Lists

Lists

- Lists
  - Grocery list
  - List of assignments to complete
  - List of phone calls to make/return
  - etc

Lists Implemented with Arrays

- A list can be implemented with an array
  - Fixed size determined at compile time - space allocated at compile time

```
int list[MAX_ARRAY_SIZE];
```

  - Fixed size determined at compile time - space allocated at runtime
    - Dynamically allocated array

```
int* list;
list = new int[MAX_ARRAY_SIZE];
```

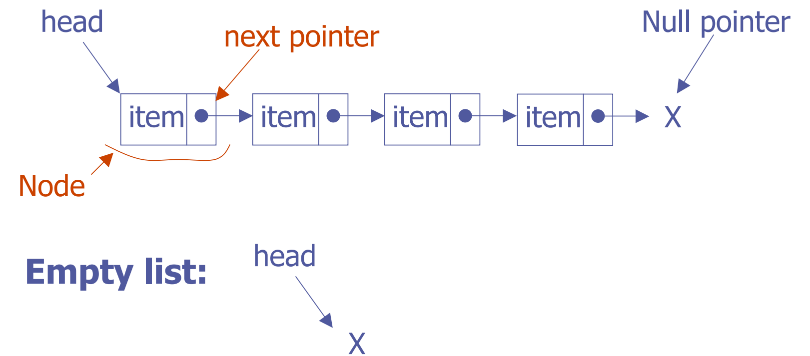

Lists Implemented with Arrays

- List implemented with an array may not be flexible
  - – Wasted space or not enough space
  - – Difficult to insert in the middle
    - Shift items
  - + Direct access

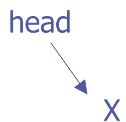
Linked List

- A linked list is a collection of elements - called nodes. Each node has a special pointer that connects it to the next node in the list
  - Size is not fixed - grows and shrinks at runtime

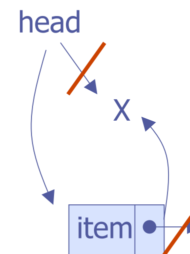
Linked List



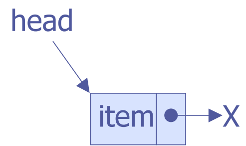
Linked list - Inserting Nodes



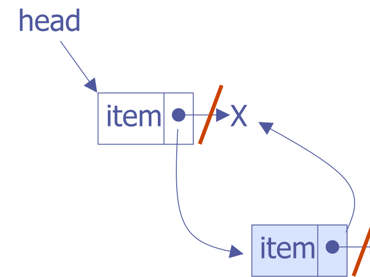
Linked list - Inserting Nodes



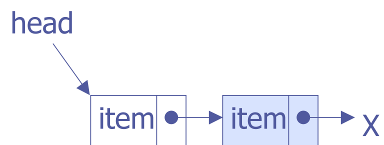
Linked list - Inserting Nodes



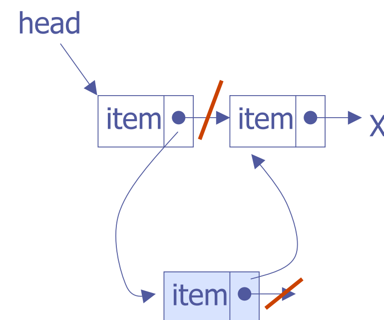
Linked list - Inserting Nodes



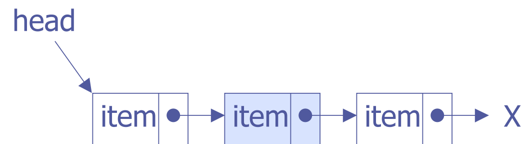
Linked list - Inserting Nodes



Linked list - Inserting Nodes



Linked list - Inserting Nodes



Linked list - Deleting Nodes

- In class exercise - show pointer changes when deleting a node

Linked list - Deleting Nodes

- In class exercise - show pointer changes when deleting a node

Linked list - Deleting Nodes

- In class exercise - show pointer changes when deleting a node

Linked List Implementation

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

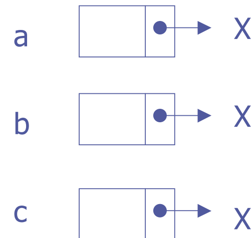
```
class Node {
public:
        itemtype Item;
        Node* next;
};
```


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
```

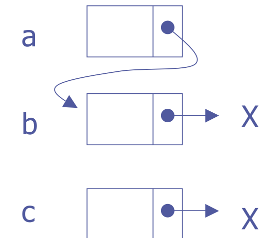

Link List Manipulation

```
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
```



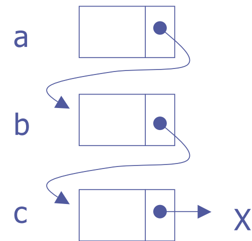
Link List Manipulation

```
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
```



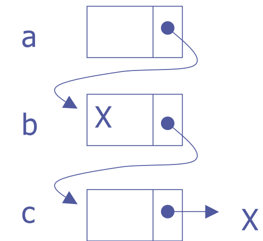
Link List Manipulation

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



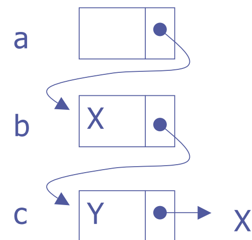
Link List Manipulation

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



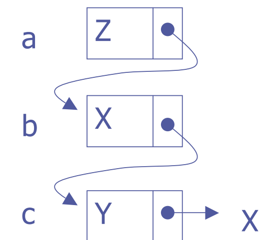
Link List Manipulation

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

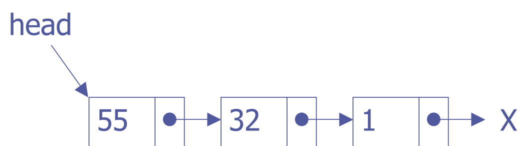
Node a = new Node;*
Node b = new Node;*
Node c = new Node;*
a->next = b;
b->next = c;
b->item = X;
c->item = Y;
a->item = Z



Linked List Code - Print

◆Printing the list

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

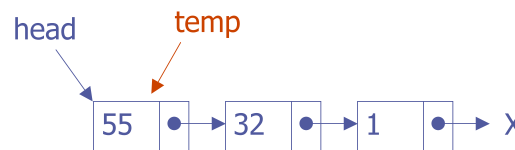


Output

Linked List Code - Print

◆Printing the list

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

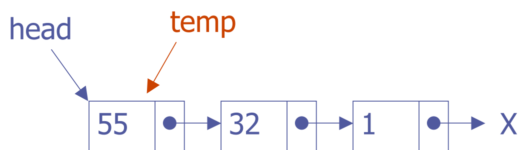


Output

Linked List Code - Print

◆Printing the list

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



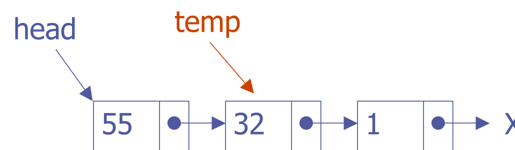
Output

55

Linked List Code - Print

◆Printing the list

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



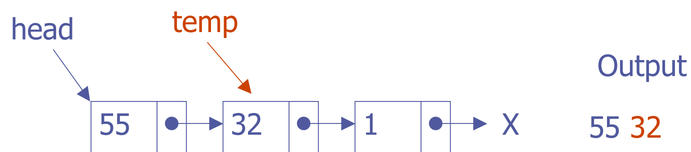
Output

55

Linked List Code - Print

◆ Printing the list

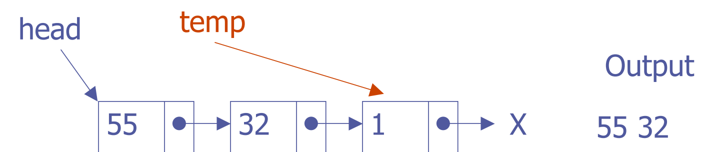
```
Node* temp;
for ( temp = head; temp != NULL; temp = temp->next ) {
        cout << temp->item << " ";
```



Linked List Code - Print

◆ Printing the list

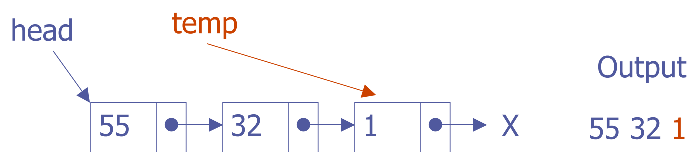
```
Node* temp;
for ( temp = head; temp != NULL; temp = temp->next ) {
        cout << temp->item << " ";
```



Linked List Code - Print

◆ Printing the list

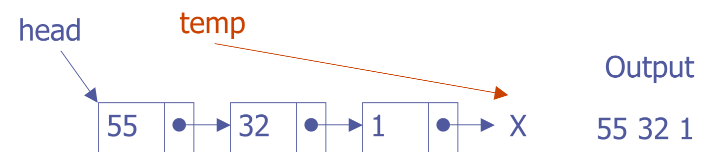
```
Node* temp;
for ( temp = head; temp != NULL; temp = temp->next ) {
        cout << temp->item << " ";
```



Linked List Code - Print

◆ Printing the list

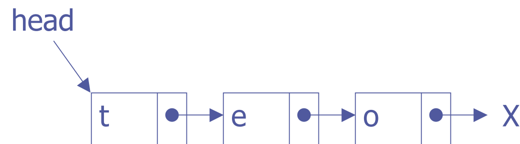
```
Node* temp;
for ( temp = head; temp != NULL; temp = temp->next ) {
        cout << temp->item << " ";
```



Linked List Code - Insert

- Inserting at the head/front

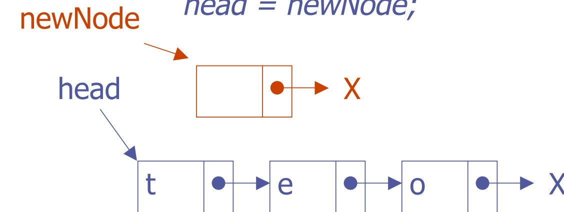
```
Node* newNode = new Node;
newNode->item = x;
newNode->next = head;
head = newNode;
```



Linked List Code - Insert

- Inserting at the head/front

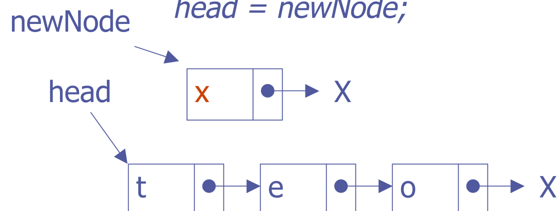
```
Node* newNode = new Node;
newNode->item = x;
newNode->next = head;
head = newNode;
```



Linked List Code - Inser

- Inserting at the head/front

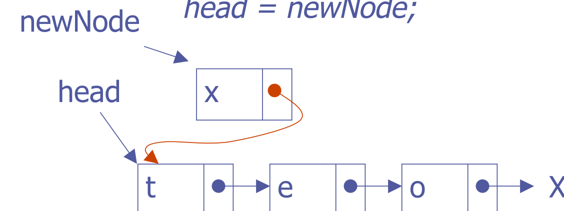
```
Node* newNode = new Node;
newNode->item = x;
newNode->next = head;
head = newNode;
```



Linked List Code - Insert

- Inserting at the head/front

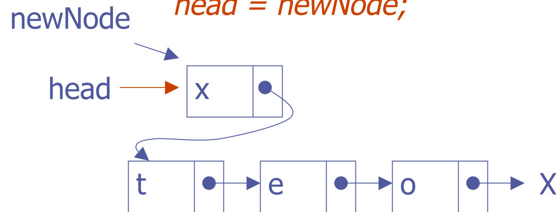
```
Node* newNode = new Node;
newNode->item = x;
newNode->next = head;
head = newNode;
```



Linked List Code - Insert

- Inserting at the head/front

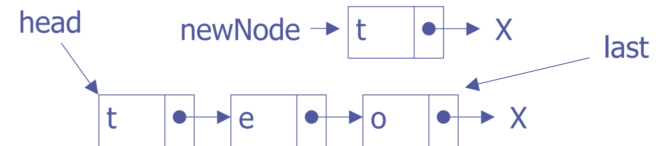
Node newNode = new Node;*
newNode->item = x;
newNode->next = head;
head = newNode;



Linked List Code - Insert

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

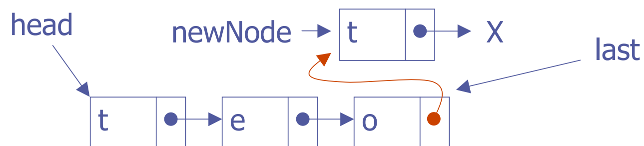
last->next = newNode;
last = newNode;



Linked List Code - Insert

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

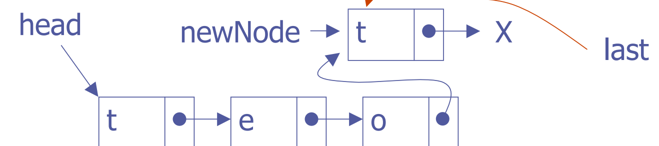
last->next = newNode;
last = newNode;



Linked List Code - Insert

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

last->next = newNode;
last = newNode;



Linked List Code - Get Last

- In class exercise - write the code to get a pointer to the last node in a list

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;
}
```

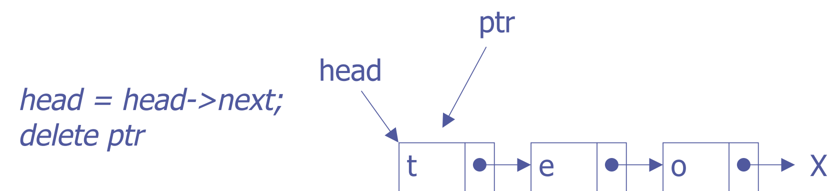

Linked List Code - Search

- In class exercise - write search function recursively

Linked List Code - Remove

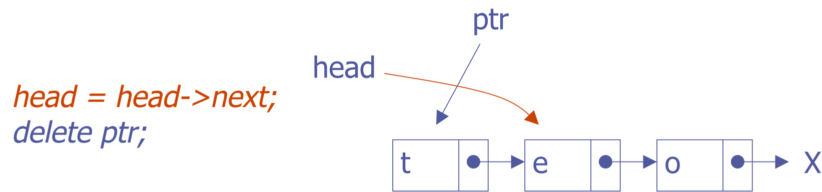
- Removing a node - assume *ptr* points to the node we want to remove
- Remove first node:

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



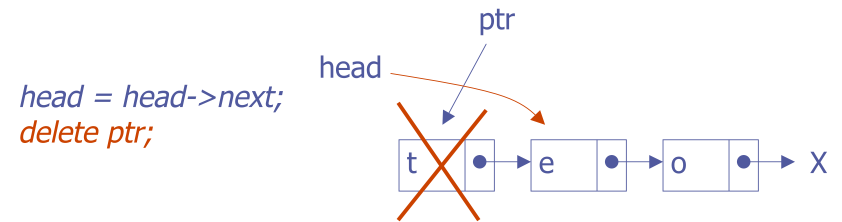
Linked List Code - Remove

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



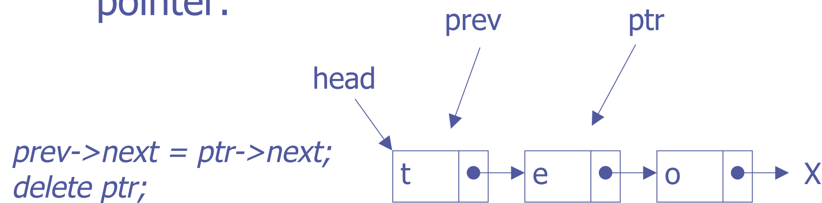
Linked List Code - Remove

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



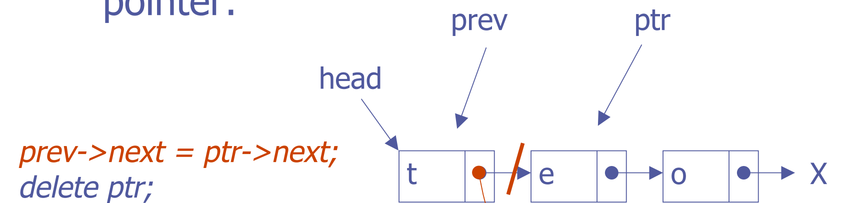
Linked List Code - Remove

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



Linked List Code - Remove

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



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;

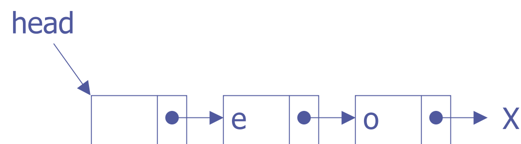
Linked List Code - Remove

- In class exercise - write the remove function.
  - Remember, removing from head and middle/end are separate cases
- Assumptions:
  - Only have key and pointer to head
  - Don't assume item exists
  - Remove first instance

Linked List Code - Remove

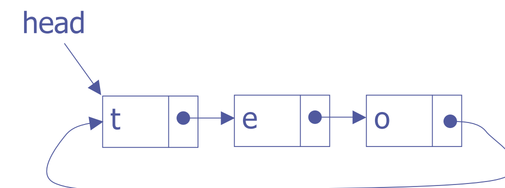
Linked List Variations

- 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



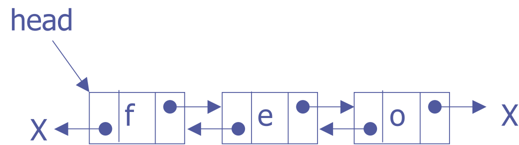
Linked List Variations

- Circularly linked list



Linked List Variations

- 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 {
public:
        itemtype Item;
        Node* next;
        Node* prev;
};
```


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
  - Insert tail
  - Search
  - Remove
  - Num items
  - Print