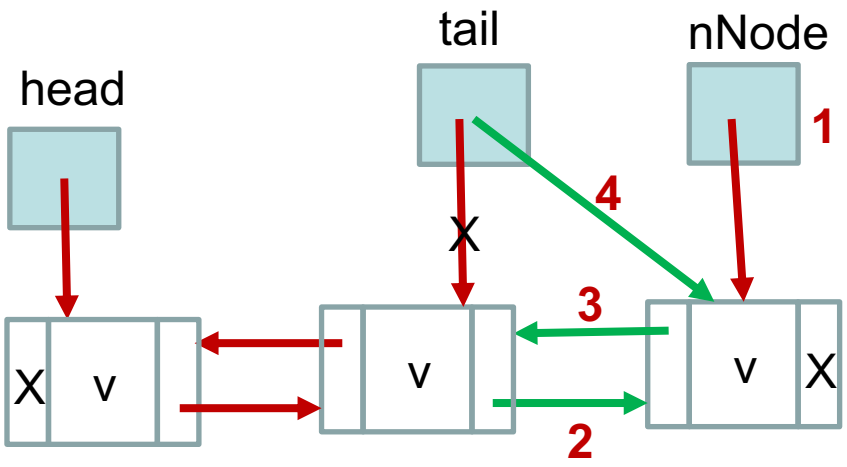
push_front to a doubly linked list

```
void push_front(T v)
{
 1    DNode<T> *nNode = new DNode<T>(v);

      if (head == nullptr)
         addFirst(v);
      else
      {
 2       nNode->next = head;

 3       head->prev = nNode;

 4       head = nNode;
      }
      sz++;
   }
```



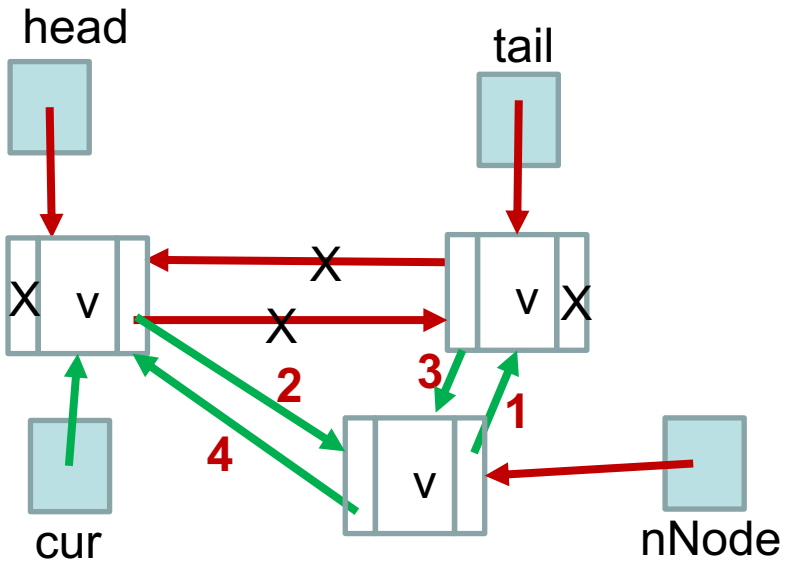
push_back to a doubly linked list

```
void push_back(T v)
{
1   DNode<T> *nNode = new DNode<T>(v);
    if (tail == nullptr)
        addFirst(v);
    else
    {
2       tail->next = nNode;

3       nNode->prev = tail;

4       tail = nNode;
    }
    sz++;
 }
```



Insert into a sorted doubly linked list

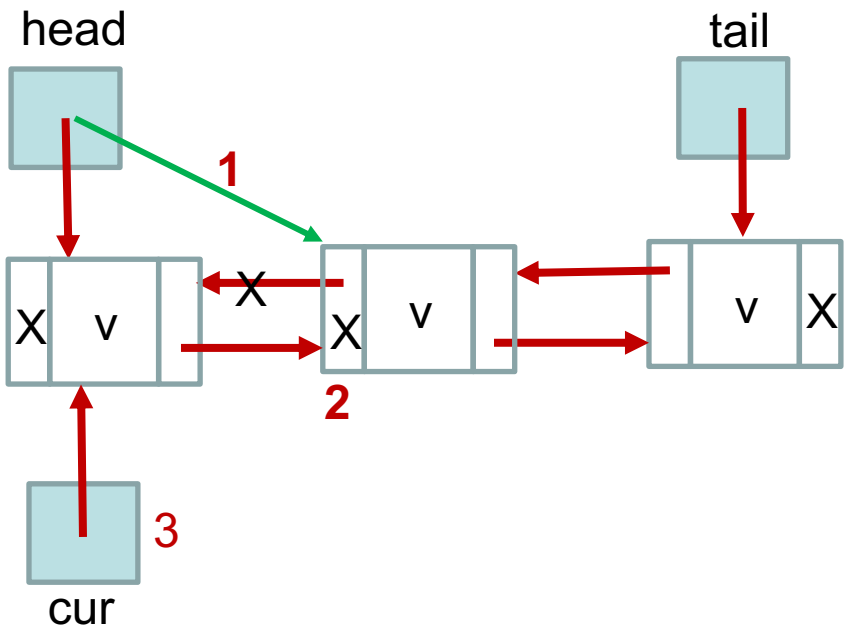
```
DNode<T> *locatePos(T v) {
        DNode<T> *ptr = head;
        DNode<T> *prev = nullptr;
        while (ptr!=nullptr && ptr->info < v)
        {
            prev = ptr;
            ptr = ptr->next;
        }
        return prev;
    }
```




```
void insert(T v){
        if (head == nullptr || tail == nullptr)
            addFirst(v);
        else
        {
            DNode<T> *cur = locatePos(v);
            DNode<T> *nNode = new DNode<T>(v);
            if (cur == head)
                push_front(v);
            else
            if (cur == tail)
                push_back(v);
            else
            {
                nNode->next = cur->next;
                cur->next = nNode;
                nNode>next->prev = nNode;
                nNode->prev = cur;
            }
            sz++;
        }
    }
```

Delete from a doubly linked list

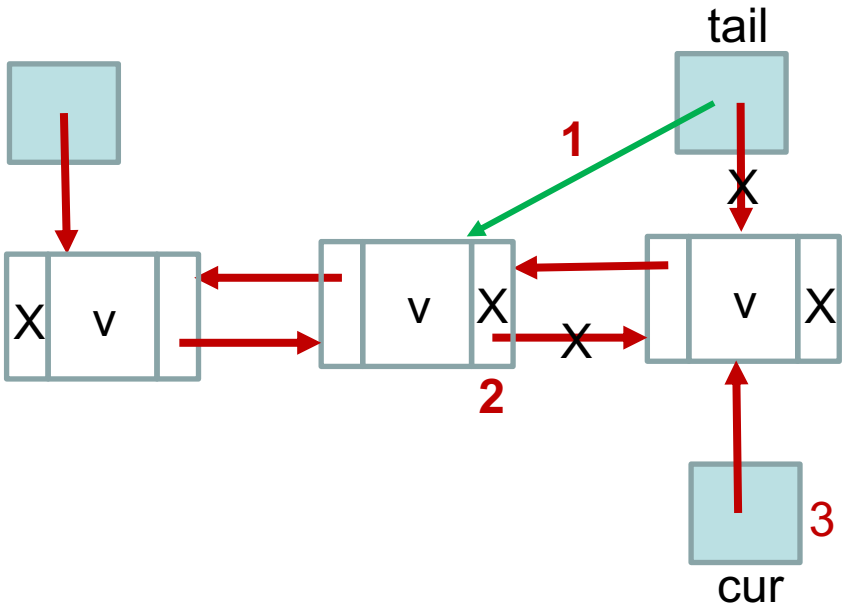
```
DNode<T> *find(T v)const
{
        DNode<T> *ptr = head;
        while (ptr!=nullptr && ptr->info!=v)
                ptr = ptr->next;
        return ptr;
}
```




```
void ddelete(T v){
        DNode<T> *cur = find(v);
        if (cur == nullptr)
            cout << "Node not found..\n";
        else {
            if (cur->prev == nullptr) {
1               head = head->next;
2               head->prev = nullptr;
            }
            else
            if(cur->next == nullptr) {
                tail = tail->prev;
                tail->next = nullptr;
            }
            else {
                cur->prev->next = cur->next;
                cur->next->prev = cur->prev;
            }
3           delete cur;
            sz--;
        }
    }
```

Delete from a doubly linked list

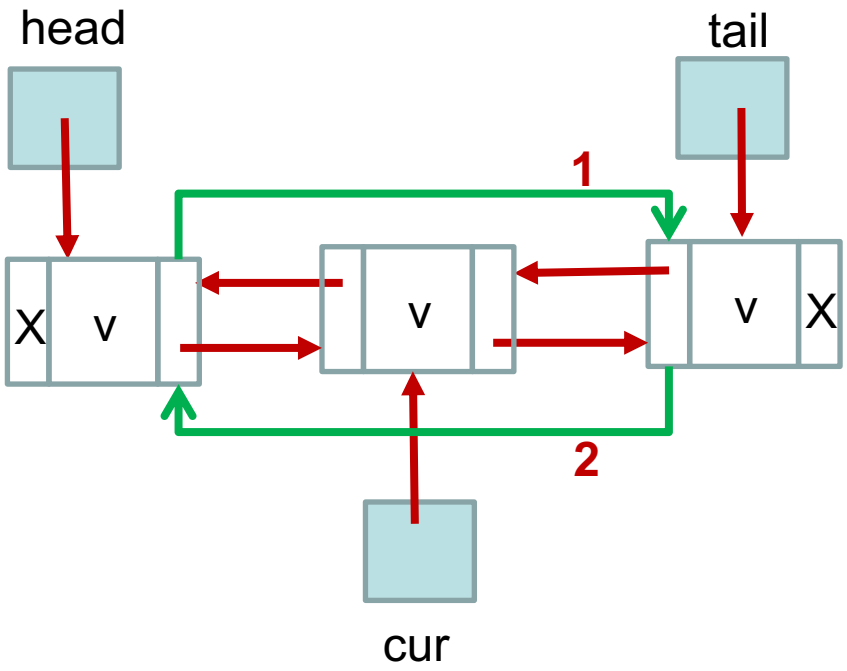
```
DNode<T> *find(T v)const
{
        DNode<T> *ptr = head;
        while (ptr!=nullptr && ptr->info!=v)
                ptr = ptr->next;
        return ptr;
}
```




```
void ddelete(T v){
        DNode<T> *cur = find(v);
        if (cur == nullptr)
            cout << "Node not found..\n";
        else {
            if (cur->prev == nullptr) {
                head = head->next;
                head->prev = nullptr;
            }
            else
            if(cur->next == nullptr) {
    1           tail = tail->prev;
    2           tail->next = nullptr;
            }
            else {
                cur->prev->next = cur->next;
                cur->next->prev = cur->prev;
            }
    3       delete cur;
            sz--;
        }
    }
```

Delete from a doubly linked list

```
DNode<T> *find(T v)const
{
        DNode<T> *ptr = head;
        while (ptr!=nullptr && ptr->info!=v)
                ptr = ptr->next;
        return ptr;
}
```




```
void ddelete(T v){
        DNode<T> *cur = find(v);
        if (cur == nullptr)
            cout << "Node not found..\n";
        else {
            if (cur->prev == nullptr) {
                head = head->next;
                head->prev = nullptr;
            }
            else
            if(cur->next == nullptr) {
                tail = tail->prev;
                tail->next = nullptr;
            }
            else {
    1           cur->prev->next = cur->next;
    2           cur->next->prev = cur->prev;
            }
    3       delete cur;
            sz--;
        }
    }
```

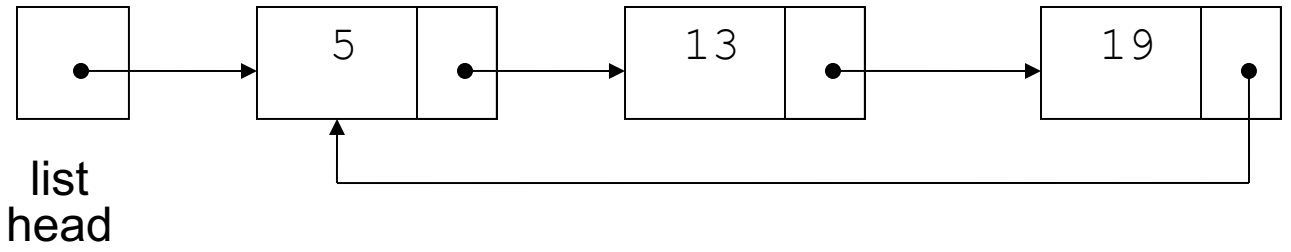
Traverse and print a doubly linked list forward and backward

```
void print() const
{
        DNode<T> *ptr = head;
        while (ptr!=nullptr){
            cout << ptr->info << "->";
            ptr=ptr->next;
        }
        cout << endl;
    }
```

```
void print_back() const
{
        DNode<T> *ptr = tail;
        while (ptr!=nullptr){
            cout << ptr->info << "->";
            ptr=ptr->prev;
        }
        cout << endl;
    }
```

Circular Linked List

- Other linked list organizations:
  - circular linked list: the last node in the list points back to the first node in the list, not to the null pointer



STL List Container

- Template for a doubly linked list
- Member functions for
  - locating beginning, end of list: front, back, end
  - adding elements to the list: insert, merge, push_back, push_front
  - removing elements from the list: erase, pop_back, pop_front, unique

```
#include <algorithm>
#include <iostream>
#include <list>

using namespace std;

int main()
{
   list<int> l = { 7, 16, 6, 8 };
   l.push_front(25);
   l.push_back(13);

   //search for 16
   auto it = find(l.begin(), l.end(), 16);

   //if found then insert 42 before 16
   if (it != l.end()) {
      l.insert(it, 42);
   }

   for(auto t=l.begin();t!=l.end();t++){
      cout << *t << '\n';
   }
}
```

Iterators:

- begin
- end
- rbegin
- rend
- cbegin C++11
- cend C++11
- crbegin C++11
- crend C++11

Capacity:

- empty
- size
- max_size

Element access:

- front
- back

Modifiers:

- assign
- emplace_front C++11
- push_front
- pop_front
- emplace_back C++11
- push_back
- pop_back
- emplace C++11
- insert
- erase
- swap
- resize
- clear

Operations:

- splice
- remove
- remove_if
- unique
- merge
- sort
- reverse

STL forward_list

- Template for a singly linked list
- You can only step forward in a `forward_list`.
- A `forward_list` uses slightly less memory than a `list`, and has takes slightly less time for inserting and removing nodes.
- Provides most, but not all, of the same member functions as the `list` container

```
#include <algorithm>
#include <iostream>
#include <forward_list>

using namespace std;

int main()
{
   forward_list<int> l = { 7, 16, 6, 8 };
   l.push_front(25);

   //search for 16
   auto it = find(l.begin(), l.end(), 16);

   //if found then insert 42 before 16
   if (it != l.end()) {
      l.insert_after(it, 42);
   }

   for(auto t=l.begin();t!=l.end();t++){
      cout << *t << '\n';
   }
}
```

Modifiers

assign	Assign content (public member function)
emplace_front	Construct and insert element at beginning (public member function)
push_front	Insert element at beginning (public member function)
pop_front	Delete first element (public member function)
emplace_after	Construct and insert element (public member function)
insert_after	Insert elements (public member function)
erase_after	Erase elements (public member function)
swap	Swap content (public member function)
resize	Change size (public member function)
clear	Clear content (public member function)

Iterators

before_begin
begin
end
cbefore_begin
cbegin
cend

Capacity

empty
max_size

Operations

splice_after	Transfer elements from another forward_list (public member function)
remove	Remove elements with specific value (public member function)
remove_if	Remove elements fulfilling condition (public member function template)
unique	Remove duplicate values (public member function)
merge	Merge sorted lists (public member function)
sort	Sort elements in container (public member function)
reverse	Reverse the order of elements (public member function)

C++11 C++14

```
default (1)   explicit forward_list (const allocator_type& alloc = allocator_type());
fill (2)      explicit forward_list (size_type n);
              explicit forward_list (size_type n, const value_type& val,
                                     const allocator_type& alloc = allocator_type());
range (3)     template <class InputIterator>
                forward_list (InputIterator first, InputIterator last,
                              const allocator_type& alloc = allocator_type());
copy (4)      forward_list (const forward_list& fwdlst);
              forward_list (const forward_list& fwdlst, const allocator_type& alloc);
move (5)      forward_list (forward_list&& fwdlst);
              forward_list (forward_list&& fwdlst, const allocator_type& alloc);
initializer list (6)
              forward_list (initializer_list<value_type> il,
                            const allocator_type& alloc = allocator_type());
```

Linked List using smart pointers

```
template<class T>
struct Node {
    T value;
    shared_ptr<Node> next;

    Node(T v) {
        value = v;
        next = nullptr;
    }
};
```

```
template<class T>
MyList<T>::MyList() {
    head=nullptr;
    tail=nullptr;
    sz = 0;
}
```

```
template<class T>
int MyList<T>::size() {
    return sz;
}
```

```
template<class T>
MyList<T>::~MyList() {
    sz =0;
}
```

```
template<class T>
class MyList {
private:
    shared_ptr<Node<T>>  head;
    shared_ptr<Node<T>>  tail;

    int sz;

    shared_ptr<Node<T>> movetoend();

public:
    MyList();
    void push_back(T);
    void push_back_tail(T);
    void push_front(T);
    void insert(T);
    bool insert(int,T);
    bool deleteNode(T);
    void print() const;
    int size();
    ~MyList();
};
```

Linked List using smart pointers

```
template<class T>
void MyList<T>::push_front(T d){
    shared_ptr<Node<T>> newNode =
make_shared<Node<T>>(d);
    sz++;
    if(head == nullptr) {
        head = newNode;
        tail = newNode;
    }
    else {
        newNode->next = head;
        head = newNode;
    }
}
```

```
template<class T>
void MyList<T>::push_back(T d){
    shared_ptr<Node<T>> newNode =
make_shared<Node<T>>(d);
    sz++;
    if (head == nullptr){
        head = newNode;
        tail = newNode;
    }
    else {
        shared_ptr<Node<T>> ptr = movetoend();
        ptr->next = newNode;
    }
}
```

```
template<class T>
shared_ptr<Node<T>> MyList<T>::movetoend() {
    shared_ptr<Node<T>> ptr = head;
    while(ptr->next != nullptr)
        ptr=ptr->next;
    return ptr;
}
```

Linked List using smart pointers

```
template<class T>
void MyList<T>::push_back_tail(T d) {
    shared_ptr<Node<T>> newNode =
make_shared<Node<T>>(d);
    sz++;
    if (tail == nullptr){
        head = newNode;
        tail = newNode;
    }
    else
    {
        tail->next = newNode;
        tail = newNode;
    }
}
```

```
template<class T>
void MyList<T>::insert(T d) {
 shared_ptr<Node<T>> newNode =
make_shared<Node<T>>(d);
    sz++;
    if (!head) {
        head = newNode;
    } else {
        shared_ptr<Node<T>> ptr = head;
        shared_ptr<Node<T>> prev=nullptr;
        while (ptr!=nullptr && ptr->value<d){
            prev = ptr;
            ptr = ptr->next;
        }
        if (prev == nullptr){
            head = newNode;
            newNode->next = ptr;
        } else {
            prev->next = newNode;
            newNode->next = ptr;
        }
    }
}
```

Linked List using smart pointers

```
template<class T>
bool MyList<T>::insert(int pos, T v) {
    if (pos > sz) return false;
    else
    {
        sz++;
        shared_ptr<Node<T>> newNode =
make_shared<Node<T>>(v);
        if(pos == 0){
            newNode->next = head;
            head = newNode;
        }
        else
        {
            shared_ptr<Node<T>> ptr = head;
            for (int i=0;i<pos-1;i++){
                ptr=ptr->next;
            }
            newNode->next = ptr->next;
            ptr->next = newNode;
        }
    }
    return true;
}
```

```
template<class T>
bool MyList<T>::deleteNode(T d)
{
    shared_ptr<Node<T>> prev=nullptr;
    shared_ptr<Node<T>> cur=head;
    while(cur->next!=nullptr && cur->value!=d){
        prev=cur;
        cur=cur->next;
    }
    if (cur == head)
        head = head->next;
    else
        if(cur->next == nullptr)
            prev->next = nullptr;
        else
            prev->next = cur->next;

    sz--;
    return true;
}
```

Linked List using smart pointers

```
template<class T>
void MyList<T>::print()const{
    shared_ptr<Node<T>> ptr = head;
    while(ptr!=nullptr){
        cout << ptr->value << "(”
                << ptr.use_count() << ")" << "->";
        ptr=ptr->next;
    }
    cout << endl;
}
```

```
int main() {
    MyList<int> lst;
    lst.push_front(10);
    lst.push_back_tail(20);
    lst.push_back_tail(30);
    lst.push_back_tail(40);
    cout << lst.size() << endl;
    lst.print();

    lst.insert(3,15);
    cout << lst.size() << endl;
    lst.print();
    lst.deleteNode(30);
    cout << lst.size() << endl;
    lst.print();

    return 0;
}
```

```
4
10(2)->20(2)->30(2)->40(3)->
5
10(2)->20(2)->30(2)->15(2)->40(3)->
4
10(2)->20(2)->15(2)->40(3)->
```

Linked Lists

Part 1&2

Week 10