CS 014: Intro to Data Structures and Algorithms

Instructor: Ryan Rusich
E-mail: rusichr@cs.ucr.edu
Office: WCH 407

Lists

Lists

- Lists
  - List of students
  - List of games
  - List of assignments to complete
  - Etc.
- Is a collection of elements.
- One of most fundamental/simple data structures.
- Implementation:
  - Array-based
  - Node-based

Lists Implemented with Arrays

- A list can be implemented with an array.
  - Static memory allocation (space allocated at compile time)

```
int MAX_ARRAY_SIZE = 50;
int list[MAX_ARRAY_SIZE];
```

  - Dynamic memory allocation (space allocated at runtime)

```
int* list = NULL;
int MAX_ARRAY_SIZE;
cin >> MAX_ARRAY_SIZE;
list = new int[MAX_ARRAY_SIZE];
 // Initialize all elements to zero.
      ...
 delete[] list;
 list = NULL;
```

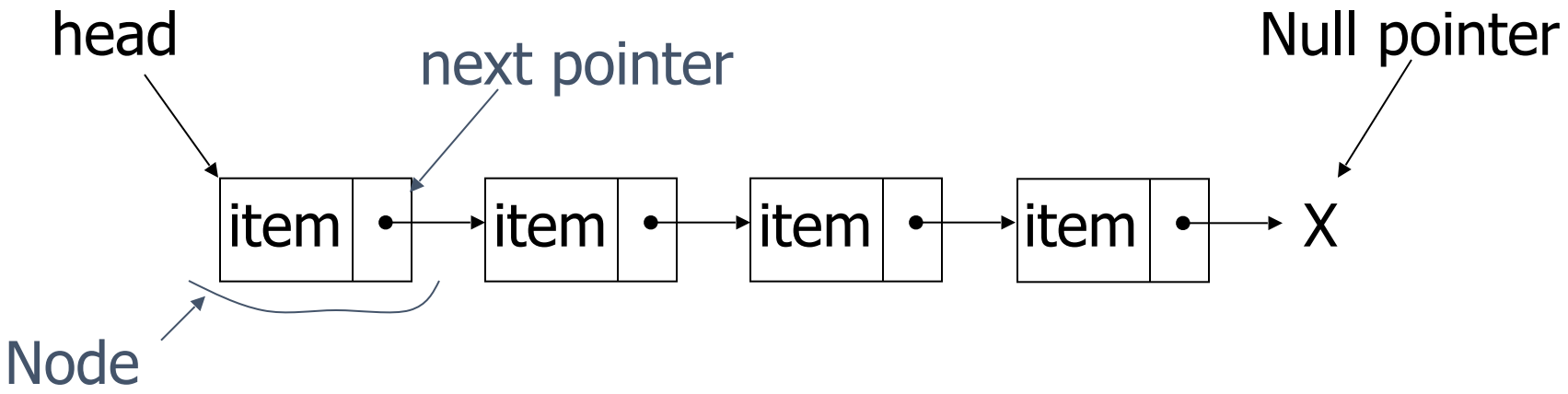
Lists Implemented with Arrays

- Strengths:
  - + Direct access
    - + Using subscript operator [ ]
  - + Good choice where list is built by inserts at end, and no deletes occur, only array accesses, i.e. find(pos)
  - + Can be very fast and efficient.
- Weaknesses:
  - – Wasted space or run out of space.
    - – Mostly empty (waste)
    - – Full (need to resize, typically double—making list half full)
  - – Costly to insert in the middle, or beginning of array-based list.
    - – Need to shift items up.

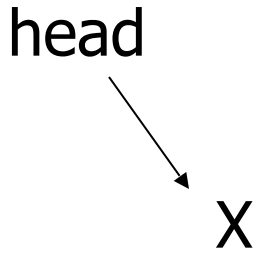
Linked List

- Composed of nodes.
- Each node holds data (item, value, element).
- Each node has a pointer (next) that connects to the next node in the list.
- Special pointer (head) to beginning of list. (Access through list ***ID->head)***
- Size of the list is flexible (grows on inserts, shrinks on deletes)
- List operations:
  - Insert item
  - Remove item
  - Read from item (access)
  - Write to item (mutate)
  - Traversal (visit each item: search, print)

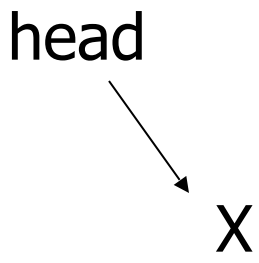
Linked List



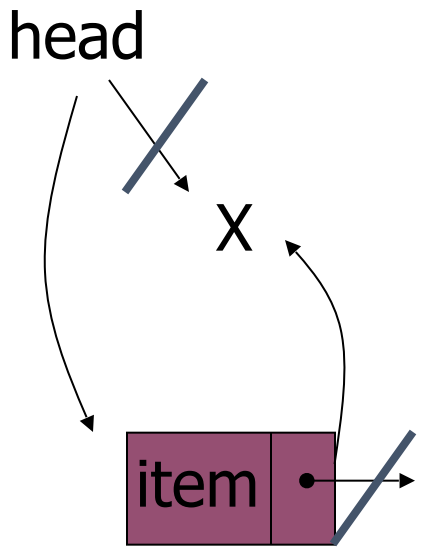
Empty list:



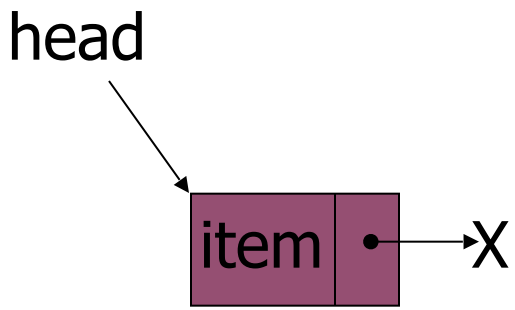
Linked list - Inserting Nodes



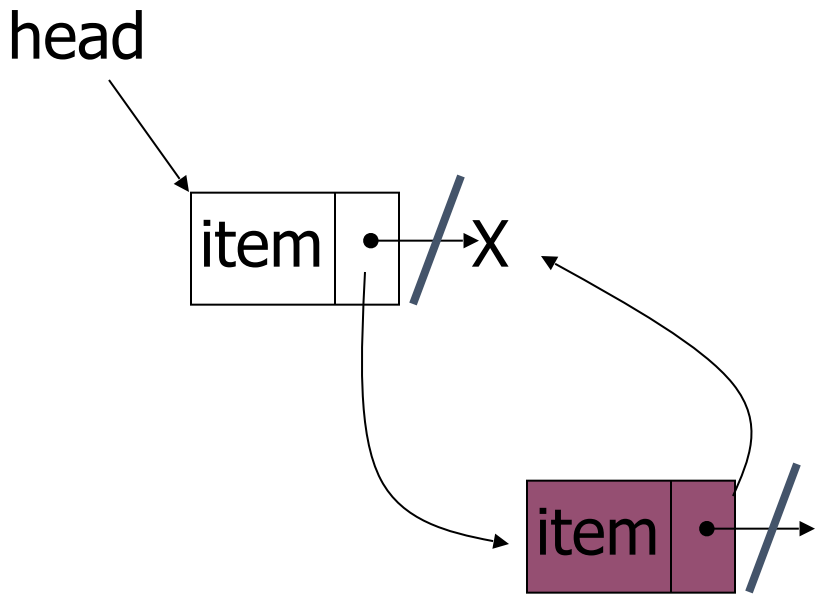
Linked list - Inserting Nodes



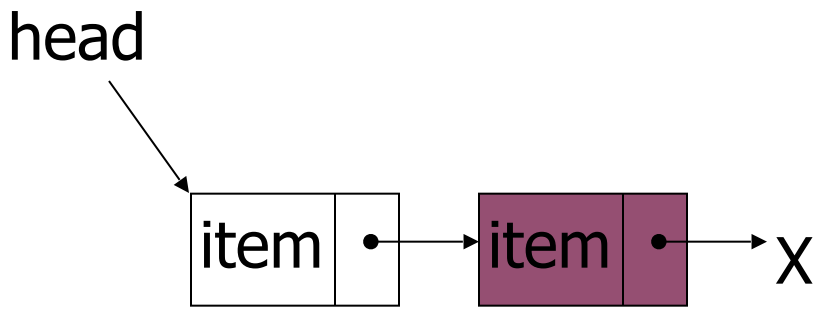
Linked list - Inserting Nodes



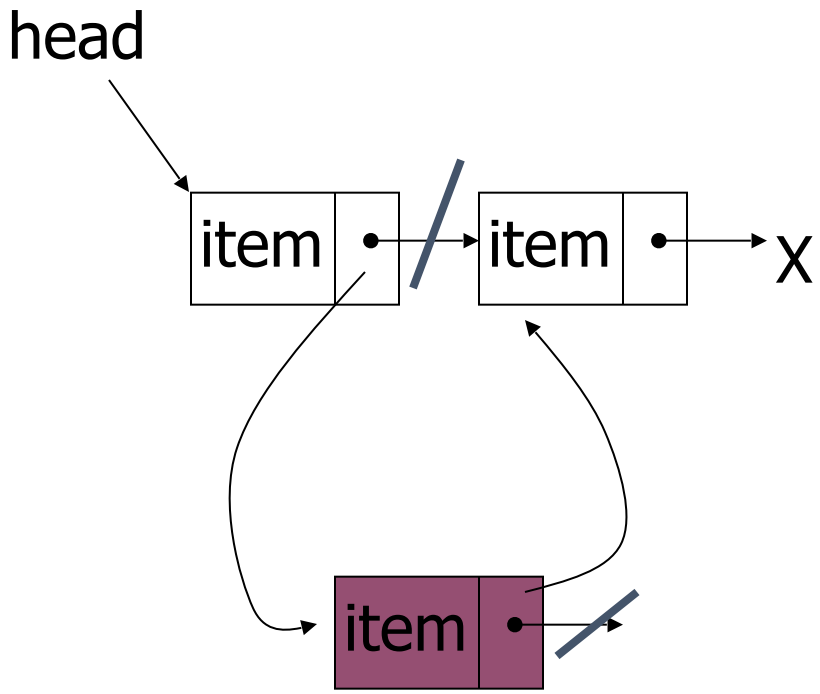
Linked list - Inserting Nodes



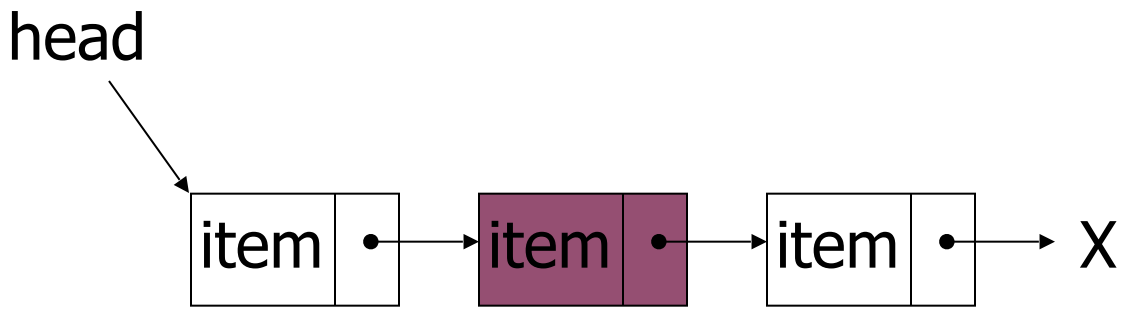
Linked list - Inserting Nodes



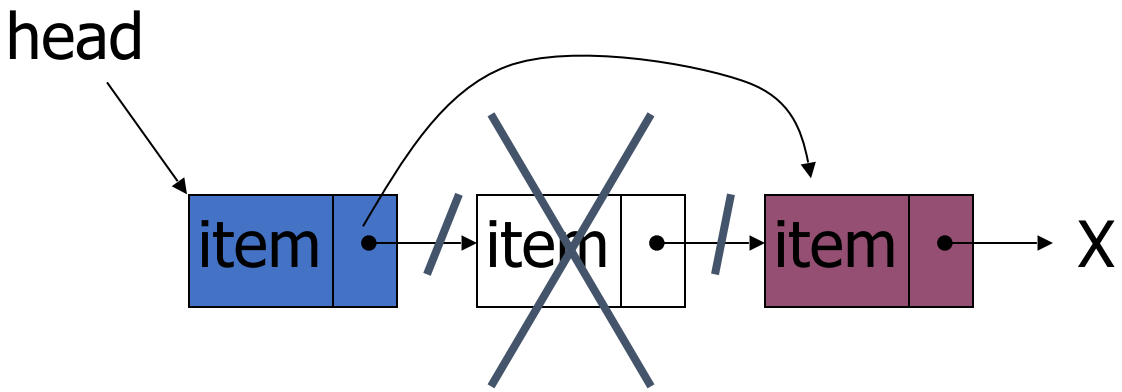
Linked list - Inserting Nodes



Linked list - Inserting Nodes



Linked list - Deleting Nodes



Linked List Implementation

```
class List {
private:
        Node* head;
public:
        List ( ) {
           head = NULL;
        };
        // Member functions
        // List Interface
};
```

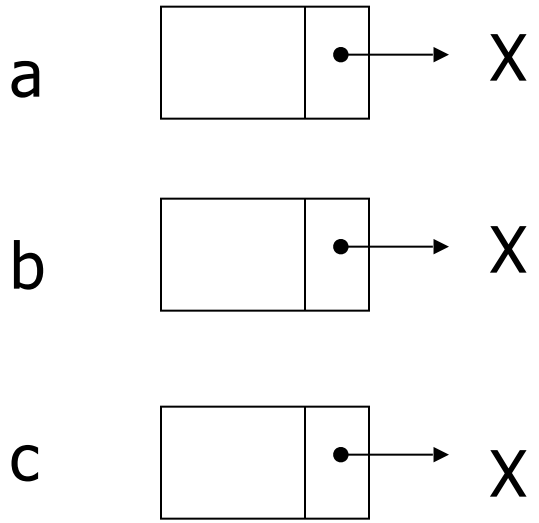
```
class Node {
    friend class List;
private:
    itemtype Item;
    Node* next;
};
```

Link List Manipulation

```
Node* a = new Node;
Node* b = new Node;
Node* c = new Node;
a->next = b;
b->next = c;
b->item = X;
c->item = Y;
a->item = Z
```

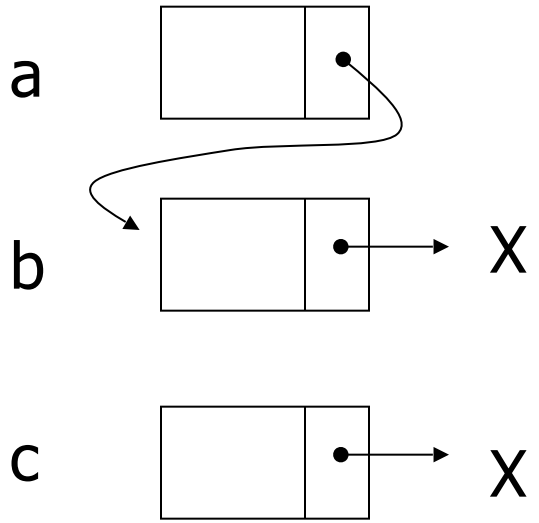
Link List Manipulation

```
Node* a = new Node;
Node* b = new Node;
Node* c = new Node;
a->next = b;
b->next = c;
b->item = X;
c->item = Y;
a->item = Z
```



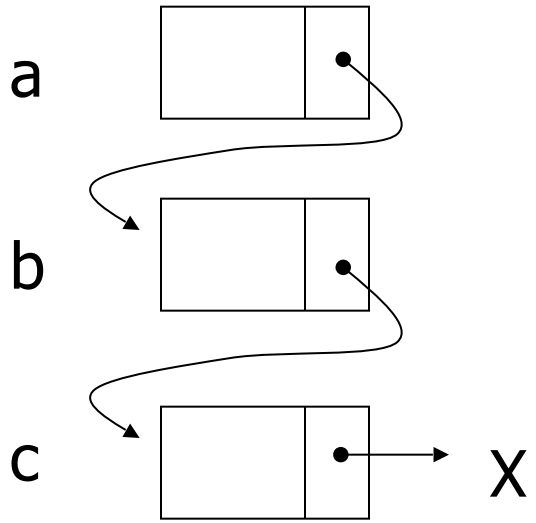
Link List Manipulation

```
Node* a = new Node;
Node* b = new Node;
Node* c = new Node;
a->next = b;
b->next = c;
b->item = X;
c->item = Y;
a->item = Z
```



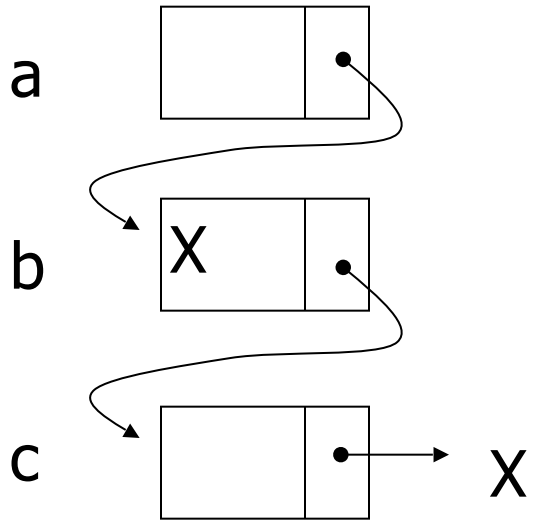
Link List Manipulation

```
Node* a = new Node;
Node* b = new Node;
Node* c = new Node;
a->next = b;
b->next = c;
b->item = X;
c->item = Y;
a->item = Z
```



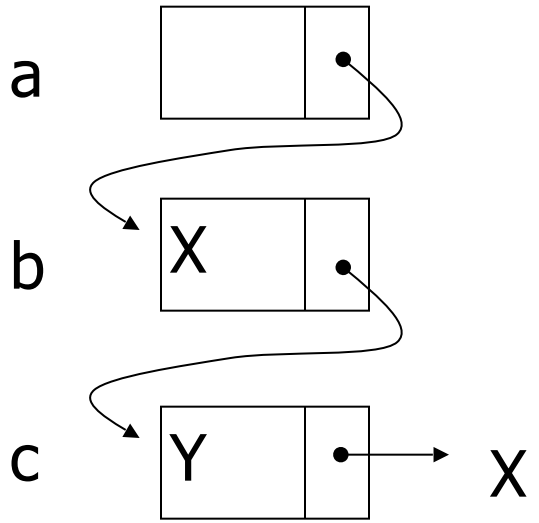
Link List Manipulation

```
Node* a = new Node;
Node* b = new Node;
Node* c = new Node;
a->next = b;
b->next = c;
b->item = X;
c->item = Y;
a->item = Z
```



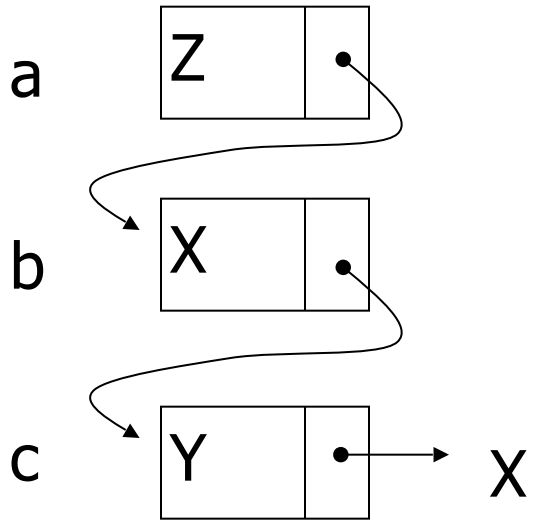
Link List Manipulation

```
Node* a = new Node;
Node* b = new Node;
Node* c = new Node;
a->next = b;
b->next = c;
b->item = X;
c->item = Y;
a->item = Z
```



Link List Manipulation

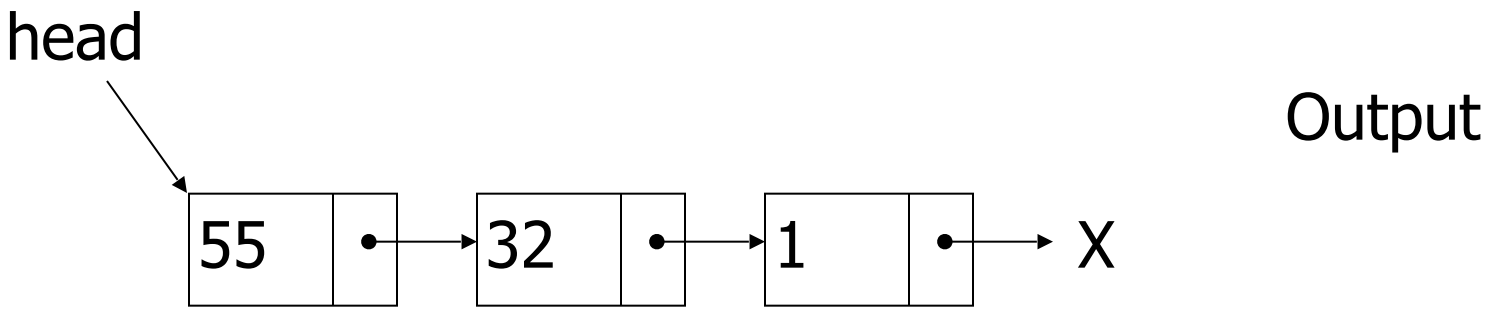
```
Node* a = new Node;
Node* b = new Node;
Node* c = new Node;
a->next = b;
b->next = c;
b->item = X;
c->item = Y;
a->item = Z
```



Linked List Code - Print

- Printing the list

```
Node* temp;
for ( temp = head; temp != NULL; temp = temp->next )
        cout << temp->item << " ";
```

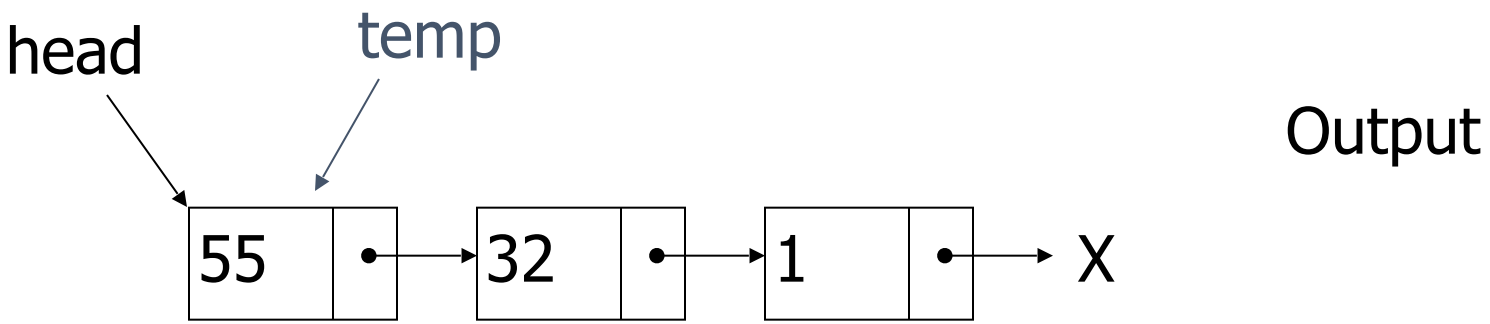


Output

Linked List Code - Print

- Printing the list

```
Node* temp;
for ( temp = head; temp != NULL; temp = temp->next )
        cout << temp->item << " ";
```

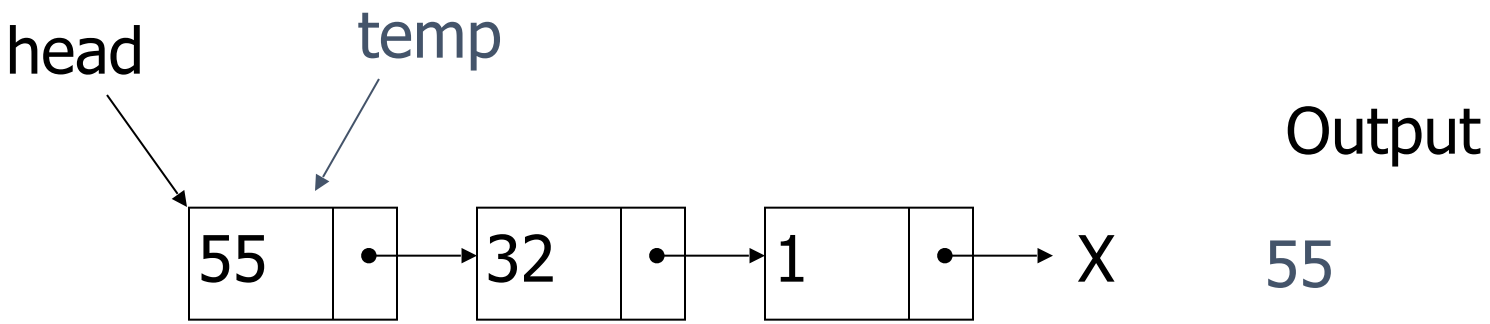


Output

Linked List Code - Print

- Printing the list

```
Node* temp;
for ( temp = head; temp != NULL; temp = temp->next )
        cout << temp->item << " ";
```



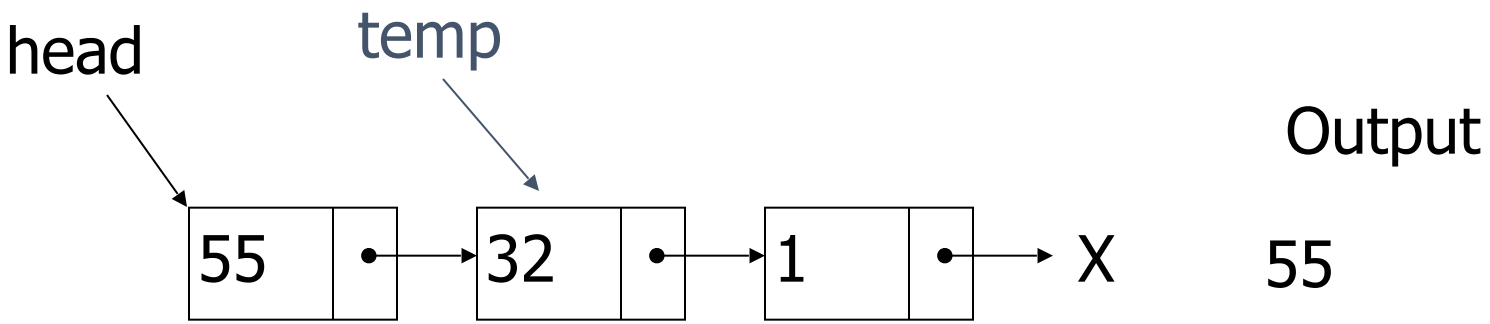
Output

55

Linked List Code - Print

- Printing the list

```
Node* temp;
for ( temp = head; temp != NULL; temp = temp->next )
        cout << temp->item << " ";
```



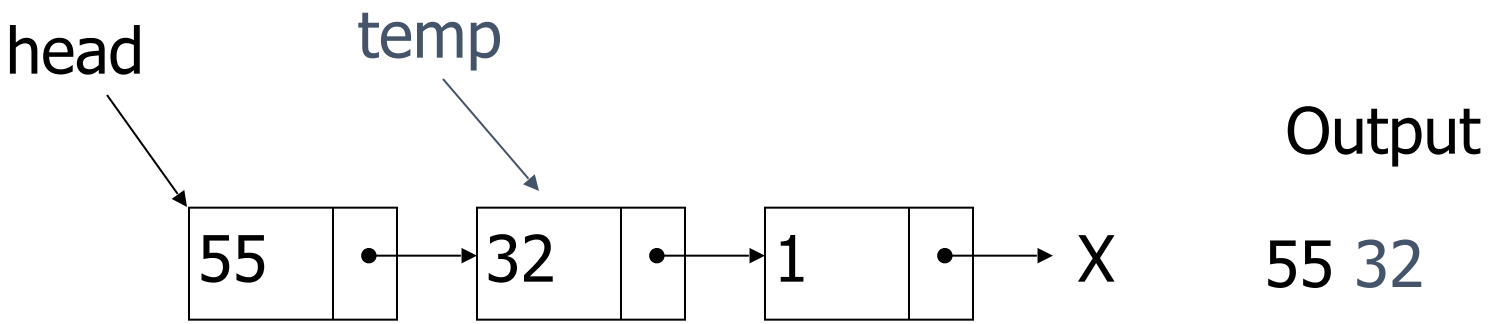
Output

55

Linked List Code - Print

- Printing the list

```
Node* temp;
for ( temp = head; temp != NULL; temp = temp->next )
        cout << temp->item << " ";
```



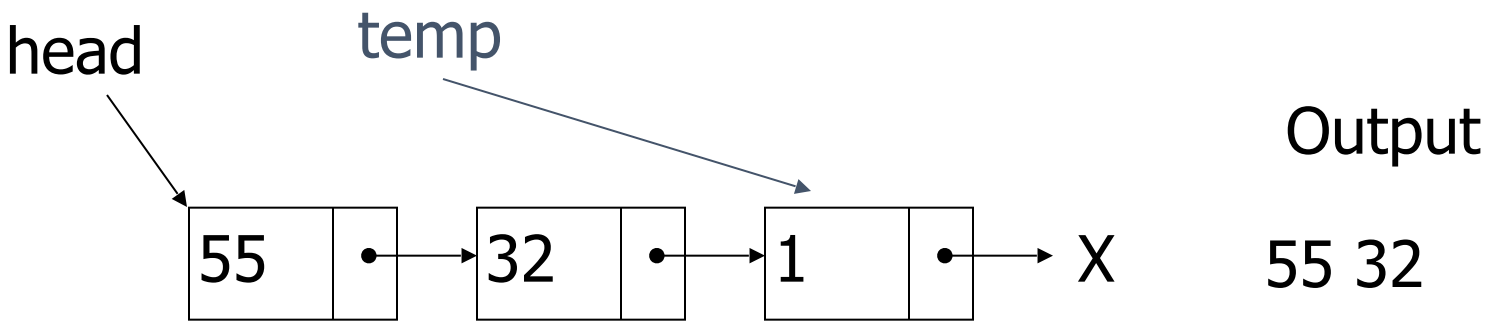
Output

55 32

Linked List Code - Print

- Printing the list

```
Node* temp;
for ( temp = head; temp != NULL; temp = temp->next )
        cout << temp->item << " ";
```



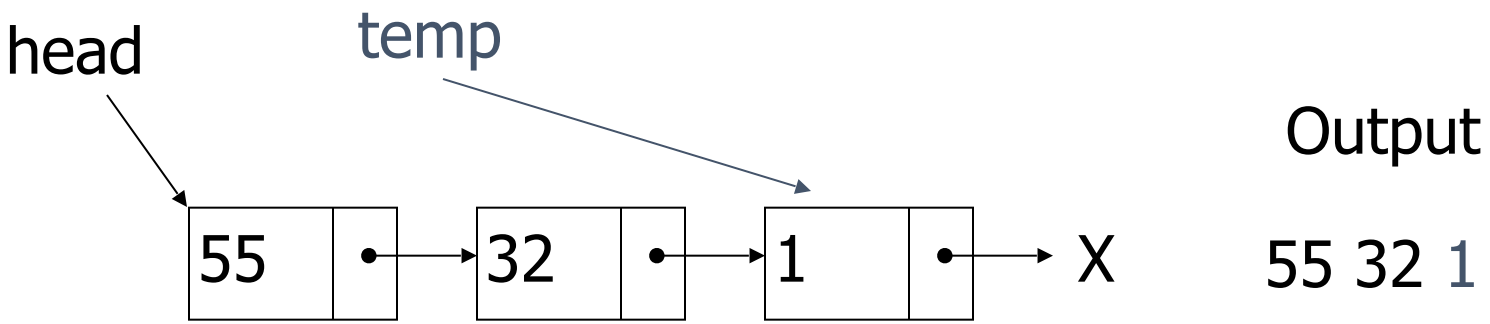
Output

55 32

Linked List Code - Print

- Printing the list

```
Node* temp;
for ( temp = head; temp != NULL; temp = temp->next )
        cout << temp->item << " ";
```



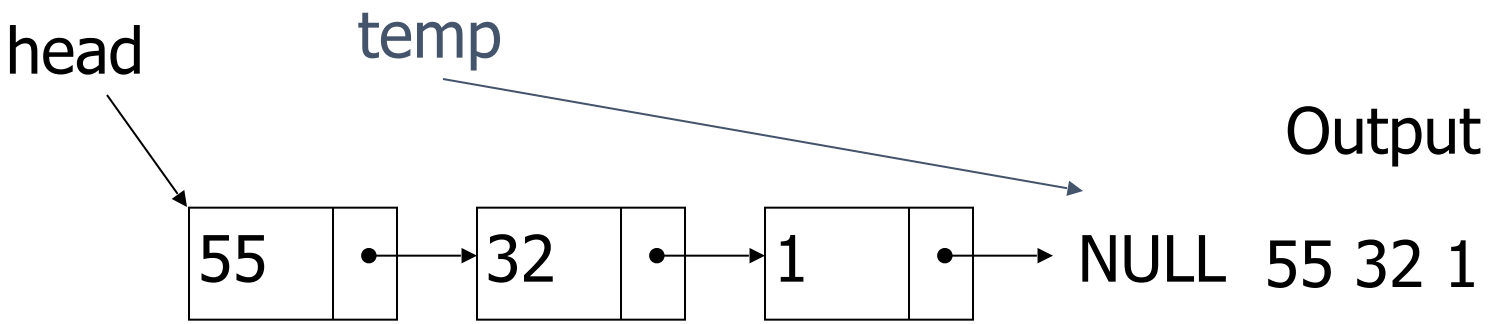
Output

55 32 1

Linked List Code - Print

- Printing the list

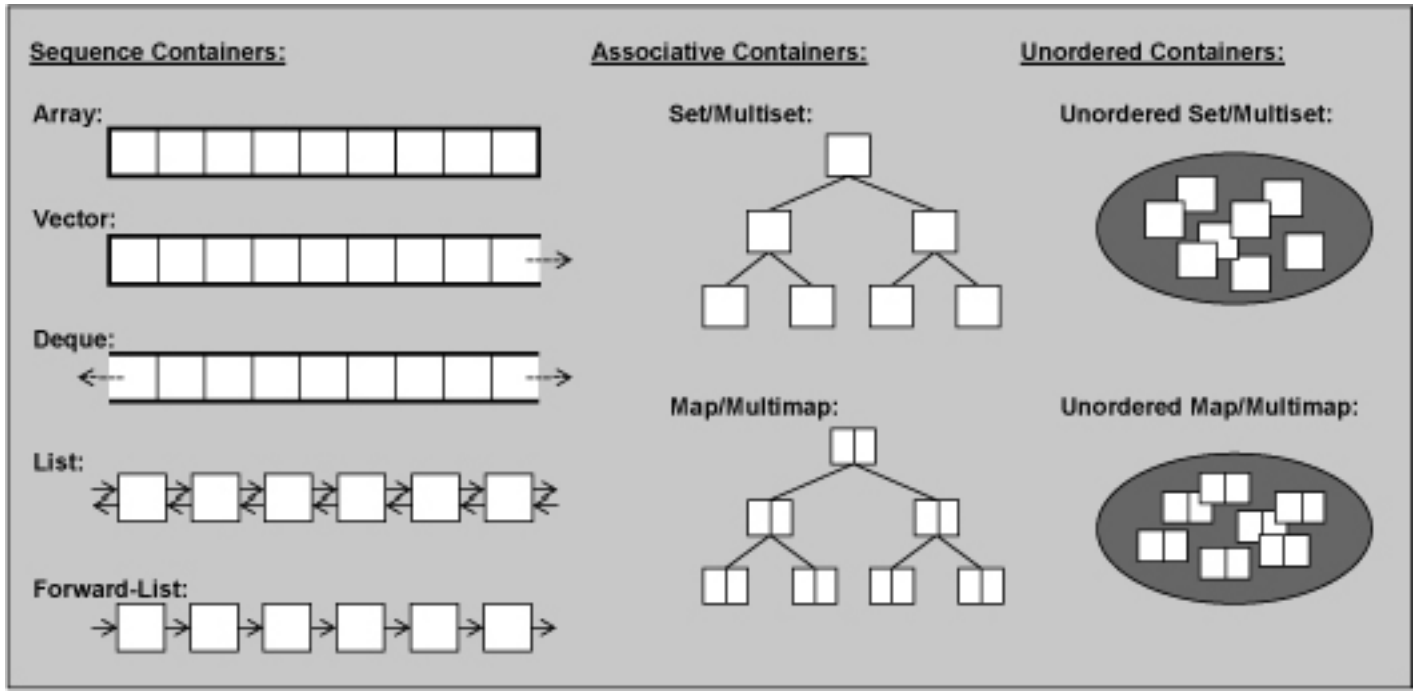
```
Node* temp;
for ( temp = head; temp != NULL; temp = temp->next )
        cout << temp->item << " ";
```



Output

55 32 1

STL Containers



"Container" Classes manage a collection of elements. - N.M Josuttis

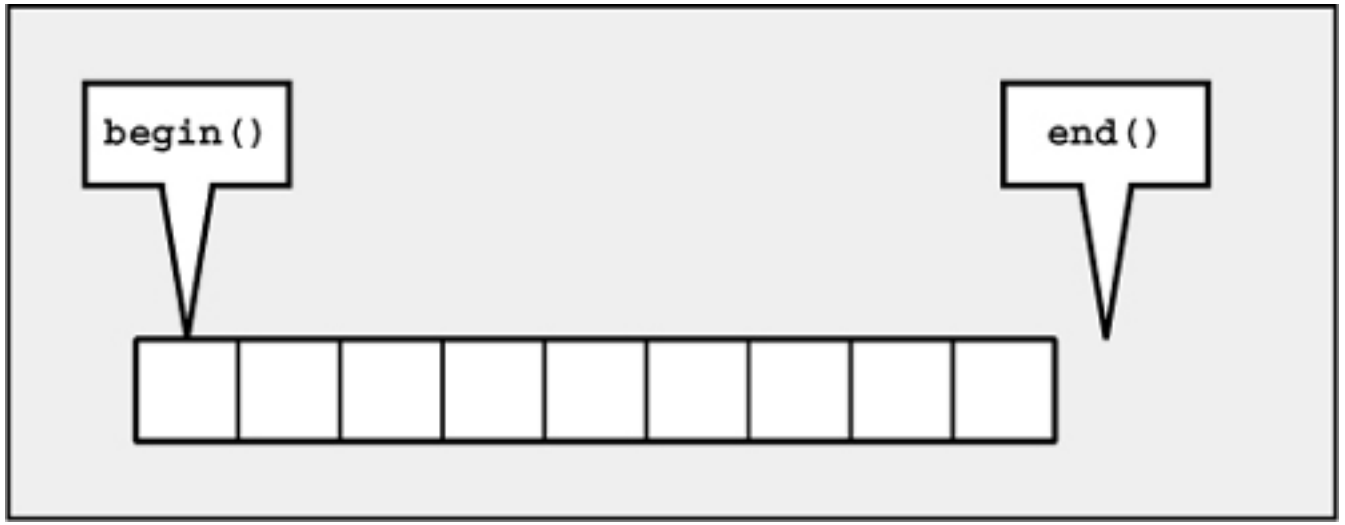
Iterators

- Some operations on lists (container), such as insert and remove require a notion of position.
- http://www.cplusplus.com/reference/stl/
- **STL** – **S**tandard **T**emplate **L**ibrary provides an **iterator** class for representing position in containers.
- **Examples…**
  - `list<string>::iterator it;`
  - `map<char,int>::iterator it;`

Iterators

- `iterator begin()` - returns an appropriate iterator that represents the beginning of elements in the container.
- `iterator end()` - returns the appropriate iterator that represents the end of the elements in container, i.e., position past the last item(valid position).
- `itr++` - advances the iterator `itr` to the next position in container. (`itr--`)
- `*itr` - returns the element of the current position of `itr`. If element has members, use -> to access members.
- `itr1 == itr2` - returns **`TRUE`** if `itr1` and `itr2` refer to same position.
- `itr1 != itr2` - returns **`TRUE`** if `itr1` and `itr2` refer to different positions.

Iterators

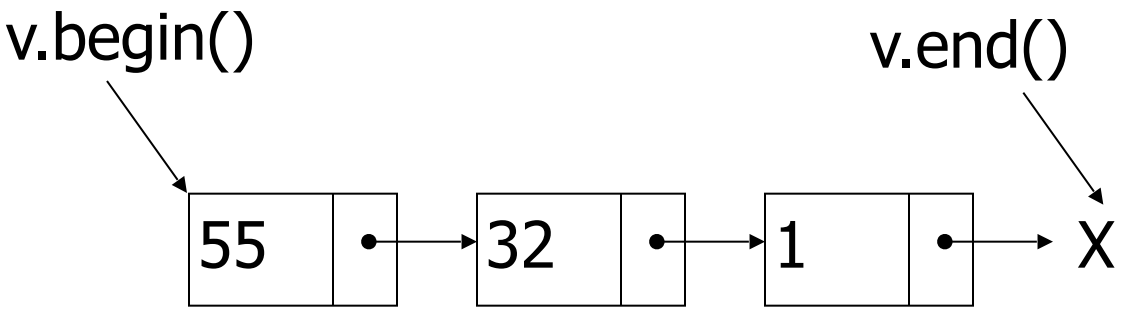


The Standard Template Library - A Tutorial and Reference - N.M Josuttis

Linked List Code – Print *

Printing the list – using an iterator

```
list<int>::iterator itr;
for ( itr = v.begin(); itr != v.end(); ++itr) {
        cout << *itr << " ";
```



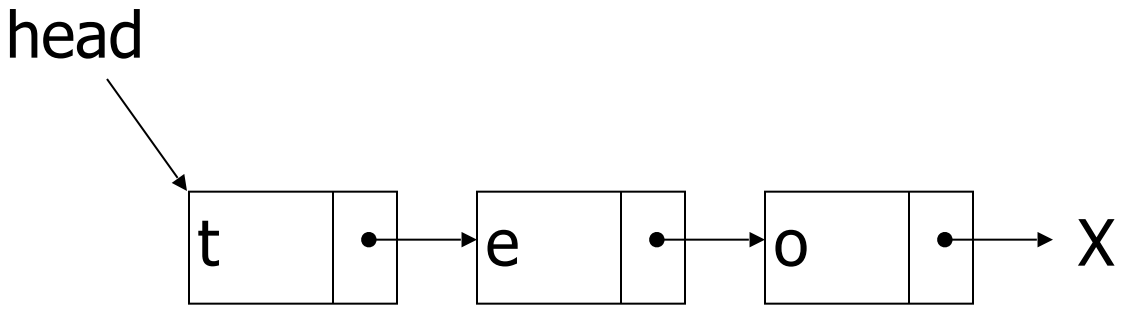
Linked List Code - Get Last

- (Stop here) Quick exercise - write the code to get a pointer to the last node in a list

Linked List Code - Insert

- Inserting at the head/front

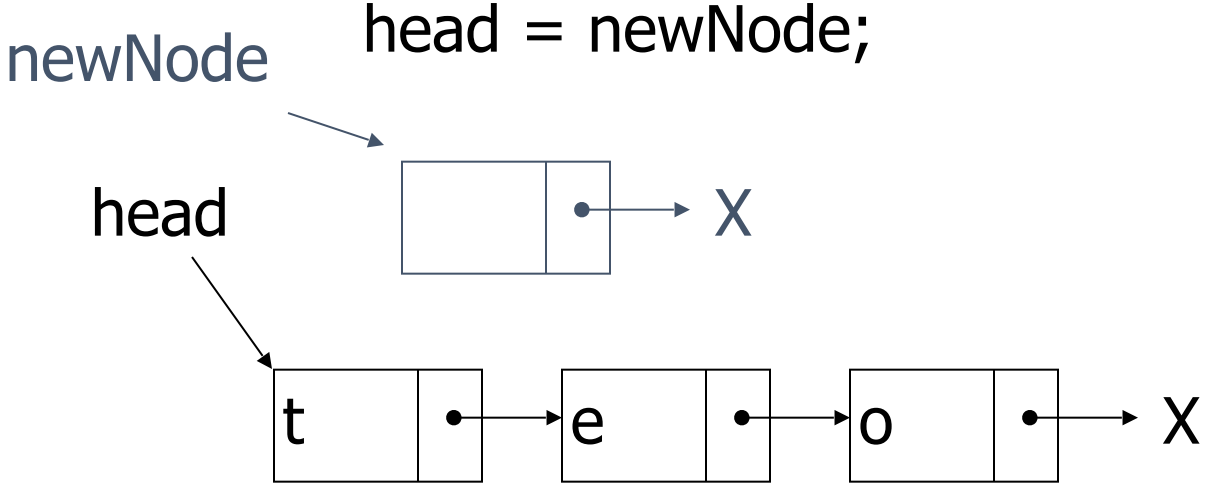
```
Node* newNode = new Node;
newNode->item = b;
newNode->next = head;
head = newNode;
```



Linked List Code - Insert

- Inserting at the head/front

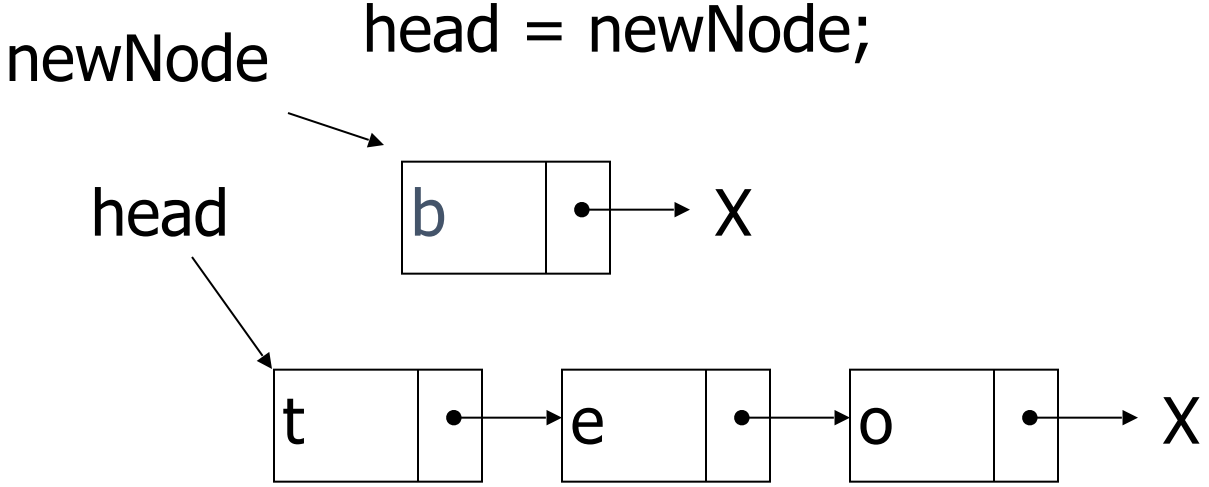
```
Node* newNode = new Node;
newNode->item = b;
newNode->next = head;
head = newNode;
```



Linked List Code - Insert

- Inserting at the head/front

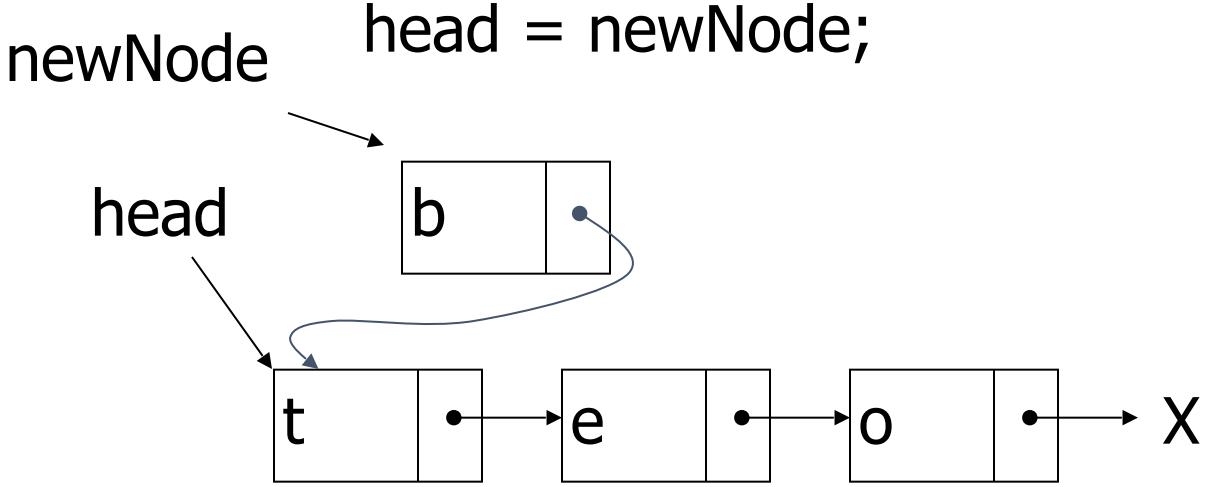
```
Node* newNode = new Node;
newNode->item = b;
newNode->next = head;
head = newNode;
```



Linked List Code - Insert

- Inserting at the head/front

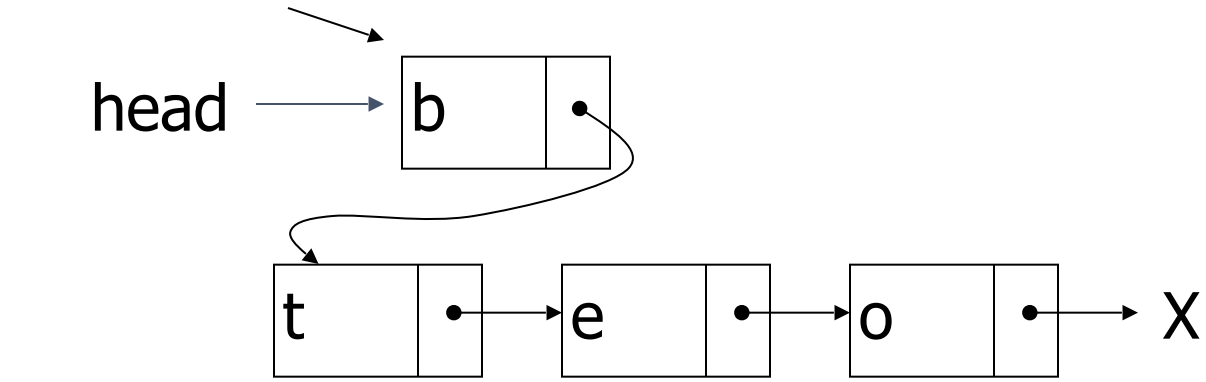
```
Node* newNode = new Node;
newNode->item = b;
newNode->next = head;
head = newNode;
```



Linked List Code - Insert

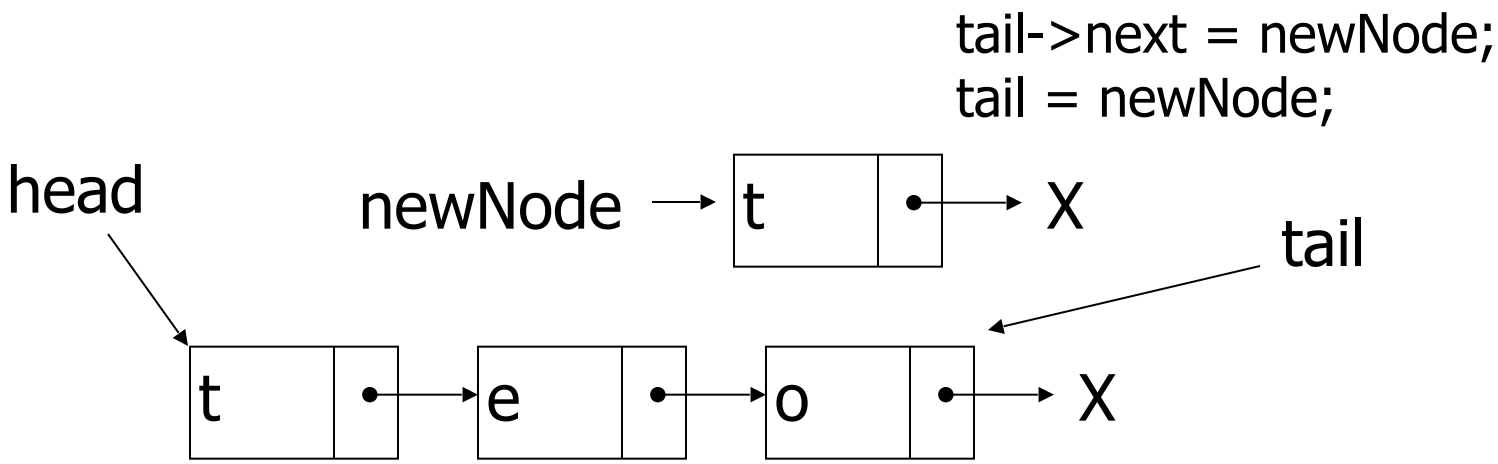
- Inserting at the head/front

```
Node* newNode = new Node;
newNode->item = b;
newNode->next = head;
head = newNode;
```



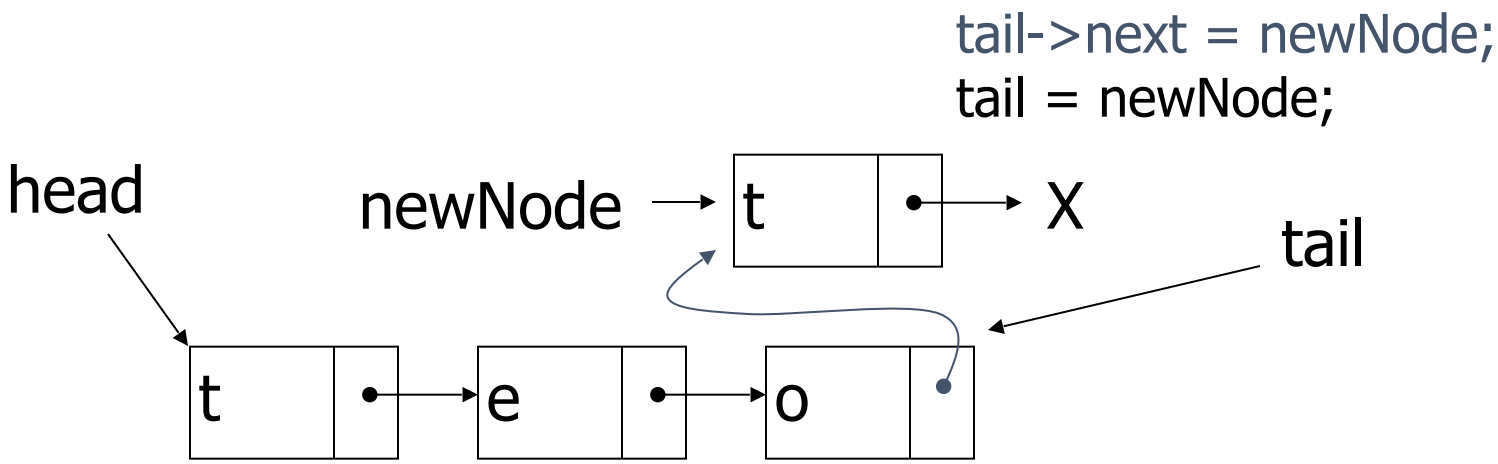
Linked List Code - Insert

- Inserting at tail/end
  - For now, we assume that we have a pointer to the last node called tail.
  - Assume newNode is created and has its value set.



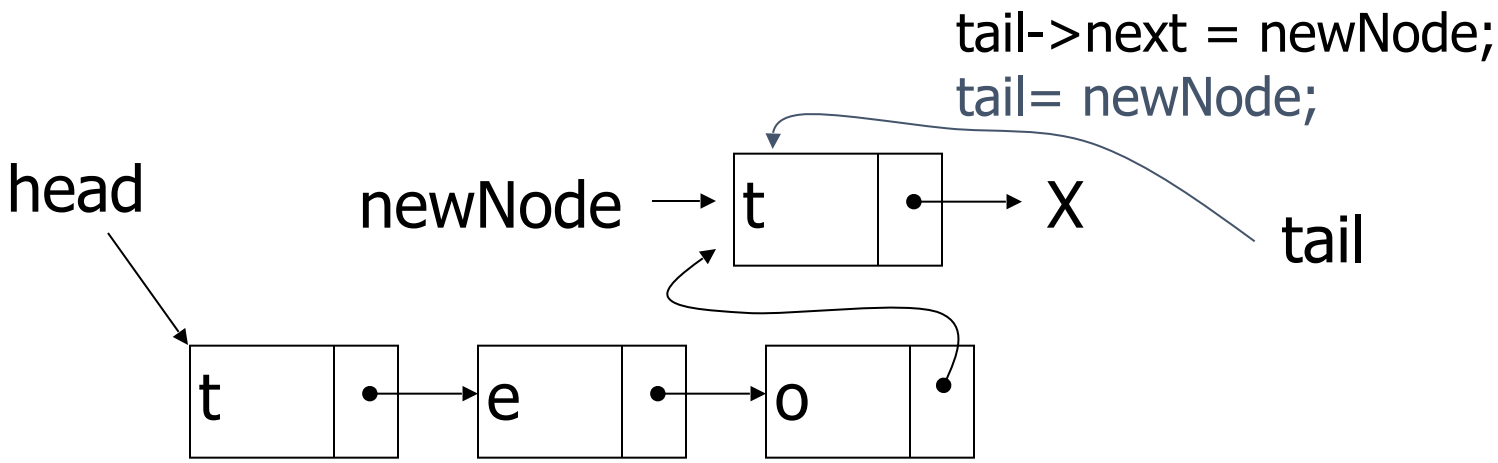
Linked List Code - Insert

- Inserting at tail/end
  - For now, we assume that we have a pointer to the last node called tail.
  - Assume newNode is created and has its value set.



Linked List Code - Insert

- Inserting at tail/end
  - For now, we assume that we have a pointer to the last node called tail.
  - Assume newNode is created and has its value set.



Linked List Code - Search

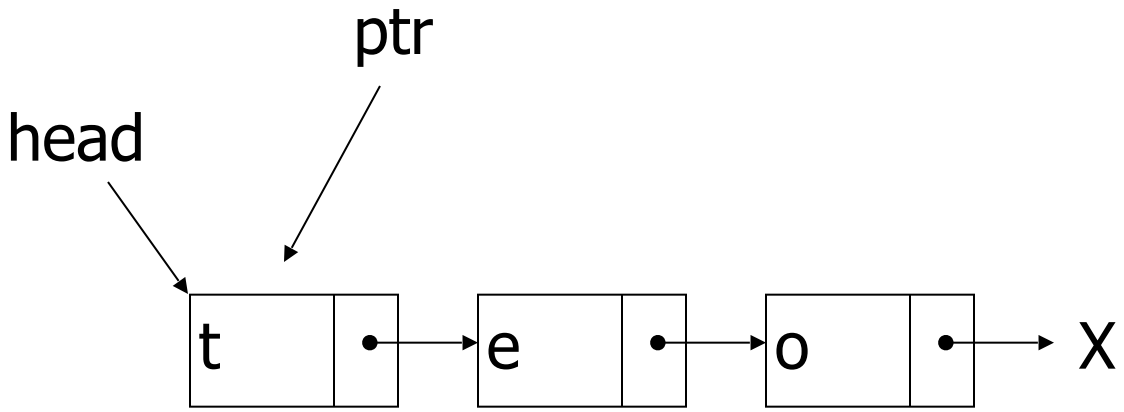
- Search - returns a pointer to the node containing the key or NULL if the key doesn't exit

```
Node* search ( itemtype key ) {
        Node* temp;
        for ( temp = head; temp != NULL; temp = temp->next ) {
                if ( temp->item == key )
                        return temp;
        }
        return NULL;
}
```

Linked List Code - Remove

- Removing a node - assume *ptr* points to the node we want to remove
- Remove first node:

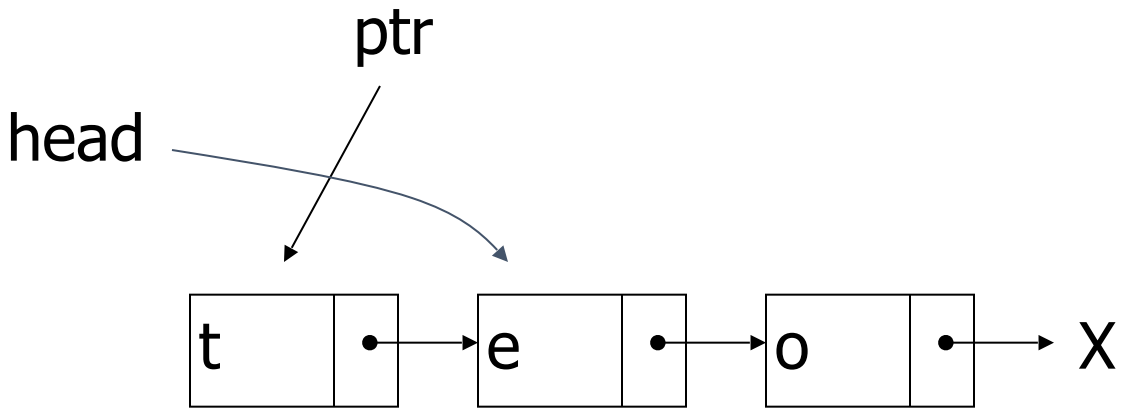
```
head = head->next;
delete ptr;
```



Linked List Code - Remove

- Removing a node - assume *ptr* points to the node we want to remove
- Remove first node:

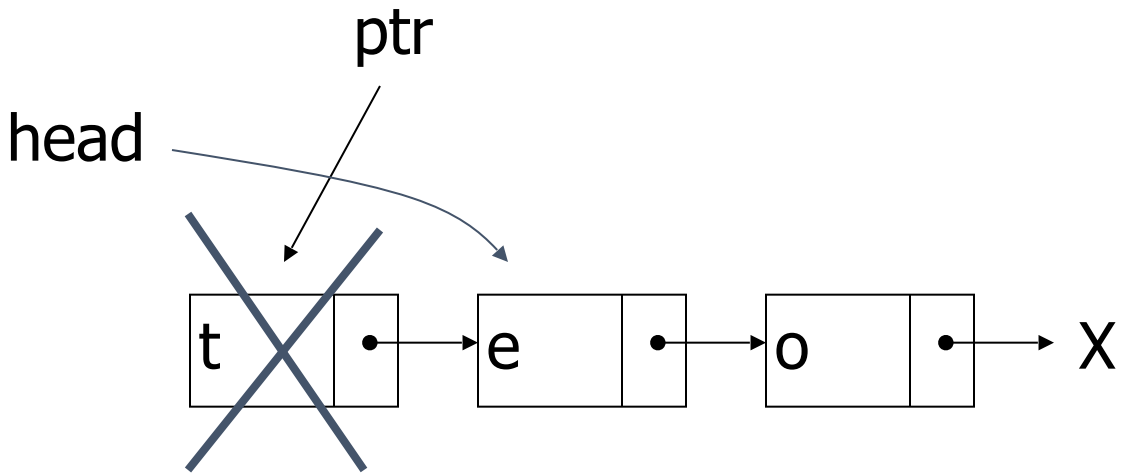
```
head = head->next;
delete ptr;
```



Linked List Code - Remove

- Removing a node - assume *ptr* points to the node we want to remove
- Remove first node:

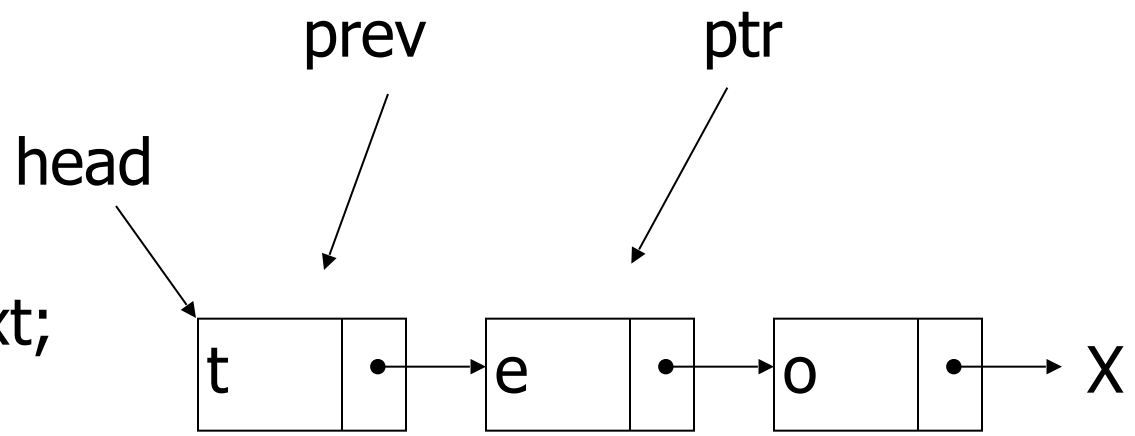
```
head = head->next;
delete ptr;
```



Linked List Code - Remove

- Removing a node - assume *ptr* points to the node we want to remove
- Remove middle node - assume *prev* pointer:

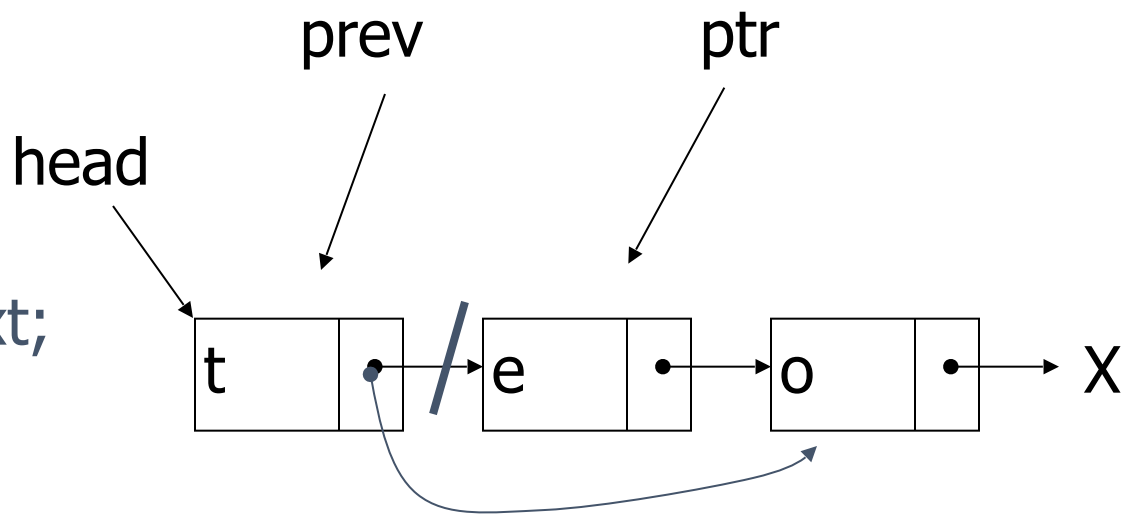
```
prev->next = ptr->next;
delete ptr;
```



Linked List Code - Remove

- Removing a node - assume *ptr* points to the node we want to remove
- Remove middle node - assume *prev* pointer:

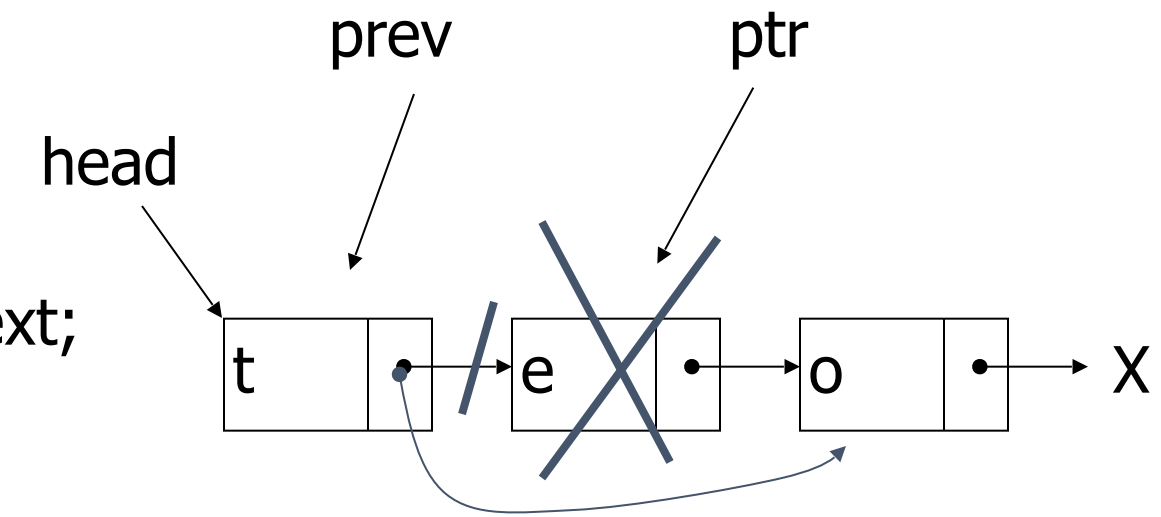
```
prev->next = ptr->next;
delete ptr;
```



Linked List Code - Remove

- Removing a node - assume *ptr* points to the node we want to remove
- Remove middle node - assume *prev* pointer:

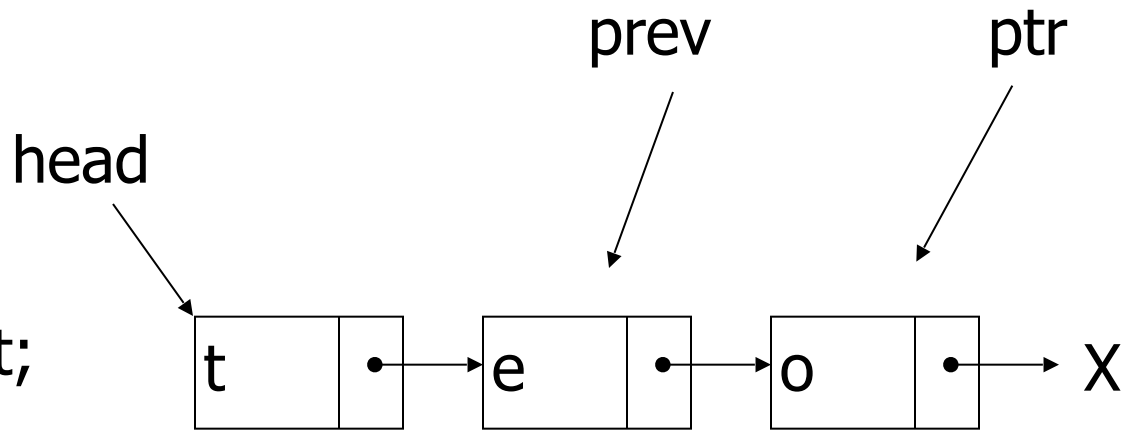
```
prev->next = ptr->next;
delete ptr;
```



Linked List Code - Remove

- Removing a node - assume *ptr* points to the node we want to remove
- Remove last node - assume *prev* pointer

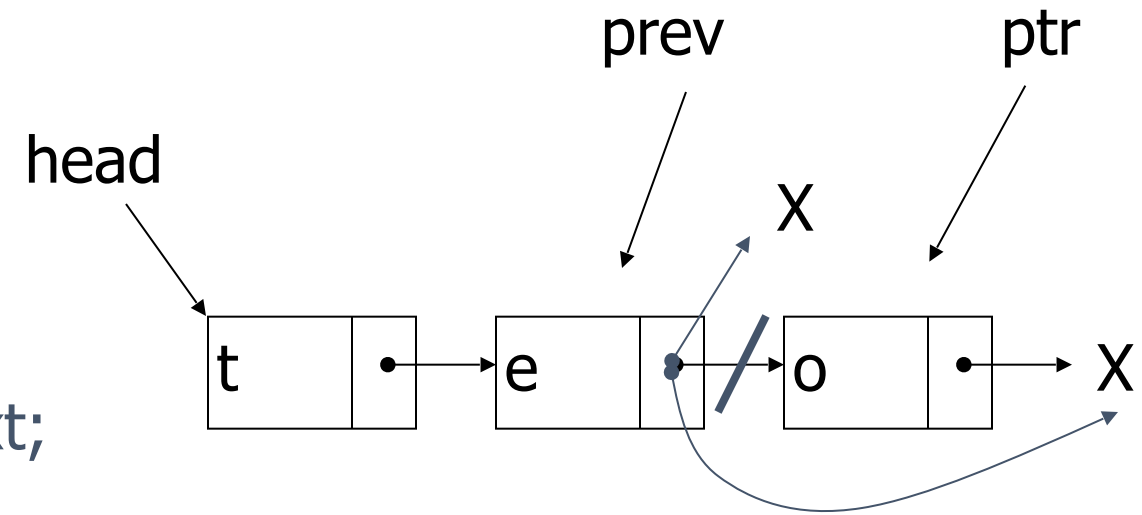
```
prev->next = NULL;
        OR
prev->next = ptr->next;

delete ptr;
```



Linked List Code - Remove

- Removing a node - assume *ptr* points to the node we want to remove
- Remove last node - assume *prev* pointer

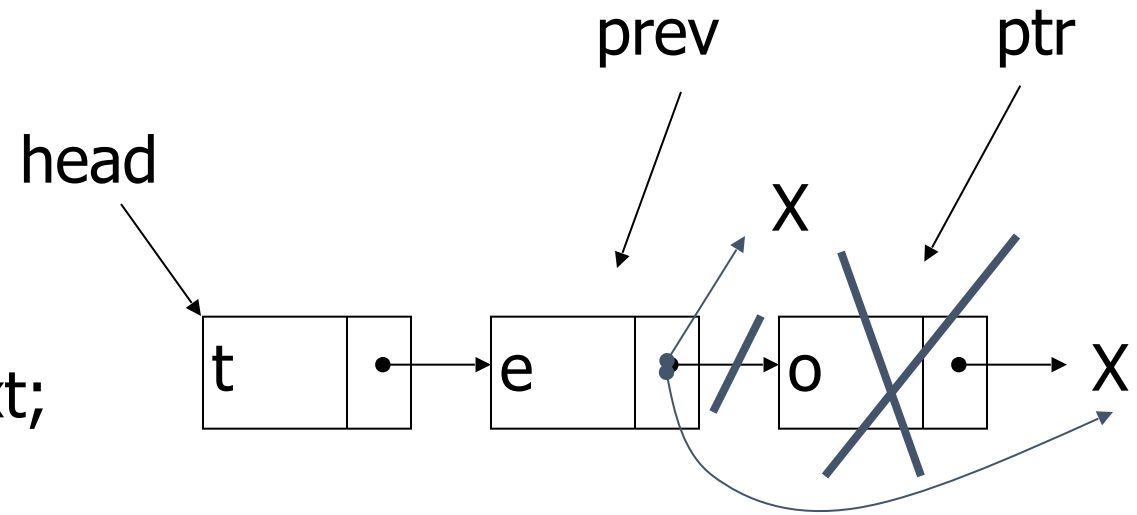
```
prev->next = NULL;
        OR
prev->next = ptr->next;
delete ptr;
```



Linked List Code - Remove

- Removing a node - assume *ptr* points to the node we want to remove
- Remove last node - assume *prev* pointer:

```
prev->next = NULL;
        OR
prev->next = ptr->next;
delete ptr;
```



Linked List Code - Destructor

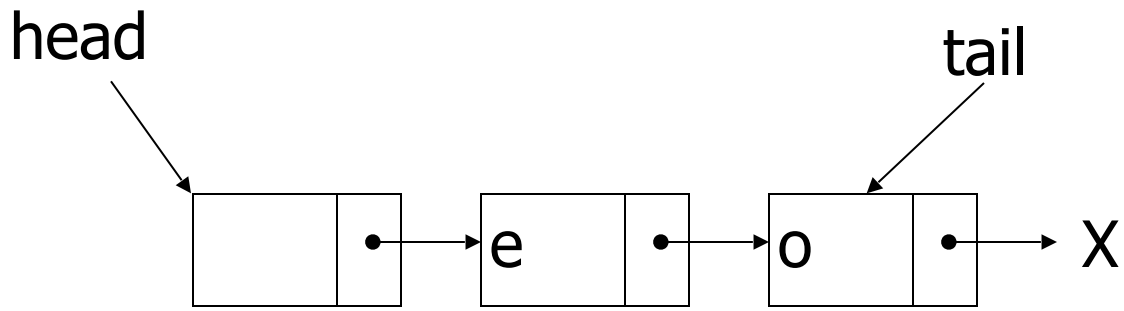
- Every **`new`** needs to have a corresponding **`delete`**.

```
int main ( ) {
        List* myList = new List;
        …
        delete myList;
}
```

```
List::~List ( ) {
        Node* temp;
        while ( head )
                temp=head;
                head=head->next;
                delete temp;
}
```

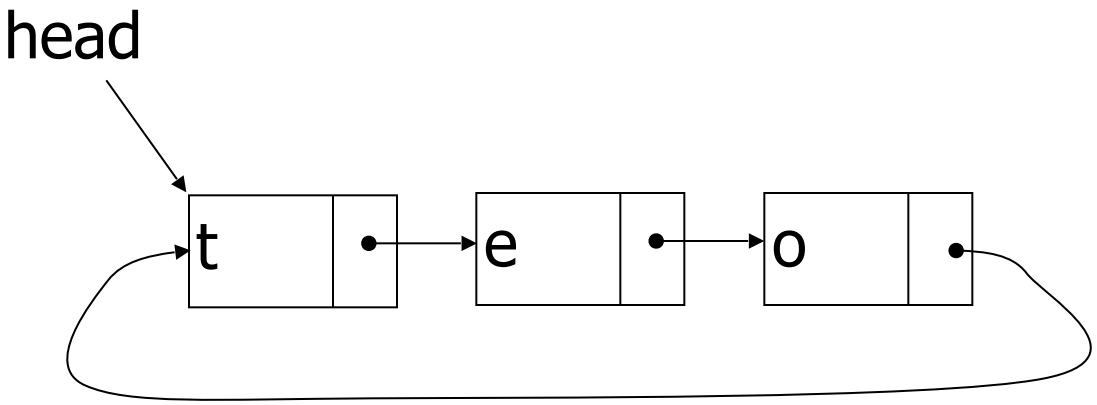
Linked List Variations

- Keep a head and tail pointer to make tail inserts faster
  - More pointer upkeep on inserts and removes
- Use a sentinel/dummy node
  - Easier removes because there is only one case



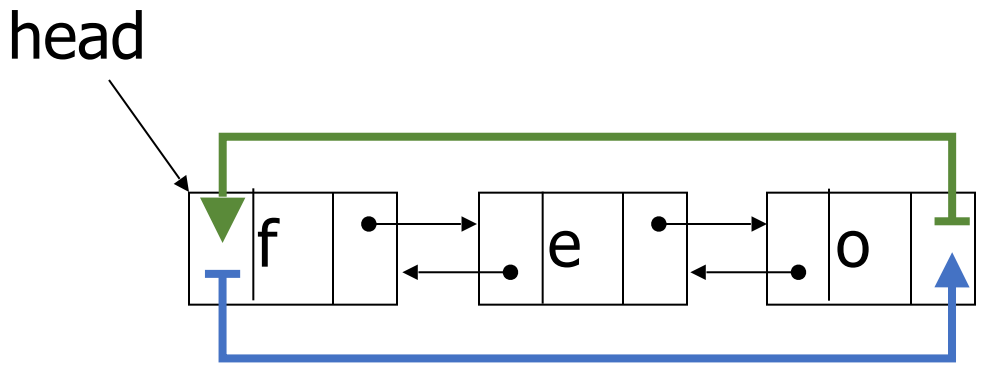
Linked List Variations

- Circularly linked list
  - No special cases for head or tail



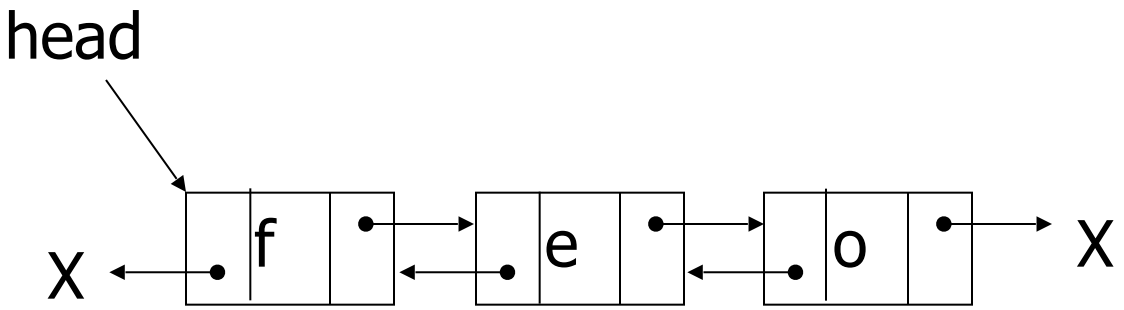
Linked List Variations

- Doubly-linked list
- Still have to find previous node for inserts and removes
- Solved with a doubly linked list



Linked List Variations

- Doubly-linked list
- Still have to find previous node for inserts and removes
- Solved with a doubly linked list



Doubly Linked Lists

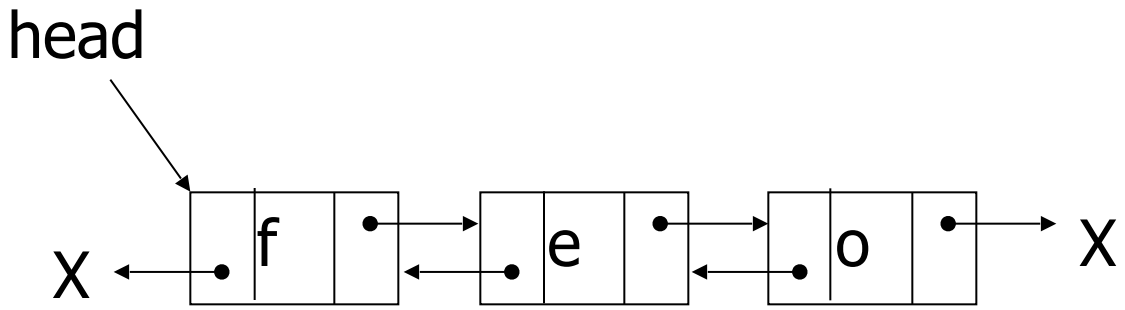
- Required adjustments for doubly linked list:
  - Size increase per node
    - extra *prev* pointer kept in each node
  - Slight increase in time to do inserts and removes
    - more pointer manipulations
  - Increases pointer complexity
    - If you don’t like pointers

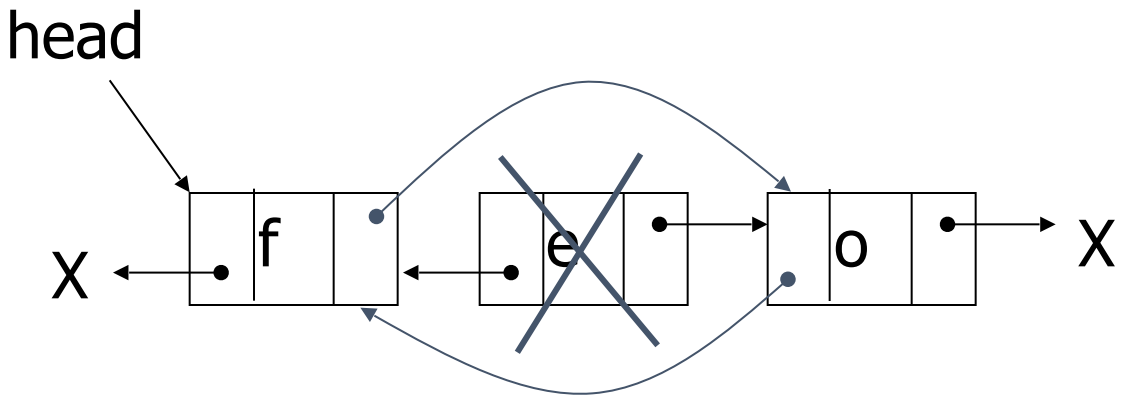
Doubly Linked List Implementation

```
class List {
private:
        Node* head;
public:
        List ( ) {
                head = NULL;
        };
        // Member functions to
        // interface with list
};
```

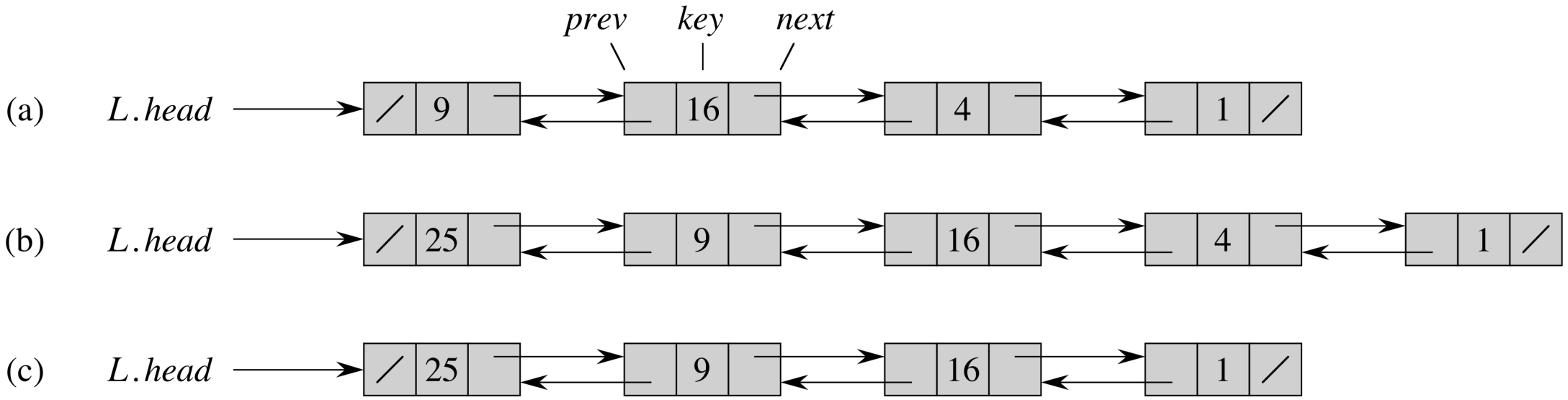
```
class Node {
friend class List;
private:
        itemtype Item;
        Node* next;
        Node* prev;
};
```

Doubly linked list - Remove





Doubly Linked List



Doubly Linked List Code - Remove

- In class exercise - write remove for a doubly linked list
- Assumptions:
  - Only have key and pointer to head
  - Don’t assume item exists
  - Remove first instance

Doubly Linked List Code - Remove

Linked List Programming Notes

- Memory leaks
  - Every insert invokes a `new`, be sure to have a complimentary `delete` for each new. Since every node will not necessarily be removed during the program, the destructor should go through and delete all the nodes still left.
- Segmentation faults
  - Be sure to compare all pointers to NULL before accessing any information

Linked List Programming Notes

- Deal with all possible cases
  - Check for empty list
  - Deal with first, middle, and last node cases if they require different code
- Most importantly….be generic.
  - Don't take type of item to store into consideration
  - Plan to reuse code

Linked List Running Times

- Assuming only a head pointer and a singly linked list
  - Insert head
    - O(1)
  - Insert tail
    - O(N)
  - Search
    - O(N)
  - Remove
  - Num items
  - Print